第5章 内存管理

内存是计算机中一种需要认真管理的重要资源。就目前来说，虽然一台普通家用计算机的内存容量已经是20世纪60年代早期全球最大的计算机IBM 7094的内存容量一万倍以上，但是程序大小的增长速度比内存容量的增长速度要快得多。正如帕金森定律所指出的："不管存储器有多大，程序都可以把它填满"。

每个程序员都梦想拥有这样的内存：它是私有的、容量无限大的、速度无限快的、价格低廉的，并且是永久性的存储器。遗憾的是，目前的技术还不能为我们提供这样的内存。

除此之外的选择是什么呢？经过多年探索，人们提出了"分层存储器体系"的概念，即在这个体系中，计算机有若干兆字节(MB)快速、昂贵且易失性的高速缓存，若干吉字节(GB)速度与价格适中且同样易失性的内存，以及若干太字节(TB)低速、廉价、非易失性的磁盘存储，另外还有诸如DVD和U盘等可移动存储装置。操作系统的工作是将这个存储体系抽象为一个有用的模型并管理这个抽象模型。

操作系统中管理分层存储体系的部分称为存储管理器。它的任务是有效地管理内存，即记录哪些内存是正在使用的，哪些内存是空闲的；在进程需要时为其分配内存，在进程使用完后释放内存。

5.0 问题导入

在现代操作系统中同时有多个进程在运行，每个进程的程序和数据都需要放在内存中，那么程序员在编写程序时是否需要知道程序和数据的存放位置呢？如果不知道，那么多个进程同时在内存中运行，每个进程应占用哪些空间呢，如何保证各个进程占用的空间不冲突呢？内存空间如何进行分配和管理呢？

5.1 内存管理概述

5.1.1 存储结构

在现代计算机系统中，存储部件通常是采用层次结构来组织，以便在成本、速度和规模等诸因素中获得较好的性价比。现代通用计算机的存储层次至少应具有三级：最高层为CPU寄存器，中间层为主存，最底层为辅存。在较高档的计算机中，还可以根据具体的功能分工细化为寄存器、高速缓存、主存储器、磁盘缓存、磁盘、可移动存储介质等。如图5-1所示，在存储层次中越往上，存储介质的访问速度越快，价格越高，相对存储容量越小。对于不同层次的存储介质，由操作系统进行统一的管理。其中，寄存器、高速缓存、主存储器和磁盘

缓存均属于操作系统存储管理的管辖范畴,掉电后它们存储的信息不再存在。固定磁盘和可移动存储介质属于设备管理的管辖范畴,它们存储的信息将被长期保存。而磁盘缓存本身并不是一种实际存在的存储介质,它依托于固定磁盘,提供对主存储器存储空间的扩充。

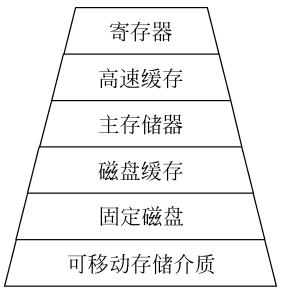

图 5-1　存储器层次结构

主存储器(简称内存或主存)是计算机系统中的一个主要部件,用于保存进程运行时的程序和数据,也称为可执行存储器,目前其容量一般为数十 MB 到数 GB,而且容量还在不断增加,而嵌入式计算机系统一般仅有几十 KB 到几 MB。CPU 的控制部件只能从主存中取得指令和数据,数据能够从主存读取并将它们装入到寄存器中,或者从寄存器存入到主存中。CPU 与外围设备之间交换的信息一般也依托主存地址空间。由于主存的访问速度远远低于 CPU 的执行速度,为缓和这一矛盾,计算机系统引入了寄存器和高速缓存。

寄存器访问速度最快,完全能与 CPU 协调工作,但价格昂贵、容量小,一般以字(Word)为单位。一个计算机系统可能包括几十个甚至上百个寄存器,而嵌入式计算机系统一般仅有几个到几十个寄存器,其用于加速存储访问速度,如用寄存器存放操作数,或用作地址寄存器加快地址转换速度。

高速缓存(Cache)是现代计算机结构中的一个主要部件,其容量大于寄存器,从几十 KB 到几 MB,访问速度快于主存。

5.1.2　内存管理的目标

对于管理来说,我们要问的第一个问题当然是内存管理要达到的目标或目的。

内存管理就是要提供一个虚幻的景象,就像钱,虚的东西,实际上一文不名,但是大家认为它有这个价值。一个东西的价值在于能否满足我们的渴望和需要。如果能,这个东西就有价值。那么内存管理就是要提供一个有价值的虚幻。用术语来说就是抽象。那么内存管理要提供哪些抽象呢?或者说,内存管理要达到什么目标呢?

首先,由于多道程序同时存放在内存中,操作系统要保证它们之间互不干扰。所谓的互不干扰就是一个进程不能随便访问另一个进程的地址空间。这是内存管理要达到的第一个目标。

那还有没有别的目标呢?我们看一下程序指令执行的过程。程序指令在执行前加载到内存,然后从内存中将一条条指令读出,执行相应的操作(从硬件层来看,指令的"读取-执行"循环是计算机的基本操作)。每条指令在执行时需要读取操作数和写入运算结果。要读取操作数,就需要给出操作数所在的内存地址,这个地址不能是物理主存地址。这是因为该程序在何种硬件配置的机器上运行并不能事先确定,操作系统自然不可能对症下药地发出对应于某台机器的物理主存地址。因此,指令里面的地址是程序空间(虚拟空间)的虚拟地址(程序地址)。即程序发出的地址与具体机器的物理主存地址是独立的。这是内存管理要达到的另外一个目标。

综上所述,内存管理要达到如下两个目标。

(1) 地址保护。一个程序不能访问另一个程序的地址空间。

(2) 地址独立。程序发出的地址应与物理主存地址无关。

这两个目标就是衡量一个内存管理系统是否完善的标准。它是所有内存管理系统必须提供的基本抽象。当然，不同的内存管理系统在此二者之上还提供了许多其他抽象。本书将在后面论及这些抽象时逐一说明。

5.1.3 操作系统在内存中的位置


操作系统在内存中的位置

从根本上来说，计算机里面运转的程序有两种：管理计算机的程序和使用计算机的程序。正如本书前面多次提到，操作系统就是管理计算机的程序。而管理者本身也需要使用资源。其中的一个资源就是内存空间。内存管理的第一个问题是操作系统本身在内存中的存放位置。应该将哪一部分的内存空间用来存放操作系统呢？或者说，如何将内存空间在操作系统和用户程序之间进行分配呢？

最简单的方式就是将内存划分为上下两个区域，操作系统和用户程序各占用一个区域，如图 5-2 所示。

图 5-2 仅有 RAM 时操作系统与用户程序的内存分配

比较起来图 5-2(a)的构造最容易理解。因为操作系统是为用户提供服务的，在逻辑上处于用户程序之下。将其置于地址空间的下面，符合人们的惯性思维。另外，操作系统处于地址空间下面还有一个实际好处：就是在复位、中断、陷入等操作时，控制移交给操作系统更方便，因为操纵系统的起始地址为 0，无须另行记录操作系统所处的位置，程序计数器清零就可以了。清零操作对于硬件来说非常简单，无须从总线或寄存器读取任何数据；而图 5-2(b)的布置虽然也可以工作，但显然与人们习惯中操作系统在下的惯性思维不符。

当然，除了上述两种分配方式外，如果愿意，也可以将操作系统和用户程序分拆，形成穿插的分配方式。只不过这样做没有半点好处，白白增加管理的复杂性。

由于现代的计算机内存除了 RAM 之外，可能还备有 ROM。而操作系统既可以全部存放在 ROM 里，也可以部分存放在 ROM 里，这样又多出了两种分配方式，如图 5-3 所示。

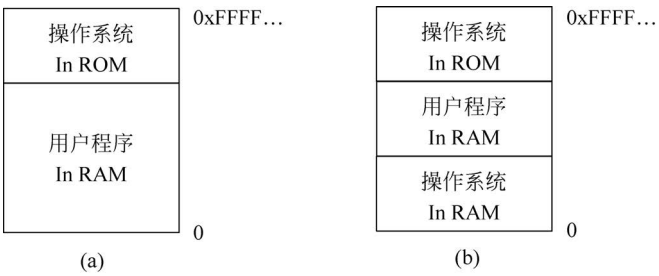

图 5-3 备有 ROM 时操作系统和用户程序之内存分配

图 5-3(a)模式下操作系统放在 ROM 里面的好处是不容易被破坏，缺点是 ROM 要做得大，能够容纳整个操作系统。由于 ROM 比较贵，通常情况下是备有少量的 ROM，只将操

作系统的一部分放在ROM里,其余部分放在RAM里,因此,这两种分配模式以图5-3(b)为佳。

图5-3(b)分配模式还有另外一个好处:可以将输入/输出和内存访问统一起来。即将输入/输出设备里面的寄存器或其他存储媒介编入内存地址(在用户程序地址之上),使得访问输入/输出设备如同访问内存一样。这种输入/输出称为内存映射的输入/输出。如果要访问的地址高于RAM的最高地址,则属于I/O操作,否则属于正常内存操作。

这样,根据操作系统是否占用ROM或是否采用内存映射的输入/输出来分,存在以下两种模式。

(1) 操作系统占用RAM的底层,用户程序占用RAM的上层。

(2) 操作系统占用RAM的底层和位于用户程序地址空间上面的ROM,用户程序位于中间。

第二种模式又分为以下三种情况。

(1) 没有使用内存映射的输入/输出,ROM里面全部是操作系统。

(2) 使用了内存映射的输入/输出,ROM的一部分是操作系统,另一部分属于I/O设备。

(3) 使用了内存映射的输入/输出,ROM全部属于I/O设备。

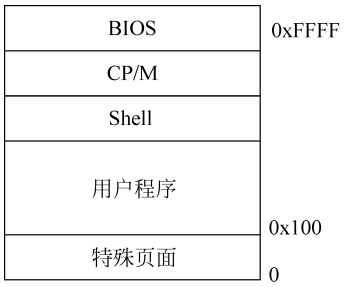

图5-4 CP/M操作系统内存布局

例如,CP/M操作系统的内存布局模式就是上述第一种情况,其BIOS和CP/M内核均处于ROM里面,而Shell和用户程序处于RAM里,如图5-4所示。

CP/M是微计算机控制程序(Control Program for Microcomputers)的缩写,它是一个运行在Intel 8080和Intel 8085微机上的早期操作系统。

多数现代操作系统采用的是第二种模式:即ROM里面包括操作系统一部分和I/O,RAM里面则包括操作系统的其他部分和用户程序。Solaris 10操作系统采用的则是图5-3(b)模式,即操作系统在上面,用户程序在下面。

5.1.4 虚拟内存的概念

虚拟内存的概念听上去有点太虚拟,但其实质则并不难理解。我们知道,一个程序如果要运行,必须加载到物理主存中。但是,物理主存的容量非常有限。因此,如果要把一个程序全部加载到物理主存,则我们所能编写的程序将是很小的程序。它的最大容量受制于主存容量(还要减去操作系统所占的空间和一些临时缓存空间)。另外,即使我们编写的每个程序的尺寸都小于物理主存容量,但还是存在一个问题:主存能够存放的程序数量将是很有限的,而这将极大地限制多道编程的发展。

如何解决物理主存容量偏小的缺陷呢?最简单的办法就是购买更大的物理主存。而这将造成计算机成本的大幅飙升,可能很多人都会买不起计算机。

有没有办法在不增加成本的情况下扩大内存容量呢?有,这就是虚拟内存。

虚拟内存的中心思想是将物理主存扩大到便宜、大容量的磁盘上,即将磁盘空间看作主存空间的一部分。用户程序存放在磁盘上就相当于存放在主存内。用户程序既可以完全存

放在主存，也可以完全存放在磁盘上，当然也可以部分存放在主存、部分存放在磁盘。而在程序执行时，程序发出的地址到底是在主存还是在磁盘则由操作系统的内存管理模块负责判断，并到相应的地方进行读写操作。事实上，可以更进一步，将缓存和磁带也包括进来，构成一个效率、价格、容量错落有致的存储架构。即虚拟内存要提供的就是一个空间像磁盘那样大、速度像缓存那样高的主存储系统，如图 5-5 所示。而对程序地址所在位置（缓存、主存和磁盘）的判断则是内存管理系统的一个中心功能。

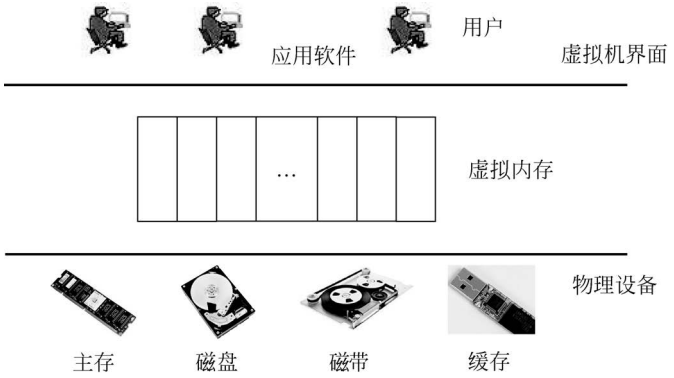

图 5-5　虚拟内存所提供的抽象

值得特别指出的是，虚拟内存是操作系统发展史上的一个革命性突破（当然，它也是使操作系统变得更加复杂的一个主要因素）。因为有了虚拟内存，我们编写的程序从此不再受尺寸的限制（当然还受制于虚地址空间大小的限制）。

虚拟内存除了让程序员感觉到内存容量大大增加之外，还让程序员感觉到内存速度也加快了。这是因为虚拟内存将尽可能从缓存满足用户访问请求，从而给人以速度提升了的感觉。从这个角度来看，虚拟内存就是实际存储架构与程序员对内存的要求之间的一座桥梁。

当然，容量增大也好，速度提升也好，都是虚拟内存提供的一个幻象，实际上并不是这么回事儿，但用户感觉到是真的，这就是魔术。操作系统的一个角色就是魔术师。

5.2　内存管理的基础

当研究与内存管理相关的各种机制和策略时，清楚内存管理的基础内容是非常有用的。内存管理的基础需求如下。

（1）重定位。

（2）保护。

（3）共享。

（4）逻辑组织。

（5）物理组织。

重定位、保护与共享

5.2.1　重定位

在多道程序设计系统中，可用的内存空间通常被多个进程共享。通常情况下，程序员并

不能事先知道在某个程序执行期间会有其他哪些程序驻留在内存中。此外还希望通过提供一个巨大的就绪进程池,能够把活动进程换入或换出内存,以便使处理器的利用率最大化。一旦程序被换出到磁盘,当下一次被换入时,如果必须放在和被换入前相同的内存区域,那么这将会是一个很大的限制。为了避免这种限制,需要把进程重定位到内存的不同区域。

因此,事先不知道程序将会被放置到哪个区域,并且必须允许程序通过交换技术(Swapping)在内存中移动。这关系到一些与寻址相关的技术问题,如图5-6所示。该图描述了一个进程映像。为简单起见,假设该进程映像占据了内存中一段相邻的区域。显然,操作系统需要知道进程控制信息和执行栈的位置,以及该进程开始执行程序的入口点。由于操作系统管理内存并负责把进程放入内存,因此可以很容易地访问到这些地址。此外,处理器必须处理程序内部的内存访问。跳转指令包含下一步将要执行的指令的地址,数据访问指令包含被访问数据的字节或字的地址。处理器硬件和操作系统软件必须能够通过某种方式把程序代码中的内存访问转换成实际的物理内存地址,并反映程序在内存中的当前位置。

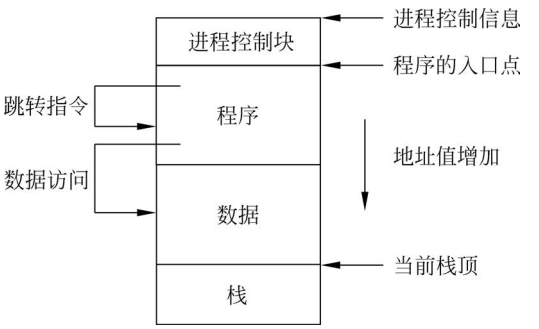

图5-6 进程在寻址方面的需求

5.2.2 保护与共享

每个进程都应该受到保护,以免被其他进程有意或无意地干涉。因此,该进程以外的其他进程中的程序不能未经授权地访问(进行读操作或写操作)该进程的内存单元。在某种意义上,要满足重定位的需求增加了满足保护需求的难度。由于程序在内存中的位置是不可预测的,因而在编译时不可能检查绝对地址来确保保护。并且,大多数程序设计语言允许在运行时进行地址的动态计算(例如,计算数组下标或数据结构中的指针)。因此,必须在运行时检查进程产生的所有内存访问,以确保它们只访问了分配给该进程的内存空间。幸运的是,既支持重定位也支持保护需求的机制已经存在。

通常,用户进程不能访问操作系统的任何部分,不论是程序还是数据。并且,一个进程中的程序通常不能跳转到另一个进程中的指令。如果没有特别的许可,一个进程中的程序不能访问其他进程的数据区。处理器必须能够在执行时终止这样的指令。

注意,内存保护的需求必须由处理器(硬件)来满足,而不是由操作系统(软件)来满足。这是因为操作系统不能预测程序可能产生的所有内存访问;即使可以预测,提前审查每个进程中可能存在的内存违法访问也是非常费时的。因此,只能在指令访问内存时来判断这个内存访问是否违法(存取数据或跳转)。为实现这一点,处理器硬件必须具有这个能力。

任何保护机制都必须具有一定的灵活性,以允许多个进程访问内存的同一部分。例如,

如果多个进程正在执行同一个程序，则允许每个进程访问该程序的同一个副本要比让每个进程有自己单独的副本更有优势。合作完成同一个任务的进程可能需要共享访问相同的数据结构。因此内存管理系统必须允许对内存共享区域进行受控访问，而不会损害基本的保护。我们将会看到用于支持重定位的机制也支持共享。

5.2.3 逻辑组织


逻辑组织、物理组织

计算机系统中的内存总是被组织成线性的（或一维的）地址空间，并且地址空间是由一系列字节或字组成的。外部存储器（简称外存）在物理层上也是按类似方式组织的，尽管这种组织方式类似于实际的机器硬件，但它并不符合程序构造的典型方法。大多数程序被组织成模块，某些模块是不可修改的（只读、只执行），某些模块包含可以修改的数据。如果操作系统和计算机硬件能够有效地处理以某种模块的形式组织的用户程序和数据，则会带来很多好处。

（1）可以独立地编写和编译模块，系统在运行时解析从一个模块到其他模块的所有引用。

（2）通过适度的额外开销，可以给不同的模块以不同的保护级别（只读、只执行）。

（3）可以引入某种机制，使得模块可以被多个进程共享。在模块级提供共享的优点在于，它符合用户看待问题的方式，因此用户也可以很容易地指定需要的共享。

最易于满足这些需求的工具是分段，这也是第 6 章要探讨的一种内存管理技术。

5.2.4 物理组织

正如 5.1.1 节所论述的，计算机存储器至少要被组织成两级，称为内存和外存。内存提供快速的访问，成本也相对比较高，并且内存是易失性的，即它不能提供永久存储。外存比内存慢而且便宜，它通常是非易失性的。因此，大容量的外存可以用于长期存储程序和数据，而较小的内存则用于保存当前使用的程序和数据。

在这种两级方案中，系统主要关注的是内存和外存之间信息流的组织。可以让程序员负责组织这个信息流，但由于以下两方面的原因，这种方式是不切实际的，也是不合乎要求的。

（1）可供程序和数据使用的内存可能不足。在这种情况下，程序员必须采用覆盖（Overlaying）技术来组织程序和数据。不同的模块被分配到内存中的同一块区域，主程序负责在需要时换入或换出模块。即使有编译工具的帮助，覆盖技术的实现仍然非常浪费程序员的时间。

（2）在多道程序设计环境中，程序员在编写代码时并不能知道可用空间的大小及位置。

显然，在两级存储器间移动信息的任务应该是一种系统责任，而该任务恰恰就是存储管理的本质所在。

5.3 单道编程中的内存管理

最简单的内存管理是单道程序下的内存管理。在单道编程环境下，整个内存里面只有两个程序：一个是用户程序，另一个是操作系统。由于只有一个用户程序，而操作系统所占用的内存空间是恒定的，可以将用户程序总是加载到同一个内存地址上，即用户程序永远从

同一个地方开始执行。在这种管理方式下,操作系统永远跳转到同一个地方来启动用户程序。这样,用户程序里面的地址都可以事先计算出来,即在程序运行前就计算出所有的物理地址。这种在运行前即将物理地址计算好的方式叫做静态地址翻译。

这种内存管理方式是如何达到内存管理的两个目的的呢?首先看地址独立。固定地址的内存管理达到地址独立了吗?那就看看用户在编写程序时是否需要知道该程序将要运行的物理内存地址。显然不需要,因而用户在编程时用的虚地址无须考虑具体的物理内存,即该管理模式达到了地址独立。那么它是如何达到目的的呢?办法就是将用户程序加载到同一个物理地址上。通过静态编译即可完成虚地址到物理地址的映射,而这个静态翻译工作可以由编译器或者加载器来实现。

那么内存管理的另一抽象,即地址保护,达到了吗?那要看该进程是否会访问到别的用户进程空间,或者别的用户进程是否会访问该进程地址。答案是不会,因为整个系统里面只有一个用户程序。因此,固定地址的内存管理因为只运行一个用户程序而达到地址保护。

固定地址的内存管理单元非常简单,实际上并不需要任何内存管理单元。因为程序发出的地址已经是物理地址,在执行过程中无须进行任何地址翻译。而这种情况的直接结果就是程序运行速度快,因为越过了地址翻译这个步骤。

当然,固定地址的内存管理其缺点也很明显:①整个程序要加载到内存空间中去。这样将导致比物理内存大的程序无法运行。②只运行一个程序造成资源浪费。如果一个程序很小,虽然所用内存空间小,但剩下的内存空间也无法使用。③可能无法在不同的操作系统下运行,因为不同操作系统占用的内存空间大小可能不一样,使得用户程序的起始地址可能不一样。这样在一个系统环境下编译出来的程序很可能无法在另一个系统环境下执行。

5.4 多道编程中的内存管理

单道编程的缺点前面已阐述过,为了克服单道编程的缺点,发明了多道编程。随着多道编程度数的增加,CPU 和内存的利用效率也随着增加。当然,这种增加有个限度,超过这个限度,则因为多道程序之间的资源竞争反而造成系统效率下降。

虽然多道编程可以极大地改善 CPU 和内存的效率,改善用户响应时间,但是天下没有免费的午餐,这种效率和响应时间的改善是需要付出代价的。

这个代价是什么呢?当然是操作系统的复杂性。因为多道编程的情况下,无法将程序总是加到固定的内存地址上,也就是无法使用静态地址翻译。这样就必须在程序加载完毕后才能计算物理地址,也就是在程序运行时进行地址翻译,这种翻译称为动态地址翻译。图 5-7 描述的就是动态地址翻译的示意图。

图 5-7 动态地址翻译

也许有读者会想,可以在内存固定几个地址出来,比如说 4 个。这样可以同时加载 4 个程序到内存,而这 4 个程序分别加到这 4 个固定的地址,不就可以进行静态地址翻译了吗?但是谁能提前知道某个特定的程序将加载到 4 个固定地址的哪一个呢?而且,这样做还将

带来地址保护上的困难。既然所有的程序皆发出物理地址，该地址是否属于该程序可以访问的空间将无法确认。这样，程序之间的互相保护就成了问题。

那么多道编程的内存管理是如何进行动态地址翻译的呢？那得看内存管理的策略。多道编程下的内存管理策略有两种：固定分区和非固定分区。

5.4.1 固定分区的多道编程内存管理


固定分区的多道编程内存管理

顾名思义，固定分区的管理就是将内存分为固定的几个区域，每个区域的大小固定。最下面的分区为操作系统占用，其他分区由用户程序使用。这些分区大小可以一样，也可以不一样。考虑到程序大小不一的实际情况，分区的大小通常也各不相同。当需要加载程序时，选择一个当前闲置且容量够大的分区进行加载，如图 5-8 所示。

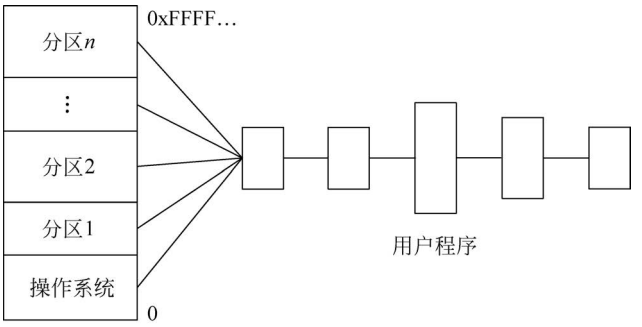

图 5-8　共享队列的固定分区

在这种模式下，当一个新的程序想要运行，必须排在一个共同的队列里等待。当有空闲分区时，才能进行加载。由于程序大小和分区大小不一定匹配，有可能形成一个小程序占用一个大分区的情况，从而造成内存里虽然有小分区闲置，但无法加载大程序的情况。如果在前面加载小程序时考虑到这一点，可以将小程序加载到小分区里，就不会出现这种情况（或者说至少降低这种情况发生的概率）。这样，我们就会想到也许可以采用多个队列，即给每个分区一个队列。程序按大小排在相应的队列里，如图 5-9 所示。

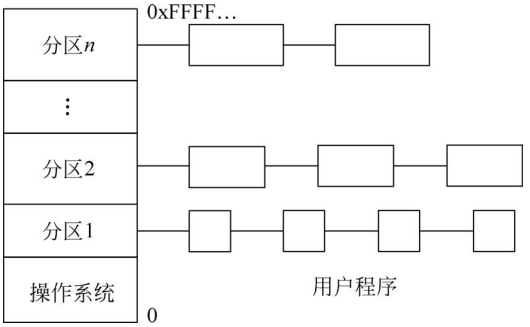

图 5-9　分开队列的固定分区

这样不同的程序有不同的队列，就像在社会中不同的社会阶层有不同的待遇一样。当然，这种方式也有缺点，就是如果还有空闲分区，但等待的程序不在该分区的等待队列上，就将造成有空间而不能运行程序的尴尬处境。

5.4.2 地址翻译的方法

在多道编程环境下,由于程序加载到内存的地址不是固定的(有多个地方可加载),必须对地址进行翻译。那么如何来翻译呢?

地址翻译的方法、动态翻译的优点

我们看到一个程序是加载到内存里事先划分好的某片区域,而且该程序是整个加载进去。该程序里面的虚地址只要加上其所占区域的起始地址即可获得物理地址。因此,翻译过程非常简单:

物理地址 = 虚拟地址 + 程序所在区域的起始地址

程序所在区域的起始地址称为(程序)基址。

另外,由于有多个程序在内存空间中,需要进行地址保护。由于每个程序占用连续的一片内存空间,因此只要其访问的地址不超出该片连续空间,则为合法访问。因此,地址保护也变得非常简单,只要访问的地址满足下面的条件即为合法访问。

程序所在区域的起始地址≤有效地址≤程序所在区域的起始地址+程序长度

由此可见,只需要设置两个端值:基址和极限,即可达到地址翻译和地址保护的目的。这两个端值可以由两个寄存器来存放,分别称为基址寄存器和极限寄存器。在固定分区下,基址就是固定内存分区中各个区域的起始内存地址,而极限则是所加载程序的长度(记住,不是内存各个分区的上限)。

这样,每次程序发出的地址需要跟极限比较大小;如果大于极限,则属非法访问。在这个时候将陷入内核,终止进程(在个别操作系统上,也可能进入一个异常处理的过程);否则将基址加上偏移获得物理地址,就可以合法地访问这个物理地址。

```
if (虚拟地址>极限) {
    陷入内核
    终止进程(core dump)
} else {
    物理地址 = 虚拟地址 + 基址
}
```

由此可见,实现基址极限管理的硬件也很简单:一个加法器和一个比较器即可。这样,对每个程序来说,它仿佛独占了一个内存空间从0到极限的计算机,如图5-10所示。

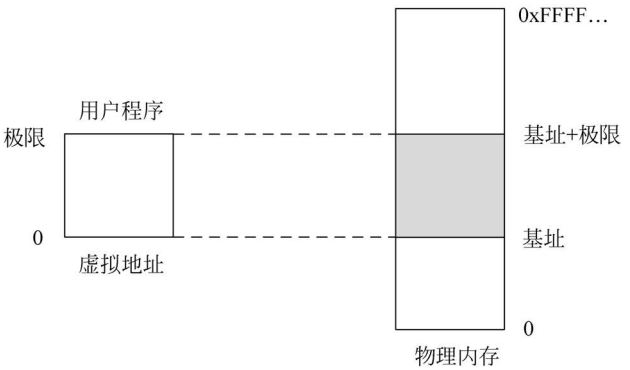

图5-10 基址极限的概念

那么怎么知道一个程序有多大呢?编译过后,就可以得到这个程序的大小。基址和极

限是很重要的两个参数，只有内核能够改变它们。如果要切换程序，只需将保存基址和极限寄存器的值按照新程序的情况重新设置即可。

5.4.3 动态地址翻译的优点

动态地址翻译虽然增加了系统消耗，不如静态地址翻译效率高，但其带来的优点远远超过静态地址翻译。第一个优点是灵活。因为实施了动态地址翻译，就无须依赖编译器或加载器来进行静态地址翻译，可以将程序随便加载到任何地方，极大地提高操作系统操作的灵活性。第二个优点是，它是实施地址保护的“不二法门”。要想进行地址保护，就必须对每个访问地址进行检查，而动态地址翻译恰恰就能做到这一点。第三个优点则更为重要，它使虚拟内存的概念得以实现。虚拟内存就是子虚乌有的内存，这个内存的空间可以非常大，比物理空间大很多。那么虚拟内存的根本是将内存扩展到磁盘上，就是将磁盘也当成内存的一部分。从这里可以获得一个重要的推论，就是一个程序可以一半放在磁盘上，一半放在内存上。这样，从物理上讲，一个程序发出的访问地址有可能在内存，也有可能在磁盘。

计算机怎么能知道这个地址所指向的数据在内存还是在磁盘上呢？这无法在静态地址编译时就知道。唯一的方法就是动态地址翻译。在每次内存访问的时候，虚拟地址就是用户每次看到的地址，这个地址只是一个抽象，它需要由内存管理单元进行翻译，变成物理内存地址才能使用。由于这个翻译是在程序执行过程中发生，因此称为动态地址翻译。有了动态地址翻译，编译器和用户进程就再也不用考虑物理地址了。

在动态地址翻译环境下，一个虚拟地址仅在被访问的时候才需要放在内存里，在其他时候并不需要占用内存。由于动态地址翻译可以动态地改变翻译参数或过程，因此可以在程序加载到不同的物理位置时，或不同的虚拟地址占用同一物理地址时，做出正确翻译。在使用基址和极限管理模式下，不同程序进入物理内存时，只需要变更基址和极限寄存器的内容即可。

5.4.4 非固定分区的内存管理


非固定分区的内存管理

前面说过固定分区的内存管理优点就是，直截了当。最大的分区就是能容纳你这个程序的分区。其缺点：一是程序大小和分区大小的匹配不容易令人满意；二是很僵硬，如果有个程序比最大的分区大怎么办呢？三是地址空间无法增长，如果程序在运行时内存需求增长怎么办？很容易想到固定分区为什么有这个缺陷，因为分区是固定大小。这样，我们自然想到用非固定分区的方式来管理多道编程的内存空间。

非固定分区的思想很简单：除了划分给操作系统的空间外，其余的内存空间是作为一个整体存在的。当一个程序需要占用内存空间时，就在该片空间里面分出一个大小刚刚满足程序所需的空间；再来一个程序，则在剩下的空间里面再这样分出一块来。在这种模式下，一个程序可以加载到任何地方，也可以和物理内存一样大。例如，一开始内存里只有操作系统。这时候进程 A 来了，从最底下分出一片与进程 A 大小一样的内存空间；然后进程 B 来了，在进程 A 上面的大片空间分出一片与进程 B 大小一样的内存空间；然后进程 C 来了，就在进程 B 上面分出一片与 C 大小一样的内存空间。这样，进程 A、进程 B、进程 C 的起始地址都不是固定的，如图 5-11 所示。

这种非固定分区内存管理的机制也很简单，就是在 5.6.3 节中讲过的基址和极限。对

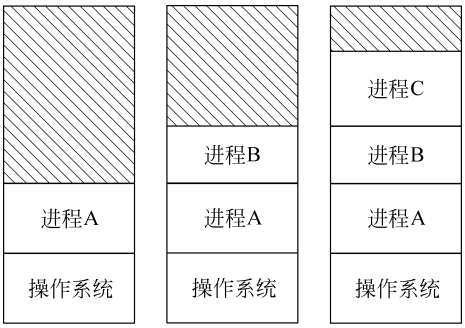

图 5-11　非固定分区的多道编程的内存管理

每一个程序配备两个寄存器：基址寄存器和极限寄存器。所有访问地址都必须在这两个寄存器值框定的空间里，否则就算非法访问。

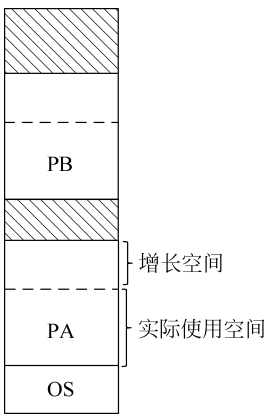

图 5-12　程序执行

仔细的读者可能已经看出，非固定分区这种管理方式存在一个重大问题：每个程序像叠罗汉一样累积，如果程序 B 在执行过程中需要更多空间，怎么办？例如，一个程序在执行过程中不断产生新的数据从而造成数据所需空间的增长，而不断进行嵌套函数调用的时候又会造成所需栈空间的增长。解决的办法当然是在一开始给程序分配空间时就分配足够大的空间，留有一片闲置空间供程序增长用，如图 5-12 所示。

图 5-12 中 A、B 两个程序分到的空间都大于其实际使用的空间。因此，程序的扩展得到支持。只要程序增长不超过所分配的空间，程序的增长将不受羁绊。

不过，在分配增长空间后需要考虑一个问题。一个程序的空间增长通常有两个来源：数据和栈。如何处理这两种空间增长的关系会对整个程序的扩展性产生影响。

最简单的方式是数据和栈往一个方向增长。这种模式下，事先给数据部分和栈部分分别留下增长空间。这样的优点是两者独立性高，缺点是空间利用率可能较低。例如，如果数据在下，栈在上，当栈长到程序所分配空间的顶时，就无法再长了。即使下面的数据部分还有闲置空间也不能利用。反之，如果数据部分长到了栈的底后也无法再增长，即使栈的上方还有空间。这种情况如图 5-13 所示。

当然，可以通过移动栈底来解决上述问题。但这样成本就高了，而且十分复杂，容易出错。一个更简单的办法是让数据和栈往相反方向增长。这样，只要本程序的自由空间还有多余，不仅可以进行函数调用，又可以增加新的数据，可以最大限度地利用这片自由空间。这是 UNIX 采取的策略，如图 5-14 所示。Windows 内存管理则为栈限制了边界，超出该边界将造成程序错误并导致程序终止。

不过，这里还有个问题：操作系统怎么知道应该分配多少空间给一个程序呢？怎么知道该程序会进行多少层嵌套调用，产生多少新的数据呢？如果为保险起见分配一个很大的空间，就有可能造成浪费；而分配小了，则可能造成程序无法继续运行。

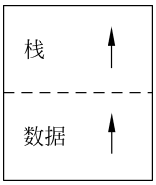

图 5-13　栈和数据同向增长

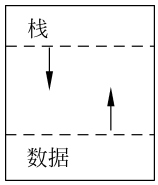

图 5-14　栈和数据逆向增长

那还有没有别的办法呢？有，给该程序换一个空间。就是当一个程序所占空间不够时，将其倒到磁盘上，再加载到一片更大的内存空间里。这种将程序倒到磁盘上，再加载进内存的管理方式称为交换(Swapping)。

5.4.5　交换

交换就是将一个进程从内存倒到磁盘上，再将其从磁盘上加载到内存中的过程。这种交换的主要目的是使程序找到一片更大的空间，从而防止一个程序因空间不够而崩溃。交换的另一目的，是实现进程切换，也就是将一个程序暂停一会儿，让另一个程序运行。不过使用交换进行进程切换的成本颇高，一般不这样做。

例如，在图 5-15(a)的状态下，进程 A 因为空间大小而无法继续执行，便将进程 A 交换到磁盘上，等待大片空间的出现(见图 5-15(b))。这时进程 D 因为占用空间小，可以执行，因此把进程 D 加载进来(见图 5-15(c))。进程 B 执行结束后空出一片空间，和本来的剩余空间合并后形成了一个可以容纳进程 A 的空间(见图 5-15(d))，这时将进程 A 再加载进来，形成图 5-15(e)的状态。

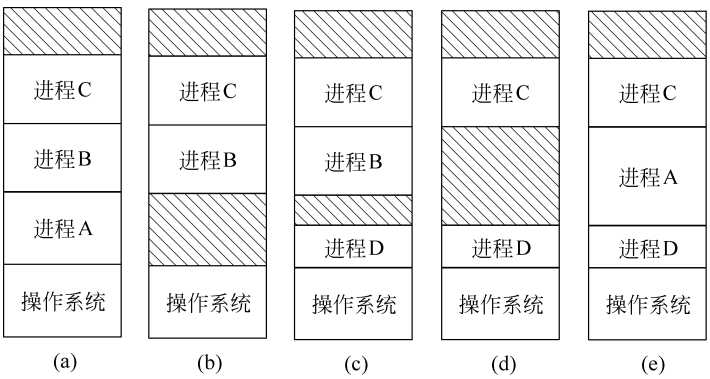

图 5-15　进程 A 因空间增长而交换到更大的空间里

交换和非固定分区一样，每个程序占用一片连续的空间，操作系统使用基址和极限来进行管理。但由于一个程序在执行中可能发生交换，其基址和极限均有可能发生变化。但这种变化对于内存管理来说，并不增加多少难度。只要每次加载程序的时候将基址和极限寄存器的内容进行重载即可。

5.4.6　重叠

前面说过，如果一个程序在执行过程中占用空间增大了，可以通过交换给它找一个更大的空间来执行。这种情况下该程序的增长无法超过物理内存空间的容量。如果一个程序超

过了物理内存,还能运行吗?

答案也许出乎意料。能。这个办法就是所谓重叠(Overlay)。重叠就是将程序按照功能分成一段一段功能相对完整的单元。一个单元执行完后,再执行下一个单元,而一旦执行到下一个单元,就不会再执行前面的单元。所以可以把后面的程序单元覆盖到当前程序单元上。这样就可以执行一个比物理内存还要大的程序。

但是这相当于把内存管理的部分功能交给了用户,是个很拙劣的方法。况且也不是每个人都能够将程序分成边界清晰的一个个执行单元的。而且,从根本上说这不能算是操作系统提供的解决方案。

5.4.7 双基址

如果运行两个一样的程序,只是数据不同,我们自然想到能否让两个程序共享部分内存空间。例如,如果同时启动两个 PPT 演示文稿,希望 PPT 的程序代码部分能共享。但在基址极限这种管理模式下,这种共享无法实现,如图 5-16 所示。

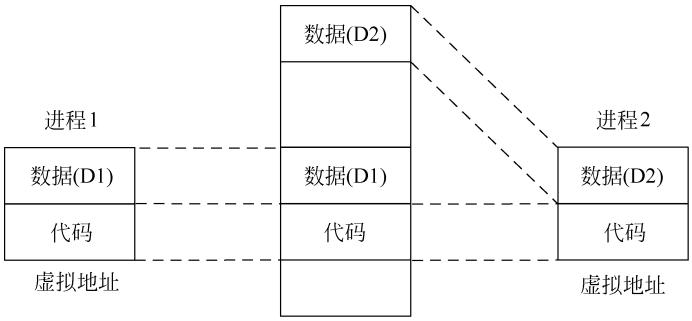

图 5-16 使用基址极限难以实现程序共享

那么有什么办法共享这段代码又不容易出错呢?答案就是设定两组基址和极限。数据和代码分别用一组基址和极限表示,这就解决了问题。

5.5 空闲空间管理

在管理内存的时候,操作系统需要知道内存空间有多少空闲。如何才能知道有哪些空闲呢?这就必须跟踪内存的使用。跟踪的办法有两种:第一种办法是给每个分配单元赋予一个字位,用来记录该分配单元是否闲置。例如,字位取值 0 表示分配单元闲置,字位取值 1 表示该分配单元已被占用。这种表示法就是所谓的位图表示法,如图 5-17 所示。

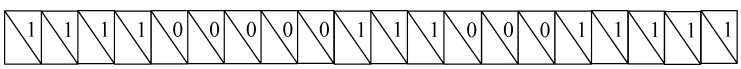

图 5-17 内存分配位图

从图 5-17 中可以看出,内存空间最前面的 4 个分配单元已经被占用,接下来是 5 个分配单元则处于闲置状态,可以供程序使用。其他以此类推。

位图表示法的优点是直观、简单。在搜索需要的闲置空间时只需要找到一片 0 个数大于等于所需分配单元数即可。例如,如果需要找一片 4 个分配单元大的闲置空间,则通过扫

描位图表，找到 4 个 0 即可。将这片空间分配给需要的程序，并将相应的位图值设置为 1。

另外一种办法是将分配单元按是否闲置链接起来，这种办法称为链表表示法。对于图 5-17 的位图所表示的内存分配状态，如果用链表表示则如图 5-18 所示。

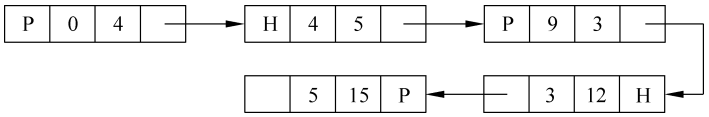

图 5-18　内存分配的链表表示

图 5-18 中的 P 代表程序，即当前这片空间由程序占用。后面的数字是本片空间的起始分配单元号和大小。H 代表的是空洞，即这是一片闲置空间。例如，图中的第 1 个链表项表示一片大小为 4 个分配单元的程序，起始地址为第 0 个分配块。第 2 个链表项表示一片大小为 5 个分配单元的闲置空间块，起始地址为第 4 个分配单元。其他以此类推。

在链表表示下，寻找一个给定大小的闲置空间意味着找到一个类型为 H 的链表项，其大小大于或等于给定的目标值。不过，扫描链表速度通常较慢。为提高查找闲置单元的速度，有人提出了将闲置空间和被占空间分开设置链表，这样就形成了两个链表的管理模式。

位图表示和链表表示各有优缺点。如果程序数量很少，那么链表比较好，因为链表的表项数量少。例如，如果只有 3 个程序在内存中，则最多只需要 7 个链表节点。但是如果程序很稠密，那么链表的节点就很多了。

位图表示法的空间成本是固定的，它不依赖于内存中程序的数量。因此，从空间成本上分析，到底使用哪种表示法得看链表表示后的空间成本是大于位图表示还是小于位图表示而定。

从可靠性上看，位图表示法没有容错能力。如果一个分配单元为 1，你并不能肯定它应该为 1，还是因为错误变成 1 的，因为链表有被占空间和闲置空间的表项，可以相互验证，具有一定的容错能力。

从时间成本上，位图表示法在修改分配单元状态时，操作很简单，直接修改其位图值即可，而链表表示法则需要对前后空间进行检查以便做出相应的合并。例如，在图 5-18 所示的情况下，如果中间的那个程序（占用位置从 9 开始，长度为 3）终止，则链表将如图 5-19 所示。如果是最前面的程序终止，则链表将如图 5-20 所示。

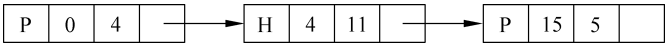

图 5-19　中间程序空间释放时链表项的合并

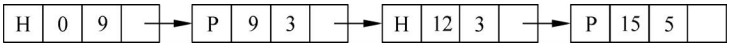

图 5-20　第一个程序空间释放时链表项的合并

当然，还可以从查找时间上进行分析。这个留给读者作为练习。

小　　结

存储器是计算机系统的重要组成部分。存储管理对主存中的用户区进行管理，其目的是尽可能地提高主存空间的利用率，使主存在成本、速度和规模之间获得较好的权衡。存储

管理的基本功能有：主存空间的分配与回收、地址转换、主存空间的共享与保护、主存空间的扩充。

在多道程序设计系统中，为了方便程序编制，用户程序中使用的地址是逻辑地址，而CPU则是按物理地址访问主存、读取指令和数据。为了保证程序的正确执行，需要进行地址转换。地址转换又称为重定位，有静态重定位和动态重定位。采用动态重定位的系统支持程序的浮动。

早期单用户单任务操作系统中主存管理采用单用户连续存储管理方式。现代操作系统支持多道程序设计，满足多道程序设计最简单的存储管理技术是分区管理，有固定分区管理和可变分区管理。分区管理中，当主存空间不足时，交换技术和覆盖技术可以达到扩充主存的目的。