第5章

内 存 管 理

计算机中的程序和数据存放于内存，CPU 执行的每一条指令都要到内存中取，这凸显了内存的重要地位，其存取速度和容量是整个计算机系统性能的关键指标。内存管理成为操作系统的核心内容，其目标就是要提高内存的利用率，为应用程序访问内存提供更方便的方式。

本章讲述连续分配、分段和分页三种典型的内存管理方法，并重点阐述当前广泛应用的分页系统的实现方法。最后介绍虚拟存储器的原理和实现技术。

5.1 连续分配内存管理

连续分配是物理内存的一种管理方法，它将应用程序装入物理内存中的一段连续区域中。当内存中存在多个应用程序时，如何高效使用内存空间、保护应用程序是本节讨论的核心内容。

5.1.1 连续分配

虚拟地址空间说明了从程序员的视角看程序和数据在一个线性的地址序列中是如何安排的。进程运行时，虚拟地址空间中所描述的程序和数据必须装入物理内存，程序中的指令才能装入 CPU 执行，所以操作系统必须在物理内存中为进程分配属于它自己的内存区域。如果操作系统给每个进程分配一个地址连续的内存区域，那么这样的分配方式称为**连续分配**，如图 5.1 所示。否则，虚拟地址空间映射到若干段彼此不相邻的内存区域，则称这样的内存管理方式为**非连续分配**，如图 5.2 所示。可以用基地址和长度来描述进程所占用的连续内存区域，而且该内存区域和虚拟地址空间都是连续的线性空间，地址映射和内存管理都非常简单。相比之下，非连续分配就复杂得多。

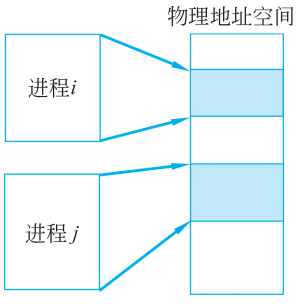


图 5.1 连续分配

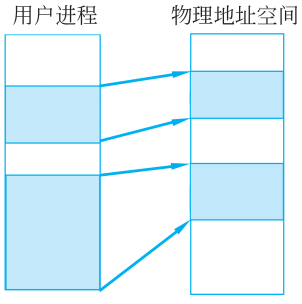


图 5.2 非连续分配

为了能够在内存中同时装下多个作业，操作系统将内存划分成大小不等或大小均等的分区，以达到不同的管理目标。日常生活中我们也这样做，如教学楼里的教室大小不等，数量各不相同，以留给不同大小的教学班，排课的时候需要花些工夫把大班分到大教室。即使人的体积各不相同，教室里所有的椅子大小也都是相等的，目的是减少管理上的代价。不管分区的大小是否相同，在装入用户作业前已经划分好了。这样的内存分区方式称为**固定分区**，固定的意思是分区的大小不会随作业大小而改变。当装入作业时，操作系统总是在比作业大的分区中找一个分区装入作业。一般情况下，装入作业后，每个分区都会留下一部分区域不能被利用，这些区域称为**内碎片**。这些内碎片不能被另外的进程使用，基于管理需求，一个分区只能存放一个进程。这降低了内存利用率，是操作系统努力避免的。

固定分区的缺点就是不能根据作业的大小动态调整分区大小，这会使一些小的作业可能占用了大分区，导致大的作业找不到足够大的分区。因此，一个自然的想法就是事先不对内存进行划分，而是当作业到来时根据作业大小动态(临时)分配一个分区给作业，这样每个作业得到的内存分区大小和作业一样，这样的分区称为**动态分区**。显然，动态分区的方法是不会存在内碎片的，图 5.3 显示了多个进程在内存中的分配情况。然而，新的情况出现了，当进程 P2 退出后，它原先所在的区域 B 成为空闲区。当一个新的进程 P5 到来时，系统会找一个比进程 P5 大的空闲区装入 P5，如找到了空闲分区 B。然后将 B 分成 B1 和 B2，B1 大小与 P5 相同，装入 P5，B2 仍作为空闲区。不过 B2 可能会很小，以至于很难再装下一个进程了，这种不能被利用的空闲区称为**外碎片**。外碎片和内碎片统称为**碎片**，其特点是不能被分配，降低了内存的利用率，还需要内核去管理，增加系统开销。

经过多轮作业的进进出出，内存也经历了多次的分配和回收，其中包含两类分区，一类是已使用的，另一类是空闲的，它们相间分布，大的空闲区可以被再次分配，小的就成为碎片。碎片并不是永恒的，当一个进程结束后，操作系统回收其空间，若其前面的或后面的分区也是空闲的，则可以将它们合并成一个大的空闲区。这样碎片会被吸收掉，其数量不会超过作业分区的数量。

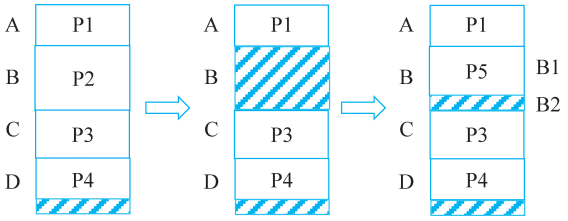


图 5.3　外碎片的产生过程

当一个作业到来时，如果内存中有多个空闲区都能装下该作业，那么应该选择哪个空闲区呢？最常用的策略有以下三种。

(1) 首次适配：所有空闲区一般按地址大小排序，分配内存时，在空闲区从头顺序检索，第一个能装下该作业的空闲区被选中，该方法简单。

(2) 最佳适配：在所有能装下该作业的空闲区中，最小的被选中，该方法尽量留下了大的空闲区，但容易产生碎片。

(3) 最差适配：在所有空闲区中，最大的被选中，该方法最不容易产生碎片，但会破坏大的空闲区。

以上空闲内存的分配策略各有所长,能在不同的应用中发挥作用。存储空间中的碎片是计算机系统中的一个一般性问题,与物理内存的分配一样,在 C/C++ 等高级语言中用 malloc()实现动态内存分配,分配的是虚拟地址空间中的内存区域,文件系统写文件时分配的是磁盘空间。这些分配方法应对的共同问题是:所分配的空间是连续的,需要尽可能留下大的空闲区,提高分配效率,减少碎片。

5.1.2 内存保护

保护是一种机制,用于控制程序、进程或用户对系统中资源的访问,该机制需要说明对访问进行什么样的控制,以及怎样实现这些控制。对于内存访问来说,所谓内存保护就是要做到:用户进程只能以它被允许的方式访问指定的内存区域。需要注意的是,系统不仅要限制进程访问系统或其他用户的内存区域,而且会限制进程访问自己的内存区域。例如,进程对代码区域的写操作就会被限制。

关于保护,制定一套原则贯穿系统的设计和使用,对于维护整个系统的一致性是非常重要的。最低权限原则就是一个经过时间检验的关键原则,其宗旨是:程序仅被授予完成任务所必需的权限,不能有多余的权限。

连续分配的内存保护非常简单,因为进程能访问的内存区域仅在一个分区内,所以只要用上限和下限两个值就可以描述进程的内存资源。一般情况下,CPU 中会有一对上、下限寄存器,分别记录当前运行在 CPU 上的进程在内存中分区的最高和最低地址。在访问内存时,如果地址超出了上、下限寄存器规定的范围,系统就会产生地址越界异常。由于采用连续分配的系统一般都是早期或小型系统,内存操作的种类往往简单地分为“允许”和“禁止”。

关于地址变换的实现,可以参阅 1.4.2 节的内容,如图 1.12 所示,应用程序使用的是虚拟地址(也称逻辑地址),而访问物理内存时需要使用物理地址,所以在程序运行前,需要将程序中的虚拟地址全部转换为物理地址。在计算机系统发展早期,在程序运行之前通过软件方法实现重定位,称为**静态重定位**。与静态重定位不同,**动态重定位**并不在程序运行之前修改程序中的地址,而是采用处理器中的内存管理器(Memory Management Unit,MMU),在指令执行过程中进行重定位。可见,动态重定位是由硬件完成的。图 1.14 显示了动态重定位的过程,CPU 从指令中解析出虚拟地址后,将虚拟地址送给 MMU。MMU 中有限长寄存器 limit 和重定位寄存器 relocation,限长寄存器存放程序的长度,重定位寄存器存放程序在物理内存中的起始地址。当然,也可以采用上限寄存器和下限寄存器,其所起的作用与限长寄存器和重定位寄存器类似,都是定义程序的访问区域。

如果系统采用静态重定位机制,那么操作系统就要承担静态重定位的功能,具体操作应该是装入程序完成的。如果系统采用动态重定位机制,重定位是在指令执行过程中完成的,那么重定位操作就是由硬件的 MMU 完成的而不是软件完成的。不过,操作系统仍然从两方面掌控动态重定位的过程:一是重定位寄存器和限长寄存器的值是由操作系统在程序运行前就设置好的,它们在重定位过程中发挥关键作用;二是动态重定位过程中一旦出现地址越界异常,CPU 的控制权就会转向操作系统,由操作系统处理越界异常。

5.1.3 交换

内存容量是有限的,很容易就被进程占满了,尤其是在多任务系统中。当运行一个新的进程时,系统可能发现已经没有内存可用了,最直接的方法就是将当前不执行的任务暂时从

内存中换出到磁盘上，以后执行该进程时，再把它换回内存，这种机制称为**交换**。

交换是以进程为单位的，即进程整个地换进/换出。交换实际上就是第 4 章 CPU 调度部分介绍的内存调度（中期调度），它决定了哪些进程应该在内存，哪些进程应该去外存。与交换不同但又非常类似的是虚拟存储技术，以进程的部分存储空间为单位换进/换出，进程执行到哪一部分再把哪一部分装入内存，提供了更加灵活的换进/换出机制。尽管都采用了内外存交换技术，然而，前者是计算机系统早期的内存管理技术，着眼于为所运行的进程腾出内存空间，后者的目标是建立一种存储机制，以透明的方式为系统提供高速、大容量的虚拟存储器。

交换需要将内存的大量数据写入磁盘，为了降低磁盘读写时间，会在磁盘中开辟专门的存储空间存放换出去的进程。即使这样，每次进程换进/换出的时间是以毫秒为单位的，所以，系统应当尽量减少交换，尤其是要杜绝那些刚换出去的进程很快又要执行的情况。例如，如果一个进程执行 I/O 而阻塞，被换出内存，结果很快用户就完成输入，又得执行该进程，这样的交换就是折腾。为了较少交换，操作系统仅把那些很有把握长时间不再运行的进程换出去，例如，在 Windows 3.1 中，仅当一个程序长期没有用户交互时，如 Word 窗口长期未激活，才会被系统换出内存。

目前，纯粹的交换技术已经很少在现代操作系统中使用，但交换的思想和改进已经融合进现代虚拟存储技术中来。

在采用连续内存分配的系统中，进程的程序和数据在物理内存空间中是连续存放的，地址变换过程和内存保护都很容易做到。然而，该方法的缺点是不能实现进程间内存共享，也不能在该方法的基础上实现虚拟存储器，这也要归因于其连续性。

5.2　分段内存管理

采用连续分配方法时，假定应用程序占用一段连续的虚拟地址空间，所以装入内存时也会寻找一段连续的物理内存空间。事实上，尽管程序的语句或指令是按线性顺序排列的，但其宏观的逻辑结构并不是线性连续的，而是模块化的，例如，子程序之间、目标模块之间并没有顺序之分，也不要求彼此之间是连续存放的。内存单元是按顺序排列的，所以采用机器级语言编程序时，程序员会把程序模块安排到一个统一的连续的线性地址空间中；采用高级语言编程时，则由编译程序安排程序模块在地址空间中的位置。图 5.4 显示了程序的各个模块在一维地址空间中的存储映像，而图 5.5 显示了程序的逻辑视图。如果允许程序员把程序模块挂在树上，他是不会把程序模块连续地排成一行的。强行把程序模块连续地存放在内存，不仅没必要，而且会限制程序的运行，例如，程序中栈模块和堆模块都是动态增长的，其相邻的模块会阻碍它们的增长。相对而言，树结构允许每一根树枝都可以自由生长。

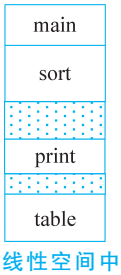


图 5.4　线性空间中的程序

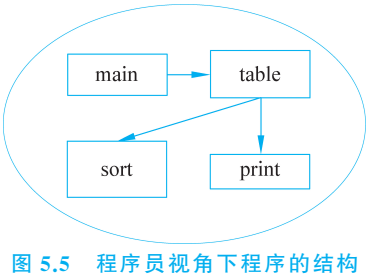


图 5.5　程序员视角下程序的结构

另外,内存连续分配管理机制已经为我们提供了管理内存中多个不相邻分区的方案,所以,把应用程序中的不同模块装入内存的不同分区,也就是这些模块在物理内存中不必相邻,就成为一种自然的选择。这样,操作系统对进程管理的粒度就细化到进程内的各个模块,操作系统可以为某个模块分配单独的分区,甚至单独装入某个模块。

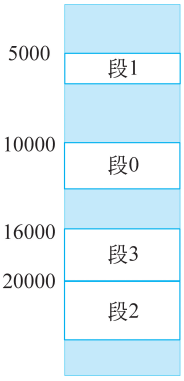
5.2.1 分段的基本方法

段(Segment)是应用程序中具有逻辑意义并作为内核管理进程地址空间的一个独立单元,它可以是一个目标模块、程序库、一组数据、栈或堆。段内的代码或数据形成线性存储空间,按地址访问。每个段有一个段号,作为段的标识。用高级语言编程时,程序员可以通过[段名,变量名]访问内存,经过编译后,在机器语言中以[段号,段内偏移量]访问内存,可见程序中的逻辑地址是二维的。在 C 语言中,仅使用变量名就可以访问全局变量,并没有使用段名,那是因为 C 语言不允许全局变量重名,编译程序完全可以根据变量名确定它在哪个段中定义;同样在很多汇编语言中也可以不使用段号,而是使用段寄存器,段寄存器和段号的作用是一样的,都可以作为段的标识,这都是一种变通。**分段**是以段为单位进行内存分配和管理的内存管理模式,不同的是,连续分配模式则把全部应用程序看作一个整体进行内存分配。

程序运行前,内核依次将应用程序的所有段装入内存,同时,还会建立一个表格,记录每个段在内存的位置,该表称为**段表**。段表如图 5.6(a)所示,它描述了如图 5.6(b)所示程序中各段在内存中的位置。每个进程都有一个段表,不同的进程,其段的数量不一样,段表的长度也不同。一般段表都是放在内存中的。

段号	段基址	段长
0	10000	3000
1	5000	1600
2	20000	5000
3	16000	4000

(a) 段表



(b) 各段在物理内存中的分布

图 5.6 应用程序中各段在内存中的存储映像

采用分段模式后,应用程序的各段分散在内存的不同位置,CPU 在执行程序时是如何找到要访问的数据的呢?这就要依靠段表来进行地址变换了。假如要执行的指令是“LOAD R, <2>|[2100]”,也就是要访问第 2 段偏移地址为 2100 的内存单元。这里“<2>|[2100]”是逻辑地址,得到物理地址的过程如下:用段号 2 去查段表,得到第 2 段在物理内存的起始地址 20000,然后加上偏移量 2100,得到物理地址 22100。

可以把上述地址的计算一般化为如图 5.7 所示的地址变换过程。CPU 从指令中解析出逻辑地址 s|d,其中,s 为段号,d 为段内位移,然后将逻辑地址交给内存管理单元

(MMU),由 MMU 实现逻辑地址到物理地址的变换。MMU 需要用到两个寄存器:段表基地址寄存器指向段表的起始地址;段表长度寄存器记录段表的长度。MMU 的地址变换过程主要包含 4 个操作:①比较段号 s 和段表长度寄存器,判断段号 s 是否超出了段长,若是,则产生异常;②通过段表基地址寄存器,用段号 s 查段表得到段基址和段长;③判断指令中给出的段内偏移量 d 是否小于段长,若不小于,则引发异常;④将段基址和偏移量相加,得到物理地址。需要注意的是,地址变换过程是指令周期的一部分,是由硬件实现的。在这个过程中,段表具有核心的地位,操作系统通过建立段表,在地址变换模式中起到了主导作用,但它并没有直接参与每一次的地址变换。分段机制的地址变换方法是基于动态重定位方法的,所不同的是,分段机制中每个应用程序都有若干分区,但是系统中并没有多个重定位寄存器,所以,段表就起到了一组重定位寄存器的作用,每个表项相当于一个重定位寄存器。

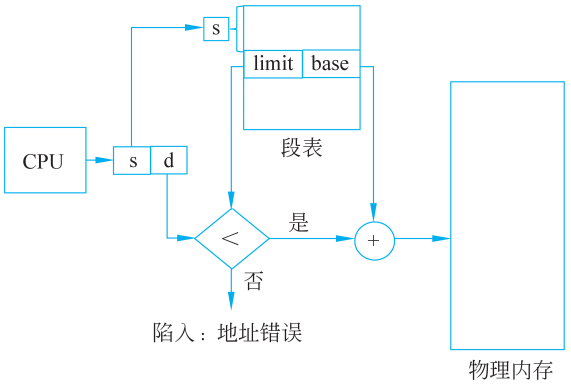


图 5.7　分段地址变换过程

采用分段模式后,操作系统的内存管理功能可以细化到应用程序的各个模块,从而可以实现对进程存储空间更精确的管理和控制,进一步实现用户进程之间的共享和保护;同时由于应用程序没必要占用连续的内存区域,也就没必要一次性装入内存,为实现虚拟存储器奠定了基础。与连续内存管理模式一样,系统会建立空闲分区表,记录内存中的可利用区域。当装入程序时,系统根据某种适配策略依次为每个段分配内存;不同的是,一个分区仅存放进程的一个段,而不是整个进程。段占用的空间比作业更小,原先不能被利用的内存碎片,也可以利用起来了,然而这也使得内存碎片变得更小、更多,分段并没有从根本上解决碎片问题。

5.2.2　段的共享

由于系统中的进程都在同一个物理内存中,操作系统一方面要限制进程对内存的访问,防止进程访问其权限之外的存储空间,这在前面已有介绍;另一方面操作系统还要为进程之间共享内存空间提供支持,例如,多个进程可以使用内存中同一份数据,调用同一组子程序,这将大大提高内存资源的利用率。

进程之间要共享的内容一般是一个数组、一个子程序、一个模块等,这些都是程序的逻辑单元,在程序中都可以段的形式存在。所以,进程之间的内存共享可以理解为进程之间段的共享。进程中的段,有的可以被其他进程分享,有的不能;有的只能读,有的则可以读写。操作系统必须对访问这些段的用户验证其所具有的权限。分段正好满足了这一要求,段作

为操作系统管理的应用程序的基本单位,每个段有独立的访问权限,可以分别处理。系统一般在段表中增加另外的字段,描述对段操作的读写权限,例如,是否可读、是否可写等。回顾一下,连续内存分配将整个进程作为一个不可分割的整体,不能对各部分分别管理,因而不可能支持共享。

假如用户要打开两个 Word 文件 A 和 B,双击第一个文件 A 后,Word 程序和文件 A 都会装入内存,同时创建进程 1,进程 1 的段表以及各段在物理内存的位置如图 5.8 所示,0 号段是 Word 代码段,1 号段是存放文件 A 的数据段。如果此时再同时打开另一个文件 B,那么文件 B 也将装入内存,系统又会创建进程 2。由于进程 2 用到的 Word 程序已经在进程 1 运行时装入内存,系统没有必要在内存中再次装入 Word 代码,进程 2 可以和进程 1 共享 Word 代码。共享的实现非常简单,假设 Word 代码段在进程 2 中的段号为 1,只要将进程 1 的段表中代码段的起始地址和长度复制给进程 2 段表中的代码段即可,当然,进程 2 的数据段仍需新建,如图 5.8 所示。依据前面介绍的地址变换规则,进程 1 访问代码段 0 和进程 2 访问代码段 1,访问的都是内存中 20000 号单元开始的 Word 代码,从而实现了代码共享。

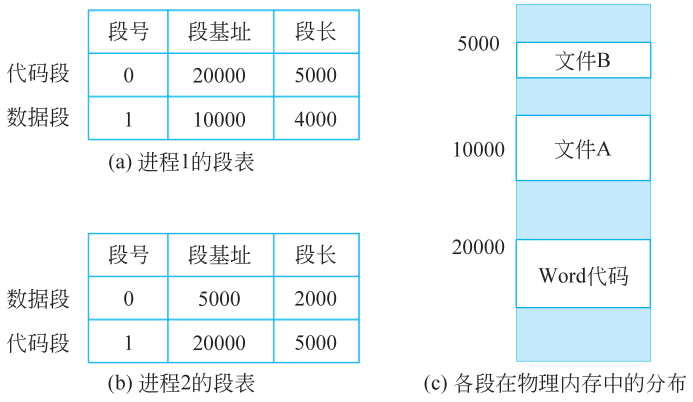


图 5.8 两个进程共享代码

如果两个进程 A 和 B 要共享的代码自身带有数据部分,那么这些数据显然也是两个进程需要共享的。例如,进程 A 和 B 要共享的代码 S 有一个计数器变量 count,初始化为 0,进程 A 在执行 S 时已经将 count 的值累加到 3 后暂停,CPU 又去执行进程 B。进程 B 将 count 重新初始化,显然,当进程 A 继续执行 S 时就会出错。出错的原因在于:进程 A 和 B 在执行共享代码 S(包含数据部分)时都要对 S 进行修改,导致两个进程相互干扰。所以,一段代码要想被多个进程共享,它自身就不允许包含被修改的部分,这样的代码称为**可重入代码**。一般代码是不能被修改的,如果一段程序自身不包含数据部分,则称为**纯码**。显然,纯码是可重入代码,是可以被共享的。

共享代码在执行时修改寄存器、局部变量,不会在进程之间造成相互干扰,因为这些数据都会保存在进程控制块或进程自身的栈中。共享代码若修改进程地址空间中非共享的数据部分,也不会造成进程之间的相互干扰。如果共享代码一定要修改进程之间可共享的数据,那么需要在编码时充分考虑进程之间的同步和互斥关系,这将在第 8 章中介绍。

共享和保护是不可分割的,内存共享包括读、写、执行等,需要限定操作的权限,分段模式下的保护的实现方法与分页模式类似,将在 5.3 节中详细介绍。

5.2.3　内存共享的程序实例

5.2.2 节介绍了进程之间共享代码或数据在内核层面的实现原理，现在来说明在应用程序层面需要做的工作，以相互印证。

要做到共享代码，系统需要知道共享什么样的代码，例如，5.2.2 节中的编辑程序 Word，这实际上是在应用程序编译、连接时就确定了的。共享程序往往以共享库的形式存在，所以一个应用程序调用共享代码时，编译、连接阶段就会加入共享库，如 Windows 系统中的动态连接库 DLL。当应用程序调用共享库代码时，必定请求操作系统装入共享库，所以系统由此知道哪些进程之间共享哪些代码。

同样，数据共享也需要应用程序告诉操作系统哪些进程之间共享哪些数据。下面的例子将说明这一点。

在 UNIX 操作系统中，两个进程要共享内存，需要先由一个进程建立一个共享内存对象，并赋予一个名字，然后另一个进程用同一个名字打开这个内存对象，就可实现二者之间对该内存对象的共享了。注意，建立和打开共享内存对象都是用 shm_open()，只是所给的参数不一样。由于进程只能直接访问自己地址空间的存储单元，所以进程在访问共享内存对象之前需要先将它映射到各自的虚拟地址空间中，这由 mmap()完成。使用完共享内存之后，要使用 shm_unlink()删除之，以断开两个进程之间的联系。

建立并将数据写入共享内存的程序如下。

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main()
{
  const int SIZE 4096;                                  //共享内存区的大小
  const char * name = "shm_example";                    //共享内存区的名字
  const char * message = "Hello World!";
  int shm_fd;                                           //共享内存区文件描述符
  void * ptr;                                           //指向共享内存区的指针
  shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);      //建立共享内存
  ftruncate(shm_fd, SIZE);                              //设定共享内存的大小
  ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);     //映射到地址空间中
  sprintf(ptr,"%s",message);                            //将 Hello World!写入共享内存
  return 0;
}
```

打开共享内存对象并显示其中数据的程序如下。

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
```

```
#include <sys/stat.h>
int main()
{
  const int SIZE 4096;
  const char * name = "shm_example";
  int shm_fd;
  void * ptr;
  shm_fd = shm_open(name, O_RDONLY, 0666);                    //打开共享内存对象
  ptr = mmap(0, SIZE, PROT READ, MAP SHARED, shm_fd, 0);      //映射到地址空间中
  printf("%s",(char *)ptr);                                   //显示共享内存中的数据
  shm_unlink(name);                                           //断开共享内存对象
  return 0;
}
```

为节省篇幅,上述程序并未对所调用的函数返回值进行判断,只是演示了关键函数调用的逻辑关系。完整的程序可参考 Linux 手册中关于 shm_open()的示例程序。

5.3 分页内存管理

分段内存管理机制中,尽管段之间可以不连续存放,但段存储区域的分配仍然采用了连续内存分配的管理方法,系统中仍然有内存管理的碎片问题。碎片是容易理解但并不容易处理的,在日常生活中也普遍存在,例如,散装货轮不停地装货、卸货。卸货之后空出的船舱会被再次利用,但难以保证会被全部利用,因为新装载的货物不一定正好占满上批货物腾出的空间,所以也会出现“碎片”。最典型的解决之道是采用集装箱,将船舱设计成标准的集装箱箱位,而货物都装入标准大小的集装箱,这样,船舱中的所有空间都是可利用的。分页内存管理模式正是借鉴了上述思想。

5.3.1 基本方法

1. 分页

分页内存管理模式于 1959 年首次提出,并在 1962 年首先应用于曼彻斯特大学研发的 Atlas 计算机上。在分页管理系统中,内存物理地址空间和应用程序的虚拟地址空间按照同一尺度,例如 1KB,进行划分,在物理地址空间中划分出来的每个区域称为**页框**,而虚拟地址空间中划分出来的每个区域称为**页**,一个页正好可以装入一个页框。

内核将应用程序中的页装入内存时,需要先为这些页分配空闲页框,并不要求这些页框必须是位置上连续的。同时,还会为每个进程建立一个**页表**,记录每个页装到了哪个页框中,如图 5.9 所示。虚拟地址空间中相邻页在物理地址空间中并不一定相邻,所以需要通过页表记住每个页在内存中所对应的页框号。页表和段表的作用是类似的,都是实现虚拟地址空间到物理地址空间的映射。不过,页表和段表中的内容还是不同的,要注意它们之间的相似和不同。

页表需要为每一个页建立到页框的映射关系,所以页表的大小是由虚拟地址空间的大小决定的。而地址空间的大小是由内存地址的所占位数决定的,例如,在 Intel x86 32 位系统中,用 32 位二进制表示地址,假如页的大小为 4KB,页内偏移量占 12 位,那么虚拟地址空

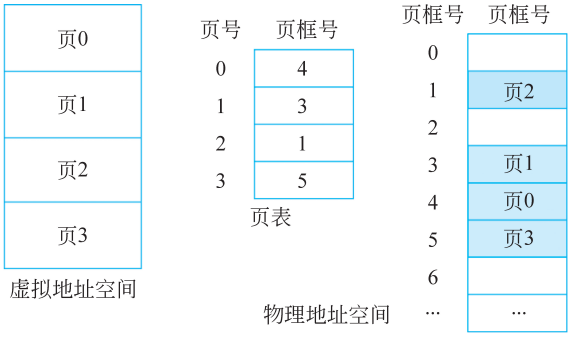


图 5.9　虚拟地址空间与物理地址空间的关系

间中页的数量(页表的长度)就是 1M。系统的虚拟地址空间很大,是应用程序理论上能访问的最大范围,事实上,它比应用程序本身大得多,绝大部分是空洞,没有必要将这些区域映射到物理内存中。这样,为了节省页表开销,在有的系统中页表仅用于记录那些应用程序所在的页对应的页框,在这种情况下,页表的大小实际上是由应用程序的大小决定的。在这样的系统中,可以采用一个页表长度寄存器,专门记录当前程序在虚拟地址空间所占的页数,即页表的长度。除此之外,后面还会介绍多级页表的方法来减少页表的开销。

页表中所存放的映射关系,无论是关于整个虚拟地址空间的,还是仅关于其中的非空洞部分(程序和数据),其所占的存储空间都不会小,只能存放在内存中。在 CPU 中有一个专门的寄存器,记录页表存放在内存中的起始位置,称为**页表基地址寄存器**(PTBR)。每个进程都有自己的页表,所以在 CPU 切换时,如果当前线程与新线程不属于同一进程,那么需要更新页表基地址寄存器的内容。

2. 地址变换

在分页模式下,虚拟地址是一维的、线性的,不像分段模式,程序中的地址并不会直接体现页号和页内地址。事实上,CPU 是可以依据虚拟地址直接计算出页号和页内地址的。假设页的大小为 2^nB,那么虚拟地址对应的页号 p = 虚拟地址/2^n,页内地址 d = 虚拟地址 mod 2^n。由于虚拟地址是以二进制形式表示的,上述计算只是说明页号和页内偏移的意义,除法运算和模运算都无须做,页号就是虚拟地址中第 n 位及其左侧部分,而页内地址就是虚拟地址中的 0～n－1 位。当然,这也要求在系统设计时页的大小一定要求必须是 2 的整数次幂才行。

如何把虚拟地址转换为物理地址是所有内存管理模式要解决的核心问题。图 5.10 说明了分页模式的地址变换过程,这是由内存管理单元(MMU)(虚线框内)完成的,其中包含 4 个关键的操作:①从虚拟地址中分离出页号 p 和内页地址 d;②根据页表基址寄存器的内容,得到页表在内存中的起始地址;③页表起始地址加上页号 p,就可定位到页表的第 p 个页表项,并从中得到页框号;④由于页和页框大小相同,虚拟地址在页内的偏移量和相应物理地址在页框内的偏移量是相同的,所以,直接将页框号和页内地址拼装成物理地址。由此可见,分页模式的地址变换过程与分段非常相似。需要强调的是,尽管地址变换过程是由 MMU 硬件完成的,但页表的内容是操作系统来填写的,即映射关系是内核确定的。

在地址变换过程中,MMU 会使用页表基址寄存器 PBR 访问页表,PBR 中存放的页表基址属于物理地址,不能再进行地址变换了,而是直接送到地址总线上,这一点,MMU 是不