第5章 存储器管理

本章学习目标

- 了解存储器管理的主要功能；
- 了解单一连接区存储方法；
- 掌握固定分区、可变分区的思想、实现关键；
- 熟练掌握分页存储管理的基本思想；
- 掌握位示图和页表数据结构及其作用；
- 熟练掌握静态分页的重定位过程；
- 系统理解虚拟存储器思想；
- 掌握请求分页的实现关键；
- 了解分页存储管理的特点；
- 理解分段和段页式存储管理的思想。

程序运行需要两个最基本的条件：一个是程序要占有足够的主存储空间；另一个是得到处理器，并且首先要得到足够的主存储器空间。在前面的章节中介绍了处理器的管理，本章将介绍操作系统对主存储器的管理，简称存储管理。

通常程序的代码和数据存储在磁盘等外存储器上，在多道程序设计环境下，如何为各道运行的程序分配合适的主存储空间？为了实现程序设计的共享、动态链接等新技术，存储管理要为程序员提供哪些功能？存储管理的主要目标是实现虚拟存储器，虚拟存储器的基本思想、理论基础和实现关键是什么？这些都是存储器管理要考虑的内容。

5.1 存储管理概述

本节先介绍存储管理相关的基本概念，然后提出存储管理的目的和主要功能。

5.1.1 计算机系统的存储器类型

存储器是存放程序和数据的部件，是计算机系统的主要资源，作为基础知识，下面介绍计算机系统中可以存放程序和数据的部件。存储器类型如图 5-1 所示。

1. 寄存器

处理器的寄存器(Register)可以存放程序的指令和运算数据。寄存器是所有存储部件

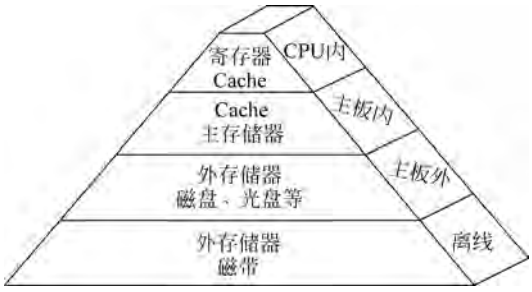

图 5-1 存储器类型

中存取速度最快的一种，但是，因处理器体系结构、工艺等的限制，寄存器数量非常有限，程序员能够使用的寄存器只有少数的几个或十几个，所以，寄存器只能用于存放当前正在执行的指令及其相关数据。

2. 高速缓冲区存储器

高速缓冲区存储器也称 Cache，是为了缓解 CPU 与主存储器之间速度不匹配而采取的技术，在 CPU 和主存储器之间加入一级或多级的静态随机存取存储器（SRAM）作为 Cache。在速度上 Cache 比主存储器快得多，现在可以把一部分 Cache 集成到 CPU 中，其工作速度接近于 CPU 的工作速度。

Cache 是如何缓解 CPU 与主存储器之间的速度不匹配呢？

这里简要描述 Cache 的工作原理：Cache 与主存储器之间的数据交换以块为单位，块的长度是固定的，一个块由若干字组成，而 Cache 与 CPU 之间的数据交换以字为单位。当 CPU 要读取主存储器中的一个字时，便发送该字的主存地址到 Cache，Cache 控制逻辑依据地址判断该字当前是否在 Cache 中，如果在 Cache 中则直接传给 CPU，否则，利用主存储器读周期把此字从主存读出送到 CPU，与此同时，把含有这个字的整个块从主存读出送到 Cache 中。管理 Cache 的控制逻辑都是硬件实现的，程序员看不到主存储器数据的 Cache 处理过程，这个特点称为 Cache 的透明性。

Cache 的容量要比主存储器小得多，一般只有几百千字节或几十兆字节，程序员不能在程序中直接使用 Cache。

3. 主存储器

主存储器也称内存储器（Main Memory），简称内存或主存，本章的存储管理就是管理主存储器。

计算机系统的主存储器是半导体介质的半导体存储器，用于存放处理器运行期间的程序和数据。CPU 可以直接访问主存储器。主存储器分为两种：只能进行读操作的只读存储器（ROM）和可以进行读或写操作的随机存取存储器（RAM）。随机存储器的数据不具有可保存性，即关机后其中的数据消失。

主存储器由存储单元组成，所有存储单元的集合称为存储空间。有两种基本的存储单元：存放一个机器字的存储单元称为字存储单元；存放一个字节的存储单元称为字节存储单元。因为一个机器字通常包含几个字节，所以字存储单元由几个连续的字节存储单元组

成。存储单元的编号称为地址,或内存地址,字存储单元对应的是字地址,字节存储单元对应的是字节地址。程序员在程序(汇编语言)中可以直接访问主存储器,而按字地址还是字节地址访问主存,可以由程序员的程序设计决定。

通常把若干地址连续的存储单元称为存储区域,简称存储区。存储区中存储单元的个数称为存储区的长度,或存储区的大小。

一个计算机系统中主存储器的存储单元总数称为存储容量,通常存储容量是以字节存储单元来计算的。

4. 外存储器

外存储器也称辅助存储器,简称外存或辅存。外存储器是计算机系统存放数据的最主要部件,人们输入到计算机系统中的数据、可执行程序文件等,都是保存在外存储器上的。

计算机系统的外存储器是磁性介质的磁表面存储器,典型的有磁盘、磁带等。外存储器的特点是存储容量大、价格低、数据可以长期保存,但是由于 CPU 不能直接访问,需要通过 I/O 操作实现外存储器中数据的存取,所以,与内存相比,外存储器的存取速度慢。

外存储器所能存放的字节总数称为存储容量,为了方便管理,外存储器在使用之前被分成若干块,存取的基本单位是块(物理块)。

人们把光盘、U 盘等存储设备也看作外存储器。

关于外存储器,将在第 6 章中介绍文件存储空间的管理方法,在第 7 章中介绍磁盘的 I/O 请求的驱动调度。

5.1.2 虚拟地址和物理地址

虚拟地址(Virtual Address)和物理地址(Physical Address)是存储管理的两个基本概念,下面介绍它们的含义。

1. 虚拟地址和虚拟地址空间

程序源代码经过编译系统的编译、链接后,生成可执行目标程序,在可执行目标程序中,源程序中的每一个变量和指令,在目标程序中都对应一个唯一的编号,这些编号称为变量或指令语句的虚拟地址。例如,图 5-2 中,源程序中变量 name 和 score 对应的虚拟地址是 10 和 30,源程序中语句 if 对应的虚拟地址为 200。

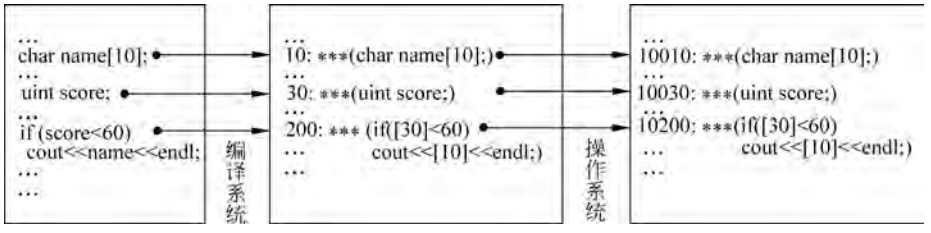

图 5-2 虚拟地址和物理地址的例子

每一道程序的虚拟地址都是从 0 开始,依次连续地进行编号。目前,可执行程序的虚拟地址有两种表示方法:一维虚拟地址和二维虚拟地址。有的存储管理方法使用一维虚拟地

址，即将程序中所有变量和语句统一从 0 开始连续地编号；有的存储管理方法则使用二维虚拟地址，二维虚拟地址由段号和段内地址组成，每一个段的段内地址也是从 0 开始，连续地进行编号。

同一道程序的所有虚拟地址的集合称为该程序的虚拟地址空间。因为程序代码是有限的，所以虚拟地址空间是有限集合，集合的元素个数称为虚拟地址空间的大小，也称程序的大小。

作业虚拟地址空间是指作业中各程序虚拟地址空间的并集，作业大小是指作业中各程序虚拟地址空间大小的总和。

在存储器管理中，侧重于存储空间的管理，所以，有时为了叙述上的方便，对作业、程序和进程不做严格区别，可以将它们看作等同的概念。

2. 物理地址和物理地址空间

可执行程序平时以文件形式存储在磁盘等外存储器上，由于 CPU 不能直接访问磁盘，因此，程序运行时由操作系统把它从磁盘上读出并装入内存，这个过程称为程序的装入。一道程序在装入内存后，程序的每一个虚拟地址所对应的变量或指令在内存中都对应唯一的一个存储单元，这个存储单元的地址称为对应变量或指令的物理地址。如图 5-2 所示，源程序中变量 name 和 score 对应的物理地址是 10010 和 10030，源程序中语句 if 对应的物理地址为 10200。

一道程序的所有变量和指令的物理地址的集合称为该程序的物理地址空间，也称进程地址空间。

5.1.3 重定位

重定位(Relocation)也称地址转换或地址映射，就是把虚拟地址转换为物理地址的过程。因为 CPU 最终是以物理地址存、取数据和指令，所以程序的运行必然需要重定位。

1. 程序装入

在批处理系统中，作业调度程序选中一个作业后，系统将从外存输入井的后备队列中读取作业的程序，将其装入内存的合适存储位置。类似地，在分时系统中，用户提交命令后，如果是外部命令，则系统需要从外存中读取命令对应的程序装入内存；在实时系统中，系统响应一个新事件时，该事件对应的处理程序也要装入内存。

操作系统将程序从外存读出并装入内存的过程称为程序装入(Programming Loading)。程序装入后，系统创建程序对应的进程。

2. 重定位

根据什么时候进行地址转换，重定位分为两种方式：静态重定位(Static Relocations)和动态重定位(Dynamic Relocations)。

1) 静态重定位

如果在程序装入时把所有的虚拟地址全部一次性地转换为物理地址，在创建相应的进程后，进程运行过程不再需要地址转换，则这种重定位方式称为静态重定位。

如图 5-3 所示，静态重定位方式中虚拟地址 200 的指令在装入物理地址 10200 存储单

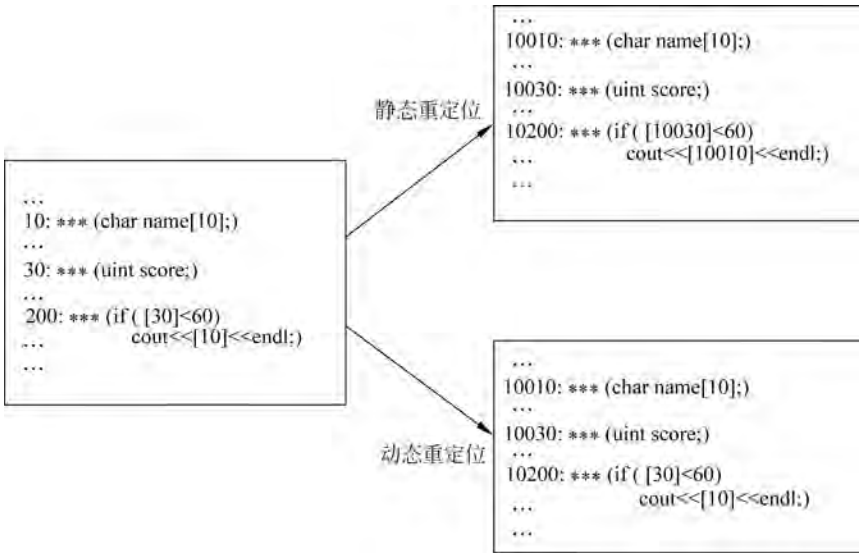

图 5-3 两种重定位方式的例子

元后,该指令中原来的虚拟地址 30 转换为 10030,原来的虚拟地址 10 转换为 10010。

静态重定位方式实现简单,不需要硬件的支持,但是,由于程序装入后虚拟地址全部都转换为物理地址了,不便于存储保护,而且改变程序在内存存放位置的工作复杂。另外,静态重定位往往要求按照虚拟地址顺序存储,同一道程序连续占用依次相邻的存储单元,影响存储空间的利用率。

单任务的 DOS 操作系统就是采用静态重定位方式。

2) 动态重定位

支持多任务的操作系统都是采用动态重定位方式,即程序装入时没有进行地址转换,而是在运行过程中,对将要访问的指令或数据的虚拟地址转换为物理地址,也就是 CPU 需要访问时才转换。

如图 5-3 所示,动态重定位方式中物理地址 10200 处的指令中原来的虚拟地址 30 和 10 在程序装入后被保留下来。

动态重定位方式增加了系统存储管理的灵活性。例如,同一道程序虚拟地址在内存中的存放位置可以不连续,允许程序装入后再改变其存放的位置,可以支持动态链接、虚拟存储器等技术。

为了加快地址转换过程,动态重定位都是硬件实现。CPU 中实现动态重定位的控制逻辑称为存储管理单元(Memory Management Unit,MMU)。MMU 不仅能够实现重定位,还能够实现存储保护等的存储管理功能。

5.1.4 存储管理的主要功能

在计算机系统启动过程中,操作系统内核程序装入内存的固定区域,这些供内核使用的存储单元统称系统区,或系统空间(System Space),其余的存储单元统称用户区,或用户空间(User Space)。

系统区已经分配给操作系统使用，所以操作系统只需对其进行保护，禁止用户程序直接访问即可，这样，主存储器管理主要是对用户区存储单元的管理。

存储管理的目的是提高主存储器的利用率、方便用户对主存储空间的使用。

存储管理实现的主要功能如下：

1. 存储空间的分配和回收

在程序装入时，根据程序的虚拟地址空间的大小，找出一些合适的存储单元，用于存放装入程序的指令和数据，这个过程称为存储空间的分配(Allocate Memory)，在进程运行结束后，进程撤销原语把进程所占用的存储单元归还或释放，即存储空间的回收。

存储空间分配回收的主要任务如下：

1) 设计合理的数据结构，登记存储单元的使用情况

存储单元的状态有两种：分配和空闲。某存储单元分配给一个进程后，其状态为分配状态，未分配或者分配后已经回收的存储单元，其状态为空闲状态，空闲的存储单元可以用来分配，供进程使用。

因此，为了存储空间的分配和回收，需要设计合理的数据结构用于登记存储单元的状态及其位置信息等。

2) 设计分配算法

在多道程序设计环境下，内存中可以同时存放多道程序，这样，在一道程序装入后，还需要装入其他程序。由于主存储器容量相对较小，在存储空间分配时，需要根据一定的策略，在数据结构登记的信息中查找、选择合适的存储单元，以便能够装入尽可能多的程序，提高存储器的利用率。

在多道程序设计中，同时装入内存的程序虽然不是越多越好，但是在很多情况下，为提高并行程度，在内存中的程序要有一定的数量。所以，存储管理需要设计或选择合适的算法，允许装入尽可能多的程序。

3) 存储空间回收

一个存储单元分配给一个进程后，其状态为分配状态，分配状态的存储单元不能再分配给其他进程使用，因此，一个进程结束后，存储管理还需要将进程占用的存储空间回收，设置为空闲状态，修改或调整数据结构中登记的相关信息。回收的存储单元可以再分配给进程使用。

存储空间的分配和回收是存储管理最基本的功能。

2. 重定位

需要决定是采用静态重定位，还是动态重定位。如果采用动态重定位，应结合 CPU 的存储管理单元，登记进程所占用存储单元的相关信息，供 MMU 进行地址转换。

3. 存储空间的共享与保护

存储空间的共享是指多个进程可以访问同一个存储区，避免将同一组数据或程序为每个进程分配一个独立的存储区造成的存储空间浪费。所以，需要在保证进程独立性的同时，提供共享方法，让多个进程共享同一个存储区域。

存储空间保护的基本要求是：各进程只能访问其占用的存储单元的指令或数据，不能访问已经分配给其他进程的存储单元或未分配的空闲存储单元，即进程地址空间是私有的。这样可以保证进程的独立性，避免进程之间的相互干扰和破坏，简化操作系统的进程管理。

存储空间的保护是在重定位过程中对虚拟地址和物理地址的检查，基本的存储保护方法有以下几种：

1）界限寄存器法

MMU 提供两个寄存器，称为基址寄存器（Base Register）和界限寄存器（Limit Register），分别存放当前运行进程可访问存储区的起始存储单元地址和连续存储单元个数（或存储区的结束单元的地址）。在程序运行过程中，检查当前访问的存储单元的地址是否在界限寄存器规定的范围，如果满足规定，则允许访问，否则，产生一个地址越界中断，进程终止。

界限寄存器法可以限制一个进程只能访问一个存储区域内信息，是最基本的存储保护方法。

2）保护键法

将存储空间分成一些存储区，每个存储区设置可以访问操作的键值（Key），所谓键值，是指操作代码，如 01 表示允许读操作，10 表示允许写操作，11 表示既可以读操作又可以写操作，00 表示禁止访问，等等。CPU 的程序状态字 PSW 设置相应的位，表示当前可访问的存储区的键值，这样，只有在 PSW 中的键值、当前进程拥有的键值，以及当前访问的操作匹配时，才能执行相应的访问操作，否则，为非法访问，进程终止。

共享存储区通常采用保护键法。

3）界限寄存器和 CPU 工作模式

当 CPU 工作模式为核心态时，可以访问所有的存储单元，当 CPU 工作模式为用户态时，只能访问界限寄存器规定范围的存储单元，这样，把 CPU 工作模式和界限寄存器法结合起来进行存储保护。

存储空间的共享增加了存储保护的复杂性。

另外，现代的处理器通过引入特权级，在硬件上进行存储保护。存储保护是安全操作系统研究的主要内容之一。

4. 虚拟存储器

随着计算机的广泛应用，要求计算机处理的问题越来越复杂，程序员设计的程序往往超过内存的实际大小，如果仍然按照程序先装入内存才能运行的传统方法，对于实现复杂功能的程序，因为超过内存的实际大小不能装入内存，程序就无法运行。在早期的 DOS 操作系统中，程序员开发应用系统时，经常遭受这样的困扰，一旦程序超过内存的实际大小，程序员就需要修改程序。这导致程序员在设计和开发过程中，一方面，要致力于应用需求的功能实现，努力地进行数据结构和算法的设计；另一方面，还要时时担心所编写的程序会不会超过可用内存的大小。

虽然，计算机技术在不断蓬勃发展，主存储器容量也得到大幅度提高，但限于硬件体系结构和工艺的限制，主存储器容量比磁盘容量仍然小得多。“内存容量有多大，程序就会有多大”，因此，主存储器的实际容量总是不能满足多道程序设计的要求。

操作系统解决这个矛盾的方法就是采用虚拟存储器技术。

什么是虚拟存储器？就是没有增加主存储器容量的实际大小，而是在软件上采取一些方法，程序装入时，只装入一部分，其余部分仍然保留在外存中；在程序运行过程中，系统把程序未装入的部分从外存调入内存，或者把内存中的程序部分信息调出；这样，使得系统可以运行比主存储器容量大的程序，或者在内存中装入尽可能多的程序。对程序员来说，好像系统拥有一个充分大的“主存储器”，这个“主存储器”就是虚拟存储器。

实现虚拟存储器不仅可以减轻程序员在编程过程中因内存大小限制而引起的顾虑，而且可以提高主存储器利用率。

5.1.5 存储管理方法

针对提出的存储管理功能，本章介绍的基本存储管理方法如下：

1. 分区管理

分区存储管理是一类存储单元连续分配的管理方法。一个进程占用一个存储区域，一个存储区域也只分配给一个进程，相比之下，分区存储管理是比较简单的存储管理方法。分区存储管理又有 3 种实现方法：单一连续区、固定分区和可变分区。

另外，分区存储管理可以利用对换或覆盖等存储技术实现内存的逻辑扩充，提高存储空间的利用率。

2. 分页管理

分页存储管理是一种非连续的存储空间管理，提高了存储空间的利用率，为实现虚拟存储器建立了基础。现代操作系统的存储管理大多数采用分页存储管理。分页存储管理分为静态分页、动态分页两种，其中，动态分页是一种虚拟存储器技术。

分页存储管理具有内存空间的分配、回收操作简单，能够支持虚拟存储器等优点，但是，分页存储管理把一个进程的数据、指令代码、堆栈等统一在同一个虚拟地址空间，不便于程序信息的保护。

3. 分段管理

在分段管理中，一个进程的数据、指令代码、堆栈等信息拥有各自独立的虚拟地址空间，因而保持了程序的完整结构，为实现动态链接、存储共享、保护等程序技术提供了方便。

在硬件支持下，分段管理可以实现虚拟存储器。分段管理的内存分配采用可变分区，内存空间的分配、回收操作比较复杂。

4. 段页式管理

把分页管理和分段管理的思想和方法结合起来，就形成了段页式存储管理。段页式存储管理综合了分页、分段存储管理的优点；但也增加了数据结构的存储开销和重定位过程的处理器开销。

本章后续各节将分别介绍这些存储管理方法的基本思想、实现关键和特点。

5.2 单一连续区存储管理

单一连续区存储管理是最简单的一种存储管理方法。DOS 操作系统就是采用这种方法管理系统的常规内存。

1. 基本思想

单一连续区存储管理的基本思想是：操作系统启动后占用系统区，整个用户区一次只能装入一道用户程序，只有在用户区中的程序运行完成后，才能装入下一道程序。

如图 5-4 所示，假定系统启动后，操作系统内核占用一个长度为 32K 的系统区，用户区长度为 256K。

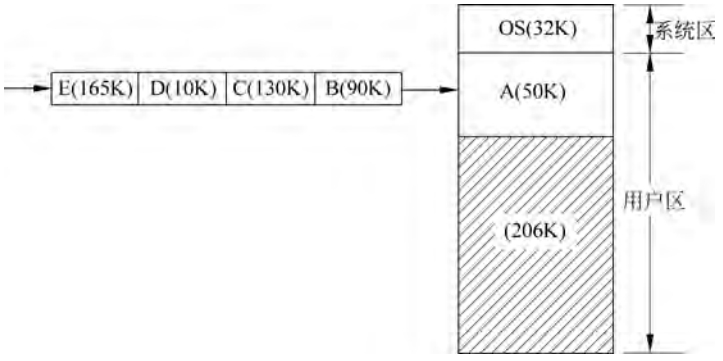

图 5-4 单一连续区存储管理

为了叙述方便，在存储区长度没有指明单位时，不考虑存储单元是字存储单元还是字节存储单元，但是以同一种存储单元计算，这里 $1K=2^{10}$。

假定有一组程序 A、B、C、D 和 E 要求运行，它们的虚拟地址空间的大小分配是 50K、90K、130K、10K 和 165K。那么，系统首先装入程序 A 并运行，程序 A 装入后，用户区还剩余 206K 的存储空间，这时，系统不管用户区剩余多少空闲的空间，都不能分配给其他程序使用。只有在程序 A 运行结束，对应进程撤销后，才能装入下一道程序 B。之后依次地装入运行 C、D 和 E 等程序。

单一连续区存储管理采用静态重定位，可以使用界限寄存器法实现存储保护。

2. 主要特点

单一连续区存储管理的主要特点是：是最简单的管理方法，不需要复杂硬件的支持；但是，只能用在单任务的操作系统中，因此存储空间的利用率低。

5.3 固定分区存储管理

在单一连续区存储管理中，整个用户区一次只分配给一道程序，分配后不管剩余多少空闲空间，其他程序都不能使用，因此存储空间的利用率低。如果事先能把这个用户区分为一

些更小的用户区，对于每个小用户区按单一连续区存储管理，一次分配一道程序，这样，内存可以同时装入多道程序，支持多道程序设计，并且提高存储空间的利用率，这就形成了固定分区存储管理方法。

5.3.1 基本思想

固定分区存储管理的基本思想是：操作系统启动时，根据事先的设置，把用户区分成若干存储区域，每个区域称为一个分区(Partition)，各个分区的长度可以不相等；启动成功后，分区的个数和每个分区的长度不再改变；程序装入时，一个分区只能分配一道程序，一道程序也只能占用一个分区，这种分配方式也称连续分配。

如图 5-5(a)所示，用户区被分成 4 个分区，长度分别是 75K、30K、140K 和 11K。各分区按地址顺序编号，4 个分区的区号分别为 1、2、3 和 4，0 号分区为系统区，供内核使用。

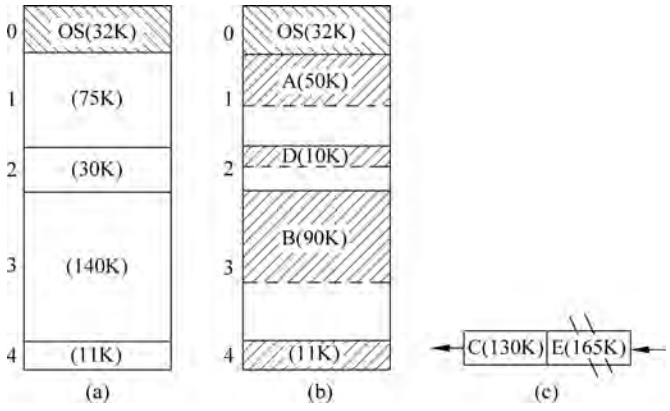

图 5-5 固定分区

5.3.2 实现关键

固定分区存储管理实现的关键技术如下：

1. 数据结构设计

固定分区采用的数据结构称为分区说明表(Descriptive Partitions Table，DPT)，其表结构由分区号、起始地址、分区长度和状态组成，其中状态表示对应的分区是分配还是空闲，用 1 表示分配，0 表示空闲。

分区说明表是在系统启动时建立并初始化，因为分区的个数和各分区的长度是事先配置的，所以，建立和初始化分区说明表比较容易。图 5-5(a)的分区结果如表 5-1 所示，描述了对应的分区说明表。

表 5-1 分区说明表结构及初始化

区 号	长 度	起始地址	状 态
1	75K	32K	0
2	30K	107K	0
3	140K	137K	0
4	11K	277K	0

2. 分配和回收

程序装入时,根据程序的虚拟地址空间大小,依次查找分区说明表,查找的条件是分区长度大于或等于当前程序的虚拟地址空间大小,且状态是0。图5-6描述了固定分区的分配流程。

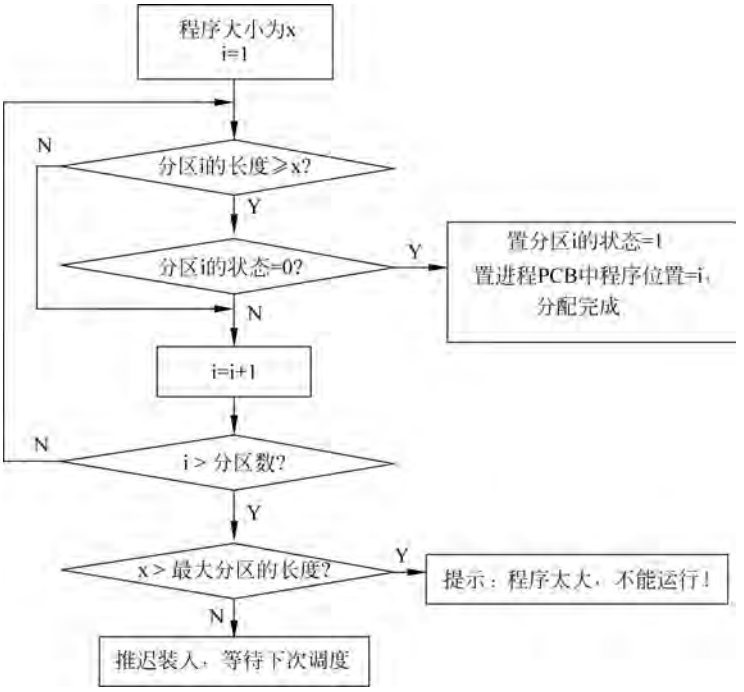

图5-6 固定分区的分配流程

如图5-5所示,假定有一组程序A、B、C、D和E要求运行,它们的虚拟地址空间的大小分配是50K、90K、130K、10K和165K。分配结果是:程序A分配在分区1,程序B分配在分区3,程序C当前不能装入,但它没有超过最大分区的长度,所以可以等待下次调度时再装入;程序D则分配在分区2,而程序E当前不能装入,但它超过最大分区的长度,因而分配结果提示系统不能运行程序E。

进程运行结束时,回收分区的过程是:从进程PCB中程序位置,得到进程占用分区的分区号,在分区说明表中将对应分区的状态置为0。

3. 重定位和存储保护

固定分区是一种简单的存储管理方式,因此重定位只需采用简单的静态重定位方式。

存储保护使用限界寄存器法。当进程调度程序选择一个进程运行时,基址寄存器存放当前进程所占用的分区地址,界限寄存器存放分区最后一个存储单元的地址。

5.3.3 主要特点

固定分区存储管理具有如下主要特点:

1. 能够支持多道程序设计

在固定分区中，将用户区分成多个分区，每个分区存放一道程序，因此，可以同时存放多道程序，为实现多道程序设计提供了可能。固定分区存储管理方法是目前能够支持多道程序设计存储管理方法中最简单的一种。

2. 并发执行的进程数受分区个数的限制

每次一个分区只分配给一道程序，内存中同时存放的程序数不能超过分区的个数，因此并发执行的进程数受分区个数的限制，只得借助对换技术来解决这种限制。

3. 程序大小受分区长度的限制

一道程序只能占用一个分区，所以程序的虚拟地址空间大小不能超过分区的大小。特别地，一道程序不能超过最大分区的长度，否则将无法在这次开机中运行，只能在关机后下次开机时修改分区配置，设置一个较大的分区，重新启动后才能执行。

在没有采用其他技术的情况下，如果一道程序超过了整个用户区长度，它就不能运行。

4. 存在“碎片”

在固定分区存储管理中，存在小程序占用大分区造成的“碎片”(Fragmentation)现象。所谓碎片，是指暂时不能使用的存储区域。例如在图 5-5 中，程序 B 的大小为 90K，而它占用了长度为 140K 的分区 3，由于一个分区只能分配给一道程序，造成此时分区 3 剩余的 50K 不能分配给其他程序，剩余的 50K 暂时不能使用，成为碎片。这类碎片也称内碎片(Internal Fragmentation)，即进程内部多余的存储区域，因为整个分区已经分配给了进程，只是进程运行过程中不需要这些存储单元而已。

为了减少这种碎片现象，一种方法是：作业调度时选择最大程序先装入运行，但这种方法可能会与短作业优先思想冲突，尽管程序的虚拟地址空间大小与它占用 CPU 时间的大小没有直接联系，但多数情况下，两者还是有一些联系。

另一种方法是：建立多个输入队列(Input Queue)，一个输入队列对应一个分区，这个输入队列中的程序大小尽可能接近于对应的分区长度，将来输入队列中的程序装入时，只能装入与输入队列对应的分区中，如图 5-7 所示。这种方法虽然可以避免小作业占用大分区造成的存储空间浪费，但是，却可能存在另一种状况，就是一个较大长度的分区对应的输入队列为空(没有要求运行的程序)，而一些长度较小分区的输入队列中又有大量要求运行的程序，导致大的分区因为对应的输入队列没有要求运行的程序，长时间处于空闲状态，而小分区的输入队列中又有许多等待装入运行的程序，从而影响存储空间的利用率。

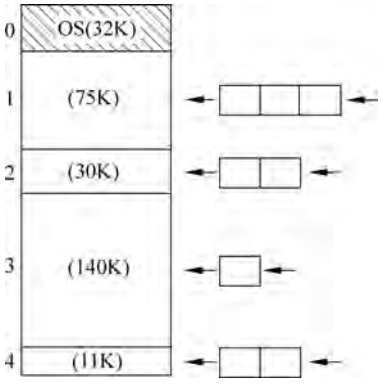

图 5-7 多输入队列的固定分区

5.4 可变分区存储管理

固定分区事先把一个大的用户区分成几个小的分区,造成程序装入时的不方便。如果事先不进行分区,而是在程序装入时,根据程序的实际需求量动态地建立分区,这样可以避免固定分区中存在的一些问题。

5.4.1 基本思想

可变分区也称动态分区,其基本思想是:操作系统启动成功后,整个用户区作为一个空闲区,一般地,在程序装入时,系统根据程序的实际需求量,查找一个合适的空闲区,如果该空闲区长度等于程序的实际需求量,就可以直接分配,否则,将其分成两个分区,其中一个分区长度等于当前程序的需求量,并分配给程序,另一个分区作为空闲区保留下来。进程运行完成被撤销后,回收其占用的分区,成为空闲区。

系统在程序装入时,如果不存在合适的空闲区,则进一步检查程序是否超过用户区的总长度,如果程序大小超过用户区的总长度,则程序不能在系统中运行。

图 5-8(b)是系统启动后的用户区,假定长度为 256K。现有一组程序 A、B、C、D 和 E 要求装入运行,它们的大小分别是 50K、90K、130K、10K 和 165K,如图 5-8(a)所示。在程序 A 装入时,将用户区分成一个长度为 50K 的分区并分配给程序 A,其余 206K 作为空闲区,如图 5-8(c)所示;接着,程序 B 装入时,将 206K 的空闲区分成一个 90K 的分区,分配给程序 B,剩余的 116K 作为空闲区,如图 5-8(d)所示;在程序 C 装入内存时,由于没有合适的空闲区,程序 C 暂时不能装入;系统继续选择下一个程序 D,在程序 D 装入时,将 116K 的空闲区分成一个长度为 10K 的分区,并分配给 D,剩余的 106K 作为空闲区,如图 5-8(e)所示。假定之后不久,内存中的程序 B 运行完成,则程序 B 占用的分区,被回收成为一个空闲区,如图 5-8(f)所示,接着内存中的程序 A 也运行完成,系统在回收其占用的分区后,与相邻的

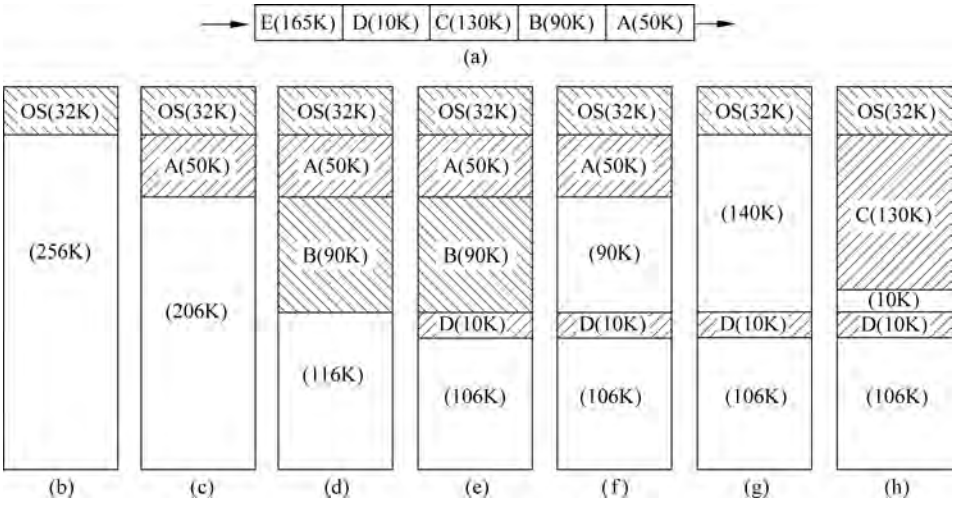

图 5-8 可变分区的例子

90K 空闲区合并，成为一个 140K 的空闲区，如图 5-8(g)所示，这时，140K 的空闲区用于分配程序 C，得到如图 5-8(h)所示的结果。将来，当内存中的程序 C 和 D 全部结束后，内存又恢复到初始时的状态如图 5-8(b)所示，之前未能装入而在等待的程序 E 也可以装入内存，得到运行。

可见，在可变分区存储管理中，随着内存的分配、回收操作的不断进行，内存中分区个数不断变化，每个分区的长度也在变化，并可能呈现出空闲区和分配区交替出现的复杂内存布局。

5.4.2 实现关键

下面分别介绍可变分区的数据结构设计、分配策略、回收处理以及重定位和存储保护。

1. 数据结构设计

在可变分区存储管理中，把被进程占用的分区称为分配区，这样，就存在两种类型的分区：空闲区和分配区。分配区可以登记在对应进程的 PCB 中，因此只需考虑空闲区管理的数据结构。

1) 可用表

首先想到的是，借鉴固定分区管理中的分区说明表 DPT，设计一个称为可用表的数据结构。那么，在可用表的结构设计时，能否直接使用分区说明表的结构呢？

可用表的结构为分区的起始地址、长度和状态。

这里“状态”的含义与分区说明表中的“状态”含义不同。因为可用表只管理空闲区，所以不需要像分区说明表中的“状态”用于描述分区是分配的还是空闲的。但是，由于空闲区的个数是变化的，而表的数据结构又要求数据连续存储，所以必须为可用表事先预留一定的长度。这就导致表中的每一行不一定就真正描述一个可用的空闲区，因为某一行可能是预留的，并未用于登记一个真实的空闲区信息。例如，在系统启动后，在初始状态时可用表只有第一行描述了一个空闲区，其余各行都是预留的。

所以，可用表中的“状态”表示对应的行是否描述一个分区，比如，用 0 表示预留，1 表示空闲区。

表 5-2 是图 5-8(h)的空闲区管理的可用表信息，假定可用表的长度为 5。

表 5-2 可用表及例子

起 始 地 址	长 度	状 态
162K	10K	1
182K	106K	1
		0
		0
		0

从结构上看，可用表与分区说明表结构相似，但是，两者在实现时有较大区别。分区说明表由于分区个数是固定的，在建立分区说明表时，容易按分区个数确定表的长度。可用表在初始状态时只有一个空闲区，但是随着系统的运行，分区个数是变化的，所以在建立可用

表时要预留一定的表长度。如果预留太多,会增加系统的存储开销;如果预留太少,在系统的运行过程中空闲区的个数不能超过表的长度,分区个数受到限制。

可用表用于登记系统中各空闲区的信息。由于表的长度是有限的,所以并发进程数也受到限制。原因是,当空间区个数等于可用表的长度时,如果这时有一个进程运行完成,系统回收其占用的分区,当这个新回收的空闲区与可用表中空闲区不能合并时,这个新的空闲区无法登记在可用表中。在这种情况下,就必须采用移动技术来解决,移动技术也称存储紧凑(Memory Compaction)技术。

移动技术的目标是:把几个空闲区汇集起来,合并成一个较大的空闲区。其做法是:在内存的分配区和空闲区交替出现的情况下,通过改变(移动)内存中若干程序的存放位置,让这些程序占用的分区集中在依次相邻的位置,原来交替出现的空闲区也就汇集在一起,形成一个较大的空闲区。

移动技术可以把几个小的空闲区合并成一个较大的空闲区,不仅减少了空闲区的个数,还因为汇集了一个更大的空闲区给程序装入带来很大方便。

但是,移动技术也面临一些问题:首先是系统的开销增加,因为需要把进程的程序代码移动到另一个区域;其次,并不是随时都可以移动一个进程,即移动是有条件的。因为正在进行 I/O 操作的进程、通信中的进程,都不能简单地改变它们的存放位置。原因是这些进程的地址空间、其他进程或系统进程可能正在使用中,如果轻易改变这些进程的地址空间,而与其相关的进程或系统进程仍然按原来的地址进行数据操作,必然造成操作错误或数据错误。另外,移动技术延迟了进程的运行,因为移动操作需要处理器时间,且正被移动的进程不能运行。

2) 空闲区链表

在可变分区中,空闲区的个数是动态变化的,因此,可以选用链表的数据结构来管理空闲区。链表非常适合于这种个数变化的状况,因为链表是一种可以非连续存储的数据结构,结点的增加和删除操作容易实现。

在空闲区链表中,一个结点表示一个空闲区,其结构设计如下:

```
struct FreeNode {
    long  start;                    //分区的起始地址
    long  length;                   //分区的长度
    struct FreeNode * next;         //向下指针
} * freePartitionsList;             //空闲区链表头指针
```

链表中结点的顺序可以根据需要,按分区的起始地址(start)从小到大排列,或者按分区的长度(length)从小到大或从大到小排列。图 5-9(a)所示的链表是对图 5-8(h)的空闲区的表示。

空闲区链表在实现时,结点结构可以简化,如图 5-9(b)所示。具体做法是:空闲区链表的首指针指向第一个空闲区的起始地址,每个空闲区中的第一个存储单元存放当前空闲区的长度,第二个存储单元存放下一个空闲区的起始地址,如果是最后一个空闲区则第二个存储单元存放空值表示链表结束。这种方法的好处是:利用空闲区登记链表结点,节省了存储空间。

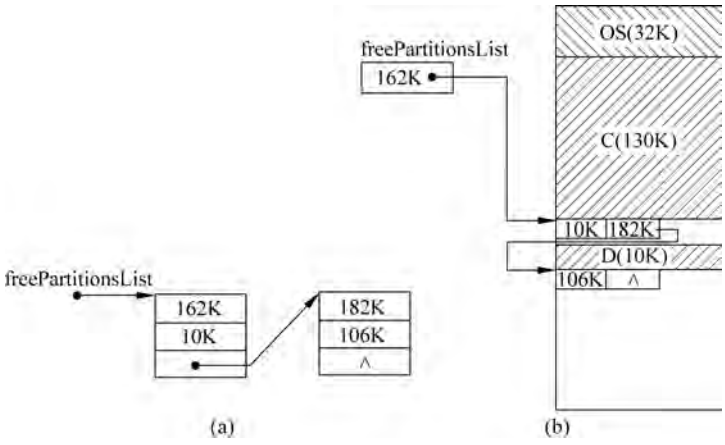

图 5-9　空闲区链表的例子

3）请求表

请求表用于管理要求装入内存的程序信息，其结构包括程序名(作业名)和虚拟地址空间大小，如表 5-3 所示。

当进程撤销时，因为系统回收了新的空闲区，所以这时需要检查请求表是否有程序可以装入内存。

表 5-3　请求表

程序名	虚拟地址空间大小
E	165K

2．分配策略

在可变分区管理中，存储空间的分配过程是：在程序装入时，查找一个合适的空闲区。如果找到的空闲区长度刚好等于程序的实际需求量，则直接分配，否则，将空闲区分成两个分区，其中一个分区正好等于程序的实际需求量，并分配给它，另一个分区作为更小的空闲区保留下来。

这里“合适”有两个含义：首先，空闲区长度必须满足当前程序的实际需求量，即空闲区长度大于或等于程序对存储空间的实际需求量；其次，所找的空闲区应该满足指定的分配策略。

存储空间分配的基本策略有最先适应法(First Fit，FF)、最佳适应法(Best Fit，BF)和最坏适应法(Worst Fit，WF)。

1）最先适应法

最先适应法的基本思想是：从用户区的低地址开始查找，找到能满足程序需求的第一个空闲区用于分配。

最先适应法的分配特点是：小程序集中运行在低地址部分的存储空间上，大程序装入较高地址部分的存储空间上。

2）最佳适应法

最佳适应法的基本思想是：查找一个能满足程序需求的、长度最小的空闲区用于分配。

最佳适应法的分配特点是：尽量把较大的空闲区保留下来，以便满足后续大程序的需求，但它将小的空闲区分割成两个分区，使得分配后剩余的空闲区更小，这些小的空闲区不利于将来的程序装入操作，容易造成碎片。

3) 最坏适应法

最坏适应法的基本思想是：查找一个能满足程序需求的、长度最大的空闲区用于分配。

最坏适应法的分配特点是：查找大的空闲区用于分配，使得分配后剩余的空闲区尽可能大，有利于其他程序的装入。

例如，某系统当前内存的初始状态如图 5-10(a)所示。有 3 个空闲区，长度分别是 100K、140K 和 40K(假定 OS 占用的分区为低地址空间)，这时有 4 个要求装入运行的程序 A、B、C 和 D，它们的虚拟地址空间大小分别是 80K、30K、130K 和 25K。那么，分别采用 FF、BF、WF 策略，这 4 道程序的装入情况怎样？

在没有采用移动技术的情况下，图 5-10 中的(b)、(c)和(d)分别表示 FF、BF、WF 策略内存分配的结果。其中，FF 策略分配时，程序 C 暂时不能装入；BF 策略分配时，程序 D 暂时不能装入；WF 策略分配时，程序 C 暂时不能装入。

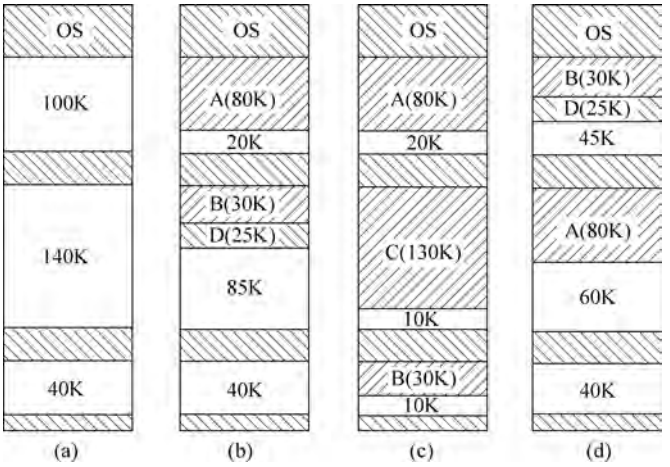

图 5-10　3 种基本分配策略的例子

系统在实现这 3 种分配策略时，可以通过空闲区数据结构的不同组织方式进行统一。以空闲区链表为例，对于 FF 策略，链表的结点按分区 start 值从小到大排列；对于 BF 策略，则将链表的结点按分区 length 值从小到大排列；对于 WF 策略，链表的结点按分区 length 值从大到小排列。这样，只要从链表的首指针开始查找第一个能满足长度要求的空闲区即可。

最后以空闲区链表为例，介绍程序装入时的分配流程，如图 5-11 所示。

3. 回收处理

在一个进程被撤销时，系统要回收其所占用的分区，分区信息从进程的 PCB 中的程序位置提取。在回收一个分区时，需要判断是否有相邻的空闲区，以便把相邻的空闲区合并为一个大的分区，有利于程序装入时新的分配。

当回收一个空闲区时，它与相邻的分区的关系有如图 5-12 所示的 4 类情况。

图 5-12 中的(a)类有 3 种情况：一是回收的新空闲区是用户区中最低地址的存储区域，且它的下邻分区是分配区；二是回收的新空闲区是用户区中最高地址的存储区域，且它的上邻空闲区是分配区；三是回收的新空闲区的上邻分区和下邻分区都是分配区。这 3 种情

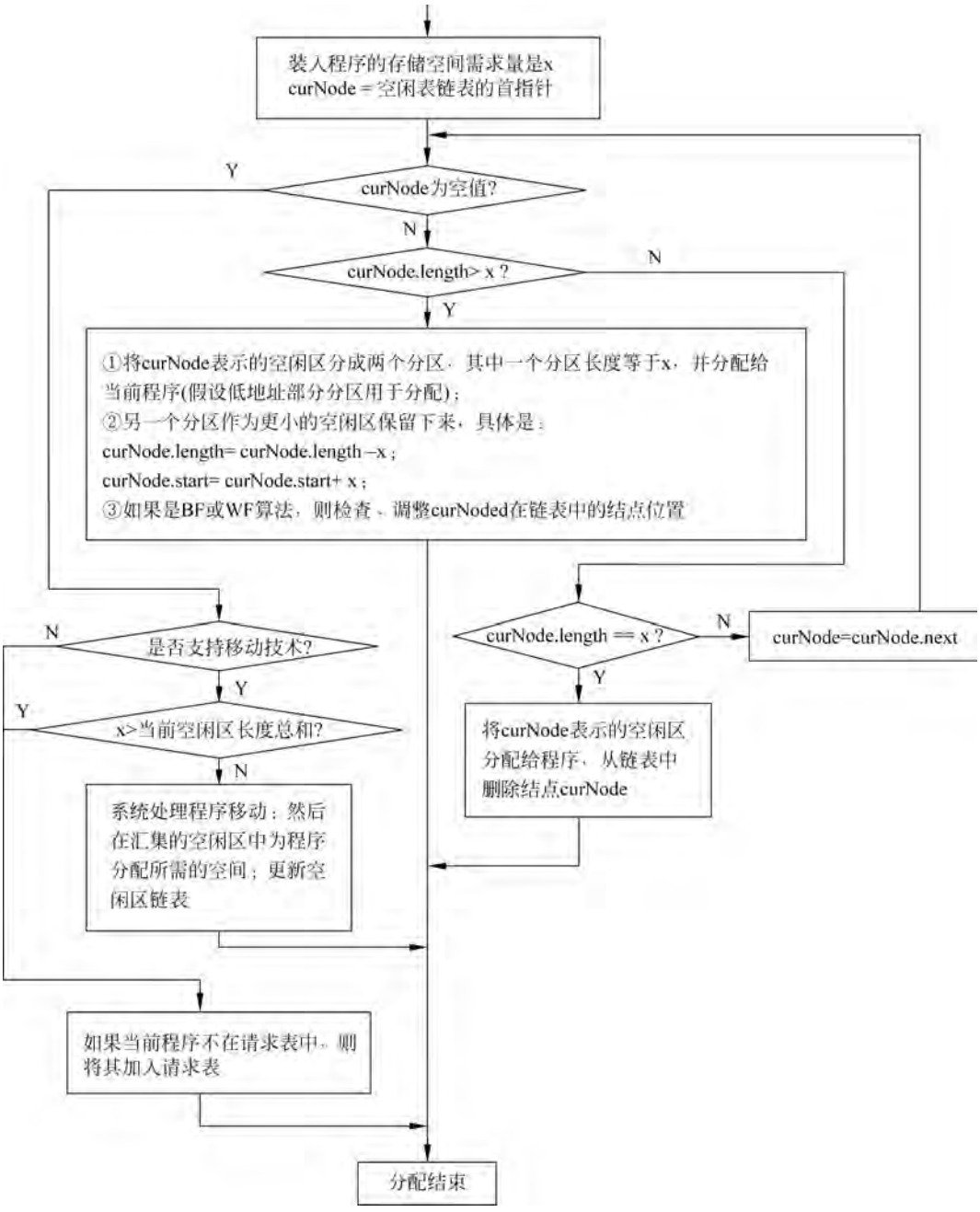

图 5-11　可变分区分配流程

况都不能合并。

图 5-12 中的(b)类有两种情况：一是回收的新空闲区是用户区中最高地址的存储区域，且它的上邻分区是空闲区；二是回收的新空闲区的上邻分区是空闲区，但下邻分区是分配区。这两种情况中，新空闲区可以与它的上邻空闲区合并成一个空闲区。

图 5-12 中的(c)类也有两种情况：一是回收的新空闲区是用户区中最低地址的存储区

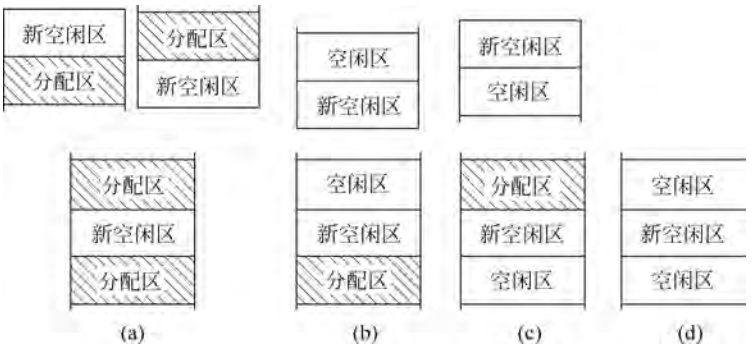

图 5-12 空闲区回收时的 4 类情况

域,且它的下邻分区是空闲区;二是回收的新空闲区的上邻分区是分配区,但下邻分区是空闲区。这两种情况中,新空闲区可以与它的下邻空闲区合并成一个空闲区。

图 5-12 中的(d)类只有一种情况,即回收的新空闲区的上邻分区和下邻分区都是空闲区,这时新空闲区可以与它的上、下邻空闲区一起合并成一个空闲区。

下面给出图 5-12(b)和(c)两类的合并的判断及其修改方法。

假设 F1 是空闲区链表中的一个结点,r 是回收的空闲区的新结点。如果满足

$$F1.start + F1.length == r.start$$

那么,F1 是 r 的上邻空闲区,合并时,只需修改 F1 结点如下即可:

$$F1.length = F1.length + r.length$$

如果满足

$$F1.start == r.start + r.length$$

那么,F1 是 r 的下邻空闲区,合并时,修改 F1 结点如下即可:

$$F1.start = r.start;\ F1.length = F1.length + r.length$$

综上分析,在回收一个分区时,系统中空闲区的个数变化情况是:满足图 5-12(a)类型时,空闲区个数增加一个;满足图 5-12(b)或(c)类型时空闲区个数不变;满足图 5-12(d)类型时,空闲区的个数减少一个。

4. 重定位和存储保护

因为可变分区可能需要采用程序的移动技术,因此,在可变分区存储管理中,地址转换采用动态重定位。

在 CPU 中,设置一个专门的机构实现动态重定位,即 MMU。MMU 的基址寄存器存放当前进程占用分区的起始地址;界限寄存器存放当前进程占用分区的长度。这两个寄存器属于 CPU 的现场信息,在进程调度时进行恢复和保护。另外,MMU 还有一个寄存器存放 CPU 当前将要访问的数据或指令的虚拟地址,称为虚拟地址寄存器(VR)。MMU 在重定位时,首先判断虚拟地址寄存器的值是否小于界限寄存器的值,如果虚拟地址寄存器的值小于界限寄存器的值,则将虚拟地址寄存器(VR)的值加上基址寄存器(BR)的值,得到对应的物理地址,供 CPU 访问;否则,MMU 产生一个地址越界中断,当前进程结束。

因此,可变分区存储管理采用界限寄存器法实现存储保护。

5.4.3 主要特点

在可变分区存储管理中，进程占用的分区长度等于进程的内存实际需求量，从表面上看，存储空间的利用率很高，实际上，随着内存空间的分配和回收的不断进行，可能存在长度很小的空闲区，在没有采用移动技术的情况下，这些小的空闲区暂时也不能使用，从而影响了存储空间的利用率。

如图 5-10 所示的例子中，3 种分配策略都存在这样的情况。例如，图 5-10(b)的结果中，程序 C 的大小为 130K 不能装入，这时 3 个空闲区长度分别是 20K、85K 和 40K，虽然它们的合计长度可以满足程序 C 的要求，但其中任一个空闲区都不能满足程序 C 的要求，对于程序 C 而言，这 3 个空闲区暂时都无法使用。同样，图 5-10(c)的结果中也有 3 个空闲区，长度分别是 20K、10K 和 10K，它们的合计长度可以满足程序 D 的 25K 存储空间的要求，但是单个空闲区的长度都不能满足程序 D，造成程序 D 无法装入；在图 5-10(d)的结果中有 3 个空闲区，长度分别 45K、60K 和 40K，它们的合计长度能够满足程序 C 的 130K 的需求，但每个空闲区都不满足程序 C。可见，可变分区存储管理的内存利用率也不高。

1. 存在碎片，降低了存储空间的利用率

可变分区中暂时不能使用的空闲区称为碎片。与固定分区存储管理中的碎片不同，这里的碎片称为外碎片(External Fragmentation)。虽然采用移动技术可以利用这些碎片，但移动是有条件的，且移动会导致系统的开销增加。

2. 分区个数和每个分区的长度都在变化

在可变分区管理中，在一次程序装入过程的存储空间分配后，空闲区的个数不一定就减少一个，只有在分配时找到的空闲区长度刚好等于当前程序的实际大小时分配后空闲区个数才减少一个，否则，分配后空闲区个数保持不变。

进程运行结束，在系统撤销进程回收其占用的分区后，空闲区的个数不一定就增加一个。因为回收一个空闲区时，由于可能与其相邻的空闲区合并，所以，回收后，根据图 5-12 的不同情况，空闲区的个数有时增加一个，有时不变，有时反而减少一个。

3. 为进程的动态扩充存储空间提供可能

进程在运行过程中，由于任务的需要，可能要动态地申请新的存储单元，用于存放新的数据，这就需要为进程动态扩充存储空间，可变分区存储管理为进程的这种存储扩充提供了可能。

4. 需要相邻空闲区的合并，增加系统的开销

在可变分区管理中，需要把一些依次相邻的空闲区合并成一个大的分区，以便新程序装入时的分配。通常是在进程撤销回收其占用的分区时进行合并处理，但是也可以在程序装入发现没有合适空闲区时进行合并处理。

对于 FF 策略，空闲区的合并比较简单。因为 FF 策略在组织数据结构的空闲区信息时，按分区的起始地址顺序排列，即按空闲区在内存位置中的自然顺序排列，这样，只要判断

数据结构中相邻的空闲区是否可以合并即可。另外,合并后的修改也比较简单,因为把几个相邻的空闲区信息合并成一个空闲区信息后,不影响数据结构中的其他空闲区的数据顺序。

对于 BF 策略和 WF 策略,空闲区的合并处理比较复杂。因为在这两种策略中,数据结构中的空闲区信息按分区的长度排序,破坏了空闲区的自然位置,在判断是否为相邻空闲区时,需要逐个比较,并且,合并后可能需要检查、调整数据结构的数据顺序。

5. 分配策略 FF、BF 和 WF 在存储空间利用率上没有很大差别

分配策略 FF、BF 和 WF 在分配上各有特点,但是在存储空间利用率上没有很大差别,它们都可能存在外碎片。

可见,在可变分区存储管理中,存储空间的分配比较复杂。

5.4.4 分区管理总结

单一连续区、固定分区、可变分区统称分区管理,它们具有如下特点:

1. 存储空间连续分配,管理方法容易实现

存储空间的连续分配是指,在程序装入时,同一道程序的信息按虚拟地址的顺序,依次存放在内存中连续的存储单元。连续分配为系统的重定位和存储保护提供了方便。

2. 存在碎片,存储空间利用率不高

存储空间的连续分配方法,在程序装入时要求有一个足够长度的空闲区,这样,在单一连续区和固定分区存储管理中,存在小程序占用大分区的内碎片;在可变分区存储管理中,因各空闲区长度太小无法分配,造成外碎片。

内碎片和外碎片的区别是:从存储单元的状态看,内碎片是分配状态,内碎片作为分区的一部分分配给进程,只是进程不需要而已,因为是分配状态,所以其他进程不能使用;外碎片是空闲状态,只是由于分区长度太小,如果没有这样的小程序,虽然是空闲状态,但暂时也无法用于分配。

从长度看,内碎片的长度可能很大,外碎片的长度往往比较小。

3. 程序大小受分区的限制

在单一连续区和可变分区存储管理中,程序的大小不能超过用户区的总长度,在固定分区存储管理中,程序大小不能超过最大分区的长度。

5.4.5 对换和覆盖

在分区管理中,程序大小受到分区长度的限制,这给程序员开发复杂的应用系统带来不便。为了减少这方面的限制,可以采用对换或覆盖技术。

1. 对换技术

对换(Swaping)技术的思想是:在外存储器(磁盘)中设置一个专门的区域,称为交换

区，系统需要时，选择内存中就绪状态或阻塞状态的一个或几个进程，将其信息写入交换区的合适位置，这一过程也称调出；调出后空出来的存储区可用于装入新的进程，或者从交换区中读入之前调出的一个进程信息，即调入，或者供当前运行的进程扩充存储空间。

对换技术是由操作系统实现的，程序员或用户看不到进程的调出/调入过程。

采用对换技术后，进程状态分为运行、活动就绪、活动阻塞、静止就绪和静止阻塞 5 个状态，如图 5-13 所示。内存中的就绪进程属于活动就绪状态，外存中的就绪进程属于静止就绪状态，内存中的阻塞进程属于活动阻塞状态，外存中的阻塞进程属于静止阻塞状态。静止阻塞状态的进程在唤醒后转换为静止就绪状态。

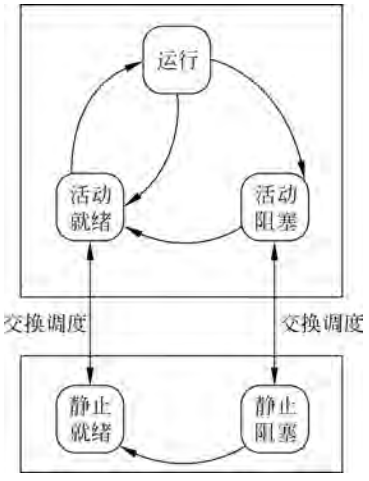

图 5-13 具有对换技术的进程状态

对换技术可以增加并发执行的进程数，或者使得当前运行的进程拥有更多的可用存储空间。在固定分区存储管理中，采用对换技术可以解决并发进程数受分区个数限制的问题。

在多道程序环境单处理器系统中，内存同时有多个进程，但是，在一个较小的时间范围内，真正运行的进程只有一个，其他进程或者是就绪状态或者是阻塞状态。系统把暂时不在运行中的进程按一定策略，选择一个或几个将其信息全部从内存中调出到外存的交换区，这样可以让当前运行的进程拥有更多的可用存储空间；如果把内存中一部分就绪状态的进程调出，还可以减少就绪队列中的进程数，从而减少进程调度程序执行的时间；这样，还增加了进程调度的灵活性，可以有选择地让若干进程保留在内存中，轮流地占用 CPU 运行，而另一部分进程调出到交换区，暂时不参与竞争处理器，在合适的时候再从交换区中选择一部分进程调入内存。

对换技术可以增加并发执行的进程数，是一种内存逻辑扩充技术。

对换技术在 UNIX 系统中得到了很好的应用，许多现代操作系统也仍保留这一技术。

2. 覆盖技术

覆盖技术(Overlay)是早期操作系统 DOS 中采用的一种内存逻辑扩充技术。

覆盖技术的思想是：操作系统提供关于内存空间的分配、撤销和设置(Setblock)等存储管理的系统调用，以及程序装入或程序装入并运行等的进程管理的系统调用(也称 EXEC 功能)。另外，程序员在设计程序时，根据程序的功能进行可覆盖结构设计，把一些彼此间没有调用关系的子程序(模块)作为一个组，同一组的子程序称为可覆盖段，这些可覆盖段的每个子程序可以独立编写在一个可执行的程序文件中。

可覆盖程序的调用方式如下：

(1) 建立可覆盖区。

程序员根据各子程序的存储空间需求量，利用操作系统存储管理的系统调用，申请分配存储区域，即可覆盖区。可覆盖区的长度不得小于同一组的可覆盖段中最大子程序的实际需求量。

在 DOS 中的 int 21H 的 48H 功能可以实现进程存储分配，其用法如下：

入口参数：AH＝48H，BX＝申请的可覆盖区大小。需要注意，存储区大小以节(Paragraphs)(16B为一节)为单位计算。

返回结果：标志寄存器的进位C表示分配结果，如果进位标志被置位(为1)，表示分配不成功，AX返回错误代码，其中AX＝7表示存储器控制块已破坏，AX＝8表示可用存储空间不足；如果进位C为0，表示成功，AX返回可覆盖区的段地址，即可覆盖区为AX：0开始的连续BX×16 B存储单元。

(2) 利用进程管理的EXEC功能装入可覆盖子程序的程序文件。

在建立可覆盖区后，程序员在需要时，利用进程管理的EXEC功能装入指定的可覆盖子程序的程序文件，同一组可覆盖段的各个子程序文件装入同一个可覆盖区中，新装入的子程序文件覆盖了原来的子程序。

DOS中提供的进程管理的EXEC功能有两种方式：

一种是装入并运行。由int 21H的AH＝4BH、AL＝0功能实现。这种方式实现了两道程序的运行，但它们没有并发执行，在新调用的程序运行完成后，CPU再执行主程序中装入系统调用指令的下一条指令，并且要求程序员在装入运行之前保存主程序的堆栈段寄存器SS和堆栈指针寄存器SP。

另一种是装入覆盖代码，供其他程序调用，由int21H的AH＝4BH、AL＝3功能实现。

(3) 调用子程序。

内存中的主程序或其他子程序可以调用覆盖区中装入的子程序。由于可覆盖段的子程序文件也是一个完整的程序，所以必须以跨段的方式实现调用。装入后可以多次调用，直到被新的子程序文件覆盖为止。

(4) 撤销可覆盖区。

建立的可覆盖区不再需要时，程序员通常要及时利用存储管理提供的系统调用，撤销可覆盖区。

在DOS中，系统int 21H的49H功能实现可覆盖区的撤销，入口参数BX是要撤销的可覆盖区的段地址。

如图5-14所示，假定程序员给出的一道程序结构设计中，A、B和C子程序之间没有相互调用关系，E、F和G子程序之间也没有相互调用，M和N也是彼此独立的子程序。这样得到3组可覆盖段。程序员在主程序main中向系统申请分配3个可覆盖区，分别对应3组可覆盖段，可覆盖区长度依次为50K、70K和150K。所以，在程序员的良好设计下，借助操作系统提供的系统调用，程序在内存中只需占用30K＋50K＋70K＋150K＝300K的存储空间，而这道程序及各子程序的合计大小为

30K＋20K＋50K＋40K＋50K＋70K＋50K＋150K＋120K＝580K

也就是说，300K的存储空间可以运行580K的程序，覆盖技术可以实现内存的逻辑扩充。

这里给出两个在DOS下覆盖技术应用的例子。

DOS内存管理采用单一连续区方式，程序装入时，整个用户区分配所装入的程序，但在程序运行过程中，真正使用的只有一部分，其余部分不需要，即成为内碎片，所以，在申请可覆盖区之前，程序员要先把内碎片归还，使用int 21H的4AH功能(SetBlock)，设置当前程序占用区域，并归还内碎片。

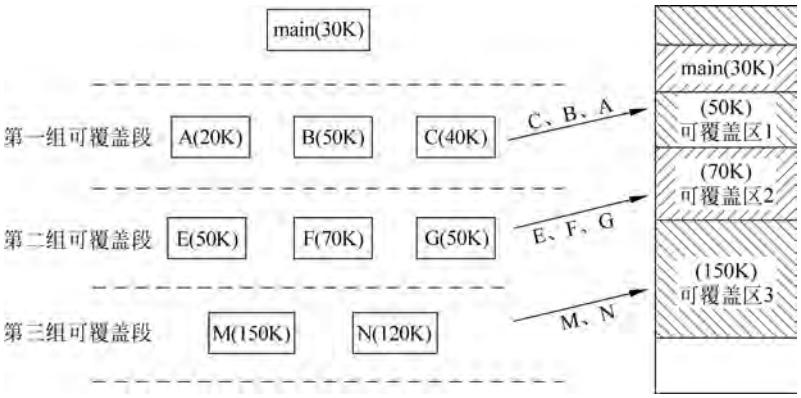

图 5-14 可覆盖结构设计例子

例 5-1 [DOS 覆盖技术应用——装入运行]下面的程序代码实现,将装入另一道程序 d:\masm5\MASM.EXE 并运行。

```
dseg segment
    filename   db   'd:\masm5\MASM.EXE',0          ;要装入运行在程序文件
    parameters  dw   7 dup(0)                       ;运行程序的命令行参数
    keep_ss   dw 0                                  ;保存主程序的堆栈段寄存器
    keep_sp   dw 0                                  ;保存主程序的堆栈指针
dseg ends
stack segment stack
    db 0100h DUP('?')
stack ends
cseg segment
    assume cs:cseg
start:                ;主程序入口
    mov bx,zseg       ; -------- 把内存中主程序未使用的存储空闲归还 ------------
    mov ax,es         ;DOS 采用单一连续区管理,程序装入时,整个用户区归其所有,
    sub bx,ax         ;利用伪段 zseg 计算主程序占用空间的长度,
    mov ah,4ah        ;DOS 的 SetBlock 功能,改变当前存储区的大小
    int 21h           ; ----------------------------------------------------
    mov ax,seg parameters
    mov es,ax
    mov bx,offset parameters
    mov keep_ss,ss              ;装入之前保存主程序的堆栈段地址
    mov keep_sp,sp              ;保存主程序的堆栈指针
    mov dx,offset filename      ;要装入运行的程序文件
    mov ax,seg filename
    mov ds,ax
    mov ah,4bh                  ;DOS 的 EXEC 装入功能
    mov al,0                    ;设置装入并运行的功能
    int 21h
    mov ax,dseg                 ;子程序运行结束
    mov ds,ax
    mov ss,keep_ss              ;恢复主程序的堆栈段地址
    mov sp,keep_sp              ;恢复主程序的堆栈指针
```

```
    lea dx,filename              ;主程序结束之前提示装入运行文件名
    mov ah,09h
    int 21h
    int 20h                      ;返回
cseg ends
zseg segment                     ;伪段 zseg,设置在主程序末尾,用于计算主程序大小
zseg ends
end start
```

例 5-2 [DOS 覆盖技术应用——装入覆盖、调用运行]下面的主程序将 3 个可覆盖子程序分别装入并调用执行,主程序代码如下:

```
dseg segment
    load_proc_error_msg  db  'load sub_proc error',0dh,0ah,'$',0
    setblk_error_msg  db  'set block error',0dh,0ah,'$',0
    alloc_error_msg  db  'allocation error',0dh,0ah,'$',0
    success_msg  db  'end',0dh,0ah,'$',0
    overlay_seg   dw  ?              ;保存覆盖区段地址
    overlay_offset   dw  ?           ;保存覆盖区偏移量(相对主程序的代码段首地址)
    code_seg  dw  ?                  ;保存主程序代码段
    block  dd  0                     ;参数块,保存可覆盖区段地址和重定位因子
    proc_file_path  db  'd:\masm5\PROC.EXE',0        ;第一个可覆盖子程序文件
    proc_file_path1  db  'd:\masm5\PROC1.EXE',0      ;第二个可覆盖子程序文件
    proc_file_path2  db  'd:\masm5\PROC2.EXE',0      ;第三个可覆盖子程序文件
dseg ends
stack segment stack
    db 0100h DUP('?')
stack ends
cseg segment PARA PUBLIC 'CODE'
    assume cs:cseg
ret_exit proc
    mov ax,dseg
    mov ds,ax
    mov ah,09h
    int 21h
    mov ah,7
    int 21h
    int 20h
    ret
ret_exit endp
allocation_error:
    lea dx,alloc_error_msg
    call ret_exit
setblk_error:
    lea dx,setblk_error_msg
    mov ah,09h
    call ret_exit
load_error:
    mov cx,ax
    add cl,30h
    mov ax,dseg
    mov ds,ax
    lea dx, load_proc_error_msg
```

```
    mov bx,dx;
    mov [bx],cl
    call ret_exit
start:                                  ;主程序入口
    mov code_seg,cs
    mov bx,zseg
    mov ax,es
    sub bx,ax                           ;计算主程序占用空间大小
        mov ah,4ah                      ;SetBlock 功能
        int 21h                         ;调整主程序占用空间,归还未使用部分
    jc setblk_error
    mov bx,100h                         ;申请一个 100H×16 字节的可覆盖区
    mov ah,48h                          ;存储器分配功能
    int 21h                             ;可覆盖区地址 AX:0
    jc allocation_error
    mov overlay_seg,ax                  ;保存可覆盖区段地址
    mov ax,code_seg                     ;保存主存程序代码段,子程序返回时自动恢复
    mov bx,overlay_seg
    sub bx,ax                           ;代码段与可覆盖段的差值计算偏移量,以节为单位
    mov cl,4                            ;转换为字节为单位
    shl bx,cl
    mov overlay_offset,bx               ;保存可覆盖区偏移量
    mov ax,seg block
    mov es,ax
    mov bx,offset block
    mov ax,overlay_seg                  ;设置参数
    mov [bx],ax                         ;可覆盖区段地址
    mov [bx]+2,ax                       ;重定位因子,子程序的首地址
    mov ax,dseg
    mov ds,ax
    mov dx,offset proc_file_path        ;装入第一个子程序文件
    mov ah,4bh                          ;EXEC 的装入功能
    mov al,3                            ;装入覆盖原有代码
    int 21h
    jc load_error
    mov ax,dseg
    mov ds,ax
    mov bx,offset overlay_offset
    call DWORD PTR [bx]                 ;跨段调用可覆盖区的子程序,运行子程序
    mov ah,7                            ;等待用户按键后,继续执行
    int 21h
    mov ax,seg block
    mov es,ax
    mov bx,offset block
    mov dx,offset proc_file_path1       ;装入第二个子程序文件
    mov ah,4bh                          ;EXEC 的装入功能
    mov al,3                            ;装入覆盖原有代码
    int 21h
    mov ax,dseg
    mov ds,ax
    mov bx,offset overlay_offset
```

```
    call DWORD PTR [bx]                ;跨段调用可覆盖区子程序运行
    mov ah,7                           ;等待用户按键后继续运行
    int 21h
    mov ax,seg block
    mov es,ax
    mov bx,offset block
    mov dx,offset proc_file_path2      ;装入第三个子程序文件
    mov ah,4bh                         ;EXEC 的装入功能
    mov al,3                           ;装入覆盖原有代码
    int 21h                            ;覆盖原来子程序
    mov ax,dseg
    mov ds,ax
    mov bx,offset overlay_offset
    call DWORD PTR [bx]                ;跨段调用可覆盖区子程序运行
    mov ax,dseg
    mov ds,ax
    mov ax,overlay_seg                 ;取可覆盖区段地址
    mov es,ax
    mov ah,49h                         ;归还分配的可覆盖区
    int 21h
    lea dx,success_msg
    mov ah,09h
    int 21h
    mov ah,7
    int 21h
    int 20h
cseg ends
zseg segment                           ;伪段 zseg,设置在主程序末尾,用于计算主程序大小
zseg ends
end start
```

第一个可覆盖子程序,运行时在屏幕上显示内容为变量 proc_msg 的字符串。源代码如下。

```
dseg segment
    proc_msg  db  '111,$ ',0
dseg ends
cseg segment  para public 'code'
    assume cs:cseg
overlay proc far
    push ds
    mov ax,dseg
    mov ds,ax
    lea dx,proc_msg
    mov ah,09h
    int 21h
    pop ds
    ret
overlay endp
cseg ends
end
```

为简单起见，第二、三两个可覆盖子程序与第一个子程序一样，只是把其中变量 proc_msg 的字符串值修改为'222'和'333'。

从上述两个例子可以看出，覆盖技术主要是由程序员通过程序设计实现程序文件的调入运行或覆盖，从而大大增加了程序员编程的工作量。虽然程序员增加了不少的负担，但是，早期在 DOS 操作系统没有提供虚拟存储技术情况下，程序员在复杂应用系统的开发中，覆盖技术也是解决程序超过内存空间情况下的一种方法。

3. 对换和覆盖的区别

对换技术和覆盖技术可以提高内存空间的利用率，两者的区别如下：

1）控制不同

对换技术中，进程的调入/调出是由操作系统自动实现的，而覆盖技术则是由程序员根据需要通过程序控制调入覆盖和调用运行。

2）单位粒度不同

在对换技术中，调入/调出发生在进程之间，即整个进程调出或调入；而覆盖技术则是发生在主程序的进程内部，对应子程序之间的调入覆盖。

3）内存扩充的效果不同

在对换技术中，单个进程的地址空间不能超过内存的实际大小，而覆盖技术允许程序的大小超过内存的大小(但需要程序员的精心设计)。

5.5 分页存储管理

在分区存储管理方法中，内存空间的连续分配造成了碎片，降低了存储空间的利用率。在分页存储管理方法中，在程序装入时突破连续分配的限制可以减少碎片的产生，同时为实现虚拟存储器提供了方便。本节介绍分页存储管理的方法。

5.5.1 基本思想

在现代操作系统中，普遍采用分页(Paging)存储管理方法，分页存储管理的基本思想如下：

1. 内存分块

把内存空间看成由一系列长度相等的存储区域组成，每个存储区域称为一个块，也称物理块、内存块或帧(Frame)，这个存储区的长度称为块长。每个块都有一个唯一的块号，从低地址开始，每个块的块号依次为 0、1、2……如图 5-15(a)所示。

那么，块长是如何确定的呢？分页存储管理需要硬件的支持，块长是由硬件和操作系统软件决定的。硬件上 CPU 的结构不同，块长可能有差别，块长是用 2 的 k 次幂表示的整数，其中 k 由 CPU 的虚拟地址结构确定。

一般地，在分页存储管理中，所有块的长度相等，这给存储管理的实现带来了方便。例如，如果已知一个块的块号，那么，这个块的存储区域就可以确定，因为块的起始地址等于

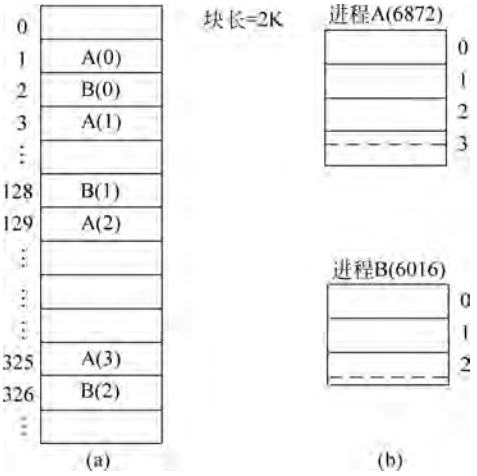

图 5-15 分页的基本思想

块长度×块号

而块长是固定的。

2. 进程分页

在程序装入时,按照块的长度,系统把程序的虚拟地址空间分为一些大小相等的页(Page),每个页对应一个页号。系统按虚拟地址的顺序依次为每一个页进行编号,页号为0、1、2……

分页存储管理由操作系统和硬件共同实现。在 CPU 中,与存储管理有关的控制逻辑称为存储管理单元,即 MMU,MMU 实现重定位、存储保护等功能。MMU 的虚拟地址寄存器结构决定了块长度,不同的 CPU 的虚拟地址寄存器结构方式有些差异,但是,基本结构可以表示为如图 5-16 所示的形式。虚拟地址寄存器中低位部分的若干位表示页内地址,其余的高位部分表示页号。所谓页内地址,是指相对于所在页起始地址的偏移量。

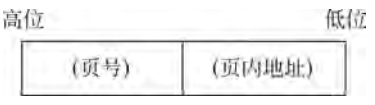

图 5-16 虚拟地址结构

例如,虚拟地址寄存器为 16 位,其中低 11 位表示页内地址,那么,可以得到:块长 $=2^{11}$,即 2K,一个进程的最大页数为 2^5,即 32 个页。如果虚拟地址寄存器为 32 位,其中低 12 位表示页内地址,那么,可以得到块长为 2^{12},即 4K,一个进程的最大页数为 2^{20},即 1M 个页。

由于一个程序的虚拟地址空间大小不一定是块长度的整数倍,所以,在程序装入时的分页过程中,最后不足块长度的程序信息也构成一个页。如图 5-15(b)所示,假定块长为 2048,即 2K,有两道程序 A 和 B,它们的大小分别是 6872 和 6016。对于程序 A,按照块长 2K 分页后,得到 4 个页,其中最后一个页只包含程序 A 的 728 个虚拟地址信息。同样地,对于程序 B,分页后得到 3 个页,其中最后一个页只包含程序 B 的 1920 个虚拟地址信息。

假定块长为 b,程序大小为 x,那么,如何计算程序分页后的页数,或程序装入时需要的块数?具体计算方法是:

$$x \div b = n \cdots\cdots r$$

如果 $r>0$ 则页数$=n+1$，否则页数$=n$。

也可以如下计算：

$$\text{页数}=(x+b-1)/b$$

这里的表达式语义采用C语言语法（没有特殊说明，本章的所有表达式语义都采用C语言语法）。注意，当 x、b 都是整数时，这里的算术表达式为

$$\text{页数}=(x+b-1)/b=\text{计算后的整数部分}$$

对于一道程序的任一个虚拟地址a，如何计算它所对应的页号p和页内地址w？

在MMU中，虚拟地址寄存器是二进制表示，设其中低k个位表示页内地址，按位运算，容易得

$$\mathrm{p}=\mathrm{a}\gg\mathrm{k}$$
$$\mathrm{w}=\mathrm{a}\ \&\ \underbrace{(11\ldots1)_2}_{\mathrm{k}}$$

例如，在图5-15(b)中，程序A的虚拟地址0x29B6（十六进制数）对应的页号p和w是多少？

因为 $\mathrm{a}=0\mathrm{x}29\mathrm{B}6=(0010,1001,1011,0110)_2$，所以

$$\begin{aligned}\mathrm{p}&=(0010,1001,1011,0110)_2\gg 11\\&=(101)_2\\&=5\\\mathrm{w}&=(0010,1001,1011,0110)_2\ \&\ \underbrace{(11\ldots1)_2}_{11}\\&=(0001,1011,0110)_2\\&=0\mathrm{x}1\mathrm{B}6\end{aligned}$$

另外，在平时习题的手工计算时，如果虚拟地址是十进制数表示，块长是b，则可以采用算术运算，具体如下：

$$\mathrm{p}=\mathrm{a}/\mathrm{b}$$
$$\mathrm{w}=\mathrm{a}\%\mathrm{b}$$

例如，在图5-15(b)中，程序B的虚拟地址4356对应的页号p和w是多少？

$$\mathrm{p}=4356/2048=2,\quad \mathrm{w}=4356\%2048=260$$

需要注意，进程的分页是在程序装入时操作系统自动进行，程序员或用户看不到分页的过程，程序员无法从源代码上准确估算变量或语句在哪个页上。

3. 非连续分配

在程序装入时，操作系统自动按块长对程序的虚拟地址空间进行分页，之后，以页为单位分配内存块，一个页分配内存的一个块，同一道程序的几个相邻的页，装入内存后，不要求在相邻的物理块上。也就是说，同一个页的程序信息在内存中是连续存放，但不同页之间，内存中的程序信息可以在不连续的存储单元上。

非连续的存储分配为程序装入带来很大的方便。因为，在为程序装入一个页时，页的长度等于块的长度，只要有一个空闲的块都可以用来分配。由此可以看出，在分页存储管理中，固定的块长度带来管理上的方便。

如图 5-15 所示,进程 A 的 4 个页分配的块号分别是 1、3、129、325,而进程 B 的 3 个页分配的块号分别是 2、128、326。

以上介绍了分页存储管理的基本思想。在实现时,分页存储管理又分为静态分页和动态分页两种。

1) 静态分页

在分页存储管理的内存分块、进程分页的基础上,如果程序装入时,要求把程序的所有页全部一次性地装入内存,那么,这种分页存储管理称为静态分页存储管理,简称静态分页或基本分页。

2) 动态分页

在分页存储管理的内存分块、进程分页的基础上,如果程序装入时只装入程序运行所需的基本页,其余的页仍然保留在外存中,那么,这种分页存储管理称为动态分页存储管理,简称动态分页。

相比之下,静态分页在进程运行过程中,其程序和数据信息已经全部装入内存,因此运行速度快,静态分页相对容易实现,但进程虚拟地址空间的大小不能超过内存的实际大小。而动态分页只装入程序运行所需要的基本页,因此允许进程的虚拟地址空间超过内存的实际大小,实现了虚拟存储器,但是动态分页需要解决 CPU 如何区分内、外存的信息、外存中的程序信息何时读入,以及如何读入内存等一系列问题。

5.5.2 静态分页的实现关键

下面介绍静态分页存储管理实现的关键,即数据结构设计及其初始化、地址转换。

1. 位示图及其作用

在静态分页存储管理中,首先需要考虑如何管理内存块的使用情况。有两种方法:位示图和空闲块链表。

1) 位示图

位示图(Bitmap)是一种应用广泛的数据结构,分页存储管理也可以采用位示图,用于表示内存块的使用状况。

在分页存储管理中,一个块只有两种状态:分配和空闲,可以用一个二进制的位表示,1 表示分配,0 表示空闲;另外,如果已知一个块的块号,就可以计算得到它的存储位置。

把表示块状态的二进制位按块号顺序依次排列,然后用一组数据表示出来,这组数据就称为位示图。

假定某内存空间共 256 个块,机器字长为 16 位,那么,表示内存块使用状况的位示图如图 5-17 所示。

存储空间的块总数和硬件机器的字长决定了位示图的结构。在位示图中,一个机器字长的字表示的数据作为一行,位示图就是一个由这些字表示的数据构成的一个矩阵。

假定,在位示图中的一个位用 bitmap[i,j]表示,其中 i 称为字号,表示第 i 行即第 i 个字;j 称为位号,表示在第 i 个字中的第 j 位,这里规定从低位开始计算。如果位示图中的第 i 个字记为 bitmap[i],那么 bitmap[i,j]=(bitmap[i]>> j)&1。

如果位示图的一个位 bitmap[i,j]表示的块号为 b,可以计算得到

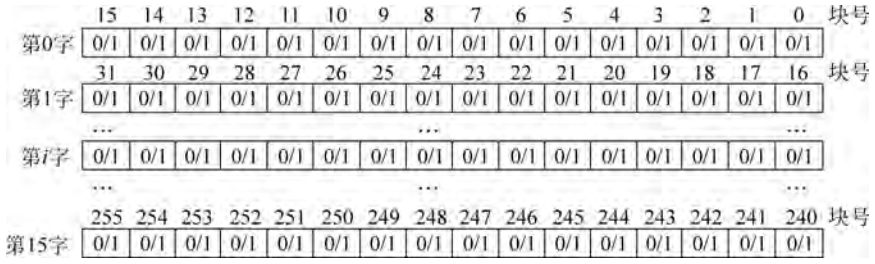

图 5-17 位示图的例子

$$b=字长*i+j$$

相反地，如果已知一个块的块号，这个块在位示图中的位为 bitmap[i,j]，则有

$$i=b/字长$$

$$j=b\%字长$$

可见，位示图的数据结构不仅计算简单，同时还节省了存储空间，因为位示图中无须直接登记块号信息，块号隐含在顺序上。

2）空闲块链表

表示内存块使用状况的数据结构，除了采用位示图之外，还可以用空闲块链表。

空闲块链表构造如下：链表的首指针存放第一个空闲块的块号，之后，每个空闲块的第一个存储单元存放下一个空闲块的块号，这样就可以把系统中的所有空闲块链接起来，链表中的最后一个空闲块的第一个存储单元存放 0 或空值，表示链表结束。

综上所述，在分页存储管理中，表示内存块使用状况的数据结构，可以使用位示图或空闲块链表。通常，为了分配方便，可以另外增加一个存储单元，用于存放当前空闲块的总数。

2. 页表

在分页存储管理中，进程占用的物理块可以不连续，因此，系统必须设计专门的数据结构，用于登记每一个进程的各个页及其所占用的内存块的关系，并提供给 MMU 实现重定位。

1）页表及其作用

登记进程页与块对应关系的数据结构称为页映射表，简称页表(Page Table)，页表的基本结构是：页号和块号，即页表中每个页表项(Page Entry)表示一个页与块的对应关系。如图 5-18 所示，分别描述了图 5-15 中进程 A 和进程 B 的页表。

每一个进程都有一个页表，页表描述进程页与块的对应关系。另外，从图 5-16 所示的虚拟地址结构看，表示页号或块号的数据中只占用机器字中的一部分位，这样，系统可以利用其余的位来登记页或块的访问控制和管理信息，因此页表的主要作用是重定位和存储保护。

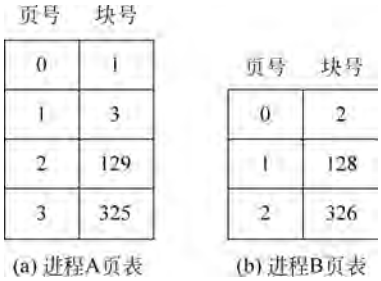

图 5-18 页表的例子

2）页表的建立和初始化过程

页表是操作系统的内核数据结构，保存在内核中。页表的建立和初始化过程如下：

在程序装入时，操作系统根据程序的大小，计算需

要的页数,然后检查系统当前的空闲块总数是否满足要求,如果能够满足,则依次为各个页分配内存块,如果不满足,则再判断是否超过用户区的块总数,决定是加入请求表,还是提示不能运行。

这里以位示图管理内存块为例,介绍分配一个空闲块过程,如图 5-19 所示。

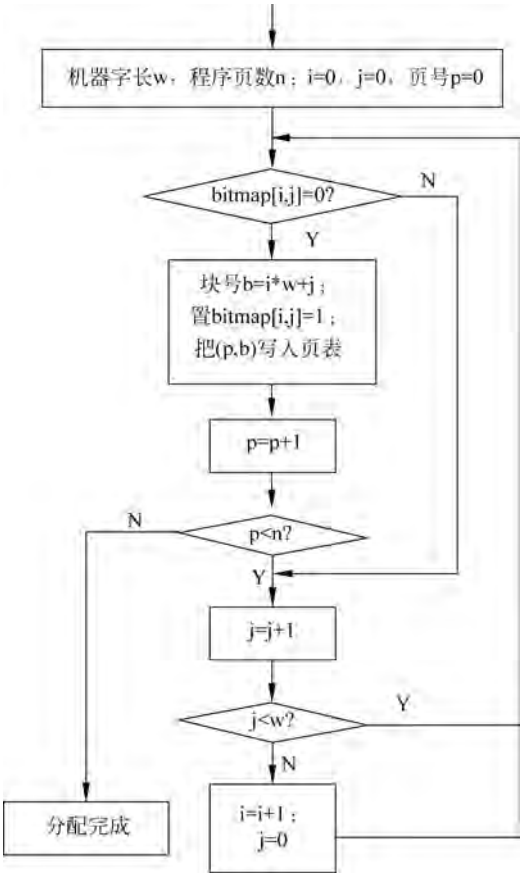

图 5-19 基于位示图的分配过程

在图 5-19 中,置 bitmap[i,j]=1 的表达式是:bitmap[i]=bitmap[i] | (1 << j)。并且,只有在空闲块总数满足当前进程的页数 n 的要求时,才进行分配,所以在图 5-19 的分配流程中 i 不会大于位示图的字数。

空闲区链表管理空闲块时的分配空闲块过程留作习题。

事实上,页表的建立和初始化过程,就是内存的分配过程。

程序装入创建进程时,建立了页表,进程的页表存放在 PCB 中。

这样,在进程撤销时,从 PCB 中得到进程的页表,页表中内存块的回收过程,如图 5-20 所示。

其中置 bitmap[i,j]=0 的表达式是:bitmap[i]=bitmap[i]^(1 << j)。

3. 请求表

请求表用于登记因内存没有足够空闲块而造成暂时无法装入的程序信息。请求表的结

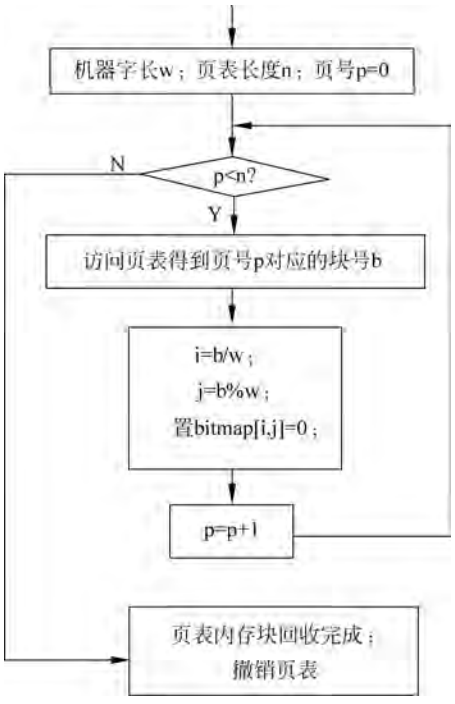

图 5-20　基于位示图的回收过程

构由程序名、用户名及申请页数等信息组成。系统在撤销一个进程时，需要检查请求表，及时把满足申请要求的程序装入内存。

4. 重定位及存储保护

分页存储管理采用动态重定位。重定位及存储保护由 CPU 的存储管理单元 MMU 实现，在 MMU 中，有两个寄存器：页表基址寄存器和页表限长寄存器，分别存放当前运行进程的页表起始地址和页表长度即页数，它们作为 CPU 现场信息的一部分，在进程调度时需要保护或恢复；另外，虚拟地址寄存器存放的是当前要访问的指令或数据的虚拟地址。

如图 5-21 所示，描述了分页存储管理的重定位过程，其步骤概括如下：

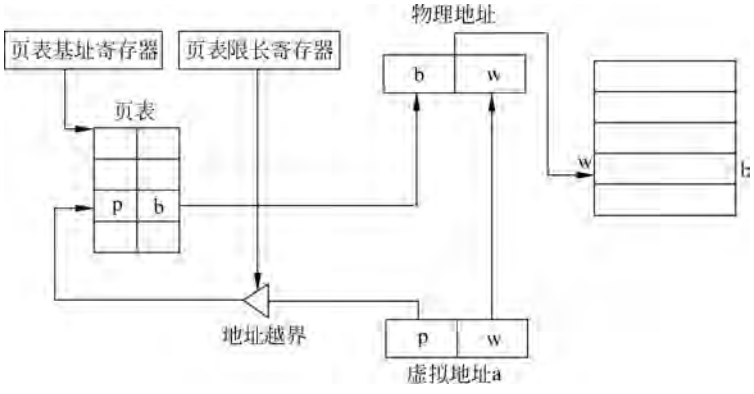

图 5-21　分页的重定位

(1) 页号 p 和页内地址 w。

MMU 按照虚拟地址寄存器的结构,取其中的低位部分位,得到页内地址 w,再通过移位操作得到高位部分的页号 p。设虚拟地址寄存器中低 k 个位表示页内地址,则

$$p = a >> k$$

$$w = a \ \&\ \underbrace{(11\ldots1)_2}_{k}$$

例如,在第 5.5.1 小节中,如图 5-15 所示,进程 A 的虚拟地址 a=0x29B6(十六进制数)对应的页号 p=5,w=0x1B6,程序 B 的虚拟地址 a=4356 对应的页号 p=2,w=260。

(2) 存储保护。

检查页号 p 是否小于页表限长寄存器的值,如果页号 p 大于或等于页表限长寄存器的值,则产生一个地址越界中断。

例如,进程 A 的虚拟地址 a=0x29B6(十六进制数)对应的页号 p=5,超过了进程 A 的页表长度,如果当前 CPU 访问进程 A 的虚拟地址 a=0x29B6,则地址越界中断;而进程 B 的虚拟地址 a=4356 对应的页号 p=2 没有超过进程 B 的页表长度,对于进程 B 是合法的虚拟地址。

(3) 访问页表得到块号。

从页表基址寄存器得到页表在内存中存放的起始地址,按页号 p 访问页表得到对应的块号 b。由于每个进程的页号都是从 0 开始连续编号,且页表中每个表项的长度是固定的,因此,可以根据页号 p 容易计算得到它在页表中的偏移量,也就是说这里并不需要查找,可以按照

$$\text{页表起始地址} + \text{页表项长度} \times p$$

地址直接读取页号 p 对应的表项而得到块号 b。

例如,进程 B 的虚拟地址 a=4356 对应的页号 p=2,从页表(见图 5-18)得到 b=326。

这时,在一些高性能处理器中,MMU 还可以利用页表进行存取控制检查。

(4) 形成物理地址。

根据页号 p 从页表中读取得到对应的 b,知道页号 p 的信息存放在内存中块号 b 的存储空间上,从而得到物理地址

$$(b << k) + w$$

其中,b << k 等于 b * 块长。

如图 5-18 所示,进程 B 的虚拟地址 a=4356 对应的物理地址是

$$326 \times 2K + 260$$

例 5-3 在某静态页式存储管理中,已知内存共有 32 块,块长度为 4K,当前位示图如图 5-22 所示,进程 P 的虚拟地址空间大小为 50000。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	1	0	1	1	1	1	1
0	0	0	0	0	1	1	0	0	0	1	0	0	1	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

图 5-22 位示图的例子

问：

(1) 进程 P 共有几页？

(2) 根据图 5-22 的位示图，给出进程 P 的页表。

(3) 给定进程 P 的虚拟地址：8192(十进制)和 0x5D8F(十六进制)，根据(2)的页表，分别计算对应的物理地址。

解：

(1) 进程 P 的页数＝(50000＋(4K－1))/4K＝13 页。

(2) 依次扫描位示图，为进程 P 分配内存块，得到进程 P 的页表，如图 5-23 所示。

0	5
1	8
2	11
3	13
4	16
5	19
6	20
7	22
8	23
9	24
10	27
11	28
12	29

图 5-23 页表的例子

(3) 重定位。

① 虚拟地址 8192(十进制)转换为物理地址的过程如下：

计算虚拟地址 8192 的页号 p 和页内地址 w，得

$$p=8192/4K=2,w=8192\%4K=0$$

访问页表得 p 对应的块号为 b＝11。

计算物理地址：

$$11*4K+0=44K$$

② 虚拟地址 0x5D8F (十六进制)转换为物理地址的过程如下：

因为 $0x5D8F=(0101,1101,1000,1111)_2$，所以

$$p=(0101,1101,1000,1111)_2 \gg 12$$
$$=(0101)_2$$
$$=5$$
$$w=(0101,1101,1000,1111)_2 \& \underbrace{(11\ldots1)_2}_{12}$$
$$=(1101,1000,1111_2$$
$$=0xD8F$$

根据 p＝5，访问页表得 b＝19。

计算物理地址：

$$19\times4K+0xD8F=0x13000+0xD8F=0x13D8F$$

5.5.3 静态分页的特点及效率的改进

静态分页存储管理是现代计算机系统存储管理的基础，其主要特点是：非连续的存储分配提高了存储空间的利用率；页长度与块长度相等使得在存储空间分配时无须采用复杂的策略；另外，分页存储管理为实现虚拟存储器提供了可能。

但是，在分页存储管理中也存在一些不足。

例如，从上述重定位过程中可以看出，在分页存储管理中，CPU 每访问一条指令或一个数据，都要两次访问内存：一次是在 MMU 重定位过程中，根据页号访问内存中的页表得到块号；另一次是 CPU 根据重定位得到的物理地址访问内存中的指令或数据。与分区管理相比，分页存储管理的两次访问内存增加了 CPU 的开销。

如何改进分页存储管理的效率？通过硬件上提供的 TLB 技术，引入快表可以减少

CPU 访问内存的次数。

在 MMU 中增加一组专用的硬件高速缓冲区,称为 TLB(Translation Lookaside Buffers),也称联想存储器(Associative Memory),用于存放一些页与块的对应关系。由于 TLB 中存储空间容量的限制,通常只能存储 8～64 个对应页与块的关系。虽然现代的处理器硬件提供 TLB 的存储空间更大,但往往还是不能把当前进程的整个页表存入 TLB 中。

把 TLB 中所存储的页与块的对应关系称为快表。由于 CPU 访问 TLB 的速度远远大于访问内存的速度,所以快表可以提高分页存储管理的效率。

在具有 TLB 的处理器中,MMU 在重定位时,根据页号 p,在快表中查找,如果能找到对应的块,则直接形成物理地址;如果页号 p 不在快表中,则访问页表得到块号,并把页与块的对应关系加入快表。由于访问 TLB 所花的时间远小于访问内存的时间,所以,如果能够把当前进程经常访问的页与块的对应关系存放在快表中,那么,大多数情况下,可以减少 CPU 访问内存的次数,也就减少了 CPU 的开销。

具有快表的重定位过程如图 5-24 所示。

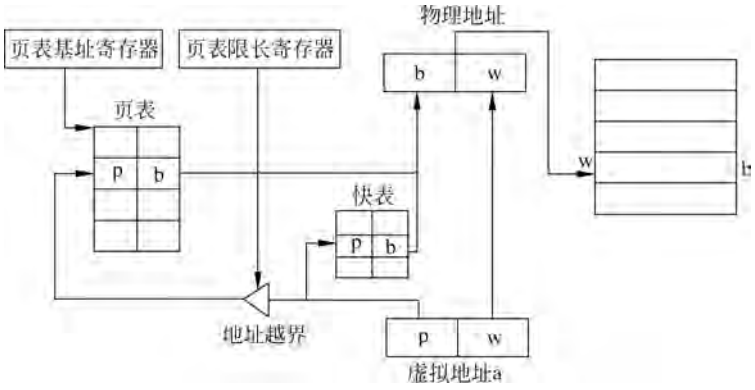

图 5-24 具有快表的重定位过程

通常系统无法把一个进程的整个页表存入 TLB,所以,MMU 在重定位过程中,有的页能够在快表中找到对应的块号,有的页只能在通过访问页表得到块号。为此,系统引入命中率的概念来描述快表的效率。给定一个时间段,MMU 在访问页对应的块时,如果有 m 次在快表中得到块号,有 n 次在页表中得到块号,那么,这个时间段 MMU 的快表命中率 h 定义为:

$$h=\frac{m}{m+n}\times 100\%$$

在具有快表的分页存储管理中,应尽可能提高快表的命中率。

5.5.4 虚拟存储器思想

相比而言,外存储器的空间容量比内存储器的空间容量大,而且,程序在运行之前,程序信息本来就保存在外存储器中,如果操作系统在管理主存储器时能够利用外存储器的部分空间充当主存储器来使用,就可以允许单个程序的大小超过内存的实际大小。

在实现虚拟存储器时,首先面临的问题是,一道程序在只装入一部分的情况下,能不能运行?也就是说,虚拟存储器的思想是否可行?

虚拟存储器要解决的主要技术包括理论基础、调入策略和置换算法。

1. 理论基础

程序的局部性原理是虚拟存储器思想的理论基础。

事实上，程序在运行过程中，在一个较小时间范围内，并不要求程序的完整信息全部都在内存中。例如，在结构化程序设计中，程序分为若干较小的模块，在这些模块中，往往包含一些错误处理模块，只有在程序运行出现非期望状态时才转入运行错误处理模块；如果程序在一次运行过程中都是正常的，则错误处理模块就不需要，这时如果这些错误处理模块代码不在内存中，也不影响程序的运行。

类似地，一个应用程序往往实现很多功能，而在一次的运行过程中，用户通常只操作其中的一部分功能，其他功能并没有选择执行，这些未被用户选择执行的功能对应的模块也可以不在内存中。

另外，程序中通常包含很多的分支结构，在程序的一次运行中，同一组的几个分支结构，往往只运行其中一个分支代码，其他分支代码不需要运行。

还有，程序具有顺序性特点，只有在当前的指令执行后，才能执行下一条指令，因此，程序在运行的开始，后半部分的程序信息可能暂时不需要，而在一道程序执行到后半部分代码时，它的前面已经运行的多数代码可能暂时也不需要。

人们在研究、分析大量程序的运行后，发现程序的运行具有一定的特点，得到程序的局部性原理：在程序运行过程的一个较小时间范围内，只需要一小部分的程序信息，其他部分暂时不需要；而且在程序的一次执行过程中，程序的所有指令和数据并没有相同的访问概率，部分指令和数据经常被访问，部分指令和数据很少被访问，甚至部分指令和数据根本没有被访问。

程序的局部性原理又分为时间局部性和空间局部性。

时间局部性是指，某一个地址的存储单元信息被访问后，在不久的将来，这个地址的存储单元信息被再次访问的可能性很大。

空间局部性是指某一个地址的存储单元信息被访问后，在不久的将来，与这个地址相邻的存储单元信息被访问的可能性很大。

程序的局部性表明，对于大多数程序，在运行过程中，如果某些指令或数据经常被访问，那么，在接下来的运行中，这些指令或数据还将经常地被访问。

2. 调入策略

现在，同一道程序的一部分信息在内存，另一部分在外存，由于 CPU 只能访问内存的信息，因此，程序中在外存中的信息需要调入内存，那么，外存中的程序信息什么时候调入内存？

调入策略就是决定程序在外存中的信息什么时候调入内存。有两种方式：请求调入策略和预调入策略。

请求调入策略的思想是：需要时调入，即在处理器要运行某指令或访问某数据时，发现它不在内存，这时再从外存将其调入。

预调入策略的思想是：在处理器运行之前，事先把将要访问的程序指令或数据调入内存，这样，处理器运行时保证所需要的指令或数据已经在内存中。

相比来看,请求调入策略语义明确,比较容易实现;预调入策略虽然可以提高程序的运行速度,但由于很难事先预计程序的运行流程,因此实现比较困难。现代的高性能处理器推出了预测执行技术,例如分支预测执行技术等具有一定程度的预调入功能。

目前,虚拟存储器主要采用请求调入策略。

3. 置换算法

在调入策略决定需要将程序的信息从外存读入内存时,可能当前没有足够的空闲存储单元,这时需要将已经在内存中的程序信息调出,把空出的存储单元用于存放新读入的程序信息,所以,需要根据一定的策略,选择在内存中的程序信息,将其调出或淘汰,这种策略称为置换算法。

5.5.5 请求分页的实现关键

动态分页按照调入策略的不同,分为请求分页和预调入分页。其中预调入分页实现比较困难,所以,动态分页主要是采用请求分页。

在请求分页存储管理中,内存空闲块管理的数据结构可以直接采用静态分页中的位示图或空闲块链表。

请求分页在实现时关键需要解决:①CPU 如何区分内、外存的页信息;②重定位过程中当发现访问的虚拟地址信息所在的页不在内存时,需要产生特殊的 I/O 操作从外存将其所在的页读入内存;③在从外存读一个新的页装入内存时,内存没有空闲块如何处理;④程序运行所需的基本页如何确定等。

下面分别介绍上述 4 方面问题的解决方法,即扩充页表、缺页中断及其处理、页面调度的置换算法和工作集原理。

1. 扩充页表

CPU 如何区分内、外存的页信息?可以通过扩充页表实现。因为,分页存储管理采用动态重定位,MMU 的重定位通过页表实现,因此可以在页表中增加一些实现请求分页所需要的信息。

扩充页表的基本结构主要由页号、块号、外存地址、中断位 P、访问位 A、修改位 M 等组成。

页表中每一个表项描述一个页的装入状态信息和位置信息,扩充部分的含义及作用如下:

中断位 P 用于标识内、外存信息,P=1 表示该页在内存,这时块号表示该页在内存中的位置;P=0 表示该页不在内存中,即未装入内存或被淘汰调出,此时,页表中的外存地址表示该页在外存中的存放位置。中断位 P 由操作系统调入或调出时设置。

访问位 A 是页面调度参数之一。由操作系统设置清零,相关硬件在执行具体的访问操作时修改。不同的页面调度置换算法,访问位 A 具有不同的含义。

修改位 M 表示该页装入内存后是否被修改,即该页是否被执行了写操作,M=0 表示没有被修改,M=1 表示已经被修改。修改位 M 在页面调度时使用,可以减少写 I/O 操作。由操作系统的装入页时设置清零,相关硬件在执行具体的写操作时置 M=1。

从图 5-16 的虚拟地址结构看，在表示页号或块号的数据中只占用机器字中的一部分位，所以，在扩充页表中，为了减少页表的存储开销，可以利用表示页号或块号数据中的其余位登记对应页的中断位 P、访问位 A、修改位 M 等信息。

可见，在请求分页中，页表起了重要作用，不仅用于进程运行过程的重定位和存储保护，而且是实现页面调度的主要数据结构。

2. 缺页中断及其处理

请求分页的重定位过程如下：

根据虚拟地址的页号，先检查快表，如果找到匹配的页，再进行存取控制的检查，如果符合访问权限，则得到对应的块，如果当前操作不符合访问权限，则产生非法存取的异常；如果该页不在快表中，则访问页表。

MMU 重定位过程在访问页表时，如果该页对应的中断位 P＝1，说明该页在内存中，这时再进行存取控制的检查，如果符合访问权限，则修改访问位 A，或设置修改位 M，利用页表中的块号形成物理地址，同时将页号和块号写入快表；如果当前操作不符合访问权限，则产生非法存取的异常。

如果该页对应的中断位 P＝0，说明该页当前不在内存中(Page Fault)，这时，再进一步判断，如果属于非法访问内存地址，如指针或数组越界等，则产生非法存取的异常；否则，表明该页未读入内存或已经被淘汰调出，因此，MMU 产生一个特殊的 I/O 中断，请求操作系统将该页从外存读入内存，把 MMU 产生的这个 I/O 中断称为缺页中断(Page Fault Trap)。

MMU 的缺页中断产生后，操作系统响应这个特殊的中断。图 5-25 描述了 MMU 重定位过程以及操作系统的缺页中断处理过程。

操作系统的缺页中断处理过程如下：

1) 现场保护

撤销当前指令的执行，现场恢复到导致缺页中断的指令执行前的状态，并保护现场。

2) 分配内存块

首先，检查系统是否有空闲块，如果存在空闲块，则从中得到一个空闲块。

如果当前内存没有空闲块，则执行页面调度程序，按照指定策略从内存中选择一个页将其信息调出到外存，这个过程称为淘汰，修改淘汰的页所对应页表的相关信息如中断位 P＝0，并判断修改位 M。

如果淘汰页的修改位 M＝0，则无须调出保存，因为页表中对应的外存地址就是该页信息在外存的一个副本。

如果淘汰页的修改位 M＝1，说明该页信息被修改，则需要调出另外保存，调出后所保存的外存地址写入页表中淘汰页对应的外存地址，同时，置淘汰页修改位 M＝0。

可见，扩充页表中的修改位 M 的作用是在缺页中断处理过程中，减少系统的写 I/O 操作。因为当 M＝0 时，不需要保存淘汰的页，只有在 M＝1 时，才需要一个额外的写 I/O 操作。淘汰后得到一个空闲块，用来装入新读入的页信息。

3) 启动读 I/O 操作，从外存读取新的页信息

按页表中登记的外存地址，将所缺的页从外存读入内存指定的空闲块；最后，修改该页在页表中的信息。例如，块号写入页表，置中断位 P＝1，访问位 A＝0，修改位 M＝0。至此

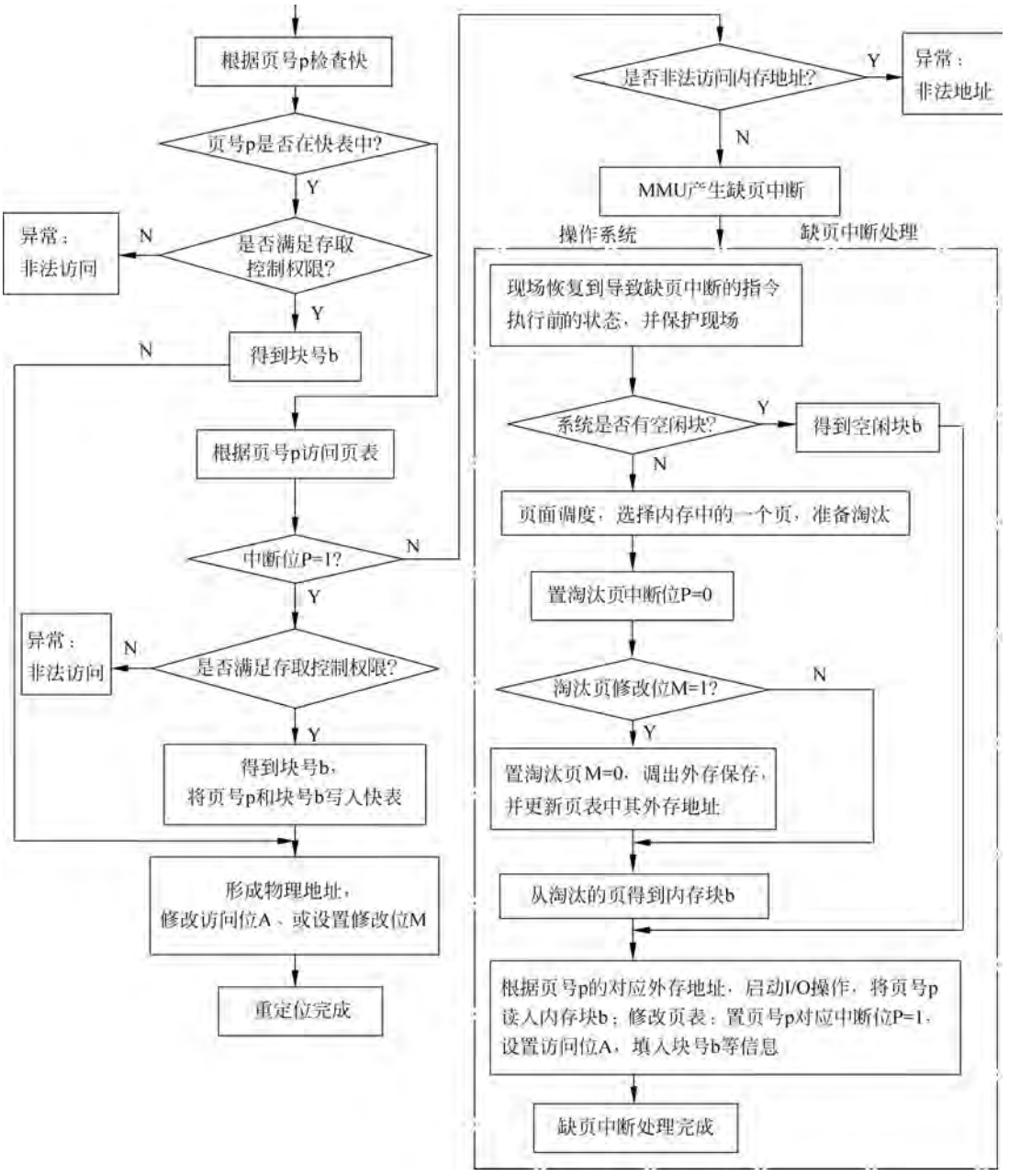

图 5-25　重定位和缺页中断处理

缺页中断处理完成。

当缺页中断处理完成后，进程经过进程调度程序选中，继续运行时，再次运行原先引起缺页中断的指令，这时指令所在的页已经读入内存了。

需要指出的是，对于复杂指令，一条指令的运行可能会产生多次的缺页中断。例如，指令本身所在的页不在内存，运行时将产生缺页中断；如果指令的某个操作数与指令不在同一个页，且操作数所在的页也不在内存中，则将又会产生缺页中断。一条指令的多次的缺页中断可能会引起复杂的问题。

3. 页面调度

操作系统的缺页中断处理过程，要为新读入的页分配一个空闲块，如果内存没有空闲块，必须按指定的策略，从内存中选择一页将其信息淘汰，空出的块分配给新的页，这个过程称为页面调度。

页面调度需要解决两方面问题：一是确定选择的范围，只允许从当前进程页表中选择，还是可以从其他进程的页表中选择？二是设计选择的策略，即在指定范围内如何选择一个页？

根据对第一个问题的不同处理，把页面调度分为局部页面调度和全局页面调度。局部页面调度只允许在当前运行进程的页表中选择要淘汰的页，全局页面调度可以在所有进程的页表中选择要淘汰的页。

第二个问题的解决称为置换算法(Page Replacement Algorithm)，即在确定是局部还是全局的情况下，根据指定的策略选择一个页将其淘汰。

对于内存中的页，如果被置换算法选中淘汰，则在系统的将来运行过程中，可能还会被访问，因而又产生缺页中断，又需要将其再次读入内存。因此，在请求分页管理中，如果置换算法设计不当，可能造成一种现象，刚刚调入内存的页，很快地被淘汰调出到外存，在淘汰后不久处理器又要访问到它，而需要将其从外存调入内存，这样，可能出现在一段较短的时间范围内，集中在少数的几个页之间，系统频繁地进行调入和调出操作。把这种状况称为抖动(Thrashing)现象，或者颠簸。抖动现象将导致 CPU 频繁地调度、切换和 I/O 操作，大大增加了 CPU 的开销，影响请求分页的效率。

4. 置换算法

为了减少 CPU 的开销，必须合理地设计置换算法，尽可能减少抖动现象。

理想或者优化(Optimal)的置换算法是，置换算法淘汰一个页时，能够选择内存中将来 CPU 没有访问的页，或者是在相比之下，选择最久以后才被访问的页。但是，由于一个进程的运行流程是不可预知的，所以这种置换算法的思想难以实现。

置换算法的目标是：在内存中尽可能保留进程运行过程中经常访问的页，以减少缺页中断的次数。

基本的置换算法有先进先出(First In First Out，FIFO)算法、最近最久未使用(Least Recently Used，LRU)算法和最近最不常用(Least Frequently Used，LFU)算法。

另外，还有一些其他算法。如最近未使用(Not Recently Used，NRU)算法、二次机会(Second Chance)算法和页缓冲(Page Buffering)算法。

1) 先进先出算法

先进先出(FIFO)算法的策略是：将内存中的页按装入内存的先后顺序排列，淘汰时，选择最先进入内存的页。

FIFO 算法的依据是程序的空间局部性原理。由于程序的顺序性特点，即一道程序运行时，按指令顺序依次执行，最先进入内存的页，之后被再次访问的可能性小。如一些初始化的代码，在进程首次执行时进行初始化，之后就不再需要了，将这些初始化的代码淘汰是较好的选择。

例如,一个进程运行过程中依次访问的页号(也称进程的引用序列)是:0、2、3、1、4、1、2、3、5、2、3、1、4、5、0、3、6、9、8、3、6、7、3、6、9、8、7。假定分配给该进程 4 个块,按局部页面调度,采用 FIFO 算法时,如何计算缺页中断的次数?依次淘汰的页号是哪些?

如图 5-26 所示,图中第一行列出进程的引用序列,引用序列下方的 4 行表示内存中的页装入的 FIFO 序列,在序列中,箭头方向上的页为较先进入的页。

进程引用序列	0	2	3	1	4	1	2	3	5	2	3	1	4	5	0	3	6	9	8	3	6	7	3	6	9	8	7
内存页 FIFO序列	0	2	3	1	4	4	4	4	5	2	3	1	4	5	0	3	6	9	8	8	8	7	3	6	9	8	7
		0	2	3	1	1	1	1	4	5	2	3	1	4	5	0	3	6	9	9	9	8	7	3	6	9	8
			0	2	3	3	3	3	1	4	5	2	3	1	4	5	0	3	6	6	6	9	8	7	3	6	9
				0	2	2	2	2	3	1	4	5	2	3	1	4	5	0	3	3	3	6	9	8	7	3	6
依次淘汰的页	×	×	×	×	0				2	3	1	4	5	2	3	1	4	5	0			3	6	9	8	7	3

图 5-26 FIFO 算法的例子

在图 5-26 中,每个访问页的正下方的列表示该页访问后的内存页的 FIFO 序列。如果当前访问的页在内存中,则该页访问后,不影响原来内存中各页的顺序,其正下方的内容直接来自上一次访问页的 FIFO 序列;如果当前访问的页不在内存中,则它是最新装入的,这时,将它加入其正下方的 FIFO 序列中,加入时,对应的序列中原来的各个页依次下移一行,如果序列中原来的页数已经达到 4 个,则箭头方向的第一个页是最先装入内存的,将其淘汰并记录在最后一行的相应列的正下方,并把当前页加入 FIFO 序列末尾。

图 5-26 中的最后一行给出了进程运行过程依次淘汰的页号:0、2、3、1、4、5、2、3、1、4、5、0、3、6、9、8、7 和 3,共 18 次,另外,加上开始运行时装入的 4 个页,则共有 22 次缺页中断,其中 18 次缺页中断运行了置换算法。

FIFO 算法比较简单,容易实现,被很多的操作系统所采用。FIFO 算法的不足是可能会将经常访问的页淘汰。在很多进程中,最先进入内存的页,在之后的运行过程中,也可能被频繁访问。例如,基于菜单的人机交互操作方式中,对于菜单初始化的模块,在进程开始时运行一次,之后,当用户选择的一个菜单项的任务完成时,通常需要在屏幕上更新菜单,再次调用菜单初始化模块。这样,可能存在一些 CPU 经常访问页面,由于最先进入内存而被 FIFO 算法选中淘汰,增加了缺页中断的次数。

2) 最近最久未使用算法

为了克服 FIFO 置换算法的不足,人们提出最近最久未使用(Least Recently Used,LRU)算法,其思想是,页表中登记每个页被 CPU 访问的时间,淘汰时选择最近一段时间最久没有被访问的页。

LRU 算法的依据是程序的时间局部性原理,即一个页最近被访问后,接下来,被再次访问的可能性很大,相反地,长时间没有被访问的页,之后很可能也不会被访问。

对于图 5-26 中的进程引用序列,采用 LRU 算法时,进程运行过程的缺页中断如图 5-27 所示。

图 5-27 的方法是模拟堆栈的方式。图中箭头方向是栈底的方向,栈底方向是较长时间没有被访问的页,而栈顶的页是新近访问的。在访问一个页时,如果该页不在“堆栈”中,则从栈顶进栈,在进栈之前,如果堆栈中的页数已满 4 页,则删除栈底的页,因为它是最长时间没有被访问的页,将其淘汰并记录在最后一行的正下方;如果当前访问的页已经在“堆栈”

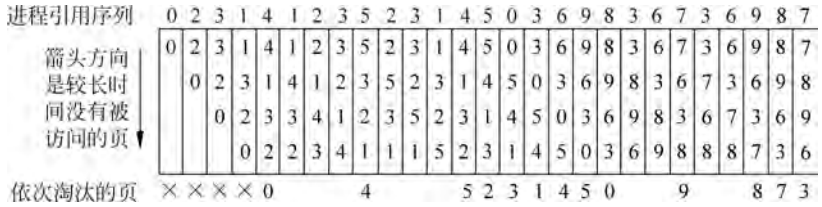

图 5-27　LRU 算法的例子

中，则将其从所在位置移至栈顶，在“堆栈”中它上方的页依次下移一个位置。

图 5-27 中的最后一行给出了进程运行过程中依次淘汰的页号：0、4、5、2、3、1、4、5、0、9、8、7 和 3，共 13 次，另外，加上开始运行时装入的 4 个页，则共有 17 次缺页中断，其中 13 次缺页中断运行了置换算法。

3）最近最不常用算法

最近最不常用(Least Frequently Used，LFU)算法也可以克服 FIFO 算法的不足，其思想是：页表中登记每个页被 CPU 访问的次数，淘汰时选择最近一段时间被访问次数最少的页。

如图 5-28 所示是一个 LFU 算法的例子，假定分配给进程 4 个块，其中，进程的引用序列是：0、2、3、1、4、1、2、3、5、2、3、1、4、5。在图中，内存每个页的访问次数标注在其右上角，并按访问次数排序，其中箭头方向是访问次数较少的页(这里规定访问次数相同的页按 LRU 算法处理)。

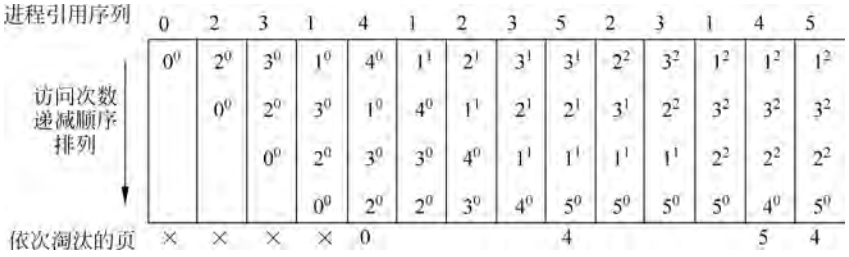

图 5-28　LFU 算法的例子

图 5-28 中的最后一行给出了进程运行过程依次淘汰的页号：0、4、5、4，共 4 次，另外，加上开始运行时装入的 4 个页，则共有 8 次缺页中断，其中 4 次缺页中断运行了淘汰算法。

LFU 算法有可能导致新装入的页很容易被淘汰，如图 5-28 中引用序列后面的页 5 和页 4，这似乎又与程序的局部性原理矛盾。

4）最近未使用算法

LFU 和 LRU 算法实现时有很大的系统开销，主要表现在：一方面，保存和修改每个页的访问时间或次数，需要多个二进制位才能实现，这必然要增加页表的开销；另一方面，如果访问的页在 TLB 的快表中，那么再修改页表中对应页的访问时间或次数，将失去 TLB 的作用。

最近未使用(Not Recently Used，NRU)算法是 LFU 和 LRU 算法的一种近似实现，其思想是，页表中每个页都关联一个访问位 A，操作系统定期设置内存中所有页的访问位 A＝0，一个页被访问时由硬件置访问位 A＝1，淘汰时选择访问位 A＝0 的一个页。

NRU 算法需要一个定时器，定时器定期产生一个中断，触发操作系统对所有访问位 A 清零的操作。那么，定时器的周期成为 NRU 算法的实现关键。因为，如果周期太长，淘汰

时可能所有页都有 A=1,如果周期太小,则又存在很多 A=0 的页。

5) 二次机会算法

二次机会(Second Chance)算法是结合页表中的访问位 A 对 FIFO 算法进行改进而得到的,其思想如下:

将内存中的页按装入内存的先后顺序排列,在缺页中断处理过程,需要淘汰一个页时,算法过程如下:

① 选择最先进入内存的页,检查其访问位 A。

② 如果该页的访问位 A=0,则该页被选中而淘汰。算法结束。

③ 如果该页的访问位 A=1,则置其访问位 A=0,并视该页为新装入的而保留在内存中转①。

在二次机会算法中,如果内存中的所有(全局或局部)页全部检查完,还没有找到要淘汰的页,那么,接下来的一次所检查的页,一定有访问位 A=0。所以,保证算法在有限次数的检查就可能够找到要淘汰的页。

在二次机会算法中,一个页如果被访问一次,则它在算法的首次检查时不会被淘汰,除非在算法执行之前所有的页都被访问了,它才可能被淘汰,因此可以保证经常访问的页在内存中。

图 5-29 描述了二次机会算法的执行过程,箭头方向上的第一个页表示当前要检查的页,检查时按箭头反方向逐个进行。

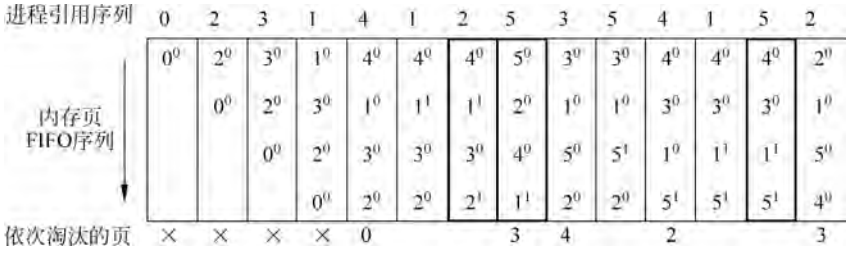

图 5-29 二次机会算法的例子

二次机会算法也称时钟(Clock)算法,因为二次机会算法的实现过程,好像时钟钟表的指针一样,不断循环地行进。具体描述如下:

内存中的页按照装入内存的时间排列,并定义一个指针,指向当前要检查的页位置。

检查时,如果指针指示的页的访问位 A=0,则将其淘汰,新装入的页存储在指针所指示的位置;如果指针指示的页的访问位 A=1,则将其保留下来,并置访问位 A=0。

检查后,指针指向下一个页,如果没有下一个页则指针又指向最先装入的页。

因为二次机会算法在检查一个页时,如果它被保留下来,则置其访问位 A=0,再继续检查下一个页,所以,至多在检查了当前内存中的所有页后,接下来一次检查就可以找到一个淘汰的页而结束算法。

对于图 5-29 中的结果,CPU 访问第 1 个页号 5 和最后一个页号 2 时二次机会算法的执行过程作解释如下:

在图 5-29 中,CPU 访问第一个页号 5 时,内存中的 4 个页面的 FIFO 序列是:2、3、1 和 4,其中,它们的访问位 A 分别是 1、0、1 和 0,如图 5-30(a)所示(图中各页的访问位标在其右

上角),这时指针指向页号 2 的位置,即图 5-29 中的箭头最前方位置的页。在访问页号 5 时,页号 5 不在内存中,二次机会算法检查时发现当前指针位置的页号 2 的访问位 A=1,所以置 A=0,指针下移;指向页号 3,页号 3 的访问位 A=0,这时,算法淘汰页号 3,并将页号 5 填入页号 3 的位置,如图 5-30(b)所示,指针下移,如图 5-30(c)所示。得到内存中的 4 个页面的 FIFO 序列是 1、4、2、5,下一次检查时从图 5-30(c)指针位置开始。

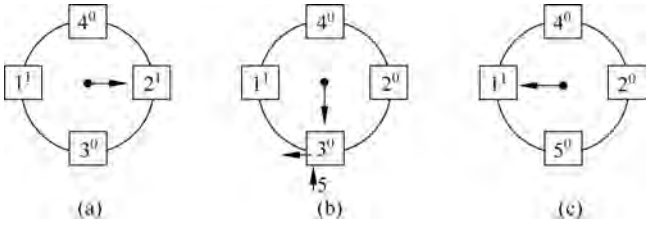

图 5-30 二次机会算法的例子(访问页号 5)

在图 5-29 中,CPU 访问最后一个页号 2 时,内存中的 4 个页面的 FIFO 序列是 5、1、3 和 4。其中,它们的访问位 A 分别是 1、1、0 和 0,如图 5-31(a)所示,指针指向页号 5 的位置。在访问页号 2 时,页号 2 不在内存中,二次机会算法检查时发现当前指针位置的页号 5 的访问位 A=1,所以置 A=0,指针下移,指向页号 1;页号 1 的访问位 A=1,再置 A=0,指针下移,指向页号 3,此时,页号 3 的访问位 A=0,算法淘汰页号 3,并将页号 2 填入页号 3 的位置,如图 5-31(b)所示,指针下移,如图 5-31(c)所示,得到内存中的 4 个页面的 FIFO 序列是 4、5、1、2。

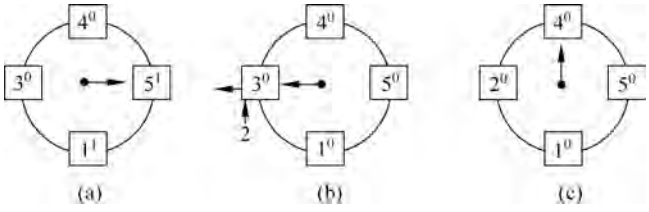

图 5-31 二次机会算法的例子(访问页号 2)

淘汰内存中修改位 M=0 的页要比淘汰 M=1 的页减少 1 次的 I/O 操作。因为 M=0 的页信息在外存中已经有一次副本,其所在的块可以直接用于装入新读入的页,而淘汰 M=1 的页时,需要先执行一个写 I/O 操作,将它保存到外存,然后才能装入新的页。

改进型二次机会算法是同时考虑访问位 A 和修改位 M 的一种置换算法,可以减少 I/O 操作。

改进型二次机会算法思想如下:

按装入内存的时间排列,得到内存页的 FIFO 序列,再按访问位 A 和修改位 M,内存的页分为如下 4 类:

① A=0,M=0。淘汰这类页可以减小 I/O 操作开销,是置换算法的首选页。

② A=0,M=1。淘汰这类页需要额外的写 I/O 操作,但可以减少抖动现象。

③ A=1,M=0。淘汰这类页可以减小 I/O 操作开销,但可能产生抖动现象。

④ A=1,M=1。这类页是在算法执行的最后不得已的选择。

算法开始时,先保存当前指针,然后按如下步骤进行检查:

① 对于内存页的FIFO序列,从当前指针位置开始,依次查找属于①类的页,如果能够找到,则淘汰,并把新装入的页号填入这个位置,指针下移,如果超过FIFO序列的末尾则指针又从首位置开始,算法结束;如果整个序列扫描完毕(通过与之前保存的指针对比来判断),没有找到属于①类的页,则进行第二遍检查。

② 第二遍检查时,从当前指针(应等于之前保存的指针)位置开始,依次查找属于②类的页,检查一个页时如果属于②类,则淘汰,并把新装入的页号填入这个位置,同时A和M清零,指针下移,算法结束;否则,置A=0,指针下移继续检查。这里指针下移时如果超过FIFO序列的末尾则指针又从首位置开始,如果整个序列扫描完毕,没有找到属于②类的页,则转步骤①再次查找。

当第二次从步骤①开始时,内存中的所有页都有A=0,因此,此时如果不存在①类的页,那么,肯定都是②类的页,所以,在重新从步骤①开始后,至多在步骤②中,就可以找到可以淘汰的页。

改进型二次机会算法目标是尽可能减少I/O操作,并减少产生抖动现象,但算法的开销有所增加。

6) 页缓冲算法

页缓冲算法(Page Buffer)不需要复杂硬件的支持,其思想如下:

系统设置剩余空闲块数量的界限,例如,设置剩余空闲块数量不足内存总块数的1/4和1/8两个界限。另外,再设置一个页缓冲区(Page Buffer),页缓冲区用于保存两个页缓冲链表:未修改页链表(M_0链表)和修改页链表(M_1链表)。

在产生缺页中断为新读入的页分配内存块时,如果剩余空闲块数量低于第一界限,例如不足内存总块数的1/4,则设置所有内存页的访问位A=0;如果剩余空闲块数量低于第二界限,例如不足内存总块数的1/8,则将内存中访问位A=0的页淘汰进入页缓冲区,具体做法是:置这些页的中断位P=0,再根据修改位M,把其中M=0的页加入页缓冲区的未修改页链表M_0,把M=1的页加入页缓冲区的修改页链表M_1,这里,只需把能够标识页的归属的进程号和页号等信息加入链表中,这些页并没有被淘汰。

在为新读入的页分配内存时,如果没有空闲块,则可以从页缓冲区中的M_0链表中移出一个页,将其对应的块分配给新的页;如果M_0链表为空,则从M_1链表中移出一个页淘汰,分配之前执行一个写I/O操作,或者一次性地把M_1链表的所有页写入磁盘,全部淘汰,回收作为空闲区。

在重定位发现访问页的中断位P=0时,则先在页缓冲区中M_0和M_1链表中查找,如果存在匹配的,则将其移出,并修改页表相关信息,如置P=1等,因为M_0和M_1链表中的页还没有被淘汰,所以不需要读I/O操作,可以直接从内存中得到页的信息。如果在页缓冲区M_0和M_1链表中都不存在当前要访问的页,则产生缺页中断。

VAXⅡ/VMS工作站操作系统(从UNIX发展而来的一个操作系统)采用了页缓冲算法。

5. 工作集模型

在请求分页存储管理中,程序运行时,只装入程序运行所需的基本页,其余页保留在外存中,那么,程序运行所需的基本页如何确定?在程序的所有页不能全部装入的情况下,是

不是装入越多越好？下面介绍工作集模型和缺页率。

1）工作集模型

一个进程的工作集(Working Set)是指该进程当前运行所需页的集合。如果一个进程的当前工作集的页都在内存中，那么它当前一段时间的运行不会产生缺页中断。但是，因为进程是动态变化的，所以，一个进程的工作集也在不断改变，随着进程的向前推进，原来工作集中的一些页会过时需要淘汰，并补充新的页。

工作集模型(Working Set Model)是指系统设置一个跟踪程序，检查每个进程的工作集，只有在一个进程的工作集在内存中后，才允许它运行。

工作集模型可以用于内存的分配，假定系统有 n 个进程，当前第 i 个进程的工作集页数为 ws_i，那么，n 个进程需要的内存块数

$$D=\sum_{i=1}^{n} ws_i$$

当 D 大于内存块总数时，采用对换技术，选择一些进程从内存调出到交换区，以保证内存中剩余进程的工作集都可以装入内存，这样可以减少频繁的缺页中断可能产生的抖动现象。

由于程序的运行流程难以事先预估，因此，工作集模型只能近似实现，即工作集保留进程近期经常访问的页的集合，跟踪程序定期检查工作集中的页，删除其中长时间没有被访问的页。

如何确定工作集的大小？如果一个进程的工作集太小，可能导致运行后有些页不在内存，而经常产生缺页中断；如果一个进程的工作集太大，要占用多余的存储空间，影响其他进程的运行。

一般地，系统为进程设置工作集大小的限制(上、下界限)，跟踪程序检查内存中的每个进程，如果它的当前工作集大于所限制的上限，则淘汰工作集中的一些页，缩小它的工作集。当一个运行的进程产生缺页中断时，如果它的工作集小于限制的下限，则可以申请一个空闲块，扩充它的工作集，否则，从它的工作集中选择一个页淘汰。如果当前空闲块总数比较多而进程数又较少，也可以适当增加一个进程的工作集的大小。

2）缺页率

如果一个进程的引用序列是 p_1、p_2……p_{n-1}、p_n，它在执行过程中，产生缺页中断的次数为 m，那么，该进程执行这个引用序列的缺页率定义为：

$$f=\frac{m}{n}\times 100\%$$

例如，在图 5-26 中，进程的缺页率＝22/27＝81%；在图 5-27 中，进程的缺页率＝17/27＝63%。

通常，对于一个进程，分配的内存块数越多，它在运行过程中产生的缺页率越小。如图 5-32 所示的曲线大致描述出缺页率与分配的内存块的关系。在曲线中，存在一个点，其横坐标是 w，当分配给进程的内存块数超过 w 时，每增加一个块，运行产生的缺页率减小不很明显，相反地，当分配给进程的内存块数小于 w 时，每减少一个块，运行

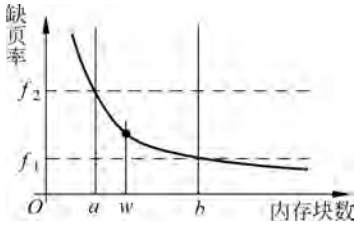

图 5-32 缺页率与内存块数的关系

产生的缺页率显著增加。这样,如果要控制进程的缺页率在 f_1 和 f_2 之间,那么,分配给进程的内存块数在 a 和 b 之间,这就为确定进程的工作集的大小限制(上、下界限)提供了依据。

图 5-32 描述的缺页率与分配的内存块数的关系对绝大多数进程而言都成立,即对于一个进程,分配的内存块数越多,它在运行过程中产生的缺页率越小。但是,对于 FIFO 算法存在个别进程,分配给内存的块数增加,缺页率没有减小,甚至反而增加,这种反常现象称为 Belady 现象。

如图 5-26 中的进程引用序列,分配的内存块分别是 2、3、4、5 和 6 时,经验算,运行产生的缺页中断次数分别是 26、20、22、11 和 11,缺页率分别是 96%、74%、81%、41%和 41%,如图 5-33 所示。

图 5-33 Belady 现象例子

图 5-33 描述的是 Belady 现象的一个例子,但是,随着内存块数的增加,缺页率曲线的总体趋势还是向下,只是在其中的个别局部范围表现出反常现象。

5.5.6 分页存储管理的主要特点

分页存储管理具有如下主要特点:

1. 非连续的存储分配,提高了存储空间的利用率

在分页存储管理中,页和块的长度相等,在程序装入时,内存中任一位置的空闲块都可能用来分配,提高了存储空间的利用率。

但是,由于程序虚拟地址空间的大小不一定是块长的整数倍,系统在为程序分页时,最后一个页往往不足块的长度,但装入内存时也占用一个块。因此,每个进程的最后一个页在装入内存后也存在与固定分区中类似的碎片,即内碎片。假定页长为 p,内存中的进程数为 n,那么,平均有 np/2 的内碎片的存储空间浪费。与分区相比,系统可以接受这种浪费。

2. 实现虚拟存储器

请求分页实现了虚拟存储器,使得系统可以运行比内存大的程序,或者在内存中装入尽可能多的程序,解除了程序员在程序设计时的额外负担,也增加了存储空间的利用率。

3. 页表占用额外的存储开销

在分页存储管理中,每个进程对应一个页表,且页表存放在内存中,这样,对于一个大进

程，它的页数可能很多。例如，在32位的CPU系统中，允许单个进程的虚拟地址空间大小为2^{32}，按块长为4K进行分页，则一个进程的页数多达2^{20}页，这样，页表占用的内存开销将非常严重；如果64位的CPU系统，那么，页表的内存开销将更大。

页表的存储开销，影响了分页存储管理的效率。如何解决这一问题？反向页表(Inverted Page Table)和多级页表(Multi-Level Page Table)是两种典型的可以减少页表的内存开销的存储管理技术。

1) 反向页表

引入反向页表的目的是减少内存页表的存储开销。反向页表曾在IBM、HP等一些工作站系统中采用，现在主要用在一些64位CPU系统中。

按照传统的方法，一个进程对应一个页表，当进程虚拟地址空间较大时，页表占用的内存空间的开销就特别严重，尤其是页表中有些页信息在进程运行过程中可能根本没有被访问。由于页表的主要作用是实现重定位，因此系统可以按照内存块统一建立一个表，称为反向页表，其结构由进程号(pid)和页号(p)组成，按块号顺序，每个块对应着反向页表中的一个表项，如果一个块是分配状态，则对应表项内容为(pid,p)，否则，表项内容为空。

基于反向页表的重定位过程如图5-34所示。

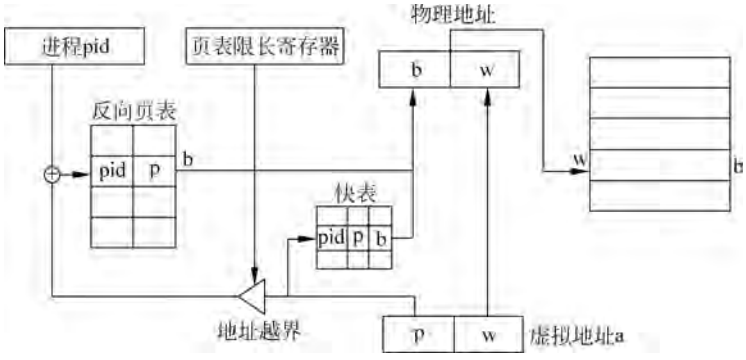

图5-34 反向页表的重定位

在基于反向页表的重定位中，需要按进程号pid和页号p查找反向页表的各表项。由于反向页表长度等于内存块的总数，因此，查找开销很大。那么，发挥TLB中快表的作用至关重要。另外，为了减少检索的开销，可以采用Hash算法，把进程号pid和页号p作为键值，建立与反向页表的映射关系。

2) 二级页表

对于多级页表技术，下面以二级页表为例，介绍二级页表结构设计和重定位。

二级页表是32位CPU系统采用的方法，主要解决大进程的页表保存在内存所带来的存储开销。在32位CPU系统中，块的长度通常是4KB，一个进程的虚拟地址空间最大可达2^{32}B，那么，一个进程的最大页数是2^{20}，页表的最大长度是2^{20}B，如果页表中每个表项的长度是4B，则页表的存储空间将达

$$4\text{B}\times 2^{20}=4\text{MB}$$

也就是，一个进程的页表可能占用4MB的存储空间，并且页表在内存中要求连续存储。

解决的一种方法是建立二级页表(Two Level Page Table)：页表和页目录表。其思想

是,将原来的一个进程的页表分成一些更小的部分,每个更小的部分称为页表(Page Table),其结构与扩充页表相同;进程运行时,并没有全部装入它的页表,而只装入当前需要的一部分页表,进程的其他页表保存在外存(磁盘)中或尚未建立。这样,在 MMU 中,虚拟地址的结构由目录号 d、页号 p 和页内地址 w 组成,如图 5-35 所示。

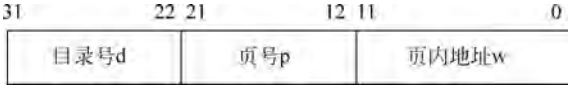

图 5-35 32 位 CPU 的虚拟地址结构

其中

块长$=2^{12}=4$(KB)

页表长度$=2^{10}=1$K

页目录表长度$=2^{10}=1$K

页表中每个表项的长度是 4B,这样一个页表的长度为

$$4\text{B}\times 1\text{K}=4\text{KB}$$

正好等于块长度。

一个进程的页目录表(Page Directory Table,PDT)用于管理该进程的页表及其装入内存的情况,页目录表 PDT 的结构与页表的结构类似,且长度相等,两者的关系如图 5-36 所示。

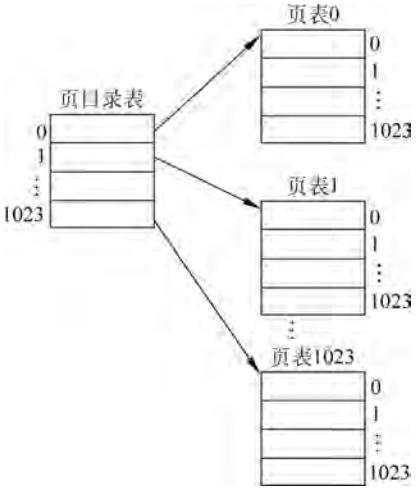

图 5-36 一个进程的页目录表及其页表的关系

每个进程都对应一个页目录表,依次登记该进程的页表状态信息。与页表一样,一个进程的页目录表 PDT 共有 1024 个表项,每个表项占 4B,所以一个页目录表共 4KB,正好占用一个内存块。

如图 5-37 所示,描述了二级页表的重定位过程。从虚拟地址寄存器得到当前访问的虚拟地址所在的目录号 d、页号 p 和页内地址 w;如果页号 p 在 TLB 快表中,则直接形成物理地址,否则,从页目录基址寄存器中得到页目录表。

根据目录号 d 检查页目录表,如果对应的页表在内存中,则根据页号 p 检查页表,如果

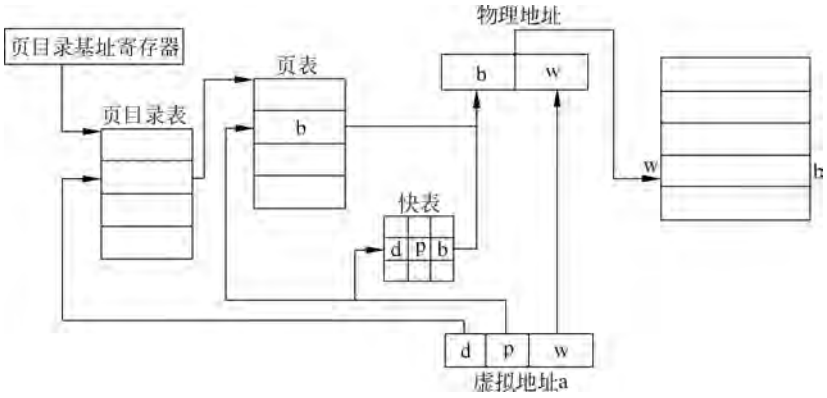

图 5-37　二级页表的重定位

页号 p 的中断位 P=1，则该页在内存中，从页表中得到块号，形成物理地址；如果页号 p 的中断位 P=0，产生缺页中断。如果目录表中目录号 d 对应的页表不在内存中，则建立或读取对应的页表。

除了二级页表，有的系统还采用三级、四级页表等多级页表技术，如 32 位 SPARC 处理器采用三级页表，Motorola 68000 处理器采用四级页表。

4. 分页破坏了程序的完整性

程序员设计的源程序具有良好的结构，如数据段、代码段、堆栈段等，代码段又由许多程序员精心设计的模块组成。在分页存储管理中，源程序经过编译、链接装入内存后，程序的这些结构信息被统一编号，形成从 0 开始编号的一组连续的虚拟地址，构成一维地址空间，如图 5-38 所示。

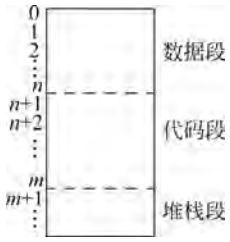

图 5-38　一维地址空间的例子

装入程序时，操作系统按块长将程序的虚拟地址空间进行分页，页长度由系统硬件决定，因此程序员看不出一个变量或指令在哪一个页。这样造成一个页的信息不完整，例如，一个页可能既有数据段信息也有代码段的信息，或者代码段中的一个模块可能被分成多个页，或者一个页包含代码段中的多个模块。

所以，分页破坏了程序的完整性，这给程序的共享、动态链接等技术的实现带来了困难。

5. 请求分页存在抖动现象，降低 CPU 的利用率

请求分页需要复杂硬件的技术，如 TLB 与页表存储一致性保证等。另外，置换算法增加了系统开销，同时可能存在抖动现象。

本节最后简要讨论页长度与系统开销的关系。

页长度 p 越大，对于一个进程，分页后的页数越少，因而可以节省页表的存储开销，减少缺页中断的次数。但是，在每次缺页中断时的 I/O 操作中读或写的信息增加，使得内、外存的调入/调出的时间延长。另外，每个进程的平均内碎片长度 $p/2$ 也越大。

相反地，页长度 p 越小，对于一个进程，分页后的页数越多，从而增加了页表的存储开销，并且可能增加缺页中断的次数。但是，在每次缺页中断时的 I/O 操作中读或写的信息

减少,使得内、外存的调入/调出的时间缩短。另外,页长度 p 越小,每个进程的平均内碎片长度 $p/2$ 也越小。

以内存开销为例,假设页长度为 p,进程的大小为 s,页表中每个表项的长度为 e,那么,一个进程的内存的开销与页长度的关系为

$$f(p)=es/p+p/2$$

其中 es/p 为进程的页表开销,$p/2$ 是进程的内碎片开销。

求函数 $f(p)$的极值,令 $f'(p)=0$,即

$$-es/p^2+1/2=0$$

解得

$$p=\sqrt{2es}$$

即 $p=\sqrt{2es}$ 是函数 $f(p)$的一个极小值,且当 $p=\sqrt{2es}$ 时,$f(p)=\sqrt{2es}$。

函数 $f(p)$的曲线如图 5-39 所示。可见,当 $p=\sqrt{2es}$ 时函数 $f(p)$的值最小。

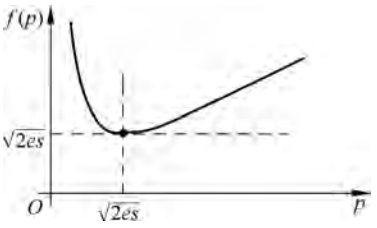

图 5-39 内存开销与页长度的关系

假定进程的平均大小 $s=2048$K,那么,当 $e=2$ 时,页长 p 约为 3K;当 $e=4$ 时,$p=4$K;当 $e=6$,则页长 p 约为 5K。

5.6 分段存储管理

在分页存储管理中,由于一道程序的虚拟地址是连续的,因而无法在两个相邻的虚拟地址之间加入一些存储单元,导致程序运行过程受到很多限制。例如,在图 5-38 中,程序运行时它的数据段中的存储单元,在使用到第 n 个存储单元后,就不能再增加新的存储单元,因为第 $n+1$ 个单元可能是代码段的指令信息;同样,代码段中在使用到第 m 个存储单元时,就不能再使用第 $m+1$ 的存储单元,因它可能存储堆栈的信息。这意味着程序运行时,某个子程序的递归调用的层次、指针变量的存储分配等受到限制。所以,如果为程序的数据段、代码段、堆栈段等建立独立的虚拟地址空间,上述限制将得到一定程度的解决。

另外,对进程分页后,一个页可能既有数据段信息也有代码段信息,或者,代码段中的一个模块可能分成多个页,一个页包含多个代码段中的几个模块,破坏了程序结构信息的完整性,不便于实现程序的共享、动态链接等技术。

还有,缺乏有效的存储保护。由于在分页存储管理中,没有区分数据段和代码段的存储空间,所以,难以防止把数据段的信息作为代码段,给处理器运行带来种种错误。

针对上述问题,人们提出了分段存储管理方法。

5.6.1 基本思想

现在的计算机系统,在硬件上处理器都支持分段(Segmentation)存储管理。分段存储管理的基本思想如下:

1. 程序“分段”

程序由若干在逻辑上具有完整独立意义的单位组成，每个单位称为段(Segment)，系统在程序链接并装入内存后，为各个段的信息建立独立的虚拟地址空间。每个段对应一个段号，一个段的虚拟地址空间从0开始连续编号，如图5-40所示。

图5-40 程序分段的例子

2. 内存动态分区

内存空间的分配采用动态分区。操作系统启动成功后，整个用户区作为一个空闲区，一般地，在程序装入时，按照程序的段来分配内存，系统根据段的实际需求量，查找一个合适的空闲区，如果该空闲区长度等于段的需求量，就可以直接分配，否则，将其分成两个分区，其中一个分区长度正好等于当前段的需求量，并分配给它，另一个分区作为空闲区保留下来。进程运行结束被撤销后，每个段占用的分区回收成为空闲区。

这与第5.4节中的可变分区类似，只是在可变分区管理中，以整个程序为单位分配内存区域，而这里是以程序中的段来分配内存区域。所以，与可变分区相比，分段存储管理减小了分配单位的粒度。

3. 非连续存储分配

一道程序通常由多个段组成，在这些段装入内存后，不同段所占用的分区之间，不要求是连续的，即同一个段的信息在内存中连续存储，但不同段之间的信息在内存中可以不连续。

4. 内、外存统一管理实现虚拟

程序装入时，可以根据空闲区的状况，只装入运行所需的基本段，其余段保留在外存中，以后运行过程中再设法将外存中的段装入内存，即提供虚拟存储管理。

*5.6.2 硬件基础

随着硬件技术的快速发展，处理器的系统结构不断创新和改进，处理器为操作系统的设计和实现提供了更加方便、灵活的方法。为了帮助理解存储器管理的思想，本节补充介绍Intel 32位处理器(简称IA-32)的系统级寄存器及其数据结构，主要包括基本寄存器、描述符和处理器操作模式，另外与任务有关的硬件基础，将在第7.2.2小节进一步介绍。

1. 基本寄存器

IA-32处理器中的基本寄存器有以下几种：

1) 8个32位通用寄存器

8个32位通用寄存器是EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP。

8个32位寄存器的使用方式如图5-41所示。为兼容原来的16位程序，可以按16位方式使用，其中，寄存器EAX、EBX、ECX、EDX的低16位中还可以按8位使用。

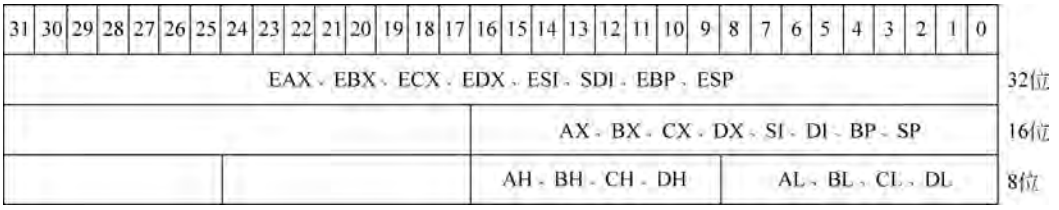

图 5-41 8 个 32 位通用寄存器的使用方式

2) 6 个 16 位段寄存器

6 个 16 位段寄存器是 CS、DS、ES、FS、GS、SS。

其中,CS 为代码段基地寄存器,初始值为 F000H; DS 为数据段基址寄存器; ES、FS、GS 为附加数据段基址寄存器,初始值为 0; SS 为堆栈段基址寄存器,初始值也是 0。

关于段寄存器将在后面做进一步介绍。

3) 32 位标志寄存器 EFLAGS

32 位标志寄存器 EFLAGS 用于跟踪指令执行状态、存放比较指令的结果和控制指令执行。其结构如图 5-42 所示。

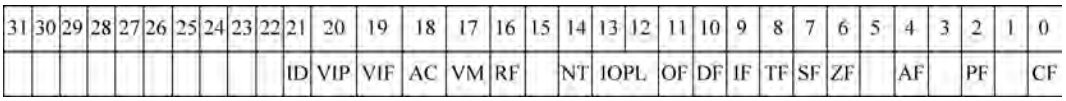

图 5-42 32 位标志寄存器的结构

EFLAGS 寄存器的初始值为 00000002H,比较、运算等操作将影响 EFLAGS 寄存器,条件转移指令根据 EFLAGS 寄存器的对应位的状态决定是否跳转。其中,EFLAGS[12:13]的 IOPL 表示 I/O 特权级(Privilege Levels)。IA-32 设置了 4 个级别的特权级,从高到低为 0、1、2、3,在保护模式下有效,对 I/O 端口、主存储器和处理器切换等进行特权级保护。这里,当进程的当前特权级 CPL≤IOPL 时,才能执行 IN/OUT 指令。操作系统内核运行在特权级 0,用户程序运行在特权级 3。EFLAGS[17]是虚拟 8086 标识,置 1 时进入虚拟 8086 模式,清零时退出。

4) 32 位指令指针寄存器 EIP

32 位指令指针寄存器 EIP 保存将要执行的下一条指令地址,即保存下一条指令在代码段中的偏移量,EIP 寄存器由系统译码器按指令顺序自动修改,通常不允许软件直接读或写 EIP 寄存器。

5) 5 个 32 位控制寄存器

在 IA-32 处理器中,有 5 个 32 位控制寄存器: CR0、CR1、CR2、CR3 和 CR4,如图 5-43 所示。

控制寄存器用于设置处理器的工作模式和存储分页等。对于 5 个控制寄存器,应用程序可以执行读操作,但不允许执行写操作; 只有在特权级 0 下,才可以执行写操作。其中:

(1) CR0.PE 表示保护模式允许,PE=1 系统启动后进入保护模式; PE=0 系统回到实地址模式。

(2) CR0.PG 表示是分页机制允许,PG=1 允许分页,PG=0 禁止分页。

图 5-43 5 个控制寄存器的结构

(3) CR0 的初始值为 60000010H。

(4) CR1 没有定义。

在 CR0.PG=1 时,CR2 有效,在请求分页中,CR2 存放产生缺页中断的线性地址。

在 CR4.PAE=0 时,CR3.PDBR 有效,占 CR3 中的[31:12]20 位,[11:0] 默认为 0,CR3.PDBR 保存当前进程的页目录表的物理地址(块号)。

而当 CR4.PAE=1 时,CR3.PDPTR 有效。这时物理地址为扩展的 36 位,需要采用三级页表,占 CR3 中的[31:5],[4:0]默认为 0,CR3.PDPTR 保存页目录表指针。

6) 4 个存储器地址寄存器

IA-32 处理器有 4 个寄存器用于存储器寻址,它们是全局描述符表寄存器 GDTR、局部描述符表寄存器 LDTR、中断描述符表寄存器 IDTR 和任务寄存器 TR。

IA-32 处理器的 6 个段寄存器的作用不仅仅在于表示段的地址,在保护模式下,每个段寄存器作为指针,指向表示描述段的类型、基址、限长及其他属性的存储区。段寄存器的结构如图 5-44 所示。

byte9	byte8	byte7	byte6	byte5	byte4	byte3	byte2	byte1	byte0	段寄存器
代码段描述符缓冲区								段选择符		CS
数据段描述符缓冲区								段选择符		DS、ES、FS、GS、SS

图 5-44 段寄存器的结构

其中,低 16 位即[byte1:byte0]保存段选择符(Segment Selector),软件可以访问。另外的 64 位[byte9:byte2]保存段描述符(Segment Descriptor),由硬件存取,对软件是透明的。

16 位段选择符的结构如图 5-45 所示。

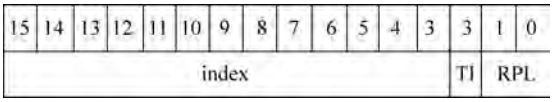

图 5-45 16 位段选择符的结构

其中：

(1) [1:0]的 RPL(Request Privilege Level)保存请求特权级，表示进程对段访问的请求权限，通常等于主程序代码段的特权级，在操作系统的进程管理中设置。

(2) TI(Table Indicator)为描述符表选择位，TI=0 表示全局描述符表(GDT)，TI=1 表示局部描述符表(LDT)。

(3) [15:3]的 index 为描述符表索引，即当前访问的段的描述符在描述符表(GDT/LDT)中的索引值。当前段的描述符地址=描述符表基址+8×index。

一个段描述符表是段描述符的一个数组，每个段描述符由 8 字节共 64 位组成，其结构如图 5-46 所示。主要描述段的 32 位基址(Base)、20 位限长(Limit)、特权级(DPL)、中断位 P、类型(Type)、是否系统数据 S 等信息。

图 5-46 段描述符的结构

其中，中断位 P=1 表示段信息在内存中，P=0 表示段信息在外存中；S=0 表示系统描述符，S=1 表示用户描述符；G 表示段长单位，G=1 为 4KB，G=0 为 1B；D/B 表示段长属性，D/B=1 表示 32 位，D/B=0 表示 16 位；AVL 标志位可以给系统软件使用，临时存放数据，或写入某些特殊意义的值。

类型 Type 占用 64 位段描述符中[43:40]的 4 个位，分为以下 3 种情况：

(1) S=1 且 Type[43]=0：表示用户数据段，此时 Type[42]为扩展位 E(Expansion Direction)，E=0 不可扩展，E=1 可以向下扩展；Type[41]为可写位 W(Writable)，W=0 不许写入，W=1 可以写入；Type[40]为访问位 A(Accessed)，表示上次清零后，是否被访问，A=1 表示被访问。

(2) S=1 且 Type[43]=1：表示用户代码段，此时 Type[42]为一致性位 C(Conforming)，用于控制段的切换，C=1 为一致性代码段，C=0 为非一致性代码段；如果执行控制转移向一致性代码(C=1)时，要求当前特权级 CPL 的数值大于或等于所转移的代码段特权级 DPL 的数值，即可以运行比当前特权级别低的代码，但转移向非一致性代码时，将需要保护控制。Type[41]为可读位 R(Readable)，R=0 不许读，W=1 可读；Type[40]为访问位 A(Accessed)，表示上次清零后是否被访问，A=1 表示被访问。

(3) 当 S=0 时，表示系统描述符，其结构及含义请参见第 7.2.2 小节关于任务状态段描述符和门描述符的结构介绍。

2. 描述符

在 IA-32 中，除了段描述符之外，还有门描述符，通过门描述符实现对系统程序和中断处理程序等运行的保护控制。

一般，描述符(Descriptor)分为段描述符和门描述符两个对象和用户描述符(User)和系统(System)描述符两种级别，如图 5-47 所示。

图 5-47 描述符的组成

描述符保存在描述符表(Descriptor Table)中，描述符表是描述符的数组，每个描述符由 8 个字节组成。系统有 3 个描述符表：全局描述符表 GDT(Global Descriptor Table)、局部描述符表 LDT(Local Descriptor Table)和中断描述符表 IDT(Interrupt Descriptor Table)。

3 个描述符表所存储的描述符如表 5-4 所示。

表 5-4 描述符表存储的描述符

描 述 符	描述符表		
	GDT	LDT	IDT
代码段描述符	√	√	
数据段描述符	√	√	
LDT 描述符	√		
TSS 描述符	√		
任务门描述符	√	√	
调用门描述符	√	√	√
中断门描述符			√
自陷门描述符			√

注：√表示存在存储关系。

GDT、LDT、IDT 和 TSS 是 IA-32 的系统数据结构，是操作系统的存储器管理和进程管理的基础。

GDT 和 IDT 是系统运行最基本的数据结构。GDT 保存在全局描述符表寄存器 GDTR 中，IDT 保存在中断描述符表寄存器 IDTR 中。在系统中，GDT 和 IDT 是唯一的，但是 LDT

和 TSS 可以有多个。

在系统启动过程中,GDT 和 IDT 由操作系统建立并初始化。例如,在建立 GDT 时,至少需要创建系统数据段、系统代码段和首个任务状态段描述符。一旦初始化完成,通过专门的指令(LGDT 和 LIDT)将表的基址和限长装入(加载)寄存器 GDTR 和 IDTR。在系统的整个运行过程中,GDT 和 IDT 的表地址是固定的,只能修改内容。

在分段存储管理中,每个进程需要建立一个 LDT,用于存放进程相关的代码段、数据段的描述符。系统需要一个专门的数据段用于保存进程的 LDT,这个数据段的描述符保存在 GDT 中。局部描述符表寄存器 LDTR 保存当前进程的 LDT,在处理器切换时,LDTR 自动加载。

任务状态段(TSS)描述符用于设置任务(进程)运行的初始环境和状态。在进程调度时,处理器先在 TSS 中保存当前进程的环境状态,然后,将调度程序选中的进程的 TSS 选择符装入寄存器 TR,并以新的 TSS 加载处理器的寄存器,之后可以开始运行选中的进程。

关于任务状态段描述符和门描述符,以及处理器切换时的特权级保护,请参看第 7.2.2 小节。

IA-32 处理器的 4 个系统寄存器的结构如图 5-48 所示。

byte9	byte8	byte7	byte6	byte5	byte4	byte3	byte2	byte1	byte0	寄存器
				全局描述符表起始地址				限长		GDTR
				中断描述符表起始地址				限长		IDTR
局部描述符表描述符缓冲区								选择符		LDTR
任务状态段描述符缓冲区								选择符		TR

图 5-48 4 个系统地址寄存器的结构

在访问内存的一个存储单元时,进程的指令或数据由逻辑地址(Logical Address)表示,逻辑地址由一个 16 位段选择符和一个 32 位偏移量组成,根据段选择符,从 GDT 或 LDT 中得到段的基址,再加上偏移量,得到存储单元的线性地址。如果系统是分段存储管理,则线性地址直接映射为物理地址;如果是分页存储管理,则从线性地址(即虚拟地址)得到页号和页内地址,再进一步重定位得到物理地址。

3. 处理器操作模式

IA-32 支持 4 种操作模式:实地址模式、保护模式、虚拟 8086 模式和系统管理模式。

1) 实地址模式

系统启动后,IA-32 处理器处于实地址模式(Real-address Mode),此时处理器支持扩展 20 位段地址和 16 位段内地址,段地址的高 16 位由段寄存器定义,低 4 位为 0。在存储器寻址时,处理器先把段寄存器左移 4 位,然后加上 16 位的段内地址,最终形成 20 位的地址。因此,在实地址模式中,寻址空间为 2^{20} 即 1MB,采用分段方式,每段最大长度为 2^{16} 即 64KB。

实地址模式完全仿真 Intel 8086 处理器的编程和运行环境。IA-32 在实地址模式下基本只能运行 Intel 8086 指令集,即 16 位处理器,操作数默认为 16 位,但通过使用指令前缀,

能够使用 32 位地址或 32 位操作数。

在实地址模式中，用户程序和操作系统都运行在特权级 0。

2) 保护模式

IA-32 处理器通常是在保护模式下运行 32 位程序。所谓保护模式(Protected Mode)就是在多进程并发执行时，对不同进程的虚拟地址空间进行完全的隔离，并通过定义一组特权级实现保护机制，保护每个进程的单独运行。系统启动后先进入实地址模式，完成系统的初始化后，通过软件把控制寄存器 CR0 的位 PE 置 1，处理器就进入保护模式。

在保护模式中，内存采用分段机制。把处理器可访问的地址空间称为线性地址空间(Linear Address Space)，把线性地址空间分为若干更小的可保护的地址空间，称为段(Segment)，段可用于保存程序的代码、数据、堆栈或系统数据(如 TSS、LDT 等)。每道程序拥有各自独立的一组段，每个段对应一个段描述符，用于描述段的基址、大小、类型、访问权限、特权级等。通过逻辑地址访问段中的指令或数据。逻辑地址的段选择符指示的段基址加上逻辑地址的偏移量得线性地址。

操作系统可以选择如下的基本模型实现存储器管理：

基本平面模型(Basic Flat Model)：在处理器保护模型下，最简单的存储管理模型是基本平面模型。在这种模型中，操作系统内核和应用程序的代码段、数据段和堆栈段的基址均置为 0，段的限长为 4G，这样，每道程序的各段共享一个一维的地址空间，隐藏了内存段的概念，无法利用硬件的段保护机制。

保护平面模型(Protected Flat Model)：在基本平面模型的基础上，实现物理地址的越界保护和特权级保护。例如，操作系统采用分页管理(CR0. PG＝1)，并支持二级页表，CR3. PDBR 保存当前进程的页目录表基地(所在块号)；把 2^{32}(4G)的虚拟地址空间分为系统空间和用户空间，系统空间分配给操作系统内核，对应特权级 0，应用程序运行在用户空间，对应特权级 3。在用户空间，每道程序的各段共享一个一维的地址空间，即各段的基址相等。

现代操作系统多数都采用保护平面模型。

多段模型(Multi-Segment Model)。在程序装入时，操作系统根据程序的段实际大小分配存储空间，并为每个进程设置一个 LDT，每个段拥有独立的段描述符，通过段描述符中的基址和限长实现存储区域的隔离，能够充分利用硬件的段保护机制。当 CR0. PG＝1 时实现段页式存储管理。

在分段模型中，一个段最大长度是 2^{32}(4G)，在图 5-45 所示的段选择符中，index 占 13 位，加上 TI 位，段选择符可以表示 2^{14} 个段，因此，虚拟地址空间为 2^{46}(64T)。

3) 虚拟 8086 模式

虚拟 8086 模式(Virtual-8086 Mode)是模拟实地址模式，提供在保护模式下运行原有的 Intel 8086 软件要求的硬件平台。在保护模式下，标志寄存器 EFLAGS. VM＝1 时，处理器进入虚拟 8086 模式。

与实地址模式不同的是，虚拟 8086 模式可调用保护模式的 32 位中断处理程序，能够与保护模式进程并行，且可以利用分页机制。

4) 系统管理模式

系统管理模式(System Management Mode，SMM)用于实现电源管理、系统诊断、配置

即插即用(PnP)设备等。当硬件设置系统管理中断(SMI)的引线 SMI#,或软件从高级可编程中断控制器(APIC)触发系统管理中断 SMI 时,处理器进入系统管理模式,SMI 处理程序通过调用 RSM 指令返回原操作模式。

另外,IA-32 处理器还有 IA-32 扩展模式(IA-32e Mode)。在 IA-32 扩展模式中,处理器支持两种工作模式:兼容模式和 64 位模式。兼容模式支持 16 位和 32 位保护模式。

5.6.3 实现关键

下面介绍分段存储管理的数据结构设计、存储空间的分配和回收、重定位和存储保护。

1. 数据结构设计

分段存储管理的内存分配采用与可变分区类似的分区管理,因此,内存中的空闲区可以采用可变分区的数据结构,如空闲区链表等。

在一道程序装入后,需要专门的数据结构来登记它的每个段的信息以及各段在内存中的存放分区位置等,把这种数据结构称为段表。

段表的结构由段号、段长度、中断位 P、分区起始地址、外存地址、存取控制信息、访问位 A 和修改位 M 等组成。其中,中断位 P、访问位 A 和修改位 M 与请求分页中的含义和作用相同,存取控制信息主要包括不可访问、只读、只写、可读可写、可执行等访问权限。

每个进程都对应一个段表。通常,段表存储在内存的系统区中,作为内核的关键数据结构之一。

2. 存储空间的分配和回收

与可变分区的存储分配一样,在装入程序的一个段时,根据段的实际大小,查找一个合适的空闲区,如果空闲区长度等于段的大小,则直接分配;否则,将空闲区分成两部分,一部分等于段的实际需求量并分配给要装入段,另一部分作为更小的空闲区保留下来。

在查找合适的空闲区时,可以采用 FF、BF 或 WF 等分配策略。

当一个进程撤销时,回收其各段所占用的分区,回收时需要合并相邻的空闲区。

3. 重定位和存储保护

在分段存储管理中,虚拟地址是二维的,每个虚拟地址由段号 s 和段内地址 d 组成。根据段号查段表,如果段内地址大于或等于段长则产生地址越界中断;如果对应的中断位 P=1,则该段已经在内存中,把段表中对应的分区起始地址加上段内地址 d,得到物理地址;如果对应的中断位 P=0,则说明该段没有装入内存,在存储保护检查符合访问操作时,产生缺段中断,从外存读入段的信息,在存储保护检查不符合访问操作时会产生一个异常,禁止访问。

操作系统在缺段中断处理过程中,首先要为新的段查找一个合适的空闲区,如果没有满足段要求的空闲区,则运行置换算法,从内存中选择一个或几个段将其淘汰,在淘汰内存中的一个段时,检查修改位 M,如果 M=1,则需要另外的写 I/O 操作,将选中的段写到外存;再建立一个与段长度相等的空闲区;接着,启动读 I/O 操作将该段从外存中读入内存,修改段表的信息;重新执行引起缺段中断的指令。

由于各段的长度往往不相等，所以与请求分页相比，分段存储管理中的置换算法比较复杂，例如，需要考虑是淘汰一个大的段，还是淘汰几个小的段等。

下面以 IA-32 为例介绍存储保护。操作系统在建立段表后，存储保护由硬件实现。

存储保护主要禁止进程对存储器的非法访问，如向代码段执行写操作、存取段范围以外的存储单元等。存储保护分为两个阶段：一是在修改段寄存器时的保护，二是重定位过程的保护。

修改段寄存器是通过修改段寄存器中的段选择符实现的，首先检查段选择符(3～15)中的 index，是否超过对应描述符表(TI＝0 时 GDT，TI＝1 时 LDT)的长度。

在分段存储管理中，虚拟地址是二维的，每个虚拟地址由段号 s 和段内地址 d 组成。在重定位过程中，要求段内地址 d 小于对应的段描述符中的限长(见图 5-46)。

4. 段的共享

段是一个在逻辑上具有完整意义的独立单位，为实现进程的共享提供了方便。

段共享的具体方法是：系统维护一个共享段表，存放可供多个进程共享的段信息，共享段表的结构与段表相似。当一个进程需要一个共享段时，在其段表中添加一个表项，填写来自共享段表中所需要的段信息。

需要指出，在几个进程共享一个代码段时，这个代码段通常要求是纯代码的(Pure Code)，即代码在执行过程中自身不会被修改，具有这种特点的代码也称可重入代码。几个进程在共享可重入代码时，要求每个进程建立各自的数据段，作为共享代码的工作区，用于可共享代码执行时的数据处理及保存处理结果。

例如，如图 5-49 所示，某多用户系统提供 C 语言编译器(Compiler)，供多个用户共享，这样，操作系统在共享段表中建立一个 C 语言编译器的段，内存中建立一个分区存放 C 编译器程序，有 3 个用户的 C 语言源程序文件：proc1.c、proc2.c 和 proc3.c，需要运行 C 编译器为其编译，那么，当 3 个用户分别运行 C 编译器时，每个进程至少需要两个段，其中一个段登记共享段信息，指向内存中的共享段即 C 编译器程序段，另一个段是数据段，存储各自的源程序文件。

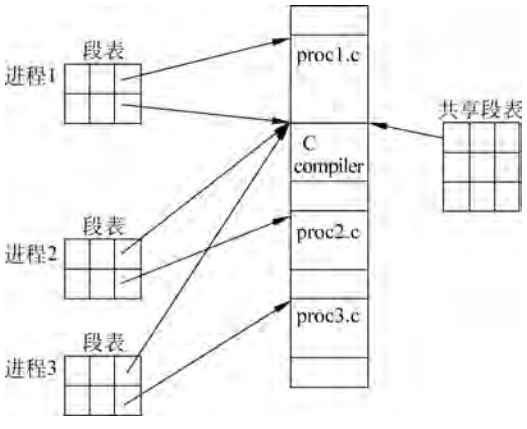

图 5-49 共享段的例子

5.6.4 分段与分页的区别

分段与分页的区别主要包括以下几方面：

1. 存储空间的分配单位粒度

在分页存储管理中以页为单位分配内存空间，页由硬件虚拟地址结构决定，页长度是固定的；在分段存储管理中，以段为单位分配内存空间，段由程序员的程序设计决定，段之间的长度往往不相等。

2. 虚拟地址空间的维数

在分页存储管理中，虚拟地址空间是一维的；在分段存储管理中，虚拟地址空间是二维的。

3. 内存分配

分页存储管理中把内存空间看成由一组大小相等的块组成；分段存储管理则采用动态分区。

4. 碎片

在分页存储管理中，每个进程的最后一个页可能不足一个块的长度，按页分配内存块时，存在内碎片；分段存储管理则采用动态分区，随着分配和回收的不断进行，可能存在很小的空闲区，造成外碎片。

5.6.5 主要特点

分段存储管理能够保持程序运行时的完整性，可以实现段的动态扩充、段的共享和动态链接等，另外，在硬件方面提供了基于特权级的存储保护机制，增加了系统的安全性。

所谓动态链接，是指源程序中需要访问的模块以独立的程序文件形式保存在外存中，这些独立的程序文件称为动态链接库文件。动态链接库文件可以由程序员设计编写，也可以是第三方提供的符合规定要求的库函数文件；源程序在编译、链接后的可执行程序与它所需要的动态链接库文件分别单独保存；在程序运行过程中，需要用到动态链接文件的模块时，再从外存中对应的动态链接库文件将其装入，并链接成为原进程地址空间的一部分，然后再运行。

与动态链接相对应的是静态链接。静态链接是把程序中所需要的模块代码(除系统调用外)，在链接时就全部加入一个可执行程序文件中。静态链接实现简单，但使得单一的可执行程序文件可能很大，不仅在装入内存时占用更多的存储空间，而且程序运行过程中可能有些模块根本没有被访问，例如，一些错误处理模块，或者一些人机交互过程用户没有选择的操作模块，等等，这些可能没有被访问的模块也要链接在一起装入内存，造成存储空间的浪费。

动态链接技术具有如下优点：

1. 增加程序的可维护性

在静态链接中，所有模块在链接时形成一个单一的可执行文件，如果库文件的模块需要更新或修改，则模块修改后，可执行程序需要重新链接。程序链接的工作需要软件专业人员才能完成。而在动态链接中，如果修改的只是动态链接文件中的模块，则只需重新编译动态链接文件，原来的可执行程序无须重新链接，只要能得到新的动态链接库文件即可，因此增加了程序的可维护性。

2. 实现进程共享一个外存中的程序模块

外存中的动态链接库文件可以供多个进程共享，这些进程在运行过程需要时从外存同一位置库文件中链接目标模块，这样不仅增加了可维护性，同时节省了外存空间。

分段存储管理存在以下一些不足：

首先，段的连续分配，降低了存储空间的利用率。因为同一段的信息要求连续存储，所以，在以段为单位分配内存时，对于一个大的程序段，需要找出一个足够长度的空闲区，因而存在外碎片。

其次，在分配和回收时，增加了系统开销。在装入一个段时，需要采取分配策略查找合适的空闲区，回收时需要相邻空闲区的合并，这些操作都增加了系统的开销。

还有，置换算法更为复杂，可能存在抖动现象。

5.7 段页式存储管理

分页和分段存储管理各有其优缺点，如果把两者结合起来，就可以发挥它们的优点，因此提出段页式存储管理。

5.7.1 基本思想

段页式存储管理的基本思想如下：

1. 内存分块

采用分页管理中的内存分块思想，把内存看成一系列固定长度的块组成，每个块对应一个块号。这与分页存储管理中的内存分块一样。

2. 程序分段

采用分段管理中的程序分段思想，程序由若干在逻辑上具有完整独立意义的单位组成，每个单位称为段，系统在链接程序时，为各个段的信息建立独立的虚拟地址空间。每个段对应一个段号，一个段的虚拟地址空间从 0 开始连续编号。

3. 段分页

类似于分页管理中的进程分页思想。但在这里，分页是针对程序中的段进行，在装入程

序的一个段时,把该段的虚拟地址空间按块的长度分成页,并按虚拟地址的顺序依次为每个页编号,页号为0、1、2……由于段的长度不一定是块长度的整数倍,每个段在分页后,最后一个页可能不足一个块的长度,但也按一个页处理。

4. 非连续的分配

以页为单位分配内存块,同一个段的几个相邻的页在内存中不要求占用相邻的内存块,也就是说,同一个页的程序信息在内存中是连续存放,但不同页之间内存中的程序信息可以是不连续的。

5. 实现虚拟存储器

在装入一个程序时,可以只装入一部分段的基本页,其余段和页保留在外存中,运行过程中再装入外存中的段或页。因此,在段页式存储管理中,可以运行比内存大的程序,或者在内存中装入尽可能多的程序,实现了虚拟存储器。

如图5-50所示,程序A有3个段:段0、段1和段2,分别分成3个页、2个页和4个页,其中段0前两个页已经装入内存中块号为120和122的块上,段1未装入内存,而段2的前3个页分别装入块号为835、836和633的块上。

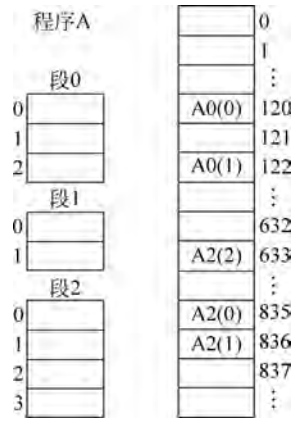

图5-50 段页式存储管理例子

5.7.2 实现关键

下面介绍段页式管理中实现时的数据结构、重定位及缺页中断。

1. 数据结构设计

管理内存块使用状况的数据结构,可以采用分页存储管理中的位示图或空闲块链表。

每个进程由多个段组成,每个段又分若干页。用于管理进程的段信息的数据结构称为段表;用于管理一个段的页信息的数据结构称为段页表,简称页表。

1) 段表

每个进程对应一个段表,段表的结构由段号、段长、中断位P、段页表基址以及其他的存取控制信息等组成,其中,中断位P表示段页表是否建立,P=0表示未建立,P=1表示已经建立;段长是段的虚拟地址空间大小(也可以是页数或段页表长度)。

2) 段页表

进程的每一个段都对应一个段页表,段页表与请求分页中扩充页表的结构相同,主要由页号、块号、中断位P、访问位A、修改位M、外存地址等组成。

一个进程的段表和段页表的关系如图5-51所示,它们的建立和初始化过程如下:在程序装入时,根据程序的段数目,建立一个段表,依次填入段号、段长(页数),中断位P=0;在为一个段装入一个页时,如果段表中该段的中断位P=0,则根据该段的页数,建立一个页表,并将页表的起始地址填入段表中对应的页表基址上,置P=1;如果段表中该段的中断位P=1,则从段表中对应的页表基址中得到该段的页表;为装入的页分配一个空闲块,并设置页表中的相应信息。

进程A的段表

段号	段长	中断位P	…	页表基址
0	3	1		
1	2	0		
2	4	1		

段0的页表

页号	块号	中断位P	访问位A	修改位M	…	外存地址
0	120	1	0	0		XX0
1	122	1	0	0		XX1
2		0				XX2

段2的页表

页号	块号	中断位P	访问位A	修改位M	…	外存地址
0	835	1	0	1		YY0
1	836	1	1	0		YY1
2	633	1	0	0		YY2
3		0				YY3

图 5-51　进程的段表与段页表关系的例子

2. 重定位

在段页式管理中，虚拟地址是二维的，由段号 s 和段内地址 d 组成。

重定位过程如图 5-52 所示。从段内地址 d 得到页号 p 和页内地址 w，首先检查快表，如果段号 s 的页号 p 在快表中则直接形成物理地址；如果不在快表中，则当段号 s 大于或等于段表限长寄存器时产生地址越界中断，在段号 s 小于段表限长寄存器时，访问段表，如果段表中段号 s 的中断位 P＝0，则产生缺段中断；在中断位 P＝1 时，如果页号 p 小于段长（页数），则从段表中的页表基地访问页表，如果页表中页号 p 对应的中断位 P＝1，则得到 b，形成物理地址，否则，产生缺页中断；如果页号 p 大于或等于段长（页数）则产生地址越界中断。

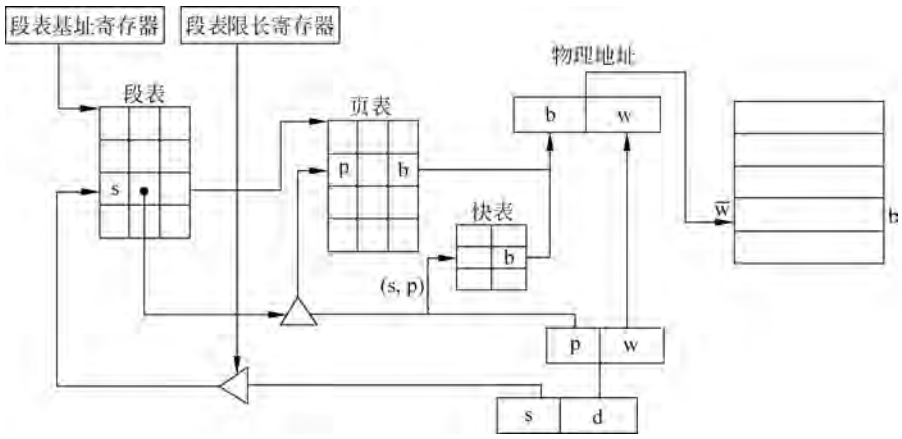

图 5-52　段页式存储管理的重定位过程

3. 缺页中断

段页式存储管理中的缺页中断处理与请求分页的缺页中断处理相同。

段页式存储管理的主要特点：综合了分页和分段存储管理的思想，发挥了两者的优点，但需要更复杂的系统，增加了系统开销，例如，重定位过程可能多次访问内存。

5.8 本章小结

多道程序设计就是在内存中同时存放多个程序,而计算机系统的内存相对较小,操作系统的存储管理目的是提高存储空间的利用率和方便程序员使用内存,存储管理的目标是实现虚拟存储器。

本章在介绍虚拟地址、物理地址和重定位的基础上,描述并分析存储管理的基本方法:单一连续区、固定分区和可变分区等的分区管理,静态分页和请求分页,以及分段和段页式存储管理。主要从方法的基本思想、数据结构、分配回收、重定位、特点等方面进行分析和总结,其中重点是可变分区、静态分页和请求分页的存储管理方法。

固定分区是能够支持多道程序设计的最简单的一种存储管理方法,在系统启动时把用户区划分为一些更小的分区,划分后分区个数和每个分区的长度不再改变,使得程序受分区长度的限制,同时可能存在小程序占用大分区造成的内碎片。可变分区试图改变这一状况,其基本思想是启动时没有进行分区,在程序装入时根据程序的实际需求量动态建立分区。可变分区的数据结构主要是空闲区链表和可用表,基本分配策略是最先适应法(FF)、最佳适应法(BF)和最坏适应法(WF),同时还需要空闲区回收及相邻空闲区合并,程序移动技术等。在可变分区中分区个数和每个分区长度是变化的,存储空间的分配、回收较复杂,而且随着系统的运行出现分配区和空闲区交替出现的存储布局,可能出现长度较小的空闲区,造成外碎片。

分页存储管理是现代操作系统普遍采用的方法。进程分页是由操作系统在程序装入时自动完成,页长度(块长度)取决于 MMU 虚拟地址寄存器结构。分页管理分静态分页和动态分页。在静态分页中,位示图用于描述存储单元的使用状态,位示图的结构紧凑,分配回收时的计算简单。页表的作用是重定位和存储保护,页表的建立和初始化是进程的分配过程。重定位过程是从虚拟地址计算得到页号 p 和页内地址 w,根据页号访问页表得到块号,从而得到物理地址。引入快表可以减少重定位过程 CPU 访问内存的次数。分页管理存储空间的分配、回收操作简单,不仅可以实现内存空间的非连续分配,而且可以实现虚拟存储器,提高存储空间的利用率和方便用户使用。

虚拟存储器思想的理论基础是程序局部性原理。请求分页是一种动态分页,是一种虚拟存储器技术。扩充页表的作用是:①处理器区分程序的内、外存信息;②MMU 重定位;③存储保护;④页面调度需要的参数。缺页中断是由 MMU 重定位过程产生的,页面调度是缺页中断处理过程的一种交换调度,页面调度主要通过置换算法实现。置换算法有:先进先出(FIFO)算法、最近最久未使用(LRU)算法、最近最不常用(LFU)算法、二次机会(Second Chance)算法和页缓冲(Page Buffer)算法。置换算法的目标是将经常访问的页保留在内存中,以减少抖动现象。工作集模型可以用于内存的分配,可以用于确定进程的基本页。一般地,对于一个进程,分配的内存块数越多,它在运行过程中产生的缺页率越小。但是,对于 FIFO 置换算法,存在个别进程,分配给内存的块数增加,缺页率没有减小,甚至反而也增加,这种反常现象称为 Belady 现象。

分页存储管理破坏了程序的完整性,分段存储管理可以保持程序结构的完整性,为进程共享存储空间、实现系统安全机制提供基础。分段管理内存分配采用可变分区相同的动态

分区，不同的是，分段管理是以程序中的段为单位分配内存，而可变分区是以整个程序为单位分配内存。在硬件支持下，分段管理可以实现虚拟存储器。内存中同一个段的程序信息是连续存储，由于程序的段长度的差异，对于长度很长的段，装入时可能没有合适的空闲区，且置换算法复杂。段页式存储管理是把分页和分段结合起来而得到的一种存储管理方法，其中段表、段页表的结构关系是一种应用广泛的数据处理方法。

1. 知识点

(1) 虚拟地址、物理地址。
(2) 重定位及两种方式。
(3) 内碎片和外碎片。
(4) 虚拟存储器。
(5) 缺页中断。
(6) 抖动现象。
(7) Belady 现象。
(8) 置换算法。

2. 原理和设计方法

(1) 固定分区基本思想和数据结构。
(2) 可变分区基本思想、空闲区个数的变化。
(3) FF、BF、WF 分配策略。
(4) 对换、覆盖的区别。
(5) 进程分页中页数的计算，虚拟地址与页号、页内地址的计算。
(6) 位示图及相关计算。
(7) 页表的建立过程及地址重定位。
(8) 快表的作用。
(9) 程序局部性原理。
(10) 扩充页表的结构及 P、A、M 位的作用。
(11) 置换算法及缺页率计算。
(12) 二级页表结构。
(13) 反向页表思想。
(14) 段式管理的提出。
(15) 分段与分页的区别。
(16) 段页式管理中数据结构及关系。

习题

1. 什么是重定位？它有哪两种方式？
2. 固定分区中“固定”表现在哪些方面？写出分区说明表的结构。
3. 在可变分区存储管理中，如果数据结构采用可用表，那么，由于表的长度是固定的，

所以在可变分区存储管理中并发进程数受限制,这种观点正确吗? 请说明理由。

4. 已知可变分区存储管理当前内存使用情况如图 5-53 所示,有 3 个空闲区,区号为 0、1 和 2,长度分别是 90K、140K 和 40K。现有 3 道程序 A、B 和 C 依次要求装入内存,它们的内存需求量分别是 80K、30K 和 130K。分别采用 FF、BF、WF 分配策略,问: A、B、C 各分配在哪个分区? 如果按 B、A 和 C 顺序装入呢?

5. 为什么内存中的程序移动是有条件的?

6. 假定分配策略为 BF,请描述基于链表的下邻空闲区合并及修改算法。

7. 简述对换技术与覆盖技术的主要区别。

8. 试比较内碎片和外碎片的区别。

9. 在某静态页式存储管理中,已知内存共有 32 块,块长度为 4K,当前位示图如图 5-54 所示,进程 P 的虚拟地址空间大小为 35655。问:

(1) 当前有几个空闲块?

(2) 进程 P 共有几页?

(3) 根据图 5-54 的位示图,写出进程 P 的页表。

(4) 给定进程 P 的虚拟地址: 9198 和 0x9D8F,根据(3)的页表,分别计算对应的物理地址。

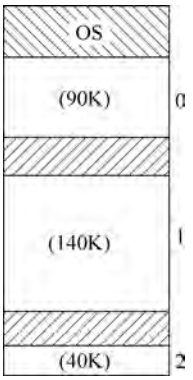

图 5-53 习题 4

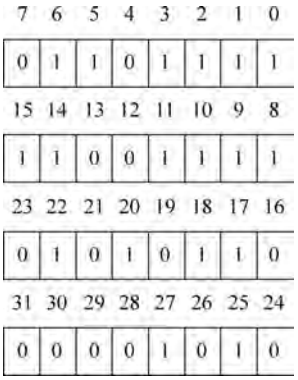

图 5-54 习题 9

10. 简述分页存储管理中基于空闲区链表管理空闲块时分配、回收一个空闲块的过程。

11. 在请求分页存储管理中,置换算法将内存的页淘汰,因而造成系统开销甚至产生抖动现象,那么,在缺页中断处理过程,内存没有空间闲块时不需要置换算法,而是让当前进程进入阻塞状态。这种方法可以吗? 为什么?

12. 已知进程的引用序列为 3、1、4、1、2、3、5、2、3、1、4、5、0、3、5、2、4、1。采用纯请求分页(开始运行时所有页还没有装入内存)的局部置换算法,如果分配给该进程的内存块数为 4,请分别给出 FIFO、LRU 和二次机会置换算法时,缺页的次数和依次淘汰的页号。

13. 在某请求分页存储管理中,已知内存共有 32 块,块长度为 4K,当前位示图如图 5-22 所示,进程 A 的虚拟地址空间大小为 30000。采用局部页面调度 LRU 置换算法,分配进程 A 的内存块数为 4。假定进程 A 的引用序列为 2、5、1、3、1、4、5、1、2、6、3、4、7、5。请完成:

(1) 进程 A 共有几页?

(2) 如果进程 A 的引用序列中前 4 页作为基本页,依次装入内存后开始执行。根据上述位示图,建立并初始化页表(页表至少包含页号、块号和中断位 P)。

(3) 在(2)的基础上,进程 A 执行了一段时间后,现在 CPU 要访问进程 A 的虚拟地址 25704,请给出这段时间基于 LRU 置换算法的页面调度过程,并给重定位后的页表。

(4) 在(3)的基础上,计算虚拟地址 25704 的物理地址。

14. 说明扩充页表中修改位 M 的作用。

15. 什么是虚拟存储器? 什么是抖动?

16. 简述快表的作用。

17. 讨论分页存储管理中块长度对系统的影响。

18. 分页与分段的主要区别有哪些?

19. 画图表示段页式存储管理中一个进程的段表与段页表的关系。