第5章

存储管理

本章首先介绍存储器的存储结构和存储管理的基本功能，描述程序的装入与链接方式，根据存储管理的功能讨论了单一连续存储管理、固定分区存储管理、可变分区存储管理、页式存储管理、段式存储管理和段页式存储管理。最后讨论虚拟存储管理方式。本章需要重点掌握以下要点：

- 了解存储管理的主要功能；
- 理解程序的装入与链接方式；
- 掌握连续存储管理方式和非连续分配管理方式（页式存储管理、段式存储管理、段页式存储管理）；掌握虚拟存储器的基本概念、请求分页存储管理以及常用的页面置换算法。

5.1 存储管理概述

存储器是冯·诺依曼型计算机的五大功能部件之一，用于存放程序（指令）、操作数（数据）以及操作结果。计算机系统中，存储器一般分为主存储器和辅助存储器两大类。CPU可以直接访问主存储器中的指令和数据，但不能直接访问辅助存储器。在I/O控制系统管理下，辅助存储器与主存储器之间可以进行信息传递。

主存储器简称主存，或称为内存。主存可分为系统区和用户区两个区域。当系统初始化启动时，操作系统内核将自己的代码和静态数据结构加载到主存的底端，这部分主存空间将不再释放，也不能被其他程序或数据覆盖，通常称为系统区。在系统初始化结束之后，操作系统内核开始对其余空间进行动态管理，为用户程序和内核服务例程的运行系统动态分配主存，并在执行结束时释放，这部分空间通常称为用户区。

存储管理是对主存中的用户区进行管理，其目的是尽可能地方便用户和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平衡。

5.1.1 存储器的存储结构


视频讲解

在现代计算机系统中，存储部件通常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能价格比。

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，在越往上的存储层次中，存储介

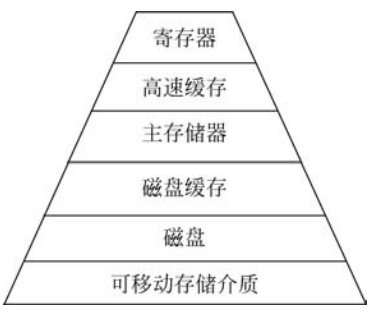


图5-1 计算机系统存储器

质的访问速度越快,价格也越高,相对存储容量也较小。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。

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十兆字节到数吉字节,而且容量还在不断增加,而嵌入式计算机系统一般仅有几十千字节到几兆字节。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于主存地址空间。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,在计算机系统中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,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,将主存中一些经常访问的信息存放在高速缓存中,可以减少访问主存的次数,大幅度提高程序执行速度。通常,运算的程序和数据存放在主存中,使用时,它被临时复制到一个速度较快的高速缓存中。当 CPU 访问一组特定信息时,首先检查它是否在高速缓存中,如果已存在,可直接从中取出使用;否则,要从主存中读出信息。通常认为这批信息被再次使用的概率很高,所以,同时还把主存中读出的信息复制到高速缓存中。在 CPU 的内部寄存器和主存之间建立了一个高速缓存,而由程序员或编译系统实现寄存器的分配或替换算法,以决定信息是保存在寄存器还是在主存中。有一些高速缓存是由硬件实现的,如大多数计算机有指令高速缓存,用来暂存下一条欲执行的指令,如果没有指令高速缓存,CPU 将会空等若干个周期,直到下一条指令从主存中取出。所以,高速缓存的管理是一个重要的设计问题,仔细确定其大小和替换策略,能够使 80%～99%的所需信息在高速缓存中找到,因而,系统性能极高。由于高速缓存的速度越高价格越贵,故在有的计算机系统中设置了两级或多级高速缓存。紧靠主存的一级高速缓存的速度最高,容量最小,二级高速缓存的容量稍大,速度也稍低。可见,高速缓存是解决主存速度与 CPU 速度不相匹配的一种部件。

由于目前磁盘的 I/O 速度远低于对主存的访问速度,因此将频繁使用的一部分磁盘数据和信息暂时存放在磁盘缓存中,可以减少访问磁盘的次数。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,提供对主存空间的扩充。主存也可以看作是辅存的高速缓存,因为辅存中的数据必须复制到主存中才能使用;反之,数据也必须先存在主存中,才能输出到辅存中。

一个文件的数据可能出现在存储器层次的不同级别中,例如,一个文件数据通常被存储在辅存中(如硬盘),当其需要运行或被访问时,就必须调入主存,也可以暂时存放在主存的磁盘高速缓存中。大容量的辅存常常使用磁盘,磁盘数据经常备份到磁带或可移动磁盘组上,以防止硬盘故障时丢失数据。有些系统自动地把旧文件数据从辅存转储到海量存储器

(如磁带)中,这样做还能降低存储价格。

5.1.2 存储管理的功能

作为一个好的计算机系统,只有一个容量大的、存储速度快的、稳定可靠的主存是不够的,更重要的是在多道程序设计系统中能合理有效地使用空间,提高存储器的利用率,方便用户的使用。具体地说,存储管理的功能如下。

1. 主存空间的分配和去配

要主存空间允许同时容纳各种软件和多个用户作业,就必须解决主存空间的分配问题。当作业装入主存时,必须按规定的方式向操作系统提出申请,由存储管理进行具体分配。由于受到多种因素的影响,不同存储管理方式所采用的主存空间分配策略是不同的。

当主存中某个作业撤离或主动回收主存资源时,存储管理则收回它所占有的全部或部分主存空间。回收存储空间的工作称为空间的去配。

2. 实现地址转换

采用多道程序设计技术后,在主存中往往同时存放多个作业的程序,而这些程序在主存中的位置是不能预知的,所以在用户程序中使用逻辑地址,但 CPU 则是按物理地址访问主存的。为了保证程序的正确执行,存储管理必须配合硬件进行地址映射工作,把一组地址空间中的逻辑地址转换成主存空间中与之对应的物理地址。这种地址转换工作亦称为重定位。

3. 主存空间的共享和保护

主存空间的共享可以提高主存空间的利用率。主存空间的共享有两方面的含义。

(1) 共享主存资源。在多道程序的系统中,若干个作业同时装入主存,各自占用了某些主存区域,共同使用同一个主存。

(2) 共享主存的某些区域。不同的作业可能有共同的程序段或数据,可以将这些共同的程序段或数据存放在一个存储区域中,各个作业执行时都可以访问它。这个主存区域又称为各个作业的共享区域。

主存中不仅有系统程序,还有若干用户作业的程序。为了防止各作业相互干扰,保护各区域内的信息不受破坏,必须实现存储保护。存储保护的工作由硬件和软件配合实现。操作系统把程序可访问的区域通知硬件,程序执行时由硬件机构检查其物理地址是否在可访问的区域内。若在此范围,则执行,否则产生地址越界中断,由操作系统的中断处理程序进行处理。一般对主存区域的保护可采取如下措施。

(1) 程序对属于自己主存区域中的信息,既可读又可写。

(2) 程序对于共享区域中的信息或获得授权可使用的其他用户信息,只可读不可修改。

(3) 程序对非共享区域或非自己的主存区域中的信息,既不可读也不可写。

4. 主存空间的扩充

由于物理主存的容量有限,难以满足用户的需要,会影响系统的性能。在计算机软、硬件的配合支持下,可把磁盘等辅助存储器作为主存的扩充部分使用,使用户编制程序时不必考虑主存的实际容量,即允许程序的逻辑地址空间大于主存的物理地址空间,使用户感到计算机系统提供了一个容量极大的主存。实际上,这个容量极大的主存空间并不是物理意义上的主存,而是操作系统的一种存储管理方式。这种方式为用户提供的是一个虚拟的存储

器。它比实际主存的容量大,起到了扩充主存空间的作用。

5.2 程序的装入与链接

5.2.1 物理地址和逻辑地址

主存的存储单元以字节(每字节为8个二进制位)为单位编址,每个存储单元都有一个地址与其相对应。假定主存的容量为 n,则该主存就有 n 个字节的存储空间,其地址编号为0,1,2,…,$n-1$。这些地址称为主存的物理地址(绝对地址);由物理地址所对应的主存空间称为物理地址空间。

在多道程序设计系统中,主存中同时存放了多个用户作业。每次调入运行时,操作系统将根据主存的使用情况为用户分配主存空间,每个用户不可能预先知道其作业存放在主存中的具体位置。因此,在用户程序中不能使用主存的物理地址。为了方便程序的编制,每个用户可以认为自己作业的程序和数据存放在一组从0地址开始的连续空间中。用户程序中所使用的地址称为逻辑地址,逻辑地址对应的存储空间称为逻辑地址空间。


视频讲解

5.2.2 程序的装入

将一个用户的源程序装入主存中并执行,通常需要经过以下几个步骤:

(1) 编译,由编译程序(Compiler)将用户源代码编译成若干个目标模块(Object Module);

(2) 链接,由链接程序(Linker)将编译后形成的一组目标模块以及它们所需要的库函数链接在一起,形成一个完整的装入模块(Load Module);

(3) 装入,由装入程序(Loader)将装入模块装入主存,如图5-2所示。

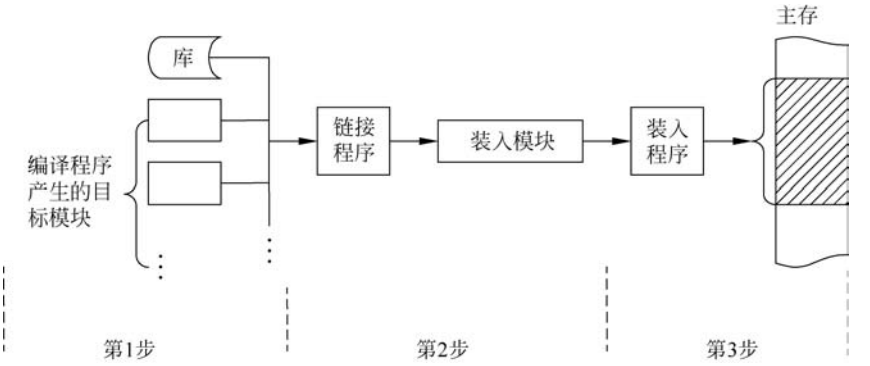


图5-2 对用户程序的处理过程

将一个装入模块装入主存,可以有绝对装入方式、静态重定位装入方式和动态重定位装入方式。

1. 绝对装入方式

如果在编译时知道程序驻留在主存中的具体位置,则编译程序将产生物理地址的目标代码。绝对装入程序按照装入模块中的地址,将程序和数据装入主存。模块装入后,由于程

序中的逻辑地址与实际主存的地址完全相同，所以不需要对程序和数据的地址进行修改。

如果由程序员直接给出程序和数据的物理地址，则不仅要求程序员熟悉主存的使用情况，一旦需要对程序或数据进行修改，或重新装入程序和数据，就可能要改变程序中的所有地址。所以，往往在程序中采用符号地址，在编译或汇编时，将其转换为物理地址。

绝对装入方式只能将目标模块装入到主存事先指定的某位置，只适用于单道程序环境。

2. 静态重定位装入方式

在装入作业时，把该作业中的指令地址和数据地址一次性全部转换成物理地址，这样在作业执行过程中无须再进行地址转换工作，这种地址转换方式称为静态重定位。这种作业装入方式称为静态重定位装入方式。静态重定位装入的过程如图 5-3 所示。

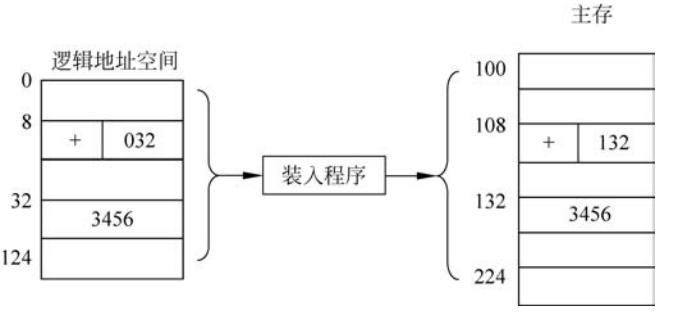


图 5-3　静态重定位装入的过程

在图 5-3 中，假定用户作业的逻辑地址空间为 0～124，其中，8 号单元处有一条加法指令，该指令要求从 32 号单元处取出操作数 3456，然后进行加法操作。如果存储管理为该作业分配的主存区域是从 100 号单元开始，那么，逻辑地址 8 号单元在主存中的对应位置是 108 号单元，32 号单元在主存中的对应位置应该是 132 号单元。如果不修改上述指令的操作数地址，则 CPU 执行该指令时，将从主存的 032 号单元中取操作数，这显然会得到错误的结果。应该把程序中所有逻辑地址（包括指令地址和数据地址）都转换成对应的物理地址。上述加法指令中的操作数地址应转换为 132，这样，在执行指令时可直接从 132 号单元中取得正确的操作数。

3. 动态重定位装入方式

在装入作业时，装入程序直接把作业装入到所分配的主存区域中。在作业执行过程中，随着每条指令或数据的访问，由硬件地址转换机制自动地将指令中的逻辑地址转换成对应的物理地址。这种地址转换的方式是在作业执行过程中动态完成的，故称为动态重定位。这种作业装入的方式称为动态重定位装入方式。如图 5-4 所示为动态重定位装入过程。

动态重定位由软件和硬件相互配合实现。硬件需要有一个地址转换机制，该机制由一个基址寄存器和一个地址转换线路组成。存储管理为作业分配存储区域后，装入程序把作业直接装入所分配的区域中，并把该主存区域的起始地址存入相应进程的 PCB 中。当进程被调度占用 CPU 时，作业所占的主存区域的起始地址也被存放到基址寄存器中。进程执行中，CPU 每执行一条指令时，都会把指令中的逻辑地址与基址寄存器中的值相加，得到相应的物理地址，然后按物理地址访问存储器。

采用动态重定位时，由于装入主存的作业仍保持逻辑地址，所以，必要时可改变作业在

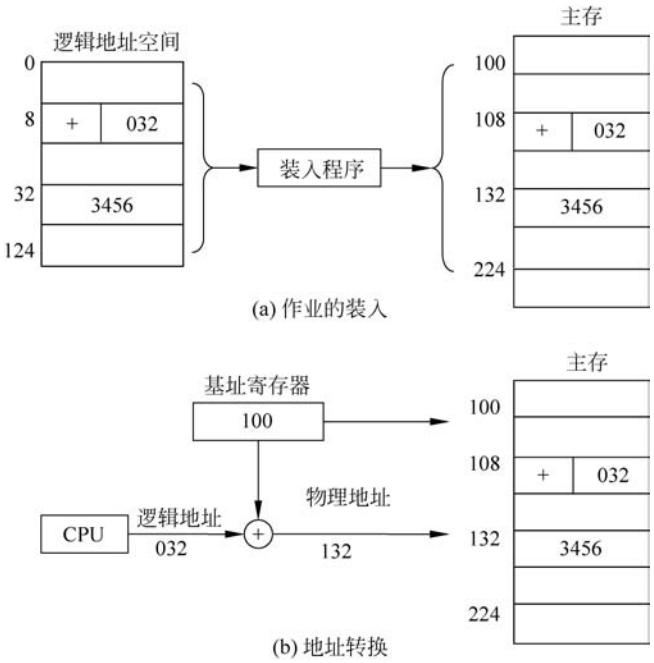


图 5-4 动态重定位装入过程

主存中的存放区域。作业在主存中被移动位置后,只需要用新区域的起始地址代替原来基址寄存器中的地址即可。这样,在执行作业时,硬件地址转换机制将按新区域的起始地址与逻辑地址相加,转换成新的物理地址,使作业仍可正确执行。

若即使改变了存储区域,作业仍能正确执行,则称程序是可浮动的。采用动态重定位的系统支持程序浮动。而采用静态重定位时,由于被装入主存中的作业信息都已转换为物理地址,作业执行过程中,不再进行地址的转换,故作业执行过程中是不能改变存放位置的,即采用静态重定位的系统不支持程序浮动。

5.2.3 程序的链接

源程序经过编译后,得到一组目标模块,再通过链接程序将这组目标模块链接,形成一个完整的装入模块。程序链接如图 5-5 所示,经过编译后得到 3 个目标模块 A、B、C,它们的长度分别为 L、M 和 N。其中,B 和 C 属于外部调用符号。根据链接时间的不同,程序的链接可分成如下 3 种方式。

1. 静态链接

在程序运行之前,首先将各个目标模块以及所需要的库函数链接成一个完整的装入模块,又称为可执行文件,运行时可直接将它装入主存。

经过编译后得到目标模块,每个模块的起始地址都为 0。模块中的地址都是相对于起始地址计算的,在链接成一个装入模块后,程序中被调用模块的起始地址不再是 0,此时必须修改被调用模块的逻辑地址,同时每个模块中使用的外部调用符号也相应地转变为逻辑地址。如图 5-5(b)所示,B 和 C 模块的起始地址分别变更为 L 和$(L+M)$,而原 B 模块中的所有逻辑地址都要加上 L,原 C 模块中的所有逻辑地址都要加上$(L+M)$。

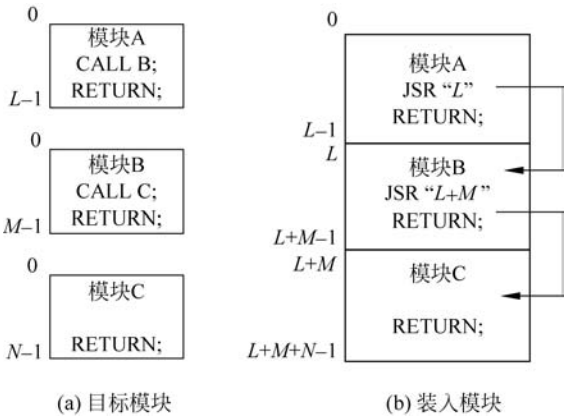


图 5-5　程序链接示意图

2. 装入时动态链接

用户源程序经编译后得到的一组目标模块，在装入主存时，采用边装入边链接的方式，即在装入一个目标模块时，若需要调用另一个模块，则找出该模块，将它装入主存，并修改目标模块中的逻辑地址。

由于采用动态链接的各个目标模块是分开存放的，操作系统可以方便地将一个目标模块链接到几个应用模块上。所以，采用动态链接方式，可以很容易地实现对目标模块的修改和更新，同时便于实现对目标模块的共享。

3. 运行时动态链接

在很多情况下，由于应用程序每次运行时的条件不同，故需要调用的模块有可能是不相同的。如果将所有目标模块装入主存，并链接在一起，就会得到一个非常大的装入模块，其中某些目标模块可能根本就没有条件运行，这样会引起程序装入在时间上和主存空间上的浪费。运行时动态链接是指在程序执行过程中，当需要该目标模块时，才把该模块装入主存，并进行链接。这样不仅可以加快程序装入的速度，而且可以节省大量的主存空间。

5.3　连续存储管理

连续存储管理是指把主存中的用户区作为一个连续区域或者分成若干个连续区域进行管理。连续存储管理方式可分为单一连续存储管理、固定分区存储管理以及可变分区存储管理方式。

5.3.1　单一连续存储管理

单一连续存储管理又称为单用户连续存储管理，是一种最简单的存储管理方式。在单一连续存储管理方式下，操作系统占用一部分主存空间，其余的主存空间作为一个连续分区全部分配给一个作业使用，即在任何时刻主存中最多只存有一个作业。这种存储管理方式适合于单用户、单任务的操作系统，在个人计算机和专用计算机系统中都有采用。如图 5-6 所示为单一连续存储管理示意图。

在单一连续存储管理方式下，CPU 中设置一个界限寄存器，用于指出主存中系统区域

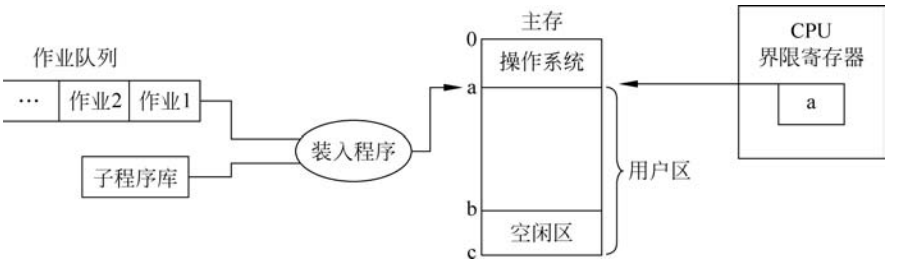


图 5-6 单一连续存储管理示意图

和用户区域的地址界限。

界限寄存器指示用户区域的起始地址。用户作业总是被装入到从该地址开始的一片连续区域中。如果作业的地址空间小于主存中的用户区,则作业占据用户区的一部分,其余部分为空闲区域。

单一连续存储管理每次只允许一个作业装入主存,因此不必考虑作业在主存中的移动问题。当作业被装入主存时,系统一般采用静态重定位方式进行地址转换。程序执行之前由装入程序完成逻辑地址到物理地址的转换工作。当然,也可以采用动态重定位方式进行地址转换。

单一连续存储管理方式下的存储保护比较简单,即判断物理地址是否大于或等于界限地址,并且物理地址是否小于或等于主存最大地址。若条件成立,则可执行;否则有地址错误,形成地址越界程序性中断事件。当采用静态重定位装入方式时,由装入程序检查其物理地址是否超过界限地址。若超过,则可以装入;否则,将产生地址错误,程序不能装入。这样,一个被装入的作业总能保证在用户区中执行,避免破坏系统区中的信息,达到存储保护的目的。

单一连续存储管理方式适用于单道程序系统。在 20 世纪 70 年代,由于小型计算机和微型计算机的主存容量有限,这种管理方式曾得到广泛应用。例如,IBM 7094 FORTRAN 监督系统、CP/M 系统、DJS 0520 系统等均采用单一连续存储管理方式,但采用这种管理方式存在几个主要缺点。

(1) CPU 利用率比较低。当正在执行的作业出现某个等待事件时,CPU 便处于空闲状态。

(2) 存储器得不到充分利用。不管用户作业的程序和数据量的多少,都是一个作业独占主存的用户区。

(3) 计算机的外围设备利用率不高。


视频讲解

5.3.2 固定分区存储管理

固定分区存储管理是预先把主存中的用户区分割成若干个连续区域,每一个连续区域称为一个分区,每个分区的大小可以相同,也可以不同。但是,一旦分割完成,主存中分区的个数就固定不变,每个分区的大小也固定不变。每个分区可以装入一个作业,不允许一个作业跨分区存储,也不允许多个作业同时存放在同一个分区中。因此,固定分区存储管理适合多道程序设计系统。如图 5-7 所示为固定分区存储管理方式的示意图。

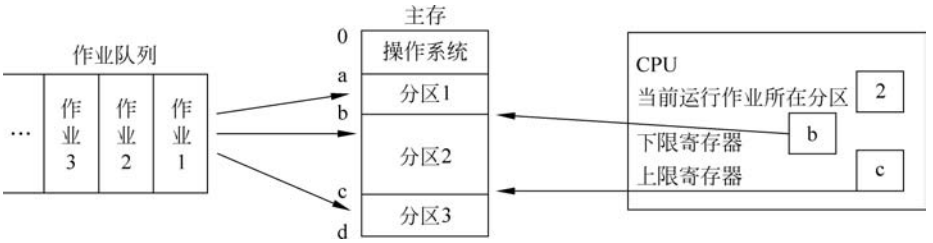


图 5-7　固定分区存储管理方式的示意图

1. 空间的分配和去配

为了管理各分区的分配和使用情况，系统需要设置一张主存分配表，以说明各分区的分配情况。一个系统中主存分配表的长度是固定的，由主存中的分区个数决定。主存分配表中记录了各个分区的起始地址和长度，并为每个分区设立一个状态标志位。当状态标志位为 0 时，表示该分区是空闲分区；当标志位为非 0 时，表示该分区已被占用。如图 5-8 所示，主存被划分成 4 个分区，每个分区的大小并不相同，其中分区 1、分区 3 和分区 4 分别被名为 JOB A、JOB B 和 JOB C 的作业所占用，分区 2 为空闲分区。

分区号	大小/KB	起址/KB	状态
1	12	20	JOB A
2	32	32	0
3	64	64	JOB B
4	128	128	JOB C

(a) 主存分配表

(b) 存储空间分配情况

图 5-8　四个分区的主存分配表示意图

当作业队列中有作业要求装入主存时，存储管理可采用顺序分配算法进行主存空间的分配。分配时，顺序查找主存分配表，选择标志位为 0 的分区，将作业地址空间大小与该分区长度进行比较，如果能容纳该作业，则将此分区分配给该作业，且在此分区的占用标志栏中填入该作业名。如果作业的地址空间大于此空闲分区的长度，则重复上述过程继续查找，查找空闲区长度大于或等于该作业的地址空间且标志位为 0 的空闲分区。若有，则分配，否则，该作业暂时不能装入主存。如图 5-9 所示为固定分区顺序分配算法的流程。

装入后的作业在执行结束后必须归还所占用的分区。存储管理根据作业名查找主存分配表，找到该作业所占用的分区，将该分区的状态标志位重新设置为 0，表示该分区现在是空闲区，可以装入新的作业。

2. 地址转换和存储保护

固定分区存储管理方式下，作业在执行过程中不会被改变存储区域，因此可以采用静态重定位装入方式装入作业。

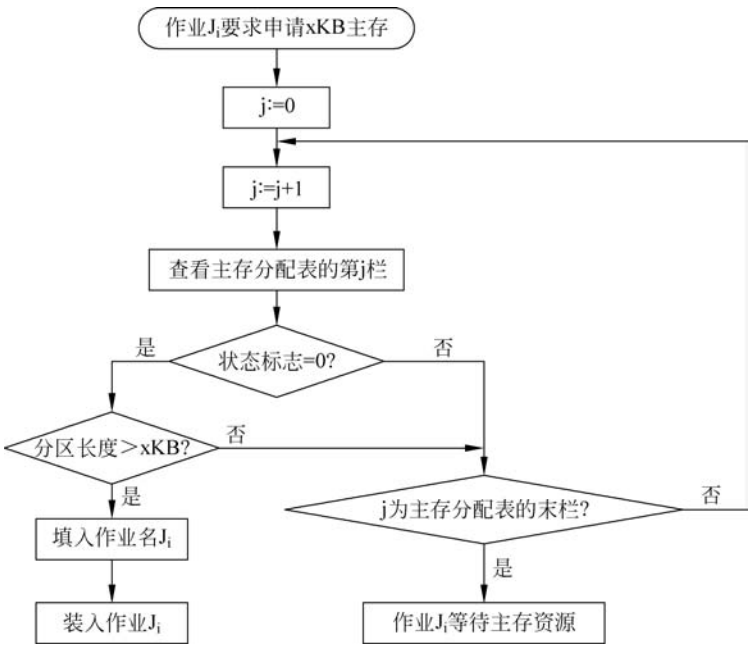


图 5-9 固定分区顺序分配算法流程

由装入程序把作业中的逻辑地址与分区的下限地址相加,得到对应的物理地址。当一个已经被装入主存的作业占有 CPU 运行时,进程调度程序将该作业所在分区的下限地址和上限地址分别存储在 CPU 的下限寄存器和上限寄存器中。CPU 执行该作业指令时必须判断:

$$\text{下限地址} \leqslant \text{物理地址} < \text{上限地址}$$

如果物理地址在上、下限地址范围内,则可按物理地址访问主存;如果条件不成立,则产生地址越界的中断事件,达到存储保护的目的。

作业运行结束时,调度程序选择另一个可运行的作业,同时修改下限寄存器和上限寄存器的内容,以保证 CPU 能控制该作业的正确执行。

3. 主存空间的利用率

固定分区存储管理方式总是为作业分配一个不小于作业地址空间的分区,因此在分区中产生了一部分空闲区域,影响了主存空间的利用率。但是固定分区存储管理方式简单,适合于程序大小和出现频繁次数已知的情形。例如,IBM 的 OS/MFT 是任务数固定的多道程序设计系统,其主存空间管理就是采用固定分区方式。

为了提高主存空间的利用率,可以采用如下几种方法。

(1) 根据经常出现的作业的大小和频率划分分区,尽可能提高各个分区的利用率。

(2) 划分分区时按分区大小顺序排列。低地址部分是较小的分区,高地址部分是较大的分区。各分区按从小到大的次序依次登记在主存分配表中。于是,在采用顺序分配算法时,从当前的空闲区中就能方便地找出一个能满足作业要求的最小空闲区分配给作业。一方面使空闲的区域尽可能少,另一方面尽可能地保留较大的空闲区,以便有大作业请求装入时容易得到满足。

(3) 按作业对主存空间的需求量排成多个作业队列,规定每个作业队列中的各作业只能依次装入对应的分区中；不同作业队列中的作业分别依次装入不同的分区中,不同的分区可同时装入作业；某作业队列为“空”时,该作业队列对应的分区也不能用来装其他作业队列中的作业。如图 5-10 所示为多个作业队列的固定分区法。

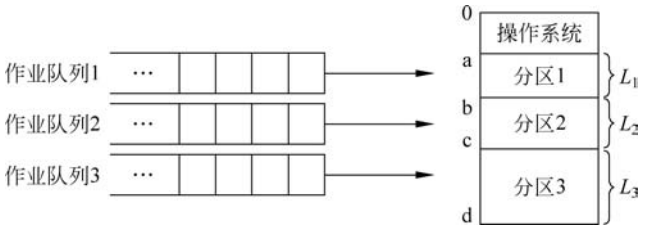


图 5-10　多个作业队列的固定分区法示意图

图中队列 1 中的作业长度小于 L_1,按规定只能装入分区 1；队列 2 中的作业长度大于 L_1,但是小于 L_2,它们按规定只能装入分区 2；队列 3 中的作业长度大于 L_2,但是小于 L_3,队列中的作业只能装入分区 3。

多个作业队列的固定分区法可以有效地防止小作业占用大分区,从而减少了闲置的主存空间量。但是如果分区划分不合适,则会造成某个作业队列经常为空队列,使对应分区经常无作业被装入,反而使分区的利用率降低。所以采用多个作业队列的固定分区法时,要结合作业的大小和出现的频率划分分区,以达到更好的空间利用率。


视频讲解

5.3.3　可变分区存储管理

可变分区存储管理并不是预先将主存中的用户区域划分成若干个固定分区,而是在作业要求装入主存时,根据作业需要的地址空间的大小和当时主存空间的实际使用情况决定是否为该作业分配一个分区。如果有足够的连续空间,则按需要分割一部分空间分区给该作业；否则令其等待主存空间。所以,在可变分区存储管理中,主存中分区的大小是可变的,可根据作业的实际需求进行分区的划分；主存中分区的个数是可变的,随着装入主存的作业数量而变化；主存中的空闲分区个数也随着作业的装入与撤离而发生变化。如图 5-11 所示为可变分区存储管理方式的存储空间分配示意图。

1. 空间的分配和去配

系统初始时,把整个主存中用户区看作一个大的空闲分区。当作业要求装入主存时,系统根据作业对主存空间的实际需求量进行分配。设作业请求的空间大小为 u. size,空闲分区的大小为 m. size。如果 m. size-u. size≤size(size 为系统事先规定的不可再分割的剩余分区的大小),则将整个空闲分区分配给该作业；否则,从空闲分区中分割出一个分区分配给该作业,其余部分仍然为空闲分区。如果 u. size > m. size,则该作业暂时不能装入,处于等待主存空间的等待状态。如图 5-11(a)所示为作业装入时主存空间分配的情况。

装入主存中的作业执行结束后,它所占用的分区被收回,成为一个新的空闲区,可以用来装入新的作业。随着作业不断地装入和作业执行完后的撤离,主存区被分成若干分区,其中有的分区被作业占用,有的分区空闲。如图 5-11(b)所示为作业被装入、执行结束后撤离时的主存空间分配情况。

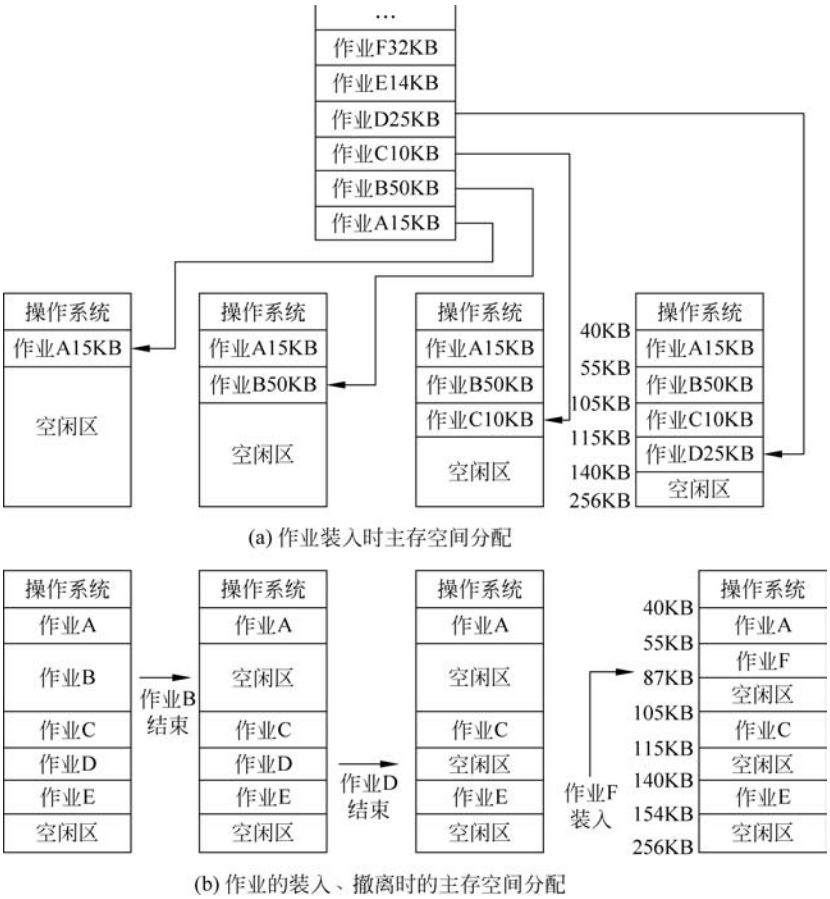


图 5-11　可变分区存储管理方式的存储空间分配示意图

1) 可变分区数据结构

可以看出,采用可变分区存储管理方式管理存储空间时,主存中已分配分区和空闲区的数目和大小都是可变的。为了实现可变分区存储管理,系统必须设置相应的数据结构,用来描述空闲分区和已分配分区的情况,为系统空间分配提供依据。常用的数据结构有以下两种形式。

(1) 分区表。系统设置空闲分区表和已分配分区表,用来描述空闲分区和已分配分区的情况,为系统空间分配提供依据。已分配分区表记录已装入的作业在主存空间中占用分区的始址和长度,用标志位指出占用该分区的作业名。空闲分区表记录主存中可供分配的空闲分区的始址和长度,用标志位指出该分区是未分配的空闲分区。由于已占用分区和空闲分区的个数在不断发生变化,因此,两张表格中都应设置适当的空栏目,分别用于登记新装入主存的作业所占用分区和作业撤离后的新空闲区。如图 5-12 的所示为可变分区存储管理方式的主存分配表,表中的内容是按图 5-11(b)中的第 4 种情况填写的。系统按一定的规则组织主存分配表,当作业要求装入时,从空闲分区表查找一个长度大于作业要求的空闲分区。

(2) 分区链。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以及用于链接前一个分区的前向指针；在分区尾部则设置一个后向指针，通过前、后向链接指针，可以将所有的空闲分区链接成一个双向链。为了方便检索，在分区尾部重复设置状态位和分区大小表目，如图 5-13 所示。

始址	长度	标志
40K	15KB	作业A
55K	32KB	作业F
105K	10KB	作业C
140K	14KB	作业E
		空
	…	

(a) 已分配分区表

始址	长度	标志
87K	18KB	未分配
115K	25KB	未分配
154K	102KB	未分配
		空
	…	

(b) 空闲分区表

图 5-12 可变分区存储管理方式的主存分配表

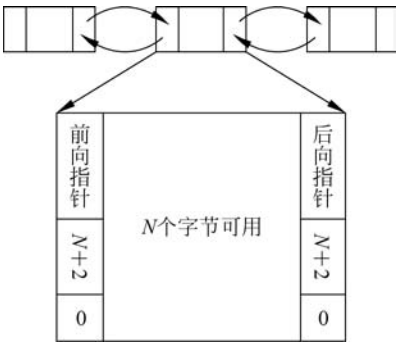


图 5-13 空闲链结构示意图

2）可变分区分配算法

将一个作业装入主存，必须按照一定的分配算法，从空闲分区表或空闲分区链中选出一个分区分配给该作业。可变分区存储管理常用的空间分配算法有：最先适应分配算法、最优适应分配算法和最坏适应分配算法。以下以空闲分区表为例，说明各种算法的原理。

（1）最先适应分配算法：在主存空间分配时，总是顺序查找空闲分区表，选择第一个满足作业地址空间要求的空闲分区进行分割，一部分分配给作业，而剩余部分仍为空闲分区。

最先适应分配算法实现简单，但是经过若干次作业的装入与撤离后，有可能把较大的主存空间分割成若干个小的、不连续的新空闲分区，这些空闲分区的长度可能比较小，不能满足主存再次分配的需要，从而使主存空间的利用率大大降低，这些空闲分区称为“碎片”。

系统在实现最先适应分配算法的过程中，往往按空闲分区的起始地址从小到大的顺序登记在空闲分区表中。这样，在分配时总是优先分配低地址部分的空闲分区，保留了高地址部分的较大空闲区，有利于大作业的装入。但是，在作业归还主存空间时，则需要按起始地址的顺序插入到空闲分区表的适当位置。

（2）最优适应分配算法：总是选择一个满足作业地址空间要求的最小空闲分区进行分配，这样每次分配后总能保留下较大的分区，使装入大作业时比较容易获得满足。

在实现过程中，空闲分区按其长度以递增顺序登记在空闲分区表中。这样，系统分配时顺序查找空闲分区表，找到的第一个满足作业空间要求的空闲分区一定是能够满足该作业要求的所有分区中的一个最小分区。

采用最优适应分配算法，每次分配后分割的剩余空间总是最小的，这样形成的“碎片”非常零散，往往难以再次分配使用，从而影响了主存空间的利用率。

（3）最坏适应分配算法：与最优适应分配算法相反，总是选择一个满足作业地址空间要求的最大空闲分区进行分割，按作业需要的空间大小分配给作业使用后，剩余部分的空间不至于太小，仍然可以供系统再次分配使用。这种分配算法对中小型作业是有利的。

实现最坏适应分配算法时，空闲分区按其长度以递减顺序登记在空闲分区表中。系统分配时顺序查找空闲分区表，表中的第一个登记项对应着当前主存的最大空闲分区。同样，

当作业归还主存空间时,则需要根据分区的大小按递减顺序登记到空闲分区表的适当位置。

以上三种算法各有优缺点。最先适应分配算法被认为是最好和最快的;而最优适应分配算法,因为它保证装入的作业大小与所选择的空闲区大小最接近,减少了碎片的大小,但是由于每次分配后剩余的碎片太小,难以满足不断到来的对于存储空间的大多数分配请求,因而性能最差。图 5-14 给出了最先适应、最优适应、最坏适应分配算法的示例。假定现在有一个作业要求分配 13KB 大小的主存空间,按如图 5-14 所示的空闲分区情况,采用最先适应分配算法时,应分割长度为 16KB 的空闲区;若采用最优适应分配算法时,则应分割长度为 14KB 的空闲区;若采用最坏适应分配算法时,则应分割长度为 30KB 的空闲区。图中斜线部分表示已占用的主存空间。

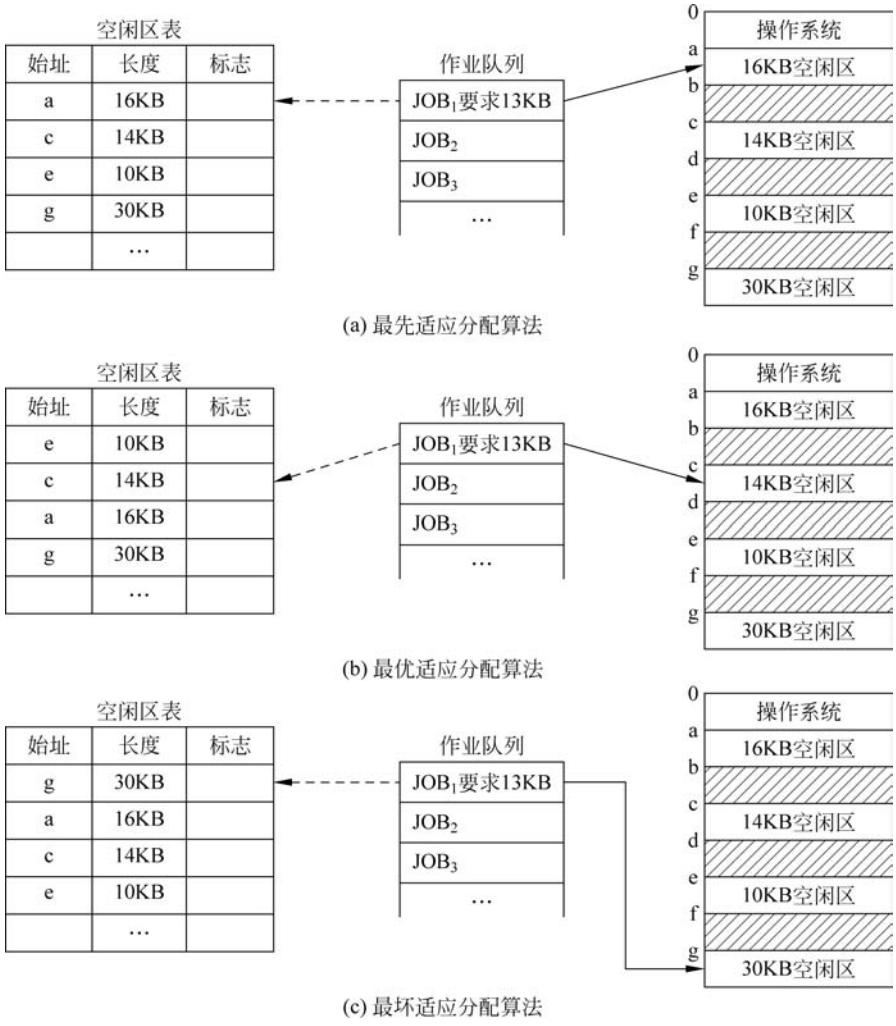


图 5-14 可变分区分配算法示意图

3) 空间的回收

装入的作业执行结束后,它所占据的分区将被回收,回收后的空闲区登记在空闲区表中,用于装入新的作业。回收空间时,应检查是否存在与回收区相邻的空闲分区,如果有,则

将其合并成为一个新的空闲分区进行登记管理。

一个回收区可能存在上邻空闲区，也可能存在下邻空闲区，或二者同时存在，或二者都不存在，如图 5-15 所示。（实际实现时，为了简化算法流程，可先对空闲分区表按始址排好序。）

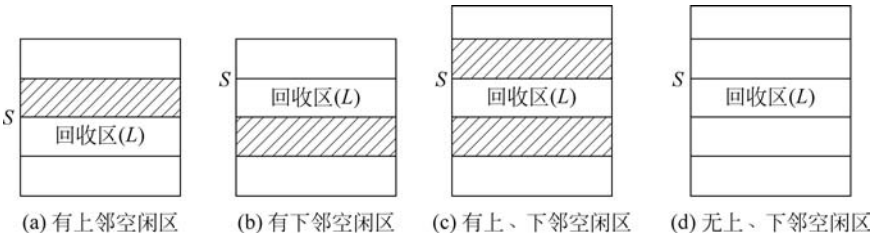


图 5-15　主存回收示意图

假定作业回收的区始址为 S，长度为 L，则有以下几种计算方法。

（1）回收区有下邻空闲区。

如果 $S+L$ 正好等于空闲区表中某个登记栏目（假定为第 j 栏）所示分区的始址，则表明归还区有一个下邻空闲区。这时应将回收区与下邻空闲分区合并，形成新的空闲分区，不必为回收区分配新的空闲分区表项，但需要修改空闲分区表中第 j 栏登记项的内容：始址修改为回收区的始址，长度为二者之和。

始址：$=S$

长度：＝原长度＋L

则此时第 j 栏指示的空闲区是回收区与下邻空闲区合并后的一个大空闲区。

（2）回收区有上邻空闲区。

如果空闲区表中第 j 个登记栏中的“始址＋长度”正好等于 S，则表明回收区有一个上邻空闲区。这时应将回收区与上邻空闲分区合并，此时，不必为回收区分配新的空闲分区表项，只要修改第 j 栏登记项的内容：始址不变，长度为上邻空闲区长度加上回收区长度 L。于是，归还区便与上邻空闲区合在一起了。

（3）回收区既有上邻空闲区又有下邻空闲区。

如果 S 正好等于第 j 个登记栏中的“始址＋长度”，并且 $S+L$ 正好等于空闲区表中某个登记栏目（假定为第 k 栏）所示分区的始址，则表明回收区既有上邻空闲区，又有下邻空闲区，此时不必为回收区分配新的空闲分区表项，应将三个分区合并为一个新的分区。可以进行如下修改：第 j 栏始址不变；第 j 栏长度为三者之和；第 k 栏的标志应修改成“空”状态。于是，第 j 栏中登记的空闲区就是合并后的空闲区，而第 k 栏成为空表目。

（4）回收区既无上邻空闲区又无下邻空闲区。

如果在检查空闲区表时上述三种情况未出现，则表明回收区既无上邻空闲区又无下邻空闲区，这时，应为回收区单独建立一个新表项，查找一个标志为“空”的登记栏，把回收区的始址和长度登记入表，且把该栏目中的标志位修改成“未分配”，表示该登记栏中指示了一个空闲区。

图 5-16 给出了合并下邻/上邻空闲区的回收算法流程。

2. 地址转换和存储保护

在可变分区存储管理方式下，一般均采用动态重定位方式装入作业。为使地址的转换

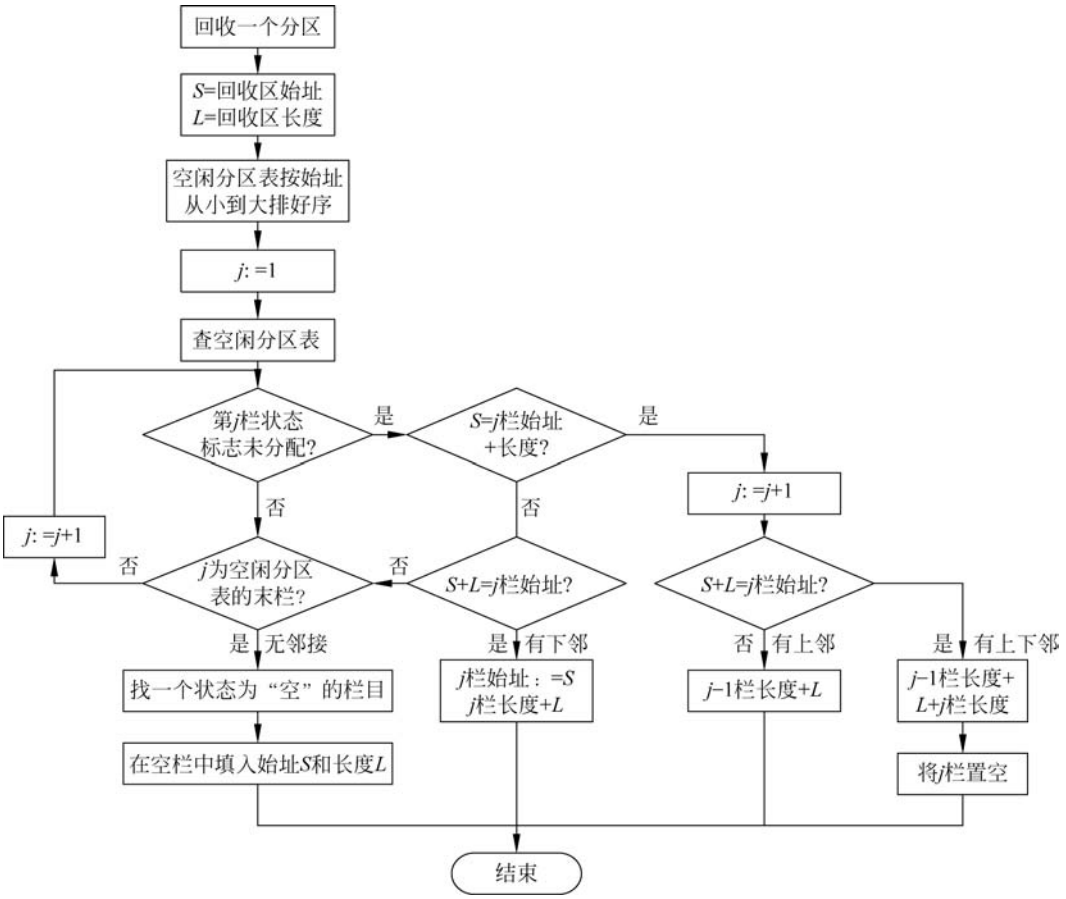


图 5-16 合并下邻/上邻空闲区的回收算法流程

不影响指令的执行速度,必须有硬件地址转换机构的支持。硬件地址转换机构包括两个专用控制寄存器:基址寄存器和限长寄存器,以及一些加法、比较线路等。基址寄存器用来存放作业所占分区的起始地址,限长寄存器用来存放作业所占分区长度。

正在运行的作业所占分区的起始地址和长度被送入基址寄存器和限长寄存器中。执行过程中,CPU 每执行一条指令都要将该指令的逻辑地址与限长寄存器中的值进行比较,当逻辑地址小于限长值时,把逻辑地址与基址寄存器的值相加,就可得到对应的物理地址。当逻辑地址大于限长寄存器中的限长值时,表示欲访问的地址超出了所分配的分区范围,这时形成一个"地址越界"的程序性中断事件,达到存储保护的目的。如图 5-17 所示为可变分区存储管理的地址转换示例。

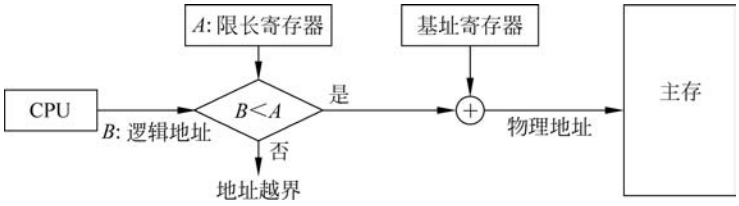


图 5-17 可变分区存储管理地址转换示意图

3. 移动技术

采用可变分区存储管理主存时，可采用移动技术使分散的空闲区集中起来以容纳新的作业，如图 5-18 所示。当主存中各个空闲区的长度都不能满足作业的要求，而主存中空闲区的总大小又大于作业需要的空间时，则可以移动已在主存中的作业，使分散的空闲区连成一片，形成一个较大的空闲区，使主存空间得到充分利用。

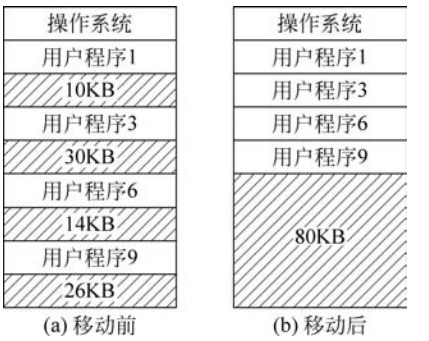


图 5-18　移动技术示意图

移动技术为作业执行过程中扩充主存空间提供方便，一道作业在执行过程中要求增加主存量时，只要适当移动邻近的作业就可增加它所占的分区长度。

移动技术可以集中分散的空闲区，提高主存空间的利用率，移动技术也为作业动态扩充主存空间提供了方便。但是，采用移动技术时必须注意以下几点。

(1) 移动会增加系统开销。移动作业时，需要进行作业信息的传送，作业移动后，作业占用的分区及空闲区的位置和长度都发生了变化，需要修改主存分配表和保存在进程控制块中的分区始址和长度，这些都增加了操作系统的工作量，也增加了操作系统占用 CPU 的时间，所以应尽量减少移动。

(2) 移动是有条件的。不是任何作业在任何时候都可以移动的。例如，外围设备与主存之间的信息交换时，通道是按确定了的主存物理地址进行传输的，如果此时移动作业，改变作业的存放地址，作业就得不到从外围设备传送来的信息，或不能将正确的信息传送到外围设备。所以，移动一道作业时，首先需要判断它是否正在与外围设备交换信息。若否，则可以移动该作业；若是，则暂时不能移动该作业，必须等待信息交换结束后才可移动。

于是，采用移动技术时，应该尽量减少移动的作业数和信息量，以降低系统的开销，提高系统的效率。一种办法是通过改变作业装入主存的方式减少移动的作业数和信息量。如图 5-19 所示为作业装入主存的两种不同方式。可见，采用两头装入作业的方式可以减少移动的作业数和信息量。

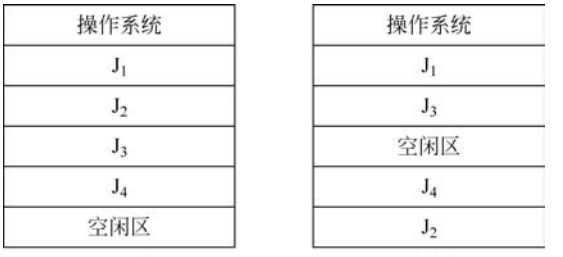


图 5-19　作业装入主存的方式

5.3.4　覆盖与交换技术

覆盖技术和交换技术是在多道环境下用来扩充主存的两种方法，这两种技术都是用来解决主存容量不足及有效利用主存的方法。覆盖技术主要用于早期的操作系统中，而交换

技术在现代操作系统中仍有较强的生命力。

单一连续存储管理和分区存储管理对作业的大小都有严格的限制,当作业要求运行时,系统将作业的全部信息一次性装入主存,并一直驻留在主存中,直至运行结束。当作业的大小大于主存可用空间时,该作业就无法运行,这就限制了计算机系统上开发较大程序的可能。覆盖和交换技术是解决大作业与小主存矛盾的两种存储管理技术,它们实质上是对主存进行了逻辑扩充。

覆盖技术的基本思想是一个程序不需要把所有的指令和数据都装入主存。可以把程序划分为若干个功能上相对独立的程序段,让那些不会同时执行的程序段共享一块主存。用户看起来好像主存扩大了,从而达到了主存扩充的目的,如图 5-20 所示。

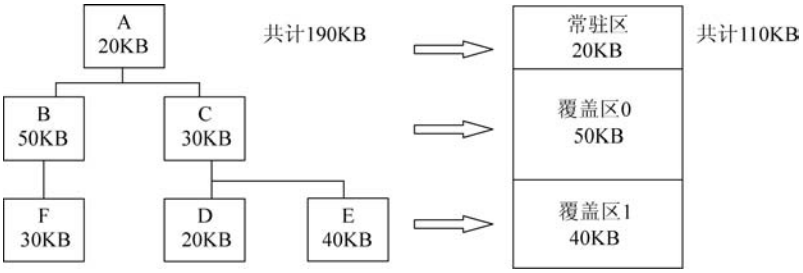


图 5-20 覆盖技术示意图

将程序的必要部分(常用功能)的代码和数据常驻主存;可选部分(不常用功能)在其他程序模块中实现,平时存放在外存中(覆盖文件),在需要用到时才装入到主存;不存在调用关系的模块不必同时装入到主存,从而可以相互覆盖(即不同时用的模块可共用一个分区)。把可以相互覆盖的程序段叫作覆盖,可共享的主存区叫作覆盖区。

为了实现覆盖管理,系统必须提供相应的覆盖管理控制程序。当作业装入运行时,由系统根据用户提供的覆盖结构进行覆盖处理。当程序中引用当前尚未装入覆盖区的覆盖中的例程时,则调用覆盖管理控制程序,请求将所需的覆盖装入覆盖区中,系统响应请求,并自动将所需覆盖装入主存覆盖区中。

覆盖技术要求程序员提供一个清楚的覆盖结构。通常,一个作业的覆盖结构要求编程人员事先给出,即程序员必须把一个程序划分成不同的程序段,并规定好它们的执行和覆盖顺序。操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖。对于一个规模较大或比较复杂的程序来说,难以分析和建立它的覆盖结构,因为这对程序员的要求较高。覆盖技术主要用于系统程序的主存管理上,如操作系统程序等设计人员清楚地了解虚空间和内部结构的程序中。例如,磁盘操作系统分为两部分:一部分是操作系统中经常用到的基本部分,它们常驻主存且占有固定区域;另一部分是不经常用的部分,它们存放在磁盘中,当调用时才被装入主存覆盖区中运行。

覆盖技术打破了必须将一个作业的全部信息装入主存后才能运行的限制,这在一定程度上解决了小主存运行大作业的矛盾。但是,采用覆盖技术编程时,必须划分程序模块和确定程序模块之间的覆盖关系,这增加了编程复杂度;从外存装入覆盖文件,是以时间延长来换取空间节省。

交换技术最早用在麻省理工学院的兼容分时系统中,其基本思想是把主存中暂时不能

运行的进程或暂时不使用的程序和数据换出到外存，以腾出足够的主存空间，把已具备运行条件的进程或进程所需要的程序和数据换入主存。

交换技术并不要求程序员做特殊的工作。整个交换过程完全由操作系统进行，对于进程是透明的。交换的对象可以是整个进程，此时成为“整体交换”或“进程交换”。图 5-21 为作业交换的示意图。

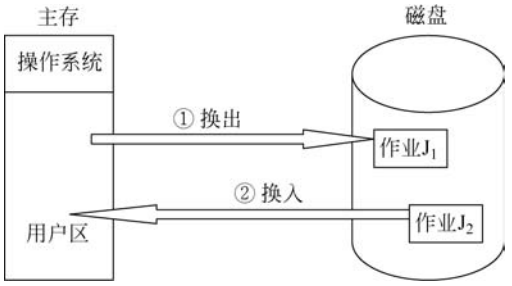


图 5-21　作业交换示意图

实现交换技术需要“后援”存储器，通常是硬盘，它必须具备两个显著的特征：数据传输快，容量足够大。交换技术可以和许多存储管理技术结合使用，如页式存储管理、段式存储管理、请求页式存储管理等。

交换技术主要是在进程或作业之间进行，而覆盖则主要是在同一个作业或进程内进行。另外，覆盖技术只能用于处理相互之间较为独立的程序段。覆盖技术主要用于早期的操作系统，交换技术广泛用于小型分时系统中，交换技术的发展导致了虚拟存储技术的出现。


视频讲解

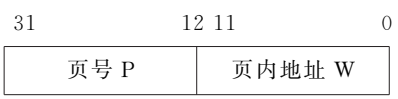
5.4　页式存储管理

连续分配方式要求作业一次性、连续装入主存空间，对空间的要求较高，而且经过若干个作业的装入与撤销后，有可能形成很多“碎片”，虽然可能通过移动技术将零散的空闲区汇集成可用的大块空间，但需要增加系统的额外开销。如果采用不连续存储的方式把逻辑地址连续的作业分散存放到几个不连续的主存区域中，并能保证作业的正确执行，就既可充分利用主存空间、减少主存的碎片，又可避免移动所带来的额外开销。页式存储管理的出现很好地解决了以上问题。

5.4.1　基本原理

页式存储管理是把主存划分成大小相等的若干区域，每个区域称为一块，并对它们加以顺序编号，如 0# 块、1# 块等。与此对应，用户程序的逻辑地址空间划分成与块大小相等的若干页，同样为它们加以顺序编号(从 0 开始)，如第 0 页、第 1 页等。页的大小与块的大小相等。

分页式存储管理的逻辑地址由两部分组成：页号和页内地址。其格式为：



地址结构确定了主存的分块大小,也就决定了页面的大小。图中的地址长度为32位,其中0～11位为页内地址;12～31位为页号。那么块的大小为2^{12}=4KB,逻辑地址有2^{20}页,每一页有4096个字节,编号为0～4095。从地址结构来看,逻辑地址是连续的,用户在编制程序时,无须考虑如何分页。

进行存储空间分配时,以块为物理单位进行主存空间分配,即把作业信息按页存放到块中。根据作业的长度可确定它的页面数,一个作业有多少页,在装入主存时就给它分配多少块主存空间,而这些主存块是可以不相邻的。这样避免了为得到连续的存储空间而进行的移动。

5.4.2 存储空间的分配与去配

分页式存储管理把主存空间划分成若干块,以块为单位进行主存空间的分配。由于块的大小是固定的,系统可以采用一张主存分配表来记录已分配的块、尚未分配的块以及当前剩余的空闲块总数。最简单的办法可用一张“位示图”来记录主存的分配情况。位示图需要占据一部分主存空间。例如,一个划分为1024块的主存,如果该主存单元字长为32位,则位示图要占据1024/32=32个字。位示图中的每一位与一个物理块对应,用0/1(空闲/已占用)表示对应块的占用标志,另用一个字记录当前系统的剩余空闲块总数,如图5-22所示。

图5-22 页式存储管理位示图示意

进行主存分配时,首先查看剩余空闲块总数是否能够满足作业要求,若不能满足,则不进行分配,作业不能装入主存;若能满足,则从位示图中找出为0的位,并且将其占用标志置为1,从空闲块总数中减去本次占用的块数,按找到的位计算出对应的块号,建立该作业的页表,并把作业装入对应的物理块中。

由于每一块的大小相等,在位示图中查找到一个为0的位后,根据它所在的字号、位号,按如下公式计算出对应的块号:

$$\text{块号}=\text{字号}\times\text{字长}+\text{位号}$$

当一个作业执行结束时,则应该收回作业所占的主存块。根据回收的块号计算出该块在位示图中对应的位置,将占用标志修改为0,同时把回收块数加入到空闲块总数中。假定回收块的块号为i,则在位示图中对应的位置为:

$$\text{字号}=\left[\frac{i}{\text{字长}}\right],\quad \text{位号}=i\,\%\,\text{字长}$$

其中，[]表示对 i 除以字长后取其整数部分；%表示对 i 除以字长后取其余数部分。

5.4.3　页表与地址转换

在分页式存储管理系统中，允许将作业的每一页离散地存储在主存的物理块中，但系统必须能够保证作业的正确运行，即能在主存中找到每个页面所对应的物理块。为此，系统为每个作业建立了一张页面映像表，简称页表。页表实现了从页号到主存块号的地址映像。作业中的所有页（0～n）依次地在页表中记录了相应页在主存中对应的物理块号，如图 5-23 所示。页表的长度由进程或作业拥有的页面数决定。

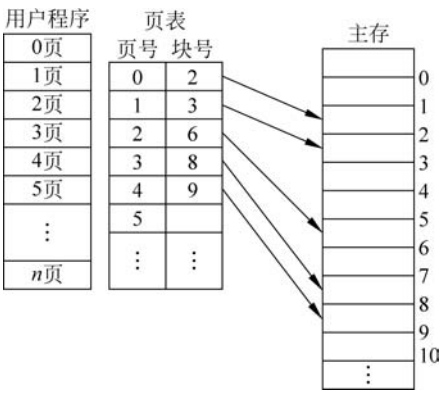


图 5-23　页表示意图

页式存储管理采用动态重定位方式装入作业，作业执行时通过硬件的地址转换机构实现从用户空间中的逻辑地址到主存空间中物理地址的转换工作。由于页内地址和物理块内的地址是一一对应的（例如，对应页面大小是 1KB 的页内地址是 0～1023，其对应的物理块内的地址也是 0～1023，无须再进行转换），因此，地址变换机构的任务实际上是将逻辑地址中的页号转换成为主存中的物理块号。页表是硬件进行地址转换的依据。

调度程序在选择作业后，将选中作业的页表始址送入硬件设置的页表控制寄存器中。地址转换时，只要从页表寄存器中就可找到相应的页表。当作业执行时，分页地址变换机构会自动将逻辑地址分为页号和页内地址两部分，以页号位索引检索页表。如果页表中无此页号，则产生一个“地址错”的程序性中断事件；如果页表中有此页号，则可得到对应的主存块号，再按逻辑地址中的页内地址计算出欲访问的主存单元的物理地址。因为块的大小相等，所以有：

$$\text{物理地址} = \text{块号} \times \text{块长} + \text{页内地址}$$

上述地址变换过程全部由硬件地址变换机构自动完成，如图 5-24 所示。

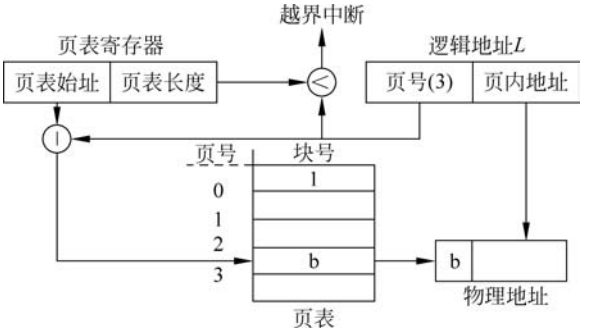


图 5-24　分页系统的硬件地址变换机构示意图

5.4.4 快表

由于页表是存放在主存中的,这样取一个数据或指令至少需要访问主存两次以上。第一次是访问主存中的页表,查找到指定页面所对应的物理块号,将块号与页内地址拼接,计算出数据或指令的物理地址。第二次访问主存时,根据第一次得到的物理地址进行数据或指令的存取操作。

为了提高存取速度,可以设想把页表存放在一组寄存器中,但寄存器成本太高,数量有限,不可行。通常在地址变换机构中增设一个具有并行查找能力的小容量高速缓冲寄存器,又称相联寄存器。利用高速缓冲寄存器存放页表的一部分,把存放在高速缓冲寄存器中的这部分页表称为快表。快表中登记了当前作业中最常用的页号与主存中块号的对应关系,图 5-25 所示为具有快表的地址变换机构示例。

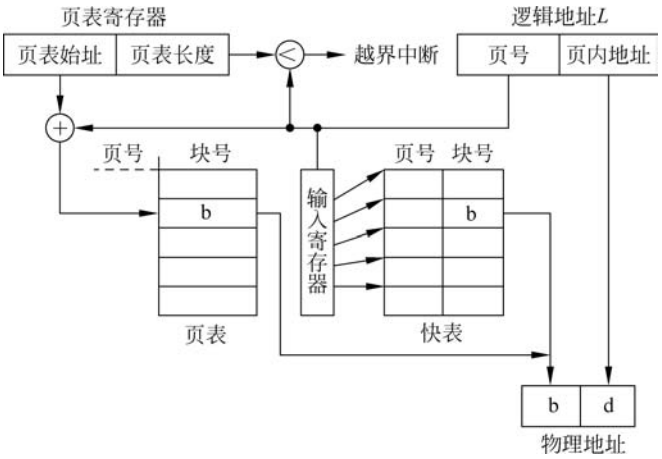


图 5-25 具有快表的地址变换机构示意图

快表的查找速度极快,但成本很高,所以一般容量非常小,通常只存放 16～512 个页表项。由于程序的执行往往具有局部性特征,如果快表中包含了最近常用的页表信息,则可实现快速查找并提高指令执行速度的目的。据统计,从快表中直接查找到所需页表项的概率可达 90%以上。

整个系统提供一个页表寄存器和一个相联寄存器,只有当前占用 CPU 的进程才能占用页表寄存器和相联寄存器。在多道程序设计系统中,当某个作业让出 CPU 时,应同时让出页表寄存器和相联寄存器。由于快表是动态变化的,所以在让出相联寄存器时,应将快表保护好,以便再次执行时使用。

5.4.5 页的共享与保护

页式存储管理能方便地实现程序和数据的共享。在多道程序系统中,编译程序、编辑程序、解释程序、公共子程序、公共数据等都是可共享的,这些共享的信息在主存中只需要保留一个副本,这大大提高了主存空间的利用率。

在实现共享时,必须区分数据的共享和程序的共享。实现数据共享时,可允许不同的作业对共享的数据页采用不同页号,只需要将各自的有关表目指向共享的数据信息块即可。

而实现程序共享时，由于页式存储结构要求逻辑地址空间是连续的，所以在程序运行前，它们的页号是确定的。假设有一个共享程序EDIT，其中含有转移指令，转移指令中的转移地址必须指明页号和页内地址，如果是转向本页，则转移地址中的页号应与本页的页号相同。假设有两个作业共享该程序EDIT，一个作业定义它的页号为3，另一个作业定义它的页号为5。既然一个EDIT程序要为两个作业以同样的方式服务，那么这个程序一定是可再入程序，转移地址中的页号不能按作业的要求随机地改成3或5，因此对共享程序必须规定一个统一的页号。当共享程序的作业数较多时，规定一个统一的页号就比较困难。如图5-26所示为两个作业共享一个程序和一个数据段的情况。

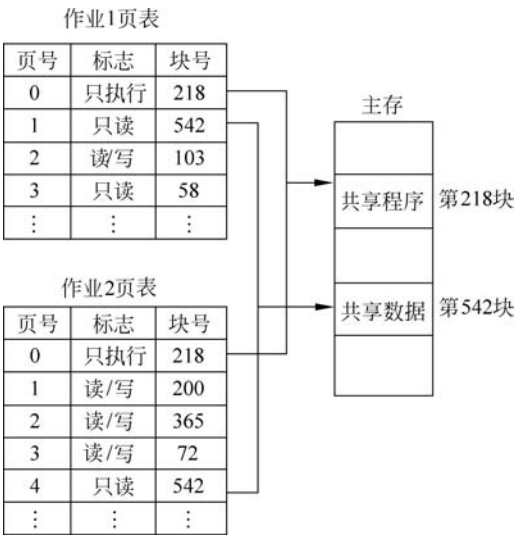


图5-26 页的共享示意图

页的共享可节省主存空间，但实现程序和数据的共享必须解决共享信息的保护问题。通常，系统可以在页表中增加一些标志位，指出该页的信息可读/写、只读、只执行、不可访问等，CPU在执行指令时进行核对，如果向只读块执行写入操作，则系统将停止该条指令的执行并产生中断。

5.5 段式存储管理

视频讲解

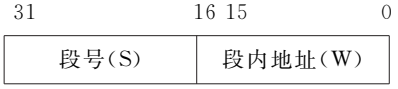
用户编制的程序是由若干段组成的：一个程序可以由一个主程序、若干子程序、符号表、栈以及数据等若干段组成。每一段都有独立、完整的逻辑意义，每一段程序都可独立编制，且每一段的长度可以不同。

段式存储管理支持用户的分段观点，具有逻辑上的清晰和完整性，它以段为单位进行存储空间的管理。

5.5.1 基本原理

每个作业由若干个相对独立的段组成，每个段都有一个段名。为了实现简单，通常可用段号代替段名，段号从0开始，每一段的逻辑地址都从0开始编址，段内地址是连续的，而段与段之间的地址是不连续的。

段式存储管理的逻辑地址由段号和段内地址两部分组成，其地址结构如下：



地址结构一旦确定，允许作业的最多段数及每段的最大长度也就限定了。在上述地址结构中，允许一个作业最多有(2^{16}=)64KB个段，每个段的最大长度为(2^{16}=)64KB。随着若干次作业的装入与撤离，主存空间被动态地划分为若干个长度不等的区域，这些区域称为

物理段,每个物理段由起始地址和长度确定。

分段方式已得到许多编译程序的支持,编译程序能自动地根据源程序的情况而产生若干个段。例如,Pascal 编译程序可以为全局变量、用于存储相应参数及返回地址的过程调用栈、每个过程或函数的代码部分、每个过程或函数的局部变量等,分别建立各自的段。装入程序将装入所有这些段,并为每个段赋予一个段号。

5.5.2 空间的分配与去配

分段式存储管理是在可变分区存储管理方式的基础上发展而来的。在分段式存储管理方式中,以段为单位进行主存分配,每一个段在主存中占有一个连续空间,但各个段之间可以离散地存放在主存不同的区域中。为了使程序能正常运行,即能从主存中正确地找出每个段所在的分区位置,系统为每个进程建立一张段映射表,简称段表。每个段在表中占有一个表项,记录该段在主存中的起始地址和长度,如图 5-27 所示。段表实现了从逻辑段到主存空间之间的映射。

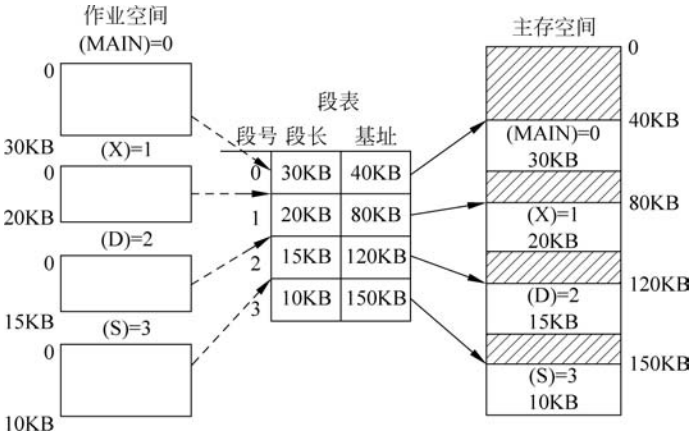


图 5-27 段的装入示意图

如果在装入某段信息时找不到满足该段地址空间大小的空闲区,则可采用移动技术合并分散的空闲区,以利于大作业的装入。

当分段式存储管理的作业执行结束后,它所占据的主存空间将被回收,回收后的主存空间登记在空闲区表中,可以用来装入新的作业。系统在回收空间时,同样需要检查是否存在与回收区相邻的空闲区。如果有,则将其合并成为一个新的空闲区进行登记管理。

段表存放在主存中,在访问一个数据或指令时,至少需要访问主存两次以上。为了提高对于段表的存取速度,通常增设一个相联寄存器,利用高速缓冲寄存器保存最近常用的段表项。

5.5.3 地址转换与存储保护

段式存储管理采用动态重定位方式装入作业。执行作业时,通过硬件的地址转换机构实现从逻辑地址到物理地址的转换工作。段表的表目起到了基址寄存器和限长寄存器的作用,是硬件进行地址转换的依据。

调度程序在选择作业后，将选中作业的段表始址和总段数对应的段表长度送入硬件设置的段表控制寄存器中。地址转换时，只要通过段表控制寄存器就可找到相应的段表。每执行一条指令时，将其段号与段表寄存器中的段表长度进行比较，若大于该作业的段表长度，则该地址无效，产生一个“地址越界”的程序性中断；否则，地址转换机构按逻辑地址中的段号查找段表，得到该段在主存中的起始地址，将逻辑地址中的段内地址与段表中的段长进行比较。若小于该段长，则起始地址加上段内地址就得到欲访问的主存物理地址；否则，该逻辑地址无效，产生一个“地址越界”的程序性中断事件。如图 5-28 所示为分段式存储管理的地址变换过程示意图。

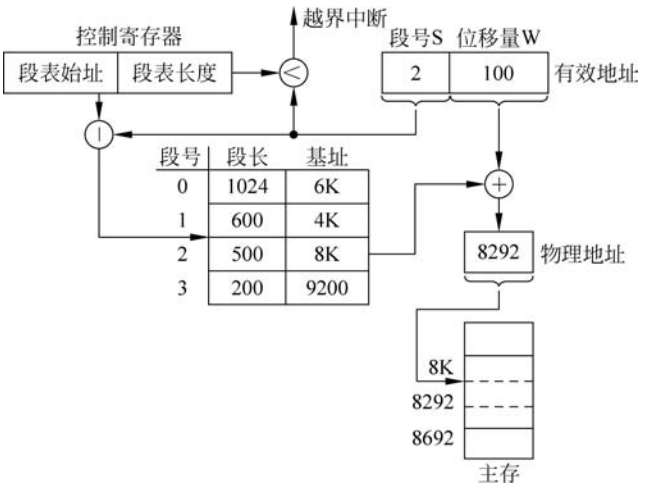


图 5-28　分段式存储管理的地址变换过程示意图

5.5.4　段的共享

由于段是按逻辑意义来划分的，可以按段名进行访问。因此分段式存储管理系统的一个突出优点是可以方便地实现段的共享，即允许若干个进程共享一个或多个段。为实现某段代码的共享，在分段式存储管理系统中，各个进程对共享段使用相同的段名，在各自的段表中填入共享段的起始地址，并置以适当的读/写控制权，即可做到共享一个逻辑上完整的主存段信息。例如，一个多用户系统可同时接纳 40 个用户，这些用户都需要执行一个文本编辑程序(Text Editor)，如果文本编辑程序有 160KB 的代码和 40KB 的数据区，则总共需要 8MB 的主存空间来支持 40 个用户的访问。如果 160KB 的代码是可重入的，则该代码只需要在主存中保留一份文本编辑程序的副本，此时所需要的主存空间仅为 1760(40×40＋160)B，只需要在每个进程的段表中为文本编辑程序设置一个段表项，如图 5-29 所示。

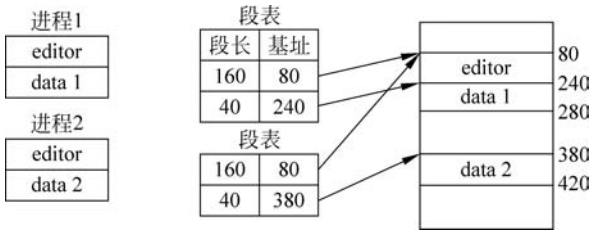


图 5-29　段的共享示意图

在多道程序设计系统中,由于进程的并发执行,一个程序段为多个进程共享时,有可能出现多次同时重复执行该段程序的情况,而该共享程序段的指令和数据在执行过程中是不能被修改的。此外,共享段也有可能被置换出主存。显然,一个正在被某个进程使用或即将被某个进程使用的共享段是不应该置换出主存的。因此,可以在段表中设置共享位来判别该段是否正在被某个进程调用。

5.5.5 分页和分段存储管理的主要区别

分页和分段系统都采用离散分配主存方式,都需要通过地址映射机构来实现地址变换,它们有许多相似之处。但两者又是完全不同的,具体表现如下。

(1) 页是信息的物理单位,是系统管理的需要而不是用户的需要;而段则是信息的逻辑单位,它含有一组意义相对完整的信息。分段是为了更好地满足用户的需要。

(2) 页的大小固定且由系统决定,因而一个系统只能有一种大小的页面;而段的长度却不固定,由用户所编写的程序决定,通常由编译程序对源程序进行编译时根据信息的性质来划分。

(3) 分页式作业的地址空间是一维的,页间的逻辑地址是连续的;而分段式作业的地址空间则是二维的,段间的逻辑地址是不连续的。


视频讲解

5.6 段页式存储管理

段式存储管理支持了用户的观点,但每段必须占据主存的连续区域,有可能需要采用移动技术汇集主存空间。为此,兼用分段和分页的方法,构成可分页的段式存储管理,通常称为段页式存储管理。段页式存储管理兼顾了段式在逻辑上的清晰和页式在管理上方便的优点。

用户对作业采用分段组织,每段独立编程,在分配主存空间时,再把每段分成若干个页面,这样每段不必占据连续的主存空间,可把它按页存放在不连续的主存块中。

段页式存储管理的逻辑地址格式如下:

段号(S)	页号(P)	页内地址(W)

段页式存储管理为每一个装入主存的作业建立一张段表,且对每一段建立一张页表。段表的长度由作业分段的个数决定,段表中的每一个表目指出本段页表的始址和长度。页表的长度则由对应段所划分的页面数所决定,页表中的每一个表目指出本段的逻辑页号与主存物理块号之间的对应关系。段页式存储管理中段表、页表与主存之间的关系如图 5-30 所示。

执行指令时,地址机构根据逻辑地址中的段号查找段表,得到该段的页表始址,然后根据逻辑地址中的页号查找该页表,得到对应的主存块号,由主存块号与逻辑地址中的页内地址形成可访问的物理地址。如果逻辑地址中的段号超出了段表中的最大段号或者页号超出了该段页表中的最大页号,都将形成“地址越界”程序性中断事件。可以看出,由逻辑地址到物理地址的变换过程中,需要访问三次主存。第一次是访问主存中的段表,获得该段对应页

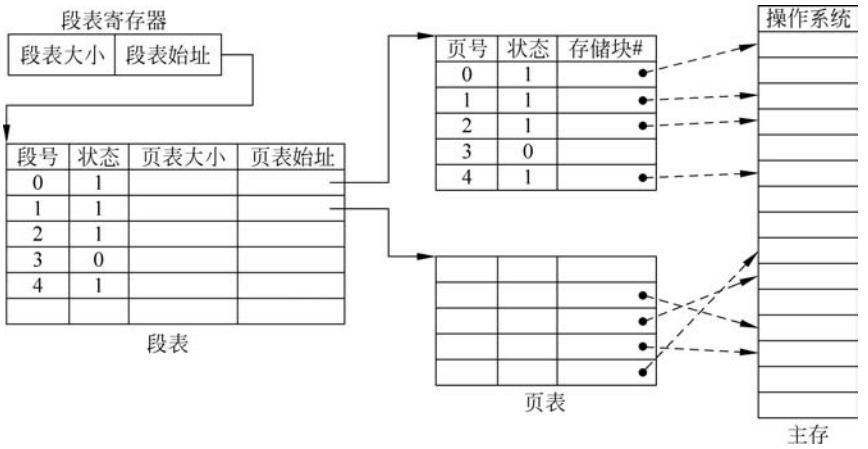


图 5-30　段页式存储管理中段表、页表与主存之间的关系

表的始址；第二次是访问页表，获得指令或数据的物理地址；第三次按物理地址存取信息。段页式存储管理的地址转换机构如图 5-31 所示。

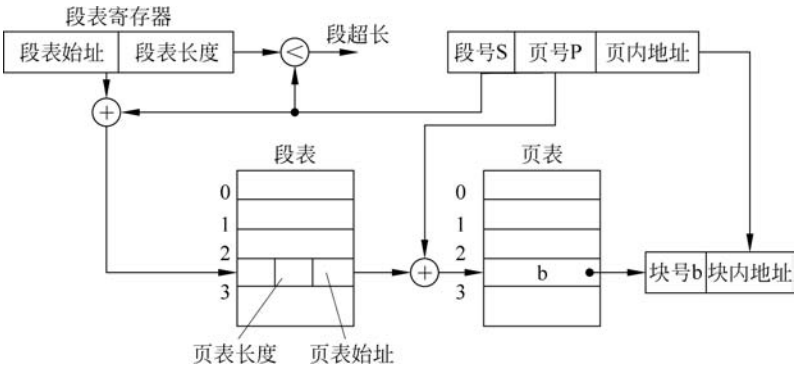


图 5-31　段页式存储管理的地址转换机构

段页式管理是段式管理和页式管理结合而成的，具有二者的优点。但由于管理软件的增加，复杂性和系统开销也随之增加；此外，需要的硬件以及占用的主存空间也有所增加。如果不采用相联寄存器方式提高 CPU 访问主存的速度，就会使得执行速度大大降低。

5.7　虚拟存储管理方式

前面所介绍的各种存储管理方式具有一个共同的特点，即作业必须一次性全部装入主存空间后才能运行，直至作业运行结束后才释放所占有的全部主存资源，这样就会出现以下情况。

（1）当主存空间不能满足作业地址空间要求时，作业就不能装入主存，无法运行。

（2）当有大量作业要求运行时，由于主存容量有限，只能将少数作业装入主存运行，而其他作业留在辅存上等待。

然而，许多在程序运行过程中不用或暂时不用的程序（数据）占据了大量的主存空间，使

得一些需要运行的作业无法装入运行。在程序运行中可以发现,程序的某些部分是相互排斥的,即在程序的一次运行中,执行了这部分程序就不会执行另一部分程序。例如,程序中的错误处理部分,仅在程序出现错误的情况下才会运行。

早在1968年,Denning就曾指出:程序执行时呈现出局部性特征,即在较短的时间内,程序的执行仅局限于某个部分;而它所访问的存储空间也局限在某个区域中。局限性表现在时间局限性和空间局限性两个方面。

(1) 时间局限性:一旦执行了程序中的某条指令,不久以后该指令可能再次执行;如果某数据被访问,则不久以后该数据可能再次被访问。例如,程序中存在大量的迭代循环、临时变量和子程序调用等。

(2) 空间局限性:一旦程序访问了某个存储单元,不久以后,其附近的存储单元也将被访问,即程序在一段时间内所访问的地址可能集中在一定范围之内。例如,对数组、表或数据堆栈进行操作。

5.7.1 虚拟存储器

基于局部性原理,可以把作业信息保存在磁盘上。当作业请求装入时,只需将当前运行所需要的一部分信息先装入主存。执行作业时,如果所要访问的信息已调入主存,则可继续执行;否则,再将这些信息调入主存,使程序继续执行;如果此时主存已满,无法再装入新的信息,则系统将主存中暂时不用的信息置换到磁盘上,腾出主存空间后,再将所需要的信息调入主存,使程序继续执行。这样,可使一个大的用户程序得以在比较小的主存空间中运行,也可以在主存中同时装入更多的作业使它们并发执行。这不仅使主存空间能充分地被利用,而且用户编制程序时可以不必考虑主存的实际容量,允许用户的逻辑地址空间大于主存的实际容量。从用户的角度来看,好像计算机系统提供了一个容量很大的主存——称为虚拟存储器。

虚拟存储器是指一种实际上并不存在的“虚假”存储器,它是系统为了满足应用对存储器容量的巨大需求而构造的一个非常大的地址空间。它使用户在编程时无须担心存储器的不足,好像有一个无限大的存储器供其使用。

虚拟存储器建立在离散分配的存储管理方式的基础上,它允许将一个作业分多次调入主存。虚拟存储器实际上是为了扩大主存容量而采用的一种设计技巧,虚拟存储器的容量由计算机的地址结构和辅助存储器的容量决定,与实际主存的容量无关;其逻辑容量由主存和辅助存储器容量之和决定,运行速度接近于主存的速度,而每位的成本却又接近于辅助存储器。

可见,虚拟存储技术是一种性能非常优越的存储器管理技术。早在20世纪60年代初期就已出现了虚拟存储器的思想,到60年代中期,较完整的虚拟存储器在两个分时系统(MULTICS和IBM系列)中得到实现,到70年代初开始推广应用,现在已广泛地应用于大、中、小型计算机和微型计算机中。

虚拟存储器具有离散性、多次性、对换性和虚拟性四大主要特征。

1. 离散性

离散性是虚拟存储器存在的基础。如果进程的主存空间必须分配在连续的物理空间中,则在将进程的某部分换出并将其他进程换入后,将很难保证下次载入时,进程的前后恰

好有足够的空闲空间。基于此限制，进程需要的部分往往无法载入主存，进程将无法继续执行下去。相反，如果将进程按照页或者段进行离散化放置，则可以将页或者段单独换出，而不必考虑载入时的位置问题。因此，分页或者分段是虚拟存储器产生的基础。

2. 多次性

多次性是指一个作业被分成多次调入主存运行，亦即在作业运行时没有必要将其全部装入，只需将当前要运行的那部分程序和数据装入主存即可；以后每当要运行到尚未调入的那部分程序时，再将其调入。多次性是虚拟存储器最重要的特征，任何其他存储管理方式都不具备这一特征。

3. 对换性

对换性是指允许在作业的运行过程中进行换进、换出，即在进程运行期间，允许将那些暂时不使用的程序和数据从主存调至外存的对换区，待以后需要时再将它们从外存调至主存；甚至还允许将暂时不运行的进程调至外存，待它们重新具备运行条件时再调入主存。换进和换出能有效地提高主存利用率。可见，虚拟存储器具有对换性的特征。

4. 虚拟性

虚拟性是指能够从逻辑上扩充主存容量，使用户所看到的主存容量远远大于实际的主存容量。这是虚拟存储器所表现出来的最重要的特征，也是实现虚拟存储器的最重要的目标。

虚拟性是以多次性和对换性为基础的，或者说，仅当系统允许将作业分多次调入主存，并能将主存中暂时不运行的程序和数据对换至硬盘中时，才有可能实现虚拟存储器；而多次性和对换性又必须建立在离散分配的基础上。


视频讲解

5.7.2 请求分页式存储管理

虚拟页式存储管理分为请求分页式管理和预调入页式管理。请求分页式管理与预调入页式管理的主要区别在于调入方式。请求分页式管理的调入方式是，当需要执行某条指令而发现它不在主存时或当执行某条指令需要访问其他数据或指令时，这些指令和数据不在主存，从而发生缺页中断，系统将外存中相应的页面调入主存。这种策略的主要优点是确保只有被访问的页面才调入主存，节省主存空间。但是处理缺页中断次数过多和调页的系统开销较大，由于每次仅调入一页，增加了磁盘的 I/O 次数。采用预调入页式管理时，操作系统依据某种算法动态预测进程最可能访问的那些页面，在使用前预先调入主存，尽量做到进程在访问页面之前已经预先调入该页，而且每次可以调入若干页面，能减少磁盘的 I/O 启动次数。但是如果调入的页面多数未被使用，则效率就很低。可见，预调入页式管理建立在预测的基础上，目前所用预调入页面的成功率在 50%左右。请求分页式管理和预调入页式管理除了在调入方式上有些区别外，其他方面基本相同。下面以请求分页式存储管理为例，介绍虚拟页式存储管理。

请求分页式存储管理是在页式存储管理的基础上，增加了请求分页功能和页面置换功能而实现的虚拟存储系统。请求分页式存储管理允许作业只装入部分页面就启动运行，在执行过程中，如果所要访问的页已调入主存，则进行地址转换，得到欲访问的主存物理地址。如果所要访问的页面不在主存中，则产生一个缺页中断，如果此时主存能容纳新页，则启动磁盘 I/O 将其调入主存；如果主存已满，则通过页面置换功能将当前所需的页面调入。

1. 请求分页式存储的管理

为了实现请求分页和页面置换功能,系统必须提供必要的硬件支持和相应的软件支持。一般需要请求分页的页表机制、缺页中断机构、地址变换机构以及页面置换算法等方面的支持。

1) 页表机制

请求分页式存储管理的主要依据就是页表。由于只是将作业的部分页面调入主存,其余部分仍存放在辅存上,因此必须指出哪些页面已在主存,哪些页面还没有装入。为此需要将页表增加若干项,修改后的页表格式如下:

页号	物理块号	状态位	访问字段	修改位	辅存地址

其中,状态位用来指出该页是否已经调入主存,如果某页对应栏的状态位为1,则表示该页已经调入主存,此时"物理块号"指出该页在主存中的占用块;如果状态位为0,则表示该页还未调入主存,此时"辅存地址"中指明了该页在磁盘上的地址,以便系统从辅存中将其调入。"访问字段"用于记录该页在一段时间内被访问的次数,或记录该页最近已有多长时间未被访问。"修改位"则表示该页在调入主存后是否被修改,由于作业在磁盘上保留一份备份,若此次调入主存后未被修改,则在置换该页时不需要再将该页写回磁盘,以减少系统的开销;如果已被修改,则必须将该页回写到磁盘上,以保证信息的更新与完整。

2) 缺页中断机构

请求分页式存储管理中,当所要访问的页面不在主存时,则由硬件发出一个缺页中断,操作系统必须处理这个中断,将所需页面调入主存,如图5-32所示为缺页中断处理流程。在处理缺页中断的过程中,同样需要保护现场、分析中断源、转入中断处理程序进行处理、恢复现场等步骤,但缺页中断又与一般的中断有着明显的区别,主要表现在以下两个方面。

(1) 在指令执行期间产生和处理中断信号。通常,CPU在一条指令结束后接收中断请求并响应,而缺页中断则是在指令执行期间所要访问的指令或数据不在主存时所产生和处理的。

(2) 一条指令在执行期间可能产生多次缺页中断。如图5-33所示,指令copy A to B跨越了两个页面,A和B分别是一个数据块并都跨页面存储,那么执行这条指令将可能产生6次缺页中断。所以系统中的硬件机构应该能够保存多次中断时的状态,以保证中断处理后能正确返回并继续执行。

3) 地址变换机构

在请求分页式存储管理中,当作业访问某页时,硬件的地址转换机构首先查找快表,若找到,并且其状态位为1,则按指定的物理块号进行地址转换,得到其对应的物理地址;若该页的状态位为0,则由硬件发出一个缺页中断,按照页表中指出的辅存地址,由操作系统将其调入主存,并在页表中填上其分配的物理块号,修改状态位、访问位,对于写指令,置修改位为1,然后按页表中的物理块号和页内地址形成物理地址。如图5-34所示为请求分页式存储管理中的地址变换过程。

如果在快表中未找到该页的页表项,则到主存中查找页表。若该页尚未调入主存,则系统产生缺页中断,请求操作系统将该页面调入,同时将此页表项写入快表。

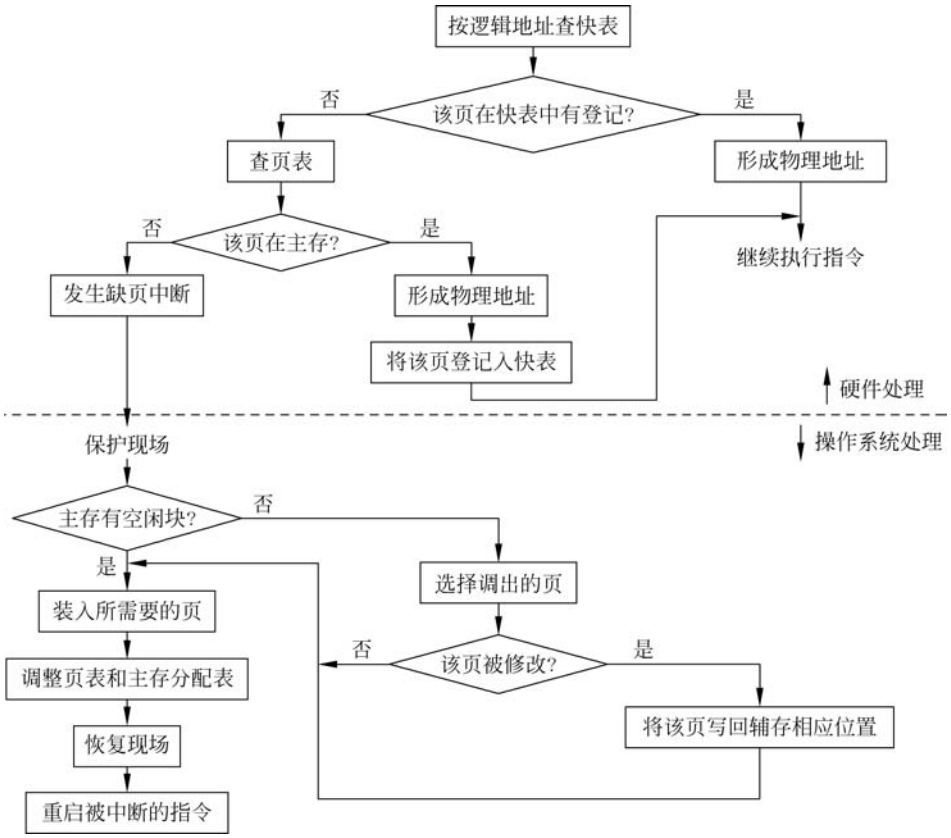


图 5-32　缺页中断处理流程

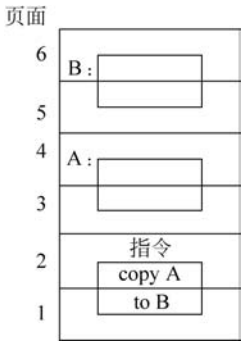


图 5-33　涉及 6 次缺页中断的指令

2. 页面置换策略

在请求分页式存储管理中，可采用两种主存分配策略，即固定分配和可变分配策略。在进行页面置换时，也可采用两种策略，即全局置换和局部置换。于是可组合成以下三种适用的策略。

1）固定分配局部置换策略

基于进程的类型（交互型或批处理型等），或根据用户、系统管理员的建议，为每个进程

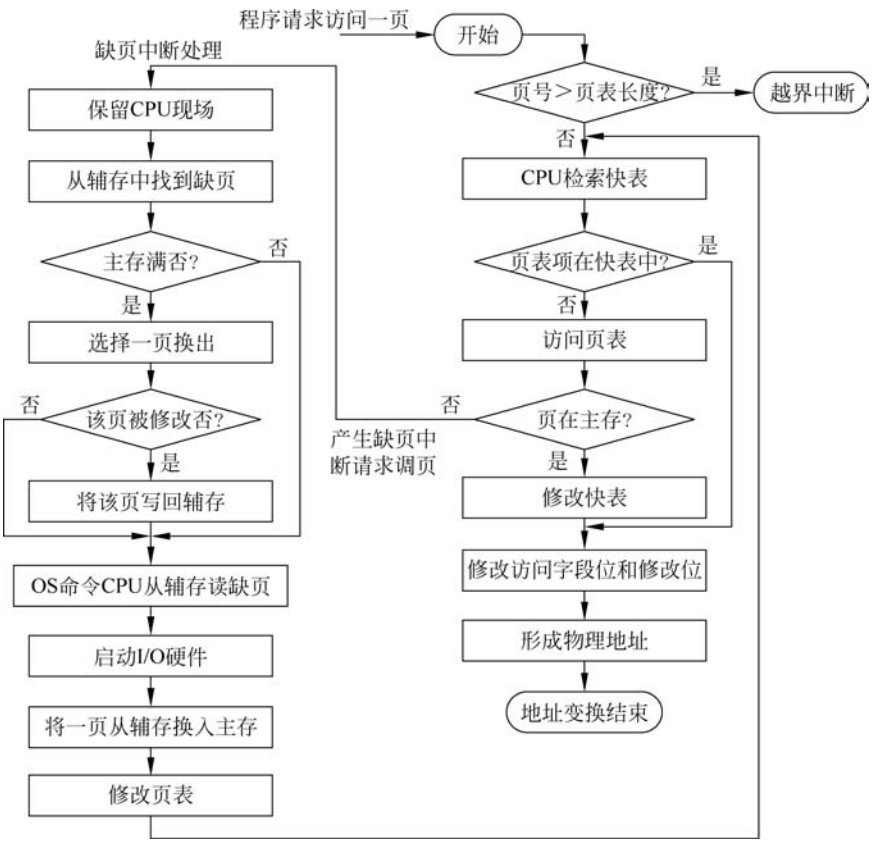


图 5-34 请求分页式存储管理地址变换

分配一定数目的主存物理块,在整个运行期间都不再改变。采用该策略时,如果进程在运行中发生缺页,则只能从该进程在主存的 n 个页面中选出一页换出,然后再调入一页,以保证分配给该进程的主存空间不变。实现这种策略的困难在于,应当为每个进程分配物理块的数量难以确定。若太少,会频繁地出现缺页中断,降低系统的吞吐量;若太多,又必然使主存中驻留的进程数目减少,可能造成 CPU 空闲或其他资源空闲的情况,而且在实现进程对换时,会花费更多的时间。

2) 可变分配局部置换策略

基于进程的类型,或根据用户、系统管理员的建议,为每个进程分配一定数目的主存物理块;但当进程发生缺页中断时,只允许从该进程在主存的页面中选择淘汰页,而不影响其他进程的运行。如果进程在运行中频繁地发生缺页中断,则系统须再为该进程分配若干附加的物理块,直至该进程的缺页率减少到适当程度为止;反之,若一个进程在运行过程中的缺页率特别低,则可适当减少分配给该进程的物理块数,但不应引起其缺页率的明显增加。

3) 可变分配全局置换策略

最易于实现的一种页面分配和置换策略。系统先为每个进程分配一定数目的物理块,而操作系统保持一个空闲块队列。当一个进程发生缺页中断时,系统从空闲块队列中取出一块,分配给该进程。当空闲物理块队列中的物理块用完时,操作系统才能从主存中选择一页面置换,该页可能是系统中任一进程的页。

采用固定分配策略时，可采用以下几种物理块分配方法。

(1) 平均分配算法。将系统中所有可供分配的物理块平均分配给每个进程。

(2) 按比例分配算法。根据进程的大小按比例分配物理块。

(3) 考虑优先权的分配算法。该方法是把主存中可供分配的所有物理块分为两部分，一部分按比例分配给每个进程；另一部分则根据进程的优先权，适当地增加其相应份额后，分配给各进程。


视频讲解

3. 页面置换算法

在作业运行过程中，如果所要访问的页面不在主存中，就需要把它们调入主存。当主存中已没有空闲空间时，为了保证作业的运行，系统必须按一定的算法选择一个已在主存中的页面，将它暂时调出主存，让出主存空间，用来存放所需调入的页面，这个工作称为页面置换。选择换出页面的算法称为页面置换算法。置换算法的好坏将直接影响到系统的性能。如果选用一个不合适的算法，就会出现这样的现象：刚被调出的页面又立即要用，因而又要把它调入，而调入不久又被选中调出，调出不久又被调入……如此反复，使调度非常频繁，以至于大部分时间都花费在来回调度上。这种现象称为“抖动”或称“颠簸”。一个好的置换算法应该尽可能地减少和避免抖动现象的发生。

从理论上讲，应将那些以后不再访问的页面换出，或把那些在今后较长时间不会访问的页面调出。这是一种理想化的算法，具有很好的性能，但实际上却难以实现。常用的页面置换算法有最佳置换算法、先进先出置换算法、最近最少用置换算法和最近最不常用置换算法等，它们都试图接近理论上的目标。

1) 最佳置换算法

最佳(Optimal，OPT)置换算法是由 Belady 于 1966 年提出的一种理论上的算法。该算法选择的被淘汰页面将永远不再使用，或者是在将来最长时间内不再被访问的页面，这样产生的缺页中断次数将会是最少的。采用 OPT 置换算法通常可获得最低的缺页中断率，然而，却需要预测出程序的页面引用串，这是无法预知的，不可能对程序的运行过程做出精确的断言，所以说这是一种理想化的算法，无法实现。但是这个算法可以作为衡量其他算法的标准。

假定某进程共有 8 页，且系统为之分配了 3 个物理块，并产生以下页面调度序列：

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

进程运行时，首先通过缺页中断，把 7、0、1 三个页面顺序装入主存。当进程访问页面 2 时，将会产生缺页中断，操作系统根据 OPT 置换算法，得出：0 号页面的下次访问将是本进程第 5 次被访问的页面，1 号页面的下次访问将是第 14 次被访问的页面，而 7 号页面将要在第 18 次页面访问时才需要调入，所以将选择 7 号页面予以淘汰。接下来访问 0 号页面，由于它已调入主存，不会产生缺页中断，当系统访问 3 号页面时，由于在已调入主存的 1、2、0 这三个页面中，1 号页面将会最迟才被访问，因此将 1 号页面淘汰。如图 5-35 所示为 OPT 置换算法的置换过程。可以看出，采用 OPT 置换算法只发生了 9 次页面置换，缺页中断率为 45%。

2) 先进先出置换算法

先进先出(First-In-First-Out，FIFO)置换算法认为刚被调入的页面在最近的将来被访问的可能性很大，而在主存中驻留时间最长的页面在最近的将来被访问的可能性最小。因此，FIFO 置换算法总是淘汰最先进入主存的页面，即淘汰在主存中驻留时间最长的页面。

FIFO 置换算法只需要把装入主存的页面按调入的先后次序连接成一个队列，并设置

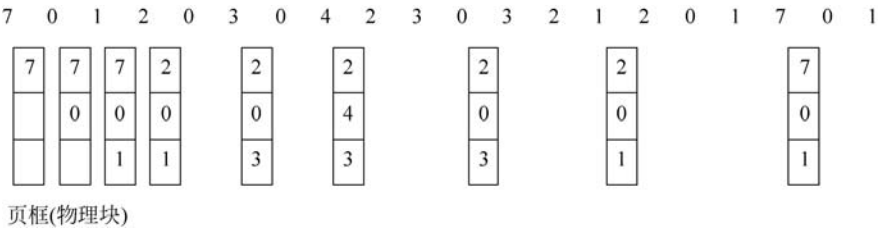


图 5-35 最佳页面置换算法的置换图

一个替换指针,指针始终指向最先装入主存的页面,每次页面置换时,总是选择替换指针所指示的页面调出。如图 5-36 所示为在 OPT 置换算法的例子中采用 FIFO 置换算法时保留在主存中页面变化的情况。进程运行时,通过缺页中断,先将 7、0、1 这 3 个页面顺序装入主存。当进程访问 2 号页面时,系统将产生缺页中断。由于 7 号页面是最先调入主存的,因此将它换出,下面访问 0 号页面。由于它已调入主存,不产生缺页中断。当系统访问 3 号页面时,由于在已调入主存的 1、2、0 三个页面中,0 号页面最早进入主存,因此将 0 号页面换出。从图 5-36 可以看出,采用 FIFO 置换算法一共发生了 15 次页面置换,缺页中断率为 75%,页面淘汰的顺序为 7、0、1、2、3、0、4、2、3、0、1、2。

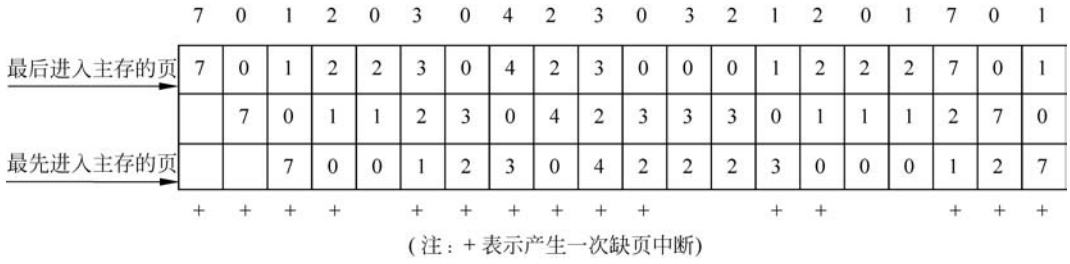


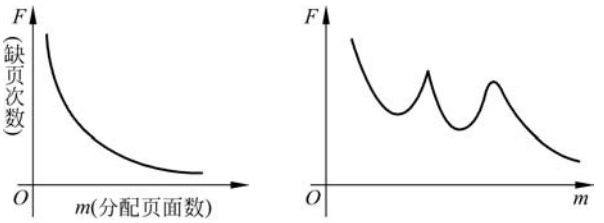
图 5-36 FIFO 置换算法主存中页面变化示意图

FIFO 置换算法简单,易实现,但效率不高。因为在主存中驻留时间最久的页面未必是在不久的将来最长时间不再被访问的页面,如页面中含有全局变量、常用函数、例程等,如果将它淘汰,可能立即又要使用,必须重新调入,尽管这些页面变“老”了,但它们被访问的概率仍然很高。据估计,采用 FIFO 置换算法产生的缺页中断率约为最佳置换算法的 3 倍。

FIFO 置换算法存在一种异常现象。一般来说,对于任何一个作业,系统分配给它的主存物理块数越接近于它所要求的页面数,发生缺页中断的次数会越少。如果一个作业能获得它所要求的全部物理块数,则不会发生缺页中断现象。但是,采用 FIFO 置换算法时,在未给作业分配足够多的它所要求的页面数时,有时会出现这样的奇怪现象:分配的物理块数增多,而缺页中断次数反而增加。这种现象称为 Belady 现象,如图 5-37 所示。

下面举例说明 FIFO 置换算法的正常置换页面情况和 Belady 现象。假如某进程共有 8 页,依次访问页面的序列为:7、0、1、2、0、3、0、4、2、3、0、3、2、1、2、0、1。当系统为该进程分配的物理块数 $M=3$ 时,缺页中断次数为 12 次,其缺页中断率为 12/17=70.5%,如图 5-38(a)所示;而当分配的物理块数 $M=4$ 时,缺页中断次数为 9 次,其缺页中断率为 9/17=52.9%,如图 5-38(b)所示。

以上是采用 FIFO 置换算法正常置换页面的例子,下面分析另一个示例。某进程共有



(a) 正常情况　　(b) Belady现象

图 5-37　FIFO 置换算法的 Belady 现象

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
最后进入主存的页	7	0	1	2	2	3	0	4	2	3	0	0	0	1	2	2	2
		7	0	1	1	2	3	0	4	2	3	3	3	0	1	1	1
最先进入主存的页			7	0	0	1	2	3	0	4	2	2	2	3	0	0	0
	+	+	+	+		+	+	+	+	+	+			+	+		

(a) M=3时，无Belady现象示例

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
最后进入主存的页	7	0	1	2	2	3	3	4	4	4	0	0	0	1	2	2	2
		7	0	1	1	2	2	3	3	3	4	4	4	0	1	1	1
			7	0	0	1	1	2	2	2	3	3	3	4	0	0	0
最先进入主存的页				7	7	0	0	1	1	1	2	2	2	3	4	4	4
	+	+	+	+		+		+			+			+	+		

(b) M=4时，无Belady现象示例

	1	2	3	4	1	2	5	1	2	3	4	5
最后进入主存的页	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
最先进入主存的页			1	2	3	4	1	1	1	2	5	5
	+	+	+	+	+	+	+			+	+	

(c) M=3时，Belady现象示例

	1	2	3	4	1	2	5	1	2	3	4	5
最后进入主存的页	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
最先进入主存的页				1	1	1	2	3	4	5	1	2
	+	+	+	+			+	+	+	+	+	+

(d) M=4时，Belady现象示例

(注：+表示产生一次缺页中断)

图 5-38　Belady 现象示例

5 页，依次访问页面的序列为：1、2、3、4、1、2、5、1、2、3、4、5。当系统为该进程分配的物理块数M＝3 时，缺页中断次数为 9 次，其缺页中断率为 9/12＝75%，如图 5-38(c)所示；但是如果为进程分配的物理块数 M＝4 时，缺页中断次数为 10 次，其缺页中断率为 10/12＝83.3%，如图 5-38(d)所示。

先进先出算法产生 Belady 现象的原因在于它根本没有考虑程序执行的动态特征。

3) 最近最少用置换算法

最近最少用(Least Recently Used,LRU)置换算法总是选择最近一段时间内最长时间没有被访问过的页面调出。

LRU 置换算法的提出基于程序执行的局部性原理,即认为那些刚被访问的页面可能在最近的将来还会经常访问它们,而那些在较长时间里未被访问的页面,一般在最近的将来再被访问的可能性较小。为了记录页面自上次被访问以来所经过的时间,需要在页表中增加一个引用位标志,在每次被访问后将引用位置 1,重新计时。这样,在发生缺页中断需要调入新的页面时,通过检查页表中各页的引用位,选择将最长时间没有被访问过的页面淘汰,并且把主存中所有页面的引用位全部清零,重新计时。

如图 5-39 所示为在 OPT 置换算法的例子中采用 LRU 置换算法时,保留在主存中页面的变化情况。该进程执行过程中,共产生了 12 次缺页中断,缺页中断率为 60%,页面淘汰的顺序为 7、1、2、3、0、4、0、3、2。

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
最近被访问的页 →	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
最长时间未被访问的页 →		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0
			7	0	1	2	2	3	0	4	2	2	0	3	3	1	2	0	1	7
	+	+	+	+		+		+	+	+	+			+		+		+		

(注:+表示产生一次缺页中断)

图 5-39 LRU 近似算法主存页面变化示意图

要完全实现 LRU 置换算法是一件非常困难的事情。因为需要对每一页被访问的情况进行实时记录和更新,这显然要花费巨大的系统开销,所以在实际系统中往往使用 LRU 近似算法。

LRU 近似算法[即时钟(clock)置换算法]是在页表中为每一页增加一个引用位信息,当该页被访问时,由硬件将它的引用位信息置为 1,操作系统选择一个时间周期 T,每隔一个周期 T,将页表中所有页面的引用位信息置 0,这样,在时间周期 T 内,被访问过的页面的引用位为 1,而没有被访问过的页面的引用位仍为 0。当产生缺页中断时,可以从引用位为 0 的页面中选择一页调出,同时将所有页面的引用位信息全部重新置 0。这种近似算法的实现比较简单,但关键在于时间周期 T 的确定。如果 T 太大,可能所有的引用位都变成 1,找不出最近最少使用的页面淘汰;如果 T 太小,引用位为 0 的页面可能很多,而无法保证所选择的页面是最近最少使用的。

淘汰一个页面时,如果该页面已被修改过,必须将它重新写回磁盘;但如果淘汰的是未被修改过的页面,就不需要写盘操作了,这样看来,淘汰修改过的页面比淘汰未被修改过的页面的开销要大。如果把页表中的“引用位”和“修改位”结合起来使用,可以改进时钟页面替换算法,它们一共组合成 4 种情况:

① 最近没有被引用,没有被修改($r=0$,$m=0$);

② 最近被引用,没有被修改($r=1$,$m=0$);

③ 最近没有被引用,但被修改过($r=0$,$m=1$);

④ 最近被引用过，也被修改过（$r=1$，$m=1$）。

改进的时钟页面替换算法就是扫描队列中的所有页面，寻找一个既没有被修改且最近又没有被引用过的页面，把这样的页面挑出来作为首选页面淘汰是因为没有被修改过，淘汰时不用把它写回磁盘。如果第一步没有找到这样的页面，算法再次扫描队列，欲寻找一个被修改过但最近没有被引用过的页面；虽然淘汰这种页面需写回磁盘，但依据程序局部性原理，这类页面一般不会马上被再次使用。如果第二步也失败了，则所有页面已被标记为最近未被引用，可进入第三步扫描。Macintosh 的虚存管理采用了这种策略，其主要优点是没有被修改过的页面会被优先选出来，淘汰这种页面时不必写回磁盘，从而节省时间，但查找一个淘汰页面可能会经过多轮扫描，算法实现的开销较大。

4）最近最不常用置换算法

最近最不常用置换算法（Least Frequently Used，LFU）总是选择被访问次数最少的页面调出，即认为在过去的一段时间里被访问次数多的页面可能经常需要访问。

一种简单的实现方法是为每一页设置一个计数器，页面每次被访问后，其对应的计数器加 1，每隔一定的时间周期 T，将所有计数器全部清 0。这样，在发生缺页中断时，选择计数器值最小的对应页面被淘汰，显然它是最近最不常用的页面，同时把所有计数器清 0。这种算法的实现比较简单，但代价很高，同时有一个关键问题是如何选择一个合适的时间周期 T。

4. 缺页中断率分析

虚拟存储系统解决了有限主存的容量限制问题，能使更多的作业同时多道运行，从而提高了系统的效率，但缺页中断处理需要系统的额外开销，影响了系统效率，因此应尽可能地减少缺页中断的次数，降低缺页中断率。

假定一个作业共有 n 页，系统分配给它的主存块是 m 块（m、n 均为正整数，且 $1\leqslant m\leqslant n$），则该作业最多有 m 页可同时被装入主存。如果作业执行中访问页面的总次数为 A，其中，有 F 次访问的页面尚未装入主存，故产生了 F 次缺页中断。现定义 $f=F/A$，则 f 称为缺页中断率。

显然，缺页中断率与缺页中断的次数有关。因此，影响缺页中断率的因素有以下几点。

1）分配给作业的主存块数

一般情况下，系统分配给作业的主存块数多，则该作业同时装入主存的页面数就多，那么缺页中断次数就少，缺页中断率就低；反之，缺页中断率就高。

从理论上说，在虚拟存储器的环境下，每个作业只要获得一块主存空间就可以开始执行，这样可以增加在主存中多道程序的道数，但是每个作业将会频繁地发生缺页中断，效率非常低下。但是如果为每个作业分配很多主存块，必然会降低系统的多道程序度，影响系统的吞吐量。图 5-40 给出了 CPU 的利用率与多道程序之间的关系。

进程的缺页中断率和进程占有主存页面数的关系如图 5-41 所示。一个程序面对较少的主存物理块数时，发生的缺页中断次数就多；当分配的主存物理块数量增加时，缺页中断次数就减少。但从图中可以看出，主存物理块数增加到临界值以后，即使再增加较多的物理块，进程的缺页中断次数也不再明显减少。大多数程序都有这样一个特定点，在这个特定点以后再增加主存容量时，缺页中断次数的减少并不明显。

工作集的理论是由 Denning 提出来的。他认为，程序在运行时对页面的访问是不均匀

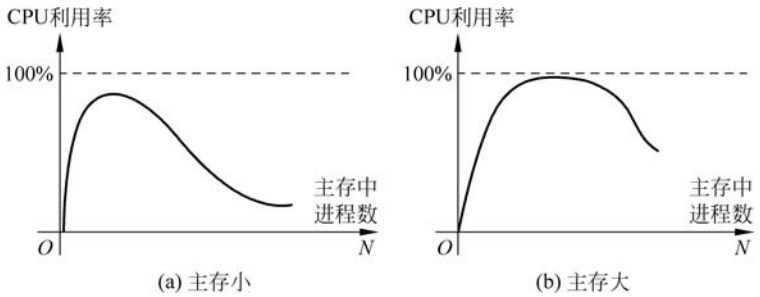


图 5-40 处理器利用率与进程数的关系

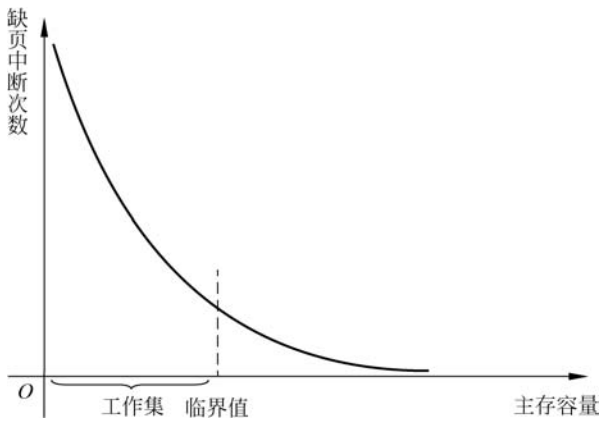


图 5-41 主存容量与缺页中断的关系

的,即往往在某段时间内的访问仅局限于较少的若干页面,如果能够预知程序在某段时间间隔内要访问哪些页面,并能将它们提前调入主存,将会大大地降低缺页率,从而减少置换工作,提高 CPU 的利用率。

对于给定的访问序列选取定长的时间间隔(Δ),称为工作集窗口,落在工作集窗口中的页面集合称为工作集。工作集是指在某段时间间隔里,进程实际要访问的页面的集合。Denning 认为,虽然程序只需要有少量几页在主存就可以运行,但为了使程序能够有效运行,较少地产生缺页,就必须使程序的工作集全部在主存中。把某进程在时间 t 的工作集记为 $w(t,\Delta)$,把变量 Δ 称为工作集"窗口尺寸"。由于无法预知一个程序在最近的将来会访问哪些页面,所以用最近在 Δ 时间间隔内访问过的页面作为实际工作集的近似。正确选择工作集窗口尺寸的大小对系统性能有很大影响,如果 Δ 过大,甚至把作业地址空间全包括在内,就成了实存管理;如果 Δ 过小,则会引起频繁缺页,降低了系统的效率。图 5-42 给出了不同窗口尺寸的工作集变化示意图,其中列出了进程的引用序列,* 表示这个时间单位里工作集没有发生改变。从图 5-42 中可以看出,工作集窗口尺寸越大,工作集就越大,产生缺页中断的频率越低。

假如有以下页面访问序列,窗口尺寸 Δ=9:

2615777516234123444343444 1327

|----------| |----------|

Δ t_1 Δ t_2

页面访问序列1	窗口尺寸 2	3	4	5
24	24	24	24	24
15	15 24	15 24	15 24	15 24
18	18 15	18 15 24	18 15 24	18 15 24
23	23 18	23 18 15	23 18 15 24	23 18 15 24
24	24 23	24 23 18	*	*
17	17 24	17 24 23	17 24 23 18	17 24 23 18 15
18	18 17	18 17 24	*	*
24	24 18	*	*	*
18	18 24	*	*	*
17	17 18	*	*	*
17	17	*	*	*
15	15 17	15 17 18	15 17 18 24	*
24	24 15	24 15 17	*	*
17	17 24	*	*	*
24	24 17	*	*	*
18	18 24	18 24 17	*	*

图 5-42　进程工作集

则 t_1 刻的工作集 $w(t_1)=\{1,2,5,6,7\}$；t_2 时刻的工作集 $w(t_2)=\{3,4\}$。

在有些操作系统中(如 Windows)，虚拟存储管理程序为每一个进程分配固定数量的物理块，并且这个数目可以进行动态调整。这个数目就是由每个进程的工作集来确定，并且根据主存的负荷和进程的缺页情况动态地调整其工作集。

其具体的做法是：一个进程在创建时就指定了一个最小工作集，该工作集的大小是保证进程运行在主存中应有的最小页面数。但在主存负荷不太大时，虚存管理程序还允许进程拥有尽可能多的页面作为其最大工作集。当主存负荷发生变化时，如空闲页不多了，虚存管理程序就使用“自动调整工作集”的技术来增加主存中可用的自由页面数，方法是检查主存中的每个进程，将当前工作集大小与最小工作集进行比较，如果大于其最小值，则从它们的工作集中移去一些页面作为主存自由页面，可被其他进程使用。若主存自由页面仍然太小，则不断进行检查，直到每个进程的工作集都达到最小值为止。

当每个工作集都已达到最小值时，虚存管理程序跟踪进程的缺页数量，根据主存中自由页面数量可以适当增加其工作集的大小。

2) 页面的大小

页面的大小影响页表的长度、页表的检索时间、置换页面的时间、可能的页内零头的大小等，对缺页中断的次数也有一定的影响。如果划分的页面大，则一个作业的页面数就少，页表所占用的主存空间少，且查表速度快，在系统分配相同主存块的情况下，发生缺页中断的次数减少，降低了缺页中断率。但是，一次换页需要的时间延长，可能产生的页内零头所带来的空间浪费较大。反之，所划分的页面小，一次换页需要的时间就短，可能产生的页内零头少，空间利用率提高；但是一个作业的页面数多，页表所占用的主存空间长，且查表速度慢，在系统分配相同主存块的情况下，发生缺页中断的次数增多，增加了缺页中断率。

因此，页面的大小应根据实际情况来确定，对于不同的计算机系统，页面大小有所不同。一般页面大小为 1～4KB。

3) 程序编制方法

不同的程序编制方法对缺页中断的次数也有很大影响。缺页中断率与程序的局部化程度密切相关。一般来说，希望编制的程序能经常集中在几个页面上进行访问，以减少缺页中

断次数,降低缺页中断率。

例如,一个程序要将128×128数组置初值0。现假设分配给该程序的主存块数只有一块,页面的大小为每页128个字,数组中每一行元素存放在一页中。开始时,第一页已经调入主存。若程序如下编制:

```
int A[128][128];
for (j = 0;j < 128;j++)
  for (i = 0;i < 128;i++)
    A[i][j] = 0;
```

则每执行一次A[i][j]=0就要产生一次缺页中断,总共需要产生(128×128-1)次缺页中断。

如果重新编制这个程序如下:

```
int A[128][128];
for (i = 0;i < 128;i++)
  for (j = 0;j < 128;j++)
    A[i][j] = 0;
```

则总共产生(128-1)次缺页中断。显然,虚拟存储器的效率与程序局部性程度密切相关,局部性程度因程序而异。

4) 页面调度算法

页面调度算法对于缺页中断率的影响也很大,如果选择不当,则有可能产生抖动现象。理想的调度算法是当要装入一个新页而必须调出一个页面时,所选择的调出页应该是以后再也不使用的页或者是距离当前最长时间以后才使用的页。算法的优劣影响缺页中断的次数,从而影响缺页中断率。

5.7.3 请求分段式存储管理

请求分段式存储管理以段式存储管理为基础,为用户提供比主存实际容量更大的虚拟空间。请求分段式存储管理把作业的各个分段信息保留在磁盘上,当作业被调度进入主存时,首先把当前需要的一段或几段装入主存,便可启动执行,若所要访问的段已在主存,则将逻辑地址转换成物理地址;如果所要访问的段尚未调入主存,则产生一个缺段中断,请求操作系统将所要访问的段调入。为实现请求分段式存储管理,需要有一定的硬件支持和相应的软件。

请求分段式存储管理所需要的硬件支持有:段表机制、缺段中断机构以及地址变换机构等。

1. 段表机制

在请求分段式存储管理中,段表是进行段调度的主要依据。在段表中需要增设一些标志信息,如段是否在主存,各段在磁盘上的存储位置,已在主存的段需要指出该段在主存中的起始地址和占用主存区的长度等,还可设置该段是否被修改、是否可扩充等。请求分段式存储管理的段表结构如下:

段名	段长	段的基址	存取方式	访问字段A	修改位M	状态位P	扩充位	辅存始址

其中,“存取方式”标识本段的存取属性(只执行或只读或读/写);“访问字段A”记录该段被

访问的频繁程度；“修改位 M”表示该段进入主存后是否已被修改，供置换段时参考；“状态位 P”指示本段是否已调入主存，若已调入，则“段的基址”给出该段在主存中的起始地址，否则在“辅存始址”中指示出本段在辅存中的起始地址；“扩充位”是请求分段式存储管理中所特有的字段，表示本段在运行过程中是否有动态增长。

2. 缺段中断机构

在请求分段式存储管理中，当所要访问的段尚未调入主存时，则由缺段中断机构产生一个缺段中断信号，请求操作系统将所要访问的段调入主存。操作系统处理缺段中断的步骤如下。

(1) 空间分配。查主存分配表，找出一个足够大的连续区以容纳该分段。如果找不到足够大的连续区，则检查主存中空闲区的总和，若空闲区总和能满足该段要求，则进行适当移动，将分散的空闲区集中；若空闲区总和不能满足该段要求，则可选择将主存中的一段或几段调出，然后把当前要访问的段装入主存。

(2) 修改段表。段被移动、调出和装入后，都要对段表中的相应表目进行修改。

(3) 新的段被装入后，应让作业重新执行被中断的指令，这时就能在主存中找到所要访问的段，可以继续执行下去。

3. 地址变换机构

请求分段式存储管理方式中的地址变换是在分段式存储管理系统的地址变换机构的基础上增加缺段中段请求及处理等形成的。如图 5-43 所示为请求分段式存储管理系统的地址变换过程。

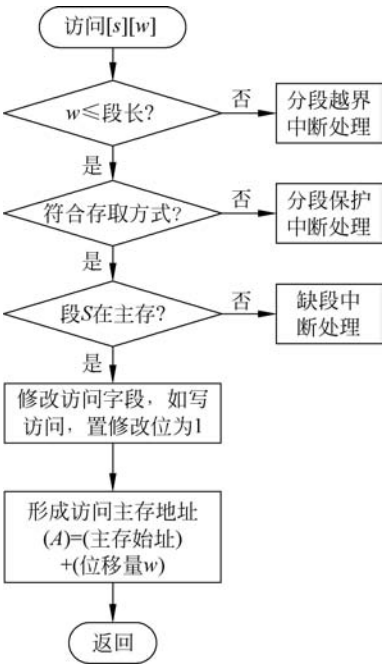


图 5-43　请求分段式存储管理系统的地址变换过程

5.7.4　请求段页式存储管理

请求分段式存储管理以段为单位进行主存空间的分配。整段信息的装入、调出，有时需要主存空间的移动，这些都要增加系统的开销。如果按段页式存储管理方式，则每个作业仍然按逻辑分段，把每一段再分成若干页面。这样，在请求式存储管理中，每一段不必占用连续的存储空间，可按页存放在不连续的主存块中，甚至当主存块不足时，可以将一段中的部分页面装入主存。这种管理方式称为请求段页式存储管理。

采用请求段页式存储管理时，需要对每一个装入主存的作业建立一张段表，对每一段建立一张页表。段表中指出该段对应页表所存放的起始地址及其长度，页表中应指出该段的每一页在磁盘上的位置以及该页是否在主存中。若在主存中，则填上占用的主存块号。作业执行时按段号查找段表，找到相应的页表，再根据页号查找页表，由状态位判定该页是否已在主存中，若在，则进行地址转换，否则进行页面调度。

请求段页式存储管理结合了请求分段式虚拟管理和请求分页式虚拟管理的优点，但增加了设置表格(段表、页表)和查表等开销。目前将实现虚拟所需支持的硬件集成在 CPU 芯片上，例如，Intel 80386 以上的 CPU 芯片都支持请求段页式存储管理。段页式虚拟存储管理一般只在大型计算机系统中采用。

5.8 Linux 存储管理

Linux 支持虚拟主存,使用磁盘作为 RAM 的扩展,使可用主存相应地有效扩大。内核把当前不用的主存块信息调出到硬盘,腾出主存用于其他目的。当调出的内容又需要使用时,再读入主存。这对于用户是透明的:运行于 Linux 的程序只看到大量的可用主存而不关心哪些部分在磁盘上。当然,读写虚拟主存(硬盘部分)比读写主存要慢(差距约为 10^3 量级),所以程序运行较慢。通常用作虚拟主存的这部分硬盘叫交换空间。

5.8.1 Linux 的请求分页存储管理

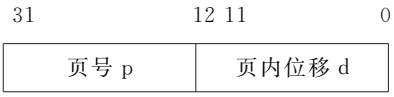
Linux 系统采用了虚拟主存管理机制,即交换和请求分页存储管理技术。当进程运行时,不必把整个进程的映像都放在主存中,只需在主存中保留当前用到的那一部分页面。当进程访问到某些尚未在主存的页面时,就由内核把这些页面装入主存。

为了实现离散存储,虚拟主存与物理主存都以大小相同的页面来组织。页面模式下的虚拟地址由两部分构成:页面框号和页面内偏移值。在页表的帮助下,它将虚拟页面框号转换成物理页面框号,然后访问物理页面中相应偏移处。

1. 请求分页机制

下面介绍分页存储管理的基本方法。

在一般的分页存储管理方式中,表示地址的结构如下:



地址由两个部分组成:前一部分表示该地址所在页面的页号 p;后一部分表示页内位移 d,即页内地址。其中 0～11 为页内位移,即每页的大小为 4KB;12～31 位为页号,表示地址空间中最多可容纳 1M 个页面。

系统又为每个进程设立一张页面映像表,简称页表。在进程地址空间内的所有页(0～$n-1$)依次在页表中有一个页表项,其中记载了相应页面在主存中对应的物理块号、页表项有效标志,以及相应主存块的访问控制属性(如只读、只写、可读写、可执行)。进程执行时,按照逻辑地址中的页号去查找页表中的对应项,可从中找到该页在主存中的物理块号。然后,将物理块号与对应的页内位移拼接起来,形成实际的访问主存地址。所以,页表的作用是实现从页号到物理块号的地址映射。

页表入口包含了访问控制信息,因此可以很方便地使用访问控制信息来判断处理器是否在以其应有的方式来访问主存。

多数通用处理器同时支持物理寻址模式和虚拟寻址模式,物理寻址模式无须页表的参与,且处理器不会进行任何地址转换,Linux 内核直接运行在物理地址空间上。多数处理器至少有两种执行方式——内核态与用户态。任何人都不会允许在用户态下执行内核代码或者在用户态下修改内核数据结构。一般的用户进程只能在限定的空间内访问,若要使用某些内核提供的功能,只有通过系统调用实现。

2. 请求分页的基本思想

请求分页存储管理技术是在简单分页技术的基础上发展起来的，二者的根本区别在于请求分页提供虚拟存储器。

它的基本思想是，当要执行一个程序时才把它换入主存；但并不把全部程序都换入主存，而是用到哪一页时才换入它。这样就减少了对换时间和所需主存的数量，允许增加程序的道数。

为了表示一个页面是否已装入主存块，在每一个页表项中增加一个状态位，Y 表示该页对应的主存块可以访问；N 表示该页不对应主存块，即该页尚未装入主存，不能立即进行访问。

如果地址转换机构遇到一个具有 N 状态的页表项时，便产生一个缺页中断，告诉 CPU 当前要访问的这个页面还未装入主存。操作系统必须处理这个中断：它装入所要求的页面，并相应地调整页表的记录，然后重新启动该指令。

由于这种页面是根据请求而被装入的，所以这种存储管理方法称为请求分页存储管理。通常在作业最初投入运行时，仅把它的少量几页装入主存，其他各页是按照请求顺序动态装入的，这样就保证了用不到的页面不会被装入主存。

3. 地址映射

执行进程时，可执行的命令文件被打开，同时其内容被映射到进程的虚拟主存。这时可执行文件实际上并没有调入物理主存，而是仅仅连接到进程的虚拟主存。当程序中其他部分运行时，在引用到这部分的时候才把它们从磁盘上调入主存。将虚拟地址转换成物理地址的过程称为地址映射。

每个进程的虚拟主存用一个 mm_struct 表示。它包含当前执行的映像以及指向 vm_area_struct 的大量指针。每个 vm_area_struct 数据结构描述了虚拟主存的起始与结束的位置、进程对此主存区域的存取权限以及一组主存操作函数。这组主存操作函数都是 Linux 在操纵虚拟主存区域时必须用到的子程序。其中一个负责处理进程试图访问不在当前物理主存中的虚拟主存(通过缺页中断)的情况，此函数叫 nopage。它用于 Linux 试图将可执行映像的页面调入主存时。

可执行程序映射到进程虚拟地址时，将产生一组相应的 vm_area_struct 数据结构。每个 vm_area_struct 数据结构表示可执行程序的一部分，包括可执行代码、初始化数据(变量)、未初始化数据等。

5.8.2　Linux 的多级页表

在 X86 平台的 Linux 系统中，地址码采用 32 位，因而每个进程的虚存空间可达 4GB。Linux 内核将这 4GB 的空间分为两部分：最高地址的 1GB 是“系统空间”，供内核本身使用；而较低地址的 3GB 是各进程的“用户空间”。

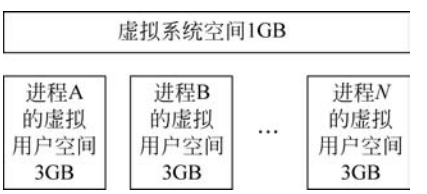


图 5-44　Linux 进程的虚存空间

系统空间由所有进程共享。虽然理论上每个进程的可用用户空间都是 3GB，但实际的存储空间的大小要受到物理存储器(包括主存及磁盘交换区或交换文件)的限制。进程的虚存空间如图 5-44 所示。

由于Linux系统中页面大小为4KB,因此进程虚存空间要划分为1M个页面。如果直接用页表描述这种映射关系,每个进程的页表就要有1M个表项。显然,用大量的主存资源来存放页表的办法是不可取的。为此,Linux系统采用三级页表的方式,如图5-45所示。每个页表通过所包含的下级页表的页框号(Page Frame Number,PFN)来访问。Linux的三级页表的说明如下:

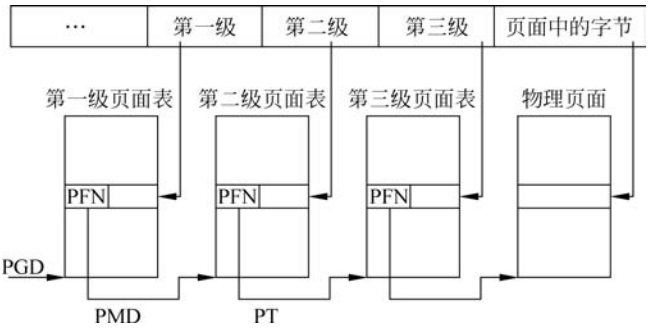


图5-45 三级页表地址映射示意图

(1) 页目录(PaGe Directory,PGD)(第一级):每个活动进程有一个一页大小的页目录,其中的每一项指向页间目录中的一页。

(2) 页间目录(Page Median Directory,PMD)(第二级):页间目录可以由多个页面组成,其中的每一项指向页表中的一页。

(3) 页表(Page Table,PT)(第三级):页表也可以由多个页面组成,每个页表项指向进程的一个虚页。

Linux的虚地址由4个域组成,页目录是第一级索引,由它找到页间目录的起始位置。第二级索引是根据页间目录的值确定页表的起始位置,它通过页表的值找到物理页面的页框号,再加上最后一个域的页内偏移量,得到页面中数据的地址。

图5-45中PGD表示页面目录,PMD表示中间目录,PT表示页表。一个线性的虚拟地址在逻辑上从高位到低位划分成4个位段,分别用作页面目录PGD中的下标、页间目录PMD中的下标、页表PT中的下标和物理页面(即主存块)内的位移。

这样,把一个线性地址映射成物理地址分为以下4步。

(1) 以线性地址中最高位段作下标,在PGD中找到相应的表项,该表项指向相应的PMD。

(2) 以线性地址中第二个位段作下标,在PMD中找到相应的表项,该表项指向相应的PT。

(3) 以线性地址中第三个位段作下标,在PT中找到相应的表项,该表项指向相应的物理页面(即该物理页面的起始地址)。

(4) 线性地址中的最低位段是物理页面内的相对位移量,将此位移量与该物理页面的起始地址相加,就得到相应的物理地址。

5.8.3 Linux主存页的缺页中断

由于系统的物理主存是一定的,当有多道进程同时在系统中运行时,物理主存往往会不够用。为了提高物理主存的使用效率,操作系统采用的方法是仅加载那些正在被执行程序

使用的虚拟页面。这种仅将要访问的虚拟页面载入的技术称为请求换页。

当进程试图访问当前不在主存中的虚拟地址时，CPU在页表中无法找到所引用地址的入口，引发一个页面访问失效，操作系统将得到有关无效虚拟地址的信息以及发生页面错误的原因。如果发生页面错误的虚拟地址是无效的，则可能是应用程序出错而引起的。此时操作系统将终止该进程的运行以保护系统中其他进程不受此出错进程的影响。若出错虚拟地址是有效的，但不在主存中，则操作系统必须将此页面从磁盘文件中读入主存。在调入页面时，调度程序会选择一个就绪进程来运行。读取回来的页面将被放在一个空闲的物理页面框中，同时修改页表。最后进程将从发生页面错误的地方重新开始运行。以上过程即为缺页中断。

如果进程需要把一个虚拟页面调入物理主存而正好系统中没有空闲的物理页面，操作系统必须淘汰位于物理主存中的某些页面来为之腾出空间。Linux使用最近最少使用(Least Recently Used，LRU)页面置换算法来公平地选择将要从系统中抛弃的页面。这种策略为系统中的每个页面设置一个年龄(age)，它随页面访问次数而变化。页面被访问的次数越多，则页面年龄越年轻；相反则越衰老。年龄较老的页面是待交换页面的最佳候选者。

5.8.4 Linux主存空间的分配与回收

1. 数据结构

当一个可执行映像被调入主存时，操作系统必须为其分配页面。当映像执行完毕和卸载时，这些页面必须被释放。虚拟主存子系统中负责页面分配与回收的数据结构用mem_map_t结构的链表mem_map来描述，这些结构在系统启动时初始化。每个mem_map_t描述了一个物理页面，其中几个重要的域如下。

(1) count：记录使用此页面的用户个数。当这个页面在多个进程之间共享时，其值大于1。

(2) age：此域用于描述页面的年龄，用于选择将适当的页面抛弃或者置换出主存。

(3) map_nr：记录本mem_map_t描述的物理页面框号。

系统使用free_area数组管理整个缓冲，实现分配和释放页面。free_area中的每个元素都包含页面块的信息，第i个元素描述$2i$个大小的页面。list域表示一个队列头，它包含指向mem_map数组中page数据结构的指针。所有的空闲页面都在此队列中。map域是指向页面组分配情况位图的指针，当页面的第N块空闲时，位图的第N位被置位。

2. 主存页的分配和释放

Linux使用Buddy算法来有效地分配与回收页面块。页面分配程序每次分配包含一个或者多个物理页面的主存块。页面以2的次幂的主存块来分配。这意味着它可以分配1个、2个和4个页面的块。只要系统中有足够的空闲页面来满足这个要求(nr_free_pages > min_free_page)，主存分配代码将在free_area中寻找一个与请求大小相同的空闲块。free_area中的每个元素保存着一个反映这样大小的已分配与空闲页面的位图。分配算法采用首次适应算法，从free_area的list域沿链搜索空闲页面，若找到的页面块大于请求的块，则对其进行分割，以使其大小与请求块匹配，剩余的空闲块被放进相应的队列。

当一个进程开始运行时，系统要为其分配一些主存页；而当该进程结束运行时，要释放其所占用的主存页。一般说来，Linux系统采用位图和链表两种方法来管理主存页。

利用位图可以记录主存单元的使用情况。用一个二进制位(bit)记录一个主存页的使用情况：如果该主存页是空闲的，则对应的位是1；如果该主存页已经分配出去，则对应的位是0。例如有1024KB的主存，主存页的大小是4KB，则可以用32个字节构成的位图来记录这些主存的使用情况。

分配主存时，检测该位图中的各个位，找到所需个数的连续位值为1的位图位置，进而就获得所需的主存空间。

利用链表可以记录已分配的主存单元和空闲的主存单元。采用双向链表结构将主存单元链接起来，从而可以加速空闲主存的查找或链表的处理。

Linux系统的物理主存页分配采用链表和位图相结合的方法，如图5-46所示。在数组free_area的每一项描述某一种主存页组(即由相邻的空闲主存页构成的组)的使用状态信息。其中，头一个元素描述孤立出现的单个主存页的信息，第二个元素描述以两个连续主存页为一组的页组的信息，第三个元素描述以4个主存页为一组的页组的信息……依此类推，页组中主存页的数量依次按2的倍数递增。free_area数组的每项有两个成分：一个是双向链表list的指针，链表中的每个节点包含对应的空闲页组的起始主存页编号；另一个是指向map位图的指针，map中记录相应页组的分配情况。free_area数组的项0中包含一个空闲主存页；而项2中包含两个空闲主存页组(该链表中有两个节点)，每个页组包括4个连续的主存页，第一个页组的起始主存页编号是4，另一个页组的起始主存页编号是100。

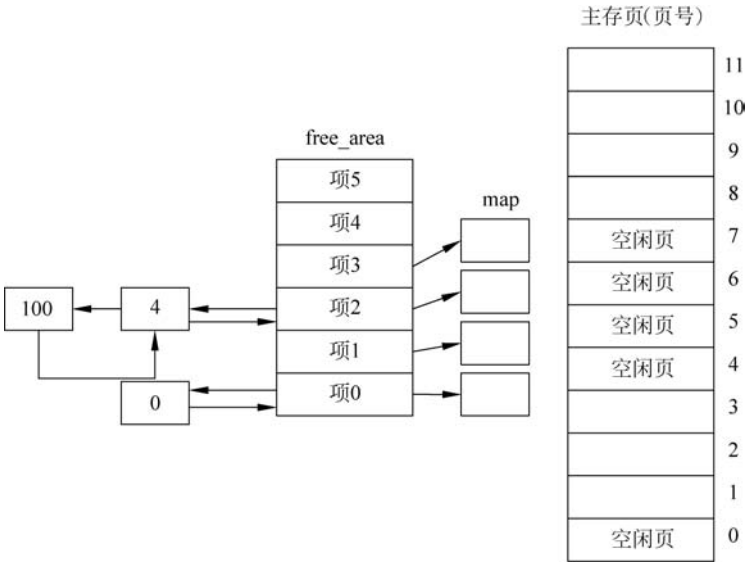


图5-46　空闲主存的组织示意图

在分配主存页组时，对于分配请求，如果系统有足够的空闲主存页，则Linux的页面分配程序首先在free_area数组中对于所要求数量的最小页组的信息进行搜索，然后在对应的list双向链表中查找空闲页组；如果没有与所需数量相同的空闲主存页组，则继续查找下一个空闲页组(其大小为上一个页组的2倍)。

如果找到的页组大于所要求的页数，则把该页组分为两部分：满足所请求的部分，把它返回给调用者；剩余的部分，按其大小插入到相应的空闲页组队列中。

当释放一个页面组时，页面释放程序就会检查其上下是否存在与它邻接的空闲页组。如果有的话，则把该释放的页组与所有邻接的空闲页组合，并成一个大的空闲页组，并修改有关的队列。上述主存页分配算法也称作“伙伴算法”。

当进程运行完毕后，将释放所占用的物理主存，系统将采用主存拼接的方法回收空间。当页面块被释放时，系统将检查是否有相同大小的相邻或者 Buddy 主存块存在。如果有，则将其合并为一个大小为原来两倍的新空闲块，之后系统继续检查和再合并，直到无法合并为止。

5.8.5 Linux 的页面交换机制

Linux 使用 LRU 算法作为其页面交换的核心算法，出于安全性、稳定性、执行效率等多方面的考虑，Linux 所使用的 LRU 交换算法已经交织在其进程管理、文件系统管理等其他机制当中，它们有机地结合成为一个整体。

1. Linux 描述页面的数据结构

在 Linux 当中，代表物理页面的 page 数据结构是在文件 include/linux/mm.h 中定义的。系统维持的物理页面供应量由两个全局量确定——freepages.high（初始化语句位于 mm/swap.c）和 inactive_target（初始化语句位于 include/linux/mm_inline.h），分别为空闲页面的数量以及不活跃页面的数量，二者之和为正常情况下潜在的供应量。

2. 页面淘汰策略

如果在发生缺页的时候才考虑页面交换，并把磁盘上待用的页面写入主存。这种完全消极的页面交换策略有一个缺点：换入换出操作总是在处理器忙碌的时候发生，这将使系统效率降低。Linux 的交换策略是定期地，特别是在系统相对空闲的时候，挑选一些页面预先交换出来，腾出一些主存空间，从而使系统始终维持一定的空闲页面供应量。一般以 LRU 为挑选准则。以上交换策略依然有可能发生系统“抖动”。为了防止“抖动”的发生，Linux 将页面的换出和主存页面的释放分成两步。当系统挑选出若干主存页面准备换出时，将这些页面的内容写入到相应的磁盘中，并且将相应的页面表项的内容设置为指向磁盘页面。

页面表项的数据结构 pte_t 的定义位于 include/asm-i386/page.h 中，它是一个 32 位的整型变量。其中它的第一位 P 用于表示该页面是否在主存中，P 位为 1 时表示在主存中；为 0 时则表示不在主存中。这里需要将换出页面的 P 位置 0，但是所占据的主存页面却并不立即释放，而是将其 page 结构留在一个 cache 队列当中，并使其从“活跃状态”转入“不活跃状态”，至于最后释放主存页面的操作就推迟到以后有条件时来进行。这样，如果在一个页面被换出以后立即又被访问而发生缺页异常时，就可以从物理页面的 cache 队列中找到相应的页面，并为之建立映射。由于该页面尚未释放，主存中还保留着原来的内容，就不需要再从磁盘上读入。反之，如果经过一段时间以后，一个不活跃的主存页面还是没有被访问，那就到了最后释放的时候了。如果位于 cache 队列中的页面又受到了访问，那么只需要恢复这个页面在主存中的映射，并使其脱离 cache 队列就可以了。

在 Linux 当中通过一个全局的 address_space 数据结构 swapper_space，把所有可交换的主存页面管理起来，每个可交换主存页面的 page 数据结构都通过其队列头结构 list 链入其中的一个队列。函数 add_to_swap_cache()将 page 结构链入相应队列，它的代码位于

mm/swap_state.c 中。将主存页面的“换出”与“释放”分两步走的策略显然可以减少抖动发生的可能,并减少系统在页面交换上的开销。

3. 修改过的页面和没有修改的页面

如前所述,引入“两步走”交换策略之后,大大提高系统效率,但是作为产品,Linux 的交换策略还可以进一步提升。

首先,在准备换出一个页面时,并不一定要把它的内容写入磁盘。如果自从最近一次换入该页面以后,就没有写过这个页面,那么这个页面的内容与磁盘上相应的页面内容是相同的,可以把这个主存页面看成“干净”的。这样的页面显然不需要写出去了。

其次,就是“脏”页面了。和“干净”页面相对应,“脏”页面就是在内容写出到磁盘上之后,主存页面发生改动的页面。这样的页面不需要立刻就写出去,而可以先将其从页面映射表中断开。经过一段时间,外存对应页面不再使用后,再写出去。这样“脏”页面就成了“干净”页面。“干净”页面可以在 cache 队列中等到真的有必要回收的时候再释放,回收一个“干净”页面的花费是很小的。

这些已经换出并且处于“释放”或者“半释放”的页面都处于不活跃状态,系统有个函数 inactive_shortage()用于统计主存中可供分配或周转的物理页面,它的代码位于 include/linux/mm_inline.h 中,其中 zone->pages_high+inactive_base 为系统正常情况下潜在的供应量,而 zone->inactive_clean_pages 和 zone->inactive_dirty_pages 分别对应上述处于不活跃状态的“干净”页面和“脏”页面。

内核当中设置了全局性的 active_list 和 inactive_dirty_list 两个 LRU 队列,还在每个页面管理区 ZONE 中设置了一个 inactive_clean_list。根据页面的 page 结构在这些 LRU 队列中的位置,就可以知道该页面转入不活跃状态后的时间长短,从而以此考虑是否应该回收。

5.9 本章小结

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,其目标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平衡。存储管理的基本功能有:主存空间的分配与去配、地址转换、主存空间的共享与保护、主存空间的扩充。

在现代计算机系统中,存储部件采用层次结构来组织,具体可分为寄存器、高速缓存、主存储器、磁盘缓存、磁盘、可移动存储介质等组成,这样在成本、速度和规模等诸多因素中能获得较好的性能价格比。

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,具体有静态重定位和动态重定位两种方式。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。固定分区管理采用顺序分配算法进行主存空间的分配,采用静态重定

位方式将作业装入主存，通过上、下限寄存器实现存储保护。可变分区管理的空间分配算法包括：最先适应、最优适应和最坏适应等算法。回收空间时，检查是否存在与回收区相邻的空闲分区，如果有，则将其合并成为一个新的空闲分区进行登记管理。可变分区管理一般采用动态重定位方式将作业装入主存，通过基址寄存器和限长寄存器等硬件实现存储保护。可变分区管理可以有条件地采用移动技术使分散的空闲区汇集起来容纳新的作业。分区管理中，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。分区管理方式要求作业信息一次性连续装入主存，对空间的要求较高。为了解决这个矛盾，操作系统引入离散存储管理方式。离散存储管理方式主要有页式存储管理、段式存储管理和段页式存储管理。离散存储管理允许将一个作业信息存储在不相邻的主存空间中，通过操作系统建立页表(或段表)数据结构实现逻辑地址空间与物理地址空间的映射，实现地址空间的保护。

虚拟存储器的实现借助于大容量的辅助存储器(如磁盘)存放作业信息，操作系统利用作业执行在时间上和空间上的局部性特点把当前需要使用的作业信息装入主存，并且利用页表、段表等数据结构构造一个用户的虚拟空间。操作系统根据中断(如缺页中断、缺段中断)进行处理，选择一种合适的调度算法对主存和辅存中的信息进行调入和调出，尽可能地避免“抖动”现象的发生。虚拟存储管理有请求分页式存储、请求分段式存储和请求段页式存储。请求式分页存储管理的实现需要必要的硬件支持和相应的软件支持，涉及请求分页的页表机制、缺页中断机构、地址变换机构、页面置换算法等。其中页面置换算法主要有：最佳置换算法、先进先出置换算法、最近最少用置换算法、clock 置换算法和最近最不常用置换算法等。虚拟存储器的性能与缺页中断率密切相关，系统分配给作业的主存物理块数、页面的大小以及程序的编制方法等对缺页中断率都有影响。

Linux 支持虚拟主存，本章最后从 Linux 的请求分页存储管理、多级页表、缺页中断、主存空间的分配与回收以及页面交换机制等方面做了介绍。


视频讲解

习 题 5

(1) 试述存储管理的基本功能。

(2) 什么是逻辑地址(空间)和物理地址(空间)？

(3) 什么是静态链接、装入时动态链接和运行时的动态链接？

(4) 什么是重定位？为什么要引入动态重定位？

(5) 试设计和描述最先适应算法的主存分配和回收过程。

(6) 在可变分区存储管理方式下，采用移动技术有什么优点？移动一道作业时，操作系统需要做哪些工作？

(7) 主存中的空闲区如图 5-47 所示，现有作业序列及其分别需要的空间依次为：Job1 要求 30KB，Job2 要求 70KB，Job3 要求 50KB。分别使用最先适应、最优适应和最坏适应算法处理这个作业序列，试问哪种算法可以满足分配要求？为什么？

(8) 设某系统中作业 J_1、J_2、J_3 占用主存的情况如图 5-48 所示。今有一个长度为 20KB 的作业 J_4 要装入主存，当采用可变分区分配方式时：

① 请列出 J_4 装入前的主存已分配表和未分配表的内容；

② 写出装入 J_4 时的工作流程，并说明所采用的分配算法。

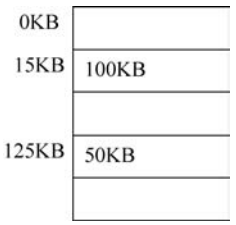


图 5-47 主存中的空闲区示意图

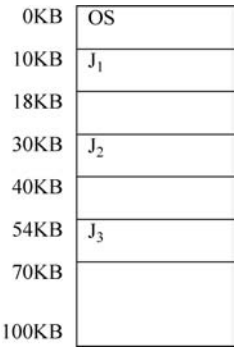


图 5-48 作业占用主存情况

(9) 给定主存空闲分区,按地址从小到大为 100KB、500KB、200KB、300KB 和 600KB。现有用户进程依次分别为 212KB、417KB、112KB 和 426KB:①分别采用最先适应、最优适应和最坏适应算法将它们装入到主存的哪个分区?②哪个算法能最有效地利用主存?

(10) 为什么要引入页式存储管理方法?在这种管理中,硬件提供了哪些支持?

(11) 在页式存储管理中为什么要设置页表和快表?

(12) 在页式存储管理中如何实现多个作业共享一个程序或数据?

(13) 在段式存储管理中如何实现共享?它与页式存储管理的共享有何不同?

(14) 分页式存储管理和分段式存储管理有何区别?

(15) 在具有快表的段页式存储管理系统中,如何实现地址变换?

(16) 在一个分页式存储管理系统中,某作业的页表如下。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。

页 号	块 号
0	2
1	3
2	1
3	6

(17) 给定段表如下:

段 号	段 首 址	段 长
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

给定地址为段号和位数,试求出对应的主存物理地址。

①[0,430];②[3,400];③[1,1];④[2,500];⑤[4,42]。

(18) 一个页式存储管理系统使用 FIFO、OPT 和 LRU 页面替换算法,如果一个作业的

页面走向为：

① 2、3、2、1、5、2、4、5、3、2、5、2；

② 4、3、2、1、4、3、5、4、3、2、1、5；

③ 1、2、3、4、1、2、5、1、2、3、4、5。

当分配给该作业的物理块数分别为3和4时，试计算访问过程中发生的缺页中断次数和缺页中断率。

(19) 在一个请求分页式系统中，假如一个作业共有5个页面，其页面调度次序为：1、4、3、1、2、5、1、4、2、1、4、5。若分配给该作业的主存块数为3，分别采用FIFO、LRU、clock页面置换算法，试计算访问过程中所发生的缺页中断次数和缺页中断率。

(20) 在一个分页虚存系统中，用户编程空间32个页，页长1KB，主存为16KB。如果用户程序有10页长，若已知虚页0、1、2、3，已分配到主存8、7、4、10物理块中，试把虚地址0AC5H和1AC5H转换成对应的物理地址。

(21) “FIFO算法有时比LRU算法的效果好。”这句话对吗？为什么？“LRU算法有时比OPT算法的效果好?”这句话对吗？为什么？

(22) 什么是抖动？产生抖动的原因是什么？

(23) 试设计和描述一个请求分页式存储管理的主存页面分配和回收算法。

(24) 什么是Belady现象？试找出一个产生Belady现象的例子。

(25) 试全面比较主存空间的连续分配方式和离散分配方式。

(26) 试述缺页中断与一般中断的主要区别。

(27) 在虚拟存储器管理中，淘汰页面时为什么要进行回写？通常采用什么方式来减少回写次数和回写量？

(28) 在请求式页式存储管理中，可采用工作集模型以决定分给进程的物理块数，有如下页面访问序列：

……2516337891623434344443443……

|----------| |----------|

Δt_1 Δt_2

窗口尺寸$\Delta=9$，试求t_1、t_2时刻的工作集。

(29) 一个程序要将100×100数组置初值0。现假设分配给该程序的主存块数有两块，页面的大小为每页100个字，数组中每一行元素存放在一页中。开始时，第一页已经调入主存。若采用LRU算法，则下列两种对数组的初始化程序段引起缺页中断次数各是多少？

①
```
for (j = 0;j < 100;j++)
    for (i = 0;i < 100;i++)
      A[i][j] = 0;
```

②
```
for (i = 0;i < 100;i++)
    for (j = 0;j < 100;j++)
      A[i][j] = 0;
```

(30) 试叙述虚拟存储器的基本原理。如何确定虚拟存储器的容量？

(31) 一个进程已分配得到4个物理块，每页的装入时间、最后访问时间、访问位R、修改位D如下表所示(所有数字为十进制，且从0开始)，当进程访问第4页时，产生缺页中

断。请分别用FIFO、LRU算法确定缺页中断服务程序选择换出的页面。

页　面	块　号	装入时间	最后访问时间	访问位R	修改位D
2	0	60	161	0	1
1	1	130	160	0	0
0	2	26	162	1	0
3	3	20	163	1	1

(32) 设某计算机的逻辑地址空间和物理地址空间均为64KB,按字节编址。若某进程最多需要6页数据存储空间,页的大小为1KB。操作系统采用固定分配局部置换策略为此进程分配4个物理块。

页　号	块　号	装入时刻	访　问　位
0	7	130	1
1	4	230	1
2	2	200	1
3	9	160	1

当该进程执行到时刻260时,要访问逻辑地址为17CAH的数据,请回答下列问题。

① 该逻辑地址对应的页号是多少?

② 若采用先进先出(FIFO)置换算法,该逻辑地址对应的物理地址是多少?请给出计算过程。

③ 若采用时钟(clock)置换算法,该逻辑地址对应的物理地址是多少?请给出计算过程。(假设搜索下一页的指针沿顺时针方向移动,且当前指向2号物理块,如图5-49所示。)

图5-49　采用时钟置换算法