第5章

存储管理

存储器是计算机系统的重要组成部分。近年来，尽管存储器的价格飞速下降，存储器的容量在不断扩大，但仍然不能保证有足够的空间存放需要运行的用户程序、操作系统程序及数据。存储器仍然是一种宝贵的资源。对存储器实施有效的管理，不仅直接影响到存储器的利用率，而且还对系统性能有重大影响。

存储管理的对象是内存。内存的管理方案有很多，从简单的单一连续分配到页式和段式存储方案，每种方案各有优缺点。为特定的系统选择存储管理方案依赖于很多因素，特别是硬件的设计，许多存储管理方案的实现都需要硬件的支持。

视频讲解

5.1 程序的装入和链接

在多道程序环境下，程序要运行首先必须装入内存。一个用户源程序变为一个可以在内存运行的程序，通常要经过编译、链接和装入三个步骤。

(1) 编译。用户源程序经过编译生成目标模块，目标模块以 0 作为开始地址。目标模块中的地址称为相对地址或逻辑地址。

(2) 链接。将编译后形成的多个目标模块以及它们所需要的库函数链接在一起形成装入模块。装入模块虽然具有统一的地址空间，但它仍是以 0 作为参考地址。

(3) 装入。将装入模块装入内存实际物理地址空间，并修改程序中与地址有关的代码，这一过程叫作地址重定位。

5.1.1 重定位

地址重定位完成的是相对地址(逻辑地址)转换成内存的绝对地址(物理地址)的工作。地址重定位又称为地址映射。按照重定位的时机，可分为静态重定位和动态重定位。

1. 静态重定位

静态重定位就是在程序执行之前进行重定位。它根据装入模块将要装入的内存起始地址修改装入模块中有关使用地址的代码，如图 5-1 所示。

图 5-1 中一个以 0 作为参考地址的装入模块要装入以 1000 为起始地址的内存实际存储空间中，装入时要做某些代码的修改。例如装入模块中有一条指令 LOAD 1,1500。该指令的意义是将相对地址为 1500 存储单元的内容 12345 装入 1 号寄存器。

当将装入模块装入起始地址为 1000 的内存空间后，原来程序中的地址 1500 就变成了 2500，即相对地址 1500 加上装入起始地址 1000。因此，LOAD 1,1500 这条指令中的地址码就变成了 LOAD 1,2500。

程序中涉及地址的每条指令都要进行这样的修改。若这种修改是在程序运行之前，程序

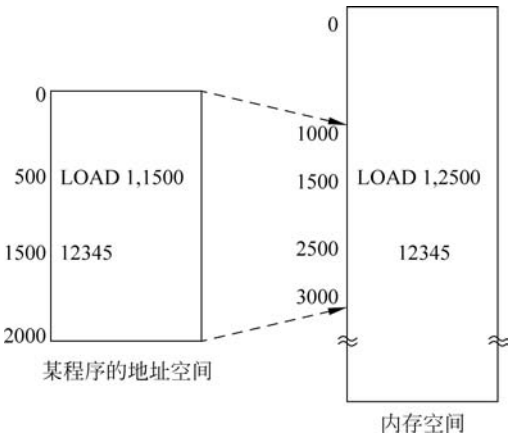

图 5-1　静态重定位示意图

装入时一次完成，以后不再改变。这种重定位就称为静态重定位。

静态重定位具有无须硬件支持的优点，但存在以下两个缺点。

（1）程序重定位之后不能再在内存中移动。

（2）要求程序的存储空间是连续的，不能把程序放在若干个不连续的存储区域内。

2. 动态重定位

动态重定位指程序在执行过程中进行地址重定位。更确切地说是在每次访问每个地址单元前再进行地址变换。动态重定位需要硬件——重定位寄存器的支持。

如图 5-2 所示，装入模块不进行任何修改就装入内存，程序中与地址相关的各项均保持原来的相对地址，如 LOAD 1,1500 这条指令中的地址仍是相对地址 1500，当该程序运行时，CPU 每取一条访问内存的指令，地址变换硬件逻辑就自动将指令中的相对地址与重定位寄存器中的值相加，再将此值作为内存绝对地址去访问该单元中的数据。

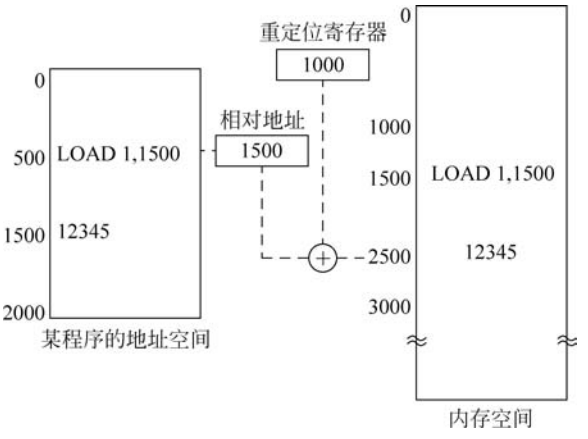

图 5-2　动态重定位示意图

由此可见，动态重定位实施的时机是在指令执行过程中，每次访问内存前动态进行。采用动态重定位可带来如下两个好处。

（1）目标模块装入内存时无须任何修改，因而装入后可以再搬迁。由进程管理的知识可知，当一个进程装入内存后，可能因内存紧张而挂起，挂起进程被换到外存。当进程被激活，重

新换入内存时，不一定还放在以前相同的内存区域，采用动态重定位方式，就可以把该进程重定位到内存的不同区域中。

(2) 一个程序是由若干个相对独立的目标模块组成的。每个目标模块装入内存时可以各放在一个存储区域，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重定位寄存器即可。

5.1.2 链接

链接程序的功能是将经过编译后得到的一组目标模块以及它们所需要的库函数装配成一个完整的装入模块。实现链接的方法有三种，分别为静态链接、装入时动态链接和运行时动态链接。

1. 静态链接

图 5-3 所示为经编译后得到的两个目标模块 MAIN 和 SUB，它们的长度分别是 L_1 和 L_2。在模块 MAIN 中有一条语句 CALL SUB，用于调用模块 SUB。SUB 属于外部符号引用。将这两个目标模块链接在一起时需要解决以下两个问题。

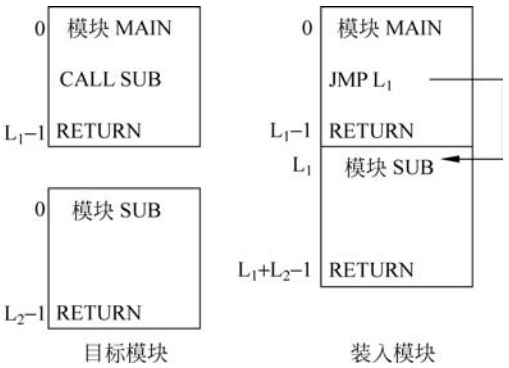

图 5-3 静态链接示意图

1) 相对地址的修改

由编译程序生成的目标模块起始地址为 0，每个模块的地址都是相对于 0 的，在链接成一个装入模块后，模块 SUB 的起始地址不再是 0，其地址空间为 $L_1 \sim L_1 + L_2 - 1$，此时需要修改模块 SUB 中所有与地址相关的指令，即用模块 SUB 中的相对地址加上 L_1。

2) 外部符号引用的变换

模块中所用的外部符号引用要变换成相对地址，如把 CALL SUB 变换成相对地址 L_1 的引用。

链接形成的完整的装入模块又称为可执行文件。链接以后的文件通常不会再拆开，要运行时可直接将它装入内存。这种在程序运行之前事先进行的链接称为静态链接。

2. 装入时动态链接

上述链接可以在程序装入内存时边装入边链接。如编译得到的两个目标模块 MAIN 和 SUB，在装入 MAIN 时，遇到了一个外部符号引用 CALL SUB，此时引用装入程序去找出相应的外部目标模块 SUB，并把它装入内存，同时修改目标模块 SUB 中的相对地址。

装入时动态链接有以下两个优点。

(1) 便于软件版本的更新。一个新的软件推出后经常需要排错，更新版本。有时错误的

出现只限于某几个模块，如果采用装入时动态链接的方法，可以在装入的过程中将新版本换入，方便易行。

（2）便于实现目标模块的共享。采用装入时动态链接的方法，操作系统可以将一个目标模块链接到几个应用模块上，从而实现模块的共享。

3. 运行时动态链接

装入时动态链接虽然比静态链接有优势，但它也有不足之处，主要表现在如下两个方面。

（1）程序的整个运行期间，装入模块是不改变的。实际应用中有时需要不断修改某些装入模块，并希望在程序不停止运行的情况下把修改后的装入模块与正在运行的应用模块链接在一起并运行。

（2）每次运行时的装入模块是相同的。而在实际应用的许多情况下，每次运行的模块可能是不同的，例如操作系统调用的驱动程序，因为每台机器的硬件配置可能发生变化，所以与硬件相关的驱动程序都是不同的。

解决这两个问题的有效方法是采用运行时动态链接的方式，即将目标模块的链接推迟到程序执行时再进行。在执行过程中，若发现被调用模块还没有装入内存，再去找出该模块，将它装入内存，并链接到调用模块上。

视频讲解

5.2 连续分配存储管理方式

连续分配是指为用户程序分配一个连续的内存空间。这种分配方式曾广泛地应用于20世纪60～70年代的操作系统中。连续分配的方式有单一连续分区、固定分区和可变分区。

5.2.1 单一连续分区

单一连续分区是最早出现的一种存储管理方式，它只能用于单用户、单任务的操作系统中，如CP/M和MS-DOS操作系统用的就是这一方案。

在该方案中，整个内存区域被分成系统区域和用户区域两部分。

（1）系统区域。提供给操作系统使用，它可以驻留在内存的低地址部分，也可以驻留在内存的高地址部分。

（2）用户区域。供应用程序使用的内存区域。

图5-4所示为操作系统和用户程序的三种组织方案，图5-4(a)所示为操作系统位于内存最低段的随机存储器(RAM)中；图5-4(b)所示为操作系统位于内存最高端的只读存储器(ROM)中；图5-4(c)所示为设备驱动程序位于内存最高端的只读存储器中，而操作系统的其他部分位于1MB内存低地址端的随机存储器中，IBM PC使用的就是这种方案，其中位于ROM的设备驱动程序处于地址空间的最高8KB，ROM中的这些程序称为BIOS(基本输入输出系统)。

为了防止操作系统程序受到用户程序有意或无意的破坏，可设置一个保护机构，常用的方法是使用界限寄存器。用户程序执行每条指令时，其物理地址与界限寄存器的地址将被进行比较，当确定没有超出用户地址范围时，再执行该指令，否则产生越界中断，并停止用户程序的执行。

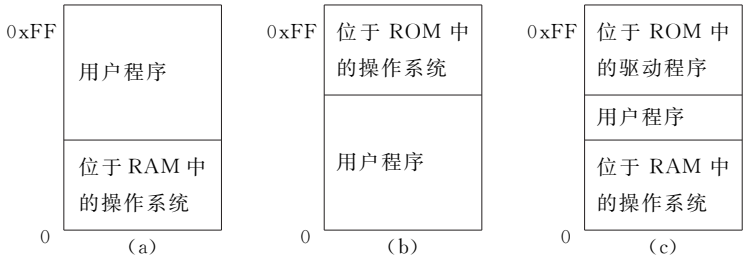

图 5-4　操作系统和用户程序的三种组织方案

5.2.2　固定分区

固定分区分配是最早使用的一种运行在多道程序中的存储管理方案。在进程装入内存之前，由操作员或操作系统把内存划分成若干个大小不等的分区。一旦划分好，在系统运行期间就不能重新划分。

为了记录每个分区的大小、起始地址和说明该分区是否已经分配，系统会建立一个分区说明表，如图 5-5(a)所示，当有一个用户进程要装入时，由内存分配程序负责检索该表，从表中找出一个能满足要求的、尚未分配的分区，并将它分配给该进程，然后修改分区说明表中的状态位；若找不到大小足够的分区，则拒绝为该用户进程分配内存空间。

分区号	大小	首址	状态
1	4MB	8MB	1
2	8MB	12MB	1
3	8MB	20MB	0
4	12MB	28MB	1
5	16MB	40MB	0

(a) 分区说明表

(b) 内存空间分配情况

图 5-5　固定分区分配

这种由操作员启动计算机后设置好，以后就不能再修改固定分区的存储系统，曾经在 IBM OS/360 大型机上运行了许多年，该系统称为 MFT 或 OS/MFT。

采用这种内存分配技术，虽然使多个用户进程共驻内存，但一个进程的大小刚好等于某个分区大小的情况非常少，于是每个分区中总有一部分被浪费。如图 5-5(b)所示的情况，如果有一个大小为 20MB 的进程申请内存，则将被系统拒绝，因为分区的大小是预先分配好的，此时分区说明表显示有两个分区是未分配状态，而它们的大小分别为 8MB 和 16M，均不能满足用户进程 20MB 的要求。

固定分区技术简单，但内存利用率不高，适用于进程的大小及数量事先能够预知的系统。

5.2.3 可变分区

可变分区是指在进程装入内存时，把可用的内存空间“切出”一个连续的区域分配给进程，以适应进程大小的需要。整个内存分区的大小和分区的个数不是固定不变的，而是根据装入进程的大小动态划分，因此也称为动态分区。

系统初启时，内存中除了常驻的操作系统程序外，其余是一个完整的大空闲区域。随后，根据进程的大小划分出一个分区。当系统运行一段时间之后，随着进程的撤销和新进程的不断装入，原来整块的内存区域就形成了空闲分区和已分配分区相间的局面，如图5-6所示。

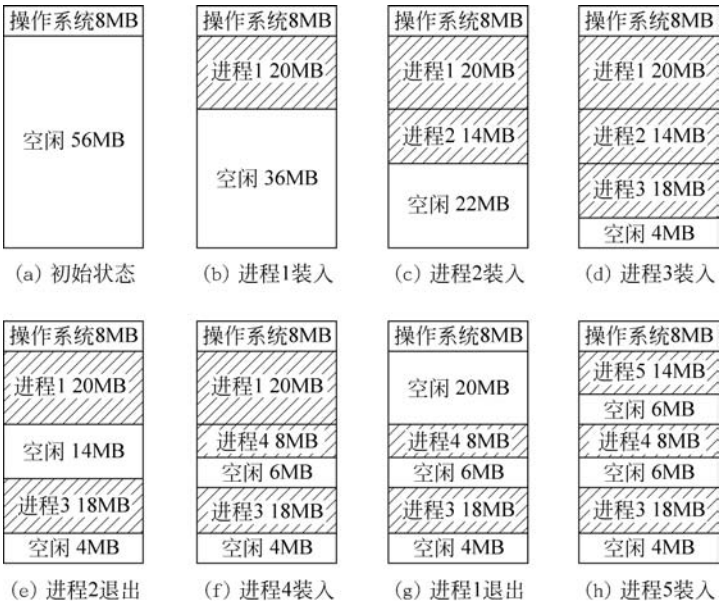

图5-6 可变分区示意图

1. 可变分区中的数据结构

为了实现可变分区的分配，必须配置相应的数据结构记录内存的使用情况，常用的数据结构有两种，即空闲分区表和空闲分区链。

(1) 空闲分区表为内存中每个尚未分配出去的分区设置一个表项，每个表项包括分区序号、分区起始地址和分区的大小。

(2) 空闲分区链即在每个空闲分区中设置用于控制分区分配的信息及用于链接各个分区的指针，将内存中的空闲分区链接成的一个链表。

2. 可变分区分配算法

为把一个进程装入内存，需要按一定的分配算法从空闲分区表或空闲分区链中选出一个分区分配给该进程。常用的可变分区分配算法有以下四种。

1) 首次适应(First Fit)算法

该算法要求空闲分区以地址递增的次序排序。如果采用的是链表结构，分配时则从链表的开始顺序进行查找，直到找到一个能够满足进程大小要求的空闲分区为止。然后按进程的大小从分区中“切出”一块内存空间分配给请求者，余下的空闲分区仍然留在链表中。

该算法倾向于优先使用内存中低地址部分的空闲空间，高地址部分很少被利用，从而保证高地址部分留有较大的空闲分区。其缺点是低地址部分不断被“切割”，致使留下许多难以利用的小空闲分区，而每次查找又都从低地址部分开始，这无疑会影响查找的速度。

2）下次适应(Next Fit)算法

该算法从首次适应算法演变而来。为了避免低地址部分小空闲分区的不断增加，在给进程分配内存空间时，不再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，直到找到一个能满足要求的空闲分区，并从中“切出”一块与请求的大小相等的内存空间分配出去。

为了实现该算法，需要设置起始查询指针，用于指示下一次查询的起始位置，链表的链接采用循环链表的形式，即如果找到最后一个空闲分区(链尾)仍然不能满足要求，应返回第一个空闲分区(链首)继续查找。

该算法的特点是可以使内存得到比较均衡的使用，减少查找空闲分区的开销，但会使系统缺乏大的空闲分区，导致比较大的进程无法运行。

3）最佳适应(Best Fit)算法

最佳的含义是指每次为进程分配内存时，总是把与进程大小最匹配的空闲分区分配出去。该算法采用的数据结构若是空闲分区链，首先要求将空闲分区按分区大小递增的顺序形成一个空闲分区链。当进程要求分配内存时，第一次找到的满足要求的空闲区必然是最优的。

该算法的优点是如果系统中有一个空闲分区的大小正好与进程的大小相等，则必然选中该空闲块；另外，系统中可能保留有较大的空闲分区。该算法的缺点是链表的头部会留下许多难以利用的小空闲区，称为碎片，从而影响分配的速度。

4）最坏适应(Worst Fit)算法

该算法与最佳适应算法相反，要求空闲分区按分区大小递减的顺序排序，每次分配时，从链首找到最大的空闲分区“切出”一块进行分配。

该算法的特点是基本上不会留下小空闲分区，不易形成碎片；缺点是大的空闲分区被“切割”，当有较大的进程需要运行时，系统往往不能满足要求。

3. 可变分区内存的回收

进程运行完毕后就要释放占有的内存，系统回收它所占的分区时，应考虑回收分区是否与空闲分区邻接，若有邻接，则应加以合并。回收分区与空闲分区的邻接情况可能有以下四种。

(1) 回收分区与前面一个(低地址)空闲分区 F_1 相邻接，如图 5-7(a)所示，此时将回收分区与空闲分区合并为一个空闲分区，并修改空闲分区 F_1 的大小为两个分区的大小之和。

(2) 回收分区与后面一个(高地址)空闲分区 F_2 相邻接，如图 5-7(b)所示，此时将回收分区与空闲分区合并为一个空闲分区，回收分区的首地址作为合并后新空闲分区的首地址，大小为两个分区的大小之和。

(3) 回收分区与前、后两个空闲分区 F_1 和 F_2 均相邻，如图 5-7(c)所示，此时将回收分区与前、后空闲分区合并为一个空闲分区，合并后形成的新空闲分区的首地址为前一个空闲分区的首地址，大小为三个空闲分区的大小之和，同时取消 F_2 在空闲分区链(表)中的表项。

(4) 回收分区不与其他空闲分区相邻接，此时应为回收分区单独建立一个新的表项，填写回收分区的首地址和大小，并将该分区插入链(表)中的适当位置。

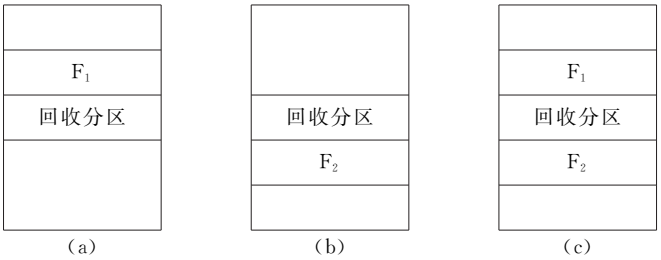

图 5-7　内存回收时的情况

5.2.4　动态重定位分区

在前文所述的可变分区分配方案中，由于必须把一个用户进程装入一个连续的内存空间，如果系统中存在若干个小的空闲分区，其总容量大于要装入的进程，但由于每个空闲分区的大小都小于进程的大小，故该进程不能装入。如图 5-8(a)所示，内存中有三个空闲分区，它们的大小分别为 15MB、12MB 和 20MB，三个分区的总容量是 47MB，现在有一个大小为 25MB 的进程，因为必须给进程分配一个连续的空闲分区，所以当前进程无法装入。

为了解决这一问题，可以采用紧凑的方法。为了能将上述 25MB 的进程装入内存，采用的方法是将内存中原来的所有进程进行移动，使它们互相邻接。这样一来，原来分散的多个空闲分区便拼接成了一个大的空闲分区，如图 5-8(b)所示。因为紧凑后的用户进程在内存中的位置发生了变化，所以要对进程中的地址进行修改，即重定位。

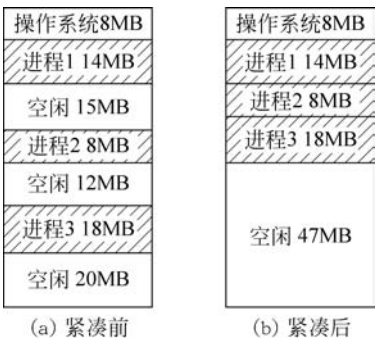

图 5-8　紧凑

允许进程运行过程中在内存中移动，必须采用动态重定位的方法，即将进程中的相对地址转换成物理地址的工作推迟到进程指令真正执行时进行。

要实现进程在内存中的移动，必须获得硬件地址变换机构的支持，即在系统中必须增加一个重定位寄存器，用它存放进程在内存中的起始地址。进程在执行时，真正要访问的内存地址是进程的相对地址加上重定位寄存器中的地址。

视频讲解

5.3　页式存储管理

动态重定位分区虽然可以解决碎片问题，但由于内存大量信息的移动，为此付出的处理机时间开销很大。因此，用户进程经常受到内存空闲空间容量的限制，常常出现不能运行的情况。出现这一问题的主要原因是一个进程必须存放在一个连续的内存空间中。如果允许将一个进程分散地分配到许多不相邻的分区中，就可以解决这一问题。基于这一思想，产生了离散分配方式，页式存储管理技术就是离散式分配方式的一种。

5.3.1　页式存储管理的基本原理

在页式存储管理方式中，把内存空间分成大小相同的若干个存储块(或称为页框)，并为这些存储块进行编号为 0 块、1 块、…、($n-1$)块。相应地，将进程的逻辑地址空间分成若干个与

内存块大小相等的页(或称为页面)。在为进程分配内存空间时,以页为单位进行。进程中的若干个页分别装入多个不相邻的存储块。进程的最后一页经常装不满一个存储块,而形成不可利用的碎片,称为页内碎片。

1. 程序分页和内存分块

图 5-9 说明了页式存储管理系统进程的装入情况。其中,进程 A 由 4 页组成,初始时,供用户进程使用的所有内存块都是空闲的,如图 5-9(a)所示。进程 A 装入存储块 0、1、2 和 3 中,如图 5-9(b)所示。随后进程 B 和进程 C 装入,进程 B 和 C 分别都有 3 页,如图 5-9(c)和图 5-9(d)所示。然后进程 B 被挂起退出内存,如图 5-9(e)所示。后来进程 D 又进入,它由 4 页组成,分别占用了存储块 4、5、6 和 10,如图 5-9(f)所示。

图 5-9 页式存储管理系统进程的装入情况

2. 页表

在页式管理系统中,进程的若干个页被离散地存储在内存的多个存储块中,为了能找到每个页所对应的存储块,系统为每个进程都建立一张页表。进程所有页依次在页表中有一个页表项,其中记录了相应页在内存中对应的物理块号。配置了页表后,进程执行时通过查找页表,就可以找到每页在内存中的存储块号。可见,页表的作用是实现从页号到存储块号的地址映射。

此外,系统除了为每个进程建立一张页表之外,还应建立一张空闲块表,该表按存储块号从小到大的次序记录内存未分配存储块的块号。图 5-10 列出了图 5-9 中进程 D 装入后进程 A、B、C 和 D 的页表和内存空闲块表。

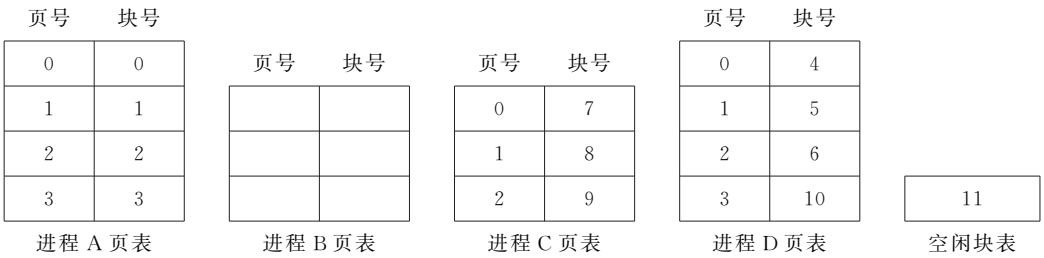

图 5-10　进程的页表和空闲块表

3. 页的大小

在页式管理系统中，页或存储块的大小由机器的地址结构决定，一般使用 2 的幂作为页的大小。若选择的页面较小，可以使页内的碎片减小，有利于提高内存的利用率，但是同时也使进程要求页面数增加，从而使页表的长度增大，占用大量内存空间；若选择的页面较大，虽然可以减小页表的长度，却又会使页内碎片增大。因此，页面的大小应适中选择，通常页的大小为 512B～4MB。

5.3.2　页式存储管理的地址变换

为了能将用户地址空间中程序的逻辑地址变换成内存空间中的物理地址，系统中必须设置地址变换机构。该机构的任务是实现逻辑地址到物理地址的动态重定位。

由于页表大多驻留在内存，因此系统中应设置一个页表寄存器(Page Table Register)，其中存放页表在内存的起始地址和页表长度。进程没有执行时，进程的页表始址和长度放在本进程的 PCB 中。当某进程被调度执行时，才将其页表始址和长度放在页表寄存器中。因此在单处理机环境中，系统只需要一个页表寄存器。

当进程的某个逻辑地址被访问执行时，硬件地址变换机构根据页的大小自动将有效的逻辑地址分成页号和页内位移两部分，如图 5-11 所示，图中的地址是 16 位的。

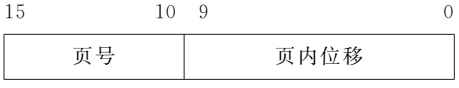

图 5-11　逻辑地址结构

首先将页号与页表寄存器中页表的长度进行比较，如果页号大于页表长度，表示本次访问的地址超出了进程的地址空间。于是系统发现该错误后将产生一个越界中断。否则通过页表寄存器中的页表始址找到页表，并根据页号找出相应的页表项，得到该页的存储块号。将存储块号装入物理地址寄存器，同时再将页内位移直接送入物理地址寄存器的块内位移字段。这样就完成了逻辑地址到物理地址的变换。

图 5-12 所示为页式地址变换机构，页的大小为 1024B，逻辑地址 1500 的二进制形式为 0000 0101 1101 1100。由于页的大小为 1024B，故页内位移占 10 位，剩下 6 位为页号。因此逻辑地址 1500 对应的页号为 1(二进制为 0000 01)，页内位移为 476(二进制为 01 1101 1100)。

查找页表得知，页号为 1 的存储块号为 6(二进制为 0001 10)，页内位移为 476(二进制为 01 1101 1100)，二者形成的物理地址为 $6\times1024+476=6620$ (16 位二进制数为 0001 1001 1101 1100)，因此进程要访问的逻辑地址 1500 对应的内存物理地址为 6620。

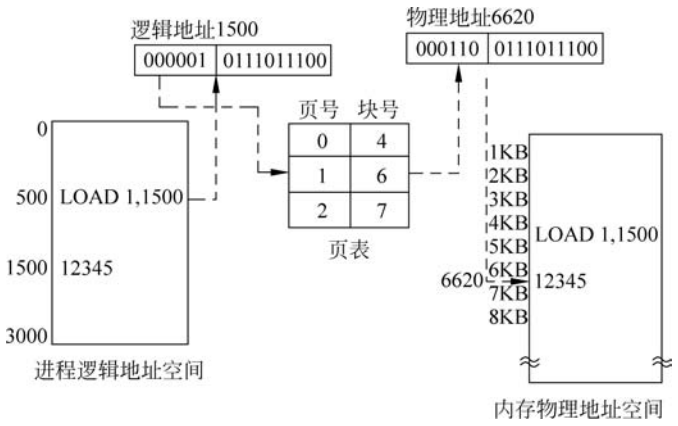

图 5-12　页式地址变换机构

5.3.3　页表的硬件实现

每种操作系统都有自己保存页表的方式，大多数系统为每个进程分配一个页表，在进程控制块中存放指向该页表的指针。

页表的实现方式有很多，最简单的方式是用一组专门的寄存器来实现，当调度程序为选中的进程加载寄存器时，这些页表寄存器一同加载。由于寄存器具有很高的访问速度，有利于提高页表查找的速度。但寄存器的成本太高，而页表又很大(可能有 100 万项)，完全用寄存器实现不太可能，因此大多数系统的页表都放在操作系统管理的内存空间内。

页表是存放在内存中的，这使 CPU 每次要存取一个数据时都要访问两次内存。第一次是访问内存中的页表，从页表中找到该页的存储块号，将此块号与页内位移形成物理地址。第二次访问内存时才真正从得到的物理地址存取数据。因此，计算机的处理速度降低了近一半。可见，引入页式存储管理系统付出的代价是昂贵的。

为了提高系统的地址变换速度，在地址变换机构中设置一组由高速寄存器组成的小容量联想存储器(Associate Memory，有时也称为 Translation Lookaside Buffer，TLB)构成一张所谓的快表，用来存放当前访问最频繁的少量活动页。如果用户要找的页在快表中能找到，即可得到相应页的块号，从而形成物理地址；如果找不到，就必须访问页表。有了快表以后，页式管理系统的地址变换过程如下。

(1) 在处理机得到进程的逻辑地址后，将该地址分成页号和页内位移两部分。

(2) 由地址变换机构自动将页号与快表中的所有页进行比较，若与其中某页相匹配，则从快表中查出对应页的存储块号，并转(4)；如在快表中没有找到，则转(3)。

(3) 访问内存中的页表，找到后读出存储块号。同时将该页表项内容存入快表中的一个单元，即修改快表。如果联想存储器已满，将快表中被认为不再需要的页换出。

(4) 由存储块号与页内位移得到物理地址。

由于成本的关系，联想存储器不可能做得很大，通常只有几十个表项。例如 Motorola 68030 处理器中有 22 个，Intel 80486 有 32 个。对于较小的进程可以将其页表中的所有内容存入联想存储器；对于较大的进程，只能将页表中的一部分存入其中。通常联想存储器的命中率为 80%～90%。

5.3.4 页表的组织

现代计算机系统大多支持非常大的逻辑地址空间。在这样的环境中，页表就变得很大，占用相当大的内存空间。例如，对于具有32位逻辑地址空间的页式管理系统，如果页的大小为4KB，即2^{12}B，则每个进程的页表项将达到2^{20}，即1MB之多。每个页表项占4字节，每个进程的页表要占用4MB的内存空间，而且还要求是连续的。显然这是不现实的，解决这一问题的方法如下。

(1) 对于页表所需要的内存空间，采用离散的方式将页表放在多个不连续的内存空间中。

(2) 只将页表的一部分页表项调入内存，其余仍驻留在磁盘上，需要时再调入。

1. 两级页表

对于难以找到连续的内存空间存放页表的问题，可将页表分页。将页表分成若干页，即依次为0页、1页、…、($n-1$)页。这样可以将每一页分别放在一个不同的存储块中。为了记录这些页在内存的存放情况，同样为离散分配的页表再建立一张页表，称为外层页表。这样就产生了两级页表。例如，逻辑地址空间为32位，页的大小为4KB，若采用一级页表结构，页表项则为1MB之多。若采用两级页表结构，对页表再进行分页，使每页包含2^{10}(即1KB)个页表项，则最多有2^{10}(即1KB)个这样的页表。或者说，外层页表的地址占10位，内层页表的地址占10位，此时的逻辑地址结构如图5-13所示。

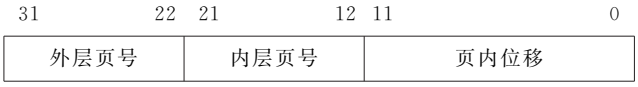

图5-13 两级页表的逻辑地址结构

图5-14所示为两级页表的结构，内层页表中的每个页表项存放的是进程的某一页在内存中的存储块号，外层页表中的每个页表项存放的是某个页表在内存中的存储块号。例如内层页表第0页页表中，页号0的存储块号为2。在外层页表中，页表项0存放的是第0页页表在内存的存储块号为2019。这样，用两级页表就可以实现逻辑地址到物理地址的变换。

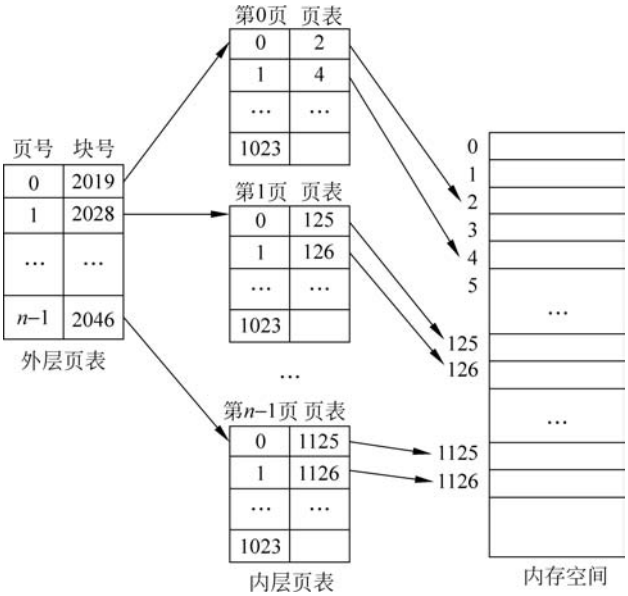

图5-14 两级页表的结构

用两级页表即可实现将页表离散存放的问题，另外还可以将部分页表放在内存，其他大多数页表放在磁盘，需要时再调入内存。具体操作是，对于正在运行的进程，先将其外层页表放入内存，然后再根据需要将内层页表的一页或几页调入内存，为了表明哪些页表在内存，可以在外层页表中增加一状态位，值为0表示对应的页表不在内存，为1表示对应的页表在内存。当需要访问不在内存的页表时，产生一个缺页中断，再由中断处理程序负责将不在内存的页表调入内存。

2. 多级页表

对于32位的计算机，使用两级页表结构是合适的，但对于64位的计算机系统，采用两级页表仍然不能解决问题。如果页的大小仍是4KB(即2^{12})，占用12位，那么将剩下52位，假定仍按2^{10}来划分页表，还余下42位用于外层页号，此时外层页表有4096G个页表项；假定按2^{20}来划分页表，还余下32位用于外层页号，此时外层页表有4G个页表项。可见，无论如何划分，结果都是不能接受的。因此必须采用多级页表，即对于4G的页表项再进行分页，这样就有了三级页表和四级页表。在实际的系统中，如SUN公司的SPARC处理器使用的是三级页表结构，而Motorola 68030处理器使用的是四级页表结构。

3. 反置页表

在页式管理系统中，每个进程有一个页表，进程逻辑地址空间的每一页在页表中都对应一个页表项。现代计算机系统中通常允许进程的逻辑地址空间非常大，因此页表项很多，从而占用大量内存空间。解决这一问题的另一个方法是采用反置页表。一般的页表都按页号进行排序，页表中的内容是存储块号。在反置页表中，为每个存储块设置一个页表项，并按存储块号排序，反置页表中的内容是页号及其所属进程的标识符。例如，对于一个具有64MB内存的计算机系统，如果页的大小为4KB，反置页表的页表项只有16KB。然而该表中只包含已经调入内存的页，并未包含不在内存的页，因此必须为每个进程建立一个外部页表，表中包含了各个页在外存的物理位置。当所访问的页在内存时，查询反置页表；当所访问的页不在内存时，通过外部页表将所需要的页调入内存。

在反置页表的使用中，是用进程的标识符和页号去检索反置页表的。由于页表项的数目很大，检索相当费时，通常采用Hash检索法。IBM AS/400、IBM RISC System 6000等系统都采用反置页表技术。

5.4 段式存储管理

视频讲解

段式存储管理方式的引入，主要是为了满足用户在编程和使用上的要求。具体来说有以下五点。

(1) 方便编程。通常，人们写的程序是分成许多子程序段的，每个段有自己的名字和长度，要访问的逻辑地址是由段名(或段号)和段内地址决定的。每个段从0开始编址，程序在执行过程中用段名和段内地址进行访问。

(2) 段的共享。实现程序和数据的共享都是以信息的逻辑单位为基础的，如共享某个函数或数据段。在页式存储管理系统中，每一页都是存放信息的物理单位，其本身并没有完整的意义，因而不便于实现页信息的共享。而段是信息的逻辑单位，实现段信息的共享更有意义。

(3) 段的保护。为了防止其他程序对某程序和数据造成破坏，必须采取某些保护措施。在段式系统中，对内存中物理信息的保护同样是对信息的逻辑单位的保护。因此，采用段式存

储管理方案，对于实现保护功能来讲更为有效和方便。

（4）动态链接。前文介绍过动态链接的概念。动态链接就是在程序运行过程中实现目标模块的链接。只有在段式存储管理方案中才能实现在程序运行过程中调用某段时，才将该段（目标模块）调入内存并进行链接。可见，动态链接也要求以段为存储管理单位。

（5）动态增长。在程序运行过程中，往往有些段，特别是数据段，会不断地增长，而事先又无法确切地知道数据段会增长到多大。这种动态增长的情况在其他几种存储管理方案中是难以应付的。段式存储管理系统却能很好地解决这一问题。

5.4.1 段式存储管理的基本原理

另一种离散分配方案是对进程分段。将进程对应的程序和数据按照其本身的特性分成若干个段，每个段定义了一组有意义的逻辑信息单位，如主程序段 MAIN、子程序段 SUB、数据段 DATA 等。每个段有自己的名字并且从 0 开始编址，段的长度由相应逻辑信息单位的长度决定。在内存中，每个段占用一段连续的分区。

进程的地址空间由于分成多个段，所以标识某一进程的地址时，要同时给出段名和段内地址。因此地址空间是二维的。进程地址的一般形式由一对数（S，W）组成，S 是段号，W 是段内地址，如图 5-15 所示。该地址结构允许一个进程最多有 64K 个段，每个段的最大长度是 64KB。

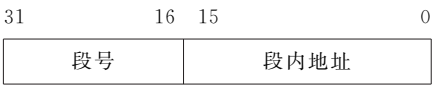

图 5-15　段式地址结构

在段式存储管理中，为每个段分配一个连续的分区，而进程的每个段可以离散地放在内存的不同分区中。为了使进程能正常运行，即对于进程的每个逻辑地址找出其在内存的实际物理地址，需要像页式存储管理一样，系统为每个进程建立一张段表。在段表中，每个段占有一表项，其中记录该段在内存的起始地址和段的长度，如图 5-16 所示，通过查找段表，可实现从进程逻辑地址到内存物理地址的映射。

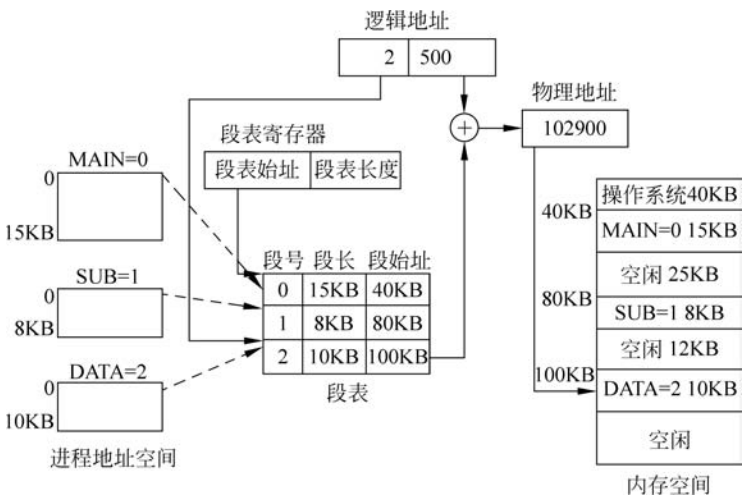

图 5-16　段式地址映射

5.4.2 段式存储管理系统的地址变换

段式存储管理系统的地址变换过程如下。

(1) 为了实现进程逻辑地址到内存物理地址的变换,系统中设置了段表寄存器,其中存放了段表始址和段表长度。在进行地址变换时,如果逻辑地址中的段号大于段表寄存器中段表的长度,则产生一个越界中断。

(2) 由段表始址找到段表在内存的位置,通过逻辑地址中的段号访问相应的段表项,如图5-20中逻辑地址中的段号为2,在段表从找到段号为2的段表项,得知该段的段长是10KB,它在内存的始址为100KB。

(3) 检查逻辑地址中的段内位移是否超过该段的段长。若超过,发出越界中断。

(4) 若没有越界,将该段的段始址与段内位移相加(100KB+500B=102 900B),即得到要访问的内存物理地址。

5.4.3 分段和分页的区别

表面上看,段式存储管理系统的地址变换过程与页式系统非常相似,但实质上它们有着本质的区别,主要表现在以下几方面。

(1) 页是信息的物理单位,分页是为了实现进程在内存的有效离散存放,以减少碎片,提高内存的利用率。而段是信息的逻辑单位,段是一组有意义的相对完整的信息。分页是系统管理的需要,而分段的目的是满足用户的需要。

(2) 页的大小是固定的,把逻辑地址分成页号和页内位移两部分,是由机器硬件实现的,因而一个系统只能有一种大小的页;段的长度是可变的,由用户所编写的程序决定。

(3) 页的逻辑地址空间是一维的,给出页的逻辑地址时只给出一个地址;而段的逻辑地址空间是二维的,在给出段的逻辑地址时既要给出段号,又要给出段内地址。

5.4.4 段的共享与保护

1. 页共享与段共享的比较

段式存储管理系统有一个突出的优点,就是易于实现段共享,即允许多个进程共享一个或多个段,而且对段的保护也十分简单易行。在页式系统中,虽然也可以实现程序和数据的共享,但远不如段式系统实现起来方便。下面通过一个例子说明这个问题。

有一个多用户系统,可同时容纳40个用户,他们都执行文本编辑程序。文本编辑程序含有160KB代码和40KB的数据区,如果不共享,则40个用户需要8MB的内存空间。如果代码段能共享,则只需1760KB(40×40KB+160KB)内存空间。

在页式系统中,假定页的大小为4KB,那么160KB的代码将占用40个页面,40KB的数据占用10个页面。为了实现代码共享,应在每个进程的页表中建立40个页表项,它们指向相同的存储块号20～59。每个进程的数据区建立10个页表项,它们可以指向不同的存储块号60～69、70～79、…,如图5-17所示。

值得注意的是,在各个进程的页表中,对于共享页面,不仅存储块号相同,其页号也必须相同。这是因为页号是由程序的逻辑地址决定的,而两个进程使用的是同一段程序,程序中每条指令的逻辑地址是一样的。

页面共享的方法是采用内存映射文件。内存映射文件的思想是进程可以通过文件映射(FileMapping)将文件所在的磁盘块映射成内存的一页(或多页)。当按普通文件访问磁盘时,就将文件的一部分读入物理内存,以后文件的读写就按通常的内存访问来处理。

如果多个进程将同一文件映射到自己的虚拟内存中,就产生了数据的共享。一个进程修

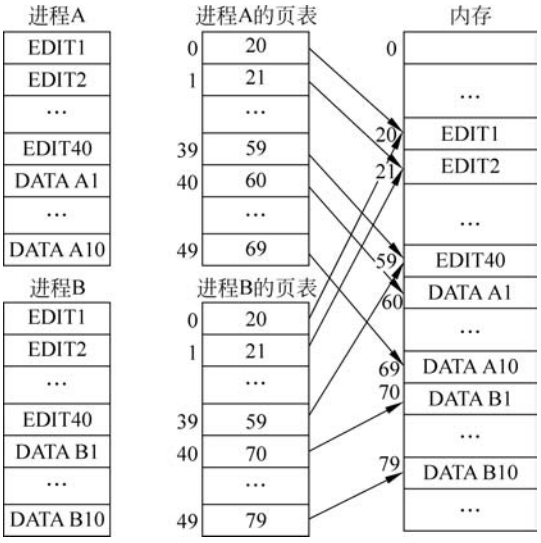

图 5-17　页式存储管理系统共享示意图

改虚拟内存中的数据，就会为其他映射相同部分文件的进程所见。

如果两个或两个以上的进程同时映射了同一个文件，它们就可以通过共享内存来通信。如果一个进程在共享内存上完成了写操作，此刻，当另一个进程在映射到这个文件的虚拟地址空间上执行读操作时，它就可以立刻看到上一个进程写操作的结果。因此，这个机制提供了一个进程之间的通信通道。

内存映射对其他设备也适用，如用来连接 Modem 和打印机的串口与并口。通过读、写这些设备的端口，CPU 可以与这些设备传递数据。

在段式系统中实现共享容易得多，在每个进程的段表中，为共享段 EDIT 设置一段表项即可，如图 5-18 所示。

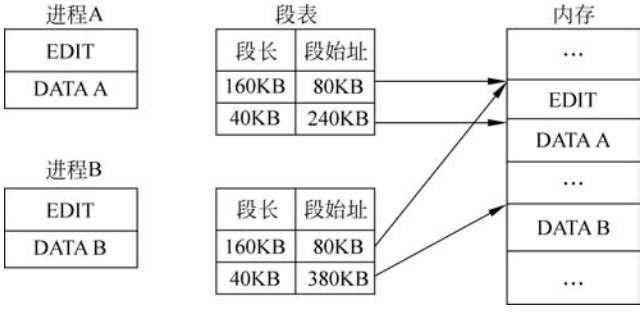

图 5-18　段式存储管理系统共享示意图

2. 共享段表

为了更好地实现段的共享，在系统中可配置一张共享段表，所有共享段都在共享段表中有一表项。表中记录了共享段的段名、段长、内存始址、状态、外存始址（如果采用段式存储管理时需要后两项）及共享该段的进程数，除此之外还记录了共享此段的进程情况，如进程名、进程号、该段在某个进程中的段号以及进程对该段的存取控制权限，如图 5-19 所示。

段名	段长	内存始址	状态	外存始址
共享进程计数 count				
进程名	进程号	段号	存取控制	
…				

图 5-19 共享段表项

表中某些字段的含义如下。

(1) 共享进程计数记录了共享某段的进程个数。对于非共享段,它仅为某个进程所有,当进程不再需要该段时,可立即释放其占有的内存空间,并由系统回收该段占用的内存空间。而共享段是为多个进程所需要的,当某个进程不再需要而释放它时,系统并不能回收其占有的内存空间。只有当所有共享该段的进程都不再需要它时,才由系统回收该段所占有的内存空间。为了记录有多少个进程共享该段,特设置一整型变量 count。

(2) 存取控制。对于一个共享段,不同的进程可以有不同的存取控制权限。例如,若共享段是数据段,对于建立该数据段的进程允许其读和写,对于其他进程可只允许读。

(3) 段号。对于同一共享段,不同的进程可以使用不同的段号去共享该段。

3. 共享段的分配与回收

由于共享段是供多个进程所共享使用的,因此对共享段的内存分配方法与非共享段有所不同。在分配共享段内存时,当第一个进程请求使用该共享段时,由系统为该共享段分配一个内存区域,并把共享段调入其中,同时将该段的始址填入该进程段表的相应项中,并在共享段表中增加一表目,填写有关数据,把共享进程计数 count 置为 1。之后,当其他进程调用该共享段时,由于该段已被调入内存,故无须再为该段分配内存,只需要在调用进程的段表中增加一个表项,填入该共享段的内存地址; 在共享段表中填入调用进程名、进程号、段号和存取控制,执行共享进程计数加 1 操作(count=count+1)。

当共享此段的进程不再需要它时,执行共享进程计数减 1 操作(count=count-1),若减 1 后结果为 0,则需要由系统回收该共享段的物理内存,以取消该段在共享段表中对应表项,表明此时已没有进程使用该段; 否则(减 1 后不为 0)只是取消调用进程在共享段表中的有关记录。

4. 段的保护

段式存储管理系统另一个突出的优点是便于对段的保护。因为段是有意义的逻辑信息单位,即使在进程执行过程中也是这样。因此段的内容可以被多个进程以相同的方式使用,如一个程序段中只含指令,指令在执行过程中是不能被修改的,对指令段的存取方式可以定义为只读和可执行。而另一段只含数据,数据段则可读可写,但不能执行。在进程执行过程中,地址变换机构对段表中的存取保护位的信息进行检验,防止对段内的信息进行非法存取。

段的保护措施有以下三种。

(1) 存取控制。在段表中增加存取保护位,用于设置对本段的存取方式,如可读、可写或可执行。

(2) 段表保护。每个进程都有自己的段表,段表本身对段可起到保护作用。由于段表中记录段的长度,在进行地址变换时,如果段内地址超过段长,便发出越界中断,这样就限制了各段的活动范围。另外,段表寄存器中有段表长度信息,如果进程逻辑地址中的段号超过段表长度,系统同样产生中断,从而进程也被限制在自己的地址空间中活动,不会发生一个进程访问另外一个进程的地址空间的现象。

（3）保护环。其基本思想是系统把所有信息按照其作用和相互调用关系分成不同的层次（即环），低编号的环具有较高的权限，编号越高，其权限越低，如图 5-20 所示。它支持四个保护级别，0 级权限最高，3 级最低。

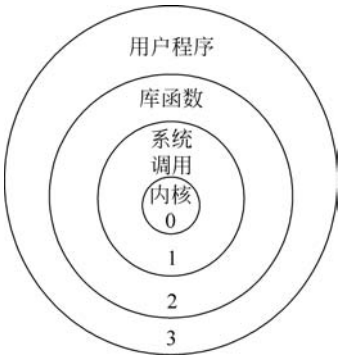

图 5-20　环保护机制

① 0 级是操作系统内核，它处理 I/O、存储管理和其他关键的操作。

② 1 级是系统调用处理程序，用户程序可以调用系统提供的系统调用，但只有一些特定的和受保护的系统调用才提供给用户。

③ 2 级是库函数，它可能是由很多正在运行的进程共享的，用户程序可以调用这些过程，读取它们的数据，但不能修改它们。

④ 用户程序运行在 3 级上，受到的保护最少。

在环保护机制下，程序的访问和调用遵循一个环内的段可以访问同环内或环号更大的环中的数据，一个环内的段可以调用同环内或环号更小的环中的服务的规则。

在任何时刻，运行程序都处于由 PSW 中的两位所指出的某个保护级别上。只要程序只使用与它同级的段，一切都会正常。对更高级别数据的存取是允许的，而对更低级别数据的存取是非法的，并会引起保护中断。调用更低级别的过程是允许的，但要通过严格的控制。为了执行越级调用，调用指令必须包含一个选择符，该选择符指向一个称为调用门（Call Gate）的描述符，由它给出被调用过程的地址。因此，要跳转到任何一个级别代码段的中间都是不可能的，只有正式指定的入口点可以使用。

5.5　段页式存储管理

段式和页式存储管理系统各有优缺点。页式系统能有效地提高内存利用率，而段式系统能更好地满足用户的需要。对这两种存储管理方案“各取所长”，将二者结合成一种新的存储管理系统，既有段式系统便于实现段的共享、段的保护、动态链接和段的动态增长等一系列优点，又能像页式系统那样很好地解决内存的外部碎片问题。这种结合就产生了段页式存储管理系统。

5.5.1　段页式存储管理的基本原理

段页式存储管理系统是段式系统和页式系统的组合。先将进程分段，再将每个段分成若干页。图 5-21 展示了段页式存储管理系统进程地址空间的结构。该进程有三个段，为主程序段、子程序段和数据段。其大小分别是 15KB、8KB 和 10KB。页的大小为 4KB。主程序段被分成四页；子程序段分成两页；数据段被分成三页。

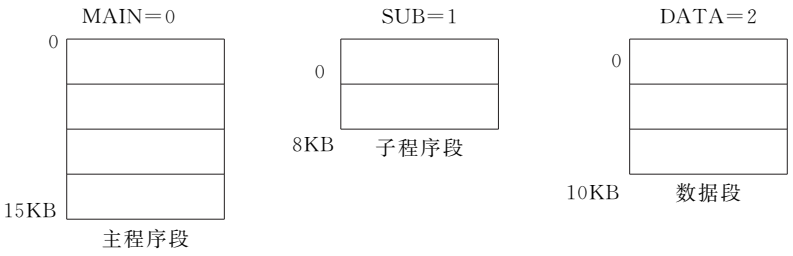

图 5-21　段页式存储管理系统进程地址空间结构

在段页式存储管理系统中，进程的逻辑地址由段号、段内页号和页内位移三部分组成，如图 5-22 所示。

段号(S)	段内页号(P)	页内位移(W)

图 5-22　段页式存储管理系统逻辑地址结构

段页式存储管理系统中，为了实现从进程逻辑地址到内存物理地址的变换，系统要配置段表和页表。由于允许将一个段的若干页离散地存放，因而段的大小变化不是段长的变化，而是页表长度的变化。

5.5.2　段页式存储管理的地址变换

在段页式存储管理系统中，为了实现地址映射，必须配置段表寄存器，其中存放段表始址和段表长度。每个进程有一个段表，每个段表项存放一个段的情况，其中包括段号、标识该段是否在内存的状态位、该段的页表长度和页表始址。每个段有一个页表，其中有页号、标识该页是否在内存的状态位和存储块号。图 5-23 所示为利用段表和页表进行地址变换的映射过程。

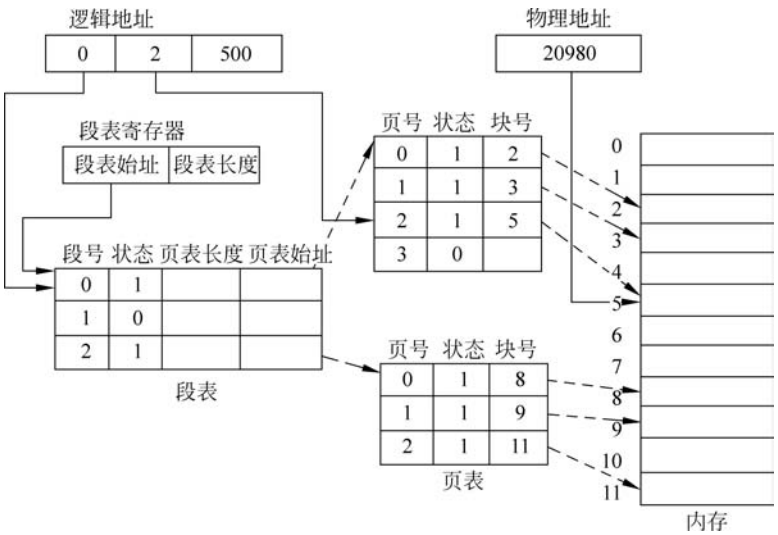

图 5-23　段页式存储管理系统地址映射

如果页的大小为 4KB，段页式系统地址变换过程如下。

(1) 对于给定的逻辑地址，用段号 S(图 5-23 中为 0)与段表寄存器中段表长度进行比较。若段号大于段表长度，产生一次越界中断。

(2) 通过段表寄存器中的段表始址找到该段在内存的段表，从段表中找出段号 S 对应的段表项。

(3) 查看该段的状态位，若该段不在内存，则产生一次缺段中断，并将所缺段调入内存。如果内存没有空间，则需要进行置换。

(4) 由该段的页表始址找到该段的页表。

(5) 用逻辑地址中的段内页号 P(图 5-23 中为 2)与页表长度进行比较。若段内页号 P 大于页表长度，则产生一次越界中断。

（6）从页表中得到页号P对应的页表项，查看该页的状态位，如果该页不在内存，则产生一次缺页中断。如果内存没有空间，则需要进行置换。

（7）得到该页在内存的存储块号（图5-23中为5），并与逻辑地址中的页内位移（为500B）构成物理地址（5×4KB+500B=20 980B）。

在段页式存储管理系统中，为了获得一条指令或数据，需要三次访问内存。第一次访问内存中的段表，从中得到页表始址。第二次访问内存中的页表，从中取得该页所对应的存储块号，并将存储块号与页内位移一起形成指令或数据的物理地址。第三次用得到的物理地址真正访问指令或数据。显然，访问内存的次数增加了两次。为了提高指令的执行速度，在地址变换时需要增加联想存储器，即快表。将段表和页表的部分内容放入其中。进行地址变换时查找快表代替查找段表和页表。但若在快表中找不到所需段和页，仍需要三次访问内存。

5.5.3 段页式存储管理系统举例

Intel 80386是一种常见的段页式存储管理系统。Intel 80386既支持分段，也支持分页。根据需要可构成以下四种存储管理方式。

（1）不分段也不分页。这种方式可用于高性能的控制器。

（2）分页不分段。这种方式成为一个单纯的页式存储管理系统，UNIX/386采用这种方式。

（3）分段不分页。

（4）段页式存储管理机制。这是性能最好的一种存储管理方式，OS/2等采用这种方式。

Intel 80386有16K个独立段，每个段最大可以有4GB。虽然段的数目较少，但段的大小增大，有利于适应具有较大数据段应用程序的需要。

Intel 80386虚拟存储器段表有局部描述符表（Local Descriptor Table，LDT）和全局描述符表（Global Descriptor Table，GDT）两类。每个进程都有自己的LDT，系统中的所有进程共享一个GDT。LDT中每个表项描述进程的一个段的情况，包括代码段、数据段和堆栈等。GDT描述系统段，包括操作系统本身。

为了访问某个段，Intel 80386将段选择符（Selector）装入计算机的6个段寄存器中的某一个，在运行过程中，CS寄存器保存代码段选择符，DS寄存器保存数据段选择符。

每个段选择符为16位，如图5-24所示。在选择符中，1位指出这个段是局部的，还是全局的（即在LDT中或GDT中）；13位是索引号，因此段描述符中最多有8K个表项；两位用于存储保护。

图5-24 Intel 80386的段选择符

在选择符被装入段寄存器的同时，对应的段描述符被从LDT或GDT中取出装入微程序寄存器，以便快速访问。一个段描述符有64位，结构如图5-25所示。

基址0～15						限长0～15					
基址24～31	G	D	0		限长16～19	P	DPL	S	类型	A	基址16～23

图5-25 Intel 80386的段描述符

（1）段基址。共32位，定义了4GB虚拟地址空间中某段的开始地址。

（2）段限长。规定了段的长度。页的大小为4KB，段长为20位，故最大段长为4GB。

Intel 80386 可以采用两种存储管理方案：如果 G 位为 0，段的长度以字节为单位计算；如果 G 位为 1，段的长度以页为单位计算。

(3) 操作数长 D。仅用于代码段描述符。D 为 1，表示用 32 位描述段长，即最大段长为 4GB；当 D 为 0 时，表示用 16 位描述段长，最大段长为 64KB。

(4) 存在位 P。P 为 0 表示段未调入内存，P 为 1 表示段在内存。

(5) 特权级别 DPL。用于描述段的特权级，共四个级别。当一个进程访问某段时，要具有访问该段的特权才可进行。

(6) 系统段标志 S。当 S=0 时为系统段描述符，当 S=1 时为非系统段描述符。

(7) 类型。对于系统段(S=0)，用于表示段的类型；对于非系统段(S=1)，表示段的存取控制，如本段是可执行、只读或可读/写。

(8) 访问位 A。用于段的置换。

通过段选择符和位移，将逻辑地址转换成物理地址的过程如下。

(1) 系统通过段选择符，定位段描述符。首先，根据段选择符的第 2 位确定是 GDT 或 LDT；然后将段选择符的低 3 位清 0，复制到段寄存器；加上 GDT 或 LDT 的地址，得到一个指向段描述符的指针。

(2) 得到段描述符后，查找段是否在内存(P 位)中，若段不在内存中，会产生一缺段中断。

(3) 检查段的长度是否出界，若出界，则产生越界中断。从逻辑上讲，段的长度应该为 32 位，但实际上只有 20 位，可以使用。

(4) 如果位移没有出界，系统将段描述符中的 32 位段基址和位移相加得到所谓的线性地址。为了和只有 24 位段基址的 Intel 80286 兼容，Intel 80386 的基址被分成三个部分分布在段描述符中。

(5) 如果系统不分页，则得到的线性地址就是内存物理地址，此时的 Intel 80386 系统是纯段式存储管理系统。但如果系统分页，线性地址将被解释为一个 32 位的虚拟地址，并通过页表机构映射为物理地址。

(6) Intel 80386 系统将上述 32 位线性地址分成三个部分，分别为页目录(10 位)、页号(10 位)和页内位移(12 位)。页目录的地址由页表寄存器给出。页目录由 1024 个 32 位的表项组成。页目录中每个表项都指向一个也包含 1024 个 32 位表项的页表。页表中，20 位为页框号，3 位留给系统操作员使用，其他为修改位 D、访问位 A、用户管理位 US(用于页保护)、读写保护位 RW 和存在位 P 等信息，如图 5-26 所示。

页框号	Avail	0	0	D	A	0	0	US	RW	P

图 5-26 Intel 80386 的页表项

一个页表描述 1024 个表项，每个表项描述一页的内存空间，即 4KB。因此一个页表可以处理 4MB 的内存。一个小于 4MB 的段将只有一个表项的页目录，这个表项指向唯一的一个页表。所以短的段的开销只有两个页。

另外，Intel 80386 的联想存储器中有 32 个表项，在每个表项中包含三个数据项，分别为页目录项、页号和存储块号。由于页的大小是 4KB，32 个表项可以覆盖 128KB 的内存。故快表中查询的命中率较高，可达 98%。

如果应用程序不需要分段，只需要分页的 32 位地址空间也是可以的。此时，所有段寄存器被设置成同一个选择符，它的段描述符基址为 0，长度为最大。段位移就是线性地址，这样

就是分页系统了。

所以 Intel 80386 可以实现纯分段、纯分页和段页式存储管理系统，并与 Intel 80286 兼容。其高效的设计实现了所有目标。

思考与练习题

1. 存储管理的基本任务是为多道程序的并发执行提供良好的存储器环境，这包括哪些方面？

2. 页式存储管理系统是否产生碎片？如何应对此现象？

3. 在页式存储管理系统中页表的功能是什么？当系统的地址空间很大时会给页表的设计带来哪些新问题？

4. 什么是动态链接？用哪种存储管理方案可以实现动态链接？

5. 某进程的大小为 25F3H 字节，被分配到内存的 3A6BH 字节开始的地址。但进程运行时，若使用上、下界寄存器，寄存器的值是多少？如何进行存储保护？若使用地址、限长寄存器，寄存器的值是多少？如何进行存储保护？

6. 在系统中采用可变分区存储管理，操作系统占用低地址部分的 126KB，用户区的大小是 386KB，采用空闲分区表管理空闲分区。若分配时从高地址开始，对于作业申请序列“作业 1 申请 80KB、作业 2 申请 56KB、作业 3 申请 120KB、作业 1 完成、作业 3 完成、作业 4 申请 156KB、作业 5 申请 80KB”，试用首次适应法处理上述作业，并回答以下问题。

（1）画出作业 1、2、3 进入内存后内存的分布情况。

（2）画出作业 1、3 完成后内存的分布情况。

（3）画出作业 4、5 进入内存后内存的分布情况。

7. 某系统采用页式存储管理策略，某进程的逻辑地址空间为 32 页，页的大小为 2KB，物理地址空间的大小是 4MB。

（1）写出逻辑地址的格式。

（2）该进程的页表有多少项？每项至少占多少位？

（3）如果物理地址空间减少一半，页表的结构有何变化？

8. 某页式存储管理系统内存大小为 64KB，被分成 16 块，块号为 0、1、2、…、15。设某进程有 4 页，其页号为 0、1、2、3，被分别装入内存的 2、4、7、5 块，回答如下问题。

（1）该进程的大小是多少字节？

（2）写出该进程每一页在内存的起始地址。

（3）逻辑地址 4146 对应的物理地址是多少？

9. 某段式存储管理系统的段表如图 5-27 所示。

段号	段长	段始址
0	15KB	40KB
1	8KB	80KB
2	10KB	100KB

图 5-27　段表

请将逻辑地址[0,137]、[1,9000]、[2,3600]、[3,230]转换成物理地址。