思想引领

视频讲解

第 5 章 内存管理

本章知识要点：本章知识要点包括内存管理的功能、内存分配形式、静态和动态重定位、内存存储的覆盖与交换，以及常用的内存管理方法，如分区、页式、段式、段页式和虚拟存储器等原理和实现方法；同时包括 UNIX、Linux、Windows、OpenHarmony 的内存管理实例概况，以及内存管理的设计与实现问题。

预习准备：了解自己使用的计算机的存储器情况，回顾程序设计中对内存空间的使用情况，接着可思考和预览内存管理的任务和功能概况，再预览各知识点的基本概念和其功能实现的基本思想。

兴趣实践：设计实现动态分区的内存分配与回收算法，设计实现页框分配和回收算法，设计实现 FIFO、LRU、NRU 和 Clock 页面置换算法，以及在 UNIX、Linux 系统中，设计动态申请内存和设置共享存储区使用的应用程序。

探索思考：现代计算机内存空间都很大，如何高效实现内存的分配和回收？如何有效保证多进程间对内存空间使用的一致性和保护各进程信息的隐私性？

主存储器（又称为内部存储器、内存、主存）的管理一直是操作系统最主要的功能之一。在现代计算机系统中，尽管主存容量已经很大，价格已相当便宜，但主存储器依然是四大硬件资源中最关键、最紧张的“瓶颈”资源。任何程序和数据及各种控制用的数据结构都必须占用一定的内存空间。因此，能否合理、有效地使用主存储器，在很大程度上反映了操作系统的性能，并直接影响整个计算机系统作用的发挥。本章将主要介绍几种常用的内存管理方法，如分区、页式、段式、段页式存储管理和虚拟存储器等原理和实现方法，最后介绍 UNIX、Linux、Windows、OpenHarmony 的内存管理实例和内存管理设计与实现问题。

5.1 内存管理的功能

▶ 5.1.1 计算机系统的多级存储结构

为了更多地存放和更快地处理用户信息，目前许多计算机把存储器分为三级：外部存储器、主存储器（内存）和高速缓冲存储器。多级存储结构如图 5-1 所示。外部存储器（简称外存）用来存放不立即使用的程序和数据，当用户的程序运行需要它们时，再从外存把它们读入主存储器。一个程序的运行总是存放在主存中，以便处理器的访问。由于处理器的运算部件和控制部件比主存的存取速度快得多，为了使处理器的处理速度和到存储器中存取的速度得到较好的匹配，就引入了高速缓冲存储器，由硬件机构自动控制主存信息块与高速缓冲存储器信息块的交换，这样处理器取指令和存取数据就在高速缓冲存储器中进行，从而平滑了主存与处理器的信息流动。从外部存储器到高速缓冲存储器，其存取速度越来越快，容量越来越小，而价格越来越昂贵。高速缓冲存储器不参与指令的编址，它只是为

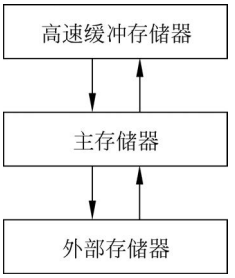


图 5-1 多级存储结构

了提高计算机的处理速度。

本章主要介绍主存储器(内存)空间的管理原理和实现技术。

▶ 5.1.2 内存管理的任务和功能

为了对内存进行合理有效的管理,一般将内存空间分为系统区和用户区两大部分。系统区主要存放操作系统常驻内存部分和一些系统软件常驻内存部分以及相关的系统数据；用户区主要用来存放用户的程序和数据。操作系统内存管理主要是针对用户区进行的,在多用户系统中,需要将内存空间划分成更多的区域,以便同时存放多个用户的进程。那么,合理、有效的内存管理机制,必将大大提高操作系统的性能。

内存管理的主要任务如下。

(1) 为多道程序的并发提供良好的环境,使每道程序都能在不受干扰的环境中运行。

(2) 提高内存利用率,尽量减少空闲的及不可利用的内存区域,使得有限的内存能更好地为多个用户程序服务。

(3) 逻辑上扩充内存空间,使大程序能在小内存中运行。

(4) 方便用户使用内存,用户无须考虑内存的分配、回收和保护等工作,这些工作对于用户来说是“透明”的,完全由操作系统进行管理。

为了完成上述任务,要求内存管理必须具备以下几个功能。

1. 内存空间的分配和回收

操作系统中的内存管理能根据记录每个存储区(分配单元)的状态作为内存分配的依据。当用户提出申请时,实施内存空间的分配管理,并能及时回收系统或用户释放的存储区,以供其他用户使用。为此,这种内存分配机制应能完成如下工作。

(1) 记住每个存储区域的状态,哪些是已经分配的,哪些还可以用于分配。保存每个存储区域的状态的数据结构称为内存分配记录表。

(2) 实施分配。在系统程序或用户提出申请时,按所需的量给予分配,并修改相应的内存分配记录表。

(3) 接收系统或用户释放的内存区域,并相应地修改内存分配记录表。

2. 地址映射和重定位

程序设计人员在进行程序设计时,用来访问信息时所用到的一系列地址单元的集合称为逻辑地址。而存储空间是内存中物理地址的集合。在多道程序环境下,程序不是事先约定存放位置,而是在执行过程中可以动态浮动,故程序的逻辑地址和物理地址是不一致的,因此需要内存管理机制提供地址映射功能,把程序地址空间中的逻辑地址转换为内存空间中对应的物理地址。

3. 内存共享与保护

由于内存区域为多名用户程序共同使用,所以内存共享有两方面的含义：①是指多个用户程序共同使用内存空间,各个程序使用各自不同的内存区域；②是指多个用户程序共同使用内存中的某些程序和数据区,这些共享程序和数据区称为共享区。因此,内存管理必须研究如何保护各内存区中的信息不被破坏和偷窃,同时当多个程序共享一个内存区时,也要对共享区进行保护,确保信息的完整性和一致性。

4. 内存扩充

计算机在实际的应用中,常常出现小内存无法满足大程序的要求。同时,内存单元的容量

受到实际存储单元的限制。因而，内存管理机制必须提供相应的技术，来达到内存单元逻辑上的扩充。现在采用的一般是虚拟存储技术或其他自动覆盖和交换技术。

5.2　内存分配的几种形式与重定位

▶ 5.2.1　内存分配的几种形式

内存分配所要解决的问题是多道程序之间如何共享内存的存储空间，即内存管理在什么时候采用什么样的方式将一个程序运行时所需要的信息分配到内存中，并使这些问题对用户来说尽可能是“透明”的。

解决内存分配问题有以下三种方式。

1. 直接内存分配方式

程序设计人员在程序设计过程中，或汇编程序对源程序进行编译时，所用的是实际物理内存地址，以确保各程序所用的地址之间互不重叠。显然，直接内存分配方式要求内存的可用空间已经确定，这对于单用户计算机系统来说是不成问题的。在多道程序设计发展初期，通常将内存空间划分成若干个固定的不同大小的分区，并对不同的程序指定不同的分区。对于程序设计人员或编译系统而言，内存的可用空间是已知的。这样，不仅用户感到不方便，而且内存的利用率也不高。

2. 静态内存分配方式

采用静态内存分配方式时，用户在编写程序或由编译系统产生的目的程序中采用的地址空间为逻辑地址。当连接程序对它们进行装入、连接时，才确定它们在内存中的相应位置（物理地址），从而产生可执行程序。这种分配方式要求用户在进行装入、连接时，系统必须分配其要求的全部内存空间，若内存空间不够，则不能装入该用户程序。同时，用户程序一旦装入内存空间后，它将一直占据着分配给它的内存空间，直到程序结束时才释放该空间。再者，在整个运行过程中，用户程序所占据的内存空间是固定不变的，也不能动态地申请内存空间。

显然，这种分配方式不仅不能实现用户对内存空间的动态扩展，也不能有效地实现内存资源的共享。

3. 动态内存分配方式

动态内存分配方式是一种能有效使用内存的方法。用户程序在内存空间中的位置，虽然也是在装入时确定的，但是，它不必一次性将整个程序装入内存中，可根据执行的需要，一部分一部分地动态装入。同时，装入内存的程序不再执行时，系统可以收回该程序所占据的内存空间。再者，用户程序装入内存后的位置，在运行期间可根据系统需要而发生改变。此外，用户程序在运行期间也可动态地申请内存空间以满足程序需求。动态内存分配通常可采用覆盖与交换技术实现。

由此可见，动态内存分配方式在内存空间的分配和释放上，表现得十分灵活，现代的操作系统常采用这种内存分配方式。

▶ 5.2.2　重定位

为了实现静态、动态内存分配方式，必须把逻辑地址和物理地址分开，并将逻辑地址定位为物理地址。为此，首先要弄清地址空间和存储空间这两个概念。

1. 地址空间和存储空间

用户在编写程序时，是通过一些符号名称来调用、访问子程序和数据的，这些符号名与存储器地址无任何直接关系。源程序经过编译或是汇编以后，产生了目标程序，而编译系统总是从零号地址单元开始，为目标程序指令顺序分配地址。这些地址被称为相对地址，或者是逻辑地址。相对地址的集合称为逻辑地址空间，简称地址空间。

存储空间是指内存中一系列存储信息的物理单元的集合。这些物理单元的编号称为物理地址或绝对地址。因此，存储空间的大小是由内存的实际容量决定的。

显然，逻辑地址空间是逻辑地址的集合，是相对于用户或程序设计人员的，是一个“虚”的概念，而存储空间是物理地址的集合，是系统管理和维护的对象，是一个“实”的物体。用户设计好的一个程序是存在于它自己的地址空间中的，只有当它要在计算机上运行时，系统才将它装入存储空间中。

2. 重定位的概念

在一般情况下，用户的一个程序在装入时所分配的存储空间和它的地址空间是不一致的，因此，用户程序在 CPU 上执行时，其所要访问的指令和数据的物理地址和地址空间中的相对地址是不同的，程序由地址空间装入存储空间如图 5-2 所示。显然，如果用户程序在装入或执行时，不对有关地址进行修改，则将会导致错误的结果，这种由于用户程序的装入而引起的地址空间中的相对地址转换为存储空间中的绝对地址的地址变换过程，称为地址重定位，也称为地址映射。实现地址重定位或地址映射的方法有两种：静态地址重定位和动态地址重定位。

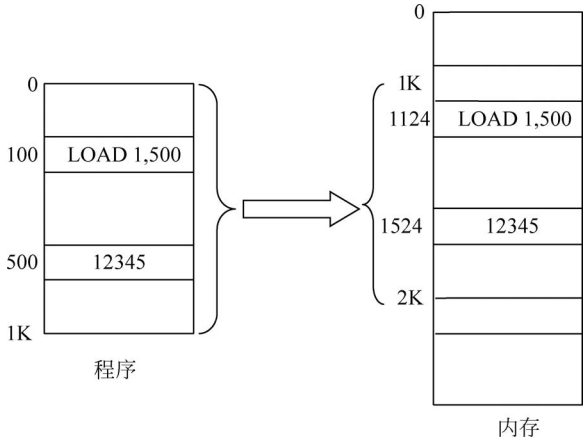


图 5-2　程序由地址空间装入存储空间

1）静态地址重定位

静态地址重定位是指用户程序在装入时由装配程序一次完成，即地址变换只是在装入时一次完成，以后不再改变，如图 5-2 中，LOAD 1，500→LOAD 1，1524。这种重定位方式实现起来比较简单容易，在早期多道程序设计中大多采用这种方案，但是，它也存在不少缺点。

（1）用户程序必须分配一个连续的内存存储空间。

（2）难以实现程序和数据的共享。

2）动态地址重定位

动态地址重定位是在程序执行的过程中，当 CPU 要对内存进行访问时，通过硬件地址变换机构，将要访问的程序和数据地址转换成内存地址。地址重定位机构至少需要一个重定位寄存器 BR 和一个相对地址寄存器 VR。指令或数据的主存地址 MA 与逻辑地址的关系为

MA＝(BR)＋(VR)。动态重定位过程如图 5-3 所示。

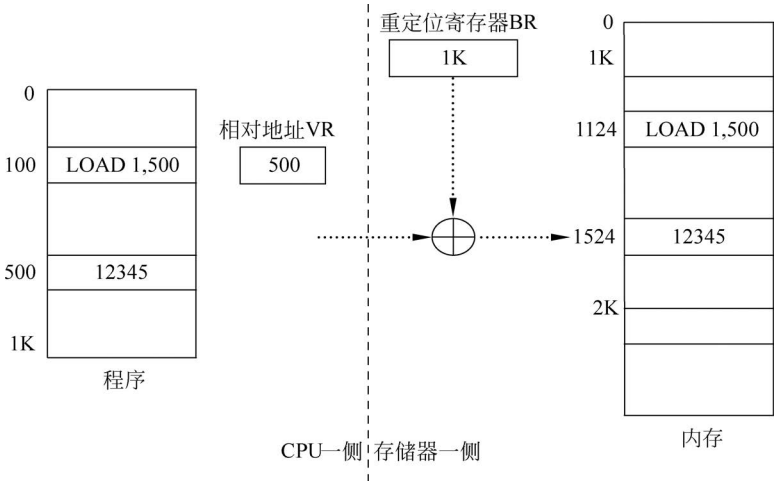


图 5-3　动态重定位过程

(1) 其具体过程如下。

① 设置重定位寄存器 BR 和相对地址寄存器 VR。

② 将程序段装入内存，且将其占用的内存区起始地址送入 BR 中，如(BR)＝1K。

③ 在程序执行过程中，将所要访问的相对地址送入 VR 中，如(VR)＝500。

④ 地址变换机构把 VR 和 BR 的内容相加，得到实际访问的物理地址。

(2) 动态地址重定位的优点。

① 执行时程序可以在内存中浮动，对于移动后的程序，只需按程序存放的起始单元地址来修改重定位寄存器 BR 的值，程序又可继续执行，有利于提高内存的利用率和内存空间使用的灵活性。

② 有利于程序段的共享实现。当系统提供多个重定位寄存器 BR 时，规定某些或某个重定位寄存器作为共享程序段使用，就可实现内存中的相应程序段为多个程序所共享。

③ 为实现虚拟存储管理提供了基础。有了动态地址重定位的概念和技术，程序中的信息块可根据执行时的需要分配在内存中的任何区域，还可以覆盖或交换不再使用的区域，使得程序的逻辑地址空间可比实际的物理存储空间大，从而实现了虚拟存储管理功能。

(3) 动态地址重定位的缺点。

① 实现存储器管理的软件比较复杂。

② 需要附加的硬件支持。

▶ 5.2.3　覆盖与交换

覆盖与交换是从逻辑上扩充内存的两种方法，主要解决在较小内存空间中如何执行大程序的问题。

1. 覆盖技术

在单 CPU 系统中，每一时刻 CPU 只能执行一条指令，而且一个用户程序并不需要一开始就将它的全部程序和数据装入内存中。因此，可以把程序划分为若干个功能相互独立的程序段，并且让那些不会同时被 CPU 执行的程序段共享同一个内存区。通常，这些程序段被保存在外存中，当 CPU 要求某一程序段执行时，才将该程序段装入内存中覆盖以前的某一个程序

段。在用户看来，内存好像扩大了，这便是覆盖技术。

覆盖技术要求程序员提供一个清楚的覆盖结构。程序员在设计过程中必须完成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的工作。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，这在无形中给程序员增加了负担。

覆盖示例如图 5-4 所示。某一用户程序由 A、B1、B2、C1、C2 和 C3 这 6 个程序段组成，它们之间的关系如图 5-4(a)所示，程序段 A 调用程序段 B1 和 B2，程序段 B1 调用程序段 C1，程序段 B2 调用程序段 C2 和 C3。

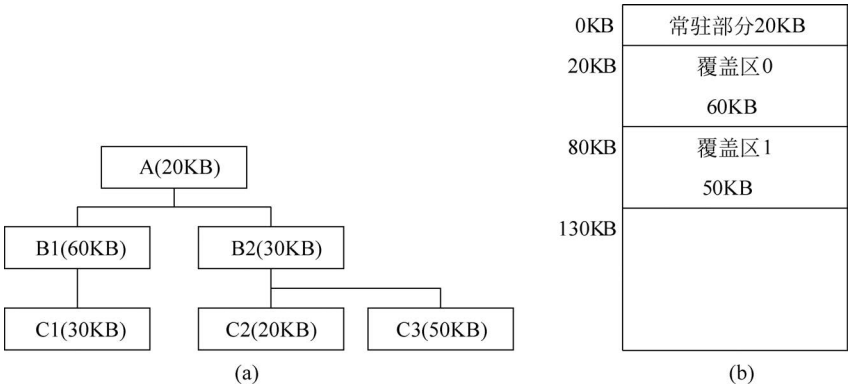


图 5-4　覆盖示例

由图 5-4 可知，程序段 B1 和 B2 之间不会相互调用，因此，可以将程序段 B1 和 B2 共享一个内存区，其分配的内存大小为 B1 和 B2 中所需内存的较大者，即 60KB。同理可知，程序段 C1、C2 和 C3 也可共享一个内存区，内存分配大小为 50KB。这样，可以按照如图 5-4(b)所示的形式来划分覆盖结构。同时，还可以看到，用户程序所要求的内存空间为 A(20KB)＋B1(60KB)＋B2(30KB)＋C1(30KB)＋C2(20KB)＋C3(50KB)＝210KB，但采用了覆盖技术后，只需要 20KB＋60KB＋50KB＝130KB 的内存空间，大大提高了内存的利用率。

2. 交换技术

交换技术就是将系统暂时不用的程序或数据部分或全部从内存中调出，以腾出更大的存储空间，同时将系统要求使用的程序和数据调入内存中，并将控制权转交给它，让其在系统上运行。实际上，这种技术是通过在内存与外存之间不断地交换程序和数据，以实现用户在较小的内存空间中完成较多程序的执行。这样，从用户角度(逻辑上)看，内存容量得到了扩充。

与覆盖技术相比，交换技术不要求程序设计人员给出程序段之间的覆盖结构，它主要是在进程之间进行，而覆盖技术则主要是在同一个进程之间进行。交换技术的运用，可以在较小的内存空间中运行较多的程序，覆盖技术的运用，可以在较小的内存空间中运行比其容量大的程序。

5.3　分区内存管理

在单道环境下，一般采用单一连续区分配方式，此方式内存空间除了被系统占用外，其他剩余空间全部被一个用户程序所占用，因此管理起来较为简单。在多道程序设计环境下，为实现各并发进程共享内存空间，可以采用分区内存管理方式。分区内存管理是将内存的用户可用区划分成若干个大小不等的区域，每一个进程占据一个区域或多个区域。分区管理根据分

区的时机不同，分为固定分区和动态分区两种方法。

固定分区是指系统在初始化时，将内存空间划分为若干个固定大小的区域。用户程序在执行过程中，不允许改变划分区域的大小，只能够根据各自的要求，由系统分配一个存储区域。固定分区存储分配技术，虽然可以使多个进程在同一时刻共享内存区，但它不能充分利用内存资源。因为一个进程占据内存的大小，只有当它在调入内存时，由调度程序分析才能确定，而分区的大小是在系统初始化时进行划定的。由于用户进程占据的主存空间不可能刚好等于某个分区的大小，所以，在已分配的分区中，通常都有一部分未被进程占用而浪费的主存空间，这一部分空间称作内存的“碎片”或“内零头”。在固定分区分配方式中，由于存在“碎片”问题，所以内存浪费现象比较严重。为了解决这一问题，引进了动态分区分配方式（又称为可变分区分配方式）。

▶ 5.3.1　动态分区的基本概念

采用动态分区分配方式，在系统初启时，除了操作系统中常驻内存部分以外，只存在一个空闲分区。随后，分配程序将该区依次划分给调度程序选中的进程，并且分配的大小可随用户进程对内存的要求而改变，内存分配情况如图 5-5 所示。显然，这种分配方式不会产生“碎片”现象，从而大大提高了内存的利用率。与固定分区法相同，动态分区也要使用分区说明表等数据结构来对内存进行管理。但由于系统在运行的过程中，无法确定分区的个数和分区的大小等情况，分区说明表的大小也难以确定。因而，在动态分区分配方式中，是采用将内存中的空闲区单独构成一个可用分区表或可用分区自由链表的形式以描述系统内存管理。此外，请求内存资源的进程也构成一个内存资源请求表。可用分区表、自由链表和请求表如图 5-6 所示。可用分区表的每个表目记录一个空闲区，其主要参数由区号、分区长度和起始地址组成。采用表格结构来管理空闲区比较直观，管理算法也简单，但表格的大小难以确定。

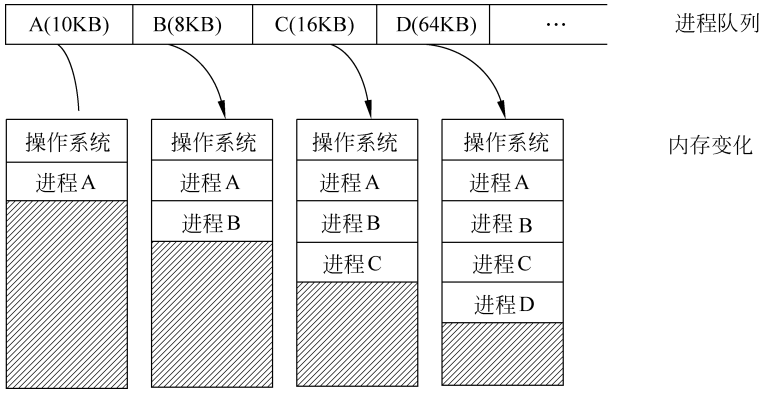
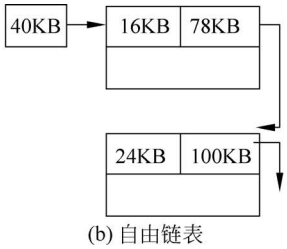


图 5-5　内存分配情况

区号	分区长度	起始地址
1	16KB	40KB
3	24KB	78KB
5	9KB	100KB

(a) 可用分区表



(b) 自由链表

进程号	请求长度
P_1	15KB
P_2	25KB

(c) 请求表

图 5-6　可用分区表、自由链表和请求表

自由链表是利用每个空闲区的开始几个存储单元来存放本空闲区的大小及下一个空闲区的起始地址，从而将所有的空闲区都链接起来。然后，系统再设置一个自由链表首指针，让其指向第一个空闲区。这样，存储管理程序可以通过自由链表的首指针查找到所有的空闲区。请求表的每个表目登记着请求内存资源的进程号以及所需的主存大小。

必须注意，无论是采用可用分区表还是自由链表方式，表中的各项都要按照一定的规则排列以利于查找和回收。以下进一步讨论动态分区的分配与回收问题。

▶ 5.3.2 动态分区的分配与回收

1. 动态分区的分配方式

动态分区的存储分配方式是指如何从可用分区表或自由链中寻找满足条件空闲区分配给相应的进程。通常有三种方式：最先适应法、最佳适应法、最坏适应法。

（1）最先适应法是将进程分配到内存的第一个足够装入它的可用空闲区中。采用这种算法实施分配时，找到的第一个适应要求的空闲区，其大小不一定正好等于进程所要求的大小。因此，该空闲区会分为两个区，一个是已分配区，其大小正好等于进程所要求的大小；另一个仍为空闲区，并保留在可用分区表或自由链中。

这种算法的缺点是可能将大的空闲区域分割成一个小区，不利于大程序的装入与运行。改进的方法是，把空闲区按地址从小到大排列在可用分区表或自由链中，分配时，尽可能地利用内存的低地址部分的空闲区，而尽量保留高地址部分为大的空闲区，以便满足当程序要求较大内存空间时的要求。

（2）最佳适应法是将进程分配到内存中与它所需大小最接近的一个可用空闲区中。采用这种算法要求可用分区表或自由链按照空闲区从小到大的次序排列。当用户进程申请一个空闲区时，存储器管理程序就从可用分区表或自由链的头部开始查找，当找到第一个满足条件的空闲区时，停止查找，进行内存区的分配。

这种算法的优点是从空闲区中挑选一个能满足程序要求的最小分区，这样可以保证不会去分割一个更大的空闲区，便于今后大程序的装入运行。其缺点是由于空闲区通常不可能正好和程序所要求的大小相等，因而要将其分割成两部分，这往往使剩下的空闲区非常小，以至几乎无法使用。随着系统的运行，这种小空闲区也逐步增多，造成了内存空间的浪费。故有些系统往往还采用与之相反的分配算法，即最坏适应法。

（3）最坏适应法是把一个进程分配到主存中最大的空闲区中。采用这种算法同样要求可用分区表或自由链按照空闲区从大到小的次序排列。当用户进程申请一个空闲区时，存储管理系统分析可用分区表或自由链中的第一个空闲区是否满足用户进程要求，若满足要求，则将第一个空闲区分配给它；否则分配失败。

这种分配方式看起来十分荒唐，但是经过分析后发现，最坏适应算法也有很强的直观性。其原因是：在大空闲区中装入程序后，剩下的空闲区常常也很大，于是也能满足以后较大的程序的要求。该算法对中、小程序的运行是很有利的。

2. 动态分区的回收

实际上，在每一种内存分配方案中，都包含一定程度的浪费。在动态内存分配中，也存在着这种的现象。动态分区方式中内存区的释放和回收如图 5-7 所示，系统将进程队列中的进程逐步装入内存中，但随着系统的运行，进程陆续完成，它们将释放掉所占用的内存空间，在内存中形成一些空白区。这些空白区可以被其他进程使用，但由于空白区和调入主存的进程要求的大小不是正好相等，因而会出现更小的空白区，这些小的空白区容量无法满足其他进程的需要而白白浪费。同时，随着系统运行的时间加长，这些小的空白区的数量也将会增多。为了

避免这种浪费现象，系统提供了相应的回收程序，将释放的分区与它相邻的空闲分区进行合并，形成一个更大的空闲分区。

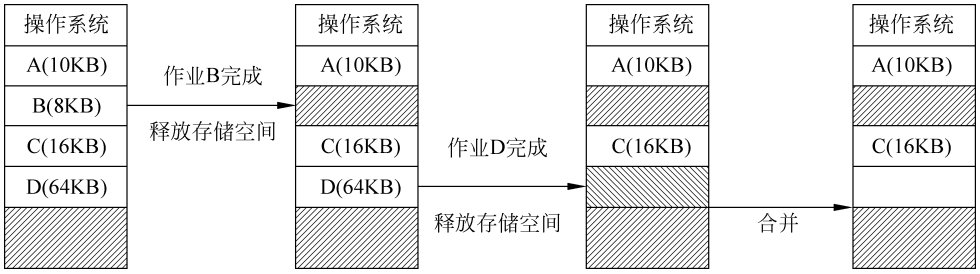


图 5-7　动态分区方式中内存区的释放和回收

通常，分区的回收有 4 种情况。

（1）释放区与上下两个空闲区相邻。在这种情况下，将三个空闲区合并为一个空闲区。新空闲区起始地址为上空闲区的起始地址，大小为三个空闲区之和。同时，修改可用分区表或自由链中的表目。

（2）释放区与上空闲区相邻。在这种情况下，将释放区与上空闲区合并为一个空闲区，其起始地址为上空闲区的起始地址，大小为释放区和上空闲区之和。同时，修改可用分区表或自由链中的表目。

（3）释放区与下空闲区相邻。在这种情况下，将释放区与下空闲区合并为一个空闲区，其起始地址为释放区的起始地址，大小为释放区和下空闲区之和。同时，修改可用分区表或自由链中的表目。

（4）释放区与上下两个空闲区都不相邻。在这种情况下，释放区作为一个新的空闲可用区插入可用分区表或自由链中。

▶ 5.3.3　分区管理的其他问题

1. 地址转换与存储保护

对动态分区方式应采用动态重定位装入程序，当程序执行时由硬件地址转换机构完成地址转换。硬件机构必须设置两个专用的特权寄存器：基址寄存器和限长寄存器。基址寄存器存放分配给程序使用的分区的最小绝对地址值；限长寄存器存放程序占用的连续内存空间的长度。当程序装入所分配的区域后，操作系统把该区域的始址和长度送入基址寄存器和限长寄存器，启动程序执行时由硬件机构根据基址寄存器和限长寄存器进行地址转换，从而得到绝对地址。地址转换过程如图 5-8 所示。

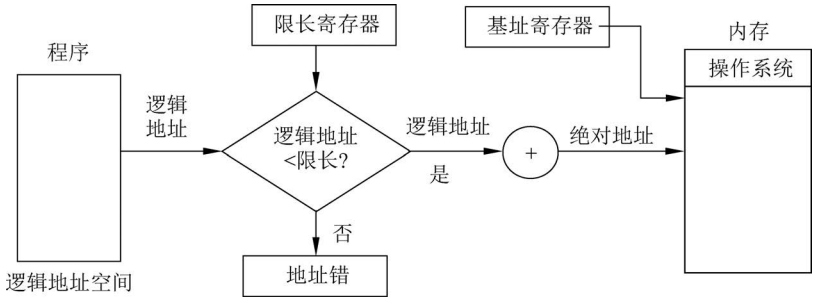


图 5-8　地址转换过程

当逻辑地址小于限长值时，则逻辑地址加基址寄存器值就可得到绝对地址；当逻辑地址大于限长值时，表示程序欲访问的地址超出了所分得的区域，这时就产生地址越界中断，终止程序执行，报告地址出错信息，从而起到保护内存的作用。

即使在多道程序设计系统中，仍然也只需一对基址/限长寄存器。基址/限长寄存器的内容为正在执行的程序的现场内容之一。某进程在执行过程中出现等待时，操作系统必须把基址/限长寄存器的内容随同该进程的其他信息，如 PSW、通用寄存器等一起保存起来。当进程被选中执行时，则把选中进程的基址/限长值再送入基址/限长寄存器中。

2. 分区的共享

在分区管理方式中，如果每个进程只能占用一个分区，那么就不允许各道进程存在公共的共享区域。这样，当几道进程都要使用某个例行程序时，就只好在各自的存储区域内各放一套了，这种方式显然降低了内存的使用效率。所以有些计算机系统提供了多对基址/限长寄存器，允许一个进程占用多个分区。系统可以规定某对基址/限长寄存器限定的区域是共享的，用来存放共享的程序和数据。对共享区的信息也必须规定只能执行或读出，而不能写入，若某进程要想往该共享区域写入信息时，则将遭到系统的拒绝，并产生保护中断。因此，几道进程共享的例行程序或数据就可存放在一个共享的分区中，只要让各道进程的共享内存区域部分有相同的基址/限长值，就可实现分区共享。

3. 移动技术

当内存分配程序在可用分区表或自由链表中找不到一个足够大的空闲区来装入进程时，可以采用移动技术改变内存中的进程存放区域，同时修改它们的基址/限长值，从而使分散的小空闲区汇集成一个大的空闲区，有利于进程的装入。移动分配示例如图 5-9 所示。

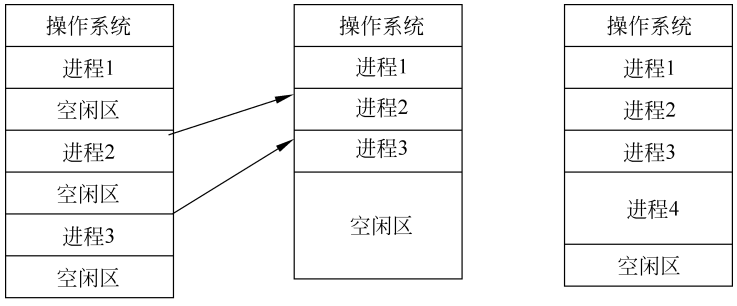


图 5-9　移动分配示例

移动的好处是可使分散的“碎片”或小空闲区汇集成一个大的空闲区，但它却增加了系统的开销，此外，也不是任何时候都能对一个进程进行移动的。例如，当某道进程正在与外部设备交换信息时，I/O 控制机构总是按已经确定的主存绝对地址完成信息的传输。若这时移动该进程，则交换信息将出错。所以当一道进程正在与外部设备交换信息时往往不能移动。由于移动增加了系统的开销，故应尽量设法减少移动。例如，当要装入一道进程时总是先挑选不经移动就可装入的进程；在不得不移动时也力求使移动的道数最少。

移动技术也为进程执行过程中扩充内存提供了方便。一道进程在执行中若要增加内存容量时，只需适当移动邻近的进程就可以增加它所占用的连续区域的长度。所有被移动后的进程基址值与该进程扩大后的限长值都应做相应的修改。当然，允许进程在执行过程中动态扩充内存，有时还会出现死锁问题。可以考虑将相应卷入死锁的某个或一些进程调出内存，存到

辅助存储器中，然后，让留在内存的那些进程获得主存资源并继续执行下去，直到它们归还主存后，再将送出去的进程逐个调回来，满足它们的内存需求，继续执行。

4. 分区内存管理的优缺点

(1) 主要优点。

① 实现了多道程序设计，从而提高了系统资源的利用率。

② 系统要求的硬件支持少，管理简单，实现容易。

(2) 主要缺点。

① 由于进程在装入时的连续性，内存的利用率不高。采用移动技术可以提高内存的利用率，但增加了系统的开销，同时也带来了其他一些较为棘手的问题，如与外部设备交换信息的出错问题、动态内存扩充的死锁问题等。

② 内存的扩充只能采用覆盖与交换技术，无法真正实现虚拟存储。

5.4　页式存储管理

▶ 5.4.1　概述

由分区管理可知，尽管分区管理从实现方法来看比较简单，但由于分区管理方式要求进程占用内存的一个或多个连续的存储区域，这样会导致整个计算机存储系统的一系列问题。

(1) 当连续空闲区不能满足进程的要求时，即使系统中所有空闲区之和大于进程对主存的要求，仍然不能装入进程。

(2) 存储区中仍然存在“碎片”的现象，使内存利用率不高，采用移动技术将分散的“碎片”合并成一个较大的可用区域，但“碎片”的合并需要占用 CPU 的时间，并且合并也不是随时都能进行的。

(3) 分区管理方式无法有效地实现虚拟存储技术，即：使进程的逻辑地址空间大于实际的内存物理空间，从而使有限的内存运行较大、较多的程序。

为了克服分区管理的这些缺点，20 世纪 60 年代，人们提出了分页管理体系。其基本思想是将进程分配在不连续的大小相同的存储区域中，实现内存“见缝插针”式的分配，同时又要保证进程的连续执行。页式管理的基本思想如下。

页式存储器管理取消了内存分配的连续性，它能够将用户进程分配到不连续的存储单元中连续执行。操作系统在初始化时，按照一定的原则，将内存空间划分为大小相等的块或页框(page frame)，通常页框的大小是 2 的整数次幂。同时，用户进程在调入内存之前，操作系统将进程的地址空间划分成与页框相等的片，称为页面(page)。经系统划分后，进程的地址空间一般由页号和页内偏移两部分组成。分页系统的地址结构如图 5-10 所示，这是一个页长为 2KB，占有 512 页的内存空间的地址结构。然后，系统将用户进程的每一个页面分配到内存的页框中。分配时，要求用户进程在页框内是连续的，但页框与页框之间不一定要连续。

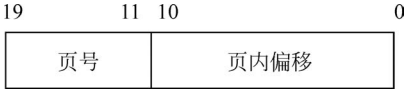


图 5-10　分页系统的地址结构

与分区管理相比，可以看到页式管理方式的优越性主要体现在两方面：其一是实现了连续存储到非连续存储的飞跃，为实现虚拟存储打下了基础；其二是解决了内存中的“碎片”问题，因为从分配思想上看已不存在空闲的页框不可利用的问题，尽管每个进程的最后一页不一定占满整个页框，这部分未占满页框的存储区域称为“内碎片”或“内零头”，任意一个“内碎片”

或“内零头”都不会大于整个页框的大小，从而提高了内存的利用率。

分页管理根据进程装入内存的时机不同，一般分为静态分页管理和虚拟页式存储管理。下面将具体介绍这些内存管理方法。

▶ 5.4.2 静态分页管理

静态分页管理是指用户进程在开始执行以前，将该进程的程序和数据全部装入内存中，然后，操作系统通过页表和硬件地址变换机构实现逻辑地址到物理地址的转换，是执行用户程序的。

1. 主存页框的分配与回收

静态分页管理首先要为要求内存的作业或进程分配足够的页框。这就需要系统建立存储页框表、请求表和页表等数据结构，依据这些数据结构完成内存的分配和回收工作。

1）页表

由于分页管理实现了程序的连续存储到非连续存储的飞跃，但是如何保证该程序能在非连续的存储空间里正确地运行呢？这就要求在执行每条指令时，要将程序中的逻辑地址变换为物理地址，即进行动态重定位。在页式管理系统中实现这种地址变换的数据结构称为页面映像表，简称页表。

页表占用内存的一块固定的存储区，它是在程序装入内存创建其相应进程时，由操作系统根据内存的分配情况建立的。页表中需要两个信息，一个是页号，另一个是页面对应的页框，记录着该进程的每个页面分配到内存的哪些页框中。静态分页内存分配映像如图 5-11 所示，进程 1 和进程 2 通过页表指出了在内存中的分配情况。显然，每个进程至少拥有一张页表。

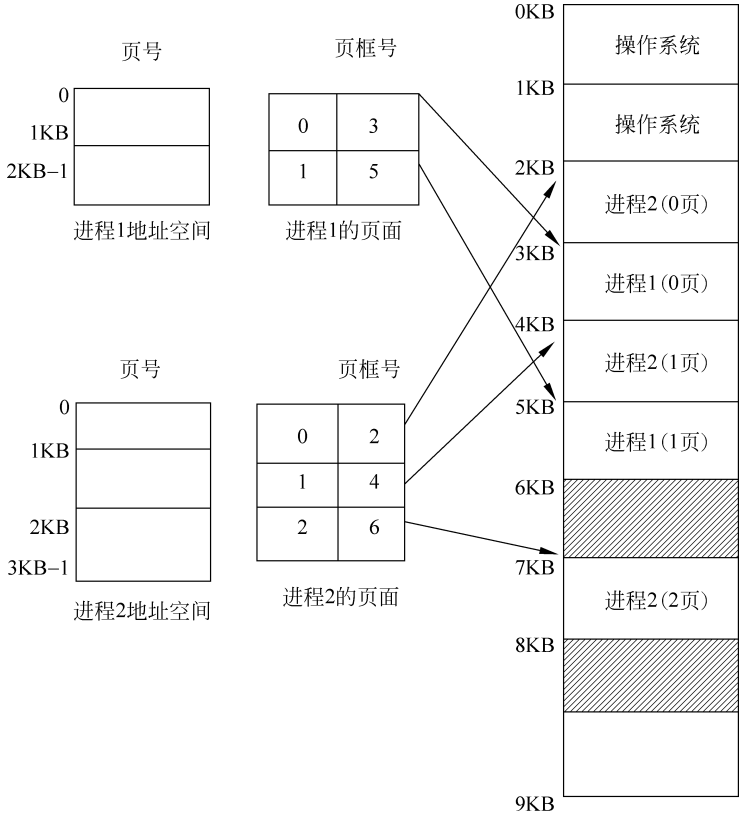


图 5-11 静态分页内存分配映像

2）请求表

当系统有多个进程时，系统必须知道每个进程的页表起始地址和长度，才能进行内存分配和地址变换。请求表就是用来确定进程的虚拟地址空间的各页表在内存中的实际对应位置。整个系统设置一张请求表，请求表如表 5-1 所示，其内容包括进程号、请求页面数、页表始址、页表长度和状态等。

表 5-1　请求表

进　程　号	请求页面数	页 表 始 址	页 表 长 度	状　　态
1	20	1024	20	已分配
2	30	1044	30	已分配
3	21			未分配
⋮	⋮	⋮	⋮	⋮

3）存储页框表

为了描述内存空间的分配情况，系统还设置一张存储页框表。存储页框表指出了内存各页框是否已被分配，以及未被分配的页框总数。存储页框表的形式有两种：一种是在内存中划分出一个固定的区域，该区域中每个单元的每个位表示一个页框的分配或空闲状况，若该位为 1，代表所对应的页框已分配，若该位为 0，代表所对应的页框空闲。这种存储页框表称为位示图。位示图如图 5-12 所示。

0	1	2	3	4	…	27	28	29	30	31
0	1	1	0	0	…	1	1	0	1	1
0	0	1	1	1	…	0	1	1	1	0
1	0	0	1	1	…	1	1	0	1	0

图 5-12　位示图

位示图要占用一部分内存容量，一个划分为 2048 页框的内存，如内存单元长度为 32 位，则位示图就占据 2048/32＝64 个内存单元。

存储页框表的另一种形式是采用空闲页框链的方法。在空闲页框链中，队首页框的第一单元和第二单元分别存放空闲页框的总数和指向下一个空闲页框的指针，其他页框的第一单元则分别存放指向下一个空闲页框的指针。空闲页框链的方法由于使用了空闲页框本身的存储单元来存放空闲页框链的指针，因此不占据额外的内存空间，是一种较为经济的存储页框表的组织法。

4）页框分配与回收算法

为进程分配页框时，首先，从请求表中查出进程所要求的页框数。然后，由存储页框表检查是否有空闲页框（块），若没有，则本次无法分配；若有，则分配并设置页表，并填写请求表中的相应表项（页表始址、页表长度和状态）。之后，再按一定的查找算法，搜索出所要求的空闲页框（块），并将对应的页框（块）号填入页表中。

页框（块）的回收算法也较为简单，当进程执行完毕时，根据进程页表中登记的页框（块）号，将这些页框（块）插入存储页框表中，使之成为空闲页框（块）。最后，拆除该进程所对应的页表即可。

2. 页式地址变换

为了保证在非连续的存储区中正确地执行程序，操作系统必须提供一套地址变换机构，来完成逻辑地址到物理地址的转换，地址变换如图 5-13 所示。页式地址变换过程：首先用户进程提出存储分配的要求，此时操作系统根据内存页框的大小(1KB)将进程要求的内存空间分成相应的页面。

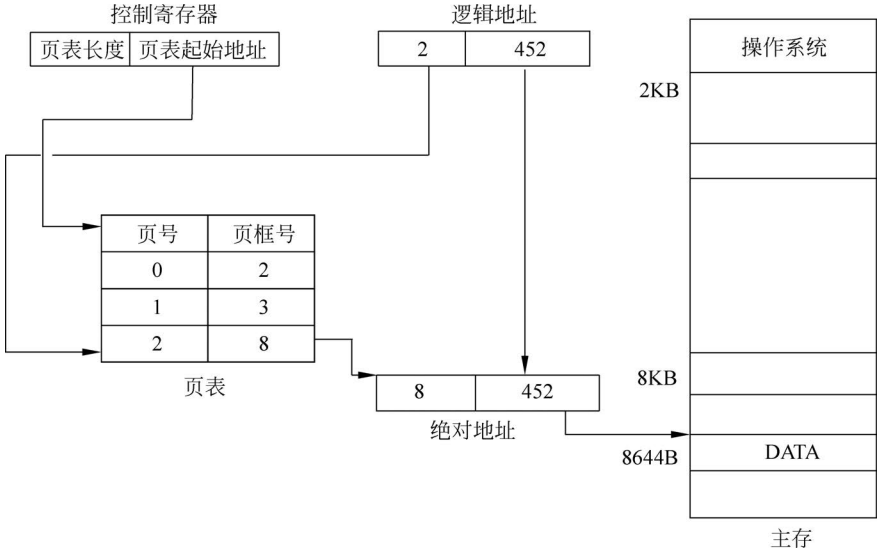


图 5-13　地址变换

(1) 根据内存的实际情况，将进程的每个页面分配到内存空闲页框中，同时，系统分配并设置页表的内容(通常一个进程具备一个页表)。此时，系统完成用户进程的内存分配。

(2) 当用户进程开始执行时，系统首先设置控制寄存器的内容，控制寄存器包括页表长度和页表起始地址两项。

(3) 为了对逻辑地址进行变换，由硬件组成的地址变换机构必须将其分成两部分：页号和页内偏移(即 2 和 452)。

(4) 根据逻辑地址中提供的页号在页表中找到相对应的页框号(2 → 8)。

(5) 将页表中的页框号和逻辑地址中的页内偏移分别写入绝对地址中的相应位置上(即 8 和 452)。

(6) 然后根据绝对地址提供的页号和页内偏移计算出内存空间的物理地址(1KB＝1024，8×1024＋452＝8644(B))。此时，用户进程便可以访问内存中的绝对地址，取出数据或取出指令执行。

综上所述，分页管理的地址变换机构简洁、清楚。但由于页表是存储在主存的某个固定的区域中，而每一个访问内存的绝对地址又必须通过页表变换才能得到，因此执行一条访问内存的指令都要访问内存两次：一次访问页表得到所要访问指令的绝对地址，另一次根据绝对地址访问实际所需的单元。这样，执行速度下降了一半。

3. 快表

为了尽量减少访问内存的次数，提高地址变换的速度，可在地址变换机构中增设一个具有并行查询能力的特殊高速缓冲存储器，用来存放页表的一部分。存放在高速缓冲存储器中的页表称为“快表”。“快表”中的每一个表项内容除了来自页表的相应表项内容外，还需增加有

利于组织“快表”的项目，如增加了有效位等。这种高速缓冲存储器又称为“相连存储器”。“相连存储器”实际上是一组硬件寄存器，它的存取速度比内存要快，具备一定的逻辑判断能力，可以实现按内容检索。

加入快表机构后，地址变换过程如下：CPU 在给出逻辑地址后，地址变换机构首先根据页号在快表中进行检索，若存在相应的页号，则直接从“快表”中读出该页号对应的页框号，形成物理地址；否则，需要再访问内存中的页表，从页表中读出相应的页框号，形成物理地址，同时将找到的页表项登记到“快表”中。当“快表”填满后，又要在“快表”中登记一个新的页表项时，则需采用一定的淘汰策略在“快表”中淘汰一个老的、已被认为不再需要的页表项。淘汰策略可以采用“先进先出”(FIFO)或“最近最少用淘汰法”(LRU)等，这些算法与后面介绍的页面置换(淘汰)算法相似，这里不再赘述。具有快表的地址变换过程如图 5-14 所示。

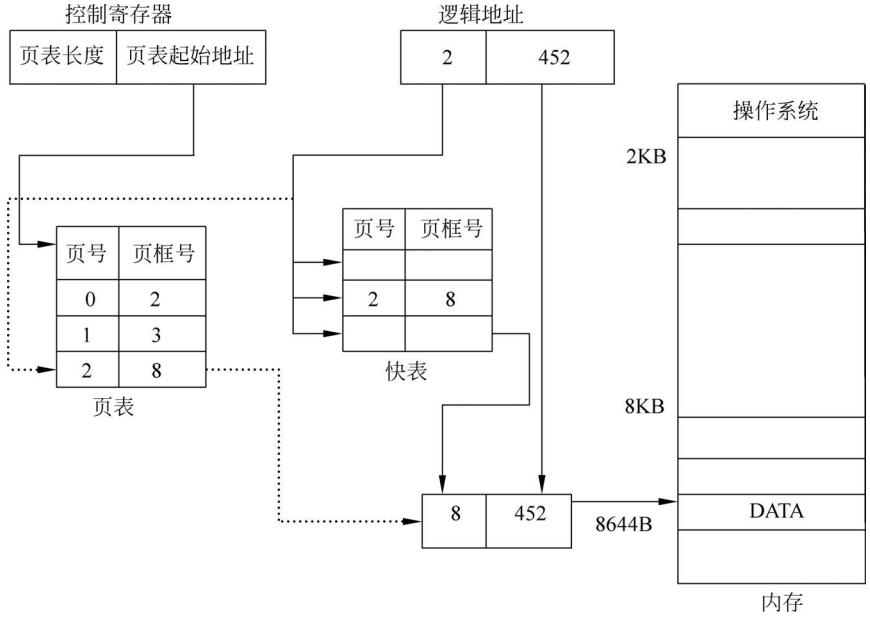


图 5-14　具有快表的地址变换过程

由于成本的关系，快表不可能做得很大，通常只能存放 64～256 个页表项，这对于中、小型作业来说，已有可能把全部页表项放在快表中，但对于大型进程，则只能将一部分页表项放入其中。由于对程序和数据的访问往往带有局部性，因此，采用一定的快表表目淘汰策略后，根据实际运行统计从快表中能找到所需页表项的概率(命中率)可达 90％以上，这使得由于增加了地址变换机构而造成的速度损失，可减少到 10％以下，达到了可接受的程度。

整个系统只有一个控制寄存器和一个相连存储器，只有占有处理器进程才占用控制寄存器和相连存储器。在多道程序系统中，某一进程让出处理器时，应同时让出控制寄存器和相连存储器。解决的办法是控制寄存器内容可作为进程的现场内容加以保护和恢复，而相连存储器的内容，为了使之与运行的进程一致，可采用两种方法来维护。一种最简单的办法是：在启动一个新的进程时，执行一条特定的机器指令使相连存储器的内容无效，即清除所有的有效位，这样，新的进程就可以使用相连存储器了。这种办法不利于提高快表的命中率。另一种解决办法是为相连存储器扩充一个进程或上下文标识符的域，同时为机器增加一个寄存器以保存当前进程的标识符。这样，硬件在查看快表时，不但要比较页号，也要比较进程标识符，是当前进程的页表项将被采用，不是当前进程的表项将被忽略。这种方法要增加额外的硬件，但节

省了上下文切换的时间，因为，如果相连存储器很大，把相连存储器也作为上下文进行切换，时间是相当可观的。

4．页的共享与保护

分页存储管理能方便地实现多个进程共享程序和数据。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、公共数据块等都是可以共享的，这些共享信息在内存中只要存放一份就行了。这些共享信息在内存中就形成了共享的页面。页的共享可大大提高内存空间的利用率。

在实现共享时，对数据的共享和程序的共享必须分别对待。实现数据共享时，可允许不同的进程对共享数据页使用不同的页号，只要让各自页表中的相关表目指向共享的数据信息块（页框）就可以了。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，共享程序必含有转移指令，这些转移指令的转移地址是确定的，所以情况就不同了，在程序运行前共享程序的页号必须是确定的。例如，假定有一个共享程序 EDIT，其中含有转移指令，转移指令中的转移地址必须指出页号和页内偏移，如果是转向本页，则页号与本页页号相同。现若有两个进程共享这个 EDIT 程序，假定一个进程定义它的页号为 3，另一个进程定义它的页号为 5，而在内存中只有一个 EDIT 程序，它要为两个进程以同样的方式服务，这个 EDIT 程序一定是可再入的（纯代码的），于是转移指令中的页号不能按进程的要求随机地改成 3 或 5。所以，对共享程序必须规定一个统一的页号。当然，对共享程序规定一个统一的页号是不自然的，实现起来也有一定的困难。

实现信息的共享必须解决共享信息的保护问题。可以在页表中增加一个保护权限域，用来指出该页的信息为可读/写、只读、只执行和不可访问等。指令执行时进行操作权限核对，若不合法，则停止执行，产生保护中断。例如，一指令要想向只读数据块写入信息，则指令将停止执行，产生中断。

另一种保护的方法是采用保护键法。系统为每个进程设置一个保护键，为进程分配主存时，根据它的保护键在相应的页表中建立键标志。进程执行时将程序状态字中的键和访问页的保护键进行核对，相符时才可访问该页框。为了使某些页框能被各进程访问，可规定保护键为“0”，此时不做核对工作。操作系统有权访问所有页框，可让操作系统程序的程序状态字中的键为“0”，规定程序状态字中的键为“0”也不进行核对。

静态分页管理解决了分区管理时的碎片问题。但是，由于静态分页管理要求进程在装入时必须一次性整体全部装入主存，如果当时系统中可用的页框数小于用户要求时，该进程只好等待，即进程的大小仍受主存中可用页框数的限制。为了解决这些问题，可采用虚拟页式存储管理技术来实现。

▶ 5.4.3　虚拟页式存储管理

随着现代计算机技术的迅速发展，用户程序的容量也随之增大。当系统在运行时，经常会出现内存容量不能满足用户程序的要求。例如，有的进程的逻辑地址空间很大，它要求的内存空间已超过了内存的总容量，进程不能全部装入内存中，致使该进程无法执行；有的系统运行时进程很多，内存无法一次将全部进程装入内存中，因而只能装入少量的进程，其他大量的进程留在外存中等待。显然，为了解决这一问题，通常最好的方式是从物理上扩充内存的容量，但是这必然将提高系统的成本，使用户无法接受；另一种方法是从逻辑上扩充内存的容量，这便是虚拟存储技术。

1. 虚拟存储的基本思想

1）常规内存管理方式的特征

（1）整体特性。整体特性是指用户进程在运行以前，必须将全部的内容一次装到主存中，这必然会导致内存容量不够；而且在大多数情况下，系统运行时并不要求使用用户进程的全部程序和数据，因而造成了内存空间的浪费。

（2）驻留特性。用户进程在装入内存运行过程中，将一直占据着内存的部分空间，即使是等待资源分配（例如因为I/O而长期等待），或有些程序段只运行一次，但它并不会释放所占据的内存空间，一直要等到用户进程运行结束。

由上面两个特性可以看到，在系统运行时，存在着大量不用或暂时不用的程序或数据段占据了内存的空间，使一些需要运行的程序段无法装入内存中。下面来看一下如何解决这一问题。

2）局部性原理

早在1968年Denning.P就提出过，程序的执行呈现出局部性规律，即在一较短的时间里，程序的执行仅局限在某个部分，相应地，它访问的内存空间也局限在某个区域。同时，他提出了以下几个论点。

（1）程序在执行时，除了少部分的转移和过程调用指令外，在大多数情况下仍是顺序执行的。

（2）过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域，但是经研究可看出，过程调用的深度在大多数情况下并不是很远。也就是说，程序将会在一段时间内都局限于某一范围内运行。

（3）程序中存在许多循环结构，它们虽由少数指令构成，但多次执行。

（4）程序中还包括许多对数据结构的处理，如对数组进行操作，它们都往往局限于很小的范围内。

通常，局部性又表现为时间局部性和空间局部性。

时间局部性表现在如果程序中某一条指令一旦执行，则在不久以后还可能被继续执行；同样，若某一个数据被访问后不久，还可能被继续访问。其典型的情况是程序中存在着大量的循环。

空间局部性表现在如果程序访问了某一个存储单元，其附近的存储单元则在不久后也会被访问。即程序在一段时间内访问的地址，可能集中在一定的范围内。其典型的情况是程序顺序执行。

3）虚拟存储器的基本思想

当用户进程要求的存储空间很大，不能被装入内存时，基于局部性原理，系统可以把当前要用的程序和数据装入内存中启动程序运行，而暂时不用的程序和数据驻留在外存中。在执行中需要用到不在内存中的信息时，通过系统的调入、调出功能和置换功能将暂时不用的程序和数据调出内存，腾出内存空间让系统调入要用的程序和数据。这样，系统便能很好地运行该用户进程了。从用户角度看，系统具备了比实际内存容量大得多的存储器，人们把这样的存储器称为虚拟存储器。

虚拟存储器是内存管理的核心概念。现代计算机系统中的物理存储器分为内存和外存，用户程序是放在内存中运行的。但由于内存价格较高，不可能一味地扩充内存空间来满足用户程序的需要。因此提出了虚拟存储器的概念，这使存储空间的逻辑容量可以由主存和外存

容量结合起来，其运行接近内存的速度，成本却没有大的增加。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，故被广泛地应用于各种计算机内存管理中。

4）虚拟存储器的特征

（1）多次性。多次性是指用户程序在运行前，并不是一次将全部内容装入内存中，而是在程序的运行过程中，系统不断地对程序和数据部分地调入、调出，完成程序的多次装入工作。

（2）对换性。程序在运行期间，允许将暂时不用的程序和数据调出内存（换出），放入外存的对换区中，待以后需要时再将它调入内存中（换入），这便是虚拟存储器的换入、换出操作，即对换性。

虚拟存储的多次性和对换性必须建立在离散分配的基础上。因为多次性允许将一个程序或数据分为多次调入内存。如果要求把它们装入一个连续内存区，必须事先就为它们申请足够大的内存空间，其中相当一部分空间都是空闲的，显然是对内存资源的极大浪费；而采用离散分配方式时，仅在需要调入某部分程序或数据时，才为其申请内存空间，这样就不会造成对内存的浪费。这就是把虚拟存储器建立在离散分配基础上的原因。

2．用分页技术实现虚拟存储器

1）数据结构

分页式虚拟存储系统中所需要的主要数据结构是页表。一种典型的分页式虚拟存储系统中的页表如图 5-15 所示，它是在分页系统的页表基础上增加了几项。

页框号	状态位	访问字段	修改位	保护权限	外存地址

图 5-15　分页式虚拟存储系统中的页表

（1）状态位：用于指示该页是否已调入内存，如用 1 表示在内存，0 表示不在内存。

（2）访问字段：用于记录本页在一段时间内被访问的情况，提供给置换机构参考。

（3）修改位：表示该页在调入内存后是否被修改过，由于内存中的每一页都在外存上保留一份副本，因此，若未被修改，在置换该页时就不需将该页写回到外存上；若已被修改，则必须将该页重写到外存上，以保证外存中所保留的始终是最新副本。

（4）保护权限：说明该页允许什么类型的访问。它指出了该页的信息可能为可读/写、只读、只执行和不可访问等。

（5）外存地址：指出该页在外存上的地址，供调入该页时使用。

现代计算机系统中程序的逻辑地址空间很大，因此，页表可能也非常大。以 Windows NT 为例，它使用了 32 位虚拟地址，每个进程可以有 $2^{32}=4$GB 的虚拟地址空间，使用 $2^{12}=4$KB 的页面，这就意味着一个进程最多可以使用多达 $2^{20}=1$M（100 多万）个页面。如果每个页表表目占 4B，那么每个进程的页表所占的空间是 $2^{22}=4$MB，占去了相当大的内存空间，显然这样大的页表是不能全放在内存中的。为此，对页表本身也采用分页措施，即把页表本身按固定大小分成为一个个页面，每个小页表形成的页面中可以存放 $2^{10}=1$K 个页表表目，共有 $2^{10}=1$K 个小页表。为了对这 1K 个小页表进行管理和索引查找，设置了一个页表目录或页目录，该页表目录称为顶级页表，它包含 1K 个页表表目，分别指出每一个次级小页表所在物理页框号和其他状态信息。这样，每个进程将有一个页目录，它的每个表目指向一个页表。页目录本身大小恰好是一个页面大小。页目录是一级页表，而每一个小页表就是二级页表。具有二级页表的地址变换过程如图 5-16 所示。

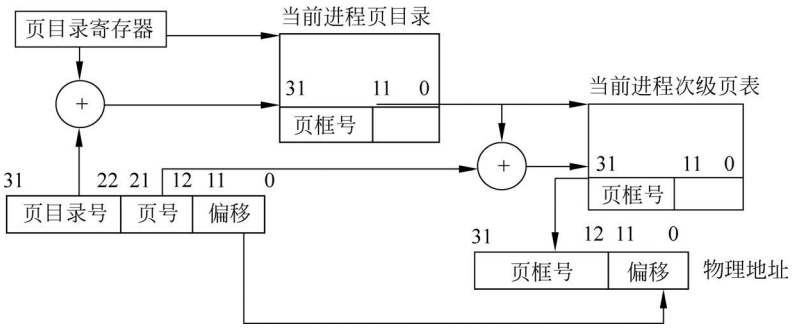


图 5-16　具有二级页表的地址变换过程

通过二级页表地址映射访问内存存取数据需要三次访问内存，一次访问页目录，一次访问页表，最后才访问数据所在的物理地址，显然影响了存取速度。当系统虚拟地址为 64 位时，还可以组成三级或四级页表，但性能的影响是不可忽视的。

随着 32 位、64 位计算机的出现，程序的虚拟地址空间很大，而物理页框数相对来说是较少的，因此有些系统（如 IBM RS/6000，HP Spectrum）就不以虚存而以物理内存来组织页表。这种页表的第 i 个表项记录着当前占用页框 i 的页面信息，表中的表项号与物理内存的页框号相等，而与虚存的页号无关，该页表称为反向页表。反向页表如图 5-17 所示。在这种系统中，当进程给出虚地址后，存储管理单元（MMU）通过一个哈希函数转换为一个哈希值，以该值为索引指向反向页表中的一个表目，若表目中的内容与该进程的虚页号相一致，即形成绝对地址，否则再到链指针中查找。

	页号	进程标识	标志位	链指针
页框0	34	P_1	1	
页框1	23	P_2	0	
页框2	78	P_1	1	
⋮	⋮	⋮	⋮	⋮

图 5-17　反向页表

2）分页式虚拟存储管理工作流程

虚拟存储器的实现必须依靠一定的物理基础，即硬件环境。

（1）具有一定容量内存，用于存放一个操作系统，以及每个进程的部分程序、数据和相应的页表表项。

（2）相当容量的外存，用于存放每个进程未装入内存的部分、后备进程以及大量的文件。

（3）地址变换机构，用于将用户程序地址空间中的逻辑地址变换为内存的物理地址。

（4）缺页中断机构，当发现所要访问的页不在内存时，应立即发出缺页中断信号，以请求操作系统将所缺之页调入内存。

这里必须注意，缺页中断是一种特殊的中断，它与一般中断的区别在于：

（1）在指令执行期间产生和处理中断信号。通常 CPU 都是在一条指令执行完后检查是否有中断请求到达，若有，便去响应，否则继续执行下一条指令。然而，缺页中断是在指令执行期间发现所要访问的指令或数据不在内存时产生和处理的。

（2）一条指令在执行期间，可能产生多次缺页中断。如采用多级页表时，页表也是动态调入的，地址翻译过程就可能出现访问页表的缺页中断，最后才产生所要访问的页的缺页中断。

分页式虚拟存储管理总的工作流程是：首先，为用户进程分配内存工作区并填写相应的页表项目；接着，由进程调度程序调度用户进程执行。进程在执行中访问某页时，硬件地址转换机构先查看快表，若在快表中命中，则立即形成绝对地址，否则再查看页表，若该页对应的状态位为 1，表明该页已在内存，即根据页表内的页框号形成访问内存的绝对地址，将该页的信息登记到快表中。若该页对应的状态位为 0，表明该页不在内存，则由硬件产生一个缺页中断。操作系统内核必须处理这个缺页中断，处理的办法是先查看内存是否有空闲的页框，若有则按该页在外存的地址将该页读出并装入内存，在页表中填上它占用的页框号且修改状态位。若内存已没有空闲页框，则必须调出已在内存中的页，再将所需的页装入，对页表和存储页框表做相应修改。为了提高系统效率，在访问某页时，若是执行写指令，则在页表中相应页的修改位上置 1。这样在选择某页调出时，必须看其修改位是否为 1，若修改位为 1，就将该页写回外存中，否则不必把该页重新写回外存中。分页式虚拟存储管理工作流程如图 5-18 所示。

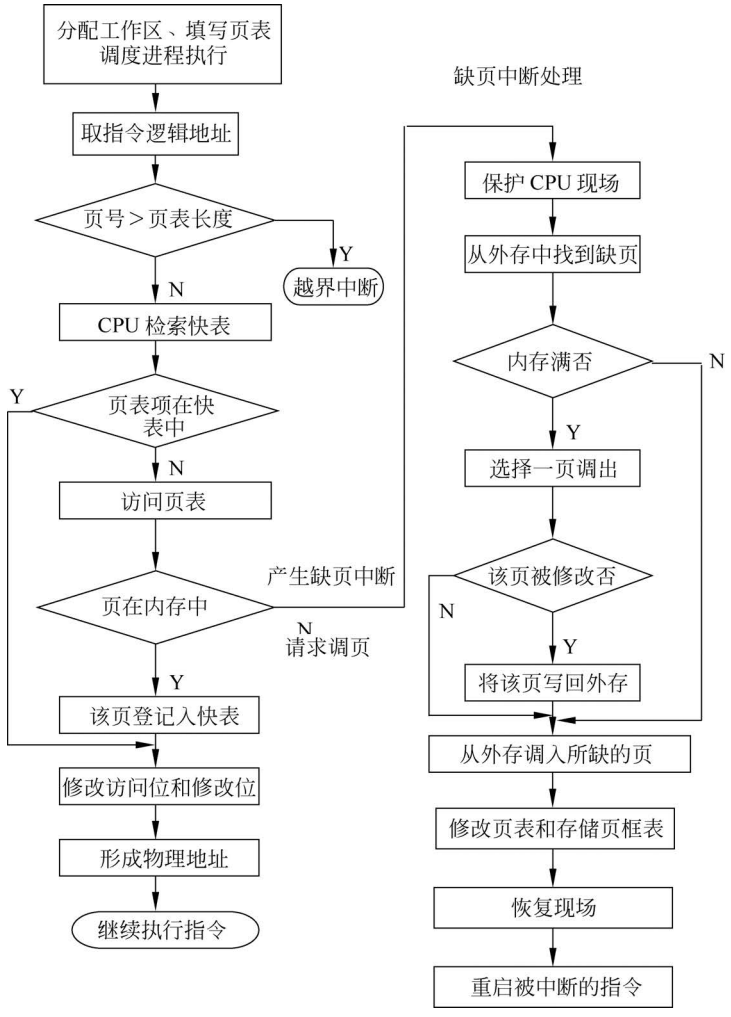


图 5-18　分页式虚拟存储管理工作流程

由于产生缺页中断时，一条指令并未执行完，所以操作系统内核在进行缺页中断处理后，应重新执行被中断的指令。当重新执行该指令时，可能由于要访问的页已经装入内存，所以就可以正常执行下去了。

3. 页面置换

1）调页策略

由上面介绍的分页式虚拟存储管理工作流程，可以看出系统必须动态地将所需的页面调入内存中。那么，系统应在何时及采用什么策略将进程所需的程序和数据页面调入内存呢？目前有两种调度页面的策略：预调页策略和请求调页策略。

（1）预调页策略。

预调页策略的优点在于当在外存上查找一页时需经历较长时间，如果进程的许多页是存放在外存的一个连续区域中，一次调入若干相邻的页，要比每次调入一页更高效。但如果调入的一批页面中的大多数都未被访问，则又是低效的。可见，该策略应以预测为基础，只将那些预计不久后便会被访问的程序或数据所在的页面预先调入内存。如果预测较准确，这种预调页策略是可取的。但遗憾的是，目前预调页的成功率只达到50%左右。故该策略主要用于进程的首次调入，这时程序员必须指出应该先调入哪些页。在有的操作系统中，也将预调页策略用于请求调页存储中，例如，在VAX计算机的VMS操作系统中，采用了一种称为群集式调页策略，当系统将进程所请求的页面调入内存时，也同时将其相邻的几个页面调入内存。

（2）请求调页策略。

当进程运行中需要访问某部分程序和数据，而其所在页面又不在内存时，立即提出请求，由系统将其所需页面调入内存。请求调页策略比较易于实现，故在目前的页式虚拟存储系统中，大多采用此策略，该类系统也称为请求分页系统。但这样需要较大的系统开销，因为每次请求后只调入一页。

一个进程被调入系统执行中，其所需信息目前如果不能驻留在内存，则必然存储在外存中。那么系统应从外存的何处将所需页面调入，视不同情况而定。

（1）从硬盘文件区调入。因为进程的程序和数据原来都是作为文件而放在文件区的，因此，对于凡是从未运行过的页面，都应从文件区调入。

（2）从硬盘对换区调入。对于曾经运行但是又被换出（调出）的页面，是被放在对换区的，当需要再次将它调入时，显然应从对换区调入。

（3）向内存中的页面缓冲池索取。由于页面可能是共享的，则进程所请求调入的页有可能已经被调入内存，显然这时可直接从页面缓冲池中找出该页并取出，这样，不仅提高了调页的速度，也可避免单为该页启动磁盘。

2）分配策略

在请求分页系统中可采取以下两种分配策略。

（1）固定分配策略。基于进程的类型（交互型或批处理型等），或根据程序员、系统管理员的建议，每个进程分配一固定页数的内存空间，在整个运行期间不再改变。采用该策略时，如果进程运行中发现缺页，则只能从该进程在内存的几个页面中选出一页调出，然后再调入一页，以保证分配给该进程的内存空间不变。这种分配方式的困难在于：应为每个进程分配多少个页框的内存难以确定。若太少，会频繁地出现缺页中断，降低了系统的吞吐量；若太多，内存中能驻留的进程数目必然减少，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）可变分配策略。同样基于进程的类型或根据程序员的要求，为每个进程分配一定数量的内存空间，如果该进程在运行过程中频繁地发生缺页中断，则系统再为该进程分配

若干附加的物理页框，直至进程减到适当的缺页率为止；反之，若一个进程在运行过程中的缺页率特别低，则此时可适当减少分配给该进程的物理页框，但不应引起其缺页率明显增加。

3）页面置换算法

进程在运行过程中，若其所访问的页面不在内存而需将它调入内存，但内存已无空闲空间时，为了保证该进程能继续运行，系统必须从内存中调出一页程序或数据到磁盘的对换区中，这一工作称为页面调度或页面置换。页面置换实际上是确定淘汰哪一页的问题。但应将哪个页面调出（淘汰）呢？从理论上讲，应该将那些以后不会再访问的页面调出，或将在较长时间内不再访问的页面调出。但是，要实现这样一个调度算法确实是很难的。目前存在着许多种置换（调度）算法，它们都试图更靠近这个理论上的目标。因此，选择一个好的置换（调度）算法是很重要的，如果选用了一个不合适的算法，就会出现这样的现象，刚被淘汰的页面又立即要用到，因此又要把它调入，而调入不久再被淘汰，淘汰不久又再被调入。如此反复，使得整个系统的页面置换非常频繁，以至于大部分时间都花费在来回调度上。这种现象称作“抖动”或“颠簸”。

为了衡量置换算法的优劣，一般均是在页面固定分配策略的前提下考虑各种置换算法的。算法好坏的一个重要衡量指标是缺页中断率。何谓缺页中断率呢？可以这样给它下一个定义：假定进程 P 共计有 n 页，而系统分配给它的主存只有 m 个页框（m、n 均为正整数，且 $1\leqslant m\leqslant n$），即最多只能容纳 m 页。如果进程 P 在运行中访问的页在内存（成功的访问）的次数为 S，不成功的访问次数为 F（即缺页中断次数），则缺页中断率为 $f=F/(S+F)$。

下面分别介绍几种典型的页面置换算法，并约定这些算法均是在请求调页策略下进行的。

（1）优化算法（Optimal replace algorithm，OPT）。

这是一种理论化的算法，其所选择的被淘汰的页将是永不使用的页，或者是在最长时间内不再访问的页。要真正做到这一点是困难的，故它也不是很实际的算法。但可将该算法作为衡量其他各种实际算法的标准。

（2）先进先出算法（First In First Out，FIFO）。

这是最早出现的置换算法。该算法总是淘汰最先进入内存的页面，即选择内存中驻留时间最久的页面予以淘汰。该算法实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指向最老页面的替换指针即可。但该算法是基于 CPU 按线性顺序访问地址空间的假设的。实际上，很多时候，CPU 并不是按线性顺序访问地址空间的，例如，进程执行循环语句时。因此，那些在内存中驻留时间最长的页往往也是经常被访问到的。所以，该算法与进程实际运行的规律不一定相适应。

先进先出算法的另一个缺点是会出现一种奇异现象——Belady 现象。一般情况下，对于一个作业，如果分配给它的内存页框越多，缺页中断率就越低，反之就越高。但是，对 FIFO 算法来说，在未给进程分配足够满足它要求的页面数时，有时会出现分配的页框数增多，而缺页中断率反而增高的奇异现象，这种现象称为 Belady 现象。

下面举例来说明 FIFO 算法的正常调页情形和出现的 Belady 现象。

设进程 P 共有 8 页，主存分配 3 个页框给它使用，程序访问页面的顺序为 7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1。FIFO 算法执行情况如表 5-2 所示，表中记录了进程 P 执行过程中内存的页面号及缺页中断和淘汰的页面号。

表 5-2　FIFO 算法执行情况(1)

访问次序	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
内存页号	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1
			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2
淘汰				7		0	1	2	3	0	4			2	3		

由表 5-2 可以看出,进程 P 在执行过程中发生了 12 次缺页中断(包括在内存有空闲时的 3 次缺页中断)。此时,缺页中断率为 12/17≈70.6%。

如果给进程 P 分配 4 个页框,则 FIFO 算法执行情况如表 5-3 所示,表中记录了进程 P 在执行过程中主存的页面号及缺页中断和淘汰的页面号。

表 5-3　FIFO 算法执行情况(2)

访问次序	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
内存页号	7	7	7	7	7	3	3	3	3	3	3	3	3	3	2	2	2
		0	0	0	0	0	0	4	4	4	4	4	4	4	4	4	4
			1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
				2	2	2	2	2	2	2	2	2	2	1	1	1	1
淘汰						7		0			1			2	3		

由表 5-3 可以看出,进程 P 在执行过程中发生了 9 次缺页中断(包括在主存有空闲时的 4 次缺页中断)。此时,缺页中断率为 9/17≈52.9%。

以上是 FIFO 算法正常调页的情形。下面再来看另一种访问顺序的情况。设进程 P 的访问顺序为 1,2,3,4,1,2,5,1,2,3,4,5。当进程 P 分配 3 个页框时,FIFO 算法的 Belady 现象如表 5-4 所示,表中记录了进程 P 执行过程中内存的页面号及缺页中断和淘汰的页面号。

由表 5-4 可以看出,进程 P 在执行过程中发生了 9 次缺页中断(包括在主存有空闲时的 4 次缺页中断)。此时,缺页中断率为 9/12=75%。

表 5-4　FIFO 算法的 Belady 现象(1)

访问次序	1	2	3	4	1	2	5	1	2	3	4	5
内存页号	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4

当进程 P 分配 4 个页框时,FIFO 算法的 Belady 现象如表 5-5 所示,表中记录了进程 P 在执行过程中主存的页面号及缺页中断和淘汰的页面号。此时,进程 P 在执行过程中共发生了 10 次缺页中断,缺页中断率为 10/12≈83.3%。即给进程分配更多页框还会使缺页中断更高,这与我们的直观感觉不符,这种现象称为 Belady 现象。

表 5-5　FIFO 算法的 Belady 现象(2)

访问次序	1	2	3	4	1	2	5	1	2	3	4	5
主存页号	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3

FIFO 算法产生 Belady 现象的根本原因是它没有考虑到程序执行的动态特征。

（3）最近最少用置换算法（Least Recently Used，LRU）。

该算法要求淘汰的页面是在最近一段时间里较久未被访问的那一页。它是根据程序执行时所具有的局部性来考虑的，即那些刚被访问过的页面可能马上要用到，而那些在较长时间里未被访问的页面，一般说来，可能不会马上使用到。

为了比较准确地淘汰最近最少使用的页面，可以采用堆栈的方法来实现。栈中存放当前主存中的页号，每当访问一页时就调整一次栈，使栈顶总是指出最近访问的页，而栈底就是最近最少使用的页号。于是，发生缺页中断时总是淘汰栈底所指示的页。

例如，给进程 P 固定分配 4 个内存页框，进程的虚拟地址空间有 7 页，其执行期间访问的页面号顺序也为 7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1。堆栈的方式 LRU 算法执行情况如表 5-6 所示，表中记录了进程 P 执行过程中内存的页面号及缺页中断和淘汰的页面号。

由表 5-6 可以看出，进程 P 在执行过程中发生了 7 次缺页中断（包括在内存有空闲时的 4 次缺页中断）。此时，缺页中断率为 7/17≈41.2%。

表 5-6　堆栈的方式 LRU 算法执行情况

访问次序	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
内存页号				2	0	3	0	4	2	3	0	3	2	1	2	0	1
			1	1	2	0	3	0	4	2	3	0	3	2	1	2	0
		0	0	0	1	2	2	3	0	4	2	2	0	3	3	1	2
	7	7	7	7	7	1	1	2	3	0	4	4	4	0	0	3	3
淘汰						7		1						4			

由该例子可以看出，堆栈方式的 LRU 算法，淘汰的页面确实是较长时间以后才使用到的，该例的 LRU 算法执行性能已经达到了 OPT 算法的性能，故 LRU 算法是一种较为有效的页面置换算法。但调整堆栈是非常费时的，所以有些系统常常采用一些特殊的硬件实现这种算法。

一种硬件的实现办法是采用一定位数的计数器，它每执行完一条指令后计数器就加 1，每一个页表必须设置一个能容纳该计数器值的域——访问字段，在每次访问内存后，就把当前计数器的值保存到访问字段中。一旦发生缺页，系统就检查页表中的所有访问字段值，找出该值最小的页，把该页淘汰，该页就是最久未使用的页。

另一种硬件实现办法是：假设内存有 n 个页框，硬件就维持一个 $n \times n$ 位的矩阵，开始时所有的位都是 0。当访问到页 k 时，硬件首先把 k 行的位都置成 1，再把 k 列的位都置成 0。当发生缺页时，就选择该矩阵中二进制最小的行所对应的页淘汰。显然，该页也是最久未使用的页。

（4）最近未用置换算法（Not Recently Used，NRU）。

这是 LRU 算法的一种退化算法，该算法要求页表中有一个访问位和一个修改位。当某页被访问时，访问位被自动置 1，若执行的指令是写指令，则修改位也被置 1。系统周期性地（设周期时间为 T）将所有访问位置 0。在选择一页来淘汰时，总是选择其访问位为 0 且修改位也为 0 的页淘汰。若无修改位为 0 的页，就选择访问位为 0 且页号最小的页淘汰。由此可见，该算法不但希望淘汰的页是最近未使用的页，而且希望被淘汰的页是在主存驻留期间其页面内容未被修改过。这种算法实现代价小，但系统对访问位清 0 的间隔时间 T 的确是很关键的。如果间隔时间 T 太大，可能所有页的访问位均已成为 1，无法选择淘汰的页面。如果间隔时间 T 太小，则可能很多页的访问位均是 0，同样也很难有效地确定淘汰的页面。

(5) 最少使用置换算法(Least Frequently Used,LFU)。

该算法要求为每一页表项配置一个一定位数的计数器作为访问字段,开始时所有的计数器均为0。一旦某页被访问时,其页表项中的计数器值加1。系统每过一段时间 T 就将所有的页表项计数器清0。在需要选择一页置换时,便比较各计数器的值,总是选择其计数值最小的页面淘汰,显然它是最少被使用的页面。该算法实现也较容易,但代价较高,而且合适的间隔时间 T 的选择也是一个难题。

(6) 第二次机会算法(Second Chance)。

该算法思路的基本出发点是淘汰不但是"老"的,而且还是最近"没用"的页面。

算法实现原理如下。

① 用链表来表示各页的建立时间先后,新来的放到表尾,表头就是最"老"的(同FIFO),页面装入或被访问时设 $R=1$。

② 选择淘汰页面时,若表头页面的 R 位(访问位)是0,则淘汰之,否则将其 R 位设为0,并把它放到表尾,然后继续从表头搜索。链表结构如图5-19所示,开始选择时,页面A的 R 位被置1,并被放在表尾。

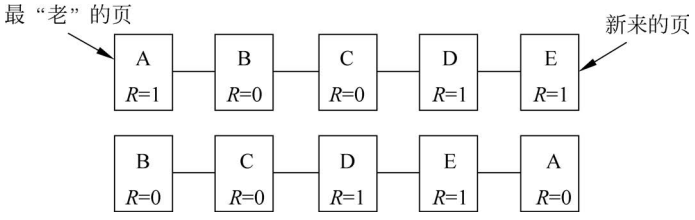


图5-19　链表结构

(7) 时钟算法(Clock)。

基本思想是用环形链表实现第二次机会算法,达到淘汰最"老"并且最近"没用"的页面的目的。

算法要点如下。

① 用环形链表来表示各页的建立时间先后,表头表尾相邻,因此选择淘汰页面过程只需要移动链表指针。

② 该算法性能近似LRU,实现开销小。

在内存的页面环形链表如图5-20所示。某进程调入页面27前在内存的各页面状况如图5-20(a)所示,由于链表指针所指的前两个页面(页面25和91) R 位均为1,不能将其淘汰,应将其 R 位均置为0,而接下来的页面56其 R 位为0,可将其淘汰并置换为页面27,并置 R 位为1。置换后的链表情形如图5-20(b)所示。

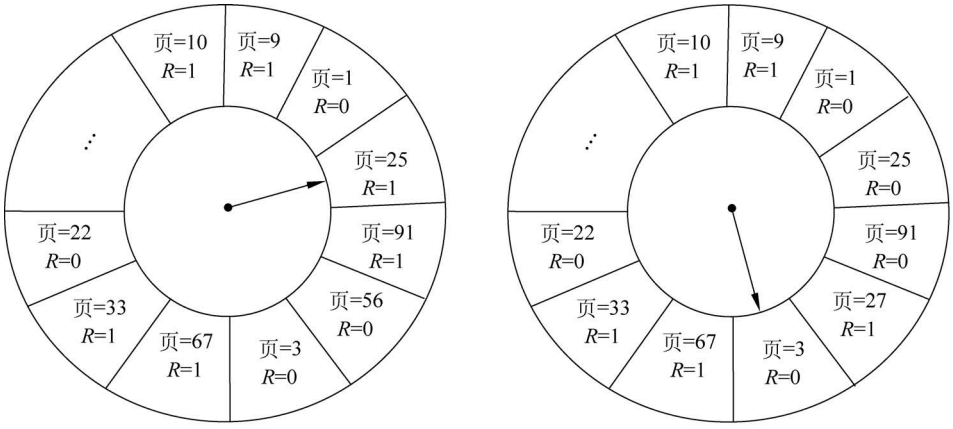
(8) 基于Clock的NRU算法。

该算法用位 R 表示页面的访问状况,用位 M 表示页面的修改状况,它要淘汰并置换的页面不仅是最"老"且最近"没用"的页面,而且也是最近"没修改过"的页面。

算法要点如下。

① 从指针位置开始扫描链表,扫描过程中不改变 R 位。淘汰遇到的第一个 $R=0\&M=0$ 的页面。

② 若第①步失败,则再次扫描,淘汰遇到的第一个 $R=0\&M=1$ 的页面。每个页面检查过后将 R 设为0。



(a) 页面置换前的内存页状况　　(b) 页面置换后的内存页状况

图 5-20　在内存的页面环形链表

③ 若第②步失败，可再次重复①和②。

4. 分页式虚拟存储系统的性能分析

分页式虚拟存储系统摆脱了内存实际容量的限制，能使更多更大的进程同时多道运行，从而提高系统的效率。但这类系统必须进行缺页中断处理，而缺页中断处理是要付出相当的代价的，不仅由于页面的调入、调出要花费较多的 I/O 时间，而且影响了系统的执行效率。因此，分页式虚拟存储系统应尽量降低缺页中断率。由前面介绍的页面置换算法可知，缺页中断率与置换算法、分给进程使用的内存页框数有密切关系，实际上，缺页中断率还与程序编制的质量、页面大小等有关。

为了降低缺页中断率，提高系统性能，分页式虚拟存储系统的实现方法都要求内存中应能存入不低于一定限度的程序和数据页，而且它们必须是那些正在被使用或即将被使用的部分。这就使得缺页中断次数减少到最低程度。

根据程序执行的局部性，Denning 认为在某段时间内较好地确定正在被使用或即将被使用的页面是可能的，因此，他提出了工作集理论。所谓工作集，简单地说，就是进程在某段时间内实际上要访问的页的集合。Denning 认为一个程序要高效地运行，其工作集必须在内存中。但是，如何确定一个进程在某个时间的工作集呢？计算机是无法预知用户程序的行为的，因此，系统仍然要依据程序的过去行为来估计它未来的行为，这种估计依据就是程序行为的局部性特征，决定了工作集的变化是缓慢的。所以，把一个运行进程在 $t-w\sim t$ 这个时间间隔内所访问的页的集合称为进程在时间 t 的工作集，记为 $WS(t,w)$。并把变量 w 称为工作集窗口大小，把工作集中所包含的页面数目称为工作集大小，记为 $|WS(t,w)|$。

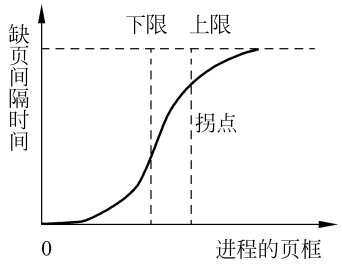


图 5-21　缺页间隔时间-内存页框数曲线

正确地选择工作集窗口的大小对工作集存储管理策略的有效工作是有很大影响的。若 w 过大，甚至会把整个进程的地址空间都包含在内，这样就失去了虚存的意义，若 w 过小，则将引起频繁缺页，降低系统效率。所以不少学者认为：进程的工作集大小可粗略地看成对应于“缺页间隔时间-内存页框数”曲线的拐点，缺页间隔时间-内存页框数曲线如图 5-21 所示。这样就可以确定出 w，只要在某时刻 t_1 起，将所有页面的访问位全清 0，然后访问一页时，置该页访问位为 1，到时刻 $t_2=t_1+w$ 时，所有访问位为 1 的页的集合就是时刻 t_2 的工

作集。

为了提高系统效率，即降低缺页中断率和提高系统的吞吐量，应使缺页间隔时间保持在一个合理的水平。若间隔时间过小时，应增加分配给进程的页框数；若过大则减少分配给进程的页框数，增加运行程序的道数。

许多系统都采用工作集概念来确定进程驻留主存的页框数，以进行页框分配工作，并以工作集概念来监视和控制运行进程的缺页中断率。

程序的局部性对降低缺页中断率、提高系统的效率也是很重要的。一般说来，要求编制的程序具有较高的局部化程度，如程序编制成结构化程序，对数据的组织和处理采用线性结构、成片处理等。这样，程序在执行时可经常集中在几个页面上进行访问，减少缺页中断次数。

设计分页式虚拟存储系统时，页面的大小也是一个值得关注的问题。页面大所需的页表表目较少，这样页表占用的内存就少，页表查找速度快。此外，缺页中断次数也相应少些。但是，在页面调度时，一次换页的时间较长，页内零头空间浪费的可能性较大。页面小时，它的利弊正好相反。一般说来，页面大小应根据实际情况来确定，它和计算机的性能、程序设计技术及用户的要求都有关系。特别是现代计算机系统中引入了面向对象的程序设计技术和多线程机制，使得进程访问内存具有较大的分散性，因而局部性下降了。若还采用小页面，则进程执行时，在快表中的命中率和访问内存的成功率将下降。一种折中的办法是：增加页面大小。所以，现代计算机系统大多使用较大的页面，如使用 4KB、8KB 的页面等。

综上所述，要使分页式虚拟存储系统有较好的性能，必须选择有效的页面置换算法，合理地控制分配给进程使用的内存页框数，切合实际地设计页面大小，同时也要求用户编制的程序有较高局部性。

▶ 5.4.4　分页存储管理的优缺点

由于分页存储分配有效地解决了存储器的零头问题，因而能同时为更多的进程提供内存存储空间，能在更高的程度上实现多道程序设计，从而相应地提高了内存和 CPU 的利用率。分页系统具备如下优点。

(1) 解决内存的零头问题，能有效地利用内存。

(2) 方便多道程序设计，并且程序运行的道数增加了。

(3) 可提供大容量的虚拟存储器，进程的地址空间不再受实际内存大小的限制。

(4) 更加方便了用户，特别是大进程的用户。当某进程地址空间超过主存空间时，用户也无须考虑覆盖结构。

同时也必须指出，分页存储管理方式也有不少的缺点。

(1) 要求有相应的硬件支持，如需要动态地址变换机构、缺页中断处理机构等，增加了计算机的成本。

(2) 必须提供相应的数据结构来管理存储器，而这些数据结构不仅占用了部分内存空间，同时它们的建立和管理要花费 CPU 的时间。

(3) 虽然解决了分区管理中区间的零头问题，但在分页系统中页内的零头问题仍然存在。

(4) 对于静态分页管理系统，用户进程应一次性装入内存，将给用户进程的运行带来一定的限制。

(5) 在请求分页管理中，需要进行缺页中断处理，特别是请求调页的算法若选择不当，还有可能出现抖动现象，增加了系统开销，降低系统效率。

5.5 段式及段页式存储管理

▶ 5.5.1 段式存储管理

1. 段式存储管理概述

在分页式存储管理中，要求用户程序的逻辑地址空间是连续的。这就要求编译程序对用户源程序进行编译、连接时，必须把主程序、子程序、数据块等按线性空间的一维地址顺序装配起来。因此，装配好的程序段和数据块的存储空间是确定的，在执行中是无法动态增长和收缩的，这就造成了用户程序设计的不灵活、不方便。此外，由于对各子程序和数据段的顺序装配，分页时，无法做到页与逻辑意义完整的子程序或数据段的唯一对应，增大了其信息共享实现的难度。再者，从连接的角度上看，分区管理和分页管理只能采用静态连接。通常，一个大的程序可能包含数百个甚至上千个程序模块，而执行时，可能仅用到其中的一部分模块，许多模块并不需要访问。静态连接必须对这些众多模块一次性地连接装配，不仅花费了大量的 CPU 时间，而且浪费了许多内存空间（至少多浪费了一些页表空间）。

为了克服分页式存储管理的上述不足，因此，人们提出了采用分段的存储管理思想：把程序按逻辑含义或过程（函数）关系分成段，每段都有自己的名字，这个名字称为段名。用户程序可用段名和入口指出调用一个段的功能，程序在编译或汇编时，再将段名定义一个段号。每段逻辑地址均是以 0 开始进行顺序编址。这样用户作业或进程的地址空间就形成了一个二维线性地址空间，任意一个地址必须首先指出段号，其次再指出段内偏移地址。段式存储管理程序以段为单位分配主存，然后，执行时通过地址转换机构把段式逻辑地址转换成内存物理地址。

2. 段式存储管理实现原理

分段式存储管理把用户进程按逻辑意义分成若干段，每段均是从 0 开始编址，段内地址是连续的，而段与段之间的地址不连续。因此，进程的逻辑地址形式为

段号	段内地址

一旦地址结构确定了，那么这个系统中一个进程允许的最多段数和每段的最大长度就确定了。例如，某系统段地址结构为 32 位，其中，段号占 12 位，段内地址占 20 位，则在该系统中一个作业最多可有 4K 个段，每段的长度可达 1MB。

分段式存储管理为每个进程的每一段分配一个连续的内存空间，而各段之间可以不连续，这种分配方式类似于动态分区管理的多个进程的分配。内存的分配与回收算法类似于动态分区管理的分配与回收算法，在此就不再叙述。

为了登记各进程的主存分配情况，每个进程必须建立一张段表，由它指出每段在内存中的起始地址和长度。一个进程各段在内存中的分配如图 5-22 所示。

段表表目实际上起着基址/限长寄存器的作用，一个进程执行时，通过段表即可将逻辑地址转换成绝对地址。类似于分页式存储管理，系统中也设置了一个段表控制寄存器，用来存放当前占用处理器的进程的段表始址和长度。段式管理地址变换过程如图 5-23 所示。

与页式管理相同，段式管理的一次访问内存也必须经过两次以上访问内存的操作。为了提高访问速度，也需要将高速相连存储器引入，把部分段表存入其中，形成段式快表。地址转换时，先查快表，若快表命中，则立即形成绝对地址，否则再通过段表进行慢地址翻译，并将该段信息填入快表中。

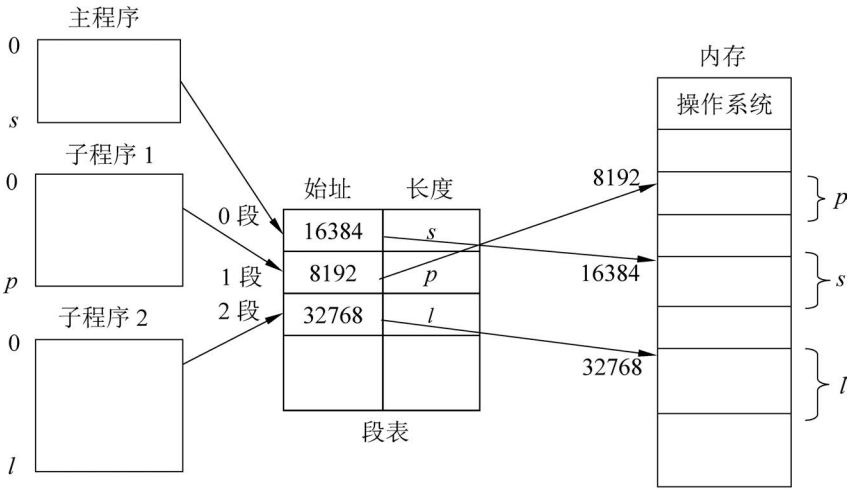


图 5-22　一个进程各段在主存中的分配

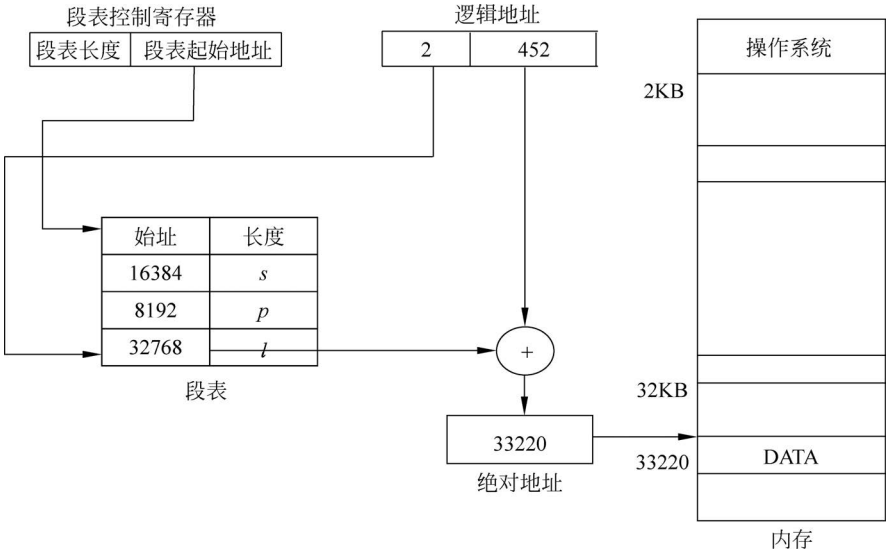


图 5-23　段式管理地址变换过程

分段式存储管理可方便地实现主存信息共享和保护。

如果用户作业需要共享内存中的某段程序或数据时，只要用户使用相同的共享段名，系统在建立段表时，只需在相应的段表栏目上填入已在内存的段的始址和长度，即可实现段的共享，从而提高系统内存的利用率。

在实现段的共享时，必须采取一定的保护措施。可在段表中增设一个存取权限域。存取权限可分为只执行(共享程序段)、只读(共享数据段)和可读/写(私人段)。访问段时，通过存取权限核对，即可实现存取保护。此外，通过段表中的长度信息，在地址转换时，将长度与段内地址比较，就可进行地址越界保护。

由上面的地址转换过程可以看出，段式与页式管理是很相似的。但必须注意这两者在概念上的不同。分段是信息的逻辑单位，是用户可见的，段的大小是用户程序决定的。而分页是信息的物理单位，分页对用户来说是不可见的，页的大小是事先固定的。

▶ 5.5.2 段式虚拟存储管理

1. 段式虚拟存储管理实现原理

分段式也可实现虚拟存储管理，为用户提供比内存实际容量大的存储空间。段式虚拟存储系统的基本思想是：把进程的所有分段的副本都存放在外存上，当进程被调度投入运行时，首先把当前需要用的一段或几段装入内存，在执行过程中，访问到不在内存的段时，再通过缺段中断机构把它从外存上调入。

因此，段表必须在静态段式存储管理的基础上加以扩充。如在段表中必须附加说明哪些段已在内存，哪些段不在内存，各段在外存中的起始地址，相应的段是否已被修改过，段是否可移动、可扩充，段能否共享等。一个典型的段表格式如图 5-24 所示。

	特征位	存取权限	标志位	扩充位	访问位	主存始址	长度	外存始址
0段								
1段								

图 5-24 段表格式

段表中，特征位可用两位表示相应的段是否在内存，是否可共享；存取权限用两位表示相应段只执行、只读或可读/写的权限；标志位用两位表示相应段是否已被修改过和能否移动；扩充位用一位表示相应段是固定长还是可扩充；访问位用来表示段的活动状况，作为淘汰段时参考；内存始址为段在内存的开始地址；长度为段的地址单元数；外存始址登记该段在外存中副本的起始地址。

在进程执行中访问某段时，由硬件地址转换机构查快表和段表，若快表命中或该段在内存，应核对存取权限，若存取合法，则按静态段式存储管理的地址转换办法得到绝对地址；若存取违法，则发保护中断，报告存取违法并停止程序执行。若该段不在内存，则由硬件产生一个缺段中断，操作系统内核处理这个中断时，查看可用分区表或自由链表，找出一个足够大的连续区域装入该分段。如果找不到足够的一个分区，则检查空闲区的总和，若空闲区的总和能满足该分段的要求，那么就采用移动技术进行适当的内存移动，合并一个空闲区将该分段装入内存。若空闲区的总和不能满足该分段的要求，则可能调出一个或几个段到外存上，再将该分段装入。段的调出算法也可采用类似于页面的一些调度算法。

在程序执行中，若由于处理数据的需要而要扩大该数据段的空间时，分段式虚拟存储系统也能较容易实现这个分段的扩展。它允许用户在可扩充的段中使用超过该段长的地址空间。若程序访问的地址超过原有的段长，硬件就产生一个越界中断。操作系统内核处理这个中断时，先检查该段的扩充位信息。若该段是不可扩充的，则越界中断处理就只能报告地址错，同时停止用户程序执行。若该段是可扩充的，则为该段增加长度。增加该段长度时，先看是否有空闲区与该段后相邻及相邻的空闲区是否满足其扩展的要求，若是则该段就直接往后扩展并调整段表与空闲区表，否则应移动或调出一个或几个分段以足够让该段扩展，同时调整段表与空闲区表。

有关整个地址翻译流程包括缺段中断、越界中断、保护中断的处理流程。

2. 段的动态连接

分段式虚拟存储系统还可实现“动态连接装配”功能。所谓动态连接装配是指在程序运行

中对它所用到的子程序段或数据段进行连接装配，节省连接装配时间和程序所占的空间。

实现动态连接系统应增加两个功能：间接编址和连接障碍指示。间接编址是指指令中的地址单元的内容仍作为地址。例如，内存情况如图 5-25 所示，有一条指令 LOAD 1，100，在直接编址时，表示将 100 单元的内容(800)装入 1 号寄存器；而在间接编址时，表示将 800 单元的内容(1000)装入 1 号寄存器。

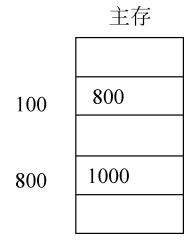
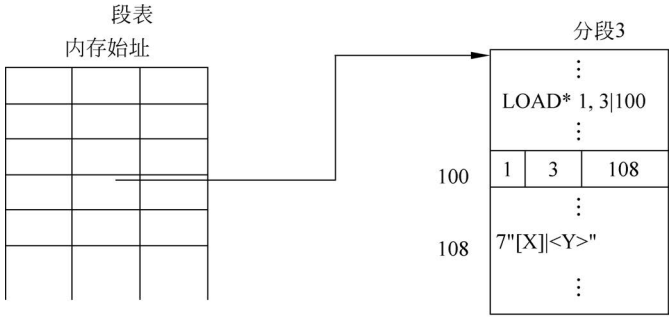


图 5-25　内存情况

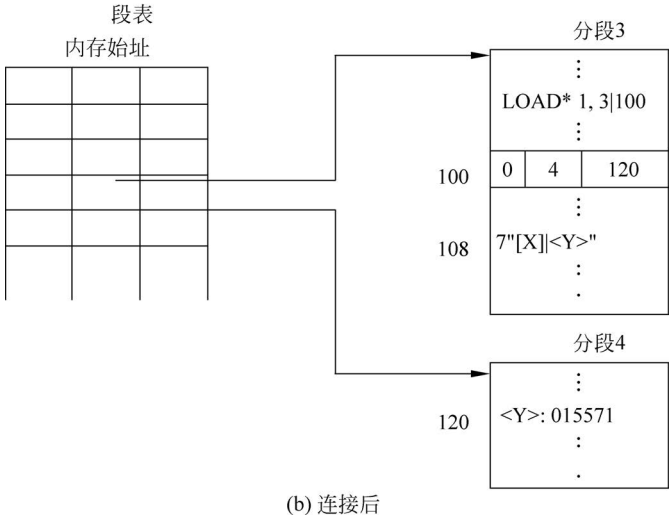
采用间接编址时，间接地址指示的单元内容称为间接字。间接字中应包含连接障碍指示位，如约定第 0 位为障碍指示位，以 L 表示。当 $L=1$ 时，表示需要动态连接，取该地址内容时，将产生连接中断，转 OS 内核处理，进行连接工作。当 $L=0$ 时，表示不要连接，其内容即为直接地址。间接字的格式为

L	直接地址

段的动态连接如图 5-26 所示，当分段 3 要对另一段产生符号形式的调用时，如 LOAD 1，[X]|〈Y〉，经编译或汇编程序处理就产生一个间接编址的指令(如 LOAD * 表示间接型指令)来代替，且将间接字中的障碍指示位置成 1，直接地址指向代表符号调用的字符串(7"[X]|〈Y〉")的所在位置。



(a) 编译或汇编后，连接前



(b) 连接后

图 5-26　段的动态连接

当程序执行到分段3的LOAD＊ 1,3|100指令时，就产生连接中断。OS内核按如下过程处理该连接中断。

(1) 从3段100单元取出间接字，获得直接地址3段108单元。

(2) 按直接地址取出要连接的符号名[X]|〈Y〉，按定义给它分配段号（如这里假定[X]＝4，〈Y〉＝120）。

(3) 查[X]段是否在内存，若不在则从外存上把它装入，修改空闲区表和登记段表，若已在则根据[X]段在内存的状况修改本程序的段表。

(4) 修改间接字，置障碍指示位为0，且使直接地址为连接的分段地址，即4段120单元。

(5) 恢复现场，重新启动被中断的指令执行。当LOAD＊1,3|100指令重新执行时，由于间接字已无障碍指示，于是就按直接地址（4段120单元）读出所需的数据015571，装入1号寄存器。

▶ 5.5.3 段页式虚拟存储管理

由于段式虚拟存储管理严格按程序的逻辑结构分配连续存储空间，方便程序和数据的共享与保护，同时也便于程序及数据段的扩充和动态连接。但是，该类系统存在的明显的问题是：一个段的长度不能大于实际的内存容量，而且为了解决碎片问题，提高内存的利用率，必须采用移动技术，移动内存信息需要较大的系统开销。为了克服这些缺点，可在分段的基础上再进行分页，兼用分段和分页的方法，构成段页式虚拟存储管理系统。

在该系统中，每个进程按逻辑分段，然后对每一段又分成若干页。这样，每一段不必占用连续的内存空间，而是按页存放在不一定连续的内存页框中，并且当内存页框不够时只将一段的部分页面放在内存，用到不在内存中的页面时再将之调入。段页式虚拟存储管理系统早先在大中型计算机（如IBM 370、Honeywell 6180）中得到应用，如今由于硬件的快速发展，在工作站或微型计算机上已被采用，如Intel Pentium也提供了段页式虚拟存储管理技术。段页式的逻辑地址必须由三部分组成：段号、页号和页内偏移。如硬件提供了如图5-27所示的地址结构。

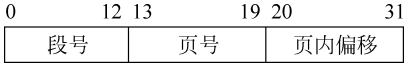


图5-27 硬件地址结构

则系统为每个进程提供了8192个分段，每段最多可以有128页，每页长度可达4KB。段页式存储管理系统必须为每个进程设立一张段表和若干张页表。段表中应指出每段的页表始址和长度及其他特征信息。页表长度可由段长和页面大小决定。段页式存储系统的段表与页表如图5-28所示。

在进行地址转换时，根据逻辑地址中的段号查段表得相应段的页表始址，然后根据页号查页表得到对应的内存页框号，由页框号和页内偏移就可形成欲访问的绝对地址。由此可看出，要存取一次信息，必须经历三次访问内存操作，一次访问段表，一次访问页表，最后才能按绝对地址存取信息，这样就降低了指令执行的速度。为了加快地址转换，也采用相连存储器来存放快表，快表中应指出段号、页号和内存页框号。段页式存储管理系统地址转换过程如图5-29所示。

由于段中分页，段页式虚拟存储管理系统不必像分段式虚拟存储系统那样每次都要把整段信息全部装入内存，而只需把当前要使用的某段中的若干页装入即可。因此，必须在段表和

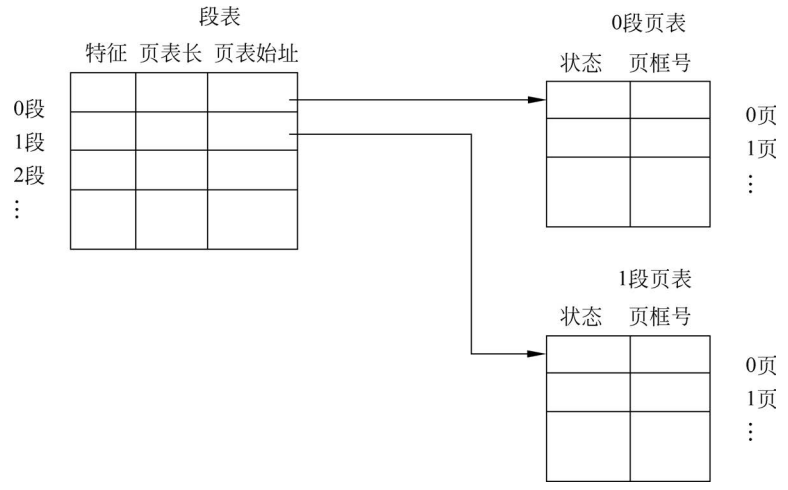


图 5-28　段页式存储系统的段表与页表

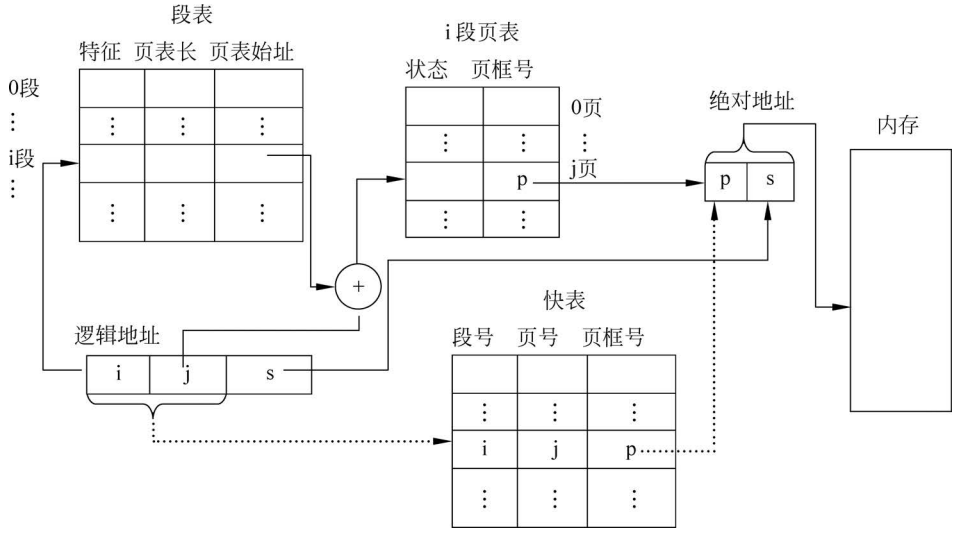


图 5-29　段页式存储管理系统地址转换过程

页表中分别指出对应的段或页是否已在内存。硬件在执行指令和进行地址转换时，可能会引起缺段中断、越界中断、保护中断、缺页中断和连接中断等情形。段页式虚拟存储管理地址转换处理流程如图 5-30 所示。

这些中断的处理思想如下。

（1）缺段中断：为该段建立一张页表，填写该段页表始址和长度及其他必要的信息。

（2）越界中断：当该段可扩充时，应增加页表表目，修改段表中的页表长度。当该段不可扩充时，报出错信息，停止用户程序执行。

（3）保护中断：报告存取违法并停止程序执行。

（4）缺页中断：找出一个内存的空闲页框或调出一页，装入所需的页面，修改相应表格（如页表、存储页框表等）。

（5）连接中断：为该段分配一个段号，若该段已连接过，则根据该段的状况填写这个段表表项，否则可按缺段中断一样进行处理。最后，根据段号、页号与页内偏移形成无障碍指示的一般间接地址。

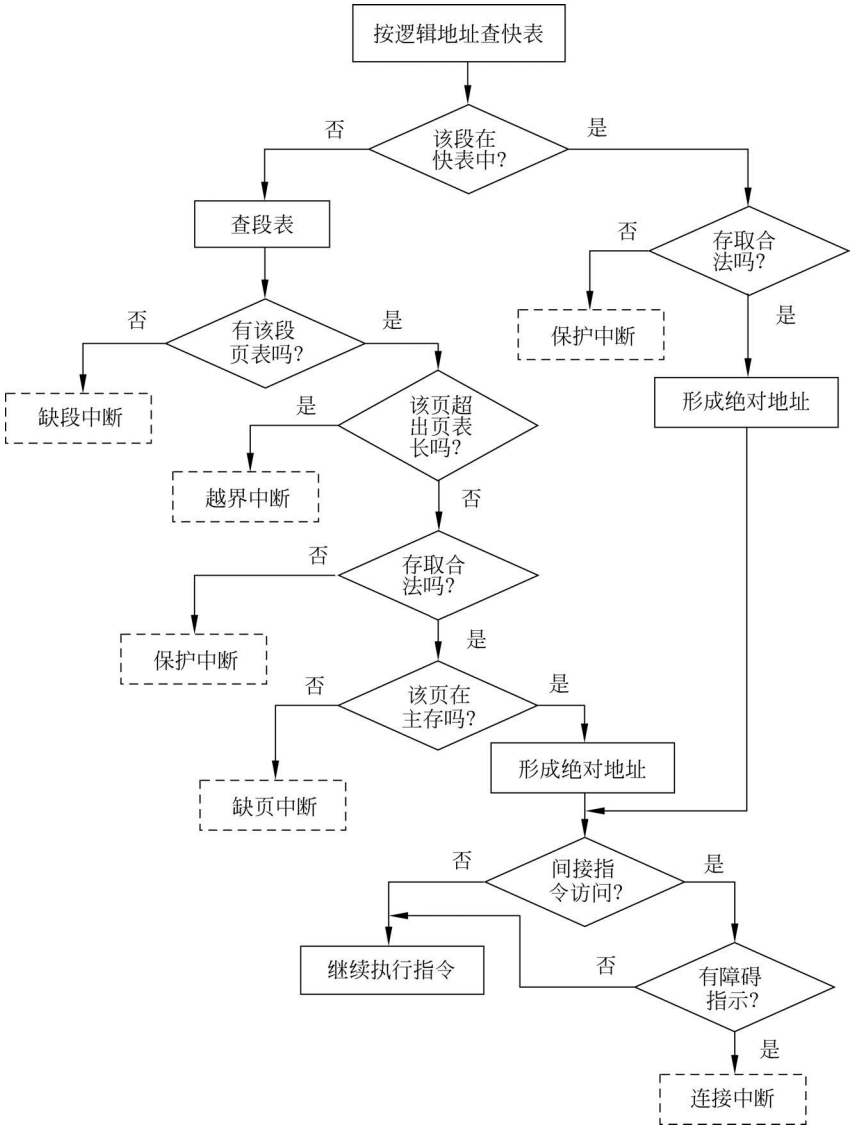


图 5-30　段页式虚拟存储管理地址转换处理流程

除了越界中断和保护中断外，其他这些中断在处理后，均应重新启动被中断的指令执行。段页式虚拟存储管理系统具有了段式和页式的全部优点，但是需要更多的硬件支持和中断处理，增加了系统的成本和复杂性。因此，是否采用段页式虚拟存储管理方式，应视具体的硬件性能和实际的应用对象而定。

5.6　内存管理实例

▶ 5.6.1　UNIX 内存管理

UNIX System Ⅴ采用请求页式和交换策略进行内存管理。交换技术与请求页式策略的主要区别在于：交换技术换进换出整个进程（proc 结构和共享正文段除外），因此，一个进程的大小受到物理存储器的限制；而请求页式策略在内存和外存之间来回传递的是存储页而不是

整个进程，从而使得进程的大小比可用的物理存储空间大得多。

1. 进程的虚拟空间描述

在 UNIX System Ⅴ中，一个进程由三个逻辑段组成：正文段、数据段和堆栈段。因此，一个进程的虚拟地址空间也被分成三个逻辑区来存放上述三个逻辑段。区是进程的虚拟地址空间上的一个连续区域，它是被共享、保护及进行内存分配和地址变换的独立实体。

为了管理每个进程中的各区，系统设有一个被称为区表的数据结构，每个在系统中存在的区都在该表中占有一个表项。区表包括下列内容。

(1) 区的类型：指明该区存放正文段、数据段或私有数据和堆栈段。

(2) 区的长度。

(3) 区所对应页表的内存地址。

(4) 区的状态：包括是否已被调入内存，是否正在调入内存过程中，是否被锁住，以及是否正在被请求调入内存等。

(5) 共享位：给出共享该区的进程数。

(6) 文件系统指针：指向外存中与该区对应的数据文件。

系统区表如表 5-7 所示。

表 5-7　系统区表

区　号	类　型	长　度	内存始址	状　态	共享位	文件指针

在系统创建新进程时，核心将从区表中分配相应的表项给所创建的进程。

为了把区表和进程联系起来，当进程中的某个逻辑段在区表中分得表项并填写了相关栏目之后，将把该表项的内存地址返回到进程的 proc 结构中。proc 结构中与区表项有关的还有该段在虚拟存储空间的起始地址、内存中的页表地址及页表长度等。

把区表和进程 proc 结构分开的原因之一是便于共享。因为每个逻辑区在不同的进程中对应的虚拟地址是不同的，但它们却可以通过区表而对应变换到同一物理内存空间中。区表和进程 proc 结构的关系如图 5-31 所示。

在系统创建一个进程或让一个进程共享其他进程的某个逻辑段时，在分配区表项后或在改变有关区表项共享计数位后，就把区表项与有关进程连接起来。对于新进程的连接是在创建 proc 结构时填写区表项地址、该区表项对应逻辑段的虚拟地址、所需页表的内存始址、逻辑段长度等。对于共享段的连接只要填写区表项地址、页表的内存始址、逻辑段长度即可。但是，共享段的虚拟地址在不同的进程中是不一样的。

请注意，UNIX System Ⅴ中的区和段页式存储管理中的段非常相像。但它们是有区别的，段页式存储管理中的虚拟地址空间是二维的，而 UNIX System Ⅴ中的各进程的分区虚拟地址空间仍是一维的。另外，UNIX System Ⅴ中的各进程的分区并不是由用户按照逻辑功能独立定义的，而是由系统设计人员预先设置好的。

2. 进程交换与请求页式页面置换

1) 进程交换

UNIX 系统为了缓解内存资源紧张的局面，将内存中处于睡眠状态的某些进程各区所在

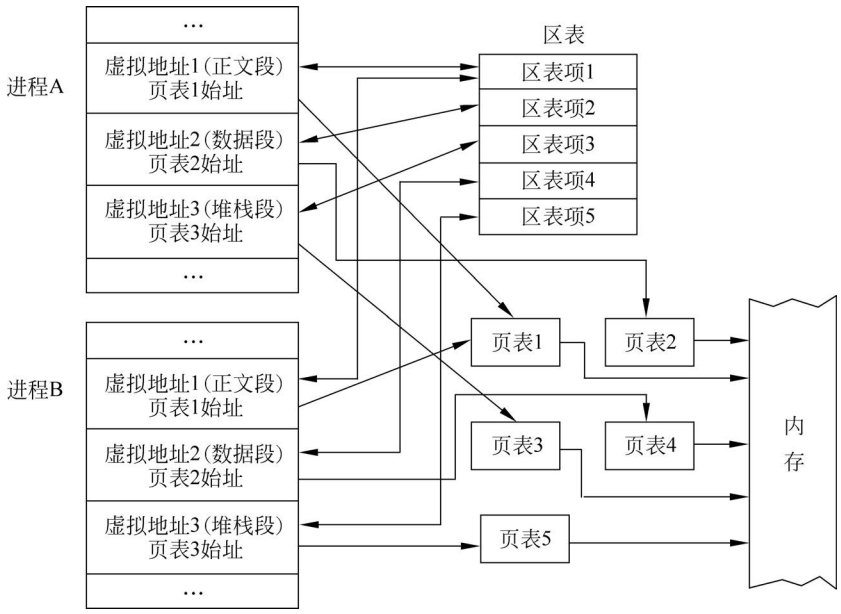


图 5-31　区表和进程 proc 结构的关系

的内存副本调到外存交换区中，而将交换区中处于就绪的进程重新调入内存。系统内核提供了交换空间管理、进程换出和进程换入三个功能，从而实现进程交换策略。

为了实现快速的内外存的交换，进程在外存交换区的存储分配是采用连续空间分配法。进程的换出功能是，当内核选定一个进程换出时，先将该进程的各区的引用计数减 1，然后选择其值为 0 区换出，同时对该进程加锁，直至该进程的内存副本复制到交换区中。最后，释放该进程所占的内存。进程换入功能是，每当睡眠进程被唤醒去执行换入操作时，内核就找出“就绪且换出”状态的进程，把其中换出时间最久的进程作为换入进程，并根据该进程的大小，为其申请内存，若内存申请成功，直接将该进程换入，否则还要考虑将内存中的某些进程换出，以腾出足够的空间再将该进程换入。

2）请求页式页面置换

UNIX 系统设置了一个核心进程称为换页进程，实现请求页式页面置换功能。换页进程的工作原理近似于“最近最少用”(LRU)页面置换算法。

每当内存空闲页框数低于某个规定的下限时，内核唤醒换页进程，换页进程检查内存中每一个活动的、非上锁的区，增加所有有效页的年龄，当进程访问某页时将该页的年龄置 0，换页进程选择年龄最大的页，并将其换出。

3. 进程的虚拟空间管理操作

UNIX System Ⅴ中进程可以使用系统调用 sbrk 来改变一个区的大小。当扩展区时，核心要保证扩展的区的虚地址不与另一个区的虚地址重叠，并且区的增大不应引起进程的大小超过所允许的最大虚存空间。注意，正文区和共享存储区在初始化后不能再被扩展。提供给用户进行动态存储分配的库函数 malloc()就是通过调用系统调用 sbrk 来实现的。

进程通过系统调用 shmget 创建一个共享存储区时，核心在系统区表中为该区建立一个区表项。此后，进程可以使用系统调用 shmat 将此共享存储区映射到本进程的虚空间中，称为附接(attach)，核心将为该进程建立相应的本进程区表项。如果该区是首次附接到一个进程，核心还要为附接区分配和初始化页表。

进程使用系统调用 fork 产生一个子进程时，核心要为子进程复制父进程的所有区。如果某个区是共享的，核心就不必物理地复制该区，而是增加该区的共享进程数，允许父、子进程共享该区。如果某个区不是共享的核心，就必须物理地复制该区，这需要分配一个新的区表项、页表以及物理存储空间。

进程可使用系统调用 exec 改变本进程的虚空间映像，核心将释放进程虚空间中现有的所有区，然后把指定的可执行文件装入该进程的虚空间中，建立相应的区，如正文区、数据区和堆栈区。这里所说的装入不是实际的装入，而只是建立相应的页表项，页表项中填的不是物理存储页地址，而是磁盘块号。可执行文件内容的实际调入要推迟到发生页面失效时，核心才分配一物理存储页，按照页表项中的设备号和磁盘块号，把访问到的指令或数据读入此页面中，并在相应的页表项中填入此页面的物理地址。

▶ 5.6.2　Linux 内存管理

Linux 将存储管理分为物理内存管理、内核内存管理、虚拟内存管理、内核虚拟内存和用户级内存管理。

1. 物理内存管理

物理内存管理以页为单位，记录、分配和回收物理内存，物理内存管理使用 Buddy(伙伴)算法。

1) 空闲物理内存单元的管理

Linux 物理内存管理使用 Buddy 算法实现。其算法思想是：把内存中所有页面按照 2^n 划分，其中，$n=0\sim10$，每个内存空间按 1 个页面、2 个页面、4 个页面、8 个页面、16 个页面、32 个页面、…、1024 个页面进行 11 次划分。划分后形成了大小不等的存储块，称为页面块，简称页块。包含 1 个页面的页块称为 1 页块，包含 2 个页面的称为 2 页块，以此类推。每种页块按前后顺序两两结合成一对 Buddy“伙伴”。系统按照 Buddy 关系把具有相同大小的空闲页面块组成页块组，即 1 页块组、2 页块组、…、1024 页块组。每个页块组用一个双向循环链表进行管理，共有 11 个链表，分别为 1、2、4、…、1024 页块链表，分别挂到 free_area[] 数组上。同时采用位图数组标记内存页面使用情况，每一组每一位表示比邻的两个页面块的使用情况。当一对 Buddy 的两个页面块中有一个是空闲的，而另一个全部或部分被占用时，该位置 1。当两个页面块都是空闲，或都被全部或部分占用时对应位置 0。Buddy 算法的内存管理如图 5-32 所示，其中，物理块 0、2、6、7、13 已被使用。

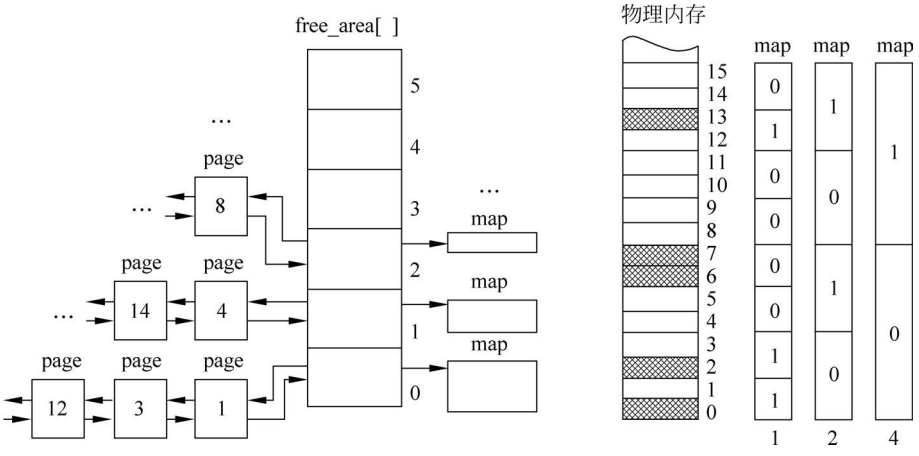


图 5-32　Buddy 算法的内存管理

2）物理页的分配

内存分配时，按照 Buddy 算法，根据请求的页面数在 free_area[]对应的空闲页块组中搜索。若请求页面数不是 2 的整数次幂，则按照稍大于请求数的 2 的整数次幂的值搜索相应的页面块组。当相应页块组中没有可使用的空闲页面块时，就查询更大一些的页块组，在找到可用的空闲页面块后，分配所需页面。当某一空闲页面块被分配后，若仍有剩余的空闲页面，则根据剩余页面的大小把它们加入相应页块组中。

3）物理页的释放

当内存页面释放时，系统将其作为空闲页面看待，并检查是否存在与这些页面相邻的其他空闲页块，若存在，则合为一个连续的空闲区，按 Buddy 算法重新分组。这样可避免存在大量的小页块组。

2．内核内存管理

内核内存管理主要负责为各种内核数据结构分配空间，其大小一般较小。如果使用以页为单位的物理内存管理则浪费较大，为此 Linux 专门提供了使用 Slab 算法的内核内存管理。

Slab 算法思想是：对象的申请和释放通过 Slab 分配器来管理。Slab 分配器有一组高速缓存(Cache)，每个高速缓存保存同一种类型的对象，如 i 节点、PCB 等。此外，还有一些通用对象，如 32B、64B、…、128KB 等。内核从它们各自的缓存中分配和释放对象。每种对象的高速缓存由一连串 Slab 构成，每个 Slab 由一个或者多个连续的物理页面组成。这些页面中包含已分配的缓存对象，也包含空闲对象。

3．虚拟内存管理

在物理内存管理的基础上，使用请求调页机制和交换机制，页面置换采用近似的“最近最少使用”(LRU)算法，为系统中的每个进程都提供高达 4GB(i386 平台)的虚拟内存空间。

1）页表的管理

一个页表条目标识一个物理页，如标识物理页的页框号、该页是否有效、该页的读写权限等。为了操作系统的可移植性，Linux 使用三级页表来存储虚拟地址转换为物理地址映射关系。一级页表只占用一个页，其中存放了二级页表的入口指针，记为 PGD；二级页表中存放了三级页表的入口的指针，记为 PMD；在三级页表中每个项都是一个页表条目。在 Linux 的 x86 版本中，只使用了两级页表，即第一级和第三级，在 Intel 系列 CPU 中，一个物理页面大小是 4KB，而每个页表条目大小是 4B，其中高 20 位存放页框号，低 10 位标识页面属性。因此，每个物理页面可以包含 1024 个页表项，则每个进程的虚拟地址空间为 1024×1024×4KB=4GB 大小。

2）虚拟存储空间的管理

在 Linux 系统中，主要使用了三个层次的数据结构 page、mm_struct 和 vm_area_struct 来表示进程的虚拟地址空间。最底层的 page 结构描述了一个物理页框及其页内信息的相关属性和链接指针，包括标志位、引用计数等。

vm_area_struct 结构是中间层次，它描述了一个虚拟内存区域(即一段连续的虚拟地址空间)的属性。其中，包括虚拟内存区域的开始地址、结束地址、访问权限、页目录、映射文件和链接指针。

mm_struct 是描述进程虚拟地址空间的最高层的数据结构，一个 mm_struct 就代表一个独立进程的虚拟内存空间。该结构中记录了实现任务管理的进程模型所需要的内存管理相关的全部信息，如进程的页目录的位置，进程的代码、数据、堆栈、堆、环境变量、入口参数等在虚

拟地址空间中的存储位置，进程占用的物理页框数目、进程的 LDT（局部描述符表）、引用计数，进程的虚拟地址空间的虚拟内存区域链表的链接信息和一些统计信息。

3）虚拟地址空间的创建

进程虚拟地址空间的创建分为两个步骤。首先，复制父进程的地址空间；然后，根据可执行映像的要求，创建新的内存地址空间。

虚拟地址空间的复制：为了减少开销，Linux 对于 fork 调用，使用了 COW 技术，同时也提供 vfork 系统调用，使用 vfork 系统调用创建的进程完全和父进程共享同一个地址空间。在 Linux 中，也使用 vfork 系统调用实现线程管理。

虚拟地址空间的重建：fork 系统调用返回后，子进程已经通过虚拟内存复制，创建了自己的地址空间。此后，当进程调用 exec 系统调用，希望执行新的程序时，将根据新的执行映像，为该进程创建新的虚拟地址空间。通常将这个过程叫作虚拟地址空间的重建。

在 Linux 中采用内存映射机制来处理映像文件的装入。在 vm_area_struct 结构中，由一个 file 结构的 vm_file 域和一个无符号长整型类型的 vm_pgoff 域分别表示该内存区域映射文件的文件指针和偏移值。实际上，内存映射机制正是使用这两个域描述了某段内存空间对应的内容在文件中的位置。这样，在重建虚拟地址空间时，只需要建立一系列的数据结构，描述某段内存区域内容在可执行映像中的位置，只有当进程真正使用该区域时，才将其装入内存。使用这种机制就避免了将映像中并不使用的部分也装入了内存中。

4. 内核虚拟内存与用户级内存管理

Linux 将每个进程的 4GB 虚拟内存分为用户区（0～3GB）和内核区（3～4GB）。内核虚拟内存管理负责内核区虚拟内存的管理。

内核态的程序有时需要申请大片的内存，如用于交换、内核模块、I/O 缓冲等，则采用申请虚地址空间连续、物理页面不连续的方法。可分配的虚拟空间在 3GB＋high_memory＋HOLE_8M 以上的高端区域，由 vmlist 链表进行管理。申请与释放这些空间时，分别需要调用系统调用 vmalloc()和 vfree()

用户任务空间的管理，即用户级进程的内存空间管理，一般提供 C 库函数，用户应用库函数编程实现用户级内存的管理。

▶ 5.6.3　Windows 内存管理

Windows 采用分页式虚拟存储管理技术，其页表的组织采用多级页表结构实现。Windows 虚拟内存管理器控制内存的分配和分页的执行。内存管理器可以在多种平台上运行，并使用 4～64KB 的页面大小。Intel 和 AMD64 平台每个页面为 4KB，Intel Itanium 平台每个页面为 8KB。

1. Windows 虚拟地址映射

在 32 位平台上，每个 Windows 用户进程都有一个单独的 32 位地址空间，每个进程可以使用 4GB 的虚拟内存。默认情况下，一半的内存是为操作系统保留的，因此每名用户实际上有 2GB 可用的虚拟地址空间，当在内核模式下运行时，所有进程共享大部分的 2GB 系统空间。在客户端和服务器上，大型内存密集型应用程序可以使用 64 位 Windows 更有效地运行。除了上网笔记本电脑，大多数现代 PC 使用 AMD64 处理器架构，它能够支持 32 位或 64 位系统运行。

2. Windows 分页

当创建一个进程时，原则上它可以使用近 2GB（或 64 位 Windows 上的 8TB）的整个用户

空间。该空间被划分为固定大小的页面，其中任何页面都可以被带入内存，但操作系统会在以64KB为边界分配的连续区域中管理地址。一个区域可以处于以下三种状态之一。

（1）可用(available)：当前进程未使用的地址。

（2）保留(reserved)：虚拟内存管理器为进程预留的地址，这些地址不能被分配给其他用途(例如，为堆栈增长保留连续空间)。

（3）已提交(committed)：虚拟内存管理器已初始化的地址，供进程访问虚拟内存页使用。这些页面可以位于磁盘或物理内存中。在磁盘上时，它们可以保存在文件(映射页)中，也可以占用分页文件(即从主内存中删除页面时将其写入的磁盘文件)中的空间。

区分保留和已提交内存是非常有用的，因为一是减少了系统所需的虚拟内存空间总量，从而使页面文件更小；二是允许程序保留地址，而无须让程序访问这些地址，也无须将这些地址计入其资源配额。

Windows所采用的驻留集管理方案是动态分配、局部范围。当一个进程首次启动时，会被分配数据结构来管理其工作集。随着进程所需的页面被调入物理内存，内存管理器使用这些数据结构来跟踪分配给该进程的页面。活动进程的工作集调整遵循以下一般约定。

（1）当主存充足时，虚拟内存管理器允许活动进程的驻留集增长。为了实现这一点，在发生页面错误时，会向进程添加一个新的物理页面，但不会将任何旧页面换出，从而使该进程的驻留集增加一个页面。

（2）当内存变得稀缺时，虚拟内存管理器通过从活动进程的工作集中移除最近较少使用的页面来为系统回收内存，从而减小这些驻留集的大小。

（3）即使在内存充足的情况下，Windows也会监视那些内存使用快速增加的大型进程。系统开始从进程中移除最近未被使用的页面。这个策略使系统更具响应性，因为一个新程序不会突然导致内存不足，而使用户在系统尝试减小已经运行的进程的驻留集时等待。

3. Windows交换

随着Metro UI的推出，Windows引入了一种新的虚拟内存系统来处理来自Windows Store应用程序的中断请求。swapfile.sys与对应的pagefile.sys一起提供了访问硬盘上临时存储区的方式。paging会保存长时间未访问的项目，而swapping则保存最近从内存中取出的项目。在pagingfile中的项目可能在很长一段时间内不再被访问，而在swapfile中的项目可能会更快地被访问。只有商店应用程序使用swapfile.sys文件，并且由于商店应用程序相对较小，因此固定大小仅为256MB。pagefile.sys文件的大小大致为系统中物理RAM量的1～2倍。swapfile.sys通过将整个进程从系统内存交换到swapfile中来运作。这立即释放内存以供其他应用程序使用。相比之下，paging文件的功能是将程序“页面”从系统内存移动到paging文件中。这些页面的大小为4KB。而整个程序无须交换到paging文件中。

▶ 5.6.4 OpenHarmony内存管理

本实例主要结合鸿蒙内核LiteOS-A的源码讲解鸿蒙内核是如何进行内存管理的。

鸿蒙内核采用虚拟段页式内存管理。打开kernel/liteos_a/kernel/base/include/los_vm_phys.h源码，可以看到两个主要结构体。结构体每个成员变量的含义都已注释，下面结合源码进行讲解。

```
#define VM_LIST_ORDER_MAX    9        //伙伴算法分组数量,从 2^0,2^1,…,2^8 (256 * 4K) = 1M
#define VM_PHYS_SEG_MAX    32         //最大支持 32 个段

typedef struct VmPhysSeg {//物理段描述符
    PADDR_T start;                    /* The start of physical memory area */
//物理内存段的开始地址
    size_t size;                      /* The size of physical memory area */
//物理内存段的大小
    LosVmPage *pageBase;              /* The first page address of this area */
//本段首个物理页框地址
    SPIN_LOCK_S freeListLock; /* The buddy list spinlock */
//伙伴算法自旋锁,用于操作 freeList 上锁
    struct VmFreeList freeList[VM_LIST_ORDER_MAX]; /* The free pages in the buddy list */
//伙伴算法的分组,默认分成 10 组 2^0,2^1,…,2^VM_LIST_ORDER_MAX
    SPIN_LOCK_S lruLock;              //用于置换的自旋锁,用于操作 lruList
    size_t lruSize[VM_NR_LRU_LISTS];  //5 个双循环链表大小,如此方便得到 size
    LOS_DL_LIST lruList[VM_NR_LRU_LISTS]; //页面置换算法,5 个双循环链表头,它们分别描述 5 种
//不同类型的链表
} LosVmPhysSeg;


//注意: vmPage 中并没有虚拟地址,只有物理地址
typedef struct VmPage { //物理页框描述符
    LOS_DL_LIST     node;             /**< vm object dl list */
//虚拟内存节点,通过它挂/摘到全局 g_vmPhysSeg[segID]->freeList[order]物理页框链表上
    UINT32         index;             /**< vm page index to vm object */ //索引位置
    PADDR_T          physAddr;        /**< vm page physical addr */
//物理页框起始物理地址,只能用于计算,不会用于操作(读/写数据 == )
    Atomic         refCounts;         /**< vm page ref count */
//被引用次数,共享内存会被多次引用
    UINT32         flags;             /**< vm page flags */
//页标签,同时可以有多个标签(共享/引用/活动/被锁)
    UINT8          order;             /**< vm page in which order list */
//被安置在伙伴算法的几号序列(  2^0,2^1,2^2,…,2^order)
    UINT8          segID;             /**< the segment id of vm page */
//所属段 ID
    UINT16         nPages;            /**< the vm page is used for kernel heap */
//分配页数,标识从本页开始连续的几页将一块被分配
} LosVmPage; //注意:关于 nPages 和 order 的关系说明,当请求分配为 5 页时,order 是等于 3 的,因为
//只有 2^3 才能满足 5 页的请求
```

内核默认最大允许管理 32 个段。段页式管理就是先将逻辑地址切成多段,每段再切成单位为 4KB 的页,页是在内核层的操作单元,物理内存的分配、置换、缺页、内存共享、文件高速缓存的读写,都是以页为单位的,所以 LosVmPage 极其重要。

结构体的每个变量代表一个功能点,结构体中频繁出现 LOS_DL_LIST(双向链表),双向链表是鸿蒙内核最重要的数据结构。

LosVmPage.refCounts 页被引用的次数,可以理解为被进程拥有的次数。当 refCounts 大于 1 时,说明该页被多个进程所拥有,是共享页;当等于 0 时,说明该页没有进程在使用,可以被释放,这类似于 Java 的内存回收机制。在内核层面,引用的概念不仅适用于内存模块,也适用于其他模块,如文件/设备模块,同样存在共享的场景。

段的划分需要进行手动配置,相关参数存在静态全局变量中,鸿蒙默认只配置了一段,在源码 kernel/liteos_a/kernel/base/vm/los_vm_phys.c 中的配置如下。

```
struct VmPhysSeg g_vmPhysSeg[VM_PHYS_SEG_MAX];   //物理段数组,最大 32 段
INT32 g_vmPhysSegNum = 0;                         //总段数
LosVmPage *g_vmPageArray = NULL;                  //物理页框数组
```

```
size_t g_vmPageArraySize;                    //总物理页框数
/* Physical memory area array */
STATIC struct VmPhysArea g_physArea[] = {//这里只有一个区域,即只生成一个段
    {
        .start = SYS_MEM_BASE,               //整个物理内存基地址,define SYS_MEM_BASE
#DDR_MEM_ADDR , 0x80000000
        .size = SYS_MEM_SIZE_DEFAULT,//整个物理内存总大小 0x07f00000
    },
};
```

设置好段和这些全局变量，就可以对内存初始化了。下面是内存初始化代码，OsVmPageStartup 是物理内存初始化函数，它被系统内存初始化模块 OsSysMemInit 调用。

```
/*****************************************************************************
*******
完成对物理内存整体初始化,本函数一定运行在实模式下
  申请大块内存 g_vmPageArray 存放 LosVmPage,按 4KB 一页划分物理内存存放在数组中
 *****************************************************************************
******/
VOID OsVmPageStartup(VOID)
{
    struct VmPhysSeg *seg = NULL;
    LosVmPage *page = NULL;
    paddr_t pa;
    UINT32 nPage;
    INT32 segID;

    OsVmPhysAreaSizeAdjust(ROUNDUP((g_vmBootMemBase - KERNEL_ASPACE_BASE), PAGE_SIZE));
//校正 g_physArea size

    nPage = OsVmPhysPageNumGet(); //得到 g_physArea 总页数
    g_vmPageArraySize = nPage * sizeof(LosVmPage); //页表总大小
    g_vmPageArray = (LosVmPage *)OsVmBootMemAlloc(g_vmPageArraySize);
//实模式下申请内存,此时还没有初始化 MMU

    OsVmPhysAreaSizeAdjust(ROUNDUP(g_vmPageArraySize, PAGE_SIZE));

    OsVmPhysSegAdd(); //完成对段的初始化
    OsVmPhysInit(); //加入空闲链表和设置置换算法,LRU(最近最久未使用)算法

    for (segID = 0; segID < g_vmPhysSegNum; segID++) {//遍历物理段,将段切成一页一页的
        seg = &g_vmPhysSeg[segID];
        nPage = seg->size >> PAGE_SHIFT;      //本段总页数
        for (page = seg->pageBase, pa = seg->start; page <= seg->pageBase + nPage;
//遍历,算出每个页框的物理地址
             page++, pa += PAGE_SIZE) {
            OsVmPageInit(page, pa, segID); //对物理页框进行初始化,注意每页的物理地址都不一样
        }
        OsVmPageOrderListInit(seg->pageBase, nPage); //伙伴算法初始化,将所有页加入空闲链
//表供分配
    }
}
```

5.7 内存管理设计与实现问题

▶ 5.7.1 内存管理设计问题

作为一个操作系统设计者，必须很好地考虑内存管理所涉及的各种因素，使得所设计的存

储管理有更好的总体性能。下面以分页式存储管理为例，分析讨论内存管理设计应考虑的主要问题。

1. 页面调度问题

对于以分页管理的系统，页面调度问题直接影响着系统运行效率。页面调度性能与页面分配策略、负载控制、页面大小、空闲页框数等因素相关。

(1) 页面分配策略。当存储空间已用完而一个进程又出现缺页时，页面置换算法是考虑在该进程已在内存中的某一页选择出来淘汰置换呢，还是考虑在当前所有进程已在内存中的某一页选择出来淘汰置换呢？有局部分配策略和全局分配策略两种策略来考虑这个问题。局部分配策略是进程的页面分配只考虑在给定自己的空间内分配和置换，而全局分配策略则是页面分配考虑在所有可运行的进程的总空间中分配与置换。基于局部分配策略置换算法参考的页面数量少，可快速做出置换决策，但当工作集增大时，就可能出现“抖动”现象。当工作集在收缩时，会因为给定了过多的内存页框，又会浪费内存。基于全局分配策略的置换算法置换的页面更合理，特别是当工作集大小随着进程生命周期变化时，由于能动态调整各进程的分配内存的量，它能有效抑制“抖动”问题并充分利用内存。

(2) 负载控制。由于内存空间有限，对于多个进程执行的系统，无论是局部分配策略还是全局分配策略，均还可能出现“抖动”问题，即会出现每个进程分配的内存量都无法达到其满意的页面失效频率以内。一个有效的解决方法是：必须考虑采用进程换出技术将某些进程从内存中驱除出去，把它存到磁盘中，以腾出内存空间让其他正在运行的进程分享这些内存空间，直到“抖动”现象消除。需要注意的是，进程换出必有进程换进调度，当进程从磁盘中被换进执行时，其他的进程还可能需换出。另外，由于进程的换出使得同时在内存运行的进程数减少了，这样会影响多道程序运行的并发度，可能有时 CPU 或 I/O 出现空闲，不利于系统效率的有效发挥，故不仅要考虑同时运行的进程数以控制系统“抖动”，还需考虑留在内存中的进程是否有利于系统效率的有效发挥。

(3) 页面大小。页面的大小也是一个值得关注的问题，这在 5.5.3 节中已讨论。这是一个内存开销和页面调度开销权衡考虑的问题。当今随着内存越来越大，页面大小也倾向于越来越大。

(4) 空闲页框数。如果系统中有足够数量的空闲页框，页面调度性能就很好。否则系统无空闲页框且每个页框均被进一步修改，缺页时首先必须选择一页写到磁盘交换区中。许多系统设置了一个调页监听进程(paging daemon)，该进程定期地被唤醒监测内存状况，它将采用相应的页面置换算法选择一些页淘汰，将这些页放进空闲页框池，如果这些页近期加载后已被修改，就将它写回磁盘交换区中。当被淘汰的页又要用时，若空闲页框池有该内容，可以将它从空闲页框池中拉回使用。保持一个空闲页框池比使用所有页框去找一个需要用的页框性能更好，至少调页监听进程保证了空闲页框是干净的，当缺页需要使用它时无须执行写磁盘动作。在 Clock 页面置换算法中，可以使用前后两个指针实现空闲页框池的高效管理，前一个指针给调页监听进程使用，而后一个指针给置换算法使用。

2. 指令与数据分离

分页内存管理传统的思路是每个进程的指令和数据使用单一的逻辑地址空间。若给定的内存空间足够大，这种地址空间的组织工作情况也很好。但多任务系统中通常给定的内存空间是较小的，使得程序员满脑袋都在考虑如何将所有的东西都放入这个地址空间，同时程序和数据难以有效考虑共享。

PDP-11(UNIX)系统率先给出了程序(指令)和数据分离组织逻辑地址空间的方案,该方案指令部分和数据部分逻辑地址空间分别都从0开始编址,它们都有自己的页表。存取指令执行时使用指令页表,存取数据时使用数据页表,独立实现页面调度。其好处是程序和数据的共享易于实现,提高了有效地址空间的利用率。

3. 信息共享

信息共享可以减少存储开销,提高进程执行效率,但为了保证信息的一致性,通常对可以共享的页或段必须是从访问权限上加以规定,只允许只读形式。对于信息的共享需要从页的共享、库的共享、文件共享和共享存储的接口设计等几个问题来考虑。

对进程采用指令部分和数据部分逻辑地址空间分离组织,信息共享更容易实现,而使用单一的逻辑地址空间组织,信息共享较为复杂。当多个进程共享代码时,若一个进程被调度换出,则它的所有页将从内存中被驱除出去,而共享该代码的进程运行时将产生大量的页面失效,又要从磁盘中调入。同样地,当一个进程终止时,还必须能够发现哪些信息还在使用,以保证它们在磁盘中的空间不被释放。搜索页表看哪些信息被共享太费时,因此必须需要一个特殊的数据结构记录那些被共享的页。共享数据部分如允许修改则考虑得要更复杂一些,如UNIX的fork()的系统调用,创建的子进程和父进程是共享代码和数据的,只要它们修改了某页数据,就违反了只读保护从而陷入操作系统核心,核心将为之建立它的私有数据页备份,每一个私有数据页备份被设成可读写模式,从而实现数据的更新。该策略称为写时备份(copy on write),可由减少备份提高性能。

许多系统提供了各种各样的程序库给程序开发使用,使得可方便集成已有特殊处理算法。若采用静态连接的方法进行可执行程序的连接装配,可执行程序代码量大,将耗费连接装配时间与空间。通常的方法是系统使用共享库,如Windows提供了动态链接库(DDLs),当一个程序使用共享库连接时,并不包含被调用的实际函数,而是连接程序加入一个小的桩(stub)例程,使得执行时去绑定被调用的函数,若共享库已经被相关程序使用时加载过,则执行时无须再次加载,加载也是按需一页一页地调入内存。多个进程可同时共享这个在内存中的共享库函数。共享库除了使得可执行代码量小与节省内存空间的好处以外,另一个好处是:共享库若有bug而被更新,调用它的程序无须重新编译,只须下载新的共享库更新老的共享库。由于进程不同共享库被定位的虚拟地址是不同的,使用共享库有一个问题需要解决,即共享库的转移指令地址必须采用相对地址,而不是绝对地址,转移指令地址应为向前或向后 n 字节地址。这就要求编译器对共享库编译能生成这种相对地址。

共享库是内存映像文件的一个特例,它更适合程序共享。内存映像文件即一个进程可以发出系统调用,将一个文件映射到它的一部分内存虚拟空间中。多数系统在实现中,映射文件时并不需要调页,当页被触及时,使用文件作为后备存储以请调方式进行调度。当进程终止时,被修改的页将被写到这个文件中。映像文件可以以内存中的数组特征进行访问,而无须考虑读写操作。在某些情形下可方便程序员使用该文件。如果多个进程同时去映射相同的文件,它们可通过共享存储进行通信。它们之间的信息交换和共享将更加及时。

此外,存储管理可以考虑为程序员提供共享存储的接口,让程序员可以以非传统的方法编写处理程序,到达高效的执行目的。

▶ 5.7.2 内存管理实现问题

1. 操作系统参与调页

操作系统在进程创建时、进程执行时、页面失效时和进程终止时需要参与调页工作。当进

程被创建时，操作系统必须考虑程序和数据有多大并为之在内存中分配空间、创建页表，同时初始化页表内容。此外，磁盘交换区中需分配一定空间以便接纳被换出的页。交换区还必须用程序正文和数据进行初始化，以便一个新的进程缺页时能从交换区中调入。页表和磁盘交换区的信息必须记录在进程表中。

当进程被调度执行时，存储管理单元需要被重新设置，快表（TLB）也要刷新，以保证驱除原先执行进程的痕迹。新进程页表必须是最近的，通常将它复制到相应的硬件寄存器中。一个可选的方案是初始时将进程的一些页或全部都调入内存，以减少缺页次数。

当页面失效时，操作系统必须读出硬件寄存器的内容判断出失效的虚拟地址，计算出所需的页并找出该页在磁盘中的位置，然后找到一个可用的页框或调用页面置换算法淘汰相关页，调入该页。最后让程序计数器指向失效的指令，让该指令重新执行。

当进程终止时，操作系统必须释放进程页表、所占页框和磁盘交换区的空间，若一些页被其他进程共享，则这些页框和其在磁盘交换区的空间将等到最后一个共享它们的进程终止时释放。

2．页面失效处理

当进程发生页面失效时，可能有以下这些事件出现并需要处理。

（1）硬件陷入内核，保存程序计数器内容到栈中。

（2）启动装配代码例程保存通用寄存器和其他已被修改的信息，以使得操作系统不破坏它们。

（3）操作系统发现一个失效的页并找出一个需要的虚拟页，通常寄存器包含该页面地址。

（4）操作系统根据该地址需要进行地址保护和存取权限检查，若访问不合法，就发一个信号给该进程或终止该进程，否则操作系统看是否有空闲的内存，如果没有，就运行页面置换算法选择一页淘汰。

（5）如果被选择出的页是脏的（最近被修改过），该页要写入磁盘交换区，并被标注忙状态以防止其他进程访问。这时上下文转换将发生，挂起页失效的进程，调度一个可执行的进程占有 CPU 执行直到磁盘写操作完成。

（6）当被淘汰的一个页面是干净的（或它已写回到磁盘），操作系统找出失效页的磁盘地址，调度磁盘操作将该页读进被淘汰页的所占内存中。当页面信息读入时，同样挂起页失效的进程，调度一个可执行的进程占有 CPU 执行直到磁盘读操作完成。

（7）当页面信息读入完成，磁盘中断到来，修改进程页失效的页表项中内存地址为被淘汰页的内存地址，并标注该页正常（在内存）。

（8）失效的指令被返回到它原先执行时的状态，程序计数器重新设置为该指令的地址。

（9）调度失效进程，操作系统返回装配代码例程去装配它。

（10）装配例程重新装入寄存器和其他状态信息，并返回到用户空间继续执行。

3．指令备份

当指令陷入内核进行页面失效处理后，如何准确重启失效指令？这个问题还是较复杂的。因为有些 CPU 有多个操作数地址，指令的长度是不同的，页面失效地址也有多个。操作系统难以知道重启该指令的地址。

有些 CPU 设计者提供了一个隐藏的寄存器，让程序计数器在一条指令执行前将指令的地址复制到该寄存器中，这些机器可能还有第二个寄存器告诉哪个寄存器以多少量自动增加或减少了。有了这些信息操作系统就容易返回失效指令地址重新执行了。若没有这些信息，操

作系统就必须找出什么发生了失效并知道怎样去修复它。

4. 内存中的页加锁

当页面置换算法选择其信息还在 I/O 缓冲区的一页淘汰时，如果 I/O 设备正在交换该页，将它移出内存就会引起部分数据被写进缓冲区而部分数据被写到正在装载的页。解决这个问题的一个方案是必须对内存中正在进行 I/O 的页进行加锁，以保证它们不被移出内存，也可以是在内核缓冲区中执行 I/O，而后将数据复制到用户空间。

5. 页面备份存储管理

当一个页被淘汰，若信息被修改过，则需要写到磁盘交换区中，以便后面执行过程中可将它重新载入。那么这些页写到磁盘的什么地方呢？

最简单的方案是：用一个特殊的分区来分配交换区，该分区不存放正常的文件，全部用于存放正在运行的进程映像，每个进程映像顺序分配。这样，只要知道进程在交换区中的起始地址，其页在交换区的位置通过相对位移就可以计算出，页表中无须指出该页在交换区中的地址，只需在进程表中指出其在交换区的起始地址即可。这个方案也有一个问题，即进程运行中会由于数据区和堆栈的动态变化而增加进程的虚拟空间长度。可以考虑将进程按正文（指令）、数据和堆栈分别进行分区域事先预约多个磁盘连续块。多数 UNIX 系统采用该方案。该方案的好处是节省页表空间，缺点是磁盘交换区空间需要足够大。

另一种方案是：事先不分配交换区的映像，而是当淘汰的页需要写到磁盘时分配磁盘空间，当它被换进时释放磁盘空间。这种方案不要考虑内存中的进程与交换区的对应，节省交换区空间，但进程的页表必须开辟一项记录在磁盘交换区中的地址，同时还要用一位标志该地址是否有效。

Windows 使用文件系统中一个或多个更大的预先分配的文件来接纳交换的信息，进程可执行代码的调入直接从可执行文件中调入，指令页淘汰无须写盘，只有可变数据页的换出需要写盘，但需要考虑减少所需磁盘空间量的优化。

6. 策略与机制分离

策略与机制分离可以降低系统管理的复杂性。一个内存管理系统可被分为三部分来实现：①一个内核中的内存管理单元（MMU）；②一个内核中的页失效处理器；③一名用户空间中运行的外部页管理器。

当进程被建立时，外部页管理器就被告知去建立进程映像并分配磁盘备份存储，进程运行时，它可能会映射新对象到地址空间中，又要告知外部页管理器。当进程运行出现缺页时，由页失效处理器找出需要使用的虚拟页并给外部页管理器发信号，外部页管理器从磁盘中读取所需的页并将它复制到自己的虚拟地址空间中，然后告知页失效处理器页已装入。页失效处理器从外部页管理器的虚拟地址空间中解除与该页的映射，并请求 MMU 把该页放到用户地址空间的恰当位置。这样用户进程就可重新启动执行。

该实现方案开放了页面置换算法的调用，由外部页管理器选择使用它。但主要问题是：外部页管理器不能访问所有页的新老和修改状态位，需要一些机制将这些信息传递给外部页管理器或者由页面置换算法必须到内核中去获取。该方案的好处是：可有更多的页面置换算法供选择，具有更大灵活性。其不足之处是：需多次地跨用户-内核边界和系统部件间发送各种消息的额外开销。然而，现代计算机越来越快，软件越来越复杂，可信的软件越来越受关注，大多数设计者认为牺牲一些性能来换取内存管理的灵活性和可信度也是可以接受的。

小结

本章首先介绍了内存管理的任务和功能。接着介绍内存分配的几种形式和重定位概念，内存分配常有三种形式，即直接内存分配、静态内存分配和动态内存分配。动态内存分配是现代操作系统常用的方式。静态重定位是在程序执行前由装配程序一次性完成，而动态重定位是程序执行中由硬件地址转换机构完成。覆盖与交换技术是从逻辑上扩充内存的两种方法，覆盖技术要求程序员提供一个覆盖结构，它用于同一进程之间；而交换技术对程序员无任何要求，它可用于进程之间。

接着介绍了分区内存管理，其中，固定分区法管理方式简单，但存储利用率低，而动态分区法可提高存储利用率，但分配与回收算法复杂且需考虑移动、合并等问题，增加了系统的开销。

然后，重点讲述目前流行的内存管理方式——页式和段式存储管理，这两种内存管理均较容易实现虚拟存储技术，解决共享与保护等问题。页式存储管理将内存划分成大小相等的页框(块)，进程地址空间相应地分成页，页到页框的分配可非连续分配，可解决内存碎片问题，采用快表后使地址转换的效率能被人接受。页式虚拟存储系统使用户程序不受内存容量的限制，但要使系统获得较好的性能必须很好地控制缺页中断率，因此页面置换算法的选择是很重要的。常用的页面置换算法有 FIFO、LRU、NRU、LFU、Clock 等。OPT 算法是无法实现的，但可作为衡量其他算法优劣的标准。段式存储管理是按程序的逻辑模块来考虑内存分配的，它类似于动态分区法，地址转换过程又类似于页式存储管理，它更加方便程序和数据的共享与保护，同时可实现动态连接，但解决碎片问题系统开销大。为了能结合页式和段式的优点，提出了段页式存储管理方法，即程序按逻辑结构分段，段内按页框大小分页，最终实现非连续存储分配。

最后，简要介绍了流行的操作系统 UNIX、Linux、Windows、OpenHarmony 等内存管理技术，同时讨论了内存管理设计与实现需考虑的问题。

习题

1. 内存管理的主要功能是什么？

2. 内存分配有哪几种形式？

3. 什么是重定位？重定位有哪几种方法？

4. 简述什么是覆盖技术和交换技术。它们之间有什么区别？

5. 在动态分区管理中，当有 1KB、9KB、33KB 和 121KB 4 个进程要求进入系统时，试分析内存空间的分配情况(内存初始状态如图 5-33 所示)。

6. 为什么要进行存储保护？分区管理中通常有哪几种保护方法？

7. 给出一种动态分区的分配算法，写出内存分配和回收(去配)的流程图。

8. 何谓内存移动？采用移动法分配内存有什么优缺点？移动一道程序时操作系统应做哪些工作？

9. 一道程序被移动或调出时，有限制条件吗？为什么？

10. 对页式存储器进行内存分配时，应设置相应的存储分配表，请设计一个满足这种内存分配的数据结构，并给出分配和回收(去配)的算法。

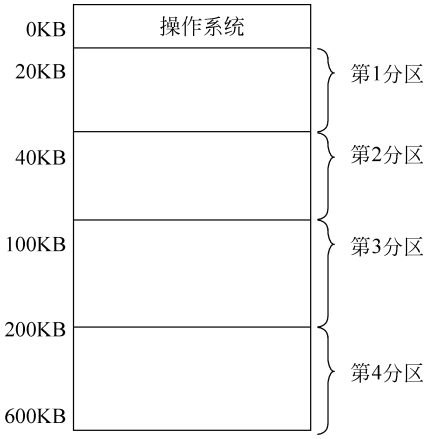


图 5-33　内存初始状态

11. 如果存放页表的区域被分成大小相等的块(页框),每个进程的页表可存放在一块或几块中,当某个进程的页表要占用多块时,应怎样构造页表?

12. 何谓页式存储器的内零头?它与页面大小有什么关系?

13. 为什么在页式存储器中实现程序共享时,必须对共享程序给出相同的页号?

14. 分页管理有哪几种形式?它们之间有什么区别?

15. 什么是虚拟存储器?虚拟存储器有哪些优点?

16. 叙述实现虚拟存储器的基本原理。

17. 采用页式存储器就是虚拟存储器吗?为什么?实现虚拟存储器的硬件与软件应增加什么功能?

18. 虚拟存储器的容量可以大于内存容量加外存容量的总和吗?

19. 简述请求分页虚拟存储中页表有哪些数据项?每项的作用是什么?

20. 请求分页虚拟存储系统中有哪几种置换策略?它们是如何实现的?

21. 如果一个进程在执行过程中,按下列的页号依次访问内存:1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6。进程固定占用 4 个内存页框(块),试问分别采用 FIFO、LRU、Clock 和 OPT 算法时,各产生多少次缺页中断?并计算相应的缺页中断率,同时写出在这 4 种调度算法下产生缺页中断时淘汰的页面号和在内存的页面号。

22. 假设一个进程固定分配 5 个页框,该进程的各页的装载时间、最近访问时间、*R* 位和 *M* 位信息如表 5-8 所示。

表 5-8　各页信息

页　　号	装 载 时 间	最近访问时间	*R*	*M*
0	125	280	1	0
1	225	260	0	1
2	150	270	0	0
3	100	290	1	1

当页号 4 要访问时,请问:

(1) FIFO 页面置换算法应置换哪个页面?

(2) LRU 页面置换算法应置换哪个页面?

(3) NRU 页面置换算法应置换哪个页面?

(4) 第二次机会页面置换算法应置换哪个页面?

23. 假设虚拟地址页访问流包含一个重复访问的长的页面访问序列,该序列偶尔会出现随机的页面访问,如访问序列为 0,1,…,511,428,0,1,…,511,202,0,1,…包含一个重复的访问页面访问序列 0,1,…,511 并伴随着随机的访问页面 428,202。请思考回答以下问题。

(1) 为什么标准的 FIFO、LRU、Clock 页面置换算法对于给定页框数小于序列长度的页面分配时其处理负载不是很好?

(2) 如果分给进程可达到 500 个页框,请给出一个比 FIFO、LRU 或 Clock 执行性能更好的页面置换方案。

24. 什么是扩充内存? 如何进行扩充内存管理? 各种扩充内存管理有何利弊? 它们各自较适合哪些情形?

25. 段式存储管理有什么优缺点? 它与页式存储管理的主要区别是什么?

26. 在段式存储器中实现程序共享时,共享段的段号是否一定要相同?

27. 叙述段页式虚拟存储管理的优缺点。

28. UNIX 怎样组织管理进程的虚拟存储空间?

29. 请叙述 Linux 的内存空间的管理思想。

30. Linux 物理内存管理使用 Buddy 算法实现,请分析这种方法的优点与不足之处。

31. 以分页式虚拟存储管理为例,简要叙述内存管理设计需考虑的问题与应对策略。

32. 简述 OpenHarmony LiteOS-M 中 membox 内存管理的特点和使用场景(liteos_m/kernel/src/mm/los_membox.c)。

33. 简述 OpenHarmony LiteOS-M 中 mempool 内存管理的特点和使用场景(liteos_m/kernel/src/mm/los_memory.c)。