第5章 存储管理

现代计算机系统中的存储器通常由内存和外存组成。存储管理主要涉及的是内存的管理，而外存则被看成是内存的直接延伸，故存储管理也称作内存管理。

内部存储器（简称为内存，或称为主存）的管理是操作系统的主要功能之一。近年来，内存容量虽然一直在不断扩大，但是程序增长的速度和内存容量的增长一样快，内存仍然是一种宝贵而又紧俏的资源。因此，对内存的管理和有效利用仍然是当今操作系统十分重要的内容。

内存一般被分为两大区域：系统区和用户区。系统区用于存放操作系统的内核程序和其他系统常驻程序，这些程序在系统初启时便装入系统区并一直驻留内存。对一个具体的计算机系统，系统区是固定的，一般占据内存的低地址部分。用户区用以存放用户程序和数据以及用户态下运行的系统程序，它是用户进程可共享的内存区。本章主要讨论用户区的管理。

各种操作系统之间最明显的区别之一，往往在于它们所采用的存储管理方案不同。目前，存储管理基本上可概括成4种方案：分区式存储管理、页式存储管理、段式存储管理、段页式存储管理。本章将逐一讨论各种方案的基本思想和实现技术。

5.1 存储管理基本概念

5.1.1 物理内存和虚拟存储空间

1. 物理内存

物理内存是由系统实际提供的硬件存储单元（通常为字节）组成的，CPU可直接访问这些存储单元，所有的程序指令和数据必须装入内存才能执行。

内存中所有的存储单元从0开始依次编号，这个编号称为这个存储单元的内存地址或物理地址。CPU通过物理地址找到相应的存储单元中存放的指令或数据。内存的地址空间是一维的，它的大小受到实际存储单元的限制，存储单元最大的内存地址加1称为内存空间大小或物理地址空间大小。内存地址编号00000000H～003FFFFFH（十六进制）构成了一个4MB大小的物理存储空间，如图5-1所示。

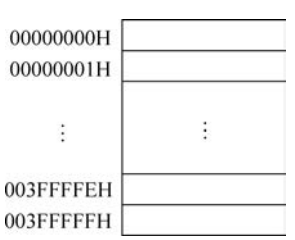


图5-1 4MB大小的物理存储空间

虽然内存的访问速度快，但其价格昂贵。因此，一个计算机系统中实际的内存容量往往有限，不可能存入大量的程序与数据，只能暂时存放将要访问的进程的程序段和数据。而系统中大量的程序和数据存入价格较便宜的外部存储器中，待需要访

问时，再将其调入内存。

2. 虚拟存储空间

将用户编写的源程序变为一个可在内存中执行的程序代码，通常要经过以下几个步骤。首先是编译，由编译程序将用户程序的源程序编译成若干目标模块；其次是链接，由链接程序将编译后的目标模块以及它们所需要的库函数链接在一起，形成一个完整的可装入模块，链接既可以是在程序执行以前由链接程序完成（称为静态链接），也可以是在程序执行过程中由于需要而进行的（称为动态链接）；最后由系统装入程序将装入模块装入内存。

那么，程序在运行中要访问的内存地址该怎样给出呢？通常有两种方法。一种是由程序员在程序中直接给出要访问的数据或指令的物理地址，但在程序员在给出直接的物理地址时，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，可能就要改变程序中所有的地址。因此，通常采用另一种方法，即用户用高级语言或汇编语言进行编程时，源程序中使用的都是符号地址，如 CALL Subpro1，GOTO A；等，用户不必关心符号地址在内存中的物理位置。源程序经过编译和链接后，形成一个以 0 地址为起始地址的线性或多维虚拟地址空间，每条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，我们把这个地址称为虚拟地址（virtual address）也称为相对地址或逻辑地址。程序运行时访问的内存物理地址由地址装入模块的地址变换机构完成。

通常将进程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟存储空间（virtual memory）。虚拟存储空间不考虑物理内存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。与实际物理内存只有一个（单机系统中）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空间，且虚拟存储空间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 的有效地址长度是 20 位，则其寻址范围为 $0 \sim 2^{20}-1$。采用虚拟存储空间技术使得用户程序空间和内存空间分离，从而使用户程序不受内存空间大小的限制，为用户提供一个比实际内存更大的虚拟存储空间。至于如何管理和实现虚拟存储空间，在第 5 章中将有介绍。

5.1.2 存储管理的主要任务

在多道的操作系统中，允许多个进程同时装入内存，通过进程的并发执行来提高 CPU 的利用率。于是如何将可用内存有效地分配给多个进程？如何让对存储容量的要求大于可用内存的大进程得以运行？如何保护和共享内存的信息？等等，是对存储管理提出的一系列要求。

现代操作系统中，存储管理既要方便用户，又要有利于提高内存的利用率。存储管理有如下 4 个主要任务。

1. 内存分配与回收

内存分配是多道程序共享内存的基础，它要解决的是如何为多个程序划分内存空间，使各个程序在指定的内存空间里运行。为此，操作系统必须随时掌握内存空间每个单元的使用情况（空闲还是占用）；当有存储申请时，根据需要选定分配区域，进行内存分配；如果占用者不再使用某个内存区域时，则应及时收回该区域。

内存分配有静态分配和动态分配两种方式。静态分配是在目标模块装入内存时一次性分配进程所需的内存空间，它不允许进程在运行过程中再申请内存空间；动态分配是在目

标模块装入内存时分配进程所需的基本内存空间,并允许进程在运行过程中申请附加的内存空间。

显然,动态存储分配具有较大的灵活性,它不需要一个进程在其全部信息进入内存后才开始运行,而是在进程运行期间需要某些信息时,系统才将其自动调入内存,进程当前暂不使用的信息可不进入内存,这对提高内存的利用率大有好处。

2. 地址变换

在多道程序环境下,要使程序运行,必须为它建立进程。建立进程时,首先必须为它分配必要的内存空间,再由装入程序将其装入内存。而在一般情况下,一个进程装入时分配到的内存空间和它的虚拟地址空间是不一致的。因此,当进程在运行时,所要访问的指令或数据的实际物理地址和虚拟地址是不同的。显然,如果在进程装入或执行时不对有关的地址部分加以相应的修改,将导致错误的结果。为了保证程序的正确执行,需将程序的虚拟地址转换为物理地址,将由虚拟地址到物理地址的变换过程称为地址重定位或地址变换。虚拟地址空间的程序和数据经过地址重定位后,就变成可由CPU直接执行的绝对地址程序,其变换过程与存储空间如图5-2所示。

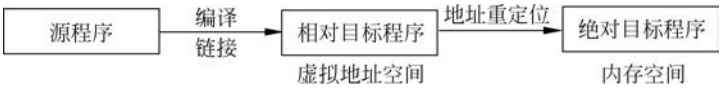


图5-2 地址变换过程与存储空间

地址重定位可分为静态重定位和动态重定位两种方式。

(1) 静态地址重定位是在目标程序装入指定的内存区域时由装配程序完成地址的变换。例如,图5-3中一个以0为起始地址的目标程序要装入以100为起始地址的存储空间。显然,在装入之前要做某些修改,程序才能正确执行。例如,"LOAD 1,1000"这条指令的意义是把虚拟地址为1000的存储单元内容222装入1号寄存器。现在内容为222的存储单元的实际地址为1100,即为虚拟地址(1000)加上装入的起始地址(100)。因此,"LOAD 1,1000"这条指令装入内存后,其中的直接地址也要做相应的修改,而成为"LOAD 1,1100"。程序在装入内存时,程序中涉及地址的每条指令都要进行这样的修改,程序才正确执行。

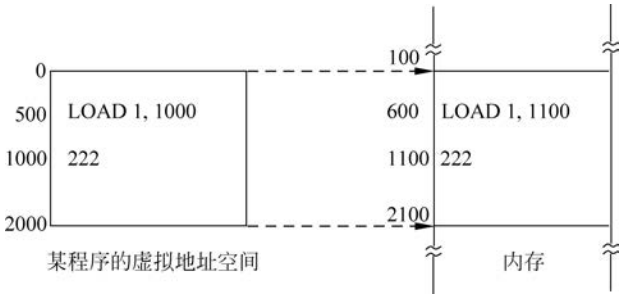


图5-3 静态地址重定位示意图

静态地址重定位的优点是不需要硬件支持。但存在如下缺点:一是程序装入内存后不能在内存中移动;二是程序必须装入连续的地址空间内,不利于程序的共享,也不能充分利用内存空间。

(2) 动态地址重定位是指程序在执行的过程中,在CPU访问内存地址之前,由地址变换机构(硬件)来完成要访问的指令或数据的虚拟地址到物理地址的转换。地址变换机构通

常设置一个公用的基址寄存器(Base Register,BR),它存放现行进程在内存空间的起始地址。当 CPU 经虚拟地址访问内存时,地址变换机构将自动把该虚拟地址加上 BR 中的地址以形成实际物理地址。显然,如果程序在内存的存放位置发生了改变,则只要改变 BR 的内容,就可以正确地访问程序。图 5-4 是实现动态地址重定位的示意图。

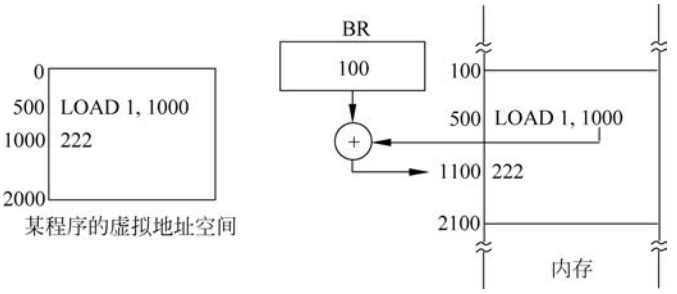


图 5-4　实现动态地址重定位的示意图

在图 5-4 中的程序目标模块装入内存时,与地址有关的各项均保持原来的虚拟地址不进行任何修改。例如,"LOAD 1,1000"这条指令在装入内存后,其中要访问的地址并没有改变(仍是虚拟地址 1000)。当此模块被操作系统调度到处理机上执行时,操作系统将把此模块装入的实际起始地址减去目标模块的相对基地址(图 5-4 中该基地址为 0),然后将其差值(100)装入基址寄存器 BR 中。当 CPU 执行"LOAD 1,1000"指令时,地址变换机构自动将指令中的虚拟地址(1000)与基址寄存器中的值(100)相加,再把和的值(1100)作为内存绝对地址去访问该单元中的数据。

由此可见,实现动态地址重定位是在指令执行过程中每次访问内存前动态地进行的。采取动态地址重定位可带来两个好处。

(1) 目标模块装入内存时无须任何修改,因而装入之后再搬迁也不会影响其正确运行,这对于后面将要介绍的存储器紧缩、解决碎片问题是极其有利的。

(2) 一个程序由多个相对独立的目标模块组成时,每个目标模块各装入一个存储区域,这些存储区域可以不是顺序相邻的,只要各个模块有自己对应的定位存储器就行。

动态地址重定位技术所付出的代价是需要硬件支持。

3. 内存信息的共享和保护

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。

下面介绍几种常用的内存信息保护方法。

(1) 上下界保护法。这是一种常用的硬件保护法。该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,若是,则访问是合法的,否则是非法的,并产生访址越界中断。

(2) 保护键法。该方法为每一个被保护存储块分配一个单独的保护键,保护键可设置

成对读写同时保护,也可设置成只对读或写进行单项保护。在程序状态字(PSW)中设置相应的保护键开关字节,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相匹配。如果开关字中的代码与保护键匹配或存储块未受到保护,则允许访问该存储块,否则将产生访问出错中断。例如,在图5-5(a)所示的内存状态中,保护键0就是对2～4KB的存储区进行读写同时保护的,而保护键5则只对4～6KB的存储区进行读保护,保护键3只对6～8KB的存储区进行写保护。假设当前的程序状态字为5,图5-5(b)给出了采用保护键法的一些合法访问指令与非法访问指令实例。其中,对于指令"LOAD 1, 5000"与"STORE 2, 5200",它们访问的内存存储单元的保护键为5,与当前的程序状态字(5)相匹配,所以是合法指令;对于指令"LOAD 2, 7000",虽然它访问的存储单元(6～8KB)的保护键为3,与当前的状态字不匹配,但是存储单元6～8KB保护是写保护,因此可以读;对于指令"LOAD 1, 2500"与"STORE 1, 7000",它们访问的存储单元的保护键与当前的状态字不匹配,同时访问的存储单元的保护分别有读写保护与写保护,因此是非法指令。

(a) 内存状态　　(b) 合法与非法访问

图5-5　保护键法

(3) 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。在这种保护模式下,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系统就采用了这种内存保护方式。

4. 内存扩充

为了满足大进程的存储请求,内存管理应该能够在内存中存放尽可能多的用户进程,给它们分配得以运行的足够的内存空间,从而提高整个系统的使用效率。但是,实际计算机的内存容量是有限的,这就要求计算机系统提供"扩充的内存",以保证内存分配的需要。这种扩充不是指物理内存的扩充,而是指利用存储管理软件为用户程序提供一个比实际内存容量更大的逻辑存储空间,即所谓的虚拟存储技术。

事实上,上述4个任务也是存储管理要解决的4个基本问题,各种存储管理方案都是以它们作为基本的出发点。内存分配、地址变换、内存保护是任何管理方案必须实现的,比较完善的管理方案则实现了内存扩充。

5.2　分区式存储管理

分区式存储管理是满足多道程序设计的最简单的一种存储管理技术。分区的基本思想是将内存区域划分成若干大小不等的区域,每个区域称为一个分区,每个分区存放一道进程

对应的程序和数据，使进程在内存中占用一个分区，而且进程只能在所在分区内运行。分区可分为固定式和可变式两类。

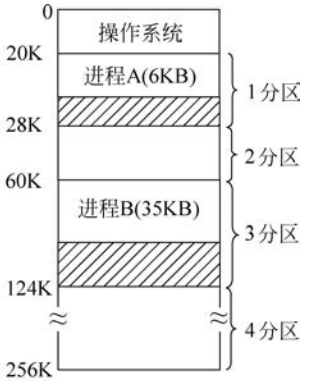
5.2.1 固定分区

固定分区是指在用户进程装入之前，系统或系统操作员将用户空间划分为若干固定大小的区域，每个进程占用一个分区，划分好的分区大小和个数在整个系统运行期间不会变化，因此也称为静态分区。

系统对内存的管理和控制通过数据结构——分区说明表来实现的。分区说明表中的每个表项记载一个分区的特性，包括分区号、分区大小、分区起始地址和状态（已分配用 1 表示，空闲用 0 表示）。内存的分配与释放、存储保护以及地址变换等都是通过分区说明表进行的。分区说明表存放在系统区内，不能由用户改动。图 5-6 给出了分区说明表和其对应的内存状态示例。

分区号	分区大小	起始地址	状态
1	8KB	20K	1
2	32KB	28K	0
3	64KB	60K	1
4	132KB	124K	0

(a) 分区说明表



(b) 内存空间分配情况

图 5-6 固定分区示例

从图 5-6 可以看出，2、4 分区为空闲区，8KB 的 1 分区被 6KB 的进程 A 占用，有 2KB 内存未被使用（图中阴影部分），64KB 的 3 分区被 35KB 的进程 B 占用，也留有 29KB 内存未被使用，分区内这些未被使用的无效的存储空间称为内碎片。内碎片的存在降低了内存空间的利用率，有时会造成内存空间的极大浪费。

当一个进程要求分配内存时，系统依次查找分区说明表中的表项，将分区大小满足进程请求容量并且状态为 0 的空闲分区分配给该进程，修改该分区对应的表项，置状态为 1。进程运行完毕后，系统将进程释放的分区收回，分区表项对应的状态置为 0。

固定分区管理能使多个进程共享内存，具有数据结构简单、分配和回收算法容易实现等优点。但是，存在小进程占据大分区造成内碎片和可调入的进程大小受到分区大小限制等问题。例如，在图 5-6 中，若进程 C 需要 135KB 内存，虽然所有的空闲区总和大于 135KB，但进程 C 也分配不到内存，因为没有一个分区的大小大于或等于 135KB。

5.2.2 可变分区

1. 可变分区基本思想

可变分区是指在装入进程的过程中，根据进程的实际需要建立分区，该分区的大小正好

等于进程的大小。随着系统的运行,内存中的分区大小和个数都会发生变化,因此,可变分区又称为动态分区。

系统初起时,内存中除操作系统区之外,其余内存空间为一个完整的大空闲区。当有进程要求装入运行时,从该空闲区中划分一块与进程大小一样的区域分配给该进程。当系统运行一段时间后,经过一系列的分配与回收,原来的一整块大空闲区形成了若干占用区(已分为两个分区)和空闲区相间的布局。

图5-7是一个可变分区分配和回收的示例。图5-7(a)是某时刻内存状态,此时,内存中装入了A、B、C这3个进程,同时又有进程D和进程E请求装入;图5-7(b)中系统为进程D分配一块内存区,此时,内存中剩下了两块较小的空闲区,进程E的需要不能满足;图5-7(c)表示进程A和C已完成,系统回收了它们占用的分区,在进行了适当的合并后,内存中形成了3块空闲区,它们的总容量虽然远大于进程E的需求量,但每块容量均小于进程E的容量,故系统仍不能为进程E实施分配。从图5-7可以看出,随着内存分配和回收的增加,分区之间出现了一些较小的空闲区,而这些空闲区将无法分配给进程使用,这种分区之间的无效的空闲区称为外碎片。解决外碎片的办法是进行碎片的"拼接",即移动已分配分区,把所有空闲区碎片合并成一个连续的大空闲区。拼接可以选择在以下时机进行:当回收某个占用区时,如果它没有邻接空闲区,但内存中有其他空闲区,则马上进行拼接;当需要为新进程分配内存空间,但找不到足够的容纳该进程的空闲区,而所有空闲区总容量却能满足需求时,再进行拼接。实际中,使用后者较多,在拼接过程中被移动了的进程需要进行重定位,这可用动态地址重定位来实现。虽然拼接技术可以解决碎片问题,但拼接工作需要大量的系统开销,在拼接时必须停止所有其他工作,对于分时系统,拼接将损害系统的响应时间。

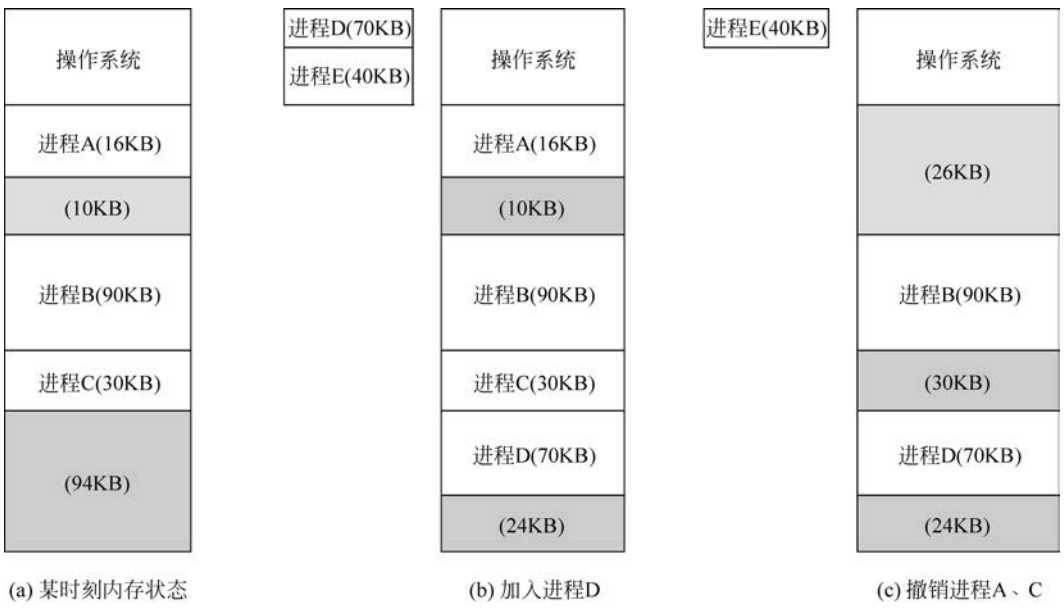


图5-7 可变分区分配和回收的示例

2. 分区使用的数据结构

为了实现分区分配,系统中必须配置相应的数据结构,用来描述空闲分区和已分配分区

的情况，为分配提供依据。常用的数据结构有以下两种形式。

(1) 分区分配表。用于记录每个已分配分区的情况。每个分区占一个表目，表目中包括分区号、起始地址及分区大小等数据项。

(2) 空闲分区链(表)。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(如大小、起址)以及用于链接各分区所用的前后指针，在分区尾部则设置一个后向指针，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。

3. 分区分配与回收操作

1) 分配内存

当一个进程要求装入内存时，系统将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，为进程分配内存空间。假设请求的分区大小为 u.size，空闲分区链中每个空闲分区的大小可表示为 m.size。若 m.size－u.size≤size(size 是事先规定的不再切割的剩余分区的大小)，说明多余部分太小，可不再分割，将整个分区分配给请求者；否则(即多余部分超过 size)，从该分区中按请求的大小划分出一块内存空间分配出去，余下的部分仍留在空闲分区链中。然后，将分配的分区的首址返回给调用者。图 5-8 给出了内存分配流程。

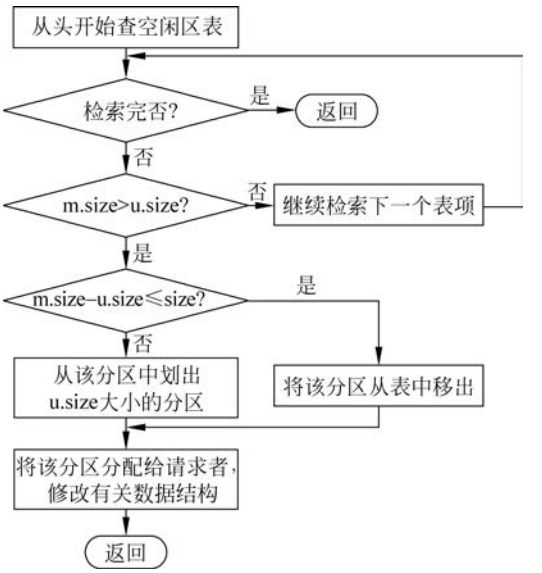


图 5-8　内存分配流程

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分区链中找到相应的插入点，此时可能出现以下 4 种情况之一。

(1) 回收区与插入点的前一个空闲分区 F1 相邻接，见图 5-9(a)。此时应将回收区与插入点的前一分区 F1 合并，不必为回收区在空闲分区链中分配新表项，而只需修改其前一分区 F1 的大小。

(2) 回收区与插入点的后一空闲分区 F2 相邻接，见图 5-9(b)。此时也可将两分区合并，形成新的空闲分区，但用回收区的首址作为新空闲分区的首址，分区大小为两者之和。

(3) 回收区同时与插入点的前一空闲分区 F1 和后一空闲分区 F2 相邻接，见图 5-9(c)。

此时将3个分区合并,使用F1的表项和F1的首址,取消F2的表项,分区大小为三者之和。

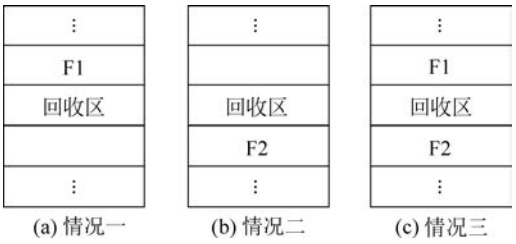


图5-9　内存回收时的情况

(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个新表项,填写回收区的首址和大小,并根据其首址插入空闲分区链中的适当位置。

4. 分区分配算法

为把一个新进程装入内存,需按照一定的分配算法,从空闲分区链中选出一个分区分配给该进程。目前常用以下7种分配算法。

(1) 首次适应算法。该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;然后再按照进程的大小,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分区链中。若从链首直至链尾不能找到一个满足要求的分区,则此次内存分配失败。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分区。这为向以后到达的大进程分配大的内存空间创造了条件。其缺点是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的,这无疑会增加查找可用空闲分区时的开销。

(2) 循环首次适应算法。该算法是由首次适应算法演变而成的。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进程。为实现该算法,应设置一个起始查找指针,用于指示下一次起始查找的空闲分区,并采用循环查找方法,即如果最后一个(链尾)空闲分区的大小仍不能满足要求,则应返回到第一个空闲分区,看其大小是否满足要求。找到后,应调整起始查找指针。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,但这样难以保留大的空闲分区。

(3) 最坏适应算法。该算法要求空闲分区按其容量以从大到小的顺序形成一个空闲分区链。分配时,先检查空闲分区链中的第一个空闲分区,若它的大小不满足要求,则分配失败;否则从该分区中划出进程所要求大小的一块内存空间分配给进程,余下的空闲分区则重新排列后插入空闲分区链。最坏适应算法的特点是,总是挑选满足进程要求的最大分区分配给进程,这样使分给进程后剩下的空闲分区也比较大,也能装下其他的进程。但是由于最大的空闲分区总是首先被划分,当以后有大的进程到来时,其申请的存储空间往往不能得到满足。

(4) 最佳适应算法。所谓“最佳”是指每次为进程分配内存时,总是把能满足要求同时又是最小的空闲分区分配给进程,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个空闲分区链。这样，第一次找到的能满足要求的空闲分区必然是最佳的。孤立地看，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，在内存中会留下许多难以利用的小空闲分区。

(5) 快速适应算法。也称为分类搜索法，为每一类具有相同容量的空闲分区设立一个空闲分区链表，每个链表的表头指针放在一个索引表中。在分配内存时，根据进程要求的内存大小从索引表中找到最小的能容纳进程的空闲分区链，从链中取下第一块空闲分区分配。该算法的优点是，分配时不会对分区进行分割，保留大的分区，不会产生碎片，查找效率高。其缺点是合并算法复杂。

(6) 伙伴系统算法。也称为 Buddy 算法，把内存中的所有空闲分区按照 2^k ($1 \leqslant k \leqslant n$) 的大小划分，划分后形成了大小不等的存储区，所有相同的空闲分区形成一个链。在某进程请求分配 m 大小的内存时，首先计算 i，使 $2^{i-1} \leqslant m \leqslant 2^i$，然后在 2^i 空闲分区链中搜索。若空闲分区链不为空，则分配一个空闲分区给请求的进程。若没有，则查询 2^{i+1} 空闲分区链，若有 2^{i+1} 大小的空闲分区，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。伙伴必须是从同一个大分区中分离出来的，其中一个用于分配，另一个加入 2^i 的空闲分区链中。若大小为 2^{i+1} 的空闲分区也不为空，则需找大小为 2^{i+2} 的空闲分区，在找到可利用的空闲分区后，则进行两次分割，一个用于分配，一个加入 2^{i+1} 的空闲分区链中，一个加入 2^i 的空闲分区链中，以此类推。一次分配可能要进行多次分割才能完成。在内存分区释放时，检查是否有空闲的伙伴，如果有则合并，因此一次释放也可能要进行多次合并。伙伴系统算法可以为进程分配连续的地址空间，也可以使内存空闲分区以各种尺寸再分配，最大限度地解决了内存分配引起的外碎片问题。但是不可避免地存在内碎片问题。在分割与合并分区时需要额外的系统开销。

(7) 哈希算法。构建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头指针。当分配内存时，以空闲分区的大小为参数，经过哈希函数计算得到的值，为相应空闲分区在哈希表中的位置，从而快速地实现内存分配。

5.2.3 地址变换与内存保护

对于分区管理，也同样存在每个用户可以自由编程的虚拟空间，因此在程序装入内存时，可采用静态重定位技术或动态重定位技术来完成程序的地址重定位。固定式分区通常采用静态方式。但对于可变式分区，如果采用拼接技术进行分区的合并，则不适合采用静态重定位技术。

分区的保护可采用界限寄存器法或保护键法。采用界限寄存器法，一般设置一对公用界限寄存器：基址寄存器(Base Register，BR)和限长寄存器(Limit Register，LR)，它们分别存放现行进程的分区界限值，即分区的起始地址和大小。也可以设置多对界限寄存器，每一对界限寄存器对应一个分区。利用界限寄存器可同时实现对分区管理的地址变换和保护，如图 5-10 所示。即假设 CPU 要访问的逻辑地址为 LA。若 LA＞LR，则说明地址越界，将产生保护性地址越界中断，系统转出错处理；若 LA≤LR，则 LA 与 BR 中的基址形成有效的物理地址。

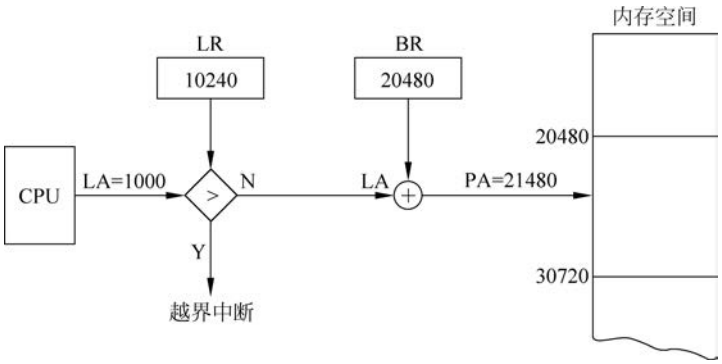


图 5-10　分区管理的地址变换和保护

5.2.4　分区式存储管理的优缺点

分区式存储管理的主要优点如下。

(1) 实现了多个作业或进程对内存的共享,有助于多道程序设计,从而提高了系统的资源利用率。

(2) 要求的硬件支持少,管理算法简单,因而实现容易。

其主要缺点如下。

(1) 内存利用率仍然不高。存在着严重的零碎空闲分区(碎片)不能利用的问题。

(2) 作业或进程的大小受分区大小控制,除非配合采用覆盖和交换技术来实现内存的扩充。

(3) 难以实现各分区间的信息共享。

5.3　页式存储管理

在分区式存储管理方案中,一个进程总占用一块连续的内存区域,因此产生了碎片问题。虽然采用拼接技术可以使零散的碎片连成一片,但要花费大量的处理机时间。为了更好地解决碎片问题,人们提出了另一种内存管理方案,即页式存储管理方案。该管理方案将使一个进程可以存放在不连续的内存区域中。

页式存储管理方案分为两种:静态页式存储管理方案和动态页式存储管理方案。

5.3.1　静态页式存储管理

1. 基本思想

系统首先把内存的存储空间划分成若干大小相等的小区域,每个小区域称为页面或内存块,页面按 0,1,2,…依次编号。同样,每个进程的虚拟地址空间也被分成若干与页面大小相等的多个片段,称之为页,编号为 0,1,2,…。分配内存时,进程的每一个页装入内存的一个页面。同一进程的多个页可以分配在编号不连续的内存内。

静态页式存储管理又称为简单页式存储管理,它的特点是:系统如能满足一个进程所要求的全部页面数,此进程才能被装入内存,否则,不为它分配任何内存。

2. 数据结构

在分页式存储管理方案中，系统使用了页表、请求表、存储页面表 3 种数据结构来实现虚拟地址到物理地址的变换，完成内存的分配和回收工作。

1）页表

一个进程往往有多个页，而这些页在内存中又可以不连续存放。那么，系统如何知道进程的每一页分别存放在内存的哪一个页面呢？分散存放在内存中的进程又是如何正确运行的呢？为解决上述问题，系统在将进程装入内存时，就为每个进程建立一个页表，用于记录一个进程在内存中的分配情况，同时还可利用页表实现逻辑地址到物理地址的变换。

页号	页面号

图 5-11　页表示例

最简单的页表由页号与页面号(块号)组成，如图 5-11 所示。页表在系统区中占有一块固定的存储区。页表的大小由进程长度决定。例如，对于一个每页长为 1KB、大小为 20KB 的进程来说，如果一个内存单元存放一个页表项，则只要分配给该页表 20 个存储单元即可。

2）请求表

为了确定各进程的页表在内存中的实际对应位置以及每个进程所要求的页面数，系统建立了一张请求表，如图 5-12 所示。请求表中记录了每个进程的页表起始地址、页表长度和进程所要求的页面数，状态表示对应的进程是否已建立了相应的页表。在实际系统中请求表由所有进程的 PCB 对应的表项组成，也就是说，每个进程的 PCB 中记录着其相应的页表的起始地址和长度。

进程号	请求页面数	页表始址	页表长度	状态
1	20	1024	20	已分配
2	34	1044	34	已分配
3	18	1078	18	已分配
4	21	…	…	未分配
⋮	⋮	⋮	⋮	⋮

图 5-12　请求表示例

3）存储页面表

存储页面表是整个系统一张，该表指出内存各页面是否已被分配出去以及未分配页面的总数。

存储页面表也有两种构成方法，一种是在内存中划分一块固定区域，每个存储单元的每一位代表一个页面，如果该页面已被分配，则对应的位置 1，否则置 0。这种方法称为位示图法。图 5-13 给出了位示图的示例。位示图要占据一部分内存。例如，一个划分为 1024 个页面的内存，如果内存单元长 20 位，则位示图要占据 1024/20＝52 个存储单元。

存储页面表的另一种构成方法是采用空闲页面链。在空闲页面链中，链首页面的第一个单元和第二个单元分别放入空闲页面总数与指向第一个空闲页面的指针，其他页面的第一个单元则分别放入指向下一个空闲页面的指针。这样所有的空闲页面就链在了一起。空闲页面链的方法由于使用了空闲页面本身的单元存放指针，因此不占用额外的内存空间。

位	19	18	17	16	15	…	4	3	2	1	0
	0	1	1	1	1	…	1	1	0	1	1
	0	0	0	1	1	…	0	0	1	1	0
	0	0	1	1	1	…	0	0	0	0	1

图 5-13　位示图示例

3. 分配与回收算法

利用上述 3 种数据结构,页式存储管理的分配算法实现起来相对简单。分配程序按请求表给出进程所要求的页面数,检查存储页面表中是否有足够的空闲页面。如果没有,则本次分配无法实现;如果有,则首先建立页表,然后搜索出满足要求的空闲页面,并将对应的页面号填入页表中。图 5-14 给出了上述页面分配算法的流程图。

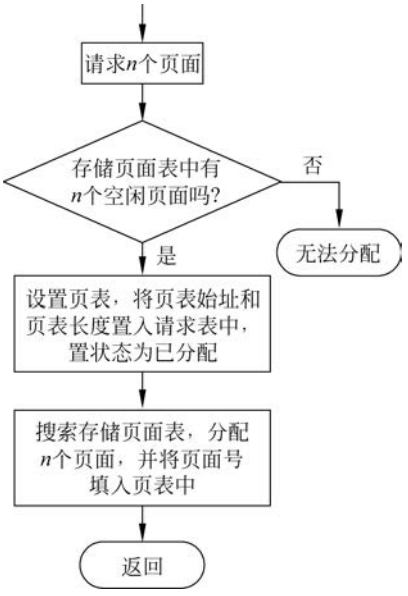


图 5-14　页面分配算法的流程图

静态页式存储管理的页面回收方法较为简单,当进程执行完成时,撤销对应的页表,并把页表中的各页面插入存储页面表的空闲页面链中。

4. 地址结构及地址变换

静态页式存储管理的一个关键问题是地址变换,即怎样利用页表将逻辑地址转换为物理地址的问题。为此,系统中必须设置地址变换机构(硬件)来完成地址的变换。

1) 地址结构

在学习利用页表是怎样完成地址变换之前,必须了解地址结构。在页式存储管理中,分页系统的地址变换机构自动把逻辑地址解释为两部分:高位部分为页号,低位部分为页内地址。因此,逻辑地址用一个数对(p,d)来表示,其中 p 表示逻辑地址所在的页号,d 表示页内地址。p、d 所占的位数取决于页的大小和有效地址长度。为了简化地址变换过程和分页,通常页的大小为 2 的 n 次幂。如取页大小为 $1024B=2^{10}B=1KB$ 或 $4096B=2^{12}B=4KB$ 等。若页的大小为 2^n,则 d 所占的位数为 n,p 所占的位数为有效地址长度 $-n$。对于某特定的计算机,其地址结构是一定的。

例如,假设计算机 CPU 有效地址为 16 位,且页大小 $L=1KB$。在地址变换时,分页系统的地址变换机构自动把逻辑地址解释为两部分:页号 p(6 位)和页内地址 d(10 位),如图 5-15 所示。

图 5-15　地址结构示例

若给定逻辑地址为 A，页面的大小为 L（单位为字节），则页号 p 和页内地址 d 可按下式求得：

$$p = \mathrm{INT}\left(\frac{A}{L}\right)$$

$$d = A \ \mathrm{MOD}\ L$$

其中，INT 是取整函数，MOD 是取余函数。例如，假设页大小 $L = 2\mathrm{KB} = 2048\mathrm{B}$，$A = 4200$，则按上式得：$p = 2$，$d = 104$。

2）地址变换

页表的功能可以由一组专门的寄存器来实现，一个页表项用一个寄存器。由于寄存器具有较高的访问速度，因而有利于提高地址变换的速度。但由于寄存器成本较高，且大多数现代计算机系统的内存较大，使页表项的总数可达几千甚至几十万个，显然这些页表不可能都用寄存器来实现。因此，页表大都是驻留在系统区内的一个数据结构，而在系统中只设置一个页表控制寄存器，在其中存放当前运行的进程的页表始址和页表长度。当进程未执行时，页表的始址和长度存放在本进程的 PCB 中；当进程被调度执行时，才将这两个数据装入页表控制寄存器中。在单处理机环境下，虽然系统中可以运行多个进程，但只需一个页表控制寄存器（因为任何时刻只能运行一个进程）。下面通过例子来说明页式存储管理的地址变换过程。

例 5-1 假设系统的页面长度为 1KB。当前正在执行的进程对应的页表如图 5-16 所示。当前进程中有一条指令“LOAD 1，2500”，如何利用页表进行地址变换才能正确地执行这条指令呢？

页号	页面号
0	2
1	3
2	8

图 5-16 页表

解：当 CPU 执行指令“LOAD 1，2500”时，CPU 将逻辑地址 2500 放入有效地址寄存器中。将逻辑地址 2500 映射为对应的物理地址的变换过程如图 5-17 所示。

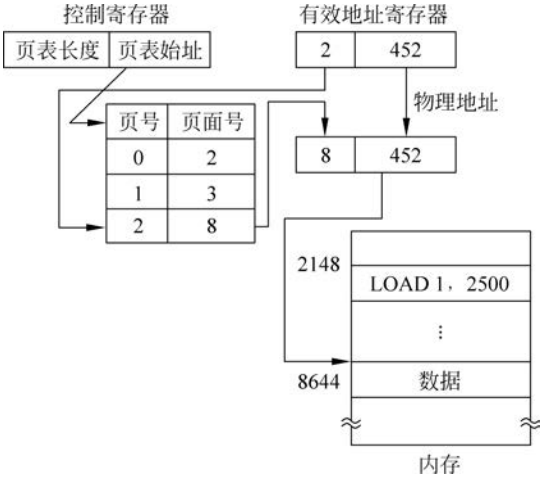


图 5-17 页式存储管理的地址变换过程

为了找出 2500 对应的物理地址，分页系统地址变换机构首先将逻辑地址 2500 转换为两部分：$p = 2$，$d = 452$。然后，以页号为索引检索页表。在执行检索前，先将页号与页表长度进行比较。如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址

空间,将发生地址越界中断;否则,将页表始址(在页表控制寄存器中)与页号(2)和页表项长度的乘积相加,便得到该表项在页表中的位置,于是可从中得到进程的2号页存放在页面号为8的内存中,将找到的页面号(8)装入物理地址寄存器中。与此同时,再将有效地址寄存器中的页内地址(452)送入物理地址寄存器的块内地址字段中。由此,页面号(8)与页内相对地址(452)相连,得到逻辑地址2500对应的物理地址为8×1024+452=8644。

由于静态页式存储管理要求进程在分配到其申请的全部页面后才可装入内存得到执行,这使得进程的大小受到可用页面数的限制。而事实上,在一段时间内,CPU总是集中地访问程序中的某一部分。例如,含有局部变量、常用函数、循环语句的程序段往往是经常被访问的部分,人们把这种现象称为局部性原理。这就使得系统为用户进程分配一定数量的内存页面就可使程序能正确地运行,为用户提供一个比实际内存更大的虚拟存储空间。由此,在静态页式存储管理的基础上产生了动态页式存储管理。

5.3.2 动态页式存储管理

1. 基本思想

动态页式存储管理的基本思想是,当一个进程要求运行时,不是把整个进程的程序代码和数据全部装入内存,只需将当前要运行的那部分程序代码和数据所对应的页装入内存,便可启动运行。以后在进程运行的过程中,当需要访问某些页时,由系统自动地将需要的页从外存调入内存。如果内存没有足够的空闲页面,则将暂不运行的进程页调出内存,以便装入新的进程页。

动态页式存储管理使得用户的程序空间不再受内存空间大小的限制,即系统从逻辑上可以为用户提供一个比实际内存更大的虚拟存储空间,从而实现了内存的逻辑扩充技术。为了区别内外存中的不同的页,通常把一个进程经分配调入内存中的页称为实页,把外存中页称为虚页。

动态页式存储管理中的程序地址空间的分页和地址变换与静态页式存储管理完全相同。但是,由于动态页式存储管理只给运行的进程分配必要的内存空间,因此,必然会产生两个问题。

(1) 当进程要访问的某个虚页不在内存中时,怎样发现这种缺页情况,发现后又怎么办?

(2) 当需要把某一虚页调入内存时,若此时内存中没有空闲页面,应该怎么处理?

以下将介绍动态页式存储管理方案中解决这两个问题的主要技术原理。

2. 页表

为了知道哪些页已经调入内存,哪些页还没有调入内存,以及对页的换进和换出进行控制,需扩充页表。扩充后的页表字段如下:

页号	页面号	驻留位	访问位	修改位	外存地址

各字段的说明如下:

(1) 驻留位:用于指示该页是否在内存中。

(2) 访问位:用于记录该页在一段时间内被访问的次数,或记录本页最近有多长时间未被访问,供页换出时参考。

(3) 修改位：表示该页在调入内存后是否被修改过。由于内存中每一页在外存中都保留了一份副本，因此，若该页在内存中存放时未被修改过，则在置换该页时就不需要将该页写回到外存上，以减少系统启动外存的次数；若已被修改过，则必须将该页写回到外存上，以保证外存中所保留的始终是最新的副本。

(4) 外存地址：用于指出该页在外存中的地址，以便系统从外存中调入该页。

3. 缺页中断机构

当 CPU 需访问的页不在内存中时，将发生缺页中断，这一过程由专门的硬件机构实现。如图 5-18 所示，当发生了缺页中断后，CPU 将转去执行相应的缺页中断处理过程。图 5-18 中的虚线下面给出了缺页中断的处理过程。当内存中没有空闲页面时，系统将按照一定的淘汰算法(有关的淘汰算法将在 5.4 节介绍)，在内存中选择一页淘汰，将需要的虚页调入内存。

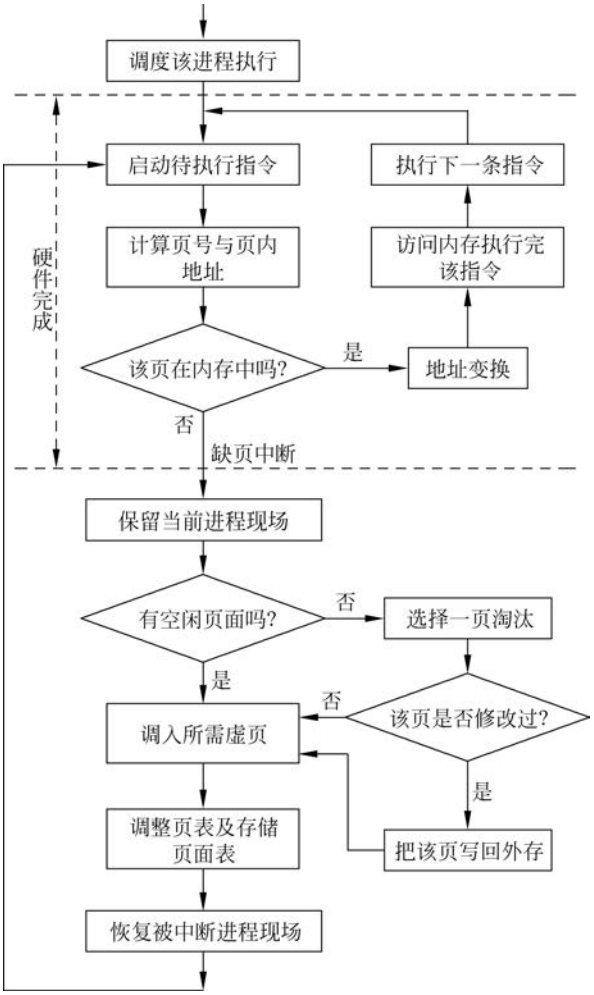


图 5-18　缺页中断的处理过程

5.3.3 指令存取速度与页面大小问题

1. 快表

在页式存储管理系统中,存取一个数据或执行一条指令要访问两次主存:一次是访问页表,另一次是访问真正的内存地址。显然,这种方法比通常执行指令的速度慢了一倍。为了加快指令的存取速度,可在地址变换机构中增设一个具有并行查找能力的特殊的公共高速缓冲寄存器,又称联想寄存器(Associative Memory)或快表。联想寄存器由一组可以与内存并行访问的寄存器(通常为16～512个)组成,用于存放当前进程经常要访问的那些页的页表项。此时的地址变换过程是:在CPU给出有效地址后,由地址变换机构自动将页号p与快表中的所有页号进行比较,若其中有与此匹配的页号,便可直接从快表中读出与该页相对应的物理块号,进行地址变换;否则访问内存中的页表,找到该页对应的物理块号,进行地址变换,同时将该页对应的表项存入快表的一个寄存器单元中。在快表中插入一个页表项时,如果寄存器已满,则要进行快表的置换,即在快表中选择一个访问位值最小的表目,将之淘汰,以便插入新的页表项。所谓访问位值最小,可认为是该页在过去一段时间内被访问的次数最少。为此,每访问一次该页,就要对该页的访问位值进行计数,即访问位值反映了该页被访问的频度;或者是让该页最先进入快表,即访问位值反映了该页进入快表的次序。

事实上,查找页表和查找快表的两项工作在硬件上是同时进行的,一旦发现该页在快表中,就立即停止查找内存页表。程序和数据的访问往往带有局部性,据统计,从快表中能找到所需页表项的概率可达90%以上,而由于访问快表的速度要比访问页表的速度高一个数量级,从而利用快表大大提高了页式存储管理地址变换的速度。图5-19描述了具有快表的地址变换过程。

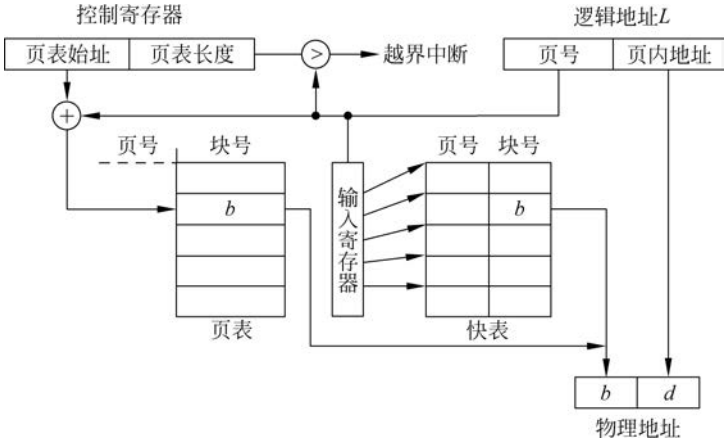


图5-19 具有快表的地址变换过程

2. 页面大小

由于进程的地址空间被连续地划分成若干页,系统以页为单位分配内存,块与页等长。因此,除了一个进程中的最后一页可能不足一个页的大小,而产生页内碎片外,内存中不会存在不可利用的空闲页面。因此,静态页式存储管理解决了分区式存储管理的碎片问题。但是页面的大小选择也要恰当。

选择最优的页面大小需要在几个相互冲突的因素之间折中。如果页面太大，页式存储管理就退化为分区式存储管理，同时导致页内碎片过大；而如果页面太小，页表将占用内存空间太多。一个系统的页表占用内存的空间大小与虚存大小和页大小有关。而现代计算机系统都支持大的逻辑地址空间($2^{32}\sim2^{64}$)，这样页表就非常大。例如，一个进程的虚存大小为16MB，页大小为2KB，则该进程的页数为16×1024/2=8192个。若一个页表的表目占2个字节，那么页表占用内存的空间大小为8192×2B=16KB。在动态页式存储管理中，如果页面太小，则还要增加内外存交换次数。因此，在实际使用中考虑众多因素，大部分的计算机使用的页面大小为512B～64KB。

5.3.4 存储保护

页式存储管理的页面保护可采用界限寄存器法或保护键法。界限寄存器法通过地址变换机构中的页表寄存器中的页表长度和所要访问的逻辑地址相比较来完成，如图5-19所示。保护键法的实现则是在页表中增加相应的保护位即可。

5.3.5 页式存储管理的优缺点

1. 优点

(1) 由于页式存储管理不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了内存的碎片问题，因此，可使内存得到有效的利用，有可能使更多的进程同时投入运行，可进一步提高处理机的利用率。

(2) 动态页式存储管理只要求每个进程部分装入便可运行，实现了内存的扩充技术。可为用户提供比实际内存更大的虚拟存储空间，使用户可利用的存储空间大大增加，有利于多道程序的组织，可以提高内存的利用率。

2. 缺点

(1) 要求有相应的硬件支持，如地址变换机构、缺页中断机构和页面淘汰机构等。这些增加了计算机的成本。

(2) 增加了系统开销。例如，页面中断处理，表格的建立和管理，这些都需花费处理机时间，且表格还要占用一定的存储空间。

(3) 淘汰算法选择不当有可能会严重影响系统的使用效率。

(4) 虽然消除了碎片，但同时还存在页内碎片问题。

5.4 淘汰算法与抖动现象

动态页式存储管理虽然能有效地提高主存的利用率，实现了内存的扩充技术，但同时也带来了一些新的问题。例如，为使一个进程能正确地运行应为它分配多少个内存页面，当访问的页不在内存时采用什么样的淘汰算法，等等。下面讨论这些问题。

5.4.1 淘汰算法

进程在运行过程中，若所访问的页不在内存时，必须把它们调入内存，但当内存空闲页面不足时，必须从内存中调出一页或多页，送到磁盘的交换区中，以便把需要的页调入内存。

然而应该将内存中的哪些页调出内存,需根据一定的算法来确定。通常把选择要换出页的算法称为淘汰算法。而一个好的淘汰算法应使缺页率尽可能小。

在进行页面置换时,可采用全局或局部置换。局部置换指进程发生缺页时,只能从分配给该进程的内存页面中选择一页换出。全局置换是指进程发生缺页时,选择淘汰的页可能是内存中任一进程的页。

下面介绍几种常用的淘汰算法。为简化算法,这些算法均采用局部置换。

1. 最佳淘汰算法

最佳(Optimal)淘汰算法是一种理想的淘汰算法,它选择将来不再使用或者在最远的将来才可能被使用的页淘汰。显然采用最佳淘汰算法可以保证得到最低的缺页率。但是实际上,当发生缺页时,操作系统根本没有办法知道每一个页将在什么时候被访问,所以,最佳淘汰算法是不能实现的。尽管如此,该算法仍然有意义,它可作为衡量其他算法优劣的一个标准。

为了比较各种算法的优劣,下面通过一个例子来说明。

例 5-2 假设一个进程 p 有 6 页,该进程执行过程中,访问页号的顺序是 0,2,5,3,2,4,2,0,3,2,1,3,2,3,4,3。如果给进程 p 分配 3 个内存页面,那么,采用最佳淘汰算法时,执行进程 p 将会发生多少次缺页中断?缺页率是多少?

解:采用最佳淘汰算法时,进程 p 的各页在内存中的变换如图 5-20 所示。

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
内存实页	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			5	3	3	4	4	4	4	4	1	1	1	1	4	4
缺页中断	√	√	√	√		√			√		√				√	

图 5-20 最佳淘汰算法的置换图

从图 5-20 中可以看出,进程 p 在执行过程中发生了 5 次缺页,缺页率是 5/16=31.25%。

2. 先进先出淘汰算法

先进先出(FIFO)淘汰算法认为最先调入内存的页不再被访问的可能性要比其他页大,因而选择最先调入内存的页换出。实现 FIFO 淘汰算法比较简单,只需把各个已分配的页按分配时间顺序链接起来,组成 FIFO 队列,并设置一个指针,称为置换指针,使它指向 FIFO 队列队首页面。在选择一页淘汰时,总是淘汰置换指针指向的页,而把换进的页链接入 FIFO 队尾。

例 5-3 对于例 5-2,如果采用 FIFO 淘汰算法,执行进程 p 将会发生多少次缺页中断?缺页率是多少?

解:采用 FIFO 淘汰算法时,进程 p 的各页在内存中的变换如图 5-21 所示。

从图 5-21 中可以看出,进程 p 在执行过程中发生了 9 次缺页,比最佳算法多了 4 次,缺页率是 9/16=56.25%。

FIFO 淘汰算法容易实现,但是它所依据的假设与普遍的进程运行规律不符。它只适用于 CPU 按线性顺序访问地址空间的进程。而实际上,由局部性原理知,大部分时候,

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
内存实页	0	0	0	3	3	3	3	0	0	0	0	0	2	2	2	2
		2	2	2	2	4	4	4	3	3	3	3	3	3	4	4
			5	5	5	5	2	2	2	2	1	1	1	1	1	3
缺页中断	√	√	√	√		√	√	√	√		√		√		√	√

图 5-21　先进先出淘汰算法的置换图(1)

CPU 不是按线性顺序访问地址空间的。例如,含有局部变量、常用函数、循环语句的页,虽然在内存中驻留了很久,但是它们往往是经常被访问的页。而 FIFO 淘汰算法可能使这些页刚刚被淘汰出去而又要立即被调回内存,从而使缺页率变大。

FIFO 淘汰算法的另一个缺点是它有一种陷阱现象。一般来说,对于任一作业或进程,如果给它分配的内存页面数越接近它所要求的页面数,则发生缺页的次数会越少。在极限情况下,这个推论是成立的。因为如果给一个进程分配了它所要求的全部页面,则不会发生缺页现象。但是,使用 FIFO 淘汰算法时,在未给进程或作业分配足它所要求的页面数时,有时会出现分配的页面数增多时,缺页的次数并不会减少陷阱现象。这种现象被称为 Belady 现象。

下面通过例子来说明 Belady 现象。

例 5-4　假设一个进程 p_1 有 5 页,该进程执行过程中访问页号的顺序是 1,2,3,4,1,2,5,1,2,3,4,5。如果给进程 p_1 分配 3 个内存页面,采用 FIFO 淘汰算法时,进程 p_1 在内存中的各页变换如图 5-22 所示。

访页顺序	1	2	3	4	1	2	5	1	2	3	4	5
内存实页	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
缺页中断	√	√	√	√	√	√	√			√	√	

图 5-22　先进先出淘汰算法的置换图(2)

由图 5-22 中可以看出,进程 p_1 在执行过程中发生了 6 次缺页。

但是,如果为进程 p_1 分配 4 个内存页面,则进程 p_1 在内存中的各页变换如图 5-23 所示。

访页顺序	1	2	3	4	1	2	5	1	2	3	4	5
内存实页	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
缺页中断	√	√	√	√			√	√	√	√	√	√

图 5-23　先进先出淘汰算法的置换图(3)

由图5-23中可以看出,进程 p_1 在执行过程中发生了6次缺页,比分配3个页面时并没有减少缺页次数。

FIFO淘汰算法产生Belady现象的原因在于它根本没有考虑进程执行的动态特征。

3. 最近最少使用淘汰算法

由于无法预知各个页面将来的访问情况,只能利用"最近的过去"作为"最远的将来"的近似。最近最少使用(Least Recently Used,LRU)淘汰算法的出发点是,如果某页很长时间未被访问,则它在最近一段时间内也不会被访问。因此,LRU淘汰算法每次选择最近最久未被访问的页淘汰。

例5-5 对于例5-2中给出的进程,采用LRU淘汰算法时,各页在内存中的变换如图5-24所示。

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
内存实页	0	0	0	3	3	3	3	0	0	0	1	1	1	1	4	4
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			5	5	5	4	4	4	3	3	3	3	3	3	3	3
缺页中断	√	√	√	√		√		√	√		√				√	

图5-24 LRU淘汰算法的置换图

由图5-24中可以看出,进程 p 在执行过程中,发生了6次缺页,比最佳淘汰算法多1次,比FIFO淘汰算法少3次,缺页率是6/16=37.5%。

LRU虽然是一种较好的淘汰算法,但是完全实现LRU淘汰算法的代价比较大。为了实现LRU淘汰算法,需要一个存放内存中所有页的链表,最近使用的页在表头,最久未使用的页在表尾。或给每一个页一个计数器 t,用来记录一个页上次被访问以来所经历的时间。当需淘汰一页时,选择 t 值最大的淘汰。由于存储器有较高的访问速度,在1ms内可能对某页连续访问成千上万次,因而,这一计数器需要足够大,并且有较快的访问速度。由此可见,为了实现LRU淘汰算法,需要一些特殊的硬件支持。下面是3种可行的方法。

1) 计数器法

这种方法要求系统中有一个64位的硬件计数器 C,它在每次执行完指令后自动加1。而进程的每个页表项必须有一个足以容纳这个计数器的值的域。在每次访问内存后,当前的 C 值存放到被访问的页的页表项中。当发生缺页时,操作系统检查页表中所有计数器的值,其中 C 值最小的对应页就是最近最少使用的页。显然这种方法除了硬件技术的支持,处理机还要花费时间去读写计数器的值,而且额外增加了页表的长度,将占用更多的内存空间。

2) 栈法

这种方法可利用一个特殊的栈来保存当前使用的各个页号。每当进程访问某页时,便将该页的页号从栈中移出,将它压入栈顶。因此,栈顶始终是最近新被访问的页号,而栈底则是最近最久未使用的页。对于例5-2中的引用序列,其访问过程如图5-25所示。

3) 寄存器法

为了记录某进程在内存中各页的访问情况,需为每个内存页面配置一个 n 位的移位寄存器,可表示为 $R=R_{n-1}\,R_{n-2}\cdots R_2\,R_1\,R_0$。当某一内存页面被访问时,将其对应的寄存器

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
栈顶	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
		0	2	5	3	2	4	2	0	3	2	1	3	2	3	4
栈底			0	2	5	3	3	4	2	0	3	2	1	1	2	2
置换				√		√		√	√		√				√	

图 5-25　LRU 淘汰算法的栈实现

的最高位 R_{n-1} 置为 1，每隔一定时间将寄存器 R 中的值右移一位。这样，当需要淘汰一页时，R 中值最小的一页就是最近最久未访问的页。例如，在图 5-26 中，某进程在内存中占用了 6 个内存页面，每一页对应的寄存器 R 的 $R_7 \sim R_0$ 位的值表示一段时间间隔内该页被访问的情况。这里，第 6 页的值最小，因此，第 6 页是最近最久未被访问的页，当发生缺页时，首先将它换出。

实页	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
1	0	1	0	1	0	0	1	0
2	1	0	1	0	1	1	0	0
3	0	0	0	0	0	1	0	0
4	0	1	1	0	1	0	1	1
5	1	1	0	1	0	1	1	0
6	0	0	0	0	0	0	1	1

图 5-26　LRU 淘汰算法的寄存器实现

4. LRU 的近似算法

因为，LRU 淘汰算法的实现需要的硬件太多，因此，在实际系统中，往往使用 LRU 的近似算法，比较常用的近似算法有以下两种。

(1) 时钟算法。

① 简单的时钟算法。系统将内存中的所有页按先进先出的顺序链接成一个类似钟面的环形链表，用一个表针指向最老的页。再为每页设置一个访问位 R，初始值为 0，表示该页没有被访问过。当某个页被访问时访问位置 $R=1$。当发生缺页中断时，算法首先检查表针指向的页。如果它的 R 位是 0，则淘汰这个页，并把新的页插入这个位置，然后把表针前移一个位置；如果 R 位是 1，则设置 R 为 0，并把表针前移一个位置。重复这个过程直到找到一个 R 位为 0 的页为止。这个算法，类似一个时钟的走动，因此称之为时钟算法。该算法每次都是淘汰最近一段时间内未被访问的页，因此该算法又被称为最近未使用算法(Not Recently Used，NUR)。

② 改进的时钟算法。如果一个页在访问的过程中被修改过，则需要将被修改过的页写入外存；如果没有被修改过，则不需要将该页写入外存。因此，淘汰被修改过的页比淘汰没有被修改过的页所需的代价要大。在淘汰时，应尽可能地选择没有被访问过并且没有被修改过的页淘汰，这就是改进的时钟算法。该算法需要在页表中增设两个状态位：访问位 R 和修改位 M。R 表示对应的页在最近一段时间内是否被访问过，M 表示对应的页是否被修改过。当一个进程启动时，所有的页对应的 M 与 R 都由系统设置为 0。当某一页被访问时，它的访问位被设置为 1，如果这个页被修改过，则将修改位 M 设置为 1。这样，内存中可能有四种类型的页面。

第1类：$R=0$，$M=0$，表示该页最近没有被访问过，没有也未被修改过，是最佳的淘汰页。

第2类：$R=0$，$M=1$，表示该页最近没有被访问过，被修改过。

第3类：$R=1$，$M=0$，表示该页最近被访问过，没有被修改过。

第4类：$R=1$，$M=1$，表示该页被访问过，也被修改过。

当发生缺页时执行过程分为三步。

第一步：从指针指向的位置开始，循环扫描队列，寻找 $R=0$、$M=0$ 的第一类页，找到后立即置换。注意，第一次扫描期间不改变访问位 R。

第二步：如果第一步失败，则开始第二轮扫描，寻找 $R=0$ 且 $M=1$ 的第二类页，找到后立即置换，在第二轮扫描中，将所有扫描过的 R 都设置为0。

第三步：如果第二步也失败，则返回指针开始位置，然后重复第一步，必要时再重复第二步，此时必然能找到淘汰页。

该算法的优点是，易于实现和具有高效性。虽然它的性能不是最好的，但在实际中常常是够用的。

(2) 最不经常使用(Not Frequently Used，NFU)算法。该算法选择到当前时间为止被访问次数最少的那一页淘汰。这只需在页表中给每一页增设一个访问计数器即可实现。每当该页被访问时，访问计数器加1。而发生缺页时，则淘汰计数器值最小的那一页，并将所有的计数器清零。

NFU算法的问题是，如果有两个页的计数器都是0，只能随机选择一页淘汰，而在实际中，有可能一个页上次被访问是在9个周期以前，另一个页在1000个周期以前，而在NFU算法中却不能反映这个差别，结果是淘汰出去的可能是有用的页而不是不再使用的页。

5.4.2 抖动现象与工作集

在动态页式存储管理中，有可能出现这样的现象：对于刚刚被淘汰出去的页，进程可能马上又要访问它，故又需将它调入内存，因无空闲内存页面而又要淘汰另一页，而后者很可能是即将被访问的页。于是造成系统需要花费大量的时间忙于进行这种频繁的页面调换，从而无法完成用户所要求的工作，这称为抖动现象。那么，抖动现象的发生与什么有关？如何才能防止抖动现象的发生呢？

实验表明，抖动与内存中并发的进程数以及系统分配给每个进程的实页数有关。

考察单CPU的多道程序系统，CPU的利用率与内存中并发进程数的关系图如图5-27所示。

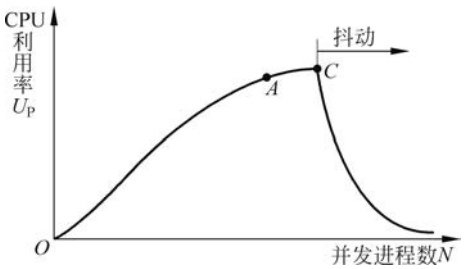


图5-27 CPU的利用率与内存中并发进程数的关系图

由图5-27可知，开始时，CPU的利用率 U_P 随着并发进程数的增加而增加。但是，当 N 达到一定的值时，U_P 将达到峰值域(图中 $A\sim C$)；如果 N 继续增加，将引起系统发生抖动而使 U_P 急剧下降，即系统内的并发进程数有一个临界值，一旦超过这个临界值就会发生抖动。造成这种异常现象的主要原因是过度地使用了内存以及内存分配不合理。因为每个进程都需要占用

一定的内存空间，随着进程数的增加，各个进程所分得的实页数就会减少，缺页现象随之增多，从而导致频繁的页面交换，进程经常因需要调页而等待，CPU 的大量时间都消耗在进程调度和决定页面淘汰上，故 CPU 的有效利用率很低。当然，如果内存中并发进程数过少，CPU 利用率也不高。显然，最理想的情况是使 $A \leqslant U_P(N) \leqslant C$，那么，并发进程数 N 如何确定呢？

并发进程数 N 的取值与系统为进程分配内存空间的大小有关。在动态页式管理系统中，从原则上说，为进程分配的实页数越多，进程的缺页率就越小，但并发进程数也就越少，从而 CPU 利用率下降；而为进程分配的实页数过少，进程的缺页率就会增大，可能导致系统发生抖动。图 5-28 给出了进程的缺页率 R_P 与进程占用实页数的关系。从图 5-28 可见，如果一个进程占有的实页数能满足 $M_j \leqslant M \leqslant M_k$，那么该进程就获得了有效运行所需的足够内存空间，缺页率将保持在合理的范围内。如果每个进程的实页数都能满足 $M_j \leqslant M \leqslant M_k$，则内存中的并发进程数就是合理的。

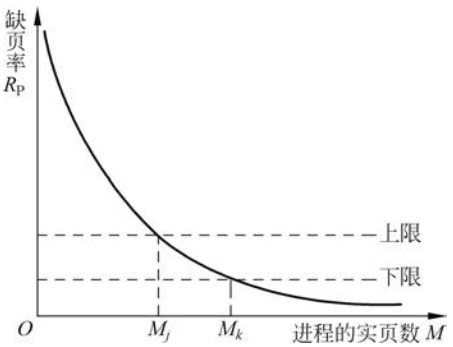


图 5-28　进程的缺页率 R_P 与进程占用实页数的关系

总之，为防止系统抖动现象的发生，根本的办法是要控制系统中的并发进程数以及为各个进程分配合理的实页数，既要使每个进程都有足够的内存空间，又要使系统中的并发进程数接近最佳值。

防止抖动现象的发生通常有两种方法。

一种是选择好的淘汰算法，以减少缺页次数。好的淘汰算法应当在置换页面时尽可能选择暂时不用或永久不用的页，使内存中存放的是一个进程一段时间内所需的页。

另一种是扩大工作集。所谓工作集，是指进程在某个时间段里要访问的页的集合。如果能够预知进程在某段时间的工作集，并在此之前把该集合调入内存，至该段时间终了时，再将其在下一时间段不需要访问的那些页换出内存，就可以减少页的交换。但是一个进程在某段时间内的工作集是很难确定的。事实上，由于各进程所包含的程序段多少以及选用的淘汰算法等不一样，工作集的选择也不一样。通常采用的方法是，当进行页置换时，把缺页的进程锁住，不让其换出，而调入的页占据那些暂时得不到执行的进程所占据的内存区域，从而扩大缺页进程的工作集，为它分配更多的内存实页。

5.5　段式存储管理

在前面介绍的几种存储管理技术中，用户的逻辑地址空间被连成一个一维（线性）空间，这难于满足将程序按其逻辑结构划分的需求，从而不便于用户的程序和数据的共享；另外，在实际应用中，一些程序段和数据段在使用过程中会不断增长，前述的几种存储管理技术都不适用于这种动态的增长。于是人们提出按程序的逻辑单位段来分配内存的段式存储管理方案，它同样有静态分配和动态分配两种方式。

5.5.1 静态段式存储管理

1. 基本思想

为了方便程序的设计,用户按程序的内容或过程(函数)关系把自己的程序分段,每段都有自己的名字。当一个进程建立时,以段为单位分配内存,一个段占用一个连续的内存区域。一个进程的各个段可以存放在不连续的区域内。只有一个进程所有的段都可得到足够的内存空间时,才可为其分配内存空间,否则拒绝分配。

2. 分段地址空间和地址结构

在段式存储管理中,一个用户作业或程序的地址空间被划分成若干段,每个段定义了一组逻辑信息,每段的长度由相应的逻辑信息组成的大小决定,且可动态增长。例如,一个程序可有主程序段、子程序段、数据段与栈段等。由于一个程序被分成多个段,因此,整个程序的地址空间是二维的。例如,一个程序可以有主程序段、X段、A段、B段,如图5-29所示。

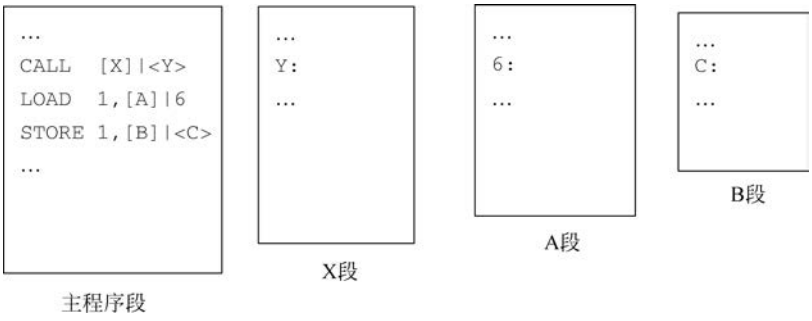


图5-29 分段的地址空间

一个程序中每段都有自己的名字,用户可通过段名和段内符号地址访问其他的段。

例如:

```
CALL  [X]|<Y>       ;转向段名为X的子程序的入口点Y
LOAD  1,[A]|6       ;将段名为A的数组中第6个元素的值读到寄存器1中
STORE 1,[B]|<C>     ;将寄存器1的内容存入段名为B、段内地址为C的单元中
```

程序经过编译和装配后,每段的段名被译成唯一的段号,每个段是一个首地址为零的连续的一维地址空间,CPU访问内存单元的指令中的段名和段内符号地址经编译后分别变成段号和段内相对地址。例如,“CALL [X]|<Y>”可编译成“CALL 3,120”,其中3是段号,120是段内相对地址。因此,段式管理系统中,程序的逻辑地址由两部分组成:段号和段内相对地址。图5-30给出了一个段式管理中的逻辑地址结构示例,其中计算机CPU的有效地址为24位,段号(s)占用了8位,段内相对地址(w)占用了16位。

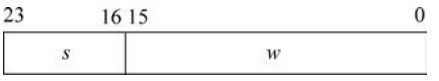


图5-30 段式管理中的逻辑地址结构示例

一旦段号和段内相对地址的长度确定后,一个进程允许的最多段数和段的最大长度也就限定了。上述结构表明,一个程序允许有256个段,一个段的最大长度为64KB。

3. 数据结构

在静态段式存储管理中,为了实现内存的分配和回收以及逻辑地址到物理地址的转换,需要建立以下几种数据结构。

(1) 段表。和页式存储管理类似,为了记录每个段在内存中的存放位置,系统为每个进程建立一张段表。每个段在表中占有一个表项,其中包括该段的段号、段的起始地址和段的长度等,如图 5-31 所示。

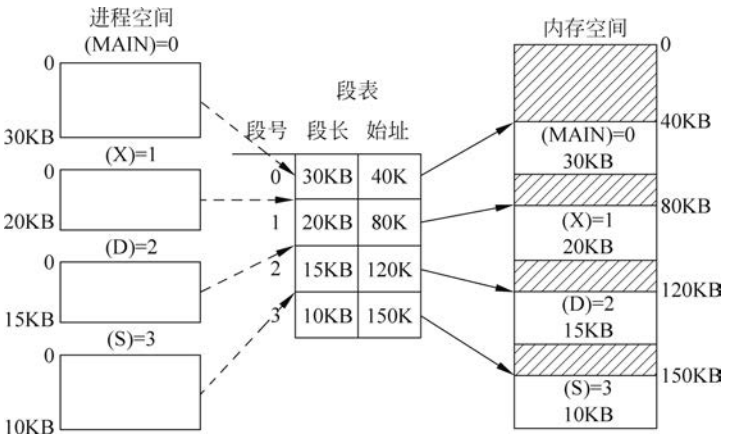


图 5-31 段、段表及段在内存空间占用情况

(2) 内存空闲区表(或链)。因为段式存储管理以段为单位,一段占有一个连续的区域,因此,与分区式存储管理类似,系统中有一张空闲区表,用于记录当前空闲区的情况。每一个连续的空闲区在其中占有一个表项,包括空闲区的大小和起始位置。

(3) 请求表。用于记录系统中每个进程的段表始址和段表长度。请求表通常由每个进程的对应的 PCB 表项组成。

4. 地址变换

为了实现地址变换的功能,系统中设置了一个段表控制寄存器,当一个进程被选中执行时,其对应的段表起始地址和段表的长度被装入段表控制寄存器。图 5-32 给出了段式存储管理的地址变换过程。每当 CPU 访问逻辑地址(s,w)时,由地址变换机构实现地址的变换过程。首先由段表寄存器中的段表始址得到该段的段表始址,然后以段号为索引,查找段

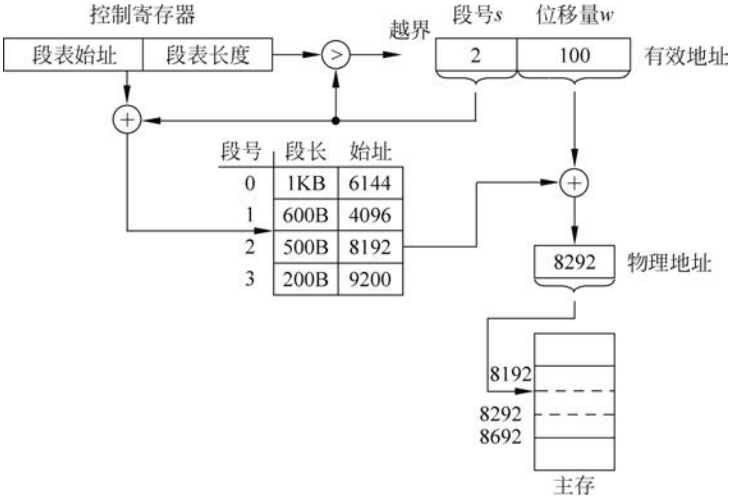


图 5-32 段式存储管理的地址变换过程

表,找到对应表中的段内存始址SD,则逻辑地址(s,w)对应的物理地址为SD+w。由此实现了段式逻辑二维地址到一维的物理地址的转换。

与页式存储管理相同,段式存储管理的地址变换过程也使得访问一条指令需经过二次内存的访问。即首先需访问段表,得到对应的段在内存的始址,然后才是对物理地址的访问。为了提高访问速度,解决的方法与分页系统类似,在系统中增设一个快表,用于保存最近常用的段的表项。由于一般情况下段比页大,因而段表的表项数目比页表的表项数目少,其所需的联想寄存器也比较少,可以显著地减少存取数据的时间。

5. 内存分配和释放

在分段分配方案中,每段分配一个连续的内存区域,如图5-31所示。由于各段长度不等,因而各段所占的存储区的大小不等,同一进程的各段之间不要求连续。因而,段式存储管理的内存分配和释放算法与可变式分区的管理方法类似,系统由相应的数据结构来管理内存空闲区,分配算法可采用首次适应算法、最佳适应算法或最坏适应算法等,分区的回收需要进行分区合并。不同的是,可变分区以进程为单位,而静态段式存储管理是在内存中有可用空间能满足进程总容量的前提下,以段为单位来分配分区的。碎片问题同样是不可避免的,需要在适当时进行碎片拼接。

静态段式存储管理中,进程的大小受内存空闲区总容量的限制。因此,同样不能实现内存的扩充。

5.5.2 动态段式存储管理

1. 基本思想

在动态段式存储管理中,段的内存分配与释放是在进程运行过程中动态进行的。当一个进程准备执行时,段式存储管理程序只调入一个进程的若干段,便可启动进程运行。进程在运行中根据需要随时申请调入新段和释放老段。空闲区的分配和释放算法与静态段式存储管理类似,所不同的是,当访问的段不在内存中时,由段式中断机构将所需的段调入内存。当内存不足时,需要从内存中选择暂时不需要的段淘汰,以便调入新的段。

与动态页式存储管理类似,为实现从逻辑上为用户提供一个比实际更大的虚拟存储空间,同样需要一定的硬件和相应的软件支持,如扩充的段表、缺段中断机构、地址变换机构等。

2. 段表

在动态段式存储管理中,系统为了知道所访问的段是否在内存以及访问的段的状态等,以静态段式存储管理为基础,对段表进行了扩充,如下所示。

段号	内存始址	段长	驻留位	访问位	修改位	外存地址	存取方式	增补位

其中各字段的说明如下。

(1) 驻留位:用于指示该段是否在内存。

(2) 访问位:用于记录本段在一段时间内被访问的次数,或记录本段最近有多长时间未被访问,供段淘汰时参考。

(3) 修改位:用于表示该段在调入内存后是否被修改过。

(4) 外存地址:用于指出该段在外存中的地址,以便系统从外存中调入该段。

(5) 存取方式：用于表示本段的存取权限，以限制未经允许的用户访问本段。

(6) 增补位：用于表示本段在执行的过程中是否增长过。

3. 缺段中断处理

在动态分段系统中，当 CPU 所要访问的指令或数据不在内存中，将由缺段中断机构产生缺段中断信号，CPU 接收到这个信号时进入缺段中断处理过程。图 5-33 给出了缺段中断处理的过程。

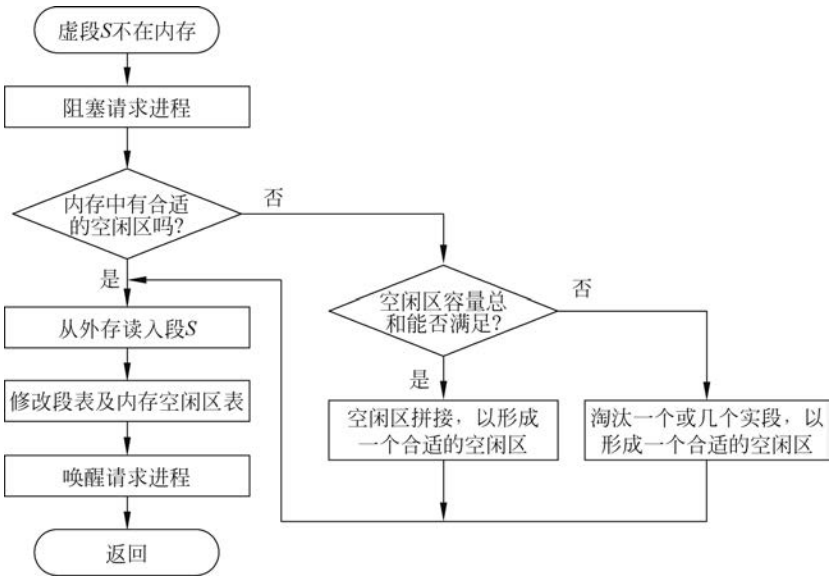


图 5-33　缺段中断处理的过程

在新段调入时，分为以下三种情况。

(1) 如果内存中有一个足够大的连续的空闲区可以容纳新调入的段，修改相应的数据结构，如段表、空闲区表等，完成新段的调入。

(2) 如果内存中没有一个大于或等于新段的连续的空闲区，则查找空闲区表，如果内存中一部分或全部空闲区总和足够容纳新段，则进行空闲区的合并以形成一个长度不小于新段的空闲区，修改相应的数据结构，调入新段。

(3) 当内存中所有的空闲区的总和不足以容纳新段时，则需选择一个或几个暂时不需要访问的段淘汰。

在段的淘汰过程中，如果调入的是一个小段，则使用 FIFO 或 LRU 淘汰算法决定一个能满足需要的实段淘汰；如果调入一个大段，可能没有一个实段能满足需求，这就需要淘汰几个段。如果完全按照 FIFO 或 LRU 淘汰算法的原则选择淘汰段，可能由于所选择的段不是连续的，需要进行空闲区的拼接；如果从空间因素的需求来选择淘汰段，则选择能满足需求的最小数目的相邻的几个段淘汰，但是这种策略可能存在一种危险，那就是刚刚被淘汰的段有可能立即再次被访问。可见，动态段式存储管理的段淘汰策略比动态页式存储管理更复杂，它要考虑空间的需求、最近将来被访问的可能性以及内存拼接等诸多因素。通常，对一种具体的系统淘汰策略主要侧重于考虑上述某个因素。

5.5.3 分段和分页的主要区别

由前所述,分页和分段有许多的相似之处,例如,都采用离散式的分配方式,都可实现虚拟内存的扩充技术,地址变换也很相似。但是二者在管理和用户的角度上有很大的差别,主要表现在以下4方面。

(1) 段是面向用户的,页是面向系统的。段是信息的逻辑单位,分段是出于用户的需求,对用户是可见的;页是信息的物理单位,分页是用户不可见的,只是为了便于内存的管理,是系统的需求。

(2) 页的大小是固定的,由系统决定;段的大小不固定,由用户决定。

(3) 从用户的角度看,分页系统的用户程序空间是一维连续的空间;段的地址空间是二维的,由段名和段内相对地址组成。

(4) 从管理的角度看,分页系统的二维地址是在地址变换过程中由系统的硬件机构实现的,对用户是透明的;分段系统的二维地址是在地址变换过程中由用户提供的。因而,页内没有地址越界问题,而段内的相对地址则存在地址越界问题。

5.5.4 段的信息共享

在多道程序环境下,有许多的系统程序,如编译程序、编辑程序、标准的库程序等,经常被若干进程同时访问,如果系统给每个进程的地址空间中都保留同一程序的一份副本,则对内存存在大量的浪费。而段式存储管理较好地解决了这个问题。

在页式存储管理和段式存储管理中,都可实现多个进程共享同一内存块里的程序和数据。对于段式存储管理来说,被共享的部分可以是一个独立的段,物理上是一段,逻辑上也是一段,因此更易于实现,只需将每个进程的段表中对应的共享段始址指向相同的始址即可,如图5-34(a)所示。对页式存储管理来说,必须保证被共享的程序或数据占用整数据块,以便与非共享部分分开。如果被共享的部分占用若干块,那么在各共享者的页表中就有若干表项,它们所指向的页面号相同,如图5-34(b)所示。在实际应用中,很难保证被共享的程序和数据占用整数据块,导致部分非共享的代码或数据被放入共享页,如图5-34(b)中的阴影部分所示。

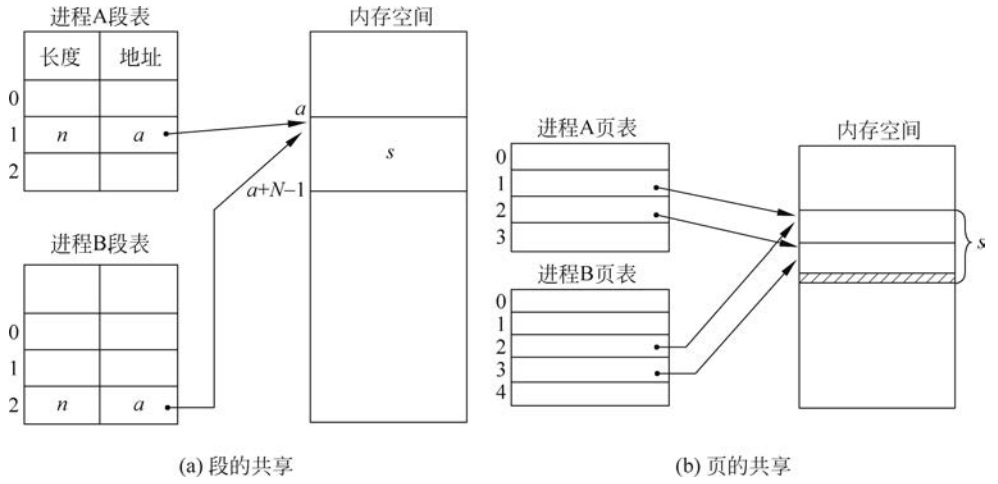


图5-34 内存信息的共享

为了保证各个进程访问的共享段代码相同，共享段必须是纯过程。但事实上，大多数代码在执行时都有可能对其处理的数据进行改变。为此，各共享进程都必须在自己的存储区域分别有一套数据。这样，进程在执行时只对自己的数据区中的内容进行修改，并不改变共享代码。为了保证共享代码不被修改，每个共享段的存取控制都设置为不可写。

为了协调分段的共享，系统中可设置一张共享段表，每一个共享段在该表中占有一个表项。每一个表项指出该段是否在内存、调用此段的进程名、进程号等情况，如图 5-35 所示。

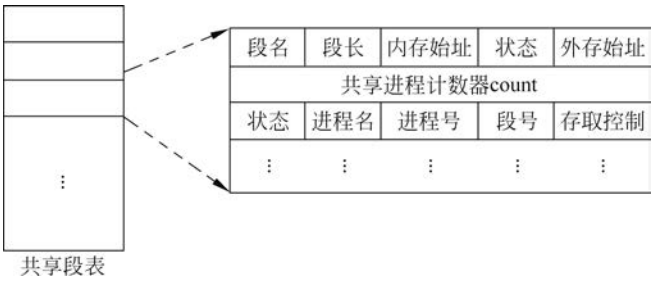


图 5-35　共享段表项

当一个共享段首次被系统中一个进程调用时，由系统为该段分配一块内存区域，同时将该区的始址填入请求进程的段表的相应项中，还在共享段表中增设一组表目，填写有关的数据。例如，共享进程计数器 count ＝1，该段的段名、段长、内存始址，调用进程的进程名、进程号、该段在调用进程中使用的段号以及存取控制等。此后，当又有进程调用该共享段时，系统查找共享段表，看该段是否已在内存中，如果在，那么需在调用进程的段表中填写该段的起地址；在共享段表该共享段对应的表项中执行 count ＝ count ＋1，填写调用进程的名称、进程号、存取控制等。当某一进程不再需要某一共享段时，执行 count ＝ count－1 操作。若 count ＝ 0，表明此时已没有进程在使用该段，则由系统收回该段的内存区域，取消该段在共享段中所对应的表项。

在动态段式存储管理中，当某一个共享段被调出内存后，必须在共享该段的每个进程相应的段表目中设置其状态为“不在内存”。相反，当一个共享段由于某个进程的调入重新装入内存后，共享该段的所有的进程对应的段表目需要重新调整。为了简化调整，将共享段的内存始址保存在共享段表中，而共享该段的所有进程的段表相应的表目总是指向这个共享段的表目。这样，当一个共享段被装入内存或移出内存时，只需修改共享段表的对应表目。

5.5.5　段的静态链接与动态链接

通常来说，一个作业或一个进程由若干程序模块组成，而这些模块必须经过编译或汇编，得到一组目标模块，再由链接程序将这些目标模块链接起来，装入内存执行。根据链接时间的不同，又可将链接分为两种：静态链接和动态链接。

1. 静态链接

静态链接是指程序在执行之前，由链接装配程序将该程序的所有目标模块进行链接和相对地址重定位，使之成为一个可运行的目标程序。

这种链接既可以在程序装入之前完成，又可以在程序装入的时候进行。对于分区和页

式存储管理方案,程序装入模块必须是一个一维的地址空间,因而链接时需对目标模块中所有的相对地址进行修改,如图5-36(a)中所示,一个程序经过编译有3个目标模块,链接后如图5-36(b)所示。对于静态段式存储管理,因为一个程序有多少个段是固定的,所以链接时由装配模块给每一段一个段号,据此将目标模块中的所有段名修改为相应的段号,如图5-36(c)所示。

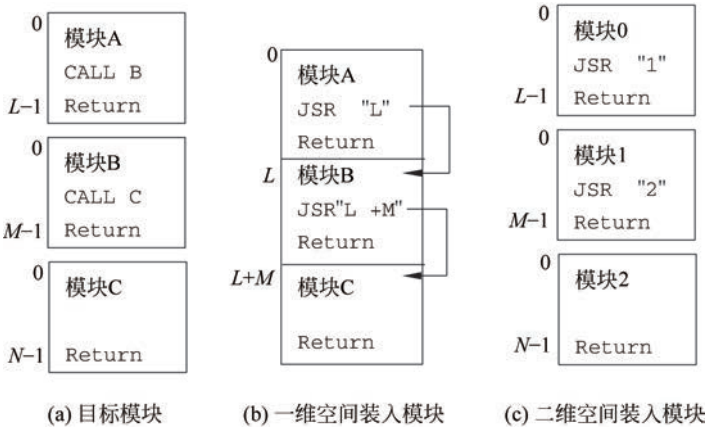


图5-36 静态链接示例

静态链接比较简单,但是也存在一些缺点。例如,在链接时需对目标模块中的一些相对的地址进行修改,因而链接时间长,有时链接过程中所花费的时间有可能比程序的执行时间还长;链接好的程序段在运行时可能根本不会用到(例如,程序错误处理子程序,在程序正确的情况下是不用的),但程序事先无法确定哪些模块会用到,只能将所有的模块装入内存,因而造成了时间和空间上的浪费。

2. 动态链接

动态链接是指程序在执行过程中需要某一段时,再将该段从外存调入内存,把它与有关的段链接在一起。这样,凡是在程序执行过程中不会用到的段都不会调入内存,也不会链接到装入模块上,因而,不仅能加快程序的装入过程,也可节省大量的内存空间。

对于动态段式存储管理方案,每个段是独立的程序模块,又有各自的段名,且可以动态分配内存空间,因而易于实现动态链接。下面通过一个例子给出MULTICS系统实现动态链接的一种方法。

为了实现动态链接,在MULTIC系统中需要有进行链接中断的硬件机构及间接寻址功能。在程序汇编或编译时,当遇到访问外段的指令时,将其编译成一条间接寻址指令,即,将访问的地址指向一个间接地址,这个间接地址被称为间接字。间接字的形式如图5-37所示。

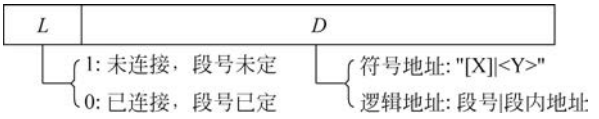


图5-37 间接字

当$L=1$时,表示要访问的段还没有链接好,此时,D为符号地址。由于符号地址可能很长,D中存放不下,所以D中实际上存放了指向该符号串的首地址。当$L=0$时,表示此段已被链接,此时,D中存放的是要访问的逻辑地址,由段号和段内地址组成。初始时$L=1$。

例 5-6 如图 5-38(a)所示，在 M 段中遇到访问外段指令“LOAD 1,[X]|<Y>”时，将其编译成“LOAD* 1,1|200”，如图 5-38(b)所示。在此，假设 M 段已链接好并分得段号为 1，X 段未链接好($L=1$)，保存在文件系统中，指令“LOAD 1,[X]|<Y>”的间接字的直接地址为 200。* 表示间接寻址，7 表示字符串“[X]|<Y>”的长度。

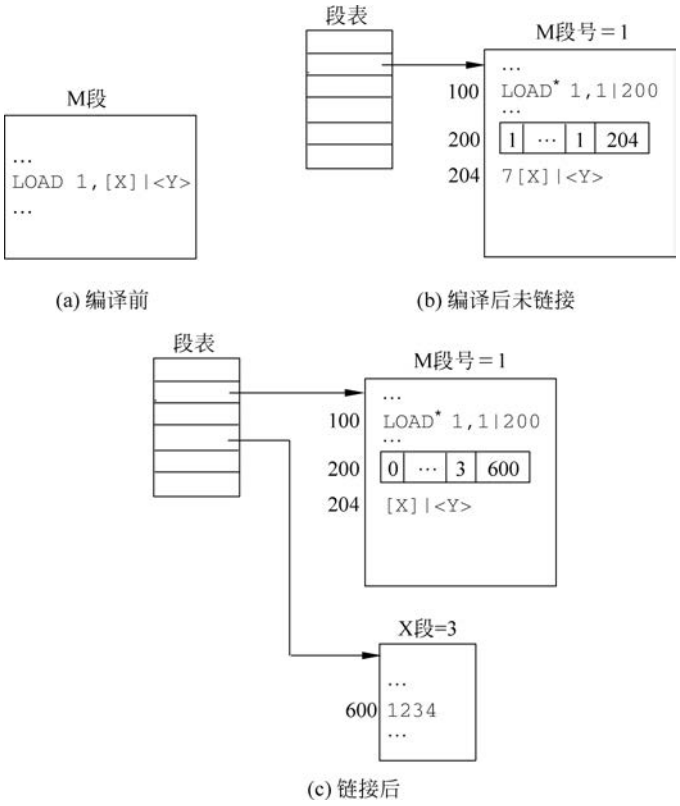


图 5-38 动态链接示例

当程序段 M 执行到“LOAD* 1,1|200”时，由于 200 单元处的间接字中的链接标志 $L=1$，于是发生链接中断，转入链接中断处理程序。它首先根据间接字中存放的符号地址(此处为 204)，找到符号[X]|<Y>，从外存中将 X 段调入内存，并且分配一个段号(此处为 3)；然后再根据 Y 找到段内地址(这里是 600)；最后，修改间接字中的值，置 $L=0$，$D=(3|600)$，至此链接完成，如图 5-38(c)所示。链接完成后，继续执行原 1 号段中被中断的指令“LOAD* 1,1|200”。此后，在 1 号段中再次调用该指令时不再发生链接中断。

由于动态链接过程修改了链接的间接字，与共享段必须是纯代码相矛盾。因此，在动态链接中，为了保证段的共享及纯代码，一种解决的办法是将代码段分为纯段和杂段两部分，即将链接的间接字等修改的内容放在杂段中，而将其他在执行过程中不会改变的内容放在纯段中。纯段可共享，杂段不共享。

5.5.6 段式存储管理的内存保护

段式存储管理的内存保护方法主要有以下两种。

(1) 地址越界保护法。可在每个进程的段表项中设置每一段的段长。当进行地址变换

时,地址变换机构首先依据段表寄存器中的段表长度,判断段号是否大于段表的长度。如果超出段的长度,则发生越界中断;如果没有超出段的长度,则找到对应的段表项,将段内相对地址与其对应的段的长度比较。这样,每个进程被限制在自己的地址空间中运行,因而不会破坏另一个进程的地址空间。

这里要说明的是,在允许段动态增长的系统中,段内相对地址可以大于段的长度。一个段是否允许动态增长,由段表的增补位来说明。

(2) 存取控制保护法。在段表的表目中,除了指明本段的段长外,还增加了存取方式控制项。这种段的保护方式,对非共享段来说,主要用来指示程序设计的错误。例如,对一个过程段应只能执行,如果企图读取一个过程段,则发生中断保护。而对共享段来说,这种保护则更为重要。例如,被共享的纯段应禁止任何进程对它进行修改。这样既可保证信息的安全,又可满足运行的需要。

5.5.7 段式存储管理的优缺点

1. 优点

段式存储管理有以下优点。

(1) 便于信息的共享和保护。由于在分段管理中,每个程序按信息的逻辑结构构成各自独立的分段,因而便于对完整独立的信息的共享和保护。

(2) 实现了内存的扩充。动态段式存储管理与页式存储管理一样,可使进程的大小不受内存大小的限制,从而实现了内外存统一管理的虚存。

(3) 便于信息的变化处理。在实际应用中,有些数据段需要不断吸取和增加新的数据,因而需要对段进行动态的增长。而在段式存储管理中,以段为单位来分配内存空间,因而在一个分段的后面增加新信息,不会影响其他部分。

(4) 便于实现动态链接。由于分段管理的地址空间是二维的,且每一个分段是一个具有独立功能的程序段,因而在进程运行过程中,可以在调用一个程序段或一个数据段时再去进行动态链接。

2. 缺点

段式存储管理有以下缺点。

(1) 增加了计算机成本。地址变换需硬件支持;缺段中断需花费处理机时间;表格的建立和管理需花费处理机时间,且占用一定的空间。

(2) 存在碎片问题。类似于分区管理,内存中存在分区间的碎片。为了满足段的动态增长和减少碎片,需采用拼接技术。

(3) 段的长度受内存可用空间大小的限制。

(4) 与页式存储管理类似,淘汰算法选择不当时可能产生抖动现象。

5.6 段页式存储管理

前面介绍的几种存储管理方案各有所长。段式存储管理大大方便了用户,也便于信息共享和信息的动态增加,但存在内存的碎片问题。页式存储管理则大大提高了主存的利用率,但不便于信息的共享。因此,结合二者的优点的一种新的存储管理方案被提了出来,这

就是段页式存储管理。

5.6.1 实现原理

1. 基本思想

段页式存储管理对用户来说与段式存储管理相同，一个进程仍由用户按照程序的逻辑信息划分成不同的段。经编译和链接后的程序，每个段有唯一的段号。因而，用户地址空间仍是一个二维的逻辑虚地址空间。而对于系统来说，段页式存储管理则与页式存储管理相同，内存空间被划分成若干大小相同的内存块，与此对应，每个进程的每个段被划分成若干与内存块大小相同的页，以页为单位来分配内存空间，一个页占用一个内存块。这样，一个进程中的一个段的信息可以存放在块号不连续的内存块中，分段的大小不受内存大小的限制。

2. 数据结构

在段页式存储管理系统中，为了实现内存分配与释放、缺页处理、地址变换等，系统为每一个进程建立一张段表，每个段建立一张页表。页表表项与页式存储管理的页表表项相同，有指向页对应的存储块号以及缺页处理和页面保护等表项。段表表项与段式存储管理的段表表项类似，不同的是，原来在段式存储管理的段表中的内存地址现在变为指向与段对应的页表的起始地址。段页式存储管理中段表、页表以及内存的关系如图 5-39 所示。

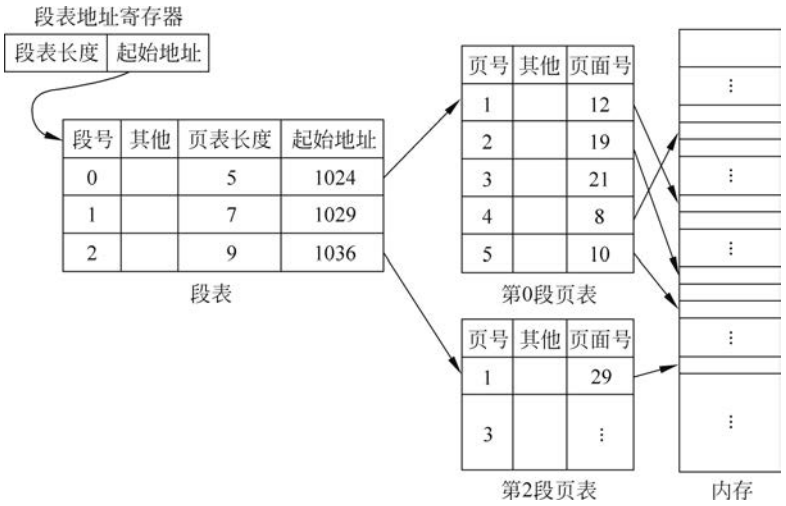


图 5-39 段页式存储管理中段表、页表以及内存的关系

与段式存储管理类似，每个进程有对应的 PCB 表项用于记录该进程对应的段表始址和段表长度，系统中有用于记录内存空闲块的空闲区链表。

3. 地址结构与地址变换

在段页式存储管理中，程序的分段由程序员决定，因此，对用户来说，用户的地址空间仍然由段号和段内相对地址(s,w)组成。而对于系统来说，以页为单位来分配内存空间，地址变换机构将根据页的大小把段内相对地址解释为页号和页内地址。因此，段页式存储管理系统中，进程的地址空间由段号、页号、页内地址(s,p,d)这三部分组成，如图 5-40 所示。

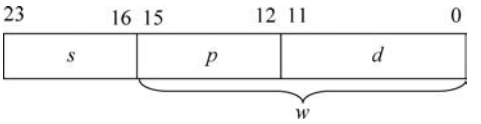


图 5-40　段页式管理地址结构示例

为了实现地址的变换,在段页式系统中设置了段表寄存器,用于存放当前运行的进程的段表起始地址和段表长度。当 CPU 访问一个逻辑地址(s,w)时,系统首先自动将地址分成 3 部分(s,p,d),然后将段号 s 与段表寄存器中的段表长度进行比较。若段号大于段表长度,则越界,否则,根据段号找到对应的段的页表始址,再利用 p 找到对应的页的内存块号(页面号),最后,将页内地址与找到的块号合并,形成与逻辑地址对应的物理地址。

显然,在段页式存储管理中,访问内存的指令或数据需要访问内存 3 次,因此,为了提高访内的速度,在系统中设置联想寄存器比段式和页式存储管理更为重要。在联想寄存器中存放当前最常用的段号、页号和与其对应的内存页面号。图 5-41 说明了采用这种方案的地址变换过程。

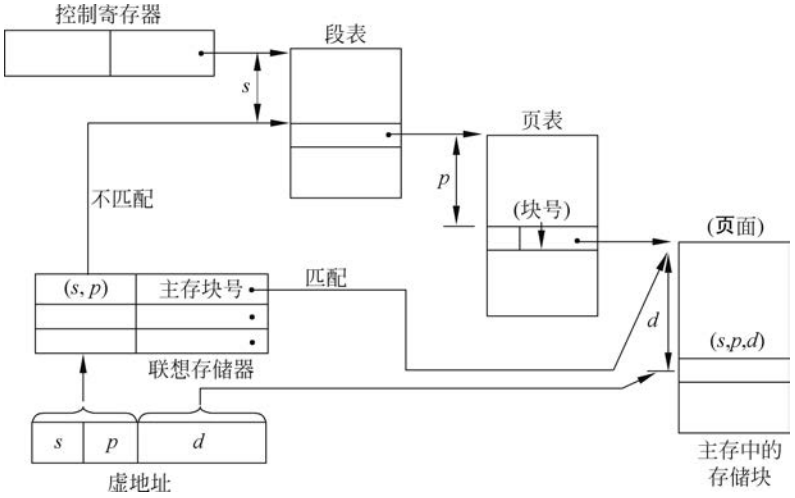


图 5-41　段页式管理的地址变换过程

5.6.2　段页式存储管理的其他问题

有关段页式存储管理中的内存分配与释放、存储保护、缺页与缺段处理等,对在段式存储管理和页式存储管理中提到的方法稍作修改便可适用,在此不再赘述。

因为段页式存储管理是页式和段式存储管理的结合,具有二者的优点。但是由于它增加了软件管理,系统的管理开销也随之增加了。所需的硬件支持和内存占用也增加了。而碎片问题与页式存储管理一样存在,且更为严重。另外,如果不采用联想寄存器的方式提高 CPU 的访问速度,将会大大降低系统的执行速度。

5.7　Linux 存储管理

Linux 系统采用了虚拟的内存管理机制,即交换和请求分页存储管理技术。这样,当进程运行时,不必把整个进程的映像都放在内存中,而只需在内存中保留当前用到的那一部分

页面。当进程访问到某些尚未装入内存的页时，就由核心把这些页装入内存。这种策略使进程的虚拟地址空间映射到内存的物理空间时具有更大的灵活性，通常允许进程的大小可大于可用内存的总量，并允许更多进程同时在内存中执行。

5.7.1 进程虚拟内存空间的管理

1. 地址空间

Linux 系统中，进程的地址空间是由虚拟内存(Virtual Memory，VM)组成的，对当前进程而言，只有属于它的虚拟内存是可见的。Linux 把进程的虚拟内存分成两部分：内核区和用户区。操作系统内核的代码和数据等被映射到内核区，进程的可执行映像(代码和数据)被映射到虚拟内存的用户区。由体系结构决定，每一个进程都有一个独立的 32 位或 64 位的连续的地址空间。进程虚拟内存的用户区分成代码段、数据段、堆栈以及进程运行的环境变量、参数传递区域等。进程在运行中还必须得到操作系统的支持，进程的虚拟内存中还包含操作系统内核。每个虚存区域具有对相关进程的相关权限，进程只能访问具有相关权限的区域。

内核使用 mm_struct 结构体来表示进程的地址空间，该结构体包含了进程地址空间有关的全部信息。

mm_struct 结构体首地址在任务结构体 task_struct 成员项 mm 中。

```
struct mm_struct   * mm;
struct mm_struct {
    int count;                                  //引用该区域的进程个数
    pgd_t * pgd;                                //为指向进程页表的指针
    unsigned long context;                      //进程上下文的地址
    …                                           //代码段、数据段、堆栈的首尾地址等
    struct vm_area_struct * mmap;               //内存区域链表
    struct vm_area_struct * mmap_avl;           //VM 形成的红黑树
    struct semaphore mmap_sem;                  //虚存区的信号量
};
```

2. 进程的虚存区域

一个虚存区域是虚拟内存空间中一个连续的区域，在这个区域中的信息具有相同的操作和访问特性。每个虚存区域用一个 vm_area_struct 结构体进行描述。

```
struct vm_area_struct
{
    struct mm_struct * vm_mm;                   //指向进程的 mm_struct 结构体
    unsigned long vm_start;                     //虚存区域的开始地址
    unsigned long vm_end;                       //虚存区域的终止地址
    pgprot_t vm_page_prot;                      //虚存区域的页面的保护特性
    unsigned short vm_flags;                    //虚存区域的操作特性
    …                                           //AVL 结构
    struct vm_operations_struct * vm_ops;       //相关操作表
    unsigned long vm_offset;                    //文件起始位置的偏移量
    struct inode * vm_inode;                    //被映射的文件
};
```

进程虚存空间的结构如图 5-42 所示。

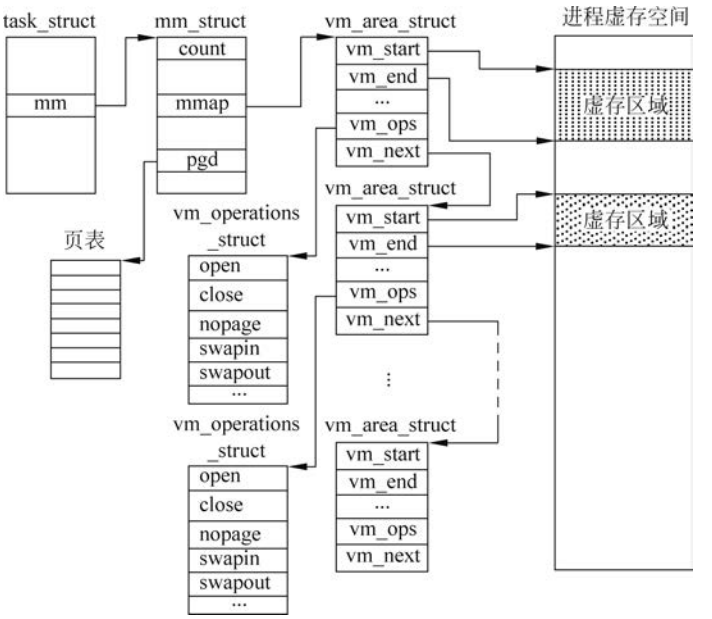


图 5-42　进程虚存空间的结构

3. 虚存空间的映射和虚存区域的建立

在虚拟存储技术中,用户的代码和数据(可执行映像)等并不是完整地装入物理内存,而是全部映射到虚拟内存空间。在进程需要访问内存时,在虚拟内存中"找到"要访问的程序代码和数据等,系统再把虚拟空间的地址转换成物理内存的物理地址。

Linux 使用 do_mmap()函数完成可执行映像向虚存区域的映射,由它建立有关的虚存区域。

```
unsigned long do_mmap(struct file * file, unsigned long addr,
                      unsigned long len,unsigned long prot,
                      unsigned long flags, unsigned long off)
```

其中,addr 是虚存区域在虚拟内存空间的开始地址; len 是这个虚存区域的长度; file 是指向该文件结构体的指针; off 是相对于文件起始位置的偏移量; 若 file 为 NULL,称为匿名映射(anonymous mapping); prot 指定了虚存区域的访问特性; flag 指定了虚存区域的属性。

5.7.2　Linux 的分页式存储管理

1. 物理内存的页面管理

Linux 对物理内存空间按照分页方式进行管理,把物理内存划分成大小相同的物理页面。大多数的 32 位体系支持 4KB 的页面,64 位体系结构支持 8KB 的页面。

Linux 内核用 page 结构体表示每个物理页面的使用状况。

```
typedef struct page {
    struct page * next;                    //把 page 结构体链接成一个双向循环链表
    struct page * prev;
    struct inode * inode;                  //有关文件的 i 节点
    unsigned long offset;                  //在文件中的偏移量
```

```
    struct page * next_hash;                    //page 结构体连成一个哈希表
    atomic_t count;                             //页面的引用次数
    unsigned flags;                             //页面的状态
    unsigned dirty:16,age:8;                    //表示该页面是否被修改过
    struct wait_queue * wait;                   //等待该页面的进程队列
    …
    unsigned long map_nr;                       //物理页号
} mem_map_t;
```

2. Linux 的三级分页结构

三级分页结构是 Linux 提供的与硬件无关的分页管理方式。当 Linux 运行在某种计算机上时，需要利用该种计算机硬件的存储管理机制来实现分页存储。

如图 5-43 所示，三级分页管理把虚拟地址分成 4 个位段：页目录、页中间目录、页表、页内偏址。其中，页目录(Page Directory，PGD)用于存放页中间目录的地址，页中间目录(Page Middle Directory，PMD)用于存放页表地址，页表(Page Table，PTE)用于存放每页对应的存储块号。

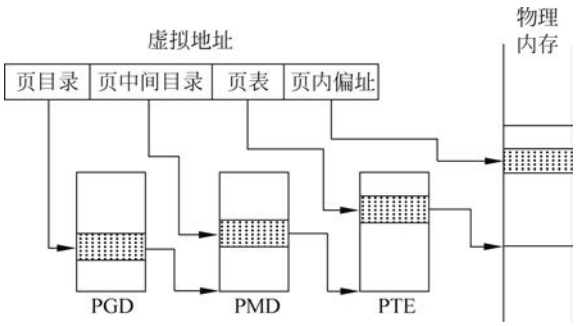


图 5-43　Linux 的三级分页管理

3. 内存的分配与释放

Linux 中用于内存分配和释放的函数主要是 kmalloc()和 kfree()，它们用于分配和释放连续的内存空间。

在使用 kmalloc()函数分配空闲块时以 Buddy 算法为基础。对 kmalloc()函数分配的内存页面块中加上一个信息头，它处于该页面块的前部。页面块中信息头后的空间是可以分配的内存空间。

```
void * kmalloc(size_t size, int priority)
```

其中，参数 size 是申请分配的内存的大小，priority 是申请优先级。

kfree()函数用于释放由 kmalloc()函数分配的内存空间。

```
void kfree(void * __ptr)
```

其中，ptr 是 kmalloc()函数分配的内存空间的首地址。

4. 虚拟内存的申请和释放

在申请和释放较小且连续的内存空间时，使用 kmalloc()和 kfree()函数在物理内存中进行分配。申请较大的内存空间时，使用 vmalloc()函数。由 vmalloc()函数申请的内存空间在虚拟内存中是连续的，它们映射到物理内存时，可以使用不连续的物理页面，而且仅把当前访问的部分放在物理页面中。

```
void * vmalloc(unsigned long size)
void vfree(void * addr)
```


其中,可以看到vmalloc()函数的参数size指出申请的内存的大小。分配成功后返回值为在虚存空间分配的虚存块首地址,失败后返回值为0。vfree()函数用来释放由vmalloc()函数分配的虚存块,参数addr是要释放的虚存块首址。

习　　题

1. 存储管理的主要内容有哪些?
2. 什么是静态地址重定位?什么是动态地址重定位?二者各有什么优缺点?
3. 内存信息保护常用的方法有哪几种?它们各自有什么特点?
4. 分区式内存管理有哪两类?它们各自的特点是什么?
5. 常用的分区内存分配算法有哪些?比较它们的优缺点。
6. 分区式存储管理有哪些优缺点?
7. 简述页式存储管理的基本思想及优缺点。
8. 动态页式存储管理中的页表内容主要有哪些?它们各自有什么作用?
9. 什么叫快表?为什么要引入快表?
10. 动态页式存储管理有哪些常用的淘汰算法?比较它们的优缺点。
11. 什么是Belady现象?试举一个Belady现象的例子。
12. 什么是抖动现象?如何防止抖动现象的产生?
13. 什么是段式存储管理?段式存储管理与页式存储管理的区别是什么?
14. 段式存储管理的优缺点是什么?
15. 段页式存储管理的基本思想是什么?
16. 为什么说段页式存储管理中的地址是三维的?
17. 什么是内碎片?什么是外碎片?
18. 某操作系统采用可变分区分配存储空间的管理方法,用户区为512KB且始址为0,用空闲分区表管理空闲区,且初始时用户区的512KB是空闲的,对下述申请序列:申请300KB,申请100KB,释放300KB,申请150KB,申请30KB,申请40KB,申请60KB,释放30KB,回答下列问题。

(1) 采用首次适应算法,给出空闲分区表内容(给出始址、大小)。

(2) 采用最佳适应算法,给出空闲分区表内容(给出始址、大小)。

(3) 如果再申请100KB,针对(1)和(2)各有什么结果?

19. 在一个页式存储管理系统中,某进程的页表如表5-1所示。已知页面大小为1024B,试将逻辑地址1011、2148、3000、4000、5012转化为相应的物理地址。

表5-1　某进程的页表

页号	块号	页号	块号
0	2	2	1
1	3	3	6

20. 在一个段式存储管理系统中，某进程的段表如表 5-2 所示。

表 5-2　某进程的段表

段号	基地址	段长/B
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

试给出下列各逻辑地址对应的物理地址：

(0,430),(1,10),(2,88),(3,444),(4,112)。

21. 一个进程的访问内存地址序列如下：

10,11,104,170,73,309,185,245,246,434,458,364

(1) 若页大小为 100B，给出访页顺序。

(2) 若分配该进程的内存空间为 200B，采用 FIFO 淘汰算法时，它的缺页次数是多少？

(3) 若采用 LRU 淘汰算法，给出缺页次数。

22. 某计算机内存按字节编址，逻辑地址和物理地址都是 32 位，若采用一级页表分页式存储管理方案，逻辑地址结构中，页号占 20 位，页表项占 4B，回答下列问题。

(1) 该计算机的页大小为多少 KB？一个页表最大占多少字节？

(2) 假设某一进程代码段的逻辑地址 1002 的指令被存放在内存的 2 号页块中，则该逻辑地址对应的物理地址是多少？

(3) 假设该进程的长度为 9KB，采用静态页式分配内存，则为该进程分配多少个页面块？是否存在页内碎片？

23. 假设有系统为某一进程分配了 4 个页面，某一时刻进程需调入 5 号页，这时进程的每个页面的使用情况如表 5-3 所示，问采用 FIFO、LRU 和改进的 CLOCK 淘汰算法，将会淘汰哪一页？

表 5-3　进程页的使用情况表

页　号	装入相对时间	上次引用的相对时间	R	M
0	120	280	0	0
1	200	260	1	0
2	100	268	1	1
3	160	290	1	1

24. Linux 内存采用哪种存储管理机制？

25. Linux 的地址结构组成部分有哪些？