



第一章 操作系统引论

1.1 操作系统的目标和作用

1.2 操作系统的发展过程

1.3 操作系统的基本特性



BACK



1.1 操作系统的目标 and 作用

1.1.1 操作系统的目标

1. 有效性

在早期(20世纪50~60年代), 由于计算机系统非常昂贵, 操作系统最重要的目标无疑是有效性。事实上, 那时有效性是推动操作系统发展最主要的动力。正因如此, 现在的大多数操作系统书籍, 都着重于介绍如何提高计算机系统的资源利用率和系统的吞吐量问题。操作系统的有效性可包含如下两方面的含意:





(1) 提高系统资源利用率。在未配置OS的计算机系统中，诸如CPU、I/O设备等各种资源，都会因它们经常处于空闲状态而得不到充分利用；内存及外存中所存放的数据太少或者无序而浪费了大量的存储空间。配置了OS之后，可使CPU和I/O设备由于能保持忙碌状态而得到有效的利用，且可使内存和外存中存放的数据因有序而节省了存储空间。

(2) 提高系统的吞吐量。操作系统还可以通过合理地组织计算机的工作流程，而进一步改善资源的利用率，加速程序的运行，缩短程序的运行周期，从而提高系统的吞吐量。





2. 方便性

配置OS后可使计算机系统更容易使用。一个未配置OS的计算机系统是极难使用的，因为计算机硬件只能识别0和1这样的机器代码。用户要直接在计算机硬件上运行自己所编写的程序，就必须用机器语言书写程序；用户要想输入数据或打印数据，也都必须自己用机器语言书写相应的输入程序或打印程序。如果我们在计算机硬件上配置了OS，用户便可通过OS所提供的各种命令来使用计算机系统。比如，用编译命令可方便地把用户用高级语言书写的程序翻译成机器代码，大大地方便了用户，从而使计算机变得易学易用。





3. 可扩充性

随着VLSI技术和计算机技术的迅速发展，计算机硬件和体系结构也随之得到迅速发展，相应地，它们也对OS提出了更高的功能和性能要求。此外，多处理机系统、计算机网络，特别是Internet的发展，又对OS提出了一系列更新的要求。因此，OS必须具有很好的可扩充性，方能适应计算机硬件、体系结构以及应用发展的要求。这就是说，现代OS应采用新的OS结构，如微内核结构和客户服务器模式，以便于方便地增加新的功能和模块，并能修改老的功能和模块。关于新的OS结构将在本章最后一节中介绍。





4. 开放性

自20世纪80年代以来，由于计算机网络的迅速发展，特别是Internet的应用的日益普及，使计算机操作系统的应用环境已由单机封闭环境转向开放的网络环境。为使来自不同厂家的计算机和设备能通过网络加以集成化，并能正确、有效地协同工作，实现应用的可移植性和互操作性，要求操作系统必须提供统一的开放环境，进而要求OS具有开放性。

开放性是指系统能遵循世界标准规范，特别是遵循开放系统互连(OSI)国际标准。凡遵循国际标准所开发的硬件和软件，均能彼此兼容，可方便地实现互连。开放性已成为20世纪90年代以后计算机技术的一个核心问题，也是一个新推出的系统或软件能否被广泛应用的至关重要的因素。





1.1.2 操作系统的作用

1. OS作为用户与计算机硬件系统之间的接口

OS作为用户与计算机硬件系统之间接口的含义是：OS处于用户与计算机硬件系统之间，用户通过OS来使用计算机系统。或者说，用户在OS帮助下，能够方便、快捷、安全、可靠地操纵计算机硬件和运行自己的程序。应注意，OS是一个系统软件，因而这种接口是软件接口。图1-1是OS作为接口的示意图。由图可看出，用户可通过以下三种方式使用计算机。



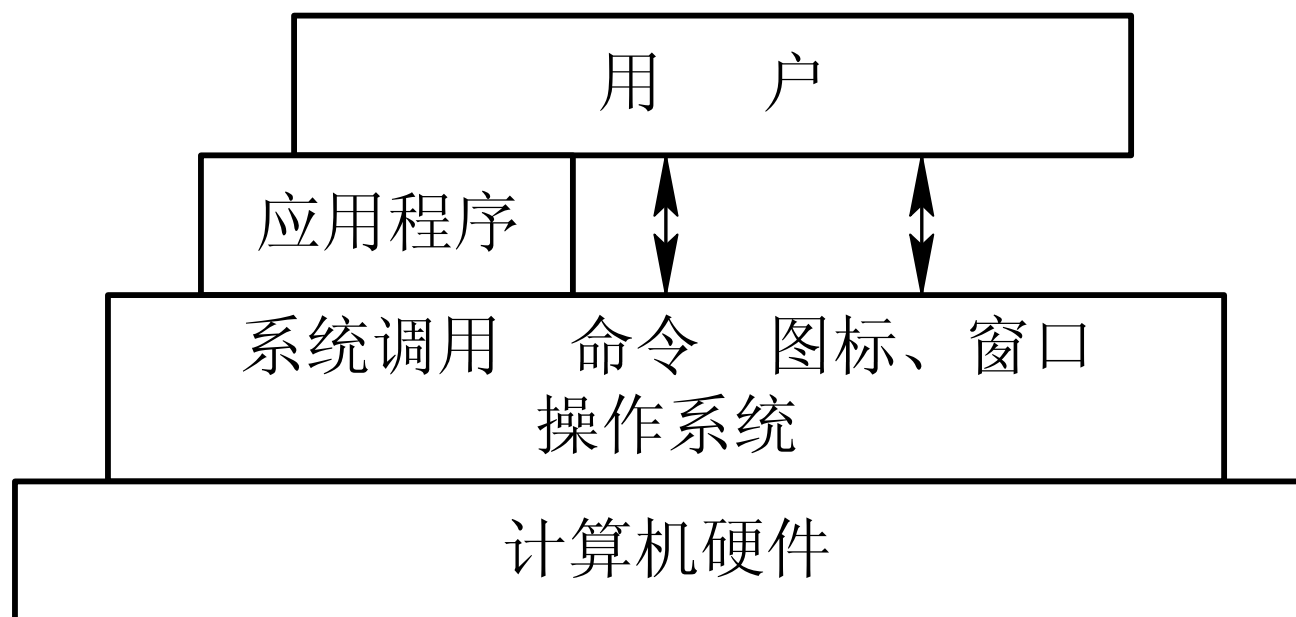


图1-1 OS作为接口的示意图





(1) 命令方式。这是指由OS提供了一组联机命令接口，以允许用户通过键盘输入有关命令来取得操作系统的服务，并控制用户程序的运行。

(2) 系统调用方式。OS提供了一组系统调用，用户可在自己的应用程序中通过相应的系统调用，来实现与操作系统的通信，并取得它的服务。

(3) 图形、窗口方式。这是当前使用最为方便、最为广泛的接口，它允许用户通过屏幕上的窗口和图标来实现与操作系统的通信，并取得它的服务。





2. OS作为计算机系统资源的管理者

在一个计算机系统中，通常都含有各种各样的硬件和软件资源。归纳起来可将资源分为四类：处理器、存储器、I/O设备以及信息(数据和程序)。相应地，OS的主要功能也正是针对这四类资源进行有效的管理，即：处理机管理，用于分配和控制处理机；存储器管理，主要负责内存的分配与回收；I/O设备管理，负责I/O设备的分配与操纵；文件管理，负责文件的存取、共享和保护。可见，OS的确是计算机系统资源的管理者。事实上，当今世界上广为流行的一个关于OS作用的观点，正是把OS作为计算机系统的资源管理者。





3. OS实现了对计算机资源的抽象

对于一个完全无软件的计算机系统(即裸机),它向用户提供的是实际硬件接口(物理接口),用户必须对物理接口的实现细节有充分的了解,并利用机器指令进行编程,因此该物理机器必定是难以使用的。为了方便用户使用I/O设备,人们在裸机上覆盖上一层I/O设备管理软件,如图1-2所示,由它来实现对I/O设备操作的细节,并向上提供一组I/O操作命令,如Read和Write命令,用户可利用它来进行数据输入或输出,而无需关心I/O是如何实现的。此时用户所看到的机器将是一台比裸机功能更强、使用更方便的机器。这就是说,在裸机上铺设的I/O软件隐藏了对I/O设备操作的具体细节,向上提供了一组抽象的I/O设备。



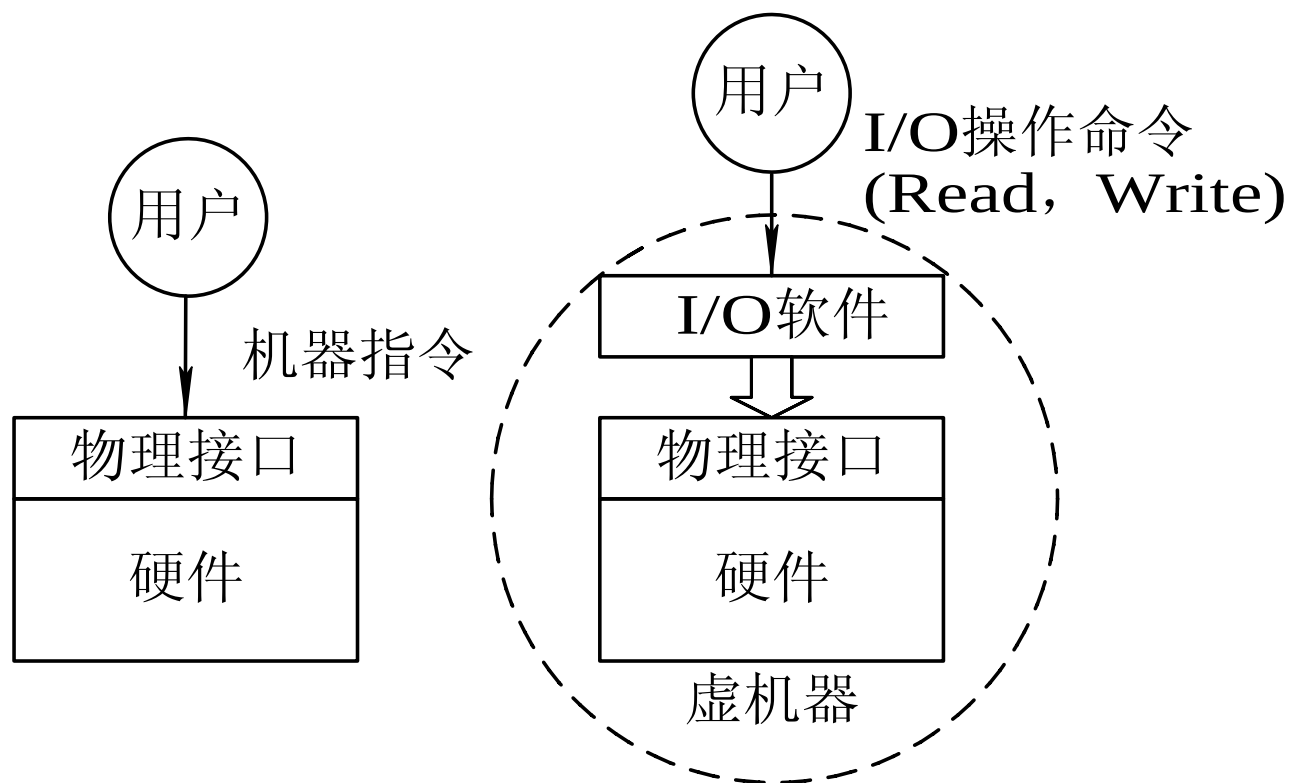


图1-2 I/O软件隐藏了I/O操作实现的细节





通常把覆盖了上述软件的机器称为扩充机器或虚机器。它向用户(进程)提供了一个对硬件操作的抽象模型，用户可利用抽象模型提供的接口使用计算机，而无需了解物理接口实现的细节，从而使用户更容易地使用计算机硬件资源。由该层软件实现了对计算机硬件操作的第一个层次的抽象。





为了方便用户使用文件系统，人们又在第一层软件上再覆盖上一层用于文件的管理软件，同样由它来实现对文件操作的细节，并向上提供一组对文件进行存取操作的命令，用户可利用这组命令进行文件的存取。此时，用户所看到的是一台功能更强、使用更方便的虚机器。该层软件实现了对硬件资源操作的第二个层次的抽象。而当人们又在文件管理软件上再覆盖一层面向用户的窗口软件后，用户便可在窗口环境下方便地使用计算机，形成一台功能更强的虚机器。





由此可知，OS是铺设在计算机硬件上的多层系统软件，它们不仅增强了系统的功能，而且还隐藏了对硬件操作的细节，由它们实现了对计算机硬件操作的多个层次的抽象。值得说明的是，对一个硬件在底层进行抽象后，在高层还可再次对该资源进行抽象，成为更高层的抽象模型。随着抽象层次的提高，抽象接口所提供的功能就越来越强，用户使用起来也更加方便。





1.1.3 推动操作系统发展的主要动力

1. 不断提高计算机资源的利用率

在计算机发展的初期，计算机系统特别昂贵，人们必须千方百计地提高计算机系统中各种资源的利用率，这就是OS最初发展的推动力。由此形成了能自动地对一批作业进行处理的多道批处理系统。在20世纪60和70年代，又分别出现了能有效提高I/O设备和CPU利用率的SPOOLing系统和改善存储器系统利用率的虚拟存储器技术，以及在网络环境下，在服务器上配置了允许所有网络用户访问的文件系统和数据库系统。





2. 方便用户

当资源利用率不高的问题得到基本解决后，用户在上机、调试程序时的不方便性便又成为主要矛盾。于是人们又想方设法改善用户上机、调试程序时的环境，这又成为继续推动OS发展的主要因素。随之便形成了允许进行人机交互的分时系统，或称为多用户系统。在20世纪90年代初出现了受到用户广泛欢迎的图形用户界面，极大地方便了用户使用计算机，使中小小学生都能很快地学会上机操作，这无疑会更加推动计算机的迅速普及。





3. 器件的不断更新换代

微电子技术的迅猛发展，推动着计算机器件，特别是微机芯片的不断更新，使得计算机的性能迅速提高，规模急剧扩大，从而推动了OS的功能和性能也迅速增强和提高。例如，当微机芯片由8位发展到16位、32位，进而又发展到64位时，相应的微机OS也就由8位发展到16位和32位，进而又发展到64位，此时相应OS的功能和性能也都有显著的增强和提高。





在多台处理机快速发展的同时，外部设备也在迅速发展。例如，早期的磁盘系统十分昂贵，只能配置在大型机中。随着磁盘价格的不断降低且小型化，很快在中、小型机以及微型机上也无一例外地配置了磁盘系统，而且其容量还远比早期配置在大型机上的大得多。现在的微机操作系统(如 Windows XP)能支持种类非常多的外部设备，除了传统的外设外，还可以支持光盘、移动硬盘、闪存盘、扫描仪等。





4. 计算机体系结构的不断发展

计算机体系结构的发展，也不断推动着OS的发展并产生新的操作系统类型。例如，当计算机由单处理机系统发展为多处理机系统时，相应地，操作系统也就由单处理机OS发展为多处理机OS。又如，当出现了计算机网络后，配置在计算机网络上的网络操作系统也就应运而生，它不仅能有效地管理好网络中的共享资源，而且还向用户提供了许多网络服务。





1.2 操作系统的发展过程

1.2.1 无操作系统的计算机系统

1. 人工操作方式

从第一台计算机诞生(1945年)到20世纪50年代中期的计算机,属于第一代计算机。此时的计算机是利用成千上万个真空管做成的,它的运行速度仅为每秒数千次,但体积却十分庞大,且功耗也非常高。这时还未出现OS。计算机操作是由用户(即程序员)采用人工操作方式直接使用计算机硬件系统,即由程序员将事先已穿孔(对应于程序和数据)的纸带(或卡片)装入纸带输入机(或卡片输入机),再启动它们将程序和数据输入计算机,然后启动计算机运行。当程序运行完毕并取走计算结果之后,才让下一个用户上机。这种人工操作方式有以下两方面的缺点:





(1) 用户独占全机。此时，计算机及其全部资源只能由上机用户独占。

(2) CPU等待人工操作。当用户进行装带(卡)、卸带(卡)等人工操作时，CPU及内存等资源是空闲的。

可见，人工操作方式严重降低了计算机资源的利用率，此即所谓的人机矛盾。随着CPU速度的提高和系统规模的扩大，人机矛盾变得日趋严重。此外，随着CPU速度的迅速提高而I/O设备的速度却提高缓慢，这又使CPU与I/O设备之间速度不匹配的矛盾更加突出。为了缓和此矛盾，曾先后出现了通道技术、缓冲技术，但都未能很好地解决上述矛盾，直至后来又引入了脱机输入/输出技术，才获得了较为令人满意的结果。





2. 脱机输入/输出方式

为了解决人机矛盾及CPU和I/O设备之间速度不匹配的矛盾，20世纪50年代末出现了脱机输入/输出(Off-Line I/O)技术。该技术是事先将装有用户程序和数据的数据带(或卡片)装入纸带输入机(或卡片机)，在一台外围机的控制下，把纸带(卡片)上的数据(程序)输入到磁带上。当CPU需要这些程序和数据时，再从磁带上将其高速地调入内存。



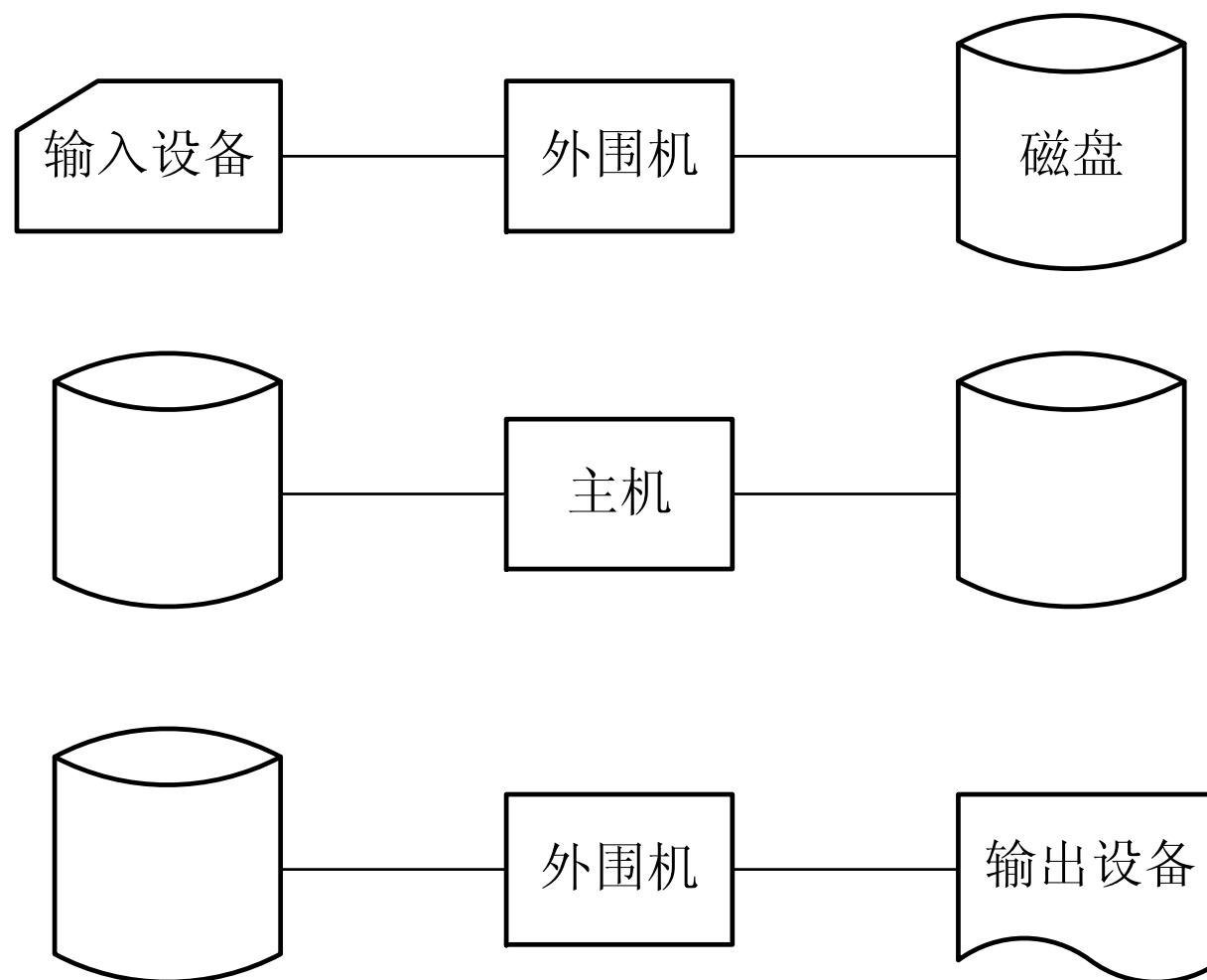


图1-3 脱机I/O示意图





类似地，当CPU需要输出时，可由CPU直接高速地把数据从内存送到磁带上，然后再在另一台外围机的控制下，将磁带上的结果通过相应的输出设备输出。图1-3示出了脱机输入/输出过程。由于程序和数据的输入和输出都是在外围机的控制下完成的，或者说，它们是在脱离主机的情况下进行的，故称为脱机输入/输出方式；反之，在主机的直接控制下进行输入/输出的方式称为联机。





输入/输出 (On-Line I/O) 方式。这种脱机I/O方式的主要优点如下：

(1) 减少了CPU的空闲时间。装带(卡)、卸带(卡)以及将数据从低速I/O设备送到高速磁带(或盘)上，都是在脱机情况下进行的，并不占用主机时间，从而有效地减少了CPU的空闲时间，缓和了人机矛盾。

(2) 提高了I/O速度。当CPU在运行中需要数据时，是直接从高速的磁带或磁盘上将数据调入内存的，不再是从低速I/O设备上输入，极大地提高了I/O速度，从而缓和了CPU和I/O设备速度不匹配的矛盾，进一步减少了CPU的空闲时间。





1.2.2 单道批处理系统

1. 单道批处理系统的处理过程

上世纪50年代中期发明了晶体管，人们开始用晶体管替代真空管来制作计算机，从而出现了第二代计算机。它不仅使计算机的体积大大减小，功耗显著降低，同时可靠性也得到大幅度提高，使计算机已具有推广应用的价值，但计算机系统仍非常昂贵。为了能充分地利用它，应尽量让该系统连续运行，以减少空闲时间。为此，通常是把一批作业以脱机方式输入到磁带上，并在系统中配上监督程序(Monitor)，在它的控制下使这批作业能一个接一个地连续处理。





其自动处理过程是：首先，由监督程序将磁带上的第一个作业装入内存，并把运行控制权交给该作业。当该作业处理完成时，又把控制权交还给监督程序，再由监督程序把磁带(盘)上的第二个作业调入内存。计算机系统就这样自动地一个作业一个作业地进行处理，直至磁带(盘)上的所有作业全部完成，这样便形成了早期的批处理系统。由于系统对作业的处理都是成批地进行的，且在内存中始终只保持一道作业，故称此系统为单道批处理系统(Simple Batch Processing System)。图1-4示出了单道批处理系统的处理流程。





第一章 操作系统引论

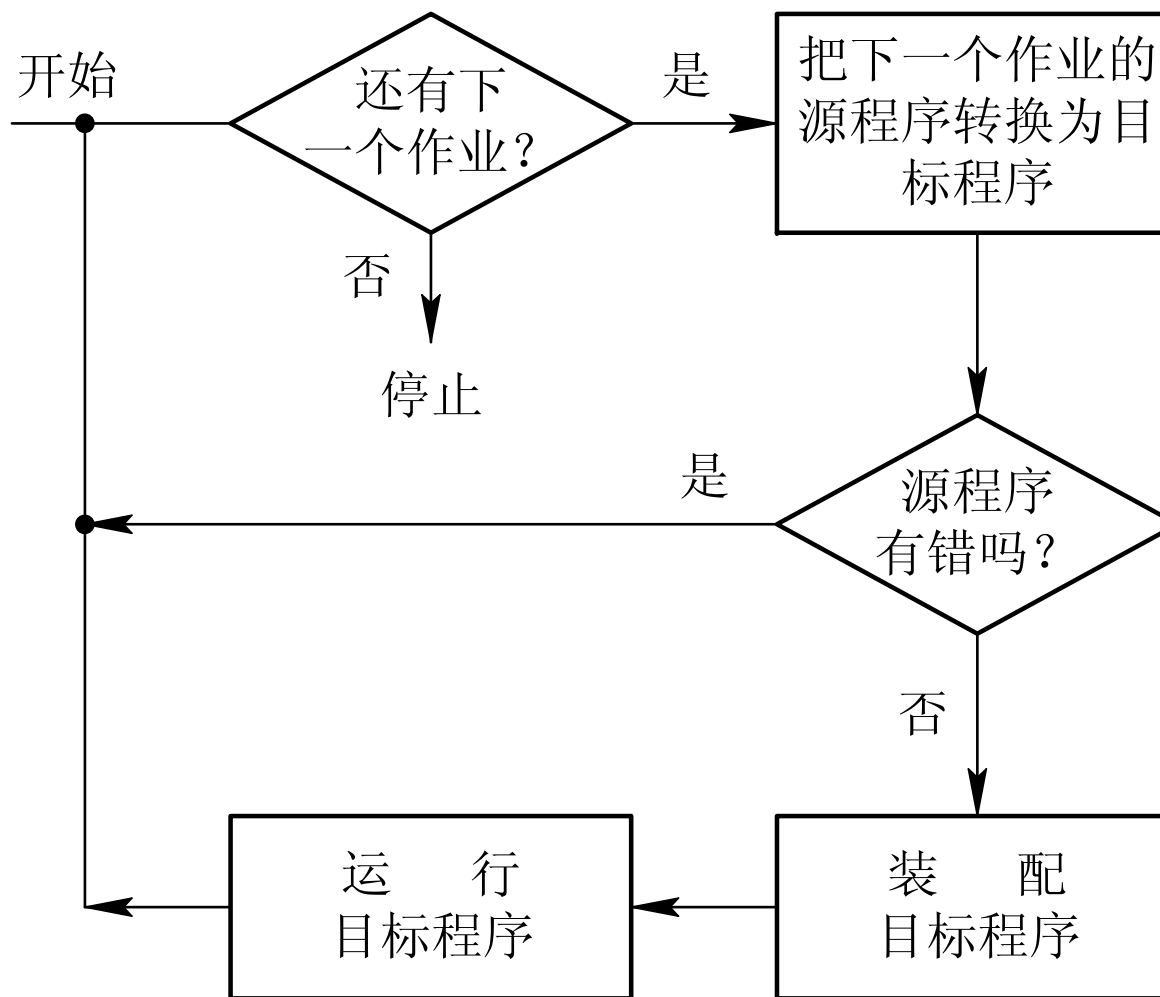


图1-4 单道批处理系统的处理流程





2. 单道批处理系统的特征

单道批处理系统是最早出现的一种OS。严格地说，它只能算作是OS的前身而并非是现在人们所理解的OS。尽管如此，该系统比起人工操作方式的系统已有很大进步。该系统的主要特征如下：

(1) 自动性。在顺利情况下，在磁带上的一批作业能自动地逐个地依次运行，而无需人工干预。





(2) 顺序性。磁带上的各道作业是顺序地进入内存，各道作业的完成顺序与它们进入内存的顺序，在正常情况下应完全相同，亦即先调入内存的作业先完成。

(3) 单道性。内存中仅有一道程序运行，即监督程序每次从磁带上只调入一道程序进入内存运行，当该程序完成或发生异常情况时，才换入其后继程序进入内存运行。





1.2.3 多道批处理系统

1. 多道程序设计的基本概念

在单道批处理系统中，内存中仅有一道作业，它无法充分利用系统中的所有资源，致使系统性能较差。为了进一步提高资源的利用率和系统吞吐量，在20世纪60年代中期又引入了多道程序设计技术，由此而形成了多道批处理系统 (Multiprogrammed Batch Processing System)。在该系统中，用户所提交的作业都先存放在外存上并排成一个队列，称为“后备队列”；然后，由作业调度程序按一定的算法从后备队列中选择若干个作业调入内存，使它们共享CPU和系统中的各种资源。具体地说，在OS中引入多道程序设计技术可带来以下好处：



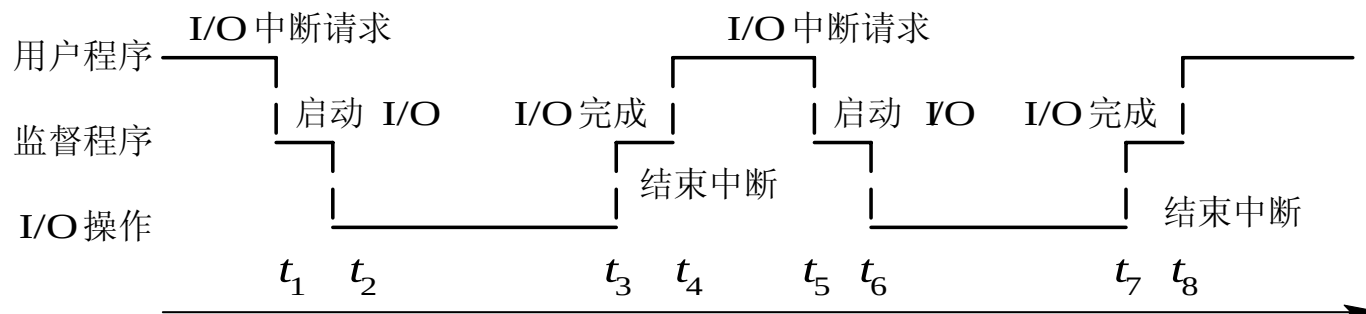


(1) 提高CPU的利用率。当内存中仅有一道程序时，每逢该程序在运行中发出I/O请求后，CPU空闲，必须在其I/O完成后CPU才继续运行；尤其因I/O设备的低速性，更使CPU的利用率显著降低。图1-5(a)示出了单道程序的运行情况，从图中可以看出：在 $t_2 \sim t_3$ 、 $t_6 \sim t_7$ 时间间隔内CPU空闲。在引入多道程序设计技术后，由于同时在内存中装有若干道程序，并使它们交替地运行，这样，当正在运行的程序因I/O而暂停执行时，系统可调度另一道程序运行，从而保持了CPU处于忙碌状态。图1-5(b)示出了四道程序时的运行情况。

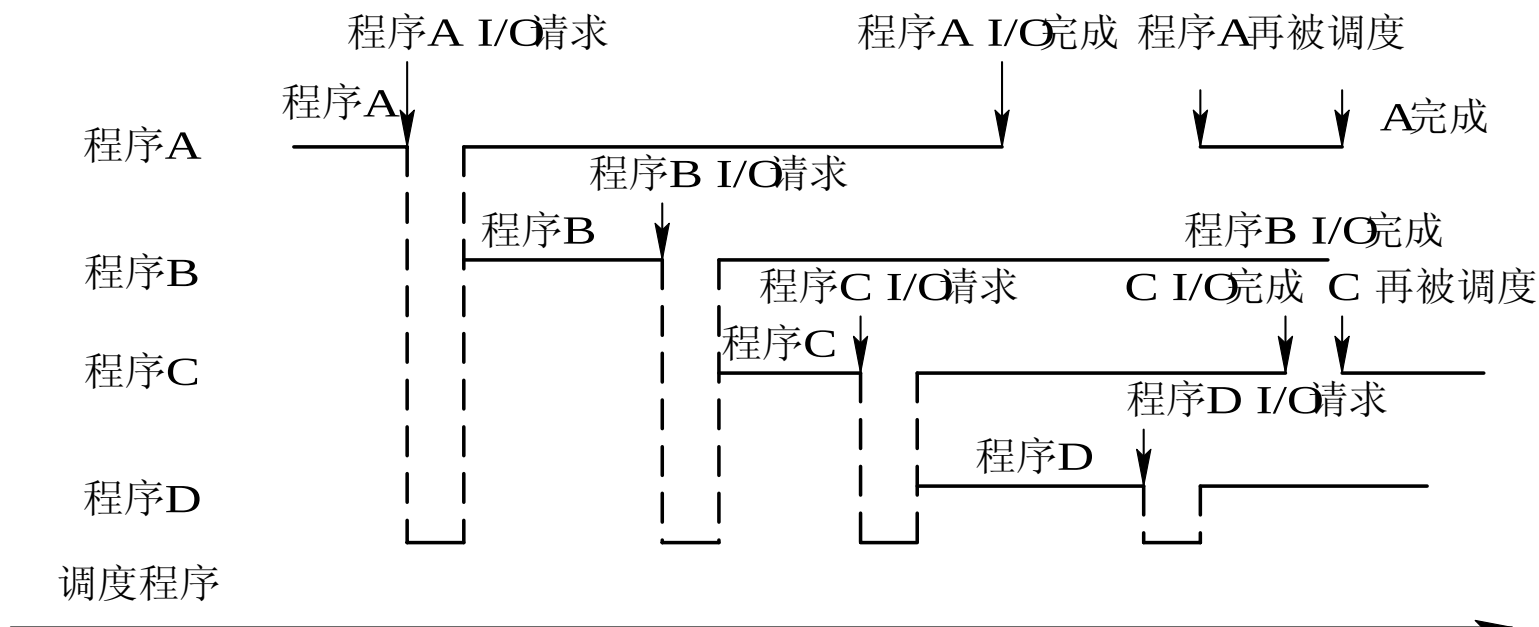




第一章 操作系统引论



(a) 单道程序运行情况



(b) 四道程序运行情况

图1-5 单道和多道程序运行情况



(2) 可提高内存和I/O设备利用率。为了能运行较大的作业，通常内存都具有较大容量，但由于80%以上的作业都属于中小型，因此在单道程序环境下，也必定造成内存的浪费。类似地，对于系统中所配置的多种类型的I/O设备，在单道程序环境下也不能充分利用。如果允许在内存中装入多道程序，并允许它们并发执行，则无疑会大大提高内存和I/O设备的利用率。

(3) 增加系统吞吐量。在保持CPU、I/O设备不断忙碌的同时，也必然会大幅度地提高系统的吞吐量，从而降低作业加工所需的费用。





2. 多道批处理系统的优缺点

虽然早在20世纪60年代就已出现了多道批处理系统，但至今它仍是三大基本操作系统类型之一。在大多数大、中、小型机中都配置了它，说明它具有其它类型OS所不具有的优点。多道批处理系统的主要优缺点如下：

(1) 资源利用率高。由于在内存中驻留了多道程序，它们共享资源，可保持资源处于忙碌状态，从而使各种资源得以充分利用。





第一章 操作系统引论



(2) 系统吞吐量大。系统吞吐量是指系统在单位时间内所完成的总工作量。能提高系统吞吐量的主要原因可归结为：第一，CPU和其它资源保持“忙碌”状态；第二，仅当作业完成时或运行不下去时才进行切换，系统开销小。





(3) 平均周转时间长。作业的周转时间是指从作业进入系统开始，直至其完成并退出系统为止所经历的时间。在批处理系统中，由于作业要排队，依次进行处理，因而作业的周转时间较长，通常需几个小时，甚至几天。

(4) 无交互能力。用户一旦把作业提交给系统后，直至作业完成，用户都不能与自己的作业进行交互，这对修改和调试程序是极不方便的。





3. 多道批处理系统需要解决的问题

多道批处理系统是一种有效、但十分复杂的系统。为使系统中的多道程序间能协调地运行，必须解决下述一系列问题。

(1) 处理机管理问题。在多道程序之间，应如何分配被它们共享的处理机，使CPU既能满足各程序运行的需要，又能提高处理机的利用率，以及一旦把处理机分配给某程序后，又应在何时收回等一系列问题，属于处理机管理问题。





(2) 内存管理问题。应如何为每道程序分配必要的内存空间，使它们“各得其所”且不致因相互重叠而丢失信息，以及应如何防止因某道程序出现异常情况而破坏其它程序等问题，就是内存管理问题。

(3) I/O设备管理问题。系统中可能具有多种类型的I/O设备供多道程序所共享，应如何分配这些I/O设备，如何做到既方便用户对设备的使用，又能提高设备的利用率，这就是I/O设备管理问题。





(4) 文件管理问题。在现代计算机系统中，通常都存放着大量的程序和数据(以文件形式存在)，应如何组织这些程序和数据，才能使它们既便于用户使用，又能保证数据的安全性和一致性，这些属于文件管理问题。

(5) 作业管理问题。对于系统中的各种应用程序，其中有的属于计算型，即以计算为主的程序；有的属于I/O型，即以I/O为主的程序；又有些作业既重要又紧迫；而有的作业则要求系统能及时响应，这时应如何组织这些作业，这便是作业管理问题。





1.2.4 分时系统

1. 分时系统的产生

分时系统(Time Sharing System)与多道批处理系统之间有着截然不同的性能差别，它能很好地将一台计算机提供给多个用户同时使用，提高计算机的利用率。它被经常应用于查询系统中，满足许多查询用户的需要。用户的需求具体表现在以下几个方面：

(1) 人-机交互。每当程序员写好一个新程序时，都需要上机进行调试。由于新编程序难免有些错误或不当之处需要修改，因而希望能像早期使用计算机时一样对它进行直接控制，并能以边运行边修改的方式，对程序中的错误进行修改，亦即，希望能进行人-机交互。





(2) 共享主机。在20世纪60年代计算机非常昂贵，不可能像现在这样每人独占一台微机，而只能是由多个用户共享一台计算机，但用户在使用机器时应能够像自己独占计算机一样，不仅可以随时与计算机交互，而且应感觉不到其他用户也在使用该计算机。

(3) 便于用户上机。在多道批处理系统中，用户上机前必须把自己的作业邮寄或亲自送到机房。这对于用户尤其是远地用户来说是十分不便的。用户希望能通过自己的终端直接将作业传送到机器上进行处理，并能对自己的作业进行控制。





2. 分时系统实现中的关键问题

(1) 及时接收。要及时接收用户键入的命令或数据并不困难，为此，只需在系统中配置一个多路卡。例如，当要在主机上连接8个终端时，须配置一个8用户的多路卡。多路卡的作用是使主机能同时接收各用户从终端上输入的数据。此外，还须为每个终端配置一个缓冲区，用来暂存用户键入的命令(或数据)。





(2) 及时处理。人机交互的关键，是使使用户键入命令后能及时地控制自己作业的运行，或修改自己的作业。为此，各个用户的作业都必须在内存中，且应能频繁地获得处理机而运行；否则，用户键入的命令将无法作用到自己的作业上。前面介绍的批处理系统是无法实现人机交互的。因为通常大多数作业都还驻留在外存上，即使是已调入内存的作业，也经常要经过较长时间的等待后方能运行，因而使用户键入的命令很难及时作用到自己的作业上。





由此可见，为实现人机交互，必须彻底地改变原来批处理系统的运行方式。首先，用户作业不能先进入磁盘，然后再调入内存。因为作业在磁盘上不能运行，当然用户也无法与机器交互，因此，作业应直接进入内存。其次，不允许一个作业长期占用处理机，直至它运行结束或出现I/O请求后，方才调度其它作业运行。为此，应该规定每个作业只运行一个很短的时间(例如0.1秒钟，通常把这段时间称为时间片)，然后便暂停该作业的运行，并立即调度下一个程序运行。如果在不长的时间(如3秒)内能使所有的用户作业都执行一次(一个时间片的时间)，便可使每个用户都能及时地与自己的作业交互，从而可使用户的请求得到及时响应。





3. 分时系统的特征

分时系统与多道批处理系统相比，具有非常明显的不同特征，由上所述可以归纳成以下四个特点：

(1) 多路性。允许在一台主机上同时联接多台联机终端，系统按分时原则为每个用户服务。宏观上，是多个用户同时工作，共享系统资源；而微观上，则是每个用户作业轮流运行一个时间片。多路性即同时性，它提高了资源利用率，降低了使用费用，从而促进了计算机更广泛的应用。





(2) 独立性。每个用户各占一个终端，彼此独立操作，互不干扰。因此，用户所感觉到的，就像是他一人独占主机。

(3) 及时性。用户的请求能在很短的时间内获得响应。此时间间隔是以人们所能接受的等待时间来确定的，通常为1~3秒钟。

(4) 交互性。用户可通过终端与系统进行广泛的人机对话。其广泛性表现在：用户可以请求系统提供多方面的服务，如文件编辑、数据处理和资源共享等。





1.2.5 实时系统

所谓“实时”，是表示“及时”，而实时系统(Real Time System)是指系统能及时(或即时)响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有实时任务协调一致地运行。

1. 应用需求

虽然多道批处理系统和分时系统已获得较为令人满意的资源利用率和响应时间，从而使计算机的应用范围日益扩大，但它们仍然不能满足以下某些应用领域的需要。





(1) 实时控制。当把计算机用于生产过程的控制，以形成以计算机为中心的控制系统时，系统要求能实时采集现场数据，并对所采集的数据进行及时处理，进而自动地控制相应的执行机构，使某些(个)参数(如温度、压力、方位等)能按预定的规律变化，以保证产品的质量和提高产量。类似地，也可将计算机用于对武器的控制，如火炮的自动控制系统、飞机的自动驾驶系统，以及导弹的制导系统等。此外，随着大规模集成电路的发展，已制作出各种类型的芯片，并可将这些芯片嵌入到各种仪器和设备中，用来对设备的工作进行实时控制，这就构成了所谓的智能仪器和设备。在这些设备中也需要配置某种类型的、能进行实时控制的系统。通常把用于进行实时控制的系统称为实时系统。





(2) 实时信息处理。通常，人们把用于对信息进行实时处理的系统称为实时信息处理系统。该系统由一台或多台主机通过通信线路连接到成百上千个远程终端上，计算机接收从远程终端上发来的服务请求，根据用户提出的请求对信息进行检索和处理，并在很短的时间内为用户做出正确的响应。典型的实时信息处理系统有早期的飞机或火车的订票系统、情报检索系统等。





2. 实时任务

1) 按任务执行时是否呈现周期性来划分

(1) 周期性实时任务。外部设备周期性地发出激励信号给计算机，要求它按指定周期循环执行，以便周期性地控制某外部设备。

(2) 非周期性实时任务。外部设备所发出的激励信号并无明显的周期性，但都必须联系着一个截止时间(Deadline)。它又可分为开始截止时间(某任务在某时间以前必须开始执行)和完成截止时间(某任务在某时间以前必须完成)两部分。





2) 根据对截止时间的要求来划分

(1) 硬实时任务(Hard real-time Task)。系统必须满足任务对截止时间的要求，否则可能出现难以预测的结果。

(2) 软实时任务(Soft real-time Task)。它也联系着一个截止时间，但并不严格，若偶尔错过了任务的截止时间，对系统产生的影响也不会太大。





3. 实时系统与分时系统特征的比较

(1) 多路性。实时信息处理系统也按分时原则为多个终端用户服务。实时控制系统的多路性则主要表现在系统周期性地对多路现场信息进行采集，以及对多个对象或多个执行机构进行控制。而分时系统中的多路性则与用户情况有关，时多时少。

(2) 独立性。实时信息处理系统中的每个终端用户在向实时系统提出服务请求时，是彼此独立地操作，互不干扰；而实时控制系统中，对信息的采集和对对象的控制也都是彼此互不干扰。





第一章 操作系统引论



(3) 及时性。实时信息处理系统对实时性的要求与分时系统类似，都是以人所能接受的等待时间来确定的；而实时控制系统的及时性，则是以控制对象所要求的开始截止时间或完成截止时间来确定的，一般为秒级到毫秒级，甚至有的要低于100微秒。

(4) 交互性。实时信息处理系统虽然也具有交互性，但这里人与系统的交互仅限于访问系统中某些特定的专用服务程序。它不像分时系统那样能向终端用户提供数据处理和资源共享等服务。





(5) 可靠性。分时系统虽然也要求系统可靠，但相比之下，实时系统则要求系统具有高度的可靠性。因为任何差错都可能带来巨大的经济损失，甚至是无法预料的灾难性后果，所以在实时系统中，往往都采取了多级容错措施来保障系统的安全性及数据的安全性。





1.2.6 微机操作系统的发展

1. 单用户单任务操作系统

单用户单任务操作系统的含义是，只允许一个用户上机，且只允许用户程序作为一个任务运行。这是最简单的微机操作系统，主要配置在8位和16位微机上。最有代表性的单用户单任务微机操作系统是CP/M和MS-DOS。





1) CP/M

1974年第一代通用8位微处理器芯片Intel 8080出现后的第二年，Digital Research公司就开发出带有软盘系统的8位微机操作系统。1977年Digital Research公司对CP/M进行了重写，使其可配置在以Intel 8080、8085、Z80等8位芯片为基础的多种微机上。1979年又推出带有硬盘管理功能的CP/M 2.2版本。由于CP/M具有较好的体系结构，可适应性强，且具有可移植性以及易学易用等优点，使之在8位微机中占据了统治地位。





2) MS-DOS

1981年IBM公司首次推出了IBM-PC个人计算机(16位微机)，在微机中采用了微软公司开发的MS-DOS(Disk Operating System)操作系统，该操作系统在CP/M的基础上进行了较大的扩充，使其在功能上有很大的增强。1983年IBM推出PC/AT(配有Intel 80286芯片)，相应地，微软又开发出MS-DOS 2.0版本，它不仅能支持硬盘设备，还采用了树形目录结构的文件系统。1987年又宣布了MS-DOS 3.3版本。从MS-DOS 1.0到3.3为止的DOS版本都属于单用户单任务操作系统，内存被限制在640 KB。从1989年到1993年又先后推出了多个MS-DOS版本，它们都可以配置在Intel 80386、80486等32位微机上。从20世纪80年代到90年代初，由于MS-DOS性能优越而受到当时用户的广泛欢迎，成为事实上的16位单用户单任务操作系统标准。





2. 单用户多任务操作系统

单用户多任务操作系统的含义是，只允许一个用户上机，但允许用户把程序分为若干个任务，使它们并发执行，从而有效地改善了系统的性能。目前在32位微机上配置的操作系统基本上都是单用户多任务操作系统，其中最有代表性的是由微软公司推出的Windows。1985年和1987年微软公司先后推出了Windows 1.0和Windows 2.0版本操作系统，由于当时的硬件平台还只是16位微机，对1.0和2.0版本不能很好的支持。1990年微软公司又发布了Windows 3.0版本，随后又宣布了Windows 3.1版本，它们主要是针对386和486等32位微机开发的，较之以前的操作系统有着重大的改进，引入了友善的图形用户界面，支持多任务和扩展内存的功能，使计算机更好使用，从而成为386和486等微机的主流操作系统。





1995年微软公司推出了Windows 95，它较之以前的Windows 3.1有许多重大改进，采用了全32位的处理技术，并兼容以前的16位应用程序，在该系统中还集成了支持Internet的网络功能。1998年微软公司又推出了Windows 95的改进版Windows 98，它已是最后一个仍然兼容以前的16位应用程序的Windows，其最主要的改进是把微软公司自己开发的Internet浏览器整合到系统中，大大方便了用户上网浏览，另一个特点是增加了对多媒体的支持。2001年微软又发布了32位版本的Windows XP，同时提供了家用和商业工作站两种版本，它是当前使用最广泛的个人操作系统。2001年还发布了64位版本的Windows XP。





3. 多用户多任务操作系统

多用户多任务操作系统的含义是，允许多个用户通过各自的终端使用同一台机器，共享主机系统中的各种资源，而每个用户程序又可进一步分为几个任务，使它们能并发执行，从而可进一步提高资源利用率和系统吞吐量。在大、中和小型机中所配置的大多是多用户多任务操作系统，而在32位微机上也有不少是配置的多用户多任务操作系统，其中最有代表性的是UNIX OS。





UNIX OS是美国电报电话公司的Bell实验室在1969～1970年期间开发的，1979年推出来的UNIX V.7已被广泛应用于多种中、小型机上。随着微机性能的提高，人们又将UNIX移植到微机上。在1980年前后，将UNIX第7版本移植到Motorola公司的MC 680xx微机上，后来又将UNIX V7.0版本进行简化后移植到Intel 8080上，把它称为Xenix。现在最有影响的两个能运行在微机上的UNIX操作系统的变型是Solaris OS和Linux OS。





(1) Solaris OS: SUN公司于1982年推出的SUN OS 1.0是一个运行在Motorola 680x0平台上的UNIX OS。在1988年宣布的SUN OS 4.0把运行平台从早期的Motorola 680x0平台迁移到SPARC平台,并开始支持Intel公司的Intel 80x86; 1992年SUN发布了Solaris 2.0。从1998年开始, Sun公司推出64位操作系统Solaris 2.7和2.8,这几款操作系统在网络特性、互操作性、兼容性以及易于配置和管理方面均有很大的提高。

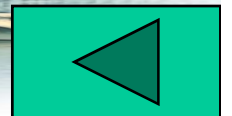




第一章 操作系统引论



(2) Linux OS: Linux 是UNIX的一个重要变种, 最初是由芬兰学生Linus Torvalds针对Intel 80386开发的。1991年在Internet网上发布第一个Linux版本, 由于源代码公开, 因此有很多人通过Internet与之合作, 使Linux的性能迅速提高, 其应用范围也日益扩大。相应地, 源代码也急剧膨胀, 此时它已是具有全面功能的UNIX系统, 大量在UNIX上运行的软件(包括1000多种实用工具软件和大量的网络软件)被移植到Linux上, 而且可以在主要的微机上运行, 如Intel 80x86 Pentium等。





1.3 操作系统的基本特性

1.3.1 并行性

1. 并行与并发

并行性和并发性(Concurrence)是既相似又有区别的两个概念，并行性是指两个或多个事件在同一时刻发生；而并发性是指两个或多个事件在同一时间间隔内发生。在多道程序环境下，并发性是指在一段时间内宏观上有多个程序在同时运行，但在单处理机系统中，每一时刻却仅能有一道程序执行，故微观上这些程序只能是分时地交替执行。倘若在计算机系统中有多个处理机，则这些可以并发执行的程序便可被分配到多个处理机上，实现并行执行，即利用每个处理机来处理一个可并发执行的程序，这样，多个程序便可同时执行。





2. 引入进程

应当指出，通常的程序是静态实体(Passive Entity)，在多道程序系统中，它们是不能独立运行的，更不能和其它程序并发执行。在操作系统中引入进程的目的，就是为了使多个程序能并发执行。例如，在一个未引入进程的系统中，在属于同一个应用程序的计算程序和I/O程序之间，两者只能是顺序执行，即只有在计算程序执行告一段落后，才允许I/O程序执行；反之，在程序执行I/O操作时，计算程序也不能执行，这意味着处理机处于空闲状态。但在引入进程后，若分别为计算程序和I/O程序各建立一个进程，则这两个进程便可并发执行。





由于在系统中具备使计算程序和I/O程序同时运行的硬件条件，因而可将系统中的CPU和I/O设备同时开动起来，实现并行工作，从而有效地提高了系统资源的利用率和系统吞吐量，并改善了系统的性能。引入进程的好处远不止于此，事实上可以在内存中存放多个用户程序，分别为它们建立进程后，这些进程可以并发执行，亦即实现前面所说的多道程序运行。这样便能极大地提高系统资源的利用率，增加系统的吞吐量。





为使多个程序能并发执行，系统必须分别为每个程序建立进程(Process)。简单说来，进程是指在系统中能独立运行并作为资源分配的基本单位，它是由一组机器指令、数据和堆栈等组成的，是一个能独立运行的活动实体。多个进程之间可以并发执行和交换信息。一个进程在运行时需要一定的资源，如CPU、存储空间及I/O设备等。





3. 引入线程

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。在引入线程的OS中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效地提高系统内多个程序间并发执行的程度。因而近年来推出的通用操作系统都引入了线程，以便进一步提高系统的并发性，并把它视作现代操作系统的一个重要标致。





1.3.2 共享性

1. 互斥共享方式

系统中的某些资源，如打印机、磁带机，虽然它们可以提供给多个进程(线程)使用，但为使所打印或记录的结果不致造成混淆，应规定在一段时间内只允许一个进程(线程)访问该资源。为此，系统中应建立一种机制，以保证对这类资源的互斥访问。当一个进程A要访问某资源时，必须先提出请求。如果此时该资源空闲，系统便可将之分配给请求进程A使用。此后若再有其它进程也要访问该资源时(只要A未用完)，则必须等待。仅当A进程访问完并释放该资源后，才允许另一进程对该资源进行访问。





第一章 操作系统引论



我们把这种资源共享方式称为互斥式共享，而把在一段时间内只允许一个进程访问的资源称为临界资源或独占资源。计算机系统的大多数物理设备，以及某些软件中所用的栈、变量和表格，都属于临界资源，它们要求被互斥地共享。为此，在系统中必需配置某种机制来保证诸进程互斥地使用独占资源。





2. 同时访问方式

系统中还有另一类资源，允许在一段时间内由多个进程“同时”对它们进行访问。这里所谓的“同时”，在单处理机环境下往往是宏观上的，而在微观上，这些进程可能是交替地对该资源进行访问。典型的可供多个进程“同时”访问的资源是磁盘设备，一些用重入码编写的文件也可以被“同时”共享，即若干个用户同时访问该文件。

并发和共享是操作系统的两个最基本的特征，它们又是互为存在的条件。一方面，资源共享是以程序(进程)的并发执行为条件的，若系统不允许程序并发执行，自然不存在资源共享问题；另一方面，若系统不能对资源共享实施有效管理，协调好诸进程对共享资源的访问，也必然影响到程序并发执行的程度，甚至根本无法并发执行。





1.3.3 虚拟技术

1. 时分复用技术

时分复用，亦即分时使用方式，它最早用于电信业中。为了提高信道的利用率，人们利用时分复用方式，将一条物理信道虚拟为多条逻辑信道，将每条信道供一对用户通话。在计算机领域中，广泛利用该技术来实现虚拟处理机、虚拟设备等，以提高资源的利用率。





1) 虚拟处理机技术

在虚拟处理机技术中，利用多道程序设计技术，为每道程序建立一个进程，让多道程序并发地执行，以此来分时使用一台处理机。此时，虽然系统中只有一台处理机，但它却能同时为多个用户服务，使每个终端用户都认为是有一个处理机在专门为他服务。亦即，利用多道程序设计技术，把一台物理上的处理机虚拟为多台逻辑上的处理机，在每台逻辑处理机上运行一道程序。我们把用户所感觉到的处理机称为虚拟处理器。





2) 虚拟设备技术

我们还可以通过虚拟设备技术，将一台物理I/O设备虚拟为多台逻辑上的I/O设备，并允许每个用户占用一台逻辑上的I/O设备，这样便可使原来仅允许在一段时间内由一个用户访问的设备(即临界资源)，变为在一段时间内允许多个用户同时访问的共享设备。例如，原来的打印机属于临界资源，而通过虚拟设备技术，可以把它变为多台逻辑上的打印机，供多个用户“同时”打印。





2. 空分复用技术

早在上世纪初，电信业中就使用频分复用技术来提高信道的利用率。它是将一个频率范围非常宽的信道，划分成多个频率范围较窄的信道，其中的任何一个频带都只供一对用户通话。早期的频分复用只能将一条物理信道划分为十几条到几十条话路，后来又很快发展成上万条话路，每条话路也只供一对用户通话。之后，在计算机中也使用了空分复用技术来提高存储空间的利用率。





1) 虚拟磁盘技术

通常在一台机器上只配置一台硬盘。我们可以通过虚拟磁盘技术将一台硬盘虚拟为多台虚拟磁盘，这样使用起来既方便又安全。虚拟磁盘技术也是采用了空分复用方式，即它将硬盘划分为若干个卷，例如1、2、3、4四个卷，再通过安装程序将它们分别安装在C、D、E、F四个逻辑驱动器上，这样，机器上便有了四个虚拟磁盘。当用户要访问D盘中的内容时，系统便会访问卷2中的内容。





2) 虚拟存储器技术

在单道程序环境下，处理机会有很多空闲时间，内存也会有很多空闲空间，显然，这会使处理机和内存的效率低下。如果说时分复用技术是利用处理机的空闲时间来运行其它的程序，使处理机的利用率得以提高，那么空分复用则是利用存储器的空闲空间来存放其它的程序，以提高内存的利用率。





但是，单纯的空分复用存储器只能提高内存的利用率，并不能实现在逻辑上扩大存储器容量的功能，必须引入虚拟存储技术才能达到此目地。而虚拟存储技术在本质上就是使内存分时复用。它可以使一道程序通过时分复用方式，在远小于它的内存空间中运行。例如，一个100 MB的应用程序可以运行在20 MB的内存空间。下一节将要介绍的用于实现内存扩充的“请求调入功能”和“置换功能”就是用于每次只把用户程序的一部分调入内存运行，这样便实现了用户程序的各个部分分时进入内存运行的功能。





应当着重指出：如果虚拟的实现是通过时分复用的方法来实现的，即对某一物理设备进行分时使用，设 N 是某物理设备所对应的虚拟的逻辑设备数，则每台虚拟设备的平均速度必然等于或低于物理设备速度的 $1/N$ 。类似地，如果是利用空分复用方法来实现虚拟，此时一台虚拟设备平均占用的空间必然也等于或低于物理设备所拥有空间的 $1/N$ 。





1.3.4 异步性

在多道程序环境下允许多个进程并发执行，但只有进程在获得所需的资源后方能执行。在单处理机环境下，由于系统中只有一台处理机，因而每次只允许一个进程执行，其余进程只能等待。当正在执行的进程提出某种资源要求时，如打印请求，而此时打印机正在为其它某进程打印，由于打印机属于临界资源，因此正在执行的进程必须等待，且放弃处理机，直到打印机空闲，并再次把处理机分配给该进程时，该进程方能继续执行。可见，由于资源等因素的限制，使进程的执行通常都不是“一气呵成”，而是以“走走停停”的方式运行。





内存中的每个进程在何时能获得处理机运行，何时又因提出某种资源请求而暂停，以及进程以怎样的速度向前推进，每道程序总共需多少时间才能完成，等等，这些都是不可预知的。由于各用户程序性能的不同，比如，有的侧重于计算而较少需要I/O，而有的程序其计算少而I/O多，这样，很可能是先进入内存的作业后完成，而后进入内存的作业先完成。或者说，进程是以人们不可预知的速度向前推进，此即进程的异步性(Asynchronism)。尽管如此，但只要在操作系统中配置有完善的进程同步机制，且运行环境相同，作业经多次运行都会获得完全相同的结果。因此，异步运行方式是允许的，而且是操作系统的一个重要特征。





1.4 操作系统的主要功能

1.4.1 处理机管理功能

在传统的多道程序系统中，处理机的分配和运行都是以进程为基本单位，因而对处理机的管理可归结为对进程的管理；在引入了线程的OS中，也包含对线程的管理。处理机管理的主要功能是创建和撤消进程(线程)，对诸进程(线程)的运行进行协调，实现进程(线程)之间的信息交换，以及按照一定的算法把处理机分配给进程(线程)。





1. 进程控制

在传统的多道程序环境下，要使作业运行，必须先为它创建一个或几个进程，并为之分配必要的资源。当进程运行结束时，立即撤消该进程，以便能及时回收该进程所占用的各类资源。进程控制的主要功能是为作业创建进程，撤消已结束的进程，以及控制进程在运行过程中的状态转换。在现代OS中，进程控制还应具有为一个进程创建若干个线程的功能和撤消(终止)已完成任务的线程的功能。





2. 进程同步

前已述及，进程是以异步方式运行的，并以人们不可预知的速度向前推进。为使多个进程能有条不紊地运行，系统中必须设置进程同步机制。进程同步的主要任务是为多个进程(含线程)的运行进行协调。有两种协调方式：

(1) 进程互斥方式。这是指诸进程(线程)在对临界资源进行访问时，应采用互斥方式；

(2) 进程同步方式。这是指在相互合作去完成共同任务的诸进程(线程)间，由同步机构对它们的执行次序加以协调。





3. 进程通信

在多道程序环境下，为了加速应用程序的运行，应在系统中建立多个进程，并且再为一个进程建立若干个线程，由这些进程(线程)相互合作去完成一个共同的任务。而在这些进程(线程)之间，又往往需要交换信息。例如，有三个相互合作的进程，它们是输入进程、计算进程和打印进程。输入进程负责将所输入的数据传送给计算进程；计算进程利用输入数据进行计算，并把计算结果传送给打印进程；最后，由打印进程把计算结果打印出来。进程通信的任务就是用来实现在相互合作的进程之间的信息交换。





4. 调度

在后备队列上等待的每个作业都需经过调度才能执行。在传统的操作系统中，包括作业调度和进程调度两步。

(1) 作业调度。作业调度的基本任务是从后备队列中按照一定的算法，选择出若干个作业，为它们分配运行所需的资源(首先是分配内存)。在将它们调入内存后，便分别为它们建立进程，使它们都成为可能获得处理机的就绪进程，并按照一定的算法将它们插入就绪队列。





第一章 操作系统引论



(2) 进程调度。进程调度的任务是从进程的就绪队列中，按照一定的算法选出一个进程，把处理机分配给它，并为其设置运行现场，使进程投入执行。值得提出的是，在多线程OS中，通常是把线程作为独立运行和分配处理机的基本单位，为此，须把就绪线程排成一个队列，每次调度时，是从就绪线程队列中选出一个线程，把处理机分配给它。





1.4.2 存储器管理功能

1. 内存分配

内存分配的主要任务是为每道程序分配内存空间，使它们“各得其所”；提高存储器的利用率，以减少不可用的内存空间；允许正在运行的程序申请附加的内存空间，以适应程序和数据动态增长的需要。





OS在实现内存分配时，可采取静态和动态两种方式。在静态分配方式中，每个作业的内存空间是在作业装入时确定的；在作业装入后的整个运行期间，不允许该作业再申请新的内存空间，也不允许作业在内存中“移动”。在动态分配方式中，每个作业所要求的基本内存空间也是在装入时确定的，但允许作业在运行过程中继续申请新的附加内存空间，以适应程序和数据的动态增长，也允许作业在内存中“移动”。





为了实现内存分配，在内存分配的机制中应具有这样的结构和功能：

(1) 内存分配数据结构。该结构用于记录内存空间的使用情况，作为内存分配的依据；

(2) 内存分配功能。系统按照一定的内存分配算法为用户程序分配内存空间；

(3) 内存回收功能。系统对于用户不再需要的内存，通过用户的释放请求去完成系统的回收功能。





2. 内存保护

内存保护的主要任务是确保每道用户程序都只在自己的内存空间内运行，彼此互不干扰；绝不允许用户程序访问操作系统的程序和数据；也不允许用户程序转移到非共享的其它用户程序中去执行。

为了确保每道程序都只在自己的内存区中运行，必须设置内存保护机制。一种比较简单的内存保护机制是设置两个界限寄存器，分别用于存放正在执行程序的上界和下界。系统须对每条指令所要访问的地址进行检查，如果发生越界，便发出越界中断请求，以停止该程序的执行。如果这种检查完全用软件实现，则每执行一条指令，便须增加若干条指令去进行越界检查，这将显著降低程序的运行速度。因此，越界检查都由硬件实现。当然，对发生越界后的处理，还须与软件配合来完成。





3. 地址映射

一个应用程序(源程序)经编译后,通常会形成若干个目标程序;这些目标程序再经过链接便形成了可装入程序。这些程序的地址都是从“0”开始的,程序中的其它地址都是相对于起始地址计算的。由这些地址所形成的地址范围称为“地址空间”,其中的地址称为“逻辑地址”或“相对地址”。此外,由内存中的一系列单元所限定的地址范围称为“内存空间”,其中的地址称为“物理地址”。

在多道程序环境下,每道程序不可能都从“0”地址开始装入(内存),这就致使地址空间内的逻辑地址和内存空间中的物理地址不相一致。为使程序能正确运行,存储器管理必须提供地址映射功能,以将地址空间中的逻辑地址转换为内存空间中与之对应的物理地址。该功能应在硬件的支持下完成。





4. 内存扩充

存储器管理中的内存扩充任务并非是去扩大物理内存的容量，而是借助于虚拟存储技术，从逻辑上去扩充内存容量，使用户所感觉到的内存容量比实际内存容量大得多，以便让更多的用户程序并发运行。这样，既满足了用户的需要，又改善了系统的性能。为此，只需增加少量的硬件。为了能在逻辑上扩充内存，系统必须具有内存扩充机制，用于实现下述各功能：

(1) 请求调入功能。允许在装入一部分用户程序和数据的情况下，便能启动该程序运行。在程序运行过程中，若发现要继续运行时所需的程序和数据尚未装入内存，可向OS发出请求，由OS从磁盘中将所需部分调入内存，以便继续运行。





第一章 操作系统引论



(2) 置换功能。若发现在内存中已无足够的空间来装入需要调入的程序和数据时，系统应能将内存中的一部分暂时不用的程序和数据调至盘上，以腾出内存空间，然后再将所需调入的部分装入内存。





1.4.3 设备管理功能

1. 缓冲管理

CPU运行的高速性和I/O低速性间的矛盾自计算机诞生时起便已存在了。而随着CPU速度迅速提高，使得此矛盾更为突出，严重降低了CPU的利用率。如果在I/O设备和CPU之间引入缓冲，则可有效地缓和CPU与I/O设备速度不匹配的矛盾，提高CPU的利用率，进而提高系统吞吐量。因此，在现代计算机系统中，都无一例外地在内存中设置了缓冲区，而且还可通过增加缓冲区容量的方法来改善系统的性能。





2. 设备分配

设备分配的基本任务是根据用户进程的I/O请求、系统的现有资源情况以及按照某种设备的分配策略，为之分配其所需的设备。如果在I/O设备和CPU之间还存在着设备控制器和I/O通道时，还须为分配出去的设备分配相应的控制器和通道。

为了实现设备分配，系统中应设置设备控制表、控制器控制表等数据结构，用于记录设备及控制器的标识符和状态。根据这些表格可以了解指定设备当前是否可用，是否忙碌，以供进行设备分配时参考。在进行设备分配时，应针对不同的设备类型而采用不同的设备分配方式。对于独占设备(临界资源)的分配，还应考虑到该设备被分配出去后系统是否安全。在设备使用完后，应立即由系统回收。





3. 设备处理

设备处理程序又称为设备驱动程序。其基本任务是用于实现CPU和设备控制器之间的通信，即由CPU向设备控制器发出I/O命令，要求它完成指定的I/O操作；反之，由CPU接收从控制器发来的中断请求，并给予迅速的响应和相应的处理。

处理过程是：设备处理程序首先检查I/O请求的合法性，了解设备状态是否是空闲的，了解有关的传递参数及设置设备的工作方式。然后，便向设备控制器发出I/O命令，启动I/O设备去完成指定的I/O操作。设备驱动程序还应能及时响应由控制器发来的中断请求，并根据该中断请求的类型，调用相应的中断处理程序进行处理。对于设置了通道的计算机系统，设备处理程序还应能根据用户的I/O请求，自动地构成通道程序。





1.4.4 文件管理功能

1. 文件存储空间的管理

为了方便用户的使用，对于一些当前需要使用的系统文件 and 用户文件，都必须放在可随机存取的磁盘上。在多用户环境下，若由用户自己对文件的存储进行管理，不仅非常困难，而且也必然是十分低效的。因而，需要由文件系统对诸多文件及文件的存储空间实施统一的管理。其主要任务是为每个文件分配必要的外存空间，提高外存的利用率，并能有助于提高文件系统的存、取速度。





为此，系统应设置相应的数据结构，用于记录文件存储空间的使用情况，以供分配存储空间时参考；系统还应具有对存储空间进行分配和回收的功能。为了提高存储空间的利用率，对存储空间的分配，通常是采用离散分配方式，以减少外存零头，并以盘块为基本分配单位。盘块的大小通常为1~8 KB。





2. 目录管理

为了使用户能方便地在外存上找到自己所需的文件，通常由系统为每个文件建立一个目录项。目录项包括文件名、文件属性、文件在磁盘上的物理位置等。由若干个目录项又可构成一个目录文件。目录管理的主要任务是为每个文件建立其目录项，并对众多的目录项加以有效的组织，以实现方便的按名存取，即用户只须提供文件名便可对该文件进行存取。其次，目录管理还应能实现文件共享，这样，只须在外存上保留一份该共享文件的副本。此外，还应能提供快速的目录查询手段，以提高对文件的检索速度。





3. 文件的读/写管理和保护

(1) 文件的读/写管理。该功能是根据用户的请求，从外存中读取数据，或将数据写入外存。在进行文件读(写)时，系统先根据用户给出的文件名去检索文件目录，从中获得文件在外存中的位置。然后，利用文件读(写)指针，对文件进行读(写)。一旦读(写)完成，便修改读(写)指针，为下一次读(写)做好准备。由于读和写操作不会同时进行，故可合用一个读/写指针。





(2) 文件保护。为了防止系统中的文件被非法窃取和破坏，在文件系统中必须提供有效的存取控制功能，以实现下述目标：

- ① 防止未经核准的用户存取文件；
- ② 防止冒名顶替存取文件；
- ③ 防止以不正确的方式使用文件。





1.4.5 操作系统与用户之间的接口

为了方便用户使用操作系统，OS又向用户提供了“用户与操作系统的接口”。该接口通常可分为两大类：

(1) 用户接口。它是提供给用户使用的接口，用户可通过该接口取得操作系统的服务；

(2) 程序接口。它是提供给程序员在编程时使用的接口，是用户程序取得操作系统服务的唯一途径。





1. 用户接口

为了便于用户直接或间接地控制自己的作业，操作系统向用户提供了命令接口。用户可通过该接口向作业发出命令以控制作业的运行。该接口又进一步分为联机用户接口和脱机用户接口。

(1) 联机用户接口。这是为联机用户提供的，它由一组键盘操作命令及命令解释程序所组成。当用户在终端或控制台上每键入一条命令后，系统便立即转入命令解释程序，对该命令加以解释并执行该命令。在完成指定功能后，控制又返回到终端或控制台上，等待用户键入下一条命令。这样，用户可通过先后键入不同命令的方式，来实现对作业的控制，直至作业完成。





(2) 脱机用户接口。该接口是为批处理作业的用户提供的，故也称为批处理用户接口。该接口由一组作业控制语言(JCL)组成。批处理作业的用户不能直接与自己的作业交互作用，只能委托系统代替用户对作业进行控制和干预。这里的作业控制语言(JCL)便是提供给批处理作业用户的、为实现所需功能而委托系统代为控制的一种语言。用户用JCL把需要对作业进行的控制和干预事先写在作业说明书上，然后将作业连同作业说明书一起提供给系统。当系统调度到该作业运行时，又调用命令解释程序，对作业说明书上的命令逐条地解释执行。如果作业在执行过程中出现异常现象，系统也将根据作业说明书上的指示进行干预。这样，作业一直在作业说明书的控制下运行，直至遇到作业结束语句时，系统才停止该作业的运行。





(3) 图形用户接口。用户虽然可以通过联机用户接口来取得OS的服务，但这时要求用户能熟记各种命令的名字和格式，并严格按照规定的格式输入命令。这既不方便又花时间，于是，另一种形式的联机用户接口——图形用户接口便应运而生。图形用户接口采用了图形化的操作界面，用非常容易识别的各种图标(Icon)来将系统的各项功能、各种应用程序和文件，直观、逼真地表示出来。用户可用鼠标或通过菜单和对话框来完成对应用程序和文件的操作。此时用户已完全不必像使用命令接口那样去记住命令名及格式，从而把用户从繁琐且单调的操作中解脱出来。





2. 程序接口

该接口是为用户程序在执行中访问系统资源而设置的，是用户程序取得操作系统服务的惟一途径。它是由一组系统调用组成，每一个系统调用都是一个能完成特定功能的子程序，每当应用程序要求OS提供某种服务(功能)时，便调用具有相应功能的系统调用。早期的系统调用都是用汇编语言提供的，只有在用汇编语言书写的程序中才能直接使用系统调用；但在高级语言以及C语言中，往往提供了与各系统调用一一对应的库函数，这样，应用程序便可通过调用对应的库函数来使用系统调用。但在近几年所推出的操作系统中，如UNIX、OS/2版本中，其系统调用本身已经采用C语言编写，并以函数形式提供，故在用C语言编制的程序中，可直接使用系统调用。





1.5 OS结构设计

1.5.1 传统的操作系统结构

1. 无结构操作系统

在早期开发操作系统时，设计者只是把注意力放在功能的实现和获得高的效率上，缺乏首尾一致的设计思想。此时的OS是为数众多的一组过程的集合，每个过程可以任意地相互调用其它过程，致使操作系统内部既复杂又混乱。因此，这种OS是无结构的，也有人把它称为整体系统结构。





2. 模块化结构OS

1) 模块化程序设计技术的基本概念

模块化程序设计技术是20世纪60年代出现的一种结构化程序设计技术。该技术是基于“分解”和“模块化”原则来控制大型软件的复杂度。为使OS具有较清晰的结构，OS不再是由众多的过程直接构成，而是将OS按其功能精心地划分为若干个具有一定独立性和大小的模块；每个模块具有某方面的管理功能，如进程管理模块、存储器管理模块、I/O设备管理模块等，并仔细地规定好各模块间的接口，使各模块之间能通过该接口实现交互。





然后，再进一步将各模块细分为若干个具有一定功能的子模块，如把进程管理模块又分为进程控制、进程同步等子模块，同样也要规定好各子模块之间的接口。若子模块较大时，可再进一步将它细分。我们把这种设计方法称为模块—接口法，由此构成的操作系统就是具有模块化结构的操作系统。图1-6示出了由模块、子模块等组成的模块化OS结构。





第一章 操作系统引论

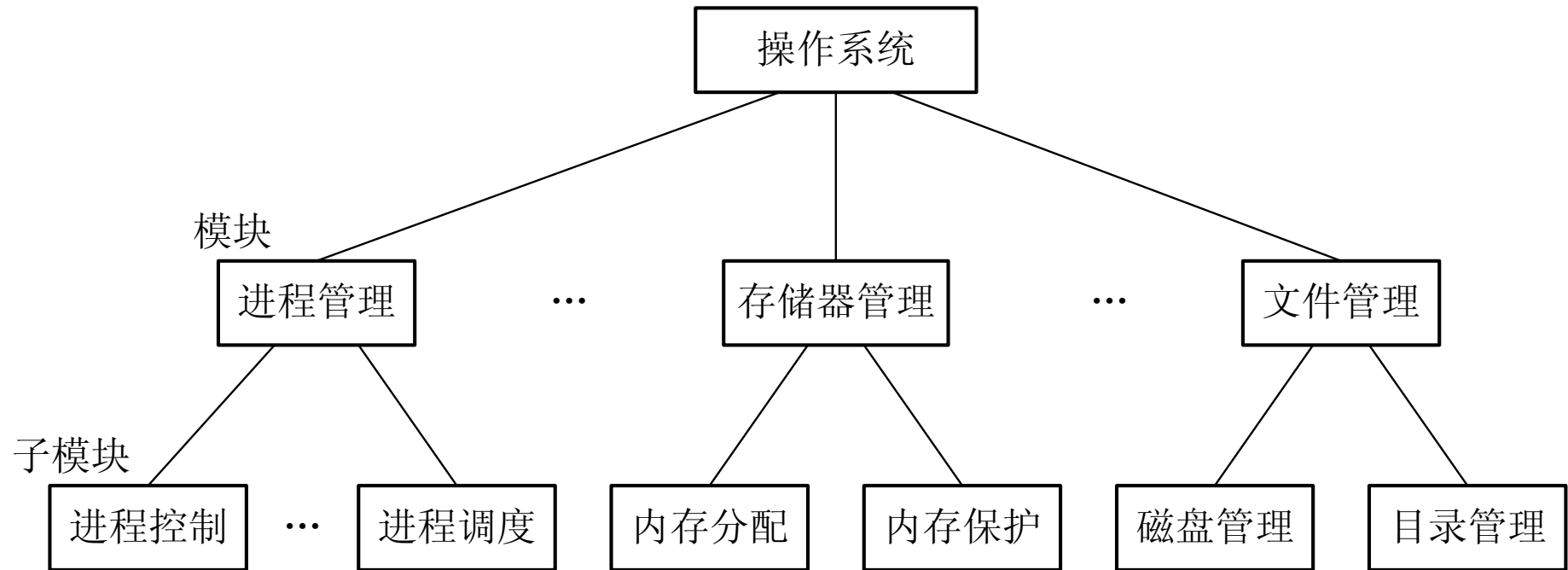


图1-6 模块化结构的操作系统





2) 模块的独立性

在模块—接口法设计方法中，关键问题是模块的划分和规定好模块之间的接口。如果我们在划分模块时，将模块划分得太小，虽然可以降低模块本身的复杂性，但会引起模块之间的联系过多，而会造成系统比较混乱；如果将模块划分得过大，又会增加模块内部的复杂性，使内部的联系增加。因此，在划分模块时，应在两者间进行权衡。





另外，在划分模块时，必须充分注意模块的独立性问题。因为模块的独立性越高，各模块间的交互就越少，系统的结构也就越清晰。衡量模块的独立性有以下两个标准：

(1) 内聚性，指模块内部各部分间联系的紧密程度。内聚性越高，模块的独立性越强。

(2) 耦合度，指模块间相互联系和相互影响的程度。显然，耦合度越低，模块的独立性越好。





第一章 操作系统引论



3) 模块接口法的优缺点

利用模块—接口法开发的OS，较之无结构OS具有以下明显的优点：

- (1) 提高OS设计的正确性、可理解性和可维护性；
- (2) 增强OS的适应性；
- (3) 加速OS的开发过程。





模块化结构设计仍存在下述问题：

(1) 在OS设计时，对各模块间的接口规定很难满足在模块完成后对接口的实际需求。

(2) 在OS设计阶段，设计者必须做出一系列的决策，每一个决定必须建立在上一个决定的基础上。但在模块化结构设计中，各模块的设计齐头并进，无法寻找到一个可靠的决定顺序，造成各种决定的“无序性”，这将使程序设计人员很难做到“设计中的每一步决定都是建立在可靠的基础上”，因此模块—接口法又被称为“无序模块法”。





3. 分层式结构OS

1) 分层式结构的基本概念

为了将模块—接口法中“决定顺序”的无序性变为有序性，引入了有序分层法。分层法的设计任务是，在目标系统 A_n 和裸机系统(又称宿主系统) A_0 之间，铺设若干个层次的软件 A_1 、 A_2 、 A_3 、...、 A_{n-1} ，使 A_n 通过 A_{n-1} 、 A_{n-2} 、...、 A_2 、 A_1 层，最终能在 A_0 上运行。在操作系统中，常采用自底向上法来铺设这些中间层。





第一章 操作系统引论



自底向上的分层设计的基本原则是：每一步设计都是建立在可靠的基础上。为此规定，每一层仅能使用其底层所提供的功能和服务，这样可使系统的调试和验证都变得更容易。例如，在调试第一层软件A1时，由于它使用的是一个完全确定的物理机器(宿主系统)所提供的功能，在对A1软件经过精心设计和几乎是穷尽无遗的测试后，可以认为A1是正确的，而且它与其所有的高层软件A2、...、An无关；同样在调试第二层软件A2时，它也只使用了软件A1和物理机器所提供的功能，而与其高层软件A3、...、An无关；如此一层一层地自底向上增添软件层，每一层都实现若干功能，最后总能构成一个能满足需要的OS。在用这种方法构成操作系统时，已将一个操作系统分为若干个层次，每层又由若干个模块组成，各层之间只存在着单向的依赖关系，即高层仅依赖于紧邻它的低层。





2) 分层结构的优缺点

分层结构的主要优点有：

(1) 易保证系统的正确性。自下而上的设计方式，使所有设计中的决定都是有序的，或者说是建立在较为可靠的基础上的，这样比较容易保证整个系统的正确性。

(2) 易扩充和易维护性。在系统中增加、修改或替换一个层次中的模块或整个层次，只要不改变相应层次间的接口，就不会影响其它层次，这必将使系统维护和扩充变得更加容易。





1.5.2 客户/服务器模式

客户/服务器 (Client/Server) 模式可简称为C/S模式。

1. 客户/服务器模式的组成

客户/服务器系统主要由客户机、服务器和网络系统三个部分组成。

(1) 客户机：通常在一个LAN网络上连接有多台网络工作站(简称客户机)，每台客户机都是一个自主计算机，具有一定的处理能力，客户进程在其上运行，平时它处理一些本地业务，也可发送一个消息给服务器，以请求某项服务。





(2) 服务器：通常是一台规模较大的机器，在其上驻留有网络文件系统或数据库系统等，它应能为网上所有的用户提供一种或多种服务。平时它一直处于工作状态，被动地等待来自客户机的请求，一旦检查到有客户提出服务请求，便去完成客户的请求，并将结果送回客户。这样，工作站中的用户进程与服务器进程便形成了客户/服务器关系。

(3) 网络系统：用于连接所有客户机和服务器，实现它们之间通信和网络资源共享的系统。





2. 客户/服务器之间的交互

在采用客户/服务器的系统中，通常是客户机和服务器共同完成对应用(程序)的处理。这时，在客户机和服务器之间就需要进行交互，即必须利用消息机制在这两者之间进行多次通信。一次完整的交互过程可分成以下四步：

(1) 客户发送请求消息。当客户机上的用户要请求服务器进行应用处理时，应输入相应的命令和有关参数。由客户机上的发送进程先把这些信息装配成请求消息，然后把它发往服务器；客户机上的接收进程则等待接收从服务器发回来的响应消息。





(2) 服务器接收消息。服务器中的接收进程平时处于等待状态，一旦有客户机发来请求，接收进程便被激活，并根据请求信息的内容，将之提供给服务器上的相应软件进行处理。

(3) 服务器回送消息。服务器的软件根据请求进行处理，在完成指定的处理后，把处理结果装配成一个响应消息，由服务器中的发送进程将之发往客户机。

(4) 客户机接收消息。客户机中的接收进程把收到的响应消息转交给客户机软件，再由后者做出适当处理后提交给发送该请求的客户。





3. 客户/服务器模式的优点

C/S模式之所以能成为当前分布式系统和网络环境下软件的主要工作模式，是由于该模式具有传统集中模式所无法比拟的一系列优点。

(1) 数据的分布处理和存储。由于客户机具有相当强的处理和存储能力，可进行本地处理和数据的分布存储，从而摆脱了由于把一切数据都存放在主机中而造成的既不可靠又容易产生瓶颈现象的困难局面。





(2) 便于集中管理。尽管C/S模式具有分布处理功能，但公司(单位)中的有关全局的重要信息、机密资料、重要设备以及网络管理等，仍可采取集中管理方式。这样可较好地保障系统的“可靠”与“安全”。

(3) 灵活性和可扩充性。C/S模式非常灵活，极易扩充。理论上，客户机和服务器的数量不受限制。其灵活性还表现在可以配置多种类型的客户机和服务器。

(4) 易于改编应用软件。在客户/服务器模式中，对于客户机程序的修改和增删，比传统集中模式要容易得多，必要时也允许由客户进行修改。





1.5.3 面向对象的程序设计

1. 面向对象技术的基本概念

面向对象技术是20世纪80年代初提出并很快流行起来的。该技术是基于“抽象”和“隐蔽”原则来控制大型软件的复杂度的。所谓对象，是指在现实世界中具有相同属性、服从相同规则的一系列事物(事物可以是一个物理实体、一个概念或一个软件模块等)的抽象，而把其中的具体事物称为对象的实例。如果在OS中的各类实体如进程、线程、消息、存储器和文件等，都使用了对象这一概念，相应地，便有了进程对象、线程对象、存储器对象和文件对象等。





1) 对象

在面向对象的技术中，是利用被封装的数据结构(变量)和一组对它进行操作的过程(方法)，来表示系统中的某个对象的，如图1-7所示。对象中的变量(数据)也称为属性，它可以是单个标量或一张表。面向对象中的方法是用于执行某种功能的过程，它可以改变对象的状态，更新对象中的某些数据值或作用于对象所要访问的外部资源。如果把一个文件作为一个对象(见图1-8)，该对象的变量便是文件类型、文件大小、文件的创建者等。对象中的方法包含对文件的操作，如创建文件、打开文件、读文件、写文件、关闭文件等。





图1-7 一个对象的示意图





2) 对象类

在实践中，有许多对象可能表示的是同一类事物，每个对象具有自己的变量集合，而它们所具有的方法是相同的。如果为每一个相似的对象都定义一组变量和方法，显然是低效的，由此产生了“对象类”的概念，利用“对象类”来定义一组大体相似的对象。一个类同样定义了一组变量和针对该变量的一组方法，用它们来描述一组对象的共同属性和行为。类是在对象上的抽象，对象则是类的实例。对象类中所定义的变量在实例中均有具体的值。





例如，我们将文件设计成一个类，类的变量同样是文件类型、文件大小和文件的创建者等。类中的方法是文件的创建、打开、读写、关闭等。图1-8示出了一个文件类，在类的变量中，没有具体数值，一旦被赋予了具体数值就成了文件A对象。对象类的概念非常有用，因为它极大地提高了创建多个相似对象的效率。





第一章 操作系统引论

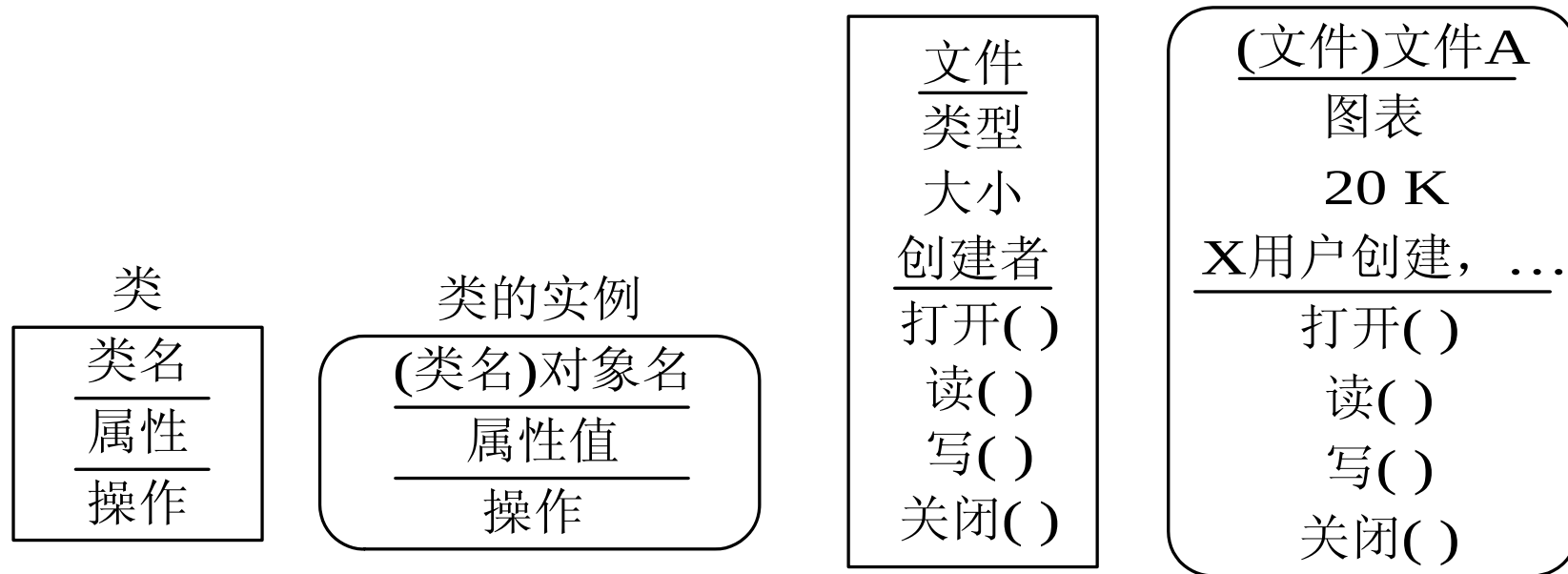


图1-8 类和对象的关系





3) 继承

在面向对象的技术中，可以根据已有类来定义一个新的类，新类被称为子类(B)，原来的类被称为父类(A)，见图1-9。继承是父类和子类之间共享变量和方法的机制，该机制规定，子类自动继承父类中定义的变量和方法，并允许子类再增加新的内容。继承特性可使定义子类变得更为容易。一个父类可以定义多个子类，它们分别是父类的某种特例，父类描述了这些子类的公共变量和方法。类似地，这些子类又可以定义自己的子类，通过此途经可以生成一个继承的层次。另外，也允许一个子类有两个父类或多个父类，它可以从多个父类获得继承，此时称为“多重继承”。



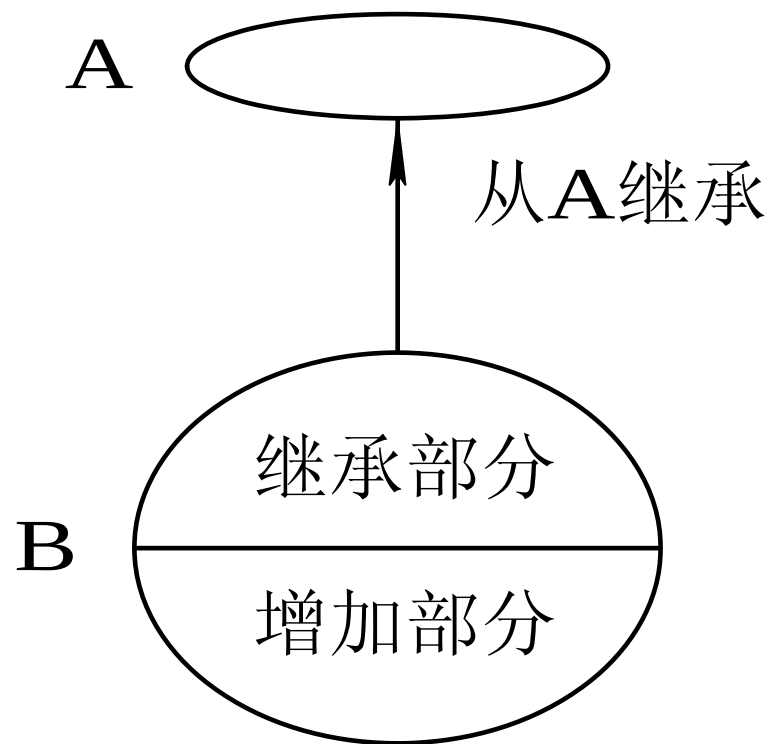


图1-9 类的继承关系





2. 面向对象技术的优点

在设计操作系统时，将计算机中的实体作为对象来处理，可带来如下好处：

(1) 通过“重用”提高产品质量和生产率。

在面向对象技术中可通过“重用”以前项目中经过精心测试的对象，或由其他人编写、测试和维护的对象类，来构建新的系统，这不仅可大大降低开发成本，而且能获得更好的系统质量。





(2) 使系统具有更好的易修改性和易扩展性。

通过封装，可隐蔽对象中的变量和方法，因而当改变对象中的变量和方法时，不会影响到其它部分，从而可方便地修改老的对象类。另外，继承是面向对象技术的重要特性，在创建一个新对象类时，通过利用继承特性，可显著地减少开发的时空开销，使系统具有更好的易扩展性和灵活性。





(3) 更易于保证系统的“正确性”和“可靠性”。

对象是构成操作系统的基本单元，由于可以独立地对它进行测试，易于保证每个对象的正确性和可靠性，因此也就比较容易保证整个系统的正确性和可靠性。此外，封装对对象类中的信息进行了隐蔽，这样又可有效地防止未经授权者的访问和用户不正确的使用，有助于构建更为安全的系统





1.5.4 微内核OS结构

1. 微内核操作系统的基本概念

为了提高操作系统的“正确性”、“灵活性”、“易维护性”和“可扩充性”，在进行现代操作系统结构设计时，即使在单处理机环境下，大多也采用基于客户/服务器模式的微内核结构，将操作系统划分为两大部分：微内核和多个服务器。至于什么是微内核操作系统结构，现在尚无一致公认的定义，但我们可以从下面四个方面，对微内核结构的操作系统进行描述。





1) 足够小的内核

在微内核操作系统中，内核是指精心设计的、能实现现代OS最基本的核心功能的部分。微内核并非是一个完整的OS，而只是操作系统中最基本的部分，它通常用于：① 实现与硬件紧密相关的处理；② 实现一些较基本的功能；③ 负责客户和服务端之间的通信。它们只是为构建通用OS提供一个重要基础，这样就可以确保把操作系统内核做得很小。





2) 基于客户/服务器模式

由于客户/服务器模式具有非常多的优点，故在单机微内核操作系统中几乎无一例外地都采用客户/服务器模式，将操作系统中最基本的部分放入内核中，而把操作系统的绝大部分功能都放在微内核外面的一组服务器(进程)中实现。例如用于提供对进程(线程)进行管理的进程(线程)服务器，提供虚拟存储器管理功能的虚拟存储器服务器，提供I/O设备管理的I/O设备管理服务器等，它们都是被作为进程来实现的，运行在用户态，客户与服务器之间是借助微内核提供的消息传递机制来实现信息交互的。图1-10示出了在单机环境下的客户/服务器模式。



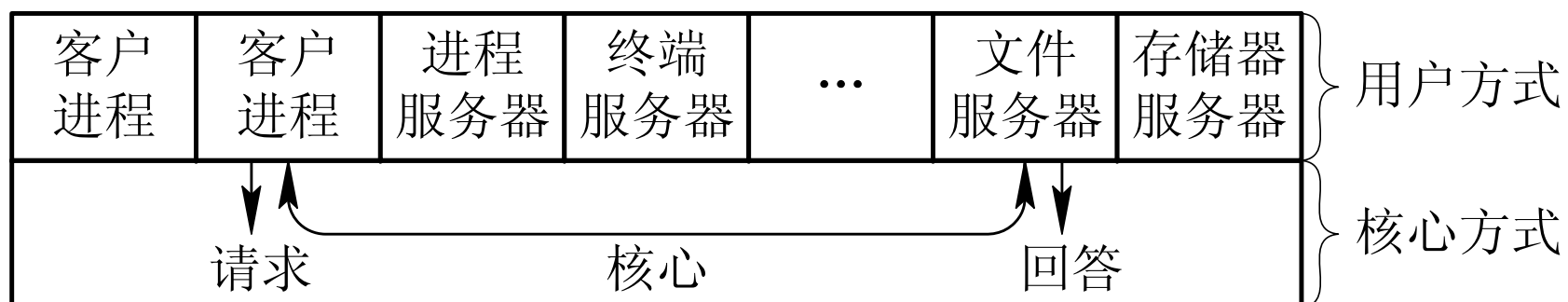


图1-10 在单机环境下的客户/服务器模式





3) 应用“机制与策略分离”原理

在现代操作系统的结构设计中，经常利用“机制与策略分离”的原理来构造OS结构。所谓机制，是指实现某一功能的具体执行机构。而策略，则是在机制的基础上，借助于某些参数和算法来实现该功能的优化，或达到不同的功能目标。通常，机制处于一个系统的基层，而策略则处于系统的高层。在传统的OS中，将机制放在OS的内核的较低层，把策略放在内核的较高层次中。而在微内核操作系统中，通常将机制放在OS的微内核中。正因为如此，才有可能将内核做得很小。





4) 采用面向对象技术

操作系统是一个极其复杂的大型软件系统，我们不仅可以通过结构设计来分解操作系统的复杂度，还可以基于面向对象技术中的“抽象”和“隐蔽”原则控制系统的复杂性，再进一步利用“对象”、“封装”和“继承”等概念来确保操作系统的“正确性”、“可靠性”、“易修改性”、“易扩展性”等，并提高操作系统的设计速度。正因为面向对象技术能带来如此多的好处，故面向对象技术被广泛应用于现代操作系统的设计中。





2. 微内核的基本功能

1) 进程(线程)管理

大多数的微内核OS，对于进程管理功能的实现，都采用“机制与策略分离”的原理。例如，为实现进程(线程)调度功能，须在进程管理中设置一个或多个进程(线程)优先级队列；能将指定优先级进程(线程)从所在队列中取出，并将其投入执行。由于这一部分属于调度功能的机制部分，应将它放入微内核中。应如何确定每类用户(进程)的优先级，以及应如何修改它们的优先级等，都属于策略问题，可将它们放入微内核外的进程(线程)管理服务器中。





2) 低级存储器管理

通常在微内核中，只配置最基本的低级存储器管理机制。如用于实现将用户空间的逻辑地址变换为内存空间的物理地址的页表机制和地址变换机制，这一部分是依赖于机器的，因此放入微内核。而实现虚拟存储器管理的策略，则包含应采取何种页面置换算法，采用何种内存分配与回收策略等，应将这部分放在微内核外的存储器管理服务器中去实现。





3) 中断和陷入处理

大多数微内核操作系统都是将与硬件紧密相关的一小部分放入微内核中处理。此时微内核的主要功能，是捕获所发生的中断和陷入事件，并进行相应的前期处理。如进行中断现场保护，识别中断和陷入的类型，然后将有关事件的信息转换成消息后，把它发送给相关的服务器。由服务器根据中断或陷入的类型，调用相应的处理程序来进行后期处理。

在微内核OS中是将进程管理、存储器管理以及I/O管理这些功能一分为二，属于机制的很小一部分放入微内核中，另外绝大部分放在微内核外的各种服务器中来实现。事实上，其中大多数服务器都比微内核大。这进一步说明了为什么能在采用客户/服务器模式后，还能把微内核做得很小的原因。





3. 微内核操作系统的优点

1) 提高了系统的可扩展性

由于微内核OS的许多功能是由相对独立的服务器软件来实现的，当开发了新的硬件和软件时，微内核OS只须在相应的服务器中增加新的功能，或再增加一个专门的服务器。与此同时，也必然改善系统的灵活性，不仅可在操作系统中增加新的功能，还可修改原有功能，以及删除已过时的功能，以形成一个更为精干有效的操作系统。





2) 增强了系统的可靠性

这一方面是由于微内核是出于精心设计和严格测试的，容易保证其正确性；另一方面是它提供了规范而精简的应用程序接口(API)，为微内核外部的程序编制高质量的代码创造了条件。此外，由于所有服务器都是运行在用户态，服务器与服务器之间采用的是消息传递通信机制，因此，当某个服务器出现错误时，不会影响内核，也不会影响其它服务器。





3) 可移植性

随着硬件的快速发展，出现了各种各样的硬件平台，作为一个好的操作系统，必须具备可移植性，使其能较容易地运行在不同的计算机硬件平台上。在微内核结构的操作系统中，所有与特定CPU和I/O设备硬件有关的代码，均放在内核和内核下面的硬件隐藏层中，而操作系统其它绝大部分(即各种服务器)均与硬件平台无关，因而，把操作系统移植到另一个计算机硬件平台上所需作的修改是比较小的。





4) 提供了对分布式系统的支持

由于在微内核OS中，客户和服务端之间以及服务端和服务端之间的通信，是采用消息传递通信机制进行的，致使微内核OS能很好地支持分布式系统和网络系统。事实上，只要在分布式系统中赋予所有进程和服务端唯一的标识符，在微内核中再配置一张系统映射表(即进程和服务端的标识符与它们所驻留的机器之间的对应表)，在进行客户与服务端通信时，只需在所发送的消息中标上发送进程和接收进程的标识符，微内核便可利用系统映射表，将消息发往目标，而无论目标是驻留在哪台机器上。





5) 融入了面向对象技术

在设计微内核OS时，采用了面向对象的技术，其中的“封装”，“继承”，“对象类”和“多态性”，以及在对象之间采用消息传递机制等，都十分有利于提高系统的“正确性”、“可靠性”、“易修改性”、“易扩展性”等，而且还能显著地减少开发系统所付出的开销。





4. 微内核操作系统存在的问题

应当指出，在微内核OS中，由于采用了非常小的内核，以及客户/服务器模式和消息传递机制，这些虽给微内核OS带来了许多优点，但由此也使微内核OS存在着潜在的缺点。其中最主要的是，较之早期OS，微内核OS的运行效率有所降低。



第一章 操作系统引论



效率降低的最主要的原因是，在完成一次客户对OS提出的服务请求时，需要利用消息实现多次交互和进行用户/内核模式及上下文的多次切换。然而，在早期的OS中，用户进程在请求取得OS服务时，一般只需进行两次上下文的切换：一次是在执行系统调用后，由用户态转向系统态时；另一次是在系统完成用户请求的服务后，由系统态返回用户态时。在微内核OS中，由于客户和服务及服务器和服务之间的通信，都需通过微内核，致使同样的服务请求至少需要进行四次上下文切换。第一次是发生在客户发送请求消息给内核，以请求取得某服务器特定的服务时；第二次是发生在由内核把客户的请求消息发往服务器时；第三次是当服务器完成客户请求后，把响应消息发送到内核时；第四次是在内核将响应消息发送给客户时。



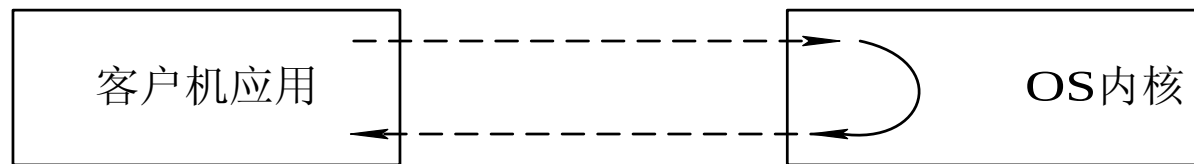


实际情况是往往还会引起更多的上下文切换。例如，当某个服务器自身尚无能力完成客户请求，而需要其它服务器的帮助时，如图1-11中所示，其中的文件服务器还需要磁盘服务器的帮助，这时就需要进行八次上下文的切换。



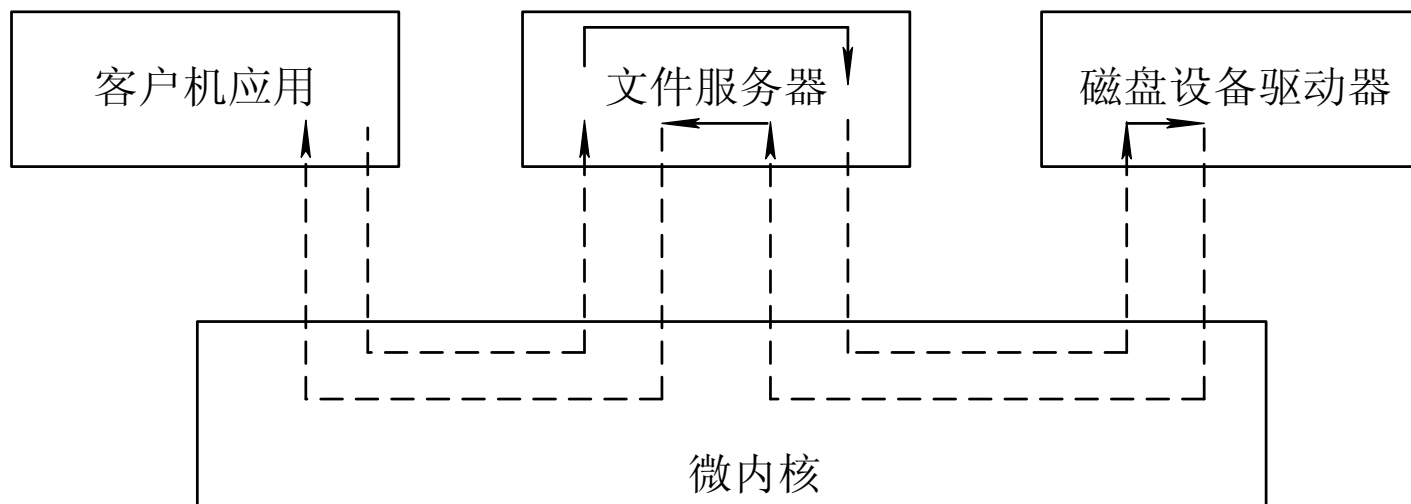


第一章 操作系统引论



返回

(a) 在整体式内核文件操作中的上下文切换



(b) 在微内核中等价操作的上下文切换

图1-11 在传统OS和微内核OS中的上下文切换





第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程的同步问题

2.5 进程通信

2.6 线程





2.1 进程的基本概念

2.1.1 程序的顺序执行及其特征

1. 程序的顺序执行

通常可以把一个应用程序分成若干个**程序段**，在各程序段之间，必须按照某种先后次序顺序执行，仅当前一操作(程序段)执行完后，才能执行后继操作。我们用**结点(Node)**代表各程序段的操作(在图2-1中用圆圈表示)，对一个程序段中的多条语句来说，也有一个执行顺序问题，例如对于下述三条语句的程序段：





第二章 进程管理

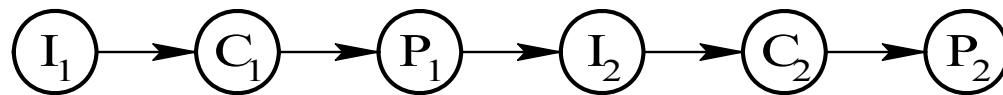


S1: $a := x + y;$

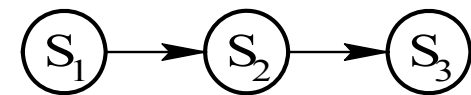
S2: $b := a - 5;$

S3: $c := b + 1;$

其中，S2必须在S1之后才能执行，语句S3只能在b被赋值后才能执行。因此，这三条语句应按图2-1(b)所示的顺序执行。



(a) 程序的顺序执行



(b) 三条语句的顺序执行

图 2-1 程序的顺序执行





2. 程序顺序执行时的特征

(1) 顺序性：处理机的操作**严格按照程序所规定的**顺序执行，即每一操作必须在上一个操作结束之后开始。

(2) 封闭性：程序是在封闭的环境下执行的，即程序运行时**独占全机资源**，资源的状态(除初始状态外)只有本程序才能改变它。开始执行，其执行结果**不受外界因素影响**。

(3) 可再现性：只要程序执行时的**环境和初始条件相同**，当程序重复执行时，不论它是从头到尾不停顿地执行，还是“**停停走走**”地执行，都将获得相同的结果。

顺序执行的特性，为程序员检测和校正程序错误带来很大的方便。





2.1.2 前趋图

前趋图(Precedence Graph)是一个有向无循环图, 记为 DAG(Directed Acyclic Graph), 用于描述进程之间执行的前后关系。图中的每个结点可用于描述一个程序段或进程, 乃至一条语句; 结点间的有向边则用于表示两个结点之间存在的偏序(Partial Order, 亦称偏序关系)或前趋关系(Precedence Relation)“ $\rightarrow$ ”。





第二章 进程管理



$\rightarrow = \{(P_i, P_j) | P_i \text{ must complete before } P_j \text{ may start}\}$, 如果 $(P_i, P_j) \in \rightarrow$, 可写成 $P_i \rightarrow P_j$, 称 P_i 是 P_j 的**直接前趋**, 而称 P_j 是 P_i 的**直接后继**。在前趋图中, 把**没有前趋**的结点称为**初始结点(Initial Node)**, 把没有后继的结点称为**终止结点(Final Node)**。此外, 每个结点还具有一个**重量(Weight)**, 用于表示该结点所含有的程序量或**结点的执行时间**。在图2-1(a)和2-1(b)中分别存在着这样的前趋关系:

$$I_i \rightarrow C_i \rightarrow P_i$$

和

$$S_1 \rightarrow S_2 \rightarrow S_3$$





第二章 进程管理



对于图2-2(a)所示的前趋图，存在下述前趋关系：

$$P_1 \rightarrow P_2, P_1 \rightarrow P_3, P_1 \rightarrow P_4, P_2 \rightarrow P_5, P_3 \rightarrow P_5, P_4 \rightarrow P_6, \\ P_4 \rightarrow P_7, P_5 \rightarrow P_8, P_6 \rightarrow P_8, P_7 \rightarrow P_9, P_8 \rightarrow P_9$$

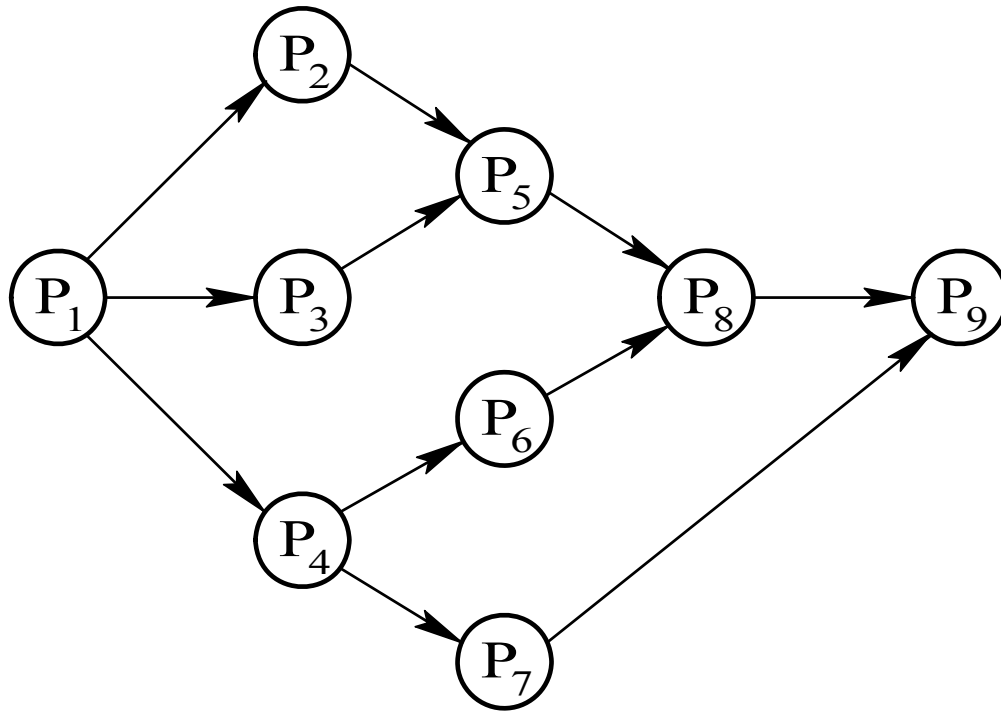
或表示为：

$$P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\} \\ \rightarrow = \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5), (P_4, P_6), \\ (P_4, P_7), (P_5, P_8), (P_6, P_8), (P_7, P_9), (P_8, P_9)\}$$

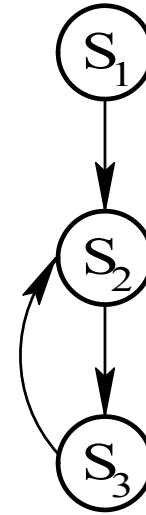
应当注意，前趋图中绝不存在循环，但在图2-2(b)中却有着下述的前趋关系：

$$S_2 \rightarrow S_3, S_3 \rightarrow S_2$$





(a) 具有九个结点的前趋图



(b) 具有循环的图

图 2-2 前趋图





2.1.3 程序的并发执行及其特征

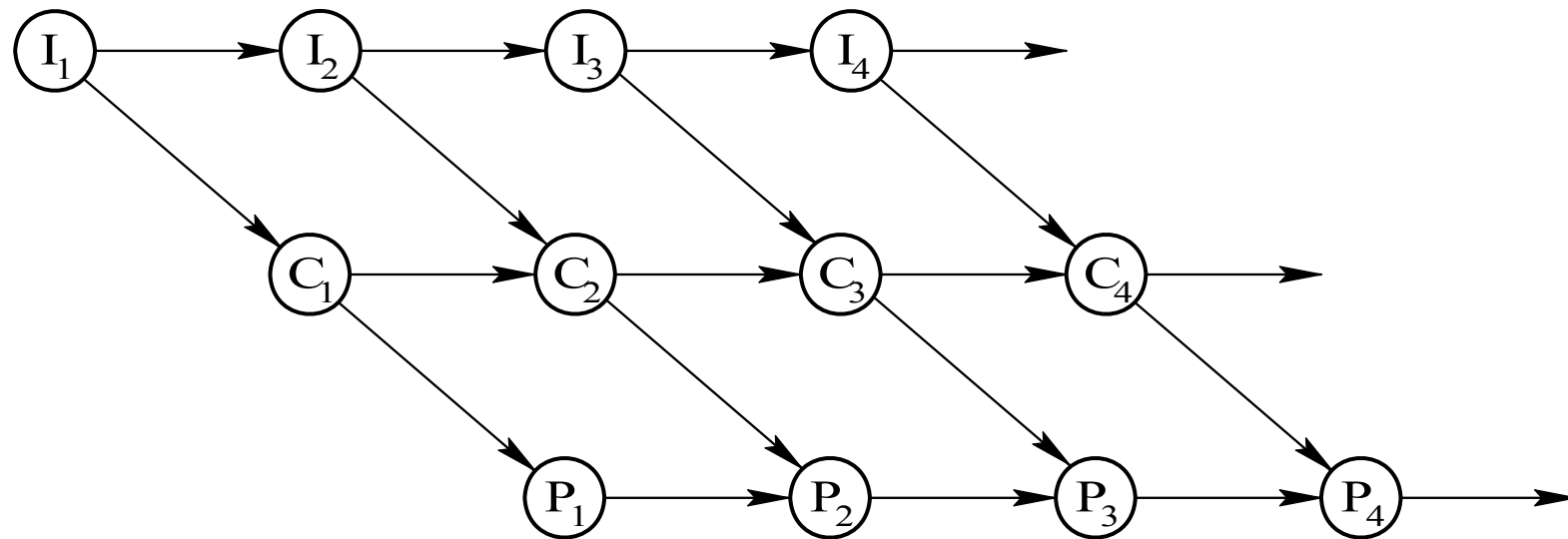
1. 程序的并发执行

在图2-1中的输入程序，存在着 $I_i \rightarrow C_i \rightarrow P_i$ 这样的前趋关系，必须顺序执行，但并不存在 $P_i \rightarrow I_{i+1}$ 的循环关系，因而在对一批程序进行处理时，可使它们并发执行。例如，输入程序在输入第一个程序后，在计算程序对该程序进行计算时，可由输入程序再输入第二个程序，从而使第一个程序的计算操作可与第二个程序的输入操作并发执行。一般来说，输入程序在输入第 $i+1$ 个程序时，计算程序可能正在对第 i 个程序进行计算，而打印程序正在打印第 $i-1$ 个程序的计算结果。图2-3示出了输入、计算和打印这三个程序对一批作业进行处理的情况。





第二章 进程管理



I3 C2 P1 之间是同时进行的并发执行

图2-3 并发执行时的前趋图





第二章 进程管理



在该例中存在下述前趋关系：

$$I_i \rightarrow C_i, I_i \rightarrow I_{i+1}, C_i \rightarrow P_i, C_i \rightarrow C_{i+1}, P_i \rightarrow P_{i+1}$$

而 I_{i+1} 和 C_i 及 P_{i-1} 是重迭的，亦即在 P_{i-1} 和 C_i 以及 I_{i+1} 之间，可以并发执行。对于具有下述四条语句的程序段：

S1: $a := x + 2$

S2: $b := y + 4$

S3: $c := a + b$

S4: $d := c + b$





第二章 进程管理

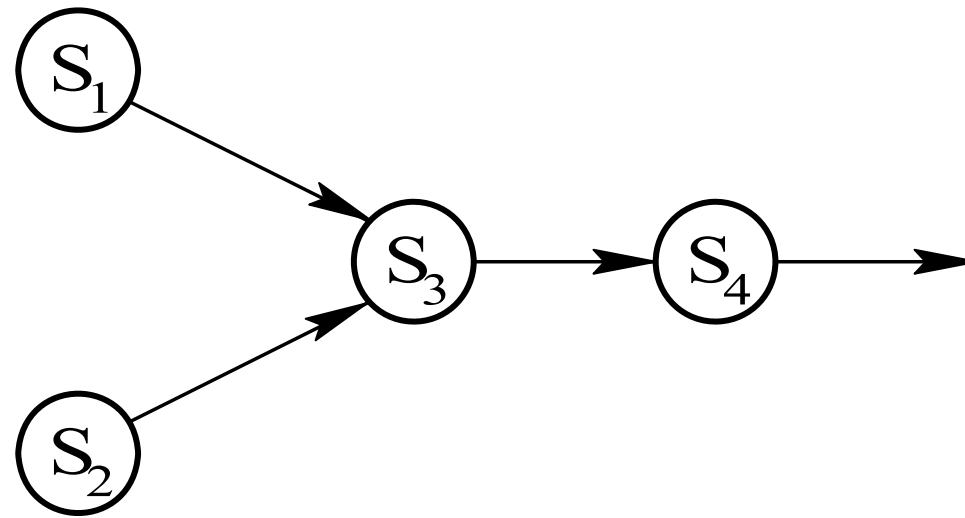


图 2-4 四条语句的前趋关系





2. 程序并发执行时的特征

1) 间断性

程序在并发执行时，由于它们共享系统资源，以及为完成同一项任务而相互合作，致使在这些并发执行的程序之间，形成了相互制约的关系。例如，图2-3 中的I、C和P是三个相互合作的程序，当计算程序完成 C_{i-1} 的计算后，如果输入程序I尚未完成 I_i 的处理，则计算程序就无法进行 C_i 的处理，致使计算程序必须暂停运行。又如，当打印程序完成 P_i 的打印后，若计算程序尚未完成 C_{i+1} 的计算，则打印程序就无法对 C_{i+1} 的计算结果进行打印。一旦使程序暂停的因素消失后(如 I_i 已处理完成)，计算程序便可恢复执行对 C_i 的处理。简而言之，相互制约将导致并发程序具有“执行—暂停—执行”这种间断性的活动规律。





第二章 进程管理



2) 失去封闭性

程序在并发执行时，是多个程序共享系统中的各种资源，因而这些资源的状态将由多个程序来改变，致使程序的运行失去了封闭性。这样，某程序在执行时，必然会受到其它程序的影响。例如，当处理机这一资源已被某个程序占有时，另一程序必须等待。





3) 不可再现性

程序在并发执行时，由于失去了封闭性，也将导致其失去可再现性。例如，有两个循环程序A和B，它们共享一个变量N。程序A每执行一次时，都要做 $N:=N+1$ 操作；程序B每执行一次时，都要执行Print(N)操作，然后再将N置成“0”。程序A和B以不同的速度运行。这样，可能出现下述三种情况(假定某时刻变量N的值为n)。





第二章 进程管理



(1) $N := N + 1$ 在 $\text{Print}(N)$ 和 $N := 0$ 之前, 此时得到的 N 值分别为 $n+1$, $n+1$, 0 。

(2) $N := N + 1$ 在 $\text{Print}(N)$ 和 $N := 0$ 之后, 此时得到的 N 值分别为 n , 0 , 1 。

(3) $N := N + 1$ 在 $\text{Print}(N)$ 和 $N := 0$ 之间, 此时得到的 N 值分别为 n , $n+1$, 0 。

上述情况说明, 程序在并发执行时, 由于失去了封闭性, 其计算结果已与并发程序的执行速度有关, 从而使程序的执行失去了可再现性, 亦即, 程序经过多次执行后, 虽然它们执行时的环境和初始条件相同, 但得到的结果却各不相同。





2.1.4 进程的特征与状态

1. 进程的特征和定义

在多道程序环境下，程序的执行属于并发执行，将失去其封闭性，并具有间断性及不可再现性的特征。这决定了通常的程序是不能参与并发执行的，因为程序执行的结果是不可再现的。这样，程序的运行也就失去了意义。为使程序能并发执行，且为了对并发执行的程序加以描述和控制，人们引入了“进程”的概念。为了能较深刻地了解什么是进程，我们将先对进程的特征加以描述。





进程的特征

1) 结构特征

通常的程序是不能并发执行的。为使程序(含数据)能独立运行，应为之配置一进程控制块，即PCB(Process Control Block)；而由程序段、相关的数据段和PCB三部分便构成了进程实体。在早期的UNIX版本中，把这三部分总称为“进程映像”。在许多情况下所说的进程，实际上是指进程实体，例如，所谓创建进程，实质上是创建进程实体中的PCB；而撤消进程，实质上是撤消进程的PCB，本教材中也是如此。





第二章 进程管理



2) 动态性

进程的实质是进程实体的一次执行过程，动态性是进程的最基本的特征。动态性还表现在：“它由创建而产生，由调度而执行，由撤消而消亡”。可见，进程实体有一定的生命期，而程序是静态的。





第二章 进程管理



3) 并发性

是指多个进程实体同存于内存中，且能在一段时间内同时运行。并发性是进程的重要特征，也是OS的重要特征。引入进程的目的也正是为了使其进程实体能和其它进程实体并发执行；而程序(没有建立PCB)是不能并发执行的。

4) 独立性

在传统的OS中，独立性是指进程实体是能独立分配资源和独立接受调度，能独立运行的基本单位。凡未建立PCB的程序都不能作为一个独立的单位参与运行。





第二章 进程管理



5) 异步性

进程实体按异步方式运行。

现在我们来讨论进程的定义。曾有许多人从不同的角度对进程下过定义，其中较典型的进程定义有：

- (1) 进程是程序的一次执行。
- (2) 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- (3) 进程是程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。





2. 进程的三种基本状态

进程执行时的间断性决定运行中的进程可能具有以下三种基本状态。

1) 就绪(Ready)状态

当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行，进程这时的状态称为就绪状态。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列。





第二章 进程管理



2) 执行状态

进程已获得CPU，其程序正在执行。在单处理机系统中，只有一个进程处于执行状态；在多处理机系统中，则有多个进程处于执行状态。

3) 阻塞状态

正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，亦即进程的执行受到阻塞，把这种暂停状态称为阻塞状态，有时也称为等待状态或封锁状态。致使进程阻塞的典型事件有：请求I/O，申请缓冲空间等。通常将这种处于阻塞状态的进程也排成一个队列。有的系统则根据阻塞原因的不同而把处于阻塞状态的进程排成多个队列。



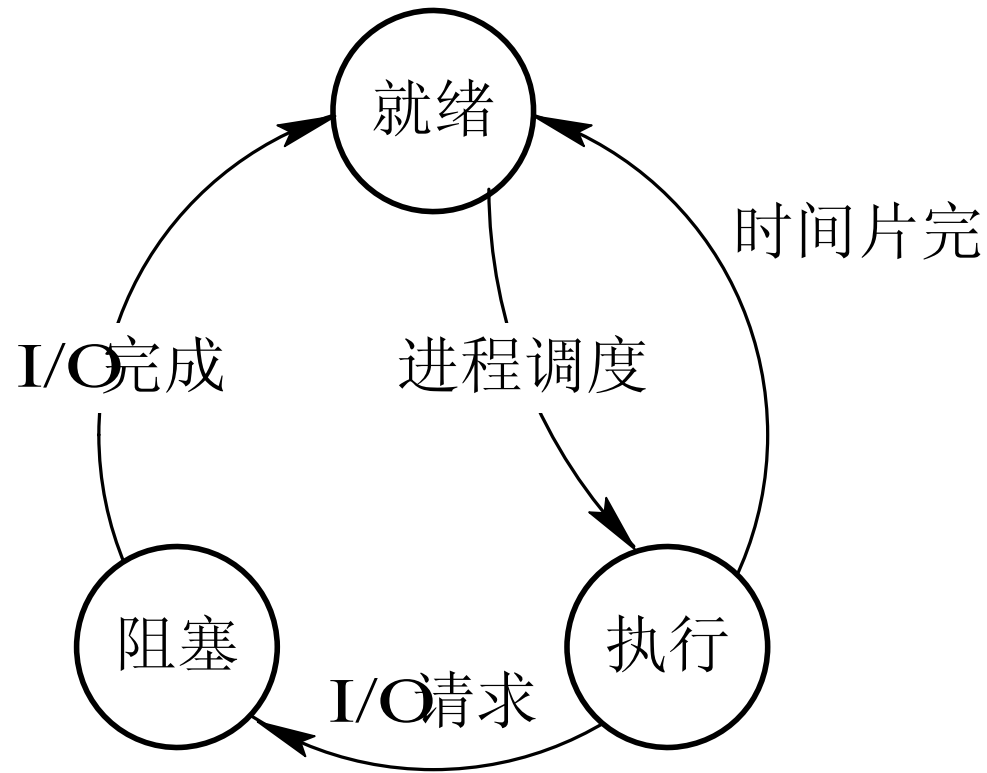


图2-5 进程的三种基本状态及其转换





3. 挂起状态

1) 引入挂起状态的原因

在不少系统中进程只有上述三种状态，但在另一些系统中，又增加了一些新状态，最重要的是挂起状态。引入挂起状态的原因有：

(1) 终端用户的请求。当终端用户在自己的程序运行期间发现有可疑问题时，希望暂时使自己的程序静止下来。亦即，
1，使正在执行的进程暂停执行；2，若此时用户进程正处于就绪状态而未执行，则该进程暂不接受调度，以便用户研究其执行情况或对程序进行修改。我们把这种静止状态称为挂起状态。





第二章 进程管理



(2) 父进程请求。有时父进程希望挂起自己的某个子进程，以便考查和修改该子进程，或者协调各子进程间的活动。

(3) 负荷调节的需要。当实时系统中的工作负荷较重，已可能影响到对实时任务的控制时，可由系统把一些不重要的进程挂起，以保证系统能正常运行。

(4) 操作系统的需要。操作系统有时希望挂起某些进程，以便检查运行中的资源使用情况或进行记账。





2) 进程状态的转换

在引入挂起状态后，又将增加从挂起状态(又称为静止状态)到非挂起状态(又称为活动状态)的转换；或者相反。可有以下几种情况：

(1) 活动就绪→静止就绪。当进程处于未被挂起的就绪状态时，称此为活动就绪状态，表示为Readya。当用挂起原语Suspend将该进程挂起后，该进程便转变为静止就绪状态，表示为Readys，处于Readys状态的进程不再被调度执行。





第二章 进程管理

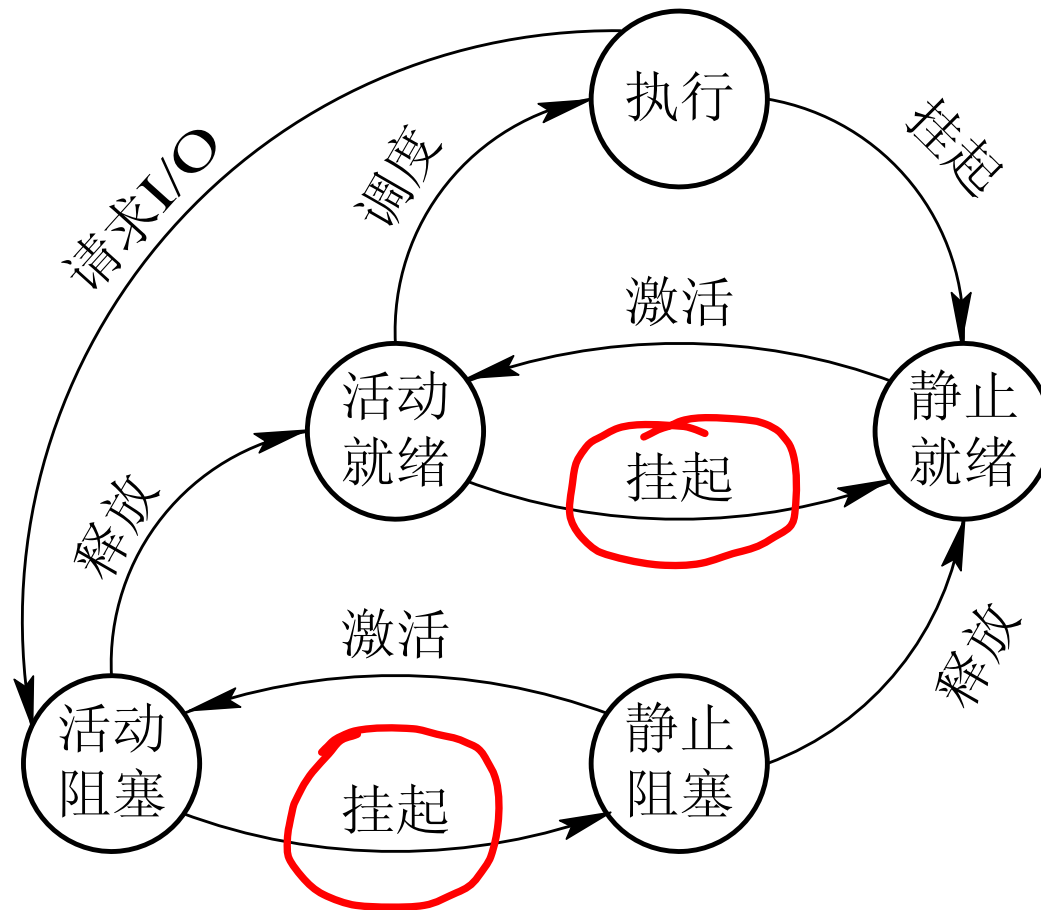


图 2-6 具有挂起状态的进程状态图





第二章 进程管理



(2) 活动阻塞→静止阻塞。当进程处于未被挂起的阻塞状态时，称它是处于活动阻塞状态，表示为Blocked_a。当用Suspend原语将它挂起后，进程便转变为静止阻塞状态，表示为Blocked_s。处于该状态的进程在其所期待的事件出现后，将从静止阻塞变为静止就绪。

(3) 静止就绪→活动就绪。处于Ready_s状态的进程，若用激活原语Active激活后，该进程将转变为Ready_a状态。

(4) 静止阻塞→活动阻塞。处于Blocked_s状态的进程，若用激活原语Active激活后，该进程将转变为Blocked_a状态。图2-6示出了具有挂起状态的进程状态图。





4. 创建状态和终止状态

1) 创建状态

创建一个进程两个步骤：1，为一个新进程创建PCB，并填写必要的管理信息；2，把该进程转入就绪状态并插入就绪队列之中。当一个新进程被创建时，系统已为其分配了PCB，填写了进程标识等信息，但由于该进程所必需的资源或其它信息，如主存资源尚未分配等，一般而言，此时的进程已拥有了自己的PCB，但进程自身还未进入主存，即创建工作尚未完成，进程还不能被调度运行，其所处的状态就是创建状态。





第二章 进程管理



引入创建状态，是为了保证进程的调度必须在创建工作完成后进行，以确保对进程控制块操作的完整性。同时，创建状态的引入，也增加了管理的灵活性，操作系统可以根据系统性能或主存容量的限制，推迟创建状态进程的提交。对于处于创建状态的进程，获得了其所必需的资源，以及对其PCB初始化工作完成后，进程状态便可由创建状态转入就绪状态。





第二章 进程管理



2) 终止状态

进程的终止也要通过2个步骤：1等待操作系统进行善后处理，然后将其PCB清零，并将PCB空间返还系统。当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结，它将进入终止状态。进入终止态的进程以后不能再执行，但在操作系统中依然保留一个记录，其中保存状态码和一些计时统计数据，供其它进程收集。一旦其它进程完成了对终止状态进程的信息提取之后，操作系统将删除该进程。

图2-7示出了增加了创建状态和终止状态后，进程的三种基本状态及转换图衍变为五种状态及转换关系图。





第二章 进程管理

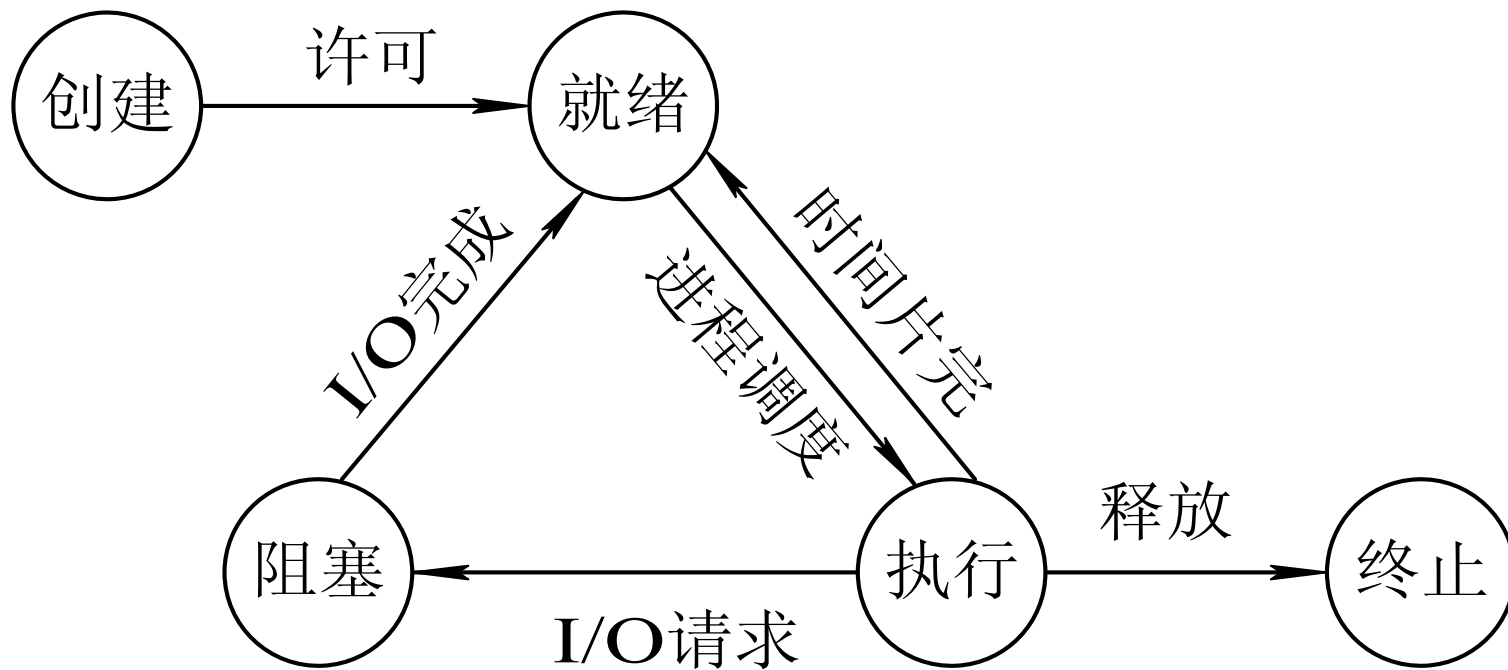


图2-7 进程的五种基本状态及转换



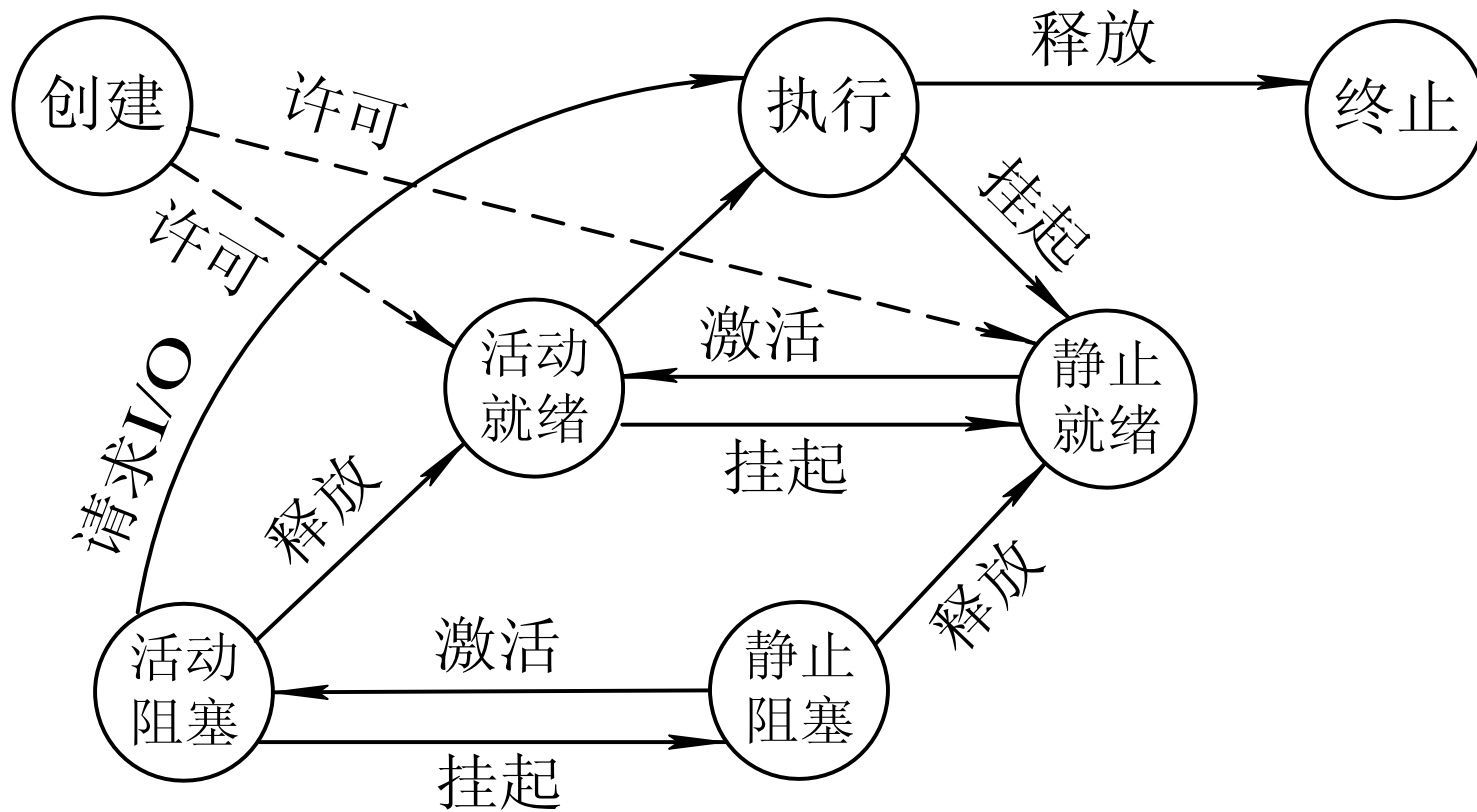


图2-8 具有创建、终止和挂起状态的进程状态图





第二章 进程管理



如图2-8所示，引进创建和终止状态后，在进程状态转换时，相比较图2-7所示的进程五状态转换而言，需要增加考虑下面的几种情况。

(1) NULL→创建：一个新进程产生时，该进程处于创建状态。

(2) 创建→活动就绪：在当前系统的性能和内存的容量均允许的情况下，完成对进程创建的必要操作后，相应的系统进程将进程的状态转换为活动就绪状态。





第二章 进程管理



(3) 创建→静止就绪：考虑到系统当前资源状况和性能要求，并不分配给新建进程所需资源，主要是主存资源，相应的系统进程将进程状态转为静止就绪状态，对换到外存，不再参与调度，此时进程创建工作尚未完成。

(4) 执行→终止：当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结，进程即进终止状态。





2.1.5 进程控制块

1. 进程控制块的作用

为了描述和控制进程的运行，系统为每个进程定义了一个数据结构——进程控制块PCB(Process Control Block)，它是进程实体的一部分，是操作系统中最重要的记录型数据结构。PCB中记录了操作系统所需的、用于描述进程的当前情况以及控制进程运行的全部信息。进程控制块的作用是使一个在多道程序环境下不能独立运行的程序(含数据)，成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程。





第二章 进程管理



或者说，OS是根据PCB来对并发执行的进程进行控制和管理。例如，当OS要调度某进程执行时，要从该进程的PCB中查出其现行状态及优先级；在调度到某进程后，要根据其PCB中所保存的处理机状态信息，设置该进程恢复运行的现场，并根据其PCB中的程序和数据内存始址，找到其程序和数据；进程在执行过程中，当需要和与之合作的进程实现同步、通信或访问文件时，也都需要访问PCB；当进程由于某种原因而暂停执行时，又须将其断点的处理机环境保存在PCB中。在进程的整个生命期中，系统总是通过PCB对进程进行控制的，PCB是进程存在的惟一标志。





第二章 进程管理



创建新进程，为它建立PCB；进程结束回收PCB，进程也消亡。PCB可以被操作系统中的多个模块读或修改，如被调度程序、资源分配程序、中断处理程序以及监督和分析程序等。因为PCB经常被系统访问，尤其是被运行频率很高的进程及分派程序访问，故PCB应常驻内存。系统将所有的PCB组织成若干个链表(或队列)，存放在操作系统中专门开辟的PCB区内。在Linux系统中用task_struct数据结构来描述每个进程的进程控制块，在Windows操作系统中则使用一个执行体进程块(EPROCESS)来表示进程对象的基本属性。





2. 进程控制块中的信息(PCB)

1) 进程标识符

进程标识符用于唯一地标识一个进程。一个进程通常有两种标识符：

(1) 内部标识符。系统中，每一个进程赋予惟一数字标识符，它通常是一个进程的序号。内部标识符是为了方便系统使用。

(2) 外部标识符。它由创建者提供，通常是由字母、数字组成，往往是由用户(进程)在访问该进程时使用。为了描述进程的家族关系，还应设置父进程标识及子进程标识。此外，还可设置用户标识，以指示拥有该进程的用户。





第二章 进程管理



2) 处理机状态

处理机状态信息主要是由处理机的各种寄存器中的内容组成的。处理机在运行时，信息放在寄存器中。处理机被中断时，所有这些信息都必须保存在PCB中，以便在该进程重新执行时，能从断点继续执行。这些寄存器包括：① 通用寄存器，又称为用户可视寄存器，它们是用用户程序可以访问的，用于暂存信息，在大多数处理机中，有8~32个通用寄存器，在RISC结构的计算机中可超过100个；② 指令计数器，其中存放了要访问的下一条指令的地址；③ 程序状态字PSW，其中含有状态信息，如条件码、执行方式、中断屏蔽标志等；④ 用户栈指针，指每个用户进程都有一个或若干个与之相关的系统栈，用于存放过程和系统调用参数及调用地址，栈指针指向该栈的栈顶。





第二章 进程管理



3) 进程调度信息

在PCB中还存放一些与进程调度和进程对换有关的信息，包括：① 进程状态，指明进程的当前状态，作为进程调度和对换时的依据；② 进程优先级，用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机；③ 进程调度所需的其它信息，它们与所采用的进程调度算法有关，比如，进程已等待CPU的时间总和、进程已执行的时间总和等；④ 事件，指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。





4) 进程控制信息

进程控制信息包括：① 程序和数据的地址，指进程的程序和数据所在的内存或外存地(首)址，以便再调度到该进程执行时，能从PCB中找到其程序和数据；② 进程同步和通信机制，指实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等，它们可能全部或部分地放在PCB中；③ 资源清单，即一张列出了除CPU以外的、进程所需的全部资源及已经分配到该进程的资源清单；④ 链接指针，它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。



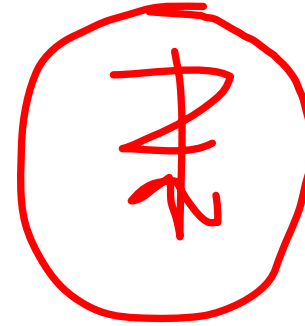


第二章 进程管理



3. 进程控制块的组织方式

1) 链接方式



把具有同一状态的PCB，用链接字链接成队列。可以形成就绪队列、若干个阻塞队列和空白队列等。对其中的就绪队列常按进程优先级的高低排列，把优先级高的进程的PCB排在队列前面。此外，也可根据阻塞原因的不同而把处于阻塞状态的进程的PCB排成等待I/O操作完成的队列和等待分配内存的队列等。图2-9示出了一种链接队列的组织方式。





第二章 进程管理

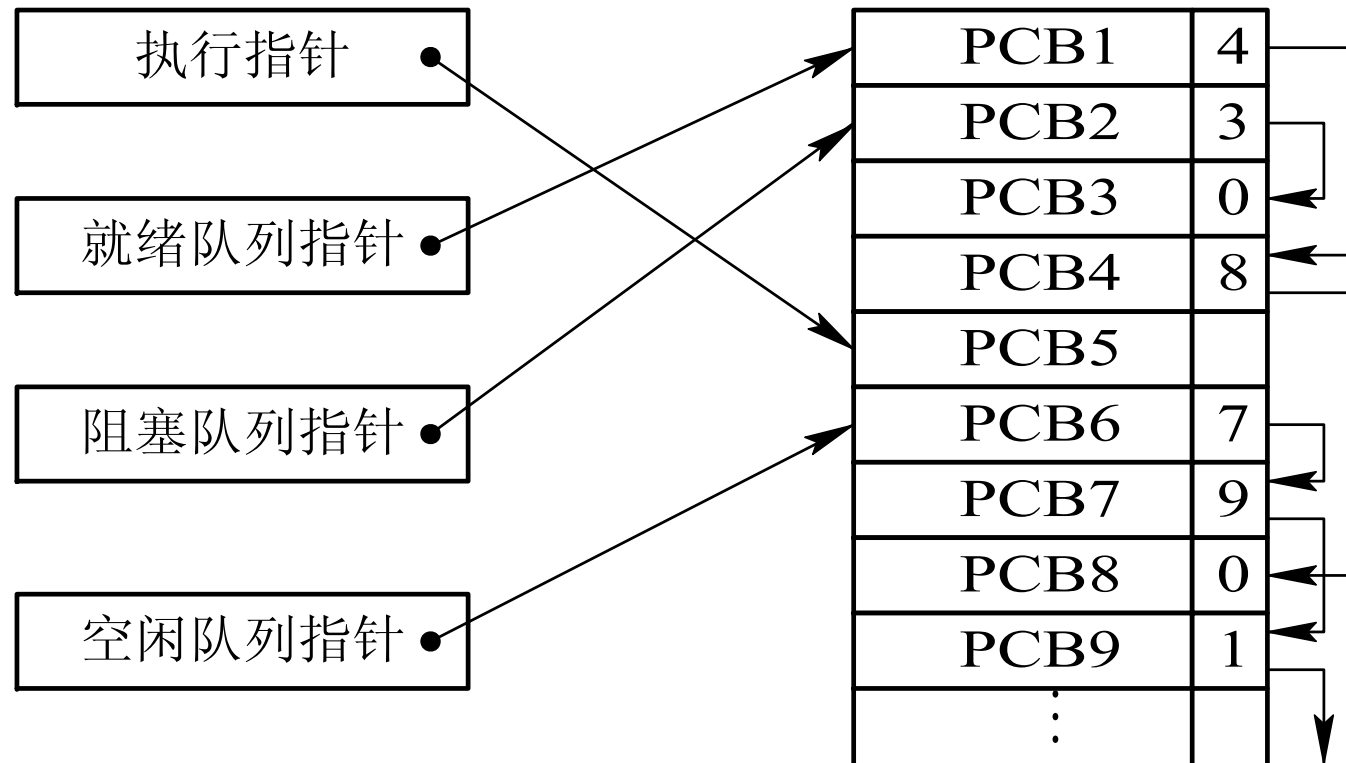


图 2-9 PCB链接队列示意图





第二章 进程管理



2) 索引方式

系统根据所有进程状态建立几张索引表。例如，就绪索引表、阻塞索引表等，并把各索引表在内存的首地址记录在内存的一些专用单元中。索引表的表目中，记录具有相应状态的某个PCB在PCB表中的地址。图2-10示出了索引方式的PCB组织。





第二章 进程管理

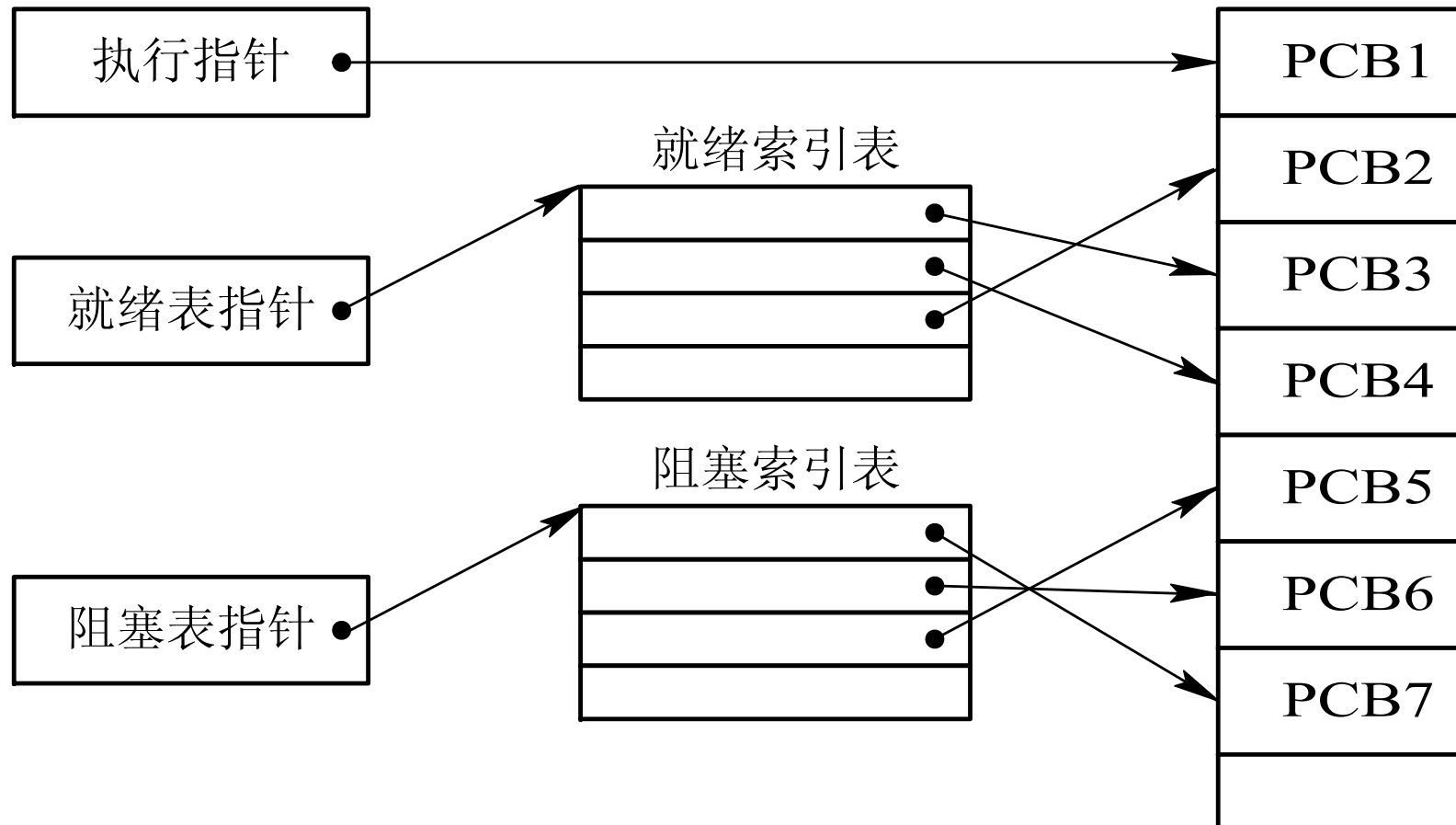
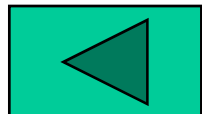


图 2-10 按索引方式组织PCB





2.2 进程控制

进程控制是进程管理中最基本的功能。它用于创建一个新进程，终止一个已完成的进程，或终止一个因出现某事件而使其无法运行下去的进程，还可负责进程运行中的状态转换。如当一个正在执行的进程因等待某事件而暂时不能继续执行时，将其转换为阻塞状态，而当该进程所期待的事件出现时，又将该进程转换为就绪状态等等。进程控制一般是由OS的内核中的原语来实现的。





第二章 进程管理



原语(Primitive)是由若干条指令组成的，用于完成一定功能的一个过程。它与一般过程的区别在于：它们是“原子操作(Action Operation)”。原子操作，是不可分割的基本单位，在执行中不允许被中断。原子操作在管态下执行，常驻内存。

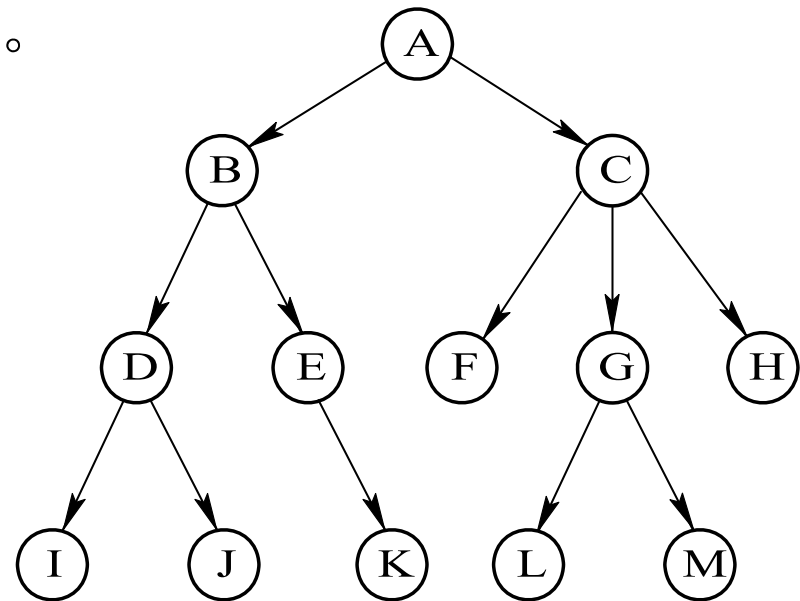




2.2.1 进程的创建

1. 进程图(Process Graph)

进程图用于描述一个进程的家族关系的有向树，如图2-11所示。图中结点代表进程。D是I的父进程(Parent Process)，I是D的子进程(Progeny Process)。





2. 引起创建进程的事件

在多道程序环境中，只有进程才能在系统中运行。因此，为使程序能运行，就必须为它创建进程。导致一个进程去创建另一个进程的典型事件，可有以下四类：

(1) **用户登录**。在分时系统中，用户在终端键入登录命令后，如果是合法用户，系统将为该终端建立一个进程，并把它插入就绪队列中。

(2) **作业调度**。在批处理系统中，当作业调度程序按算法调度到某作业时，便装入内存，分配资源，并立即为它创建进程，再插入就绪队列中。





第二章 进程管理



(3) **提供服务**。当运行中的用户程序提出某种请求后，系统将专门创建一个进程来提供用户所需要的服务，例如，用户程序要求进行文件打印，操作系统将为它创建一个打印进程，可使打印进程与该用户进程并发执行，而且还便于计算出为完成打印任务所花费的时间。





第二章 进程管理



(4) **应用请求**。上述三种情况，都是由系统内核创建一个新进程；**第4类事件则是基于应用进程的需求**，由它自己创建一个新进程，以便使新进程以并发运行方式完成特定任务。例如，某应用程序需要不断地从键盘终端输入数据，继而又要对输入数据进行相应的处理，然后，再将处理结果以表格形式在屏幕上显示。该应用进程为使这几个操作能并发执行，**以加速任务的完成，可以分别建立键盘输入进程、表格输出进程。**





3. 进程的创建(Creation of Process)

一旦操作系统发现了要求创建新进程的事件后，便调用进程创建原语Creat()按下述步骤

(1) 申请空白PCB。为新进程申请获得惟一的数字标识符，并从PCB集合中索取一个空白PCB。





第二章 进程管理



(2) 为新进程分配资源。为程序和数据以及用户栈分配必要内存空间。此时操作系统必须知道新进程所需内存的大小。对于批处理作业，其大小可在用户提出创建进程要求时提供。若是为应用进程创建子进程，也应是在该进程提出创建进程的请求中给出所需内存的大小。对于交互型作业，用户可以不给出内存要求而由系统分配一定的空间。如果新进程要共享某个已在内存的地址空间(即已装入内存的共享段)，则必须建立相应的链接。





第二章 进程管理



(3) 初始化PCB。（进程控制块）PCB的初始化包括：① 初始化标识信息，将系统分配的标识符和父进程标识符填入新PCB中；② 初始化处理机状态信息，使程序计数器指向程序的入口地址，使栈指针指向栈顶；③ 初始化处理机控制信息，将进程的状态设置为就绪状态或静止就绪状态，通常把新建的设置最低优先级，除非用户以显式方式提出高优先级要求。

(4) 将新进程插入就绪队列，如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。





2.2.2 进程的终止

1. 引起进程终止的事件

1) 正常结束

系统中，都有一个用于表示进程已经运行完成的指示。例如，在批处理系统中，通常在程序的最后安排一条Holt指令或终止的系统调用。当程序运行到Holt指令时，将产生一个中断，通知OS进程已经完成。在分时系统中，用户可利用Logs off去表示进程运行完毕，同样可产生一个中断，通知OS进程已运行完毕。





第二章 进程管理



2) 异常结束

进程运行期间，由于出现某些错误和故障而迫使进程终止 (Termination of Process)。常见的有下述几种：

- (1) 越界错误。程序访问的存储区超出该进程的区域。
- (2) 保护错。进程试图去访问一个不允许访问的资源或文件，或者以不适当的方式进行访问，例如，进程试图去写一个只读文件。
- (3) 非法指令。试图去执行一条不存在的指令。出现该错误的原因，可能是程序错误地转移到数据区，把数据当成了指令。





第二章 进程管理



- (4) 特权指令错。进程试图去执行一条只允许OS执行的指令。
- (5) 运行超时。进程的执行时间超过指定最大值。
- (6) 等待超时。进程等待某事件的时间超过规定的最大值。
- (7) 算术运算错。进程试图去执行一个被禁止的运算，例如被0除。
- (8) I/O故障。这是指在I/O过程中发生了错误等。





第二章 进程管理



3) 外界干预

是指进程应外界的请求而终止运行。这些干预有：

(1) 操作员或系统干预。由于某种原因，例如，发生了死锁，由操作员或操作系统终止该进程。

(2) 父进程请求。由于父进程具有终止自己的任何子孙进程的权力，因而当父进程提出请求时，系统将终止该进程。

(3) 父进程终止。当父进程终止时，OS也将它的所有子孙进程终止。





2. 进程的终止过程

如果系统中发生了上述要求终止进程的某事件，OS便调用进程终止原语，按下述过程去终止指定的进程。

(1) 根据进程标识符，从PCB集合中检索出该进程的PCB，从中读出该进程的状态。

(2) 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真，用于指示该进程被终止后应重新进行调度。





第二章 进程管理



- (3) 若有子孙进程，应将所有子孙进程予以终止
- (4) 将被终止进程所拥有的全部资源，或者归还给其父进程，或者归还给系统。
- (5) 被终止进程(PCB)从所在队列(或链表)中移出，等待其他程序来搜集信息。





2.2.3 进程的阻塞与唤醒

1. 引起进程阻塞和唤醒的事件

1) 请求系统服务

当正在执行的进程请求操作系统提供服务时，由于某种原因，操作系统并不立即满足该进程的要求时，该进程只能转变为阻塞状态来等待。例如，一进程请求使用某资源，如打印机，由于系统已将打印机分配给其他进程而不能分配给请求进程，这时请求者进程只能被阻塞，仅在其他进程在释放出打印机的同时，才将请求进程唤醒。





2) 启动某种操作

当进程启动某种操作后，如果该进程必须在该操作完成之后才能继续执行，则必须先使该进程阻塞，以等待该操作完成。例如，进程启动了某I/O设备，如果只有在I/O设备完成了指定的I/O操作任务后进程才能继续执行，则该进程在启动了I/O操作后，便自动进入阻塞状态去等待。在I/O操作完成后，再由中断处理程序或中断进程将该进程唤醒。

3) 新数据尚未到达

对于相互合作的进程，如果其中一个进程需要先获得另一(合作)进程提供的数据后才能对数据进行处理，则只要其所需数据尚未到达，该进程只有(等待)阻塞。例如，有两个进程，进程A用于输入数据，进程B对输入数据进行加工。假如A尚未将数据输入完毕，则进程B将因没有所需的处理数据而阻塞；一旦进程A把数据输入完毕，便可去唤醒进程B。





第二章 进程管理



4) 无新工作可做

具有特定功能的系统进程，这种进程完成任务后，便把自己阻塞起来以等待新任务到来。例如，系统中的发送进程，其主要工作是发送数据，若已有的数据已全部发送完成而又无新的发送请求，这时(发送)进程将使自己进入阻塞状态；仅当又有进程提出新的发送请求时，才将发送进程唤醒。





2. 进程阻塞过程

正在执行的进程，当发现上述某事件时，由于无法继续执行，于是进程便通过调用阻塞原语block把自己阻塞。可见，进程的阻塞是进程自身的一种主动行为。进入block过程后，由于此时该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为“阻塞”，并将PCB插入阻塞队列。如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞(等待)队列。最后，转调度程序进行重新调度，





3. 进程唤醒过程

当阻塞进程期待的事件出现时，则由有关进程调用唤醒原语wakeup()，将等待该事件的进程唤醒。唤醒原语执行的过程是：首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态改为就绪，然后再将该PCB插入到就绪队列中。

block原语和wakeup原语作用刚好相反。因此，如果在某进程中调用了阻塞原语，则必须在与之相合作的另一进程中或其他相关的进程中安排唤醒原语，以能唤醒阻塞进程；否则，被阻塞进程将会因不能被唤醒而长久地处于阻塞状态，从而再无机会继续运行。





2.2.4 进程的挂起与激活

1. 进程的挂起

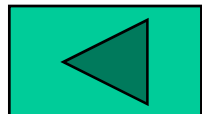
当出现了引起进程挂起的事件时，比如，用户进程请求将自己挂起，或父进程请求将自己的某个子进程挂起，系统将利用挂起原语suspend()将指定进程或处于阻塞状态的进程挂起。挂起原语的执行过程是：首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态的进程，则将之改为静止阻塞。为了方便用户或父进程考查该进程的运行情况而把该进程的PCB复制到某指定的内存区域。最后，若被挂起的进程正在执行，则转向调度程序重新调度。





2. 进程的激活过程

当发生激活进程的事件时，例如，父进程或用户进程请求激活指定进程，若该进程驻留在外存而内存中已有足够的空间时，则可将在外存上处于静止就绪状态的该进程换入内存。系统将利用激活原语`active()`将指定进程激活。激活原语先将进程从外存调入内存，检查该进程的现行状态，若是静止就绪，便将之改为活动就绪；若为静止阻塞，便将之改为活动阻塞。假如采用的是**抢占调度策略**，则每当有新进程进入就绪队列时，应检查是否要进行重新调度，即由调度程序将被激活进程与当前进程进行优先级的比较，**如果被激活进程的优先级更低，就不必重新调度**；否则，立即剥夺当前进程的运行，把处理机分配给刚被激活的进程。





2.3 进程同步

2.3.1 进程同步的基本概念

1. 两种形式的制约关系 并发执行间可能存在两种制约关系。

(1) 间接相互制约关系。同处于一个系统中的进程，通常都共享着某种系统资源，如共享CPU、共享I/O设备等。所谓间接相互制约即源于这种资源共享，例如，有两个进程A和B，如果在A进程提出打印请求时，系统已将惟一的一台打印机分配给了进程B，则此时进程A只能阻塞；一旦进程B将打印机释放，则A进程才能由阻塞改为就绪状态。

(2) 直接相互制约关系。这种制约主要源于进程间的合作。例如，有一输入进程A通过单缓冲向进程B提供数据。当该缓冲空时，计算进程因不能获得所需数据而阻塞，而当进程A把数据输入缓冲区后，便将进程B唤醒；反之，当缓冲区已满时，进程A因不能再向缓冲区投放数据而阻塞，当进程B将缓冲区数据取走后便可唤醒A。





第二章 进程管理



2. 临界资源（同意时刻只能被一个进程使用）

许多硬件资源如打印机，临界资源(Critical Resouce)，进程间应采取互斥方式对临界资源资源的共享。

生产者-消费者(producer-consumer)问题是一个著名的进程同步问题。它描述的是：有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池，生产者进程将它所生产的产品放入一个缓冲区中；消费者进程可从一个缓冲区中取走产品去消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品，也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。





第二章 进程管理



我们可利用一个数组来表示上述的具有 n 个 $(0, 1, \dots, n-1)$ 缓冲区的缓冲池。用输入指针 in 来指示下一个可投放产品的缓冲区，每当生产者进程生产并投放一个产品后，输入指针加1；用一个输出指针 out 来指示下一个可从中获取产品的缓冲区，每当消费者进程取走一个产品后，输出指针加1。由于这里的缓冲池是组织成循环缓冲的，故应把输入指针加1表示成 $in := (in+1) \bmod n$ （这样保证 in 永远在 n 中成为循环）；输出指针加1表示成 $out := (out+1) \bmod n$ 。当 $(in+1) \bmod n = out$ 时表示缓冲池满；而 $in = out$ 则表示缓冲池空。此外，还引入了一个整型变量 $counter$ ，其初始值为0。每当生产者进程向缓冲池中投放一个产品后，使 $counter$ 加1；反之，每当消费者进程从中取走一个产品时，使 $counter$ 减1。生产者和消费者两进程共享下面的变量：





第二章 进程管理



Var n, integer;

type item=...;

var buffer: array [0, 1, ..., n-1] of item;

in, out: 0, 1, ..., n-1;

counter: 0, 1, ..., n;





第二章 进程管理



指针in和out初始化为1。在生产者和消费者进程的描述中，noop是一条空操作指令，while condition do no-op语句表示重复的测试条件(condication)，重复测试应进行到该条件变为false(假)，即到该条件不成立时为止。在生产者进程中使用一局部变量nextp，用于暂时存放每次刚生产出来的产品；而在消费者进程中，则使用一个局部变量nextc，用于存放每次要消费的产品。





第二章 进程管理



```
producer: repeat
    :
    produce an item in nextp;
    :
    while counter=n do no-op;
    buffer [in] :=nextp;
    in:=in+1 mod n;
    counter:=counter+1;
until false;
```





第二章 进程管理



consumer: repeat

while counter=0 do no-op;

nextc:=buffer [out] ;

out:=(out+1) mod n;

counter:=counter-1;

consumer the item in nextc;

until false;





第二章 进程管理



虽然上面的生产者程序和消费者程序在分别看时都是正确的，而且两者在顺序执行时其结果也会是正确的，但若并发执行时就会出现差错，问题就在于这两个进程共享变量 `counter`。生产者对它做加1操作，消费者对它做减1操作，这两个操作在用机器语言实现时，常可用下面的形式描述：

```
register1:=counter;      register2:=counter;
```

```
register1:=register1+1;   register2:=register2-1;
```

```
counter:=register1;      counter:=register2;
```





第二章 进程管理



假设counter的当前值是5。如果生产者进程先执行左列的三条机器语言语句，然后消费者进程再执行右列的三条语句，则最后共享变量counter的值仍为5；反之，如果让消费者进程先执行右列的三条语句，然后再让生产者进程执行左列的三条语句，则counter值也还是5，但是，如果按下述顺序执行：

register1:=counter;	(register1=5)

register1:=register1+1;	(register1=6)

register2:=counter;	(register2=5)

register2:=register2-1;	(register2=4)

counter:=register1;	(counter=6)

counter:=register2;	(counter=4)





第二章 进程管理



正确的counter值应当是5，但现在是4。读者可以自己试试，倘若再将两段程序中各语句交叉执行的顺序改变，将可看到又可能得到counter=6的答案，这表明程序的执行已经失去了再现性。为了预防产生这种错误，解决此问题的关键是应把变量counter作为临界资源处理，亦即，令生产者进程和消费者进程互斥地访问变量counter。





3. 临界区

不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。进程中访问临界资源的那段代码称为临界区(critical section)。若能保证诸进程互斥地进入自己的临界区，便可实现诸进程对临界资源的互斥访问。为此，每个进程在进入临界区之前，应先对欲访问的临界资源进行检查，看它是否正被访问。如果此刻该临界资源未被访问，进程便可进入临界区对该资源进行访问，并设置它正被访问的标志；如果此刻该临界资源正被某进程访问，则本进程不能进入临界区。因此，必须在临界区前面增加一段用于进行上述检查的代码，把这段代码称为进入区(entry section)。相应地，在临界区后面也要加上一段称为退出区(exit section)的代码，用于将临界区正被访问的标志恢复为未被访问的标志。





第二章 进程管理



进程中除进入区、临界区及退出区之外的其它部分的代码，在这里都称为剩余区。这样，可把一个访问临界资源的循环进程描述如下：

repeat

entry section

critical section;

exit section

remainder section;

until false;





4. 同步机制应遵循的规则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统中设置专门的同步机构来协调各进程间的运行。所有同步机制都应遵循下述四条准则：

(1) 空闲让进。当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效地利用临界资源。

(2) 忙则等待。当已有进程进入临界区时，表明临界资源正在被访问，因而其它试图进入临界区的进程必须等待，以保证对临界资源的互斥访问。





第二章 进程管理



(3) 有限等待。对要求访问临界资源的进程，应保证在有限时间内能进入自己的临界区，以免陷入“死等”状态。

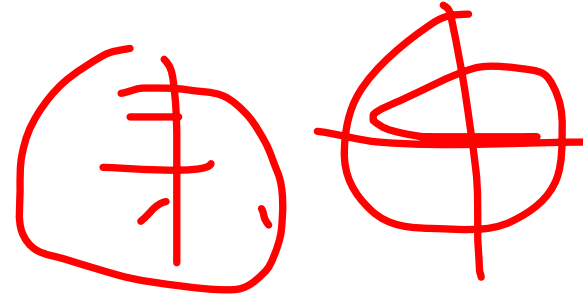
(4) 让权等待。当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”状态。





2.3.2 信号量机制

1. 整型信号量



最初由Dijkstra把整型信号量定义为一个用于表示资源数目的整型量 S ，它与一般整型量不同，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) $\text{wait}(S)$ 和 $\text{signal}(S)$ 来访问。很长时间以来，这两个操作一直被分别称为P、V操作。

$\text{Wait}(S)$ 和 $\text{signal}(S)$ 操作可描述为：

$\text{wait}(S)$: while $S \leq 0$ do no-op;

$S := S - 1$;

$\text{signal}(S)$: $S := S + 1$;





第二章 进程管理



$\text{wait}(S)$ 和 $\text{signal}(S)$ 是两个原子操作，因此，它们在执行时是不可中断的。亦即，当一个进程在修改某信号量时，没有其他进程可同时对该信号量进行修改。此外，在 wait 操作中，对 S 值的测试和做 $S:=S-1$ 操作时都不可中断。





2. 记录型信号量

在整型信号量机制中的wait操作，只要是信号量 $S \leq 0$ ，就会不断地测试。因此，该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态。记录型信号量机制则是一种不存在“忙等”现象的进程同步机制。但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表指针L，用于链接上述的所有等待进程。记录型信号量是由于它采用了记录型的数据结构而得名的。它所包含的上述两个数据项可描述为：





第二章 进程管理



```
type semaphore=record
```

```
    value: integer;
```

```
    L: list of process;
```

```
end
```





第二章 进程管理



相应地，wait(S)和signal(S)操作可描述为：

```
procedure wait(S)
```

```
    var S: semaphore;
```

```
    begin
```

```
        S.value:=S.value-1;
```

```
        if S.value<0 then block(S.L);
```

```
    end
```

```
procedure signal(S)
```

```
    var S: semaphore;
```

```
    begin
```

```
        S.value:=S.value+1;
```

```
        if S.value<=0 then wakeup(S.L);
```

```
    end
```





第二章 进程管理



在记录型信号量机制中， $S.value$ 的初值表示系统中某类资源的数目，因而又称为资源信号量。对它的每次wait操作，意味着进程请求一个单位的该类资源，使系统中可供分配的该类资源数减少一个，因此描述为 $S.value:=S.value-1$ ；当 $S.value<0$ 时，表示该类资源已分配完毕，因此进程应调用block原语，进行自我阻塞，放弃处理机，并插入到信号量链表 $S.L$ 中。可见，该机制遵循了“让权等待”准则。此时 $S.value$ 的绝对值表示在该信号量链表中已阻塞进程的数目。对信号量的每次signal操作，表示执行进程释放一个单位资源，使系统中可供分配的该类资源数增加一个，故 $S.value:=S.value+1$ 操作表示资源数目加1。若加1后仍是 $S.value\leq 0$ ，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用wakeup原语，将 $S.L$ 链表中的第一个等待进程唤醒。如果 $S.value$ 的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量，用于进程互斥。





3. AND型信号量

上述的进程互斥问题，是针对各进程之间只共享一个临界资源而言的。在有些应用场合，是一个进程需要先获得两个或更多的共享资源后方能执行其任务。假定现有两个进程A和B，他们都要求访问共享数据D和E。当然，共享数据都应作为临界资源。为此，可为这两个数据分别设置用于互斥的信号量Dmutex和Emutex，并令它们的初值都是1。相应地，在两个进程中都要包含两个对Dmutex和Emutex的操作，即

process A:

wait(Dmutex);

wait(Emutex);

process B:

wait(Emutex);

wait(Dmutex);





第二章 进程管理



若进程A和B按下述次序交替执行wait操作：

process A: wait(Dmutex); 于是Dmutex=0

process B: wait(Emutex); 于是Emutex=0

process A: wait(Emutex); 于是Emutex=-1 A阻塞

process B: wait(Dmutex); 于是Dmutex=-1 B阻塞

最后，进程A和B处于僵持状态。在无外力作用下，两者都将无法从僵持状态中解脱出来。我们称此时的进程A和B已进入死锁状态。显然，当进程同时要求的共享资源愈多时，发生进程死锁的可能性也就愈大。





第二章 进程管理



AND同步机制的基本思想是：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源也不分配给它。亦即，对若干个临界资源的分配，采取原子操作方式：要么把它所请求的资源全部分配到进程，要么一个也不分配。由死锁理论可知，这样就可避免上述死锁情况的发生。为此，在wait操作中，增加了一个“AND”条件，故称为AND同步，或称为同时wait操作，即Swait(Simultaneous wait)定义如下：





第二章 进程管理



```
Swait( $S_1, S_2, \dots, S_n$ )  
  if  $S_i \geq 1$  and ... and  $S_n \geq 1$  then  
    for  $i:=1$  to  $n$  do  
       $S_i := S_i - 1$ ;  
    endfor  
  else  
    place the process in the waiting queue associated with the  
    first  $S_i$  found with  $S_i < 1$ , and set the program count of this  
    process to the beginning of Swait operation  
  endif  
Signal( $S_1, S_2, \dots, S_n$ )  
  for  $i:=1$  to  $n$  do  
     $S_i := S_i + 1$ ;  
    Remove all the process waiting in the queue associated with  $S_i$   
    into the ready queue.  
  endfor;
```





4. 信号量集

在记录型信号量机制中，`wait(S)`或`signal(S)`操作仅能对信号量施以加1或减1操作，意味着每次只能获得或释放一个单位的临界资源。而当一次需要 N 个某类临界资源时，便要进行 N 次`wait(S)`操作，显然这是低效的。此外，在有些情况下，当资源数量低于某一下限值时，便不予以分配。因而，在每次分配之前，都必须测试该资源的数量，看其是否大于其下限值。基于上述两点，可以对AND信号量机制加以扩充，形成一般化的“信号量集”机制。`Swait`操作可描述如下，其中 S 为信号量， d 为需求值，而 t 为下限值。





第二章 进程管理



Swait($S_1, t_1, d_1, \dots, S_n, t_n, d_n$)

if $S_i \geq t_1$ and ... and $S_n \geq t_n$ then

for $i:=1$ to n do

$S_i := S_i - d_i;$

endfor

else

Place the executing process in the waiting queue of the first S_i with $S_i < t_i$ and set its program counter to the beginning of the Swait Operation.

endive





第二章 进程管理



$S_{\text{signal}}(S_1, d_1, \dots, S_n, d_n)$

for $i:=1$ to n do

$S_i := S_i + d_i;$

Remove all the process waiting in the queue associated with
 S_i into the ready queue

endfor;





第二章 进程管理



下面我们讨论一般“信号量集”的几种特殊情况：

(1) $\text{Swait}(S, d, d)$ 。此时在信号量集中只有一个信号量 S ，但允许它每次申请 d 个资源，当现有资源数少于 d 时，不予分配。

(2) $\text{Swait}(S, 1, 1)$ 。此时的信号量集已蜕化为一般的记录型信号量($S > 1$ 时)或互斥信号量($S = 1$ 时)。

(3) $\text{Swait}(S, 1, 0)$ 。这是一种很特殊且很有用的信号量操作。当 $S \geq 1$ 时，允许多个进程进入某特定区；当 S 变为 0 后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。





2.3.3 信号量的应用

1. 利用信号量实现进程互斥

为使多个进程能互斥地访问某临界资源，只须为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。这样，每个欲访问该临界资源的进程在进入临界区之前，都要先对mutex执行wait操作，若该资源此刻未被访问，本次wait操作必然成功，进程便可进入自己的临界区，这时若再有其他进程也欲进入自己的临界区，此时由于对mutex执行wait操作定会失败，因而该进程阻塞，从而保证了该临界资源能被互斥地访问。当访问临界资源的进程退出临界区后，又应对mutex执行signal操作，以便释放该临界资源。利用信号量实现进程互斥的进程可描述如下：





第二章 进程管理



```
Var mutex: semaphore:=1;  
begin  
  parbegin  
    process 1: begin  
      repeat  
        wait(mutex);  
        critical section  
        signal(mutex);  
        remainder section  
      until false;  
    end
```





第二章 进程管理



process 2: begin

repeat

wait(mutex);

critical section

signal(mutex);

remainder section

until false;

end

parend





2. 利用信号量实现前趋关系

还可利用信号量来描述程序或语句之间的前趋关系。设有两个并发执行的进程 P_1 和 P_2 。 P_1 中有语句 S_1 ； P_2 中有语句 S_2 。我们希望在 S_1 执行后再执行 S_2 。为实现这种前趋关系，我们只须使进程 P_1 和 P_2 共享一个公用信号量 S ，并赋予其初值为0，将 $\text{signal}(S)$ 操作放在语句 S_1 后面；而在 S_2 语句前面插入 $\text{wait}(S)$ 操作，即

在进程 P_1 中，用 S_1 ； $\text{signal}(S)$ ；

在进程 P_2 中，用 $\text{wait}(S)$ ； S_2 ；





第二章 进程管理



由于S被初始化为0，这样，若 P_2 先执行必定阻塞，只有在进程 P_1 执行完 S_1 ；`signal(S)`；操作后使S增为1时， P_2 进程方能执行语句 S_2 成功。同样，我们可以利用信号量，按照语句间的前趋关系(见图2-12)，写出一个更为复杂的可并发执行的程序。

图2-12示出了一个前趋图，其中 $S_1, S_2, S_3, \dots, S_6$ 是最简单的程序段(只有一条语句)。为使各程序段能正确执行，应设置若干个初始值为“0”的信号量。如为保证 $S_1 \rightarrow S_2$ ， $S_1 \rightarrow S_3$ 的前趋关系，应分别设置信号量a和b，同样，为了保证 $S_2 \rightarrow S_4$ ， $S_2 \rightarrow S_5$ ， $S_3 \rightarrow S_6$ ， $S_4 \rightarrow S_6$ 和 $S_5 \rightarrow S_6$ ，应设置信号量c, d, e, f, g。





第二章 进程管理



```
Var a,b,c,d,e,f,g: semaphore: =0,0,0,0,0,0,0;  
begin  
  parbegin  
    begin S1;  signal(a);  signal(b);  end;  
    begin wait(a);  S2;  signal(c);  signal(d);  end;  
    begin wait(b);  S3;  signal(e);  end;  
    begin wait(c);  S4;  signal(f);  end;  
    begin wait(d);  S5;  signal(g);  end;  
    begin wait(e);  wait(f);  wait(g);  S6;  end;  
  parend  
end
```



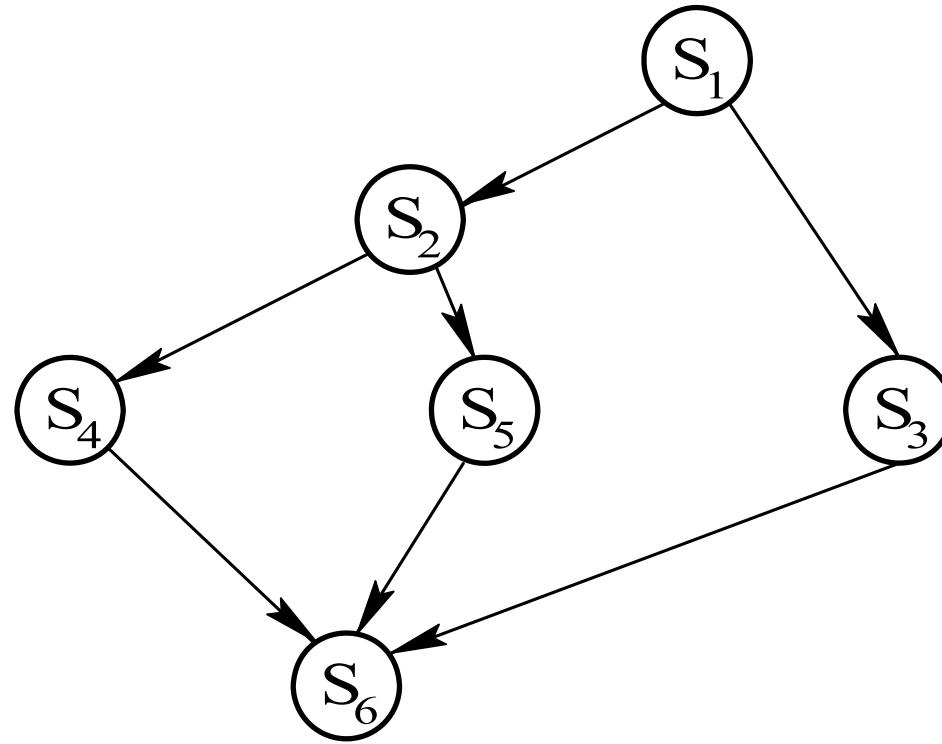


图2-12 前趋图举例





2.3.4 管程机制

1. 管程的定义

系统中的各种硬件资源和软件资源，均可用数据结构抽象地描述其资源特性，即用少量信息和对该资源所执行的操作来表征该资源，而忽略了它们的内部结构和实现细节。例如，对一台电传机，可用与分配该资源有关的状态信息(busy或free)和对它执行请求与释放的操作，以及等待该资源的进程队列来描述。又如，一个FIFO队列，可用其队长、队首和队尾以及在该队列上执行的一组操作来描述。





第二章 进程管理



利用共享数据结构抽象地表示系统中的共享资源，而把对该共享数据结构实施的操作定义为一组过程，如资源的请求和释放过程request和release。进程对共享资源的申请、释放和其它操作，都是通过这组过程对共享数据结构的操作来实现的，这组过程还可以根据资源的情况，或接受或阻塞进程的访问，确保每次仅有一个进程使用共享资源，这样就可以统一管理对共享资源的所有访问，实现进程互斥。





第二章 进程管理



代表共享资源的数据结构，以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序，共同构成了一个操作系统的资源管理模块，我们称之为管程。管程被请求和释放资源的进程所调用。Hansan为管程所下的定义是：“一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作，这组操作能同步进程和改变管程中的数据”。

由上述的定义可知，管程由四部分组成：① 管程的名称；② 局部于管程内部的共享数据结构说明；③ 对该数据结构进行操作的一组过程；④ 对局部于管程内部的共享数据设置初始值的语句。图2-13是一个管程的示意图。





第二章 进程管理

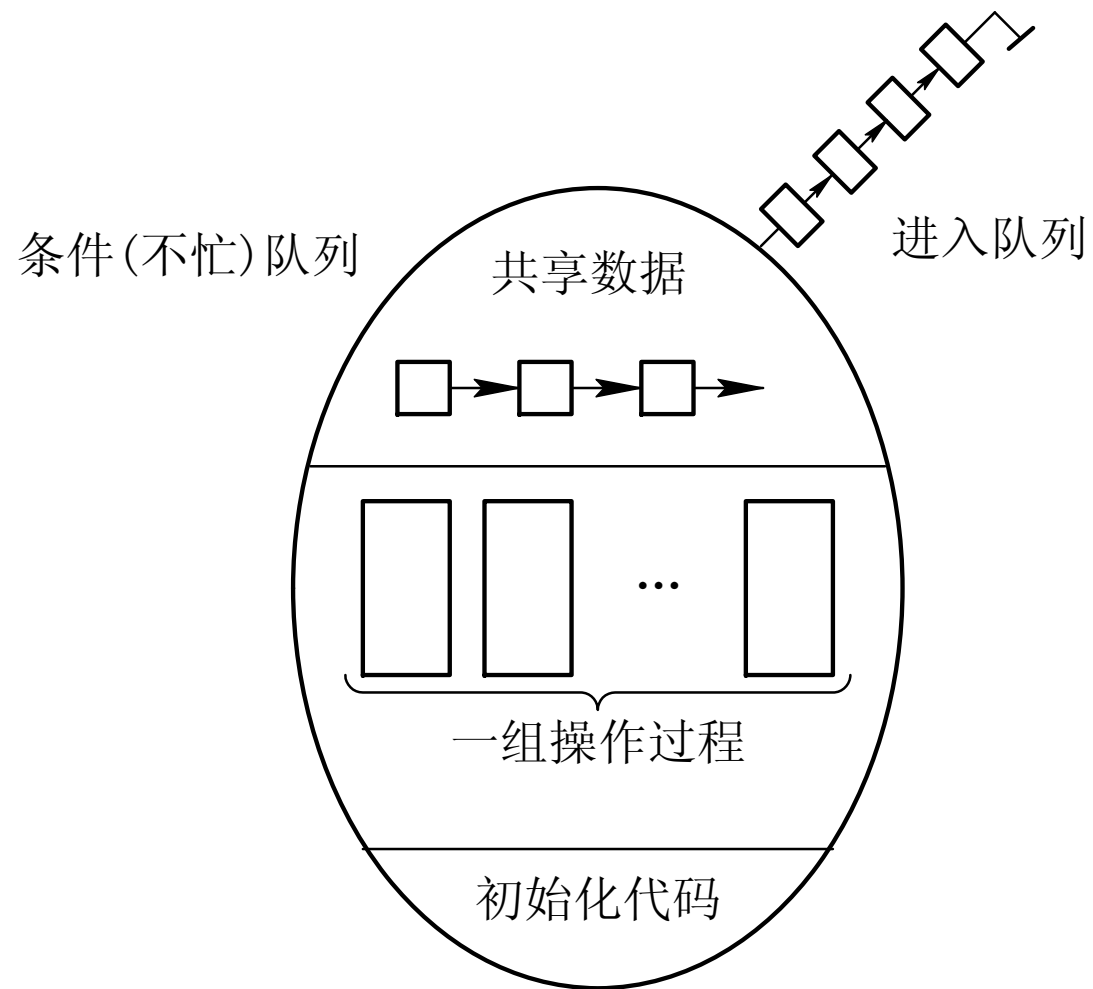


图2-13 管程的示意图



第二章 进程管理



管程的语法描述如下：

```
type monitor_name = MONITOR;
```

```
<共享变量说明>;
```

```
define <(能被其他模块引用的)过程名列表>;
```

```
use <(要调用的本模块外定义的)过程名列表>;
```

```
procedure <过程名>(<形式参数表>);
```

```
begin
```

```
  :
```

```
end;
```

```
  :
```





第二章 进程管理



function <函数名>(<形式参数表>): 值类型;

begin

:

end;

:

begin

<管程的局部数据初始化语句序列>;

end





第二章 进程管理



需要指出的是，局部于管程内部的数据结构，仅能被局部于管程内部的过程所访问，任何管程外的过程都不能访问它；反之，局部于管程内部的过程也仅能访问管程内的数据结构。由此可见，管程相当于围墙，它把共享变量和对它进行操作的若干过程围了起来，所有进程要访问临界资源时，都必须经过管程(相当于通过围墙的门)才能进入，而管程每次只准许一个进程进入管程，从而实现了进程互斥。

管程是一种程序设计语言结构成分，它和信号量有同等的表达能力，从语言的角度看，管程主要有以下特性：





第二章 进程管理



- (1) 模块化。管程是一个基本程序单位，可以单独编译。
- (2) 抽象数据类型。管程中不仅有数据，而且有对数据的操作。
- (3) 信息掩蔽。管程中的数据结构只能被管程中的过程访问，这些过程也是在管程内部定义的，供管程外的进程调用，而管程中的数据结构以及过程(函数)的具体实现外部不可见。





第二章 进程管理



管程和进程不同，主要体现在以下几个方面：

- (1) 虽然二者都定义了数据结构，但进程定义的是私有数据结构PCB，管程定义的是公共数据结构，如消息队列等；
- (2) 二者都存在对各自数据结构上的操作，但进程是由顺序程序执行有关的操作，而管程主要是进行同步操作和初始化操作；
- (3) 设置进程的目的在于实现系统的并发性，而管程的设置则是解决共享资源的互斥使用问题；





第二章 进程管理



(4) 进程通过调用管程中的过程对共享数据结构实行操作，该过程就如通常的子程序一样被调用，因而管程为被动工作方式，进程则为主动工作方式；

(5) 进程之间能并发执行，而管程则不能与其调用者并发；

(6) 进程具有动态性，由“创建”而诞生，由“撤销”而消亡，而管程则是操作系统中的一个资源管理模块，供进程调用。





2. 条件变量

在利用管程实现进程同步时，必须设置同步工具，如两个同步操作原语wait和signal。当某进程通过管程请求获得临界资源而未能满足时，管程便调用wait原语使该进程等待，并将其排在等待队列上，如图2-13所示。仅当另一进程访问完成并释放该资源之后，管程才又调用signal原语，唤醒等待队列中的队首进程。





2. 条件变量

在利用管程实现进程同步时，必须设置同步工具，如两个同步操作原语wait和signal。当某进程通过管程请求获得临界资源而未能满足时，管程便调用wait原语使该进程等待，并将其排在等待队列上，如图2-13所示。仅当另一进程访问完成并释放该资源之后，管程才又调用signal原语，唤醒等待队列中的队首进程。





第二章 进程管理



但是仅仅有上述的同步工具是不够的。考虑一种情况：当一个进程调用了管程，在管程中时被阻塞或挂起，直到阻塞或挂起的原因解除，而在此期间，如果该进程不释放管程，则其它进程无法进入管程，被迫长时间地等待。为了解决这个问题，引入了条件变量condition。通常，一个进程被阻塞或挂起的条件(原因)可有多多个，因此在管程中设置了多个条件变量，对这些条件变量的访问，只能在管程中进行。





第二章 进程管理



管程中对每个条件变量都须予以说明，其形式为：Var x, y: condition。对条件变量的操作仅仅是wait和signal，因此条件变量也是一种抽象数据类型，每个条件变量保存了一个链表，用于记录因该条件变量而阻塞的所有进程，同时提供的两个操作即可表示为x.wait和x.signal，其含义为：

① x.wait: 正在调用管程的进程因x条件需要被阻塞或挂起，则调用x.wait将自己插入到x条件的等待队列上，并释放管程，直到x条件变化。此时其它进程可以使用该管程。





第二章 进程管理



② `x.signal`: 正在调用管程的进程发现`x`条件发生了变化, 则调用`x.signal`, 重新启动一个因`x`条件而阻塞或挂起的进程。如果存在多个这样的进程, 则选择其中的一个, 如果没有, 则继续执行原进程, 而不产生任何结果。这与信号量机制中的`signal`操作不同, 因为后者总是要执行`s:=s+1`操作, 因而总会改变信号量的状态。

如果有进程`Q`因`x`条件处于阻塞状态, 当正在调用管程的进程`P`执行了`x.signal`操作后, 进程`Q` 被重新启动, 此时两个进程`P`和`Q`, 如何确定哪个执行, 哪个等待, 可采用下述两种方式之一进行处理:

- (1) `P`等待, 直至`Q`离开管程或等待另一条件。
- (2) `Q`等待, 直至`P`离开管程或等待另一条件。

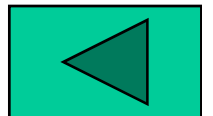




第二章 进程管理



采用哪种处理方式，当然是各执一词。Hoare采用了第一种处理方式，而Hansan选择了两者的折衷，他规定管程中的过程所执行的signal操作是过程体的最后一个操作，于是，进程P执行signal操作后立即退出管程，因而进程Q马上被恢复执行。





2.4 经典进程的同步问题

2.4.1 生产者-消费者问题

1. 利用记录型信号量解决生产者-消费者问题

假定在生产者和消费者之间的公用缓冲池中，具有 n 个缓冲区，这时可利用互斥信号量`mutex`实现诸进程对缓冲池的互斥使用。利用信号量`empty`和`full`分别表示缓冲池中空缓冲区和满缓冲区的数量。又假定这些生产者和消费者相互等效，只要缓冲池未空，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。对生产者—消费者问题可描述如下：





第二章 进程管理



```
Var mutex, empty, full: semaphore:=1,n,0;  
    buffer:array[0, ..., n-1] of item;  
    in, out: integer:=0, 0;  
begin  
    parbegin  
        proceducer: begin  
            repeat  
                ⋮  
            producer an item nextp;  
            ⋮  
            wait(empty);
```





第二章 进程管理



```
wait(mutex);  
buffer(in):=nextp;  
in:=(in+1) mod n;  
signal(mutex);  
signal(full);  
until false;  
  
end  
  
consumer: begin  
    repeat  
        wait(full);  
        wait(mutex);  
        nextc:=buffer(out);
```





第二章 进程管理



```
out:=(out+1) mod n;  
signal(mutex);  
signal(empty);  
consumer the item in nextc;  
until false;  
end  
  
parend  
end
```





第二章 进程管理



在生产者—消费者问题中应注意：首先，在每个程序中用于实现互斥的`wait(mutex)`和`signal(mutex)`必须成对地出现；其次，对资源信号量`empty`和`full`的`wait`和`signal`操作，同样需要成对地出现，但它们分别处于不同的程序中。例如，`wait(empty)`在计算进程中，而`signal(empty)`则在打印进程中，计算进程若因执行`wait(empty)`而阻塞，则以后将由打印进程将它唤醒；最后，在每个程序中的多个`wait`操作顺序不能颠倒，应先执行对资源信号量的`wait`操作，然后再执行对互斥信号量的`wait`操作，否则可能引起进程死锁。





2. 利用AND信号量解决生产者-消费者问题

对于生产者-消费者问题，也可利用AND信号量来解决，即用Swait(empty, mutex)来代替wait(empty)和wait(mutex)；用Ssignal(mutex, full)来代替signal(mutex)和signal(full)；用Swait(full, mutex)来代替wait(full)和wait(mutex)，以及用Ssignal(mutex, empty)代替Signal(mutex)和Signal(empty)。利用AND信号量来解决生产者-消费者问题的算法描述如下：





第二章 进程管理



```
Var mutex, empty, full: semaphore:=1, n, 0;  
    buffer:array[0, ..., n-1] of item;  
    in out: integer:=0, 0;  
begin  
    parbegin  
        producer: begin  
            repeat  
                ⋮  
                produce an item in nextp;  
                ⋮  
                Swait(empty, mutex);  
                buffer(in):=nextp;  
                in:=(in+1)mod n;  
                Ssignal(mutex, full);  
            until false;  
        end
```





第二章 进程管理



```
consumer:begin
    repeat
        Swait(full, mutex);
        Nextc:=buffer(out);
        Out:=(out+1) mod n;
        Ssignal(mutex, empty);
        consumer the item in nextc;
    until false;
end
parend
end
```





3. 利用管程解决生产者-消费者问题

在利用管程方法来解决生产者-消费者问题时，首先便是为它们建立一个管程，并命名为ProclucerConsumer，或简称为PC。其中包括两个过程：

(1) put(item)过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量count来表示在缓冲池中已有的产品数目，当 $\text{count} \geq n$ 时，表示缓冲池已满，生产者须等待。

(2) get(item)过程。消费者利用该过程从缓冲池中取出一个产品，当 $\text{count} \leq 0$ 时，表示缓冲池中已无可取用的产品，消费者应等待。





第二章 进程管理



PC管程可描述如下：

```
type producer-consumer=monitor
```

```
  Var in,out,count: integer;
```

```
    buffer: array[0, ..., n-1] of item;
```

```
    notfull, notempty:condition;
```

```
    procedure entry put(item)
```

```
      begin
```

```
        if count>=n then notfull.wait;
```

```
        buffer(in):=nextp;
```

```
        in:=(in+1) mod n;
```

```
        count:=count+1;
```

```
        if notempty.queue then notempty.signal;
```

```
      end
```





第二章 进程管理



```
procedure entry get(item)
begin
    if count<=0 then notempty.wait;
    nextc:=buffer(out);
    out:=(out+1) mod n;
    count:=count-1;
    if notfull.quene then notfull.signal;
end
begin in:=out:=0;
count:=0
end
```





第二章 进程管理



在利用管程解决生产者—消费者问题时，其中的生产者和消费者可描述为：

```
producer: begin
    repeat
        produce an item in nextp;
        PC.put(item);
    until false;
end
consumer: begin
    repeat
        PC.get(item);
        consume the item in nextc;
    until false;
end
```





2.4.2 哲学家进餐问题

1. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

```
Var chopstick: array[0, ..., 4] of semaphore;
```





第二章 进程管理



所有信号量均被初始化为1，第*i*位哲学家的活动可描述为：

```
repeat
    wait(chopstick[i]);
    wait(chopstick[(i+1)mod 5]);
    :
    eat;
    :
    signal(chopstick[i]);
    signal(chopstick[(i+1)mod 5]);
    :
    think;
until false;
```





第二章 进程管理



在以上描述中，当哲学家饥饿时，总是先去拿他左边的筷子，即执行`wait(chopstick[i]);` 成功后，再去拿他右边的筷子，即执行`wait(chopstick[(i+1)mod 5]);` 又成功后便可进餐。进餐完毕，又先放下他左边的筷子，然后再放右边的筷子。虽然，上述解法可保证不会有两个相邻的哲学家同时进餐，但有可能引起死锁。假如五位哲学家同时饥饿而各自拿起左边的筷子时，就会使五个信号量`chopstick`均为0；当他们再试图去拿右边的筷子时，都将因无筷子可拿而无限期地等待。对于这样的死锁问题，可采取以下几种解决方法：





第二章 进程管理



(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子，而偶数号哲学家则相反。按此规定，将是1、2号哲学家竞争1号筷子；3、4号哲学家竞争3号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。





2. 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。描述如下：

```
Var chopstick array of semaphore:=(1,1,1,1,1);  
processi  
repeat  
    think;  
    Sswait(chopstick[(i+1)mod 5], chopstick[i]);  
    eat;  
    Ssignat(chopstick[(i+1)mod 5], chopstick[i]);  
until false;
```





2.4.3 读者-写者问题

1. 利用记录型信号量解决读者-写者问题

为实现Reader与Writer进程间在读或写时的互斥而设置了一个互斥信号量Wmutex。另外，再设置一个整型变量Readcount表示正在读的进程数目。由于只要有一个Reader进程在读，便不允许Writer进程去写。因此，仅当Readcount=0，表示尚无Reader进程在读时，Reader进程才需要执行Wait(Wmutex)操作。若Wait(Wmutex)操作成功，Reader进程便可去读，相应地，做Readcount+1操作。同理，仅当Reader进程在执行了Readcount减1操作后其值为0时，才须执行signal(Wmutex)操作，以便让Writer进程写。又因为Readcount是一个可被多个Reader进程访问的临界资源，因此，也应该为它设置一个互斥信号量rmutex。





第二章 进程管理



读者-写者问题可描述如下:

```
Var rmutex, wmutex: semaphore:=1,1;
```

```
Readcount: integer:=0;
```

```
begin
```

```
parbegin
```

```
    Reader: begin
```

```
        repeat
```

```
            wait(rmutex);
```

```
            if readcount=0 then wait(wmutex);
```

```
                Readcount:=Readcount+1;
```

```
            signal(rmutex);
```





第二章 进程管理



```
        ⋮  
        perform read operation;  
        ⋮  
        wait(rmutex);  
        readcount:=readcount-1;  
        if readcount=0 then signal(wmutex);  
        signal(rmutex);  
    until false;  
end  
writer: begin  
    repeat  
        wait(wmutex);  
        perform write operation;  
        signal(wmutex);  
    until false;  
end  
parend  
end
```





2. 利用信号量集机制解决读者—写者问题

这里的读者—写者问题与前面的略有不同，它增加了一个限制，即最多只允许 RN 个读者同时读。为此，又引入了一个信号量 L ，并赋予其初值为 RN ，通过执行 $\text{wait}(L, 1, 1)$ 操作，来控制读者的数目。每当有一个读者进入时，就要先执行 $\text{wait}(L, 1, 1)$ 操作，使 L 的值减1。当有 RN 个读者进入读后， L 便减为0，第 $RN+1$ 个读者要进入读时，必然会因 $\text{wait}(L, 1, 1)$ 操作失败而阻塞。对利用信号量集来解决读者—写者问题的描述如下：





第二章 进程管理



```
Var RN integer;  
    L, mx: semaphore:=RN,1;  
begin  
    parbegin  
        reader: begin  
            repeat  
                Swait(L,1,1);  
                Swait(mx,1,0);  
                :  
                perform read operation;  
                :  
                Ssignal(L,1);  
            until false;  
        end
```





第二章 进程管理



```
writer: begin
    repeat
        Swait(mx,1,1; L,RN,0);
        perform write operation;
        Ssignal(mx,1);
    until false;
end
parend
end
```

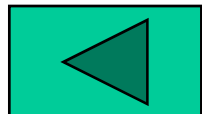




第二章 进程管理



其中， $\text{Swait}(mx, 1, 0)$ 语句起着开关的作用。只要无writer进程进入写， $mx=1$ ，reader进程就都可以进入读。但只要一旦有writer进程进入写时，其 $mx=0$ ，则任何reader进程就都无法进入读。 $\text{Swait}(mx, 1, 1; L, RN, 0)$ 语句表示仅当既无writer进程在写($mx=1$)，又无reader进程在读($L=RN$)时，writer进程才能进入临界区写。





2.5 进程通信

2.5.1 进程通信的类型

1. 共享存储器系统

(1) 基于共享数据结构的通信方式。在这种通信方式中，要求诸进程公用某些数据结构，借以实现诸进程间的信息交换。如在生产者—消费者问题中，就是用有界缓冲区这种数据结构来实现通信的。这里，公用数据结构的设置及对进程间同步的处理，都是程序员的职责。这无疑增加了程序员的负担，而操作系统却只须提供共享存储器。因此，这种通信方式是低效的，只适于传递相对少量的数据。





第二章 进程管理



(2) 基于共享存储区的通信方式。为了传输大量数据，在存储器中划出了一块共享存储区，诸进程可通过对共享存储区中数据的读或写来实现通信。这种通信方式属于高级通信。进程在通信前，先向系统申请获得共享存储区中的一个分区，并指定该分区的关键字；若系统已经给其他进程分配了这样的分区，则将该分区的描述符返回给申请者，继之，由申请者把获得的共享存储分区连接到本进程上；此后，便可像读、写普通存储器一样地读、写该公用存储分区。





2. 消息传递系统

消息传递系统(Message passing system)是当前应用最为广泛的一种进程间的通信机制。在该机制中,进程间的数据交换是以格式化的消息(message)为单位的;在计算机网络中,又把message称为报文。程序员直接利用操作系统提供的一组通信命令(原语),不仅能实现大量数据的传递,而且还隐藏了通信的实现细节,使通信过程对用户是透明的,从而大大减化了通信程序编制的复杂性,因而获得了广泛的应用。





第二章 进程管理



特别值得一提的是，在当今最为流行的微内核操作系统中，微内核与服务器之间的通信，无一例外地都采用了消息传递机制。又由于它能很好地支持多处理机系统、分布式系统和计算机网络，因此它也成为这些领域最主要的通信工具。消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。





3. 管道通信

所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它的操作系统中。





第二章 进程管理



为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

(1) 互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。

(2) 同步，指当写(输入)进程把一定数量(如4 KB)的数据写入pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把它唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

(3) 确定对方是否存在，只有确定了对方已存在时，才能进行通信。





2.5.2 消息传递通信的实现方法

1. 直接通信方式

这是指发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符。通常，系统提供下述两条通信命令(原语):

`Send(Receiver, message);` 发送一个消息给接收进程;

`Receive(Sender, message);` 接收Sender发来的消息;

例如，原语`Send( $P_2$ ,  $m_1$ )`表示将消息 m_1 发送给接收进程 P_2 ；而原语`Receive( $P_1$ ,  $m_1$ )`则表示接收由 P_1 发来的消息 m_1 。





第二章 进程管理



在某些情况下，接收进程可与多个发送进程通信，因此，它不可能事先指定发送进程。例如，用于提供打印服务的进程，它可以接收来自任何一个进程的“打印请求”消息。对于这样的应用，在接收进程接收消息的原语中，表示源进程的参数，也是完成通信后的返回值，接收原语可表示为：

Receive (id, message);





第二章 进程管理



我们还可以利用直接通信原语来解决生产者—消费者问题。当生产者生产出一个产品(消息)后, 使用Send原语将消息发送给消费者进程; 而消费者进程则利用Receive原语来得到一个消息。如果消息尚未生产出来, 消费者必须等待, 直至生产者进程将消息发送过来。生产者—消费者的通信过程可分别描述如下:





第二章 进程管理



repeat

⋮

produce an item in nextp;

⋮

send(consumer, nextp);

until false;

repeat

receive(producer, nextc);

⋮

consume the item in nextc;

until false;





2. 间接通信方式

间接通信方式指进程之间的通信需要通过作为共享数据结构的实体。该实体用来暂存发送进程发送给目标进程的消息；接收进程则从该实体中取出对方发送给自己的消息。通常把这种中间实体称为信箱。消息在信箱中可以安全地保存，只允许核准的目标用户随时读取。因此，利用信箱通信方式，既可实现实时通信，又可实现非实时通信。

系统为信箱通信提供了若干条原语，分别用于信箱的创建、撤消和消息的发送、接收等。





第二章 进程管理



(1) 信箱的创建和撤消。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享); 对于共享信箱, 还应给出共享者的名字。当进程不再需要读信箱时, 可用信箱撤消原语将之撤消。





第二章 进程管理



(2) 消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信：

`Send(mailbox, message);` 将一个消息发送到指定信箱；

`Receive(mailbox, message);` 从指定信箱中接收一个消息；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。





第二章 进程管理



(2) 消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信：

Send(mailbox, message); 将一个消息发送到指定信箱；

Receive(mailbox, message); 从指定信箱中接收一个消息；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。





第二章 进程管理



1) 私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。当拥有该信箱的进程结束时，信箱也随之消失。

2) 公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。





第二章 进程管理



1) 私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。当拥有该信箱的进程结束时，信箱也随之消失。

2) 公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。





第二章 进程管理



(3) 一对多关系。允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式向接收者(多个)发送消息。

(4) 多对多关系。允许建立一个公用信箱，让多个进程都能向信箱中投递消息；也可从信箱中取走属于自己的消息。





2.5.3 消息传递系统实现中的若干问题

1. 通信链路

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路(communication link)。有两种方式建立通信链路。第一种方式是由发送进程在通信之前用显式的“建立连接”命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也用显式方式拆除链路。这种方式主要用于计算机网络中。第二种方式是发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。这种方式主要用于单机系统中。





第二章 进程管理



根据通信链路的连接方法，又可把通信链路分为两类：

- (1) 点—点连接通信链路，这时的一条链路只连接两个结点(进程)；
- (2) 多点连接链路，指用一条链路连接多个($n>2$)结点(进程)。





第二章 进程管理



而根据通信方式的不同，则又可把链路分成两种：

(1) 单向通信链路，只允许发送进程向接收进程发送消息，或者相反；

还可根据通信链路容量的不同而把链路分成两类：一是无容量通信链路，在这种通信链路上没有缓冲区，因而不能暂存任何消息；再者就是有容量通信链路，指在通信链路中设置了缓冲区，因而能暂存消息。缓冲区数目愈多，通信链路的容量愈大。





2. 消息的格式

在消息传递系统中所传递的消息，必须具有一定的消息格式。在单机系统环境中，由于发送进程和接收进程处于同一台机器中，有着相同的环境，故其消息格式比较简单；但在计算机网络环境下，不仅源和目标进程所处的环境不同，而且信息的传输距离很远，可能要跨越若干个完全不同的网络，致使所用的消息格式比较复杂。通常，可把一个消息分成消息头和消息正文两部分。消息头包括消息在传输时所需的控制信息，如源进程名、目标进程名、消息长度、消息类型、消息编号及发送的日期和时间；而消息正文则是发送进程实际上所发送的数据。





第二章 进程管理



在某些OS中，消息采用比较短的定长消息格式，这便减少了对消息的处理和存储开销。这种方式可用于办公自动化系统中，为用户提供快速的便笺式通信；但这对要发送较长消息的用户是不方便的。在有的OS中，采用变长的消息格式，即进程所发送消息的长度是可变的。系统无论在处理还是在存储变长消息时，都可能会付出更多的开销，但这方便了用户。这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。





3. 进程同步方式

在进程之间进行通信时，同样需要有进程同步机制，以使诸进程间能协调通信。不论是发送进程，还是接收进程，在完成消息的发送或接收后，都存在两种可能性，即进程或者继续发送(接收)，或者阻塞。由此，我们可得到以下三种情况：

(1) 发送进程阻塞，接收进程阻塞。这种情况主要用于进程之间紧密同步(tight synchronization)，发送进程和接收进程之间无缓冲时。这两个进程平时都处于阻塞状态，直到有消息传递时。这种同步方式称为汇合(rendezvous)。





第二章 进程管理



(2) 发送进程不阻塞，接收进程阻塞。这是一种应用最广的进程同步方式。平时，发送进程不阻塞，因而它可以尽快地把一个或多个消息发送给多个目标；而接收进程平时则处于阻塞状态，直到发送进程发来消息时才被唤醒。例如，在服务器上通常都设置了多个服务进程，它们分别用于提供不同的服务，如打印服务。平时，这些服务进程都处于阻塞状态，一旦有请求服务的消息到达时，系统便唤醒相应的服务进程，去完成用户所要求的服务。处理完后，若无新的服务请求，服务进程又阻塞。





第二章 进程管理



(3) 发送进程和接收进程均不阻塞。这也是一种较常见的进程同步形式。平时，发送进程和接收进程都在忙于自己的事情，仅当发生某事件使它无法继续运行时，才把自己阻塞起来等待。例如，在发送进程和接收进程之间联系着一个消息队列时，该消息队列最多能接纳 n 个消息，这样，发送进程便可以连续地向消息队列中发送消息而不必等待；接收进程也可以连续地从消息队列中取得消息，也不必等待。只有当消息队列中的消息数已达到 n 个时，即消息队列已满，发送进程无法向消息队列中发送消息时才会阻塞；类似地，只有当消息队列中的消息数为0，接收进程已无法从消息队列中取得消息时才会阻塞。





2.5.4 消息缓冲队列通信机制

1) 消息缓冲区

在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

```
type message buffer=record
```

```
    sender ; 发送者进程标识符
```

```
    size   ; 消息长度
```

```
    text   ; 消息正文
```

```
    next   ; 指向下一个消息缓冲区的指针
```

```
end
```





第二章 进程管理



2) PCB中有关通信的数据项

在操作系统中采用了消息缓冲队列通信机制时，除了需要为进程设置消息缓冲队列外，还应在进程的PCB中增加消息队列队首指针，用于对消息队列进行操作，以及用于实现同步的互斥信号量mutex和资源信号量sm。在PCB中应增加的数据项可描述如下：





第二章 进程管理



type processcontrol block=record

⋮

mq ; 消息队列队首指针

mutex ; 消息队列互斥信号量

sm ; 消息队列资源信号量

⋮

end





2. 发送原语

发送进程在利用发送原语发送消息之前，应先在自己的内存空间设置一发送区a，见图2-14所示。把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的内部标识符j，然后将i挂在j.mq上。由于该队列属于临界资源，故在执行insert操作的前后，都要执行wait和signal操作。





第二章 进程管理



发送原语可描述如下：

```
procedure send(receiver, a)
begin
    getbuf(a.size,i);      根据a.size申请缓冲区;
    i.sender:= a.sender;   将发送区a中的信息复制到消息缓冲区i中;
    i.size:=a.size;
    i.text:=a.text;
    i.next:=0;
    getid(PCB set, receiver.j); 获得接收进程内部标识符;
    wait(j.mutex);
    insert(j.mq, i);         将消息缓冲区插入消息队列;
    signal(j.mutex);
    signal(j.sm);
end
```





第二章 进程管理

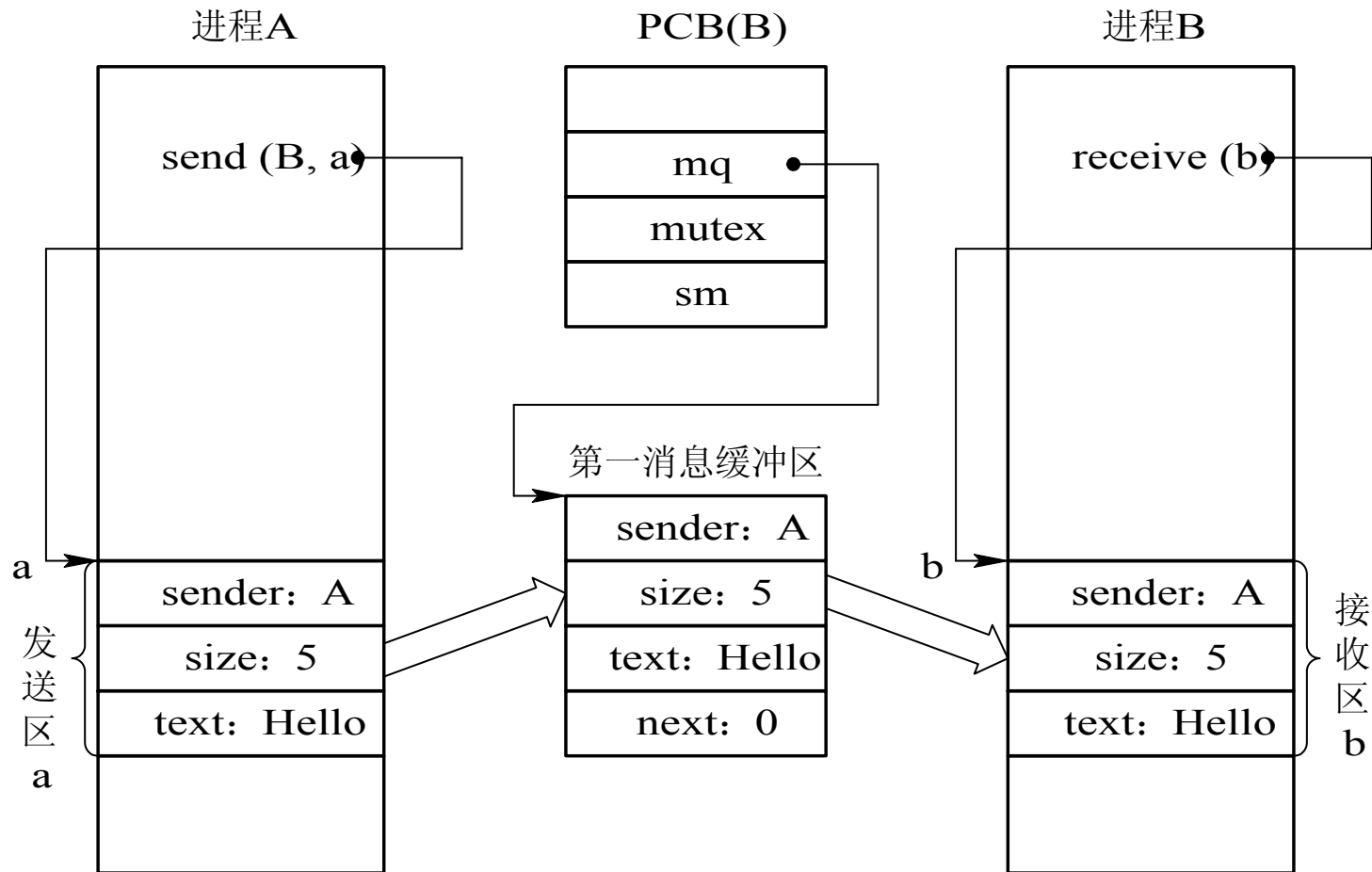


图 2-14 消息缓冲通信





3. 接收原语

接收进程调用接收原语receive(b)，从自己的消息缓冲队列mq中摘下第一个消息缓冲区i，并将其中的数据复制到以b为首址的指定消息接收区内。接收原语描述如下：





第二章 进程管理



```
procedure receive(b)
```

```
begin
```

```
    j:= internal name;    j为接收进程内部的标识符;
```

```
    wait(j.sm);
```

```
    wait(j.mutex);
```

```
    remove(j.mq, i);    将消息队列中第一个消息移出;
```

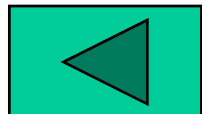
```
    signal(j.mutex);
```

```
    b.sender:=i.sender;    将消息缓冲区i中的信息复制到接收区b;
```

```
    b.size:=i.size;
```

```
    b.text:=i.text;
```

```
end
```





2.6 线程

2.6.1 线程的基本概念

1. 线程的引入

如果说，在操作系统中引入进程的目的，是为了使多个程序能并发执行，以提高资源利用率和系统吞吐量，那么，在操作系统中再引入线程，则是为了减少程序在并发执行时所付出的时空开销，使OS具有更好的并发性。为了说明这一点，我们首先来回顾进程的两个基本属性：① 进程是一个可拥有资源的独立单位；② 进程同时又是一个可独立调度和分派的基本单位。正是由于进程有这两个基本属性，才使之成为一个能独立运行的基本单位，从而也就构成了进程并发执行的基础。然而，为使程序能并发执行，系统还必须进行以下的一系列操作。





第二章 进程管理



1) 创建进程

系统在创建一个进程时，必须为它分配其所必需的、除处理机以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB。

2) 撤消进程

系统在撤消进程时，又必须先对其所占有的资源执行回收操作，然后再撤消PCB。





3) 进程切换

对进程进行切换时，由于要保留当前进程的CPU环境和设置新选中进程的CPU环境，因而须花费不少的处理机时间。

换言之，由于进程是一个资源的拥有者，因而在创建、撤消和切换中，系统必须为之付出较大的时空开销。正因如此，在系统中所设置的进程，其数目不宜过多，进程切换的频率也不宜过高，这也就限制了并发程度的进一步提高。





第二章 进程管理



如何能使多个程序更好地并发执行同时又尽量减少系统的开销，已成为近年来设计操作系统时所追求的重要目标。有不少研究操作系统的学者们想到，若能将进程的上述两个属性分开，由操作系统分开处理，亦即对于作为调度和分派的基本单位，不同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之进行频繁的切换。正是在这种思想的指导下，形成了线程的概念。





2. 线程与进程的比较

1) 调度

在传统的操作系统中，作为拥有资源的基本单位和独立调度、分派的基本单位都是进程。而在引入线程的操作系统中，则把线程作为调度和分派的基本单位，而进程作为资源拥有的基本单位，把传统进程的两个属性分开，使线程基本上不拥有资源，这样线程便能轻装前进，从而可显著地提高系统的并发程度。在同一进程中，线程的切换不会引起进程的切换，但从一个进程中的线程切换到另一个进程中的线程时，将会引起进程的切换。





2) 并发性

在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，使得操作系统具有更好的并发性，从而能更加有效地提高系统资源的利用率和系统的吞吐量。例如，在一个未引入线程的单CPU操作系统中，若仅设置一个文件服务进程，当该进程由于某种原因而被阻塞时，便没有其它的文件服务进程来提供服务。在引入线程的操作系统中，则可以在一个文件服务进程中设置多个服务线程。当第一个线程等待时，文件服务进程中的第二个线程可以继续运行，以提供文件服务；当第二个线程阻塞时，则可由第三个继续执行，提供服务。显然，这样的方法可以显著地提高文件服务的质量和系统的吞吐量。





第二章 进程管理



3) 拥有资源

不论是传统的操作系统，还是引入了线程的操作系统，进程都可以拥有资源，是系统中拥有资源的一个基本单位。一般而言，线程自己不拥有系统资源(也有一点必不可少的资源)，但它可以访问其隶属进程的资源，即一个进程的代码段、数据段及所拥有的系统资源，如已打开的文件、I/O设备等，可以供该进程中的所有线程所共享。





4) 系统开销

在创建或撤消进程时，系统都要为之创建和回收进程控制块，分配或回收资源，如内存空间和I/O设备等，操作系统所付出的开销明显大于线程创建或撤消时的开销。类似地，在进程切换时，涉及到当前进程CPU环境的保存及新被调度运行进程的CPU环境的设置，而线程的切换则仅需保存和设置少量寄存器内容，不涉及存储器管理方面的操作，所以就切换代价而言，进程也是远高于线程的。此外，由于一个进程中的多个线程具有相同的地址空间，在同步和通信的实现方面线程也比进程容易。在一些操作系统中，线程的切换、同步和通信都无须操作系统内核的干预。





3. 线程的属性

在多线程OS中，通常是在一个进程中包括多个线程，每个线程都是作为利用CPU的基本单位，是花费最小开销的实体。线程具有下述属性。

(1) 轻型实体。线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证其独立运行的资源，比如，在每个线程中都应具有一个用于控制线程运行的线程控制块TCB，用于指示被执行指令序列的程序计数器，保留局部变量、少数状态参数和返回地址等的一组寄存器和堆栈。





第二章 进程管理



(2) 独立调度和分派的基本单位。在多线程OS中，线程是能独立运行的基本单位，因而也是独立调度和分派的基本单位。由于线程很“轻”，故线程的切换非常迅速且开销小。

(3) 可并发执行。在一个进程中的多个线程之间可以并发执行，甚至允许在一个进程中的所有线程都能并发执行；同样，不同进程中的线程也能并发执行。

(4) 共享进程资源。在同一进程中的各个线程都可以共享该进程所拥有的资源，这首先表现在所有线程都具有相同的地址空间(进程的地址空间)。这意味着线程可以访问该地址空间中的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。





4. 线程的状态

(1) 状态参数。在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述。状态参数通常有这样几项：

- ① 寄存器状态，它包括程序计数器PC和堆栈指针中的内容；
- ② 堆栈，在堆栈中通常保存有局部变量和返回地址；
- ③ 线程运行状态，用于描述线程正处于何种运行状态；
- ④ 优先级，描述线程执行的优先程度；
- ⑤ 线程专有存储器，用于保存线程自己的局部变量拷贝；
- ⑥ 信号屏蔽，即对某些信号加以屏蔽。





第二章 进程管理



(2) 线程运行状态。如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时也具有下述三种基本状态：① 执行状态，表示线程正获得处理机而运行；② 就绪状态，指线程已具备了各种执行条件，一旦获得CPU便可执行的状态；③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。





5. 线程的创建和终止

在多线程OS环境下，应用程序在启动时，通常仅有一个线程在执行，该线程被人们称为“初始化线程”。它可根据需要再去创建若干个线程。在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程创建函数执行完后，将返回一个线程标识符供以后使用。





第二章 进程管理



如同进程一样，线程也是具有生命期的。终止线程的方式有两种：一种是在线程完成了自己的工作后自愿退出；另一种是线程在运行中出现错误或由于某种原因而被其它线程强行终止。但有些线程(主要是系统线程)，在它们一旦被建立起来之后，便一直运行下去而不再被终止。在大多数的OS中，线程被中止后并不立即释放它所占有的资源，只有当进程中的其它线程执行了分离函数后，被终止的线程才与资源分离，此时的资源才能被其它线程利用。





第二章 进程管理



虽已被终止但尚未释放资源的线程，仍可以被需要它的线程所调用，以使被终止线程重新恢复运行。为此，调用者线程须调用一条被称为“等待线程终止”的连接命令，来与该线程进行连接。如果在一个调用者线程调用“等待线程终止”的连接命令试图与指定线程相连接时，若指定线程尚未被终止，则调用连接命令的线程将会阻塞，直至指定线程被终止后才能实现它与调用者线程的连接并继续执行；若指定线程已被终止，则调用者线程不会被阻塞而是继续执行。





6. 多线程OS中的进程

在多线程OS中，进程是作为拥有系统资源的基本单位，通常的进程都包含多个线程并为它们提供资源，但此时的进程就不再作为一个执行的实体。多线程OS中的进程有以下属性：

(1) 作为系统资源分配的单位。在多线程OS中，仍是将进程作为系统资源分配的基本单位，在任一进程中所拥有的资源包括受到分别保护的用户地址空间、用于实现进程间和线程间同步和通信的机制、已打开的文件和已申请到的I/O设备，以及一张由核心进程维护的地址映射表，该表用于实现用户程序的逻辑地址到其内存物理地址的映射。





第二章 进程管理



(2) 可包括多个线程。通常，一个进程都含有多个相对独立的线程，其数目可多可少，但至少也要有一个线程，由进程为这些(个)线程提供资源及运行环境，使这些线程可并发执行。在OS中的所有线程都只能属于某一个特定进程。

(3) 进程不是一个可执行的实体。在多线程OS中，是把线程作为独立运行的基本单位，所以此时的进程已不再是一个可执行的实体。虽然如此，进程仍具有与执行相关的状态。例如，所谓进程处于“执行”状态，实际上是指该进程中的某线程正在执行。此外，对进程所施加的与进程状态有关的操作，也对其线程起作用。例如，在把某个进程挂起时，该进程中的所有线程也都将被挂起；又如，在把某进程激活时，属于该进程的所有线程也都将被激活。





2.6.2 线程间的同步和通信

1. 互斥锁(mutex)

互斥锁是一种比较简单的、用于实现线程间对资源互斥访问的机制。由于操作互斥锁的时间和空间开销都较低，因而较适合于高频度使用的关键共享数据和程序段。互斥锁可以有两种状态，即开锁(unlock)和关锁(lock)状态。相应地，可用两条命令(函数)对互斥锁进行操作。其中的关锁lock操作用于将mutex关上，开锁操作unlock则用于打开mutex。





第二章 进程管理



当一个线程需要读/写一个共享数据段时，线程首先应为此数据段所设置的mutex执行关锁命令。命令首先判别mutex的状态，如果它已处于关锁状态，则试图访问该数据段的线程将被阻塞；而如果mutex是处于开锁状态，则将mutex关上后便去读/写该数据段。在线程完成对数据的读/写后，必须再发出开锁命令将mutex打开，同时还须将阻塞在该互斥锁上的一个线程唤醒，其它的线程仍被阻塞在等待mutex打开的队列上。

另外，为了减少线程被阻塞的机会，在有的系统中还提供了一种用于mutex上的操作命令Trylock。当一个线程在利用Trylock命令去访问mutex时，若mutex处于开锁状态，Trylock将返回一个指示成功的状态码；反之，若mutex处于关锁状态，则Trylock并不会阻塞该线程，而只是返回一个指示操作失败的状态码。





2. 条件变量

在许多情况下，只利用mutex来实现互斥访问可能会引起死锁，我们通过一个例子来说明这一点。有一个线程在对mutex 1执行关锁操作成功后，便进入一临界区C，若在临界区内该线程又须访问某个临界资源R，同样也为R设置另一互斥锁mutex 2。假如资源R此时正处于忙碌状态，线程在对mutex 2执行关锁操作后必将被阻塞，这样将使mutex 1一直保持关锁状态；如果保持了资源R的线程也要求进入临界区C，但由于mutex 1一直保持关锁状态而无法进入临界区，这样便形成了死锁。为了解决这个问题便引入了条件变量。





第二章 进程管理



每一个条件变量通常都与一个互斥锁一起使用，亦即，在创建一个互斥锁时便联系着一个条件变量。单纯的互斥锁用于短期锁定，主要是用来保证对临界区的互斥进入。而条件变量则用于线程的长期等待，直至所等待的资源成为可用的资源。

现在，我们看看如何利用互斥锁和条件变量来实现对资源R的访问。线程首先对mutex执行关锁操作，若成功便进入临界区，然后查找用于描述该资源状态的数据结构，以了解资源的情况。只要发现所需资源R正处于忙碌状态，线程便转为等待状态，并对mutex执行开锁操作后，等待该资源被释放；若资源处于空闲状态，表明线程可以使用该资源，于是将该资源设置为忙碌状态，再对mutex执行开锁操作。下面给出了对上述资源的申请(左半部分)和释放(右半部分)操作的描述。





第二章 进程管理



Lock mutex

check data structures;

while(resource busy);

wait(condition variable);

mark resource as busy;

unlock mutex;

Lock mutex

mark resource as free;

unlock mutex;

wakeup(condition variable);





第二章 进程管理



原来占有资源R的线程在使用完该资源后，便按照右半部分的描述释放该资源，其中的wakeup(condition variable)表示去唤醒在指定条件变量上等待的一个或多个线程。在大多数情况下，由于所释放的是临界资源，此时所唤醒的只能是在条件变量上等待的某一个线程，其它线程仍继续在该队列上等待。但如果线程所释放的是一个数据文件，该文件允许多个线程同时对它执行读操作。在这种情况下，当一个写线程完成写操作并释放该文件后，如果此时在该条件变量上还有多个读线程在等待，则该线程可以唤醒所有的等待线程。





3. 信号量机制

1) 私用信号量(private semaphore)

当某线程需利用信号量来实现同一进程中各线程之间的同步时，可调用创建信号量的命令来创建一私用信号量，其数据结构存放在应用程序的地址空间中。私用信号量属于特定的进程所有，OS并不知道私用信号量的存在，因此，一旦发生私用信号量的占用者异常结束或正常结束，但并未释放该信号量所占有空间的情况时，系统将无法使它恢复为0(空)，也不能将它传送给下一个请求它的线程。





2) 公用信号量(public semaphore)

公用信号量是为实现不同进程间或不同进程中各线程之间的同步而设置的。由于它有着一个公开的名字供所有的进程使用，故而把它称为公用信号量。其数据结构是存放在受保护的系统存储区中，由OS为它分配空间并进行管理，故也称为系统信号量。如果信号量的占有者在结束时未释放该公用信号量，则OS会自动将该信号量空间回收，并通知下一进程。可见，公用信号量是一种比较安全的同步机制。





2.6.3 线程的实现方式

1. 内核支持线程

对于通常的进程，无论是系统进程还是用户进程，进程的创建、撤消，以及要求由系统设备完成的I/O操作，都是利用系统调用而进入内核，再由内核中的相应处理程序予以完成的。进程的切换同样是在内核的支持下实现的。因此我们说，不论什么进程，它们都是在操作系统内核的支持下运行的，是与内核紧密相关的。





第二章 进程管理



这里所谓的内核支持线程KST(Kernel Supported Threads), 也都同样是在内核的支持下运行的, 即无论是用户进程中的线程, 还是系统进程中的线程, 他们的创建、撤消和切换等也是依靠内核, 在内核空间实现的。此外, 在内核空间还为每一个内核支持线程设置了一个线程控制块, 内核是根据该控制块而感知某线程的存在, 并对其加以控制。

这种线程实现方式主要有如下四个优点:

(1) 在多处理器系统中, 内核能够同时调度同一进程中多个线程并行执行;





第二章 进程管理



(2) 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程；

(3) 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小；

(4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。

内核支持线程的主要缺点是：对于用户的线程切换而言，其模式切换的开销较大，在同一个进程中，从一个线程切换到另一个线程时，需要从用户态转到内核态进行，这是因为用户进程的线程在用户态运行，而线程调度和管理是在内核实现的，系统开销较大。





2. 用户级线程

用户级线程ULT(User Level Threads)仅存在于用户空间中。对于这种线程的创建、撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现。对于用户级线程的切换，通常发生在一个应用进程的诸多线程之间，这时，也同样无须内核的支持。由于切换的规则远比进程调度和切换的规则简单，因而使线程的切换速度特别快。可见，这种线程是与内核无关的。我们可以为一个应用程序建立多个用户级线程。在一个系统中的用户级线程的数目可以达到数百个至数千个。由于这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无须内核的帮助，因而内核完全不知道用户级线程的存在。





第二章 进程管理



值得说明的是，对于设置了用户级线程的系统，其调度仍是以进程为单位进行的。在采用轮转调度算法时，各个进程轮流执行一个时间片，这对诸进程而言似乎是公平的。但假如在进程A中包含了一个用户级线程，而在另一个进程B中含有100个用户级线程，这样，进程A中线程的运行时间将是进程B中各线程运行时间的100倍；相应地，其速度要快上100倍。

假如系统中设置的是内核支持线程，则调度便是以线程为单位进行的。在采用轮转法调度时，是各个线程轮流执行一个时间片。同样假定进程A中只有一个内核支持线程，而在进程B中有100个内核支持线程。此时进程B可以获得的CPU时间是进程A的100倍，且进程B可使100个系统调用并发工作。





第二章 进程管理



使用用户级线程方式有许多优点，主要表现在如下三个方面：

(1) 线程切换不需要转换到内核空间，对于一个进程而言，其所有线程的管理数据结构均在该进程的用户空间中，管理线程切换的线程库也在用户地址空间运行。因此，进程不必切换到内核方式来做线程管理，从而节省了模式切换的开销，也节省了内核的宝贵资源。

(2) 调度算法可以是进程专用的。在不干扰操作系统调度的情况下，不同的进程可以根据自身需要，选择不同的调度算法对自己的线程进行管理和调度，而与操作系统的低级调度算法是无关的。





第二章 进程管理



(3) 用户级线程的实现与操作系统平台无关，因为对于线程管理的代码是在用户程序内的，属于用户程序的一部分，所有的应用程序都可以对之进行共享。因此，用户级线程甚至可以在不支持线程机制的操作系统平台上实现。





第二章 进程管理



用户级线程实现方式的主要缺点在于如下两个方面：

(1) 系统调用的阻塞问题。在基于进程机制的操作系统中，大多数系统调用将阻塞进程，因此，当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都会被阻塞。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。

(2) 在单纯的用户级线程实现方式中，多线程应用不能利用多处理机进行多重处理的优点。内核每次分配给一个进程的仅有一个CPU，因此进程中仅有一个线程能执行，在该线程放弃CPU之前，其它线程只能等待。





3. 组合方式

有些操作系统把用户级线程和内核支持线程两种方式进行组合，提供了组合方式ULT/KST线程。在组合方式线程系统中，内核支持多KST线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程。一些内核支持线程对应多个用户级线程，程序员可按应用需要和机器配置对内核支持线程数目进行调整，以达到较好的效果。组合方式线程中，同一个进程内的多个线程可以同时有多处理器上并行执行，而且在阻塞一个线程时，并不需要将整个进程阻塞。所以，组合方式多线程机制能够结合KST和ULT两者的优点，并克服了其各自的不足。





2.6.4 线程的实现

1. 内核支持线程的实现

在仅设置了内核支持线程的OS中，一种可能的线程控制方法是，系统在创建一个新进程时，便为它分配一个任务数据区PTDA(Per Task Data Area)，其中包括若干个线程控制块TCB空间，如图2-15所示。在每一个TCB中可保存线程标识符、优先级、线程运行的CPU状态等信息。虽然这些信息与用户级线程TCB中的信息相同，但现在却是被保存在内核空间中。





第二章 进程管理



PTDA 进程资源

TCB #1

TCB #2

TCB #3

图 2-15 任务数据区空间





第二章 进程管理



每当进程要创建一个线程时，便为新线程分配一个TCB，将有关信息填入该TCB中，并为之分配必要的资源，如为线程分配数百至数千个字节的栈空间和局部存储区，于是新创建的线程便有条件立即执行。当PTDA中的所有TCB空间已用完，而进程又要创建新的线程时，只要其所创建的线程数目未超过系统的允许值(通常为数十至数百个)，系统可再为之分配新的TCB空间；在撤消一个线程时，也应回收该线程的所有资源和TCB。可见，内核支持线程的创建、撤消均与进程的相类似。在有的系统中为了减少创建和撤消一个线程时的开销，在撤消一个线程时，并不立即回收该线程的资源和TCB，当以后再要创建一个新线程时，便可直接利用已被撤消但仍保持有资源和TCB的线程作为新线程。





第二章 进程管理



内核支持线程的调度和切换与进程的调度和切换十分相似，也分抢占式方式和非抢占式方式两种。在线程的调度算法上，同样可采用时间片轮转法、优先权算法等。当线程调度选中一个线程后，便将处理机分配给它。当然，线程在调度和切换上所花费的开销，要比进程的小得多。





2. 用户级线程的实现

用户级线程是在用户空间实现的。所有的用户级线程都具有相同的结构，它们都运行在一个中间系统的上面。当前有两种方式实现的中间系统，即运行时系统和内核控制线程。

1) 运行时系统(Runtime System)

所谓“运行时系统”，实质上是用于管理和控制线程的函数(过程)的集合，其中包括用于创建和撤消线程的函数、线程同步和通信的函数以及实现线程调度的函数等。正因为有这些函数，才能使用户级线程与内核无关。运行时系统中的所有函数都驻留在用户空间，并作为用户级线程与内核之间的接口。





第二章 进程管理



在传统的OS中，进程在切换时必须先由用户态转为核心态，再由核心来执行切换任务；而用户级线程在切换时则不需转入核心态，而是由运行时系统中的线程切换过程来执行切换任务。该过程将线程的CPU状态保存在该线程的堆栈中，然后按照一定的算法选择一个处于就绪状态的新线程运行，将新线程堆栈中的CPU状态装入到CPU相应的寄存器中，一旦将栈指针和程序计数器切换后，便开始了新线程的运行。由于用户级线程的切换无需进入内核，且切换操作简单，因而使用户级线程的切换速度非常快。





第二章 进程管理



不论在传统的OS中，还是在多线程OS中，系统资源都是由内核管理的。在传统的OS中，进程是利用OS提供的系统调用来请求系统资源的，系统调用通过软中断(如trap)机制进入OS内核，由内核来完成相应资源的分配。用户级线程是不能利用系统调用的。当线程需要系统资源时，是将该要求传送给运行时系统，由后者通过相应的系统调用来获得系统资源的。





2) 内核控制线程

这种线程又称为轻型进程LWP(Light Weight Process)。每一个进程都可拥有多个LWP，同用户级线程一样，每个LWP都有自己的数据结构(如TCB)，其中包括线程标识符、优先级、状态，另外还有栈和局部存储区等。它们也可以共享进程所拥有的资源。LWP可通过系统调用来获得内核提供的服务，这样，当一个用户级线程运行时，只要将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。这种线程实现方式就是组合方式。





第二章 进程管理



在一个系统中的用户级线程数量可能很大，为了节省系统开销，不可能设置太多的LWP，而把这些LWP做成一个缓冲池，称为“线程池”。用户进程中的任一用户线程都可以连接到LWP池中的任何一个LWP上。为使每一用户级线程都能利用LWP与内核通信，可以使多个用户级线程多路复用同一个LWP，但只有当前连接到LWP上的线程才能与内核通信，其余进程或者阻塞，或者等待LWP。而每一个LWP都要连接到一个内核级线程上，这样，通过LWP可把用户级线程与内核线程连接起来，用户级线程可通过LWP来访问内核，但内核所看到的总是多个LWP而看不到用户级线程。亦即，由LWP实现了在内核与用户级线程之间的隔离，从而使用户级线程与内核无关。图2-16示出了利用轻型进程作为中间系统时用户级线程的实现方法。





第二章 进程管理

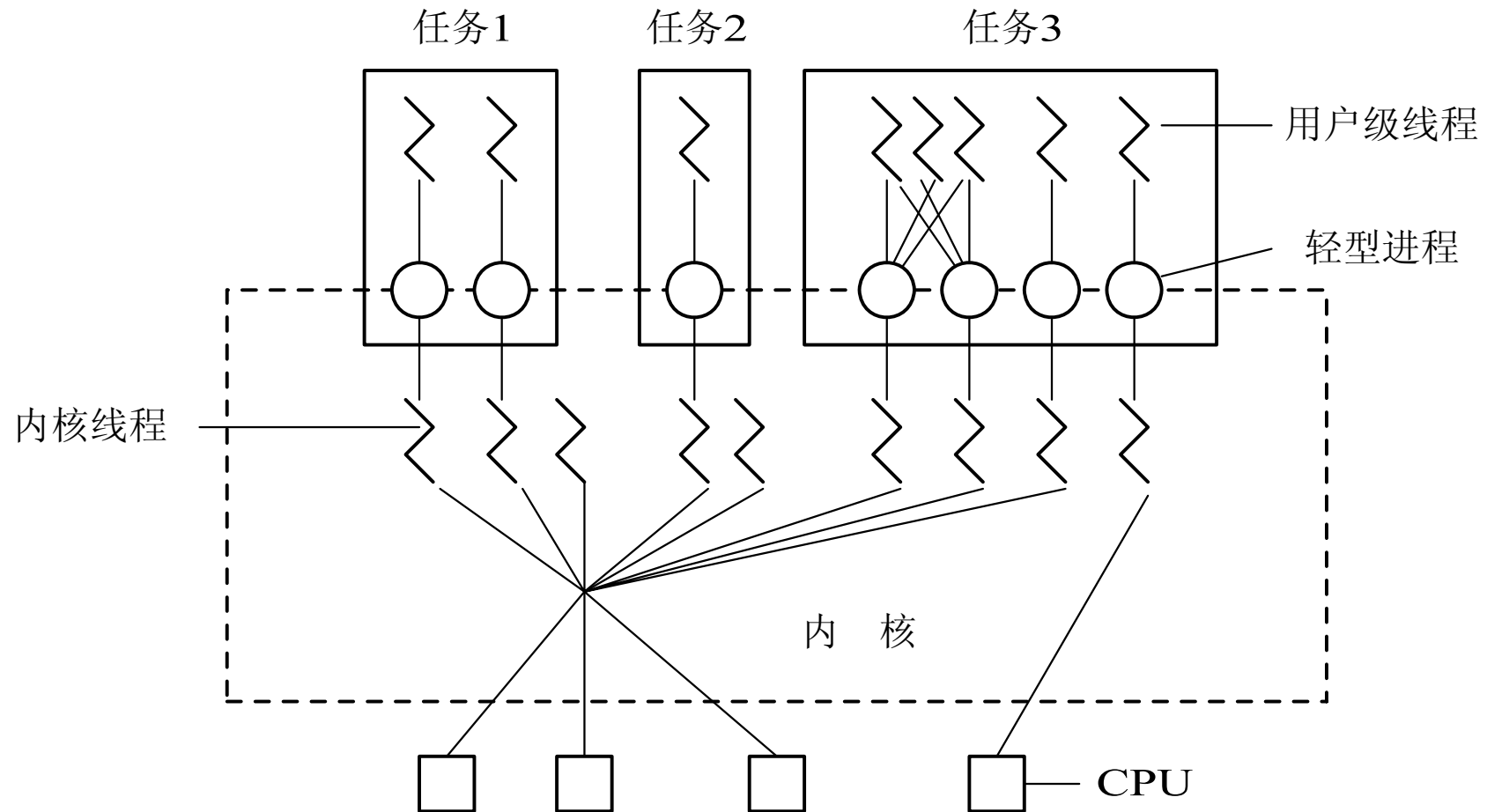


图 2-16 利用轻型进程作为中间系统





第二章 进程管理



当用户级线程不需要与内核通信时，并不需要LWP；而当要通信时，便需借助于LWP，而且每个要通信的用户级线程都需要一个LWP。例如，在一个任务中，如果同时有5个用户级线程发出了对文件的读、写请求，这就需要有5个LWP来予以帮助，即由LWP将对文件的读、写请求发送给相应的内核级线程，再由后者执行具体的读、写操作。如果一个任务中只有4个LWP，则只能有4个用户级线程的读、写请求被传送给内核线程，余下的一个用户级线程必须等待。





第二章 进程管理



在内核级线程执行操作时，如果发生阻塞，则与之相连接的多个LWP也将随之阻塞，进而使连接到LWP上的用户级线程也被阻塞。如果进程中只包含了一个LWP，此时进程也应阻塞。这种情况与前述的传统OS一样，在进程执行系统调用时，该进程实际上是阻塞的。但如果在一个进程中含有多个LWP，则当一个LWP阻塞时，进程中的另一个LWP可继续执行；即使进程中的所有LWP全部阻塞，进程中的线程也仍然能继续执行，只是不能再去访问内核。





3. 用户级线程与内核控制线程的连接

1) 一对一模型

该模型是为每一个用户线程都设置一个内核控制线程与之连接，当一个线程阻塞时，允许调度另一个线程运行。在多处理机系统中，则有多个线程并行执行。

该模型并行能力较强，但每创建一个用户线程相应地就需要创建一个内核线程，开销较大，因此需要限制整个系统的线程数。Windows 2000、Windows NT、OS/2等系统上都实现了该模型。





2) 多对一模型

该模型是将多个用户线程映射到一个内核控制线程，为了管理方便，这些用户线程一般属于一个进程，运行在该进程的用户空间，对这些线程的调度和管理也是在该进程的用户空间中完成。当用户线程需要访问内核时，才将其映射到一个内核控制线程上，但每次只允许一个线程进行映射。

该模型的主要优点是线程管理的开销小，效率高，但当—一个线程在访问内核时发生阻塞，则整个进程都会被阻塞，而且在多处理机系统中，一个进程的多个线程无法实现并行。



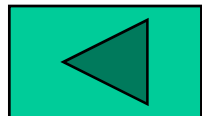


第二章 进程管理



3) 多对多模型

该模型结合上述两种模型的优点，将多个用户线程映射到多个内核控制线程，内核控制线程的数目可以根据应用进程和系统的不同而变化，可以比用户线程少，也可以与之相同。





第三章 处理机调度与死锁

3.1 处理机调度的层次

3.2 调度队列模型和调度准则

3.3 调度算法

3.4 实时调度

3.5 产生死锁的原因和必要条件

3.6 预防死锁的方法

3.7 死锁的检测与解除





3.1 处理机调度的层次

3.1.1 高级调度

1. 作业和作业步

(1) 作业(Job)。作业是一个比程序更为广泛的概念，它不仅包含了通常的程序和数据，而且还应配有一份作业说明书，系统根据该说明书来对程序的运行进行控制。在批处理系统中，是以作业为基本单位从外存调入内存的。





(2) 作业步(Job Step)。通常，在作业运行期间，每个作业都必须经过若干个相对独立，又相互关联的顺序加工步骤才能得到结果，我们把其中的每一个加工步骤称为一个作业步，各作业步之间存在着相互联系，往往是把上一个作业步的输出作为下一个作业步的输入。例如，一个典型的作业可分成三个作业步：① “编译”作业步，通过执行编译程序对源程序进行编译，产生若干个目标程序段；② “连结装配”作业步，将“编译”作业步所产生的若干个目标程序段装配成可执行的目标程序；③ “运行”作业步，将可执行的目标程序读入内存并控制其运行。

(3) 作业流。若干个作业进入系统后，被依次存放在外存上，这便形成了输入的作业流；在操作系统的控制下，逐个作业进行处理，于是便形成了处理作业流。





2. 作业控制块JCB(Job Control Block)

为了管理和调度作业，在多道批处理系统中为每个作业设置了一个作业控制块，如同进程控制块是进程在系统中存在的标志一样，它是作业在系统中存在的标志，其中保存了系统对作业进行管理和调度所需的全部信息。在JCB中所包含的内容因系统而异，通常应包含的内容有：作业标识、用户名称、用户帐户、作业类型(CPU 繁忙型、I/O 繁忙型、批量型、终端型)、作业状态、调度信息(优先级、作业已运行时间)、资源需求(预计运行时间、要求内存大小、要求I/O设备的类型和数量等)、进入系统时间、开始处理时间、作业完成时间、作业退出时间、资源使用情况等。





第三章 处理机调度与死锁



每当作业进入系统时，系统便为每个作业建立一个JCB，根据作业类型将它插入相应的后备队列中。作业调度程序依据一定的调度算法来调度它们，被调度到的作业将会装入内存。在作业运行期间，系统就按照JCB中的信息对作业进行控制。当一个作业执行结束进入完成状态时，系统负责回收分配给它的资源，撤消它的作业控制块。





3. 作业调度

作业调度的主要功能是根据作业控制块中的信息，审查系统能否满足用户作业的资源需求，以及按照一定的算法，从外存的后备队列中选取某些作业调入内存，并为它们创建进程、分配必要的资源。然后再将新创建的进程插入就绪队列，准备执行。因此，有时也把作业调度称为接纳调度 (Admission Scheduling)。





第三章 处理机调度与死锁



对用户而言，总希望自己作业的周转时间尽可能的少，最好周转时间就等于作业的执行时间。然而对系统来说，则希望作业的平均周转时间尽可能少，有利于提高CPU 的利用率和系统的吞吐量。为此，每个系统在选择作业调度算法时，既应考虑用户的要求，又能确保系统具有较高的效率。在每次执行作业调度时，都须做出以下两个决定。





1) 决定接纳多少个作业

作业调度每次要接纳多少个作业进入内存，取决于多道程序度(Degree of Multiprogramming)，即允许多少个作业同时在内存中运行。当内存中同时运行的作业数目太多时，可能会影响到系统的服务质量，比如，使周转时间太长。但如果在内存中同时运行作业的数量太少时，又会导致系统的资源利用率和系统吞吐量太低，因此，多道程序度的确定应根据系统的规模和运行速度等情况做适当的折衷。





2) 决定接纳哪些作业

应将哪些作业从外存调入内存，这将取决于所采用的调度算法。最简单的是先来先服务调度算法，这是指将最早进入外存的作业最先调入内存；较常用的一种算法是短作业优先调度算法，是将外存上最短的作业最先调入内存；另一种较常用的是基于作业优先级的调度算法，该算法是将外存上优先级最高的作业优先调入内存；比较好的一种算法是“响应比高者优先”的调度算法。我们将在后面对上述几种算法作较为详细的介绍。





第三章 处理机调度与死锁



在批处理系统中，作业进入系统后，总是先驻留在外存的后备队列上，因此需要有作业调度的过程，以便将它们分批地装入内存。然而在分时系统中，为了做到及时响应，用户通过键盘输入的命令或数据等都是被直接送入内存的，因而无需再配置上述的作业调度机制，但也需要有某些限制性措施来限制进入系统的用户数。即，如果系统尚未饱和，将接纳所有授权用户，否则，将拒绝接纳。类似地，在实时系统中通常也不需要作业调度。





3.1.2 低级调度

通常也把低级调度 (Low Level Scheduling) 称为进程调度或短程调度 (ShortTerm Scheduling)，它所调度的对象是进程 (或内核级线程)。进程调度是最基本的一种调度，在多道批处理、分时和实时三种类型的OS中，都必须配置这级调度。

1. 低级调度的功能

低级调度用于决定就绪队列中的哪个进程 (或内核级线程，为叙述方便，以后只写进程) 应获得处理机，然后再由分派程序执行把处理机分配给该进程的具体操作。





第三章 处理机调度与死锁



低级调度的主要功能如下：

(1) 保存处理机的现场信息。在进程调度进行调度时，首先需要保存当前进程的处理机的现场信息，如程序计数器、多个通用寄存器中的内容等，将它们送入该进程的进程控制块(PCB)中的相应单元。

(2) 按某种算法选取进程。低级调度程序按某种算法如优先数算法、轮转法等，从就绪队列中选取一个进程，把它的状态改为运行状态，并准备把处理机分配给它。

(3) 把处理器分配给进程。由分派程序(Dispatcher)把处理器分配给进程。此时需为选中的进程恢复处理机现场，即把选中进程的进程控制块内有关处理机现场的信息装入处理器相应的各个寄存器中，把处理器的控制权交给该进程，让它从取出的断点处开始继续运行。





2. 进程调度中的三个基本机制

为了实现进程调度，应具有如下三个基本机制：

(1) 排队器。为了提高进程调度的效率，应事先将系统中所有的就绪进程按照一定的方式排成一个或多个队列，以便调度程序能最快地找到它。

(2) 分派器(分派程序)。分派器把由进程调度程序所选定的进程，从就绪队列中取出该进程，然后进行上下文切换，将处理机分配给它

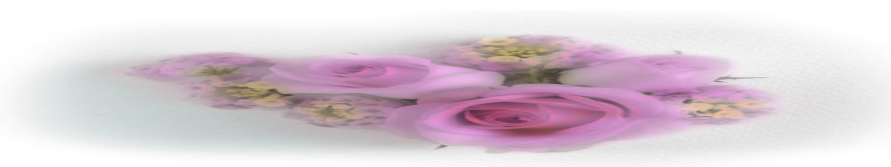




第三章 处理机调度与死锁



(3) 上下文切换机制。当对处理机进行切换时，会发生两对上下文切换操作。在第一对上下文切换时，操作系统将保存当前进程的上下文，而装入分派程序的上下文，以便分派程序运行；在第二对上下文切换时，将移出分派程序，而把新选进程的CPU现场信息装入到处理机的各个相应寄存器中。





第三章 处理机调度与死锁



应当指出，上下文切换将花去不少的处理机时间，即使是现代计算机，每一次上下文切换大约需要花费几毫秒的时间，该时间大约可执行上千条指令。为此，现在已有通过硬件(采用两组或多组寄存器)的方法来减少上下文切换的时间。一组寄存器供处理机在系统态时使用，另一组寄存器供应用程序使用。在这种条件下的上下文切换只需改变指针，使其指向当前寄存器组即可。





3. 进程调度方式

进程调度可采用下述两种调度方式。

1) 非抢占方式 (Nonpreemptive Mode)

在采用这种调度方式时，一旦把处理机分配给某进程后，不管它要运行多长时间，都一直让它运行下去，决不会因为时钟中断等原因而抢占正在运行进程的处理机，也不允许其它进程抢占已经分配给它的处理机。直至该进程完成，自愿释放处理机，或发生某事件而被阻塞时，才再把处理机分配给其他进程。





第三章 处理机调度与死锁



在采用非抢占调度方式时，可能引起进程调度的因素可归结为如下几个：

- (1) 正在执行的进程执行完毕，或因发生某事件而不能再继续执行；
- (2) 执行中的进程因提出I/O请求而暂停执行；
- (3) 在进程通信或同步过程中执行了某种原语操作，如P操作(wait操作)、Block原语、Wakeup原语等。

这种调度方式的优点是实现简单，系统开销小，适用于大多数的批处理系统环境。但它难以满足紧急任务的要求——立即执行，因而可能造成难以预料的后果。显然，在要求比较严格的实时系统中，不宜采用这种调度方式。





2) 抢占方式(Preemptive Mode)

这种调度方式允许调度程序根据某种原则去暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程。抢占方式的优点是，可以防止一个长进程长时间占用处理机，能为大多数进程提供更公平的服务，特别是能满足对响应时间有着较严格要求的实时任务的需求。但抢占方式比非抢占方式调度所需付出的开销较大。抢占调度方式是基于一定原则的，主要有如下几条：





第三章 处理机调度与死锁



(1) 优先权原则。通常是对一些重要的和紧急的作业赋予较高的优先权。当这种作业到达时，如果其优先权比正在执行进程的优先权高，便停止正在执行(当前)的进程，将处理机分配给优先权高的新到的进程，使之执行；或者说，允许优先权高的新到进程抢占当前进程的处理机。

(2) 短作业(进程)优先原则。当新到达的作业(进程)比正在执行的作业(进程)明显的短时，将暂停当前长作业(进程)的执行，将处理机分配给新到的短作业(进程)，使之优先执行；或者说，短作业(进程)可以抢占当前较长作业(进程)的处理机。

(3) 时间片原则。各进程按时间片轮流运行，当一个时间片用完后，便停止该进程的执行而重新进行调度。这种原则适用于分时系统、大多数的实时系统，以及要求较高的批处理系统。





3.1.3 中级调度

中级调度(Intermediate Level Scheduling)又称中程调度(Medium-Term Scheduling)。引入中级调度的主要目的是为了提高内存利用率和系统吞吐量。为此,应使那些暂时不能运行的进程不再占用宝贵的内存资源,而将它们调至外存上去等待,把此时的进程状态称为就绪驻外存状态或挂起状态。当这些进程重又具备运行条件且内存又稍有空闲时,由中级调度来决定把外存上的那些又具备运行条件的就绪进程重新调入内存,并修改其状态为就绪状态,挂在就绪队列上等待进程调度。中级调度实际上就是存储器管理中的对换功能,我们将在第四章中做详细阐述。

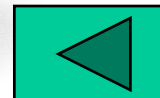




第三章 处理机调度与死锁



在上述三种调度中，进程调度的运行频率最高，在分时系统中通常是10~100 ms便进行一次进程调度，因此把它称为短程调度。为避免进程调度占用太多的CPU时间，进程调度算法不宜太复杂。作业调度往往是发生在一个(批)作业运行完毕，退出系统，而需要重新调入一个(批)作业进入内存时，故作业调度的周期较长，大约几分钟一次，因此把它称为长程调度。由于其运行频率较低，故允许作业调度算法花费较多的时间。中级调度的运行频率基本上介于上述两种调度之间，因此把它称为中程调度。





3.2 调度队列模型和调度准则

3.2.1 调度队列模型

1. 仅有进程调度的调度队列模型

在分时系统中，通常仅设置了进程调度，用户键入的命令和数据都直接送入内存。对于命令，是由OS为之建立一个进程。系统可以把处于就绪状态的进程组织成栈、树或一个无序链表，至于到底采用其中哪种形式，则与OS类型和所采用的调度算法有关。例如，在分时系统中，常把就绪进程组织成FIFO队列形式。每当OS创建一个新进程时，便将它挂在就绪队列的末尾，然后按时间片轮转方式运行。





每个进程在执行时都可能出现以下三种情况：

- (1) 任务在给定的时间片内已经完成，该进程便在释放处理机后进入完成状态；
- (2) 任务在本次分得的时间片内尚未完成，OS便将该任务再放入就绪队列的末尾；
- (3) 在执行期间，进程因为某事件而被阻塞后，被OS放入阻塞队列。

图3-1示出了仅具有进程调度的调度队列模型。





第三章 处理机调度与死锁

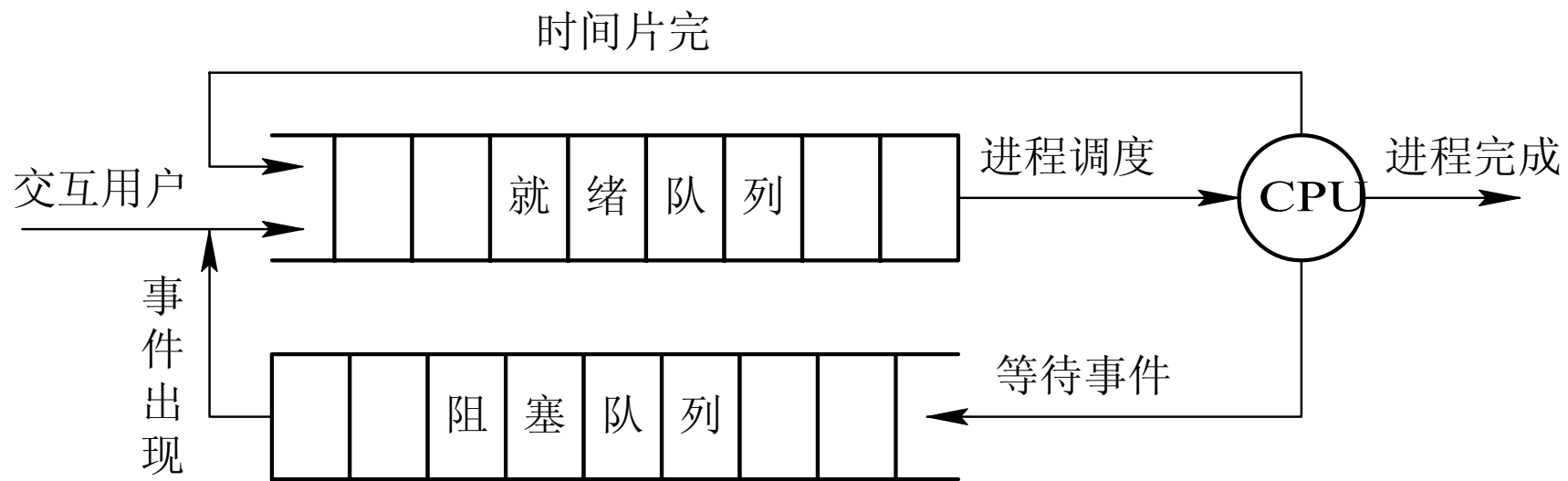


图 3-1 仅具有进程调度的调度队列模型





2. 具有高级和低级调度的调度队列模型

在批处理系统中，不仅需要进程调度，而且还需有作业调度，由后者按一定的作业调度算法，从外存的后备队列中选择一批作业调入内存，并为它们建立进程，送入就绪队列，然后才由进程调度按照一定的进程调度算法选择一个进程，把处理机分配给该进程。图3-2示出了具有高、低两级调度的调度队列模型。该模型与上一模型的主要区别在于如下两个方面。





(1) 就绪队列的形式。在批处理系统中，最常用的是最高优先权优先调度算法，相应地，最常用的就绪队列形式是优先权队列。进程在进入优先级队列时，根据其优先权的高低，被插入具有相应优先权的位置上，这样，调度程序总是把处理机分配给就绪队列中的队首进程。在最高优先权优先的调度算法中，也可采用无序链表方式，即每次把新到的进程挂在链尾，而调度程序每次调度时，是依次比较该链中各进程的优先权，从中找出优先权最高的进程，将之从链中摘下，并把处理机分配给它。显然，无序链表方式与优先权队列相比，这种方式的调度效率较低。





第三章 处理机调度与死锁

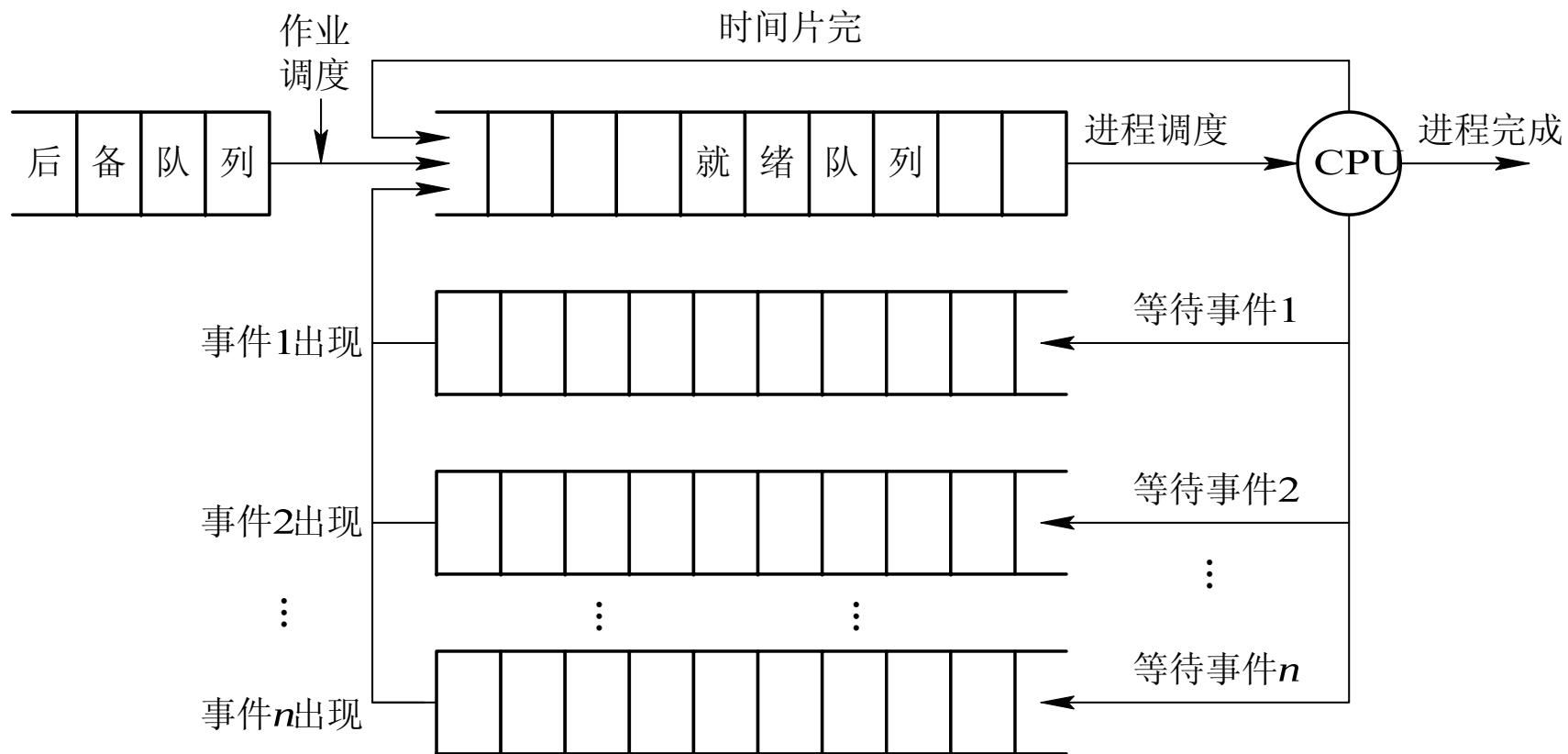


图 3-2 具有高、低两级调度的调度队列模型



(2) 设置多个阻塞队列。对于小型系统，可以只设置一个阻塞队列；但当系统较大时，若仍只有一个阻塞队列，其长度必然会很长，队列中的进程数可以达到数百个，这将严重影响对阻塞队列操作的效率。故在大、中型系统中通常都设置了若干个阻塞队列，每个队列对应于某一种进程阻塞事件。





3. 同时具有三级调度的调度队列模型

当在OS中引入中级调度后，人们可把进程的就绪状态分为内存就绪(表示进程在内存中就绪)和外存就绪(进程在外存中就绪)。类似地，也可把阻塞状态进一步分成内存阻塞和外存阻塞两种状态。在调出操作的作用下，可使进程状态由内存就绪转为外存就绪，由内存阻塞转为外存阻塞；在中级调度的作用下，又可使外存就绪转为内存就绪。图3-3示出了具有三级调度的调度队列模型。





第三章 处理机调度与死锁

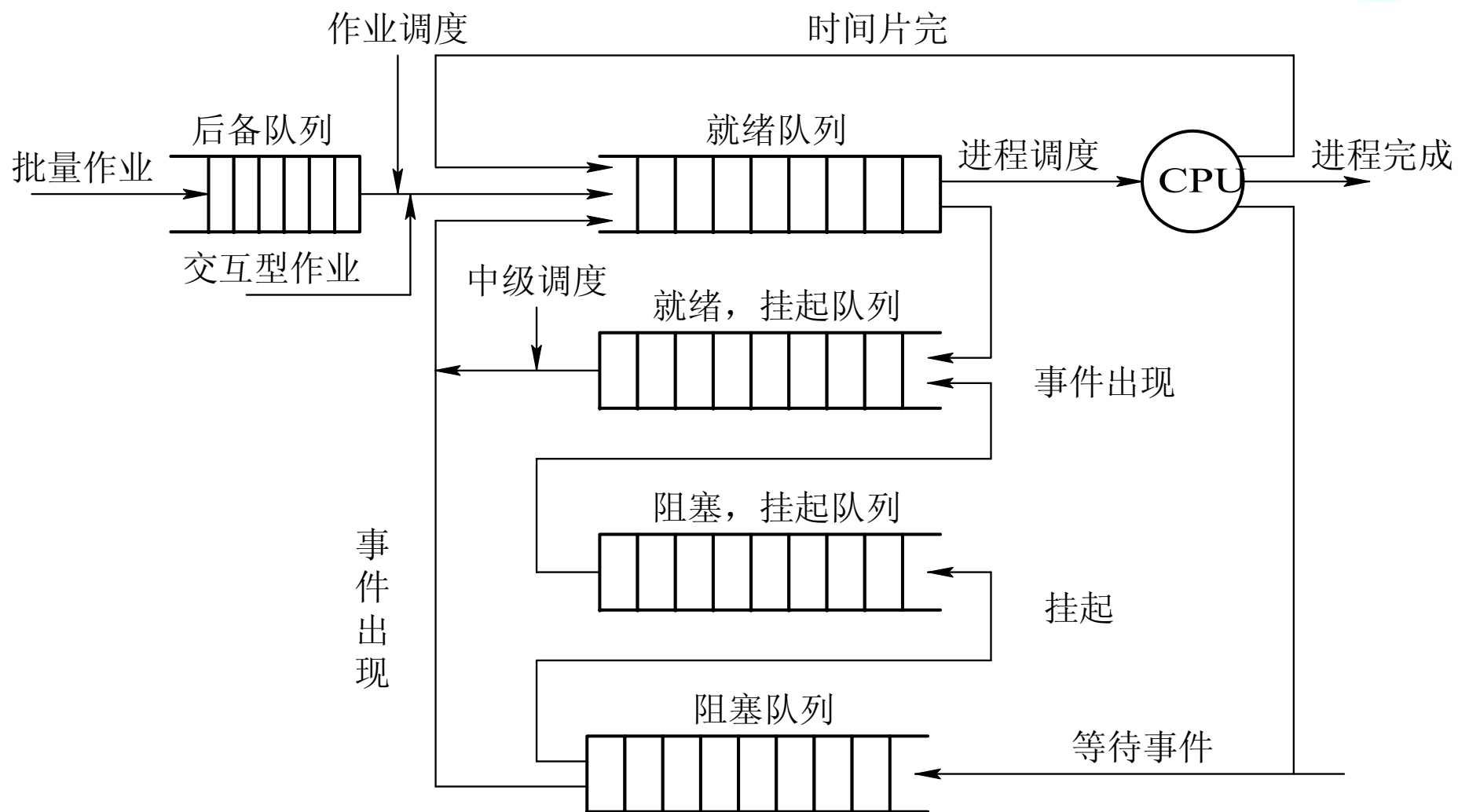


图 3-3 具有三级调度时的调度队列模型



3.2.2 选择调度方式和调度算法的若干准则

1. 面向用户的准则

(1) 周转时间短。通常把周转时间的长短作为评价批处理系统的性能、选择作业调度方式与算法的重要准则之一。所谓周转时间，是指从作业被提交给系统开始，到作业完成为止的这段时间间隔(称为作业周转时间)。它包括四部分时间：作业在外存后备队列上等待(作业)调度的时间，进程在就绪队列上等待进程调度的时间，进程在CPU上执行的时间，以及进程等待I/O操作完成的时间。其中的后三项在一个作业的整个处理过程中可能会发生多次。



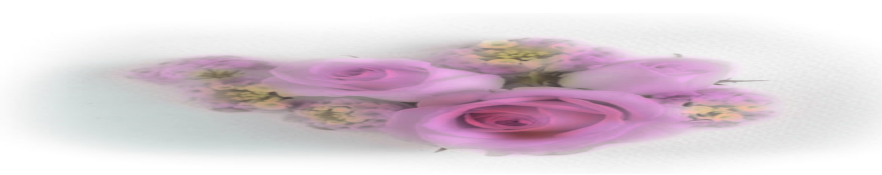


第三章 处理机调度与死锁



对每个用户而言，都希望自己作业的周转时间最短。但作为计算机系统的管理者，则总是希望能使平均周转时间最短，这不仅会有效地提高系统资源的利用率，而且还可使大多数用户都感到满意。可把平均周转时间描述为：

$$T = \frac{1}{n} \left[\sum_{i=1}^n T_i \right]$$



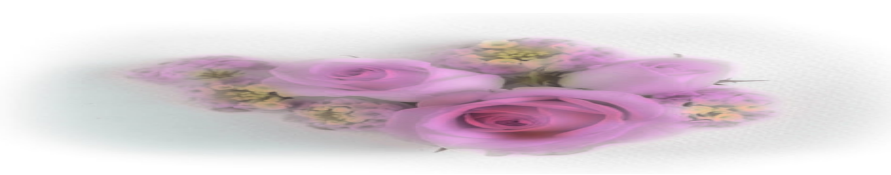


第三章 处理机调度与死锁



作业的周转时间 T 与系统为它提供服务的时间 T_s 之比，即 $W = T/T_s$ ，称为带权周转时间，而平均带权周转时间则可表示为：

$$W = \frac{1}{n} \left[\sum_{i=1}^n \frac{T_i}{T_s} \right]$$





(2) 响应时间快。常把响应时间的长短用来评价分时系统的性能，这是选择分时系统中进程调度算法的重要准则之一。所谓响应时间，是从用户通过键盘提交一个请求开始，直至系统首次产生响应为止的时间，或者说，直到屏幕上显示出结果为止的一段时间间隔。它包括三部分时间：从键盘输入的请求信息传送到处理机的时间，处理机对请求信息进行处理的时间，以及将所形成的响应信息回送到终端显示器的时间。





(3) 截止时间的保证。这是评价实时系统性能的重要指标，因而是选择实时调度算法的重要准则。所谓截止时间，是指某任务必须开始执行的最迟时间，或必须完成的最迟时间。对于严格的实时系统，其调度方式和调度算法必须能保证这一点，否则将可能造成难以预料的后果。

(4) 优先权准则。在批处理、分时和实时系统中选择调度算法时，都可遵循优先权准则，以便让某些紧急的作业能得到及时处理。在要求较严格的场合，往往还须选择抢占式调度方式，才能保证紧急作业得到及时处理。





2. 面向系统的准则

这是为了满足系统要求而应遵循的一些准则。其中，较重要的有以下几点：

(1) 系统吞吐量高。这是用于评价批处理系统性能的另一个重要指标，因而是选择批处理作业调度的重要准则。由于吞吐量是指在单位时间内系统所完成的作业数，因而它与批处理作业的平均长度具有密切关系。对于大型作业，一般吞吐量约为每小时一道作业；对于中、小型作业，其吞吐量则可能达到数十道作业之多。作业调度的方式和算法对吞吐量的大小也将产生较大影响。事实上，对于同一批作业，若采用了较好的调度方式和算法，则可显著地提高系统的吞吐量。





(2) 处理机利用率好。对于大、中型多用户系统，由于CPU价格十分昂贵，致使处理机的利用率成为衡量系统性能的十分重要的指标；而调度方式和算法对处理机的利用率起着十分重要的作用。在实际系统中，CPU的利用率一般在40%(系统负荷较轻)到90%之间。在大、中型系统中，在选择调度方式和算法时，应考虑到这一准则。但对于单用户微机或某些实时系统，则此准则就不那么重要了。

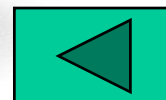
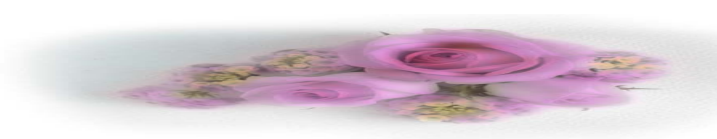
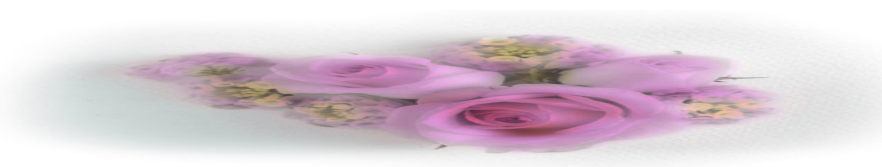




第三章 处理机调度与死锁



(3) 各类资源的平衡利用。在大、中型系统中，不仅要使处理机的利用率高，而且还应能有效地利用其它各类资源，如内存、外存和I/O设备等。选择适当的调度方式和算法可以保持系统中各类资源都处于忙碌状态。但对于微型机和某些实时系统而言，该准则并不重要。





3.3 调度算法

3.2.1 先来先服务和短作业(进程)优先调度算法

1. 先来先服务调度算法

先来先服务(FCFS)调度算法是一种最简单的调度算法, 该算法既可用于作业调度, 也可用于进程调度。当在作业调度中采用该算法时, 每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业, 将它们调入内存, 为它们分配资源、创建进程, 然后放入就绪队列。在进程调度中采用FCFS算法时, 则每次调度是从就绪队列中选择一个最先进入该队列的进程, 为之分配处理机, 使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。





第三章 处理机调度与死锁



FCFS算法比较有利于长作业(进程)，而不利于短作业(进程)。下表列出了A、B、C、D四个作业分别到达系统的时间、要求服务的时间、开始执行的时间及各自的完成时间，并计算出各自的周转时间和带权周转时间。

进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99





第三章 处理机调度与死锁



从表上可以看出，其中短作业C的带权周转时间竟高达100，这是不能容忍的；而长作业D的带权周转时间仅为1.99。据此可知，FCFS调度算法有利于CPU繁忙型的作业，而不利于I/O繁忙型的作业(进程)。CPU繁忙型作业是指该类作业需要大量的CPU时间进行计算，而很少请求I/O。通常的科学计算便属于CPU繁忙型作业。I/O繁忙型作业是指CPU进行处理时需频繁地请求I/O。目前的大多数事务处理都属于I/O繁忙型作业。





第三章 处理机调度与死锁



在此，我们通过一个例子来说明采用FCFS调度算法时的调度性能。图3-4(a)示出有五个进程A、B、C、D、E，它们到达的时间分别是0、1、2、3和4，所要求的服务时间分别是4、3、5、2和4，其完成时间分别是4、7、12、14和18。从每个进程的完成时间中减去其到达时间，即得到其周转时间，进而可以算出每个进程的带权周转时间。





第三章 处理机调度与死锁



作业 情况 调度 算法	进程名	A	B	C	D	E	平 均
	到达时间	0	1	2	3	4	
	服务时间	4	3	5	2	4	
FCFS (a)	完成时间	4	7	12	14	18	
	周转时间	4	6	10	11	14	9
	带权周转时间	1	2	2	5.5	3.5	2.8
SJF (b)	完成时间	4	9	18	6	13	
	周转时间	4	8	16	3	9	8
	带权周转时间	1	2.67	3.1	1.5	2.25	2.1

图3-4 FCFS和SJF调度算法的性能





2. 短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

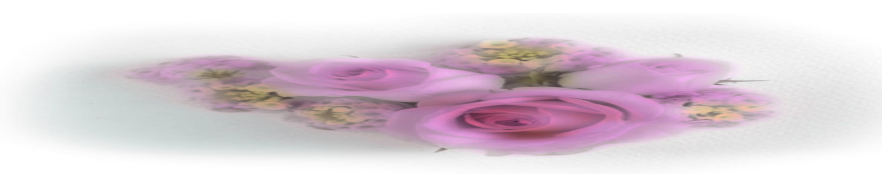




第三章 处理机调度与死锁



为了和FCFS调度算法进行比较，我们仍利用FCFS算法中所使用的实例，并改用SJ(P)F算法重新调度，再进行性能分析。由图3-4 中的(a)和(b)可以看出，采用SJ(P)F算法后，不论是平均周转时间还是平均带权周转时间，都有较明显的改善，尤其是对短作业D，其周转时间由原来的(用FCFS算法时)11降为3；而平均带权周转时间是从5.5降到1.5。这说明SJF调度算法能有效地降低作业的平均等待时间，提高系统吞吐量。





SJ(P)F调度算法也存在不容忽视的缺点：

(1) 该算法对长作业不利，如作业C的周转时间由10增至16，其带权周转时间由2增至3.1。更严重的是，如果有一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度。

(2) 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业(进程)会被及时处理。

(3) 由于作业(进程)的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。





3.3.2 高优先权优先调度算法

1. 优先权调度算法的类型

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。





1) 非抢占式优先权算法

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。





2) 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程 i 时，就将其优先权 P_i 与正在执行的进程 j 的优先权 P_j 进行比较。如果 $P_i \leq P_j$ ，原进程 P_j 便继续执行；但如果是 $P_i > P_j$ ，则立即停止 P_j 的执行，做进程切换，使 i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。





2. 优先权的类型

对于最高优先权优先调度算法，其关键在于：它是使用静态优先权，还是用动态优先权，以及如何确定进程的优先权。

1) 静态优先权

静态优先权是在创建进程时确定的，且在进程的整个运行期间保持不变。一般地，优先权是利用某一范围内的一个整数来表示的，例如， $0\sim 7$ 或 $0\sim 255$ 中的某一整数，又把该整数称为优先数，只是具体用法各异：有的系统用“0”表示最高优先权，当数值愈大时，其优先权愈低；而有的系统恰恰相反。





确定进程优先权的依据有如下三个方面：

(1) 进程类型。通常，系统进程(如接收进程、对换进程、磁盘I/O进程)的优先权高于一般用户进程的优先权。

(2) 进程对资源的需求。如进程的估计执行时间及内存需要量的多少，对这些要求少的进程应赋予较高的优先权。

(3) 用户要求。这是由用户进程的紧迫程度及用户所付费用的多少来确定优先权的。

静态优先权法简单易行，系统开销小，但不够精确，很可能出现优先权低的作业(进程)长期没有被调度的情况。因此，仅在要求不高的系统中才使用静态优先权。





2) 动态优先权

动态优先权是指在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。例如，我们可以规定，在就绪队列中的进程，随其等待时间的增长，其优先权以速率 a 提高。若所有的进程都具有相同的优先权初值，则显然是最先进入就绪队列的进程将因其动态优先权变得最高而优先获得处理机，此即FCFS算法。若所有的就绪进程具有各不相同的优先权初值，那么，对于优先权初值低的进程，在等待了足够的时间后，其优先权便可能升为最高，从而可以获得处理机。当采用抢占式优先权调度算法时，如果再规定当前进程的优先权以速率 b 下降，则可防止一个长作业长期地垄断处理机。





3. 高响应比优先调度算法

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先权随着等待时间的增加而以速率 a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$$



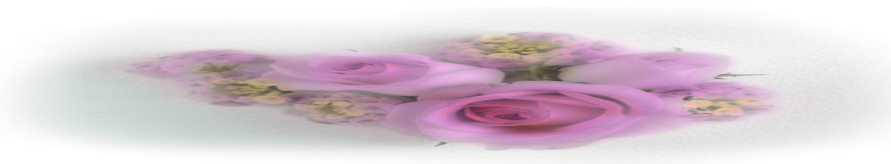


第三章 处理机调度与死锁



由于等待时间与服务时间之和就是系统对该作业的响应时间，故该优先权又相当于响应比 R_p 。据此，又可表示为：

$$R_p = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$$





第三章 处理机调度与死锁



由上式可以看出：

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。





(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。

简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。





3.3.3 基于时间片的轮转调度算法

1. 时间片轮转法

1) 基本原理

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。





2) 时间片大小的确定

在时间片轮转算法中，时间片的大小对系统性能有很大的影响，如选择很小的时间片将有利于短作业，因为它能较快地完成，但会频繁地发生中断、进程上下文的切换，从而增加系统的开销；反之，如选择太长的时间片，使得每个进程都能在一个时间片内完成，时间片轮转算法便退化为FCFS算法，无法满足交互式用户的需求。一个较为可取的大小是，时间片略大于一次典型的交互所需要的时间。这样可使大多数进程在一个时间片内完成。

图3-5示出了时间片分别为 $q=1$ 和 $q=4$ 时，A、B、C、D、E五个进程的运行情况，而图3-6为 $q=1$ 和 $q=4$ 时各进程的平均周转时间和带权平均周转时间。图中的RR(Round Robin)表示轮转调度算法。





第三章 处理机调度与死锁

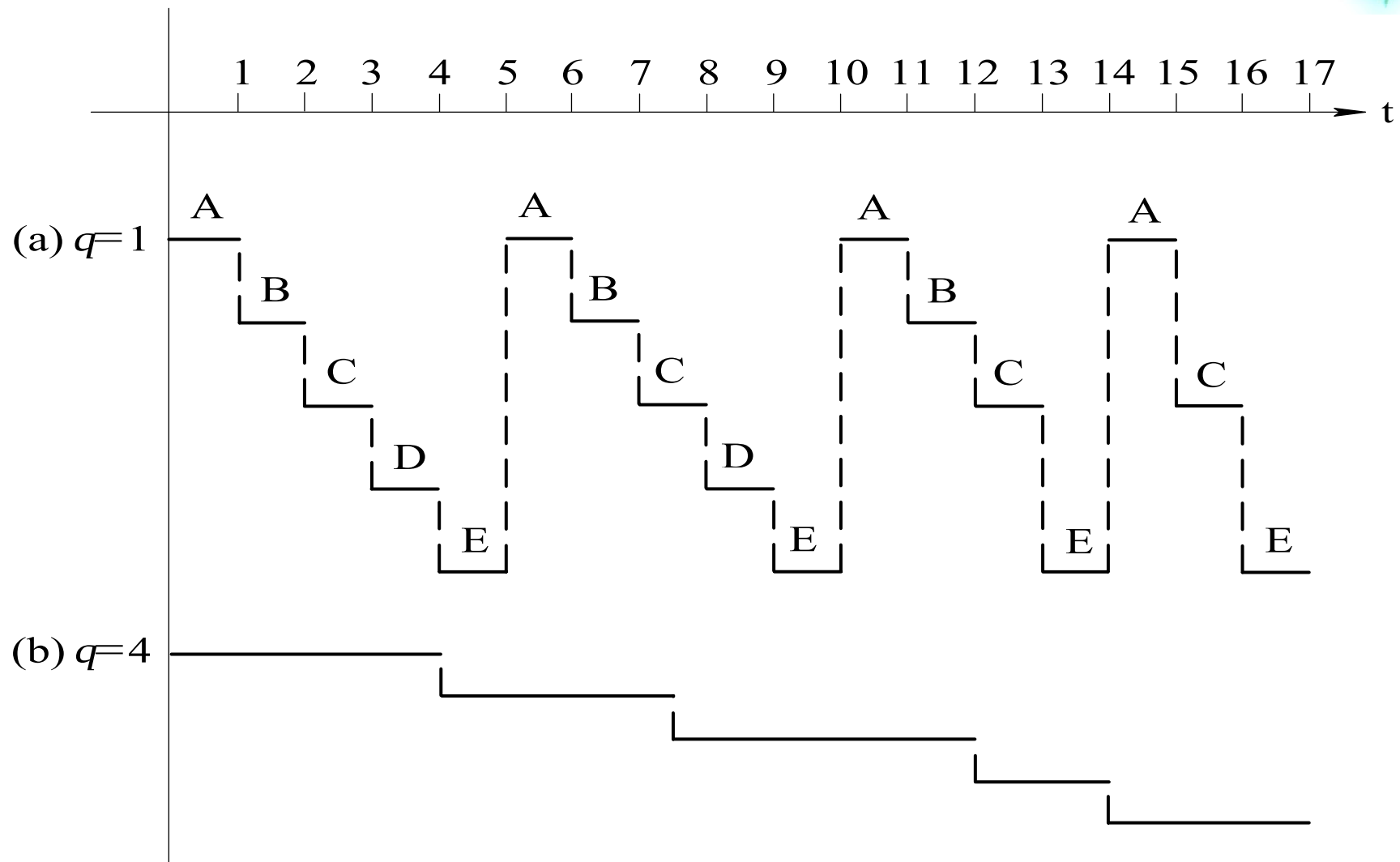


图3-5 $q=1$ 和 $q=4$ 时的进程运行情况



第三章 处理机调度与死锁



作业 情况 时 间 片	进程名	A	B	C	D	E	平均
	到达时间	0	1	2	3	4	
	服务时间	4	3	4	2	4	
RR $q=1$	完成时间	15	12	16	9	17	
	周转时间	15	11	14	6	13	11.8
	带权周转时间	3.75	3.67	3.5	3	3.33	3.46
RR $q=4$	完成时间	4	7	11	13	17	
	周转时间	4	6	9	10	13	8.4
	带权周转时间	1	2	2.25	5	3.33	2.5

图3-6 $q=1$ 和 $q=4$ 时进程的周转时间





2. 多级反馈队列调度算法

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，.....，第 $i+1$ 个队列的时间片要比第 i 个队列的时间片长一倍。图3-7是多级反馈队列算法的示意。





第三章 处理机调度与死锁

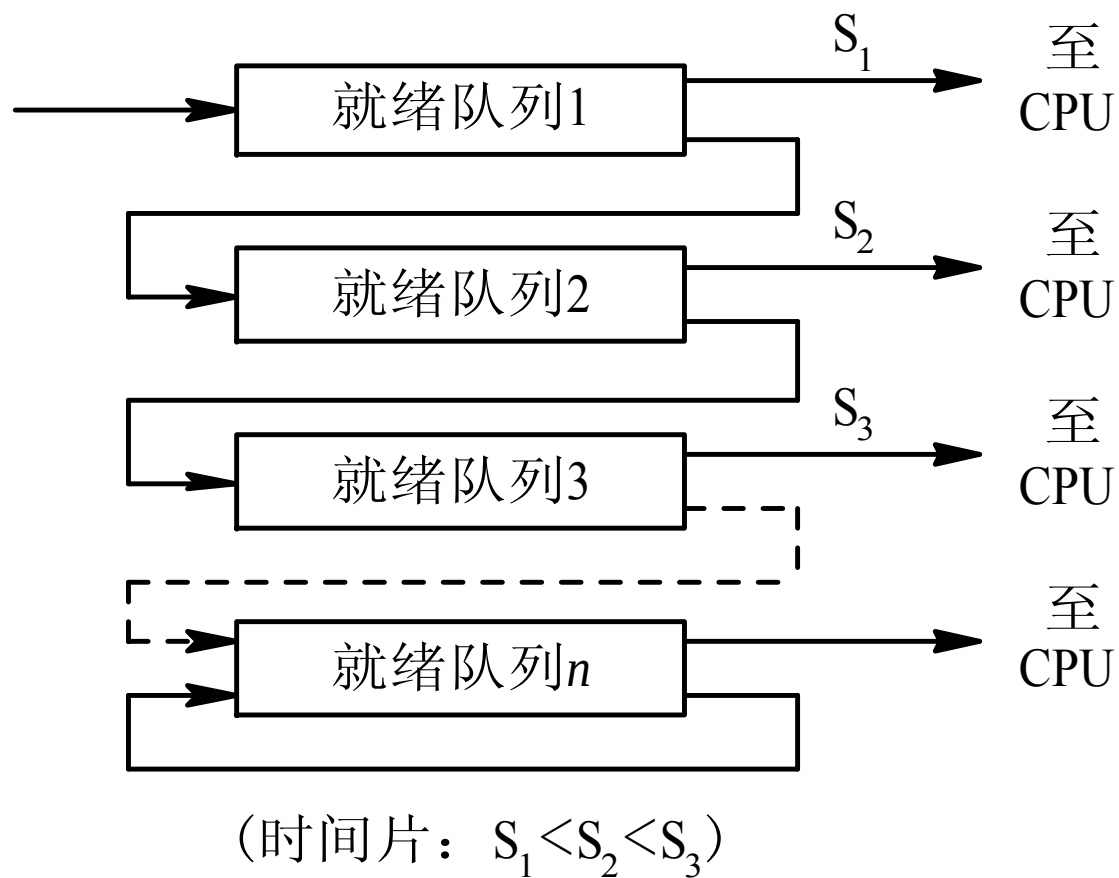


图 3-7 多级反馈队列调度算法





(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列中便采取按时间片轮转的方式运行。





(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第 $1 \sim (i-1)$ 队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 $1 \sim (i-1)$ 中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第 i 队列的末尾，把处理机分配给新到的高优先权进程。





3. 多级反馈队列调度算法的性能

多级反馈队列调度算法具有较好的性能，能很好地满足各种类型用户的需要。

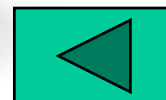
(1) 终端型作业用户。由于终端型作业用户所提交的作业大多属于交互型作业，作业通常较小，系统只要能使这些作业(进程)在第一队列所规定的时间片内完成，便可使终端型作业用户都感到满意。





(2) 短批处理作业用户。对于很短的批处理型作业，开始时像终端型作业一样，如果仅在第一队列中执行一个时间片即可完成，便可获得与终端型作业一样的响应时间。对于稍长的作业，通常也只需在第二队列和第三队列各执行一个时间片即可完成，其周转时间仍然较短。

(3) 长批处理作业用户。对于长作业，它将依次在第1，2，...， n 个队列中运行，然后再按轮转方式运行，用户不必担心其作业长期得不到处理。





3.4 实时调度

3.4.1 实现实时调度的基本条件

1. 提供必要的信息

为了实现实时调度，系统应向调度程序提供有关任务的下述一些信息：

(1) 就绪时间。这是该任务成为就绪状态的起始时间，在周期任务的情况下，它就是事先预知的一串时间序列；而在非周期任务的情况下，它也可能是预知的。





(2) 开始截止时间和完成截止时间。对于典型的实时应用，只须知道开始截止时间，或者知道完成截止时间。

(3) 处理时间。这是指一个任务从开始执行直至完成所需的时间。在某些情况下，该时间也是系统提供的。

(4) 资源要求。这是指任务执行时所需的一组资源。

(5) 优先级。如果某任务的开始截止时间已经错过，就会引起故障，则应为该任务赋予“绝对”优先级；如果开始截止时间的推迟对任务的继续运行无重大影响，则可为该任务赋予“相对”优先级，供调度程序参考。





2. 系统处理能力强

在实时系统中，通常都有着多个实时任务。若处理机的处理能力不够强，则有可能因处理机忙不过来而使某些实时任务不能得到及时处理，从而导致发生难以预料的后果。假定系统中有 m 个周期性的硬实时任务，它们的处理时间可表示为 C_i ，周期时间表示为 P_i ，则在单处理机情况下，必须满足下面的限制条件：

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$





第三章 处理机调度与死锁



系统才是可调度的。假如系统中有6个硬实时任务，它们的周期时间都是 50 ms，而每次的处理时间为 10 ms，则不难算出，此时是不能满足上式的，因而系统是不可调度的。

解决的方法是提高系统的处理能力，其途径有二：其一仍是采用单处理机系统，但须增强其处理能力，以显著地减少对每一个任务的处理时间；其二是采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq N$$





3. 采用抢占式调度机制

在含有硬实时任务的实时系统中，广泛采用抢占机制。当一个优先权更高的任务到达时，允许将当前任务暂时挂起，而令高优先权任务立即投入运行，这样便可满足该硬实时任务对截止时间的要求。但这种调度机制比较复杂。

对于一些小型实时系统，如果能预知任务的开始截止时间，则对实时任务的调度可采用非抢占调度机制，以简化调度程序和对任务调度时所花费的系统开销。但在设计这种调度机制时，应使所有的实时任务都比较小，并在执行完关键性程序和临界区后，能及时地将自己阻塞起来，以便释放出处理机，供调度程序去调度那种开始截止时间即将到达的任务。





4. 具有快速切换机制

为保证要求较高的硬实时任务能及时运行，在实时系统中还应具有快速切换机制，以保证能进行任务的快速切换。该机制应具有如下两方面的能力：

(1) 对外部中断的快速响应能力。为使在紧迫的外部事件请求中断时系统能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短，以免耽误时机(其它紧迫任务)。

(2) 快速的任務分派能力。在完成任務调度后，便应进行任务切换。为了提高分派程序进行任务切换时的速度，应使系统中的每个运行功能单位适当地小，以减少任务切换的时间开销。





3.4.2 实时调度算法的分类

1. 非抢占式调度算法

1) 非抢占式轮转调度算法

该算法常用于工业生产的群控系统中，由一台计算机控制若干个相同的(或类似的)对象，为每一个被控对象建立一个实时任务，并将它们排成一个轮转队列。调度程序每次选择队列中的第一个任务投入运行。当该任务完成后，便把它挂在轮转队列的末尾，等待下次调度运行，而调度程序再选择下一个(队首)任务运行。这种调度算法可获得数秒至数十秒的响应时间，可用于要求不太严格的实时控制系统中。





2) 非抢占式优先调度算法

如果在实时系统中存在着要求较为严格(响应时间为数百毫秒)的任务,则可采用非抢占式优先调度算法为这些任务赋予较高的优先级。当这些实时任务到达时,把它们安排就绪队列的队首,等待当前任务自我终止或运行完成后才能被调度执行。这种调度算法在做了精心的处理后,有可能获得仅为数秒至数百毫秒级的响应时间,因而可用于有一定要求的实时控制系统中。





2. 抢占式调度算法

在要求较严格的(响应时间为数十毫秒以下)的实时系统中,应采用抢占式优先权调度算法。可根据抢占发生时间的不同而进一步分成以下两种调度算法。

1) 基于时钟中断的抢占式优先权调度算法

在某实时任务到达后,如果该任务的优先级高于当前任务的优先级,这时并不立即抢占当前任务的处理机,而是等到时钟中断到来时,调度程序才剥夺当前任务的执行,将处理机分配给新到的高优先权任务。这种调度算法能获得较好的响应效果,其调度延迟可降为几十毫秒至几毫秒。因此,此算法可用于大多数的实时系统中。





2) 立即抢占(Immediate Preemption)的优先权调度算法

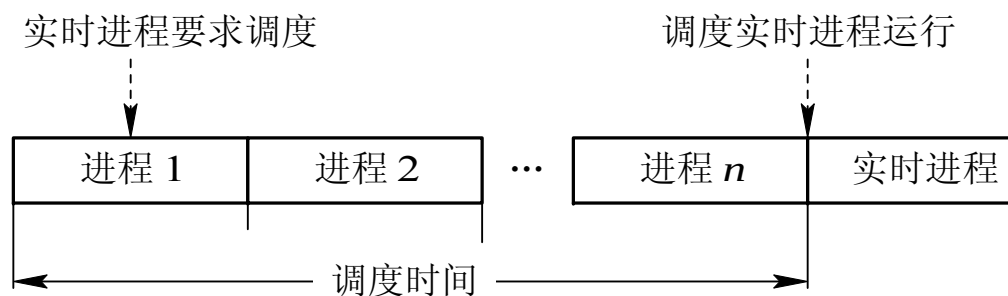
在这种调度策略中，要求操作系统具有快速响应外部事件中断的能力。一旦出现外部中断，只要当前任务未处于临界区，便立即剥夺当前任务的执行，把处理机分配给请求中断的紧迫任务。这种算法能获得非常快的响应，可把调度延迟降低到几毫秒至100微秒，甚至更低。

图3-8中的(a)、(b)、(c)、(d)分别示出了采用非抢占式轮转调度算法、非抢占式优先权调度算法、基于时钟中断抢占的优先权调度算法和立即抢占的优先权调度算法四种情况的调度时间。

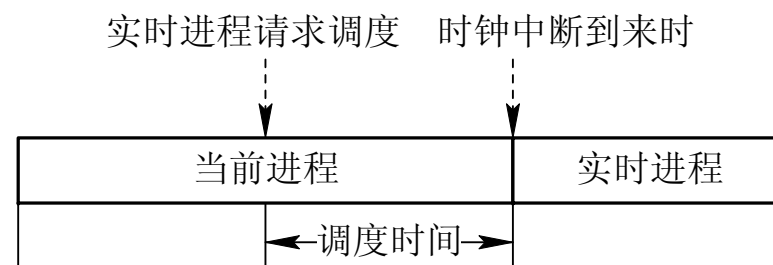




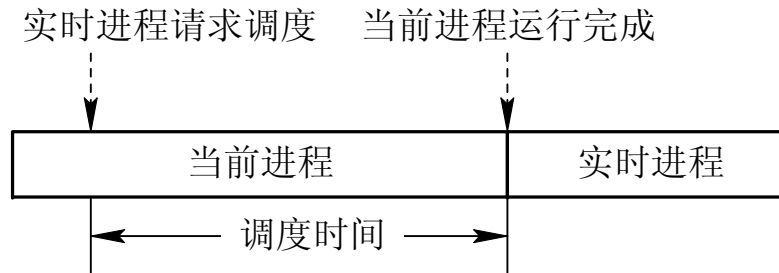
第三章 处理机调度与死锁



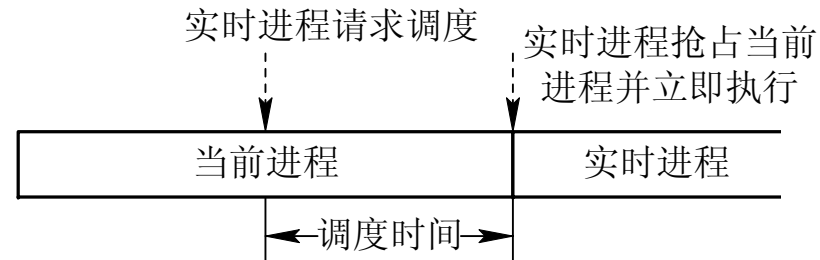
(a) 非抢占式轮转调度



(c) 基于时钟中断抢占的优先级抢占调度



(b) 非抢占式优先级调度



(d) 立即抢占的优先级调度

图3-8 实时进程调度





3.4.3 常用的几种实时调度算法

1. 最早截止时间优先即EDF (Earliest Deadline First) 算法

该算法是根据任务的开始截止时间来确定任务的优先级。截止时间愈早，其优先级愈高。该算法要求在系统中保持一个实时任务就绪队列，该队列按各任务截止时间的早晚排序；当然，具有最早截止时间的任务排在队列的最前面。调度程序在选择任务时，总是选择就绪队列中的第一个任务，为之分配处理机，使之投入运行。最早截止时间优先算法既可用于抢占式调度，也可用于非抢占式调度方式中。





1) 非抢占式调度方式用于非周期实时任务

图3-9示出了将该算法用于非抢占调度方式之例。该例中具有四个非周期任务，它们先后到达。系统首先调度任务1执行，在任务1执行期间，任务2、3又先后到达。由于任务3的开始截止时间早于任务2，故系统在任务1后将调度任务3执行。在此期间又到达作业4，其开始截止时间仍是早于任务2的，故在任务3执行完后，系统又调度任务4执行，最后才调度任务2执行。





第三章 处理机调度与死锁

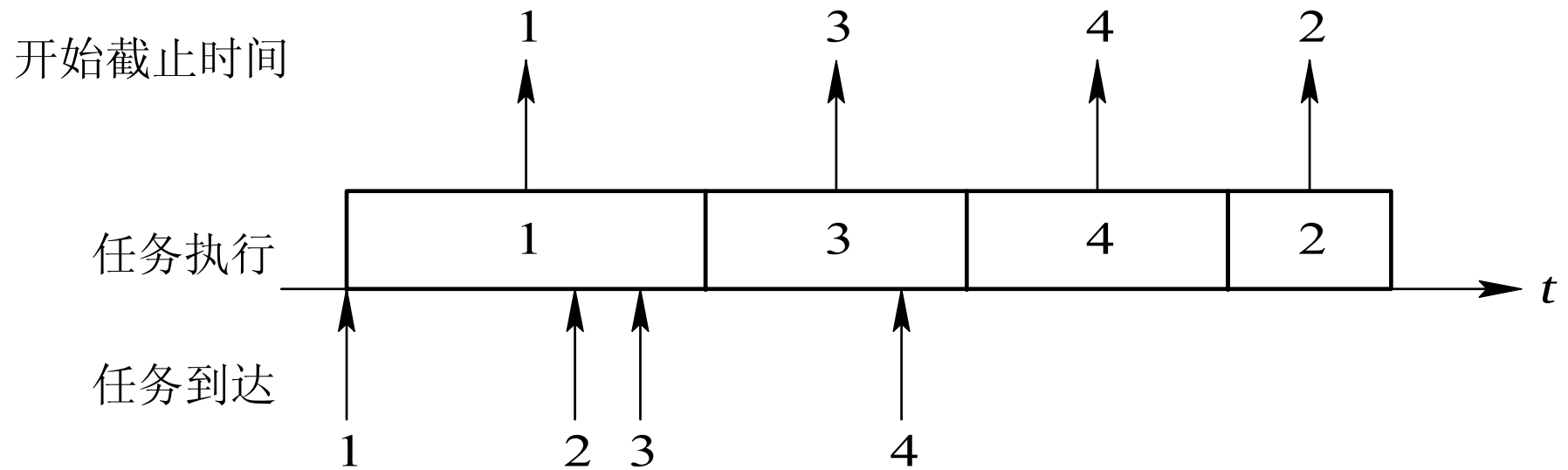


图 3-9 EDF算法用于非抢占调度的调度方式





2) 抢占式调度方式用于周期实时任务

图3-10示出了将最早截止时间优先算法用于抢占调度方式之例。在该例中有两个周期性任务，任务A的周期时间为20 ms，每个周期的处理时间为10 ms；任务B的周期时间为50 ms，每个周期的处理时间为25 ms。图中的第一行示出了两个任务的到达时间、最后期限和执行时间图。其中任务A的到达时间为0、20、40、...；任务A的最后期限为20、40、60、...；任务B的到达时间为0、50、100、...；任务B的最后期限为50、100、150、...(注：单位皆为ms)。





第三章 处理机调度与死锁

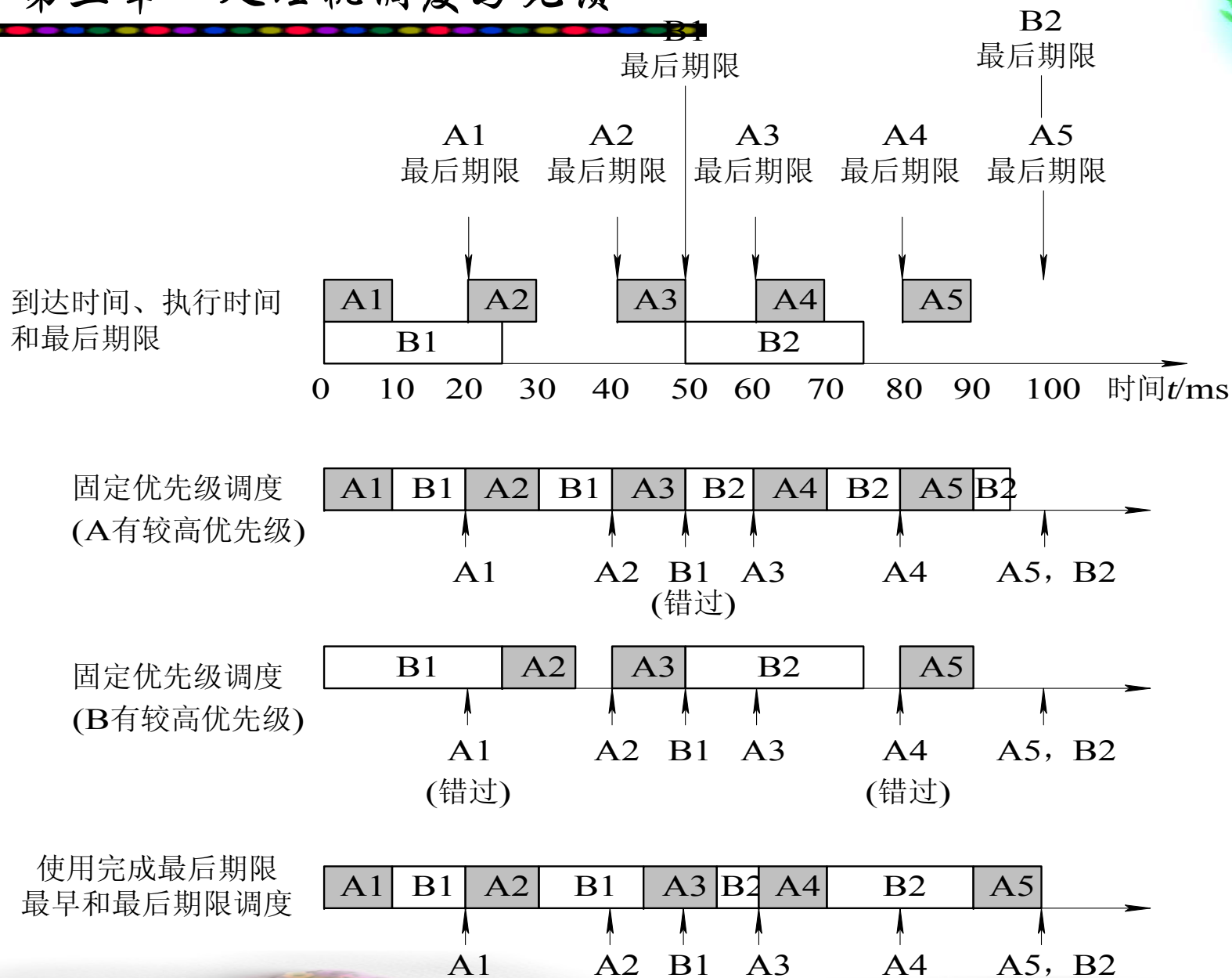


图3-10 最早截止时间优先算法用于抢占调度方式之例



第三章 处理机调度与死锁



为了说明通常的优先级调度不能适用于实时系统，该图特增加了第二和第三行。在第二行中假定任务A具有较高的优先级，所以在 $t=0$ ms时，先调度A1执行，在A1完成后($t = 10$ ms)才调度B1执行；在 $t = 20$ ms时，调度A2执行；在 $t = 30$ ms时，A2完成，又调度B1执行；在 $t = 40$ ms时，调度A3执行；在 $t = 50$ ms时，虽然A3已完成，但B1已错过了它的最后期限，这说明了利用通常的优先级调度已经失败。第三行与第二行类似，只是假定任务B具有较高的优先级。





第三章 处理机调度与死锁



第四行是采用最早截止时间优先算法的时间图。在 $t = 0$ 时，A1和B1同时到达，由于A1的截止时间比B1早，故调度A1执行；在 $t = 10$ 时，A1完成，又调度B1执行；在 $t = 20$ 时，A2到达，由于A2的截止时间比B2早，B1被中断而调度A2执行；在 $t = 30$ 时，A2完成，又重新调度B1执行；在 $t = 40$ 时，A3又到达，但B1的截止时间要比A3早，仍应让B1继续执行直到完成($t = 45$)，然后再调度A3执行；在 $t = 55$ 时，A3完成，又调度B2执行。在该例中利用最早截止时间优先算法可以满足系统的要求。





2. 最低松弛度优先即LLF (Least Laxity First) 算法

该算法是根据任务紧急(或松弛)的程度, 来确定任务的优先级。任务的紧急程度愈高, 为该任务所赋予的优先级就愈高, 以使之优先执行。例如, 一个任务在200 ms时必须完成, 而它本身所需的运行时间就有100 ms, 因此, 调度程序必须在100 ms之前调度执行, 该任务的紧急程度(松弛程度)为100 ms。又如, 另一任务在400 ms时必须完成, 它本身需要运行 150 ms, 则其松弛程度为 250 ms。在实现该算法时要求系统中有一个按松弛度排序的实时任务就绪队列, 松弛度最低的任务排在队列最前面, 调度程序总是选择就绪队列中的队首任务执行。





第三章 处理机调度与死锁



该算法主要用于可抢占调度方式中。假如在一个实时系统中，有两个周期性实时任务A和B，任务A要求每 20 ms执行一次，执行时间为 10 ms；任务B只要求每50 ms执行一次，执行时间为 25 ms。由此可得知任务A和B每次必须完成的时间分别为：A1、A2、A3、...和B1、B2、B3、...，见图3-11。为保证不遗漏任何一次截止时间，应采用最低松弛度优先的抢占调度策略。





第三章 处理机调度与死锁

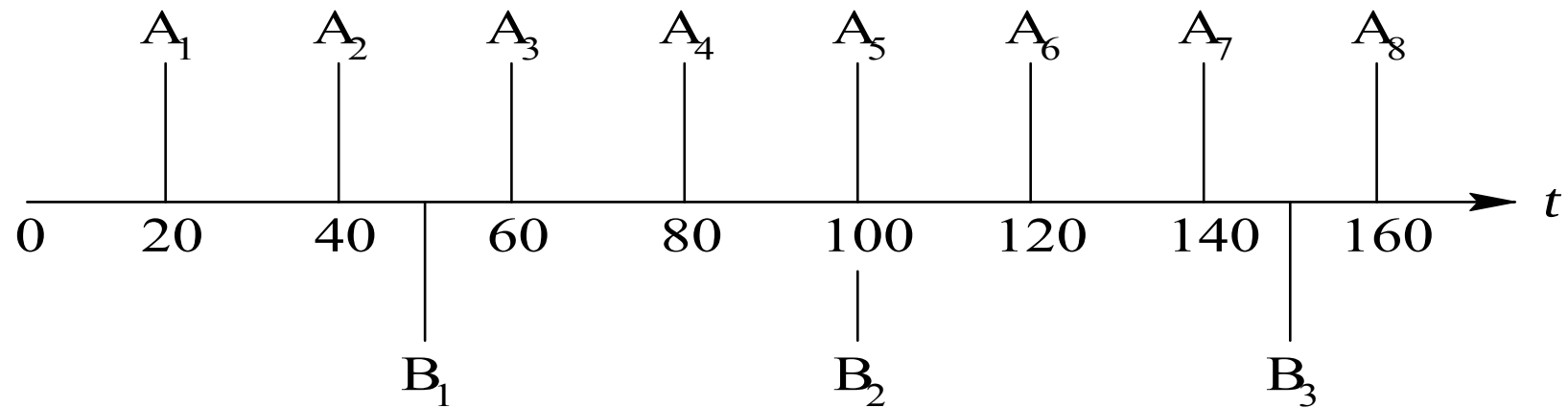


图 3-11 A和B任务每次必须完成的时间



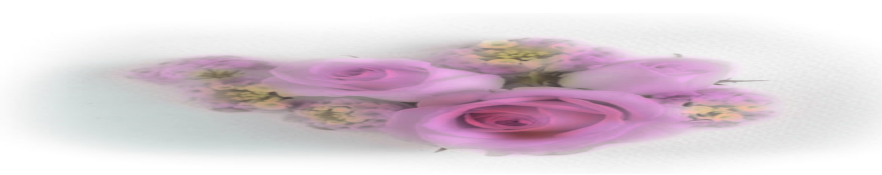


第三章 处理机调度与死锁



在刚开始时($t_1 = 0$), A_1 必须在20 ms时完成, 而它本身运行又需 10 ms, 可算出 A_1 的松弛度为10 ms; B_1 必须在50 ms时完成, 而它本身运行就需25 ms, 可算出 B_1 的松弛度为25 ms, 故调度程序应先调度 A_1 执行。在 $t_2 = 10$ ms时, A_2 的松弛度可按下式算出:

$$\begin{aligned} A_2 \text{的松弛度} &= \text{必须完成时间} - \text{其本身的运行时间} - \text{当前时间} \\ &= 40 \text{ ms} - 10 \text{ ms} - 10 \text{ ms} = 20 \text{ ms} \end{aligned}$$





第三章 处理机调度与死锁



类似地，可算出 B_1 的松弛度为15 ms，故调度程序应选择 B_2 运行。在 $t_3 = 30$ ms时， A_2 的松弛度已减为0(即 $40 - 10 - 30$)，而 B_1 的松弛度为15 ms(即 $50 - 5 - 30$)，于是调度程序应抢占 B_1 的处理机而调度 A_2 运行。在 $t_4 = 40$ ms时， A_3 的松弛度为10 ms(即 $60 - 10 - 40$)，而 B_1 的松弛度仅为5 ms(即 $50 - 5 - 40$)，故又应重新调度 B_1 执行。在 $t_5 = 45$ ms时， B_1 执行完成，而此时 A_3 的松弛度已减为5 ms(即 $60 - 10 - 45$)，而 B_2 的松弛度为30 ms(即 $100 - 25 - 45$)，于是又应调度 A_3 执行。在 $t_6 = 55$ ms时，任务A尚未进入第4周期，而任务B已进入第2周期，故再调度 B_2 执行。在 $t_7 = 70$ ms时， A_4 的松弛度已减至0 ms(即 $80 - 10 - 70$)，而 B_2 的松弛度为20 ms(即 $100 - 10 - 70$)，故此时调度又应抢占 B_2 的处理机而调度 A_4 执行。图3-12示出了具有两个周期性实时任务的调度情况。

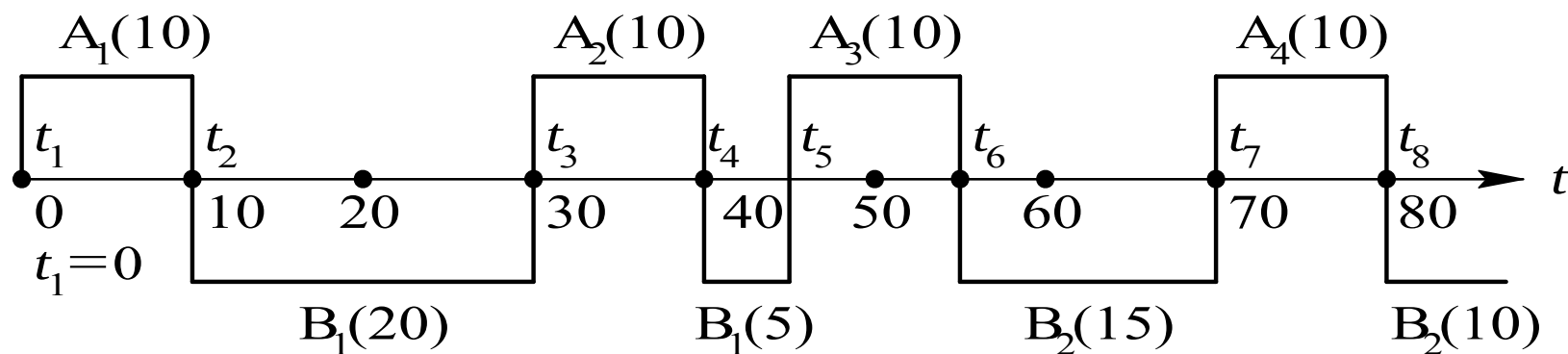


图 3-12 利用LLF算法进行调度的情况





3.5 产生死锁的原因和必要条件

3.5.1 产生死锁的原因

产生死锁的原因可归结为如下两点：

(1) 竞争资源。当系统中供多个进程共享的资源如打印机、公用队列等，其数目不足以满足诸进程的需要时，会引起诸进程对资源的竞争而产生死锁。

(2) 进程间推进顺序非法。进程在运行过程中，请求和释放资源的顺序不当，也同样会导致产生进程死锁。





1. 竞争资源引起进程死锁

1) 可剥夺和非剥夺性资源

可把系统中的资源分成两类，一类是可剥夺性资源，是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。例如，优先权高的进程可以剥夺优先权低的进程的处理机。又如，内存区可由存储器管理程序把一个进程从一个存储区移到另一个存储区，此即剥夺了该进程原来占有的存储区。甚至可将一个进程从内存调出到外存上。可见，CPU和主存均属于可剥夺性资源。另一类资源是不可剥夺性资源，当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，如磁带机、打印机等。





2) 竞争非剥夺性资源

在系统中所配置的非剥夺性资源，由于它们的数量不能满足诸进程运行的需要，会使进程在运行过程中，因争夺这些资源而陷入僵局。例如，系统中只有一台打印机 R_1 和一台磁带机 R_2 ，可供进程 P_1 和 P_2 共享。假定 P_1 已占用了打印机 R_1 ， P_2 已占用了磁带机 R_2 。此时，若 P_2 继续要求打印机， P_2 将阻塞； P_1 若又要求磁带机， P_1 也将阻塞。于是，在 P_1 与 P_2 之间便形成了僵局，两个进程都在等待对方释放出自己所需的资源。但它们又都因不能继续获得自己所需的资源而不能继续推进，从而也不能释放出自己已占有的资源，以致进入死锁状态。为便于说明，我们用方块代表资源，用圆圈代表进程，见图3-13所示。当箭头从进程指向资源时，表示进程请求资源；当箭头从资源指向进程时，表示该资源已被分配给该进程。从中可以看出，这时在 P_1 、 P_2 及 R_1 和 R_2 之间已经形成了一个环路，说明已进入死锁状态。

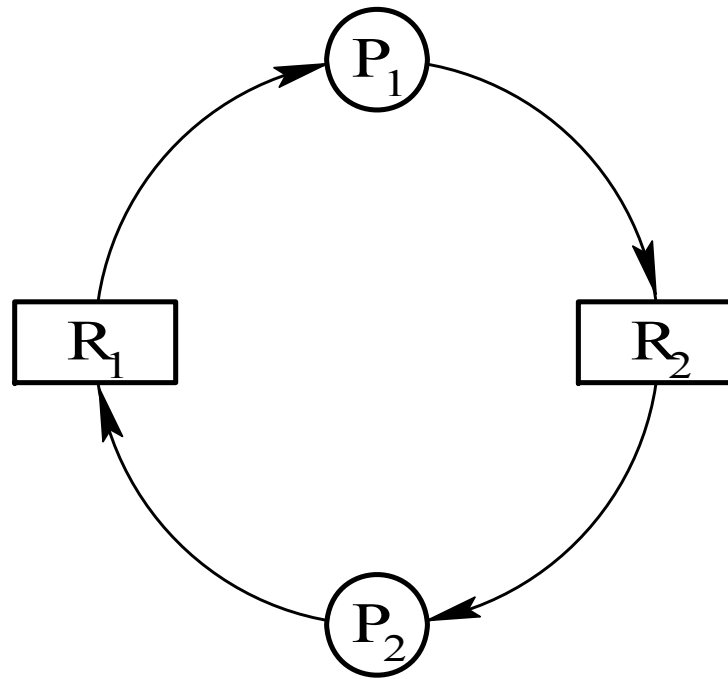
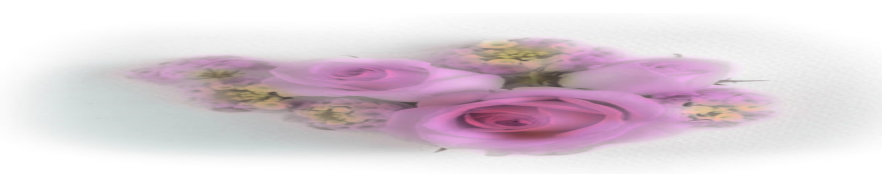


图3-13 I/O设备共享时的死锁情况





3) 竞争临时性资源

上述的打印机资源属于可顺序重复使用型资源，称为永久性资源。还有一种是所谓的临时性资源，这是指由一个进程产生，被另一进程使用一短暂时间后便无用的资源，故也称之为消耗性资源，它也可能引起死锁。图3-14示出了在进程之间通信时形成死锁的情况。图中 S_1 、 S_2 和 S_3 是临时性资源。进程 P_1 产生消息 S_1 ，又要求从 P_3 接收消息 S_3 ；进程 P_3 产生消息 S_3 ，又要求从进程 P_2 接收其所产生的消息 S_2 ；进程 P_2 产生消息 S_2 ，又需要接收进程 P_1 所产生的消息 S_1 。如果消息通信按下述顺序进行：

P_1 : ...Release(S_1); Request(S_3); ...

P_2 : ...Release(S_2); Request(S_1); ...

P_3 : ...Release(S_3); Request(S_2); ...





第三章 处理机调度与死锁



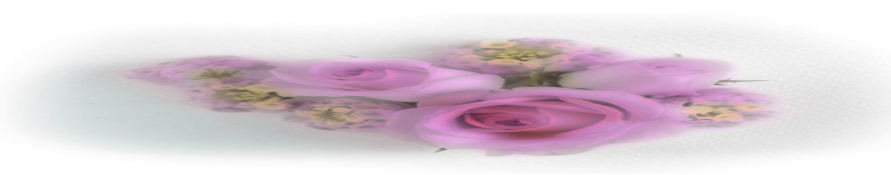
并不可能发生死锁，但若改成下述的运行顺序：

P_1 : ...Request (S_3); Release (S_1); ...

P_2 : ...Request (S_1); Release (S_2); ...

P_3 : ...Request (S_2); Release (S_3); ...

则可能发生死锁。



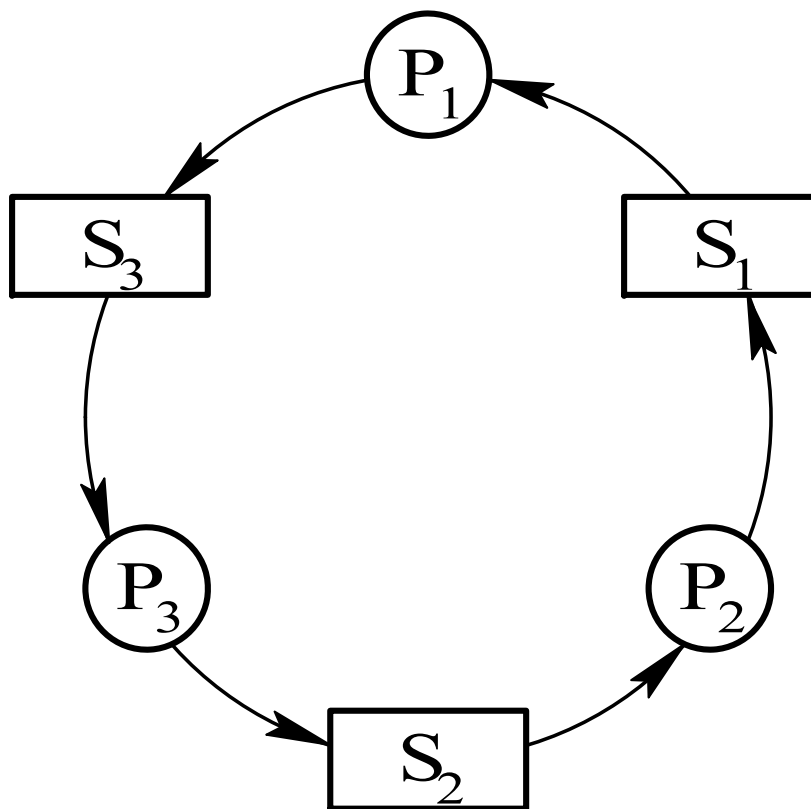


图3-14 进程之间通信时的死锁





2. 进程推进顺序不当引起死锁

由于进程在运行中具有异步性特征，这就可能使上述 P_1 和 P_2 两个进程按下述两种顺序向前推进。

1) 进程推进顺序合法

在进程 P_1 和 P_2 并发执行时，如果按下述顺序推进：

P_1 : Request(R_1); Request(R_2);

P_1 : Release(R_1); Release(R_2);

P_2 : Request(R_2); Request(R_1);

P_2 : Release(R_2); Release(R_1);





第三章 处理机调度与死锁

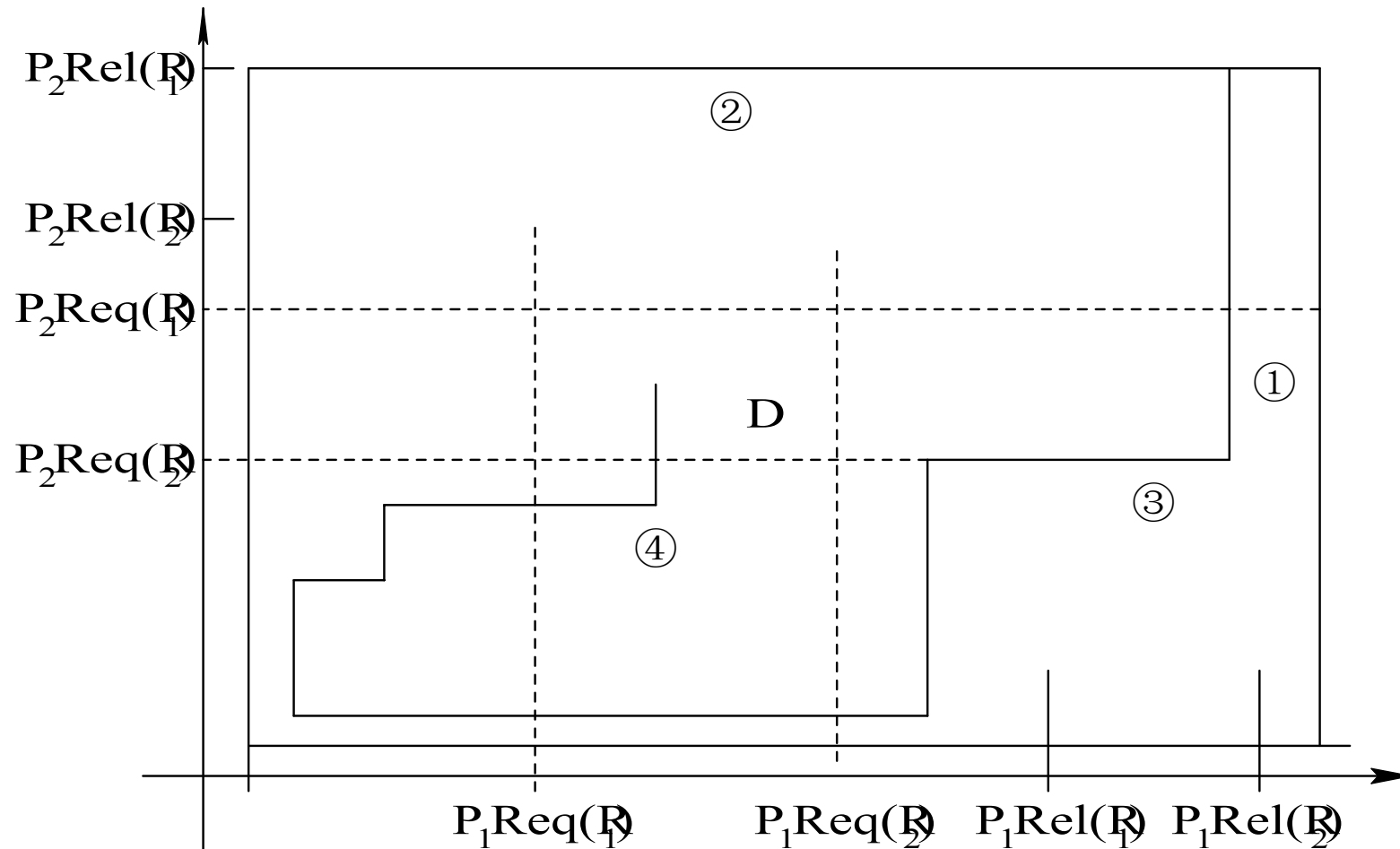


图3-15 进程推进顺序对死锁的影响





2) 进程推进顺序非法

若并发进程 P_1 和 P_2 按曲线④所示的顺序推进，它们将进入不安全区D内。此时 P_1 保持了资源 R_1 ， P_2 保持了资源 R_2 ，系统处于不安全状态。因为这时两进程再向前推进，便可能发生死锁。例如，当 P_1 运行到 $P_1: \text{Request}(R_2)$ 时，将因 R_2 已被 P_2 占用而阻塞；当 P_2 运行到 $P_2: \text{Request}(R_1)$ 时，也将因 R_1 已被 P_1 占用而阻塞，于是发生了进程死锁。





3.5.2 产生死锁的必要条件

虽然进程在运行过程中可能发生死锁，但死锁的发生也必须具备一定的条件。综上所述不难看出，死锁的发生必须具备下列四个必要条件。

(1) 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求该资源，则请求者只能等待，直至占有该资源的进程用毕释放。

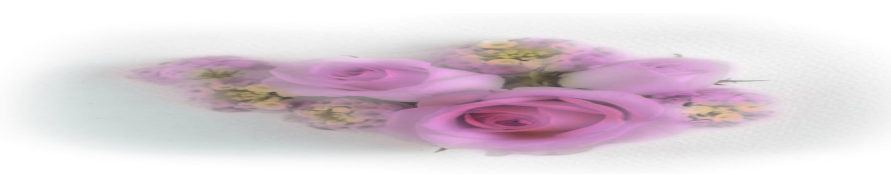
(2) 请求和保持条件：指进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。





(3) 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

(4) 环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合 $\{P_0, P_1, P_2, \dots, P_n\}$ 中的 P_0 正在等待一个 P_1 占用的资源； P_1 正在等待 P_2 占用的资源，……， P_n 正在等待已被 P_0 占用的资源。





3.5.3 处理死锁的基本方法

(1) 预防死锁。这是一种较简单和直观的事先预防的方法。该方法是通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来预防发生死锁。预防死锁是一种较易实现的方法，已被广泛使用。但由于所施加的限制条件往往太严格，因而可能会导致系统资源利用率和系统吞吐量降低。





第三章 处理机调度与死锁



(2) 避免死锁。该方法同样是属于事先预防的策略，但它并不须事先采取各种限制措施去破坏产生死锁的四个必要条件，而是在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免发生死锁。这种方法只需事先施加较弱的限制条件，便可获得较高的资源利用率及系统吞吐量，但在实现上有一定的难度。目前在较完善的系统中常用此方法来避免发生死锁。





第三章 处理机调度与死锁



(3) 检测死锁。这种方法并不须事先采取任何限制性措施，也不必检查系统是否已经进入不安全区，而是允许系统在运行过程中发生死锁。但可通过系统所设置的检测机构，及时地检测出死锁的发生，并精确地确定与死锁有关的进程和资源；然后，采取适当措施，从系统中将已发生的死锁清除掉。

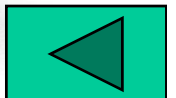




第三章 处理机调度与死锁



(4) 解除死锁。这是与检测死锁相配套的一种措施。当检测到系统中已发生死锁时，须将进程从死锁状态中解脱出来。常用的实施方法是撤消或挂起一些进程，以便回收一些资源，再将 these 资源分配给已处于阻塞状态的进程，使之转为就绪状态，以继续运行。死锁的检测和解除措施有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。





3.6 预防死锁的方法

3.6.1 预防死锁

1. 摒弃“请求和保持”条件

在采用这种方法时，系统规定所有进程在开始运行之前，都必须一次性地申请其在整个运行过程所需的全部资源。此时，若系统有足够的资源分配给某进程，便可把其需要的所有资源分配给该进程，这样，该进程在整个运行期间便不会再提出资源要求，从而摒弃了请求条件。但在分配资源时，只要有一种资源不能满足某进程的要求，即使其它所需的各资源都空闲，也不分配给该进程，而让该进程等待。由于在该进程的等待期间，它并未占有任何资源，因而也摒弃了保持条件，从而可以避免发生死锁。





第三章 处理机调度与死锁



这种预防死锁的方法其优点是简单、易于实现且很安全。但其缺点却也极其明显：首先表现为资源被严重浪费，因为一个进程是一次性地获得其整个运行过程所需的全部资源的，且独占资源，其中可能有些资源很少使用，甚至在整个运行期间都未使用，这就严重地恶化了系统资源的利用率；其次是使进程延迟运行，仅当进程在获得了其所需的全部资源后，才能开始运行，但可能因有些资源已长期被其它进程占用而致使等待该资源的进程迟迟不能运行。





2. 摒弃“不剥夺”条件

在采用这种方法时系统规定，进程是逐个地提出对资源的要求的。当一个已经保持了某些资源的进程，再提出新的资源请求而不能立即得到满足时，必须释放它已经保持了的所有资源，待以后需要时再重新申请。这意味着某一进程已经占有的资源，在运行过程中会被暂时地释放掉，也可认为是被剥夺了，从而摒弃了“不剥夺”条件。





第三章 处理机调度与死锁



这种预防死锁的方法实现起来比较复杂且要付出很大的代价。因为一个资源在使用一段时间后，它的被迫释放可能会造成前段工作的失效，即使是采取了某些防范措施，也还会使进程前后两次运行的信息不连续，例如，进程在运行过程中已用打印机输出信息，但中途又因申请另一资源未果而被迫暂停运行并释放打印机，后来系统又把打印机分配给其它进程使用。当进程再次恢复运行并再次获得打印机继续打印时，这前后两次打印输出的数据并不连续，即打印输出的信息其中间有一段是另一进程的。此外，这种策略还可能因为反复地申请和释放资源，致使进程的执行被无限地推迟，这不仅延长了进程的周转时间，而且也增加了系统开销，降低了系统吞吐量。





3. 摒弃“环路等待”条件

这种方法中规定，系统将所有资源按类型进行线性排队，并赋予不同的序号。例如，令输入机的序号为1，打印机的序号为2，磁带机为3，磁盘为4。所有进程对资源的请求必须严格按照资源序号递增的次序提出，这样，在所形成的资源分配图中，不可能再出现环路，因而摒弃了“环路等待”条件。事实上，在采用这种策略时，总有一个进程占据了较高序号的资源，此后它继续申请的资源必然是空闲的，因而进程可以一直向前推进。





第三章 处理机调度与死锁



这种预防死锁的策略与前两种策略比较，其资源利用率和系统吞吐量都有较明显的改善。但也存在下述严重问题：

首先是为系统中各类资源所分配(确定)的序号必须相对稳定，这就限制了新类型设备的增加。

其次，尽管在为资源的类型分配序号时，已经考虑到大多数作业在实际使用这些资源时的顺序，但也经常会发生这种情况：即作业(进程)使用各类资源的顺序与系统规定的顺序不同，造成对资源的浪费。例如，某进程先用磁带机，后用打印机，但按系统规定，该进程应先申请打印机而后申请磁带机，致使先获得的打印机被长时间闲置。

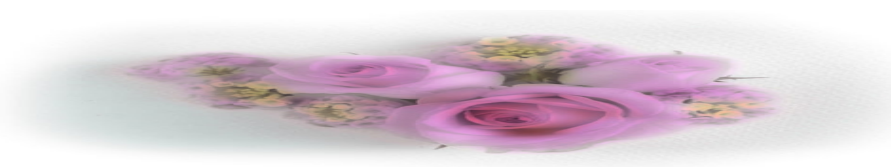




第三章 处理机调度与死锁



第三，为方便用户，系统对用户编程时所施加的限制条件应尽量少。然而这种按规定次序申请的方法，必然会限制用户简单、自主地编程。





3.6.2 系统安全状态

1. 安全状态

在避免死锁的方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，令进程等待。





第三章 处理机调度与死锁



所谓安全状态，是指系统能按某种进程顺序($P_1, P_2, \dots, P_n$) (称 $\langle P_1, P_2, \dots, P_n \rangle$ 序列为安全序列)，来为每个进程 P_i 分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地地完成。如果系统无法找到这样一个安全序列，则称系统处于不安全状态。

虽然并非所有的不安全状态都必然会转为死锁状态，但当系统进入不安全状态后，便有可能进而进入死锁状态；反之，只要系统处于安全状态，系统便可避免进入死锁状态。因此，避免死锁的实质在于：系统在进行资源分配时，如何使系统不进入不安全状态。





2. 安全状态之例

我们通过一个例子来说明安全性。假定系统中有三个进程 P_1 、 P_2 和 P_3 ，共有12台磁带机。进程 P_1 总共要求10台磁带机， P_2 和 P_3 分别要求4台和9台。假设在 T_0 时刻，进程 P_1 、 P_2 和 P_3 已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

进 程	最大需求	已 分 配	可 用
P_1	10	5	3
P_2	4	2	
P_3	9	2	





3. 由安全状态向不安全状态的转换

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。例如，在 T_0 时刻以后， P_3 又请求1台磁带机，若此时系统把剩余3台中的1台分配给 P_3 ，则系统便进入不安全状态。因为此时也无法再找到一个安全序列，例如，把其余的2台分配给 P_2 ，这样，在 P_2 完成后只能释放出4台，既不能满足 P_1 尚需5台的要求，也不能满足 P_3 尚需6台的要求，致使它们都无法推进到完成，彼此都在等待对方释放资源，即陷入僵局，结果导致死锁。类似地，如果我们将剩余的2台磁带机先分配给 P_1 或 P_3 ，也同样都无法使它们推进到完成，因此，从给 P_3 分配了第3台磁带机开始，系统便又进入了不安全状态。由此可见，在 P_3 请求资源时，尽管系统中尚有可用的磁带机，但却不能分配给它，必须让 P_3 一直等待到 P_1 和 P_2 完成，释放出资源后再将足够的资源分配给 P_3 ，它才能顺利完成。



3.6.3 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量Available。这是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 $Available[j]=K$ ，则表示系统中现有 R_j 类资源 K 个。

(2) 最大需求矩阵Max。这是一个 $n \times m$ 的矩阵，它定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。如果 $Max[i, j]=K$ ，则表示进程 i 需要 R_j 类资源的最大数目为 K 。





(3) 分配矩阵Allocation。这也是一个 $n \times m$ 的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation[i,j]=K，则表示进程i当前已分得 R_j 类资源的数目为K。

(4) 需求矩阵Need。这也是一个 $n \times m$ 的矩阵，用以表示每一个进程尚需的各类资源数。如果Need[i,j]=K，则表示进程i还需要 R_j 类资源K个，方能完成其任务。

上述三个矩阵间存在下述关系：

$$\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$$





2. 银行家算法

设 $Request_i$ 是进程 P_i 的请求向量，如果 $Request_i[j]=K$ ，表示进程 P_i 需要 K 个 R_j 类型的资源。当 P_i 发出资源请求后，系统按下述步骤进行检查：

(1) 如果 $Request_i[j] \leq Need[i,j]$ ，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果 $Request_i[j] \leq Available[j]$ ，便转向步骤(3)；否则，表示尚无足够资源， P_i 须等待。





(3) 系统试探着把资源分配给进程 P_i ，并修改下面数据结构中的数值：

$$\text{Available}[j] := \text{Available}[j] - \text{Request}_i[j];$$
$$\text{Allocation}[i,j] := \text{Allocation}[i,j] + \text{Request}_i[j];$$
$$\text{Need}[i,j] := \text{Need}[i,j] - \text{Request}_i[j];$$

(4) 系统执行安全性算法，检查此次资源分配后系统是否处于安全状态。若安全，才正式将资源分配给进程 P_i ，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程 P_i 等待。





3. 安全性算法

系统所执行的安全性算法可描述如下：

(1) 设置两个向量：

① 工作向量Work，它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时， $Work := Available$ 。

② Finish，它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做 $Finish[i] := false$ ；当有足够资源分配给进程时，再令 $Finish[i] := true$ 。





第三章 处理机调度与死锁



(2) 从进程集合中找到一个能满足下述条件的进程:

① $\text{Finish}[i] = \text{false};$

② $\text{Need}[i, j] \leq \text{Work}[j];$ 若找到, 执行步骤(3), 否则, 执行步骤(4)。

(3) 当进程 P_i 获得资源后, 可顺利执行, 直至完成, 并释放出分配给它的资源, 故应执行:

$\text{Work}[j] := \text{Work}[j] + \text{Allocation}[i, j];$

$\text{Finish}[i] := \text{true};$

go to step 2;

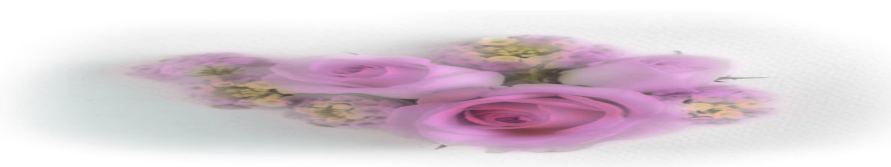




第三章 处理机调度与死锁



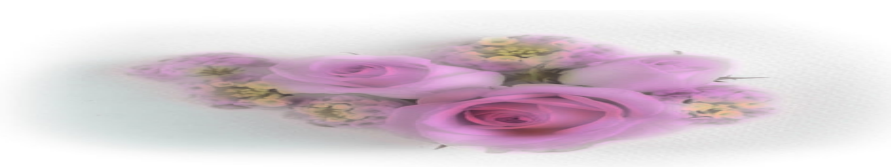
(4) 如果所有进程的 $\text{Finish}[i]=\text{true}$ 都满足，则表示系统处于安全状态；否则，系统处于不安全状态。





4. 银行家算法之例

假定系统中有五个进程 $\{P_0, P_1, P_2, P_3, P_4\}$ 和三类资源 $\{A, B, C\}$ ，各种资源的数量分别为10、5、7，在 T_0 时刻的资源分配情况如图3-16所示。





第三章 处理机调度与死锁



资源 情况 进 程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P ₀	7	5	3	0	1	0	7	4	3	3 (2	3 3	2 0)
P ₁	3	2	2	2	0	0	1	2	2			
P ₂	9	0	2	(3	0	2)	(0	2	0)			
P ₃	2	2	2	3	0	2	6	0	0			
P ₄	4	3	3	2	1	1	0	1	1			
	4	3	3	0	0	2	4	3	1			

图3-16 T₀时刻的资源分配表





第三章 处理机调度与死锁



(1) T_0 时刻的安全性：利用安全性算法对 T_0 时刻的资源分配情况进行分析(见图 3-17所示)可知，在 T_0 时刻存在着一个安全序列 $\{P_1, P_3, P_4, P_2, P_0\}$ ，故系统是安全的。

资源 情况 进 程	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	3	3	2	1	2	2	2	0	0	5	3	2	true
P_3	5	3	2	0	1	1	2	1	1	7	4	3	true
P_4	7	4	3	4	3	1	0	0	2	7	4	5	true
P_2	7	4	5	6	0	0	3	0	2	10	4	7	true
P_0	10	4	7	7	4	3	0	1	0	10	5	7	true

图3-17 T_0 时刻的安全序列





(2) P_1 请求资源: P_1 发出请求向量 $Request_1(1, 0, 2)$, 系统按银行家算法进行检查:

① $Request_1(1, 0, 2) \leq Need_1(1, 2, 2)$

② $Request_1(1, 0, 2) \leq Available_1(3, 3, 2)$

③ 系统先假定可为 P_1 分配资源, 并修改 $Available$, $Allocation_1$ 和 $Need_1$ 向量, 由此形成的资源变化情况如图3-16中的圆括号所示。





第三章 处理机调度与死锁



④ 再利用安全性算法检查此时系统是否安全。如图3-18所示。

资源 情况 进 程	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P ₁	2	3	0	0	2	0	3	0	2	5	3	2	true
P ₃	5	3	2	0	1	1	2	1	1	7	4	3	true
P ₄	7	4	3	4	3	1	0	0	2	7	4	5	true
P ₀	7	4	5	7	4	3	0	1	0	7	5	5	true
P ₂	7	5	5	6	0	0	3	0	2	10	5	7	true

图3-18 P₁申请资源时的安全性检查





第三章 处理机调度与死锁



(3) P_4 请求资源: P_4 发出请求向量 $Request_4(3, 3, 0)$, 系统按银行家算法进行检查:

① $Request_4(3, 3, 0) \leq Need_4(4, 3, 1)$;

② $Request_4(3, 3, 0) \leq Available(2, 3, 0)$, 让 P_4 等待。

(4) P_0 请求资源: P_0 发出请求向量 $Request_0(0, 2, 0)$, 系统按银行家算法进行检查:

① $Request_0(0, 2, 0) \leq Need_0(7, 4, 3)$;

② $Request_0(0, 2, 0) \leq Available(2, 3, 0)$;

③ 系统暂时先假定可为 P_0 分配资源, 并修改有关数据, 如图3-19所示。





第三章 处理机调度与死锁



资源 情况 进 程	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P ₀	0	3	0	7	3	2	2	1	0
P ₁	3	0	2	0	2	0			
P ₂	3	0	2	6	0	0			
P ₃	2	1	1	0	1	1			
P ₄	0	0	2	4	3	1			

图3-19 为P₀分配资源后的有关资源数据



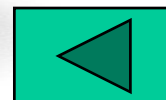
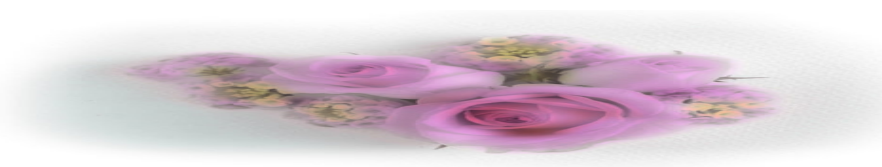


第三章 处理机调度与死锁



(5) 进行安全性检查：可用资源Available(2, 1, 0)已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。

如果在银行家算法中，把P0发出的请求向量改为Request₀(0, 1, 0)，系统是否能将资源分配给它，请读者考虑。





3.7 死锁的检测与解除

3.7.1 死锁的检测

当系统为进程分配资源时，若未采取任何限制性措施，则系统必须提供检测和解除死锁的手段，为此，系统必须做到：

- (1) 保存有关资源的请求和分配信息；
- (2) 提供一种算法，以利用这些信息来检测系统是否已进入死锁状态。





1. 资源分配图(Resource Allocation Graph)

系统死锁可利用资源分配图来描述。该图是由一组结点N和一组边E所组成的一个对偶 $G=(N, E)$ ，它具有下述形式的定义和限制：

(1) 把N分为两个互斥的子集，即一组进程结点 $P=\{p_1, p_2, \dots, p_n\}$ 和一组资源结点 $R=\{r_1, r_2, \dots, r_n\}$ ， $N=P \cup R$ 。在图3-20所示的例子中， $P=\{p_1, p_2\}$ ， $R=\{r_1, r_2\}$ ， $N=\{r_1, r_2\} \cup \{p_1, p_2\}$ 。





第三章 处理机调度与死锁



(2) 凡属于 E 中的一个边 $e \in E$ ，都连接着 P 中的一个结点和 R 中的一个结点， $e = \{p_i, r_j\}$ 是资源请求边，由进程 p_i 指向资源 r_j ，它表示进程 p_i 请求一个单位的 r_j 资源。 $e = \{r_j, p_i\}$ 是资源分配边，由资源 r_j 指向进程 p_i ，它表示把一个单位的资源 r_j 分配给进程 p_i 。图3-13中示出了两个请求边和两个分配边，即 $E = \{(p_1, r_2), (r_2, p_2), (p_2, r_1), (r_1, p_1)\}$ 。

我们用圆圈代表一个进程，用方框代表一类资源。由于一种类型的资源可能有多个，我们用方框中的一个点代表一类资源中的一个资源。此时，请求边是由进程指向方框中的 r_j ，而分配边则应始于方框中的一个点。图3-20示出了一个资源分配图。图中， p_1 进程已经分得了两个 r_1 资源，并又请求一个 r_2 资源； p_2 进程分得了—个 r_1 和—个 r_2 资源，并又请求 r_1 资源。



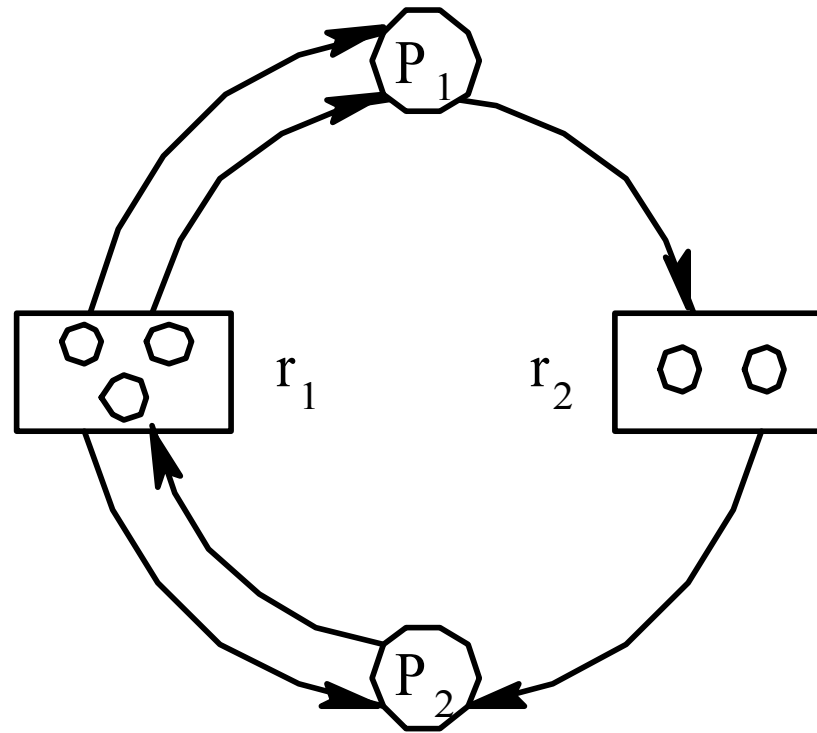


图3-20 每类资源有多个时的情况





2. 死锁定理

我们可以利用把资源分配图加以简化的方法(图3-21)，来检测当系统处于S状态时是否为死锁状态。简化方法如下：

(1) 在资源分配图中，找出一个既不阻塞又非独立的进程结点 P_i 。在顺利的情况下， P_i 可获得所需资源而继续运行，直至运行完毕，再释放其所占有的全部资源，这相当于消去 p_i 所求的请求边和分配边，使之成为孤立的结点。在图3-21(a)中，将 p_1 的两个分配边和一个请求边消去，便形成图(b)所示的情况。



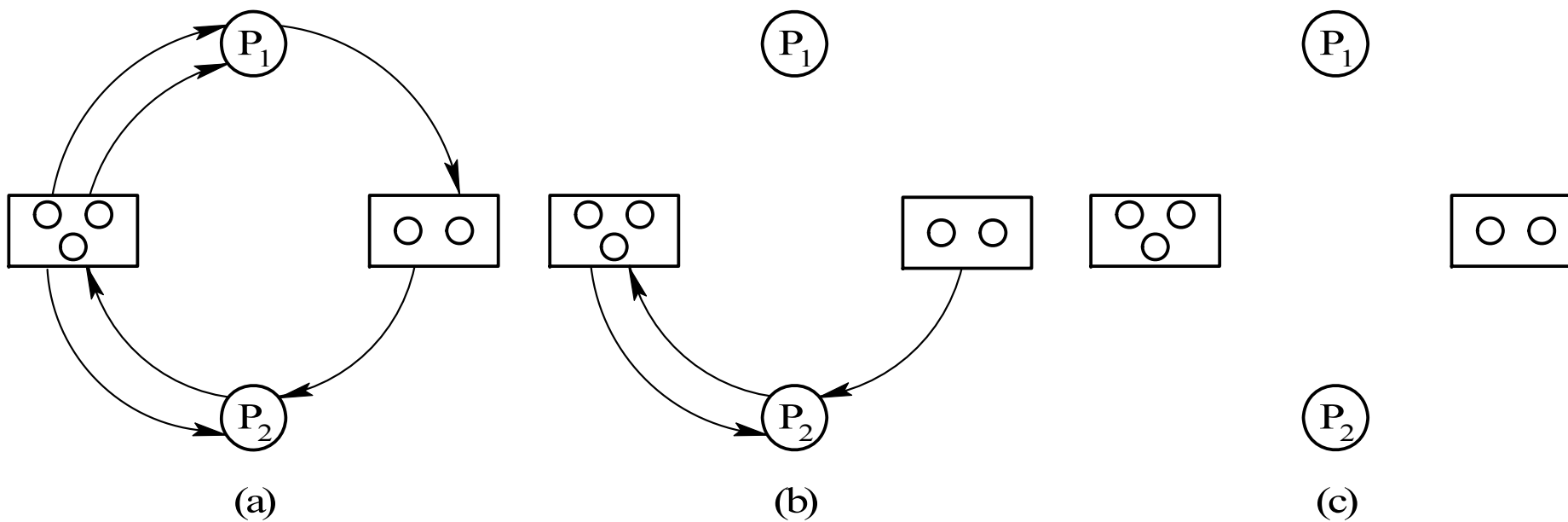


图3-21 资源分配图的简化



第三章 处理机调度与死锁



(2) p_1 释放资源后，便可使 p_2 获得资源而继续运行，直至 p_2 完成后又释放出它所占有的全部资源，形成图(c)所示的情况。

(3) 在进行一系列的简化后，若能消去图中所有的边，使所有的进程结点都成为孤立结点，则称该图是可完全简化的；若不能通过任何过程使该图完全简化，则称该图是不可完全简化的。

对于较复杂的资源分配图，可能有多个既未阻塞，又非孤立的进程结点，不同的简化顺序是否会得到不同的简化图？有关文献已经证明，所有的简化顺序，都将得到相同的不可简化图。同样可以证明： S 为死锁状态的充分条件是：当且仅当 S 状态的资源分配图是不可完全简化的。该充分条件被称为死锁定理。





3. 死锁检测中的数据结构

死锁检测中的数据结构类似于银行家算法中的数据结构：

(1) 可利用资源向量Available，它表示了 m 类资源中每一类资源的可用数目。

(2) 把不占用资源的进程(向量 $\text{Allocation}_i := 0$)记入L表中，即 $L_i \cup L$ 。

(3) 从进程集合中找到一个 $\text{Request}_i \leq \text{Work}$ 的进程，做如下处理：

① 将其资源分配图简化，释放出资源，增加工作向量
 $\text{Work} := \text{Work} + \text{Allocation}_i$ 。

② 将它记入L表中。





第三章 处理机调度与死锁



(4) 若不能把所有进程都记入L表中, 便表明系统状态S的资源分配图是不可完全简化的。因此, 该系统状态将发生死锁。

Work:=Available;

$L := \{L_i \mid \text{Allocation}_i = 0 \cap \text{Request}_i = 0\}$

for all $L_i \notin L$ do

begin

for all $\text{Request}_i \leq \text{Work}$ do

begin

Work := Work + Allocation_i;

$L_i \cup L$;

end

end

deadlock := $\neg (L = \{p_1, p_2, \dots, p_n\})$;





3.7.2 死锁的解除

当发现有进程死锁时，便应立即把它们从死锁状态中解脱出来。常采用解除死锁的两种方法是：

(1) 剥夺资源。从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态。

(2) 撤消进程。最简单的撤消进程的方法是使全部死锁进程都夭折掉；稍微温和一点的方法是按照某种顺序逐个地撤消进程，直至有足够的资源可用，使死锁状态消除为止。

在出现死锁时，可采用各种策略来撤消进程。例如，为解除死锁状态所需撤消的进程数目最小；或者，撤消进程所付出的代价最小等。一种付出最小代价的方法如图3-22所示。



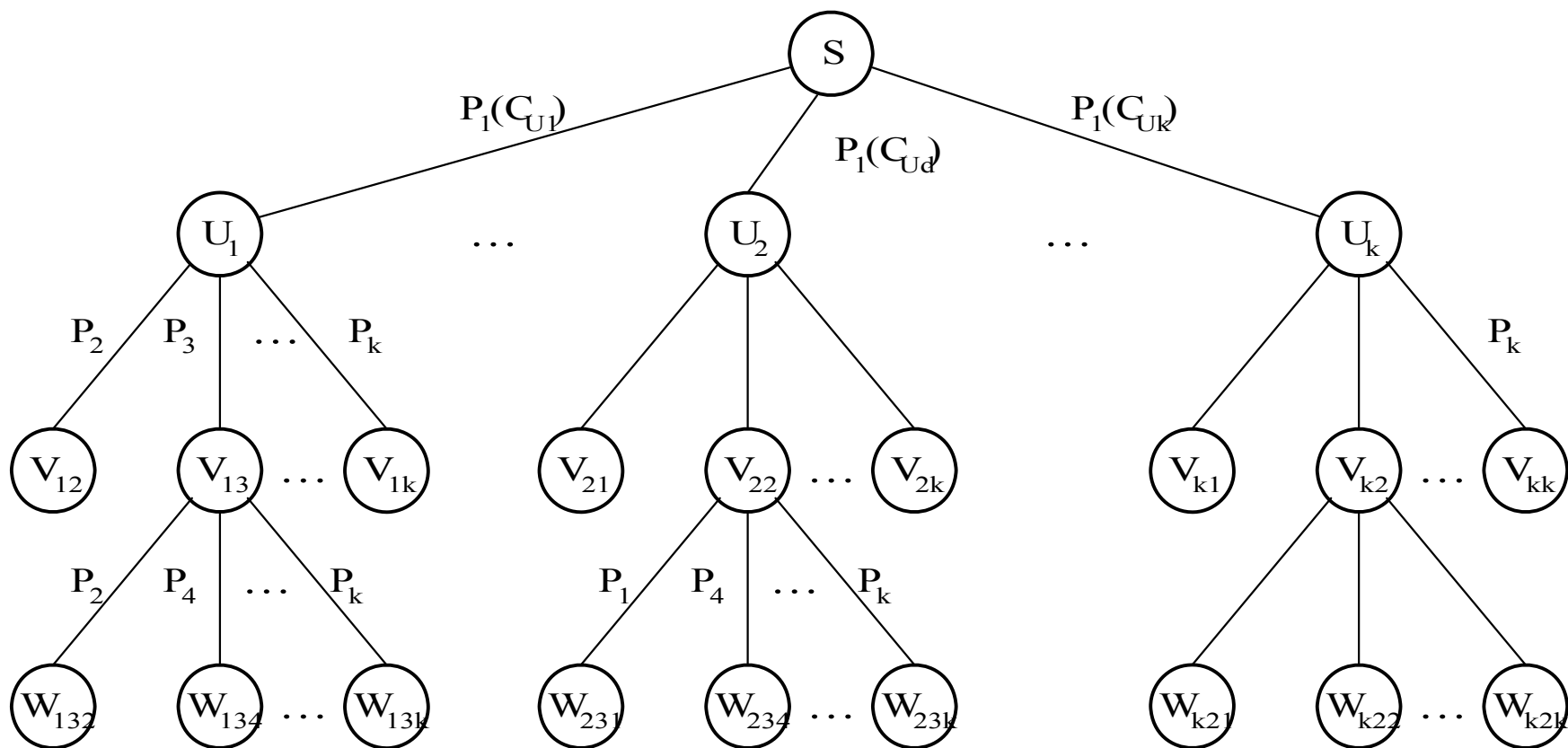


图 3-22 付出代价最小的死锁解除方法





第三章 处理机调度与死锁



假定在死锁状态时，有死锁进程 $P_1, P_2, \dots, P_k$ 。首先，撤消进程 P_1 ，使系统状态由 $S \rightarrow U_1$ ，付出的代价为 C_{U_1} ，然后，仍然从 S 状态中撤消进程 P_2 ，使状态由 $S \rightarrow U_2$ ，其代价为 C_{U_2} ， $\dots$ ，如此下去可得到状态 $U_1, U_2, \dots, U_n$ 。若此时系统仍处于死锁状态，需再进一步撤消进程，如此下去，直至解除死锁状态为止。这种方法为解除死锁状态可能付出的代价将是 $k(k-1)(k-2)\dots/2C$ 。显然，所花费的代价很大，因此，这是一种很不实际的方法。



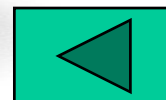


第三章 处理机调度与死锁



一个比较有效的方法是对死锁状态S做如下处理：从死锁状态S中先撤消一个死锁进程 P_1 ，使系统状态由S演变成 U_1 ，将 P_1 记入被撤消进程的集合 $d(T)$ 中，并把所付出的代价 C_1 加入到 $rc(T)$ 中；对死锁进程 P_2 、 P_3 等重复上述过程，得到状态 $U_1, U_2, \dots, U_i, U_n$ ，然后，再按撤消进程时所花费代价的大小，把它插入到由S状态所演变的新状态的队列L中。显然，队列L中的第一个状态 U_1 是由S状态花最小代价撤消一个进程所演变成的状态。在撤消一个进程后，若系统仍处于死锁状态，则再从 U_1 状态按照上述处理方式再依次地撤消一个进程，得到 $U'_1, U'_2, U'_3, \dots, U'_k$ ，状态，再从状态中选取一个代价最小的，如此下去，直到死锁状态解除为止。为把系统从死锁状态中解脱出来，所花费的代价可表示为：

$$R(S)_{\min} = \min\{C_{U_i}\} + \min\{C_{U_j}\} + \min\{C_{U_k}\} + \dots$$





第四章 存储器管理

4.1 存储器的层次结构

4.2 程序的装入和链接

4.3 连续分配方式

4.4 基本分页存储管理方式

4.5 基本分段存储管理方式

4.6 虚拟存储器的基本概念

4.7 请求分页存储管理方式

4.8 页面置换算法

4.9 请求分段存储管理方式





4.1 存储器的层次结构

4.1.1 多级存储器结构

对于通用计算机而言，存储层次至少应具有三级：最高层为CPU寄存器，中间为主存，最底层是辅存。在较高档的计算机中，还可以根据具体的功能分工细划为寄存器、高速缓存、主存储器、磁盘缓存、固定磁盘、可移动存储介质等6层。如图4-1所示，在存储层次中越往上，存储介质的访问速度越快，价格也越高，相对存储容量也越小。其中，寄存器、高速缓存、主存储器和磁盘缓存均属于操作系统存储管理的管辖范畴，掉电后它们存储的信息不再存在。固定磁盘和可移动存储介质属于设备管理的管辖范畴，它们存储的信息将被长期保存。



第四章 存储器管理

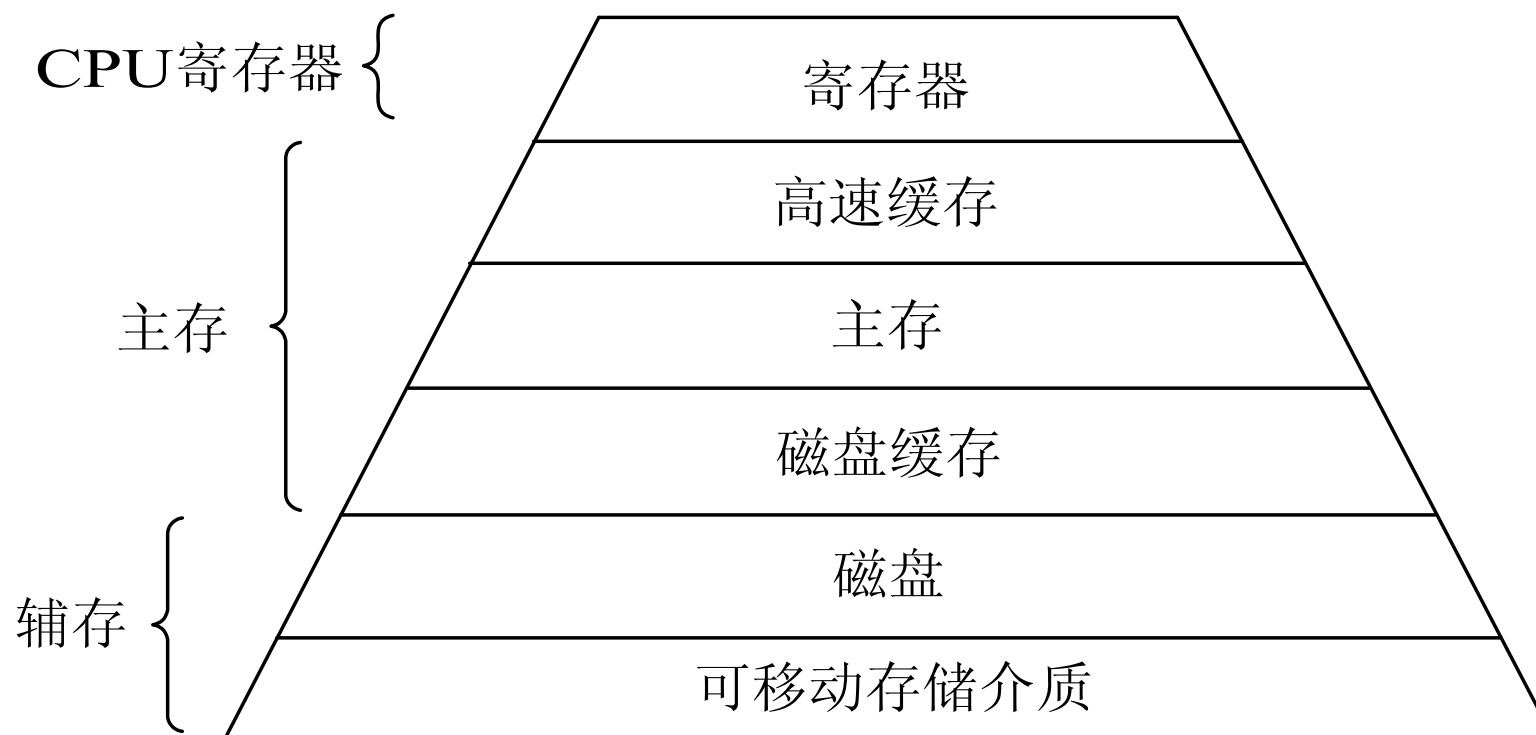


图4-1 计算机系统存储层次示意





4.1.2 主存储器与寄存器

1. 主存储器

主存储器(简称内存或主存)是计算机系统中一个主要部件,用于保存进程运行时的程序和数据,也称可执行存储器,其容量对于当前的微机系统和大中型机,可能一般为数十MB到数GB,而且容量还在不断增加,而嵌入式计算机系统一般仅有几十KB到几MB。CPU的控制部件只能从主存储器中取得指令和数据,数据能够从主存储器读取并将它们装入到寄存器中,或者从寄存器存入到主存储器。CPU与外围设备交换的信息一般也依托于主存储器地址空间。由于主存储器的访问速度远低于CPU执行指令的速度,为缓和这一矛盾,在计算机系统中引入了寄存器和高速缓存。





2. 寄存器

寄存器访问速度最快，完全能与CPU协调工作，但价格却十分昂贵，因此容量不可能做得很大。寄存器的长度一般以字(word)为单位。寄存器的数目，对于当前的微机系统和大中型机，可能有几十个甚至上百个；而嵌入式计算机系统一般仅有几个到几十个。寄存器用于加速存储器的访问速度，如用寄存器存放操作数，或用作地址寄存器加快地址转换速度等。





4.1.3 高速缓存和磁盘缓存

1. 高速缓存

高速缓存是现代计算机结构中的一个重要部件，其容量大于或远大于寄存器，而比内存约小两到三个数量级左右，从几十KB到几MB，访问速度快于主存储器。

根据程序执行的局部性原理(即程序在执行时将呈现出局部性规律，在一较短的时间内，程序的执行仅局限于某个部分)，将主存中一些经常访问的信息存放在高速缓存中，减少访问主存储器的次数，可大幅度提高程序执行速度。





2. 磁盘缓存

由于目前磁盘的I/O速度远低于对主存的访问速度，因此将频繁使用的一部分磁盘数据和信息，暂时存放在磁盘缓存中，可减少访问磁盘的次数。磁盘缓存本身并不是一种实际存在的存储介质，它依托于固定磁盘，提供对主存储器存储空间的扩充，即利用主存中的存储空间，来暂存从磁盘中读出(或写入)的信息。主存也可以看做是辅存的高速缓存，因为，辅存中的数据必须复制到主存方能使用；反之，数据也必须先存在主存中，才能输出到辅存。





第四章 存储器管理



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





4.2 程序的装入和链接

在多道程序环境下，要使程序运行，必须先为之创建进程。而创建进程的第一件事，便是将程序和数据装入内存。如何将一个用户源程序变为一个可在内存中执行的程序，通常都要经过以下几个步骤：首先是要编译，由编译程序(Compiler)将用户源代码编译成若干个目标模块(Object Module)；其次是链接，由链接程序(Linker)将编译后形成的一组目标模块，以及它们所需要的库函数链接在一起，形成一个完整的装入模块(Load Module)；最后是装入，由装入程序(Loader)将装入模块装入内存。图4-2示出了这样的三步过程。本节将扼要阐述程序(含数据)的链接和装入过程。





第四章 存储器管理

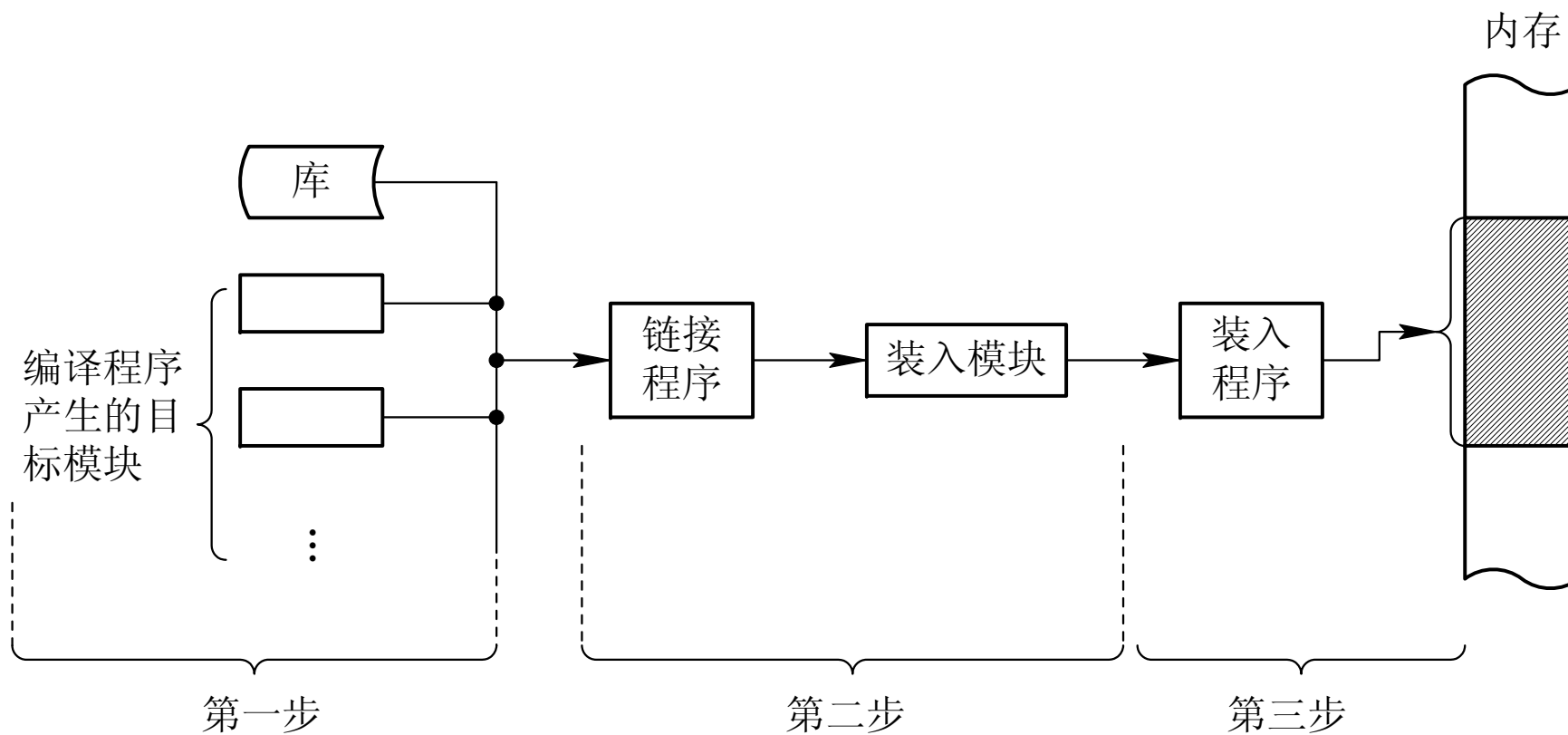


图4-2 对用户程序的处理步骤



4.2.1 程序的装入

1. 绝对装入方式(Absolute Loading Mode)

在编译时，如果知道程序将驻留在内存的什么位置，那么，编译程序将产生绝对地址的目标代码。例如，事先已知用户程序(进程)驻留在从R处开始的位置，则编译程序所产生的目标模块(即装入模块)便从R处开始向上扩展。绝对装入程序按照装入模块中的地址，将程序和数据装入内存。装入模块被装入内存后，由于程序中的逻辑地址与实际内存地址完全相同，故不须对程序和数据地址进行修改。





第四章 存储器管理



程序中所使用的绝对地址，既可在编译或汇编时给出，也可由程序员直接赋予。但在由程序员直接给出绝对地址时，不仅要求程序员熟悉内存的使用情况，而且一旦程序或数据被修改后，可能要改变程序中的所有地址。因此，通常是宁可在程序中采用符号地址，然后在编译或汇编时，再将这些符号地址转换为绝对地址。





2. 可重定位装入方式(Relocation Loading Mode)

绝对装入方式只能将目标模块装入到内存中事先指定的位置。在多道程序环境下，编译程序不可能预知所编译的目标模块应放在内存的何处，因此，绝对装入方式只适用于单道程序环境。在多道程序环境下，所得到的目标模块的起始地址通常是从0开始的，程序中的其它地址也都是相对于起始地址计算的。此时应采用可重定位装入方式，根据内存的当前情况，将装入模块装入到内存的适当位置。





第四章 存储器管理

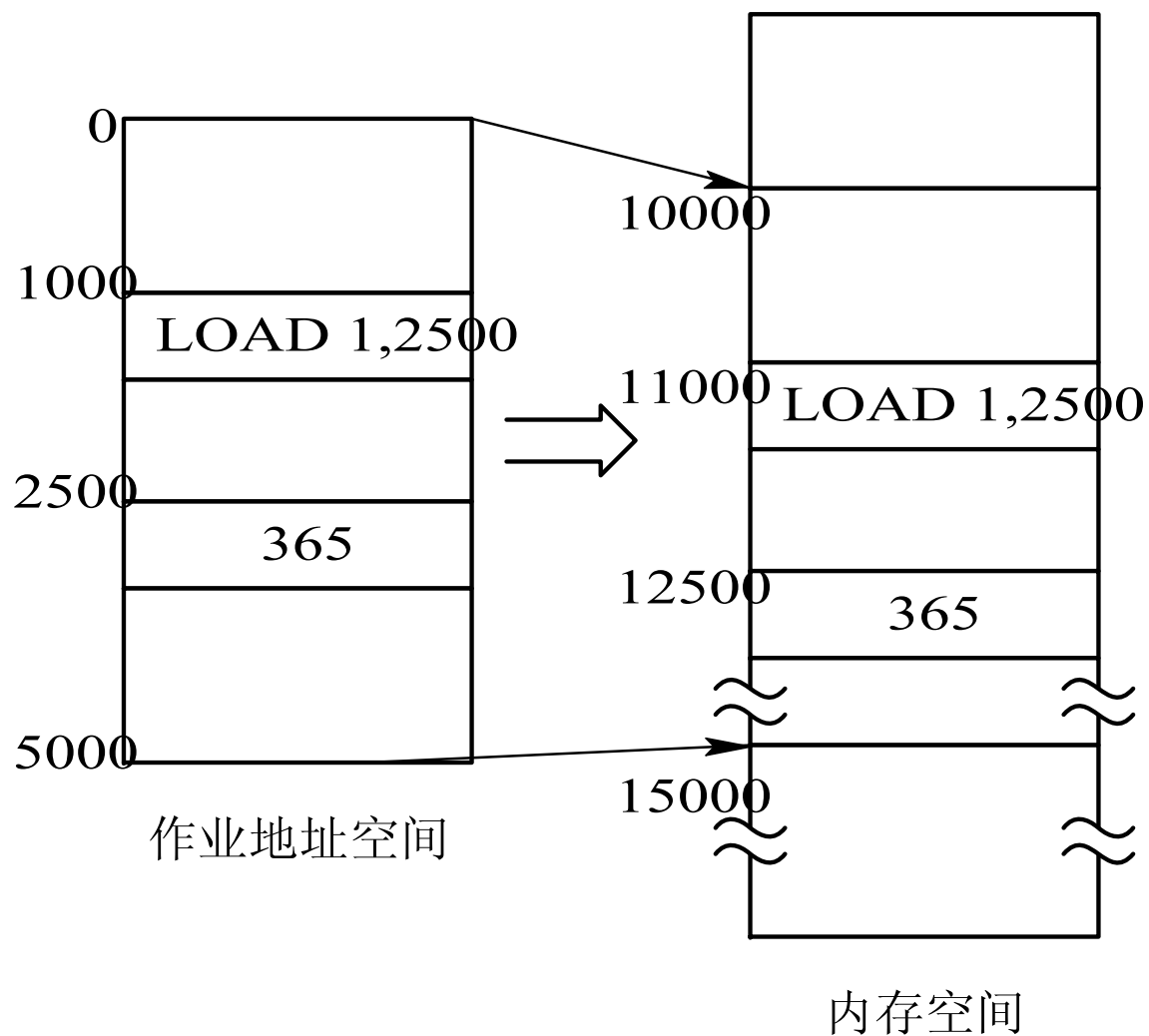


图4-3 作业装入内存时的情况



第四章 存储器管理



例如，在用户程序的1000号单元处有一条指令LOAD 1, 2500，该指令的功能是将2500单元中的整数365取至寄存器1。但若将该用户程序装入到内存的10000~15000号单元而不进行地址变换，则在执行11000号单元中的指令时，它将仍从2500号单元中把数据取至寄存器1而导致数据错误。由图4-3可见，正确的方法应该是将取数指令中的地址2500修改成12500，即把指令中的相对地址2500与本程序在内存中的起始地址10000相加，才得到正确的物理地址12500。除了数据地址应修改外，指令地址也须做同样的修改，即将指令的相对地址1000与起始地址10000相加，得到绝对地址11000。通常是在装入时对目标程序中指令和数据的修改过程称为重定位。又因为地址变换通常是在装入时一次完成的，以后不再改变，故称为静态重定位。





3. 动态运行时装入方式(Dynamic Run-time Loading)

可重定位装入方式可将装入模块装入到内存中任何允许的位置，故可用于多道程序环境；但这种方式并不允许程序运行时在内存中移动位置。因为，程序在内存中的移动，意味着它的物理位置发生了变化，这时必须对程序和数据的地址(是绝对地址)进行修改后方能运行。然而，实际情况是，在运行过程中它在内存中的位置可能经常要改变，此时就应采用动态运行时装入的方式。





4.2.2 程序的链接

根据链接时间的不同，可把链接分成如下三种：

(1) 静态链接。在程序运行之前，先将各目标模块及它们所需的库函数，链接成一个完整的装配模块，以后不再拆开。我们把这种事先进行链接的方式称为静态链接方式。

(2) 装入时动态链接。这是指将用户源程序编译后得到的一组目标模块，在装入内存时，采用边装入边链接的链接方式。

(3) 运行时动态链接。这是指对某些目标模块的链接，是在程序执行中需要该(目标)模块时，才对它进行的链接。





1. 静态链接方式(Static Linking)

(1) 对相对地址进行修改。在由编译程序所产生的所有目标模块中，使用的都是相对地址，其起始地址都为0，每个模块中的地址都是相对于起始地址计算的。在链接成一个装入模块后，原模块B和C在装入模块的起始地址不再是0，而分别是L和L+M，所以此时须修改模块B和C中的相对地址，即把原B中的所有相对地址都加上L，把原C中的所有相对地址都加上L+M。





第四章 存储器管理

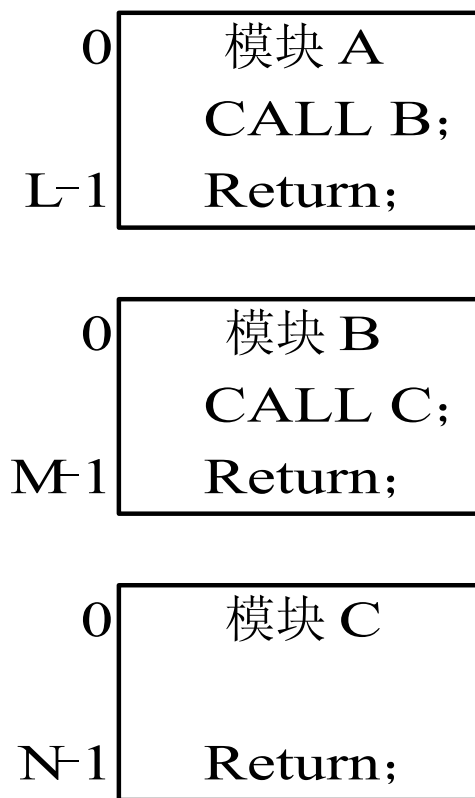


(2) 变换外部调用符号。将每个模块中所用的外部调用符号也都变换为相对地址，如把B的起始地址变换为L，把C的起始地址变换为L+M，如图4-4(b)所示。这种先进行链接所形成的一个完整的装入模块，又称为可执行文件。通常都不再拆开它，要运行时可直接将它装入内存。这种事先进行链接，以后不再拆开的链接方式，称为静态链接方式。

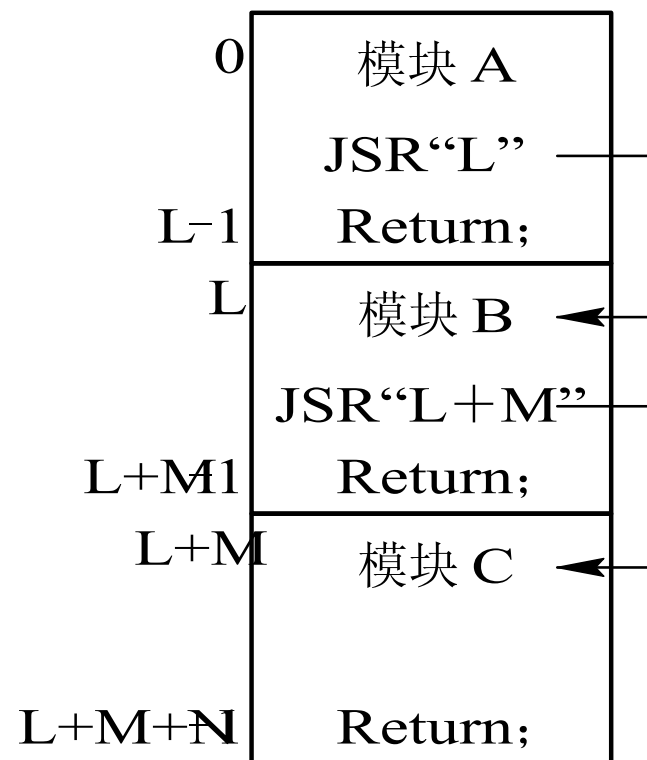




第四章 存储器管理



(a) 目标模块



(b) 装入模块

图 4-4 程序链接示意图



2. 装入时动态链接(Load-time Dynamic Linking)

用户源程序经编译后所得的目标模块，是在装入内存时边装入边链接的，即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要按照图4-4所示的方式来修改目标模块中的相对地址。装入时动态链接方式有以下优点：

(1) 便于修改和更新。对于经静态链接装配在一起的装入模块，如果要修改或更新其中的某个目标模块，则要求重新打开装入模块。这不仅是低效的，而且有时是不可能的。若采用动态链接方式，由于各目标模块是分开存放的，所以要修改或更新各目标模块是件非常容易的事。





第四章 存储器管理



(2) 便于实现对目标模块的共享。在采用静态链接方式时，每个应用模块都必须含有其目标模块的拷贝，无法实现对目标模块的共享。但采用装入时动态链接方式，OS则很容易将一个目标模块链接到几个应用模块上，实现多个应用程序对该模块的共享。





3. 运行时动态链接(Run-time Dynamic Linking)

在许多情况下，应用程序在运行时，每次要运行的模块可能是不相同的。但由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块都全部装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有些目标模块根本就不运行。比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。

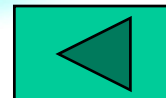




第四章 存储器管理



近几年流行起来的运行时动态链接方式，是对上述在装入时链接方式的一种改进。这种链接方式是将对某些模块的链接推迟到程序执行时才进行链接，亦即，在执行过程中，当发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将之装入内存，把它链接到调用者模块上。凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。





4.3 连续分配方式

4.3.1 单一连续分配

这是最简单的一种存储管理方式，但只能用于单用户、单任务的操作系统中。采用这种存储管理方式时，可把内存分为系统区和用户区两部分，系统区仅提供给OS使用，通常是放在内存的低址部分；用户区是指除系统区以外的全部内存空间，提供给用户使用。





第四章 存储器管理



虽然在早期的单用户、单任务操作系统中，有不少都配置了存储器保护机构，用于防止用户程序对操作系统的破坏，但近年来常见的几种单用户操作系统中，如CP/M、MS-DOS及RT11等，都未采取存储器保护措施。这是因为，一方面可以节省硬件，另一方面也因为这是可行的。在单用户环境下，机器由一用户独占，不可能存在其他用户干扰的问题；这时可能出现的破坏行为也只是用户程序自己去破坏操作系统，其后果并不严重，只是会影响该用户程序的运行，且操作系统也很容易通过系统的再启动而重新装入内存。





4.3.2 固定分区分配

1. 划分分区的方法

可用下述两种方法将内存的用户空间划分为若干个固定大小的分区：

(1) 分区大小相等，即使所有的内存分区大小相等。其缺点是缺乏灵活性，即当程序太小时，会造成内存空间的浪费；当程序太大时，一个分区又不足以装入该程序，致使该程序无法运行。尽管如此，这种划分方式仍被用于利用一台计算机去控制多个相同对象的场合，因为这些对象所需的内存空间是大小相等的。例如，炉温群控系统，就是利用一台计算机去控制多台相同的冶炼炉。





第四章 存储器管理



(2) 分区大小不等。为了克服分区大小相等而缺乏灵活性的这个缺点，可把内存区划分成含有多个较小的分区、适量的中等分区及少量的大分区。这样，便可根据程序的大小为之分配适当的分区。





2. 内存分配

为了便于内存分配，通常将分区按大小进行排队，并为之建立一张分区使用表，其中各表项包括每个分区的起始地址、大小及状态(是否已分配)，见图4-5(a)所示。当有一用户程序要装入时，由内存分配程序检索该表，从中找出一个能满足要求的、尚未分配的分区，将之分配给该程序，然后将该表项中的状态置为“已分配”；若未找到大小足够的分区，则拒绝为该用户程序分配内存。存储空间分配情况如图4-5(b)所示。





第四章 存储器管理



分区号	大小/KB	起址/KB	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

(a) 分区说明表



(b) 存储空间分配情况

图 4-5 固定分区使用表



4.3.3 动态分区分配

动态分区分配是根据进程的实际需要，动态地为之分配内存空间。在实现可变分区分配时，将涉及到分区分配中所用的数据结构、分区分配算法和分区的分配与回收操作这样三个问题。

1. 分区分配中的数据结构

为了实现分区分配，系统中必须配置相应的数据结构，用来描述空闲分区和已分配分区的情况，为分配提供依据。常用的数据结构有以下两种形式：





第四章 存储器管理

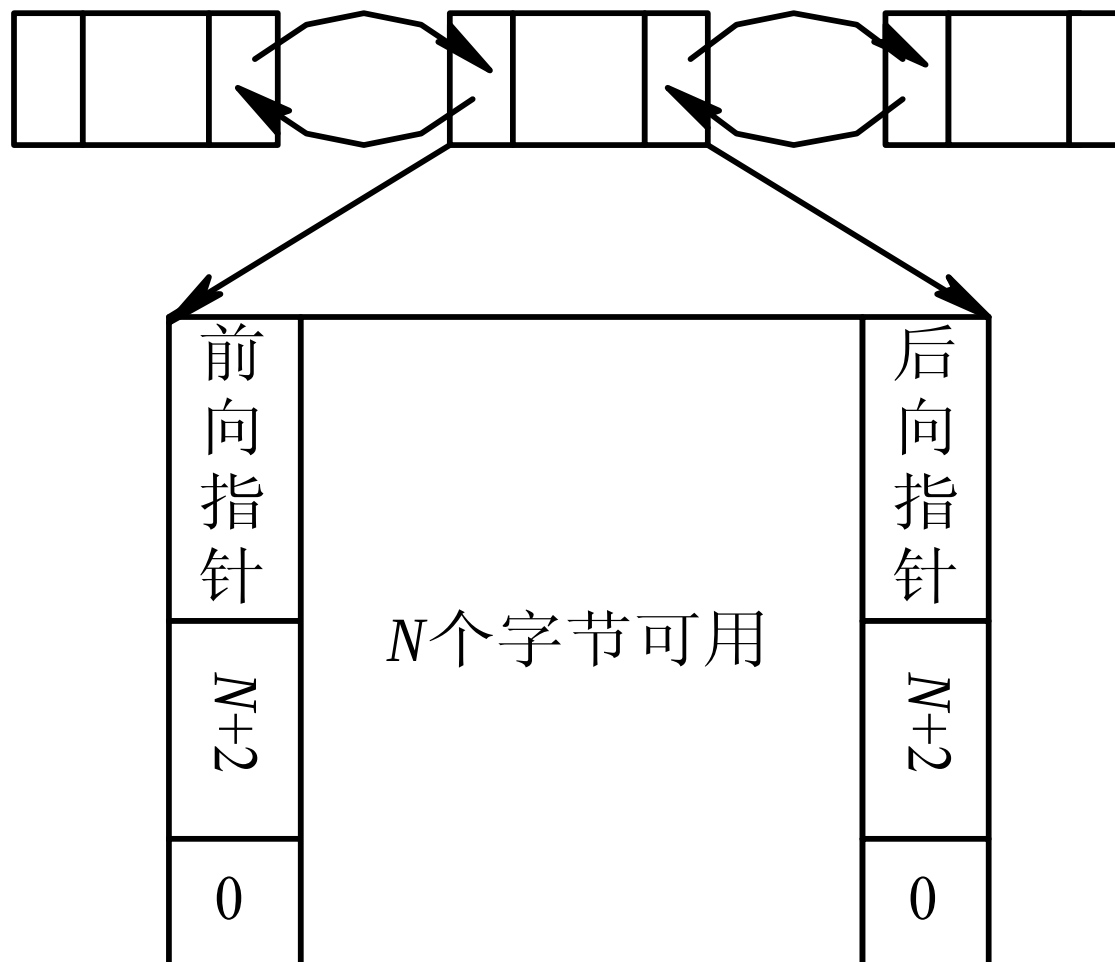


图4-6 空闲链结构



第四章 存储器管理



(1) 空闲分区表。在系统中设置一张空闲分区表，用于记录每个空闲分区的情况。每个空闲分区占一个表目，表目中包括分区序号、分区始址及分区的大小等数据项。

(2) 空闲分区链。为了实现对空闲分区的分配和链接，在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的前向指针；在分区尾部则设置一后向指针，通过前、后向链接指针，可将所有的空闲分区链接成一个双向链，如图4-6所示。为了检索方便，在分区尾部重复设置状态位和分区大小表目。当分区被分配出去以后，把状态位由“0”改为“1”，此时，前、后向指针已无意义。





2. 分区分配算法

1) 首次适应算法(first fit)

我们以空闲分区链为例来说明采用FF算法时的分配情况。FF算法要求空闲分区链以地址递增的次序链接。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止；然后再按照作业的大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲链中。若从链首直至链尾都不能找到一个能满足要求的分区，则此次内存分配失败，返回。该算法倾向于优先利用内存中低址部分的空间，从而保留了高址部分的大空闲区。这给以后到达的大作业分配大的内存空间创造了条件。其缺点是低址部分不断被划分，会留下许多难以利用的、很小的空闲分区，而每次查找又都是从低址部分开始，这无疑会增加查找可用空闲分区时的开销。



第四章 存储器管理



2) 循环首次适应算法(next fit)

该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。为实现该算法，应设置一起始查寻指针，用于指示下一次起始查寻的空闲分区，并采用循环查找方式，即如果最后一个(链尾)空闲分区的大小仍不能满足要求，则应返回到第一个空闲分区，比较其大小是否满足要求。找到后，应调整起始查寻指针。该算法能使内存中的空闲分区分布得更均匀，从而减少了查找空闲分区时的开销，但这样会缺乏大的空闲分区。





第四章 存储器管理



3) 最佳适应算法(best fit)

所谓“最佳”是指每次为作业分配内存时，总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。这样，第一次找到的能满足要求的空闲区，必然是最佳的。孤立地看，最佳适应算法似乎是最佳的，然而在宏观上却不一定。因为每次分配后所切割下来的剩余部分总是最小的，这样，在存储器中会留下许多难以利用的小空闲区。





第四章 存储器管理



4) 最坏适应算法(worst fit)

最坏适应分配算法要扫描整个空闲分区表或链表，总是挑选一个最大的空闲区分割给作业使用，其优点是可使剩下的空闲区不至于太小，产生碎片的几率最小，对中、小作业有利，同时最坏适应分配算法查找效率很高。该算法要求将所有的空闲分区按其容量以从大到小的顺序形成一空闲分区链，查找时只要看第一个分区能否满足作业要求。但是该算法的缺点也是明显的，它会使存储器中缺乏大的空闲分区。最坏适应算法与前面所述的首次适应算法、循环首次适应算法、最佳适应算法一起，也称为顺序搜索法。





5) 快速适应算法(quick fit)

该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，这样，系统中存在多个空闲分区链表，同时在内存中设立一张管理索引表，该表的每一个表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。空闲分区的分类是根据进程常用的空间大小进行划分，如2 KB、4 KB、8 KB等，对于其它大小的分区，如7 KB这样的空闲区，既可以放在8 KB的链表中，也可以放在一个特殊的空闲区链表中。





第四章 存储器管理



该算法的优点是查找效率高，仅需要根据进程的长度，寻找到能容纳它的最小空闲区链表，并取下第一块进行分配即可。另外该算法在进行空闲分区分配时，不会对任何分区产生分割，所以能保留大的分区，满足对大空间的需求，也不会产生内存碎片。

该算法的缺点是在分区归还主存时算法复杂，系统开销较大。此外，该算法在分配空闲分区时是以进程为单位，一个分区只属于一个进程，因此在为进程所分配的一个分区中，或多或少地存在一定的浪费。空闲分区划分越细，浪费则越严重，整体上会造成可观的存储空间浪费，这是典型的以空间换时间的作法。





3. 分区分配操作

1) 分配内存

系统应利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。设请求的分区大小为 $u.size$ ，表中每个空闲分区的大小可表示为 $m.size$ 。若 $m.size - u.size \leq size$ ($size$ 是事先规定的不再切割的剩余分区的大小)，说明多余部分太小，可不再切割，将整个分区分配给请求者；否则(即多余部分超过 $size$)，从该分区中按请求的大小划分出一块内存空间分配出去，余下的部分仍留在空闲分区链(表)中。然后，将分配区的首址返回给调用者。图4-7示出了分配流程。





第四章 存储器管理

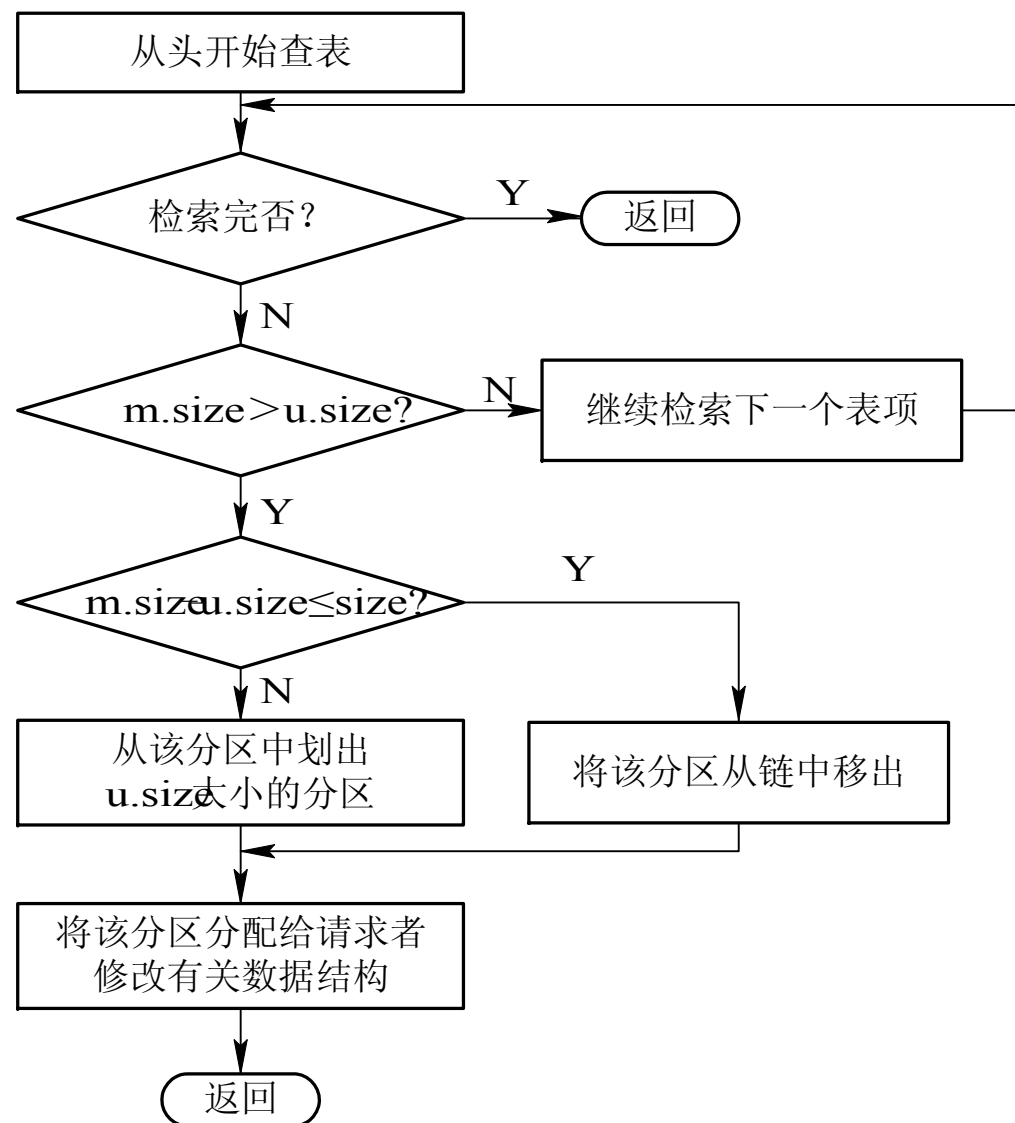


图 4-7 内存分配流程



2) 回收内存

当进程运行完毕释放内存时，系统根据回收区的首址，从空闲区链(表)中找到相应的插入点，此时可能出现以下四种情况之一：

(1) 回收区与插入点的前一个空闲分区F1相邻接，见图4-8(a)。此时应将回收区与插入点的前一分区合并，不必为回收分区分配新表项，而只需修改其前一分区F1的大小。

(2) 回收分区与插入点的后一空闲分区F2相邻接，见图4-8(b)。此时也可将两分区合并，形成新的空闲分区，但用回收区的首址作为新空闲区的首址，大小为两者之和。





第四章 存储器管理



(3) 回收区同时与插入点的前、后两个分区邻接，见图4-8(c)。此时将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和。

(4) 回收区既不与F1邻接，又不与F2邻接。这时应为回收区单独建立一新表项，填写回收区的首址和大小，并根据其首址插入到空闲链中的适当位置。





第四章 存储器管理

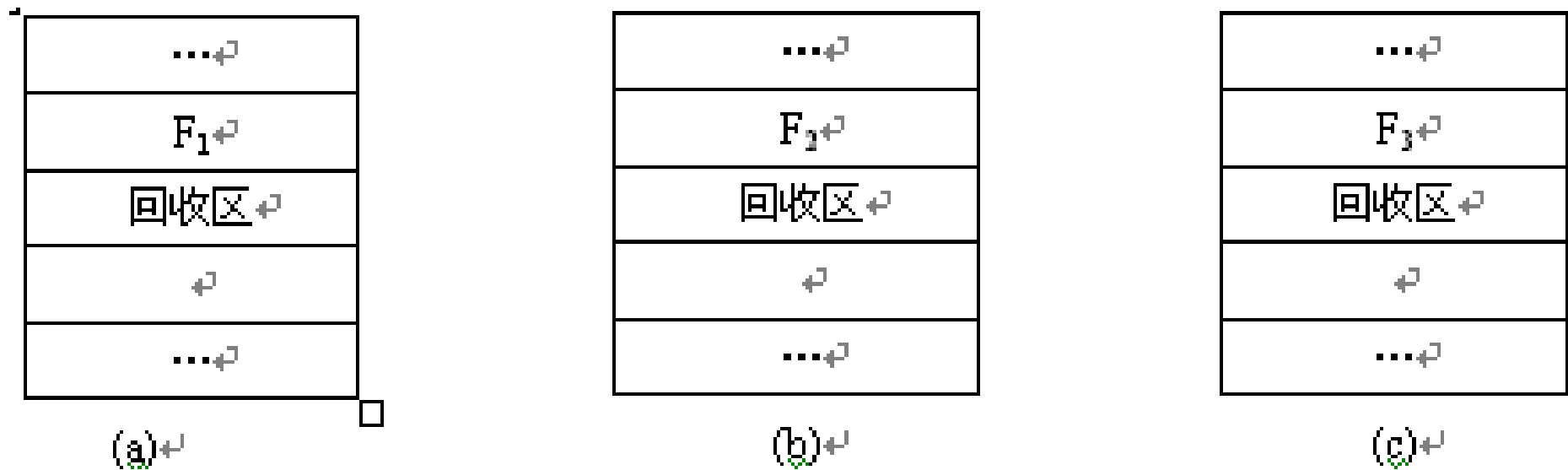


图 4-8 内存回收时的情况





4.3.4 伙伴系统

固定分区和动态分区方式都有不足之处。固定分区方式限制了活动进程的数目，当进程大小与空闲分区大小不匹配时，内存空间利用率很低。动态分区方式算法复杂，回收空闲分区时需要进行分区合并等，系统开销较大。伙伴系统方式是对以上两种内存方式的一种折衷方案。

伙伴系统规定，无论已分配分区或空闲分区，其大小均为2的 k 次幂， k 为整数， $1 \leq k \leq m$ ，其中： 2^1 表示分配的最小分区的大小， 2^m 表示分配的最大分区的大小，通常 2^m 是整个可分配内存的大小。





第四章 存储器管理



假设系统的可利用空间容量为 2^m 个字，则系统开始运行时，整个内存区是一个大小为 2^m 的空闲分区。在系统运行过程中，由于不断的划分，可能会形成若干个不连续的空闲分区，将这些空闲分区根据分区的大小进行分类，对于每一类具有相同大小的所有空闲分区，单独设立一个空闲分区双向链表。这样，不同大小的空闲分区形成了 $k(0 \leq k \leq m)$ 个空闲分区链表。





第四章 存储器管理



当需要为进程分配一个长度为 n 的存储空间时，首先计算一个 i 值，使 $2^{i-1} < n \leq 2^i$ ，然后在空闲分区大小为 2^i 的空闲分区链表中查找。若找到，即把该空闲分区分配给进程。否则，表明长度为 2^i 的空闲分区已经耗尽，则在分区大小为 2^{i+1} 的空闲分区链表中寻找。若存在 2^{i+1} 的一个空闲分区，则把该空闲分区分为相等的两个分区，这两个分区称为一对伙伴，其中的一个分区用于分配，而把另一个加入分区大小为 2^i 的空闲分区链表中。若大小为 2^{i+1} 的空闲分区也不存在，则需要查找大小为 2^{i+2} 的空闲分区，若找到则对其进行两次分割：第一次，将其分割为大小为 2^{i+1} 的两个分区，一个用于分配，一个加入到大小为 2^{i+1} 的空闲分区链表中；第二次，将第一次用于分配的空闲区分割为 2^i 的两个分区，一个用于分配，一个加入到大小为 2^i 的空闲分区链表中。若仍然找不到，则继续查找大小为 2^{i+3} 的空闲分区，以此类推。由此可见，在最坏的情况下，可能需要对 2^k 的空闲分区进行 k 次分割才能得到所需分区。



第四章 存储器管理



与一次分配可能要进行多次分割一样，一次回收也可能要进行多次合并，如回收大小为 2^i 的空闲分区时，若事先已存在 2^i 的空闲分区时，则应将其与伙伴分区合并为大小为 2^{i+1} 的空闲分区，若事先已存在 2^{i+1} 的空闲分区时，又应继续与其伙伴分区合并为大小为 2^{i+2} 的空闲分区，依此类推。

在伙伴系统中，其分配和回收的时间性能取决于查找空闲分区的位置和分割、合并空闲分区所花费的时间。与前面所述的多种方法相比较，由于该算法在回收空闲分区时，需要对空闲分区进行合并，所以其时间性能比前面所述的分类搜索算法差，但比顺序搜索算法好，而其空间性能则远优于前面所述的分类搜索法，比顺序搜索法略差。





4.3.5 哈希算法

在上述的分类搜索算法和伙伴系统算法中，都是将空闲分区根据分区大小进行分类，对于每一类具有相同大小的空闲分区，单独设立一个空闲分区链表。在为进程分配空间时，需要在一张管理索引表中查找到所需空间大小所对应的表项，从中得到对应的空闲分区链表表头指针，从而通过查找得到一个空闲分区。如果对空闲分区分类较细，则相应的空闲分区链表也较多，因此选择合适的空闲链表的开销也相应增加，且时间性能降低。





第四章 存储器管理



哈希算法就是利用哈希快速查找的优点，以及空闲分区在可利用空间表中的分布规律，建立哈希函数，构造一张以空闲分区大小为关键字的哈希表，该表的每一个表项记录了一个对应的空闲分区链表表头指针。

当进行空闲分区分配时，根据所需空闲分区大小，通过哈希函数计算，即得到在哈希表中的位置，从中得到相应的空闲分区链表，实现最佳分配策略。





4.3.6 可重定位分区分配

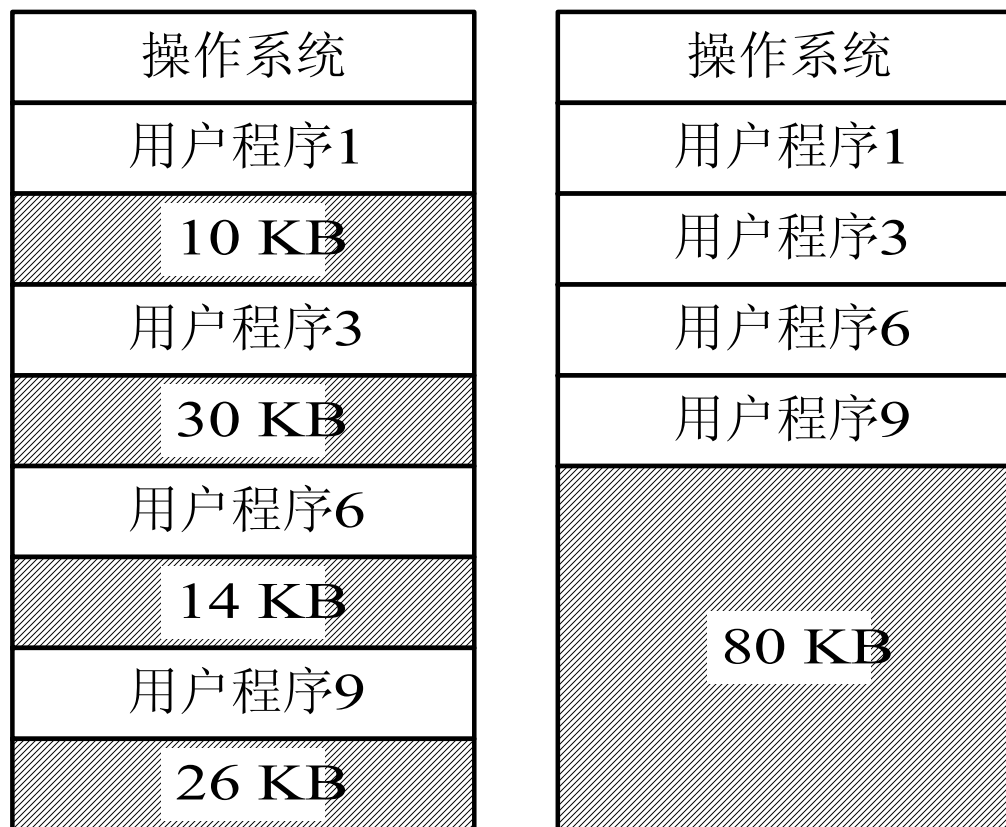
1. 动态重定位的引入

在连续分配方式中，必须把一个系统或用户程序装入一连续的内存空间。如果在系统中只有若干个小的分区，即使它们容量的总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。例如，图4-9(a)中示出了在内存中现有四个互不邻接的小分区，它们的容量分别为10 KB、30 KB、14 KB和26 KB，其总容量是80 KB。但如果现在有一作业到达，要求获得40 KB的内存空间，由于必须为它分配一连续空间，故此作业无法装入。这种不能被利用的小分区称为“零头”或“碎片”。





第四章 存储器管理



(a) 紧凑前

(b) 紧凑后

图4-9 紧凑的示意



第四章 存储器管理



若想把作业装入，可采用的一种方法是：将内存中的所有作业进行移动，使它们全都相邻接，这样，即可把原来分散的多个小分区拼接成一个大分区，这时就可把作业装入该区。这种通过移动内存中作业的位置，以把原来多个分散的小分区拼接成一个大分区的方法，称为“拼接”或“紧凑”，见图4-9(b)。由于经过紧凑后的某些用户程序在内存中的位置发生了变化，此时若不对程序和数据的地址加以修改(变换)，则程序必将无法执行。为此，在每次“紧凑”后，都必须对移动了的程序或数据进行重定位。





2. 动态重定位的实现

在动态运行时装入的方式中，作业装入内存后的所有地址都仍然是相对地址，将相对地址转换为物理地址的工作，被推迟到程序指令要真正执行时进行。为使地址的转换不会影响到指令的执行速度，必须有硬件地址变换机构的支持，即须在系统中增设一个重定位寄存器，用它来存放程序(数据)在内存中的起始地址。程序在执行时，真正访问的内存地址是相对地址与重定位寄存器中的地址相加而形成的。图4-10示出了动态重定位的实现原理。地址变换过程是在程序执行期间，随着对每条指令或数据的访问自动进行的，故称为动态重定位。当系统对内存进行了“紧凑”而使若干程序从内存的某处移至另一处时，不需对程序做任何修改，只要用该程序在内存的新起始地址，去置换原来的起始地址即可。



第四章 存储器管理

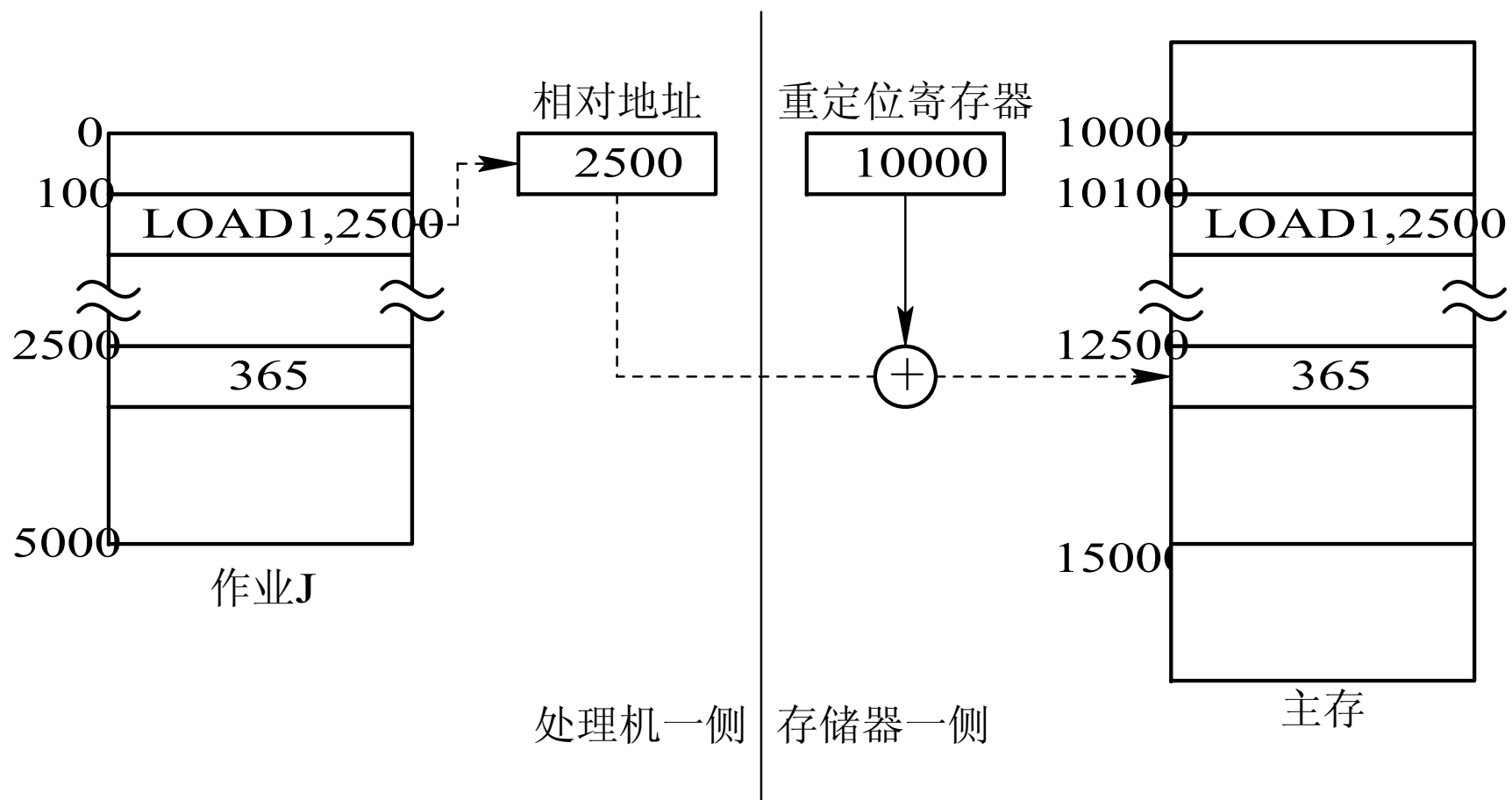


图 4-10 动态重定位示意图



3. 动态重定位分区分配算法

动态重定位分区分配算法与动态分区分配算法基本上相同，差别仅在于：在这种分配算法中，增加了紧凑的功能，通常，在找不到足够大的空闲分区来满足用户需求时进行紧凑。图4-11示出了动态重定位分区分配算法。





第四章 存储器管理

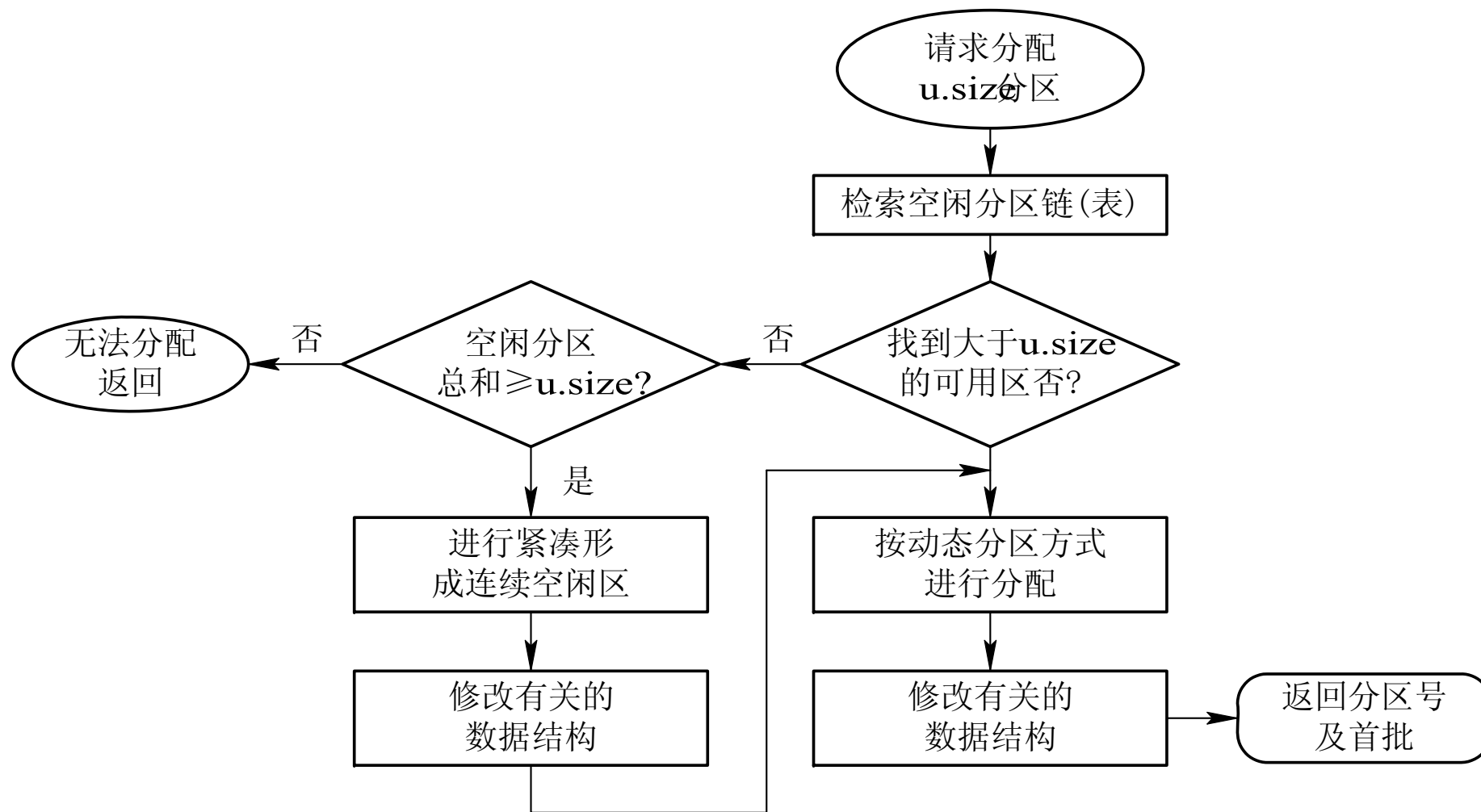


图4-11 动态分区分配算法流程图



4.3.7 对换

1. 对换(Swapping)的引入

在多道程序环境下，一方面，在内存中的某些进程由于某事件尚未发生而被阻塞运行，但它却占用了大量的内存空间，甚至有时可能出现在内存中所有进程都被阻塞而迫使CPU停止下来等待的情况；另一方面，却又有着许多作业在外存上等待，因无内存而不能进入内存运行的情况。显然这对系统资源是一种严重的浪费，且使系统吞吐量下降。为了解决这一问题，在系统中又增设了对换(也称交换)设施。所谓“对换”，是指把内存中暂时不能运行的进程或者暂时不用的程序和数据调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据调入内存。对换是提高内存利用率的有效措施。自从在60年代初期出现“对换”技术后，它便引起了人们的重视，现在该技术已被广泛地应用于操作系统中。



第四章 存储器管理



如果对换是以整个进程为单位的，便称之为“整体对换”或“进程对换”。这种对换被广泛地应用于分时系统中，其目的是用来解决内存紧张问题，并可进一步提高内存的利用率。而如果对换是以“页”或“段”为单位进行的，则分别称之为“页面对换”或“分段对换”，又统称为“部分对换”。这种对换方法是实现后面要讲到的请求分页和请求分段式存储管理的基础，其目的是为了支持虚拟存储系统。在此，我们只介绍进程对换，而分页或分段对换将放在虚拟存储器中介绍。为了实现进程对换，系统必须能实现三方面的功能：对换空间的管理、进程的换出，以及进程的换入。





2. 对换空间的管理

在具有对换功能的OS中，通常把外存分为文件区和对换区。前者用于存放文件，后者用于存放从内存换出的进程。由于通常的文件都是较长久地驻留在外存上，故对文件区管理的主要目标，是提高文件存储空间的利用率，为此，对文件区采取离散分配方式。然而，进程在对换区中驻留的时间是短暂的，对换操作又较频繁，故对对换空间管理的主要目标，是提高进程换入和换出的速度。为此，采取的是连续分配方式，较少考虑外存中的碎片问题。





第四章 存储器管理



为了能对对换区中的空闲盘块进行管理，在系统中应配置相应的数据结构，以记录外存的使用情况。其形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表中的每个表目中应包含两项，即对换区的首址及其大小，分别用盘块号和盘块数表示。

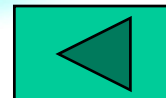




3. 进程的换出与换入

(1) 进程的换出。每当一进程由于创建子进程而需要更多的内存空间，但又无足够的内存空间等情况发生时，系统应将某进程换出。其过程是：系统首先选择处于阻塞状态且优先级最低的进程作为换出进程，然后启动磁盘，将该进程的程序和数据传送到磁盘的对换区上。若传送过程未出现错误，便可回收该进程所占用的内存空间，并对该进程的进程控制块做相应的修改。

(2) 进程的换入。系统应定时地查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将其中换出时间最久(换出到磁盘上)的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。





4.4 基本分页存储管理方式

4.4.1 页面与页表

1. 页面

1) 页面和物理块

分页存储管理是将一个进程的逻辑地址空间分成若干个大小的相等的片，称为页面或页，并为各页加以编号，从0开始，如第0页、第1页等。相应地，也把内存空间分成与页面相同大小的若干个存储块，称为(物理)块或页框(frame)，也同样为它们加以编号，如0#块、1#块等等。在为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于进程的最后一页经常装不满一块而形成了不可利用的碎片，称之为“页内碎片”。



2) 页面大小

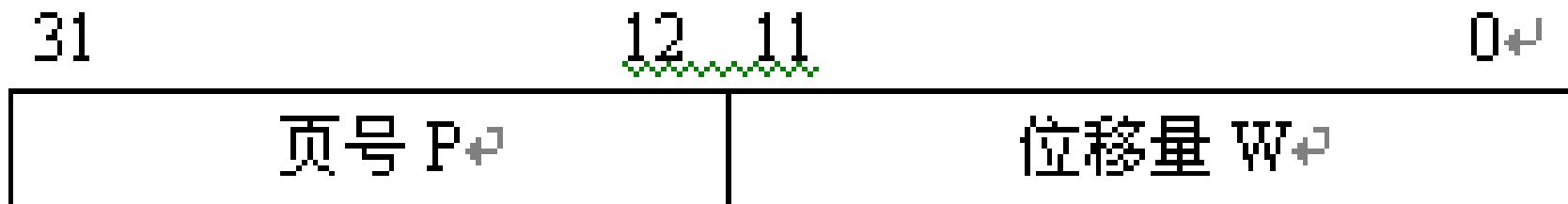
在分页系统中的页面其大小应适中。页面若太小，一方面虽然可使内存碎片减小，从而减少了内存碎片的总空间，有利于提高内存利用率，但另一方面也会使每个进程占用较多的页面，从而导致进程的页表过长，占用大量内存；此外，还会降低页面换进换出的效率。然而，如果选择的页面较大，虽然可以减少页表的长度，提高页面换进换出的速度，但却又会使页内碎片增大。因此，页面的大小应选择适中，且页面大小应是2的幂，通常为512 B~8 KB。





2. 地址结构

分页地址中的地址结构如下：



它含有两部分：前一部分为页号P，后一部分为位移量W(或称为页内地址)。图中的地址长度为32位，其中0~11位为页内地址，即每页的大小为4 KB；12~31位为页号，地址空间最多允许有1 M页。





第四章 存储器管理



对于某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

$$P = \text{INT} \left[\frac{A}{L} \right]$$
$$d = [A] \text{MOD} L$$

其中，INT是整除函数，MOD是取余函数。例如，其系统的页面大小为1 KB，设A = 2170 B，则由上式可以求得P = 2，d = 122。





3. 页表

在分页系统中，允许将进程的各个页离散地存储在内存不同的物理块中，但系统应能保证进程的正确运行，即能在内存中找到每个页面所对应的物理块。为此，系统又为每个进程建立了一张页面映像表，简称页表。在进程地址空间内的所有页($0 \sim n$)，依次在页表中有一页表项，其中记录了相应页在内存中对应的物理块号，见图4-12的中间部分。在配置了页表后，进程执行时，通过查找该表，即可找到每页在内存中的物理块号。可见，页表的作用是实现从页号到物理块号的地址映射。





第四章 存储器管理

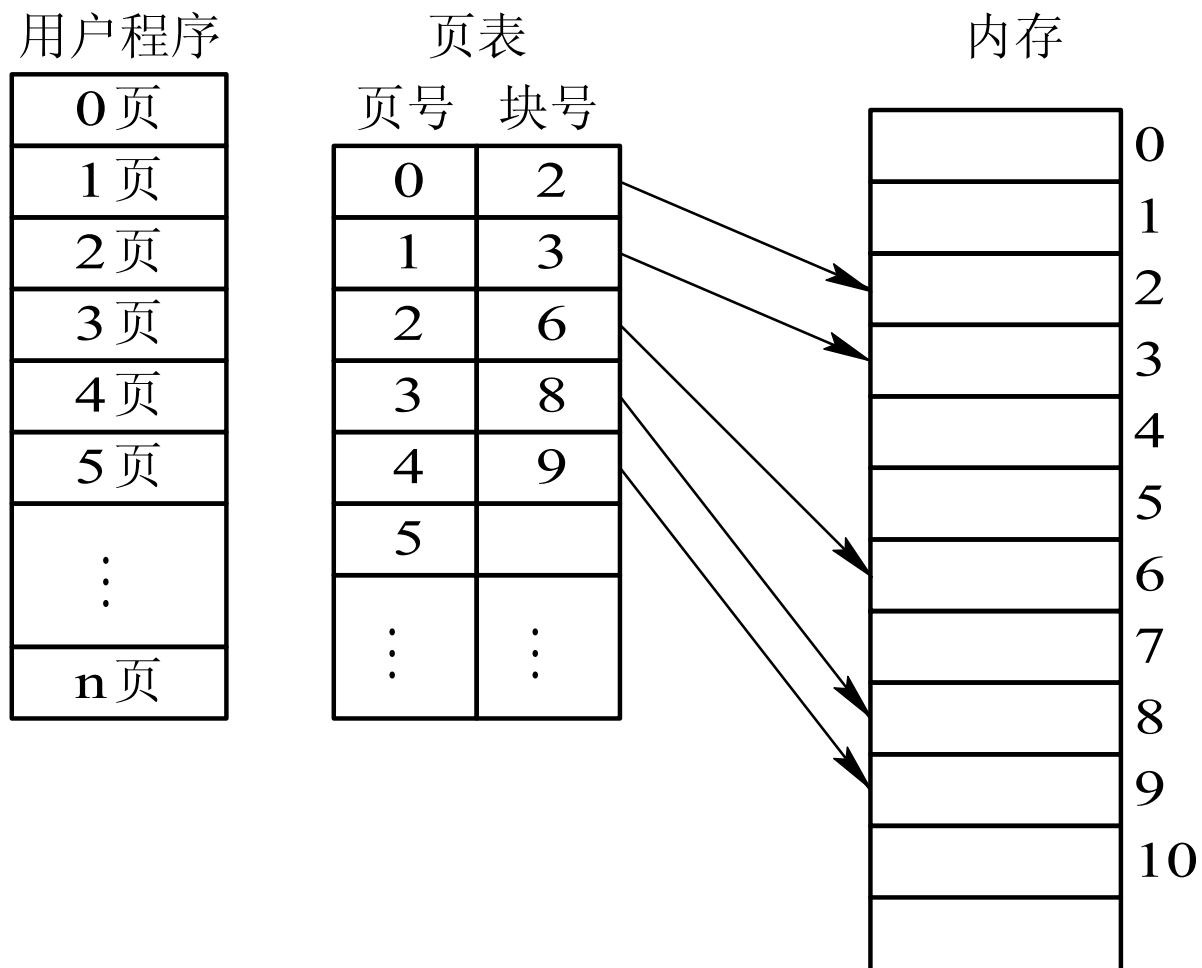


图4-12 页表的作用



4.4.2 地址变换机构

1. 基本的地址变换机构

页表的功能可以由一组专门的寄存器来实现。一个页表项用一个寄存器。由于寄存器具有较高的访问速度，因而有利于提高地址变换的速度；但由于寄存器成本较高，且大多数现代计算机的页表又可能很大，使页表项的总数可达几千甚至几十万个，显然这些页表项不可能都用寄存器来实现，因此，页表大多驻留在内存中。在系统中只设置一个页表寄存器PTR(Page-Table Register)，在其中存放页表在内存的始址和页表的长度。平时，进程未执行时，页表的始址和页表长度存放在本进程的PCB中。当调度程序调度到某进程时，才将这两个数据装入页表寄存器中。因此，在单处理机环境下，虽然系统中可以运行多个进程，但只需一个页表寄存器。



第四章 存储器管理



当进程要访问某个逻辑地址中的数据时，分页地址变换机构会自动地将有效地址(相对地址)分为页号和页内地址两部分，再以页号为索引去检索页表。查找操作由硬件执行。在执行检索之前，先将页号与页表长度进行比较，如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址空间。于是，这一错误将被系统发现并产生一地址越界中断。若未出现越界错误，则将页表始址与页号和页表项长度的乘积相加，便得到该表项在页表中的位置，于是可从中得到该页的物理块号，将之装入物理地址寄存器中。与此同时，再将有效地址寄存器中的页内地址送入物理地址寄存器的块内地址字段中。这样便完成了从逻辑地址到物理地址的变换。图4-13示出了分页系统的地址变换机构。





第四章 存储器管理

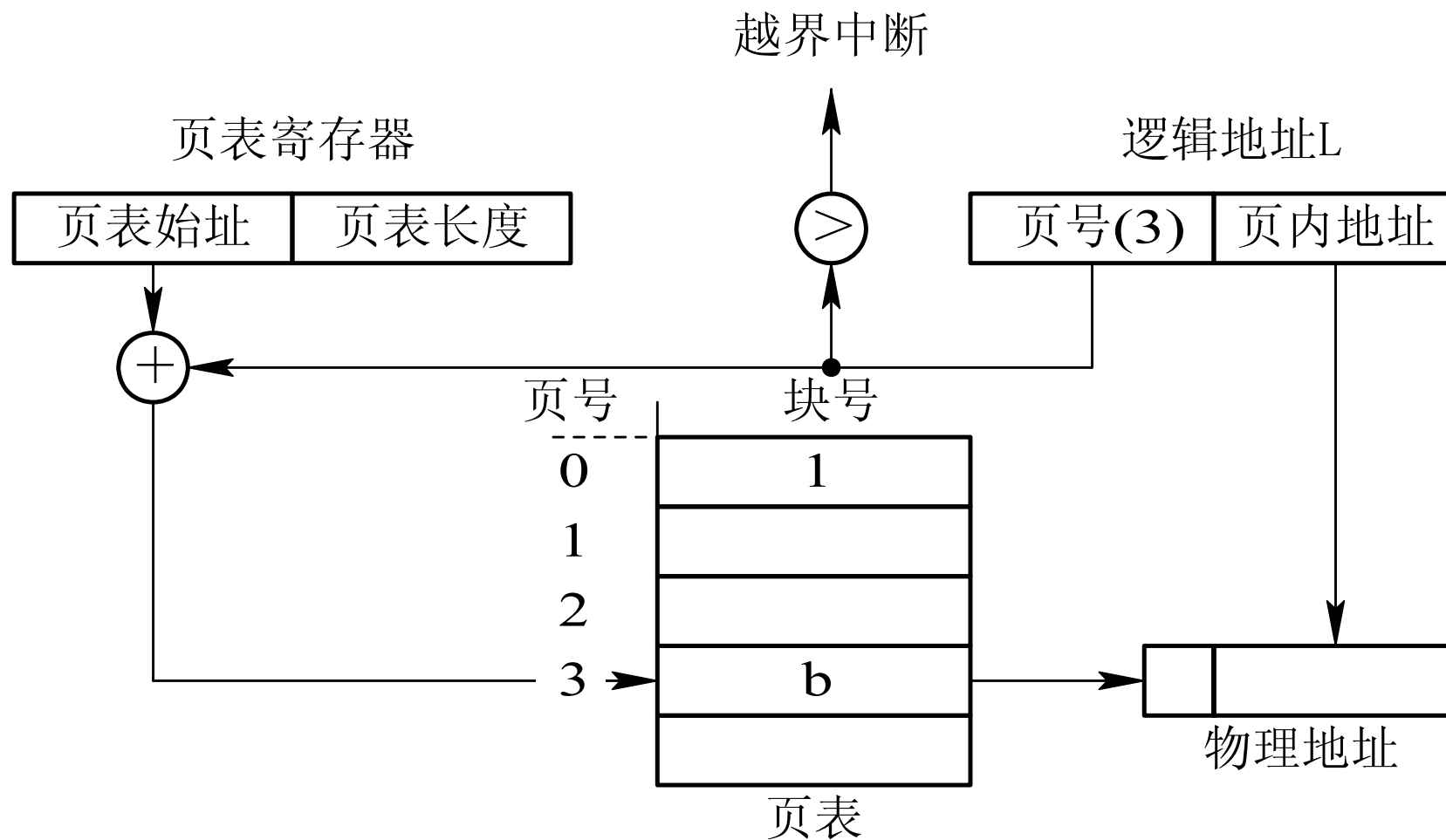


图4-13 分页系统的地址变换机构



2. 具有快表的地址变换机构

由于页表是存放在内存中的，这使CPU在每存取一个数据时，都要两次访问内存。第一次是访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量 W 拼接，以形成物理地址。第二次访问内存时，才是从第一次所得地址中获得所需数据(或向此地址中写入数据)。因此，采用这种方式将使计算机的处理速度降低近 $1/2$ 。可见，以此高昂代价来换取存储器空间利用率的提高，是得不偿失的。





第四章 存储器管理



为了提高地址变换速度，可在地址变换机构中增设一个具有并行查寻能力的特殊高速缓冲寄存器，又称为“联想寄存器”(Associative Memory)，或称为“快表”，在IBM系统中又取名为TLB(Translation Lookaside Buffer)，用以存放当前访问的那些页表项。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动地将页号P送入高速缓冲寄存器，并将此页号与高速缓存中的所有页号进行比较，若其中有与此相匹配的页号，便表示所要访问的页表项在快表中。于是，可直接从快表中读出该页所对应的物理块号，并送到物理地址寄存器中。如在块表中未找到对应的页表项，则还须再访问内存中的页表，找到后，把从页表项中读出的物理块号送地址寄存器；同时，再将此页表项存入快表的一个寄存器单元中，亦即，重新修改快表。但如果联想寄存器已满，则OS必须找到一个老的且已被认为不再需要的页表项，将它换出。图4-14示出了具有快表的地址变换机构。



第四章 存储器管理

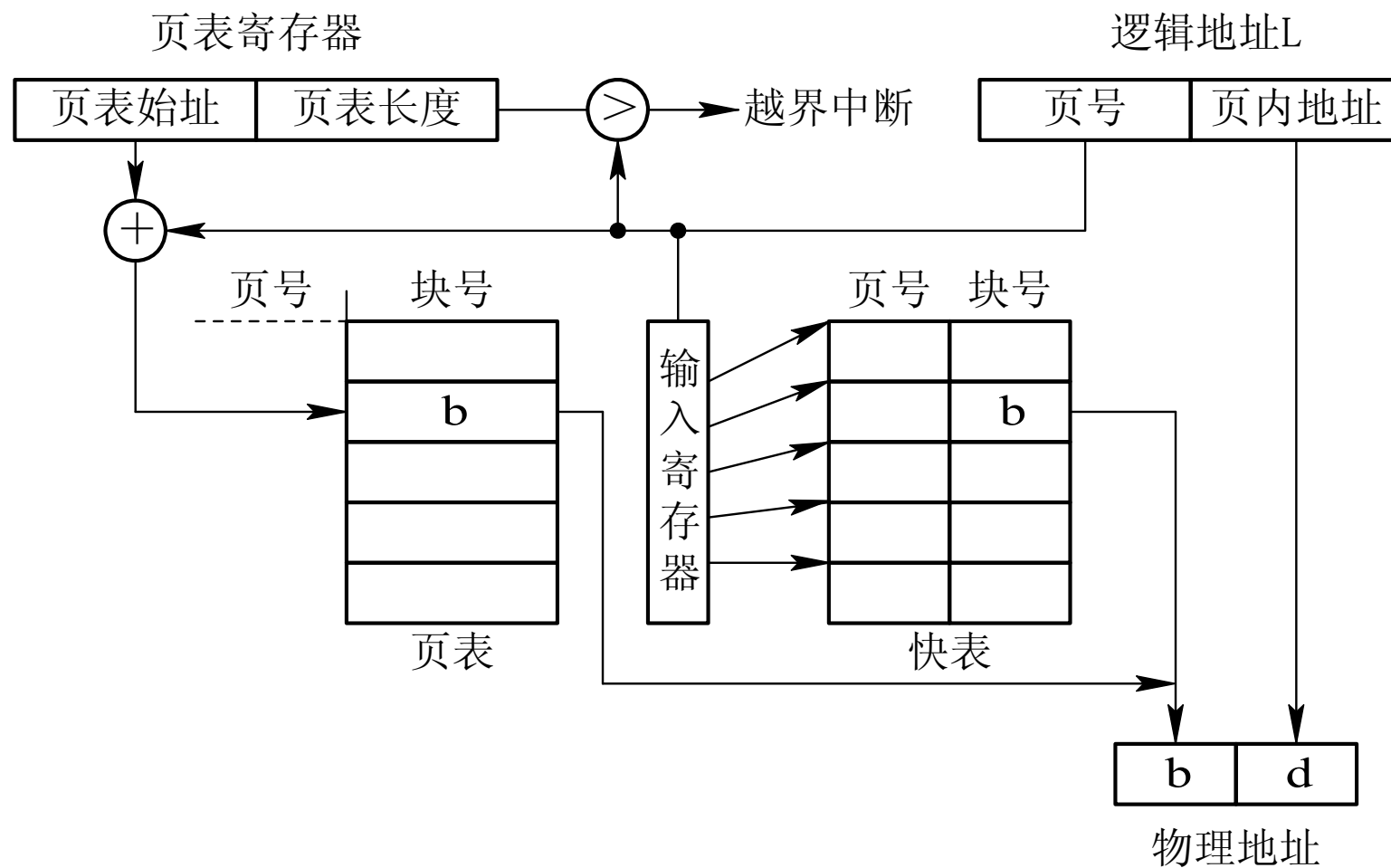


图4-14 具有快表的地址变换机构



4.4.3 两级和多级页表

现代的大多数计算机系统，都支持非常大的逻辑地址空间($2^{32} \sim 2^{64}$)。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。例如，对于一个具有32位逻辑地址空间的分页系统，规定页面大小为4 KB即 2^{12} B，则在每个进程页表中的页表项可达1兆个之多。又因为每个页表项占用一个字节，故每个进程仅仅其页表就要占用1 MB的内存空间，而且还要求是连续的。显然这是不现实的，我们可以采用下述两个方法来解决这一问题：

(1) 采用离散分配方式来解决难以找到一块连续的大内存空间的问题；

(2) 只将当前需要的部分页表项调入内存，其余的页表项仍驻留在磁盘上，需要时再调入。





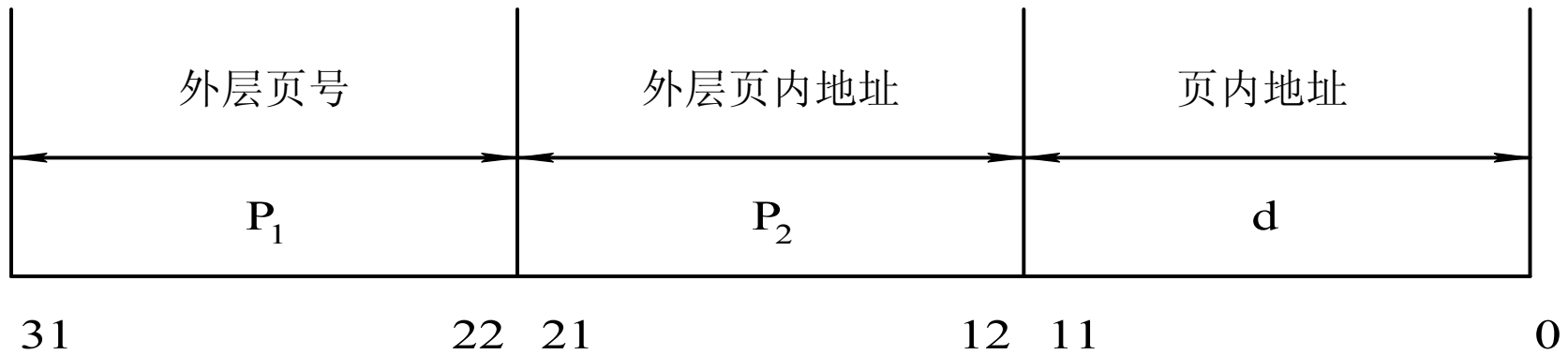
1. 两级页表(Two-Level Page Table)

对于要求连续的内存空间来存放页表的问题，可利用将页表进行分页，并离散地将各个页面分别存放在不同的物理块中的办法来加以解决，同样也要为离散分配的页表再建立一张页表，称为外层页表(Outer Page Table)，在每个页表项中记录了页表页面的物理块号。下面我们仍以前面的32位逻辑地址空间为例来说明。当页面大小为 4 KB时(12位)，若采用一级页表结构，应具有20位的页号，即页表项应有1兆个；在采用两级页表结构时，再对页表进行分页，使每页中包含 2^{10} (即1024)个页表项，最多允许有 2^{10} 个页表分页；或者说，外层页表中的外层页内地址P2为10位，外层页号P1也为10位。此时的逻辑地址结构可描述如下：





第四章 存储器管理





第四章 存储器管理



由图可以看出，在页表的每个表项中存放的是进程的某页在内存中的物理块号，如第0#页存放在1#物理块中；1#页存放在4#物理块中。而在外层页表的每个页表项中，所存放的是某页表分页的首址，如第0#页表是存放在第1011#物理块中。我们可以利用外层页表和页表这两级页表，来实现从进程的逻辑地址到内存中物理地址间的变换。





第四章 存储器管理

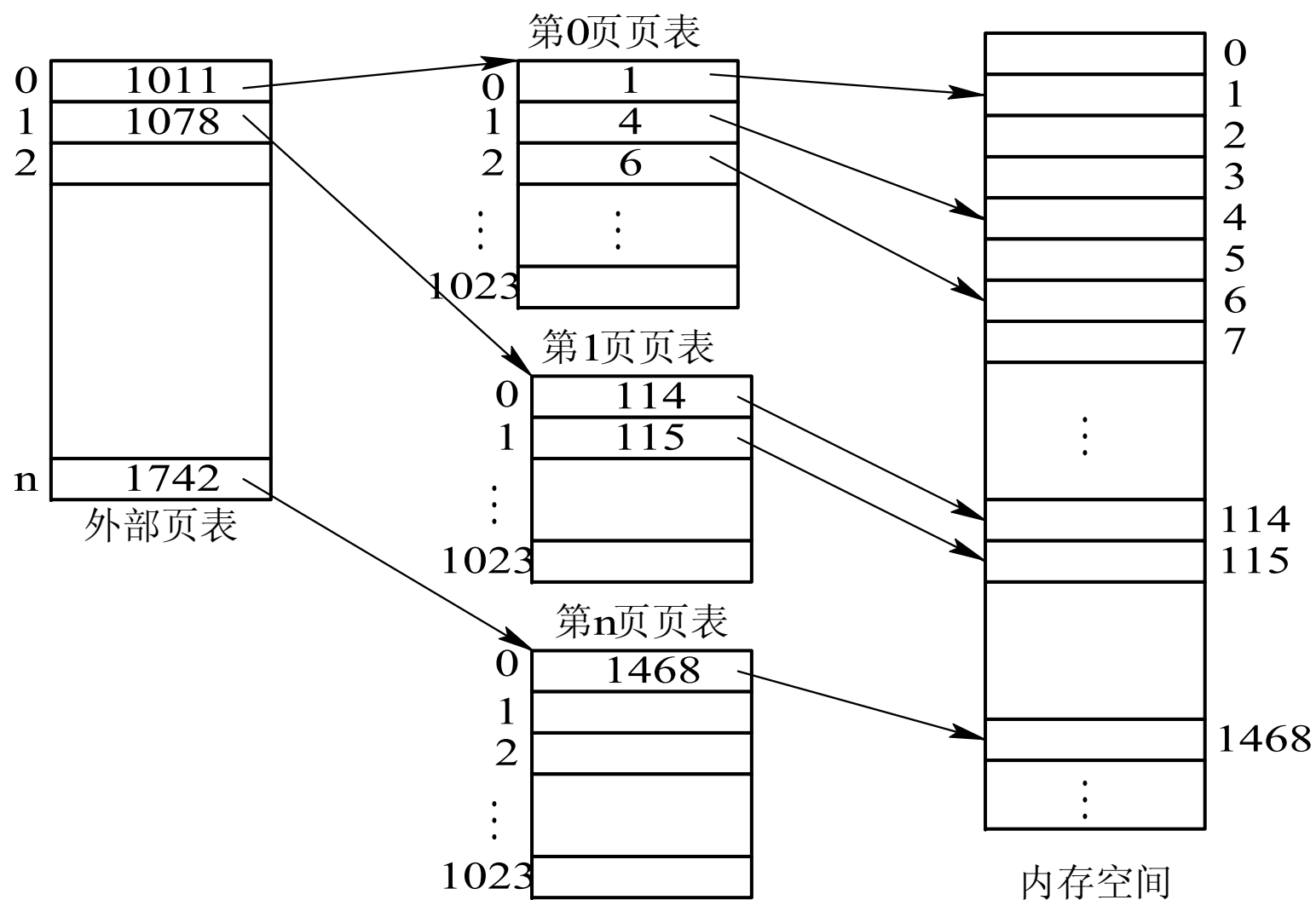


图4-15 两级页表结构



第四章 存储器管理



为了地址变换实现上的方便起见，在地址变换机构中同样需要增设一个外层页表寄存器，用于存放外层页表的始址，并利用逻辑地址中的外层页号，作为外层页表的索引，从中找到指定页表分页的始址，再利用P2作为指定页表分页的索引，找到指定的页表项，其中即含有该页在内存的物理块号，用该块号和页内地址d即可构成访问的内存物理地址。图4-16示出了两级页表时的地址变换机构。





第四章 存储器管理

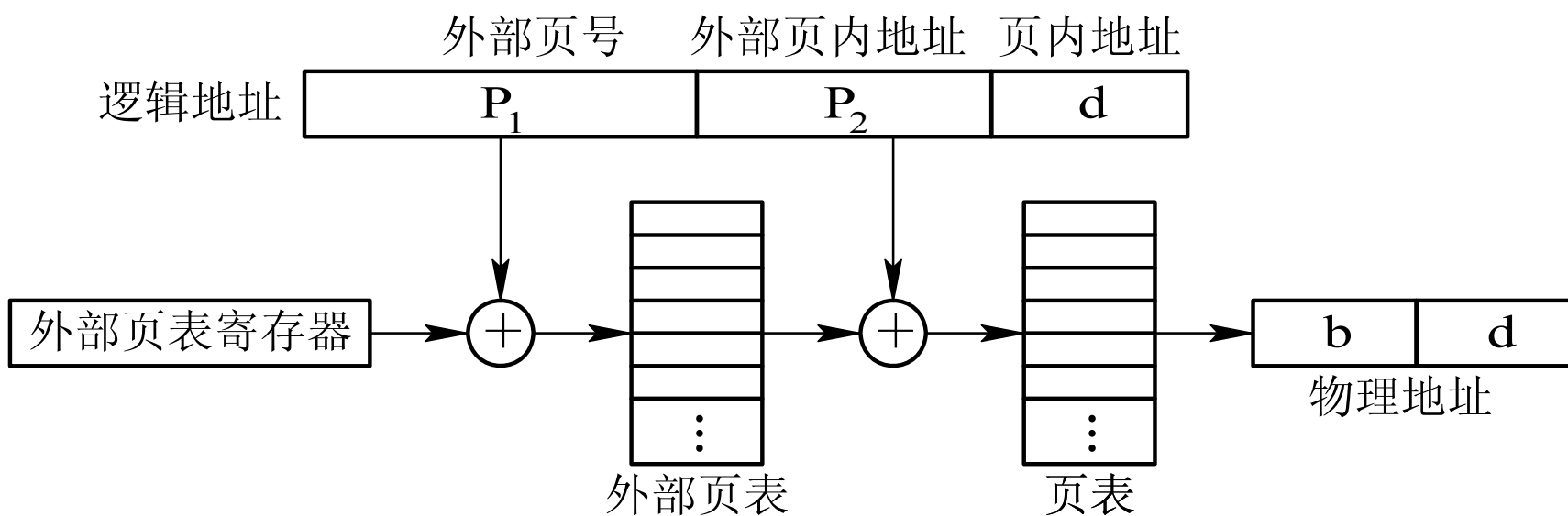


图4-16 具有两级页表的地址变换机构



2. 多级页表

对于32位的机器，采用两级页表结构是合适的；但对于64位的机器，采用两级页表是否仍可适用的问题，须做以下简单分析。如果页面大小仍采用4 KB即 2^{12} B，那么还剩下52位，假定仍按物理块的大小(2^{12} 位)来划分页表，则将余下的42位用于外层页号。此时在外层页表中可能有4096 G个页表项，要占用16 384 GB的连续内存空间。这样的结果显然是不能令人接受的，因此必须采用多级页表，将外层页表再进行分页，也就是将各分页离散地装入到不相邻接的物理块中，再利用第2级的外层页表来映射它们之间的关系。

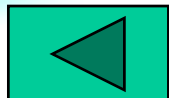




第四章 存储器管理



对于64位的计算机，如果要求它能支持 2^{64} B(= 1 844 744 TB)规模的物理存储空间，则即使是采用三级页表结构也是难以办到的；而在当前的实际应用中也无此必要。故在近两年推出的64位OS中，把可直接寻址的存储器空间减少为45位长度(即 2^{45})左右，这样便可利用三级页表结构来实现分页存储管理。





4.5 基本分段存储管理方式

4.5.1 分段存储管理方式的引入

1) 方便编程

通常，用户把自己的作业按照逻辑关系划分为若干个段，每个段都是从0开始编址，并有自己的名字和长度。因此，希望要访问的逻辑地址是由段名(段号)和段内偏移量(段内地址)决定的。例如，下述的两条指令便是使用段名和段内地址：

LOAD 1, [A] | <D> ;

STORE 1, [B] | <C> ;

其中，前一条指令的含义是将分段A中D单元内的值读入寄存器1；后一条指令的含义是将寄存器1的内容存入B分段的C单元中。



第四章 存储器管理



2) 信息共享

在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。比如，共享某个例程和函数。分页系统中的“页”只是存放信息的物理单位(块)，并无完整的意义，不便于实现共享；然而段却是信息的逻辑单位。由此可知，为了实现段的共享，希望存储管理能与用户程序分段的组织方式相适应。





第四章 存储器管理



3) 信息保护

信息保护同样是对信息的逻辑单位进行保护，因此，分段管理方式能更有效和方便地实现信息保护功能。

4) 动态增长

在实际应用中，往往有些段，特别是数据段，在使用过程中会不断地增长，而事先又无法确切地知道数据段会增长到多大。前述的其它几种存储管理方式，都难以应付这种动态增长的情况，而分段存储管理方式却能较好地解决这一问题。





第四章 存储器管理



5) 动态链接

动态链接是指在作业运行之前，并不把几个目标程序段链接起来。要运行时，先将主程序所对应的目标程序装入内存并启动运行，当运行过程中又需要调用某段时，才将该段(目标程序)调入内存并进行链接。可见，动态链接也要求以段作为管理的单位。





4.5.2 分段系统的基本原理

1. 分段

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等，如图4-17所示。每个段都有自己的名字。为了实现简单起见，通常可用一个段号来代替段名，每个段都从0开始编址，并采用一段连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而各段长度不等。整个作业的地址空间由于是分成多个段，因而是二维的，亦即，其逻辑地址由段号(段名)和段内地址所组成。

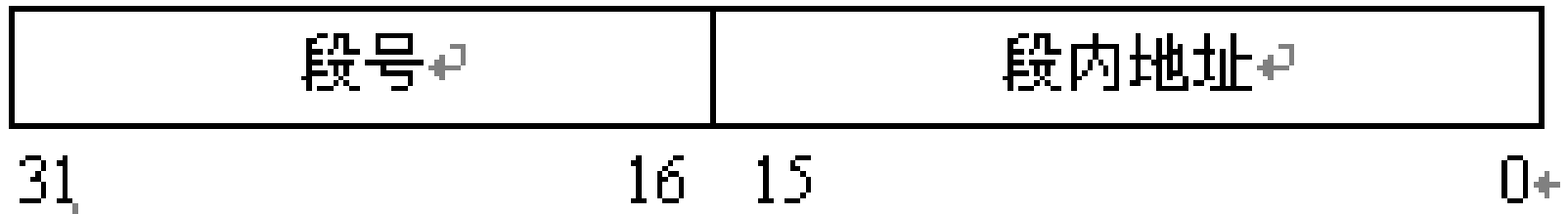




第四章 存储器管理



分段地址中的地址具有如下结构：





第四章 存储器管理



在该地址结构中，允许一个作业最长有 64 K个段，每个段的最大长度为64 KB。

分段方式已得到许多编译程序的支持，编译程序能自动地根据源程序的情况而产生若干个段。例如，Pascal编译程序可以为全局变量、用于存储相应参数及返回地址的过程调用栈、每个过程或函数的代码部分、每个过程或函数的局部变量等等，分别建立各自的段。类似地，Fortran编译程序可以为公共块(Common block)建立单独的段，也可以为数组分配一个单独的段。装入程序将装入所有这些段，并为每个段赋予一个段号。





2. 段表

在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在分段式存储管理系统中，则是为每个分段分配一个连续的分区，而进程中的各个段可以离散地移入内存中不同的分区中。为使程序能正常运行，亦即，能从物理内存中找出每个逻辑段所对应的位置，应像分页系统那样，在系统中为每个进程建立一张段映射表，简称“段表”。每个段在表中占有一个表项，其中记录了该段在内存中的起始地址(又称为“基址”)和段的长度，如图4-17所示。段表可以存放在一组寄存器中，这样有利于提高地址转换速度，但更常见的是将段表放在内存中。



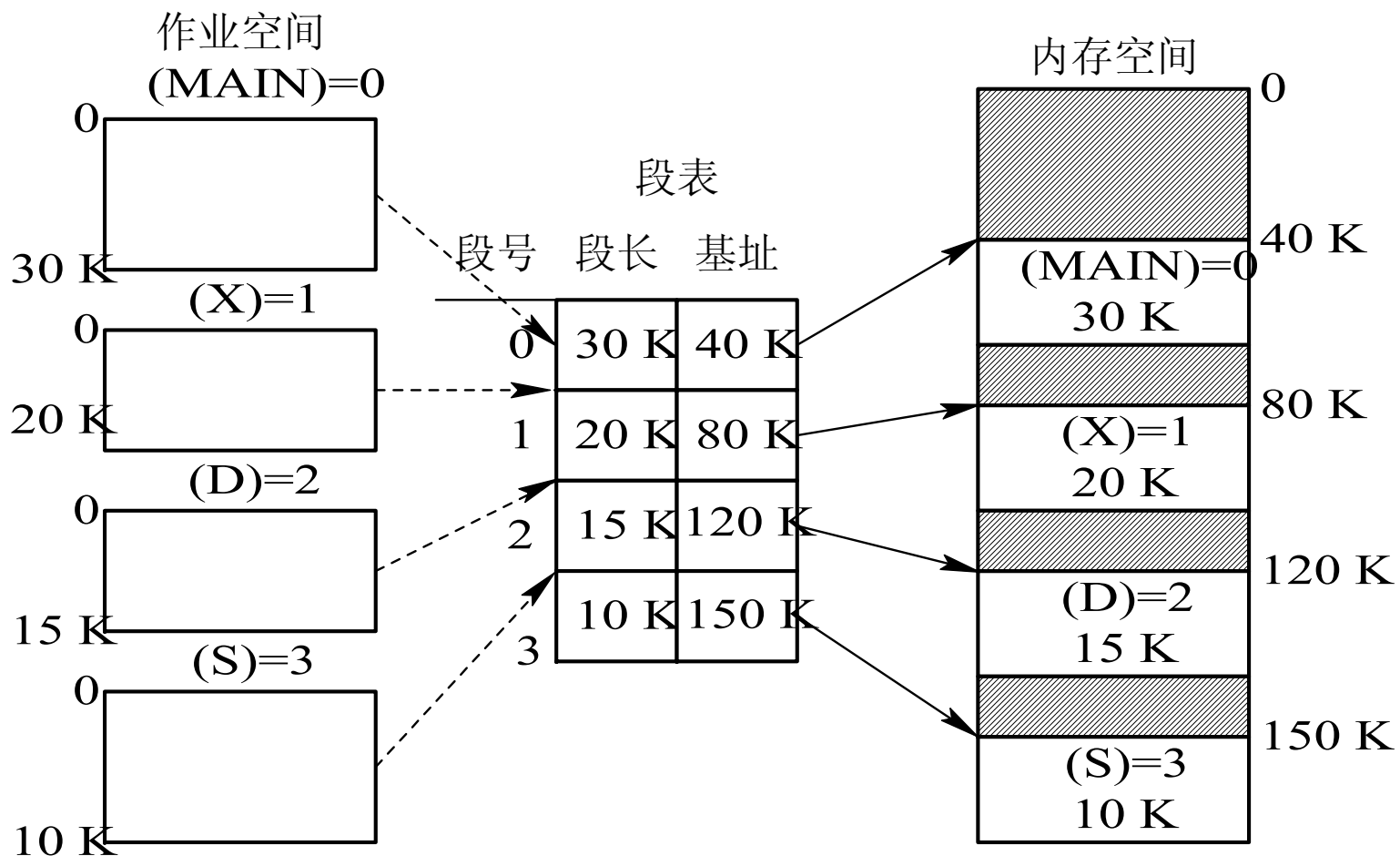


图4-17 利用段表实现地址映射



3. 地址变换机构

为了实现从进程的逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表始址和段表长度TL。在进行地址变换时，系统将逻辑地址中的段号与段表长度TL进行比较。若 $S > TL$ ，表示段号太大，是访问越界，于是产生越界中断信号；若未越界，则根据段表的始址和该段的段号，计算出该段对应段表项的位置，从中读出该段在内存的起始地址，然后，再检查段内地址d是否超过该段的段长SL。若超过，即 $d > SL$ ，同样发出越界中断信号；若未越界，则将该段的基址d与段内地址相加，即可得到要访问的内存物理地址。

图4-18示出了分段系统的地址变换过程。





第四章 存储器管理

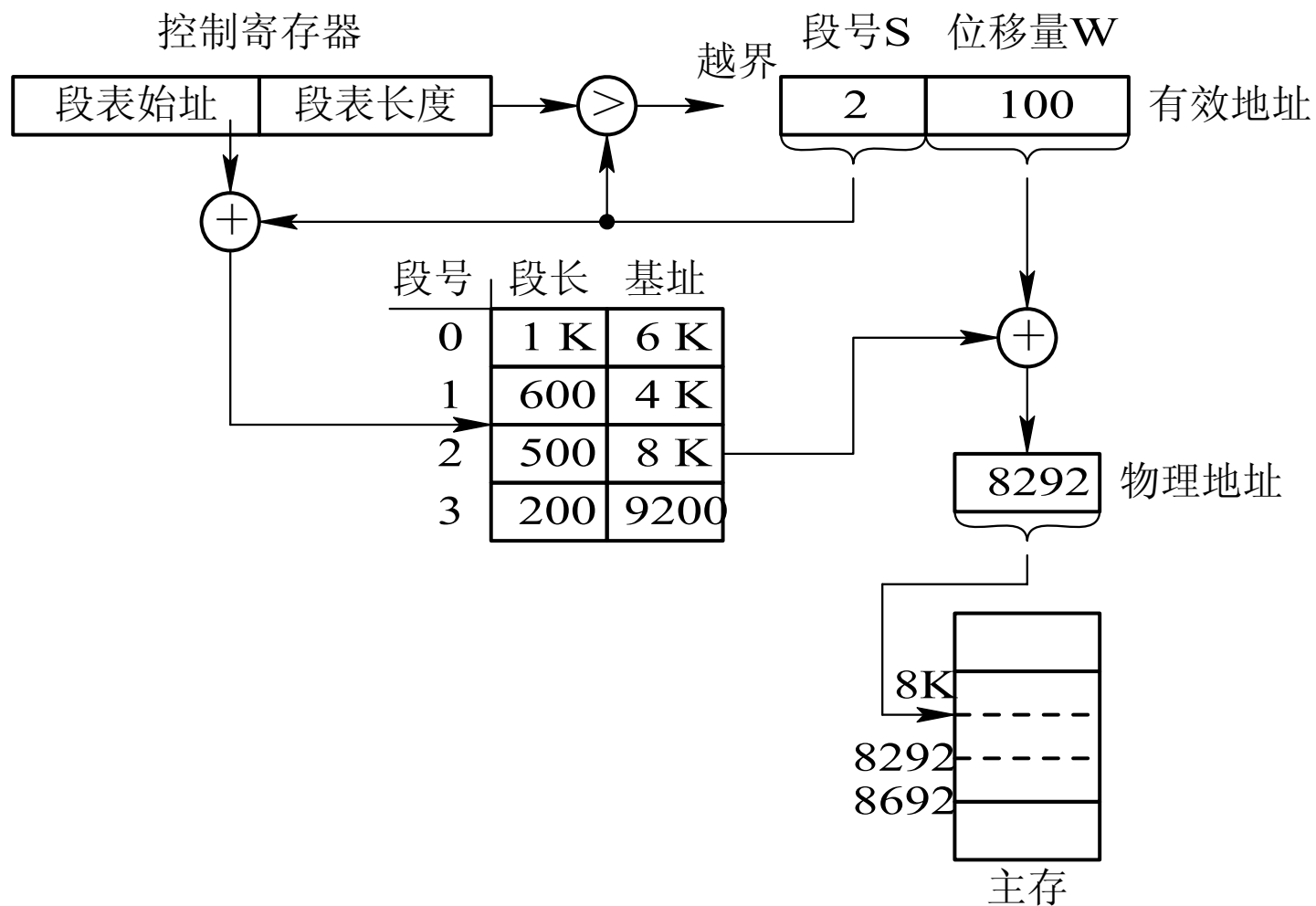


图4-18 分段系统的地址变换过程



第四章 存储器管理



像分页系统一样，当段表放在内存中时，每要访问一个数据，都须访问两次内存，从而极大地降低了计算机的速率。解决的方法也和分页系统类似，再增设一个联想存储器，用于保存最近常用的段表项。由于一般情况是段比页大，因而段表项的数目比页表项的数目少，其所需的联想存储器也相对较小，便可以显著地减少存取数据的时间，比起没有地址变换的常规存储器的存取速度来仅慢约10%~15%。





4. 分页和分段的主要区别

由上所述不难看出，分页和分段系统有许多相似之处。比如，两者都采用离散分配方式，且都要通过地址映射机构来实现地址变换。但在概念上两者完全不同，主要表现在下述三个方面。

(1) 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要而不是用户的需要。段则是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。





第四章 存储器管理



(2) 页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

(3) 分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址；而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。





4.5.3 信息共享

分段系统的一个突出优点，是易于实现段的共享，即允许若干个进程共享一个或多个分段，且对段的保护也十分简单易行。在分页系统中，虽然也能实现程序和数据的共享，但远不如分段系统来得方便。我们通过一个例子来说明这个问题。例如，有一个多用户系统，可同时接纳40个用户，他们都执行一个文本编辑程序(Text Editor)。如果文本编辑程序有160 KB的代码和另外40 KB的数据区，则总共需有 8 MB 的内存空间来支持40个用户。如果160 KB的代码是可重入的(Reentrant)，则无论是在分页系统还是在分段系统中，该代码都能被共享，在内存中只需保留一份文本编辑程序的副本，此时所需的内存空间仅为1760 KB($40 \times 40 + 160$)，而不是8000 KB。





第四章 存储器管理



假定每个页面的大小为4 KB，那么，160 KB的代码将占用40个页面，数据区占10个页面。为实现代码的共享，应在每个进程的页表中都建立40个页表项，它们的物理块号都是21#~60#。在每个进程的页表中，还须为自己的数据区建立页表项，它们的物理块号分别是61#~70#、71#~80#、81#~90#，...，等等。图4-19是分页系统中共享editor的示意图。





第四章 存储器管理

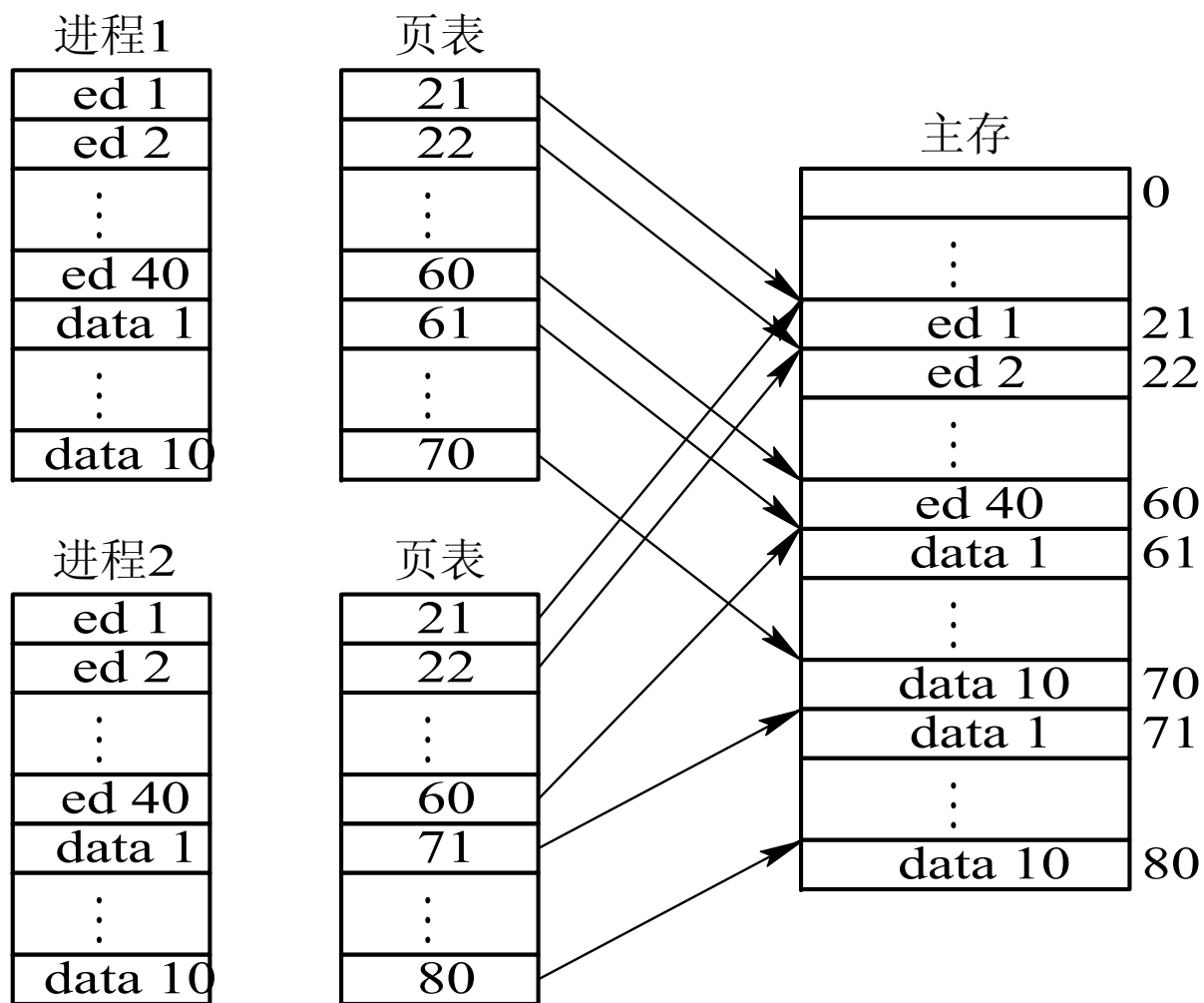


图4-19 分页系统中共享editor的示意图



第四章 存储器管理



在分段系统中，实现共享则容易得多，只需在每个进程的段表中为文本编辑程序设置一个段表项。图4-20是分段系统中共享editor的示意图。

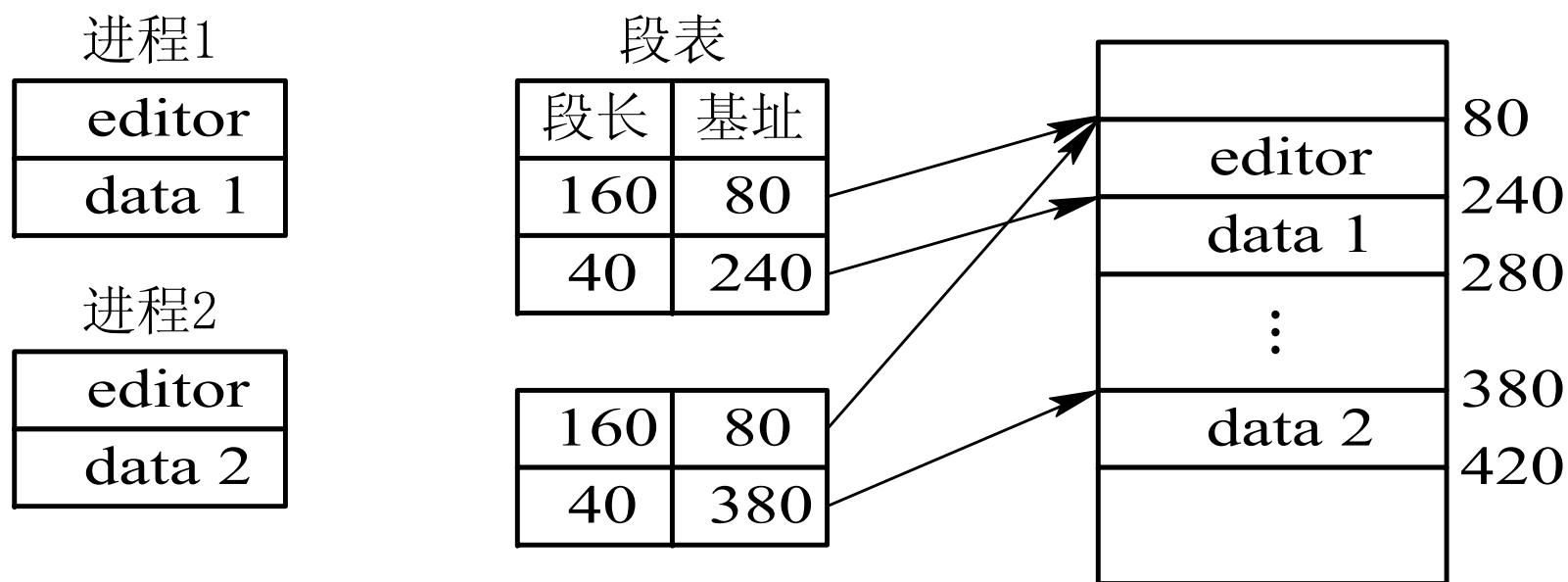


图 4-20 分段系统中共享editor的示意图



第四章 存储器管理



可重入代码(Reentrant Code)又称为“纯代码”(Pure Code), 是一种允许多个进程同时访问的代码。为使各个进程所执行的代码完全相同, 绝对不允许可重入代码在执行中有任何改变。因此, 可重入代码是一种不允许任何进程对它进行修改的代码。但事实上, 大多数代码在执行时都可能有些改变, 例如, 用于控制程序执行次数的变量以及指针、信号量及数组等。为此, 在每个进程中, 都必须配以局部数据区, 把在执行中可能改变的部分拷贝到该数据区, 这样, 程序在执行时, 只需对该数据区(属于该进程私有)中的内容进行修改, 并不去改变共享的代码, 这时的可共享代码即成为可重入码。





4.5.4 段页式存储管理方式

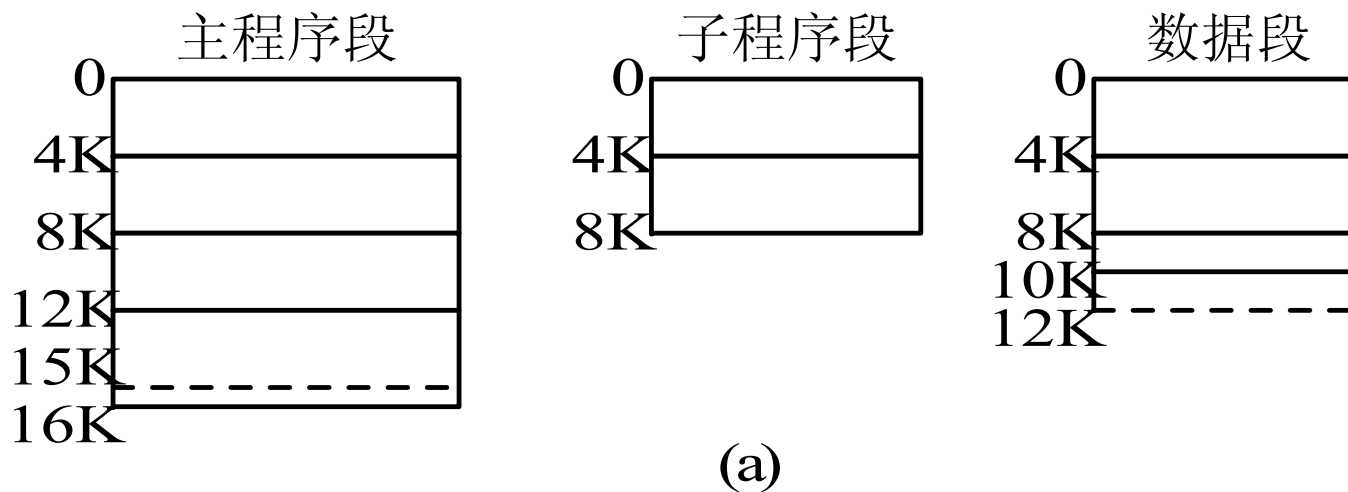
1. 基本原理

段页式系统的基本原理，是分段和分页原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。图4-21示出了一个作业地址空间的结构。该作业有三个段，页面大小为4 KB。在段页式系统中，其地址结构由段号、段内页号及页内地址三部分所组成，如图4-22所示。





第四章 存储器管理



(b)

图 4-21 作业地址空间和地址结构





第四章 存储器管理

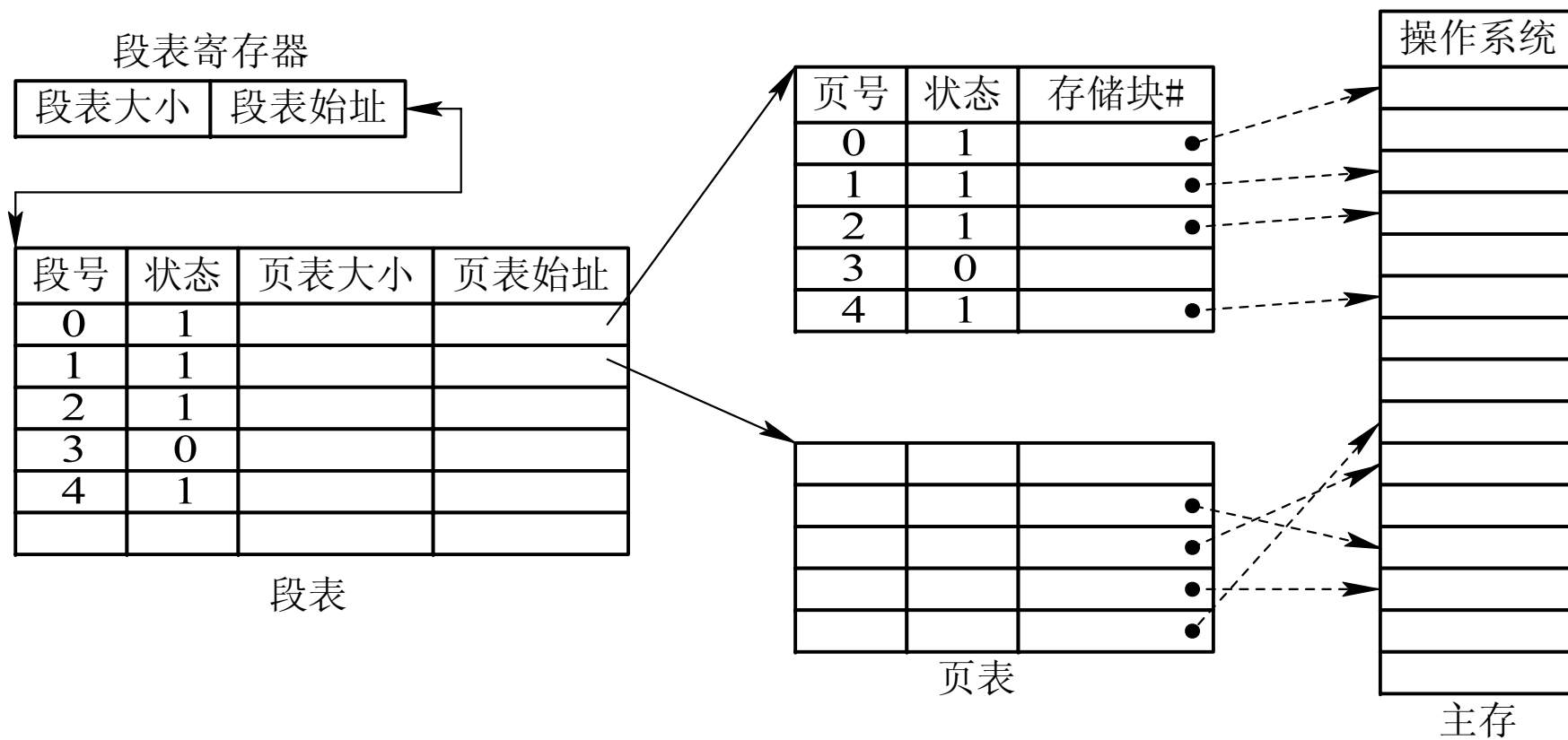


图4-22 利用段表和页表实现地址映射



2. 地址变换过程

在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中存放段表始址和段表长TL。进行地址变换时，首先利用段号S，将它与段表长TL进行比较。若 $S < TL$ ，表示未越界，于是利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，并利用逻辑地址中的段内页号P来获得对应页的页表项位置，从中读出该页所在的物理块号b，再利用块号b和页内地址来构成物理地址。图4-23示出了段页式系统中的地址变换机构。





第四章 存储器管理

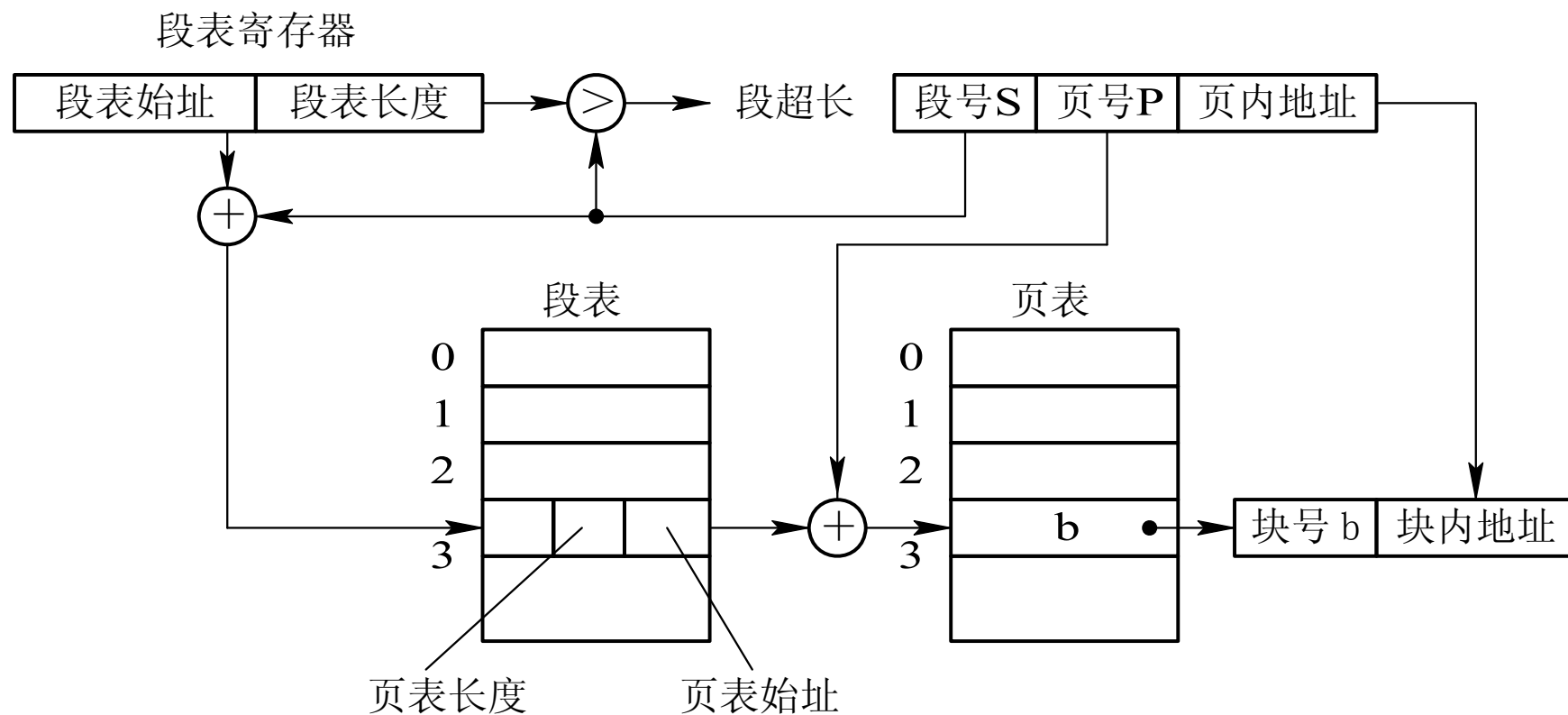


图4-23 段页式系统中的地址变换机构

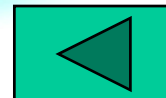


第四章 存储器管理



在段页式系统中，为了获得一条指令或数据，须三次访问内存。第一次访问是访问内存中的段表，从中取得页表始址；第二次访问是访问内存中的页表，从中取出该页所在的物理块号，并将该块号与页内地址一起形成指令或数据的物理地址；第三次访问才是真正从第二次访问所得的地址中，取出指令或数据。

显然，这使访问内存的次数增加了近两倍。为了提高执行速度，在地址变换机构中增设一个高速缓冲寄存器。每次访问它时，都须同时利用段号和页号去检索高速缓存，若找到匹配的表项，便可从中得到相应页的物理块号，用来与页内地址一起形成物理地址；若未找到匹配表项，则仍须再三次访问内存。





4.6 虚拟存储器的基本概念

前面所介绍的各种存储器管理方式有一个共同的特点，即它们都要求将一个作业全部装入内存后方能运行，于是，出现了下面这样两种情况：

(1) 有的作业很大，其所要求的内存空间超过了内存总容量，作业不能全部被装入内存，致使该作业无法运行。

(2) 有大量作业要求运行，但由于内存容量不足以容纳所有这些作业，只能将少数作业装入内存让它们先运行，而将其它大量的作业留在外存上等待。





4.6.1 虚拟存储器的引入

1. 常规存储器管理方式的特征

(1) 一次性。在前面所介绍的几种存储管理方式中，都要求将作业全部装入内存后方能运行，即作业在运行前需一次性地全部装入内存，而正是这一特征导致了上述两种情况的发生。此外，还有许多作业在每次运行时，并非其全部程序和数据都要用到。如果一次性地装入其全部程序，也是一种对内存空间的浪费。





第四章 存储器管理



(2) 驻留性。作业装入内存后，便一直驻留在内存中，直至作业运行结束。尽管运行中的进程会因I/O而长期等待，或有的程序模块在运行过一次后就不再需要(运行)了，但它们都仍将继续占用宝贵的内存资源。

由此可以看出，上述的一次性及驻留性，使许多在程序运行中不用或暂不用的程序(数据)占据了大量的内存空间，使得一些需要运行的作业无法装入运行。现在要研究的问题是：一次性及驻留性在程序运行时是否是必需的。





2. 局部性原理

早在1968年, Denning. P就曾指出: 程序在执行时将呈现出局部性规律, 即在一较短的时间内, 程序的执行仅局限于某个部分; 相应地, 它所访问的存储空间也局限于某个区域。他提出了下述几个论点:

(1) 程序执行时, 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。该论点也在后来的许多学者对高级程序设计语言(如FORTRAN语言、PASCAL语言)及C语言规律的研究中被证实。





第四章 存储器管理



(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，但经研究看出，过程调用的深度在大多数情况下都不超过5。这就是说，程序将会在一段时间内都局限在这些过程的范围内运行。

(3) 程序中存在许多循环结构，这些虽然只由少数指令构成，但是它们将多次执行。

(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，它们往往都局限于很小的范围内。





第四章 存储器管理



局限性还表现在下述两个方面：

(1) 时间局限性。如果程序中的某条指令一旦执行，则不久以后该指令可能再次执行；如果某数据被访问过，则不久以后该数据可能再次被访问。产生时间局限性的典型原因是由于在程序中存在大量的循环操作。

(2) 空间局限性。一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。





3. 虚拟存储器的定义

基于局部性原理，应用程序在运行之前，没有必要全部装入内存，仅须将那些当前要运行的少数页面或段先装入内存便可运行，其余部分暂留在盘上。程序在运行时，如果它所访问的页(段)已调入内存，便可继续执行下去；但如果程序所要访问的页(段)尚未调入内存(称为缺页或缺段)，此时程序应利用OS所提供的请求调页(段)功能，将它们调入内存，以使进程能继续执行下去。如果此时内存已满，无法再装入新的页(段)，则还须再利用页(段)的置换功能，将内存中暂时不用的页(段)调至盘上，腾出足够的内存空间后，再将要访问的页(段)调入内存，使程序继续执行下去。





4.6.2 虚拟存储器的实现方法

1. 分页请求系统

这是在分页系统的基础上，增加了请求调页功能和页面置换功能所形成的页式虚拟存储系统。它允许只装入少数页面的程序(及数据)，便启动运行。以后，再通过调页功能及页面置换功能，陆续地把即将要运行的页面调入内存，同时把暂不运行的页面换出到外存上。置换时以页面为单位。为了能够实现请求调页和置换功能，系统必须提供必要的硬件支持和相应的软件。





第四章 存储器管理



1) 硬件支持

主要的硬件支持有：

① 请求分页的页表机制，它是在纯分页的页表机制上增加若干项而形成的，作为请求分页的数据结构；

② 缺页中断机构，即每当用户程序要访问的页面尚未调入内存时，便产生一缺页中断，以请求OS将所缺的页调入内存；

③ 地址变换机构，它同样是在纯分页地址变换机构的基础上发展形成的。





第四章 存储器管理



2) 实现请求分页的软件

这里包括有用于实现请求调页的软件和实现页面置换的软件。它们在硬件的支持下，将程序正在运行时所需的页面(尚未在内存中的)调入内存，再将内存中暂时不用的页面从内存置换到磁盘上。





2. 请求分段系统

这是在分段系统的基础上，增加了请求调段及分段置换功能后所形成的段式虚拟存储系统。它允许只装入少数段(而非所有的段)的用户程序和数据，即可启动运行。以后再通过调段功能和段的置换功能将暂不运行的段调出，同时调入即将运行的段。置换是以段为单位进行的。





第四章 存储器管理



为了实现请求分段，系统同样需要必要的硬件支持。一般需要下列支持：

- (1) 请求分段的段表机制。这是在纯分段的段表机制基础上增加若干项而形成的。
- (2) 缺段中断机构。每当用户程序所要访问的段尚未调入内存时，产生一个缺段中断，请求OS将所缺的段调入内存。
- (3) 地址变换机构。





4.6.3 虚拟存储器的特征

1. 多次性

多次性是指一个作业被分成多次调入内存运行，亦即在作业运行时没有必要将其全部装入，只需将当前要运行的那部分程序和数据装入内存即可；以后每当要运行到尚未调入的那部分程序时，再将它调入。多次性是虚拟存储器最重要的特征，任何其它的存储管理方式都不具有这一特征。因此，我们也可以认为虚拟存储器是具有多次性特征的存储器系统。





2. 对换性

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

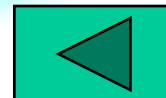




3. 虚拟性

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

值得说明的是，虚拟性是以多次性和对换性为基础的，或者说，仅当系统允许将作业分多次调入内存，并能将内存中暂时不运行的程序和数据换至盘上时，才有可能实现虚拟存储器；而多次性和对换性又必须建立在离散分配的基础上。





4.7 请求分页存储管理方式

4.7.1 请求分页中的硬件支持

1. 页表机制

在请求分页系统所需要的主要数据结构是页表。其基本作用仍然是将用户地址空间中的逻辑地址变换为内存空间中的物理地址。由于只将应用程序的一部分调入内存，还有一部分仍在盘上，故须在页表中再增加若干项，供程序(数据)在换进、换出时参考。在请求分页系统中的每个页表项如下所示：

页号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址





第四章 存储器管理



现对其中各字段说明如下：

(1) 状态位P：用于指示该页是否已调入内存，供程序访问时参考。

(2) 访问字段A：用于记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访问，供选择换出页面时参考。





第四章 存储器管理



(3) 修改位M: 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再将该页写回到外存上, 以减少系统的开销和启动磁盘的次数; 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。简言之, M位供置换页面时参考。

(4) 外存地址: 用于指出该页在外存上的地址, 通常是物理块号, 供调入该页时参考。





2. 缺页中断机构

在请求分页系统中，每当所要访问的页面不在内存时，便产生一缺页中断，请求OS将所缺之页调入内存。缺页中断作为中断，它们同样需要经历诸如保护CPU环境、分析中断原因、转入缺页中断处理程序进行处理、恢复CPU环境等几个步骤。但缺页中断又是一种特殊的中断，它与一般的中断相比，有着明显的区别，主要表现在下面两个方面：

(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，才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所产生的处理的。





第四章 存储器管理



(2) 一条指令在执行期间，可能产生多次缺页中断。在图4-24中示出了一个例子。如在执行一条指令COPY A TO B时，可能要产生6次缺页中断，其中指令本身跨了两个页面，A和B又分别各是一个数据块，也都跨了两个页面。基于这些特征，系统中的硬件机构应能保存多次中断时的状态，并保证最后能返回到中断前产生缺页中断的指令处继续执行。





第四章 存储器管理



页面

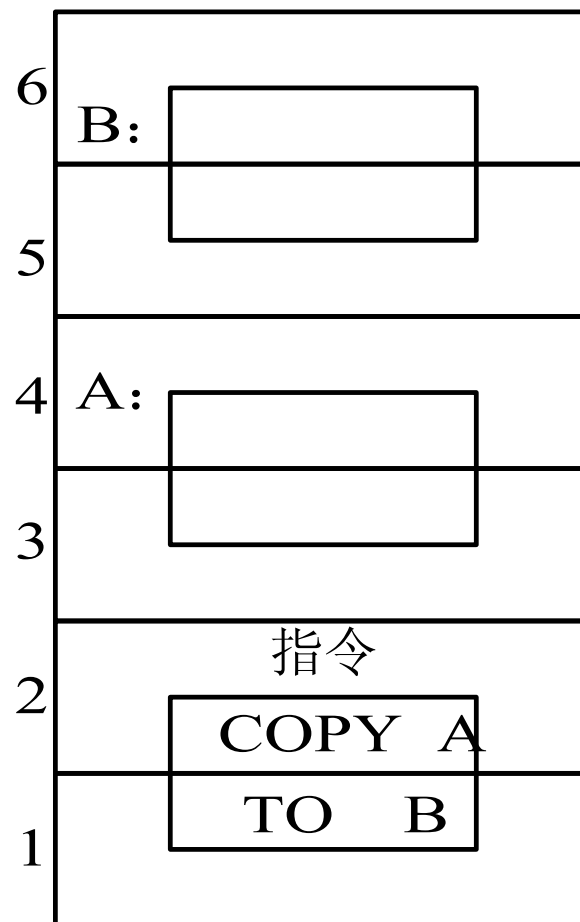


图4-24 涉及6次缺页中断的指令



3. 地址变换机构

请求分页系统中的地址变换机构，是在分页系统地址变换机构的基础上，再为实现虚拟存储器而增加了某些功能而形成的，如产生和处理缺页中断，以及从内存中换出一页的功能等等。图4-25示出了请求分页系统中的地址变换过程。在进行地址变换时，首先去检索快表，试图从中找出所要访问的页。若找到，便修改页表项中的访问位。对于写指令，还须将修改位置成“1”，然后利用页表项中给出的物理块号和页内地址形成物理地址。地址变换过程到此结束。





第四章 存储器管理

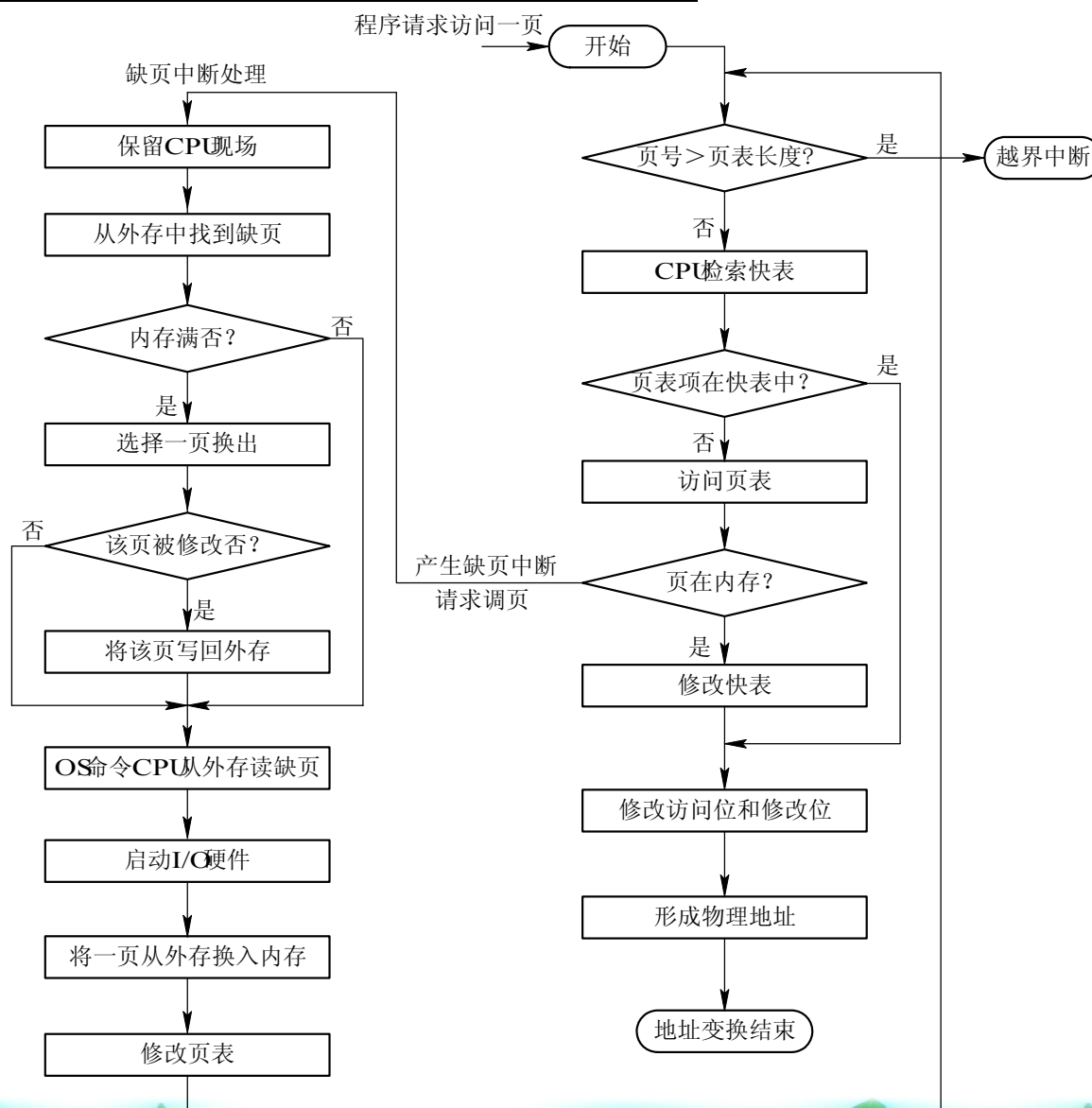


图 4-25 请求分页中的地址变换过程



第四章 存储器管理



如果在快表中未找到该页的页表项时，应到内存中去查找页表，再从找到的页表项中的状态位P，来了解该页是否已调入内存。若该页已调入内存，这时应将此页的页表项写入快表，当快表已满时，应先调出按某种算法所确定的页的页表项，然后再写入该页的页表项；若该页尚未调入内存，这时应产生缺页中断，请求OS从外存把该页调入内存。





4.7.2 内存分配策略和分配算法

1. 最小物理块数的确定

这里所说的最小物理块数，是指能保证进程正常运行所需的最小物理块数。当系统为进程分配的物理块数少于此值时，进程将无法运行。进程应获得的最少物理块数与计算机的硬件结构有关，取决于指令的格式、功能和寻址方式。对于某些简单的机器，若是单地址指令且采用直接寻址方式，则所需的最少物理块数为2。其中，一块是用于存放指令的页面，另一块则是用于存放数据的页面。





第四章 存储器管理



如果该机器允许间接寻址时，则至少要求有三个物理块。对于某些功能较强的机器，其指令长度可能是两个或多于两个字节，因而其指令本身有可能跨两个页面，且源地址和目标地址所涉及的区域也都可能跨两个页面。正如前面所介绍的在缺页中断机构中要发生6次中断的情况一样，对于这种机器，至少要为每个进程分配6个物理块，以装入6个页面。





2. 物理块的分配策略

1) 固定分配局部置换 (Fixed Allocation, Local Replacement)

这是指基于进程的类型(交互型或批处理型等), 或根据程序员、程序管理员的建议, 为每个进程分配一定数目的物理块, 在整个运行期间都不再改变。采用该策略时, 如果进程在运行中发现缺页, 则只能从该进程在内存的 n 个页面中选出一个页换出, 然后再调入一页, 以保证分配给该进程的内存空间不变。实现这种策略的困难在于: 应为每个进程分配多少个物理块难以确定。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, 又必然使内存中驻留的进程数目减少, 进而可能造成CPU空闲或其它资源空闲的情况, 而且在实现进程对换时, 会花费更多的时间。



第四章 存储器管理



2) 可变分配全局置换 (Variable Allocation, Global Replacement)

这可能是最易于实现的一种物理块分配和置换策略，已用于若干个OS中。在采用这种策略时，先为系统中的每个进程分配一定数目的物理块，而OS自身也保持一个空闲物理块队列。当某进程发现缺页时，由系统从空闲物理块队列中取出一个物理块分配给该进程，并将欲调入的(缺)页装入其中。这样，凡产生缺页(中断)的进程，都将获得新的物理块。仅当空闲物理块队列中的物理块用完时，OS才能从内存中选择一页调出，该页可能是系统中任一进程的页，这样，自然又会使那个进程的物理块减少，进而使其缺页率增加。





第四章 存储器管理



3) 可变分配局部置换 (Variable Allocation, Local Replacement)

这同样是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目的物理块，但当某进程发现缺页时，只允许从该进程在内存的页面中选出一页换出，这样就不会影响其它进程的运行。如果进程在运行中频繁地发生缺页中断，则系统须再为该进程分配若干附加的物理块，直至该进程的缺页率减少到适当程度为止；反之，若一个进程在运行过程中的缺页率特别低，则此时可适当减少分配给该进程的物理块数，但不应引起其缺页率的明显增加。





3. 物理块分配算法

1) 平均分配算法

这是将系统中所有可供分配的物理块平均分配给各个进程。例如，当系统中有100个物理块，有5个进程在运行时，每个进程可分得20个物理块。这种方式貌似公平，但实际上是不公平的，因为它未考虑到各进程本身的大小。如有一个进程其大小为200页，只分配给它20个块，这样，它必然会有很高的缺页率；而另一个进程只有10页，却有10个物理块闲置未用。





第四章 存储器管理



2) 按比例分配算法

这是根据进程的大小按比例分配物理块的算法。如果系统中共有 n 个进程，每个进程的页面数为 S_i ，则系统中各进程页面数的总和为：

$$S = \sum_{i=1}^n S_i$$

又假定系统中可用的物理块总数为 m ，则每个进程所能分到的物理块数为 b_i ，将有：

$$b_i = \frac{S_i}{S} \times m$$

b 应该取整，它必须大于最小物理块数。





3) 考虑优先权的分配算法

在实际应用中，为了照顾到重要的、紧迫的作业能尽快地完成，应为它分配较多的内存空间。通常采取的方法是把内存中可供分配的所有物理块分成两部分：一部分按比例地分配给各进程；另一部分则根据各进程的优先权，适当地增加其相应份额后，分配给各进程。在有的系统中，如重要的实时控制系统，则可能是完全按优先权来为各进程分配其物理块的。





4.7.3 调页策略

1. 调入页面的时机

1) 预调页策略

如果进程的许多页是存放在外存的一个连续区域中，则一次调入若干个相邻的页，会比一次调入一页更高效些。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。可采用一种以预测为基础的预调页策略，将那些预计在不久之后便会被访问的页面预先调入内存。如果预测较准确，那么，这种策略显然是很有吸引力的。但遗憾的是，目前预调页的成功率仅约50%。故这种策略主要用于进程的首次调入时，由程序员指出应该先调入哪些页。





2) 请求调页策略

当进程在运行中需要访问某部分程序和数据时，若发现其所在的页面不在内存，便立即提出请求，由OS将其所需页面调入内存。由请求调页策略所确定调入的页，是一定会被访问的，再加之请求调页策略比较易于实现，故在目前的虚拟存储器中大多采用此策略。但这种策略每次仅调入一页，故须花费较大的系统开销，增加了磁盘I/O的启动频率。





2. 确定从何处调入页面

在请求分页系统中的外存分为两部分：用于存放文件的文件区和用于存放对换页面的对换区。通常，由于对换区是采用连续分配方式，而文件区是采用离散分配方式，故对换区的磁盘I/O速度比文件区的高。这样，每当发生缺页请求时，系统应从何处将缺页调入内存，可分成如下三种情况：

(1) 系统拥有足够的对换区空间，这时可以全部从对换区调入所需页面，以提高调页速度。为此，在进程运行前，便须将与该进程有关的文件从文件区拷贝到对换区。





第四章 存储器管理



(2) 系统缺少足够的对换区空间，这时凡是不会被修改的文件都直接从文件区调入；而当换出这些页面时，由于它们未被修改而不必再将它们换出，以后再调入时，仍从文件区直接调入。但对于那些可能被修改的部分，在将它们换出时，便须调到对换区，以后需要时，再从对换区调入。

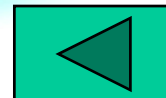
(3) UNIX方式。由于与进程有关的文件都放在文件区，故凡是未运行过的页面，都应从文件区调入。而对于曾经运行过但又被换出的页面，由于是被放在对换区，因此在下次调入时，应从对换区调入。由于UNIX系统允许页面共享，因此，某进程所请求的页面有可能已被其它进程调入内存，此时也就无须再从对换区调入。





3. 页面调入过程

每当程序所要访问的页面未在内存时，便向CPU发出一缺页中断，中断处理程序首先保留CPU环境，分析中断原因后转入缺页中断处理程序。该程序通过查找页表，得到该页在外存的物理块后，如果此时内存能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满，则须先按照某种置换算法从内存中选出一页准备换出；如果该页未被修改过，可不必将该页写回磁盘；但如果此页已被修改，则必须将它写回磁盘，然后再把所缺的页调入内存，并修改页表中的相应表项，置其存在位为“1”，并将此页表项写入快表中。在缺页调入内存后，利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。





4.8 页面置换算法

4.8.1 最佳置换算法和先进先出置换算法

1. 最佳(Optimal)置换算法

最佳置换算法是由Belady于1966年提出的一种理论上的算法。其所选择的被淘汰页面，将是以后永不使用的，或许是在最长(未来)时间内不再被访问的页面。采用最佳置换算法，通常可保证获得最低的缺页率。但由于人们目前还无法预知一个进程在内存的若干个页面中，哪一个页面是未来最长时间不再被访问的，因而该算法是无法实现的，但可以利用该算法去评价其它算法。现举例说明如下。





第四章 存储器管理



假定系统为某进程分配了三个物理块，并考虑有以下的页面号引用串：

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

进程运行时，先将7, 0, 1三个页面装入内存。以后，当进程要访问页面2时，将会产生缺页中断。此时OS根据最佳置换算法，将选择页面7予以淘汰。这是因为页面0将作为第5个被访问的页面，页面1是第14个被访问的页面，而页面7则要在第18次页面访问时才需调入。下次访问页面0时，因它已在内存而不必产生缺页中断。当进程访问页面3时，又将引起页面1被淘汰；因为，它在现有的1, 2, 0三个页面中，将是以后最晚才被访问的。图4-26示出了采用最佳置换算法时的置换图。由图可看出，采用最佳置换算法发生了6次页面置换。



第四章 存储器管理

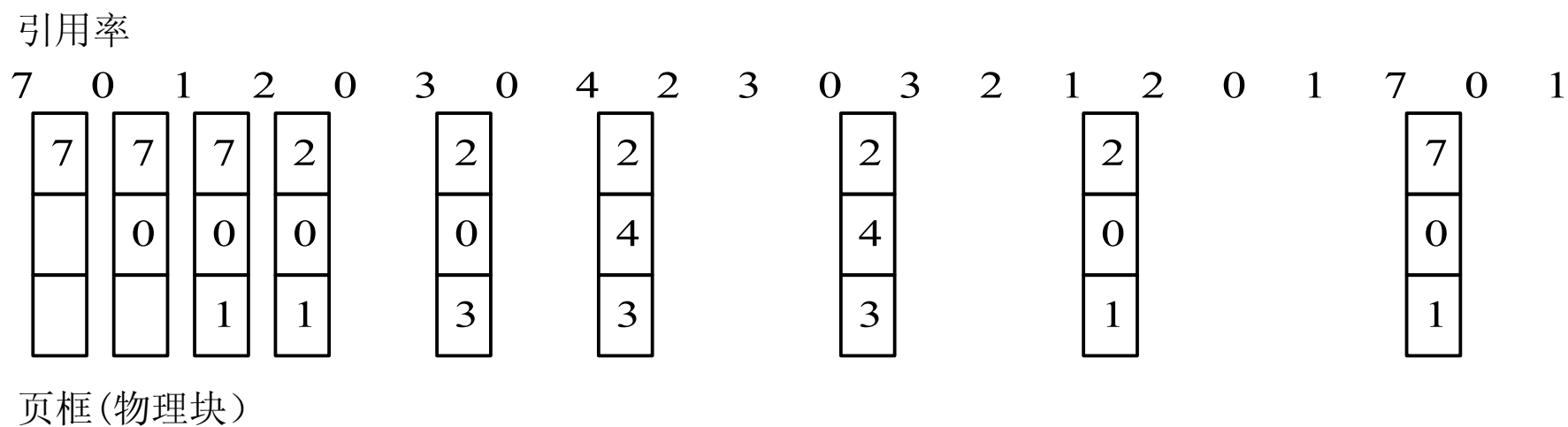


图 4-26 利用最佳页面置换算法时的置换图



2. 先进先出(FIFO)页面置换算法

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，即选择在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，并设置一个指针，称为替换指针，使它总是指向最老的页面。但该算法与进程实际运行的规律不相适应，因为在进程中，有些页面经常被访问，比如，含有全局变量、常用函数、例程等的页面，FIFO算法并不能保证这些页面不被淘汰。





第四章 存储器管理



这里，我们仍用上面的例子，但采用FIFO算法进行页面置换(图4-27)。当进程第一次访问页面2时，将把第7页换出，因为它是最先被调入内存的；在第一次访问页面3时，又将把第0页换出，因为它在现有的2，0，1三个页面中是最老的页。由图4-27 可以看出，利用FIFO算法时进行了12次页面置换，比最佳置换算法正好多一倍。

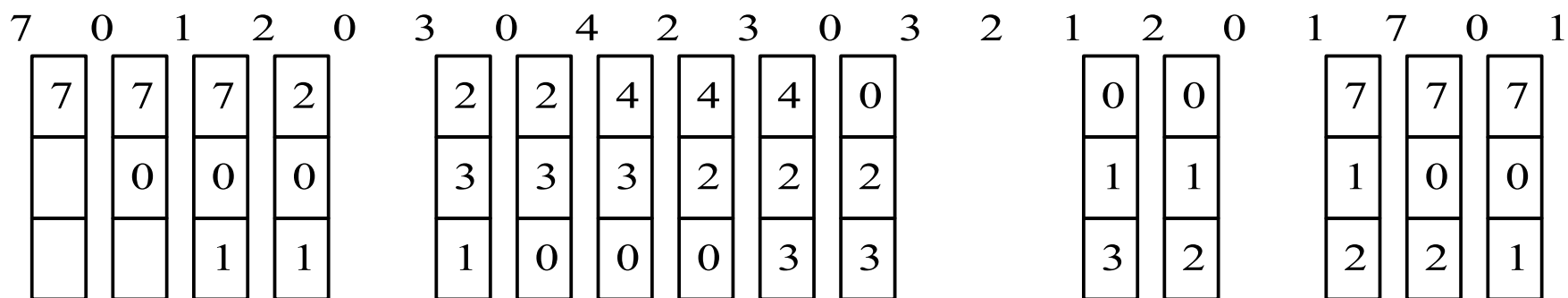




第四章 存储器管理



引用率



页框

图 4-27 利用FIFO置换算法时的置换图



4.8.2 最近最久未使用(LRU)置换算法

1. LRU(Least Recently Used)置换算法的描述

FIFO置换算法性能之所以较差，是因为它所依据的条件是各个页面调入内存的时间，而页面调入的先后并不能反映页面的使用情况。最近最久未使用(LRU)的页面置换算法，是根据页面调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，因此，LRU置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t ，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。





第四章 存储器管理



利用LRU算法对上例进行页面置换的结果如图4-28所示。当进程第一次对页面2进行访问时，由于页面7是最近最久未被访问的，故将它置换出去。当进程第一次对页面3进行访问时，第1页成为最近最久未使用的页，将它换出。由图可以看出，前5个时间的图像与最佳置换算法时的相同，但这并非是必然的结果。因为，最佳置换算法是从“向后看”的观点出发的，即它是依据以后各页的使用情况；而LRU算法则是“向前看”的，即根据各页以前的使用情况来判断，而页面过去和未来的走向之间并无必然的联系。





第四章 存储器管理



引用率

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

页框

图4-28 LRU页面置换算法



2. LRU置换算法的硬件支持

LRU置换算法虽然是一种比较好的算法，但要求系统有较多的支持硬件。为了了解一个进程在内存中的各个页面各有多少时间未被进程访问，以及如何快速地知道哪一页是最近最久未使用的页面，须有两类硬件之一的支持：寄存器或栈。

1) 寄存器

为了记录某进程在内存中各页的使用情况，须为每个在内存中的页面配置一个移位寄存器，可表示为

$$R = R_{n-1}R_{n-2}R_{n-3} \dots R_2R_1R_0$$





第四章 存储器管理



当进程访问某物理块时，要将相应寄存器的 R_{n-1} 位置成1。此时，定时信号将每隔一定时间(例如100 ms)将寄存器右移一位。如果我们把 n 位寄存器的数看做是一个整数，那么，具有最小数值的寄存器所对应的页面，就是最近最久未使用的页面。图4-29示出了某进程在内存中具有8个页面，为每个内存页面配置一个8位寄存器时的LRU访问情况。这里，把8个内存页面的序号分别定为1~8。由图可以看出，第3个内存页面的 R 值最小，当发生缺页时，首先将它置换出去。





第四章 存储器管理



实 页	R							
	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
1	0	1	0	1	0	0	1	0
2	1	0	1	0	1	1	0	0
3	0	0	0	0	0	1	0	0
4	0	1	1	0	1	0	1	1
5	1	1	0	1	0	1	1	0
6	0	0	1	0	1	0	1	1
7	0	0	0	0	0	1	1	1
8	0	1	1	0	1	1	0	1

图4-29 某进程具有8个页面时的LRU访问情况





2) 栈

可利用一个特殊的栈来保存当前使用的各个页面的页面号。每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。因此，栈顶始终是最新被访问页面的编号，而栈底则是最近最久未使用页面的页面号。假定现有一进程所访问的页面的页面号序列为：

4, 7, 0, 7, 1, 0, 1, 2, 1, 2, 6

随着进程的访问，栈中页面号的变化情况如图4-30所示。在访问页面6时发生了缺页，此时页面4是最近最久未被访问的页，应将它置换出去。





第四章 存储器管理

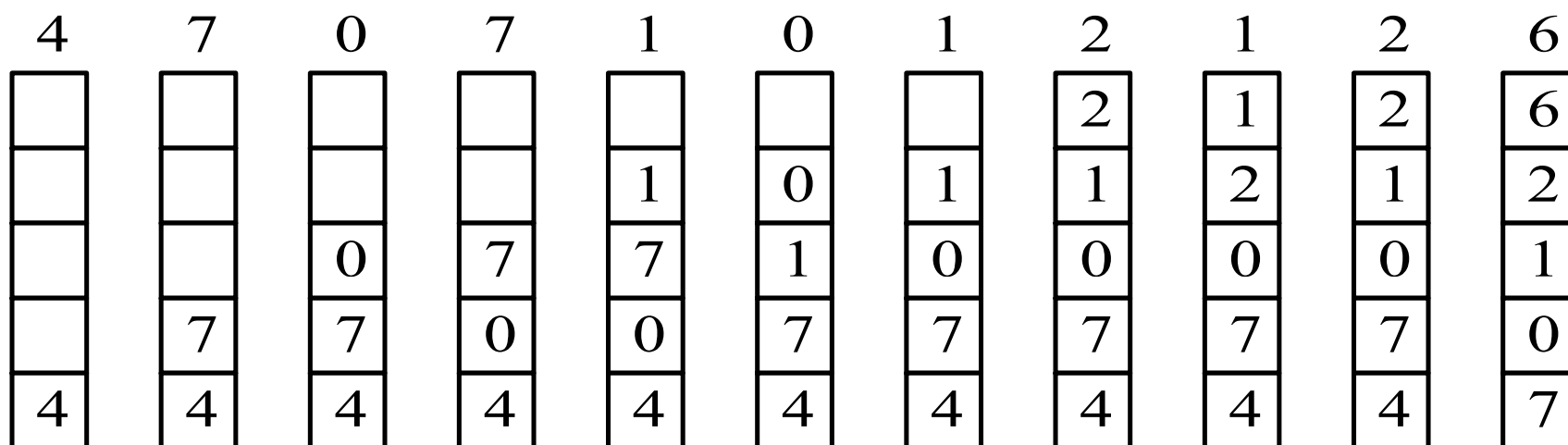


图4-30 用栈保存当前使用页面时栈的变化情况





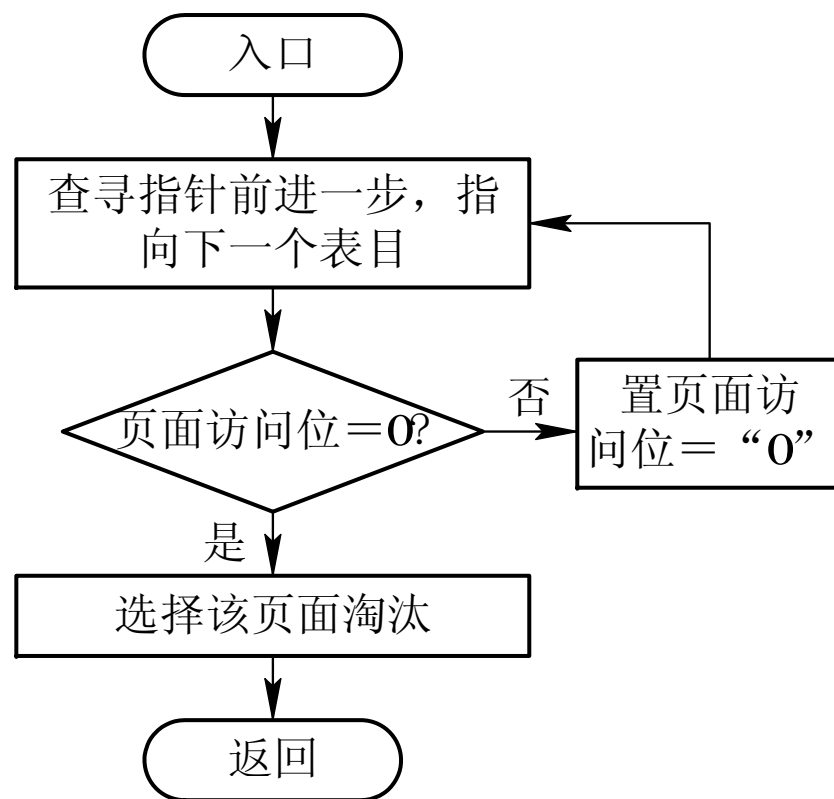
4.8.3 Clock置换算法

1. 简单的Clock置换算法

当采用简单Clock算法时，只需为每页设置一位访问位，再将内存中的所有页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位被置1。置换算法在选择一页淘汰时，只需检查页的访问位。如果是0，就选择该页换出；若为1，则重新将它置0，暂不换出，而给该页第二次驻留内存的机会，再按照FIFO算法检查下一个页面。当检查到队列中的最后一个页面时，若其访问位仍为1，则再返回到队首去检查第一个页面。图4-31示出了该算法的流程和示例。由于该算法是循环地检查各页面的使用情况，故称为Clock算法。但因该算法只有一位访问位，只能用它表示该页是否已经使用过，而置换时是将未使用过的页面换出去，故又把该算法称为最近未用算法NRU(Not Recently Used)。



第四章 存储器管理



块号	页号	访问位	指针
0			
1			
2	4	0	
3			
4	2	1	
5			
6	5	0	
7	1	1	

替换
指针

图4-31 简单Clock置换算法的流程和示例



2. 改进型Clock置换算法

在将一个页面换出时，如果该页已被修改过，便须将该页重新写回到磁盘上；但如果该页未被修改过，则不必将它拷回磁盘。在改进型Clock算法中，除须考虑页面的使用情况外，还须再增加一个因素，即置换代价，这样，选择页面换出时，既要是未使用过的页面，又要是未被修改过的页面。把同时满足这两个条件的页面作为首选淘汰的页面。由访问位A和修改位M可以组合成下面四种类型的页面：





第四章 存储器管理



1类 ($A=0, M=0$): 表示该页最近既未被访问, 又未被修改, 是最佳淘汰页。

2类 ($A=0, M=1$): 表示该页最近未被访问, 但已被修改, 并不是很好的淘汰页。

3类 ($A=1, M=0$): 表示该页最近已被访问, 但未被修改, 该页有可能再被访问。

4类 ($A=1, M=1$): 表示该页最近已被访问且被修改, 该页可能再被访问。





第四章 存储器管理



在内存中的每个页必定是这四类页面之一，在进行页面置换时，可采用与简单Clock算法相类似的算法，其差别在于该算法须同时检查访问位与修改位，以确定该页是四类页面中的哪一种。其执行过程可分成以下三步：

(1) 从指针所指示的当前位置开始，扫描循环队列，寻找 $A=0$ 且 $M=0$ 的第一类页面，将所遇到的第一个页面作为所选中的淘汰页。在第一次扫描期间不改变访问位 A 。





第四章 存储器管理



(2) 如果第一步失败，即查找一周后未遇到第一类页面，则开始第二轮扫描，寻找 $A=0$ 且 $M=1$ 的第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将所有扫描过的页面的访问位都置0。

(3) 如果第二步也失败，亦即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复0。然后重复第一步，如果仍失败，必要时再重复第二步，此时就一定能找到被淘汰的页。

该算法与简单Clock算法比较，可减少磁盘的I/O操作次数。但为了找到一个可置换的页，可能须经过几轮扫描。换言之，实现该算法本身的开销将有所增加。





4.8.4 其它置换算法

1. 最少使用(LFU: Least Frequently Used)置换算法

在采用最少使用置换算法时，应为在内存中的每个页面设置一个移位寄存器，用来记录该页面被访问的频率。该置换算法选择在最近时期使用最少的页面作为淘汰页。由于存储器具有较高的访问速度，例如100 ns，在1 ms时间内可能对某页面连续访问成千上万次，因此，通常不能直接利用计数器来记录某页被访问的次数，而是采用移位寄存器方式。每次访问某页时，便将该移位寄存器的最高位置1，再每隔一定时间(例如100 ms)右移一次。这样，在最近一段时间使用最少的页面将是 $\sum R_i$ 最小的页。





第四章 存储器管理



LFU置换算法的页面访问图与LRU置换算法的访问图完全相同；或者说，利用这样一套硬件既可实现LRU算法，又可实现LFU算法。应该指出，LFU算法并不能真正反映出页面的使用情况，因为在每一时间间隔内，只是用寄存器的一位来记录页的使用情况，因此，访问一次和访问10 000次是等效的。





2. 页面缓冲算法(PBA: Page Buffering Algorithm)

虽然LRU和Clock置换算法都比FIFO算法好，但它们都需要一定的硬件支持，并需付出较多的开销，而且，置换一个已修改的页比置换未修改页的开销要大。而页面缓冲算法(PBA)则既可改善分页系统的性能，又可采用一种较简单的置换策略。VAX/VMS操作系统便是使用页面缓冲算法。它采用了前述的可变分配和局部置换方式，置换算法采用的是FIFO。该算法规定将一个被淘汰的页放入两个链表中的一个，即如果页面未被修改，就将它直接放入空闲链表中；否则，便放入已修改页面的链表中。须注意的是，这时页面在内存中并不做物理上的移动，而只是将页表中的表项移到上述两个链表之一中。

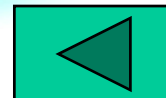




第四章 存储器管理



空闲页面链表，实际上是一个空闲物理块链表，其中的每个物理块都是空闲的，因此，可在其中装入程序或数据。当需要读入一个页面时，便可利用空闲物理块链表中的第一个物理块来装入该页。当有一个未被修改的页要换出时，实际上并不将它换出内存，而是把该未被修改的页所在的物理块挂在自由页链表的末尾。类似地，在置换一个已修改的页面时，也将其所在的物理块挂在修改页面链表的末尾。利用这种方式可使已被修改的页面和未被修改的页面都仍然保留在内存中。当该进程以后再次访问这些页面时，只需花费较小的开销，使这些页面又返回到该进程的驻留集中。当被修改的页面数目达到一定值时，例如64个页面，再将它们一起写回到磁盘上，从而显著地减少了磁盘I/O的操作次数。一个较简单的页面缓冲算法已在MACH操作系统中实现了，只是它没有区分已修改页面和未修改页面。





4.9 请求分段存储管理方式

4.9.1 请求分段中的硬件支持

1. 段表机制

在请求分段式管理所需的主要数据结构是段表。由于在应用程序的许多段中，只有一部分段装入内存，其余的一些段仍留在外存上，故须在段表中增加若干项，以供程序在调进、调出时参考。下面给出请求分段的段表项。

段名	段长	段的基址	存取方式	访问字段 A	修改位 M	存在位 P	增补位	外存始址





第四章 存储器管理



在段表项中，除了段名(号)、段长、段在内存中的起始地址外，还增加了以下诸项。

- (1) 存取方式：用于标识本分段的存取属性是只执行、只读，还是允许读/写。
- (2) 访问字段A：其含义与请求分页的相应字段相同，用于记录该段被访问的频繁程度。
- (3) 修改位M：用于表示该页在进入内存后是否已被修改过，供置换页面时参考。





第四章 存储器管理



(4) 存在位P：指示本段是否已调入内存，供程序访问时参考。

(5) 增补位：这是请求分段式管理中所特有的字段，用于表示本段在运行过程中是否做过动态增长。

(6) 外存始址：指示本段在外存中的起始地址，即起始盘块号。





2. 缺段中断机构

在请求分段系统中，每当发现运行进程所要访问的段尚未调入内存时，便由缺段中断机构产生一缺段中断信号，进入OS后由缺段中断处理程序将所需的段调入内存。缺段中断机构与缺页中断机构类似，它同样需要在一条指令的执行期间，产生和处理中断，以及在一条指令执行期间，可能产生多次缺段中断。但由于分段是信息的逻辑单位，因而不可能出现一条指令被分割在两个分段中和一组信息被分割在两个分段中的情况。缺段中断的处理过程如图4-32所示。由于段不是定长的，这使对缺段中断的处理要比对缺页中断的处理复杂。





第四章 存储器管理

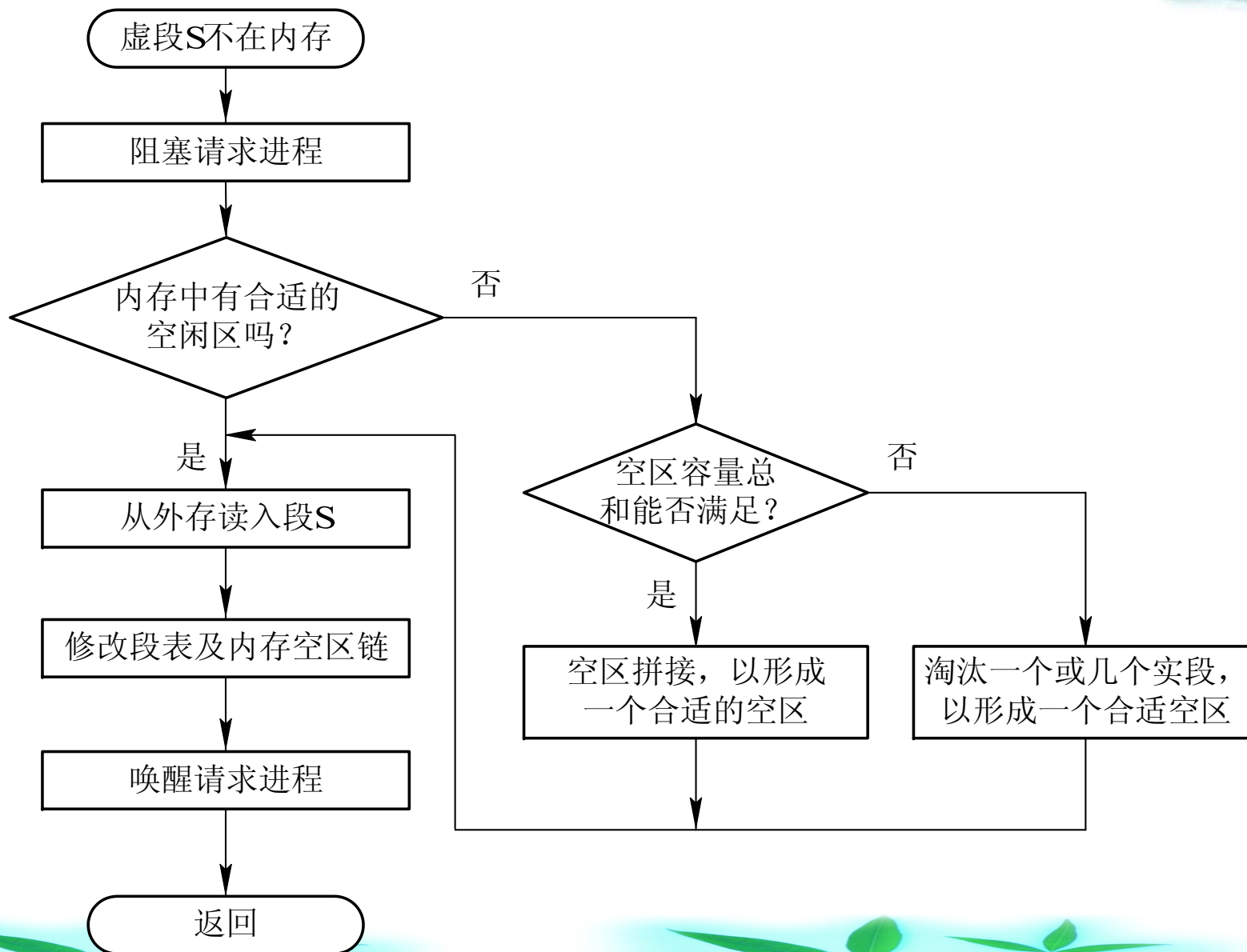


图4-32 请求分段系统中的中断处理过程



3. 地址变换机构

请求分段系统中的地址变换机构是在分段系统地址变换机构的基础上形成的。因为被访问的段并非全在内存，所以在地址变换时，若发现所要访问的段不在内存，必须先将所缺的段调入内存，并修改段表，然后才能再利用段表进行地址变换。为此，在地址变换机构中又增加了某些功能，如缺段中断的请求及处理等。图4-33示出了请求分段系统的地址变换过程。



访问[s][w]

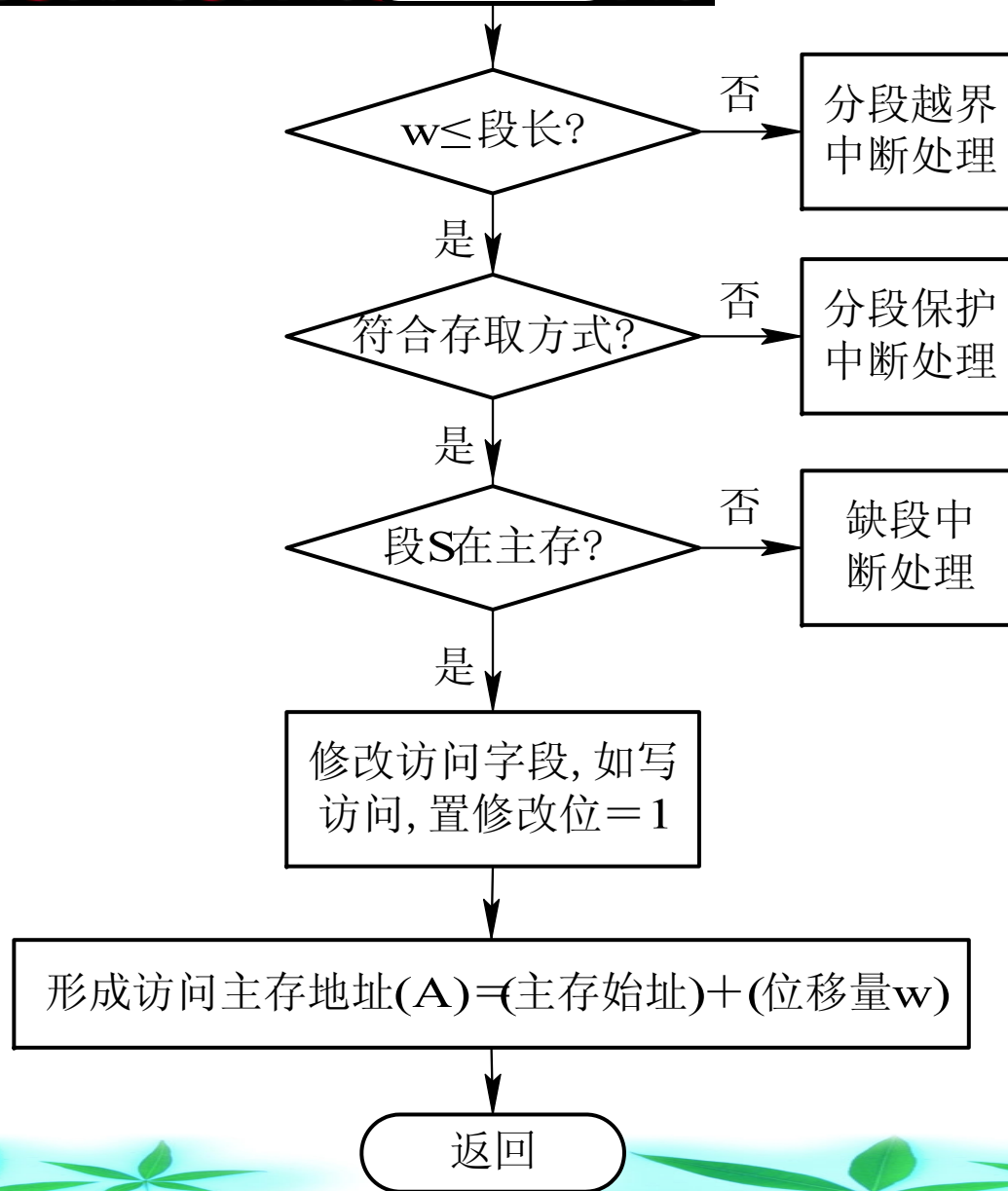


图4-33 请求分段系统的地址变换过程



4.9.2 分段的共享与保护

1. 共享段表

为了实现分段共享，可在系统中配置一张共享段表，所有各共享段都在共享段表中占有一表项。表项中记录了共享段的段号、段长、内存始址、存在位等信息，并记录了共享此分段的每个进程的情况。共享段表如图4-34所示。其中各项说明如下。





第四章 存储器管理



(1) 共享进程计数count。非共享段仅为一个进程所需要。当进程不再需要该段时，可立即释放该段，并由系统回收该段所占用的空间。而共享段是为多个进程所需要的，当某进程不再需要而释放它时，系统并不回收该段所占内存区，仅当所有共享该段的进程全都不再需要它时，才由系统回收该段所占内存区。为了记录有多少个进程需要共享该分段，特设置了一个整型变量count。

(2) 存取控制字段。对于一个共享段，应给不同的进程以不同的存取权限。例如，对于文件主，通常允许他读和写；而对其它进程，则可能只允许读，甚至只允许执行。





第四章 存储器管理



(3) 段号。对于一个共享段，不同的进程可以各用不同的段号去共享该段。

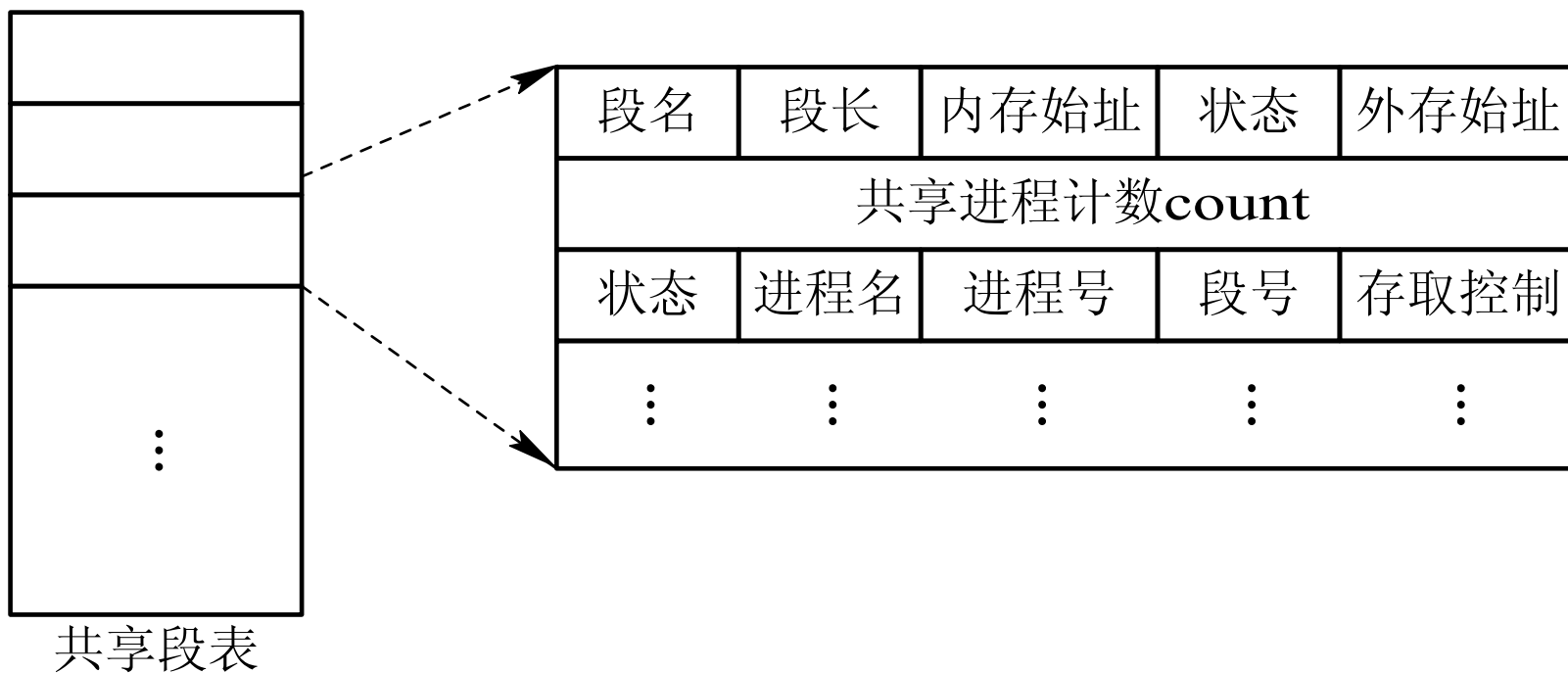


图4-34 共享段表项





2. 共享段的分配与回收

1) 共享段的分配

由于共享段是供多个进程所共享的，因此，对共享段的内存分配方法与非共享段的内存分配方法有所不同。在为共享段分配内存时，对第一个请求使用该共享段的进程，由系统为该共享段分配一物理区，再把共享段调入该区，同时将该区的始址填入请求进程的段表的相应项中，还须在共享段表中增加一表项，填写有关数据，把count置为1；之后，当又有其它进程需要调用该共享段时，由于该共享段已被调入内存，故此时无须再为该段分配内存，而只需在调用进程的段表中增加一表项，填写该共享段的物理地址；在共享段的段表中，填上调用进程的进程名、存取控制等，再执行 $\text{count} := \text{count} + 1$ 操作，以表明有两个进程共享该段。



2) 共享段的回收

当共享此段的某进程不再需要该段时，应将该段释放，包括撤消在该进程段表中共享段所对应的表项，以及执行 $\text{count} := \text{count} - 1$ 操作。若结果为0，则须由系统回收该共享段的物理内存，以及取消在共享段表中该段所对应的表项，表明此时已没有进程使用该段；否则(减1结果不为0)，只是取消调用者进程在共享段表中的有关记录。





3. 分段保护

在分段系统中，由于每个分段在逻辑上是独立的，因而比较容易实现信息保护。目前，常采用以下几种措施来确保信息的安全。

1) 越界检查

在段表寄存器中放有段表长度信息；同样，在段表中也为每个段设置有段长字段。在进行存储访问时，首先将逻辑地址空间的段号与段表长度进行比较，如果段号等于或大于段表长度，将发出地址越界中断信号；其次，还要检查段内地址是否等于或大于段长，若大于段长，将产生地址越界中断信号，从而保证了每个进程只能在自己的地址空间内运行。





2) 存取控制检查

在段表的每个表项中，都设置了一个“存取控制”字段，用于规定对该段的访问方式。通常的访问方式有：

(1) 只读，即只允许进程对该段中的程序或数据进行读访问。

(2) 只执行，即只允许进程调用该段去执行，但不准读该段的内容，也不允许对该段执行写操作。

(3) 读/写，即允许进程对该段进行读/写访问。





第四章 存储器管理



3) 环保护机构

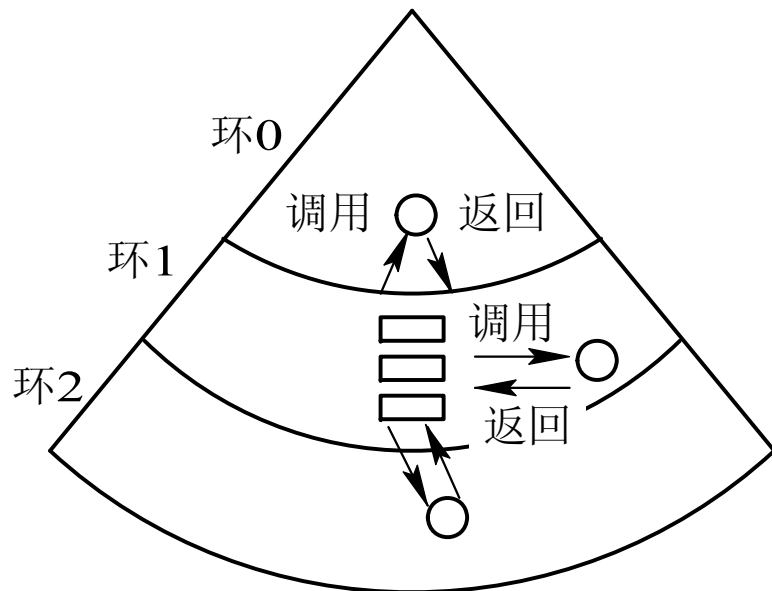
这是一种功能较完善的保护机制。在该机制中规定：低编号的环具有高优先权。OS核心处于0环内；某些重要的实用程序和操作系统服务占居中间环；而一般的应用程序则被安排在外环上。在环系统中，程序的访问和调用应遵循以下规则：

- (1) 一个程序可以访问驻留在相同环或较低特权环中的数据。
- (2) 一个程序可以调用驻留在相同环或较高特权环中的服务。

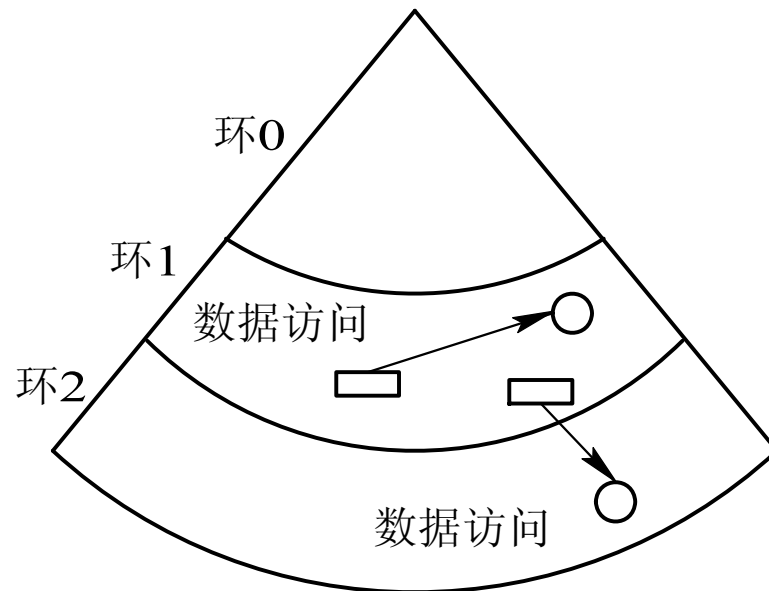




第四章 存储器管理



(a) 程序间的控制传输



(b) 数据访问

图4-35 环保护机构





第五章 设备管理

[5.1 I/O系统](#)

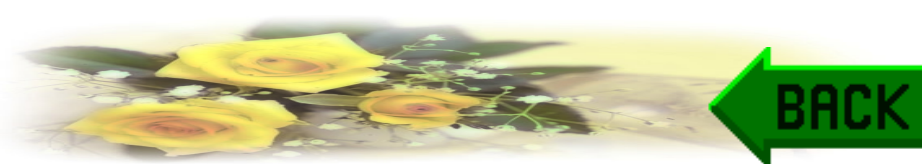
[5.2 I/O控制方式](#)

[5.3 缓冲管理](#)

[5.4 I/O软件](#)

[5.5 设备分配](#)

[5.6 磁盘存储器的管理](#)



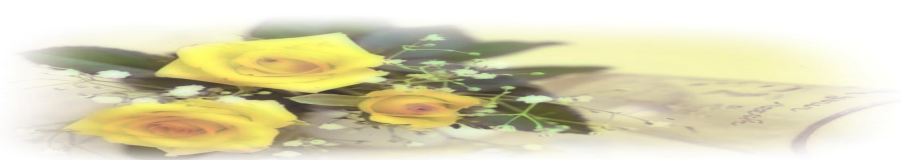


5.1 I/O系统

5.1.1 I/O设备

1. I/O设备的类型

I/O设备的类型繁多，从OS观点看，其重要的性能指标有：设备使用特性、数据传输速率、数据的传输单位、设备共享属性等。因而可从不同角度对它们进行分类。

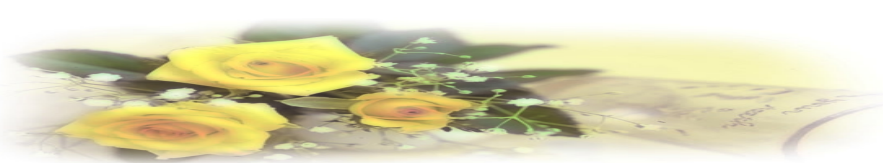




第五章 设备管理

1) 按设备的使用特性分类

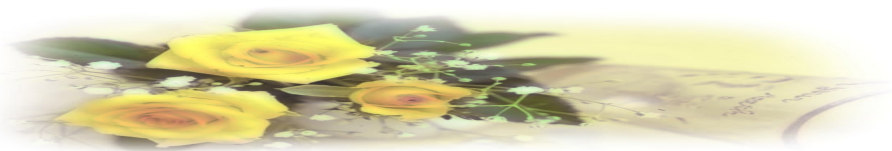
按设备的使用特性，可将设备分为两类。第一类是存储设备，也称外存或后备存储器、辅助存储器，是计算机系统用以存储信息的主要设备。该类设备存取速度较内存慢，但容量比内存大得多，相对价格也便宜。第二类就是输入/输出设备，又具体可分为输入设备、输出设备和交互式设备。输入设备用来接收外部信息，如键盘、鼠标、扫描仪、视频摄像、各类传感器等。输出设备是用于将计算机加工处理后的信息送向外部的设备，如打印机、绘图仪、显示器、数字视频显示设备、音响输出设备等。交互式设备则是集成上述两类设备，利用输入设备接收用户命令信息，并通过输出设备(主要是显示器)同步显示用户命令以及命令执行的结果。





2) 按传输速率分类

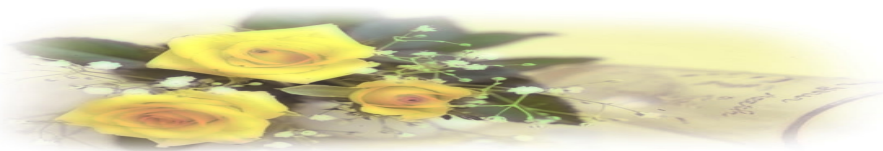
按传输速度的高低，可将I/O设备分为三类。第一类是低速设备，这是指其传输速率仅为每秒钟几个字节至数百个字节的一类设备。属于低速设备的典型设备有键盘、鼠标器、语音的输入和输出等设备。第二类是中速设备，这是指其传输速率在每秒钟数千个字节至数十万个字节的一类设备。典型的中速设备有行式打印机、激光打印机等。第三类是高速设备，这是指其传输速率在数百个千字节至千兆字节的一类设备。典型的高速设备有磁带机、磁盘机、光盘机等。





3) 按信息交换的单位分类

按信息交换的单位, 可将I/O设备分成两类。第一类是块设备(Block Device), 这类设备用于存储信息。由于信息的存取总是以数据块为单位, 故而得名。它属于有结构设备。典型的块设备是磁盘, 每个盘块的大小为512 B~4 KB。磁盘设备的基本特征是其传输速率较高, 通常每秒钟为几兆位; 另一特征是可寻址, 即对它可随机地读/写任一块; 此外, 磁盘设备的I/O常采用DMA方式。第二类是字符设备(Character Device), 用于数据的输入和输出。其基本单位是字符, 故称为字符设备。它属于无结构类型。字符设备的种类繁多, 如交互式终端、打印机等。字符设备的基本特征是其传输速率较低, 通常为几个字节至数千字节; 另一特征是不可寻址, 即输入/输出时不能指定数据的输入源地址及输出的目标地址; 此外, 字符设备在输入/输出时, 常采用中断驱动方式。



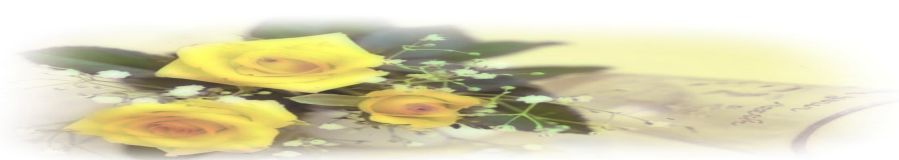


第五章 设备管理

4) 按设备的共享属性分类

这种分类方式可将I/O设备分为如下三类：

(1) 独占设备。这是指在一段时间内只允许一个用户(进程)访问的设备，即临界资源。因而，对多个并发进程而言，应互斥地访问这类设备。系统一旦把这类设备分配给了某进程后，便由该进程独占，直至用完释放。应当注意，独占设备的分配有可能引起进程死锁。

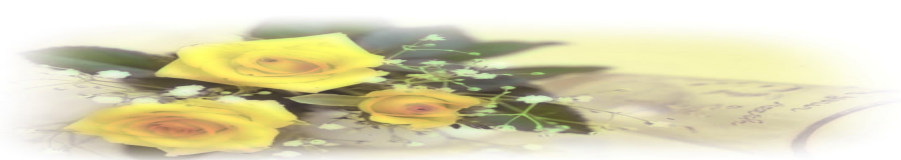




第五章 设备管理

(2) 共享设备。这是指在一段时间内允许多个进程同时访问的设备。当然，对于每一时刻而言，该类设备仍然只允许一个进程访问。显然，共享设备必须是可寻址的和可随机访问的设备。典型的共享设备是磁盘。对共享设备不仅可获得良好的设备利用率，而且它也是实现文件系统和数据库系统的物质基础。

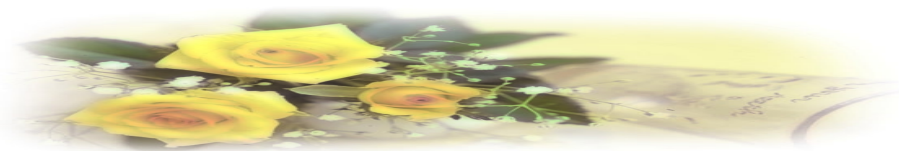
(3) 虚拟设备。这是指通过虚拟技术将一台独占设备变换为若干台逻辑设备，供若干个用户(进程)同时使用。





2. 设备与控制器之间的接口

通常，设备并不是直接与CPU进行通信，而是与设备控制器通信，因此，在I/O设备中应含有与设备控制器间的接口，在该接口中有三种类型的信号(见图5-1 所示)，各对应一条信号线。



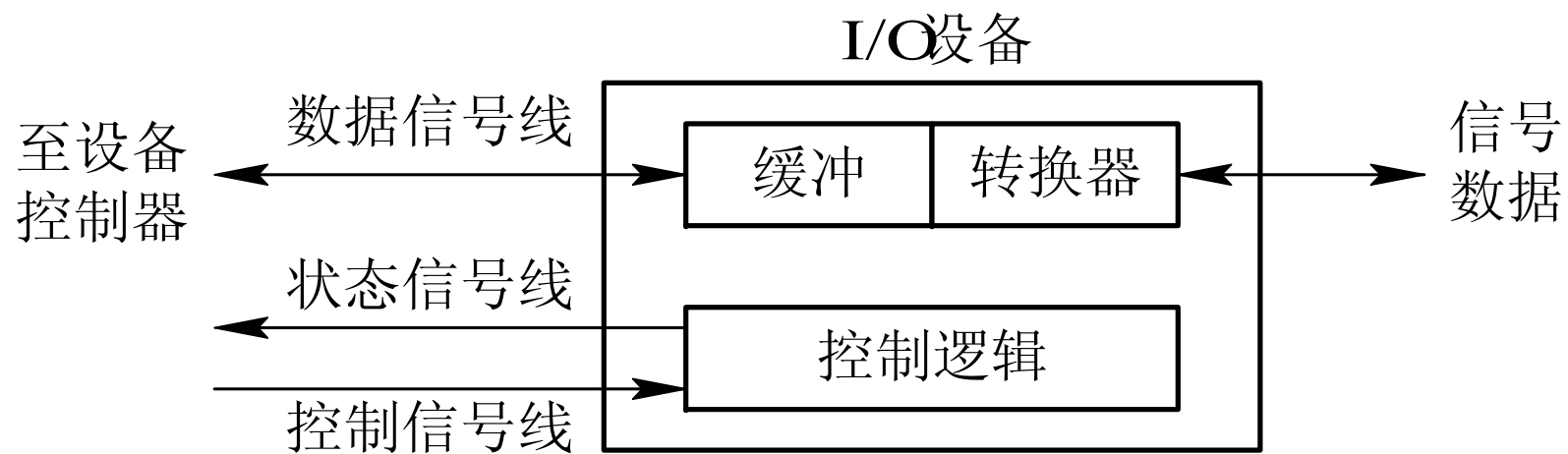
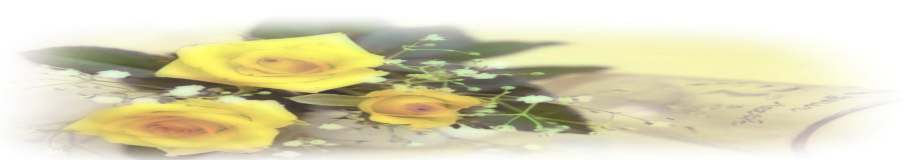


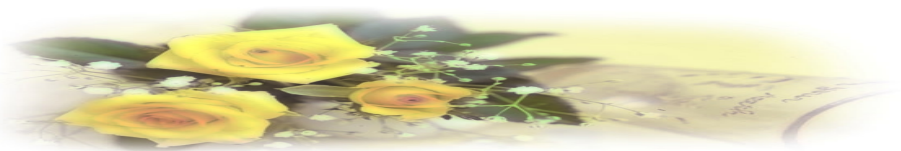
图5-1 设备与控制器间的接口





1) 数据信号线

这类信号线用于在设备和设备控制器之间传送数据信号。对输入设备而言，由外界输入的信号经转换器转换后所形成的数据，通常先送入缓冲器中，当数据量达到一定的比特(字符)数后，再从缓冲器通过一组数据信号线传送给设备控制器，如图5-1所示。对输出设备而言，则是将从设备控制器经过数据信号线传送来的一批数据先暂存于缓冲器中，经转换器作适当转换后，再逐个字符地输出。



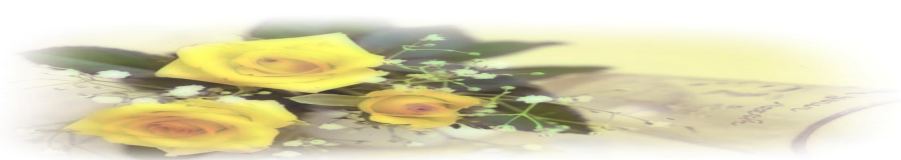


2) 控制信号线

这是作为由设备控制器向I/O设备发送控制信号时的通路。该信号规定了设备将要执行的操作，如读操作(指由设备向控制器传送数据)或写操作(从控制器接收数据)，或执行磁头移动等操作。

3) 状态信号线

这类信号线用于传送指示设备当前状态的信号。设备的当前状态有正在读(或写)；设备已读(写)完成，并准备好新的数据传送。



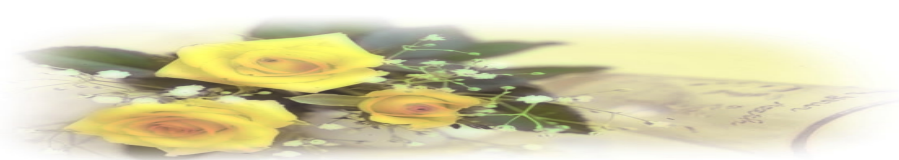


5.1.2 设备控制器

1. 设备控制器的基本功能

1) 接收和识别命令

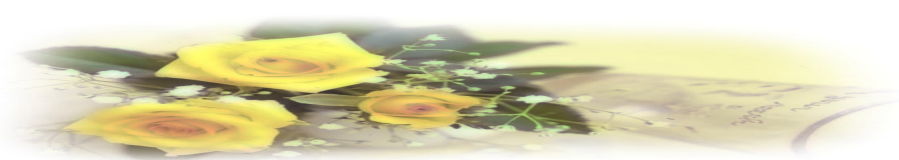
CPU可以向控制器发送多种不同的命令，设备控制器应能接收并识别这些命令。为此，在控制器中应具有相应的控制寄存器，用来存放接收的命令和参数，并对所接收的命令进行译码。例如，磁盘控制器可以接收CPU发来的Read、Write、Format等15条不同的命令，而且有些命令还带有参数；相应地，在磁盘控制器中有多个寄存器和命令译码器等。





2) 数据交换

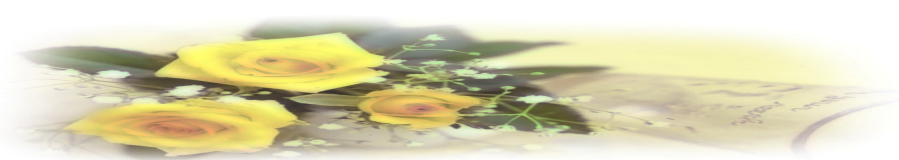
这是指实现CPU与控制器之间、控制器与设备之间的数据交换。对于前者，是通过数据总线，由CPU并行地把数据写入控制器，或从控制器中并行地读出数据；对于后者，是设备将数据输入到控制器，或从控制器传送给设备。为此，在控制器中须设置数据寄存器。





3) 标识和报告设备的状态

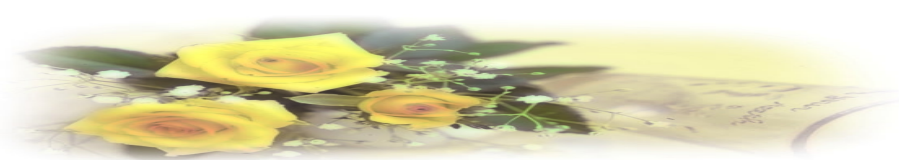
控制器应记下设备的状态供CPU了解。例如，仅当该设备处于发送就绪状态时，CPU才能启动控制器从设备中读出数据。为此，在控制器中应设置一状态寄存器，用其中的每一位来反映设备的某一种状态。当CPU将该寄存器的内容读入后，便可了解该设备的状态。





4) 地址识别

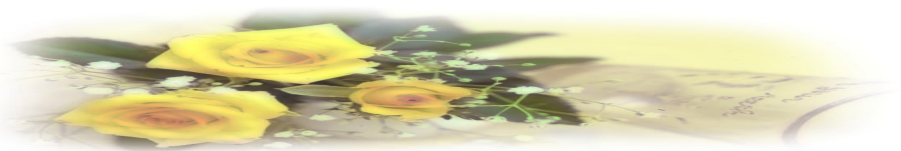
就像内存中的每一个单元都有一个地址一样，系统中的每一个设备也都有一个地址，而设备控制器又必须能够识别它所控制的每个设备的地址。此外，为使CPU能向(或从)寄存器中写入(或读出)数据，这些寄存器都应具有唯一的地址。例如，在IB-MPC机中规定，硬盘控制器中各寄存器的地址分别为320~32F之一。控制器应能正确识别这些地址，为此，在控制器中应配置地址译码器。





5) 数据缓冲

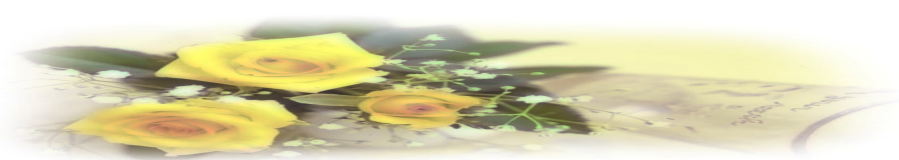
由于I/O设备的速率较低而CPU和内存的速率却很高，故在控制器中必须设置一缓冲器。在输出时，用此缓冲器暂存由主机高速传来的数据，然后才以I/O设备所具有的速率将缓冲器中的数据传送给I/O设备；在输入时，缓冲器则用于暂存从I/O设备送来的数据，待接收到一批数据后，再将缓冲器中的数据高速地传送给主机。





6) 差错控制

设备控制器还兼管对由I/O设备传送来的数据进行差错检测。若发现传送中出现了错误，通常是将差错检测码置位，并向CPU报告，于是CPU将本次传送来的数据作废，并重新进行一次传送。这样便可保证数据输入的正确性。

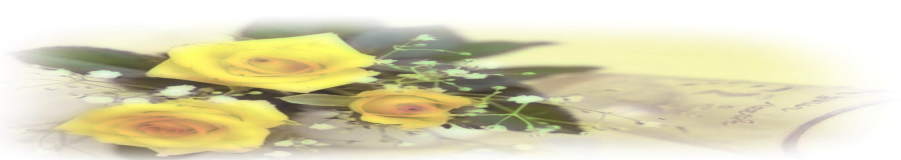




2. 设备控制器的组成

1) 设备控制器与处理机的接口

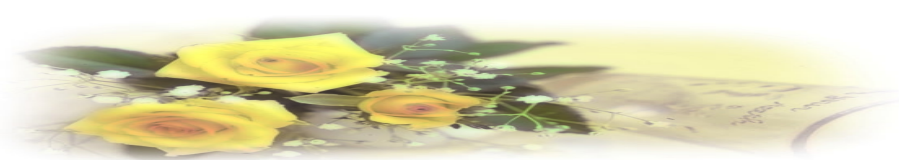
该接口用于实现CPU与设备控制器之间的通信。共有三类信号线: 数据线、地址线和控制线。数据线通常与两类寄存器相连接, 第一类是数据寄存器(在控制器中可以有一个或多个数据寄存器, 用于存放从设备送来的数据(输入)或从CPU送来的数据(输出)); 第二类是控制/状态寄存器(在控制器中可以有一个或多个这类寄存器, 用于存放从CPU送来的控制信息或设备的状态信息)。





2) 设备控制器与设备的接口

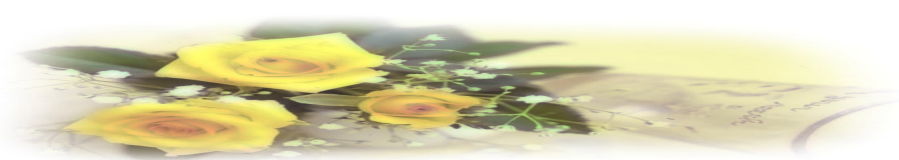
在一个设备控制器上，可以连接一个或多个设备。相应地，在控制器中便有一个或多个设备接口，一个接口连接一台设备。在每个接口中都存在数据、控制和状态三种类型的信号。控制器中的I/O逻辑根据处理机发来的地址信号去选择一个设备接口。





3) I/O逻辑

在设备控制器中的I/O逻辑用于实现对设备的控制。它通过一组控制线与处理机交互，处理机利用该逻辑向控制器发送I/O命令；I/O逻辑对收到的命令进行译码。每当CPU要启动一个设备时，一方面将启动命令发送给控制器；另一方面又同时通过地址线把地址发送给控制器，由控制器的I/O逻辑对收到的地址进行译码，再根据所译出的命令对所选设备进行控制。设备控制器的组成示于图5-2中。



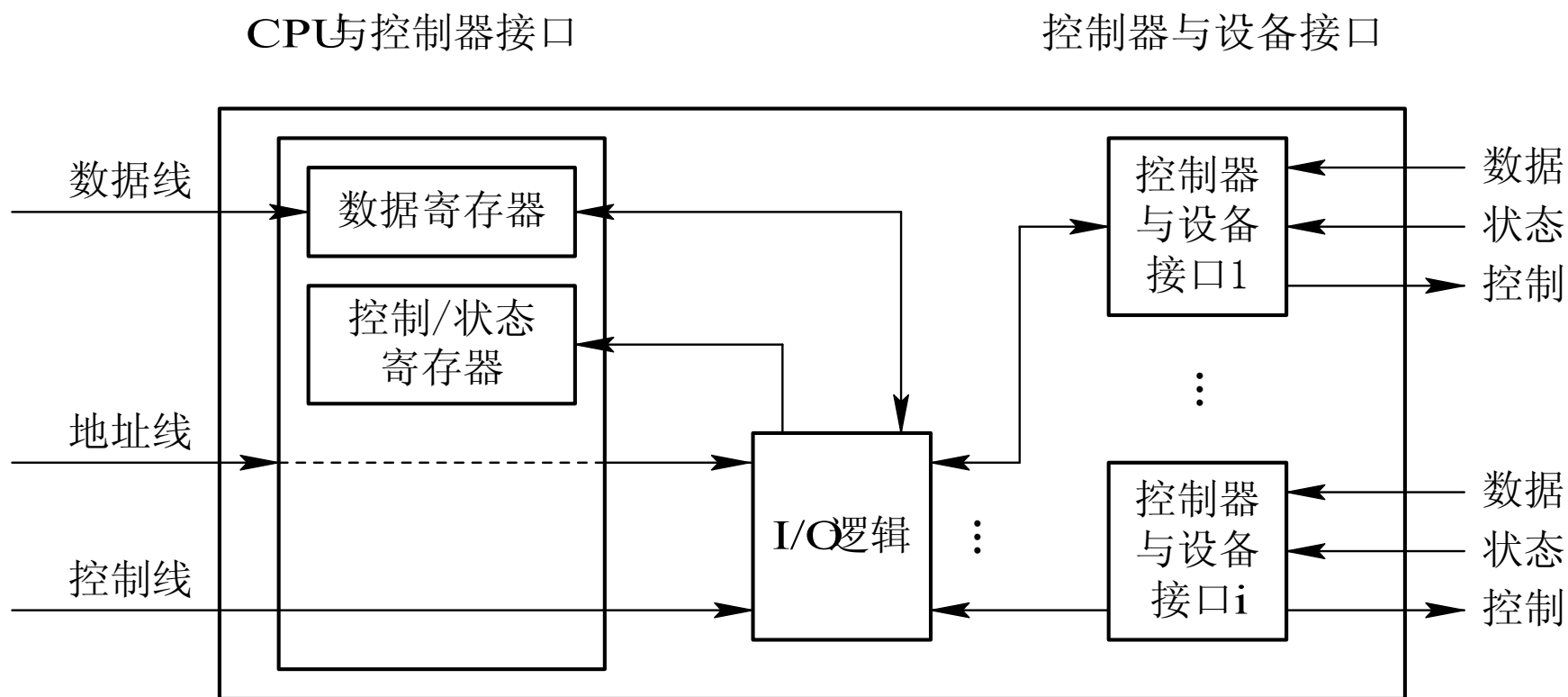
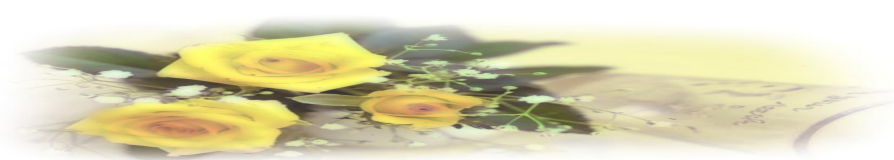


图5-2 设备控制器的组成

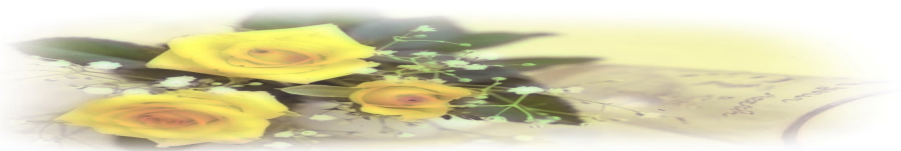




5.1.3 I/O通道

1. I/O通道(I/O Channel)设备的引入

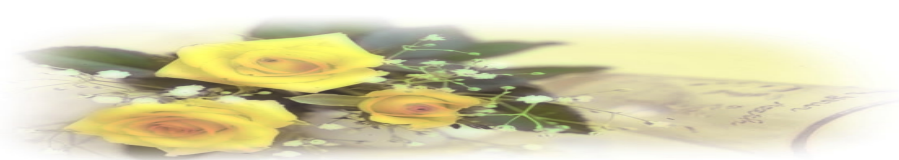
虽然在CPU与I/O设备之间增加了设备控制器后，已能大大减少CPU对I/O的干预，但当主机所配置的外设很多时，CPU的负担仍然很重。为此，在CPU和设备控制器之间又增设了通道。其主要目的是为了建立独立的I/O操作，不仅使数据的传送能独立于CPU，而且也希望有关对I/O操作的组织、管理及其结束处理尽量独立，以保证CPU有更多的时间去进行数据处理；或者说，其目的是使一些原来由CPU处理的I/O任务转由通道来承担，从而把CPU从繁杂的I/O任务中解脱出来。在设置了通道后，CPU只需向通道发送一条I/O指令。通道在收到该指令后，便从内存中取出本次要执行的通道程序，然后执行该通道程序，仅当通道完成了规定的I/O任务后，才向CPU发中断信号。





第五章 设备管理

实际上，I/O通道是一种特殊的处理机，它具有执行I/O指令的能力，并通过执行通道(I/O)程序来控制I/O操作。但I/O通道又与一般的处理机不同，主要表现在以下两个方面：一是其指令类型单一，这是由于通道硬件比较简单，其所能执行的命令主要局限于与I/O操作有关的指令；二是通道没有自己的内存，通道所执行的通道程序是放在主机的内存中的，换言之，是通道与CPU共享内存。

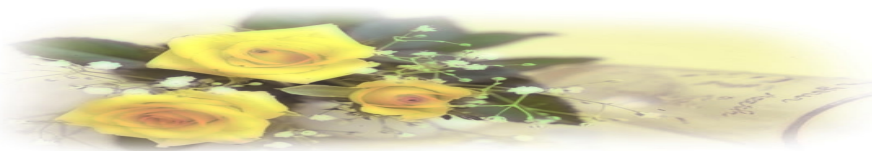




2. 通道类型

1) 字节多路通道(Byte Multiplexor Channel)

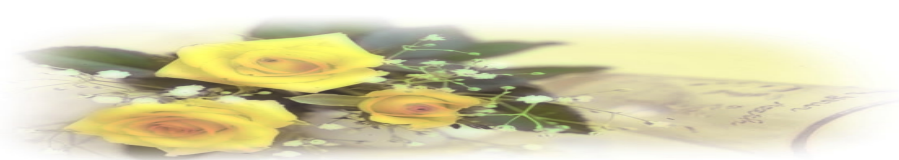
这是一种按字节交叉方式工作的通道。它通常都含有许多非分配型子通道，其数量可从几十到数百个，每一个子通道连接一台I/O设备，并控制该设备的I/O操作。这些子通道按时间片轮转方式共享主通道。当第一个子通道控制其I/O设备完成一个字节的交换后，便立即腾出主通道，让给第二个子通道使用；当第二个子通道也完成一个字节的交换后，同样也把主通道让给第三个子通道；依此类推。当所有子通道轮转一周后，重又返回来由第一个子通道去使用字节多路主通道。这样，只要字节多路通道扫描每个子通道的速率足够快，而连接到子通道上的设备的速率不是太高时，便不致丢失信息。





第五章 设备管理

图5-3示出了字节多路通道的工作原理。它所含有的多个子通道A, B, C, D, E, ..., N, ...分别通过控制器各与一台设备相连。假定这些设备的速率相近, 且都同时向主机传送数据。设备A所传送的数据流为A1A2A3...; 设备B所传送的数据流为B1B2B3...把这些数据流合成后(通过主通道)送往主机的数据流为A1B1C1D1 ...A2B2C2D2 ... A3B3C3D3 ...。



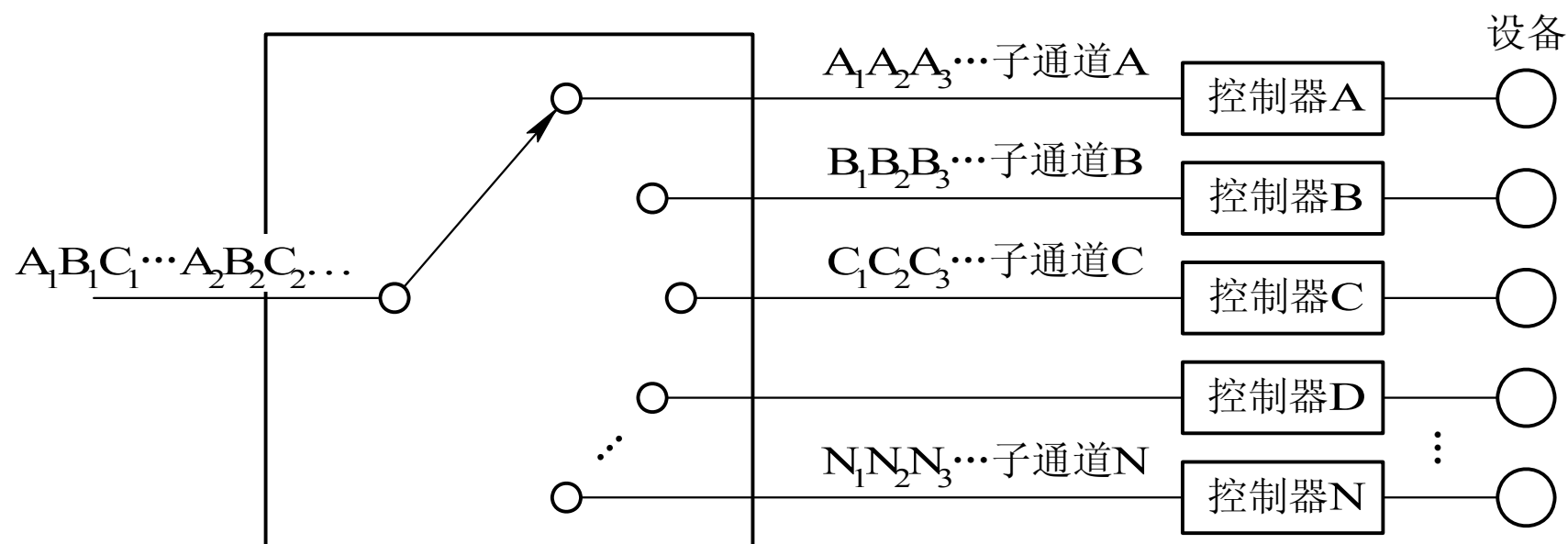
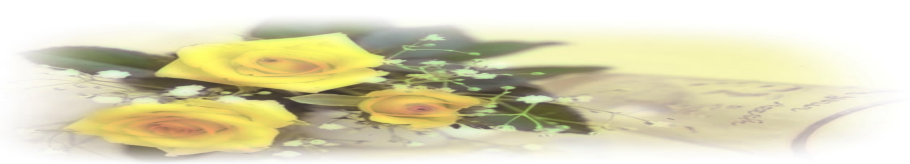


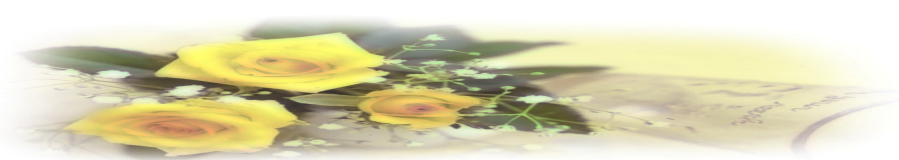
图5-3 字节多路通道的工作原理





2) 数组选择通道(Block Selector Channel)

字节多路通道不适于连接高速设备，这推动了按数组方式进行数据传送的数组选择通道的形成。这种通道虽然可以连接多台高速设备，但由于它只含有一个分配型子通道，在一段时间内只能执行一道通道程序，控制一台设备进行数据传送，致使当某台设备占用了该通道后，便一直由它独占，即使是它无数据传送，通道被闲置，也不允许其它设备使用该通道，直至该设备传送完毕释放该通道。可见，这种通道的利用率很低。

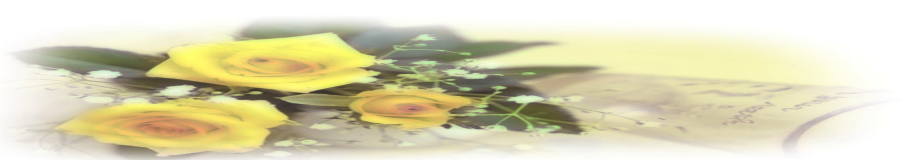




第五章 设备管理

3) 数组多路通道(Block Multiplexor Channel)

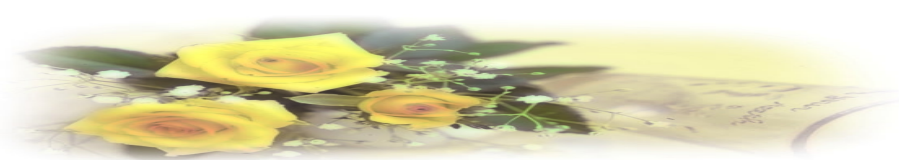
数组选择通道虽有很高的传输速率，但它却每次只允许一个设备传输数据。数组多路通道是将数组选择通道传输速率高和字节多路通道能使各子通道(设备)分时并行操作的优点相结合而形成的一种新通道。它含有多个非分配型子通道，因而这种通道既具有很高的数据传输速率，又能获得令人满意的通道利用率。也正因此，才使该通道能被广泛地用于连接多台高、中速的外围设备，其数据传送是按数组方式进行的。





3. “瓶颈”问题

由于通道价格昂贵，致使机器中所设置的通道数量势必较少，这往往又使它成了I/O的瓶颈，进而造成整个系统吞吐量的下降。例如，在图5-4中，假设设备1至设备4是四个磁盘，为了启动磁盘4，必须用通道1和控制器2；但若这两者已被其它设备占用，必然无法启动磁盘4。类似地，若要启动盘1和盘2，由于它们都要用到通道1，因而也不可能启动。这些就是由于通道不足所造成的“瓶颈”现象。



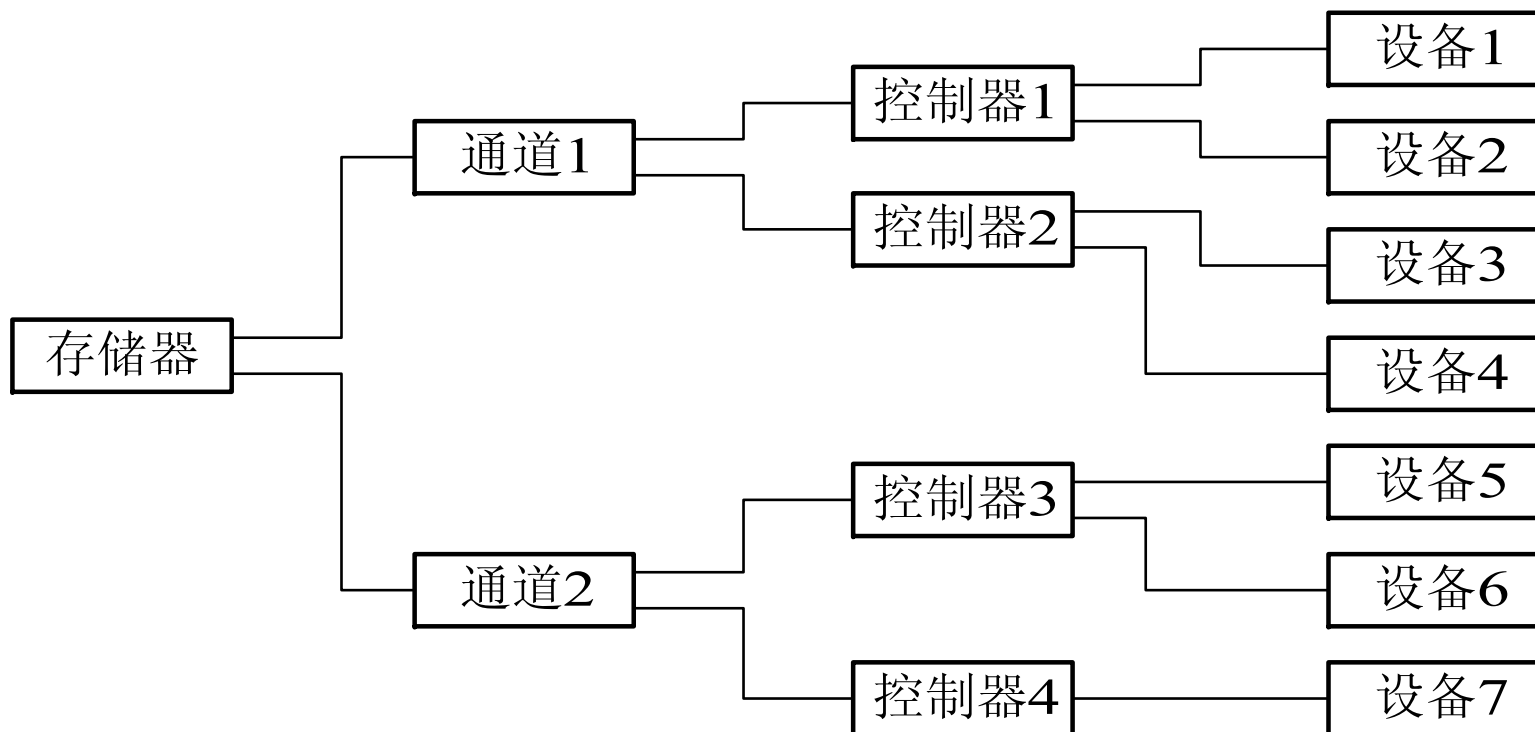
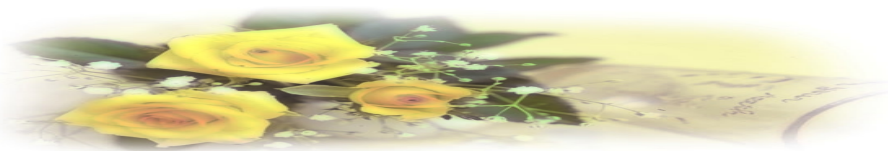


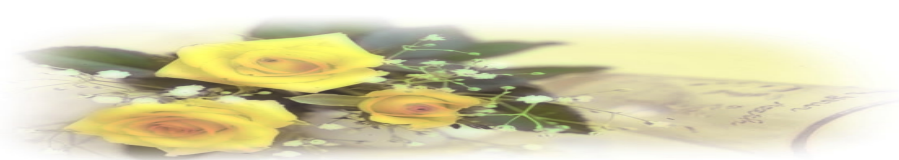
图5-4 单通路I/O系统





第五章 设备管理

解决“瓶颈”问题的最有效的方法，便是增加设备到主机间的通路而不增加通道，如图5-5所示。换言之，就是把一个设备连接到多个控制器上，而一个控制器又连接到多个通道上。图中的设备1、2、3和4，都有四条通往存储器的通路。例如，通过控制器1和通道1到存储器；也可通过控制器2和通道1到存储器。多通路方式不仅解决了“瓶颈”问题，而且提高了系统的可靠性，因为个别通道或控制器的故障不会使设备和存储器之间没有通路。



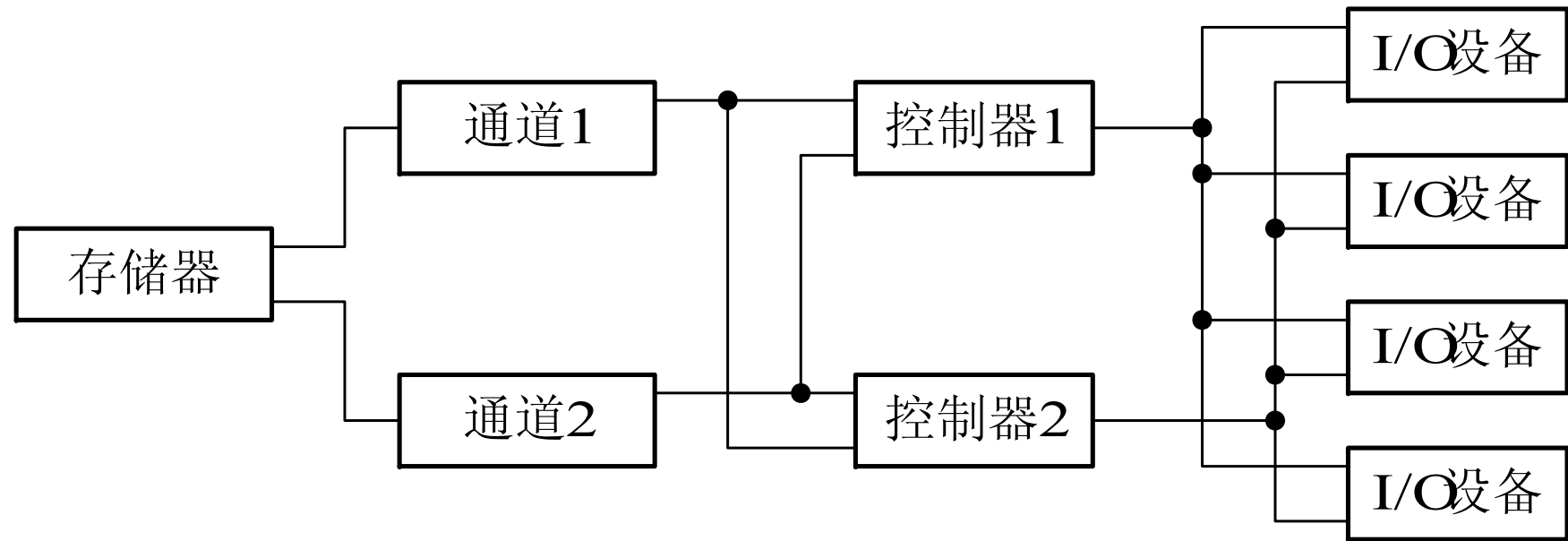
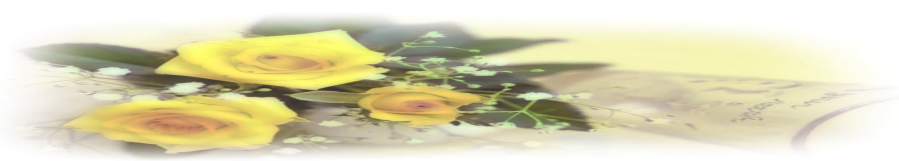


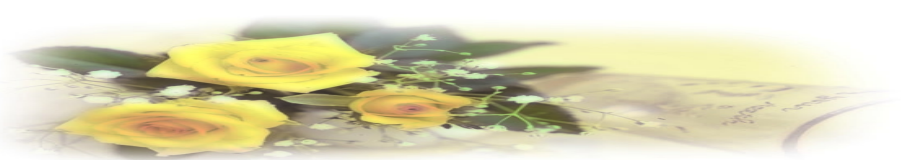
图5-5 多通路I/O系统





5.1.4 总线系统

由图5-6 可以看出，在计算机系统各部件，如CPU、存储器以及各种I/O设备之间的联系，都是通过总线来实现的。总线的性能是用总线的时钟频率、带宽和相应的总线传输速率等指标来衡量的。随着计算机中CPU和内存速率的提高，字长的增加，以及不断地引入新型设备，促使人们对总线的时钟频率、带宽和传输速率的要求也不断提高。这便推动了总线的不断发展，使之由早期的ISA总线发展为EISA总线、VESA总线，进而又演变成当前广为流行的PCI总线。



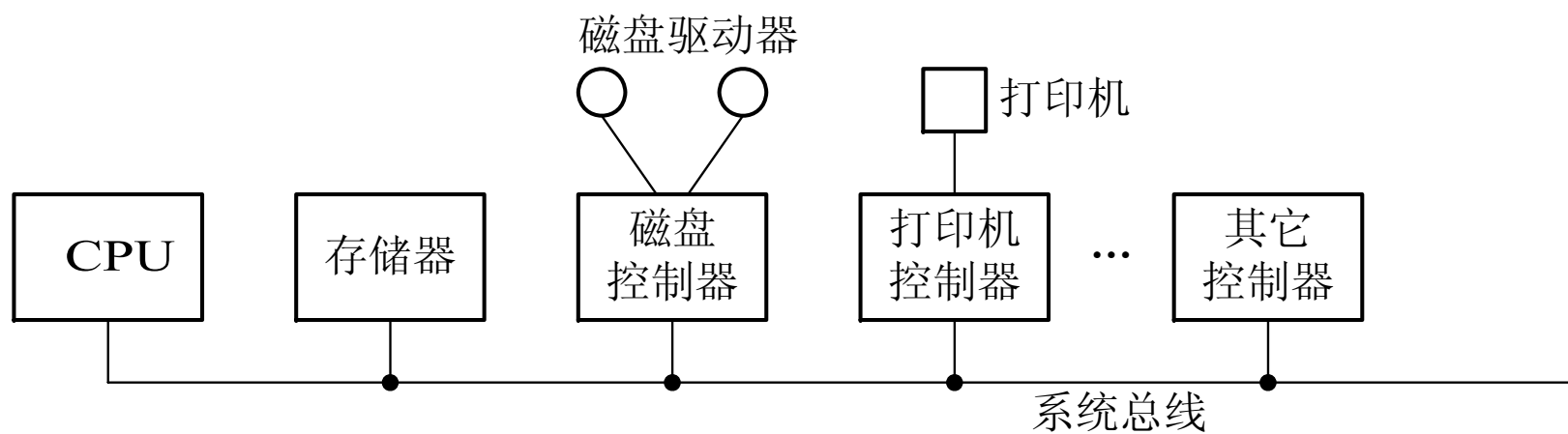
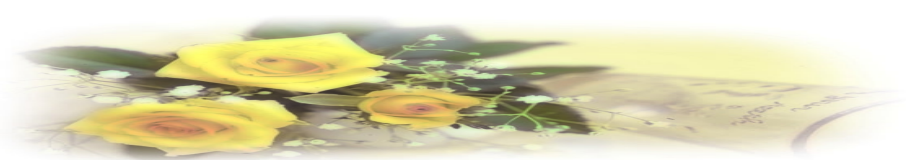


图5-6 总线型I/O系统结构





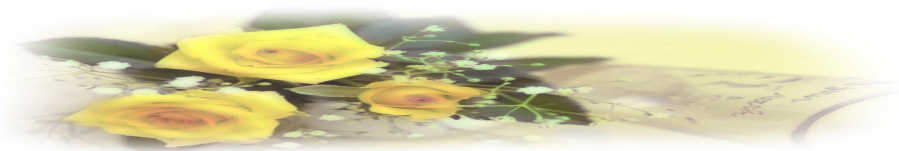
1. ISA和EISA总线

1) ISA (Industry Standard Architecture) 总线

这是为在1984年推出的80286型微机而设计的总线结构。其总线的带宽为8位，最高传输速率为2 Mb/s。之后不久又推出了16位的 (EISA) 总线，其最高传输速率为8 Mb/s，后又升至16 Mb/s，能连接12台设备。

2) EISA (Extended ISA) 总线

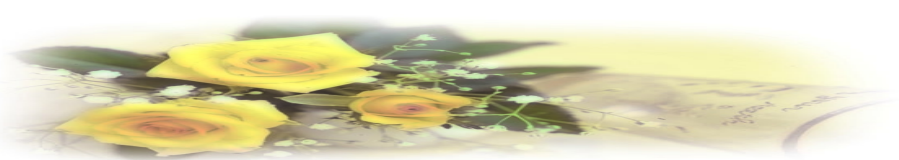
到20世纪80年代末期，ISA总线已难于满足带宽和传输速率的要求，于是人们又开发出扩展ISA(EISA)总线，其带宽为32位，总线的传输速率高达32 Mb/s，同样可以连接12台外部设备。





2. 局部总线(Local Bus)

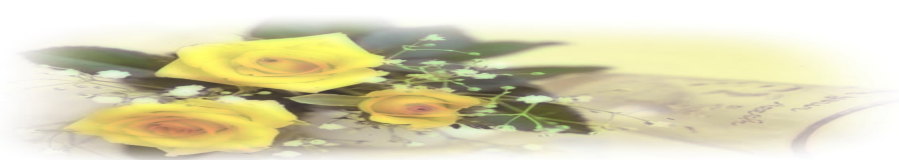
多媒体技术的兴起，特别是全运动视频处理、高保真音响、高速LAN，以及高质量图形处理等技术，都要求总线具有更高的传输速率，这时的EISA总线已难于满足要求，于是，局部总线便应运而生。所谓局部总线，是指将多媒体卡、高速LAN网卡、高性能图形板等，从ISA总线上卸下来，再通过局部总线控制器直接接到CPU总线上，使之与高速CPU总线相匹配，而打印机、FAX/Modem、CDROM等仍挂在ISA总线上。在局部总线中较有影响的是VESA总线和PCI总线。





1) VESA (Video Electronic Standard Association) 总线

该总线的设计思想是以低价位迅速占领市场。VESA总线的带宽为32位，最高传输速率为132 Mb/s。它在20世纪90年代初被推出时，广泛应用于486微机中。但VESA总线仍存在较严重的缺点，比如，它所能连接的设备数仅为2~4台，在控制器中无缓冲，故难于适应处理器速度的不断提高，也不能支持后来出现的Pentium微机。

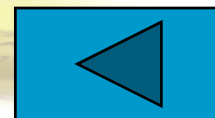
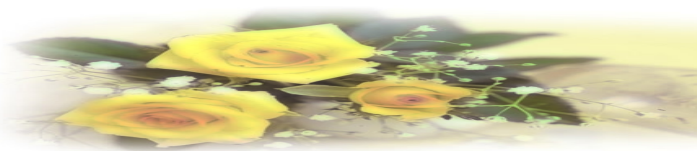




第五章 设备管理

2) PCI (Peripheral Component Interface) 总线

随着Pentium系列芯片的推出，Intel公司分别在1992年和1995年颁布了PCI总线的V1.0和V2.1规范，后者支持64位系统。PCI在CPU和外设间插入一复杂的管理层，用于协调数据传输和提供一致的接口。在管理层中配有数据缓冲，通过该缓冲可将线路的驱动能力放大，使PCI最多能支持10种外设，并使高时钟频率的CPU能很好地运行，最大传输速率可达132 Mb/s。PCI既可连接ISA、EISA等传统型总线，又可支持Pentium的64位系统，是基于奔腾等新一代微处理器而发展的总线。

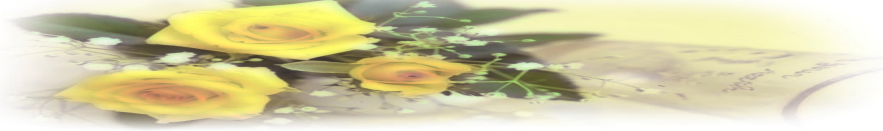




5.2 I/O控制方式

5.2.1 程序I/O方式

早期的计算机系统中，由于无中断机构，处理机对I/O设备的控制采取程序I/O(Programmed I/O)方式，或称为忙—等待方式，即在处理机向控制器发出一条I/O指令启动输入设备输入数据时，要同时把状态寄存器中的忙/闲标志busy置为1，然后便不断地循环测试busy。当busy=1时，表示输入机尚未输完一个字(符)，处理机应继续对该标志进行测试，直至busy=0，表明输入机已将输入数据送入控制器的数据寄存器中。于是处理机将数据寄存器中的数据取出，送入内存指定单元中，这样便完成了一个字(符)的I/O。接着再去启动读下一个数据，并置busy=1。图5-7(a)示出了程序I/O方式的流程。



第五章 设备管理

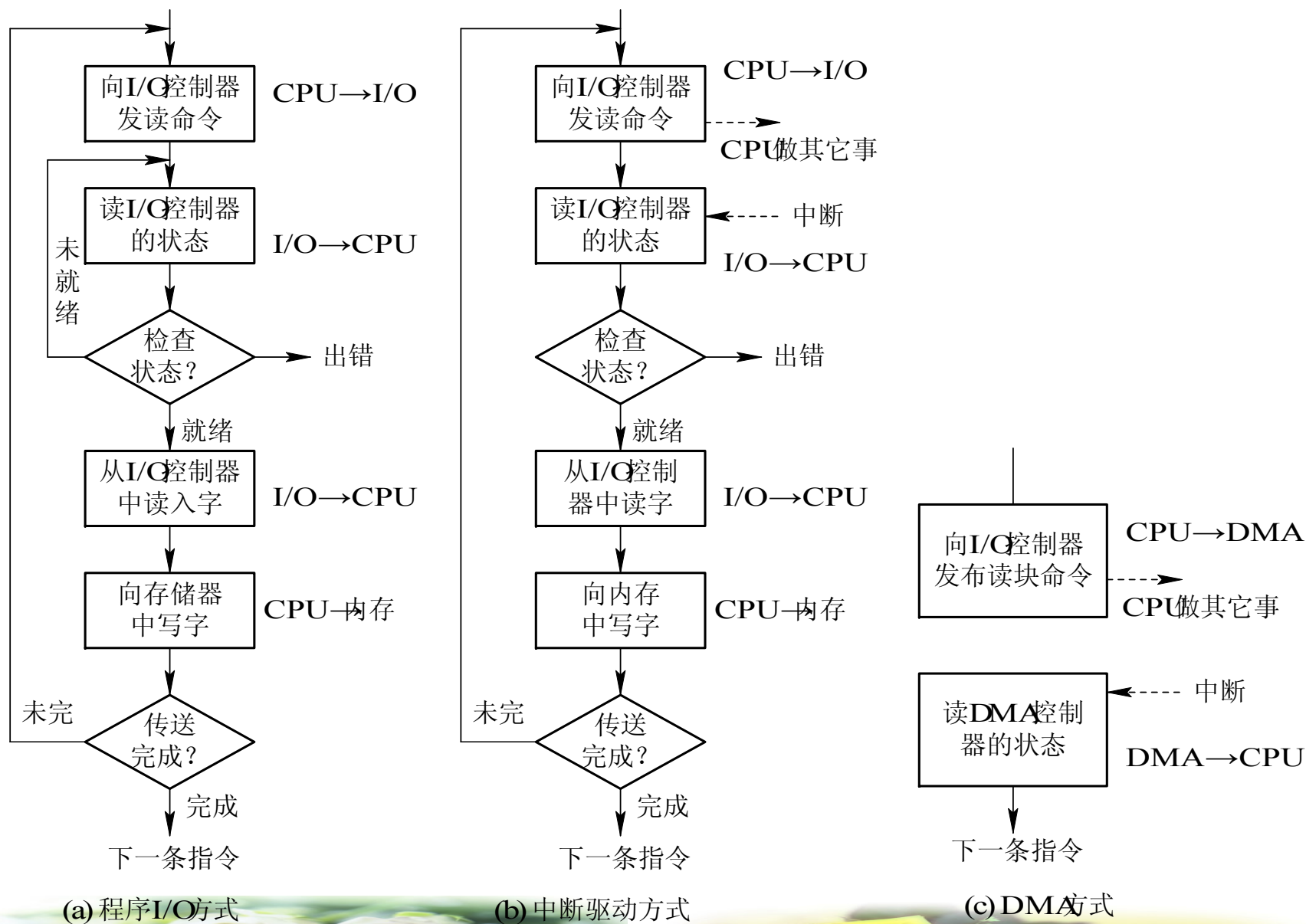
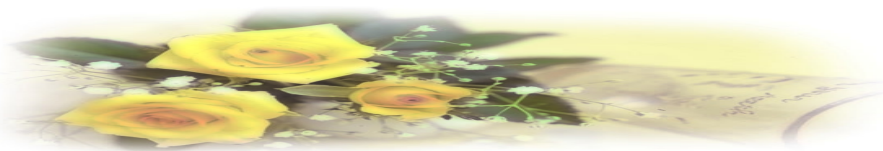


图5-7 程序I/O和中断驱动方式的流程



5.2.2 中断驱动I/O控制方式

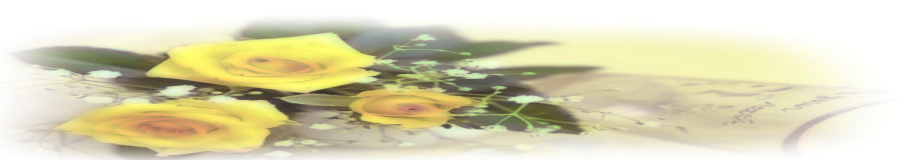
现代计算机系统中，都毫无例外地引入了中断机构，致使对I/O设备的控制，广泛采用中断驱动(Interrupt Driven)方式，即当某进程要启动某个I/O设备工作时，便由CPU向相应的设备控制器发出一条I/O命令，然后立即返回继续执行原来的任务。设备控制器于是按照该命令的要求去控制指定I/O设备。此时，CPU与I/O设备并行操作。例如，在输入时，当设备控制器收到CPU发来的读命令后，便去控制相应的输入设备读数据。一旦数据进入数据寄存器，控制器便通过控制线向CPU发送一中断信号，由CPU检查输入过程中是否出错，若无错，便向控制器发送取走数据的信号，然后再通过控制器及数据线将数据写入内存指定单元中。图 5-7(b)示出了中断驱动方式的流程。





第五章 设备管理

在I/O设备输入每个数据的过程中，由于无需CPU干预，因而可使CPU与I/O设备并行工作。仅当输完一个数据时，才需CPU花费极短的时间去做些中断处理。可见，这样可使CPU和I/O设备都处于忙碌状态，从而提高了整个系统的资源利用率及吞吐量。例如，从终端输入一个字符的时间约为100 ms，而将字符送入终端缓冲区的时间小于0.1 ms。若采用程序I/O方式，CPU约有99.9 ms的时间处于忙—等待的过程中。但采用中断驱动方式后，CPU可利用这99.9 ms的时间去做其它的事情，而仅用0.1 ms的时间来处理由控制器发来的中断请求。可见，中断驱动方式可以成百倍地提高CPU的利用率。



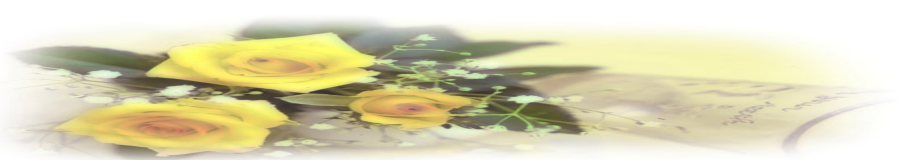


5.2.3 直接存储器访问(DMA) I/O控制方式

1. DMA(Direct Memory Access)控制方式的引入

虽然中断驱动I/O比程序I/O方式更有效,但须注意,它仍是以字(节)为单位进行I/O的,每当完成一个字(节)的I/O时,控制器便要向CPU请求一次中断。换言之,采用中断驱动I/O方式时的CPU是以字(节)为单位进行干预的。如果将这种方式用于块设备的I/O,显然是极其低效的。例如,为了从磁盘中读出1 KB的数据块,需要中断CPU 1K次。为了进一步减少CPU对I/O的干预而引入了直接存储器访问方式,见图 5-7(c)所示。该方式的特点是:

(1) 数据传输的基本单位是数据块,即在CPU与I/O设备之间,每次传送至少一个数据块;

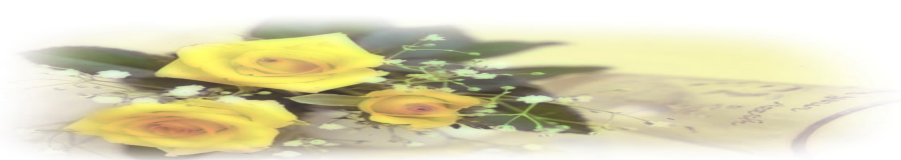




(2) 所传送的数据是从设备直接送入内存的，或者相反；

(3) 仅在传送一个或多个数据块的开始和结束时，才需CPU干预，整块数据的传送是在控制器的控制下完成的。

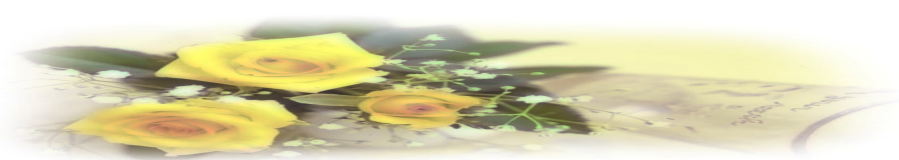
可见，DMA方式较之中断驱动方式，又是成百倍地减少了CPU对I/O的干预，进一步提高了CPU与I/O设备的并行操作程度。





2. DMA控制器的组成

DMA控制器由三部分组成：主机与DMA控制器的接口；DMA控制器与块设备的接口；I/O控制逻辑。图5-8 示出了DMA控制器的组成。这里主要介绍主机与控制器之间的接口。



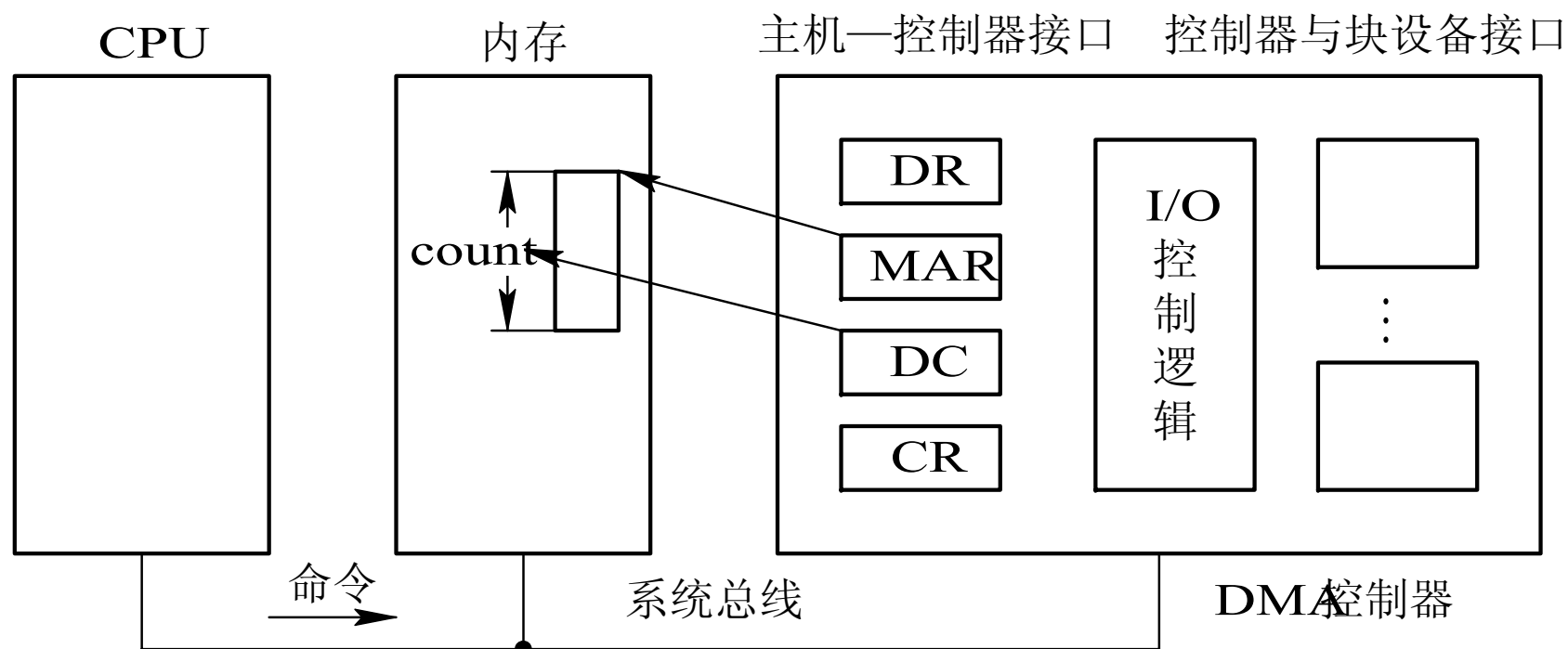
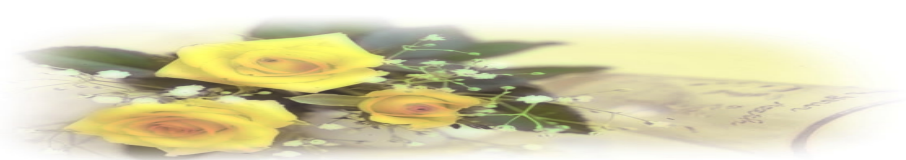


图5-8 DMA控制器的组成





第五章 设备管理

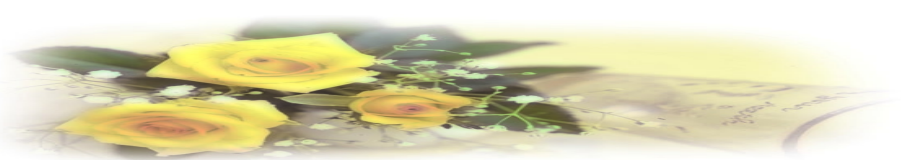
为了实现在主机与控制器之间成块数据的直接交换，必须在DMA控制器中设置如下四类寄存器：

(1) 命令/状态寄存器(CR)。用于接收从CPU发来的I/O命令，或有关控制信息，或设备的状态。

(2) 内存地址寄存器(MAR)。在输入时，它存放把数据从设备传送到内存的起始目标地址；在输出时，它存放由内存到设备的内存源地址。

(3) 数据寄存器(DR)。用于暂存从设备到内存，或从内存到设备的数据。

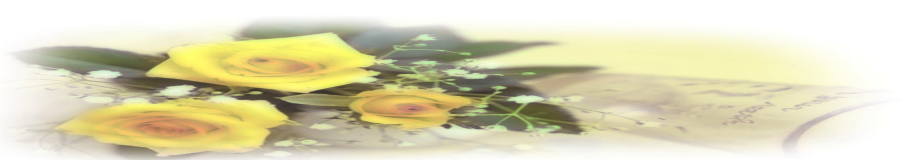
(4) 数据计数器(DC)。存放本次CPU要读或写的字(节)数。





3. DMA工作过程

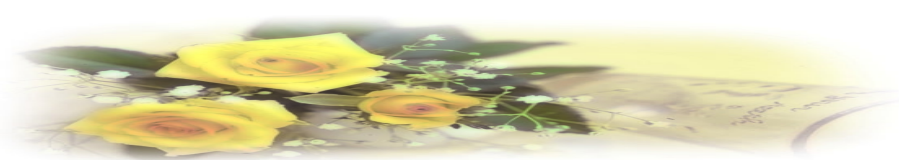
我们以从磁盘读入数据为例，来说明DMA方式的工作流程。当CPU要从磁盘读入一数据块时，便向磁盘控制器发送一条读命令。该命令被送到其中的命令寄存器(CR)中。同时，还须发送本次要将数据读入的内存起始目标地址，该地址被送入内存地址寄存器(MAR)中；本次要读数据的字(节)数则送入数据计数器(DC)中，还须将磁盘中的源地址直接送至DMA控制器的I/O控制逻辑上。然后，启动DMA控制器进行数据传送，以后，CPU便可去处理其它任务。此后，整个数据传送过程便由DMA控制器进行控制。





第五章 设备管理

当DMA控制器已从磁盘中读入一个字(节)的数据并送入数据寄存器(DR)后,再挪用一個存储器周期,将该字(节)传送到MAR所指示的内存单元中。接着便对MAR内容加1,将DC内容减1。若减1后DC内容不为0,表示传送未完,便继续传送下一个字(节);否则,由DMA控制器发出中断请求。图5-9是DMA方式的工作流程。



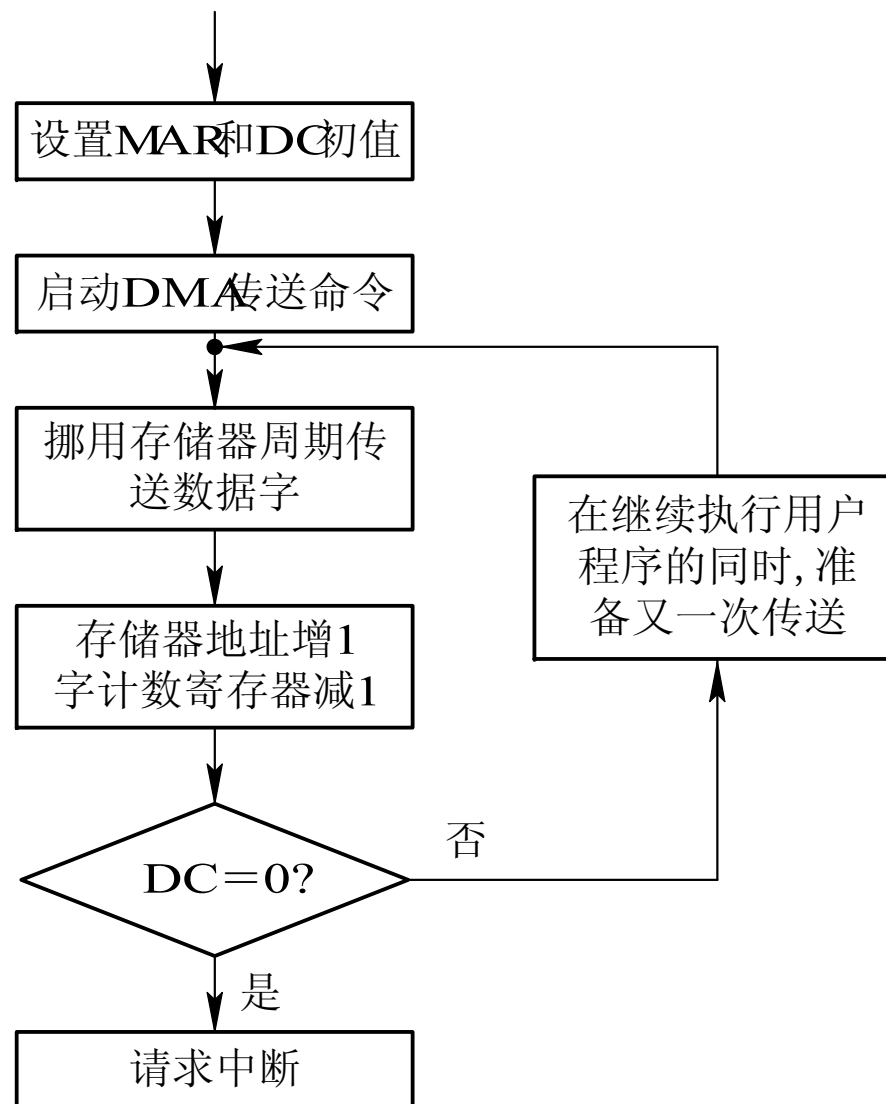


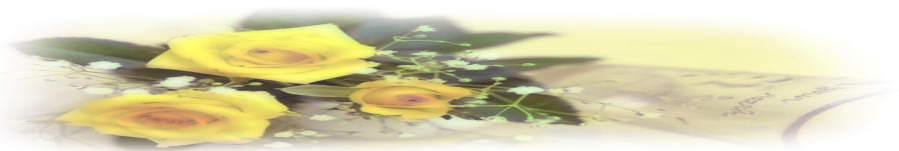
图5-9 DMA方式的工作流程图



5.2.4 I/O通道控制方式

1. I/O通道控制方式的引入

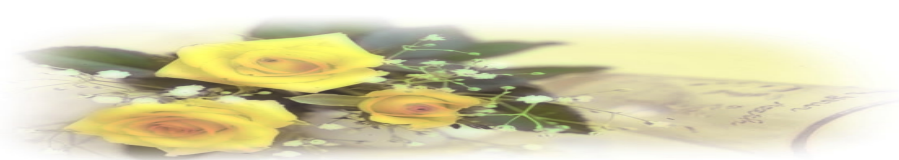
虽然DMA方式比起中断方式来已经显著地减少了CPU的干预，即已由以字(节)为单位的干预减少到以数据块为单位的干预，但CPU每发出一条I/O指令，也只能去读(或写)一个连续的数据块。而当我们一次去读多个数据块且将它们分别传送到不同的内存区域，或者相反时，则须由CPU分别发出多条I/O指令及进行多次中断处理才能完成。





第五章 设备管理

I/O通道方式是DMA方式的发展，它可进一步减少CPU的干预，即把对一个数据块的读(或写)为单位的干预减少为对一组数据块的读(或写)及有关的管理为单位的干预。同时，又可实现CPU、通道和I/O设备三者的并行操作，从而更有效地提高整个系统的资源利用率。例如，当CPU要完成一组相关的读(或写)操作及有关控制时，只需向I/O通道发送一条I/O指令，以给出其所要执行的通道程序的首址和要访问的I/O设备，通道接到该指令后，通过执行通道程序便可完成CPU指定的I/O任务。





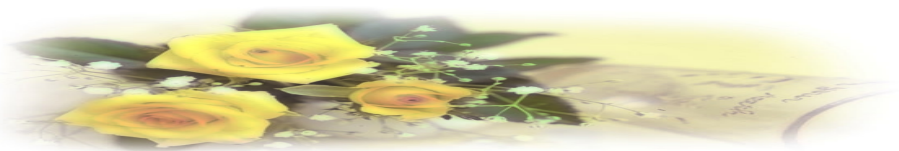
2. 通道程序

通道是通过执行通道程序，并与设备控制器共同实现对I/O设备的控制的。通道程序是由一系列通道指令(或称为通道命令)所构成的。通道指令与一般的机器指令不同，在它的每条指令中都包含下列诸信息：

(1) 操作码。操作码规定了指令所执行的操作，如读、写、控制等操作。

(2) 内存地址。内存地址标明字符送入内存(读操作)和从内存取出(写操作)时的内存首址。

(3) 计数。该信息表示本条指令所要读(或写)数据的字节数。

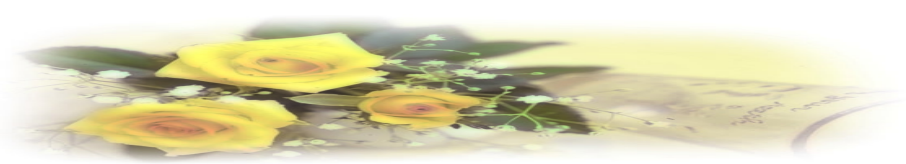




第五章 设备管理

(4) 通道程序结束位P。该位用于表示通道程序是否结束。
P=1表示本条指令是通道程序的最后一条指令。

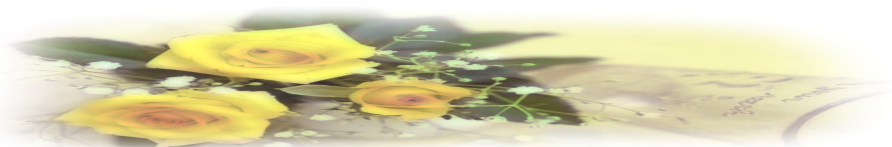
(5) 记录结束标志R。R=0表示本通道指令与下一条指令所处理的数据是同属于一个记录；R=1表示这是处理某记录的最后一条指令。





第五章 设备管理

下面示出了一个由六条通道指令所构成的简单的通道程序。该程序的功能是将内存中不同地址的数据写成多个记录。其中，前三条指令是分别将813~892单元中的80个字符和1034~1173单元中的140个字符及5830~5889单元中的60个字符写成一个记录；第4条指令是单独写一个具有300个字符的记录；第5、6条指令共写含500个字符的记录。



第五章 设备管理



操 作	P	R	计 数	内 存 地 址
WRITE	0	0	80	813
WRITE	0	0	140	1034
WRITE	0	1	60	5830
WRITE	0	1	300	2000
WRITE	0	0	250	1650
WRITE	1	1	250	2720

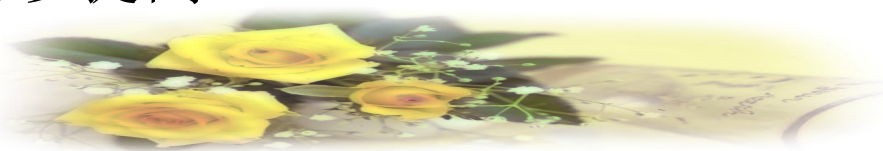




5.3 缓冲管理

5.3.1 缓冲的引入

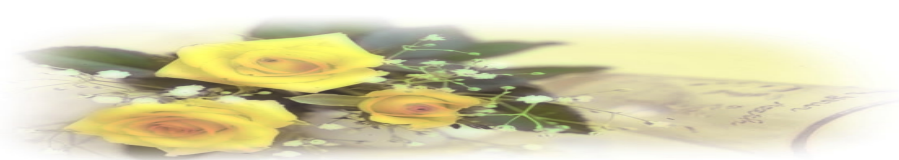
(1) 缓和CPU与I/O设备间速度不匹配的矛盾。事实上，凡在数据到达速率与其离去速率不同的地方，都可设置缓冲区，以缓和它们之间速率不匹配的矛盾。众所周知，CPU的运算速率远远高于I/O设备的速率，如果没有缓冲区，则在输出数据时，必然会由于打印机的速度跟不上而使CPU停下来等待；然而在计算阶段，打印机又空闲无事。显然，如果在打印机或控制器中设置一缓冲区，用于快速暂存程序的输出数据，以后由打印机“慢慢地”从中取出数据打印，这样，就可提高CPU的工作效率。类似地，在输入设备与CPU之间也设置缓冲区，也可使CPU的工作效率得以提高。





第五章 设备管理

(2) 减少对CPU的中断频率，放宽对CPU中断响应时间的限制。在远程通信系统中，如果从远地终端发来的数据仅用一位缓冲来接收，如图5-10(a)所示，则必须在每收到一位数据时便中断一次CPU，这样，对于速率为9.6 Kb/s的数据通信来说，就意味着其中断CPU的频率也为9.6 Kb/s，即每100 μ s就要中断CPU一次，而且CPU必须在100 μ s内予以响应，否则缓冲区内的数据将被冲掉。倘若设置一个具有8位的缓冲(移位)寄存器，如图5-10(b)所示，则可使CPU被中断的频率降低为原来的1/8；若再设置一个8位寄存器，如图5-10(c)所示，则又可将CPU对中断的响应时间放宽到800 μ s。



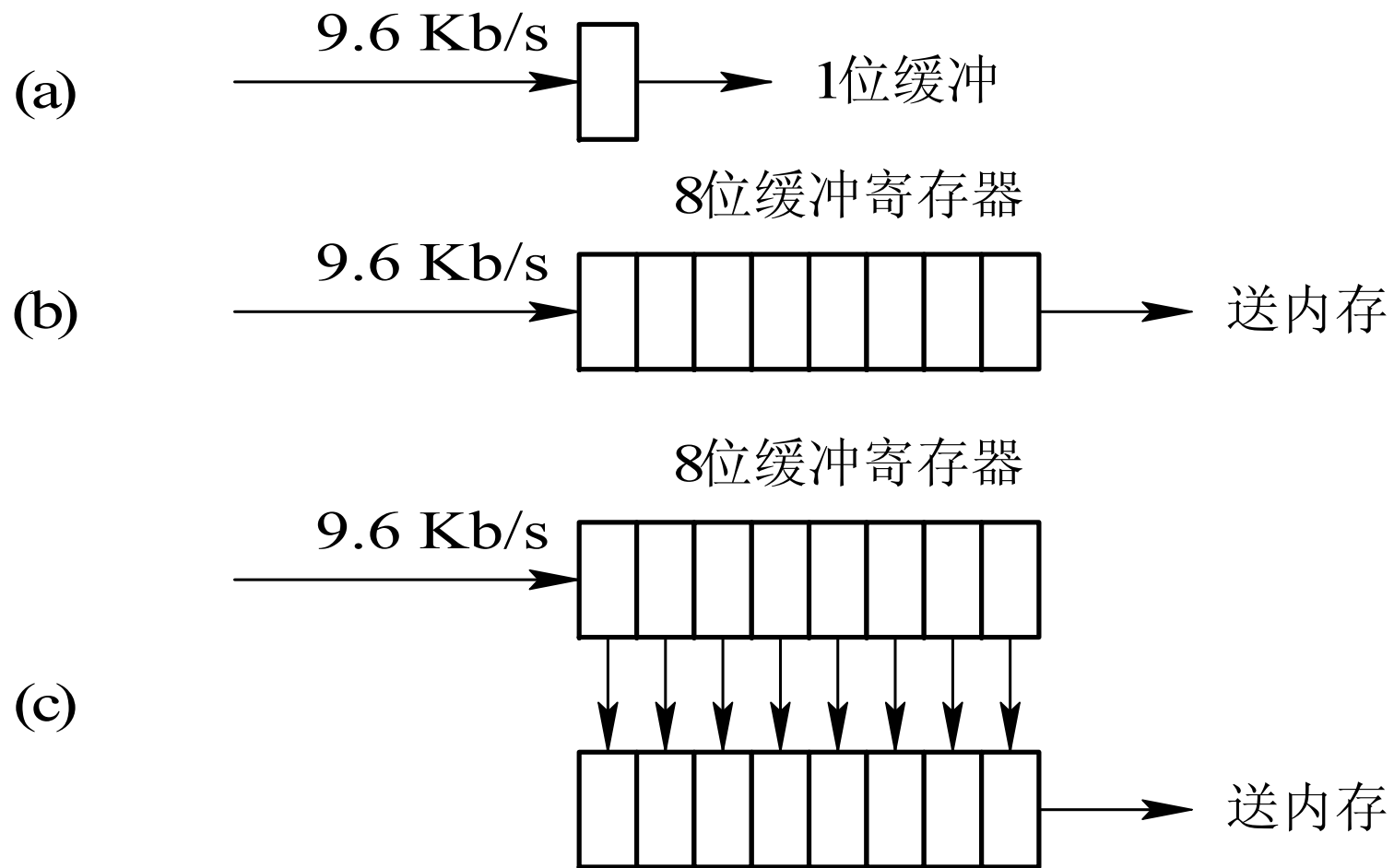
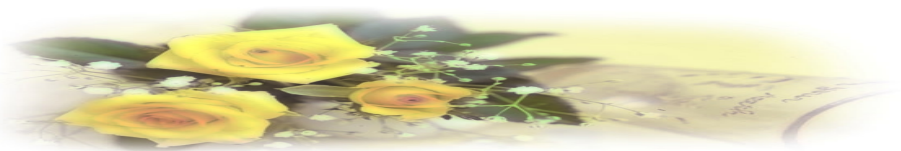


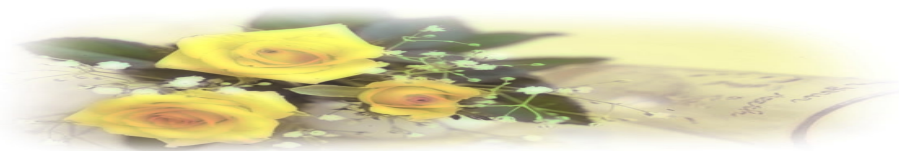
图5-10 利用缓冲寄存器实现缓冲





第五章 设备管理

(3) 提高CPU和I/O设备之间的并行性。缓冲的引入可显著地提高CPU和I/O设备间的并行操作程度，提高系统的吞吐量和设备的利用率。例如，在CPU和打印机之间设置了缓冲区后，便可使CPU与打印机并行工作。

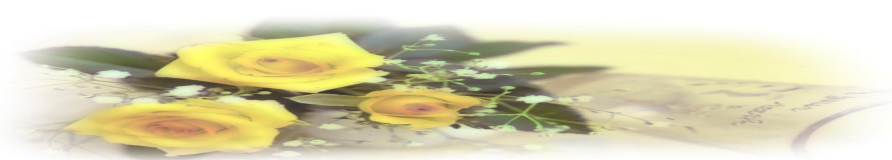




5.3.2 单缓冲和双缓冲

1. 单缓冲(Single Buffer)

在单缓冲情况下，每当用户进程发出一I/O请求时，操作系统便在主存中为之分配一缓冲区，如图5-11 所示。在块设备输入时，假定从磁盘把一块数据输入到缓冲区的时间为 T ，操作系统将该缓冲区中的数据传送到用户区的时间为 M ，而CPU对这一块数据处理(计算)的时间为 C 。由于 T 和 C 是可以并行的(见图5-11)，当 $T > C$ 时，系统对每一块数据的处理时间为 $M + T$ ，反之则为 $M + C$ ，故可把系统对每一块数据的处理时间表示为 $\text{Max}(C, T) + M$ 。



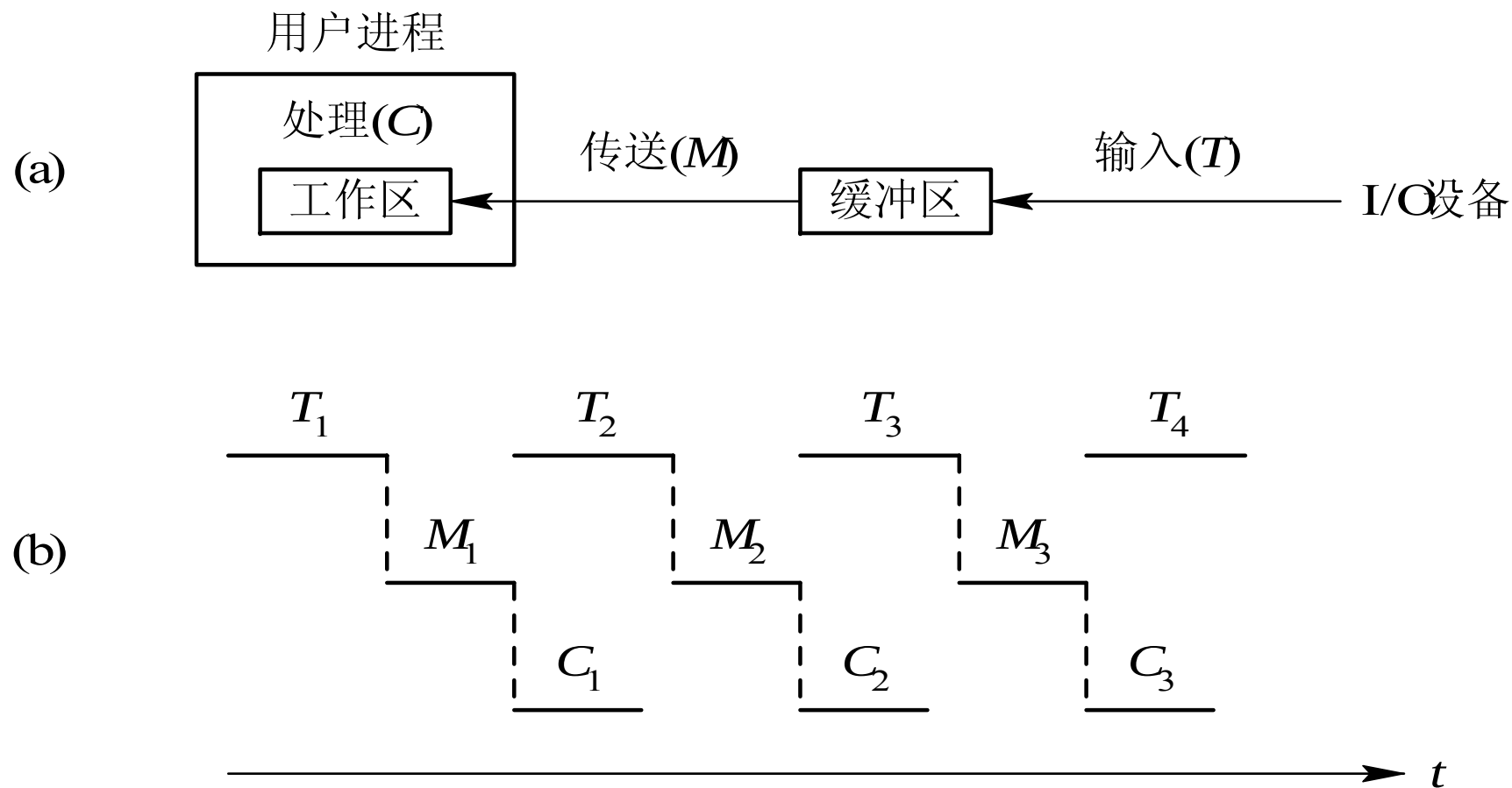
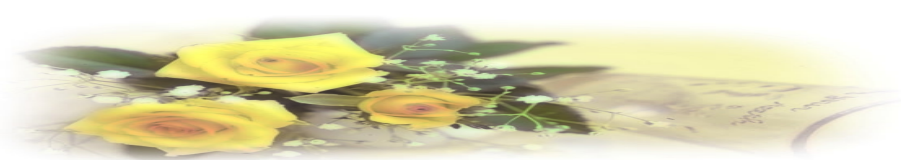


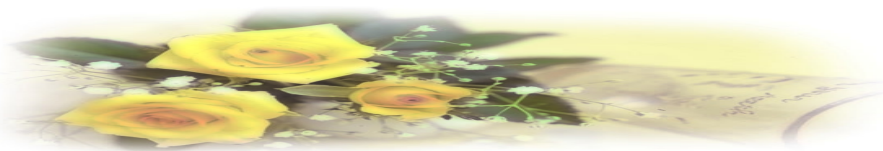
图5-11 单缓冲工作示意图





2. 双缓冲(Double Buffer)

为了加快输入和输出速度，提高设备利用率，人们又引入了双缓冲区机制，也称为缓冲对换(Buffer Swapping)。在设备输入时，先将数据送入第一缓冲区，装满后便转向第二缓冲区。此时操作系统可以从第一缓冲区中移出数据，并送入用户进程(见图5-12)。接着由CPU对数据进行计算。在双缓冲时，系统处理一块数据的时间可以粗略地认为是 $\text{Max}(C, T)$ 。如果 $C < T$ ，可使块设备连续输入；如果 $C > T$ ，则可使CPU不必等待设备输入。对于字符设备，若采用行输入方式，则采用双缓冲通常能消除用户的等待时间，即用户在输入完第一行之后，在CPU执行第一行中的命令时，用户可继续向第二缓冲区输入下一行数据。



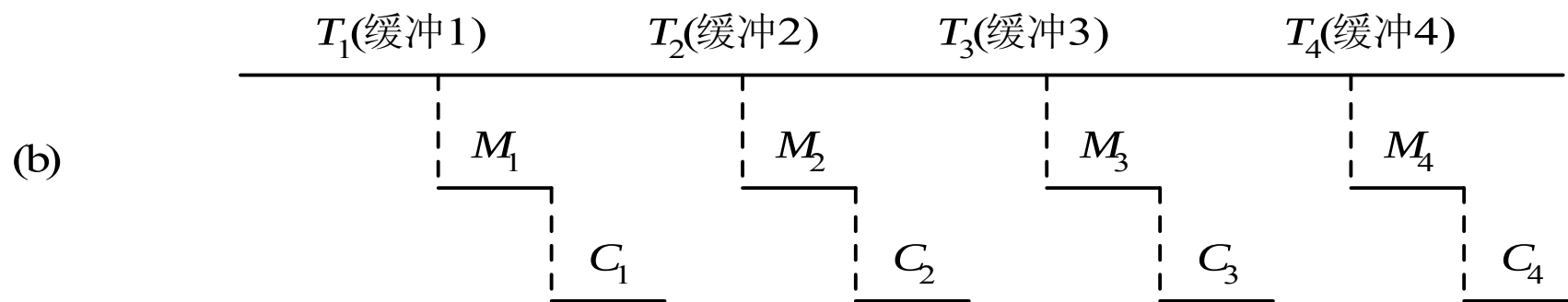
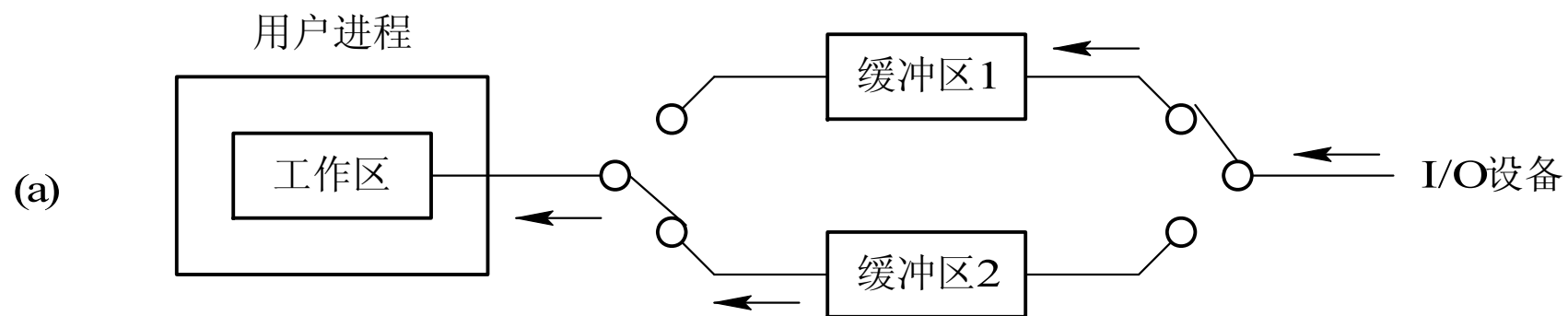
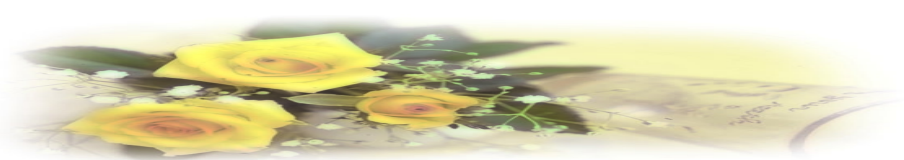
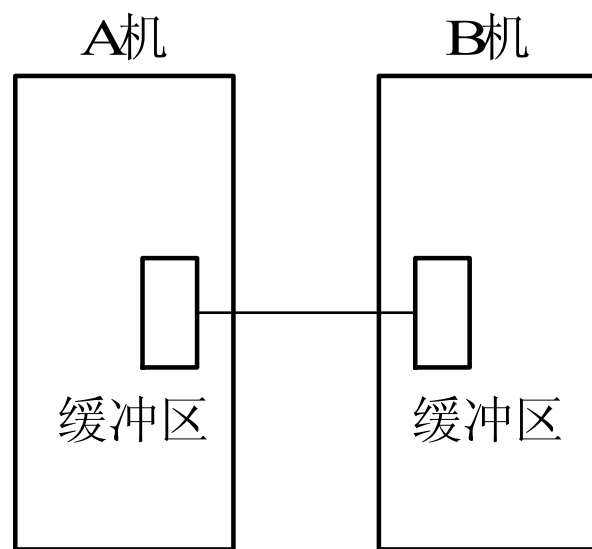
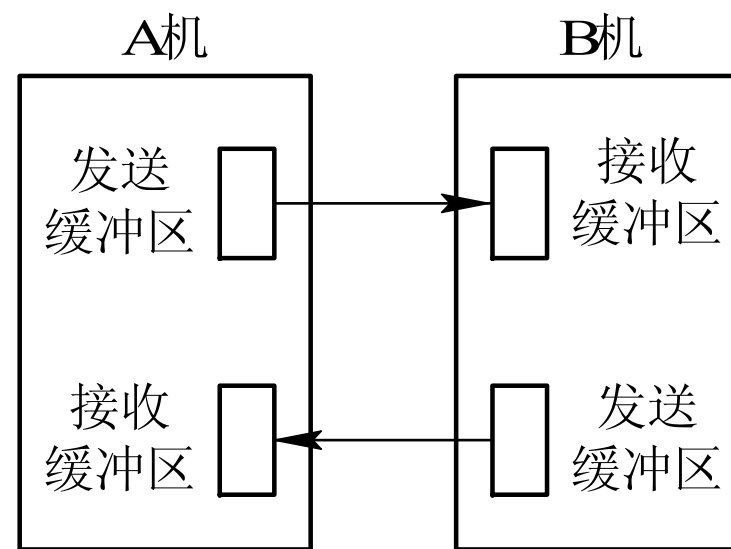


图5-12 双缓冲工作示意图



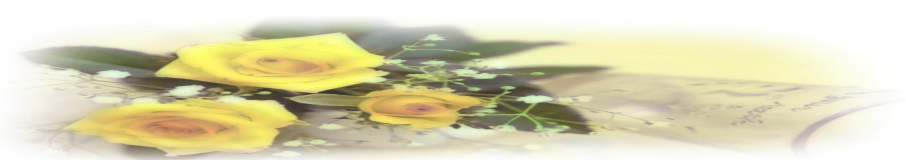


(a) 单缓冲



(b) 双缓冲

图5-13 双机通信时缓冲区的设置

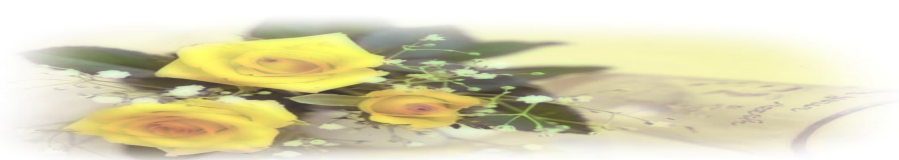




5.3.3 循环缓冲

1. 循环缓冲的组成

(1) 多个缓冲区。在循环缓冲中包括多个缓冲区，其每个缓冲区的大小相同。作为输入的多缓冲区可分为三种类型：用于装输入数据的空缓冲区R、已装满数据的缓冲区G以及计算进程正在使用的现行工作缓冲区C，如图5-14所示。



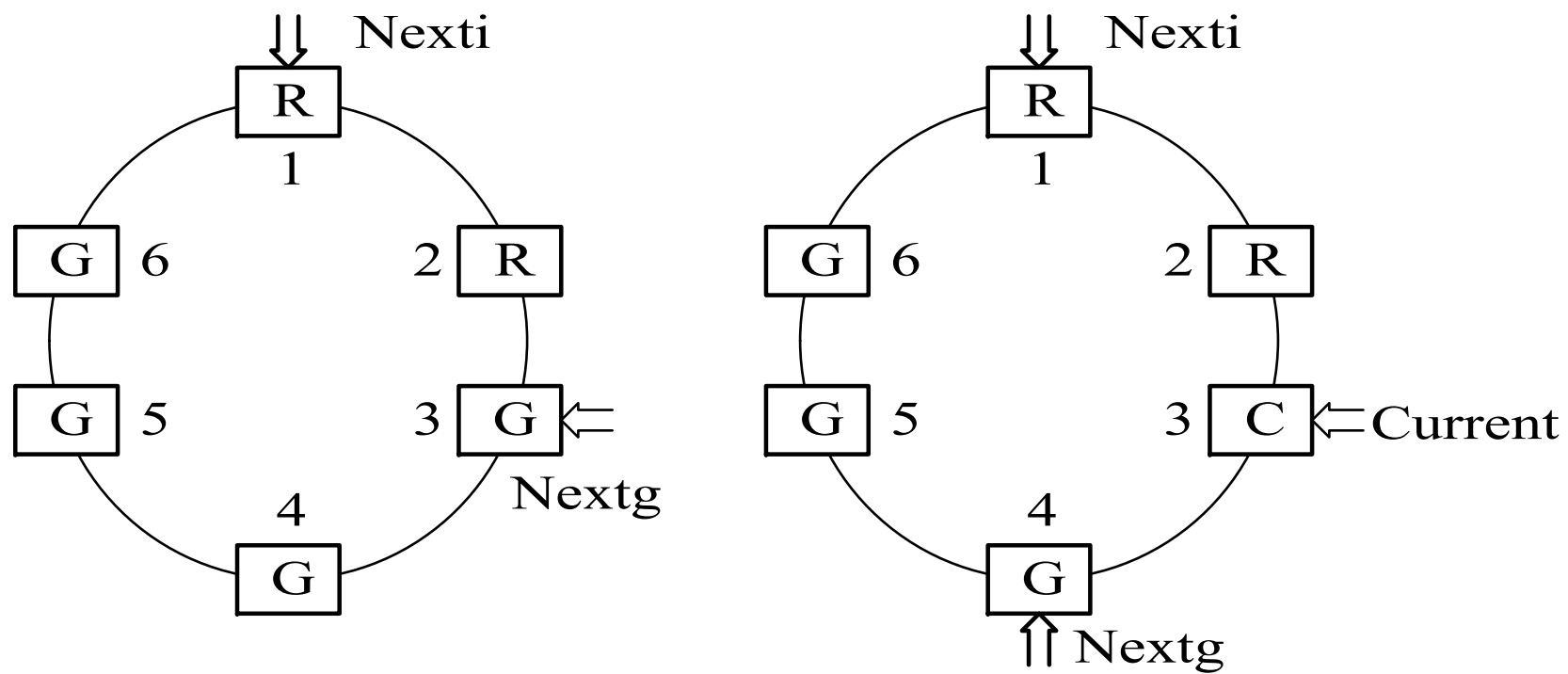
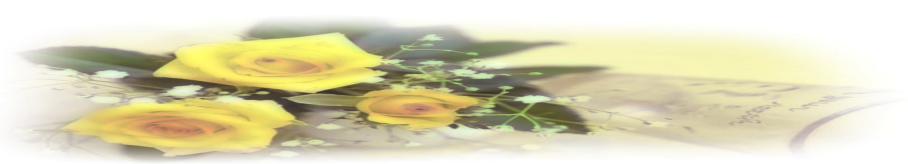


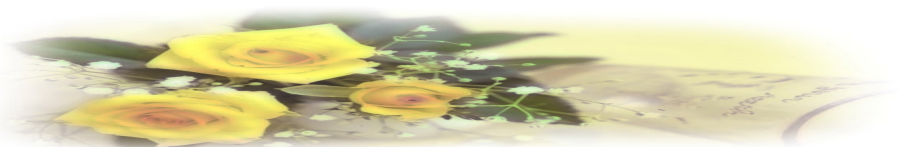
图5-14 循环缓冲





第五章 设备管理

(2) 多个指针。作为输入的缓冲区可设置三个指针：用于指示计算进程下一个可用缓冲区G的指针Nextg、指示输入进程下次可用的空缓冲区R的指针Nexti，以及用于指示计算进程正在使用的缓冲区C的指针Current。

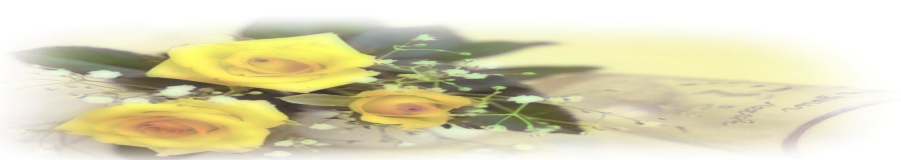




2. 循环缓冲区的使用

计算进程和输入进程可利用下述两个过程来使用循环缓冲区。

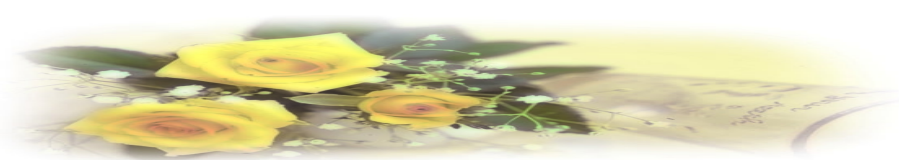
(1) Getbuf过程。当计算进程要使用缓冲区中的数据时，可调用Getbuf过程。该过程将由指针Nextg所指示的缓冲区提供给进程使用，相应地，须把它改为现行工作缓冲区，并令Current指针指向该缓冲区的第一个单元，同时将Nextg移向下一个G缓冲区。类似地，每当输入进程要使用空缓冲区来装入数据时，也调用Getbuf过程，由该过程将指针Nexti所指示的缓冲区提供给输入进程使用，同时将Nexti指针移向下一个R缓冲区。





第五章 设备管理

(2) Releasebuf过程。当计算进程把C缓冲区中的数据提取完毕时，便调用Releasebuf过程，将缓冲区C释放。此时，把该缓冲区由当前(现行)工作缓冲区C改为空缓冲区R。类似地，当输入进程把缓冲区装满时，也应调用Releasebuf过程，将该缓冲区释放，并改为G缓冲区。

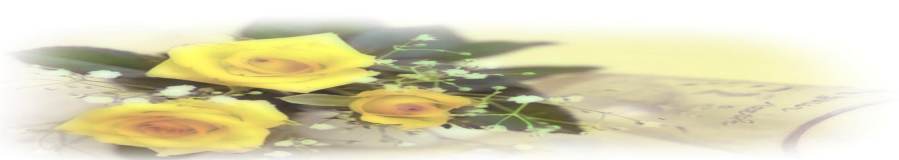




3. 进程同步

使用输入循环缓冲，可使输入进程和计算进程并行执行。相应地，指针Nexti和指针Nextg将不断地沿着顺时针方向移动，这样就可能出现下述两种情况：

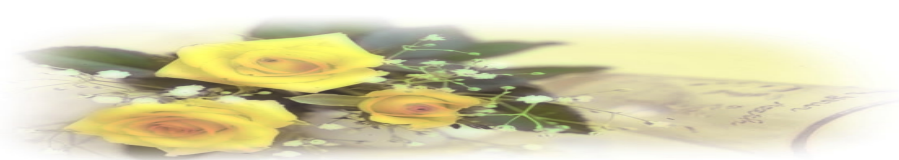
(1) Nexti指针追赶上Nextg指针。这意味着输入进程输入数据的速度大于计算进程处理数据的速度，已把全部可用的空缓冲区装满，再无缓冲区可用。此时，输入进程应阻塞，直到计算进程把某个缓冲区中的数据全部提取完，使之成为空缓冲区R，并调用Releasebuf过程将它释放时，才将输入进程唤醒。这种情况被称为系统受计算限制。





第五章 设备管理

(2) Nextg指针追赶上Nexti指针。这意味着输入数据的速度低于计算进程处理数据的速度，使全部装有输入数据的缓冲区都被抽空，再无装有数据的缓冲区供计算进程提取数据。这时，计算进程只能阻塞，直至输入进程又装满某个缓冲区，并调用Releasebuf过程将它释放时，才去唤醒计算进程。这种情况被称为系统受I/O限制。



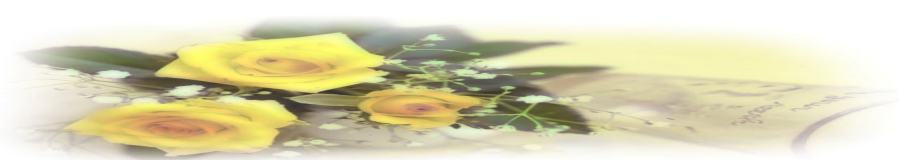


5.3.4 缓冲池

1. 缓冲池的组成

对于既可用于输入又可用于输出的公用缓冲池，其中至少应含有以下三种类型的缓冲区：

- ① 空(闲)缓冲区；
- ② 装满输入数据的缓冲区；
- ③ 装满输出数据的缓冲区。



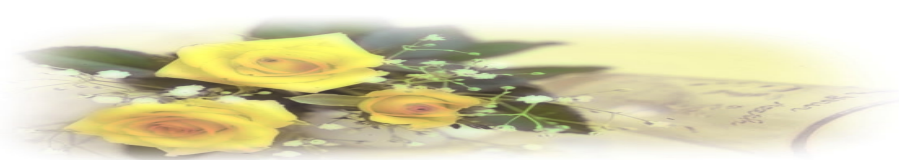


第五章 设备管理

为了管理上的方便，可将相同类型的缓冲区链成一个队列，于是可形成以下三个队列：

(1) 空缓冲队列emq。这是由空缓冲区所链成的队列。其队首指针 $F(emq)$ 和队尾指针 $L(emq)$ 分别指向该队列的首缓冲区和尾缓冲区。

(2) 输入队列inq。这是由装满输入数据的缓冲区所链成的队列。其队首指针 $F(inq)$ 和队尾指针 $L(inq)$ 分别指向该队列的首缓冲区和尾缓冲区。

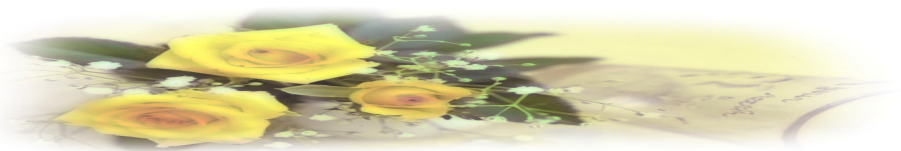




第五章 设备管理

(3) 输出队列outq。这是由装满输出数据的缓冲区所链成的队列。其队首指针 $F(outq)$ 和队尾指针 $L(outq)$ 分别指向该队列的首缓冲区和尾缓冲区。

除了上述三个队列外，还应具有四种工作缓冲区：①用于收容输入数据的工作缓冲区；②用于提取输入数据的工作缓冲区；③用于收容输出数据的工作缓冲区；④用于提取输出数据的工作缓冲区。



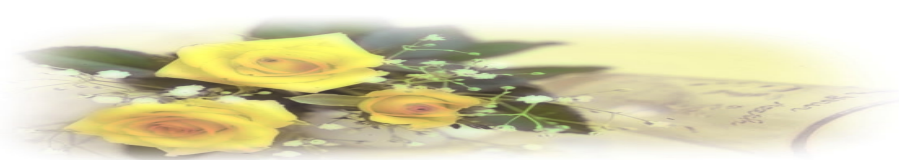


2. Getbuf过程和Putbuf过程

在“数据结构”课程中，曾介绍过队列和对队列进行操作的两个过程，它们是：

(1) Addbuf(type, number)过程。该过程用于将由参数number所指示的缓冲区B挂在type队列上。

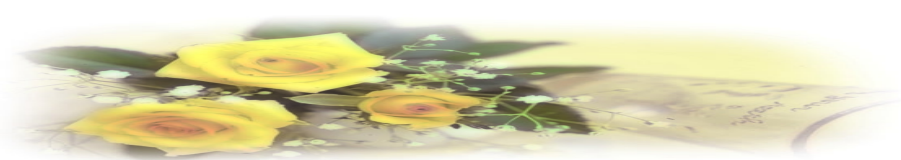
(2) Takebuf(type)过程。该过程用于从type所指示的队列的队首摘下一个缓冲区。





第五章 设备管理

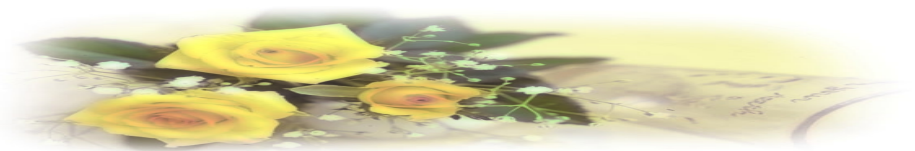
为使诸进程能互斥地访问缓冲池队列，可为每一队列设置一个互斥信号量 $MS(type)$ 。此外，为了保证诸进程同步地使用缓冲区，又为每个缓冲队列设置了一个资源信号量 $RS(type)$ 。既可实现互斥又可保证同步的Getbuf过程和Putbuf过程描述如下：





第五章 设备管理

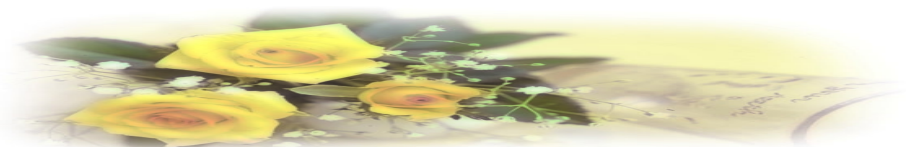
```
Procedure Getbuf(type)
begin
    Wait(RS(type));
    Wait(MS(type));
    B(number):=Takebuf(type);
    Signal(MS(type));
end
Procedure Putbuf(type, number)
begin
    Wait(MS(type));
    Addbuf(type, number);
    Signal(MS(type));
    Signal(RS(type));
end
```





3. 缓冲区的工作方式

缓冲区可以工作在收容输入、提取输入、收容输出和提取输出四种工作方式下，如图5-15 所示。



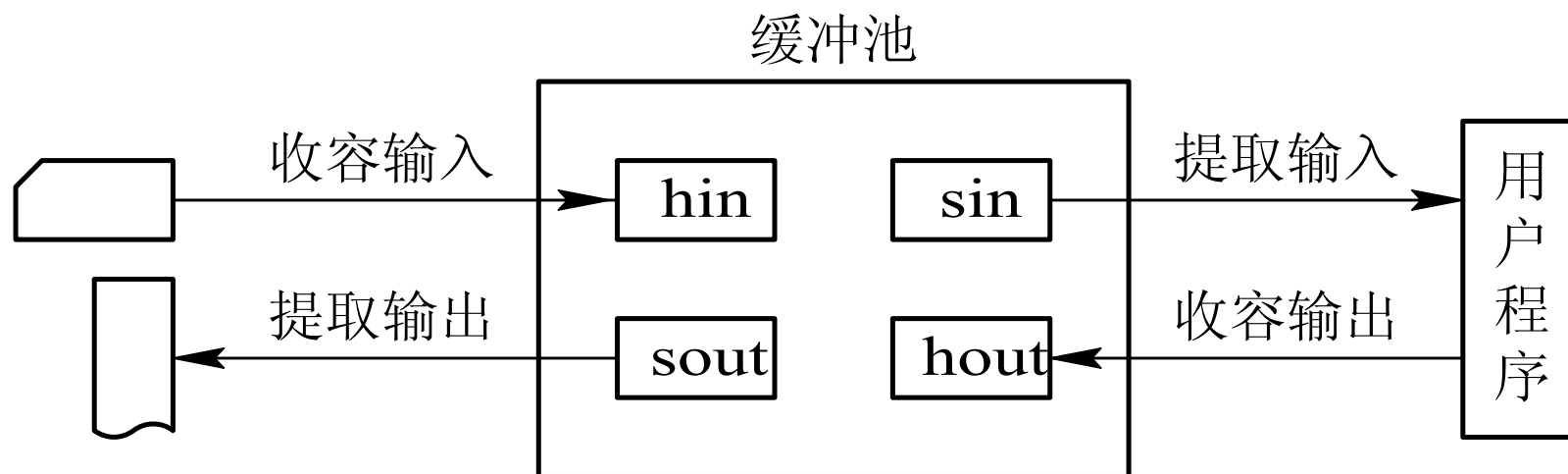
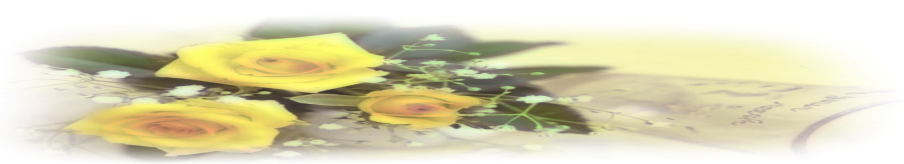


图5-15 缓冲区的工作方式

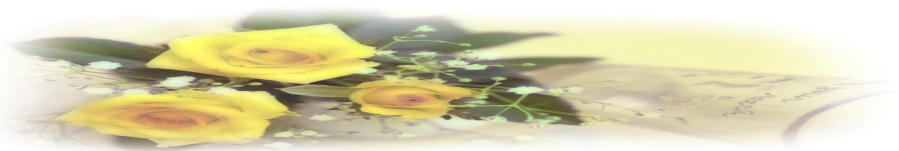




第五章 设备管理

(1) 收容输入。在输入进程需要输入数据时，便调用Getbuf(emq)过程，从空缓冲队列emq的队首摘下一空缓冲区，把它作为收容输入工作缓冲区hin。然后，把数据输入其中，装满后再调用Putbuf(inq, hin)过程，将该缓冲区挂在输入队列inq上。

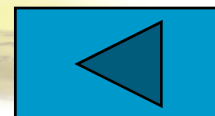
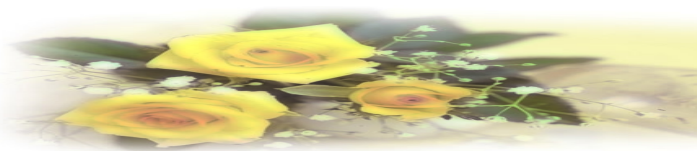
(2) 提取输入。当计算进程需要输入数据时，调用Getbuf(inq)过程，从输入队列inq的队首取得一个缓冲区，作为提取输入工作缓冲区(sin)，计算进程从中提取数据。计算进程用完该数据后，再调用Putbuf(emq, sin)过程，将该缓冲区挂到空缓冲队列emq上。





(3) 收容输出。当计算进程需要输出时，调用Getbuf(emq)过程从空缓冲队列emq的队首取得一个空缓冲区，作为收容输出工作缓冲区hout。当其中装满输出数据后，又调用Putbuf(outq, hout)过程，将该缓冲区挂在outq末尾。

(4) 提取输出。由输出进程调用Getbuf(outq)过程，从输出队列的队首取得一装满输出数据的缓冲区，作为提取输出工作缓冲区sout。在数据提取完后，再调用Putbuf(emq, sout)过程，将该缓冲区挂在空缓冲队列末尾。

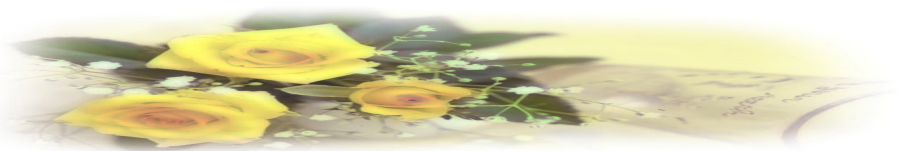




5.4 I/O 软件

5.4.1 I/O软件的设计目标和原则

计算机系统中包含了众多的I/O设备，种类繁多，硬件构造复杂，物理特性各异，速度慢，与CPU速度不匹配，并涉及到大量专用CPU及数字逻辑运算等细节，如寄存器、中断、控制字符和设备字符集等，造成对设备的操作和管理非常复杂和琐碎。因此，从系统的观点出发，采用多种技术和措施，解决由于外部设备与CPU速度不匹配所引起的问题，提高主机和外设的并行工作能力，提高系统效率，成为操作系统的一个重要目标。另一方面，对设备的操作和管理的复杂性，也给用户的使用带来了极大的困难。

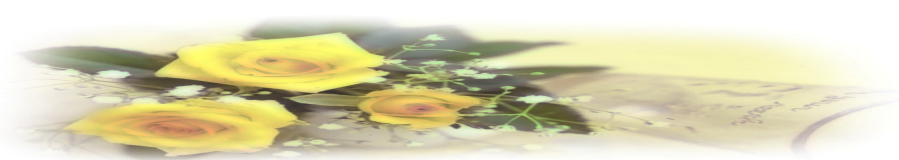




第五章 设备管理

用户必须掌握I/O 系统的原理，对接口和控制器及设备的物理特性要有深入了解，这就使计算机的推广应用受到很大限制。所以，设法消除或屏蔽设备硬件内部的低级处理过程，为用户提供一个简便、易用、抽象的逻辑设备接口，保证用户安全、方便地使用各类设备，也是I/O软件设计的一个重要原则。

具体而言，I/O软件应达到下面的几个目标：

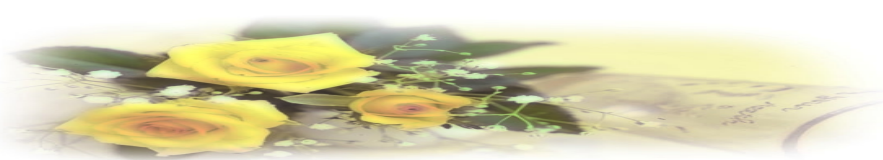




1) 与具体设备无关

对于I/O系统中许多种类不同的设备，作为程序员，只需要知道如何使用这些资源来完成所需要的操作，而无需了解设备的有关具体实现细节。例如，应用程序访问文件时，不必去考虑被访问的是硬盘、软盘还是CD-ROM；对于管理软件，也无需因为I/O设备变化，而重新编写涉及设备管理的程序。

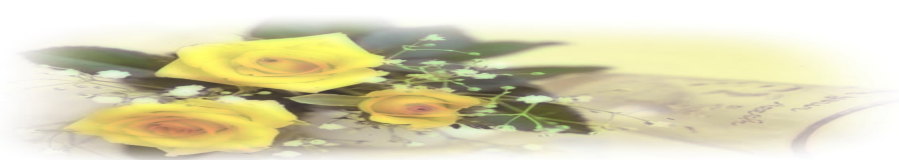
为了提高OS的可移植性和易适应性，I/O软件应负责屏蔽设备的具体细节，向高层软件提供抽象的逻辑设备，并完成逻辑设备与具体物理设备的映射。





2) 统一命名

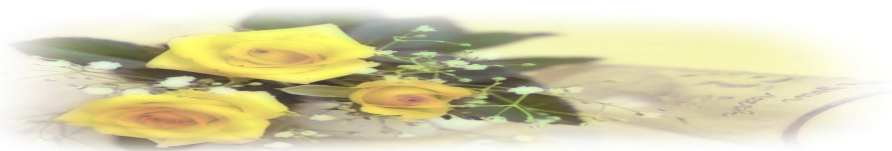
要实现上述的设备无关性，其中一项重要的工作就是如何给I/O设备命名。不同的操作系统有不同的命名规则，一般而言，是在系统中对各类设备采取预先设计的、统一的逻辑名称进行命名，所有软件都以逻辑名称访问设备。这种统一命名与具体设备无关，换言之，同一个逻辑设备的名称，在不同的情况下可能对应于不同的物理设备。





3) 对错误的处理

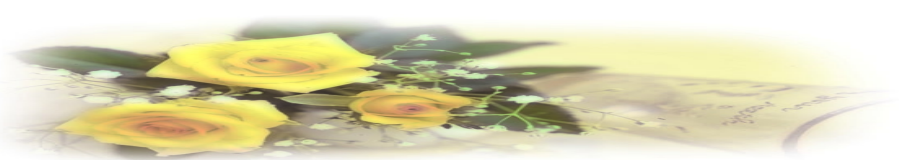
一般而言，错误多数是与设备紧密相关的，因此对于错误的处理，应该尽可能在接近硬件的层面处理，在低层软件能够解决的错误就不让高层软件感知，只有低层软件解决不了的错误才通知高层软件解决。许多情况下，错误恢复可以在低层得到解决，而高层软件不需要知道。





4) 缓冲技术

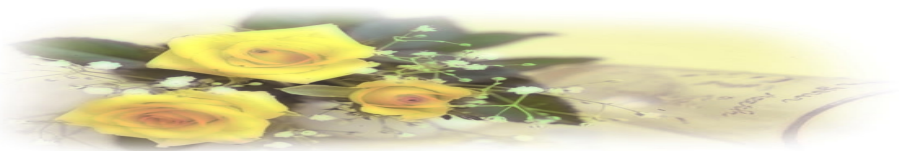
由于CPU与设备之间的速度差异，无论是块设备还是字符设备，都需要使用缓冲技术。对于不同类型的设备，其缓冲区(块)的大小是不一样的，块设备的缓冲是以数据块为单位的，而字符设备的缓冲则以字节为单位。就是同类型的设备，其缓冲区(块)的大小也是存在差异的，如不同的磁盘，其扇区的大小有可能不同。因此，I/O软件应能屏蔽这种差异，向高层软件提供统一大小的数据块或字符单元，使得高层软件能够只与逻辑块大小一致的抽象设备进行交互。





5) 设备的分配和释放

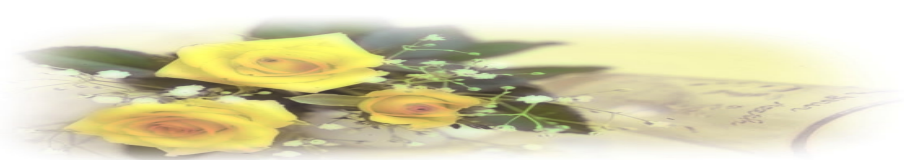
对于系统中的共享设备，如磁盘等，可以同时为多个用户服务。对于这样的设备，应该允许多个进程同时对其提出I/O请求。但对于独占设备，如键盘和打印机等，在某一段时间只能供一个用户使用，对其分配和释放的不当，将引起混乱，甚至死锁。对于独占设备和共享设备带来的许多问题，I/O软件必须能够同时进行妥善的解决。





6) I/O控制方式

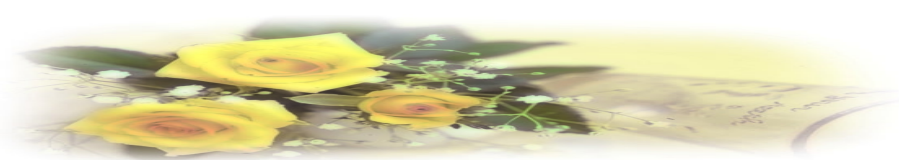
针对具有不同传输速率的设备，综合系统效率和系统代价等因素，合理选择I/O控制方式，如像打印机等低速设备应采用中断驱动方式，而对磁盘等高速设备则采用DMA控制方式等，以提高系统的利用率。为方便用户，I/O软件也应屏蔽这种差异，向高层软件提供统一的操作接口。





第五章 设备管理

综上所述，I/O软件涉及的面非常宽，往下与硬件有着密切的关系，往上又与用户直接交互，它与进程管理、存储器、文件管理等都存在着一一定的联系，即它们都可能需要I/O软件来实现I/O操作。为使十分复杂的I/O软件能具有清晰的结构，更好的可移植性和易适应性，目前在I/O软件中已普遍采用了层次式结构，将系统中的设备操作和管理软件分为若干个层次，每一层都利用其下层提供的服务，完成输入、输出功能中的某些子功能，并屏蔽这些功能实现的细节，向高层提供服务。

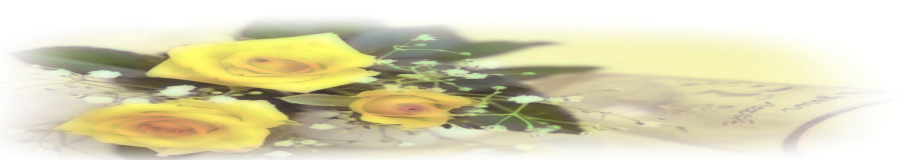




第五章 设备管理

在层次式结构的I/O软件中，只要层次间的接口不变，对每个层次中的软件进行的修改都不会引起其下层或高层代码的变更，仅最低层才会涉及到硬件的具体特性。通常把I/O 软件组织成四个层次，如图5-16所示(图中的箭头表示I/O的控制流)。各层次及其功能如下所述：

(1) 用户层软件：实现与用户交互的接口，用户可直接调用在用户层提供的、与I/O操作有关的库函数，对设备进行操作。



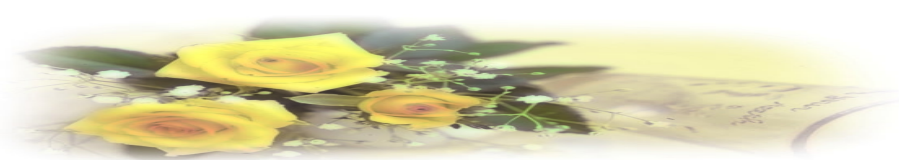


第五章 设备管理

(2) 设备独立性软件：负责实现与设备驱动器的统一接口、设备命名、设备的保护以及设备的分配与释放等，同时为设备管理和数据传送提供必要的存储空间。

(3) 设备驱动程序：与硬件直接相关，负责具体实现系统对设备发出的操作指令，驱动I/O设备工作的驱动程序。

(4) 中断处理程序：用于保存被中断进程的CPU环境，转入相应的中断处理程序进行处理，处理完后再恢复被中断进程的现场后返回到被中断进程。



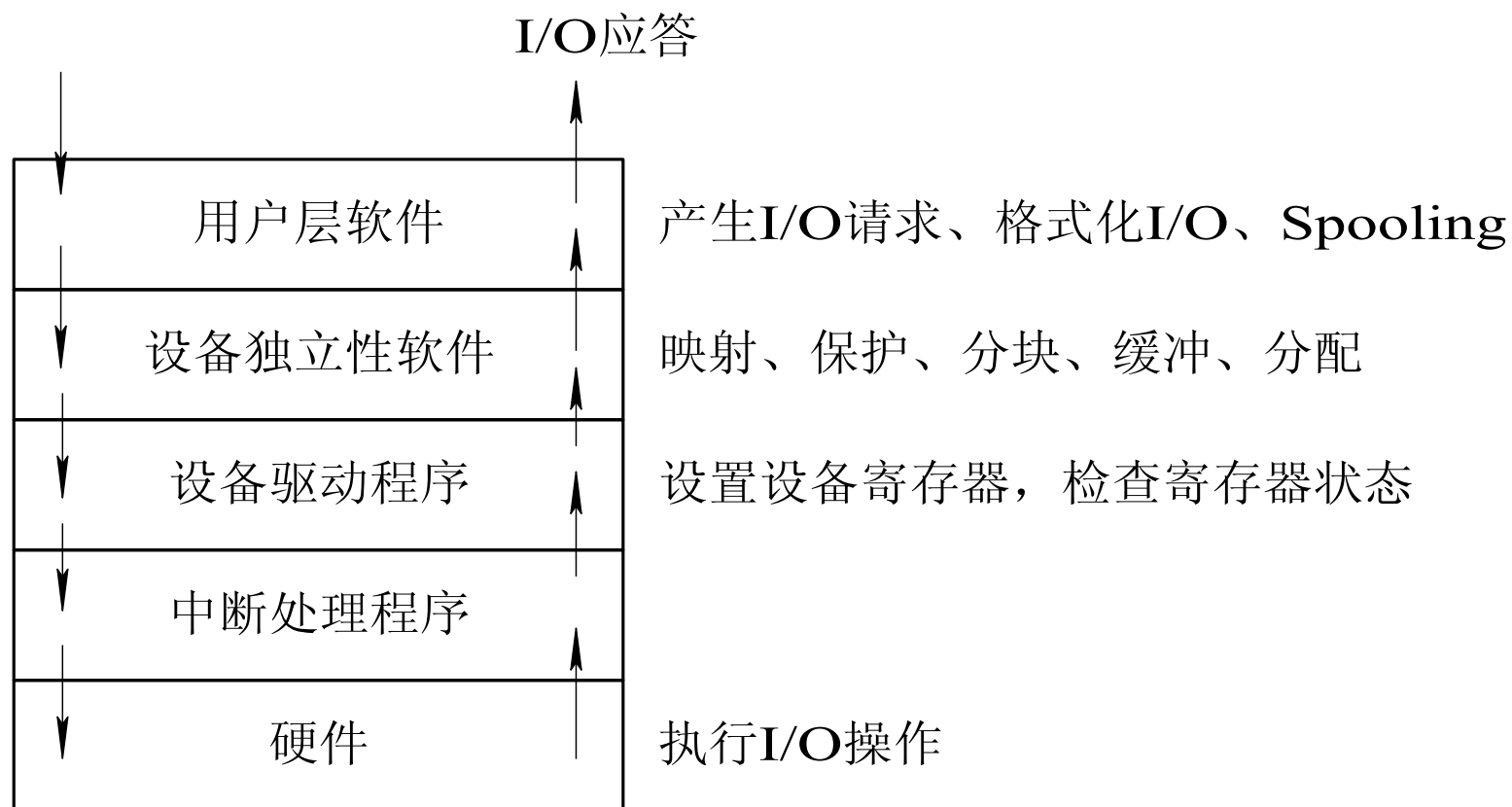
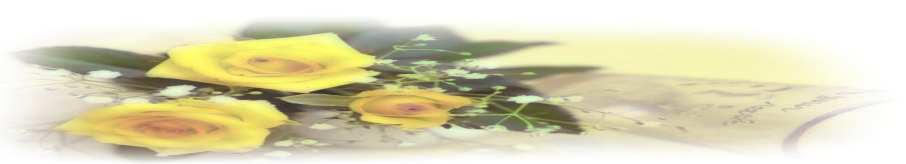


图5-16 I/O系统的层次及功能

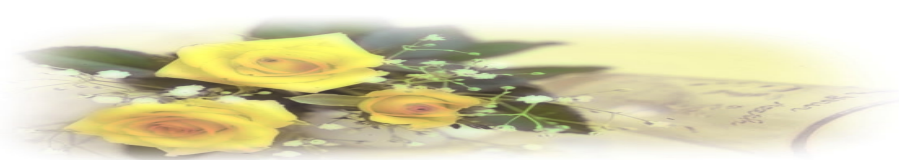




5.4.2 中断处理程序

1. 唤醒被阻塞的驱动(程序)进程

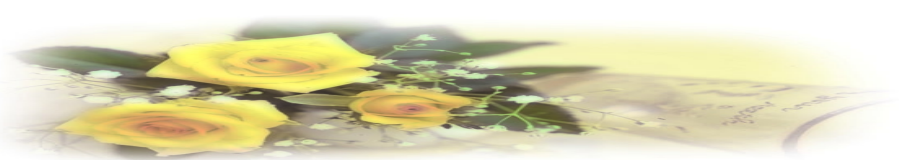
当中断处理程序开始执行时，首先去唤醒处于阻塞状态的驱动(程序)进程。如果是采用了信号量机制，则可通过执行signal操作，将处于阻塞状态的驱动(程序)进程唤醒；在采用信号机制时，将发送一信号给阻塞进程。





2. 保护被中断进程的CPU环境

通常由硬件自动将处理机状态字PSW和程序计数器(PC)中的内容,保存在中断保留区(栈)中,然后把被中断进程的CPU现场信息(即包括所有的CPU寄存器,如通用寄存器、段寄存器等内容)都压入中断栈中,因为在中断处理时可能会用到这些寄存器。图5-17给出了一个简单的保护中断现场的示意图。该程序是指令在N位置时被中断的,程序计数器中的内容为N+1,所有寄存器的内容都被保留在栈中。



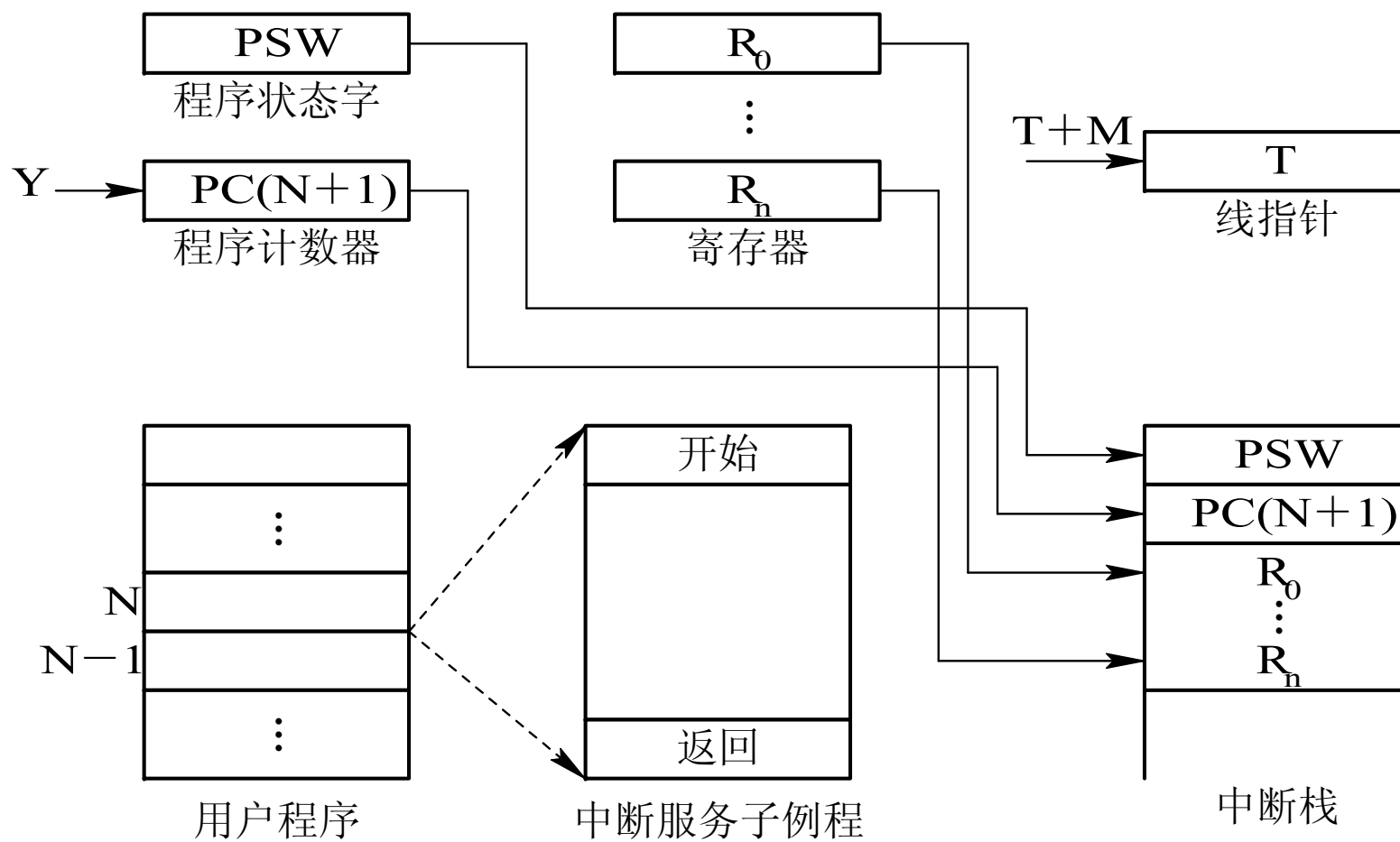
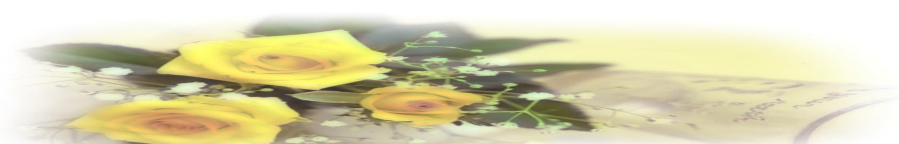


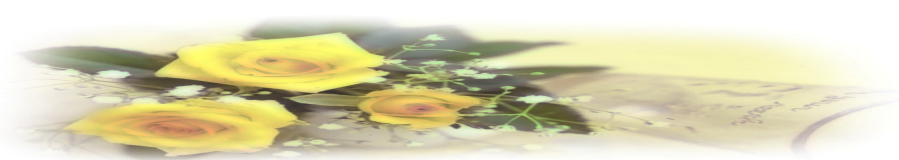
图5-17 中断现场保护示意图





3. 转入相应的设备处理程序

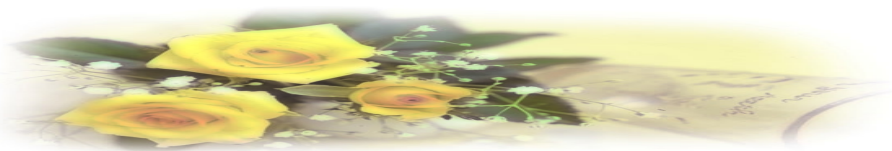
由处理机对各个中断源进行测试，以确定引起本次中断的I/O设备，并发送一应答信号给发出中断请求的进程，使之消除该中断请求信号，然后将相应的设备中断处理程序的入口地址装入到程序计数器中，使处理机转向中断处理程序。





4. 中断处理

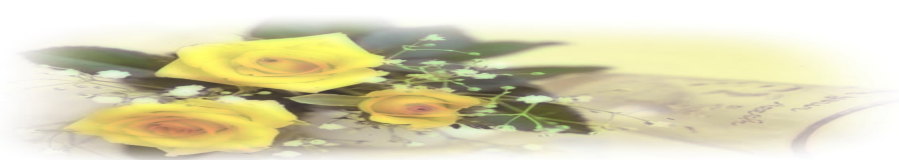
对于不同的设备，有不同的中断处理程序。该程序首先从设备控制器中读出设备状态，以判别本次中断是正常完成中断，还是异常结束中断。若是前者，中断程序便进行结束处理；若还有命令，可再向控制器发送新的命令，进行新一轮的数据传送。若是异常结束中断，则根据发生异常的原因做相应的处理。





5. 恢复被中断进程的现场

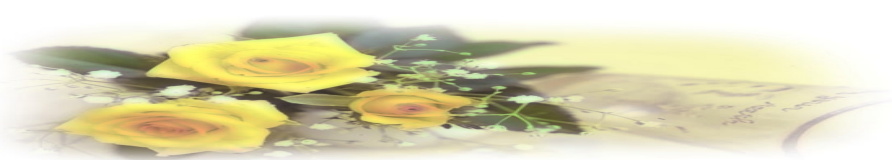
当中断处理完成以后，便可将保存在中断栈中的被中断进程的现场信息取出，并装入到相应的寄存器中，其中包括该程序下一次要执行的指令的地址 $N+1$ 、处理机状态字PSW，以及各通用寄存器和段寄存器的内容。这样，当处理机再执行本程序时，便从 $N+1$ 处开始，最终返回到被中断的程序。





第五章 设备管理

I/O操作完成后，驱动程序必须检查本次I/O操作中是否发生了错误，并向上层软件报告，最终向调用者报告本次I/O的执行情况。除了上述的第4步外，其它各步骤对所有I/O设备都是相同的，因而对于某种操作系统，例如UNIX系统，是把这些共同的部分集中起来，形成中断总控程序。每当要进行中断处理时，都要首先进入中断总控程序。而对于第4步，则对不同设备须采用不同的设备中断处理程序继续执行。图5-18示出了中断处理流程。



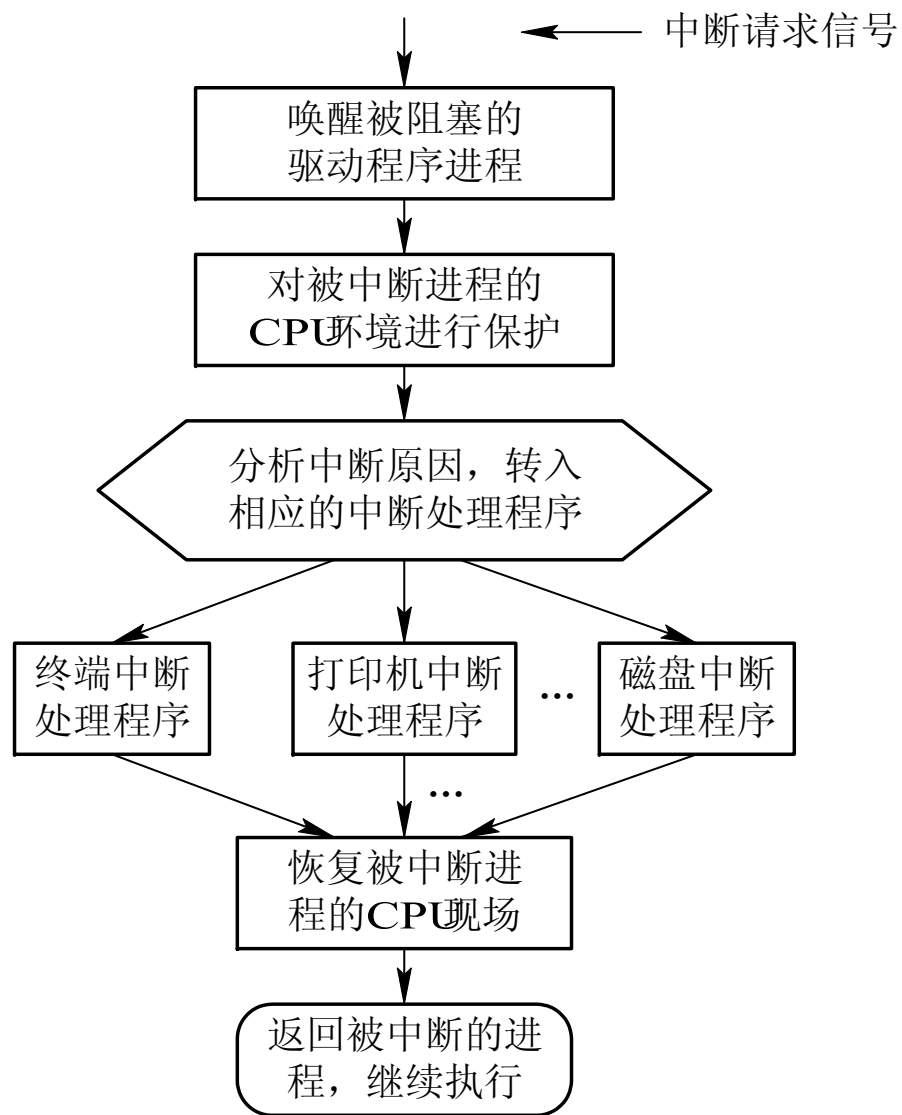
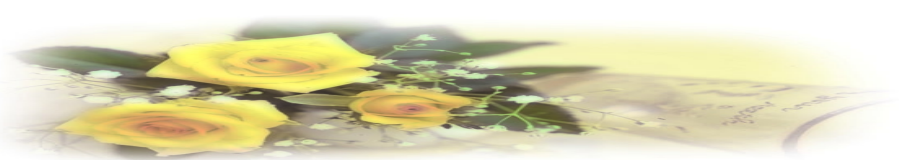


图5-18 中断处理流程



5.4.3 设备驱动程序

设备驱动程序通常又称为设备处理程序，它是I/O进程与设备控制器之间的通信程序，又由于它常以进程的形式存在，故以后就简称之为设备驱动进程。其主要任务是接收上层软件发来的抽象I/O要求，如read或write命令，在把它转换为具体要求后，发送给设备控制器，启动设备去执行；此外，它也将由设备控制器发来的信号传送给上层软件。由于驱动程序与硬件密切相关，故应为每一类设备配置一种驱动程序；有时也可为非常类似的两类设备配置一个驱动程序。例如，打印机和显示器需要不同的驱动程序，但SCSI磁盘驱动程序通常可以处理不同大小和不同速度的多个SCSI磁盘，甚至可以处理SCSI CD-ROM。





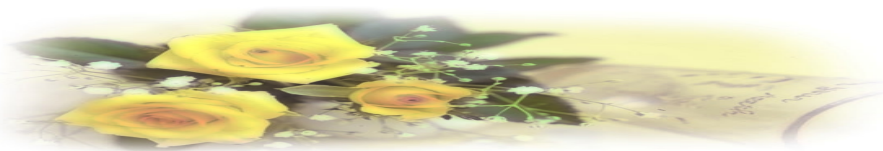
1. 设备驱动程序的功能

为了实现I/O进程与设备控制器之间的通信，设备驱动程序应具有以下功能：

(1) 接收由设备独立性软件发来的命令和参数，并将命令中的抽象要求转换为具体要求，例如，将磁盘块号转换为磁盘的盘面、磁道号及扇区号。

(2) 检查用户I/O请求的合法性，了解I/O设备的状态，传递有关参数，设置设备的工作方式。

(3) 发出I/O命令。如果设备空闲，便立即启动I/O设备去完成指定的I/O操作；如果设备处于忙碌状态，则将请求者的请求块挂在设备队列上等待。

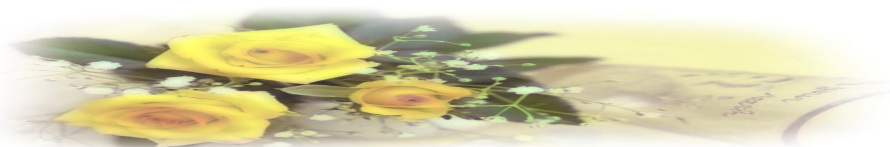




第五章 设备管理

(4) 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。

(5) 对于设置有通道的计算机系统，驱动程序还应能够根据用户的I/O请求，自动地构成通道程序。



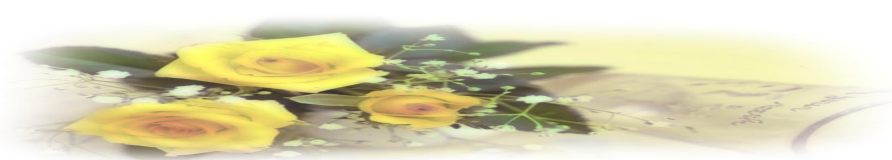


2. 设备处理方式

(1) 为每一类设备设置一个进程，专门用于执行这类设备的I/O操作。比如，为所有的交互式终端设置一个交互式终端进程；又如，为同一类型的打印机设置一个打印进程。

(2) 在整个系统中设置一个I/O进程，专门用于执行系统中所有各类设备的I/O操作。也可以设置一个输入进程和一个输出进程，分别处理系统中所有各类设备的输入或输出操作。

(3) 不设置专门的设备处理进程，而只为各类设备设置相应的设备处理程序(模块)，供用户进程或系统进程调用。

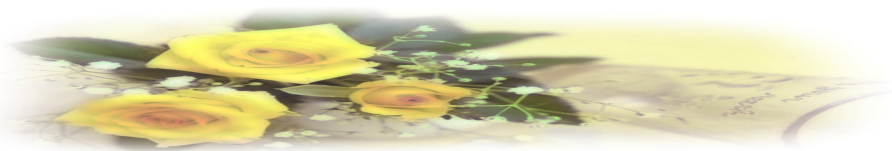




3. 设备驱动程序的特点

设备驱动程序属于低级的系统例程，它与一般的应用程序及系统程序之间有下列明显差异：

(1) 驱动程序主要是指在请求I/O的进程与设备控制器之间的一个通信和转换程序。它将进程的I/O请求经过转换后，传送给控制器；又把控制器中所记录的设备状态和I/O操作完成情况及时地反映给请求I/O的进程。

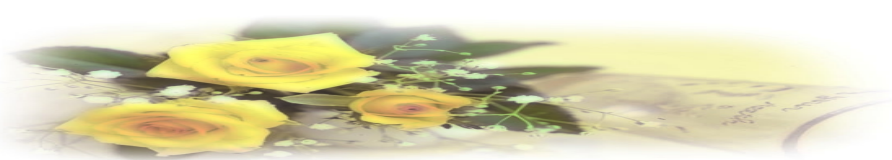




第五章 设备管理

(2) 驱动程序与设备控制器和I/O设备的硬件特性紧密相关，因而对不同类型的设备应配置不同的驱动程序。例如，可以为相同的多个终端设置一个终端驱动程序，但有时即使是同一类型的设备，由于其生产厂家不同，它们也可能并不完全兼容，此时也须为它们配置不同的驱动程序。

(3) 驱动程序与I/O设备所采用的I/O控制方式紧密相关。常用的I/O控制方式是中断驱动和DMA方式，这两种方式的驱动程序明显不同，因为后者应按数组方式启动设备及进行中断处理。



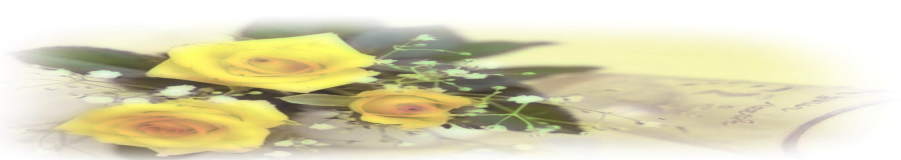


第五章 设备管理

(4) 由于驱动程序与硬件紧密相关，因而其中的一部分必须用汇编语言书写。目前有很多驱动程序的基本部分，已经固化在ROM中。

(5) 驱动程序应允许可重入。一个正在运行的驱动程序常会在一次调用完成前被再次调用。例如，网络驱动程序正在处理一个到来的数据包时，另一个数据包可能到达。

(6) 驱动程序不允许系统调用。但是为了满足其与内核其它部分的交互，可以允许对某些内核过程的调用，如通过调用内核过程来分配和释放内存页面作为缓冲区，以及调用其它过程来管理MMU定时器、DMA控制器、中断控制器等。

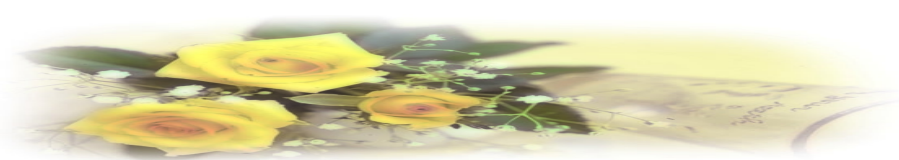




4. 设备驱动程序的处理过程

不同类型的设备应有不同的设备驱动程序，但大体上它们都可以分成两部分，其中，除了要有能够驱动I/O设备工作的驱动程序外，还需要有设备中断处理程序，以处理I/O完成后的工作。

设备驱动程序的主要任务是启动指定设备。但在启动之前，还必须完成必要的准备工作，如检测设备状态是否为“忙”等。在完成所有的准备工作后，才最后向设备控制器发送一条启动命令。



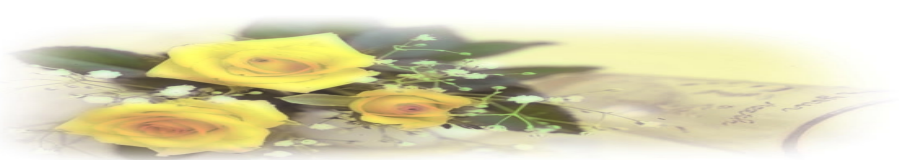


第五章 设备管理

以下是设备驱动程序的处理过程。

1) 将抽象要求转换为具体要求

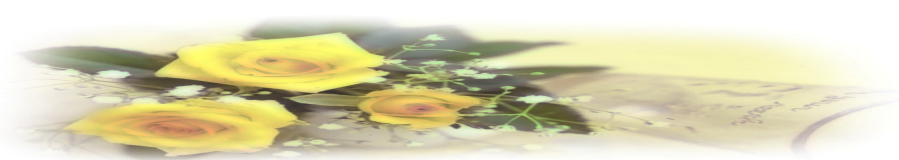
通常在每个设备控制器中都含有若干个寄存器，分别用于暂存命令、数据和参数等。由于用户及上层软件对设备控制器的具体情况毫无了解，因而只能向它发出抽象的要求(命令)，但这些命令无法传送给设备控制器。因此，就需要将这些抽象要求转换为具体要求。例如，将抽象要求中的盘块号转换为磁盘的盘面、磁道号及扇区。这一转换工作只能由驱动程序来完成，因为在OS中只有驱动程序才同时了解抽象要求和设备控制器中的寄存器情况；也只有它才知道命令、数据和参数应分别送往哪个寄存器。





2) 检查I/O请求的合法性

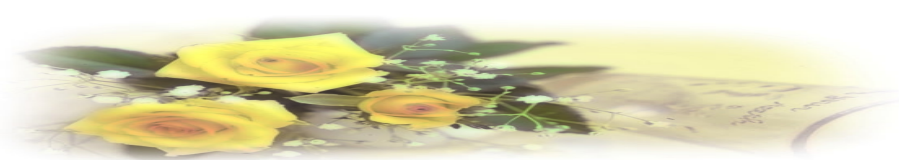
对于任何输入设备，都是只能完成一组特定的功能，若该设备不支持这次的I/O请求，则认为这次I/O请求非法。例如，用户试图请求从打印机输入数据，显然系统应予以拒绝。此外，还有些设备如磁盘和终端，它们虽然都是既可读又可写的，但若在打开这些设备时规定的是读，则用户的写请求必然被拒绝。





3) 读出和检查设备的状态

在启动某个设备进行I/O操作时，其前提条件应是该设备正处于空闲状态。因此在启动设备之前，要从设备控制器的状态寄存器中，读出设备的状态。例如，为了向某设备写入数据，此前应先检查该设备是否处于接收就绪状态，仅当它处于接收就绪状态时，才能启动其设备控制器，否则只能等待。



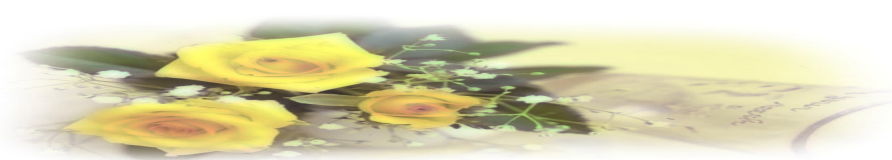


4) 传送必要的参数

对于许多设备，特别是块设备，除必须向其控制器发出启动命令外，还需传送必要的参数。例如在启动磁盘进行读/写之前，应先将本次要传送的字节数和数据应到达的主存始址，送入控制器的相应寄存器中。

5) 工作方式的设置

有些设备可具有多种工作方式，典型情况是利用RS-232接口进行异步通信。在启动该接口之前，应先按通信规程设定参数：波特率、奇偶校验方式、停止位数目及数据字节长度等。

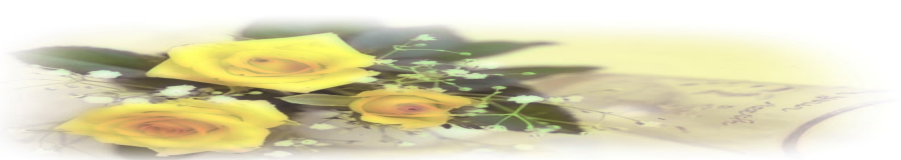




6) 启动I/O设备

在完成上述各项准备工作之后，驱动程序可以向控制器中的命令寄存器传送相应的控制命令。对于字符设备，若发出的是写命令，驱动程序将把一个数据传送给控制器；若发出的是读命令，则驱动程序等待接收数据，并通过从控制器中的状态寄存器读入状态字的方法，来确定数据是否到达。

驱动程序发出I/O命令后，基本的I/O操作是在设备控制器的控制下进行的。通常，I/O操作所要完成的工作较多，需要一定的时间，如读/写一个盘块中的数据，此时驱动(程序)进程把自己阻塞起来，直到中断到来时才将它唤醒。

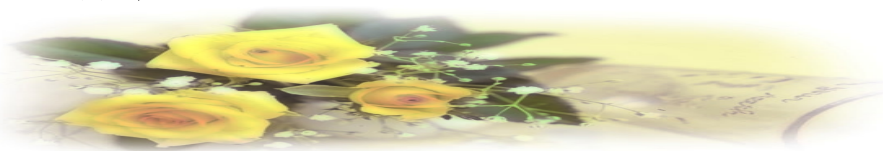




5.4.4 设备独立性软件

1. 设备独立性的概念

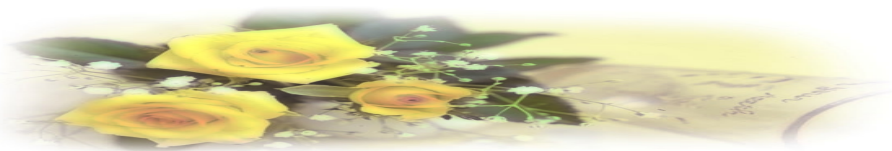
为了提高OS的可适应性和可扩展性，在现代OS中都毫无例外地实现了设备独立性(Device Independence)，也称为设备无关性。其基本含义是：应用程序独立于具体使用的物理设备。为了实现设备独立性而引入了逻辑设备和物理设备这两个概念。在应用程序中，使用逻辑设备名称来请求使用某类设备；而系统在实际执行时，还必须使用物理设备名称。因此，系统须具有将逻辑设备名称转换为某物理设备名称的功能，这非常类似于存储器管理中所介绍的逻辑地址和物理地址的概念。在应用程序中所使用的是逻辑地址，而系统在分配和使用内存时，必须使用物理地址。在实现了设备独立性的功能后，可带来以下两方面的好处。





1) 设备分配时的灵活性

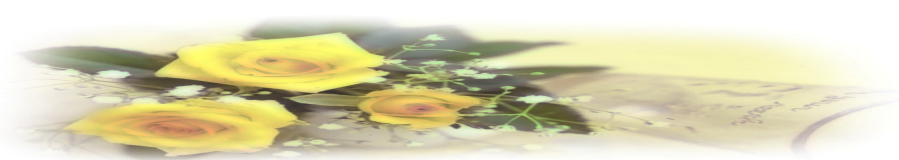
当应用程序(进程)以物理设备名称来请求使用指定的某台设备时，如果该设备已经分配给其他进程或正在检修，而此时尽管还有几台其它的相同设备正在空闲，该进程却仍阻塞。但若进程能以逻辑设备名称来请求某类设备时，系统可立即将该类设备中的任一分配给进程，仅当所有此类设备已全部分配完毕时，进程才会阻塞。





2) 易于实现I/O重定向

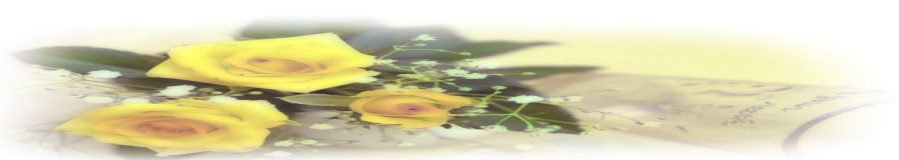
所谓I/O重定向，是指用于I/O操作的设备可以更换(即重定向)，而不必改变应用程序。例如，我们在调试一个应用程序时，可将程序的所有输出送往屏幕显示；而在程序调试完后，如需正式将程序的运行结果打印出来，此时便须将I/O重定向的数据结构——逻辑设备表中的显示终端改为打印机，而不必修改应用程序。I/O重定向功能具有很大的实用价值，现已被广泛地引入到各类OS中。





2. 设备独立性软件

驱动程序是一个与硬件(或设备)紧密相关的软件。为了实现设备独立性，必须再在驱动程序之上设置一层软件，称为设备独立性软件。至于设备独立性软件和设备驱动程序之间的界限，根据不同的操作系统和设备有所差异，主要取决于操作系统、设备独立性和设备驱动程序的运行效率等多方面因素的权衡，因为对于一些本应由设备独立性软件实现的功能，可能由于效率等诸多因素，实际上设计在设备驱动程序中。总的来说，设备独立性软件的主要功能可分为以下两个方面：





第五章 设备管理

(1) 执行所有设备的公有操作。这些公有操作包括：

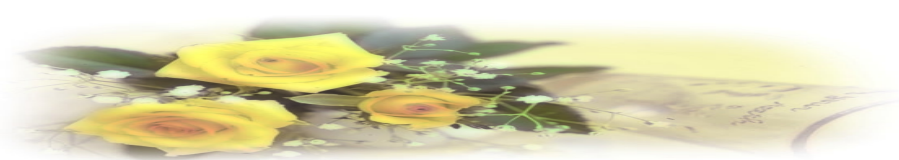
① 对独立设备的分配与回收；

② 将逻辑设备名映射为物理设备名，进一步可以找到相应物理设备的驱动程序；

③ 对设备进行保护，禁止用户直接访问设备；

④ 缓冲管理，即对字符设备和块设备的缓冲区进行有效的管理，以提高I/O的效率；

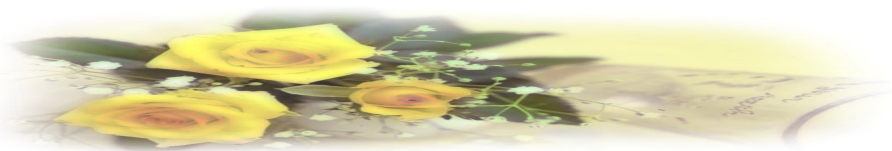
⑤ 差错控制，由于在I/O操作中的绝大多数错误都与设备无关，故主要由设备驱动程序处理，而设备独立性软件只处理那些设备驱动程序无法处理的错误；





第五章 设备管理

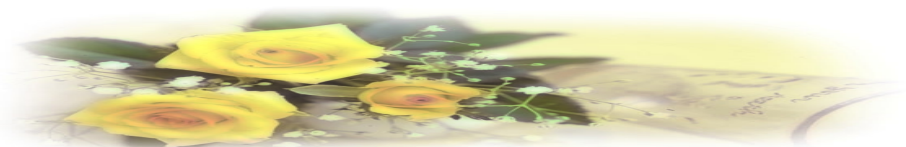
⑥ 提供独立于设备的逻辑块，不同类型的设备信息交换单位是不同的，读取和传输速率也各不相同，如字符型设备以单个字符为单位，块设备是以一个数据块为单位，即使同一类型的设备，其信息交换单位大小也是有差异的，如不同磁盘由于扇区大小的不同，可能造成数据块大小的不一致，因此设备独立性软件应负责隐藏这些差异，对逻辑设备使用并向高层软件提供大小统一的逻辑数据块。





第五章 设备管理

(2) 向用户层(或文件层)软件提供统一接口。无论何种设备，它们向用户所提供的接口应该是相同的。例如，对各种设备的读操作，在应用程序中都使用`read`；而对各种设备的写操作，也都使用`write`。

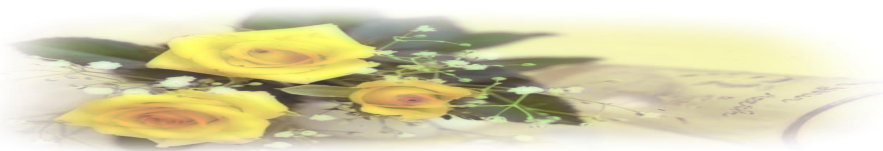




3. 逻辑设备名到物理设备名映射的实现

1) 逻辑设备表

为了实现设备的独立性，系统必须设置一张逻辑设备表 (LUT, Logical Unit Table)，用于将应用程序中所使用的逻辑设备名映射为物理设备名。在该表的每个表目中包含了三项：逻辑设备名、物理设备名和设备驱动程序的入口地址，如图 5-19(a)所示。当进程用逻辑设备名请求分配I/O设备时，系统为它分配相应的物理设备，并在LUT上建立一个表目，填上应用程序中使用的逻辑设备名和系统分配的物理设备名，以及该设备驱动程序的入口地址。当以后进程再利用该逻辑设备名请求I/O操作时，系统通过查找LUT，便可找到物理设备和驱动程序。



第五章 设备管理



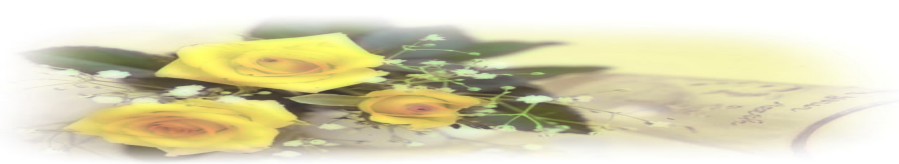
逻辑设备名	物理设备名	驱动程序入口地址
/dev/tty	3	1024
/dev/printe	5	2046
⋮	⋮	⋮

(a)

逻辑设备名	系统设备表指针
/dev/tty	3
/dev/printe	5
⋮	

(b)

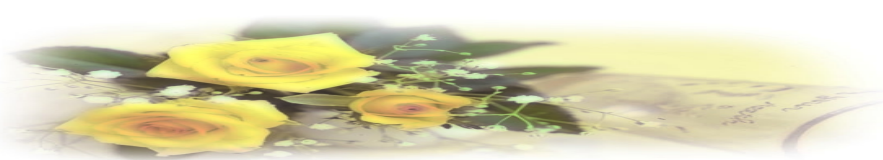
图5-19 逻辑设备表





2) LUT的设置问题

LUT的设置可采取两种方式：第一种方式是在整个系统中只设置一张LUT。由于系统中所有进程的设备分配情况都记录在同一张LUT中，因而不允许在LUT中具有相同的逻辑设备名，这就要求所有用户都不使用相同的逻辑设备名。在多用户环境下这通常是难以做到的，因而这种方式主要用于单用户系统中。第二种方式是为每个用户设置一张LUT。每当用户登录时，便为该用户建立一个进程，同时也为之建立一张LUT，并将该表放入进程的PCB中。由于通常在多用户系统中，都配置了系统设备表，故此时的逻辑设备表可以采用图5-19(b)中的格式。

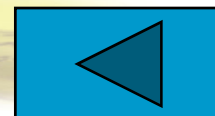
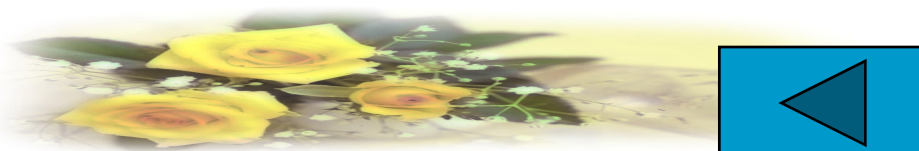




5.4.5 用户层的I/O软件

一般而言，大部分的I/O软件都在操作系统内部，但仍有一小部分在用户层，包括与用户程序链接在一起的库函数，以及完全运行于内核之外的一些程序。

用户层软件必须通过一组系统调用来取得操作系统服务。在现代的高级语言以及C语言中，通常提供了与各系统调用一一对应的库函数，用户程序通过调用对应的库函数使用系统调用。这些库函数与调用程序连接在一起，包含在运行时装入在内存的二进制程序中，如C语言中的库函数write等，显然这些库函数的集合也是I/O系统的组成部分。但在许多现代操作系统中，系统调用本身已经采用C语言编写，并以函数形式提供，所以在使用C语言编写的用户程序中，可以直接使用这些系统调用。



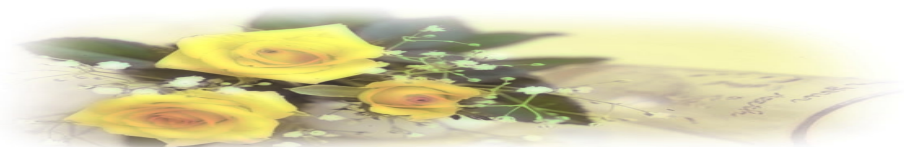


5.5 设备分配

5.5.1 设备分配中的数据结构

1. 设备控制表(DCT)

系统为每一个设备都配置了一张设备控制表，用于记录本设备的情况，如图5-20所示。



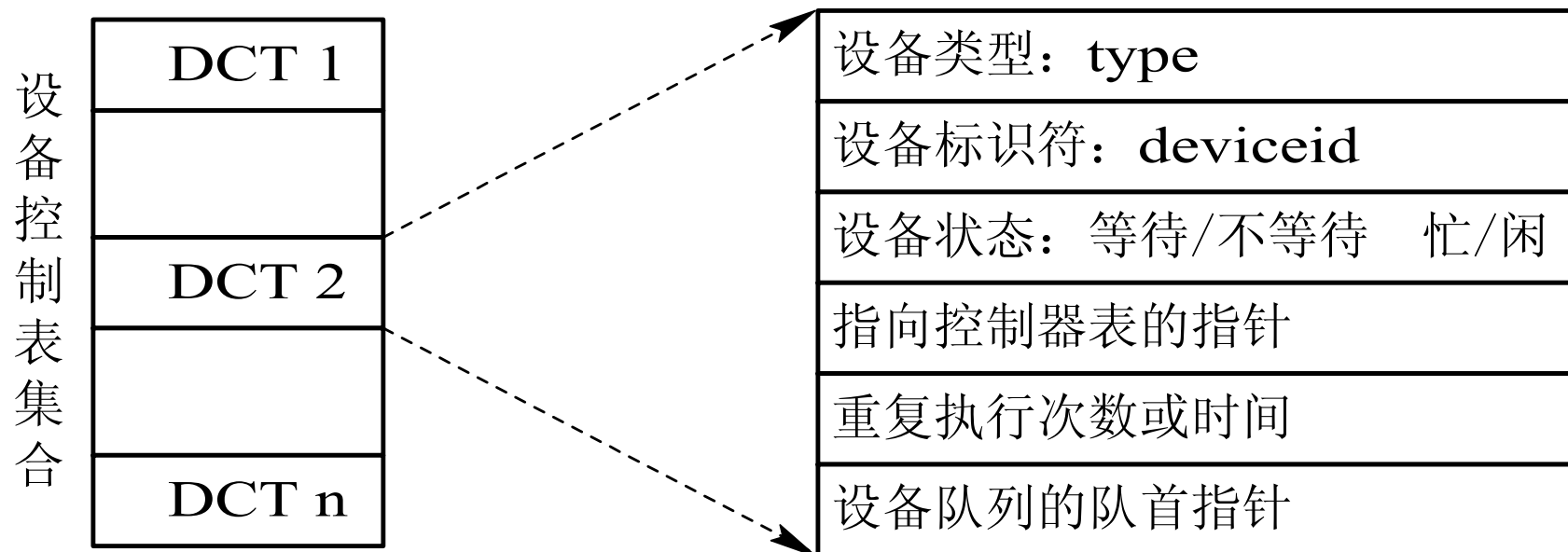
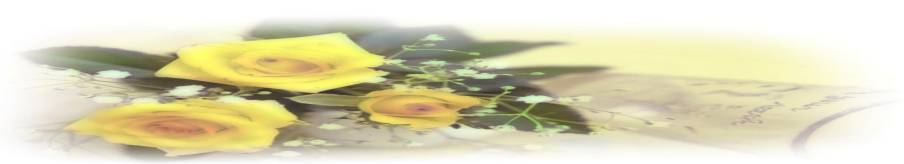


图5-20 设备控制表



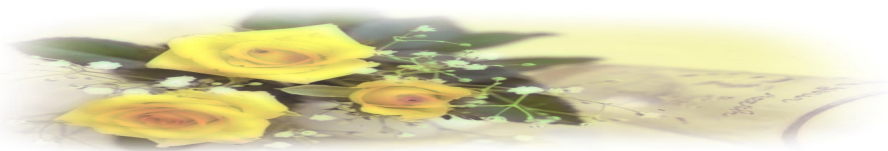


第五章 设备管理

设备控制表中，除了有用于指示设备类型的字段type和设备标识字段deviceid外，还应含有下列字段：

(1) 设备队列队首指针。凡因请求本设备而未得到满足的进程，其PCB都应按照一定的策略排成一个队列，称该队列为设备请求队列或简称设备队列。其队首指针指向队首PCB。在有的系统中还设置了队尾指针。

(2) 设备状态。当设备自身正处于使用状态时，应将设备的忙/闲标志置“1”。若与该设备相连接的控制器或通道正忙，也不能启动该设备，此时则应将设备的等待标志置“1”。

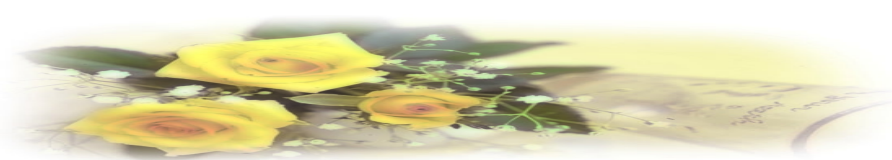




第五章 设备管理

(3) 与设备连接的控制器表指针。该指针指向该设备所连接的控制器表。在设备到主机之间具有多条通路的情况下，一个设备将与多个控制器相连接。此时，在DCT中还应设置多个控制器表指针。

(4) 重复执行次数。由于外部设备在传送数据时，较易发生数据传送错误，因而在许多系统中，如果发生传送错误，并不立即认为传送失败，而是令它重新传送，并由系统规定设备在工作中发生错误时应重复执行的次数。在重复执行时，若能恢复正常传送，则仍认为传送成功。仅当屡次失败，致使重复执行次数达到规定值而传送仍不成功时，才认为传送失败。



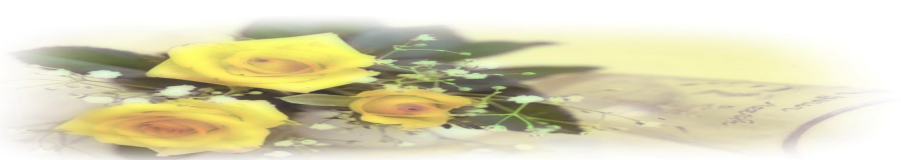


2. 控制器控制表、通道控制表和系统设备表

(1) 控制器控制表(COCT)。系统为每一个控制器都设置了一张用于记录本控制器情况的控制器控制表，如图5-21(a)所示。

(2) 通道控制表(CHCT)。每个通道都配有一张通道控制表，如图5-21(b)所示。

(3) 系统设备表(SDT)。这是系统范围的数据结构，其中记录了系统中全部设备的情况。每个设备占一个表目，其中包括有设备类型、设备标识符、设备控制表及设备驱动程序的入口等项，如图5-21(c)所示。



第五章 设备管理

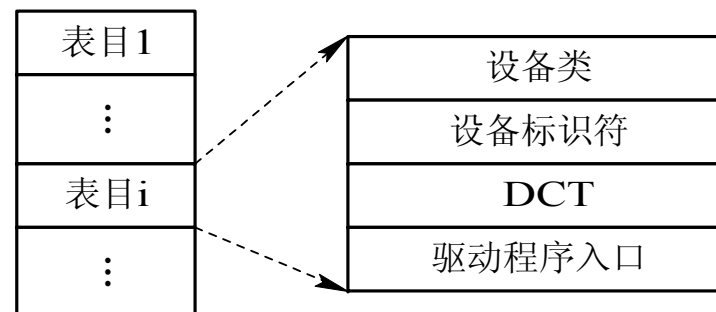


控制器标识符: controllerid
控制器状态: 忙/闲
与控制器连接的通道表指针
控制器队列的队首指针
控制器队列的队尾指针

(a) 控制器表COCT

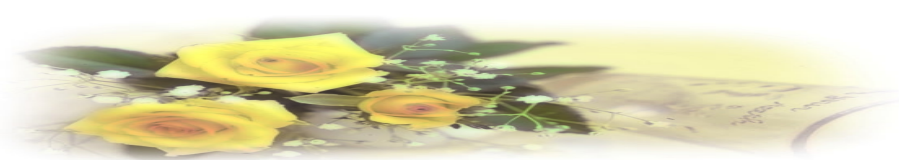
通道标识符: channelid
通道状态: 忙/闲
与通道连接的控制器表首址
通道队列的队首指针
通道队列的队尾指针

(b) 通道表CHCT



(c) 系统设备表SDT

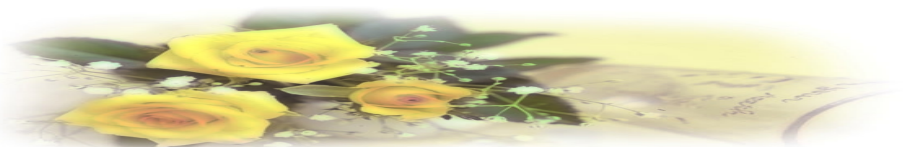
图5-21 COCT、CHCT和SDT





5.5.2 设备分配时应考虑的因素

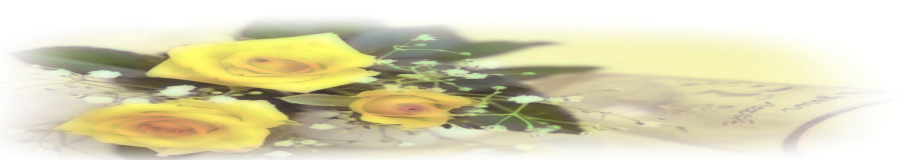
为了使系统有条不紊地工作，系统在分配设备时，应考虑这样几个因素：① 设备的固有属性；② 设备分配算法；③ 设备分配时的安全性；④ 设备独立性。本小节介绍前三个问题，下一小节专门介绍设备独立性问题。





1. 设备的固有属性

在分配设备时，首先应考虑与设备分配有关的设备属性。设备的固有属性可分成三种：第一种是独占性，是指这种设备在一段时间内只允许一个进程独占，此即第二章所说的“临界资源”；第二种是共享性，指这种设备允许多个进程同时共享；第三种是可虚拟设备，指设备本身虽是独占设备，但经过某种技术处理，可以把它改造成虚拟设备。对上述的独占、共享、可虚拟三种设备应采取不同的分配策略。





第五章 设备管理

(1) 独占设备。对于独占设备，应采用独享分配策略，即将一个设备分配给某进程后，便由该进程独占，直至该进程完成或释放该设备，然后，系统才能再将该设备分配给其他进程使用。这种分配策略的缺点是，设备得不到充分利用，而且还可能引起死锁。

(2) 共享设备。对于共享设备，可同时分配给多个进程使用，此时须注意对这些进程访问该设备的先后次序进行合理的调度。

(3) 可虚拟设备。由于可虚拟设备是指一台物理设备在采用虚拟技术后，可变成多台逻辑上的所谓虚拟设备，因而说，一台可虚拟设备是可共享的设备，可以将它同时分配给多个进程使用，并对这些访问该(物理)设备的先后次序进行控制。

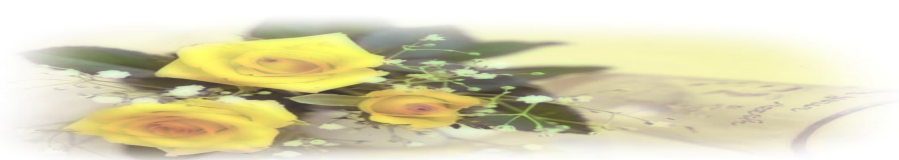




2. 设备分配算法

对设备进行分配的算法，与进程调度的算法有些相似之处，但前者相对简单，通常只采用以下两种分配算法：

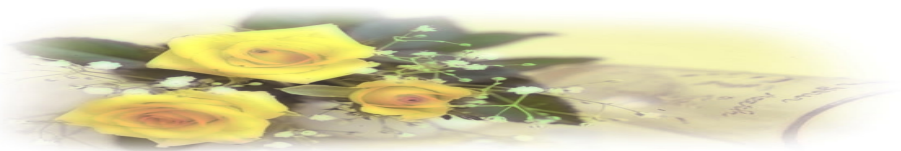
(1) 先来先服务。当有多个进程对同一设备提出I/O请求时，该算法是根据诸进程对某设备提出请求的先后次序，将这些进程排成一个设备请求队列，设备分配程序总是把设备首先分配给队首进程。





第五章 设备管理

(2) 优先级高者优先。在进程调度中的这种策略，是优先权高的进程优先获得处理机。如果对这种高优先权进程所提出的I/O请求也赋予高优先权，显然有助于这种进程尽快完成。在利用该算法形成设备队列时，将优先权高的进程排在设备队列前面，而对于优先级相同的I/O请求，则按先来先服务原则排队。



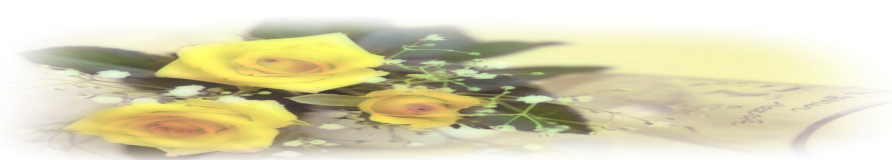


3. 设备分配中的安全性

从进程运行的安全性考虑，设备分配有以下两种方式。

1) 安全分配方式

在这种分配方式中，每当进程发出I/O请求后，便进入阻塞状态，直到其I/O操作完成时才被唤醒。在采用这种分配策略时，一旦进程已经获得某种设备(资源)后便阻塞，使该进程不可能再请求任何资源，而在它运行时又不保持任何资源。因此，这种分配方式已经摒弃了造成死锁的四个必要条件之一的“请求和保持”条件，从而使设备分配是安全的。其缺点是进程进展缓慢，即CPU与I/O设备是串行工作的。

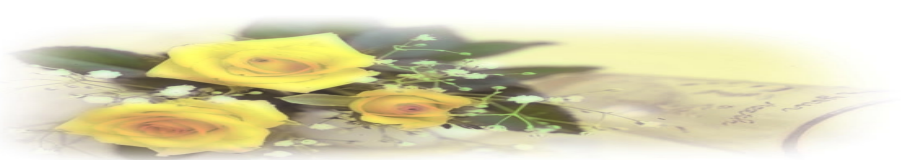




2) 不安全分配方式

在这种分配方式中，进程在发出I/O请求后仍继续运行，需要时又发出第二个I/O请求、第三个I/O请求等。仅当进程所请求的设备已被另一进程占用时，请求进程才进入阻塞状态。这种分配方式的优点是，一个进程可同时操作多个设备，使进程推进迅速。其缺点是分配不安全，因为它可能具备

“请求和保持”条件，从而可能造成死锁。因此，在设备分配程序中，还应再增加一个功能，以用于对本次的设备分配是否会发生死锁进行安全性计算，仅当计算结果说明分配是安全的情况下才进行设备分配。



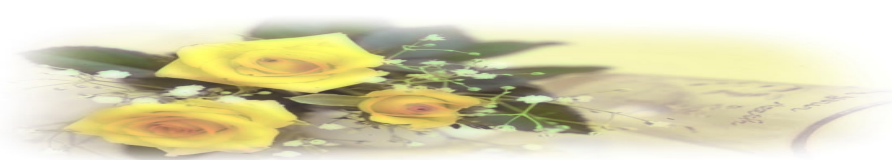


5.5.3 独占设备的分配程序

1. 基本的设备分配程序

1) 分配设备

首先根据I/O请求中的物理设备名，查找系统设备表(SDT)，从中找出该设备的DCT，再根据DCT中的设备状态字段，可知该设备是否正忙。若忙，便将请求I/O进程的PCB挂在设备队列上；否则，便按照一定的算法来计算本次设备分配的安全性。如果不会导致系统进入不安全状态，便将设备分配给请求进程；否则，仍将其PCB插入设备等待队列。





第五章 设备管理

2) 分配控制器

在系统把设备分配给请求I/O的进程后，再到其DCT中找出与该设备连接的控制器的COCT，从COCT的状态字段中可知该控制器是否忙碌。若忙，便将请求I/O进程的PCB挂在该控制器的等待队列上；否则，便将该控制器分配给进程。

3) 分配通道

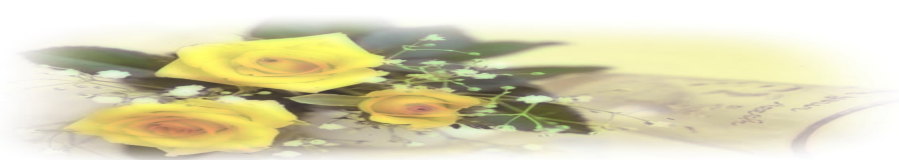
在该COCT中又可找到与该控制器连接的通道的CHCT，再根据CHCT内的状态信息，可知该通道是否忙碌。若忙，便将请求I/O的进程挂在该通道的等待队列上；否则，将该通道分配给进程。只有在设备、控制器和通道三者都分配成功时，这次的设备分配才算成功。然后，便可启动该I/O设备进行数据传送。





2. 设备分配程序的改进

仔细研究上述基本的设备分配程序后可以发现: ① 进程是以物理设备名来提出I/O请求的; ② 采用的是单通路的I/O系统结构, 容易产生“瓶颈”现象。为此, 应从以下两方面对基本的设备分配程序加以改进, 以使独占设备的分配程序具有更强的灵活性, 并提高分配的成功率。

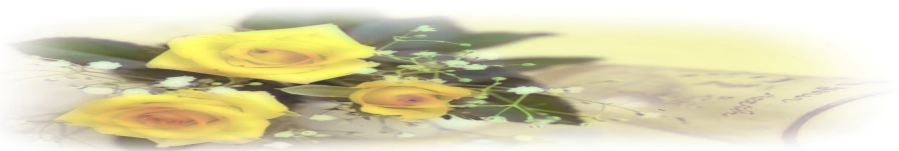




第五章 设备管理

1) 增加设备的独立性

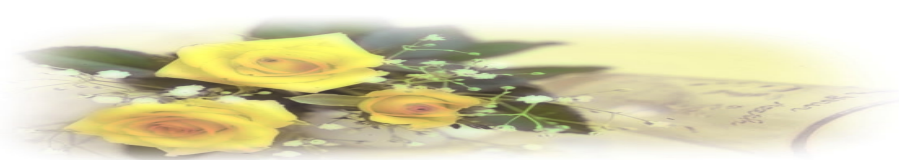
为了获得设备的独立性，进程应使用逻辑设备名请求I/O。这样，系统首先从SDT中找出第一个该类设备的DCT。若该设备忙，又查找第二个该类设备的DCT，仅当所有该类设备都忙时，才把进程挂在该类设备的等待队列上；而只要有一个该类设备可用，系统便进一步计算分配该设备的安全性。





2) 考虑多通路情况

为了防止在I/O系统中出现“瓶颈”现象，通常都采用多通路的I/O系统结构。此时对控制器和通道的分配同样要经过几次反复，即若设备(控制器)所连接的第一个控制器(通道)忙时，应查看其所连接的第二个控制器(通道)，仅当所有的控制器(通道)都忙时，此次的控制器(通道)分配才算失败，才把进程挂在控制器(通道)的等待队列上。而只要有一个控制器(通道)可用，系统便可将它分配给进程。

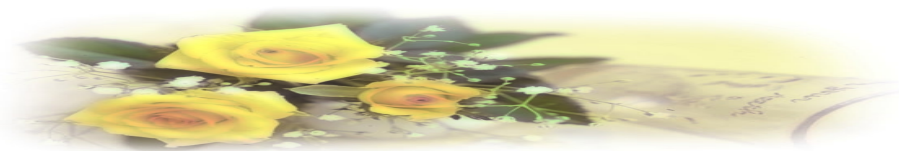




5.5.4 SP00Ling技术

1. 什么是SP00Ling

为了缓和CPU的高速性与I/O设备低速性间的矛盾而引入了脱机输入、脱机输出技术。该技术是利用专门的外围控制机，将低速I/O设备上的数据传送到高速磁盘上；或者相反。事实上，当系统中引入了多道程序技术后，完全可以利用其中的一道程序，来模拟脱机输入时的外围控制机功能，把低速I/O设备上的数据传送到高速磁盘上；再用另一道程序来模拟脱机输出时外围控制机的功能，把数据从磁盘传送到低速输出设备上。这样，便可在主机的直接控制下，实现脱机输入、输出功能。此时的外围操作与CPU对数据的处理同时进行，我们把这种在联机情况下实现的同时外围操作称为SPOOLing(Simultaneous Peripheral Operating On Line)，或称为假脱机操作。



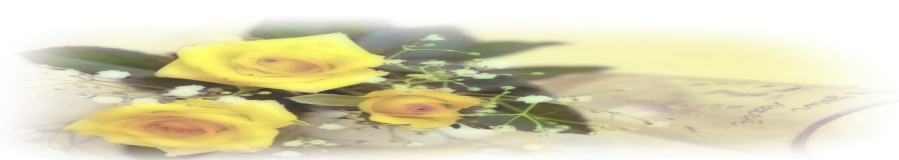


2. SP00Ling系统的组成

由上所述得知，SP00Ling技术是对脱机输入、输出系统的模拟。相应地，SP00Ling系统必须建立在具有多道程序功能的操作系统上，而且还应有高速随机外存的支持，这通常是采用磁盘存储技术。

SP00Ling系统主要有以下三部分：

(1) 输入井和输出井。这是在磁盘上开辟的两个大存储空间。输入井是模拟脱机输入时的磁盘设备，用于暂存I/O设备输入的数据；输出井是模拟脱机输出时的磁盘，用于暂存用户程序的输出数据。

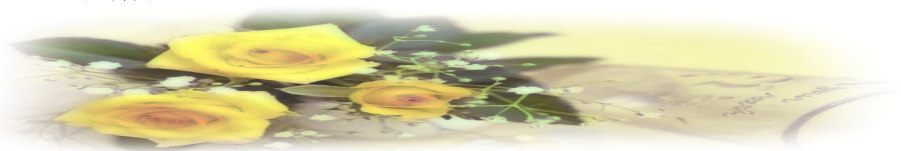




第五章 设备管理

(2) 输入缓冲区和输出缓冲区。为了缓和CPU和磁盘之间速度不匹配的矛盾，在内存中要开辟两个缓冲区：输入缓冲区和输出缓冲区。输入缓冲区用于暂存由输入设备送来的数据，以后再传送到输入井。输出缓冲区用于暂存从输出井送来的数据，以后再传送给输出设备。

(3) 输入进程SPi和输出进程SPo。这里利用两个进程来模拟脱机I/O时的外围控制机。其中，进程SPi模拟脱机输入时的外围控制机，将用户要求的数据从输入机通过输入缓冲区再送到输入井，当CPU需要输入数据时，直接从输入井读入内存；进程SPo模拟脱机输出时的外围控制机，把用户要求输出的数据先从内存送到输出井，待输出设备空闲时，再将输出井中的数据经过输出缓冲区送到输出设备上。



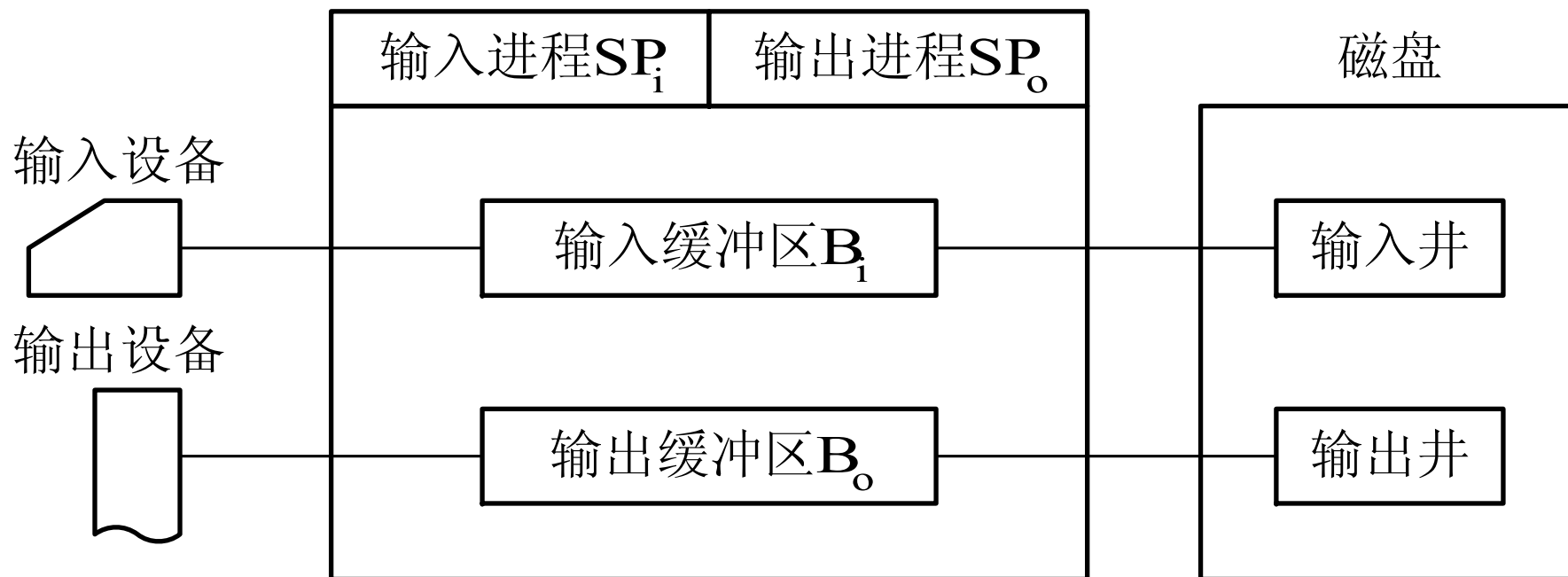
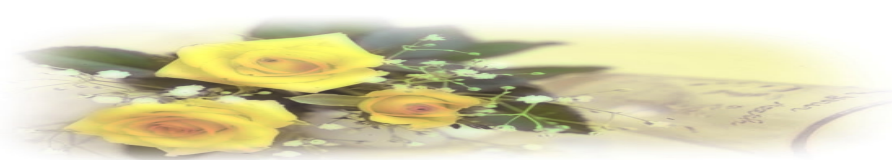


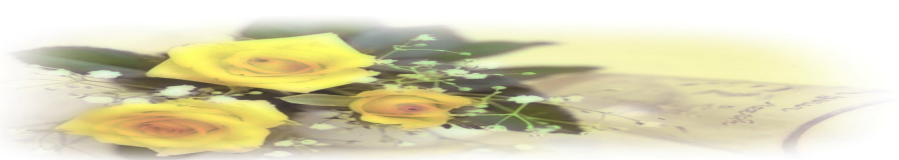
图5-22 SPOOLing系统的组成





3. 共享打印机

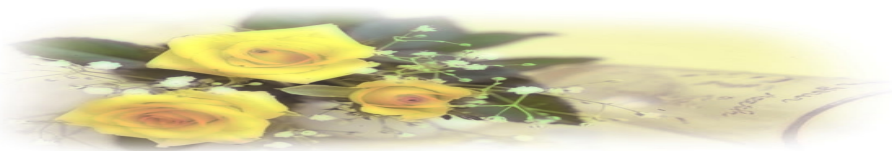
打印机是经常要用到的输出设备，属于独占设备。利用 SPOOLing 技术，可将之改造为一台可供多个用户共享的设备，从而提高设备的利用率，也方便了用户。共享打印机技术已被广泛地用于多用户系统和局域网络中。当用户进程请求打印输出时，SPOOLing 系统同意为它打印输出，但并不真正立即把打印机分配给该用户进程，而只为它做两件事：① 由输出进程在输出井中为之申请一个空闲磁盘块区，并将要打印的数据送入其中；② 输出进程再为用户进程申请一张空白的用户请求打印表，并将用户的打印要求填入其中，再将该表挂到请求打印队列上。如果还有进程要求打印输出，系统仍可接受该请求，也同样为该进程做上述两件事。





第五章 设备管理

如果打印机空闲，输出进程将从请求打印队列的队首取出一张请求打印表，根据表中的要求将要打印的数据，从输出并传送到内存缓冲区，再由打印机进行打印。打印完后，输出进程再查看请求打印队列中是否还有等待打印的请求表。若有，又取出队列中的第一张表，并根据其中的要求进行打印，如此下去，直至请求打印队列为空，输出进程才将自己阻塞起来。仅当下次再有打印请求时，输出进程才被唤醒。

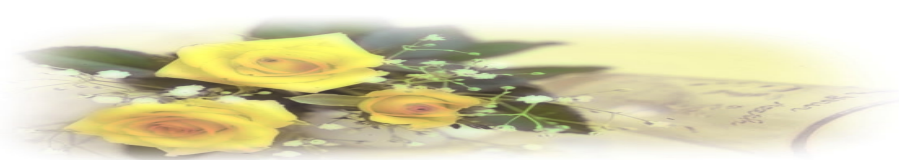




4. SP00Ling系统的特点

SP00Ling系统具有如下主要特点：

(1) 提高了I/O的速度。这里，对数据所进行的I/O操作，已从对低速I/O设备进行的I/O操作，演变为对输入井或输出井中数据的存取，如同脱机输入输出一样，提高了I/O速度，缓和了CPU与低速I/O设备之间速度不匹配的矛盾。

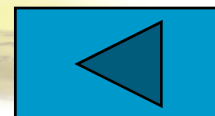
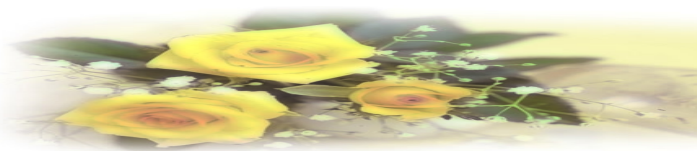




第五章 设备管理

(2) 将独占设备改造为共享设备。因为在SP00Ling系统中，实际上并没为任何进程分配设备，而只是在输入井或输出井中为进程分配一个存储区和建立一张I/O请求表。这样，便把独占设备改造为共享设备。

(3) 实现了虚拟设备功能。宏观上，虽然是多个进程在同时使用一台独占设备，而对于每一个进程而言，他们都会认为自己是独占了一个设备。当然，该设备只是逻辑上的设备。SPOOLing系统实现了将独占设备变换为若干台对应的逻辑设备的功能。



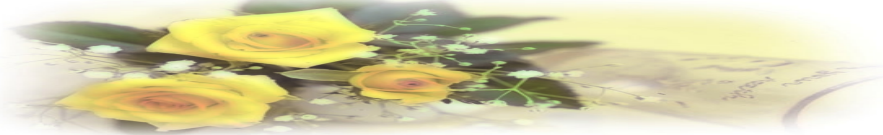


5.6 磁盘存储器的管理

5.6.1 磁盘性能简述

1. 数据的组织和格式

磁盘设备可包括一或多个物理盘片，每个磁盘片分一个或两个存储面(surface)(见图5-23(a))，每个磁盘面被组织成若干个同心环，这种环称为磁道(track)，各磁道之间留有必要的间隙。为使处理简单起见，在每条磁道上可存储相同数目的二进制位。这样，磁盘密度即每英寸中所存储的位数，显然是内层磁道的密度较外层磁道的密度高。每条磁道又被逻辑上划分成若干个扇区(sectors)，软盘大约为8~32个扇区，硬盘则可多达数百个，图5-23(b)显示了一个磁道分成8个扇区。一个扇区称为一个盘块(或数据块)，常常叫做磁盘扇区。各扇区之间保留一定的间隙。



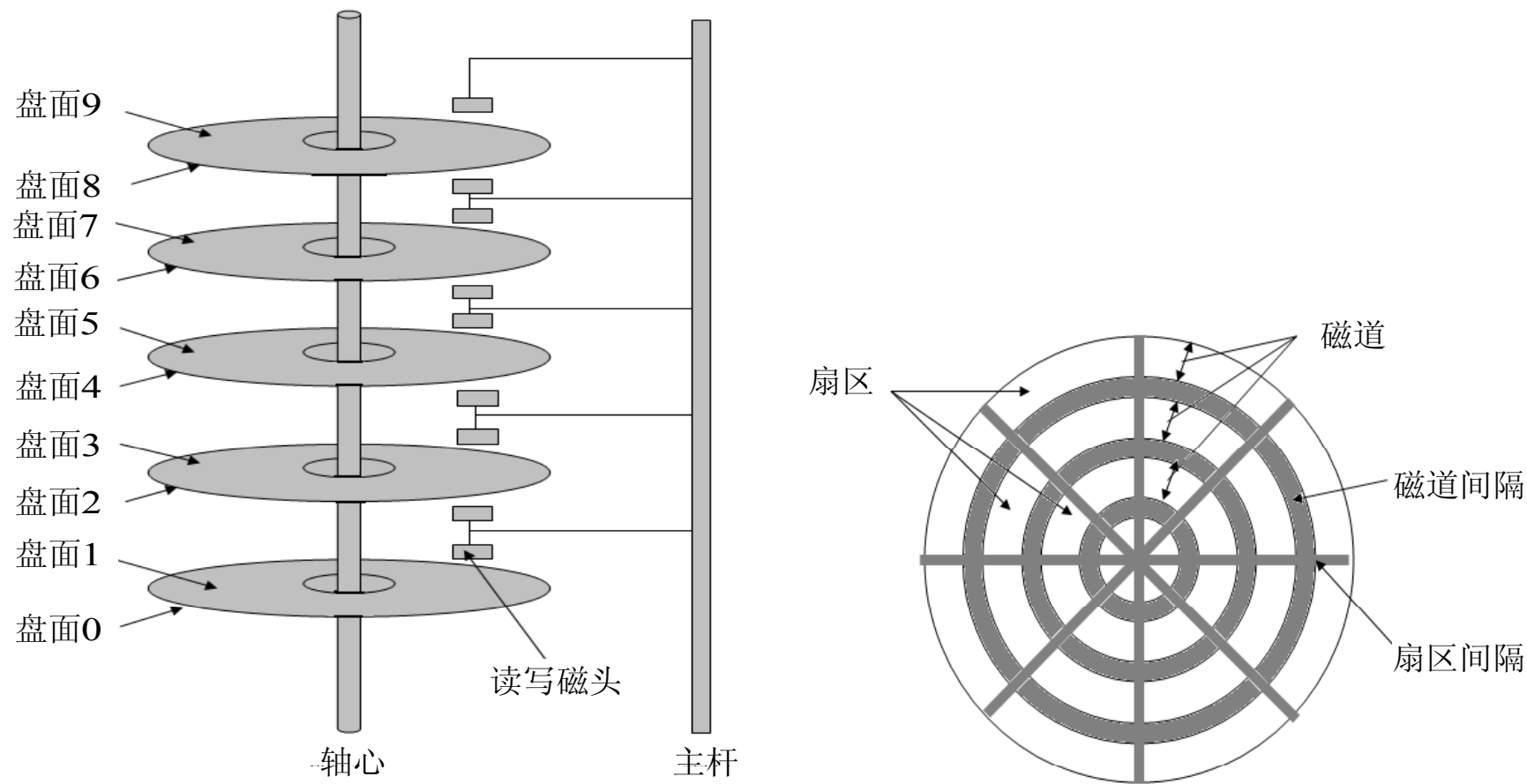
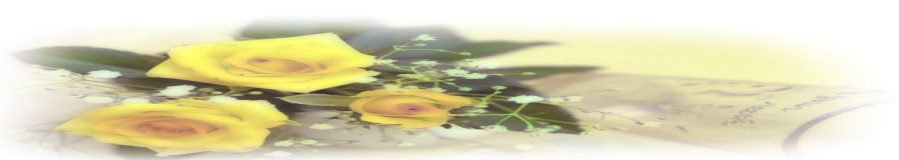


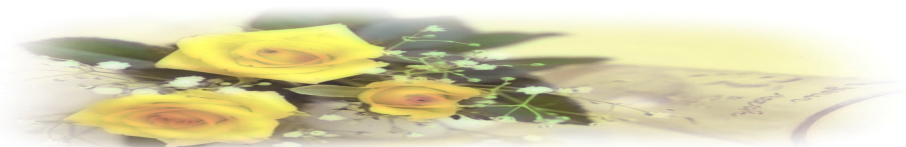
图5-23 磁盘的结构和布局





第五章 设备管理

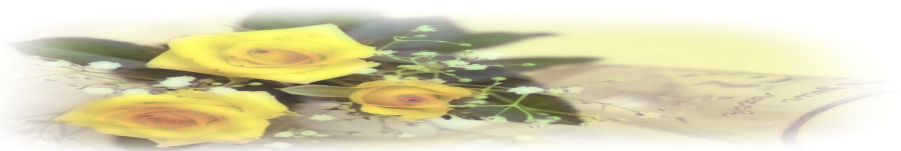
一个物理记录存储在一个扇区上，磁盘上存储的物理记录块数目是由扇区数、磁道数以及磁盘面数所决定的。例如，一个10 GB容量的磁盘，有8个双面可存储盘片，共16个存储面(盘面)，每面有16 383个磁道(也称柱面)，63个扇区。





第五章 设备管理

为了提高磁盘的存储容量，充分利用磁盘外面磁道的存储能力，现代磁盘不再把内外磁道划分为相同数目的扇区，而是利用外层磁道容量较内层磁道大的特点，将盘面划分成若干条环带，使得同一环带内的所有磁道具有相同的扇区数。显然，外层环带的磁道拥有较内层环带的磁道更多的扇区。为了减少这种磁道和扇区在盘面分布的几何形式变化对驱动程序的影响，大多数现代磁盘都隐藏了这些细节，向操作系统提供虚拟几何的磁盘规格，而不是实际的物理几何规格。



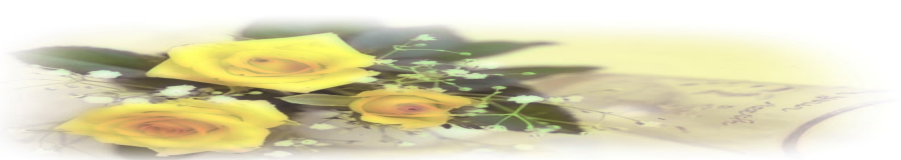


第五章 设备管理

为了在磁盘上存储数据，必须先将磁盘低级格式化。图5-24示出了一种温盘(温切斯特盘)中一条磁道格式化的情况。其中每条磁道含有30个固定大小的扇区，每个扇区容量为600个字节，其中512个字节存放数据，其余的用于存放控制信息。每个扇区包括两个字段：

(1) 标识符字段，其中一个字节的SYNCH具有特定的位图像，作为该字段的定界符，利用磁道号、磁头号及扇区号三者来标识一个扇区；CRC字段用于段校验。

(2) 数据字段，其中可存放512个字节的数据。



第五章 设备管理

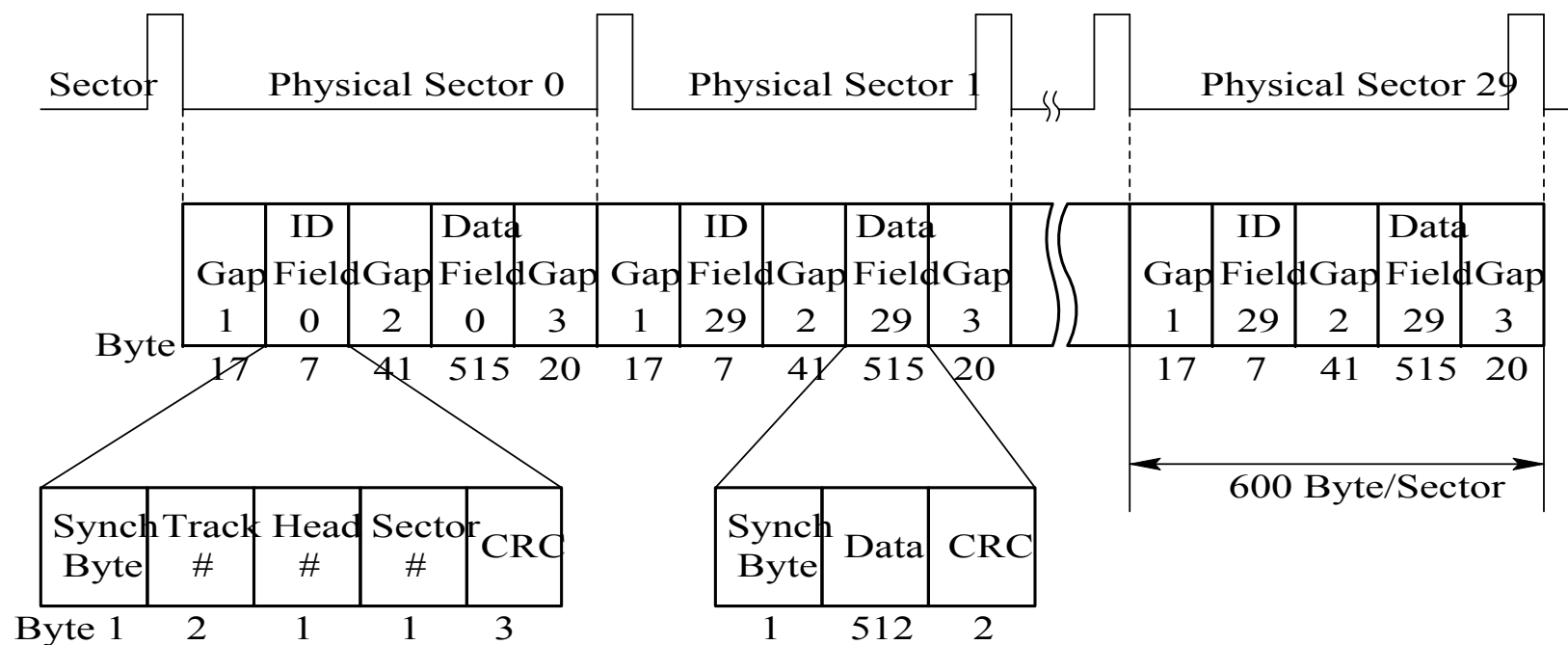
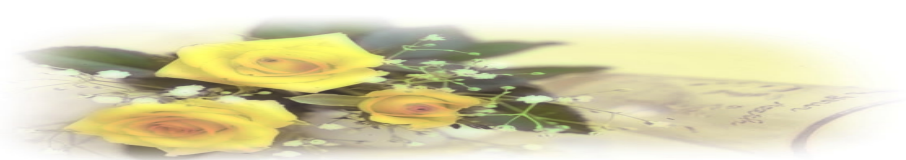


图5-24 磁盘的格式化



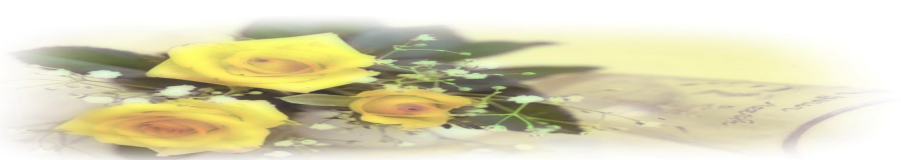


2. 磁盘的类型

对磁盘，可以从不同的角度进行分类。最常见的有：将磁盘分成硬盘和软盘、单片盘和多片盘、固定头磁盘和活动头(移动头)磁盘等。下面仅对固定头磁盘和移动头磁盘做些介绍。

1) 固定头磁盘

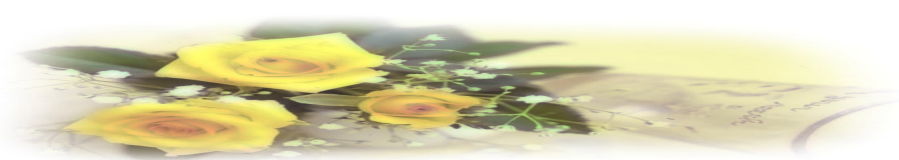
这种磁盘在每条磁道上都有一读/写磁头，所有的磁头都被装在一刚性磁臂中。通过这些磁头可访问所有各磁道，并进行并行读/写，有效地提高了磁盘的I/O速度。这种结构的磁盘主要用于大容量磁盘上。





2) 移动头磁盘

每一个盘面仅配有一个磁头，也被装入磁臂中。为能访问该盘面上的所有磁道，该磁头必须能移动以进行寻道。可见，移动磁头仅能以串行方式读/写，致使其I/O速度较慢；但由于其结构简单，故仍广泛应用于中小型磁盘设备中。在微型机上配置的温盘和软盘都采用移动磁头结构，故本节主要针对这类磁盘的I/O进行讨论。





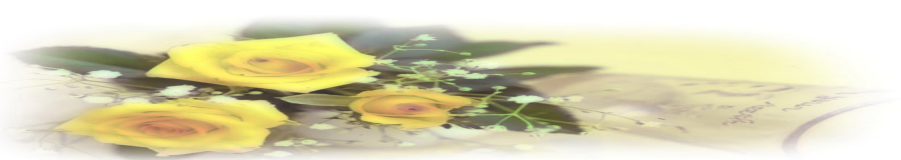
3. 磁盘访问时间

1) 寻道时间 T_s

这是指把磁臂(磁头)移动到指定磁道上所经历的时间。该时间是启动磁臂的时间 s 与磁头移动 n 条磁道所花费的时间之和，即

$$T_s = m \times n + s$$

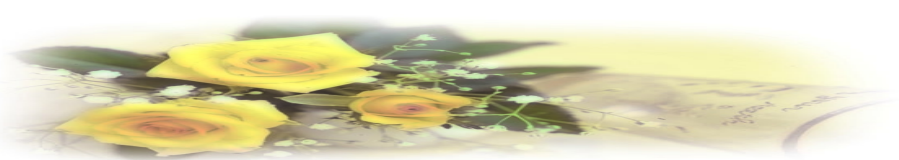
其中， m 是一常数，与磁盘驱动器的速度有关。对于一般磁盘， $m=0.2$ ；对于高速磁盘， $m \leq 0.1$ ，磁臂的启动时间约为2 ms。这样，对于一般的温盘，其寻道时间将随寻道距离的增加而增大，大体上是5~30 ms。





2) 旋转延迟时间 T_r

这是指定扇区移动到磁头下面所经历的时间。不同的磁盘类型中，旋转速度至少相差一个数量级，如软盘为300 r/min，硬盘一般为7200~15 000 r/min，甚至更高。对于磁盘旋转延迟时间而言，如硬盘，旋转速度为15 000 r/min，每转需时4 ms，平均旋转延迟时间 T_r 为2 ms；而软盘，其旋转速度为300 r/min或600 r/min，这样，平均 T_r 为50~100 ms。





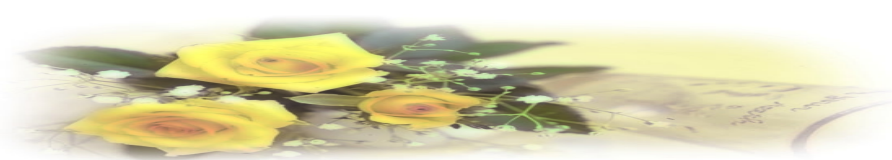
3) 传输时间 T_t

这是指把数据从磁盘读出或向磁盘写入数据所经历的时间。 T_t 的大小与每次所读/写的字节数 b 和旋转速度有关:

$$T_t = \frac{b}{rN}$$

其中, r 为磁盘每秒钟的转数; N 为一条磁道上的字节数, 当一次读/写的字节数相当于半条磁道上的字节数时, T_t 与 T_r 相同。因此, 可将访问时间 T_a 表示为

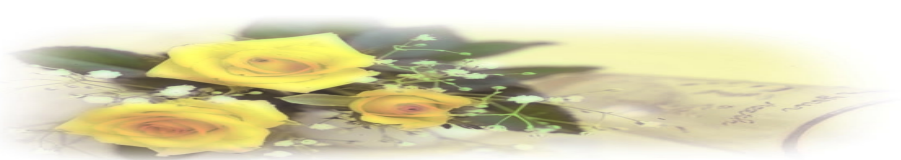
$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$





第五章 设备管理

由上式可以看出，在访问时间中，寻道时间和旋转延迟时间基本上都与所读/写数据的多少无关，而且它通常占据了访问时间中的大头。例如，我们假定寻道时间和旋转延迟时间平均为20 ms，而磁盘的传输速率为10 MB/s，如果要传输10 KB的数据，此时总的访问时间为21 ms，可见传输时间所占比例是非常小的。当传输100 KB数据时，其访问时间也只是30 ms，即当传输的数据量增大10倍时，访问时间只增加约50%。目前磁盘的传输速率已达80 MB/s以上，数据传输时间所占的比例更低。可见，适当地集中数据(不要太零散)传输，将有利于提高传输效率。

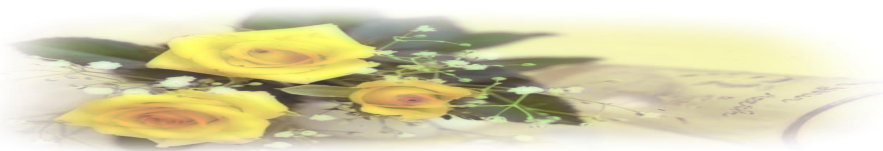




5.6.2 磁盘调度

1. 先来先服务(FCFS, First Come First Served)

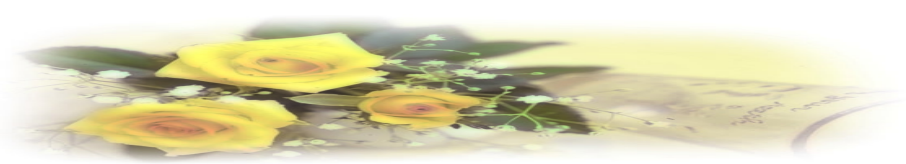
这是一种最简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。此算法的优点是公平、简单，且每个进程的请求都能依次地得到处理，不会出现某一进程的请求长期得不到满足的情况。但此算法由于未对寻道进行优化，致使平均寻道时间可能较长。图5-25示出了有9个进程先后提出磁盘I/O请求时，按FCFS算法进行调度的情况。这里将进程号(请求者)按他们发出请求的先后次序排队。这样，平均寻道距离为55.3条磁道，与后面即将讲到的几种调度算法相比，其平均寻道距离较大，故FCFS算法仅适用于请求磁盘I/O的进程数目较少的场合。





(从 100 号磁道开始)	
被访问的下一个磁道号	移动距离 (磁道数)
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
平均寻道长度: 55.3	

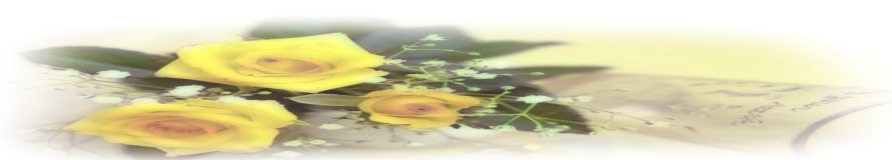
图5-25 FCFS调度算法





2. 最短寻道时间优先 (SSTF, Shortest Seek Time First)

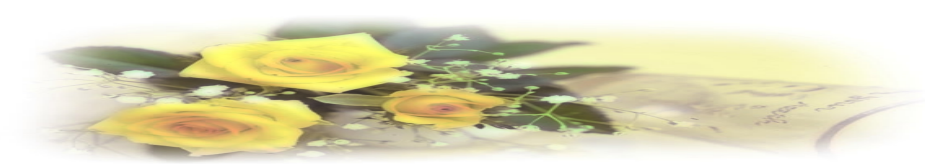
该算法选择这样的进程：其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短。但这种算法不能保证平均寻道时间最短。图5-26示出了按SSTF算法进行调度时，各进程被调度的次序、每次磁头移动的距离，以及9次调度磁头平均移动的距离。比较图5-25和图5-26可以看出，SSTF算法的平均每次磁头移动距离明显低于FCFS的距离，因而SSTF较之FCFS有更好的寻道性能，故过去曾一度被广泛采用。





(从 100 号磁道开始)	
被访问的下一个磁道号	移动距离 (磁道数)
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
平均寻道长度: 27.5	

图5-26 SSTF调度算法

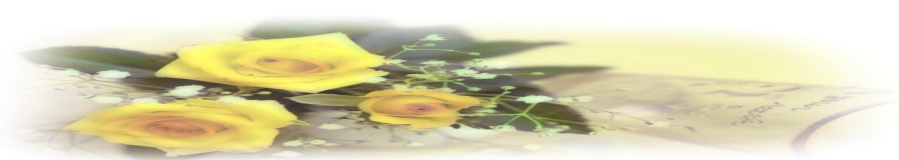




3. 扫描(SCAN)算法

1) 进程“饥饿”现象

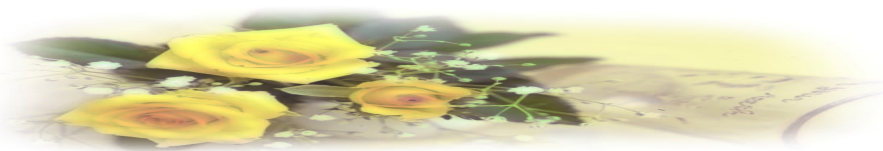
SSTF算法虽然能获得较好的寻道性能，但却可能导致某个进程发生“饥饿”(Starvation)现象。因为只要不断有新进程的请求到达，且其所要访问的磁道与磁头当前所在磁道的距离较近，这种新进程的I/O请求必然优先满足。对SSTF算法略加修改后所形成的SCAN算法，即可防止老进程出现“饥饿”现象。





2) SCAN算法

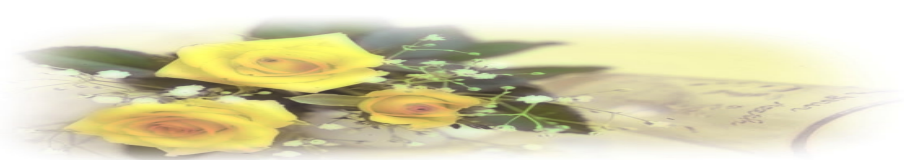
该算法不仅考虑到欲访问的磁道与当前磁道间的距离，更优先考虑的是磁头当前的移动方向。例如，当磁头正在自里向外移动时，SCAN算法所考虑的下一个访问对象，应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直至再无更外的磁道需要访问时，才将磁臂换向为自外向里移动。这时，同样也是每次选择这样的进程来调度，即要访问的磁道在当前位置内距离最近者，这样，磁头又逐步地从外向里移动，直至再无更里面的磁道要访问，从而避免了出现“饥饿”现象。由于在这种算法中磁头移动的规律颇似电梯的运行，因而又常称之为电梯调度算法。图5-27示出了按SCAN算法对9个进程进行调度及磁头移动的情况。





(从 100#磁道开始, 向磁道号增加方向访问)	
被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20
平均寻道长度: 27.8	

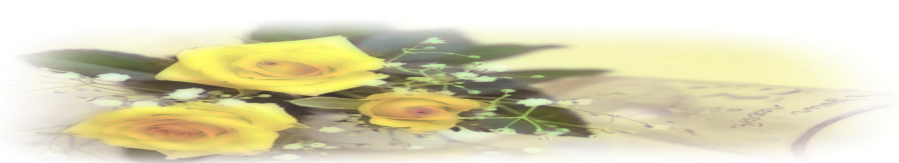
图5-27 SCAN调度算法示例





4. 循环扫描(CSCAN)算法

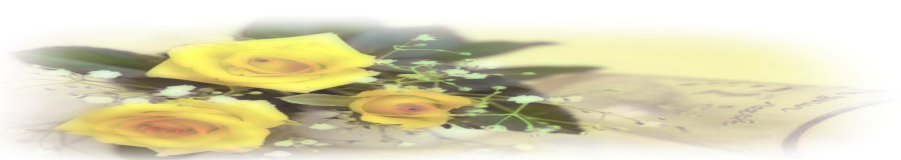
SCAN算法既能获得较好的寻道性能，又能防止“饥饿”现象，故被广泛用于大、中、小型机器和网络中的磁盘调度。但SCAN也存在这样的问题：当磁头刚从里向外移动而越过了某一磁道时，恰好又有一进程请求访问此磁道，这时，该进程必须等待，待磁头继续从里向外，然后再从外向里扫描完所有要访问的磁道后，才处理该进程的请求，致使该进程的请求被大大地推迟。





第五章 设备管理

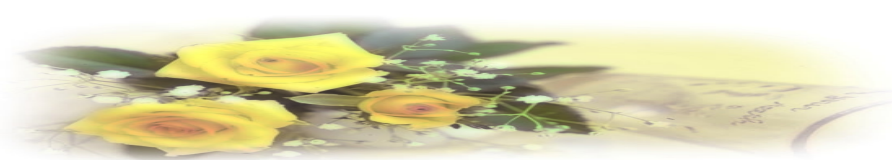
为了减少这种延迟，CSCAN算法规定磁头单向移动，例如，只是自里向外移动，当磁头移到最外的磁道并访问后，磁头立即返回到最里的欲访问的磁道，亦即将最小磁道号紧接着最大磁道号构成循环，进行循环扫描。采用循环扫描方式后，上述请求进程的请求延迟将从原来的 $2T$ 减为 $T + S_{\max}$ ，其中， T 为由里向外或由外向里单向扫描完要访问的磁道所需的寻道时间，而 $S_{\max}$ 是将磁头从最外面被访问的磁道直接移到最里面欲访问的磁道(或相反)的寻道时间。图5-28示出了CSCAN算法对9个进程调度的次序及每次磁头移动的距离。





(从 100#磁道开始, 向磁道号增加方向访问)	
被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
平均寻道长度: 35.8	

图5-28 CSCAN调度算法示例

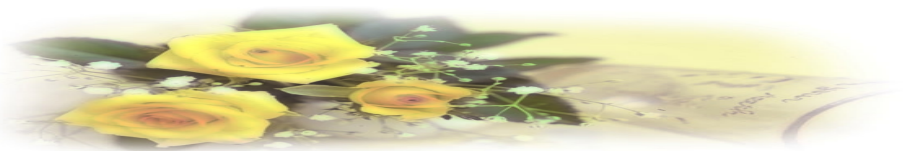




5. NStepSCAN和FSCAN调度算法

1) NStepSCAN算法

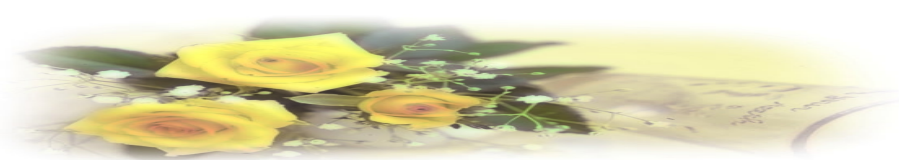
在SSTF、SCAN及CSCAN几种调度算法中，都可能会出现磁臂停留在某处不动的情况，例如，有一个或几个进程对某一磁道有较高的访问频率，即这个(些)进程反复请求对某一磁道的I/O操作，从而垄断了整个磁盘设备。我们把这一现象称为“磁臂粘着”(Armstickiness)。在高密度磁盘上容易出现此情况。N步SCAN算法是将磁盘请求队列分成若干个长度为N的子队列，磁盘调度将按FCFS算法依次处理这些子队列。而每处理一个队列时又是按SCAN算法，对一个队列处理完后，再处理其他队列。当正在处理某子队列时，如果又出现新的磁盘I/O请求，便将新请求进程放入其他队列，这样就可避免出现粘着现象。当N值取得很大时，会使N步扫描法的性能接近于SCAN算法的性能；当N=1时，N步SCAN算法便蜕化为FCFS算法。





2) FSCAN算法

FSCAN算法实质上是N步SCAN算法的简化，即FSCAN只将磁盘请求队列分成两个子队列。一个是由当前所有请求磁盘I/O的进程形成的队列，由磁盘调度按SCAN算法进行处理。在扫描期间，将新出现的所有请求磁盘I/O的进程，放入另一个等待处理的请求队列。这样，所有的新请求都将被推迟到下一次扫描时处理。

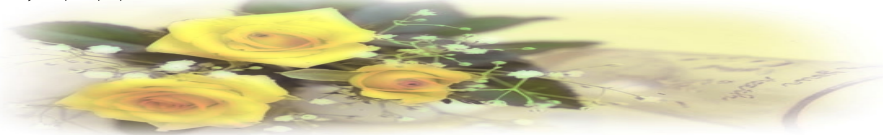




5.6.3 磁盘高速缓存

1. 磁盘高速缓存的形式

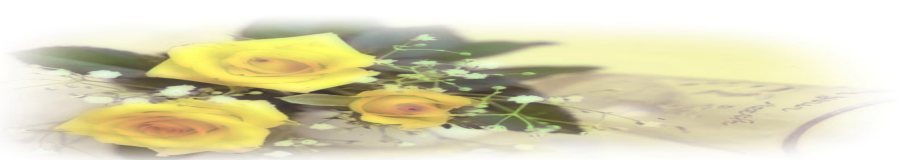
这里所说的磁盘高速缓存，并非通常意义下的内存和CPU之间所增设的一个小容量高速存储器，而是指利用内存中的存储空间来暂存从磁盘中读出的一系列盘块中的信息。因此，这里的高速缓存是一组在逻辑上属于磁盘，而物理上是驻留在内存中的盘块。高速缓存在内存中可分成两种形式。第一种是在内存中开辟一个单独的存储空间来作为磁盘高速缓存，其大小是固定的，不会受应用程序多少的影响；第二种是把所有未利用的内存空间变为一个缓冲池，供请求分页系统和磁盘I/O时(作为磁盘高速缓存)共享。此时，高速缓存的大小显然不再是固定的。当磁盘I/O的频繁程度较高时，该缓冲池可能包含更多的内存空间；而在应用程序运行得较多时，该缓冲池可能只剩下较少的内存空间。





2. 数据交付方式

数据交付(Data Delivery)是指将磁盘高速缓存中的数据传送给请求者进程。当有一进程请求访问某个盘块中的数据时,由核心先去查看磁盘高速缓冲器,看其中是否存在进程所需访问的盘块数据的拷贝。若有其拷贝,便直接从高速缓存中提取数据交付给请求者进程,这样,就避免了访盘操作,从而使本次访问速度提高4~6个数量级;否则,应先从磁盘中将所要访问的数据读入并交付给请求者进程,同时也将数据送高速缓存。当以后又需要访问该盘块的数据时,便可直接从高速缓存中提取。





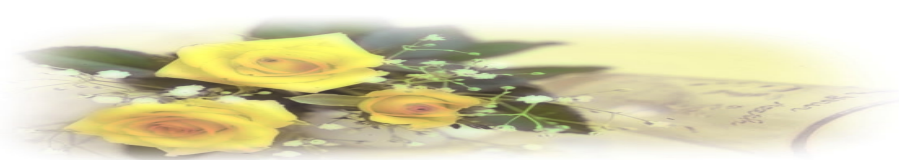
第五章 设备管理

系统可以采取两种方式将数据交付给请求进程：

(1) 数据交付。这是直接将高速缓存中的数据，传送到请求者进程的内存工作区中。

(2) 指针交付。这是只将指向高速缓存中某区域的指针交付给请求者进程。

后一种方式由于所传送的数据量少，因而节省了数据从磁盘高速缓存到进程的内存工作区的时间。

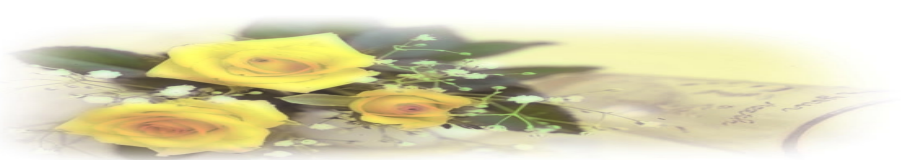




3. 置换算法

如同请求调页(段)一样, 在将磁盘中的盘块数据读入高速缓存时, 同样会出现因高速缓存中已装满盘块数据而需要将该数据先换出的问题。相应地, 也必然存在着采用哪种置换算法的问题。较常用的置换算法仍然是最近最久未使用算法LRU、最近未使用算法NRU及最少使用算法LFU等。

由于请求调页中的联想存储器与高速缓存(磁盘I/O中)的工作情况不同, 因而使得在置换算法中所应考虑的问题也有所差异。因此, 现在不少系统在设计其高速缓存的置换算法时, 除了考虑到最近最久未使用这一原则外, 还考虑了以下几点:

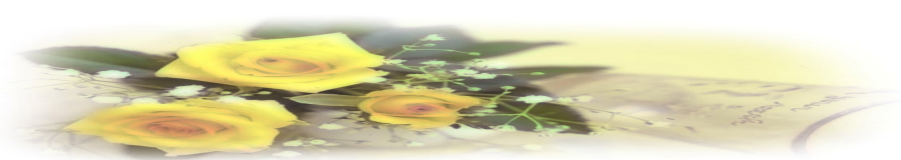




第五章 设备管理

1) 访问频率

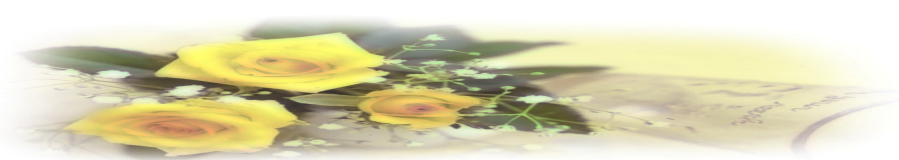
通常，每执行一条指令时，便可能访问一次联想存储器，亦即联想存储器的访问频率，基本上与指令执行的频率相当。而对高速缓存的访问频率，则与磁盘I/O的频率相当。因此，对联想存储器的访问频率远远高于对高速缓存的访问频率。





2) 可预见性

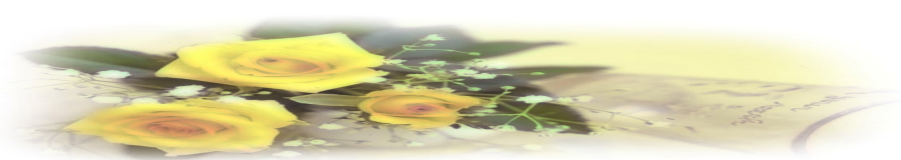
在高速缓存中的各盘块数据，有哪些数据可能在较长时间内不会再被访问，又有哪些数据可能很快就再被访问，会有相当一部分是可预知的。例如，对二次地址及目录块等，在它被访问后，可能会很久都不再被访问。又如，正在写入数据的未满足盘块，可能会很快又被访问。





3) 数据的一致性

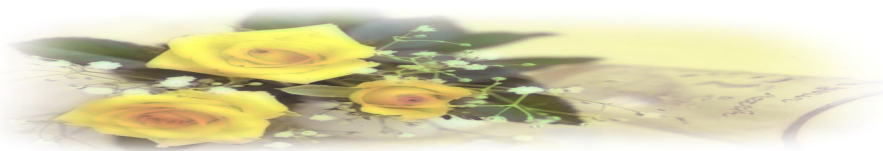
由于高速缓存是做在内存中的，而内存一般又是一种易失性的存储器，一旦系统发生故障，存放在高速缓存中的数据将会丢失；而其中有些盘块(如索引结点盘块)中的数据已被修改，但尚未拷回磁盘，因此，当系统发生故障后，可能会造成数据的不一致性。





第五章 设备管理

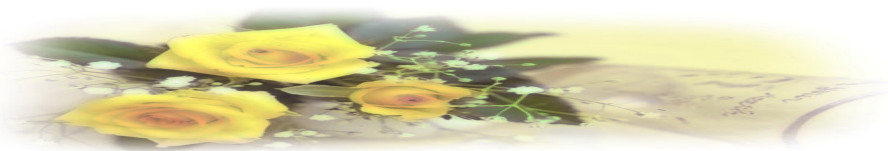
基于上述考虑，在有的系统中便将高速缓存中的所有盘块数据拉成一条LRU链。对于那些会严重影响到数据一致性的盘块数据和很久都可能不再使用的盘块数据，都放在LRU链的头部，使它们能被优先写回磁盘，以减少发生数据不一致性的概率，或者可以尽早地腾出高速缓存的空间。对于那些可能在不久之后便要再使用的盘块数据，应挂在LRU链的尾部，以便在不久以后需要时，只要该数据块尚未从链中移至链首而被写回磁盘，便可直接到高速缓存中(即LRU链中)去找到它们。





4. 周期性地写回磁盘

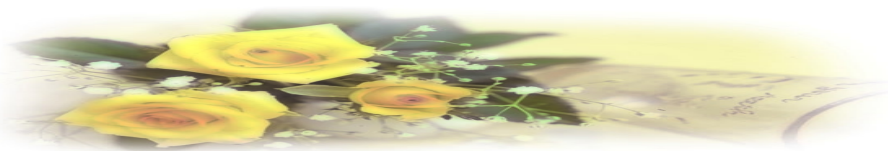
还有一种情况值得注意:那就是根据LRU算法,那些经常要被访问的盘块数据,可能会一直保留在高速缓存中,长期不会被写回磁盘。(注意,LRU链意味着链中任一元素在被访问之后,总是又被挂到链尾而不被写回磁盘;只是一直未被访问的元素,才有可能移到链首,而被写回磁盘。)例如,一位学者一上班便开始撰写论文,并边写边修改,他正在写作的论文就一直保存在高速缓存的LRU链中。如果在快下班时,系统突然发生故障,这样,存放在高速缓存中的已写论文将随之消失,致使他枉费了一天的劳动。





第五章 设备管理

为了解决这一问题，在UNIX系统中专门增设了一个修改(update)程序，使之在后台运行，该程序周期性地调用一个系统调用SYNC。该调用的主要功能是强制性地将所有在高速缓存中已修改的盘块数据写回磁盘。一般是把两次调用SYNC的时间间隔定为30 s。这样，因系统故障所造成的工作损失不会超过30 s的劳动量。而在MS-DOS中所采用的方法是：只要高速缓存中的某盘块数据被修改，便立即将它写回磁盘，并将这种高速缓存称为“写穿透、高速缓存”(write-through cache)。MS-DOS所采用的写回方式，几乎不会造成数据的丢失，但须频繁地启动磁盘。

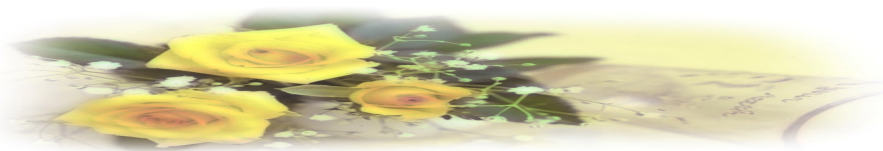




5.6.4 提高磁盘I/O速度的其它方法

1. 提前读(Read-ahead)

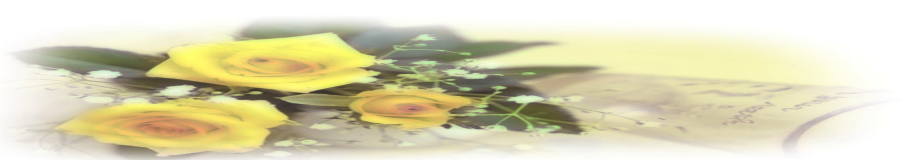
用户(进程)对文件进行访问时,经常采用顺序访问方式,即顺序地访问文件各盘块的数据。在这种情况下,在读当前块时可以预知下一次要读的盘块。因此,可以采取预先读方式,即在读当前块的同时,还要求将下一个盘块(提前读的块)中的数据也读入缓冲区。这样,当下一次要读该盘块中的数据时,由于该数据已被提前读入缓冲区,因而此时便可直接从缓冲区中取得下一盘块的数据,而不需再去启动磁盘I/O,从而大大减少了读数据的时间。这也就等效于提高了磁盘I/O的速度。“提前读”功能已被广泛采用,如在UNIX系统、OS/2,以及在3 Plus和Netware等的网络OS中,都已采用该功能。





2. 延迟写

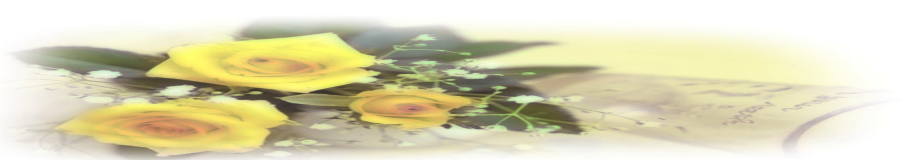
延迟写是指在缓冲区A中的数据，本应立即写回磁盘，但考虑到该缓冲区中的数据在不久之后可能还会再被本进程或其它进程访问(共享资源)，因而并不立即将该缓冲区A中的数据写入磁盘，而是将它挂在空闲缓冲区队列的末尾。随着空闲缓冲区的使用，缓冲区也缓缓往前移动，直至移到空闲缓冲队列之首。当再有进程申请到该缓冲区时，才将该缓冲区中的数据写入磁盘，而把该缓冲区作为空闲缓冲区分配出去。当该缓冲区A仍在队列中时，任何访问该数据的进程，都可直接读出其中的数据而不必去访问磁盘。这样，又可进一步减小等效的磁盘I/O时间。同样，“延迟写”功能已在UNIX系统、OS/2等OS中被广泛采用。





3. 优化物理块的分布

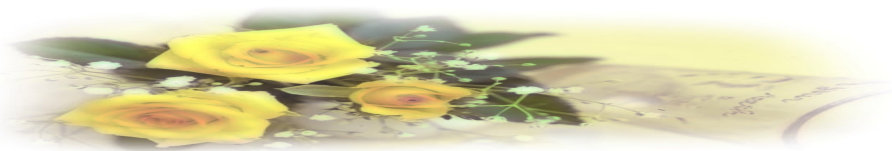
另一种提高磁盘I/O速度的重要措施是优化文件物理块的分布，使磁头的移动距离最小。虽然链接分配和索引分配方式都允许将一个文件的物理块分散在磁盘的任意位置，但如果将一个文件的多个物理块安排得过于分散，会增加磁头的移动距离。例如，将文件的第一个盘块安排在最里的一条磁道上，而把第二个盘块安排在最外的一条磁道上，这样，在读完第一个盘块后转去读第二个盘块时，磁头要从最里的磁道移到最外的磁道上。如果我们将这两个数据块安排在属于同一条磁道的两个盘块上，显然会由于消除了磁头在磁道间的移动，而大大提高对这两个盘块的访问速度。





第五章 设备管理

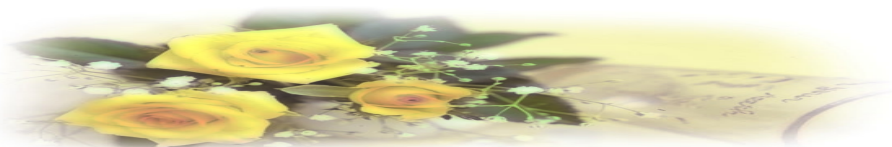
对文件盘块位置的优化，应在为文件分配盘块时进行。如果系统中的空白存储空间是采用位示图方式表示的，则要将同属于一个文件的盘块安排在同一条磁道上或相邻的磁道上是十分容易的事。这时，只要从位示图中找到一片相邻接的多个空闲盘块即可。但当系统采用线性表(链)法来组织空闲存储空间时，要为一文件分配多个相邻接的盘块，就要困难一些。此时，我们可以将在同一条磁道上的若干个盘块组成一簇，例如，一簇包括4个盘块，在分配存储空间时，以簇为单位进行分配。这样就可以保证在访问这几个盘块时，不必移动磁头或者仅移动一条磁道的距离，从而减少了磁头的平均移动距离。





4. 虚拟盘

所谓虚拟盘，是指利用内存空间去仿真磁盘，又称为RAM盘。该盘的设备驱动程序也可以接受所有标准的磁盘操作，但这些操作的执行，不是在磁盘上而是在内存中。这些对用户都是透明的。换言之，用户并不会发现这与真正的磁盘操作有什么不同，而仅仅是略微快些而已。虚拟盘的主要问题是：它是易失性存储器，故一旦系统或电源发生故障，或系统再启动时，原来保存在虚拟盘中的数据将会丢失。因此，虚拟盘通常用于存放临时文件，如编译程序所产生的目标程序等。虚拟盘与磁盘高速缓存的主要区别在于：虚拟盘中的内容完全由用户控制，而高速磁盘缓存中的内容则是由OS控制的。例如，RAM盘在开始时是空的，仅当用户(程序)在RAM盘中创建了文件后，RAM盘中才有内容。

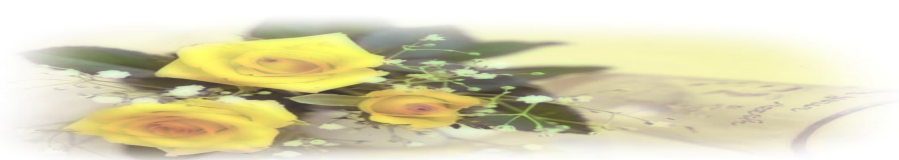




5.6.5 廉价磁盘冗余阵列

1. 并行交叉存取

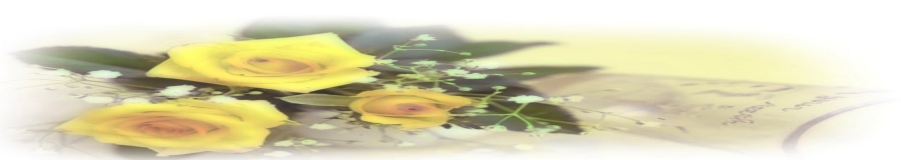
为了提高对磁盘的访问速度，已把在大、中型机中应用的交叉存取(Interleave)技术应用到了磁盘存储系统中。在该系统中，有多台磁盘驱动器，系统将每一盘块中的数据分为若干个子盘块数据，再把每一个子盘块的数据分别存储到各个不同磁盘中的相同位置上。





第五章 设备管理

在以后，当要将一个盘块的数据传送到内存时，采取并行传输方式，将各个盘块中的子盘块数据同时向内存中传输，从而使传输时间大大减少。例如，在存放一个文件时，可将该文件中的第一个数据子块放在第一个磁盘驱动器上；将文件的第二个数据子块放在第二个磁盘上；.....；将第 N 个数据子块，放在第 N 个驱动器上。以后在读取数据时，采取并行读取方式，即同时从第 $1 \sim N$ 个数据子块读出数据，这样便把磁盘I/O的速度提高了 $N-1$ 倍。图5-29示出了磁盘并行交叉存取方式。



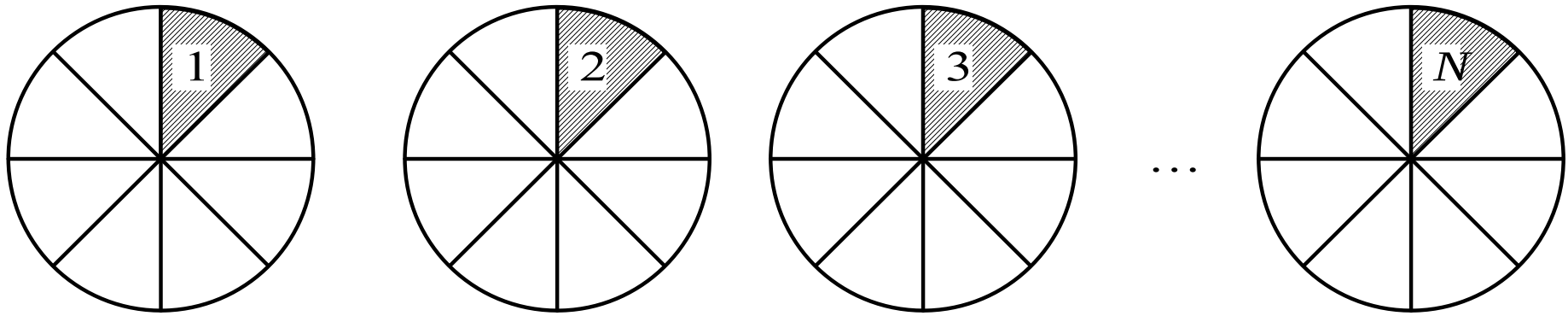
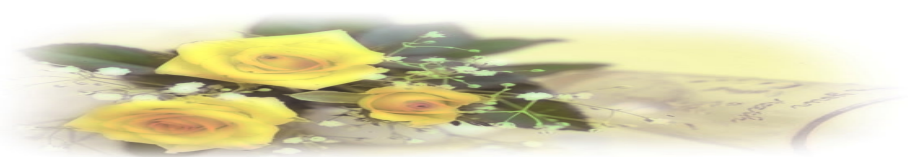


图5-29 磁盘并行交叉存取方式

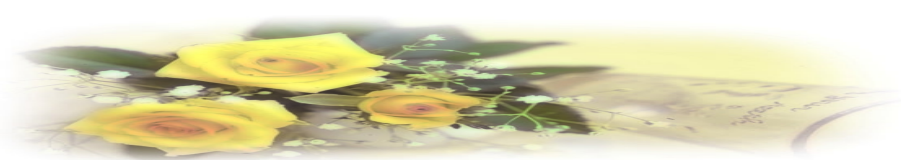




2. RAID的分级

RAID在刚被推出时，是分成6级的，即RAID 0级至RAID 5级，后来又增加了RAID 6级和RAID 7级。

(1) RAID 0级。本级仅提供了并行交叉存取。它虽能有效地提高磁盘I/O速度，但并无冗余校验功能，致使磁盘系统的可靠性不好。只要阵列中有一个磁盘损坏，便会造成不可弥补的数据丢失，故较少使用。

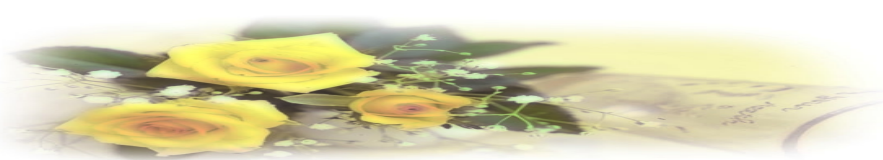




第五章 设备管理

(2) RAID 1级。它具有磁盘镜像功能，例如，当磁盘阵列中具有8个盘时，可利用其中4个作为数据盘，另外 4 个作为镜像盘，在每次访问磁盘时，可利用并行读、写特性，将数据分块同时写入主盘和镜像盘。故其比传统的镜像盘速度快，但其磁盘容量的利用率只有50%，它是以牺牲磁盘容量为代价的。

(3) RAID 3级。这是具有并行传输功能的磁盘阵列。它利用一台奇偶校验盘来完成数据的校验功能，比起磁盘镜像，它减少了所需要的冗余磁盘数。例如，当阵列中只有7个盘时，可利用6个盘作数据盘，一个盘作校验盘。磁盘的利用率为6/7。RAID 3级经常用于科学计算和图像处理。

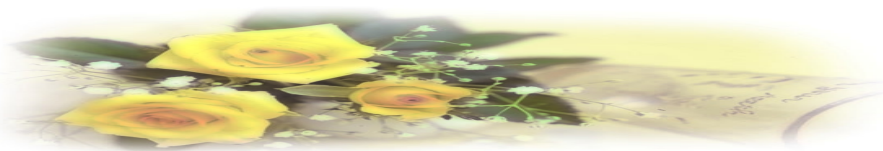




第五章 设备管理

(4) RAID 5级。这是一种具有独立传送功能的磁盘阵列。每个驱动器都各有自己独立的数据通路，独立地进行读/写，且无专门的校验盘。用来进行纠错的校验信息，是以螺旋 (Spiral) 方式散布在所有数据盘上。RAID 5级常用于 I/O较频繁的事务处理中。

(5) RAID 6级和RAID 7级。这是强化了了的RAID。在 RAID 6级的阵列中，设置了一个专用的、可快速访问的异步校验盘。该盘具有独立的数据访问通路，具有比RAID 3级及 RAID 5级更好的性能，但其性能改进得很有限，且价格昂贵。RAID 7级是对RAID 6级的改进，在该阵列中的所有磁盘，都具有较高的传输速率和优异的性能，是目前最高档次的磁盘阵列，但其价格也较高。

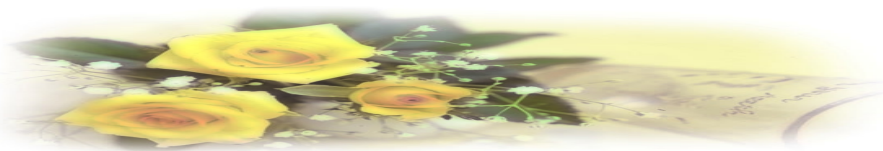




3. RAID的优点

RAID自1988年问世后，便引起了人们的普遍关注，并很快地流行起来。这主要是因为RAID具有下述一系列明显的优点：

(1) 可靠性高。RAID最大的特点就是它的高可靠性。除了RAID 0级外，其余各级都采用了容错技术。当阵列中某一磁盘损坏时，并不会造成数据的丢失，因为它既可实现磁盘镜像，又可实现磁盘双工，还可实现其它的冗余方式。所以此时可根据其它未损坏磁盘中的信息，来恢复已损坏的盘中的信息。它与单台磁盘机相比，其可靠性高出了一个数量级。

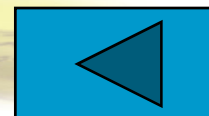
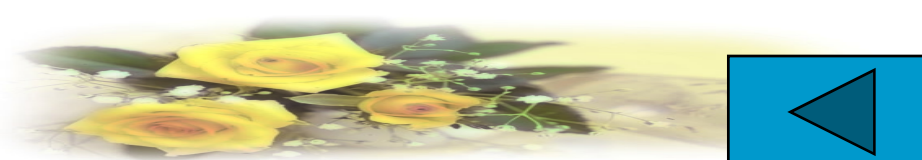




第五章 设备管理

(2) 磁盘I/O速度高。由于磁盘阵列可采取并行交叉存取方式，故可将磁盘I/O速度提高 $N-1$ 倍(N 为磁盘数目)。或者说，磁盘阵列可将磁盘I/O速度提高数倍至数十倍。

(3) 性能/价格比高。利用RAID技术来实现大容量高速存储器时，其体积与具有相同容量和速度的大型磁盘系统相比，只是后者的 $1/3$ ，价格也只是后者的 $1/3$ ，且可靠性高。换言之，它仅以牺牲 $1/N$ 的容量为代价，换取了高可靠性；而不像磁盘镜像及磁盘双工那样，须付出50%容量的代价。





第六章 文件管理

6.1 文件和文件系统

6.2 文件的逻辑结构

6.3 外存分配方式

6.4 目录管理

6.5 文件存储空间的管理

6.6 文件共享与文件保护

6.7 数据一致性控制



BACK



6.1 文件和文件系统

6.1.1 文件、记录和数据项

1. 数据项

在文件系统中，数据项是最低级的数据组织形式，可把它分成以下两种类型：

(1) 基本数据项。这是用于描述一个对象的某种属性的字符集，是数据组织中可以命名的最小逻辑数据单位，即原子数据，又称为数据元素或字段。它的命名往往与其属性一致。例如，用于描述一个学生的基本数据项有学号、姓名、年龄、所在班级等。





第六章 文件管理



(2) 组合数据项。它是由若干个基本数据项组成的，简称组项。例如，经理便是个组项，它由正经理和副经理两个基本项组成。又如，工资也是个组项，它可由基本工资、工龄工资和奖励工资等基本项所组成。

基本数据项除了数据名外，还应有数据类型。因为基本项仅是描述某个对象的属性，根据属性的不同，需要用不同的数据类型来描述。例如，在描述学生的学号时，应使用整数；描述学生的姓名则应使用字符串(含汉字)；描述性别时，可用逻辑变量或汉字。可见，由数据项的名字和类型两者共同定义了一个数据项的“型”。而表征一个实体在数据项上的数据则称为“值”。例如，学号/30211、姓名/王有年、性别/男等。





第六章 文件管理



2. 记录

记录是一组相关数据项的集合，用于描述一个对象在某方面的属性。一个记录应包含哪些数据项，取决于需要描述对象的哪个方面。而一个对象，由于他所处的环境不同可把他作为不同的对象。例如，一个学生，当把他作为班上的一名学生时，对他的描述应使用学号、姓名、年龄及所在系班，也可能还包括他所学过的课程的名称、成绩等数据项。但若把学生作为一个医疗对象时，对他描述的数据项则应使用诸如病历号、姓名、性别、出生年月、身高、体重、血压及病史等项。





第六章 文件管理



在诸多记录中，为了能惟一地标识一个记录，必须在一个记录的各个数据项中，确定出一个或几个数据项，把它们的集合称为关键字(key)。或者说，关键字是惟一能标识一个记录的数据项。通常，只需用一个数据项作为关键字。例如，前面的病历号或学号便可用来从诸多记录中标识出惟一的一个记录。然而有时找不到这样的数据项，只好把几个数据项定为能在诸多记录中惟一地标识出某个记录的关键字。





3. 文件

文件是指由创建者所定义的、具有文件名的一组相关元素的集合，可分为有结构文件和无结构文件两种。在有结构的文件中，文件由若干个相关记录组成；而无结构文件则被看成是一个字符流。文件在文件系统中是一个最大的数据单位，它描述了一个对象集。例如，可以将一个班的学生记录作为一个文件。一个文件必须要有一个文件名，它通常是由一串ASCII码或(和)汉字构成的，名字的长度因系统不同而异。如在有的系统中把名字规定为8个字符，而在有的系统中又规定可用14个字符。用户利用文件名来访问文件。





第六章 文件管理



此外，文件应具有自己的属性，属性可以包括：

- (1) 文件类型。可以从不同的角度来规定文件的类型，如源文件、目标文件及可执行文件等。
- (2) 文件长度。文件长度指文件的当前长度，长度的单位可以是字节、字或块，也可能是最大允许的长度。
- (3) 文件的物理位置。该项属性通常是用于指示文件在哪一个设备上及在该设备的哪个位置的指针。
- (4) 文件的建立时间。这是指文件最后一次修改时间等。





第六章 文件管理

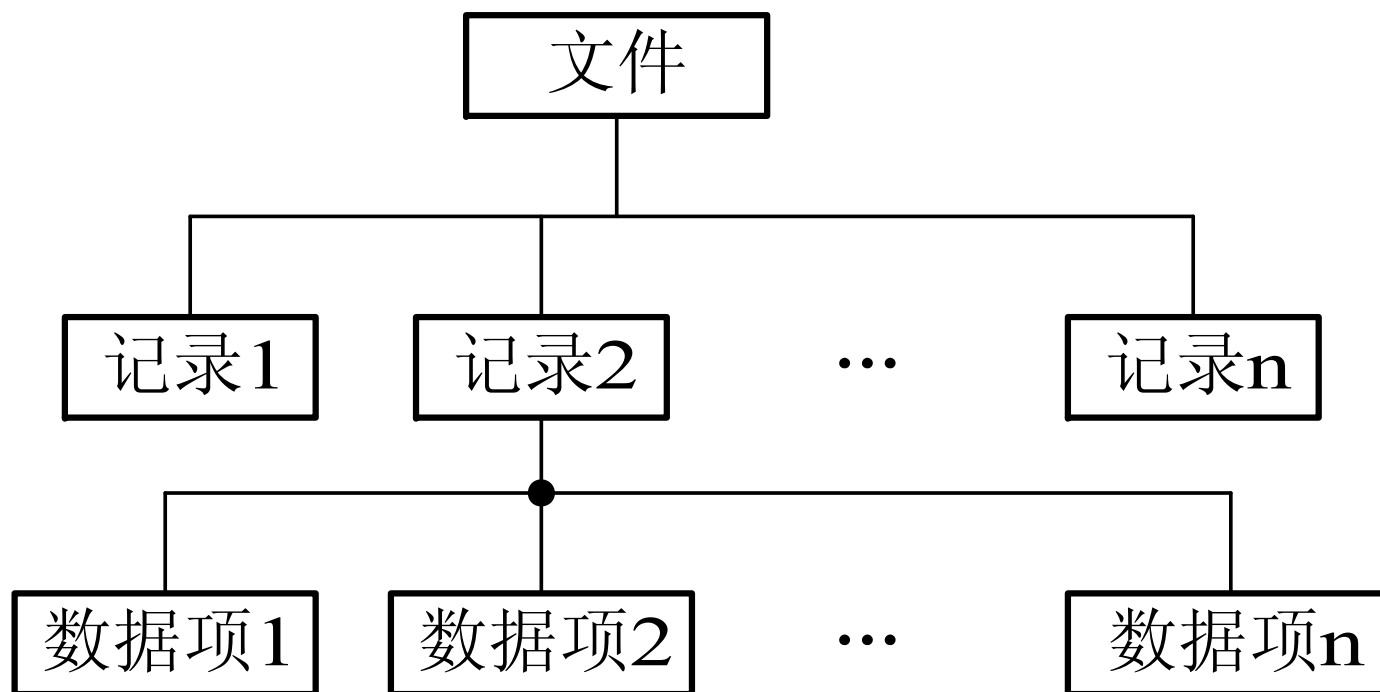


图6-1 文件、记录和数据项之间的层次关系





6.1.2 文件类型和文件系统模型

1. 文件类型

为了便于管理和控制文件而将文件分成若干种类型。由于不同系统对文件的管理方式不同，因而它们对文件的分类方法也有很大差异。为了方便系统和用户了解文件的类型，在许多OS中都把文件类型作为扩展名而缀在文件名的后面，在文件名和扩展名之间用“.”号隔开。下面是常用的几种文件分类方法。





第六章 文件管理



1) 按用途分类

根据文件的性质和用途的不同，可将文件分为三类：

(1) 系统文件。这是指由系统软件构成的文件。大多数的系统文件只允许用户调用，但不允许用户去读，更不允许修改；有的系统文件不直接对用户开放。

(2) 用户文件。指由用户的源代码、目标文件、可执行文件或数据等所构成的文件。用户将这些文件委托给系统保管。

(3) 库文件。这是由标准子例程及常用的例程等所构成的文件。这类文件允许用户调用，但不允许修改。





第六章 文件管理



2) 按文件中数据的形式分类

按这种方式分类，也可把文件分为三类：

(1) 源文件。这是指由源程序和数据构成的文件。通常由终端或输入设备输入的源程序和数据所形成的文件都属于源文件。它通常是由ASCII码或汉字所组成的。

(2) 目标文件。这是指把源程序经过相应语言的编译程序编译过，但尚未经过链接程序链接的目标代码所构成的文件。它属于二进制文件。通常，目标文件所使用的后缀名是“`.obj`”。

(3) 可执行文件。这是指把编译后所产生的目标代码再经过链接程序链接后所形成的文件。





第六章 文件管理



3) 按存取控制属性分类

根据系统管理员或用户所规定的存取控制属性，可将文件分为三类：

(1) 只执行文件。该类文件只允许被核准的用户调用执行，既不允许读，更不允许写。

(2) 只读文件。该类文件只允许文件主及被核准的用户去读，但不允许写。

(3) 读写文件。这是指允许文件主和被核准的用户去读或写的文件。





第六章 文件管理



4) 按组织形式和处理方式分类

根据文件的组织形式和系统对其的处理方式，可将文件分为三类：

(1) 普通文件：由ASCII码或二进制码组成的字符文件。一般用户建立的源程序文件、数据文件、目标代码文件及操作系统自身代码文件、库文件、实用程序文件等都是普通文件，它们通常存储在外存储设备上。

(2) 目录文件：由文件目录组成的，用来管理和实现文件系统功能的系统文件，通过目录文件可以对其它文件的信息进行检索。由于目录文件也是由字符序列构成，因此对其可进行与普通文件一样的种种文件操作。





第六章 文件管理



(3) 特殊文件：特指系统中的各类I/O设备。为了便于统一管理，系统将所有的输入/输出设备都视为文件，按文件方式提供给用户使用，如目录的检索、权限的验证等都与普通文件相似，只是对这些文件的操作是和设备驱动程序紧密相连的，系统将这些操作转为对具体设备的操作。根据设备数据交换单位的不同，又可将特殊文件分为块设备文件和字符设备文件。前者用于磁盘、光盘或磁带等块设备的I/O 操作，而后者用于终端、打印机等字符设备的I/O 操作。





2. 文件系统模型

图6-2示出了文件系统的模型。可将该模型分为三个层次，其最底层是对象及其属性；中间层是对对象进行操纵和管理的软件集合；最高层是文件系统提供给用户的接口。





第六章 文件管理

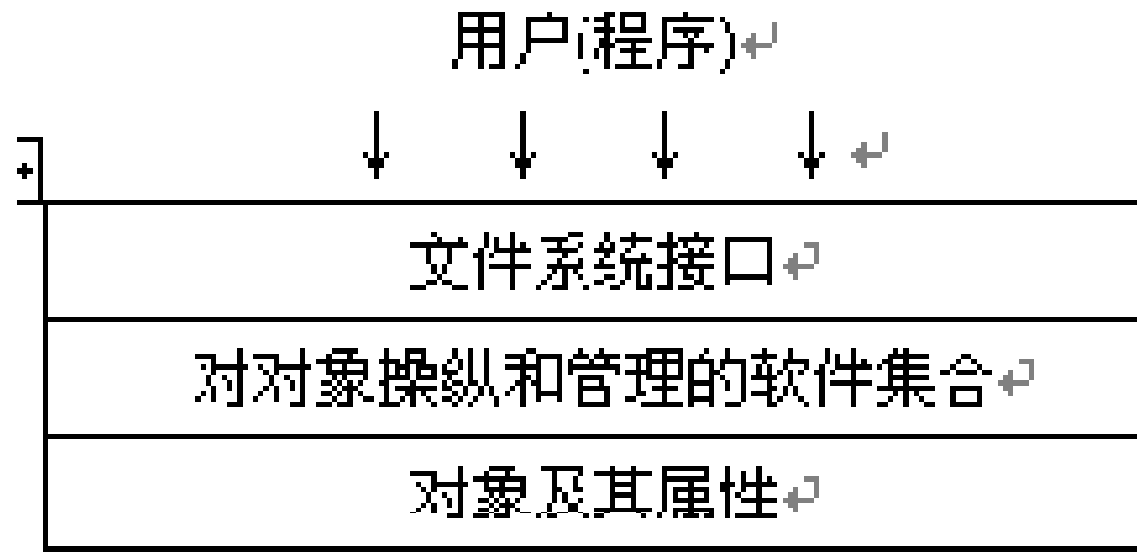


图6-2 文件系统模型





第六章 文件管理



1) 对象及其属性

文件管理系统管理的对象有：① 文件。它作为文件管理的直接对象。② 目录。为了方便用户对文件的存取和检索，在文件系统中必须配置目录，每个目录项中，必须含有文件名及该文件所在的物理地址(或指针)。对目录的组织和管理是方便用户和提高对文件存取速度的关键。③ 磁盘(磁带)存储空间。文件和目录必定占用存储空间，对这部分空间的有效管理，不仅能提高外存的利用率，而且能提高对文件的存取速度。





第六章 文件管理



2) 对对象操纵和管理的软件集合

这是文件管理系统的核心部分。文件系统的功能大多是在这一层实现的，其中包括：对文件存储空间的管理、对文件目录的管理、用于将文件的逻辑地址转换为物理地址的机制、对文件读和写的管理，以及对文件的共享与保护等功能。





第六章 文件管理



3) 文件系统的接口

为方便用户使用文件系统，文件系统通常向用户提供两种类型的接口：

(1) 命令接口。 这是指作为用户与文件系统交互的接口。 用户可通过键盘终端键入命令，取得文件系统的服务。

(2) 程序接口。 这是指作为用户程序与文件系统的接口。 用户程序可通过系统调用来取得文件系统的服务。





6.1.3 文件操作

1. 最基本的文件操作

(1) 创建文件。在创建一个新文件时，系统首先要为新文件分配必要的外存空间，并在文件系统的目录中，为之建立一个目录项。目录项中应记录新文件的文件名及其在外存的地址等属性。

(2) 删除文件。当已不再需要某文件时，可将它从文件系统中删除。在删除时，系统应先从目录中找到要删除文件的目录项，使之成为空项，然后回收该文件所占用的存储空间。





第六章 文件管理



(3) 读文件。在读一个文件时，须在相应系统调用中给出文件名和应读入的内存目标地址。此时，系统同样要查找目录，找到指定的目录项，从中得到被读文件在外存中的位置。在目录项中，还有一个指针用于对文件的读/写。

(4) 写文件。在写一个文件时，须在相应系统调用中给出该文件名及该文件在内存中的(源)地址。为此，也同样须先查找目录，找到指定文件的目录项，再利用目录中的写指针进行写操作。





第六章 文件管理



(5) 截断文件。如果一个文件的内容已经陈旧而需要全部更新时，一种方法是将此文件删除，再重新创建一个新文件。但如果文件名及其属性均无改变时，则可采取另一种所谓的截断文件的方法，此即将原有文件的长度设置为0，或者说是放弃原有的文件内容。

(6) 设置文件的读/写位置。前述的文件读/写操作都只提供了对文件顺序存取的手段，即每次都是从文件的始端读或写。设置文件读/写位置的操作，用于设置文件读/写指针的位置，以便每次读/写文件时，不是从其始端而是从所设置的位置开始操作。也正因如此，才能改顺序存取为随机存取。





2. 文件的“打开”和“关闭”操作

当前OS所提供的大多数对文件的操作，其过程大致都是这样两步：第一步是通过检索文件目录来找到指定文件的属性及其在外存上的位置；第二步是对文件实施相应的操作，如读文件或写文件等。当用户要求对一个文件实施多次读/写或其它操作时，每次都要从检索目录开始。为了避免多次重复地检索目录，在大多数OS中都引入了“打开”(open)这一文件系统调用，当用户第一次请求对某文件进行操作时，先利用open系统调用将该文件打开。





第六章 文件管理



所谓“打开”，是指系统将指名文件的属性(包括该文件在外存上的物理位置)从外存拷贝到内存打开文件表的一个表目中，并将该表目的编号(或称为索引)返回给用户。以后，当用户再要求对该文件进行相应的操作时，便可利用系统所返回的索引号向系统提出操作请求。系统这时便可直接利用该索引号到打开文件表中去查找，从而避免了对该文件的再次检索。这样不仅节省了大量的检索开销，也显著地提高了对文件的操作速度。如果用户已不再需要对该文件实施相应的操作时，可利用“关闭”(close)系统调用来关闭此文件，OS将会把该文件从打开文件表中的表目上删除掉。





3. 其它文件操作

为了方便用户使用文件，通常，OS都提供了数条有关文件操作的系统调用，可将这些调用分成若干类：最常用的一类是有关对文件属性进行操作的，即允许用户直接设置和获得文件的属性，如改变已存文件的文件名、改变文件的拥有者(文件主)、改变对文件的访问权，以及查询文件的状态(包括文件类型、大小和拥有者以及对文件的访问权等)；另一类是有关目录的，如创建一个目录，删除一个目录，改变当前目录和工作目录等；此外，还有用于实现文件共享的系统调用和用于对文件系统进行操作的系统调用等。





6.2 文件的逻辑结构

6.2.1 文件逻辑结构的类型

1. 有结构文件

在记录式文件中，每个记录都用于描述实体集中的一个实体，各记录有着相同或不同数目的数据项。记录的长度可分为定长和不定长两类。

(1) 定长记录。这是指文件中所有记录的长度都是相同的，所有记录中的各数据项都处在记录中相同的位置，具有相同的顺序和长度。文件的长度用记录数目表示。对定长记录的处理方便、开销小，所以这是目前较常用的一种记录格式，被广泛用于数据处理中。





第六章 文件管理



(2) 变长记录。这是指文件中各记录的长度不相同。产生变长记录的原因，可能是由于一个记录中所包含的数据项数目并不相同，如书的著作者、论文中的关键词等；也可能是数据项本身的长度不定，例如，病历记录中的病因、病史；科技情报记录中的摘要等。不论是哪一种，在处理前，每个记录的长度是可知的。

根据用户和系统管理上的需要，可采用多种方式来组织这些记录，形成下述的几种文件：

(1) 顺序文件。这是由一系列记录按某种顺序排列所形成的文件。其中的记录通常是定长记录，因而能用较快的速度查找文件中的记录。





第六章 文件管理



(2) 索引文件。当记录为可变长度时，通常为之建立一张索引表，并为每个记录设置一个表项，以加快对记录检索的速度。

(3) 索引顺序文件。这是上述两种文件构成方式的结合。它为文件建立一张索引表，为每一组记录中的第一个记录设置一个表项。





2. 无结构文件

如果说大量的数据结构和数据库是采用有结构的文件形式的话，则大量的源程序、可执行文件、库函数等，所采用的就是无结构的文件形式，即流式文件。其长度以字节为单位。对流式文件的访问，则是采用读/写指针来指出下一个要访问的字符。可以把流式文件看做是记录式文件的一个特例。在UNIX系统中，所有的文件都被看做是流式文件，即使是有结构文件，也被视为流式文件，系统不对文件进行格式处理。





第六章 文件管理



6.2.2 顺序文件

1. 逻辑记录的排序

文件是记录的集合。文件中的记录可以是任意顺序的，因此，它可以按照各种不同的顺序进行排列。一般地，可归纳为以下两种情况：

第一种是串结构，各记录之间的顺序与关键字无关。通常的办法是由时间来决定，即按存入时间的先后排列，最先存入的记录作为第一个记录，其次存入的为第二个记录.....，依此类推。





第六章 文件管理



第二种情况是顺序结构，指文件中的所有记录按关键字(词)排列。可以按关键词的长短从小到大排序，也可以从大到小排序；或按其英文字母顺序排序。

对顺序结构文件可有更高的检索效率，因为在检索串结构文件时，每次都必须从头开始，逐个记录地查找，直至找到指定的记录，或查完所有的记录为止。而对顺序结构文件，则可利用某种有效的查找算法，如折半查找法、插值查找法、跳步查找法等方法来提高检索效率。





2. 对顺序文件(Sequential File)的读/写操作

顺序文件中的记录可以是定长的，也可以是变长的。对于定长记录的顺序文件，如果已知当前记录的逻辑地址，便很容易确定下一个记录的逻辑地址。在读一个文件时，可设置一个读指针Rptr，令它指向下一个记录的首地址，每当读完一个记录时，便执行

$$Rptr := Rptr + L$$





第六章 文件管理



操作，使之指向下一个记录的首地址，其中的L为记录长度。类似地，在写一个文件时，也应设置一个写指针Wptr，使之指向要写的记录的首地址。同样，在每写完一个记录时，又须执行以下操作：

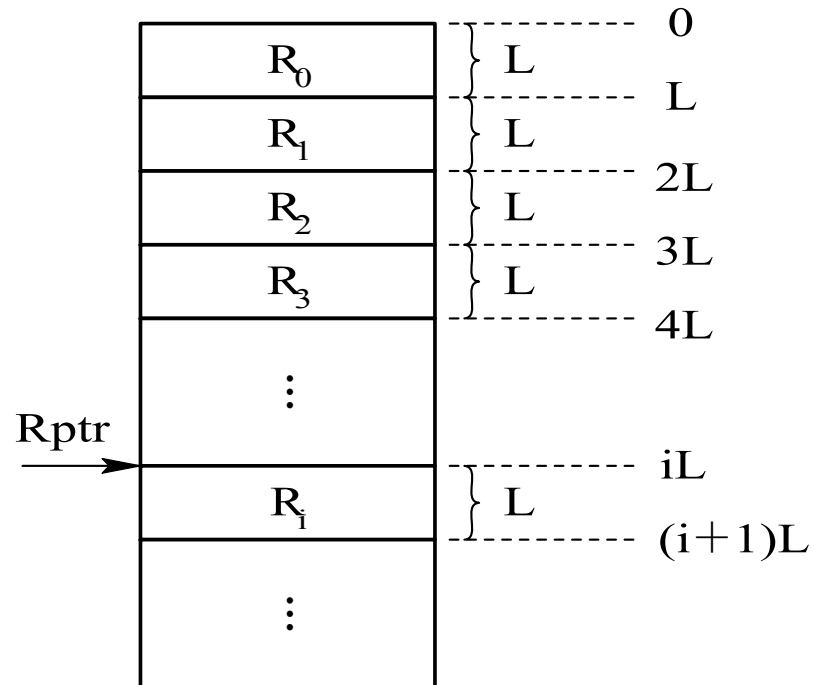
$$Wptr := Wptr + L$$

对于变长记录的顺序文件，在顺序读或写时的情况相似，但应分别为它们设置读或写指针，在每次读或写完一个记录后，须将读或写指针加上Li。Li是刚读或刚写完的记录的长度。图6-3所示为定长和变长记录文件。

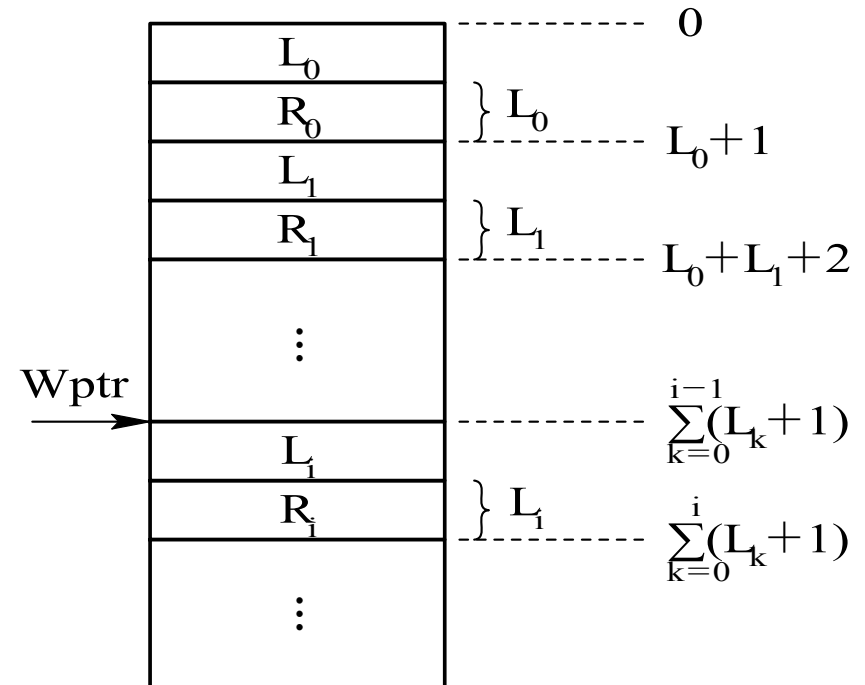




第六章 文件管理



(a) 定长记录文件



(b) 变长记录文件

图6-3 定长和变长记录文件





3. 顺序文件的优缺点

顺序文件的最佳应用场合是在对诸记录进行批量存取时，即每次要读或写一大批记录时。此时，对顺序文件的存取效率是所有逻辑文件中最高的；此外，也只有顺序文件才能存储在磁带上，并能有效地工作。





第六章 文件管理



在交互应用的场合，如果用户(程序)要求查找或修改单个记录，为此系统便要去逐个地查找诸记录。这时，顺序文件所表现出来的性能就可能很差，尤其是当文件较大时，情况更为严重。例如，有一个含有 10^4 个记录的顺序文件，如果对它采用顺序查找法去查找一个指定的记录，则平均需要查找 5×10^3 个记录；如果是可变长记录的顺序文件，则为查找一个记录所需付出的开销将更大，这就限制了顺序文件的长度。





第六章 文件管理



顺序文件的另一个缺点是，如果想增加或删除一个记录都比较困难。为了解决这一问题，可以为顺序文件配置一个运行记录文件(Log File)，或称为事务文件(Transaction File)，把试图增加、删除或修改的信息记录于其中，规定每隔一定时间，例如4小时，将运行记录文件与原来的主文件加以合并，产生一个按关键字排序的新文件。





第六章 文件管理



6.2.3 索引文件

对于定长记录文件，如果要查找第*i*个记录，可直接根据下式计算来获得第*i*个记录相对于第一个记录首址的地址：

$$A_i = i \times L$$

然而，对于可变长度记录的文件，要查找其第*i*个记录时，须首先计算出该记录的首地址。为此，须顺序地查找每个记录，从中获得相应记录的长度 L_i ，然后才能按下式计算出第*i*个记录的首址。假定在每个记录前用一个字节指明该记录的长度，则

$$A_i = \sum_{i=0}^{i-1} L_i + i$$





第六章 文件管理



可见，对于定长记录，除了可以方便地实现顺序存取外，还可较方便地实现直接存取。然而，对于变长记录就较难实现直接存取了，因为用直接存取方法来访问变长记录文件中的一个记录是十分低效的，其检索时间也很难令人接受。为了解决这一问题，可为变长记录文件建立一张索引表，对主文件中的每个记录，在索引表中设有一个相应的表项，用于记录该记录的长度 L 及指向该记录的指针(指向该记录在逻辑地址空间的首址)。由于索引表是按记录键排序的，因此，索引表本身是一个定长记录的顺序文件，从而也就可以方便地实现直接存取。图6-4示出了索引文件(Index File)的组织形式。





第六章 文件管理

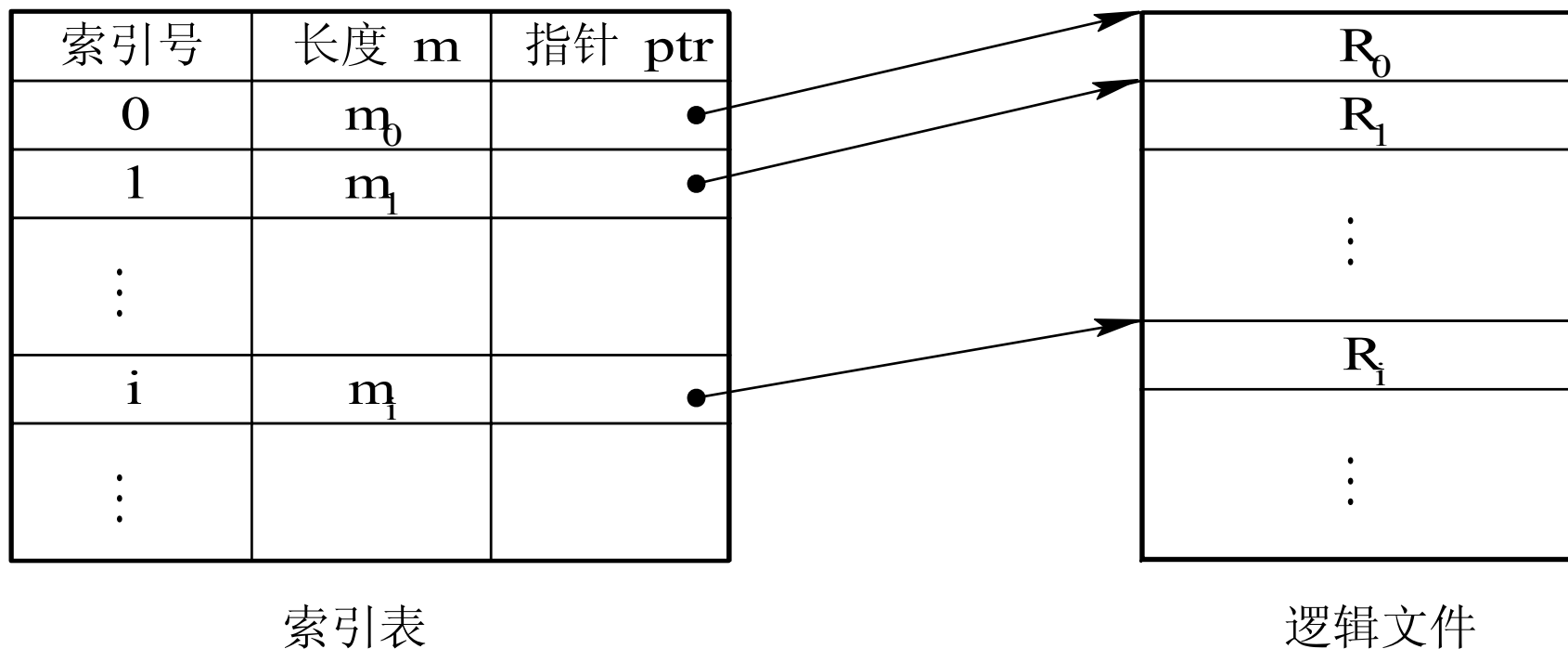


图6-4 索引文件的组织





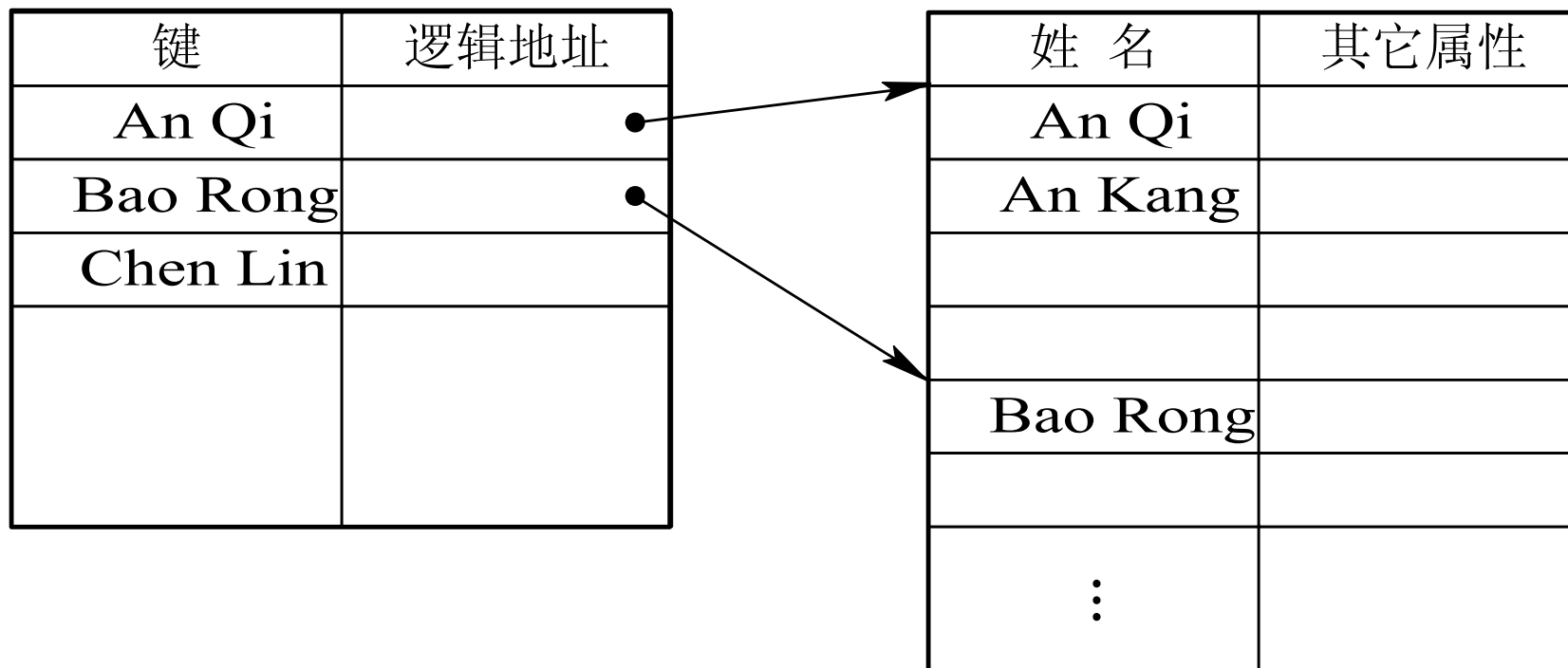
6.2.4 索引顺序文件

索引顺序文件(Index Sequential File)可能是最常见的一种逻辑文件形式。它有效地克服了变长记录文件不便于直接存取的缺点，而且所付出的代价也不算太大。前已述及，它是顺序文件和索引文件相结合的产物。它将顺序文件中的所有记录分为若干个组(例如，50个记录为一个组)；为顺序文件建立一张索引表，在索引表中为每组中的第一个记录建立一个索引项，其中含有该记录的键值和指向该记录的指针。索引顺序文件如图6-5所示。





第六章 文件管理



逻辑文件

图6-5 索引顺序文件





6.2.5 直接文件和哈希文件

1. 直接文件

采用前述几种文件结构对记录进行存取时，都须利用给定的记录键值，先对线性表或链表进行检索，以找到指定记录的物理地址。然而对于直接文件，则可根据给定的记录键值，直接获得指定记录的物理地址。换言之，记录键值本身就决定了记录的物理地址。这种由记录键值到记录物理地址的转换被称为键值转换(Key to address transformation)。组织直接文件的关键，在于用什么方法进行从记录值到物理地址的转换。





2. 哈希(Hash)文件

这是目前应用最为广泛的一种直接文件。它利用Hash函数(或称散列函数), 可将记录键值转换为相应记录的地址。但为了能实现文件存储空间的动态分配, 通常由Hash函数所求得的并非是相应记录的地址, 而是指向一目录表相应表目的指针, 该表目的内容指向相应记录所在的物理块, 如图6-6所示。例如, 若令K为记录键值, 用A作为通过Hash函数H的转换所形成的该记录在目录表中对应表目的位置, 则有关系 $A=H(K)$ 。通常, 把Hash函数作为标准函数存于系统中, 供存取文件时调用。





第六章 文件管理

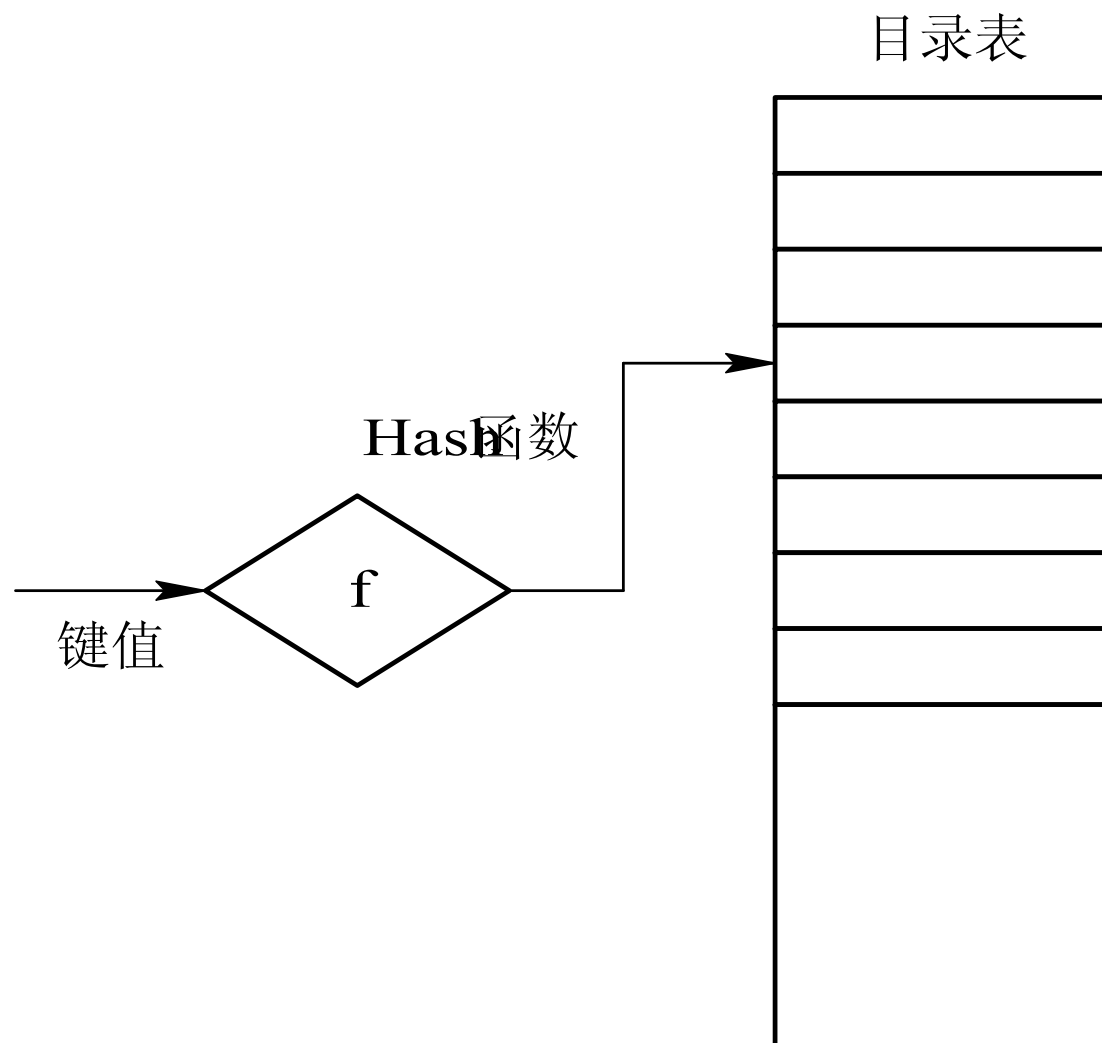
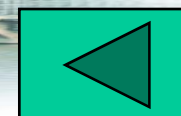


图6-6 Hash文件的逻辑结构





6.3 外存分配方式

6.3.1 连续分配

1. 连续分配方式

连续分配(Continuous Allocation)要求为每一个文件分配一组相邻接的盘块。一组盘块的地址定义了磁盘上的一段线性地址。例如, 第一个盘块的地址为 b , 则第二个盘块的地址为 $b+1$, 第三个盘块的地址为 $b+2$。通常, 它们都位于一条磁道上, 在进行读/写时, 不必移动磁头, 仅当访问到一条磁道的最后一个盘块后, 才需要移到下一条磁道, 于是又去连续地读/写多个盘块。





第六章 文件管理

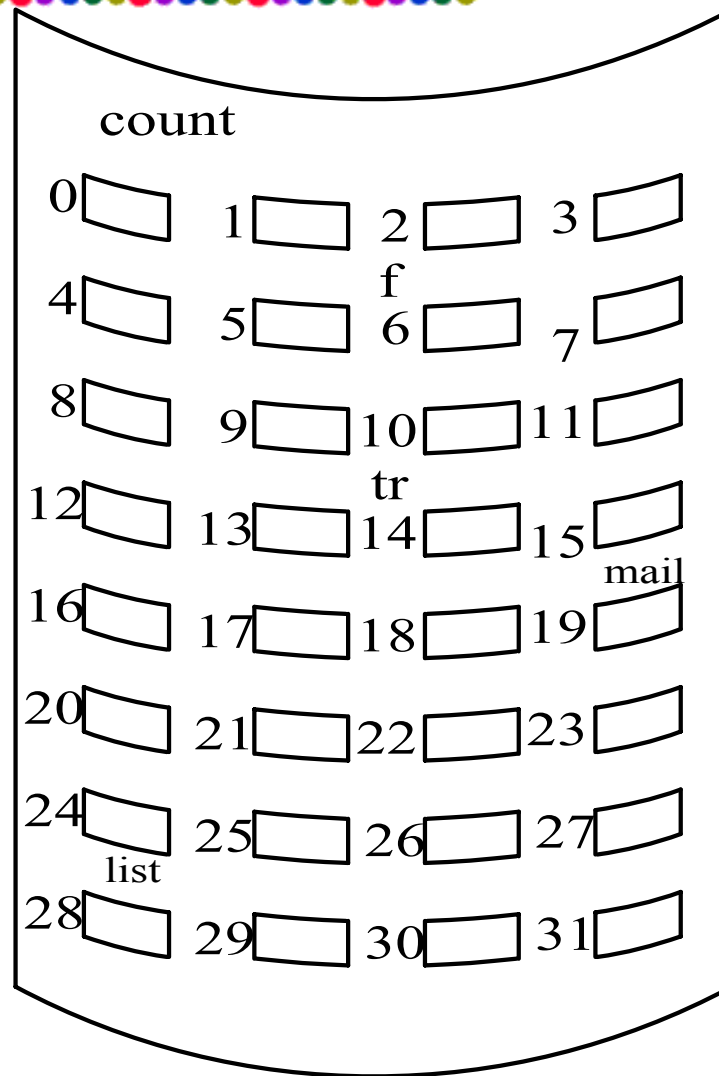


在采用连续分配方式时，可把逻辑文件中的记录顺序地存储到邻接的各物理盘块中，这样所形成的文件结构称为顺序文件结构，此时的物理文件称为顺序文件。这种分配方式保证了逻辑文件中的记录顺序与存储器中文件占用盘块的顺序的一致性。为使系统能找到文件存放的地址，应在目录项的“文件物理地址”字段中，记录该文件第一个记录所在的盘块号和文件长度(以盘块数进行计量)。图6-7 示出了连续分配的情况。图中假定了记录与盘块的大小相同。Count文件的第一个盘块号是0，文件长度为2，因此是在盘块号为0和1的两盘块中存放文件1的数据。





第六章 文件管理



目录

file	start	length
count	0	2
tr	15	3
mail	21	6
list	29	3
f	7	2

图6-7 磁盘空间的连续分配





第六章 文件管理



如同内存的动态分区分配一样，随着文件建立时空间的分配和文件删除时空间的回收，将使磁盘空间被分割成许多小块，这些较小的连续区已难于用来存储文件，此即外存的碎片。同样，我们也可以利用紧凑的方法，将盘上所有的文件紧靠在一起，把所有的碎片拼接成一大片连续的存储空间。例如，可以运行一个再装配例程(repack routine)，由它将磁盘A上的大量文件拷贝到一张软盘B或几张软盘(C, D, ...)上，并释放原来的A盘，使之成为一个空闲盘。然后再将软盘B(C, D, ...)上的文件拷回A盘上。这种方法能将含有多个文件的盘上的所有空闲盘块都集中在一起，从而消除了外部碎片。但为了将外存上的空闲空间进行一次紧凑，所花费的时间远比将内存紧凑一次所花费的时间多得多。





2. 连续分配的主要优缺点

连续分配的主要优点如下：

(1) 顺序访问容易。访问一个占有连续空间的文件非常容易。系统可从目录中找到该顺序文件所在的第一个盘块号，从此开始顺序地、逐个盘块地往下读/写。连续分配也支持直接存取。例如，要访问一个从 b 块开始存放的文件中的第 i 个盘块的内容，就可直接访问 $b+i$ 号盘块。

(2) 顺序访问速度快。因为由连续分配所装入的文件，其所占用的盘块可能是位于一条或几条相邻的磁道上，这时，磁头的移动距离最少，因此，这种对文件访问的速度是几种存储空间分配方式中最高的一种。





第六章 文件管理



连续分配的主要缺点如下：

(1) 要求有连续的存储空间。要为每一个文件分配一段连续的存储空间，这样，便会产生出许多外部碎片，严重地降低了外存空间的利用率。如果是定期地利用紧凑方法来消除碎片，则又需花费大量的机器时间。





第六章 文件管理



(2) 必须事先知道文件的长度。要将一个文件装入一个连续的存储区中，必须事先知道文件的大小，然后根据其大小，在存储空间中找出一块其大小足够的存储区，将文件装入。在有些情况下，知道文件的大小是件非常容易的事，如可拷贝一个已存文件。但有时却很难，在此情况下，只能靠估算。如果估计的文件大小比实际文件小，就可能因存储空间不足而中止文件的拷贝，须再要求用户重新估算，然后再次执行。这样，显然既费时又麻烦。这就促使用户往往将文件长度估得比实际的大，甚至使所计算的文件长度比实际长度大得多，显然，这会严重地浪费外存空间。对于那些动态增长的文件，由于开始时文件很小，在运行中逐渐增大，比如，这种增长要经历几天、几个月。在此情况下，即使事先知道文件的最终大小，在采用预分配存储空间的方法时，显然也将是很低效的，即它使大量的存储空间长期地空闲着。





6.3.2 链接分配

1. 隐式链接

在采用隐式链接分配方式时，在文件目录的每个目录项中，都须含有指向链接文件第一个盘块和最后一个盘块的指针。图6-8中示出了一个占用5个盘块的链接式文件。在相应的目录项中，指示了其第一个盘块号是9，最后一个盘块号是25。而在每个盘块中都含有一个指向下一个盘块的指针，如在第一个盘块9中设置了第二个盘块的盘块号是16；在16号盘块中又设置了第三个盘块的盘块号1。如果指针占用4个字节，对于盘块大小为512字节的磁盘，则每个盘块中只有508个字节可供用户使用。





第六章 文件管理

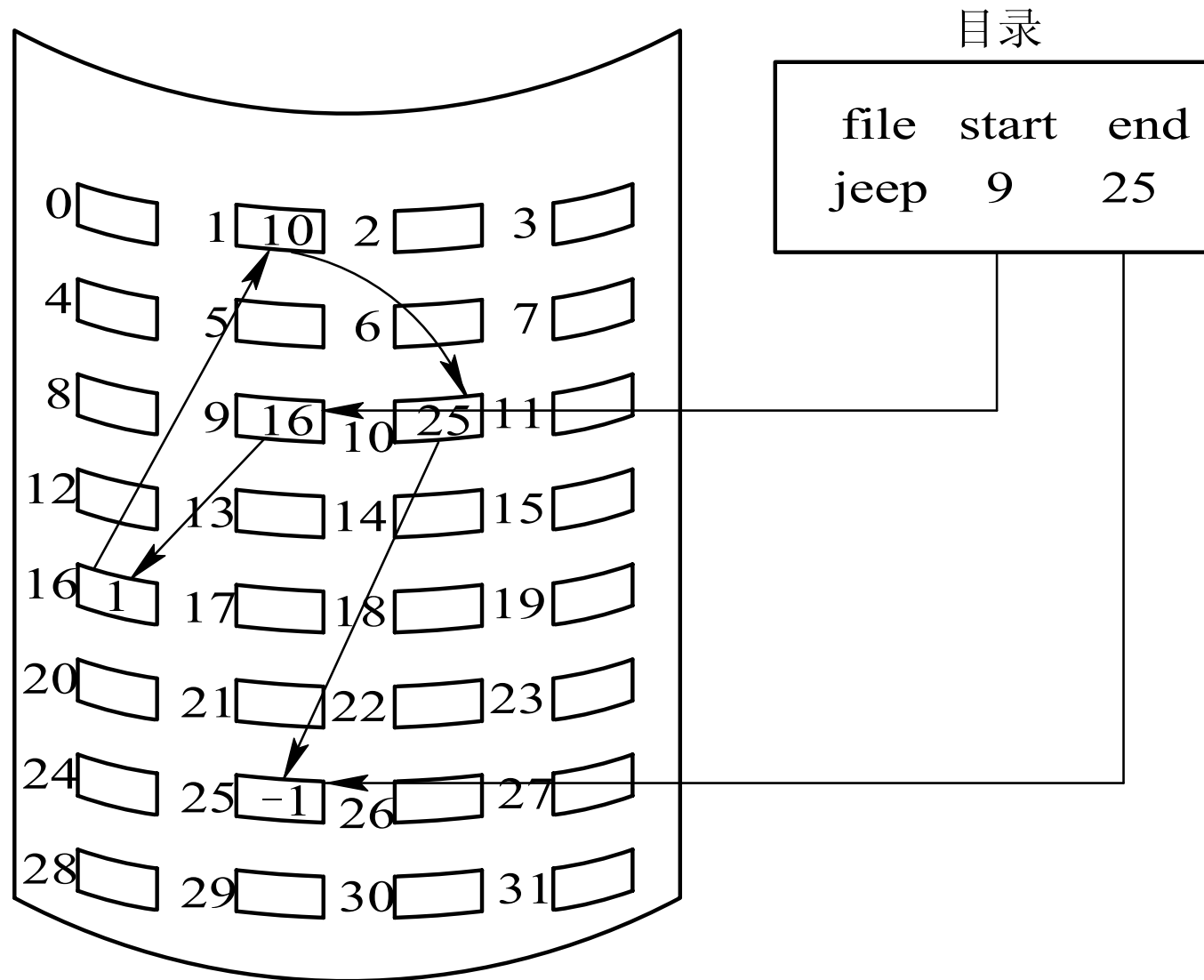


图6-8 磁盘空间的链接式分配



第六章 文件管理



隐式链接分配方式的主要问题在于：它只适合于顺序访问，它对随机访问是极其低效的。如果要访问文件所在的第 i 个盘块，则必须先读出文件的第一个盘块.....，就这样顺序地查找直至第 i 块。当 $i=100$ 时，须启动100次磁盘去实现读盘块的操作，平均每次都要花费几十毫秒。可见，随机访问的速度相当低。此外，只通过链接指针来将一大批离散的盘块链接起来，其可靠性较差，因为只要其中的任何一个指针出现问题，都会导致整个链的断开。





2. 显式链接

这是指把用于链接文件各物理块的指针，显式地存放在内存的一张链接表中。该表在整个磁盘仅设置一张，如图6-9所示。表的序号是物理盘块号，从0开始，直至N-1；N为盘块总数。在每个表项中存放链接指针，即下一个盘块号。在该表中，凡是属于某一文件的第一个盘块号，或者说是每一条链的链首指针所对应的盘块号，均作为文件地址被填入相应文件的FCB的“物理地址”字段中。由于查找记录的过程是在内存中进行的，因而不仅显著地提高了检索速度，而且大大减少了访问磁盘的次数。由于分配给文件的所有盘块号都放在该表中，故把该表称为文件分配表FAT(File Allocation Table)。





第六章 文件管理

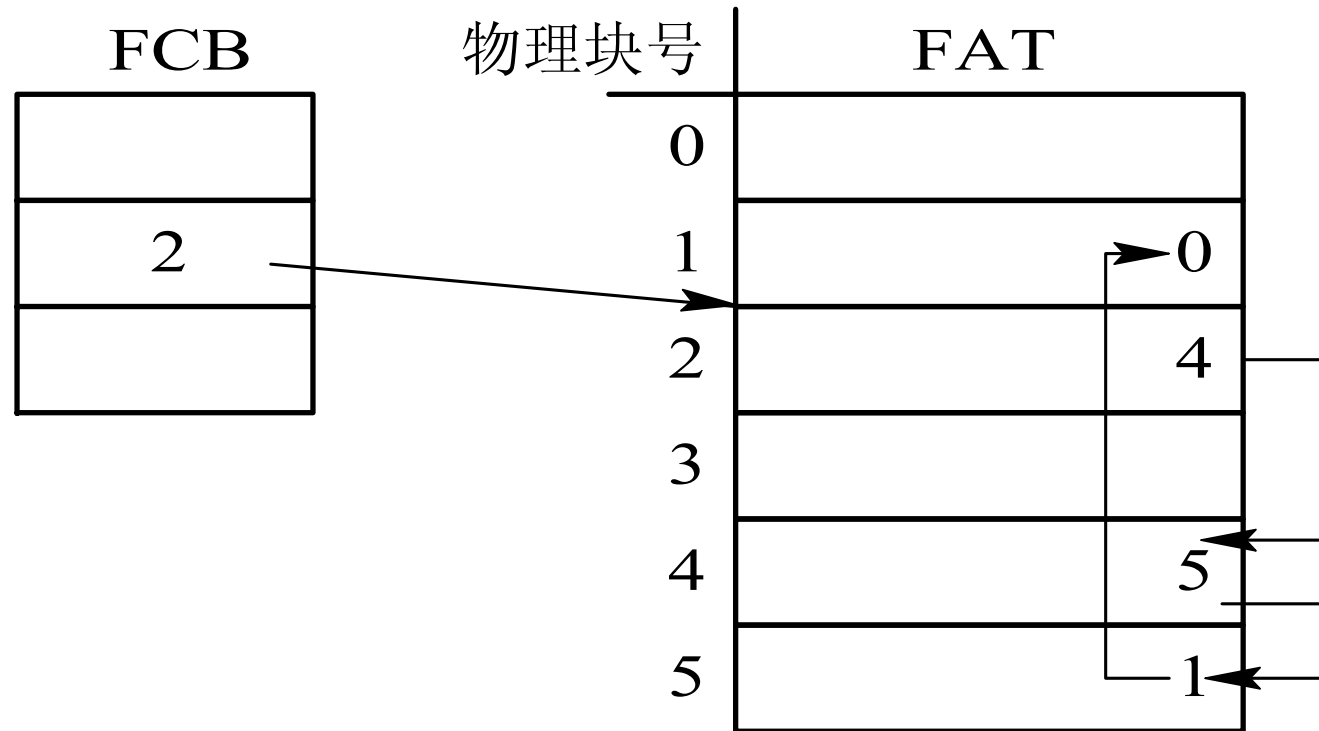


图6-9 显式链接结构





6.3.3 FAT和NTFS技术

1. FAT12

1) 以盘块为基本分配单位

早期MS-DOS操作系统所使用的是FAT12文件系统，在每个分区中都配有两张文件分配表FAT1和FAT2，在FAT的每个表项中存放下一个盘块号，它实际上是用于盘块之间的链接的指针，通过它可以将一个文件的所有的盘块链接起来，而将文件的第一个盘块号放在自己的FCB中。





第六章 文件管理



图6-10示出了MS-DOS的文件物理结构，这里示出了两个文件，文件A占用三个盘块，其盘块号依次为4、6、11；文件B则依次占用9、10及5号三个盘块。每个文件的第一个盘块号放在自己的FCB中。整个系统有一张文件分配表FAT。在FAT的每个表项中存放下一个盘块号。对于1.2 MB的软盘，每个盘块的大小为512 B，在每个FAT中共含有2.4 K个表项，由于每个FAT表项占12位，故FAT表占用3.6 KB的存储空间。





第六章 文件管理

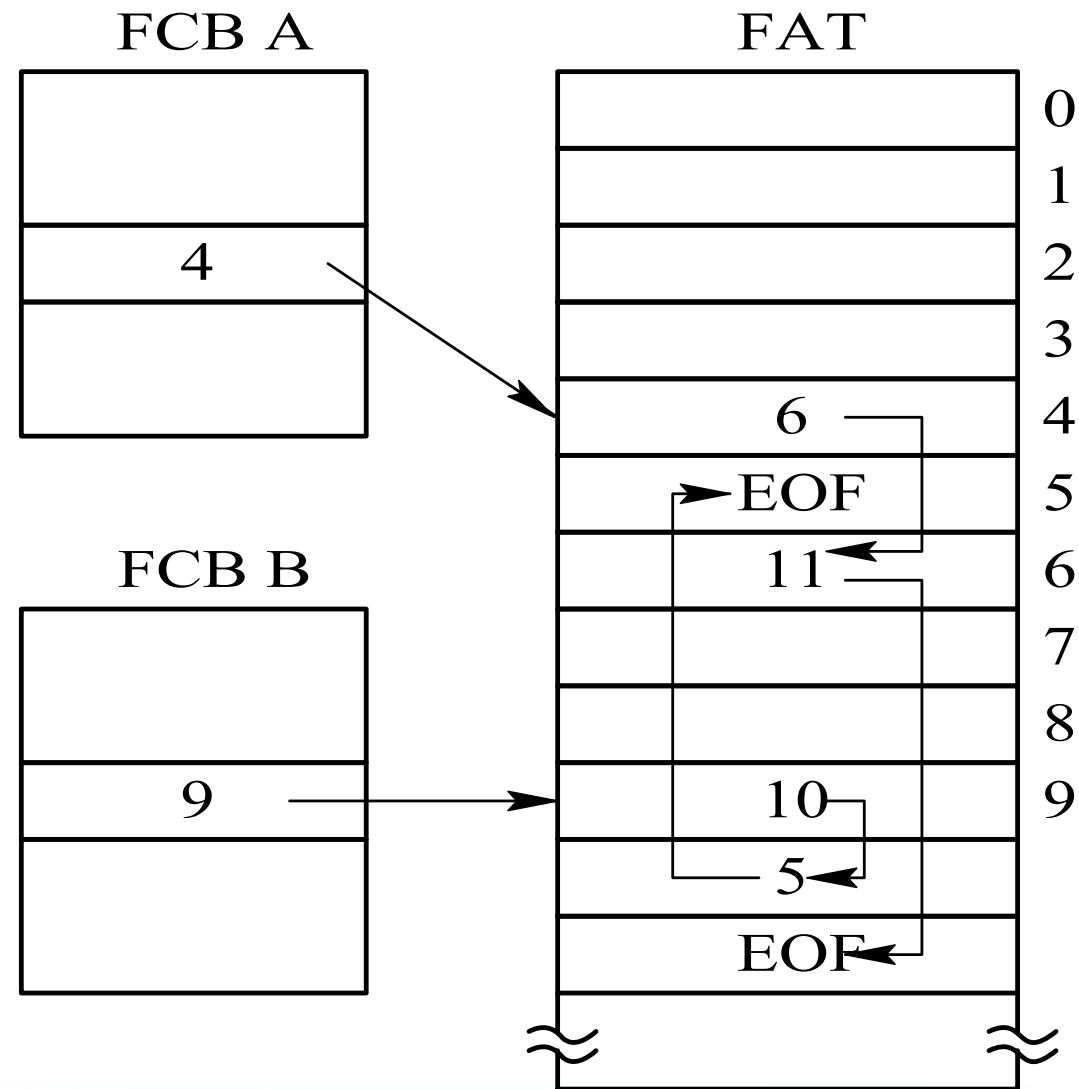


图6-10 MS-DOS的文件物理结构



第六章 文件管理



现在我们来计算以盘块为分配单位时，所允许的最大磁盘容量。由于每个FAT表项为12位，因此，在FAT表中最多允许有4096个表项，如果采用以盘块作为基本分配单位，每个盘块(也称扇区)的大小一般是512字节，那么，每个磁盘分区的容量为2 MB(4096×512 B)。同时，一个物理磁盘支持4个逻辑磁盘分区，所以相应的磁盘最大容量仅为8 MB。这对最早时期的硬盘还可应付，但很快磁盘的容量就超过了8 MB，FAT12是否还可继续用呢，回答虽是肯定的，但需要引入一个新的分配单位——簇。





第六章 文件管理



2) 簇的基本概念

为了适应磁盘容量不断增大的需要，在进行盘块分配时，不再以盘块而是以簇(cluster)为基本单位。簇是一组连续的扇区，在FAT中它是作为一个虚拟扇区，簇的大小一般是 2^n (n 为整数)个盘块，在MS-DOS的实际运用中，簇的容量可以仅有一个扇区(512 B)、两个扇区(1 KB)、四个扇区(2 KB)、八个扇区(4 KB)等。一个簇应包含扇区的数量与磁盘容量的大小直接有关。例如，当一个簇仅有一个扇区时，磁盘的最大容量为8 MB；当一个簇包含两个扇区时，磁盘的最大容量可以达到16 MB；当一个簇包含了八个扇区时，磁盘的最大容量便可达到64 MB。





第六章 文件管理



由上所述可以看出，以簇作为基本的分配单位所带来的最主要的好处是，能适应磁盘容量不断增大的情况。值得注意的是，使用簇作为基本的分配单位虽可减少FAT表中的项数(在相同的磁盘容量下，FAT表的项数是与簇的大小成反比的)。这一方面会使FAT表占用更少的存储空间，并减少访问FAT表的存取开销，提高文件系统的效率；但这也会造成更大的簇内零头(它与存储器管理中的页内零头相似)。





第六章 文件管理



3) FAT12存在的问题

尽管FAT12曾是一个不错的文件系统，但毕竟已老化，已不能满足操作系统发展的需要，其表现出来的主要问题是，对所允许的磁盘容量存在着严重的限制，通常只能是数十兆字节，虽然可以用继续增加簇的大小来提高所允许的最大磁盘容量，但随着支持的硬盘容量的增加，相应的簇内碎片也将随之成倍地增加。此外，它只能支持8+3格式的文件名。





2. FAT16

对FAT12所存在的问题进行简单的分析即可看出，其根本原因在于，FAT12表最多只允许4096个表项，亦即最多只能将一个磁盘分区分为4096个簇。这样，随着磁盘容量的增加，必定会引起簇的大小和簇内碎片也随之增加。由此可以得出解决方法，那就是增加FAT表的表项数，亦即应增加FAT表的宽度，如果我们将FAT表的宽度增至16位，最大表项数将增至65536个，此时便能将一个磁盘分区分为65536(2¹⁶)个簇。我们把具有16位表宽的FAT表称为FAT16。在FAT16的每个簇中可以有的盘块数为4、8、16、32直到64，由此得出FAT16可以管理的最大分区空间为 $2^{16} \times 64 \times 512 = 2048 \text{ MB}$ 。





第六章 文件管理



由上述分析不难看出，FAT16对FAT12的局限性有所改善，但改善很有限。当磁盘容量迅速增加时，如果再继续使用FAT16，由此所形成的簇内碎片所造成的浪费也越大。例如，当要求磁盘分区的大小为8 GB时，则每个簇的大小达到128 KB，这意味着内部零头最大可达到128 KB。一般而言，对于1~4 GB的硬盘来说，大约会浪费10%~20%的空间。为了解决这一问题，微软推出了FAT32。





3. FAT32

如同存储器管理中的分页管理，所选择的页面越大，可能造成的页内零头也会越大。为减少页内零头就应该选择适当大小的页面。在这里，为了减小磁盘的簇内零头，也就应当选择适当大小的簇。问题是FAT16表的长度只有65 535项，随着磁盘容量的增加，簇的大小也必然会随之增加，为了减少簇内零头，也就应当增加FAT表的长度。为此，需要再增加FAT表的宽度，这样也就由FAT16演变为FAT32。





第六章 文件管理



FAT32是FAT系列文件系统的最后一个产品。每一簇在FAT表中的表项占据4字节(2³²)，FAT表可以表示4 294 967 296项，即FAT32允许管理比FAT16更多的簇。这样就允许在FAT32中采用较小的簇，FAT32的每个簇都固定为4 KB，即每簇用8个盘块代替FAT16的64个盘块，每个盘块仍为512字节，FAT32分区格式可以管理的单个最大磁盘空间大到4 KB×2³² = 2 TB。三种FAT类型的最大分区以及所对应的块的大小如图6-11所示。





第六章 文件管理



块大小/KB	FAT12/MB	FAT16/MB	FAT32/TB
0.5	2		
1	4		
2	8	128	
4	16	256	1
8		512	2
16		1024	2
32		2048	2

图6-11 FAT中簇的大小与最大分区的对应关系





第六章 文件管理



4. NTFS

1) NTFS新特征

NTFS(New Technology File System)是一个专门为Windows NT开发的、全新的文件系统，并适用于Windows 2000/XP/2003。NTFS具有许多新的特征：首先，它使用了64位磁盘地址，理论上可以支持 2^{64} 次方字节的磁盘分区；其次，在NTFS中可以很好地支持长文件名，单个文件名限制在255个字符以内，全路径名为32 767个字符；第三，具有系统容错功能，即在系统出现故障或差错时，仍能保证系统正常运行，这一点我们将在6.6节中介绍；第四，提供了数据的一致性，这是一个非常有用的功能，我们将在本章的最后一节介绍；此外，NTFS还提供了文件加密、文件压缩等功能。





第六章 文件管理



2) 磁盘组织

同FAT文件系统一样，NTFS也是以簇作为磁盘空间分配和回收的基本单位。一个文件占用若干个簇，一个簇只属于一个文件。通过簇来间接管理磁盘，可以不需要知道盘块(扇区)的大小，使NTFS具有了与磁盘物理扇区大小无关的独立性，很容易支持扇区大小不是512字节的非标准磁盘，从而可以根据不同的磁盘选择匹配的簇大小。





第六章 文件管理



在NTFS文件系统中，把卷上簇的大小称为“卷因子”，卷因子是在磁盘格式化时确定的，其大小同FAT一样，也是物理磁盘扇区的整数倍，即一个簇包含 2^n (n 为整数)个盘块，簇的大小可由格式化命令或格式化程序按磁盘容量和应用需求来确定，可以为512 B、1 KB、2 KB……，最大可达64 KB。对于小磁盘(≤ 512 MB)，默认簇大小为512字节；对于1 GB磁盘，默认簇大小为1 KB；对于2 GB磁盘，则默认簇大小为4 KB。事实上，为了在传输效率和簇内碎片之间进行折中，NTFS在大多数情况下都是使用4 KB。





第六章 文件管理



对于簇的定位，NTFS是采用逻辑簇号LCN(Logical Cluster Number)和虚拟簇号VCN(Virtual Cluster Number)进行的。LCN是以卷为单位，将整个卷中所有的簇按顺序进行简单的编号，NTFS在进行地址映射时，可以通过卷因子与LCN的乘积，便可算出卷上的物理字节偏移量，从而得到文件数据所在的物理磁盘地址。为了方便文件中数据的引用，NTFS还可以使用VCN，以文件为单位，将属于某个文件的簇按顺序进行编号。只要知道了文件开始的簇地址，便可将VCN映射到LCN。





第六章 文件管理



3) 文件的组织

在NTFS中，以卷为单位，将一个卷中的所有文件信息、目录信息以及可用的未分配空间信息，都以文件记录的方式记录在一张主控文件表MFT(Master File Table)中。该表是NTFS 卷结构的中心，从逻辑上讲，卷中的每个文件作为一条记录，在MFT 表中占有一行，其中还包括MFT 自己的这一行。每行大小固定为1 KB，每行称为该行所对应文件的元数据(metadata)，也称为文件控制字。





第六章 文件管理



在MFT表中，每个元数据将其所对应文件的所有信息，包括文件的内容等，都被组织在所对应文件的一组属性中。由于文件大小相差悬殊，其属性所需空间大小也相差很大，因此，在MFT表中，对于元数据的1 KB空间，可能记录不下文件的全部信息。所以当文件较小时，其属性值所占空间也较小，可以将文件的所有属性直接记录在元数据中。而当文件较大时，元数据仅记录该文件的一部分属性，其余属性，如文件的内容等，可以记录到卷中的其它可用簇中，并将这些簇按其所记录文件的属性进行分类，分别链接成多个队列，将指向这些队列的指针保存在元数据中。





6.3.4 索引分配

1. 单级索引分配

链接分配方式虽然解决了连续分配方式所存在的问题，但又出现了下述另外两个问题：

(1) 不能支持高效的直接存取。要对于一个较大的文件进行直接存取，须首先在FAT中顺序地查找许多盘块号。





第六章 文件管理



(2) FAT需占用较大的内存空间。由于一个文件所占用盘块的盘块号是随机地分布在FAT中的，因而只有将整个FAT调入内存，才能保证在FAT中找到一个文件的所有盘块号。当磁盘容量较大时，FAT可能要占用数兆字节以上的内存空间，这是令人难以接受的。

事实上，在打开某个文件时，只需把该文件占用的盘块的编号调入内存即可，完全没有必要将整个FAT调入内存。为此，应将每个文件所对应的盘块号集中地放在一起。索引分配方法就是基于这种想法所形成的一种分配方法。它为每个文件分配一个索引块(表)，再把分配给该文件的所有盘块号都记录在该索引块中，因而该索引块就是一个含有许多盘块号的数组。在建立一个文件时，只需在为之建立的目录项中填上指向该索引块的指针。图6-12 示出了磁盘空间的索引分配图。





第六章 文件管理

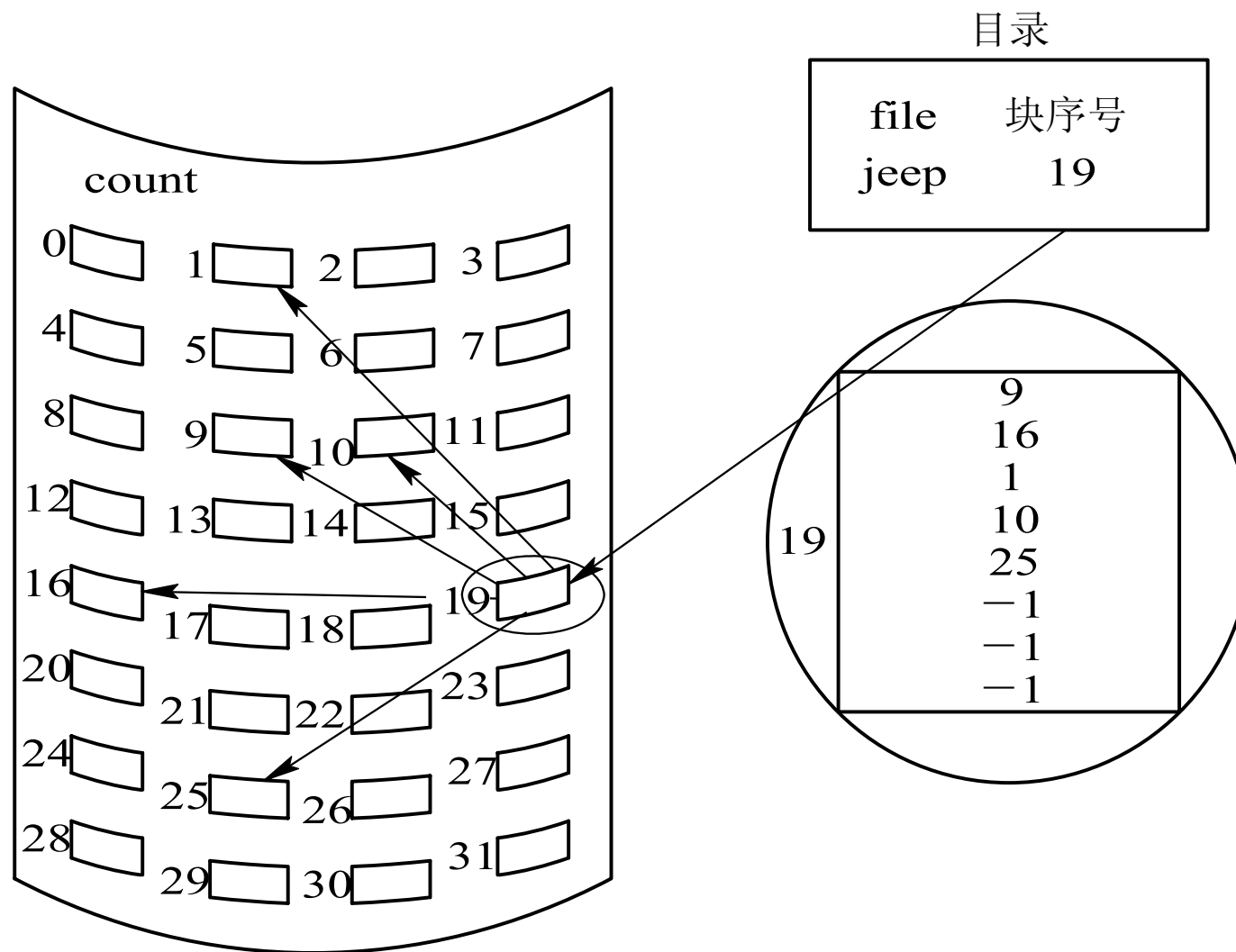


图6-12 索引分配方式





2. 多级索引分配

当OS为一个文件分配磁盘空间时，如果所分配出去的盘块的盘块号已经装满一个索引块时，OS便为该文件分配另一个索引块，用于将以后继续为之分配的盘块号记录于其中。依此类推，再通过链指针将各索引块按序链接起来。显然，当文件太大，其索引块太多时，这种方法是低效的。此时，应为这些索引块再建立一级索引，称为第一级索引，即系统再分配一个索引块，作为第一级索引的索引块，将第一块、第二块……等索引块的盘块号填入到此索引表中，这样便形成了两级索引分配方式。如果文件非常大时，还可用三级、四级索引分配方式。





第六章 文件管理



图6-13示出了两级索引分配方式下各索引块之间的链接情况。如果每个盘块的大小为1 KB，每个盘块号占4个字节，则在一个索引块中可存放256个盘块号。这样，在两级索引时，最多可包含的存放文件的盘块的盘块号总数 $N = 256 \times 256 = 64 \text{ K}$ 个盘块号。由此可得出结论：采用两级索引时，所允许的文件最大长度为64 MB。倘若盘块的大小为4 KB，在采用单级索引时所允许的最大文件长度为4 MB；而在采用两级索引时所允许的最大文件长度可达4 GB。





第六章 文件管理

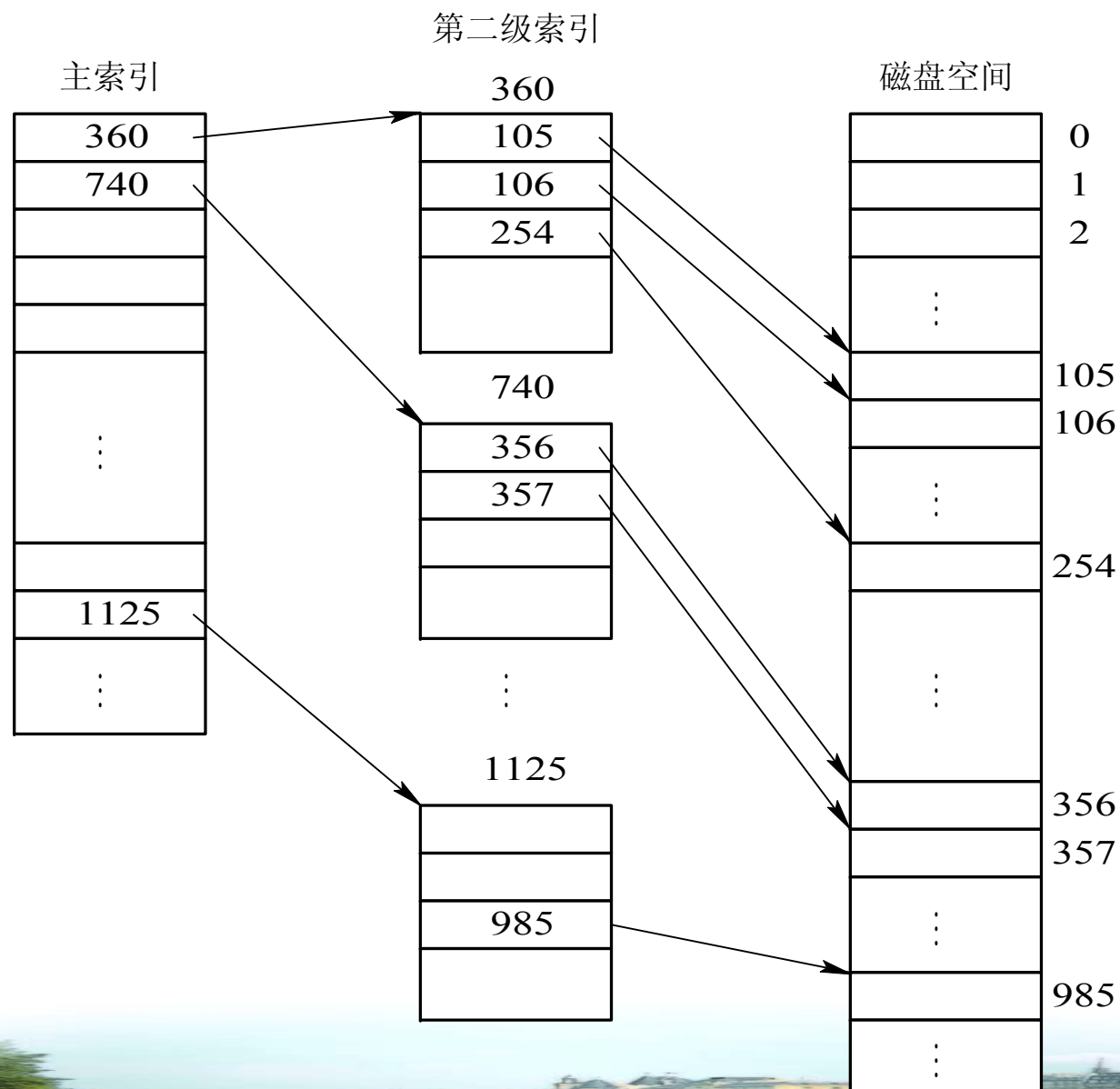


图6-13 两级索引分配



3. 混合索引分配方式

所谓混合索引分配方式，是指将多种索引分配方式相结合而形成的一种分配方式。例如，系统既采用了直接地址，又采用了一级索引分配方式，或两级索引分配方式，甚至还采用了三级索引分配方式。这种混合索引分配方式已在UNIX系统中采用。在UNIX System V的索引结点中，共设置了13个地址项，即 $iaddr(0) \sim iaddr(12)$ ，如图6-14所示。在BSD UNIX的索引结点中，共设置了13个地址项，它们都把所有的地址项分成两类，即直接地址和间接地址。





第六章 文件管理

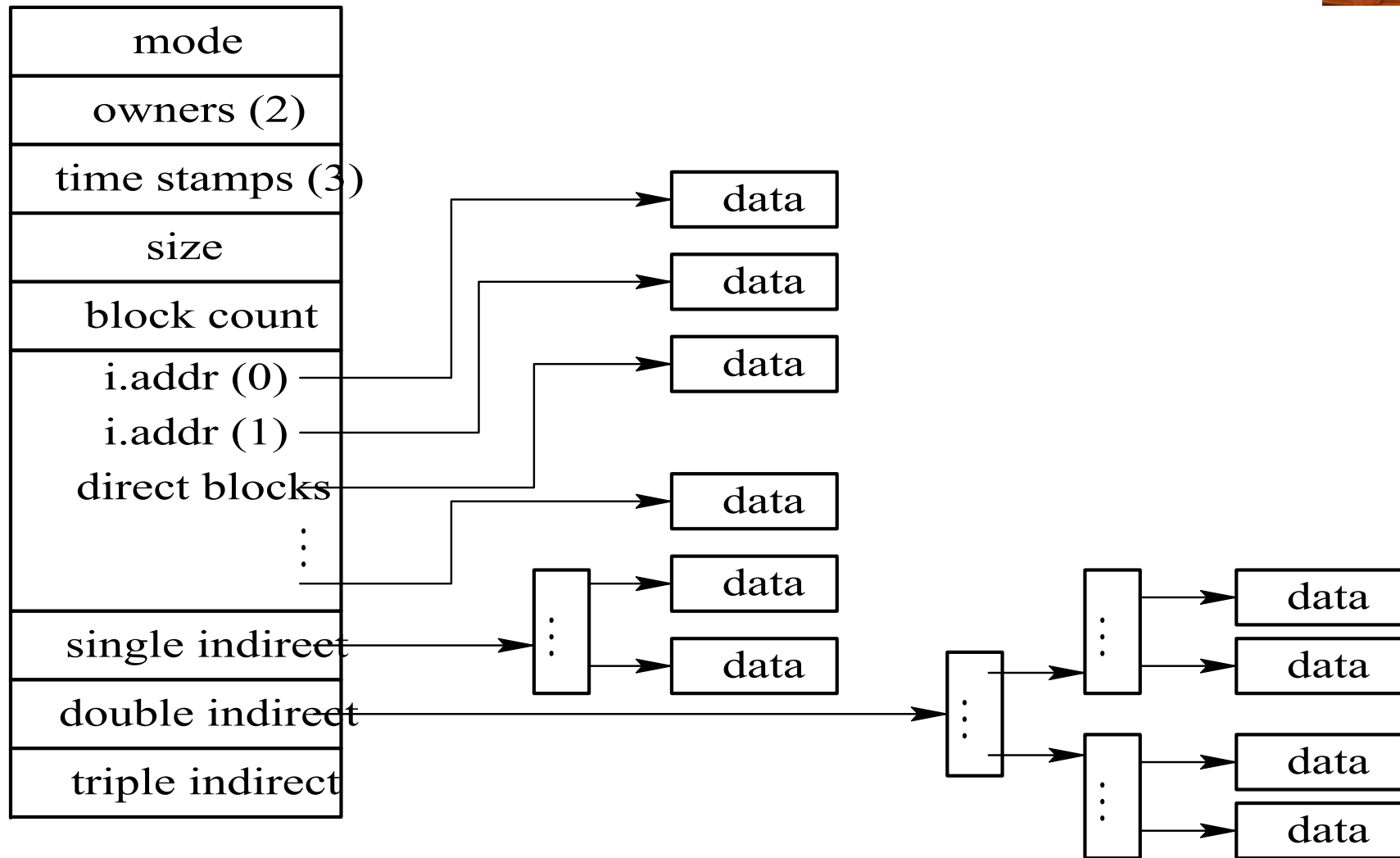


图6-14 混合索引方式



第六章 文件管理



1) 直接地址

为了提高对文件的检索速度，在索引结点中可设置10个直接地址项，即用*iaddr*(0)~*iaddr*(9)来存放直接地址。换言之，在这里的每项中所存放的是该文件数据所在盘块的盘块号。假如每个盘块的大小为 4 KB，当文件不大于40 KB时，便可直接从索引结点中读出该文件的全部盘块号。





第六章 文件管理



2) 一次间接地址

对于大、中型文件，只采用直接地址是不现实的。为此，可再利用索引结点中的地址项*iaddr*(10)来提供一次间接地址。这种方式的实质就是一级索引分配方式。图中的一次间址块也就是索引块，系统将分配给文件的多个盘块号记入其中。在一次间址块中可存放1 K个盘块号，因而允许文件长达4 MB。





3) 多次间接地址

当文件长度大于 $4\text{ MB} + 40\text{ KB}$ 时(一次间址与10个直接地址项), 系统还须采用二次间址分配方式。这时, 用地址项 `iaddr(11)` 提供二次间接地址。该方式的实质是两级索引分配方式。系统此时是在二次间址块中记入所有一次间址块的盘号。在采用二次间址方式时, 文件最大长度可达 4 GB 。同理, 地址项 `iaddr(12)` 作为三次间接地址, 其所允许的文件最大长度可达 4 TB 。





6.4 目录管理

通常，在现代计算机系统中，都要存储大量的文件。为了能对这些文件实施有效的管理，必须对它们加以妥善组织，这主要是通过文件目录实现的。文件目录也是一种数据结构，用于标识系统中的文件及其物理地址，供检索时使用。对目录管理的要求如下：

(1) 实现“按名存取”，即用户只须向系统提供所需访问文件的名称，便能快速准确地找到指定文件在外存上的存储位置。这是目录管理中最基本的功能，也是文件系统向用户提供的最基本的服务。





第六章 文件管理



(2) 提高对目录的检索速度。通过合理地组织目录结构的方法，可加快对目录的检索速度，从而提高对文件的存取速度。这是在设计一个大、中型文件系统时所追求的主要目标。

(3) 文件共享。在多用户系统中，应允许多个用户共享一个文件。这样就须在外存中只保留一份该文件的副本，供不同用户使用，以节省大量的存储空间，并方便用户和提高文件利用率。

(4) 允许文件重名。系统应允许不同用户对不同文件采用相同的名字，以便于用户按照自己的习惯给文件命名和使用文件。





第六章 文件管理



6.4.1 文件控制块和索引结点

1. 文件控制块

为了能对系统中的大量文件施以有效的管理，在文件控制块中，通常应含有三类信息，即基本信息、存取控制信息及使用信息。





第六章 文件管理



1) 基本信息类

基本信息类包括: ① 文件名, 指用于标识一个文件的符号名。在每个系统中, 每一个文件都必须有惟一的名称, 用户利用该名称进行存取。② 文件物理位置, 指文件在外存上的存储位置, 它包括存放文件的设备名、文件在外存上的起始盘块号、指示文件所占用的盘块数或字节数的文件长度。③ 文件逻辑结构, 指示文件是流式文件还是记录式文件、记录数; 文件是定长记录还是变长记录等。④ 文件的物理结构, 指示文件是顺序文件, 还是链接式文件或索引文件。





第六章 文件管理



2) 存取控制信息类

存取控制信息类包括：文件主的存取权限、核准用户的存取权限以及一般用户的存取权限。





第六章 文件管理



3) 使用信息类

使用信息类包括：文件的建立日期和时间、文件上一次修改的日期和时间及当前使用信息（这项信息包括当前已打开该文件的进程数、是否被其它进程锁住、文件在内存中是否已被修改但尚未拷贝到盘上）。应该说明，对于不同OS的文件系统，由于功能不同，可能只含有上述信息中的某些部分。

图6-15示出了MS-DOS中的文件控制块，其中含有文件名、文件所在的第一个盘块号、文件属性、文件建立日期和时间及文件长度等。FCB的长度为32个字节，对于360 KB的软盘，总共可包含112个FCB，共占4 KB的存储空间。





第六章 文件管理



文件名	扩展名	属性	备用	时间	日期	第一块号	盘块数

图6-15 MS-DOS的文件控制块





第六章 文件管理



2. 索引结点

1) 索引结点的引入

文件目录通常是存放在磁盘上的，当文件很多时，文件目录可能要占用大量的盘块。在查找目录的过程中，先将存放目录文件的第一个盘块中的目录调入内存，然后把用户所给定的文件名与目录项中的文件名逐一比较。若未找到指定文件，便再将下一个盘块中的目录项调入内存。设目录文件所占用的盘块数为 N ，按此方法查找，则查找一个目录项平均需要调入盘块 $(N+1)/2$ 次。假如一个FCB为64 B，盘块大小为1 KB，则每个盘块中只能存放16个FCB；若一个文件目录中共有640个FCB，需占用40个盘块，故平均查找一个文件需启动磁盘20次。





第六章 文件管理



稍加分析可以发现，在检索目录文件的过程中，只用到了文件名，仅当找到一个目录项(即其中的文件名与指定要查找的文件名相匹配)时，才需从该目录项中读出该文件的物理地址。而其它一些对该文件进行描述的信息，在检索目录时一概不用。显然，这些信息在检索目录时不需调入内存。为此，在有的系统中，如UNIX系统，便采用了把文件名与文件描述信息分开的办法，亦即，使文件描述信息单独形成一个称为索引结点的数据结构，简称为i结点。在文件目录中的每个目录项仅由文件名和指向该文件所对应的i结点的指针所构成。在UNIX系统中一个目录仅占16个字节，其中14个字节是文件名，2个字节为i结点指针。在1 KB的盘块中可做64个目录项，这样，为找到一个文件，可使平均启动磁盘次数减少到原来的1/4，大大节省了系统开销。图6-16示出了UNIX的文件目录项。





第六章 文件管理



文件名	索引结点编号
文件名 1	
文件名 2	
...	...

0

13 14

15

图6-16 UNIX的文件目录





第六章 文件管理



2) 磁盘索引结点

这是存放在磁盘上的索引结点。每个文件有惟一的一个磁盘索引结点，它主要包括以下内容：

- (1) 文件主标识符，即拥有该文件的个人或小组的标识符。
- (2) 文件类型，包括正规文件、目录文件或特别文件。
- (3) 文件存取权限，指各类用户对该文件的存取权限。
- (4) 文件物理地址，每一个索引结点中含有13个地址项，即 $iaddr(0) \sim iaddr(12)$ ，它们以直接或间接方式给出数据文件所在盘块的编号。





第六章 文件管理



(5) 文件长度，指以字节为单位的文件长度。

(6) 文件连接计数，表明在本文件系统中所有指向该(文件的)文件名的指针计数。

(7) 文件存取时间，指本文件最近被进程存取的时间、最近被修改的时间及索引结点最近被修改的时间。





第六章 文件管理



3) 内存索引结点

这是存放在内存中的索引结点。当文件被打开时，要将磁盘索引结点拷贝到内存的索引结点中，便于以后使用。在内存索引结点中又增加了以下内容：

- (1) 索引结点编号，用于标识内存索引结点。
- (2) 状态，指示i结点是否上锁或被修改。
- (3) 访问计数，每当有一进程要访问此i结点时，将该访问计数加1，访问完再减1。
- (4) 文件所属文件系统的逻辑设备号。
- (5) 链接指针。设置有分别指向空闲链表和散列队列的指针。





6.4.2 目录结构

1. 单级目录结构

这是最简单的目录结构。在整个文件系统中只建立一张目录表，每个文件占一个目录项，目录项中含文件名、文件扩展名、文件长度、文件类型、文件物理地址以及其它文件属性。此外，为表明每个目录项是否空闲，又设置了一个状态位。单级目录如图6-17所示。





第六章 文件管理



文件名	物理地址	文件说明	状态位
文件名 1			
文件名 2			
...			

图6-17 单级目录





第六章 文件管理



每当要建立一个新文件时，必须先检索所有的目录项，以保证新文件名在目录中是惟一的。然后再从目录表中找出一个空白目录项，填入新文件的文件名及其它说明信息，并置状态位为1。删除文件时，先从目录中找到该文件的目录项，回收该文件所占用的存储空间，然后再清除该目录项。

单级目录的优点是简单且能实现目录管理的基本功能——按名存取，但却存在下述一些缺点：

(1) 查找速度慢。对于稍具规模的文件系统，会拥有数目可观的目录项，致使为找到一个指定的目录项要花费较多的时间。对于一个具有 N 个目录项的单级目录，为检索出一个目录项，平均需查找 $N/2$ 个目录项。





第六章 文件管理



(2) 不允许重名。在一个目录表中的所有文件，都不能与另一个文件有相同的名字。然而，重名问题在多道程序环境下却又是难以避免的；即使在单用户环境下，当文件数超过数百个时，也难于记忆。

(3) 不便于实现文件共享。通常，每个用户都有自己的名字空间或命名习惯。因此，应当允许不同用户使用不同的文件名来访问同一个文件。然而，单级目录却要求所有用户都用同一个名字来访问同一文件。简言之，单级目录只能满足对目录管理的四点要求中的第一点，因而，它只能适用于单用户环境。





2. 两级目录

为了克服单级目录所存在的缺点，可以为每一个用户建立一个单独的用户文件目录UFD(User File Directory)。这些文件目录具有相似的结构，它由用户所有文件的文件控制块组成。此外，在系统中再建立一个主文件目录MFD(Master File Directory)；在主文件目录中，每个用户目录文件都占有一个目录项，其目录项中包括用户名和指向该用户目录文件的指针。如图6-18所示，图中的主目录中示出了三个用户名，即Wang、Zhang和Gao。





第六章 文件管理

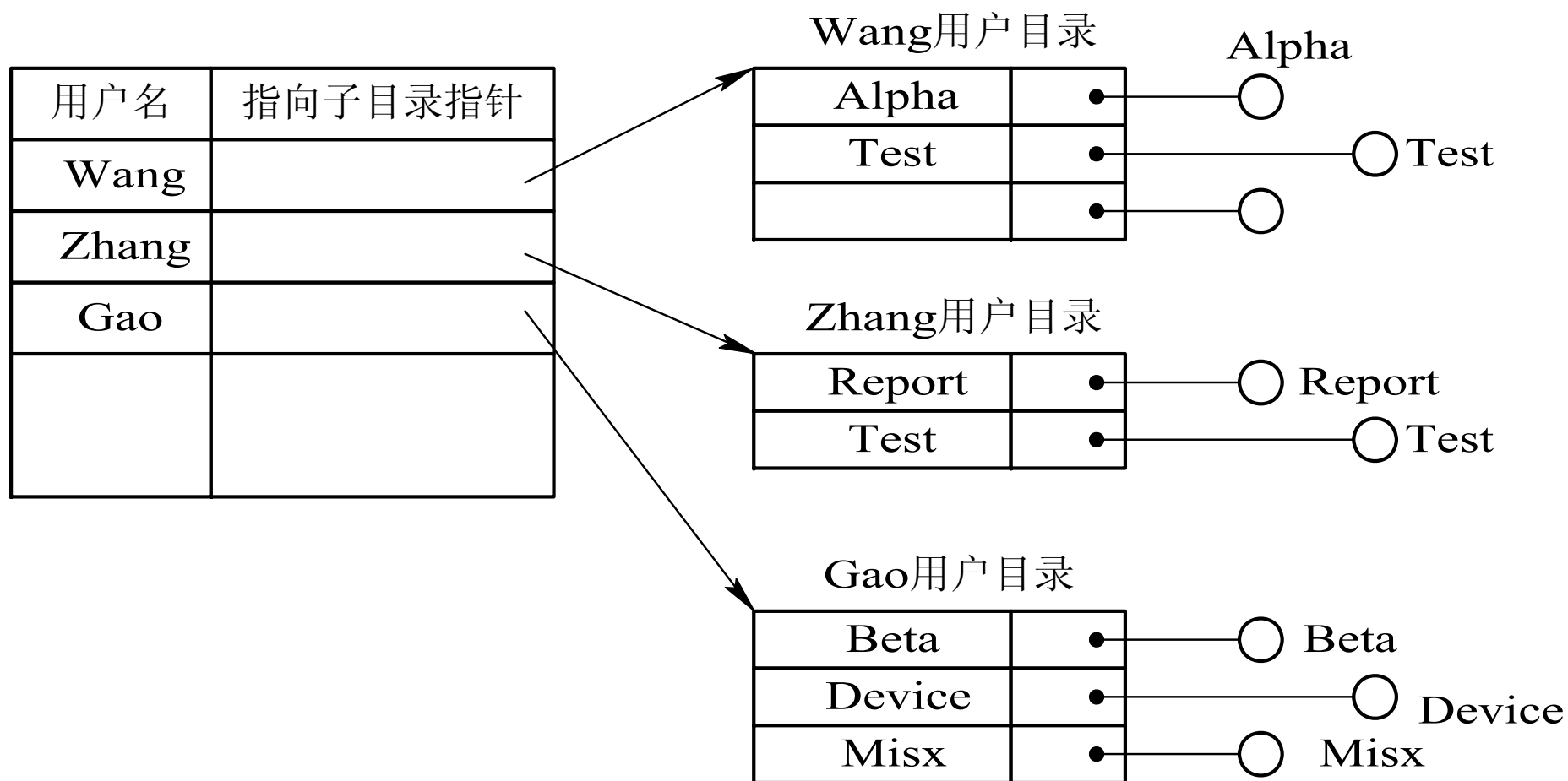


图6-18 两级目录结构





第六章 文件管理



在两级目录结构中，如果用户希望有自己的用户文件目录UFD，可以请求系统为自己建立一个用户文件目录；如果自己不再需要UFD，也可以请求系统管理员将它撤消。在有了UFD后，用户可以根据自己的需要创建新文件。每当此时，OS只需检查该用户的UFD，判定在该UFD中是否已有同名的另一个文件。若有，用户必须为新文件重新命名；若无，便在UFD中建立一个新目录项，将新文件名及其有关属性填入目录项中，并置其状态位为“1”。当用户要删除一个文件时，OS也只需查找该用户的UFD，从中找出指定文件的目录项，在回收该文件所占用的存储空间后，将该目录项删除。





第六章 文件管理



两级目录结构基本上克服了单级目录的缺点，并具有以下优点：

(1) 提高了检索目录的速度。如果在主目录中有 n 个子目录，每个用户目录最多为 m 个目录项，则为查找一指定的目录项，最多只需检索 $n + m$ 个目录项。但如果是采用单级目录结构，则最多需检索 $n \times m$ 个目录项。假定 $n = m$ ，可以看出，采用两级目录可使检索效率提高 $n/2$ 倍。

(2) 在不同的用户目录中，可以使用相同的文件名。只要在用户自己的UFD中，每一个文件名都是惟一的。例如，用户Wang可以用Test来命名自己的一个测试文件；而用户Zhang则可用Test来命名自己的一个并不同于Wang的Test的测试文件。





第六章 文件管理



(3) 不同用户还可使用不同的文件名来访问系统中的同一个共享文件。采用两级目录结构也存在一些问题。该结构虽然能有效地将多个用户隔开，在各用户之间完全无关时，这种隔离是一个优点；但当多个用户之间要相互合作去完成一个大任务，且一用户又需去访问其他用户的文件时，这种隔离便成为一个缺点，因为这种隔离会使诸用户之间不便于共享文件。





3. 多级目录结构

1) 目录结构

对于大型文件系统，通常采用三级或三级以上的目录结构，以提高对目录的检索速度和文件系统的性能。多级目录结构又称为树型目录结构，主目录在这里被称为根目录，把数据文件称为树叶，其它的目录均作为树的结点。图 6-19 示出了多级目录结构。图中，用方框代表目录文件，圆圈代表数据文件。在该树型目录结构中，主(根)目录中有三个用户的总目录项A、B和C。在B项所指出的B用户的总目录B中，又包括三个分目录F、E和D，其中每个分目录中又包含多个文件。如B目录中的F分目录中，包含J和N两个文件。为了提高文件系统的灵活性，应允许在一个目录文件中的目录项既是作为目录文件的FCB，又是数据文件的FCB，这一信息可用目录项中的一位来指示。例如，在图6-19中，用户A的总目录中，目录项A是目录文件的FCB，而目录项B和D则是数据文件的FCB。



第六章 文件管理

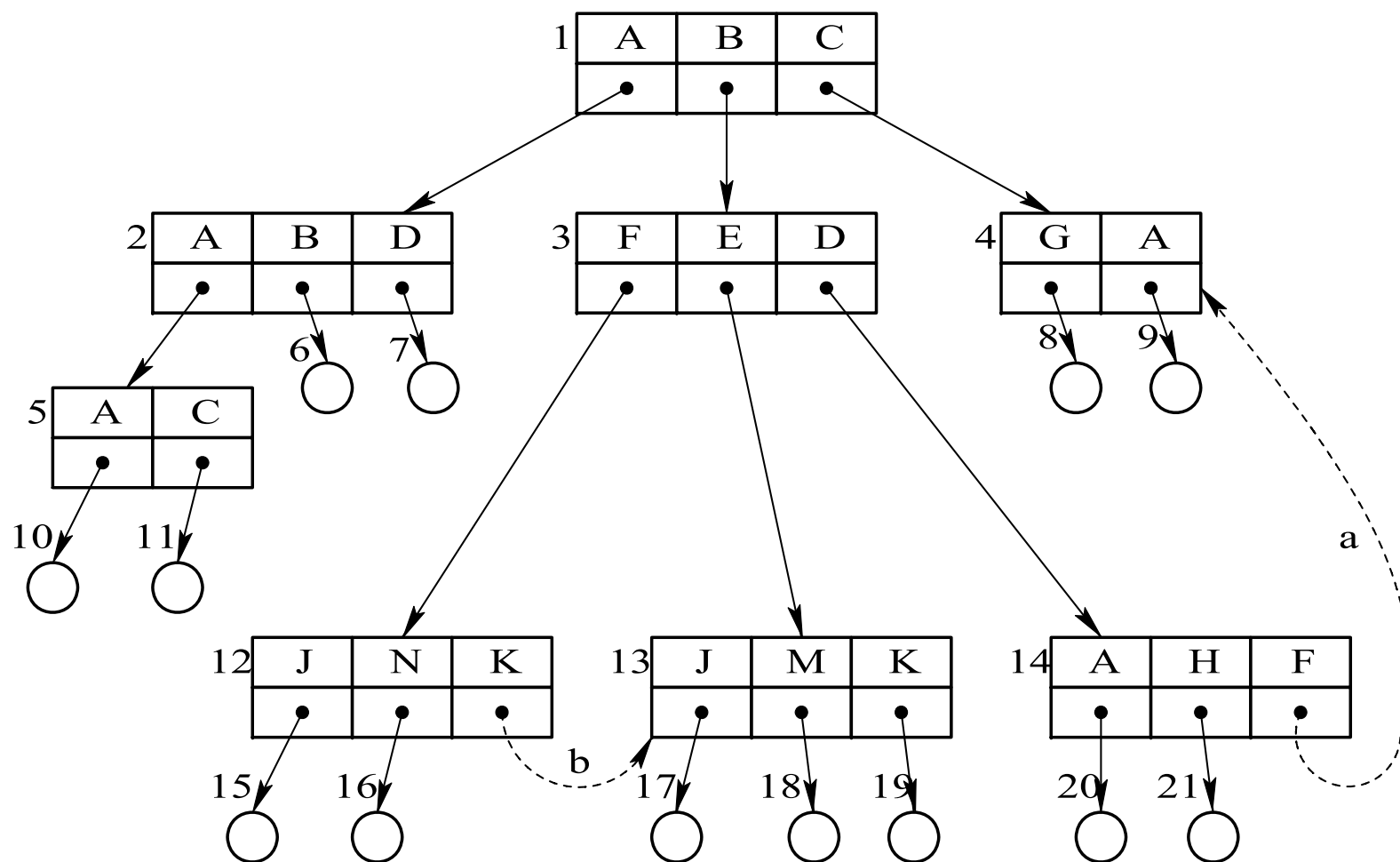


图6-19 多级目录结构





第六章 文件管理



2) 路径名

在树形目录结构中，从根目录到任何数据文件，都只有一条惟一的通路。在该路径上从树的根(即主目录)开始，把全部目录文件名与数据文件名依次地用“/”连接起来，即构成该数据文件的路径名(path name)。系统中的每一个文件都有惟一的路径名。例如，在图6-19 中用户B为访问文件J，应使用其路径名/B/F/J来访问。





第六章 文件管理



3) 当前目录(Current Directory)

当一个文件系统含有许多级时，每访问一个文件，都要使用从树根开始直到树叶(数据文件)为止的、包括各中间节点(目录)名的全路径名。这是相当麻烦的事，同时由于一个进程运行时所访问的文件大多仅局限于某个范围，因而非常不便。基于这一点，可为每个进程设置一个“当前目录”，又称为“工作目录”。进程对各文件的访问都相对于“当前目录”而进行。此时各文件所使用的路径名，只需从当前目录开始，逐级经过中间的目录文件，最后到达要访问的数据文件。把这一路径上的全部目录文件名与数据文件名用“/”连接形成路径名。如用户B的当前目录是F，则此时文件J的相对路径名仅是J本身。这样，把从当前目录开始直到数据文件为止所构成的路径名，称为相对路径名(relative path name)；而把从树根开始的路径名称为绝对路径名(absolute path name)。





第六章 文件管理



就多级目录较两级目录而言，其查询速度更快，同时层次结构更加清晰，能够更加有效地进行文件的管理和保护。在多级目录中，不同性质、不同用户的文件可以构成不同的目录子树，不同层次、不同用户的文件分别呈现在系统目录树中的不同层次或不同子树中，可以容易地赋予不同的存取权限。

但是在多级目录中查找一个文件，需要按路径名逐级访问中间节点，这就增加了磁盘访问次数，无疑将影响查询速度。

目前，大多数操作系统如UNIX、Linux和Windows系列都采用了多级目录结构。





4. 增加和删除目录

在树型目录结构中，用户可为自己建立UFD，并可再创建子目录。在用户要创建一个新文件时，只需查看在自己的UFD及其子目录中是否有与新建文件相同的文件名。若无，便可在UFD或其某个子目录中增加一个新目录项。

在树型目录中，对于一个已不再需要的目录，应如何删除其目录项，须视情况而定。这时，如果所要删除的目录是空的，即在该目录中已不再有任何文件，就可简单地将该目录项删除，使它在其上一级目录中对应的目录项为空；如果要删除的目录不空，即其中尚有几个文件或子目录，则可采用下述两种方法处理：





第六章 文件管理



(1) 不删除非空目录。当目录(文件)不空时,不能将其删除,而为了删除一个非空目录,必须先删除目录中的所有文件,使之先成为空目录,然后再予以删除。如果目录中还包含有子目录,还必须采取递归调用方式来将其删除,在MS-DOS中就是采用这种删除方式。

(2) 可删除非空目录。当要删除一个目录时,如果在该目录中还包含有文件,则目录中的所有文件和子目录也同时被删除。

上述两种方法实现起来都比较容易,第二种方法则更为方便,但比较危险。因为整个目录结构虽然用一条命令即能删除,但如果是一条错误命令,其后果则可能很严重。





6.4.3 目录查询技术

1. 线性检索法

线性检索法又称为顺序检索法。在单级目录中，利用用户提供的文件名，用顺序查找法直接从文件目录中找到指名文件的目录项。在树型目录中，用户提供的文件名是由多个文件分量名组成的路径名，此时须对多级目录进行查找。假定用户给定的文件路径名是`/usr/ast/mbox`，则查找`/usr/ast/mbox`文件的过程如图6-20所示。





第六章 文件管理



根目录

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

结点6是
/usr的目录

132

132号盘块是
/usr的目录

6	.
1	..
19	dick
30	erik
51	jim
26	ast
45	bal

结点26是
/usr/ast的目录

496

496号盘块是
/usr/ast的目录

26	.
6	..
64	grants
92	books
60	mbox
81	minik
17	src

在结点6中查找
usr字段

图6-20 查找/usr/ast/mbox的步骤





第六章 文件管理



具体查找过程说明如下：

首先，系统应先读入第一个文件分量名usr，用它与根目录文件(或当前目录文件)中各目录项中的文件名顺序地进行比较，从中找出匹配者，并得到匹配项的索引结点号6，再从6号索引结点中得知usr目录文件放在132号盘块中，将该盘块内容读入内存。

接着，系统再将路径名中的第二个文件分量名ast读入，用它与放在132号盘块中的第二级目录文件中各目录项的文件名顺序进行比较，又找到匹配项，从中得到ast的目录文件放在26号索引结点中，再从26号索引结点中得知/usr/ast是存放在496号盘块中，再读入496号盘块。





第六章 文件管理



然后，系统又将该文件的第三个分量名mbox读入，用它与第三级目录文件/usr/ast中各目录项中的文件名进行比较，最后得到/usr/ast/mbox的索引结点号为60，即在60号索引结点中存放了指定文件的物理地址。目录查询操作到此结束。如果在顺序查找过程中发现有一个文件分量名未能找到，则应停止查找，并返回“文件未找到”信息。





2. Hash方法

在6.2.5节中曾介绍了Hash文件。如果我们建立了一张Hash索引文件目录，便可利用Hash方法进行查询，即系统利用用户提供的文件名并将它变换为文件目录的索引值，再利用该索引值到目录中去查找，这将显著地提高检索速度。

顺便指出，在现代操作系统中，通常都提供了模式匹配功能，即在文件名中使用了通配符“*”、“?”等。对于使用了通配符的文件名，系统此时便无法利用Hash方法检索目录，因此，这时系统还是需要利用线性查找法查找目录。





第六章 文件管理



在进行文件名的转换时，有可能把 n 个不同的文件名转换为相同的Hash值，即出现了所谓的“冲突”。一种处理此“冲突”的有效规则是：

(1) 在利用Hash法索引查找目录时，如果目录表中相应的目录项是空的，则表示系统中并无指定文件。

(2) 如果目录项中的文件名与指定文件名相匹配，则表示该目录项正是所要寻找的文件所对应的目录项，故而可从中找到该文件所在的物理地址。

(3) 如果在目录表的相应目录项中的文件名与指定文件名并不匹配，则表示发生了“冲突”，此时须将其Hash值再加上一个常数(该常数应与目录的长度值互质)，形成新的索引值，再返回到第一步重新开始查找。





6.5 文件存储空间的管理

6.5.1 空闲表法和空闲链表法

1. 空闲表法

1) 空闲表

空闲表法属于连续分配方式，它与内存的动态分配方式雷同，它为每个文件分配一块连续的存储空间，即系统也为外存上的所有空闲区建立一张空闲表，每个空闲区对应于一个空闲表项，其中包括表项序号、该空闲区的第一个盘块号、该区的空闲盘块数等信息。再将所有空闲区按其起始盘块号递增的次序排列，如图6-21所示。





第六章 文件管理



序 号	第一空闲盘块号	空闲盘块数
1	2	4
2	9	3
3	15	5
4	—	—

图6-21 空闲盘块表





2) 存储空间的分配与回收

空闲盘区的分配与内存的动态分配类似，同样是采用首次适应算法、循环首次适应算法等。例如，在系统为某新建的文件分配空闲盘块时，先顺序地检索空闲表的各表项，直至找到第一个其大小能满足要求的空闲区，再将该盘区分配给用户(进程)，同时修改空闲表。系统在对用户所释放的存储空间进行回收时，也采取类似于内存回收的方法，即要考虑回收区是否与空闲表中插入点的前区和后区相邻接，对相邻接者应予以合并。





第六章 文件管理



应该说明，在内存分配上，虽然很少采用连续分配方式，然而在外存的管理中，由于这种分配方式具有较高的分配速度，可减少访问磁盘的I/O频率，故它在诸多分配方式中仍占有一席之地。例如，在前面所介绍的对换方式中，对对换空间一般都采用连续分配方式。对于文件系统，当文件较小(1~4个盘块)时，仍采用连续分配方式，为文件分配相邻接的几个盘块；当文件较大时，便采用离散分配方式。





2. 空闲链表法

空闲链表法是将所有空闲盘区拉成一条空闲链。根据构成链所用基本元素的不同，可把链表分成两种形式：空闲盘块链和空闲盘区链。

(1) 空闲盘块链。这是将磁盘上的所有空闲空间，以盘块为单位拉成一条链。当用户因创建文件而请求分配存储空间时，系统从链首开始，依次摘下适当数目的空闲盘块分配给用户。当用户因删除文件而释放存储空间时，系统将回收的盘块依次插入空闲盘块链的末尾。这种方法的优点是用于分配和回收一个盘块的过程非常简单，但在为一个文件分配盘块时，可能要重复操作多次。





第六章 文件管理



(2) 空闲盘区链。这是将磁盘上的所有空闲盘区(每个盘区可包含若干个盘块)拉成一条链。在每个盘区上除含有用于指示下一个空闲盘区的指针外,还应有能指明本盘区大小(盘块数)的信息。分配盘区的方法与内存的动态分区分配类似,通常采用首次适应算法。在回收盘区时,同样也要将回收区与相邻接的空闲盘区相合并。在采用首次适应算法时,为了提高对空闲盘区的检索速度,可以采用显式链接方法,亦即,在内存中为空闲盘区建立一张链表。





6.5.2 位示图法

1. 位示图

位示图是利用二进制的一位来表示磁盘中一个盘块的使用情况。当其值为“0”时，表示对应的盘块空闲；为“1”时，表示已分配。有的系统把“0”作为盘块已分配的标志，把“1”作为空闲标志。(它们在本质上是相同的，都是用一位的两种状态来标志空闲和已分配两种情况。)磁盘上的所有盘块都有一个二进制位与之对应，这样，由所有盘块所对应的位构成一个集合，称为位示图。通常可用 $m \times n$ 个位数来构成位示图，并使 $m \times n$ 等于磁盘的总块数，如图6-22所示。

位示图也可描述为一个二维数组map:

```
Var map: array of bit;
```





第六章 文件管理



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0	0	0	1	1	1	0	0	1	0	0	1	1	0
2	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1
3	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0
4																
⋮																
16																

图6-22 位示图





2. 盘块的分配

根据位示图进行盘块分配时，可分三步进行：

(1) 顺序扫描位示图，从中找出一个或一组其值为“0”的二进制位(“0”表示空闲时)。

(2) 将所找到的一个或一组二进制位转换成与之相应的盘块号。假定找到的其值为“0”的二进制位位于位示图的第*i*行、第*j*列，则其相应的盘块号应按下式计算：

$$b = n(i - 1) + j$$

式中，*n*代表每行的位数。





第六章 文件管理



(3) 修改位示图，令 $\text{map}[i,j]=1$ 。





第六章 文件管理



3. 盘块的回收

盘块的回收分两步：

(1) 将回收盘块的盘块号转换成位示图中的行号和列号。
转换公式为：

$$i = (b - 1) \text{DIV } n + 1$$

$$j = (b - 1) \text{MOD } n + 1$$





第六章 文件管理



(2) 修改位示图。令 $\text{map}[i,j] = 0$ 。

这种方法的主要优点是，从位示图中很容易找到一个或一组相邻接的空闲盘块。例如，我们需要找到6个相邻接的空闲盘块，这只需在位示图中找出6个其值连续为“0”的位即可。此外，由于位示图很小，占用空间少，因而可将它保存在内存中，进而使在每次进行盘区分配时，无需首先把盘区分配表读入内存，从而节省了许多磁盘的启动操作。因此，位示图常用于微型机和小型机中，如CP/M、Apple-DOS等OS中。





6.5.3 成组链接法

1. 空闲盘块的组织

(1) 空闲盘块号栈用来存放当前可用的一组空闲盘块的盘块号(最多含100个号), 以及栈中尚有的空闲盘块号数 N 。顺便指出, N 还兼作栈顶指针用。例如, 当 $N=100$ 时, 它指向 $S.free(99)$ 。由于栈是临界资源, 每次只允许一个进程去访问, 故系统为栈设置了一把锁。图6-23左部示出了空闲盘块号栈的结构。其中, $S.free(0)$ 是栈底, 栈满时的栈顶为 $S.free(99)$ 。





第六章 文件管理

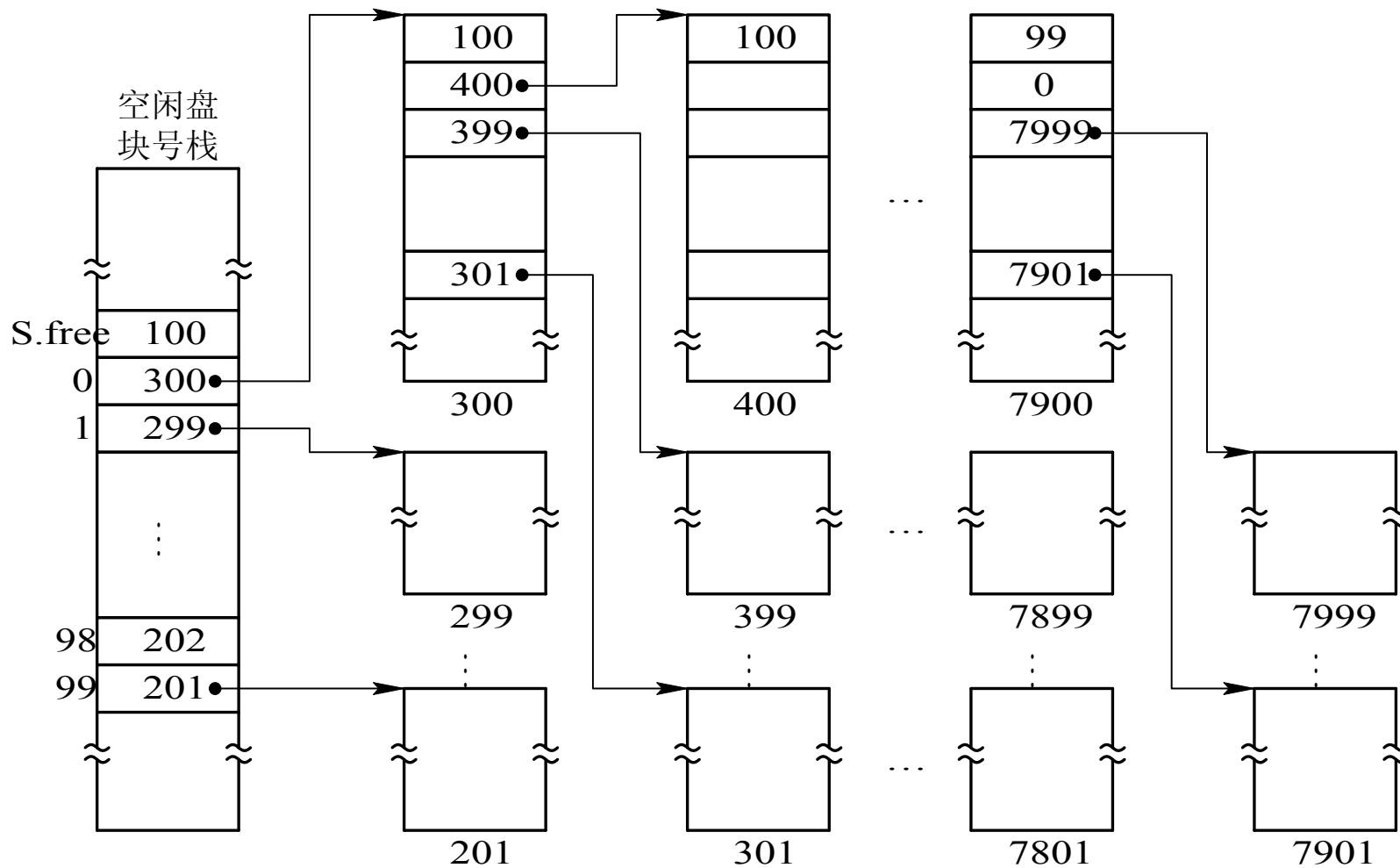


图6-23 空闲盘块的成组链接法





第六章 文件管理



(2) 文件区中的所有空闲盘块被分成若干个组，比如，将每100个盘块作为一组。假定盘上共有10 000个盘块，每块大小为1 KB，其中第201~7999号盘块用于存放文件，即作为文件区，这样，该区的最末一组盘块号应为7901~7999；次末组为7801~7900.....；第二组的盘块号为301~400；第一组为201~300，如图 6-23右部所示。

(3) 将每一组含有的盘块总数N和该组所有的盘块号记入其前一组的第一个盘块的S.free(0)~S.free(99)中。这样，由各组的第一个盘块可链成一条链。





第六章 文件管理



(4) 将第一组的盘块总数和所有的盘块号记入空闲盘块号栈中，作为当前可供分配的空闲盘块号。

(5) 最末一组只有99个盘块，其盘块号分别记入其前一组的S.free(1) ~ S.free(99)中，而在S.free(0)中则存放“0”，作为空闲盘块链的结束标志。(注：最后一组的盘块数应为99，不应是100，因为这是指可供使用的空闲盘块，其编号应为(1~99)，0号中放空闲盘块链的结尾标志。)





2. 空闲盘块的分配与回收

当系统要为用户分配文件所需的盘块时，须调用盘块分配过程来完成。该过程首先检查空闲盘块号栈是否上锁，如未上锁，便从栈顶取出一空闲盘块号，将与之对应的盘块分配给用户，然后将栈顶指针下移一格。若该盘块号已是栈底，即S.free(0)，这是当前栈中最后一个可分配的盘块号。由于在该盘块号所对应的盘块中记有下一组可用的盘块号，因此，须调用磁盘读过程，将栈底盘块号所对应盘块的内容读入栈中，作为新的盘块号栈的内容，并把原栈底对应的盘块分配出去(其中的有用数据已读入栈中)。然后，再分配一相应的缓冲区(作为该盘块的缓冲区)。最后，把栈中的空闲盘块数减1并返回。





第六章 文件管理



在系统回收空闲盘块时，须调用盘块回收过程进行回收。它是将回收盘块的盘块号记入空闲盘块号栈的顶部，并执行空闲盘块数加1操作。当栈中空闲盘块号数目已达100时，表示栈已满，便将现有栈中的100个盘块号记入新回收的盘块中，再将其盘块号作为新栈底。





6.6 文件共享与文件保护

6.6.1 基于索引结点的共享方式

在树型结构的目录中，当有两个(或多个)用户要共享一个子目录或文件时，必须将共享文件或子目录链接到两个(或多个)用户的目录中，才能方便地找到该文件，如图6-24所示。此时该文件系统的目录结构已不再是树型结构，而是个有向非循环图DAG(Directed Acyclic Graph)。





第六章 文件管理

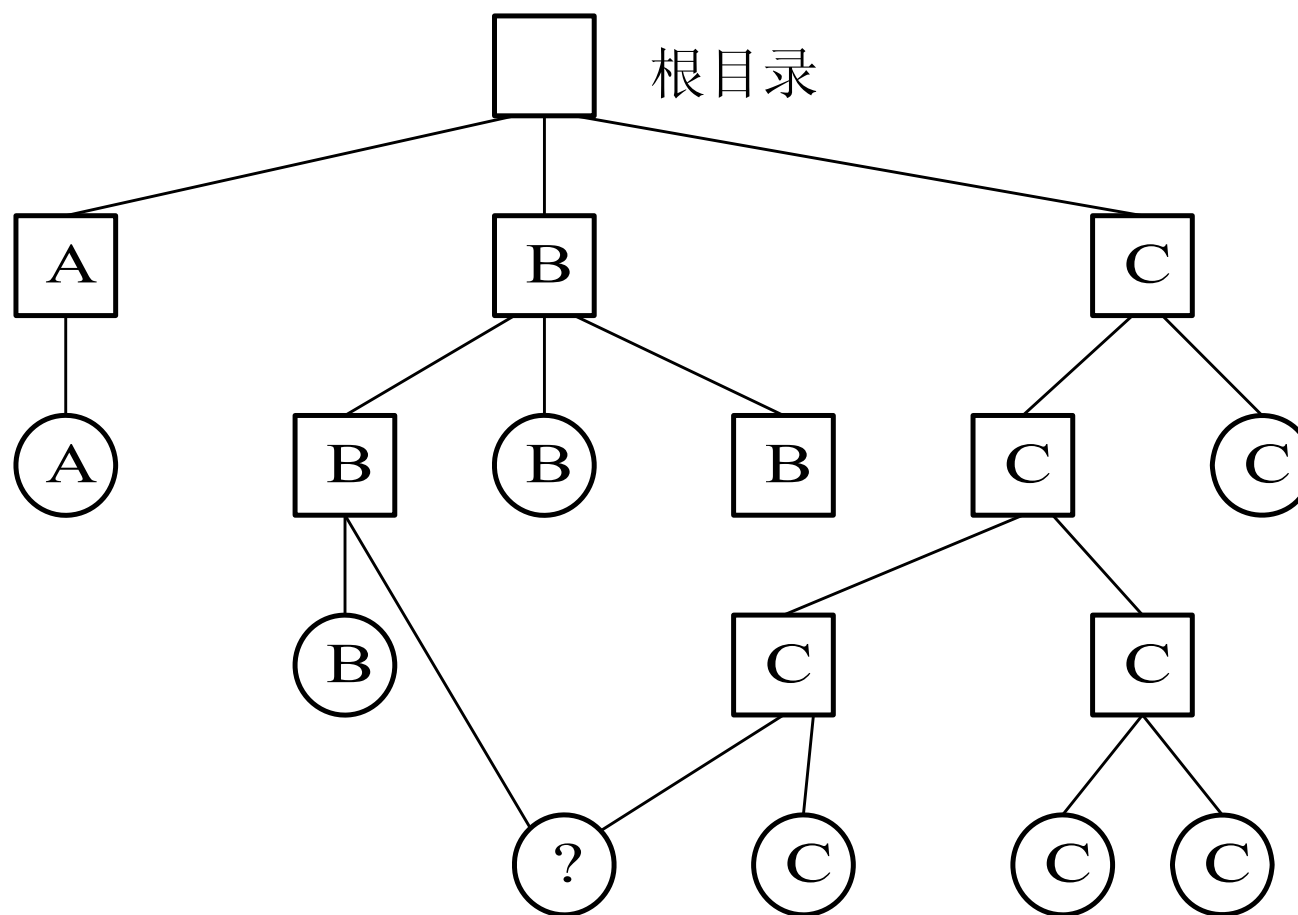


图6-24 包含有共享文件的文件系统





第六章 文件管理



如何建立B目录与共享文件之间的链接呢？如果在文件目录中包含了文件的物理地址，即文件所在盘块的盘块号，则在链接时，必须将文件的物理地址拷贝到B目录中去。但如果以后B或C还要继续向该文件中添加新内容，也必然要相应地再增加新的盘块，这须由附加操作Append来完成。而这些新增加的盘块，也只能出现在执行了操作的目录中。可见，这种变化对其他用户而言是不可见的，因而新增加的这部分内容已不能被共享。





第六章 文件管理



为了解决这个问题，可以引用索引结点，即诸如文件的物理地址及其它的文件属性等信息，不再是放在目录项中，而是放在索引结点中。在文件目录中只设置文件名及指向相应索引结点的指针，如图6-25所示。此时，由任何用户对文件进行Append操作或修改，所引起的相应结点内容的改变(例如，增加了新的盘块号和文件长度等)，都是其他用户可见的，从而也就能提供给其他用户来共享。





第六章 文件管理

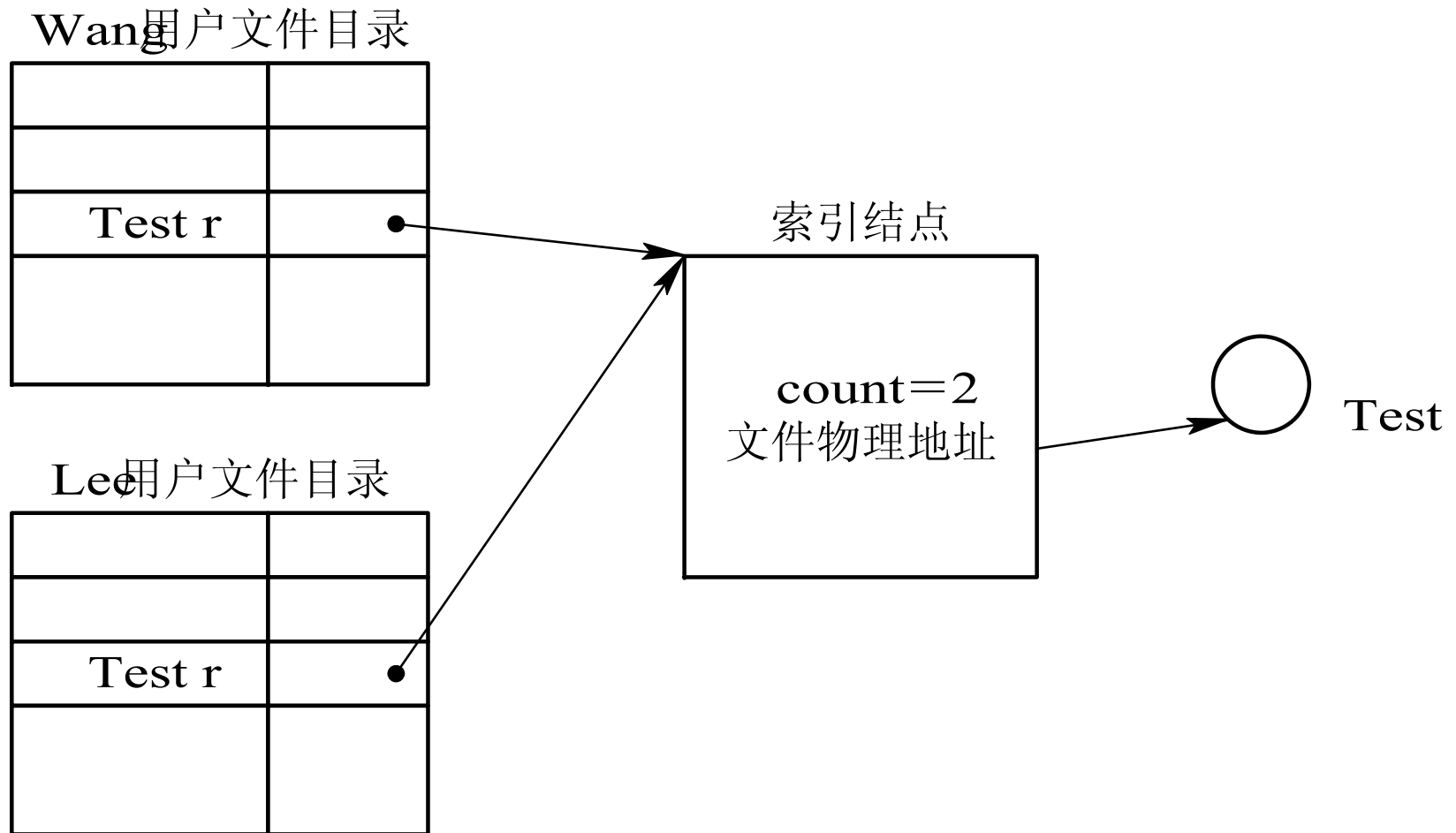


图6-25 基于索引结点的共享方式





第六章 文件管理



在索引结点中还应有一个链接计数count，用于表示链接到本索引结点(亦即文件)上的用户目录项的数目。当count=3时，表示有三个用户目录项连接到本文件上，或者说是三个用户共享此文件。





第六章 文件管理



当用户C创建一个新文件时，他便是该文件的所有者，此时将count置1。当有用户B要共享此文件时，在用户B的目录中增加一目录项，并设置一指针指向该文件的索引结点，此时，文件主仍然是C，count=2。如果用户C不再需要此文件，是否能将此文件删除呢？回答是否定的。因为，若删除了该文件，也必然删除了该文件的索引结点，这样便会使B的指针悬空，而B则可能正在此文件上执行写操作，此时将因此半途而废。但如果C不删除此文件而等待B继续使用，这样，由于文件主是C，如果系统要记账收费，则C必须为B使用此共享文件而付账，直至B不再需要。图6-26示出了B链接到文件上的前、后情况。





第六章 文件管理

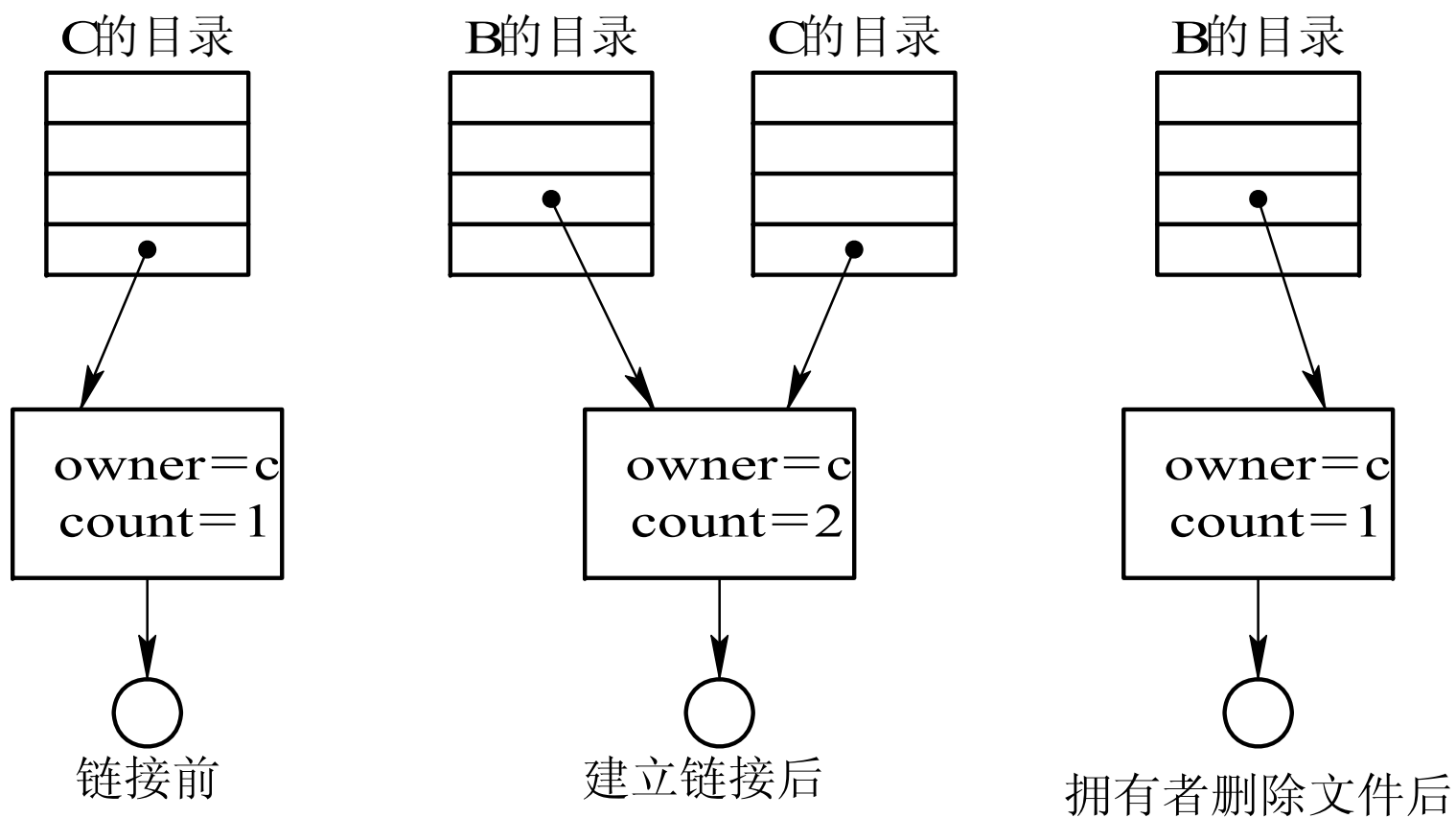


图6-26 进程B链接前后的情况





6.6.2 利用符号链实现文件共享

为使B能共享C的一个文件F，可以由系统创建一个LINK类型的新文件，也取名为F，并将F写入B的目录中，以实现B的目录与文件F的链接。在新文件中只包含被链接文件F的路径名。这样的链接方法被称为符号链接(Symbolic Linking)。新文件中的路径名则只被看作是符号链(Symbolic Link)，当B要访问被链接的文件F且正要读LINK类新文件时，此要求将被OS截获，OS根据新文件中的路径名去读该文件，于是就实现了用户B对文件F的共享。





第六章 文件管理



在利用符号链方式实现文件共享时，只是文件主才拥有指向其索引结点的指针；而共享该文件的其他用户则只有该文件的路径名，并不拥有指向其索引结点的指针。这样，也就不会发生在文件主删除一共享文件后留下一悬空指针的情况。当文件的拥有者把一个共享文件删除后，其他用户试图通过符号链去访问一个已被删除的共享文件时，会因系统找不到该文件而使访问失败，于是再将符号链删除，此时不会产生任何影响。





第六章 文件管理



然而符号链的共享方式也存在自己的问题：当其他用户去读共享文件时，系统是根据给定的文件路径名，逐个分量(名)地去查找目录，直至找到该文件的索引结点。因此，在每次访问共享文件时，都可能要多次地读盘。这使每次访问文件的开销甚大，且增加了启动磁盘的频率。此外，要为每个共享用户建立一条符号链，而由于该链实际上是一个文件，尽管该文件非常简单，却仍要为其配置一个索引结点，这也要耗费一定的磁盘空间。

符号链方式有一个很大的优点，是它能够用于链接(通过计算机网络)世界上任何地方的计算机中的文件，此时只需提供该文件所在机器的网络地址以及该机器中的文件路径即可。





6.6.3 磁盘容错技术

在现代计算机系统中，通常都存放了愈来愈多的宝贵信息供用户使用，给人们带来了极大的好处和方便，但同时也潜在着不安全性。影响文件安全性的主要因素有三：

(1) 人为因素，即由于人们有意或无意的行为，而使文件系统中的数据遭到破坏或丢失。

(2) 系统因素，即由于系统的某部分出现异常情况，而造成对数据的破坏或丢失。特别是作为数据存储介质的磁盘，在出现故障或损坏时，会对文件系统的安全性造成影响；

(3) 自然因素，即存放在磁盘上的数据，随着时间的推移将可能发生溢出或逐渐消失。





第六章 文件管理



为了确保文件系统的安全性，可针对上述原因而采取以下措施：

(1) 通过存取控制机制来防止由人为因素所造成的文件不安全性。

(2) 通过磁盘容错技术来防止由磁盘部分的故障所造成的文件不安全性。

(3) 通过“后备系统”来防止由自然因素所造成的不安全性。





第六章 文件管理



1. 第一级容错技术SFT- I

第一级容错技术(SFT- I)是最基本的一种磁盘容错技术,主要用于防止因磁盘表面缺陷所造成的数据丢失。它包含双份目录、双份文件分配表及写后读校验等措施。

1) 双份目录和双份文件分配表

在磁盘上存放的文件目录和文件分配表FAT, 是文件管理所用的重要数据结构。为了防止这些表格被破坏, 可在不同的磁盘上或在磁盘的不同区域中, 分别建立(双份)目录表和FAT。其中一份为主目录及主FAT; 另一份为备份目录及备份FAT。一旦由于磁盘表面缺陷而造成主文件目录或主FAT的损坏时, 系统便自动启用备份文件目录及备份FAT, 从而可以保证磁盘上的数据仍是可访问的。





2) 热修复重定向和写后读校验

由于磁盘价格昂贵，当磁盘表面有少量缺陷时，则可采取某种补救措施后继续使用磁盘。一般主要采取以下两个补救措施：

(1) 热修复重定向：系统将磁盘容量的一部分(例如2%~3%)作为热修复重定向区，用于存放当发现磁盘有缺陷时的待写数据，并对写入该区的所有数据进行登记，以便于以后对数据进行访问。





第六章 文件管理



(2) 写后读校验方式。为了保证所有写入磁盘的数据都能写入到完好的盘块中，应该在每次从内存缓冲区向磁盘中写入一个数据块后，又立即从磁盘上读出该数据块，并送至另一缓冲区中，再将该缓冲区内容与内存缓冲区中在写后仍保留的数据进行比较。若两者一致，便认为此次写入成功，可继续写下一个盘块；否则，再重写。若重写后两者仍不一致，则认为该盘块有缺陷，此时，便将应写入该盘块的数据，写入到热修复重定向区中。





2. 第二级容错技术SFT-II

1) 磁盘镜像(Disk Mirroring)

为了避免磁盘驱动器发生故障而丢失数据，便增设了磁盘镜像功能。为实现该功能，须在同一磁盘控制器下再增设一个完全相同的磁盘驱动器，如图6-27所示。当采用磁盘镜像方式时，在每次向主磁盘写入数据后，都需要将数据再写到备份磁盘上，使两个磁盘上具有完全相同的位像图。把备份磁盘看作是主磁盘的一面镜子，当主磁盘驱动器发生故障时，由于有备份磁盘的存在，在进行切换后，使主机仍能正常工作。磁盘镜像虽然实现了容错功能，但未能使服务器的磁盘I/O速度得到提高，却使磁盘的利用率降至仅为50%。如图6-27所示。





第六章 文件管理

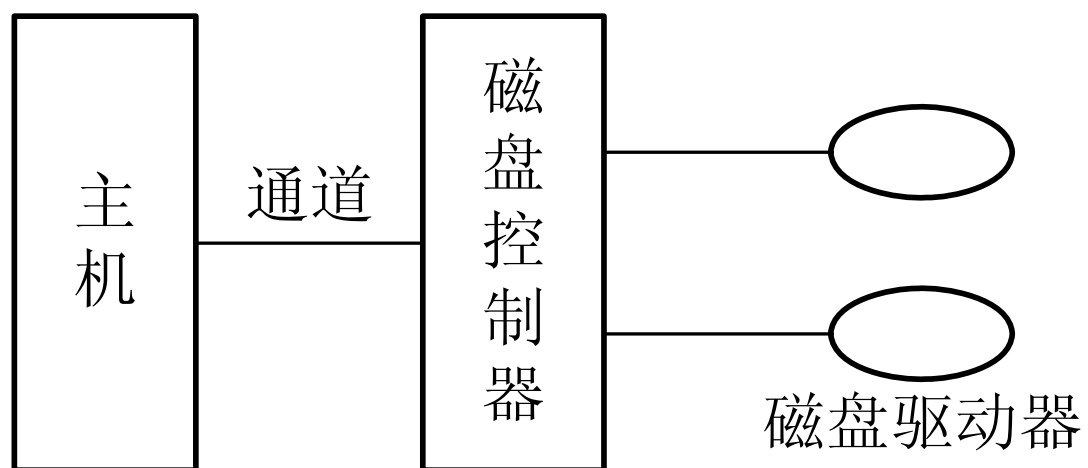


图6-27 磁盘镜像示意





2) 磁盘双工(Disk Duplexing)

如果控制这两台磁盘驱动器的磁盘控制器发生故障，或主机到磁盘控制器之间的通道发生了故障，磁盘镜像功能便起不到数据保护的作用。因此，在第二级容错技术中，又增加了磁盘双工功能，即将两台磁盘驱动器分别接到两个磁盘控制器上，同样使这两台磁盘机镜像成对，如图6-28所示。

在磁盘双工时，文件服务器同时将数据写到两个处于不同控制器下的磁盘上，使两者有完全相同的位像图。如果某个通道或控制器发生故障时，另一通道上的磁盘仍能正常工作，不会造成数据的丢失。在磁盘双工时，由于每一个磁盘都有自己的独立通道，故可同时(并行)地将数据写入磁盘，或读出数据。





第六章 文件管理

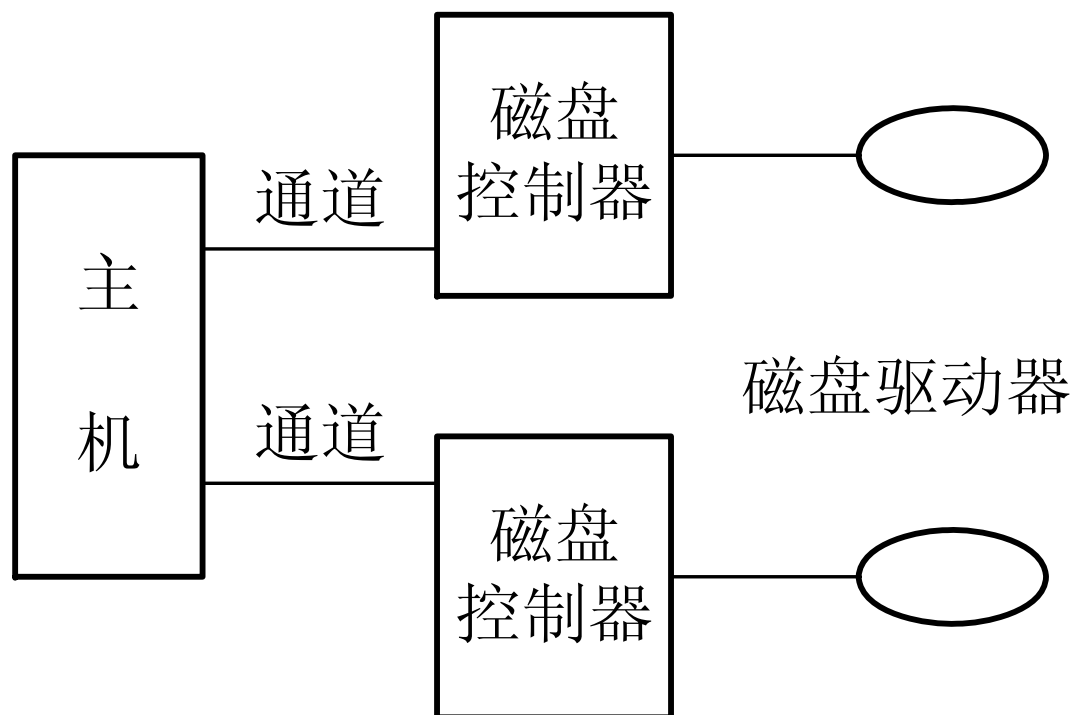


图6-28 磁盘双工示意





3. 基于集群技术的容错功能

进入20世纪90年代后，为了进一步增强服务器的并行处理能力和可用性，采用了多台SMP服务器来实现集群系统服务器。所谓集群，是指由一组互连的自主计算机组成统一的计算机系统，给人们的感觉是，它们是一台机器。利用集群系统不仅可提高系统的并行处理能力，还可用于提高系统的可用性，它们是当前使用最广泛的一类具有容错功能的集群系统。其主要工作模式有三种：① 热备份模式；② 互为备份模式；③ 公用磁盘模式。





1) 双机热备份模式

如图6-29所示，在这种模式的系统中，备有两台服务器，两者的处理能力通常是完全相同的，一台作为主服务器，另一台作为备份服务器。平时主服务器运行，备份服务器则时刻监视着主服务器的运行，一旦主服务器出现故障，备份服务器便立即接替主服务器的工作而成为系统中的主服务器，修复后的服务器再作为备份服务器。





第六章 文件管理

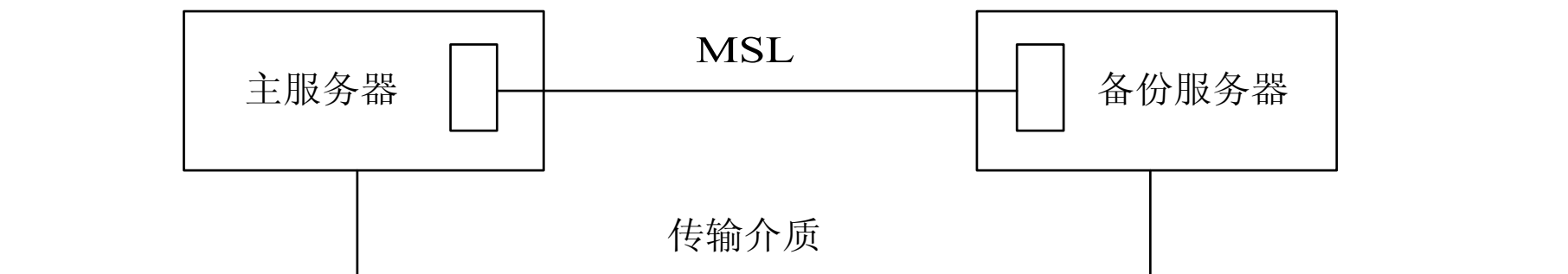


图6-29 双机热备份模式





2) 双机互为备份模式

在双机互为备份的模式中，平时，两台服务器均为在线服务器，它们各自完成自己的任务，例如，一台作为数据库服务器，另一台作为电子邮件服务器。为了实现两者互为备份，在两台服务器之间，应通过某种专线连接起来。如果希望两台服务器之间能相距较远，最好利用FDDI单模光纤来连接两台服务器，在此情况下，最好再通过路由器将两台服务器互连起来，作为备份通信线路。图6-30示出了双机互为备份系统的情况。





第六章 文件管理

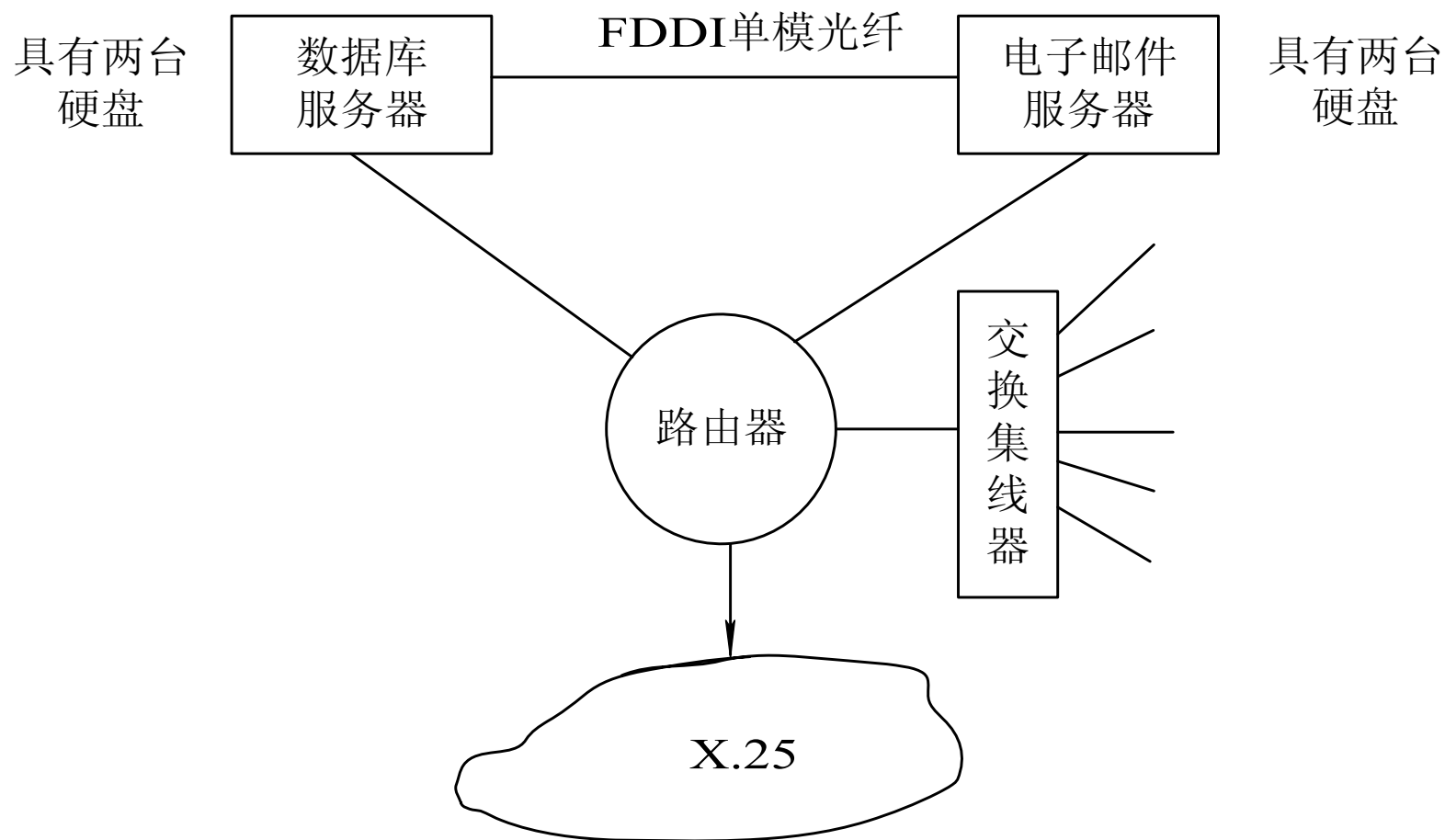


图6-30 双机互为备份系统的示意图





第六章 文件管理



在互为备份的模式中，最好在每台服务器内都配置两台硬盘，一个用于装载系统程序和应用程序，另一个用于接收由另一台服务器发来的备份数据，作为该服务器的镜像盘。在正常运行时，镜像盘对本地用户是锁死的，这样就较易于保证在镜像盘中数据的正确性。如果仅有一个硬盘，则可用建立虚拟盘的方式或分区方式来分别存放系统程序和应用程序，以及另一台服务器的备份数据。





第六章 文件管理



如果通过专线链接检查到某台服务器发生了故障，此时，再通过路由器去验证这台服务器是否真的发生了故障。如果故障被证实，则由正常服务器向故障服务器的客户机发出广播信息，表明要进行切换。连接到故障服务器上的客户机在切换过程中会感觉到网络服务器的短暂停顿。在切换成功后，客户机无需重新登录便可继续使用网络提供的服务和访问服务器上的数据。而对于接在非故障服务器上的客户机，则只会感觉到网络服务稍有减慢而已，不会有任何影响。当故障服务器修复并重新连到网上后，已被迁移到无故障服务器上的服务功能将被返回，恢复正常工作。





第六章 文件管理



这种模式的优点是两台服务器都可用于处理任务，因而系统效率较高，现在已将这种模式从两台机器扩大到4台、8台、16台甚至更多。系统中所有的机器都可用于处理任务，当其中一台发生故障时，系统可指定另一台机器来接替它的工作。





第六章 文件管理



3) 公用磁盘模式

为了减少信息复制的开销，可以将多台计算机连接到一台公共的磁盘系统上去。该公共磁盘被划分为若干个卷。每台计算机使用一个卷。如果某台计算机发生故障，此时系统将重新进行配置，根据某种调度策略来选择另一台替代机器，后者对发生故障的机器的卷拥有所有权，从而来接替故障计算机所承担的任务。这种模式的优点是：消除了信息的复制时间，因而减少了网络和服务器的开销。





6.7 数据一致性控制

6.7.1 事务

1. 事务的定义

事务是用于访问和修改各种数据项的一个程序单位。事务也可以被看做是一系列相关读和写操作。被访问的数据可以分散地存放在同一文件的不同记录中，也可放在多个文件中。只有对分布在不同位置的同一数据所进行的读和写(含修改)操作全部完成时，才能再以托付操作(Commit Operation)来终止事务。只要有一个读、写或修改操作失败，便须执行夭折操作(Abort Operation)。读或写操作的失败可能是由于逻辑错误，也可能是系统故障所导致的。





第六章 文件管理



一个夭折的事务，通常已执行了一些操作，因而可能已对某些数据做了修改。为使夭折的事务不会引起数据的不一致性，须将该事务内刚被修改的数据项恢复成原来的情况，使系统中各数据项与该事务未执行时的数据项内容完全相同。此时，可以说该事务“已被退回”(rolled back)。不难看出，一个事务在对一批数据执行修改操作时，要么全部完成，并用修改后的数据去代替原来的数据，要么一个也不修改。事务操作所具有的这种特性，就是我们在第二章中曾讲过的“原子性”。





2. 事务记录(Transaction Record)

为了实现上述的原子修改，通常须借助于称为事务记录的数据结构来实现。这些数据结构被放在稳定存储器中，用来记录在事务运行时数据项修改的全部信息，故又称为运行记录(Log)。该记录中包括有下列字段：

- 事务名：用于标识该事务的惟一名字；
- 数据项名：指被修改数据项的惟一名字；
- 旧值：修改前数据项的值；
- 新值：修改后数据项将具有的值。





第六章 文件管理



在事务记录表中的每一记录，描述了在事务运行中的重要事务操作，如修改操作、开始事务、托付事务或夭折事务等。在一个事务 T_i 开始执行时，〈 T_i 开始〉记录被写入事务记录表中；在 T_i 执行期间，在 T_i 的任何写(修改)操作之前，便写一适当的新记录到事务记录表中；当 T_i 进行托付时，把一个〈 T_i 托付〉记录写入事务记录表中。





3. 恢复算法

由于一组被事务 T_i 修改的数据以及它们被修改前和修改后的值都能在事务记录表中找到，因此，利用事务记录表，系统能处理任何故障而不致使故障造成非易失性存储器中信息的丢失。恢复算法可利用以下两个过程：

(1) undo $\langle T_i \rangle$ 。该过程把所有被事务 T_i 修改过的数据恢复为修改前的值。

(2) redo $\langle T_i \rangle$ 。该过程把所有被事务 T_i 修改过的数据设置为新值。





第六章 文件管理



如果系统发生故障，系统应对以前所发生的事务进行清理。通过查找事务记录表，可以把尚未清理的事务分成两类。一类是其所包含的各类操作都已完成的事务。确定为这一类事务的依据是，在事务记录表中，既包含了〈 T_i 开始〉记录，又包含了〈 T_i 托付〉记录。此时系统利用redo 〈 T_i 〉过程，把所有已被修改的数据设置成新值。另一类是其所包含的各个操作并未全部完成的事务。对于事务 T_i ，如果在Log表中只有〈 T_i 开始〉记录而无〈 T_i 托付〉记录，则此 T_i 便属于这类事务。此时，系统便利用undo 〈 T_i 〉过程，将所有已被修改的数据，恢复为修改前的值。





6.7.2 检查点

1. 检查点(Check Points)的作用

如前所述，当系统发生故障时，必须去检查整个Log表，以确定哪些事务需要利用redo $\langle T_i \rangle$ 过程去设置新值，而哪些事务需要利用undo $\langle T_i \rangle$ 过程去恢复数据的旧值。由于在系统中可能存在着许多并发执行的事务，因而在事务记录表中就会有許多事务执行操作的记录。随着时间的推移，记录的数据也会愈来愈多。因此，一旦系统发生故障，在事务记录表中的记录清理起来就非常费时。





第六章 文件管理



引入检查点的主要目的，是使对事务记录表中事务记录的清理工作经常化，即每隔一定时间便做一次下述工作：首先是将驻留在易失性存储器(内存)中的当前事务记录表中的所有记录输出到稳定存储器中；其次是将驻留在易失性存储器中的所有已修改数据输出到稳定存储器中；然后将事务记录表中的〈检查点〉记录输出到稳定存储器中；最后是每当出现一个〈检查点〉记录时，系统便执行上小节所介绍的恢复操作，利用redo和undo过程实现恢复功能。





第六章 文件管理



如果一个事务 T_i 在检查点前就做了托付, 则在事务记录表中便会出现一个在检查点记录前的〈 T_i 托付〉记录。在这种情况下, 所有被 T_i 修改过的数据, 或者是在检查点前已写入稳定存储器, 或者是作为检查点记录自身的一部分写入稳定存储器中。因此, 以后在系统出现故障时, 就不必再执行redo操作了。





第六章 文件管理



2. 新的恢复算法



在引入检查点后，可以大大减少恢复处理的开销。因为在发生故障后，并不需要对事务记录表中的所有事务记录进行处理，而只需对最后一个检查点之后的事务记录进行处理。因此，恢复例程首先查找事务记录表，确定在最近检查点以前开始执行的最后的事务 T_i 。在找到这样的事务后，再返回去搜索事务记录表，便可找到第一个检查点记录，恢复例程便从该检查点开始，返回搜索各个事务的记录，并利用redo和undo过程对它们进行处理。

如果把所有在事务 T_i 以后开始执行的事务表示为事务集 T ，则新的恢复操作要求是：对所有在 T 中的事务 T_K ，如果在事务记录表中出现了〈 T_K 托付〉记录，则执行redo 〈 T_K 〉操作；反之，如果在事务记录表中并未出现〈 T_K 托付〉记录，则执行undo 〈 T_K 〉操作。





6.7.3 并发控制

1. 利用互斥锁实现“顺序性”

实现顺序性的一种最简单的方法是，设置一种用于实现互斥的锁，简称为互斥锁(Exclusive Lock)。在利用互斥锁实现顺序性时，应为每一个共享对象设置一把互斥锁。当一事务 T_i 要去访问某对象时，应先获得该对象的互斥锁。若成功，使用该锁将该对象锁住，于是事务 T_i 便可对该对象执行读或写操作；而其它事务由于未能获得该锁而不能访问该对象。如果 T_i 需要对一批对象进行访问，则为了保证事务操作的原子性， T_i 应先获得这一批对象的互斥锁，以将这些对象全部锁住。如果成功，便可对这一批对象执行读或写操作；操作完成后又将所有这些锁释放。但如果在这一批对象中的某一个对象已被其它事物锁住，则此时 T_i 应对此前已被 T_i 锁住的其它对象进行开锁，宣布此次事务运行失败，但不致引起数据的变化。





2. 利用互斥锁和共享锁实现顺序性

利用互斥锁实现顺序性的方法简单易行。目前有不少系统都是采用这种方法来保证事务操作的顺序性，但这却存在着效率不高的问题。因为一个共享文件虽然只允许一个事务去写，但却允许多个事务同时去读；而在利用互斥锁来锁住文件后，则只允许一个事务去读。为了提高运行效率而又引入了另一种形式的锁——共享锁(Shared Lock)。共享锁与互斥锁的区别在于：互斥锁仅允许一个事务对相应对象执行读或写操作，而共享锁则允许多个事务对相应对象执行读操作，不允许其中任何一个事务对对象执行写操作。





第六章 文件管理



在为一个对象设置了互斥锁和共享锁的情况下，如果事务 T_i 要对 Q 执行读操作，则只需去获得对象 Q 的共享锁。如果对象 Q 已被互斥锁锁住，则 T_i 必须等待；否则，便可获得共享锁而对 Q 执行读操作。如果 T_i 要对 Q 执行写操作，则 T_i 还须去获得 Q 的互斥锁。若失败，须等待；否则，可获得互斥锁而对 Q 执行写操作。利用共享锁和互斥锁来实现顺序性的方法，非常类似于我们在第二章中所介绍的读者—写者问题的解法。





6.7.4 重复数据的数据一致性问题

1. 重复文件的一致性

我们以UNIX类型的文件系统为例，来说明如何保证重复文件的一致性问题。对于通常的UNIX文件目录，其每个目录项中含有一个ASCII码的文件名和一个索引结点号，后者指向一个索引结点。当有重复文件时，一个目录项可由一个文件名和若干个索引结点号组成，每个索引结点号都是指向各自的索引结点。图6-31 示出了UNIX类型的目录和具有重复文件的目录。





第六章 文件管理



文件名	i 结点
文件 1	17
文件 2	22
文件 3	12
文件 4	84

文件名	i 结点		
文件 1	17	19	40
文件 2	22	72	91
文件 3	12	30	29
文件 4	84	15	66

图6-31 UNIX类型的目录





第六章 文件管理



在有重复文件时，如果一个文件拷贝被修改，则必须也同时修改其它几个文件拷贝，以保证各相应文件中数据的一致性。这可采用两种方法来实现：第一种方法是当一个文件被修改后，可查找文件目录，以得到其它几个拷贝的索引结点号，再从这些索引结点中找到各拷贝的物理位置，然后对这些拷贝做同样的修改；第二种方法是为新修改的文件建立几个拷贝，并用新拷贝去取代原来的文件拷贝。





2. 盘块号一致性的检查

为了描述盘块的使用情况，通常利用空闲盘块表(链)来记录所有尚未使用的空闲盘块的编号。文件分配表FAT则是用于记录已分配盘块的使用情况。由于OS经常访问这些数据结构，也对它们进行修改，而如果正在修改时，机器突然发生故障，此时也会使盘块数据结构中的数据产生不一致性现象。因此，在每次启动机器时，都应该检查相应的多个数据结构，看它们之间是否保持了数据的一致性。





第六章 文件管理



为了保证盘块数据结构(中数据)的一致性,可利用软件方法构成一个计数器表,每个盘块号占一个表项,可有0, ..., $N-1$ 项, N 为盘块总数。每一个表项中包含两个计数器,分别用作空闲盘块号计数器和数据盘块号计数器。计数器表中的表项数目等于盘块数 N 。在对盘块的数据结构进行检查时,应该先将计数器表中的所有表项初始化为0,然后,用 N 个空闲盘块号计数器所组成的第一组计数器来对从空闲盘块表(链)中读出的盘块号进行计数;再用 N 个数据盘块号计数器所组成的第二组计数器去对从文件分配表中读出的、已分配给文件使用的盘块号进行计数。如果情况正常,则上述两组计数器中对应的一对(计数器中的)数据应该互补,亦即,若某个盘块号在被第一组计数器进行计数后,使该盘块号计数器为1,则在第二组计数器中相应盘块号计数器中的计数必为0;反之亦然。但如果情况并非如此,则说明发生了某种错误。





第六章 文件管理



图6-32(a)示出了在正常情况下，在第一组计数器和第二组计数器中的盘块号计数值是互补的；而图 6-32(b)示出的则是一种不正常的情况，对盘块号2的计数值在两组计数器中都未出现(即均为0)。当检查出这种情况时，应向系统报告。该错误的影响并不大，只是盘块2未被利用。其解决方法也较简单，只需在空闲盘块表(链)中增加一个盘块号2。图 6-32(c)中示出了另一种错误，即盘块号4在空闲盘块表(链)中出现了两次，其解决方法是从空闲盘块表(链)中删除一个空闲盘块号4。图 6-32(d)中所示出的情况是相同的数据盘块号5出现了两次(或多次)，此种情况影响较严重，必须立即报告。





第六章 文件管理



计数器组 \ 盘块号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
空闲盘块号计数器组	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
数据盘块号计数器组	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a) 正常情况盘块号

计数器组 \ 盘块号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
空闲盘块号计数器组	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
数据盘块号计数器组	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

(b) 丢失了盘块盘块号

图6-32 检查盘块号一致性情况





第六章 文件管理



计数器组 \ 盘块号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
空闲盘块号计数器组	1	1	0	1	2	1	1	1	1	0	0	1	1	1	0	0
数据盘块号计数器组	0	0	1	0	0	0	0	0	0	1	1	0	0	0	1	0

(c) 空闲盘块号重复出现盘块号

计数器组 \ 盘块号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
空闲盘块号计数器组	1	1	0	1	1	0	1	1	1	0	0	1	1	1	0	0
数据盘块号计数器组	0	0	1	0	0	2	0	0	0	1	1	0	0	0	1	1

(d) 数据盘块号重复出现





3. 链接数一致性检查

在UNIX类型的文件目录中，其每个目录项内都含有一个索引结点号，用于指向该文件的索引结点。对于一个共享文件，其索引结点号会在目录中出现多次。例如，当有5个用户(进程)共享某文件时，其索引结点号会在目录中出现5次；另一方面，在该共享文件的索引结点中有一个链接计数count，用来指出共享本文件的用户(进程)数。在正常情况下这两个数据应该一致，否则就会出现数据不一致性差错。





第六章 文件管理



为了检查这种数据不一致性差错，同样要配置一张计数器表，此时应是为每个文件而不是为每个盘块建立一个表项，其中含有该索引结点号的计数值。在进行检查时，从根目录开始查找，每当在目录中遇到该索引结点号时，便在该计数器表中相应文件的表项上加1。当把所有目录都检查完后，便可将该计数器表中每个表项中的索引结点号计数值与该文件索引结点中的链接计数count值加以比较，如果两者一致，表示是正确的；否则，便是产生了链接数据不一致的错误。





第六章 文件管理



如果索引结点中的链接计数count值大于计数器表中相应索引结点号的计数值，则即使在所有共享此文件的用户都不再使用此文件时，其count值仍不为0，因而该文件不会被删除。这种错误的后果是使一些已无用户需要的文件仍驻留在磁盘上，浪费了存储空间。当然这种错误的性质并不严重。解决的方法是用计数器表中的正确的计数值去为count重新赋值。





第六章 文件管理



反之，如果出现count值小于计数器表中索引结点号计数值的情况时，就有潜在的危险。假如有两个用户共享一个文件，但是count值仍为1，这样，只要其中有一个用户不再需要此文件时，count值就会减为0，从而使系统将此文件删除，并释放其索引结点及文件所占用的盘块，导致另一共享此文件的用户所对应的目录项指向了一个空索引结点，最终是使该用户再无法访问此文件。如果该索引结点很快又被分配给其它文件，则又会带来潜在的危险。解决的方法是将count值置为正确值。





第七章 操作系统接口

7.1 联机用户接口

7.2 Shell命令语言

7.3 系统调用

7.4 UNIX系统调用

7.5 图形用户接口





7.1 联机用户接口

7.1.1 联机用户接口

联机用户接口，也称为联机命令接口。不同操作系统的联机命令接口有所不同，这不仅指命令的种类、数量及功能方面，也可能体现在命令的形式、用法等方面。不同的用法和形式构成了不同的用户界面，可分成以下两种：

- (1) 字符显示式用户界面；
- (2) 图形化用户界面。



1. 命令行方式

该方式是指以行为单位输入和显示不同的命令。每行长度一般不超过256个字符，命令的结束通常以回车符为标记。命令的执行是串行、间断的，后一个命令的输入一般需等到前一个命令执行结束，如用户键入的一条命令处理完成后，系统发出新的命令输入提示符，用户才可以继续输入下一条命令。



第七章 操作系统接口

也有许多操作系统提供了命令的并行执行方式，例如一条命令的执行需要耗费较长时间，并且用户也不急需其结果时（即两条命令执行是不相关的），则可以在一个命令的结尾输入特定的标记，将该命令作为后台命令处理，用户接着即可继续输入下一条命令，系统便可对两条命令进行并行处理。一般而言，对新用户来说，命令行方式十分繁琐，难以记忆，但对有经验的用户而言，命令行方式用起来快捷便当、十分灵活，所以，至今许多操作员仍常使用这种命令方式。

简单命令的一般形式为：

Command arg1 arg2 . . . argn

其中，Command 是命令名，又称命令动词，其余为该命令所带的执行参数，有些命令可以没有参数。



2. 批命令方式

在操作命令的实际使用过程中，经常遇到需要对多条命令的连续使用，或若干条命令的重复使用，或对不同命令进行选择性的使用的情况。如果用户每次都采用命令行方式，将命令一条条由键盘输入，既浪费时间，又容易出错。因此，操作系统都支持一种称为批命令的特别命令方式，允许用户预先把一系列命令组织在一种称为批命令文件的文件中，一次建立，多次执行。使用这种方式可减少用户输入命令的次数，既节省了时间和减少了出错概率，又方便了用户。通常批命令文件都有特殊的文件扩展名，如MS-DOS系统的 .BAT 文件。



同时，操作系统还提供了一套控制子命令，增强对命令文件使用的支持。用户可以使用这些子命令和形式参数书写批命令文件，使得这样的批命令文件可以执行不同的命令序列，从而增强了命令接口的处理能力。如UNIX 和Linux 中的Shell不仅是一种交互型命令解释程序，也是一种命令级程序设计语言解释系统，它允许用户使用Shell 简单命令、位置参数和控制流语句编制带形式参数的批命令文件，称做Shell 文件或Shell 过程，Shell 可以自动解释和执行该文件或过程中的命令。



7.1.2 联机命令的类型

为了能向用户提供多方面的服务，通常，OS都向用户提供了几十条甚至上百条的联机命令。根据这些命令所完成功能的不同，可把它们分成以下几类：① 系统访问类；② 磁盘操作类；③ 文件操作类；④ 目录操作类；⑤ 通信类；⑥ 其他命令。现分述如下。



1. 系统访问类

在单用户微型机中，一般没有设置系统访问命令。然而，在多用户系统中，为了保证系统的安全性，都毫无例外地设置了系统访问命令，即注册命令Login。用户在每次开始使用某终端时，都须使用该命令，使系统能识别该用户。凡要在多用户系统的终端上上机的用户，都必须先在系统管理员处获得一合法的注册名和口令。以后，每当用户在接通其所用终端的电源后，便由系统直接调用，并在屏幕上显示出以下的注册命令：



第七章 操作系统接口

Login: /提示用户键入自己的注册名

当用户键入正确的注册名，并按下回车键后，屏幕上又会出现：

Password: /提示用户键入自己的口令

用户在键入口令时，系统将关闭掉回送显示，以使口令不在屏幕上显示出来。如果键入的口令正确而使注册成功时，屏幕上会立即出现系统提示符(所用符号随系统而异)，表示用户可以开始键入命令。如果用户多次(通常不超过三次)键入的注册名或口令都有错，系统将解除与用户的联接。



2. 磁盘操作命令

在微机操作系统中，通常都提供了若干条磁盘操作命令。

(1) 磁盘格式化命令Format。它被用于对指定驱动器上的软盘进行格式化。每张新盘在使用前都必须先格式化。其目的是使磁盘记录格式能为操作系统所接受。可见，不同操作系统将磁盘初始化后的格式各异。此外，在格式化过程中，还将对有缺陷的磁道和扇区加保留记号，以防止将它分配给数据文件。



(2) 复制整个软盘命令Diskcopy。该命令用于复制整个磁盘，另外它还有附加的格式化功能。如果目标盘片是尚未格式化的，则该命令在执行时，首先将未格式化的软盘格式化，然后再进行复制。

(3) 软盘比较命令Diskcomp。该命令用于将源盘与目标盘的各磁道及各扇区中的数据逐一进行比较。

(4) 备份命令Backup。该命令用于把硬盘上的文件复制到软盘上，而RESTORE命令则完成相反的操作。



3. 文件操作命令

每个操作系统都提供了一组文件操作命令。在微机OS中的文件操作命令有下述几种：

- (1) 显示文件命令type：用于将指定文件内容显示在屏幕上。
- (2) 拷贝文件命令copy：用于实现文件的拷贝。
- (3) 文件比较命令comp：用于对两个指定文件进行比较。两文件可以在同一个或不同的驱动器上。
- (4) 重新命名命令Rename：用于将以第一参数命名的文件改成用第二参数给定的名字。
- (5) 删除文件命令erase：用于删除一个或一组文件，当参数路径名为*.BAK时，表示删除指定目录下的所有其扩展名为 .Bak的文件。



4. 目录操作命令

- (1) 建立子目录命令`mkdir`: 用于建立指定名字的新目录。
- (2) 显示目录命令`dir`: 用于显示指定磁盘中的目录项。
- (3) 删除子目录命令`rmdir`: 用于删除指定的子目录文件, 但不能删除普通文件, 而且, 一次只能删除一个空目录(其中仅含“.”和“..”两个文件), 不能删除根及当前目录。
- (4) 显示目录结构命令`tree`: 用于显示指定盘上的所有目录路径及其层次关系。
- (5) 改变当前目录命令`chdir`: 用于将当前目录改变为由路径名参数给定的目录。用“..”作参数时, 表示应返回到上一级目录下。



5. 其它命令

(1) 输入输出重定向命令。在有的OS中定义了两个标准I/O设备。通常，命令的输入取自标准输入设备，即键盘；而命令的输出通常是送往标准输出设备，即显示终端。如果在命令中设置输出重定向“>”符，其后接文件名或设备名，表示将命令的输出改向，送到指定文件或设备上。类似地，若在命令中设置输入重定向“<”符，则不再是从键盘而是从重定向符左边参数所指定的文件或设备上，取得输入信息。



(2) 管道连接。这是指把第一条命令的输出信息作为第二条命令的输入信息；类似地，又可把第二条命令的输出信息作为第三条命令的输入信息。这样，由两个(含两条)以上的命令可形成一条管道。在MS-DOS和UNIX中，都用“|”作为管道符号，其一般格式为：

Command1 | Command2 | ... | Commandn;



(3) 过滤命令。在UNIX及MS-DOS中都有过滤命令，用于读取指定文件或标准输入，从中找出由参数指定的模式，然后把所有包含该模式的行都打印出来。例如，MS-DOS中用命令

`find/N"erase" (路径名)`

可对由路径名指定的输入文件逐行检索，把含有字符串

“erase”的行输出。其中，/N是选择开关，表示输出含有指定字符串的行；如果不用N而用C，则表示只输出含有指定字符串的行数；若用V，则表示输出不含指定字符串的行。



(4) 批命令。为了能连续地使用多条键盘命令，或多次反复地执行指定的若干条命令，而又免去每次重敲这些命令的麻烦，可以提供一特定文件。在MS-DOS中提供了一种特殊文件，其后缀名用“.BAT”；在UNIX系统中称为命令文件。它们都是利用一些键盘命令构成一个程序，一次建立供多次使用。在MS-DOS中用batch命令去执行由指定或默认驱动器的工作目录上指定文件中所包含的一些命令。



7.1.3 键盘终端处理程序

为了实现人机交互，还须在微机或终端上配置相应的键盘终端处理程序，它应具有下述几方面的功能：

- (1) 接收用户从终端上打入的字符。
- (2) 字符缓冲，用于暂存所接收的字符。
- (3) 回送显示。
- (4) 屏幕编辑。
- (5) 特殊字符处理。



1. 字符接收功能

为了实现人机交互，键盘终端处理程序必须能够接收从终端输入的字符，并将之传送给用户程序。有两种方式来实现字符接收功能：

(1) 面向字符方式。驱动程序只接收从终端打入的字符，并且不加修改地将它传送给用户程序。这通常是一串未加工的ASCII码。但大多数的用户并不喜欢这种方式。



(2) 面向行方式。终端处理程序将所接收的字符暂存在行缓冲中，并可对行内字符进行编辑。仅在收到行结束符后，才将一行正确的信息送命令解释程序。在有的计算机中，从键盘硬件送出的是键的编码(简称键码)，而不是ASCII码。例如，当打入a键时，是将键码“30”放入I/O寄存器，此时，终端处理程序必须参照某种表格，将键码转换成ASCII码。应当注意，某些IBM的兼容机使用的不是标准键码。此时，处理程序还须选用相应的表格将其转换成标准键码。



2. 字符缓冲功能

为了能暂存从终端键入的字符，以降低中断处理器的频率，在终端处理程序中，还必须具有字符缓冲功能。字符缓冲可采用以下两种方式之一：

(1) 专用缓冲区方式。这是指系统为每个终端设置一个缓冲区，暂存用户键入的一批字符，缓冲区的典型长度为200个字符左右。这种方式较适合于单用户微机或终端很少的多用户机。当终端数目较多时，需要的缓冲区数目可能很大，且每个缓冲区的利用率也很低。例如，当有100个终端时，要求有20 KB的缓冲区。但专用缓冲区方式可使终端处理程序简化。图7-1(a)示出了专用缓冲区方式。

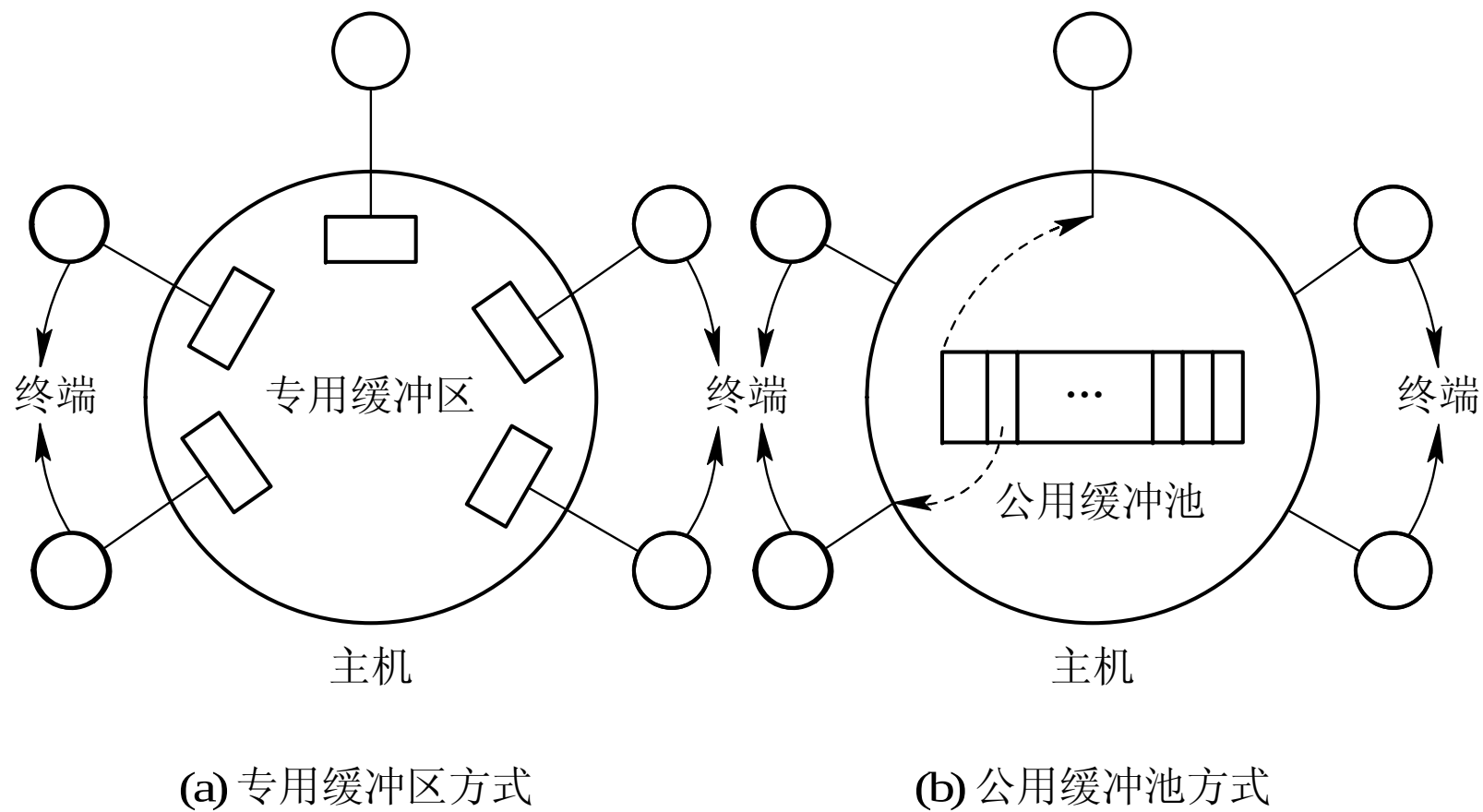


图7-1 两种缓冲方式



(2) 公用缓冲池方式。系统不必为每个终端设置专用缓冲区，只须设置一个由多个缓冲区构成的公用缓冲池。其中的每个缓冲区大小相同，如为20个字符，再将所有的空缓冲区链接成一个空缓冲区链。当终端有数据输入时，可先向空缓冲区链申请一空缓冲区来接收输入字符；当该缓冲区装满后，再申请一空缓冲区。这样，直至全部输入完毕，并利用链接指针将这些装有输入数据的缓冲区链接成一条输入链。每当该输入链中一个缓冲区内的字符被全部传送给用户程序后，便将该缓冲区从输入链中移出，再重新链入空缓冲区链中。显然，利用公用缓冲池方式可有效地提高缓冲的利用率。图7-1(b)示出了公用缓冲池方式。



3. 回送显示

回送显示(回显)是指每当用户从键盘输入一个字符后, 终端处理程序便将该字符送往屏幕显示。有些终端的回显由硬件实现, 其速度较快, 但往往会引起麻烦。如当用户键入口令时, 为防止口令被盗用, 显然不该有回显。此外, 用硬件实现回显也缺乏灵活性, 因而近年来多改用软件来实现回显, 这样可以做到在用户需要时才回显。用软件实现回显, 还可方便地进行字符变换, 如将键盘输入的小写英文字母变成大写, 或相反。驱动程序在将输入的字符送往屏幕回显时, 应打印在正确的位置上; 当光标走到一行的最后一个位置后, 便应返回到下一行的开始位置。例如, 当所键入的字符数目超过一行的80个(字符)时, 应自动地将下一个字符打印到下一行的开始位置。



4. 屏幕编辑

用户经常希望能对从键盘打入的数据(字符)进行修改,如删除(插入)一个或多个字符。为此,在终端处理程序中,还应能实现屏幕编辑功能,包括能提供若干个编辑键。常用的编辑键有:

(1) 删除字符键。它允许将用户刚键入的字符删除。在有的系统中是利用退格键即Backspace (Ctrl+H) 键。当用户敲该键时, 处理程序并不将刚键入的字符送入字符队列, 而是从字符队列中移出其前的一个字符。



- (2) 删除一行键。该键用于将刚输入的一行删去。
- (3) 插入键。利用该键在光标处可插入一个字符或一行正文。
- (4) 移动光标键。在键盘上有用于对光标进行上、下、左、右移动的键。
- (5) 屏幕上卷或下移键，等等。



5. 特殊字符处理

终端处理程序必须能对若干特殊字符进行及时处理，这些字符是：

(1) 中断字符。当程序在运行中出现异常情况时，用户可通过键入中断字符的办法来中止当前程序的运行。在许多系统中是利用Break或删除或Ctrl+C键作为中断字符。对中断字符的处理比较复杂。当终端处理程序收到用户键入的中断字符后，将向该终端上的所有进程发送一个要求进程终止的软中断信号，这些进程收到该软中断信号后，便进行自我终止。



(2) 停止上卷字符。用户键入此字符后，终端处理程序应使正在上卷的屏幕暂停上卷，以便用户仔细观察屏幕内容。在有的系统中，是利用Ctrl+S键来停止屏幕上卷的。

(3) 恢复上卷字符。有的系统利用Ctrl+Q键使停止上卷的屏幕恢复上卷。终端处理程序收到该字符后，便恢复屏幕的上卷功能。

上述的Ctrl+S与Ctrl+Q两字符并不被存储，而是被用去设置终端数据结构中的某个标志。每当终端试图输出时，都须先检查该标志。若该标志已被设置，便不再把字符送至屏幕。



7.1.4 命令解释程序

1. 命令解释程序的作用

在联机操作方式下，终端处理程序把用户键入的信息送键盘缓冲区中保存。一旦用户键入回车符，便立即把控制权交给命令处理程序。显然，对于不同的命令，应有能完成特定功能的命令处理程序与之对应。可见，命令解释程序的主要作用是在屏幕上给出提示符，请用户键入命令，然后读入该命令，识别命令，再转到相应命令处理程序的入口地址，把控制权交给该处理程序去执行，并将处理结果送屏幕上显示。若用户键入的命令有错，而命令解释程序未能予以识别，或在执行中间出现问题时，则应显示出某一出错信息。



2. 命令解释程序的组成

(1) 常驻部分。这部分包括一些中断服务子程序。例如：正常退出中断INT 20，它用于在用户程序执行完毕后，退回操作系统；驻留退出中断INT 27，用这种方式，退出程序可驻留在内存中；还有用于处理和显示标准错误信息的INT 24等。常驻部分还包括这样的程序：当用户程序终止后，它检查暂存部分是否已被用户程序覆盖，若已被覆盖，便重新将暂存部分调入内存。



(2) 初始化部分。它跟随在常驻内存部分之后，在启动时获得控制权。这部分还包括对AUTOEXEC. BAT文件的处理程序，并决定应用程序装入的基地址。每当系统接电或重新启动后，由处理程序找到并执行AUTOEXEC. BAT文件。由于该文件在用完后再不再被需要，因而它将被第一个由COMMAND. COM装入的文件所覆盖。

(3) 暂存部分。这部分主要是命令解释程序，并包含了所有的内部命令处理程序、批文件处理程序，以及装入和执行外部命令的程序。它们都驻留在内存中，但用户程序可以使用并覆盖这部分内存，在用户程序结束时，常驻程序又会将它们重新从磁盘调入内存，恢复暂存部分。



3. 命令解释程序的工作流程

系统在接通电源或复位后，初始化部分获得控制权，对整个系统完成初始化工作，并自动执行AUTOEXEC.BAT文件，之后便把控制权交给暂存部分。暂存部分首先读入键盘缓冲区中的命令，判别其文件名、扩展名及驱动器名是否正确。若发现有错，在给出出错信息后返回；若无错，再识别该命令。一种简单的识别命令的方法是基于一张表格，其中的每一表目都是由命令名及其处理程序的入口地址两项所组成的。如果暂存部分在该表中能找到键入的命令，且是内部命令，便可以直接从对应表项中获得该命令处理程序的入口地址，然后把控制权交给该处理程序去执行该命令。如果发现键入的命令不属于内部命令而是外部命令，则暂存部分还须为之建立命令行；再通过执行系统调用exec来装入该命令的处理程序，并得到其基地址；然后把控制权交给该程序去执行相应的命令。图7-2 示出了MS-DOS的COMMAND.COM的工作流程。

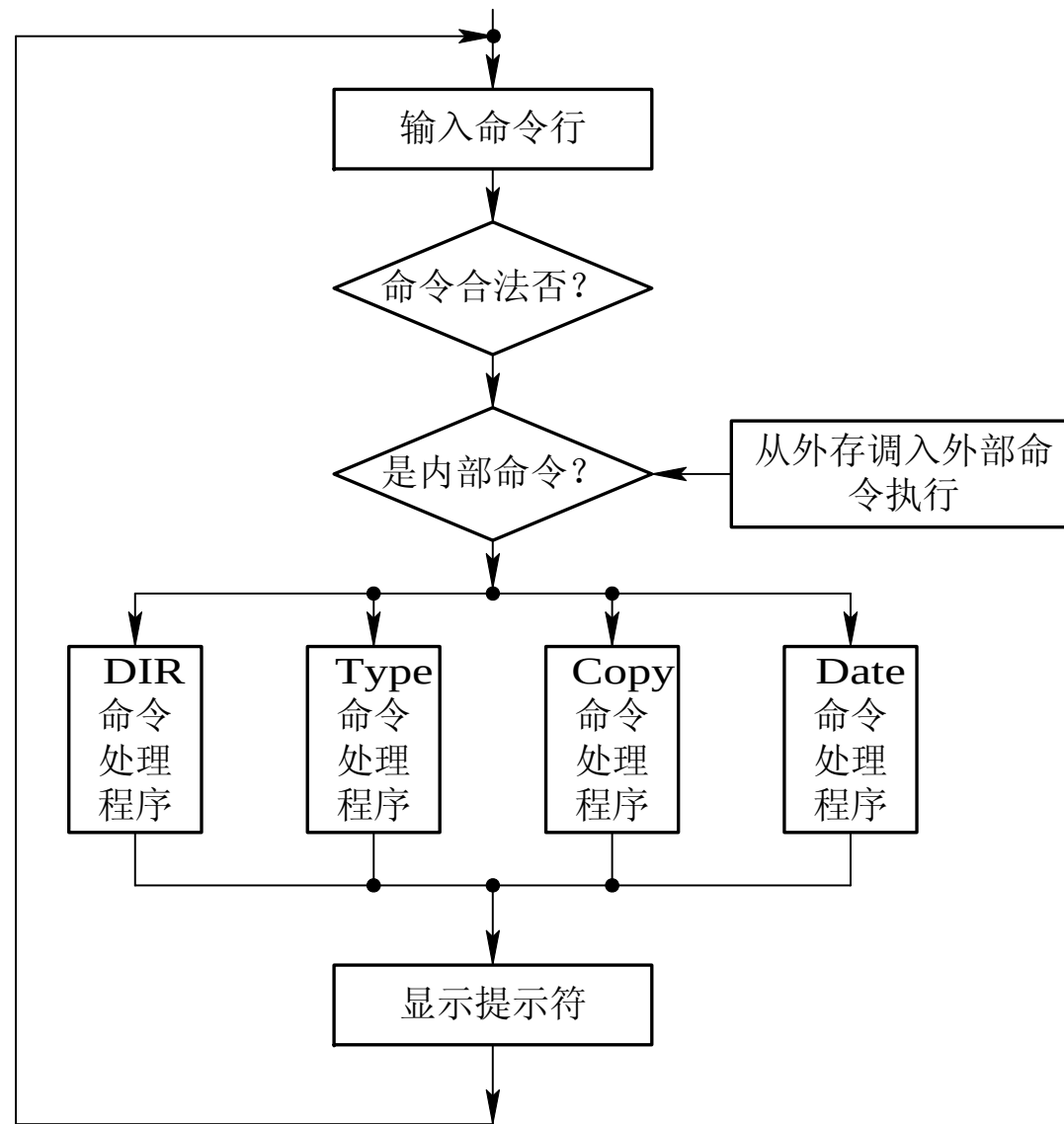
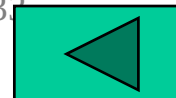


图7-2 COMMAND.COM的工作流程





7.2 Shell命令语言

7.2.1 简单命令

所谓简单命令，实际上是一个能完成某种功能的目标程序的名字。UNIX系统规定的命令由小写字母构成(仅前8个字母有效)。命令可带有参数表，用于给出执行命令时的附加信息。命令名与参数表之间还可使用一种称为选项的自变量，用破折号开始，后跟一个或多个字母、数字。选项是对命令的正常操作加以修改，一条命令可有多个选项，命令的格式如下：

```
$ Command-option argument list
```

例如：

```
$ LS file1 file2 ✓
```




第七章 操作系统接口

这是一条不带选项的列目录命令，\$是系统提示符。该命令用于列出file1和file2两个目录文件中所包含的目录项，并隐含地指出按英文字母顺序列表。若给出-tr选项，该命令可表示成：

```
$ LS-tr file1 file 2 ✓
```

其中，选项t和r分别表示按最近修改次序及按反字母顺序列表。

通常，命令名与该程序的功能紧密相关，以便于记忆。命令参数可多可少，也可缺省。例如：

```
$ LS ✓
```



1. 进入与退出系统

(1) 进入系统，也称为注册。事先，用户须与系统管理员商定一个唯一的用户名。管理员用该名字在系统文件树上，为用户建立一个子目录树的根结点。当用户打开自己的终端时，屏幕上会出现Login:提示，这时用户便可键入自己的注册名，并用回车符结束。然后，系统又询问用户口令，用户可用回车符或事先约定的口令键入。这两步均须正确通过检查，才能出现系统提示符(随系统而异)，以提示用户自己已通过检查，可以使用系统。若任一步骤有错，系统均通过提示要求用户重新键入。



(2) 退出系统。每当用户用完系统后，应向系统报告自己不再往系统装入任何处理要求。系统得知后，便马上为用户记账，清除用户的使用环境。若用户使用系统是免费的，退出操作仅仅是一种礼貌。如果用户使用的是多终端中的一个终端，为了退出，用户只需按下Control-D键即可，系统会重新给出提示符即Login，以表明该终端可供另一新用户使用。用户的进入与退出过程，实际上是由系统直接调用Login及Logout程序完成的。



2. 文件操作命令

(1) 显示文件内容命令cat。如果用户想了解自己在当前目录中的某个或某几个指定文件的内容时，便可使用下述格式的cat命令：

```
$ cat filename1 filename2 ✓
```

执行上述命令后，将按参数指定的顺序，依次把所列名字的文件内容送屏幕显示。若键入文件名有错，或该文件不在当前目录下，则该命令执行结果将显示指定文件不能打开的信息。



第七章 操作系统接口

(2) 复制文件副本的命令cp。其格式为：

```
cp source target
```

该命令用于对已存在的文件source建立一个名为target的副本。

(3) 对已有文件改名的命令mv。其格式为

```
mv oldname newname
```

用于把原来的老名字改成指定的新名字。

(4) 撤消文件的命令rm。它给出一个参数表，是要撤消的文件名清单。

(5) 确定文件类型的命令file。该命令带有一个参数表，用于给出想了解其(文件)类型的文件名清单。命令执行的结果将在屏幕上显示出各个文件的类型。



3. 目录操作命令

(1) 建立目录的命令`mkdir`(简称`md`)。当用户要创建或保存较多的文件时,应该以自己的注册名作为根结点,建立一棵子目录树,子树中的各结点(除树叶外)都是目录文件。可用`md`命令来构建一个目录,参数是新创建目录的名字。但应注意该命令的使用,必须在其父目录中有写许可时,才允许为其创建子目录。

(2) 撤消目录的命令`rmdir`(简称`rd`)。它实际上是`rm`命令的一个特例,用于删除一个或多个指定的下级空目录。若目录下仍有文件,该命令将被认为是一个错误操作,这样可以防止因不慎而消除了一个想保留的文件。命令的参数表用于给出要撤消的目录文件清单。



(3) 改变工作目录的命令cd。不带参数的cd命令将使用户从任何其它目录回到自己的注册目录上；若用全路径名作参数，cd命令将使用户来到由该路径名确定的结点上；若用当前目录的子目录名作参数，将把用户移到当前目录指定的下一级目录上(即用其下一级目录作为新的当前目录)；用“..”号或“*”号将使当前目录上移一级，即移到其父结点上。

(4) 改变对文件的存取方式的命令chmod。其格式为

```
chmod op-code permission filename
```




第七章 操作系统接口

其中，用于指明访问者的身份，可以是用户自己、用户组、所有其他用户及全部，分别用u、g、o和a表示；op-code是操作码，分别用 +、- 及 = 表示增加、消除及赋予访问者以某种权利；而permission则是分别用r、 w及x表示读、 写及执行许可。例如，命令

```
chmod go-w temp
```

表示消除用户组及所有其他用户对文件temp的写许可。



4. 系统询问命令

(1) 访问当前日期和时间命令date。例如，用命令

```
$ date ✓
```

屏幕上将给出当前的日期和时间，如为

```
Wed Aug 14 09:27:20 PDT 1991
```

表示当前日期是1991年9月14日、星期三，还有时间信息。若在命令名后给出参数，则date程序把参数作为重置系统时钟的时间。



(2) 询问系统当前用户的命令who。who命令可列出当前每一个处在系统中的用户的注册名、终端名和注册进入时间，并按终端标志的字母顺序排序。例如，报告有下列三用户：

```
Veronica bxo66 Aug 27 13:28
```

```
Rathomas dz24 Aug 28 07:42
```

```
Jlyates tty5 Aug 28 07:39
```



用户可用who命令了解系统的当前负荷情况；也可在与其他用户通信之前，用此命令去核实一下当前进入系统的用户及其所使用终端名和所用的正确的注册名。例如，用户在使用系统的过程中，有时会发现在打入一个请求后，系统响应很慢，这时用户可用“who|we-L”命令，使系统打印出当前的用户数目而不显示系统用户名等的完整清单，以得知当前用户数目。



(3) 显示当前目录路径名的命令pwd。当前目录的路径名是从根结点开始，通过分支上的所有结点到达当前目录结点为止的路径上的所有结点的名字拼起来构成的。用户的当前目录可能经常在树上移动。如果用户忘记了自己在哪里，便可用pwd确定自己的位置。



7.2.2 重定向与管道命令

1. 重定向命令

在UNIX系统中，由系统定义了三个文件。其中，有两个分别称为标准输入和标准输出的文件，各对应于终端键盘输入和终端屏幕输出。它们是在用户注册时，由Login程序打开的。这样，在用户程序执行时，隐含的标准输入是键盘输入，标准输出即屏幕(输出)显示。但用户程序中可能不要求从键盘输入，而是从某个指定文件上读取信息供程序使用；同样，用户可能希望把程序执行时所产生的结果数据，写到某个指定文件中而非屏幕上。这就使用户必须去改变输入与输出文件，即不使用标准输入、标准输出，而是把另外的某个指定文件或设备，作为输入或输出文件。



第七章 操作系统接口

Shell向用户提供了这种用于改变输入、输出设备的手段，此即标准输入与标准输出的重新定向。用重定向符“<”和“>”分别表示输入转向与输出转向。例如，对于命令

```
$ cat file1 ✓
```

表示将文件file1的内容在标准输出上打印出来。若改变其输出，用命令

```
$ cat file1>file2 ✓
```

时，表示把文件file1的内容打印输出到文件file2上。同理，对于命令

```
$ wc ✓
```




须指明的是，在做输出转向时，若上述的文件file2并不存在，则先创建它；若已存在，则认为它是空白的，执行上述输出转向命令时，是用命令的输出数据去重写该文件；如果文件file2事先已有内容，则命令执行结果将用文件file1的内容去更新文件file2的原有内容。现在，如果又要求把file4的内容附加到现有的文件file2的末尾，则应使用另一个输出转向符“>>”，即此时应再用命令

```
$ cat file4>>file2 ✓
```



第七章 操作系统接口

便可在文件file2中，除了上次复制的file1内容外，后面又附加了file4的内容。

当然，若想一次把两个文件file1和file4全部复制到file2中，则可用命令

```
$ cat file1 file4 >> file2 ✓
```

此外，也可在一个命令行中，同时改变输入与输出。例如，命令行

```
a.out <file1 >file0 ✓
```

表示在可执行文件a.out执行时，将从文件file1中提取数据，而把a.out的执行结果数据输出到文件file0中。



2. 管道命令

在有了上述的重定向思想后，为了进一步增强功能，人们又进一步把这种思想加以扩充，用符号“|”来连接两条命令，使其前一条命令的输出作为后一条命令的输入。即

```
$ command 1 | command 2 ✓
```

例如，对于下述输入

```
cat file|wc ✓
```

将使命令cat把文件file中的数据作为wc命令的计数用输入。



从概念上说，系统执行上述输入时，将为管道建立一个作为通信通道的pipe文件。这时，cat命令的输出既不出现在终端(屏幕)上，也不存入某中间文件，而是由UNIX系统来“缓冲”第一条命令的输出，并作为第二条命令的输入。在用管道线所连接的命令之间，实现单向、同步运行。其单向性表现在：只把管道线前面的命令的输出送入管道，而管道的输出数据仅供管道线后面的命令去读取。管道的同步特性则表现为：当一条管道满时，其前一条命令停止执行；而当管道空时，则其后一条命令停止运行。除此两种情况外，用管道所连接的两条命令“同时”运行。可见，利用管道功能，可以流水线方式实现命令的流水线化，即在单一命令行下，同时运行多条命令，以加速复杂任务的完成。



7.2.3 通信命令

1. 信箱通信命令mail

mail命令被作为在UNIX的各用户之间进行非交互式通信的工具。mail采用信箱通信方式。发信者把要发送的消息写成信件，“邮寄”到对方的信箱中。通常各用户的私有信箱采用各自的注册名命名，即它是目录/usr/spool/mail中的一个文件，而文件名又是用接收者的注册名来命名的。信箱中的信件可以一直保留到被信箱所有者消除为止。



因而，用mail进行通信时，不要求接收者利用终端与发送者会话。亦即，在发信者发送信息时，虽然接收者已在系统中注册过，但允许他此时没有使用系统；也可以是虽在使用系统，但拒绝接收任何信息。mail命令在用于发信时，把接收者的注册名当作参数打入后，便可在新行开始键入信件正文，最后仍在一个新行上用“.”来结束信件或用“^D”退出mail程序(也可带选项，此处从略)。



接收者也用mail命令读取信件，可使用可选项r、q或p等。其命令格式为

```
mail [-r][-q][-p][-file][-F persons]
```

由于信箱中可存放所接收的多个信件，这就存在一个选取信件的问题。上述几个选项分别表示：按先进先出顺序显示各信件的内容；在输入中断字符(Del或Return)后，退出mail程序而不改变信箱的内容；一次性地显示信箱全部内容而不带询问；把指定文件当作信件来显示。在不使用-p选项时，表示在显示完一个信件后，便出现“？”，以询问用户是否继续显示下一条消息，或选读完最后一条消息后退出mail。



2. 对话通信命令write

用这条命令可以使用户与当前在系统中的其他用户直接进行联机通信。由于UNIX系统允许一个用户同时在几个终端上注册，故在用此命令前，要用who命令去查看目标用户当前是否联机，或确定接收者所使用的终端名。命令格式为

```
write user[ttyname]
```

当接收者只有一个终端时，终端名可缺省。当接收者的终端被允许接收消息时，屏幕提示会通知接收者源用户名及其所用终端名。



3. 允许或拒绝接收消息命令mesg

mesg命令的格式为：

`mesg [-n] [-y]`

选项n表示拒绝对方的写许可(即拒绝接收消息)；选项y指示恢复对方的写许可，仅在此时，双方才可联机通信。当用户正在联机编写一份资料而不愿被别人干扰时，常选用n选项来拒绝对方的写许可。编辑完毕，再用带有y选项的mesg命令来恢复对方的写许可，不带自变量的mesg命令只报告当前状态而不改变它。



7.2.4 后台命令

有些命令需要执行很长的时间，这样，当用户键入该命令后，便会发现自己已无事可做，要一直等到该命令执行完毕，方可再键入下一条命令。这时用户自然会想到应该利用这段时间去做些别的事。UNIX系统提供了这种机制，用户可以在这种命令后面再加上“&”号，以告诉Shell将该命令放在后台执行，以便用户在前台继续键入其它命令。



第七章 操作系统接口

在后台运行的程序仍然把终端作为它的标准输出和标准错误文件，除非对它们进行重新定向。其标准输入文件是自动地被从终端定向到一个被称为“/dev/null”的空文件中。若shell未重定向标准输入，则shell和后台进程将会同时从终端进行读入。这时，用户从终端键入的字符可能被发送到一个进程或另一个进程，并不能预测哪个进程将得到该字符。

因此，对所有在后台运行的命令的标准输入，都必须加以重定向，从而使从终端键入的所有字符都被送到Shell进程。用户可使用ps、wait及Kill命令去了解和控制后台进程的运行。





7.3 系统调用

7.3.1 系统调用的基本概念

通常，在OS的核心中都设置了一组用于实现各种系统功能的子程序(过程)，并将它们提供给应用程序调用。由于这些程序或过程是OS系统本身程序模块中的一部分，为了保护操作系统程序不被用户程序破坏，一般都不允许用户程序访问操作系统的程序和数据，所以也不允许应用程序采用一般的过程调用方式来直接调用这些过程，而是向应用程序提供了一系列的系统调用命令，让应用程序通过系统调用去调用所需的系统过程。



1. 系统态和用户态

在计算机系统中，通常运行着两类程序：系统程序和应用程序，为了保证系统程序不被应用程序有意或无意地破坏，为计算机设置了两种状态：系统态(也称为管态或核心态)和用户态(也称为目态)。操作系统在系统态运行，而应用程序只能在用户态运行。在实际运行过程中，处理机会在系统态和用户态间切换。相应地，现代多数操作系统将CPU的指令集分为特权指令和非特权指令两类。



1) 特权指令

所谓特权指令，就是在系统态时运行的指令，是关系到系统全局的指令。其对内存空间的访问范围基本不受限制，不仅能访问用户存储空间，也能访问系统存储空间，如启动各种外部设备、设置系统时钟时间、关中断、清主存、修改存储器管理寄存器、执行停机指令、转换执行状态等。特权指令只允许操作系统使用，不允许应用程序使用，否则会引起系统混乱。



2) 非特权指令

非特权指令是在用户态时运行的指令。一般应用程序所使用的都是非特权指令，它只能完成一般性的操作和任务，不能对系统中的硬件和软件直接进行访问，其对内存的访问范围也局限于用户空间。这样，可以防止应用程序的运行异常对系统造成的破坏。

这种限制是由硬件实现的，如果在应用程序中使用了特权指令，就会发出权限出错信号，操作系统捕获到这个信号后，将转入相应的错误处理程序，并将停止该应用程序的运行，重新调度。



2. 系统调用

如上所述，一方面由于系统提供了保护机制，防止应用程序直接调用操作系统的过程，从而避免了系统的不安全性。但另一方面，应用程序又必须取得操作系统所提供的服务，否则，应用程序几乎无法作任何有价值的事情，甚至无法运行。为此，在操作系统中提供了系统调用，使应用程序可以通过系统调用的方法，间接调用操作系统的相关过程，取得相应的服务。

当应用程序中需要操作系统提供服务时，如请求I/O资源或执行I/O操作，应用程序必须使用系统调用命令。由操作系统捕获到该命令后，便将CPU的状态从用户态转换到系统态，然后执行操作系统中相应的子程序(例程)，完成所需的功能。执行完成后，系统又将CPU状态从系统态转换到用户态，再继续执行应用程序。



可见，系统调用在本质上是应用程序请求OS内核完成某功能时的一种过程调用，但它是一种特殊的过程调用，它与一般的过程调用有下述几方面的明显差别：

(1) 运行在不同的系统状态。一般的过程调用，其调用程序和被调用程序都运行在相同的状态——系统态或用户态；而系统调用与一般调用的最大区别就在于：调用程序是运行在用户态，而被调用程序是运行在系统态。

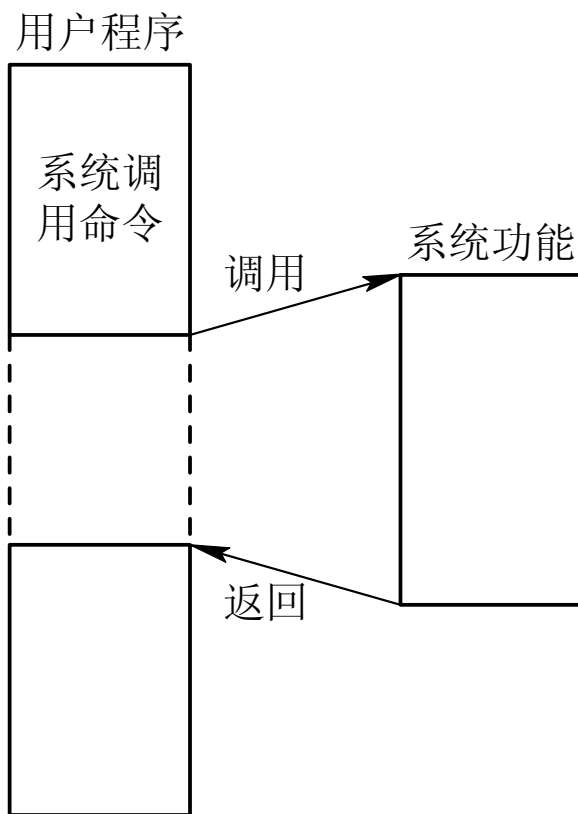


(2) 状态的转换通过软中断进入。由于一般的过程调用并不涉及到系统状态的转换，可直接由调用过程转向被调用过程。但在运行系统调用时，由于调用和被调用过程是工作在不同的系统状态，因而不允许由调用过程直接转向被调用过程。通常都是通过软中断机制，先由用户态转换为系统态，经核心分析后，才能转向相应的系统调用处理子程序。

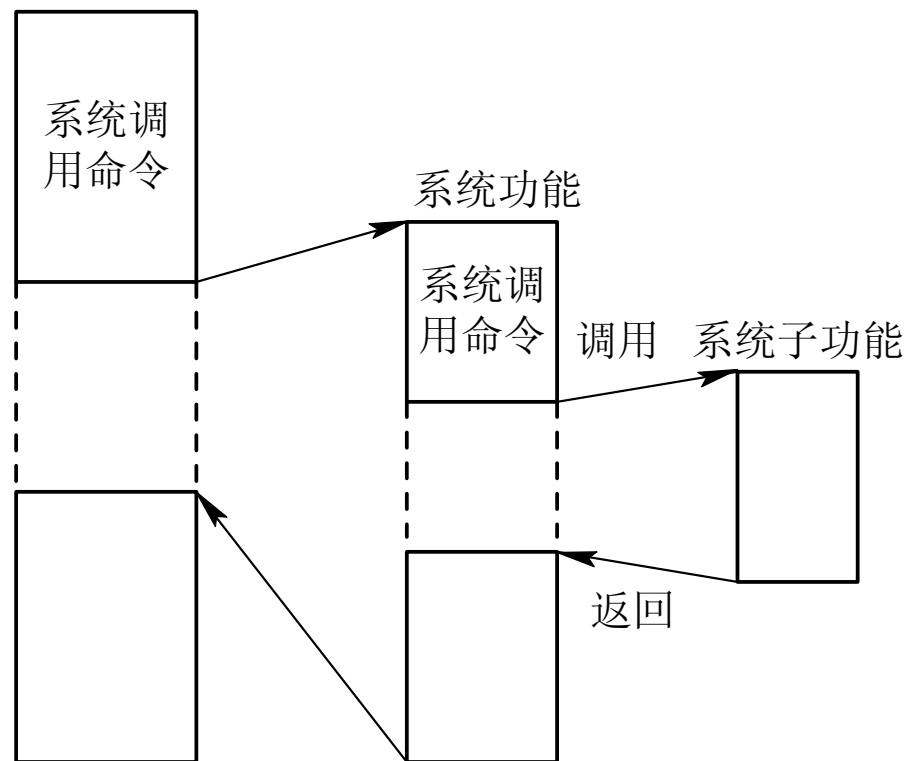
(3) 返回问题。在采用了抢占式(剥夺)调度方式的系统中，在被调用过程执行完后，要对系统中所有要求运行的进程做优先权分析。当调用进程仍具有最高优先级时，才返回到调用进程继续执行；否则，将引起重新调度，以便让优先权最高的进程优先执行。此时，将把调用进程放入就绪队列。



(4) 嵌套调用。像一般过程一样，系统调用也可以嵌套进行，即在一个被调用过程的执行期间，还可以利用系统调用命令去调用另一个系统调用。当然，每个系统对嵌套调用的深度都有一定的限制，例如最大深度为6。但一般的过程对嵌套的深度则没有什么限制。图7-3示出了没有嵌套及有嵌套的两种系统调用情况。



(a) 系统调用与返回



(b) 程序之间的嵌套调用

图7-3 系统功能的调用



3. 中断机制

系统调用是通过中断机制实现的，并且一个操作系统的所有系统调用都通过同一个中断入口来实现。如MS-DOS提供了INT 21H，应用程序通过该中断获取操作系统的服务。

对于拥有保护机制的操作系统来说，中断机制本身也是受保护的，在IBM PC上，Intel提供了多达255个中断号，但只有授权给应用程序保护等级的中断号，才是可以被应用程序调用的。对于未被授权的中断号，如果应用程序进行调用，同样会引起保护异常，而导致自己被操作系统停止。如Linux仅仅给应用程序授权了4个中断号：3、4、5以及80h，前三个中断号是提供给应用程序调试所使用的，而80h正是系统调用(system call)的中断号。



7.3.2 系统调用的类型

1. 进程控制类系统调用

这类系统调用主要用于对进程的控制，如创建一个新的进程和终止一个进程的运行，获得和设置进程属性等。

1) 创建和终止进程的系统调用

在多道程序环境下，为使多道程序能并发执行，必须先利用创建进程的系统调用来为欲参加并发执行的各程序分别创建一个进程。当进程已经执行结束时、或因发生异常情况而不能继续执行时，可利用终止进程的系统调用来结束该进程的运行。



2) 获得和设置进程属性的系统调用

当我们创建了一个(些)新进程后, 为了能控制它(们)的运行, 应当能了解、确定和重新设置它(们)的属性。这些属性包括: 进程标识符、进程优先级、最大允许执行时间等。此时, 我们可利用获得进程属性的系统调用, 来了解某进程的属性, 利用设置进程属性的系统调用, 来确定和重新设置进程的属性。



3) 等待某事件出现的系统调用

进程在运行过程中，有时需要等待某事件(条件)出现后方可继续执行。例如，一进程在创建了一个(些)新进程后，需要等待它(们)运行结束后，才能继续执行，此时可利用等待子进程结束的系统调用进行等待；又如，在客户/服务器模式中，若无任何客户向服务器发出消息，则服务器接收进程便无事可做，此时该进程就可利用等待(事件)的系统调用，使自己处于等待状态，一旦有客户发来消息时，接收进程便被唤醒，进行消息接收的处理。



2. 文件操纵类系统调用

对文件进行操纵的系统调用数量较多，有创建文件、删除文件、打开文件、关闭文件、读文件、写文件、建立目录、移动文件的读/写指针、改变文件的属性等。

1) 创建和删除文件

当用户需要在系统中存放程序或数据时，可利用创建文件的系统调用`creat`，由系统根据用户提供的文件名和存取方式来创建一个新文件；当用户已不再需要某文件时，可利用删除文件的系统调用`unlink`将指名文件删除。



2) 打开和关闭文件

用户在第一次访问某个文件之前，应先利用打开文件的系统调用`open`，将指名文件打开，即系统将在用户(程序)与该文件之间建立一条快捷通路。在文件被打开后，系统将给用户返回一个该文件的句柄或描述符；当用户不再访问某文件时，又可利用关闭文件的系统调用`close`，将此文件关闭，即断开该用户程序与该文件之间的快捷通路。



3) 读和写文件

用户可利用读系统调用`read`，从已打开的文件中读出给定数目的字符，并送至指定的缓冲区中；同样，用户也可利用写系统调用`write`，从指定的缓冲区中将给定数目的字符写入指定文件中。`read`和`write`两个系统调用是文件操纵类系统调用中使用最频繁的。



3. 进程通信类系统调用

在OS中经常采用两种进程通信方式，即消息传递方式和共享存储区方式。当系统中采用消息传递方式时，在通信前，必须先打开一个连接。为此，应由源进程发出一条打开连接的系统调用open connection，而目标进程则应利用接受连接的系统调用accept connection表示同意进行通信；然后，在源和目标进程之间便可开始通信。可以利用发送消息的系统调用send message或者用接收消息的系统调用receive message来交换信息。通信结束后，还须再利用关闭连接的系统调用close connection结束通信。



用户在利用共享存储区进行通信之前，须先利用建立共享存储区的系统调用来建立一个共享存储区，再利用建立连接的系统调用将该共享存储区连接到进程自身的虚地址空间上，然后便可利用读和写共享存储区的系统调用实现相互通信。

除上述的三类外，常用的系统调用还包括设备管理类系统调用和信息维护类系统调用，前者主要用于实现申请设备、释放设备、设备I/O和重定向、获得和设置设备属性、逻辑上连接和释放设备等功能，后者主要用来获得包括有关系统和文件的时间、日期信息、操作系统版本、当前用户以及有关空闲内存和磁盘空间大小等多方面的信息。



7.3.3 POSIX标准

目前许多操作系统都提供了上面所介绍的各种类型的系统调用，实现的功能也相类似，但在实现的细节和形式方面却相差很大，这种差异给实现应用程序与操作系统平台的无关性带来了很大的困难。为解决这一问题，国际标准化组织ISO给出的有关系统调用的国际标准POSIX1003.1 (Portable Operating System IX)，也称为“基于UNIX 的可移植操作系统接口”。

POSIX 定义了标准应用程序接口(API)，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植运行。只有符合这一标准的应用程序，才有可能完全兼容多种操作系统，即在多种操作系统下都能够运行。



POSIX 标准定义了一组过程，这组过程是构造系统调用所必须的。通过调用这些过程所提供的服务，确定了一系列系统调用的功能。一般而言，在POSIX标准中，大多数的系统调用是一个系统调用直接映射一个过程，但也有一个系统调用对应若干个过程的情形，如一个系统调用所需要的过程是其它系统调用的组合或变形时，则往往会对应多个过程。



第七章 操作系统接口

需要明确的是，POSIX标准所定义的一组过程虽然指定了系统调用的功能，但并没有明确规定系统调用是以什么形式实现的，是库函数还是其它形式。如早期操作系统的系统调用使用汇编语言编写，这时的系统调用可看成是扩展的机器指令，因而，能在汇编语言编程中直接使用。而在一些高级语言或C语言中，尤其是最新推出的一些操作系统，如UNIX新版本、Linux、Windows 和OS/2 等，其系统调用干脆用C语言编写，并以库函数形式提供，所以在用C语言编制的应用程序中，可直接通过使用对应的库函数来使用系统调用，库函数的目的是隐藏访管指令的细节，使系统调用更像过程调用。但一般地说，库函数属于用户程序而非系统调用程序。如图7-4示出了UNIX/Linux的系统程序、库函数、系统调用的层次关系。



第七章 操作系统接口

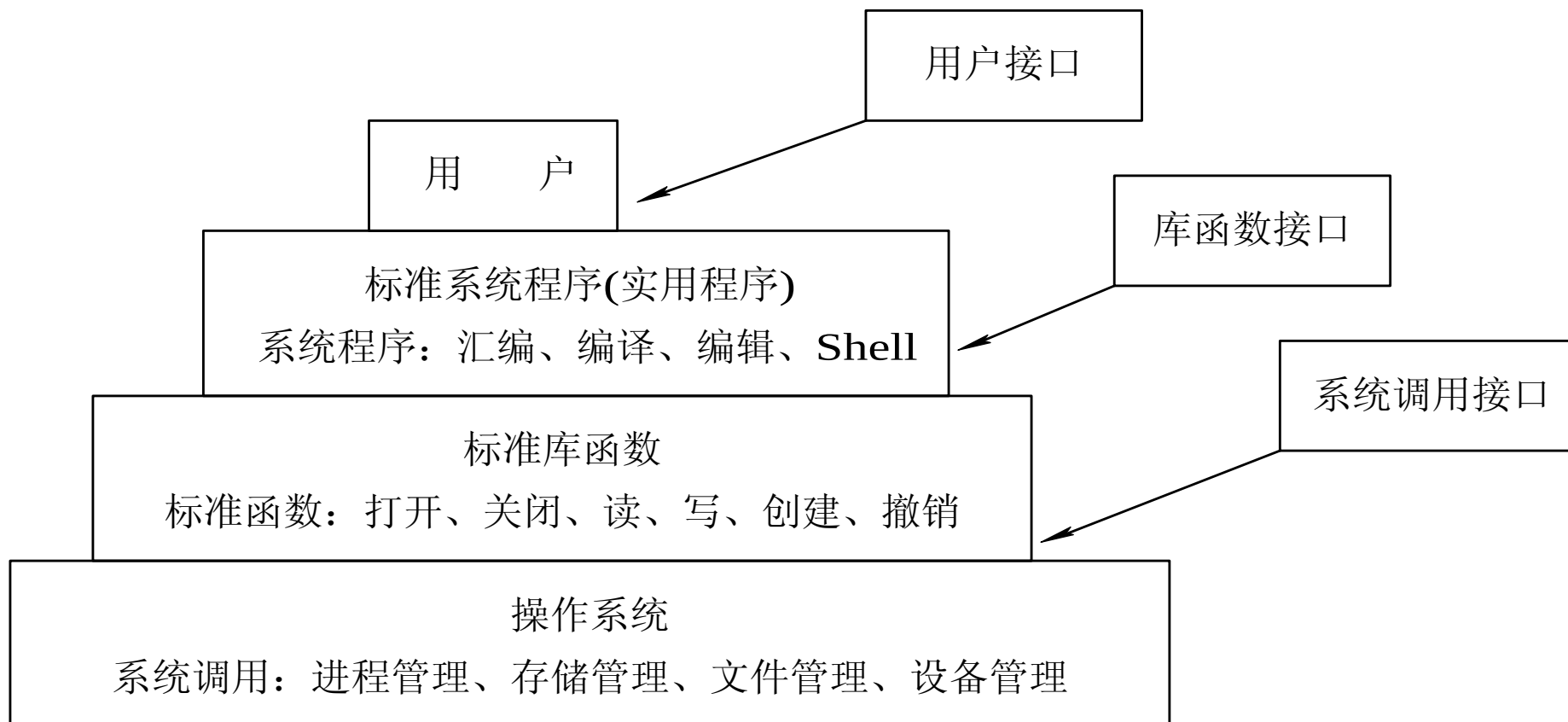


图7-4 Unix/Linux系统程序、库函数、系统调用的分层关系



7.3.4 系统调用的实现

1. 中断和陷入硬件机构

1) 中断和陷入的概念

中断是指CPU对系统发生某事件时的这样一种响应：CPU暂停正在执行的程序，在保留现场后自动地转去执行该事件的中断处理程序；执行完后，再返回到原程序的断点处继续执行。图7-5表示中断时CPU的活动轨迹。还可进一步把中断分为外中断和内中断。所谓外中断，是指由于外部设备事件所引起的中断，如通常的磁盘中断、打印机中断等；而内中断则是指由于CPU内部事件所引起的中断，如程序出错（非法指令、地址越界）、电源故障等。内中断(trap)也被译为“捕获”或“陷入”。通常，陷入是由于执行了现行指令所引起的；而中断则是由于系统中某事件引起的，该事件与现行指令无关。由于系统调用引起的中断属于内中断，因此把由于系统调用引起中断的指令称为陷入指令。

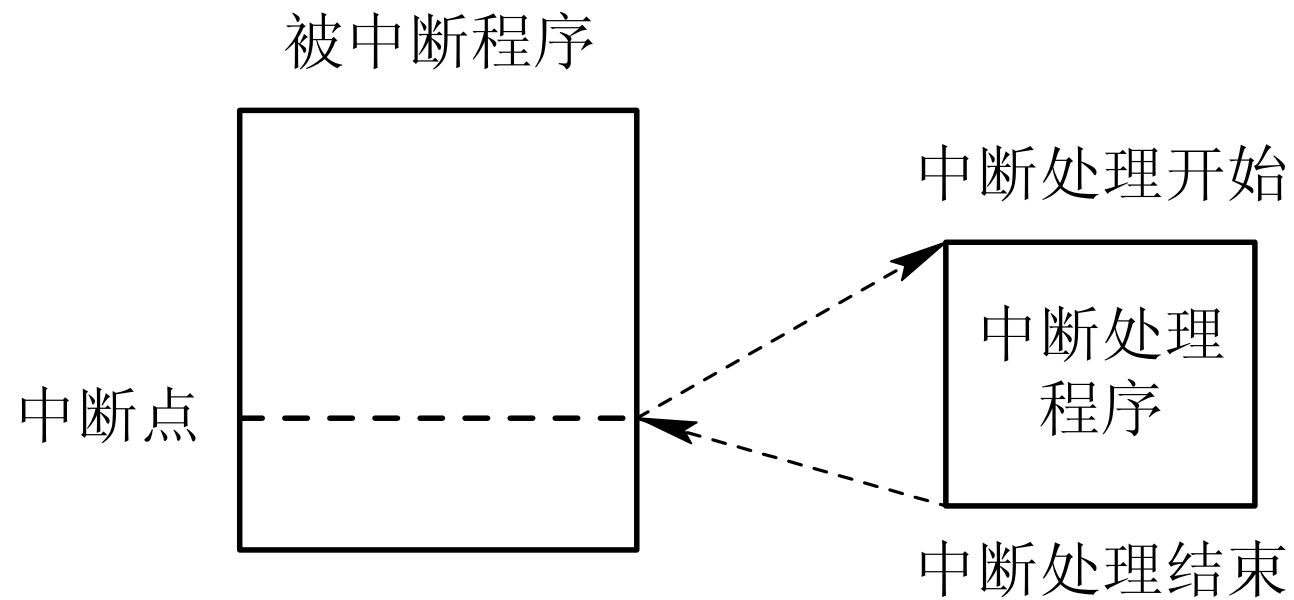


图7-5 中断时的CPU轨迹



2) 中断和陷入向量

为了处理上的方便，通常都是针对不同的设备编制不同的中断处理程序，并把该程序的入口地址放在某特定的内存单元中。此外，不同的设备也对应着不同的处理机状态字PSW，且把它放在与中断处理程序入口指针相邻接的特定单元中。在进行中断处理时，只要有了这样两个字，便可转入相应设备的中断处理程序，重新装配处理机的状态字和优先级，进行对该设备的处理。因此，我们把这两个字称为中断向量。相应地，把存放这两个字的单元称为中断向量单元。类似地，对于陷入，也有陷入向量，不同的系统调用对应不同的陷入向量，在进行陷入处理时，根据陷入指令中的陷入向量，转入实现相应的系统调用功能的子程序，即陷入处理程序。由所有的中断向量和陷入向量构成了中断和陷入向量表，如图7-6所示。



第七章 操作系统接口

中断向量单元	外设种类	优先级	中断处理程序入口地址
060	电传输出	4	klrint
064	电传输入	4	klxint
070	纸带机输入	4	perint
074	纸带机输出	4	pcpint
...	...	...	...

(a) 中断向量

陷入向量单元	陷入种类	优先级	陷入处理程序入口地址
004	总线超时	7	trap
064	非法指令	7	trap
070	电源故障	7	trap
074	trap 指令	7	trap
...	...	...	...

(b) 陷入向量

图7-6 中断向量与陷入向量



2. 系统调用号和参数的设置

往往在一个系统中设置了许多条系统调用，并赋予每条系统调用一个唯一的系统调用号。在系统调用命令(陷入指令)中把相应的系统调用号传递给中断和陷入机制的方法有很多种，在有的系统中，直接把系统调用号放在系统调用命令(陷入指令)中；如 IBM 370和早期的UNIX系统，是把系统调用命令的低8位用于存放系统调用号；在另一些系统中，则将系统调用号装入某指定寄存器或内存单元中，如MS-DOS是将系统调用号放在AH寄存器中，Linux则是利用EAX寄存器来存放应用程序传递的系统调用号。



每一条系统调用都含有若干个参数，在执行系统调用时，如何设置系统调用所需的参数，即如何将这些参数传递给陷入处理机构和系统内部的子程序(过程)，常用的实现方式有以下几种：

(1) 陷入指令自带方式。陷入指令除了携带一个系统调用号外，还要自带几个参数进入系统内部，由于一条陷入指令的长度是有限的，因此自带的只能是少量的、有限的参数。



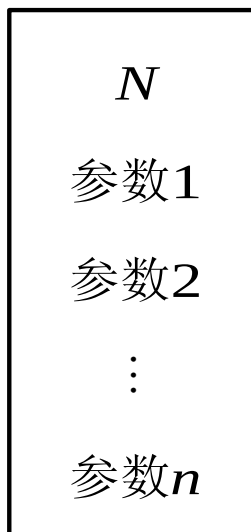
(2) 直接将参数送入相应的寄存器中。MS-DOS便是采用的这种方式，即用MOV指令将各个参数送入相应的寄存器中。系统程序和应用程序显然应是都可以访问这种寄存器的。这种方式的主要问题是由于这种寄存器数量有限，因而限制了所设置参数的数目。

(3) 参数表方式。将系统调用所需的参数放入一张参数表中，再将指向该参数表的指针放在某个指定的寄存器中。当前大多数的OS中，如UNIX系统和Linux系统，便是采用了这种方式。该方式又可进一步分成直接和间接两种方式，如图7-7所示。在直接参数方式中，所有的参数值和参数的个数N，都放入一张参数表中；而在间接参数方式中，则在参数表中仅存放参数个数和指向真正参数数据表的指针。



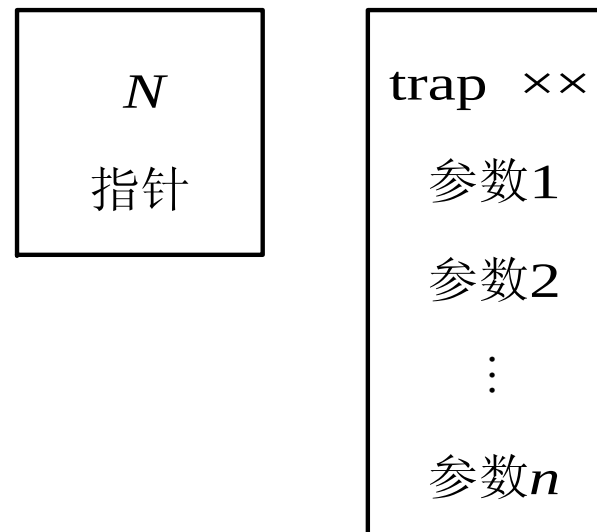
第七章 操作系统接口

变元表:



(a) 直接方式

变元表:



(b) 间接方式

图7-7 系统调用的参数形式



3. 系统调用的处理步骤

在设置了系统调用号和参数后，便可执行一条系统调用命令。不同的系统可采用不同的执行方式。在UNIX系统中，是执行CHMK命令；而在MS-DOS中则是执行INT 21软中断。

系统调用的处理过程可分成以下三步：首先，将处理机状态由用户态转为系统态；之后，由硬件和内核程序进行系统调用的一般性处理，即首先保护被中断进程的CPU环境，将处理机状态字PSW、程序计数器PC、系统调用号、用户栈指针以及通用寄存器内容等，压入堆栈；然后，将用户定义参数传送到指定的地址保存起来。



其次，是分析系统调用类型，转入相应的系统调用处理子程序。为使不同的系统调用能方便地转向相应的系统调用处理子程序，在系统中配置了一张系统调用入口表。表中的每个表目都对应一条系统调用，其中包含该系统调用自带参数的数目、系统调用处理子程序的入口地址等。因此，核心可利用系统调用号去查找该表，即可找到相应处理子程序的入口地址而转去执行它。

最后，在系统调用处理子程序执行完后，应恢复被中断的或设置新进程的CPU现场，然后返回被中断进程或新进程，继续往下执行。



4. 系统调用处理子程序的处理过程

系统调用的功能主要是由系统调用子程序来完成的。对于不同的系统调用，其处理程序将执行不同的功能。我们以一条在文件操纵中常用的Creat命令为例来说明之。



第七章 操作系统接口

进入Creat的处理子程序后，核心将根据用户给定的文件路径名Path，利用目录检索过程去查找指定文件的目录项。查找目录的方式可以用顺序查找法，也可用Hash查找法。如果在文件目录中找到了指定文件的目录项，表示用户要利用一个已有文件来建立一个新文件。但如果在该已有(存)文件的属性中有不允许写属性，或者创建者不具有对该文件进行修改的权限，便认为是出错而做出错处理；若不存在访问权限问题，便将已存文件的数据盘块释放掉，准备写入新的数据文件。如未找到指名文件，则表示要创建一个新文件，核心便从其目录文件中找出一个空目录项，并初始化该目录项，包括填写文件名、文件属性、文件建立日期等，然后将新建文件打开。





7.4 UNIX系统调用

7.4.1 UNIX系统调用的类型

1. 进程控制

该类系统调用包括：创建进程的系统调用fork、终止进程的系统调用exit、等待子进程结束的系统调用wait等十多条。

(1) 创建进程(fork)。一个进程可以利用fork系统调用来创建一个新进程。新进程是作为调用者的子进程，它继承了其父进程的环境、已打开的所有文件、根目录和当前目录等，即它继承了父进程几乎所有的属性，并具有与其父进程基本上相同的进程映像。



(2) 终止进程(exit)。一个进程可以利用exit实现自我终止。通常，在父进程创建子进程时，便在子进程的末尾安排一条exit系统调用。这样，子进程在完成规定的任务后，便可进行自我终止。子进程终止后，留下一记账信息status，其中包含了子进程运行时记录下来的各种统计信息。

(3) 等待子进程结束(wait)。wait用于将调用者进程自身挂起，直至它的某一子进程终止为止。这样，父进程可以利用wait使自身的执行与子进程的终止同步。



(4) 执行一个文件(exec)。exec可使调用者进程的进程映像(包括用户程序和数据等)被一个可执行的文件覆盖,此即改变调用者进程的进程映像。该系统调用是UNIX系统中最复杂的系统调用之一。

(5) 获得进程ID。UNIX系统提供了一组用于获得进程标识符的系统调用,比如,可利用getp-id系统调用来获得调用进程的标识符,利用getpgrp系统调用来获得调用进程的进程组ID,以及利用getppid系统调用来获得调用进程的父进程ID等。



(6) 获得用户ID。UNIX系统提供了一组用于获得用户ID的系统调用，如getuid可用于获得真正的用户ID，geteuid用于获得有效用户ID，getgid用于获得真正用户组ID等。

(7) 进程暂停(pause)。可用此系统调用将调用进程挂起，直至它收到一个信号为止。



2. 文件操纵

(1) 创建文件(creat)。系统调用creat的功能是根据用户提供的文件名和许可权方式，来创建一个新文件或重写一个已存文件。如果系统中不存在指名文件，核心便以给定的文件名和许可权方式来创建一个新文件；如果系统中已有同名文件，核心便释放其已有的数据块。创建后的文件随即被打开，并返回其文件描述符fd。若creat执行失败，便返回“-1”。



(2) 打开文件(open)。设置系统调用open的目的，是为了方便用户及简化系统的处理。open的功能是把有关的文件属性从磁盘拷贝到内存中，以及在用户和指名文件之间建立一条快捷的通路，并给用户返回一个文件描述符fd。文件被打开后，用户对文件的任何操作都只须使用fd而非路径名。



(3) 关闭文件(close)。当把一个文件用毕且暂不访问时，可调用close将文件关闭，即断开用户程序与该文件之间已经建立的快捷通路。在UNIX系统中，由于允许一个文件被多个进程所共享，故只有在无其他任何进程需要此文件时，或者说，在对其索引结点中的访问计数i-count执行减1操作后其值为0，表示已无进程再访问该文件时，才能真正关闭该文件。

(4) 读和写文件read和write。仅当用户利用open打开指定文件后，方可调用read或write对文件执行读或写操作。两个系统调用都要求用户提供三个输入参数：① 文件描述符fd。



② buf缓冲区首址。对读而言，这是用户所要求的信息传送的目标地址；对写而言，这则是信息传送的源地址。③ 用户要求传送的字节数n byte。

系统调用read的功能是试图从fd所指示的文件中去读入n byte个字节的数据，并将它们送至由指针buf所指示的缓冲区中；系统调用write的功能是试图把n byte个字节数据，从指针buf所指示的缓冲区中写到由fd所指向的文件中。



(5) 连接和去连接(link和unlink)。为了实现文件共享，必须记住所有共享该文件的用户数目。为此，在该文件的索引结点中设置了一个连接计数*i.link*。每当有一用户要共享某文件时，须利用系统调用link来建立该用户(进程)与此文件之间的连接，并对*i.link*做加1操作。当用户不再使用此文件时，应利用系统调用unlink去断开此连接，亦即做*i.link*的减1操作。当*i.link*减1后结果为0时，表示已无用户需要此文件，此时才能将该文件从文件系统中删除。故在UNIX系统中并无一条删除文件的系统调用。



3. 进程间的通信

为了实现进程间的通信，在UNIX系统中提供了一个用于进程间通信的软件包，简称IPC。它由消息机制、共享存储器机制和信号量机制三部分组成。在每一种通信机制中，都提供了相应的系统调用供用户程序进行进程间的同步与通信之用。

(1) 消息机制。用户(进程)在利用消息机制进行通信时，必须先利用msgget系统调用来建立一个消息队列。若成功，便返回消息队列描述符msgid，以后用户便可利用msgid去访问该消息队列。用户(进程)可利用发送消息的系统调用msgsend向用户指定的消息队列发送消息；利用msgrcv系统调用从指定的消息队列中接收指定类型的消息。



(2) 共享存储器机制。当用户(进程)要利用共享存储器机制进行通信时,必须先利用shmget系统调用来建立一个共享存储区,若成功,便返回该共享存储区描述符shmid。以后,用户便可利用shmid去访问该共享存储区。进程在建立了共享存储区之后,还必须再利用shmat将该共享存储区连接到本进程的虚地址空间上。以后,在进程之间便可利用该共享存储区进行通信。当进程不再需要该共享存储区时,可利用shmdt系统调用来拆除进程与共享存储区间的连接。

(3) 信号量机制。在UNIX系统中所采用的信号量机制,与第二章中所介绍的一般信号量集机制相似,允许将一组信号量形成一个信号量集,并对这组信号量施以原子操作,详见第十章。



4. 信息维护

在UNIX系统中，设置了许多条用于系统维护的系统调用。

(1) 设置和获得时间。超级用户可利用设置时间的系统调用(stime)，来设置系统的日期和时间。如果调用进程并非超级用户，则stime失败。一般用户可利用获得时间的系统调用time来获得当前的日期和时间。

(2) 获得进程和子进程时间(times)。利用该系统调用可获得进程及其子进程所使用的CPU时间，其中包括调用进程在用户空间执行指令所花费的时间，系统为调用进程所花费的CPU时间、子进程在用户空间所用的CPU时间、系统为各子进程所花费的CPU时间等，并可将这些时间填写到一个指定的缓冲区。



(3) 设置文件访问和修改时间(utime)。该系统调用用于设置指名文件被访问和修改的时间。如果该系统调用的参数times为NULL时, 文件主和对该文件具有写权限的用户, 可将对该文件的访问和修改时间设置为当前时间; 如果times不为NULL, 则把times解释为指向utim buf结构的指针, 此时, 文件主和超级用户能将访问时间和修改时间置入utim buf结构中。

(4) 获得当前UNIX系统的名称(uname)。利用该系统调用可将有关UNIX系统的信息存储在utsname结构中。这些信息包括UNIX系统名称的字符串、系统在网络中的名称、硬件的标准名称等。



7.4.2 被中断进程的环境保护

1. CPU环境保护

当用户程序处在用户态，且在执行系统调用命令(即CHMK命令)之前，应在用户空间提供系统调用所需的参数表，并将该参数表的地址送入 R_0 寄存器。在执行CHMK命令后，处理机将由用户态转为核心态，并由硬件自动地将处理机状态长字(PSL)、程序计数器(PC)和代码操作数(code)压入用户核心栈，继而从中断和陷入向量表中取出trap.S的入口地址，然后便转入中断和陷入总控程序trap.S中执行。



trap.S程序执行后，继续将陷入类型type和用户栈指针usp压入用户核心栈，接着还要将被中断进程的CPU环境中的一系列寄存器如R0~R11的部分或全部内容压入栈中。至于哪些寄存器的内容要压入栈中，这取决于特定寄存器中的屏蔽码，该屏蔽码的每一位都与R₀~R₁₁中的一个寄存器相对应。当某一位置成1时，表示对应寄存器的内容应压入栈中。



2. AP和FP指针

为了实现系统调用的嵌套使用，在系统中还设置了两个指针，其一是系统调用参数表指针AP，用于指示正在执行的系统调用所需参数表的地址，通常是把该地址放在某个寄存器中，例如放在R12中；再者，还须设置一个调用栈帧指针。所谓调用栈帧(或简称栈帧)，是指每个系统调用需要保存而被压入用户核心栈的所有数据项；而栈帧指针FP，则是用于指示本次系统调用所保存的数据项。每当出现新的系统调用时，还须将AP和FP压入栈中。图7-8示出了在trap.S总控程序执行后，用户核心栈的情况。

当trap.S完成被中断进程的CPU环境和AP及FP指针的保存后，将会调用由C语言书写的公共处理程序trap.C，以继续处理本次的系统调用所要完成的公共处理部分。



由中断和
陷入总控
程序压入

陷入时由
硬件压入

AP
FP
R_0
$\vdots$
R_n
usp
type
code
PC
PSL

用户核心栈

图7-8 用户核心栈



7.4.3 系统调用陷入后需处理的公共问题

1. 确定系统调用号

由上所述得知，在中断和陷入发生后，是先经硬件陷入机构予以处理，再进入trap.S，然后再调用trap.C继续处理。其调用形式为：

trap(usp, type, code, PC, PSL)



第七章 操作系统接口

其中，参数PSL为陷入时处理机状态字长，PC为程序计数器，code为代码操作数，type为陷入类型号，usp为用户栈指针。对陷入的处理可分为多种情况，如果陷入是由于系统调用所引起的，则对此陷入的第一步处理，便是确定系统调用号。通常，系统调用号是包含在代码操作数中，故可利用code来确定系统调用号i。其方法是令

$$i = \text{code} \ \& \ 0377$$

若 $0 < i < 64$ ，此i便是系统调用号，可根据系统调用号i和系统调用定义表，转向相应的处理子程序。若 $i = 0$ ，则表示系统调用号并未包含在代码操作数中，此时应采用间接参数方式，利用间接参数指针来找到系统调用号。



2. 参数传送

这是对因系统调用引起的陷入的第二步处理。参数传送是指由trap.C程序将系统调用参数表中的内容，从用户区传送到User结构的U.U-arg中，供系统调用处理程序使用。由于用户程序在执行系统调用命令之前，已将参数表的首址放入R0寄存器中，在进入trap.C程序后，该程序便将该首址赋予U.U-arg指针，因此，trap.C在处理参数传送时，可读取该指针的内容，以获得用户所提供的参数表，并将之送至U.U-arg中。应当注意，对于不同的系统调用，所需传送参数的个数并不相同，trap.C程序应根据在系统调用定义表中所规定的参数个数来进行传送，最多允许10个参数。



3. 利用系统调用定义表转入相应的处理程序

在UNIX系统中，对于不同(编号)的系统调用，都设置了与之相应的处理子程序。为使不同的系统调用能方便地转入其相应的处理子程序，也将各处理子程序的入口地址放入了系统调用定义表即`Sysent[]`中。该表实际上是一个结构数组，在每个结构中包含三个元素，其中第一个元素是相应系统调用所需参数的个数；第二个元素是系统调用经寄存器传送的参数个数；第三个元素是相应系统调用处理子程序的入口地址。在系统中设置了该表之后，便可根据系统调用号*i*从系统调用定义表中找出相应的表目，再按照表目中的入口地址转入相应的处理子程序，由该程序去完成相应系统调用的特定功能。在该子程序执行完后，仍返回到中断和陷入总控程序中的`trap.C`程序中，去完成返回到断点前的公共处理部分。

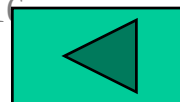


4. 系统调用返回前的公共处理

在UNIX系统中，进程调度的主要依据是进程的动态优先级。随着进程执行时间的加长，其优先级将逐步降低。每当执行了系统调用命令、并由系统调用处理子程序返回到trap.C后，都将重新计算该进程的优先级；另外，在系统调用执行过程中，若发生了错误使进程无法继续运行时，系统会设置再调度标志。处理子程序在计算了进程的优先级后，便去检查该再调度标志是否已又被设置。若已设置，便调用switch调度程序，再去从所有的就绪进程中选择优先级最高的进程，把处理机让给该进程去运行。



UNIX系统规定，当进程的运行是处于系统态时，即使再有其他进程又发来了信号，也不予理睬；仅当进程已从系统态返回到用户态时，内核才检查该进程是否已收到了由其他进程发来的信号。若有信号，便立即按该信号的规定执行相应的动作。在从信号处理程序返回后，还将执行一条返回指令RET，该指令将把已被压入用户核心栈中的所有数据(如PSL、PC、FP及AP等)都退还到相应的寄存器中，这样，控制就将从系统调用返回到被中断进程，后者继续执行下去。





7.5 图形用户接口

7.5.1 图形化用户界面

图形化用户界面(GUI)是近年来最为流行的联机用户接口形式，并已制定了国际GUI标准。20世纪90年代推出的主流操作系统都提供了GUI。1981年，Xerox 公司在Star 8010工作站操作系统中，首次推出了图形用户接口。1983年，Apple公司又在Apple Lisa 机和Macintosh 机上的操作系统中成功使用了GUI。之后，还有Microsoft 公司的Windows，IBM 公司的OS/2，UNIX 和Linux 使用的X-Window都使用了GUI。



GUI采用了图形化的操作界面，使用WIMP 技术，将窗口(Window)、图标(Icon)、菜单(Menu)、鼠标(Pointing device)和面向对象技术等集成在一起，引入形象的各种图符将系统的各项功能、各种应用程序和文件，直观、逼真地表示出来，形成一个图文并茂的视窗操作环境。用户可以轻松地通过选择窗口、菜单、对话框和滚动条完成对他们作业和文件的各种控制与操作。



以Microsoft公司的Windows系列操作系统为例，在系统初始化后，操作系统为终端用户生成了一个运行explorer.exe的进程，它运行一个具有窗口界面的命令解释程序，该窗口为一个特殊的窗口，即桌面。在“开始”菜单中罗列了系统的各种应用程序，点击某个程序，则解释程序会产生一个新进程，由新进程弹出一个新窗口，并运行该应用程序，该新窗口的菜单栏或图符栏会显示应用程序的子命令。用户可进一步选择并点击子命令，如果该子命令需要用户输入参数，则会弹出一个对话框，指导用户进行命令参数的输入，完成后用户点击“确定”按钮，命令即进入执行处理过程。



在Windows系统中，采用的是事件驱动控制方式，用户通过动作来产生事件以驱动程序工作。事件实质就是发送给应用程序的一个消息，用户按键或点击鼠标等动作都会产生一个事件，通过中断系统引出事件驱动控制程序工作，对事件进行接收、分析、处理和清除。各种命令和系统中所有的资源，如文件、目录、打印机、磁盘、各种系统应用程序等都可以定义为一个菜单、一个按钮或一个图标，所有的程序都拥有窗口界面，窗口中所使用的滚动条、按钮、编辑框、对话框等各种操作对象都采用统一的图形显示方式和操作方法。用户可以通过鼠标(或键盘)点击操作选择所需要的菜单、图标或按钮，从而达到控制系统、运行某个程序、执行某个操作(命令)的目的。



7.5.2 桌面、图标和任务栏

1. 桌面与图标的初步概念

在运行Windows时，其操作都是在桌面进行的。所谓桌面，是指整个屏幕空间，即在运行Windows时用户所看到的屏幕。该桌面是由多个任务共享的。为了避免混淆，每个任务都通过各自的窗口显示其操作和运行情况，因此，Windows允许在桌面上同时出现多个窗口。所谓窗口，是指屏幕上的一块矩形区域。应用程序(包括文档)可通过窗口向用户展示出系统所能提供的各种服务及其需要用户输入的信息；用户可通过窗口中的图标去查看和操纵应用程序或文档。



在面向字符的窗口中，并不提供图标。在面向图形的窗口中，图标也是作为图形用户接口中的一个重要元素。所谓图标，是代表一个对象的小图像，如代表一个文件夹或程序的图标，它是最小化的窗口。当用户暂时不用某窗口时，可利用鼠标去点击最小化按钮，即可将该窗口缩小为图标；而通过对该图标双击的操作，又可将之恢复为窗口。



2. 桌面上常见的图标

随着计算机设置的不同，在启动Windows时，在桌面左边也会出现一些不同的图标。在Windows中文版的桌面上(见图7-9)比较常见的图标有以下几个：

(1) “我的电脑”。双击此图标后，桌面上将出现“我的电脑”窗口，并在窗口中会显现出用户计算机的所有资源。

(2) “回收站”。该图标用于暂存用户所删除的文件及文件夹，以便在需要时将之恢复。



第七章 操作系统接口

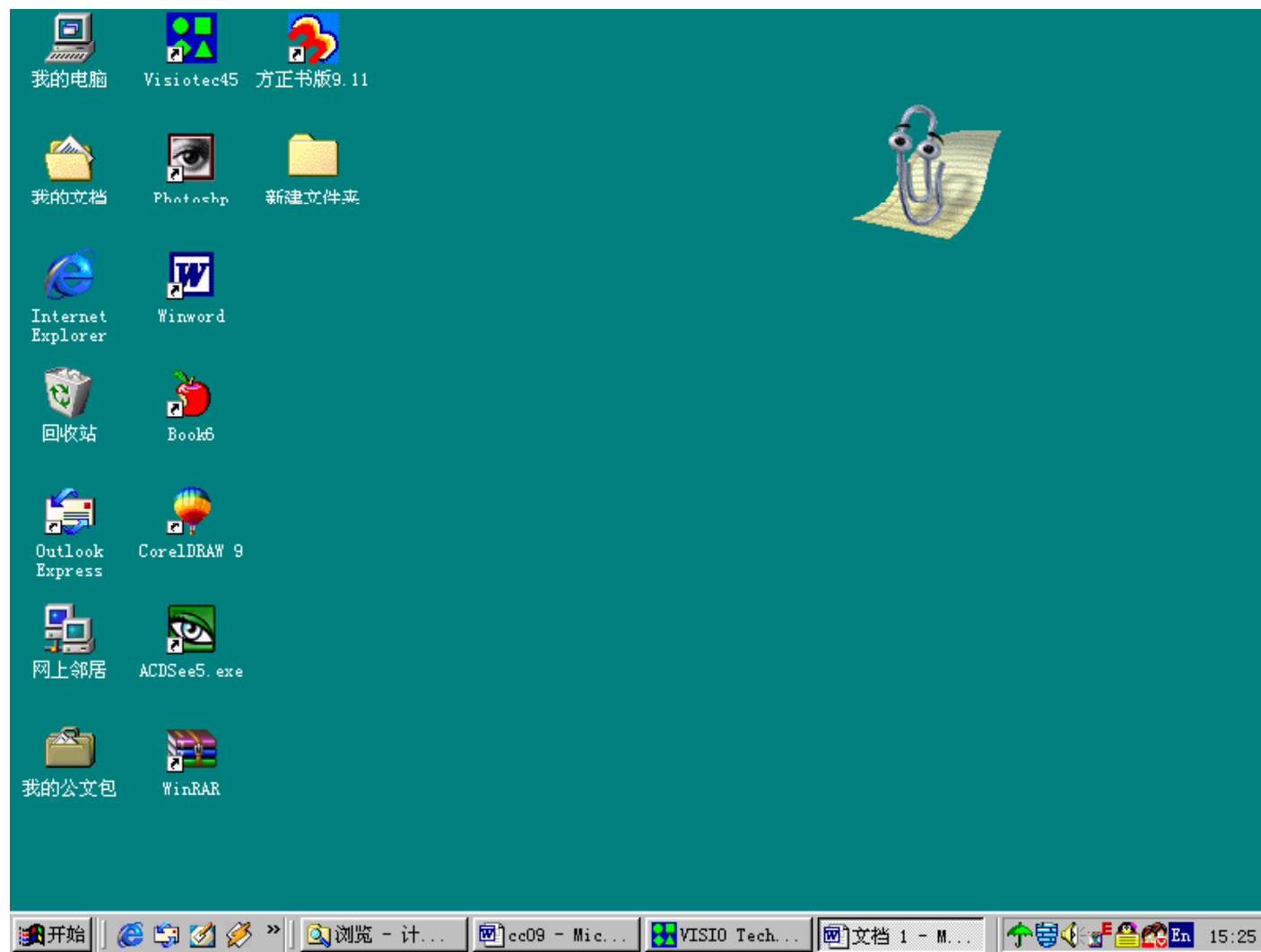


图7-9 桌面与图标



(3) “我的文档”。该图标用于供用户存放自己建立的文件夹和文件。

(4) “Internet Explorer”(简称为IE)。这是Microsoft公司开发的WWW浏览器。在用户电脑与Internet服务提供商ISP连接成功后, 便可通过双击IE图标, 实现对Internet中网页信息的浏览。

(5) “Outlook Express”。这是Microsoft公司推出的电子邮件应用软件。在用户电脑与ISP连接成功后, 可再双击Outlook Express图标, 以进一步连接ISP的电子邮件服务器。连接成功后, 便可发送电子邮件和接收电子邮件。



(6) “网上邻居”。如果用户的电脑已连接到局域网上，那么用户便可通过该图标方便地使用局域网中其它计算机上可共享的资源。

(7) 收件箱。该软件用于发送或接收远程电子邮件和传真。

(8) “我的公文包”。在用户携带便携机出差期间，无论是便携机中的文件，还是办公室计算机中的文件，都可能发生变化，用户在返回单位后，应尽快使两者保持一致(称之为“同步”)。“我的公文包”软件便是用于保持两者(便携式电脑和台式电脑中的文件)同步的软件。



3. “开始”按钮和任务栏

在Windows桌面的下方，一般都设置了“开始”按钮和任务栏，并作为系统的默认设置。只要Windows在运行，在屏幕下方即可见到它。

(1) “开始”按钮。“开始”按钮位于任务栏的左边。当用鼠标的左键单击“开始”按钮时，便可打开一个开始菜单，其中包括了用户常用的工具软件 and 应用程序，如程序选项、文档选项、设置选项等。因此，用户会经常使用“开始”按钮来运行一个程序。如果用右键单击“开始”按钮，将打开一个快捷菜单，其中包括“资源管理器”选项。此外，在关闭机器之前，应先关闭Windows，此时同样是单(左)击“开始”按钮，然后再单击菜单中的“关闭系统”选项。



(2) 任务栏。设置任务栏的目的是帮助用户快速启动常用的程序，方便地切换当前的程序。因此，在任务栏中包含若干个常用的应用程序小图标，如用于实现英文输入或汉字拼音输入等的小图标、控制音量大小的小图标、查看和改变系统时钟的小图标等。

为了便于任务之间的切换，凡曾经运行过且尚未关闭的任务，在任务栏中都有其相应的小图标。因此，如果用户希望运行其中的某个程序，只需单击代表该程序的小图标，该程序的窗口便可显现在屏幕上。应用程序之间的切换就像看电视时的频道切换一样简单。



(3) 任务栏的隐藏方式。任务栏在桌面中所占的大小可根据用户需要进行调整。任务栏可以始终完整地显现在屏幕上，不论窗口是如何切换或移动，都不能把任务栏覆盖掉。当然，这样一来任务栏将占用一定的可用屏幕空间。如果用户希望尽可能拓宽屏幕的可用空间，也可选用任务栏的隐藏方式，这时，任务栏并未真正被消除，只是暂时在屏幕上看不见，相应地，在屏幕底部会留下一条白线，当用户又想操作任务栏时，只需将鼠标移到此白线上，任务栏又会立即显现出来，当鼠标离开该线后，任务栏又会隐藏起来。



(4) 任务子栏。在Windows的任务栏中，可以增加若干个任务子栏。例如，增加“地址子栏”后，可在其中存放许多地址，如文件夹名、局域网上某计算机的地址、WWW地址等；又如“桌面”子栏，用于显示当前桌面上的组件(“我的电脑”、“我的公文包”等)。任务子栏可以不同的形式放在桌面上，如可利用鼠标将某子栏从桌面上拖出，形成一个独立的窗口，也可将某子栏拖至桌面的边缘，系统会自动地将它变为一个独立的工具栏。



7.5.3 窗口

1. 窗口的组成

在熟练使用Windows之前，必须先了解其窗口的组成，即了解组成窗口的各元素。图7-10 示出了Windows XP的一个典型窗口，在该窗口中包括如下诸元素：

(1) 标题栏和窗口标题。标题栏是位于窗口顶行的横条，其中含有窗口标题，即窗口名称，如“我的电脑”、“我的文档”、“控制面板”等。



第七章 操作系统接口



图7-10 “我的电脑”窗口的组成



(2) 菜单栏。通常，菜单栏都在窗口标题栏的下面，以菜单条的形式出现。在菜单条中列出了可选的各菜单项，用于提供各类不同的操作功能，比如在“我的电脑”窗口的菜单条中，有文件(F)、编辑(E)、查看(V)等菜单项。

(3) 工具栏。工具栏位于菜单栏的下方。其内容是各类可选工具，或说它由许多命令按钮组成，每一个按钮代表一种工具。例如，我们可利用删除命令按钮来删除一个文件或文件夹；可利用属性命令按钮来查看文件(夹)的属性，包括文件(夹)的类型、大小，在文件夹中包含多少文件和文件的大小等。



(4) 控制菜单按钮。它位于窗口标题的左端。可用它打开窗口的控制菜单，在菜单中有用于实现窗口最大化、最小化、关闭等操作的选项按钮。

(5) 最大化、最小化和关闭按钮。在窗口标题栏的右边有三个按钮，单击其中间的最大化按钮，可把窗口放大到最大(占据整个桌面)；当窗口已经最大化时，最大化按钮就变成还原按钮，单击之，又可将窗口还原为原来的大小；单击左边的最小化按钮，可将窗口缩小成图标；如需关闭该窗口，可单击关闭按钮。



(6) 滚动条。当窗口的大小不足以显示出整个文件(档)的内容时,可使用位于窗口底部或右边的滚动块(向右或向下移动),以观察该文件(档)中的其余部分。

(7) 窗口边框。界定窗口周边的网条边被称为窗口边框。用鼠标移动一条边框的位置可改变窗口的大小;也可利用鼠标去移动窗口的一个角,来同时改变窗口两个边框的位置,以改变窗口的位置和大小。

(8) 工作区域。窗口内部的区域称为工作区域。



2. 窗口的性质

1) 窗口的状态

当用户双击图标A而打开相应的窗口A时，该窗口便处于激活状态。此时用户可以看见窗口A中的所有元素，且窗口的标题条呈高亮度蓝色。被激活窗口的应用程序在前台运行，它能接收用户键入的信息。如果用户再双击图标B而打开窗口B时，窗口B又处于激活状态。此时窗口A则转为非激活状态，且窗口A被窗口B所覆盖。我们把窗口虽然已被打开，但却是处于非激活状态的称为打开状态。在Windows的桌面上，允许同时有多个处于打开状态的窗口，但其中只能有一个窗口处于激活状态，亦即，仅有一个应用程序在前台运行，其余的程序都在后台运行。



2) 窗口的改变

用户可用鼠标来改变窗口的大小及其在桌面上的位置。因此，既可用鼠标来拖拽窗口边框或窗口角，以改变窗口的大小，又可利用最大化和最小化按钮或控制菜单，来将窗口最大化或最小化。



7.5.4 对话框

1. 对话框的用途

对话框是在桌面上带有标题条、输入框和按钮的一个临时窗口，也称为对话窗口。虽然对话框与窗口有些相似，但也有明显差别，主要表现为：在所有对话框上都没有工具栏，而且对话框的大小是固定不变的，因而也没有其相应的最大化和最小化按钮；对话框也不能像窗口那样用鼠标拖拽其边框或窗口角来改变其大小和位置；此外，对话框是临时窗口，用完后便自动消失，或用取消命令将它消除。



对话框的主要用途是实现人—机对话，即系统可通过对话框提示用户输入与任务有关的信息，比如提示用户输入要打开文件的名称、其所在目录、所在驱动器及文件类型等信息；或者供用户对对象的属性、窗口等的环境进行重新设置，比如设置文件的属性、设置显示器的颜色和分辨率、设置桌面的显示效果；还可以提供用户可能需要的信息等。

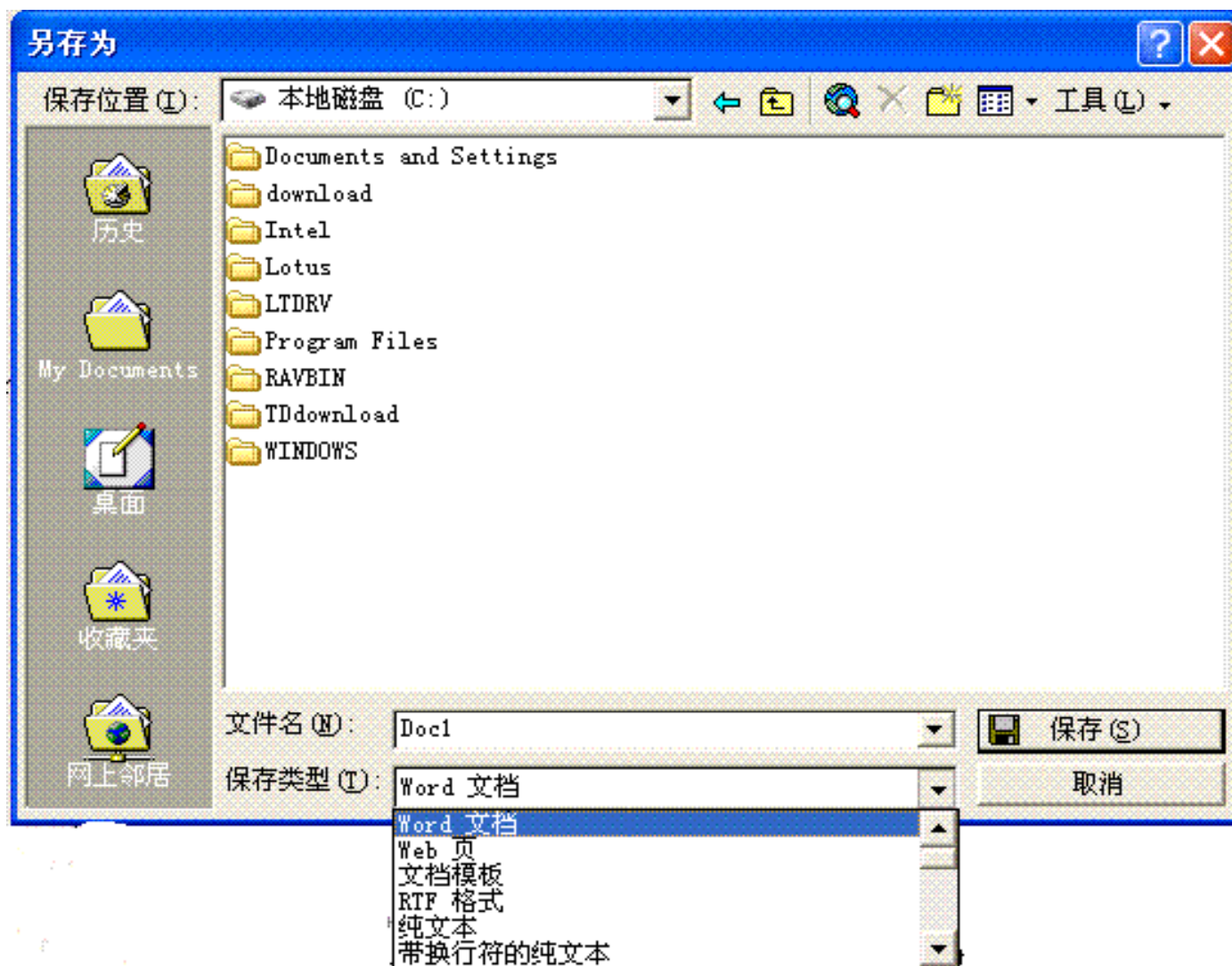


2. 对话框的组成

Windows的对话框可由以下几个元素组成，其中有的简单，有的复杂。图7-11(a)、(b)分别示出了Windows XP的“另存为”和“格式化”两个常用的对话框。



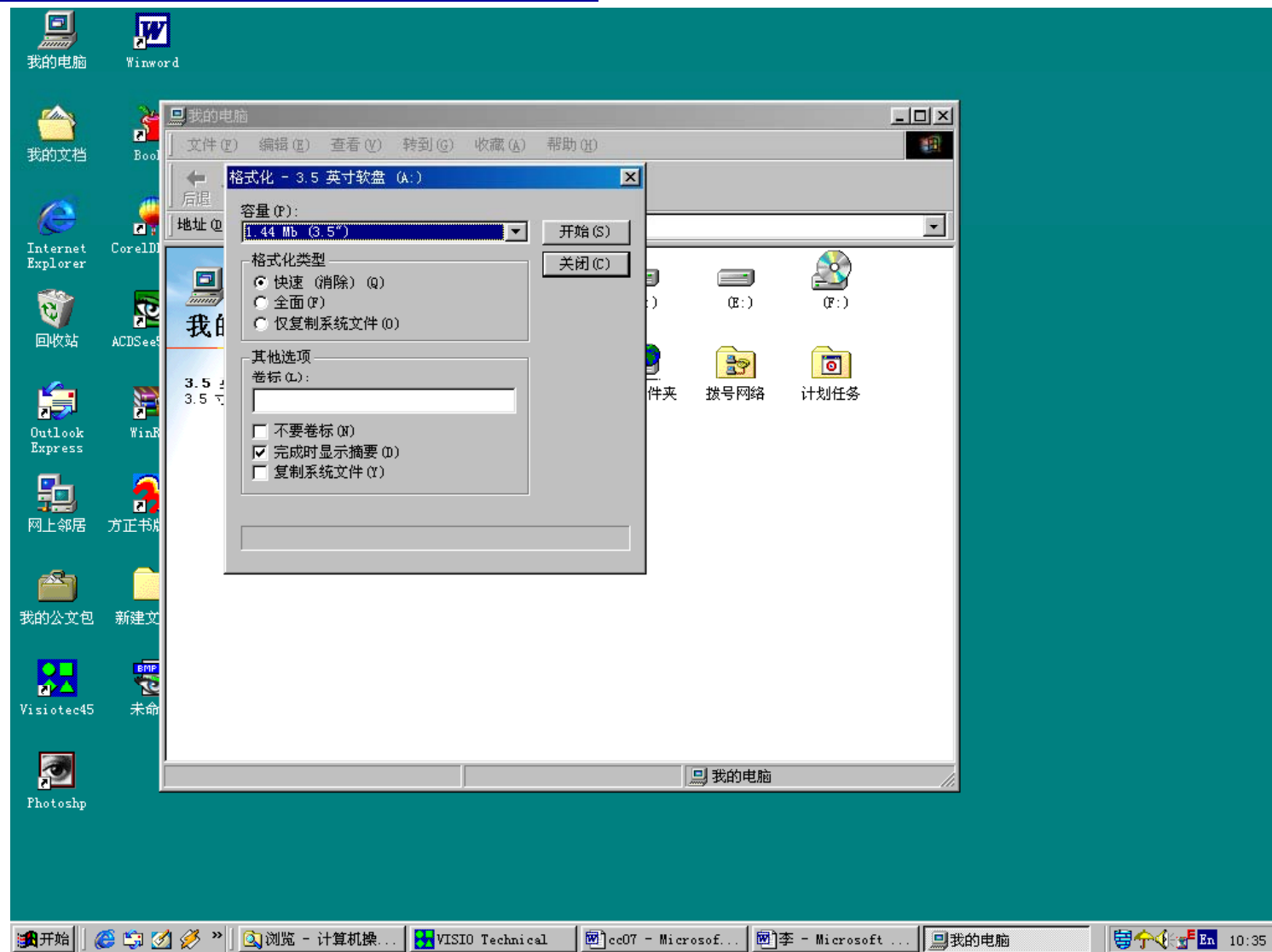
第七章 操作系统接口



(a) “另存为” 对话框

图 7 -11 对话框

第七章 操作系统接口



(b) “格式化” 对话框

图 7 -11 对话框



1) 标题栏

如同窗口一样，对话框的标题栏也是位于其顶部，其中，左边部分为对话框名称(如名称为“显示属性”)，右边部分是关闭按钮和帮助按钮。



2) 输入框

输入框可分为两类：一类是文本框，是一个供用户输入文本信息的矩形框，用户可通过键盘向文本框内输入任何符合要求的字符串。见图7-11(a)中的文件名文本框。

另一类是列表框。在列表框中为用户提供参考信息供用户选择，但用户不能对列表框中的内容进行修改。列表框有三种形式：第一种是简单列表框，需要显示的内容全部列于该框中；第二种是滚动式列表框，在框的右边框处有一滚动条(滑块)，可用来查看该框中未显示部分的内容；第三种是下拉式列表框，在框中仅有一行文字(一个选项)，其右边有一个朝下的三角形符号，对它单击后，可弹出一个下拉式列表供用户选择。图7-11(a)中的是“保存类型”的下拉式列表框。



3) 按钮

在Windows中，提供了多种形式的按钮，如命令按钮、选择按钮、滑动式按钮、数字式增减按钮等。

(1) 命令按钮。可用该按钮来启动一个立即响应的动作，如“确定”按钮、“取消”按钮、“关闭”按钮、“开始”按钮等。命令按钮通常是含有文字的矩形按钮，在对话框的底部或右部。

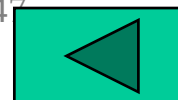


(2) 选择按钮。它又可分为单选按钮和复选按钮两种。前者是指在同一组的多个单选按钮中，每次必须且只能选择其一。单选按钮为圆形。当某一选项被选中时，该圆形按钮中会增加一个同心圆点，见图7-11(b)中的“格式化类型”下面的三个单选按钮。复选按钮是指用户可根据需要在多个复选按钮中选择其中一个或多个按钮。复选按钮呈方框形，如被选中，相应方框中会出现“√”标记，见图7-11(b)中“其他选项”下面的几个复选按钮。



(3) 滑块式按钮。某些对象的属性是可在一定范围内进行连续调节的，比如鼠标被双击的速度、键盘的重复速率、音响音量的调节等。

(4) 数字式增减按钮。有些属性已被数字化，且可在一定范围内调节。如在“日期/时间”属性中，便有一对用于改变时间的数字式增减按钮。在上述两种按钮上，都有三角形箭头标记，对箭头向上的标记单击时，可使数字增加，单击箭头朝下的标记时，可使数字减小。





第八章 网络操作系统

8.1 计算机网络概述

8.2 网络体系结构

8.3 Internet与Intranet

8.4 客户/服务器模式

8.5 网络操作系统的功能

8.6 网络操作系统提供的服务





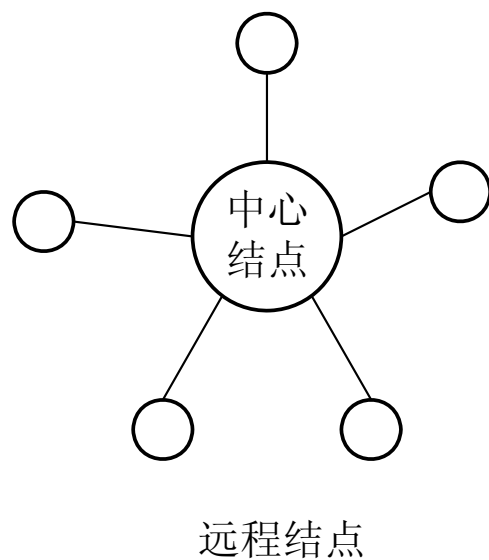
8.1 计算机网络概述

8.1.1 计算机网络的拓扑结构

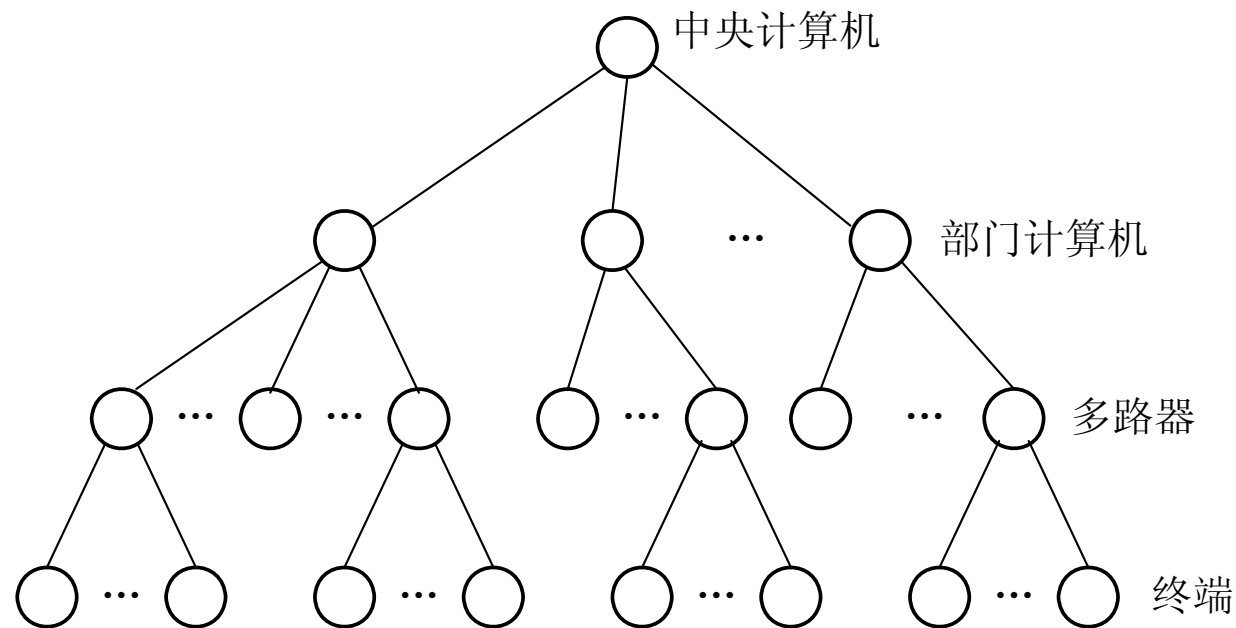
1. 星形和树形网络拓扑结构

1) 星形网络拓扑结构

这是指每一个中心结点通过点—点方式与若干个远程结点相连，使网络的拓扑结构呈现放射状的星形。如图8-1(a)所示，其所有各个远程结点之间因无连接而不能直接通信。



(a) 星形网络



(b) 树形网络

图8-1 星形和树形网络拓扑结构



2) 树形网络拓扑结构

鉴于单级星形网络的诸多不利条件，使之不适用于构建大型网络，于是产生了多级星形网络拓扑结构。如果将多级星形重新按层次方式排列，则形成了树形网络，如图 8-1(b) 所示。树形网络(拓扑结构)是对星形网络的一种改进。由于在中间层各结点上的处理机都具有控制和处理能力，因而使整个系统具有一定的分布控制和处理能力，即使中央处理机瘫痪，其它结点处理机仍可维持网络的局部运行。



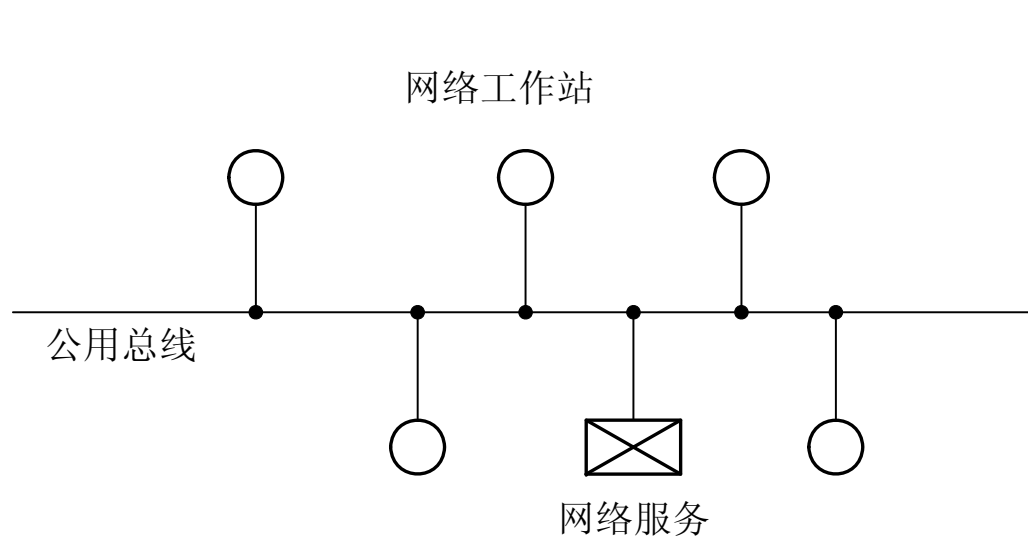
2. 公用总线形和环形网络拓扑结构

1) 公用总线形网络拓扑结构

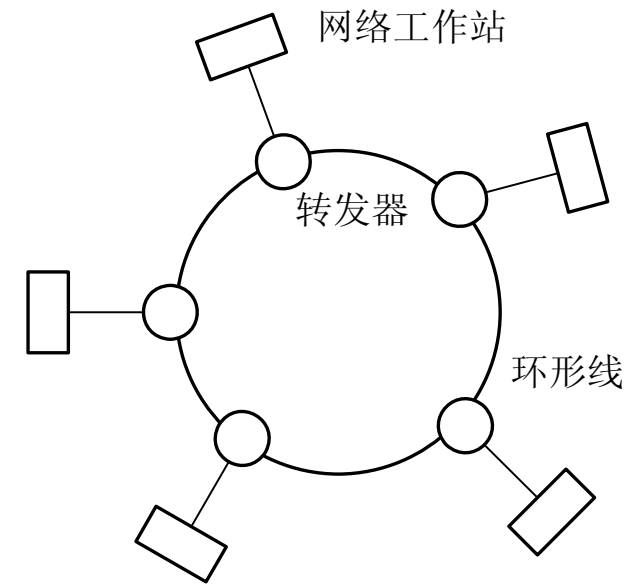
这是将若干个网络工作站(结点)分别通过一个连接器，连接到一条高速公用总线上所形成的网络拓扑结构，如图8-2(a)所示。通常在总线形网络上都连接有几个至几十个网络结点和一两个用于提供服务的网络服务器。总线形网络拓扑结构的最大特点，是由多个结点共享一条传输总线，使网络的物理结构简单、信道利用率高，而且是广播通信方式，亦即，由总线上任一结点所发出的信息，能被总线上的所有其它结点接收。但由于公用总线的长度受到一定限制，因而使总线网的地理覆盖范围一般局限于某个单位或部门。



第八章 网络操作系统



(a) 总线形网络



(b) 环形网络

图8-2 总线形和环形网络拓扑结构



2) 环形网络拓扑结构

这是通过点一点的连接方式，将所有的转发器连接成一个环形，其中的每个转发器可用于连接一个网络工作站，站上的信息通过转发器传送到环路上，信息在环路上只作单方向流动。环形网络拓扑结构的最大特点，仍然是由多个结点共享一条传输总线，使网络的物理结构简单，信道利用率高，而且是广播通信方式，见图8-2(b)所示。但基本的环形网络的可靠性差，当环上任一转发器发生故障时，都会导致整个网络瘫痪。公用总线形网络拓扑结构和环形网络拓扑结构主要用于局域网络。



3. 网状形网络拓扑结构

在广域网中最广泛采用的是网状形网络拓扑结构。它是通过点一点的连接方式，将分布在不同地点的、用于实现数据通信的分组交换设备PSE(Packet Switch Equipment)连接在一起，形成一个不规则的网状形网络。该网络专门用于实现数据通信，因而称为通信子网，如图8-3所示。

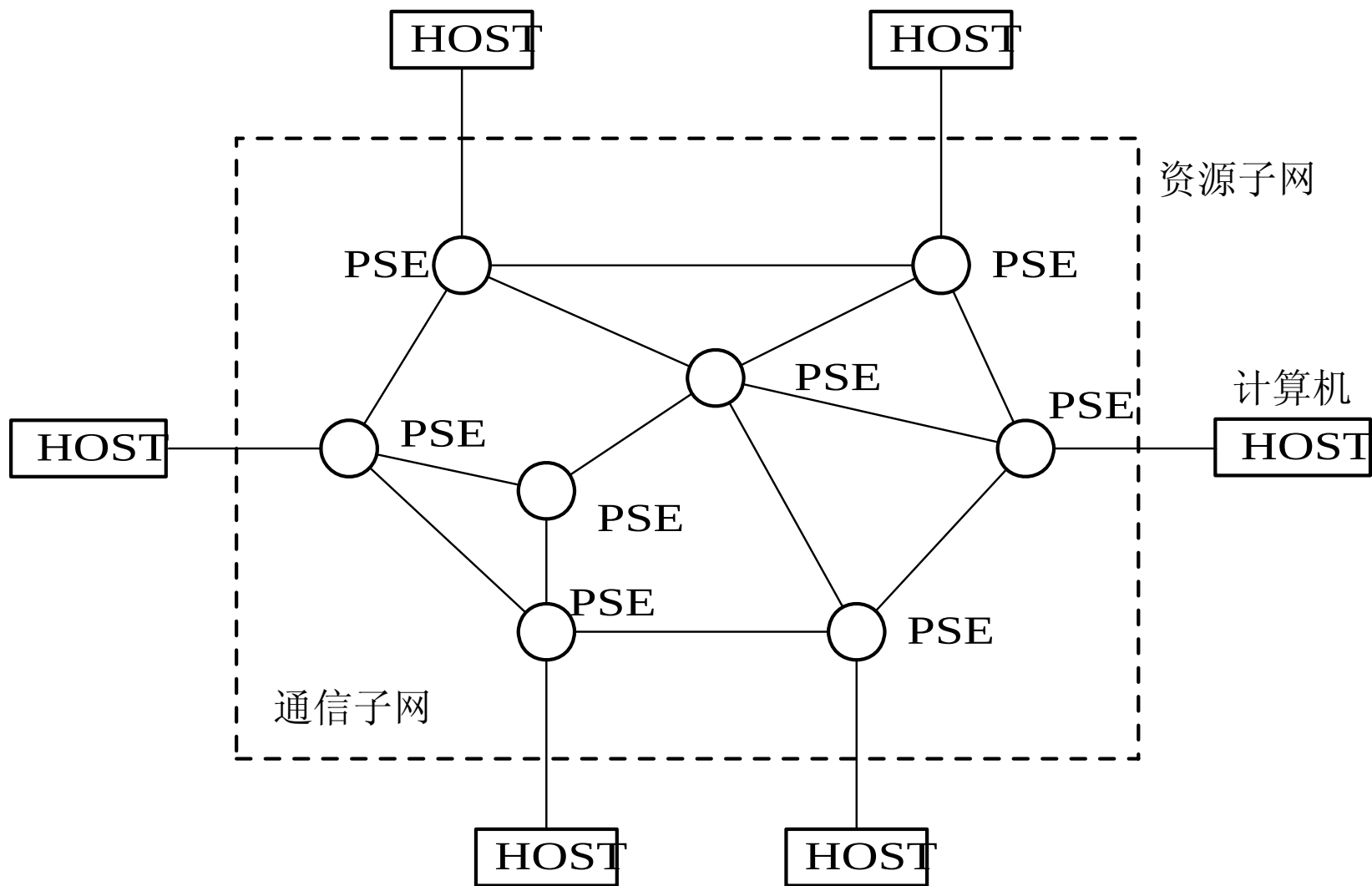


图8-3 网状形网络拓扑结构



凡要入网的主机 (HOST) 都应连接到该网络的一个PSE上，任何两主机之间的通信都必须通过该通信子网。所有这些主机都作为网络的信源和信宿，且其中有相当数量的主机都对信息具有强大的处理能力，驻留了大量的可供网络用户共享的文件和资料。由通信子网之外的所有主机构成了数据处理子网，也称为资源子网。可见，网状形网络在逻辑上可分为通信子网和资源子网两部分。这种网络形式的最大好处是便于将各种类型的计算机连接成异构型网络。



网络拓扑结构的主要特点是它具有分布性，亦即，在通信子网中的PSE都是分布在不同的地理位置上。类似地，用作数据处理的主机也是分布在不同的地理位置上。分布性可使信息得以就近处理，减少了网络中的信息流量；分布性提高了网络的可靠性，在任何两个重要的通信设备之间，都有两条以上的传输路由；分布性也改善了网络的可扩充性，网络的扩充几乎不受到限制。

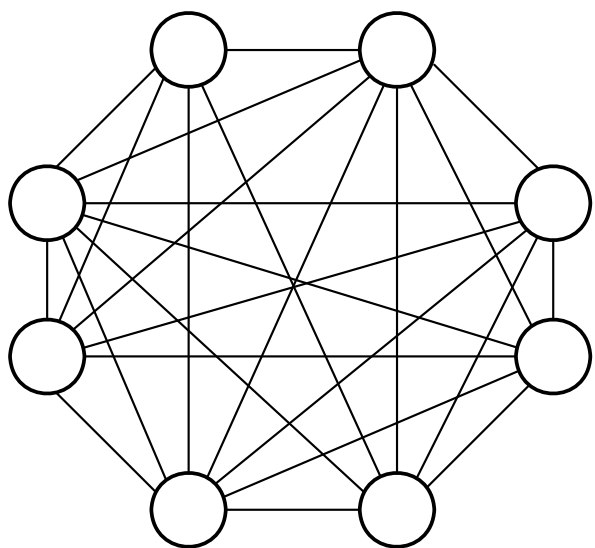


8.1.2 计算机广域网络

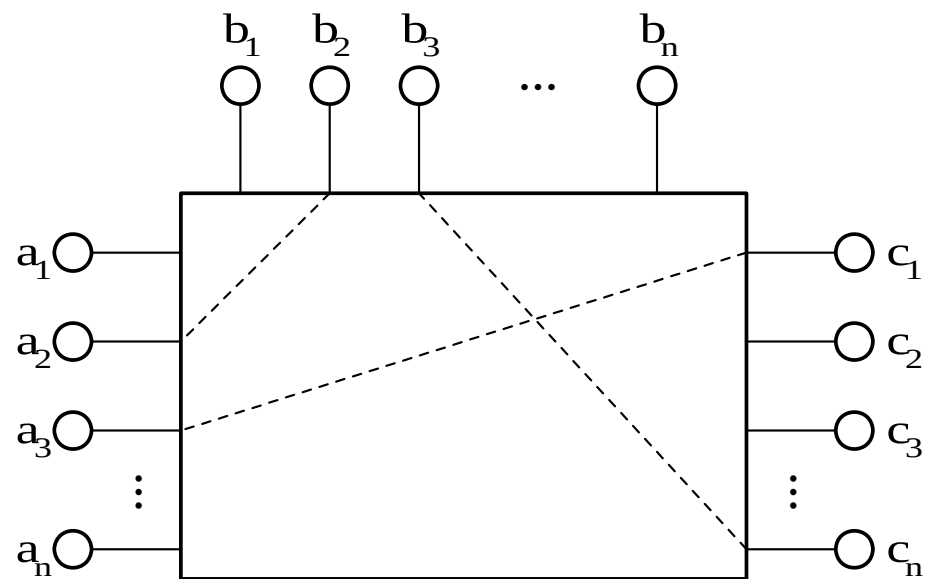
1. 公用交换电话网

1) 交换方式的引入

在点一点式的网络中，如果一个结点要与另一个结点进行通信，则必须在该两结点之间建立一条通信线路。随着结点数目的增多，其互连线的数目将呈平方关系增多，见图8-4(a)。显然，这种互连方法是不实际的。为了解决在众多结点之间的通信问题而引入了“交换技术”。所谓“交换”(Switching)，是指在两个或多个结点之间建立暂时通信线路(或链路)的操作。建立链路的操作是由交换中心完成的。两个结点在通信之前，须先建立链接，然后源结点把信息通过该链路发送给交换中心，再由交换中心把信息转发到目标结点，通信结束后便拆除该链接。图8-4(b)示出了具有交换中心时的连接方式。由图可见，其连接线数目与结点数目成比例。



(a) 全互连连接



(b) 具有交换中心的连接

图8-4 全互连和具有交换中心的连接



2) 线路交换方式

线路交互方式广泛用于电话系统中，它通过直接接通或断开某些线路来形成所要求的连接，使用户之间能直接通信；通信完后便拆除该连接，以便将线路让给其他用户对进行通信，如图8-5所示。线路交换方式主要适用于传输模拟信号。

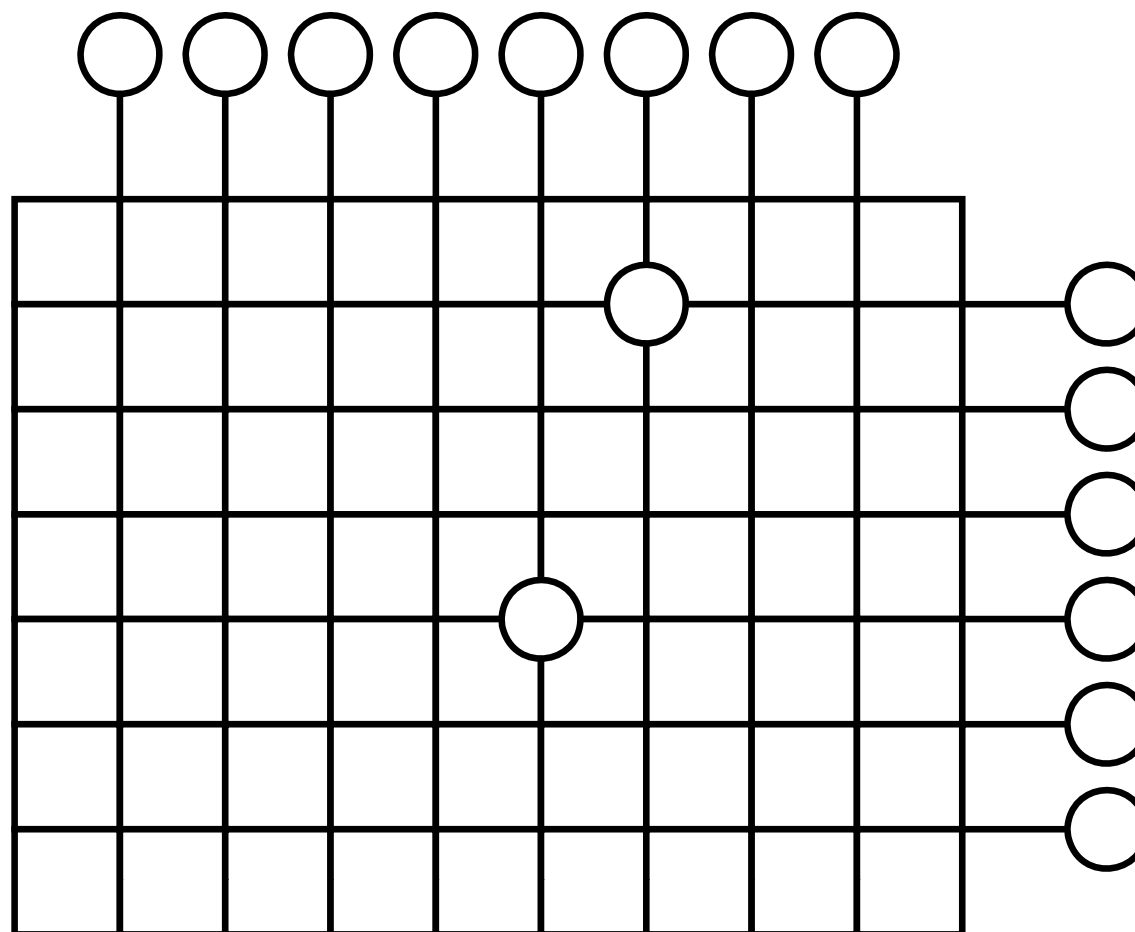


图8-5 线路交换方式示意图



3) 线路交换网

若将数字设备连接到网上时，必须通过调制解调器。在源主机处，由调制器将数字信号转换成模拟信号；而在目标主机处，则由解调器完成模拟信号到数字信号的反变换。利用电话网来传输数据的传输速率较低，一般为2400 b/s~64 kb/s。



2. 分组交换网

1) 报文交换方式

报文交换方式是最早用于电报系统中的数字式交换方式。它是基于“存储—转发”方式进行报文交换的，即数字式报文交换中心先将各用户发来的电报接收下来，存储在报文缓冲区中，经过适当的处理(如判别目标地址、报文优先级等)后，为该报文选择一条转发路由，并将它送至该路由的输出队列中排队，再依次将该队列中各报文转发出去。报文交换方式适用于传输数字信号，相应地，数字设备可直接入网。



2) 分组交换方式

分组交换方式是对报文交换方式的一种改进，它同样是基于“存储—转发”方式来传输信息的。为了提高传输效率而将不定长的报文分解成定长的(报文)分组(packet)，然后以分组为单位进行传输。这种方式的好处是：简化了对缓冲区的管理，加速了对信息的传输，减少了传输出错率以及重发信息量。



3) 分组交换网

分组交换网是以分组作为传输的基本单位。一个分组由分组头和正文两部分组成。正文是用户要传送的信息，而分组头则是用于控制该分组在网络中传输所必需的(控制)信息。当源主机要发送一份报文时，须首先将报文分解成若干个定长的信息正文段，并为每个正文段配上分组头，形成若干个分组，然后再逐个地发送分组。网络中的中继结点即分组交换设备PSE先将各分组接收下来，存储在定长的多个分组缓冲区中，再对所接收的信息进行差错检测，若无错，再为每个分组选择一条适当的传输路由，并将分组转发出去。应当指出，分组的格式及分组在网络中的传输都应遵循X.25协议，因而也常把分组交换网称为X.25网。



3. 帧中继网

长期以来，人们一直认为分组交换是实现数据通信的最好方式。但到20世纪80年代中、后期，一方面由于网上信息流量急剧增加，使分组交换网64 kb/s的传输速率已远远不能满足需求；另一方面，由于在WAN中已可利用光缆作为传输信道，大大降低了信息传输的误码率，因而可采用较简单的差错检测机制来提高信息的传输速率，于是，在1992年帧中继网便应运而生。目前已形成了几种帧中继网。最常用的是帧交换方式和信元交换方式两种。



1) 帧交换方式的帧中继网

帧交换方式是在传统分组交换方式的基础上发展起来的一种快速交换技术。帧交换方式中传输的基本单位是帧，其长度是可变的，它们同样都采用“存储—转发”方式，即帧交换器每收到一个新到的帧时，都是先将该帧送帧缓冲区中排队，然后按照该帧中的目标地址，将该帧转发给相应路径上的下一个帧交换器。这种帧中继的传输时延比起分组交换网来，要低一个数量级。



2) 信元交换方式的帧中继网

信元交换方式是对帧交换方式加以改进形成的、具有较好性能的帧中继的交换方式。在该方式中，网络中所传输和交换的基本单位是具有固定长度的“信元”。当源帧交换器收到用户设备发来的帧后，便将之分割为多个定长的信元，在整个帧中继网络中传输和交换时，都是以信元为基本单位，直至它们到达目标帧交换器后，才被重新组装成帧。与帧交换方式相比，信元交换方式可以获得更小的传输时延及更大的网络吞吐量。此外，由于信元长度固定且很小，各字节的含义及其位置都固定，因而完全可以用硬件方法来实现信元交换，大大提高了交换速度，从而使信元交换方式的帧中继具有更高的传输速率、更小的传输时延且时延大小固定，能够满足多种通信业务的需求，其中包括语音和视频业务。



4. 异步传输模式(ATM)

1989年, CCITT在研究和综合了多种快速交换机的基础上, 提出了一种新的传输模式, 即异步传输模式 ATM(Asynchronous Transfer Mode), 并为ATM规定了三方面的目标: ① 建造高速广域数字网络, 所传输的可以是任何形式的数字信号; ② 不仅在广域网中采用ATM, 还可在LAN和企业网中使用ATM, 这样便可实现WAN和LAN之间的无缝连接; ③ ATM网必须能提供各种业务服务, 并能满足用户对服务的合理要求。



1) ATM的传输原理

ATM是以信元(Cell)为基本传输单位的。信元由信头(5个字节)和信息段(48个字节)组成, ATM通过信头来识别信元。由于传输每个信元花费相同的时间, 因而可把信道的时间划分为一个时间片序列。在每个时间片中传输一个信元。当ATM有信元要发送时, 只要信道空闲, 便可将信元投入信道。由于信元的发送无固定的周期, 因而将这种传输方式称为异步传输方式。



2) ATM的交换

在分组交换网中，提供了面向连接和面向无连接的传输服务，而在ATM交换方式中，主要提供的是面向连接的方式，其交换方法与分组交换中的交换方法相似，但两者也有着明显的差别：① 按时间片交换。ATM交换采取按时间片交换的方式，即每个信元都必须放在一个时间片中，因此，信元的接收和转发都是按时间片进行的。② 定长交换。ATM中的交换单位是信元，是定长的，因而可大大减少对交换的处理。③ 硬件实现。由于在ATM中对信元的交换处理简单，故可用硬件实现，这无疑是ATM能获得极高传输速率的重要原因。



3) ATM的优点

ATM的主要优点有：① 高传输速率。ATM的传输速率有多个档次，即45 Mb/s，155 Mb/s，622 Mb/s，直到2.4 Gb/s等。②极大的灵活性。这主要表现为无业务限制，即无论哪种通信业务都可进入ATM网；无通路限制，因为ATM可提供数以万计的虚电路，供很多用户同时使用；传输时延确定，且因为ATM网中传输的时延是可以预见的，因而可支持实时业务。



8.1.3 计算机局域网

1. 基本型局域网

(1) 以太网(Ethernet)。以太网一直是国内外最流行的一种局域网，它采用的是公用总线型网络拓扑结构，传输速率为10 Mb/s。其所用的传输介质，在早期主要是同轴电缆，网络的最大覆盖范围为2.5 km，到20世纪90年代则主要是使用双绞线。这种以太网也称为10 BASE-T。为了控制公用总线信道的使用，在以太网中采用了带有冲突检测的载波侦听多重访问控制规程，亦即CSMA/CD规程，其最主要的特点是简单。



(2) 令牌环(Token-Ring)网。令牌环局域网也是当前较流行的一种局域网。它采用的是环形网络拓扑结构，传输速率为16 Mb/s。其传输介质可以是屏蔽双绞线，也可以是非屏蔽双绞线。网络的覆盖范围比以太网大。此外，还引入了优先机制来保证重要和紧急信息的优先传送。



2. 快速局域网

无论是以太网还是令牌环网，都是网上所有的站点共享一条公用信道，因而每个站点的平均带宽(平均传输速率)为网络传输速率/网络上的站点数。随着网上站点数目的增加，其平均带宽将随之下降。可通过两种途径来扩展LAN站点的平均带宽。其一是提高LAN的传输速率；另一途径是减少每个网段上的站点数目。快速LAN便是试图通过提高LAN的传输速率来增加每个站点的带宽的。于20世纪80年代后期，首先推出了FDDI光纤网；到20世纪90年代中期，研制出了快速以太网。



(1) FDDI光纤环网。FDDI是Fiber Distributed Data Interface的缩写。FDDI具有100 Mb/s的传输速率，光纤环网的最大覆盖范围可达100 km。为了提高环形网的可靠性，FDDI采用了两个光纤环，其中一个作为主环，另一个作为副环，某些重要设备可同时接到两个环上。由于FDDI具有很高的传输速率，故主要用作互联局域网的主干网。



(2) 快速以太网100 BASE-T。这是一种具有100 Mb/s传输速率的局域网。由于它采用了与10 BASE-T完全相同的介质访问控制规程，故把它称为快速以太网。它与10 BASE-T之间具有很好的兼容性，因而很容易将10 BASE-T升级为100 BASE-T以太网。又由于建造100 BASE-T网的成本大大低于FDDI的成本，因而它很快就成为具有100 Mb/s传输速率的主流局域网。



3. 交换式LAN

如果说，快速局域网的出现，是试图通过提高局域网传输速率的方法来扩展网段上每个站点的平均带宽，那么，交换局域网的引入，则是通过减少每个局域网段上的站点数目的方法，来增加站点的平均带宽。构建交换式局域网要比构建快速局域网更方便、经济。当我们将一个具有N个站点的以太网划分为M个网段，再利用交换器将各网段互连起来后，便形成了一个交换式局域网，这时，其中每个站点的平均带宽为 $10(\text{Mb/s}) \times M/N$ ，是原来的M倍。例如，当 $M = 10$ 时，能将平均带宽提高9倍，显然，这是一种扩展站点平均带宽的有效方法。所以，从20世纪90年代中期开始，便已广泛采用交换器将企业内部多个局域网互连起来，形成能覆盖整个企业的企业网络。



4. 千兆位以太网

在1997年中期，便已有许多公司推出了千兆位传输速率的以太网产品。1998年6月，千兆位以太网 (Gigabit Ethernet) 标准——IEEE 802.3z 终于获得通过，从而为千兆位以太网的迅速发展铺平了道路，人们把它称为第三代以太网。如今第一、二、三代以太网都处在并行使用的阶段，但分别用于企事业网的不同层次中，通常是将10BASE-T用于构建底层的工作组LAN，100BASE用于构建部门级LAN，而千兆位以太网则作为企业的主干网。



千兆位以太网仍采用CSMA/CD规程。在传输介质上，主要利用光纤系统，在利用短波激光发射器时，其传输距离可达300~550 m；如改为长波激光发射器，则可使传输距离提高到550 m，到上世纪末达13 000 m。IEEE 802.3中，还规定了利用高品质的铜质传输介质的以太网标准，但其传输距离仅限于25~100 m。



5. 10 Gb/s以太网

在本世纪初，便已有公司推出了10 Gb/s传输速率的试验性以太网产品，人们把它称为第四代以太网，10 Gb/s以太网仍采用CSMA/CD规程，其帧格式与前面三种以太网相同，帧的最小长度和最大长度也遵循802.3标准。在物理层标准中，规定传输介质只利用光纤系统，另外也规定了10 Gb/s以太网只能工作在全双工方式，因而不存在争用总线问题。可利用它来组建企业网络，此时一个10 Gb/s以太网交换集线器可支持10个千兆位以太网端口，也可将它们用于广域网中。



8.1.4 网络互连

1. 网桥

网桥是用于连接同构LAN的网络互连设备。一般来说，同构LAN是指从应用层到逻辑链路控制子层这几个层次中，相对应的层次采用相同的协议，而对于数据链路层中的MAC子层和物理层中的对应层次，则可遵循不同的协议。因此，可以把所有用于连接符合IEEE802标准网络的互连设备(如CSMA/CD总线网、令牌环网或令牌传送总线网)，称为网桥。



由于网桥是用于连接同构型LAN的，因此，网桥所实现的功能应属于MAC子层和物理层。从网桥的工作原理中不难得知网桥应具有以下功能：

(1) 帧的发送和接收。从其所连接的LAN端口中接收无差错帧，从帧中获得目标站地址的名字，可以得知目标站是否属于本网桥所连接的另一个LAN。若是，便接收该帧，并做进一步处理；否则，将该帧抛弃。



(2) 缓冲管理。在网桥中必须设置足够大的两类缓冲区，一类是接收缓冲区，用于暂存从端口收到的、要发往另一LAN的帧；另一类是发送缓冲区，用于暂存已经过协议转换等处理后要发往相邻LAN的帧。

(3) 协议转换。将源LAN中所采用的帧格式和物理层规程，转换为目标LAN所采用的帧格式和物理层规程。



2. 路由器

当要互连的LAN数目较多时，或者是实现异构型LAN的互连或实现LAN与WAN的互连时，网桥便难于胜任。此时应采用路由器。因为路由器是在网络层上实现的互连，它能识别不同的网络层协议，如IP、IPX协议等，因而具有更强的互连能力。路由器的功能涉及到物理层、数据链路层和网络层，其主要功能如下：

(1) “拆包和打包”功能。路由器在接收到任何一种数据包后，都将去掉数据链路层所加上的控制信息；根据网络层所加上的控制信息中的目标地址，为数据包选择一条最佳传输路径；然后在数据前加上新选择的路由信息，形成新的数据包。



(2) 路由选择功能。为了优化网络的传输性能，路由选择功能是要按照某种策略(如数据包在网中传输的时延最小，或传输路由最短，传输费用最低等)，为须转发的数据包选择一条最佳传输路由。

(3) 进行协议转换。路由器实现将一个网络所用的数据链路层和网络层协议转换为另一个网络中的相应协议。例如在将数据从以太网送入分组交换网时，须将数据包中的以太网协议转换成X.25协议。



(4) 分段和重新组装。由于在不同的网络中所采用的数据包的大小可能不同，如源站所用数据包大于目标站的数据包，使目标站无法接收。这时可将源站发出的数据分成若干段，再分别封装后发往目标站所在的网络。反之，路由器又可将属于同一报文的多个数据包按序号组装成一个大的数据包后传送，以提高传输效率。



3. 网关

网关用于互连异构型网络。所谓互连异构型网络，一般是指不同类型的网络，更确切地说，是指在两个网络中，至少是从网络层到物理层其协议都不相同的网络，甚至可以从应用层到物理层所有对应各层的协议均不相同的网络。因此，一般说来，在网关中至少要进行网络层、数据链路层及物理层的协议转换。显然，网关的实现是比较困难的。目前对异构型网络的互连，通常是在网络层或传输层上实现的。



第八章 网络操作系统

网关可用在下述几种场合的网络互连上，这些都属于异构型网络的互连。

(1) 异构型LAN互连：利用网关将几种完全不同的LAN互连起来；

(2) LAN与WAN互连：LAN与WAN比较，至少它们的低三层（网络层、数据链路层、物理层）协议不相同，因此，它们属于异构型网络，可用网关实现互连；

(3) WAN与WAN互连：主要用于不同的WAN之间互连；

(4) LAN与主机互连：这虽已不属于网络互连范畴，但在将主机连接到LAN上时，因为主机的操作系统与网络操作系统并不兼容，故仍须通过网关实现互连。由于主机的类型很多，因此，目前市场上这类网关很多。



8.2 网络体系结构

8.2.1 网络体系结构的基本概念

1. 何谓网络体系结构

在一个网络中需要许多网络协议，其中有的协议是解决较低层次的问题，如实现源主机和目标主机之间的通信；有的是解决较高层次的问题，如允许源主机系统去访问目标系统中的文件系统。因此，我们可以如同处理操作系统结构问题一样，即将操作系统分为多个层次，在每个层次中设置若干个功能的方法，来处理网络协议。于是我们将网络分为多个层次，上层功能的实现依赖于下层提供的功能；在每个层次设置一个或多个协议，分别用于实现不同的功能。



所谓网络体系结构，就是计算机网络的层次及其协议的集合。具体地说，网络体系结构是关于计算机网络应设置哪几层，每个层次又应提供哪些协议的精确定义。至于这些功能应如何实现，则不属于网络体系结构部分。从上面的介绍可以看到，网络体系结构是从层次结构及功能上来描述计算机网络结构，并不涉及每一层硬件和软件的组成，更不涉及这些硬件和软件本身的实现问题。对于同样的网络体系结构，可采用不同方法设计出完全不同的硬件和软件，为相应的层次提供完全相同的功能和接口。



2. 开放系统互连参考模型OSI/RM

1) 开放系统定义及其互连参考模型

(1) 开放系统(OSI)的内容。所谓开放系统,是指在与其它系统通信和相互合作方面,能遵循OSI标准的、能对信息进行处理或传送的自治整体。OSI主要涉及这样两方面的内容: ① 开放系统之间的信息交换,这是每一个单独的开放系统的内部行为; ② 开放系统之间相互合作去完成一项共同任务。相互合作包括相当广泛的内容,如进程间的通信,数据的表示,进程和资源的管理等。



(2) OSI/RM的组成。ISO7498描述了OSI参考模型，它建立了一种体制，用于协调现有的和将来的系统互连标准。

OSI/RM由四部分组成：① 开放系统；② 应用实体，即与OSI有关的应用进程的外貌；③ 连接，即为在两个或多个实体之间进行信息交换，所建立起来的一种连接(Association)；④ 物理介质，用于在开放系统之间进行连接。图8-6示出了OSI/RM的四个组成部分。



第八章 网络操作系统

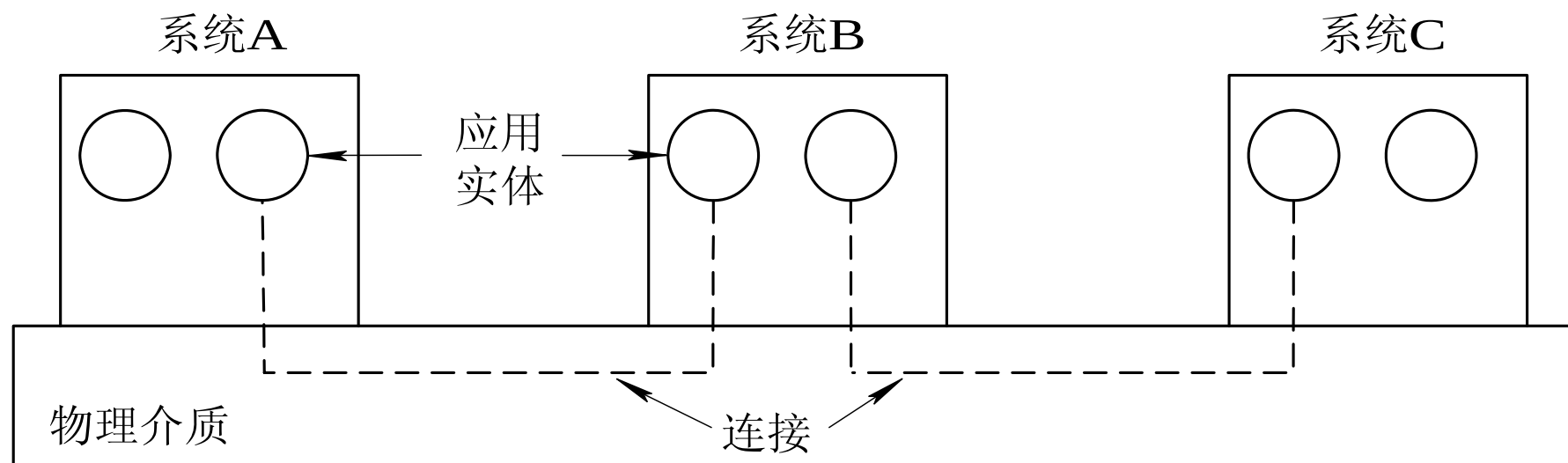


图8-6 OSI/RM的组成



2) 分层

在OSI/RM中所采用的基本结构技术是分层，每个系统可被看成是由有序的一组子系统所组成，如图8-7所示。由图可看出，一个系统被分成若干个层次，其中第N层是由若干个处于N层的子系统所组成。(N)子系统又包括了若干个(N)实体。在同一层中的实体为对等实体(Peer Entity)。除最高层外，分布在(N)层中的(N)实体相互合作，向(N+1)层的实体提供(N)服务。

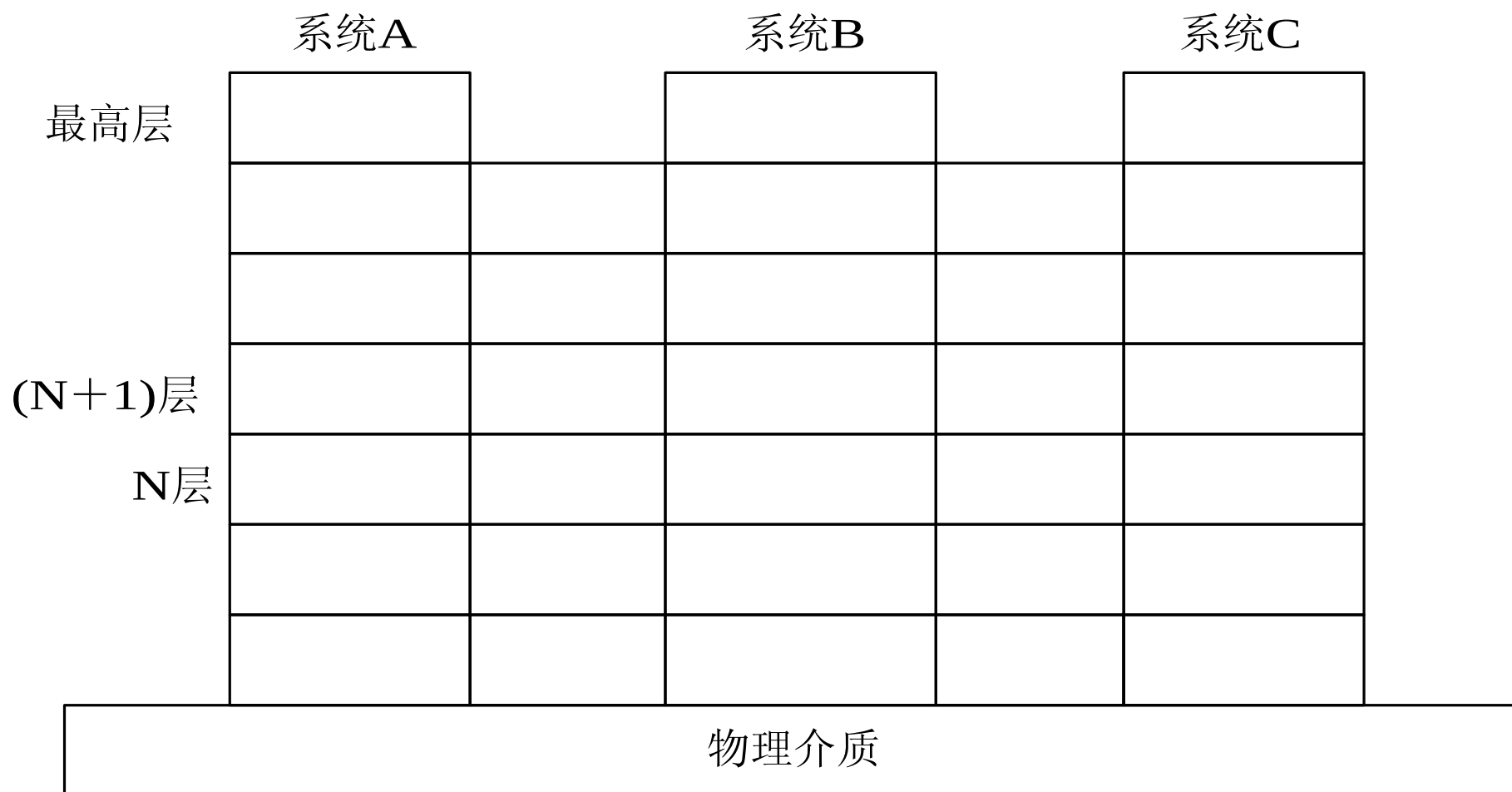


图8-7 OSI/RM的分层



3) 网络协议

(N)实体之间的合作，是受一个或几个(N)协议支配的。(N)协议精确地规定了(N)实体如何利用(N-1)服务协同工作去实现(N)功能。所谓(N)协议，是指一组局部于(N)层的规则和格式(语义和语法的)，它决定着(N)实体在执行(N)功能时的通信行为。其中，语义规定协议元素的类型，亦即，通信双方所要表达的内容；语法规则规定把若干个协议元素组合在一起来表达一个更完善的内容时，所须遵循的格式，即规定内容的表达形式；规则是指应答关系，即通信过程中双方的应答规则。



4) 数据单元

OSI把对等实体之间所传送的信息称为(N)协议数据单元(N)-PDU, 由两部分组成:

① (N)协议控制信息(N)-PCI, 用它来协调两个实体之间的连接操作; ② (N)服务数据单元(N)-SDU, 其中存放由(N+1)实体提供的数据。图8-8示出了相邻层数据单元间的关系。

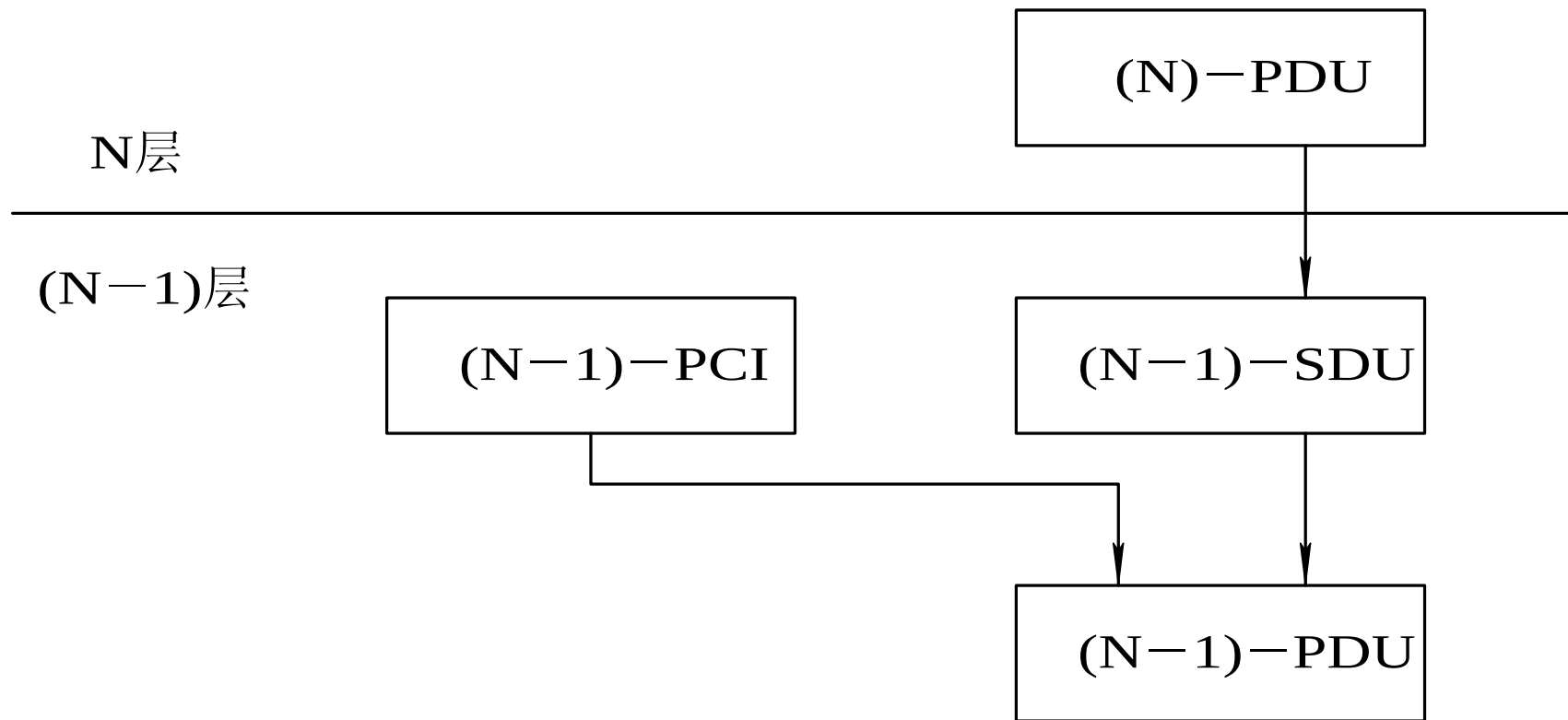


图8-8 相邻层数据单位的逻辑关系



3. OSI七层模型

OSI/RM共分七层，如图8-9所示。其中，最低层是物理层；次低层是数据链路层，再上是网络层、传输层、会话层、表示层；最高层是应用层。其中低三层即物理层、数据链路层和网络层用于实现通信子网中的信息传输，或者说，它们是面向通信的(一般称之为通信子网)；最高三层即会话层、表示层和应用层向应用进程提供资源子网功能的服务，因此它是面向应用的；中间层即传输层，它是在高三层和低三层之间起桥梁作用。





8.2.2 OSI/RM中的低三层

1. 物理层(Physical Layer)

物理层是OSI的最低层，它建立在通信介质(它不在OSI七层之内)的基础上实现系统和通信介质的接口功能，为数据链路实体之间透明的传输比特流提供服务。为实现数据链路实体之间比特流的透明传输，物理层应提供下述功能：

(1) 物理链接的建立与拆除。在两个数据链实体之间通信之前，先由物理层在它们之间建立物理链接，通信完后再拆除该物理链接。



(2) 物理服务数据单元传输。物理层既可采用同步传输方式，亦可采用异步传输方式传输物理服务数据单元，两种方式分别需要在系统中配置同步适配器或异步适配器来完成数据的发送和接收。

(3) 物理层管理。可以管理本层的某些事务，如功能的激活(何时发送或接收等)、差错控制等。



2. 数据链路层(Data-Link Layer)

数据链路层的主要用途是在相邻两系统的网络实体之间建立、维持和释放数据链路连接，以及正确无误地传输数据链路服务数据单元。数据链路层应具有下述功能：

(1) 数据链路连接的建立和释放：在两个相邻系统的网络实体之间，提供一条或多条数据链路连接，该连接是动态地建立和释放的。

(2) 数据链路协议数据单元的形成：将数据链路服务数据单元配上数据链路协议控制信息，形成数据链路协议数据单元。



(3) 定界和同步：可识别出在物理链接上传输的数据链路服务数据单元的开始和结束。

(4) 顺序和流量控制：控制数据单元的传输顺序和流量。

(5) 差错的检测和恢复：能检测出传输出错、格式出错和操作出错，并从出错状态中解脱出来；对于不能恢复的差错，则向网络实体报告。



3. 网络层(Network Layer)

1) 网络层的功能

网络层应具有的功能如下：

(1) 网络连接服务：利用数据链路连接构成源和目标两个传输实体间的网络连接；网络连接还可由若干个通信子网以串连形式构成。

(2) 路径选择：在两个网络地址之间选择一条适当的传输路径。



(3) 网络连接多路复用：提供多条网络连接以多路复用一条数据链路来提高数据链路连接的利用率。

(4) 分段与组段：当数据单元太长时可将它们分段；反之，亦可将几个较短的数据单元组成一个较大的数据单元一起传输。

(5) 有序传送和流量控制：利用有序传送方法来传送网络服务数据单元，并对网络连接上的信息流量进行控制。

(6) 差错的检测和恢复：检测网络连接上所传输的数据单元是否出错，并使之从错误状态中解脱出来。



2) 网络层提供的数据传输服务

(1) 数据报服务。在数据报传输方式中，发方网络层从传输层接收报文，再为它配上完整的目标地址后，作为一个独立的信息单位传送出去。数据报每经过一个中继结点时，都要根据当时当地的情况，并按一定算法选择一条最佳传输路径转发出去。事实上，数据报类似于通常的电报。在采用数据报服务时，收、发双方无需建立连接。



(2) 虚电路服务。通信前先由源主机发送呼叫报文分组，其中包含源和目标主机的全网地址。目标主机若同意通信，便由其网络层在双方之间建立一条虚电路(虚电路是通过分时复用一条物理链路后所形成的一条逻辑电路)。在以后的通信中，便只需填上虚电路的逻辑信道号即可；通信结束时，将该虚电路拆除。在概念上，虚电路服务类似于通常的电话。



8.2.3 OSI/RM中的高四层

1. 传输层(Transport Layer)

传输层在低三层和高三层间起桥梁作用。该层消除了OSI高层所要求的服务与各类网络层所提供的服务之间的差异，具体表现在以下三方面。

(1) 传输出错率和建立连接的失败率：在OSI标准中根据出错率和失败率，可把网络提供的服务分为A，B，C三类。A类最好，C类最差。传输层应针对不同的网络服务再增加相应的服务，使之满足高层的要求。



(2) 数据传输速率、吞吐量和传输时延：如果通信子网所提供的这几项指标不能满足上层要求，传输层应增加相应的措施来改善这些指标。

(3) 分段和组段功能：网络层所传送数据单元的大小是一定的，但会话层数据单元的大小是任意的，因而传输层应具有分段和组段功能，通过对会话层数据单元进行分段、组段，来构成适合于在网络连接上传送的数据单元。



2. 会话层(Session Layer)

会话层的作用是对基本的传输连接服务进行“增值”，以提供一个能满足多方面要求的会话连接服务。会话层的“增值”是基于下述几种应用要求的。

(1) 半双工通信。在某些系统中，系统与远程终端之间采用双方轮流发送的方式，这只要求网络提供半双工通信方式。为此，在会话层设置了数据令牌，仅仅持有数据令牌的会话用户才具有传输数据的权力。



(2) 更有效的差错纠正机制。会话实体必须保留已发送的全部数据，以备传输出错时重发。但若所传输的数据很长，其所付出的空间和时间开销是可观的。为了减少时空开销，会话层向会话用户提供了同步点，用以把传输的数据隔离为若干段，然后，逐个分段地发送和确认。对已经收到确认的段，便释放其占用的空间(缓冲区)；仅对未收到确认的段，才认为出错并予以重发。

(3) 允许暂停发送消息。当发送方已发完已有消息后，允许暂停发送一段时间。若暂停时间较短，可不断开已建立的连接。恢复发送时，从原中断点开始。



3. 表示层 (Presentation Layer)

表示层的主要用途是对不同系统的表示方法进行转换，消除网内各应用实体之间的语言差异，以实现不同系统之间的数据交换。事实上，在异构网络中，各系统对传输数据的表示方法因机器而异。为使各开放系统中的应用实体之间能进行通信，要求各系统按相同的规则对数据进行编码。这种共同的规则被称为传送语法(Transfer Syntax)。不按这种规则进行编码的数据，将无法被对方理解。这就要求各系统都具有将本地语法转换为传送语法和将传送语法转换为本地语法的能力。所以从语法角度上看，表示层最基本的作用是协商一种传送语法，以实现传送语法和本地语法之间的变换。表示层还可提供一些附加功能，如数据加密、数据压缩等。



4. 应用层

应用层是OSI/RM的最高层，它为应用进程访问OSI环境提供了手段，并直接为应用进程服务，其它各层也都通过应用层向应用进程提供服务。应用层所提供的服务可分为两类：

(1) 公共应用服务元素(CASE)。公共应用服务元素是用户元素和特定应用服务元素中公共使用的那部分元素。它提供了应用层中最基本的服务，其中包含了为多个应用实体协作所提供的服务，以及为分布式处理的同步和分布式数据库的更新同步提供保证。



(2) 特定应用服务元素(SAEA)。为特定需要提供服务。目前ISO只对已广泛使用的某特定应用服务元素进行了标准化。如文件传送、存取和管理(FTAM)，虚拟终端(VT)及作业传送和操纵(JTM)等。



8.2.4 TCP/IP 网络体系结构

1. TCP/IP模型

由网络互连协议IP和传输控制协议TCP一起，构成了著名的TCP/IP协议，它一直是Internet网络的核心协议，并已成为事实上的网络互连协议的标准，几乎所有的WAN和LAN都支持该协议。TCP/IP是一个协议族，其中包含了多种协议，由这些协议构成了TCP/IP的网络体系结构。虽然没有官方公布的TCP/IP的分层模型，但可根据已制定的许多协议而将TCP/IP模型分为四层(物理层不在模型中)。图8-10示出了TCP/IP模型与OSI模型的比较。



图8-10 TCP/IP模型与OSI/RM的比较



1) 网络访问层

在源主机系统，网络访问层接收由网络互连层送下来的IP数据报，并对它做些处理后，将它发送给选定的网络，后者又将它传送给目标主机。目标主机系统的网络访问层接收由目标主机物理层向上传送的IP数据报，经处理后，再向上送给网络互连层。由此可知，该层主要关注的是两个端系统之间的数据通信，以及两个端系统借以通信的网络类型。所使用的网络可能是电路交换网、分组交换网、ATM网或者以太网等。可见，网络访问层是与网络相关的，因此，人们将与网络访问相关的功能分离出来，单独形成一个独立的网络访问层是必要的，这样便可把对网络访问的实现细节隐藏起来，使在网络访问层以上的各层通信协议，与所使用的网络无关。



2) 网络互连层

网络互连层是TCP/IP模型中最重要的层次，其中的IP协议主要用于异构型网络之间的相互连接和路由选择。IP所提供的是面向无连接的、不可靠的传输服务，它可使由源主机发送的IP数据报，穿越由各种WAN和不同的LAN互连形成的互连网络，到达目标主机。从20世纪70年代中期到90年代中期，一直使用IP协议的第4版本IP V4.0；后来在IP V4的基础上又提出了新的IP协议，此即IP V6，它继承了IP V4的一切优点，又对IP V4的不足之处做了修改和补充。



3) 传输层

传输层中最主要的协议是传输控制协议TCP(Transmission Control Protocol)，它所提供的是面向连接的、可靠的端—端通信机制。由于TCP和IP两个协议是同时使用在一个系统中的，故人们把它们称为TCP/IP协议。TCP协议是建立在网络层的基础上的，在制定TCP时，已考虑到它所依赖的通信子网可能是不可靠的，因此，在TCP协议中，采取了增强可靠性的措施，以确保传输层能正确无误地运行。



4) 应用层

应用层处于TCP/IP模型的最高层。它提供了许多用于支持各种应用程序的网络服务，相应地，在应用层就有许多应用层协议，如用于支持文件传输的文件传输协议FTP、提供电子邮件服务的简单邮件传送协议SMTP、远程登录协议TELNET，以及用于实现网络管理的简单网络管理协议SNMP等。



2. 互联网协议IP V4和IP V6

1) IP V4 协议

IP V4是早期在Internet上使用的网络互连协议(亦称“互联网协议”),可利用它来实现网络互连。为了能使IP V4数据报穿越由各种不同的网络互连所形成的互联网,IP V4协议主要应解决三个问题,即寻址、数据报的分段和重新组装、路由选择。



(1) 寻址。为了能在由多个不同网络构成的互连环境下，唯一地标识网络中每一个可寻址的实体，应为这些实体赋予全局性标识符。当前在网络中经常采用两种名字结构：① 分级地址结构。在分级地址结构中的标识符，通常由网络号、主机号和信口号组成。在国际性网络中还需缀上国家号。② 平面地址结构。这是直接用若干个字节的一个整数来标识一个对象，例如，用2个字节便可标识 $2^{16} = 64 \text{ K}$ 个对象，若用6个字节便可标识 $2^{48} = 256 \text{ T}$ 个对象。



(2) 分段和重新组装。在不同的网络中，所规定的帧长度并不相同。例如在X.25网中优先选用的最大帧长度为128个字节，而在以太网中则为1518个字节。这样，当信息从以太网送入X.25网时，就应先进行分段，并为每个分段重新配置一个帧头，形成一系列新的帧；在由WAN把信息传送到目标LAN时，又应对它们进行重新组装。

(3) 路由选择。如果用户希望IP数据报能沿着指定的路由传送，则应采用源路由选择方式。这时，应在IP数据报中指定由源主机到达目标主机的显式路由。这里又可分为两种：① 完全路由选择：在IP数据报中记录下它所应经历的全部路由；② 部分路由选择：在IP数据报中记录下它所经历的部分路由。



2) IP V6协议

IP V6协议继承了IP V4协议的一切优点，而针对其不足之处做了以下几方面的修改，使之能更好地满足当今Internet网络的需要。

(1) 扩大了地址空间。IP V4协议的规定地址长度为4个字节，它只能提供 $2^{32} \approx 4 \times 10^9$ 个地址；而在IP V6协议中的地址长度已扩充到16个字节，其可提供的地址空间为 $2^{128} \approx 3.4 \times 10^{38}$ 个地址。

(2) 增设了安全机制。在IP V6协议中引入了认证技术，以保证被确认的用户仅能去做已核准他做的事。



(3) 提高了路由的转发效率。IP V6协议规定仅由源端系统进行数据的分段，而途经的所有路由都不得对数据进行分段。

(4) 增强了协议的可扩充性。IP V6包含了一个可扩展的数据报头，增加了选择设定的灵活性。



3. 传输层协议 TCP和UDP

1) 传输控制协议TCP

针对IP协议是提供面向无连接的、不可靠的数据报服务，TCP则提供了面向连接的、可靠的端—端通信机制。所谓面向连接，是指在端系统要传送数据前，应先进行端—端之间的连接；在数据传送完后，应拆除连接。而所谓可靠是指，即使网络层(通信子网)出现了差错，TCP协议仍能正确地控制连接的建立、数据的传输和连接的释放。



为了确保数据传输的可靠性，在TCP中采用了确认和重发措施，即接收方每收到一个正确的数据段时，都应根据数据段中的发送序号，给发送方回送一个用于确认的ACK段。如果所接收的数据有错，则要求对方重发。此外，在进行正常的交换时，也要有流量控制，即控制发方发送数据段的速度不应超过接收方接收数据的能力。而对于紧急的数据段，则不应受流量控制。



2) 用户数据报协议UDP

应当指出，虽然TCP协议提供了可靠的数据传输服务，但它却降低了传输效率，这对于早期通信网络不太可靠，而要传输的数据服务又非常重要，TCP协议是十分必要的；但如果所传输的数据并非很重要，仍采用TCP协议则会显得有些浪费，此时可考虑利用UDP协议来传输数据。



该协议是一种无连接的、不可靠的协议。它不要求网络中的端系统在数据传送之前先建立端—端之间的连接；同样，在数据传送结束后，也不要拆除连接。在数据传送过程中，无需对传送的数据进行差错检测，也不必对丢失的数据进行重发等。换言之，它是以一种比较简单的方式来传送数据，因而有效地提高了传输速率，比较适合于对传送可靠性要求不太严格，或能自己进行错误检测的应用程序，如简单网络管理协议等。



8.2.5 LAN网络体系结构

1. 局域网参考模型LAN/RM

图8-11示出了LAN的层次结构与OSI/RM层次结构间的对应关系。在IEEE 802标准中，只定义了物理层和数据链路层两层，但802委员会在制定LAN/RM时，做出了一个关键性的决定，即将数据链路层分为两个子层：逻辑链路控制子层LLC和介质(媒体)访问控制子层MAC。正是由于这一关键性的划分，使LAN获得了LLC规程与具体的LAN所采用的介质访问方法与网络结构形式无关这一理想特性。



OSI/RM

7	应用层
6	表示层
5	会话层
4	传输层
3	网络层
2	数据链路层
1	物理层

LAN/RM

IEEE 802 参考模型

逻辑链路控制子层
介质访问控制子层
物理层

图8- 11 LAN/RM与OSI/RM的对应关系



2. 逻辑链路控制LLC子层

LLC(Logical Link Control)子层是数据链路层的顶部子层，其主要功能是在任何一个源LLC实体和目标实体之间进行信息传输。IEEE 802.2标准对LLC子层所用的协议数据单元进行了定义，描述了任何两数据链路实体间的数据传输规程。在LLC子层中提供了两种类型的链路操作，其中类型1操作提供的是无连接服务，类型2操作提供的是面向连接的服务。相应地，在IEEE 802.2标准中还规定了两种规程：



(1) LLC 1型规程。该规程只支持类型1操作。类型1操作作为无连接操作，即在双方通信之前，无需建立逻辑链路，其所传输的PDU既不被确认，也没有流量控制和出错恢复机制。它类似于以前介绍的“数据报”服务。

(2) LLC 2 型规程。该规程既支持类型1操作，又支持类型2操作。该类操作规定，在两个LLC间交换PDU之前，必须先建立逻辑链路连接。在逻辑链路上的正常通信周期内，包括两种PDU，即由源LLC到目标LLC的带有信息的PDU和反向的确认PDU。在此类操作中，应按PDU的序号进行传输。在逻辑链路上所传输的信息PDU应受到流量控制。标准中还对LLC子层向高层提供的服务以及LLC子层请求MAC子层的服务做了描述。



3. 介质访问控制MAC子层

IEEE 802 委员会在讨论MAC (Medium Access Control) 方式时，出现了两种矛盾，一种是推荐CSMA/CD为MAC标准，另一种是推荐令牌传送。这两种方式各有其优点，都获得了很多委员的支持，因此，委员会决定把它们同时作为MAC推荐标准。由于令牌传送既可用于环形网，也可用于总线网，于是形成了三种局域网标准。下面列出了部分IEEE 802标准。



第八章 网络操作系统

IEEE 802.1 (A)	局域和城域网络标准——概述和结构。
IEEE 802.1 (B)	寻址、网络互连和网络管理。
IEEE 802.2	逻辑链路控制规范。
IEEE 802.3	CSMA/CD访问方法和物理层规范。
IEEE 802.3u	快速以太网标准。
IEEE 802.3z	千兆位以太网标准。
IEEE 802.4	令牌传送总线访问方法和物理层规范。
IEEE 802.5	令牌传送环访问方法和物理层规范。
IEEE 802.6	城域网(MAN)标准。



IEEE 802. 7	宽带局域网标准。
IEEE 802. 8	光纤局域网标准。
IEEE 802. 9	语音和数字综合局域网标准。
IEEE 802. 10	可互操作的局域网安全标准。
IEEE 802. 11	无线局域网标准。
IEEE 802. 12	100 VG AnyLAN(百兆位请求优先级标准)。

IEEE 802 标准与OSI/RM之间的关系如图8-12所示。



802.10 可互操作的局域网安全标准

802.1 体系结构与网络互连

802.2 逻辑链路控制子层

802.3
CSMA/CD

802.4
令牌总线

802.5
令牌环

802.6
城域网

802.9
语音与
数据综合
局域网

802.11
无线
局域网

图8-12 IEEE 802标准



8.3 Internet与Intranet

8.3.1 Internet简介

1. Internet的发展过程

Internet的发展过程可分为三个阶段。第一阶段是从1969年至1988年，在此阶段是以美国的ARPA网络作为主干网，网上的主机数目由最初的4台发展到近10万台，主干网的传输速率由56 kb/s提高到T1(1.544 Mb/s)，并于1982年决定利用TCP/IP来取代以前在ARPA中所用的协议；



第二阶段是从1988年至1992年，在此阶段是以美国的NSFNET为主干网，网上的主机数目由近10万台发展到超过100万台，主干网的传输速率也由T1升级到T3(44.7 Mb/s)。第三阶段是从1993年起至目前，网上的主机数在1997年已超过2000万台，并建立了一个具有更高传输速率的WAN—ANSNET，1996年时ANSNET又升级为Internet的主干网，传输速率增至155 Mb/s，到本世纪初，不少干线的速率已达到2.4 Gb/s。



Internet的应用已从以科学教育为主迅速扩展到社会的各个领域，使Internet进入了商业化阶段。在20世纪90年代后期，美国一些大学申请建立Internet 2，为其成员组织服务，初始运行速率提高到10Gb/s；Internet 2的应用领域可为多媒体虚拟图书馆、远程医疗、远程教学、视频会议、视频点播VOD、天气预报等领域。



2. Internet的特征

Internet是当今世界上空前规模的特大型互连网络，并作为21世纪信息高速公路的雏型，它具有任何其它网络都没有的一系列特征，其中主要是广域性、广泛性、高速性和综合性等特征。

(1) 广域性。在现今世界上有许多类型的WAN，它们都具有很宽的地理覆盖面，几乎所有主要的WAN都已接入Internet，因而Internet所覆盖的地理范围就是这些WAN所覆盖范围的总和。它已遍布于世界上的170多个国家和地区，可见Internet所具有的广域性特征，是任何其它WAN所无法比拟的。



(2) 广泛性。上述的广域性是指Internet所覆盖的地理范围十分辽阔，而广泛性则是指Internet所涉及到的领域、行业、部门及人员十分广泛，如有政府机关、科研机构、高等院校和中小学校、金融和商业系统、医院等几乎所有的行业。

(3) 高速性。在Internet的第一阶段尚不具有高速性；第二阶段时已具有一定的高速性；而到Internet的第三阶段时，已经建立起以ATM交换器为中心的主干网，其传输速率已提高到155 Mb/s和2.4 Gb/s，而Internet 2 则突破了10 Gb/s大关，可以认为Internet已成为信息高速公路。



(4) 综合性。从Internet最近的发展趋势可以看出，它最终必将走向“三网合一”之路。所谓“三网合一”，是指公用交换电话网(PSTN)、公用数据网(X.25)及有线电视网(CATV)三者融合为综合性通信网，向用户提供文字、数据、音频和视频等种类繁多的多媒体业务。



3. IP地址和域名

1) IP地址

IP地址是在Internet中主机(包括工作站、服务器、路由器等)的地址标识。在Internet中只为运行的主机分配一个IP地址,该地址在整个Internet中是惟一的。IP地址共有32位二进制数,分为4个字节,用每个字节来表示一个十进制整数,因而32位二进制数可以表示为4个十进制数,在各十进制数之间均用小数点隔开,例如202.96.15.65。



Internet委员会规定：每个主机的IP地址都是由网络标识和主机标识两部分组成，可分为A、B、C三类。A类IP地址用第1个字节作为网络标识，用后3个字节作为主机标识；B类IP地址用前两个字节作为网络地址，后两个字节作为主机标识；C类地址用前3个字节作为网络地址标识，后一个字节作为主机地址标识。图8-13示出了A、B、C三类IP地址的格式。



第八章 网络操作系统

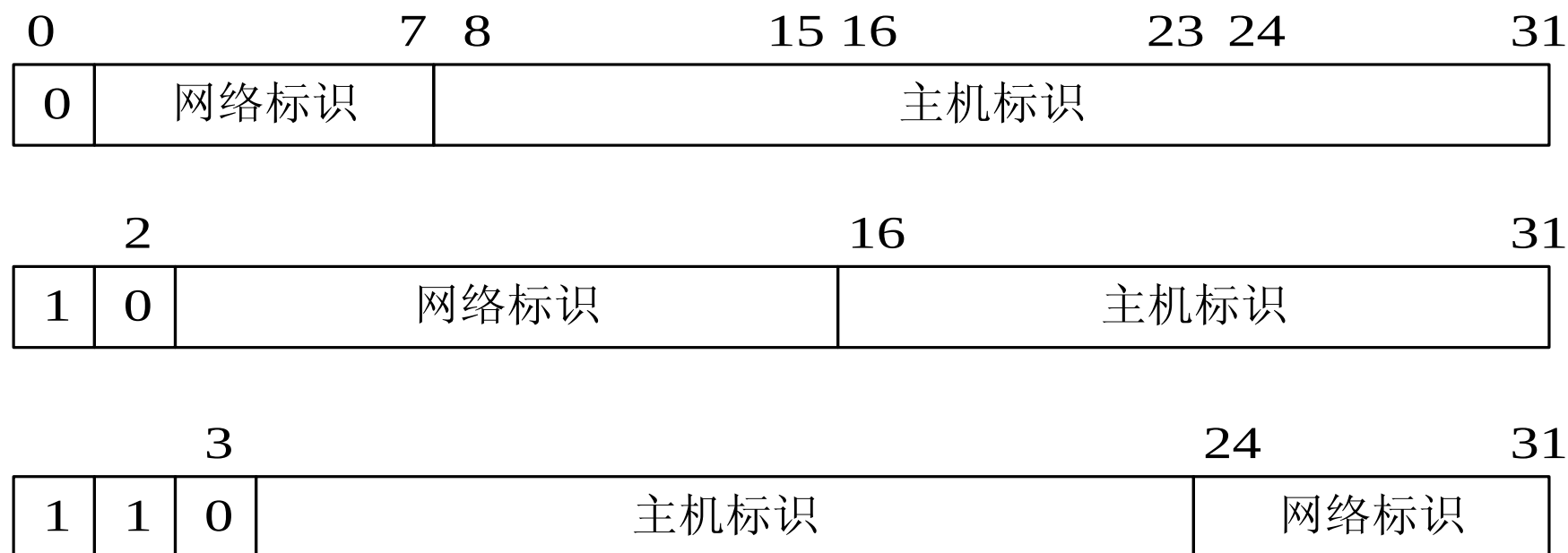


图8-13 A、B、C三类IP地址格式



2) 域名

如果说，IP地址是面向网络的主机标识符，适合于网络(计算机)识别和处理，那么，域名则是面向用户的主机标识符，它便于人们的理解和记忆。IP地址和域名之间一一对应，意味着每个主机都有IP地址和域名两个标识符。每个域名通常由几个部分(段)组成，我们把域名中的每个段称为一个子域，各子域之间用小数点分隔开。放在域名最后的子域称为最高级子域，或称为一级域，在它前面的子域称为二级域。一级域通常是com(商业机构)，或edu(教育机构)，或gov(政府部门)，或net(网络机构)，或mil(军事机构)等。如果主机、网络不是在美国，则还须加上国家名的简称，通常是用两个字符来表示，如中国用CN、日本用JP。



8.3.2 Internet提供的传统信息服务

1. 电子邮件(E-mail)服务

用户可以借助于Internet所提供的E-mail服务与世界上几乎所有的国家和地区的网络用户,以电子邮件方式进行通信。E-mail已成为Internet上应用最多的信息服务之一。值得说明的是,早期的E-mail只能是文本,而在现代的E-mail中已可包含多种不同类型的文件,如文本、图像、音频和视频信息等。换言之,现代的E-mail已可支持多媒体的信息传输。为了实现E-mail功能,用户须向Internet服务供应商申请一个账号,以便得到一个确定的E-mail信箱地址。该地址实际上代表某主机上的某块磁盘空间的首地址,该磁盘空间专供用户存放往来的邮件。



Internet中电子邮件系统的实现，是基于客户/服务器模式的，而在客户与服务器之间或在服务器与服务器之间的信息传输，都是采用简单邮件传输协议SMTP(Simple Mail Transfer Protocol)。由于SMTP特别简单，这使电子邮件系统的实现变得非常容易，但它所传送的邮件必须是ASCII码文本。

由于SMTP协议中的电子邮件仅限于使用ASCII码文本，为了能在单个电子邮件中允许含有多种成分，每种成分的数据类型或子类型可以各不相同，如可以是文本，也可以是图像或声音，人们又制定了一种多用途的Internet电子邮件扩充协议，此即MIME (Multipurpose Internet Mail Extensions)协议，这是SMTP的一种伙伴标准，是SMTP的扩充，MIME协议支持多媒体电子邮件的传输。



2. 文件传输服务

为了实现在异构网络环境下的文件传输，在Internet中建立了统一的文件传输协议FTP(File Transfer Protocol)。而FTP服务是指在FTP协议的支持下，用户可把文件从一台主机拷贝到另一台主机上。利用FTP在两台主机间拷贝文件，已成为当今世界上最大的软件流通渠道。在Internet中提供了以下两种形式的FTP服务：

(1) 内部用户FTP。所谓内部用户FTP，是指只允许那些在文件服务器上拥有账户的用户使用FTP服务。每当用户要使用FTP服务时，必须先输入正确的账号和口令，然后才能访问文件服务器上自己拥有读权限的文本，也可以向自己具有写权限的目录中上载数据。



(2) 匿名FTP。在Internet上实现资源共享的重要手段，是Internet提供的匿名FTP(Anonymous FTP)服务。该服务允许非注册用户拷贝(下载)文件。用户在与FTP服务器建立连接时，可用“Anonymous”作为用户名，这时，FTP服务器可能会提示用户在原应输入口令之处输入自己的电子邮件(信箱)地址，此后，用户便可有限地访问FTP服务器上的免费文件。在Internet上有成千上万的结点，可通过匿名服务向用户提供免费软件。



3. 远程登录服务TELNET

TELNET实质上是一个基于网络的终端仿真程序(Terminal Emulator)，即把用户使用的终端或主机通过Internet变为远程主机的仿真终端，其目的是使终端或主机能访问远程系统中的资源，而且能像远程系统中的用户一样访问资源。当然，为能在远程系统主机上登录，须首先成为该系统的合法用户，并有相应的账号和口令。一旦登录成功，用户便可实时地使用远程主机对外开放的全部资源。如今，世界上已有许多大学的图书馆都通过TELNET对外提供数据库联机检索服务；一些政府部门、研究机构等，也将它们的数据库对外开放，供用户通过TELNET进行查询。



TELNET程序可分为两部分：一部分是用于发出登录请求的TELNET客户程序；另一部分是用于应答登录的TELNET服务器程序。当用户要进行远程登录时，应在TELNET命令中给出远程主机的域名或IP地址，然后根据对方系统的询问，正确键入自己的用户名和口令，有时还要回答自己所用仿真终端的类型；但在TELNET中也有一些数据库对外提供开放式远程登录服务，即在用户查询这些数据库时，不需提供用户的账号和口令。



4. 电子公告板系统BBS

BBS(Bulletin Board System)是Internet上较早提供的一种服务。Internet用户可以利用BBS进行交流。当前BBS在国外已发展得相当成熟，规模也已很大，甚至出现了全球性的BBS。近几年，国内也出现了一批BBS站点，其规模和数量也在与日俱增。



BBS是在某些主机或服务器上开辟的一块公共存储空间，供所有用户使用，有时也把这块公共空间称为公共电子白板。每个用户都可在此公共电子白板上“张贴”供他人阅读的文件、消息。其他用户均可从白板上选择自己感兴趣的文章或新闻来阅读，然后，再利用该白板来发表自己对某篇文章或新闻的评论，大家也可在白板上围绕某个专题展开讨论。参加讨论的用户可以提问、发表意见，也可以只是“旁听”。利用BBS进行交流和开展讨论的方式，第一次打破了空间和时间的限制；在相互交流时，也无须考虑参与者的年龄、学历、社会地位、财富及其外貌、健康状况等。换言之，参加讨论的用户相互间处于完全平等的地位，这一点是现在任何其它交流、讨论方式所做不到的。



8.3.3 Web服务

1. WWW的基本概念

WWW(Word Wide Web)称为环球网或Web。它是当前最为流行的信息服务类型。利用该服务可使人们在网上漫游、进行信息浏览和发布信息。所谓漫游,是指对Internet上分布于世界各地的Web服务器进行访问。Web是一种信息检索工具,但它与一般的信息检索工具之间有很大差异,主要表现在:一般的检索工具每次只能从一台主机(服务器)上查找所需文件,且文件中只含有一种类型的数据,比如ASCII码数据;而Web检索则可以一次从多台主机中找到所需数据,且允许在这多台主机中使用不同类型的数据,并将这些数据形成一份文件,如用ASCII码数据或二进制数据,也可以是声音或图像。我们把这样形成的文件称为超文本文件。



2. 超文本标识语言HTML(Hyper Text Markup Language)

HTML是用于创建超文本文件的编程语言。可用该语言向普通文件中添加一些特殊的标识符，使在所生成的文件中，含有其它多种类型的文件，如声音，图像等，我们把这种文件称为超文本文件。但实际上，超文本文件本身并不含有上述的多媒体数据，而是只含有指向这些数据的指针。用这些指针可以把用户端的客户程序从一台计算机转移到另一台计算机。在HTML中把这些指针称为链接，这种转移是自动实现的，因而对用户是透明的。超文本文件可以由Web浏览器或其它Web工具解释执行，这些工具必须支持HTML。



HTML具有以下特点：

- (1) 通用性。HTML是Internet上的共同语言和通用的信息描述方式。
- (2) 简易性。HTML文件制作简单，HTML版本的升级采取超集方式，即在原有版本的基础上增加新内容，形成新版本。
- (3) 可扩充性。HTML采取子类元素的方式，使HTML具有很好的可扩充性。
- (4) 平台无关性。HTML可用于各种类型的平台上。



3. 超文本传输协议HTTP

HTTP (Hyper Text Transfer Protocol) 是一个通用的、面向对象的客户 (Web浏览器)/服务器 (Web服务器) 协议。该协议所包含的内容涉及到一般语法和标识符的约定, 定义了协议中所用的字符集、编码方式、媒体类型等参数。该协议属于TCP/IP协议族中的应用层通信协议, 是建立在TCP协议基础上的, 依赖于TCP协议来确保传输的正确性。可利用该协议来传输简单的文本、超文本、声音信号和图像, 以及任何在Internet上可以访问的信息。



HTML中还引入了一个特殊的参数URL(Uniform Resource Locator), URL是一个简单的格式化定位器(字), 用于描述浏览器在检索资源时所用的协议、资源所在的主机名、资源的路径名和文件名。URL的一般格式为:

协议 + “: //” + 主机域名或IP地址 + “: ” 或 “/” + 路径名或文件名

URL的最后部分可以没有, 例如,

`http://WWW.netscape.com/index.html`

协议名 主机名 文件路径与文件名



4. WWW的基本特征

(1) 对信息资源访问的分布性。用户可利用Web浏览器进行基于超文本方式的访问，通过在超文本文件中的许多指针，去链接各地Web服务器中的信息资源，从而可使用户去访问遍布于世界各地Web服务器中的信息。

(2) 信息形式的多样性。在Web服务器上所存储和传输的信息，是真正的多媒体信息，这不仅使Web的信息服务更具有吸引力，而且也为电话、电视和计算机的三者合一奠定了坚实的基础。



(3) 用户界面的统一性。Web服务器向用户提供了统一的用户界面和友好的信息访问接口，对世界各地的信息都可进行访问，不论这些信息来自何处，其访问方法都是相同的，即用户只须提出访问请求。

(4) Web服务应用的广泛性。由于Web服务是建立在Internet上的，因而不仅信息资源丰富、信息传播范围广，而且对信息的获取也极为方便，致使Web信息服务的应用极为广泛，已被用于信息查询、广告宣传、电子商务、电子银行、电子出版等诸多方面。



8.4 客户/服务器模式

8.4.1 两层结构客户/服务器模式的局限性

在前面所介绍的客户/服务器模式中，只有客户机和服务器两级，客户机直接与服务器进行交互。我们把这种客户/服务器模式称为两层结构的客户/服务器模式。早期以简单的局域网作为信息处理平台时，广泛采用两层客户/服务器模式。此时由于信息系统规模较小，所配置的又是单个LAN或少数几个同构型网络，因而采用两层的客户/服务器模式比较合适。但随着网络规模的扩大，在多个异构型LAN互连时，由于不同LAN可能采用了不同的网络工作站和协议及不同的数据库等，此时若仍用两层客户/服务器模式，就会暴露出许多不适应之处。



两层客户/服务器模式的主要问题在于：它不能适应应用不断增多的情况。在两层客户/服务器模式下，为实现客户与服务器之间的交互，应该在客户机与服务器中都装上特定的传输协议软件(如SPX/IPX)，以实现客户机与服务器之间信息的互通性；另外，还需要在客户机与服务器上安装特定的高层(表示层和应用层)网络软件(如NCP)，以实现客户机与服务器之间信息的互用性，即客户机能访问服务器上的文件系统，以实现信息共享。如果需要将客户机连接到另一台数据库服务器(主机)上，而该服务器又使用了其他的传输协议(如TCP/IP)和数据库系统软件(如SYBASE)，则此时须在客户机上增配能用于PC机上的TCP/IP软件，方能与服务器进行通信。再配置SYBASE Client软件，才能访问服务器上的数据库。如果又要将这台客户机连接到IBM主机上时，还须在客户机中再增配相应的软件。可见，随着应用的扩大，在客户机上所配置的软件就愈来愈多，这就使客户机变得愈来愈“胖”，形成所谓的“胖客户机”。



服务器通常都与许多客户机相连。如果对服务器中的某种软件做了修改或升级，就可能导致客户机上的软件必须重新装配，或者还须随之升级，否则将无法获得服务器软件修改或升级带来的好处，甚至有时还要求硬件也随之升级。可见，在采用两层客户/服务器模式时，为能适应应用不断变化和发展的需要，就必须付出高昂代价。因此，这种客户/服务器模式通常只适用于较小规模的信息系统和网络中。



8.4.2 三层结构的客户/服务器模式

1. 三层结构的客户/服务器模式的引入

稍加分析便可得知，形成上述两层客户/服务器模式局限性的原因在于：客户机是直接与服务器交互的，服务器的变化也就会直接影响到客户机。由此可以得出解决这个问题的基本方法是：设法使客户机与提供数据等服务的服务器无关。为此，可在客户机与服务器之间，增设一中间实体，用该实体把客户机与服务器隔开。通常把这个中间实体称为应用服务器或中间层服务器，把提供数据服务的服务器称为数据服务器或后端服务器，这样就形成了如图 8-14所示的三层结构的客户/服务器模式。

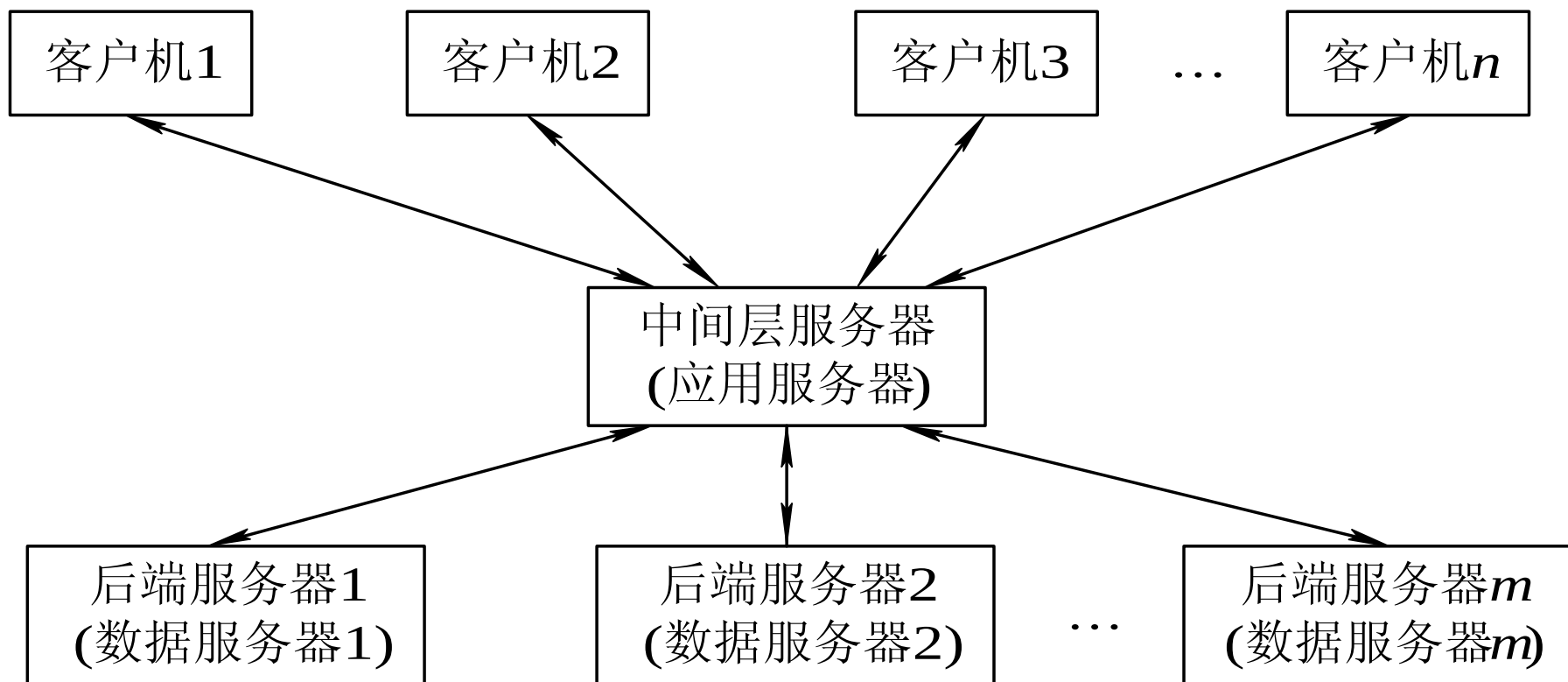


图8-14 三层C/S模式



在三层C/S模式中，由于增加了应用服务器，且该应用服务器又可连接到多个不同类型的数据(库)服务器上，也具有访问它们的软件和接口，因而客户机便可通过应用服务器去访问多个数据(库)服务器，但这时客户机并无必要为访问这些数据(库)服务器而增加任何软件和接口；或者说，可以把两层客户/服务器模式客户机中的大部分应用软件和接口移到应用服务器上，从而简化客户机，使之由“胖客户机”变为“瘦客户机”。

采用三层结构的客户/服务器模式时，客户机与应用服务器之间的交互遵循客户/服务器模式，应用服务器与数据服务器之间的交互也遵循客户/服务器模式。此时应用服务器作为客户机，由它向数据服务器发出请求消息。



2. 应用服务器的组成和功能

应用服务器大体由三部分组成，如图8-15所示。其左边部分是它与客户机交互的接口，用于接收从客户机发来的请求消息和向客户机发送响应消息。右边部分是它与数据(库)服务器交互的接口，中间部分是事务逻辑。事物逻辑的主要功能有两个：功能一是将用户的请求包转换为对数据(库)服务器访问的请求包，功能二是将数据(库)服务器返回的响应包转换为对客户机的响应包。



对于功能一，又可分为如下三步：

(1) 对请求消息进行分析。根据客户发来请求消息的内容和格式，从中得知该消息应发往的目标服务器、请求服务的内容、所采用的网络协议等。

(2) 进行网络协议的转换。通常由客户机到应用服务器这一段网络中所采用的各层网络协议(从第1层到最高层)，可能不同于应用服务器到数据服务器这一段网络中所采用的各层网络协议，因此在应用服务器中可能需要执行多层协议转换，即根据应用服务器与数据服务器之间网络所采用的网络协议，对请求消息所用的网络协议进行转换。



(3) 组装发往数据服务器的请求消息。利用目标服务器地址、请求服务内容等信息，按协议要求将请求消息组装成发往数据服务器的请求消息，并将它提交给与数据服务器交互的接口。

对于功能二，有着与功能一类似的三步，这里就不再赘述。

由上所述不难得知，应用服务器在客户机与数据服务器之间起着“桥梁”作用，或称为“网关”作用。应用服务器的组成如图8-15所示。

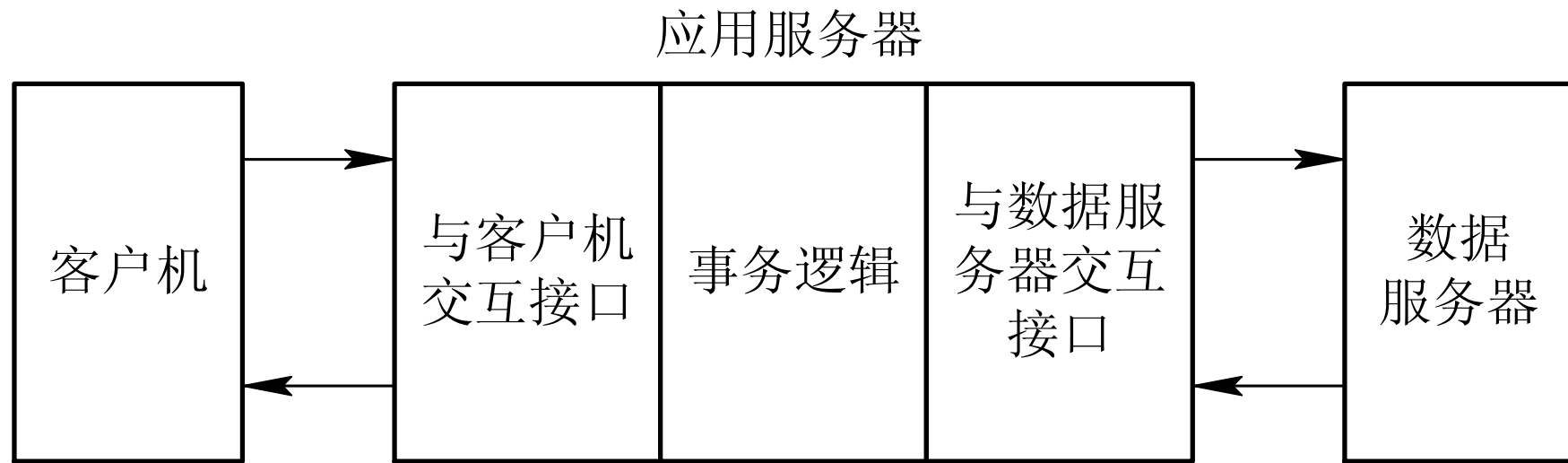


图8-15 应用服务器的组成



8.4.3 两层客户/服务器与三层客户/服务器的比较

(1) 增加了系统的灵活性和可扩充性。在两层客户/服务器模式中，对应用的处理是由服务器和客户机协同完成的。每当增加一个新的应用时，都须在客户机上配置相应的客户软件，但由于客户机本身无论在性能上，还是其内存容量，都非常有限。因而严重地影响到信息系统的灵活性和可扩充性。而对于三层客户/服务器模式，每当在系统中增加了新的应用和数据(库)服务器时，都只须在应用服务器中增加新的相应软件，而应用服务器通常是采用高性能、大容量的机器，所以它具有更大的灵活性和可扩充性。此外，在一个大型系统中，允许配置多个应用服务器和数据库服务器。这些服务器可以是本地的，也可以是远程的，这使系统更为合理、灵活。



(2) 简化了客户机，降低了整个系统的费用。把大部分应用软件从客户机移至应用服务器的结果，不单是简化了客户机，而且，由于在客户机中的应用软件只能是客户机独占，但在将它移至应用服务器后，只需稍加修改，便可供多个客户机共享，因而大大地节约了内存空间，降低了整个系统的费用。



(3) 使客户机的安装、配置和维护更为方便。在两层客户/服务器模式时，如果应用的规模较大，就需在客户机中配置较多的应用软件，这会增加客户机软件安装和配置的复杂性，而且还会因数据库服务器中软件的变化而需要重新配置客户软件，或予以更新；而在采用三层客户/服务器模式时，由于减少了客户机中的软件，而且数据库服务器中的软件若有所修改、更新或升级，都只须对应用服务器中的软件加以修改、更新或升级，这并不影响为数众多的客户机，因而使客户机的安装、配置更为方便，并可显著降低维护费用。



但是，三层客户/服务器模式也存在以下缺点：

(1) 开发难度大，开发周期长。基于两层模式的客户/服务器，其应用开发要比三层模式时的客户/服务器容易，开发周期也短，这是因为：① 基于两层模式的客户/服务器，其面向对象技术及与之配套的功能强大的应用开发工具较多，利用这些工具去开发应用时，不仅降低了开发难度，而且可大大缩短开发周期；然而基于三层模式的客户/服务器的开发工具目前还较少。② 在两层客户/服务器模式时，客户机中的应用软件是独占的，而用三层C/S模式时，在应用服务器中的应用软件通常是共享的，为了提高共享效率，要求这些软件编码是可重入的，因而在开发应用软件时，须采用可重入码，这无疑增加了开发难度。



(2) 访问效率低。对于两层客户/服务器模式，客户机直接访问数据库服务器，这种访问方式通常可获得较高的访问效率(在中、小型信息系统中)，而且还具有很强的实时性；而对于三层客户/服务器模式，由于客户机在每次访问数据库服务器时，都必须通过应用服务器，这对于大型信息系统以及在Internet/Intranet环境下是必要的，但对于中、小型信息系统，则是低效的。



8.4.4 浏览器/服务器(Browser/Server)模式

上面介绍的客户/服务器模式可分为两层客户/服务器模式和三层客户/服务器模式。传统的小型LAN采用两层客户/服务器模式；大型企业网中应当采用三层客户/服务器模式；而在基于Internet的Internet内部网络中，应采用哪种模式呢？

众所周知，在Internet中，客户机上的用户可以“进入”Internet进行“漫游”，去访问成百上千种类型的服务器。如果在这种情况下仍采用两层的客户/服务器模式，就意味着在客户机上必须配置访问这些服务器的接口和大量的客户访问软件，而且，Internet还以极迅速的速度发展，因此，这样做显然是不现实的。



解决这一问题的最佳方法，是在Internet中再增加一个Web服务器，它相当于前面所介绍的应用服务器，此时的客户机不是直接去访问Internet中的(数据库)服务器，而是访问Web服务器，再由Web服务器代理客户机去访问某个(些)(数据库)服务器。由于此时的客户机(已配上浏览器软件)可以浏览在Internet中几乎所有的允许访问的服务器，因此，这时便把客户机称为Web浏览器，这样一来便形成了Web浏览器、Web服务器和数据库服务器三层的客户/服务器模式。通常把这种三层结构的模式称为浏览器/服务器模式。



在Internet上的浏览器要访问Internet中的数据库服务器，是通过Web浏览器与Web服务器之间的交互和Web服务器与数据库服务器之间的交互实现的。浏览器与服务器之间的交互与传统的C/S之间的交互方式相似，都属于请求/响应方式，所不同的是浏览器所检索的对象通常是超文本文件，因此在浏览器与Web服务器之间所采用的是HTTP传输协议。



8.5 网络操作系统的功能

8.5.1 数据通信功能

1) 连接的建立与拆除

为实现应用进程之间的可靠通信，需要在两个系统的物理层之间建立物理连接，为信息传输提供一条传输路径；再在相邻结点的数据链路层间，建立数据链路连接，以实现相邻结点间无差错的信息传输；在网络层中，又需在源传输实体和目标实体之间，建立一条网络连接，……直至在两系统的表示层中，为两个应用实体建立了表示连接，即除应用层外的所有各层都要为数据通信建立相应的连接。当通信结束后，又需将各层中的相应连接拆除。



2) 报文的分解与组装

如果所传输的信息较长，在源主机中将信息由传输层送至网络层之前，应将报文的正文数据分解成若干个适合于在网络层传输的分组，然后逐个按序将它们送至网络层；再由网络层把分组发送给下一个结点。当这些分组到达目标主机时，由传输层按分组序号将这些分组重新组装成报文，再通过会话层、表示层送至应用层中的目标进程。



3) 传输控制

在通信双方已建立起连接之后，为使用户数据在网络中能正常传输，必须为用户数据(称为报文体)配上报头，其中含有用于控制数据传输的信息，如目标地址、源主机地址、报文序号等。然后利用该连接传送用户的数据，网络根据报头中的信息控制报文的传输。对传输中数据单元的控制，可采用下述两种方式之一：



(1) 发送-等待方式。当源(N)实体发出一数据单元后，要等待目标系统的(N)实体发回应答，然后源(N)实体再根据应答情况决定下一步的操作。若对方发回的是确认，源(N)实体可继续发送下一个数据单元；若对方发回的是否认应答，则原(N)实体应重发该数据单元。

(2) 连续发送方式。源(N)实体在发出一个数据单元后，无需等待对方的确认，便可继续发送下一个数据单元，直到全部数据发送完毕，或按某种规则暂停发送数据为止。这样就消除了每包发送后的等待时间。收方可在收到几个数据单元后，汇总起来一起发回一个确认应答。显然，这种连续发送方式提高了传输效率。



4) 流量控制

在分组交换网中，信息的传输采用“存储—转发”方式。在每个结点中，都准备有一定数量的缓冲区，用来接收远地发来的信息。倘若某条传输路由上的信息流量太大，致使其路由上某结点的缓冲区很快用完，再无空缓冲来接收新到达的信息，此时必然会造成信息的丢失。为了避免发生这种情况，在网络中必须设置流量控制功能。可见，流量控制的主要任务是：控制从源实体所发出的(N)-PDU的速度，不应超过目标(N)实体的接收和处理能力，以及和(N-1)实体连接的传输能力。



5) 差错的检测与纠正

数据在网络中传输必须是无差错的，否则，后果是难以预料的。但数据在网络中传输时，难免会出现差错，通常用误码率来度量信道的出错程度，一般信道误码率为 $10^{-9} \sim 10^{-4}$ 。为了减少数据在传输过程中的错误，网络中必须有差错控制设施，以完成下述两个具体任务：

- (1) 检测差错：即发现数据在传输过程中所出现的错误；
- (2) 纠正错误：对已发现的错误加以纠正。



8.5.2 网络资源共享功能

1. 硬盘共享

在20世纪80年代初期，LAN上的网络工作站(客户机)有不少微机都未配置硬盘(把这种工作站称为无盘工作站)，或者配置了个容量不大的硬盘。那时的用户要想建立自己的文件系统，可以在服务器上申请一块硬盘空间，然后在上面建立属于自己的文件系统。因而在网络操作系统功能中最早提供的是“以虚拟软盘方式实现硬盘共享”的功能。



1) 以虚拟软盘方式实现硬盘共享

这种方法是将服务器上的共享硬盘空间划分成若干个分区，把每个分区称作盘卷或卷，其容量根据用户要求而定，可从几兆到几百兆字节。当用户需要扩充外存或共享数据时，可首先利用建卷命令，在硬盘上建立一个卷，然后再用安装命令或链接命令把指定的已分配给自己的盘卷安装到自己工作站上的某个尚未使用的逻辑驱动器上，此即在盘和逻辑驱动器间建立了链接，这样便形成了用户工作上的一个虚拟盘。每个虚拟软盘都有自己的文件分配表FAT和根目录等。



以虚拟软盘方式实现硬盘共享所存在的问题是，在用虚拟软盘方式的服务器软件时，并未提供对虚拟软盘盘卷上的文件进行管理的机制，这时系统要求用户对其上的文件进行管理。每当用户对虚拟软盘上的文件进行读、写操作时，需先由工作站上的软件将用户的读、写请求分解为对特定磁盘扇区的请求，再将该扇区的请求传送给服务器，因此虚拟软盘方式对用户是不方便的。



2) 以文件服务方式实现硬盘共享

以文件服务方式提供的硬盘共享，是通过系统向用户(程序)提供对服务器上文件系统中的目录和文件进行有效的、可控的存取手段来实现的。由于用户对所有文件的访问，都是由服务器而非工作站来管理的，故在每个工作站上无需再配置文件分配表FAT，也不再需要由工作站进行从文件名到文件物理地址的转换，并能保证文件中数据的安全性。因此，这种文件服务方式更受用户欢迎。20世纪80年代中期以后推出的各NOS，大多采用文件服务方式实现硬盘共享。此外也有不少NOS在提供以文件服务方式共享硬盘的同时，也兼而使用虚拟软盘方式，以便用户来扩充自己的存储空间。



2. 网络打印

1) 假脱机打印系统的组成

在LAN中以假脱机方式实现共享打印的原理，和操作系统中假脱机打印的实现原理相似。为了实现打印机的共享，应在服务器硬盘上建立一个输出井，在井中设置两种类型的盘块，即空闲盘块和装有用户输出数据的盘块；同时，在系统中设置两个进程，即假脱机管理进程和假脱机打印进程；以及在内存中开辟一个打印缓冲区。为了实现用户的打印要求，须由假脱机管理进程为每个要求打印的用户建立一个假脱机文件，并将这些文件放入一个假脱机文件队列。图8-16是以假脱机方式共享打印的示意图。



第八章 网络操作系统

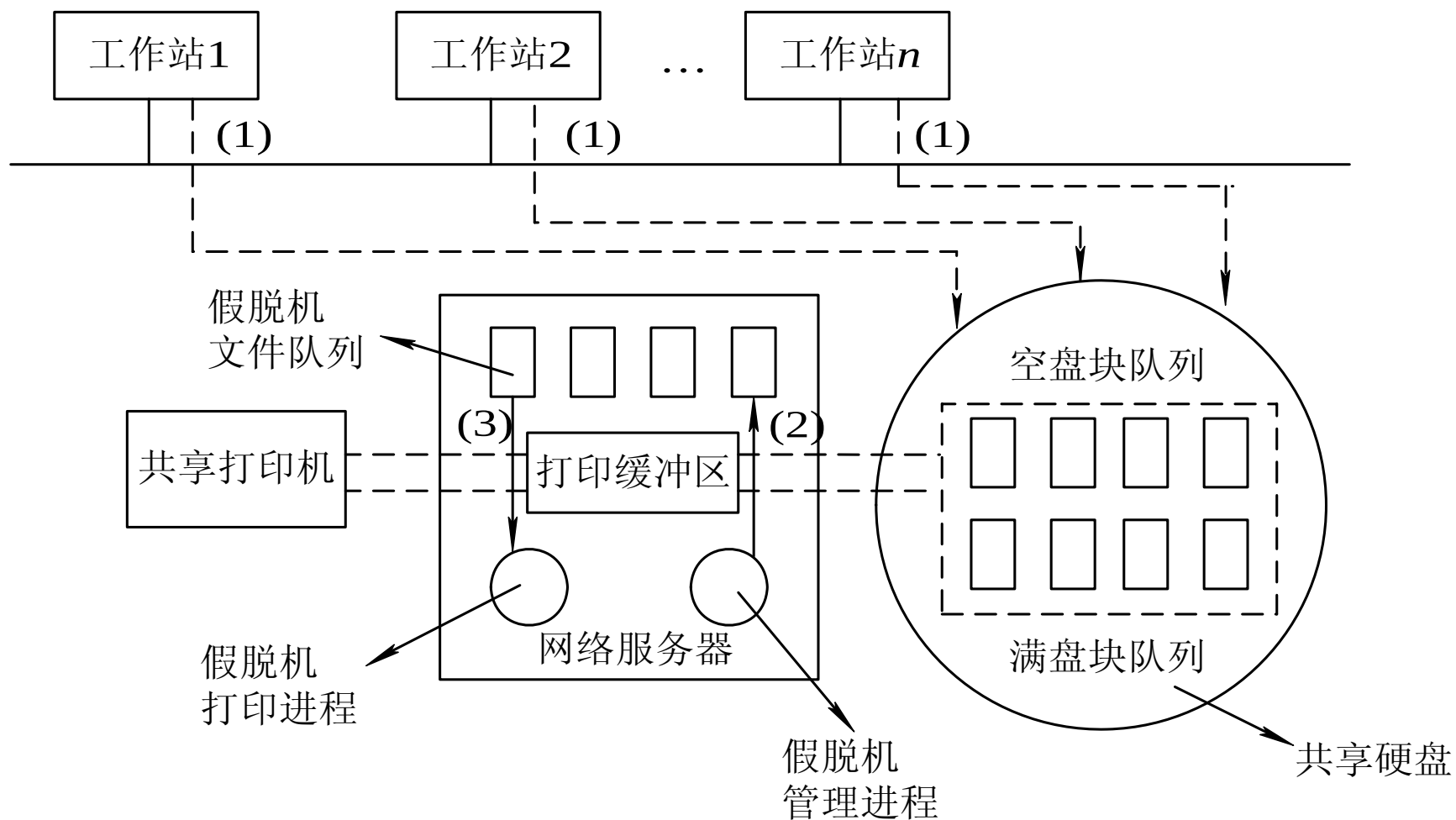


图8-16 以假脱机方式共享打印的示意



2) 共享打印模式

(1) 客户/服务器模式。该模式又可采取两种方式控制共享打印机：① 在局域网中配置一台专门的打印机服务器，用来管理全网中的多个共享打印机。② 由网络(文件)服务器管理共享打印。目前大多数LAN都采用这种方式。这时，网络服务器上可连接一台或多台打印机，由服务器中的文件/打印服务管理程序，统一管理网络服务器中的文件及各共享打印机。此时，对文件的操作应有较高的优先权，它可以暂停正在进行的打印操作。



(2) 对等模式。该模式不仅允许在文件服务器上配置网络打印机，也允许在网络工作站上配置网络打印机。两者可以同样的方式提供网络打印服务。从网络打印服务的角度说，此时所有的服务器和工作站都是对等的，从而形成了完全分布式的网络打印模式，提高了打印服务的质量。



3. 分布式文件系统DFS

在现代大型信息系统中，可供共享的目录和文件通常都是分散地存放在多台计算机(服务器)系统中，这样，用户要访问分散的目录和文件是不方便的，只有将这些分散的目录和文件有机地组织在一起，以形成一个分布式文件系统后，用户才能方便、迅速地访问各个服务器中的共享目录和文件。为此，在Windows NT Server 4.0 (简称NTS 4.0)版本发表后不久，Microsoft就提供了分布式文件系统DFS(Distrabuted File System)，但它并未包含在NTS 4.0中，而是在NTS 5.0(即Windows 2000)版本中内置了经过改造的DFS版本。



当人们已经拥有了一台配置了Windows 2000的计算机系统时，便可利用DFS工具来建立一个分布式文件系统。即先利用DFS工具来建立一个共享目录，称之为DFS根目录；再在此根目录下建立若干个子目录，这些子目录既可以是常规的本地子目录，也可以是一个个连接点。令这些连接点指向一些其它计算机系统上的共享目录和文件，这样，就把人们感兴趣的所有有关的共享目录和文件与DFS根目录下的分布式文件系统建立了连接。



由上所述可知，连接点是构造DFS的关键。在NTS 5.0中共设置了以下几类这种连接点：第一类连接点为Inter-DFS，用于指向另一个计算机系统上的DFS根目录，如图8-17中的\\Server\\Public\\Intranet；第二类连接点是Mid-level，用于指向另一系统中的一个普通目录和文件，如同图中的\\Server\\Public\\Intranet/Corpinfo连接点；第三类是Alternate Volume连接点，它同时指向多个完全相同的文件和目录，用于提供容错功能，如该图中的\\Server\\Public\\Users\\Bob连接点；第四类是Down-level Volume连接点，用于指向F级卷。所谓F级卷，是指非NTS 4.0和NTS 5.0的共享目录和文件，如图中的\\Server\\Public\\Users\\Ray连接点。



第八章 网络操作系统

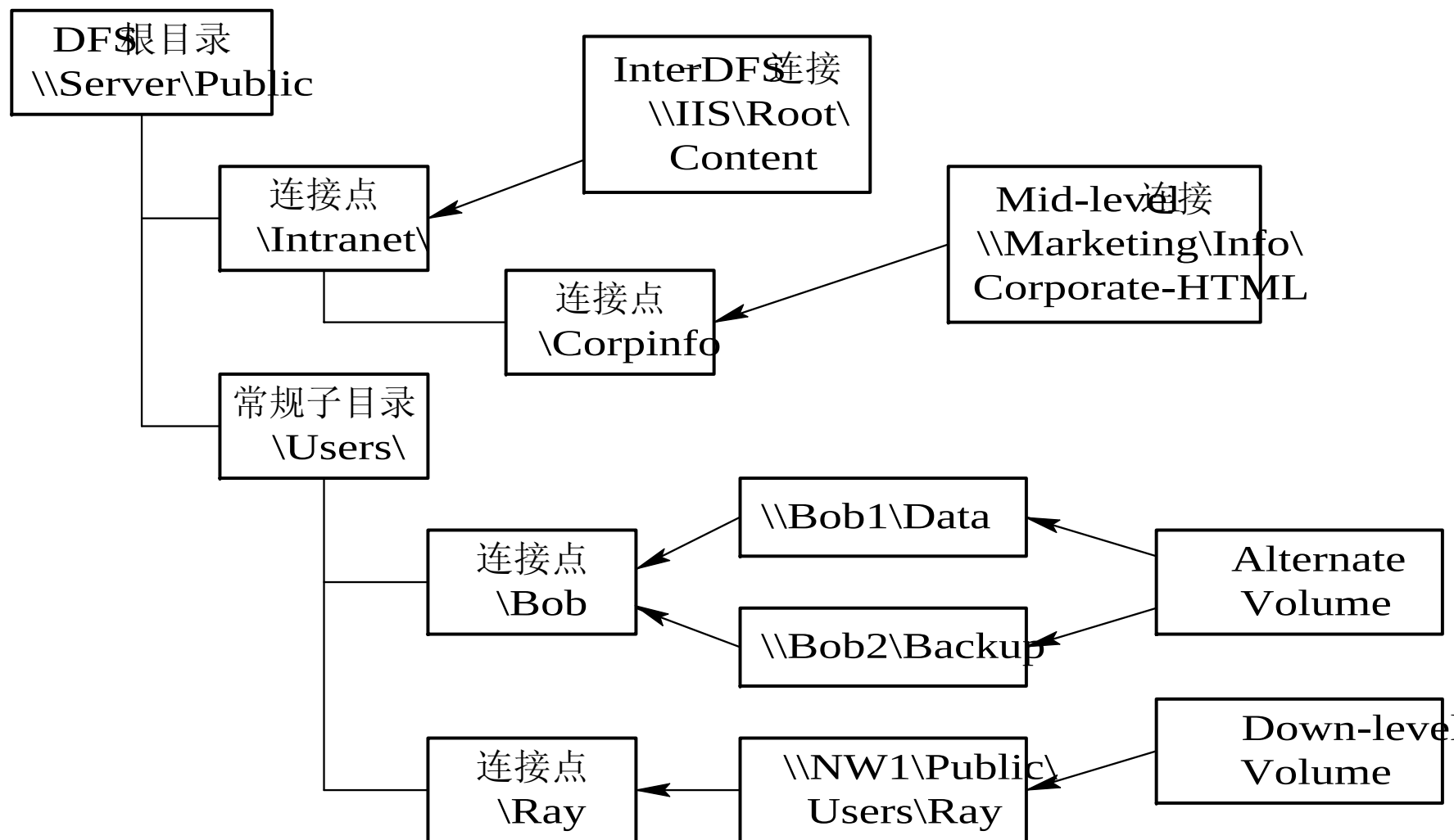


图8-17 分布式文件系统DFS



8.5.3 应用互操作功能

1. 信息的“互通性”

所谓信息的“互通性”，是指在不同网络的结点之间能实现通信。而妨碍信息“互通性”的主要因素，是各个网络使用了各不相同的传输协议。例如在将X.25分组交换网和一个配置了Netware 网络操作系统的以太网互连时，由于在分组交换网中配置的是X.25协议，而Netware 所用的是SPX/IPX协议，因此在这两个网络之间无法实现通信。又如，将一台UNIX机器连接到一个配置了Netware 网络操作系统的以太网上时，由于在UNIX中所用的是TCP/IP协议，因而它无法与以太网上的服务器通信，当然就更不能获得服务器提供的服务。



因此，在完成了网络之间物理上的连接之后，应再采取措施实现信息的互通。实现信息的互通的一种有效方法是互连网络中的所有各网站，都配置同一类型的传输协议，以实现各个网络之间的通信。由于传输控制协议/互联网协议（简称TCP/IP）是当前用得最为广泛的传输协议，几乎所有的广域网和局域网都能支持这种协议，该协议已在事实上成为传输协议的国际标准，因此，目前主要是利用TCP/IP来实现信息的“互通性”。

值得一提的是，当前世界上最大的互连网络为Internet，它是由千千万万个各种类型的WAN互连而成的，上面接入了无数的LAN，所有这些网络都是利用TCP/IP传输协议来实现信息互通性的。如果用户希望把自己的网络连接到Internet上，那么，就应在自己的网络中配置TCP/IP传输协议。



2. 信息的“互用性”

在利用TCP/IP协议实现了不同网络之间信息的“互通性”之后，也只是在各网络之间能进行通信，即只能将某一网络中的一个(批)文件传送到另一个网络中去。但此时，一个网络中的用户并不能访问另一个网络文件系统中的文件，或者说，一个网络中的用户不能操作另一网络文件系统的文件，这是因为两个系统中与文件操作相关的命令是各不相同的，即此时尚不具有信息的“互用性”。所谓信息的“互用性”是指，在不同的网络中的站点之间能实现信息的互用，亦即一个网络中的用户能够去访问另一个网络文件系统(或数据库系统)中的文件(数据)。



然而在互连网络中，由于在不同网络中所配置的网络文件系统(或数据库系统)各不相同，他们各有自己的文件命名方式和存取文件的命令，以及使用了各不相同的文件结构，于是此时便发生了与实现不同网络之间信息互通性时相类似的问题，亦即，由一个源网络中的用户发往一个目标网络去的文件访问命令不能被目标网络所识别。

是否可以采取与信息的“互通性”相同的方法，利用一个网络文件系统协议，来沟通不同网络中的文件系统。遗憾的是，由于网络文件系统协议的复杂性，目前世界上尚无一个类似于TCP/IP那样能被广泛接受的网络文件系统协议。当前相对比较流行的是由SUN公司推出的网络文件系统协议NFS。类似地，信息的“互用性”除了包括能访问另一系统中的网络文件系统之外，还应能解决在不同网络数据库系统中的数据共享问题。



1) 网络文件系统协议NFS

NFS(Network File System)是一种用于TCP/IP网络上的客户/服务器协议。在NFS协议中包括一系列的命令和服务，这些命令和服务涉及到客户访问文件服务器上的文件系统、共享打印机，以及文件传输等，客户还可利用NFS命令去控制和访问远程文件系统；服务器则是根据请求者的请求做相应的处理，并将结果返回给请求者。NFS在提供这些服务时，要利用外部数据表示协议XDR(External Data Representation)和远程过程调用RPC(Remote Procedure Call)，因此，NFS是处于它们的上层，这三者之间的层次关系以及与其它模块之间的关系如图8-18所示。

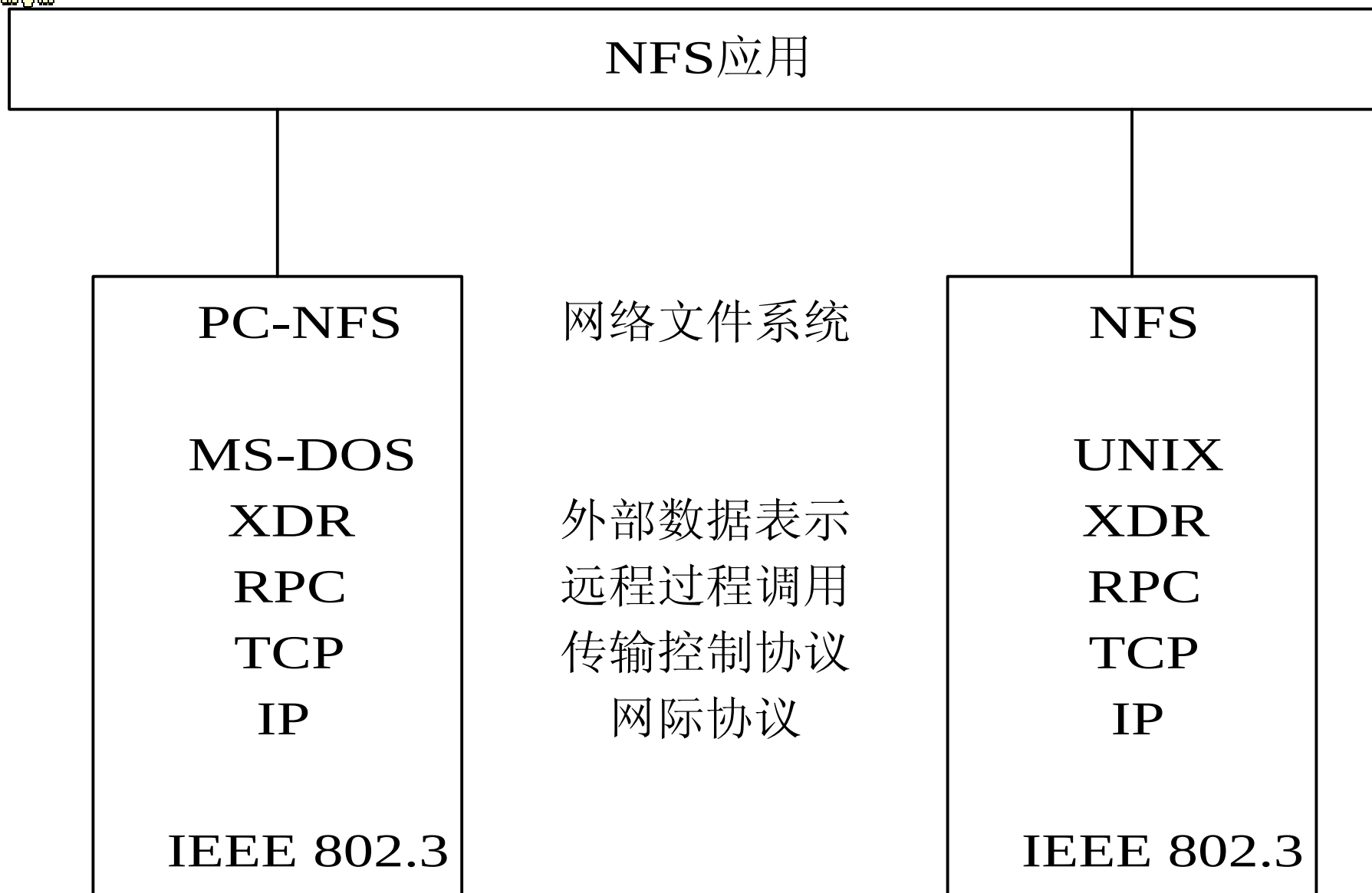


图8-18 NFS、XDR和RPC之间的关系



2) 外部数据表示协议XDR

XDR是处于OSI/RM中的表示层，主要用于处理在不同体系结构中的计算机之间的数据表示不一致时所出现的问题。假如在网络中，有一台客户机上的用户要访问某服务器上的文件系统，此时，它将利用NFS协议中的命令去控制和访问远程文件传输。客户机上的NFS将调用XDR过程，后者把要传送的数据表示转换为与机器无关的数据表示，然后，调用RPC将数据发送给服务器。该服务器上的接收程序又调用XDR过程，把与机器无关的数据表示转换为服务器上的数据表示。XDR所使用的语言与C语言类似，能简捷地描述复杂的数据结构。RPC和NFS等协议都可使用XDR来描述它的数据结构。



8.5.4 网络管理功能

1. 网络管理的目标

(1) 增强网络的可用性。可以通过多种途径来增强网络的可用性：① 通过预测及时地检测出网络设备和线路的故障，并迅速采取措施将故障修复；② 采取冗余措施，亦即，为关键设备和线路配置冗余的设备和线路。



(2) 提高网络的运行质量。为了提高网络的运行质量，应随时对网络中各主要设备的负荷及各线路上的流量进行监测，以便及时发现问题，及时进行调整。

(3) 提高网络的资源利用率。通过网管功能，对网络中的关键设备和线路进行长期的监测，然后根据监测资料来充分、准确地掌握网络中各网络设备和线路的利用情况，在此基础上再对网络资源进行合理的调整，以提高这些资源的利用率。



(4) 保障网络数据的安全性。对安全性的基本要求是：所有在网络中保存的文件和数据，不能被非核准的用户截获，更不能被他们修改和伪造。为了保障网络中文件和数据的安全性，通常采取了多级安全保障机制。

(5) 提高网络的社会和经济效益。通过增强网络的可靠性，改善和提高网络资源的利用率，保障网络中数据的安全性，以及提供网络的优质服务等措施，可以吸引更多的用户入网，降低网络的运行成本，这样也就有效地提高了网络的社会和经济效益。



2. 网络管理的功能

网络管理功能涉及到网络资源和活动的规划、组织监视、计算和控制等方面。已往各公司所推出的网络管理软件，都各有所侧重；而国际标准化组织(ISO)为网络管理定义了差错、配置、性能、计费和安全五大管理功能。

(1) 配置管理。配置管理涉及到定义、收集、监视和控制以及使用配置数据。配置数据包括网络中重要资源的静态和动态信息，这些数据通常会被广泛使用。配置管理用来监控网络的配置数据，允许网络管理人员能生成、查询和修改软硬件的运行参数及条件，以保证网络的正常运行。



(2) 故障管理。故障管理的主要目的是为网络操作员提供快速发现和修复故障的手段。故障管理设施通常是用来检测网络中所发生的异常事件，以发现故障，然后根据故障的现象采取相应的跟踪、诊断和测试措施；还要在日志上记录下故障情况。

(3) 性能管理。通过收集网络各部分使用情况的统计数据，来分析网络的运行情况，如网络的响应时间、网络的吞吐量、网络的阻塞情况，以及网络的运行趋势，从而可以得出对网络的整体和长期的评价；也可以通过性能管理，将网络性能控制在用户能接受的水平。



(4) 安全管理。由安全管理提供某种安全策略，据以实现对受管资源的访问。安全管理的主要工作有：① 操作员管理：定义合法操作员及其权限和管理域；② 访问控制管理：规定网络用户对网络资源的访问权限，限制非法用户对网络资源的访问；③ 审计管理：通过使用日志等手段，对所关心的事件进行调查；④ 密钥分配：为保密设备的保密协议分配应使用的密钥等；⑤ 防止病毒。

(5) 计费管理。计费管理用于监视和记录用户使用网络资源的种类、数量和时间，对用户所分配到的资源的使用进行计费。在计费管理中所涉及到的具体功能有：搜集计费记录，计算用户账单，网络经济预算，检查资费变更情况，分配网络运行成本等。



3. 网络管理模型

在现代网络中，普遍采用管理者/代理者模型。管理者是指驻留在管理系统中的，用于发出管理命令和接收代理者发来的通知的软件；代理者是指驻留在受管对象系统中，用于接收并执行管理者发来的命令，提供受管对象的视图并发出用于反映受管对象行为的通知的软件。在基于管理者/代理者模型的网络管理中，网络管理软件是由网络管理者(软件)和代理程序Agent组成的。网络管理功能的实现，是通过它们之间的多次交互来完成的。图8-19是管理者、代理者以及它们之间交互的示意图。

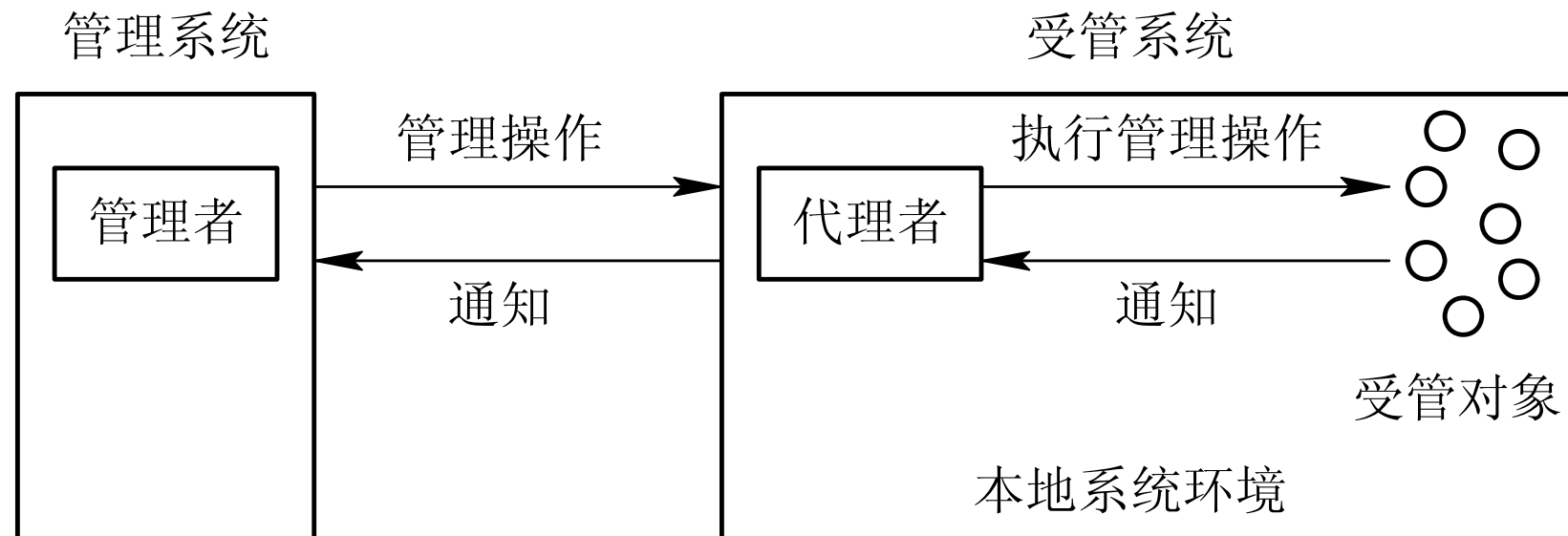


图8-19 管理者/代理者及它们之间的交互



8.6 网络操作系统提供的服务

8.6.1 域名系统(DNS)

1. 域名系统的层次结构

如上所述，Internet的域名是由几段构成的，相应地，域名空间呈现倒树形结构，即它可分为若干个层次，顶层是树根，在树根下是若干个顶级域名，再下是二级域名、三级域名和四级域名。域名系统(Domain Name System)同样也采用倒树形结构，它对应于域名的层次。在DNS的顶部是根服务器，下面是若干个顶级域名服务器，再下面是二级域名服务器……。由上级域名服务器管理属于它的下一级域名服务器。虽说域名系统的层次结构对应于域名的层次结构，但两者中每一层的成分并不一一对应。



例如，一个企业网络将可以选择的所有域名放在一个域名服务器中，也可以选择运行在若干个域名服务器中；同样，在一个域名服务器中都只装入域名体制中对应层次的一部分域名和IP地址。这里，一部分的含义是：它可以是完全对应于一个域名服务器，也可以是几个域名服务器。当前大多数连接到Internet上的公司、学校等网络，都具有一个本地域名服务器，每个服务器都包含了连接到其它域名服务器的信息，这样，由这一大批(成千上万个)服务器形成一个大型的分布式数据库系统。



2. 域名解析

在域名系统中的每台本地域名服务器，都配置了一个域名解析器软件。所谓域名解析，是指将主机域名转换为对应的IP地址，而把完成该功能的软件称为域名解析器。由于每个域名服务器都掌握其下属的域名服务器的地址和根域名服务器的地址，因此，经过几次查询，在域名树中总可以找到用户所需域名对应的IP地址。DNS是基于客户/服务器模式的系统，因此在查询IP时，通常需要经过多次客户和服务器之间的交互。



当一个用户A进程要查询用户B所在主机的域名时，他可以按下述步骤进行：

- (1) 客户首先向本地域名服务器发出查询请求包IP(netlab.cs.nakai.edu.cn)；
- (2) 如果本地服务器不知道该域名，便向顶级域名服务器发出查询请求包IP(netlab.cs.hankai.edu.cn)；
- (3) 顶级服务器通常不知道查询的域名，但它知道所有下属域名服务器的IP地址(202.113.16.10)，因此，它便将所查询的第二级域名服务器的IP地址，返回给本地服务器；
- (4) 本地服务器又向第二级域名服务器进行查询；



(5) 由于它仍不知道所查询的域名，于是它便将所查询的第三级域名服务器的IP地址(202. 113. 27. 1)返回给本地服务器；

(6) 本地服务器又向第三级域名服务器进行查询；

(7) 由于在第三级服务器中找到了所查询的域名，于是把该域名所对应的IP地址(202. 113. 56. 10)回送给本地服务器；

(8) 由本地服务器将客户所查询的域名对应的IP地址传送给客户机。



第八章 网络操作系统

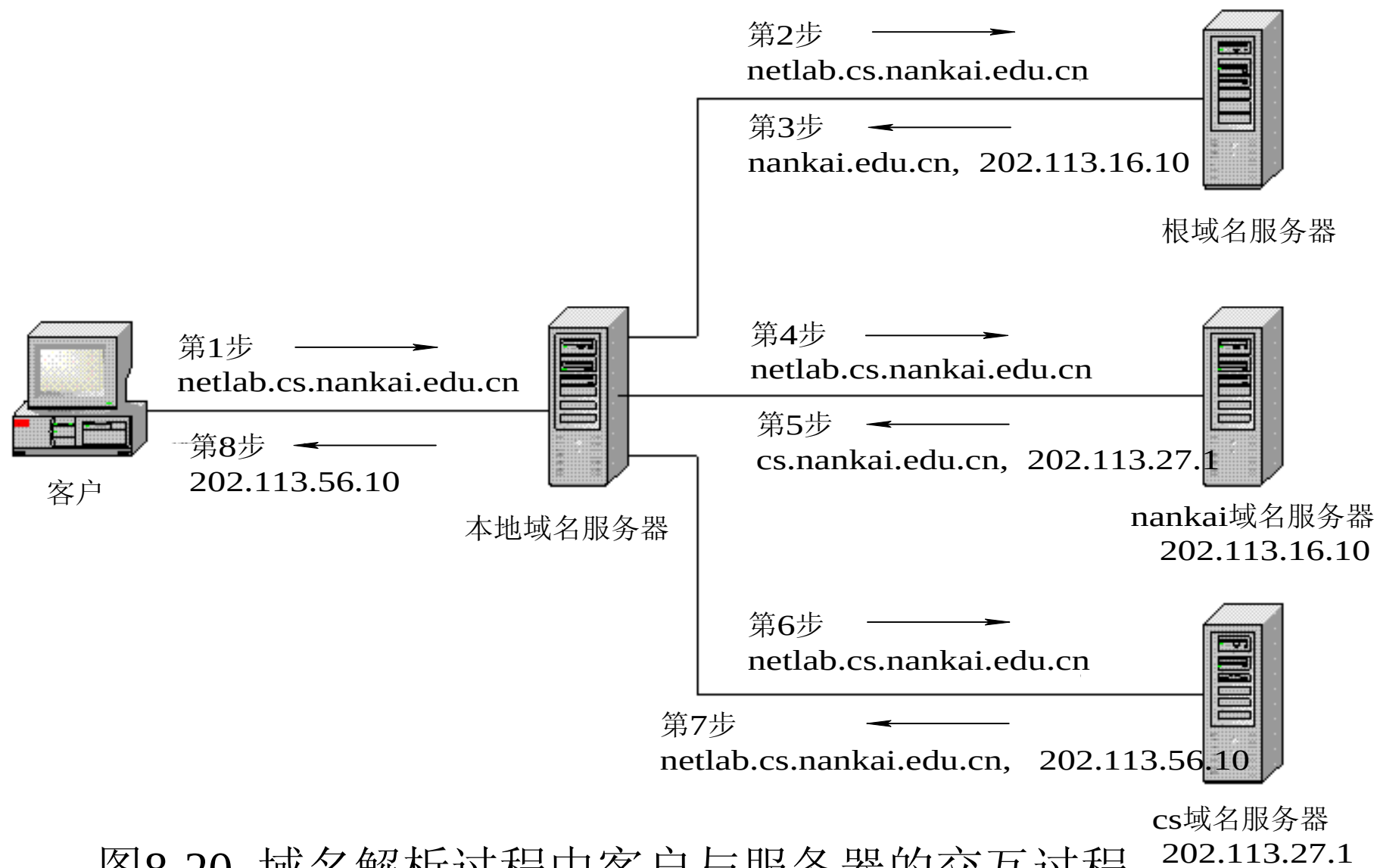


图8-20 域名解析过程中客户与服务器的交互过程



8.6.2 目录服务

1. 目录服务管理的对象

1) 物理设备

物理设备包括网络中的所有网络工作站、各种类型的服务器、网桥、路由器等。为了对这些物理设备进行管理，通常为它们建立一张目录表，表中的每个目录项中可以包括物理设备的标识符、设备类型。对工作站而言的设备类型，系指工作站所配置的OS，于是设备类型就有Windows 2000、Windows XP、UNIX 等；对服务器而言的设备类型，则是指其所提供服务的类型，于是有文件/打印服务器、数据库服务器、电子邮件服务器、文件传输服务器、Web服务器等类型。此外，目录项中还包括设备的物理地址。



2) 网络服务

目录服务还应能对每台物理设备所提供的网络服务进行管理。对于服务器而言，它所提供的网络服务可以是文件/打印服务、数据库服务等；而对于某些工作站而言，它所提供的服务则取决于在工作站上所安装的应用。

3) 用户

在目录服务中也把用户作为管理对象。用户管理往往是通过用户对用户账户的管理实现的。为此，系统为每个用户设置一个账户，其中记录该用户的有关信息，包括用户名、用户口令、允许用户访问的时间段、允许用户使用的计算机名、用户账户的有效时限等。



2. 目录服务的功能

1) 用户管理

用户管理的主要任务，是保证核准用户能够方便地访问各种网络服务，禁止非核准用户的访问。对用户的管理是采用注册和登录的方式。所谓注册，是指由网络管理员在服务器上为用户建立账户、设置口令和设置对资源的访问权限等；登录则是指每当用户要求上网时，便输入自己的账户名和口令，如果正确，即允许用户上网访问网上的各种自己有权访问的资源。如果用户希望访问网络中N台服务器，就必须在这N台服务器上都要进行登录，才可以对这N台服务器进行访问。但这样会使用户感到非常不便。有了目录服务后，任何用户都只需登录一次，便能访问网上所有被授权访问的资源。



2) 分区和复制功能

随着网络规模的扩大和被管理对象的增多，在目录库中的目录项就非常多。如果将这些目录集中存放在某一个服务器上(集中式目录服务)，则每当用户访问网络资源时，都要求先访问该服务器，这不仅会使网络上通往目录服务器的信息流量增加，而且也潜在着不可靠性。一种行之有效的方法是将一个庞大的目录库分成若干个分区，再将这些分区的目录库分别复制到多台服务器中，且使每个分区被复制的位置尽量靠近最常使用这些对象的用户，在有的目录服务中还允许在一台服务器上存放多个不同分区的拷贝。



3) 创建、扩充和继承功能

由于当前的一些目录服务都采用了面向对象的结构，这使得对目录对象的管理变得简单而有效。在这种管理中包括三方面的功能：

(1) 创建。在目录中创建新的对象，并对新的对象进行属性设置。

(2) 扩充。指对原有的目录服务进行功能上的扩充。

(3) 继承。目录管理下的继承，是指目录对象继承其它对象的属性和权力的能力。不论是Novell公司还是Microsoft公司，都提供了用于创建、扩充和继承的目录管理工具。



4) 多平台支持功能

对于一个大型企业网络，通常可能包含多种类型的网络工作站和网络服务器，因而其目录服务也存在着在管理对象上的差异。这就要求目录服务具有跨越平台的能力，亦即，要求目录服务能支持网络上各种类型的工作站和服务器，更进一步，则要求目录服务应该与企业网络的平台无关，具有支持多种类型平台的能力。



3. 目录服务带来的好处

(1) 简化了网络管理。当要在一个含有数百个服务器、数千个用户的网络中，增加一台新服务器时，如果没有目录服务的帮助，则为使所有用户都能访问这台服务器，就需由网络管理员将这数千个用户的标识和口令等信息，以人工方式输入到新服务器上。但在设置了目录服务之后，只需为新服务器在网络的目录树上建立一个新的目录结点(项)，这样，网络上的所有用户便都可以访问该服务器。可见，设置目录服务大大地方便了网络管理员。



(2) 方便了用户入网和访问。在无目录服务时, 用户为了进入同一网络(或互连网络)中分散在各地的多个系统(服务器), 必须在各系统中都建立该用户的账户, 从而造成了一个用户有多个账户的情况。多账户一直是网络管理中最麻烦的事, 因为这样不仅要求用户和管理员都持有多多个账户的记录, 记住每个账户所用的口令, 而且通常还要采取许多安全性措施, 以解决多个账户之间的一致性问题。但在有了目录服务后, 便可实现一个用户一个账户的单一登录性, 也允许用户从网络中的任一结点对网络中的各个服务器进行访问。在访问时, 网络上的业务还可以从一台服务器转移到另一台服务器。



(3) 提高了网络的可用性。由于一方面在网络上的所有重要设备和所提供的服务，在目录中都有相应的目录项，在其中有着较详细的记录；另一方面，用户请求的网络服务并非限定是仅由某个服务器提供的，因此，当网络中有某个(些)服务器发生故障时，目录服务可以及时发现，并可用户对该服务器的访问请求转送到其它服务器上，从而仍可保持网络的正常工作。



8.6.3 支持Internet提供的服务

1. Web浏览器软件和Web服务器软件

用户广泛使用的Web服务具有这样三个特点：一是它所管理和传输的信息是超文本文件，是由超文本标识语言HTML编写的文件(称为HTML文件)；二是Web所使用的基本协议是超文本传输协议HTTP；三是HTML文件是由统一资源定位器URL来惟一标识的。



1) Web浏览器软件

为使客户机上的用户能进入Internet，在客户机上必须安装Web浏览器软件，安装了浏览器软件的客户机被称为浏览器，它是用户与Internet网之间的接口。用户可通过浏览器上网浏览。浏览器的基本功能是向Web服务器提出服务请求，并显示由Web服务器制作的显示信息。它本身并不具有很强的信息处理能力，因为信息的处理是由服务器来完成的。为了实现它与Web服务器之间的交互，浏览器必须支持HTTP协议。



2) Web服务器软件

如前所述，在LAN中配置了Web服务器，而在Web服务器上所配置的软件，是用于向Web浏览器提供Web服务的，由此形成了浏览器/服务器模式。在小型LAN中可能只配置了一台综合服务器，此时该服务器兼作数据服务器和Web服务器等，这样，网络上的用户可通过浏览器向Web服务器提出请求，实现上网浏览的功能。政府部门、公司、学校等可以在Web服务上发布信息和广告。



2. 安装E-mail、FTP等多种服务软件

与Web类似,为使用户能获得Internet所提供的其它服务,如电子邮件服务、文件传输服务等,在客户机上也应当安装相应的协议软件(客户部分),在LAN中也需要配置相应的服务器软件。显然这是一件相当繁琐的事。所幸的是,近几年来所推出的新的浏览器软件,已不是单纯地实现浏览器功能,还包括许多其它的服务功能,如E-mail服务、FTP服务、Telnet服务等。通常把这样的浏览器软件称为浏览器套件,如微软公司推出的Internet Explorer,简称IE,其IE 4.0、5.0和6.0版本都集成了许多其它服务软件。在近几年所推出的Web服务器软件中,除了能提供Web服务,同样也包括了许多其它的服务功能。如由微软推出的IIS(Internet Information Server),它就能支持Internet所提供的多种服务,如Web、FTP、E-mail、SMTP等。



3. 在客户机上配置TCP/IP协议软件

为实现用户上网功能，在客户机上还必须配置TCP/IP协议软件，这可归结为如下两方面的原因：

1) 实现信息互通

当某个局域网LAN要和Internet互连时，首先必须解决的问题是信息互通，即从LAN某客户机上发出的信息，能顺利地进入Internet并到达指定的目标。由于在Internet中用于信息传输的是TCP/IP协议，这就要求LAN也必须支持TCP/IP协议，因此，在LAN中的客户机上都必须配置支持TCP/IP的协议软件。



2) 建立在TCP/IP基础上的Internet

在浏览器和服务端之间交互时，必须使用HTTP协议。而该协议是建立在TCP/IP协议基础上的。另外，Internet所提供的许多服务也都是建立在TCP/IP上的。因此，要求LAN能支持TCP/IP协议，也就要求所有的客户机上都配置TCP/IP协议软件。事实上，从上世纪90年代中期以后推出的OS，如Windows 2000、Windows XP以及UNIX，都无一例外地配置了TCP/IP协议软件。



第九章 系统安全性

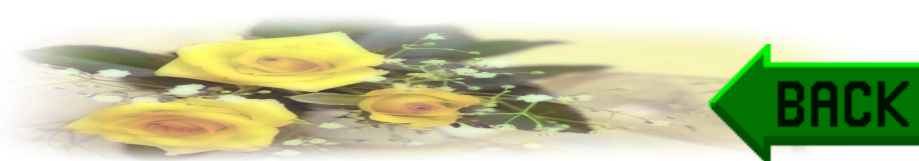
9.1 系统安全的基本概念

9.2 数据加密技术

9.3 认证技术

9.4 访问控制技术

9.5 计算机病毒



BACK

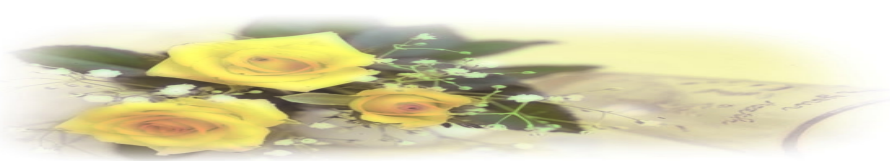


9.1 系统安全的基本概念

9.1.1 系统安全性的内容和性质

1. 系统安全性的内容

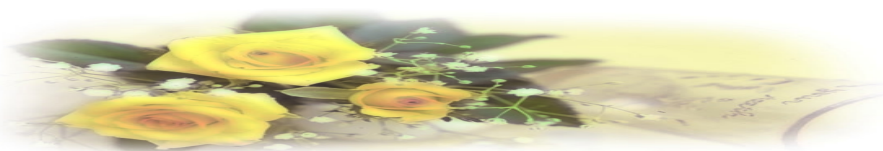
(1) 数据机密性(Data Secrecy): 指将机密的数据置于保密状态, 仅允许被授权的用户访问计算机系统中的信息(访问包括显示和打印文件中的信息)。





(2) 数据完整性(Data Integrity): 指未经授权的用户不能擅自修改系统中所保存的信息, 且能保持系统中数据的一致性。这里的修改包括建立和删除文件以及在文件中增加新内容和改变原有内容等。

(3) 系统可用性(System Availability): 指授权用户的正常请求能及时、正确、安全地得到服务或响应。或者说, 计算机中的资源可供授权用户随时进行访问, 系统不会拒绝服务。但是系统拒绝服务的情况在互联网中却很容易出现, 因为连续不断地向某个服务器发送请求就可能会使该服务器瘫痪, 以致系统无法提供服务, 表现为拒绝服务。

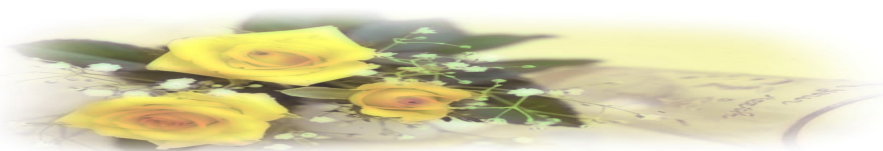




2. 系统安全的性质

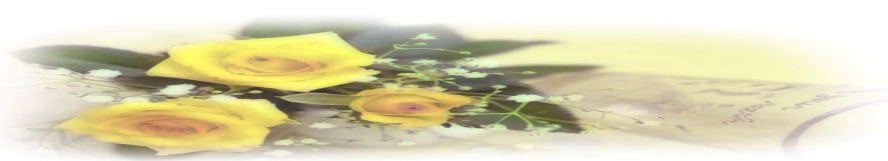
系统安全问题涉及面较广，它不仅与系统中所用的硬、软件设备的安全性能有关，而且也与构造系统时所采用的方法有关，这就导致了系统安全问题的性质更为复杂，主要表现为如下几点：

(1) 多面性。在较大规模的系统中，通常都存在着多个风险点，在这些风险点处又都包括物理安全、逻辑安全以及安全管理三方面的内容，其中任一方面出现问题，都可能引起安全事故。



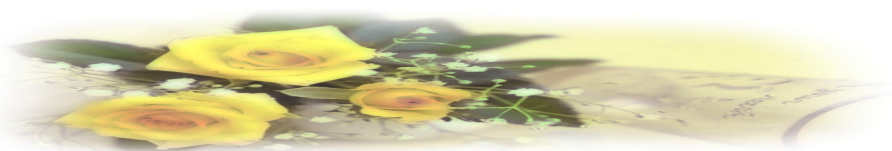


(2) 动态性。由于信息技术的不断发展和攻击者的攻击手段层出不穷，使得系统的安全问题呈现出动态性。例如，在今天还是十分紧要的信息，到明天可能就失去了作用，而同时可能又产生了新的紧要信息；又如，今天还是多数攻击者所采用的攻击手段，到明天却又较少使用，而又出现了另一种新的攻击手段。这种系统安全的动态性，导致人们无法找到一种能将安全问题一劳永逸地解决的方案。



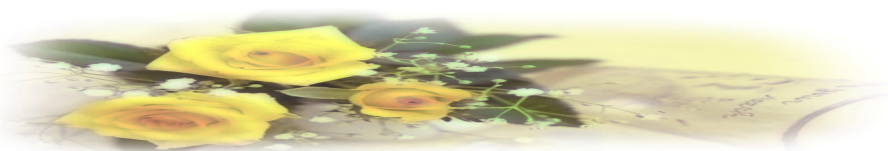


(3) 层次性。系统安全是一个涉及诸多方面、且相当复杂的问题，因此需要采用系统工程的方法来解决。如同大型软件工程一样，解决系统安全问题通常也采用层次化方法，将系统安全的功能按层次化方式加以组织，即首先将系统安全问题划分为若干个安全主题(功能)作为最高层；然后再将其中一个安全主题划分成若干个子功能作为次高层；此后，再进一步将一个子功能分为若干孙功能；其最低一层是一组最小可选择的安全功能，它不可再分解。这样，利用多个层次的安全功能来覆盖系统安全的各个方面。





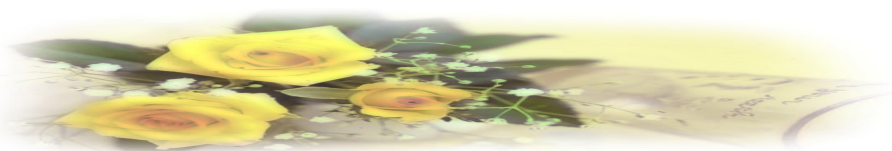
(4) 适度性。当前几乎所有的企、事业单位在实现系统安全工程时，都遵循了适度安全的准则，即根据实际需要，提供适度的安全目标加以实现。这是因为：一方面，由于系统安全的多面性和动态性，使得对安全问题的全面覆盖难于实现；另一方面，即使是存在着这样的可能，其所需的资源和成本之高，也是难以令人接受的。这就是系统安全的适度性。





9.1.2 系统安全威胁的类型

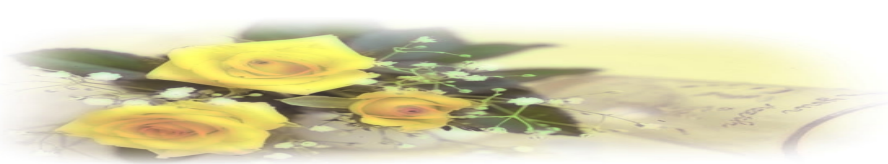
(1) 假冒(Masquerading)用户身份。这种类型也称为身份攻击，指用户身份被非法窃取，亦即，攻击者伪装成一个合法用户，利用安全体制所允许的操作去破坏系统安全。在网络环境下，假冒者又可分为发方假冒和收方假冒两种。为防止假冒，用户在进行通信或交易之前，必须对发方和收方的身份进行认证。





(2) 数据截取(Data Interception)。未经核准的人可能通过非正当途径截取网络中的文件和数据，由此造成网络信息的泄漏。截取方式可以是直接从电话线上窃听，也可以是利用计算机和相应的软件来截取信息。

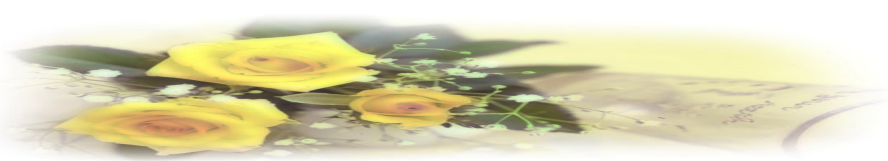
(3) 拒绝服务(Denial of Server)。这是指未经主管部门的许可，而拒绝接受一些用户对网络中的资源进行访问。比如，攻击者可能通过删除在某一网络连接上传送的所有数据包的方式，使网络表现为拒绝接收某用户的数据；还可能是攻击者通过修改合法用户的名字，使之成为非法用户，从而使网络拒绝向该用户提供服务。





(4) 修改(Modification)信息。未经核准的用户不仅可能从系统中截获信息，而且还可以修改数据包中的信息，比如，可以修改数据包中的协议控制信息，使该数据包被传送到非指定的目标；也可修改数据包中的数据部分，以改变传送到目标的消息内容；还可能修改协议控制信息中数据包的序号，以搅乱消息内容。

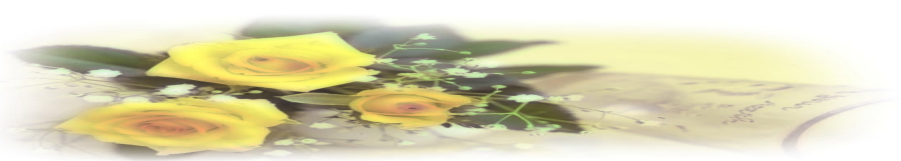
(5) 伪造(Fabrication)信息。未经核准的人可将一些经过精心编造的虚假信息送入计算机，或者在某些文件中增加一些虚假的记录，这同样会威胁到系统中数据的完整性。





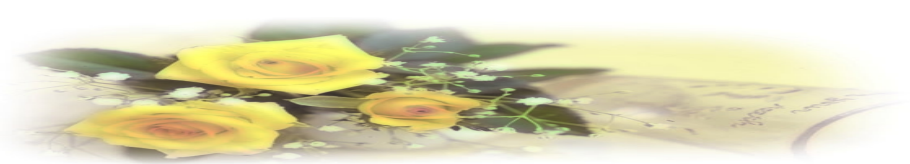
(6) 否认 (Repudiation) 操作。这种类型又称为抵赖，是指某人不承认自己曾经做过的事情。如某人在向某目标发出一条消息后却又矢口否认；类似地，也指某人在收到某条消息或某笔汇款后不予承认的做法。

(7) 中断 (Interruption) 传输。这是指系统中因某资源被破坏而造成信息传输的中断。这将威胁到系统的可用性。中断可能由硬件故障引起，如磁盘故障、电源掉电和通信线路断开等；也可能由软件故障引起。





(8) 通信量分析(Traffic Analysis)。攻击者通过窃听手段窃取在线路中传输的信息，再考察数据包中的协议控制信息，可以了解到通信者的身份、地址；通过研究数据包的长度和通信频度，攻击者可以了解到所交换数据的性质。

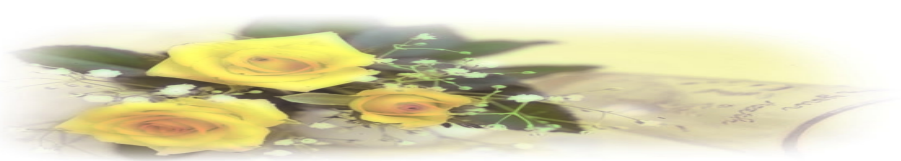




9.1.3 信息技术安全评价公共准则

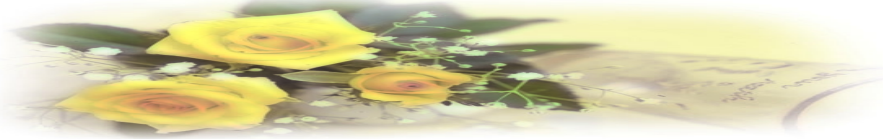
1. CC的由来

对一个安全产品(系统)进行评估,是件十分复杂的事,它对公正性和一致性要求很严。因此,需要有一个能被广泛接受的评估标准。为此,美国国防部在20世纪80年代中期制定了一组计算机系统安全需求标准,共包括20多个文件,每个文件都使用了彼此不同颜色的封面,统称为“彩虹系列”。其中最核心的是具有橙色封皮的“可信任计算机系统评价标准(TCSEC)”,简称为“橙皮书”。





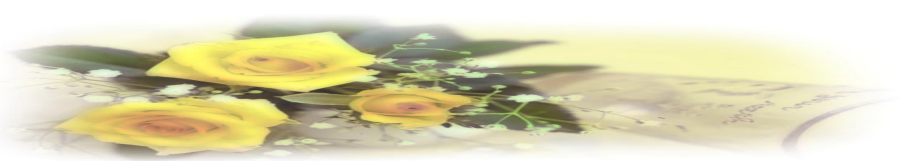
该标准中将计算机系统的安全程度划分为8个等级，有D₁、C₁、C₂、B₁、B₂、B₃、A₁和A₂。在橙皮书中，对每个评价级别的资源访问控制功能和访问的不可抵赖性、信任度及产品制造商应提供的文档作了一系列的规定，其中以D₁级为安全度最低级，称为安全保护欠缺级。常见的无密码保护的 personal 计算机系统便属于D₁级。C₁级称为自由安全保护级，通常具有密码保护的多用户工作站便属于C₁级。C₂级称为受控存取控制级，当前广泛使用的软件，如UNIX操作系统、ORACLE数据库系统等，都能达到C₂级。从B₁级开始，要求具有强制存取控制和形式化模型技术的应用。B₃、A₁级进一步要求对系统中的内核进行形式化的最高级描述和验证。一个网络所能达到的最高安全等级，不超过网络上其安全性能最低的设备(系统)的安全等级。





2. CC的组成

CC由两部分组成，一部分是信息技术产品的安全功能需求定义，这是面向用户的，用户可以按照安全功能需求来定义“产品的保护框架(PP)”，CC要求对PP进行评价以检查它是否能满足对安全的要求；CC的另一部分是安全保证需求定义，这是面向厂商的，厂商应根据PP文件制定产品的“安全目标文件”(ST)，CC同样要求对ST进行评价，然后根据产品规格和ST去开发产品。

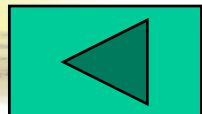
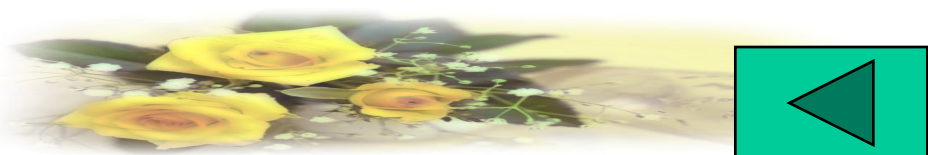


第九章 系统安全性



CC的安全功能需求部分包括一系列的安全功能定义，它们是按层次式结构组织起来的，其最高层为类(Class)。CC将整个产品(系统)的安全问题分为11类，每一类侧重于一个安全主题。中间层为簇(Family)，在一类中的若干个簇都基于相同的安全目标，但每个簇各侧重于不同的方面。最低层为组件(Component)，这是最小可选择的安全功能需求。安全保证需求部分同样是按层次式结构组织起来的。

必须指出的是，保障计算机和系统的安全性将涉及到许多方面，其中有工程问题、经济问题、技术问题、管理问题，甚至涉及到国家的立法问题。但在此，我们仅限于介绍用来保障计算机和系统安全的基本技术，包括认证技术、访问控制技术、密码技术、数字签名技术、防火墙技术等等。



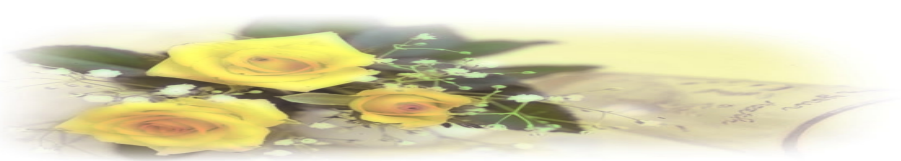


9.2 数据加密技术

9.2.1 数据加密的基本概念

1. 数据加密技术的发展

密码学是一门既古老又年轻的学科。说它古老，是因为早在几千年前，人类就已经有了通信保密的思想，并先后出现了易位法和置换法等加密方法。到了1949年，信息论的创始人香农(C.E.Shannon)论证了由传统的加密方法所获得的密文几乎都是可攻破的，这使得密码学的研究面临着严重的危机。

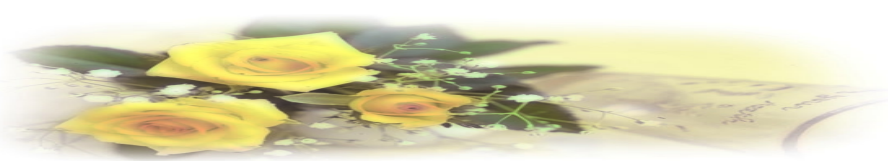




2. 数据加密模型

一个数据加密模型如图9-1 所示。它由下述四部分组成。

- (1) 明文(plain text)。准备加密的文本，称为明文P。
- (2) 密文(cipher text)。加密后的文本，称为密文Y。
- (3) 加密(解密)算法E(D)。用于实现从明文(密文)到密文(明文)转换的公式、规则或程序。
- (4) 密钥K。密钥是加密和解密算法中的关键参数。



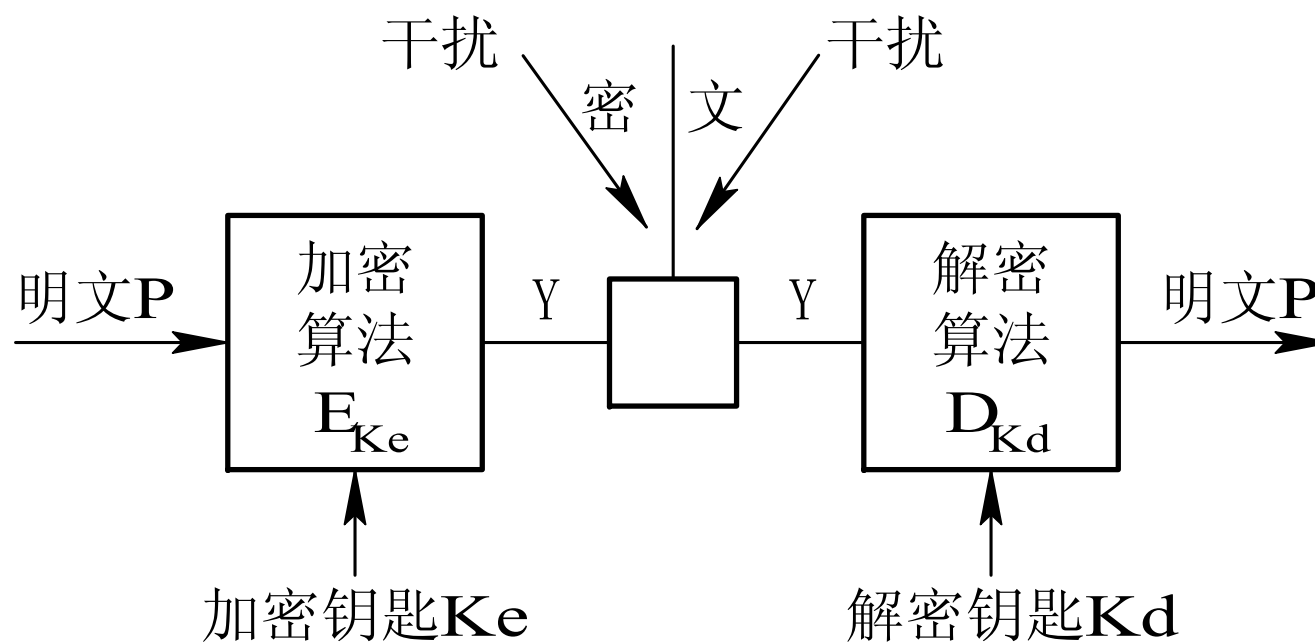
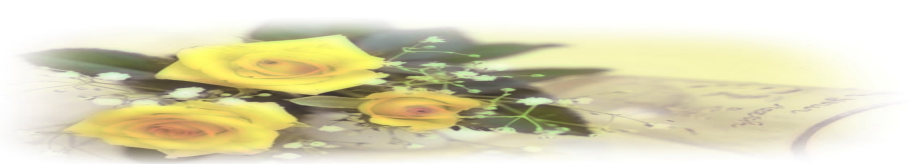


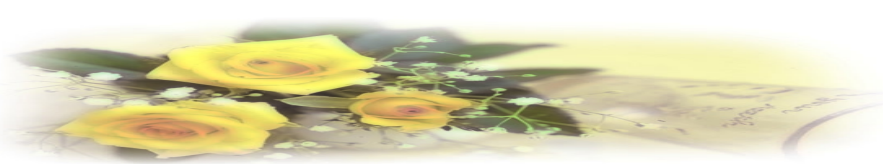
图9-1 数据加密模型





加密过程可描述为：在发送端利用加密算法 E_{K_e} 和加密密钥 K_e 对明文 P 进行加密，得到密文 $Y=E_{K_e}(P)$ 。密文 Y 被传送到接收端后应进行解密。解密过程可描述为：接收端利用解密算法 D_{K_d} 和解密密钥 K_d 对密文 Y 进行解密，将密文恢复为明文 $P=D_{K_d}(Y)$ 。

在密码学中，把设计密码的技术称为密码编码，把破译密码的技术称为密码分析。密码编码和密码分析合起来称为密码学。在加密系统中，算法是相对稳定的。为了加密数据的安全性，应经常改变密钥，例如，在每加密一个新信息时改变密钥，或每天、甚至每个小时改变一次密钥。

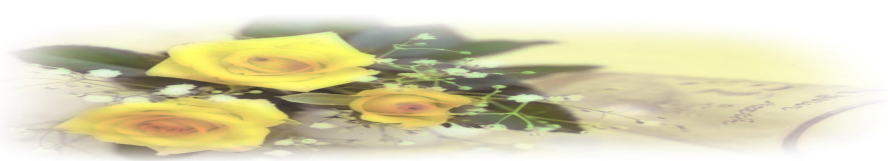




3. 加密算法的类型

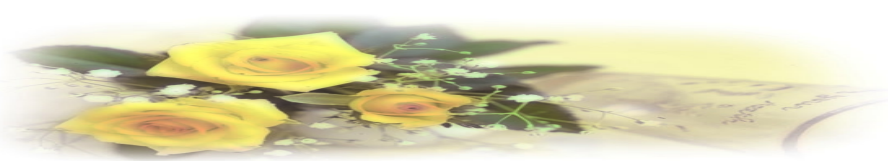
1) 按其对称性分类

(1) 对称加密算法。在这种方式中，在加密算法和解密算法之间存在着一一定的相依关系，即加密和解密算法往往使用相同的密钥；或者在知道了加密密钥 K_e 后，就很容易推导出解密密钥 K_d 。该算法中的安全性在于双方能否妥善地保护密钥，因而把这种算法称为保密密钥算法。该算法的优点是加密速度快，但密钥的分配与管理复杂。





(2) 非对称加密算法。这种方式的加密密钥 K_e 和解密密钥 K_d 不同，而且难以从 K_e 推导出 K_d 来。可以将其中的一个密钥公开而成为公开密钥，因而把该算法称为公开密钥算法。用公开密钥加密后，能用另一把专用密钥解密；反之亦然。该算法的优点是密钥管理简单，但加密算法复杂。



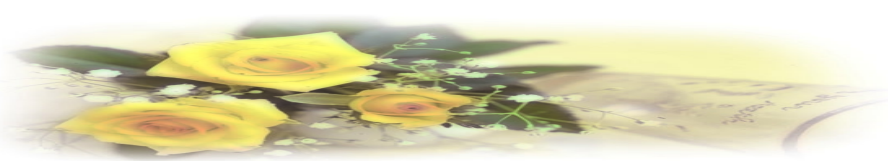


2) 按所变换明文的单位分类

(1) 序列加密算法。该算法是把明文P看做是连续的比特流或字符流 $P_1, P_2, P_3, \dots$ ，在一个密钥序列 $K=K_1, K_2, K_3, \dots$ 的控制下，逐个比特(或字符)地把明文转换成密文。具体可表达成：

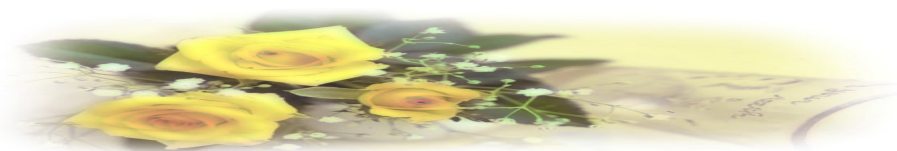
$$E_K(P)=E_{K1}(P_1), E_{K2}(P_2), E_{K3}(P_3), \dots$$

这种算法可用于对明文进行实时加密。





(2) 分组加密算法。该算法是将明文 P 划分成多个固定长度的比特分组，然后，在加密密钥的控制下，每次变换一个明文分组。最著名的DES算法便是以64位为一个分组进行加密的。





4. 基本加密方法

1) 易位法

易位法是指按照一定的规则，重新安排明文中的比特或字符的顺序来形成密文，而字符本身保持不变。按易位单位的不同又可分成比特易位和字符易位两种易位方式。前者的实现方法简单易行，并可用硬件实现，主要用于数字通信中；而后者即字符易位法则是利用密钥对明文进行易位后形成密文。字符易位的具体方法是：假定有一密钥MEGABUCK，其长度为8，则其明文是以8个字符为一组写在密钥的下面，如图9-2所示。按密钥中字母在英文字母表中的顺序来确定明文排列后的列号。如密钥中的A所对应的列号为1，B为2，C为3，E为4等。然后再按照密钥所指示的列号，先读出第1列中的字符，读完第1列之后，再读出第2列中的字符……。这样，即完成了将明文Please transfer ...转换为密文AFLLSKSOSELAWAIA...的加密过程。





M	E	G	A	B	U	C	K
7	4	5	1	2	8	3	6
p	l	e	a	s	e	t	r
a	n	s	f	e	r	o	n
e	m	i	l	l	i	o	n
d	o	l	l	a	r	s	t
o	m	y	s	w	i	s	s
b	a	n	k	a	c	c	o
u	n	t	s	i	x	t	w
o	t	w	o	a	b	c	d

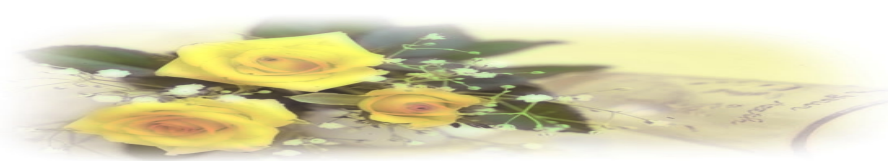
原文

Please transfer one
million dollars to my
Swiss Bank account six
two two ...

密文

AFLLSKSOSELAWAIA
TOOSSCTCLNMOMAN
ESIL YNTWRNNTSOW
FAEDOBNO ...

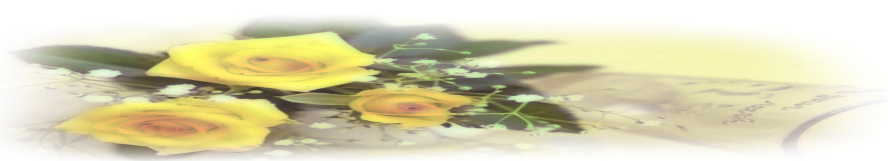
图9-2 按字符易位加密算法





2) 置换法

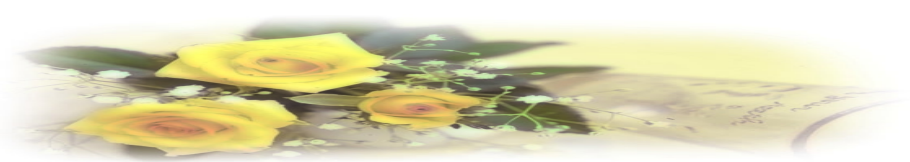
置换法是按照一定的规则，用一个字符去置换(替代)另一个字符来形成密文。最早由朱叶斯·凯撒(Julius Caesar)提出的算法非常简单，它是将字母a, b, c, ..., x, y, z循环右移三位后，形成d, e, f, ..., a, b, c字符序列，再利用移位后的序列中的字母去分别置换未移位序列中对应位置的字母，即利用d置换a，用e置换b等。凯撒算法的推广是移动K位。单纯移动K位的置换算法很容易被破译，比较好的置换算法是进行映像。例如，将26个英文字母映像到另外26个特定字母中，见图9-3所示。利用置换法可对attack进行加密，使其变为QZZQEA。





a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B	N	M

图9-3 26个字母的映像

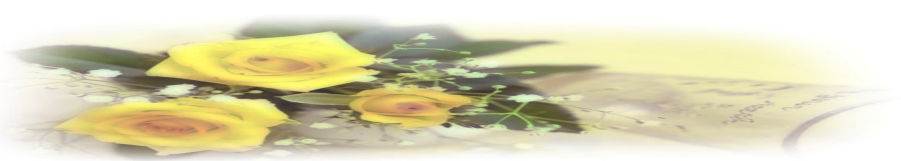




9.2.2 对称加密算法与非对称加密算法

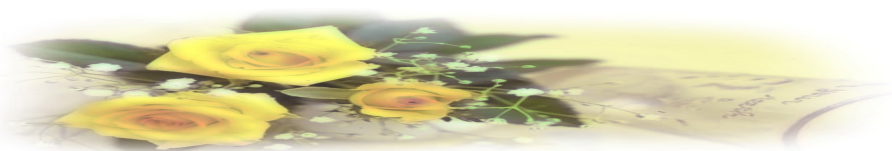
1. 对称加密算法

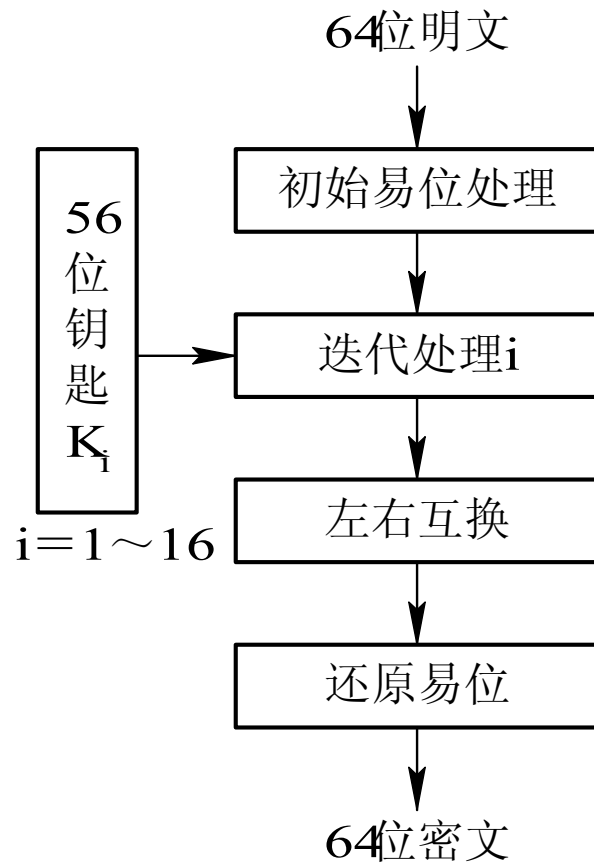
现代加密技术所用的基本手段，仍然是易位法和置换法，但它们与古典方法的重点不同。在古典法中通常采用的算法较简单，而密钥则较长；现代加密技术则采用十分复杂的算法，将易位法和置换法交替使用多次而形成乘积密码。最有代表性的对称加密算法是数据加密标准DES(Data Encryption Standard)。该算法原来是IBM公司于1971~1972年研制成功的，它旨在保护本公司的机密产品，后被美国国家标准局选为数据加密标准，并于1977年颁布使用。ISO现在已将DES作为数据加密标准。随着VLSI的发展，现在可利用VLSI芯片来实现DES算法，并用它做成数据加密处理器DEP。



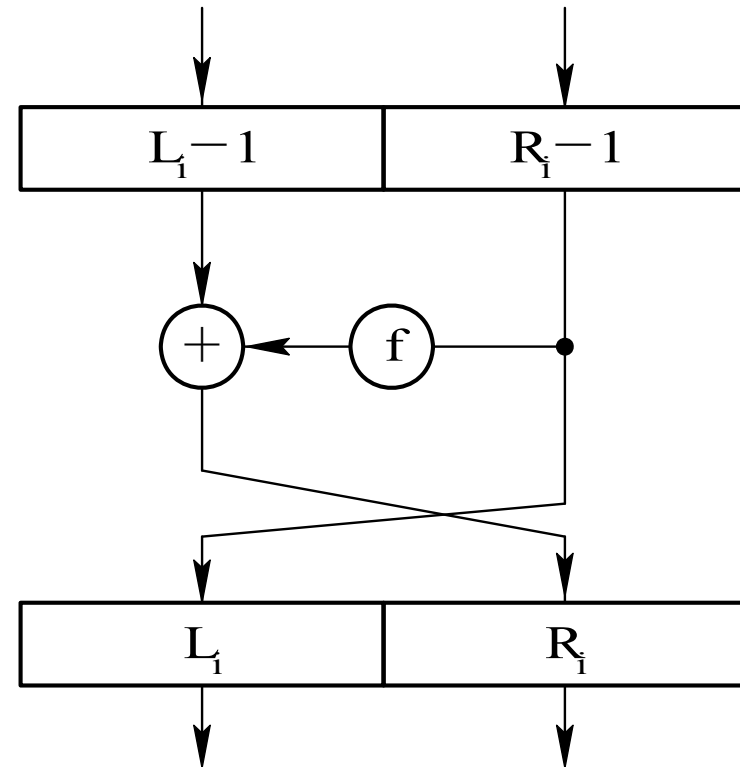


在DES中所使用的密钥长度为64位，它由两部分组成，一部分是实际密钥，占56位；另一部分是8位奇偶校验码。DES属于分组加密算法，它将明文按64位一组分成若干个明文组，每次利用56位密钥对64位的二进制明文数据进行加密，产生64位密文数据。DES算法的总框图如图9-4(a)所示。整个加密处理过程可分为四个阶段(共19步)，见图9-4(b)所示。



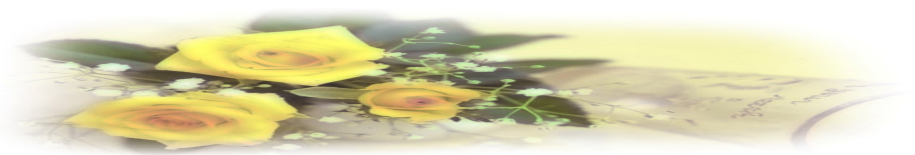


(a) DES算法总框图



(b) 迭代过程示意图

图9-4 DES加密标准



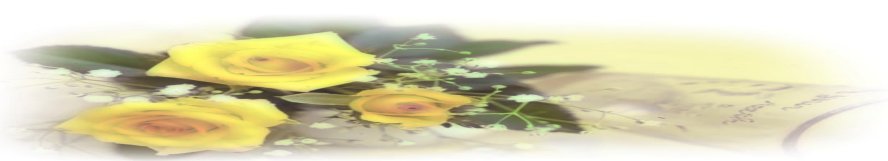


第一阶段：先将明文分出64位的明文段，然后对64位明文段做初始易位处理，得到 X_0 ，将其左移32位，记为 L_0 ，右移32位，记为 R_0 。

第二阶段：对初始易位结果 X_0 进行16次迭代处理(相应于第2~17步)，每一次使用56位加密密钥 K_i 。第2~17步的迭代过程如图9-4(b)所示。由图可以看出，输出的左32位 L_i 是输入右32位 R_{i-1} 的拷贝；而输出的右32位 R_i ，则是在密钥 K_i 的控制下，对输入右32位 R_{i-1} 做函数 f 的变换后的结果，再与输入左32位 L_{i-1} 进行异或运算而形成的，即

$$L_i = R_{i-1}$$

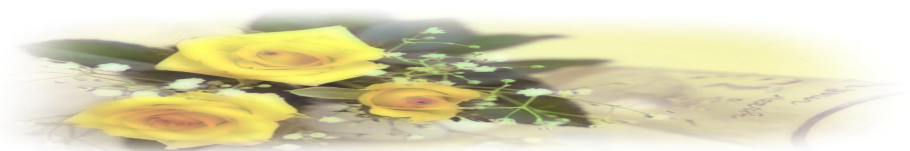
$$R_i = f(R_{i-1}, K_i) \oplus L_{i-1}$$





第三阶段：把经过16次迭代处理的结果(64位)的左32位与右32位互易位置。

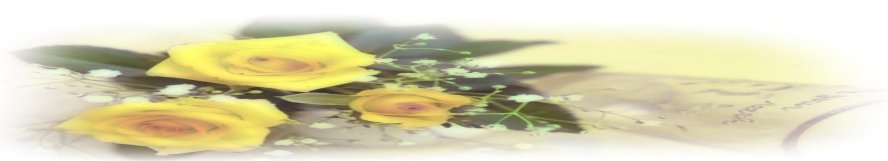
第四阶段：进行初始易位的逆变换。





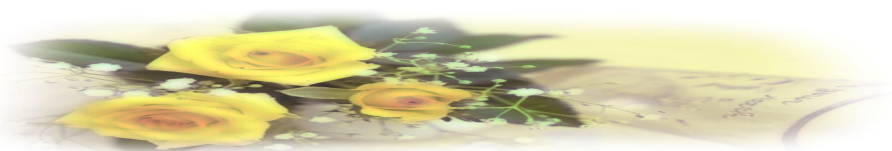
2. 非对称加密算法

DES加密算法属于对称加密算法。加密和解密所使用的密钥是相同的。DES的保密性主要取决于对密钥的保密程度。加密者必须用非常安全的方法(如通过个人信使)将密钥送给接收者(解密者)。如果通过计算机网络传送密钥,则必须先对密钥本身予以加密后再传送,通常把这种算法称为对称保密密钥算法。





1976年美国的Diffie和Hallman提出了一个新的非对称密码体制。其最主要的特点是：在对数据进行加密和解密时，使用不同的密钥。每个用户都保存着一对密钥，每个人的公开密钥都对外公开。假如某用户要与另一用户通信，他可用公开密钥对数据进行加密，而收信者则用自己的私用密钥进行解密。这样就可以保证信息不会外泄。





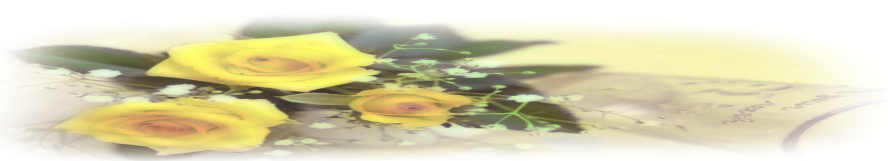
公开密钥算法的特点如下：

(1) 设加密算法为E、加密密钥为K_e，可利用它们对明文P进行加密，得到E_{K_e}(P)密文。设解密算法为D，解密密钥为K_d，可利用它们将密文恢复为明文，即

$$D_{K_d}(E_{K_e}(P)) = P$$

(2) 要保证从K_e推出K_d是极为困难的，或者说，从K_e推出K_d实际上是不可能的。

(3) 在计算机上很容易产生成对的K_e和K_d。

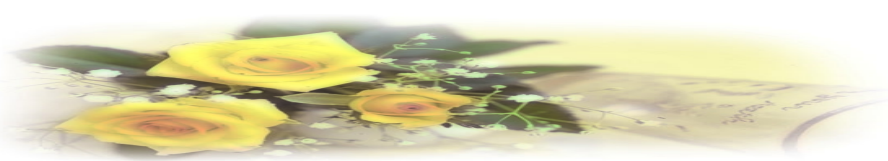




(4) 加密和解密运算可以对调，即利用 D_{K_d} 对明文进行加密形成密文，然后用 E_{K_e} 对密文进行解密，即

$$E_{K_e}(D_{K_d}(P)) = P$$

在此情况下，将解密密钥或加密密钥公开也无妨。因而这种加密方法称为公开密钥法(Public Key)。在公开密钥体制中，最著名的是RSA体制，它已被ISO推荐为公开密钥数据加密标准。



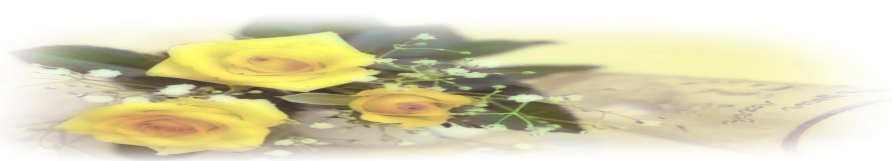


9.2.3 数字签名和数字证明书

1. 数字签名

在金融和商业等系统中，许多业务都要求在单据上加以签名或加盖印章，以证实其真实性，备日后查验。在利用计算机网络传送报文时，可将公开密钥法用于电子(数字)签名，来代替传统的签名。而为使数字签名能代替传统的签名，必须满足下述三个条件：

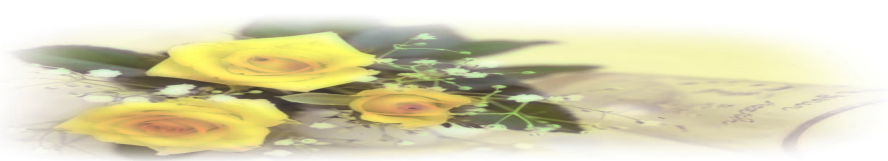
- (1) 接收者能够核实发送者对报文的签名。
- (2) 发送者事后不能抵赖其对报文的签名。
- (3) 接收者无法伪造对报文的签名。





1) 简单数字签名

在这种数字签名方式中，发送者A可使用私用密钥 K_{da} 对明文 P 进行加密，形成 $D_{K_{da}}(P)$ 后传送给接收者B。B可利用A的公开密钥 K_{ea} 对 $D_{K_{da}}(P)$ 进行解密，得到 $E_{K_{ea}}(D_{K_{da}}(P))=P$ ，如图9-5(a)所示。



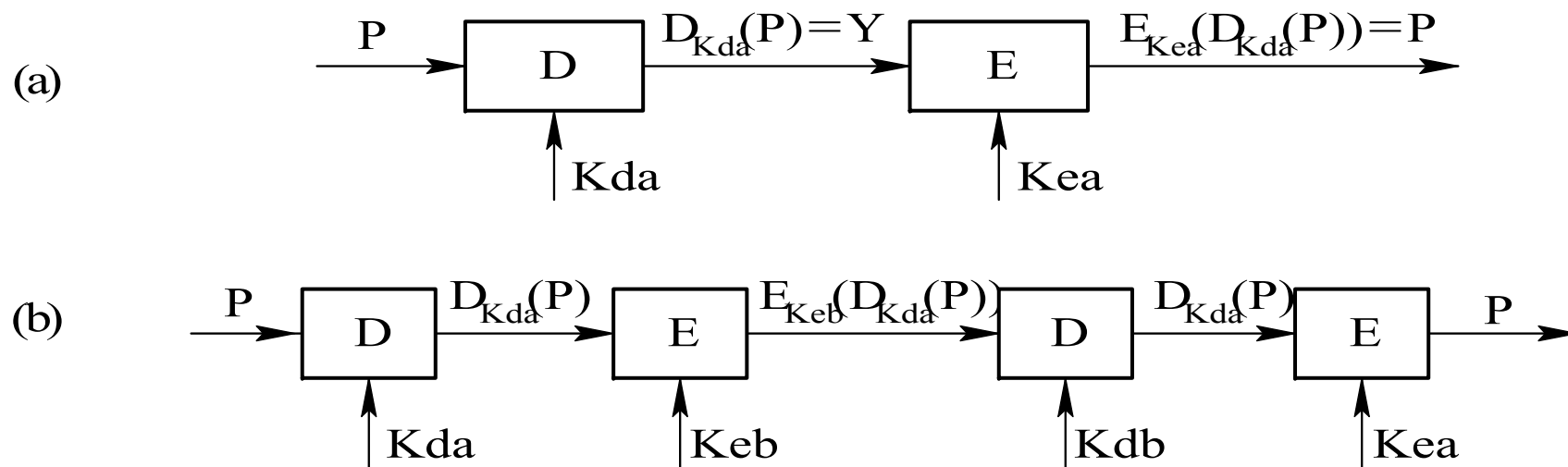
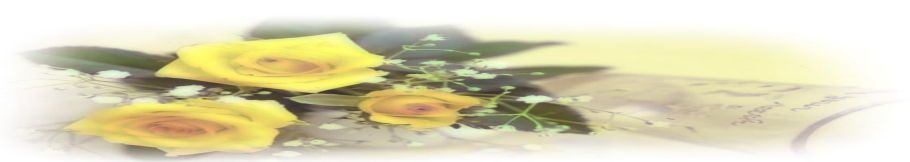


图9-5 数字签名示意图

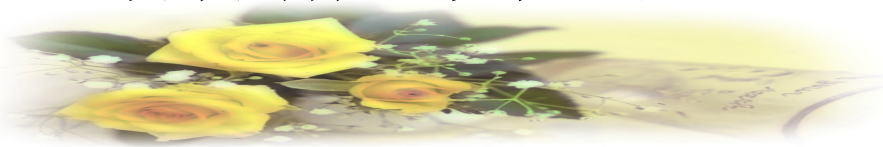




我们按照对数字签名的三点基本要求进行分析后可以得知：

- (1) 接收者能利用A的公开密钥 K_{ea} 对 $D_{K_{da}}(P)$ 进行解密，这便证实了发送者对报文的签名。
- (2) 由于只有发送者A才能发送出 $D_{K_{da}}(P)$ 密文，故不容A进行抵赖。
- (3) 由于B没有A所拥有的私用密钥，故B无法伪造对报文的签名。

由此可见，图9-5(a)所示的简单方法可以实现对传送的数据进行签名，但并不能达到保密的目的，因为任何人都能接收 $D_{K_{da}}(P)$ ，且可用A的公开密钥 K_{ea} 对 $D_{K_{da}}(P)$ 进行解密。为使A所传送的数据只能为B所接收，必须采用保密数字签名。





2) 保密数字签名

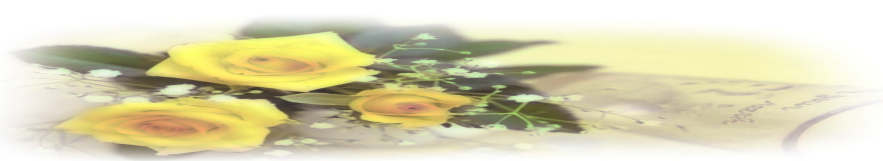
为了实现在发送者A和接收者B之间的保密数字签名，要求A和B都具有密钥，再按照图9-5(b)所示的方法进行加密和解密。

(1) 发送者A可用自己的私用密钥 K_{da} 对明文 P 加密，得到密文 $D_{K_{da}}(P)$ 。

(2) A再用B的公开密钥 K_{eb} 对 $D_{K_{da}}(P)$ 进行加密，得到 $E_{K_{eb}}(D_{K_{da}}(P))$ 后送B。

(3) B收到后，先用私用密钥 K_{db} 进行解密，即 $D_{K_{db}}(E_{K_{eb}}(D_{K_{da}}(P)))=D_{K_{da}}(P)$ 。

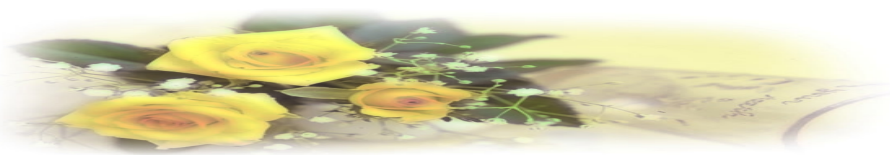
(4) B再用A的公开密钥 K_{ea} 对 $D_{K_{da}}(P)$ 进行解密，得到 $E_{K_{ea}}(D_{K_{da}}(P))=P$ 。





2. 数字证书(Certificate)

虽然可以利用公开密钥方法进行数字签名，但事实上又无法证明公开密钥的持有者是合法的持有者。为此，必须有一个大家都信得过的认证机构CA(Certification Authority)，由该机构为公开密钥发放一份公开密钥证明书，又把该公开密钥证明书称为数字证明书，用于证明通信请求者的身份。在网络上进行通信时，数字证明书的作用如同司机的驾驶执照、出国人员的护照、学生的学生证。在ITU制定的X.509标准中，规定了数字证明书的内容应包括：用户名称、发证机构名称、公开密钥、公开密钥的有效日期、证明书的编号以及发证者的签名。下面通过一个具体的例子来说明数字证明书的申请、发放和使用过程。

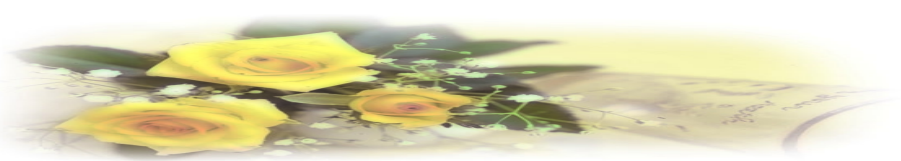




(1) 用户A在使用数字证书之前，应先向认证机构CA申请数字证书，此时A应提供身份证明和希望使用的公开密钥A。

(2) CA在收到用户A发来的申请报告后，若决定接受其申请，便发给A一份数字证书，在证书中包括公开密钥A和CA发证者的签名等信息，并对所有这些信思利用CA的私用密钥进行加密(即对CA进行数字签名)。

(3) 用户A在向用户B发送报文信息时，由A用私用密钥对报文加密(数字签名)，并连同已加密的数字证书一起发送给B。

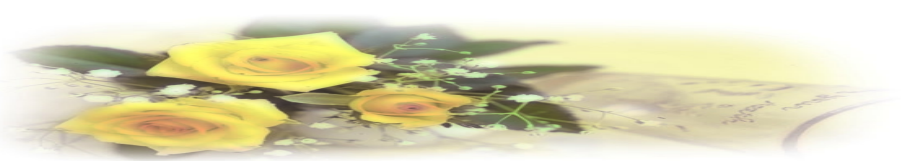




(4) 为了能对所收到的数字证明书进行解密，用户B须向CA机构申请获得CA的公开密钥B。CA收到用户B的申请后，可决定将公开密钥B发送给用户B。

(5) 用户B利用CA的公开密钥B对数字证明书加以解密，以确认该数字证明书确系原件，并从数字证明书中获得公开密钥A，并且也确认该公开密钥A确系用户A的密钥。

(6) 用户B再利用公开密钥A对用户A发来的加密报文进行解密，得到用户A发来的报文的真实明文。

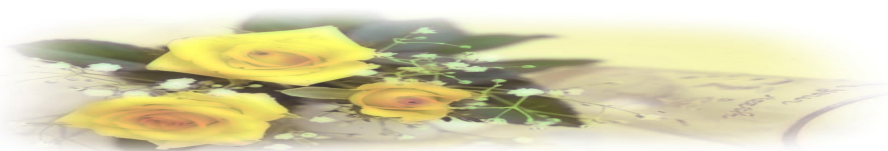




9.2.4 网络加密技术

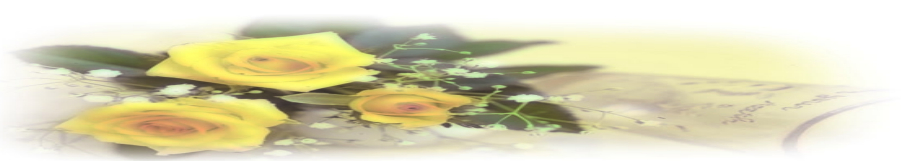
1. 链路加密(Link Encryption)

链路加密，是对在网络相邻结点之间通信线路上传输的数据进行加密。链路加密常采用序列加密算法，它能有效地防止由搭线窃听所造成的威胁。两个数据加密设备分别置于通信线路的两端，它们使用相同的数据加密密钥。





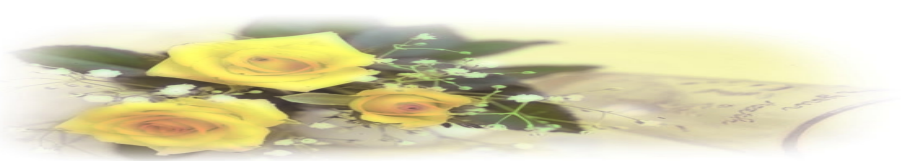
如果在网络中只采用了链路加密，而未使用端—端加密，那么，报文从最高层(应用层)到数据链路层之间，都是以明文的形式出现的，只是从数据链路层进入物理层时，才对报文进行了加密，并把加密后的数据通过传输线路传送到对方结点上。为了防止攻击者对网络中的信息流进行分析，在链路加密方式中，不仅对正文做了加密，而且对所有各层的控制信息也进行了加密。





接收结点在收到加密报文后，为了能对报文进行转发，必须知道报文的目标地址。为此，接收结点上的数据加密设备应对所接收到的加密报文进行解密，从中找出目标地址并进行转发。当该报文从数据链路层送入物理层(转发)时，须再次对报文进行加密。

由上所述得知，在链路加密方式中，在相邻结点间的物理信道上传输的报文是密文，而在所有中间结点中的报文则是明文，这给攻击者造成了可乘之机，使其可从中间结点对传输中的信息进行攻击。这就要求能对所有各中间结点进行有效的保护。



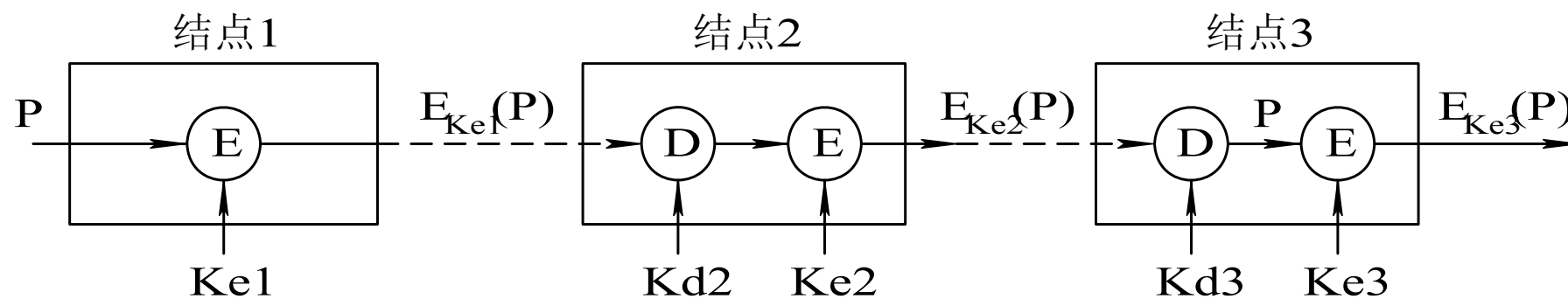
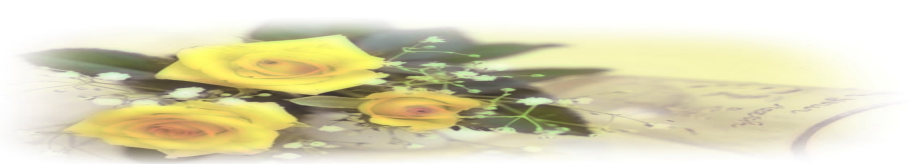


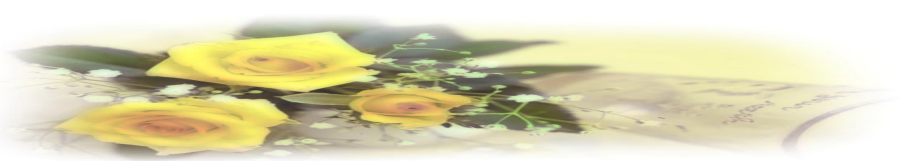
图9-6 链路加密方式





2. 端—端加密(End to End Encryption)

在单纯采用链路加密方式时，所传送的数据在中间结点中将被恢复为明文，因此，链路加密方式尚不能保证通信的安全性；而端—端加密方式是在源主机或前端机FEP中的高层(从传输层到应用层)对所传输的数据进行加密。在整个网络的传输过程中，不论是在物理信道上，还是在中间结点中，报文的正文始终是密文，直至信息到达目标主机后才被译成明文，因而这样可以保证在中间结点不会出现明文。图9-7 示出了端—端加密的情况。



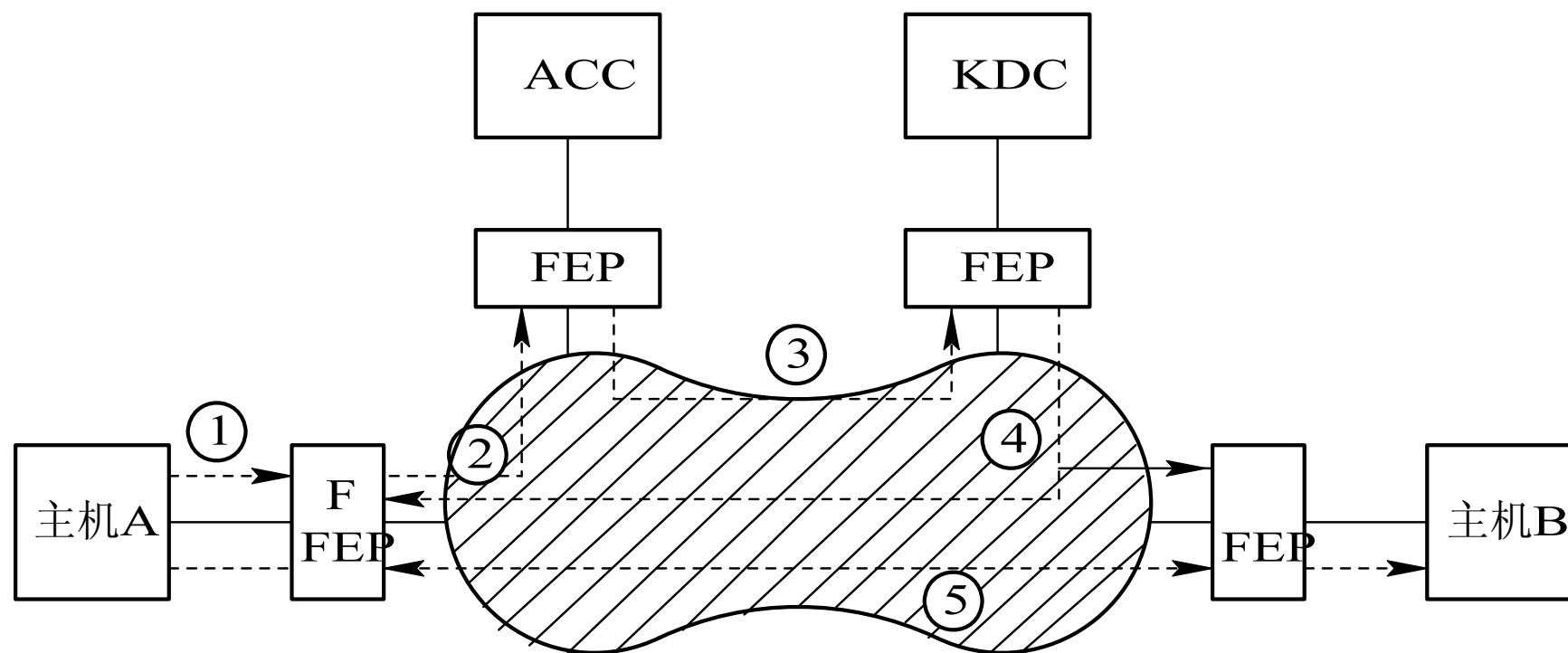
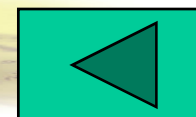


图9-7 端—端加密方式



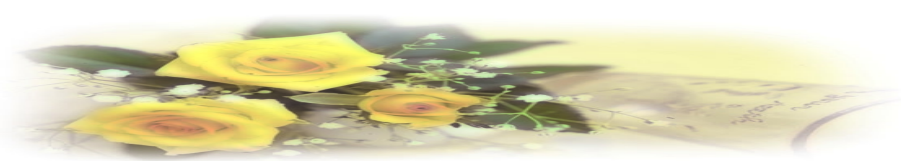


9.3 认证技术

9.3.1 基于口令的身份认证

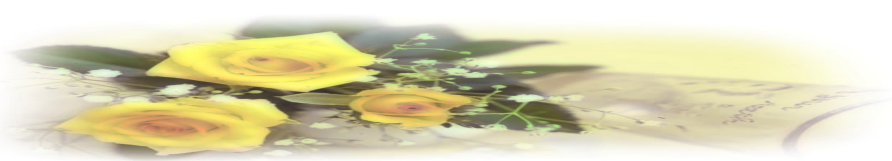
1. 口令

当一个用户要登录某台计算机时，操作系统通常都要认证用户的身份。而利用口令来确认用户的身份是当前最常用的认证技术。



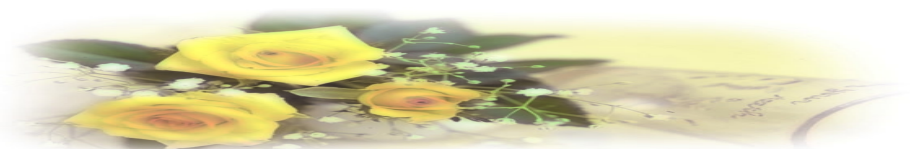


通常，每当用户要上机时，系统中的登录程序都首先要求用户输入用户名，登录程序利用用户输入的名字去查找一张用户注册表或口令文件。在该表中，每个已注册用户都有一个表目，其中记录有用户名和口令等。登录程序从中找到匹配的用户名后，再要求用户输入口令，如果用户输入的口令也与注册表中用户所设置的口令一致，系统便认为该用户是合法用户，于是允许该用户进入系统；否则将拒绝该用户登录。





口令是由字母或数字、或字母和数字混合组成的，它可由系统产生，也可由用户自己选定。系统所产生的口令不便于用户记忆，而用户自己规定的口令则通常是很容易记忆的字母、数字，例如生日、住址、电话号码，以及某人或宠物的名字等。这种口令虽便于记忆，但也很容易被攻击者猜中。

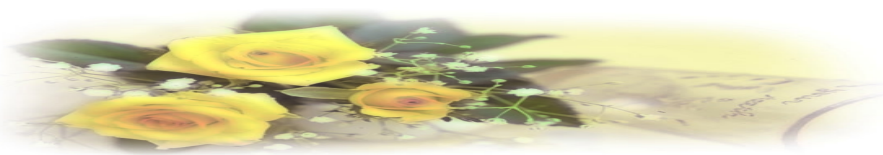




2. 对口令机制的基本要求

1) 口令长度适中

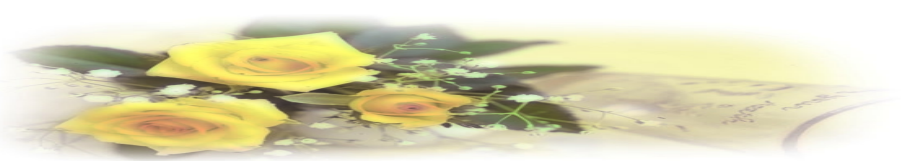
通常的口令是由一串字母和数字组成。如果口令太短，则很容易被攻击者猜中。例如，一个由四位十进制数所组成的口令，其搜索空间仅为 10^4 ，在利用一个专门的程序来破解时，平均只需5000次即可猜中口令。假如每猜一次口令需花费0.1 ms的时间，则平均每猜中一个口令仅需0.5 s。而如果采用较长的口令，假如口令由ASCII码组成，则可以显著地增加猜中一个口令的时间。例如，口令由7位ASCII码组成，其搜索空间变为 95^7 (95是可打印的ASCII码)，大约是 7×10^{13} ，此时要猜中口令平均需要几十年。因此建议口令长度不少于7个字符，而且在口令中应包含大写和小写字母及数字，最好还能引入特殊符号。





2) 自动断开连接

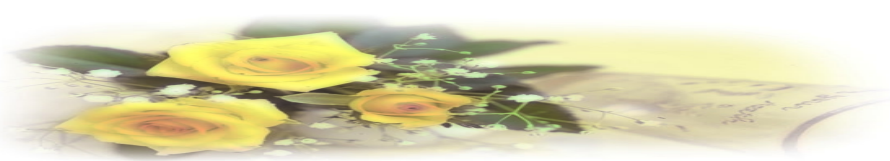
为了给攻击者猜中口令增加难度，在口令机制中还应引入自动断开连接的功能，即只允许用户输入有限次数的不正确口令，通常规定3~5次。如果用户输入不正确口令的次数超过了规定的次数，系统便自动断开该用户所在终端的连接。当然，此时用户还可能重新拨号请求登录，但若在重新输入指定次数的不正确口令后仍未猜中，系统会再次断开连接。这种自动断开连接的功能，无疑又会给攻击者增加猜中口令的难度，从而会增加猜中口令所需的时间。





3) 隐蔽回送显示

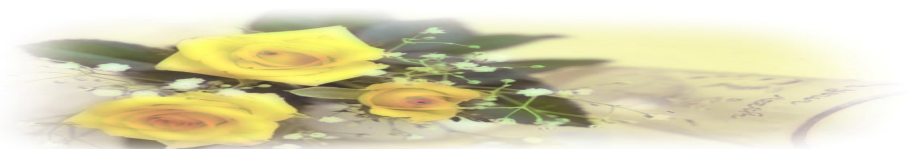
在用户输入口令时，登录程序不应将该口令回送到屏幕上显示，以防止被就近的人发现。在Windows 2000系统中，将每一个输入字符显示为星号；在UNIX系统中，在输入口令时没有任何显示。从保密角度看，攻击者可从Windows 2000的回送显示了解到口令的长度，而在UNIX中则什么也看不出。还有另一种情况值得注意，在有的系统中，只要看到非法登录名就禁止登录，这样攻击者就知道登录名是错误的。而有的系统，即使看到非法登录名后也不作任何表示，仍要求其输入口令，等输完口令才显示禁止登录信息。这样攻击者只是知道登录名和口令的组合是错误的。





4) 记录和报告

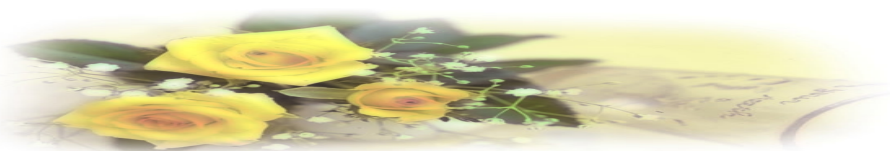
该功能用于记录所有用户登录进入系统和退出系统的时间；也用来记录和报告攻击者非法猜测口令的企图，以及所发生的与安全性有关的其它不轨行为，这样便能及时发现有人在对系统的安全性进行攻击。





3. 一次性口令(One Time Password)

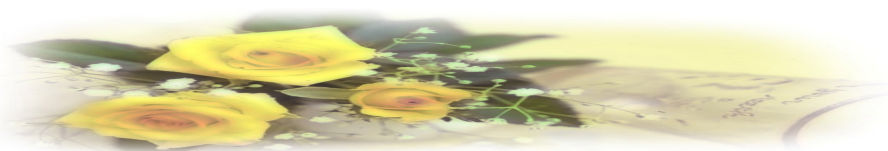
为了把由于口令泄露所造成的损失减到最小，用户应当经常改变口令。例如，一个月改变一次，或者一个星期改变一次。一种极端的情况是采用一次性口令机制，即口令被使用一次后，就换另一个口令。在采用该机制时，用户必须提供记录有一系列口令的一张表，并将该表保存在系统中。系统为该表设置一指针用于指示下次用户登录时所应使用的口令。这样，用户在每次登录时，登录程序便将用户输入的口令与该指针所指示的口令相比较，若相同，便允许用户进入系统，并将指针指向表中的下一个口令。在采用一次性口令的机制时，即使攻击者获得了本次用户上机时所使用的口令，他也无法进入系统。必须注意，用户所使用的口令表要妥善保存好。





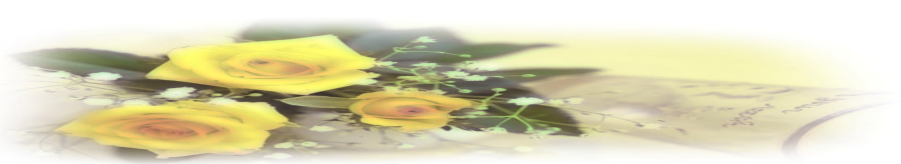
4. 口令文件

通常在口令机制中，都配置有一份口令文件，用于保存合法用户的口令和与口令相联系的特权。该文件的安全性至关重要，一旦攻击者成功地访问了该文件，攻击者便可随心所欲地访问他感兴趣的所有资源，这对整个计算机系统的资源和网络将无安全性可言。显然，如何保证口令文件的安全性，已成为系统安全性的头等重要问题。





保证口令文件安全性的最有效的方法是，利用加密技术，其中一个行之有效的方法是选择一个函数来对口令进行加密，该函数 $f(x)$ 具有这样的特性：在给定了 x 值后，很容易算出 $f(x)$ ；然而，如果给定了 $f(x)$ 的值，却不能算出 x 的值。利用 $f(x)$ 函数去编码(即加密)所有的口令，再将加密后的口令存入口令文件中。当某用户输入一个口令时，系统利用函数 $f(x)$ 对该口令进行编码，然后将编码(加密)后的口令与存储在口令文件中的已编码的口令进行比较，如果两者相匹配，便认为是合法用户。顺便说明一下，即使攻击者能获取口令文件中的已编码口令，他也无法对它们进行译码，因而不会影响到系统的安全性。图9-8示出了一种对加密口令进行验证的方法。



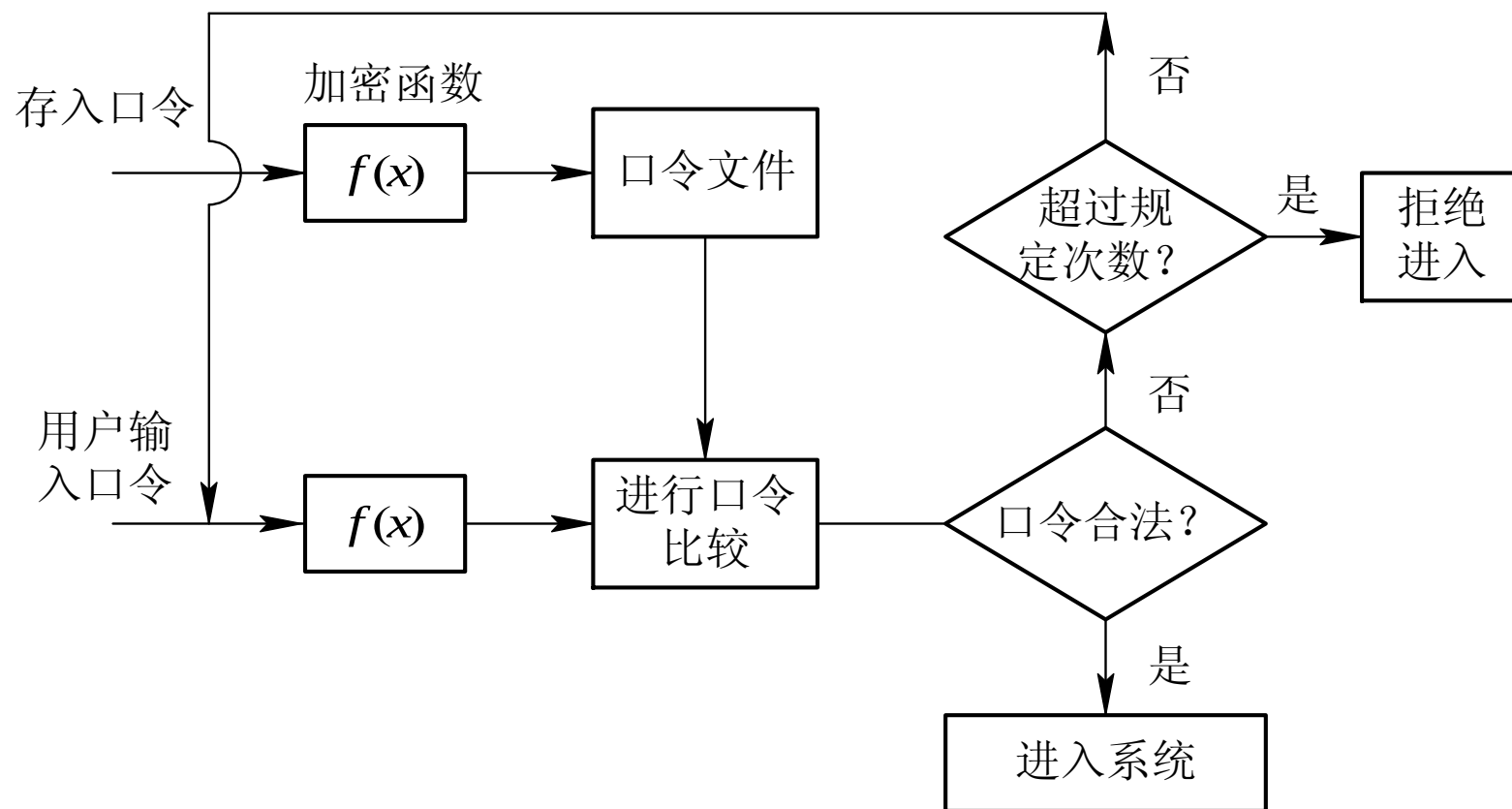


图9-8 对加密口令的认证方法

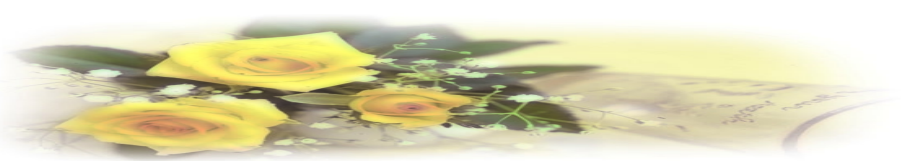


尽管对口令进行加密是一个很好的方法，但它也不是绝对的安全可靠。其主要威胁来自于两个方面：

(1) 当攻击者已掌握了口令的解密密钥时，就可用它来破译口令。

(2) 利用加密程序来破译口令，如果运行加密程序的计算机速度足够快，则通常只要几个小时便可破译口令。

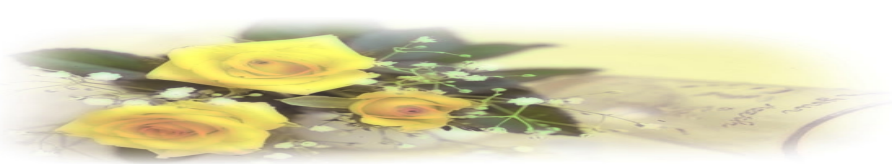
因此，人们还是应该妥善保管好已加密的口令文件，来防止攻击者轻易地获取该文件。





5. 挑战-响应验证

在该方法中，由用户自己选择一个算法，算法可以很简单，也可以比较复杂，如X2运算。该算法也需告知服务器。每当用户登录时，服务器就给用户发来一个随机数，如12，用户收到后便对该数据进行平方运算得到144，用户就用它作为口令。服务器将所收到的口令与自己计算(利用X2算法)的结果进行比较，如相同，则允许用户上机，否则，拒绝用户登录。由于该方法所使用的口令不是一个固定数据，而是基于服务器随机产生的数据，再经过计算得到的，因此攻击者难于猜测。如果再频繁地改变算法，就会令攻击者更加难于猜测。

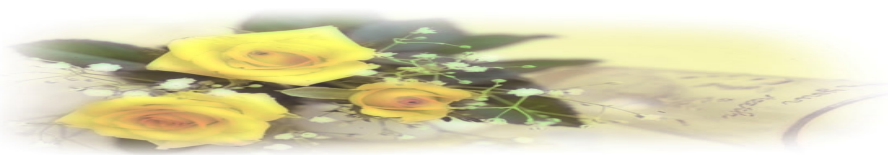




9.3.2 基于物理标志的认证技术

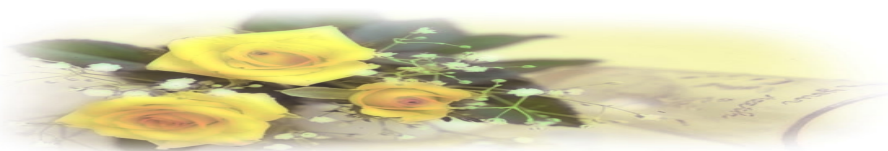
1. 基于磁卡的认证技术

根据数据记录原理，可将当前使用的卡分为磁卡和IC卡两种。磁卡是基于磁性原理来记录数据的，目前世界各国使用的信用卡和银行现金卡等，都普遍采用磁卡。这是一块其大小和名片大小相仿的塑料卡，在其上贴有含若干条磁道的磁条。一般在磁条上有三条磁道，每条磁道都可用来记录不同标准和不同数量的数据。磁道上可有两种记录密度，一种是每英寸含有15 bit信息的低密度磁道；另一种是每英寸含有210 bit信息的高密度磁道。如果在磁条上记录了用户名、用户密码、账号和金额，这就是金融卡或银行卡；而如果在磁条上记录的是有关用户的信息，则该卡便可作为识别用户身份的物理标志。





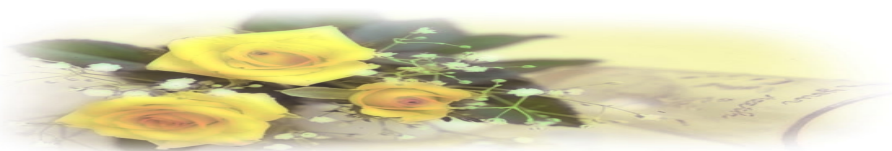
在磁卡上所存储的信息，可利用磁卡读写器将之读出：只要将磁卡插入或划过磁卡读写器，便可将存储在磁卡中的数据读出，并传送到相应的计算机中。用户识别程序便利用读出的信息去查找一张用户信息表(该表中包含有若干个表目，每个用户占有一个表目，表目中记录了有关该用户的信息)。若找到匹配的表目，便认为该用户是合法用户；否则便认为是非法用户。为了保证持卡者是该卡的主人，通常在基于磁卡认证技术的基础上，又增设了口令机制，每当进行用户身份认证时，都先要求用户输入口令。





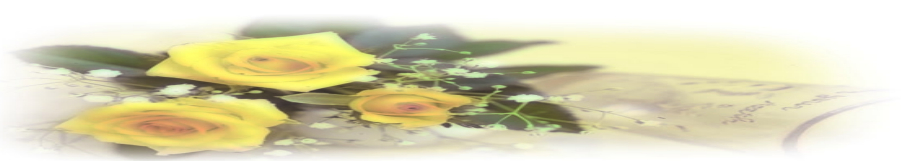
2. 基于IC卡的认证技术

IC卡即集成电路卡的英文缩写。在外观上IC卡与磁卡并无明显差异，但在IC卡中可装入CPU和存储器芯片，使该卡具有一定的智能，故又称为智能卡或灵巧卡。IC卡中的CPU用于对内部数据的访问和与外部数据进行交换，还可利用较复杂的加密算法，对数据进行处理，这使IC卡比磁卡具有更强的防伪性和保密性，因而IC卡会逐步取代磁卡。根据在磁卡中所装入芯片的不同可把IC卡分为以下三种类型：



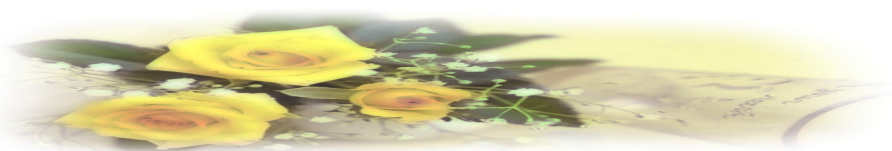


(1) 存储器卡。在这种卡中只有一个E2PROM(可电擦、可编程只读存储器)芯片，而没有微处理器芯片。它的智能主要依赖于终端，就像IC电话卡的功能是依赖于电话机一样。由于此智能卡不具有安全功能，故只能作为储值卡，用来存储少量金额的现金与信息。常见的这类智能卡有电话卡、健康卡，其只读存储器的容量一般为4~20 KB。



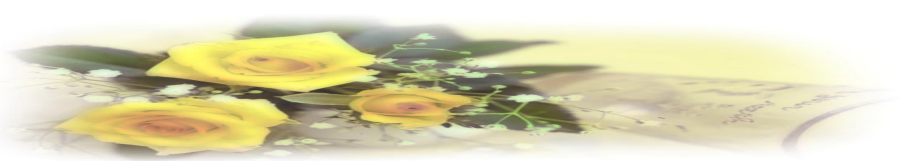


(2) 微处理器卡。它除具有E2PROM外，还增加了一个微处理器。只读存储器的容量一般是数千字节至数万字节；处理器的字长多为8位。在这种智能卡中已具有一定的加密设施，增强了IC卡的安全性，因此有着更为广泛的用途，被广泛用作信用卡。用户可以在商场把信用卡插入读卡机后，授权进行一定数额的转账，信用卡将一段加密后的信息发送到商场，商场再将该信息转发到银行，从用户在该银行中的账户中扣除所需付出的金额。





(3) 密码卡。在这种卡中又增加了加密运算协处理器和RAM。之所以把这种卡称为密码卡，是由于它能支持非对称加密体制RSA；所支持的密钥长度可长达1024位，因而极大地增强了IC卡的安全性。一种专门用于确保安全的智能卡，在卡中存储了一个很长的用户专门密钥和数字证明书，完全可以作为用户的数字身份证明。当前在Internet上所开展的电子交易中，已有不少密码卡使用了基于RSA的密码体制。



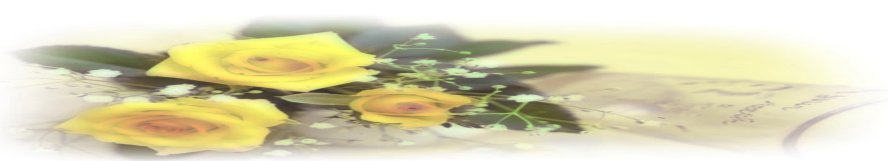


9.3.3 基于生物标志的认证技术

1. 常用于身份识别的生理标志

被选用的生理标志应具有这样三个条件：

- (1) 足够的可变性，系统可根据它来区别成千上万的不同用户；
- (2) 被选用的生理标志应保持稳定，不会经常发生变化；
- (3) 不易被伪装。

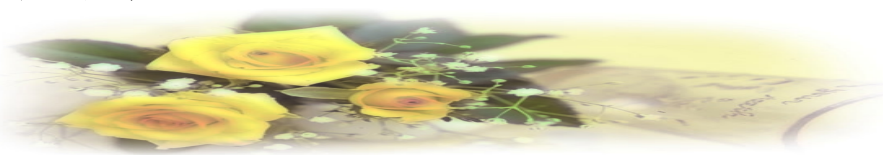




1) 指纹

指纹有着“物证之首”的美誉。尽管目前全球已有近60亿人口，但绝对不可能找到两个完全相同的指纹，而且它的形状不会随时间而改变，因而利用指纹来进行身份认证是万无一失的。又因为它不会像其它一些物理标志那样出现用户忘记携带或丢失等问题，而且使用起来也特别方便，因此，指纹验证很早就用于契约签证和侦查破案，既准确又可靠。

人的手指的纹路可分为两大类，一类是环状，另一类是涡状，每一类又可进一步分为50~200种不同的图样。以前是依靠专家进行指纹鉴别，随着计算机技术的发展，人们已成功地开发出指纹自动识别系统。利用指纹来进行身份识别是有广阔前景的一种识别技术，世界上已有愈来愈多的国家开展了对指纹识别技术的研究和应用。





2) 视网膜组织

视网膜组织通常又简称为眼纹。它与指纹一样，世界上也绝对不可能找到两个人有完全相同的视网膜组织，因而利用视网膜组织来进行身份认证同样是非常可靠的。用户的视网膜组织所含的信息量远比指纹复杂，其信息需要用256个字节来编码。利用视网膜组织进行身份验证的效果非常好，如果注册人数不超过200万，其出错率为0，所需时间也仅为秒级，现已在军事部门和银行系统中采用，目前成本还比较高。

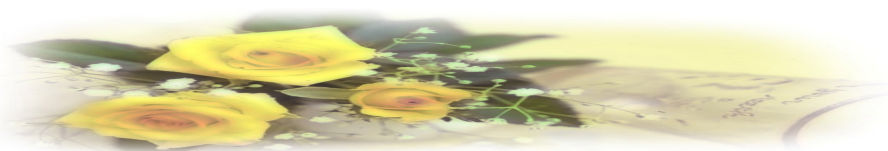
但是这种身份验证方式也还存着抗欺骗能力问题。在早期的系统中，用户的视网膜组织是由一米外的照相机对人眼进行拍摄来认证的，如果有人戴上墨镜，在墨镜上贴上别人的视网膜，这样便可以蒙混过关。但如果改用摄像机，它可以拍下视网膜的震动影像，就不易被假冒。





3) 声音

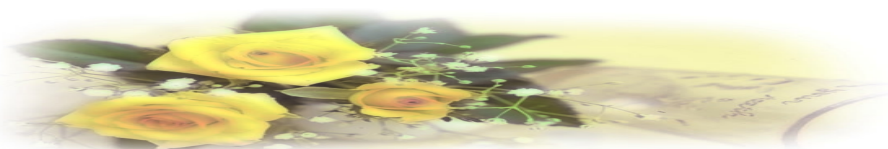
每个人在说话时都会发出不同的声音，人对语音非常敏感，即使在强干扰的环境下，也能很好地分辨出每个人的语音。事实上，人们主要依据听对方的声音来确定对方的身份。现在又广泛采用与计算机技术相结合的办法来实现身份验证，其基本方法是，对一个人说话的录音进行分析，将其全部特征存储起来，通常把所存储的语音特征称为语声纹。然后再利用这些声纹制作成语音口令系统。该系统的出错率在1%~1‰，制作成本较低，为数百到数千美元。





4) 手指长度

由于每个人的五个手指的长度并不是完全相同的，因此可基于它来识别每一个用户。可通过把手插入一个手指长度测量设备，测出五个手指的长度，与数据库中所保存的相应样本进行核对。这种方式比较容易遭受欺骗，例如可利用手指石膏模型或其它仿制品来进行欺骗。



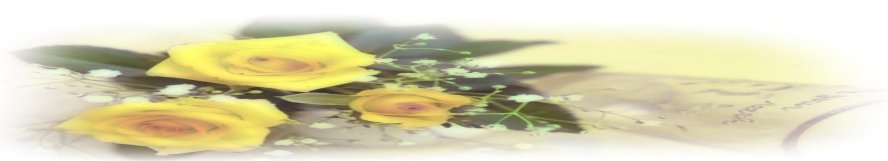


2. 生物识别系统的组成

1) 对生物识别系统的要求

生物识别系统是一个相当复杂的系统，要设计出一个非常实用的生物识别系统并非易事，必须满足如下三方面的要求。

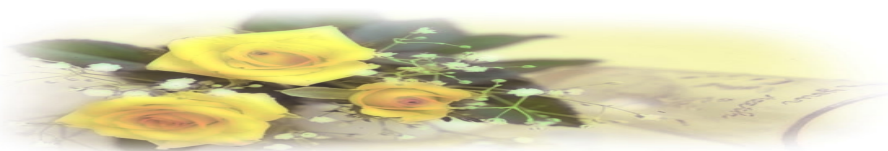
(1) 识别系统的性能必须满足需求。这包括应具有很强的抗欺骗和防伪造能力，而且还应能防范攻击者设置陷阱。





(2) 能被用户接受。完成一次识别的时间不应太长，应不超过1~2 s；出错率应足够低，这随应用场合的不同而异。对于用在极为重要场合中的识别系统，将会要求绝对不能出错，可靠性和可维护性也要好。

(3) 系统成本适当。系统成本包含系统本身的成本、运营期间所需的费用和系统维护(包含消耗性材料等)的费用。

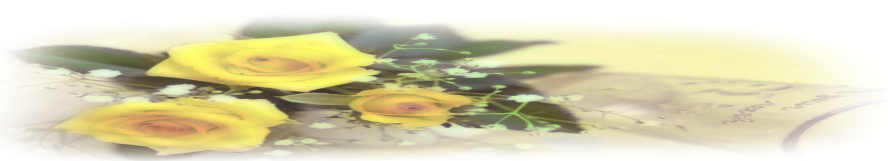




2) 生物识别系统的组成

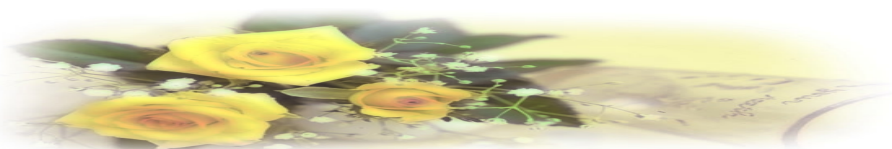
生物识别系统通常是由注册和识别两部分组成的。

(1) 注册部分。在该系统中，配置有一张注册表，每个注册用户在表中都有一个记录。记录中至少有两项，其中一项用于存放用户姓名，另一项用于存放用户的重要特征(用户的生物特征被数字化后形成用户样本，再从中提取出重要特征)。该记录通常存放在中心数据库中，供多个生物识别系统共享，但也可放在用户的身份智能卡中。





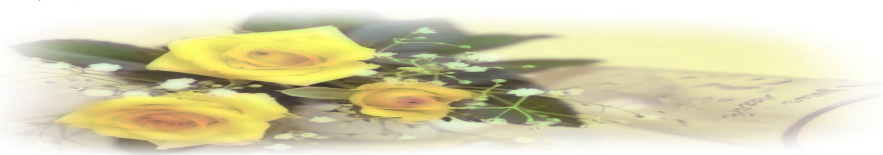
(2) 识别部分。它可分为两步，第一步是要求用户输入用户登录名，这样可使系统尽快找到该用户在系统中的记录；第二步是对用户输入的生物特征进行识别，即把用户的生物特征与用户记录中的样本信息特征进行比较，若相同，便允许用户登录，否则，拒绝用户登录。





3. 指纹识别系统

从20世纪70年代开始，美国及其它发达国家便开始研究利用计算机进行指纹自动识别，并取得了很大的进展，到80年代指纹自动识别系统已在许多国家使用。在所构成的指纹识别系统中包括指纹输入、指纹图像压缩、指纹自动比较等8个子系统。但他们的指纹识别系统是建立在中、小型计算机系统的基础上的，每一个新用户注册大约需要四分钟，记录下一个人的两个手指图样的时间约为2分钟，每次识别的时间不超过5秒钟，出错率小于千分之一。由于系统比较庞大，价格也昂贵，每套设备的售价约为10 000美元，因此使该技术难于普及。直至20世纪90年代中期，随着VLSI的迅速发展，才使指纹识别系统小型化并进入了广泛应用的阶段。



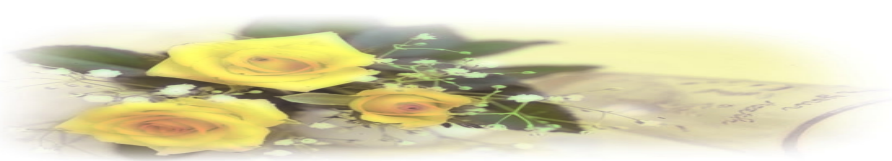


9.3.4 基于公开密钥的认证技术

1. 申请数字证书

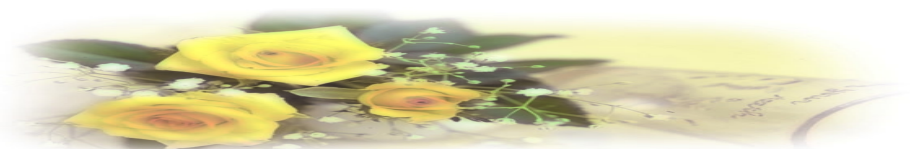
由于SSL所提供的安全服务是基于公开密钥证明书(数字证书)的身份认证,因此,凡是要利用SSL的用户和服务端,都必须先向认证机构(CA)申请公开密钥证明书。

(1) 服务器申请数字证书。首先由服务管理器生成一密钥对和申请书。服务器一方面将密钥和申请书的备份保存在安全之处;另一方面则向CA提交包括密钥对和签名证明书申请(即CSR)的加密文件,通常以电子邮件方式发送。CA接收并检查该申请的合法性后,将会把数字证书以电子邮件方式寄给服务器。





(2) 客户申请数字证书。首先由浏览器生成一密钥对，私有密钥被保存在客户的私有密钥数据库中，将公开密钥连同客户提供的其它信息一起发往CA。如果该客户符合CA要求的条件，CA将会把数字证书以电子邮件方式寄给客户。

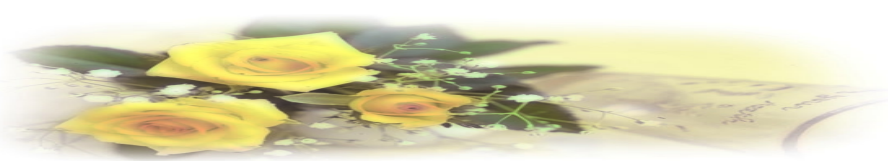




2. SSL握手协议

客户和服务端在进行通信之前，必须先运行SSL握手协议，以完成身份认证、协商密码算法和加密密钥。

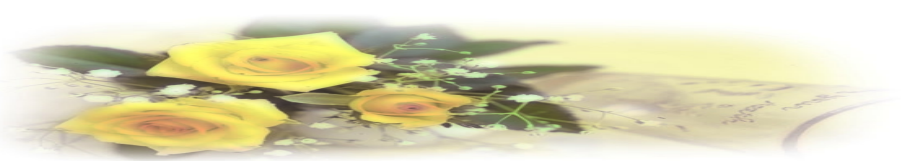
(1) 身份认证。SSL协议要求通信的双方都利用自己的私用密钥对所要交换的数据进行数字签名，并连同数字证书一起发送给对方，以便双方相互检验。如上节所述，通过数字签名和数字证书的验证可以认证对方的身份是否真实。





(2) 协商加密算法。为了增加加密系统的灵活性，SSL协议允许采用多种加密算法。客户和服务端在通信之前，应首先协商好所使用的某一种加密算法。通常先由客户提供自己能实现的所有加密算法清单，然后由服务器从中选择出一种最有效的加密算法，并通知客户，此后，双方便可利用该算法对所传送的信息进行加密。

(3) 协商加密密钥。先由客户机随机地产生一组密钥，再利用服务器的公开密钥对这组密钥进行加密后，送往服务器，由服务器从中选择四个密钥，并通知客户机，将之用于对所传输的信息进行加密。

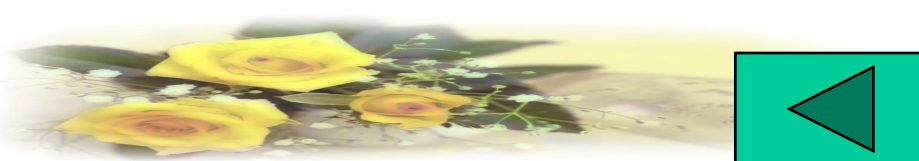




3. 数据加密和检查数据的完整性

(1) 数据加密。在客户机和服务器间传送的所有信息，都应利用协商后所确定的加密算法和密钥进行加密处理，以防止被攻击。

(2) 检查数据的完整性。为了保证经过长途传输后所收到的数据是可信任的，SSL协议还利用某种算法对所传送的数据进行计算，以产生能保证数据完整性的数据识别码(MAC)，再把MAC和业务数据一起传送给对方；而收方则利用MAC来检查所收到数据的完整性。





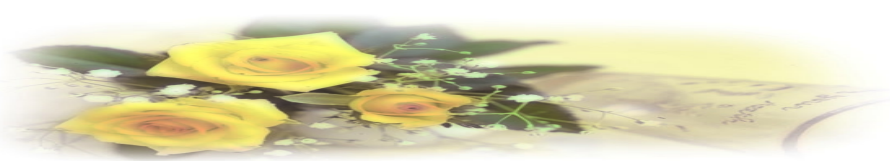
9.4 访问控制技术

9.4.1 访问矩阵

1. 保护域(Protection Domain)

1) 访问权

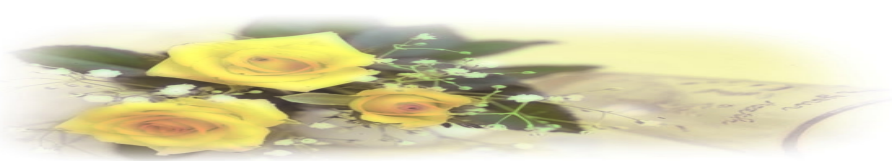
为了对系统中的对象加以保护，应由系统来控制进程对对象的访问。我们把一个进程能对某对象执行操作的权力称为访问权(Access Right)。每个访问权可以用一个有序对(对象名，权集)来表示，例如，某进程有对文件 F_1 执行读和写操作的权力，这时，可将该进程的访问权表示成 $(F_1, \{R/W\})$ 。





2) 保护域

为了对系统中的资源进行保护而引入了保护域的概念，保护域简称为“域”。“域”是进程对一组对象访问权的集合，进程只能在指定域内执行操作，这样，“域”也就规定了进程所能访问的对象和能执行的操作。在图9-9中示出了三个保护域。在域1中有两个对象，即文件 F_1 和 F_2 ，只允许进程对 F_1 读，而允许对 F_2 读和写；而对象Printer 1同时出现在域2和域3中，这表示在这两个域中运行的进程都能使用打印机。



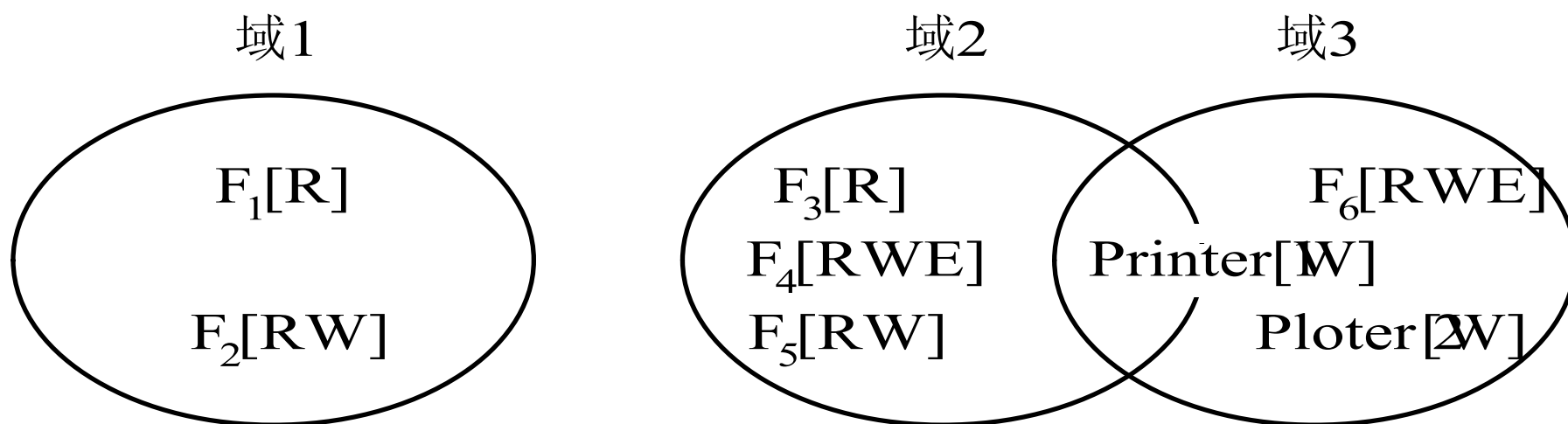
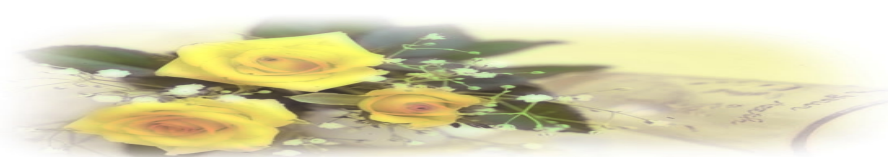


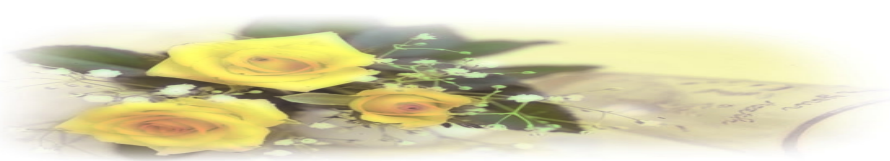
图9-9 三个保护域





3) 进程和域间的静态联系方式

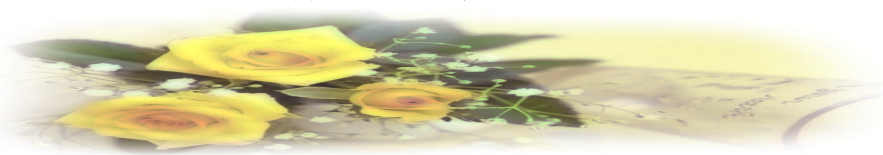
在进程和域之间，可以一一对应，即一个进程只联系着一个域。这意味着，在进程的整个生命期中，其可用资源是固定的，我们把这种域称为“静态域”。在这种情况下，进程运行的全过程都是受限于同一个域，这将会使赋予进程的访问权超过了实际需要。例如，某进程在运行开始时需要磁带机输入数据；而在进程快结束时，又需要用打印机打印数据。在一个进程只联系着一个域的情况下，则需要在该域中同时设置磁带机和打印机这两个对象，这将超过进程运行的实际需要。





4) 进程和域间的动态联系方式

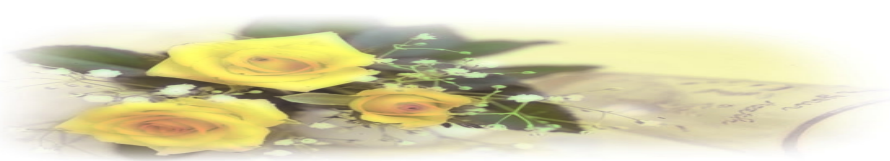
在进程和域之间，也可以是一对多的关系，即一个进程可以联系着多个域。在此情况下，可将进程的运行分为若干个阶段，其每个阶段联系着一个域，这样便可根据运行的实际需要，来规定在进程运行的每个阶段中所能访问的对象。用上述的同一个例子，我们可以把进程的运行分成三个阶段：进程在开始运行的阶段联系着域 D_1 ，其中包括用磁带机输入；在运行快结束的第三阶段联系着域 D_3 ，其中是用打印机输出；中间运行阶段联系着域 D_2 ，其中既不含磁带机，也不含打印机。我们把这种一对多的联系方式称为动态联系方式，在采用这种方式的系统中，应增设保护域切换功能，以使进程能在不同的运行阶段，从一个保护域切换到另一个保护域。





2. 访问矩阵

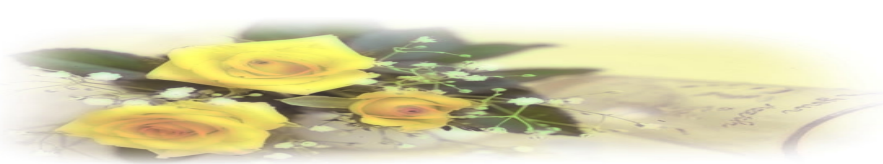
我们可以利用一个矩阵来描述系统的访问控制，并把该矩阵称为访问矩阵(Access Matrix)。访问矩阵中的行代表域，列代表对象，矩阵中的每一项是由一组访问权组成的。因为对象已由列显式地定义，故可以只写出访问权而不必写出是对哪个对象的访问权，每一项访问权 $\text{access}(i, j)$ 定义了域 D_i 中执行的进程能对对象 Q_j 所施加的操作集。





访问矩阵中的访问权，通常是由资源的拥有者或者管理者所决定的。当用户创建一个新文件时，创建者便是拥有者，系统在访问矩阵中为新文件增加一列，由用户决定在该列的某个项中应具有哪些访问权，而在另一项中又具有哪些访问权。当用户删除此文件时，系统也要相应地在访问矩阵中将该文件对应的列撤消。

图9-9的访问矩阵如图9-10所示。它是由三个域和8个对象所组成的。当进程在域 D_1 中运行时，它能读文件 F_1 、读和写文件 F_2 。进程在域 D_2 中运行时，它能读文件 F_3 、 F_4 和 F_5 ，以及写文件 F_4 、 F_5 和执行文件 F_4 ，此外还可以使用打印机1。只有当进程在域 D_3 中运行时，才可使用绘图仪2。

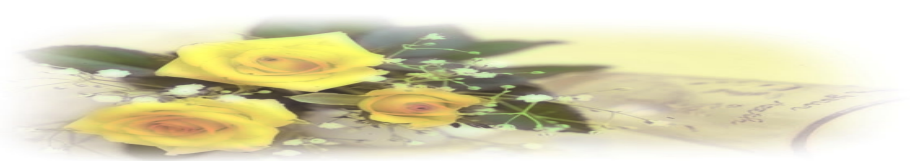




域 \ 对象	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	打印机1	绘图仪2
D ₁	R	R, W						
D ₂			R	R, W, E	R, W		W	
D ₃						R, W, E	W	W

注：R—读，W—写，E—执行。

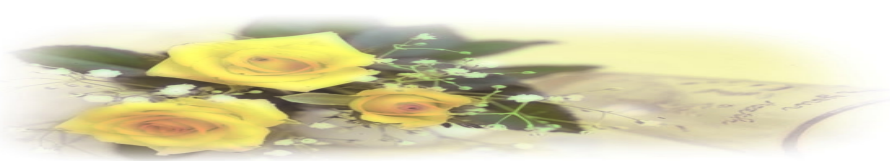
图9-10 一个访问矩阵





3. 具有域切换权的访问矩阵

为了实现在进程和域之间的动态联系，应能够将进程从一个保护域切换到另一个保护域。为了能对进程进行控制，同样应将切换作为一种权力，仅当进程有切换权时，才能进行这种切换。为此，在访问矩阵中又增加了几个对象，分别把它们作为访问矩阵中的几个域；当且仅当 $\text{switch} \in \text{access}(i, j)$ 时，才允许进程从域 i 切换到域 j 。例如，在图9-11中，由于域 D_1 和 D_2 所对应的项目中有一个S即Switch，故而允许在域 D_1 中的进程切换到域 D_2 中。类似地，在域 D_2 和对象 D_3 所对应的项中，也有Switch，这表示在 D_2 域中运行的进程可以切换到域 D_3 中，但不允许该进程再从域 D_3 返回到域 D_1 。

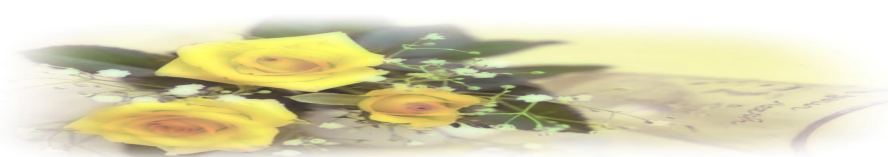




域 \ 对象	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	打印机1	绘图仪2	域D ₁	域D ₂	域D ₃
D ₁	R	R, W								S	
D ₂			R	R, W, E	R, W		W				S
D ₃						R, W, E	W	W			

注：R—读，W—写，E—执行，S—切换。

图9-11 具有切换权的访问控制矩阵

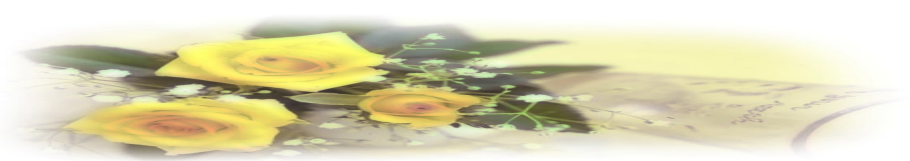




9.4.2 访问矩阵的修改

1. 拷贝权(Copy Right)

我们可利用拷贝权将在某个域中所拥有的访问权($\text{access}(i, j)$)扩展到同一列的其它域中，亦即，为进程在其它的域中也赋予对同一对象的访问权($\text{access}(k, j)$)，如图9-12 所示。





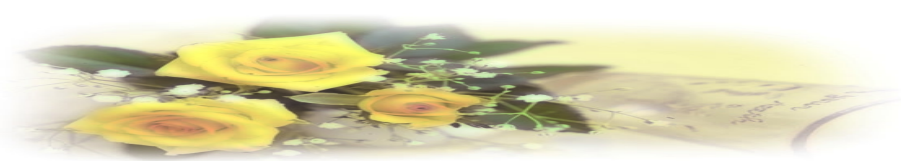
域 \ 对象	F ₁	F ₂	F ₃
D ₁	E		W*
D ₂	E	R*	E
D ₃	E		

(a)

域 \ 对象	F ₁	F ₂	F ₃
D ₁	E		W*
D ₂	E	R*	E
D ₃	E	R	W

(b)

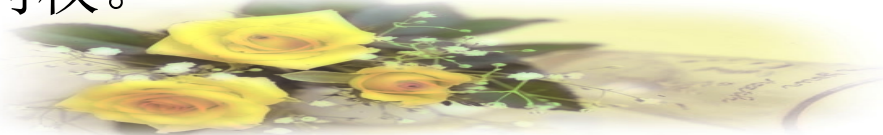
图 9-12 具有拷贝权的访问控制矩阵





2. 所有权(Owner Right)

人们不仅要求能将已有的访问权进行有控制的扩散，而且同样需要能增加某种访问权，或者能删除某种访问权。此时，可利用所有权(O)来实现这些操作。如图9-13所示，如果在 $\text{access}(i, j)$ 中包含所有访问权，则在域 D_i 上运行的进程，可以增加或删除其在 j 列上任何项中的访问权。换言之，进程可以增加或删除在任何其它域中运行的进程对对象 j 的访问权。例如，在图9-13(a)中，在域 D_1 中运行的进程(用户)是文件 F_1 的所有者，他能增加或删除在其它域中的运行进程对文件 F_1 的访问权；类似地，在域 D_2 中运行的进程(用户)是文件 F_2 和文件 F_3 的拥有者，该进程可以增加或删除在其它域中运行的进程对这两个文件的访问权。在图9-13(b)中示出了在域 D_1 中运行的进程删除了在域 D_3 中运行的进程对文件 F_1 的执行权；在域 D_2 中运行的进程增加了在域 D_3 中运行进程对文件 F_2 和 F_3 的写访问权。



第九章 系统安全性



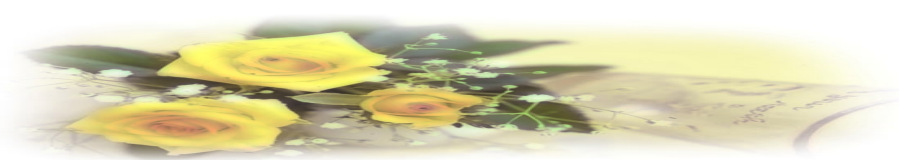
域 \ 对象	F ₁	F ₂	F ₃
D ₁	O, E		W
D ₂		R*, O	R*, O, W
D ₃	E		

(a)

域 \ 对象	F ₁	F ₂	F ₃
D ₁	O, E		
D ₂		O, R, W*	R*, O, W
D ₃		W	W

(b)

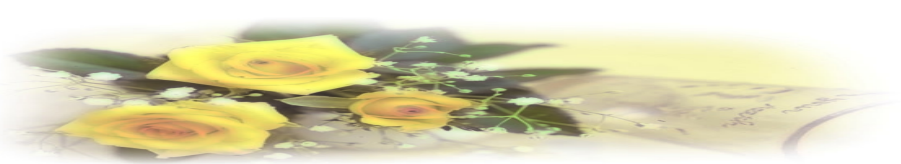
图9-13 带所有权的访问矩阵





3. 控制权(Control Right)

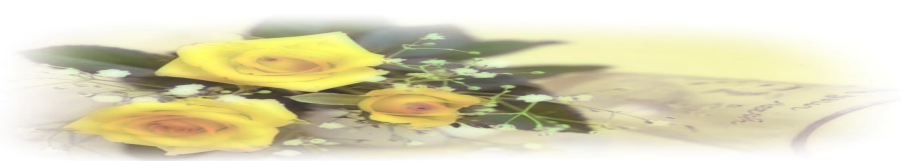
拷贝权和所有权都是用于改变矩阵内同一列的各项访问权的，或者说，是用于改变在不同域中运行的进程对同一对象的访问权。控制权则可用于改变矩阵内同一行中(域中)的各项访问权，亦即，用于改变在某个域中运行进程对不同对象的访问权。如果在 $\text{access}(i, j)$ 中包含了控制权，则在域 D_i 中运行的进程可以删除在域 D_j 中运行进程对各对象的任何访问权。例如在图9-14中，在 $\text{access}(D_2, D_3)$ 中包括了控制权，则一个在域 D_2 中运行的进程能够改变对域 D_3 内各项的访问权。比较图9-11和图9-14可以看出，在 D_3 中已无对文件 F_6 和Plotter 2的写访问权。





域 \ 对象	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	打印机1	绘图仪2	域D ₁	域D ₂	域D ₃
D ₁	R	R, W									
D ₂			R	R, W, E	R, W		W				Control
D ₃						R, W	W				

图9-14 具有控制权的访问矩阵

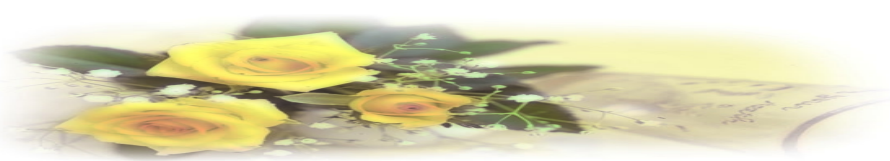




9.4.3 访问控制矩阵的实现

1. 访问控制表(Access Control List)

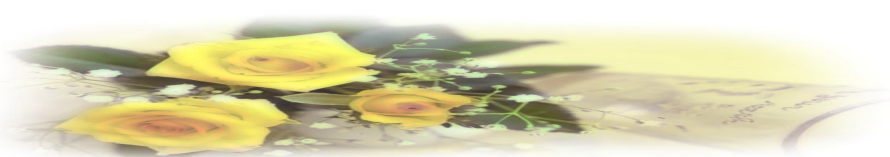
这是指对访问矩阵按列(对象)划分, 为每一列建立一张访问控制表ACL。在该表中, 已把矩阵中属于该列的所有空项删除, 此时的访问控制表是由一有序对(域, 权集)所组成的。由于在大多数情况下, 矩阵中的空项远多于非空项, 因而使用访问控制表可以显著地减少所占用的存储空间, 并能提高查找速度。在不少系统中, 当对象是文件时, 便把访问控制表存放在该文件的文件控制表中, 或放在文件的索引结点中, 作为该文件的存取控制信息。





域是一个抽象的概念，可用各种方式实现。最常见的一种情况是每一个用户是一个域，而对象则是文件。此时，用户能够访问的文件集和访问权限取决于用户的身份。通常，在一个用户退出而另一个用户进入时，即用户发生改变时，要进行域的切换；另一种情况是，每个进程是一个域，此时，能够访问的对象集中的各访问权，取决于进程的身份。

访问控制表也可用于定义缺省的访问权集，即在该表中列出了各个域对某对象的缺省访问权集。在系统中配置了这种表后，当某用户(进程)要访问某资源时，通常是首先由系统到缺省的访问控制表中去查找该用户(进程)是否具有对指定资源进行访问的权力。如果找不到，再到相应对象的访问控制表中去查找。





2. 访问权限(Capabilities)表

如果把访问矩阵按行(即域)划分, 便可由每一行构成一张访问权限表。换言之, 这是由一个域对每一个对象可以执行的一组操作所构成的表。表中的每一项即为该域对某对象的访问权限。当域为用户(进程)、对象为文件时, 访问权限表便可用来描述一个用户(进程)对每一个文件所能执行的一组操作。

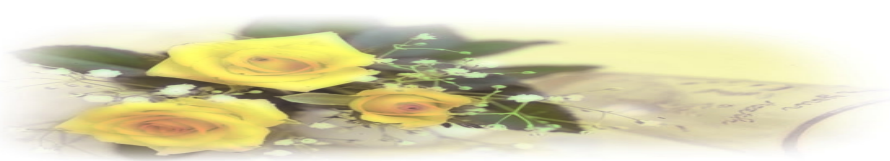
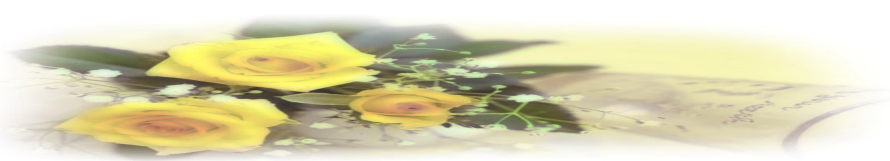




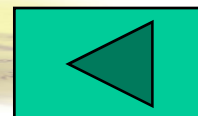
图9-15示出了对应于图9-11 中域 D_2 的访问权限表。在表中共有三个字段，其中类型字段用于说明对象的类型；权力字段是指域 D_2 对该对象所拥有的访问权限；对象字段是一个指向相应对象的指针，对UNIX系统来说，它就是索引结点的编号。由该表可以看出，域 D_2 可以访问的对象有4个，即文件3、4、5和打印机，对文件3的访问权限是只读；对文件4的访问权限是读、写和执行等。





	类型	权 力	对 象
0	文件	R— —	指向文件3的指针
1	文件	RWE	指向文件4的指针
2	文件	RW—	指向文件5的指针
3	打印机	—W—	指向打印机1的指针

图9-15 访问权限表



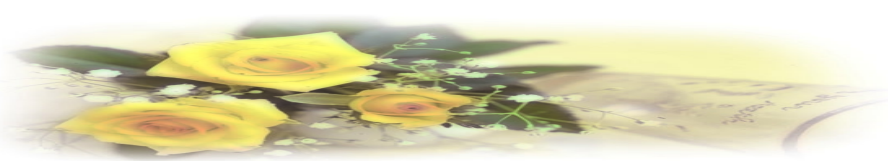


9.5 计算机病毒

9.5.1 计算机病毒的基本概念

1. 计算机病毒的定义

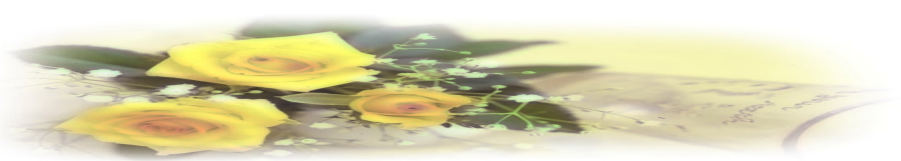
计算机病毒是一段程序，它能不断地进行复制和感染其它程序，无需人为介入便能由被感染的程序和系统传播出去。一般的病毒并不长。对于用C语言编写的病毒程序，通常不超过一页，经编译后小于2 KB；用汇编语言编写的病毒程序则更小，可以小到只有几十到几百个字节。





之所以把这样一段程序称为病毒，是因为它非常像生物学上的病毒。生物学上的病毒能控制一个活细胞的组织，生成成千上万个与原始病毒相同的复制品，并将它们传播到各处。类似地，计算机病毒也可在系统中复制出千千万万个与它自身(一段病毒程序)一样的计算机病毒，并把它传播到各个计算机系统中。

在《中华人民共和国计算机信息系统安全保护条例》中，计算机病毒被定义为：“编制或者在计算机程序中插入的破坏计算机功能或破坏数据，影响计算机系统使用并且能够自我复制的一组计算机指令或者程序代码”。



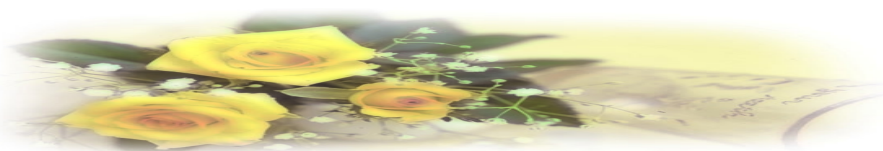


2. 计算机病毒的危害

计算机病毒的危害可表现在如下几个方面：

(1) 占用系统空间。既然病毒是一段程序，就必然会占用一定的磁盘空间和内存空间。虽然一个病毒程序可能并不大，但随着病毒的增加，空闲磁盘空间和内存空间将会迅速减小，以致使存储空间消耗殆尽。

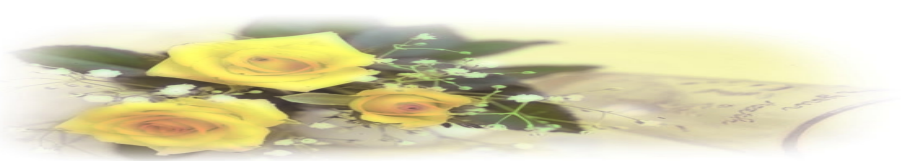
(2) 占用处理机时间。病毒在执行时会占用处理机时间，随着病毒的增加，将会占用更多的处理机时间，这会引起系统运行的速度变得异常缓慢，进一步还可能完全独占处理机时间，而使计算机系统无法再向用户提供服务。





(3) 对文件造成破坏。计算机病毒可以使文件的长度增加或减少，文件的内容发生改变，甚至被删除，使文件丢失。它还可以通过对磁盘的格式化使整个系统中的文件全部消失。

(4) 使机器运行异常。计算机病毒可使计算机屏幕出现异常情况，如提供一些莫名其妙的指示信息，屏幕发生异常滚动，显示异常图形等；还可使机器发生异常情况，使系统的运行明显放缓，以至于完全停机。



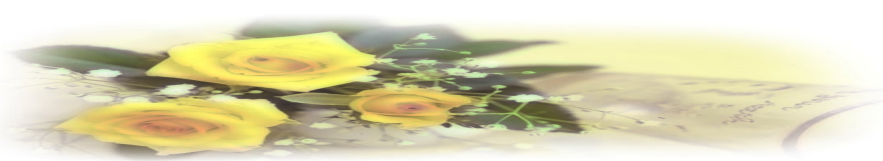


3. 病毒产生的原因

病毒产生的原因有很多种，下面列出几种常见的原因。

(1) 显示自己的能力。有不少编程高手，为表现自己的能力或挑战他人，看别人是否能破解自己编制的病毒程序而编制病毒程序。这种人不仅错误地运用了自己的能力，更目无法纪。

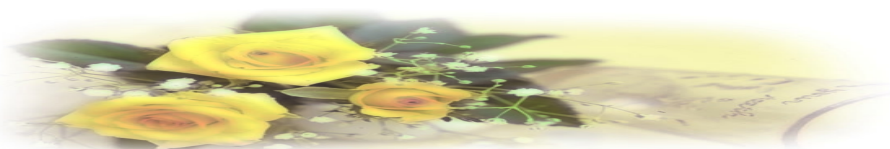
(2) 恶意报复。个别员工对自己所在公司的领导不满，特意制造出病毒来攻击公司中的信息系统，给公司造成损失，以达到报复、发泄私愤的目的。有极少数人对本地政府不满，也通过制造病毒来进行报复。





(3) 恶意攻击。个别宗教和政治狂热者，他们对本国政府或者对其它国家的政府强烈不满，通过制造病毒来进行攻击。

(4) 出错程序。当程序员在编制一个新程序时，若其中存在着一些错误或问题，但他并未及时排除，在运行这样的程序时，有时也会产生一些类似于病毒的不良影响。

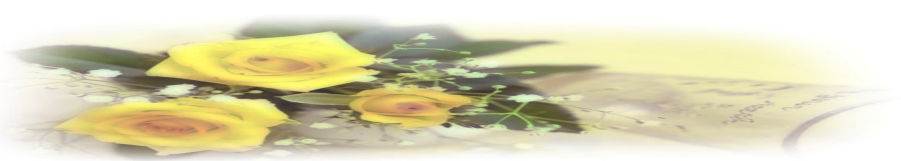




4. 计算机病毒的特征

(1) 寄生性。计算机病毒最大的特点是寄生性，即病毒程序通常都不是一个独立的程序，经常是寄生在某个文件中或者是磁盘的系统区中。寄生于文件中的病毒称为文件型病毒，而侵入到磁盘系统区的则称为系统型病毒。还有一种综合型病毒，它既寄生于文件中，又侵占系统区。

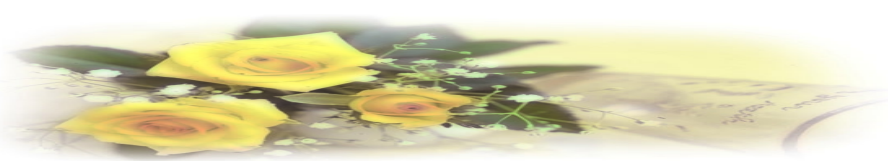
(2) 传染性。计算机病毒在运行过程中将进行自我复制，并将复制品放置在其它文件中或盘上的某个系统区中。文件被感染后便含有了该病毒的一个克隆体，而这个克隆体也同样会再传染给其它的文件，如此不断地传染，使病毒迅速蔓延开来。





(3) 隐蔽性。为了逃避反病毒软件的检测，计算机病毒的设计者通过伪装、隐藏、变态等手段，将病毒隐藏起来，以逃避反病毒软件的检测，使病毒能在系统中长期生存。

(4) 破坏性。如前所述，计算机病毒的破坏性可表现为四个方面，即占用系统空间，占用处理机时间，对系统中的文件造成破坏，使机器运行产生异常情况。

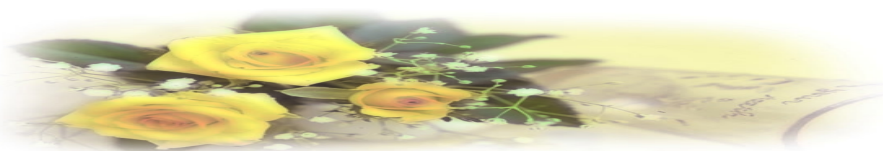




9.5.2 计算机病毒的类型

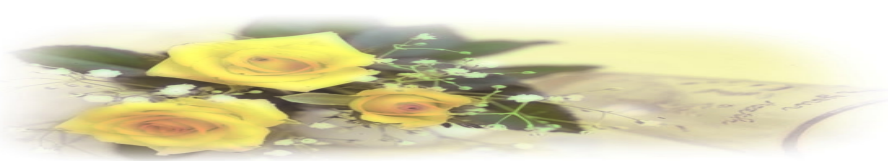
1. 文件型病毒

早期的病毒是覆盖在正常程序上的，但这样，对病毒也存在着一个致命的问题，即由于它会使程序不能正常运行，因此病毒很快就会被用户发现。所以现在大多数病毒都采用寄生的方法，把自己附着在正常程序上，在病毒发作时，原来程序仍能正常运行，以致用户不能及时发现，这样，病毒就有可能长期潜伏下来。我们把这种病毒称为文件型病毒。





文件型病毒是当前最为普遍的病毒形式，病毒程序依附在可执行文件的前面或后面，但要从文件的前端装入病毒(见图9-16(b))，会涉及到文件头中的许多选项，有一定的难度，故大多数病毒是被从程序的后面装入的(见图9-16(c))，并把文件头中起始地址指向病毒的始端。病毒也可以被放在文件的中间，即充斥在程序里的空闲空间中(见图9-16(d))。图9-16示出了上述的几种情况。当受感染的程序执行时，病毒将寻找其它可执行文件继续散播。



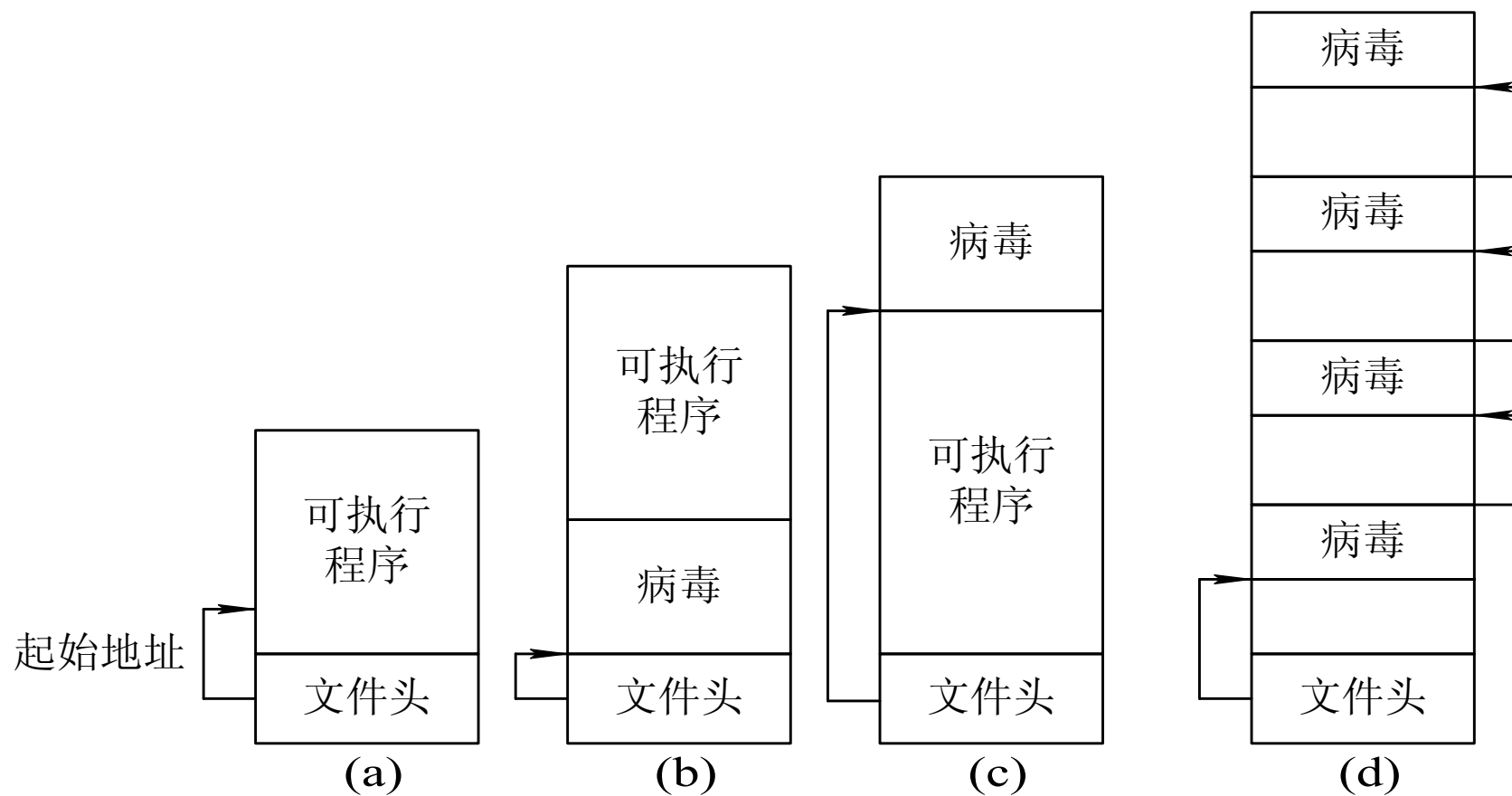
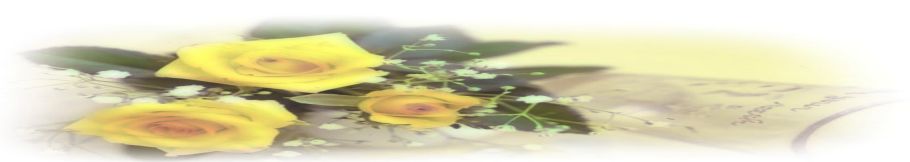


图9-16 病毒附加在文件中的情况

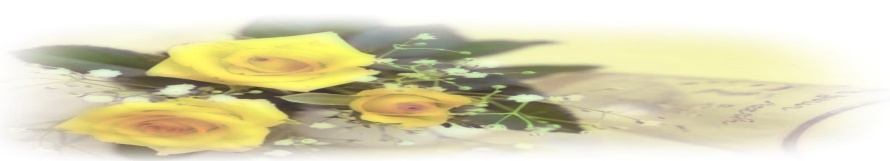




文件型病毒使文件受感染的方式可分为两种，即主动攻击型感染和执行时感染。

(1) 主动攻击型感染。当病毒程序在执行时，它将不断地对磁盘上的文件进行检查，当发现被检测文件尚未被感染时，就去感染它，使其带有病毒。

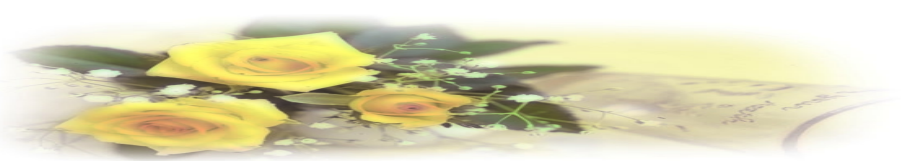
(2) 执行时感染。在病毒环境中，每当一个未被感染的程序在执行时，如果它是病毒所期待的文件类型，且磁盘没有写保护，该程序就会被感染病毒。病毒在感染其它文件时，通常是有针对性的，有的病毒是针对 .com 文件，或者是针对 .exe 文件，或者同时针对 .com 文件和 .exe 文件，也有的病毒则针对其它类型的文件。





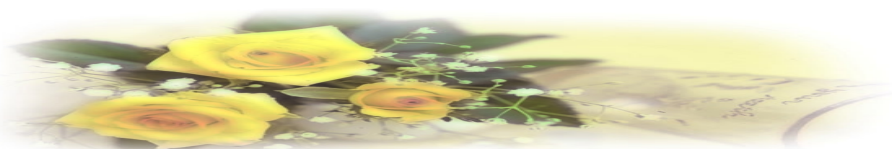
2. 内存驻留病毒

这原本也是一种文件型病毒，但它一旦执行，自己便占据内存驻留区，通常选择占据在内存的上端或下端的中断变量中(通常不会使用的数百个字节的内存区域)。有的病毒为避免其所占据的内存被其它程序覆盖，还会改变操作系统的RAM位图，给系统一个错觉，认为相应的部分内存已分配，便不再将之分配出去。





为了能使自己频繁地执行，通常内存驻留病毒会把陷阱或中断向量的内容复制到其它地方去，而把自己的地址放入其中，使中断或陷阱指向病毒程序的入口；尤其是常选择调用陷阱程序的指针，那么病毒便可在每次系统调用时运行，并且是在核心态，一旦它发现一个可执行的二进制文件便去感染它。当病毒运行完后，再跳转到病毒所保存的陷阱地址，去激活真正的系统调用。



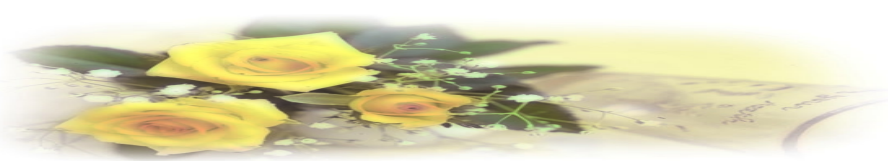


3. 引导扇区病毒

病毒也会寄生于磁盘上用于引导系统的引导区。这样，当系统开机时，病毒便借助于引导过程进入系统。引导型病毒又可分为迁移型和替代型两种。

(1) 迁移型病毒会把真正的引导扇区复制到磁盘的安全区域，以便在完成操作后仍能正常引导操作系统。例如，微软的磁盘格式化程序往往会跳过磁盘的第一条磁道，因此可把引导记录安全地隐藏在这里。

(2) 替代型病毒会取消被入侵扇区的原有内容，而将磁盘所必须用到的程序段和相应的数据融入到病毒程序中。

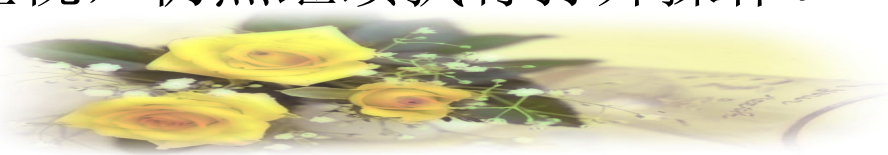




4. 宏病毒

为了提高输入命令或操作的效率，许多软件（如Word、Office）都允许用户把一大串命令写入宏文件，以便用户可以按一次键就能执行多条命令。宏也可以被附加在菜单项里，当该菜单项被选时，宏就可以执行。宏病毒可利用软件所提供的宏功能将病毒插入到带宏的doc文件或dot文件中。由于宏允许包含任何程序，因此也就可以做任何事情，这样宏病毒也就可以做任何事情，如删除文件、导致系统中其它各部分的破坏等。

在Microsoft Word软件中，提供了抵抗宏病毒的保护软件，该软件能够检测到可疑的Word文件，并会警告，以告知用户打开的文件具有潜在危险。但大多数用户并不是很理解此警告的含意，因此未能予以足够的重视，仍然继续执行打开操作。

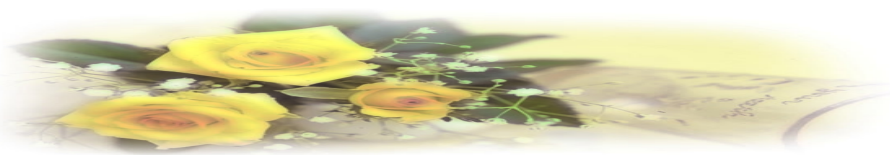




5. 电子邮件病毒

病毒的最新发展是电子邮件病毒的出现。第一个快速传播的电子邮件病毒便是嵌入在电子邮件附件中的Word宏病毒。只要接收者打开电子邮件中的附件，Word宏病毒就会被激活，它将把自身发送给该用户邮件列表中的每个人，然后进行某种破坏活动。

后来出现了更具破坏性的电子邮件病毒，它被直接嵌入到电子邮件中，只要接收者打开含有该病毒的电子邮件，病毒就会被激活。由于电子邮件病毒是通过Internet传播的，因此使病毒的传播速度显著加快，由过去的数个月甚至数年时间减少到数小时。显然，这样快的传播速度，使得反病毒软件很难在病毒造成大规模破坏前，能够作出及时的反应。

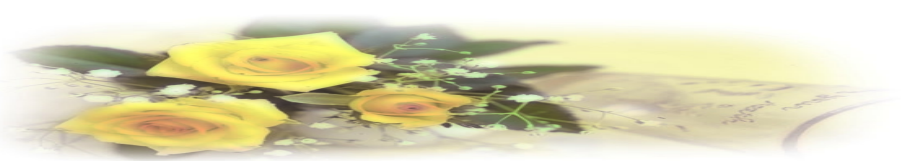




9.5.3 病毒的隐藏方式

1. 伪装

(1) 通过压缩伪装。例如当一个文件感染上病毒后，由于病毒本身是一段程序，当它附加到该文件上后，会使该文件的长度相应地变长，因此人们可利用文件长度发生改变来发现病毒。病毒程序的设计者为了隐藏病毒，通过压缩技术，使感染上病毒的文件长度与原有文件的长度一致，以逃避检查。在使用压缩方法时，在病毒程序中应包含压缩程序和解压缩程序，如图9-17所示。



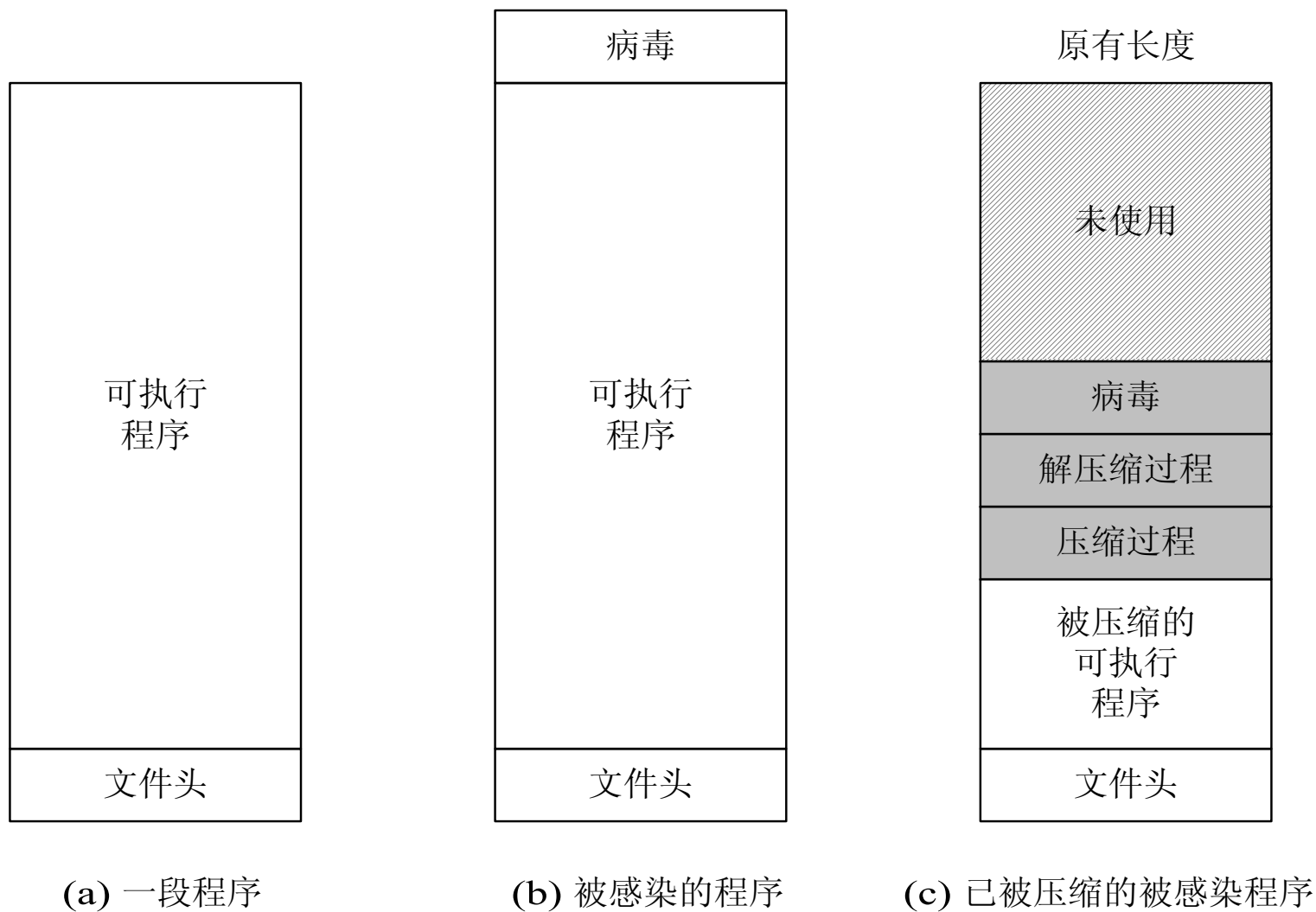
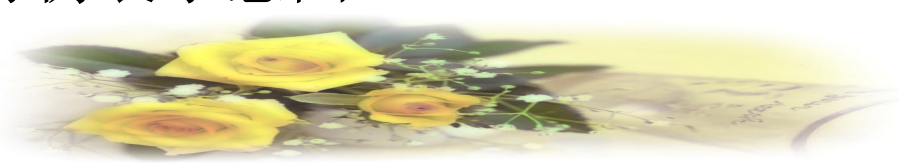
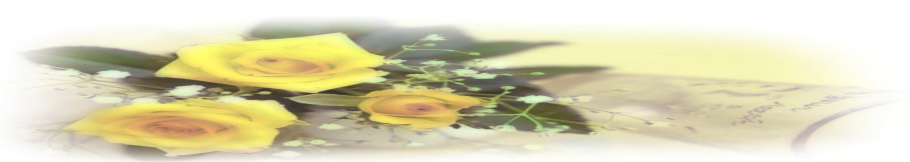


图9-17 病毒伪装示意图





(2) 通过修改日期或时间来伪装。感染上病毒的文件还可能使文件的修改日期和时间改变，因此可通过检测文件的修改日期和时间有无变化，来确定该文件是否已感染上病毒。为此，病毒程序的设计者还会修改感染上病毒的文件的修改日期和时间，使之与原文件相同。

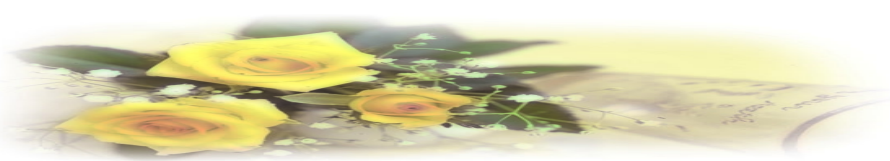




2. 隐藏

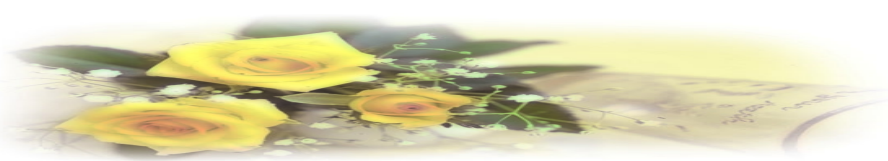
为了逃避反病毒软件的检测，病毒程序的设计者常把病毒隐藏在一个不易检查到的地方。作祟者当前常采用的隐藏方法有以下几种：

(1) 隐藏于目录和注册表空间。在Windows OS中，根目录有足够大的固定空间，目录通常都不会用完，还会有不小的剩余空间，反病毒检测程序也不会检测这里。因此，在目录的末端是一个隐藏病毒的好地方。另外，Windows OS的注册表区同样会有剩余空间，也是病毒可以隐藏的好地方。当然，采用该方法的人需要对Windows OS的内部情况有一定的了解。





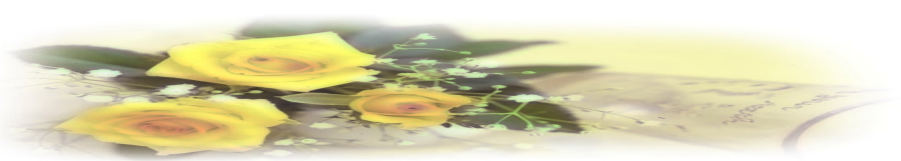
(2) 隐藏于程序的页内零头里。通常一个可执行文件是由若干个程序段和数据段组成的，而现代操作系统大都采取分页管理方式，这样，一个程序段和数据段可能被装入若干个页面中，通常在最后一页都会有页内零头。因此，在系统中就可能存在许多的页面零头，病毒就可隐藏在这些零头中，当病毒占用多个零头时，可由指针将这些零头链接起来。这种隐藏方式不会改变被感染文件的长度。





(3) 更改用于磁盘分配的数据结构。在这种方法中，病毒程序可以为真正的引导记录扇区和病毒自身重新分配磁盘空间，然后再更改用于磁盘分配的数据结构的内容，使病毒合法地占据存储空间，既不会被发现，也不会被覆盖。采用该方法的人需要对Windows OS的内部情况有较详细的了解。

(4) 更改坏扇区列表。病毒程序可以把真正的引导记录扇区和病毒程序分配到磁盘的任意空闲扇区，然后就把这些扇区作为坏扇区，再相应地修改磁盘的坏扇区列表。这样也就可以逃避反病毒软件的检测了。

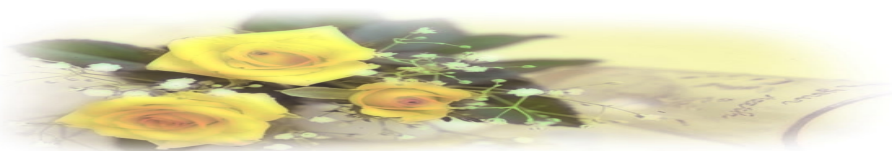




3. 多形态

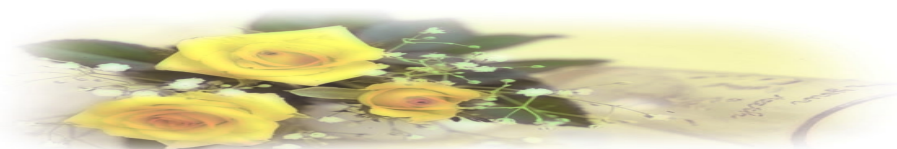
为了逃避反病毒软件的检测，病毒设计者又设计了多形态病毒。这种病毒在进行病毒复制时，采用了较为复杂的技术，使所产生的病毒在功能上是相同的，但形态各异，病毒的形态少者数十种，多者成千上万，然后将这些病毒附加到其它尚未感染上病毒的文件或介质上。常用的产生多态病毒的方法如下：

(1) 插入多余的指令。这是一种较为简单的方法，病毒程序可以在它所生成的病毒中随意地插入多条多余的指令，或改变指令的执行顺序，使所复制的病毒程序发生一些变化，这样就使新病毒发生变异。





(2) 对病毒程序进行加密。在病毒程序中设置一个变量引擎来生成一个随机的密钥，用来加密病毒程序的其余部分，随着每次加密时密钥的不断改变，使所生成的病毒形态各异。

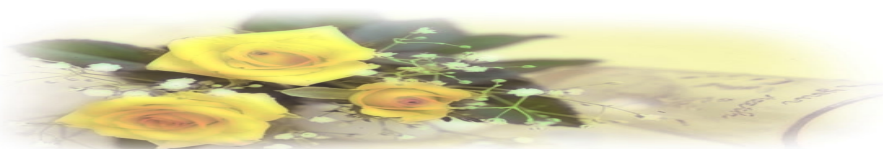




(2) 使用具有高安全性的操作系统，这样的操作系统具有明确的核心和用户界线，以及许多安全保护措施来保障系统的安全，使病毒不能感染到系统代码。

(3) 应使用从正规渠道进来的正版软件，应当清楚地知道，从网上一般的Web站点和公告板下载软件的做法是十分冒险的行为，如果必须下载时，要使用最新的防病毒软件进行扫描以防范病毒入侵。

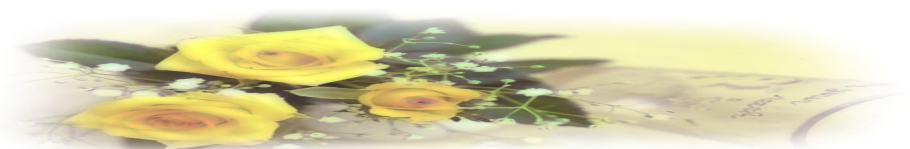
(4) 购买性能优良的反病毒软件，按照规定正确使用，并定期升级。





(5) 对于来历不明的电子邮件，不要轻易打开，因为现在有许多病毒都是通过电子邮件来传染的。

(6) 要定期检查硬盘、软盘及U盘，利用反病毒软件来清除其中的病毒。

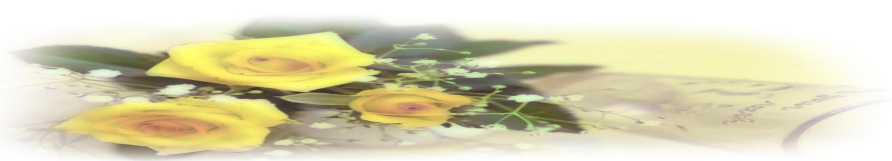




2. 基于病毒数据库的病毒检测方法

1) 建立病毒数据库

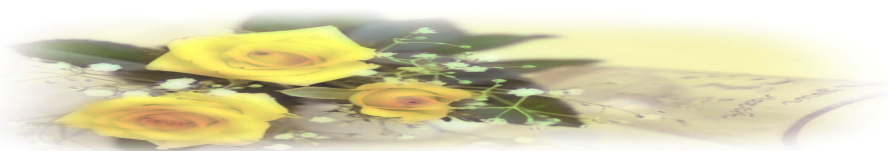
为了建立病毒数据库，首先应当采集病毒的样本。为此，人们设计了一个称为“诱饵文件”的程序，它能让病毒感染，但不执行任何操作。用它来获取病毒的完整内容，然后将病毒的完整代码输入到病毒数据库中。病毒数据库中所收集的病毒样本的种类越多，利用它去检测病毒的成功率也就越高。





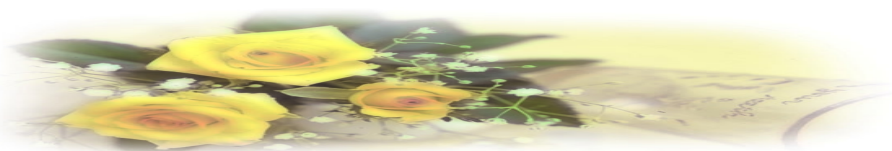
2) 扫描硬盘上的可执行文件

将反病毒软件安装到计算机上之后，便可对硬盘上的可执行文件进行扫描，检查盘上的所有可执行文件，看是否有与病毒数据库中的病毒样本相同的，如发现有，则将它清除。这种采取将硬盘上的可执行文件与病毒数据库中的病毒样本严格匹配的方式，可能会漏掉许多多形态病毒。



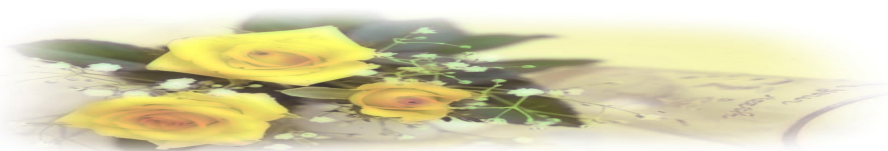


解决这一问题的方法是采用模糊查询软件，这样即使病毒有所变化，只要变化不是太大，如改变不超过三个字节，模糊查询软件都可以将它们检测出来。但模糊查询方法不仅会使查询速度减慢，而且还会导致病毒扩大化，即把正常程序也误认为是病毒。因此模糊查询也应掌握一个度。一个比较完善的方法是，使扫描软件能识别病毒的核心代码，尽管病毒会千变万化，但病毒的核心代码不会改变，因此就不会漏掉病毒。





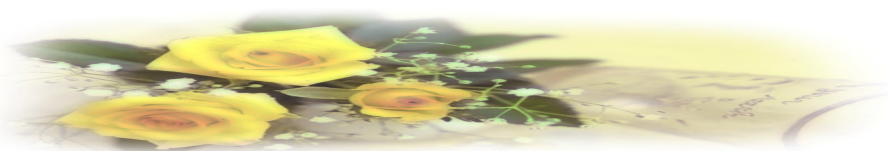
解决这一问题的方法是采用模糊查询软件，这样即使病毒有所变化，只要变化不是太大，如改变不超过三个字节，模糊查询软件都可以将它们检测出来。但模糊查询方法不仅会使查询速度减慢，而且还会导致病毒扩大化，即把正常程序也误认为是病毒。因此模糊查询也应掌握一个度。一个比较完善的方法是，使扫描软件能识别病毒的核心代码，尽管病毒会千变万化，但病毒的核心代码不会改变，因此就不会漏掉病毒。





3. 基于文件改变的病毒检测方法

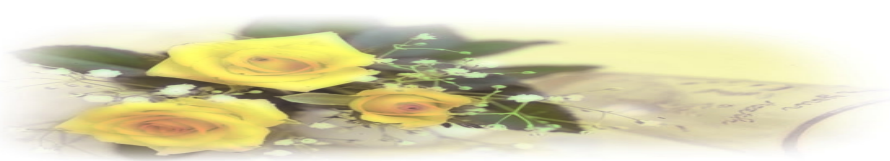
基于病毒数据库的病毒检测方法虽然比较好，但由于有成千上万个病毒样本和需要检查成千上万个文件，因此每次检查都需要花费很长的时间，这将使正常程序的运行变得十分缓慢。一种比较简单的方法是基于文件改变的病毒检测方法。可以通过被感染文件的长度或者日期和时间的改变来发现病毒。这种方法在早期还可奏效，但现在的病毒会利用压缩技术使感染后的文件长度保持不变，使基于文件改变的病毒检测方法不能检测出这种伪装病毒。但这种伪装病毒难于逃避基于病毒数据库的病毒检测方法的检查。





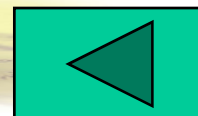
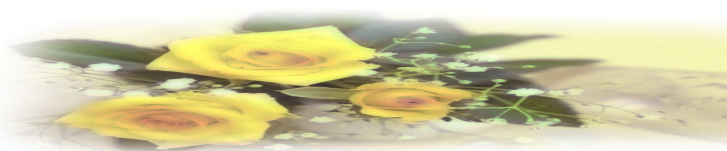
4. 完整性检测方法

完整性检测程序首先扫描硬盘，检查是否有病毒，当确信硬盘是干净的后，它才正式工作。它首先计算每个文件的检查和(或称校验和)，然后再计算目录中所有相关文件的检查和，将所有这些检查和都写入一个检查和文件中，利用它们来对文件是否被病毒感染进行检查。等到下一次检测病毒时，完整性检测程序将重新计算所有文件的检查和，并分别与原来文件的检查和进行比较，若不匹配，就表明该文件已发生改变，由此可以认定该文件已被感染上病毒。





当病毒设计者了解这种反病毒方法后，它可以仿造完整性检测方法，计算已感染病毒的文件의 检查和，用它来代替检查和文件中的正常值。为保证检查和文件中数据不被更改，应将检查和文件隐藏起来，但这也还是可能被找到。更好的方法是对检查和文件进行加密，而且可以采用智能卡技术，将加密密钥直接做在芯片上。





第十章 UNIX系统内核结构

[10.1 UNIX系统概述](#)

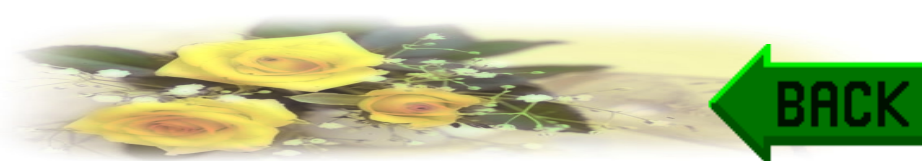
[10.2 进程的描述和控制](#)

[10.3 进程的同步与通信](#)

[10.4 存储器管理](#)

[10.5 设备管理](#)

[10.6 文件管理](#)



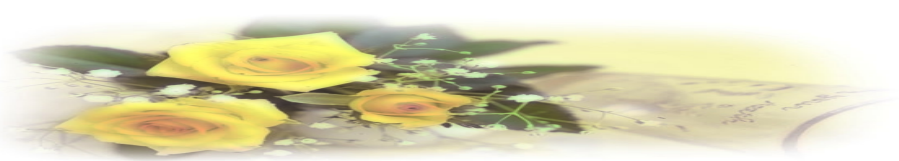


10.1 UNIX系统概述

10.1.1 UNIX系统的发展史

1. UNIX系统的发展

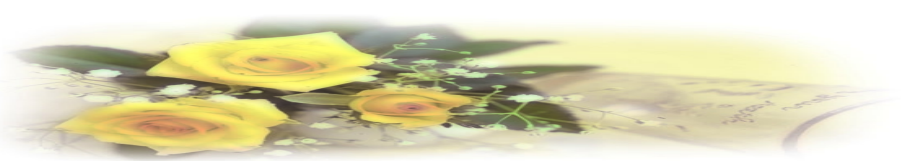
UNIX系统是美国电报电话公司(AT&T)Bell实验室的Ennis Ritchie和Ken Thompson合作设计和实现的。他们在设计时，充分地吸取了以往OS(其中包括著名的CTSS和MULTICS系统)设计实践中的各种成功经验和教训。在DEC公司的小型机PDP7上实现并于1971年正式移植到PDP11计算机上。





第十章 UNIX系统内核结构

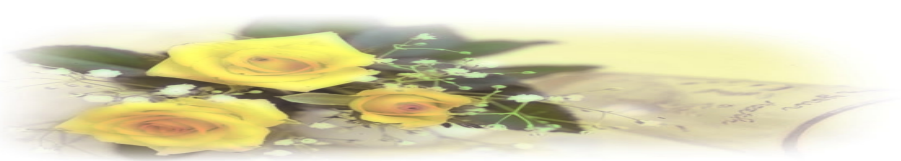
最初的UNIX版本是用汇编语言编写的。不久，Thompson用一种较高级的B语言重写了该系统。1973年Ritchie又用C语言对UNIX进行了重写，形成了最早的正式文件UNIX V5 版本。1976年正式公开发表了UNIX V6版本，并开始向美国各大学及研究机构颁发了使用UNIX的许可证，并提供了源代码，以鼓励他们对UNIX加以改进，因而又推动了UNIX 的迅速发展。





第十章 UNIX系统内核结构

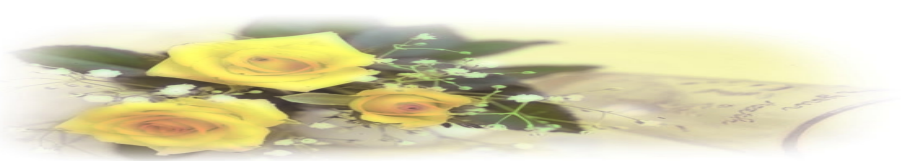
1978年发表了UNIX V7版本，它是在PDP 11/70上运行的，后来移植到DEC公司的VAX系列计算机上。1982至1983年期间，又先后宣布了UNIX System III和UNIX System V；1984年推出了UNIX System V 2.0；1987年发布了UNIX System V 3.0版本，分别称为UNIX SVR 2和UNIX SVR 3；1989年宣布了UNIX SVR 4；1992年又发表了UNIX SVR 4.2版本。





2. 两大集团对峙

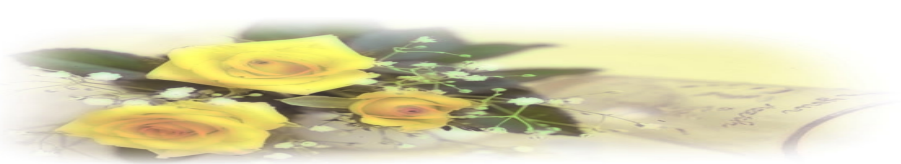
在UNIX系统的发展史上必须说明的是，由于UNIX的开放性、发展概念和商业利益等因素，使UNIX呈现出“百家争鸣”的盛况，后又进一步形成了两大阵营对峙的局面。此即，由IBM和DEC等公司于1988年5月结成了开放软件基金会OSF集团，以及由AT&T、SUN和NCR等公司于同年12月结成了UI集团。他们分别推出了自己的UNIX系统产品。其中，UI推出的是“SVR 4”，而OSF推出的是“OSF/I”。虽然两者都是UNIX，但它们在系统构架、命令操作以及管理方式上，都有所不同。两者在市场上展开了激烈的竞争。





第十章 UNIX系统内核结构

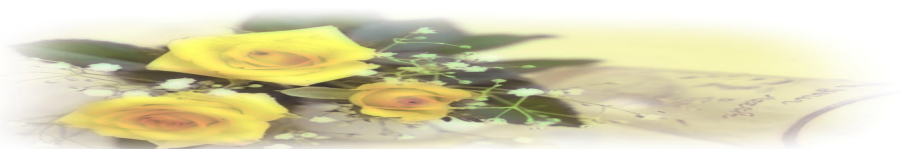
应当看到，虽然两种UNIX系统并存，但在他们之间并不相互兼容，这对用户非常不利。因而，这无疑会影响到UNIX对用户的吸引力，加之，随着Microsoft公司的迅速崛起，并以惊人的速度由传统的PC机市场向工作站和网络市场扩张，迫使UI和OSF两大集团不得不相互让步、携手言和，从而共同制定了应用程序接口API(Application Program Interface)标准技术规范，并联合开发共同开放软件环境COSE(Common Open Software Enviroment)。





第十章 UNIX系统内核结构

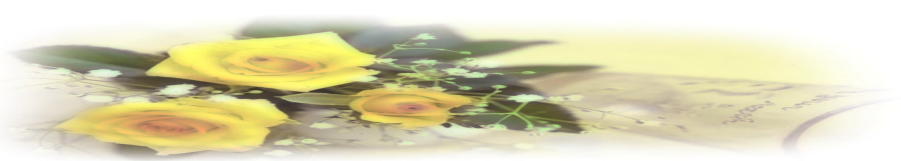
由于UNIX这一名字已被X/Open用作注册商标，因而其他公司所开发的UNIX产品不能再用UNIX这一名字，致使不同的UNIX系统在不同的公司，甚至是在不同的机器上，都各用自己的名字，如IBM RS/6000上的“AIX”(System V)操作系统、Sun公司的“Sun OS”(4.3 BSD或SVR 4)和“Solarix”操作系统、HP公司的“HPUX”(System V)以及SCO公司的“SCO UNIX”(SVR 3.2)操作系统等。





3. 网络操作系统UNIX

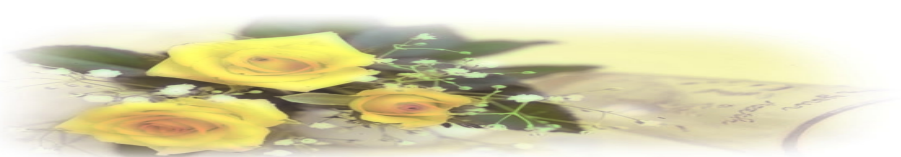
UNIX凭借其“开放性”、“先进性”以及先入为主的优势，使20世纪70年代成为UNIX时代。70年代同时也是网络的萌芽时代，相应地，1976年人们便开发了一个UNIX网络应用程序。该程序随着UNIX V7版本一并发行。1980年9月Bell实验室等又为美国国防部在Berkeley的UNIX上开发了TCP/IP协议系统，于1983年8月对外发行，该协议得以很快发表和普及，从而成为后来的Internet上最重要的网络协议。到80年代中期，他们已在UNIX System V上开发出多种基于TCP/IP的网络软件，并将它们构成一个TCP/IP协议软件包。





第十章 2000X系统内核结构

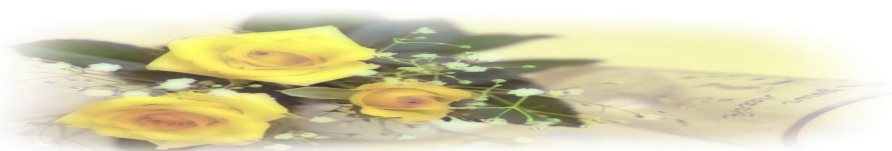
20世纪80年代是LAN快速发展的10年。1984年，Novell公司推出了以LAN为环境的Netware V1.0，继之它经过了不断的改进并增强了其功能，相应的版本由V1.0经过V2.X到V3.X。仅经历短短的3年，到1987年时，其销量已占居全球第一位，并由于此后它长期保持优势而使之成为网络工业界的标准。





第十章 UNIX系统内核结构

进入90年代后，企业网络和Internet得到极其迅速的发展和广泛应用，致使计算机网络已经无所不在，也形成了巨大的网络软件市场。此时，一些主要的UNIX系统开发商也加强了在网络方面更深入的研究，于是也不断地推出用于企业网络的UNIX网络OS版本，如SCO公司的Unixware NOS和Sun公司的Solaris NOS等。

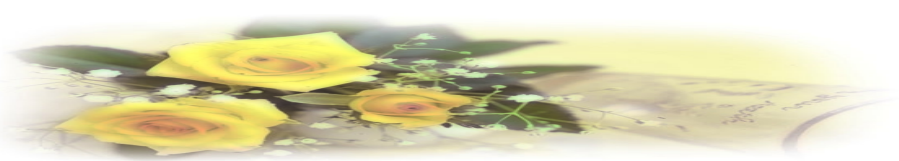




10.1.2 UNIX系统的特征

1. 开放性

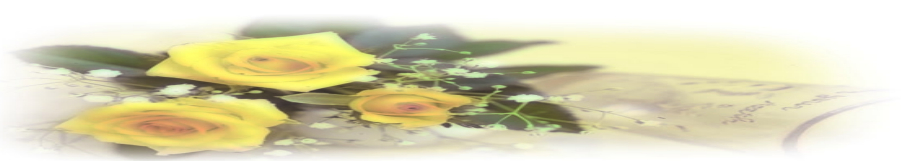
UNIX系统最本质的特征是开放性。所谓开放性，是指系统遵循国际标准规范；凡遵循国际标准所开发的硬件和软件，均能彼此兼容，并可方便地实现互连。开放性已成为20世纪90年代计算机技术的核心问题，也是一个新推出的系统或软件能否被广泛应用的重要因素。人们普遍认为：UNIX是目前开放性最好的OS，是目前惟一能稳定运行在从微型机到大、中型等各种机器上的OS，而且还能方便地将已配置了UNIX OS的机器互连成计算机网络。





2. 多用户、多任务环境

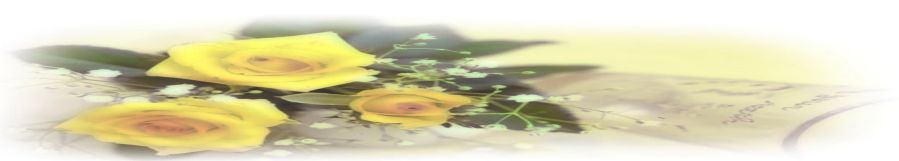
UNIX系统是一个多用户、多任务OS，它既可以同时支持数十个乃至数百个用户通过各自的联机终端同时使用一台计算机，而且还允许每个用户同时执行多个任务。例如，在进行字符图形处理时，用户可建立多个任务，分别用于处理字符的输入、图形的制作和编辑等任务。





3. 功能强大且高效

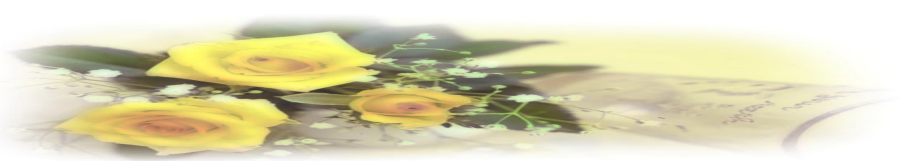
UNIX系统提供了精选的、丰富的系统功能，使用户可方便、快速地完成许多其它OS所难于实现的功能。UNIX已成为世界上功能最强大的操作系统之一，而且它在许多功能的实现上还有其独到之处，并且是高效的。例如，UNIX的目录结构、磁盘空间的管理方式、I/O重定向和管道功能等。其中，不少功能及其实现技术已被其它OS所借鉴。





4. 丰富的网络功能

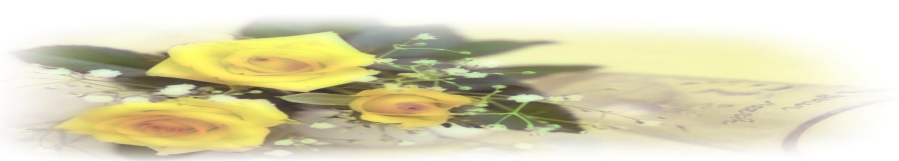
UNIX系统还提供了十分丰富的网络功能。作为Internet网络技术基础的TCP/IP协议，便是在UNIX系统上开发出来的，并已成为UNIX系统不可分割的部分。UNIX系统还提供了许多最常用的网络通信协议软件，其中包括网络文件系统NFS软件、客户/服务器协议软件Lan Manager Client/Server及IPX/SPX软件等。通过这些产品可以实现在各UNIX系统之间，UNIX与Novell的Netware，以及MS-Windows NT、IBM LAN Server等网络之间的互连和互操作。





5. 支持多处理器功能

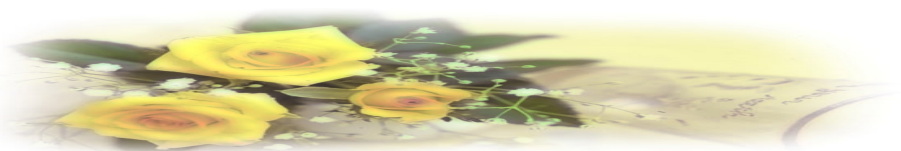
与Windows NT及Netware等OS相比较，UNIX是最早提供支持多处理器功能的OS，它所能支持的多处理器数目也一直处于领先水平。例如，1996年推出的NT 4.0只能支持1~4个处理器，而Windows 2000最多也只支持16个处理器，然而UNIX系统在20世纪90年代中期便已能支持32~64个处理器，而且拥有数百个乃至数千个处理器的超级并行机也普遍支持UNIX。





10.1.3 UNIX系统的内核结构

可以把整个UNIX系统分成四个层次。其最低层是硬件，作为整个系统的基础。次低层是OS核心，包括前面所介绍的进程管理、存储器管理、设备管理和文件管理四大资源管理功能。上面第二层是OS与用户的接口Shell以及编译程序等。最高层是应用程序。作为OS的核心，它应具有两方面的接口：一方面是核心与硬件的接口，它通常是由一组驱动程序和一些基本的例程所组成的；另一方面就是核心与Shell的接口，它由两组系统调用及命令解释程序等所组成。核心本身又可分成两大部分：一部分是进程控制子系统；另一部分则是文件子系统。两组系统调用分别与这两大子系统交互。图10-1示出了UNIX核心的框图。



第十章 UNIX系统内核结构

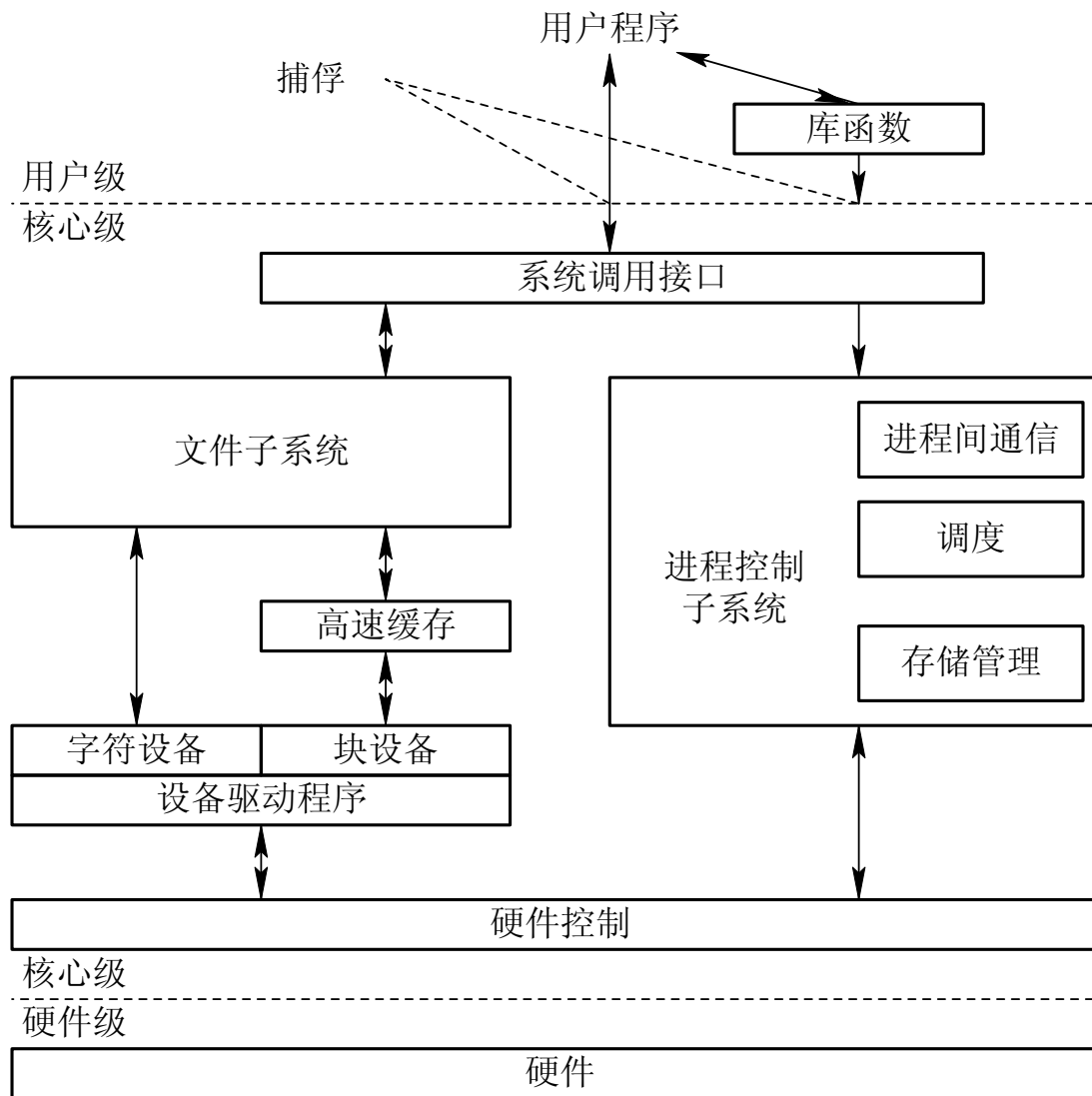


图10-1 UNIX核心的框图

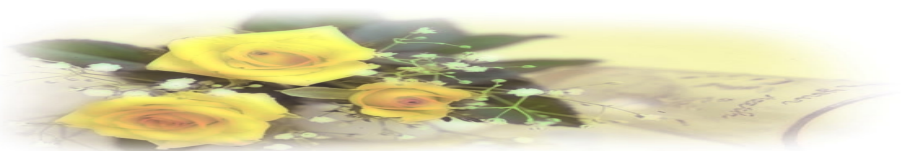


1. 进程控制子系统

进程控制子系统负责对四大资源中的两大资源——处理机和存储器进行管理。进程控制子系统的功能可分成以下几个方面：

(1) 进程控制。在UNIX系统中提供了一系列用于对进程进行控制的系统调用，例如，应用程序可利用系统调用fork创建一个新进程；用系统调用exit结束一个进程的执行。

(2) 进程通信。在UNIX系统中提供了许多进程间通信的手段，例如，用于实现进程之间通信的消息机制，用于在同一用户的各进程之间通信的“信号”通信工具以及性能优良的信号量机制等。

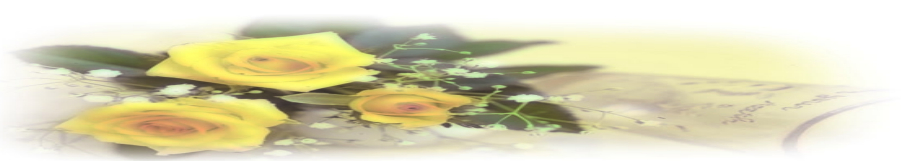




第十章 UNIX系统内核结构

(3) 存储器管理。该功能用于为进程分配物理存储空间。为了提高内存利用率且方便用户，可采用段页式存储管理方式；可利用请求调页法实现虚拟存储器功能，以便从逻辑上扩充内存。此外，还实现了外存与内存间的对换功能。

(4) 进程调度。在UNIX系统中所采用的进程调度算法，是动态优先数轮转调度算法。系统按优先数最小者优先的策略，为选中的某一进程分配一个CPU时间片。当进程运行完一个时间片后，内核便把它送回就绪队列的末尾。



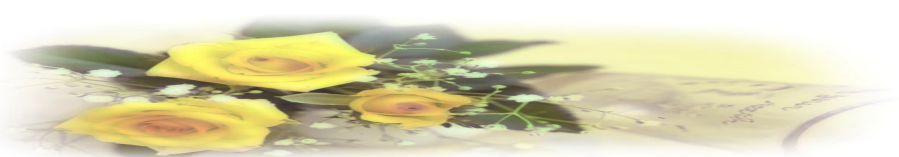


2. 文件子系统

文件子系统用于有效地管理系统中的所有设备和文件。其功能可分成以下三个方面：

(1) 文件管理。该功能用于为文件分配存储空间、管理空闲磁盘块、控制对文件的存取以及为用户检索数据。用户可通过一组系统调用来实现对文件的各种操作。

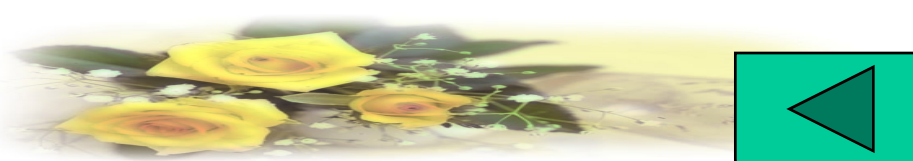
(2) 高速缓冲机制。为使核心与外设之间的数据流在速率上相匹配，设置了多个缓冲区，每个缓冲区的大小与一个盘块的大小相当。这些缓冲区被分别链入各种链表中，如空闲缓冲区链表等。





第十章 UNIX系统内核结构

(3) 设备驱动程序。UNIX系统把设备分成块设备(如磁盘、磁带等)和字符设备(如打印机)两类。相应地，也把驱动程序分成两类，文件子系统将在缓冲机制的支持下，与块设备的驱动程序之间交互作用。



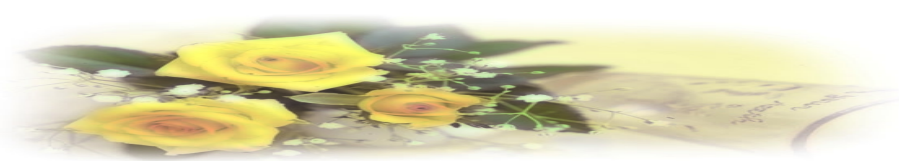


10.2 进程的描述和控制

10.2.1 进程控制块

在UNIX系统V中，把进程控制块(PCB)分为四部分：

- (1) 进程表项，其中包括最常用的核心数据。
- (2) U区，用于存放用户进程表项的一些扩充数据。
- (3) 系统区表，存放各个区在物理存储器中的地址信息等。
- (4) 进程区表，用于存放各区的起始虚地址及指向系统区表中对应区表项的指针。



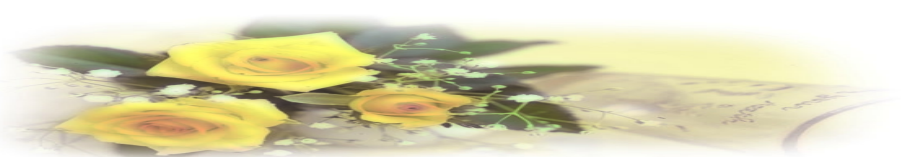


1. 进程表项(Process Table Entry)

用于描述和控制一个进程的信息通常都很多，其中有些是经常要被访问的，如进程标识符、进程状态等。为了提高对这些信息访问的效率，系统设计者将这些信息放在进程表项中，又称之为Proc表或Proc结构，使之常驻内存。在每个进程表项中，含有下述一些具体信息：

(1) 进程标识符(PID)，也称内部标识符，为方便用户使用，这里惟一地标识一个进程的某个整数。

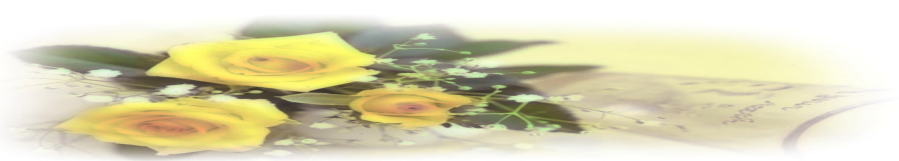
(2) 用户标识符(UID)，标识拥有该进程的用户。





第十章 UNIX系统内核结构

- (3) 进程状态，表示该进程的当前状态。
- (4) 事件描述符，记录使进程进入睡眠状态的事件。
- (5) 进程和U区在内存或外存的地址，核心可利用这些信息做上、下文切换。
- (6) 软中断信息，记录其它进程发来的软中断信号。
- (7) 计时域，给出进程的执行时间和对资源的利用情况。
- (8) 进程的大小，这是核心在为进程分配存储空间时的依据，包括正文段长度和栈段长度等。



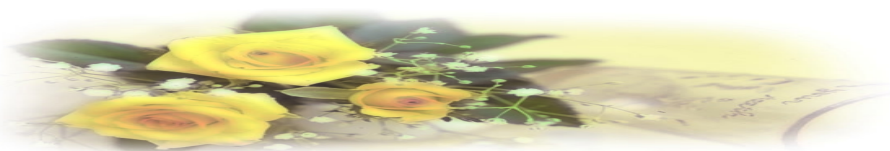


第十章 UNIX系统内核结构

(9) 偏置值nice，供计算该进程的优先数时使用，可由用户设置。

(10) P_Link指针，这是指向就绪队列中下一个PCB的指针。

(11) 指向U区进程正文、数据及栈在内存区域的指针。



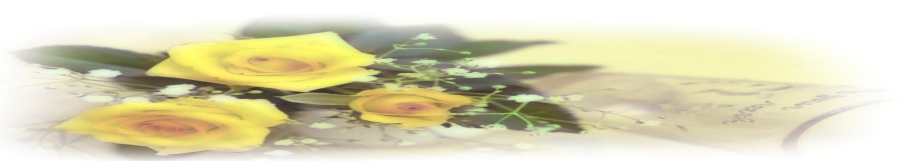


2. U区(U Area)

为了存放用于描述和控制进程的另一部分信息，系统为每一个进程设置了一个私用的U区，又称之为User结构，这部分数据并非常驻内存，其中含有下述信息：

(1) 进程表项指针，指向当前(正在执行)进程的进程表项。

(2) 真正用户标识符u-ruid(real user ID)，这是由超级用户分配给用户的标识符，以后，每次用户在登录进入系统时，均须输入此标识符。





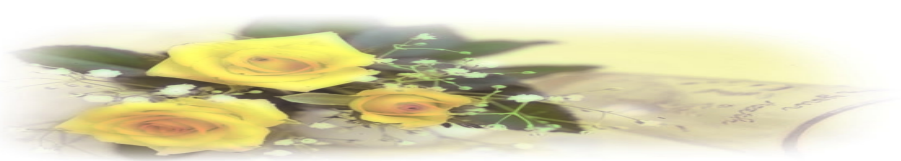
第十章 UNIX系统内核结构

(3) 有效用户标识符u-euid(effective user ID)，在一般情况下，它与ruid相同，但在其他用户允许的情况下，可用系统调用setuid将它改变为其他用户标识符，以获得对该用户的文件进行操作的权力。

(4) 用户文件描述符表，其中记录了该进程已打开的所有文件。

(5) 当前目录和当前根，用于给出进程的文件系统环境。

(6) 计时器，记录该进程及其后代在用户态和核心态运行的时间。





第十章 UNIX系统内核结构

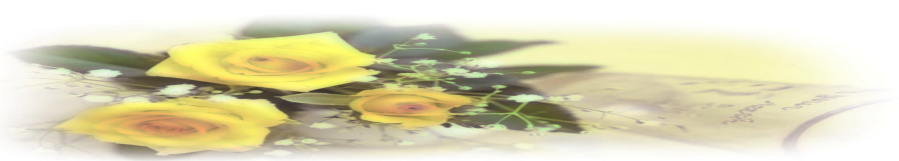
(7) 内部I/O参数，给出要传输的数据量、源(或目标)数据的地址、文件的输入/输出偏移量。

(8) 限制字段，指对进程的大小及其能“写”的文件大小进行限制。

(9) 差错字段，记录系统调用执行期间所发生的错误。

(10) 返回值，指出系统调用的执行结果。

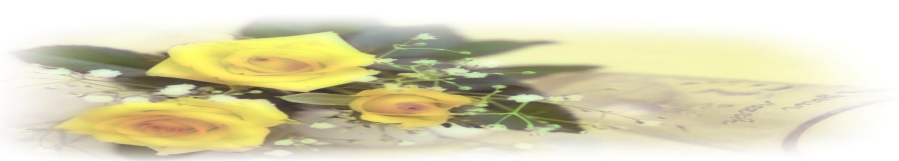
(11) 信号处理数组，用于指示在接收到每一种信号时的处理方式。





3. 系统区表(System Region Table)

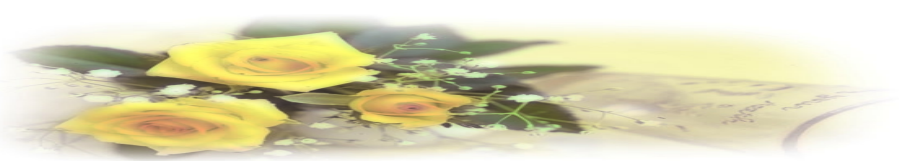
系统V把一个进程的虚地址空间划分为若干个连续的区域：正文区、数据区、栈区等。这些区是可被共享和保护独立实体。多个进程可共享一个区，例如，多个进程共享一个正文区，即几个进程将执行同一个(子)程序。同样，多个进程也可共享一个数据区。为了对区进行管理，在核心中设置了一个系统区表(简称区表)，在各表项中记录了以下有关描述活动区的信息：





第十章 UNIX系统内核结构

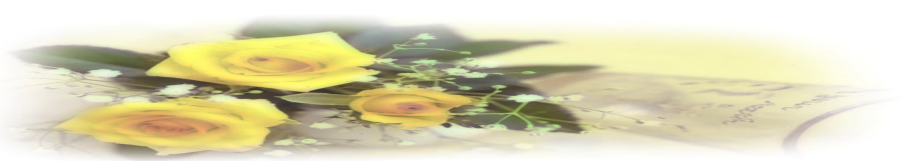
- (1) 区的类型和大小。
- (2) 区的状态。一个区有这样几种状态：锁住、在请求中、在装入过程、有效(区已装入内存)。
- (3) 区在物理存储器中的位置。
- (4) 引用计数，即共享该区的进程数。
- (5) 指向文件索引结点的指针。





4. 进程区表(Process Region Table)

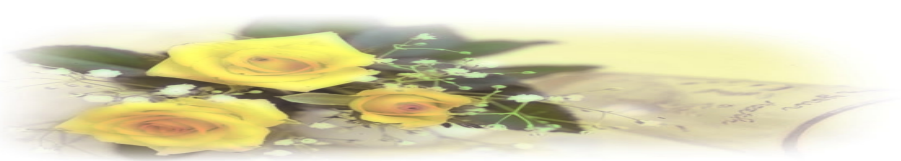
为了记录进程的每个区在进程中的虚地址，并通过它找到该区在物理存储器中的实地址，系统为每个进程配置了一张进程区表。表中的每一项记录一个区的起始虚地址及指向系统区表中对应的区表项的指针。这样，核心可通过查找进程区表和系统区表，将区的逻辑地址变换为物理地址。可见，进程区表和系统区表用于对区地址进行映像(射)。这里用两张区表实现地址映射，是为了便于实现对区的共享。





第十章 UNIX系统内核结构

图10-2示出A和B两个进程的进程区表和系统区表。在A进程区表中的正文区、数据区和栈区中的指针，分别指向相应于a、b、c区的系统区表项。由于A和B进程共享正文区，故它们都指向同一个正文区a。一个进程的数据结构是由上述的进程表项、U区、进程区表与系统区表项所组成的，它们之间的关系如图10-3所示。



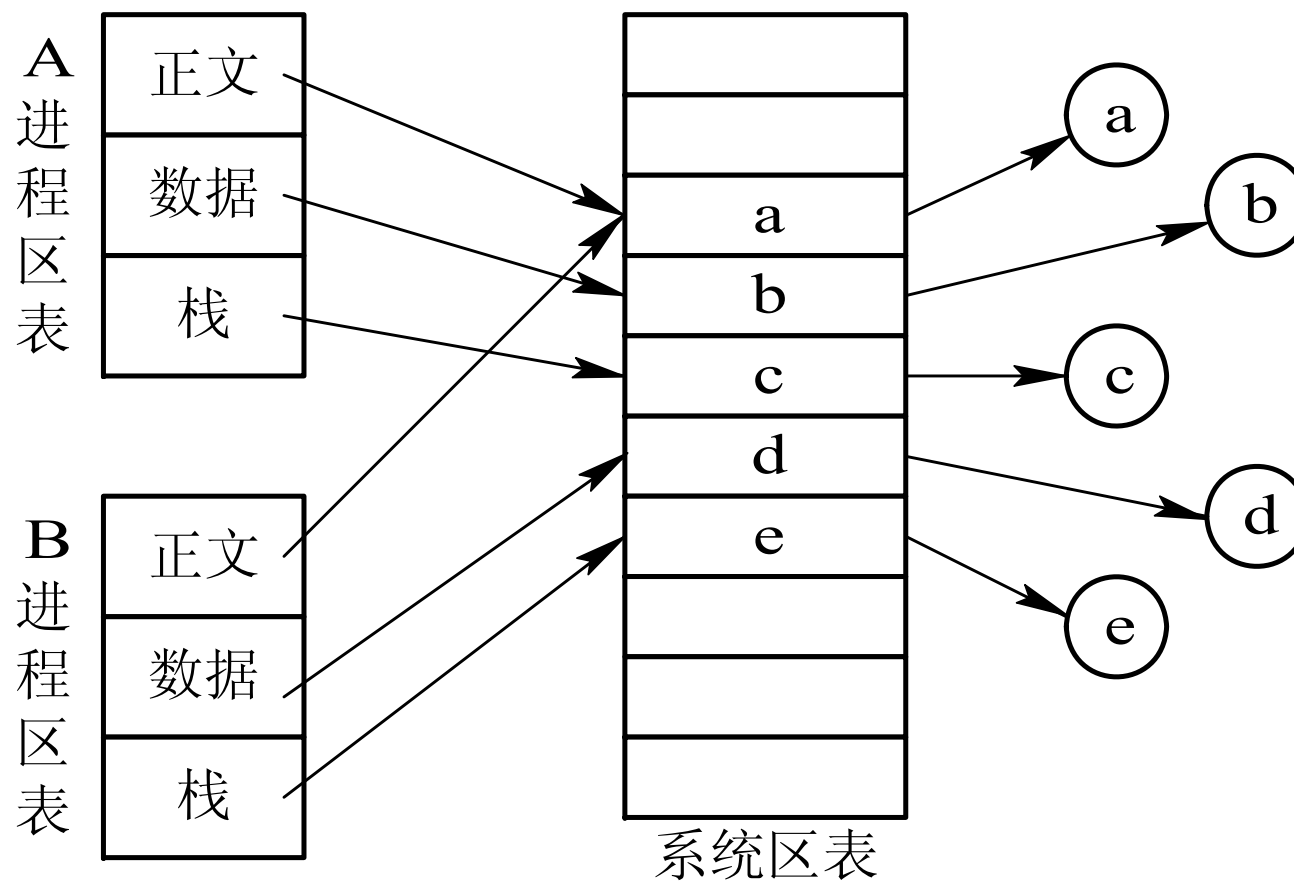


图10-2 进程区表项、系统区表项和区的关系

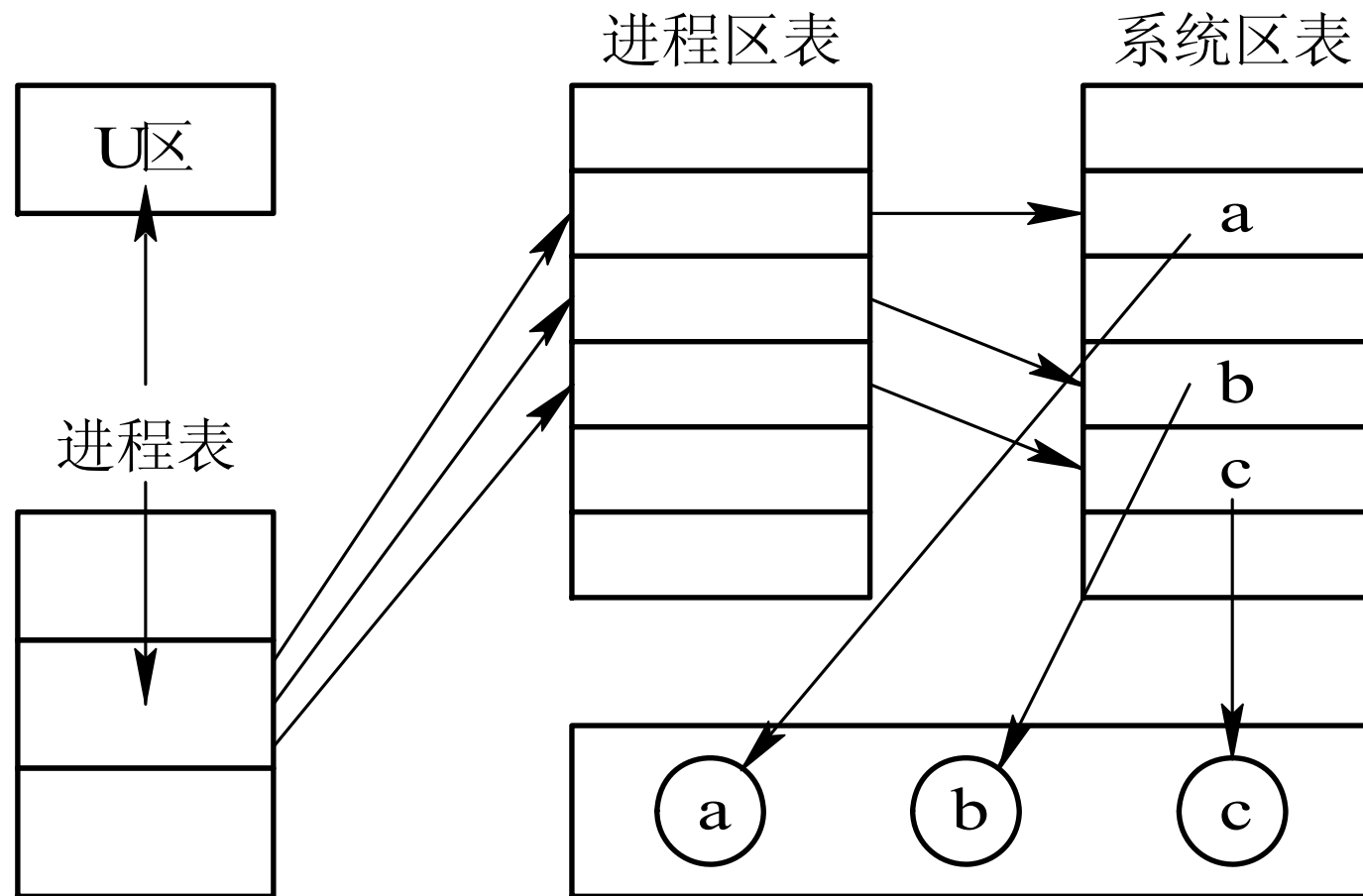
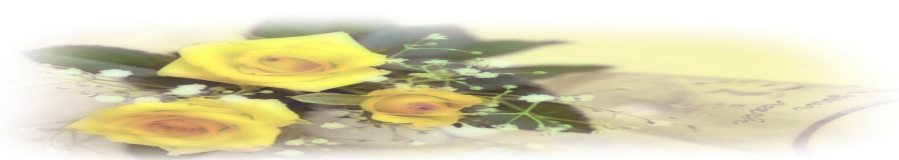


图10-3 进程的数据结构

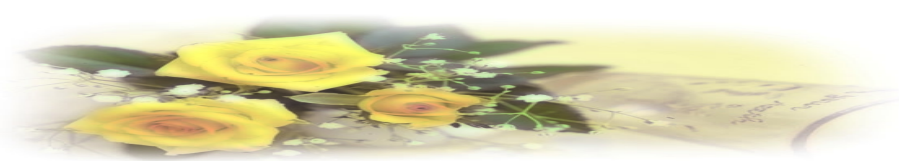




10.2.2 进程状态与进程映像

1. 进程状态

在UNIX内核中，为进程设置了如图10-4所示的9种状态。



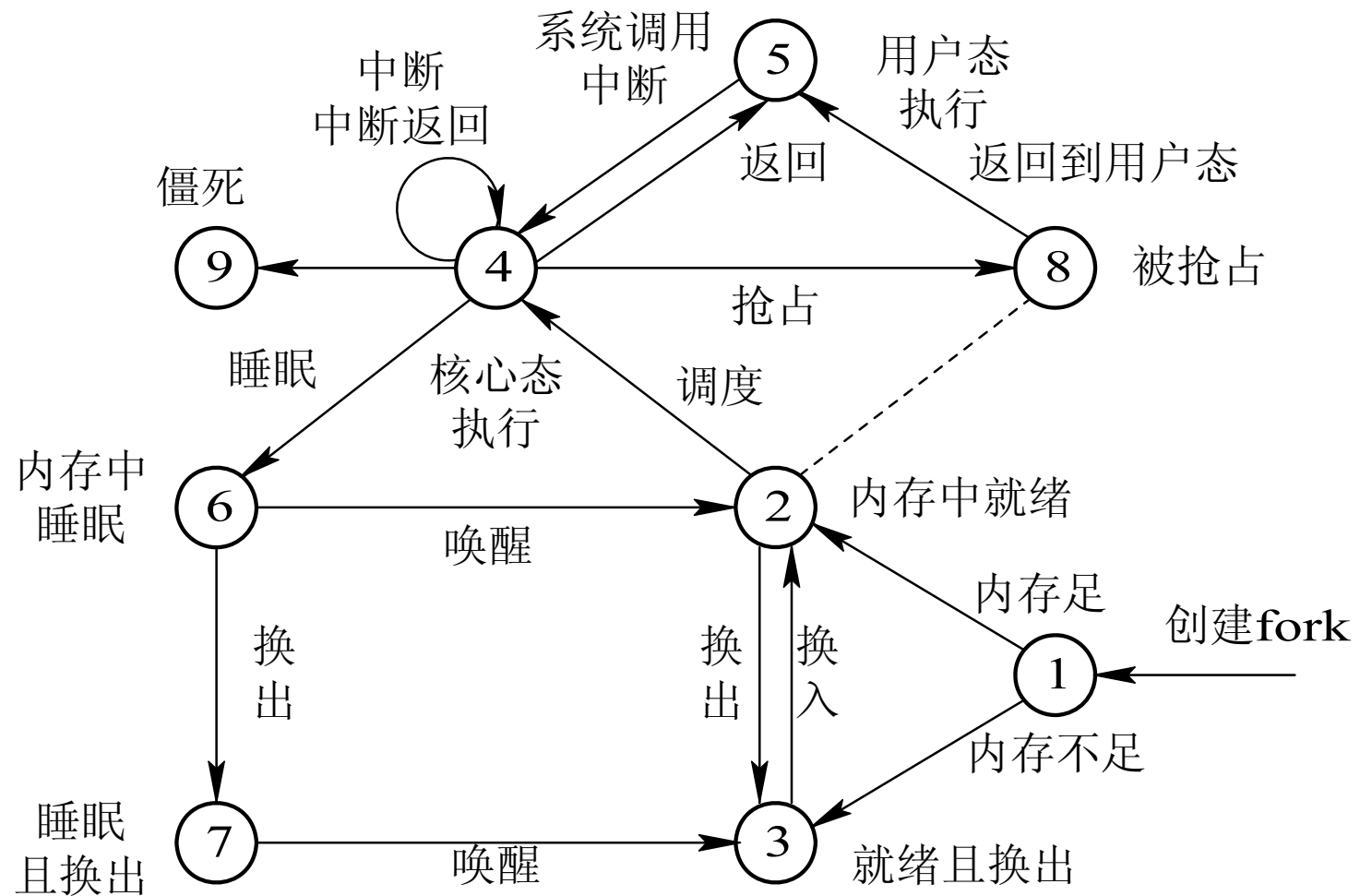
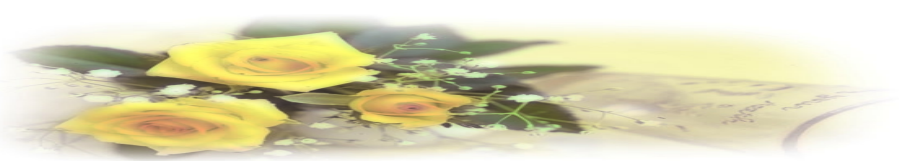


图10-4 进程的状态转换



第十章 UNIX系统内核结构

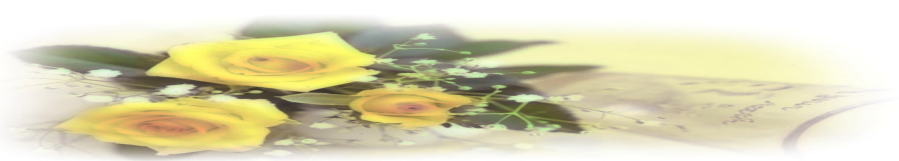
(1) 执行状态。这表示进程已获得处理机而正在执行。UNIX系统把执行状态又进一步分为两种:一种是用户态执行,表示进程正处于用户状态中执行;另一种是核心态执行,表示一个应用进程执行系统调用后,或I/O中断、时钟中断后,进程便处于核心态执行。这两种状态的主要差别在于:处于用户态执行时,进程所能访问的内存空间和对象受到限制,其所占有的处理机是可被抢占的;而处于核心态执行中的进程,则能访问所有的内存空间和对象,且所占用的处理机是不允许被抢占的。





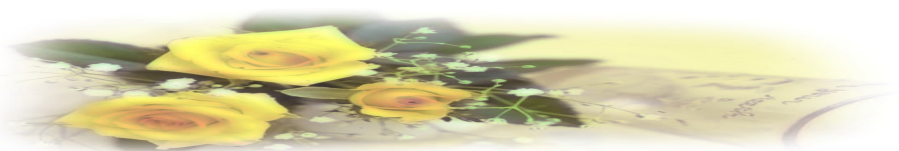
第十章 UNIX系统内核结构

(2) 就绪状态。这是指进程处于一种只需再获得处理机便可执行的状态。由于UNIX内核提供了对换功能，因而又可把就绪状态分为“内存中就绪”和“就绪且换出”两种状态。当调度程序调度到“内存中就绪”状态的进程时，该进程便可立即执行；而调度到“就绪且换出”状态的进程时，则须先将该进程映像全部调入内存后，再使其执行。





(3) 睡眠状态。使一个进程由执行状态转换为睡眠状态的原因有许多，如因进程请求使用某系统资源而未能得到满足时；又如进程使用了系统调用wait()后，便主动暂停自己的执行，以等待某事件的出现等。在UNIX中把睡眠原因分为64种，相应地，最多也可设置64个睡眠队列。同样，由于对换功能的原因又可将睡眠状态分为“内存睡眠”状态和“睡眠且换出”状态。当内存紧张时，在内存中睡眠的进程可被内核换出到外存上，相应地，此时进程的状态便由“内存睡眠”状态转换为“睡眠且换出”状态。





第十章 UNIX系统内核结构

(4) “创建”与“僵死”状态。创建状态是指利用fork系统调用来创建子进程时，被创建的新进程所处的状态；僵死状态是在进程执行了exit系统调用后所处的状态，此时该进程实际上已不存在，但还留下一些信息供父进程搜集。

(5) “被抢占”状态，也称为“被剥夺状态”。当正在核心态执行的进程要从核心态返回到用户态执行时，如果此时已有一优先级更高的进程在等待处理机，则此时内核可以抢占(剥夺)已分配给正在执行进程的处理机，去调度该优先级更高的进程执行。这时，被抢占了处理机的进程便转换为“被抢占”状态。处于“被抢占”状态的进程与处于“内存中就绪”状态的进程是等效的，它们都被排列在同一就绪队列中等待再次被调度。

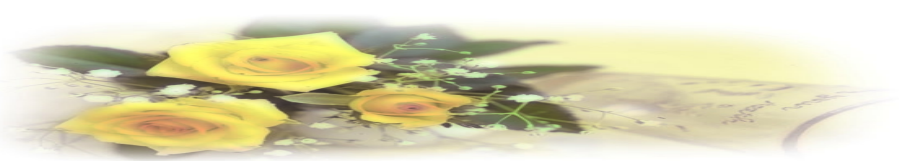




2. 进程映像

1) 用户级上下文

用户级上下文的主要成分为用户程序。它在系统中可分为正文区和数据区两部分。正文区是只读的，它主要包括一些程序指令。进程在执行时，可利用用户栈区来保存过程调用时的传送参数和返回值。共享存储区是一个能与其它进程共享的数据区。存储区中的数据可由有权共享该存储区的进程所共享。





2) 寄存器上下文

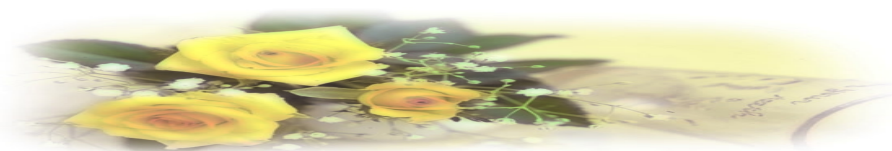
寄存器上下文主要是由CPU中的一些寄存器的内容所组成的。主要的寄存器有下述几种。

(1) 程序寄存器。在其中存放的是CPU要执行的下一条指令的虚地址。

(2) 处理机状态寄存器(PSR)。其中包括运行方式(用户态或核心态)、处理机当前的运行级以及记录处理机与该进程有关的硬件状态信息,如产生进位和溢出等。

(3) 栈指针。该指针指向栈的下一个自由项或栈中最后使用的项(因机器而异)。

(4) 通用寄存器。该寄存器用于存放进程在运行过程中所产生的数据,通用寄存器的数目也因机器而异。



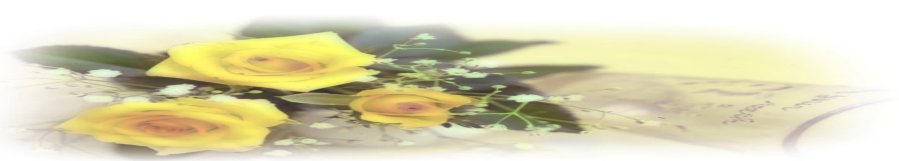


3) 系统级上下文

系统级上下文包括OS为管理该进程所用的信息，可分为静态和动态两部分：

(1) 静态部分。在进程的整个生命期中，系统级上下文的静态部分只有一个，其大小保持不变，又可再进一步把它分成三部分：进程表项、U区及进程区表项、系统区表项和页表。

(2) 动态部分。在整个进程的生命期中，系统级上下文动态部分的大小是可变的，它包括：① 核心栈，这是进程在核心态时使用的栈；② 若干层寄存器上下文，其中，每一层都保存了前一层的上下文。

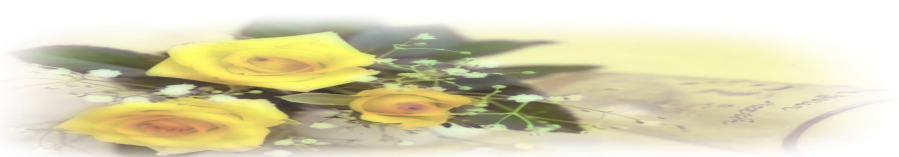




10.2.3 进程控制

1. fork系统调用

在UNIX的内核中设置了一个0进程，它是惟一一个在系统引导时被创建的进程。在系统初启时，由0进程再创建1进程，以后0进程变为对换进程，1进程成为系统中的始祖进程。UNIX利用fork为每个终端创建一个子进程为用户服务，如等待用户登录、执行shell命令解释程序等。每个终端进程又可利用fork来创建自己的子进程，如此下去可以形成一棵进程树。因此说，系统中除0进程外的所有进程都是用fork创建的。



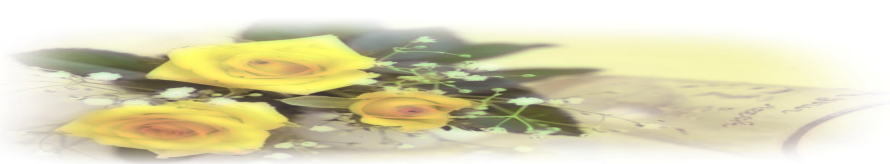


第十章 UNIX系统内核结构

fork系统调用如果执行成功，便可创建一个子进程，子进程继承父进程的许多特性，并具有与父进程完全相同的用户级上下文。核心需为fork完成下列操作：

(1) 为新进程分配一个进程表项和进程标识符。进入fork后，核心检查系统是否有足够的资源以创建一个新进程。若资源不足，fork调用必失败；否则，核心为新进程分配一个进程表项和惟一的进程标识符。

(2) 检查同时运行的进程数目。对于普通用户，是否能为之建立进程要受事先设定的、系统允许同时存在的进程数目的限制，超过此限制值时，fork系统调用失败。

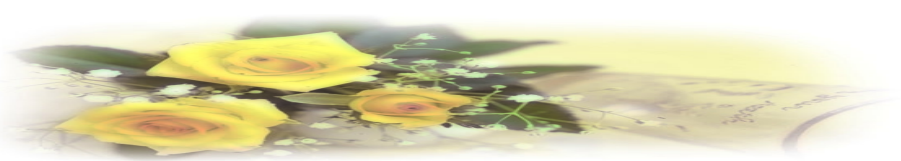




第十章 UNIX系统内核结构

(3) 拷贝进程表项中的数据。将父进程表项中的数据拷贝到子进程的进程表项中，并置进程的状态为“创建”状态。

(4) 子进程继承父进程的所有文件。对父进程的当前目录和所有已打开文件的文件表项中的引用计数f.count做加1操作，表示这些文件又增加了一个访问者，详见10.6节。

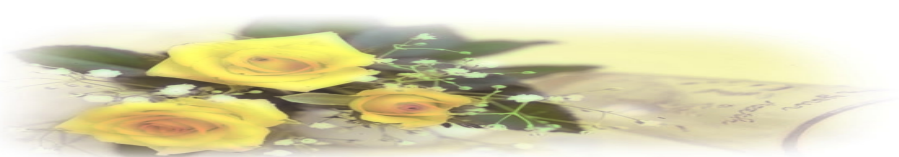




第十章 VIX系统内核结构

(5) 为子进程创建进程上下文。首先，由核心为子进程创建进程上下文的静态部分，并将其父进程上下文的静态部分拷贝到子进程的上下文中。然后，再为子进程创建进程上下文的动态部分。至此，进程的创建即告结束，再置子进程的状态为“内存中就绪”。最后，返回该子进程的标识符。

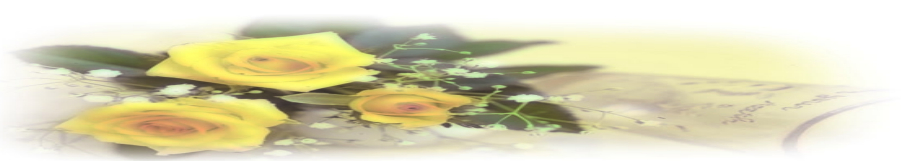
(6) 子进程执行。当子进程被调度执行时，将U区的计时字段初始化后返回0。





2. exec系统调用

在UNIX系统中，当利用fork系统调用创建一个新进程时，只是将父进程的用户级上下文拷贝到新建的子进程中。而UNIX系统又提供了一组系统调用exec，用于将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上，以更新新进程的用户级上下文。UNIX所提供的这一组exec系统调用，它们的基本功能相同，只是它们须各自以不同的方式提供参数，且参数各异。一种方式是直接给出指向参数的指针，另一种方式则是给出指向参数表的指针。



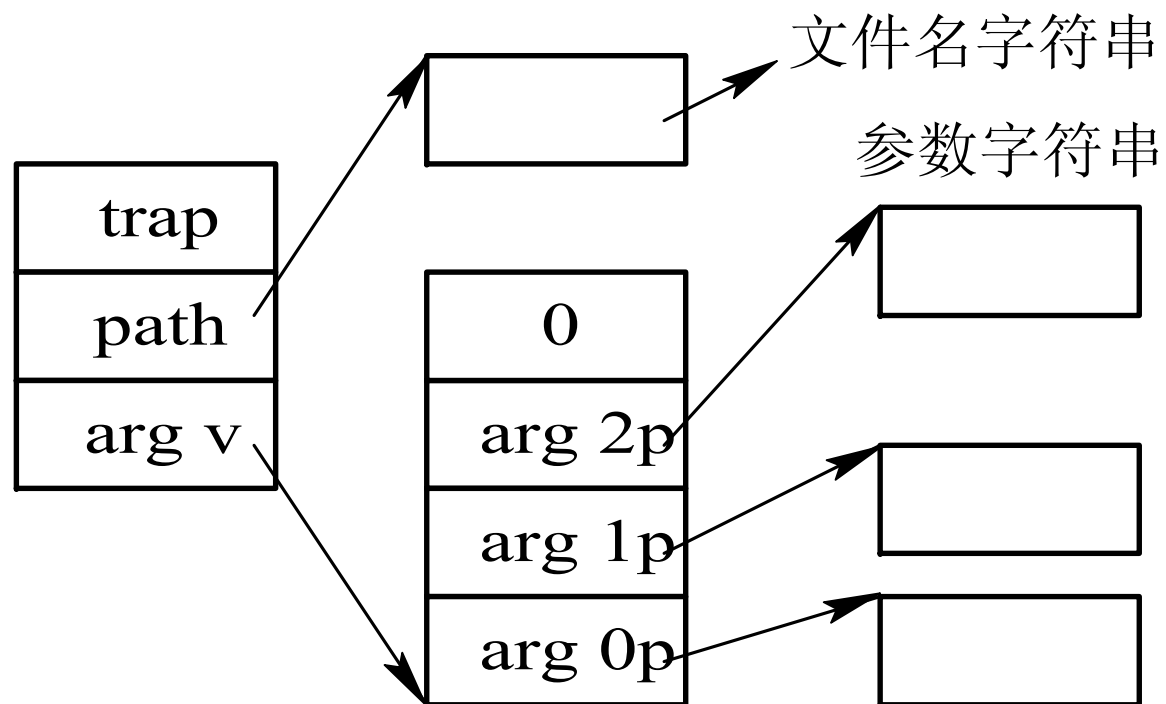
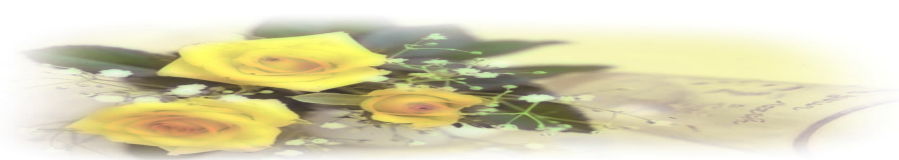


图10-5 exec V 的参数组织方式

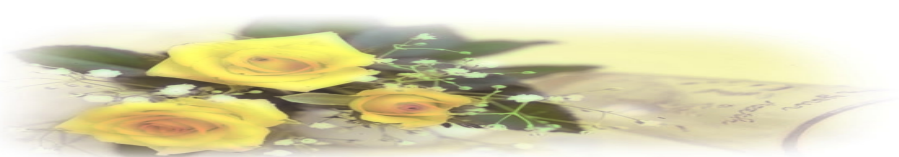




第十章 UNIX系统内核结构

(1) 对可执行文件进行检查。先调用检索目录的过程 `namei`，以沿着目录树去获得指名可执行文件的索引结点。由于在索引结点中存放了相应文件的全部属性，因而可以从中得知该文件是否是可执行的，用户是否拥有执行的许可(权)。若可执行，便再将新的 `exec` V 参数拷贝到一个临时缓冲区，以腾出参数占用的空间，然后将这些空间释放。

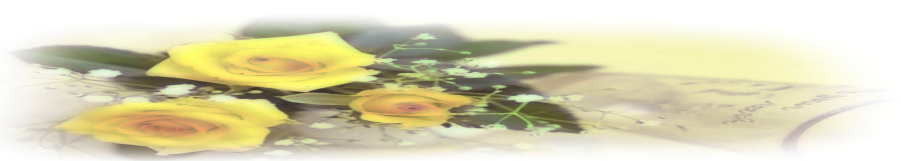
(2) 回收内存空间。对原来与进程连接的各个区，逐个断开其连接，并收回它们所占用的内存空间。





(3) 分配存储空间。根据可执行文件头中的信息(包括文件中的所有正文段和数据段的大小、进程执行时的起始地址等)，为每个段分配新区，并将它们连接到进程上。若有足够的内存空间，可将这些新区装入内存。

(4) 参数拷贝。将exec参数由临时缓冲区拷贝到新的用户栈区，并设置用户态寄存器上下文，如用户栈指针、程序计数器等。

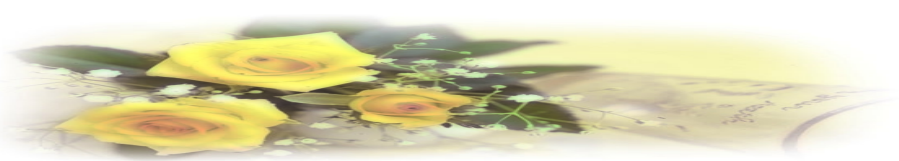




3. `exit`系统调用

为了及时回收进程所占用的资源，对于一般的用户进程，在其任务完成后应尽快撤消该进程。UNIX内核利用`exit`来实现进程的自我终止。通常，父进程在创建子进程时，应在子进程的末尾安排一条`exit`，使子进程能自我终止。内核需为`exit`完成以下操作：

(1) 关闭软中断。由于进程即将终止而不再处理任何软中断信号，故应将软中断处理函数关闭。如果终止的进程是与某一终端相联系的进程组组长，则还须向本组中的各进程发送“挂起”软中断信号，并将它们交给进程1处理。



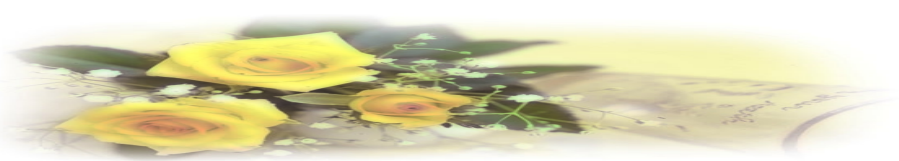


第十章 UNIX系统内核结构

(2) 回收资源。关闭所有已打开的文件，释放进程所有的区及相应的内存，释放当前目录及修改根目录的索引结点。

(3) 写记账信息。将进程在运行过程中所产生的记账数据(其中含有进程运行时的各种统计数据)记录到一个全局记账文件中。

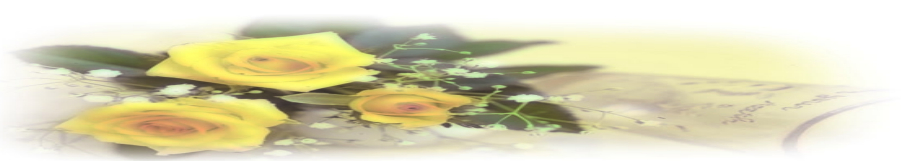
(4) 置进程为“僵死”状态。在做完上述处理后，便可将进程置为“僵死”状态，执行exit系统调用的子进程还应向父进程发送“子进程死”的软中断信号，最后由内核进行上下文切换，调度另一进程执行。





4. wait系统调用

wait系统调用用于将调用进程挂起，直至其子进程因暂停或终止而发来软中断信号为止。如果在wait调用前已有子进程暂停或终止，则调用进程做适当处理后便返回。核心对wait调用做以下处理：核心查找调用进程是否还有子进程，若无，便返回出错码；如果找到一个处于“僵死”状态的子进程，便将子进程的执行时间加到其父进程的执行时间上，并释放该子进程的进程表项；如果未找到处于“僵死”状态的子进程，则调用进程便在可被中断的优先级上睡眠，等待其子进程发来软中断信号时被唤醒。

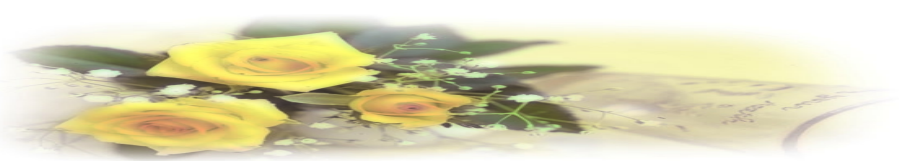




10.2.4 进程调度与切换

1. 引起进程调度的原因

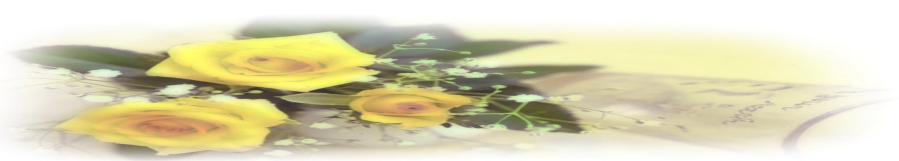
首先，由于UNIX系统是分时系统，因而其时钟中断处理程序须每隔一定时间便对要求进程调度程序进行调度的标志runrun予以置位，以引起调度程序重新调度。其次，当进程执行了wait、exit及sleep等系统调用后要放弃处理机时，也会引起调度程序重新进行调度。此外，当进程执行完系统调用功能而从核心态返回到用户态时，如果系统中又出现了更高优先级的进程在等待处理机时，内核应抢占当前进程的处理机，这也会引起调度。





2. 调度算法

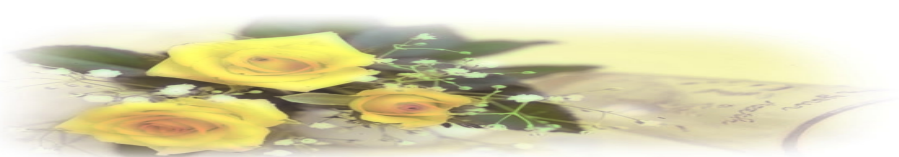
采用动态优先数轮转调度算法进行进程调度。调度程序在进行调度时，首先从处于“内存就绪”或“被抢占”状态的进程中选择一个其优先数最小(优先级最高)的进程。若此时系统中(同时)有多个进程都具有相同的最高优先级，则内核将选择其中处于就绪状态或被抢占状态最久的进程，将它从其所在队列中移出，并进行进程上下文的切换，恢复其运行。





3. 进程优先级的分类

UNIX系统把进程的优先级分成两类，第一类是核心优先级，又可进一步把它分为可中断和不可中断两种。当一个软中断信号到达时，若有进程正在可中断优先级上睡眠，该进程将立即被唤醒；若有进程处于不可中断优先级上，则该进程继续睡眠。诸如“对换”、“等待磁盘I/O”、“等待缓冲区”等几个优先级，都属于不可中断优先级；而“等待输入”、“等待终端输出”、“等待子进程退出”几个优先级，都是可中断优先级。另一类是用户优先级，它又被分成 $n+1$ 级，其中第0级为最高优先级，第 n 级的优先级最低。



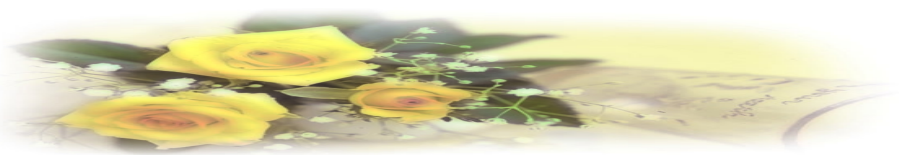


4. 进程优先数的计算

在进程调度算法中，非常重要的部分是如何计算进程的优先数。在系统V中，进程优先数的计算公式可表示如下：

$$\text{优先数} = \frac{\text{最近使用CPU的时间}}{2} + \text{基本用优先数}$$

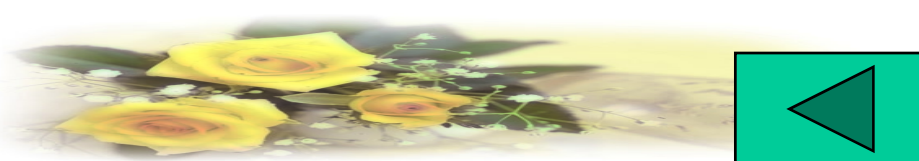
其中，基本用户优先数即proc结构中的偏移值nice，可由用户将它设置成0~40中的任一个数。一旦设定后，用户仅能使其值增加，特权用户才有权减小nice的值。而最近使用CPU的时间，则是指当前占有处理机的进程本次使用CPU的时间。内核每隔16.667 ms便对该时间做加1操作，这样，占有CPU的进程其优先数将会随着它占有CPU时间的增加而加大，相应地，其优先级便随之降低。





5. 进程切换

在OS中，凡要进行中断处理和执行系统调用时，都将涉及到进程上下文的保存和恢复问题，此时系统所保存或恢复的上下文都是属于同一个进程的。而在进程调度之后，内核所应执行的是进程上下文的切换，即内核是把当前进程的上下文保存起来，而所恢复的则是进程调度程序所选中的进程的上下文，以使该进程能恢复执行。



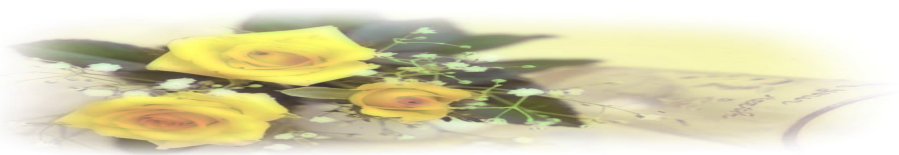


10.3 进程的同步与通信

10.3.1 sleep与wakeup同步机制

1. sleep过程

进入sleep过程后，核心首先保存进入睡眠时的处理机运行级，再提高处理机的运行优先级来屏蔽所有的中断，接着将该进程置为“睡眠”状态，将睡眠地址(对应着某个睡眠事件)保存在进程表项中，并将该进程放入睡眠队列中。如果进程的睡眠是不可中断的，做了进程上下文的切换后，进程便可安稳地睡眠。当进程被唤醒并被调度执行时，将恢复处理机的运行级为进入睡眠时的值，此时允许中断处理机。

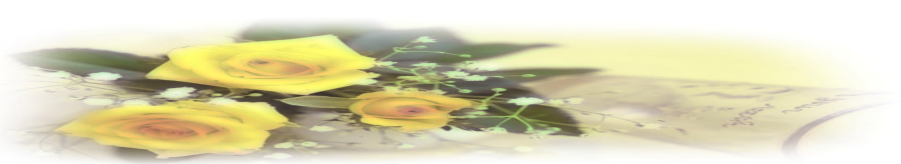




第十章 VMMX系统内核结构

如果进程的睡眠可被中断，但该进程并未收到软中断信号，则在做了进程上下文的切换后也进入睡眠状态。当它被唤醒并被调度执行时，应再次检查是否有待处理的软中断信号；若仍无，则恢复处理机的运行级为进入睡眠时的值，最后返回0。

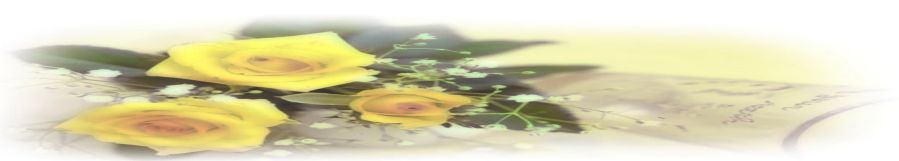
如果在进入睡眠前检测到有软中断信号，那么进程是否还要进入睡眠状态将取决于该进程的优先级。如果它是不可中断的优先级，进程仍进入睡眠，直至被wakeup唤醒；否则，即若进程的优先级为可中断的，则进程便不再进入睡眠，而是响应软中断。





2. wakeup过程

该过程的主要功能是唤醒在指定事件队列上睡眠的所有进程，并将它们放入可被调度的进程队列中。如果进程尚未被装入内存，应唤醒对换进程；如果被唤醒进程的优先级高于当前进程的优先级，则应重置调度标志。最后，在恢复处理机的运行级后返回。

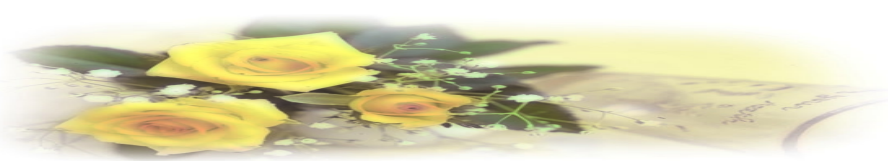




10.3.2 信号机制

1. 信号机制的基本概念

信号机制主要是作为在同一用户的诸进程之间通信的简单工具。信号本身是一个1~19中的某个整数，用来代表某一种事先约定好的简单消息。每个进程在执行时，都要通过信号机制来检查是否有信号到达。若有信号到达，表示某进程已发生了某种异常事件，便立即中断正在执行的进程，转向由该信号(某整数)所指示的处理程序，去完成对所发生的事件(事先约定)的处理。处理完毕，再返回到此前的断点处继续执行。可见，信号机制是对硬中断的一种模拟，故在早期的UNIX版本中又称其为软中断。

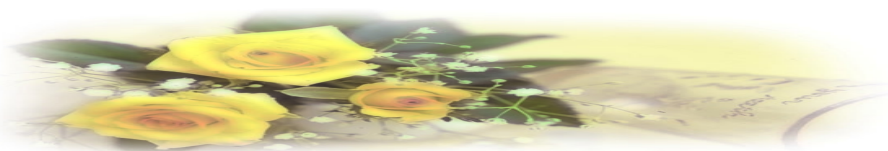




第十章 UNIX系统内核结构

信号机制与中断机制之间的相似之处表现为：信号和中断都同样采用异步通信方式，在检测出有信号或有中断请求时，两者都是暂停正在执行的程序而转去执行相应的处理程序，处理完后都再返回到原来的断点；再有就是两者对信号或中断都可加以屏蔽。

信号与中断两机制之间的差异是：中断有优先级，而信号机制则没有，即所有的信号都是平等的；再者是信号处理程序是在用户态下运行的，而中断处理程序则是在核心态下运行；还有，中断响应是及时的，而对信号的响应通常都有较长的时间延迟。

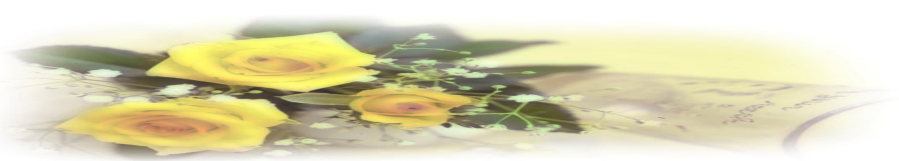




2. 信号机制的功能

1) 发送信号

这是指由发送进程把信号送至指定目标进程的proc结构中信号域的某一位上。如果目标进程正在一个可被中断的优先级上睡眠，核心便将目标进程唤醒，发送过程就此结束。一个进程可能在其信号域中有多个位被置位，代表已有多种类型的信号到达，但对于一类信号，进程却只能记住其中的某一个。进程可利用系统调用kill向另一进程或一组进程发送一个信号。





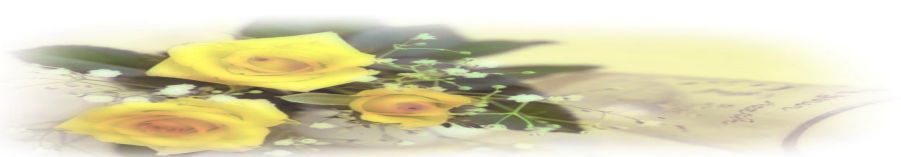
2) 设置对信号的处理方式

在UNIX系统中，可利用系统调用`signal(sig, func)`来预置对信号的处理方式。其中，参数`sig`为信号名，`func`用于预置处理方式，可分成三种情况：

(1) `func=1`时，进程对`sig`类信号不予理睬，亦即屏蔽了该信号。

(2) `func=0`，即为缺省值时，进程在收到`sig`信号后应自我终止。

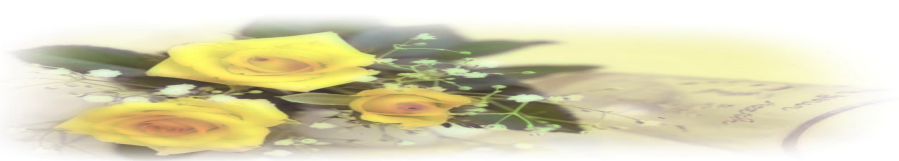
(3) `func`为非0、非1类整数时，就把`func`的值作为指向某信号处理程序的指针。





3) 对信号的处理

当一个进程要进入或退出一个低优先级睡眠状态时，或一个进程即将从核心态返回到用户态时，核心都要检查该进程是否已收到信号。当进程处于核心态时，即使收到信号也不予理睬；仅当进程返回到用户态时，才处理信号。对信号的处理分三种情况进行：当func=1时，进程不做任何处理便返回；当func=0时，进程自我终止；当其值为非0、非1的整数时，系统从核心态转为用户态后，便转向相应信号的处理程序(因为该程序是用户程序，故应运行在用户态)，处理完后，再返回到用户程序的断点。



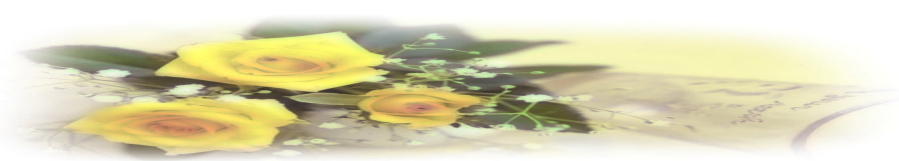


10.3.3 管道机制

1. 管道的类型

1) 无名管道(Unnamed Pipes)

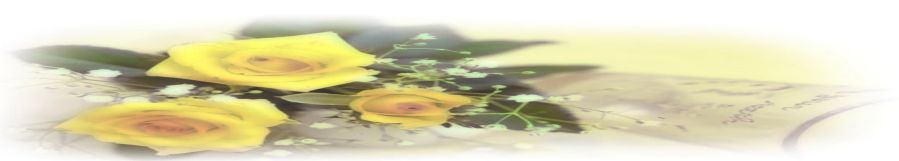
在早期的UNIX版本中，只提供了无名管道。这是一个临时文件，是利用系统调用`pipe()`建立起来的无名文件(指无路径名)。只用该系统调用所返回的文件描述符来标识该文件，因而，只有调用`pipe`的进程及其子孙进程才能识别此文件描述符，从而才能利用该文件(管道)进行通信。当这些进程不再需要此管道时，由核心收回其索引结点。





2) 有名管道(Named Pipes)

为了克服无名管道在使用上的局限性，以便让更多的进程能够利用管道进行通信，在UNIX系统中又增加了有名管道。有名管道是利用mknod系统调用建立的、可以在文件系统中长期存在的、具有路径名的文件，因而其它进程可以感知它的存在，并能利用该路径名来访问该文件。对有名管道的访问方式像访问其它文件一样，都需先用Open系统调用将它打开。不论是有名管道、还是无名管道，对它们的读写方式是相同的。

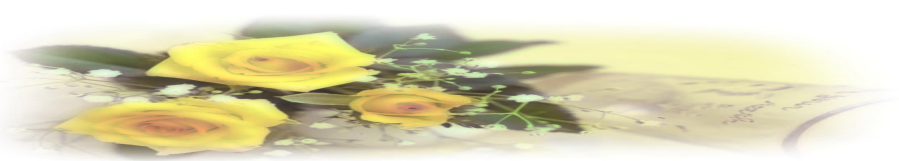




2. 对无名管道的读写

1) 对pipe文件大小的限制

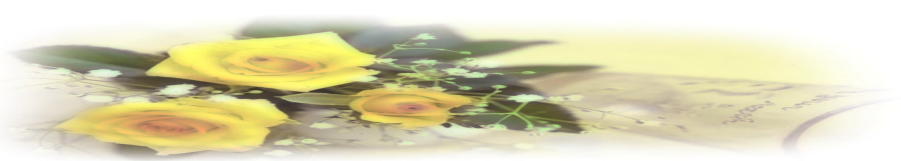
为了提高进程的运行效率，pipe文件只使用索引结点中的直接地址i-addr(0)~i-addr(9)。如果每个盘块的大小为4 KB，则pipe文件的最大长度被限制在40 KB之内。核心将索引结点中的直接地址项作为一个循环队列来管理，为它设置一个读指针和一个写指针，按先进先出顺序读写。





2) 进程互斥

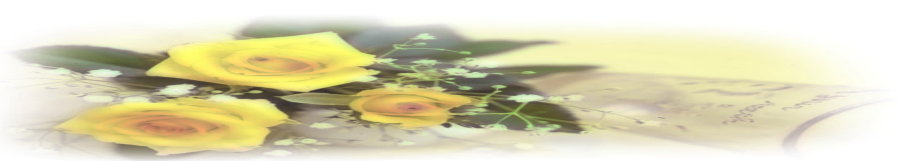
为使读、写进程互斥地访问pipe文件，只须使诸进程互斥地访问pipe文件索引结点中的直接地址项。因此，每逢进程在访问pipe文件前，都须先检查该索引结点是否已经上锁。若已锁住，进程便睡眠等待；否则，将索引结点上锁，然后再执行读、写操作。操作结束后又将该索引结点解锁，并唤醒因该索引结点上锁而睡眠的进程。





3) 进程写管道

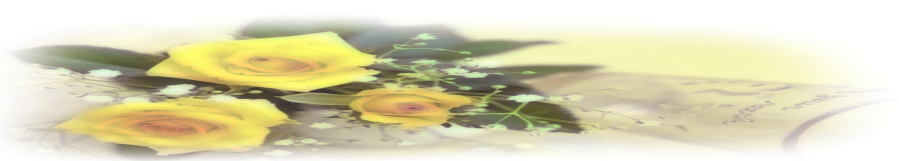
当进程向管道写数据时，可能有以下两种情况：如果管道中有足够的空间能存放要写的数据，在每写完一(盘)块后，核心便自动增加地址项的大小。当写完i-addr(9)地址项中所指示的盘块时，便又向i-addr(0)地址项所指示的盘块中写数据，全部写完后，核心修改索引结点中的写指针，并唤醒因该管道空而睡眠等待的进程。如果管道中无足够的空间来存放要写入的数据，核心将对该索引结点做出标志，然后让写进程睡眠等待，直到读进程将数据从管道中读出后，才唤醒等待写进程





4) 进程读管道

当进程从管道中读数据时，也同样会有两种可能：如果管道中已有足够的数据供进程读，读进程便根据读指针的初始值去读数据。每读出一块后，便增加地址项的大小。读完时，核心修改索引结点中的读指针，并唤醒所有等待的写进程。如果进程所要读的数据比管道中的数据多，则可令读进程把管道中已有数据读完后，暂时进入睡眠状态等待，直至写进程又将数据写入管道后，再将读进程唤醒。



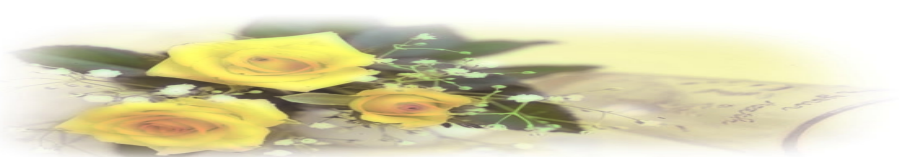


10.3.4 消息机制

1. 消息和消息队列

1) 消息(message)

消息是一个格式化的、可变长度的信息单元。消息机制允许进程发送一个消息给任何其它进程。由于消息的长度是可变的，因而为便于管理而把消息分为消息首部和消息数据区两部分。在消息首部中，记录了消息的类型和大小且指向消息数据区的指针以及消息队列的链接指针等是定长的。在图10-6中示出了消息队列i中的三个消息首部msg0、msg3和msg2。它们分别利用一个指向缓冲区(数据区)的指针指出消息的位置。



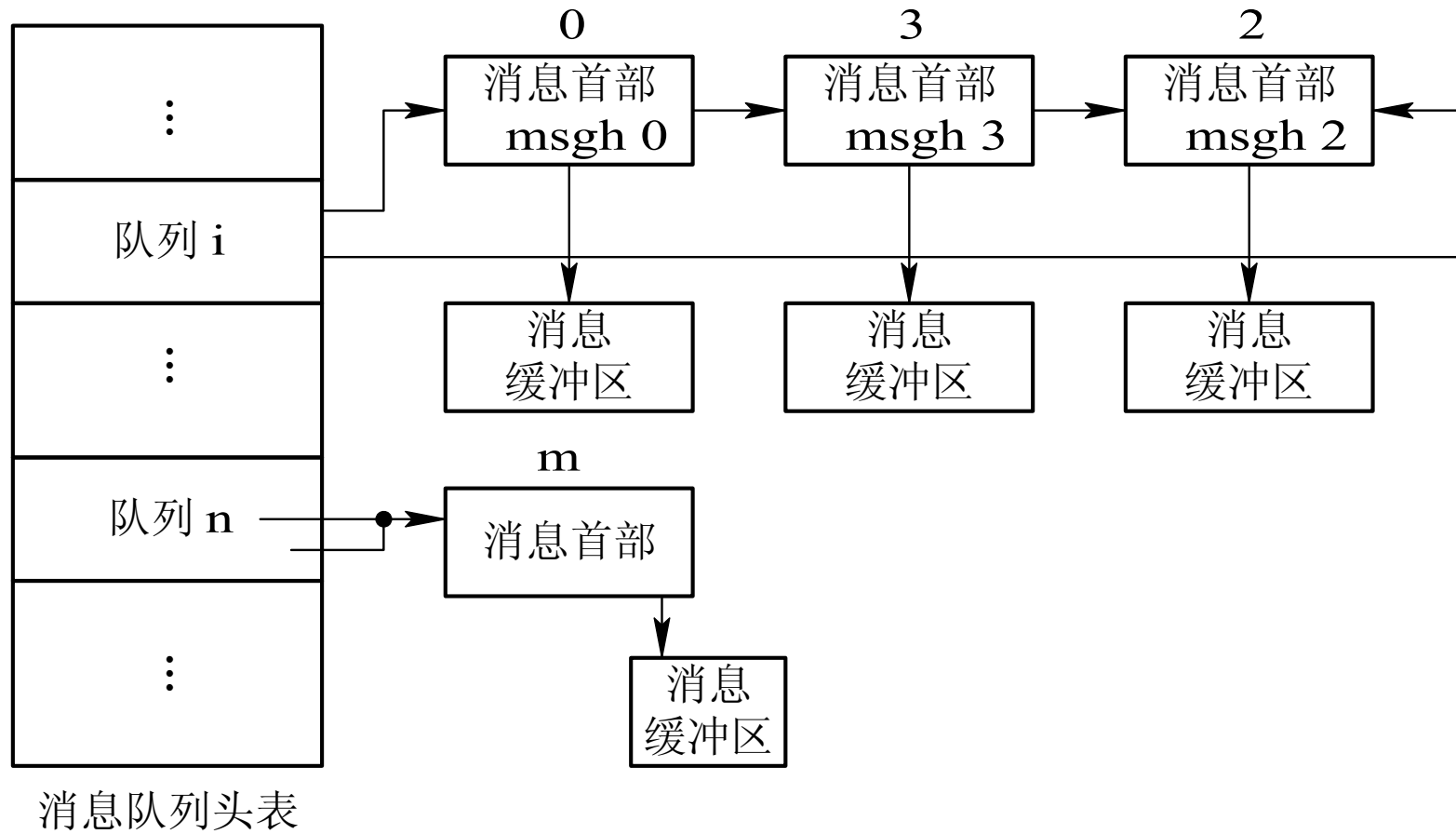
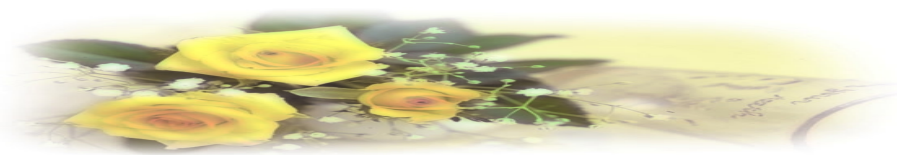


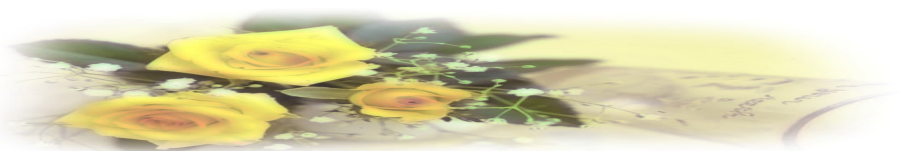
图10-6 消息机制中的数据结构





2) 消息队列

当一个进程收到由其它多个进程发来的消息时，可将这些消息排成一个消息队列，每个消息队列有一个称为关键字key的名称，它是由用户指定的。每个消息队列还有一个消息队列描述符，其作用与用户文件描述符一样，以方便用户和系统对消息队列的访问。在一个系统中可能有若干个消息队列，由所有的消息队列的头标组成一个头标数组。图10-6示出了消息和消息队列的数据结构。

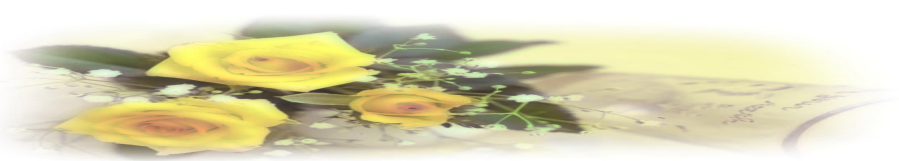




2. 消息队列的建立与操作

1) 消息队列的建立

在一个进程要利用消息机制与其它进程通信之前，应利用系统调用msgget()先建立一个指名的消息队列。对于该系统调用，核心将搜索消息队列头标表，确定是否有指定名字的消息队列。若无，核心将分配一个新的消息队列头标，并对它进行初始化，然后给用户返回一个消息队列描述符；否则，它只是检查该消息队列的许可权后便返回。



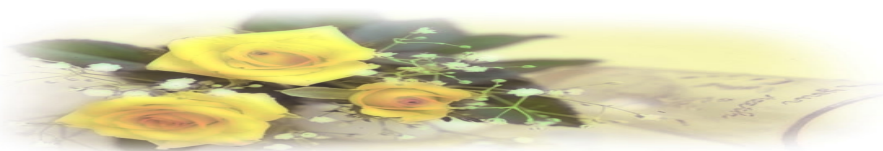


2) 消息队列的操纵

用户可利用msgctl()系统调用对指定的消息队列进行操纵。在该系统调用中包括三个参数，其中，msgid为消息队列描述符；cmd是规定的命令；buf是用户缓冲区地址，用户可在此缓冲区中存放控制参数和查询结果。命令可分为三类：（1）用于查询有关消息队列的情况的命令，如查询队列中的消息数目、队列中的最大字节数、最后一个发送消息的进程的标识符、发送时间等。

（2）用于设置和改变有关消息队列的属性的命令，如改变消息队列的用户标识符、用户组标识符、消息队列的许可权等。

（3）消除消息队列的标识符。

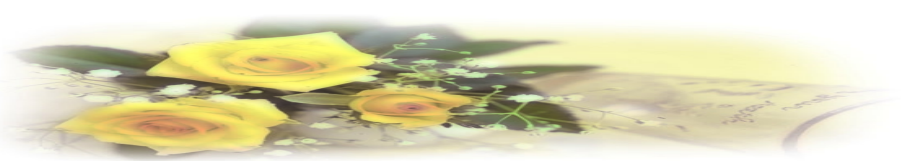




3. 消息的发送和接收

1) 消息的发送

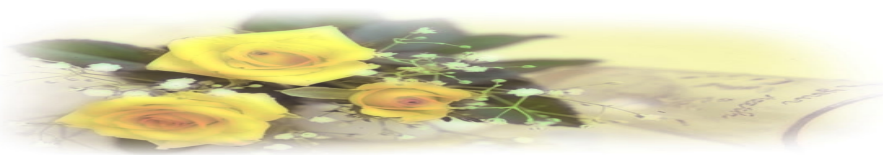
当进程要与其它进程通信时，可利用msgsnd()系统调用来发送消息。对于msgsnd()系统调用，核心检查消息队列描述符和许可权是否合法，消息长度是否超过系统规定的长度。通过检查后，核心为消息分配消息数据区，并将消息从用户消息缓冲区拷贝到消息数据区。分配消息首部，将它链入消息队列的末尾；在消息首部中填写消息的类型、大小以及指向消息数据区的指针等；还要修改消息队列头标中的数据(如消息队列中的消息数、字节数等)。然后，唤醒在等待消息到来的睡眠进程。





2) 消息的接收

进程可利用msgrcv()系统调用，从指定消息队列中读一个消息。对于msgrcv()系统调用，先由核心检查消息队列标识符和许可权，继而根据用户指定的消息类型做相应的处理。消息类型msgtyp的参数可能有三种情况：当msgtyp=0时，核心寻找消息队列中的第一个消息，并将它返回给调用进程；当msgtyp为正整数时，核心返回指定类型的第一个消息；当msgtyp为负整数时，核心应在其类型值小于或等于msgtyp绝对值的所有消息中，选出类型值最低的第一个消息返回。如果所返回消息的大小等于或小于用户的请求，核心便将消息正文拷贝到用户区，再从队列中删除该消息，并唤醒睡眠的发送进程；如果消息长度比用户要求的大，则系统返回出错信息。用户也可忽略对消息大小的限制，此时，核心无需理会消息的大小而一概把消息内容拷贝到用户区。

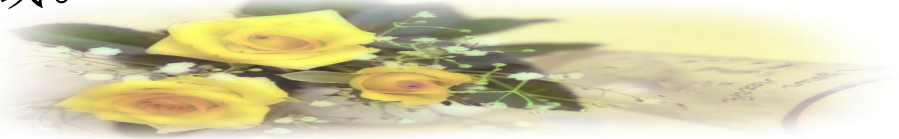




10.3.5 共享存储区机制

1. 共享存储区

共享存储区(Shared Memory)机制是UNIX系统中通信速度最高的一种通信机制。该机制一方面可使若干进程共享主存中的某个区域，且使该区域出现在多个进程的虚地址空间中。另一方面，在一个进程的虚地址空间中又可连接多个共享存储区，每个共享存储区都有自己的名字。当进程间欲利用共享存储区进行通信时，须首先在主存中建立一个共享存储区，然后将该区附接到自己的虚地址空间上。此后，进程之间便可通过对共享存储区中数据的读和写来实现直接通信。图 10-7 中示出了两个进程通过共享一个存储区进行通信的例子。其中，进程A将建立的共享存储区附到自己的AA' 区域，进程B则将它附接到自己的BB' 区域。



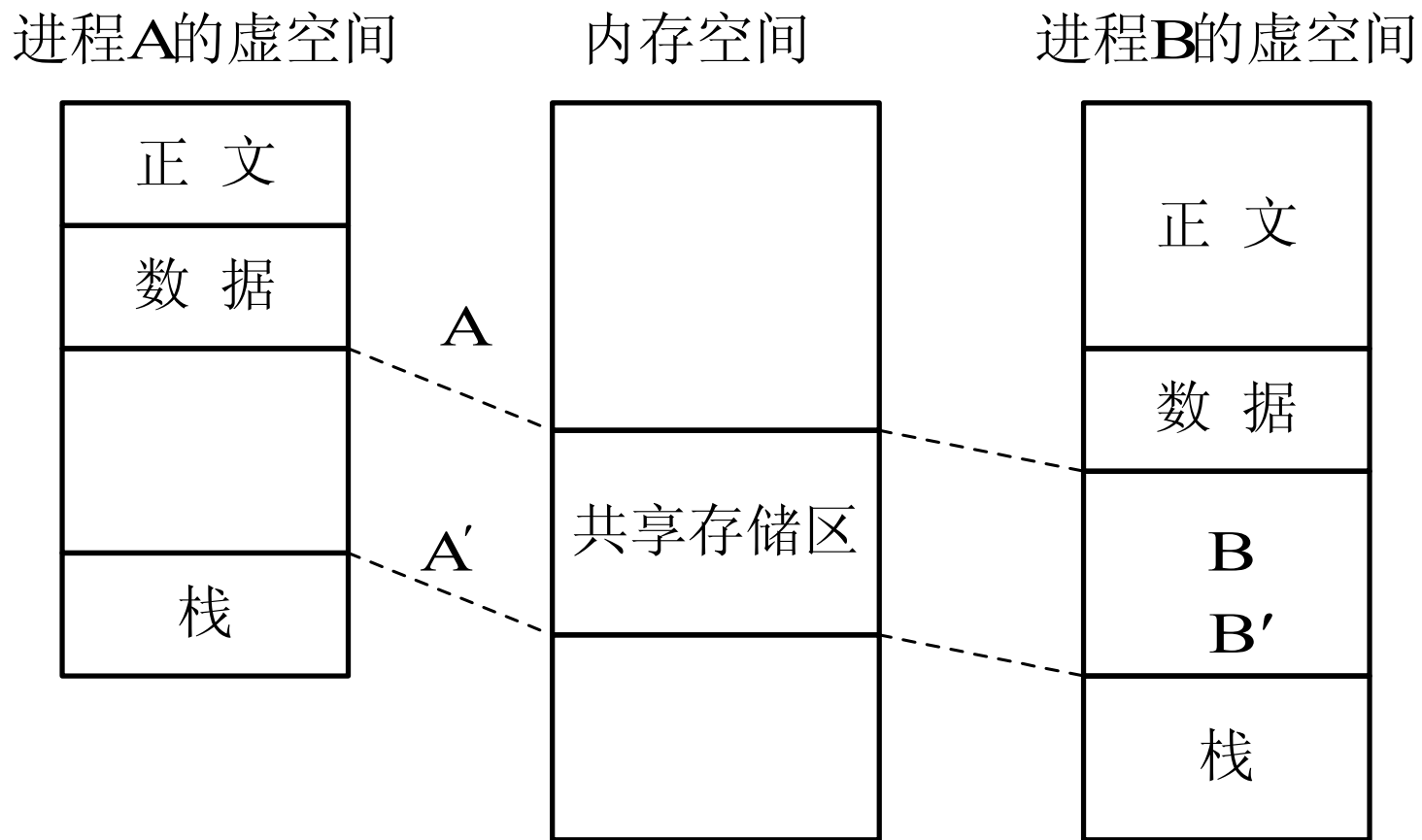
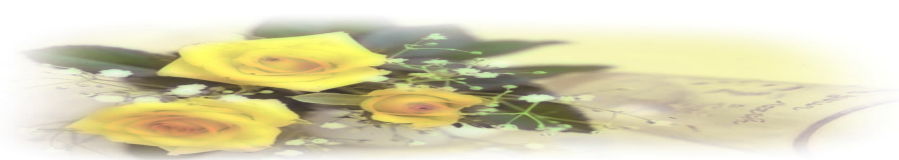


图10-7 利用共享存储区进行通信

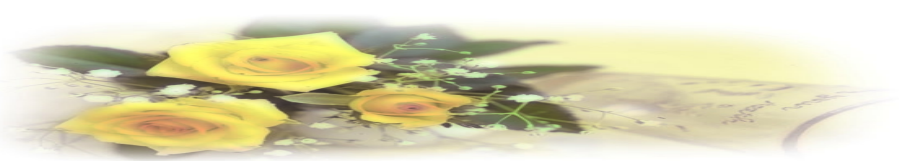




2. 共享存储区的建立与操纵

1) 共享存储区的建立

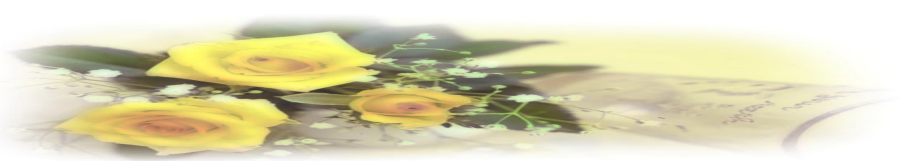
当进程要利用共享存储区与另一进程进行通信时，须先利用系统调用`shmget( )`建立一块共享存储区，并提供该共享存储区的名字`key`和共享存储区以字节为单位的长度`size`等参数。若系统中已经建立了指名的共享存储区，则该系统调用将返回该共享存储区的描述符`shmid`；若尚未建立，便为进程建立一个指定大小的共享存储区。





2) 共享存储区的操纵

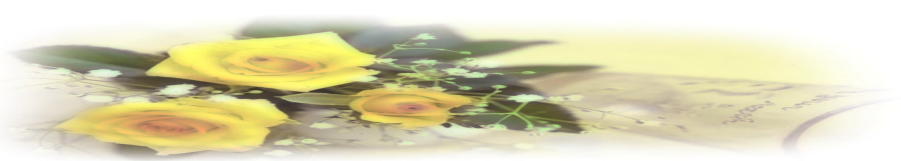
如同消息机制一样，可以用shmctl()系统调用对共享存储区的状态信息进行查询，如其长度、所连接的进程数、创建者标识符等；也可设置或修改其属性，如共享存储区的许可权、当前连接的进程计数等；还可用来对共享存储区加锁或解锁，以及修改共享存储区标识符等。





3. 共享存储区的附接与断开

在进程已经建立了共享存储区或已获得了其描述符后，还须利用系统调用`shmat()`将该共享存储区附接到用户给定的某个进程的虚地址`shmaddr`上，并指定该存储区的访问属性，即指明该区是只读，还是可读可写。此后，该共享存储区便成为该进程虚地址空间的一部分。进程可采取与对其它虚地址空间一样的存取方法来访问。当进程不再需要该共享存储区时，再利用系统调用`shmdt()`把该区与进程断开。



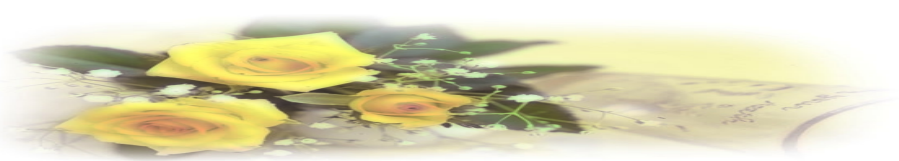


10.3.6 信号量集机制

1. 信号量与信号量集

1) 信号量

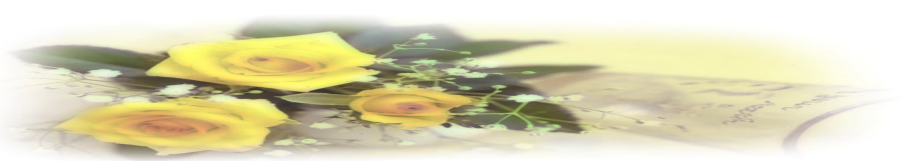
在UNIX系统中规定，每个信号量有一个可用来表示某类资源数目的信号量值和一个操作值，该操作值可为正整数、零或负整数三种情况之一。传统的信号量机构是对信号量施加wait及signal操作。而在UNIX系统中则并未采用wait及signal，而是利用semop()系统调用对指定的信号量施加操作。此外，还可利用semget()来建立信号量及利用semctl()系统调用对信号量进行操纵。





2) 信号量集

在一个信号量集中，通常都包含有若干个信号量。对这组信号量的操作方式应当是原子操作方式，此即，把对这组信号量视为一个整体，要么全做，要么全不做。如果核心不能完成对这组所有信号量的操作，则核心应将已经操作过的信号量恢复到操作前的状态，这样便可实现要么全做、要么全不做的原子操作方式。

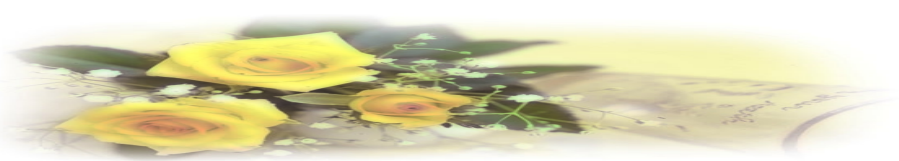




2. 信号量集的数据结构

1) 信号量表

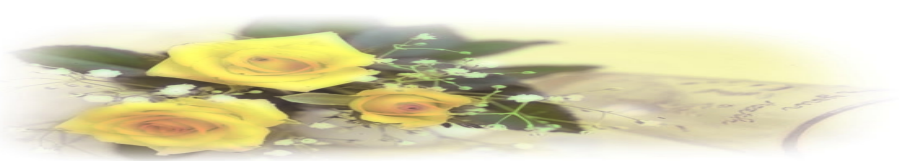
信号量表是信号量的结构数组。在系统V中，每个信号量用一个信号量结构表示。其中，包括信号量值`semval`及最近一次对信号量进行操作的进程标识符`sempid`、等待该信号量值增加的进程数等。





2) 信号量集表

信号量集表是信号量集的索引结构数组，其中的每一个元素都对应于一个信号量集，其内容有：访问权限、指向信号量集中第一个信号量的指针、信号量集中信号量的数目、最近对信号量执行操作的时间。信号量表与信号量集表间的关系示于图10-8中。



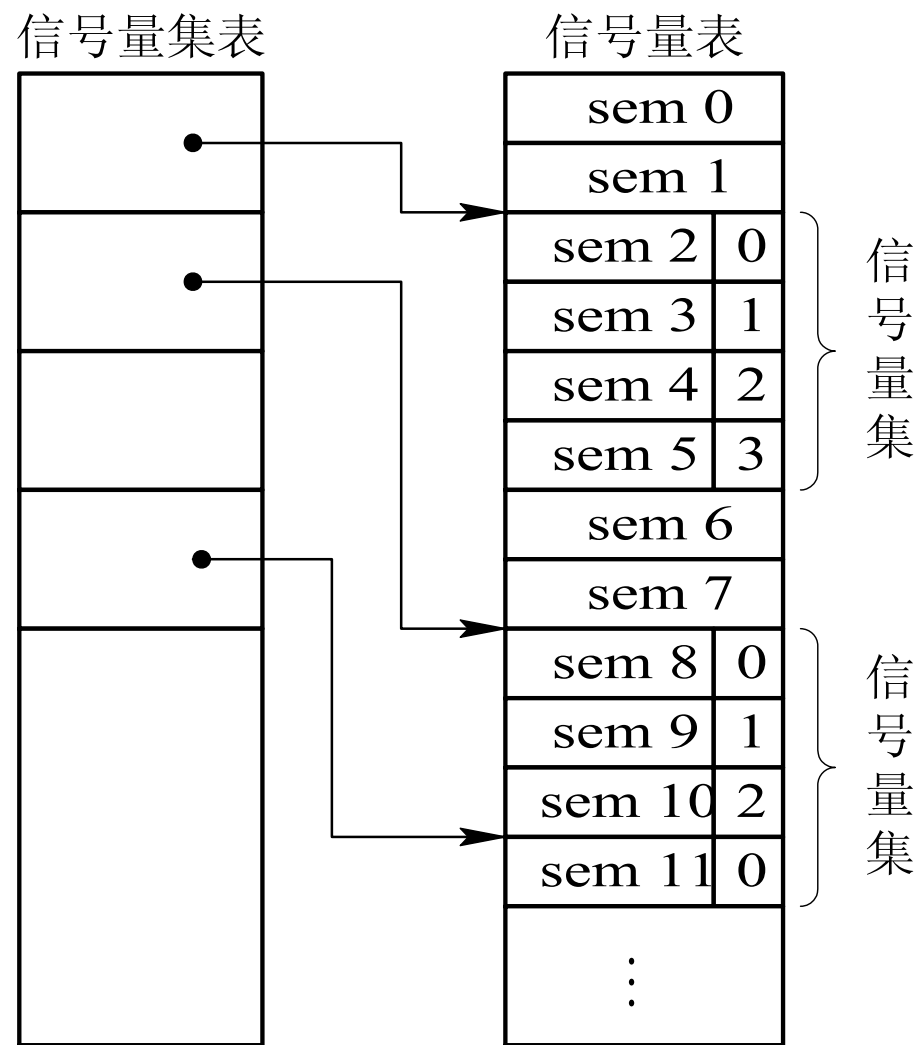


图10-8 信号量集表与信号量表

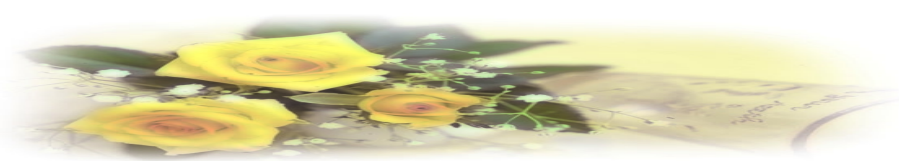


3. 系统调用

在信号量机制中，同样也提供了若干条系统调用，分别用于对信号量执行各种操作。

1) semget()系统调用

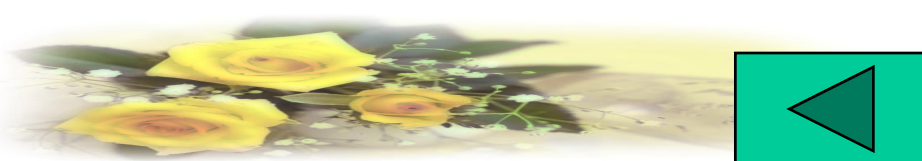
用户可利用该系统调用来建立信号量集。用户应提供信号量的名字、信号量集中信号量的数目等。若信号量集的建立成功，将返回信号量集的描述符semid。





2) semop()系统调用

该系统调用可用于对信号量集进行操作。用户需提供信号量集的描述符、信号量的编号，即信号量在信号量集中的序号，以及所要施加操作的操作数semop。内核根据semop来改变信号量的值。当semop为正值时，便将该正值加到信号量的值上。当semop为负值时，若信号量的值大于semop的绝对值，应将该负值加到信号量值上；否则，操作失败，内核将已经操作过的信号量恢复到该系统调用开始执行时的值。





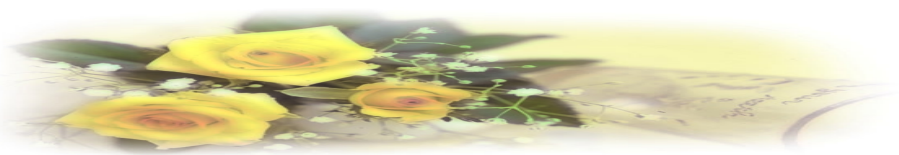
10.4 存储器管理

10.4.1 请求调页管理的数据结构

1. 页表和磁盘描述表

1) 页表

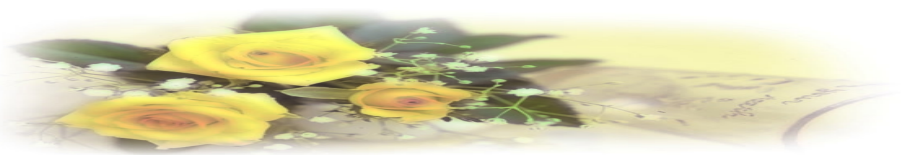
为了实现请求调页策略，UNIX系统V将进程的每个区分为若干个虚页，可把这些虚页分配到不相邻接的页框中，为此而设置了一张页表。在其每个表项中，记录了每个虚页和页框间的对照关系。为了支持请求调页，在页表项中需包含下述若干字段(见图10-9(a)所示):





第十章 VMX系统内核结构

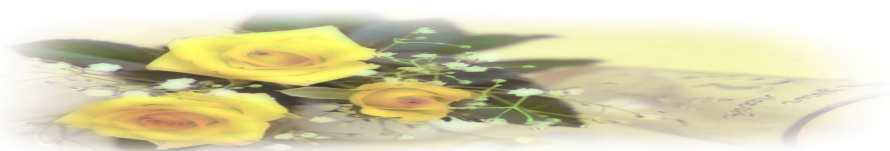
- 页框号：此即第五章中所介绍的、在内存中的物理块号；
- 年龄位：用于指示该页在内存中最近已有多少时间未被访问；
- 访问位：指示该页最近是否被访问过；
- 修改位：指示该页内容是否被修改过，修改位在该页第一次被装入时置为0；
- 有效位：指示该页内容是否有效；
- 写时拷贝 (copy on write) 字段：当有多个进程共享一页时，须设置此字段，用于指示在某共享该页的进程要修改该页时，系统是否已为该页建立了拷贝；
- 保护位：指示此页所允许的访问方式，是只读还是读/写。





2) 磁盘块描述表

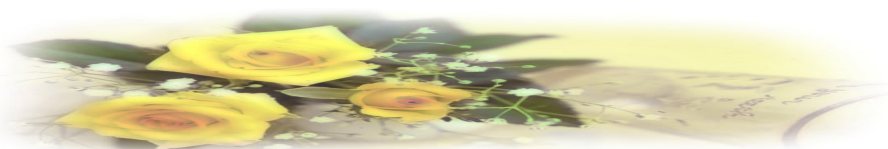
在请求调页机制中，若发现缺页，系统应将所缺页调入内存。但应从何处将其调入内存呢？对于同一个虚页，在不同情况下应从不同的地方调入内存。例如，对于进程中从未执行过的虚页，在第一次调入内存时，应从可执行文件中取出。对于那些已经执行过的虚页，则应从对换区中将之调入内存。为此而引入了磁盘块描述表，用它来记录进程在不同时候的每个虚页在硬盘中的盘块号。这样，当进程在运行中发现缺页时，可通过查找该页表的方法来找到所需调入页面的位置。





第十章 VMX系统内核结构

一个进程的每一页对应一个磁盘描述表项，如图10-9(b)所示。它描述了每一个虚页的磁盘拷贝。其中，包括对换设备号、设备块号和存储器类型。当一个虚页的拷贝在可执行文件的盘块中时，此时在盘块描述表中的存储器类型为File，设备块号是文件的逻辑块号。若虚页的内容已经拷贝到对换设备上，则此时的存储器类型应为Disk。对换设备号和设备块号则用于指示该虚页的拷贝所驻留的逻辑对换设备和相应的盘块号。



第十章 VMIX系统内核结构



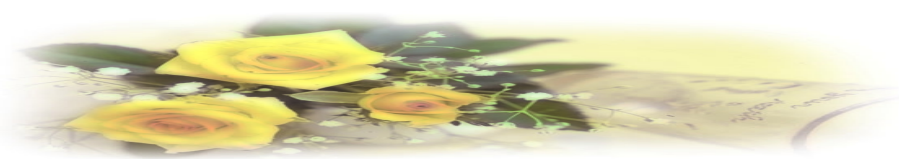
物理块号	年龄位	写时拷贝	修改位	访问位	有效位	保护位

(a) 页表项

对换设备号	设备块号	存储器类型

(b) 盘块说明

图10-9 页表项和磁盘描述表项



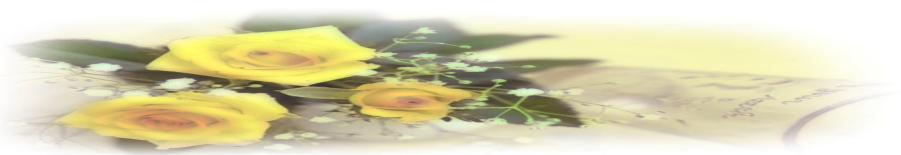


2. 页框数据表和对换使用表

1) 页框数据表

每个页框数据表项描述了内存的一个物理页。每个表项包括有下列各项：

- 页状态：指示该页的拷贝是在对换设备上，还是在可执行文件中。
- 内存引用计数：指出引用该页面的进程数目。
- 逻辑设备：指含有此拷贝的逻辑设备，它可以是对换设备，也可以是文件系统。

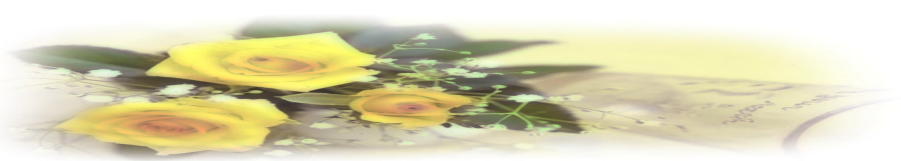




第十章 VMX系统内核结构

- 块号：当逻辑设备为对换设备时，这是盘块号；而当逻辑设备为文件系统时，这指的是文件的逻辑块号。
- 指针1：指向空闲页链表中的下一个页框数据表的指针。
- 指针2：指向散列队列中下一个页框数据表的指针。

系统初启时，核心将所有的页框数据表项链接为一个空闲页链表，形成空闲页缓冲池。为给一个区分配一个物理页，核心从空闲页链表之首摘下一个空闲页表项，修改其对换设备号和块号后，将它放到相应的散列队列中。图10-10示出了页框数据表项的若干散列队列。



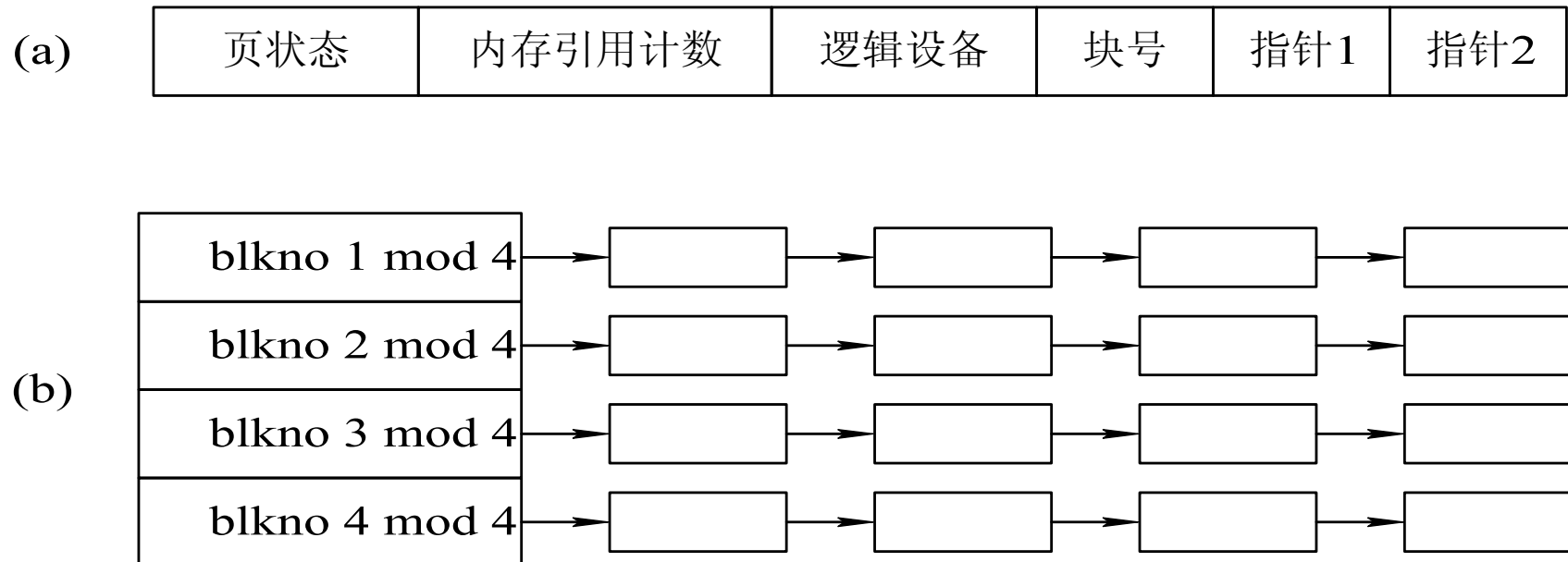
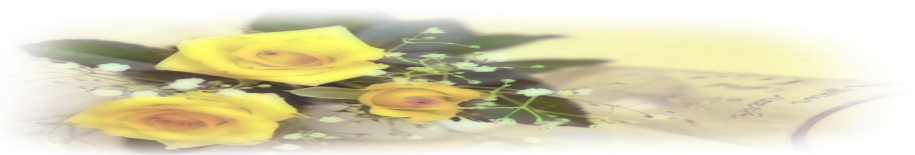


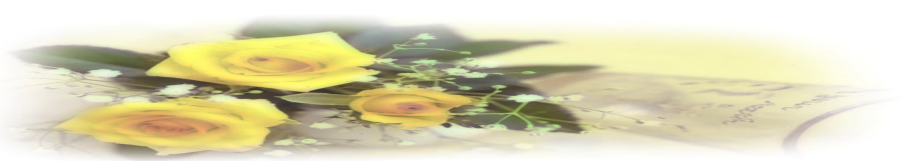
图10-10 页框数据表项及其散列队列





2) 对换使用表

对换设备上的每一页都占有对换使用表的一个表项，表项中含有一个引用计数，其数值表示有多少页表项指向该页。图10-11示出了页表项、磁盘块描述项、页面数据表项和对换使用表项四者间的关系。例如，一个进程的虚地址为1493 K，由页表项可得知其物理页号为794，其拷贝在对换设备1的2743号盘块上。物理页794有一对应的页面数据表项，在该表项中同样指出该页在对换设备1上的2743号盘块中有一拷贝，其引用数为1。



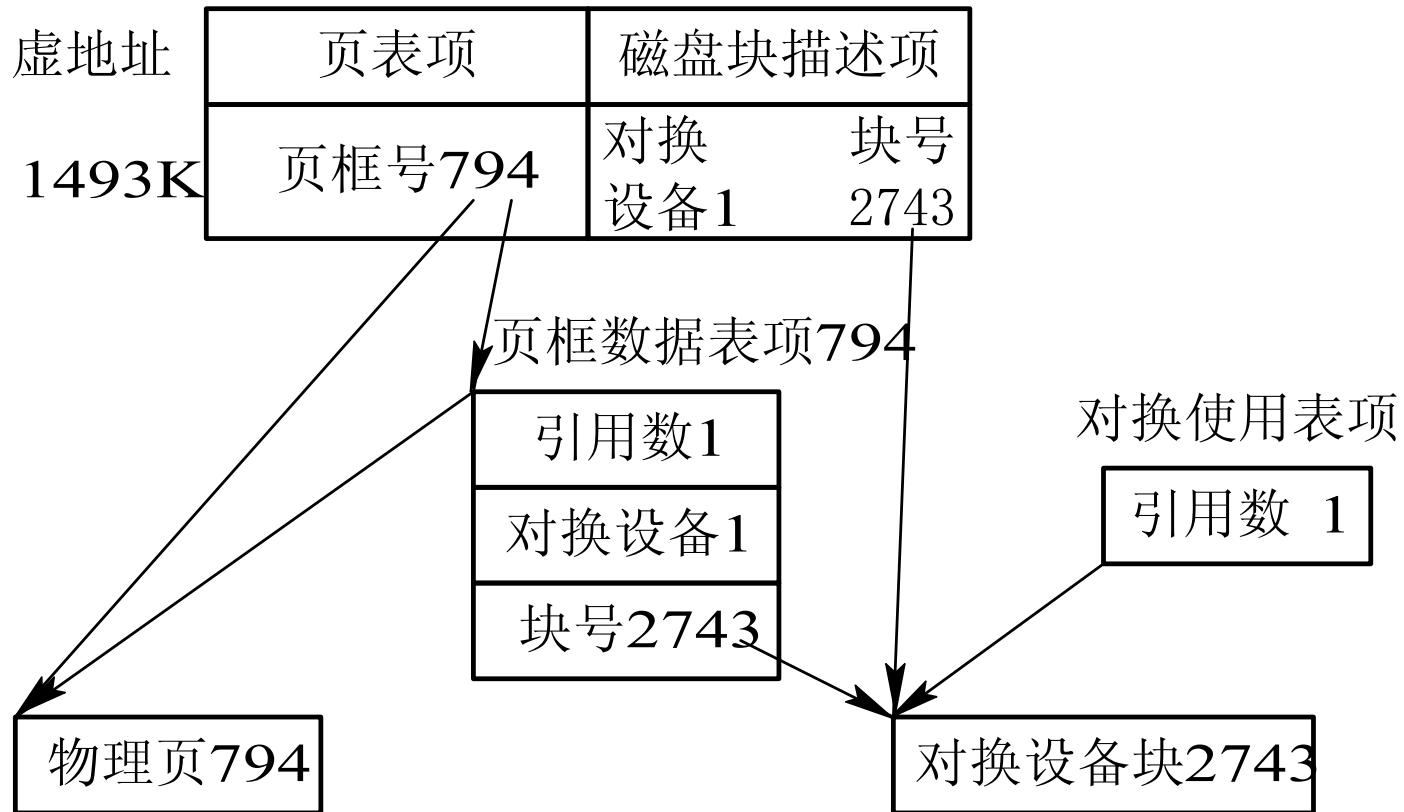
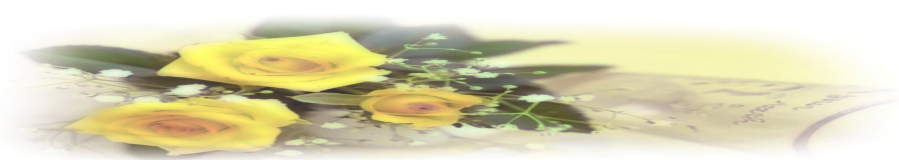


图10-11 四种数据结构之间的关系

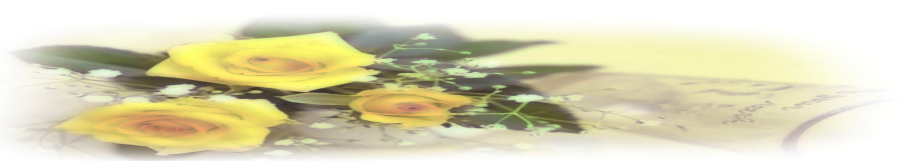




10.4.2 换页进程

1. 增加有效页的年龄

一个页可计数的最大年龄，取决于它的硬件设施。对于只设置两位作为年龄域的页，其有效页的年龄只能取值为0、1、2和3。当该页的年龄为0、1、2时，该页处于不可换出状态；而当其年龄达到3时，该页便为换出状态。每当内存中的空闲页面数低于某规定的低限时，核心便唤醒换页进程，由换页进程去检查内存中的每一个活动的、非上锁的区，对所有有效页的年龄字段加1。对于那些其年龄已增至3的页，便不再加1，而是将它们换出。如果这种页已被进程访问过，便将其年龄域中的年龄降为0。

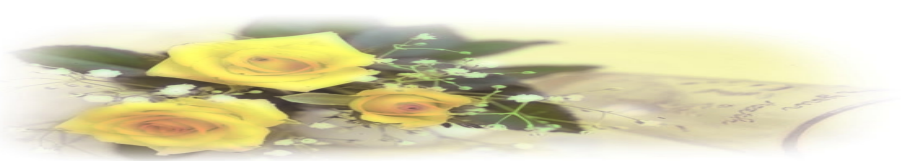




2. 对换出页的几种处理方式

当换出进程从内存的有效页中找到可换出的页后，可采取以下三种方式之一进行处理：

(1) 若在对换设备上已有被换出页的拷贝，且该页的内容未被修改，此时，核心只需将该页页表项中的有效位清零，并将页框数据表项中的引用计数减1，最后将该页框数据表项放入空闲页链表中。

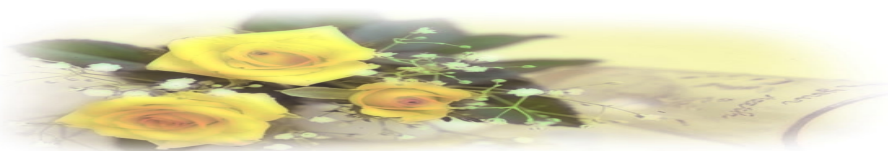




第十章 VMIX系统内核结构

(2) 若在对换设备上没有被换出页的拷贝，则换出进程应将该页写到对换设备上。但为了提高对换效率，对换进程并不是随有随换，而是先将所有要换出的页链入到一个要换出的页面链上。当换出页面链上的页面数达到某一规定值时，比如64个页，核心才真正将这些页面写到对换区中。

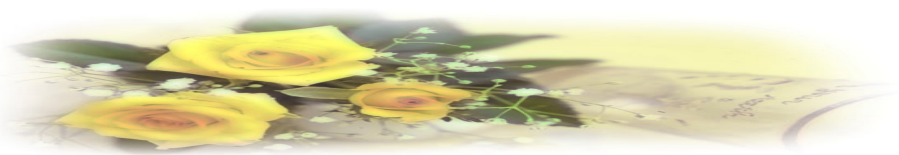
(3) 虽然在对换设备上已有换出页的副本，但该页的内容已被修改过，此时核心应将该页在对换设备上原来占有的空间释放，再重新将该页拷贝到对换设备上，使在对换设备上的拷贝内容总是最新的。





3. 将换出页面写到对换设备上

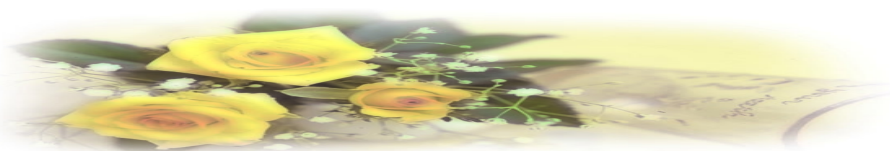
当在换出页面链表中的页面数已达到规定值时，核心应将它们换出。为此，应首先为它们分配一个连续的对换空间，以便一起将它们换出；但如果在对换设备上没有足够大的连续空间，而其空闲存储空间的总和又大于64 KB时，核心可采取每次换出一页的方式将它们换出。每当核心向对换设备上写一页时，须首先清除该页页表项的有效位，并将页框数据表项中的引用计数减1。若引用计数为0，表明已无其它进程再引用该页，核心便将其页框数据表项链入空闲页链表的尾部。若虽引用计数不为0，表明仍有进程共享该页，但如果该页已长期未被访问过，则也须将该页换出。最后，核心将分配给该页的对换空间的地址填入相应的磁盘描述表项中，并将对换使用表中的计数加1。





10.4.3 请求调页

当一个运行进程试图访问一个其有效位为0的页面时，将产生一个有效性错误。这时由核心调用有效性出错处理程序进行处理，可能有下列两种情况：一种情况是，由于在将可执行文件调入内存时，该文件的所有页所在的磁盘块号已记入磁盘块描述表项中，因此，如果在磁盘块描述表项中又找不到所需(缺)的页，则表明此次内存访问非法，核心将向违例进程发送一个“段违例”软中断信号。

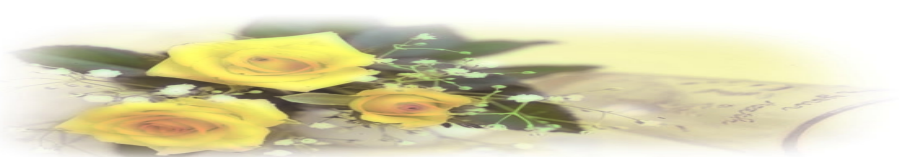




第十章 VMIX系统内核结构

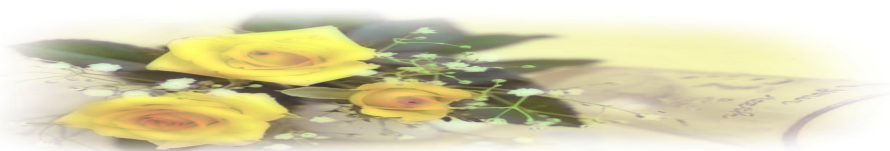
另一种情况是，若在磁盘块描述表项中找到所需的页，表示内存访问合法；但若该页尚未调入内存，则核心为之分配一个页面的内存，将所缺的页调入。至于如何将所缺之页调入内存，这将与所缺页面应从何处调入有关，这又可分成以下三种情况：

(1) 缺页在可执行文件上。如果欲访问虚页对应的磁盘块描述表项中的类型项是File，表示该缺页尚未运行过，其拷贝在可执行文件中，于是核心应从可执行文件中将该页调入内存。其调入的具体过程是：根据该文件所对应的系统区表项中的索引结点指针，找到该文件的索引结点，即可把从磁盘块描述表项中得到的该页的逻辑块号作为偏移量，查找索引结点中的磁盘块号表，便可找到该页的磁盘块号，再将该页调入内存。





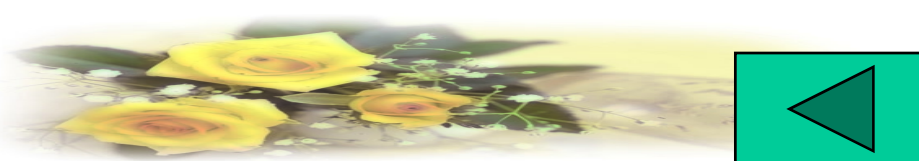
(2) 缺页在对换设备上。如果要访问的虚页对应的磁盘块描述表项中的类型是Disk，表示该缺页的拷贝是在对换设备上，因此，核心应从对换设备上将该页调入内存。其调入过程是：核心先为该缺页分配一个内存页，修改该页页表，使之指向内存页，并将页框数据表项放入相应的散列队列中，然后把该页从对换设备上调入内存。当I/O完成时，核心把请求调入该页的进程唤醒。





第十章 UNIX系统内核结构

(3) 缺页在内存页面缓冲区中。在进程运行过程中，当一个页被调出后又被要求访问时，须重新将之调入。但并非每次都要从对换设备上调入，因为被换出的页可能又被其它进程(指共享页)调入另一个物理页上，这时就可在页面缓冲池中找到该页。此时，只需适当地修改页面表项等数据结构中的信息。

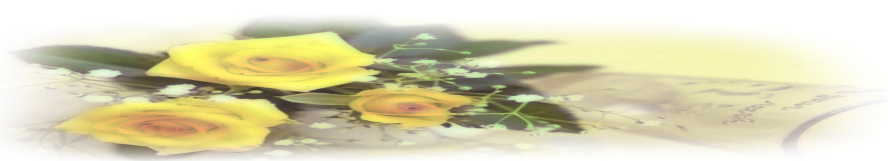




10.5 设备管理

10.5.1 字符设备缓冲区管理

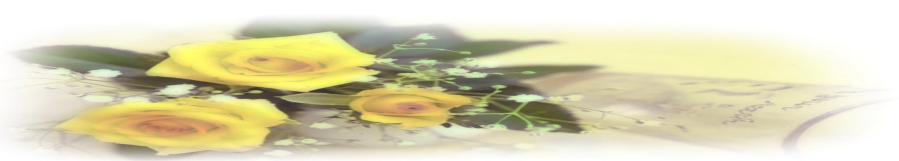
为了缓和CPU和I/O设备速度不匹配的矛盾并提高CPU和I/O设备操作的并行程度，在现代OS中，都设置了缓冲管理功能。在UNIX系统中，分别为字符设备和块设备设置了缓冲池。字符缓冲区的大小是以字节为单位计量的，而块缓冲则是以盘块大小为单位的。系统同时为每一种缓冲池提供了一组相应的操作，以便从缓冲池中获取或释放一个缓冲区。





1. 空闲字符缓冲区队列

在系统中设置了一组公用的字符缓冲区，并形成了一个公用字符缓冲池，提供给各种字符设备使用。每个缓冲区的大小为70个字节，可用cblock结构来描述。在该结构中包括四项：第一个字符的位置、最后一个字符的位置、指向下一个缓冲区的指针和用于存放64个字符的缓冲区。所有的空闲字符缓冲区都通过各自的链接指针C-next链接成一个空闲字符缓冲区队列，由队列头标中的队首指针cfreelist指向其第一个缓冲区。图10-12示出了空闲字符缓冲区队列。



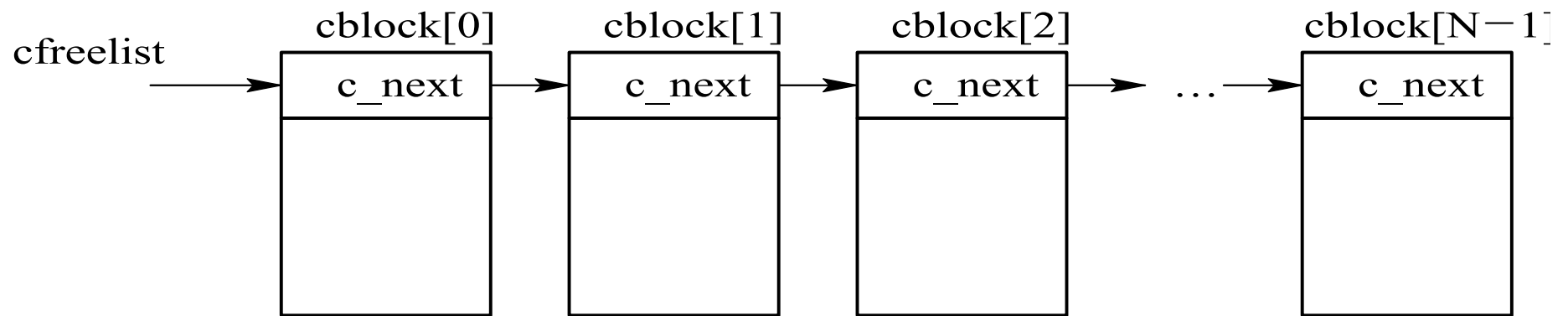
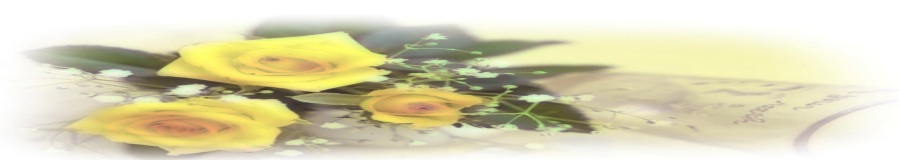


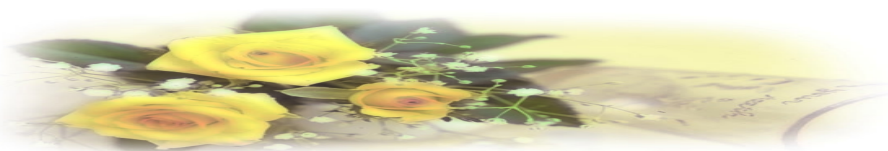
图10-12 空闲字符缓冲区队列





2. 空闲字符缓冲区的分配与回收

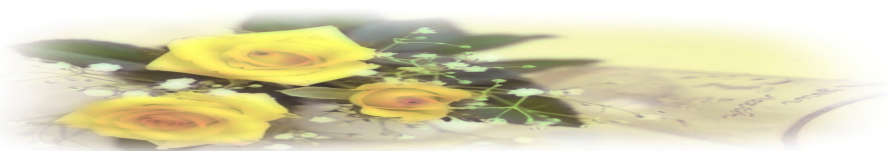
在字符设备进行I/O时，内核可利用getc过程从空闲字符缓冲区队列中取得一个空闲缓冲区，若队列空，表明已无空闲缓冲区可提供，便返回；否则，从队首取得一个空闲缓冲区，并把指向该缓冲区的指针bp返回给调用者。由于空闲缓冲区队列属于临界资源，故还须采取互斥访问措施，即在过程开始处，将处理机的优先级提升为6，在取得空缓冲区之后，再恢复处理机的优先级。





第十章 UNIX系统内核结构

当缓冲区中数据已被提取完而不再被需要时，可调用 `putcf` 过程释放缓冲区。该过程的输入参数是指向已不再需要的缓冲区的指针 `bp` 的，以便把该缓冲区送回到由空闲缓冲区队列的队首指针 `cfreelist` 指向的头部。若此时已有因申请空缓冲区而阻塞的进程，则将它唤醒。同样，对空闲缓冲区队列的访问应该互斥地进行。

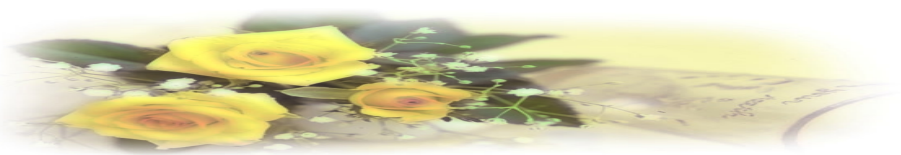




3. 设备的字符缓冲区队列

所有的字符设备在进行I/O操作时，都采用了字符缓冲区。当字符设备工作时，通常都拥有一个至几个不同的字符缓冲区队列，所有的队列都由一个称为clist结构的信息块加以控制。在该结构中包括：该队列的可用字符计数、指向该队列的队首指针和队尾指针。内核提供了getc和putc等多个过程，以对该队列进行各种操作。

(1) `getc`过程。该过程用于从一个clist结构的队首指针所指示的字符缓冲队列中取出为首的字符，然后修改该队列的可用字符计数和队首指针。当取完一个缓冲区中的所有字符时，将释放该缓冲区。该过程的返回值是取出的字符。



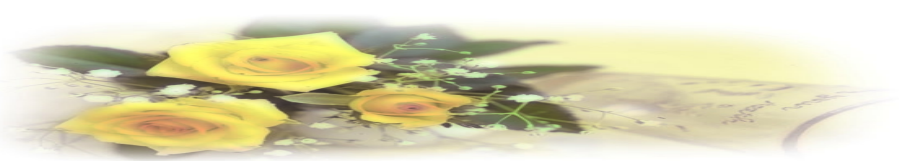


第十章 UNIX系统内核结构

(2) `putc`过程。该过程用于将一个字符C放入设备的指定字符缓冲区队列的末尾。若此时该队列空，或队列的最后一个缓冲区已满，且空闲字符缓冲区队列也空，该过程无法将字符放入队列中，则返回“-1”。

(3) `getc`过程。该过程用于从指定的设备字符缓冲区队列中取出第一个缓冲区，并将该队列的可用字符计数减去第一个缓冲区中的字符数，然后返回指向该缓冲区的指针bp。若该缓冲区已是该队列中惟一的缓冲区，则置队尾指针为空。

(4) `putc`过程。该过程用于将由bp所指向的缓冲区放入指定的设备字符缓冲区队列的末尾，然后将该队列的可用字符计数加上bp缓冲区中的字符数后返回。

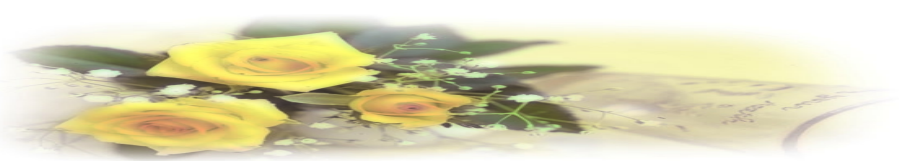




10.5.2 块设备缓冲区管理

1. 盘块缓冲区及其首部

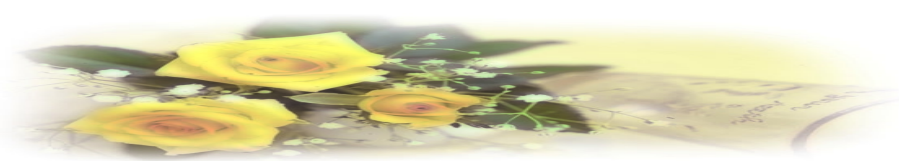
在UNIX系统中，每一个盘块缓冲区均由两部分组成：一部分用于存放数据本身，即数据缓冲区；另一部分是缓冲控制块，也称缓冲首部，用于存放对应缓冲区的管理信息。以上两者一一对应，但缓冲首部与缓冲区在物理上并不相连，只是在缓冲首部中用一个指向对应缓冲区的指针把两者联系起来，如图10-13所示。





设备号
块号
状态bflag
缓冲区指针
散列队列的前向指针
散列队列的后向指针
空闲表上的前向指针
空闲表上的后向指针

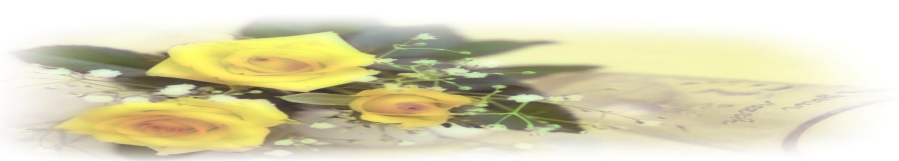
图10-13 缓冲首部





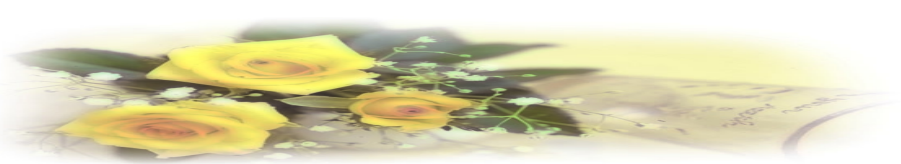
2. 盘块缓冲池结构

(1) 空闲链表。为了对缓冲区进行管理，在核心中设置了一个双向链接的空闲链表。当需要一个空闲缓冲区时，可利用getblk过程从空闲链的首部摘下一个缓冲区；在释放缓冲区时，利用brelse过程，将缓冲区挂在空闲链的末尾。





(2) 散列队列。为了加速对缓冲区的查找，系统把所有的缓冲区逐个设备地、按其块号所计算的散列值的不同，组织成多个队列，每个散列队列仍是一个双向链，其中缓冲区的数目不断地变化，各块的散列值用散列函数计算。图10-14示出了某一设备的若干散列队列。在“ $\text{block } 0 \bmod 4$ ”散列队列上，有磁盘块号为28、4和64等的缓冲区；由于每一个缓冲区都在一个散列队列中，而空闲缓冲区又应链入空闲链中，因此，一个空闲缓冲区可同时链入两个队列，并使对某一空闲缓冲区的查找可用两种方法之一来进行，即如果要求获得任一空闲缓冲区，便到空闲链上去摘取第一个缓冲区，这是最方便的；但如果要求寻找某一特定的空闲缓冲区，则搜索散列队列更方便。





第十章 UNIX系统内核结构

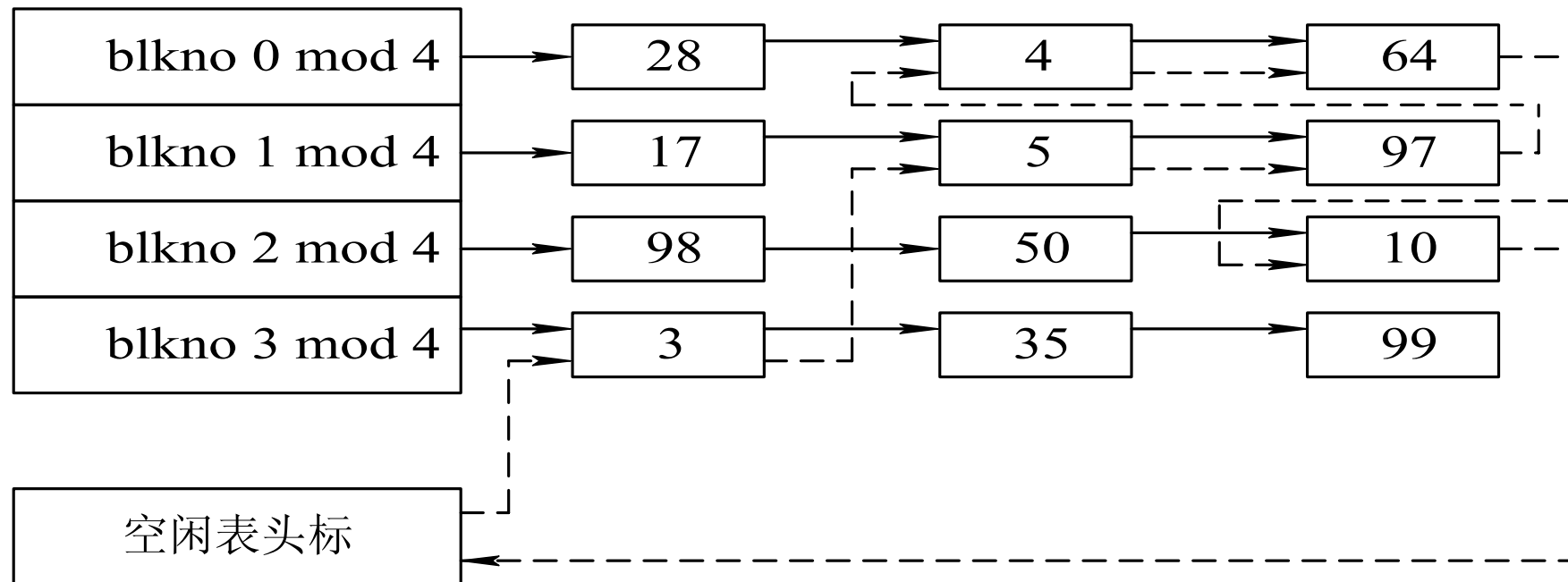
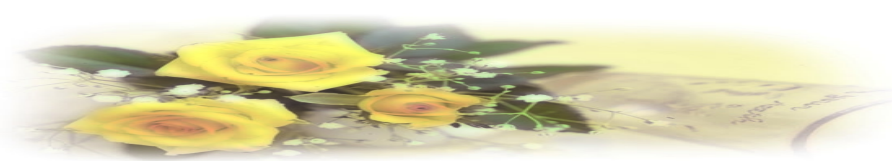


图10-14 空闲队列(链)及散列队列

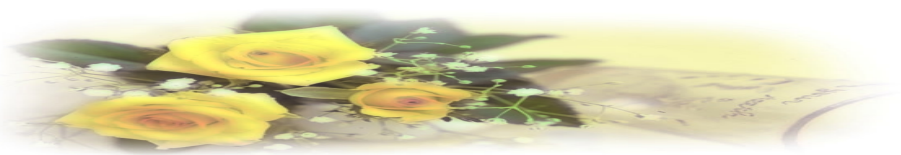




3. 盘块缓冲区的分配

由于在系统中有两种块缓冲队列，相应地，也就有两个获得块缓冲区的过程：`getblk()`过程和`getblk(dev, blkno)`过程。

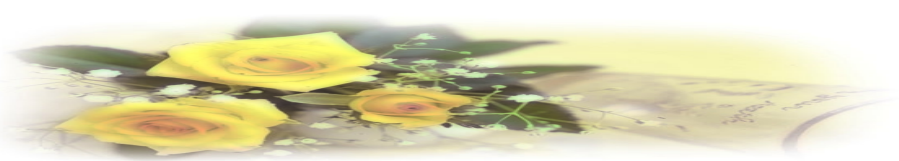
(1) `getblk()`过程。该过程用于从空闲缓冲区队列中获得任一空闲缓冲区。该过程首先检查空闲块缓冲队列是否为空，若空，便调用`sleep`过程睡眠等待，直至在空闲块缓冲队列中出现空闲缓冲区为止；否则，从空闲块缓冲队列中摘下第一个缓冲区。若在其缓冲首部中还有延迟写标志，则还须调用`bdwrite`过程，将此缓冲区中的数据写回到磁盘中，再从空闲队列中取得一个空缓冲区；否则，便将`b-flag`中的`b-busy`标志置为1，并返回指向该缓冲区的指针`bp`。





第十章 UNIX系统内核结构

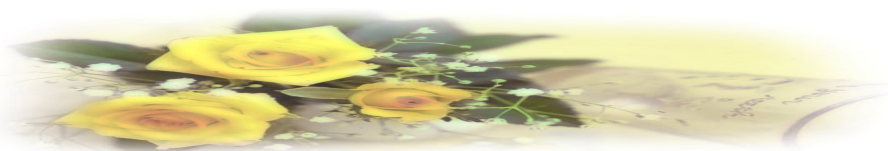
(2) `getblk(dev, blkno)`过程。该过程用于为指定设备`dev`和盘块号为`blkno`的盘块申请一个缓冲区。核心首先检查要读入的盘块内容是否已在某个缓冲区中，若发现已在某缓冲区中，便不再从磁盘上读入；否则，核心须从磁盘上将数据读入，这时才需为其分配一个空缓冲区。类似地，当要把数据写入一特定盘块时，核心先检查该盘块的内容是否已在某缓冲区，仅当该块的内容尚不在缓冲区中时，才需调用`getblk()`过程，分配一个空缓冲区。





4. 盘块缓冲区的回收

当核心用完某缓冲区时，可调用**brelse**过程将之收回。此前，可能有些进程因等待使用该缓冲区而睡眠，此时，释放者进程应将睡眠队列的队首进程唤醒。此外，还有可能有进程因空闲链表空而处于等待状态，同样也应将之唤醒。如果在所释放的缓冲区中的数据是有效的，为使以后在某进程需要它时也能直接从缓冲区中读出而不必启动磁盘的I/O操作，可将该缓冲区链入空闲链表的末尾；否则(缓冲区中数据无效)，应将它链入空闲队列的头部。空闲链表属于临界资源，为了保证对其操作的互斥性，UNIX系统通过提高处理机的运行级对中断加以屏蔽的方法来实现。

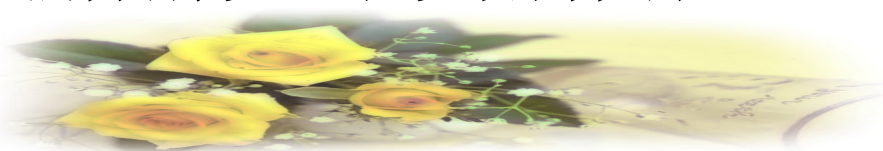




10.5.3 内核与驱动程序接口

1. 设备开关表的作用

通常，任何一个驱动程序都包括用于执行不同操作的多个函数，如打开、关闭、读或写等。为了能方便地找到各函数的入口地址，使系统控制能方便地转向各函数，系统为每类设备提供了一个设备开关，其中含有各函数的入口地址。由多种类型的设备开关构成一张设备开关表，表中的每一行是一类设备驱动程序的各函数的入口地址；表的每一列是执行相同操作的不同(设备类型)函数。图10-15示出了字符和块设备开关表与系统调用和驱动函数的关系。由图可以看出，设备开关表是核心与驱动程序间的接口，系统调用通过设备开关表转向相应驱动程序的函数。由于块设备与字符设备的驱动程序有所不同，故系统为它们分别设置了设备开关表。



第十章 UNIX系统内核结构

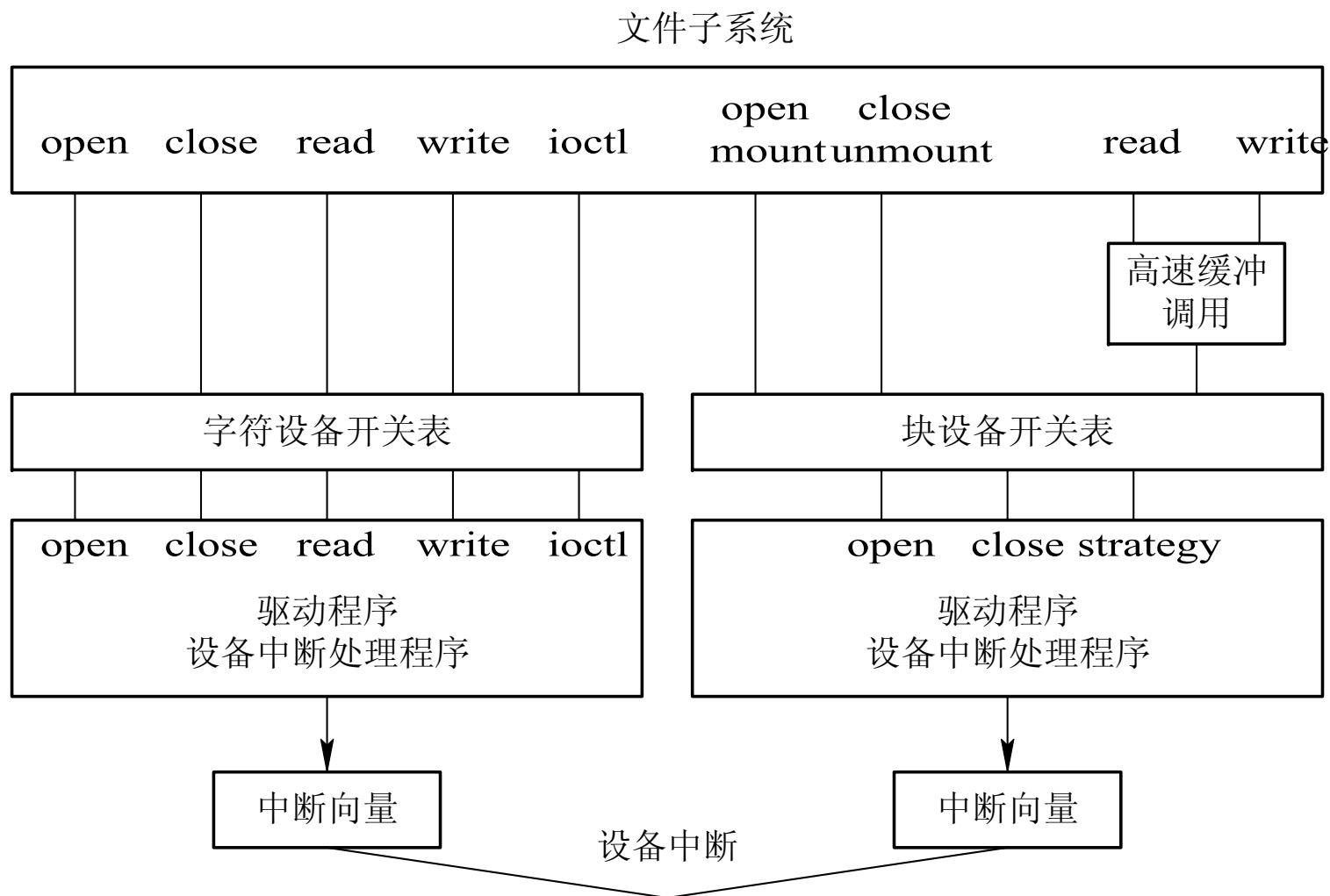
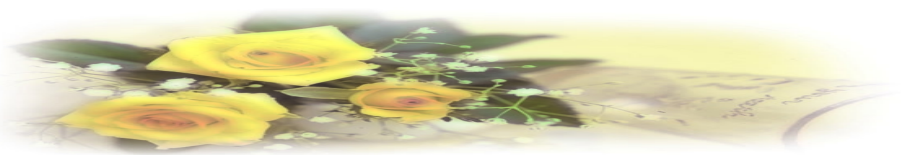


图10-15 设备开关表及系统调用和驱动程序间的接口



2. 块设备开关表

图10-16 示出了块设备开关表，其中，第一行为0号设备的开关，它包括三个表项，其中gdopen是0号设备专用的打开函数的入口地址，该函数用于打开指定的磁盘驱动器；gdclose是0号设备专用的关闭函数的入口地址，该函数用于关闭指定的磁盘驱动器；gdstrategy是策略函数的入口地址，该函数用于在数据缓冲区与块设备之间传输数据。块设备开关表的第二行是1号块设备的开关。

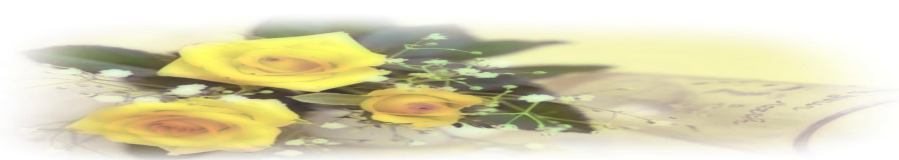


第十章 VMX系统内核结构



表项 \ 函数	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gdopen	gdclose	gdstrategy
⋮	⋮	⋮	⋮

图10-16 块设备开关表



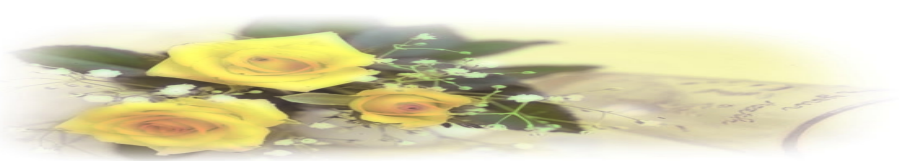


3. 字符设备开关表

图10-17示出了字符设备开关表。表中共有三行，表明共有三类字符设备驱动程序。每个驱动程序含有下述五个函数的入口地址：

- (1) 打开特定字符设备的函数open。
- (2) 关闭特定字符设备的函数close。
- (3) 读特定字符设备的函数read。
- (4) 写特定字符设备的函数write。

(5) 预置该设备参数的函数及读取该设备预置参数的函数等的入口地址。

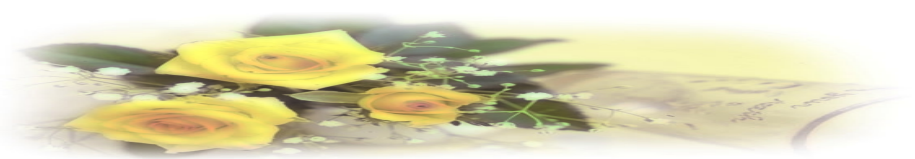


第十章 UNIX系统内核结构



表项 \ 函数	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioclt
2	syopen	nulldev	syread	sywrite	syioctl

图10-17 字符设备开关表

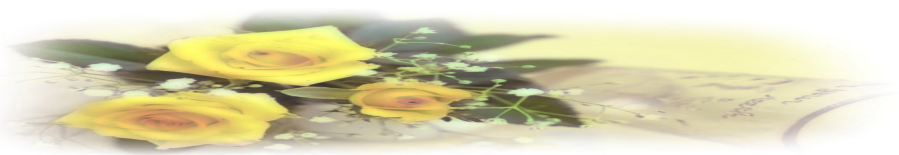




10.5.4 磁盘驱动程序

1. 打开磁盘驱动器的过程gdopen

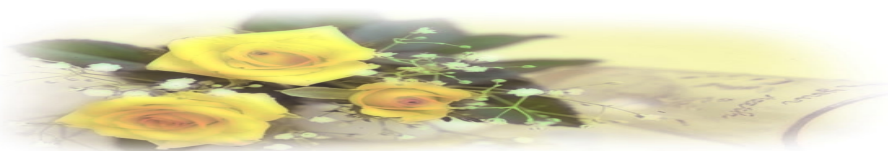
在UNIX系统中，设备被看做是一种特殊类型的文件，因而在使用该文件之前，也须先将它打开。gdopen便是用于打开磁盘驱动器的过程，该过程的输入参数是设备号，无输出参数。进入该过程后，首先检查系统中是否有由输入参数dev所指定类型的磁盘驱动器。若有，再检查它是否已被打开，如果尚未打开，便将此驱动器打开，亦即，将该磁盘控制器表中的标志b-flag设置为B-ONCE；再调用gdtimer过程启动对应的控制器和设备短期时钟闹钟，用于控制磁盘驱动器的执行时间。若系统中无指定类型的磁盘驱动器，则置相应的出错信息后返回。





2. 启动磁盘控制器的过程gdstart

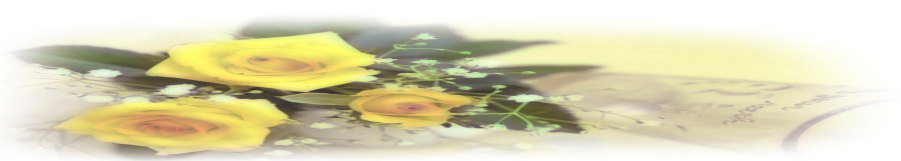
在进行磁盘的读、写之前，应首先装配磁盘控制器中的各个寄存器，然后再启动磁盘控制器。这些功能是由gdstart过程完成的。该过程的输入参数是控制器号ctl，无输出参数。进入该过程后，先从磁盘设备控制表中找到I/O队列的队首指针，若它为0，表示I/O队列空，无I/O缓冲区可取，于是返回；否则，将控制器表中的忙闲标志b-active置“1”。设置磁盘控制器中的各寄存器，如磁盘地址寄存器、内存总线地址寄存器、控制状态寄存器、字计数器等，最后启动磁盘控制器读(或写)后返回。





第十章 UNIX系统内核结构

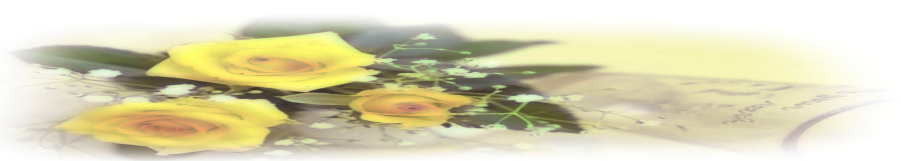
而gdstartegy过程的主要功能则是把指定的缓冲首部排在磁盘控制器I/O队列的末尾，并启动磁盘控制器。输入参数是指向缓冲首部的指针bp，无输出参数。进入该过程后，先检查磁盘控制器队列是否空，若空，便把缓冲首部的始址赋予I/O队列的队首指针；否则，便将该缓冲首部放在I/O队列的末尾。再判别磁盘设备是否忙，若不忙，便调用gdstart过程启动磁盘控制器，以传输I/O队列中第一个缓冲区中的数据；否则返回。





3. 磁盘中断处理过程gdintr

当磁盘I/O传送完成并发出中断请求信号时，CPU响应后将通过中断总控程序进入磁盘中断处理过程gdintr。该过程的输入参数是控制器号ctl。进入该过程后，先检查磁盘是否已经启动，若尚未启动，程序便不予理睬即返回；若已启动，则还须先通过对状态寄存器的检查，来了解本次传送是否出错。若已出错，便在控制终端上显示出错信息。由于磁盘的出错率较高，因而并不采取一旦出错便停止传送的策略，而是做好重新执行的准备，然后再传送。仅当重试多次都失败且超过规定的执行时间时，才设置出错标志。如未出错，则继续传送下一个缓冲区中的数据。





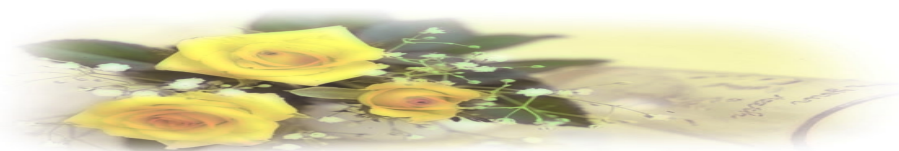
10.5.5 磁盘读/写程序

1. 磁盘的读/写方式

1) 读方式

在UNIX系统中有两种读方式：一般读方式：只把盘块中的信息读入缓冲区，由bread过程完成。

提前读方式：当一个进程要顺序地读一个文件所在的各个盘块时，会预见到所要读的下一个盘块，因而在读出指定盘块(作为当前块)的同时，可要求提前将下一个盘块(提前块)中的信息读入缓冲区。这样，当以后需要该盘块的数据时，由于它已在内存，故而可缩短读这块数据的时间，从而改善了系统性能。提前读功能由breada过程完成。





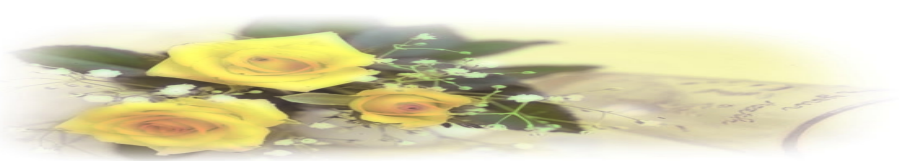
第十章 UNIX系统内核结构

2) 写方式

UNIX系统有三种写方式：

一般写方式：这种方式是真正地把缓冲区中的数据写到磁盘上，且进程须等待写操作完成，由bwrite过程完成。

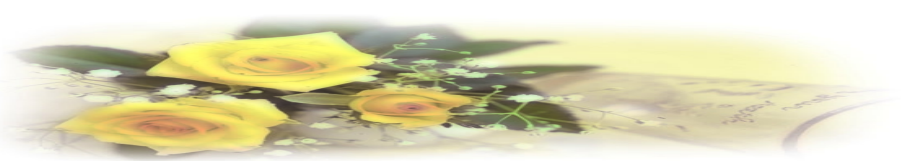
异步写方式：进程无需等待写操作完成便可返回，异步写过程是bawrite。





第十章 UNIX系统内核结构

延迟写方式: 该方式并不真正启动磁盘, 而只是在缓冲首部设置延迟写标志, 然后便释放该缓冲区, 并将之链入空闲链表的末尾。以后, 当有进程申请到该缓冲区时, 才将其内容写入磁盘。引入延迟写的目的是为了减少不必要的磁盘 I/O, 因为只要没有进程申请到此缓冲区, 其中的数据便不会被写入磁盘, 倘若再有进程需要访问其中的数据时, 便可直接从空闲链表中摘下该缓冲区, 而不必从磁盘读入。延迟写方式由过程bdwrite完成。



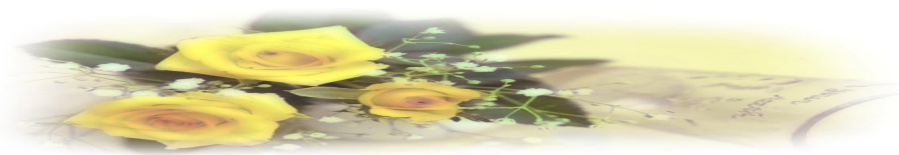


第十章 UNIX系统内核结构

2. 读过程bread和breada

1) 一般读过程bread

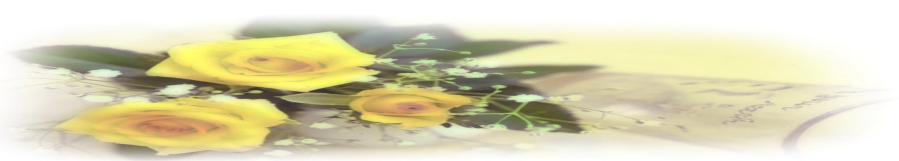
该过程的输入参数是文件系统号(即逻辑设备号)及块号。进入该过程后,先调用getblk过程申请一缓冲区,若缓冲首部标明该缓冲区中数据是有效的,便无需再从磁盘读入,直接返回;若所需数据尚未读入,则应先填写缓冲首部,如设置读标志,以表明本次是读操作。其次是设置块的初始字符计数值(如1024),再通过块设备开关表转入相应的gdstartegy过程,启动磁盘传送。由于一般读方式下进程要等待读操作完成,故须调用sleep过程使自己睡眠,直至读操作完成时再由中断处理程序将该进程唤醒,最后将缓冲区首部的指针bp作为输出参数,返回给调用进程。





2) 提前读过程breada

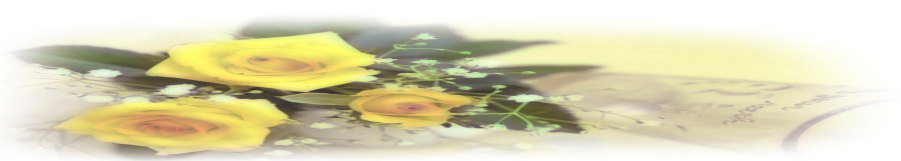
该过程的输入参数是当前读的文件系统号(设备号)、盘块号, 以及提前读的文件系统号和盘块号。进入该过程后, 先判别当前块是否在缓冲池中。若当前块不在缓冲池, 须调用getblk过程为其分配一缓冲区。若缓冲区中的数据无效, 则应填写缓冲区首部, 通过设备开关表转入策略过程, 启动磁盘读入。若当前块已在缓冲池中, 或所分配的缓冲区中数据有效, 都将转去读提前块。





第十章 VMIX系统内核结构

若提前块的数据缓冲区不在缓冲池中，同样要调用getblk过程为读提前块而分配一缓冲区。若所得数据缓冲区中的数据有效，则调用brelse过程将该缓冲区释放，以便其他进程能对该缓冲区进行访问；否则，填写缓冲区首部后启动磁盘。最后，若当前块缓冲区在缓冲池中，则调用bread过程去读当前块；否则，调用sleep过程使自己睡眠，以等待读操作完成。在读操作完成而被唤醒后，将该缓冲区首部的指针返回。

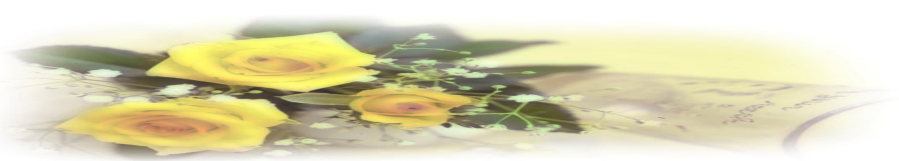




3. 写过程bwrite、bawrite和bdwrite

1) 一般写过程bwrite

该过程的输入参数是缓冲区指针bp。进入该过程后，根据bp指针找到缓冲区首部，设置缓冲区首部的初值，通过设备开关表转入策略过程，启动磁盘。如是一般写，应等待I/O完成，为此，须调用sleep过程使自己睡眠。I/O完成后才被唤醒，再调用brelse过程释放该缓冲区。如是异步写，且有延迟写标志，则在给缓冲区打上标志后，将之放入空闲链表的首部。



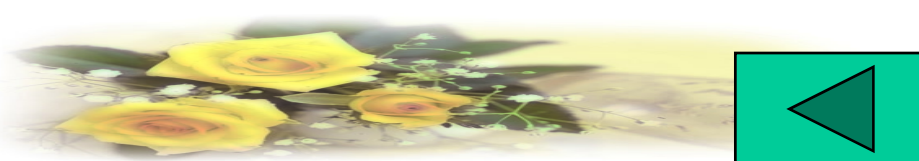


2) 异步写过程bawrite

它与一般写过程很相似，但不需等待I/O完成即可返回。进入bawrite过程后，设置异步写标志，再调用bwrite过程实现之。

3) 延迟写过程bdwrite

延迟写过程也很简单。这里只需设置延迟写标志及数据有效标志，再调用brelse过程将该缓冲区释放，并链入空闲链表的尾部。以后，当某进程调用getblk获得该缓冲区时，再用异步写方式将缓冲区内容写入磁盘。



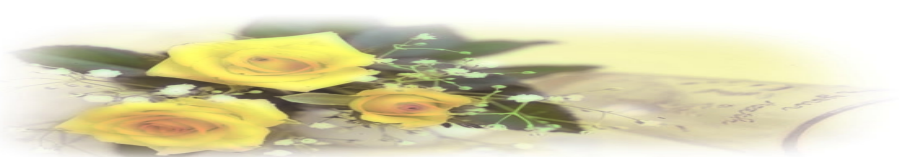


10.6 文件管理

10.6.1 UNIX文件系统概述

1. UNIX文件系统的特点

(1) 文件系统的组织是分级树形结构形式。UNIX文件系统的结构基本上是一棵倒向的树，这棵树的根是根目录，树上的每个结点都是一个目录，而树的叶则是信息文件。每个用户都可建立自己的文件系统，并把它安装到UNIX文件系统中，从而形成一棵更大的树。当然，也可以把安装上去的文件系统完整地拆卸下来，因而，整个文件系统显得十分灵活、方便。

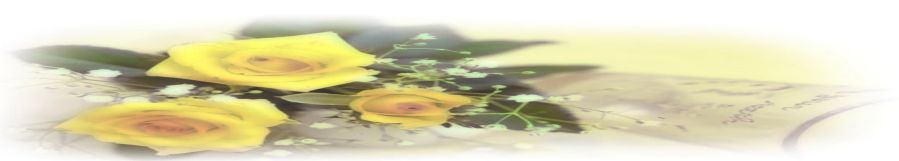




第十章 UNIX系统内核结构

(2) 文件的物理结构为混合索引式文件结构。所谓混合索引文件结构，是指文件的物理结构可能包括多种索引文件结构形式，如单级索引文件结构、两级索引文件结构和多级索引文件结构形式。这种物理结构既可提高对文件的查询速度，又能节省存放文件地址所需的空間。

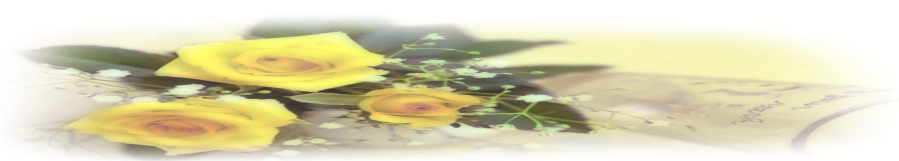
(3) 采用了成组链接法管理空闲盘块。这种方法实际上是空闲表法和空闲链法相结合的产物，它兼备了这两种方法的优点而克服了这两种方法都有的表(链)太长的缺点，这样，既可提高查找空闲盘块的速度，又可节省存放盘块号的存储空间。





第十章 UNIX系统内核结构

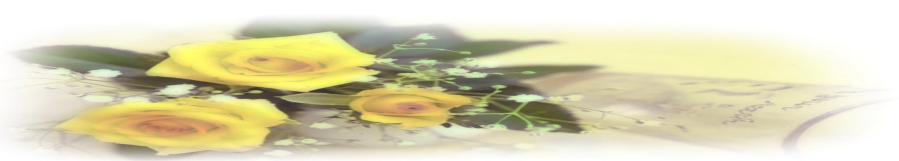
(4) 引入了索引结点的概念。在UNIX系统中，把文件名和文件的说明部分分开，分别作为目录文件和索引结点表中的一个表项，这不仅可加速对文件的检索过程，减轻通道的I/O压力，而且还可给文件的联接和共享带来极大的方便。





2. 文件系统的结构

由于在UNIX系统中，文件名和文件属性(说明)分开存放，由文件属性构成文件的索引结点，这使UNIX的目录项与一般文件系统的目录项不同，故UNIX文件系统的结构也与一般文件有所差异。图10-18示出了引入索引结点后所形成的UNIX文件系统结构。其中，根目录中的bin是二进制系统文件的根目录；usr为用户文件的根目录；dev为特殊文件的根目录。



第十章 UNIX系统内核结构

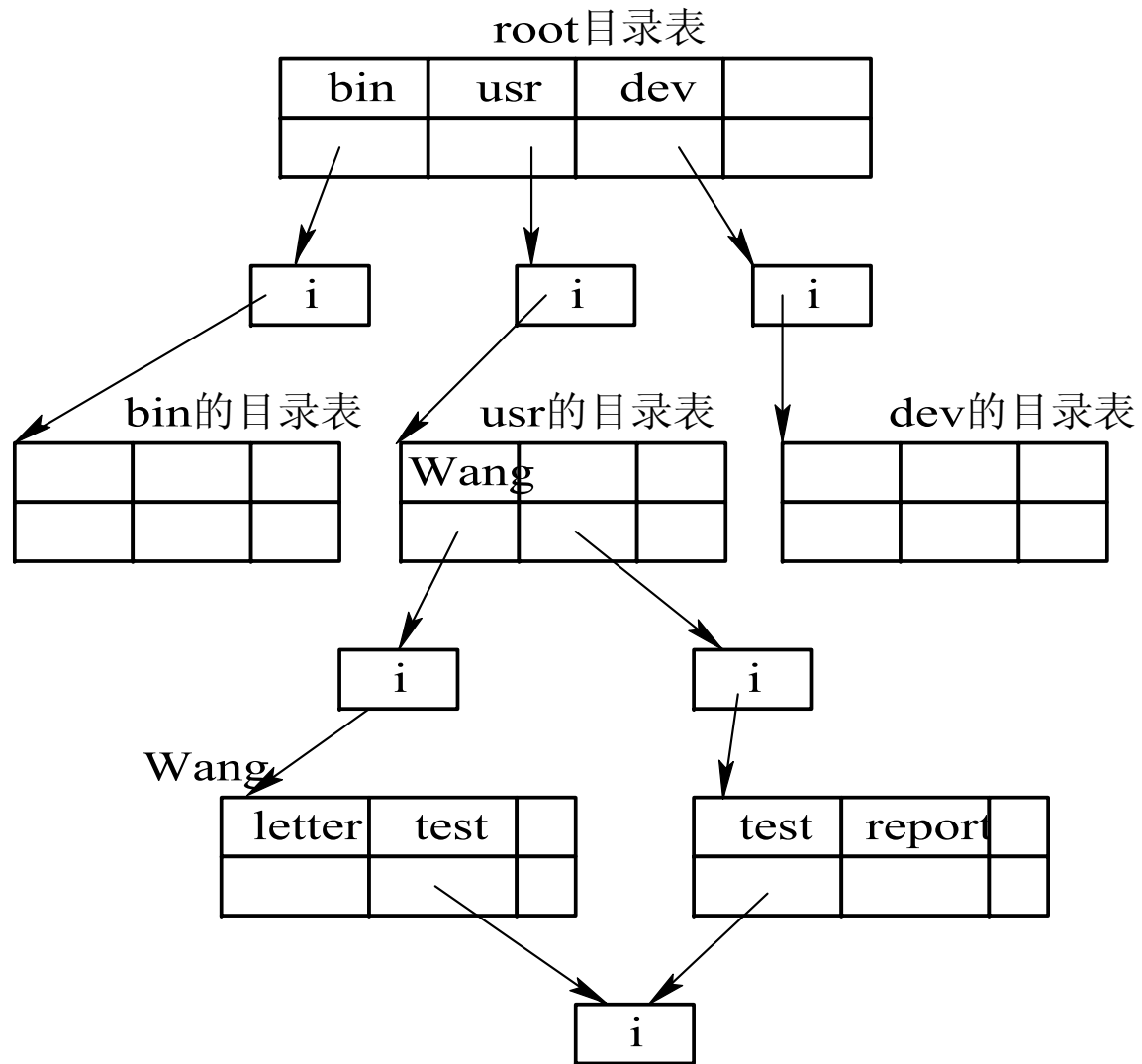


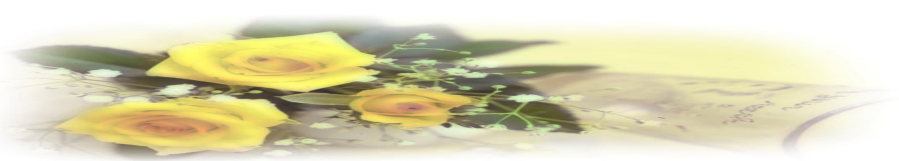
图10-18 UNIX文件系统的结构



3. 文件系统的资源管理

为了在系统中保存一份文件，必须花费一定的资源，当文件处于“未打开”状态时，文件需占用三种资源：

- (1) 一个目录项，用以记录文件的名称和对应索引结点的编号。
- (2) 一个磁盘索引结点项，用以记录文件的属性和说明信息，这些都驻留在磁盘上。
- (3) 若干个盘块，用以保存文件本身。

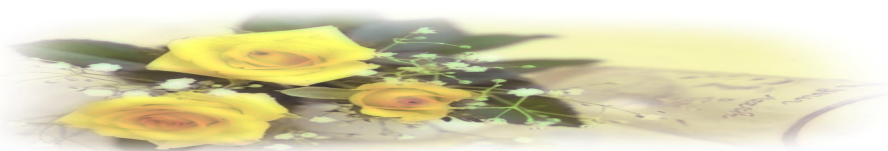




第十章 VMIX系统内核结构

当文件被引用或“打开”时，须再增加三种资源：

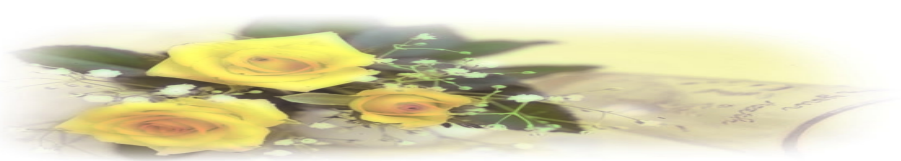
- (1) 一个内存索引结点项，该项驻留在内存中。
- (2) 文件表中的一个登记项。
- (3) 用户文件描述符表中的一个登记项。





第十章 UNIX系统内核结构

由于对文件的读写管理必须涉及到上述各种资源，因而对文件的读写管理又在很大程度上依赖于对这些资源的管理，故可从资源管理的观点来介绍文件系统。这样，对文件的管理就必然包括：① 对索引结点的管理；② 对空闲盘块的管理；③ 对目录文件的管理；④ 对文件表和描述符表的管理；⑤ 对文件的使用。下面我们先介绍文件的物理结构。



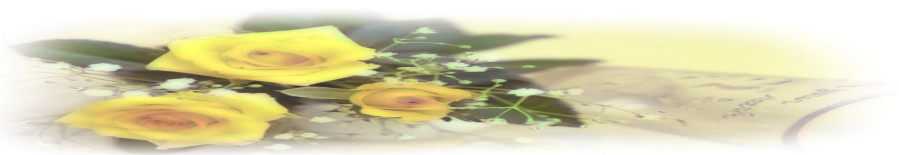


10.6.2 文件的物理结构

1. 寻址方式

1) 直接寻址

在UNIX系统中的作业，是以中、小型的作业为主的。根据对大量文件调查的结果得知，文件长度字节数在10 KB以下的占大多数。为了提高对中、小型作业的检索速度，宜采用直接寻址方式。为此，在索引结点中建立了10个地址项i-addr(0)~i-addr(9)，在每个地址项中直接置入该文件占用盘块的编号，如图10-19所示。这相当于第六章所述的单级索引文件的寻址方式。如果盘块的大小为1 KB，则当文件长度不大于10 KB时，可直接从索引结点中找出该文件的所有盘块号。



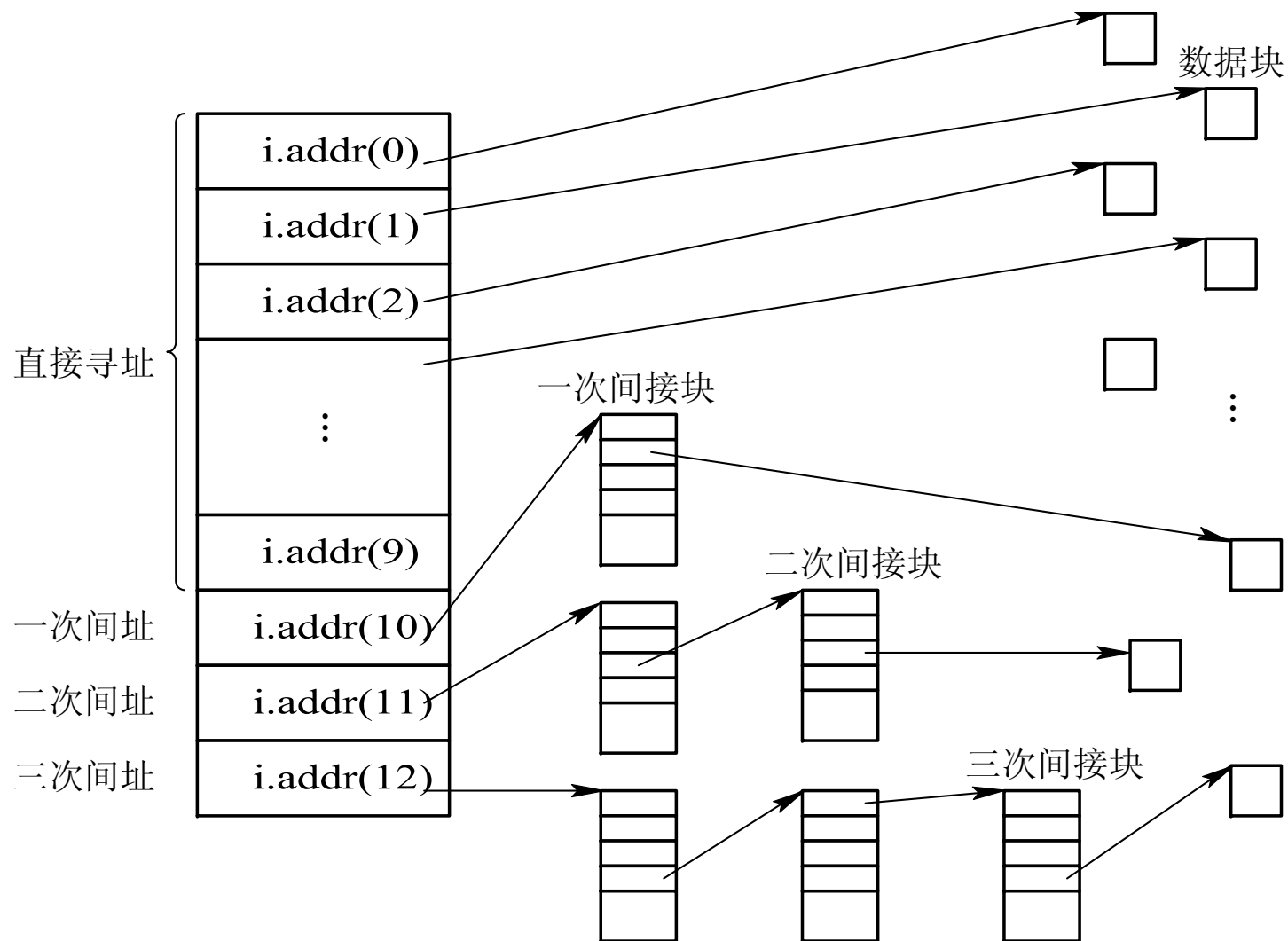
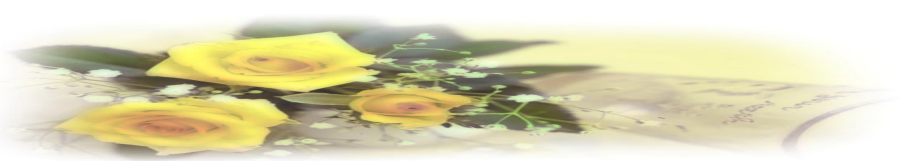


图10-19 直接寻址和间接寻址



2) 一次间接寻址方式

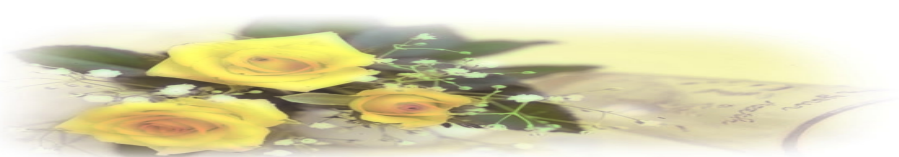
采用间接寻址方式是基于这样的事实：有不少文件，其长度可能达到几十千字节、几十兆字节甚至更长。如果全部采用直接寻址方式，就必须在索引结点中设置几百个或更多的地址项，这显然是不现实的。为了节省存放文件盘块号所占用的存储空间，在UNIX系统中又提供了所谓的“一次间接”寻址方式，即在i-addr(10)地址项中，存放的不再是存放文件的一个物理盘块号，而是将存放了直接地址的1~256个物理盘块号的盘块的编号放于其中，这相当于第6章所述的两级索引文件中的寻址方式。假定一个盘块仍为1 KB，一个盘块号占用32位，这样，i-addr(10)的寻址范围可达256 KB。





3) 多次间接寻址

为了进一步扩大寻址范围，又采用了二次间接和三次间接寻址方式，使用的地址项分别为索引结点中的i-addr(11)和i-addr(12)。二次间址相当于三级索引文件中的寻址方式，若盘块大小仍为1 KB，则可将寻址范围扩大到64 MB+256 KB；三次间址则可将寻址范围扩大到16 GB。

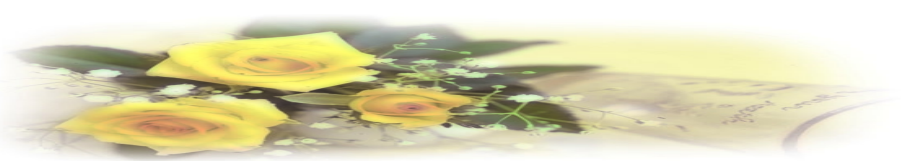




2. 地址转换

1) 将字节偏移量转换为文件逻辑块号

在UNIX文件系统中，文件被视为流式文件，每个文件有一个读、写指针，用来指示下一次要读或写的字符的偏移量。该偏移量是从文件的头一个字符的位置开始计算的。在每次读、写时，都要先把字节偏移量转换为文件的逻辑块号和块内位移量。其转换方法是：将字节偏移量除以盘块大小的字节数，其商即为文件的逻辑块号，余数为块内位(偏)移量。



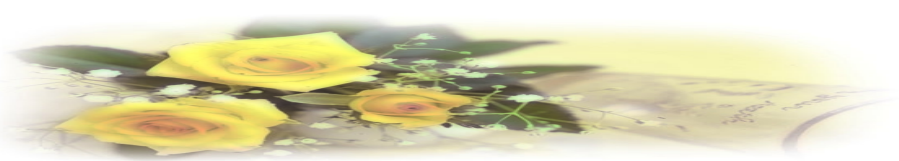


第十章 UNIX系统内核结构

2) 把文件逻辑块号转换为物理盘块号

(1) 直接寻址。当文件的逻辑块号小于或等于10时，索引结点中所存放的是直接地址。将逻辑块号转换为物理块号的方法是：首先，将文件的逻辑块号转换为索引结点中地址项的下标；其次，从该下标所指的地址项中，即可获得物理盘块号。

例如，某进程要访问其字节偏移量为9000处的数据，为此，核心须先将9000转换成文件逻辑块号8，块内偏移量为808。由于逻辑块号小于10，故可直接从相应的地址项*i-addr*(8)中找到物理盘块号为367的块(见图10-20)，在该块中的第808字节处即为文件的第9000字节处。



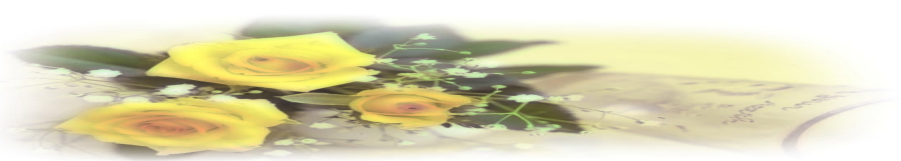


第十章 VMIX系统内核结构

(2) 一次间址。当文件系统中的逻辑块号大于10而小于或等于256+10时，为一次间址寻址方式。其地址转换过程分为两步：

第一步，由于一次间址的地址项下标为10，所以可以从 $i\text{-addr}(10)$ 中得到一次间址盘块号，由bread过程读间址块。

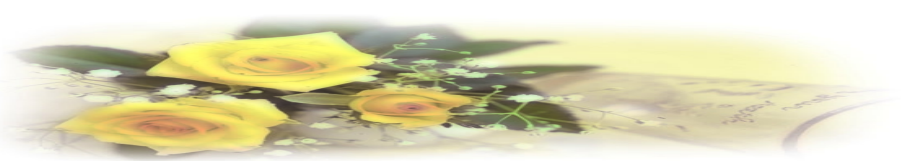
第二步，计算一次间址中文件的逻辑块号，即将文件的逻辑块号减10，根据一次间址中的逻辑块号得到间址块号中地址项的下标，再从相应下标的地址项中得到物理盘块号。





第十章 UNIX系统内核结构

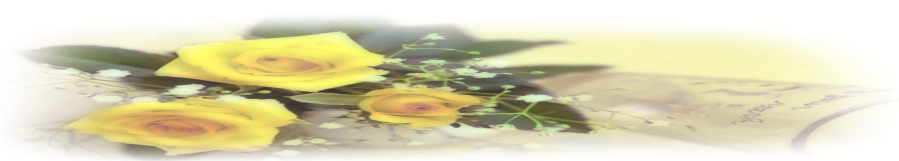
例如，进程要访问偏移量为14 000字节处的数据。核心先将14 000转换为逻辑块号13及块内偏移量688。由于 $10 < 13 < 266$ ，故应属于一次间址。这样，再从i-addr(10)中得到一次间址的盘块号为428，调用bread过程读该盘块，计算一次间址块中的文件逻辑块号为3，从中可得到盘块号为952。于是，该块中的第688字节处便是要访问的文件的第14 000字节处，见图10-20所示。





第十章 UNIX系统内核结构

(3) 多次间址。当文件的逻辑块号大于266，又小于或等于64 266时，应采用二次间址；逻辑块号大于64 266时应采用三次间址。多次间址的转换方法和一次间址时相似，但要多次循环，这里不再赘述。



第十章 UNIX系统内核结构

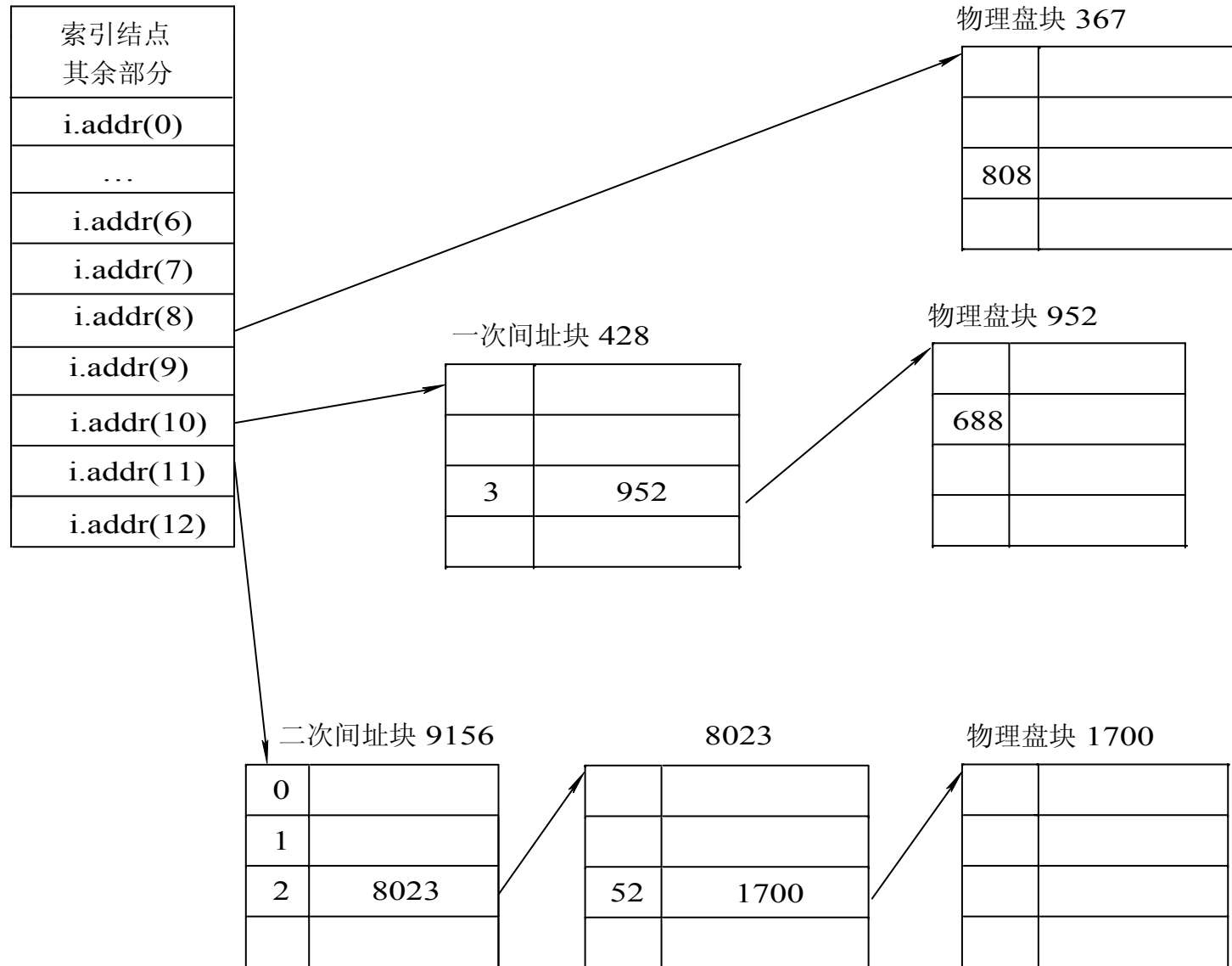


图10-20 文件的地址映射示例

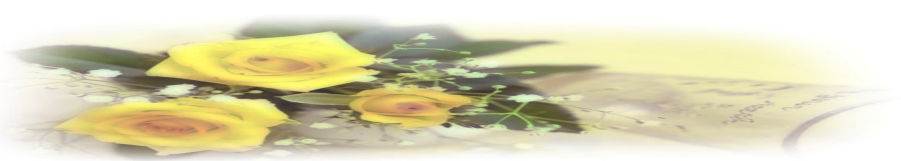


10.6.3 索引结点的管理

1. 超级块(Superblock)

在UNIX系统中，通常把每个磁盘或磁带看做是一个文件卷，在每个文件卷上可存放一个文件系统。文件卷可占用许多物理盘块，其中0#盘块一般用于系统引导；1#块为超级块。磁盘块和索引结点的分配与回收，将涉及到超级块。超级块是专门用于记录文件系统中盘块和磁盘索引结点使用情况的一个盘块，其中含有以下各字段：

- (1) 文件系统的盘块数目。
- (2) 空闲盘块号栈，即用于记录当前可用的空闲盘块编号的栈。





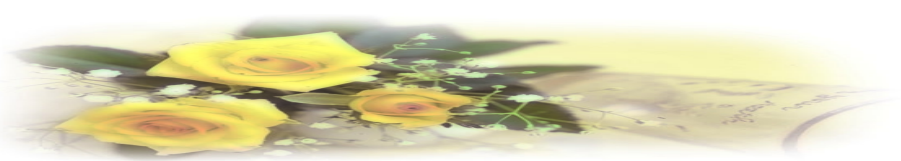
第十章 VMX系统内核结构

(3) 当前空闲盘块号数目，即在空闲盘块号栈中保存的空闲盘块号的数目。它也可被视为空闲盘块编号栈的指针。

(4) 空闲磁盘i结点编号栈，即记录了当前可用的所有空闲i结点编号的栈。

(5) 空闲磁盘i结点数目，指在磁盘i结点栈中保存的空闲i结点编号的数目，也可视为当前空闲i结点栈顶的指针。

(6) 空闲盘块编号栈的锁字段，是对空闲盘块进行分配与回收时的互斥标志。



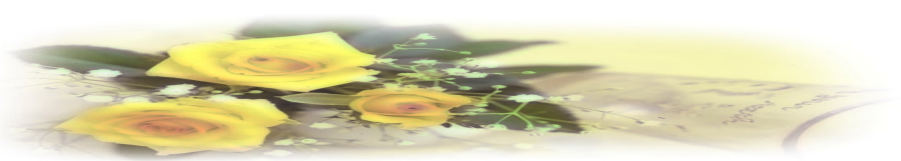


第十章 VMFS系统内核结构

(7) 空闲磁盘i结点栈的锁字段，是对空闲磁盘i结点进行分配与回收时的互斥标志。

(8) 超级块修改标志，用于标志超级块是否被修改，若已被修改，则在定期转储时，应将超级块内容从内存复制到文件系统中。

(9) 修改时间，指记录超级块最近一次被修改的时间。





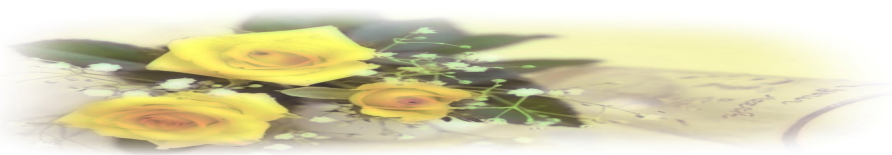
2. 磁盘索引结点的分配与回收

1) 分配过程ialloc

该过程的主要功能是每当核心创建一个新文件时，都要为之分配一个空闲磁盘i结点。若分配成功，便再分配一内存i结点。其分配过程如下：

(1) 检查超级块上锁否。超级块是临界资源，诸进程必须互斥地对它进行访问。因此在进入ialloc过程后，要首先检查它是否已被上锁，若已被锁住，进程便睡眠等待。

(2) 检索i结点栈空否。若发现i结点栈中已无空闲i结点编号，则应从盘中再调入一批结点号进栈。若盘中也已无空闲i结点，便做出错处理后返回。



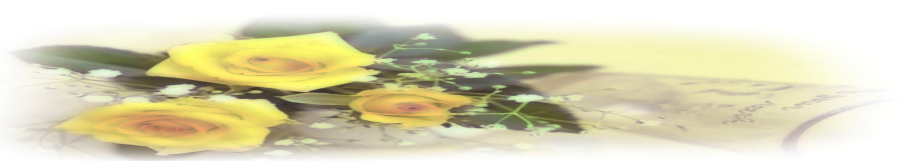


第十章 UNIX系统内核结构

(3) 从空闲i结点编号栈中分配一个i结点，并且加以初始化，填写有关文件的属性。

(4) 分配内存i结点。在UNIX系统中，每当创建一个新文件后，便随之把它打开。因此在分配了磁盘i结点后，便调用iget过程再为之分配一个内存i结点，并对它进行初始化，然后把磁盘i结点的内容复制到内存i结点中。

(5) 将磁盘i结点总数减1，并在置超级块的修改标志后返回。





第十章 UNIX系统内核结构

2) 回收过程ifree

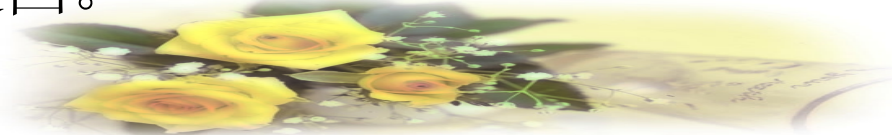
当一个文件已不再被任何进程需要时，应将该文件从磁盘上删除，并回收其所占用的盘块及相应的磁盘i结点。过程ifree便是用于回收磁盘i结点的。其所包括的操作有：

(1) 检查超级块上锁否。若已上锁便直接返回，即不能把本次回收的i结点号记入空闲i结点编号栈中。

(2) 检查i结点编号栈满否。若栈中的i结点编号的个数已等于100，则表示栈已满，无法再装入新的i结点号，也须立即返回。

(3) 若i结点编号栈未滿，便使回收的i结点的编号进栈，并使当前空闲i结点数加1。

(4) 置超级块修改标志后返回。

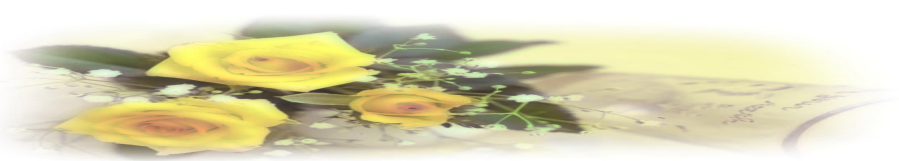




3. 内存索引结点的分配与回收

1) 分配过程iget

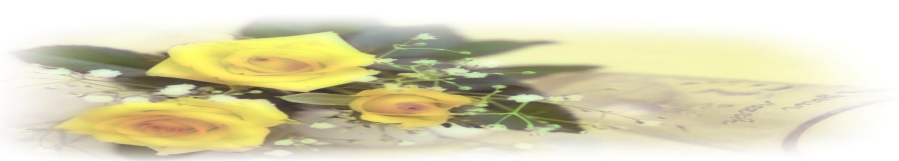
该过程的主要功能是在打开文件时，为之分配内存i结点。由于允许文件被共享，因此，如果一文件早已被其他用户打开并有了内存i结点，此时便只需将该i结点中的引用计数加1；如果文件尚未被其他用户打开，则由iget过程为该文件分配一个内存i结点，并调用bread过程将其磁盘i结点的内容拷贝到内存i结点中，同时进行初始化。





2) 回收过程iput

每当进程要关闭某文件时，须调用iput过程先对该文件的内存i结点中的引用计数做减1操作。若结果为0，便回收该内存i结点，再判断其磁盘的i结点的联接计数，若其结果也为0，便删除此文件，并回收分配给该文件的盘块和磁盘i结点。

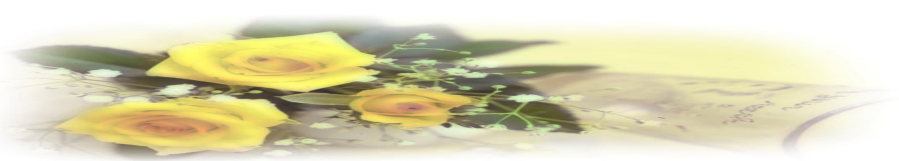




10.6.4 空闲磁盘空间的管理

1. 文件卷的组织

在UNIX系统中的文件存储介质，可采用磁盘或磁带。通常可把每个磁盘或磁带看做是一个文件卷，每个文件卷上可存放一个具有独立目录结构的文件系统。一个文件卷包括许多物理块，并按照块号排列成如图10-21所示的结构。



第十章 *UNIX*系统内核结构

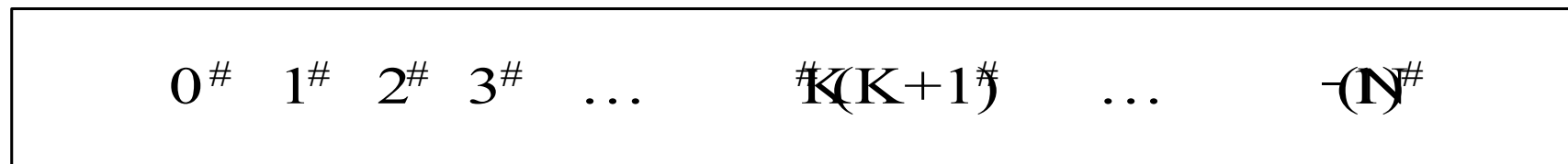
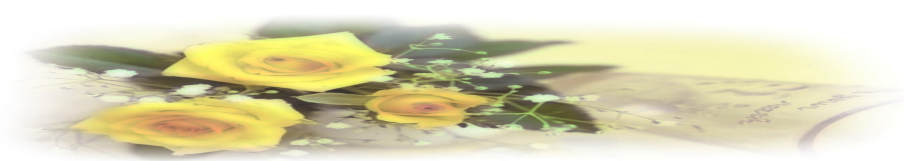


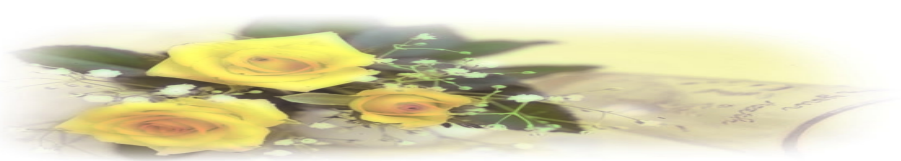
图10-21 文件卷的组织





第十章 UNIX系统内核结构

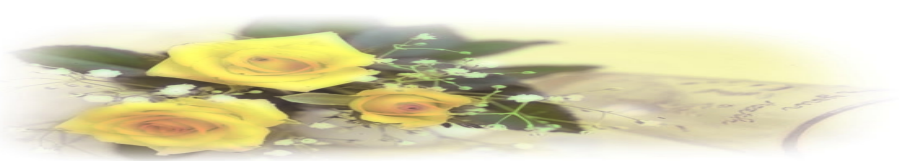
在文件卷中，0#块一般用于系统引导或空闲；1#块为超级块(Superblock)，用于存放文件卷的资源管理信息，包括整个文件卷的盘块数、磁盘索引结点的盘块数、空闲盘块号栈和空闲盘块号栈指针、空闲盘块号栈锁、空闲索引结点栈和空闲索引结点栈指针，以及空闲索引结点栈锁等。从2#块起存放磁盘索引结点，直到K#块。每个索引结点为64个字节，当盘块大小为1 KB时，每个盘块中可存放16个索引结点；从(K+1) #块起及其以后各块都存放文件数据，直至文件卷的最后一块即(N-1)#块。





2. 空闲盘块的组织

UNIX系统采用了第六章所介绍的成组链接法，对空闲盘块加以组织。该方法是将若干个(如100个)空闲盘块划归为一个组，将每一组中的所有盘块号存放在其前一组的第一个空闲盘块中，而仅把第一组中的所有空闲盘块号放入超级块的空闲盘块号栈中。例如，在图10-22所示的空闲盘块的组织中，将第四组的盘块号409、406、403等存入第三组的第一个即310号盘块中；又将第三组的盘块号310、307、304等放入第二组的第211号盘块中；而第一组的盘块号109、106、103等则放入超级块的空闲盘块号栈中。





超级块表

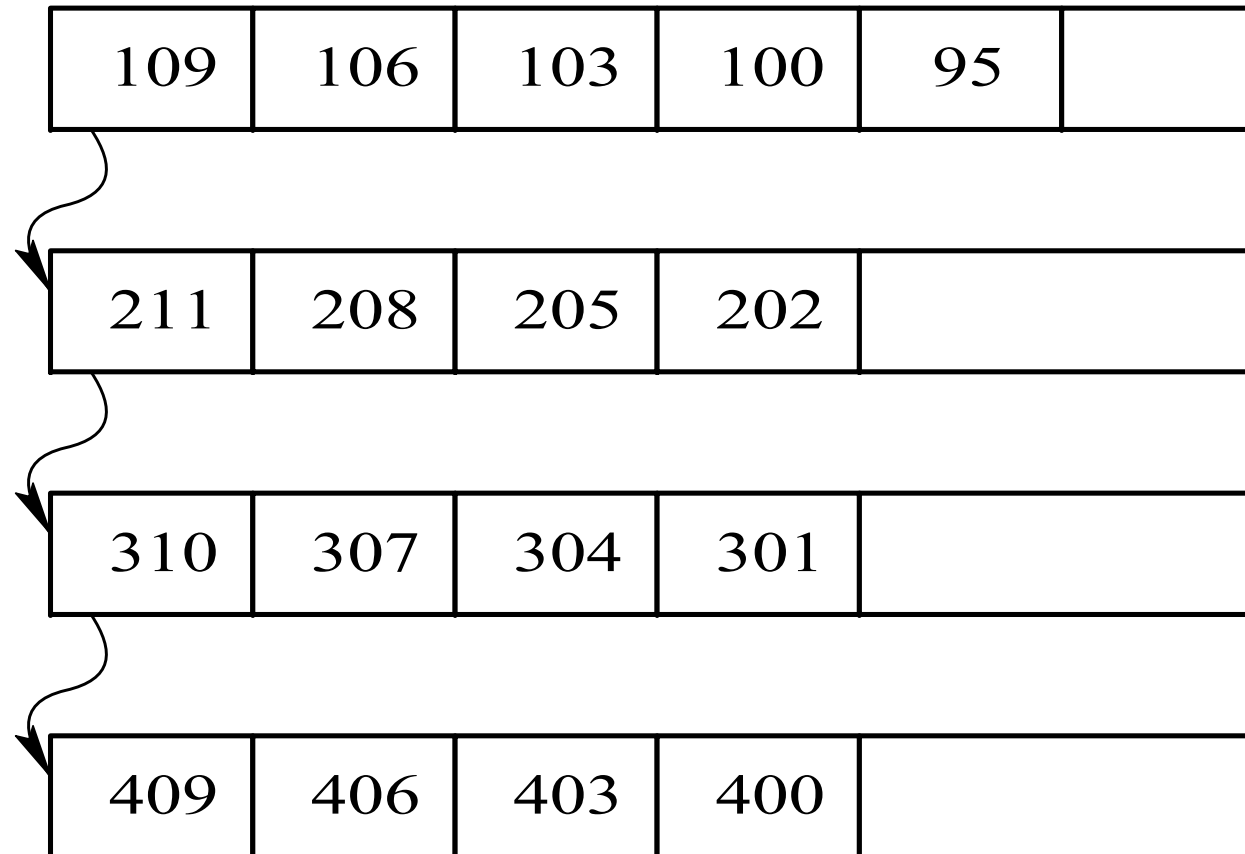
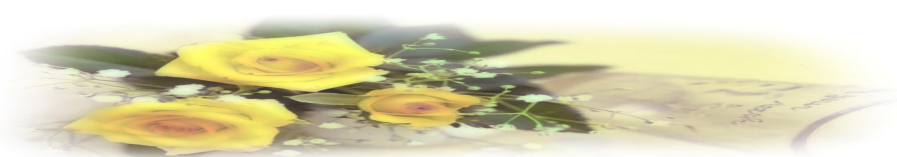


图10-22 空闲盘块的组织

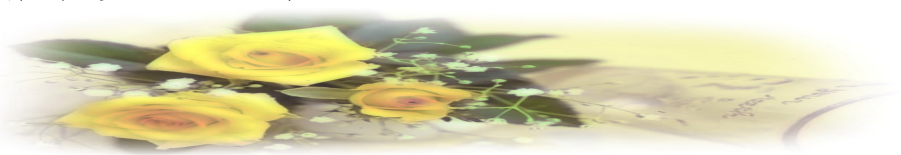




3. 空闲盘块的分配与回收

1) 空闲盘块的分配

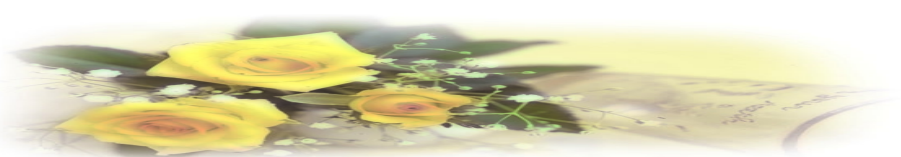
空闲盘块的分配是由alloc过程完成的, 该过程的主要功能是从空闲盘块号栈中获得一空闲盘块号。当核心要从文件系统中分配一个盘块时, 首先检查超级块中的盘块号栈是否已经上锁。若已锁上, 便调用sleep过程睡眠; 否则, 将超级块的空闲盘块号栈顶的盘块号(如95号)分配出去。如果所分配的空闲盘块号是在栈底(如109号), 由于在该号盘块中又包含了第二组盘块的所有盘块号(如211、208等), 于是核心在给超级块上锁后, 应先调用bread过程将该栈底盘块号对应盘块中的内容读出, 作为新栈的内容进栈; 然后, 再将原有栈底所对应的盘块作为空闲盘块分配出去(即109号盘块); 最后, 将超级块解锁, 唤醒等待超级块解锁的进程。





2) 空闲盘块的回收

空闲盘块的回收是由free过程完成的。在回收空闲盘块时，首先检查超级块中的盘块号栈是否已经上锁，若已上锁，便调用sleep睡眠；否则，再检查空闲盘块号栈是否已满。如果空闲盘块号栈未满，可直接将回收盘块的编号记入空闲盘块号栈中；若栈已满，须调用betblk过程申请一个缓冲区，将栈中的所有空闲盘块号复制到新回收的盘块中，再将新回收盘块的编号作为新栈的栈底块号进栈。



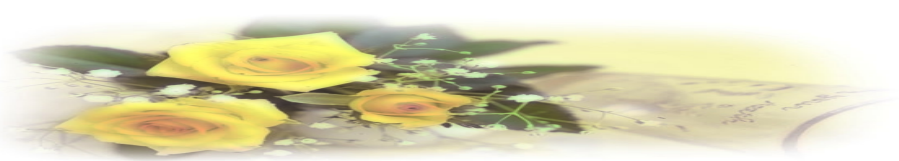


10.6.5 文件表的管理

1. 用户文件描述符表的管理

1) 用户文件描述符表

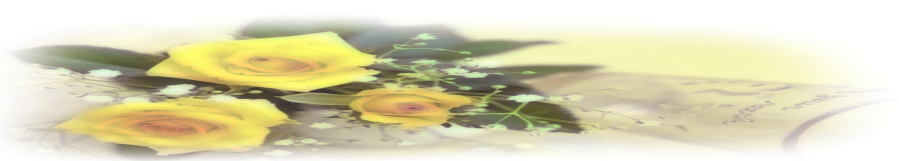
为了方便用户和简化系统的处理过程，在UNIX系统V中，在每个进程的U区中都设置了一张用户文件描述符表。核心先对其打开请求做仔细检查后，便在该进程的用户文件描述符表中分配一个空项，取其在表中的位移量作为文件描述符fd(file descriptor)返回给用户。以后，当用户再访问该文件时，只需提供该文件描述符fd，系统根据fd便可找到相应文件的内存索引结点。





2) ufalloc过程

用户文件描述符表项的分配是由ufalloc过程来完成的。该过程首先是从用户文件描述符表中查找一个空项，若找到，便将该表项的序号fd作为文件描述符写入进程的U区，然后返回；否则，置出错标志后返回“NULL”。





2. 文件表的管理

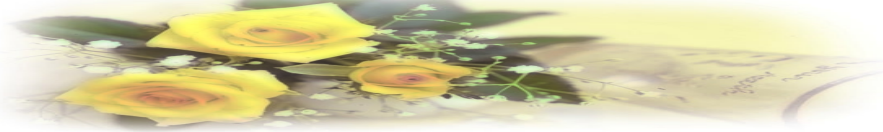
1) 文件表

系统为了对文件进行读/写，设置了一个确定读/写位置偏移量的读/写指针。该读/写指针应放在何处，是否可放在用户文件描述符表中，我们对此做了简单的分析。用户在读/写文件时，可采用三种方式：

第一种方式，多个用户读/写各自的文件，见图10-23中的上部；

第二种方式，多个用户共享一个文件，但彼此独立地对文件进行读/写，见图10-23的中部；

第三种方式，多个用户共享一个文件，且共享一个读/写指针，见图10-23的下部。



第十章 UNIX系统内核结构

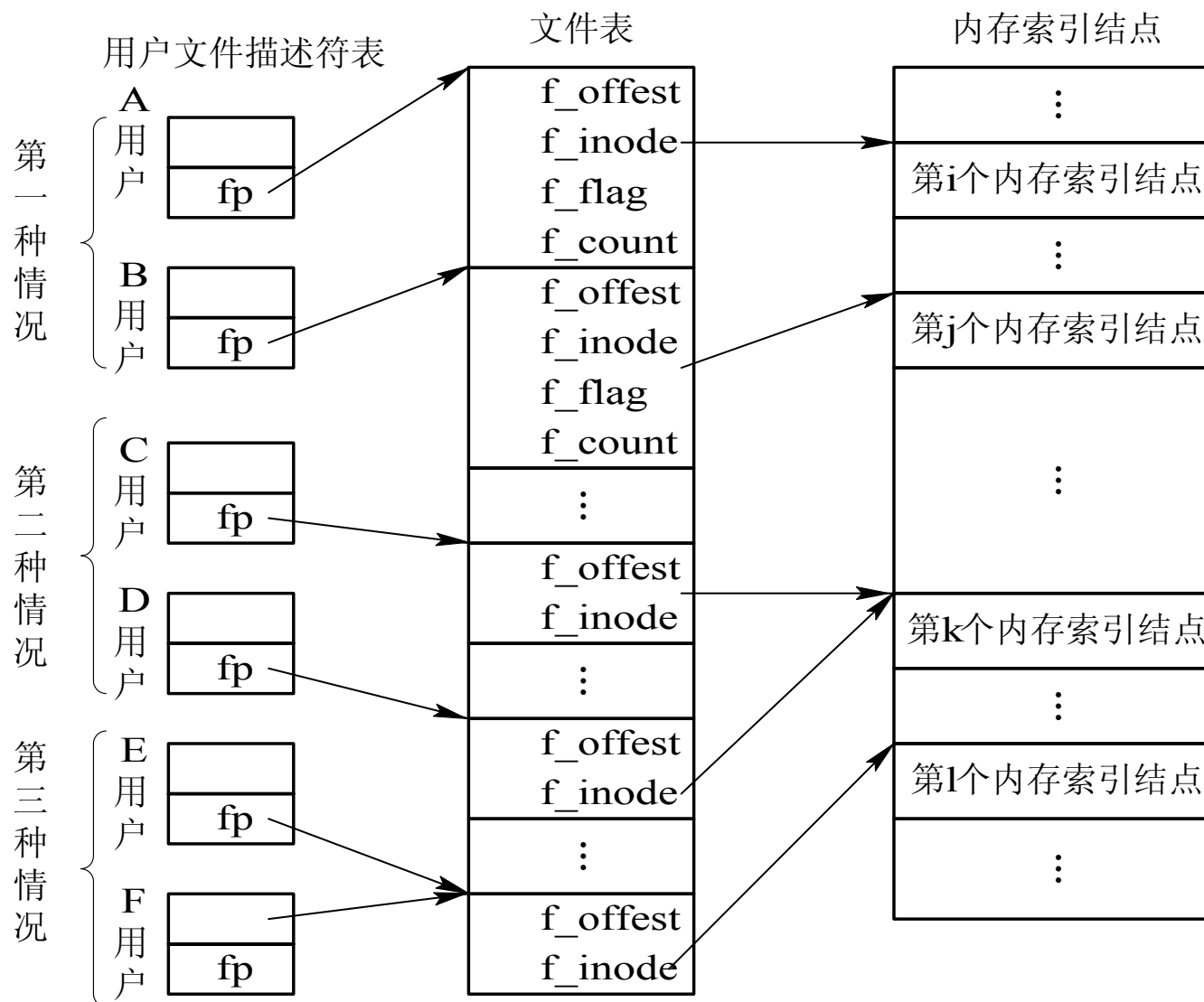
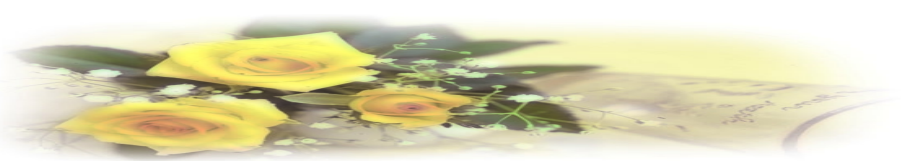


图10-23 对文件的三种读/写方式



第十章 UNIX系统内核结构

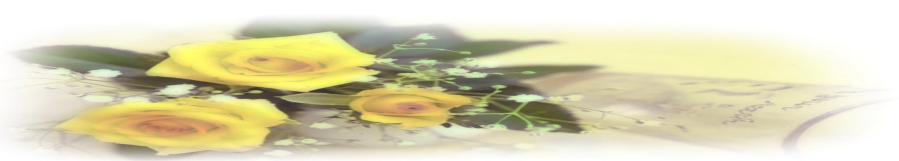
这样，若将读/写指针设置在文件描述符表项中，对前面两种情况固然可行，但要实现对读/写指针的共享就很困难了。为解决此问题，在UNIX系统中引入了文件表，在其表项中存放各用户的读/写指针。这样，对上述三种读/写方式的读/写要求都能很好地满足。在文件表项中，除了读/写偏移量f-offset外，还设置了文件引用计数f-count，用于指示正在利用该文件表项的进程数目，以及用于指向对应内存索引结点的指针f-inode和读/写标志f-flag。





2) falloc过程

该过程的功能是分配文件表项。进入falloc过程后，调用ufalloc过程分配用户文件描述符表项。若未分配成功，便返回NULL；否则，继续从文件表中查找一个空闲文件表项。若找到空闲文件表项，便将该项的始址置入用户文件描述符表项中。在设置完文件描述符表项的初始值后便返回(fp)。若未找到空闲文件表项，则返回NULL。

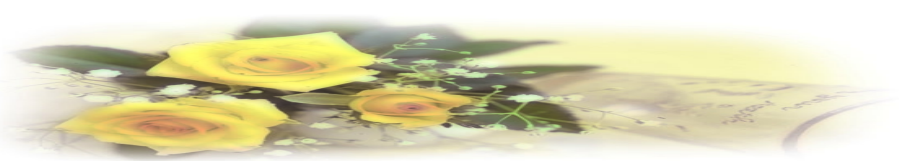




10.6.6 目录管理

1. 构造目录项

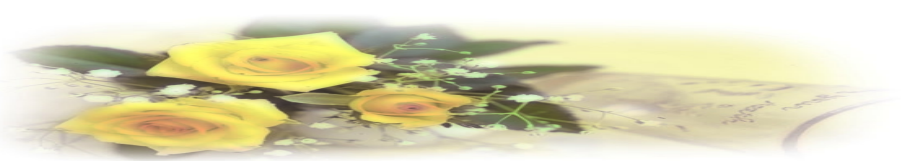
每当某用户(进程)要创建一个新文件时, 内核便应在其父目录文件中为之构造一个目录项; 另外, 当某进程需要共享另一用户的某文件时, 内核也将为要共享该文件的用户建立一个目录项。因此, 当一个共享文件与N个用户相联接时, 该文件将有N个目录项。下面我们介绍在第一种情况下的目录构造过程。





第十章 UNIX系统内核结构

在UNIX系统中，构造目录的任务是由过程makenode完成的。在创建一个新文件时，由系统调用creat过程来为文件构造一个目录项。makenode过程首先调用ialloc过程为新文件分配一个磁盘i结点及内存i结点。若分配失败便返回。当分配成功时，须先设内存i结点的初值(含拷贝)，然后又调写目录过程wdir，将用户提供的文件名与分配给该文件的磁盘i结点号一起，构成一个新目录项，再将它记入其父目录文件中。

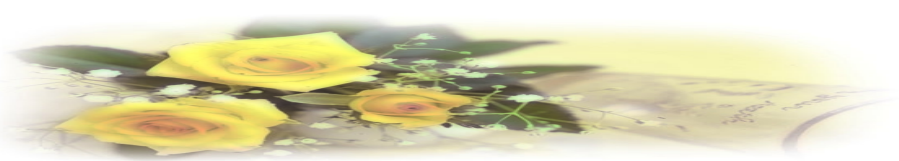




2. 删除目录项

对于一个只由某用户独享的文件，在该用户不再需要它时，应将它从文件系统中删除，以便及时腾出存储空间。

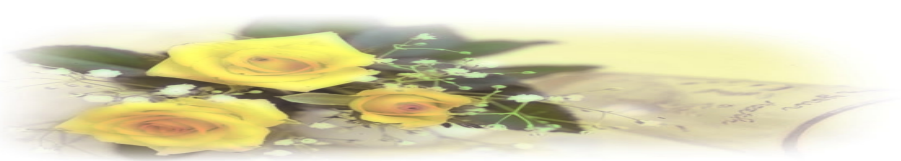
对于一个可供若干个用户(进程)共享的文件，为了表示有多个用户(进程)要共享此文件，内核将利用link系统调用为各用户分别建立一个与该文件的联接，并设置联接计数nlink，使之等于要共享该文件的进程数目。如有6个进程要共享该文件，则应设置nlink为6。对于共享文件，由于该文件在某时刻可能正在被若干个用户所访问，因而不允许任何一个用户(进程，包括文件主)将该文件删除，这也正是UNIX系统中不存在一条用于删除文件的系统调用的原因。





第十章 UNIX系统内核结构

如果某时刻任何一个用户(进程)已不再需要某个共享文件, 则只能利用去联接系统调用`unlink`, 请求系统将本进程与该文件间的联接断开。此时, 系统须对该文件的联接计数`nlink`执行减1操作, 并相应地删除该用户(进程)的一个指名的文件目录。仅当所有联接到该文件上的用户都不再需要该文件时, 其`nlink`值必为0, 系统才执行删除此共享文件的操作。相应地, 将此文件的最后一个目录项从其文件目录中删除。

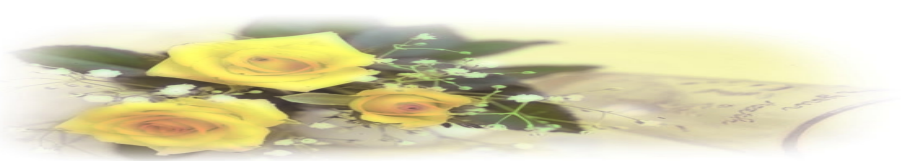




3. 检索目录

在UNIX系统中，用户在第一次访问某文件时，须使用文件的路径名，系统按路径名去检索文件目录，得到该文件的磁盘索引结点，且返回给用户一个文件描述符。以后，用户便利用该文件描述符来访问该文件，这时系统不再去检索文件目录。

对文件目录的检索是由namei过程完成的。该过程可被许多不同的系统调用所调用，如有creat、open、link、chdir、chmod等，但它们对namei的要求并不完全相同。可将这些系统调用分成三类，分别用flag=0、flag=1和flag=2来表示。



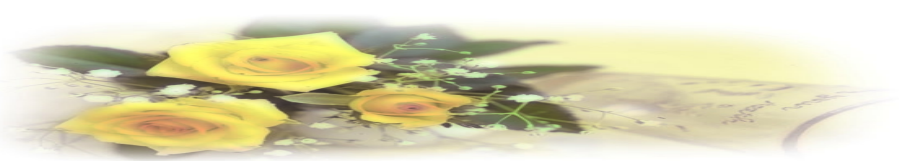


第十章 UNIX系统内核结构

对于flag=0的类，是由chmod、chdir等调用namei；如果namei查找到指名文件，便返回其相应的内存索引结点指针。

对于flag=1的类，主要是由creat调用namei；如果namei未找到指名文件，便做正常返回。

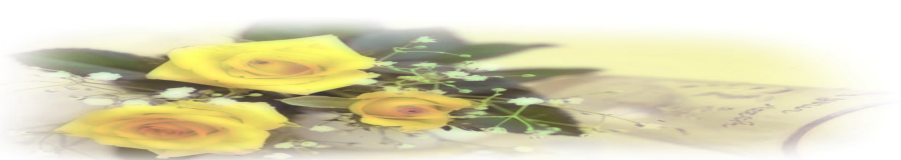
对于flag=2的类，主要是由unlink调用；在正常情况下，应返回由路径名所指出的父目录文件的内存i结点指针。这样就使得namei过程变得非常复杂，成为UNIX系统中最复杂的过程。





第十章 UNIX系统内核结构

检索目录的过程namei是根据用户给出的文件路径名，从高层到低层顺序地查找各级文件目录，寻找指定文件的索引结点号。在检索路径名时，对于以“/”开头的路径名，须从根目录开始检索；否则，应从当前目录开始检索，并把与它对应的i结点作为工作索引结点，然后用文件路径名中的第一分量名与根或当前目录文件中各目录项的文件名逐一进行比较。由于一个目录文件可能占用多个盘块，因此，在检索完一个盘块中的所有目录项而又未找到匹配的文件分量名时，须调用bmap和bread过程，将下一个盘块中的所有目录项读出来后，再逐一检索。若检索完该目录文件的所有盘块而仍未找到匹配的目录项时，才认为无此文件分量名。

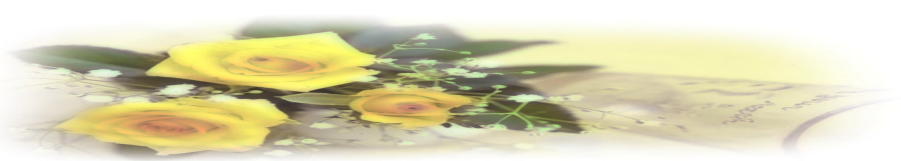




第十章 VMX系统内核结构

在未找到指定分量名的匹配目录项时，可分成以下几种情况处理：

第一种情况：对于最后一个路径分量名(要找的)，若 $flag \neq 1$ ，做出错处理后返回；否则，属于正常情况。此时使 namei 过程继续执行，去检查其父目录文件是否允许写，父目录文件中是否有空目录项，若有，便分配一个空目录项，然后返回NULL。





第十章 VNX系统内核结构

第二种情况：若指定分量名不是最后一个路径分量名，也做出错处理后返回。

第三种情况：如果找到匹配项，便把该目录项中所指示的索引结点作为工作索引结点，并取出文件路径名的下一个分量名，继续重复上述过程，直至路径名中的所有分量名全部查找完毕，最后便可得到指名文件的索引结点。

