操作系统原理与设计

CH1 操作系统概论

1.1 操作系统的定义

1.1.1 操作系统的定义和目标

近四十年来，计算机软件的一个重大进展乃是：操作系统的出现、使用和发展。

计算机系统由两部分组成：硬件和软件。硬件是所有软件运行的物质基础，没有硬件软件便失去了效用;软件能充分发挥硬件潜能和扩充硬件功能，完成各种系统及应用任务，两者互相促进、相辅相成、缺一不可。在软件中，有一种与硬件直接相关，它对硬件作首次扩充和改造，其它软件均要通过它才能发挥作用，在计算机系统中占有特别重要地位的软件，它就是操作系统。计算机发展到今天，从个人机到巨型机，无一例外都配置一种或多种操作系统。

操作系统（Operating System）是管理硬件资源、控制程序执行，改善人机界面，合理组织计算机工作流程和为用户使用计算机提供良好运行环境的一种系统软件。它可被看作是用户和计算机硬件之间的一种接口，是现代计算机系统不可分割的重要组成部分。

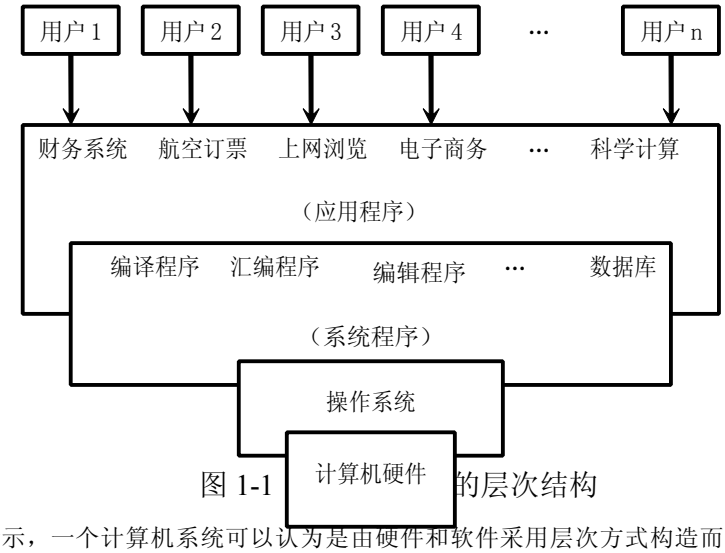


图 1-1 计算机系统的层次结构

如图 1-1 所示，一个计算机系统可以认为是由硬件和软件采用层次方式构造而成的分层系统。其中，每一层具有一组功能并提供相应的接口，接口对层内掩盖了实现细节，对层外提供了使用约定。

硬件层提供了基本的可计算性资源，包括：具有一组指令的处理器、可被访问的寄存器和存储器，可被使用的各种 I/O 设施和设备。这些是操作系统赖以工作的基础，也是操作系统设计者可以使用的功能和资源。**操作系统层**对硬件作首次扩充和改造，提供了操作系统接口，为编译程度、编辑程序、数据库系统等的设计者提供了有力支撑。此外，操作系统还要做资源的调度和分配，信息的存取和保护，并发活动的协调和控制等许多工作。**语言处理层**的工作基础是由操作系统改造和扩充过的机器，提供了许多种比机器指令更强的功能，可较为容易地开发各种各样的语言处理程序。**应用层**解决用户不同的应用问题，应用程序开发者借助于程序设计语言来表达应用问题，开发各种应用程序，既快捷又方便。由此可以看出，操作系统和硬件组成了一个运行平台，其他软件都运行在这个平台上。

综上所述，现代计算机的用户通过应用程序与计算机交互来解决他的应用问题。通常，应用程序用程序设计语言来表达，而不是直接用机器语言来开发。应用程序运行时，除依赖于语言处理程序的支持外，更多地依赖于操作系统提供的各种各样的功能和服务。

计算机系统中配置操作系统的主要目标可归结为：

- 方便用户使用——OS 应该使计算机系统使用起来十分方便。
- 扩大机器功能——OS 应该能改造硬件设施，扩充机器功能。
- 管理系统资源——OS 应该管理好系统中的所有硬件软件资源。
- 提高系统效率——OS 应该使计算机系统的资源得到充分利用，使计算机系统的效率非常高。
- 构筑开放环境——OS 应该构筑出一个开放环境，主要是指：遵循有关国际标准；支持体系结构的可伸缩性和可扩展性；支持应用程序在不同平台上的可移植性和可互操作性。

1.1.2 操作系统的作用

1.1.2.1 OS作为用户与计算机硬件之间的接口

操作系统能扩大机器功能，方便用户使用。它紧靠着硬件并在其基础上提供了许多新的设施和能力，进一步扩充和改进了机器的功能，例如，改造各种硬件设施，使之更容易使用；提供原语或广义指令，扩展机器的指令系统，而这些功能到目前为止还难于由硬件直接实现。操作系统还合理组织计算机的工作流程，协调各个部件有效工作，为用户提供一个良好的运行环境。由于操作系统处于用户和计算机硬件之间，用户总是通过操作系统来使用计算机系统，用户总是在操作系统控制下来运行，因而，可以将操作系统看作是用户和计算机硬件之间的一种接口，用户无需了解硬件和软件的许多细节，却反而可以利用操作系统提供的许多功能，经过操作系统改造和扩充过的计算机不但功能更强，使用也更为方便，用户在操作系统的帮助下能方便、可靠、安全、高效地操纵计算机硬件和运行自己的程序。后面我们将要详细介绍，这种接口主要以系统调用和系统程序两种形式来提供。

1.1.2.2 OS作为计算机系统的资源管理者

在操作系统中，能分配给用户使用的各种硬件和软件设施总称为资源。资源包括两大类：硬件资源和信息资源。硬件资源又分：处理器、存储器、I/O 设备等；信息资源又分：程序和数据等。用户使用计算机，就是使用系统的硬件软件资源来解决应用问题。比如，用户想把一批信息存储到某个设备上，必须先弄清楚该设备信息的存储格式、相应的读写命令和各种情况下的中断处理步骤。而让用户了解设备的物理细节将会十分困难，甚至束手无策。这些工作只能让系统代劳。又如，若内存中能装入两道程序并同时启动它们运行，不但可充分利用内存资源，当一道程序等待 I/O 完成暂不用 CPU 时，可让另一道程序占有 CPU 运行，使得 I/O 设备和 CPU 同时保持忙碌，这类高效的工作方式，需要解决许多技术问题，这只有靠操作系统来做。下面再举第三个例子，假如系统中有三个需要打印输出信息的应用程序在同时运行，那么，打印机上三个程序的输出结果会交错夹杂、混乱不堪，这种做法用户绝对不会欢迎。有一种解决方案：三个程序输出时，先不要直接对打印机操作，而是把各自的结果存在磁盘的暂存区，等到某个程序生成的输出全部存入后，才启动打印机输出，从而，消除了杂乱无章的局面，这种做法只有操作系统才办得到。由于计算机系统中资源种类繁多、数量很大，特性各异，必须加以有效的管理。操作系统将决定处理器在何时、分配给何用户，让它占用多长时间？操作系统将决定 I/O 设备能否分配给申请的用户使用？操作系统将决定多少个用户可同时进入主存，并被启动执行？操作系统将作出合理的调度策略，尽可能提高各种资源的利用率。所以，操作系统的任务之一是有序地管理计算机中的硬件、软件资源，跟踪资源使用状况，满足用户对资源的需求，协调各程序对资源的使用冲突，为用户提供简单、有效的资源使用方法，最大限度地实现各类资源的共享，提高资源利用率，从而，使计算机系统的效率有很大提高。操作系统管理好系统资源，充分发挥它们的作用，不仅仅出于经济上的考虑，更是为了方便用户使用的需要。也有人将操作系统定义为：是能使诸用户有效、方便地共享一套计算机系统资源的一种系统软件。可见，操作系统可以看作计算机系统资源的管理者。

1.1.2.3 OS作为虚拟计算机

许多年以前，人们就认识到必须找到某种方法把硬件的复杂性与用户隔离开来，经过不断的探索和研究，目前采用的方法是在计算机裸机上加上一层又一层的软件来组成整个计算机系统，同时，为用户提供一个容易理解和便于程序设计的接口。

众所周知，裸机是极难使用的，即使提供了很强的指令系统，从功能上来说局限性很大。加上软件之后，就可以在硬件基础上，对其功能和性能进行扩充和完善。至于软件之间的关系，也采用同样办法，一些软件的运行以另外一些软件的存在并为其提供了一定的运行支撑作为基础，而新添加的这些软件是在原来那些软件基础上的扩充和完善。例如，在裸机上加上一层虚拟存储管理软件，用户就可以在这样的空间中编程，要多大存储空间就可以使用多大存储空间，完全不必涉及物理存储空间的容量、地址转换、程序重定位等物理细节。虚拟存储器是现代操作系统对计算机系统中多级物理存储体系进行高度抽象的结果。如果又加上一层 I/O 设备管理软件，用户就可以使用 I/O 命令来进行数据的输入和输出，完全不必涉及显示器、打印机、扫描仪、键盘和鼠标等的物理细节，就可以使用 I/O 设备。如果又加上一层文件管理软件，它将磁盘和其它 I/O 设备抽象成一组命名的文件，用户通过各种文件操作，按文件名来存取信息，完全不必涉及诸如数据物理地址、磁盘记录命令、移动磁头臂 、搜索物理块及设备驱动等物理细节，便于使用、效率又高。如果又加上一层窗口管理软件，由该软件把一台物理屏幕改造成许许多多窗口，每个应用可以在各自的窗口中操作，用户可以在窗口环境中方便地与计算机交互。

在操作系统中，类似地把硬件细节隐藏并把它与用户隔离开来的情况处处可见，例如，中断、时钟等。由此可见，每当在计算机上复盖一层软件，提供了一种抽象，系统的功能便增加一点，使用就更加方便一点，用户可用的运行环境就更加好一点。由于操作系统是紧靠硬件的第一层软件(不排除它自身又是由许多层软件组成的)，所以，当计算机上复盖了操作系统后，便为用户提供了一台功能显著增强，使用更加方便，效率明显提高的机器。可以认为操作系统是建立在计算机硬件平台上的虚拟计算机（Virtual Machine）。

1.1.3 操作系统的主要特性

1.1.3.1 并发性

并发性（Concurrence）是指两个或两个以上的活动在同一时间间隔内发生，而操作系统是一个并发系统，第一个特征是具有并发性，它应该具有处理多个同时性活动的能力。多个 I/O 设备同时在 I/O；设备 I/O 和 CPU 计算同时进行；内存中同时有多个作业被启动交替、穿插地执行，这些都是并发性活动 的例子。发挥并发性能够消除计算机系统中部件和部件之间的相互等待，有效地改善了系统资源的利用率，改进了系统的吞吐率，提高了系统效率。例如，一个程序等待 I/O 完成时，就出让 CPU，而调度另一个程序运行，在程序等待 I/O 时，CPU 便不会空闲，这就是采用了并发技术。但由此引发了一系列的问题，使系统变得复杂化：如何从一个活动切换到另一个活动？怎样将各个活动隔离开来，使之互不干扰，免遭对方破坏？怎样让多个活动协作完成任务？怎样协调多个活动对资源的竞争？如何保证每个活动的资源不被其它进程侵犯?多个活动共享文件数据时，如何保证数据的一致性？操作系统必须具有控制和管 理并发活动的能力，为了更好的解决上述问题，操作系统中很早就引入了一个重要的概念——进程，由于进程能清淅刻划操作系统中的并发性，实现并发活动的执行，因而它已成为现代操作系统的一个重要基础。

采用了并发技术的系统又称为多任务系统(Multitasking)，计算机系统中，并发的实质是一个物理 CPU(也可以多个物理 CPU) 在若干道程序之间多路复用，这样就可以实现程序之间的并发，以及 CPU 与 I/O 设备、I/O 设备与 I/O 设备之间的并行，并发性是对有限物理资源强制行使多用户共享以提高效率。从这里可以看出，实现并发技术的关键之一是如何对系统内的多个活动(进程)进行切换的技术。

1.1.3.2 共享性

操作系统的第二个特征是共享性。共享指操作系统中的资源包括硬件资源和信息资源，可被多个并发执行的进程所使用。出于经济上的考虑，向每个用户分别提供足够的资源不但是浪费的，有时也是不可能的，总是让多个用户共用一套计算机系统资源，因而必然会产生共享资源的需要。可以分成两种资源共享方式：

- 互斥共享：系统中的某些资源如打印机、磁带机、卡片机，虽然它们可提供给多个进程使用，但在同一时间内却只允许一个进程访问这些资源。当一个进程还在使用该资源时，其它欲访 问该资源的进程必须等待，仅当该进程访问完毕并释放资源后，才允许另一进程对该资源访问。这种同一时间内只允许一个进程访问的资源称临界资源，许多物理设备，以及某些数据和表 格都是临界资源，它们只能互斥地被共享。
- 同时访问：系统中的还有许多资源，允许同一时间内多个进程对它进行访问，这里“同时”是宏观上的说法。典型的可供多进程同时访问的资源是磁盘，可重入程序也可被同时共享。

与共享性有关的问题是资源分配、信息保护、存取控制等，必须要妥善解决好这些问题。

共享性和并发性是操作系统两个最基本的特征，它们互为依存。一方面，资源的共享是因为进程的并发执行而引起的，若系统不允许进程并发执行，系统中就没有并发活动，自然就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的管理，必在会影响到进程的并发执行，甚至进程无法并发执行，操作系统也就失去了并发性，导致整个系统效率低下。

1.1.3.3 异步性

操作系统的第三个特点是异步性（Asynchronism），或称随机性。在多道程序环境中，允许多个进程并发执行，由于资源有限而进程众多，多数情况，进程的执行不是一贯到低，而是“走走停停”，例如，一个进程在 CPU 上运行一段时间后，由于等待资源满足或事件发生，它被暂停执行，CPU 转让给另一个进程执行。系统中的进程何时执行？何时暂停？以什么样的速度向前推进？进程总共要多少时间执行才能完成？这些都是不可予知的，或者说该进程是以异步方式运行的，异步性给系统带来了潜在的危险,有可能导致与时间有关的错误,但只要运行环境相同，操作系统必须保证多次运行作业，都会获得完全相同的结果。

操作系统中的随机性处处可见，例如，作业到达系统的类型和时间是随机的；操作员发出命令或按按钮的时刻是随机的；程序运行发生错误或异常的时刻是随机的；各种各样硬件和软件中断事件发生的时刻是随机的等等，操作系统内部产生的事件序列有许许多多可能，而操作系统的一个重要任务是必须确保捕捉任何一种随机事件，正确处理可能发生的随机事件，正确处理任何一种产生的事件序列，否则将会导致严重后果。

1.1.4 操作系统需要解决的主要问题

操作系统具有三大特征：并发性、共享性和异步性，为了解决进程并发执行和资源共享，以及处理随机事件产生时所引起的各种各样的新的矛看，操作系统必须解决好如下问题：

1.1.4.1 提供解决资源冲突的策略和技术

操作系统中，并发进程共享了处理器、存储空间、$^{1}/_{O}$ 设备和软件资源，必须提出资源分配办法和解决资源冲突的各种策略和技术。为用户提供简单、有效的资源使用方法，充分发挥系统资源的利用率。

要研究各类资源的共性和个性，研究资源分配方法和管理策略。经过多年的研究，操作系统中提出了解决资源冲突的一种基本技术--“多重化”（Multiplex），或称“虚拟化”（Virtual）技术，这种技术的基本思想是：通过用一类物理设备来模拟另一类物理设备，或通过分时地使用一类物理设备，把一个物理实体改变成若干个逻辑上的对应物。物理实体是实际存在的，而逻辑上的对应物是虚幻的、感觉上的。例如，在多道程序环境中，虽然只有一个物理 CPU，通过设置进程控制块以及分时使用实际的 CPU，把它虚拟化成多台逻辑上的 CPU，每个用户都认为自己获得了专有 CPU；通过 Spooling(Simultaneous Peripheral Operations On Line)技术，可以用一类物理 $^{I}/_{0}$ 设备来模拟另一类物理设备，“构造”出许多台静态设备供用户使用；通过多路复用技术，可以把一条物理信道虚拟化为若干条逻辑信道，每个用户都认为自己获得专有的信道在进行数据通信；通过虚拟存储技术，把一个统一编址相对较小的物理主存变成多个逻辑上独立编址的虚拟存储器，使得每个用户认为自己获得了硕大无比的编程和运行程序的主存空间。

更有甚者，IBM 公司开发的 VM/370(Virtual Machine/370) 操作系统，它将上述的“多重化”技术发挥到淋漓尽致。虚机器监控程序 VM/370。它向上层提供了若干台虚拟计算机，与传统的操作系统不同的是：这些虚拟计算机不是具有文件管理，设备管理和作业控制之类的虚拟机，而仅仅是实际物理计算机（裸机）的逻辑复制品。其多重化的过程如下：CPU 调度程序使各个进程共享物理 CPU，或者说多重化出许多虚 CPU，每个进程可分得一个；虚存管理使每台虚 CPU 都有自己的虚存空间；SPOOLING 技术和文件系统提供了虚拟读卡机、穿卡机和行式打印机；各个用户的终端通过分时使用处理器时间，提供了虚拟机操作员控制台；每台虚拟机的磁盘是通过划分物理磁盘若干磁道而形成的，称作“小盘”。这样一来，每台复制出来的虚拟计算机包含有：核心态/用户态，中断，CPU、$^{I}/_{O}$ 设备、内存、辅存等，以及物理计算机具有的全部部件。

因为每台虚拟机与一台裸机完全一样，所以，同一台裸机的不同虚拟机上可运行任何操作系统，允许不同虚拟机上运行不同的操作系统而且往往如此。

1.1.4.2 协调并发活动的关系

在操作系统中，有时候一组并发进程协作完成一项任务，而有时候一组并发进程又在竞争某种资源，所以，并发进程之间有一种相互制约的关系。并发进程之间的制约关系必须由系统提供机制或策略来进行协调，以使各个并发进程能顺利推进，并获得正确的运行结果。另外，操作系统还要合理组织计算机工作流程，协调各类硬软件设施工作，充分提高资源的利用率，充分发挥系统的并行性，这些也都是在操作系统的统一指挥和管理下进行的。

1.1.4.3 保证系统的安全性

影响计算机系统安全性的因素很多。首先，是操作系统的安全性，操作系统是一个共享资源系统，支持多用户同时共享一套计算机系统的资源，有资源共享就需要有资源保护，涉及到种种安全性问题。最最基本的有以下三类保护问题：(1) 对操作系统程序的保护，(2) 对系统中的多道程序的保护，(3) 对共享的表格和数据的保护；其次，随着计算机网络的迅速发展，客户机要访问服务器，一台计算机要传送数据给另一台计算机，于是就需要有网络安全和数据信息的保护；另外，在应用系统中，主要依赖数据库来存储大量信息，它是各个部门十分重要的一种资源，数据库中的数据会被广泛应用，特别是在网络环境中的数据库，这就提出了信息系统——数据库的安全性问题；最后计算机安全性中的一个特殊问题是计算机病毒，需要采用措施预防、发现、解除它。上述计算机安全性问题大部份要求操作系统来保证，所以操作系统的安全性是计算机系统安全性的基础。

1.2 操作系统的发展和形成

1.2.1 人工操作阶段

从计算机诞生到五十年代中期的计算机属于第一代计算机，机器速度慢、规模小、外设少，操作系统尚未出现。计算机的操作由程序员采用手工操作直接控制和使用计算机硬件。程序员使用机器语言编程，并将事先准备好的程序和数据穿孔在纸带或卡片上，从纸带或卡片输入机将程序和数据输入计算机。然后，启动计算机运行，程序员可以通过控制台上的按钮、开关和氖灯来操纵和控制程序，运行完毕，取走计算的结果，才轮到下一个用户上机。

随着时间的推移，汇编语言产生了。在汇编系统中，数字操作码被记忆码代替，程序按固定格式的汇编语言书写。系统程序员预先编制一个汇编解释程序，它把用汇编语言书写的“源程序” 解释成计算机能直接执行的机器语言格式的目标程序。稍后，一些高级程序设计语言出现，FORTRAN、ALGOL、和 COBOL 语言分别于 1956、1958、和 1959 年设计完成并投入使用，进一步方便了编程。

执行时需要把汇编解释程序或编译系统以及“源程序” 都穿在卡片或纸带上，然后再装入和执行。其大致过程为：

- 人工把源程序用穿孔机穿制在卡片或纸带上；
- 将准备好的汇编解释程序或编译系统装入计算机；

- 汇编程序或编译系统读入人工装在输入机上的穿孔卡或穿孔带；
- 执行汇编过程或编译过程，产生目标程序，并输出目标卡片迭或纸带；
- 通过引导程序把装在输入机上的目标程序读入计算机；
- 启动目标程序执行，从输入机上读入人工装好的数据卡或数据带；
- 产生计算结果，执行结果从打印机上或卡片机上输出。

上述方式算题比直接用机器语言前进了一步，程序易于编制和易读性好，汇编或编译系统可执行存储分配等辅助工作，一定程序上减轻了用户的负担。但是计算机工作的方式和流程没有多大改变，仍然是在人工控制下，进行装入和执行程序。这种工作方式存在严重缺点：

- 用户一个个、一道道算题，当一个用户上机时，他独占了全机资源，造成计算机资源利用率不高，计算机系统效率低下。
- 许多操作要求程序员人工干预，例如，装纸带或卡片、按开关等等。手工操作多了，不但浪费处理机时间，而且，也极易发生差错。
- 由于数据的输入，程序的执行、结果的输出均是联机进行的，因而，每个用户从上机到下机的时间拉得非常长。

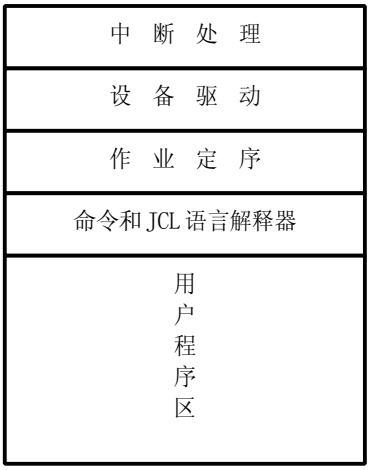
这种人工操作方式在慢速的计算机上还能容忍，随着计算机速度的提高，其缺点就更加暴露出来了。譬如，一个作业在每秒 1 万次的计算机上，需运行 1 个小时，作业的建立和人工干预化了 3 分钟，那么，手工操作时间占总运行时间的 5%；当计算机速度提高到每秒 10 万次，此时，作业运行时间仅需 6 分钟，而手工操作不会有多大变化，仍为 3 分钟，这时手工操作时间占了总运行时间的 50%。由此看出缩短手工操作时间十分必要。此外，随着 CPU 速度迅速提高而 I/O 设备速度却提高不多，导致 CPU 与 I/O 设备之间的速度不匹配，矛盾越来越突出，需要妥然解决这些问题。

1.2.2 管理程序阶段

早期批处理系统中，作业的输入和输出均是联机的，联机 I/O 的缺点是速度慢，I/O 设备和 CPU 仍然串行工作，CPU 时间浪费相当大，为此，在批处理中引进脱机 I/O 技术。除主机外，另设一台辅机，该机仅与 I/O 设备打交道，不与主机连接。输入设备上的作业通过辅机输到磁带上，主机负责从磁带上把作业读入内存执行，作业完成后，主机负责把结果输出到磁带上，然后由辅机把磁带上的结果信息在打印机上打印输出。这样一来，I/O 工作脱离了主机，辅机和主机可以并行工作，称脱机批处理，这比早期联机批处理系统提高了处理能力。

在五十年代末至六十年代初期，硬件又取得了两大进展，中断机构和通道技术出现，硬部件具有了较强的并行工作的能力，为了充分发挥计算机的并行性，提高 CPU 和 I/O 设备的利用率，缩短作业的准备和建立时间，减少人工操作干予，人们研究了驻留在内存的管理程序（Resident Monitor）。FMS（FORTRAN Monitor System）和 IBSYS （IBM 7094 Monitor System）是典型的这类系统。这些系统都配备了专门的计算机操作员，程序员不再直接操作机器，以减少人工干予和操作失误。

在管理程序的控制下，对作业进行自动控制和成批处理，其工作流程如下：操作员集中一批用户提交的作业，由管理程序将这一批作业从纸带或卡片机输入到磁带上，每当一批作业输入完成后，管理程序自动把磁带上的第一个作业装入内存，并把控制权交给作业。当该作业执行完成后，作业又把控制权缴回管理程序，管理程序再调入磁带上的第二个作业到内存执行。计算机在管理程序的控制下就这样连续地一个作业一个作业执行，直至磁带上的作业全部做完。这种系统称为早期批处理系统，也称执行程序，它能实现作业到作业的自动转换，缩短作业的准备和建立时间，减少人工操作和干预，让计算机尽可能地连续工作。



管理程序 Monitor 的内存组织如图 1-2 所示，管理程序控制下的算题过程如图 1-3 所示，它的主要功

能小结如下：

- 自动控制和处理作业流：管理程序把控制传送给一个作业，当作业运行结束时，它又收回控制，继续调度下一个作业执行，自动控制和处理作业流，减少了作业的准备和建立时间。作业流的自动控制和处理依靠作业控制语言，因而，促进了作业控制语言的发展。作业控制语言是由一些描述作业控制过程的语句组成的，每个语句附有一行作业或作业步信息编码，并以穿孔卡的形式提供。例如，Job 卡表示启动一个新作业；FORTRAN 卡表示调用 FORTRAN 编译系统；Load 卡表示调用装配程序；Data 卡指定数据；End 卡表示一个作业结束。管理程序通过输入、解释并执行嵌入用户作业的作业控制卡规定的功能，就能自动地处理用户作业流。每个作业完成后，管理程序又自动地从输入机上读取下一个作业运行，直到整批作业结束。
- 提供一套操作命令：操作员通过打字机打入命令，管理程序识别并执行命令，这样不仅速度快，操作员也可进行一些复杂的控制。输出信息也可由打字机输出，代替了早期氖灯显示，易于理解。这种交互方式不仅提高了效率，也便于使用。
- 提供设备驱动和 I/O 控制功能：系统提供标准 I/O 程序，用户通过管理程序获得和使用 I/O 设备，减轻了用户驱动物理设备的负担。管理程序还能处理某些设备特殊和设备故障，改进了设备的可靠性和可用性。
- 提供库程序和程序装配功能：库程序中包括：汇编程序、FORTRAN 语言编译程序、标准 I/O 程序、标准子程序等，通常，用户程序必须调用库程序才能执行下去，装配工作由管理程序完成。所有程序都按相对地址编址，管理程序把相应库程序和用户程序进行装配，并转换成绝对地址形式的目标程序，以便执行。
- 提供简单的文件管理功能：用个户通过输入设备输入大量的程序和数据，为了反复使用，用户希望能把这些信息保存起来，以便随时使用，这就产生了文件系统。从此，用户可按文件名字，而不是信息的物理地址进行存取，方便灵活，安全可靠。

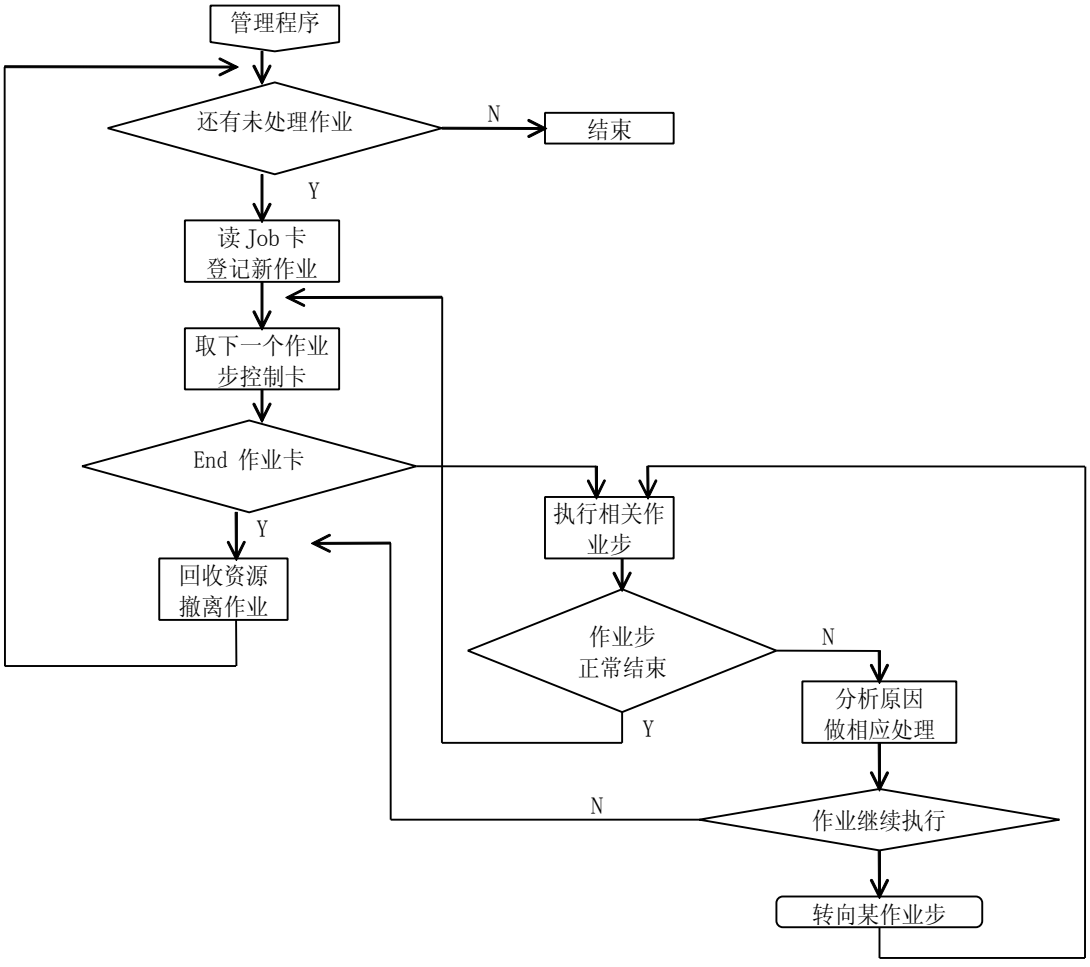


图 1-3　管理程序算题过程

1.2.3 多道程序设计与操作系统的形成

1.2.3.1 多道程序设计

1）多道程序设计的概念

在早期的单道批处理系统中，内存中仅有单作业在运行，致使系统中仍有许多资源空闲，设备利用率低，系统性能较差。如图 1-4 所示，当 CPU 工作时，外部设备不能工作；而外部设备工作时，CPU 必须等待。为了进一步提高资源利用率和系统吞吐率，六十年代中后期，引入了多道程序设计技术，由此形成了多道批处理系统。

多道程序设计(Multiprogramming)是指允许多个程序同时进入一个计算机系统的主存储器并启动进行计算的方法。从宏观上看，多道程序都处于运行过程中，但都未运行完一毕；从微观上看，各道程序轮流占用 CPU，交替地执行。引入多道程序设计技术的根本目的是提高 CPU 的利用率，充分发挥系统部件的并行性。

下面我们来分析多道程序设计技术提高资源利用率和系统吞吐率的原理。从第二代计算机开始，计算机系统具有处理器和外围设备并行工作的能力，这使得计算机的效率有所提高。但是，仅仅这样做，计算机的效率仍不会很高。例如计算某个数据处理问题，要求从输入机(速度为 6400 字符 / 秒)输入 500 个字符，经处理(费时 52 毫秒)后，将结果(假定为 2000 个字符)存到磁带上(磁带机速度为 10 万字符 / 秒)，然后，再读 500 个字符处理，直至所有的输入数据全部处理完毕。如果处理器不具有和外围设备并行工作的能力，那么上述计算过程如图 1-4 所示，不难看出在这个计算过程中，处理器的利用率为：

$$52 /（78 十 52 十 20）\approx 35\%$$

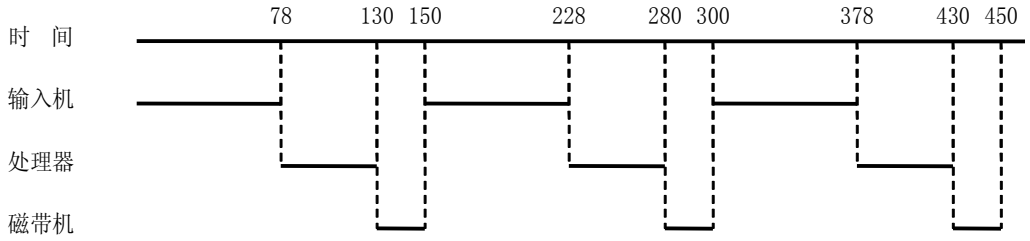


图 1-4 单道算题运行时处理器的使用效率

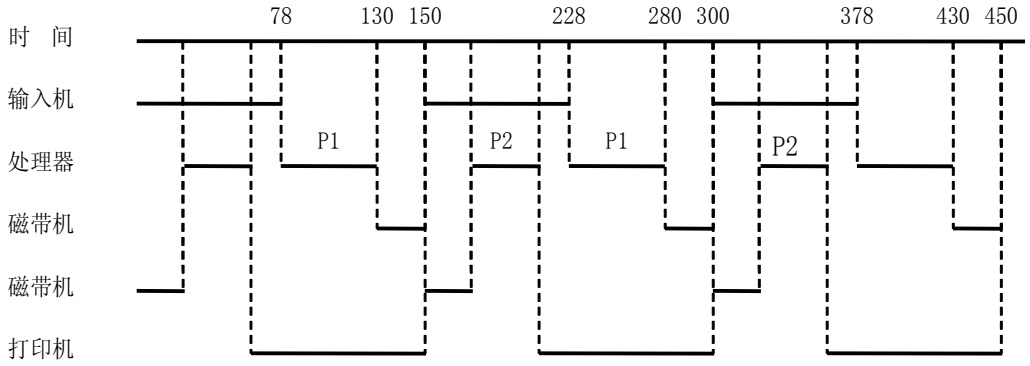


图 1-5 两道算题运行时处理器的使用效率

分析上面的例子，可以看出效率不高的原因，当输入机输入 500 个字符后，处理器只花了 52 毫秒就处理完了，而这时第二批输入数据还要再等 98 毫秒时间才能输入完毕。

上述例子说明单道程序工作时，计算机系统的各部件的利用率没有得到充分发挥。为了提高效率，我们考虑让计算机同时接受两道算题，当第一道程序在等待外围设备的时候，让第二道运行，降低了 CPU 空等时间，那么处理器的利用率显然可以有所提高。例如，计算机在接受上述算题时还接受了另一算题：从另一台磁带机上输入 2000 个字符，经 42 毫秒的处理后，从行式打印机(速度为 1350 行 / 分)上输出两行。

当这两个算题同时进入计算时，这个计算过程如图 3-2 所示。其中，P1 表示程序甲占用 CPU 对输入机输入的 500 个字符进行处理，由于 52 毫秒处理便结束，下次处理要侍 98 毫秒之后，故这个时间段内CPU 是空闲的。系统调度程序乙工作，它从磁带机上输入 2000 个字符后，P2 表示对这批数据进行处理。相应的 I/O 设备和 CPU 的操作都是并行的。不难算出，此时处理器的利用率为：

（52+42）/ 150 ≈ 63%

由此可以看出，让几道程序同时进入计算比一道道串行地进行计算，CPU 效率要高，因为，当某道程序因故不能继续运行下去时，管理程序便把 CPU 分给另外一道程序执行，这样可使 CPU 和 I/O 设备尽量都处于忙碌状态，这就是要采用多道程序设计方法的主要原因。具有处理器和外围设备并行功能的计算机采用多道程序设计的方技术后，可以提高处理器和 I/O 设备的并行性，从而也就能提高整个系统的效率，即增加单位时间内算题的数量。例如有甲、乙两道程序，如果让一道程序独占计算机单道运行时要花去一个小时，而此时处理器的利用率为 30%，粗略地说，甲(或乙)一道程序执行时所需要的处理器时间为：

1 小时×30%＝18 分钟

假定甲、乙两道程序按多道程序设计方法同时运行，处理器的利用率达 50%，那么要提供 36 分钟的处理器时间，大约要运行 72 分钟。所以，粗略地估计，采用多道程序设计技术时只要大约 72 分钟就可以将两道程序计算完毕。然而，由于操作系统本身要花费处理器时间，所以实际花费的时间可能还要长些，例如要花 80 分钟。而单道运行时，甲、乙依次执行完需 120 分钟。因而；采用多道程序设计方法后可以提高效率：

(120—80) / 120≈33%

但是从甲、乙两道程序来看，如果单道运行，它花 60 分钟就可以得到结果，而多道运行时，却要花 80 分钟才有结果，延长了 20 分钟，即延长了 33%的时间。所以，采用多道程序设计方法后，提高了效率，即增长了单位时间的算题量，但是，对于每一道程序来说，却延长了计算时间。所以，多道程序设计技术提高资源利用率和系统吞吐率是以牺牲用户的响应时间为代价的。对于一些实时响应的计算问题，延长 10%的计算时间可能都是难以接受的。因此，这个问题在多道程序设计中必须引起注意。在多道程序设计中，还有一个值得注意的问题是道数的多少。从表面上看，似乎道数越多越能提高效率，但是道数的多少绝不是任意的，它往往由系统的资源以及用户的要求而定。例如：如果上述甲、乙两道程序都要用行式打印机，而系统只有一台行式打印机，那么它们被同时接受进入计算机时，未必能提高效率。因为可能程序甲计算了一段时间后要等程序乙不再使用行式打印机时，即程序乙结束后，才能继续运行。此外，主存储器的容量和用户的响应时间等因素也影响道数的多寡。

下面小结一下操作系统中引入多道程序设计的好处：一是提高了 CPU 的利用率，二是提高了内存和 I/O 设备的利润率，三是改进了系统的吞吐率，四是充分发挥了系统的并行性。其主要缺点是作业周转时间延长。

注意，多道程序设计系统与多重处理系统(Multiprocessing) 有差别，后者是指配置了多个物理 CPU，从而能真正同时执行多道程序。当然要有效地使用多重处理系统，必须采用多道程序设计技术；反过来，多道程序设计不一定要求有多重处理系统支持。多重处理系统的硬件结构可以多种多样，如共享内存的多 CPU 结构、网络连接的独立计计算机结构。虽然多重处理系统增加了硬件，但却换来了提高系统吞吐量、可靠性、计算能力和并行处理能力的好处。

2）多道程序设计的实现

实现多道程序设计必须妥善地解决三个问题：存储保护与程序浮动；处理器的管理和调度，系统资源的管理和调度。

在多道程序设计的环境中，主存储器为几道程序所共享，因此，硬件必须提供必要的手段，使得在主存储器中的各道程序只能访问它自己的区域，以避免相互干扰。特别是当一道程序发生错误时，不致影响其它的程序，更不能影响系统程序，这就是**存储保护**。同时，由于每道程序不是独占全机，这样，不能事先规定它运行时将放在哪个区域，所以，程序员在编制程序时无法知道程序在主存储器的确切地址。甚至，在运行过程中，一个程序也可能改变其运行区域，所有这些，都要求一个程序或某一部分能随机地从某个主存储器区域移动到另一个区域，而不影响其执行，这就是**程序浮动**。

在多道程序设计系统里，如果系统仅配置一个物理处理器，那么多个程序必须轮流占有处理器。为了说明一个程序是否占有或可以占有处理器，我们把程序在执行中的状态分成三种。当一程序正占有处理器运行时，就说它是处于运行状态(运行态)；当一个程序因等待某个事件的发生时，就说它处于等待状态(等待态)；当一个程序等待的条件已满足可以运行而未占用处理器时，则说它处于就绪状态(就绪态)，所以一道程序在执行中总是处于运行、就绪、等待三种状态之一。一道程序在执行过程中，它的程序状态是变化的，从运行态到等待态的转换是在发生了某种事件时产生的。这些事件可能是由于启动外围设备输入输出而使程序要等待输入输出结束后才能继续下去；也可能是在运行中发生了某种故障使程序不能继续运行下去等等。从等待态转换成就绪态也是在发生了某种事件时产生的。例如程序甲处于等待外围设备传输完毕的等待状态，当传输结束时，程序甲就从等待态转为就绪态。从运行态也能转变为就绪态。例如，当程序乙运行时发生了设备传输结束事件，而设备的传输结束，使得程序甲从等待态转变为就绪态；假定程序甲的优先级高于程序乙，因此让程序甲占有处理器运行，这样，程序乙就从运行态转为就绪态。

在多道程序设计系统里，系统的资源为几道程序所共享，上段谈到的处理器就是一例。此外，如主存储器、外围设备以及一些数据等也需要按一定策略去分配和调度，有关的调度算法与实现将在以后各

章叙述。

1.2.3.2 操作系统的形成

第三代计算机的性能有了更大提高，机器速度更快，内外存容量增大，I/O 设备增多，特别是大容量高速磁盘存储器的出现，为软件的发展提供了有力支持。如何更好地发挥硬件功效，如何更好地满足各种应用的需要，这些都迫切要求扩充管理程序的功能。大约到六十年代中期以后，随着多道程序的引入和分时系统、实时系统的出现，标志着操作系统正式形成。

计算机配置操作系统后，其资源管理水平和操作自动化程度有了进一步提高，具体表现在：

- 操作系统实现了计算机操作过程的自动化。批处理方式更为完善和方便，作业控制语言有了进一步发展，除了作业控制卡外，又出现了作业说明书，为优化调度和管理控制提供了新手段。
- 资源管理水平有了提高，实现了外国设备的联机同时操作(即 SPOOLING)，进一步提高了计算机资源的利用率。
- 提供虚存管理功能，由于多个用户作业同时在内存中运行，在硬件设施的支持下，操作系统为多个用户作业提供了存储分配、共享、保护和扩充的功能。
- 支持分时操作，多个用户通过终端可以同时联机地与一个计算机系统交互。
- 文件管理功能有改进，数据库系统开始出现。
- 多道程序设计趋于完善，采用复杂的调度算法，充分利用各类资源，最大限度地提高计算机系统效率。

1.2.4 操作系统发展的主要动力和进一步发展

1.2.4.1 操作系统发展的主要动力

促使操作系统不断发展的主要动力有以下几个方面：

- 器件快速更新换代。微电子技术是推动计算机技术飞速发展的“引擎”，每隔 18 个月其性能要翻一翻。推动微机快速更新换代，它由 8 位机、16 位机发展到 32 位，当前已经研制出了 64 位机，相应的微机操作系统也就由 8 位微机操作系统发展到 16 位、32 位微机系统，而 64 位微机操作系统也在研制。
- 计算体系结构不断发展。硬件的改进导致操作系统的发展的例子很多，内存管理支撑硬件由分页或分段设施代替了界寄存器以后，操作系统中便增加了分页或分段存储管理功能。图形终端代替逐行显示终端后，操作系统中增加了窗口管理功能，允许用户通过多个窗口在同一时间提出多个操作请求。引进了中断和通道等设施后，操作系统中引入了多道程序设计功能。计算机体系结构的不断发展有力地推动着操作系统的发展。例如，计算机由单处理机改进为多处理机系统时，操作系统也由单处理机操作系统发展到多处理机操作系统和并行操作系统；随着计算机网络的出现和发展，出现了分布式操作系统和网络操作系统。随着信息家电的发展，又出现了嵌入式操作系统。
- 提高计算机系统资源利用率的需要。多用户共享一套计算机系统的资源，必须千方百计地提高计算机系统中各种资源的利用率，各种调度算法和分配策略相继被研究和采用，这也成为操作系统发展的一个动力。
- 让用户使用计算机越来越方便的需要。从批处理到交互型分时操作系统的出现，大大改变了用户上机、调试程序的环境；从命令行交互进化到 GUI 用户界面。操作系统的界面还会变得更加友善。
- 满足用户新要求，提供给用户新服务。当用户要求解决实时性应用时，便出现了实操作系统；当发现现有的工具和功能不能满足用户需要时，操作系统往往要进行升级换代，开发新工具，加入新功能。

1.2.4.2 操作系统的进一步发展

随着计算机硬件和体系结构的发展以及随着计算机应用的日益深入广泛而发展，操作系统得到了进一步的发展，目前操作系统的发展可以归结为以下几个方面。

1）微机操作系统的发展

七十年代中期到八十年代初为第一阶段，特点是单用户、单任务微机操作系统。继 CP/M 之后，还有 CDOS（Cromemco 磁盘操作系统）、MDOS（Motorola 磁盘操作系统）和早期 MSDOS（Microsoft 磁盘操作系统）。八十年代以后为第二阶段， 特点是单用户、 多任务和支持分时操作。 以 MP/M、XENIX 和后期 MS-DOS 为代表。

近年来，微机操作系统得到了进一步发展，我们常称为新一代微机操作系统，它们具有以下功能：GUI、多用户和多任务、虚拟存储管理、网络通信支持、数据库支持、多媒体支持、应用编程支持 API。并且还具有以下特点：

- 开放性。支持不同系统互连、支持分布式处理和支持多 CPU 系统。
- 通用性。支持应用程序的独立性和在不同平台上的可移植性。
- 高性能。随着硬件性能提高、64 位机逐步普及、CPU 速度进一步提高，微机操作系统中引进

了许多以前在中、大型机上才能实现的技术，支持多线程，支持对称处理器 SMP，导致计算机系统性能大大提高。

- 采用微内核结构。提供基本支撑功能的内核极小；大部分操作系统功能由内核之外运行的服务器来实现。

由于微型计算机使用量大，微型机操作系统拥有最广泛的用户。

2）并行操作系统的发展

计算机的应用经历了从数据处理到信息处理，从信息处理到知识处理，每前进一步都要求增加计算机的处理能力。为了达到极高的性能，除提高元器件的速度外，计算机系统结构必须不断改进，而这一点主要用采用增加同一时间间隔内的操作数量，通过并行处理（Parallel processing）技术，研究并行计算机来达到的，已经开发出的并行计算机有：阵列处理机、流水线处理机、多处理机。

并行处理技术已成为近年来计算机的热门研究课题，它在气象预报、石油勘探、空气动力学、基因研究、核技术及航空航天飞行器设计等领域均有广泛应用。

为了发挥并行计算机的性能，需要有并行算法、并行语言等许多软件的配合，而并行操作系统则是并行计算机发挥高性能的基础和保证。所以，人们越来越重视并行操作系统的研究和开发。目前已经研究出来的并行操作系统有：美国 Stanford 大学的 V-Kernel、美国 Bell 实验室的 Meglos、美国卡内基梅隆大学的 MACH 等。

3）分布式操作系统的发展

分布式计算机系统（又称分布式系统）是用通信网（广域网、局域网）联接，并用消息传送方进行通信的并行计算机系统。随着微型，小型机的发展，人们开始研究由若干台微、小型机组成的分布式系统，由于它和单计算机的集中式系统相比有坚定性强、可靠性高、容易扩充和价格低廉的优点，因而，越来越受到人们重视，是一个很有前途的发展方向。而分布式系统的控制和管理依赖于分布式操作系统，已经研制出来的分布式操作系统有：Cm *(美国卡内基梅隆大学)，X 树系统(美国加州大学伯克利分校)，Arachne(美国威斯康星大学)，Chorus(法国国家信息与自动化研究所)，Plan9(美国 Bell 实验室)，Amoeba(荷兰自由大学)，Guide(法国 Bull 研究中心)，Clouds(美国乔治亚理工学院)，CMDS(英国剑桥大学)。

4）并行分布式操作系统的发展

超级计算机系统结构的研究重点集中在三个层次上。一是共享存储多处理机和可缩放共享存储多处理机；二是以单处理机、对称多处理机和共享存储多处理机为基本结点的大规模并行计算机；三是连接分布在不同地点的各类同构或异构计算机的计算网格(Computational Grid)，为最终实现全国或全球的元计算(Metacomputing) 打好基础。

为此有许多关键技术需要突破。在并行算法方面，对大规模稀疏矩阵、排序、检索和匹配等问题都急需找到快速有效的算法。针对万亿次量级的系统，如何把问题分解为具有百万路以上的并行性是一个研究重点；在编程环境和编程模式方面，一个挑战性的研究课题是并行可扩展编程环境，重点是并行编译器的优化、调试工具、监测工具的友善性和标准化。还要研究新的编程模式。

上述一切离不开操作系统的支撑，面对上万个处理机和万亿次量级的系统，需要研究和设计可扩展、可移植、开放性、容错性、鲁棒性、易使用、效率高、标准化的并行分布式操作系统，解决全系统范围内的资源分配和管理以及单一系统映象问题。利用自由软件设计自主操作系统将成为趋势。

1.3 操作系统的分类

1.3.1 批处理操作系统

过去，在计算中心的计算机上一般所配置的操作系统采用以下方式工作：用户把要计算的应用问题编成程序，连同数据和作业说明书一起交给操作员，操作员集中一批作业，并输入到计算机中。然后，由操作系统来调度和控制用户作业的执行。通常，采用这种批量化处理作业方式的操作系统称为批处理操作系统（Batch Operating System）。

批处理操作系统根据一定的调度策略把要求计算的算题按一定的组合和次序去执行，从而，系统资源利用率高，作业的吞吐量大。批处理系统的主要特征是：

- 用户脱机工作　用户提交作业之后直至获得结果之前不再和计算机及他的作业交互。因而，作业控制语言对脱机工作的作业来说是必不可少的。这种工作方式对调试和修改程序是极不方便的。
- 成批处理作业　操作员集中一批用户提交的作业，输入计算机成为后备作业。后备作业由批处理操作系统一批批地选择并调入主存执行。
- 多道程序运行　按预先规定的调度算法，从后备作业中选取多个作业进入主存，并启动它们运行，实现了多道批处理。
- 作业周转时间长　由于作业进入计算机成为后备作业后要等待选择，因而，作业从进入计算

机开始到完成并获得最后结果为止所经历的时间一般相当长，一般需等待数小时至几天。

1.3.2 分时操作系统

在批处理系统中，用户不能干预自己程序的运行，无法得知程序运行情况，对程序的调试和排错不利。为了克服这一缺点，便产生了分时操作系统。

允许多个联机用户同时使用一台计算机系统进行计算的操作系统称分时操作系统（Time Sharing Operting System）。其实现思想如下：每个用户在各自的终端上以问答方式控制程序运行，系统把中央处理器的时间划分成时间片，轮流分配给各个联机终端用户，每个用户只能在极短时间内执行，若时间片用完，而程序还未做完，则挂起等待下次分得时间片。这样一来，每个用户的每次要求都能得到快速响应，每个用户获得这样的印象，好像他独占了这台计算机一样。实质上，分时系统是多道程序的一个变种，不同之处在于每个用户都有一台联机终端。

分时的思想于 1959 年由 MIT 正式提出，并在 1962 年开发出了第一个分时系统 CTSS(Compatible Time Sharing System)，成功地运行在 IBM 7094 机上，能支持 32 个交互式用户同时工作。1965 年 8 月 IBM 公司公布了 360 机上的分时系统 TSS/360，这是一个失败的系统，由于它太大太慢，没有一家用户愿意使用。

1965 年在美国国防部的支持下，MIT、BELL 和 GE 公司决定开发一个“公用计算服务系统”，以支持整个波士顿地区所有分时用户，这个系统就是 MULTICS (MULTiplexed Information and Computing Service) 。MULTICS 运行在 GE635、GE645 计算机上使用高级语言 PL/1 编程，约 30 万行代码。特别值得一提的是，MULTICS 引入了许多现代操作系统领域的概念雏形，如分时处理、远程联机、段页式虚拟存储器、文件系统、多级反馈调度、保护环安全机制、多 CPU 管理，多种程序设计环境等，对后来操作系统的设计有着极大的影响。

分时操作系统具有以下特性：

- 同时性：又称多路性，若干个终端用户同时联机使用计算机，分时就是指多个用户分享使用同一台计算机。每个终端用户感觉上好像他独占了这台计算机。
- 独立性：终端用户彼此独立，互不干扰，每个终端用户感觉上好像他独占了这台计算机。
- 及时性：终端用户的立即型请求(即不要求大量 CPU 时间处理的请求)能在足够快的时间之内得到响应。这一特性与计算机 CPU 的处理速度、分时系统中联机终端用户数和时间片的长短密切相关。
- 交互性：人机交互，联机工作，用户直接控制其程序的运行，便于程序的调试和排错。

CTSS(Compatible Time Sharing Syshen)和 Multics(Multiplexed Information and Computing Service)是在 60 年，由 MIT 、Bell 实验室和 GE 公司等开发的分时系统，对后来分时系统的设计有很大影响。当今最流行的一种多用户分时操作系统是 Unix。

分时操作系统和批处理操作系统虽然有共性，它们都基于多道程序设计技术，但存在下列不同点：

- 目标不同，批处理系统以提高系统资源利用率和作业吞吐率为目标；分时系统则要满足多个联机用户的快速响应。
- 适应作业的性质不同，批处理适应已经调试好的大型作业；而分时系统适应正在调试的小型作业。
- 资源使用率不同，批处理操作系统可合理安排不同负载的作业，使各种资源利用率较佳；分时操作系统中，多个终端作业使用相同类型编译系统和公共子程序时，系统调用它们的开销较小。
- 作业控制方式不同，批处理由用户通过 JCL 的语句书写作业控制流，预先提交，脱机工作;交互型作业，由用户从键盘输入操作命令控制，联机工作。

1.3.3 实时操作系统

虽然多道批处理操作系统和分时操作系统获得了较佳的资源利用率和快速的响应时间，从而，使计算机的应用范围日益扩大，但它们难以满足实时控制和实时信息处理领域的需要。于是，便产生了实时操作系统，目前有三种典型的实时系统；过程控制系统、信息查询系统和事务处理系统。计算机用于生产过程控制时，要求系统能现场实时采集数据，并对采集的数据进行及时处理，进而能自动地发出控制信号控制相应执行机构，使某些参数（压力、温度、距离、湿度）能按予定规律变化，以保证产品质量。导弹制导系统，飞机自动驾驶系统，火炮自动控制系统都是实时过程控制系统。计算机还可用于控制进行实时信息处理，情报检索系统是典型的实时信息处理系统。计算机接收成千上百从各处终端发来的服务请求和提问，系统应在极快的时间内做出回答和响应。事务处理系统不仅对终端用户及时作出响应，而且要对系统中的文件或数据库频繁更新。例如，银行业务处理系统，每次银行客户发生业务往来，均需修改文件或数据库。要求这样的系统响应快、安全保密，可靠性高。

实时操作系统（Real Time Operating System)是指当外界事件或数据产生时，能够接收并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制监控的生产过程或对处理系统作出快速响应，并控制所有实行任务协调一致运行的操作系统。由实时操作系统控制的过程控制系统，较为复杂，通常

由四部分组成：

- 数据采集：它用来收集、按收和录入系统工作必须的信息或进行信号检测。
- 加工处理：它对进入系统的信息进行加工处理，获得控制系统工作必须的参数或作出决定，然后，进行输出，记录或显示。
- 操作控制：它根据加工处理的结果采取适当措施或动作，达到控制或适应环境的目的。
- 反馈处理：它监督执行机构的执行结果，并将该结果馈送至信号检测或数据按收部件，以便系统根据反馈信息采取进一步措施，达到控制的予期目的。

在实时系统中通常存在若干个实时任务，它们常常通过“队列驱动” 或“事件驱动” 开始工作，当系统接受来自某些外部事件后，分析这些消息，驱动实时任务完成相应处理和控制。可以从不同角度对实时任务加以分类。按任务执行是否呈现周期性可分成：周期性实时任务和非周期性实时任务；按实时任务截止时间可分成：硬实时任务和软实时任务。

1.3.4 网络操作系统

计算机网络是通过通信设施将地理上分散并具有自治功能的多个计算机系统互连起来，可互操作协作处理的系统。它具有三个主要组成部分：

- 若干台主机：通常它们之间要求交换和共享信息，每台机器是自治的各自独立工作。
- 通信子网：它由通信链路和通信处理机组成，用于数据通信。
- 通信协议：网络中传送数据必须遵守的约定和规则，通信协议使网络中计算机能协同工作。

为了使网络中的计算机能方便地传送信息和共享网络资源而加到网络中的计算机上的操作系统称网络操作系统（Network Operating System）。

网络操作系统具有两种工作模式：第一种是客户机-服务器(Client-Server) 模式，这类网络中分成两类站点，一类作为网络控制中心或数据中心的服务器，提供文件打印、通信传输、数据库等各种服务；另一类是本地处理和访问服务器的客户机。这是目前较为流行的工作模式。另一种是对等(Peer-to-Peer) 模式，这种网络中的站点都是对等的，每一个站点既可作为服务器，而又可作为客户机。网络操作系统应该具有以下几项功能：

- 网络通信：其任务是在源计算机和目标计算机之间，实现无差错的数据传输。具体来说完成建立/拆除通信链路、传输控制、差错控制、流量控制、路由选择等功能。
- 资源管理：对网络中的所有硬、软件资源实施有效管理，协调诸用户对共享资源的使用，保证数据的一致性、完整性。典型的网络资源有：硬盘、打印机、文件和数据。
- 网络管理：包括安全控制、性能监视、维护功能等
- 网络服务：如电子邮件、文件传输、共享设备服务、远程作业录入服务等

目前，计算机网络操作系统有三大主流：Unix、Netware 和 Windows。Unix 是唯一能跨多种平台的操作系统；Windows NT 工作在微机和工作站上；Netware 则主要面向微机。支持 C/S 结构的微机网络操作系统则主要有：Netware、Unix ware、Windows NT、 LAN Manager 和 LAN Server 等。

下一代网络操作系统应能提供以下功能支撑：

- 位置透明性　支持客户机、服务器和系统资源不停地在网络中装入卸出，且不固定确切位置的工作方式。
- 名空间透明性　网络中的任何实体都必须从属于同一个名空间。
- 管理维护透明性 如果一个目录在多台机器上有映象，应负责对其同步维护；应能将用户和网络故障相隔离；同步多台地域上分散的机器的时钟。
- 安全权限透明性 用户仅需使用一个注册名及口令，就可在任何地点对任何服务器的资源进行存取，请求的合法性由操作系统验证，数据的安全性由操作系统保证。
- 通信透明性　提供对多种通信协议支持，缩短通信的延时。

1.3.5 分布式操作系统

以往的计算机系统中，其处理和控制功能都高度地集中在一台计算机上，所有的任务都由它完成，这种的系统称集中式计算机系统。而分布式计算机系统是指由多台分散的计算机，经互连网络连接而成的系统。每台计算机高度自治，又相互协同，能在系统范围内实现资源管理，任务分配、能并行地运行分布式程序。通常分布式计算机系统满足以下条件：

- 系统中任意两台计算机可以通过系统的安全通信机制来交换信息。
- 系统中的资源为所有用户共享，用户只要考虑系统中是否有所需资源，而无需考虑资源在哪台计算机上。
- 系统中的若干台机器可以互相协作来完成同一个任务，换句话说，一个程度可以分布于几台计算机上并行运行，一般的网络是不满足这个条件的，所以，分布式系统是一种特殊的计算机网络。
- 系统中的一个结点出错不影响其它结点运行、即具有较好的容错性和健壮性。

从上面叙述可以看出，分布式操作系统(Distributed Operating System)应该具备四项基本功能：

- 进程通信：提供有力的通信手段，让运行在不同计算机上的进程可通过通信来交换数据；
- 资源共享：提供访问它机资源的功能，使得用户可以访问或使用位于它机上的资源,例如 A 结点上的一个进程使用 B 结点上的一台打印机，而 B 结点上的一个进程却在存取 A 结点上的一个文件。
- 并行运算：提供某种程度设计语言，使用户可编写分布式程序，该程序可在系统中多个节点上并行运行；
- 网络管理：高效地控制和管理网络资源，对用户具有透明性、即使用分布式系统与传统单机相似。

分布式计算机系统的主要优点是：坚定性强、扩充容易、可靠性好、维护方便和效率较高。

用于管理分布式计算机系统的操作系统称分布式操作系统（Distrabuted Operatying System）。它与单机的集中式操作系统的主要区别在于资源管理，进程通信和系统结构三个方面。和计算机网络类似，分布式操作系统中必须有通信规程，计算机之间的发信按规程进行。分布式系统的通信机构、通信规程和路径算法都是十分主要的研究课题。集中式操作系统的资源管理比较简单，一类资源由一个资源管理程序来管。这种管理方式不适合于分布式系统，例如，一台机器上的文件系统来管理其它计算机上的文件是有困难的。所以，分布式系统中，对于一类资源往往有多个资源管理程序，这些管理者必须协调一致的工作，才能管好资源。这种管理比单个资源管理程序的方式复杂得多，人们已开展了许多研究工作，提出了许多分布式同步算法和同步机制。分布式操作系统的结构也和集中式操作系统不一样，它往往有若干相对独立部分，各部分分布于各台计算机上，每一部分在另外的计算机上往往有一个副本，当一台机器发生故障时，由于操作系统的每个部分在它机上有副本，因而，仍可维持原有功能。

Plan9 是由 AT&T 公司的 BELL 实验室于 1987 年由 Ken Thompson(Unix 设计者之一) 参与开发的一个具有全新概念的分布式操作系统。开发 Plan9 的主要动机是为解决 Unix 系统日趋庞大，在可移植性、可维护性和对硬件环境的适应性方面所遇到的种种困难。小而精的思想贯彻到整个系统谍设计中，整个系统仅有约 15000 行 C 源代码，是小而精思想的新体现。Plan9 是运行在由不同网络联接 CPU 服务器、文件服务器及终端机的分布式硬件上的一个分布式操作系统。Plan9 实现中引入和使用了以下概念和技术：命名空间(Naming Space) 、进程文件系统(Process File System) 、窗口系统(Window System) 、CPU 命令(CPU Command) 等。

分布式操作系统 Amoeba 由荷兰自由大学和数学信息科学中心联合研制。Amoeba 的基本思想是：用户就象使用传统的计算机那样来使用 Amoeba、即用户不知道他正在用那些机器？用了几台？是那几台？用户也不知道文件存在那台机器上？共有几个备份？其主要目标是：进行分解式系统的研究，建立一个良好的试验平台，以便在上面进行算法、语言、和应用的试验。Amoeba 将网络中的机器分成若干个组，包括 CPU 池组、工作站组、专用服务器组，通过一专用的 Gateway 将局域网联到全局网中构成 Amoeba 的硬件体系结构。Amoeba 的微内核具有四项基本功能：管理进程和线程，支持底层内存管理；支持线程间的透明通信；实现 I/O 处理。Amoeba 的服务功能由下列服务程序实现：快速文件服务程序，目录服务程序，监控服务程序等。

分布式系统研究和开发的主要方向有：

- 分布式系统结构：研究非共享通路结构和共享通路结构。
- 分布式操作系统：研究资源管理方法、同步控制机制、死锁的检测和解除和进程通信模型及手段等。
- 分布式程序设计：扩充顺序程序设计语言使其具有分布程序设计能力；开发新的分布式程序设计语言。
- 分布式数据库：设计开发新的分布式数据库。
- 分布式应用 ：研究各种分布分式并行算法，研究在办公自动化、自动控制、管理信息系统 等各个领域的应用。

1.3.6 嵌入式操作系统

随着计算机技术、通信技术为主的信息技术的快速发展和 Internet 网的广泛应用，3C（Computer ，Communication ，Consumer Electronics）合一的趋势已初露端倪，计算机是贯穿信息社会的核心技术，网络和通信是信息社会赖以存在的基础设施，消费电子是人与信息社会的主要接口。3C 合一的必然产物是信息电器；同时，计算机的微型化和专业化趋势已成事实，在这些领域内部产生了一个共同需求：嵌入式软件，嵌入式操作系统 (Embedded Operating System)是嵌入式软件的基本支撑。随着信息电器和信息产业的迅速发展，面对巨大的生产量和用户量，嵌入式软件和嵌入式操作系统的应用前景十分广阔。

国外公司已于几年前开始投入嵌入式软件开发，至今已有几十种嵌入式操作系统面世，嵌入式应用软件更是丰富多彩。具有代表性的嵌入式操作系统有：chorus (Chorus 公司), Diba (Sun 公司), Navio (Oracle 公司), Os-9(Microsoft 公司)， Psos(ISI 公司)，QNX（QSSL 公司），VxWork(WindRiver 公司)和 Win CE (Microsoft 公司)。其中，VxWorks 嵌入式操作系统被美国火星探险计划使用。我国中科院北京软件工程研究中心已研制出具有自主版权的嵌入式操作系统 HOPEN。

一般说，嵌入式软件具有以下特点：（1）微型化　由于嵌入式软件运行的信息装置（平台）无多少可用内存，更没有可用外存，因而，微型化是嵌入式软件的重要特点，（2）专业化　嵌入式软件运行的平台多种多样，应用更是五花八门，因而，嵌入式软件表现出专业化的特点。（3）实时性　嵌入式软件广泛应用于过程控制，数据采集、通信、多媒体信息等要求迅速响应的场合，因而，实时性成为嵌入式软件的又一特点。

由嵌入式软件的特性可以看出，嵌入式操作系统将向微内核、模块化、Java 虚拟机技术、多任务多线程方向发展；嵌入式应用软件将向模块化、专门化、多样化和简易化方向发展。

嵌入式操作系统的结构大致可以分成以下几个层次：与硬件相关的底层软件、操作系统内核(文件系统、内存管理、设备管理、进程管理和中断处理) 、API、图形界面、通信协议、标准化浏览器等。

Windows CE 是微软开发的，用于通信、娱乐和移动式计算设备的操作系统(平台) ，它是微软“维纳斯” 计划的核心。CE 是具有开放的结构设计，32 位多任务、多线程操作系统。 Personal Java 是 SUN 公司开发的一种用于给家庭、办公室和移动信息电器创建连网应用而专门开发的 Java 应用环境(Java Application Environment) 。它继承了 Java 产品家属跨硬件平台的优秀特性，非常适宜更新换快的信息电器的应用开发。同时，SUN 公司又开发出专门用于信息电器应用开发的实时操作系统 Java OS for Consumers 和适用于存储空间有限的专用实时操作系统 Embedded Java。 Hopen 是由中科院凯思软件集团开发的嵌入式操作系统(又称“女娲” 操作系统) 。Hopen 是一个微内核结构的多任务可抢占实时操作系统，核心程序约占 10kb，用 C 语言编写。主要特点有：单用户多任务、支持多进程多线程、提供多种设备驱动程序、图形用户界面、Win32API、支持 Gb2312-80 字符集(汉字) 。Hopen 支持面向信息电器产品的 Personal Java 应用环境，可以开发在机顶盒、媒体电话、汽车导航器、嵌入式工控设备、联网服务等许多方面的应用。

1.4 操作系统的功能

操作系统的主要职责是管理硬件资源、控制程序执行，改善人机界面，合理组织计算机工作流程和为用户使用计算机提供良好的运行环境。计算机系统的主要硬件资源有处理机、存储器、I/O 设备；信息资源有程序和数据，它们又往往以文件形式存放在外存储器上，所以，从资源管理和用户接口的观点来看操作系统具有以下主要功能。

1.4.1 处理机管理

处理器管理的第一项工作是处理中事断件，处理器硬件只能发现中断事件，捕捉它并产生中断信号，但不能进行处理。配置了操作系统，就能对中断事件进行处理，这是最基本的功能之一。

处理器管理的第二项工作是处理器调度。在单用户、单任务的情况下，处理器仅为一个用户或任务所独占，对处理器的管理就十分简单。但在多道程序或多用户的情况下，组织多个作业或任务执行时，就要解决对处理器的分配调度、分配和回收资源等问题，这些是处理器管理要做的重要工作。近年来设计出各种各样的多处理器系统，处理器的管理就更加复杂。为了较好的实现处理器管理功能，一个非常重要的概念进程（process）引入操作系统，处理器的分配和执行都是以进程为基本单位；随着分布式系统的发展，为了进一步提高系统并行性，使并发执行单位的粒度变细，又把线程（Thread）概念引入操作系统。因而，对处理器的管理可以归结对进程和线程的管理，包括：

- 进程控制和管理
- 进程同步和互斥
- 进程通信
- 进程死锁
- 处理器调度，又分作业调度，中程调度，进程调度等
- 线程控制和管理

正是由于操作系统对处理器的管理策略不同，其提供的作业处理方式也就不同，例如，批处理方式、分时处理方式、实时处理方式等等。从而，呈现在用户面前，成为具有不同性质和不同功能的操作系统。

1.4.2 存储管理

存储管理的主要任务是管理存储器资源，为多道程序运行提供有力的支撑。存储管理将根据用户需要给它分配存储器资源；尽可能地让主存中的多个用户实现存储资源的共享，以提高存储器的利用率；能保护用户存放在存储器中的信息不被破坏，要把诸用户相互隔离起来互不干扰，更不允许用户程序访问操作系统的程序和数据；由于物理内存容量有限，难于满足用户的需求，应该能从逻辑上来扩充内存储器，为用户提供一个比内存实际容量大得多的编程空间，方便用户的编程和使用。为此，存储管理具有四大功能：

- 存储分配
- 存储共享

- 存储保护
- 存储扩充

操作系统的这一部分功能与硬件存储器的组织结构和支撑设施密切相关，操作系统设计者应根据硬件情况和使用需要，采用各种相应的有效存储资源分配策略和保护措施。

1.4.3 设备管理

设备管理的主要任务是完成用户提出的 I/O 请求，为用户分配 I/O 设备；加快 I/O 信息的传送速度，发挥 I/O 设备的并行性，提高 I/O 设备的利用率；以及提供每种设备的设备驱动程序和中断处理程序，使用户不必了解硬件细节就能方便地使用 I/O 设备。为了实现这些任务，设备管理应该具有以下功能：

- 缓冲管理
- 设备分配
- 设备驱动
- 设备独立性
- 实现虚拟设备

1.4.4 文件管理

上述三种管理是针对计算机硬件资源的管理。文件管理则是对系统的信息资源的管理。在现代计算机中，通常把程序和数据以文件形式存储在外存储器上，供用户使用，这样，外存储器上保存了大量文件，对这些文件如不能很好的管理，就会导致混乱或破坏，造成严重后果。为此，在操作系统中配置了文件管理，它的主要任务是对用户文件和系统文件进行有效管理，实现按名存取；实现文件的共享、保护和保密，保证文件的安全性；并提供给用户一套能方便使用文件的操作和命令。具体来说，文件管理要完成以下任务：

- 提供文件逻辑组织方法
- 提供文件物理组织方法
- 提供文件的存取方法
- 提供文件的使用方法
- 实现文件的目录管理
- 实现文件的存取控制
- 实现文件的存储空间管理

1.4.5 网络与通信管理

计算机网络源于计算机与通信技术的结合, 近二十年来, 从单机与终端之间的远程通信, 到今天全世界成千上万台计算机联网工作, 计算机网络的应用已十分广泛。联网操作系统至少应具有以下管理功能：

- 网上资源管理功能　计算机网络的主要目的之一是共享资源网计算机上必须配置网络操作系统, 具体来说, 网络操作系统具有以下功能:, 实现网上资滚源的共享, 管理用户应用程序对资源的访问, 保证信息资源的安全性和一致性。
- 数据通信管理功能　计算机联网后, 站点之间可以互相传送数据, 进行通信, 通过通信软件, 按照通信协议的规定, 完成网站之间的信息传送。
- 网络管理功能　包括: 故障管理、安全管理、性能管理、记帐管理和配置管理。

1.4.6 用户接口

上面已经叙述了操作系统对资源的管理，对四大类资源，提供了四种管理。除此之外，为了使用户能灵活、方便地使用计算机和操作系统，操作系统还提供了一组友好的用户接口，这也是操作系统的另一个重要功能。我们将在下一小节再作介绍。

1.5 操作系统提供的服务和用户接口

1.5.1 操作系统提供的基本服务

操作系统要为用户程序的执行提供一个良好的运行环境，它要为程序及其用户提供各种服务，当然不同的操作系统提供的服务不完全相同，但有许多是共同的。提供操作系统共性服务为程序员带来了方便，使编程任务变得更加容易，操作系统提供给程序和用户的共性服务大致有：

- 创建程序：提供各种工具和服务，如编辑程序和调试程序，帮助用户编程并生成高质量的源程序。
- 执行程序：将用户程序和数据装入主存，为其运行做好一切准备工作并启动它执行。当程序编译或运行执行出现异常时，应能报告发生的情况，终止程序执行或进行适当处理。
- 数据 I/O：程序运行过程中需要 I/O 设备上的数据时，可以通过 I/O 命令或 I/O 指令，请求操作系统的服务。操作系统不允许用户直接控制 I/O 设备，而能让用户以简单方式实现 I/O 控制和读写数据。
- 信息存取：文件系统让用户按文件名来建立、读写、修改、删除文件，使用方便，安全可靠。当涉及多用户访问文件时，操作系统将提供信息保护机制。
- 通信服务：在许多情况下，一个进程要与另外的进程交换信息，这种通信发生在两种场合，一是在同一台计算机上执行的进程之间通信；二是在被网络连接在一起的不同计算机上执行 的进程之间通信。进程通信可以借助共享内存(Shared Memory) 实现，也可以使用消息传送(Message Passing) 技术实现。采用前一种方法，操作系统要让两个进程连结到共享存储区;采用后一种方法，操作系统完成消息在进程之间的移动。
- 错误检测和处理：操作系统能捕捉和处理各种硬件或软件造成的差错或异常，并让这些差错或异常造成的影响缩小在最小的范围内，必要时及时报告给操作员或用户。

此外，操作系统除上述提供给用户的服务外，还具有另外一些功能，以保证它自身高效率、高质量地工作，从而，能使多个用户有效地共享系统资源，提高系统效率，这些功能有：

- 资源分配：多个用户或多道作业同时运行时，每一个必须获得系统资源。系统中的各类资源均由操作系统管理，如 CPU 时间、主存资源、文件存储空间等，都配有专门的分配程序，而其它资源(如 I/O 设备) 配有更为通用的申请与释放程序。例如，怎样才能最好地利用 CPU？操作系统配有 CPU 调度例行程序，专门关注 CPU 的速度、必须被执行的作业、可用的寄存器数及其它因素。有一个例行程序探查未被使用的磁带驱动器，并且标记在内部表格，以便把它分给新用户。这些例行程序也可用来分配绘图仪、调制/解调器和其它外围设备。
- 统计：我们希望知道用户使用计算机资源的情况，如用了多少？什么类型？以便用户付款或简单地进行使用情况统计，可以作为进一步改进系统服务，对系统进行重组合的有价值的工具。
- 保护：在多用户多任务计算机系统中，文件所有者能对其创建的文件进行控制使用，保护意味着要确保对系统资源的所有存取要受到控制。用户对各种资源的需求经常发生冲突，为此，操作系统必须做出合理的调度。

1.5.2 操作系统提供的用户接口

从系统的角度来说，操作系统的服务和功能可以不同方式提供给用户，最基本的有两种：

- 系统调用(System Call) 是由操作系统实现完成某种功能的过程。它是程序与操作系统的接口，又称程序接口或编程级接口，在编写的程序中使用“系统调用” 就可以获得操作系统的底层服务，访问系统的各种软硬件资源。
- 系统程序(System Program) 可以看作是操作系统提供给用户的功能级接口。它是操作系统为用户提供的解决使用计算机和计算的共性问题所有服务的集合。这可以当作操作系统高层提 供的服务(虽然它们不能算作操作系统的一部分)，用户常常通过操作命令来调用系统程序。

1.5.2.1 系统调用

1）系统调用的分类

程序接口又称应用编程接口 API(Application Programming Interface) ，是为用户程序或其它系统程序在执行过程中访问系统资源，调用操作系统功能而建立的，是用户程序获得操作系统服务的唯一途径。系统调用也称为广义指令，每一个系统调用都是一个由操作系统实现的，能完成特定功能的过程或子程序。系统调用是由访管指令实现的，用户程序中编写访管指令，便可以调用操作系统的特定功能。国际标准化组织给出的”基于 Unix 的可移植操作系统接口”有关系统调用的国际标准 POSIX1003．1 (Portable Operating System IX)，这个标准指定了系统调用的功能，但未明确规定以什么形式表现，是系统调用、库函数，还是其它形式。目前许多操作系统都有完成类似功能的系统调用，至于在细节上的差异就比较大了。早期操作系统的系统调用使用汇编语言编写，这时的系统调用也是汇编指令,因而,能在汇编语言编程中直接使用；而在高级语言中，如早期 Unix,往往提供了一个与各系统调用一一对应的库函数，因而，应用程序或其它系统程序可通过对应的库函数来使用系统调用。最新推出的一些操作系统，如 Unix 新版本、Windows 和 OS2 等，其系统调用干脆用 C 语言编写，并以库函数形式提供，故在用 C 语言编制的程序中，可直接使用系统调用(其它许多操作系统却不能)。系统调用大致可分成五类：

- 进程和作业管理 终止或异常终止进程、装入和执行进程、创建和撤销进程、获取和设置进程属性、修改进程状态、分配和回收进程存区。
- 文件操作 建立文件、删除文件、打开文件、关闭文件、读写文件、获得和设置文件属性。
- 设备操作 申请设备、释放设备、设备 I/O 和重定位、获得和设置设备属性逻辑上连接和释放

设备。

- 信息维护　获取和设置曰期及时间、获得和设置系统数据、
- 通信　建立和断开通信连接、发送和接近消息、传送状态信息、联接和断开远程设备。

Windows 通过三个组件来支持 API:：Kernel、User 和 GDI。Kerenl 包含了大多数操作系统函数，如内存管理、进程管理;User 集中了窗口管理函数，如窗口创建、撤销、移动、对话及各种相关函数;GDI 是提供画图函数、打印函数。所有应用程序都共享这三个模块的代码，每个 Windows 的 API 函数都可通过名字来访问，具体做法是：应用程序中使用函数名，并用适当的函数库进行编译和链接，然后，应用程序便可运行。实际上 Windows 将三个组件置于动态链接库 DLL(Dynamic Link Library)中，DLL 是可作为二进制映像可链接的、可调用的函数集，Windows 在许多场合都使用 DLL 和一种允许应用程序调用 DLL 的”动态链接”技术。

2）系统调用的实现要点

每个操作系统都提供几十到几百条系统调用，Unix 提供了五十余条，按功能大致可以分成：进程管理、文件管理、存储管理、资源申请、进程通信、系统控制等类别。

在操作系统中，为控制和实现系统调用的机制称陷入或异常处理机制，相应地由于系统调用而引起处理器中断的机器指令称访管指令(Supervisor) ，陷入指令(Trap) 或异常中断指令(Interupt) 。在操作系统中，每个系统调用都事先规定了编号，称功能号，在访管或陷入指令中必须指明对应系统调用的功能号，在大多数情况下，还附带有传递给内部处理程序的参数。

系统调用的实现有以下几点：一是编写系统调用处理程序；二是设计一张系统调用入口地址表，每个入口地址都指向一个系统调用的处理程序，有的系统还包含系统调用自带参数的个数；三是陷入处理机制需开辟现场保护区，以保存发生系统调用时的处理器现场。图 1-6 是系统调用的处理过程。

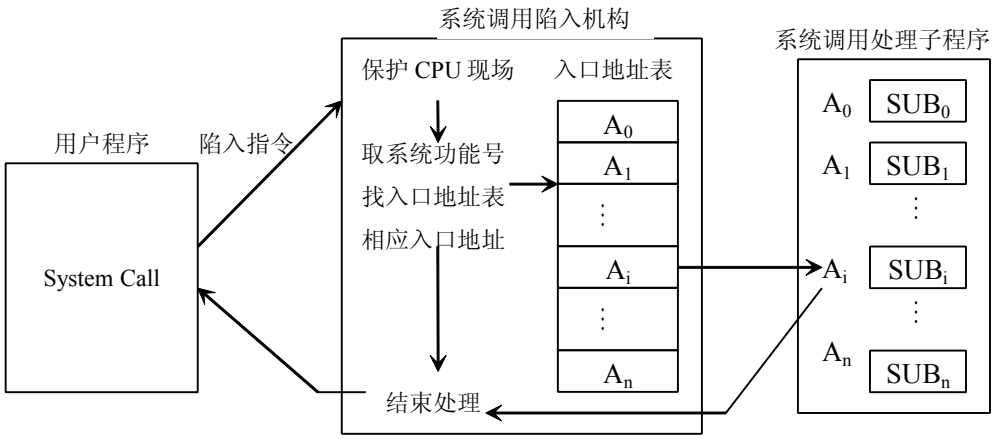


图 1-6　陷入机构和系统调用处理过程

参数传递是系统调用中应处理好的问题，不同的系统调用需传递给系统调用处理程序不同参数，反之，系统调用的结果也要以参数返回给用户程序。实现用户程序和系统调用之间的参数传递可采用以下方法：一是由访管指令或陷入指令自带参数，可以规定指令之后的若干单元存放的是参数，这叫直接参数；或者在指令之后紧靠的单元中存放参数的地址，这叫间接参数。二是通过 CPU 的通用寄存器传递参数，这种方法不宜传递大量参数。改进的方法是：在内存的一个块或表中存放参数，其首地址送入寄存器，实现参数传递，图 1-7 是这种方式参数传递的示意。三是在内存中开辟专用堆栈区域传递参数。

下面看一下系统调用的执行过程。在 Unix 中，’写操作’系统调用命令 C 语言写成如下形式：

```
nw = write(fd,buf,count)
```

其中，fd 为文件描述字、buf 为存放写数据的内存首址、count 为写字符个数，nw 为执行后实际返回的传送字节个数。经过 C 编译后，该语句由四条汇编指令组成的库函数替换成：

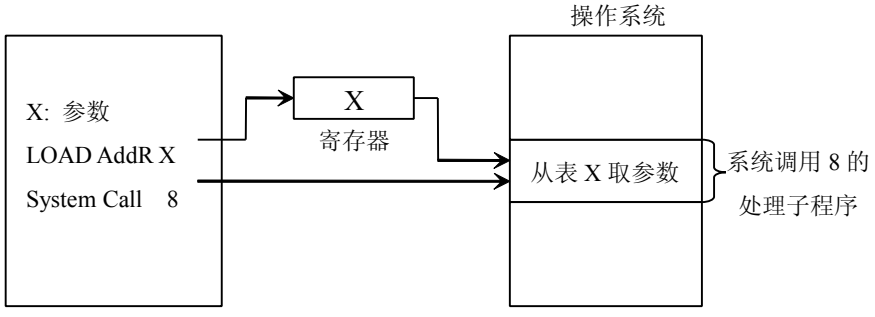


图 1-7　传递参数的一种方法

```
Load    r0,fd
Trap    4
Buf(参数 1)
Count(参数 2)
```

在运行目标程序时，图中 XXXXX 表示取 r0 中装着的 fd 值;*****表示写数据的内存起址;$$$$$表示应传送的字节个数;PDP 机器 Trap 指令的机器码为十六进制 89xx，8904 则为写操作系统调用代码。当执行到 trap 04 时，硬件自动产生中断，陷入操作系统的系统调用处理程序，这时做以下工作：①保护 CPU 现场，②查系统调用入口表 sysent 找到 write 处理程序入口，同时取到了本次调用所需参数个数，write 需三个，接收三个参数并作相应处理，③恢复保护现场，将实际传送字节个数送到 nw 返回断点 trap 的下一条指令执行。图 1-8 是 UNIX 系统调用执行过程示意。

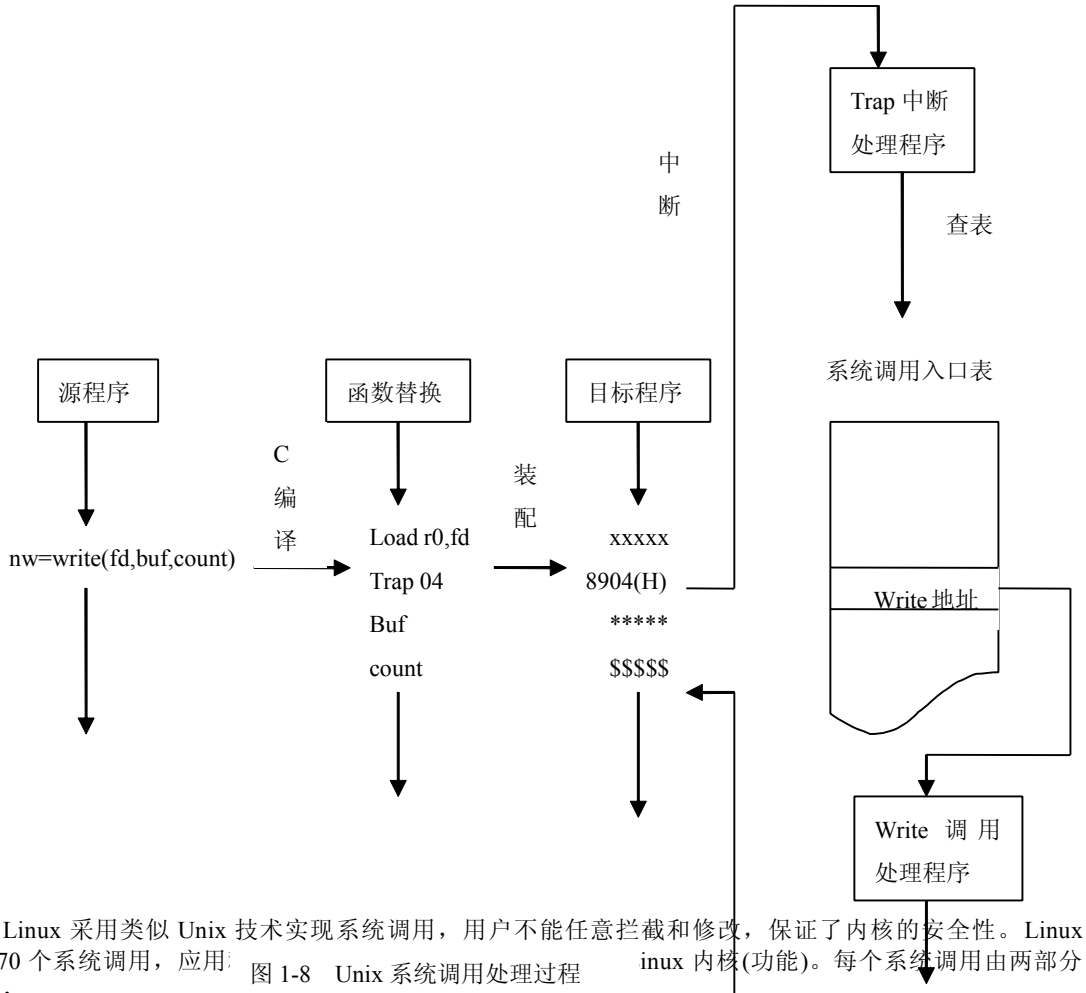


图 1-8　Unix 系统调用处理过程

Linux 采用类似 Unix 技术实现系统调用，用户不能任意拦截和修改，保证了内核的安全性。Linux 有 170 个系统调用，应用 inux 内核(功能)。每个系统调用由两部分组成：

- 核心函数　运行在核心态,是实现系统调用功能的(内核)代码,作为操作系 :内存中，是一种共享代码，用 C 语言书写。它运行在核心态，数据也存放在内核空间，通常它不能使用系统调用，也不能使用应用程序可用的库函数。
- 接口函数　是提供给应用程序的 API,以库函数形式存在 Linux 的 lib.a 中,该库中存放了所有系统调用的接口函数的目标代码，用汇编语言书写。其主要功能是把：系统调用号、入口参数地址传送给相应的核心函数，并使用户态下运行的应用程序陷入核心态。

Linux 中有一个系统调用入口程序 entry.s，是用汇编写的，它包含了系统调用入口地址表，给出了所有系统调用核心函数的名字：

```
ENTRY(sys-call-table)
    .long SYMBOL-NAME(sys-ni-syscall)    0
    .long SYMBOL-NAME (sys-exit)         1
    .long SYMBOL-NAME (sys-fork)             2
    .long SYMBOL-NAME (sys-read)             3
    .long SYMBOL-NAME (sys-write)        4
    .long SYMBOL-NAME (sys-open)         5
```

```
.long SYMBOL-NAME(sys-close)        6
 … … …
.long SYMBOL-NAME (sys-vfork )              190
```

Linux 的系统调用号就是系统调用入口表中位置的序号，所有系统调用通过接口函数将系统调用号传给内核，内核转入系统调用控制程序再通过调用号位置来定位核心函数,Linux 内核的陷入由 0x80(int80h)中断实现。系统调用控制程序的主要功能为：①取系统调用号;②根据系统调用号定位核心函数地址;③根据通用寄存器内容，从用户栈中取入口参数;④核心函数执行,把结果返回应用程序。

3）系统调用与过程(函数) 调用的区别

程序中执行系统调用或过程(函数) 调用，虽然都是对某种功能或服务的需求，但两者从调用形式到具体实现都有很大区别。

- 一是调用形式　过程(函数) 使用一般调用指令，其转向地址是固定不变的，包含在跳转语句中；但系统调用中不包含处理程序入口，而仅仅提供功能号，按功能号调用。
- 二是被调用代码的位置　过程(函数) 调用是一种静态调用，程序和被调代码在同一程序内，经过连接编辑后作为目标代码的一部份。当过程(函数) 升级或修改时，必须重新编译连结。而系统调用是一种动态调用，系统调用的处理代码在调用程序之外(在操作系统中) ，这样一来，系统调用处理代码升级或修改时，与调用程序无关。而且，调用程序的长度也大大缩短，减少占用的存储空间。
- 三是提供方式　过程(函数) 往往由高级语言或汇编语言的编译系统提供，不同编译系统提供的过程(函数) 可以不同；系统调用由操作系统提供，一旦操作系统设计好，系统调用的功能、种类与数量便固定不变了。
- 四是调用的实现　程序使用一般机器指令转子指令来调用过程(函数) ，是在用户态运行的；程序执行系统调用，是通过中断机构来实现，需要从用户态转变到核心态，在管理状态下运行，因此，安全性好。

1.5.2.2 系统程序

1）系统程序的分类

系统程序又称标准程序或实用程序(Utilities)，大多数用户只要求计算机解决自己的应用问题，对操作系统的特性、结构和实现不感兴趣。实用程序虽非操作系统的核心，但却是必不可少的，它们为用户程序的开发、调试、执行、和维护解决带有共性问题或执行公共操作,于是，操作系统常以外部命令形式向用户提供了许多系统程序。常用的有：汇编程序、编辑程序、编译系统、调试和排错程序等。用户看待操作系统，不是看系统调用怎么样，而是看系统程序怎么样，所以，系统程序功能和性能很大程度上反映了一个操作系统的功能和性能。操作系统的高层功能-系统程序，它的功能的实现从根本上来说要借助系统调用的实现。系统程序大致可分成以下几类：

- 文件管理。这些系统程序用来对文件和目录进行建立、删除、复制、改名、打印、列表、转储、和各种管理工作。
- 状态信息。这些系统程序供给用户向操作系统提问、以获得日期、时间、可用内存和盘空间数量、用户数、或其它状态信息，然后把这些信息格式化并打印到终端或其它输出设备或输到文件中。
- 程序设计语言支持。操作系统以系统程序方式提供给用户通用程序设计语言的编译程序、汇编程序和解释程序，如 Fortran、Cobol、Pascal、Basic、C、和 Lisp 等。这反映了操作系统对程序设计环境的支持能力，这些系统程序随硬件和操作系统一起出售，有一些则独立提供和幼另外收费。
- 程序的装入和执行支持。每当一个程序被编译或汇编后，必须被装入主存才能运行，于是，又提供了如绝对装入工具、重定位装入工具、复 装入工具、连接辑程序等系统程序。对高级语言或机器语言来说，调试工具(Debugging)也是必不可少的。
- 通信。操作系统提供一类系统程序，它们为建立多个进程、多个用户及不同计系统之间的逻辑连接提供了机制。这些机制允许用户发送消息到其他用户的屏幕上；或以电子邮件发送出一个大容量信息；或从一台机器传送文件到另一台机器；或甚至可以远程登录，使用远地的计算机。
- 其它软件工具。操作系统提供这类系统软件为用户解决共性问题，如 Web 览器、字处理工具、正文格式化工具、电子表格、数据库系统、编译程序的编译程序(YACC) 、画图软件包、统计分析包及游戏程序。

2）命令解释程序

操作系统提供的一个最最重要的系统程序是**命令解释程序**(Command Interpreter) ，用户通常通过操作命令来调用系统程序。命令解释程序的主要功能是接受和执行下一条用户从键盘输入的命令。当一个新的批作业被启动，或新的交互型用户登录进系统时，系统就自动地执行命令解释程序，它负责读入控制卡或命令行，并作出相应解释和执行。

命令解释程序的实现有两种常见方式。一种是它自身包含了命令的执行代码，于是收到命令后，便转向该命令处理代码区执行，在执行过程中常常会使用“系统调用” 帮助完成相应功能。在这种情况下，提供命令的数目就决定了命令解释程序功能的大小。另一种是全部命令都由专门的“系统程序” 实现，它自身不含命令处理代码，也不进行处理，而仅仅把这条命令对应的命令文件装入内存执行。例如，键入命令，

delete G

命令解释程序将寻找名字叫 delete 的命令文件，把它装入内存并将参数 G 传给它，由这个文件中的代码执行相应操作。因而，与 delete 命令相关的功能全部由 delete 命令文件代码决定，而与命令解释程序无关。这样一来可把命令解释程序做得很小，添加命令也很方便，只要创建一个实现新命令功能的命令文件就行了。这种方法也有缺点，由于命令处理程序是独立的系统程序，因而，参数传递会增加难度。所以，很多操作系统把两者结合起来，像列目录、查询状态之类的简单命令由命令解释程序处理，而像编译、编辑这样的复杂命令由独立的命令文件完成。

下面讨论命令解释程序的处理过程。操作系统做完准备工作后便启动命令解释程序，它输出命令提示符，等待键盘中断到来。每当用户打入一条命令(暂存在命令缓冲区)并按回车换行时，申请键盘中断。CPU 响应后，将控制权交给命令解释程序，接着读入命令缓冲区内容，分析命令、接受参数。若为简单命令立即转向命令处理代码执行。否则查找命令处理文件，装入主存，传递参数，将控制权交给其执行。命令处理结束后，再次输出命令提示符，等待下一条命令。

从用户使用系统的角度来说，他如何来向操作系统提交作业和说明运行意图呢？操作系统一般都提供了**联机作业控制**方式和**脱机作业控制**方式两个作业级的接口，这两个接口使用的手段为：操作命令(语言)和作业控制语言。

3）联机用户接口——操作命令

这是为联机用户提供的调用操作系统功能，请求操作系统为其服务的手段，它由一组命令及命令解释程序组成，所以也称为命令接口。当用户在键盘上每键入一条命令后，系统便立即转入命令解释程序，对该命令进行处理和执行。用户可先后进入不同命令，来实现对他的作业的控制，直至作业完成。

不同操作系统的命令接口有所有同，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。从用法和形式来看，可以把操作命令分成三种：

(1) 命令行方式

这是传统的命令形式，一个命令行由命令动词和一组参数构成，它指示操作系统完成规定的功能。对新手用户来说，命令行方式十分繁琐，难以记忆；但对有经验的用户而言，命令行方式用起来十分灵活，所以至今许多操作者仍支持这种命令形式。

Unix 操作系统面向用户的接口称作 Shell，它既是 Shell 命令组成的命令语言，又是 Shell 命令的解释程序。Shell 是作为用户进程运行的，每个用户登录后，Unix 便为他创建一个 Shell 进程。下面是一个简化了的 Shell 程序：

```
While (TRUE) {                                  /*TRUE=1,无限循环
        type-prompt( );                         /*输出屏幕提示符
        read-command(command,parameters);       /*从键盘读入参数
        pid=fork( );                            /*创建子进程
        if(pid<0 {
              printf("unable to fork!");        /*输出创建失败信息
              continue;                         /*继续循环
         }
        if(pid!=0) {                            /*创建成功
              waitpid(-1,&status,0);            /*父进程等子进程结束
         }
        else {
              execve(command,prraters,0);       /*子进程执行命令
         }
     }
```

Shell 程序反复从键盘读入命令及参数，每收到一条命令后,便创建一个子进程，父进程等待命令执行结束，子进程便执行命令。其中，所用系统调用 fork( )、waitpid( )、execve( )等，将在本书后面介绍。

简单命令是 Shell 命令语言的基础，其一般形式为：

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。Unix 的 Shell 命令语言有三种版本，贝尔实验室提供 Bourne-Shell；加里福尼亚大学贝克利分校提供 C-Shell；Microsoft 提供 B-Shell。每种提供了多达上百条简单命令，下面是 csh 中的列目录命令：

ls [任选项] (名字)

例如， % ls -l

```
drwxr-xr-x  2  feixl  user    87  May  23 08:08  Desktop
drwxr-xr-x  2  feixl  user    87  May  23 08:08  Dumpster
```

```
drwxr-xr-x   2   feixl   user      87   May   23 08:08   Public-html
-rw-r—r--    1   feixl   user      69   May   23 08;08   feixl.outbox
```

该命令列出目录的内容，任选项可用于控制每个文件的列表信息；名字可为目录名或文件名。Unix 常用的命令可以分成五大类：

- 文件管理类　cd、chmod、chown、chgrp、comm、cp、crypt、diff、file、find、ln、ls、mkdir、mv、od、pr、pwd、rm、rmdir。
- 进程管理类　at、kill、mail、nice、nohup、ps、time、write、mesg。
- 文本加工类　cat、crypt、grep、norff、uniq、wc、sort、spell、tail、troff。
- 软件开发类　cc、f77、login、logout、size、yacc、vi、emacs、dbs、lex、make、lint、ld。
- 系统维护类　date、man、passwd、stty、tty、who。

命令的细节请参见 Unix 使用手册。

MS-DOS 命令的一般形式为：

{<盘符>：}[<路径>]<命令>[<开关符>][<参数表>]

DOS 命令对应的处理程序如果常驻内存中，则该命令称“内部命令”；如果对应的处理程序是磁盘上的一个文件，该文件就叫命令文件或“外部命令”。内部命令有：cls(清屏幕)、date(显示与设置日期)、time(显示与设置时间)、ren(文件改名)、path(路径)、dir(列目录)、cd(改变目录)、md(建立目录)，rd(删除目录)、copy(文件复制)、type(文件显示)、del(文件删除)等；外部命令有：diskcopy(全盘复制)、diskcomp(全盘比较)、format(格式化)、chkdsk(检查磁盘)、fdisk(硬盘分区)、backup(备份磁盘)、restore(恢复盘文件)、xcopy(加强文件拷贝)、move(文件移动)、replace(文件替换)、deltree(删所有目录和文件)等。

下面是 MS-DOS 下执行列目录命令，以及系统的回答信息：

```
c>dir   c:\zjk
Volume   in   drive   C   has   no   label
Volume   searial   number   is   325B-1107
Directory   of   C:\zjk
.          <DIR>
. .        <DIR>
KHDG     wps        21,199     08-26-97    10:54a
BAK      <DIR>                 08-26-97    10:54a
KHDG     DOC        24,567     08-26-97     5:19p
KHDG     TXT         9,314      08-19-97    11:00a
Wang     DOC        18,944      08-18-97    12:05p
7 file(s)          74,967 bytes
```

(2) 批命令方式

在使用操作命令过程中，有时需要连续使用多条命令，有时需要多次重复使用若干条命令；还有时需要选择地使用不同命令，用户每次都将这一条条命令由键盘输入，既浪费时间，又容易出借。现代操作系统都支持一种特别的命令称为批命令，其实现思想如下：规定一种特别的文件称批命令文件，通常该文件有特殊的文件扩展名，例如，MS-DOS 约定为 BAT。用户可预先把一系列命令组织在该 BAT 文件中，一次建立，多次执行。从而减少输入次数，方便用户操作，节省时间、减少出借。更进一步，操作系统还支持命令文件使用一套控制子命令，从而，可以写出带形式参数的批命令文件。当带形式参数的批命令文件执行时，可用不同的实际参数去替换，从而，一个这样的批命令文件可以执行不同的命令序列，大大增强了命令接口的处理能力。

Unix 的 Shell 不但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，称作 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。myrun 是一个带位置参数的 Shell 文件，其功能是接受两个文件名作为实际参数，对第一个 C 语言编制的文件名的文件执行显示和编译，然后，把目标文件改成第二个文件名并执行之。

Myrun 包含：

```
cat      $1              /*  显示文件
cc       $1              /*  编译源程序
mv       a.out   $2      /*把目标程序改为$2
$2                       /*运行$2
```

执行 csh myrun prog1.c prog ，将对源程序 prog1.c 进行显示和编译，把目标程序改名为 prog 并运行。

再举一个 Unix 的例子，从/user/user1/fei 文件中，利用循环控制语句找出含单词：process、thread 信息行。其 Shell 程序 feipro 为：

```
for   i   do
    grep   $i/usr/usr1/fei
```

done

执行 csh feipro process thread 依次打印出含 process 和 thread 的信息行。

MS-DOS 操作系统中的批文件用“．bat” 作扩展名，与 Unix 类似，MS-DOS 的批命令文件可以使用的控制子命令有：rem(显示注释) 、pause(暂停) 、echo(显示开关) 、for(循环语句) 、if(条件语句) 、goto(转向语句) 、shift(参数左移) 、call(调用批文件) 等。Test．bat 是用来查找某文件是否存在的批文件，其内容为：

```
echo  off
if  exist  %1  goto  lab1
echo  %1  not  found!
goto  lab2
：lab1
echo  %1  is  found!
：lab2
```

用户键入命令：test wps．exe 执行批文件，若找到文件显示 wps．exe is found! ；否则显示 wps.exe not found!

(3) 图形化方式

用户虽然可以通过命令行方式和批命令方式来获得操作系统的服务，并控制自己的作业运行，但却要牢记各种命令的动词和参数，必须严格按规定的格式输入命令，这样既不方便又花费时间，于是，图形化用户接口 GUI（Graphics User Interface）便应运而生，是近年来最为流行的联机用户接口。

GUI 采用了图形化的操作界面，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。用户可以通过鼠标、菜单、对活框和滚动条完成对他们作业和文件的各种控制和操作。此时，用户不必死记硬背操作命令，而能轻松自如地完成各项工作，使计算机系统成为一种非常有效且生动有趣的工具。

九十年代推出的主流操作系统都提供了 GUI，GUI 的鼻祖首推 Apple 公司的 Macintosh 操作系统。此外，如 Microsoft 公司的 Windows, IBM 公司的 OS/2。为了促进 GUI 的发展，已制订了国际 GUI 标准，该标准规定了 GUI 由以下部件构成：窗口、菜单、列表框、消息框、对话框、按钮、滚动条等。许多系统软件如 Windows NT、Visual C++、Visual Basic 等，均可应用户程序要求自动生成应用程序的 GUI，大大缩短了应用程序的开发周期。图 1-10 是用 Visual Basic 和 Visual Foxpro 开发应用程序的一个界面。

图 1-9 应用程序 GUI 的一个例子

随着个人计算机的广泛流行，缺乏计算机专业知识的用户随之增多，如何不断更新技术，为用户提

供形象直观、功能强大、使用简便、掌握容易的用户接口，便成为操作系统领域的一个热门研究课题，例如具有沉浸式和临场感的虚拟现实应用环境已走向实用。目前多通道用户接口，自然化、人性化用户接口，甚至智能化用户接口的研究都取得了一定的进展。

4）脱机用户接口——作业控制语言

这种接口是专为批处理作业的用户提供的，所以也称批处理用户接口。操作系统提供了一个作业控制语言（Job Control Language）,它由一组作业控制语句或作业控制操作命令组成，Unix 中的 Shell 是一种 JCL 语言。由于批处理作业的用户不能直接与他们作业交互，只能委托操作系统来对作业进行控制和干予，作业控制语言便是提供给用户，为实现所需作业控制功能委托系统代为控制的一种语言。用户使用 JCL 的语句，把他的运行意图、即需要对作业进行的控制和干予，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交给系统。当调度到该批处理作业运行时，系统调用 JCL 语句处理程序或命令解释程序，对作业说明书上的语句或命令，逐条地解释执行。如果作业在执行过程中出现异常情况，系统会根据用户在作业说明书上的指示进行干予。这样，作业一直在作业说明书的控制下运行，直到作业运行结束。可见 JCL 为用户提供了一种作业一级的接口。

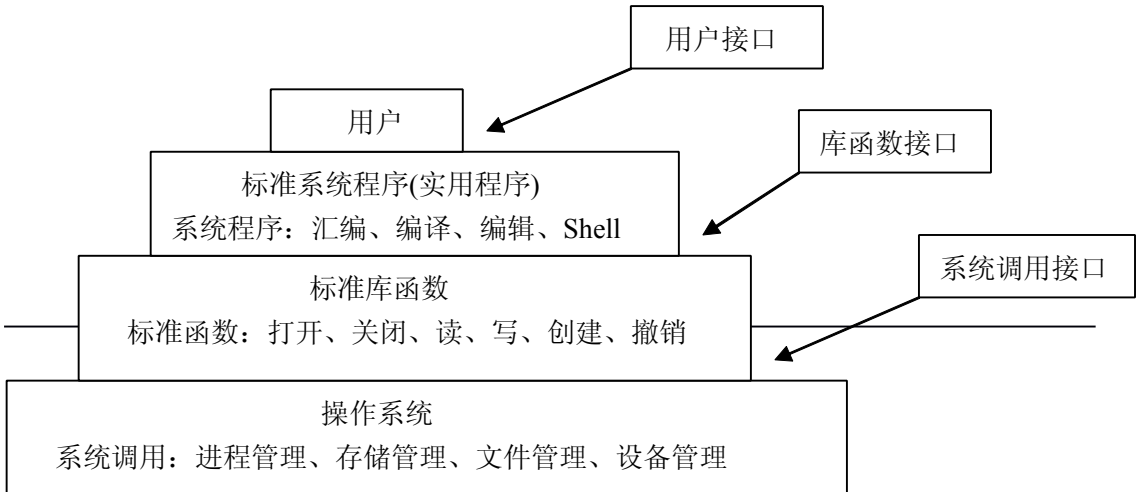
下面通过 IBM 的 JCL 的一个例子来说明作业控制接口。有一个需要编译、连结编辑的作业，它的源程序和数据穿在卡片上，编译、连结编辑的结果均需在行式打印机上输出，编译结果要存盘，连结编辑结果还要从穿孔机输出，此作业用 JCL 语句组织如下:

```
// HAROLD JOB,WILSON,MSGLEVEL=(2,0),PRTY=6,CLASS=b
// COMP EXEC PGM=`                    IEYFORT
// SYSPRINT DD SYSOUT=A
// SYSLIN DD DSNAME=SYSL,DISP=OLD
// SYSIN DD*
                    .
                    .
              <Source Program Card>
                    .
                    .
/*
// GO EXEC PGM=FORTLINK
// SYSPRINT DD SYSOUT=A
// FTOTF001 DD UNIT=SYSCP
// GO SYSIN DD*
                    .
                    .
               <Data Card>
                    .
                    .
/*
//
```

必须为采用批处理处理的作业书写作业说明书，用 JCL 语句书写的作业控制流仅仅是其中的一部分，此外还应包括作业基本情况、作业资源需求等其它各种控制信息。

1.5.2.3 Unix的系统调用、库函数和系统程序

图 1-10 列出了 Unix 的系统程序、库函数、系统调用的层次关系。兼容的 Unix 库函数和系统调用符合 POSIX1003.1 标准，用户可以使用库函数或系统调用直接调用 Unix 底层功能。Unix 的标准系统调用分进程管理、文件管理、设备管理、及状态控制等五十余条，而根据 POSIX1003.2 标准，描述了近百个系统程序的语法和语义，其中大部分集中在文件管理、目录管理、过滤器、编译和程序开发工具方面。现有的各种 Unix 版本均实现了其中一部分最常用的系统程序约五十多条。系统调用是软件开发者进入 Unix 内核的入口，内核常驻内存，对所有用户进程的工作、文件系统的访问进行监视和控制。



Unix 还提供了许多标准函数(又叫标准例行程序)，它们组成了标准函数库，其中较重要的是 I/O 库函数。有些功能系统调用不能实现，而函数却能做到，如格式变换、自动缓冲等。一般来说标准函数库建立在系统调用的上层，因此它提供的功能较系统调用强，使用也较方便。但是，库函数的实现最终要用到系统调用，他们是基于系统调用的较高层软件。

Unix 的内核之外包含有非常丰富的系统程序，包括语言处理程序、文件编辑程序和种种软件开发及维护工具。据 AT&T 公司称，Unix 下运行的微机软件包已有 1000 多个，正是这些应用程序大大加强了 Unix 的功能。通常，在高级语言中没有系统调用的概念，它通过标准函数或语句来间接地调用操作系统的功能。不论高级语言还是汇编语言，最终实际执行的不是源程序，而是经过编译或汇编产生的目标代码。需要调用操作系统功能的语句或标准函数，经编译及连接装配成目标程序段，其中包含了一系列对应的系统调用。

采用这样的层次结构后，外层系统程序很容易被替换，例如更换 Shell、和文件编辑器，某些 Unix 版本已经使用 GUI 替代命令用户界面，但其自身并未有多大改变，这种灵活性使得 Unix 虽经几次重大技术变革，但仍能经久不衰。

1.6 流行操作系统简介

1.6.1 DOS 操作系统

DOS 是在微型计算机上使用最为广泛的操作系统，是一种单用户、普及型的磁盘操作系统。主要用于以 Intel 公司的微处理器为 CPU 的微机及其兼容机，曾经风靡了整个八十年代。

DOS 1.0 版于 1981 年随 IBM PC 微型机一起推出，此后的十多年里，随着微机的发展，DOS 也不断更新改进，直到 1994 年推出了最后的版本 DOS 6.22。

DOS 的主要功能有：命令处理、文件管理和设备管理。DOS 4.0 版以后，引入了多任务概念，强化了对 CPU 的调度和加强了对内存的管理，但 DOS 的管理功能比其他操作系统却简单得多。

DOS 采用汇编语言书写，因而，系统开销小，运行效率高；DOS 针对微机环境来设计，因而实用性好，较好地满足了低档微机工作的需要。但是，随着微机性能的突飞猛进，DOS 的缺点不断显露出来，已经无法发挥硬件的能力，由于缺乏对数据库、网络通信等的支持，没有通用的应用程序接口，加上用户界面不友善，操作使用不方便，现在已经逐步让位于 Windows 操作系统。

1.6.2 Windows 操作系统

1.6.2.1 Windows操作系统概况

Microsoft 公司于 1983 年 11 月开始研制图形化用户界面操作环境 Windows，1985 年 11 月推出早期版 Windows 1.01 投放市场，1987 年推出 Windows2.0，1990 年宣布 Windows 3.0，对原来系统做了彻底改造，这是一个重要的里程碑；1992 年 4 月 Windows 3.1 发布后在全世界流行，此后又推出了中文版 Windows 3.2。上述 Windows 系统都依靠 DOS 提供的基本硬件管理功能才能工作。1993 年 5 月推出的 Windows NT，1995 年 8 月推出的 Windows 95，以及 1997 年、1998 年推出的 Windows 97 和 Windows98 能独立在硬件上运行，是真正的新型操作系统。上述版本分两个系列，个人操作系统 Windows3.x 和 Windows9x 主要运行于个人计算机系列;多用户操作系统 Windows NT 系列除了可运行于个人计算机外，也能在服务器和工作站上运行。1996 年 11 月微软又推出嵌入式 Windows CE 操作系统。目前个人计算机上采用 Windows 操作系统的占 90%，微软公司垄断了 PC 行业。

Windows 的主要特点是：

- 全新的图形界面，操作直观方便。
- 新的内存管理，突破 640KB 限制，实现了虚存管理。

- 提供多用户多任务能力，多任务之间既可切换，又能交换信息。
- 提供各种系统管理工具，如程序管理器、文件管理器、打印管理器及各种实用程序。
- Windows 环境下允许装入和运行 DOS 下开发的程序。
- 提供数据库接口、网络通信接口。

1.6.2.2 Windows 95/98

Windows 95 与以前 Windows 版本相比，不仅具有更直观的工作方式和优良的性能，而且还具有支持新一代软硬件需要的强大能力。Windows 95 的主要特点是：

- Windows 95 是一个 32 位操作系统，同时也能运行 16 位的程序。在 32 位的方式下具有抢先多任务能力，真正达到了 Windows 多用户多任务目标。
- Windows 95 提供“即插即用”功能，系统增加新设备时，只需把硬件插入系统，由 Windows 95 解决设备驱动程序的选择和中断设置等问题，对用户来说插好即能用。
- Windows 95 具有内置网络功能，直接支持联网和网络通信。同时，也提供环球邮箱和对 Internet 的访问工具。
- Windows 95 具有多媒体功能，如 Autoplay 特性让用户方便地安装和运行 CD-ROM；内置 CD-Player，用户可在 CD-ROM 上播放 CD 唱盘；支持数字视频文件格式转换等。
- Windows 95 提供了一组适应新界面的工具，如 32 位写字板，32 位画图工具，32 位备份工具等。

1.6.2.3 Windows NT

在 Windows 发展过程中，硬件技术和系统性能在不断进步，如基于 RISC 芯片和多 CPU 结构的微机出现；客户机/服务器模式的广泛采用；微机存储容量增大及配置多样化；同时，对微机系统的安全性也提出了更高的要求。1993 年推出的 Windows NT（New Technology）便是为这些目标而设计的，被认为是目前最有前途的工作站之一。除了 Windows 产品的上述功能外，它还有以下特点：

- 支持对称多处理(Symmetric Multiprocessing,SMP)和多线程，即多个任务可基于多个线程对称地分布到各个 CPU 上工作，具有良好的可伸缩性，从而大大提高了系统性能。
- 支持抢先的可重入多任务处理。
- 32 位页式授权虚拟存储管理。
- 支持多种 API 和源码级兼容性。
- 支持多种可装卸文件系统，包括 DOS 的 FAT 文件系统、OS2 的 HPFS、CD_ROM 文件系统 CDFS 和 NT 文件系统 NTFS。
- 具有各种容错功能，C2 安全级。
- 可移植性好，既可在 Intel x86、PowerPC、Digital Alpha AXP 也可在 MIPS RISC 平台上运行;既可作为网络客户，又可作为网络服务。
- 集成网络计算，支持 LAN Manager, 为其他网络产品提供接口。
- 采用 16 位标准字符集的单一代码方法支持世界范围字符集。
- 符合 POSIX，提供了 POSIX 应用执行环境选件。
- 能与其 Microsoft SQL Server 结合，提供 C/S 数据库应用系统的最好组合。

1.6.2.4 Windows 2000

Windows2000 是在 Window NT4.0 基础上修改和扩充而成的(即是 NT5.0)。它不是单个操作系统，而包括了四个系统用来支持不同对象的应用。“专业版”为普通用户设计，可支持 2 个 CPU，最大内存可配 4GB；“服务器版”为中小企业设计，可支持 4 个 CPU，最大内存 4GB；“高级服务器版”为大型企业设计，支持 8 个 CPU 和最大 8GB 内存；“数据中心服务器版”专为大型数据中心开发，支持最多 32 个 CPU 和最大 64GB 内存。

1.6.3 Netware 操作系统

Novell 公司是主要的 PC 网络产品制造商之一，成立于 1983 年，它的网络产品可以和 IBM、Apple、Unix 及 Dec 系统并存和互操作，组成开放的分布式集成计算环境，Netware 是其开发的网络操作系统(Networking Operating System,NOS)。

Netware 具有高性能文件系统、支持 DOS、OS/2、MAC、及 Unix 文件格式;具有三级容错，可靠性高;具有良好的权限管理，安全保密性好;具有开放性，提供开放的开发环境。

Netware 有多种版本，Netware lite 是廉价点对点 NOS，最多支持 25 个用户，每个结点可作为对等机，安装和管理都很方便。Netware2.2 是为小单位和工作组开发的 NOS，是一个多功能产品，满足工作组用户的各种需要。Netware3.1x 是一个 32 位的 NOS，更能发挥高档 PC 的计算能力，支持用户可达 256 个。同时，能支持 DOS，Windows、Macintosh、OS/2 和 Unix 工作站访问 Netware 服务器。Netware4.xx 是其 NOS 新版，增强了网络目录服务，支持用户数可达 1000 个。Netware3.11 是最为成功的 NOS，已占有很大市场份额，Novell 对其进行了改进于 1993 年 8 月推出替代产品 Netware3.12，其中，主要加进了以下功能：电子邮件、支持 Windows 用户的工具、支持 CD-ROM、加强了工作站 Shell 功能。

1.6.4 Unix 操作系统

Unix 操作系统是一个通用、交互型分时操作系统。它最早由美国电报电话公司的贝尔实验室于 1969 年在 DEC 公司的小型系列机 PDP-7 上开发成功，1973 年开发出 C 语言并用它改写 Unix(形成了 Unix 第5 版)，从而使得 Unix 具有高度易读性、可移植性，为迅速推广和普及走出了决定性的一步。1974 年 7 月，"Unix 分时系统"一文在美国权威杂志 CACM 上发表，这是第一次正式公布于众，引起了广泛注意。最早外界可获得的 Unix 是 1975 年的 Unix 第 6 版;1978 年的 Unix 第 7 版，可以看作当今 Unix 的祖先，该版为 Unix 走进商界奠定了基础。1981 年，AT&T 公司发表了 Unix System Ⅲ，从此开始，Unix 不再按 Version 而改为按 System 号排列。1983 年，AT&T 公司公布了 Unix System Ⅴ，至今它还是最新版本。由大学开发的非 AT&T 系统的 Unix 是美国加利福尼州立大学 Berkeley 分校运行在 VAX-11 机上的 Unix BSD(Berkelry Software Distribution,BSD)，其中，最著名的是 4.1BSD、4.2BSD、4.3BSD 和 4.4BSD，这些版本中加入了页式虚存、长文件名、加进网络协议 TCP/IP 等，在 Unix 的发展中起了重要作用，该校的版本已成为教学、科研、商用的主流系统。Unix 取得成功的最重要原因是系统的开放性，公开源代码，用户可以方便地向 Unix 系统中逐步添加新功能和工具，这样可使 UINX 越来越完善，能提供更多服务，成为有效的程序开发支撑平台。它是目前唯一可以安装和运行在从微型机、工作站直到大型机和巨型机上的操作系统。

Unix 的主要特点：

- Unix 的结构分核心部分和应用子系统，便于做成开放系统。
- 具有分层可装卸卷的文件系统，提供文件保护功能。
- 提供 I/O 缓冲技术，系统效率高。
- 剥夺式动态优先的 CPU 调度，有力地支持分时操作。
- 命令语言丰富齐全。
- 具有强大的网络与通信功能。
- 支持多用户、多任务
- 请求分页式虚拟存储管理

实际上，Unix 已成为操作系统标准，而不是指一个具体操作系统。许多公司和大学都推出了自己的 Unix 系统，如 IBM 公司的 AIX 操作系统，SUN 公司的 Solaris 操作系统，柏克利大学的 Unix BSD 操作系统，DEC 公司的 ULTRIX 操作系统，CMU 的 Mach 操作系统，SW 公司的 Unix SCO 以及 AT&T 公司自己的 Unix SystemV 等。上述版本互不兼容，非常混乱。为了使同一个程序能在所有不同 Unix 版本上运行，IEEE 拟定了一个 Unix 标准，称作 POSIX。它定义了相互兼容的 Unix 系统必须支持的最少系统调用接口。该标准已被多数 Unix 支持，冈时，其他一些操作系统也在支持 POSIX 标准。

1.6.5 Solaris 操作系统

SUN Microsystem 公司开发的 Solaris 是具有完全对称多处理和多线程支持的 32 位分布式计算环境的 Unix 操作系统变种。目前 Solaris2.x 基于 SPARC 和 Intel 平台，但是一个可移植操作系统，可以移植到任何新的主流平台上。1998 年 10 月，SUN 公司隆重推出 64 位操作系统 Solaris7.0，这是基于 2.6 版的最新版本，在网络特性、可靠性、兼容性、互操作性、易于配置和管理方面均有很好改进。Solaris 的设计追求三个目标：一是高性能，通过多用户、多任务、SMP、和多线程能力，最大限度发挥硬件的潜力;二是满足用户变化的需要，建立分布式 C/S 解决方案，允许企业用户利用企业计算环境的能力和资源，更加高效率地解决业务问题;三是支持异构计算环境，用户可以从现有的信息技术投资中得到最大回报。

下面着重介绍 SUN 的另一个流行产品。SUN NFS(Network File System)是 1984 年推出的一个网络文件系统产品，如今已成为一种事实上的标准。NFS 提供了在异种机、异种操作系统的网络环境下共享文件的简单有效的方法。NFS 基于 C/S 模式，使用 UDP/FP 协议和 RPC(Remote Procedure Call)机制，主要特点有：①提供透明的文件访问和文件传送，用户使用本地或远程的文件没有区别，真正实现了分布式数据处理;②易于扩充，扩充新的资源或软件，不需改变现有工作环境;③性能高、可靠性好、具有可伸缩性。

1.6.6 Macintosh 操作系统

1986 年，美国 Apple 公司推出 Macintosh 机操作系统。MAC 是全图形化界面和操作方式的鼻祖。由于它拥有全新的窗口系统、强有力的多媒体开发工具和操作简便的网络结构而风光一时。Apple 公司也就成为当时唯一能与 IBM 公司抗衡的 PC 机生产公司。MAC 是基于 Macintosh 的 M680X 系列芯片的微型机，MAC 操作系统的主要特点有：1）采用面向对象技术；2）全图形化界面；3）虚拟存储管理技术；4）应用程序间的相互通信；5）强有力的多媒体功能；6）简便的分布式网络支持；7）丰富的应用软件。

MAC 的主要应用领域为：桌面彩色印刷系统、科学和工程可视化计算、广告和市场经营、教育、财会和营销等。

1.6.7 MINIX 操作系统

Unix 早期版本的源代码可以免费获得并被人们加以广泛的研究，世界上许多大学的操作系统课程 都讲解 Unix 部分源码。在 AT & T 公司发布 Unix 7.0 时，它开始认识到 Unix 的商业价值，并禁止在课程中研究其源代码，以免商业利益受到损害。许多学校为了遵守该规定，就在课程中略去 Unix 的源码分析而只讲操作系统原理。不幸的是，只讲理论使学生不能掌握操作系统的许多关键技术。为了扭转这种局面，荷兰 Vrije 大学计算机系教授 Andrew S. Tanenbavm 决定开发一个与 Unix 兼容，然而内核全新的操作系统 Minix。Minix 没有借用 AT & T 一行代码，所以不受其许可证的限制，学生可以通过它来剖析一个操作系统，研究其内部如何运作，其名称源于‘小 Unix’，因为它非常简洁，短小，故称 Minix。

Minix 用 C 语言编写，着眼于可读性好，代码中加入了数千行注释。可运行在 IBM PC，Macintosh，Sparc，Amiga，Atari 等许多平台上。Minix 一直恪守“Small is Beautiful”的原则，早期 Minix 甚至没有硬盘就能运行。目前最常用的是 Minix2.0，它具有多任务处理能力，可支持三个用户同时工作，支持 TCP/IP，支持 4GB 内存。提供 5 个编辑器、200 个实用程序。

在 Minix 发布后不久，Internet 上出现了一个面向它的 USENET 新闻组，数以万计的用户订阅读新闻。其中的许多人都想向 Minix 中加入新功能或新特性，使之更大更实用，成百上千的人提供建议，思想甚至代码。而 Minix 作者数年来一直坚持不采纳这些建议，目的是使 Minix 保持足够的短小精悍，便于学生理解。人们终于意识到作者的立场不可动摇，于是芬兰的一个学生 Linvs Torvalds 决定编写一个类似 Minix 的系统，但是它功能繁多，面向实用而非教学，这就是 Linux 操作系统。

附：Tanenbavm 主页地址： http: //www.cs.vu.nl/~ast/ Minix 的版权为 Prentice Hall 所有，可免费下载用于教学。

1.6.8 自由软件和 Linux 操作系统

自由软件（Free Software or Freeware）是指遵循通用公共许可证 GPL（General public License）规则，保证您有使用上的自由、获得源程序的自由，可以自己修改的自由，可以复制和推广的自由，也可以有收费的自由的一种软件。

自由软件的出现意义深远。众所周知，科技是人类社会发展的阶梯，而科技知识的探索和积累是组成这个阶梯的一个个台阶。人类社会的发展是以知识积累为依托的，不断地在前人获得知识的基础上发展，更新才得以提高。软件产业也是如此，如果能把已有的成果加以利用，避免每次都重复开发，将大大提高目前软件的生产率，借鉴别人的开发经验，互相利用，共同提高。带有源程序和设计思想的自由软件对学习和进一步开发，起到极大促进作用。自由软件的定义就确定了它是为了人类科技的共同发展和交流而出现的。free 指是的自由，但并不是免费。‘自由软件之父’Richard Stallman 先生将自由软件划分为若干等级：其中,自由之 0 级是指对软件的自由使用；自由之 1 级是指对软件的自由修改；自由之 2 级指对软件的自由获利。

自由软件赋予人们极大的自由空间，但这并不意味自由软件是完全无规则的，例如 GPL 就是自由软件必须遵循的规则，由于自由软件是‘贡献型’，而不是‘索取型’的，只有人人贡献出自己的一份力量，自由软件才能得以健康发展。Bill Gates 早在八十年代，曾大声斥责“软件窃取行为” ，警告世人这样会破坏整个社会享有好的软件的时候，那么自由软件的出现是软件产业的一个分水岭。在拥戴自由软件的人们眼中，微软的做法的实质是阻止帮助邻人，抑制了软件对社会的作用,剥夺了人们’共享’与’修改’软件的自由。

GNU 的含义是 GNU is not Unix 的意思，由自由软件的积极倡导者 Richard stallman 先生指导并启动的一个组织。七十年代后期起很多软件不再提供源码，使用户无法修改软件中的错误，使用尤为不便。为此，在 1984 年，Stallman 先生启动了 GNU 计划，并成立了自由软件基金会（Free Software Foundation, FSF）。Stallman 先生通过 GNU 写出一套和 Unix 兼容，但同时又是自由软件的 Unix 系统，GNU 完成了大部分外围工作，包括外国命令 gcc/ gcc^{++}, shell 等，最终 Linux 内核为 GNU 工程划上了一个完美句号，现在所有工作继续在向前发展。目前人们熟悉的一些软件如 C++编译器、Objective C、FORTRAN77、C 库、BSD email、BIND、Perl、Apache、TCP/IP、IP accounting、HTTPserver、Lynx Web 都是自由软件的经典之作。

GPL 协议可以看成为一个伟大的协议，是征求和发扬人类智慧和科技成果的宣 言书，是所有自由软件的支撑点，没有 GPL 就没有今天的自由软件。

Linux 是由芬兰藉科学家 Linus Torvalds 于 1991 年编写完成的一个操作系统内核，当时他还是芬兰首都赫尔辛基大学计算机系的学生，在学习操作系统课程中，自己动手编写了一个操作系统原型，从此，一个新的操作系统诞生了。Linus 把这个系统放在 Internet 上，允许自由下载，许多人对这个系统进行改进、扩充、完善，许多人做出了关键性贡献。Linux 由最初一个人写的原型变化成在 Internet 上由无数志同道合的程序高手们参与的一场运动。

Linux 属于自由软件，而操作系统内核是所有其他软件最最基础的支撑环境，再加上 Linux 的出现时间正好是 GNU 工程已完成了大部分操作系统外围软件，水到渠成，可以说 Linux 为 GNU 工程画上了一个园满句号。除了 Linux 外，还有许多自由软件，如 FreeBSD、NetBSD、OpenBSD 等都是较优秀的具有

自由版权的 Unix 类操作系统。Apache 也是一个著名的自由软件，已在服务器上广泛使用，支持包括 Linux、Free BSD、Solaris 及 HP-Vx 等很多操作系统平台。GNU 工程下还有很多自由软件，如 gcc/gcc++、perl、bash/tcsh 和 bin86 等等。

短短几年，Linux 操作系统已得到广泛使用。1998 年，Linux 已在构建 Internet 服务器上超越 Windows NT。计算机的许多大公司如 IBM、Intel、Oracle、Sun、Compaq 等都大力支持 Linux 操作系统，各种成名软件纷纷移植到 Linux 平台上，运行在 Linux 下的应用软件越来越多，Linux 的中文版已开发出来，Linux 已经开始在中国流行，同时，也为发展我国自主操作系统提供了良好条件。

Linux 是一个开放源代码，Unix 类的操作系统。它继承了历史悠久和技术成熟的 Unix 操作系统的特点和优点外，还作了许多改进，成为一个真正的多用户、多任务通用操作系统。1993 年，第一个产品版 Linux1.0 问世的时候，全部按自由扩散版权进行扩散，即公开源码，不准获利。不久发现这种纯粹理想化的自由软件会阻碍 Linux 的扩散和发展，特别限制了商业公司参与并提供技术支持的积极性。于是 Linux 转向 GPL 版权，除允许享有自由软件的各项许可权外，还允许用户出售自由软件拷贝程序。这一版权上的转变后来证明对 Linux 的进一步发展十分重要。

从 Linux 的发展可以看出，是 Internet 孕育了 Linux，没有 Internet 就不可能有 Linux 今天的成功。从某种意义上来说，Linux 是 Unix 和 Internet 国际互联网结合的一个产物。自由软件 Linux 是一个充满生机，已有巨大用户群和广泛应用领域的操作系统，看来它是唯一能与 Unix 和 Windows 较量和抗衡的一个操作系统了。

Linux 作为一个充满生机，有着巨大用户群繁和广泛应用领域的操作系统，已在软件业中有着重要地位，从技术上讲它有如下特点：(1) 继承了 Unix 的优点，又有了许多更好的改进，开放、协作的开发模式，是集体智慧的结晶，能紧跟技术发展潮流，具有极强的生命力；(2) 通用的操作系统，可作为 Internet 上的服务器；可用做网关路由器；可用做文件和打印服务器；也可供个人使用入；(3) 内置通信联网功能，可让异种机联网；(4) 开放的源代码，有利于发展各种特色的操作系统；(5) 符合 POSIX 标准，各种 Unix 应用可方便地移植到 Linux 下；(6) 提供庞大的管理功能和远程管理功能；(7) 支持大量外部设备；(8) 支持 32 种文件系统,如 Ext2、Ext、Xiafs、Isofs、Hpfs、MSDOS、UMSDOS、Proc、NFS、SYSV、Minix、SMB、Ufs、Ncp、VFAT、AFFS 等；(9) 提供 GUI，有图形接口 X-Windows，有多种窗口管理器；(10) 支持并行处理和实时处理，能充分发挥硬件性能；(11) 可自由获得源代码，在 Linux 平台上开发软件成本低。

1.6.9 IBM 系列操作系统

1.6.9.1 IBM系列操作系统概述

美国 IBM(Internetional Business Machines Corp.)-国际商业机器公司成立于 1914 年，是当今世界上最大的 IT 跨国公司，在全球有数十万雇员，业务遍及 150 多个国家。计算机发展史上的许多重大革新是由 IBM 开创：第一个生产系列计机、第一个研制出集成电路计算机、第一个制造出计算机用磁盘、开发出迄今最小的硬盘、开发出迄今最快的商用机。人类许多重大事件都有 IBM 技术的参与：1969 年阿波罗宇航器登月、1981 年哥伦比亚航天飞机进入太空、1996 年 IBM 超级计算机 Deep Blue 战胜世界象棋冠军 Garry Kasparov、迄今最快的巨型机是 IBM 的 Blue Pacific。

在过去的几十年里，IBM 研制开发和生产销售的 IT 产品包罗万象、品种繁多。从主机到外设;由系统软件到应用软件。从原始到先进;由简单到成熟，功能逐步完善，性能不断改进。

回顾 IBM 操作系统的演变，大致可以看出现代操作系统的发展历程。

- 第一阶段六十年代以前，计算机型号为：IBM1404、1620、704、709、7094 等;使用操作系统为 FMS(Fortran Mornitor System)、IBMSYS(IBM7094 Mornitor)等;主要使用的技术有：自动批处理、脱机外围设备操作、特权指令、管态/目态、人工设备分配和负载平衡、原始文件系统、程序复盖和重迭支持。
- 第二阶段七十年代以前，计算机型号为：IBM S/360 系列机等; 使用操作系统为 OS/360(又分成任务数固定多道程序系统 OS360/MFT(Multiprogramming With a Fixed number of Tasks)和任务数可变多道程序系统 OS360/MVT(Multiprogramming With a Variable number of Tasks)和磁盘操作系统 DOS/360。主要使用的技术有：多道程序设计、远程作业进入、SPOOLING、内存复盖重迭的完整支持、完善的文件系统、库子程序、多种高级语言。
- IBM 公司推出 S/360 系列计算机试图达到两个目标：一是兼容性，由于所有计算机都有相同的体系结构和指令系统，因而，为一种型号机器编写的程序可以在其他所有机器上运行;二是通用性，该系列机既可用于科学计算，又可用于商业数据处理，以便一个系列机能满足所有用户的要求。IBM 基本上达到了这两个目标，取得了成功。
- 第三阶段七十年代开始，计算机型号为：IBM S/370 系列机、IBM43xx 系列机、IBM308x 系列机、IBM309x 系列机等; 使用操作系统为 OS/VS1(OS360/MFT 的后裔)和 DOS/VS。主要使用的技术有：虚拟存储器、多任务处理、Checkpoint、交互式处理。

九十年代之前，IBM 系列操作系统又有进一步发展。①在 OS360/MVT 操作系统的基础上扩充，开发出能为每个用户提供私有虚存空间的多虚存操作系统 OS/VS2(又叫 MVS-Multi Virtual Storage);②随着

用户希望系统提供交互型程序开发环境愿望越来越迫切，六十年代中，IBM 公布了其分时系统 TSS/360，结果由于太大太慢，导致无人使用，成为一个失败的系统。③七十年代起，IBM 开始研究 Virtual Machine 操作系统，从 CP/40 和 CP/67 开始，而后发展成为 VM/370 和 VM/SP，目前，该系统仍在 S/390 上运行。

目前，IBM 的主要计算机产品是服务器和 PC，下面对这些产品及其上运行的操作系统作简单介绍。

RS/6000 系列 Unix 服务器及 SP 结点群集计算机　采用 POWERpc(Performance Optimized With Enhanced RISC,POWER)604 CPU，运行 AIX 操作系统。

RS/6000 是 IBM 于七十年代末推出的第二代 RISC 服务器，1994 年 10 月开发出 SMP 的 RS/6000 产品，之后又设计出并行机 RS/6000SP。RS/6000 上运行 AIX(Advanced Interactive executive,AIX)操作系统，AIX 操作系统是 1990 年推出的超强设计重负载 Unix 操作系统，目前最新版本是 4.3。它是一个具有可伸缩性、高安全性、高可靠性的软实时操作系统，配合相关硬软件后，可以全年不停机工作。它提供了一个安全的图形化界面的多用户环境，支持多线程、支持动态装卸设备驱动程序、虚存空间可达 2E+52 字节，网络特性出色、管理工具多样，各种语言支持、商用 Unix 软件大都可在其上运行，许多自由软件可以移植过来。世界上许多 Internet 网站和研究中心都采用 RS/6000 及 AIX 系统，主要用在 FTP、email、Web 服务器、数据库服务器，及 CAD、CAE、可视化等各种科学和工程应用。

1.6.9.2 S/390企业级服务器的操作系统

S/390 企业级服务器基于新一代 CMOS 电路，运行 OS/390、VM 和 DOS/VSE 操作系统。

IBM 系列计算机从 S/360，S/370，发展到九十年代的 S/390，经过四十年的不断改进，IBM S/390 已成为有高度可靠性、可扩展性、安全性、可用性的现代大型计算机系统。据统计，目前全世界商用数据处理 70%以上都运行了 S/390 企业级服务器。由于中小型服务器组合的公司网因处理能力所限，导致维修成本过高，只能适应现代商业的需要;此外，电子商务的蓬勃发展，大大刺激了对计算能力的需求，导致大型机市场再度升温，用一台或几台大型机代替众多的中小服务器已成为 IT 业户与大势所趋。这也反映了计算机应用遵循：集中—分散—再集中的曲折历程。S/390 大型机及其操作系统 OS/390 便在这样的大趋势下应运而生。最新一代 S/390G6 是当今世界上第一个使用铜质互联芯片技术的企业级服身器，速度达 1600MIPS。　OS/390 的前身是 MVS，1996 年 IBM 宣布 OS390 1.1 版，1998 年 IBM 宣布 OS/390 2.5 版，目前最新版本是 OS/390 2.7 版。OS/390 是一个 基于对象技术的现代开放式的操作系统，支持 X/OpenXPG4 unix 标准。除了 Unix 应用程序可在 OS/390 上运行外，兼容 S/370 上开发的所有应用程序。它支持 TCP/IP 在内的多种通信协议，网络安全性很高。OS/390 采用了面向对象程序设计技术、并行处理技术、分布式处理技术、Client/Server 技术，具有较强的互操作性、可扩展性和可移植性。由于历史的原因，OS/390 有几种不同的运行方式：S/370 模式支持原 S/370 下运行的程序;MVS/ESA390(Enterprise System Architecture,ESA)模式可支持 10 个 240MB 处理器内存和 256 个通道;ESA/390LPAR 模式可把系统从逻辑上分成(Logical Partitioning)最多十个部分，有些多 CPU 型号甚至可分成 20 个 LPAR，每个部分有自己的 CPU、内存和通道，且分别运行不同操作系统，如 OS/370、MVS/ESA370 和 MVS/ESA390，也可以运行 IBM 原有操作系统虚机器操作系统 VM 和虚存扩充操作系统 DOS/VSE。

1.6.9.3 Definity通用服务器操作系统

Definity 通用服务器可运行基于 Intel 的 Windows NT、Netware 等操作系统。

1.6.9.4 AS/400服务器的操作系统

AS/400 服务器首次采用 64 位的 RISC 技术，运行 OS/400 操作系统。

AS/400 服务器是 IBM 开发的最新中型商用机器，有四种类型，小型的用在旅行、家庭、小型办公室中，大型的可用在大型商业应用中，每一类型还有若干型号。各种设备如：工作站、PC 机、磁带机、CD-ROM、远程控制器、打印机等都可连接到服务器上。AS/400 上配置 OS/400 操作系统，在硬件之上自底向上共设置了四层软件:许可证内部代码、OS/400、程序设计支持和应用支持。OS/400 主要提供以下功能：控制语言和菜单、系统操作员服务、程序员服务、工作管理、设备管理、数据管理、消息处理、通信和安全性保证。程序设计支持层提供 C、C++、Cobol、RPG、Java 等语言。应用支持层提供网络管理、工业应用、数控库和系统管理服务。许可证内部代码(Licensed Internal Code,LIC)由 IBM 提供并在提交系统之前预先安装在 AS/400 上的一组用户不可见指令，用户程序需经硬件自动转换成 LIC 才能被 CPU 执行。

OS/400 是在 IBM AS/400 服务器上运行的专有多用户操作系统，其主要特色是内置 IBM DB2 数据库管理系统，最新版本是 4.4。在 AS/400 机器上，硬件、OS、DB、联网、工具、应用软件紧密结合，从而，给用户带来易用、可靠的优点，目前，世界上装机达五十万台。OS400 支持系统逻辑划分，每个有独立的处理器、内存和辅存，这对服务器控制台和商用业务控制台混合的产品来说，AS/400 这种紧密组合非常理想。

1.6.9.5 PC微型机的操作系统

PC 微型机运行 Windows9x、OS2、MS-DOS 等操作系统。

OS/2 是 Microsoft 公司和 IBM 公司合作于 1987 年开发的配置在 PS/2 微机上的图形化用户界面的操作系统。目前，IBM 公司继续在研究和开发 OS/2 2.0，并于 1994 年推出了高性能(新的文件系统和内存管理)、图形界面(Workplace Shell)、高速度、低配置的 32 位抢先多任务操作系统 OS/2 Warp，它的功能和性能与 Windows 95 相当。1996 年，推出网络操作系统 OS/2 最新版本 OS/2 Warp Server4.0 和 OS/2 Warp

Server SMP，后者为对称多处理器版，TCP/IP 变成原生协议，配有支持 6 国语言的声音 I/O 的 Navigator、Java 运行服务也已推出。

OS/2 的主要特点有：

- 采用图形化用户接口，操作直观方便。
- 可以在 16 位和 32 位两种 CPU 上工作。
- 使用虚存可扩充到 4GB。
- 引入会话、进程、线程概念，实现多任务控制。
- 提供高性能文件系统，采用长文件名和扩展文件属性。
- 提供应用程序设计接口（API），可以支持多任务、多线程和动态连接。
- 具有和 MS-DOS 的向上兼容性，MS-DOS 的文件可在 OS/2 下读写。

习题

1. 简述现代计算机系统的组成及其层次结构。
2. 计算机系统的资源可分成哪几类?试举例说明。
3. 什么是操作系统?计算机系统中配置操作系统的主要目标是什么?
4. 操作系统怎样实现计算操作过程的自动化?
5. 为什么对作业进行批处理可以提高系统效率?
6. 举例说明计算机体系结构不断改进是操作系统发展的主要动力之一。
7. 什么是多道程序设计?采用多道程序设计技术有什么特点?
8. 简述实现多道程序设计必须解决的基本问题。
9. 计算机系统采用通道部件后，已能实现处理器与外用设备的并行工作，为什么还要引入多道程序设计技术?
10. 什么是实时操作系统?叙述实时操作系统的分类。
11. 分时系统中，什么是响应时间?它与哪些因素有关?
12. 试比较批处理和分时操作系统的不同点。
13. 试比较单道和多道批外理系统的特点和优缺点。
14. 叙述网络操作系统的主要功能。
15. 叙述分布式操作系统的主要功能。
16. 试比较网络操作系统和分布操作系统。
17. 叙述嵌入式操作系统的发展背景及其特点。
18. 现代操作系统具有哪些基本功能?简单叙述之。
19. 叙述操作系统的用户接口及其使用场合。
20. 叙述现代操作系统的基本特性及其要解决的主要问题。
21. 为什么操作系统会具有随机性特性。
22. 试从资源管理的观点出发，分析操作系统在计算机系统中的角色。
23. 试从服务用户的观点出发，分析操作系统在计算机系统中的角色。
24. 试论述操作系统是建立在计算机硬件平台上的虚拟计算机系统。

CH2 处理器管理

处理器管理是操作系统的重要组成部分，它负责管理、调度和分派计算机系统的重要资源处理器，并控制程序的执行。由于处理器管理是操作系统中最内核的组成部分，任何程序的执行都必须真正占有处理器，因此处理器管理直接影响系统的性能。

本章在简介处理器的硬件运行环境之后，首先注重讨论了计算机系统的中断管理，然后详细介绍了进程和线程的基本概念及其实现，最后全面分析了各个层次的处理机调度方法。

2.1 中央处理器

2.1.1 单处理器系统和多处理器系统

程序最终是要在处理器上执行的，操作系统是一类程序，也要占用处理器运行，以便实现其功能。计算机系统的核心是中央处理器（CPU）。如果一个计算机系统只包括一个处理器，称之为单机系统。如果有多个处理器（不包括 I/O 处理器），称之为多机系统。

早期的计算机系统是基于单个处理器(CPU)的顺序处理的机器，程序员编写串行执行的代码，让其在 CPU 上串行执行，甚至每一条指令的执行也是串行的（取指令、取操作数、执行操作、存储结果）。

为提高计算机处理的速度，首先发展起来的是联想存储器系统和流水线系统，前者提出了数据驱动的思想，后者解决了指令串行执行的问题，这两者都是最初计算机并行化发展的例子。随着硬件技术的进步，并行处理技术得到了迅猛的发展，计算机系统不再局限于单处理器和单数据流，各种各样的并行结构得到了应用。目前计算机系统可以分作以下四类：

- 单指令流单数据流（SISD）：一个处理器在一个存储器中的数据上执行单条指令流。
- 单指令流多数据流（SIMD）：单条指令流控制多个处理单元同时执行，每个处理单元包括处理器和相关的数据存储，一条指令事实上控制了不同的处理器对不同的数据进行了操作。向量机和阵列机是这类计算机系统的代表。
- 多指令流单数据流（MISD）：一个数据流被传送给一组处理器，通过这一组处理器上的不同指令操作最终得到处理结果。该类计算机系统的研究尚在实验室阶段。
- 多指令流多数据流（MIMD）：多个处理器对各自不同的数据集同时执行不同的指令流。

可以把 MIMD 系统划分为共享内存的紧密耦合 MIMD 系统和内存分布的松散耦合 MIMD 系统两大类。在松散耦合 MIMD 系统中，每个处理单元都有一个独立的内存储器，各个处理单元之间通过设定的线路或网络通信，多计算机系统或 cluster 系统都是松散耦合 MIMD 系统的例子。

根据处理器分配策略，紧密耦合 MIMD 系统可以分为主从式系统（main/slave multiprocessor）和对称式系统（symmetric multiprocessor，简称 SMP）两类。

主从式系统的基本思想是：在一个特别的处理器上运行操作系统内核，其他处理器上则运行用户程序和操作系统例行程序，内核负责分配和调度各个处理器，并向其它程序提供各种服务（如输入输出）。这种方式实现简单，但是主处理器的崩溃会导致整个系统的崩溃，并且极可能在主处理器形成性能瓶颈。

在对称式多处理器系统（SMP）中，操作系统内核可以运行在任意一个处理器上，每个处理器都可以自我调度运行的进程和线程，并且操作系统内核也被设计成多进程或多线程，内核的各个部分可以并行执行。

对称多处理机(Symmetric Multiprocessor,SMP)是迄今开发出的最成功的并行机，有一种 SMP 机最多可支持 64 个处理器，多个处理器之间采用共享主存储器。SMP 机有对称性、单一地址空间、低通信延迟和一致的高速缓存等特点，具有高可靠性、可扩充性、易伸缩性。这一系统中任何 CPU 可访问任何存储单元及 I/O 设备；CPU 间通信代价很低，而并行度较高；由于共享存储器中只要保存一个操作系统和数据库副本，既有利于动态负载平衡，又有利于保证数据的完整性和一致性。Dec Alpha Server、HP9000/T600、IBMRS600/40、Sun Ultra Enterprise 6000、SGI Power Challenge XL 都`是 SMP 机，主要用于在线数据服务、数据库和数据仓库等应用。

2.1.2 寄存器

计算机系统的处理器包括一组寄存器，其个数随机型不同而不同，它们构成了一级存储，虽然比主存储器容量要小的多，但是访问速度要快的多。这组寄存器所存储的信息与程序的执行有很大的关系，构成了处理器现场。

不同的处理器具有一组不同的寄存器。一般来说，这些寄存器可以分为以下几类：

- 通用寄存器 可由程序设计者指定许多功能，如存放操作数或用作寻址寄存器。
- 数据寄存器 它们作为内存数据的高速缓存，可以被系统程序和用户程序直接使用并进行计算。用以存放操作数。

- 地址寄存器 用于指明内存地址。如索引寄存器、段寄存器（基址/限长）、堆栈指针寄存器、…、等等。
- I/O 地址寄存器 用于指定 I/O 设备。
- I/O 缓冲寄存器 用于处理器和 I/O 设备交换数据。
- 控制寄存器 用于存放处理器的控制和状态信息，它至少应该包括程序计数器（Program Counter,PC）和指令寄存器（Instruction Register,IR），中断寄存器以及用于存储器和 I/O 模块控制的寄存器也属于这一类。此外还有：存放将被访问的存储单元地址的存储器地址寄存器和存放从存储器读出或欲写入的数据的存储器数据寄存器。
- 程序状态字寄存器也属于 CPU，将在下面介绍。

2.1.3 机器指令

计算机的基本功能是执行程序，而最终被执行的程序是存储在内存中的机器指令。处理器根据程序计数器（PC）从内存中取一条指令到指令寄存器（IR）并执行它，PC 将自动地增长或改变为转移地址以指明下一条执行的指令。

每台计算机机器指令的集合称指令系统，它反映了一台机器的功能和处理能力,可以分为以下四类：

- 数据处理类指令：用于执行算术和逻辑运算。
- 控制类指令：如转移，用于改变执行指令序列。
- 寄存器数据交换类指令：用于在处理器的寄存器和存储器之间交换数据。
- I/O 类指令：用于启动外围设备，让主存和设备交换数据。

2.1.4 特权指令

处理器在运行过程中，将交替执行操作系统程序和用户程序，但是它们能够执行的机器指令集合是不同的，操作系统(指操作系统的核心程序)可使用全部指令，用户程序只能使用指令系统的子集。如果用户程序执行一些有关资源管理的机器指令很容易会产生混乱，如置程序状态字指令将导致处理器占有程序的变更，它只能被操作系统使用；同样启动外围设备进行输入输出的指令也只能在操作系统程序中执行，否则会出现多个用户程序竞争使用外围设备导致 I/O 混乱。

因此，在多道程序设计环境中，从资源管理和控制程序执行的角度出发，必须把指令系统中的指令分作两个部分：特权指令(Privileged Instructions)和非特权指令。所谓特权指令是指那些只能在特态下才能正常执行的，提供给操作系统的核心程序使用的指令，如启动输入输出设备、设置时钟、控制中断屏蔽位、清内存、建立存储键，加载 PSW，…，等。一般用户在目态下运行，只能执行非特权指令，否则会导致非法执行特权指令而产生中断。只有操作系统才能执行全部指令（特权指令和非特权指令）。

2.1.5 处理器状态

那么中央处理器怎么知道当前是操作系统还是一般用户在其上运行呢，这将依赖于处理器状态的标志。在执行不同程序时，根据执行程序对资源和机器指令的使用权限把处理器设置成不同状态。

处理器状态又称为处理器的运行模式，有些系统把处理器状态划分为核心状态、管理状态和用户状态，而大多数系统把处理器状态简单的划分为管理状态（又称特权状态、系统模式、特态或管态）和用户状态（又称目标状态、用户模式、常态或目态）。

当处理器处于管理状态时，可以执行全部指令，使用所有资源，并具有改变处理器状态的能力；当处理器处于用户状态时，只能执行非特权指令。没有硬件支持的多运行模式会引起系统严重后果，如 MS-DOS 是为 Intel8088 结构配的操作系统，它没有双模式，可能发生用户把数据写到操作系统区，或几个用户同时使用一台设备。

PDP 系列计算机具有两个处理器状态, 用户态和核心态。

Pentium 的处理器状态有四种，支持 4 个保护级别，0 级权限最高，3 级权限最`低。一种典型的应用是把 4 个保护级别依次设定为：

- 0 级为操作系统内核级。处理 I/O、存储管理、和其他关键操作。
- 1 级为系统调用处理程序级。用户程序可以通过调用这里的过程执行系统调用，但是只有一些特定的和受保护的过程可以被调用。
- 2 级为共享库过程级。它可以被很多正在运行的程序共享，用户程序可以调用这些过程，都去它们的数据，但是不能修改它们。
- 3 级为用户程序级。它受到的保护最少。

当然，上面的保护级别划分并不是固定的，各个操作系统实现可以采用不同的策略，如运行在 Pentium 上的 Windows 操作系统只使用了 0 级和 3 级,原因是目前所有主流的基于 RISC 处理器的计算机只支持两个特权级，所以，Windows 仅用两个特权级，以保持源代码在 Windows 操作系统支持下的基于 RISC 的结构上兼容。

下面两类情况会导致从用户态向核心态转换，一是程序请求操作系统服务，执行一条系统调用；二

是程序运行时，产生了一个中断事件，中断处理程序进行工作。

Unix 系统上进程执行在两个状态之下：用户态和核心态。在用户态下的进程执行一个系统调用时，进程的执行态从用户态变为核心态，由操作系统执行并试图为用户的请求服务，很多机器都支持更多的处理器状态。Unix 两个处理器状态之间的差别是：用户态下的进程能存取自己的指令和数据,但不能存取内核指令和数据，也不能存取其它进程的指令和数据。然而，核心态下的进程能够存取内核和用户地址。另外，在用户态下的进程不能执行特权指令,这是系统用以保证安全性的措施之一。注意到内核是为用户进程工作的，它不是与用户进程平行运行.的软件，而是作为用户进程的一部分。例如，Shell 通过系统调用读用户终端，在执行读操作时，由用户态转入核心态。于是，正在为该 Shell 进程执行的内核软件对终端的操作进行控制，并把键入的字符返回给 Shell，此时，处理器慨又从核心态转回用户态，然后，Shell 在用户态下继续执行，对字符流作分析和解释完成规定的操作，而解释执行过程中又允许请求引用其它系统调用。

2.1.6 程序状态字寄存器

计算机如何知道当前处于何种工作状态？这时能否执行特权指令？一般都采用设置程序状态字 PSW（Program Status Word）的办法。**程序状态字 PSW** 是 CPU 中的特殊寄存器寄存器，用来控制指令的执行顺序并且保留和指示与程序有关的系统状态的集合。它的主要作用是方便地实现程序状态的保护和恢复。一般来说，程序状态字寄存器包括以下几类内容：

- 程序基本状态。包括：1）程序计数器：指明下一条执行的指令地址；2）条件码：表示指令执行的结果状态；3）处理器状态位：指明当前的处理器状态,如目态或管态、运行或等待。
- 中断码。保存程序执行时当前发生的中断事件。
- 中断屏蔽位。指明程序执行中发生中断事件时，是否响应出现的中断事件。

由于不同处理器中的控制寄存器组织方式不同，程序状态字寄存器可能对应于一个具体的处理器中的一个或一组控制寄存器。IBM 370 的 PSW 寄存器和 PENTIUM 的 EFLAGS 寄存器都是程序状态字寄存器的例子。

IBM360/370 系列计算机的程序状态字分基本格式和扩充格式，均由 64 个二进位组成。IBM 基本格式的 PSW 内容如图 2-1 所示：

- 系统屏蔽位 8 位(0-7 位)　表示允许或禁止某个中断事件发生，相应位为 0 或 1 时分别表示屏蔽或响应。0-7 依次为通道 0-6 和外中断屏蔽位。
- 保护键 4 位(6-11 位)　当没有设置存储器保护时，该 4 位为 0 当设置存储器保护时，PSW 中的这四位保护键与欲访问的存储区的存储键相匹配，否则指令不能执行。
- CMWP 位(12-15 位)　依次为 PSW 基本/扩充控制方式位、开/关中断位、运行/等待位、目态/特态位。
- 中断码 16 位　中断码字段与中断事件对应，记录当前产生的中断源。
- 指令长度字段 2 位(32-33 位)　01/10/11 分别表示半字长指令、整字长指令和一字半长指令。
- 条件码 2 位(34-35 位)
- 程序屏蔽 4 位(36-39 位)　表示允许(为 1) 或禁止(为 0) 程序性中断，自左向右各位体对应的程序性事件是：定点溢出、十进溢出、阶下溢、39 位备用。
- 指令地址 24 位(40-63 位)

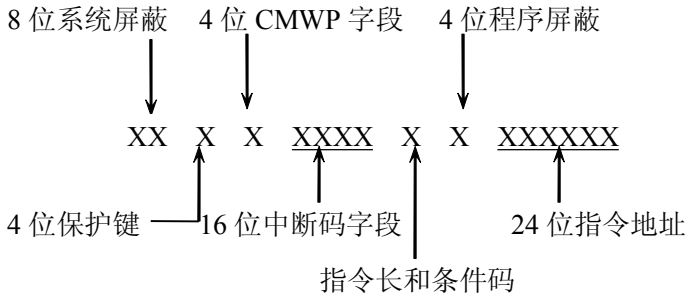


图 2-1　IBM 370 系统程序状态字的其本控制格式

Pentium 的程序状态字由标志寄存器 EFLAGS 和指令指针寄存器 EIP 组成，均为 32 位。EFLAGS 的低 16 位称 FLAGS，可当作一个单元来处理。标志可划分为三组：状态标志、控制标志、系统标志。

- 状态标志　它使得一条指令的执行结果影响后面的指令。算术运算指令使用 OF(溢出标志)，SF(符号标志)，ZF(结果为零标志)，AF(辅助进位标志)，CF(进位标志)，PF(奇偶校验标志)；串扫描、串比较、循环指令使用 ZF 通知其操作结束。
- 控制标志　有以下几位：DF(方向标志)控制串指令操作，设定 DF 为 1，使得串指令自动减量、

即从高地址向低地址处理串操作；DF 为 0 时，串指令自动增量。VM(虚拟 86 方式标志)为 1 时，从保护模式进入虚拟 8086 模式。TF(步进标志) 为 1 时，使处理机执行单步操作。IF(陷阱标志)为 1 时，允许响应中断，否则关中断。

- 系统标志 共三个：IOPL(I/O 特权级标志)、NT(嵌套任务标志)和 RF(恢复标志)，被用于保护模式。指令指针寄存器 EIP 的低 16 位称为 IP，存放下一条顺序执行的指令相对于当前代码段开始地址的一个偏移地址，IP 可当作一个单元使用，这在某些情况下是很有用的。

2.2 中断技术

2.2.1 中断的概念

采用中断技术实现 CPU 和 I/O 设备交换信息可使 CPU 与 I/O 并行工作。在计算机运行过程中，除了会遇到 I/O 中断外，还有许多事件会发生，如硬件故障、电源掉电、人机联系和程序出错、请求操作系统服务等，这些事件必须及时加以处理。此外，在实时系统，如生产自动控制系统中，必须及时将传感器传来的温度、距离、压力、湿度等变化信息送给计算机，计算机则暂停当前工作，转去处理和解决异常情况。所以，为了提高系统效率，处理突发事件，满足实时要求，中断概念被提出来了。**中断**是指程序执行过程中，当发生某个事件时，中止 CPU 上现行程序的运行，引出处理该事件的程序执行的过程。现代计算机系统一般都具有处理突发事件的能力。例如：从磁带上读入一组信息，当发现读入信息有错误时，会产生一个读数据错中断, 操作系统暂停当前的工作，并组织让磁带退回重读该组信息就可能克服错误，而得到正确的信息。

这种处理突发事件的能力是由硬件和软件协作完成的。首先由硬件的中断装置发现产生的事件，然后，中断装置中止现行程序的执行，引出处理该事件的程序来处理。计算机系统不仅可以处理由于硬件或软件错误而产生的事件，而且可以处理某种预定处理伪事件。例如：外围设备工作结束时，也发出中断请求，向系统报告它已完成任务，系统根据具体情况作出相应处理。引起中断的事件称为中断源。发现中断源并产生中断的硬件称中断装置。在不同的硬件结构中，通常有不同的中断源和平不同的中断装置，但它们有一个共性，即：当中断事件发生后，中断装置能改变处理器 内操作执行的顺序,可见中断是现代操作系统实现并发性的基础之一。

2.2.2 中断源的分类

不同硬件结构的中断源各不相同，从中断事件的性质来说，可以分成强迫性中断事件和自愿性中断事件两大类：

强迫性中断事件不是正在运行的程序所期待的，而是由于某种事故或外部请求信息所引起的。这类中断事件大致有以下几种：

- 处理器中断事件。例如电源故障，主存储器出错等。
- 程序性中断事件。例如定点溢出，除数为 0，地址越界等。
- 外部中断事件。例如时钟的定时中断，控制台发控制信息等。
- 输入输出中断事件。例如设备出错，传输结束等。

自愿性中断事件是正在运行的程序所期待的事件。这种事件是由于执行了一条访管指令而引起的，它表示正在运行的程序对操作系统有某种需求，一旦机器执行这一中断时，便自愿停止现行程序而转入访管中断处理程序处理。例如，要求操作系统协助启动外围设备工作。图 2-2 表示出了上述的两类中断事件。

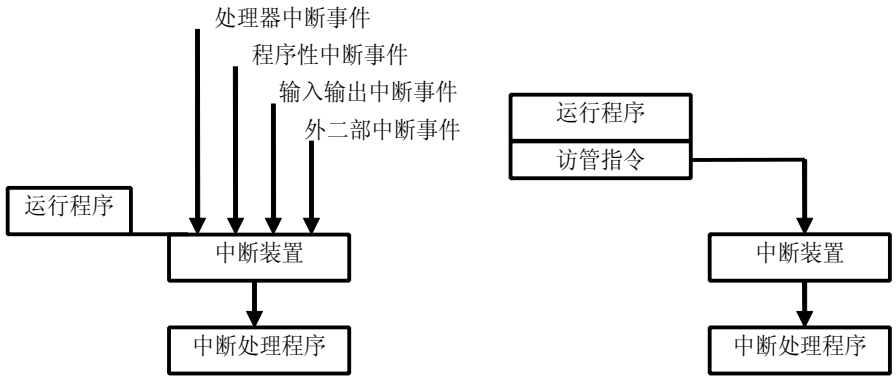


图 2-2　两类中断事件

中断源可以分成两类，一类是不可屏蔽中断，如电源掉电，CPU 不能禁止响应；另一类是可屏蔽中断，通常 PSW 的一些位指示某个可屏蔽中断源是否禁止，CPU 可根据该中断源是否被屏蔽来确定是否给予响应。

有些机器中断源并不分类,依据中断优先级的高低排成中断级，但按中断来源的不同，可以分为两类：中断和捕俘。中断指由 CPU 以外产生的事件引起的中断，如 I/O 中断、时钟中断、外中断;捕俘又称陷入，指 CPU 内部事件或运行程序执行中产生的事件引起的中断，如电源故障、程序故障和访管指令。当中断发生时，内核进行关键处理期间，往往要阻止另一些中断的发生，因为如果允许中断，可能会导致毁灭性后果。如图 2-3 所示，当发生某级中断,响应并处理期间，硬件自动把同级和所有低级的中断源屏蔽掉，而仅允许响应更高级的中断。例如，内核屏蔽掉磁盘中断，则除了机器故障和时钟中断外，所有中断要被禁止；如果仅屏蔽软件中断，则所有中断可能发生。

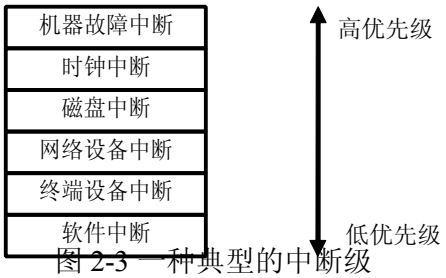


图 2-3 一种典型的中断级

2.2.3 中断装置

迄今为止，所有的计算机系统都采用硬件和软件结合的方法实现中断处理。一般说，中断装置主要做以下三件事：

- 发现中断源，提出中断请求。当发现多个中断源时，它将根据规定的优先级，先后发出中断请求。
- 保护现场。将处理器中某些寄存器内的信息存放于内存储器，使得中断处理程序运行时，不会破坏被中断程序的有用信息， 以便在中断处理结束后它能够继续运行。
- 启动处理中断事件的程序。

通常，中断装置保护现场时，并不一定要将处理器中所有 寄存器中的信息全部存于存储器中。但是，对存放程序状态字的寄存器中的那些信息一定要保护起来。程序状态字用于控制指令执行顺序，并且保留和指示与相应程序有关的系统状态。它通常包括指令地址、中断码和中断屏蔽位等刻画程序运行状态的信息。

当发生中断事件时，中断装置将程序状态字寄存器的内容存放到主存约定单元，然后,将处理中断事件程序的程序状态字送入处理器的现行程序状态字寄存器，从而就引出了处理中断事件的程序。处理中断事件的程序在执行中可能要使用处理器的某些寄存器，因此，它在使用前必须将原有信息保护好，这就不会破坏被中断程序运行的现场了。由此可见，中断装置的后两个职能不过是将程序状态字寄存器中的信息送入主存储器某几个约定单元中，然后从主存储器的几个约定单元中取出信息送入程序状态字寄存器中。或者说，中断装置在响应中断请求后，先存放旧程序状态字(保护运行程序现场)，然后，取出新程序状态字(恢复处理程序现场、即引出中断事件处理程序)。图 2-4 是微型计算机采用堆栈保存被中断程序的状态信息，中断响应时，把现行寄存器内容压进堆栈，并接受中断处理程序的中断向量地址和有关信息，这就引出了中断处理程序。返回原程序时，只要把栈顶内容弹出送入现行寄存器。

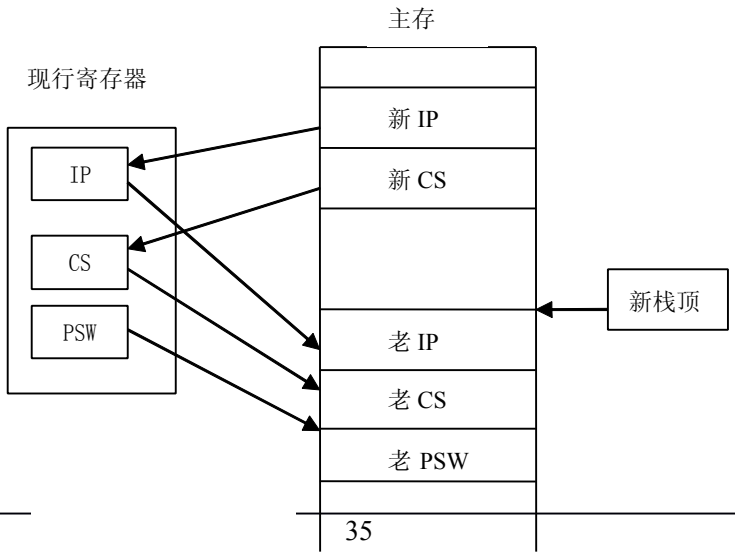


图 2-4 中断处理

中断寄存器是记录强迫性中断事件的寄存器，中断寄存器的内容称中断字。每一种中断可设置一个中断寄存器，对应输入输出中断的中断字为“通道号、设备号”，对应其它中断的中断字，其每一位对应一个中断事件。对每一种中断都有主存的固定单元存放新的和旧的程序状态字。当中断发生后，中断装置把中断寄存器的内容送入程序状态字寄存器的中断码字段，且把中断寄存器清“0”。然后，通过交换程序状态字，把旧程序状态字送到对应中断事件的主存单元，再把相应的新程序状态字送入程序状态字寄存器。处理中断事件的程序执行时从对应的主存固定单元中读出中断字，就可以知道发生的中断事件或知道是哪个外围设备发生了中断事件。

2.2.4 中断事件的处理

2.2.4.1 中断响应和中断处理程序

在 PSW 中已介绍过，响应中断的条件是某中断源或中断级没有屏蔽，有的计算机还设置了开/关所有中断的特征位，如果也是开启的，当有中断事件产生时，CPU 便可响应中断了。由于不同中断源或不同中断级对应不同中断处理程序，故快速找到中断处理程序的入口地址是一个关键问题。寻找入口地址可用如下办法：在主存储器(常在低地址区) 设置一张向量地址表，存储单元的的地址为向量地址，存储单元的的内容为入口地址。CPU 响应中断后，根据予先规定的次序找到相应向量地址，便可获得该中断事件处理程序的入口地址。

处理中断事件的控制程序称为中断处理程序。它的主要任务是处理中断事件和恢复正常操作。

一个操作系统设计者将根据中断的不同类型和不同的应用环境，而确定不同的处理原则。具体地讲，一个中断处理程序主要做以下四项工作：

- 保护未被硬件保护的一些必需的处理状态。例如，将通用寄存器的内容送入主存储器，从而使中断处理程序在运行中可以使用通用寄存器。
- 识别各个中断源，即分析产生中断的原因。
- 处理发生的中断事件。中断处理程序将根据不同的中断源，进行各种处理操作。有简单的操作，如置一个特征标志；也有相当复杂的操作，如重新启动磁带机倒带并执行重读操作。
- 恢复正常操作。恢复正常操作一般有几种情况：恢复中断前的程序按断点执行；重新启动一个新的程序或者甚至重新启动操作系统。

2.2.4.2 处理器中断事件的处理

一般说，这种事件是由硬件的故障而产生，排除这种故障必须进行人工干预。中断处理能做的工作一般是保护现场，防止故障曼延，报告操作员并提供故障信息以便维修和校正，以及对程序中所造成的破坏进行估价和恢复。下面列举一些处理器中断事件的处理办法。

1）电源故障的处理

当电源发生故障，例如掉电时，硬设备能保证继续正常工作一段时间。操作系统利用这段时间可以做以下三项工作：

- 将处理器中有关寄存器内的信息经主存储器送到磁盘保存起来，以便在故障排除后恢复现场继续工作。
- 停止外围设备工作。有些外围设备，例如磁带机，不能立即停止，中断处理程序将把这些正在交换信息又不能立即停止的设备记录下来。
- 停止处理器工作。一般可以让主机处于停机状态，此时，整个系统既不执行指令又不响应中断。

当故障排除后，操作员可以从一个约定点启动操作系统以恢复工作。恢复程序做的主要工作是：

- 恢复中断前的有关现场。
- 启动被停止的外围设备继续工作。
- 如果发生故障时，有不能立即停止的外围设备正在工作，那么涉及这些外围设备的程序将被停止执行而等待操作员的干预命令。

完成上述各项工作后，它将选择可以运行的程序继续运行。

2）主存储器故障的处理

主存储器的奇偶校验或海明校验装置发现主存储器读写错误时，就产生这种中断事件。中断处理程序首先停止与出现的中断事件有关的程序运行，然后向操作员报告出错单元的地址和错误的性质。

2.2.4.3 程序性中断事件的处理

处理程序性中断事件大体上有两种办法。对于那些纯属程序错误而又难以克服的事件，例如非法使用特权指令，企图访问一个不允许其使用的主存储器单元等，操作系统只能将出错程序的名字，出错地点和错误性质报告操作员请求干预。对于其它一些程序性中断，例如定点溢出，阶码下溢等，不同的用

户往往有不同的处理要求。所以，操作系统可以将这种程序性中断事件转交给用户程序，让它自行处理。如果用户程序对发生的中断事件没有提出处理办法，那么，操作系统进行标准处理。

用户怎样来编制处理中断事件的程序呢?有些语言提供了称之为 on 语句的调试语句，它的形式如下：

on 〈条件〉 〈中断续元入口〉

它表示当指定条件的中断发生时，由中断续元来进行处理。例如：

on fixed overflow go to LA;

每当发生定点溢出时，转向以 LA 为标号的语句。对于发生在不同地方的同一种程序性中断事件允许用户采用不同的处理方法。例如，在执行了上述调试语句后又执行调试语句：

on fixed overflow go to LB;

就表示今后再发生溢出时将转向 LB 而不是转向 LA 去处理了。

有了调试语句后，用户用程序设计语言编制程序时，也就可以编写处理程序性中断事件的程序了。编译程序为每个用户设置一张中断续元入口表，且在编译源程序产生目标程序时，把调试语句翻译成：将中断续元入口地址送入中断续元入口表中对应该语句的中断条件的那一栏内。中断续元入口表的形式如下：

中断事件 0	中断事件 1	0	0
中断续元入口 0			
中断续元入口 1			
… … … … … …			
中断续元入口 N			

对应每一个用户处理的中断事件，表格中有一栏用以填写处理该中断事件的中断续元入口地址。如果用户没有给出处理其中断事件的中断续元时，相应栏的内容为 0。执行调试语句时，就将中断续元的入口地址送入相应栏内。显然，对于同一中断事件，当执行第二次对应该事件的调试语句时，就将第二次规定的中断续元入口地址填入表内相应栏中而冲去了第一次填写的内容。这就是上面所说的，利用对同一条件多次使用调试语句时，可以做到对发生于不同地点的同一种中断事件采用不同的处理方法。

当发生中断事件后，操作系统是怎样转交给用户程序去处理的呢？

操作系统只要根据中断事件查看表中对应栏，如果对应栏为“0”它表示用户未定义该类中断续元，此时系统将按规定的标准办法进行处理。例如，将程序停止下来，向操作员报告出错地点和性质，或者置之不顾，就好象什么事也没有发生一样。如果对应栏不为“0”，则强迫用户程序转向中断续元去处理。但是，如果在中继续元的执行中又发生中断事件时，就不能这样简单地处理了。首先中断续元的嵌套一般应规定重数，在上面的表格中规定嵌套重数为 2。表格第一栏的第 0 字节记录了第二次进入中断续元的事件号；第 1 个字节记录了第二次(嵌套)进入中断续元的事件号。其次，中断续元的嵌套不能递归，例如处理定点溢出的中断续元，在执行时又发生定点溢出时就不能转向中断续元处理。

下面按步骤小结一下中断续元的处理过程：

- 编译程序编译到 on 语句时，生成填写相应中断续元入口表的目标代码段
- 程序运行执行到 on 语句时，根据中断条件号，将中断续元入口填入相应栏，这是通过执行上述代码段来实现的
- 执行同一中断条件号的 on 语句时，中断续元入口被填入同一栏，从而，用户可在他的程序的不同部分对同一中断条件采用不同的处理方法
- 每当一个中断条件发生时，检查中断续元入口表相应栏，或转入中断续元处理，或进行操作系统标准处理
- 程序性中断处理允许嵌套，应预先规定嵌套重数，但不允许递归

2.2.4.4 自愿中断事件的处理

这类中断是由于系统程序或用户程序执行访管指令(例如，Unix 中的 trap 指令，MS-DOS 中的 int 指令，IBM 中的 supervisor 指令等)而引起的，它表示运行的程序对操作系统功能的调用，所以也称系统调用，可以看作是机器指令的一种扩充。

访管指令包括操作码和访管参数两部分，前者表示这条指令是访管指令，后者表示具体的访管要求。硬件在执行访管指令时，把访管参数作为中断字并入程序状态字，同时将它送入主存指定单元，然后转向操作系统处理。操作系统分析访管参数，进行合法性检查后按照访管参数的要求进行相应的处理。不

同的访管参数对应

不同的要求，就象机器指令的不同操作码对应不同的要求一样。我们把这样的访管指令也称作广义指令，把访管参数称作广义指令的操作码。

操作系统的基本服务是通过系统调用来处理的，是操作系统为用户程序调用其功能提供的接口和手段。系统调用机制本质上通过特殊硬指令和中断系统来实现。不同机器系统调用命令的格式和功能号的解释不尽相同，但任何机器的系统调用都有共性处理流程。这一共性处理流程如下：

- 用户程序执行 n 号系统调用
- 通过中断系统进入访管中断处理，保护现场，按功能号跳转
- 通过系统调用入口表找到相应功能入口地址
- 执行相应例行程序，结束后正常情况返回系统调用的下一条指令执行

2.2.4.5 外部中断事件的处理

时钟定时中断以及来自控制台的信息都属外部中断事件，它们的处理原则如下：

1）时钟中断事件的处理

时钟是操作系统进行调度工作的重要工具，如让分时进程作时间片轮转、让实时进程定时发出或接收控制信号、定时唤醒或阻塞一个进程、检测陷入死循环的进程、对用户进程进行记帐。时钟可以分成绝对时钟和间隔时钟(即闹钟)两种。

系统设置一个绝对时钟寄存器，计算机的绝对时钟定时地(例如每 20 毫秒)把该寄存器的内容加 1。如果开始时这个寄存器的内容为 0，那么只要操作员告诉系统开机时的年、月、日、时、分、秒，以后就可一推算出当前的年、月、日、时、分、秒了。当绝对时钟寄存器记满溢出时，就产生一次绝对时钟中断，操作系统处理这个中断时，只要在主存的固定单元上加 1 就行了。这个固定单元记录了绝对时钟中断的次数，这样就可保证有足够的计时量。计算当前时间时，只要按绝对时钟中断的次数和绝对时钟寄存器的内容推算就可得到。

间隔时钟是定时将一个间隔时钟寄存器的内容减 1，当间隔时钟寄存器的内容为 0 时，就产生一个间隔时钟中断。所以，只要在间隔时钟寄存器中放一个预定的值，那么就可起到闹钟的作用，每当产生一个间隔时钟中断，就意味着预定的时间到了。操作系统经常利用间隔时钟作控制调度。

时钟硬件做的工作仅仅是按已知时间间隔产生中断，其余与时间有关的任务必须由软件来做，不同的操作系统有关时钟的任务不同，但一般包括以下内容：

- 维护绝对日期和时间;
- 防止进程的运行时间超出其允许值，发现陷入死循环的进程;
- 对使用 CPU 的用户进程记帐;
- 处理进程的间隔时钟(闹钟);
- 对系统的功能或部件提供监视定时器。

2）控制台中断事件的处理

操作员可以利用控制台开关请求操作系统工作，当使用控制台开关后，就产生一个控制台中断事件通知操作系统。操作系统处理这种中断就如同接受一条操作命令一样，转向处理操作命令的程序执行。

2.2.5 中断的优先级和多重中断

2.2.5.1 中断的优先级

在计算机执行的每一瞬间，可能有几个中断事件同时发生。例如，由非法指令引起程序性中断的同时可能发生外部中断要求。这时，中断装置如何来响应这些同时发生的中断呢?一般说，中断装置按照预定的顺序来响应，这个预定的顺序称为中断的优先级，中断装置首先响应优先级高的中断事件。

在一个计算机系统中，各中断源的优先顺序是根据某个中断源或中断级若得不到及时响应，造成计算机出错的严重性程度而定。例如，机器校验中断表明产生了一个硬件故障，对计算机及当前执行任务的影响最大，因而，优先级排在最高；至于 I/O 中断事件的优先级可放低，这样做只会推迟一次 I/O 启动或推迟处理一个 I/O 事件，对系统工作的正确性影响不大。当某一时刻有多个中断源或中断级提出中断请求时，中断系统如何按予先规定的优先顺序响应呢？可以使用硬件和软件两种办法。前者根据排定的优先次序做一个硬件链式排队器，当有高一级的中断事件产生时，应该封住比它优先级低的所有中断源;后者编写一个查询程序，依据优先级次序，自高到低进行查询，一旦发现有一个中断请求，便转入该中断事件处理程序入口。

例如，1BM360/370 系统的中断优先级由高到低的顺序是：机器校验中断；自愿访管中断；程序性中断；外部中断；输入输出中断；重新启动中断。读者注意，中断的优先级只是表示中断装置响应中断的次序，而并不表示处理它的先后顺序。

2.2.5.2 中断的屏蔽

主机可以允许或禁止某类中断的响应。如主机可以允许或禁止所有的输入输出中断、外部中断、机器校验中断以及某些程序中断。对于被禁止的中断，有些以后可继续响应，有些将被丢弃。例如，对于被禁止的输入输出中断的条件将被保留以便以后响应和处理，对于被禁止的程序中断条件，除了少数置

特征码以外，都将丢弃不管。

有些中断是不能被禁止的，例如，计算机中的自愿访管中断就不能被禁止。

主机是否允许某类中断；由当前程序状态字中的某些位来决定。一般，当屏蔽位为 1 时，主机允许相应的中断，当屏蔽位为 0 时，相应中断被禁止。按照屏蔽位的标志，可能禁止某一类内的全部中断，也可能有选择地禁止某一类内的部分中断。

2.2.5.3 多重中断事件的处理

在一个计算机系统运行过程中，由于中断可能同时出现，或者虽不同时出现但却被硬件同时发现，或者出现在其它中断正在进行处理期间，这时 CPU 又响应了这个新的中断事件，于是暂时停止正在运行的中断处理程序，转去执行新的中断处理程序，这就叫多重中断(又称中断嵌套) 。一般说，优先级高的中断有打断优先级别低的中断处理程序的权利，但反之,不允许优先级别低的中断干扰优先级别高的中断处理程序的运行。系统如何来处理出现的多个中断呢?这是操作系统的中断处理程序所必须解决的问题。

对于多个中断，可能是同一中断类型的不同中断源，也可能是不同类型的中断。对于前者，一般由同一个中断处理程序按预定的次序分别处理之；对于后者，我们将区别不同情况作如下处理：

- 在运行一个中断处理程序时，往往屏蔽某些中断；例如，在运行处理 I/O 中断的例行程序时，可以屏蔽外部中断或其它 I/O 中断。
- 如前所述，中断可以分优先级，对于有些必须处理且优先级更高的中断源，采用屏蔽方法有时可能是不妥的，因此，在中断系统中往往允许在运行某些中断例行程序时，仍然可以响应中断，这时，系统应负责保护被中断的中断处理例行程序的现场(有的计算机中断系统对断点的保存是在中断周期内，由中断隐指令实现，对用户是透明的)，然后，再转向处理新中断的例行程序，以便处理结束时有可能返回原来的中断处理例行程序继续运行。操作系统必须预先作出规定，哪些中断类型允许嵌套？嵌套的最大级数？嵌套的级数视系统规模而定，一般不超过三重为宜，因为过多重的‘嵌套’将会增加不必要的系统开销。
- 在运行中断处理例行程序时，如果出现任何程序性中断源，一般情况下，表明这时中断处理程序有错误，应立即响应并进行处理。

每个中断处理程序的程序状态字中，究竟应该屏蔽哪些中断源；将由系统设计而定，需要考虑的情况有：硬件的中断优先级，应用的需要，软件处理所希望的优先级，可能丢失的中断源及其对系统的影响等。

Unix 的中断和捕俘允许多重中断，中断优先级共 6 级,优先级从高到低为：机器故障、时钟、磁盘 I/O、网络设备、终端和软件中断。紧急中断事件优先级高，用户程序运行时中断事件优先级最低，可见任何中断事件都能中断用户程序运行。内核开辟了一个中断现场保护区-内核堆栈，总把当前被中断程序的现场信息压进堆栈。例如，若一个用户程序发出了一条系统调用，于是该用户程序被暂停，用户程序现场信息压进堆栈。在此系统调用处理期间，磁盘 I/O 中断发生，这时系统调用处理程序被中断，系统调用处理程序现场信息压入堆栈。在磁盘 I/O 中断处理程序处理期间，又发生了时钟中断，这时磁盘 I/O 中断处理程序被中断，现场信息进入堆栈，时钟中断处理程序开始工作。返回时逐级上推内核堆栈，依次逐级返回。

2.2.6 实例研究：Windows 2000 的中断处理

2.2.6.1 Windows 2000的中断处理概述

在 Pentium 上 Windows 中断的实现中，使用了陷阱、中断和异常等术语。

中断和异常是把处理器转向正常控制流之外的代码的操作系统情况，这两个术语相当于本节前面所讨论的“中断概念”。内核按照以下的方法来区分中断和异常。中断是异步事件，可能随时发生，与处理器正在执行的内容无关。

中断主要由 I/O 设备、处理器时钟或定时器、以及软件等产生，可以启用或禁用。

异常是同步事件，它是某一个特定指令执行的结果。异常的一些例子是内存访问错误、调试指令、除数为零。内核也将系统服务调用视作异常，在技术也可以把它作为系统陷阱。

陷阱是指这样一种机制，当中断或异常发生时，它俘获正在执行的进程，把它从用户态切换到核心态，并将控制权交给内核的陷阱处理程序。不难看出，陷阱机制相当于本节前面所讨论的中断响应和中断处理机构。

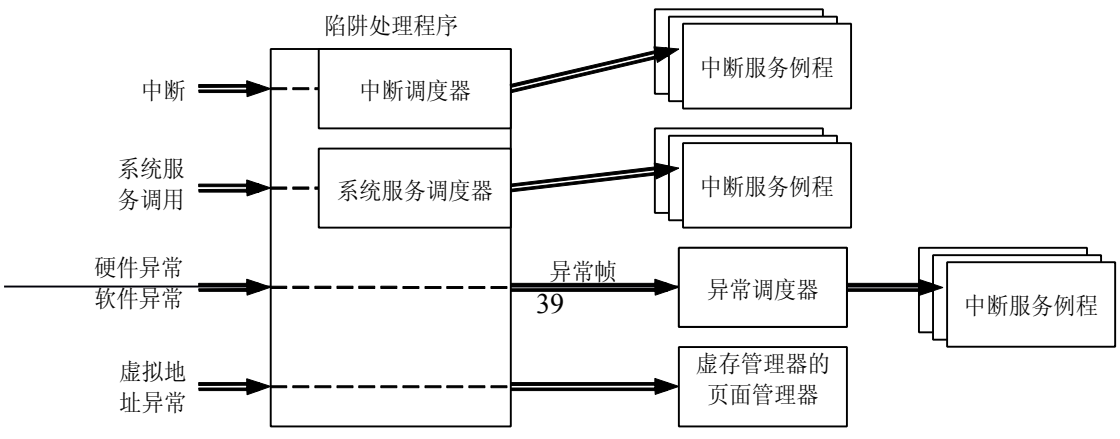


图 2-5　Windows2000 的陷阱调度

图 2-5 说明了激活陷阱处理程序的情况和由陷阱处理程序调用来为它们服务的模块。陷阱处理程序被调用时，它在保存机器状态（机器状态将在另一个中断或异常发生时抹去）期间，暂时禁用中断。同时，创建一个“陷阱帧”来保存被中断线程的执行状态，当内核处理完中断或异常后恢复线程执行前，通过陷阱帧中保存的信息恢复处理器现场。陷阱处理程序本身可以解决一些问题，如一些虚拟地址异常，但在大多数情况下，陷阱处理程序只是确定发生的情况，并把控制转交给其它的内核或执行体模块。例如，如果情况是设备中断产生的，陷阱处理程序把控制转交给设备驱动程序提供给该设备的中断服务例程 ISR(Interrupt Service Routine)；如果情况是调用系统服务产生的，陷阱处理程序把控制转交给执行体中的系统服务代码；其它异常由内核自身的异常调度器响应。

2.2.6.2 Windows 2000的中断调度

1）中断类型和优先级

由于不同处理器能识别不同的中断号和中断类型,为此，先由 Windows2000 的中断调度器将中断级映射到由有操作系统识别的中断请求级 IRQL(Interrupt Request Level)的标准集上。这一组内核维护的 IRQL 是可以移植的，如果处理器具有特殊的与中断相关的特性（如第二时钟），则可以增加可移植的 IRQL。IRQL 将按照优先级排列中断，并按照优先级顺序服务中断，较高优先级中断抢占较低优先级中断服务。

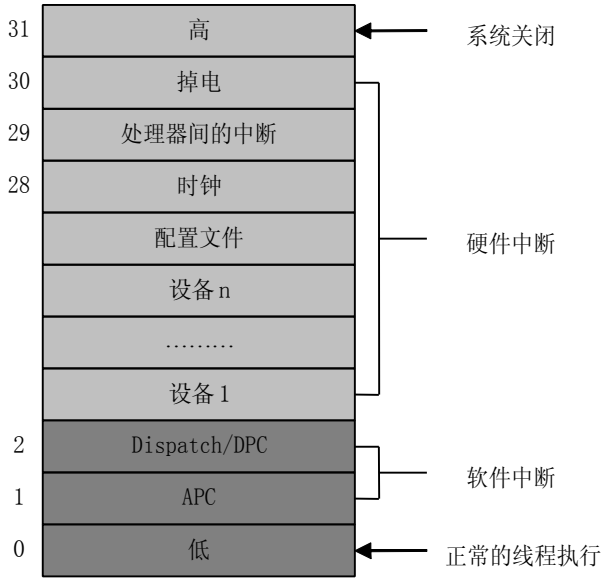


图 2-6　x86 体系结构 Windows 的中断请求级

图 2-6 显示了 x86 体系结构上可移植 IRQL 的映射。IRQL 从高往下直到设备级都是为硬件中断保留的，Dispatch/DPC 和 APC 中断是内核和设备驱动程序产生的软件中断，并不是真正的中断级，在该级上运行普通线程，并允许发生所有中断。

每一个处理器都有一个 IRQL 设置，其值随着操作系统代码的执行而改变，决定了该处理器可以接收哪些中断。IRQL 也被用于同步访问核心数据结构。当核心态线程运行时，它可以提高或降低处理器的 IRQL。如图 2-7 所示，如果中断源高于当前的 IRQL 设置，则响应中断；否则该中断将被屏蔽，处理器不会响应该中断，直到一个正在执行的线程降低了 IRQL。

核心态根据它要做的事情来提高或降低它所使用的处理器的 IRQL,例如，当中断产生时，陷阱处理程序提高处理器的 IRQL 直到与中断事件所指定的 IRQL 相同。这种提高了所有等于或低于此 IRQL 的中断(仅在这个处理器上),确保正在服务于该中断的处理器不会被同级或较低级的中断抢先。被屏蔽的中断将被另一个处理器处理或阻挡，直到 IRQL 降低。由于改变处理器的 IRQL 对操作系统具有十分重要的影响，所以，它只能在核心态下改变，用户态线程无权改变 IRQL。

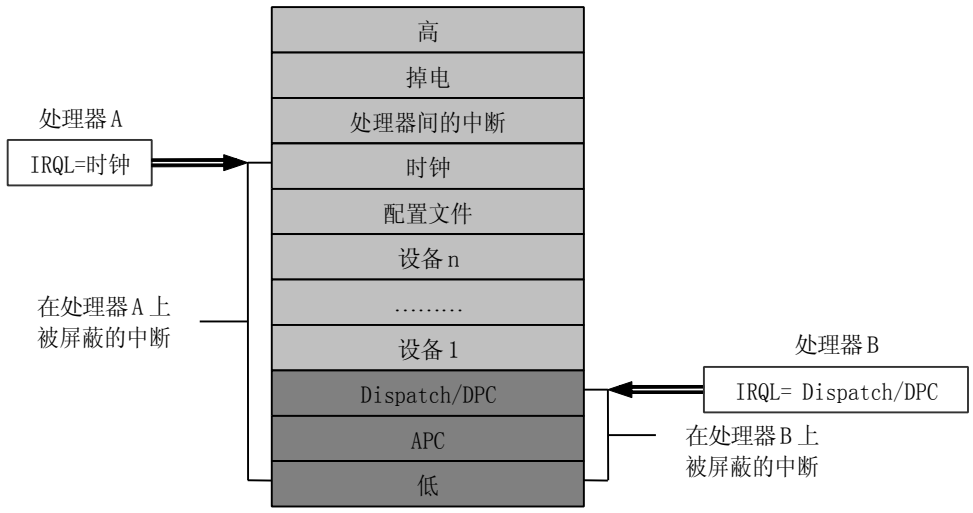


图 2-7　Windows 的中断屏蔽

2）中断处理

当中断产生时，陷阱处理程序将保存计算机运行程序的状态，然后禁用中断并调用中断调度程序。中断调度程序立刻提高处理器的 IRQL 到中断源的级别，以使得在中断服务过程中屏蔽等于和低于当前中断源级别的其他中断。然后，重新启用中断，以使高优先级的中断仍然能够得到服务。

Windows2000 使用中断分配表 IDT(Interrupt Dispatch Table)来查找处理特定中断的例程。中断源的 IRQL 作为表的索引，表的入口指向中断处理例程，如图 2-8 所示。

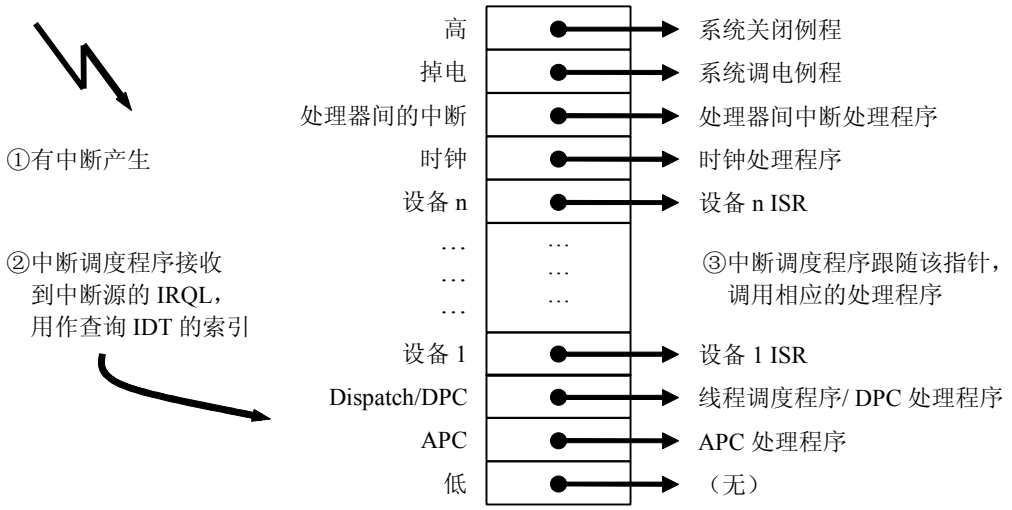


图 2-8　Windows 的中断服务

在 x86 系统中，IDT 是处理器控制区 PCR(Processing Control Region)指向的硬件结构，有的系统用软件实现。PCR 和它的扩展——处理器控制块 PRCB 包括了系统中各种处理器状态信息。内核和硬件抽象层 HAL 使用该信息来执行体系结构特定的操作和机器特定的操作。这些信息包括：当前运行线程、选定下一个运行的线程、处理器的中断级等等。

在 x86 的体系结构中，中断控制器可以支持 256 个中断行，但是特定机器可以支持的中断行数量仍旧依赖于具体的中断控制器设计，大多数 x86 PC 机的中断控制器使用 16 个中断行。中断实际上进入了中断控制器的某一行。中断控制器依次在单个行上中断处理器。一旦处理器被中断，它将询问控制器以获得中断向量。处理器利用此中断向量索引进入 IDT 并将控制交给适当的中断服务例程。

在中断服务例程执行之后，中断调度程序将降低处理器的 IRQL 到该中断发生前的级别，然后加载保存的机器状态。被中断的线程将从它停止的位置继续执行。在内核降低了 IRQL 后，被封锁的低优先级中断就可能出现。在这种情况下，内核将重复以上过程来处理新的中断。

每个处理器都有单独的 IDT，这样，不同的处理器就可以运行不同的中断服务例程 ISR。在多处理器系统中，每个处理器都可以收到时钟中断，但只有一个处理器在响应该中断时更新系统时钟。然而所有处理器都使用该中断来测量线程的时间片并在时间片结束后启动线程调度。同样的，某些系统配置可能

要求特殊的处理器处理某一设备中断。

大多数处理中断的例程都在内核中，例如：内核更新时钟时间，在电源级中断产生时关闭系统。然而，键盘、I/O 设备和磁盘驱动器等外部设备也会产生许多中断，这些设备的种类很多、变化很大,设备驱动程序需要一种方法来告诉内核：当设备中断发生时应调用哪个例程。为此，内核提供了一个可移植的机制——中断对象，它是一种内核控制对象，允许设备驱动程序注册其设备的 ISR。中断对象包含内核所需的将设备 ISR 和中断特定级相联系的所有信息，其中有：ISR 地址、设备中断的 IRQL、以及与 ISR 相联系的内核入口。当中断对象被初始化后，称为“调度代码”的一些汇编语言代码指令就会被存储在对象中。当中断发生时执行此代码，这个中断对象常驻代码调用真正的中断调度程序，给它传递一个指向中断对象的指针。中断对象包括了第二个调度程序例程所需要的信息，以便定位和正确使用设备驱动程序提供的 ISR。

把 ISR 与特殊中断级相关联称为连接一个中断对象,而从 IDT 入口分离 ISR 叫做断开一个中断对象。这些操作允许在设备驱动程序加载到系统时打开 ISR，在卸载设备驱动程序时关闭 ISR。如果多个设备驱动程序创建多个中断对象并将它们连接到同一个 IDT 入口，那么当中断在指定中断级上发生时，中断调度程序会调用每一个例程。这样就使得内核很容易地支持“菊花链”配置，在这种构造中几个设备在相同的中断行上中断。

使用中断对象来注册 ISR，可以防止设备驱动程序直接随意中断硬件，并使设备驱动程序无需了解 IDT 的任何细节，从而有助于创建可移植的设备驱动程序。另外，通过使用中断对象，内核可以使 ISR 与可能同 ISR 共享数据的设备驱动程序的其他部分同步执行。进而，中断对象使内核更容易调用多个任何中断级的 ISR。

3）软件中断

虽然硬件产生了大多数中断，Windows2000 也为多种任务产生软件中断，他们包括：启动线程调度、处理定时器到时、在特定线程的描述表中异步执行一个过程、支持异步 I/O 操作等。

（1）调度或延迟过程调用 DPC(Deferred Procedure Call)中断

当一个线程不能继续执行时，内核应该直接调用调度程序实现描述符表切换。然而，有时内核在深入多层代码内检测到应该进行重调度，最好的方法就是请求调度，但应延迟调度的产生直到内核完成当前的活动为止。使用 DPC 软件中断是实现这种延迟的简单方法。

当需要同步访问共享的内核结构时，内核总是将处理器的 IRQL 提高到 Dispatch/DPC 级之上，这样就禁止了其他的软件中断和线程调度。当内核检测到调度应该发生时，它将请求一个 Dispatch/DPC 级中断；但是由于 IRQL 等于或高于 Dispatch/DPC 级，处理器将在检查期间保存该中断。当内核完成了当前活动后，它将 IRQL 降低到 Dispatch/DPC 级之下，于是调度中断就可以出现。

除了延迟线程调度之外，内核在这个 IRQL 上也处理延迟过程调用 DPC。DPC 是执行系统任务的函数，该任务比当前任务次要，被称作‘延迟函数’,因为,他们可能不立即执行。DPC 为操作系统提供了在内核态下产生中断，并执行系统函数的能力。内核使用 DPC 处理定时器到时并释放在定时器上等待的线程,在线程时间片结束后重调度处理器。设备驱动程序还可以通过 DPC 完成延迟的 I/O 请求。

DPC 由“DPC 对象”表示，它也是一个内核控制对象。该对象对于用户态程序是不可见的，但对于设备驱动程序和其他系统代码是可见的。DPC 对象包含的最重要信息是当内核处理 DPC 中断时将调用的系统函数的地址。等待执行的 DPC 例程被保存在叫做 DPC 队列的内核管理队列中。为了调用一个 DPC,系统代码将调用内核来初始化 DPC 对象，并把它放入 DPC 队列中。

将一个 DPC 放入 DPC 队列会促使内核请求一个在 Dispatch/DPC 级的中断。因为通常 DPC 是运行在较高 IR QL 级的软件对它进行排队的，所以被请求中断直到内核降低 IRQL 到 Dispatch/DPC 级之下才出现，图 2-9 描述了 DPC 的处理。

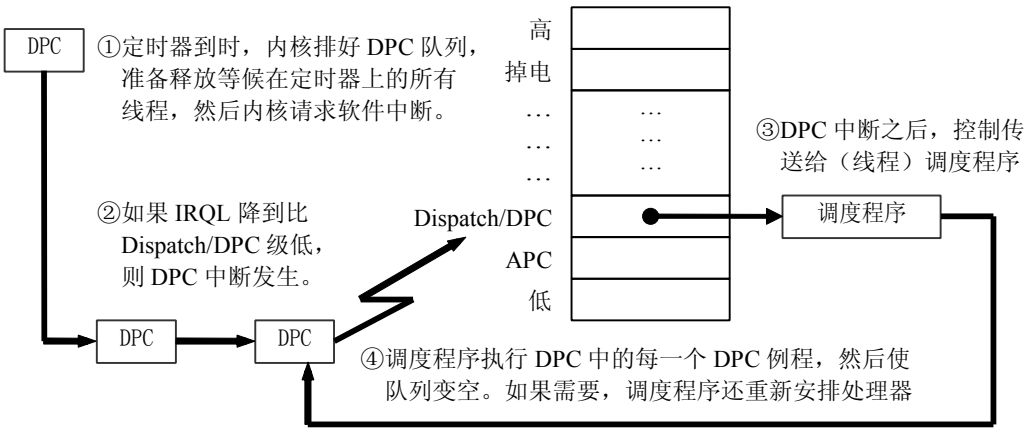


图 2-9　提交 DPC

DPC 主要是为设备驱动程序提供的，但内核也使用它。内核经常使用 DPC 去处理时间片到，系统时钟的每个跳动，在时钟 IRQL 都产生一个中断。运行在时钟 IRQL 级的时钟中断处理程序使更新系统时间，减小当前线程运行时间的计数器的值,当计数值达到零时，线程的本次时间片用光，内核的调度程序需要重新调度 CPU，这是一个应该在 Dispatch/DPC IRQL 完成的低优先级任务。时钟中断处理程序对 DPC 排队以启动线程调度，然后完成它的工作并降低处理器的 IRQL。因为 DPC 中断的优先级低于设备中断的优先级，所以任何挂起的设备中断将在 DPC 中断产生之前得到处理。

（2）异步过程调用中断

异步过程调用 APC(Asynchronous procedure Call)为用户程序/系统代码提供了一种在特殊用户线程的描述表（一个特殊的地址空间）中执行代码的方法。

像 DPC 一样，APC 由内核控制对象描述，称为 APC 对象，等待执行的 APC 在由内核管理的 APC 队列中。APC 队列和 DPC 队列的不同之处在于：DPC 队列是系统范围的，而 APC 队列是属于每个线程的。当内核被要求对 APC 排队时，内核将 APC 插入到将要执行 APC 例程的线程的 APC 队列中。内核依次请求 APC 级的软件中断，并当线程最终开始运行时，执行 APC。

有两种 APC，用户态 APC 和核心态 APC。核心态 APC 在线程描述表中运行并不需要得到目标线程的允许，而用户态 APC 则需要得到认可。核心态 APC 还可以中断线程及执行过程，而不需要线程的干预和同意。

执行体使用核心态 APC 来执行必须在特定线程地址空间中完成的操作系统工作，如可以使用核心态 APC 命令一个线程停止执行可中断的系统服务。设备驱动程序也使用核心态 APC，如启动了一个 I/O 操作并且线程进入等待状态，则另一个进程中的另一个线程就可以被调度而去运行；当设备完成传输数据时，I/O 系统必须以某种方式恢复进入到启动 I/O 系统线程的描述表中，以便它能够将 I/O 操作的结果复制到包含该线程地址空间的缓冲区内；I/O 系统利用核心态 APC 来执行如上的操作。

几个 WIN32 函数，如 ReadFileEx、WriteFileEx 和 QueueUserAPC，使用用户态 APC。该完成例程是通过把 APC 排队到发出 I/O 操作的线程来实现的。

2.2.6.3 异常调度

与随时可能发生的中断相比，异常是直接由运行程序的执行产生的情况。WIN32 引入了结构化异常处理(Structures Exception Handling)工具，它允许应用程序在异常发生时可以得到控制。然后，应用程序可以固定这个状态并返回到异常发生的地方，从而终止引发异常的子例程的执行；也可以向系统声明不能识别异常，并继续搜寻能处理异常的异常处理程序。

除了那些简单的可以由陷阱处理程序解决的异常外，所有异常均由异常调度程序提供服务。异常调度程序的任务是找到能够处理该异常的异常处理程序。

如果异常产生于核心态，异常调度程序将简单地调用一个例程来定位处理该异常的基于框架的异常处理程序。由于没有被处理的核心态异常是一种致命的操作系统错误，所以异常调度程序必须能找到异常处理程序。

内核俘获和处理某些对用户程序透明的异常，如调试断点异常,此时内核将调用为试程序来处理这个异常。少数异常可以被允许原封不动地过滤回用户态，如操作系统不处理的内存越界或算术异常。环境子系统能够建立“基于框架的异常处理程序”来处理这些异常。“基于框架的异常处理程序”是指与特殊过程激活相关的异常处理程序，当调用过程时，代表该过程激活的堆栈框架就会被推入堆栈，堆栈框架可以有一个或多个与它相关的异常处理程序，每个程序都保存在源程序的一个特定代码块内。当异常发生时，内核将查找与当前堆栈框架相关的异常处理程序，如果没有，内核将查找与前一个堆栈框架相关的异常处理程序，如此往复，如果还没有找到，内核将调用系统默认的异常处理程序。

当异常发生时都将在内存产生一个事件链。硬件把控制交给陷阱处理程序，陷阱处理程序将创建一个陷阱框架。如果完成了异常处理，陷阱框架将允许系统从中断处继续运行。陷阱处理程序同时还要创建一个包含异常原因和其他有关信息的异常纪录。

2.2.6.4 系统服务调度

在 x86 处理器上执行 INT 2E 指令将引起一个系统陷阱，进入系统服务调度(System Service Dispatcher)。如图 2-10 所示，内核将根据入口参数在系统服务调度表中查找系统服务信息。

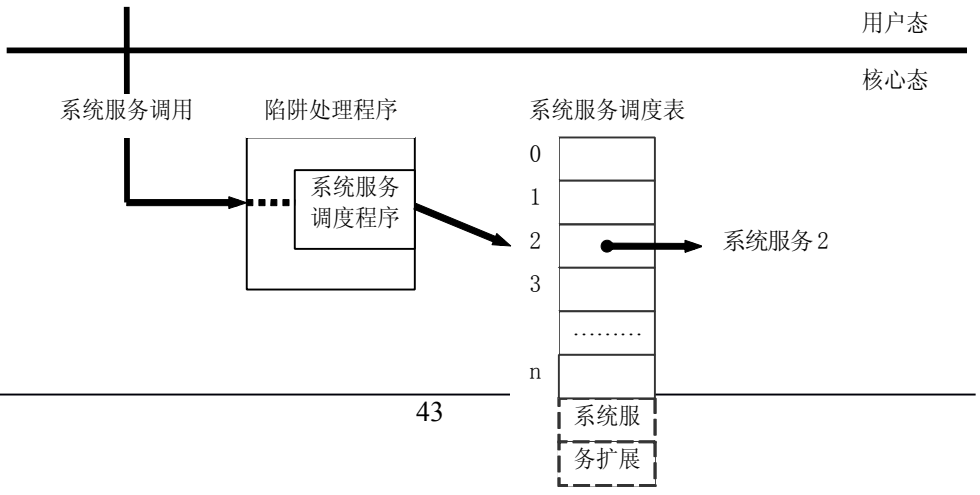


图 2-10　Windows200 的系统服务异常

系统服务调度程序将校验正确的参数最小值，并且将调用者的参数从线程的用户态堆栈复制到它的核心态堆栈中，然后执行系统服务。如果传递给系统服务的参数指向了在用户空间中的缓冲区，则在核心态代码访问用户缓冲区前，必须查明这些缓冲区的可访问性。

每个线程都有一个指向系统服务表的指针。Windows2000 有两个内置的系统服务表，第一个默认表定义了在 NTOSKRNL.EXE 中实现的核心执行体系统服务；另一个包含了在 WIN32 子系统 WIN32K.SYS 的核心态部分中实现的 WIN32 USER 及 GDI 服务。当 WIN32 线程调用 WIN32 USER 及 GDI 服务时，线程系统服务表的地址将指向包含 WIN32 USER 及 GDI 的服务表。

用于 Windows2000 执行体服务的系统服务调度指令存在于系统库 NTDLL.DLL 中。子系统的 DLL 通过调用 NTDLL 中的函数来实现它们的文档化函数。例外的是，WIN32 USER 及 GDI 函数的系统服务调度指令是在 USER32.DLL 和 GDI32.DLL 中直接实现。

如图 2-11 所示，KERNEL32.DLL 中的 WIN32 WriteFile 函数调用 NTDLL.DLL 中的 NtWriteFile 函数，它依次执行适当的指令以引发系统陷阱，传递代表 NtWriteFile 的系统服务号码。然后系统服务调度程序（NTOSKRNL.EXE 中的 KiSystemService 函数）调用真正的 NtWriteFile 来处理 I/O 请求。对于 WIN32 USER 及 GDI 函数，系统服务调度调用在 WIN32 子系统可加载核心态部分 WIN32K.SYS 中的函数。

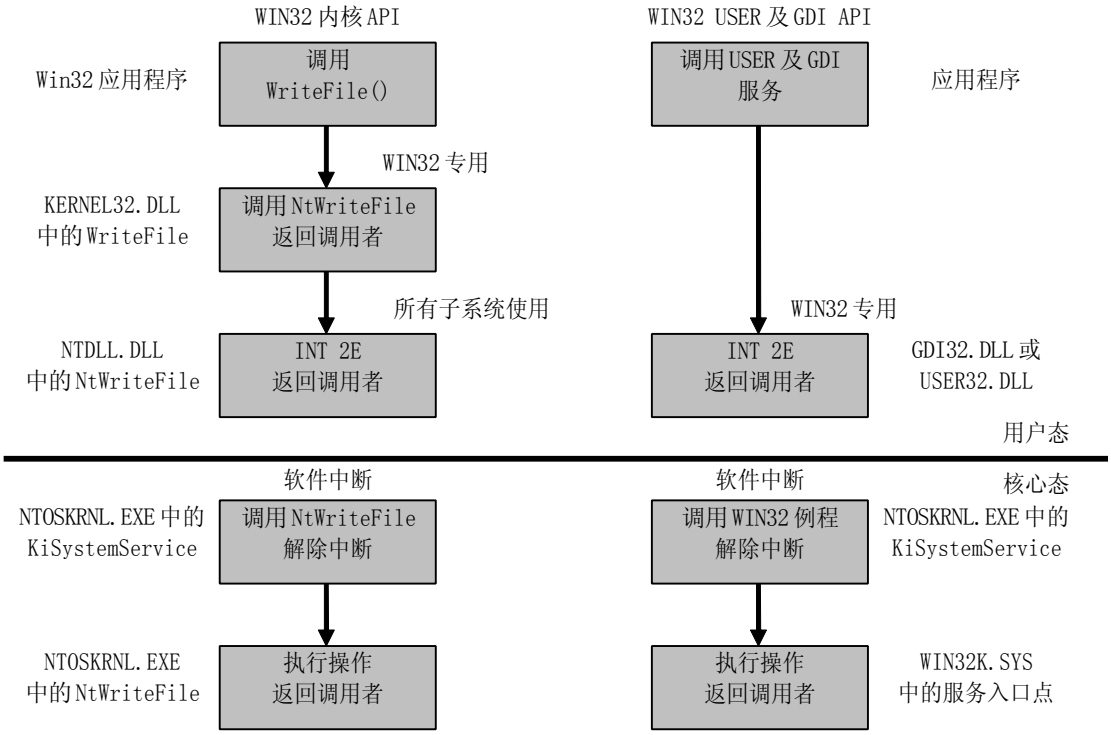


图 2-11　Windows200 的系统服务调度

2.3 进程及其实现

2.3.1 进程的定义和性质

进程的概念是操作系统中最基本、最重要的概念。它是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规律而引进的一个新概念,所有的多道程序设计操作系统都建立在进程的基础上。操作系统专门引入进程的概念，从理论角度看，是对正在运行的程序过程的抽象；从实现角度看，则是一种数据结构，目的在于清晰地刻划动态系统的内在规律，有效管理和调度进入计

算机系统主存储器运行的程序。

进程（process）这个名词最早是在 MIT 的 MULTICS 和 IBM 公司的 TSS/360 系统中于 1960 年提出的，直到目前对进程的定义和名称均不统一，不同的系统中采用不同的术语名称，例如，MIT 称进程 IBM 公司称任务(Task)和 Univac 公司称活动(Active)。进程的定义也是多种多样的，国内学术界较为一致的看法是：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动（1978 年全国操作系统学术会议）。从操作系统管理的角度出发，进程由数据结构以及在其上执行的程序(语句序列)组成，是程序在这个数据集合上的运行过程，也是操作系统进行资源分配和保护的基本单位。它具有如下属性：

- 结构性：进程包含了数据集合和运行于其上的程序。
- 共享性：同一程序同时运行于不同数据集合上时，构成不同的进程。或者说，多个不同的进程可以共享相同的程序。
- 动态性：进程是程序在数据集合上的一次执行过程，是动态概念，同时，它还有生命周期，由创建而产生，由撤销而消亡；而程序是一组有序指令序列，是静态概念，所以，程序作为一种系统资源是永久存在的。
- 独立性：进程既是系统中资源分配和保护的基本单位，也是系统调度的独立单位(单线程进程)。凡是未建立进程的程序，都不能作为独立单位参与运行。通常，每个进程都可以各自独立的速度在 CPU 上进行。
- 制约性：并发进程之间存在着制约关系，进程在进行的关键点上需要相互等待或互通消息，以保证程序执行的可再现性和计算结果的唯一性。
- 并发性：进程可以并发地执行。对于一个单处理器的系统来说，m 个进程 P_1，P_2，…，P_m 是轮流占用处理器并发地执行。例如可能是这样进行的：，进程 P_1 执行了 n_l 条指令后让出处理器给 P_2, P_2 执行了 n_2 条指令后让出处理器给 P_3, …, P_m 执行了 n_m 条指令后让出处理器给 P_1, …。因此，进程的执行是可以被打断的，或者说，进程执行完一条指令后在执行下一条指令前，可能被迫让出处理器，由其它若干个进程执行若干条指令后才能再次获得处理器而执行。

同静态的程序相比较，进程依赖于处理器和主存储器资源，具有动态性和暂时性，进程随着一个程序模块进入主存储器并获得一个数据块和一个进程控制块而创建，随着运行的结束退出主存储器而消亡，从创建到消亡期间，进程处于不断的动态变化之中。从进程的定义和属性看出它是并发程序设计的一种有力工具，操作系统中引入进程概念能较好地解决系统的“并发性”和刻划系统的“动态性”。

我们也可以从解决“共享性” 来看操作系统中引入进程概念的必要性。首先，引入“可再入” 程序的概念，所谓“可再入”程序是指能被多个程序同时调用的程序。另一种称“可再用” 程序由于它被调用过程中具有自身修改，在调用它的程序退出以前是不允许其它程序来调用它的。“可再入”程序具有以下性质：它是纯代码的，即它在执行中自身不改变；调用它的各程序应提供工作区，因此，可再入的程序可以同时被几个程序调用。

在多道程序设计的系统里，编译程序常常是“可再入”程序，因此它可以同时编译若干个源程序。假定编译程序 P 现在正在编译源程序甲，编译程序从 A 点开始工作，当执行到 B 点时需要将信息记到磁盘上，且程序 P 在 B 点等待磁盘传输。这时处理器空闲，为了提高系统效率，利用编译程序 P 的“可再入”性，可让编译程序 P 再为源程序乙进行编译，仍从 A 点开始工作。现在应怎样来描述编译程序 P 的状态呢?称它为在 B 点等待磁盘传输状态,还是称它为正在从 A 点开始执行的状态?编译程序 P 只有一个，但加工对象有甲、乙两个源程序，所以再以程序作为占用处理器的单位显然是不适合的了。为此，我们把编译程序 P，与服务对象联系起来，P 为甲服务就说构成进程 P 甲，P 为乙服务则构成进程 P 乙。这两个进程虽共享程序 P，但它们可同时执行且彼此按各自的速度独立执行。现在我们可以说进程 P 甲在 B 点处于等待磁盘传输，而进程 P 乙正在从 A 点开始执行。可见程序与计算(程序的执行)不再一一对应，延用程序概念不能描述这种共享性，因而，引入了新的概念-进程。

2.3.2 进程的状态和转换

2.3.2.1 三态模型

一个进程从创建而产生至撤销而消亡的整个生命周期，可以用一组状态加以刻划，为了便于管理进程，一般来说,按进程在执行过程中的不同状况至少定义三种不同的进程状态：

- **运行态**（running）：占有处理器正在运行。
- **就绪态**（ready）：具备运行条件，等待系统分配处理器以便运行。
- **等待态**（blocked）：不具备运行条件，正在等待某个事件的完成。

一个进程在创建后将处于就绪状态。每个进程在执行过程中，任一时刻当且仅当处于上述三种状态之一。同时，在一个进程执行过程中，它的状态将会发生改变。图 2-12 表示进程的状态转换。

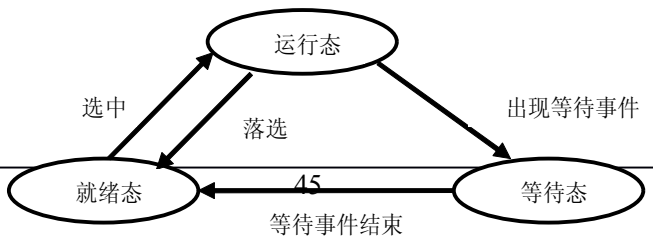


图 2-12　进程三态模型及其状态转换

运行状态的进程将由于出现等待事件而进入等待状态，当等待事件结束之后等待状态的进程将进入就绪状态，而处理器的调度策略又会引起运行状态和就绪状态之间的切换。引起进程状态转换的具体原因如下：

- 运行态—→等待态：等待使用资源；如等待外设传输；等待人工干预。
- 等待态—→就绪态：资源得到满足；如外设传输结束；人工干预完成。
- 运行态—→就绪态：运行时间片到；出现有更高优先权进程。
- 就绪态—→运行态：CPU 空闲时选择一个就绪进程。

2.3.2.2 五态模型

在一个实际的系统里进程的状态及其转换比上节叙述的会复杂一些，例如引入专门的新建态（new）和终止态（exit）。图 2-13 给出了进程五态模型及其转换。

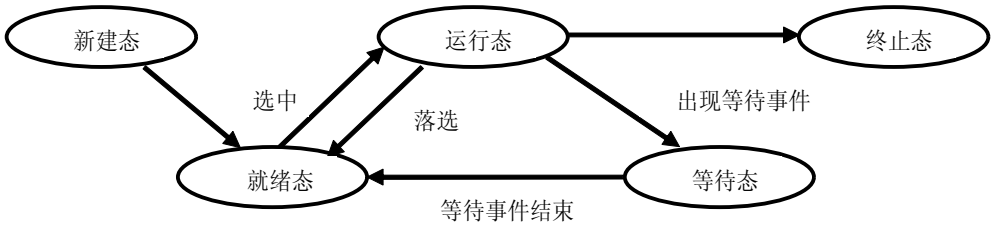


图 2-13　进程五态模型及其状态转换

引入新建态和终止态对于进程管理来说是非常有用的。新建态对应于进程刚刚被创建的状态。创建一个进程要通过两个步骤，首先,是为一个新进程创建必要的管理信息，然后,让该进程进入就绪态。此时进程将处于新建态，它并没有被提交执行，而是在等待操作系统完成创建进程的必要操作。必须指出的是，操作系统有时将根据系统性能或主存容量的限制推迟新建态进程的提交。

类似地，进程的终止也要通过两个步骤，首先,是等待操作系统进行善后，然后,退出主存。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，它将进入终止态。进入终止态的进程以后不再执行，但依然临时保留在操作系统中等待善后。一旦其他进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程。引起进程状态转换的具体原因如下：

- NULL—→新建态：执行一个程序，创建一个子进程。
- 新建态—→就绪态：当操作系统完成了进程创建的必要操作，并且当前系统的性能和虚拟内存的容量均允许。
- 运行态—→终止态：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。
- 终止态—→NULL：完成善后操作。
- 就绪态—→终止态：未在状态转换图中显示，但某些操作系统允许父进程终结子进程。
- 等待态—→终止态：未在状态转换图中显示，但某些操作系统允许父进程终结子进程。

2.3.2.3 进程的挂起

到目前为止，我们或多或少总是假设所有的进程都在内存中。事实上，可能出现这样一些情况，例如由于进程的不断创建，系统的资源已经不能满足进程运行的要求，这个时候就必须把某些进程挂起（suspend），对换到磁盘镜像区中，暂时不参与进程调度，起到平滑系统操作负荷的目的。引起进程挂起的原因是多样的，主要有：

- 系统中的进程均处于等待状态，处理器空闲，此时需要把一些阻塞进程对换出去，以腾出足够的内存装入就绪进程运行。
- 进程竞争资源，导致系统资源不足，负荷过重，此时需要挂起部分进程以调整系统负荷,保证系统的实时性或让系统正常运行。
- 把一些定期执行的进程（如审计程序、监控程序、记账程序）对换出去，以减轻系统负荷。
- 用户要求挂起自己的进程，以便根据中间执行情况和中间结果进行某些调试、检查和改正。
- 父进程要求挂起自己的后代进程，以进行某些检查和改正。

- 操作系统需要挂起某些进程，检查运行中资源使用情况，以改善系统性能;或当系统出现故障或某些功能受到破坏时，需要挂起某些进程以排除故障。

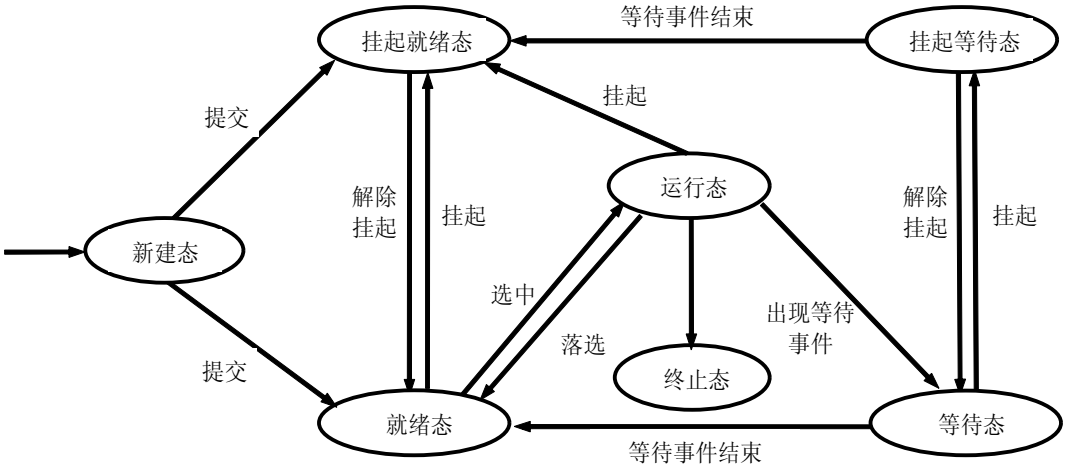


图 2-14　具有挂起功能系统的进程状态及其转换

图 3-3 给出了具有挂起进程功能的系统中的进程状态。在此类系统中，进程增加了两个新状态：挂起就绪态（ready,suspend）和挂起等待态（blocked,suspend）。挂起就绪态表明了进程具备运行条件但目前在二级存储器中，只有当它被对换到主存才能被调度执行。挂起等待态则表明了进程正在等待某一个事件且在二级存储器中。

引起进程状态转换的具体原因如下：

- 等待态—→挂起等待态：如果当前不存在就绪进程，那么至少有一个等待态进程将被对换出去成为挂起等待态；操作系统根据当前资源状况和性能要求，可以决定把等待态进程对换出 去成为挂起等待态。
- 挂起等待态—→挂起就绪态：引起进程等待的事件发生之后，相应的挂起等待态进程将转换为挂起就绪态。
- 挂起就绪态—→就绪态：当内存中没有就绪态进程，或者挂起就绪态进程具有比就绪态进程更高的优先级，系统将把挂起就绪态进程转换成就绪态。
- 就绪态—→挂起就绪态：操作系统根据当前资源状况和性能要求，也可以决定把就绪态进程对换出去成为挂起就绪态。
- 挂起等待态—→等待态：当一个进程等待一个事件时，原则上不需要把它调入内存。但是在下面一种情况下，这一状态变化是可能的。当一个进程退出后，主存已经有了一大块自由空间，而某个挂起等待态进程具有较高的优先级并且操作系统已经得知导致它阻塞的事件即将结束，此时便发生了这一状态变化。
- 运行态—→挂起就绪态：当一个具有较高优先级的挂起等待态进程的等待事件结束后，它需要抢占 CPU，，而此时主存空间不够，从而可能导致正在运行的进程转化为挂起就绪态。另外处于运行态的进程也可以自己挂起自己。
- 新建态—→挂起就绪态：考虑到系统当前资源状况和性能要求，可以决定新建的进程将被对换出去成为挂起就绪态。

不难看出，可以把一个挂起进程等同于不在主存的进程，因此挂起的进程将不参与进程调度直到它们被对换进主存。一个挂起进程具有如下特征：

- 该进程不能立即被执行。
- 挂起进程可能会等待一个事件，但所等待的事件是独立于挂起条件的，事件结束并不能导致进程具备执行条件。
- 进程进入挂起状态是由于操作系统、父进程或进程本身阻止它的运行。
- 结束进程挂起状态的命令只能通过操作系统或父进程发出。

2.3.3 进程的描述

2.3.3.1 操作系统的控制结构

操作系统作为资源管理和分配程序，其本质任务是自动控制程序的执行，并满足进程执行过程中提出的资源使用要求。因此操作系统的核心控制结构是进程结构，资源管理的数据结构将围绕进程结构展开。

本节在研究进程的控制结构之前，首先介绍一下操作系统的控制结构。为了有效的管理进程和资源，

操作系统必须掌握每一个进程和资源的当前状态。从效率出发，操作系统的控制结构及其管理方式必须是简明有效的，通常是通过构造一组表来管理和维护进程和每一类资源的信息。操作系统的控制表分为四类，进程控制表，存储控制表，I/O 控制表和文件控制表。

- 进程控制表用来管理进程及其相关信息。
- 存储控制表用来管理一级（主）存储器和二级（虚拟）存储器，主要内容包括：主存储器的分配信息，二级存储器的分配信息，存储保护和分区共享信息，虚拟存储器管理信息。
- I/O 控制表用来管理计算机系统的 I/O 设备和通道，主要内容包括：I/O 设备和通道是否可用，I/O 设备和通道的分配信息，I/O 操作的状态和进展，I/O 操作传输数据所在的主存区。
- 文件控制表用来管理文件，主要内容包括：被打开文件的信息，文件在主存储器和二级存储器中的位置信息，被打开文件的状态和其他属性信息。

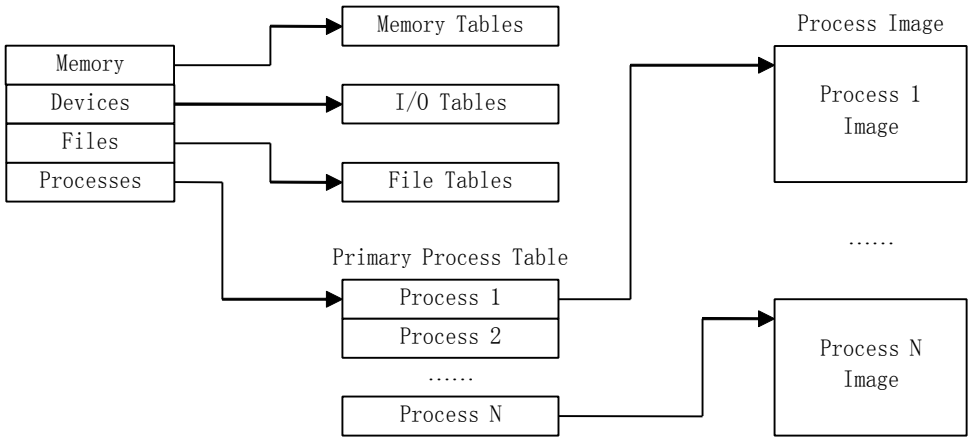


图 2-15　操作系统控制表的通用结构

图 2-15 给出了操作系统控制表的通用结构，虽然具体操作系统的实现各有特色，但其控制表的基本结构是类似的。值得指出的是，图 3-4 仅仅是操作系统控制结构的示意图，在实际的操作系统中，其具体形式要复杂的多，并且这四类控制表也是交叉引用的。

2.3.3.2 进程映像

当一个程序进入计算机的主存储器进行计算就构成了进程，主存储器中的进程到底是如何组成的？操作系统中把进程物理实体和支持进程运行的环境合称为进程上下文（context）。当系统调度新进程占有处理器时，新老进程随之发生上下文切换。因此，进程的运行被认为是在上下文中执行。在操作系统中，进程上下文包括三个组成部分：

- 用户级上下文：由用户程序块、用户数据块（含共享数据块）和用户堆栈组成的进程地址空间。
- 系统级上下文：包括进程的标识信息、现场信息和控制信息，进程环境块，以及系统堆栈等组成的进程地址空间。
- 寄存器上下文：由程序状态字寄存器和各类控制寄存器、地址寄存器、通用寄存器组成。

由于一个进程让出处理器时，其寄存器上下文将被保存到系统级上下文的相应的现场信息位置，因此进程的内存映像可以很好地说明进程的组成。简单的说，一个进程映像（Process Image）包括：

- 进程程序块，即被执行的程序，规定了进程一次运行应完成的功能。通常它是纯代码，作为一种系统资源可被多个进程共享。
- 进程数据块，即程序运行时加工处理对象，包括全局变量、局部变量和常量等的存放区以及开辟的工作区，常常为一个进程专用。
- 系统/用户堆栈，每一个进程都将捆绑一个系统/用户堆栈。用来解决过程调用或系统调用时的地址存储和参数传递。
- 进程控制块，每一个进程都将捆绑一个进程控制块，用来存储进程的标志信息、现场信息和控制信息。进程创建时，建立一个 PCB；进程撤销时，回收 PCB，它与进程一一对应。

可见每个进程有四个要素组成：控制块、程序块、数据块和堆栈。例如，用户进程在虚拟内存中的组织如图 2-16 所示：

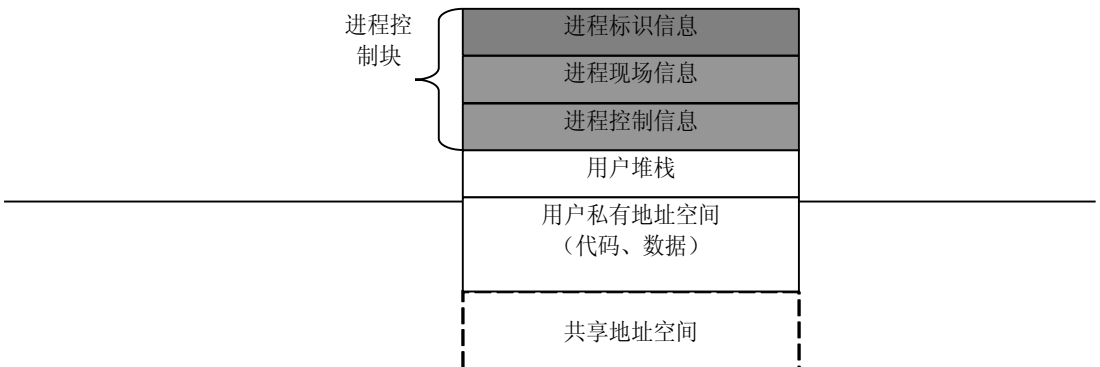


图 2-16　用户进程在虚拟内存中的组织

2.3.3.3 进程控制块

每一个进程都有一个也只有一个进程控制块(Process Control Block) ，是操作系统用于记录和刻划进程状态及有关信息的数据结构。也是操作系统掌握进程的唯一资料结构，它包括了进程执行时的情况，以及进程让出处理器后所处的状态、断点等信息。一般说，进程控制块包含三类信息：

- 标识信息。用于唯一地标识一个进程，常常分由用户使用的外部标识符和被系统使用的内部标识号。几乎所有操作系统中进程都被赋予一个唯一的、内部使用的数值型的进程号，操作系统的其他控制表可以通过进程号来交叉引用进程控制表。常用的标识信息包括进程标识符、父进程的标识符、用户进程名、用户组名等。
- 现场信息。用于保留一个进程在运行时存放在处理器现场中的各种信息，任何一个进程在让出处理器时必须把此时的处理器现场信息保存到进程控制块中，而当该进程重新恢复运行时 也应恢复处理器现场。常用的现场信息包括通用寄存器的内容、控制寄存器(如 PSW 寄存器)的内容、用户堆栈指针、系统堆栈指针等。
- 控制信息。用于管理和调度一个进程。常用的控制信息包括：1）进程的调度相关信息，如进程状态、等待事件和等待原因、进程优先级、队列指引元等；2) 进程组成信息,如正文段指针、数据段指针;3）进程间通信相关信息，如消息队列指针、信号量等互斥和同步机制；4）进程在二级存储器内的地址；5）CPU 资源的占用和使用信息，如时向片余量、进程已占用 CPU 的时间、进程已执行的时间总和,记帐信息;6）进程特权信息，如在内存访问和处理器状态方面的特权。7) 资源清单，包括进程所需全部资源、已经分得的资源,如主存资源、I/O 设备、打开文件表等。

进程控制块是操作系统中最为重要的数据结构，每个进程控制块包含了操作系统管理所需的所有进程信息，进程控制块的集合事实上定义了一个操作系统的当前状态。进程控制块使用或修改权仅属于操作系统程序，包括调度程序、资源分配程序、中断处理程序、性能监视和分析程序等。当系统创建一个进程时，就为它建立一个 PCB，而进程执行结束便回收它占用的 PCB,有了进程控制块进程才能被调度执行,操作系统是根据 PCB 来对并发执行的进程行控制和管理的。因而,进程控制块可看作是一个虚拟 CPU。

2.3.3.4 进程管理

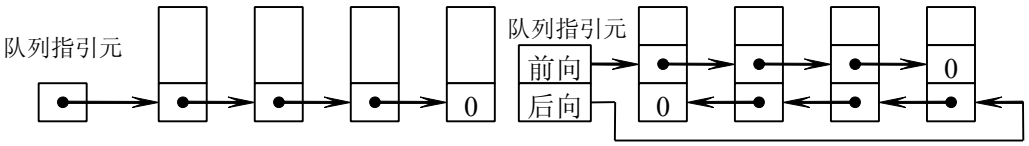
并发系统中同时存在许多进程，有的处于就绪态,有的处于等待态，等待原因各种各样。进程的特征主要是的 PCB 来刻划的,为了便于系统的管理和调度,常常把各个进程的 PCB 用某种方法组织起来。用得较多的是用队列来组织 PCB,下面先介绍这种方法。

一般说来，把处于同一状态(例如就绪态)的所有进程控制块链接在一起,这样的数据结构称为进程队列(Process Queues),简称队列。同一状态进程的 PCB 既可按先来先到的原则排成队列;也可以按优先数或其它原则排成多个队列。例如,对于等待态的进程队列可以进一步细分，每一个进程按等待的原因进入相应的等待队列。例如，如果一个进程要求使用某个设备，而该设备已经被占用时，此进程就链接到与该设备相关的等待态队列中去。

在一个队列中，链接进程控制块的方法可以是多样的，常用的是单向链接和双向链接。单向链接方法是在每个进程控制块内设置一个队列指引元，它指出在队列中跟随着它的下一个进程的进程控制块内队列指引元的位置。双向链接方法是在每个进程控制块内设置两个指引元，其中一个指出队列中该进程的上一个进程的进程控制块内队列指引元的位置，另一个指出队列中该进程的下一个进程的进程控制块的队列指引元的位置。为了标志和识别一个队列,系统为每一个队列设置一个队列标志，单向链接时，队列标志指引元指向队列中第一个进程的队列指引元的位置;双向链接时，队列标志的前向指引元指向队列中第一个进程的前向队列指引元的位置；队列标志的后向指引元指向队列中最后一个进程的后向队列指引元的位置。这两种链接方式如图 2-17(a)和(b)所示。

图 2-17　进程控制块的链接

当发生的某个事件使一个进程的状态发生变化时，这个进程就要退出所在的某个队列而排入到另一个队列中去。一个进程从一个所在的队列中退出的工作称为出队，相反，一个进程排入到一个指定的队



（a） 单向连接　　（b） 双向连接

列中的工作称为入队。处理器调度中负责入队和出队工作的功能模块称为队列管理模块，简称队列管理。图 2-18 给出了操作系统的队列管理和状态转换示意图。

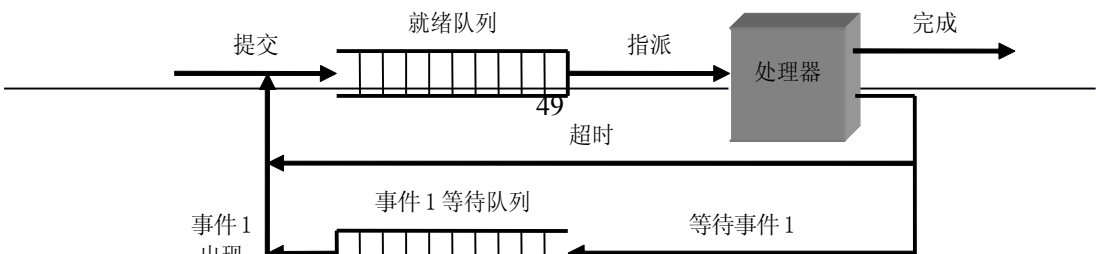


图 2-18　操作系统的队列管理和状态转换示意图

操作系统中的进程队列可能是标准队列，也可以是优先队列。下面来考虑队列管理是如何将一个进程从某个队列中移出而加入到另一个队列中去。假设采用双向接，如图 3-6(b)所示，每个进程有两个队列指引元，用来指示该进程在队列中的位置，其中第一个队列指引元为前向指引元，第二个为后向指引元。根据一个进程排在一个队列中的情况，前(后)向指引元的内容规定如下：

- 情况 1：它是队列之首。此时，它的前向指引元为 0，而后向指引元指出它的下一个进程的后向指引元位置。
- 情况 2：它是队列之尾。此时，它的后向指引元为 0，而它的前向指引元指出它的上一个进程的前向指引元位置。
- 情况 3：它的前后均有进程。此时，前(后)向指引元指出它的上(下)一个进程的前(后)向指引元位置。

我们首先考虑一个进程的出队，假设进程 Q 在某个队列中，它的前面是进程 P，后面是进程 R。进程 Q 出队过程为：把 Q 的前向指引元的内容送到 R 的前向指引元中，把 Q 的后向指引元的内容送到 P 的后向指引元中。于是 P 的后向指引元指向 R，而 R 的前向指引元指向 P，Q 就从队列中退出。类似的可实现队首进程，队尾进程的出队，或者让一个进程加入到队列中去。

此外,用来组织 PCB 的方法为表格法。把所有进程的 PCB 都组织在一个线性表中,进程调度时需要查找整个 PCB 表;也可以把相同状态进程的 PCB 组织在一个线性表中,系统有多个线性表,这样可缩短查表时间。

2.3.4 进程的控制

2.3.4.1 进程的创建

每一个进程都有生命期，即从创建到消亡的时间周期。当操作系统为一个程序构造一个进程控制块并分配地址空间之后，就创建了一个进程。进程的创建来源于以下四个事件：

- 提交一个批处理作业。
- 在终端上交互式的登录。
- 操作系统创建一个服务进程。
- 存在的进程孵化（spawn）新的进程。

下面来讨论一下孵化操作，当一个用户作业被接受进入系统后，可能要创建一个或多个进程来完成这个作业；一个进程在请求某种服务时，也可能要创建一个或多个进程来为之服务。例如，当一个进程要求读卡片上的一段数据时，可能创建一个卡片输入机管理进程。有的系统把“孵化”用父子进程关系来表示，当一个进程创建另一个进程时，生成进程称父进程(Parent Process) ，被生成进程称子进程(Child Process) 、即一个父进程可以创建子进程，从而形成树形的结构，如 Unix 就是这样，当父进程创建了子进程后，子进程就继承了父进程的全部资源，父子进程常常要相互通信和协作，当子进程结束时，又必须要求父进程对其作某些善后处理。

进程的创建过程如下描述：

- 在主进程表中增加一项，并从 PCB 池中取一个空白 PCB。
- 为新进程的进程映像中的所有成分分配地址空间。对于进程孵化操作还需要传递环境变量，构造共享地址空间。
- 为新进程分配资源，除内存空间外，还有其它各种资源。

- 查找辅存，找到进程正文段并装到正文区。
- 初始化进程控制块，为新进程分配一个唯一的进程标识符，初始化 PSW。
- 加入某一就绪进程队列,或直接将进程投入运行。
- 通知操作系统的某些模块，如记账程序、性能监控程序。

2.3.4.2 进程上下文切换

中断是激活操作系统的唯一方法，它暂时中止当前运行进程的执行，把处理器切换到操作系统的控制之下。而当操作系统获得了处理器的控制权之后，它就可以实现进程的切换。顾名思义，进程的切换就是让处于运行态的进程中断运行，让出处理器，这时要做一次进程上下文切换、即保存老进程状态而装入被保护了的新进程的状态，以便新进程运行。进程切换的步骤如下：

- 保存被中断进程的处理器现场信息。
- 修改被中断进程的进程控制块的有关信息，如进程状态等。
- 把被中断进程的进程控制块加入有关队列。
- 选择下一个占有处理器运行的进程。
- 修改被选中进程的进程控制块的有关信息。
- 根据被选中进程设置操作系统用到的地址转换和存储保护信息。
- 根一据被选中进程恢复处理器现场。

从上面介绍的切换工作可以看出，当进行上下文切换时，内核保留足够信息，为的是能切换回原进程，并恢复它执行;类似地，当从用户态转到核心态时，内核保留足够信息以便后来能返回到用户态，并让进程从它的断点继续执行。用户态到核心态或者核心态到用户态的转变是 CPU 模式的改变，而不是进程上下文切换。为了进一步说明进程的上下文切换，我们来讨论一下模式切换。当中断发生的时候，暂时中断正在执行的用户进程，把进程从用户状态切换到内核状态，去执行操作系统例行程序以获得服务，这就是一次模式切换，注意，此时仍在该进程的上下文中执行，仅仅模式变了。内核在被中断了的进程的上下文中对这个中断事件作处理，即使该中断可能不是此进程引起的。另一点要注意的是被中断的进程可以是正在用户态下执行的，也可以是正在核心态下执行的，内核都要保留足够信息以便在后来能恢复被中断了的进程执行。内核在核心态下对中断事件进行处理,决不会再产生或调度一个特殊进程来处理中断事件。模式切换的步骤如下：

- 保存被中断进程的处理器现场信息。
- 根据中断号置程序计数器。
- 把用户状态切换到内核状态，以便执行中断处理程序。

显然模式切换不同于进程切换，它并不引起进程状态的变化，在大多数操作系统中，它也不一定引起进程的切换，在完成了中断调用之后，完全可以再通过一次逆向的模式切换来继续执行用户进程。Unix 中存在两类进程：系统进程和用户进程，系统进程在核心态下执行操作系统代码，用户进程在用户态下执行用户程序。用户进程因中断和系统调用进入内核态，系统进程开始执行，这两个进程(用户进程和系统进程)使用同一个 PCB，所以实质上是一个进程。但是这两个进程所执行的程序不同，映射到不同物理地址空间、使用不同堆栈。一个系统进程的地址空间中包含所有的系统核心程序和各进程的进程数据区，所以，各进程的系统进程除数据区不同外，其余部分全相同，但各进程的用户进程部分则各不相同。图 2- 19 是 Unix 中进程上下文切换和模式切换示意。其中，1 为进程在用户态下执行，2 为进程在核心态下执行，3 进程为就绪态,4 进程为等待态。任何时刻一个处理器仅能执行一个进程，所以至多有一个进程可以处在状态 1 或状态 2，这两个状态相应于用户态和核心态。在多道程序设计系统中，有许多进程在并发执行，如果两个以上进程同时执行系统调用,并要求在核心态下执行，则有可能破坏核心数据结构中的信息。通过禁止任意的上下文切换和控制中断的响应,就能保证数据的一致性。图中示意，仅当进程从”核心态运行”状态转为”在内存中等待”状态时,内核才允许上下文切换。在核心态下的进程不能被其他进程抢占，或者说进程在核心态下执行时不允许上下文切换。由于内核处于不可抢占态，所以内核可保持其数据结构的完整和一致。

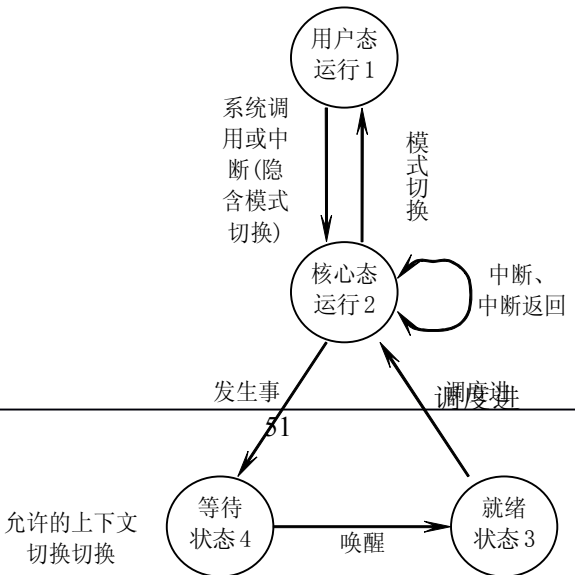


图 2-19 进程上下文切换和模式切换

2.3.4.3 进程的阻塞和唤醒

当一个等待事件结束之后会产生一个中断，从而激活操作系统，在操作系统的控制之下将被阻塞的进程唤醒，如 I/O 操作结束、某个资源可用或期待事件出现。进程唤醒的步骤如下：

- 从相应的等待进程队列中取出进程控制块。
- 修改进程控制块的有关信息，如进程状态等。
- 把修改后进程控制块加入有关就绪进程队列。

2.3.4.4 进程的撤销

一个进程完成了特定的工作或出现了严重的异常后，操作系统则收回它占有的地址空间和进程控制块，此时就说撤销了一个进程。进程撤销的主要原因包括：

- 进程正常运行结束。
- 进程执行了非法指令。
- 进程在常态下执行了特权指令。
- 进程运行时间超越了分配给它的最大时间段。
- 进程等待时间超越了所设定的最大等待时间。
- 进程申请的内存超过了系统所能提供最大量。
- 越界错误。
- 对共享内存区的非法使用。
- 算术错误，如除零和操作数溢出。
- 严重的输入输出错误。
- 操作员或操作系统干预。
- 父进程撤销其子进程。
- 父进程撤销。
- 操作系统终止。

一旦发生了上述事件后，系统将调用撤销原语终止进程，具体步骤如下：

- 根据撤销进程标识号，从相应队列中找到它的 PCB；
- 将该进程拥有的资源归还给父进程或操作系统；
- 若该进程拥有子进程，应先撤销它的所有子孙进程，以防它们脱离控制；
- 撤销进程出队，将它的 PCB 归还到 PCB 池。

2.3.5 进程管理的实现

2.3.5.1 进程管理的实现

操作系统是一种系统软件，作为其核心的组成部分，进程管理也毫不例外是一个程序集合，通常在实现进程管理时包括以下一些程序模块：

- 队列管理模块：负责进程队列的入队和出队工作。
- 进程调度程序：负责选择下一个占有处理器运行的进程。
- 资源分配程序：负责分配资源给进程。
- 上下文切换程序：负责进程切换和模式切换。
- 进程控制原语：提供了若干基本操作以管理和控制进程。如：建立进程原语、撤销进程原语、阻塞进程原语、唤醒进程原语、挂起进程原语、解除挂起进程原语、改变进程优先数原语、修改进程状态原语等等。
- 进程通信原语：提供了若干基本操作以便进程间通信。如：P 操作原语、V 操作原语、管程原语、send 原语、receive 原语等等。
- 其他。

显然，这些程序是操作系统的核心部分，它们的效率直接关系到整个操作系统的运行效率。

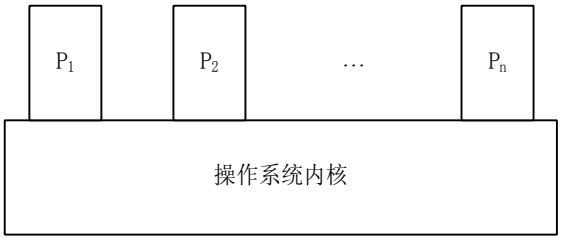
2.3.5.2 进程管理的实现模型

本节讨论进程管理的几种主要实现模型。

1）非进程内核模型

许多老式操作系统的实现采用非进程内核模型，亦即操作系统的功能都不组织成进程来实现。如图 2-20 所示，该模型包括一个较大的操作系统内核程序，进程的执行在内核之外。当中断发生时，当前运 行进程的上下文现场信息将被保存，并把控制权传递给操作系统内核。操作系统具有自己的内存区和系统堆栈区，它将在核心态执行相应的操作，并根据中断的类型和具体的情况，或者是恢复被中断进程的现场并让它继续执行，或是转向进程调度指派另一个就绪进程运行。

图 2-20　非进程内核模型



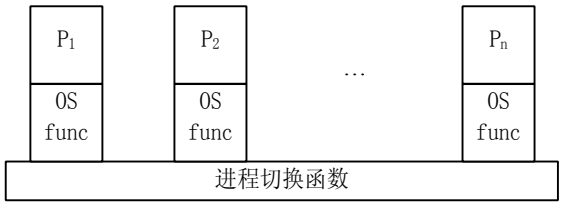
在这种情况下，进程的概念仅仅是针对用户程序而言的，操作系统代码作为一个分离实体在内核模式下运行。

2）OS 功能在用户进程内执行的实现模型

小型机和微型机操作系统(如 Unix 等)往往采用 OS 功能在用户进程内执行的实现模型，如图 2-21 所示，在这种实现模型中，大部分操作系统功能组织成一组例行程序供用户程序调用，认为操作系统例程与用户进程是上不文相关的，操作系统的地址空间被包含在用户进程的地址空间中，因而，操作系统例行程序也在用户进程的上下文环境中执行。

图 2-22 给出了 OS 功能在用户进程内执行的实现模型中的进程映像，它既包含进程控制块、用户堆栈、容纳用户程序和数据的地址空间等，还包括操作系统内核的程序、数据和堆栈。

当发生一个中断后，处理器状态将被置成内核状态，控制被传递给操作系统例行程序。此时发生了模式切换，模式上下文（现场）信息被保存，但是进程上下文切换并没有发生，仍在该用户进程中执行，提供单独的内核堆栈用于管理进程在核心态下执行时的调用和返回，操作系统例行程序和数据放在共享



地址空间，且被所有用户进程共享。

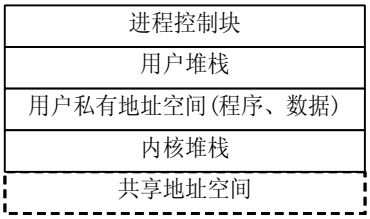


图 2-21　OS 功能在用户进程内执行的实现模型

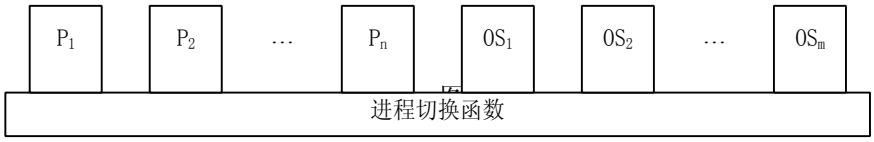
图 2-22　OS 在用户进程内执行的实现模型的进程映像

当操作系统程序完成了工作之后，如果应该让当前进程继续运行的话，就可以做一次模式切换来恢复执行原先被中断的用户进程。这种技术提供了不必要通过进程上下文切换就可以中断用户进程来调用操作系统例行程序的手段。

如果应该发生进程切换的话，控制就被传递给操作系统的进程切换例行程序，由它来实现进程切换操作，把当前进程的状态置为非运行状态，而指派另一个就绪进程来占有处理器运行。值得指出的是，一些系统中进程切换例行程序是在当前进程中执行的，而另一些系统则不是，在图 3-10（a）中，我们把它在逻辑上分离出来。

3）OS 功能由进程实现的模型

OS 功能由进程实现的模型把操作系统组织成一组进程、即操作系统功能是这些系统进程集合运行 的结果，这些系统进程也称服务器或服务器进程，于是与用户进程构成了 Client/Server 关系，Window NT 采用了这种结构。如图 2-23 所示,除了极少部分功能在内核模式下运行，大部分操作系统功能被组织在一组分离的进程内实现，这组进程在用户模式下运行，而进程切换例行程序的执行仍然在进程之外。



2-23　OS 功能由进程实现模型

这一实现模型有很多优点。首先，它采用了模块化的操作系统实现方法，模块之间具有最小化的简洁的接口。其次，大多数操作系统功能被组织成分离的进程，有利于操作系统的实现、配置和扩充，例如，性能监视程序用来记录各种资源的利用率;系统中用户进程推进比率，因为，这些程序并不提供给进程特别的服务，仅仅被系统调用，把它设计成一个服务器进程，便可赋予一定的优先级，夹在其他进程中运行。最后,这一结构在多处理器和多计算机的环境下非常有效，有利于系统性能的改进。

2.3.6 实例研究——Unix SVR4 的进程管理

Unix SVR4 的进程管理的实现采用基于用户进程的实现模型，大多数操作系统功能在用户进程的环境中执行，因此它需要在用户模式和内核模式切换。Unix SVR4 允许两类进程：用户进程和系统进程。系统进程在内核模式下执行，完成操作系统的一些重要功能，如内存分配和进程对换。而用户进程在用户模式下执行用户程序，在内核模式下执行操作系统的代码，系统调用、中断和异常将引起模式切换。

2.3.6.1 Unix SVR4的进程状态

图 2-24 给出了 Unix SVR4 进程状态及其转换，其中：包括两种运行状态，分别为核心运行态和用户运行态 Kernel running；两种等待状态，分别对应了在内存或被换出内存；三种就绪状态，Preempted、Ready to run, in memory 和 Ready to run, swapped，后者被换出内存，再换入内存前不能被调度执行。Preempted 和 Ready to run, in memory 本质上是同一种状态，也被组织在同一个就绪进程队列中，因此在图 3-10 中用虚线连接在一起。但是 Preempted 态和 Ready to run, in memory 态也是由区别的，当一个 Kernel running 态进程完成了相应的操作准备切换到 User running 态时，出现了更高优先级的就绪进程，那么原来那个进程将转化为 Preempted 态。

具体的进程状态包括：

- User running：在用户模式下运行。
- Kernel running：在内核模式下运行。
- Preempted：当一个进程从内核模式返回用户模式时，发生了进程切换后处于的就绪状态。
- Ready to run, in memory：就绪状态，在内存。
- Asleep in memory：等待状态，在内存。
- Ready to run, swapped：就绪状态，被对换出内存，不能被调度执行。
- Sleeping, swapped：睡眠状态，被对换出内存。
- Created：新建状态。
- Zombie：终止状态：进程已不存在，留下状态码和有关信息使父进程收集。

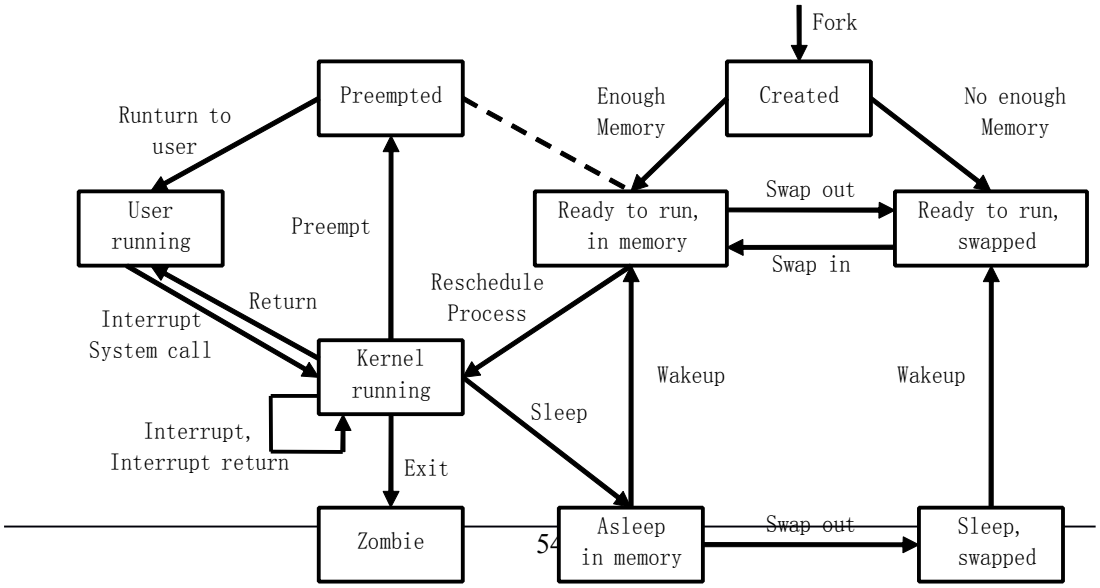


图 2-24　Unix SVR4 进程状态及其转换

Unix 操作系统中有两个固定的进程，0 号进程是 swap 进程，在系统自举时被创建；1 号进程是 init 进程，由 0 号进程孵化而创建。系统中的其他进程都是 1 号进程的子进程，当一个交互式用户登录到系统中时，1 号进程为这个用户创建一个用户进程，用户进程在执行具体应用是进一步的创建子进程，从而构成一棵进程树。

2.3.6.2 Unix SVR4的进程描述

UNIX 的进程组成如图 2-25 所示，它由三部分组成：proc 结构、数据段和正文段,它们合称为进程映像，Unix 中把进程定义为映像的执行。其中，PCB 由基本控制块 proc 结构和扩充控制块 user 结构两部分组成。在 proc 结构里存放着关于一个进程的最基本、最必需的信息，因此它常驻内存；在 user 结构里存放着只有进程运行时才用到的数据和状态信息，为了节省内存空间，当进程暂时不在处理机上运行时，就把它放在磁盘上的对换区中，进程的 user 结构总和进程的数据段一起，在主存和磁盘对换区之间换进/换出。

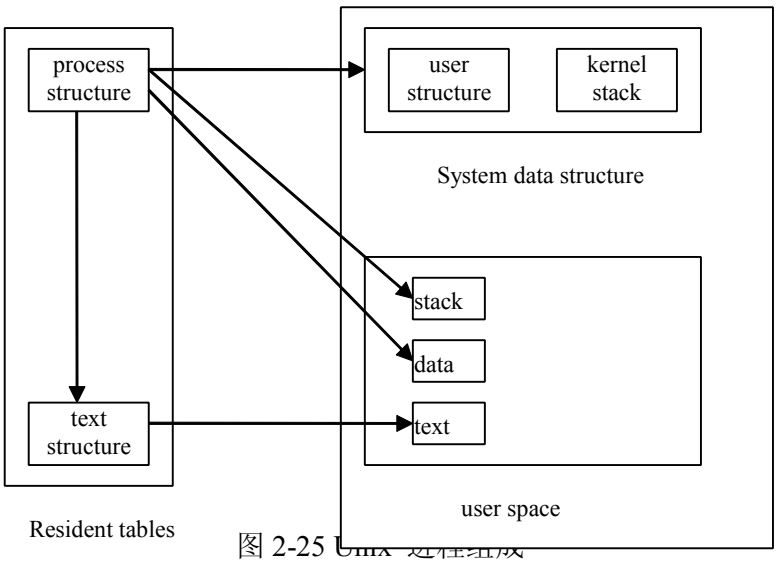


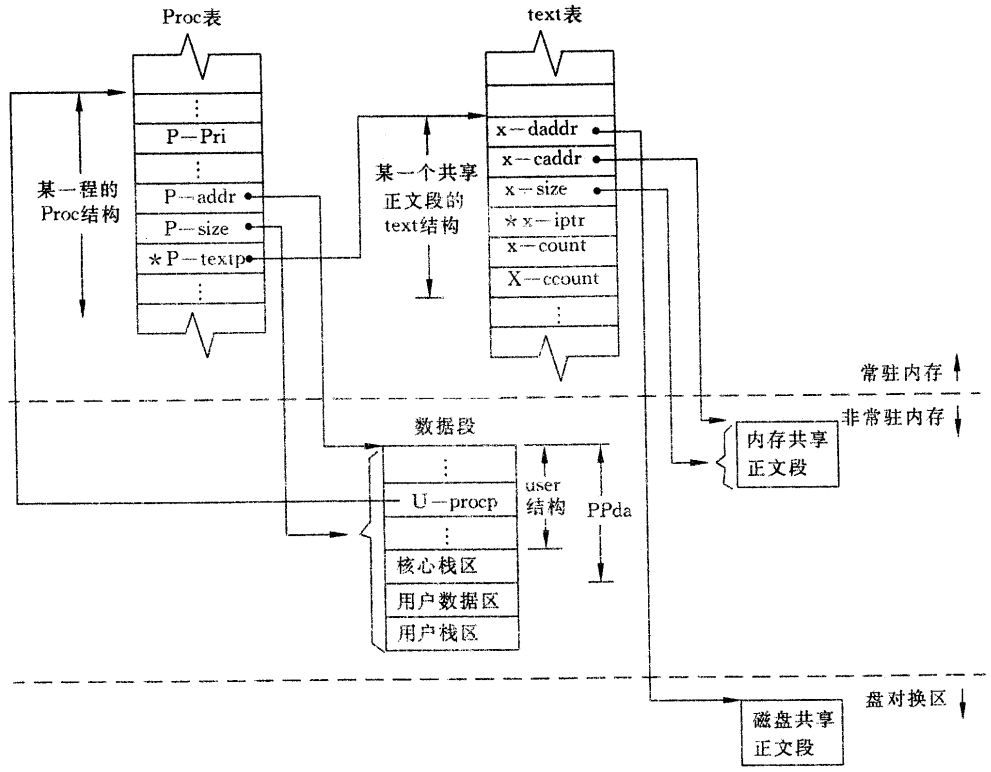
图 2-25 Unix 进程组成

系统中维持一[illegible]表目，Swappable process image 构，供一个进程使用，因此,UNIX 中最多同时可存在 50 个进程。创建进程时，在 proc 表中找一个空表目，以建立起相应于该进程的 proc 结构。

- proc 结构　包括进程标识符、父进程标识符、进程用户标识符、进程状态、等待的事件、调度优先数、进程的大小、指向 user 结构和进程存储区(text/data/stack)的指针、有关进程执行时间/核心资源使用/用户设置示警信号等的计时器、就绪队列指针等。
- user 结构　包括现场保护、内存管理、系统调用、文件管理、文件读写、时间信息、映象位置、用户标识、用户组标识、用户打开文件表、各种标志等。
- 系统数据结构　进程系统数据区，通常称作 ppda，它位于数据段的前面，进程 proc 结构中的 P-addr 指向这个区域的首址。该区共有 1024 个字节，由两块内容组成：最前面的 289 个字节为进程的扩充控制块 user 结构，剩下的 734 个字节为核心栈，当进程运行在核心态时，这里是它的工作区，用来保存过程调用和中断访问时用到的地址和参数。
- 用户数据区　通常存放程序运行时用到的数据，如果进程运行的程序是非共享的，那么这个程序也放于此地。
- 用户栈区　当进程运行在用户态时，这里是它的工作区。
- text 结构　正文段在磁盘上和主存中的位置和大小、访问正文段进程数、在主存中访问正文段进程数、标志信息、地址转换信息。
- 由于共享正文段在进程映象中的特殊性，为了便于对它们的管理，UNIX 系统在内存中设置了一张正文段表。该表共有 40 个表目，每一个表目都是一个 text 结构，用来记录一个共享正文段的属性（磁盘和主存中的位置、尺寸、共享的进程数等、正文段文件节点指针），有时也把这种结构称为正文段控制（信息）块。
- 这是可以被多个进程共享的可重入程序和常数，如果一个进程的程序是不被共享的，那么它的映象中就不出现这一部分。若一个进程有共享正文段，那么当把该进程的非常驻内存部分调入内存时，应该关注共享正文段是否也在内存，如果发现不在内存，则要将它调入；当把该进程的非常驻内存部分调出内存时，同样要关注它的共享正文段目前被共享的情况，只要还有一个别的共享进程的映象全部在内存，那么这个共享正文段就不得调出去。如果一个进程有共享

正文段，那么该共享正文段在正文段表里一定有一个 text 结构与之相对应，而在该进程的基本控制块 proc 里，p-textp 就指向这一个 text 结构。综上所述，在 UNIX 进程映象的三个组成部分中，proc、user 和 text 这三个数据结构是最为重要的角色，图 2-26 反映了这三者之间的基本关系。

- 进程区域表　系统为每个进程建立一张进程区域表 PPRT（Per Process Region Table）由存储管理系统使用，它定义了物理地址与虚拟地址之间的对应关系，还定义了进程对存储区域的访问权限。其中含有正文段、数据段和堆栈的区域表的指针和各区域的逻辑起始地址;区域表中含有该区域属性(正文/数据，可否共享)的信息和页表的指针;而每个页表中含有相应区域的页面在内存的起始地址。



2. 面的操作：

- 为子进程分配一个进程表。
- 为子进程分配一个进程标识符。
- 复制父进程的进程映像，但不复制共享内存区。
- 增加父进程所打开文件的计数，表示新进程也在使用这些文件。
- 把子进程置为 Ready to Run 状态。
- 返回子进程的标识符给父进程，把 0 值返回给子进程。

以上的操作都是父进程在内核模式下完成的，然后父进程还应执行以下的操作之一，以完成进程的指派：

- 继续呆在父进程中。此时把进程控制切换到父进程的用户模式，在 fork()点继续向下运行，而子进程进入 Ready to Run 状态。
- 把进程控制传递到子进程，子进程在 fork()点继续向下运行，而父进程进入 Ready to Run 状态。
- 把进程控制传递到其他进程，父进程和子进程进入 Ready to Run 状态。

用 fork( )创建子进程之后，父子进程均执行相同的代码段，即子进程的程序是父进程程序的复制品，父子进程的指令执行点均在 fork( )之后的那个语句。那么，如何来区别父子进程呢?事实上，执行 fork()调用后，子进程 pid 的返回值是 0，而父进程 pid 的返回值为非 0 正整数(即子进程唯一内部标识号)，这样就可以通过程序控制流判别 pid 的值,在父子进程中各自执行需要的操作。

2.3.7 实例研究——Linux 的进程管理

Linux 是一个多用户的操作系统，支持多道程序设计、分时处理和软实时处理，它的 module 机制也带有某些微内核的特征。

2.3.7.1 进程和进程状态

Linux 的进程概念与传统操作系统中的进程概念完全一致，它目前不支持线程概念，进程是操作系统调度的最小单位。

如图 2-27 所示，Linux 的进程状态共有 6 种：

- TASK_RUNNING：正在运行或准备运行的进程。
- TASK_INTERRUPTIBLE：处于等待队列中的进程，一旦资源可用时被唤醒，也可以由其他进程通过信号（SIGNAL）或定时中断唤醒。
- TASK_UNINTERRUPTIBLE：处于等待队列中的进程，一旦资源可用时被唤醒，也不可以由其他进程通过信号（SIGNAL）或定时中断唤醒。
- TASK_ZOMBIE：进程运行结束但是尚未消亡时处于的状态。
- TASK_STOPPED：进程被暂停，正在等待其他进程发出的唤醒信号。
- TASK_SWAPPING：页面被交换出内存的进程。

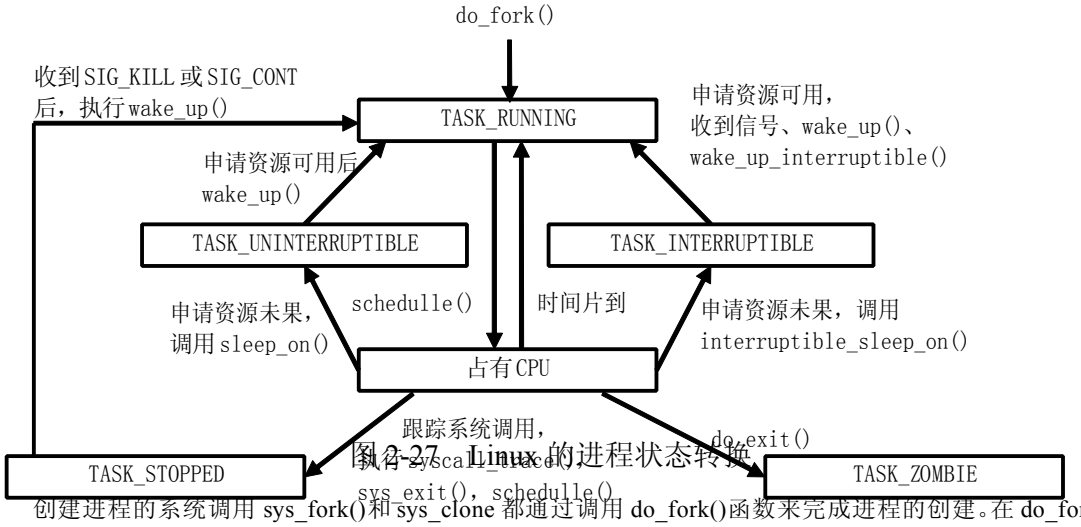


图 2-27 Linux 的进程状态转换

创建进程的系统调用 sys_fork()和 sys_clone 都通过调用 do_fork()函数来完成进程的创建。在 do_fork()函数中，它首先分配进程控制块 task_struct 的内存和进程所需的堆栈，并监测系统是否可以增加新的进程；然后拷贝当前进程的内容，并对一些数据成员进行初始化；再为进程的运行做准备；最后返回生成的新进程的进程标识号（pid）。如果进程是根据 sys_clone()产生的，那么它的进程标识号就是当前进程的进程标识号，并且对于进程控制块中的一些成员指针并不进行复制，而仅仅把这些成员指针的计数 count 增加 1。这样，父子进程可以有效地共享资源。

进程终止的系统调用 sys_exit()通过调用 do_exit()函数实现。函数 do_exit()首先释放进程占用的大部分资源，然后进入 TASK_ZOMBIE 状态，调用 exit_notify()通知父子进程，调用 schedule()重新调度。

2.3.7.2 进程控制块

Linux 的进程控制块由结构 struct task_struct 描述，包括在/include/linux/sched.h 中。它的构成简述如下：

调度用数据成员	state	进程状态
	flags	进程状态标记
	priority	进程优先数
	rt_priority	实时进程优先数
	counter	时间片
	policy	调度策略（0 基于优先权的时间片轮转、1 基于先进先出的实时调度，2 基于优先数的实时调度）
信号处理	signal	记录进程接收到的信号，共 32 位，每位对应一种信号
	blocked	进程接收到信号的屏蔽位
	sig	信号对应的处理函数
进程队	next_task prev_task	双向链接指针

列指针	next_run prev_run	就绪队列双向链接指针
	p_opptr，p_pptr p_cptr p_ysptr，p_osptr	分别指向原始父进程、父进程，子进程、新老兄弟进程
进程标识	uid，gid	用户标识和组标识
	group	允许进程同时拥有的一组用户组号
	euid，egid	有效的 uid 和 gid，用于系统安全考虑
	fsuid,fsgid	文件系统的 uid 和 gid
	suid,sgid	系统调用改变 uid 和 gid 时,用于存放真正的 uid 和 gid
	pid,pgrp,session	进程标识号、组标识号、session 标识号
	leader	是否 session 的主管
时间数据成员	timeout	指出进程间隔多久被重新唤醒
	it_real_value it_real_incr	用于时间片计时
	real_timer	一种定时器结构
	it_virt_value it_virt_incr	进程用户态执行时间的软件定时
	it_prof_value it_prof_incr	进程执行时间的软件定时
	utime,stime,cutime cstime,start_time	进程在用户态、内核态的运行时间，所有层次进程在用户态、内核态的运行时间，创建进程的时间
信号量	semundo	进程每次操作信号量的 undo 操作
	semsleeping	信号量对应的等待队列
上下文	ldt	进程关于段式存储管理的局部描述符指针
	tss	通用寄存器
	saved_kernel_stack	为 MSDOS 仿真程序保存的堆栈指针
	saved_kernel_page	内核堆栈基地址
文件系统	fs	保存进程与 VFS 的关系信息
	files	系统打开文件表
	link_count	文件链的数目
内存数据成员	mm	指向存储管理的 mm_struct 结构
	swappable	指示页面是否可以换出
	swap_address	换出用的地址
	min_flt,maj_flt	该进程累计的缺页次数
	nswap	该进程累计换出的页面数
	cmin_flt,cmaj_flt	该进程及其所有子进程累计的缺页次数
	cnswap	该进程及其所有子进程累计换出的页面数
	swap_cnt	下一次循环最多可以换出的页数
SMP 支持	processor	进程正在使用的 cpu
	last_processor	进程上一次使用的 cpu
	lock_depth	上下文切换时系统内核锁的深度

其他	used_math	是否使用浮点运算器
	comm	进程对应的可执行文件的文件名
	rlim	系统使用资源的限制
	errno	错误号
	debugreg	调试寄存器值
	exec_domain personality	与运行 iBCS2 标准程序有关
	binfmt	指向全局执行文件格式结构，包括 a.out, script, elf, java
	exit_code,exit_signal	返回代码，出错信号名
	dumpable	出错时是否能够进行 memory dump
	did_exec	用于区分新老程序代码
	tty_old_pgrp	进程显示终端所在的组标识
	tty	指向进程所在的终端信息
	wait_chldexit	在进程结束需要等待子进程时处于的等待队列，

2.4 线程及其实现

2.4.1 引入多线程技术的动机

在传统的操作系统中，进程是系统进行资源分配的单位，按进程为单位分给存放其映象的虚地址空间、执行需要的主存空间、完成任务需要的其他各类外围设备资源和文件。同时，进程也是处理器调度的独立单位，进程在任一时刻只有一个执行控制流，我们将这种结构的进程称单线程(结构)进程（Single Threaded Process）。

我们来考察一个文件服务器的例子,当它接受一个文件服务请求后,由于等待磁盘传输而经常被阻塞，假如不阻塞可继续接受新的文件服务请求并进行处理,则文件服务器的性能和效率便可以提高。由于处理这些请求时要共享一个磁盘缓冲区,程序和数据要处于同一个地址空间，而单线程结构的进程难以达到这一目标,需要寻求新概念、提出新机制。随着并行技术、网络技术和软件设计技术的发展，给并发程序设计效率带来了一系列新的问题，主要表现在：

- 进程切换的开销大，频繁的进程调度将耗费大量处理器时间。
- 进程之间通信的代价大，每次通信均要涉及通信进程之间以及通信进程与操作系统之间的切换。
- 进程之间的并发性粒度较粗，并发度不高，过多的进程切换和通信使得细粒度的并发得不偿失。
- 不适合并行计算和分布并行计算的要求。对于多处理器和分布式的计算环境来说，进程之间大量频繁的通信和切换，会大大降低并行度。
- 不适合客户/服务器计算的要求。对于 C/S 结构来说，那些需要频繁输入输出并同时大量计算的服务器进程(如数据库服务器、事务监督程序)很难体现效率。

这就迫切要求操作系统改进进程结构，提供新的机制，使得应用能够按照需求在同一进程中设计出多条控制流，多控制流之间可以并行执行，切换不需通过进程调度；多控制流之间还可以通过内存区直接通信，降低通信开销。这就是近年来流行的多线程（结构）进程（Multiple Threaded process）。如果说操作系统中引入进程的目的是为了使多个程序并发执行，以改善资源使用率和提高系统效率，那么，在操作系统中再引入线程，则是为了减少程序并发执行时所付出的时空开销，使得并发粒度更细、并发性更好。这里解决问题的基本思路是：把进程的两项任务分离开来,单独处理，并由进程和线程分别完成。进程作为系统资源分配和保护的独立单位，它不需要频繁地切换;线程作为系统调度和分派的基本单位，会被频繁地调度和切换，在这种指导思想下，产生了线程的概念。

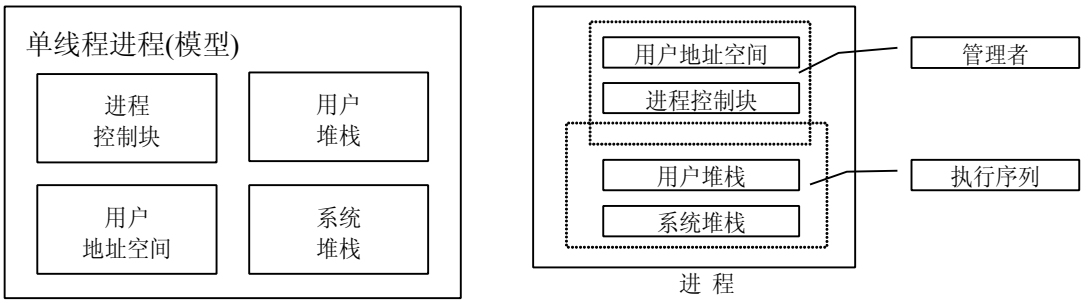
传统操作系统一般只支持单线程进程，如 MS-DOS 支持单用户进程，进程是单线程的；传统的 Unix 支持多用户进程，每个进程也是单线程的。目前，很多著名的操作系统都支持多线程（结构）进程，如：Solaris 、Mach 、SVR4、OS/390、OS/2、WindowNT、Chorus 等；JAVA 的运行引擎则是单进程多线程的例子。许多计算机公司都推出了自己的线程接口规范，如 Solaris Thread 接口规范、OS/2 Thread 接口规范、Windows NT Thread 接口规范等；IEEE 也推出了多线程程序设计标准 POSIX 1003.4a，可以相信多线程技术在程序设计中将会被越来越广泛地采用。

2.4.2 多线程环境中的进程与线程

2.4.2.1 多线程环境中的进程概念

在传统操作系统的单线程进程(模型)中，进程和线程概念可以不加区别。图 2-28 给出了单线程进程的内存布局和运行，它由进程控制块和用户地址空间，以及管理进程执行的调用/返回行为的系统堆栈或用户堆栈构成。一个进程的运行可以划分成两个部分：对资源的管理和实际的指令执行序列。显然，采用并发多进程程序设计时，并发进程之间的切换和通信均要要借助于操作系统的进程管理和进程通信机制，因而实现代价较大,而较大的进程切换和进程通信代价，又进一步影响了并发的粒度。

图 2-28　单线程进程的内存布局和运行



设想是否可以把进程的管理和执行相分离，如图 2-29 所示，进程是操作系统中进行保护和资源分配的单位，允许一个进程中包含多个可并发执行的控制流，这些控制流切换时不必通过进程调度，通信时可以直接借助于共享内存区，这就是并发多线程程序设计。

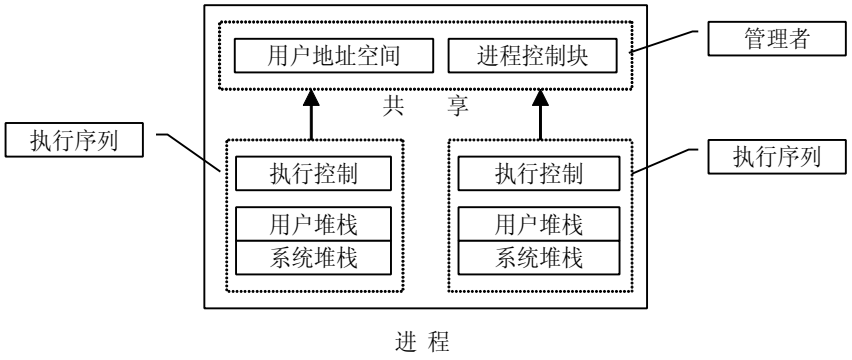
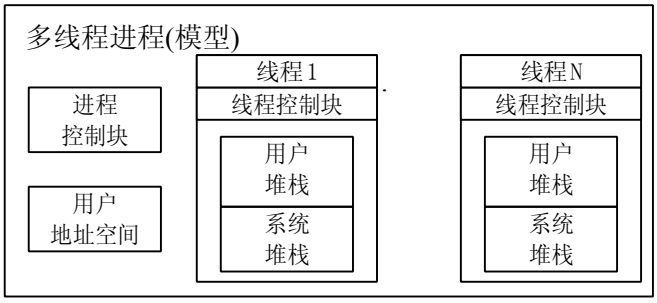


图 2-29　管理和执行相分离的进程模型

多线程进程的内存布局如图 2-30 所示，在多线程环境中，仍然有与进程相关的是 PCB 和用户地址空间，而每个线程除了有独立堆栈，以及包含现场信息和其它状态信息外,也要设置线程控制块 TCB(Thread Control Block)。线程间的关系较为密切，一个进程中的所有线程共享其所属进程拥有的资源。它们驻留在相同的地址空间，可以存取相同的数据。例如，当一个线程改变了主存中一个数据项时，如果这时其它线程也存取这个数据项，它便能看到相同的结果。

图 2-30　多线程进程的内存布局



最后给出多线程环境中进程的定义：进程是操作系统中进行保护和资源分配的独立单位。它具有：

- 一个虚拟地址空间，用来容纳进程的映像；
- 对处理器、其他(通信的)进程、文件和 I/O 资源等的存取保护机制。

2.4.2.2 多线程环境中的线程概念

线程则是指进程中的一条独立执行路径（控制流），每个进程内允许包含多个并行执行的路径（控制流），这就是多线程。线程是系统进行处理器调度的基本单位，同一个进程中的所有线程共享进程获得的主存空间和资源，但不拥有资源。线程具有：

- 一个线程执行状态（运行、就绪、…）;
- 当线程不运行时,有一个受保护的线程上下文,用于存储现场信息。所以，线程也可被看作是执行在进程内的一个独立的程序计数器;
- 一个执行堆栈
- 一个容纳局部变量的主存存储区。

线程还具有以下特性：

- 并行性：同一进程的多个线程可在一个/多个处理器上并发或并行地运行。
- 共享性：同一个进程中的所有线程共享进程获得的主存空间和一切资源，因而，线程需要同步和通信机制。
- 动态性：线程也是程序在相应数据集上的一次执行，由创建而产生，至撤销而消亡，有其生命周期。

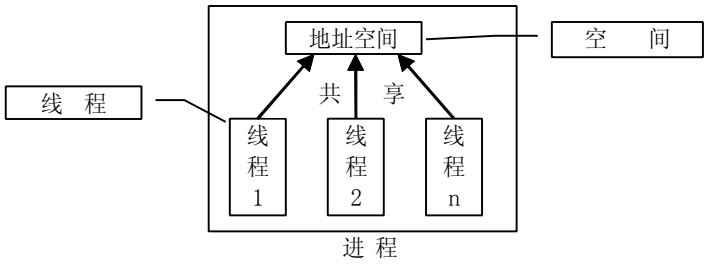


图 2-31　线程的内存布局

如图 2-31 所示，进程支撑线程运行，为线程提供地址空间和各种资源,它封装了管理信息，包括对指令代码、全局数据和 I/O 状态数据等共享部分的管理。线程封装了执行信息，具有并发性，包括对 CPU 寄存器、执行栈（用户栈、内核栈）和局部变量、过程调用参数、返回值等线程私有部分的管理。

线程又称轻量进程(Light weight Process),因为,它运行在进程的上下文中,并使用进程的资源和环境。注意，系统调度的基本单位是线程而不是进程,所以,每当创建一个进程时，至少要同时为该进程创建一个线程，否则该进程无法被调度执行。

2.4.2.3 线程的状态

和进程类似，线程也有一个生命周期，因而也存在各种状态。从理论上说，线程的关键状态有：运行、就绪和阻塞。另外，线程的状态转换也类似于进程。挂起状态对线程是没有意义的，如果一个进程挂起后被对换出主存，则它的所有线程因共享了进程的地址空间，也必须全部对换出去。

当处于运行态的线程阻塞时，对于某些线程实现机制，所在进程也转换为阻塞态，即使这个进程存在另外一个处于就绪态的线程；对于另一些线程实现机制，如果存在另外一个处于就绪态的线程，则调度该线程处于运行状态，否则进程才转换为阻塞态。显然前一种做法欠妥，丢失了多线程机制的优越性，降低了系统的灵活性。

多线程进程的进程状态是怎样的?由于进程不是调度单位，不必划分成过细的状态，如有的系统仅分成可运行和不可运行态，挂起状态属于不可运行态。

2.4.2.4 线程管理和线程库

和进程类似，也可以按照线程状态用队列指引元把 TCB 链接起来进行管理。操作系统进程管理程序为一个应用程序创建进程时，同时为该进程创建第一个线程，以后在线程运行过程中，根据需要由线程创建新线程，但线程之间不存在父子关系。不同的操作系统为线程的管理和控制提供了不同的线程控制原语，基本的线程控制原语有：

- 孵化（Spawn）：又称创建线程。当一个新进程被生成后，该进程的一个线程也就被创建。此后，该进程中的一个线程可以孵化同一进程中的其它线程，并为新线程提供指令计数器和变量。一个新线程还将被分配寄存器上下文和堆栈空间，并将其加入就绪队列。
- 封锁（Block）：又称线程阻塞或等待。当一个线程等待一个事件时，将变成阻塞态,保护它的用户寄存器、程序计数器和堆栈指针等现场。处理器现在就可以转向执行其它就绪线程。
- 活化（Unblock）：又称恢复线程。当被阻塞线程等待的事件发生时，线程变成就绪态或相应状态。
- 结束（Finish）：又称撤销线程。当一个线程正常完成时，便回收它占有的寄存器和堆栈等资

源, 撤销线程 TCB。当一个线程运行出现异常时，允许强行撤销一个线程。

很多基于多线程的操作系统和语言都提供了线程库，如 Mach 的 C-threads 和 Java 线程库,供应用程序共享，支持应用程序创建、调度、和管理用户级线程的运行。线程库实质上是多线程应用程序的开发和运行支撑环境。一般地说，线程库至少应提供以下功能的过程调用：孵化、封锁、活化、结束、通信、同步、调度等。每个线程库应提供给用户级的 API 编程使用。

2.4.2.5 并发多线程程序设计的优点

在一个进程中包含多个并行执行的控制流，而不是把多个可并发的控制流一一分散在多个进程中，这是并发多线程程序设计与并发多进程程序设计的不同之处。

并发多线程程序设计的主要优点是使系统性能获得很大提高，具体表现在：

- 快速线程切换。进程具有独立的虚地址空间，以进程为单位进行任务调度，系统必须交换地址空间，切换时间长，而在同一进程中的多线程共享同一地址空间，因而，能使线程快速切换。
- 减少（系统）管理开销。对多个进程的管理（创建、调度、终止等）系统开销大，如响应客户请求建立一个新的服务进程的服务器应用中，创建的开销比较显著。面对创建、终止线程，虽也有系统开销，但要比进程小得多。
- （线程）通信易于实现。为了实现协作，进程或线程间需要交换数据。对于自动共享同一地址空间的各线程来说，所有全局数据均可自由访问，不需什么特殊设施就能实现数据共享。而进程通信则相当复杂，必须借助诸如通信机制、消息缓冲、管道机制等设施，而且还要调用内核功能，才能实现。
- 并行程度提高。许多多任务操作系统限制用户能拥有的最多进程数目，如 Unix 一般为 50 个，这对许多并发应用来说是不够的。而对多线程技术来说，一般可达几千个，基本上不存在线程数目的限制。
- 节省内存空间。多线程合用进程地址空间，而不同进程独占地址空间，使用不经济。由于队列管理和处理器调度是以线程为单位的，因此，多数涉及执行的状态信息被维护在线程组数据结构中。然而，存在一些影响到一个进程中的所有线程的活动，操作系统必须在进程级进行管理。挂起（Suspension）意味着将主存中的地址空间对换到盘上，因为，在一个进程中的所有线程共享同一地址空间，此时，所有线程也必须进入挂起状态。相似地，终止一个进程时，所有线程应被终止。

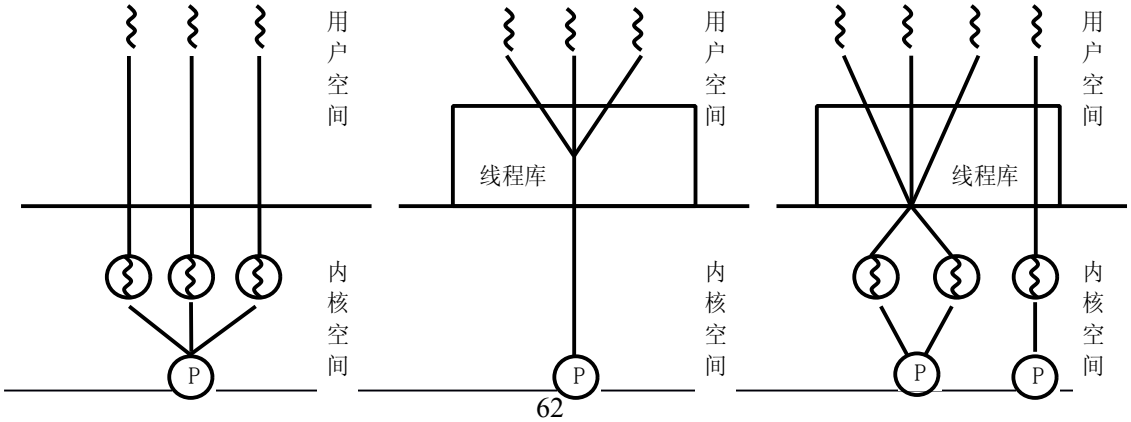
2.4.2.6 多线程技术的应用

多线程技术在现代计算机软件中得到了广泛的应用，取得了较好的效果。下面举例说明多线程技术的一些主要应用：

- 前台和后台工作。如在一个电子表格软件中，一个线程执行显示菜单和读入用户输入，同时，另一个线程执行用户命令和修改电子表格。
- C/S 应用模式。局域网上文件（网络）服务器处理多个用户文件（任务）请求时，创建多个线程，若该服务器是多 CPU 的，则同一进程中的多线程可以同时运行在不同 CPU 上。
- 加快执行速度。一个多线程进程在计算一批数据的同时，读入设备（网络、终端、打印机、硬盘）上的下一批数据，而这分别由两个线程实现。
- 设计用户接口。每当用户要求执行一个动作时，就建立一个独立线程来完成这项动作。当用户要求有多个动作时，就由多个线程来实现，窗口系统应有一个线程专门处理鼠标的动作。例如，GUI 中，后台进行屏幕输出或真正计算；同时，要求对用户输入（鼠标）作出反映。有了多线程，可用处理 GUI 输入线程和后台计算线程，便能实现这一功能。

2.4.3 线程的实现

从实现的角度看，线程可以分成用户级线程 ULT(如，Java 和 Informix)和内核级线程 KLT(如 OS/2)，也有一些系统(如，Solaris)提供了混合式线程，同时支持两种线程实现。图 2-32 给出了各种线程实现方法。



1）内核级线程　　2）用户级线程　　3）混合式线程

图 2-32　线程实现方法

2.4.3.1 内核级线程

在纯内核级线程设施中，线程管理的所有工作由操作系统内核来做。内核专门提供了一个 KLT(Kernel Level Threads)应用程序设计接口（API），供开发者使用，应用程序区不需要有线程管理的代码。Windows NT 和 OS/2 都是采用这种方法的例子。

任何应用都可以被程序设计成多个线程，当提交给操作系统执行时，内核为它创建一个进程和一个线程，线程在执行中可以通过内核创建线程和原语来创建其他线程，这个应用的所有线程均在一个进程中获得支持。内核要为整个进程及进程中的单个线程维护现场信息，所以，应在内核中建立和维护 PCB 及 TCB,内核的调度是在线程的基础上进行的。

这一方法有二个主要优点，首先，在多个处理器上，内核能够同时调度同一进程中多个线程并行执行；其次，若进程中的一个线程被阻塞了，内核能调度同一进程的其它线程占有处理器运行。最后,由于内核线程仅有很小的数据结构和堆栈，KLT 的切换也不需要改变内存信息,切换比较快，内核自身也可以用多线程技术实现，从而，能提高系统的执行速度和效率。

KLT 的主要缺点是：应用程序线程在用户态运行，而线程调度和管理在内核实现,在同一进程中，控制权从一个线程传送到另一个线程时需要用户态-内核态-用户态的模式切换，系统开销较大。

2.4.3.2 用户级线程

纯 ULT(User Level Threads)设施中，线程管理的全部工作都由应用程序来做，内核是不知道线程的存在的。用户级多线程由线程库来实现，任何应用程序均需通过线程库进行程序设计，再与线程库连接后运行来实现多线程。线程库是一个 ULT 管理的例行程序包，它包含了建立/撤销线程的代码、在线程间传送消息和数据的代码、调度线程执行的代码、以及保护和恢复线程上下文的代码，实质上线程库是线程的运行支撑环境。

当一个应用程序提交给系统后，系统为它建立一个由内核管理的进程，该应用程序在线程库环境下开始运行时，只有一个由线程库为进程建立的线程。首先运行这个线程,当应用进程处于运行状态时，线程通过调用线程库中的“孵化”过程，可以孵化出运行在同一进程中的新线程,步骤如下：通过过程调用把控制权传送给这个“孵化”过程， 线程库为新线程创建一个 TCB 数据结构，并置为就绪态，由线程库按一定的调度算法把控制权传送给进程中处于就绪态的一个线程。当控制权传送到线程库时，当前线程的现场信息被保存，而当控制权由库传送给线程时，便恢复它的现场信息。现场信息主要包括：用户寄存器内容、程序指令计数器、堆栈指针。

上述活动均发生在用户空间，且在单个进程中，内核并不知道这些活动。内核按进程为单位调度，并赋予一个进程状态（就绪、运行、阻塞...）。下面的例子清楚地表明了线程调度和进程调度之间的关系。假设进程 B 正在执行它的线程 3，则可能出现下列情况：

- 正在执行的进程 B 的线程 3 发出了一个封锁 B 的系统调用，例如，做了一个 I/O 操作，通知内核进行 I/O 并将进程 B 置为等待状态，按照由线程库所维护的数据结构，进程 B 的线程 3 仍然处在运行态。十分重要的是线程 3 并不实际地在一个处理器上运行，而是可理解为在线程库的运行态中。这时，进程 B 为等待态，但线程为线程库运行态。
- 一个时钟中断传送控制给内核，内核中止当前时间片用完的进程 B，并把它放入就绪队列，切换到另一个就绪进程，此时，按由线程库维护的数据结构，进程 B 的线程 3 仍处于运行态。这时，进程 B 已处于就绪态，但线程为线程库运行态。

上述两种情况中，当内核切换控制权返回到进程 B 时，便恢复执行线程 3。注意到当正在执行线程库中的代码时，一个进程也有可能由于时间片用完或被更高优先级的进程剥夺而被中断。当中断发生时，一个进程可能正处在从一个线程切换到另一个线程的过程中间；当一个进程恢复时，继续在线程库中执行，完成线程切换，并传送控制权给进程中的一个新线程。使用 ULT 代替 KLT 有许多优点：

- 线程切换不需要内核特权方式，因为，所有线程管理数据结构均在单个进程的用户空间中，管理线程切换的线程库也在用户地址空间运行，因而，进程不要切换到内核方式来做线程管理。这就节省了模式切换的开销,也节省了内核的宝贵资源。
- 按应用特定需要来调度，一种应用可能从简单轮转调度算法得益，同时，另一种应用可能从优先级调度算法获得好处。在不干扰 OS 调度的情况下，根据应用需要可以裁剪调度算法,也就是说，线程库的线程调度算法与操作系统的进程调度算法是无关的。
- ULT 能运行在任何 OS 上，内核支持 ULT 方面不需要做任何改变。线程库是可以被所有应用共享的应用级实用程序,许多当代操作系统和语言均提供了线程库,传统 Unix 并不支持多线程，但已有了多个基于 Unix 的用户线程库。

和 KLT 比较，ULT 有二个明显的缺点：

- 在传统的基于进程操作系统中，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，进程内的所有线程会被阻塞。
- 在纯 ULT 中，多线程应用不能利用多重处理的优点。内核在一段时间里，分配一个进程仅占用一个 CPU，因而进程中仅有一个线程能执行。因此尽管多道程序设计能够明显地能加快应用

处理速度，也具备了在一个进程中进行多线程设计的能力，但我们不可能得益于并发地执行一部分代码。

克服上述问题的方法有两种。一是用多进程并发程序设计来代替多线程并发程序设计，这种方法事实上放弃了多线程带来的所有优点。第二种方法是采用 jacketing 技术来解决阻塞线程的问题。主要思想是把阻塞式的系统调用改造成非阻塞式的，当线程调用系统调用，首先调用 jacketing 实用例程，来检查资源使用情况，以决定是否执行系统调用或传递控制权给另一个线程。

2.4.3.3 混合式线程

有些操作系统提供了混合式 ULT/KLT，Solaris 便是一个例子。在混合系统中，内核支持 KLT 多线程的建立、调度和管理，同时,也提供线程库，允许用户应用程序建立、调度和管理 ULT。一个应用程序的多个 ULT 映射成一些 KLT,程序员可按应用需要和机器配置调整 KLT 数目，以达到较好效果。

混合式中，一个应用中的多个线程能同时在多处理器上并行运行，且阻塞一个线程时并不需要封锁整个进程。如果设计得当的话，则混合式多线程机制能够结合了两者优点，并舍去它们的缺点。

2.4.4 实例研究：Solaris 的进程与线程

2.4.4.1 Solaris中的进程与线程概念

Solaris 采用了 4 个与进程和线程有关的概念,用来支持完全可抢占、SMP、和核心多线程结构：

- 进程（Process）：通常的 Unix 进程，它包含用户的地址空间和 PCB。
- 用户级线程（User-Level Threads）：通过线程库在用户地址空间中实现，对操作系统来讲是不可见的，用户级线程（ULT）是应用程序并行机制的接口。
- 轻量进程（Light Weight Process）：一个 LWP 可看作是 ULT 和 KLT 之间的映射，每个 LWP 支持一个或多个 ULT 并映射到一个 KLT 上。LWP 是核心独立调度的单位，它可以在多个处理器上并行执行。
- 内核级线程（Kernel-Level Threads）：即 KLT，它们是能被调度和指派到处理器上去运行的基本实体。

Solaris 的线程实现分为二个层次：用户层和核心层，用户层在用户线程库中实现；核心层在操作系统内核中实现。处于两个层次中的线程分别叫用户级线程和内核级线程。

ULT 是一个代表对应线程的数据结构，它是纯用户级的概念，占用用户空间资源，对核心是透明的。ULT 和 KLT 不是一一对应，通过轻量级进程 LWP 来映射两者之间的联系。一个进程可以设计为一个或多个 LWP，在一个 LWP 上又可以开发多个 ULT，LWP 与 ULT 一样共享进程的资源。KLT 和 LWP 是一一对应的，一个 ULT 要通过核心 KLT 和 LWP 二级调度后才真正占有处理器运行。

有了 LWP，就可以在用户级实现 ULT，每个进程可以创建几十个 ULT，而又不占用核心资源。由于 ULT 共享用户空间，因此当 LWP 在一个进程的不同 ULT 间切换时，仅是数据结构的切换，其时间开销远低于两个 KLT 间的切换时间。同样线程间的同步也独立于核心的同步体系，在用户空间中独立实现，而不陷入内核。核心能看见的是 LWP，而 ULT 是透明的。

多线程的程序员接口包括二个层次：一层为线程接口，提供给用户，以编写多线程应用程序，这一层由动态连接库引用 LWP 实现，线程库在 LWP 之上调度线程，同步也在线程库实现。第二层为 LWP 接口，提供给线程库，以管理 LWP。这一层由核心通过 KLT 实现。LWP 通过这些接口访问核心资源。

多线程的两个程序员接口是类似的，线程接口的切换代价较小，特别适用于解决逻辑并行性问题；而 LWP 接口则适用于那些需要在多个处理器进行并行计算的问题。如果程序设计得当的话，混合应用两种程序员接口可以带来更大的灵活性和优越性。

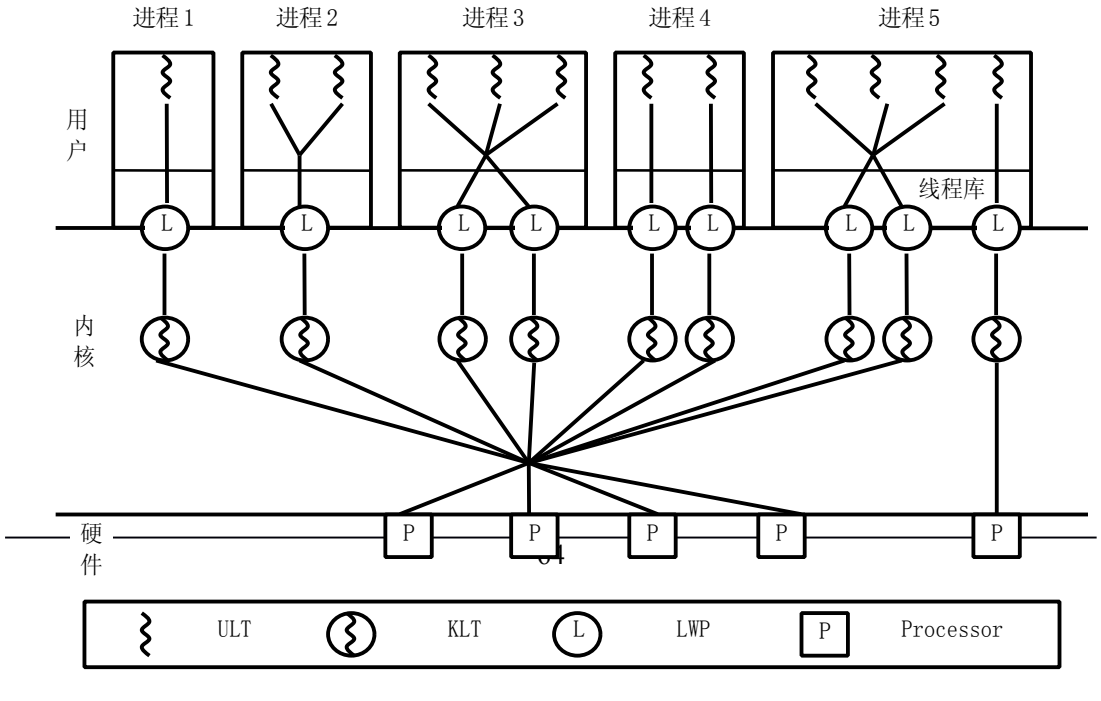


图 2-33　Solaris 线程的使用

图 2-33 显示了 4 个实体间的关系，LWP 和 KLT 是一一对应的，在一个应用进程中 LWP 是可见的，LWP 的数据结构存在于它们各自的进程地址空间中。同时，每个 LWP 与一个可调度的内核线程绑定，而内核线程的数据结构则维护在内核地址空间中。

图中，进程 1 是传统的单线程进程，当应用不需要并发运行时，可以采用这种结构。进程 2 是一个纯的 ULT 应用，所有的 ULT 由单个内核线程支撑，因而，某一时刻仅有一个线程可运行。对于那些在程序设计时要表示并发而不真正需要多线程并行执行的应用来说，这种方式是有用的。进程 3 是多线程与多 LWP 的对应，Solaris 允许应用程序开发多个 ULT 工作在小于或等于 ULT 个数的 LWP 上。这样应用可以定义进程在内核级上的某种程度的并行。进程 4 的线程与 LWP 是一对一地捆扎起来的，这种结构使得内核并行机制完全对应用可见，这在线程常常因阻塞而被挂起时是有用的。进程 5 包括多 ULT 映射到多 LWP 上，以及 ULT 与 LWP 的一一捆绑，并且还有一个 LWP 捆在单个处理器上。

2.4.4.2 Solaris的进程结构

在 Solaris 中已经有了进程、KLT 和 ULT，为什么还要引入 LWP?我们先来讨论这个问题。最主要的原因是各种应用的需求,有些应用逻辑并行性程度高, 有些应用物理并行性要求高。像窗口系统是典型的逻辑并行性程度高的应用,同时在屏幕上开出了多个窗口，窗口切换很频繁，但一个时刻仅有少数窗口处于活跃状态。我们可以用一组 ULT 来表达窗口系统，用一个或很少几个 LWP 来支持这一组 ULT,可根据活跃窗口数目调正 LWP 的个数。由于 ULT 是由用户线程库实现的，他们的管理不涉及内核;内核仅要管理与 ULT 对应的一个或几个 LWP，系统开销小，窗口系统的效率高。

大规模并行计算是物理并行性要求高的应用,可以把数组按列划分给不同的 ULT 线程处理,如果每个 CPU 对应一个 LWP,而一个 LWP 对应多个 ULT，那么，CPU 有很多时间化在线程切换上。这种情况下，最好把列分给少量 ULT,而一个 ULT 和一个 LWP 绑定,以减少线程切换次数,提高并行计算的效率。

Solaris 中，每个进程包括不同数目的轻量进程，进程结构如图 2-34 所示，因进程不再是调度单位，所以其进程结构中的处理器信息取消，而加进一个包含 LWP 的数据结构。

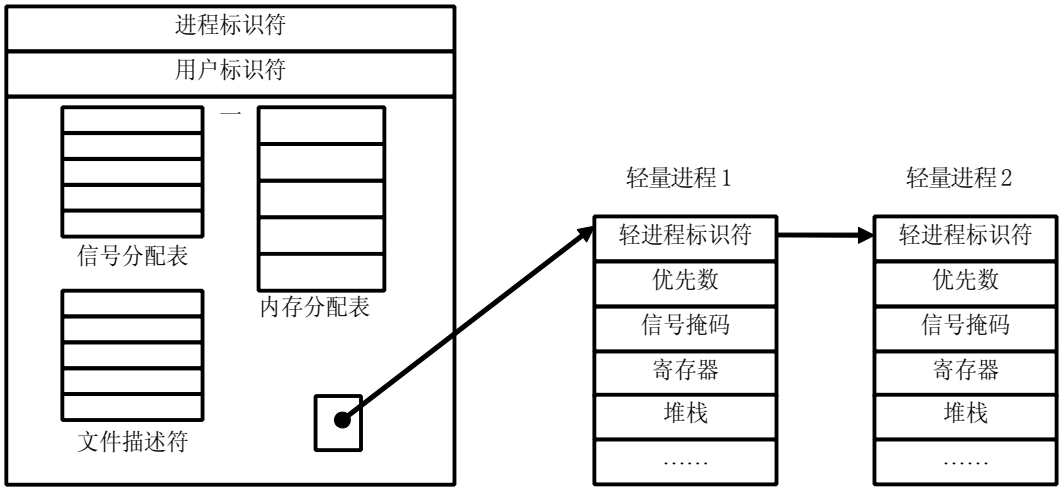


图 2-34　Solaris 的进程结构

LWP 的数据结构包括：

- 轻量进程标识符标识了轻量进程
- 优先数定义了轻量进程执行的优先数
- 信号掩码定义了内核能够接受的信号
- 寄存器域用于存放轻量进程让出处理器时的现场信息
- 轻量进程的内核栈包括每个调用层次的系统调用的参数、返回值和出错码
- 交替的信号堆栈
- 用户，或用户和系统共同的虚时间警示
- 用户时间和系统处理器使用
- 资源使用和预定义数据
- 指向对应内核线程的指针
- 指向进程结构的指针

轻量进程实际上不是进程，只是一个内核支持的 ULT，因此它有一个内核堆栈，都有一个对应的 KLT 支持。LWT 被独立调度，进程中的 LWP 共享进程地址空间和资源，一个进程由于有多个 LWP，才有可

能分得多个 CPU,才能实现物理上的并行性。

ULT 的数据结构包括：

- 线程标识符标识了线程
- 优先数定义了线程执行的优先数
- 信号掩码定义了能够接受的信号
- 寄存器域用于存放线程让出处理器时的现场信息
- 堆栈用于存放线程运行数据
- 线程局部存储器用于存放线程局部数据
- KLT 的数据结构(每个 LWP 对应一个) 是由内核创建的线程，每个 KLT 有一个堆栈和一个数据结构,包括：
- 内核寄存器数据保存区
- 优先级和调度信息
- KLT 的队列指针和堆栈指针
- 相关 LWP 的指针及信息
- 进程数据结构,包括：
- 与进程相关的 KLT
- 进程地址空间指针
- 用户权限表
- 信号处理程序清单
- 与用户执行有关信息

Solaris 支持实时类进程，其内核是可抢占的，它的内核函数和过程全部用线程来实现，所以，Solaris 的内核是 KLT 的集合。这些 KLT 分两种，一种是负责执行一个指定的内核函数;另一种用来支持和运行 LWP，KLT 是独立被调度和分配到 CPU 上去执行。

2.4.4.3 Solaris的线程状态

图 2-35 简单说明了 ULT 和 LWP 的状态。

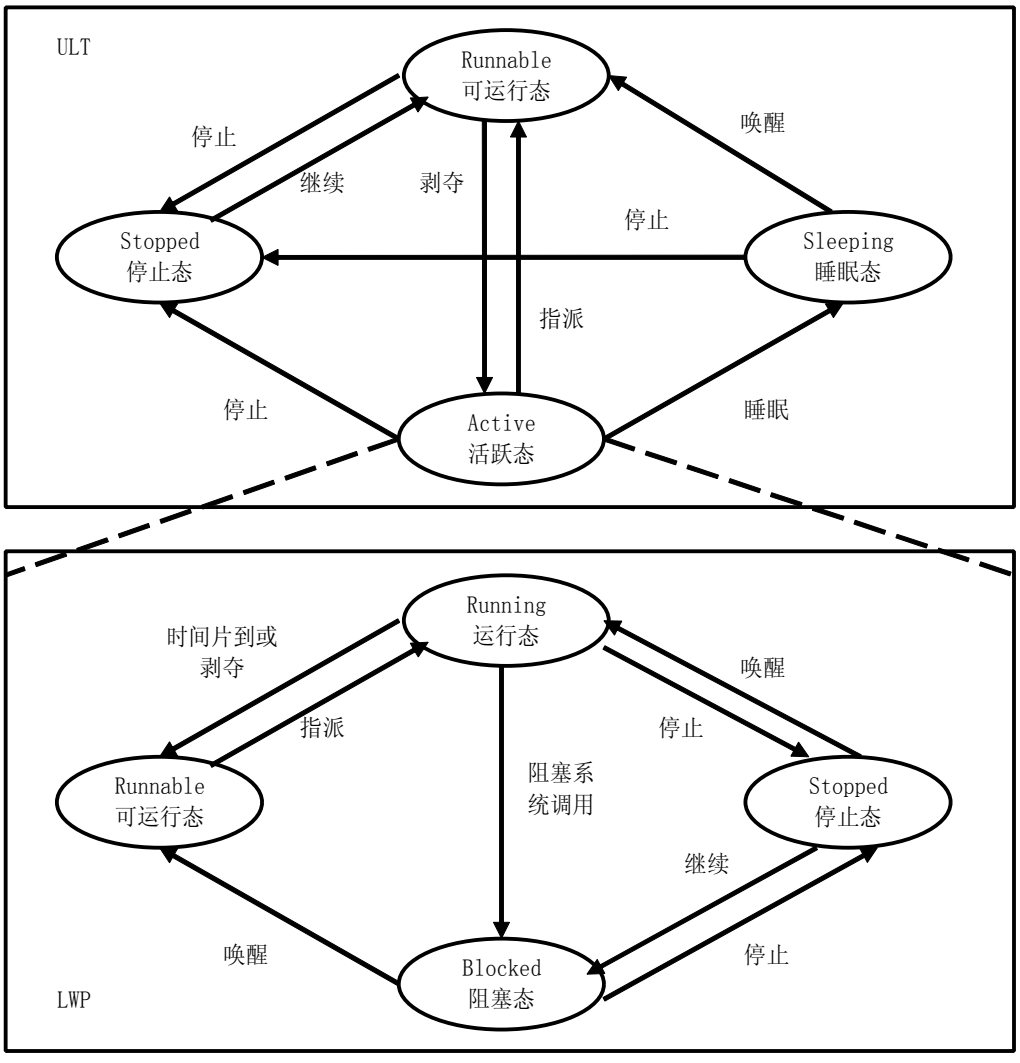


图 2-35　ULT 和 LWP 的状态

用户级线程的执行是由线程库管理的，让我们首先考虑非捆绑的线程、即多个 ULT 可共享 LWP，线程可能处于四个状态之一：可运行、活跃、睡眠和停止。处在活跃状态的一个 ULT 是指目前分配到 LWP 上，并且在对应的内核线程 KLT 执行时，它被执行。出现许多事件会导致 ULT 离开活跃态，让我们考虑一个活跃的 ULT 正在执行，下面的事件可能发生：

- 同步：有一个 ULT 调用了一条同步原语与其它线程协调活动。于是它就进入睡眠态，直到同步条件满足后，这个 ULT 才被放入可运行队列。
- 挂起：任何线程可挂起它线程;一个线程也可挂起自己并进入停止状态。直到别的线程发出一个让挂起线程继续运行的请求，再把该线程移入可运行队列。
- 剥夺：一个活跃线程在执行时,另一个更高优先级的线程变为可运行状态，如果处于活跃状态的运行线程优先级较低，它就要被剥夺并放入可运行状态，而更高优先级的线程被分配到可用的 LWP 上执行。
- 让位：若运行线程执行了 thread_yield（）库命令，库中的线程调度程序将查看是否有另一个可运行线程，它与当前执行的线程具有相同的优先级，则把执行线程放入可运行队列，另一个可运行线程分配到可用 LWP 上;否则原线程继续运行。

在所有上述情况中，当 T1 让出活跃态时，线程库选择另一个非捆帮的线程进入可运行队列，并在新的可用的 LWP 上运行它。

图 2-35 还显示了 LWP 的状态转换图，我们可以把它看作是 ULT 活跃状态的详尽描述，因为当一个 ULT 处于活跃状态时只能捆绑到一个 LWP 上。只有当 LWP 处于运行状态时，一个处于活跃状态的 ULT 才真正的处于活跃状态并执行。当一个处于活跃状态的 ULT 调用一个阻塞的系统调用时，则它对应的 LWP 进入阻塞状态，这个 ULT 将继续处于活跃状态并继续与相应的 LWP 绑定，直到线程库显式地做出变动。

下面再讨论 ULT 和 LWP 一一绑定的情况，ULT 和 LWP 的关系只有轻微的不同。例如，当一个 ULT 等待一个同步事件而进入睡眠状态之后，相应的 LWP 也必须停止运行。这种状态转换是通过让 LWP 阻塞在核心层同步变量上完成的。

Solaris 提供了以下线程操作供用户线程程序设计。

thread_create()	创建一个新的线程
thread_setconcurrency()	置并发程度（即 LWP 的数目）
thread _exit()	终止当前进程，收回线程库分配给它的资源
thread_wait()	阻塞当前线程，直到有关线程退出
thread_get_id()	获得线程标识符
thread_sigsetmask() thread_sigprocmask()	置线程信号掩码
Thread_kill()	生成一个送给指定线程的信号
Thread_stop()	停止一个线程的执行
Thread_priority()	置一个线程的优先数

2.4.5 实例研究：Windows 2000 的进程与线程

2.4.5.1 Windows 2000中的进程与线程概念

Windows2000 包括三个层次的执行对象：进程、线程和作业。其中作业是 Windows2000 新引进的，在 NT4 中不存在，它是共享一组配额限制和安全性限制的进程的集合;进程是相应于一个应用的实体，它拥有自己的资源，如主存，打开的文件；线程是顺序执行的工作调度单位，它可以被中断,使 CPU 能转向另一线程执行。

Windows 2000 进程设计的目标是提供对不同操作系统环境的支持,具有：多任务(多进程)、多线程、支持 SMP、采用了 C/S 模型、能在任何可用 CPU 上运行的特点。由内核提供的进程结构和服务相对来说简单、适用，其重要的特性如下：

- 作业、进程和线程是用对象来实现的。
- 一个可执行的进程可以包含一个或多个线程。
- 进程或线程两者均有内在的同步设施。

进程以及它控制和使用的资源的关系如图 3-36 所示。当一个用户首次注册时，Windows2000 为用户

建立一个访问令牌 Access Token，它包括安全标识和进程凭证。这个用户建立的每一个进程均有这个访问令牌的拷贝。内核使用访问令牌来验证用户是否具有存取安全对象或在系统上和安全对象上执行受限功能的能力。访问令牌还控制了进程能否改变自己的属性，在这种情况下，进程没有获得它的访问令牌的句柄，进程若想打开它，安全系统首先决定这是否允许，进而决定进程能否改变自己的属性。

与进程有关的还有一组当前分配给这个进程的虚拟地址空间块，进程不能直接改它，必须依靠为进程提供内存分配服务的虚存管理例程。

进程包括一个对象表，其中包括了这个进程与其它使用资源之间的联系,通过对象句柄便可对某资源进行引用,存取令牌控制进程是否能改变其自身属性。图 2-35 包含了一个线程对象，这个线程可以访问一个文件对象和一个共享内存区对象,有一系列分给进程的虚拟地址空间块。在 Window2000 中，对象(object)是对象类的实例，对象类是实现系统功能的操作和私有变量的封装体；语柄则是对打开了的对象实例的引用。

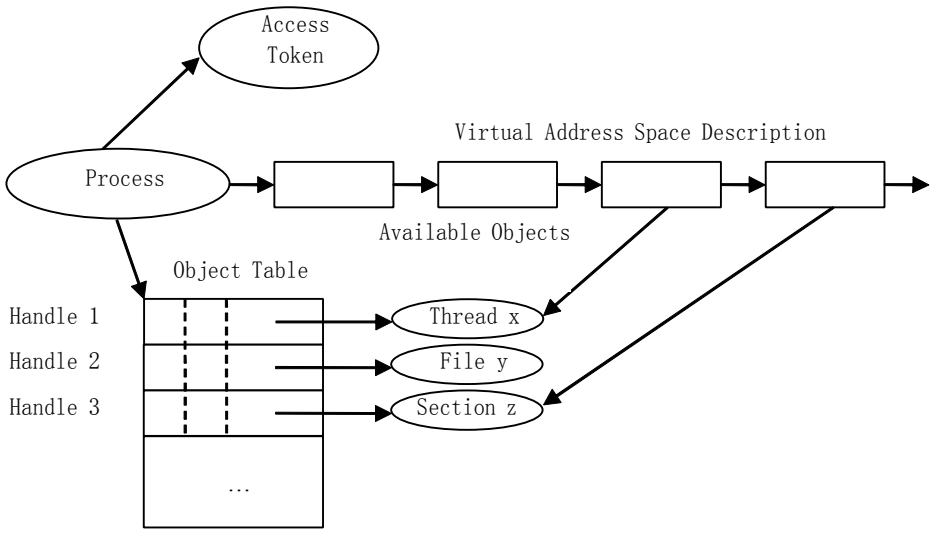


图 2-36　进程以及控制和使用的资源

2.4.5.2 进程对象

进程是由一个通用结构的对象来表示的。每个进程由属性和封装了的若干可以执行的动作和服务所定义。当接受到适当消息时，进程就执行一个服务，只能通过传递消息给提供服务的进程对象来调用这一服务。用户使用进程对象类（或类型）来建立一个新进程，对进程来说，这个对象类被定义作为一个能够生成新的对象实例的模板，并且在建立对象实例时，属性将被赋值。

每一个进程都由一个执行体进程（EPROCESS）块表示。下面给出了 EPROCESS 的结构，它不仅包括进程的许多属性，还包括并指向许多其它相关的属性，如每个进程都有一个或多个执行体线程（ETHREAD）块表示的线程。除了进程环境块（PEB）存在于进程地址空间中以外，EPROCESS 块及其相关的其它数据结构存在于系统空间中。另外，WIN32 子系统进程 CSRSS 为执行 WIN32 程序的进程保持一个平行的结构，该结构存在于 WIN32 子系统的核心态部分 WIN32K.SYS，在线程第一次调用在核心态实现的 WIN32 USER 或 GDI 函数时被创建。

EPROCESS 块中的有关项目的内容如下：

- 内核进程块 KRPROCESS：公共调度程序对象头、指向进程页面目录的指针、线程调度默认的基本优先级、时间片、相似性掩码、用于进程中线程的总内核和用户时间。
- 进程标识符等：操作系统中唯一的进程标识符、父进程标识符、运行映像的名称、进程正在运行的窗口位置。
- 配额块：限制非页交换区、页交换区和页面文件的使用，进程能使用的 CPU 时间。
- 虚拟地址空间描述符 VAD：一系列的数据结构，描述了存在于进程地址空间的状态。
- 工作集信息：指向工作集列表的指针，当前的、峰值的、最小的和最大的工作集大小，上次裁剪时间，页错误计数，内存优先级，交换出标志，页错误历史纪录。
- 虚拟内存信息：当前和峰值虚拟值，页面文件的使用，用于进程页面目录的硬件页表入口。
- 异常/调试端口：当进程的一个线程发生异常或引起调试事件时，进程管理程序发送消息的进程间通信通道。
- 访问令牌：指明谁建立的对象，谁能存取对象，谁被拒绝存取该对象。
- 句柄表：整个进程的句柄表地址。
- 进程环境块 PEB：映像基址、模块列表、线程本地存储数据、代码页数据、临界区域超时、堆栈的数量、大小、进程堆栈指针、GDI 共享的句柄表、操作系统版本号信息、映像版本号信息、映像进程相似性掩码。
- WIN32 子系统进程块：WIN32 子系统的核心组件需要的进程细节。

操作系统还提供了一组用于进程的 WIN32 函数：

- CreateProcess：使用调用程序的安全标识，创建新的进程和线程。
- CreateProcessAsUser：使用交替的安全标识，创建新的进程和线程，然后执行指定的 EXE。
- OpenProcess：返回指定进程对象的句柄。
- ExitProcess：退出当前进程。
- TerminateProcess：终止进程。
- FlushInstructionCache：清空另一个进程的指令高速缓存。
- GetProcessTimes：得到另一个进程的时间信息，描述进程在用户态和核心态所用的时间。
- GetExitCodeProcess：返回另一个进程的退出代码，指出关闭这个进程的方法和原因。
- GetCommandLine：返回传递给进程的命令行字符串。
- GetCurrentProcessID：返回当前进程的 ID。
- GetProcessVersion：返回指定进程希望运行的 Windows 的主要和次要版本信息。
- GetStartupInfo：返回在 CreateProcess 时指定的 STARTUPINFO 结构的内容。
- GetEnvironmentStrings：返回环境块的地址。
- GetEnvironmentVariable：返回一个指定的环境变量。
- GetProcessShutdownParameters：取当前进程的关闭优先级和重试次数。
- SetProcessShutdownParameters：置当前进程的关闭优先级和重试次数。

当应用程序调用 CreateProcess 函数时，就将创建一个 WIN32 进程。创建的 WIN32 进程的过程在操作系统的 3 个部分中分阶段完成，这三个部分是：WIN32 客户方的 KERNEL32.DLL、Windows2000 执行体和 WIN32 子系统进程 CSRSS。具体步骤如下：

- 打开将在进程中被执行的映像文件（.EXE）。
- 创建 Windows2000 执行体进程对象。
- 创建初始线程（堆栈、描述表、执行体线程对象）。
- 通知 WIN32 子系统已经创建了一个新的进程，以便它可以设置新的进程和线程。
- 启动初始线程的执行（除非指定了 CREATE_SUSPENDED 标志）。
- 在新进程和线程的描述表中，完成地址空间的初始化，加载所需的 DLL，并开始程序的执行

2.4.5.3 线程对象

为了能运行，一个进程至少包含一个线程，此后，线程可以创建其它线程，在多处理器系统中，进程的多个线程可以并行执行。线程中的有些属性是进程中复制来的。

每一个线程都由一个执行体线程（ETHREAD）块表示。图 2-37 给出了 ETHREAD 的结构。除了线程环境块（TEB）存在于进程地址空间中以外，ETHREAD 块及其相关的其它数据结构存在于系统空间中。另外，WIN32 子系统进程 CSRSS 为执行 WIN32 程序的线程保持一个平行的结构，该结构存在于 WIN32 子系统的核心态部分 WIN32K.SYS。

ETHRED 块中的有关项目的内容如下：

- 创建和退出时间：线程的创建和退出时间。
- 进程识别信息：进程标识符和指向 EPROCESS 的指引元。
- 线程启动地址：线程启动例程的地址。
- LPC 消息信息：线程正在等待的消息 ID 和消息地址。
- 挂起的 I/O 请求：挂起的 I/O 请求数据包列表。
- 调度程序头信息：指向标准的内核调度程序对象。
- 执行时间：在用户态运行的时间总计和在核心态运行的时间总计。
- 内核堆栈信息指引元：内核堆栈的栈底和栈顶信息。
- 系统服务表指引元：指向系统服务表的指针。
- 调度信息：基本的和当前的优先级、时间片、相似性掩码、首选处理器、调度状态、冻结计数、挂起计数。

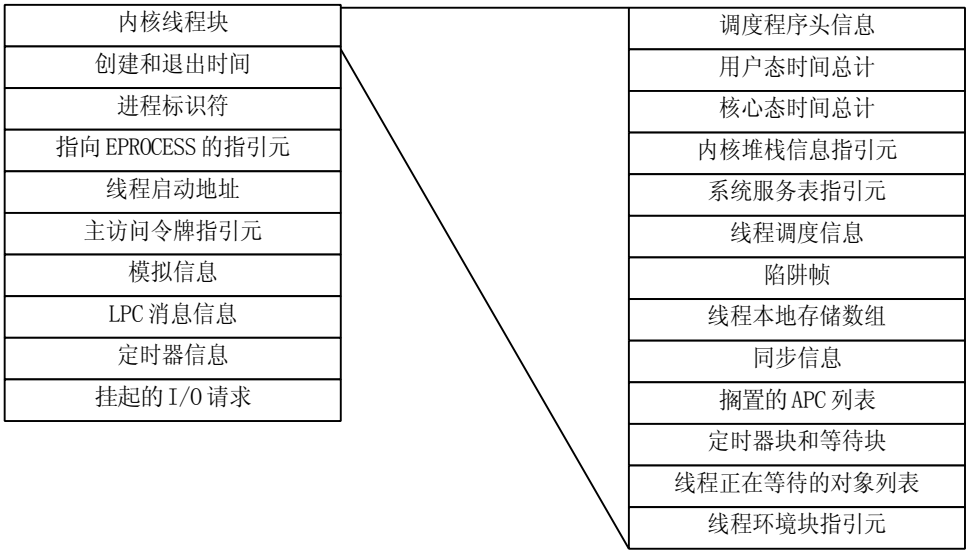


图 2-37　ETHRED 块的结构

操作系统提供了一组用于线程的 WIN32 函数：

- CreateThread：创建新线程。
- CreateRemoteThread：在另一个进程创建线程。
- ExitThread：退出当前线程。
- TerminateThread：终止线程。
- GetExitCodeThread：返回另一个线程的退出代码。
- GetThreadTimes：返回另一个线程的定时信息。
- GetThreadSelectorEntry：返回另一个线程的描述符表入口。
- GetThreadContext：返回线程的 CPU 寄存器。
- SetThreadContext：更改线程的 CPU 寄存器。

当应用程序调用 CreateThread 函数创建一个 WIN32 线程的具体步骤如下：

- 在进程地址空间为线程创建用户态堆栈。
- 初始化线程的硬件描述表。
- 调用 NtCreateThread 创建处于挂起状态的执行体线程对象。包括：增加进程对象中的线程计数，创建并初始化执行体线程块，为新线程生成线程 ID，从非页交换区分配线程的内核堆栈，设置 TEB，设置线程起始地址和用户指定的 WIN32 起始地址，调用 KeInitializeThread 设置 KTHREAD 块，调用任何在系统范围内注册的线程创建注册例程，把线程访问令牌设置为进程访问令牌并检查调用程序是否有权创建线程。
- 通知 WIN32 子系统已经创建了一个新的线程，以便它可以设置新的进程和线程。
- 线程句柄和 ID 被返回到调用程序。
- 除非调用程序用 CREATE_SUSPEND 标识设置创建线程，否则线程将被恢复以便调度执行。

线程是 Windows2000 操作系统的最终调度实体，如图 2-38 所示，它可能处于以下 6 个状态之一：

- 就绪态——可被调度去执行的状态，微内核的调度程序维护所有就绪线程队列，并按优先级次序调度。
- 准备态——已被选中下一个在一个特定处理器上运行。此线程处于该状态等待直到该处理器

可用。如果准备态线程的优先级足够高，则可以从正在运行的线程手中抢占处理器，否则将等待直到运行线程等待或时间片用完。

- 运行态——每当微内核执行进程或线程切换时，准备态线程进入运行态，并开始执行，直到它被剥夺、或用完时间片、或阻塞、或终止为止。在前两种情况下，线程进入到就绪态。
- 等待态——线程进入等待态是由于以下原因： 1）出现了一个阻塞事件（如，I/O）；2）等待同步信号；3）环境子系统要求线程挂起自己。当等待条件满足时，且所有资源可用，线程就进入就绪态态。
- 过渡态——一个线程完成等待后准备运行，但这时资源不可用，就进入过渡态，例如，线程的堆栈被调出内存。当资源可用时，过渡态线程进入就绪态。
- 终止态——线程可被自己、其它线程、或父进程终止。一旦结束工作完成后，线程就可从系统中移去，或者保留下来以备将来初始化再用。

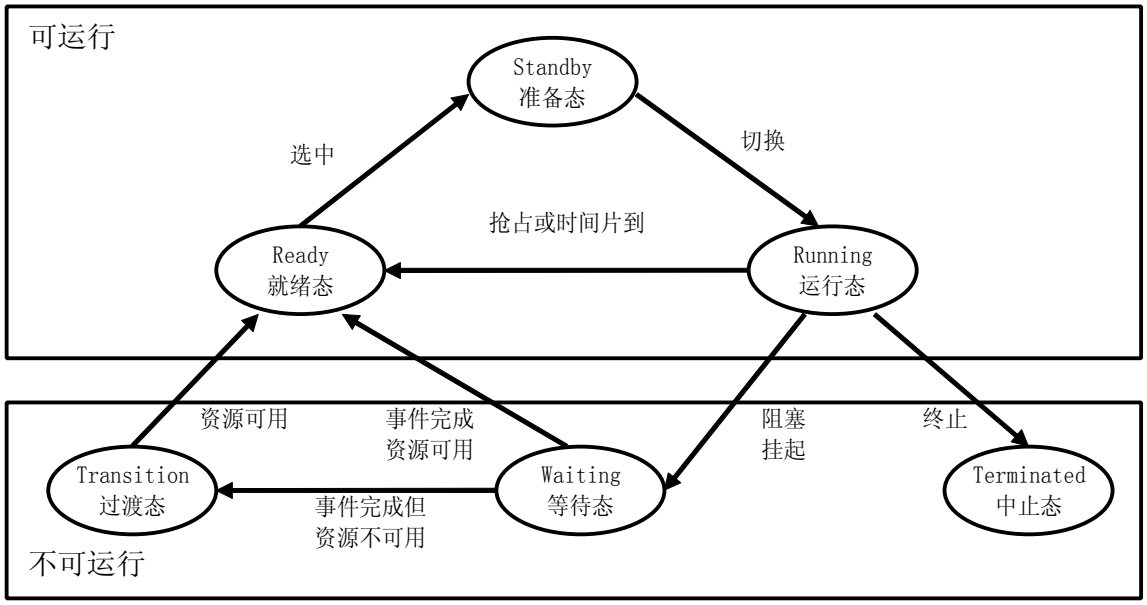


图 2-38　Windows2000 的线程状态

因为在不同进程中的线程可以并发执行，所以操作系统支撑进程之间的并发性。进一步，同一进程中的多个线程可被分配到单独 CPU 去并发执行，一个多线程进程可以达到并发性而不必付出象多进程并发那样的开销。同一进程中的线程能通过公共地址空间来交换信息和存取进程的可享资源。在不同进程中的线程，可通过在二个进程中已建立起的共享存储来交换信息。

一个面向对象的多线程进程是实现服务器应用的有效手段，如当一个服务器进程给多个客户服务时，每个客户请求将触发建立服务器进程创建一个新线程为其服务。

2.4.5.4 作业对象

Windows2000 包含一个被称为“作业”的进程模式扩展。作业对象是一个可命名的、保护、共享的对象，它能够控制与作业有关的进程属性。作业对象的基本功能是允许系统将进程组看作是一个单元，对其进行管理和操作。有些时候，作业对象可以补偿 NT4 在结构化进程树方面的缺陷。作业对象也为所有与作业有关的进程和所有与作业有关但已被终止的进程纪录基本的账号信息。

作业对象包含一些对每一个与该作业有关的进程的强制限制，这些限制包括：

- 默认工作集的最小值和最大值。
- 作业范围用户态 CPU 时限。
- 每个进程用户态 CPU 时限。
- 活动进程的最大数目。
- 作业处理器相似性。
- 作业进程优先级类。

用户也能够在作业中的进程上设置安全性限制。用户可以设置一个作业，使得每一个进程运行在同样的作业范围的访问令牌下。然后，用户就能够创建一个作业，限制进程模仿或创建其访问令牌中包括本地管理员组的进程。另外，用户还可以应用安全筛选，当进程中的线程包含在作业模仿客户机线程中时，将从模仿令牌中删除特定的特权和安全 ID(SID)。

最后，用户也能够在作业中的进程上设置用户接口限制。其中包括：限制进程打开作业以外的线程所拥有的窗口句柄，对剪贴板的读取或写入，通过 WIN32 SystemParameterInfo 函数更改某些用户接口系

统参数等。

一个进程只能属于一个作业，一旦进程建立，它与作业的联系便不能中断；所有由进程创建的进程和它们的后代也和同样的作业相联系。在作业对象上的操作会影响与作业对象相联系的所有进程。

有关作业对象的 WIN32 函数包括：

- CreateJobObject：创建作业对象。
- Open JobObject：通过名称打开现有的作业对象。
- AssignProcessToJobObject：添加一个进程到作业。
- TerminateJobObject：终止作业中的所有进程。
- SetInformationToJobObject：设置限制。
- QueryInformationToJobObject：获取有关作业的信息，如：CPU 时间、页错误技术、进程的数目、进程 ID 列表、配额或限制、安全限制等。

2.5 处理机调度

在计算机系统中，可能同时有数百个批处理作业存放在磁盘的作业队列中，或者有数百个终端与主机相连接。如何从这些作业中挑选作业进入主存运行、如何在进程之间分配处理器时间，无疑是操作系统资源管理中的一个重要问题。这一涉及处理机分配的问题，称之为处理机调度。

2.5.1 处理机调度的层次

用户作业从进入系统成为后备作业开始，直到运行结束退出系统为止，可能会经历三级调度。如图 2-39 所示，处理机调度可以分为以下三个层次：

- 高级调度(High Level Scheduling)：又称作业调度、长程调度(Long-term Scheduling)。它将按照系统预定的调度策略决定把后备队列作业中的哪些作业调入主存，为它们创建进程并启动它们运行。在批处理操作系统中，作业首先进入系统在辅存上的后备作业队列等候调度，因此，作业调度是必须的。在纯粹的分时或实时操作系统中，通常不需要配备作业调度。
- 中级调度(Medium Level Scheduling)：又称平衡负载调度，中程调度(Medium-term Scheduling)。它决定主存储器中所能容纳的进程数，这些进程将允许参与竞争处理器资源。而有些暂时不能运行的进程被调出主存，这时这个进程处于挂起状态，当进程具备了运行条件，且主存又有空闲区域时，再由中级调度决定把一部分这样的进程重新调回主存工作。中级调度根据存储资源量和进程的当前状态来决定辅存和主存中的进程的对换。
- 低级调度(Low Level Scheduling)：又称进程调度、短程调度(Short_term Scheduling)。它的主要功能是按照某种原则决定就绪队列中的哪个进程或内核级线程能获得处理器，并将处理机出让给它进行工作。

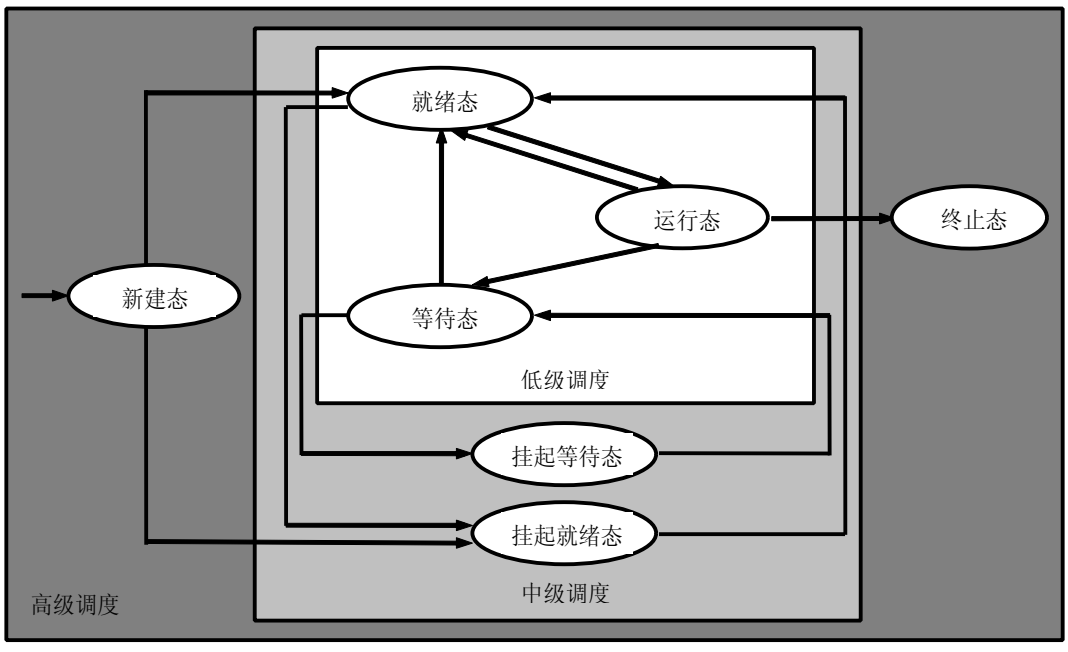
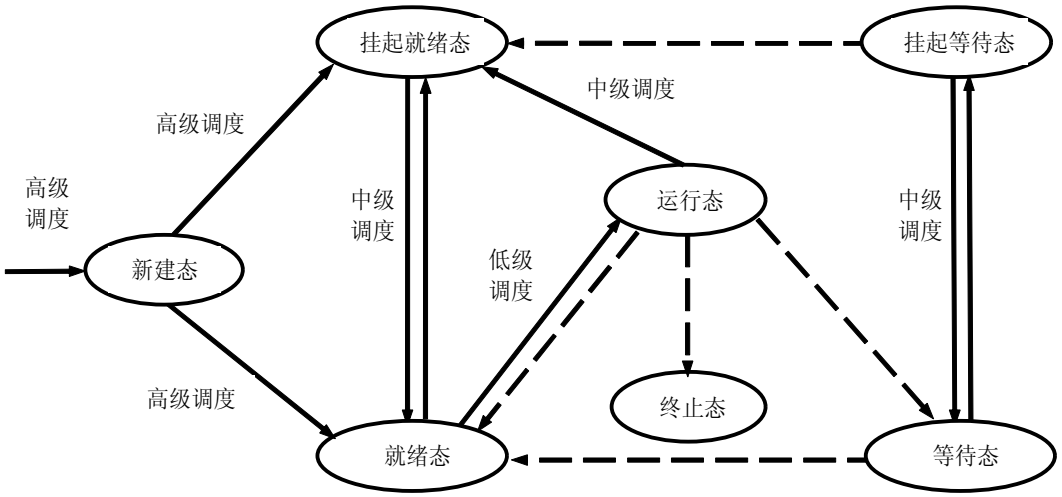


图 2-39　调度的层次

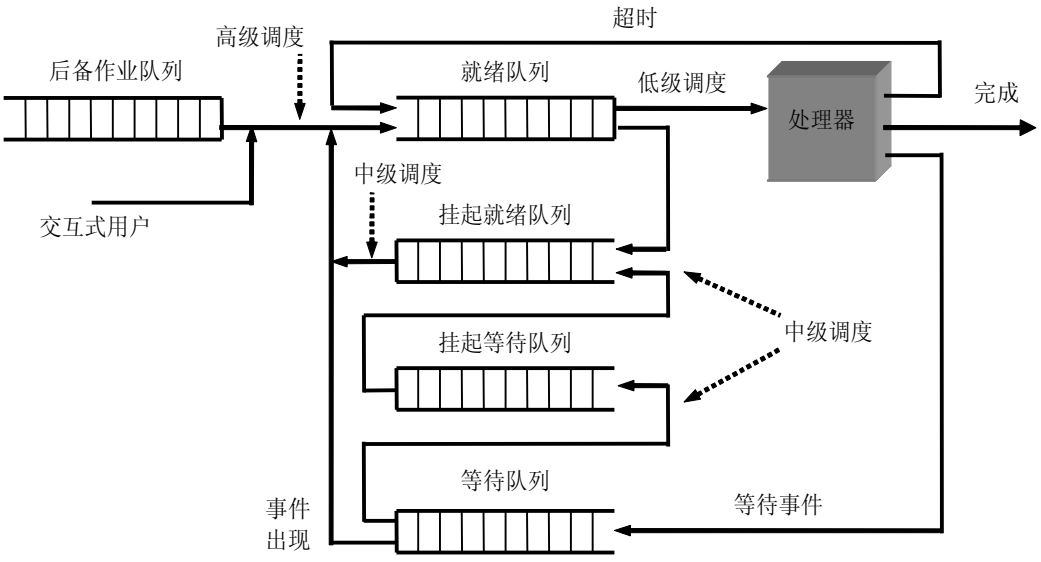
在三个层次的处理机调度中，所有操作系统必须配备进程调度。图 2-40 给出了三级调度功能与进程状态转换的关系。长程调度发生在新进程的创建中，它决定一个进程能否被创建，或者是创建后能否被置成就绪状态，以参与竞争处理器资源获得运行；中级调度反映到进程状态上就是挂起和解除挂起，它根据系统的当前负荷情况决定停留在主存中进程数；低级调度则是决定哪一个就绪进程区占有 CPU 运行。

图 2-40 处理器调度与进程状态转换



2.5.2 高级调度

对于分时操作系统来说，如图 2-41 所示，高级调度决定：1）是否接受一个终端用户的连接；2）一个程序能否被计算机系统接纳并构成进程；3）一个新建态的进程是否能够加入就绪进程队列。有的分时



操作系统虽没有配置高级调度程序，但上述的调度功能是必须提供的。

图 2-41 批处理操作系统的调度模型

在多道批处理操作系统中，作业是用户要求计算机系统完成的一项相对独立的工作。高级调度的功能是按照某种原则从后备作业队列中选取作业进入主存，并为作业做好运行前的准备工作和作业完成后的善后工作。当然一个程序能否被计算机系统接纳并构成可运行进程也作高级调度的一项任务。图 2-41 给出了批处理操作系统的调度模型。

2.5.3 中级调度

很多操作系统为了提高内存利用率和作业吞吐量，专门引进了中级调度。中级调度决定那些进程被允许参与竞争处理器资源，起到短期调整系统负荷的作用。它所使用的方法是通过把一些进程换出主存，从而使之进入“挂起”状态，不参与进程调度，起到滑系统的作用。有关中级调度的详细内容参见第三章中有关“进程挂起”小节。

2.5.4 低级调度

低级调度又称进程调度、短程调度。它的主要功能是按照某种原则把处理器分配给就绪进程或内核级线程。进程调度程序是操作系统最为核心的部分，进程调度策略的优劣直接影响到整个系统的性能。

2.5.5 选择调度算法的原则

无论是哪一个层次的处理机调度，都由操作系统的调度程序（scheduler）实施，而调度程序所使用的算法称为调度算法（scheduling algorithm）。不同类型的操作系统，其调度算法通常不同。

在讨论具体的调度算法之前，首先讨论一下调度所要达到的目标。设计调度程序首先要考虑的是确定策略，然后才是提供机制。一个好的调度算法应该考虑很多方面，其中可能有：

- 资源利用率——使得 CPU 或其他资源的使用率尽可能高且能够并行工作。
- 响应时间——使交互式用户的响应时间尽可能短，或尽快处理实时任务。
- 周转时间——批处理用户从作业提交给系统开始到作业完成获得结果为止这段时间间隔称作业周转时间，应该使作业周转时间或作业平均周转时间尽可能短。
- 吞吐量——使得单位时间处理的作业数尽可能多。
- 公平性——确保每个用户每个进程获得合理的 CPU 份额或其他资源份额。

当然，这些目标本身就存在着矛盾之处，操作系统在设计时必须根据其类型的不同进行权衡，以达到较好的效果。下面着重看一下批处理系统的调度性能。

批处理系统的调度性能主要用作业周转时间和作业带权周转时间来衡量。如果作业 i 提交给系统的时刻是 t_s，完成时刻是 t_f，那么该作业的周转时间 t_i 为：

$$t_i = t_f - t_s$$

实际上，它是作业在系统里的等待时间与运行时间之和。

从操作系统来说，为了提高系统的性能，要让若干个用户的平均作业周转时间和平均带权周转时间最小。

$$\text{平均作业周转时间 } T = (\Sigma t_i) / n$$

如果作业 i 的周转时间为 t_i，所需运行时间为 t_k，则称 $w_i=t_i/t_k$ 为该作业的带权周转时间。因为，t_i 是等待时间与运行时间之和，故带权周转时间总大于 1。

$$\text{平均作业带权周转时间 } W = (\Sigma w_i) / n$$

通常，用平均作业周转时间来衡量对同一作业流施行不同作业调度算法时，它们呈现的调度性能；用平均作业带权周转时间来衡量对不同作业流施行同一作业调度算法时，它们呈现的调度性能。这两个数值均越小越好。

2.6 批处理作业的管理与调度

2.6.1 作业和进程的关系

作业(JOB)是用户提交给操作系统计算的一个独立任务。一般每个作业必须经过若干个相对独立又相互关连的``顺序加工步骤才能得到结果，其中，每一个加工步骤称一个作业步(Job Step)，例如，一个作业可分成”编译””连结装配”和”运行”三个作业步，往往上一个作业步的输出是下一个作业步的输入。作业由用户组织，作业步由用户指定，一个作业从提交给系统，直到运行结束获得结果，要经过提交、收容、执行和完成四个阶段。当收容态作业被作业调度程序选中进入主存并投入运行时，操作系统将为此用户作业生成相应用户根进程或第一个作业步进程。进程是对系统中已提交完毕的任务(程序)的执行过程，它在”运行””就绪””等待”等多个状态的交替之中，在 CPU 上推进，最终完成一个程序的任务。用户根进程或作业步进程在执行过程中可生成作业步子进程，当然子进程还可以生成其它的子进程，这些进程并发执行，高效地协作完成用户作业的任务。在多道程序设计环境中，由于多个作业可同时被投入运行，因此，并发系统中，某一时刻并发执行的进程是相当多的(如 Unix 中有几十个)，操作系统要负责众多进程的协调和管理。后面将要引进线程概念，一个进程中还可以孵化出多个线程，由线程并发执行来完成作业任务。

综上所述，可以看出作业和进程之间的主要关系：作业是任务实体，进程是完成任务的执行实体;没

有作业任务，进程无事可干，没有进程，作业任务没法完成。作业概念更多地用在批处理操作系统，而进程则可以用在各种多道程序设计系统。

2.6.2 批处理作业的管理

多道批处理操作系统采用脱机控制方式，它提供一个作业控制语言，用户使用作业控制语言书写作业说明书，它是按规定格式书写的一个文件,把用户对系统的各种请求和对作业的控制要求集中描述,并与程序和数据一起提交给系统(管理员)。计算机系统成批接受用户作业输入，把它们放到输入井，然后在操作系统的管理和控制下执行。

多道批处理操作系统具有独立的作业管理模块，为了有效管理作业，必须像进程管理一样为每一个作业建立作业控制块（JCB）。JCB 通常是在批作业进入系统时，由 Spooling 系统建立的，它是作业存在于系统的标志，作业撤离时，JCB 也被撤销。JCB 的主要内容从用户作业说明书中获得，包括：作业情况(用户名、作业名、语言名等),资源需求(估计 CPU 运行时间、最迟截止期、主存量、设备类型/台数、文件数和数据量、函数库/实用程序等),资源使用情况(进入系统时间、开始运行时间、已运行时间等),作业控制(优先数、控制方式、操作顺序、出错处理等)。

当一个作业被操作系统接受，就必须创建一个作业控制块，并且这个作业在它的整个生命周期中将顺序地处于以下四个状态：

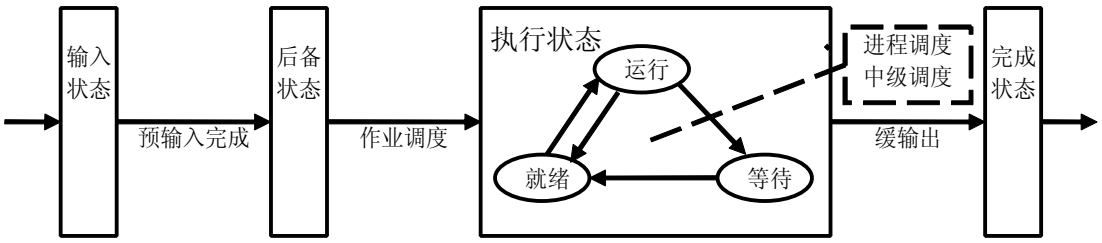
- 输入状态：此时作业的信息正在从输入设备上预输入。
- 后备状态：此时作业预输入结束但尚未被选中执行。
- 执行状态：作业已经被选中并构成进程去竞争处理器资源以获得运行。
- 完成状态：作业已经运行结束，正在等待缓输出。

多道批处理操作系统的处理机调度至少应该包括作业调度和进程调度两个层次。作业调度属于高级调度层次，处于后备状态的作业在系统资源满足的前提下可以被选中从而进入内存计算。而只有处于执行状态的作业才真正构成进程获得计算的机会。

作业调度选中了一个作业且把它装入主存储器时就为该作业创建一个用户进程。这些进程将在进程调度的控制下占有处理器运行。为了充分利用处理器，往往可以把多个作业同时装入主存储器，这样就会同时有多个用户进程，这些进程都要竞争处理器。

所以，进入计算机系统的作业只有经过两级调度后才能占用处理器。第一级是作业调度，使作业进入主存储器；第二级是处理器调度，使作业进程占用处理器。作业调度与处理器调度的配合能实现多道作业的同时执行，作业调度与进程调度的关系如图 2-42。

图 2-42　作业调度与进程调度的关系



2.6.3 批处理作业的调度

对成批进入系统的用户作业，按一定的策略选取若干个作业使它们可以去获得处理器运行，这项工作称为作业调度。

上面已经说过，对于每个用户来说总希望自己的作业的周转时间 T_i 尽可能的小，最理想的情况是进入系统后能立即投入运行，即希望 T_i 等于作业执行的时间。对于系统来说，则希望进入系统的作业的平均周转时间：$T=(T_1+T_2+\ldots+T_n)/n$ 尽可能的小。

于是，每个计算机系统都必须选择适当的作业调度算法，既考虑用户的要求又要有利于系统效率的提高。

2.6.4 作业调度算法

2.6.4.1 先来先服务算法

先来先服务(First Come,First Served)算法是按照作业进入系统的先后次序来挑选作业，先进入系统的作业优先被挑选。这种算法容易实现，但效率不高，只顾及到作业等候时间，而没考虑作业要求服务时间的长短。显然这不利于短作业而优待了长作业，或者说有利于 CPU 繁忙型作业不利于 I/O 繁忙型作业。有时为了等待长作业的执行，而使短作业的周转时间变得很大。从而平均周转时间也变大。

例如，下面三个作业同时到达系统并立即进入调度：

作业名	所需 CPU 时间
作业 1	28
作业 2	9
作业 3	3

假设系统中没有其他作业，现采用 FCFS 算法进行调度，那么，三个作业的周转时间分别为：28、37 和 40，因此，

平均作业周转时间 T = (28+37+40)/3 = 35

若这三个作业提交顺序改为作业 2、1、3，平均作业周转时间缩短为约 29。如果三个作业提交顺序改为作业 3、2、1，则平均作业周转时间缩短为约 18。由此可以看出，FCFS 调度算法的平均作业周转时间与作业提交的顺序有关。

2.6.4.2 最短作业优先算法

最短作业优先(Shortest Job First)算法是以进入系统的作业所要求的 CPU 时间为标准，总是选取估计计算时间最短的作业投入运行。这一算法也易于实现，但效率也不高，它的主要弱点是忽视了作业等待时间。由于系统不断地接受新作业，而作业调度又总是选择计算时间短的作业投入运行，因此，使进入系统时间早但计算时间长的作业等待时间过长，会出现饥饿的现象。

例如，若有以下四个作业同时到达系统并立即进入调度：

作业名	所需 CPU 时间
作业 1	9
作业 2	4
作业 3	10
作业 4	8

假设系统中没有其他作业，现对它们实施 SJF 调度算法，这时的作业调度顺序为作业 2、4、1、3，

平均作业周转时间 T = (4+12+21+31)/4 = 17

平均带权作业周转时间 W = (4/4+12/8+21/9+31/10)/4 = 1.98

如果对它们施行 FCFS 调度算法，

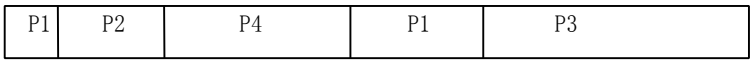
平均作业周转时间 T = (9+13+23+31)/4 = 19

平均带权作业周转时间 W = (9/9+13/4+23/10+31/8)/4 = 2.51

由此可见，SJF 的平均作业周转时间比 FCFS 要小，故它的调度性能比 FCFS 好。但实现 SJF 调度算法需要知道作业所需运行时间，否则调度就没有依据，作业运行时间只知道估计值，要精确知道一个作业的运行时间是办不到的。

SJF 算法既可以是非抢占式的，也可以是抢占式的。当一个作业正在执行时，一个新作业进入就绪状态，如果新作业需要的 CPU 时间比当前正在执行的作业剩余下来还需的 CPU 时间短，抢占式短作业优先算法强行赶走当前正在执行作业,这种方式也叫最短剩余时间优先算法(Shortest Remaining Time First)算法。此算法不但适用于 JOB 调度，同样也适用于进程调度。下面来看一个例子,假如现有四个就绪作业其到达系统和所需 CPU 时间如下：

作业名	到达系统时间	CPU 时间(毫秒)
Job1	0	8
Job2	1	4
Job3	2	9
Job4	3	5



0 1 5 10 17 26

Job1 从 0 开始执行，这时系统就绪队列仅有一个作业。Job2 在时间 1 到达，而 Job1 的剩余时间(7 毫秒)大于 JOB2 所需时间(4 毫秒),所以,Job1 被剥夺,Job2 被调度执行。这个例子的平均等待时间是((10-1)+(1-1)+(17-2)+(5-3))/4=26/4=6.5 毫秒。如果采用非抢占式 SJF 调度，那么，平均等待时间是 7.75 毫秒。

2.6.4.3 响应比最高者优先(HRN)算法

先来服务算法与最短作业优先算法都是比较片面的调度算法。先来先服务算法只考虑作业的等候时间而忽视了作业的计算时间，而最短作业优先算法恰好与之相反，它只考虑用户估计的作业计算时间而忽视了作业的等待时间。响应比最高者优先算法是介乎这两种算法之间的一种折衷的算法，既考虑作业等待时间，又考虑作业的运行时间，这样既照顾了短作业又不使长作业的等待时间过长，改进了调度性能。我们把作业进入系统后的等待时间与估计运行时间之比称作响应比，现定义：

响应比 ＝已等待时间/估计计算时间

显然，计算时间短的作业容易得到较高的响应比，因为,这时分母较小，使得 HRN 较高,因此本算法是优待短作业的。但是，如果一个长作业在系统中等待的时间足够长后，由于，分子足够大，使得 HRN 较大,那么它也将获得足够高的响应比，从而可以被选中执行，不至于长时间地等待下去，饥饿的现象 不会发生。

例如，若有以下四个作业先后到达系统进入调度：

作业名	到达时间	所需 CPU 时间
作业 1	0	20
作业 2	5	15
作业 3	10	5
作业 4	15	10

假设系统中没有其他作业，现对它们实施 SJF 调度算法，这时的作业调度顺序为作业 1、3、4、2,

平均作业周转时间 T = (20+15+20+45)/4 = 25

平均带权作业周转时间 W = (20/20+15/5+25/10+45/15)/4 = 2.25

如果对它们施行 FCFS 调度算法，

平均作业周转时间 T = (20+30+30+35)/4 = 38.75

平均带权作业周转时间 W = (20/20+30/15+30/5+35/10)/4 = 3.13

如果对这个作业流执行 HRN 调度算法，

- 开始时只有作业 1，作业 1 被选中，执行时间 20；
- 作业 1 执行完毕后，响应比依次为 15/15、10/5、5/10，作业 3 被选中，执行时间 5；
- 作业 3 执行完毕后，响应比依次为 20/15、10/10，作业 2 被选中，执行时间 15；
- 作业 2 执行完毕后，作业 4 被选中，执行时间 10；

平均作业周转时间 T = (20+15+35+35)/4 = 26.25

平均带权作业周转时间 W = (20/20+15/5+35/15+35/10)/4 = 2.42

2.6.4.4 优先数法

这种算法是根据确定的优先数来选取作业，每次总是选择优先数高的作业。规定用户作业优先数的方法是多种多样的。一种是由用户自己提出作业的优先数。有的用户为了自己的作业尽快的被系统选中就设法提高自己作业的优先数，这时系统可以规定优先数越高则需付出的计算机使用费就越多，以作限制。另一种是由系统综合考虑有关因素来确定用户作业的优先数。例如，根据作业的缓急程度；作业的类型；作业计算时间的长短、等待时间的多少、资源申请情况等来确定优先数。确定优先数时各因素的比例应根据系统设计目标来分析这些因素在系统中的地位而决定。上述确定优先数的方法称静态优先数法；如果在作业运行过程中，根据实际情况和作业发生的事件动态的改变其优先数，这称之为动态优先数法。

2.6.4.5 分类调度算法

分类调度算法预先按一定的原则把作业划分成若干类，以达到均衡使用操作系统资源和兼顾大小作业的目的。分类原则包括作业计算时间、对内存的需求、对外围设备的需求等。作业调度时还可以为每类作业设置优先级，从而照顾到同类作业中的轻重缓急。

2.6.4.6 用磁带与不用磁带的作业搭配

这种算法将需要使用磁带机的作业分开来。在作业调度时，把使用磁带机的作业和不使用磁带机的作业搭配挑选。在不使用磁带机的作业执行时，可预先通知操作员将下一批作业要用的磁带预先装上，这样可使要用磁带机的作业在执行时省去等待装磁带的时间。显然这对缩短系统的平均周转时间是有益的。

2.7 进程调度

2.7.1 进程调度的功能

进程调度负责动态地把处理器分配给进程或内核级线程。因此，它又叫处理器调度或低级调度。操作系统中实现进程调度的程序称为进程调度程序，或分派程序(Dispatcher)。进程调度算法多数适用于线程调度,下面着重介绍进程调度。那么，何时发生 CPU 调度呢?有四种情况都会发生 CPU 调度:当一个进程从运行态切换成等待态时; 当一个进程从运行态切换成就绪态时; 当一个进程从等待态切换成就绪态时和当一个进程中止时。进程调度的主要功能是：

- 记住进程的状态。这个信息一般记录在一个进程的进程控制块内。

- 决定某个进程什么时候获得处理器，以及占用多长时间。
- 把处理器分配给进程。即进行进程上下文切换，把选中进程的进程控制块内有关现场的信息；如程序状态字，通用寄存器等内容送入处理器相应的寄存器中，从而让它占用处理器运行。
- 收回处理器。将处理器有关寄存器内容送入该进程的进程控制块内的相应单元，从而使该进程让出处理器。

进程调度有两种基本方式:非抢占式和抢占式。非抢占式进程调度方式中,一旦某个高优先级的进程上有了 CPU，它就一直运行下去，直到其自身原因(结束或等待)主动出让 CPU 时,才调度另一个高优先级进程运行。抢占式进程调度方式中,任何时刻严格按高优先级进程在 CPU 上运行的原则进行进程/线程调度,每当一个高优先级进程运行期间,系统中又有更高优先级进程就绪，进程调度将迫使当前运行进程出让 CPU。前者实现简单、管理方便,批处理系统中用得多;后者能缩短紧迫作业响应时间,分时和实时系统中用得很多。可以把两种基本方式组合起来,形成折衷方式，实现思路如下：把就绪进程分成不同优先权的几个队列，如按系统进程和用户进程分成高低优先级两队,队列内采用非抢占式，但队列间采用抢占式。仅当高优先级队列空时,才能调度低优先级队列;而低优先级队列进程运行时，高优先级队列中若有进程就绪,可以抢占处理器。

2.7.2 进程调度算法

处理器的调度策略很多；现介绍如下几种：

2.7.2.1 先来先服务算法

先来先服务算法是按照进程进入就绪队列的先后次序来分配处理器。先进入就绪队列的进程优先被挑选，运行进程一旦占有处理器将一直运行下去直到运行结束或阻塞。这种算法容易实现，但效率不高，显然不利于 I/O 频繁的进程。

2.7.2.2 时间片轮转调度

轮转法调度也称之为时间片调度，具体做法是调度程序每次把 CPU 分配给就绪队列首进程使用一个时间片，例如 100ms，就绪队列中的每个进程轮流地运行一个这样的时间片。当这个时间片结束时，就强迫一个进程让出处理器，让它排列到就绪队列的尾部，等候下一轮调度。实现这种调度要使用一个间隔时钟，例如，当一个进程开始运行时，就将时间片的值置入间隔时钟内，当发生间隔时钟中断时，就表明该进程连续运行的时间已超过一个规定的时间片。此时，中断处理程序就通知处理器调度进行处理器的切换工作。这种调度策略可以防止那些很少使用外围设备的进程过长的占用处理器而使得要使用外围设备的那些进程没有机会去启动外围设备。

最常用的轮转法是基本轮转法。它要求每个进程轮流地运行相同的一个时间片。在分时系统中，这是一种较简单又有效的调度策略。一个分时系统有许多终端设备，终端用户在各自的终端设备上同时使用计算机，如果某个终端用户的程序长时间的占用处理器，势必使其它终端用户的要求不能得到及时响应。一般说分时系统的终端用户提出要求后到计算机响应给出回答的时间只能是几秒钟，这样才能使终端用户感到满意。采用基本轮转的调度策略可以使系统及时响应。例如，一个分时系统有 10 个终端，如果每个终端用户进程的时间片为 100ms，那么，粗略地说，每个终端用户在每秒钟内可以得到大数 100ms 的处理器时间，如果对于终端用户的每个要求，处理器花费 300ms 的时间就可以给出回答时，那么终端响应的时间大致就在 3 秒左右，这样可算得上及时响应了。

基本轮转法的策略可以略加修改。例如，对于不同的进程给以不同的时间片；时间片的长短可以动态地修改等等，这些做法主要是为了进一步提高效率。

轮转法调度是一种剥夺式调度，系统耗费在进程切换上的开销比较大，这个开销与时间片的大小很有关系。如果时间片取值太小，以致于大多数进程都不可能在一个时间片内运行完毕，切换就会频繁，系统开销显著增大，所以，从系统效率来看，时间片取大一点好。另一方面，时间片长度较大，那么随着就绪队列里进程数目的增加，轮转一次的总时间增大，亦即对每个进程的响应速度放慢了。为了满足用户对响应时间的要求，要么限制就绪队列中的进程数量，要么采用动态时间片法，根据当前负载状况，及时调整时间片的大小。所以，时间片大小的确定要从进程个数、切换开销、系统效率和响应时间等方面考虑。

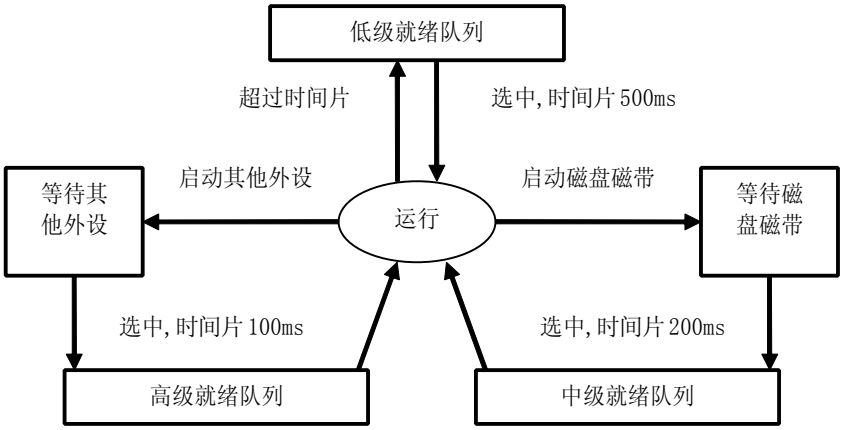
2.7.2.3 优先权调度

每一个进程给出一个优先数，处理器调度每次选择就绪进程中优先数最大者，让它占用处理器运行。怎样确定优先数呢?可以有以下几种考虑，使用外围设备频繁者优先数大，这样有利于提高效率；重要算题程序的进程优先数大，这样有利于用户；进入计算机时间长的进程优先数大，这样有利于缩短作业完成的时间；交互式用户的进程优先数大，这样有利于终端用户的响应时间等等,以上采用了静态优先数法。效率高性能好的进程调度可采用动静态优先数法,基本原则是：①根据进程占有 CPU 时间多少来决定,当一个进程占有 CPU 时间愈长,那么，在它被阻塞之后再次获得调度的优先级就越低，反之,进程获得调度的可能性越大;②根据进程等待 CPU 时间多少来决定,当一个进程在就绪队列中等待时间愈长,那么，在它被阻塞之后再次获得调度的优先级就越高，反之,进程获得调度的可能性越小。

2.7.2.4 多级反馈队列调度

这种方法又称反馈循环队列或多队列策略。其主要思想是将就绪进程分为两级或多级，系统相应建立两个或多个就绪进程队列，较高优先级的队列一般分配给较短的时间片。处理器调度每次先从高级的就绪进程队列中选取可占有处理器的进程，只有在选不到时，才从较低级的就绪进程队列中选取。

图 2-44　一个三级调度策略



进程的分级可以事先规定，例如使用外围设备频繁者属于高级。在分时系统中可以将终端用户进程定为高级，而非终端用户进程为低级。进程分级也可以事先不规定，例如，凡是运行超越时间片后，就进入低级就绪队列，以后给较长的时间片；凡是运行中启动磁盘或磁带而成为等待的进程，在结束等待后就进入中级就绪队列等，这种调度策略如图 2-44 所示。多级反馈队列调度算法具有较好的性能，能满足各类用户的需要。对分时交互型短作业，系统通常可在第一队列规定的时间片内让其完成工作，使终端型用户都感到满意；对短的批处理作业，通常，只需在第一或第一、第二队列中各执行一个时间片就能完成工作，周转时间仍然很短；对长的批处理作业，它将依次在第一、第二、．．．队列中获得时间片并运行，决不会出现得不到处理的情况。

2.7.2.5 保证调度算法

一种完全不同的调度算法是向用户做出明确的性能保证，然后去实现它。一种很实际并很容易实现的保证是：当你工作时已有 n 个用户登录在系统，则你将获得 CPU 处理能力的 1/n。类似的，如果在一个有 n 个进程运行的用户系统中，每个进程将获得 CPU 处理能力的 1/n。

为了实现所作的保证，系统必须跟踪各个进程自创建以来已经使用了多少 CPU 时间。然后它计算各个进程应获得的 CPU 时间，即自创建以来的时间除以 n。由于各个进程实际获得的 CPU 时间已知，所以很容易计算出实际获得的 CPU 时间和应获得的 CPU 时间之比，于是调度将转向比率最低的进程。

2.7.2.6 彩票调度算法

尽管向用户做出承诺并履行它是一个好主意，但实现却很困难。不过有另一种可以给出类似的可预见结果，而且实现起来简单许多，这种算法称为彩票调度算法。

其基本思想是：为进程发放针对系统各种资源（如 CPU 时间）的彩票。当调度程序需要做出决策时，随机选择一张彩票，持有该彩票的进程将获得系统资源。对于 CPU 调度，系统可能每秒钟抽 50 次彩票，每次中奖者可以获得 20ms 的运行时间。

在此种情况下，所有的进程都是平等的，它们有相同的运行机会。如果某些进程需要更多的机会，就可以被给予更多的额外彩票，以增加其中奖机会。如果发出 100 张彩票，某一个进程拥有 20 张，它就有 20%的中奖概率，它也将获得大约 20%的 CPU 时间。

彩票调度与优先级调度完全不同，后者很难说明优先级为 40 到底意味着什么，而前者则很清楚，进程拥有多少彩票份额，它将获得多少资源。

彩票调度法有几点有趣的特性。彩票调度的反映非常迅速，例如，如果一个新进程创建并得到了一些彩票，则在下次抽奖时，它中奖的机会就立即与其持有的彩票成正比。

如果愿意的话，合作的进程可以交换彩票。例如，一个客户进程向服务器进程发送一条消息并阻塞，它可以把所持有的彩票全部交给服务器进程，以增加后者下一次被选中运行的机会；当服务器进程完成响应服务后，它又将彩票交还给客户进程使其能够再次运行；实际上，在没有客户时，服务器进程根本不需要彩票。

彩票调度还可以用来解决其他算法难以解决的问题。例如，一个视频服务器，其中有若干个在不同的视频下将视频信息传送给各自的客户，假设它分别需要 10、20 和 25 帧/秒的传输速度，则分别给这些进程分配 10、20 和 25 张彩票，它们将自动按照正确的比率分配 CPU 资源。

2.7.3 实时调度

2.7.3.1 实时操作系统的特性

实时系统是那些时间因素非常关键的系统。例如，计算机的一个或多个外设发出信号，计算机必须在一段固定时间内做出适当的反应。一个实例是，计算机用 CD-ROM 放 VCD 时，从驱动器中获得的二进制数据必须在很短时间转化成视频和音频信号，如果转换的时间太长，图像显示和声音都会失真。其他的实时系统还包括监控系统、自动驾驶系统、安全控制系统等等，在这些系统中，迟到的响应即使正确，也和没有响应一样糟糕。

实时系统通常分为硬实时（hard real time）系统和软实时（soft real time）系统。前者意味着存在必须满足的时间限制；后者意味着偶尔超过时间限制时可以容忍的。这两种系统中，实时性的获得时通过将程序分成很多进程，而每个进程的行为都预先可知，这些进程处理周期通常都很短，往往在一秒钟内就运行结束，当检测到一个外部事件时，调度程序按满足他们最后期限的方式调度这些进程。

实时系统要响应的事件可以进一步划分为周期性（每个一段固定的时间发生）事件和非周期性（在不可预测的时间发生）事件。一个系统可能必须响应多个周期的事件流，根据每个事件需要的处理时间，系统可能根本来不及处理所有事件。例如，有 m 个周期性事件，事件 i 的周期为 P_i，其中每个事件需要C_i秒的 CPU 时间来处理，则只有满足以下条件：

$$C_1/P_1 + C_2/P_2 + \cdots + C_m/P_m \leqslant 1$$

时，才可能处理所有的负载。满足该条件的实时系统称作可时刻调度的（ schedulable）。

举例来说，一个软实时系统处理三个事件流，其周期分别为 100ms，200ms 和 500ms，如果事件处理时间分别为 50ms，30ms 和 100ms，则这个系统是可调度的，因为

$$0.5 + 0.15 + 0.2 \leqslant 1$$

如果加入周期为 1 秒的第 4 个事件，则只要其处理时间不超过 150ms，该系统仍将时可调度的。当然，这个运算的隐含条件是进程切换的时间足够小，可以忽略。

尽管在理论上采取了下面将要讨论的实时调度算法后就可以把一个通用操作系统改造成实时操作系统，但实际上，通用操作系统的进程切换开销太大，以至于只能满足那些时间限制较松的应用的实时性能要求。这就导致多数实时系统使用专用的实时操作系统。这些系统具有一些很重要的特征，典型的包括：规模小、中断时间很短、进程切换很快、中断被屏蔽的时间很短，以及能够管理毫秒或微秒级的多个定时器。

2.7.3.2 实时调度算法

实时调度算法可以分为动态实时调度算法和静态实时调度算法两类。动态实时调度算法在运行时作出调度决定，静态实时调度算法在系统启动之前完成所有的调度决策。下面来介绍几种经典的实时调度算法。

1）单比率调度算法

单比率调度事先为每个进程分配一个与事件发生频率成正比的优先数。例如，周期为 20ms 的进程优先数为 50，周期为 100ms 的进程优先数为 10，运行时调度程序总是调度优先数最高的就绪进程，并采取抢占式分配策略。可以证明概算法是最优的。

2）限期调度算法

限期调度算法的基本思想是：当一个事件发生时，对应的进程就被加入就绪进程队列。该就绪队列按照截止期限排序，对于一个周期性事件，其截止期限即为事件下一次发生的时间。该调度算法首先运行队首进程，即截止时间最近的那个进程。

3）最少裕度法

最少裕度法的基本思想是：首先计算各个进程的富裕时间，即裕度（ laxity），然后选择裕度最少的进程执行。

2.7.4 多处理器调度

一些计算机系统包括多个处理器，目前应用较多的、较为流行的多处理器系统有：

- 松散耦合多处理器系统：如 cluster，它包括一组独立的处理器，每个处理其拥有自己的主存和 I/O 通道。
- 紧密耦合多处理器系统：它包括一组处理器，共享主存和外设。

因此操作系统的调度程序必须考虑多粒处理器的调度。显然，单个处理器的调度和多粒处理器的调度有一定的区别，现代操作系统往往采用进程调度与线程调度相结合的方式来完成多处理器调度。

2.7.4.1 同步的粒度

同步的粒度，就是系统中多个进程之间同步的频率，它是刻画多处理系统特征和描述进程并发度的一个重要指标。一般来说，我们可以根据进程或线程之间同步的周期（即每间隔多少条指令发生一次同步事件），把同步的粒度划分成以下 5 个层次：

- 细粒度（fine-grained）：同步周期小于 20 条指令。这是一类非常复杂的对并行操作的使用，类似于多指令并行执行。它属于超高并行度的应用，目前有很多不同的解决方案，本书将不涉及这些解决方案，有兴趣的可以参见有关资料。
- 中粒度（medium-grained）：同步周期为 20-200 条指令。此类应用适合用多线程技术实现，即一个进程包括多个线程，多线程并发或并行执行，以降低操作系统在切换和通信上的代价。
- 粗粒度（coarse-grained）：同步周期为 200-2000 条指令。此类应用可以用多进程并发程序设计来实现。
- 超粗粒度（very coarse-grained）：同步周期为 2000 条指令以上。由于进程之间的交互是非常不频繁，因此这一类应用可以在分布式环境中通过网络实现并发执行。
- 独立（independent）：进程或线程之间不存在同步。

对于那些具有独立并行性的进程来说，多处理器环境将得到比多道程序系统更快的响应。考虑到信息和文件的共享问题，在多数情况下，共享主存的多处理器系统将比那些需要通过分布式处理实现共享的多处理器系统的效率更高。

对于那些具有粗粒度和超粗粒度并行性的进程来说，并发进程可以得益于多处理器环境。如果进程之间交互不频繁的话，分布式系统就可以提供很好的支持，而对于进程之间交互频繁的情况，多处理器系统的效率更高。

无论是有独立并行性的进程，还是具有粗粒度和超粗粒度并行性的进程，在多处理器环境中的调度原则和多道程序系统并没有太大的区别。但在多处理器环境中，一个应用的多个线程之间交互非常频繁，针对一个线程的调度策略可能影响到整个应用的性能。因此在多处理器环境中，我们主要关注的是线程的调度。

2.7.4.2 多处理器调度的设计要点

多处理器调度的设计要点之一是如何把进程分配给处理器。我们假定在多处理器系统中所有的处理器都是相同的，即对主存和 I/O 设备的访问方式相同，那么所有的处理器可以被作为一个处理器池（pool）来对待。我们可以采取静态分配策略，把一个进程永久的分配给一个处理器，分配在进程创建时执行，每个处理器对应一个低级调度队列。这种策略调度代价较低，但容易造成在一些处理器忙碌时另一些处理器空闲。我们也可以采取动态分配策略，所有处理器共用一个就绪进程队列，当某一个处理器空闲时，就选择一个就绪进程占有该处理器运行，这样，一个进程就可以在任意时间在任意处理器上运行。对于紧密耦合的共享内存的多处理器系统来说，由于所有处理器的现场相同，因此采用此策略时进程调度实现较为方便，效率也较好。

无论采取哪一种分配策略，操作系统都必须提供一些机制来执行分配和调度，那么操作系统程序在多处理器系统中又是怎样分布呢？方法之一是采用主从式（master/slave）管理结构，操作系统的和形成运行在一个特殊的处理器上，其他处理器运行用户程序，当用户程序需要请求操作系统服务时，请求将被传递到主处理器上的操作系统程序。显然这种方式实现上较为简单，并且比多道程序系统的调度效率高，但也有两个缺点：1）整个系统的坚定性与在主处理器上运行的操作系统程序关系过大；2）主处理器极易成为系统性能的瓶颈。因此还可以采用分布式（peer-to-peer）管理结构，在此种管理结构下，操作系统程序可以在所有处理器上执行，每一个处理器也可以自我调度。这种方式虽然比较灵活，但实现比较复杂，操作系统程序本身也需要同步。作为前面两种方法的折衷，我们可以把操作系统内核程序组成成几部分，分别放在不同的处理器上。

多处理器调度的设计要点之二是是否要在单个处理器上支持多道程序设计。对于独立、超粗粒度和粗粒度并行性的进程来说，回答是肯定的。但是对于中粒度并行性的进程来说，答案这是不明朗的。当很多的处理器可用时，尽可能的使单个处理器繁忙已经是那么重要，系统要追求的可能是给应用提供最好的性能，事实上，一个带有大量线程的进程可能会一直运行下去。

多处理器调度的设计要点之三是如何指派进程。在单处理器的进程调度中我们讨论了很多复杂的调度算法，但是在多处理器环境中这些复杂的算法可能是不必要的、甚至难以达到预期目的，调度策略的目标是简单有效且实现代价低，线程的调度尤其是这样。

2.7.4.3 多处理器的调度算法

大量的实验数据证明，随着处理器数目的增多，复杂进程调度算法的有效性却逐步下降。因此在大多数采取动态分配策略的多处理器系统中，进程调度算法往往采用最简单的先来先服务算法或优先数算法，就绪进程组成一个队列或多个按照优先数排列的队列。

多处理器调度的主要研究对象是线程调度算法。线程概念的引进把执行流从进程中分离出来，同一进程的多个线程能够并发执行并且共享进程地址空间。尽管线程也给单处理器系统带来很大益处，但在多处理器环境中线程的作用才真正得到充分发挥。多个线程能够在多个处理器上并行执行，共享用户地址空间进行通信，线程切换的代价也远低于进程切换。多处理器环境中的线程调度是一个研究热点，下面讨论几种经典的调度算法。

1）负载共享调度算法

负载共享（load sharing）调度算法的基本思想是：进程并不分配给一个处理器，系统维护一个全局性就绪线程队列，当一个处理器空闲时，就选择一个就绪线程占有处理器运行。这一算法有如下优点：

- 把负载均分到所有的可用处理器上，保证了处理器效率的提高。
- 不需要一个集中的调度程序，一旦一个处理器空闲，操作系统的调度程序就可以运行在该处理器上以选择下一个运行的线程。
- 运行线程的选择可以采用各种可行的策略（雷同与前面介绍的各种进程调度算法）。

这一算法也有一些不足：

- 就绪线程队列必须被互斥访问，当系统包括很多处理器，并且同时有多个处理器同时挑选运行线程时，它将成为性能的瓶颈。
- 被抢占的线程很难在同一个处理器上恢复运行，因此当处理器带有高速缓存时，恢复高速缓存的信息会带来性能的下降。
- 如果所有的线程都被放在一个公共的线程池中的话，所有的线程获得处理器的机会是相同的。如果一个程序的线程希望获得较高的优先级，进程切换将导致性能的折衷。

尽管有这样一些缺点，均分负载调度算法依然是多处理器系统最常用的线程调度算法。如著名的 mach 操作系统，它包括一个全局共享的就绪线程队列，并且每一个处理器还对应于一个局部的就绪线程队列，其中包括了一些临时绑定到该处理器上的就绪线程，处理器调度时首先在局部就绪线程队列中选择绑定线程，如没有，才到全局就绪线程队列中选择未绑定线程。

2）群调度算法

群调度（gang scheduling）算法的基本思想是：把一组相关的线程在同一时间一次性调度到一组处理器上运行。它具有以下的优点：

- 当紧密相关的进程同时执行时，同步造成的等待将减少，进程切换也相应减少，系统性能自然得到提高。
- 由于一次性同时调度一组处理器，调度的代价也将减少。

从上面两个优点来看，群调度算法针对多线程并行执行的单个应用来说具有较好的效率，因此它被广泛应用在支持细粒度和中粒度并行的多处理器系统中。

3）处理器专派调度算法

处理器专派（dedicated processor assignment）调度算法的基本思想是：给一个应用专门指派一组处理器，一旦一个应用被调度，它的每一个线程被分配一个处理器并一直占有这个处理器运行直到整个应用运行结束。采用这一算法之后，这些处理器将不适用多道程序设计，即该应用的一个线程阻塞后，该线程对应的处理器不会被调度给其他线程，而将处于空闲状态。

显然，这一调度算法追求的是通过高度并行来达到最快的执行速度，它在应用进程的整个生命周期避免进程调度和切换，且毫不考虑处理器的使用效率。对于高度并行的计算机系统来说，可能包括几十或数百个处理器，它们完全可以不考虑单个处理器的使用效率，而集中关注于提高计算效率。处理器专派调度算法适用于此类系统的调度。

最后值得指出的是，无论从理论上还是从实践中都可以证明，任何一个应用任务，并不是划分的越细，使用的处理器越多，它的求解速度就越快。在多处理器并行计算环境中，任何一种算法的加速比提高是有上限的。

4）动态调度算法

在实际应用中，一些应用提供了语言或工具以允许动态地改变进程中的线程数，这就要求操作系统能够调整负载以提高可用性。

动态调度（dynamic scheduling）算法的基本思想是由操作系统和应用进程共同完成调度。操作系统负责在应用进程之间划分处理器。应用进程在分配给它的处理器上执行可运行线程的子集，哪一些线程应该执行，哪一些线程应该挂起完全是应用进程自己的事（当然系统可能提供一组缺省的运行库例程）。相当的应用将得益于操作系统的这一特征时，但一些单线程进程则不适用这一算法。

在这一算法中，当一个进程达到或要求新的处理器时，操作系统的调度程序主要限制处理器的分配，并且按照下面的步骤处理：

- 如果有空闲的处理器，满足要求。
- 否则，对于新到达进程，这从当前分配了一个以上处理器的进程手中收回一个，并把它分配给新到达进程。
- 如果一部分要求不能被满足，则保留申请直到出现可用的处理器或要求取消。
- 当释放了一个或多个处理器后，扫描申请处理器的进程队列，按照先来先服务的原则把处理器逐一分配给每个申请进程直到没有可用处理器。

2.7.5 实例研究——传统 Unix 调度算法

本节讨论 Unix SVR3 和 Unix BSD 4.3 的进程调度。这些系统的主要设计目标是提供交互式的分时操作环境，因此调度算法在保证低优先级的后台作业不饿死的前提下，尽可能向交互式用户提供快速的响应。

传统 Unix 的进程调度采用多级反馈队列，对于每一个优先级队列采用时间片调度策略。系统遵从 1

秒抢占的原则，即一个进程运行了 1 秒还没有阻塞或完成的话，它将被抢占。优先数根据进程类型和执行情况确定，计算公式如下：

$$P_j(i) = Base_j + CPU_j(i-1)/2 + nice_j$$

$$CPU_j(i) = U_j(i)/2 + CPU_j(i-1)/2$$

其中：

- $P_j(i)$：进程 j 的优先数，时间间隔为 i；取值越小，优先级越高
- $Base_j$：进程 j 的基本优先数
- $U_j(i)$：进程 j 在时间间隔 i 内的处理器使用情况
- $CPU_j(i)$：进程 j 在时间间隔 i 内的处理器使用情况的指数加权平均数
- $nice_j$：用户控制的调节因子

每个进程的优先数每秒计算一次，并随后做出调度决定。基础优先数用作把所有进程划分到固定的优先级队列中，CPU_j 和 $nice_j$ 受到限制以防止进程迁移出分配给它的由基础优先数确定的队列。这些队列用作优化访问块设备或允许操作系统快速响应系统调用。根据有利于 I/O 设备有效使用的原则，进程队列按优先级从高到低排列有：

- 对换
- 块设备控制
- 文件操纵
- 字符设备控制
- 用户进程

2.7.6 实例研究——Unix SVR4 调度算法

Unix SVR4 的调度算法同传统 Unix 相比有了较大变动，其设计目的是优先考虑实时进程，次优先考虑内核模式进程，最后考虑用户模式进程。Unix SVR4 对调度算法的主要修改包括：

- 提供了基于静态优先数的抢占式调度，包括 3 类优先级层次，160 个优先数。
- 引入了抢占点。由于 Unix 的基本内核不是抢占式的，它将被划分划分成一些处理步骤，如果不发生中断的话，这些处理步骤将一直运行直到结束。在这些处理步骤之间，存在着抢占点，称为 safe place，此时内核可以安全地中断处理过程并调度新进程。每个 safe place 被定义成临界区，从而保证内核数据结构通过信号量上锁并被一致地修改。

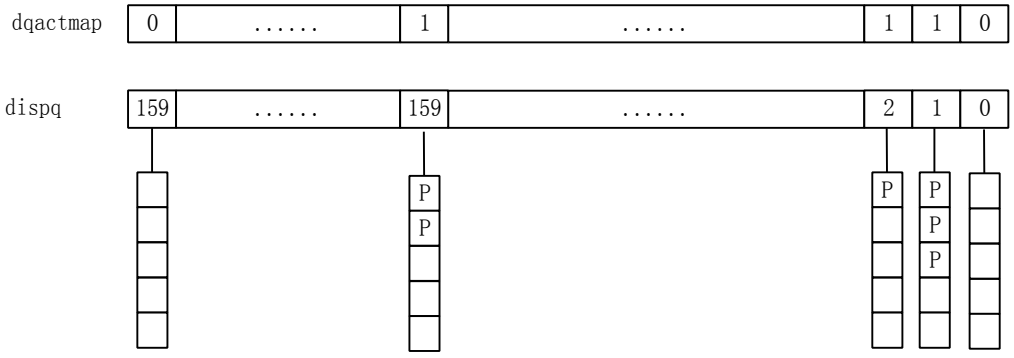
在 Unix SVR4 中，每一个进程必须被分配一个优先数，从而属于一类优先级层次。优先级和优先数的划分如下：

- 实时优先级层次（优先数为 159-100）：这一优先级层次的进程先于内核优先级层次和分时优先级层次的进程运行，并能利用抢占点抢占内核进程和用户进程。
- 内核优先级层次（优先数为 99-60）：这一优先级层次的进程先于分时优先级层次进程但迟于实时优先级层次进程运行。
- 分时优先级层次（优先数为 59-0）：最低的优先级层次，一般用于非实时的用户应用。

Unix SVR4 的进程调度参见图 2-45，它事实上还是一个多级反馈队列，每一个优先数都对应于一个就绪进程队列，而每一个进程队列中的进程按照时间片方式调度。位向量 dqactmap 用来标志每一个优先数就绪进程队列是否为空。当一个运行进程由于阻塞、时间片或抢占让出处理器，调度程序首先查找 dqactmap 已发现一个较高优先级的非空队列，然后指派进程占有处理器运行。另外，当执行到一个定义的抢占点，内核程序将检查一个叫作 kprunrun 的标志位，如果发现有高优先级的实时进程处于就绪状态，就执行抢占。

图 2-45　Unix SVR4 的就绪进程队列

对于分时优先级层次，进程的优先数是可变的，当运行进程用完了时间片，调度程序将降低它的优



先数，而当运行进程阻塞后，调度程序则将提高它的优先数。不同优先数的进程所获得的时间片也是不

同的，从 0 优先数的 100ms 到 59 优先数的 10ms。实时进程的优先数和时间片都是固定的。

2.7.7 实例研究——Windows NT 调度算法

Windows NT 的设计目标有两个，一是向单个用户提供交互式的计算环境，二是支持各种服务器（server）程序。

Windows NT 的调度是基于内核级线程的，它支持抢占式调度，包括多个优先数层次，在某些层次线程的优先数是固定的，在另一些层次线程的优先数将根据执行的情况动态调整。它的调度策略是一个多级反馈队列，每一个优先数都对应于一个就绪队列，而每一个进程队列中的进程按照时间片方式调度。

如图 2-46 所示，Windows NT 有两个优先级层次：

- 实时优先级层次（优先数为 31-16）：用于通信任务和实时任务。当一个线程被赋予一个实时优先数，在执行过程中这一优先数是不可变的。NT 支持优先数驱动的抢占式调度，一旦一个就绪线程的实时优先数比运行线程高，它将抢占处理器运行。
- 可变优先级层次（优先数为 15-0）：用于用户提交的交互式任务。具有这一层次优先数的线程，可以根据执行过程中的具体情况动态地调整优先数，但是 15 这个优先数是不能被突破的。

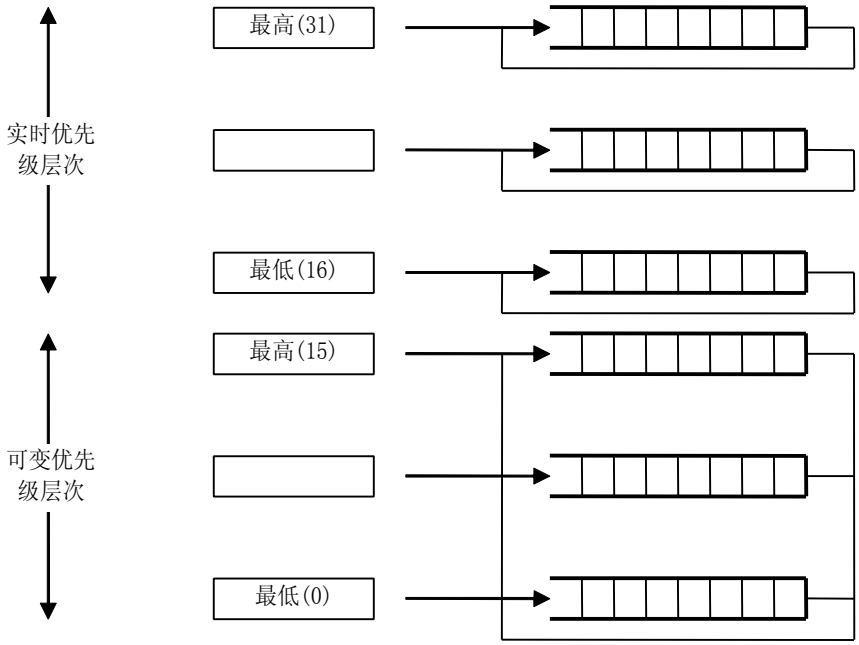


图 2-46　Windows NT 的线程优先级

一个线程如果被赋予可变优先数，那么它的优先数调整服从下面规则：

- 线程所属的进程对象有一个进程基本优先数，取值范围从 0 到 15。
- 线程对象有一个线程基本优先数，取值范围从-2 到 2。
- 线程的初始优先数为进程基本优先数加上线程基本优先数，但必须在 0 到 15 的范围内。
- 线程的动态优先数必须在初始优先数到 15 的范围内。

当运行线程用完了时间片，调度程序将降低它的优先数，而当运行进程因为 I/O 阻塞后，调度程序则将提高它的优先数。并且优先数的提高与等待的 I/O 设备有关，等待交互式外围设备（如键盘和显示器）时，优先数的提高幅度大于等待其他类型的外围设备（如磁盘），显然这种优先数调整方式有利于交互式任务。

当 NT 运行在单个处理器上时，最高优先级的线程将运行直到它结束、阻塞或被抢占。而当 NT 运行在 N 个处理器上时，N-1 个处理器上将运行 N-1 个最高优先级的线程，其他线程将共享剩下的一个处理器。值得指出的是，当线程的处理器亲和属性（processor affinity attribute）指定了线程执行的处理器时，虽然系统有可用的处理器，但指定处理器被更高优先级的线程占用，此时该线程必须等待，而可用处理器将被调度给优先级较低的就绪线程。

习题

1.　什么是 PSW?其主要作用是什么?

2. 当从具备运行条件的程序中选取一道后，怎样才能让它占有处理器工作?
3. 为什么是现代计算机要设置两个或多个 CPU 状态。
4. 为什么要把机器指令分成特权指令和非特权指令?
5. 硬件如何发现中断事件?发现中断事件后应做什么操作?
6. 从中断事件性质来说，可把它分成哪些类型?
7. 在处理程序性中断时，什么情况下，可转用户中断续元处理?
8. 叙述中断处理程序应完成的操作。
9. 何谓中断的优先级?为什么要对中断事件分级?
10. 简述程序性中断的处理过程。
11. 为什么中断续元处理可以嵌套，但不能递归?
12. 简述程序调用的一般执行过程。
13. 执行系统调用和正常的对程序调用有何区别?
14. 叙述中断屏蔽的作用。
15. OS 如何处理多个中断事件。
16. 解释 WINDOWS200 的中断、异常和陷阱。
17. 解释 WINDOWS200 中，如何动态地实现中断屏蔽功能?
18. 下列指令中哪些只能在核心态运行?
    (1)读时钟日期；(2)访管；(3)设时钟日期；(4)加载 PSW；(5)置特殊寄存器
19. 什么是进程?计算机操作系统中为什么引入进程概念?其意义何在?
20. 进程有哪些主要属性?试解释之。
21. 进程最基本的状态有哪些?哪些事件可能引起不同状态之间的转换?
22. 五态模型的进程中，新建态和终止态的主要作用是什么?
23. 试说明引起创建一个进程的主要事件?
24. 什么是进程的挂起状态?列出挂起进程的主要特征。
25. 什么情况下会产生：挂起等待态和挂起就绪态，试举例说明。
26. 叙述组成进程的基本要素，并说明它们的作用。
27. 何谓进程控制块(PCB)?它包含哪些基本信息?
28. 何谓进程队列、入队和出队?
29. 叙述创建进程系统所要做的主要工作。
30. 什么叫进程切换?叙述进程切换的主要步骤。
31. 进程切换主要应该保存哪些处理器状态?
32. 叙述进程唤醒的主要步骤。
33. 试说明引起撤销一个进程的主要事件。
34. 叙述系统撤销进程时应做的主要工作。
35. 简单解释进程管理的三种实现模型。
36. 小结 Unix SVR4 进程管理的特点。
37. 列举进程被阻塞和唤醒的主要事件。
38. 操作系统中引入进程概念后，为什么又引入线程概念?
39. 列举支持线程概念的商业性操作系统。
40. 试述多线程环境中，进程和线程的定义。
41. 试从资源分配单位和调度的基本单位两方面对进程和线程进行比较。
42. 试对下列系统任务作出比较：

a) 创建一个进程与创建一个线程
b) 两个进程间通信与同一进程中两个线程间通信
c) 同一进程中两个线程的上下文切换与不同进程中两个线程的上下文切换

43. 列举与线程状态变化有关的主要线程操作。
44. 叙述并发多线程程序设计的主要优点及其应用。
45. 什么是内核级线程、用户级线程和混合式线程?对它们进行比较。
46. 叙述 Solaris 中，设置了哪几种线程?轻量级进程的作用是什么?
47. 叙述 Solaris 中的进程与线程概念。
48. Solaris 中，设置了哪几种线程?轻量级进程的作用是什么?
49. Solaris 中，用户级线程通过什么方法来访问内核?
50. 叙述 Solaris 进程的数据结构。
51. 叙述 Solaris 用户级线程的数据结构。
52. 叙述 Solaris 用户级线程的状态及其转换原因。
53. 叙述 Windows 2000 中的进程和线程概念。
54. 试说明 Windows 2000 中的进程对象。
55. 试说明 Windows 2000 中的线程对象。
56. 叙述 Windows 2000 中的线程状态及其转换原因。
57. 试说明 Windows 2000 的作业对象。
58. 假定一个处理器正在执行两道作业，一道以计算为主，另一道以输入输出为主，你将怎样赋予它们占有处理器的优先级?为什么?
59. 有一个单向连接的进程队列，它的队首由系统队列标志提计指出，队列尾进程的队列指引元为 0。分别写出一个进程从队首、队中和队尾入队/出队的工作流程。
60. 采用双向连接的进程队列。假定队首和队尾进程的前(后)向指引元由队列标志指出；且队首(尾)的前(后)向指引元为‘0'，写出任一进程入队和出队的工作流程。
61. 假设有一种进程调度算法是让‘最近使用处理器较少的进程’运行，试解释这种算法对 I/O 繁重的作业有利，但也不会总是拒绝给予处理器繁重的进程处理器时间。
62. 设有三道程序，按 A、B、C 优先次序运行，其内部计算和 I/O 操作时间由图给出。

A	B	C
C_{11}＝30ms	C_{21}=60ms	C_{31}=20ms
|	|	|
I_{12}＝40ms	I_{22}=30ms	I_{32}=40ms
|	|	|
C_{13}＝10ms	C_{23}=10ms	C_{33}=20ms

试画出按开道运行的时间关系图（忽略调度执行时间）。完成三道程序共花多少时间?比单道运行节省了多少时间？若处理器调度程序每次进行程序状态转换化时 1ms，试画出各程序状态转换的时间关系图。

63. 试说明访管指令与特权指令的区别。
64. 试说明访管指令与系统调用的联系和区别。
65. 处理器调度分哪几种类型？简述各类调度的主要任务。
66. 叙述衡量一个处理器调度算法好坏的主要标准。
67. 叙述作业调度和进程调度的关系。
68. 解释：（1）作业周转时间；（2）作业带权周转时间。

69. 简述批处理作业的管理和调度过程。
70. 若有一组作业 J_1…，J_n，其执行时间依次为 S_1，…S_n 。如果这些作业同时到达系统中，并在一台单 CPU 处理机上按单道方式执行。试找出一种作业调度算法，使得平均作业周转时间最短。
71. 试叙述作业，进程和程序三者的关系。
72. 何谓响应比最高优先算法？它有何主要特点？
73. 时间片轮转进程调度算法中，根据哪些因素确定时间片长短？
74. 为什么多级反馈队列算法能较好地满足各种用户的需求？
75. 分析静态优先级和动态优先级调度算法各自的优缺点。
76. 试述剥夺式和非剥夺式调度策略的主要区别。
77. 假定执行表中所列作业，作业号即为到达顺序，且已全部进入系统。
    1） 分别用先来先服务调度算法、时间片轮转算法、短作业优先算法及非强占优先权调度算法算出各作业的执行先后次序；
    2） 计算每个作业的周转时间；
    3） 对所有作业而言，哪种调度算法具有最小平均等待时间。

作业号	执行时间	优先权
1	8	3
2	1	1
3	2	3
4	1	4
5	5	2

78. 对某系统进行监测后表明平均每个进程在 I/O 阻塞之前的运行时间为 T。一次进程切换的系统开销时间为 S。若采用时间片长度为 Q 的时向片轮转法，对下列各种情况算出 CPU 利用率。
    1）Q＝α　2）Q＞T　3）S＜Q＜T　4）Q＝S　5）Q 接近于 0
79. 有 5 个待运行的作业，各自预计运行时间分别是：9、6、3、5 和 x，采用哪种运行次序使得平均响应时间最短？
80. 有 5 个批处理作业 A 到 E 均已到达计算中心；地运行时间分别 10、6、2、4 和 8 分钟；各自的优先级分别被规定为 3、5、2、1 和 4，这里 5 为最高级。对于 1）时间片轮转算法、2）优先数法、3）短作业优先算法、4）先来先服务调度算法，在忽略进程切换时间的前提下，计算出平均作业周转时间。（对 1）每个作业获得相同的 CPU 时间片；对 2）到 4）采用单道运行，直到结束。）
81. 叙述典型的实时调度算法。
82. 试述与 CPU 调度算法的设计要点。
83. 列举并说明多处理器的经典线程调度算法
84. 算出并解释传统 UNIX 的动态优先数计算公式。
85. 叙述 UNIX SVR4 的处理器调度算法的主要特点。
86. 叙述 Windows NT 的处理器调度算法的主要等点。
87. 证明在非抢占式调度算法中，最短作业优先算法具有最小平均等待时间
88. 定义响应时间 R 是一个进程等待和服务的平均时间。证明对于先来先服务调度算法，R=S/(1-p)；其中，S 为平均服务时间，p 为处理机利用率。

89. 现有两类用户进程处理器繁忙或 I/O 繁忙型，哪类进程适合于多级反馈队列调度？说明理由。
90. 叙述高级调度与低级调度的主要任务。为什么要引入中级调度？
91. 在剥夺式调度中，有哪些剥夺原则？
92. 多级反馈队列中，给不同队列以大小不同的时间片值，意义何在？
93. 试论述先来先服务调度算法和时间片轮转算法的本质区别？
94. 单道批处理系统中，下列三个作业采用先来先服务调度算法和最高响应比优先算法进行调度，哪一种算法性能较好？请完成下表：

作业	提高时间	运行时间	开始时间	完成时间	周转时间	带权周转时间
1	10：00	2：00				
2	10：10	1：00				
3	10：25	0：25				
平均作业周转时间＝ 平均作业带权周转时间 W=						

CH3 并发进程

3.1 并发进程

3.1.1 顺序程序设计

进程执行的顺序性是指每个进程在顺序处理器上的执行是严格按序的，即只有当一个操作结束后，才能开始后继操作。

传统的程序设计方法是顺序程序设计(Sequential Programming)，即把一个程序设计成一个顺序执行的程序模块。顺序程序设计具有如下的特点：

- 执行的顺序性。一个程序在顺序处理器上的执行是严格按序的，即每个操作必须在下一个操作开始之前结束。
- 环境的封闭性。在顺序处理情况下，运行程序独占系统全部资源，除初始环境之外，其所处的环境都是由程序本身决定的，只有程序本身的动作才能改变其环境，不会受到任何其他程 序和外界因素的干扰。
- 执行结果的确定性。程序执行过程中允许出现中断，但这种中断对程序的最终结果没有影响，也就是说程序的执行结果与它的执行速率无关。
- 计算过程的可再现性。一个程序针对同一个数据集合一次执行的结果，在下一次执行时会重现。这样当程序中出现了错误时，往往可以重现错误，以便进行分析。

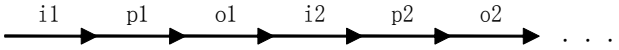
顺序程序设计的顺序性、封闭性、确定性和再现性给程序的编制、调试带来很大方便，其缺点是计算机系统效率不高。

3.1.2 进程的并发性

进程的并发性是指一组进程的在执行在时间上是重叠的。所谓执行在时间上是重叠的，是指执行一个进程的第一条指令是在执行另一个进程的最后一条指令完成之前开始。例如：有两个进程 A 和 B，它们分别执行操作 a1，a2，a3 和 b1。b2，b3。在一个单处理器上，就 A 和 B 两个进程而言，它们的执行顺序分别为 al，a2，a3 和 bl，b2，b3，这是进程执行的顺序性。然而，这两个进程在单处理器上可能是交叉执行，如执行序列为 al，bl，a2，b2，a3，b3 或 a1，bl，a2，b2，b3，a3 等，则说 A 和 B 两个进程的执行是并发的。

在采用多道程序设计的系统中，利用了外围设备与处理器，外围设备与外围设备的并行工作能力，从而提高了计算机的工作效率。怎样才能充分利用外围设备与处理器，外围设备与外围设备的并行能力呢?很重要的一个方面是取决于程序的编制。在图 3-1 的例子中，由于程序是按照 while(TRUE) { input，process，output }串行地输入-处理-输出的来编制的，所以这个程序只能顺序地执行。这时系统的效率是相当低的。

图 3-1　串行工作



如果把这个问题的程序分成三部分：

while(TRUE) { input，send }
while(TRUE) { receive，process，send }
while(TRUE) { receive，output }

每一部分称为一个程序模块，于是这三个程序模块的功能是：

模块 1：循环执行：读入字符，将读入字符送缓冲区 1。
模块 2：循环执行：处理缓冲区 1 中的字符，把计算结果送缓冲区 2。
模块 3：循环执行：取出缓冲区 2 中的计算结果并写到磁带上。

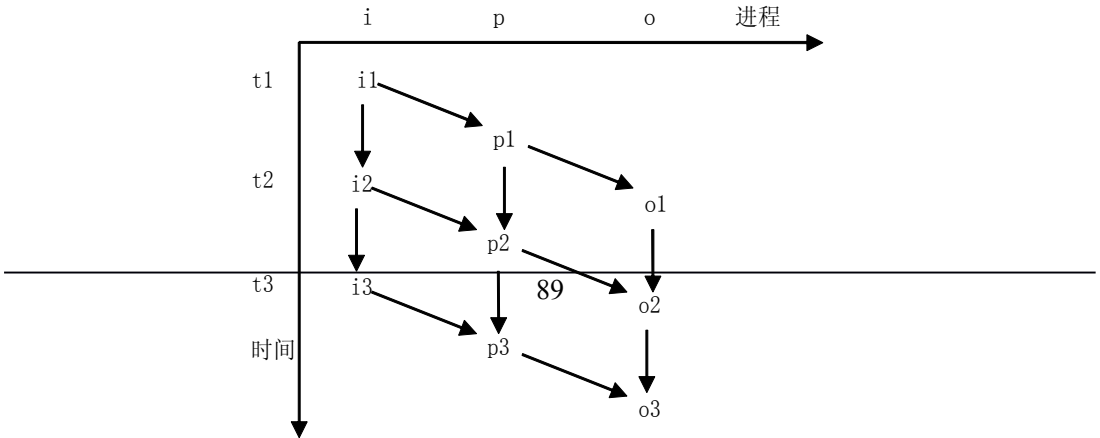


图 3-2 并行工作

从图 3-2 可以看出，这三个程序模块能同时执行，在 t3 时刻输入 i3、处理 p2 与输出 o1 可以并行工作，在 t4、t5 等时刻同样可以并行工作。于是就得到了图 3-3 所示的情形，假定循环的次数为 n，处理器的使用效率是：

$$（52 * n） /（78 * n十52十20）$$

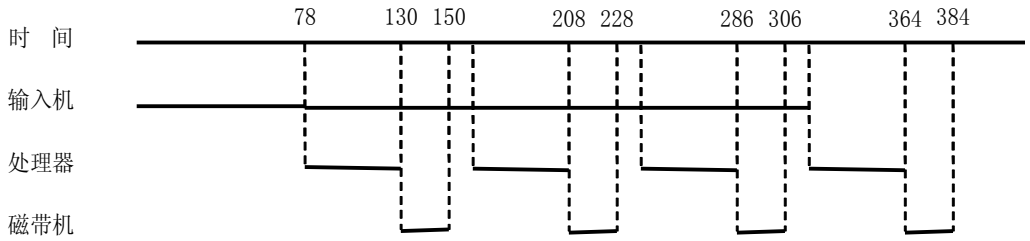


图 3-3 采用并发程序设计时处理器的使用效率

当 n 趋向于无穷大时，处理器的使用效率是 67%。显然这种程序设计方法发挥了处理器与外围设备的并行能力，从而提高了计算机的效率。这种使一个程序分成若干个可同时执行的程序模块的方法称并发程序设计(Concurrent Programming)，每个程序模块和它执行时所处理的数据就组成一个进程。

并发的进程可能是无关的，也可能是交往的。无关的并发进程是指它们分别在不同的变量集合上操作，所以一个进程的执行与其它并发进程的进展无关，即一个并发进程不会改变另一个并发进程的变量值。然而，交往的并发进程，它们共享某些变量，所以一个进程的执行可能影响其它进程的结果，因此，这种交往必须是有控制的，否则会出现不正确的结果。

并发进程的无关性是进程的执行与时间无关的一个充分条件。该条件在 1966 年首先由 Bernstein 提出，又称之为 Bernstein 条件。设 R(pi)={a1,a2,…an}，表示程序 pi 在执行期间引用的变量集；W(pi)={b1,b2,…bm}，表示程序 pi 在执行期间改变的变量集，若两个程序能满足 Bernstein 条件、即变量集交集之和为空集：

$$R(p1)\&W(p2) \cup R(p2)\&W(p1) \cup W(p1)\&W(p2)=\{\ \}$$

则并发进程的执行与时间无关。例如，有如下四条语句：

S1: a := x + y
S2: b := z + 1
S3: c := a - b
S4: w := c + 1

于是有：R(S1)={x,y} ,R(S2)={z}，R(S3)={a,b}，R(S4)={c}；W(S1)={a},W(S2)={b}，W(S3)={c}，W(S4)={w}。可见 S1 和 S2 可并发执行，因为，满足 Bernstein 条件。其他语句间因变量交集之和非空，并发执行可能会产生与时间有关的错误。

采用并发程序设计思想构造的一组程序模块在执行时，具有如下的特征：

- 并行性，它们的执行在时间上可以重迭，在单处理器系统中可以并发执行；在多处理器环境中可以并行执行。
- 共享性，它们可以共享某些变量，通过引用这些共享变量或使用通信机制可以互相交换信息，从而程序的运行环境不再是封闭的。
- 交往性，正因为它们具有共享性，所以一个程序的执行可能影响其它程序的执行结果，因此，这种交往必须是有控制的，否则会出现不正确的结果。即使在正确运行的前提下，程序结果也将可能是不确定的，计算过程具有不可再现性。

采用并发程序设计的好处是：一是若单处理器系统，可有效利用资源，让处理器和 I/O 设备，I/O 设备和 I/O 设备同时工作，充分发挥硬部件的并行能力；二是若多处理器系统，可让这些模块在不同处理器上物理地并行工作，从而加快计算速度；三是简化了程序设计任务，一般地说，编制并发执行的小程序模块进度快，容易保证正确性。

采用并发程序设计的目的是：充分发挥硬件的并行性，提高系统效率。硬件能并行工作仅仅有了提高效率的可能性，而硬部件并行性的实现还需要软件技术去利用和发挥。这种软件技术就是并发程序设计。并发程序设计是多道程序设计的基础，多道程序的实质就是把并发程序设计引入到系统中。

3.1.3 与时间有关的错误

两个交往的并发进程，其中一个进程对另一个进程的影响常常是不可预期的，甚至无法再现。这是因为两个并发进程执行的相对速度无法相互控制，交往进程的速率不仅受到处理器调度的影响，而且还受到与这两个交往的并发进程无关的其它进程的影响，所以一个进程的速率通常无法为另一个进程所知。因此，各种与时间有关的错误就可能出现，与时间有关的错误有两种表现形式，一种是结果不唯一；另

一种是永远等待。为了说明与时间有关的错误，现观察下面的例子。

例 1 (结果不唯一) 机票问题。

假设一个飞机订票系统有两个终端，分别运行进程 Tl 和 T2。该系统的公共数据区中的一些单元 Aj(j=l，2，…)分别存放某月某日某次航班的余票数，而 xl 和 x2 表示进程 T1 和 T2 执行时所用的工作单元。程序如下：

```
process Ti ( i = 1, 2 )
var Xi:integer;
begin
        {按旅客定票要求找到Aj}；
        Xi := Aj;
        if Xi>=1 then begin Xi:=Xi-1; Aj:=Xi;{输出一张票}; end
            else {输出票已售完}；
    end;
```

由于 T1 和 T2 是两个可同时运行的并发进程，它们在同一个计算机系统中运行，共享同一批票源数据，因此可能出现如下所示的运行情况。

```
T1: X1 := Aj;                          X1 = nn ( nn > 0 )
T2: X2 := Aj;                          X2 = nn
T2: X2:=X2-1; Aj:=X2;输出一张票；      Aj = nn-1
T1: X1:=X1-1; Aj:=X1;输出一张票；      Aj = nn-1
```

显然此时出现了把同一张票卖给了两个旅客的情况，两个旅客可能各自都买到一张同天同次航班的机票，可是，Aj 的值实际上只减去了 1，造成余票数的不正确。特别是，当某次航班只有一张余票时，就可能把这一张票同时售给了两位旅客，显然这是不能允许的。

例 2 (永远等待)内存管理问题。

假定有两个并发进程 borrow 和 return 分别负责申请和归还主存资源，算法描述中，x 表示现有空闲主存量，B 表示申请或归还的主存量。并发进程算法及执行描述如下：

```
procedure borrow (var B:integer)
  begin
      if B>x then   {申请进程进入等待队列等主存资源}
      x:=x-B;
      {修改主存分配表，申请进程获得主存资源}
  end;

procedure return (var B:integer)
  begin
      x:=x+B;
      {修改主存分配表}
      {释放等主存资源的进程}
   end;
cobegin
    Var x:integer;
     Repeat   borrow (var B:integer)
     Repeat   return(var B:integer)
coend
```

由于 borrow 和 return 共享了表示主存物理资源的临界变量 x，对并发执行不加限 会导致错误。例如，一个进程调用 borrow 申请主存,在执行了比较 B 和 x 的指令后, 发现 B>x, 但在执行{申请进程进入等待队列等主存资源}前, 另一个进程调用 return 抢先执行, 归还了所借全部主存资源. 这时, 由于前一个进程还未成为等待状态, return 中的{释放等主存资源的进程}相当于空操作. 以后当调用 borrow 的进程被置成等主存资源时, 可能已经没有其它进程来归还主存资源了, 从而, 申请资源的进程处于永远等待状态。

3.1.4 进程的交互(Interaction Among Processes)——协作和竞争

在多道程序设计系统中，同一时刻可能有许多进程，这些进程之间存在两种基本关系：竞争关系和协作关系。

第一种是竞争关系，系统中的多个进程之间彼此无关，它们并不知道其他进程的存在。例如，批处理系统中建立的多个用户进程，分时系统中建立的多个终端进程。由于这些进程共用了一套计算机系统资源，因而，必然要出现多个进程竞争资源的问题。当多个进程竞争共享硬设备、变量、表格、链表、文件等资源时，可能导致处理出错。

由于相互竞争资源时进程间并不交换信息，但是一个进程的执行可能影响到同其竞争的进程，如果两个进程要访问同一资源，那么，一个进程通过操作系统分配得到该资源，另一个将不得不等待。在极端的情况下，被阻塞进程永远得不到访问权，从而不能成功地终止。所以，资源竞争出现了两个控制问题：一个是死锁(Deadlock)问题，一组进程如果都获得了部分资源，还想要得到其他进程所占有的资源，最终所有的进程将陷入死锁。另一个是饥饿(Starvation) 问题，例如，三个进程 p1、p2、p3 均要周期性访问资源 R。若 p1 占有资源，p2 和 p3 等待资源。当 p1 离开临界区时，p3 获得了资源 R。如果 p3 退出临界区之前，p1 又申请并得到资源 R，如此重复造成 p2 老是得不到资源 R。尽管这里没有产生死锁，但出现了饥饿。对于临界资源，操作系统需要保证诸进程能互斥地访问这些资源，既要解决饥饿问题，又要解决死锁问题。

进程的互斥(Mutual Exclusion)是解决进程间竞争关系的手段。指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其它要使用该资源的进程必须等待，直到占有资源的进程释放该资源。

临界区管理可以解决进程互斥问题，本章第二节将详细介绍临界区的解决方案。

第二种是协作关系，某些进程为完成同一任务需要分工协作。我们在前面给出了一个例子， input 、process、和 output 三个进程分工协作完成读入数据、加工处理和打印输出任务，这是一种典型的协作关系，各自都知道对方的存在。这时操作系统要确保诸进程在执行次序上协调一致，没有输入完一块数据之前不能加工处理，没有加工处理完一块数据之前不能打印输出等等，每个进程都要接收到它进程完成一次处理的消息后，才能进行下一步工作。进程间的协作可以是双方不知道对方名字的间接协作，例如通过共享访问一个缓冲区进行松散式协作；也可以是双方知道对方名字，直接通过通信进行紧密协作。

进程的同步(Synchronization)是解决进程间协作关系的手段。指一个进程的执行依赖于另一个进程的消息，当一个进程没有得到来自于另一个进程的消息时则等待，直到消息到达才被唤醒。

不难看出，进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源。

3.2 临界区管理

3.2.1 互斥和临界区

例 1 中的售票管理系统之所以会产生错误，原因在于两个进程交叉访问了共享变量 Aj。我们把并发进程中与共享变量有关的程序段称为“临界区”(Critical Section) ， 共享变量代表的资源叫“临界资源”(Critical Resource)。在售票管理系统中，进程 T1 的临界区为：

```
X1 := Aj;
    if X1>=1 then begin X1:=X1-1; Aj:=X1;
```

进程 T2 的临界区为：

```
    X2 := Aj;
    if X2>=1 then begin X2:=X2-1; Aj:=X2;
```

与同一变量有关的临界区分散在各有关进程的程序段中，而各进程的执行速度不可预知。如果能保证一个进程在临界区执行时，不让另一个进程进入相关的临界区，即各进程对共享变量的访问是互斥的，那么就不会造成与时间有关的错误。

关于临界区的概念是由 Dijkstra 在 1965 年首先提出的。可以用与一个共享变量相关的临界区的语句结构来书写交往并发进程。我们用 shared 说明共享变量。

```
shared variable
region variable do statement
```

一个进程前进到一个临界区的语句时，不管该进程目前是否正在运行，都说它是在临界区内。根据它们的临界区重写的售票管理进程如下。

```
shared Aj
process Ti ( i = 1, 2 )
    var Xi:integer;
  begin
        按旅客定票要求找到Aj;
        region Aj do begin
```

```
        Xi := Aj;
            if Xi>=1 then begin Xi:=Xi-1; Aj:=Xi;输出一张票；end
                    else 输出票已售完；
        end;
    end;
```

对若干个进程共享一个变量的相关的临界区，有三个调度原则：

- 一次至多一个进程能够在它的临界区内；
- 不能让一个进程无限地留在它的临界区内；
- 不能强迫一个进程无限地等待进入它的临界区。特别，进入临界区的任一进程不能妨碍正等待进入的其它进程的进展；
- 我们可把临界区的调度原则总结成四句话：无空等待、有空让进、择一而入、算法可行。

临界区是允许嵌套的，例如

region x do begin … region y do … end

但是粗心的嵌套可能导致进程无限地留在它的临界区内，例如，如果又有一个进程执行：

region y do begin … region x do … end

这样，当两个进程在大约差不多的时间进入了外层的临界区后，将发现它们每个都被排斥在内层临界区之外，造成无限地等待进入临界区。

3.2.2 临界区管理的尝试

按前述要求实现临界区的管理，可采用一种标志的方式，即用标志来表示哪个进程可以进入临界区。然而如何使用标志，仍是值得讨论的问题。下面的程序是第一种尝试。对进程 P1 和 P2 分别用标志 insidel 和 inside2 与其相连，当该进程在它的临界区内时其值为真(true)，不在临界区时其值为假(false)。P1(P2)要进入它的临界区前先测试 inside2(insidel)以确保当前无进程在临界区，然后把 insidel(inside2)置成 true 以封锁进程 P2(P1)进入临界区，直至 P1(P2)退出临界区，再将相应标志置成 false。

```
inside1,inside2:boolean
inside1 := false;     /* P1 不在其临界区内 */
inside2 := false;     /* P2 不在其临界区内 */
cobegin
process P1
Begin
    while inside2 do begin end;
    inside1 := true;
    临界区;
    inside1 := false;
end;
process P2
begin
    while inside1 do begin end;
    inside2 = true;
    临界区;
    inside2 := false;
end;
coend
```

但是，这种管理是不正确的，原因是在 P1(P2)测试 inside2(insidel)与随后置 insidel(inside2)之间，并发进程 P2(P1)可能发现 insidel(inside2)有值 false，于是它将置 inside2(insidel)为 true，并且与 P1(P2)同时进入临界区。

第二种尝试是对第一种作如下修正：延迟 P1(P2)对 inside2(insidel)的测试，而先置 insidel(inside2)为

true 以封锁 P2(P1)。修正后的程序如下。不幸，它也是无效的。因为，有可能每个进程都把它的标志置成 true，从而出现死循环。这时，没有一个进程能在有限的时间内进入临界区。

```
inside1,inside2:boolean;
inside1 := false;     /* P1 不在其临界区内 */
inside2 := false;     /* P2 不在其临界区内 */
cobegin
process P1
begin
    inside1 := true;
    while inside2 do begin end;
    临界区;
    inside1 := false;
end;
process P2
begin
    inside2 := true;
    while inside1 do begin end;
    临界区;
    inside2 := false;
end;
coend
```

3.2.3 实现临界区管理的软件方法

3.2.3.1 Dekker算法

荷兰数学家 T.Dekker 算法能保证进程互斥地进入临界区，这是最早提出的一个不需轮换的软件互斥方法，此方法用一个指示器 turn 来指示应该哪一个进程进入临界区。若 turn=1 则进程 P1 可以进入临界区；若 turn=2 则进程 P2 可以进入临界区。Dekker 算法的程序如下：

```
var inside : array[1..2] of Boolean;
Turn :integer;
turn := 1 or 2;
inside[1]:=false;
inside[2]:=false;
cobegin
process P1
begin
    inside[1]:=true;
    while inside[2] do  if turn=2 then
                  begin
                   inside[1]:=false;
                   while turn=2 do begin end;
                   inside[1]:=true;
                  end
    临界区;
    turn = 2;
    inside[1]:=false;
end;
process P2
begin
     inside[2]:=true;
     while inside[1] do  if turn=1 then
                    begin
                     inside[2]:=false;
                     while turn=1 do begin end;
                     inside[2]:=true;
                    end
    临界区;
    turn = 1;
    inside[2]:=false;
end;
coend
```

这种方法显然能保证互斥进入临界区的要求，这是因为仅当 tum = i (i = 1，2)时进程 Pi (i = l，2)才能进入其临界区。因此，一次只有一个进程能进入临界区，且在一个进程退出临界区之前， turn 的值是不会改变的，保证不会有另一个进程进入相关临界区。同时，因为 turn 的值不是 1 就是 2，故不可能同时出现两个进程均在 while 语句中等待而进不了临界区。Dekker 虽能解决互斥问题,但算法复杂难于理解，peterson 解法较为简单。

3.2.3.2 Peterson算法

在 1981 年,G.L.Peteron 提出了一个简单得多的软件互斥算法来解决互斥进入临界区的问题。此方法为每个进程设置一个标志，当标志为 true 时表示该进程要求进入临界区。另外再设置一个指针 turn 以指示可以由哪个进程进入临界区，当 turn=i 时则可由进程 Pi 进入临界区。Petrson 算法的程序如下：

```
var inside:array[1..2] of boolean;
turn:integer;
turn := 1 or 2;
inside[1] := false;    /* P1 不在其临界区内 */
inside[2] := false;    /* P2 不在其临界区内 */
cobegin
```

```
process P1
begin
    inside[1]:= true;
    turn := 2;
    while (inside[2] and turn=2)
         do begin end;
    临界区;
    inside[1] := false;
end;
process P2
begin
    inside[2] := true;
    turn := 1;
    while (inside[1] and turn=1)
        do begin end;
    临界区;
    inside[2] := false;
end;
coend
```

在上面的程序中，用对 turn 的置值和 while 语句来限制每次最多只有一个进程可以进入临界区，当有进程在临界区执行时不会有另一个进程闯入临界区；进程执行完临界区程序后，修改 insidei 的状态而使等待进入临界区的进程可在有限的时间内进入临界区。所以，Peterson 算法满足了对临界区管理的三个条件。由于在 while 语句中的判别条件是“insidei 和 turni”，因此，任意一个进程进入临界区的条件是对方不在临界区或对方不请求进入临界区。于是，任何一个进程均可以多次进入临界区，克服了 Dekker 算法必须交替进入临界区的限制。

3.2.4 实现临界区管理的硬件设施

分析临界区管理的尝试中的两种算法，问题出在管理临界区的标志时要用到两条指令，而这两条指令在执行过程中有可能被中断，从而导致了执行的不正确。能否把标志看作为一个锁，开始时锁是打开的，在一个进程进入临界区时便把锁锁上以封锁其它进程进入临界区，直至它离开其临界区，再把锁打开以允许其它进程进入临界区。如果希望进入其临界区的一个进程发现锁未开，它将等待，直到锁被打开。可见，要进入临界区的每个进程必须首先测试锁是否打开，如果是打开的则应立即把它锁上，以排斥其它进程进入临界区。显然，测试和上锁这两个动作不能分开，以防两个或多个进程同时测试到允许进入临界区的状态。下面是一些硬件设施，可用来实现对临界区的管理。

3.2.4.1 关中断

实现这种管理的方法之一是关中断。当进入锁测试之前关闭中断，直到完成锁测试并上锁之后再开中断。在这一短暂的期间，计算机系统不响应中断，因此不会转向调度，也就不会引起进程或线程切换，从而保证了锁测试和上锁操作的连续性和完整性，有效的实现了临界区管理。但关中断时间过长会影响系统效率，关中断方法也不适用于多 CPU 系统。

3.2.4.2 测试并建立指令

实现这种管理的另一种办法是使用硬件提供的“测试并建立”指令 TS(Test and Set)。可把这条指令看作为函数过程，它有一个布尔参数 x 和一个返回条件码，当 TS(x)测到 x 为 true 时则置 x 为 false，且根据测试到的 x 值形成条件码。下面给出了 TS 指令的处理过程。

TS(x)： 若 x=true，则 x:=false; return true；否则 return false;

用 TS 指令管理临界区时，我们把一个临界区与一个布尔参数 s 相连，s 初值置为 true，表示没有进程在临界区内。在进入临界区之前，我们首先用 TS 指令测试 s，如果没有进程在临界区内，则可以进入，否则必须循环测试直到 TS（s）为 true，此时 s 的值一定为 false；当进程退出临界区时，把 s 置为 true。由于 TS 指令是一个不可分指令，所以在测试和形成条件码之间不可能有另一进程去测试 x 值，从而保证临界区管理的正确性。

```
s : boolean;
```

```
s := true;
process Pi          /* i = 1,2,...,n */
    pi : boolean;
begin
    repeat pi := TS(s) until pi;
    临界区;
    s := true;
end;
```

3.2.4.3 对换指令

对换（swap）指令的功能是交换两个字的内容，处理过程描述如下：

```
swap (a,b):  temp:=a; a:=b; b:=temp;
```

在 80x86 中，对换指令称为 XCHG 指令。用对换指令可以简单有效地实现互斥，方法是为每个临界区设置一个布尔变量，例如称为 lock，当其值为 false 时表示临界区未被使用，实现进程互斥的程序如下：

```
lock : boolean;
lock := false;
process Pi          /* i = 1,2,...,n */
    pi : boolean;
begin
    pi := true;
    repeat swap(lock, pi) until pi = false;
    临界区;
    lock := false;
end;
```

3.3 信号量与 PV 操作

3.3.1 同步和同步机制

下面通过例子来进一步阐明进程同步的概念。著名的生产者--消费者(Producer-Consumer Problem) 问题是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题。在操作系统中，生产者进程可以是计算进程、发送进程；而消费者进程可以是打印进程、接收进程等等。解决好了生产者--消费者问题就解决好了一类并发进程的同步问题。

生产者--消费者问题表述如下：有 n 个生产者和 m 个消费者，连接在一个有 k 个单位缓冲区的有界缓冲上，故又叫有界缓冲问题。其中，pi 和 cj 都是并发进程，只要缓冲区未满，生产者 pi 生产的产品就可投入缓冲区；类似地，只要缓冲区不空，消费者进程 cj 就可从缓冲区取走并消耗产品。

可以把生产者-消费者问题的算法描述如下：

```
var   k:integer;
      type item:any;
      buffer:array[0..k-1] of item;
      in,out:integer:=0;
      coumter:integer:=0;
process producer
      while (TRUE)                          /* 无限循环
      produce an item in nextp;             /* 生产一个产品
      if (counter==k)   sleep( );           /* 缓冲满时，生产者睡眠
      buffer[in]:=nextp;                    /* 将一个产品放入缓冲区
      in:=(in+1) mod k;                     /* 指针推进
      counter:=counter+1;                   /* 缓冲内产品数加 1
```

```
    if (counter==1)    wakeup( consumer);        /* 缓冲为空了，加进一件产品
                                                    并唤醒消费者
process consumer
    while (TRUE)                                 /* 无限循环
    if (counter==0) sleep ( );                   /* 缓冲区空，消费者睡眠
    nextc:=buffer[out];                          /* 取一个产品到 nextc
    out:=(out+1) mod k;                          /* 指针推进
    counter:=counter-1;                          /* 取走一个产品，计数减 1
    if (counter==k-1) wakeup( producer);         /* 缓冲满了，取走一件产品
                                                    并唤醒生产者
    consume thr item in nextc;                   /* 消耗产品
```

其中，假如一般的高级语言都有 sleep( ) 和 wakeup( ) 这样的系统调用。从上面的程序可以看出，算法是正确的，两进程顺序执行结果也正确。但若并发执行，就会出现错误结果，出错的根子在于进程之间共享了变量 counter，对 counter 的访问未加限制。

生产者和消费者进程对 counter 的交替执行会使其结果不唯一。例如，counter 当前值为 8，如果生产者生产了一件产品，投入缓冲区，拟做 conuter 加 1 操作。同时消费者获取一个产品消费，拟做 conuter 减 1 操作。假如两者交替执行加或减 1 操作，取决于它们的进行速度，counter 的值可能是 9，也可能是7，正确值应为 8。

更为严重的是生产者和消费者进程的交替执行会导致进程永远等待，造成系统死锁。假定消费者读取 conuter 发现它为 0。此时调度程序暂停消费者让生产者运行，生产者加入一个产品，将 counter 加 1，现在 counter 等于 1 了。它想当然地推想由于 counter 刚刚为 0，所以，此时消费者一定在睡眠，于是生产者调用 wakeup 来唤醒消费者。不幸的是，消费者还未去睡觉，唤醒信号被丢失掉。当消费者下次运行时，因已测到 counter 为 0，于是去睡眠。这样生产者迟早会添满缓冲区，然后去睡觉，形成了进程都永远处于睡眠状态。

出现不正确结果不是因为并发进程共享了缓冲区，而是因为它们访问缓冲区的速率不匹配，或者说 pi，cj 的相对速度不协调，需要调整并发进程的进行速度。并发进程间的这种制约关系称进程同步，交往的并发进程之间通过交换信号或消息来达到调整相互速率，保证进程协调运行的目的。

操作系统实现进程同步的机制称同步机制，它通常由同步原语组成。不同的同步机制采用不同的同步方法，迄今已设计出许多种同步机制，本书中将介绍几种最常用的同步机制：信号量及 PV，管程和消息传递。

3.3.2 记录型信号量与 PV 操作

前一节介绍的种种方法虽能保证互斥，可正确解决临界区调度问题，但有明显缺点。对不能进入临界区的进程，采用忙式等待(busy waiting)测试法，浪费 CPU 时间。将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重了用户编程负担。

1965 年荷兰的计算机科学家 E.W.Dijkstra 提出了新的同步工具--信号量和 P、V 操作。他将交通管制中多种颜色的信号灯管理交通的方法引入操作系统，让两个或多个进程通过信号量(Semaphore) 展开交互。进程在某一特殊点上停止执行直到得到一个对应的信号量，通过信号量这一设施，任何复杂的进程交互要求可得到满足，这种特殊的变量就是信号量。在操作系统中，信号量用以表示物理资源的实体，它是一个与队列有关的整型变量。实现时，信号量是一种变量类型，常常用一个记录型数据结构表示，它有两个分项：一个是信号量的值，另一个是信号量队列的队列指针。

信号量仅能由同步原语对其进行操作，原语是操作系统中执行时不可中断的过程、即原子操作(Atomic Action)。Dijkstra 发明了两个同步原语：P 操作和 V 操作(荷兰语中“测试(Proberen)” 和“增量(Verhogen)” 的头字母，此外还用的符号有：wait 和 signal；up 和 down；sleep 和 wakeup 等。本书中采用 Dijkstra 最早论文中使用的符号 P 和 V。利用信号量和 P、V 操作既可以解决并发进程的竞争问题，又可以解决并发进程的协作问题。

信号量按其用途可分为两种：

- 公用信号量：联系一组并发进程，相关的进程均可在此信号量上执行 P 和 V 操作。初值常常为 1，用于实现进程互斥。
- 私有信号量：联系一组并发进程，仅允许此信号量拥有的进程执行 P 操作，而其他相关进程可在其上施行 V 操作。初值常常为 0 或正整数，多用于并发进程同步。

信号量按其取值可分为两种：

- 二元信号量：仅允许取值为 0 和 1，主要用于解决进程互斥问题。
- 一般信号量：允许取值为非负整数，主要用于解决进程间的同步问题。

下面讨论整形信号量、记录型信号量和二元信号量。

1、整型信号量

设 s 为一个正整形量，除初始化外，仅能通过 P、V 操作来访问它，这时 P 操作原语和 V 操作原语

定义如下：。

- P(s)：当信号量 s 大于 0 时，把信号量 s 减去 1，否则调用 P(s)的进程等待直到信号量 s 大于 0 时。
- V(s)：把信号量 s 加 1。

P(s) 和 V(s) 可以写成：

P(s)： while s≤0 do null operation
s：=s-1；

V(s)： s：=s+1；

整型信号量机制中的 P 操作，只要信号量 s<=0，就会不断测试，进程处于“忙式等待” 。后来对整型信号量进行了扩充，增加了一个等待 s 信号量所代表资源的等待进程的队列，以实现让权等待。这就是下面要介绍的记录型信号量机制。

2、记录型信号量

设 s 为一个记录型数据结构,其中一个分量为整形量 value,另一个分量为信号量队列 queue, 这时 P 操作原语和 V 操作原语的定义修改如下：

- P(s)：将信号量 s 减去 1，若结果小于 0，则调用 P(s)的进程被置成等待信号量 s 的状态。
- V(s)：将信号量 s 加 1，若结果不大于 0，则释放一个等待信号量 s 的进程。

记录型信号量和 P 操作、V 操作可表示成如下的数据结构和不可中断过程：

```
type  semaphore=record
      value:integer;
      queue: list of process;
      end
procedure  P(var s:semaphore);
begin
    s.value:= s.value - 1;          /* 把信号量减去 1 */
    if s.value< 0 then W(s.queue);/* 若信号量小于 0，则执行 P(s)的进程调用
                                     W(s.queue) 进行自我封锁，被置成等待
                                     信号量 s 的状态，进入信号量队 queue*/
end;

procedure  V(var s:semaphore);
begin
    s.value:= s.value + 1;          /* 把信号量加 1 */
    if s.value≤ 0 then R(s.queue); /* 若信号量小于等于 0,则调用 R(s.queue)
                                     从信号量 s 队列 queue 中释放一个等待信
                                     号量 s 的进程并置成就绪态*/
end;
```

其中 W(s.queue)表示把调用过程的进程置成等待信号量 s 的状态, 并链入 s 信号量队列，同时 CPU 获得释放；R(s.queue)表示释放一个等待信号量 s 的进程，从信号量 s 队列中移出一个进程, 置成就绪态并投入就绪队列。信号量 s 的初值可定义为 0，1 或其它整数，在系统初始化时确定。从信号量和 P、V 操作的定义可以获得如下推论：

推论 1：若信号量 s 为正值，则该值等于在封锁进程之前对信号量 s 可施行的 P 操作数、亦即等于 s 所代表的实际还可以使用的物理资源数。

推论 2：若信号量 s 为负值，则其绝对值等于登记排列在该信号量 s 队列之中等待的进程个数、亦即恰好等于对信号量 s 实施 P 操作而被封锁起来并进入信号量 s 队列的进程数。

推论 3：通常，P 操作意味着请求一个资源，V 操作意味着释放一个资源。在一定条件下，P 操作代表挂起进程操作，而 V 操作代表唤醒被挂起进程的操作。

3、二元信号量

设 s 为一个记录型数据结构,其中一个分量为 value,它仅能取值 0 和 1,另一个分量为信号量队列 queue, 这时我们把二元信号量上的 P、V 操作记为 BP 和 BV,BP 操作原语和 BV 操作原语的定义如下：

```
type binary semaphore=record
```

```
            value(0,1);
            queue: list of process
          end;
procedure BP(var s:semaphore);
            if s.value=1;
                then
                  s.value=0;
            else begin
                  w(s.queue);
              end;
procedure BV(var s:semaphore);
            if s.queue is empty;
                then
                  s.value=1;
            else begin
                   R(s.queue);
              End;
```

虽然二元信号量仅能取 0 和 1 值，但可以证明它与记录型信号量一样，有着同等的表达能力。下面我们来看一下，如何用二元信号量实现记录型信号量。

```
var s1: binary-semaphore;
    s2: binary-semaphory;
    c:integer;
```

其中，初始化为 s1=1,s2=0,c 被赋予记录型信号量 s 所需的值，那么，在记录型信号量 s 上的 PV 操作定义为：

```
P(s)                                V(s)
BP(s1);                              BP(s1);
c:=c-1;                             c:=c+1;
if c<0                              if c≤0
   then begin                           then BV(s2);
          BV(s1);                           else BV(s1);
          BP(s2);
        end
BV(s1);
```

3.3.3 用记录型信号量实现互斥

记录型信号量和 PV 操作可以用来解决进程互斥问题。与 TS 指令相比较，PV 操作也是用测试信号量的办法来决定是否能进入临界区，但不同的是 PV 操作只对信号量测试一次，而用 TS 指令则必须反复测试。用 PV 操作管理几个进程互斥进入临界区的一般形式如下：

```
Var  mutex: semaphore;
     mutex := 1;
   cobegin
       ……
       process Pi
           begin
                   ……
               P(mutex);
                   临界区；
                   V(mutex);
                   ……
           end;
       ……
   coend;
```

下面的程序用记录型信号量和 PV 操作解决了机票问题。

```
Var A : ARRAY[1..m] OF integer;
  mutex : semaphore;
  mutex:= 1;

cobegin
process Pi
    var Xi:integer;
begin
    L1:
    按旅客定票要求找到A[j];
    P(mutex)
    Xi := A[j];
    if Xi>=1
    then begin
       Xi:=Xi-1; A[j]:=Xi;
       V(mutex);输出一张票;
    end;
    else begin
       V(mutex);输出票已售完;
    end;
    goto L1;
 end;
coend;
```

```
Var A : ARRAY[1..m] OF integer;
   s : ARRAY[1..m] OF semaphore;
   s[j] := 1;
cobegin
process Pi
    var Xi:integer;
begin
    L1:
    按旅客定票要求找到A[j];
    P(s[j])
    Xi := A[j];
    if Xi>=1
    then begin
       Xi:=Xi-1; A[j]:=Xi;
       V(s[j]); 输出一张票;
    end;
    else begin
       V(s[j]);输出票已售完;
    end;
    goto L1;
end;
coend;
```

左面的程序引入一个信号量 mutex，用于管理票源数据，其初值为 1。假设进程 T1 首先调用 P 操作，则 P 操作过程把信号量 mutex 减 1，T1 进入临界区；此时，若进程 T2 也想进入临界区而调用 P 操作，那么 P 操作过程便阻塞 T2 并使它等待 mutex。当 T1 离开临界区时，它调用 V 操作，V 操作过程唤醒等待 mutex 的进程 T2；于是 T2 就可进入临界区执行。事实上，当进程 T1 和 T2 只有同时买一个航班的机票时才会发生与时间有关的错误，因此临界区应该是与 A[j]有关的，所以我们可以对左面的程序进行改进，引入一组信号量 s[j]，从而得到了右面的程序，不难看出，它提高了进程的并发程度。

要提醒注意的是：任何粗心地使用 PV 操作会违反临界区的管理要求。如忽略了 else 部分的 V 操作，将致使进程在临界区中判到条件不成立时无法退出临界区，而违反了对临界区的管理要求。

若有多个进程在等待进入临界区的队列中排队，当允许一个进程进入临界区时，应先唤醒哪一个进程进入临界区?一个使用 PV 操作的程序，如果它是正确的话，那么，这种唤醒应该是无选择的。所以在证明使用 PV 操作的程序的正确性时，必须证明进程按任意次序进入临界区都不影响程序的正确执行。

下面再来看一下使用互斥信号量和 PV 操作解决操作系统经典的**五个哲学家吃通心面问题**。有五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘于，每两人之间放一把叉子。每个哲学家思考、饥饿、然后吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。

在这道题目中，每一把叉子都是必须互斥使用的，因此应为每把叉子设置一个互斥信号量 S_i，初值均为 1。当一个哲学家吃通心面前必须获得自己左边和右边的两把叉子，即执行两个 P 操作，吃完通心面后必须放下叉子，即执行两个 V 操作。程序如下：

```
Var forki :array[0..4] of semaphore;
    forki := 1;

cobegin
process Pi              // i=0,1,2,3,4,
begin
    L1:
    思考;
```

```
        P(fork[i]);
        P(fork[i+1] mod 5);
        吃通心面;
        V(fork[i]);
        V(fork[i+1]);
        goto L1;
    end;
coend;
```

请大家注意，如果第五个哲学家先执行 P(fork[4])，再执行 P(fork[0])的话，就有可能出现每个哲学家举起右边一把叉子，却又在永远等待相邻哲学家手中的叉子的情况。有若干种办法可避免这类死锁：

- 至多允许四个哲学家同时吃；
- 奇数号先取左手边的叉子，偶数号先取右手边的叉子；
- 每个哲学家取到手边的两把叉子才吃，否则一把叉子也不取。

3.3.4 记录型信号量解决生产者-消费者问题

记录型信号量和 PV 操作不仅可以解决进程的互斥，而且更是实现进程同步的好办法。进程的同步是指一个进程的执行依赖于另一个进程的消息，当一个进程没有得到来自于另一个进程的消息时则等待，直到消息到达才被唤醒。

生产者和消费者问题就是一个典型的进程同步问题，出现不正确结果的原因在于它们访问缓冲器的速率。为了能使它们正确工作，生产者和消费者必须按一定的生产率和消费率来访问共享的缓冲器。用 PV 操作来解决生产者和消费者共享一个缓冲器的问题，可以使用两个信号量 sl 和 s2，它们的初值分别为 1 和 0，sl 指示能否向缓冲器内存放产品，s2 指示是否能从缓冲器内取产品。于是生产者和消费者问题的程序如下所示。

```
var B : integer;
    sput:semaphore;          /* 可以使用的空缓冲区数 */
    sget:semaphore;          /* 缓冲区内可以使用的产品数 */
    sput := 1;               /* 缓冲区内允许放入一件产品 */
    sget := 0;               /* 缓冲区内没有产品 */
cobegin
Process producer
Begin
    L1:
    Produce a product;
    P(sput);
    B := product;
    V(sget);
    Goto L1;
end;
process consumer
begin
    L2:
    P(sget);
    Product:= B;
    V(sput);
    Consume a product;
    Goto L2;
end;
coend
```

另外，要提醒注意的是 PV 操作使用不当的话，则仍会出现与时间有关的错误。例如，有 m 个生产者和 n 个消费者，它们共享可存放 k 件产品的缓冲器。为了使它们能协调的工作，必须使用一个信号量 mutex(初值为 1)，以限制它们对缓冲器的存取互斥地进行，另用两个信号量 sput(初值为 k)和 sget(初值为 0)，以保证生产者不往满的缓冲器中存产品，消费者不从空的缓冲器中取产品。程序如下：

```
var B : array[0..k-1] of item;
    sput:semaphore:=k;       /* 可以使用的空缓冲区数 */
    sget:semaphore:=0;       /* 缓冲区内可以使用的产品数 */
    mutex:semaphore:=1;
    in :integer:= 0;          /* 放入缓冲区指针*/
    out :integer:= 0;         /* 取出缓冲区指针*/


cobegin
Process producer_i
begin
    L1:produce a product;
    P(sput);
    P(mutex);
    B[putptr] := product;
    In:=(in+1) mod k;
    V(mutex);
    V(sget);
    Goto L1;
end;
process consumer_j
begin
    L2:P(sget);
    P(mutex);
    Product:= B[out];
    out:=(out+1) mod k;
    V(mutex);
    V(sput);
    Consume a product;
    Goto L2;
end;
coend
```

在这个问题中 P 操作的次序是很重要的，如果我们把生产者进程中的两个 P 操作交换次序，即那么，当缓冲器中存满了 k 件产品(此时，s1＝0，s＝l，s2＝k)时，生产者又生产了一件产品，它欲向缓冲器存放时将在 P(s1)上等待(注意，现在 s＝0)，但它已经占有了使用缓冲器的权力。这时消费者欲取产品时将停留在 P(s)上得不到使用缓冲器的权力。导致生产者等待消费者取走产品，而消费者却在等生产者释放使用缓冲器的权力，这种相互等待永远结束不了。

所以在使用 PV 操作实现进程同步时，特别要当心 P 操作的次序，而 V 操作的次序倒是无关紧要的。一般来说，用于互斥的信号量上的 P 操作，总是在后执行。

下面再来研究一个较为复杂的生产者/消费者问题。桌上有一只盘子，每次只能放入一只水果。爸爸专向盘子中放苹果(apple)，妈妈专向盘子中放桔于(orange)，一个儿子专等吃盘子中的桔子，一个女儿专等吃盘子里的苹果。

这个问题实际上是两个生产者和两个消费者被连结到仅能放一个产品的缓冲器上。生产者各自生产不同的产品，但就其本质而言，他们是同一类生产者。而消费者则各自取需要的产品消费，他们的消费方式不同。程序如下：

```
plate : integer;
sp:semaphore;              /* 盘子里可以放几个水果 */
sg1:semaphore;             /* 盘子里有桔子 */
sg2:semaphore;             /* 盘子里有苹果 */
sp := 1;                   /* 盘子里允许放入一个水果*/
sg1 := 0;                  /* 盘子里没有桔子 */
sg2 := 0;                  /* 盘子里没有苹果*/
cobegin                                process son
process father                         begin
begin                                      L3: P(sg1);
    L1: 削一个苹果;                        从 plate 中取桔子;
    P(sp);                                 V(sp);
    把苹果放入 plate;                      吃桔子;
    V(sg2);                                goto L3;
    goto L1;                           end;
end;                                   process daughter
process mother                         begin
begin                                      L4: P(sg2);
    L2: 剥一个桔子;                        从 plate 中取苹果;
    P(sp);                                 V(sp);
    把桔子放入 plate;                      吃苹果;
    V(sg1);                                goto L4;
    goto L2;                           end;
end;                                   coend
```

3.3.5 记录型信号量解决读者-写者问题

读者与写者问题(reader-writer problem)也是一个经典的并发程序设计问题。有两组并发进程：读者和写者，共享一个文件 F，要求：(1)允许多个读者同时执行读操作，(2)任一写者在完成写操作之前不允许其它读者或写者工作。(3) 写者执行写操作前，应让已有的写者和读者全部退出。

单纯使用信号量不能完成读者与写者问题，必须引入计数器 rc 对读进程计数，s 是用于对计数器 rc 互斥的信号量，W 表示允许写的信号量，于是管理该文件的程序可如下设计：

```
var rc, wc : integer:=0;
    W, R: semaphore;
    Rc := 0;       /* 读进程计数 */
    W := 1;
    R := 1;
procedure read;
begin
  P(R);
  rc := rc + 1;
  if rc=1 then P(W);
  V(R);
  读文件;
  P(R);
  rc := rc - 1;
```

```
  if rc = 0 then V(W);
  V(R);
end;
procedure write;
begin
  P(W);
  写文件;
  V(W);
end;
```

在上面的解法中，读者是优先的，当存在读者时，写操作将被延迟，并且只要有一个读者活跃，随后而来的读者都将被允许访问文件，从而导致了写者长时间等待。

为了有效解决读者写者问题，有的操作系统专门引进了读者/写者锁。读者/写者锁允许多个读者同时以只读方式存取有锁保护的对象；或一个写者以写方式存取有锁保护的对象。当一个或多个读者已上锁后，此时形成了读锁，写者将不能访问有读锁保护的对象；当锁被请求者用于写操作时，形成了写状态，所有其它进程的读写操作必须等待。

3.3.6 记录型信号量解决理发师问题

另一个经典的进程同步问题是理发师问题。理发店理有一位理发师、一把理发椅和 n 把供等候理发的顾客坐的椅子。如果没有顾客，理发师便在理发椅上睡觉；当一个顾客到来时，它必须叫醒理发师；如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，他们就坐下来等待，否则就离开。

我们的解法引入 3 个信号量和一个控制变量：控制变量 waiting 用来记录等候理发的顾客数，初值均为 0；信号量 customers 用来纪录等候理发的顾客数，并用作阻塞理发师进程，初值为 0；信号量 barbers 用来纪录正在等候顾客的理发师数，并用作阻塞顾客进程，初值为 0；信号量 mutex 用于互斥，初值为 1。程序如下：

```
var waiting : integer;  /*等候理发的顾客数
    CHAIRS:integer;    /*为顾客准备的椅子数
    customers, barbers, mutex : semaphore;
    customers := 0; barbers := 0;
waiting := 0; mutex := 1;
Procedure barber;
begin
while(TRUE);              /*理完一人,还有顾客吗?
P(cutomers);             /*若无顾客,理发师睡眠
  P(mutex);               /*进程互斥
  waiting := waiting – 1;/*等候顾客数少一个
  V(barbers);             /*理发师去为一个顾客理发
  V(mutex);               /*开放临界区
  cut-hair( );            /*正在理发
end;
procedure customer
begin
  P(mutex);               /*进程互斥
  if waiting < CHAIRS     /*看看有没有空椅子
begin
  waiting := waiting + 1;/* 等候顾客数加 1
  V(customers);           /*必要的话唤醒理发师
  V(mutex);               /*开放临界区
  P(barbers);             /*无理发师, 顾客坐着养神
```

```
  get-haircut( )          /*一个顾客坐下等理发
end
  V(mutex);               /*人满了,走吧!
end;
```

3.3.7 AND 型信号量机制

记录型信号量适用于进程之间共享一个临界资源的场合，在更多应用中，一个进程需要先获得两个或多个共享资源后，才能执行其任务。AND 型信号量的基本思想是：把进程在整个运行其间所要的临界资源，一次性全部分配给进程，待该进程使用完临界资源后再全部释放。只要有一个资源未能分配给该进程，其他可以分配的资源，也不分配给他。亦即要么全部分配，要么一个也不分配，这样做可以消除由于部分分配而导致的进程死锁。为此在 P 操作中增加了与条件“AND” ，故称“同时”P 操作，记为 SP(Simultaneous P) 于是 SP(s1，s2，．．．，sn) 和 VS(s1，s2，．．．，sn) 其定义为如下的原语操作：

```
procedure SP(Var s1，．．．sn：semaphore)
begin
    if   s1>=1 &  ．．．& sn>=1    then    begin
       for i:= 1 to n do
           si:= si-1;
    end
    else   begin
           进程进入第一个迁到的满足 si<1 条件的 si 信号量队列等待，
           同时将该进程的程序计数器地址回退，置为 SP 操作处。
    end

procedure VP(Var s1，．．．sn：semaphore)
begin
    for i:=1 to n do    begin
         si:=si+1;
         从所有 si 信号量等待队列中移出进程并置入就绪队列。
    end
```

用 AND 型信号量和 SP、SV 操作解决生产者-消费者问题的算法描述如下：

```
var B : arrary [0,...k-1] of item;
   sput:semaphore:=k;       /*指示有可用的空缓冲区的信号量
   sget:semaphore:=0;       /*指示缓冲区有可用的产品信号量
   mutex:semaphore:=1;      /* 互斥信号量
   sput := k;               /* 缓冲区允许放入的产品数
   sget := 0;               /* 缓冲区内没有产品
   in:integer:= 0;
   out:integer:= 0;
begin
  cobegin
      process producer_i
       begin
         L1:produce a product;
          SP(sput,mutex);
          B[in] := product;
          in:=(in+1) mod k;
          SV(mutex,sget);
        goto L1;
  end;
```

```
process consumer_j
   begin
     L2:SP(sget,mutex);
     Product:= B[out];
     out:=(out+1) mod k;
     SV(mutex,sput);
     consume a product;
    goto L2;
  end;
 coend
end
```

3.3.8 一般型信号量机制

在记录型和同时型信号量机制中，P、V 或 SP、SV 仅仅能对信号量施行增 1 或减 1 操作，每次只能获得或释放一个临界资源。当一请求 n 个资源时，便需要 n 次信号量操作，这样做效率很低。此外，在有些情况下，当资源数量小于一个下限时，便不预分配。为此，可以在分配之前，测试某资源的数量是否大于阀值 t。对 AND 型信号量机制作扩充，便形成了一般型信号量机制，Sp(s1,t1,d1;…;sn,tn,dn) 和 SV(s1,d1;…sn,dn) 的定义如下：

```
procedure    SP(s1,t1,d1;…,sn,tn,dn)
    var s1,…sn:semaphore;
        t1,…tn:integer;
        d1,…dn:integer;
begin
    if s1>=t1& …&sn>=tn then begin
          for i :=1 to n do
            si:=si-di;
    end
    else
        进程进入第一个迂到的满足 si<ti 条件的 si 信号量队列等待，
        同时将该进程的程序计数器地址回退，置为 SP 操作处。
    end
end
procedure    SV((s1,d1;…sn,dn)
    var s1,…sn:semaphore;
        d1,…dn:integer;
begin
       for i:=1 to n do    begin
            si:=si+di;
            从所有 si 信号量等待队列中移出进程并置入就绪队列。
       end
end
```

其中，ti 为这类临界资源的阀值，di 为这类临界资源的本次请求数。

下面是一般信号量的一些特殊情况：

- SP(s,d,d)　此时在信号量集合中只有一个信号量、即仅处理一种临界资源，但允许每次可以申请 d 个，当资源数少于 d 个时，不予分配。
- SP(s,1,1)　此时信号量集合已蜕化为记录型信号量(当 s>1 时)或互斥信号量(s=1 时)。
- SP(s,1,0)　这是一个特殊且很有用的信号量，当 s>=1 时，允许多个进程进入指定区域；当 s 变成 0 后，将阻止任何进程进入该区域。也就是说，它成了一个可控开关。

利用一般信号量机制可以解决读者-写者问题。现在我们对读者-写者问题作一条限制，最多只允许 rn 个读者同时读。为此，又引入了一个信号量 L，赋予其初值为 rn，通过执行 SP(L,1,1) 操作来控制读者的数目，每当一个读者进入时，都要做一次 SP(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，而第 rn+1 个读者必然会因执行 SP(L,1,1) 操作失败而被封锁。

利用一般信号量机制解决读者-写者问题的算法描述如下：

```
var   rn:integer;                  /*允许冈时读的读进程数
      L:semaphore:=rn;             /*控制读进程数信号量,最多 rn
      W:semaphore:=1;
begin
       cobegin
         process reader
           begin
             repeat
               SP(L,1,1 ;W,1,0);
               Read the file;
               SV(L,1);
           Until false;
          end
         process writer
           begin
            Repeat
              SP(W,1,1;L,rn,0);

              Write the file;

              SV(W,1);
          Until false;
         end
    coend
    end
```

上述算法中，SP(W,1,0) 语句起开关作用，只要没有写者进程进入写，由于这时 W=1，读者进程就都可以进入读文件。但一旦有写者进程进入写时，其 W=0，则任何读者进程及其他写者进程就无法进入读写。SP(W,1,1;L,rn,o) 语句表示仅当既无写者进程在写(这时 W=1) 、又无读者进程在读(这时 L=rn) 时，写者进程才能进行临界区写文件。

3.4 管程

3.4.1 管程和条件变量

使用 PV 操作实现同步时，对共享资源的管理分散在各个进程之中，进程能直接对共享变量进行处理，因此，难以防止有意或无意的违法同步操作，而且容易造成程序设计错误。如果能把有关共享变量的操作集中在一起，就可使并发进程之间的相互作用更为清晰,更容易编写出正确的并发程序。

在 1974 年和 1975 年,霍尔(Hoare) 和汉森(Brinch Hansen)提出了一个新的同步机制——管程。把系统中的资源用数据结构抽象地表示出来，因此，对资源的管理就可用数据及在其上实施操作的若干过程来表示;对资源的申请和释放通过过程在数据结构上的操作来实现。而代表共享资源的数据及在其上操作 的一组过程就构成了管程，管程被请求和释放资源的进程所调用。管程是一种程序设计语言结构成分 ,便于用高级语言来书写,它和信号量有同等的表达能力。

管程有以下属性，因而，调用管程的过程时要有一定限制：

- 共享性：管程中的移出过程可被所有要调用管程的进程所共享。
- 安全性：管程的局部变量只能由该管程的过程存取，不允许进程或其它管程来直接存取，一个管程的过程也不应该存取任何非局部于它的变量。
- 互斥性：在任一时刻，共享资源的进程可访问管理该资源的过程，最多只有一个调用者能真正地进入管程，而任何其它调用者必须等待。直到访问者退出。

由上面的讨论可以看出：管程是由若干公共变量及其说明和所有访问这些变量的过程所组成的；进程可以互斥地调用这些过程；管程把分散在各个进程中互斥地访问公共变量的那些临界区集中了起来。管程可以作为语言的一个成分，采用管程作为同步机制便于用高级语言来书写程序，也便于程序正确性验证。

每一个管程都要有一个名字以供标识，如果用语言来写一个管程，它的形式如下：

```
TYPE 〈管程名〉 = MONITOR
  〈管程变量说明〉;
  define 〈（能被其他模块引用的）过程名列表〉;
```

```
  use <（要引用的模块外定义的）过程名列表>;
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
end;
  ……
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
end;
  begin
    <管程的局部数据初始化语句>;
  end;
```

注意，在正常情况下，管程的过程体可以有局部数据。管程中的过程可以有两种，由 define 定义的过程可以被其它模块引用，而未定义的则仅在管程内部使用。管程要引用模块外定义的过程，则必须用 use 说明。

管程的结构可以如图 3-4 所示。

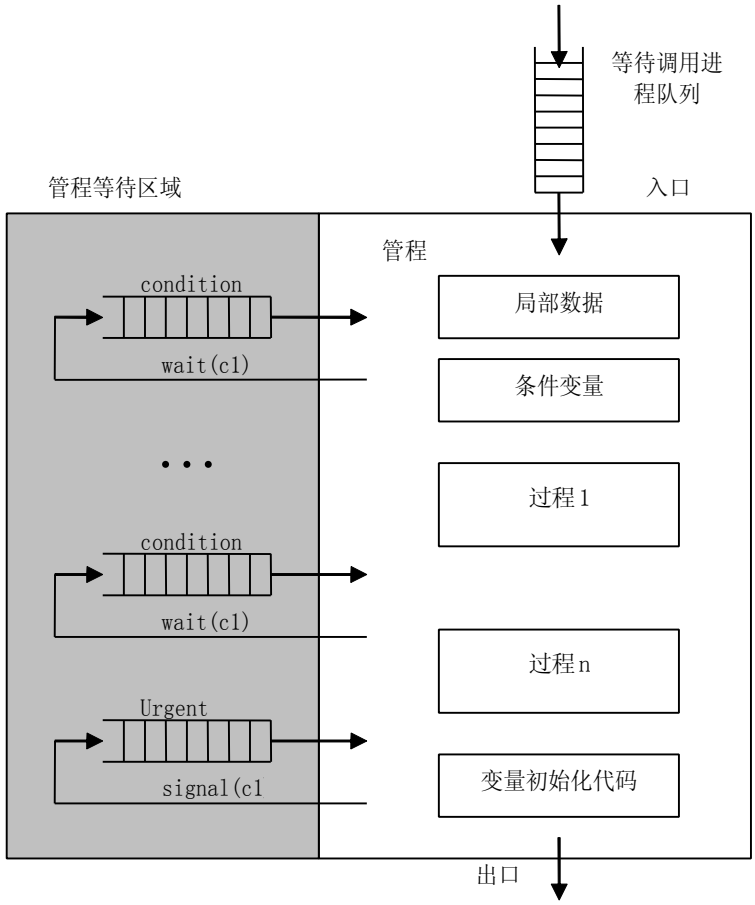


图 3-4　管程的结构示意图

下面先举一个例子来说明管程。系统中有一个资源可以为若干个进程所共享，但每次只能由一个进程使用，用管程作同步机制，对该资源的管理可如下实现：

```
TYPE SSU = MONITOR
  var busy : boolean;
```

```
    nobusy : semaphore;
define require, return;
use wait, signal;
procedure require;
  begin
    if busy then wait(nobusy); /*调用进程加入等待队列*/
    busy := ture;
  end;
procedure return;
  begin
    busy := false;
    signal(nobusy);          /*从等待队列中释放进程*/
  end;
begin                        /*管程变量初始化*/
  busy := false;
end;
```

这是一个名叫 SSU 的管程，它有管理申请和释放资源的两个过程，这两个过程可以在模块外引用，在管程中用 define 子句声明；这两个过程又要用到模块外定义的操作 wait 和 signal，在管程中用 use 子句声明。

尽管如前面所述，对管程过程的调用是互斥的，从而提供了一种实现互斥的简单途径，但是这还不够，我们需要一种办法以使得进程在无法继续运行时被阻塞。解决的方法在于引进条件变量（condition variables），以及在其上操作的两个同步原语 wait 和 signal 。当一个管程过程发现无法继续时（如发现没有可用资源时），它在某些条件变量上执行 wait，这个动作引起调用进程阻塞,当然这时允许先前被挡在管程之外的一个进程进入管程。另一个进程可以通过对其伙伴在在等待的一个条件变量上执行 signal 撑作来唤醒等待进程。值得注意的是，虽然条件变量是一个信号量，但它并不是 PV 操作中所论述纯粹的计数信号量，不能像信号量那样积累供以后使用,它仅仅起到维护等待进程队列的作用，当在一个条件变量上不存在等待条件变量的进程时，signal 操作等于是空操作,wait 操作一般应在 signal 操作之前发出。这一规则大大简化了实现。

管程 SSU 自身定义了一个局部变量 busy，它是一个条件变量，用来表示共享资源的忙闲状态。当有进程要使用资源时调用 require 过程，rquire 过程判 busy 的状态，若有进程在使用资源时，则 busy 为 true，这时调用者进入等待队列,同时开放管程,让其他进程进入工作；若无进程在使用资源，则 busy 为 false，这时调用者可以去使用资源，同时将 busy 置成 true。当进程归还资源时调用 return 过程，该过程把 busy 恢复成 false。此时，若有进程等待使用资源，则它将被释放且立即可得到资源，同时 busy 再次变成 true；若没有进程在等待使用资源，那么，busy 保持 false 状态。

wait 和 signal 是两条原语，在执行时不允许被中断。它们分别表示把某个进程排入等待使用资源的条件变量和从等待资源的条件变量上释放出来。当执行 wait 之后，相应的进程被置成等待状态，同时开放管程，允许其它进程调用管程中过程。当执行 signal 之后，指定条件变量上的一个进程被释放，如果被指定的条件变量上没有进程，则它相当于空操作。

从上面的例中可看到，管程有时要延迟一个不能得到共享资源的进程的执行，当别的进程释放了它所需要的资源时，再恢复该进程的执行。wait 原语可延迟进程的执行，signal 原语使得被延迟进程中的某一个恢复执行。现在的问题是当某进程执行了 signal 操作后，另一个被延迟的进程可恢复执行，于是就可能有两个进程同时调用一个管程中的两个过程，造成管程中两个进程同时执行。可采用两种方法来防止这种现象的出现。

假定进程 P 执行 signal 操作，而队列中有一个进程 Q 在等待，当进程 P 执行了 signal 操作后，进程 Q 被释放，对它们可作如下处理：

- 进程 P 等待直至进程 Q 退出管程，或者进程 Q 等待另一个条件。
- 进程 Q 等待直至进程 P 退出管程，或者进程 P 等待另一个条件。

霍尔采用了第一种办法，而汉森选择了两者的折衷，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。下面将分别介绍汉森和霍尔的实现方法。

3.4.2 Hanson 方法实现管程

用汉森方法实现管程，要用四条原语：wait，signal，check，release。

- 等待原语 wait：执行这条原语后相应进程被置成等待状态，同时开放管程，允许调用管程中其它过程。
- 释放原语 signal：执行这条原语后指定等待队列中的一个进程被释放。如果指定等待队列中没有进程。则它相当于空操作。

只有这两条原语是不够的，为了实现管程的互斥调用功能，还应有另外两条原语：

- 调用查看原语 check：如果管程是开放的，则执行这条原语后关闭管程，相应进程继续执行下去；如果管程是关闭的，则执行这条原语后相应进程被置成等待调用状态。
- 开放原语 release：如果除了发出这条原语的进程外，不再有调用了管程中的过程但文不处于等待状态的进程，那么就释放一个等待调用者(有等待调用者时)，或开放管程(无等待调用者时)。执行这条原语后就认为相应的进程结束了一次调用。

不难看出，在管程的每个过程的头尾分别增加 check 和 release 原语后，互斥调用功能就可实现。因为，进程调用管程中的过程时，过程执行的第一个语句是 check，它的执行保证了互斥调用；当管程中的过程执行结束时，最后一个语句是 release，它的执行保证了管程的开放。

假定对每个管程都定义了一个如下类型的变量：

```
TYPE interf = RECORD
      intsem : semaphore; /*开放和关闭管程的信号量
      count1 : integer;  /* 等待调用的进程个数
      count2 : integer;  /* 调用了管程中的过程且不
END;                          处于等待状态的进程个数
```

其中 intsem 是开放管程的信号量，countl 是等待调用的进程个数，count2 是调用了管程中过程且不处于等待状态的进程个数。

那么，上述四条原语可描述为如下四个过程。

```
procedure wait(var s:semaphore; var IM interf);
  begin
    s := s + 1;
    IM.count2 := IM.count2 - 1;
    if IM.count1 > 0 then
      begin
        IM.count1 := IM.count1 - 1;
        IM.count2 := IM.count2 + 1;
        R(IM.intsem);
      end;
    W(s);
  end;
procedure signal(var s:semaphore; var IM interf);
  begin
    if s > 0 then
      begin
        s := s - 1;
        IM.count2 := IM.count2 + 1;
        R(s);
      end;
  end;
procedure check(var IM interf);
  begin
    if IM.count2 = 0
    then IM.count2 := IM.count2 + 1;
    else
```

```
        begin
          IM.count1 := IM.count1 + 1;
          W(IM.intsem);
        end;
    end;
  procedure release(var IM interf);
    begin
      IM.count2 := IM.count2 - 1;
      if IM.count2 = 0 and IM.count1 > 0 then
        begin
          IM.count1 := IM.count1 - 1;
          IM.count2 := IM.count2 + 1;
          R(IM.intsem);
        end;
    end;
```

注：

- signal 所能释放的一定是同一管程中的某个 wait 原语的执行者。
- 参数 s 表示等待管程中某个条件(资源)的进程个数，它的初值为零。
- W(s)表示将调用过程的进程置成等待信号量 s 的状态；R(s)表示释放一个等待信号量 s 的进程。同样，W(1M．intsem)和 R(1M．intsem)分别表示把调用者置成等待调用管程的状态和释放一个等待调用管程的进程。

用管程实现进程同步时，每个进程应按下列次序工作：

- 请求资源。
- 使用资源。
- 释放资源。

其中，请求资源是调用管程中的一个管理资源分配的过程；释放资源是调用管程中的一个管理回收资源的过程。

现在用例子来说明如何用汉森方法实现进程同步。

例 1：读者与写者问题。

这是一个经典的并发程序设计问题。有两组并发进程：读者和写者，共享一个文件 F，要求：(1)允许多个读者同时执行读操作，(2)任一写者在完成写操作之前不允许其它读者或写者工作；(3)写者欲工作，但在他之前已有读者在执行读操作，那么，待现有读者完成读操作后才能执行写操作，新的读者和写者均被拒绝。

用两个计数器 rc 和 wc 分别对读进程和写进程计数，用 R 和 W 分别表示允许读和允许写的信号量，于是管理该文件的管程可如下设计：

```
type read-writer = MONITOR
  var rc, wc : integer;
      R, W : semaphore;
  define start-read, end-read, start-writer, end-writer;
  use wait, signal, check, release;
procedure start-read;
begin
  check(IM);
  if wc>0 then wait(R,IM);
  rc := rc + 1;
  signal(R, IM);
  release(IM);
end;
procedure end-read;
```

```
begin
  check(IM);
  rc := rc - 1;
  if rc=0 then signal(W,IM);
  release(IM);
end;
procedure start-write;
begin
  check(IM);
  wc := wc + 1;
  if rc>0 or wc>1 then wait(W,IM);
  release(IM);
end;
procedure end-write;
begin
  check(IM);
  wc := wc - 1;
  if wc>0 then signal(W,IM);
  else signal(R, IM);
  release(IM);
end;
begin
  rc := 0; wc := 0; R := 0; W := 0;
end;
```

任何一个进程读（写）文件前，首先调用 start-read（start-write），执行完读（写）操作后，调用 end-read（end-write）。即：

```
cobegin
  process reader
begin
  ……
  call read-writer.start-read;
  ……
  read;
  ……
  call read-writer.end-read;
  ……
end;
  process writer
begin
  ……
  call read-writer.start-write;
  ……
  write;
  ……
  call rear-writer.end-write;
  ……
end;
```

```
coend;
```

上述程序能保证在各种并发执行的情况下,读写进程都能正确工作,请读者自行验证。

例 2：桌上有一只盘子，每次只能放入一只水果。爸爸专向盘子中放苹果(apple)，妈妈专向盘子中放桔于(orange)，一个儿子专等吃盘子中的桔子，一个女儿专等吃盘子里的苹果。写出能使爸爸、妈妈、儿子、女儿同步的管程。

这个问题实际上是两个生产者和两个消费者被连结到仅能放一个产品的缓冲器上，生产者各自生产不同的产品，消费者各自取需要的产品消费。用一个包括两个过程：put 和 get 的管程来实现同步：

```
TYPE FMSD = MONITOR
  var plate : (apple, orange);
      full : boolean;
      SP, SS, SD : semaphore;
  define put, get;
  use wait, signal, check, release;
procedure put(var fruit:(apple, orange));
begin
 check(IM);
  if full then wait(SP,IM);
  full := true;
  plate := fruit;
  if fruit=orange
  then signal(SS,IM);
  else signal(SD,IM);
  release(IM);
end;
procedure get(varfruit:(apple,orange),x:plate);
begin
  check(IM);
  if not full or plate<>fruit
  then begin
    if fruit = orange
    then wait(SS,IM);
    else wait(SD,IM);
  end;
  x := plate;
  full := false;
  signal(SP, IM);
  release(IM);
end;
begin
  full := false; SP := 0; SS := 0; SD := 0;
end;
```

爸爸妈妈向盘中放水果时调用过程 put，儿子女儿取水果时调用过程 get。即：

```
cobegin
```

```
  process father
begin
  ……
  准备好苹果;
  call FMSD.put(apple);
  ……
end;
  process mother
begin
  ……
  准备好桔子;
  call FMSD.put(orange);
  ……
end;
  process son
begin
  ……
  call FMSD.get(orange,x);
  吃取到的桔子;
  ……
end;
  process daughter
begin
  ……
  call FMSD.get(apple,x);
  吃取到的苹果;
  ……
end;
coend;
```

例 3 用 monitor 解决生产者和消费者问题。

```
type producer-consumer = MONITOR
  var B:array[0..k-1] of item;/*缓冲区个数
      in,out:integer;          /*存取指针
      count:integer;           /*缓冲中产品数
      notfull,notempty:semaphore;/*信号量
  define append,take;
  use wait, signal, check, release;
procedure append(x:item);
begin
  check(IM);
  if count=k then wait(notfull,IM); /*缓冲已满
  B[in]:=x;
  in:=(in+1) mod k;
  count:=count+1;                   /*增加一个产品
```

```
  signal(notempty,IM);                  /*唤醒等待者
  release(IM);
end;
procedure take(x:item);
begin
  check(IM);
  if count=0 then wait(notempty,IM); /*缓冲已空
  x:=B[out];
  out:=(out+1) mod k;
  count:=count-1;                       /*减少一个产品
  signal(notfull,IM);                   /*唤醒等待者
  release(IM);
end;

Begin                                   /*初始化
  in := 0; out := 0; count:= 0;
end;

cobegin                                 /*主程序
process producer;
 var   x:item;
 begin
   produce(x);
   append(x);
 end;
 process consumer;
   var x:item;
  begin
   take(x);
   consume(x);
  end;
coend
```

3.4.3 Hoare 方法实现管程

霍尔方法是当有进程等待资源时让执行 signal 操作的进程挂起自己，直到被它释放的进程退出管程或产生了其它的等待条件。这种方法不要求 signal 操作是过程体的最后一个操作。霍尔使用 P 操作原语和 V 操作原语来实现对管程中过程互斥调用的功能，以及实现对共享资源互斥使用的管理。因此， wait 和 signal 操作被设计成两个可以中断的过程。

对于每个管程，使用一个用于管程中过程互斥调用的信号量 mutex(其初值为 1)。任何一个进程调用管程中的任何一个过程时，应执行 P(mutex)；一个进程退出管程时应执行 V(mutex)开放管程，以便让其它调用者进入。为了使一个进程在等待资源期间，其它进程能进入管程，故在 wait 操作中也必须执行 V(mutex)，否则会妨碍其它进程进入管程，导致无法释放资源。

对于每个管程，还必须引入另一个信号量 next(其初值为 0)，凡发出 signal 操作的进程应该用 P(next)挂起自己，直到被释放进程退出管程或产生其它等待条件。每个进程在退出管程的过程之前，都必须检查是否有别的进程在信号量 next 上等待，若有，则用 V(next)唤醒它。在引入信号量 next 的同时，提供一个 next-count(初值为 0)，用来记录在 next 上等待的进程个数。

为了使申请资源者在资源被占用时能将其封锁起来，引入信号量 x-sem(其初值为 0)，申请资源得不到满足时，执行 P(x-sem)挂起自己。由于释放资源时，需要知道是否有别的进程正在等待资源，因而，要用一个计数器 x-count(初值为 0)记录等待资源的进程数。执行 signal 操作时，应让等待资源的诸进程中

的某个进程立即恢复运行，而不让其它进程抢先进入管程，这可以用 V(x-sem)来实现。

于是每个管程都应该定义一个如下的变量：

```
TYPE interf = RECORD
  mutex:semaphore;/*进程调用管程过程前使用的互斥信号量
  next:semaphore; /*发出 signal 的进程挂起自己的信号量
  next_count:integer;/* 在 next 上等待的进程数
END;
```

现在来写 wait 操作和 signal 操作的两个过程：

```
procedure wait(var x_sem:semaphore,x_count:integer, IM:interf);
begin
  x_count := x_count + 1;
  if IM.next_count > 0 then V(IM.next); else V(IM.mutex);
  P(x_sem);
  X_count := x_count – 1;
end;
procedure signal(var x_sem:semaphore,x_count:integer, IM:interf);
begin
  if x_count > 0 then begin
    IM.next_count := IM.next_count + 1;
    V(x_sem);
    P(IM.next);
    IM.next_count := IM.next_count - 1;
  end;
end;
```

任何一个调用管程中过程的外部过程都应该组织成下列形式，以确保互斥地进入管程。

```
P(IM.mutex);
  <过程体>;
if IM.next_count > 0 then V(IM.next);
                     else V(IM.mutex);
```

下面的例子将说明如何用霍尔方法实现进程的同步。

例 1：五个哲学家吃通心面问题。有五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘于，每两人之间放一把叉子。每个哲学家思考、饥饿、然后吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。

首先，我们引入表示哲学家状态的变量：

var state：array[0…4] of (thinking，hungry，eating)

哲学家 i 能建立状态 state[i]=eating，仅当他的两个邻座不在吃的时候，即 state[(i-1)mod 5]≠eathg，以及 state[(i+1)mod5]≠eating。另外还要引入信号量：

var self：array[0··4] of semaphore

当哲学家 i 饥饿但又不能获得两把叉子时，进入等待信号量的队列。于是：

```
TYPE dining-philosophers = MONITOR
  var state : array[0..4] of (thinking, hungry, eating);
```

```
        s : array[0..4] of semaphore;
        s-count : array[0..4] of integer;
    define pickup, putdown;
    use wait, signal;
  procedure test(k : 0..4);
  begin
    if state[(k-1) mod 5]<>eating and state[k]=hungry
       and state[(k+1) mod 5]<>eating then begin
          state[k]  :=  eating;  signal(s[k],  s-count[k],
  IM);
    end;
  end;
  procedure pickup(i:0..4);
  begin
    state[i] := hungry;
    test(i);
    if state[i]<>eating then wait(s[i],s-count[i],IM);
  end;
  procedure putdown(i:0..4);
  begin
    state[i] := thinking;
    test((i-1) mod 5);
    test((i+1) mod 5);
  end;
  begin
    for i := 0 to 4 do state[i] := thinking;
  end;
```

任一个哲学家想吃通心面时调用过程 pickup，吃完通心面之后调用过程 putdown。即：

```
  cobegin
    process philosopher-i
  begin
    ......
        P(IM.mutex);
        call dining-philosopher.pickup(i);
        if IM.next-count > 0 then V(IM.next);
                             else V(IM.mutex);
    吃通心面;
    ......
        P(IM.mutex);
        call dining-philosopher.putdown(i);
        if IM.next-count > 0 then V(IM.next);
                             else V(IM.mutex);
        ......
      end;
  coend;
```

3.5 进程通信

并发进程之间的交往本质上是互相交换信息。有些情况下进程之间交换的信息量很少，例如仅仅交换某个状态信息。有些情况下进程之间交换大批数据，例如传送一批信息或整个文件。进程之间互相交换信息的工作称之为进程通信 IPC(InterProcess Communication)。

进程间通信的方式很多，包括通过软中断提供的信号（signal）通信机制; 使用信号量及其原语操作（PV、读写锁、管程或其他操作）控制的共享存储区(shared memory)通信机制;通过管道(pipeline)提供的共享文件(shared file)通信机制;以及使用信箱和发信/收信原语的消息传递(message passing)通信机制。其中前两种通信方式属于低级通信机制，仅适用于集中式操作系统。消息传递机制属于高级通信机制，共享文件通信机制是消息传递机制的变种，这两种通信机制，既适用于集中式操作系统，又适用于分布式操作系统。

3.5.1 信号通信机制

信号机制又称软中断，是一种简单的通信机制，通过发送一个指定信号来通知进程某个异常事件发生。一般地可以分成 OS 标准信号和用户进程自定义信号,进程收到该信号后便可执行规定的操作。这种机制类似硬件中断但不分优先级,简单有效，但不能传送数据。例如，当系统正在运行一个耗时的程序，如若已发现有错误，并断定该程序要失败，为了节省时间，用户可以按软中断键(一般为 del+ctrl+c)停止程序的执行，这一过程中就用到了信号(signal)。系统具体的操作为：响应键盘输入的中断处理程序向发来中断信号的终端进程发一个信号,进程收到信号后，完成相关处理，然后终止。

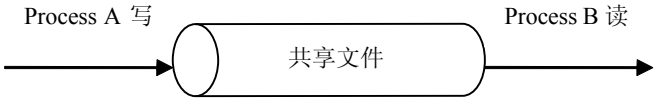
信号不但能从内核发给一个进程，也能由一个进程发给另一个进程。一个信号通过将该信号要送到的进程的 PCB 中的一个域修改来传送，由于每个信号都被看作一个单位,给定类型的信号不能排队,只是在进程被唤醒后才处理信号，或者在进程要从系统请求返回时，可执行操作来响应信号。像 Unix 系统信号多达几十种(不超过 32 种),主要分成以下几类：

- 与进程终止相关的信号 SIGCLD、SIGHUP、SIGKILL、SIGCHLD 等,如进程结束、进程杀死子进程,;
- 与进程例外事件相关的信号 SIGBUS、SIGSEGV、SIGPWR、SIGFPE 等,如进程执行特权指令、写只读区、地址越界、总线超时,硬件故障;
- 与进程执行系统调用相关的信号 SIGPIPE、SIGSYS、SIGILL 等,如进程执行非法系统调用、管道存取错;
- 与进程终端交互相关的信号 SIGINT、SIGQUT 等,如进程挂断终端、用户按 delete 键或 break 健。
- 用户进程发信号 SIGTERM、SIGALRM、SIGUSR1、SIGUS2 等,如进程向另一进程发一个信号、要求报警;
- 跟踪进程执行的信号 SIGTRAP 等。

关于软中信号的处理和实现有以下几点：内核如何向一个进程发软中断信号?进程如何接收软中断信号?以及进程如何控制对软中断信号的动作?①软中断信号通信机制涉及的数据结构在 proc 和 user 结构中，p-clktim 是报警时钟计数器，由系统调用 alarm 放置，时间到就发出 SIGALAM 报警信号。P-sigign 是信号忽略标记，共 32 位，当进程要忽略信号时就把对应位置 1。P-sig 进程接收信号位，共 32 位，进程收到信号时，对应位置 1。U-signal[NSIG]是 32 个元素的数组，每个元素存放一个软中断处理程序的入口地址，让用户进程提供特殊处理。②信号发送工作由系统调用 kill(pid,sig)完成,pid 确定了 sig 发往进程,sig 是信号类型,由于这种信号要知道发往进程的标识号,所以信号发送通常在关系密切进程之间进行。信号发送需有一定权力,超级用户进程可把信号发送给任何进程,但普通用户进程向非同组用户发送信号,那么 kill 调用会失败。③信号的接收和处理使用系统调用 signal(sig,func),sig 指出信号类型,function 描述与信号关联的操作。如果 signal 调用成功,则将转入关联函数执行,否则返回错误值供程序处理。

3.5.2 共享文件通信机制`

管道(pipeline)是连接读写进程的一个特殊文件，允许进程按先进先出方式传送数据,也能使进程同步执行操作。如下图所示，发送进程以字符流形式把大量数据送入管道，接收进程从管道中接收数据，所以，也叫管道通信。由于方便有效，目前已被引入到许多操作系统中。



管道（pipe）是 Unix 和 C 语言的传统通信方式,也是 Unix 发展最有意义的贡献之一。管道通信的方式是发送者进程和接收者进程之间通过一个管道交流信息，管道是单向的，发送者进程只能写入信息，接收者进程也只能接收信息。管道的实质是一个共享文件，因此管道通信基本上可以借助于文件系统原有的机制实现，包括（管道）文件的创建、打开、关闭和读写。但是，写入进程和读出进程之间的相互协调单靠文件系统机制是解决不了的。读写进程相互协调，必须做到以下三点：

- 进程对通信机构的使用应该是互斥的，一个进程正在使用某个管道写入或读出数据时，另一个进程就必须等待。这一点是进程在读写管道之前,通过测试文件 i 节点的特征位来保证的;
- 发送者和接收者双方必须能够知道对方是否存在，如果对方已经不存在，就没有必要再发送信息。这时会发出 SIGPIPE 信号通知进程;
- 由于管道长度有限,发送信息和接收信息之间一定要实现正确的同步关系。管道文件最多只能提供 5120 字节的缓冲,管道的长度对 write 和 read 操作会有影响。如果执行一次写操作,且管道有足够空间,那么,write 把数据写入管道后立即返回;如果这次操作会引起管道溢出,则本次 write 操作必须暂停,直到其他进程从管道中读出数据,使管道有空间为止,这叫 write 阻塞。解决此问题的办法是：把数据进行切分,每次最多 5120 字节,写完后该进程睡眠,直到读进程把管道中的数据取走,并判别有进程等待时应唤醒他,以便继续写下一批数据。反之,当读进程读空管道时,要出现读阻塞，读进程应睡眠, 直到写进程唤醒他。

在 UNIX 中，管道的定义如下：

```
int pipe(files);
int files[2];
```

核心在执行完 pipe( )系统调用创建无名管道后返回文件句柄 files[0]和 files[1]，接收者进程通过 files[0]从管道中取走信息，发送者进程则通过 files[1]向管道写入数据。下面是父子进程通过管道传送信息的一个例子：

```
#include<stdio.h>
#define MSGSIZE 16
char *msg1=”hello,world#1”;
char *msg2=”hello,world#2”;
char *msg3=”hello,world#3”;
 main( )
  {
      char inbuf[MSGSIZE];
      int p[2],j,pid;
      /*open pipe*/
       if(pipe<0) {
            perror(“pipe call”);
            exit(1);
   }
       if((pid=fork( )<0) {
            perror(“fork call”);
            exit(2);
   }
  /*if parent, then close read file descriptor and write down pipe*/
      if(pid>0 {
            close(p[0]);
            write(p[1],msg1,MSGSIZE);
            write(p[1],msg2,MSGSIZE);
            write(p[1],msg3,MSGSIZE);
            wait((int*)0);
     }
 /*if child ,then close write file descriptor and read from pipe*/
       if (pid==0) {
            close(p[1]);
            for(j=0;j<3;j++)
               read(p[0],inbuf,MSGSIZE);
               printf(“%s\n,inbuf”);
      }
   }
  exit(0);
}
```

管道是一种功能很强的通信机制,但它仅能用于连接具有共同祖先的进程,管道也不是常设的,需临时

建立,难于提供全局服务。为了克服这些缺点,Unix 中又推出了管道的一个变种称有名管道或 FIFO 通信机制。这是一种永久性机制,具有 Unix 文件名、访问权限,能像一般文件一样被打开、关闭、删除,但 write 和 read 时,其性能与管道相同。可以通过系统调用 mknod(pipename, S_FIFO+rw, 0)而不是 pipe 来创建命名管道，然后接收者进程可以通过系统调用 open(pipename, O_RDONLY)来打开管道以取走信息，发送者进程则通过系统调用 open(pipename, O_WRONLY)来打开管道以写入数据。对管道的读写通过系统调用 read 和 write 来实现。

另外，在操作控制命令层也可以使用管道：

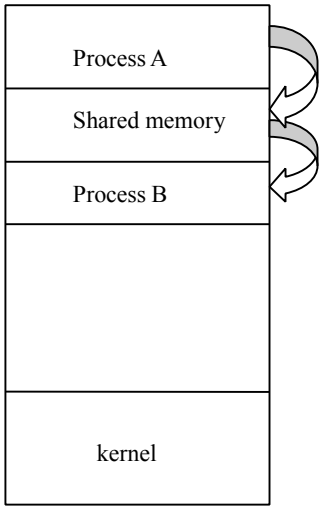
```
who | sort | more
```

管道是基于信箱的消息传递方式的一种变体，它们与传统的信箱方式等价，区别在于没有预先设定消息的边界。换言之，如果一个进程发送 10 条 100 字节的消息，而另一个进程接收 1000 个字节，那么接收者将一次获得 10 条消息。

随着应用的扩展，管道机制不仅仅适用于集中式系统中的进程一对一通信，多路转接器被用来解决一对一通信的限制，在 AT&T UNIX 的近期版本中管道机制也适用于网络通信。

3.5.3 共享存储区通信机制

内存中开辟一个共享存储区，如图所示,诸进程通过该区实现通信,这是进程通信中最快的方法。进程通信之前，向共享存储区申请一个分区段，并指定关键字;若系统已为其他进程分配了这个分区，则返回关键字给申请者，于是该分区段就可连到进程，以后，进程便像通常存储器一样共享存储区段。



与共享存储有关的系统调用有四个：

- shmget(key,size,permflags)
  用于建立共享存储段,或返回一个已存在的共享存储段,相应信息登入共享存储段表中。size 给出共享存储段的最小字节数;key 是标识这个段的描述字;permflags 给出该存储段的权限。
- shmat(shm-id,daddr,shmflags)
  用于把建立的共共享存储段连入进程的逻辑地址空间。Shm-id 标识存储段,其值从 shmget 调用中得到;daddr 用户的逻辑地址;permflags 表示共享存储段可读可写或其它性质。
- Shmdt(memptr)
  用于把建立的共享存储段从进程的逻辑地址空间中分离出来。Memptr 为被分离的存储段指针。
- Shmctl(shm-id,command,&shm-stat)
  实现共享存储段的控制操作。Shm-id 为共享存储段描述字;command 为规定操作;&shm-stat 为用户数据结构的地址。

当执行 shmget 时,内核查找共享存储段中具有给定 key 的段,若已发现这样的段且许可权可接受,便返回共享存储段的 key;否则,在合法性检查后,分配一个存储段,在共享存储段表中填入各项参数,并设标志指

示尚未存储空间与该区相联。执行 shmat 时,首先查证进程对该共享段的存取权,然后把进程合适的虚空间与共享存储段相联。执行 shmdt 时,其过程与 shmat 类似,但把共享存储段从进程的虚空间断开。

3.5.4 消息传递通信机制

3.5.4.1 消息传递的概念

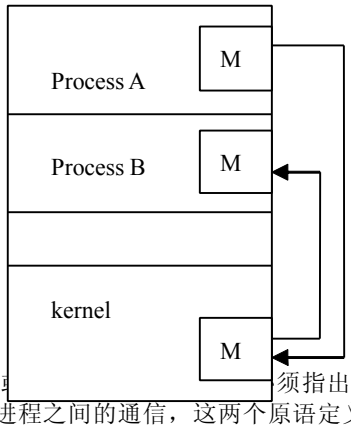
在前面几节讨论中已经看到，系统中的交往进程通过信号量及有关操作可以实现进程互斥和同步。例如，生产者和消费者问题是一组相互协作的进程，它们通过交换信号量达到产品递交和使用缓冲器的目的。这可以看作是一种低级的通信方式。有时进程间可能需要交换更多的信息，例如，一个输入输出操作请求，要求把数据从一个进程传送给另一个进程，这种大量的信息传递可使用一种高级通信方式——消息传递（message passing）来实现。由于操作系统隐蔽了许多实现细节，通过消息传递机制通信，能简化程序编制的复杂性，方便易用，得到了广泛应用。

消息传递机制至少需要提供两条原语 send 和 receive，前者向一个给定的目标发送一个消息，后者则从一个给定的源接受一条消息。如果没有消息可用，则接收者可能阻塞直到一条消息到达，或者也可以立即返回，并带回一个错误码。

采用了消息传递机制后，进程间用消息来交换信息。一个正在执行的进程可以在任何时刻向另一个正在执行的进程发送一个消息；一个正在执行的进程也可以在任何时刻向正在执行的另一个进程请求一个消息。如果一个进程在某一时刻的执行依赖于另一进程的消息或等待它进程对发出消息的回答，那么，消息传递机制将紧密地与进程的阻塞和释放相联系。这样，消息传递就进一步扩充了并发进程间对数据的共享，提供了进程同步的能力。

3.5.4.2 消息传递的方式

消息传递系统的变体很多，常用的有直接通信（消息缓冲区）方式和间接通信（信箱）方式， Unix 的 pipeline 和 socket 机制属于一种信箱方式的变体。下图是消息传递机制通信模型。



1）直接通信方式

在直接通信方式下，企图发送[illegible]须指出信件发给谁或从谁那里接收消息，可用 send 原语和 receive 原语为实现进程之间的通信，这两个原语定义如下：

- send（P，消息）：把一个消息发送给进程 P。
- receive（Q，消息）：从进程 Q 接收一个消息。

这样，进程 P 和 Q 通过执行这两个操作而自动建立了一种联结，并且这一种联结仅仅发生在这一对进程之间。

消息可以有固定长度或可变长度两种。固定长度便于物理实现，但使程序设计增加困难；而消息长度可变使程序设计变得简单，但使物理实现复杂化。

2）间接通信方式

采用间接通信方式时，进程间发送或接收消息通过一个信箱来进行，消息可以被理解成信件，每个信箱有一个唯一的标识符。当两个以上的进程有一个共享的信箱时，它们就能进行通信。一个进程也可以分别与多个进程共享多个不同的信箱，这样，一个进程可以同时和多个进程进行通信。在间接通信方式“发送”和“接收”原语的形式如下：

- send（A，信件）：把一封信件（消息）传送到信箱 A。
- receive（A，信件）：从信箱 A 接收一封信件（消息）。

信箱是存放信件的存储区域，每个信箱可以分成信箱特征和信箱体两部分。信箱特征指出信箱容量、信件格式、指针等；信箱体用来存放信件，信箱体分成若干个区，每个区可容纳一封信。

“发送’和“接收”两条原语的功能为：

- 发送信件。如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待该信箱中信件的等待者；否则发送信件者被置成等待信箱状态。
- 接收信件。如果指定信箱中有信，则取出一封信件，并释放等待信箱的等待者，否则接收信

件者被置成等待信箱中信件的状态。
两个原语的算法描述如下，其中，R( ) 和 W( ) 是让进程入队和出队的两个过程。

```
type box=record
      size:integer;                              /*信箱大小
      count:integer;                             /*现有信件数
      letter:array[1..n] of message;             /*信箱
      S1,S2:semaphore;                           /*等信箱和等信件信号量
end

procedure send(varB:box,M:message)
    var I:integer;
    begin
       if B.count=B.size    then W(B.s1);
       i:=B.count+1;
    B.letter[i]:=M;
       B.coumt:=I;
       R(B.S2)
     end;{send}

procedure receive(varB:box,x:message)
    var i:integer;
    begin
       if B.count=0    then W(B.s2);
       B.count:=B.count-1;
       x:=B.letter[1];
       if B.count not=0 then for i=1 to b.count do B.letter[i]:=B.letter[i+1];
       R(B.S1);
  end;{receive}
```

下面是用消息传递机制解决生产者-消费者问题的程序。

```
var    capacity:integer;                          /*缓冲大小
       i:integer;

procedure producer;
         var pmsg:message;
      begin
         while true do
         begin
            receive (mapproduce,pmsg);             /*等待空消息
            pmsg:=produce;                         /*生产消息
            send(mayconsume,pmsg);                 /*发送消息
         end
       end;

procedure consumer;
         var cmsg:message;
      begin
         while true do
         begin
            receive (mayconsume,cmsg);             /*接收消息
            consume(csmg);                         /*消耗消息
            send(mayproduce,null);                 /*发送空消息
         end
       end;

begin                                              /*主程序
      creat-mailbox(mayprocuce);                   /*创建信箱
      creat-mailbox(mayconsume);
      for i=1 to capacity do send (mayproduce,null);    /*发送空消息

    cobegin
```

```
        producer;
        consumer;
    coend
end
```

3.5.5 有关消息传递实现的若干问题

下面讨论消息传递系统中的几个问题。

首先,是信箱容量问题。一个极端的情况是信箱容量为 0，那么当 send 在 receive 之前执行的话，则发送进程被阻塞，直到 receive 做完。执行 receive 时信件可从发送者直接拷贝到接收者，不用任何中间缓冲。类似的，如果 receive 先被执行，接受者将被阻塞直到 send 发生。上述策略人称为回合（rendezvous）原则。这种方案实现较为容易，但却降低了灵活性，发送者和接收者一定要以步步紧接的方式运行。通常情况采用带有信件缓冲的方案、即信箱可放有限封信，这时一个进程可以连续做发送信件操作而无需等待直到信箱满，这种方式下，系统具有迫使一个进程等信箱和释放等信箱的功能。

其次，关于多进程与信箱相连的信件接收问题。采用间接通信时，有时会出现如下问题，假设进程 P1, P2 和 P3 都共享信箱 A，P1 把一封信件送到了信箱 A，而 P2 和 P3 都企图从信箱 A 取这个信件，那么，究竟应由谁来取 Pl 发送的信件呢?解决的办法有以下三种：

- 预先规定能取 Pl 所发送的信件的接收者。
- 预先规定在一个时间至多一个进程执行一个接收操作。
- 由系统选择谁是接收者。

第三，关于信箱的所有权问题。一个信箱可以由一个进程所有，也可以由操作系统所有。如果一个信箱为一个进程所有，那么必须区分信箱的所有者和它的用户，区分信箱的所有者和它的用户的一个方法是允许进程说明信箱类型 mailbox，说明这个 mailbox 的进程就是信箱的所有者，其它任何知道这个 mailbox 名字的进程都可成为它的用户。当拥有信箱的进程执行结束时，它的信箱也就消失，这时必须 把这一情况及时通知这个信箱的用户。信箱为操作系统所有是指由操作系统统一设置信箱，消息缓冲就是一个著名的例子。

消息缓冲通信是 1973 年由 P.B.Hansan 提出的一种进程间高级通信原语，并在

RC4000 系统中实现。消息缓冲通信的基本思想是：由操作系统统一管理一组用于通信的消息缓冲 存储区，每一个消息缓冲存储区可存放一个消息(信件) 。当一个进程要发送消息时，先在自己的消息发送区里生成发送的消息，包括：接收进程名、消息长度、消息正文等。然后向系统申请一个消息缓冲区，把消息从发送区复制到消息缓冲区中，注意在复制过程中系统会将接近进程名换成发送进程名，以便接收者识别。随后该消息缓冲区被挂到接收消息的进程的消息队列上，供接近者在需要时从消息队列中摘下并复制到消息接近区去使用，同时释放消息缓冲区。如图 3-5 所示：

消息缓冲通信涉及的数据结构有：

- sender：发送消息的进程名或标识符
- size：发送的消息长度
- text：发送的消息正文
- next-ptr：指向下一个消息缓冲区的指针

在进程的 PCB 中涉及通信的数据结构：

- mptr：消息队列队首指针
- mutex：消息队列互斥信号量, 初值为 1
- sm：表示接收进程消息队列上消息的个数，初值为 0，是控制收发进程同步的信号量

发送原语和接收原语的实现如下：

- 发送原语 send：申请一个消息缓冲区，把发送区内容复制到这个缓冲区中；找到接收进程的 PCB，执行互斥操作 P(mutex) ；把缓冲区挂到接收进程消息队列的尾部，执行 V(sm) 、即消息数加 1；执行 V(mutex) 。
- 接收原语 receive：执行 V(sm) 查看有否信件；执行互斥操作 P(mutex) ，从消息队列中摘下第一个消息，执行 V(mutex) ；把消息缓冲区内容复制到接收区，释放消息缓冲区。

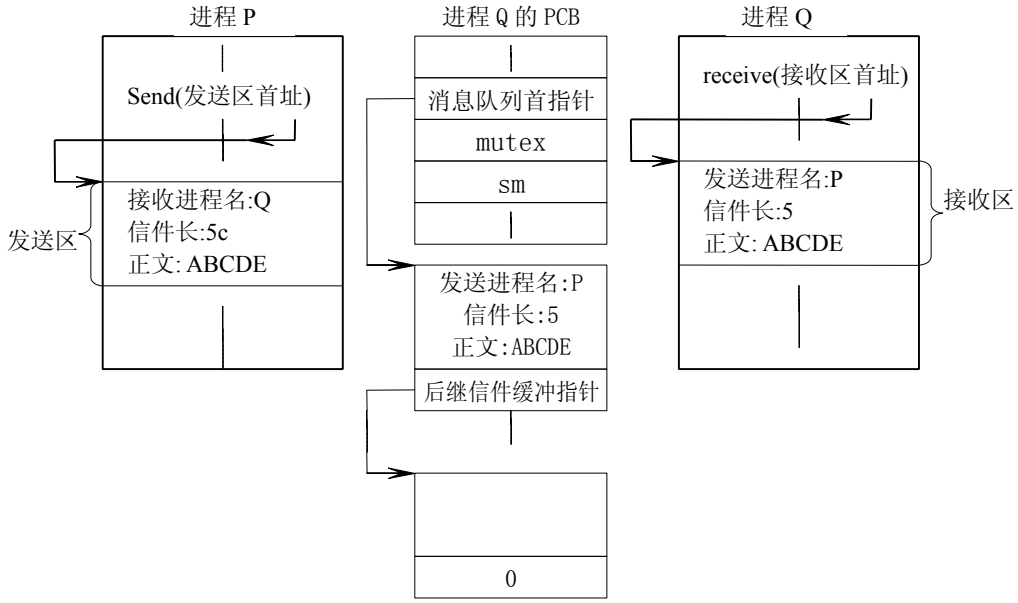


图 3-5　消息缓冲通信

Unix 的消息传递机制与信箱通信很类似,进程间的通信通过消息队列进行，消息队列可以单消息队列,也可以多消息队列(按消息类型);既可以单向,也可以双向通信。所用数据结构有:

- 消息缓冲池和消息缓冲区(msgbuf),前者包含消息缓冲池大小和首地址;后者除存放消息正文外,还有消息类型字段。
- 消息头结构和消息头表 ,消息头表是由消息头结构组成的数组,个数为 100。消息头结构包含消息类型、消息正文长度、消息缓冲区指针和消息队列中下一个消息头结构的链指针。
- 消息队列头结构和消息队列头表,由于可有多个消息队列,于是对应每个消息队列都有一个消息队列头结构, 消息队列头表是由消息队列头结构组成的数组。消息队列头结构包括：指向队列中第一个消息的头指针、指向队列中最后一个消息的尾指针、队列中消息个数、队列中消息数据的总字节数、队列允许的消息数据最大字节数、最近一次发送/接收消息进程标识和时间。

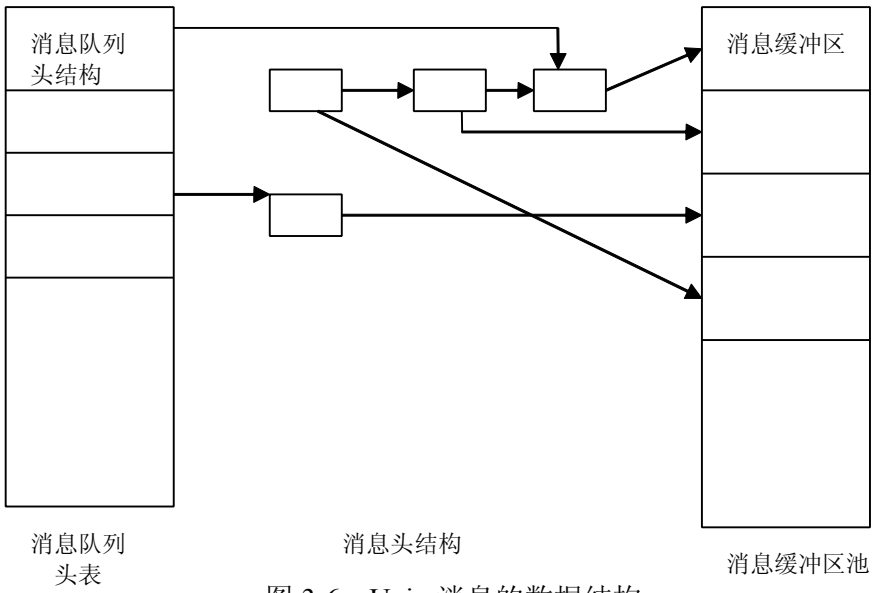


图 3-6　Unix 消息的数据结构

Unix 消息传递机制的系统调用有四个：

- 建立一个消息队列 msgget
- 向消息队列发送消息 msgsnd
- 从消息队列接收消息 msgrcr

- 取或送消息队列控制信息 msgctl

当用户使用 msgget 系统调用来建立一个消息队列时，内核查遍消息队列头表以确定是否已有一个给定关键字的队列存在,如果没有,内核创建一个新的消息队列,并返回给用户一个队列消息描述符;否则,内核检查许可权后返回。进程使用 msgsnd 发送一个消息，内核检查发送进程是否对该消息描述符有写许可权，消息长度不超过规定的限制等;接着分配给一个消息头结构,链入该消息头结构链的尾部;在消息头结构中填入相应信息,把用户空间的消息复制到消息缓冲池的一个缓冲区,让消息头结构的指针指向消息缓冲区，修改数据结构;然后内核便唤醒等待该消息队列消息的所有进程。进程使用 msgrcv 接收一个消息,内核检查接收进程是否对该消息描述符有读许可权,根据消息类型(=0,<0,>0)找出所需消息,从内核消息缓冲区复制内容到用户空间，消息队列中删去该消息,修改数据结构,如果有发送进程因表满而等待，内核便唤醒等待该消息队列的所有进程。

第四,关于信件的格式问题。单机系统中信件的格式可以分直接信件(又叫定长格式)和间接信件(又叫变长格式)。前者将消息放在信件中直接交给收信者，但信息量较小；后者信件中仅传送消息的地址，一般说信息量没有限制。计算机网络环境下的信件格式较为复杂，通常分成消息头和消息体，前者包括了发送者、接收者、消息长度、消息类型、发送时间等各种控制信息；后者包含了消息内容。

第五，关于通信进程并行性问题。发送进程发出一封信件后，它本身的执行可以分两种情况，一种是等待收到接收进程回答消息后才继续进行下去；另一种是发出信件后不等回信立即执行下去，直到某个时刻需要接收进程送来的消息时，才对回答信件进行处理。显然后一种情况并行性高些，但是，要求增加两条原语。

- Answer(P,result)：向进程 P 送回信
- Wait(Q,result)：等待进程 Q 的回信

于是，并行的通信进程的程序应如下编制：

```
cobegin
     procedure  P
       begin
         ...
         send(Q,message);
         ...
         wait(Q,result);
         ...
       end;

procedure Q
  begin
       ...
       receive(P,message);
       ...
       answer(P,result);
       ...
  end;
coend;
```

3.6 死锁

3.6.1 死锁的产生

计算机系统中有许多独占资源，它们在任一时刻都只能被一个进程使用，如磁带机、键盘、绘图仪等独占型外围设备，或进程表、临界区等软件资源。两个进程同时向一台打印机输出将导致一片混乱，两个进程同时进入临界区将导致数据错误乃至程序崩溃。正因为这些原因，所有操作系统都具有授权一个进程独立访问某一资源的能力。一个进程需要使用独占资源必须通过以下的次序：

- 申请资源

- 使用资源
- 归还资源

若申请时资源不可用，则申请进程等待。对于不同的独占资源，进程等待的方式是有差异的，如申请打印机资源、临界区资源时，申请失败将意味着阻塞申请进程；而申请打开文件资源时，申请失败将返回一个错误码，由申请进程等待一段时间之后重试。值得指出的是，不同的操作系统对于同一种资源采取的等待方式也是有差异的。

在许多应用中，一个进程需要独占访问不止一种资源。而操作系统允许多个进程并发执行共享系统资源时，此时可能会出现进程永远被阻塞的现象。例如，两个进程分别等待对方占有的一个资源，于是两者都不能执行而处于永远等待。这种现象称为“死锁”。

为了清楚地说明死锁情况，下面列举若干死锁的例子。

例 1　竞争资源产生死锁。

设系统有打印机、读卡机各一台，它们被进程 P 和 Q 共享。两个进程并发执行，它们按下列次序请求和释放资源：

进程 P	进程 Q
请求读卡机	请求打印机
请求打印机	请求读卡机
释放读卡机	请求读卡机
释放打印机	释放打印机

它们执行时，相对速度无法预知，当出现进程 P 占用了读卡机，进程 Q 占用了打印机后，进程 P 又请求打印机，但因打印机被进程 Q 占用，故进程 P 处于等待资源状态；这时，进程 Q 执行，它又请求读卡机，但因读卡机被进程 P 占用而也只好处于等待资源状态。它们分别等待对方占用的资源，致使无法结束这种等待，产生了死锁。

例 2　PV 操作使用不当产生死锁。

设进程 Q1 和 Q2 共享两个资源 r1 和 r2，s1 和 s2 是分别代表资源 r1 和 r2 能否被使用的信号时，由于资源是共享的，所以必须互斥使用，因而 s1 和 s2 的初值均为 1。假定两个进程都要求使用两个资源，它们的程序编制如下：

进程 Q1	进程 Q2
……	……
P(S1);	P(s2);
P(s2);	P(s1);
……	……
使用 r1 和 r2;	使用 r1 和 r2
……	……
V(S1);	V(s2);
V(S2);	V(S1);
……	……

由于 Q1 和 Q2 并发执行，于是可能产生这样的情况：进程 Q1 执行了 P(s1)后，在执行 P(s2)之前，进程 Q2 执行了 P(s2)，当进程 Q1 再执行 P(s2)时将等待，此时，Q2 再继续执行 P(s1)，也处于等待。这种等待都必须由对方来释放，显然，这是不可能的，又产生了死锁。

例 3　资源分配不当引起死锁

若系统中有 m 个资源被 n 个进程共享，当每个进程都要求 K 个资源，而 $m < n \cdot K$ 时，即资源数小于进程所要求的总数时，如果分配不得当就可能引起死锁。例如，m=5, n=5, k=2，采用的分配策略是为每个进程轮流分配。首先，为每个进程轮流分配一个资源，这时，系统中的资源都已分配完了；于是第二轮分配时，各进程都处于等待状态，导致了死锁。

例 4　对临时性资源使用不加限制引起的死锁

在进程通信时使用的信件可以看作是一种临时性资源，如果对信件的发送和接收不加限制的话，则可能引起死锁。比如，进程 P1 等待进程 P3 的信件 S3 来到后再向进程 P2 发送信件 S1；P2 又要等待 P1

的信件 S1 来到后再向 P3 发送信件 S2；而 P3 也要等待 P2 的信件 S2 来到后才能发出信件 S3。在这种情况下就形成了循环等待，永远结束不了，产生死锁。

综合上面的例子可见，产生死锁的因素不仅与系统拥有的资源数量有关，而且与资源分配策略，进程对资源的使用要求以及并发进程的速率有关。

出现死锁会造成很大的损失，因此，必须花费额外的代价来预防死锁的出现。可从三个方面来解决死锁问题。它们是“死锁的防止；死锁的避免；死锁的检测和解除。

3.6.2 死锁的定义

死锁可能是由于竞争资源而产生，也可能是由于程序设计的错误所造成，因此，在讨论死锁的问题时，为了避免和硬件故障以及其它程序性错误纠缠在一起，我们作如下假定：

假定 1：任意一个进程要求资源的最大数量不超过系统能提供的最大量；

假定 2：如果一个进程在执行中所提出的资源要求能够得到满足，那么它一定能在有限的时间内结束。

假定 3：一个资源在任何时间最多只为一个进程所占有。

假定 4：一个进程一次申请一个资源，且只在申请资源得不到满足时才处于等待状态。换言之，其它一些等待状态，例如：人工干预、等待外围设备．传输结束等，在没有故障的条件下，可以在有限长的时间内结束，不会产生死锁。因此，这里不考虑这种等待。

假定 5：一个进程结束时释放它占有全部资源。

假定 6：系统具有有限个进程和资源。

现在来给出死锁的定义：

我们说一组进程处于死锁状态是指：如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件,则称一组进程或系统此时发生了死锁。例如，n 个进程 P1、P2……Pn，Pi（i=…n）因为申请不到资源 Rj(j=1、..、m)而处于等待状态，而 Rj 又被 Pi+1（i=1,…n-1）占有，Pn 欲申请的资源被 P1 占有，显然，此时这 n 个进程的等待状态永远不能结束，我们说这 n 个进程处于死锁状态。

3.6.3 驼鸟算法

最简单的方法是象鸵鸟一样对死锁视而不见。对该方法各人的看法不同。数学家认为不管花多大代价也要彻底防止死锁的发生；工程师们则要了解死锁发生的频率、系统因其他原因崩溃的频率、以及死锁有多严重，如果死锁平均每 50 年发生一次，而系统每个月会因硬件故障、编译器错误或操作系统错误而崩溃一次，那么大多数工程师不会不惜工本地去消除死锁。

更具体地说，Unix 和 MINIX 潜在地受到某些死锁的威胁，不过这些死锁从来没有发生过，甚至从没有被检测到过。举个例子，系统中进程的数目受进程表项多少的制约，进程表项是有限的资源，如果一个 FORK 调用由于进程表用完而失败，那么一种合理的办法是等待一段随后的时间后重试。

现假设一个 Unix 系统的进程表有 100 项，有 10 个进程在执行，每一个都要创造 12 个子进程。在每个进程创建 9 个子进程后，进程表项被全部用完。则这 10 个进程将进入一个无休止的循环：执行 FORK，失败，等待一段时间后执行 FORK，再失败…，这实际上是死锁。发生这类事件的概率是很小的，但它的确存在！我们难道会为了消除这种状况就放弃进程、FORK 这些概念和方法吗？

打开文件的最大数目受 i-节点表大小的限制，所以当 i-节点表满时会发生类似的问题。磁盘上的对换空间也是另一种有限的资源。实际上，几乎操作系统中的每一种表格都代表了一种有限的资源。难道我们会因为可能出现类似的死锁而抛掉这一切吗？

Unix 处理这一问题的办法是忽略它，因为大多数用户宁可在极偶然的情况下发生死锁，也不愿限制每个用户只能创建一个进程，只能打开一个文件等等。如果死锁可以不花什么代价就能够解决，则什么问题都没有了。问题是，这种代价很大，而且常常给用户带来许多不便的限制。于是我们不得不在方便性和正确性之间作出折衷。

3.6.4 死锁的防止

3.6.4.1 死锁产生的条件

1971 年 Coffman 总结出了系统产生死锁必定同时保持四个必要条件：

互斥条件(Mutual Exclusion) 进程应互斥使用资源,任一时刻一个资源仅为一个进程独占，若另一个进程请求一个已被占用的资源时，它被置成等待状态，直到占用者释放资源。

占有和等待条件(Hold and Wait) 一个进程请求资源得不到满足而等待时，不释放已占有的资源。

不剥夺条件(No Preemption) 任一进程不能从另一进程那里抢夺资源，即已被占用的资源，只能由占用进程自己来释放。

循环等待条件(Circular Wait) 存在一个循环等待链，其中，每一个进程分别等待它前一个进程所持有的资源,造成永远等待。

只要能破坏这四个必要条件之一，死锁就可防止。破坏第一个条件，使资源可同时访问而不是互斥

使用，是个简单的办法，磁盘可用这种办法管理,但有许多资源往往是不能同时访问的，所以这种做法许多场合行不通。采用剥夺式调度方法可以破坏第三个条件，但剥夺调度方法目前只适用于对主存资源和处理器资源的分配，当进程在申请资源未获准许的情况下,如主动释放资源(一种剥夺式),然后才去等待，以后再一起向系统提出申请,也能防止死锁,但这些办法不适用于所有资源。由于种种死锁防止办法施加于资源的限制条件太严格,会造成资源利用率和吞吐率低。下面介绍两种比较实用的死锁防止方法，它们能破坏第二个条件或第四个条件。

3.6.4.2 静态分配策略

所谓静态分配是指一个进程必须在执行前就申请它所要的全部资源，并且直到它所要的资源都得到满足后才开始执行。无疑的，所有并发执行的进程要求的资源总和不超过系统拥有的资源数。采用静态分配后，进程在执行中不再申请资源，因而不会出现占有了某些资源再等待另一些资源的情况，即破坏了第二个条件的出现。静态分配策略实现简单，因而被评多操作系统采用。但这种策略严重地降低了资源利用率，因为在每个进程所占有资源中，有些资源在进程较后的执行时间里使用的，甚至，有些资源在例外的情况下被使用。这样，就可能是一个进程占有了一些几乎不用的资源而使它想用的这些资源的进程产生等待。

3.6.4.3 层次分配策略

这种分配策略将阻止循环等条件的出现。在层次分配策略下，资源被分成多个层次，一个进程得到某一层的一个资源后，它只能再申请在较高一层的资源；当一个进程要释放某层的一个资源时，必须先释放所占用的较高层的资源，当另一个进程获得了某一个层的一次资源后，它想再申请该层中的另一个资源，那么，必须先释放该层中的已占资源。

这种策略的一个变种是按序分配策略。把系统的所有资源排一个顺序，例如，系统若共有 n 个进程,共有 m 个资源，用 ri 表示第 i 个资源，于是这 m 个资源是：

$$r1, r2\cdots\cdots, rm$$

规定如果进程不得在占用资源 ri($1\leqslant i\leqslant m$)后再申请 rj($j<i$)。不难证明，按这种策略分配资源时系统不会发生死锁。可以用反证法来证明按序分配不会产生死锁,事实上，若在时刻 t1，进程 P1 处于等资源 rk1 的状态，则 rk1 必为另一进程假定是 P2 所占用,若 P2 在有限时间里可以运行结束，那么 P1 就不会处于永远等待状态；所以一定在某个时刻 t2，进程 P2 占有了资源 rk1 而处于永远等待资源 rk2 状态。如此推下去，按假定系统只有有限个进程，即必有某个 n，在时刻 tn 时，进程 Pn 永远等待资源 rkn 的状态，而 rkn 必为前面的某一个进程 Pi 占用($1\leqslant i<n$)。于是，按照上述的按序分配策略，当 P2 占用了 rk1 后再申请 rk2 必有：

$$k1 < k2$$

依此类推，可得：

$$k2 < k3 < \cdots < ki < \cdots < kn$$

但是，由于进程 Pi 是占有了 rkn 却要申请 rki，那么，必定有：

$$kn < ki$$

这就产生了矛盾。所以按序分配策略可以防止死锁。

层次分配比静态分配在实现上要多花一点代价，但它提高了资源使用率。然而，如果一个进程使用资源的次序和系统内的规定各层资源的次序不同时，这种提高可能不明显。假如系统中的资源从高到低按序排列为：卡片输入机、行式打印机、卡片输出机、绘图仪和磁带机。若一个进程在执行中，较早地使用绘图仪，而仅到快结束时才用磁带机。但是，系统规定，磁带机所在层次低于绘图仪所在层次。这样，进程使用绘图仪前就必须先申请到磁带机，这台磁带机就在一长段时间里空闲着直到进程执行到执行结束前才使用，这无疑是低效率的。

3.6.5 死锁的避免

当不能防止死锁的产生时，如果能掌握并发进程中与每个进程有关的资源动态申请情况，仍然可以避免死锁的发生。只须在为申请者分配资源前先测试系统状态，若把资源分配给申请者会产生死锁的话，则拒绝分配，否则接受申请，为它分配资源。

我们看到死锁避免不是通过对进程随意强加一些规则，而是通过对每一次资源申请进行认真的分析来判断它是否能安全地分配。问题是：是否存在一种算法总能作出正确的选择从而避免死锁？答案是肯定的，但条件是必须事先获得一些特定的信息。本节我们将讨论使用 **银行家算法(banker's algorithm)** 避免死锁的方法。

3.6.5.1 资源轨迹图

避免死锁的主要方法是让系统处于安全状态,在图 3-7 中，我们看到一个处理两个进程和两种资源（打印机和绘图仪）的模型。横轴表示进程 A 的指令执行过程，纵轴表示进程 B 的指令执行过程。进程 A 在 I1 处请求一台打印机，在 I3 处释放，在 I2 处申请一台绘图仪，在 I4 处释放。进程 B 在 I5 到 I7 之间需要绘图仪，在 I6 到 I8 之间需要打印机。

图中的每一点都示出了两个进程的状态。初始点为 P，若 A 先运行，则在 A 执行一段指令后到达 q，在 q 点若 B 开始运行，则轨迹向垂直方向移动。在单处理机情况下，所有路径都只能是水平或垂直方向的。同时运动方向一定是向右或向上，而不会是向左或向下，因为进程的执行不可能后退。

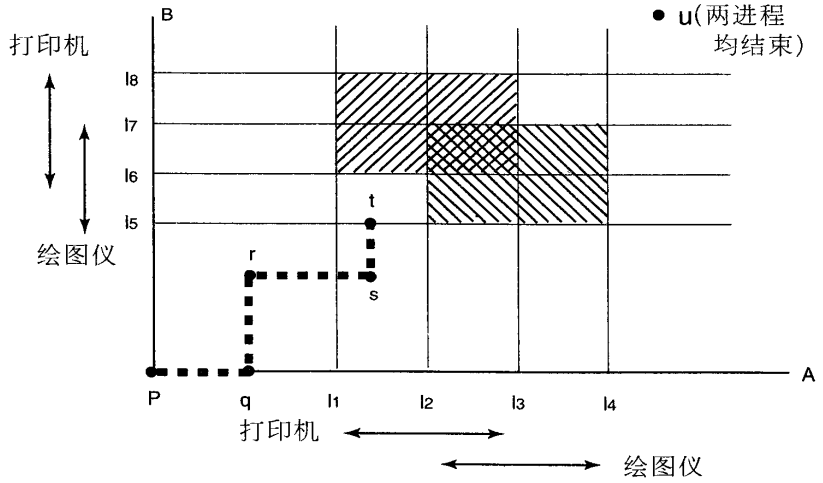


图 3-7　两个进程的资源轨迹图

当进程 A 由 r 向 s 移动，穿过 I1 线时，它请求打印机并获得。当进程 B 到达 t 时，它申请绘图仪。

图中的阴影部分很重要，打着从左下到右上斜线的部分表示在该区域中两个进程都拥有打印机，而互斥使用的规则决定了不可能进入该区域。同样的，另一种斜线的区域表示两个进程都拥有绘图仪，且同样不可进入。

如果系统一旦进入由 I1 和 I2 和 I5、I6 组成的矩形区域，那么最后一定会到达 I2 和 I6 交叉点，此时进程处在不安全区域,就会发生死锁。在该点处，A 申请绘图仪，B 申请打印机，而且这两种资源均已被分配。这整个矩形区域都是不安全的，因此绝不能进入这个区域。在 t 处唯一的办法是运行进程 A 直到I4，过了 I4 后则可以按任何路线前进，直到终点 u。

3.6.5.2 单种资源的银行家算法

Dijkstra（1965）提出了一种能够避免死锁的调度方法，称为银行家算法,它的模型基于一个小城镇的银行家，现将该算法描述如下：假定一个银行家拥有资金(资源),数量为Σ,被 N 个客户(进程)共享。银行家(操作系统)对客户提出下列约束条件,

每个客户必须预先说明自己所要求的最大资金量;

每个客户每次提出部分资金量申请和获得分配;

如果银行满足了客户对资金的最大需求量,那么,客户在资金运作后,应在有限时间内全部归还银行。

只要每个客户遵守上述约束，银行将保证做到：若一个客户所要求的最大资金量不超过Σ,则银行一定接纳该客户,并可处理他的资金需求；银行在收到一个客户的资金申请时,可能因资金不足而让客户等待,但保证在有限时间内让客户获得资金。在银行家算法中,客户可看作进程，资金可看作资源，银行可看作操作系统，这里叙述的是单资源银行家算法,还可以扩展到多资源银行家算法。在图 3-8（a）中我们看到 4 个客户，每个客户都有一个贷款额度，银行家知道不可能所有客户同时都需要大量贷款额，所以他只保留 10 个单位的资金来作为客户服务，而不是 22 个单位。到其所有的贷款（即所有的进程得到所需的全部资源并终止）。图 3-8（b）所示的状态是客户们各自做自己的生意，在某些时刻需要贷款。在某一时刻，具体情况如图 3-8（b）所示。客户已获得的贷款（已分配的资源）和可用的最大数额贷款称为与资源分配相关的系统状态。

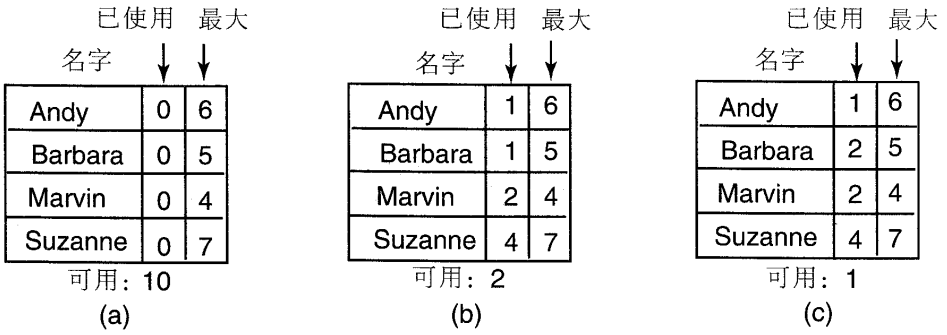


图 3-8　三种资源分配状态　(a) 安全　(b) 安全　(c) 不安全

一个状态被称为是安全的，其条件是存在一个状态序列能够使所有的客户均得到其所有的贷款（即所有的进程得到所需的全部资源并终止）。图 5-1（b）所示的状态是安全的，以使 Marvin 运行结束，然后释放所有的 4 个单位资金。如此这样下去便可以满足 Suzanne 或者 Barbara 的请求，等等。

考虑假如给 Barbara 另一个她申请的资源，如图 3-8（b），则我们得到如图 3-8（c）所示的状态，该状态是不安全的。如果忽然所有的客户都申请，希望得到最大贷款额，而银行家无法满足其中任何一个的要求，则发生死锁。不安全状态并不一定导致死锁，因为客户未必需要其最大贷款额度，但银行家不敢抱这种侥幸心理。

银行家算法就是对每一个请求进行检查，检查这次资源申请是否会导致不安全状态。若是，则不满足该请求；否则便满足。检查状态是否安全的方法是看他是否有足够的资源满足一个距最大需求最近的客户。如果可以，则这笔投资认为是能够收回的，然后接着检查下一个距最大需求最近的客户，如此反复下去。如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。

3.6.5.3 多种资源的银行家算法

资源轨迹图的方法很难被扩充到系统中有任意数目的进程、任意种类的资源，并且每种资源有多个实例的情况。但银行家算法可以被推广用来处理这个问题。图 3-9 示出了其工作原理。

在图 3-9 中我们看到两个矩阵。左边的显示出对 5 个进程分别已分配的各种资源数，右边的则显示了使各进程运行完所需的各种资源数。与单种资源的情况一样，各进程在执行前给出其所需的全部资源量，所以系统的每一步都可以计算出右边的矩阵。

图 3-9 最右边的三个向量分别表示总的资源 E、已分配资源 P，和剩余资源 A。

由 E 可知系统中共有 6 台磁带机，3 台绘图仪，4 台打印机和 2 台 CD-ROM。由 P 可知当前已分配了 5 台磁盘机，3 台绘图仪，2 台打印机和 2 台 CD-ROM。该向量可通过将左边矩阵的各列相加得到，剩余资源向量可通过从资源总数中减去已分配资源数得到。

检查一个状态是否安全的步骤如下：

(1) 查找右边矩阵是否有一行，其未被满足的设备数均小于或等于向量 A。如果找不到，则系统将死锁，因为任何进程都无法运行结束。

(2) 若找到这样一行，则可以假设它获得所需的资源并运行结束，将该进程标记为结束，并将资源加到向量 A 上。

(3) 重复以上两步，直到所有的进程都标记为结束。若达到所有进程结束，则状态是安全的，否则将发生死锁。

如果在第 1 步中同时存在若干进程均符合条件，则不管挑选哪一个运行都没有关系，因为可用资源或者将增多，或者在最坏情况下保持不变。

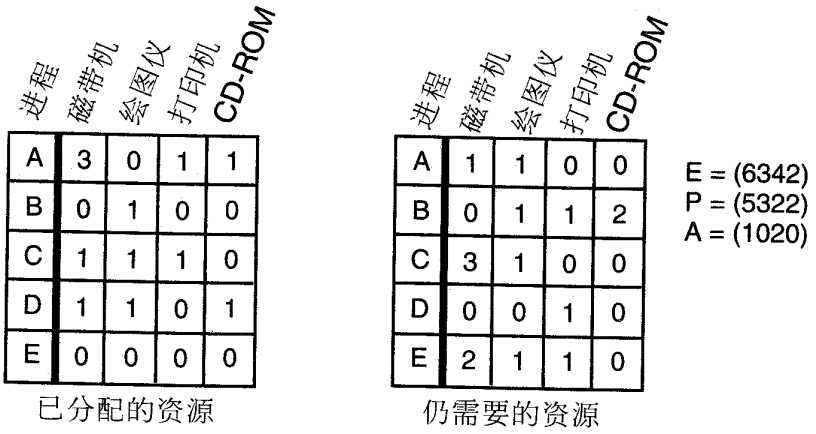


图 3-9　多种资源的银行家算法

图中所示的状态是安全的，因为进程 B 现在在申请一台打印机，可以满足它的请求，而且保持系统状态仍然是安全的（进程 D 可以结束，然后是 A 或 E，剩下的进程最后结束）。

假设进程 B 获得一台打印机后，E 试图获得最后的一台打印机，若分配给 E，可用资源向量将减到（1000），这时将导致死锁。则显然 E 的请求不能立即满足，必须延迟一段时间。

该算法最早由 Dijkstra 于 1965 年发表。从那之后几乎每本操作系统的专著都详细地描述它，许多论文的内容也围绕该算法，但很少有作者指出该算法缺乏实用价值。因为很少有进程能够在运行前就知道其所需资源的最大值，而且进程数不是固定的，往往在不断地变化，况且原本可用的资源也可能突然间变成不可用（如磁带机可能会坏掉）。

总之，死锁预防的方案过于严格，死锁避免的算法又需要无法得到的信息。如果你能想到一种理论上和实际中都适用的通用解法，那么就可以在计算机科学的杂志上发表一篇论文。

对于特殊的应用有许多很好的算法。例如在许多数据库系统中，常常需要将若干记录上锁然后进行更新。当有多个进程同时运行时，有可能发生死锁。

常用的一种解法是两阶段上锁法。第一阶段，进程试图将其所需的全部记录加锁，一次锁一个记录。若成功，则数据进行更新并解锁。若有些记录已被上锁，则它将已上锁的记录解锁并重新开始执行，该解法有点类似提前申请全部资源的方法。

但这种方法不通用，在实时系统和过程控制系统中不能够因为资源不可用而将进程中途终止并重新执行。同样，若一个进程已进行过网络消息的读写、更新文件、或其他不宜重复的操作，则将进程重新从头执行是不可接受的。该算法仅适用于那些在第一阶段可以随时停止并重新执行的程序。遗憾的是并非所有的应用都可以按这种方式组织。

3.6.5.4 银行家算法的数据结构和安全性测试算法

考虑一个系统有 n 个进程和 m 种不同类型的资源,现定义包含以下向量和矩阵的数据结构：

- 系统每类资源总数--该 m 个元素的向量为系统中每类资源的数量 Resource=(R_1,R_2,…,R_m)
- 每类资源未分配数量--该 m 个元素的向量为系统中每类资源尚可供分配的数量：Avilable=(V_1,V_2,…,V_m)
- 最大需求矩阵--每个进程对每类资源的最大需求量,C_{ij} 表示进程 P_i 需 R_j 类资源最大数

$$Claim=\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ & & & \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$$

- 分配矩阵—表示进程当前已分得的资源数,A_{ij} 表示进程 P_i 已分到 R_j 类资源的个数

$$Allocation=\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ & & & \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

于是下列关系式确保成立：

- $R_i=V_i+\sum A_{ki}$　对 i=1,..,m,k=1,..,n;　表示所有的资源要么已被分配、要么尚可分配
- $C_{ki} \leqslant R_j$　对 i=1,..,m,k=1,..,n;　表示进程申请资源数不能超过系统拥有的资源总数
- $A_{ki} \leqslant C_{ki}$　对 i=1,..,m,k=1,..,n;　表示进程申请任何类资源数不能超过声明的最大资源需求数

下面是一种死锁避免策略：系统中若要启动一个新进程工作,其对资源 Ri 的需求仅当满足下列不等式：

$$R_i \geqslant C_{(n+1)I} + \sum C_{ki} \quad 对\ i=1,..,m,k=1,..,n;$$

也就是说,应满足当前系统中所有进程对资源 R_i 的最大资源需求数加上启动的新进程的最大资源需求数不超过系统拥有的最大数。该算法考虑了最坏的情况、即所有进程同时要使用他们声明的最大资源需求数。根据上面的讨论,可以给出系统安全性定义：

在时刻 T0 系统是安全的,仅当存在一个进程序列 $P_{1,..},P_n$,对进程 $P_k(k=1,..,n)$满足公式

$$C_{ki}-A_{ki} \leqslant Available_i+\sum A_{ji} \qquad j=1,..,k-1;k=1,..,n;i=1,..,m$$

该序列称安全序列,其中公式左边表示进程 P_k 尚缺少的各类资源;$Available_i$ 是 T_0 时刻系统尚可用于分配且为 P_k 所想要的那类资源数;A_{ji} 的进程序列,则系统处于不安全状态。

显然,一个进程 P_k 所需资源若不能立即被满足,那么，在所有 $P_j(j=1,..,k-1)$运行完成后可以满足，然后 P_k 也能获得资源完成任务;当 P_k 释放全部资源后,P_{k+1} 也能获得资源完成任务;如此下去,直到最后一个进程完成任务，从 T_0 时刻起按这个进程序列运行,系统是安全的,绝不会产生死锁。

我们先用一个实例来说明系统所处的安全或不安全状态。如果系统中共有五个进程和 A、B、C 三类资源;A 类资源共有 10 个,B 类资源共有 5 个,C 类资源共有 7 个。在时刻 T_0,系统目前资源分配情况如下：

process	Allocation			Claim			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

每个进程目前还需资源为 Cki-Aki,

process	$C_{ki}-A_{ki}$		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

我们可以断言目前系统处于安全状态,因为,序列{P_1,P3,P4,P2,P0}能满足安全性条件。

现在假定进程 P1 又要申请 1 个 A 类资源和 2 个 C 类资源,所以,request1=(1,0,2)。为了判别这个申请能否立即准许,首先检查 request1≤Available、也就是比较(1,0,2) ≤(3,3,2),结果满足条件,我们尝试进行分配,得到了下面的新状态：

process	Allocation			Claim			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

我们要判定这个新状态是否安全?可以执行安全性测试算法，并且找到一个进程序列 {P1,P3,P4,P0,P2} 能满足安全性条件,因而,可正式把资源分配给进程 P1;然而,系统若处在下面状态中，进程 P4 请求资源(3,3,0),由于可用资源不足,申请被系统拒绝;此时，系统能满足进程 P0 的资源请求(0,2,0);但可以看出系统已处于不安全状态了。

银行家算法的基本思想是：系统中的所有进程进入进程集合,在安全状态下系统收到一个进程的资源请求后,先把资源试探性分配给它。现在,系统用剩下的可用资源和每个进程集合中其他进程还要的资源数作比较，在进程集合中找到剩余资源能满足最大需求量的进程,从而,保证这个进程运行完毕并归还全部资源。这时,把这个进程从集合中去掉，系统的剩余资源更多了,再反复执行上述步骤。最后,检查进程集合,若为空表明本次申请可行,系统处于安全状态,可真正实施本次分配;否则,有进程执行不完，系统处于不安全状态,本次资源分配暂不实施,让申请进程等待。下面是银行家算法的程序及简短说明。

```
Type state= record                          /*全局数据结构
          resource,available:array[0…m-1] of integer;
          claim,allocated:array[0…n-1,0…m-1] of inteher;
        end
```

```
                                        /*资源分配算法
  if alloc[i,*]+request[*]>claim[i,*]   then   <error>      /*申请量超过最大需求量
  else
      if request[*]>available[*]   then   <suspend process.>
      else                                                  /*模拟分配
         < define newstate by:
         allocation[i,*]:=allocation[i,*]+request[*]
         available[*]:=available[*]-request[*] >
      end;
      if safe(newstate)   then
      < carry out allocation>
      else
      <restore original state>
      < suspend process>
      end
  end
                                        /*安全性测试算法(banker's algorithm)
  function safe(state:s):boolean;
  var currentavail:array{0…m-1} of integer;
      rest:set of process;
  begin
  currentavail:=available;
  rest:={all process};
  possible:=true;
  while possible do
find a Pk in rest such that
       claim[k,*]-alloc[k,*]≤currentavail;
      if found   then
         currentavail:=currentavail+allocation[k,*];
         rest:=rest-[Pk];
      else
         possible:=false;
      end
     end;
     safe:=(rest=null)
 end.
```

再对上述算法作简短说明:

(1) 申请量超过最大需求量时出错,否则转(2);

(2) 申请量超过目前系统拥有的可分配量时,挂起进程等待，否则转(3);

(3) 系统对 P_i 进程请求资源作试探性分配、执行：

allocation[i,*]:=allocation[i,*]+request[*]

available[*]:=available[*]-request[*]

(4) 执行安全性测试算法(5),如果安全状态则承认试分配,否则抛弃试分配，进程 P_i 等待。

(5) 安全性测试算法

- 定义工作向量 currentavail 和布尔型标志 possible; 初始化让 currentavail:=available,possible:=true;
- 保持 possible:=true,从进程集合 rest 中找出 claim[k,*]-alloc[k,*]≤currentavail 的进程来,如找到,则释放这个进程 P_k 的全部资源、执行以下操作 currentavail:=currentavail+allocation[k,*],把 Pk 从进程集合中去掉 rest:=rest-[P_k];否则 possible:=false,停止执行算法;
- 最后查看进程集 rest,若为空集返回安全标记;否则返回不安全标记。

3.6.6 死锁的检测和解除

对资源的分配加以限制可以防止和避免死锁的发生，但这不利于各进程对系统资源的充分共享。解决死锁问题的另一条途径是死锁检测方法，这种方法对资源的分配不加任何限制，也不采取死锁避免措施，但系统定时地运行一个“死锁检查”程序，判断系统内是否已出现死锁，若检测到死锁则设法加以解除。下面介绍两种死锁检测方法。

第一种方法借助于死锁的安全性测试算法来实现。今定义布尔型向量 posible[k],k=1,..,n。检测死锁算法如下：

currentavail:=available;

如果 claim[k,*]-alloc[k,*]=0,则 possible[k]:=true;否则 possible:=false;

在 rest 中找一个进程 Pk,需满足条件：

(possible[k]=false&(request[*]≤currentavail)

若找到这样的 Pk 便转(4);否则转(5);

currnetavail:=currentavail+allocation;possible[k]:=true;然后转(3);

如果对 k=1,..,n 若 possible[k]=true 不成立,那么,系统出现了死锁,并且 possible[k]=false 的 Pk 为死锁进程。

另一种死锁检测方法是可设置两张表格来记录进程使用资源的情况。一张表记录每个阻塞的进程正在等什么资源，另一张表记录哪些进程占有了什么资源。任一进程申请资源时，先查该资源是否为它进程所占用，若该资源空闲，则把该资源分配给申请者登入占用表；若该资源被它进程占用，则登入进程等待资源表。

死锁检测程序定时检测这两张表，如果有进程 Pi 等待资源 rk，且 rk 被进程 Pj 占用，则说 Pi 和 Pj 具有“等待占用关系”，记为W（Pi，Pj）。死锁检测程序反复检测这两张表，可以列出所有的“等待占用关系”。当出现W（Pi ,Pj），W(pj, Pk)，……W（Pl，Pm），W(Pm，Pi)时，显然，系统中存在一组循环等待资源的进程：Pi，Pj，Pk，……，P1，Pm。也就是说出现了死锁。

下面介绍一种实现死锁检测的方法，把两张表格中记录的进程使用和等待资源的情况用一个矩阵A来表示：

其中 Bij 为 1 ：　当 Pi 等待被 Pj 占用的资源时厂

Bij 为 0 ：　当 Pi 和 Pj 不存在等待占用关系时

于是“死锁检测”可用 Warshall 的传递闭包算法检测是否有死锁发生。Warshall 的传递闭包算法对矩阵 A 构成传递闭包 A^*[bij]中的每一个 bij 是对 A [bij]执行如下算法得到的：

```
for k:=1 to n do
    for I:=1 to n do
    for j:=  to n do
            bij:=bij ∨ (bik ∧bkj)
```

其中 bik∧bki 表示当前有 P I 等待 Pk 所占的资源且 P k 等待 P j 所占的资源时取值为 1，也就是说 Pi 与 P j 有间接等待关系。Warshall 传递闭包算法循环检测了矩阵中 A 中的各个元素，把等待资源的关系（包括间接等待关系）都在传递闭包 A^*中表示出来。显然，当 A^*中有一个 bii=1(i= 1，2……,n)时，就说明存在着一组进程，它们循环等待资源，也即系统出现了死锁。

死锁的检测和解除往往配套使用,当死锁被检测到后，用各种办法解除系统的死锁，可采用以下一些办法。

- 立即结束所有进程的执行，并重新启动操作系统。方法简单,但以前工作全部作废,损失可能很大。
- 撤销陷于死锁的所有进程,解除死锁继续运行。
- 逐个撤销陷于死锁的进程,回收其资源,直至死锁解除。
- 剥夺陷于死锁的进程占用的资源，但并不撤销它, 直至死锁解除。
- 根据系统保存的 checkpoint,让所有进程回退,直到足以解除死锁。
- 当检测到死锁时，如果存在某些未卷入死锁的进程，而这些进程随着建立一些新的抑制进程能执行到结束，则它们可能释放足够的资源来解除这个死锁。

尽管检测死锁是否出现和发现死锁后实现恢复的代价大于防止和避免死锁所花的代价，但由于死锁不是经常出现的，因而这样做还是值得的，检测策略的代价依赖于频率，而恢复的代价是时间的损失。

3.6.7 混合策略

在一个操作系统中，为了保证不出现死锁，往往采用死锁的防止、避免和检测的混合策略。对不同类的资源采用不同的调度策略，以使整个系统不出现死锁。

如果对每一类资源 R 采用了某一种分配策略后，在任何时间都不存在永远等待 R 类资源的进程，则在此分类策略下 R 是非死锁资源类，简称 R 是非死锁资源类。如果M是由若干类资源组成的的资源组，对M中各类资源规定了调度策略后，若在任何时刻都不会有循环等待进程序列 L：

P1，P2，……，Pn

其中 Pi(1≤i≤n)占有M中某个资源 ri-1 而等待另一个资源 ri，且 rn = r0，则称资源组M是安全的。

现在我们来验证资源的安全性和非死锁性的一些性质：

性质 1：假设资源类R是非死锁的，资源组 M1 是安全的，则由资源类R和资源组M 1 组成的资源组M是安全的。

实际上，如果存在资源组 M 中资源的循环等待进程序列：

P1，P2，……，Pn

那么，只可能有两种情况。

- 情况 1：均不等待R中资源，它们一定都在等 M1 中资源，这和 M1 是安全的有矛盾。
- 情况 2：P1, P2, ……，Pn 中有某个 Pi 等待 R 中资源，这就和 R 是非死锁资源类有矛盾。

所以，两种情况都不可能出现，从而证明了M是安全的。

性质 2：假设资源组 M1 和 M2 具有如下性质：在任何时刻都不同时存在两个进程 P1 和 P2，它们分别占用 M1 和 M2 中资源而等待 M2 和 M1 中资源，则说 M1 和 M2 是无关的。两个无关的安全资源组 M1 和 M2 所组成的资源组 M 是安全的。

实际上，如果存在等待M中资源的循环等待进程序列：

$$P1,\ P2,\ \cdots\cdots,\ Pn$$

其中 Pi 占用资源 ri-1 等待资源 ri(i=1,2…,n)，且

$$rn = r0$$

不失一般性，可设 r0 属于 M1，因为 M1 是安全的，故 r0, r1，…, 不会全属于 M1，假设 ri 是第一个属于 M2 的，即 ri-1 属于 M1 而 ri 属于 M2。这样就有进程 Pi，它占用资源 ri-1(属于 M1)而等待资源 r(属于 M2)。如果 ri 至 rn 属于 M2，那么进程 Pn 占用资源 rn-1（属于 M2）而等待 r0（属于 M1）。如果 ri 至 rn 不全属于 M2，则必有某个资源 ri+k-1 属于 M2 而 ri+k 属于 M1，此时进程 Pi+k 占有资源 ri+k-1（属于 M2）而等待资源 ri+k(属于 M1)。无论哪种可能情况都与 M1 和 M2 是无关的假设相矛盾，所以 M 是安全的。

从上面的讨论，可以看到如果把系统的资源分成 r 类：

$$R1,\ R2,\ \cdots,\ Rr$$

和 m 个无关的资源组：

$$M1,\ M2,\ \cdots,\ Mm$$

对它们分别采用了不同的调度策略后，使 R1，R2，…，Rr 均为非死锁资源类；M1, M2，…, Mm 都是安全的，那么在这种混合调度策略下，系统全部资源组成的资源组 M：

$$M = M1\ U\ M2\ U\ \cdots\ U\ Mm\ U\ R1\ U\ R2\ U\ \cdots\ U\ Rr$$

将是安全的，因此系统不会发生死锁。

下面我们来列举一些简单的调度策略。

1）如果规定占用原进程在释放它所战胜的R类资源前不得申请任何资源，则R类资源是非死锁资源类。

实际上，任何申请R类资源的进程P，或者立即获得R类资源，或者P类资源均被其它进程占用。根据占用者在释放前不再申请其它资源，因此它必在有限长的时间内释放占用的资源。所以P不可能永远等待R类资源，即R是非死锁资源类。

2）如果允许从占用资源的进程强行夺取占用的资源来重新分配，在适当的时候再归还给它，则资源是非死锁的，并把它称为可抢占的。

实际上，因为进度对资源的最大需求量不得超过系统拥有的R类资源的总量，所以用抢占的办法总可以结束对R类资源的等待。这就是说，R是非死锁资源类。

3）如果规定占用R类资源的进程在释放它所占用的资源前不得申请R类资源或任何死锁资源，那么R是非死锁资源类。

可以采用与1）类似的办法验证。

4）如果规定占用R类资源的进程在释放它所占用的R类资源前不得申请资源，则R是安全的。

实际上，这样规定后就不存在占用资源并申请资源的进程了。因此，不可能存在等待R类资源的循环等待序列。所以，R是安全的。

5）如果将资源组M中的资源按序分配，则M是安全的。

6）如果对资源组M中的资源采用预先分配（静态分配），即在进程开始前就将它所要的属于M中的资源分配给它，那么M是安全的。

7）如果M是采用预先分配策略的资源组，那么任何资源组M和它是相关的。

习题

1. 叙述顺序程序设计的特点以及采用顺序程序设计的优缺点。
2. 程序并发执行为什么会失去封闭性和结果可再现性?
3. 叙述并发程序设计的特点以及采用并发程序设计的优缺点。
4. 解释并发进程的无关性和交往性。
5. 并发进程的执行可能产生与时间有关的错误，试各举一例来说明与时间有关错误的两种表现形式。
6. 解释进程的竞争关系和协作关系。
7. 试说明进程的互斥和同步两个概念之间的区别。
8. 什么是临界区和临界资源？对临界区管理的基本原则是什么？

9. 哪些硬件设施可以实现临界区管理？
10. 什么是信号量？如何对它进行分类。
11. 为什么 PV 操作均为不可分割的原语操作？
12. 从信号量和 PV 操作的定义，可以获得哪些推论？
13. 叙述 AND 型信号量机制的特点
14. 叙述一般信量机制的特点
15. 有三个并发进程：R 负责从输入设备读入信息，M 负责对信号加工处理；P 负责打印输出。今提供；
    1）一个缓冲区，可放置 K 个信息；
    2）二个缓冲区，每个可放置 K 个信息；
    3）试用 PV 操作写出三个进程正确工作的流程
16. 没有 n 个进程共享一个互斥段，如果：
    1）每次只允许一个进程进入互斥段；
    2）每次最多允许 m 个进程（m≤n）同时进入互斥段。
    试问：所采用的互斥信号量初值是否相同？信号量值的变化范围如何？
17. 三个进程并发活动，其算法描述如下，试分析是否有错，如有则指出原因并改正。

```
Begin
S：＝－1；
  Cobegin
  P1：Begin
        ……
        V（S）；
        End
  P2：Begin
        ……
        V（S）
        End；
  P3：Begin
        P（S）
        ……
        End
  Cobegin
End
```

18. 何谓管程？它有哪些属性？管程中的过程在执行中能被中断吗？为什么？
19. 试比较 Hanson 和 Hoare 两种管程实现方法。
20. 已经有 PV 操作可用作同步工具，为什么还要有消息传递机制。
21. 叙述信件，信箱和间接通信原语。
22. 简述消息缓冲通信机制的实现思想。
23. 什么是管道（pipeline）？如何通过管道机制实现进程间通信？
24. 有一阅览室，读者进入时必须先在一张登记表上登记，该表为每一座位列出一个表目，包括座号、姓名读者离开时要批销登记信息；假如阅览室共有 100 个座位。试用：1）信号量和 P．V 标作；2）管程；来实现用户进程的同步算法

25. 在一个盒子里，混装了数量相等的黑白围棋子。现在用自动分拣系统把黑子、白子分开，设分拣系统有二个进程 P1 和 P2，其中 P1 拣白子；P2 拣黑子。规定每个进程每次拣一子；当一个进程在拣时，不允许另一个进程去拣；当一个进程拣了一子时，必须让另一个进程去拣。试写出两进程 P1 和 P2 能并发正确执行的程序。
26. 管程的同步机制使用条件变量和 Wait 及 Signal，尝试为管程设计一种仅仅使用一个操作的同步机制。
27. 一个快餐厅有 4 种职员：（1）领班：接受顾客点菜；（2）厨师：准备顾客的饭菜；（3）打包工：将做好的饭菜打包；（4）出纳员－收款并提交食品。每个职员可被看作一个进程，试写出能让四类职员正确并发运行的程序。
28. 在信号量 S 上作 P．V 操作时，S 的值发生变化，当 S＞0、S=0、S〈 0 时，它们的物理意义是什么？
29. 二个并发进程并发执行，其中，A、B、C、D、E 是原语，试给出可能的并发执行路径。

```
Process P          Process Q
begin              begin
A；                C；
B；                D；
C；                end；
end；
```

是否在任何情况下，‘忙式等待’都比‘阻塞等待’方法的效率低？试解释之。

30. 证明信号量与管程的功能是等价的：

（1）用信号量实现管程；

（2）用管程实现信号量。

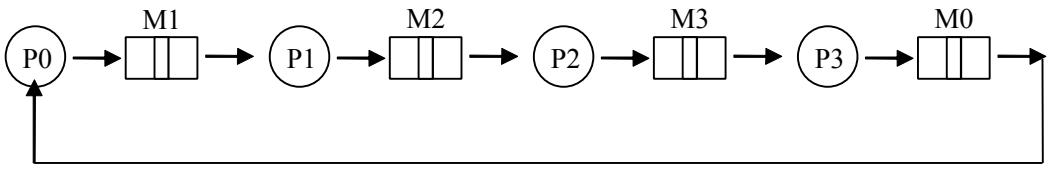
31. 面包房（Bakery Algorithm）算法，解决互斥问题的另一个算法：

```
Var choosing：array [0…n－1]  of boolean；
    number：array [0…n－1]  of integer；
Repeat
    Choosing [i]：=True；
    Number[i]：＝1＋max ( number [0]，…nuber  [n－1] )；
    Choosing[i]：=false；
    for j：＝0 to n－1 do
    begin
    while choosing [j] do {nothing}；
    while number[j]≠0 ∧（number [j],  j）＜（number[i]，i）
    do {nothing}；
end；
    {critical section}；
    number [i]：=0；
    {remaider}；
forever
```

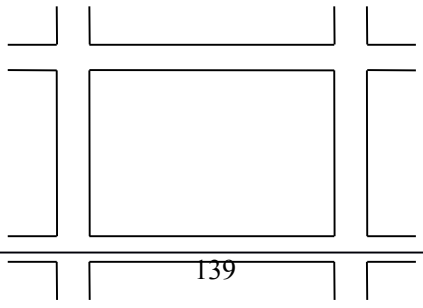
其中，choosing 和 Number 的初值分别为 false 和 0，每个数值中的第三个元素只能由进程 I 进行读和写，而其它进程只能读。（a，b）＜（c，d）的定义为：（a＜c）or（a=c∧

b＜d）试回答：（1）用自然语言描述该算法　（2）该算法能否实现互斥，为什么？

32. 试利用记录型信号量写出一个不会出现死锁的五个哲学家进餐问题的算法
33. 试利用 AND 型信号量写出生产者一消费者问题的算法
34. 试论述进程的低级通信工具和高级通信工具。
35. 如图所示，四个进程 Pi（i=0…3）和四个信箱 Mj（j=0…3），进程间借助相邻信箱传递消息，即 Pi 每次从 Mi 中取一条消息，经加工后送入 $M_{(i+1)mod4}$，其中 M0、M1、M2、M3 分别可存放 3、3、2、2 个消息。初始状态下，M0 装了三条消息，其余为空。试以 PV 操作为工具，写出 Pi（i=0…3）的同步工作算法。



36. 试用管程来解决 37 中的问题。
37. 在一个实时系统中，有两个进程 P 和 Q，它们循环工作。P 每隔 1 秒由脉冲寄存器获得输入，并把它累计到整型变量 W 上，同时清除脉冲寄存器。Q 每隔 1 小时输出这个整型变量的内容并将它复位。系统提供了标准例程 INPUT 和 OUTPUT 供 I / 0，提供了延时系统调用 Delay（Seconds）。试写出两个并发进程循环工作的算法。
38. 什么是死锁?试举日常生活中一例说明之。
39. 叙述产生死锁的必要条件。
40. 列举死锁的防止策略。
41. 何谓银行家算法?它是如何来避免死锁的?
42. 若系统有同类资源几个，被几个进程共享，问：当 m＞n 和 m≤n 时，每个进程最多可以请求多少个这类资源时，使系统一定不会发生死锁?
43. 系统有输入机和行印机各一台，今有两个进程都要使用它们，采用 PV 操作实现请求使用和归还资源后，还会产生死锁吗?说明理由，若是，则给出一种防止死锁的方法。
44. 假设三个进程共享四个资源，每个进程一次只能申请/释放一个资源，每个进程最多需要两个资源，证明该系统不会产生死锁。
45. N 个进程共享 M 个资源，每个进程一次只能申请/释放一个资源，每个进程最多需要 M 个资源，所有进程总共的需求少于 M+N 个资源，证明该系统此时不会产生死锁。
46. 一条公路两次横跨运河，两个运河桥相距 100 米，均带有闸门，以供船只通过运河桥 。运河和公路的交通均是单方向的。运河上的运输由驳船担负。在一驳船按近吊桥 A 时就拉汽笛警告，若桥上无车辆，吊桥就吊起，直到驳船尾 P 通过此桥为止。对吊桥 B 也按同样次序处理。一般典型的驳船长度为 200 米，当它在河上航行时是否会产生死锁?若会，说明理由，请提出一个防止死锁的办法，并用信号量来实现车辆与驳船的同步。
47. 如图所示，四条路上的汽车均单向直线行驶，小汽车长度为 L，大汽车长度为 2L，请举出交通死锁的例子。



1）产生死锁的必要条件中的哪些条件适用于此例。

2）举出一种简单原则，它能避免死锁。

3）若由计算机实现自动交通管理，试用同步工具来实现各方向汽车行驶的同步。

CH4 存储管理

存储管理是操作系统的重要组成部分，它负责管理计算机系统的重要资源主存储器。由于任何程序、数据必须占用主存空间后才能执行，因此存储管理直接影响系统的性能。主存储空间一般分为两部分：一部分是系统区，存放操作系统以及一些标准子程序，例行程序等;另一部分是用户区，存放用户的程序和数据等。存储管理主要是对主存储器中的用户区域进行管理,当然也包括对辅存储器的管理。目的是要尽可能地方便用户使用和提高主存储器的效率。具体地说，存储管理有下面几个方面的功能：

- 主存储空间的分配和去配。
- 地址转换和存储保护。
- 主存储空间的共享。
- 主存储空间的扩充。

本章在简介计算机存储器的层次之后，先后分析了多个连续存储管理方法、以及页式和段式存储管理方法，并进一步讨论了虚拟存储管理系统，最后介绍了 Intel 的 Pentium 芯片的存储管理支持，以及 Linux 的存储管理实例。

4.1 主存储器

4.1.1 存储器的层次

目前，计算机系统均采用分层结构的存储子系统，以便在容量大小、速度快慢、价格高低诸因素中取得平衡点，获得较好的性能价格比。计算机系统的存储器可以分为寄存器、高速缓存、主存储器、磁盘缓存、固定磁盘、可移动存储介质等 7 个层次组成了层次结构。如图 4-1 所示，越往上，存储介质的访问速度越快，价格也越高。其中，寄存器、高速缓存、主存储器和磁盘缓存均属于操作系统存储管理的管辖范畴，掉电后它们存储的信息不再存在。固定磁盘和可移动存储介质属于设备管理的管辖范畴，它们存储的信息将被长期保存。而磁盘缓存本身并不是一种实际存在的存储介质，它依托于固定磁盘，提供对主存储器存储空间的扩充。

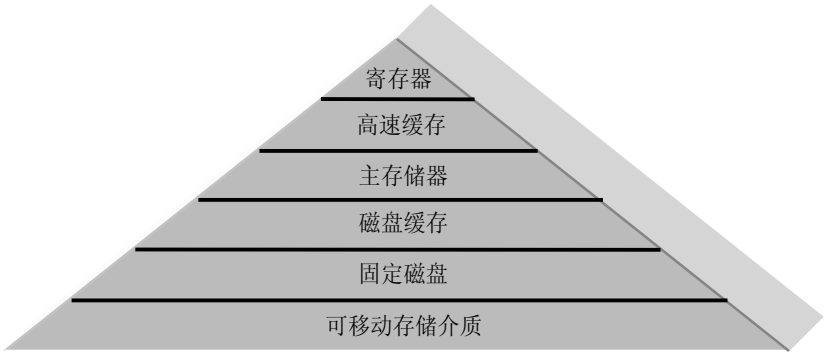


图 4-1 计算机系统存储器的层次

可执行的程序必须被保存在计算机的主存储器中，与外围设备交换的信息一般也依托于主存储器地址空间。由于处理器在执行指令时主存访问时间远大于其处理时间，寄存器和高速缓存被引入来加快指令的执行。

寄存器是访问速度最快但最昂贵的存储器，它的容量小，一般以字（word）为单位。一个计算机系统可能包括几十个甚至上百个寄存器，用于加速存储访问速度，如：寄存器存放操作数，或用地址寄存器加快地址转换速度。

高速缓存的容量稍大，其访问速度快于主存储器，利用它存放主存中一些经常访问的信息可以大幅度提高程序执行速度。例如，主存访问速度为 1us，高速缓存为 0.1us，假使访问信息在高速缓存中的几率为 50%，那么存储器访问速度可以提高到 0.55us。例如，某台计算机的存储器层次其配置如下：CPU 中的寄存器 100 个字；高速缓存 512KB，存取周期 15ns；主存储器 128MB，存取周期 60ns；磁盘容量 20GB，存取周期毫秒级；后援存储容量 1TB，存取周期秒级。这台机器足可用于中小型科研项目的开发，多层次组成的存储体系十分有效与可靠，能达到很高的性能/价格比。

由于程序在执行和处理数据时存在着顺序性、局部性、循环性和排他性，因此在程序执行是有时并不需要把程序和数据全部调入内存，而只需先调入一部分，待需要时逐步调入。这样，计算机系统为了容纳更多的算题数，或是为了处理更大批量的数据，就可以在磁盘上建立磁盘缓存以扩充主存储器的存储空间。算题的程序和处理的数据可以装入磁盘缓存，操作系统自动实现主存储器和磁盘缓存之间数据的调进调出，从而向用户提供了比实际主存存储容量大的多的存储空间。

4.1.2 快速缓存(Chaching)

快速缓存 Caching 是现代计算机结构中的一个重要部件，通常，运算的信息存放在主存中，每当使用它时，被临时复制到一个速度较快的 Cache 中。当 CPU 访问一组特定信息时，首先，检查它是否在 Cache 中，如果已存在，可直接从中取出使用;否则，要从主存中读出信息，通常认为这批信息被再次用到的概率很高，所以，同时还把主存中读出的信息复制到 Cache 中。在 CPU 的内部寄存器和主存之间建立了一个 Cache，而由程序员或编译系统实现寄存器的分配或替换算法，以决定信息是保存在寄存器还是在主存。有一些 Cache 是硬件实现的，如大多数计算机有指令 Cache，用来暂存下一条欲执行的指令，如果没有指令 Cache，CPU 将会空等若干个周期，直到下一条指令从主存中取出。同样道理，多数计算机系统的存储层次中设置了一个或多个高速数据 Cache，，我们并不关心纯硬件实现的 Cache，因为，它们并不需要操作系统控制。

由于高速 Cache 的容量有限，所以，Cache 的管理是一个重要的设计问题，仔细确定其大小和替换策略，能够做到 80%-99%的所需信息在 Cache 中找到，因而，系统性能极高。主存也可以看作是辅存的 Cache，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。文件系统的数据可能出现在存储器层次的不同级别中，在最高层，操作系统可在主存中维护一个文件数据的 Cache。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。

下面我们来讨论采用 Cache 后数据的一致性问题。在层次式存储结构中，相同的数据可能出现在不同的层次上。例如，考虑对文件 B 中的一个整数 A 做加 1 操作，假如文件 B 储存在磁盘上。首先发出 I/O 指令，它把 A 所在的盘块读到主存中，再做加法操作。跟着这个操作，可能把 A 的一个副本复制到 Cache, 而 A 的另一个副本复制到 CPU 寄存器。因此，A 的副本出现在若干个地方，当加 1 操作在 CPU 寄存器执行之后，在存储器各层次中，A 的值是不同的，仅当 A 的新值被写回磁盘之后，A 的值才会变成相同。

在每次仅有一个进程执行的环境中，上述安排没有什么问题，因为，对整数 A 的存取总是得到存储层次中最高层的值。然而，在一个多任务环境中，CPU 在许多进程之间来回切换，如果多个进程需要存取 A，那么，每个进程获得了 A 的最新被修改过的值。

多 CPU 环境中，情况变得较为复杂，除了维护 CPU 寄存器外，CPU 还包含一个局部数据 Cache,于是，A 的副本可能同时出现在若干个 Cache 中。所有的 CPU 都并发地执行，对 A 值在某个 Cache 中的修改，立即会影响到 A 所在的所有其它 Cache，这个问题称作缓冲一致性(Coherency)问题，这种一致性通常由硬件来保证。

在分布式环境中，情况变得尤其复杂，同一个文件的不同副本可能存储在地理上分散的不同计算机中，因为，各个副本在一个地方被修次，所有其它副本立刻就成为过时的了。有许多途径能确保文件一致性，这些将在分布式系统中讨论。

4.1.3 地址转换与存储保护

用户编写应用程序时,是从 0 地址开始编排用户地址空间的，我们把用户编程时使用的地址称为逻辑地址（相对地址）。而当程序运行时，它将被装入主存储器地址空间的某些部分，此时程序和数据的实际地址一般不可能同原来的逻辑地址一致，我们把程序在内存中的实际地址称为物理地址（绝对地址）。相应构成了用户编程使用的逻辑地址空间和用户程序实际运行的物理地址空间。

为了保证程序的正确运行，必须把程序和数据的逻辑地址转换为物理地址，这一工作称为地址转换或重定位。地址转换有两种方式，一种方式是在作业装入时由作业装入程序实现地址转换，称为静态重定位；另一种方式是在程序执行时实现地址转换，称为动态重定位。动态重定位必须借助于硬件的地址转换机构实现。

在计算机系统中可能同时存在操作系统程序和多道用户程序，操作系统程序和各个用户程序在主存储器中各有自己的存储区域，各道程序只能访问自己的工作区而不能互相干扰，因此操作系统必须对主存中的程序和数据进行保护，称为存储保护。同样存储保护的工作也必须借助硬件来完成。计算机中使用的存储保护硬件主要有：界地址和存储键方式等，将在存储管理详细介绍。

无论是地址转换机构还是存储保护，都必须借助于前面提到的地址寄存器以及一些硬件线路。用软件来模拟实现地址转换机构或存储保护都是不可行的，因为每一条命令都可能牵涉到地址转换和存储保护，模拟的结果将使得每一条指令的执行代价升级为一段程序的执行代价。

4.2 连续存储空间管理

4.2.1 单用户连续存储管理

单用户连续存储管理适用于单用户的情况，个人计算机和专用计算机系统可采用这种存储管理方式。采用单连续存储管理时主存分配十分简单，主存储器中的用户区域全部归一个用户作业所占用。在这种情况管理方式下，在任一时刻主存储器最多只有一道程序，各个作业的程序只能按次序一个个地装入主存储器。

单用户连续存储管理的地址转换多采用静态定位，静态地址定位是指程序执行之前(由装配程序)完成逻辑地址到物理地址的转换工作。如图 4-2 所示。具体来说，可设置一个栅栏寄存器（Fence Register）用来指出主存中的系统区和用户区的地址界限；通过装入程序把程序装入到从界限地址开始的区域，由于用户是按逻辑地址来编排程序的，所以当程序被装入主存时，装入程序必须对它的指令和数据进行重定位；存储保护也是很容易实现的，由装入程序检查其绝对地址是否超过栅栏地址，若是，则可以装入；否则，产生地址错误，不能装入，于是一个被装入的程序执行时，总是在它自己的区域内进行，而不会破坏系统区的信息。单用户连续存储管理的地址转换也可以采用动态定位，动态地址定位是指程序执行过程中,在 CPU 访问主存前,把要访问的程序和数据的逻辑地址转换成物理地址的转换工作。如图 4-3 所示。具体来说，可设置一个定位寄存器，它既用来指出主存中的系统区和用户区的地址界限，又作为用户区的基地址；通过装入程序把程序装入到从界限地址开始的区域，但不同时进行地址转换；程序执行过程中动态地将逻辑地址与定位寄存器中的值相加就可得到绝对地址；存储保护的实现很容易，程序执行中由硬件的地址转换机构根据逻辑地址和定位寄存器的值产生绝对地址，且检查该绝对地址是否存在所分配的存储区域内，若超出所分配的区域，则产生地址错误，不允许访问该单元中的信息。

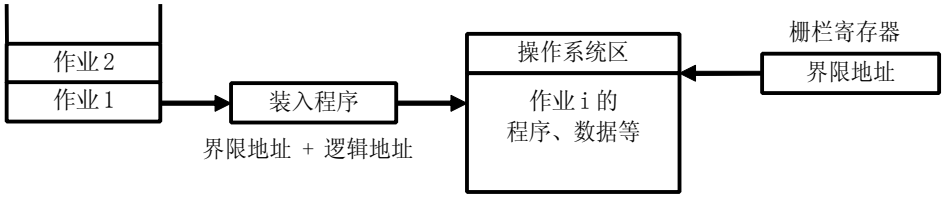


图 4-2　采用静态重定位的单连续存储管理

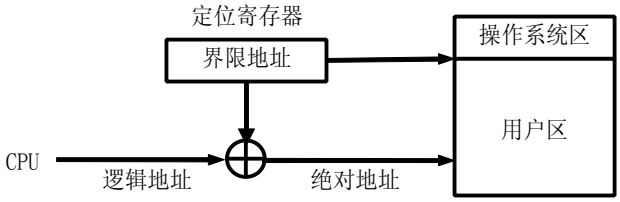


图 4-3　采用动态重定位的单连续存储管理

由于单用户连续存储管理只适合单道程序系统，采用这种管理有几个主要缺点：

- 当正在执行的程序因等待某个事件，比如，等待从外部输入数据，处理器便处于空闲状态；
- 不管用户作业的程序和数据量的多少，都是一个作业独占主存储空间，这就可能降低存储空间的利用率；
- 计算机的外围设备利用率不高。

在 70 年代由于小型计算机和微型计算机的主存容量不大，所以单用户连续存储管理曾得到了广泛的应用。例如 IBM7094 的 FORTRAN 监督系统，IBM1130 磁盘监督系统，MIT 兼容分时系统 CISS 以及微型计算机 cromemco 的 CDOS 系统，Digital Research 和 Dyhabyte 的 CP/M 系统，DJS0520 的 0520FDOS 等等均采用单用户连续存储管理。

4.2.2 固定分区存储管理

分区存储管理的基本思想是给进入主存的用户进程划分一块连续存储区域，把进程装入该连续存储区域,使各进程能并发执行,这是能满足多道程序设计需要的最简单的存储管理技术。

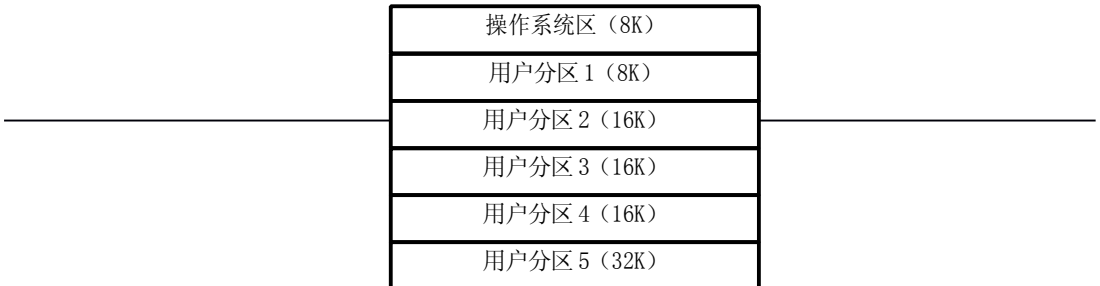


图 4-4　固定分区存储管理示意

固定分区(fixed partition)存储管理是预先把可分配的主存储器空间分割成若干个连续区域，每个区域的大小可以相同，也可以不同。如图 5-4 所示。为了说明各分区的分配和使用情况，存储管理需设置一张“主存分配表”，该表如图 4-5 所示。

分区号	起始地址	长度	占用标志
1	8K	8K	0
2	16K	16K	Job1
3	32K	16K	0
4	48K	64K	0
5	64K	32K	Job2
6	96K	32K	0

图 4-5　固定分区存储管理的主存分配表

主存分配表指出各分区的起始地址和长度，表中的占用标志位用来指示该分区是否被占用了，当占用的标志位为“0”时，表示该分区尚未被占用。进行主存分配时总是选择那些标志为“0”的分区，当某一分区分配给一个作业后，则在占用标志栏填上占用该分区的作业名，在图 4-5 中，第 2、5 分区分别被作业 Job1 和 Job2 占用，而其余分区为空闲。

由于固定分区存储管理是预先将主存分割成若干个区，如果分割时各区的大小是按顺序排列的，如图 4-4，那么固定分区存储管理的主存分配算法十分简单，有兴趣的同学可以把它作为课后练习。

固定分区存储管理的地址转换可以采用静态定位方式，装入程序在进行地址转换时检查其绝对地址是否在指定的分区中，若是，则可把程序装入，否则不能装入，且应归还所分得的存储区域。固定分区方式的主存去配很简单，只需将主存分配表中相应分区的占用标志位置成“0”即可。

固定分区存储管理的地址转换也可以采用动态定位方式。如图 4-6 所示，系统专门设置一对地址寄存器——上限/下限寄存器；当一个进程占有 CPU 执行时，操作系统就从主存分配表中取出相应的地址占有上限/下限寄存器；硬件的地址转换机构根据下限寄存器中保存的基地址 B 与逻辑地址得到绝对地址；硬件的地址转换机构同时把绝对地址和上限/下限寄存器中保存的相应地址进行比较，而实现存储保护。

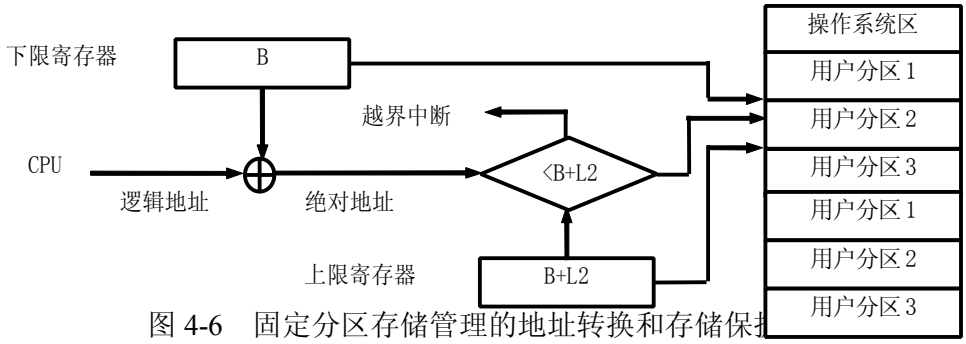


图 4-6　固定分区存储管理的地址转换和存储保护

作业进入分区有两种排队策略:一是每个分区排一个等待处理的队列,随着等待处理作业大小不均,导致有的分区空闲而有的分区忙碌;二是所有等待处理的作业排成一个进程队列,当调度其中一个进入分区运行时,选择可容纳它的最小可用分区,以充分利用主存。采用固定分区存储管理有许多缺点，首先,由于预先规定了分区大小,使得大程序无法装入，用户不得不采用复盖等技术补救,不但加重用户负担,而且极不方便。其次,主存空间的利用率不高，例如图 4-5 中若 Job1 和 Job2 两个作业实际只是 10 K 和 18 K 的主

存，但它们却占用了 16 K 和 32 K 的区域，共有 20 K 的主存区域占而不用浪费了空间。最后,因为分区的数目是在系统生成时确定局的,这就限制了并发运行的进程数。然而这种方法实现简单，因此，对于程序大小和出现频繁次数已知的情形，还是较合适的。例如 IBM 的 OS/MFT，它是任务数固定的多道程序设计系统，它的主存分配就采用固定分区方式。

4.2.3 可变分区存储管理

4.2.3.1 主存空间的分配和去配

可变分区(variable partition) 存储管理是按作业的大小来划分分区。系统在作业装入主存执行之前并不建立分区,当要装入一个作业时，根据作业需要的主存量查看主存中是否有足够的空间，若有，则按 需要量分割一个分区分配给该作业；若无，则令该作业等待主存空间。由于分区的大小是按作业的实际需要量来定的，且分区的个数也是随机的，所以可以克服固定分区方式中的主存空间的浪费,有利于多道程序设计，实现了多个作业对内存的共享，进一步提高了内存资源利用率。

随着作业的装入、撤离，主存空间被分成许多个分区，有的分区被作业占用，而有的分区是空闲的。当一个新的作业要求装入时，必须找一个足够大的空闲区，把作业装入该区，如果找到的空闲区大于作业需要量，则作业装入后又把原来的空闲区分成两部分，一部分给作业占用了；另一部分又分成为一个较小的空闲区。当一个作业运行结束撤离时，它归还的区域如果与其它空闲区相邻，则可合成一个较大的空闲区，以利大作业的装入。采用可变分区方式的主存分配示例如图 4-7。

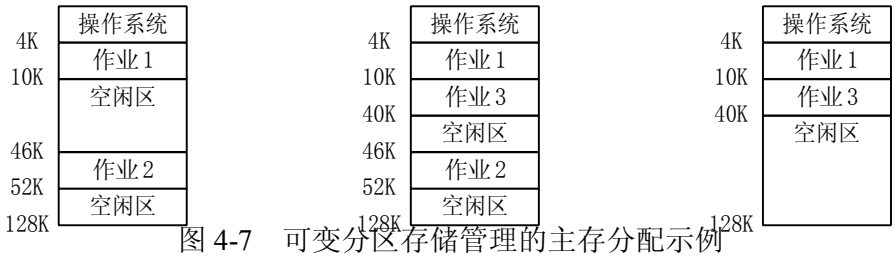


图 4-7　可变分区存储管理的主存分配示例

从图中可以看出，主存中分区的数目和大小随作业的执行而不断改变。为了方便主存的分配和去配，主存分配表可由两张表格组成，一张称”已分配区表”，另一张是”未分配区表”，如图 4-8。

分区号	起始地址	长度	标志
1	4K	6K	Job1
2	46K	6K	Job2

（a）已分配区表

分区号	起始地址	长度	标志
1	10K	36K	未分配
2	52K	76K	未分配

（b）未分配区表

图 4-8　可变分区存储管理的主存分配表

图 4-8 的两张表的内容是按图 5-6 最左边的情况填写的，当要装入长度为 30K 的作业时，从未分配表中可找一个足够容纳它的长度 36K 空闲区，将该区分成两部分，一部分为 30K，用来装入作业 3，成为已分配区；另一部分为 6K，仍是空闲区。这时，应从已分配区表中找一个空栏目登记作业 3 占用的起址、长度，同时修改未分配区表中空闲区的长度和起址。当作业撤离时则已分配区表中的相应状态改成“空”，而将收回的分区登记到未分配表中，若有相邻空闲区则将其连成一片后登记。可变分区的回收算法较为复杂,当一个进程 X 撤离时,可分成以下四种情况:其邻近都有进程(A 和 B),一边有进程(A 或 B),两边均为空闲区(黑色区域)。回收后如下图 4-9 右边所示,同时应修改主存分配表或链表。

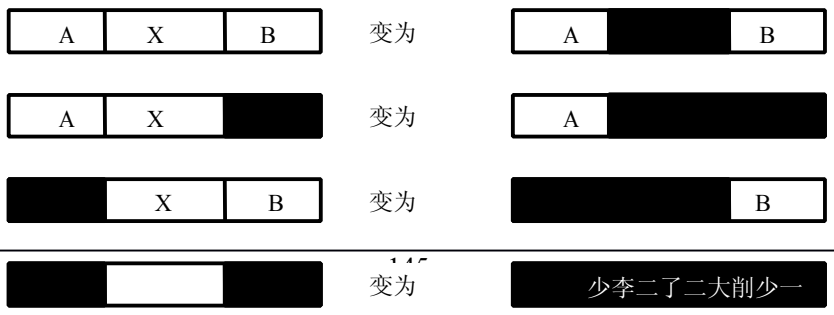


图 4-9 可变分区的回收情况

由于分区的个数不定，采用链表是较好的另一种空闲区管理方法。每个内存空闲区的开头单元存放本空闲区长度及下一个空闲区的起始地址,于是所有的空闲区都被链接起来,系统设置一个第一块空闲区地址指针,让它指向第一块空闲区的地址。使用时,沿链查找并摘下一个长度能满足申请要求的空闲区交给进程,修改链接;归还时,把空闲区链入空闲区链即可。空闲区链表管理比空闲区表格管理较为复杂,但其优点是链表自身不占用存储单元。

不论是空闲区链表管理还是空闲区表格管理,链和表中的空闲区都可按一定规则排列,例如按空闲区从大到小排或从小到大排,以方便空闲区的查找和回收。

常用的可变分区管理的分配算法有：

1）最先适用分配算法。对可变分区方式可采用最先适用分配算法，每次分配时，总是顺序查找未分配表或链表，找到第一个能满足长度要求的空闲区为止。分割这个找到的未分配区，一部分分配给作业，另一部分仍为空闲区。这种分配算法可能将大的空间分割成小区，造成较多的主存“碎片”。作为改进，可把空闲区按地址从小到大排列在未分配表或链表中，于是为作业分配主存空间时，尽量利用了低地址部分的区域，而可使高地址部分保持一个大的空闲区，有利于大作业的装入。但是，这给回收分区带来一些麻烦，每次收回一个分区后，必须搜索未分配区表或链表来确定它在表格或链表中的位置且要移动相应的登记项。

2）最优适应分配算法。可变分区方式的另一种分配算法是最优适应分配算法，它是从空闲区中挑选一个能满足作业要求的最小分区，这样可保证不去分割一个更大的区域，使装入大作业时比较容易得到满足。采用这种分配算法时可把空闲区按长度以递增顺利排列，查找时总是从最小的一个区开始，直到找到一个满足要求的分区为止。按这种方法，在回收一个分区时也必须对分配表或链表重新排列。最优适应分配算法找出的分区如果正好满足要求则是最合适的了，如果比所要求的略大则分割后使剩下的空闲区就很小，以致无法使用。

3）最坏适应分配算法。最坏适应分配算法是挑选一个最大的空闲区分割给作业使用，这样可使剩下的空闲区不至于太小，这种算法对中、小作业是有利的。

下面简单地比较一下三种查找和分配算法。从搜索空闲区速度来看，最先适用分配算法具有较好的性能,但如果空闲区按从小到大排列,则最先适用分配算法等于最优适应分配算法。反之，如果空闲区按从大到小排列,则最先适用分配算法等于最坏适应分配算法。空闲区按从小到大排列时，最先适用分配算法能尽可能使用低地址,从而在高地址空间有较多较大的空闲区来容纳大的进程。最优适应分配算法的主存利用率最好,因为,它把刚好或最接近申请要求的空闲区分给进程;但是它可能会导致空闲区分割下来的部分很小。在处理某种作业序列时，最坏适应分配算法可能性能最佳,因为它选择最大空闲区,使得分配后剩余下来的空闲区不会太小,仍能用于再分配。

4.2.3.2 地址转换与存储保护

对可变分区方式采用动态定位装入作业，作业程序和数据的地址转换是硬件完成的。硬件设置两个专门控制寄存器：基址寄存器和限长寄存器。基址寄存器存放分配给作业使用的分区的最小绝对地址值，限长寄存器存放作业占用的连续存储空间长度。

当作业占有 CPU 运行后，操作系统可把该区的始址和长度送入基址寄存器和限长寄存器，启动作业执行时由硬件根据基址寄存器进行地址转换得到绝对地址，地址关系转换如图 4-10。

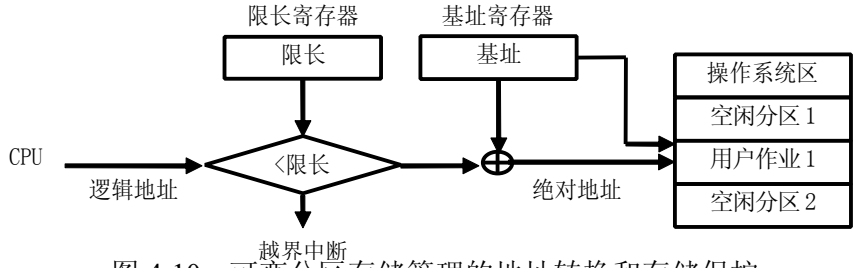


图 4-10 可变分区存储管理的地址转换和存储保护

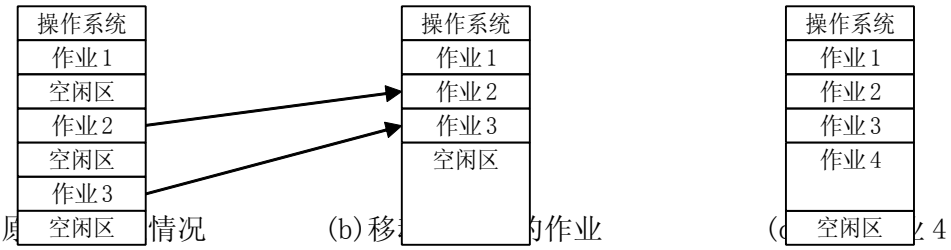
当逻辑地址小于限长值时，则逻辑地址加基址寄存器值就可获得绝对值地址；当逻辑地址大于限长值时，表示作业欲访问的地址超出了所分得的区域，这时，不允许访问，达到了保护的目的。

在多道程序设计系统中，硬件只需设置一对基址/限长寄存器，进程在执行过程中出现等待时，操作系统把基址/限长寄存器的内容随同该进程的其它信息，如 PSW，通用寄存器等一起保存起来。当进程被选中执行时，则把选中作业的基址/限长值再送入基址/限长寄存器。CDC6600 世界上最早的巨型机便采用这一方案。

如果每个进程只能占用一个分区，那么就不允许各个进程之间有公共的区域，这样，当几个进程共享一个例行程序时就只好在各自的主存区域各放一套，从而主存利用率差。所以,有些计算机硬件提供多对基址/限长寄存器，允许一个进程占用两个或多个分区。可以规定某对基址/限长寄存器的区域是共享的，用来存放共享的程序和常数，当然对共享区域的信息只能读出不能写入。于是几个进程共享的例行程序就可放在限定的公用区域中,而让进程的共享部分具有相同的基址/限长值就行了。

4.2.3.3 移动技术

当在未分配表中找不到一个足够大的空闲区来装入作业时，可采用移动技术把在主存中的作业改变存放区域，同时修改它们的基址 / 限长值，从而使分散的空闲区汇集成一片而有利于作业的装入。移动分配的示例如图 4-11。



(a)原情况　　(b)移的作业　　(c)之4

图 4-11　移动分配示例

移动虽可汇集主存的空闲区，但也增加了系统的开销，而且不是任何时候都能对一道程序进行移动的。由于外围设备与主存储器交换信息时，通道总是按已经确定的主存绝对地址完成信息传输,所以当一道程序正在与外围设备交换数据时往往不能移动，故应尽量设法减少移动。比如，当要装入一道作业时总是先挑选不经移动就可装入的作业；在不得不移动时力求移动的道数最少。采用移动技术分配主存的算法如下：

作业 i 请求分配 xK 主存
步骤 1　　查主存分配表
步骤 2　　若有大于 xK 主存的空闲区，则转步骤 5
步骤 3　　若空闲区总和小于 xK，则作业 i 等待主存资源
步骤 4　　移动主存中其它信息；
　　修改主存分配表中有关项；
　　修改被移动者的基址/限长；
步骤 5　　分配 xK 主存；
　　修改主存分配表中有关项；
　　置作业 i 的基址/限长；
步骤 6　　算法结束

移动也为作业执行过程中扩充主存提供了方便。一道作业在执行中要求增加主存量时，只需适当移动邻近的作业就可增加它所占的连续区的长度。移动后的基址值和扩大后的限长值都应作相应的修改。

在多道程序系统中，如果允许作业(进程)在执行过程中动态扩充主存，那么有时会遇到死锁问题。例如，主存已被 A、B 两道作业全部占用了，当 A 申请主存时，由于空闲区不够不能继续执行下去而处于等待主存资源状态；以后，B 占有处理器执行，执行中也要申请主存，同样，它也变成等待主存资源状态。显然，A 、B 的等待永远不能结束，这就是死锁。为了避免死锁，可以考虑发生 A 、B 竞争主存资源而都得不到满足的情形，操作系统将 A 或 B 送出主存，存到辅助存储器中，然后让留在主存的那道作业获得主存资源并继续执行下去，直到它归还主存后，再将送出去的 作业调回来。

为了接纳送出去的作业，辅助存储器中要留下一个暂存区域称映像区。为使这个区域小一些，可以考虑每次总是将最小的一道作业从主存送出去。假定主存中最多容纳 n 道作业，主存能提供给一道作业的最大容量为 V（一般说来，V 为主存用户区域的容量），那么，送出去的作业的辅助存储器区域大小为：

$$(1/2 + 1/3 + \text{---} + 1/n) \cdot V$$

就足够了。因为当主存中有 n 道作业时，送出去的那道所占主存量一定不超过 1 /n • V。而最坏的情况是主存中开始有 n 道，发生竞争主存后送出一道，剩下 n-1 道，在作业执行中又发生竞争主存，再送出一道，这样直到最后两道时发生竞争主存还要送出一道，从而

得出上式个。

4.3 分页式存储管理

4.3.1 分页式存储管理的基本原理

用分区方式管理的存储器，每道程序总是要求占用主存的一个或几个连续存储区域，作业或进程的大小仍受到分区大小或内存可用空间的限制,因此，有时为了接纳一个新的作业而往往要移动已在主存 的信息。这不仅不方便，而且开销不小。采用分页存储器既可免去移动信息的工作，又可尽量减少主存的碎片。分页式存储管理的基本原理如下：

1、页框：物理地址分成大小相等的许多区,每个区称为一块(又称页框 page frame);

2、页面：逻辑地址分成大小相等的区, 区的大小与块的大小相等,每个区称一个页面(page)。

3、逻辑地址形式：与此对应，分页存储器的逻辑地址由两部分组成：页号和单元号。逻辑地址格 式如下：

页　号	单　元　号

采用分页式存储管理时，逻辑地址是连续的。所以，用户在编制程序时仍只须使用顺序的地址，而不必考虑如何去分页。由地址结构和操作系统管理的需要来决定页面的大小，从而，也就确定了主存分块的大小。用户进程在内存空间中的每个页框内的地址是连续的, 但页框和页框之间的地址可以不连续。存储地址由连续到离散的变化, 为以后实现程序的”部分装入、部分对换” 奠定了基础。

4、页表和地址转换：在进行存储分配时，总是以块(页框)为单位进行分配，一个作业的信息有多少页，那么在把它装入主存时就给它分配多少块。但是，分配给作业的主存块是可以不连续的，即作业的信息可按页分散存放在主存的空闲块中，这就避免了为得到连续存储空间而进行的移动。那么，当作业的程序和数据被分散存放后，作业的页面与分给的页框如何建立联系呢?页式虚拟地址如何变换成页框物理地址呢?作业的物理地址空间由连续变成分散后,如何保证程序正确执行呢?采用的办法是动态重定位技术,让程序的指令执行时作地址变换,由于程序段以页为单位,所以,我们给每个页设立一个重定位寄存器, 这些重定位寄存器的集合便称页表(page table)。页表是操作系统为每个用户作业建立的,用来记录程序页面和主存对应页框的对照表, 页表中的每一栏指明了程序中的一个页面和分得的页框的对应关系。通常为了减少开销,不是用硬件,而是在主存中开辟存储区存放页表,系统中另设一个页表主存起址和长度控制寄存器(page table control register),存放当前运行作业的页表起址和页表长,以加快地址转换速度。每当选中作业运行时，应进行存储分配，为进入主存的每个用户作业建立一张页表，指出逻辑地址中页号与主存中块号的对应关系，页表的长度随作业的大小而定。同时页式存储管理系统还建立一张作业表，将这些作业的页表进行登记，每个作业在作业表中有一个登记项。作业表和页表的一般格式如图 4-12。然后, 借助于硬件的地址转换机构，在作业执行过程中按页动态定位。调度程序在选择作业后，从作业表的登记项中得到被选中作业的页表始址和长度，将其送入硬件设置的页表控制寄存器。地址转换时，只要从页表控制寄存器就可以找到相应的页表，再按照逻辑地址中的页号作索引查页表，得到对应的块号，根据关系式：

$$\text{绝对地址} = \text{块号} \times \text{块长} + \text{单元号}$$

计算出欲访问的主存单元的地址。因此，虽然作业存放在若干个不连续的块中，但在作业执行中总是能按正确的地址进行存取。

页表

页号	块号
第 0 页	块号 1
第 1 页	块号 2
…	…

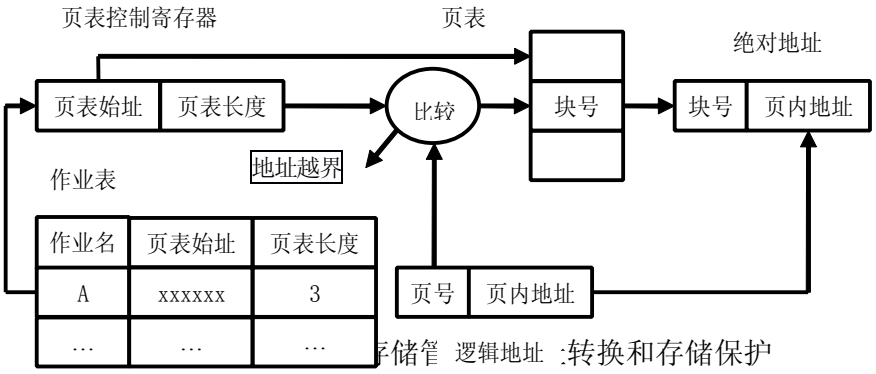
作业表

作业名	页表始址	页表长度
A	XXX	XX
B	XXX	XX
…	…	…

图 4-12　页表和作业表的一般格式

图 4-13 给出了页式存储管理的地址转换和存储保护，根据地址转换公式：块号 ×块长＋单元号，在实际进行地址转换时，只要把逻辑地址中的单元号作为绝对地址中的低地址部分，而根据页号从表中查

得的块号作为绝对地址中的高地址部分，就组成了访问主存储器的绝对地址。



整个系统只有一个页表控制寄存器，只有占用 CPU 者才占有页表控制寄存器。在多道程序中，当某道程序让出处理器时，应同时让出页表控制寄存器。

4.3.2 相联存储器和快表

页表可以存放在一组寄存器中，地址转换时只要从相应寄存器中取值就可得到块号，这虽然方便了地址转换，但硬件花费代价太高，如果把页表放在主存中就可降低计算机的成本。但是，当要按给定的逻辑地址进行读/写时，必须访问两次主存。第一次按页号读出页表中相应栏内容的块号，第二次根据 计算出来的绝对地址进行读/写，降低了运算速度。

为了提高运算速度，通常都设置一个专用的高速存储器，用来存放页表的一部分，这种高速存储器称为相联存储器（associative memory），存放在相联存储器中的页表称快表。相联存储器的存取时间是远小于主存的，但造价高，故一般都是小容量的，例如 Intel 80486 的快表为 32 个单元。

根据程序执行局部性的特点，即它在一定时间内总是经常访问某些页，若把这些页登记在快表中，无疑地将大大加快指令的执行速度。快表的格式如下：

页　号	块　号
----	----
页　号	块　号

它指出已在快表中的页及其对应主存的块号。有了快表后，绝对地址形成的过程是，按逻辑地址中的页号查快表，若该页已登记在快表中，则由块号和单元号形成绝对地址；若快表中查不到对应页号，则再查主存中的页表而形成绝对地址，同时将该页登记到快表中。当快表填满后，又要在快表中登记新页时，则需在快表中按一定策略淘汰一个旧的登记项，最简单的策略是“先进先出”，总是淘汰最先登记的那一页。

采用相联存储器的方法后，地址转换时间大大下降。假定访问主存的时间为 100 毫微秒，访问相联存储器的时间为 20 毫微秒，相联存储器为 32 个单元时查快表的命中率可达 90%，于是按逻辑地址进行存取的平均时间为：

$$（100＋20）×90\%＋(100+100+20)×(1-90\%)＝130\ 毫微秒$$

比两次访问主存的时间 100 毫微秒×2+20＝220 毫微秒下降了四成多。

同样，整个系统也只有一个相联存储器，只有占用 CPU 者才占有相联存储器。在多道程序中，当某道程序让出处理器时，应同时让出相联存储器。由于快表是动态变化的，所以让出相联存储器时应把快表保护好以便再执行时使用。当一道程序占用处理器时，除置页表控制寄存器外还应将它的快表送入相联存储器。

4.3.3 分页式存储空间的分配和去配

分页式存储管理把主存的可分配区按页面大小分成若干块，主存分配以块为单位。最简单的办法可用一张位示图来记录主存分配情况，指出已分配的块和尚未分配的块以及当前有多少空闲块。该表可存放在主存的专门区域中,表格的每一位与一个物理块对应，用 0/1 表示对应块为空闲/已占用，用另一专门字记录当前空闲块数，如图 4-14。

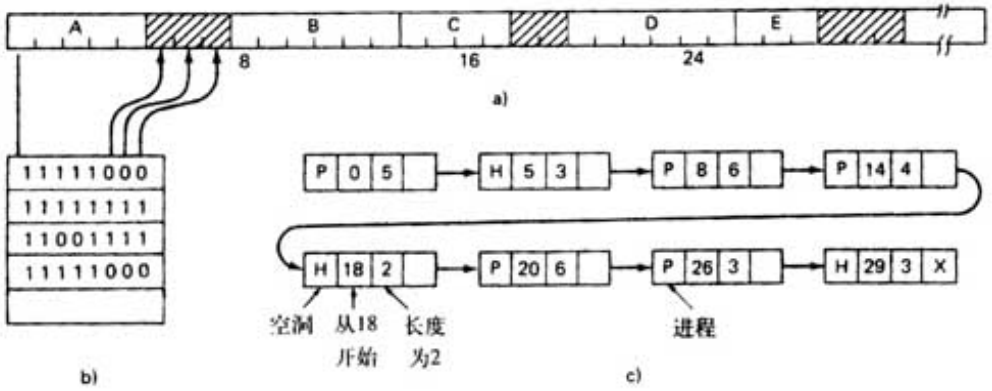


图 4-14　位示图

分页式存储管理的页框分配算法如下：进行主存分配时，先查空闲块数能否满足作业要求，若不能满足,令该作业等待；若能满足，则查位示图，找出为“0”的那些位，置上占用标志，从空闲块数中减去本次占用块数，按找到的位计算出对应的块号，建立该作业的页表。当一个作业执行结束，归还主存时，则根据归还的块号，计算出在位示图中的位置，将占有标志清成“0”，归还块数加入到空闲块数中。图 5-12(c)是内存分配的链表方法,表中每一项都含以下内容:是进程占用区(P)还是空闲区(H)、起始地址、长度和指向下一表项的指针。

分页存储管理能方便地实现多个作业共享程序和数据。在多道程序系统中，编译程序、编辑程序、解释程序、公共子程序、公用数据等都是可共享的，这些共享的信息在主存中只要保留一个副本。

页的共享可大大提高主存空间的利用率，例如一个编辑程序共 30K，现有 10 个作业，他们所处理的数据平均为 5K，如果不采用共享技术的话，那么 10 个作业共需 350K 主存，而共享编辑程序的话，则只需 80K 主存。

在实现共享时，必须区分数据的共享和程序的共享。实现数据共享时，可允许不同的作业对共享的数据页用不同的页号，只要让各自页表中的有关表目指向共享的数据信息块就行了。实现程度共享时，情况就不同了，由于页式存储结构要求逻辑地址空间是连续的，所以程序运行前它们的页号是确定的。现假定有一个被共享的编辑程序 EDIT，其中含有转移指令，转移指令中的转移地址必须指出页号和单元号，如果是转向本页，则页号应与本页的页号相同。现在若有两个作业共享这个 EDIT 程序，假定一个作业定义它的页号为 3，另一作业定义它的页号为 5，然而在主存中只有一个 EDIT 程序，它要为两个作业以同样的方式服务，这个程序一定是可再入的，于是转移地址中的页号不能按作业的要求随机的改成 3 或 5，因此对共享程序必须规定一个统一的页号。当共享程序的作业数增多时，要规定一个统一的页号是较困难的。

实现信息共享必须解决共享信息的保护问题。通常的办法是在页表中增加一些标志位，用来指出该页的信息可读/写；只读；只可执行；不可访问等，指令执行时进行核对。例如，要想向只读块写入信息则指令停止执行，产生中断。

另外也可采取键保护的方法。系统为每道作业设置一个保护键，为某作业分配主存时根据它的保护键在页表中建立键标志。程序执行时将程序状态字中的键和访问页的键进行核对，相符时才可访问该块。为了使某块能被各程序访问，可规定键标志为“0”，此时不进行核对工作。操作系统有权访问所有块，可让操作系统程序的程序状态字中的存储键为“0”。IBM370 操作系统便采用这种方法，主存储器被分割成 2KB 大小的存储块,每一块都由硬件另外设置了由 4 位二进数组成的”存储保护键”,可由操作系统置为 0-15,每个用户作业分得一个与它作业不同的存储保护键码,该作业的所有存储块的存储键都置成这个存储保护键码。当作业被挑选进入主存运行时，操作系统把它的存储保护键码放入 PSW 的存储键字段(密钥),这样每当处理器访问主存时,都要核对钥键匹配情况,以决定是允许访问还是拒绝访问,若运行进程试图对钥键不同的主存进行访问，则会产生一个主存保护中断。通常用户使用 1～15 存储保护键码,操作系统使用 0 键,不论匹配与否,它可以访问整个主存,而且它还有权修改和设置密钥和保护键。

4.3.4 多级页表

现代计算机已普遍使用 32 或 64 位虚拟地址,可以支持 2^{32}～2^{64} 容量的逻辑地址空间，采用页式存储管理时,页表会相当的大。以 Windows NT 为例,其运行的 x86 CPU 具有 32 位地址,它使用 2^{32} 逻辑地址空间的分页系统,规定页面 4KB 时,每个进程的页表的表项有 1 兆(2^{20})个,若以每个表项占用 4 个字节计算,则每个进程需要占用 4KB 连续内存空间存放页表,这样做存储开销太大了。为此,提出了多级页表的概念,采取以下措施:内存仅存放当前使用的页表,其余暂时不用部分放在磁盘上,待用到时再行调进;同时,页表占用内存空间不必连续可分散存放。具体做法是:把整个页表进行分页,分成一张张小页表,每个小页表的大

小与页框相同，例如每个小页表形成的页面可以有 1K(2^{10})个页表表目。我们可对小页表顺序编号,允许小页表分散存放在不连续的页框中,为了进行索引查找,应该为这些小页表建一张页目录表,其表项指出小页表所在页框号及相关信息。于是系统要为每个进程建一张页目录表,它的每个表项对应一个小页表,而小页表的每个表项给出了页面和页框的对应关系,页目录表是一级页表,小页表是二级页表。于是逻辑地址结构有三部分组成：页目录、页号和位移。图 4-15 是二级页表实现逻辑地址到物理地址的转换过程,具体步骤如下：

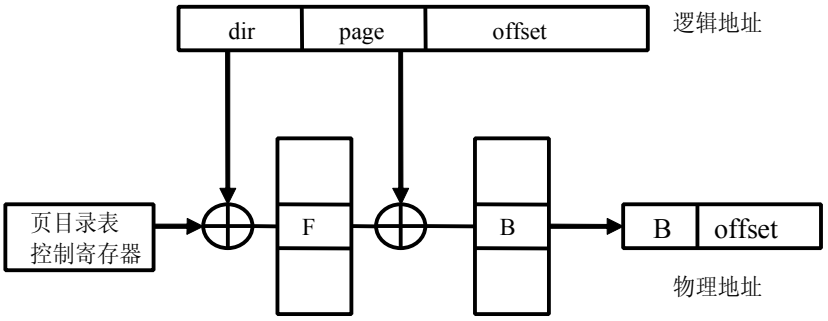


图 4-15 二级页表地址转换过程

由页目录表控制寄存器指出当前运行进程的页目录表内存所在地址,由页目录表起址加上 dir 作索引,可找到某个小页表在内存页框的地址,再以 page 作索引,找到小页表的页表项,而该表项中包含了页面对应的页框号, 页框号和位移 offset 便可生成物理地址。

上面的方法虽解决了可分散存放小页表的问题,但并未解决占用内存空间的问题。解决后一个问题的方法如下，对于进程运行涉及页面的小页表应放在主存,而其它小页表使用时再行调入。为了实现这一点,需要在页目录表中增加一个特征位,用来指示对应的小页表是否已调入内存,地址转换机构根据逻辑地址中的 dir,去查页目录表对应表项,如未调入内存,应产生一个"缺小页表"中断信号，请求操作系统将这张小页表调入主存。

二级页表地址变换需三次访问主存，一次访问页目录、一次访问页表、一次访问指令或数据,访问时间加了两倍。随着 64 位地址出现,三级、四级页表也被引入系统。

SUN 微系统公司的计算机使用 SPARC 芯片，采用如图 4-16 的三级分页结构。为了避免进程切换时重新装入页表,硬件可以支持多达 4096 个上下文，每个进程一个；当一个新进程装入主存时,操作系统分给一个上下文号，进程保持这个上下文号直到终止。当 CPU 访问内存时,上下文号和逻辑地址一起送入称作 MMU(memory management unit)的地址转换机构(现代计算机中的 MMU 完成逻辑地址到物理地址的转换功能，通常它是由一个或一组芯片组成，接受虚拟地址作为输入，物理地址作为输出，直接送到总线上)，它使用上下文号作上下文表的索引，以找到进程的顶级页目录，然后，使用逻辑地址中的索引值找下一级表项,直至找到访问页面,形成物理地址。在分页系统中，为了加快地址转换过程，都会使用相联存储器。

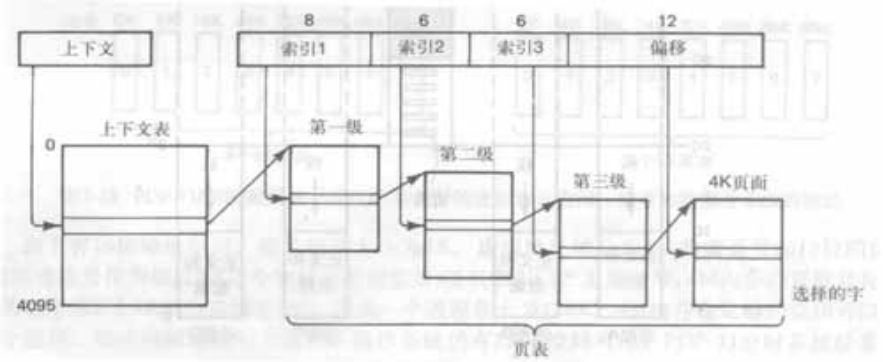


图 4-16　SPARC 三级分页结构

4.3.5 反置页表

由于计算机逻辑空间越来越大，页表占用的主存空间也越来越多,为了减少主存空间的开销,许多机器和操作系统如 IBM AS/400、IBM RS/6000、Mach 等,采用了反置页表 IPT(Inverted Page Table),IPT 是为内存中的每一个物理块建立一个页表并按照块号排序,该表的每个表项包含正在访问该页框的进程标识、页

号及特征位,用来完成主存页框到访问进程的页号、即物理地址到逻辑地址的转换。图 4-17 是反置页表及地址转换,其地址转换过程如下:逻辑地址给出进程标识和页号,用它们去比较 IPT,若整个反置页表中未能找到匹配的页表项,说明该页不在主存,产生请页中断,请求操作系统调入;否则，该表项的序号便是页框号,块号加上位移,便形成物理地址。

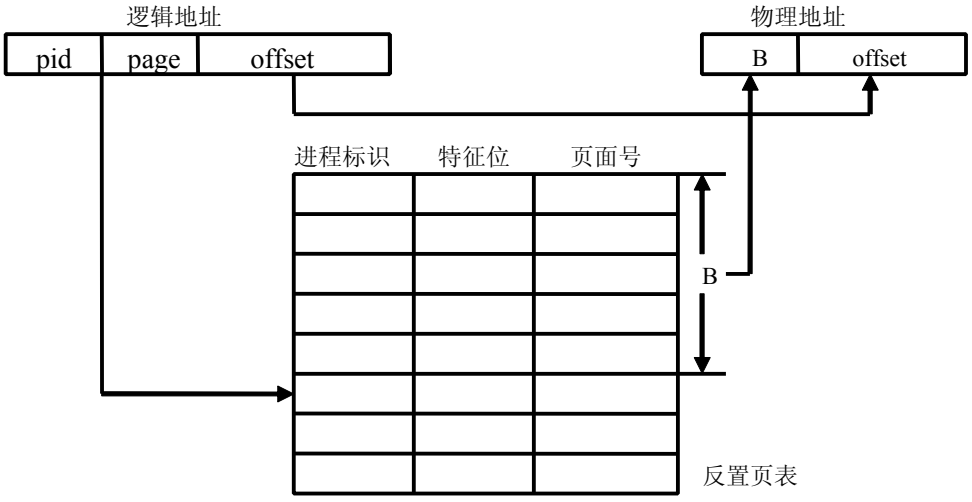


图 4-17　反置页表及其地址转换

虽然 IPT 能减少页表占用内存,如一个 128MB 的主存空间，若页面大小为 1KB,则 IPT 只需 128KB。然而，IPT 仅包含了调入内存的页面,不包含未调入内存的页面,所以,仍需要为进程建立传统的页表,不过这种页表不再放在主存中,而存在磁盘上。当发生缺页中断时,把所需页面调入主存要多访问一次盘，这时的速度是较慢的。

4.4 分段式存储管理

4.4.1 程序的分段结构

促使存储管理方式从固定分区到动态分区,从分区方式到分页方式发展的主要原因是为了提高主存空间利用率。那么,分段存储管理的引入,主要是满足用户(程序员)编程和使用上的要求,这些要求其它各种存储管理技术难以满足。在分页存储管理中,经连结编辑处理得到了一维地址结构的可装配模块,这是从 0 开始编址的一个单一连续的逻辑地址空间,虽然操作系统可把程序划分成页面,但页面与源程序无逻辑关系,也就难以实现对源程序以模块为单位进行分配、共享和保护。事实上，程序还可以有一种分段结构，现代高级语言常常采用模块化程序设计。如图 4-18 所示，一个程序由若干程序段(模块)组成，例如由一个主程序段、若干子程序段、数组段和工作区段所组成，每个段都从“0”开始编址，每个段都有模块名，且具有完整的逻辑意义。段与段之间的地址不连续，而段内地址是连续的。用户程序中可用符号形式（指出段名和入口）调用某段的功能，程序在编译或汇编时给每个段再定义一个段号。可见这是一个二维地址结构,分段方式的程序被装入物理地址空间后,仍应保持二维地址结构,这样才能满足用户模块化程序设计的需要。

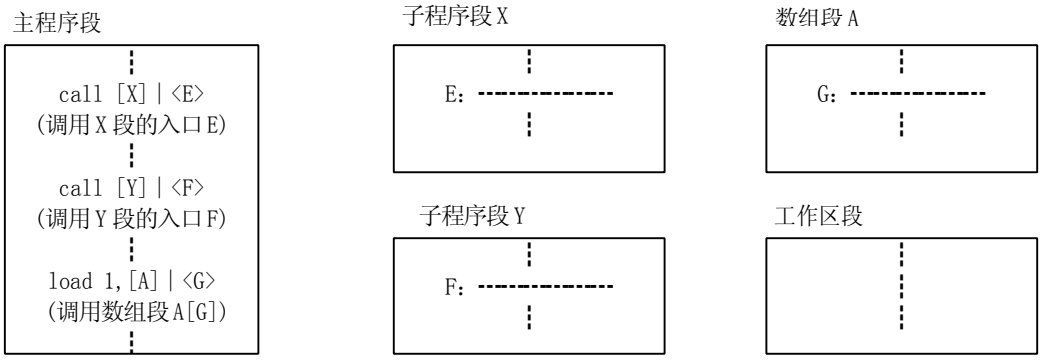


图 4-18　程序的分段结构

4.4.2 分段式存储管理的基本原理

分段式存储管理是以段为单位进行存储分配，为此提供如下形式的两维逻辑地址：

段号：段内地址

在分页式存储管理中，页的划分——即逻辑地址划分为页号和单元号是用户不可见的，连续的用户地址空间将根据页框架（块）的大小自动分页；而在分段式存储管理中，地址结构是用户可见的，即用户知道逻辑地址如何划分为段号和单元号，用户在程序设计时，每个段的最大长度受到地址结构的限制，进一步，每一个程序中允许的最多段数也可能受到限制。例如，PDP-11/45 的段址结构为：段号占 3 位，单元号占 13 位，也就是一个作业最多可分 8 段，每段的长度可达 8K 字节。

分段式存储管理的实现可以基于可变分区存储管理的原理，为作业的每一段分配一个连续的主存空间，而各段之间可以不连续。在进行存储分配时，应为进入主存的每个用户作业建立一张段表，各段在主存的情况可用一张段表来记录，它指出主存储器中每个分段的起始地址和长度。同时段式存储管理系统包括一张作业表，将这些作业的段表进行登记，每个作业在作业表中有一个登记项。作业表和段表的一般格式如图 4-19：

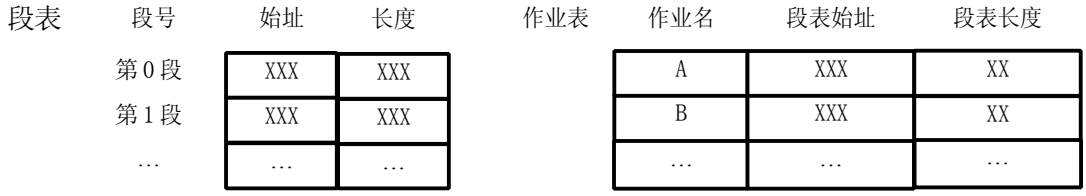


图 4-19　段表和作业表的一般格式

段表表目实际上起到了基址/限长寄存器的作用。作业执行时通过段表可将逻辑地址转换成绝对地址。由于每个作业都有自己的段表，地址转换应按各自的段

表进行。类似于分页存储器那样，分段存储器也设置一个段表控制寄存器，用来存放当前占用处理器的作业的段表始址和长度。段式存储管理的地址转换和存储保护流程图 4-20。

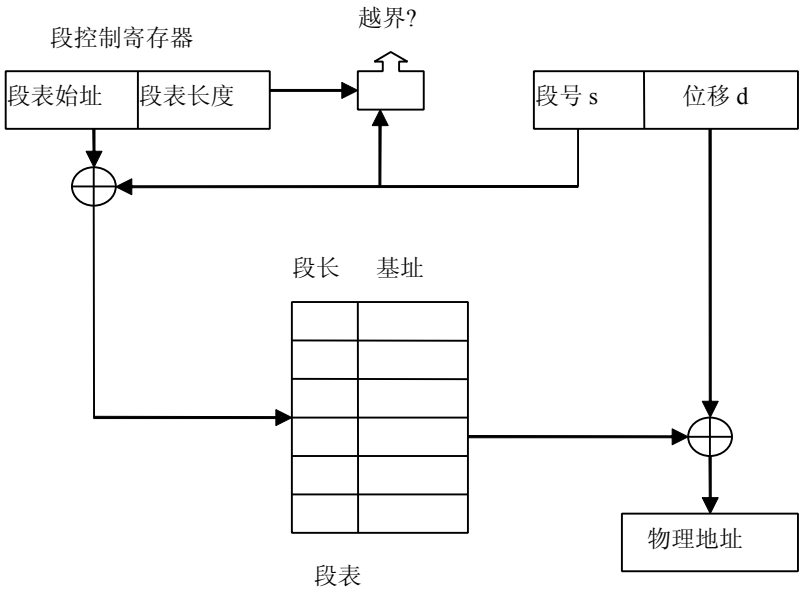


图 4-20　分段式存储管理的地址转换和存储保护

4.4.3 段的共享

在可变分区存储管理中，每个作业只能占用一分区，那么就不允许各道作业有公共的区域。这样，当几道作业都要用某个例行程序时就只好在各自的区域内各放一套。显然，降低了主存的使用效率。

在分段式存储管理中，由于每个作业可以有几个段组成，所以可以实现段的共享，以存放共享的程序和常数。所谓段的共享，事实上就是共享分区，为此计算机系统要提供多对基址/限长寄存器。于是，几道作业共享的例行程序就可放在一个公共的分区中，只要让各道的共享部分有相同的基址/限长值就行了。

由于段号仅仅用于段之间的相互访问，段内程序的执行和访问只使用段内地址，因此不会出现页共享时出现的问题，对数据段和代码段的共享都不要求段号相同。当然对共享区的信息必须进行保护，如规定只能读出不能写入，欲想往该区域写入信息时将遭到拒绝并产生中断。

4.4.4 分段和分页的比较

分段是信息的逻辑单位，由源程序的逻辑结构所决定，用户可见，段长可根据用户需要来规定，段起始地址可以从任何主存地址开始。在分段方式中，源程序(段号，段内位移)经连结装配后仍保持二维结构。

分页是信息的物理单位，与源程序的逻辑结构无关，用户不可见，页长由系统确定，页面只能以页大小的整倍数地址开始。在分页方式中，源程序(页号，页内位移)经连结装配后变成了一维结构。

4.5 虚拟存储管理

4.5.1 虚拟存储管理的概念

在前面介绍的各种存储管理方式中，必须为作业分配足够的存储空间，以装入有关作业的全部信息，当然作业的大小不能超出主存的可用空间,否则这个作业是无法运行的。但当把有关作业的全部信息都装入主存储器后，作业执行时实际上不是同时使用全部信息的，有些部分运行一遍便再也不用,甚至有些部分在作业执行的整个过程中都不会被使用到(如错误处理部分)。进程在运行时不用的，或暂时不用的，或某种条件下才用的程序和数据,全部驻留于内存中是对宝贵的主存资源的一种浪费，大大降低了主存利用率。于是，提出了这样的问题：作业提交时，先全部进入辅助存储器,作业投入运行时,能否不把作业的全部信息同时装入主存储器，而是将其中当前使用部分先装入主存储器，其余暂时不用的部分先存放在作为主存扩充的辅助存储器中，待用到这些信息时，再由系统自动把它们装入到主存储器中,这就是虚拟存储器的基本思路。如果“部分装入、部分对换”这个问题能解决的话，那么当主存空间小于作业需要量时，这个作业也能执行;更进一步，多个作业存储总量超出主存总容量时,也可以把它们全部装入主存,实现多道程序运行。这样，不仅使主存空间能充分地被利用，而且用户编制程序时可以不必考虑主存储器的实际容量，允许用户的逻辑地址空间大于主存储器的绝对地址空间。对于用户来说，好象计算机系统具有一个容量很大的主存储器，我们把它称作为“**虚拟存储器**”(virtual memory)。

我们对虚拟存储器的定义如下：具有部分装入和部分对换功能,能从逻辑上对内存容量进行大幅度扩充,使用方便的一种存储器系统。实际上是为扩大主存而采用的一种设计技巧。虚拟存储器的容量与主存大小无关。虚拟存储器的实现对用户来说是感觉不到的，他们总以为有足够的主存空间可容纳他的作业。

现在，进而讨论在作业信息不全部装入主存的情况下能否保证作业的正确运行？回答是肯定的,早在1968 年 P.Denning 就研究了程序执行时的局部性原理，即进程的程序和数据的访问都有聚集成群的倾向，在一个时间段内，程序仅在某个部分执行，或仅访问存储空间的某个区域。这只要对程序的执行进行分析就可以发现以下一些情况：

第一、程序往往包含若干个循环，在一段时间里这些程序部分被多次调用(如计数循环、转子程序、堆栈)。第二，程序在一段时间运行中，往往集中访问某一存储区域中的数据或附件位置的数据(如动态数组、顺序代码)。第三，程序中有些部分是彼此互斥的，不是每次运行时都用到的，例如出错处理程序，仅当在数据和计算中出现错误时才会用到,正常情况下, 出错处理程序不放在主存,不影响整个程序的运行。上述种种情况表现为时间局部性和空间局部性,这些现象充分说明，作业执行时没有必要把全部信息同时存放在主存储器中,而仅仅只装入一部分的假设是合理的。在装入部分信息的情况下，只要调度得好，不仅可以正确运行，而且能提高系统效率。

虚拟存储器是基于程序局部性原理上的一种假想的而不是物理存在的存储器，允许用户程序以逻辑地址来寻址，而不必考虑物理上可获得的内存大小，这种将物理空间和逻辑空间分开编址但又统一管理和使用的技术为用户编程提供了极大方便。此时，用户作业空间称虚拟地址空间，其中的地址称虚地址。为了要实现虚拟存储器，必须解决好以下有关问题：主存辅存统一管理问题、逻辑地址到物理地址的转换问题、部分装入和部分对换问题。。

虚拟存储器的思想早在 60 年代初期就已出现了,到 60 年代中期，较完整的虚拟存储器在分时系统 MULTICS（Multiplexed Information and computing service）和 IBM 系列操作系统中得到实现。70 年代初期开始推广应用，逐步为广大计算机研制者和用户接受，虚拟存储技术不仅用于大型机上，而且随着微型机的迅速发展，也研制出了微型机虚拟存储系统。目前，虚拟存储管理主要采用以下几种技术实现：请求分页式、请求分段式和段页式虚拟存储管理。

4.5.2 分页式虚拟存储系统

4.5.2.1 分页式虚拟存储系统的基本原理

分页式虚拟存储系统是将作业信息的副本存放在磁盘这一类辅助存储器中，当作业被调度投入运行时，并不把作业的程序和数据全部装入主存，而仅仅装入立即使用的那些页面，至少要将作业的第一页信息装入主存，在执行过程中访问到不在主存的页面时，再把它们动态地装入。用得较多的分页式虚拟存储管理是请页式(demand Paging),当需要执行某条指令或使用某个数据，而发现它们并不在主存时，产生一个缺页中断，系统从辅存中把该指令或数据所在的页面调入内存。

由于请页式虚存管理与分页式实存管理不同,仅让作业或进程当前使用部分放在主存中，所以，执行过程中必然会发生某些页面不在内存中的情况，那么，怎样才能发现页面不在内存中呢?怎样处理这种情况呢?采用的办法是：扩充页表的内容,增加驻留标志位和页面辅存的地址等信息,扩充后的页表，如图 4-21 所示：

页号	驻留标志	块号	辅存地址	其它标志
第 0 页	---	---	---	---
第 1 页	---	---	---	---
---	---	---	---	---

图 4-21　页式虚拟存储管理的页表扩展

驻留标志位(又称中断位)用来指出对应页是否已经装入主存，如果某页所对应栏的驻留标志位为 1，则表示该页已经在主存；若驻留标志位为 0，此时产生一个缺页中断信号，可以根据辅存地址知道该页在辅助存储器中的位址,将这个页面调入主存。

在作业执行中访问某页时，硬件的地址转换机构查页表，若该页对应驻留标志为 1，则按上面给出的办法进行地址转换，得到绝对地址。若该页驻留标志为 0，则由硬件发出一个缺页中断，表示该页不在主存。操作系统必须处理这个缺页中断，处理的办法是先查看主存是否有空闲块，若有则按该页在辅助存储器中的地址将这个页面找出且装入主存，在页表中填上它占用的块号且修改标志位。若主存已没有空闲块，则必须先淘汰已在主存中的某一页，再将所需的页面装入，对页表和主存分配表作相应的修改,见图 4-22。由于产生缺页中断时，一条指令并没有执行完，所以在操作系统进行缺页中断处理后，应重新执行中断的指令。当重新执行时，由于要访问的页已经装入主存，所以就可正常执行下去。

为了提高系统效率，可在页表中增加标志位，其它标志包括修改位（Modified）、引用位（Renferenced）、禁止缓存位和访问位，用来跟踪页的使用情况。当一个页被修改后，硬件自动设置修改位，一旦修改位被设置，当该页被调出主存时必须重新被写回辅存;若一页在执行过程中没有被修改过，那么不必重新写回到存储器中。引用位则在该页被引用时设置，无论是读或写，它的值被用来帮助操作系统进行页面淘汰。禁止缓存位可以禁止该页被缓存，这一特性对于那些正在与外设进行数据交换的页面时非常重要。访问位则限定了该页允许什么样的访问权限如可读、可写和可执。

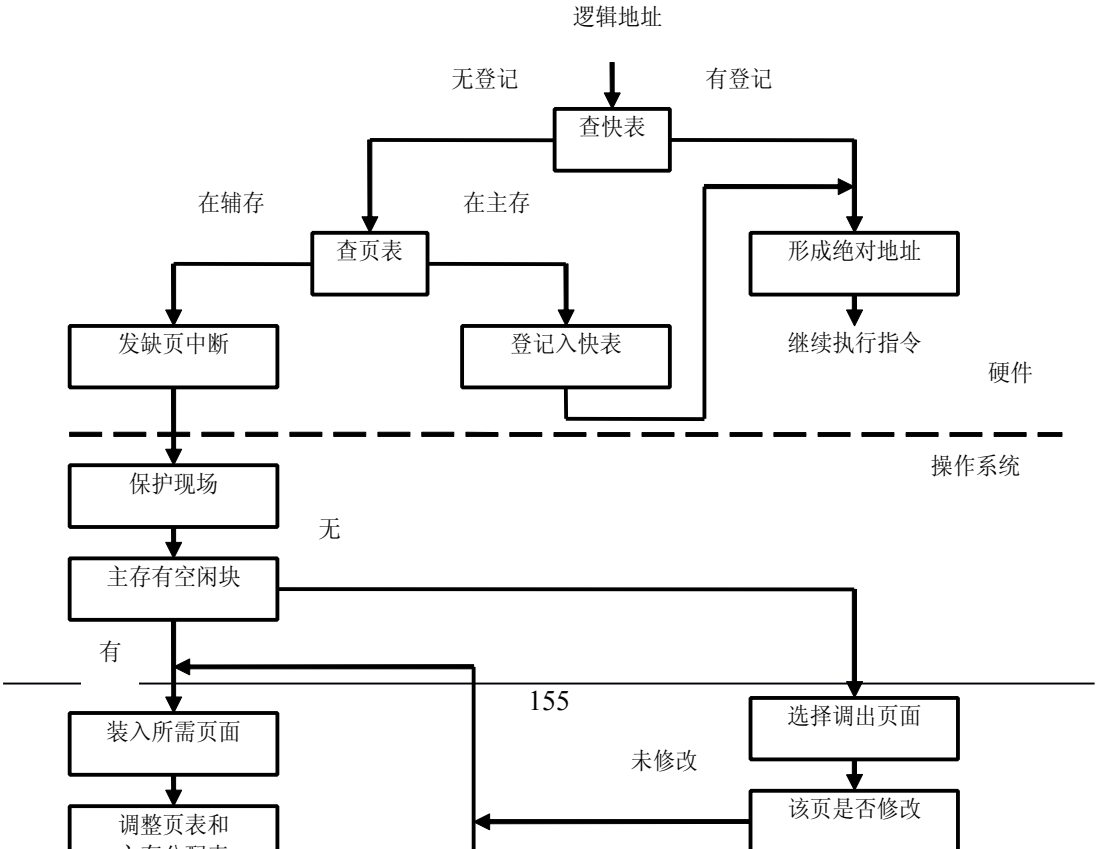


图 4-22　缺页中断处理流程

在分页式虚拟存储系统中，由于作业的诸页是根据请求把页面被装入主存的，因此，这种存储系统也称为请求页式存储管理。IBM/370 系统的 VS/1，VS/2 和 VM/370，Honeywell 6180 的 MULTICS 以及 UNIVAC 系列 70/64 的 VMS 等都采用请求页式虚拟存储系统。请求页式虚拟存储系统有以下优点：作业的程序和数据可以按页分散存放在内存中，减少了移动的开销，有效地解决了碎片问题;由于采用请页式管理，用户可用的主存空间大大扩展，既有利于改进主存利用率，又有利于多道程序运行。其主要缺点是：要有一定硬件支持，要进行缺页中断处理，因而机器成本增加，系统开销加大,此外页内会出现碎片，如果页面较大，则主存的损失仍然很大。

4.5.2.2 页面装入策略和清除策略

页面装入策略决定何时把一个页面装入主存,有两种策略可供选择：请页式调度和预调式调度。

请页式调度是仅当需要访问程序和数据时，才把所在页面装入主存。那么当某个进程第一次执行时，开始会有许多缺页中断，随着越来越多的页面装入主存，根据局部性原理,大多数未来访问的程序和数据都在最近被装入主存的页面中，一段时间之后,缺页中断就会下降到很低的水平,程序进入相对平稳阶段。这种策略的主要缺点是处理缺页中断和调页的系统开销较大，由于每次仅调一页,增加了磁盘 I/O 次数。

预调式调度由操作系统预测进程将要使用的那些页面,在使用入之前预先调入主存,每次调入若干个页面,而不是像请页式那样仅调一个页面。由于进程的页面大多数连续存放在辅存储中器，一次调入多 个页面能减少磁盘 I/O 启动次数,节省了寻道和搜索时间。但是如果调入的一批页面中多数未被使用,则效率就很低了，可见预调页要建立在预测的基础上，目前所用预调页的成功率在 50%左右。

许多系统把预调式调度应用在一个进程刚开始执行时(装入进程初始工作集)或每次缺页中断发生时(同时调入马上要用到的一些页面)，对于前一种情况往往要程序员提供所需程序部分的信息。预调式调 度和进程的对换不是一码事，当一个进程挂起被换出主存时,它的常驻页面统统移出;当该进程继续执行时，先前移出的页面都会重新调入主存,但不是由预调页调度来完成的。

清除策略是与装入策略相对的，它要考虑何时把一个修改过的页面写回辅存储器,常用的两种方法是:请页式清除和预清除。请页式清除是仅当一页选中被替换,且之前它又被修改过,才把这个页面写回辅助存储器。预清除方法对更改过的页面，在需要之前就把它们都回辅助存储器,因此可以成批进行。两个方法都有缺点，对于预清除，写出的页仍然在内存中,直到页页替换算法选中一页从内存中移出,它允许成批地把页面写出，但如若刚刚写出了很多页面，在它们被替换前，其中大部分又被更改，那么，预清除就毫无意义。对于请页式清除，写出一页是在读进一个新页之前进行的，它要成双操作,它仅需要写出一页，但引起进程不得不等待两次 I/O 操作完成，可能会降低 CPU 使用效率。

较好的方法是采用页缓冲，其策略如下:仅清除淘汰的页面，并使清除操作和替换操作不必成双进行。在页缓冲中，淘汰了的页面进入两个队列中,修改页面和非修改页面队列。在修改页面队列中的页的不时地式批写出并加入到非修改页面队列; 非修改页面队列中的页面，当它被再次引用时回收，或者淘汰掉以作替换。

4.5.2.3 页面分配策略

分页式虚拟存储系统排除了主存储器实际容量的约束，能使更多的作业同时多道运行，从而提高了系统的效率，但缺页中断的处理要付出相当的代价，由于页面的调入、调出要增加 I/O 的负担而且影响系统效率，因此应尽可能的减少缺页中断的次数。到目前为止，我们一直没有讨论如何在相互竞争的多个可运行进程之间分配内存资源,究竟如何为进程分配页框?当出现一次缺页中断时，页面替换算法的作用范围究竟应该局限于本进程的页面还是整个系统的页面?这两个问题涉及到进程常驻集的管理。

在分页式虚拟存储管理中,不可能也不必要把一个进程的所有页面调入主存,那么操作系统决定为某

进程分配多大的主存空间,需要考虑以下因素:①分配给一个进程的空间越小,同一时间处于内存的进程就越多，于是至少有一个进程处于就绪态的可能性就越大,从而可减少进程交换的时间。②如果进程只有一小部分在入主存里,即使它的局部性很好,缺页中断率还会相当高。③因为程序的局部性原理，分配给一个进程的内存超过一定限度后，再增加内存空间,也不会明显降低进程的缺页中断率。基于这些因素，在请页式系统中,可采用两种策略分配页框给进程：**固定分配和可变分配**。

如果进程生命周期中,保持页框数固定不变,称页面分配为固定分配;通常在进程创建时,根据进程类型和程序员的要求决定页框数,只要有一个缺页中断产生,进程就会有一页被替换。如果进程生命周期中,分得的页框数可变, 称页面分配为可变分配;当进程执行的某一阶段缺页率较高,说明进程程序目前的局部性较差，系统可多分些页框以降低缺页率，反之说明进程程序目前的局部性较好,可以减少分给进程的页框数。固定分配策略缺少灵活性，而可变分配的性能会更好些,被许多操作系统所采用。采用可变分配策略的困难在于操作系统要经常监视活动进程的行为和进程缺页中断率的情况,这会增加操作系统的开销。

在进行页面替换时，也可采用两种策略：**局部替换和全局替换**。如果页面替换算法的作用范围是整个系统，称为全局页面替换算法,它可以在可运行进程之间动态地分配页框。如果页面替换算法的作用范围局限于本进程，称为局部页面替换算法,它实际上需要为每个进程分配固定的页框。　在通常情况下，尤其是常驻集大小会在进程运行期间发生变化时，全局算法比局部算法好。如果使用局部算法，那么即使有大量的空闲页框存在，工作集的增长仍然会导致颠簸；如果工作集收缩了，局部算法又会浪费内存。但是使用全局算法时，系统必须不断地确定应该给每个进程分配多少内存，这是比较困难的。

固定分配往往和局部替换策略配合使用，每个进程运行期间分得的页框数不再改变,如果发生缺页中断,只能从进程在内存的页面中选出一页替换,以保证进程的页框总数不变。这种策略的主要难点在于:应给每个进程分配多少个页框?给少了,缺页中断率高;给多了,会使内存中能同时执行的进程数减少，进而造成处理器空闲和其它设备空闲。采用固定分配算法时，系统把可供分配的页框分配给进程，可采用下列方式：①平均分配，不论进程大小，每个进程得到的页框数相等,会造成大进程缺页中断率高。②按比例分配, 大进程获得页框多,小进程获得页框少。③优先权分配,可以把页框分成两类,一类按上比例分配，另一类根据进程优先权适当增加份额。`

可变分配往往和全局替换策略配合使用,这是采用得较多的一种分配和替换算法。先为系统中的每个进程分配一定数目的页框,接作系统自己保留若干空闲页框,当一个进程发生缺页中断时，从系统空闲页框中选一个给进程,于是可把缺页调入这个页框中,这样产生缺页中断的进程的内存空间会逐渐增大。凡是产生缺页中断的进程都能获得新的页框,有助于减少系统的缺页中断次数。直到系统拥有的空闲页框耗尽，才会从内存中选择一页淘汰，该页可以是内存中任一进程的页面，这样又会使那个进程的页框数减少，缺页中断率上升。这个方法的难点在于选择哪个页面进行替换，选择范围是主存中除锁定外的全部页框，应用一种淘汰策略选页时,并没有规则可以确定哪一个进程会失去一页，因而，如果选择了一个进程，这个进程的工作集的减少会严重影响它的执行，那么这个选择就不是最佳的。

可变分配配合局部替换可以克服可变分配全局替换的缺点，其实现要点如下:①新进程装入主存时，根据应用类型、程序要求，分配给一定数目页框,可用请页式或预调式完成这个分配。②当产生缺页中断时,从产生缺页中断的进程的常驻集中选一个页面替换。③不时重新评价进程的分配，增加或减少分配给进程的页框以改善系统性能。

4.5.2.4 页面替换策略

实现虚拟存储器能给用户提供一个容量很大的存储器，但当主存空间已装满而又要装入新页时，必须按一定的算法把已在主存的一些页调出去，这个工作称页面替换。所以，页面就是用来确定应该淘汰哪页的算法,也称淘汰算法。算法的选择是很重要的，选用了一个不适合的算法，就会出现这样的现象：刚被淘汰的页面又立即要用，因而又要把它调入，而调入不久再被淘汰，淘汰不久再被调入。如此反复，使得整个系统的页面调度非常频繁以至于大部时间都化在来回调度页面上。这种现象叫做“抖动”(Thrashing)，又称“颠簸”，一个好的调度算法应减少和避免抖动现象。

替换策略要处理的是:当把一个新的页面调入主存时,选择已在主存的哪个页面作替换。由于下列因素,使得替换策略变得比较困难:①分给每个活跃进程多少页框?②页面替换时,仅限于缺页中断进程还是包括主存中所有页面?③在被考虑的压面集合中,选出哪个页面进行替换?为了衡量替换算法的优劣，我们考虑在固定空间的前提下来讨论各种页面替换算法。这一类算法是假定每道作业都给固定数的主存空间，即每道作业占用的主存块数不允许页面替换算法加以改变。在 这样的假定下，怎样来衡量一个算法的好坏呢？我们先来叙述一个理论算法。假定作业 p 共计 n 页，而系统分配给它的主存块只有 m 块（m,n 均为正整数，且 1 ≤m≤n），即最多主存中只能容纳该作业的 m 页。如果作业 p 在运行中成功的访问次数为 s（即所访问的页在主存中）， 不成功的访问次数为 F（即缺页中断次数），则总的访问次数 A 为：

$$A = S + F$$

又定义：

$$f = F / A$$

则称 f 为缺页中断率。影响缺页中断率 f 的因素有：

- 主存页框数。作业分得的主存块数多，则缺页中断率就低，反之，缺页中断率就高。
- 页面大小。如果划分的页面大，则缺页中断率就低，否则缺页中断率就高。

- 页面替换算法。
- 程序特性。程序编制的方法不同，对缺页中断的次数有很大影响,程序的局部性要好。

例如：有一个程序要将 128×128 的数组置初值“0”。现假定分给这个程序的主存块数只有一块，页面的尺寸为每页 128 个字，数组中的元素每一行存放在一页中，开始时第一页在主存。若程序如下编制：

```
Var A: array[1..128] of array [1..128] of    integer;
      for j := 1 to 128
         do for i := 1 to 128
               do A[i][j]:=0
```

则每执行一次 A [i][j] :＝ 0 就要产生一次缺页中断，于是总共要产生（128×128－1）次缺页中断。如果重新编制这个程序如下：

```
Var A: array[1..128] of    array[1..128] of    integer;
      for j := 1 to128
         do for i := 1 to 128
               do A[i][j] := 0
```

那么,总共只产生（128－1）次缺页中断。

显然，虚拟存储器的效率与程序的局部性程度密切相关,局部化的程度因程序而异，一般说，总希望编出的程序具有较好的局部性。这样，程序执行时可经常集中在几个页面上进行访问，减少缺页中断率。

同样存储容量与缺页中断次数的关系很大。从原理上说，提供虚拟存储器以后，每个作业只要能分到一块主存储空间就可以执行，从表面上看，这增加了可同时运行的作业个数，但实际上是低效率的。试验表明当主存容量增大到一定程度，缺页中断次数的减少就不明显了。大多数程序都有一个特定点，在这个特定点以后再增加主存容量收效就不大，这个特定点是随程序而变的，试验分析表明，对每个程序来说，要使其有效的工作，它在主存中的页面数应不低于它的总页面数的一半。所以，如果一个作业总共有 n 页那么只有当主存至少有 n/2 块页框时才让它进入主存执行，这样可以使系统获得高效率。

下面介绍页面替换算法。一个理想的替换算法是：当要调入一页而必须淘汰一个旧页时，所淘汰的页应该是以后不再访问的页或距现在最长时间后再访问的页。这样的调度算法使缺页中断率为最低。然而这样的算法是无法实现的因为在程序运行中无法对以后要使用的页面作出精确的断言。不过，这个理论上的算法可以用来作为衡量各种具体算法的标准。这个算法是由 Belady 提出来的，所以叫做 Belady 算法，又叫做最佳算法（Optimal）。

Belady 算法是一种理想化的页面调度算法，下面分别介绍几个比较典型又实用的页面调度算法。

1）随机页面替换算法

要淘汰的页面是由一个随机数产生程序所产生的随机数来确定,选择一个不常使用的页面会使系统性能较好,但这种调度算法做不到这一点,虽很简单但效率却低，一般不采用。

2）先进先出页面替换算法（FIFO）

先进先出调度算法是一种低开销的页面替换算法，基于程序总是按线性顺序来访问物理空间这一假设。这种算法总是淘汰最先调入主存的那一页，或者说在主存中驻留时间最长的那一页（常驻的除外）。这种算法可以采用不同的技术来实现。一种实现方法是系统中设置一张具有 m 个元素的页号表，它是由 M个数：

P[0]， P[1]， …， P[m-1]

所组成的一个数组，其中每个 P[i] (i =0,1,…m-1) 存储一个在主存中的页面的页号。假设用指针 k 指示当前调入新页时应淘汰的那一页在页号表中的位置，则淘汰的页号应是 P[k]。每当调入一个新页后，执行

```
P[k] := 新页的页号;
k := (k+1) mod m;
```

假定主存中已经装了 m 页，k 的初值为 0，那么第一次淘汰的页号应为 P[0]，而调入新页后 P[0]的值为新页的页号，k 取值为 1；…；第 m 次淘汰的页号为 P[m-1]，调入新页后，P[m-1]的值为新页的页号，k 取值为 0；显然，第 m+1 次页面淘汰时，应淘汰页 号为 P[0]的页面，因为它是主存中驻留时间最长的那一页。

这种算法较易实现，但效率不高，因为在主存中驻留时间最长的页面未必是最长时间以后才使用的页面。也就是说，如果某一个页面要不断地和经常地被使用，采用 FIFO 算法,在一定的时间以后就会变成驻留时间最长的页，这时若把它淘汰了，可能立即又要用，必须重新调入。据估计，采用 FIFO 调度算法，缺页中断率为最佳算法的三倍。

另一个简单的实现算法是引入指针链成队列，只要把进入主存的页面按时间的先后次序链接,新进入的页面从队尾入队,淘汰总是从队列头进行。

3）最近最少用页面替换算法（LRU，least Recently used）

最近最少用调度算法是一种通用的有效算法，被操作系统、数据库管理系统和专用文件系统广泛采用。该算法淘汰的页面是在最近一段时间里较久未被访问的那一页。它是根据程序执行时所具有的局部

性来考虑的，即那些刚被使用过的页面，可能马上还要被使用，而那些在较长时间里未被使用的页面，一般说可能不会马上使用到。

为了能比较准确地淘汰最近最少使用的页，从理论上来说，必须维护一个特殊的队列（本书中称它为页面淘汰序列）。该队列中存放当前在主存中的页号，每当访问一页时就调整一次，使队列尾总指向最近访问的页，队列头就是最近最少用的页。显然，发生缺页中断时总淘汰队列头所指示的页；而执行一次页面访问后，需要从队列中把该页调整到队列尾。

例：给某作业分配了三块主存，该作业依次访问的页号为：4，3，0，4，1，1，2，3，2。于是当访问这些页时，页面淘汰序列的变化情况如下：

访问页号	页面淘汰序列	被淘汰页面
4	4	
3	4，3	
0	4，3，0	
4	3，0，4	
1	0，4，1	3
1	0，4，1	
2	4，1，2	0
3	1，2，3	4
2	1，3，2	

从实现角度来看，LRU 算法的操作复杂，代价极高，因此在实现时往往采用模拟的方法。

第一种模拟方法可以通过设置标志位来实现。给每一页设置一个引用标志位 R，每次访问某一页时，由硬件将该页的标志 R 置 1，隔一定的时间 t 将所有页的标志 R 均清 0。在发生缺页中断时，从标志位 R 为 0 的那些页中挑选一页淘汰。在挑选到要淘汰的页后，也将所有页的标志 R 清 0。这种实现方法开销小，但 t 的大小不易确定而使精确性差。t 大了，缺页中断时所有页的标志 R 值均为 1；t 小了，缺页中断时，可能所有页的 R 值均为“0”，同样很难挑选出应该淘汰的页面。

第二种模拟方法是为每个页设置一个多位寄存器 r。当页面被访问时，对应的寄存器的最左边位置 1；每隔时间 t，将 r 寄存器右移一位；在发生缺页中断时，找最小数值的 r 寄存器对应的页面淘汰。

例如，r 寄存器共有四位，页面 P0、P1、P2 在 T1、T2、T3 时刻的 r 寄存器内容如下：

页面	时刻		
	T1	T2	T3
P0	1000	0100	1010
P1	1000	1100	0110
P2	0000	1000	0100

在时刻 T3 时，该淘汰的页面是 P2。这是因为，同 P0 比较，它不是最近被访问的页面；同 P1 比较，虽然它们在时刻 T3 都没有被访问，且在时刻 T2 都被访问过，但在时刻 T1 时 P2 没有被访问。

显然，第二种模拟方法优于第一种模拟方法，它又被称为“老化算法”。老化算法可以比较好地模拟运行进程的当前工作集，使得系统达到比较好的性能。有关工作集模型的讨论参见下节。

4）第二次机会页面替换算法

FIFO 算法可能会把经常使用的页面淘汰掉，可以对 FIFO 算法进行改进,把 FIFO 算法与页表中的”引用位”结合起来使用,算法可实现如下：首先检查 FIFO 中的队首页面(这是最早进入主存的页面),如果它的”引用位”是 0,那么这个页面既老又没有用，选择该页面淘汰；如果它的”引用位”是 1,说明虽然它进入主存较早，但最近仍在使用。于是把它的”引用位”清成 0,并把这个页面移到队尾，把它看作是一个新调入的页。这一算法称为第二次机会(second chance)算法,其含义是最先进入主存的页面，如果最近还在被使用的话,仍然有机会作为像一个新调入页面一样留在主存中。

5）时钟页面替换算法(Clock Policy)

如果利用标准队列机制构造 FIFO 队列，第二次机会页面调度算法将可能产生频繁地出队入队，实现代价较大。因此，往往采用循环队列机制构造页面队列，这样就形成了一个类似于钟表面的环形表，队列指针则相当于钟表面上的表针，指向要淘汰的页面，这就是时钟页面替换算法的得名。Clock 与第二次机会算法本质上没有区别,仅仅是实现方法不同,仍然要使用页表中的”引用位”,把作业已调入主存的页面链成循环队列，用一个指针指向循环队列中下一个将被替换的页面,算法的实现要点如下：

- 一个页面首次装入主存时，其”引用位”置 0。
- 在主存中的任何一个页面被访问时，其”引用位”置 1。
- 淘汰页面时,存储管理从指针当前指向的页面开始扫描循环队列，把所迁到的”引用位”是 1 的页面的”引用位”清成 0,并跳过这个页面；把所迁到的”引用位”是 0 的页面淘汰掉,指针推进一步。

扫描循环队列时，如果迁到的所有页面的”引用位”为 1，指针就会绕整个循环队列一圈，把碰到的所有页面的”引用位”清 0;指针停在起始位置，并淘汰掉这一页,然后，指针推进一步。

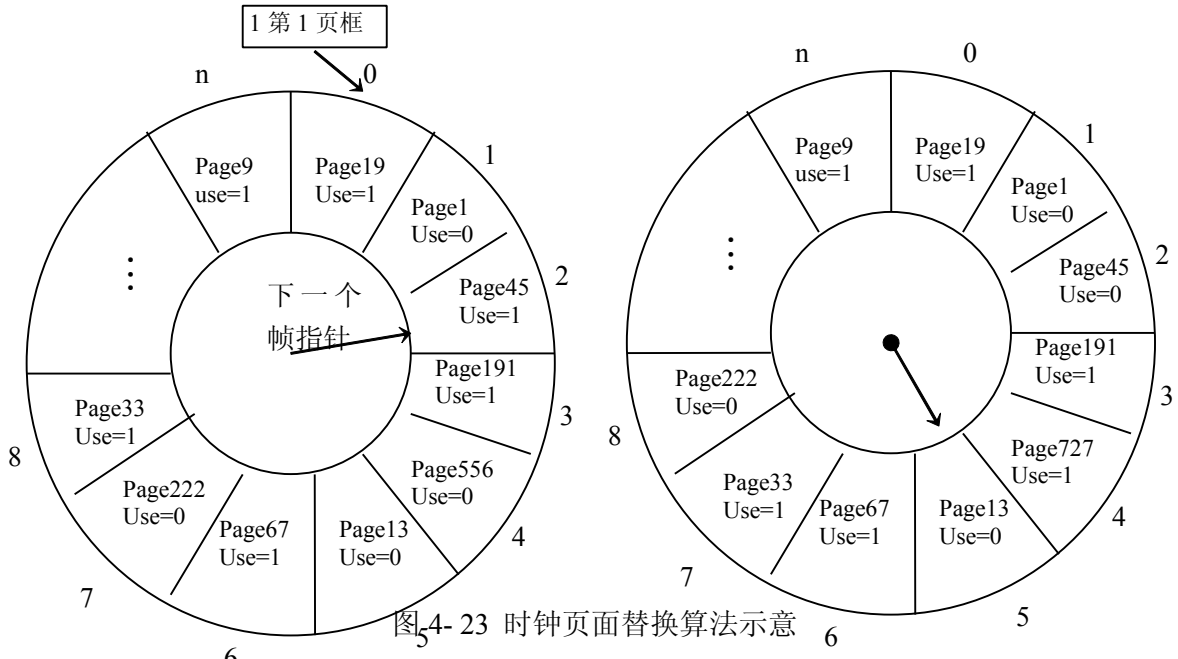


图 4- 23 时钟页面替换算法示意

(a) 一个页替换前的缓冲区状态　　(b) 即将替换后的缓冲区状态

图 4-23 给出了时钟页面替换算法的一个例子。当发生缺页时，即将替换后的缓冲区状态page727, 指针指向的是 page45(在页框 2 中)。Clock 页面替换算法执行如下：page45 的”引用位”是 1,故它不能被淘汰掉，仅把其”引用位”清 0,指针推进。同样道理，page191(在页框 3 中)也不能被替换，把其”引用位”清 0, 指针继续推进。在下一页即 page556(在页框 4 中),它的”引用位”是 0,于是,page556 被 page727 替换,并把 page727 的”引用位”置 1,指针前进到下一页 page13(在页框 5 中)。算法执行到此结束。

淘汰一个页面时，如果该页面已被修改过，必须将它重新写回磁盘;但如果淘汰的是未被修改过的页面,就不需要写盘操作了,这样看来淘汰修改过的页面比淘汰未被修改过的页面开销要大。如果把页表中的”引用位”和”修改位”结合起来使用可以改进时钟页面替换算法，它们一共组合成四种情况：

(1)最近没有被引用,没有被修改(r=0,m=0)

(2)最近被引用,没有被修改(r=1,m=0)

(3)最近没有被引用,但被修改(r=0,m=1)

(4)最近被引用过,也被修改过(r=1,m=1)

于是,改进的时钟算法可如下执行：

步 1：选择最佳淘汰页面，从指针当前位置开始,扫描循环队列。扫描过程中不改变”引用位”，把迁到的第一个 r=0,m=0 的页面作为淘汰页面。

步 2：如果步 1 失败，再次从原位置开始，查找 r=0 且 m=1 的页面，把把迁到的第一个这样的页面作为淘汰页面，而在扫描过程中把指针所扫过的页面的”引用位”r 置 0。

步 3：如果步 2 失败，指针再次回到了起始位置，由于此时所有页面的”引用位”r 均已为 0，再转向步 1 操作，必要时再做步 2 操作，这次一定可以挑出一个可淘汰的页面。

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

Unix SVR4 使用改进的 clock 算法、称双指针 clock 算法。其实现思想如下:主存中每个合法的页面(指不锁住,可淘汰的页面)都有”引用位”相连;当一个页面被替换进主存时,”引用位”置 0; 当一个页面被引用时,”引用位”置 1;系统设置了两个时钟指针，称前指针和后指针,每隔固定时间间隔,两个指针都扫描一遍;

第一遍,前向指针依次扫过合法页面并把”引用位”置 0;再延迟一定时间之后,后指针依次扫描合法页面,若该页面的”引用位”为 1,表明期间页面被引用过,则跳过该页面,继续扫描, 若该页面的”引用位”为 0,表明期间页面未被引用过,那么,此页面可被淘汰。双指针 clock 算法的关键是两个指针之间时间间隔的选 Unix SVR4 使用改进的 clock 算法、称双指针 clock 算法。其实现思想如下:主存中每个合法的页面(指不锁住,可淘汰的页面)都有”引用位”相连;当一个页面被替换进主存时,”引用位”置 0; 当一个页面被引用时,”引用位”置 1;系统设置了两个时钟指针，称前指针和后指针,每隔固定时间间隔,两个指针都扫描一遍;第一遍,前向指针依次扫过合法页面并把”引用位”置 0;再延迟一定时间之后,后指针依次扫描合法页面,若该页面的”引用位”为 1,表明期间页面被引用过,则跳过该页面,继续扫描, 若该页面的”引用位”为 0,表明期间页面未被引用过,那么,此页面可被淘汰。双指针 clock 算法的关键是两个指针之间时间间隔的选择。

6）最不常用页面替换算法（LFU：Least Erequently used）

如果对应每一页设置一个计数器，每当访问一页时，就使它对应的计数器加 1 。过一定时间 t 后，将所有计数器全部清 0 。当发生缺页中断时，可选择计数值最小的对应页面淘汰，显然它是在最近一段时间里最不常用的页面。这种算法实现不难，但代价太高而且选择多大的 t 最适宜也是个难题。

下面我们给出一个例子，分别用 Opt、FIFI、LRU 和 Clock 替换算法来计算缺页中断次数和被淘汰的页面，并对他们的性能作简单的比较。假设采用固定分配策略，进程分得三个页框,它在执行中按下列次序引用 5 个独立的页面: 2 3 2 1 5 2 4 5 3 2 5 2。图 4-24 给出了四种算法的计算过程和结果。

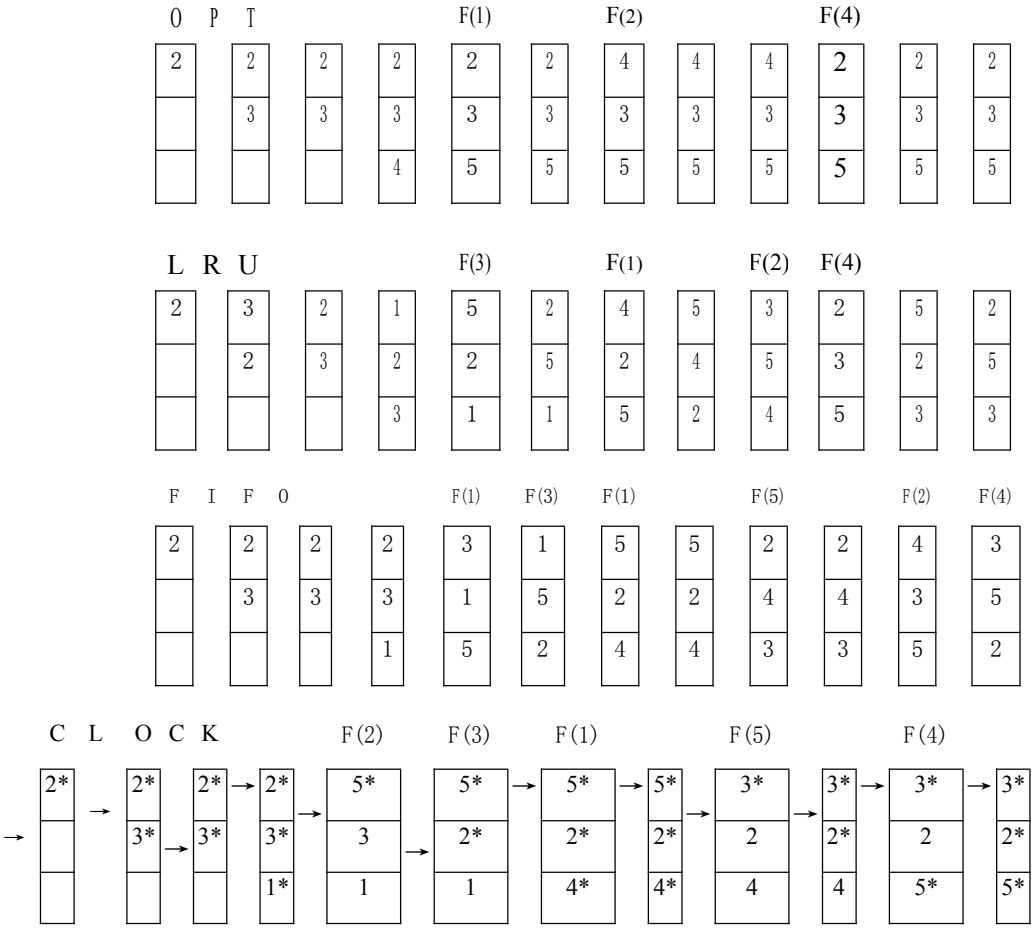


图 4-24 四种算法计算的结果

从图中可以看出，Opt 共产生 3 次缺页中断，第一次淘汰 page1，因为它以后不再使用了；第二次淘汰 page2，它要在最远的将来才被再次使用；第三次淘汰 page4，因为它以后不再被使用。LRU 共产生 4 次缺页中断，它将内存中最长时间内没有被引用的页面淘汰掉，根据局部性原理，最长时间内没有被引用的页面应该在最近的将来最不可能被引用到的页面，所以，淘汰的页面依次为 page3、page1、page2、page4,图中顶上一行为最近被访问的页面，最低下一行为最近最少被访问的页面。 FIFO 共产生 6 次缺页中断,它认为进入内存时间最长的页面,目前最不可能被引用,因此，应该被淘汰掉，所淘汰的页面依次为 page2、page3、page1、page5、page2、page4, 图中顶上一行为最先进入主存的页面，最低下一行为最近进入主存的页面。Clock 共产生 5 次缺页中断，图中*表示相应页面的”引用位”为 1，箭头→表示指针的

当前位置，当第一次引用 page5 时，由于此时循环队列中所有页面的”引用位”为 1，所以指针绕过一圈并指向 page2，故 page5 替换了 page2，同时 page3 和 page1 的”引用位”被置 0；第二次引用 page2 时，很容易看出应淘汰 page3，所以 page2 替换 page3；同样，当引用 page4 时，page4 替换 page1；第二次引用 page3 时，因为此时循环队列中所有页面的”引用位”再次为 1，因此，指针绕过一圈后 page3 替换了 page5； 当第三次引用 page2 时, 循环队列中三个页面 page3 的“引用位”为 1、page2 的“引用位”为 1 和 page4 的引用位”为 0，且指针指向 page2，所以，当第三次引用 page5 时显然应替换 page4。可以看出 FIFO 性能最差, Opt 性能最好，而 Clock 与 LRU 十分接近。

图 4-25 是用上述四种算法计算的结构曲线。假设分配给进程的页框数是固定的,运行的 FORTRAN 程序共有 0.25×10^6 次页面引用，页面大小为 256 个字。在这个实验中,分配给进程的页框数分别为 6、8、10、12 和 14。当分配给进程的页框数比较少时，四种算法的差距明显，且 FIFO 所产生的缺页中断基本上是 Opt 的 2 倍,Clock 则比较接近于 LRU。

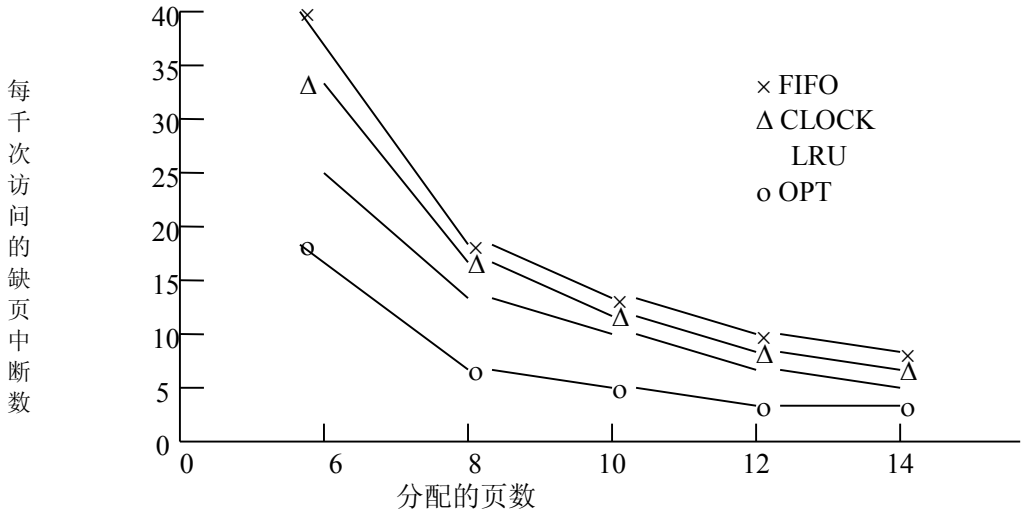


图 4-25　四种算法性能比较

4.5.2.5 页式虚拟存储系统的几个设计问题

1）页面大小

1）从页表大小考虑。在虚拟空间一定的前提下，不难看出页面大小的变化对页表大小的影响，如果页面较小，虚拟空间的页面数就增加，页表也随之扩大。由于每一个作业读必须有自己的页表，因此为了控制页表所占的主存量，页面的尺寸还是较大一点为好。

2）从主存利用率考虑。主存是以块为分配单位的，作业程序一般不可能正好划分为整数倍的页面。于是，作业的最后一个页面进入主存时，总会产生内部碎片，平均起来一个作业将造成半块的浪费。为了减少内部碎片，页面的尺寸还是小一点为好。

3）从读写一个页面所需的时间考虑。作业都存放在辅助存储器上，从磁盘读入一个页面的时间包括等待时间（移臂时间+旋转时间）和传输时间，通常等待时间远大于传输时间。显然，加大页面的尺寸，有利于提高 I/O 的效率。

4）最佳页面尺寸。目前实行页式虚拟存储管理的计算机系统中，页面大小大多选择在 512 字节到 4K 字节之间。所以如此，是从减少内部碎片和页表耗费的存储空间两个角度出发推导出来的。

假定 S 表示用户作业程序的平均长度，P 表示以字为单位的页面长度，且有 S>>P。则每个作业的也表长度为 S/P（为简单起见，假定每个页表项占用一个字）。在作业的最后一页，假定耗费主存 P/2 字。

若定义

$$f = (\text{浪费的存储字}/\text{作业所需总存储量})$$

来表示一个作业耗损主存量的比率，则对一个作业而言，有：

$$\text{浪费的存储字} = \text{页表使用的主存空间} + \text{内部碎片} = S/P + P/2$$

$$f = (S/P + P/2) / S = 1/P + P/2S$$

对于确定的 S，可视 f 是页面尺寸 P 的函数。于是可以求使 f 最小的 P 值。方法是先对 P 求一阶导数

$$f'(P) = df/dP = -1/P^2 + 1/2S$$

令 df /dP = 0，求得 P_0 = 开根号(2S)

再对 P 求二阶导数

$$f''(P) = d^2f/dP^2 = 2/P^3$$

将 P_0 = 开根号(2S)代入，得到

$$f''(P_0) = 2/(\text{开根号}(2S))^3 > 0$$

这表明函数 f(P)在 P_0 处取得极小值。也就是说，当选取 P_0 = 开根号(2S)时，存储耗损比率 f_0 = (开根号(2/S)为最小。这是称 P_0 为最佳页面尺寸。

反之，对于给定的页面尺寸，也可以使用同样的方法求得一个最佳的作业长度 S_0。下表给出了各种页面尺寸 P_0 (2 的幂次)时对应的 f_0 和 S_0 值。

页面尺寸（字）	作业大小（S_0）	存储耗损率（f_0）
8	32	25
16	128	13
32	512	6
64	2K	3
128	8K	1.6
256	32K	0.4
512	128K	0.2
1024	512K	

可以看出，总的趋势是当程序和页面尺寸增加时，存储耗损率（包括内部碎片和页表）下降。下面是一些著名操作系统选择的页面尺寸:Atlas 为 512 字(每字 48 位)、IBM370 系列机为 2048 或 4096 字节、VAX 为 512 字节、IBM as/400 为 512 字节、Intel 486 和 Motorola 68040(Macintosh)为 4096 字节。

2）工作集模型

从充分地共享系统资源这一角度出发，当然希望主存中的作业数越多越好。但是从保证作业顺利执行、使 CPU 能够有效地得到利用的角度出发，就应该限制主存中的作业数，以避免频繁地进行页面调入/调出，导致系统的抖动。为此，P.J.Denning 认为，应该将处理机调度和主存管理结合起来进行考虑，并在 1968 年提出了工作集模型。

从程序的局部性原理可知，在一段时间内作业的运行只涉及某几个页面。所谓工作集，就是指在某一段时间内作业运行所要访问的那些页面的集合。可以想象，随着作业的执行，工作集不断变化，所包含的页面数时而增多，时而减少。当执行进入某一个新阶段时，由于过渡，这一段时间的工作集所包含的页面会出现剧烈的变动，如图 4-26 所示。

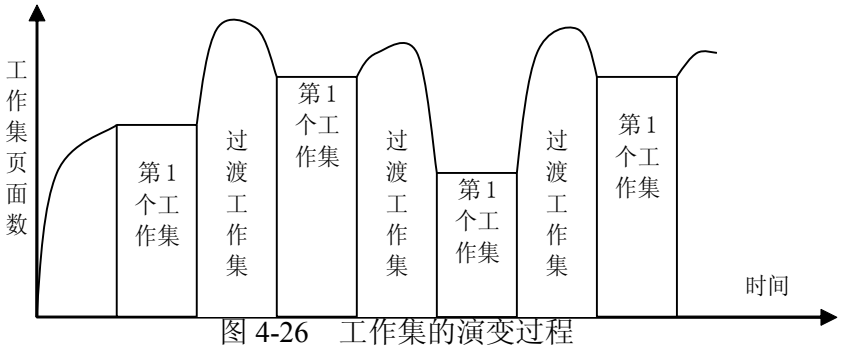


图 4-26　工作集的演变过程

从图上看，当作业开始运行而一次一次地把页面从辅存调入主存时，系统将分配给它很多主存块。当对存储的需求暂时趋于平稳状态时，就形成了第一个工作集。随着时间的推移，作业的访问将进到下一个工作集，中间出现一个用曲线表示的过渡工作集。最初曲线是上升趋势，这是因为程序运行促使新页面的调入，工作集膨胀。一旦下一个工作集逐渐稳定时，就可以淘汰不必要的页面，而形成第二个工作集。所以工作集的一次转移，会出现先上升后下降的曲线。

由此可见，如果在一段时间内，作业占用的主存块数目小于工作集时，运行过程中就会不断出现缺页中断，从而导致系统的抖动。所以为了保证作业的有效运行，在相应时间段内就应该根据工作集的大小分配给它主存块，以保证工作集中所需要的页面能够进入主存。推而广之，为了避免系统发生抖动，就应该限制系统内的作业数，使它们的工作集总尺寸不超过主存块总数。

窗口大小

页面访问序列	2	3	4	5
24	24	24	24	24
15	15 24	15 24	15 24	15 24
18	18 15	18 15 24	18 15 24	18 15 24
23	23 18	23 18 15	23 18 15 24	23 18 15 24
24	24 23	24 23 18	*	*
17	17 24	17 24 23	17 24 23 18	17 24 23 18 15
18	18 17	18 17 24	*	*
24	24 18	*	*	*
18	18 24	*	*	*

图 4-27　进程工作集

因为工作集也是时间 t 的函数，如果一个进程的执行超过了△个时间单位，而仅用了一通常，用 W(t, △)表示从时刻 t-△到时刻 t 之间所访问的不同页面的集合，t 是进程实际耗用的时间,可以通过执行的指令周期来计算; △是时间窗口尺寸,通过窗口来观察进程的行为。W(t, △)就是作业在时刻 t 的工作集,表示在最近△个实际时间单位内进程所引用过的页面的集合;｜W(t, △)｜表示工作集中的页面数目,称工作集尺寸。如果系统能随｜W(t, △)｜的大小来分配主存块的话，就既能有效的利用主存，又可以使缺页中断尽量少地发生,或者说程序要有效运行，其工作集必须在主存中。

我们来考察二元函数 W 的两个变量,首先 W 是 t 的函数，即随时间不同,工作集也不同。其一是不同时间的工作集所包含的页面数可能不同(工作集尺寸不同);其二是不同时间的工作集所包含的页面可能不同(不同内容的页面)。其次，W 是窗口尺寸△的函数,而且工作集的大小是窗口大小的非递减函数。如图4-27 所示，其中列出了进程的引用序列，星*表示这个时间单位里工作集没有发生改变。从图中可以看出，工作集越大，产生缺页中断的频率越低，其结果可用如下关系表示：

$$W(t, \triangle+1) \geqslant W(t, \triangle)$$

个页面，则｜W(t, △)｜=1,工作集的大小也会增加，在工作窗口尺寸允许的情况下，工作集的大小也可能会和进程所拥有的总页面数 n 一样大，这样，

$$1 \leqslant |W(t, \triangle)| \leqslant \min(t, n)$$

正确选择工作集窗口尺寸的大小对系统性能有很大影响，如果△过大，甚至把作业地址空间全包括在内，就成了实存管理; 如果△过小，则会引起频繁缺页,降低了系统的效率。图 5-　描述了在固定△值下,工作集大小随时间的变化而变化的情况，对许多程序来说，工作集大小相对稳定的时期和工作集大小快速改变的时期是交替存在的。当一个进程刚开始执行时，随着进程访问新页面，逐渐构造工作集。根据局部性原理，进程最终会稳定在页面的某个集合上，这就是稳定期,而随后的过渡期表明进程向一个新的局部转移。在过渡期中,仍然会有一些上一个局部中的页保留在窗口中，这些页在新页面访问时，会使工作集大小产生波动，但经过新页面被访问后，工作集大小会减少直到又包括这个新的局部中的页面。可以通过工作集概念来指导确定常驻集的大小:①监视每个进程的工作集。②定期地从一个进程常驻集中删去那些不在工作集中的页。③仅当一个进程的工作集在主存即常驻集包含了它的工作集时，进程才能执行。

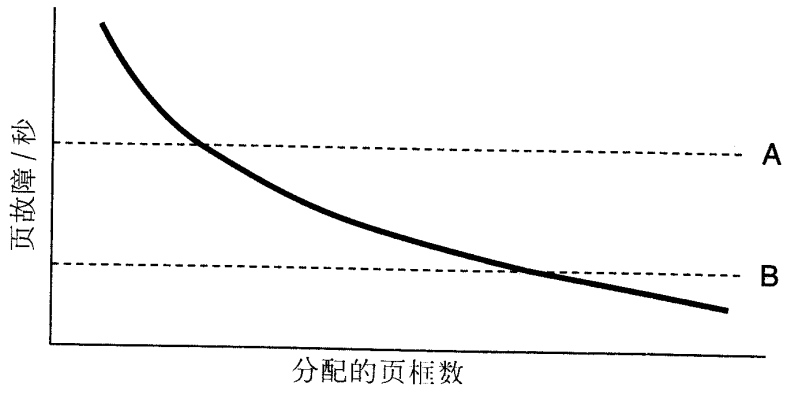


图 4-28　页面故障率是分配的页框数的函数

正确的策略并不是消除缺页现象,而应使缺页间隔时间保持在合理水平，当此间隔过小时，应增加其页框数，过大则应减少分给进程的页框数，一个更直接的改善系统性能的方法是使用页面故障率（Page Fault Frequency），或者叫做 PFF 分配算法。我们曾经讨论过，包括 LRU 在内的一大类页面替换算法的故障率随着分配的页框数的增加而减少，如图 4-28 所示。虚线 A 对应的是一个过高的故障率，发生故障的进程将被分配更多的页框以减少故障率；虚线 B 对应的是一个过低的故障率，我们可以得出结论分配给这个进程的内存太多了，可以收回一些页框。可以看出，PFF 试图把页面故障率保持在一个可以接受的范围内。如果发现内存中进程太多以至于不可能使所有进程的故障率都低于 A，那么就必须从内存中移去某些进程，把他们的页框分给余下的进程或放进一个空闲页框池中供后面发生页面故障时使用。从内存中移去一个进程的决定实际上是一种负载控制，它表明即使在分页系统中交换仍然是需要的，不同的只是现在交换是为了降低潜在的对内存的需求，而不是为了立刻使用收回的内存页框。

3）页面交换区

替换算法常常要挑选一个页面淘汰出主存，但是这个被淘汰出去的页面可能很快又要使用，被重新装入主存。因而，操作系统必须把被淘汰的页面内容保存在磁盘上,例如 Unix 使用交换区临时保存页面,系统初始化时,保留一定盘空间作交换区,不能被文件系统使用。

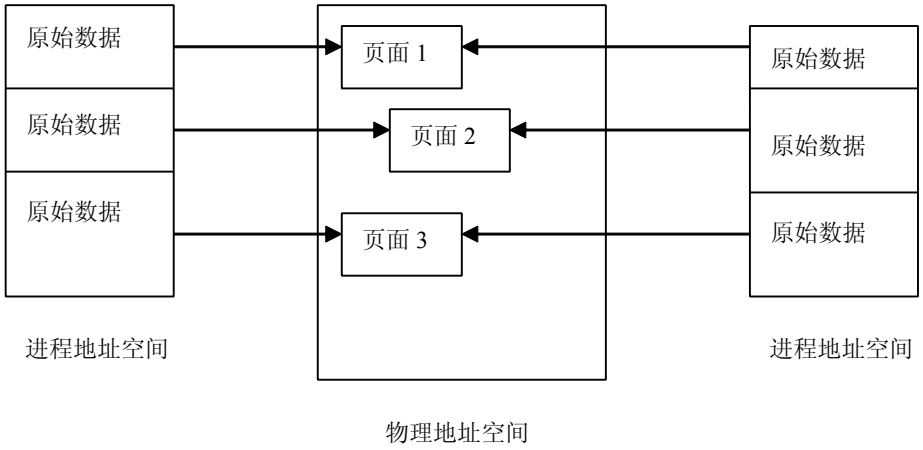
当某个被保存在交换区的页面再次被使用时，操作系统就从交换区中读出它们，所以，要建立和维护交换区映射表，记录所有被换出的页面在交换区中的位置，如果页面又要被换出主存，仅当其内容与保存在交换区的副本不同时才进行复制。

交换区映射表一般都设计为高效的数据结构，并驻留在主存的内核区中。但进程的页面被交换出来后，并不都放在交换区，通常只有数据页面、堆栈页面保留在交换区。

4、写时复制

写时复制(copy-on-write)是存储管理用来节省物理内存(页框)的一种页面级优化技术,已被 unix 和 Windows 等许多操作系统所采用，它能减少主存页面内容的复制操作,减少相同内容页面在主存的副本数目。

当两个进程(如父子进程)共享一个页面时,并不是立即为每个进程各建一个页面副本,而是把该页面定义为只读方式,让诸进程共享。当其中某个进程要修改页面内容执行写操作时,会产生一个”写时复制”中断,操作系统处理这个中断信号,为该进程分配一个空闲页框,复制页面的一个副本,且修改相应的页表项,当进程重新执行写页面操作时指令被顺利执行,图 5-28 是写时复制的示意。可见操作系统采用写时复制技术后,就可以延迟到修改时才对共享页面做出副本,从而.节省了大量页面复制操作和副本占用空间。



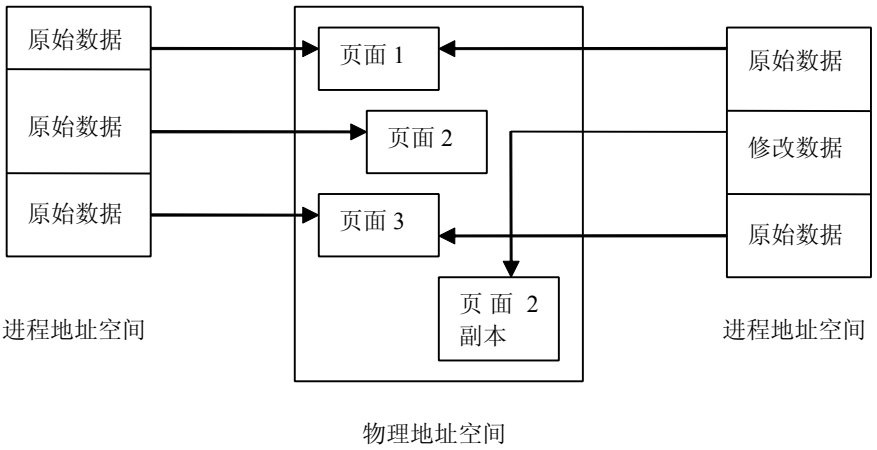


图 5-28 写时复制前(上图), 写时复制后(下图)

4.5.3 分段式虚拟存储系统

分段式虚拟存储系统也为用户提供比主存实际容量大的存储空间。分段式虚拟存储系统把作业的所有分段的副本都存放在辅助存储器中，当作业被调度投入运行时，首先把当前需要的一段或几段装入主存，在执行过程中访问到不在主存的段时再把它们装入。因此，在段表中必须说明哪些段已在主存，存放在什么位置，段长是多少。哪些段不在主存，它们的副本在辅助存储器的位置。还可设置该是否被修改过，是否能移动，是否可扩充，能否共享等标志。格式如图 4-29：

段号　　特征　　存取权限　　扩充位　　标志　　主存始址　　限长　　辅存始址

图 4-29　段式虚拟存储管理的段表扩展

其中：

- 特征位: 00(不在内存)；01(在内存)；11(共享段)；
- 存取权限: 00(可执行)；01(可读)；11(可写)；
- 扩充位: 0(固定长)；1(可扩充)；
- 标志位: 00(未修改)；01(已修改)；11(不可移动)；

在作业执行中访问某段时，由硬件的地址转移机构查段表，若该段在主存，则按分段式存储管理中给出的办法进行地址转换得到绝对地址。若该段不在主存中，则硬件发出一个缺段中断。操作系统处理这个中断时，查找主存分配表，找出一个足够大的连续区域容纳该分段。如果找不到足够大的连续区域则检查空闲区的总和，若空闲区总和能满足该分段要求，那么进行适当移动后，将该分装入主存。若空闲区总和不能满足要求，则可调出一个或几个分段在辅助存储器上，再将该分段装入主存。

在执行过程中，有些表格或数据段随输入数据多少而变化。例如，某个分段在执行期间因表格空间用完而要求扩大分段。这只要在该分段后添置新信息就行，添加后的长度应不超过硬件允许的每段最大长度。对于这种变化的数据段，当要往其中添加新数据时，由于欲访问的地址超出原有的段长，硬件产生一个越界中断。操作系统处理这个中断时，先判别一下该段的“扩充位”标志，如可以扩充，则增加段的长度，必要时还要移动或调出一个分段以腾出主存空间。如该段不允许扩充，那么这个越界中断就表示程序出错。

缺段中断和段扩充处理流程如图 4-30。

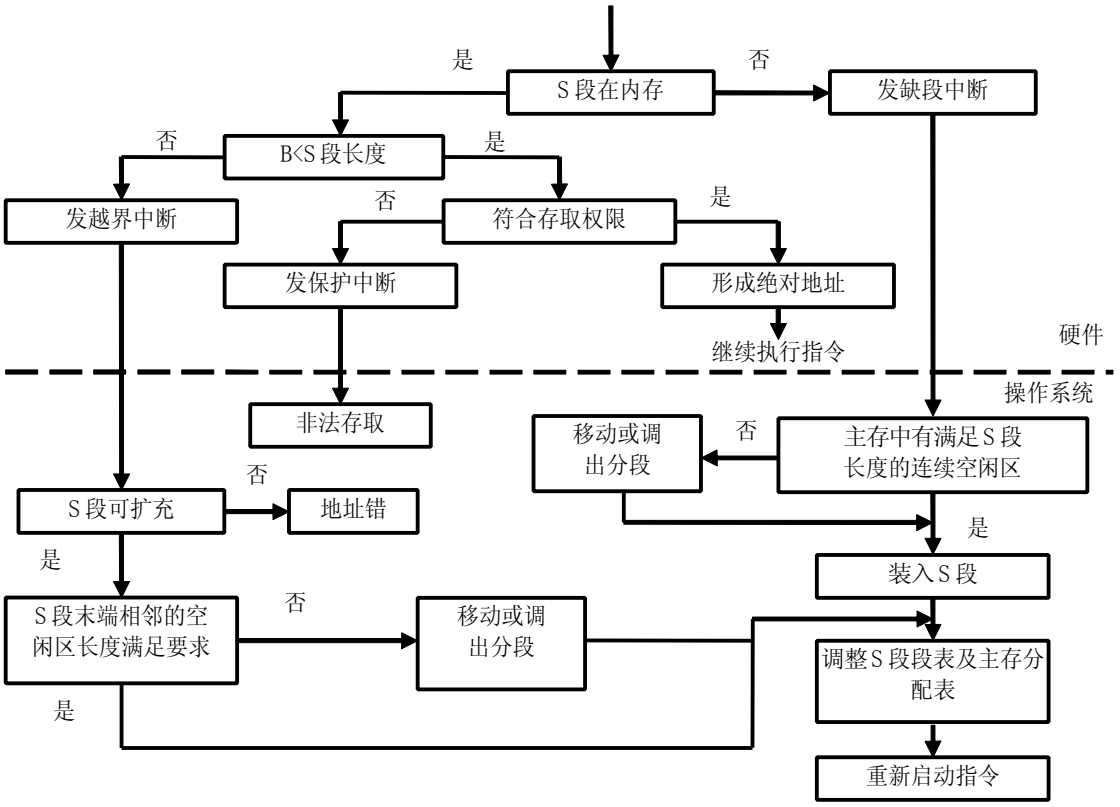


图 4-30　分段式存储管理的地址转换和存储保护

4.5.4 段页式存储管理

段式存储是基于用户程序结构的存储管理技术，有利于模块化程序设计，便于段的扩充、动态链接、共享和保护,但往往会生成段内碎片浪费存储空间;页式存储是基于系统存储器结构的存储管理技术，存储利用率高,便于系统管理，但不易实现存储共享、保护和动态扩充。如果把两者优点结合起来，在分页式存储管理的基础上实现分段式存储管理这就是段页式存储管理,下面介绍 段页式存储管理的基本原理。

1、虚地址以程序的逻辑结构划分成段,这是段页式存储管理的段式特征。

2、实地址划分成位置固定、大小相等的页框(块) ,这是段页式存储管理的页式特征

3、将每一段的线性地址空间划分成与页框大小相等的页面，于是形成了段页式存储管理的特征

4、逻辑地址形式为:

段号(s)	段内页号 (p)	页内位移(d)

对于用户来说，段式虚拟地址应该由段号 s 和段内位移 d’组成，操作系统内部再自动把 d’解释成两部分：段内页号 p 和页内位移 d，也就是说,d’=p×块长+d。

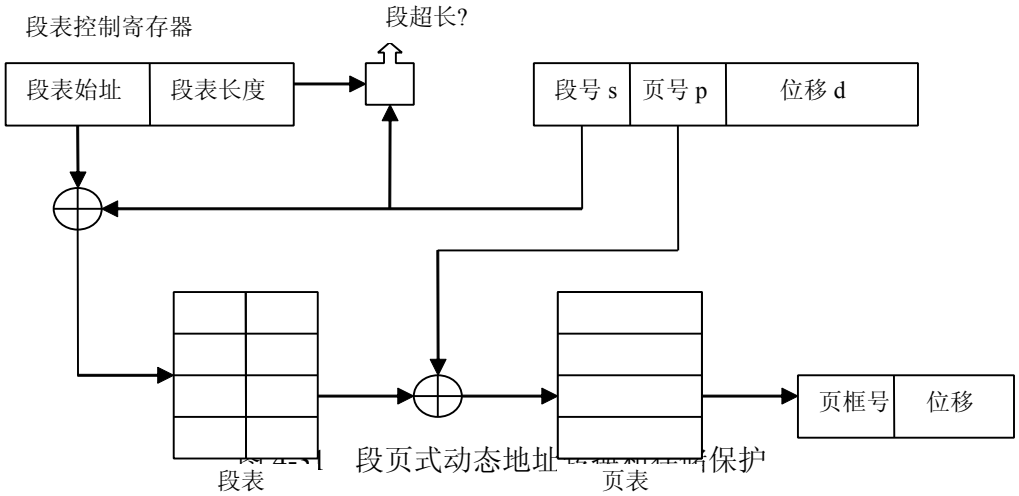
5、数据结构

段页式存储管理的数据结构更为复杂，包括作业表、段表和页表三级结构。作业表中登记了进入系统中的所有作业及该作业段表的起始地址，段表中至少包含这个段是否在内存，以及该段页表的起始地址,页表中包含了该页是否在主存(中断位)、对应主存块号。

6、动态地址转换

段页式存储管理的动态地址转换机构由、段表、页表和快表构成。当前运行作业的段表起始地址已被操作系统置入段表控制寄存器,其动态地址转换过程如下：从逻辑地址出发，先以段号 s 和页号 p 作索引去查快表,如果找到,那么立即获得页 p 的页框号 p',并与位移 d 一起拼装得到访问主存的实地址,从而完成了地址转换。若查快表失败,就要通过段表和页表来作地址转换了，用段号 s 作索引,找到相应表目,由此得到 s 段的页表的起始地址 s',再以 p 作索引得到 s 段 p 页对应的表目,由此得到页框号 p';这时一方面把 s 段 p 页和页框号 p'置换进快表，另一方面用 p'和 d 生成主存的实地址,从而完成地址转换。

上述过程是假设所需信息都在主存的情况下进行的，事实上，许多情况会产生,如查段表时,发现 s 段不在主存,于是产生'缺段中断',引起操作系统查找 s 段在辅存的位置,并将该段页表调入主存; 如查页表时,发现 s 段的 p 页不在主存,于是产生'缺页中断',引起操作系统查找 s 段 p 页在辅存的位置,并将该页调入主存,当主存已无空闲页框时,就会导致淘汰页面。图 4-31 是段页式动态地址转换和存储保护示意。



段页式动态地址[illegible]保护

4.6 实例研究：Intel Pentium

Intel 的 Pentium 和 Pentium Pro 既可以作为分段系统，又可以作为分页系统，也可以作为一个段页式存储管理系统来运行，通过设置段描述符中的控制位就可以作出以上选择。当然，在这个机制上也可以轻易实现各种连续空间存储管理。

4.6.1 Pentium 虚拟存储器的核心数据结构——描述符表

Pentium 虚拟存储器的核心是两张表：局部描述符表 LDT(local descriptor table)和全局描述符表 GDT(global descriptor table)。每个进程都有一个自己的 LDT，它描述局部于每个进程的段，包括代码段、数据段、堆栈段的基地、大小和有关控制信息等;系统的所有进程共享一个 GDT, 它描述系统段，包括操作系统自己的基地、大小和有关控制信息等。

4.6.2 段选择符和段描述符

为了引用一个段，Pentium 程序必须把这个段的选择符（selector）装入机器的 6 个段寄存器中的某一个。在运行过程中，要求段寄存器 CS 保存代码段的选择符，段寄存器 DS 保存数据段的选择符。每个选择符是一个 16 位数，如图 4-32 所示。选择符中的一位指出这个段是局部的还是全局的，其他 13 位是 LDT 和 GDT 的入口号。因此最多允许 8K 个段描述符索引，段描述符索引 0 是禁止使用的，它可以被装入一个段寄存器中来表示这个段寄存器不可用。另外还有两位用作保护。

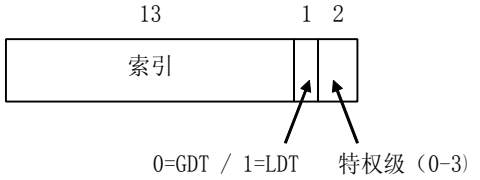


图 4-32　Pentium 的选择符

在选择符被装入段寄存器时，对应的描述符被从 LDT 和 GDT 中取出装入微程序寄存器中，以便被快速引用。一个段描述符由 8 个字节组成`，包括段的基址、长度和其他信息，如图 4-33 所示。

图 4-33　Pentium 的代码段描述符(数据段描述符略有区别)

其中：

- 基地址共 32 位（分三处合并），生成内存段的首址，加上 32 位偏移形成内存地址。对于 286 程序，基地址的 24—31 位不用，恒为 0；所以，286 只能处理 24 位基址。
- 长度位共 20 位，限定段描述符寻址的内存段的长度，注意段长度的计量单位可以是字节或页。
- G 位用于描述颗粒大小，即段长度的计量单位。G=0 表示长度以字节为单位；G=1 表示长度以页为单位，在 Pentium 中页的长度是固定的，为 4KB。于是段的长度分别为 2^{20} 字节或 $2^{20}\times4KB=2^{32}$ 字节。
- D 位：当 D=1 时，为 32 位段；当 D=0 时，为 16 位段。
- P 位表示内存段是否在物理主存中，若 P=1，表示段在内存中，若 P=0，表示段不在内存中。
- Dpl 位（2 位）表示特权级（0—3），用于保护。0 为内核级；1 为系统调用级，2 为共享库级，3 为用户程序级。Windows 95 只用两级：0 级和 3 级，即系统级和用户级。
- S 位为段位，当 S=1 时，表示当前段为应用程序；当 S=0 时，表示描述符将引用内存段外的系统信息（如 OS 的特别数据结构）。
- type 类型字段（3 位）,表示内存段类型，如可执行代码段、只读数据段、调用门等等。
- A 位为访问位，表示是否访问过内存段，为淘汰作准备

4.6.3 虚拟存储运行模式选择

当设置 G 控制位 1,系统运行于**段页模式,**此时每个进程可有 8192 个段,每个段 2^{20} 个页,每个页 4KB,也就是说段可有 2^{32} 字节大小。当设置 G 控制位 1,把系统六个段寄存器设置为同一个段选择符,段描述符基地址设置为”0”,段大小设置为最大;这时系统运行于**分页模式**,单段、分页，32 位地址空间模式运行。当设置 G 控制位 0, 系统运行于**分段模式**,每个进程可有 2^{13}=8K(8192)个段,每个段可有 2^{20} 字节大小,这种模式是为了与 286 兼容。

4.6.4 地址转换

Pentium 采用分段模式时,每一个虚地址包括一个 16 位段选择符和一个 32 位偏移量,不分段时,用户的虚拟内存是 2^{32}=4GB;而分段时,虚拟地址空间为 $2^{(32+14)}$=64TB;物理地址空间使用 32 个地址位,最大有 4GB。下面来讨论地址转换过程如下:

- 首先根据段选择符第二位 T 选择是查找 LDT 或 GDT；
- 以段选择符索引域的值为索引，把它拷贝进一个内部寄存器中并且它的低三位被清零,然后 LDT 或 GDT 表的起始地址被加到它上面，找到所要访问的段描述符。
- 检查段描述符中的 P 控制位,如果段不在主存（选择符为 0），就会发生一次陷入（中断）处理；如果段已经被换出，就会发生一次陷入（缺段中断）处理;如果段在主存,则随后检查偏移量是否超出了段的结尾，如果是也会发生一次陷入（越界中断）处理。
- 假设段在内存中并且偏移量也在范围内，就把描述符中 32 位的基地址和偏移量相加形成 32 位线性地址（linear address），如图 4-34 所示。为了和只有 24 位基地址的保护模式（80286）以及使用 16 位段寄存器来描述 20 位基地址的实模式（8086/8088）兼容，基址被分成 3 片分布在描述符的各个位置。实际上，基址允许各个段的起始地址在 32 位线性地址空间的任何位置。至此段转换已经完成。

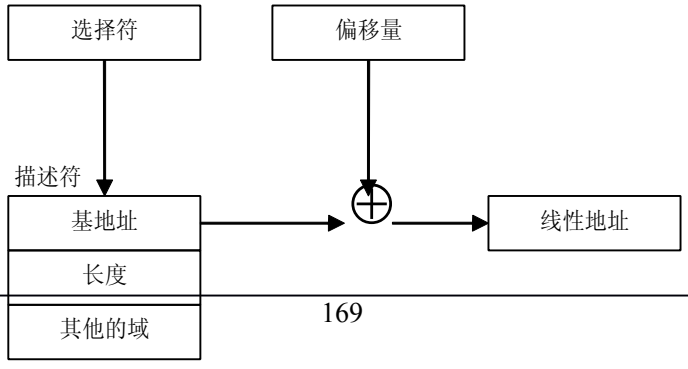


图 4-34 选择符，偏移被转换为线性地址的过程

是否要分页依据 G 控制位,若为 0,说明目前运行于分段模式，那么段转换得到的线性地址就是物理地址并被送往存储器用于读写;因此在分页被禁止时，就得到了一个纯的分段解决方案，各个段的基址在它的描述符中。注意，段允许互相覆盖，这是因为用硬件来检查所有的段都互不重叠代价太大，完全可以通过操作系统的软件机制加以解决。

若 G 控制位为 1，这时以分页模式运行，那么段转换得到的线性地址将被解释成分页模式的虚地址，需要通过页表找到页框后才映射成物理地址。这里真正复杂的是在 32 位虚地址和 4K 页面的情况下，一个段可能包括多达一百万个页。因此 Pentium 使用了一种两级页表以在段较小的情况下减少页表的尺寸。

4.6.5 二级页表和页式地址转换

每个运行进程都有一个由 1024 个 32 位表项组成的页目录（page directory），它的地址由一个全局寄存器指出。目录中的每一个表项都指向一个也包含 1024 个 32 位表项的页表，页表项指向页框（块），这个方案如图 4-35 和图 4-36 所示。线性地址被分成 3 个域：Dir、Page 和 Offset。Dir 域被作为索引在页目录找到指向正确页表的指针；随后，Page 域被作为索引在页表找到页框的物理地址；最后，Offset 被加到页框的地址上得到需要的字节或字的物理地址。

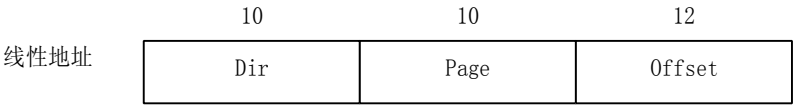


图 4-35　允许分页时线性地址的组成

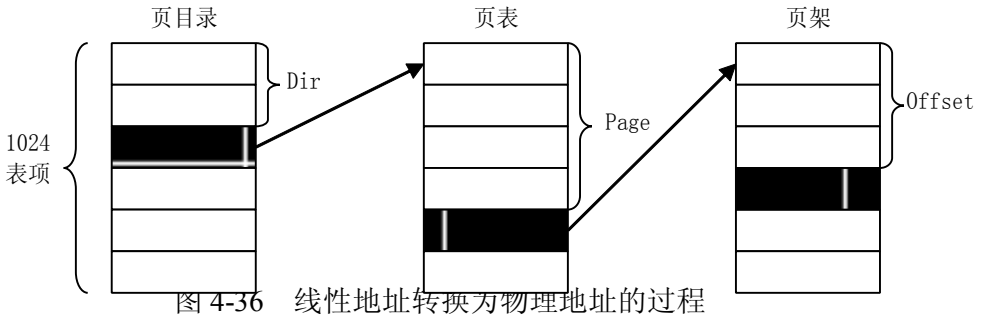


图 4-36　线性地址转换为物理地址的过程

如图 4-37 所示，页目录项和页表项的结构是类似的，均为 32 位，其中 20 位是页表位置/页架号，其余的未包括了由硬件设置的供操作系统使用的引用位、修改位、保护位、和其他一些有用的位。

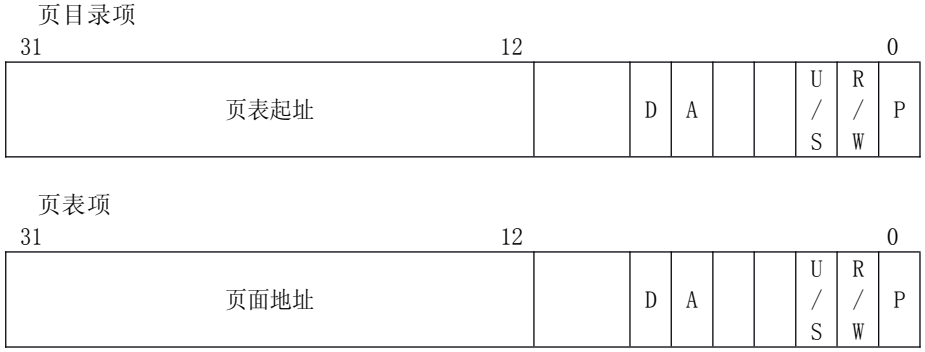


图 4-37　页目录项和页表项的结构

其中：

- D 位：修改位，当该页程序修改后，应先纪录为淘汰做准备。
- A 位：（读/写）引用位，凡对程序读写均置为已访问（accessed），淘汰时先找 A=0 的页面（Windows 用 LRU 法）。
- P 位：存在位，页面在内存时，P=1。

- U/S 位：用户/管理员（User/Supervisor）位，若 U/S=0，该页为一个管理员页面（OS 页面），用户不能访问。
- R/W 位：读/写（Read/Write）位，R/W=1 时允许页修改；R/W=0 时，不允许页修改；通常程序区的 R/W=0。

每个页表有描述 1024 个 4K 页架的表项，因此一个页表可以处理 4M 的内存。一个小于 4M 的段将有且仅有一个页表。因此通过这种方法，短的段的地址转换只需要访问两个页。

为了加快地址转换的速度，Pentium 也设置了相联存储器，又称为翻译后援存储器 TLB(Translation Lookaside Buffer)，它把最近使用过的 Dir/Page 暂存起来，这样就可以不通过页表而快速地把线性地址转化成物理地址。

Pentium 允许禁止分页，如 80286 以前的保护模式和实模式就是纯的段式系统。同样 Pentium 也可以被设置成纯的页式系统，这是只要让所有的段寄存器都被设置成同一个选择符，它的描述符基址是 0，长度是最大的。这样，只有一个 32 位的地址空间被使用，指令偏移将成为线性地址，实际上就是一般的分页。

必须承认，Pentium 的设计非常巧妙，它实现了互相冲突的目标：段页式存储管理、纯的页式存储管理、纯的段式存储管理、同 80286 以前的保护模式和实模式兼容。

4.6.6 Pentium 的保护机制

Pentium 的保护机制与虚存管理相关。它支持 4 个保护级别，0 级权限最高，3 级权限最低。一种典型的应用是把 4 个保护级别依次设定为：

- 0 级为操作系统内核级。处理 I/O、存储管理、和其他关键操作。该级可使用特权指令,可访问所有段和所有页。
- 1 级为系统调用处理程序级。用户程序可以通过调用这里的过程执行系统调用，但是只有一些特定的和受保护的过程可以被调用。
- 2 级为共享库过程级。它可以被很多正在运行的程序共享，用户程序可以调用这些过程，都去它们的数据，但是不能修改它们。
- 3 级为用户程序级。它受到的保护最少。

当然，上面的保护级别划分并不是一定的，各个操作系统实现可以采用不同的策略，如 Windows-95 只使用了 0 级和 3 级。

在任何时刻，运行程序都处在由 PSW 中的两位所指出的某个保护级别上，系统中的每一个段页由一个级别。只要运行程序使用与它同级的段，一切都会很正常。对更高级别数据的存取是允许的，而对更低级别数据的存取是非法的并会引起陷入（保护中断）。

调用不同级别的过程是允许的，但是要通过一种被严格控制着的方法。为了执行越级调用，CALL 指令必须包含一个选择符而不是地址，选择符指向一个称为调用门（call gate）的描述符，由它给出被调用过程的地址。因此要跳转到任何一个不同级别的代码段的中间都是不可能的，必须正式地指定入口点。

陷入和中断采用了一种和调用门类似的机构，它们引用的也是描述符而不是地址，这些描述符指向将被执行的特定过程。描述符的类型域可以区分是代码段、数据段、还是各种门。

4.7 实例研究：Linux 存储管理

本节基于 Pentium 平台讨论 Linux 的存储管理。

4.7.1 Linux 的分页管理机制

在 Linux 中，每一个用户进程都可以访问 4GB 的线性地址空间。其中从 0 从 3GB 的虚拟内存地址是用户空间，用户进程都可以直接对其进行访问。从 3GB 到 4 GB 的虚拟内存地址是内核态空间，存放仅供内核态访问的代码和数据，用户进程处于用户态时不能访问。当中断或系统调用发生，用户进程进行模式切换（处理器特权级别从 3 转为 0）。

所有进程从 3GB 到 4 GB 的虚拟内存地址都是一样的，又相同的页目录项和页表，对应到同样的物理内存段。Linux 以此方式让内核态进程共享代码段和数据段。另外，虚拟内存地址从 3GB 到 3 GB+4MB 的一段和 0 到 4MB 的一段是对应的。

Linux 才有请求页式技术管理虚拟内存。页表分为三层：页目录 PGD、中间页目录 PMD 和页表 PTE。在 Pentium 计算机上它被简化成两层，PGD 和 PMD 合二为一。

每一个进程都有一个页目录，存储该进程所使用的内存页面情况。当使用 fork()创建一个进程时，分配内存页面的情况如下：

- 进程控制块 1 页；
- 内核态堆栈 1 页；
- 页目录 1 页；

- 页表若干页。

而使用 exec()系统调用时，分配内存页面的情况如下：

- 可执行文件的第 1 页；
- 用户堆栈的 1 页或几页。

这样，当进程开始运行时，如果执行代码不在内存中，将产生第 1 次缺页中断，让操作系统参与分配内存，并将执行代码装入内存。此后，按照需要，不断的通过缺页中断调进代码和数据。当系统内存资源不足时，由操作系统决定是否调出一些页面。

4.7.2 内存的共享和保护

Linux 中内存共享是以页共享的方式实现的，共享该页的各个进程的页表项直接指向共享页，这种共享不需建立共享页表，节省内存空间，胆小率较低。当共享页状态发生变化时，共享该页的各进程页表均需修改，要多次访问页表。

为了能以自然的方式管理进程虚拟地址空间，Linux 定义了虚存段 VMA。一个 VMA 段时一个进程的一段连续的虚存空间，它的每个页单元拥有相同的特征，如属于同一进程、相同的访问权限、同时被锁定、同时受保护。

Linux 可以对虚存段中的任一部分加锁或保护。与加锁有关的操作共有 4 种：对指定的一段虚拟空间加锁或解锁（mlock 和 munlock）；对进程所有的虚拟空间加锁或解锁（mlockall 和 munlockall）。虚存加锁后，它对应的物理页面驻留内存，不再被页面置换程序换出。而对进程的虚拟地址空间实施保护操作，就是重新设置 VMA 段的访问权限，保护操作由系统调用 mprotect 实施。

对虚存段的加锁和保护操作可以有四种方式：对整个虚存段、或虚存段的前部、中部和后部分别进行加锁和保护。

4.7.3 物理地址空间管理

物理内存以页帧（page frame）为单位，长度固定，在 Pentium 上为 4KB。Linux 对物理内存的管理通过 mem_map 表描述，它在系统初始化时创建，其中每一个表项的结构如下：

```
typedef struct page {
   struct page    *next, *prev;
   /*  如果该页帧的内容是文件，则 inode 和 offset 指出文件的 inode 和偏移位置  */
   struct inode    *inode;
   unsigned long    offset;
   struct page    *next_hash;                  /*  页 cache 和 hash 表后向指针  */
   atomic_t    count;                          /*  访问该页帧的进程计数  */
   unsigned    flags;
   unsigned    dirty;                          /*  修改标记  */
   unsigned    age;                            /*  页帧年龄，越小越先换出  */
   struct wait_queue    *wait;
   struct page    *prev_hash;                  /*  页 cache 和 hash 表前向指针  */
   struct buffer_head    *buffers;             /*  如果该页帧作为缓冲区，则指示地址  */
   unsigned long    swap_unlock_entry;
   unsigned long    map_nr;                    /*  页帧在 mem_map 表中的下标  */
}    mem_map_t;
```

在物理内存低端，紧跟 mem_map 表的 bitmap 表以 0 位示图方式记录了所有物理内存空间的空闲状况。与一般位示图不同，bitmap 表分割成 NR_MEM_LISTS（缺省为 6）的组。首先是第 0 组，初始化长度为（end_mem – start_mem） / PAGE_SIZE / 2^{0+3}，每位表示一个页帧的占用情况。第 1 组初始化长度为（end_mem – start_mem） / PAGE_SIZE / 2^{1+3}，每位表示连续 2 个页帧的占用情况。第 i 组初始化长度为（end_mem – start_mem） / PAGE_SIZE / 2^{i+3}，每位表示连续 2^{i-1} 个页帧的占用情况。

Linux 还把所有的空闲物理页帧组织成 NR_MEM_LISTS 的双向链表，存储在 free_area 数组中。每个链表节点的包括三个数据向：next 和 prev 是链表指针，map 指向 bitmap 表。

Linux 采用 buddy 算法分配空闲块。当请求分配长度为 2^i 个页帧的块时，首先从 free_area 数组的第i条链表开始受搜索，如找不到，在搜索第 i+1 条链表，以此类推。如果找到的空闲块正好等于需求，则直接把它从来链表中删除，返回首地址。如果找到的空闲块大于需求，则需要把它一分为二，前半部分插入前一条链表，取后半部分。如果还大，则继续对分，取一半留一半，直至相等。同时，bitmap 表页必须相应调整。

回收空闲块时，应该根据 bitmap 表中的对应组，判断回收块的前后是否为空闲块。若是则合并，并调整 bitmap 表对应位，从 free_area 的相应空闲链表中取下该空闲块并归还。这是一个递归过程，直到找不到空闲块邻居，将最大的空闲块插入 free_area 的相应空闲链表。

4.7.4 交换空间

计算机的物理内存是影响机器性能的关键因素。相对于以 GB 计算的硬盘空间，内存的容量显得太少，尤其在多任务系统中更是如此。所以存储管理系统应该设法把暂时不用的内存数据转储到外存中。早期操作系统的解决方法是“交换”，即把暂时没有拥有 CPU 的进程整体性地转存到外存空间，直到进程重新获得 CPU 之后才被整体装回内存。显然这一交换操作回影响到效率。

70 年代后，按需调页算法得到应用，该算法以页为单位进行转出和调入，大幅度提高了读写效率。现在，驻留 CPU 的体系结构都支持按需调页策略，Linux 也采用此策略进行虚拟存储管理。

在 Linux 中，内核态内存空间的内容不允许患处，道理很简单，因为驻留该空间的函数和数据结构都用于系统管理，有的甚至是为虚拟存储管理服务的，必须时刻准备着给 CPU 使用。

Linux 采用两种方式保存换出地页面。一是使用整个块设备，如硬盘的一个分区，称作交换设备；另一种是使用文件系统的一个固定长度的文件，称作交换文件。两者统称为交换空间。

交换设备和交换文件的内部格式是一致的。前 4096 个字节是一个以字符串“SWAP_SPACE”结尾的位图。位图的每一位对应于一个交换空间地页面，置位表示对应的页面可以用于换页操作。第 4096 字节之后是真正存放换出页面的空间。这样每个交换空间最多可以容纳（4096-10）*8-1=32687 个页面。如果一个交换空间不够用，Linux 最多允许管理 MAX_SWAPFILES（缺省值为 8）个交换空间。

交换设备远比交换文件更加有效。在交换设备中，属于同一页面的数据总是连续存放的，第一个数据块地址一经确定，后续的数据块可以按照顺序读出或写入。而在交换文件中，属于同一页面的数据虽然在逻辑上是连续的，但实际存储缺决定于拥有交换文件的文件系统。在大多数文件系统中，一个页面的物理存储是零散的，交换这样的页面，必须多次访问磁盘扇区，这意味着磁头的反复移动、寻道时间的增加和效率的降低。

4.7.5 页的换进换出

4.7.5.1 页交换进程和页面换出

当物理页面不够用时，Linux 存储管理系统必须释放部分物理页面，把它们的内容写倒交换空间。内核态交换线程 kswapd 完成这项功能，注意内核态线程是没有虚拟空间的线程，它运行在内核态，只见使用物理地址空间。

Kswapd 不仅能够把页面换出到交换空间，也能保证系统中有足够的空闲页面以保持存储管理系统高效的运行。

Kswapd 在系统初启时由 init 创建，然后调用 init_swap_timer()函数进行设定，并马上转入睡眠。以后每隔 10ms 响应函数 swap_tick()被周期性激活，它首先察看系统中空闲页面是否变得太少，如果空闲页面足够，kswapd 继续睡眠，否则环形 kswapd 处理。Kswapd 依次从三条途径缩减系统使用的物理页面：

- 缩减 page cache 和 buffer cache；
- 换出 SYSTEM V 共享内存占用得页面；
- 换出或丢弃进程占用得页面。

4.7.5.2 缺页中断和页面换入

磁盘中的可执行文件映像（image）一旦被映射到一个进程的虚拟空间，它就开始执行。由于一开始只有该映像区的开始部分被调入内存，因此进程迟早会执行那些未被装入内存的部分。当一个进程访问了一个还没有有效页表项的虚拟地址时，处理器将产生缺页中断，通知操作系统，并把缺页的虚拟地址（保存在 CR2 寄存器中）和缺页时访问虚存的模式一并传给 Linux 的缺页中断处理程序。

系统初始化时首先设定缺页中断处理程序位 do_page_fault()。根据控制寄存器 CR2 传递的缺页地址，Linux 必须找到用来表示出现缺页的虚拟存储区的 vm_area_struct 结构，如果没有找到，那么说明进程访问了一个非法存储区，系统将发出一个信号告知进程出错。然后系统检测缺页时访问模式是否合法，如果进程对该页的访问超越权限，系统也将发出一个信号，通知进程的存储访问出错。通过以上两步检查，可以确定缺页中断是否合法，进而进程进一步通过页表项中的位 P 来区分缺页对应的页面是在交换空间（P=0 且页表项非空）还是在磁盘中某一执行文件映像的一部分。最后进行页面调入操作。

习题

1. 简述存储管理的基本功能。
2. 叙述计算机系统中的存储器层次，为什么要配置层次式存储器?
3. 什么是逻辑地址(空间)和物理地址(空间)。
4. 何谓地址转换(重定位)?有哪些方法可以实现地址转换?
5. 分区存储管理中常用哪些分配策略？比较它们的优缺点。
6. 什么是移动技术？什么情况下采用这种技术？

7. 若采用表格方式管理可变分区，试画出分配和释放一个存储区的算法流程
8. 什么是存储保护？分区存储管理中如何实现分区的保护？
9. 什么是虚拟存储器？采用虚拟存储技术的必要性和可能性是什么？
10. 试述分页式虚拟存储管理的实现原理。
11. 试述分段式虚拟存储管理的实现原理。
12. 分页式存储管理中有哪几种常见的页面淘汰算法。
13. 试比较分页式存储管理和分段式存储管理
14. 试给出几种存储保护方法，各运用何种场合？
15. 试述存储管理中的碎片，各种存储管理中可能产生何种碎片。
16. 在一个请页式存储管理系统中，一个程序运行的页面走向是：6、5、4、3、2、1、1、5、4、6、3、2、1、4、6、5 分别用 FIFO 和 LRU 算法，对分配给程序 4 个页面和 5 个页面的情况下，分别求出缺页中断次数和缺页中断率。
17. 一个页式存储管理系统使用 LRU 页面替换算法，如果一个作业的页面走向为：1、2、3、4、5、2、1、3、4，当分配给该作业的物理块数分别为 3 和 4 时，试计算访问过程中发生的缺页中断次数和缺页中断率。
18. 在可变分区存储管理下，按地址排列的内存空闲区为：10K、4K、20K、18K、7K、9K、12K 和 15K。对于下列的连续存储区的请求：1）12K、2）10K、3）9K，使用首次适应算法，将使用哪一个空闲区？使用最佳适应算法和最差适应算法呢？
19. 一个 32 位地址的计算机系统使用二级页表，虚地址被分为 9 位顶级页表，11 位二级页表。试问：页区长度是多少？虚地址空间共有多少个页面？
20. 一进程以下列次序访问 5 个页：A、B、C、D、A、B、E、A、B、C、D、E；假定使用 FIFO 替换算法，在内存有 3 个和 4 个空闲页框的情况下，分别给出页面替换次数。
21. 采用分区方式进行存储管理，做如允许用户运行时动态中请/归还主存资源，这时，系统可能因竞争主存资源而产生死锁吗？如果否，说明之；如果是，试设计一种解决死锁的方案。
22. 试论述分页式存储管理中，决定页面大小的主要因素。
23. 叙述实现虚拟存储器的基本原理。
24. 采用页式存储管理的存储器是否就是虚拟存储器？为什么？实现虚拟存储器必须要有哪些硬件/软件设施支撑。
25. 如果主存中某页正在与外围设备交换信息，那么发生缺页中断时，可以将该页淘汰吗？为什么？出现这种情况时，你能提出什么样的处理办法？
26. 为什么在页式存储器中实现程序共享时，必须对共享程序给出相同的页号？
27. 在段式存储器中实现程序共享时，共享段的段号是否一定要相同？为什么？
28. 叙述段页式存储器的主要优点。
29. 某计算机有缓存、内存、辅存来实现虚拟存储器。如果数据在缓存中，访问它需要 Ans；如果在内存但不在缓存，需要 Bns 将其装入缓存，然后才能访问；如果不在内存而在辅存，需要 Cns 将其读入内存，然后，用 Bns 再读入缓存。假设缓存命中率为（n-1）/n，内存命中率为（m-1）/m，则数据平均访问时间是多少？
30. 什么叫‘抖动’？试给出一个抖动的例子。
31. 有一个分页系统，其页表存放在主存里，1）如果对内存的一次存取要 1.2 微秒，试向实现一次页面访问的存取需花多少时间？2）若系统配置了联想存储器，命中率为 75%，假定页表表目在联想存储器的查找时间为 0，试问实现一次页面访问的存取时

间是多少？

32. 给定段表如下：

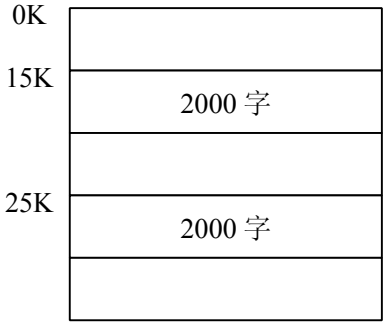
段　　号	段　首　址	段　　长
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

给定这样地址为段号和位数：1）[0,430]、2）[3,400]、3) [1,1]、4) [2,500]、5) [4,42]，试求出对应的内存物理地址。

33. 某计算机系统提供 24 位虚存空间，主存为 2^{18} 字，采用分页式虚拟存储管理，页面尺寸为 256 字。假定用户程序产生了虚拟地址 11123456（八进制），说明该系统如何产生相应的物理地址。

34. 在可变分区存储管理中，回收一个分区时有多种不同的邻接情况，试讨论各种情况的处理方法。

35. 主存中有两个空间区如图所示，



现有作业序列：Job1 要求 1200 字；Job2 要求 700 字；Job3 要求 1400 字；使用首次适应和最佳适应算法对处理这个作业序列，试问哪种算法可以满足分配？为什么？

36. 某可变分区存储管理系统中，空闲区按地址从小到大排列为：10K、4K、20K、18K、7K、9K、12K 和 15K，对于内存需求：1）12K；2）10K；3）9K；使用（1）首次适应算法，哪个空闲区被使用？2）最佳适应算法，哪个空闲区被使用？3）最差适应算法，哪个空闲区被使用？4）下次适应算法，哪个空闲区被使用？

CH5 设备管理

现代计算机系统中配置了大量外围设备。一般说，计算机的外围设备分为两大类：一类是存储型设备，如磁带机、磁盘机等。以存储大量信息和快速检索为目标，它在系统中作为主存储器的扩充，所以，又称辅助存储器；另一类是输入输出型设备，如显示器、卡片机、打印机等。它们把外界信息输入计算机，把运算结果从计算机输出。

设备管理是操作系统中最庞杂和琐碎的部分，普遍使用 I/O 中断、缓冲器管理、通道、设备驱动调度等多种技术，这些措施较好地克服了由于外部设备和主机速度上不配所引起的问题，使主机和外设并行工作，提高了使用效率。但是，在另一方面却给用户的使用带来极大的困难，它必须掌握 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，这就使计算机推广应用受到很大限制。

为了方便用户使用各种外围设备，设备管理要达到提供统一界面、方便使用、发挥系统并行性，提高 I/O 设备使用效率等目标。为此，设备管理通常应具有以下功能：

- 外围设备中断处理
- 缓冲区管理
- 外围设备的分配
- 外围设备驱动调度
- 虚拟设备及其实现

其中，前四项是设备管理的基本功能，最后一项是为了进一步提高系统效率而设置的，往往在规模较大操作中才提供，每一种功能对不同的系统、不同的外围设备配置也有强有弱。

5.1 I/O 硬件原理

不同的人对于 I/O 硬件有着不同的理解。在电气工程师看来，I/O 硬件就是一堆芯片、电线、电源、马达和其他设备的集合体；而程序员则主要注意它为软件提供的接口，即硬件能够接受的命令、它能够完成的功能、以及能报告的各种错误等。作为操作系统的设计者，我们的立足点主要是针对如何利用 I/O 硬件的功能进行程序设计提供一个方便用户的实用接口，而并非研究 I/O 硬件的设计、制造和维护。

5.1.1 I/O 系统

通常把 I/O 设备及其接口线路、控制部件、通道和管理软件称为 I/O 系统，把计算机的主存和外围设备的介质之间的信息传送操作称为输入输出操作。随着计算机技术的飞速进步和应用领域扩大，计算机的输入输出信息量急剧增加，I/O 设备的种类和数量越来越多，它们与主机的联络与信息交换方式越来越各不相同。输入输出操作不仅影响计算机的通用性和扩充性，而且成为计算机系统综合处理能力及性能价格比的重要因素。

按照输入输出特性，I/O 设备可以划分为输入型外围设备、输出型外围设备和存储型外围设备三类。按照输入输出信息交换的单位，I/O 设备则可以划分为字符设备和块设备。输入型外围设备和输出型外围设备一般为字符设备，它与内存进行信息交换的单位是字节，即一次交换 1 个或多个字节。所谓块是连续信息所组成的一个区域，块设备则一次与内存交换的一个或几个块的信息，存储型外围设备一般为块设备。

存储型外围设备又可以划分为顺序存取存储设备和直接存取存储设备。顺序存取存储设备严格依赖信息的物理位置进行定位和读写，如磁带。直接存取存储设备的重要特性是存取任何一个物理块所需的事件几乎不依赖于此信息的位置，如磁盘。

不同设备的物理特性存在很大差异，其主要差别在于:

- 数据传输率　从每秒几十个字符(键盘输入)到每秒几个 KB(磁盘),相差万倍;
- 数据表示方式　不同设备采用不同字符表和奇偶校验码;
- 传输单位　慢速设备以字符为单位,快速设备以块为单位,可相差几千倍;
- 出错条件　错误的性质、形式、后果、报借方法，应对措施等每类设备都不一样。

这些差异使得不论从操作系统还是从用户角度,都难以获得一个规范一致的 I/O 解决方案。

5.1.2 I/O 控制方式

输入输出控制在计算机处理中具有重要的地位，为了有效地实现物理 I/O 操作，必须通过硬、软件技术，对 CPU 和 I/O 设备的职能进行合理分工，以调解系统性能和硬件成本之间的矛盾。按照 I/O 控制器功能的强弱，以及和 CPU 之间联系方式的不同，可把 I/O 设备的控制方式分为四类，它们的主要差别在于中央处理器和外围设备并行工作的方式不同，并行工作的程度不同。中央处理器和外围设备并行工作有重要意义，它能大幅度提高计算机效率和系统资源的利用率。

5.1.2.1 询问方式

询问方式又称程序直接控制方式，在这种方式下，输入输出指令或询问指令测试一台设备的忙闲标志位，决定主存储器和外围设备是否交换一个字符或一个字。下面我们来看一下数据输入的过程，如图 5-1 所示，假如 CPU 上运行的现行程序需要从 I/O 设备读入一批数据，CPU 程序设置交换字节数和数据读入主存的起始地址，然后，向 I/O 设备发读指令或查询标志指令，I/O 设备便把状态返回给 CPU。如果 I/O 忙或未就绪，则重复上述测试过程，继续进行查询；如果 I/O 设备就绪，数据传送便开始，CPU 从 I/O 接口读一个字，再向主存写一个字。如果传送还未结束，再次向设备发出读指令，直到全部数据传输完成再返回现行程序执行。为了正确完成这种查询，通常要使用三条指令：(1) 查询指令，用来查询设备是否就绪；(2) 传送指令，当设备就绪时，执行数据交换；(3) 转移指令，当设备未就绪时，执行转移指令转向查询指令继续查询。需要注意，这种方式数据传输，需要用到 CPU 的寄存器，由于传送的往往是一批数据，需要设置交换数据的计数值，以及数据在内存缓冲区的首址。数据输出的过程类似不赘。

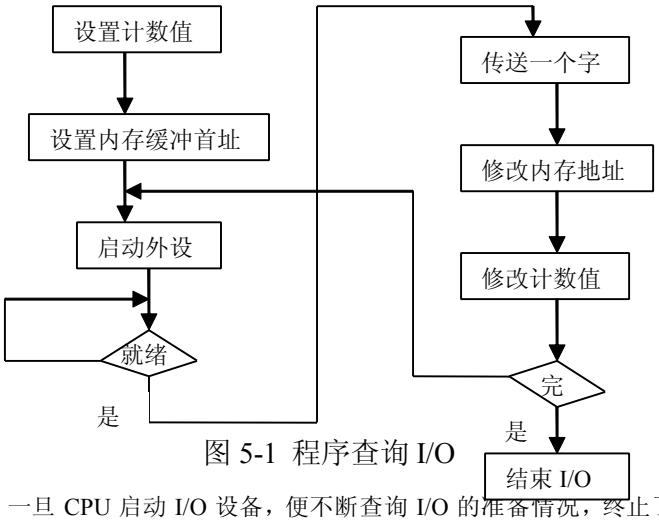


图 5-1 程序查询 I/O

由上述过程可见，一旦 CPU 启动 I/O 设备，便不断查询 I/O 的准备情况，终止了原程序的执行。CPU 在反复查询过程中，浪费了宝贵的 CPU 时间；另一方面，I/O 准备就绪后，CPU 参与数据的传送工作，此时 CPU 也不能执行原程序，可见 CPU 和 I/O 设备串行工作，使主机不能充分发挥效率，外围设备也不能得到合理使用，整个系统的效率很低。

5.1.2.2 中断方式

中断机构引入后，外围设备有了反映其状态的能力，仅当操作正常或异常结束时才中断中央处理机。实现了一定程度的并行操作，这叫程序中断方式。仍用上述例子来说明，假如 CPU 在启动 I/O 设备后，不必查询 I/O 设备是否就绪，而是继续执行现行程序，对设备是否就绪不加过问。直到在启动指令之后的某条指令(如第 K 条) 执行完毕，CPU 响应了 I/O 中断请求，才中断现行程序转至 I/O 中断处理程序执行。在中断处理程序中，CPU 全程参与数据传输操作，它从 I/O 接口读一个字(字节) 并写入主存，如果I/O 设备上的数据尚未传送完成，转向现行程序再次启动 I/O 设备，于是命令 I/O 设备再次作准备并重复上述过程；否则，中断处理程序结束后，继续从 K+1 条指令执行。图 5-2 为程序中断方式工作流程。

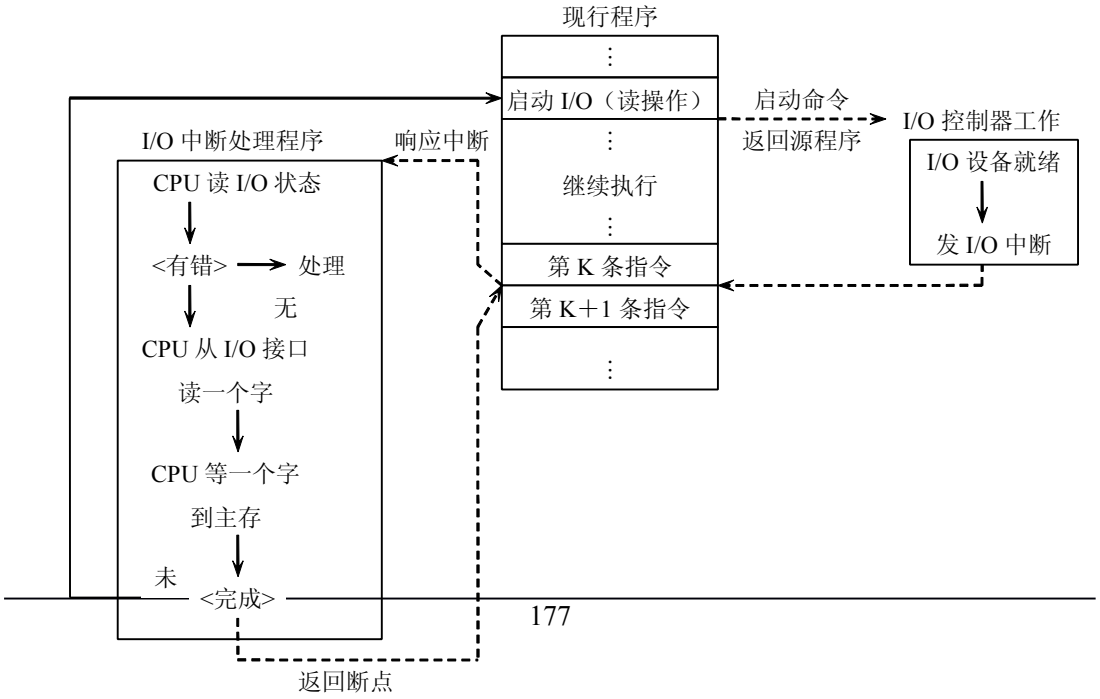


图 5-2　程序中断方式 I/O

但是，由于输入输出操作直接由中央处理器控制，每传送一个字符或一个字，都要发生一次中断，因而仍然消耗大量中央处理器时间。例如，输入机每秒传送 1000 个字符，若每次中断处理平均花 100 微秒，为了传输 1000 个字符，要发生 1000 次中断，所以，每秒内中断处理要花去约 100 毫秒。但是程序中断方式 I/O，由于不必忙式查询 I/O 准备情况，在 CPU 和 I/O 设备可实现部分并行，与程序查询的串行工作方式相比，使 CPU 资源得到较充分利用。

5.1.2.3 DMA方式

虽然程序中断方式消除了程序查询方式的忙式测试，提高了 CPU 资源的利用率，但是在响应中断请求后，必须停止现行程序转入中断处理程序并参与数据传输操作。如果 I/O 设备能直接与主存交换数据而不占用 CPU，那么，CPU 资源的利用率还可提高，这就出现了直接存储器存取(Direct Memory Access，DMA)方式。

在 DMA(直接主存存取)方式中，主存和 I/O 设备之间有一条数据通路，在主存和 I/O 设备之间成块传送数据过程中，不需要 CPU 干予，实际操作由 DMA 直接执行完成。图 5-3 为 DMA 方式 I/O 流程，为此，DMA 至少需要以下逻辑部件：

- 主存地址寄存器　存放主存中需要交换数据的地址，DMA 传送前，由程序送入首地址，在 DMA 传送中，每交换一次数据，把地址寄存器内容加 1。
- 字计数器　记录传送数据的总字数，每传送一个字，字计数器减 1。
- 数据缓冲寄存器或数据缓冲区　暂存每次传送的数据。DMA 与主存间采用字传送，DMA 与设备间可能是字位或字节传送。所以，DMA 中还可能包括有数据移位寄存器、字节计数器等硬件逻辑。可能有人会提出疑问：为什么控制器从设备读到数据后不立即将其送入内存，而是需要一个内部缓冲区呢？原因是一旦磁盘开始读数据，从磁盘读出比特流的速率时恒定的，不论控制器是否做好接受这些比特的准备。若此时控制器要将数据直接拷贝到内存中，则它必须在每个字传送完毕后获得对系统总线的控制权。如果由于其他设备争用总线，则只能等待。当上一个字还未送入内存前另一个字到达时，控制器只能另找一个地方暂存。如果总线非常忙，则控制器可能需要大量的信息暂存，而且要做大量的管理工作。从另一方面来看，如果采用内部缓冲区，则在 DMA 操作启动前不需要使用总线，这样控制器的设计就比较简单，因为从 DMA 到主存的传输对时间要求并不严格。
- 设备地址寄存器　存放 I/O 设备信息，如磁盘的柱面号、磁道号、块号。
- 中断机制和控制逻辑　用于向 CPU 提出 I/O 中断请求和保存 CPU 发来的 I/O 命令及管理 DMA 的传送过程。

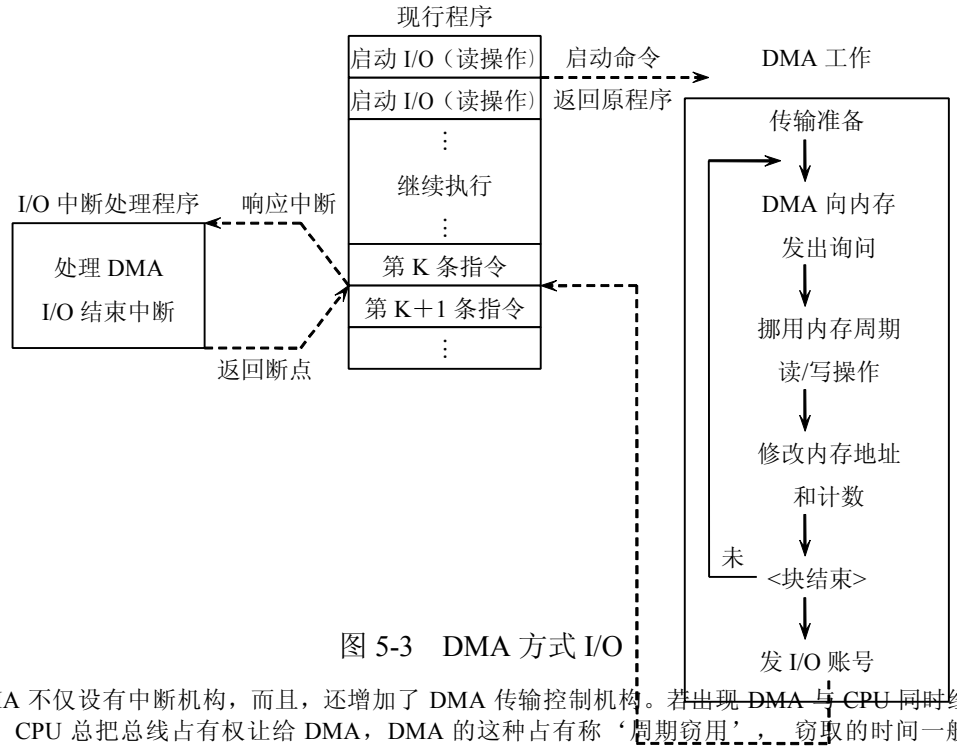


图 5-3　DMA 方式 I/O

DMA 不仅设有中断机构，而且，还增加了 DMA 传输控制机构。若出现 DMA 与 CPU 同时经总线访问主存，CPU 总把总线占有权让给 DMA，DMA 的这种占有称‘周期窃用’，窃取的时间一般为一个存取周期，让设备和主存之间交换数据，而

且在 DMA 周期窃取期间，非但不要 CPU 干予，CPU 尚能做运算操作。这样可减轻 CPU 的负担，每次传送数据时，不必进入中断系统，进一步提高了 CPU 的资源利用率。

并非所有的计算机都使用 DMA，反对意见认为 CPU 比 DMA 控制器快的多，当 I/O 设备的速度不构成瓶颈时，CPU 完全可以更快的完成这项工作。特别对于个人计算机来说，如果 CPU 无私可作，而又被迫等待慢速的 DMA 控制器，是完全没有意义的。同时。省去 DMA 控制器还可以节省一些成本。

目前，在小型、微型机中的快速设备均采用这种方式，DMA 方式线路简单，价格低廉，但功能较差，不能满足复杂的 I / O 要求。因而，在中大型机中使用通道技术。

5.1.2.4 通道方式

DMA 方式与程序中断方式相比，又减少了 CPU 对 I/O 的干予，已经从字(字节) 为单位的干予减少到以数据块为单位的干予。而且，每次 CPU 干予时，并不要做数据拷贝，仅仅需要发一条启动 I/O 指令，以及完成 I/O 结束中断处理。但是，每发出一次 I/O 指令，只能读写一个数据块，如果用户希望一次读写多个离散的数据块，并能把它们传送到不同的内存区域，或相反时，则需要由 CPU 分别发出多条启动 I/O 指令及进行多次 I/O 中断处理才能完成。通道方式是 DMA 方式的发展，它又进一步减少了 CPU 对 I/O 操作的干予，减少为对多个数据块，而不是仅仅一个数据块，及有关管理和控制的干予。同时，为了获得中央处理器和外围设备之间更高的并行工作能力，也为了让种类繁多，物理特性各异的外围设备能以标准的接口连接到系统中，计算机系统引入了自成独立体系的通道结构。通道的出现是现代计算机系统功能不断完善，性能不断提高的结果，是计算机技术的一个重要进步。

通道又称输入输出处理器。它能完成主存储器和外围设备之间的信息传送，与中央处理器并行地执行操作。采用通道技术主要解决了输入输出操作的独立性和各部件工作的并行性。由通道管理和控制输入输出操作，大大减少了外围设备和中央处理器的逻辑联系。从而，把中央处理器从琐碎的输入输出操作中解放出来。此外，外围设备和中央处理器能实现并行操作；通道和通道之间能实现并行操作；各通道上的外围设备也能实现并行操作，以达到提高整个系统效率这一根本目的。

具有通道装置的计算机，主机、通道、控制器和设备之间采用四级连接，实施三级控制。通常，一个中央处理器可以连接若干通道，一个通道可以连接若干控制器，一个控制器可以连接若于台设备。中央处理器执行输入输出指令对通道实施控制，通道执行通道命令(CCW)对控制器实施控制，控制器发出动作序列对设备实施控制，设备执行相应的输入输出操作。

采用输入输出通道设计后，输入输出操作过程如下：中央处理机在执行主程序时遇到输入输出请求，则它启动指定通道上选址的外围设备，一旦启动成功，通道开始控制外围设备进行操作。这时中央处理器就可执行其它任务并与通道并行工作，直到输入输出操作完成。通道发出操作结束中断时，中央处理器才停止当前工作，转向处理输入输出操作结束事件。

按照信息交换方式和加接设备种类不同，通道可分为三种类型：

- 字节多路通道。它是为连接大量慢速外围设备，如软盘输入输出机、纸带输入输出机、卡片输入输出机、控制台打字机等设置的。以字节为单位交叉地工作，当为一台设备传送一个字节后，立即转去为另一台设备传送一个字节。在 IBM370 系统中，这样的通道可接 256 台设备。
- 选择通道。它用于连接磁带和磁盘快速设备。以成组方式工作，每次传送一批数据；故传送速度很高，但在这段时间只能为一台设备服务。每当一个输入输出操作请求完成后，再选择与通道相连接的另一设备。
- 数组多路通道。对于磁盘这样的外围设备，虽然传输信息很快，但是移臂定位时间很长。如果接在字节多路通道上，那么通道很难承受这样高的传输率；如果接在选择通道上，那么；磁盘臂移动所花费的较长时间内，通道只能空等。数组多路通道可以解决这个矛盾，它先为一台设备执行一条通道命令，然后自动转换，为另一台设备执行一条通道命令。对于连接在数组多路通道上的若干台磁盘机，可以启动它们同时进行移臂，查找欲访问的柱面，然后，按次序交叉传输一批批信息，这样就避免了移臂操作过长地占用通道。由于它在任一时刻只能为一台设备作数据传送服务，这类似于选择通道；但它不等整个通道程序执行结束就能执行另一设备的通道程序命令，这类似于字节多路通道。数组多路通道的实质是：对通道程序采用多道程序设计技术的硬件实现。

有关通道方式 I/O 的基本原理将在 CH6 进行介绍。

5.1.3 设备控制器

I/O 设备通常包括一个机械部件和一个电子部件。为了达到设计的模块性和通用性，一般将其分开。电子部件称为设备控制器或适配器，在个人计算机中，它常常是一块可以插入主板扩充槽的印刷电路板；机械部件则是设备本身。

控制器卡上一般都有一个接线器，它可以把与设备相连的电缆线解进来。许多控制器可以控制 2 个、4 个甚至 8 个相同设备。控制器和设备之间的接口越来越多的采用国际标准，如 ANSI、IEEE、ISO、或者事实上的工业标准。依据这些标准，各个计算机厂家都可以制造与标准接口相匹配的控制器和设备。例如 IDE（集成设备电子器件）接口、SCSI（小型计算机系统接口）接口的硬盘。

之所以区分控制器和设备本身是因为操作系统基本上是与控制器打交道，而非设备本身。大多数微

型计算机的 CPU 和控制器之间的通信采用单总线模型，CPU 直接控制设备控制器进行输入输出；而主机则采用多总线结构和通道方式，以提高 CPU 与输入输出的并行程度。

控制器与设备之间的接口是一种很低层次的接口。例如一个磁盘，可以被格式化成为一个每道 16 个 512 字节的扇区，实际从磁盘读出来的是一个比特流，以一个前缀开始，随后是一个扇区的 4096 比特，最后是一个纠错码 ECC；其中前缀是磁盘格式化时写进的，包括柱面数、扇区数、扇区大小等，以及同步信息。控制器的任务是把这个串行的转换成字节块并在必要时进行纠错，通常该字节块是在控制其中的一个缓冲区中逐个比特汇集而成，在检查和校验后，该块数据将被拷贝到主存中。

CRT 控制器也是一个比特串行设备，它从内存中读取欲显示字符的字节流，然后产生用来调制 CRT 射线的信号，最后将结果显示在屏幕上。控制器还产生当水平方向扫描结束后的折返信号以及当整个屏幕被扫描后的垂直方向的折返信号。

不难看出，如果没有控制器，这些复杂的操作必须由操作系统程序员自己编写程序来解决；而引入了控制器后，操作系统只需通过传递几个简单的参数就可以对控制器进行操作和初始化，从而大大简化了操作系统的设计，特别是有利于计算机系统和操作系统对各类控制器和设备的兼容性。

每个控制器都有一些用来与 CPU 通信的寄存器，在某些计算机上，这些寄存器占用内存地址的一部分，称为内存映像 I/O；另一些计算机则采用 I/O 专用地址，每个寄存器占用其中的一部分。设备的 I/O 地址分配由控制器上的总线解码逻辑完成。除 I/O 端口外，许多控制器还通过中断通知 CPU 它们已经做好准备，寄存器可以读写。以 IBM 奔腾系列为例，它向 I/O 设备提供 15 条可用中断。

有些控制器做在计算机主板上，如 IBM PC 机的键盘控制器。对于那些单独插在主板插槽上的控制器，有时上面设有一些可以用来设置 IRQ 号的开关和跳线，以便避免 IRQ 冲突。中断控制器芯片将每个 IRQ 输入并映像到一个中断向量，通过这个中断向量就可以找到相应的中断服务程序。图 5-4 给出了 PC 机部分控制器的 I/O 地址、硬件中断和中断向量号。

操作系统通过向控制器寄存器写命令字来执行 I/O 功能。例如 PC 机的软盘控制器可以接收 15 条命令，包括读、写、格式化、重新校准等。许多命令字带有参数，这些参数也要同时装入控制器寄存器。一旦某个控制器接受到一条命令后，CPU 可以转向其它工作，而让该设备控制器自行完成具体的 I/O 操作。当命令执行完毕后，控制器发出一个中断信号，以便使操作系统重新获得 CPU 的控制权并检查执行结果，此时 CPU 仍旧是从控制器寄存器中读取若干字节信息来获得执行结果和设备的状态信息。

I/O 控制器	I/O 地址	硬件中断号	中断向量号
时钟	040--043	0	8
键盘	060--063	1	9
硬盘	1F0—1F7	14	118
软盘	3F0-3F7	6	14
LPT1	378—37F	7	15
COM1	3F8—3FF	4	12
COM2	2F8-2FF	3	11

图 5-4　PC 机部分控制器的 I/O 地址、硬件中断和中断向量号

我们来小结一下设备控制器的功能和结构。设备控制器是 CPU 和设备之间的一个接口,它接收从 CPU 发来的命令,控制 I/O 设备操作,实现主存和设备之间的数据传输操作。设备控制器是一个可编址设备,当它连接多台设备时,则应具有多个设备地址。设备控制器的主要功能为:①接收和识别 CPU 或通道发来的命令,例如磁盘控制器能接收读、写、查找、搜索等各种命令;②实现数据交换,包括设备和控制器之间的数据传输;通过数据总线或通道,控制器和主存之间的数据传输;③发现和记录设备及自身的状态信息,供 CPU 处理使用;④设备地址识别。为了实现上面列举的各项功能，设备控制器必须有以下组成部分:命令寄存器及译码器,数据寄存器,状态寄存器,地址译码器,以及用于对设备操作进行控制的 I/O 逻辑。

5.2 I/O 软件原理

5.2.1 I/O 软件的设计目标和原则

I/O 软件的总体设计目标是:高效率和通用性。高效率是不言而语的,在改善 I/O 设备的效率中,最应关注的是磁盘 I/O 的效率。通用性意味着用统一标准的方法来管理所有设备，为了达到这一目标，通常，把软件组织成一种层次结构，低层软件用来屏蔽硬件的具体细节，高层软件则主要向用户提供一个简洁、规范的界面。

I/O 软件设计主要要考虑以下 4 个问题：

- 设备无关性。即程序员写出的软件在访问不同的外围设备时应该尽可能地与设备的具体类型无关，如访问文件是不必考虑它是存储在硬盘、软盘还是 CD-ROM 上。
- 出错处理。总的来说，错误应该在尽可能靠近硬件的地方处理，在低层软件能够解决的错误不让高层软件感知，只有低层软件解决不了的错误才通知高层软件解决。
- 同步（阻塞）——异步（中断驱动）传输。多数物理 I/O 是异步传输，即 CPU 在启动传输操作后便转向其他工作，直到中断到达。I/O 操作可以采用阻塞语义，发出一条 READ 命令后，程序将自动被挂起，直到数据被送到内存缓冲区。
- 独占性外围设备和共享性外围设备。某些设备可以同时为几个用户服务，如磁盘；另一些设备在某一段时间只能供一个用户使用，如键盘。独占性外围设备和共享性外围设备带来了许多问题，操作系统必须能够同时加以解决。

为了合理、高效地解决以上问题，操作系统通常把 I/O 软件组织成以下四个层次。

- I/O 中断处理程序（底层）。
- 设备驱动程序。
- 与设备无关的操作系统 I/O 软件。
- 用户层 I/O 软件。

5.2.2 I/O 中断处理程序

中断是应该尽量加以屏蔽的概念，应该放在操作系统的底层进行处理，一边其余部分尽可能少地与之发生联系。

当一个进程请求 I/O 操作时，该进程将被挂起，直到 I/O 操作结束并发生中断。当中断发生时，中断处理程序执行相应的处理，并解除相应进程的阻塞状态。

输入输出中断的类型和功能如下：

- 通知用户程序输入输出操作沿链推进的程度。此类中断有程序进程中断。
- 通知用户程序输入输出操作正常结束。当输入输出控制器或设备发现通道结束、控制结束、设备结束等信号时，就向通道发出一个报告输入输出操作正常结束的中断。
- 通知用户程序发现的输入输出操作异常，包括设备出错、接口出错、I/O 程序出错、设备特殊、设备忙等，以及提前中止操作的原因。
- 通知程序外围设备上重要的异步信号。此类中断有注意、设备报到、设备结束等。

当输入输出中断被响应后，中断装置交换程序状态字引出输入输出中断处理程序。输入输出中断处理程序 PSW 中得到产生中断的通道号和设备号，并分析通道状态字，弄清产生中断的输入输出中断事件的原则如下：

1）如果是操作正常结束，那么，系统要查看是否有等待该设备或通道者，若有则释放。例如接在数组多路通道上的某台行式打印机，当完成主存到行打机缓冲器之间的一 行信息后，虽然行式打印机并未完成这一行通道上的另一台设备。当行式打印机打印出一行信息，完成了一次输出的所有任务后，它还要发出中断请求报告系统。操作系统分析通道状态字的设备状态字节使可知道是“通道结束”还是“设备结束”，从而释放等待通道者或释放等待设备等。

2）如果由于操作中发生故障或某种特殊事件而产生的中断，那么，操作系统要进一步查明原因，采取相应措施。

操作中发生的故障及其处理的方法可能有以下几种：

- 设备本身的故障。例如读写操作中校验装置发现的错误。操作系统可以从设备状态字节的“设备错误”位为 1 来发现这类故障。系统处理这种故障时，先向相应设备发命令索取断定状态字节，然后分析断定状态字节就可以知道故障的确切原因。如果该外围设备的控制器没有复执功能，那么，对于某些故障，系统可组织软复执。例如读磁带上的信息，当校验装置发现错误时，操作系统可组织回退，再读若干遍。对于不能复执的故障或复执多次仍不能克服的故障，系统将向操作员报告，请求人工干预。
- 通道的故障。对于这种故障也可进行复执。如果硬件已具备复执功能或软复执比较困难，那么，系统应将错误情况报告给操作员。
- 通道程序错。由通道识别的各种通道程序错误，例如通道命令非双字边界；通道命令地址无效；通道命令的命令码无效；ＣＡＷ格式错；连用两条通道转移命令等，均由系统报告给操作员。
- 启动命令的错误。例如启动外围设备的命令要求从输入机上读入 1000 个字符，然而，读了 500 个字符就遇到“停码”，输入机硬停止了。操作系统从通道状态字节的长度错误位为 1 可判断这类错误，再把处理转交给用户，例如转向用户 程序的中断续元，由用户自己处理。

如果设备在操作中发生了某些特殊事件，那么，在设备操作结束发生中断时，也要将这个情况向系统报告。操作系统从设备状态字节中的设备特殊位为 1 ，可以判知设备在操作中发生了某个特殊事件。对于磁带机，这意味着在写入一块信息遇到了带末点或读出信息时遇到了带标。在写操作的情况，系统知道磁带即将用完，如果文件还未写完，应立即组织并写入卷尾标，然后，通知操作员换卷以便将文件

的剩余部分写在后继卷上。在读操作的情况，系统判知这个文件已经读完或这个文件在此卷上的部分已经读完，进行文件结束的处理；若只读了一部分，则带标后面是卷尾标，系统将通知操作员换卷，以便继续读入文件。对于行式打印机，这意味着纸将用完，因此，系统可暂停输出，通知操作员装纸，然后继续输出。

3）如果是人为要求而产生的中断，那么，系统将响应并启动外围设备。例如要求从控制台打字机输入时，操作员先揿“询问键”，随之产生中断请求。操作系统从设备状态字节的“注意”位为 1 就知道控制台打字机请求输入。此时，系统启动控制台打字机并开放键盘，接着操作员便可打入信息。

4）如果是外围设备上来的“设备结束”等异步信号，表示有外围设备接入可供使用或断开暂停使用。操作系统应修改系统表格中相应设备的状态。

5.2.3 设备驱动程序

设备驱动程序中包括了所有与设备相关的代码。每个设备驱动程序只处理一种设备，或者一类紧密相关的设备。例如，若系统所支持的不同品牌的所有终端只有很细微的差别，则较好的办法是为所有这些终端提供一个终端驱动程序。另一方面，一个机械式的硬拷贝终端和一个带鼠标的智能化图形终端差别太大，于是只能使用不同的驱动程序。

在本章的前半部分我们了解了设备控制器的功能，知道每个控制器都有一个或多个寄存器来接收命令。设备驱动程序发出这些命令并对其进行检查，因此操作系统中只有硬盘驱动程序才知道磁盘控制器有多少个寄存器，以及它们的用途。驱动程序知道使磁盘正确操作所需要的全部参数，包括扇区、磁道、柱面、磁头、磁头臂的移动、交叉系数、步进电机、磁头定位时间等等。

笼统地说，设备驱动程序的功能是从与设备无关的软件中接收抽象的请求，并执行之。一条典型的请求是读第 n 块。如果请求到来时驱动程序空闲，则它立即执行该请求。但如果它正在处理另一条请求，则它将该请求挂在一个等待队列中。

执行一条 I/O 请求的第一步，是将它转换为更具体的形式。例如对磁盘驱动程序，它包含：计算出所请求块的物理地址、检查驱动器电机是否在运转、检测磁头臂是否定位在正确的柱面等等。简而言之，它必须确定需要哪些控制器命令以及命令的执行次序。

一旦决定应向控制器发送什么命令，驱动程序将向控制器的设备寄存器中写入这些命令。某些控制器一次只能处理一条命令，另一些则可以接收一串命令并自动进行处理。

这些控制命令发出后有两种可能。在许多情况下，驱动程序需等待控制器完成一些操作，所以驱动程序阻塞，直到中断信号到达才解除阻塞。另一种情况是操作没有任何延迟，所以驱动程序无需阻塞。后一种情况的例子如：在有些终端上滚动屏幕只需往控制器寄存器中写入几个字节，无需任何机械操作，所以整个操作可在几微秒内完成。

对前一种情况，被阻塞的驱动程序须由中断唤醒，而后一种情况下它根本无需睡眠。无论哪种情况，都要进行错误检查。如果一切正常，则驱动程序将数据传送给上层的设备无关软件。最后，它将向它的调用者返回一些关于错误报告的状态信息。如果请求队列中有别的请求则它选中一个进行处理，若没有则它阻塞，等待下一个请求。

5.2.4 与硬件无关的操作系统 I/O 软件

尽管某些 I/O 软件是设备相关的，但大部分独立于设备。设备无关软件和设备驱动程序之间的精确界限在各个系统都不尽相同。对于一些以设备无关方式完成的功能，在实际中由于考虑到执行效率等因素，也可以考虑由驱动程序完成。

下面罗列了一般都是由设备无关软件完成的功能：

- 对设备驱动程序的统一接口
- 设备命名
- 设备保护
- 提供独立于设备的块大小
- 缓冲区管理
- 块设备的存储分配
- 独占性外围设备的分配和释放
- 错误报告

设备无关软件的基本功能能是执行适用于所有设备的常用 I/O 功能，并向用户层软件提供一个一致的接口。

操作系统的一个主要论题是文件和 I/O 设备的命名方式。设备无关软件负责将设备名映射到相应的驱动程序。在 Unix 中，一个设备名，如/dec/tty00 唯一地确定了一个 i-节点，其中包含了主设备号（major device number），通过主设备号就可以找到相应的设备驱动程序，i-节点也包含了次设备号（minor device number），它作为传给驱动程序的参数指定具体的物理设备。

与命令相关的是保护。操作系统如何保护对设备的未授权访问呢？多数个人计算机系统根本就不提供任何保护，所有进程都可以为所欲为。在多数大型主机系统中，用户进程绝对不允许访问 I/O 设备。

在 Unix 中使用一种更为灵活的方法。对应于 I/O 设备的设备文件的保护采用通常的 rwx 权限机制，所以系统管理员可以为每一台设备设置合理的访问权限。

不同磁盘的扇区大小可能不同，设备无关软件屏蔽了这一事实并向高层软件提供统一的数据块大小，比如将若干扇区作为一个逻辑块。这样高层软件就只和逻辑块大小都相同的抽象设备交互，而不管物理扇区的大小。类似地，有些字符设备（modem）对字节进行操作，另一些字符设备（网卡）则使用比字节大一些的单元，这类差别也可以进行屏蔽。

块设备和字符设备都需要缓冲技术。对于块设备，硬件每次读写均以块为单元，而用户程序则可以读写任意大小的单元。如果用户进程写半个块，操作系统将在内部保留这些数据，直到其余数据到齐后才一次性地将这些数据写到盘上。对字符设备，用户向系统写数据的速度可能比向设备输出的速度快，所以需要进行缓冲。超前的键盘输入同样也需要缓冲。

当创建了一个文件并向其输入数据时，该文件必须被分配新的磁盘块。为了完成这种分配工作，操作系统需要为每个磁盘都配置一张记录空闲盘块的表或位图，但写位一个空闲块的算法是独立于设备的，因此可以在高于驱动程序的层次处理。

一些设备，如 CD-ROM 记录器，在同一时刻只能由一个进程使用。这要求操作系统检查对该设备的使用请求，并根据设备的忙闲状况来决定是接受或拒绝此请求。一种简单的处理方法是通过直接用 OPEN 打开相应的设备文件来进行申请。苦设备不可用，则 OPEN 失败。关闭独占设备的同时将释放该设备。

错误处理多数由驱动程序完成。多数错误是与设备紧密相关的，因此只有驱动程序知道应如何处理（如重试、忽略、严重错误）。一种典型错误是磁盘块受损导致不能读写。驱动程序在尝试若干次读操作不成功后将放弃，并向设备无关软件报错。从此处往后错误处理就与设备无关了。如果在读一个用户文件时出借，则向调用者报错即可。但如果是在读一些关键系统数据结构时出错，比如磁盘使用状况位图，则操作系统只能打印出错信息，并终止运行。

5.2.5 用户空间的 I/O 软件

尽管大部分 I/O 软件属于操作系统，但是有一小部分是与用户程序链接在一起的库例程，甚至是在核心外运行的完整的程序。系统调用，包括 I/O 系统调用通常先是库例程调用。如下 C 语句

```
count = write（fd，buffer，nbytes）;
```

中，所调用的库函数 write 将与程序链接在一起，并包含在运行时的二进制程序代码中。这一类库例程显然也是 I/O 系统的一部分。

此类库例程的主要工作是提供参数给相应的系统调用并调用之。但也有一些库例程，它们确实做非常实际的工作，例如格式化输入输出就是用库例程实现的。C 语言中的一个例子是 printf 函数，它的输入为一个格式字符串，其中可能带有一些变量，它随后调用 write，输出格式化后的一个 ASCII 码串。与此类似的 scanf，它采用与 printf 相同的语法规则来读取输入。标准 I/O 库包含相当多的涉及 I/O 的库例程，它们作为用户程序的一部分运行。

并非所有的用户层 I/O 软件都由库例程构成。另一个重要的类别的就是 spooling 系统，spooling 是在多道程序系统中处理独占设备的一种方法。例如对于打印机，尽管可以采用打开其设备文件来进行申请，但假设一个进程打开它而长达几个小时不用，则其他进程都无法打印。

避免这种情况的方法是创建一个特殊的守护进程（daemon）以及一个特殊的目录，称为 spooling 目录。打印一个文件之前，进程首先产生完整的待打印文件并将其放在 spooling 目录下。而由该守护进程进行打印，这里只有该守护进程能够使用打印机设备文件。通过禁止用户直接使用打印机设备文件便解决了上述打印机空占的问题。

spooling 还可用于打印机以外的其他情况。例如在网络上传输文件常使用网络守护进程，发送文件前先将其放在一特定目录下，而后由网络守护进程将其取出发送。这种文件传送方式的用途之一是 Internet 电子邮件系统。Internet 通过许多网络将大量的计算机联在一起。当向某人发送 Email 时，用户使用某一个程序如 send，该程序接收要发的信件并将其送入一个固定的 spooling 目录，待以后发送。整个 Email 系统在操作系统之外运行。

图 5-5 总结了 I/O 系统，标示出了每一层软件及其功能。从底层开始分别是硬件、中断处理程序、设备驱动程序、设备无关软件，最上面是用户进程。

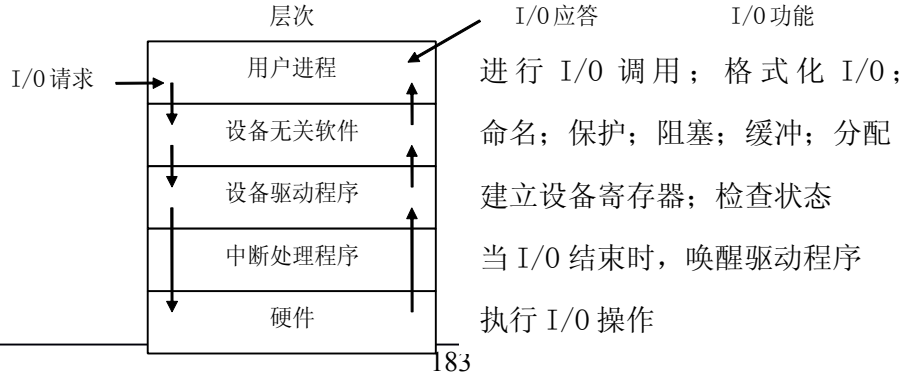


图 5-5　I/O 系统的层次及其功能

该图中的箭头表示控制流。如当用户程序试图从文件中读一数据块时，需通过操作系统来执行此操作。设备无关软件首先在数据块缓冲区中查找此块，若未找到，则它调用设备驱动程序向硬件发出相应的请求。用户进程随即阻塞直到数据块被读出。

当磁盘操作结束时，硬件发出一个中断，它将激活中断处理程序。中断处理程序则从设备获取返回状态值并唤醒睡眠的进程来结束此次 I/O 请求，并使用户进程继续执行。

5.3 具有通道的 I/O 系统管理

具有通道的计算机系统，输入输出程序设计涉及 CPU 执行 I/O 指令，通道执行通道命令，以及 CPU 和通道之间的通信。

5.3.1 通道命令和通道程序

5.3.1.1 通道命令

通道又称为 I/O 处理机，具有自己的指令系统，常常把 I/O 处理机的指令称通道命令。通道命令(Channel Command Word，CCW) 是通道从主存取出并控制 I/O 设备执行 I/O 操作的命令字，用通道命令编写的程序称通道程序，一条通道命令往往只能实现一种功能，由于通道程序由多条通道命令组成，每次启动就可以完成复杂的 I/O 控制。

IBM370 系统的通道命令双字长，格式如下：

通道命令字为双字长，各字段的含义如下：

- 命令码　规定了外围设备所执行的操作。通道命令码分三类：数据传输类(读、反读、写、取状态) ，通道转移类(转移) ，设备控制类(随设备类不同执行不同控制) 。
- 数据主存地址　对数据传输类命令，规定了本条通道命令访问的主存数据区起始(或末)地址，而“传送字节个数” 指出了数据区的大小。对通道转移类命令，用来规定转移地址。
- 标志码　用来定义通道程序的链接方式或标志通道命令的特点，32 位至 36 位依次为：数据链、命令链、禁发长度错、封锁读入主存、程序进程中断。32 和 33 位均为 0，称无链，表示本条通道命令是通道程序的最后一条；为 01 时，称命令链，表示本命令的操作已是最后一条，后面还有阄通道命令但为其他命令；32 位为 1 时，称数据链，表示下一条通道命令将延用本条的命令码但由下一条通道命令指明新的主存区域。34 位为 1 时，通该条通道命令执行中，禁止发长度错。35 位为 1 时，能使读型操作实现假读功能。36 位为 1 时，执行到该条通道命令将发出程序进程中断，将通道程序操作沿链推进的程度用中断方式通知操作系统。
- 传送字节个数　对数据传输类命令，规定了本次交换的字节个数；对通道转移类命令，规定填一个非 0 数。

5.3.1.2 通道程序

启动外围设备按指定要求工作，首先要编写出实现指定功能的通道程序。编制通道程序并不困难，关键在于：记住通道命令的格式，不同外围设备有不同的命令码，不能混用。下面是用汇编格式写的一个通道程序的例子。

```
CCW    X'02' , inarea ,      X'40' ,  80
CCW    X'02' ,    *   ,      X'40' ,  80
CCW    X'02' , inarea +80,   X'40' ,  80
CCW    X'02' ,    *   ,      X'40' ,  80
CCW    X'02' , inarea +160,  X'40' ,  80
                    .
                    .
inarea      DS      CL240
```

该通道程序把磁带上三个不连续的信息块读主存连续区域，其中，*表示不用主存地址，使用铐了“封锁读入主存” 标志位。

5.3.1.3 通道地址字和通道状态字

通道方式 I/O 时，要使用两个固定存储单元：通道地址字(Channel Address Word，CAW) 和通道状态字(Channel Status Word，CSW) 。

编好的通道程序放在主存中，为了使通道能取到通道命令去执行，在主存的一一个固定单元中存放当前启动的外围设备要求的通道程序的首地址，以后的命令地址可由前一个地址加 8 获得，这个用来存放通道程序的首地址的单元称通道地址字。

通道状态字是通道向操作系统报告情况的汇集。通道利用通道状态字可以提供通道和外围设备执行 I/O 操作的情况。IBM 系统中的通道状态字也采用双字表示。其中各字段的含义为：

- 通道命令地址：一般指向最后一条执行的通道命令地址加 8。
- 设备状态：是由控制器或设备产生、记录和供给的信息，包括：注意、状态修正位、控制器结束、忙、通道结束、设备结束、设备出错和设备特殊。
- 通道状态：由通道发现、记录和供给的信息，包括：程序进程中断、长度错误、程序出错、存储保护错、通道数据错、通道控制错、接口操作错和链溢出。
- 剩余字节个数：最后一条通道命令执行后还剩余多少字节未交换。

5.3.2 I/O 指令和主机 I/O 程序

IBM 系统主机提供一组 I/O 指令，以便完成 I/O 操作。I/O 指令有：启动 I/O(Start I/O，SIO) ，查询 I/O(Test I/O，TIO) ，查询通道(Test Channel，TCH) ，停止 aI/O(Halt I/O，HIO) 和停止设备(Halt Device，HDV) ,它们都是特权指令，以防用户擅自使用而引起 I/O 操作错误。例如，

SIO X’00E’

将启动 0 号通道，0E 号设备工作, 而根据系统的约定可把通道程序的首地址存放在主存中的通道地址字单元。CPU 执行 I/O 时只是简单地将 I/O 指令发给通道就行了，SIO 指令发出后，如果条件码为 0，表示设备已被成功启动，通道从 CAW 取通道程序首地址开始工作，CPU 可执行计算任务；如果条件码为 1，或表示启动成功（对于立即型命令），或启动不成功，通道有情况要报告，对此检查通道状态字 CSW。如果条件码为 2，表示通道或设备忙碌，启动不成功，本指令执行结束。如果条件码为 3，表示指定通道或设备断开，因而启动不成功，本指令执行结束。

每次执行 I/O 操作，要为通道编制通道程序，要为主机编制主机 I/O 程序。CPU 执行驱动外围设备指令时，同时，将首地址放在 CAW 中的通道程序交给通道，通道将根据 CPU 发来的 I/O 指令和通道程序对外围设备进行具体的控制。正确执行一次 I/O 操作的步骤可归纳如下：

- 确定 I/O 任务，了解使用何种设备，属于哪个通道操作方法如何等。
- 确定算法，决定例外情况处理方法。
- 编写通道程序，完成相应的 I/O 操作。
- 编写主机 I/O 程序，对不同条件码进行不同处理。

下面例子是用 IBM 汇编语言编写的采用双缓冲把磁带上的块记录在行式打印机上输出。

```
START
        BALR   11，0
        USING   *，11
        SSM = X’00’                 /*开中断
        LA   8，READ0
        ST   8，CAW
        SIO   X’0182’               /*启动磁带机反绕
        BC   7，*-4                 /*循环直到启动
        TIO   X’0182’
        BC   7，*-4                 /*测试直到磁带完成反绕
LOOP LA   8，READ1
        ST   8，CAW
        SIO   X’0182’               /*启动磁带读入缓冲 1
        BC   7，*-4
        TIO   X’0182’
        BC   7，*-4                 /*测试直到磁带完成
        LA   8，PRINT1
        ST   8，CAW
        TIO   X’00E’
        BC   7，*-4                 /*测试直到缓冲 2 打印完
        SIO   X’00E’                /*启动行印机印缓冲 1 的内容
        LA   8，READ2
        ST   8，CAW
```

```
    SIO   X'0182'          /*启动磁带读入缓冲 2
    BC   7，*-4
    TIO   X'0182'
    BC   7，*-4            /*测试直到磁带完成
    LA   8，PRINT2
    ST   8，CAW
    TIO   X'00E'           /*查询行印机
    BC   7，*-4            /*测试直到缓冲 1 打印完
    SIO   X'00E'           /*启动行印机印缓冲 1 的内容
    B   LOOP

READ0   CCW   X'07'，   *   ，    X'20'，1
READ1   CCW   X'02'，BUFFER1，  X'00'，512
READ2   CCW   X'02'，BUFFER2，  X'00'，512
```

5.3.3 通道启动和 I/O 操作过程

CPU 是主设备，通道是从设备，CPU 和设备之间是主从关系，需要相互配合协调才能完成 I/O 操作。那么 CPU 如何通知通道做什么？通道又如何鞍告知 CPU 其状态和工作情况呢？通道方式 I/O 过程可以分成三个阶段：

- I/O 启动阶段　用户在 I/O 主程序中调用文件操作请求传输信息，文件系统根据用户给予的参数可以确定哪台设备、传输信息的位置、传送个数和信息内存区的地址。然后，文件系统把存取要求通知设备管理，设备管理按规定组织好通道程序并将首地址放入 CAW。CPU 向通道发出 SIO，命令通道工作，通道根据自身状态形成条件码作为回答，若通道可用，则 CPU 传送本次设备地址，I/O 操作开始。这一通信过程发生在操作开始期，CPU 根据条件码便可决定转移方向。
- I/O 操作阶段　启动成功后，通道从主存固定单元取 CAW，根据该地址取得第一条通道命令，通道执行通道程序，同时将 I/O 地址传送给控制器，向它发出读、写或控制命令,控制外围设备进行数据传输。控制器接收通道发来的命令之后，检查设备状态，若设备不忙，则告知通道释放 CPU,并开始 I/O 操作，向设备发出一系列动作序列，设备则执行相应动作。之后，通道独立执行通道程序中各条 CCW，直到通道程序执行结束。从通道被启动成功开始，CPU 已被解放可执行其它任务并与通道并行工作,直到本次 I/O 结束，通道向 CPU 发出 I/O 操作结束中断,再次请求 CPU 干予。
- I/O 结束阶段　通道发现通道状态字中出现通道结束、控制器结束、设备结束或其它能产生中断的信号时，就应向 CPU 申请 I/O 中断。同时，把产生中断的通道号和设备号，以及 CSW 存入主存固定单元。中断装置响应中断后，CPU 上的现行程序才被暂停，调出 I/O 中断处理程序处理 I/O 中断。图 6-3 是通道方式 I/O 的示意。

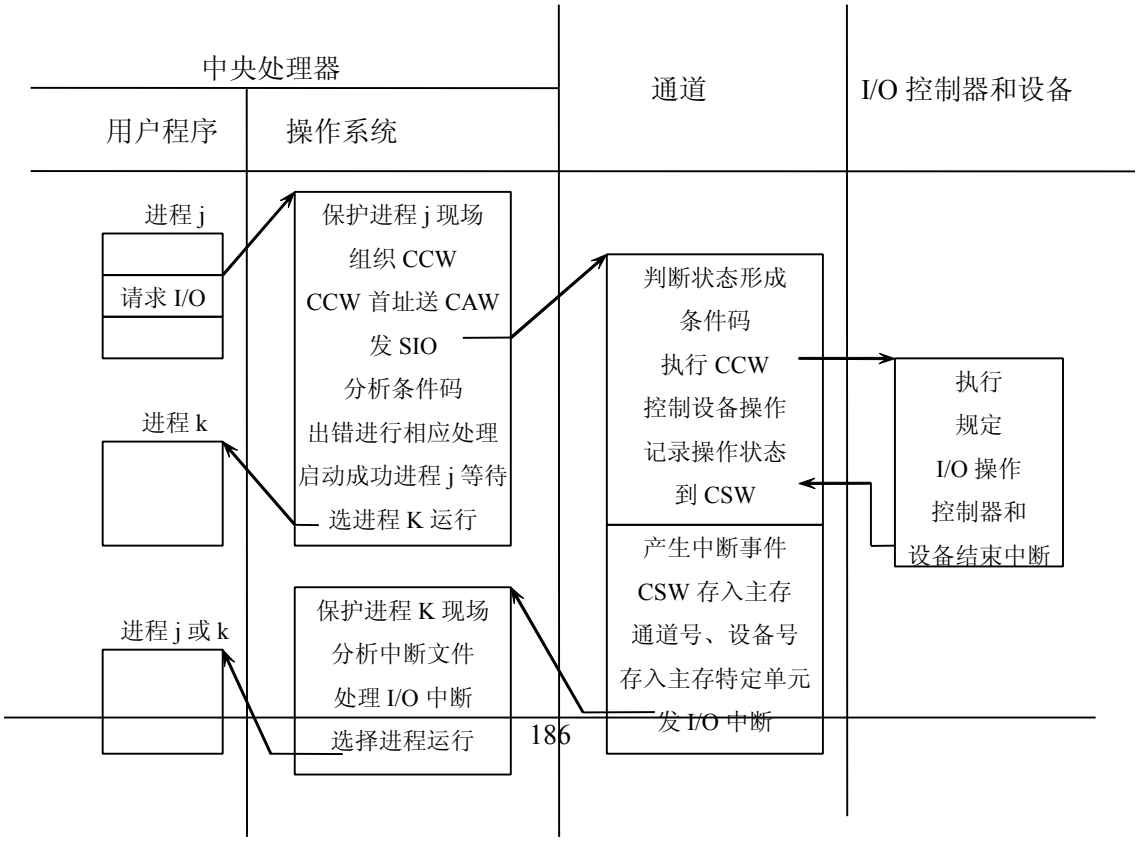


图 5-6　通道方式 I/O

5.4 缓冲技术

为了改善中央处理器与外围设备之间速度不配的矛盾，以及协调逻辑记录大小与物理记录大小不一致的问题，提高 CPU 和 I/O 设备的并行性,在操作系统中普遍采用了缓冲技术。

缓冲技术实现基本思想如下：当一个进程执行写操作输出数据时，先向系统申请一个主存区域——缓冲区，然后，将数据高速送到缓冲区。若为顺序写请求，则不断把数据填到缓冲区，直到它被装满为止。此后，进程可以继续它的计算，同时，系统将缓冲区内容写到 I/O 设备上。当一个进程执行操作输入数据时，先向系统申请一个主存区域——缓冲区，系统将一个物理记录的内容读到缓冲区域中，然后根据进程要求，把当前需要的逻辑记录从缓冲区中选出并传送给进程。

用于上述目的的专用主存区域称为 I/O 缓冲区，在输出数据时，只有在系统还来不及腾空缓冲而进程又要写数据时，它才需要等待；在输入数据时，仅当缓冲区空而进程又要从中读取数据时，它才被迫等待。其它时间可以进一步提高 CPU 和 I/O 设备的并行性，以及 I/O 设备和 I/O 设备之间的并行性，从而，提高整个系统的效率。

在操作系统管理下，常常辟出许多专用主存区域的缓冲区用来服务于各种设备，支持 I/O 管理功能。常用的缓冲技术有：单缓冲、双缓冲、多缓冲。

5.4.1 单缓冲

单缓冲是操作系统提供的一种简单的缓冲技术。每当一个用户进程发出一个 I/O 请求时,操作系统在主存的系统区中开设一个缓冲区。

对于块设备输入,单缓冲机制如下工作:先从磁盘把一块数据传送到缓冲区,假如所花费的时间为 T;接着操作系统把缓冲区数据送到用户区,设所化时间为 M,由于这时缓冲区已空,操作系统可预读紧接的下一块,大多数应用将要使用邻接块,;然后用户进程对这批数据进行计算,共耗时 C;那么,不采用缓冲,数据直接从磁盘到用户区,每批数据处理时间约为 T+C,而采用单缓冲，每批数据处理时间约为 max[C,T]+M,通常 M 远小于 C 或 T,故速度快了很多。对于块设备输出,单缓冲机制工作方式类似，先把数据从用户区拷贝到系统缓冲区，用户进程可以继续请求输出，直到缓冲区填满后，才启动 I/O 写到磁盘上。

对于字符设备输入,缓冲区用于暂存用户输入的一行数据，在输入期间,用户进程被挂起等待一行数据输入完毕;在输出时，用户进程将第一行数据送入缓冲区后，继续执行。如果在第一个输出操作没有腾 空缓冲区之前，又有第二行数据要输出,用户进程应等待。

5.4.2 双缓冲

为了加快 I/O 速度和提高设备利用率，需要引入双缓冲工作方式,又称缓冲交换(buffer swapping)。在输入数据时，首先填满缓冲区 1,操作系统可从缓冲区 1 把数据送到用户进程区,用户进程便可对数据进行加工计算;与此同时，输入设备填充缓冲区 2。当缓冲区 1 空出后，输入设备再次向缓冲区 1 输入。操作系统又可以把缓冲区 2 的数据传送到用户进程区,用户进程开始加工缓冲 2 的数据。两个缓冲区交替使用，使 CPU 和 I/O 设备、设备和设备的并行性进一步提高， 仅当两个缓冲区都取空，进程还要提取数据时，它再被迫等待。我们粗略估计一下,对于块设备,处理或传输一块的时间为 max(C,T),如果 C<T,可以保证块设备连续工作;如果 C>T,使得进程不必要等待 I/O。双缓冲使`效率提高了,但复杂性也增加了。

5.4.3 多缓冲

采用双缓冲技术虽然提高了 I/O 设备的并行工作程度，减少了进程调度开销，但在输入设备、输出设备和处理进程速度不匹配的情况下仍不十分理想。举例来说，若输入设备的速度高于进程消耗这些数据的速度，则输入设备很快就把两个缓冲区填满；有时由于进程处理输入数据速度高于输入的速度，很快又把两个缓冲区抽空，造成进程等待。为改善上述情形,获得较高的并行度，常常采用多缓冲组成的循环缓冲(circular buffer)技术。

操作系统从自由主存区域中分配一组缓冲区组成循环缓冲，每个缓冲区的大小可以等于物理记录的大小。多缓冲的缓冲区是系统的公共资源，可供各个进程共享，并由系统统一分配和管理。缓冲区可用途分为：输入缓冲区，处理缓冲区和输出缓冲区。为了管理各类缓冲区，进行各种操作，必须设计专门的软件，这就是缓冲区自动管理系统。

在 Unix 系统中，不论是块设备管理，还是字符设备管理，都采用循环缓冲技术，其目的有两个：一是尽力提高 CPU 和 I/O 设备的并行工作程度；二是力争提高文件系统信息读写的速度和效率。Unix 的块设备共设立了 15 个 512 字节的缓冲区；字符设备共设立了 100 个 8 字节的缓冲区。每类设备设计了相应

的数据结构以及缓冲区自动管理软件,采用了完善的缓冲技术,引入了"预先读"、"异步写"、"延迟写"方式,提高 CPU 与设备 I/O 的并行性。

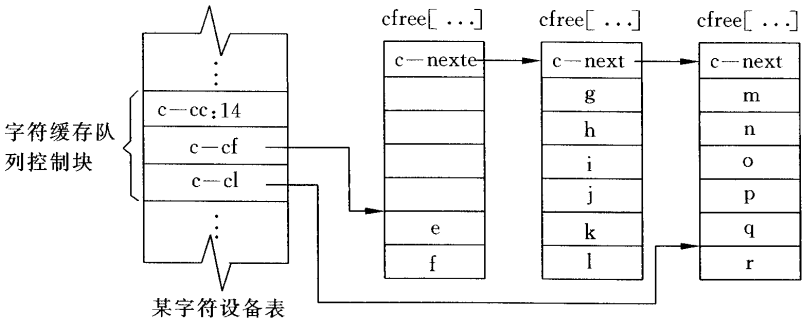


图 5-7　Unix I/O 字符缓存队列

图 5-7 是 Unix 字符设备的 I/O 字符缓存队列，每个字符设备都有一张设备表，它包含有输入/输出字符缓存队列的控制块，其中包括三项内容: c-cc 队列中可用字符计数；c-cf 指向队列中的第一个字符;c-cl 指向队列的最后一个字符。现在来看一下取字符和释放字符缓存的问题以及送字符和申请字符缓存的过程。根据 c-cf 指点，逐一从字符缓存里取出字符 e 和 f，c-cc 做计数调整，c-cf 也依次下移。若发现 c-cf 的最低三位已为 0 时，这说明该字符缓存中的字符已全部取完，c-cf 应指向下一字符缓存的第一个字符，即要把释放的这一个字符缓存中的 c-next 加工送给 c-cf。反之,根据 c-cl 指点往字符缓存里送字符,显然，当 c-cl 获取下一存放字符的位置时，发现地址的最后三位为 0 时，表示目前的字符缓存已经存满，需要先申请一个新的字符缓存才能把字符存入。于是向系统申请一个新的字符缓存，链入到 I/O 字符缓存队列之尾，然后再根据调整后的 c-cl 送入字符。

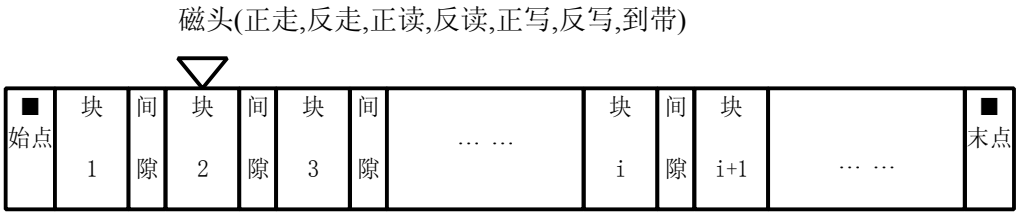
5.5 驱动调度技术

作为操作系统的辅助存储器，用来存放文件的磁盘一类高速大容量旋转型存储设备，在繁重的输入输出负载之下，同时会有若干个输入输出请求来到并等待处理。系统必须采用一种调度策略，使能按最佳次序执行要求访问的诸请求，这就叫驱动调度，使用的算法叫驱动调度算法。驱动调度能减少为若干个输入输出请求服务所需的总时间，从而提高系统效率、除了输入输出请求的优化排序外，信息在辅助存储器上的排列方式，存储空间的分配方法都能影响存取访问速度。

不同的外存储设备优质不同的信息安排方式，本节首先介绍顺序存取存储设备和直接存取存储设备的结构，然后针对磁盘讨论与驱动调度有关的技术。

5.5.1 存储设备的物理结构

顺序存取存储设备是严格依赖信息的物理位置进行定位和读写的存储设备，所以，从存取一个信息



块到存取另一信息块要花费较多的时间。磁带机是最常用的一种顺序存取存储设备，由于它具有存储容量大、稳定可靠、卷可装卸和便于保存等优点，已被广泛用作存档的文件存储设备。磁带的存储如图 5-8 所示。

图 5-8　磁带的存储示意图

磁带的一个突出优点是物理块长的变化范围较大，块可以很小，也可以很大，原则上没有限制。为了保证可靠性，块长取适中较好，过小时不易区别干扰还是记录信息，过大对产生的误码就难以发现和校正。

磁带上的物理块没有确定的物理地址，而是由它在带上的物理位置来标识。例如磁头在磁带的始端，为了读出第 100 块上的记录信息，必须正向引带走过前面 99 块。对于磁带机，除了读/写一块物理记录

的通道命令外，通常还有辅助命令，如反读、前跳、后退和标识，有一个称作带标的特殊记录块，只有使用写带标命令才能刻写。在执行读出、前跳和后退时，如果磁头遇到带标，硬件能产生设备特殊中断，通知操作系统进行相应处理。

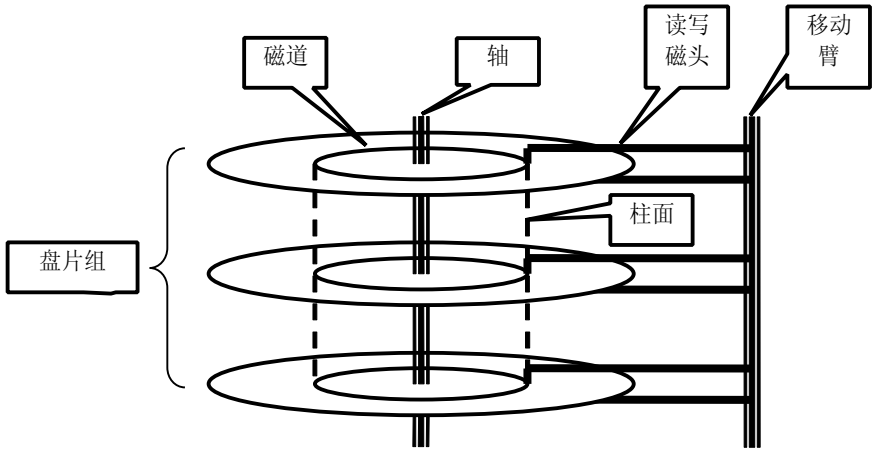


图 5-9　磁带的存储示意图

磁盘是一种直接存取存储设备，又叫随机存取存储设备。它的每个物理记录有确定的位置和唯一的地址，存取任何一个物理块所需的时间几乎不依赖于此信息的位置。磁盘的结构如图 5-9 所示，它包括多个盘面用于存储数据。每个盘面有一个读写磁头，所有的读写磁头都有固定在唯一的移动臂上同时移动。在一个盘面上的读写磁头的轨迹磁道，在磁头位置下的所有磁道组成地圆柱体称柱面，一个磁道又可被划分成一个或多个物理块。

微型计算机使用 5.25 英寸软盘存储信息。双面低密度软盘每面建立 40 个磁道，每磁道划分为 9 个扇区，每个扇区存储 512 个字节，其存储容量为：40 磁道×9 扇区×2 面×512 字节＝360K 字节。双面高密度软磁盘的存储容量为：80 磁道×15 扇区×2 面×512 字节＝1.2M 字节。

文件的信息通常不是记录在同一盘面的各个磁道上，而是记录在同一柱面的不同磁道上，这样可使移动臂的移动次数减少，从而缩短存取信息的时间。为了访问磁盘上的一个物理记录，必须给出三个参数：柱面号、磁头号、块号。磁盘机根据柱面号控制臂作机械的横向移动，带动读写磁头到达指定柱面，这个动作较慢，一般称作‘查找时间’，平均需 20 毫秒左右。下一步从磁头号可以确定数据所在的盘，然后等待被访问的信息块旋转到读写头下时，按块号进行存取，这段等待时间称为“搜索延迟”，平均要 10 毫秒。磁盘机实现些操作 的通道命令是：查找、搜索、转移和读写。

5.5.2 循环排序

旋转型存储设备的不同记录的存取时间有关明显的差别，所以输入输出请求的某种排序有实际意义。考虑每一磁道保存 4 个记录的旋转型设备，假定收到以下四个输入输出请求并且存在一条到该设备的可用道路。

请示次序	记录号
（1）	读记录 4
（2）	读记录 3
（3）	读记录 2
（4）	读记录 1

对这些输入输出请求有多种排序方法：

- 方法 1：如果调度算法按照输入输出请求次序读记录 4、3、2、1，假定平均要用 1/2 的周来定位，再加上 1/4 周读出记录，则总的处理时间等于 3 周，即 60 毫秒。
- 方法 2：如果调度算法决定的读入次序为读记录 1、2、3、4。那么，总的处理时间等于 1.5 周，即 30 毫秒。
- 方法 3：如果我们知道当前读位置是记录 3，则调度算法采用的次序为读记录 4、1、2、3 会更好。总的处理时间等于 1 周，即 20 毫秒。

为了实现方法 3，驱动调度算法必须知道旋转型设备的当前位置，这种硬设备叫做旋转位置测定。如果没有这种硬件装置，那么因无法测定当前记录而可能会平均多花费半圈左右的时间。

循环排序时，还必须考虑某些输入输出的互斥问题。例如读写磁鼓的记录信息需要两个参数：道号和记录号。如果请求是：

请求次序	磁道号	记录号

（1）　　道 1　记录 2
（2）　　道 1　纪录 3
（3）　　道 1　记录 1
（4）　　道 6　记录 3
（5）　　道 4　记录 2

那么请求 1 和 5、请求 2 和 4，都互相排斥，因为它们涉及的记录号相同。这样在第一转为请求 1 服务，第二转才能为请求 5 服务，或者反之。对于相同记录号的所有输入输出请求会产生竞争，如果硬件允许一次从多个磁道上读写，就可减少这种拥挤现象，但是这通常需要附加的控制器，设备中还要增加电子部件。

5.5.3 优化分布

信息在存储空间的排列方式也会影响存取等待时间。考虑 10 个逻辑记录 A，B……，J 被存于旋转型设备上，每道存放 10 个记录，可安排如下：

物理块	逻辑纪录
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	I
10	J

假定要经常顺序处理这些记录，而旋转速度为 20 毫秒，处理程度读出每个记录后花 4 毫秒进行处理。则读出并处理记录 A 之后将转到记录 D 的开始。所以，为了读出 B，必须再转一周。于是，处理 10 个记录的总时间为：10 毫秒(移动到记录 A 的平均时间)+ 2 毫秒(读记录 A)+4 毫秒(处理记录 A)+9×[16 毫秒(访问下一记录) +2 毫秒(读记录)+4 毫秒(处理记录)]＝214 毫秒

按照下面方式对信息分布优化：

物理块	逻辑纪录
1	A
2	H
3	E
4	B
5	I
6	F
7	C
8	J
9	G
10	D

当读出记录 A 并处理结器束后，恰巧转至记录 B 的位置，立即就可读出并处理。按照这一方案，处理 10 个记录的总时间为：10 毫秒(移动到记录 A 的平均时间)+10×[2 毫秒（读记录）×4 毫秒（处理记录）]=70 毫秒

比原方案速度几乎快 3 倍，如果有众多记录需要处理，节省时间更可观了。

5.5.4 交替地址

旋转型存储设备上的任意记录的存取时间主要由旋转速度来确定，对于给定地装置这一速度是常数。把每个记录重复记录在这台设备的多个区域，可以显著地减少存取时间。这样读相同的数据，不有几个交替地址，这种方法也称为多重副本或折迭。

让我们考虑一种设备。若每道有 8 个记录，则旋转速度 20 毫秒，如果记录 A 存于道 1，记录 1，存取记录 A 平均占半周，即 10 毫秒。如果记录 A 的副本存于道 1，记录 1 和道 1，记录 5，那么，使用旋转位置测定，总是存取“最近”的副本，有效的平均存取时间可降为 5 毫秒。类似地，存储更多相同数据记录的副本，可以把存取时间进一步折半。这一技术的主要缺点是耗用较多存储空间，有效容量随副本个数增加而减少，如果每个记录有 n 个副本，存储空间就被“折迭”了 n 次。

此法成功与否取决于下列因素：数据记录总是读出使用，不需修改写入；数据记录占用的存储空间总量不太大；数据使用极为频繁。所以，通常对系统程序采用了这一技巧，它们能满足上述诸因素。

5.5.5 搜查定位

对于移动臂磁盘设备，除了旋转位置外，还有搜查定位的问题。输入输出请求需要三部分地址：柱面号、道号和记录号。例如对磁盘同时有以下 5 个访问请求。

柱面号	磁道号	记录号
7	4	1
7	4	8
7	4	5
40	6	4
2	7	7

如果当前移动臂处于 0 号柱面，若按上述次序访问磁盘，移动臂将从 0 号柱面移至 7 号柱面，再移至 40 号柱面，然后回到 2 号柱面，显然，这样移臂很不合理。如果将访问请求按照柱面号 2，7，7，7，40 的次序处理，这可将节省移臂时间。进一步考查 7 号柱面的三个访问，按上述次序，那么，必须使磁盘旋转近 2 圈才能访问完毕。若再次将访问请求排序，按照：

柱面号	磁道号	记录号
7	4	1
7	4	5
7	4	8

执行，显然，对 7 号柱面的三次访问大约只要旋转 1 圈或更少就能访问完毕。由此可见，对于磁盘一类设备，在启动之前按驱动调度策略对访问的请求优化排序是十分必要的。除了应有使旋转圈数最少的调度策略外，还应考虑使移臂时间最短的调度策略。

移臂调度有若干策略，“电梯调度”算法是简单而实用的一种算法。按照这种策略每次总是选择沿臂的移动方向最近的那个柱面；如果沿这个方向没有访问的请求时，就改变臂的移动方向，使用移动频率极小化。每当要求访问磁盘时，操作系统查看磁盘机是否空闲。如果空闲就立即移臂，然后将当前移动方向和本次停留的位置都登记下来。如果不空，就让请求者等待并把它要求访问的位置登记下来，按照既定的调度算法对全体等待者进行寻查定序，下次按照优化的次序执行。如果有多个盘驱动器的请求同时到达时，系统还必须有优先启动哪一个盘组的 I/O 请求决策。

对于移动臂磁盘还有许多其它驱动调度算法，其中最简单的一种是“先来先服务”算法，在这种调度策略下，磁盘臂的移动完全是随机的，不考虑各个 I/O 请求之间的相次对序和移动臂当前所处位置，进程等待 I/O 请求的时间会过长，寻道性能较差。对“先来先服务”算法的改进有以下几种方法：

1）“最短查找时间优先”算法。本算法考虑了各个请求之间的区别，总是先执行查找时间最短的那个请求,从而,较”先来先服务”算法有较好的寻道性能。

2）“扫描”算法。磁盘臂移动一次沿一个方向移动，扫过所有的柱面，迁到最近的 I/O 请求使进行处理，然后再向相反方向移动回来。与“电梯调度”算法的不同在于:即使该移动方向暂时没有了 I/O 请求,移动臂也扫描到头。“最短查找时间优先”算法虽有较好的寻道性能,但可能会造成进程”饥饿”状态，只要不断有新的 I/O 请求到达,就有可能无限期推迟某些很早就到达的 I/O 请求，而本算法克服了这一缺点。

3）“分步扫描”算法。将 I/O 请求分成组，每组不超过 N 个请求，每次选一个组进行扫描，处理完一组后再选下一组。这种调度算法能保证每个存取请求的等待时间不至太长。当 N 很大时,接近于”扫描”算法的性能,当 N=1 时,接近于“先来先服务”算法的性能。

4）“单向扫描”算法。这是为适应不断有大量存取请求进入系统的情况而设计的一种扫描方式。移动臂总是从 0 号柱面至最大号柱面顺序扫描，然后直接返回 0 号柱面重复进行。在一柱面上，移动臂停留至磁盘旋转过一定圈数，然后再移向下一个柱面。这样能够缩短刚离开的柱面上又到达的大量 I/O 请

求的等待时间，为了在磁盘转动每一圈的时间内执行更多的存取，必须考虑旋转优化问题。

第 1 和第 2 两种算法，在单位时间内处理的输入输出请求较多即吞吐量较大，但是请求的等待时间较长，第 1 种算法使等待时间更长一些。一般说来“扫描”算法较好，但它不分具体情况而扫过所有柱面造成性能不够好。“分步扫描”算法使得各个输入输出请求等待时间之间的差距最小，而吞吐量适中。“单向扫描”仅适应有不断大批量输入输出存取请求,且磁道上存放记录数量较大的情况。

上面讨论的驱动调度算法能减少输入输出请求时间，但都是以增加处理器时间为代价的。排队技术并不是在所有场合都适用的。这些算法的价值依赖于处理器的速度和输入输出请求的数量。如果输入输出请求较少，采用多道程序设计后就可以达到较高的吞吐量。如果处理器速度很慢，处理器的开销可能掩盖这些调度算法带来的好处。

5.5.6 独立磁盘冗余阵列

独立磁盘冗余阵列 RAID(Reundant Array of Independent Disks)概念早在 1987 年由美国加里福尼亚大学 berkeley 分校一个研究小组的论文中提出,现已得到工业界的认可,作为一种多磁盘数据库设计的一种标准样式(scheme),已被广泛地应用于大中型计算机和计算机网络系统。它是利用一台磁盘阵列控制器统一管理和控制一组磁盘驱动器,组成一个速度块、可靠性高、性能价格比好的大容量磁盘系统。RAID 的提出填补了 CPU 速度快与磁盘设备速度慢之间的间隙,其策略是:用一组较小容量的、独立的、可并行工作的磁盘驱动器组成阵列来代替单一的大容量磁盘,再加进冗余技术,数据能用多种方式组织和分布存储,于是,独立的 I/O 请求能被并行处理,数据分布的单个 I/O 请求也能并行地从多个磁盘驱动器同时存取数据,从而,改进了 I/O 性能和系统可靠性。

RAID 样式共有 6 级组成，RAID0 至 RAID5，最新又扩充了 RAID6 和 RAID7，它们之间并不隐含层次关系，而是标明了不同的设计结构，并有三个共同特性:①RAID 是一组物理磁盘驱动器，可以被操作系统看作是单一的逻辑磁盘驱动器;②数据被分布存储在阵列横跨的物理驱动器上;③冗余磁盘的作用是保存奇偶校验信息，当磁盘出现失误时它能确保数据的恢复。虽然让多个磁头和驱动机构同时操作能达到较高 I/O 数据传输速率，但使用多台硬设备也增加了故障的概率，为了补偿可能造成的可靠性下降，RAID 利用存储的奇偶校验信息来恢复由于磁盘故障丢失的数据。RAID0 不支持第三个特性，采用把大块数据分割成数据子块(strip)，交替间隔地分布存储，但未引入冗余磁盘，适用于性能要求高，但非要害数据这类应用。DAID1 采用镜像技术，适用于做系统驱动器，存放关键的系统文件。RAID2 和 RAID3 主要采用了并行存取技术，分别还引入海明(Hamming code)校验码和位扦入(bit interleaved)奇偶校验码改进可靠性，适用于大数据量 I/O 请求，如图像和 CAD 这类的应用。RAID4 和 RAID5 主要采用了独立存取技术， 分别还引入块扦入(block interleaved) 奇偶校验码和块扦入分布式 (block interleaved distributed) 奇偶校验码改进可靠性，适用于 I/O 请求频繁的事务处理。

5.5.6.1 RAID level 0

用户和系统的数据划成子块被分布存储在横跨阵列中的所有磁盘上，逻辑上连续的数据子块，在物理上可被依次存储在横向相邻的磁盘驱动器上，通过一个阵列管理软件进行逻辑地址空间到物理地址空间的映射。与单个大磁盘相比有着明显优点，它能并行地处理要求位于不同磁盘上数据的不同 I/O 请求。RAID level0 并不能真正划入 RAID 家族，因为它没有引入冗余校验来改进性能，导致磁盘系统的可靠性差，容易丢失数据，故已较少使用。

5.5.6.2 RAID level 1

RAID1 与 RAID2 至 RAID5 的主要差别在于实现冗余校验的方法不同。RAID1 简单地采用双份所有数据的办法，类似于 RAID0， 数据划成子块被分布存储在横跨阵列中的所有磁盘上，但是，每个数据子块被存储到两个独立的物理磁盘上，故阵列中的每个盘都有一个包含相同数据的它的镜像盘。这种数据组织方法的主要可取之处在于：①读请求能通过包含相同请求数据中的任何一个磁盘来提供服务，其中的一个所化查找和搜索时间最少;②写操作时，要求改写对应的两个数据子块，但这一点可采用并行操作实现，写操作的性能由并行操作中较慢的一个决定;③出现故障后的恢复很简单，当一个驱动器出现故障，数据可以从镜像盘获得。RAID1 的主要缺点是价格太贵，空间利用率仅有一半，往往作为存放系统软件、关键数据和要害文件的驱动器;但是当磁盘故障时，它提供了一个实时的数据备份，所有的数据信息立即可用。

5.5.6.3 RAID level 2

RAID2 和 RAID3 采用了并行存取技术。在并行存取阵列中，所有的磁盘参与每个 I/O 请求的执行，且每个驱动器的移动臂同步工作，使得任何时刻每个磁盘的磁头都在相同的位置。在其它 RAID 样式中，使用了数据子块，RAID2 和 RAID3 的数据子块非常小，常常小到单个字节或一个字。对于 RAID2 纠错码按照横跨的每个数据盘的相应位进行计算，并存储在多只奇偶校验盘的相应位的位置。典型地使用海明校验码，它能纠正单位错，发现双位错。

虽然 RAID2 所需磁盘数少于 RAID1，但价格仍然很贵。所需奇偶校验磁盘的数量与数据盘的多少成比例。在执行单个读操作时，所有的磁盘要被同时存取，请求的数据和关联的纠错码都提交给阵列控制器，如果发现有一个二进位错， 阵列控制器能立即确认和纠正错误，所以读操作时间并未变慢;在执行

单个写请求时，所有的数据盘和奇偶校验盘必须被存取。

5.5.6.4 RAID level 3

RAID3 的组织与 RAID2 相似，其差别是它仅使用一只冗余盘，而不论是多大的磁盘阵列。RAID3 也使用并行存取技术，数据分割成较小子块分布式存储，它用简单的奇偶校验代替上述复杂的海明校验，仍然按所有数据盘上相同位置的每个二进位的集合进行计算，而奇偶校验磁盘只要一个了。

当出现磁盘故障时，要使用奇偶校验盘的信息，数据可以用剩下的磁盘中的信息来重新构造。数据的构造十分简单，考虑有五个磁盘驱动器组成的阵列，若 X0 到 X3 存放数据，X4 为奇偶校验盘，对于第 i 位的奇偶校验位可如下计算:

$$X4(i)=X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

假定驱动器 X1 出故障，如果把 X4(i) X1(i)加到上面等式的两边，得到

$$X1(i)=X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

因此，在阵列中的任何一只数据盘上的任一数据子块的内容，都能从阵列中剩余下来的其它磁盘的对应数据子块的内容来重新生成，这个原理适用于 RAID3、RAID4 和 RAID5。阵列中有一只磁盘发生故障时，少了一只磁盘后的其它所有数据依然可用，对于读操作来说，丢失的数据通过求异或或计算被重新生成。当数据被写到少了一只磁盘的 RAID 阵列时，必须维护奇偶校验的一致性。返回正常操作时需替换发生故障的磁盘，且故障盘的内容应当被重新生成到新的磁盘上。

5.5.6.5 RAID level 4

RAID4 和 RAID5 使用了独立存取技术，在一个独立存取的磁盘阵列中，每个驱动器都可以独立地工作，所以，独立的 I/O 请求可以被并行地得到满足。因此独立存取阵列适合于有频繁 I/O 请求的应用。在 RAID4 和 RAID5 中，数据子块划得较大，逐位(bit-by-bit)二进位奇偶校验数据块按横跨每个数据盘上对应的数据子块来计算，奇偶校验位存储在奇偶磁盘上的对应数据子块中。

每当执行一个小数据量写操作时，阵列管理软件不旦要修改用户数据，而且也要修改对应的奇偶校验位。考虑有五个驱动器组成的一个阵列，X0 到 X3 存储数据，X4 是奇偶校验磁盘。假如执行一个仅仅涉及磁盘 X1 上数据的写操作。开始的时候，对每一个二进位 i，有下列关系式:

$$X4(i)=X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

在数据修改之后，对于改变了的二进位用’符号来指出，得到:

$$X4'(i)=X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i)$$
$$=X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i)$$
$$=X4(i) \oplus X1(i) \oplus X1'(i)$$

为了计算新的奇偶校验位，陈列管理软件必须读出老的用户数据子块和老的奇偶校验码，因此每个数据写操作包含了两次读操作和两次写操作。

执行一个大数据量写操作时，它涉及数据子块位于所有数据盘上，通过计算新数据的二进位便可很容易地计算出奇偶校验码，因此奇偶磁盘与数据磁盘能被并行地修改，也就是说，没有额外的读写操作。由于任何写操作均涉及到奇偶校验盘，因此它就成了一个瓶颈口。

5.5.6.6 RAID level 5

RAID5 的组织形式与 RAID4 类似，差别仅奇偶校验码是分布横跨存放在所有的磁盘上，典型的存放方法为轮转法，设有 n 个磁盘的一个阵列，则开头的 n 个奇偶校验码螺旋式地位于 n 个磁盘上(即每个盘上有一个奇偶校验码)，接着按这个模式再次重复。采用螺旋式把奇偶校验码分布横跨存放在所有的磁盘上，能避免 RAID4 中发生的奇偶校验盘湖瓶颈口问题。

5.5.6.7 RAID level 6和RAID level 7

这是增强型 RAID。RAID6 中设置了专用快速的异步校验磁盘，具有独立的数据访问通路，比低级 RAID 性能更好，但价格昂贵。RAID7 对 RAID6 作了改进，该阵列中的所有磁盘都有较高传输速率，性能优异，但价格也很高。

5.5.7 提高磁盘 I/O 速度的一些方法

通常为磁盘设置高速缓存，这样能显著减少等待磁盘 I/O 的时间。下面介绍的一些方法也能有效提高磁盘 I/O 的速度。

提前读。用户经常采用顺序方式访问文件的各个盘块上的数据，在读当前盘块时已能知道下次要读出的盘块的地址 ，因此，可在读当前盘块的同时，提前把下一个盘块数据也读入磁盘缓冲区。这样一来，当下次要读盘块中的那些数据时，由于已经提前把它们读入了缓冲区，便可直接使用数据，而不必再启动磁盘 I/O，从而，减少了读数据的时间，也就相当于提高了磁盘 I/O 速度。”提前读”功能已被许多操作系统如 Unix、Windows 等广泛采用。

延迟写。在执行写操作时，磁盘缓冲区中的数据本来应该立即写回磁盘，但考虑到该缓冲区中的数据不久之后再次被输出进程或其它进程访问，因此，并不马上把缓冲区中数据写盘，而是把它挂在空闲缓冲区队列的末尾。随着空闲缓冲区的使用，存有输出数据的缓冲区也不停地向队列头移动，直至移动到空闲缓冲区队列之首。当再有进程申请缓冲区，且分到了该缓冲区时，才把其中的数据写到磁盘上，

于是这个缓冲区可作为空闲缓冲区分配了。只要存有输出数据的缓冲区还在队列中，任何访问该数据的进程，可直接从中读出数据，不必再去访问磁盘。这样做，可以减少磁盘的 I/O 时间，相当于提高了 I/O 速度。同样，在 Unix 和 Windows 中也采用了这一技术。

5.6 设备分配

5.6.1 设备独立性

现代计算机系统常常配置许多类型的外围设备，同类设备又有多台，尤其是多台磁盘机，磁带机的情况很普遍。作业在执行前，应对静态分配的外围设备提出申请要求，如果申请时指定某一台具体的物理设备，那么分配工作就很简单，但当指定的某台设备有故障时，就不能满足申请，该作业也就不能投入运行。例如系统拥有 A、B 两台卡片输入机，现有作业 J2 申请一台卡片输入机，如果它指定使用 A，那么作业 J1 已经占用 A 或者设备 A 坏了，虽然系统还有同类设备 B 是好的且未被占用，但也不能接受作业 J2，显然这样做很不合理。为了解决这一问题，通常用户不指定特定的设备，而指定逻辑设备，使得用户作业和物理设备独立开来，再通过其它途径建立逻辑设备和物理设备之间的对应关系，我们称这种特性为“设备独立性”。具有设备独立性的系统中，用户编写程序时使用的设备与实际使用的设备无关，亦即逻辑设备名是用户命名的，可以更改是系统规定的，是不可更改的。设备管理的功能之一就是把逻辑设备名转换成物理设备名。

设备独立性带来的好处是：用户与物理的外围设备无关，系统增减或变更外围设备时程序不必修改；易于对付输入输出设备的故障，例如，某台行式打印机发生故障时，可用另一台替换，甚至可用磁带机或磁盘机等不同类型的设备代替，从而提高了系统的可靠性，增加了外围设备分配的灵活性，能更有效地利用外围设备资源，实现多道程序设计技术。

操作系统提供了设备独立特性后，程序员可利用逻辑设备进行行输入输出，而逻辑设备与物理设备之间的转换通常由操作系统的命令或语言来实现。由于操作系统大小和功能不同，具体实现逻辑设备到物理设备的转换就有差别，一般使用以下方法：利用作业控制语言实现批处理系统的设备转换；利用操作命令实现设备转换；利用高级语言的语句实现设备转换。

5.6.2 设备分配

现代计算机系统可以同时承担若干个用户的多个计算任务，设备管理的一个功能就是为计算机系统接纳的每个计算任务分配所需要的外围设备。从设备的特性来看，可以把设备分成独占设备、共享设备和虚拟设备三类，相应的管理和分配外围设备的技术可分成：独占方式、共享方式和虚拟方式，本节讨论前两种技术。

有些外围设备，如卡片输入机、卡片穿孔机、行式打印机、磁带机等，往往只能让一个作业独占使用，这是由这类设备的物理特性决定的。例如，用户在一台分配给他的卡片上输入机上装上一叠卡片，卡片上存放着该用户作业要处理的数据，由于作业执行中将随机地读入卡片上的数据进行加工处理，因此，不可能在该作业暂时不使用卡片输入机时，人为地换上另一作业的一叠卡片，让卡片输入机为另一作业服务。只有当某作业归还卡片输入机后，才能让另一作业去占用。

另一类设备，如磁盘、磁鼓等，往往可让多个作业共同使用，或者说，是多个作业可共享的设备。这是因为这一类设备容量大、存取速度愉且可直接存取。例如，可把每个作业的信息组织成文件存放在磁盘上，使用信息时也按名查询文件，从磁盘上读出。用户提出存取文件要求时，总是先由文件管理进行处理，确定信息存放位置，然后再后设备管理提出驱动要求。所以，对于这一类设备，设备管理的主要工作是驱动工作，这包括驱动调度和实施驱动。

对独占使用的设备，往往采用静态分配方式，即在作业执行前，将作业所要用的这一类设备分配给它。当作业执行中不再需要使用这类设备，或作业结束撤离时，收回分配给它的这类设备。静态分配方式实现简单，能防止系统死锁，但采用这种分配方式，会降低设备的利用率。例如，对行式打印机，若采用静态分配，则在作业执行前把行式打印机分配给它，但一直到作业产生结果时才使用分配给它的行式打印机。这样，尽管这台行式打印机在大部分时间里处于空闲状态，但是，其它作业却不能使用它。

如果对行式打印机采用动态分配方式，即在作业执行过程中，要求建立一个行式打印机文件输出一批信息量，系统才把一台行式打印机分配给该作业，当一个文件输出完毕关闭时，系统就收回分配给该作业的行式打印机。采用动态分配方式后，在行式打印机上可能依次输了若十个作业的信息，由于输出信息以文件为单位，每个文件的头和尾均设有标志，如：用户名、作业名、文件名等，操作员很容易辩论输出信息是属于哪个用户。所以，对某些独占使用的设备，采用动态分配方式，不仅是可行能提高设备的利用率。

对于磁盘、磁鼓等可共享的设备，一般不必进行分配。但有些系统也采用静态分配方式把各柱面鼓分配给不同的作业使用，这可提高存取速度，但使存储空间利用降低，用户动态扩充困难。

操作系统中，对 I/O 设备的分配算法常用的有：先请求先服务，优先级高者先服务等。此外，在多

进程请求 I/O 设备分配时，应防止因循环等待对方所占用的设备而产生死锁，应预先进行性检查。

为了实现 I/O 设备的分配，系统中应设有设备分配的数据结构：设备类表和设备表。系统中拥有一张设备类表，每类设备对应于设备表中的一栏，其包括的内容通常有：设备类、总台 、空闲台数和设备表起始地址等。每一类设备，如输入机、行式打印机等都有各自的设备表，该表用来登记这类设备中每一台设备的状态，其包含的内容通常有：物理设备名、逻辑设备名、占有设备的进程号、已分配/未分配、好、坏等。按照上述分配使用的数据结构，不难设计出 I/O 设备的分配流程。

5.7 虚拟设备

5.7.1 问题的提出

对于卡片输入输出机、行式打印机之类的设备采用静态分配方式是不利于提高系统效率的。首先，占有这些设备的作业不能有效地充分利用它们。一台设备在作业执行期间，往往只有一部分，甚至很少一部分时间在工作，其余时间均处于空闲状态。其次，这些设备分配给一个作业时后，再有申请这类设备的作业将被拒绝接受。例如，一个系统拥有两台卡片输入机，它就难于接受 4 个要求使用卡片输入机的作业同时执行，而占用卡片输入机的作业却又在占用的大部分时间里让它闲着。另外，这类设备传输而大大延长了作业的执行时间。为此，现代操作系统都提供虚拟设备的功 能来解决这些问题。

早期，采用脱机外围设备操作，使用一台外围计算机，它的功能是以最大速度从读卡机上读取信息并记录到输入磁盘上。然后，把包含有输入信息的输入磁盘人工移动到主处理机上。在多道程序环境下，可让作业从磁盘上读取各自的数据，运行的结果信息写入到输出磁盘上。最后，把输出磁盘移动到另一台外围计算机上，其任务是以最大速度读出信息并从打印机上输出。

完成上述输入和输出任务的计算机叫外围计算机，因为它不进行计算，只实现把信息从一台外围设备传送另一台外围设备上。这种操作独立于主机处理，而不在主处理机的直接控制下进行，所以称作脱机外围设备操作。脱机外围设备操作把独占使用的设备转化为可共享的设备，在一定程度上提高了效率。但却带来了若干新的问题：

- 增加了外围计算机，不能充分发挥这些计算机的功效。
- 增加了操作员的手工操作，在主处理机手外围处理机之间要来回援输入输出卷，这种手工操作出错机会多，效率低。
- 不易实现优先级调度，不同批次中的作业无法搭配运行。

因此，应进一步考虑是否能不使用外围计算机呢？由于现代计算机有较强的并行操作能力，在执行计算机的同时可进行联机外围操作，故只需使用一台计算机就可完成上述三台计算机实现的功能。操作系统将大批信息从输入设备上预先输入到辅助存储器磁盘的输入缓冲区域中暂时保存，这种方式称为“预输入”。此后，由作业调度程序调出执行。作业使用数据时不必再启动输入设备，而只要从磁盘的输入缓区域中读入。类似地，作业执行中不必直接启动输出设备输出数据，而只要将作业的输出数据暂时保存到磁盘的输出缓冲区域中，在作业执行完毕后，由操作系统组织信息成批输出。这种方式称为“缓输出”。这种设备的利用提高率提高了，其次，作业执行中不再和低速的设备联系，而直接从磁盘的输入缓冲区获得输入数据，且只要把输出信息写到磁盘的输出缓冲区就认为输出结束了，这样就减少了作业等待输入输出数据的时间，也就缩短了它的执行时间。此外，还具有能增加多道程序的道数，增加作业调度的灵活性等优点。从上述分析可以看出，操作系统提供了外围设备联机同时操作功能后，系统的效率会有很大提高。与脱机外围设备操作相比，辅助存储器上的输入和输出缓冲区域相当于输入磁盘和输出磁盘，预输入和缓输出程序完成了外围计算机做的工作。联机的同时外围设备操作又称作假脱机操作，采用这种技术后使得每个作业感到各自拥有独占使用的设备若干台。例如虽然系统只有两台行式打印机，但是可使在处理机中的 5 个作业都感到各自有一台速度如同磁盘一样快的行式打印机，所以我们说采用这种技术的操作系统提供了虚拟设备。这种技术是用一类物理设备模拟另一类物理设备技术，是使用独占使用的设备变成可共享的设备的技术。操作系统中实现这种技术的功能模块称斯普林系统。

5.7.2 斯普林系统的设计和实现

为了存放从输入设备输入的信息以及作业执行的结果，系统在辅助存储器上开辟了输入井和输出井。“井”是用作缓冲的存储区域，采用井的技术能调节供求之间的矛盾，消除人工干预带来的损失。

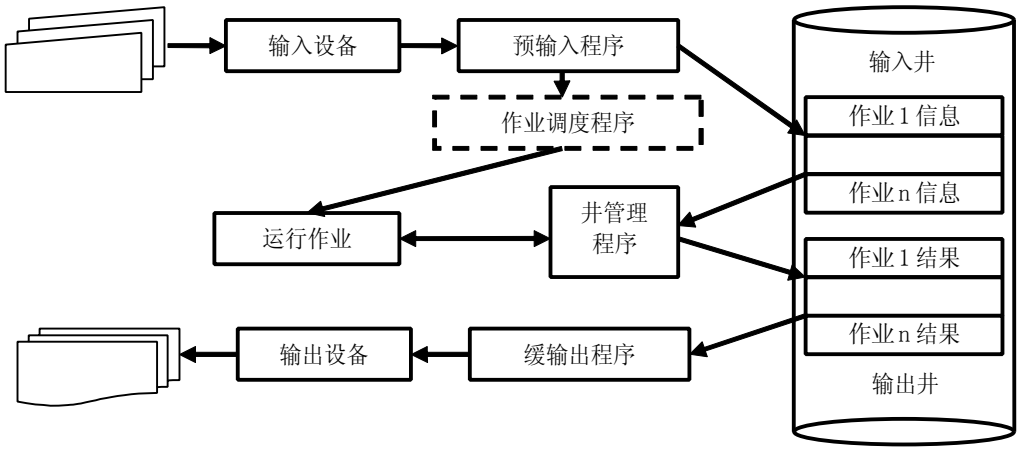


图 5-10　斯普林系统的组成和结构

图 5-10 给出了斯普林系统的组成和结构。为了实现联机同时外围操作功能，必须具有能将信息从输入设备输入到辅助存储器缓冲区域的“预输入程序”；能将信息从辅助存储器输出缓冲区域输出到输出设备的“缓输出程序”以及控制作业和辅助存储器缓冲区域之间交换信息的“井管理程序”。

预输入程序的主要任务是控制信息从输入设备输入到输入井存放，并填写好输入表以便在作业执行中要求输入信息量，可以随时找到它们的存放位置。

系统拥有一张作业表用来登记进入系统的所有作业的作业名、状态、预输入表位置等信息。每个用户作业拥有一张预输入表用来登记该作业的各个文件的情况，包括设备类、信息长度及存放位置等。

输入井中的作业有四种状态：

- 输入状态：作业的信息正从输入设备上预输入。
- 收容收态：作业预输入结束但未被选中执行。
- 执行状态：作业已被选中，它可从输入井读取信息可向输出井写信息。
- 完成状态：作业已经撤离，该作业的执行结果等待缓输出。

作业表指示了哪些作业正在预输入，哪些作业已经预输入完成，哪些作业正在执行等等。作业调度程序根据预定的调度算法选择收容状态的作业执行，作业表是作业调度程序进行作业调度的依据，是斯普林系统和作业调度程序共享的数据结构。

1、预输入程序

通常，由操作员打入预输入命令启动预输入程序进行工作。系统响应预输入命令后，调出预输入程序，它查看作业表及输入井能否新的作业。如果允许便通知操作员在输入设备上安装输入介质，然后，预输入程序在工作过程中读出和组织作业的信息，如作业名、优先数、处理机运行时间等，将获得的作业信息以及预输入表的位置登记入作业表。此后，依次读入作业的信息，在输入中井中寻找空闲块存放，把读入的信息以文件的形式登记到预输入表中，直到预输入结束。

存放在井中的文件称井文件，井文件空间的管理比较简单，它被划分成等长的物理块，用于存放一个或多个逻辑记录。可采用两种方式存放作业的数据信息。第一种方式是连接方式，输入的信息被组织成连接文件，文件的第一块信息的位置登记在预输入表中，以后各块用指针连起来，读出 n 块后，由连接指针就可找到第 n＋ 1 块数据的位置。这种方式的优点是数据信息可以不连续存放，文件空间利用率高。

第二种是计算方式，假定从读卡机上读入信息并存放到磁盘的井文件空间，每张卡片为 80 个字节，每个磁道可存放 100 个 80 字节的记录，若每个柱面有 20 个磁道，则一个柱面可以存放 2000 张卡片信息。如果把 2000 张卡片作为一叠存放在一个柱面上，于是输入数据在磁盘上的位置为：第一张卡片信息是 0 号磁道的第 1 个记录，第二张卡片信息是 0 号第 2 个记录，第 101 张卡片信息是 1 号磁道的第 1 个记录，那么，第 n 张卡片信息被存放在：

磁道号＝卡片号 n /100

记录号＝（卡片号 n）mod100

用卡片号 n 除以 100 的整数和余数部分分别为其存放的磁道号和记录号。

2、井管理程序

当作业执行过程中要求启动某台设备进行输入或输出操作时，操作系统截获这个要求并调出井管理程序控制从相应输入井读取信息或将信息送至输出井内。例如，作业 J 执行中要求从它指定的设备上读入某文件信息时，井输入管理程序根据文件名查看其预输入表获得文件 起始盘地址，可以算出每欠读请求所需的信息的存放位置。采用连接方式时，每次保留连接指针，就可将后继块的信息读入。当输入井中的信息被作业取出后，相应的井区应归还。通过预输入管理程序从输入井读入信息和通过设备管理从设备上输入信息，对用户而言是一样的。

井管理程序处理输出操作的过程与上述类似。用户作业的输出信息一律通过输出井缓冲存放，有在输出表中登记。缓输出表的格式与预输入表的格式类似，包括作业名、作业状态、文件名、设备类、数据起始位置、数据当前位置等项。在作业执行当中要求输出数据时，井输出管理程序根据有关信息查看输出表。如果表中没有这个文件名，则为第一次请求输出。文件的第一个信息块的物理位置被填入起始位置，每块信息写入井区前可用计算法计算出块号并将卡片数加 1，或将后继块位置以连接方式写入信息块的连接字中，再写到输出井，并将下一次输出时接受输出信息的位置填入数据当前位置。如果不是第一次请求输出，则缓输出表中已有登记，只要从数据当前位置便可得到当前输出信息的井区。

3、缓输出程序

当计算机的 CPU 有空闲时，操作系统调出缓输出程序进行缓输出工作，它查看缓输出表，将需要输这些文件时，可能需要组织作业或文件标题，还可能对从输出井中读出的信息进行一定格式加工。当一个作业的文件信息输出完毕后，将它占用的井区回收以供其它作业使用。

5.8 实例研究：Windows2000 的设备管理

Windows2000 设备管理继承了 NT4 设备管理的主要功能，并在 NT4 设备管理之上扩展即插即用和电源管理的功能。因此在本节中首先介绍 NT4 设备管理的主要思想，这些概念和技术在 Windows2000 设备管理中得到了继承和应用。随后介绍 Windows2000 设备管理为支持即插即用和电源管理功能，在 NT4 设备管理之上所作的修改。

5.8.1 Windows NT4 的设备管理

5.8.1.1 设计目标

Windows NT4 设备管理（I/O 系统）的设计目标如下：

- 加快单处理器或多处理器系统的 I/O 处理。
- 使用标准的 Windows2000 安全机制保护共享资源。
- 满足 WIN32、OS2 和 POSIX 子系统指定的 I/O 服务的需要。
- 提供服务，使得设备驱动程序的开发尽可能简单。
- 允许在系统中动态地添加和删除设备驱动程序。
- 支持 FAT、NTFS 和 CDFS 等多种可安装的文件系统。
- 为映像活动、文件高速缓存和应用程序提供映射文件 I/O 的能力。

5.8.1.2 I/O系统结构和模型

在 Windows NT4 中，程序在虚拟文件中执行 I/O。虚拟文件适用于所有的源与目标，例如文件、目录、管道和邮箱，它们都被当作文件来处理。所有被读取或写入的数据都可以被看作是直接到这些虚拟文件的简单的字节流。无论是 WIN32、OS2 还是 POSIX 的用户态应用程序调用文档化的函数，这些函数再依次调用内部 I/O 子系统函数来读取、写入文件和执行其他操作。I/O 管理器动态地把这些虚拟文件请求指向适当的设备驱动程序。图 5-11 说明了这种基本结构。

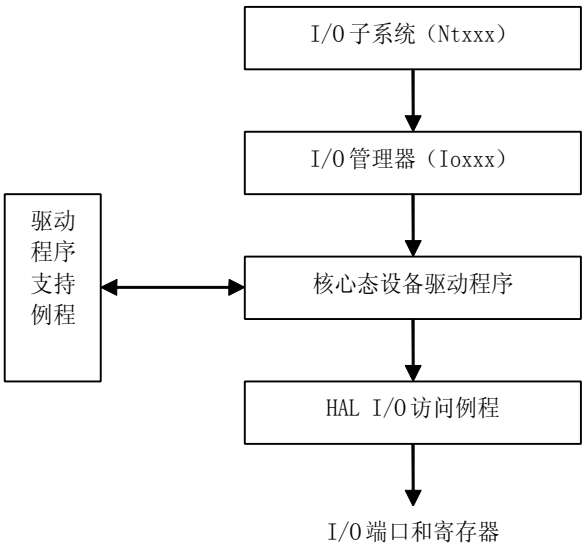


图 5-11　I/O 系统结构

其中：

- I/O 子系统 API 是内部的执行体系统服务，子系统 DLL 调用它们来实现子系统的文档化的 I/O 函数。
- I/O 管理器负责驱动 I/O 请求的处理。
- 核心态设备驱动程序把 I/O 请求转化为对硬件设备的特定的控制请求。
- 驱动程序支持例程被设备驱动程序调用来完成它们的 I/O 请求。
- 硬件抽象层 HAL I/O 访问例程把设备驱动程序与各种各样的硬件平台隔离开来，是它们在给定的体系结构家族中是二进制可移植的，并且在 Windows2000 支持的硬件体系结构中是源代码可移植的。

5.8.1.3 设备驱动程序

Windows NT4 支持以下三类设备驱动程序：

- 虚拟设备驱动程序 VDD：用于模拟 16 位 DOS 应用程序，它们俘获 DOS 应用程序对 I/O 端口的引用，并将它们转化为本机 WIN32 I/O 函数。所以 DOS 应用程序不能直接访问硬件，必须通过一个真正的核心态设备驱动程序。
- WIN32 子系统显示驱动程序和打印驱动程序（总称核心态图形驱动程序）：用于将与设备无关的图形 GDI 请求转化为设备专用请求。
- 核心态设备驱动程序：它们是能够直接控制和访问硬件设备的唯一驱动程序类型。

核心态设备驱动程序的类型有：

- 低层硬件设备驱动程序：它直接访问和控制硬件设备。
- 类驱动程序：为某一类设备执行 I/O 处理，例如磁盘、磁带、光盘。
- 端口驱动程序：实现了对特定于某一种类型的 I/O 端口的 I/O 请求的处理，如 SCSI。
- 小端口驱动程序：把对端口类型的一般的 I/O 请求映射到适配器类型，如一个特定的 SCSI 适配器。
- 文件系统驱动程序：接受到文件的 I/O 请求，并通过发布它们自己的、更明确的请求给物理设备驱动程序来满足该请求。
- 文件系统过滤器驱动程序：截取 I/O 请求，执行另外的处理，并且将它们传递给更低层的驱动程序，例如容错磁盘驱动程序 FTDISK.SYS。

不难看出，Windows NT4 的驱动程序是分层次实现的，图 5-12 给出了核心态设备驱动程序之间的关系。

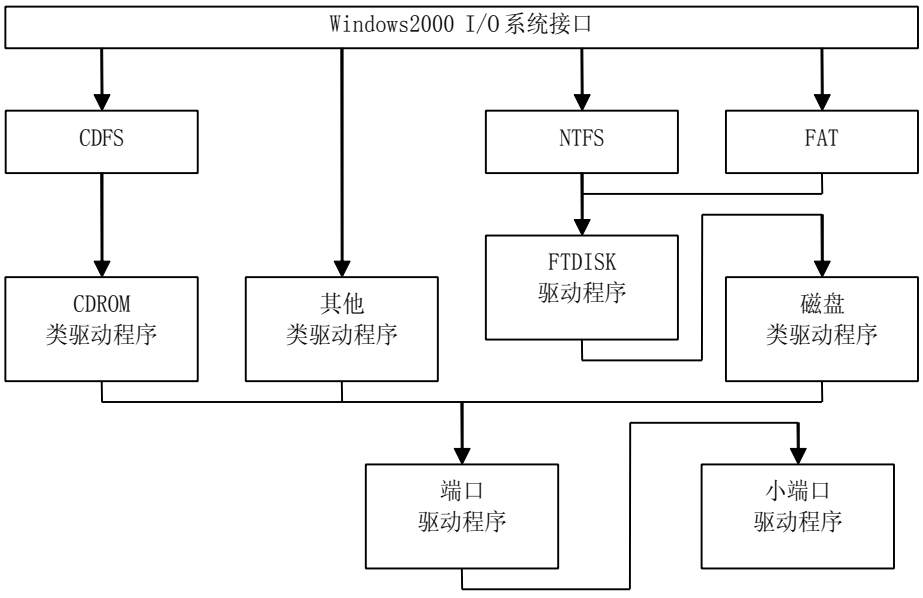


图 5-12 核心态设备驱动程序之间的关系

I/O 系统驱动设备驱动程序执行。设备驱动程序包括一组被调用去处理 I/O 请求的不同阶段的例程。以下是 5 个主要的设备驱动程序例程：

- 初始化例程：I/O 系统加载驱动程序后首先执行，以创建系统对象。
- 调度例程集：提供设备驱动调度函数。
- 启动 I/O 例程：初始化与设备之间的数据传输。
- 中断服务例程：处理设备中断。
- 中断服务 DPC 例程：在 ISR 执行以后的设备中断处理。

另外，设备驱动程序还可能有以下一些例程：

- 完成例程：用来告知分层驱动程序一个较低层的驱动程序何时完成一个 IRP 处理。
- 取消 I/O 例程：用来取消一个可以被取消的 I/O 操作。
- 卸载例程：用于释放驱动程序正在使用的系统资源，使得 I/O 管理器可以从内存当中删除它们。
- 系统关闭通知例程：用于在系统关闭时做清理工作。
- 错误纪录例程：用于在错误发生时纪录发生的事情。

5.8.2 Windows 2000 设备管理的扩展

5.8.2.1 即插即用和电源管理

即插即用和电源管理是硬件和软件支持的结合，这种结合使得计算机只需要极少的用户干预或完全不需要用户干预就能识别和适应硬件配置的更改。

最早提供即插即用支持的是 Windows95，随后得到了迅猛的发展。Windows 98 和 Windows 2000 即插即用支持不再依赖于早期的“高级电源管理”BIOS 或即插即用 BIOS，而是专门依赖于 OnNow 提出的高级配置与电源接口 ACPI。ACPI 是独立于操作系统和 CPU 的，它指定了寄存器级接口，并为其他附加的硬件特性定义描述性接口。当使用相同的操作系统驱动程序时，这些安排赋予系统设计者以不同的硬件设计实现即插即用功能及电源管理特性的能力。

5.8.2.2 设计目标

Windows2000 即插即用结构的设计目标有两个：

- 在支持用于即插即用工业硬件标准的同时，为了实现即插即用和电源管理，扩展原有的 NT 输入输出底层结构。
- 实现通用设备驱动程序接口，这些接口在 Windows98 和 Windows 2000 下支持许多设备类的即插即用和电源管理。

Windows2000 提供以下的即插即用支持：

- 已安装硬件的自动和动态识别。
- 硬件资源的分配和再分配。
- 加载适当的驱动程序。
- 与即插即用系统相互作用的驱动程序接口。
- 与电源管理的相互作用。
- 设备通知事件的登记。

5.8.2.3 驱动程序的更改

Windows2000 为了能够支持即插即用和电源管理，设备程序的初始化较 NT4 作了很大修改。这些修改如下：

- 总线驱动程序从 HAL 中分离。在新的构造中，为了与现有的核心态组件如执行体、驱动程序和 HAL 的更改和扩展一致，总线驱动程序从 HAL 中分离出去。
- 支持设备安装和设备配置的新方法和新功能。新的设计包括对 NT4 中用户态组件的更改和扩展，如：假脱机、类安装程序、控制面板应用程序和安装程序。另外系统增加了新的核心态和用户态即插即用组件。
- 从注册表重读写信息的新的即插即用 API。在新的设计中，对注册表结构进行了更改和扩展，其结构支持即插即用，同时具备向后兼容功能。

Windows2000 将支持老的设备驱动程序，但是这些设备驱动程序将不支持即插即用和电源管理。生产厂家为了使他们的设备在 Windows2000 或 Windows98 下支持即插即用和电源管理，必须重新开发新的设备驱动程序。

5.8.2.4 即插即用结构

Windows2000 的即插即用组件如图 5-13 所示。

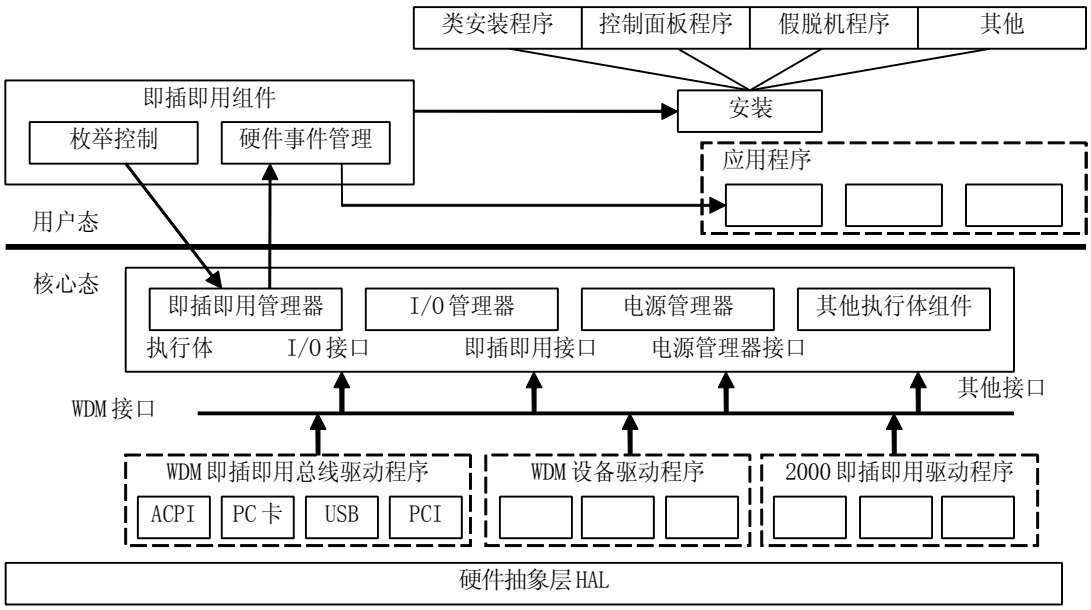


图 5-13　Windows2000 的即插即用结构

各个功能部件的作用是：

- 用户态即插即用组件：用于控制和配置设备的用户态 API。
- I/O 管理器：负责驱动 I/O 请求的处理，为设备驱动程序提供核心服务。它把用户态的读写转化为 I/O 请求包 IRP。其工作方式和工作原理与 NT4 相同。
- 核心态即插即用管理器：指导总线驱动程序执行枚举和配置、执行添加设备和启动设备操作，同时还负责协调即插即用程序的用户态部分来暂停或删除可用于这类操作的设备。即插即用管理器维护着一棵设备树，驱动程序管理器可以查看该设备树，跟踪系统中活动驱动程序及其与活动设备有关的信息。当系统添加和删除设备或进行资源再分配时，即插即用管理器便更新设备树。
- 电源管理器和策略管理器：电源管理器是核心态组件，它和策略管理器一起工作，处理电源管理 API，协调电源事件并生成电源管理 IRP。策略管理器监控系统中的活动，将用户状态、应用程序状态和设备管理程序状态集成为电源策略，在特定环境或请求条件中生成更改电源状态的 IRP。
- 即插即用 WDM 接口：I/O 系统为驱动程序提供了分层结构，这一结构包括 WDM 驱动程序、驱动程序层和设备对象。WDM 驱动程序可以分为三类：总线驱动程序、功能驱动程序和筛选驱动程序。每一个设备都含有两个以上的驱动程序层：用于支持它所基于的 I/O 总线的总线驱动程序，用于支持设备的功能驱动程序，以及可选的对总线、设备或设备类的 I/O 请求进行分类的筛选驱动程序。另外，驱动程序为它所控制的每一个设备创建设备对象。
- WDM 总线驱动程序：它控制总线电源控制和即插即用。在即插急用描述表中，任何枚举其他设备的设备都被看作是一个总线。总线驱动程序的主要任务是：枚举总线上的设备，标志它们病危它们创建设备对象；向操作系统报告总线上的动态事件；响应即插即用和电源管理 IRP；对总线的多路复用；管理总线上的设备。
- WDM 驱动程序：包括功能驱动程序/小功能驱动程序对和筛选驱动程序。在驱动程序对中，类驱动程序（一般由操作系统提供）提供所有这一类设备所需要的功能，小功能驱动程序（一般由厂家提供）提供一个特定设备所需要的功能。功能驱动程序除了为设备提供操作接口以外，还提供电源管理功能和执行操作的有关信息，该操作与休眠和满负荷工作状态之间的转换有关。

5.9 实例研究：Linux 的设备管理

5.9.1 Linux 的设备管理概述

在 Linux 操作系统中，输入输出设备可以分为字符设备、块设备和网络设备。块设备把信息存储在可寻址的固定大小的数据块中，数据块均可以被独立地读写，建立块缓冲，能随机访问数据块。字符设备可以发送或接收字符流，通常无法编址，也不存在任何寻址操作。网络设备在 Linux 中是一种独立的

设备类型，有一些特殊的处理方法。也有一些设备无法利用上述方法分类，如时钟，他们也需要特殊的处理。

在 Linux 中，所有的硬件设备均当作特殊的设备文件处理，可以使用标准的文件操作。对于字符设备和块设备，其设备文件用 mknod 命令创建，用主设备号和次设备号标识，同一个设备驱动程序控制的所有设备具有相同的主设备号，并用不同的次设备号加以区别。网络设备也是当作设备文件来处理，不同的是这类设备由 Linux 创建，并由网络控制器初始化。

Linux 核心具体负责 I/O 设备的操作，这些管理和控制硬件设备控制器的程序代码称为设备驱动程序，它们是常驻内存的底层硬件处理子程序，具体控制和管理 I/O 设备。虽然设备驱动程序的类型很多，它们都有以下的共同特性：

- 核心代码。设备驱动程序是 Linux 核心的重要组成部分，在内核运行。如果出现错误，则可能造成系统的严重破坏。
- 核心接口。设备驱动程序提供标准的核心接口，供上层软件使用。
- 核心机制和服务。设备驱动程序使用标准的核心系统服务，如内存分配、中断处理、进等待队列等待。
- 可装载性。绝大多数设备驱动程序可以根据需要以核心模块的方式装入，在不需要是可以卸装。
- 可配置性。设备驱动程序可以编译并链接进入 Linux 核心。当编译 Linux 核心时，可以指定并配置你所需要的设备驱动程序。
- 动态性。系统启动时将监测所有的设备，当一个设备驱动程序对应的设备不存在时，该驱动程序将被闲置，仅占用了一点内存而已。

Linux 的设备驱动程序可以通过查询(polling)、中断和直接内存访问等多种形式来控制设备进行输入输出。

为解决查询方式的低效率，Linux 专门引入了系统定时器，以便每隔一段时间才查询一次设备的状态，从而解决忙式查询带来的效率下降问题。Linux 的软盘驱动程序就是以这样一种方式工作的。即便如此，查询方式依然存在着效率问题。

一种高效率的 I/O 控制方式是中断。在中断方式下，Linux 核心能够把中断传递到发出 I/O 命令的设备驱动程序。为了做到这一点，设备驱动程序必须在初始化时向 Linux 核心注册所使用的中断编号和中断处理子程序入口地址，/proc/interrupts 文件列出了设备驱动程序所使用的中断编号。

对于诸如硬盘设备、SCSI 设备等高速 I/O 设备，Linux 采用 DMA 方式进行 I/O 控制，这是稀有资源一共只有 7 个。DMA 控制器不能使用虚拟内存，且由于其地址寄存器只有 16 位（加上页面寄存器 8 位），它只能访问系统最低端的 16M 内存。DMA 也不能被不同的设备驱动程序共享，因此一些设备独占专用的 DMA，另一些设备互斥使用 DMA。Linux 使用 dma_chan 数据结构跟踪 DMA 的使用情况，它包括拥有者的名字和分配标志两个字段，可以使用 cat /proc/dma 命令列出 dma_chan 的内容。

Linux 核心与设备驱动程序以统一的标准方式交互，因此设备驱动程序必须提供与核心通信的标准接口，从而使得 Linux 核心在不知道设备具体细节的情况下，仍能够用标准方式来控制和管理设备。

字符设备是最简单的设备，Linux 把这种设备当作文件来管理。在初始化时，设置驱动程序入口到 device_struct（在 fs/devices.h 文件中定义）数据结构的 chrdev 向量内，并在 Linux 核心注册。设备的主标识符是访问 chrdev 的索引。device_struct 包括两个元素，分别指向设备驱动程序和文件操作块。而文件操作块则指向诸如打开、读写、关闭等一些文件操作例行程序的地址。

块设备的标准接口及其操作方式非常类似于字符设备。Linux 采用 blk_devs 向量管理块设备。与 chrdev 一样，blk_devs 用主设备号作为索引，并指向 blk_dev_struct 数据结构。除了文件操作接口以外，块设备还必须提供缓冲区缓存接口，blk_dev_struct 结构包括一个请求子程序和一个指向 request 队列的指针，该队列中的每一个 request 表示一个来自于缓冲区的数据块读写请求。

5.9.2 Linux 的硬盘管理

一个典型的 Linux 系统一般包括一个 DOS 分区、一个 EXT2 分区（Linux 主分区）、一个 Linux 交换分区、以及零个或多个扩展用户分区。Linux 系统在初始化时要先获取系统所有硬盘的结构信息以及所有硬盘的分区信息并用 gendisk 数据结构构成的链表表示，有兴趣的同学可以参见 /include/linux/genhd 文件。

在 Linux 系统中，IDE 系统(Inergrated Disk Electronic，一种磁盘接口)和 SCSI 系统(Small Computer System Interface，一种 I/O 总线)的管理有所不同。Linux 系统使用的大多数硬盘都是 IDE 硬盘，每一个 IDE 控制器可以挂接两个 IDE 硬盘，一个称为主硬盘，一个称为从硬盘。一个系统可以有多个 IDE 控制器，第一个称为主 IDE 控制器，第二个称为从 IDE 控制器。Linux 系统最多支持 4 个 IDE 控制器，每一个控制器用 ide_hwif_t 数据结构描述，所有这些描述集中存放在 ide_hwifs 向量中。每一个 ide_hwif_t 包括两个 ide_drive_t 数据结构，分别用于描述主 IDE 硬盘和从 IDE 硬盘。

初始化时，Linux 系统在 CMOS 中查找关于硬盘的信息，并依次为依据构造上面的数据结构。Linux 系统将按照查找到的顺序给 IDE 硬盘命名。主控制器上的主硬盘的名字为 /dev/had，以下依次为/dev/hdb、

/dev/hdc、…。IDE 子系统向 Linux 注册的是的 IDE 控制器而不是硬盘，主 IDE 控制器的主设备号为 3，从 IDE 控制器的主设备号为 22。这意味着，如果系统只有两个 IDE 控制器，blk_devs 中只有两个元素，分别用 3 和 22 标识。

SCSI 总线是一种高效率的数据总线，每条 SCSI 总线最多可以挂接八个 SCSI 设备。每个设备有唯一的标识符，并且这些标识符可以通过设备上的跳线来摄制。总线上的任意两个设备之间可以同步或异步地传输数据，在数据线为 32 位时数据传输率可以达到 40MB/秒。SCSI 总线可以在设备间同时传输数据与状态信息。源设备和目标设备间的数据传输步骤最多可以有 8 个不同的阶段：

1）BUS FREE：没有设备在总线的控制下，总线上无事务发生。

2）ARBITRATION：一个 SCSI 设备试图获得 SCSI 总线的控制权，这时它把自己的 SCSI 标识符放到地址引脚上。具有最高 SCSI 标识符编号的设备将获得总线控制权。

3）SELECTION：当设备成功地获得了对 SCSI 总线的控制权之后，必须向它准备发送命令的那个 SCSI 设备发出信号。具体做法是将目标设备的 SCSI 标识符放置到地址引脚上。

4）RESELECTION：在一个请求的处理过程中，SCSI 设备可能会断开连接。目标设备将再次选择源设备。不是所有的 SCSI 设备都支持这个阶段。

5）COMMAND：源设备向目标设备发送 6B、10B 或 12B 命令。

6）DATA IN、DATA OUT：数据在源设备和目标设备之间传输。

7）STATUS：所有命令执行完毕后允许目标设备向源设备发送状态信息，以指示操作是否成功。

8）MESSAGE IN、MESSAGE OUT：信息在源设备和目标设备之间传输。

Linux SCSI 子系统包括两个基本组成部分，其数据结构分别用 host 和 device 来表示。Host 用来描述 SCSI 控制器，每个系统可以支持多个相同类型的 SCSI 控制器，每个均用一个单独的 SCSI host 来表示。Device 用来描述各种类型的 SCSI 设备，每个 SCSI 设备都有一个设备号，登记在 Device 表中。

5.9.3 Linux 的网络设备

网络设备是传送和接收数据的一种硬件设备，如以太网卡，与字符设备和块设备不一样，网络设备文件在网络设备被检测到和初始化时由系统动态产生。在系统自举或网络初始化时，网络设备驱动程序向 Linux 内核注册。网络设备用 device 数据结构描述，该数据结构包含一些设备信息以及一些操作例程，这些例程用来支持各种网络协议，可以用于传送和接收数据包。Device 数据结构包括以下几个方面的内容：

1）名称。网络设备名称是标准化的，每一个名字都能表达设备的类型，同类设备从 0 开始编号，如：/dev/ethN（以太网设备）、/dev/seN（SLIP 设备）、/dev/pppN（PPP 设备）、/dev/lo（回路测试设备）。

2）总线信息。总线信息被设备驱动程序用来控制设备，包括设备使用的中断 irq、设备控制和状态寄存器的基地址 base address、设备所使用的 DMA 通道编号 DMA channel。

3）接口标志。接口标志用来描述网络设备的特性和能力，如是否点到点连接、是否接收 IP 多路广播帧等。

4）协议信息。协议信息描述网络层如何使用设备，其中：mtu 表示网络层可以传输的最大数据包尺寸；协议表示设备支持的协议方案，如 internet 地址方案为 AF_INET；类型表示所连接的网络介质的硬件接口类型，Linux 支持的介质类型有以太网、令牌环、X.25、SLIP、PPP、以及 Apple Localtalk；地址包括域网络设备有关的地址信息。

5）包队列。等待由该网络设备发送的数据包队列，所有的网络数据包用 sk_buff 数据结构描述，这一数据结构非常灵活，可以方便地添加或删除网络协议信息头。

6）支持函数。指向每个设备的一组标准子程序，包括设置、帧传输、添加标准数据头、收集统计信息等子程序。

习题

1. 叙述设备管理的基本功能？
2. 简述各种 I/O 控制方式及其主要优缺点。
3. 试述直按内存存取 DMA 传输信息的工作原理？
4. 大型机常常采用通道实现信息传输，试问什么是通道？为什么要引入通道？
5. I/O 软件主要涉及哪些问题？分别简单说明之。
6. 叙述 I/O 中断的类型及其功能。
7. 叙述 I/O 系统的层次及其功能。
8. 什么是通道命令和通道程序？
9. 什么是通道地址字和通道状态字？

10. 叙述采用通道技术时，I/O 操作的全过程。
11. 为什么要引入缓冲技术？其实现的基本思想是什么？
12. 简述常用的缓冲技术？
13. 什么是驱动调度？有哪些常用的驱动调度技术？
14. 什么是设备独立性？它能带来什么好处？
15. 从驱动的角度来看，可数外国设备分成哪些类型？各类设备的物理特点是什么？
16. 旋转型设备上信息的优化分布能减少为若干个 I/O 服务的总时间。设磁鼓上分为 20 个区，每区存放一个记录，磁鼓旋转一周需 20 毫秒，读出每个记录平均需用 1 毫秒，读出后经 2 毫秒处理，再延续处理下一个记录。在不知当前磁鼓位置的情况下：（1）顺序存放记录取 1、……，记录 20 时，试计算说出并处理 20 个记录的总时间；（2）给出优先分布 20 个记录的一种方案，使得所花的总处理时间减少，且计算出这个方案所花的总时间。
17. 假定一种分区算法既能按指定‘设备类'，又能按指定‘设备号’进行分区，试道出这个分区外用设备的流程图。
18. 什么是‘井’？什么叫输入井和输出井？
19. 井管理程序有什么功能？它是如何工作的？
20. 什么叫虚拟设备？实现虚拟设备的主要条件是什么？
21. 什么原因使得旋转型设备比顺序型设备更适宜于共享？
22. 叙述 Spooling 系统和作业调度的关系。
23. 操作系统提供了 Spooling 功能后，系统在单位时间内处理的作业数是否增加了，为什么？每个作业的周转时间是延长了还是缩短了，为什么？
24. 试述 Windows NT4 设备管理的设计目标
25. 叙述 Windows NT4 的 I/O 系统结构和模型
26. Windows NT4 支持哪些类型的设备驱动程序？
27. Windows NT4 核心态设备驱动程序由哪些例程组成？
28. 何谓外围设备的即插即用？叙述实现该功能的基本原理。
29. 现有如下请求队列：8，18，27，129，110，186，78，147，41，10，64，12；试用查找时间最短优先算法计算处理所有请求移动的总柱面数。假设磁头当前位置下在磁道 100。
30. 上题中，分别按升序和降序移动，讨论电梯硬度算法计算处理所有请求移动的总柱面数。
31. 设单缓冲情况下，磁盘把一块数据输入缓冲区花费时间为 T；系统从缓冲区将数据传到用户区花费时间为 M；处理器处理这块数据花费时间为 C，试证明系统对一块数据的处理时间为 max(C,T)+M。
32. 试证明在双缓冲情况下，系统对一块数据的处理时间为 max(C,T)。
33. 为什么要引入设备独立性？如何实现设备独立性？
34. 目前常用的磁盘驱动调度算法有哪几种？每种适用于何种数据应用场合？
35. 假如对磁盘空间进行连续分配，试讨论其优缺点。
36. 对磁盘存在下面五个请求：

请　　求	柱　面　号	磁　头　号	扇　区　号
1	7	2	8
2	7	2	5

3	7	1	2
4	30	5	3
5	3	6	6

假如当前磁头位于 1 号柱面。试分析对这五个请求如何调度，可使磁盘的旋转圈数为最少？

37. 有一具有 40 个磁道的盘面，编号为 0~39，当磁头位于第 11 磁道时，顺序来到如下磁道请求：磁道号：1、36、16、34、9、12；试用 1）先来先服务算法 FCFS、2）最短查找时间优先算法 SSTF、3）扫描算法 SCAN 等三种磁盘驱动调度算法，计算出它们各自要来回穿越多少磁道？

38. 假定磁盘有 200 个柱面，编号 0~199，当前存取臂的位置在 142 号柱面上，并刚刚完成了 125 号柱面的服务请求，如果请求队列的先后顺序是：86，147，91，177，94，150，102，175，130；试向：为完成上述请求，下列算法存取臂移动的总量是多少？并算出存取臂移动的顺序。

    4）先来先服务算法 FCFS；

    5）最短查找时间优先算法 SSTF；

    6）扫描算法 SCAN。

CH6 文件管理

文件系统是操作系统中负责存取和管理信息的模块，它用统一的方式管理用户和系统信息的存储、检索、更新、共享和保护，并为用户提供一整套方便有效的文件使用和操作方法。文件这一术语不但反映了用户概念中的逻辑结构，而且和存放它的辅助存储器（也称文件存储器）的存储结构紧密相关。所以，同一个文件必须从逻辑文件和物理文件两个侧面来观察它。对于用户来说，可按自己的愿望并遵循文件系统的规则来定义文件信息的逻辑结构，由文件系统提供“按名存取”来实现对用户文件信息的存储和检索。可见，使用者在处理他的信息时，只需关心所执行的文件操作及文件的逻辑结构，而不必涉及存储结构。但对文件系统本身来说，必须采用特定的数据结构和有效算法，实现文件的逻辑结构到存储结构的映射，实现对文件存储空间和用户信息的管理，提供多种存取方法。所以，文件系统面向用户的功能是：

- 文件的按名存取
- 文件目录建立和维护
- 实现从逻辑文件到物理文件的转换
- 文件存储空间的分配和管理
- 提供合适的文件存取方法
- 实现文件的共享、保护和保密
- 提供一组可供用户使用的文件操作

为了实现这些功能，操作系统必须考虑文件目录的建立和维护、存储空间的分配和回收、数据的保密和监护、监督用户存取和修改文件的权限、在不同存储介质上信息的表示方式、信息的编址方法、信息的存储次序、以及怎样检索用户信息等问题。

6.1 文件

本节将从用户角度来研究文件，即逻辑文件的建立和使用。

6.1.1 文件的概念

早期计算机系统中没有文件管理机构，用户自行管理辅助存储器上的信息，按照物理地址安排信息，组织数据的输入输出，还要记住信息在存储介质上的分布情况，繁琐复杂、易于出错、可靠性差。大容量直接存取存储器的问世为建立文件系统提供了良好的物质基础。多道程序、分时系统的出现，多个用户以及系统都要共享大容量辅助存储器。因而，现代操作系统中都配备了文件系统，以适应系统管理和用户使用软件资源的需要。对计算机系统中软件资源的管理形成了操作系统的文件系统。

文件是由文件名字标识的一组相关信息的集合。文件名是字母或数字组成的字母数字串，它的格式和长度因系统而异。

组成文件的信息可以是各式各样的：一个源程序、一批数据、各类语言的编译程序可以各自组成一个文件。文件名可以按各种方法进行分类：如按用途可分成：系统文件、库文件和用户文件；按保护级别可分成：只读文件、读写文件和不保护文件；按信息流向可分成：输入文件、输出文件和输入输出文件；按存放时限可分成：临时文件、永久文件、档案文件；按设备类型可分成：磁盘文件、磁带文件、软盘文件。此钱同学 可以按文件的逻辑结构或物理结构进行分类。

操作系统提供文件系统后，首先，用户使用方便，使用者无需记住信息存放在辅助存储器中的物理位置，也无需考虑如何将信息存放在存储介质上，只要知道文件名，给出有关操作要求便可存取信息，实现了“按名存取”。特别，当文件存放位置作了改变，甚至更换了文件的存储设备，对文件的使用者也没有丝毫影响;其次，文件安全可靠，由于用户通过文件系统才能实现对文件的访问，而文件系统能 提供各种安全、保密和保护措施，故可防止对文件信息的有意或无意的破坏或窃用。此外，在文件使用过程中可能出现硬件故障，这时文件系统可组织重执，对于硬件失效而可 能造成文件信息破坏，可组织转储以提高文件的可靠性。最后，文件系统还能提供文件的共享功能，如不同的用户可以使用同名或异名的同一文件。这样，既节省了文件存放空间，又减少了传递文件的交换时间，进一步提高了文件和文件空间的利用率。把数据组织成文件形式加以管理和控制是计算机数据管理的重大发展。

6.1.2 文件的命名

文件是一个抽象机制，它提供了一种把文件保存在磁盘上而且便于以后读取的方法，用户不必了解信息存储的方法、位置以及存储设备实际运作方式等细节。在这一抽象机制中最重要的是文件命名，当一个进程创建一个文件时必须给出文件名字，以后这个文件将独立于进程存在直到它被显式地删除；当其他进程要使用这一文件时必须显式地指出该文件名字；操作系统也将根据该文件名字对文件进行保护。

各个操作系统的文件命名规则略有不同，即文件名的格式和长度因系统而异。但一般来说，文件名

都是字母或数字组成的字母数字串。

例如，在 MS-DOS 系统中，一个文件的名字包括文件名和扩展名两部分；前者用于识别文件，长度为 1 到 8 个字符；后者用于标识文件特性，长度为 0 到 3 个字符；两者之间用一个圆点隔开。文件名和扩展名均不区分大小写，可用字符包括字母、数字及一些特殊符号。扩展名常常用作定义各种类型的文件，系统有一些约定扩展名，例如：COM 表示可执行的二进制代码文件；EXE 表示可执行的浮动二进制代码文件；LIB 表示库程序文件；BAT 表示批命令文件；OBJ 表示编译或汇编生成的目标文件等。

定义文件扩展名是一种习惯，它并不是源于 MS-DOS，尽管一些操作系统的文件名中允许存在多个圆点，如 Windows-98 和 Unix，但是文件扩展名的定义习惯依然被大多数前端用户和应用所默认。

许多文件系统支持多达 255 个字符的文件名，如 Windows，也有很多操作系统的文件命名需要区分大小写，如 Unix。

6.1.3 文件的类型

在现代操作系统中，对于文件乃至设备的访问都是基于文件进行的，例如，打印一批数据就是向打印机设备文件写数据，从键盘接收一批数据就是从键盘设备文件读数据。操作系统一般支持以下几种不同类型的文件：

- 普通文件：即前面所讨论的存储在外存储设备上的数据文件。
- 目录文件：管理和实现文件系统的系统文件。
- 块设备文件：用于磁盘、光盘或磁带等块设备的 I/O。
- 字符设备文件：用于终端、打印机等字符设备的 I/O。

一般来说，普通文件包括 ASCII 文件或者二进制文件，ASCII 文件有多行正文组成，在 DOS、Windows 等系统中每一行以回车换行结束，整个文件以 CTRL+Z 结束；在 Unix 等系统中每一行以换行结束，整个文件以 CTRL+D 结束。专门的实用命令 Unix2dos 和 dos2Unix 可以实现两类 ASCII 文件的转换。ASCII 文件的最大优点是可以原样显示和打印，也可以用通常的文本编辑器进行编辑。另一种正规文件是二进制文件，它往往有一定的内部结构，组织成字节的流，如：可执行文件是指令和数据的流，记录式文件是逻辑纪录的流。

6.1.4 文件的属性

大多数操作系统设置了专门的文件属性用于文件保护，这组属性包括：

- 文件的类型属性：如普通文件、目录文件、系统文件、隐式文件、设备文件等。
- 文件的保护属性：如可读、可写、可执行、可创建、可删除等。

6.1.5 文件的存取

从用户使用观点来看，他们关心的是数据的逻辑结构，即记录及其逻辑关系，数据独立于物理环境；从系统实现观点来看，数据则被文件系统按照某种规则排列和存放到物理存储介质上。那么，输入的数据如何存储？处理的数据如何检索？数据的逻辑结构和数据物理结构之间怎样接口？谁来完成数据的成组和分解操作？这些都是存取方法的任务。存取方法是操作系统为用户程序提供的使用文件的技术和手段。

在有些系统中，对每种类型文件仅提供一种存取方法。但象 IBM 系统能支撑许多不同的存取方法，以适应用户的不同需要，因而，存取方法也就成为文件系统中重要的设计问题了。文件类型和存取方法之间存在密切关系，因为，设备的物理特性和文件类型决定了数据的组织，也就在很大程度上决定了能够施加于文件的存取方法。

6.1.5.1 顺序存取

按记录顺序进行读／写操作的存取方法称顺序存取。固定长记录的顺序存取是十分简单的。读操作总是读出下一次要读出的文件的下一个记录，同时，自动让文件记录读指针推进，以指向下一次要读出的记录位置。如果文件是可读可写的。再设置一个文件记录指针，它总指向下一次要写入记录的存放位置，执行写操作时，将一个记录写到文件末端。允许对这种文件进行前跳或后退 N（整数）个记录的操作。顺序存取主要用于磁带文件，但也适用于磁盘上的顺序文件。

对于可变长记录的顺序文件，每个记录的长度信息存放于记录前面一个单元中，它的存取操作分两步进行。读出时，根据读指针值先读出存放记录长度的单元，然后，得到当前记录长后再把当前记录一起写到指针指向的记录位置，同时，调整写指针值。

由于顺序文件是顺序存取的，可采用成组和分解操作来加速文件的输入输出。

6.1.5.2 直接存取

很多应用场合要求以任意次序直接读写某个记录，例如，航空订票系统，把特定航班的所有信息用航班号作标识，存放在某物理块中，用户预订某航班时，需要直接将该航班的信息取出。直接存取方法便适合于这类应用，它通常用于磁盘文件。

为了实现直接存取，一个文件可以看作由顺序编号的物理块组成的，这些块常常划成等长，作为定位和存取的一个最小单位，如一块为 1024 字节、4096 字节，视系统和应用而定。于是用户可以请求读块 22、然后，写块 48，再读块 9 等等，直接存取文件对读或写块的次序没有限制。用户提供给操作系统的是相对块号，它是相对于文件开始位置的一个位移量，而绝对块号则由系统换算得到。

6.1.5.3 索引存取

第三种类型的存取是基于索引文件的索引存取方法。由于文件中的记录不按它在文件中的位置，而按它的记录键来编址，所以，用户提供给操作系统记录键后就可查找到所需记录。

通常记录按记录键的某种顺序存放，例如，按代表健的字母先后次序来排序。对于这种文件，除可采用按键存取外，也可以采用顺序存取或直接存取的方法。信息块的地址都 可以通过查找记录键而换算出。实际的系统中，大都采用多级索引，以加速记录查找过程。

6.1.6 文件的使用

用户通过两类接口与文件系统联系：第一类是与文件有关的操作命令或作业控制语言中与文件有关的语句，例如，Unix 中的 cat， cd， cp， find， mv， rm， mkdir， rmdir 等等，这些构成了必不可少的文件系统人——机接口。第二类是提供给用户程序使用的文件类系统调用，构成了用户和文件系统的另一个接口，通过这些指令用户能获得文件系统的各种服务。一般地讲，文件系统提供的基本文件类系统调用有：

- 建立文件：当用户要求把一批信息作为一个文件存放在存储器中时，使用建立操作向系统提出建立一个文件的要求。
- 打开文件：文件建立之后能立即使用，要通过‘打开’文件操作建立起文件和用户之间的联系。文件打开以后，直至关闭之前，可被反复使用，不必多次打开，这样做能减少查找目录的时间，加快文件存取速度，从而，提高文件系统的运行效率。
- 读/写文件：文件打开以后，就可以用读/写系统调用访问文件，调用这两个操作，应给出以下参数：文件名、主存缓冲地址、读写的记录或字节个数；对有些文件类型还要给出读/写起始逻辑记录号。
- 文件控制：文件打开以后，把文件的读写指针定位到文件头、文件尾、或文件中的任意位置，或执行前跳、后退等各种控制操作。
- 关闭文件：当一个文件使用完毕后，使用者应关闭文件以便让别的使用者用此文件。关闭文件的要求可以直接向系统提出；也可用隐含了关闭上次使用同一设备上的另外一个文件时，就可以认为隐含了关闭上次使用过的文件要求。调用关闭系统调用的参数与打开操作相同。
- 撤销文件：当一个文件不再需要时，可向系统提出撤销文件。

6.2 文件目录

6.2.1 文件目录与文件目录项

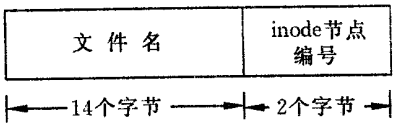
文件系统怎样实现文件的“按名存取”？如何查找文件存储器中的指定文件？如何有效地管理用户文件和系统文件？文件目录便是用于这些方面的重要手段。文件系统的基本功能之一就是负责文件目录的建立、维护和检索，要求编排的目录便于查找、防止冲突，目录的检索方便迅速。

有了文件目录后，就可实现文件的“按名存取”。每一个文件在文件目录中登记一项。文件目录项一般应该包括以下内容：

- 有关文件存取控制的信息：如文件名、用户名、授权者存取权限：文件类型和文件属性，如读写文件、执行文件、只读文件等。
- 有关文件结构的信息：文件的逻辑结构，如记录类型、记录个数、记录长度、成组因子数等。文件的物理结构，如记录存放相对位置或文件第一块的物理块号，也可指出文件索引的所在 位置。
- 有关文件管理的信息：如文件建立日期、文件最近修改日期、访问日期、文件保留期限、记帐信息等。

有了文件目录后，就可实现文件的“按名存取”。当用户要求存取某个文件时，系统查找目录项并比较文件名就可找到 所寻文件的目录项。然后，通过目录项指出的文件名就可找到所寻文件的目录项，然后通过目录项指出文件的文件信息相对位置或文件信息首块物理位置等就能依次存取文件信息。

Unix 采用了一种比较特殊的目录项建立方法。为了减少检索文件访问的物理块数，Unix 把目录中的文件和其它管理信息分开，后者单独组成定长的一个数据结构，称为索引节点（i-node）。索引号，记为 I-NO。于是，文件目录项中仅剩下 14 个字节的文件名和两个字节的 I-NO，因此，一个物理块可存放 32 个目录项，系统把由目录项组成的目录文件和普通文件一样对待，均存放在文件存储器中。



文件存储设备上的每一个文件，都有一个外存文件控制块(又称外存索引节点)inode 与之对应，这些 inode 被集中放在文件存储设备上的 inode 区。文件控制块 inode 对于文件的作用，犹如进程控制块 proc、user 对于每个进程的作用，集中了这个文件的属性及有关信息，找到了 inode，就获得了它所对应的文件的一切必要信息。每一个外存的 inode 结构使用 32 个字节，包含如下信息：文件长度及在存储设备上的物理位置、文件主的各种标识、文件类型、存取权限、文件勾连数、文件访问和修改时间、以及 inode 节点是否空闲。下面列出索引节点的部分内容:

- di-mode 文件属性，如文件类型、存取权限。
- di nlike 连接该索引节点的目录项数（共享数）。
- di-uid 文件主用户标识。
- di-gid 文件同组用户标识。
- di-size 文件大小（以字节计数）
- di-add[8]存放文件所在物理块号的索引表。
- di-atime 文件最近被访问的时间。
- di-mtime 文件最近被修改的时间。
- di-ctime 文件最近创建的时间。

外存索引节点 inode 记录了一个文件的属性和有关信息。可以想象，在对某一文件的访问过程中，会频繁地涉及到它，不断来回于内、外存之间引用它，当然是极不经济的。为此，UNIX 在系统占用的内存区里开辟了一张表——内存索引节点 inode 表，(又称活动文件控制块表或活动索引节点表)，该表共有 100 个表目，每个表目称为一个内存索引节点 inode，当需要使用某文件的信息，而在内存 inode 表中找不到其相应的 inode 时，就申请一个内存 inode，把外存 inode 的内容拷贝到这个内存 inode 中，随之就使用这个内存 inode 来控制文件的读写。通常，在最后一个用户关闭此文件后，内存索引节点 inode 的内容被写到外存索引节点 inode 中，然后释放以供它用。

把文件目录与索引节点分开，不仅加快了目录检索速度，而且，便于实现文件共享，有利于系统的控制和管理。

6.2.2 一级目录结构

如图 6-1 所示，最简单的文件目录是一级目录结构，在操作系统中构造一张线性表，与每个文件有关的属性占用一个目录项就成了一级目录结构。单用户微型机操作系统 CP/M 的软盘文件便采用这一结构，每个磁盘上设置一张一级文件目录表，不同磁盘红色动器上的文件目录互不相关。文件目录表由长度为 32 字节的目录项组成，目录项 0 称目录头，记录有关文件目录表的信息，其它每个目录项又称文件控制块。文件目录中列出了盘上全部文件的有关信息。CP/M 操作系统中文件目录项包括：盘号、文件名、扩展名、文件范围、记录数、存放位置等。

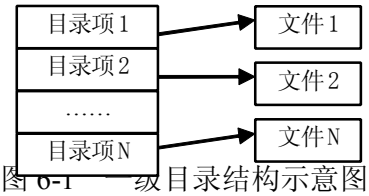


图 6-1 一级目录结构示意图

一级文件目录结构存在若干缺点：一是重名问题，它要求文件名和文件之间有一一对应关系，但要在多用户的系统中，由于都使用同一文件目录，一旦文件名用重，就会出现混淆而无法实现‘按名存取’。如果人为地限制文件名命名规则，对用户来说又极不方便；二是难于实现文件共享，如果允许不同用户使用不同文件名来共享同一个文件，这在一级目录中是很难实现的，为了解决上述问题，操作系统往往采用二级目录结构，使得每个用户有各自独立的文件目录。

6.2.3 二级目录结构

在二级目录中，第一级为主文件目录，它用于管理所有用户文件目录，它的目录项登记了系统接受的用户的名字及该用户文件目录的地址。第二级为用户文件目录，它为该用户的每个文件保存一登记栏，其内容与一级目录的目录项相同。每一用户只允许查看自己的文件目录。图 6-2 是二级文件目录结构示意。当一个新用户作业进入系统执行时，系统为其在主文件目录中开辟一个区域的地址填入主文件目录中的该用户名所在项。当用户需要访问某个文件时系统根据用户名从主文件目录中找出该用户的文件目录的物理位置，其余的工作与一级文件目录类似。

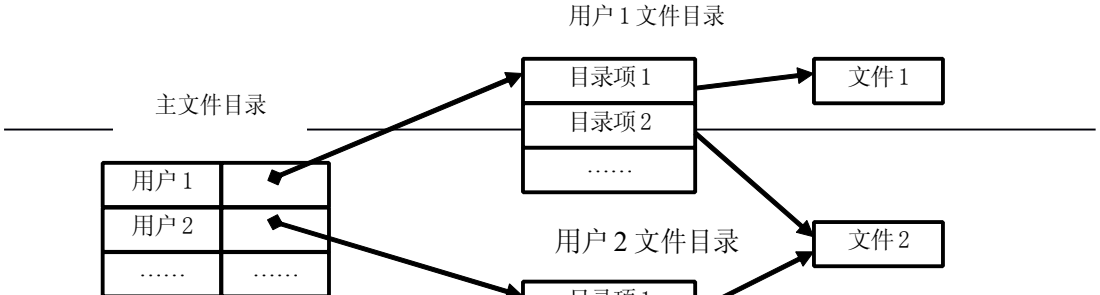


图 6-2　二级目录结构示意图

采用二级目录管理文件时，因为任何文件的存取都通过主文件目录，于是可以检查访问文件者的存取权限，避免一个用户未经授权就存取另一个用户的文件，使用户文件的私有性得到保证，实现了对文件的保密和保护。特别是不同用户具有同名文件时，由于各自有不同的用户文件目录而不会导致混乱。对于文件的共享，原则上只要把对应目录项指向同一物理位置的文件即可。

6.2.4 树形目录结构

二级目录的推广形成了多级目录。每一级目录可以是下一级目录的说明，也可以是文件的说明，从而，形成了层次关系。如图 6-3 所示，多级目录结构通常采用树形结构，它是一棵倒向的有根的树，树根是根目录；从根向下，每一个树枝是一个子目录；而树叶是文件。树型多级目录有许多优点；较好地反映现实世界中具有层次关系的数据集合和较确切地反映系统内部文件的分支结构；不同文件可以重名，只要它们不是同一末端的子目录中，易于规定不同层次或子树中文件的不同存取权限便于文件的保护、保密和共享等。

在树形目录结构中，一个文件的全名将包括从根目录开始到文件为止，通路上遇到的所有子目录路径。各子目录名之间用正斜线/（或反斜线/）隔开，其中，由于子目录名组成的部分又称为路径名。

MS-DOS2.0 以上版本采用树形目录结构。在对磁盘进行格式化时，在其上所建一目录，称为根目录，对于双面盘言，根目录能存放的的文件数为 112 个。但根目录除了能存放文件目录外，还可以存放子目录，依次类推。可以形成一个树形目录结构。子目录和根目录不同，可以看作一般的文件，可以像文件一样进行读写操作，它们被存放在盘上的数据区中，也就是说允许存放任何数目的文件和子目录。

Unix 操作系统的文件系统也采用树形多级目录结构，在根目录之下有：dev 设备子目录； bin 实用程序子目录；lib 库文件子目录；etc 基本数据和维护实用程序子目录；tmp 临时文件子目录；usr 通用目录。 在 usr 下通常包含有一个已安装的文件系统，包括：小型化的 bin、小型化的 tmp、小型化的文件库文件 lib 包括文件 include 及各用户的多种文件。此外，Unix 操作系统自身 vmUnix 也在根目录下。

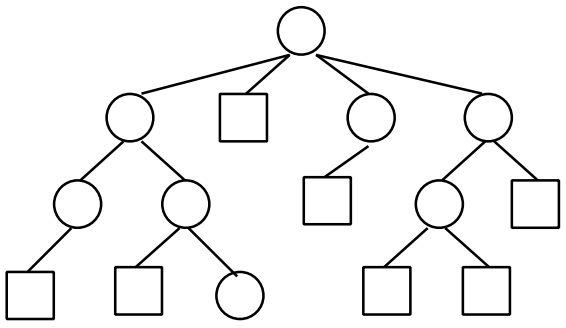


图 6-3　树形目录结构示意图

6.3 文件组织与数据存储

6.3.1 文件的存储

目前广泛使用的文件存储介质是磁盘、光盘和磁带，在微型机上则多数采用软盘及硬盘。一盘磁带、

一张光盘片、一个硬盘分区或一张软盘片都称为一卷。卷是存储介质的物理单位。对于软盘驱动器、光盘驱动器、磁带驱动器和可撤卸硬盘驱动器等设备而言，由于存储介质与存储设备可以分离，所以，物理卷和物理设备不总是一致的，不能混为一谈。一个卷上可以保存一个文件（叫单文件卷）或多个文件（叫多文件卷），也可以一个文件保存在多个卷上（叫多卷文件）或多个文件保存在多个卷上（叫多卷多文件）。

块是存储介质上连续信息所组成的一个区域，也叫做物理记录。块是主存储器和辅助存储设备进行信息交换的物理单位，每次总是交换一块或整数块信息。决定块的大小要考虑到用户使用方式、数据传输效率和存储设备类型等多种因素。不同类型的存储介质，块的长短常常各不相同；对同一类型的存储介质，块的长短也可以不同。有些外围设备由于启停机机械动作的要求或识别不同块的特殊需要，两个相邻块之间必须留有间隙。间隙是块之间不记录用户代码信息的区域。

6.3.2 文件的逻辑结构

6.3.2.1 流式文件和记录式文件

文件的组织是指文件中信息的配置和构造方式，通常应该从文件的逻辑结构和组织及文件的物理结构和组织两方面加以考虑。文件的逻辑结构和组织是从用户观点出发，研究用户概念中的抽象的信息组织方式，这是用户能观察到的，可加以处理的数据集合。由于数据可独立于物理环境加以构造，所以称为逻辑结构。一些相关数据项的集合称作逻辑记录，而相关逻辑记录的集合称作逻辑文件。系统提供若干操作以便使用者构造他的文件，这样，用户不必顾及文件信息的逻辑构造，利用文件名和有关操作结构的问题，而只需了解文件信息的逻辑构造，利用文件名和有关操作就能存储、检索和处理文件信息。显然，用户对于逻辑文件的兴趣远远大于物理文件。然而，为了提高操作效率，对于各类设备的物理特性及其适宜的文件类型仍应心中有效，换句话说，存储设备的物理特性会影响到数据的逻辑组织和采用的存取方法。

文件的逻辑结构分两种形式：一种是流式文件，另一种是记录式文件。流式文件指文件内的数据不再组成记录，只是依次的一串信息集合，也可以看成是只有一个记录的记录式文件。这种文件常常按长度来读取所需信息，也可以用插入的特殊字符作为分界。事实上，有许多类型的文件并不需要分记录，象用户作业的源程序就是一个顺序字符流，硬要分割源程序文件成若干记录只会带来操作复杂、开销增大的缺点。因而，为了简化系统，大多数现代操作系统对用户仅仅提供流式文件，纪录式文件往往由高级语言或简单的数据库管理系统提供。

记录式文件内包含若干逻辑记录，逻辑记录是文件中按信息在逻辑上的独立含意划分的一个信息单位，记录在文件中的排列可能有顺序关系，但除此以外，记录与记录之间不存在其它关系。在这一点上，文件有别于数据库。早期有计算机常常使用卡片输入输出机，一个文件由一迭卡片组成，每张卡片对应于一个逻辑记录，这类文件中的逻辑记录可以依次编号。逻辑记录的概念被应用于许多场合，特别象数据库管理系统中已是必不可少的了。如，某单位财务管理文件中，每个职工的工资信息是包托姓名、部门、应发工资、扣除工资、奖金和实发工资等若干数据项组成的一个逻辑记录。整个单位职工的工资信息、即全部逻辑记录便组成了该单位的工资信息文件。

从操作系统管理的角度来看，逻辑记录是文件内独立的最小信息单位，每次总是为使用者存储、检索或更新一个逻辑记录。但使用者使用语言的角度来看，还可以把逻辑记录进一步划分成一个或多个更小的数据项。以 COBOL 语言为便，数据项是具有标识名的最小的不可分割的数据单位，数据项的集合构成逻辑记录，相关逻辑记录的信合构成了文件。所以，逻辑记录又可被定义为：与同一实体有关的逻辑记录的集合。这样逻辑文件也就可被定义为：与同一实体有关的逻辑记录的集合。图 4-已清楚地表明了数据项、逻辑记录和逻辑文件的关系。上述工资信息逻辑文件中有若干逻辑记录，每一个逻辑记录表示一个职工的工资信息，其中每个记录中的姓名、扣除工资等都是独立的数据项，当然，扣除工资数据项还可进一步划分为扣除房租、水电费……，扣除水电费又可进一步划分水费、电费等等更低层次的数据项。在 COBOL 语言的数据描述中要对逻辑记录的数据项的类型、位数和层次等详加说明。一个数据项有一个标识名并赋予它某种物理含义，一个数据项还包含一个值，即数据、数据的类型及其表示法。语言处理程序知道逻辑文件中逻辑记录的组织，知道逻辑记录中每个数据项的物理含义，当从文件系统获得一个逻辑记录后，便可自行分解出数据项进行处理。

通常，用户是按照他的特定需要设计数据项、逻辑记录和逻辑文件的。虽然，用户并不要了解文件的存储结构，但是他应该考虑到各种可能的数据表示法，考虑到数据处理的简易性、有效性和扩充性，考虑到某种类型数据的检索方法。因为，对于用户的某种应用来说，一种类型的表示和组织方法可能比另一种更为适合。所以，在设计逻辑文件时，应该考虑到下到诸因素：

- 如果文件信息经常要增、删、改，那么，便要求能方便地添加数据项到逻辑记录中，添加逻辑记录到逻辑文件中，否则可能造成重新组织整个文件。
- 数据项的数据表示法主要取决于数据的用法。例如，大多数是数值计算，只有少量作字符处理，则数据项以十进制或二进制数的算术表示为宜。
- 数据项的数据应有最普遍的使用形式。例如，数据项“性别”，可用 1 表示男性，2 表示女性。在显示性别这一项时，需要把 1 转换成男性，2 转换成女性。如果性别采用西文的‘M’或‘F'

或者中文的‘男’和‘女’分别表示男性和女性时，这种转换是不需要的。

- 依赖计算机的数据表示法不能被不同的计算机处理，但是字符串数据在两个计算机系统之间作数据交换时，是最容易的数据表示法。因而，这类应用尽可能采用字符串数据。
- 所有高级程序设计语言都不能支持所有的数据表示法。例如 FORTRAN 使用各个分时应是同类型的向量数组，而 COBOL 却可使用不同类型数据项的记录型数据结构。

6.3.2.2 成组和分解

现在讨论逻辑记录和块之间的关系。由于逻辑记录是按信息在逻辑上的独立含义划分的单位，而块是存储介质上连续信息所组成的区域。因此，一个逻辑记录被存放到文件存储器的存储介质上时，可能占用一块或多块，也可以一个物理块包含多个逻辑记录。如果把文件比作书，逻辑记录比作书中的章节，那么，卷是册而块是页。一本名叫《操作系统教程》的书可以是一册（单卷文件），也可以为多册（多卷文件），当然，也允许《操作系统教程》及《操作系统习题和实习题》两本或多本书装成一册（多文件郑）或多册（多卷多文件）。书中的一个章节占一页或多页，也允许一页中包含若干章节。书和章节相当于文件和逻辑记录，它们是逻辑概念；而册和页相当于卷和块，它们是物理概念，两者不能混淆。

若干个逻辑记录合并成一组，写入一个块叫记录成组，这时每块中的逻辑记录的个数称块因子。成组操作一般先在输出缓冲区内进行，凑满一块后才将缓冲区内的信息写到存储介质上。反之，当存储介质上的一个物理记录读进输入缓冲区后，把逻辑记录从块中分离出来的操作叫记录的分解。通常，对于穿孔卡片和行 式打印同，把存储介质上的． 划分成 80 字节或 160 字节长。那第，一张卡片的 80 个字节的数据是一个逻辑记录，也是一个物理记录。在这两个例子中，逻辑记录和物理记录是等长的。假定把卡片上的数据写到磁带上，可以规定磁带上的物理记录为 800 字节，这样，每块内就可放 10 张卡片数据，这时块因子数等于 10，如果卡片上的数据存放到磁盘上，可以规定磁盘存储介质的物理记录长为 1600 字节，每块内就可容纳 20 张卡片数据，这时块因子数等于 20。后两者的逻辑记录长小于物理记录长，是成组处理的例子。

记录成组和分解处理不仅节省存储空间，还能减少输入输出操作次数，提高系统效率。记录成组和分解的处理过程如图 6-4 所示，当记录成组和分解处理时，用户的第一个读请求，导致文件管理将包含逻辑记录的整个物理块读入主存输入缓冲区，使用户获得所需的第一个逻辑。随后的读请求可直接从主缓冲区取得相继的逻辑记录，直到该块中的逻辑记录全部处理完毕，紧接着的读请求便重复上述过程。用户写请求的操作过程相反，开始的若干命令仅将所处理的逻辑记录依次传送到输出缓冲区装配。当某一个写请求传送的逻辑记录恰好填满缓冲区时，文件管理才发出一次输入输出操作请求，将该缓冲区的内容写到存储介质的相应块中。

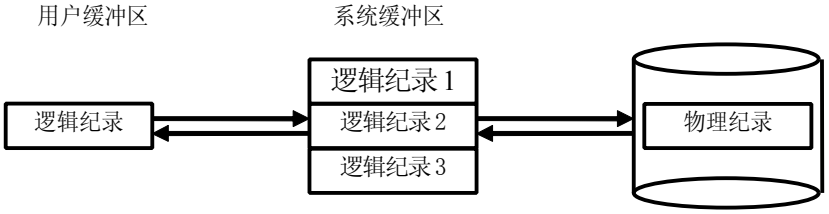


图 6-4　记录成组和分解的处理过程

采用成组和分解方式处理记录的主要缺点是：需要软件进行成组和分解的额外操作；需要能容纳最大块长的输入输出缓冲区。

6.3.2.3 记录格式和记录键

记录式文件中的记录可以有不同的记录格式，提供多种记录格式是考虑到数据处理和各种应用、输入输出传输功效、存储空间利用率和存储设备硬件特点等多种因素。为了存储、检索、处理和加工，必须按预先建立的可受的类型和格式把信息提交给操作系统。一个逻辑记录中所有数据项长度的总和称该逻辑记录的长度。

现将前面已经介绍过的术语，加以小结：

- 逻辑记录——文件中按信息在逻辑上的独立含义划分的一种信息单位。它是可以用一个记录键唯一地标识的相关信息的集合；它被操作系统看作一个作独立处理的单位；应用程序往往 分解逻辑记录成若干数据项进行相应处理。
- 物理记录——存储介质上连续信息所组成的上个区域，它是主存储器和辅助存储设备进行信息交换的物理单位。综合考虑应用程序的需要、存储设备类型等因素由操作设定块长。
- 存储记录——指附加了操作系统控制信息的逻辑记录，它被文件管理看作一个独立处理单位。存储记录除了包含与逻辑记录相同的内容外，还增加了系统描述和处理记录所需要的信息。

系统应将存储记录映射成用户可接受的逻辑记录格式和存储设备上能存储的物理记录格式。图 6-5 表明了三种记录之间的关系。

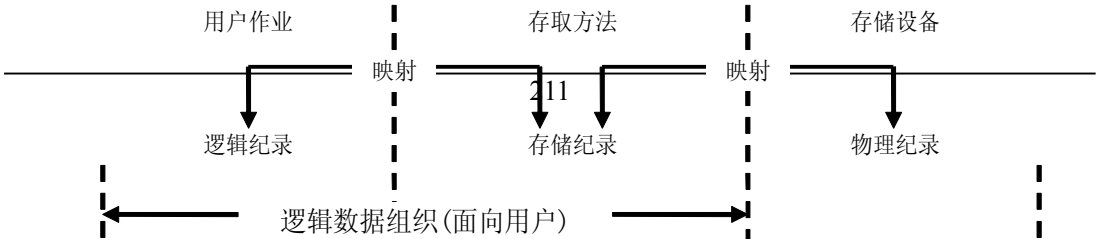


图 6-5　逻辑记录、存储记录和物理记录之间的关系

现在讨论记录的格式和结构。记录格式就是记录内数据的排列方式，显然，这种方式要保证能为操作系统接受。在记录式文件中，记录的长度可以为定长的或变长的，这取决于应用程序的需要。用户可以选择下列一种或多种记录格式：

- 格式 F：定长记录
- 格式 V：变长记录
- 格式 S：跨块记录

记录格式主要由用户按所处理数据的性质来选取，用户着眼于逻辑记录结构，然而，输入输出设备的物理特性会影响使用的记录格式。为了处理记录，系统必须获得记录的有关信息。在有些情况下，这些信息来自应用程序，其它场合则由系统自动提供。

定长记录（格式 F）指一个记录式文件中所有的逻辑记录都具有相同的长度，同时所有数据项的相对位置也是固定的。定长记录由于处理方便、控制容易，在传统的数据处理中普启遍采用。定长记录可以成组或不成组，成组时除最未一块外，每块中的逻辑记录数为一常数。在搜索到文件未端且最后一块的逻辑记录数小于块因子数时，操作系统能发现并加以处理。

变长记录（格式 V）指一个记录式文件中，逻辑记录的长度不相等，但每个逻辑记录的长度处理之前能预先确定。有两种情况会造成变长记录：

- 包含一个或多个可变的长度的数据项
- 包含了可变数目的定长数据项。

虽然变长记录处理复杂，但却有很大优点，那么，这个长度应为诸记录中可能变化很大，如果采用定长记录，那么，这个定长应为诸记灵中可能出现的最大长度。这样做会浪费许多存储空间，对于一个大型文件是不适合的，对于这类文件如采用变长记录能节省大量存储空间。变长记录可以是成组的，变长逻辑记录附加了控制信息，已被扩展成存储记录。对每个逻辑记录而言，控制信息 RL 为记录长度，指示单个变长记录的字节个数（注意也包括了控制信息 RL 本身的长度）。对于每个物理记录而言，控制信息 BL 为块长度， 指示物理记录中包括控制信息在内的字节总数。控制信息均用双字，但 RL 和 BL 实用 4 个字节，剩余部分保留扩充时使用。

通常，存储介质上的块划分成固定长度后，当处理的可变长记录大于块长时，不论是否成组处理，均会发生逻辑记录跨越物理块的情形，这就是跨块记录（格式 S）。在跨块记录的情况下，逻辑记录被分割成段写到块中，读出时再作装配，段的分割和装配工作不需用户承担，而由文件系统自动实现。

文件在不同物理特性的设备类型之间传送时，跨块记录特别有用。例如可用于具有不同块长的接收类型设备和发送类型设备间的信息传送。另一个应用例子是在正文编辑处理中，可存放非常长的正文行记录。跨块记录是可变长记录 处理的一种延伸，其主要差别是：变长记录处理时，程序员必须知道输入输出缓冲区的大小，而跨块记录处理时却不必要。文件系统将自动分割和装配跨块逻辑记录的信息段，而这些块不会超过输入输出缓冲区的容量。

跨块记录的结构和变长记录完全相同。当应用程序中指明处理跨块记录后，根据需要可将逻辑记录分割成若干段输出，或在输入时将一段或多段再装配成一个逻辑记录。一个完整的逻辑记录可以全部写入一块中，也可以写到相邻的若干中。前者一段就是一个逻辑记录，后者若干段组成一个逻辑记录。跨块记录可以成组或不成组，因此，一个物理块可能包含一个或多个逻辑记录的一部分或全部。在 处理成组跨块记录时，一个逻辑记录可以开始于任何一个物理的任何位置。

为了方便文件的组织和管理，提高文件记录的查找效率，通常，对逻辑文件的每个逻辑记录至少设置一个与之对应的基本数据项，利用它可与同一文件中的其它记录区别开来。这个用于标识记录某个逻辑记录的记录键，称为主键，也叫关键字，简称键。能唯一地标识某个逻辑记录的记录键，称为主键。例如

- 工资信息记录中的职员编号；
- 住房信息记录中的住址；
- 订货信息记录中订货号；
- 银行存款信息记录中的存折号；

都可用作相应记录的主键。一个逻辑文件中，主键是最重要的，但它不是唯一的，如果一个部门没有本同姓名的职员，那么，工资信息记录既允许使用职工编号，也允许使用职工姓名作为主键。

如果采用单键记录，即不同的逻辑记录仅设置一个不同的键，它就能唯一地标识这个记录。因而，存取一个记录的信息不必根据内容或记录地址，只需按照记录键就可实现。在很多应用场合，要求多个键对应于同一记录，按其中的任何一个键均可搜索到此记录，这叫多键记录。事实上一个记录中的任一数据项或若干数据项的组合均可作为记录键，我们总可以为一个记录找到若干个键。除主键外的其它键都称作次键。次键一般不能在若干记录中将特定记录唯一的地标识出来。例如，工资信息文件，可将应发工资作为记录键，但月薪 1000 元的职工却不止一个人，因而，它可当作次键，而不能为主键使用。多键记录类似于图书馆给图书编索引，可根据书名，著名和主题分别编目，最终都可找到同一本书。使用多键记录，可以查工具有某个键的那些记录，这就引出了有广泛用途的倒排文件的概念。记录键常为一字符串，由使用者提供。最简单的记录键可以和逻辑记录号对应，这时就简化成一字数字串。

6.3.3 文件的物理结构

文件系统往往根据存储设备类型、存取要求、记录使用频度和存储空间容量等因素提供若干种文件存储结构。用户看到的是逻辑文件，处理的是逻辑记录，按照逻辑文件形式去存储，检索和加工有关的文件信息，也就是说数据的逻辑结构和组织是面向应用程序的。然而，这种逻辑上的文件总得以不同方式保存到物理存储设备的存储介质上去，所以，文件的物理结构和组织是指逻辑文件在物理存储空间中存放方法和组织关系。这时，文件看作为物理文件，即相关物理块的集合。文件的存储结构涉及块的划分、记录的排列、索引的组织、信息的搜索等许多问题。因而，其优劣直接影响文件系统的性能。

有两类方法可用来构造文件的物理结构。第一类称计算法，其实现原理是设计一映射算法，例如线性计算法、杂凑法等，通过对记录键的计算转换成对应的物理块地址，从而找到所需记录。直接寻址文件、计算寻址文件，顺序文件均属此类。计算法的存取效率教高，又不必增加存储空间存放附加控制信息，能把分成布范围较广的键均匀地映射到一个存储区域中。第二类称指针法，这类方法设置专门指针，指明相应记录的物理地址或表达各记录之间的关联。索引文件、索引顺序文件、连接文件、倒排文件等均属此类。使用指针的优点是可将文件信息的逻辑次序与在存储介质上的物理排列次序完全分开，便于随机存取，便于更新，能加快存取速度。但使用指针要耗用较多存储空间，大型文件的索引查找要耗用较多处理机理机时间，所以，究竟采用哪种文件存储结构，必须根据应用目标、响应时间和存储空间等多种因素进行权衡折。

下面介绍常用的几种文件的物理结构和组织。

6.3.3.1 顺序文件

将一个文件中逻辑上连续的信息存放到存储介质的依次相邻的块上便形成顺序结构，这类文件叫顺序文件，又称连续文件。显然，这是一种逻辑记录顺序和物理记录顺序完全一致的文件，通常，记录按出现的次被读出或修改。

一切存于磁带上的文件都只能是顺序文件，此外，卡片机、打印机、纸带机介质上的文件也属类。这类文件是一种最简单的文件组织形式，在数据处理历史上最早使用。而存储在磁盘或软盘上的文件，也可以组织成顺序。 时可由文件目录指出存放该文件信息的第一块存储地址和文件长度。为了改善顺序文件的处理效率，用户常常对顺序文件中的记录按某一个或几个数据项的值从小（大）到大（小）重新排列，经排列处理后，记录有某种确定的次序，成为有顺序文件。有序文件能较好地适应批处理等顺序应用。

顺序文件的基本优点是：顺序存取记录时速度较快。所以，批处理文件，系统文件用得最多。采用磁带存放顺序文件时，总可以保持快速存取的优点。若以磁盘作存储介质时，顺序文件的记录也按物理邻接次序排列，因而，顺序的盘文件能象带文件一样进行严格的顺序处理。然而，由于多程序访问，在同一时间另外的用户作业可能驱动磁头移向其它文件，因而可能要花费较多的处理器时间隆低了这一优越性。顺序文件的主要缺点是：建立文件前需要能预先确定文件长度，以便分配存储空间；修改、插入和增生文件记录有困难；对直接存储器作连续分配，会造成少量空闲块的浪费。

逻辑记录连续地存储在存储介质的相邻物理块上的文件也叫紧凑顺序文件。当插入和修改记录时，往往导致移动大部记录，为了克服这一严重缺点，对直接存取存储器，可以采用许多顺序文件的变种。扩展顺序文件是在文件内设有空白区域以备预先估计的添加记录使用，这样做虽需较大的存储容量，但在一定程度上适应了文件的扩展性。连接顺序文件中设置溢出区，添加的记录通过连接方法保存到溢出区中，所以对逻辑记录来说是顺序的但文件的存储结构已不一定顺序排列。划分顺序文件是在直接存取设备上有较地利用文件的一种方法。把一个文件划分成几个能独立存取的顺序文件。每个文件由数据区和索引区组成，数据区内存放各个文件，索引区内存放各个子文件的引，包括名字、地址、长度和状态等，空闲区也由索引指明。划分顺序文件本质上是顺序文件，它是有能直接存取各子文件的优点，特别适用于程序库和宏指令库等系统文件的使用。

6.3.3.2 连接文件

连接结构的特点是使用连接字，又叫指针来表示文件中各个记录之间的关系。如图 6-6 所示，第一块文件信息的物理地址由文件目录给出，而每一块的连接字指出了文件的下一个物理块。通常，连接字内容为 0 时，表示文件至本块结束。这种文件叫连接文件，又称串联文件，像输入井、输出井等都用此类文件。

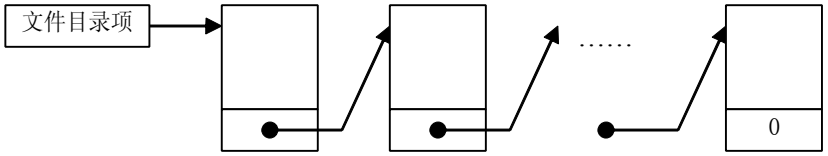


图 6-6　连接文件结构示意图

引进指向其它数据的连接表示是计算机程序设计的一种重要手段，是表示复杂数据关系的一种重要方法。使用指针可以将文件的逻辑记录顺序与它所在存储空间；此外，必须将连接字与数据信息存放在

一起，而破坏了物理块的完整性；由于存取须通过缓冲区，待获得连接字后，才能找到下一物理块的地址，因而，仅适宜于顺序存取。连接结构恰好克服了顺序结构不适宜于增、删、改等的固有缺点，对某些操作带来很大好处，但工其它方面又失去了一些性能。

在系统软件中，常常使用某些连接存储结构，它们按记录之间的相对线性位置进行组织。按其插入和删除记录操作的方式不同，可以分成三种情况：

- 堆栈——其所有记录的插入和删除操作只能在同一端进行，这一端称栈顶。堆栈中一个‘后进先出’型数据结构，这是因为后进入栈的记录，一定比先进入栈的所有记录先退出栈。由于记录之间没有循环或相交的关系，且记录的逻辑顺序与物理顺序相一致，所以，除栈指针，可以不再设置连接字，这时的存放方式退化为顺序结构。对于不能事先估计栈元素最大值的场合，难以采用上这固定存区法，这时就在每个记录中增加指向前一个记录地址的连接字，这就是堆栈的连接存储法。堆栈运算有特殊的名称，把一个新的记录插入栈中，使之成为栈的新顶项叫下推运算；反之，删除栈顶记录叫上推运算，大多数上推需要读取顶项记录以便运算。
- 队列——其记录的插入在后端进行，而删除在前端进行，又叫‘先进先出’型数据结构。这是因为从时间上看，先进入队列的记录总比后进入队列的记录先退出队列。为了进行排队及实现入队和出队操作，可采用复杂的连接结构，常用的有：前向、双向和环状指针。前向指针，指每个记录中含有下一个记录的地址，大多数使用的是相对地址。双向指针，指每个记录中除保留有下一个记录的地址外，还含有上一个记录的地址，即同时包含前向和后向指针。使用双向指针的优点是：便于双向搜索；队列中任意记录出队后，便于剩余记录构成连接。环状指针，指首尾两个记录也分别用后向指针和前向指针连接起来的结构。
- 两端队列——左右两端均可进行插入和删除记录操作的队列。

上述连接存储结构在操作系统和语言编辑中用得很多，例如，用作符号运算栈、信息保护栈、进程状态队列、井文件等，但通常不提供给应用程序使用。

6.3.3.3 直接文件

在直接存取存储设备上，记录的关键字与其地址之间可以通过某种方式建立对应关系，利用这种关系实现存取的文件叫直接文件。这种存储结构是通过指定记录在介质上的位置进行直接存取的，记录无所谓次序。而记录在介质上的位置是通过对记录的键施加变换而获得相应地址，这种变换法就是常用的散列法，或叫杂凑法，利用这种方法构造的文件常称直接文件或散列文件。这种存储结构用在不能采用顺序组织方法、次序较乱、又需在极短时间内存取的场合，象实时处理文件、操作系统目录文件、编译程序变量名表等特别有效；此外，又不需索引，节省了索引存储空间和索引查找时间。

计算寻址结构中较困难的是‘冲突’问题。一般说来，地址的总数和可能选择的关键字之间不存在一一对应关系。因此，不同的关键字可能变换出相同的地址来，这就叫冲突。一种散列算法是否成功的一个重要标志是将不同键映射成相同地址的几率有多大，几率越小冲突就越小，则此散列算法的性能也越好。解决冲突会增加相当多的额外代价，因而，‘冲突’是计算寻址结构性能变坏的主要因素。解决冲突的办法叫溢出处理技术，这是设计散列文件需要考虑的主要内容。常用的溢出处理技术有：顺序探查法、两次散列法、拉链法、独立溢出区法等。

计算机寻址结构的一个特例是：把键作为记录的存取地址。这是一种直接了当的散列方法，又叫直接寻址法。如果键的范围和所使用的实际地址范围相等时，这是一种十分理想的存取方法。但大部分情形下，键值范围大大超过所用的地址范围，因而，这种方法仅在个别场合采用。

6.3.3.4 索引文件

索引结构是实现非连续存储的另一种方法，适用于数据记录保存有随机存取存储设备上的文件。如图 6-7 所示，它使用了一张索引表，其中每个表目包含一个记录的键及其记录数据的存储地址，存储地址可以是记录的物理地址，也可以是记录的符号地址，这种类型的文件称索引文件。通常，索引表的地址可由文件目录指出，查阅索引表先找到的相应记录键，然后获得数据存储地址。

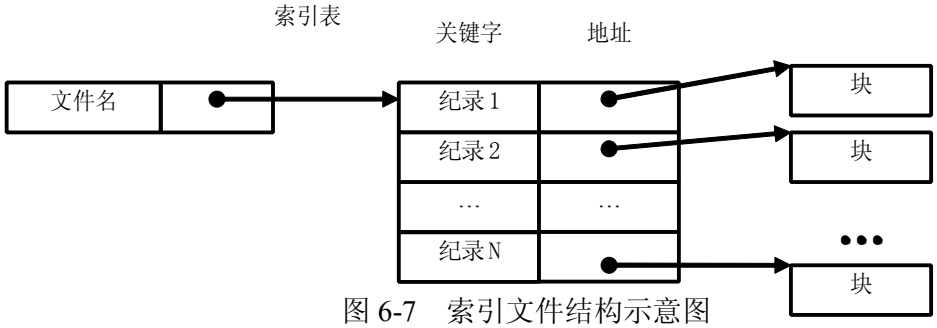


图 6-7　索引文件结构示意图

索引文件在文件存储器上分两个区：索外区和数据区。访问索引文件需两步操作：第一步查找文件索引，第二步以相应键登记项内容作为物理或符号地址而获得记录数据。这样，至少需要两次访问辅助

存储器，但若文件索引已预先调入主存储器，那么就可减少一次内外存信息交换。

索引结构是连接结构的一种扩展，除了具备连接文件的优点外，还克服了它只能作顺序存取的缺点，具有直接读写任意一个记录的能力，便于文件的增、删、改。索引文件的缺点是：增加了索引表的空间开销和查找时间，索引表的信息量甚至可能远远超过文件记录本身的信息量。

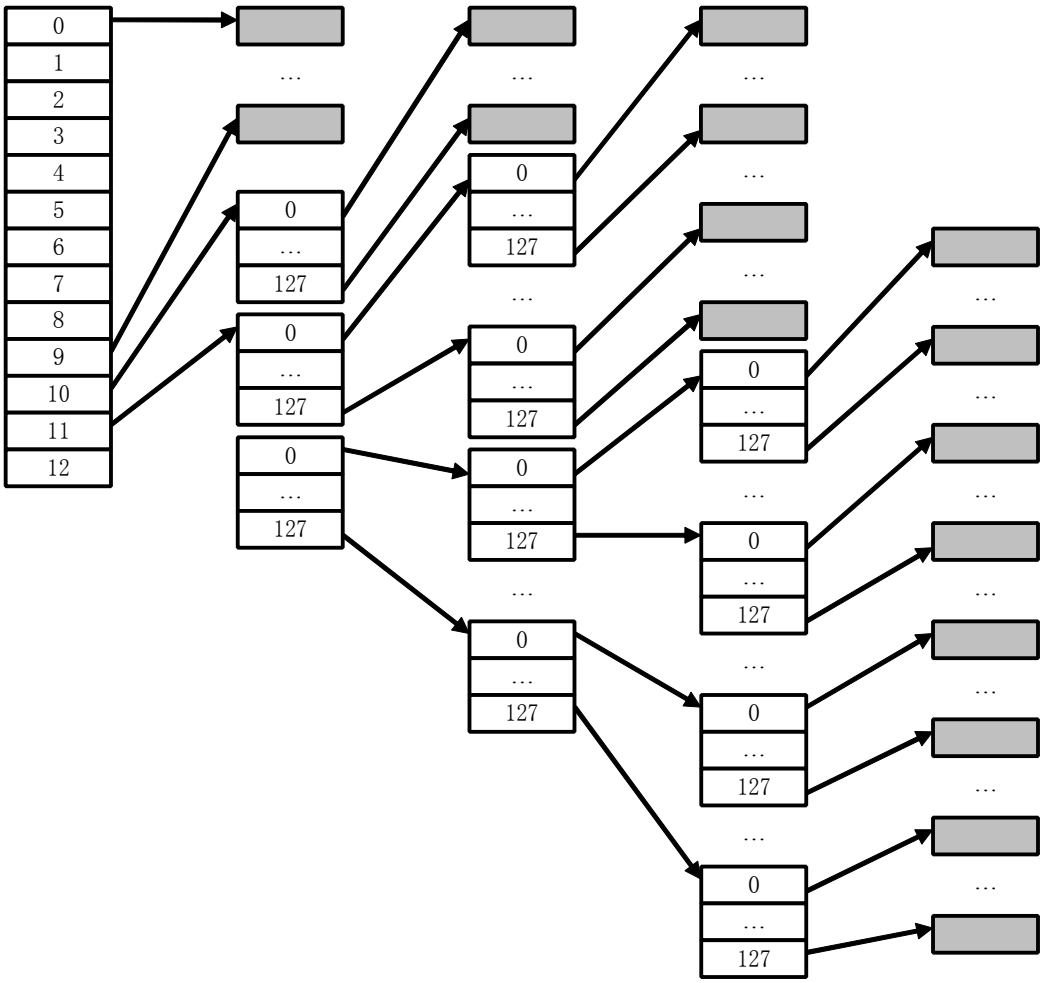
索引文件中的索引项可以分为两类：一类稠密索引，即对每个数据记录，在索引表里都有一个索引项。因而索引本身很大，但可以不要求数据记录排序，通过对索引的依次查找就可确定记录的位置或记录是否存在。另一种称稀疏索引，它对每一组数据记录有一索引项。因而索引表本身较小，但数据记录必须按某种次序排列。注意，虽然稀疏索引本身较小，但是，在查找时又要花出一定代价。因为找到索引之后，只判定了记录所在的组，而该记录是否存在？是组内哪一个记录？还要进一步查找。

索引顺序文件是顺序文件的扩展，其中各记录本身在介质上也是顺序排列的，它包含了直接处理和修改记录的能力。索引顺序文件能象顺序文件一样进行快速顺序处理，既允许按物理存放次序（记录出现的次序）；也允许按逻辑顺序（由记录主键决定的次序）进行处理。

有时记录数目很多，索引表要占用许多物理块。因此，在查找某键对应的索引项时，可能依次需交换很多块。若索引表占用 n 块，则平均要交换(n+1) / 2 次，才能找到所需记录的物理地址，当 n 很大时，这是很费时间的操作。提高查找速度的另一种办法是：做一个索引的索引，叫二级索引。二级索引表的表项列出一级索引表每一块最后一个索引项的键值及该索引表区的地址，也就是说，若干个记录的索引本身也是一种记录。查找时先查看二级索引表找到某键所在的索引表区地址，再搜索一级索引表找出数据记录。当记录数目十分大时，索引的索引也可能占用许许多多块，那样，可以做索引的索引的索引，叫三级索引。有的计算机系统还建立更多层次的索引，当然这些工作都由文件系统完成。

图 6-8　Linux 的多重索引结构

Linux 操作系统的多重索引结构稍有不同。如图 6-8 所示，每个文件的索引表规定为 13 个索引项，



每项 4 个字节，登记一个存放文件信息的物理块号。由于 Unix 文件系统仅提供流式文件，无记录概念，因而，登记项中没有键与之。前面 10 项．存放文件今年的物理块号，叫直接寻址，而 0 到 9，可以理解为文件的逻辑块号。如果文件大于 10 块，则利用第 11 项指向一个物理块，该块中最多可放 128 个存放

文件信息的物理块的块号叫一次间接寻址，因为每个大型文件还可以利用第 12 和 13 项作二次和三次间接寻址，因为每个物理块存放 512 个字节，所以 Unix 每个文件 最大长度这 11 亿字节。这种方式的优点是与一般索引文件相同，其缺点是多次间接寻址降低了查找速度。对分时使用环境统计表明，长度不超过 10 个物理块的文件占总数的 80%，通过直接寻址便能找到文件的信息。对仅占总数的 20%的超过 10 个物理块的文件才施行间接寻址。

6.4 文件的保护和保密

6.4.1 文件的保护

文件保护是指防止文件被破坏，它包括两个方面：一是防止系统崩溃所造成的文件破坏；二是防止其他用户的非法操作所造成的文件破坏。

为防止系统崩溃造成文件破坏，定时转储是一种经常采用的方法，系统的管理员每隔一段时间，或一日、或一周、或一月、或一个期间，把需要保护的文件保存到另一个介质上，以备数据破坏后恢复。如一个单位建立了信息系统，往往会准备多个磁带，以便数据库管理员每天下班前把数据库文件转储到磁带上，这样即使出现了数据库损坏，最多只会丢失一天的数据。由于需要备份的数据文件可能非常多，增量备份时必须的，为此操作系统专门为文件设置了档案属性，用以指明该文件是否被备份过。操作系统往往也提供备份和转储工具以方便用户转储，如 DOS 的 XCOPY 命令、BACKUP 命令和 RESTORE 命令，Windows 的备份工具，Unix 的 compress 命令、tar 命令、bar 命令等。第三方公司也提供这样一些备份工具，比较著名的有 arj、lha、winzip 等等。一些应用程序本身也携带备份工具，如数据库管理系统。另外，一些备份工具甚至支持自动定时转储。

必须看到，虽然定时转储的成本较小，但是它不能完全做到百分之一百的数据恢复，这对于实时应用和重要的商业应用来说是不够的。为此又引入了多副本技术，即把重要的数据存放在不同的物理磁盘、乃至不同的联网计算机上，这样系统崩溃造成文件破坏后，系统完全可以从另一个文件副本中读取数据。多副本技术的实现又分为静态多副本和动态多副本，静态多副本实现的代价较小，但不能做到多个副本的完全同步。基于文件的多副本技术的实现，特别是动态多副本的实现较为复杂，开销很大，因此现代操作系统和数据库系统往往采用磁盘冗余的方法实现多副本，即两个磁盘存放同样的信息，如磁盘镜像技术和 RAID5 技术。采用磁盘冗余后，读数据时可以从任何一个磁盘中读，不会造成系统性能降低，但是写数据则需要向两个磁盘各写一次，这就需要考虑性能问题。显然，由操作系统的设备驱动程序直接控制两次写的效率将明显高于应用进程控制两次写磁盘，这就是磁盘镜像往往由操作系统而不是数据库系统来提供的原因。一种更高效的解决方法是计算机系统拥有两个通道，通过两个通道各控制一个磁盘，在硬件上实现同时读写。

至于要防止其他用户的非法操作所造成的文件破坏，这往往通过操作系统的安全性策略来实现，其基本思想是建立如下的三元组：

（用户、对象、存取权限）

其中：

- 用户是指每一个操作系统使用者的标识。
- 对象在操作系统中一般是文件，因为操作系统把对资源的统一到文件层次，如通过设备文件使用设备、通过 socket 关联文件使用进程通信等。
- 存取权限定义了用户对文件的访问权，如：读、写、删除、创建、执行等等。一个安全性较高的系统权限划分的较多较细。

要实现这一机制必须建立一个如图 6-9 所示的存取控制矩阵，它包括两个维，一维列出所有用户名，另一维列出全部文件，矩阵元素的内容是一个用户对于一个文件的存取权限，如用户 1 对文件 1 有读权R，用户 3 对文件 1 既有读权 R，又有写权 W 和执行权 X。

	用户 1	用户 2	用户 3	……
文件 1	R--	---	RWX	……
文件 2	RW-	RWX	---	……
文件 3	---	---	R-X	……
……	……	……	……	……

图 6-9　存取控制矩阵

显然存取控制矩阵中有大量的空数据，即用户 X 对文件 Y 没有存取权，为节省存储、方便实现，可以它线性化成如图 6-10 所示的存取控制表，当用户 X 对文件 Y 有存取权，则在该表中插入一个元组，否则不执行插入。

用户名	文件名	存取权限
用户 1	文件 1	R--
用户 1	文件 2	RW-
用户 2	文件 2	RWX
……	……	……

图 6-10　存取控制表

不难看出，存取控制表的存储量和检索开销也不小，对操作系统这样一种效率要求极高的软件来说有没有更好的解决方案呢？我们可以把用户划分为几类，如：文件属主、合作者、其他用户，规定这几类用户对文件的存取权限并把它保存在文件目录项中，称之为文件属性。以 Unix 和 Linux 为例，它把用户分为属主、同组用户、其他用户三类，分别定义存取权限可读 r、可写 w、可执行 x，目录项中的文件属性共有 10 位：

-rwxrwxrwx

其中：

- 第 1 位：表示文件是普通文件(-)，还是目录文件(d)、符号链文件(l)、设备文件(b/c)。
- 第 2-4 位：表示文件属主对文件的存取权限。
- 第 5-7 位：表示同组用户对文件的存取权限。
- 第 8-10 位：表示其他用户对文件的存取权限。

如一个文件的属性是-rwxr-x--x，表示该文件是普通文件，属主对它可读、可写、可执行，同组用户对它可读、可执行，其他用户对它只可执行。

6.4.2 文件的保护

文件保密的目的是防止文件被窃取。主要方法有设置口令和使用密码。

口令分成两种：文件口令是用户为每个文件规定一个口令，它可写在文件目录中并隐蔽起来，只是提供的口令与文件目录中的口令一致时， 才能使用这个文件。另一种是终端口令，由系统分配或用户预先设定一个口令，仅当回答的口令相符时才能使用该终端。但是它有一个明显的缺点，当要回收某个用户的使用权时，必须更改口令，而更改后的新口令又必须通知其他的授权用户，这无疑是不方便的。

使用密码是一种更加有效的文件保密方法，它将文件中的信息翻译成密码形式，使用时再解密。

在网络上进行数据传输时，为保证安全性，经常采用密码技术；进一步还可以对在网络上传输的数字或模拟信号采用脉码调制技术，进行硬加密。

6.5 文件系统其他功能的实现

6.5.1 文件操作的实现

文件系统提供给用户程序的一组系统调用，包括：建立、打开、关闭、撤销、读、写和控制，通过这些系统调用用户能获得文件系统的各种服务。

文件系统在为用户程序服务时，需要沿路径查找目录时以获得有关该文件的各种信息，这往往要多次访问文件存储器，使访问速度大大减慢。若把所有文件目录都复制到主存，访问速度是加快了，但又增加了主存的开销。一种行之有效的办法是把常用和正在使用的那些文件目录复制进主存，这样，既不增加太多的主存开销，又可明显减少查找时间，系统可以为每个用户进程建立一张活动文件表，当用户使用一个文件之前先通过‘打开’操作，把该文件有关目录复制到指定主存区域。当不再使用该文件时，使用‘关闭’切断用户进程和该文件索引的联系。同时，若该目录已被修改过，则应更新辅存中对应的文件目录。在 Unix 操作系统中，第个进程图的 usr 结构中专门设立了有 15 个表目的活动文件表—用户打开文件表，因此，一个进程最多可同时打开 15 个文件。采用打开文件表的办法之后，每当访问一个文件时，只需先查找活动文件表就可知道文件是否打开，若已打开，就可以对这个文件进行读写操作。这样一个文件被打开以后，可被用户多次使用，直至文件被关闭或撤销，大大节省了文件操作时间。

6.5.1.1 建立文件

当用户要求把一批信息作为一个文件存放在存储器中时，使用建立操作向系统提出建立一个文件的要求。用户使用‘建立’系统调用时，通常应提供以下参数：文件名、设备类（号）、文件属性及存取控制信息，如：文件类型、记录大小、保护级别等。

文件系统完成此系统调用的主要工作是：

- 根据设备类（号）在选中的相应调设备上建立一个文件目录，并返回一个用户文件标识，用户在以后的读写操作中可以利用此文件标识；

- 将文件名及文件属性等数据填入文件目录；
- 调用辅存空间管理程序，为文件分配第一个物理；
- 需要时发出装卷信息（如磁带或可磁盘组）；
- 在活动文件表中登记该文件有关信息，文件定位，卷标处理；

在某些操作系统中，可以隐含的执行‘建立’操作，即当系统发现有一批信息要写进一个尚未建立的文件中时，就自动先建立文件，完成上述步骤后，接着再写入信息。

6.5.1.2 打开文件

文件建立之后能立即使用，要通过‘打开’文件操作建立起文件和用户之间的联系。打开文件常常使用显式、即用户使用‘打开’系统调用直接向系统提出。用户打开文件时需要给出文件名和设备类（号）。

文件系统完成此系统调用的主要工作是：

- 在主存活动文件表中目录申请一个空项，用以存放该文件的文件目录信息；
- 根据文件名查找目录文件，将找到的文件目录信息复制到活动文件表占用栏；
- 若打开的是共享文件，则应有相应处理，如使用共享文件的用户数加 1；
- 文件定位，卷标处理；

文件打开以后，直至关闭之前，可被反复使用，不必多次打开，这样做能减少查找目录的时间，加快文件存取速度，从而，提高文件系统的运行效率。

6.5.1.3 读 / 写文件

文件打开以后，就可以用读 / 写系统调用访问文件，调用这两个操作，应给出以下参数：文件名、主存缓冲地址、读写的记录或字节个数；对有些文件类型还要给出读 / 写起始逻辑记录号。

文件系统完成此系统调用的主要工作是：

- 按文件名从活动文件表中找到该文件的目录项：
- 按存取控制说明检查访问的合法性；
- 根据目录项指出的该文件的逻辑和物理组织方式将逻辑记录号或个数转换成物理块号；
- 向设备管理程序发 I/O 请求，完成数据交换工作。

6.5.1.4 关闭文件

当一个文件使用完毕后，使用者应关闭文件以便让别的使用者用此文件。关闭文件的要求可以通显式，直接向系统提出；也可用隐含了关闭上次使用同一设备上的另外一个文件时，就可以认为隐含了关闭上次使用过的文件要求。调用关闭系统调用的参数与打开操作相同。

文件系统完成此操作的主要工作是：

- 将活动文件表中该文件的‘当前使用用户’减 1；若此值为 0，则撤销此表目；
- 若活动文件表目内容已被改过，则应先将表目内容写回文件存储器上相应表目中，以使文件目录保持最新状态；
- 卷定位工作。

6.5.1.5 撤销文件

当一个文件不再需要时，可向系统提出撤销文件，该系统调用所需的参数为文件名和设备类（号）。

撤销文件时，系统要做的主要工作是：

- 若文件没有关闭，先做关闭工作；若为共享文件，应进行联访处理；
- 在目录文件中删去相应目录项；
- 释放文件占用的文件存储空间。

6.5.2 文件操作的执行过程

下面简单介绍系统调用的控制和执行过程，即从用户发出文件系统调用开始，进入文件系统，直到存取文件存储器上的信息的实现。这一执行过程大致可以分成下列层次：用户接口、逻辑文件控制子系统、文件保护子系统、物理文件控制子系统和 I/O 控制子系统。

6.5.2.1 用户接口

接受用户发来的文件系统调用，进行必要的语法检查，根据用户对文件的存取要求，转换成统一的内部系统调用，并进入逻辑文件控制子系统。

6.5.2.2 逻辑文件控制子系统

根据文件名或文件路径名，建立或搜索文件目录，生成或找到相应文件目录项，把有关信息复制到活动文件表中，获得文件内部标识，供后面存取操作使用。此外，根据文件结构和取方法，把指定的逻辑记录地址换成相对物理块内相对地址。

6.5.2.3 文件保护子系统

根据活动文件表相应目录项识别调用者的身份，验证存取权限，判定本次文件操作的合法性。

6.5.2.4 物理文件控制子系统

根据活动文件表相应目录项中的物理结构信息，将相对块号及块内相对地址转换为文件存储器的物理块号和块内相对地址。本子系统还要负责文件存储空间的分配，若为写操作，则动态地为调用者申请物理块；实现缓冲区信息管理。根据物理块号生成 I/O 控制系统的调用形式。

6.5.2.5 I/O控制系统

具体执行 I/O 操作，实现文件信息的存取。这一层属于设备管理功能。

6.5.3 辅存空间管理

磁盘等大容量辅存空间操作系统及许多用户共享，用户作业运行期间常常要建立和删除文件，操作系统应能自动管理和控制辅存空间。辅存空间的有效分配和释放是文件系统应解决的一个重要问题。

辅存空间的分配和释放算法是较为简单的，最初，整个存储空间可连续分配给文件使用，但随着用户文件不断建立和撤销，文件存储空间会出现许多‘碎片’收集。在收集过程中，往往集过程中，往往对文件重新组织，让其存放到连续存储区中。

辅存空间分配常采用以下两种办法。

- 连续分配：文件被存放在辅存空间连续存储区中，在建立文件时，用户必须给出文件大小，然后，查找到能满足的连续存储区供使用；否则文件不能建立，用户进程必须等待。连续分配的优点是文件查找速度快，管理较为简单，但为了获得足够大的连续存储区。需定时进行‘碎片’收集。因而，不适宜于文件频繁进行动态扩充和缩小的情况，用户事先不知道文件长度也无法进行分配。
- 非连续分配：一种非连续分配方法是以块（或扇区）为单位，按文件动态要求分配给它若干扇区，这些扇区不一定要连续，属于同一文件的扇区按文件记录的逻辑次序用链指针连接或用位示图指示。另一种非连续分配方法是以簇为单位，簇是由若干个连续扇区组成的分配单位；实质上是连续分配和非连续分配的结合。各个簇可以用链指针、索引表，位示图来管理。非连续分配的优点是辅存空间管理效率高，访问文件执行速度快，特别是以簇为单位的分配方法已被广泛使用。

下面介绍常用的几种具体辅存空间管理方法。

6.5.3.1 字位映象表（位示图）

字位映象表使用若干字节构成一张表，每一位对应一个物理块，’1’状态表示相应块已占用，’0’状态表示该块空闲，微型机操作系统 CP/M 和 IBM 操作系统 VM/SP 等均使用这种技术管理存储空间。其主要优点是，它可以全部或部分保存在主存中，故可实现高速分配。

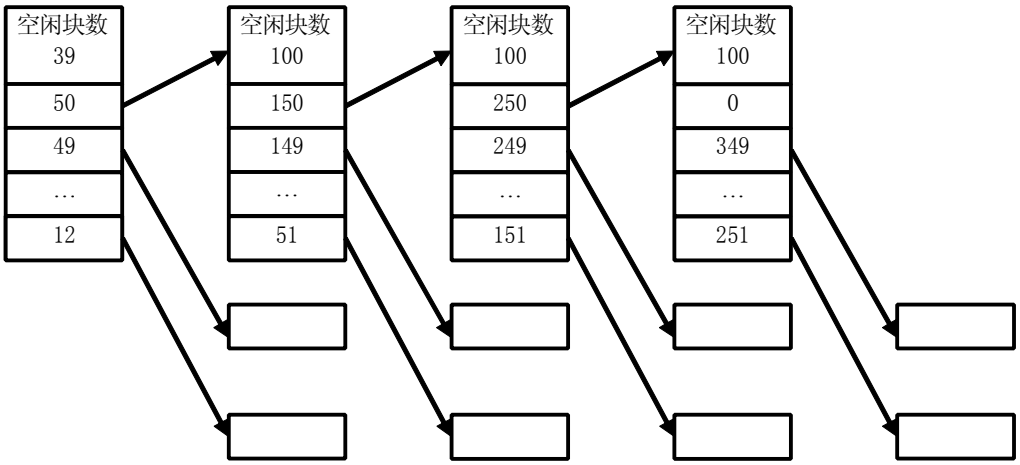
6.5.3.2 空闲区表

另一种分配方法常常用于连续文件。将空闲的储块的位置及其连续空闲的块数构成一张表。分配时，系统依次扫描空闲区表，寻找合适的空闲块并修改登记项。删除文件释放空闲区时，把空闲块位置及连续的空闲块数填入空闲区表，如果出现邻接的空闲块，还需执行合并操作并修改登记项。空闲区表的搜索算法有优先适应、最佳适应和最坏适应算法，这些已经在第六章中介绍过。

6.5.3.3 空闲块链

另一种管理方法是把所有空闲块连接在一起，系统保持一个指针指向第一个空闲块，每一空闲块包含有指向下一空闲块的指针。申请一块时，从链头取一块并修改系统指针； 删除时释放占用块使其成为空闲块并将它挂到空闲链上。这种方法效率很低，每申请一块都要读出空闲块并取得指针。

图 6-11 给出了 Unix 采用的是空闲块成组连接法。存储空间分成 512 字节一块。为讨论方便起见，假定文件卷启用时共有可用文件 438 块，编号从 12 至 349。每 100 块划分一组，每组第一块登记下一组空闲块的盘物理块号和空闲总数。图 8-13 给出了 Unix 系统中空闲块成组连接示意图。其中，50#-12#—组中，50#物理块中登记了一下 100 个空闲块物理块号 150#-51#。同样下一组的最后一块 150#中登记了再下一组 100 个空闲块物理号 250#-151#。注意，最后一组中，即 250#块中第 1 项是 0，作为标志，表明系统文件空闲块链已经结束。



(磁盘)专用块←→(内存)专用块	
分配算法 IF 空闲块数=1 THEN IF 第一个单元=0 THEN 等待 ELSE 复制第一个单元对应块到专用块，并分配之 ELSE 分配第(空闲块数)个单元对应块，空闲块数减 1	归还算法 IF 空闲块数<100 THEN 专用块的空闲块数加一，第(空闲块数)个单元置归还块号 ELSE 复制专用块到归还块，专用块的空闲块数置一，第一单元置归还块号

图 6-11　Unix 系统的空闲块成组连接示意图

当设备安装完毕，系统就将专用块复制到主存中。专用块指示的空闲块分配完后再有申请要求，就把下一组空闲块数及盘物理块号复制到专用块中申请要求，就把下一组空闲块数用盘物理块号复制到专用块中重复进行。搜索到全 0 块时，系统应向操作员发出警告，表明空闲块已经用完。需要注意，开始时空闲块是按顺序排列的，但只要符合分组及组间连接原则，空闲块可按任意次序排列。事实上，经过若干次分配，释放操作后，空闲块物理块号必定不能按序排列了。

6.6 实例研究：Linux 的文件管理

6.6.1 Linux 文件管理概述

Linux 支持多种不同类型的文件系统，包括 EXT、EXT2、MINIX、UMSDOS、NCP、ISO9660、HPFS、MSDOS、NTFS、XIA、VFAT、PROC、NFS、SMB、SYSV、AFFS 以及 UFS 等。由于每一种文件系统都有自己的组织结构和文件操作函数，并且相互之间的差别很大，从而给 Linux 文件系统的实现带来了一定的难度。

为支持上述的各种文件系统，Linux 在实现文件系统时借助了虚拟文件系统 VFS。VFS 只存在于内存中，在系统启动时产生，并随着系统的关闭而注销。它的作用是屏蔽各类文件系统的差异，给用户、应用程序和 Linux 的其他管理模块提供一个统一的接口。管理 VFS 数据结构的组成部分主要包括超级块和 inode。

Linux 的文件操作面向外存空间，它采用缓冲技术和 hash 表来解决外存与内存在 I/O 速度上差异。

在众多的文件系统类型中，EXT2 是 Linux 自行设计的具有较高效率的一种文件系统类型，它建立在超级块、块组、inode、目录项等结构的基础上，并在内存映射。

6.6.2 Linux 文件系统的管理

同其他操作系统一样，Linux 支持多个物理硬盘，每个物理磁盘可以划分为一个或多个磁盘分区，在每个磁盘分区上就可以建立一个文件系统。

Linux 中，一个文件系统在物理数据组织上一般划分成引导块、超级块、inode 区以及数据区。引导块位于文件系统开头，通常为一个扇区，存放引导程序、用于读入并启动操作系统。超级块由于记录文件系统的管理信息，根据特定文件系统的需要超级块中存储的信息不同。inode 区用于登记每个文件的目录项，第一个 inode 是该文件系统的根节点。数据区则存放文件数据或一些管理数据。

一个安装好的 Linux 操作系统究竟支持几种不同类型的文件系统，是通过文件系统类型注册链表来描述的。向系统注册文件系统类型有两种途径，一是在编译操作系统内核时确定，并在系统初始化时通过函数调用向注册表登记；另一种是把文件系统当作一个模块，通过 kerneld 或 insmod 命令在装入该文件系统模块时向注册表登记它的类型。

图 6-12 给出了文件系统注册表的数据结构，file_systems 指向文件系统注册表，每一个文件系统类型在注册表中有一个登记项，记录了该文件系统类型的名 name、支持该文件系统的设备 requires_dev、读出该文件系统在外存超级块的函数 read_super、以及注册表的链表指针 next。函数 register_filesystem 用于注册一个文件系统类型，函数 unregister_filesystem 用于从注册表中卸装一个文件系统类型。

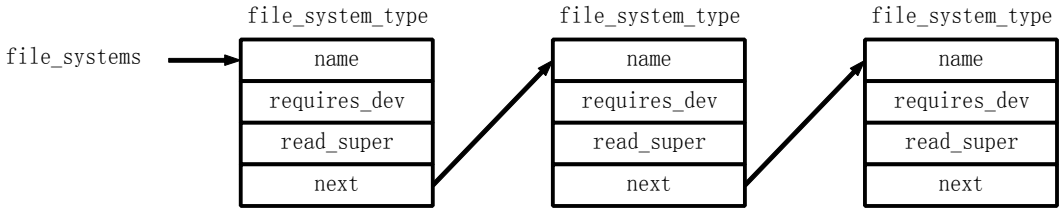


图 6-12　Linux 文件系统注册表的数据结构

每一个具体的文件系统不仅包括文件和数据，还包括文件系统本身的树形目录结构，以及子目录、链接 link、访问权限等信息，它还必须保证数据的安全性和可靠性。

Linux 操作系统不通过设备标识访问某个具体文件系统，而是通过 mount 命令把它安装到整个文件系统树的某一个目录节点，该文件系统的所有文件和子目录就是该目录的文件和子目录，直到用 umount 命令显式的撤卸该文件系统。

当 Linux 自举时，首先装入根文件系统，然后根据/etc/fstab 中的登记项使用 mount 命令自动逐个安装文件系统。此外用户也可以显式地通过 mount 和 umount 命令安装和卸装文件系统。

当装入/卸装一个文件系统时，应函数 add_vfsmnt/remove_vfsmnt 向操作系统注册/注销该文件系统。另外，函数 lookup_vfsmnt 用于检查注册的文件系统。

执行文件系统的注册和注销操作时，将在以 vfsmntlist 为链表头和 vfsmnttail 为链表尾的单项链表中增加或删除一个 vfsmount 节点，具体数据结构如下：

```
static struct vfsmount vfsmntlist = (static struct vfsmount )NULL;          /* 头 */
static struct vfsmount *vfsmnttail = (static struct vfsmount *)NULL;        /* 尾 */
static struct vfsmount *mru_vfsmnt = (static struct vfsmount *)NULL;    /*当前*/

struct vfsmount {
   kdev_t    mnt_dev;                                    /* 文件系统所在的主次设备号 */
   char*    mnt_devname;                                /* 文件系统所在的设备名，如/dev/hda1 */
   char*    mnt_dirname;                                /* 安装目录名 */
   unsigned int    mnt_flags;                          /* 设备标志，如 ro */
   struct semaphore    mnt_sem;                        /* 设备有关的信号量 */
   struct super_block*    mnt_sb;                  /* 指向超级块 */
   struct file*    mnt_quotas[MAXQUOTAS];          /* 指向配额文件的指针 */
   time_t    mnt_iexp[MAXQUOTAS];          /* expiretime for inodes */
   time_t    mnt_bexp[MAXQUOTAS];          /* expiretime for blocks */
   struct vfsmount* mnt_next;                        /* 后继指针 */
};
```

超级用户安装一个文件系统的命令格式是：
mount 参数 文件系统类型 文件系统设备名 文件系统安装目录
文件管理接收 mount 命令的处理过程是：

步骤 1：　如果文件系统类型注册表中存在对应的文件系统类型，转步骤 3。
步骤 2：　如果文件系统类型不合法，
　　则出错返回；

否则在文件系统类型注册表注册对应的文件系统类型。

步骤 3： 如果该文件系统对应的物理设备不存在或已经被安装，则出错返回。

步骤 4： 如果文件系统安装目录不存在或已经安装有其他文件系统，则出错返回。

步骤 5： 向内存超级块数组 super_blocks[]申请一个空闲的内存超级块。

步骤 6： 调用文件系统类型节点提供的 read_super 函数读入安装文件系统的外存超级块，写入内存超级块。

步骤 7： 申请一个 vfsmount 节点，填充正确内容后，假如文件系统注册表。

在使用 umount 卸装文件系统时，首先必须检查文件系统是否正在被其他进程使用，若正在被使用 umount 操作必须等待，否则可以把内存超级块写回外存，并在文件系统注册表中删除相应节点。

6.6.3 虚拟文件系统 VFS

虚拟文件系统 VFS 是物理文件系统与服务之间的一个接口层，它对每一个具体的文件系统的所有细节进行抽象，使得 Linux 用户能够用同一个接口使用不同的文件系统。VFS 只是一种存在于内存的文件系统，在操作系统自举时建立，在系统关闭时消亡。它的主要功能包括：

- 记录可用的文件系统的类型。
- 把设备与对应的文件系统联系起来。
- 处理一些面向文件的通用操作。
- 涉及到针对具体文件系统的操作时，把它们映射到与控制文件、目录以及 inode 相关的物理文件系统。

在引入了 VFS 后，Linux 文件管理的实现层次如图 6-13。

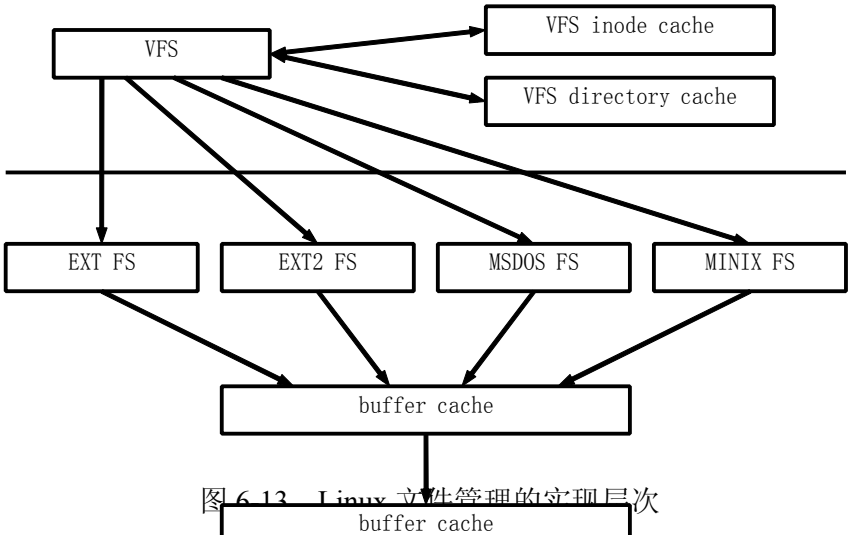


图 6-13 Linux 文件管理的实现层次

VFS 描述文件时使用超级块和 inode 的方式。当系统初启时，所有被初始化的文件系统类型都要向 VFS 登记。每种文件系统类型的读超级块函数 read_super 必须识别该文件系统的结构，并且把信息映射到一个 VFS 超级块数据结构上。超级块的数据结构如下：

```
struct super_block {
    kdev_t    s_dev;                    /* 该文件系统的主次设备号 */
    unsigned long    s_blocksize;       /* 块大小 */
    unsigned char    s_blocksize_bits;  /* 以 2 的幂次表示块大小 */
    unsigned char    s_lock;            /* 锁定标志，置位表示拒绝其他进程访问 */
    unsigned char    s_rd_only;         /* 只读标志 */
    unsigned char    s_dirt;            /* 已修改标志 */
    struct file_system_type* s_type;    /* 指向文件系统类型注册表相应项 */
    struct super_operations* s_op;      /* 指向一组操作该文件系统的函数 */
    struct dquot_operations* dq_op;
    unsigned long    s_flags;
```

```
    unsigned long    s_magic;
    unsigned long    s_time;
    struct inode*    s_covered;              /* 指向安装点目录的 inode */
    struct inode*    s_mounted;              /* 指向被安装文件系统的第一个 inode */
    struct wait_queue*    s_wait;            /* 在该超级块上的等待队列 */
    union { 各个物理文件系统超级块的结构类型 }  u;
};
```

值得指出的是，物理文件系统上超级块必须驻留在内存中，具体来说，就是利用 super_block.u 来存储具体的超级块。

文件系统中的每一个子目录和文件读对应于一个唯一的 inode，它是 Linux 管理文件系统的最基本单位。VFS inode 只存在于内存中的 inode_cache，其内容来自于物理文件系统，并有文件系统指定的操作函数填写。inode 的数据结构如下：

```
struct inode {
    kdev_t    i_dev;                              /* 该文件系统的主次设备号 */
    umode_t    i_mode;                           /* 文件类型以及存取权限 */
    ulink_t    i_nlink;                          /* 连接到该文件的 link 数 */
    uid_t    i_uid;
    gid_t    i_gid;
    kdev_t    i_rdev;                           /* 该文件系统的主次设备号 */
    off_t    i_size;                             /* 文件长度 */
    time_t    i_atime，  i_mtime，  i_ctime;
    unsigned long    i_blocksize，  i_blocks;   /* 字节/块为单位的文件长度 */
    unsigned long    i_version;
    unsigned long    i_nrpages;                /* 文件所占的内存页数 */
    struct semaphore    i_sem;
    struct inode_operations* i_op;/* 指向一组针对该文件的操作函数 */
    struct super_block*    i_sb;               /* 指向内存中的 VFS 超级块 */
    struct wait_queue*    i_wait;              /* 在该文件上的等待队列 */
    struct file_lock*    i_flock;              /* 操作该文件的文件锁链表的首地址 */
    struct vm_area_struct*    i_mmap;
    struct page*    i_pages;                   /* 文件所占页面构成的单向链 */
    struct dquot*    i_dquot[MAXQUOTAS];
    struct inode *i_next，*i_prev，  *i_hash_next，*i_hash_prev，*i_bound_to，*i_bound_by;
    struct inode *i_mount;                      /* 指向下挂文件系统的 inode 的根目录 */
    unsigned long    i_count;                   /* 引用计数，0 表示空闲 */
    unsigned short    i_flags;
    unsigned short    i_writecount;
    unsigned char    i_lock;                     /* inode 的锁定标志 */
    unsigned char    i_dirt;                     /* 已修改标志 */
    unsigned char    i_pipe，  i_sock，  i_seek，  i_update，  i_condemned;
    union { 各个物理文件系统 inode 的结构类型 }  u;
};
```

同超级块一样，inode.u 用于存储每一个特定文件系统的特定 inode。系统所有的 inode 通过 i_prev，i_next 连接成双向链表，头指针是 first_inode。每个 inode 通过 i_dev 和 i_ino 唯一地对应到某一个设备上的某一个文件或子目录。i_count 为 0 时表明该 inode 空闲，空闲的 inode 总是放在 first_inode 链表的前面，当没有空闲的 inode 时，VFS 会调用函数 grow_inodes 从系统内核空间申请一个页面，并将该页面分割成若干个空闲 inode，加入 first_inode 链表。围绕 first_inode 链，VFS 还提供一组操作函数，有兴趣的同学可以参考有关资料。

6.6.4 文件系统管理的缓冲机制

Linux 既支持多种类型的文件系统，又保持了很高的性能。探究其原因，除了 VFS 以外，多种复杂的 cache 起到了关键作用。

6.6.4.1 VFS inode cache

从效率角度出发，为提高对 first_inode 链表进行线性搜索的速度，VFS 为已经分配的 inode 构造了 cache 和 hash 表。图 6-14 给出了这一结构，VFS 访问 inode 时，它首先根据 hash 函数计算出 h，然后找到对应的 hash_table[h]指向的双向链表，通过 i_hash_next 和 i_hash_prev 进行查找。

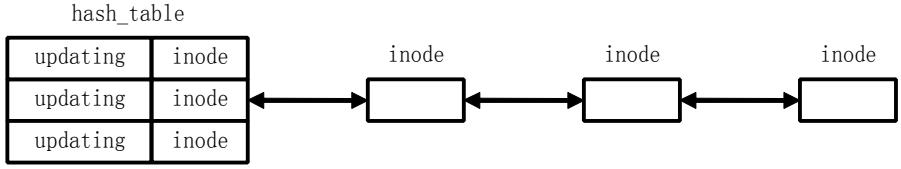


图 6-14 VFS inode cache 的结构

6.6.4.2 VFS directory cache

通过路径名访问文件时使用频率极高的操作，为提高此类操作的效率，Linux 维护了表达路径与 inode 对应关系的 VFS directory cache。被访问过的目录将会被存入 directory cache，这样当同一目录被再次访问时就可以快速获得。数据结构如下：

```
struct hash_list { struct dir_cache_entry *next,  *prev; };
struct dir_cache_entry {
   struct hash_list   h;
   kdev_t   dc_dev;
   unsigned long   dir;                          /* 父目录的 inode */
   unsigned long   version;
   unsigned long   ino;                        /* 本目录的 inode */
   unsigned char   name_len;
   char   name[DCACHE_NAME_LEN];
   struct dir_cache_entry   **lru_head;
   struct dir_cache_entry   *next_lru,  *prev_lru;
};
```

VFS directory cache 由 level1_cache 和 level2_cache 组成，头指针存放在 level1_head 和 level2_head 中。level1_cache 和 level2_cache 均采用 dir_cache_entry.next_lru 和 dir_cache_entry.prev_lru 指针构成包含 128 个节点的双向循环链表，使用 LRU 算法增删节点。level1_cache 和 level2_cache 各有分工，新增加的目录存放在 level1_cache 尾部；查询信息命中后则放在 level2_cache 尾部。为进一步提高效率，Linux 还为 level1_cache 和 level2_cache 建立了 hash 表。

6.6.4.3 Buffer cache

为加快对物理设备的访问，Linux 维护一组数据块缓冲区，称为 buffer cache。Buffer cache 就时文件组织中所提到的主存缓冲区，它独立于任何类型的文件系统，被所有的物理设备所共享。

Linux 用 struct buffer_head 类型封装数据缓冲区，进行缓冲区数据的读写操作。结构 buffer_head 给出了设备驱动程序需要的全部信息。操作系统把设备上的数据看作是等长数据块的线性列表，通过 buffer_head 的属性值唯一指明向什么设备写第几个数据块。对 buffer_head 感兴趣的同学可以参见有关资料。

为提高访问效率，Buffer cache 系统被精心设计成一个 hash 表和四类 buffer 链表：

- hash 表，用于进一步提高访问效率。

```
static struct buffer_head ** hash_table;
```

- 最近最少用链表，它包括 4 个链表。lru_list[0]保存数据已经写出的干净的缓冲区；lru_list[1]保存正在进行写出操作的缓冲区；lru_list[2]保存超级块和 inode 的缓冲区；lru_list[3]保存已有新数据但尚未安排写出的缓冲区。不难看出，此类链表的 buffer_head 节点都封装了数据缓冲区。

```
static struct buffer_head *lru_list[NR_LIST];
```

- 空闲链表，它按照 buffer 长度（512、1024、2048、4096、8192 字节）分成 5 个链表，其中的 buffer_head 节点封装了数据缓冲区。

```
static struct buffer_head *free_list[NR_SIZES];
```

- 未使用链表，buffer_head 节点未封装数据缓冲区。

```
static struct buffer_head *unused_list;
```

- 重用链表，buffer_head 节点未封装数据缓冲区。

```
static struct buffer_head *reuse_list;
```

6.6.5 系统打开文件表

系统打开文件表记录系统已经打开的文件，用于文件的读写操作。系统打开文件表是一张以 file 结构作为节点的双向链表，表头指针为 first_file，每个节点对应一个已打开的文件，包含了此文件的 inode、操作函数，对文件的所有操作都离不开它。数据结构如下：

```
struct file {
    mode_t    f_mode;
    loff_t    f_pos;
    unsigned short    f_flags;
    unsigned short    f_count;
    off_t    f_reada;
    struct file    *f_next，    *f_prev;
    int    f_owner;
    struct inode    *f_inode;
    struct file_operations    *f_op;
    unsigned long    f_version;
    void    *private_data;
};
```

对于每一个进程，PCB 包含一个打开文件表，结构如下：

```
struct files_struct {
    int    count;                          /* 引用计数器 */
    fd_set    close_on_exec;               /* 系统调用 exec 时关闭的文件的屏蔽字数组 */
    fd_set    open_fds;                    /* 对所有文件描述字 fd 的屏蔽字数组 */
    struct file    *fd[NR_OPEN];           /* 进程打开文件数组 */
};
```

Linux 引入 file、file_struct 结构后，可以通过两种途径共享文件，一是多个进程共享同一个 file 结构，二是多个进程的 file 结构共享 inode。后者是通过文件的 link 机制实现的，在这种方式中，Linux 专门设计了文件锁，解决两个进程同时读写文件时的互斥访问问题。有兴趣的同学可以在 include/linux/fs.h 中找到文件锁的数据结构。

6.6.6 EXT2 文件系统

EXT（1992 年）和 EXT2（1994 年）是专门为 Linux 设计的可扩展的文件系统。在 EXT2 中，文件系统组织成数据块的序列，这些数据块的长度相同，块大小在创建时被固定下来。EXT2 把它所占用的磁盘逻辑分区划分为块组，每一个块组依次包括超级块、组描述符表、块位图、inode 位图、inode 表以及数据块。块位图集了本组各个数据块的使用情况；inode 位图则记录了 inode 表中 inode 的使用情况。inode 表保存了本组所有的 inode，inode 用于描述文件，一个 inode 对应一个文件和子目录，有一个唯一的 inode 号，并记录了文件在外存的位置、存取权限、修改时间、类型等信息。

6.6.6.1 EXT2的超级块

EXT2 的超级块用来描述目录和文件在磁盘上的静态分布，包括尺寸和结构。每个块组都有一个超级块，一般来说只有组 0 的超级块才被读入内存超级块，其它块组的超级块仅仅作为备份。EXT2 文件系统的超级块主要包括 inode 数量、块数量、保留块数量、空闲块数量、空闲 inode 数量、第一个数据块位置、块长度、片长度、每个块组块数、每个块组片数、每个块组 inode 数，以及安装时间、最后一次写时间、安装信息、文件系统状态信息、…、等等内容。具体的 EXT2 外存超级块和内存超级块数据结构参见 include/linux/ext2_fs.h 中的结构 ext2_super_block 和结构 ext2_sb_info。

6.6.6.2 EXT2的组描述符

每个块组都有一个组描述符，记录了该块组的块位图位置、inode 位图位置、inode 节点位置、空闲块数、inode 数、目录数等内容。具体的组描述符数据结构参见 include/linux/ext2_fs.h 中的结构 ext2_group_desc。

所有的组描述符一个接一个存放，构成了组描述附表。同超级块一样，组描述符表在每个块组中都有备份，这样，当文件系统崩溃时，可以用来恢复文件系统。

6.6.6.3 EXT2的inode

inode 用于描述文件，一个 inode 对应一个文件和子目录，有一个唯一的 inode 号，并记录了文件的类型及存取权限、用户和组标识、修改/访问/创建/删除时间、link 数、文件长度和占用块数、在外存的位置、以及其他控制信息。具体的数据结构参见 include/linux/ext2_fs.h 中的结构 ext2_inode。文件的类型及存取权限、在外存的位置等参见原理部分的实例。

6.6.6.4 EXT2的目录文件

目录是用来创建和保存对文件系统中的文件的存取路径的特殊文件，它是一个目录项的列表，其中头两项是标准目录项“.”（本目录）和“..”（父目录）。目录项的数据结构如下：

```
struct ext2_dir_entry {
    _u32    inode;                                  /* 该目录项的 inode 号 */
    _u16    rec_len;                                /* 目录项长度 */
    _u16    name_len;                               /* 文件名长度 */
    char    name[EXT2_NAME_LEN];        /* 文件名 */
```

};

6.7 实例研究：Windows 2000 文件系统

6.7.1 Windows 2000 文件系统概述

Windows 2000 支持传统的 FAT 文件系统，对 FAT 文件系统的支持起源于 DOS 系统，以后的 Windows3.x 系统和 Windows95 系统 FAT 文件系统。FAT 文件系统最初是针对相对较小容量的硬盘设计的，每个磁盘卷被划分为相同大小的分配单元——簇，每个簇包括相同多个扇区组成。文件系统使用一个 16 位宽的 FAT 表中来记录磁盘卷的分配状况，每个簇在 FAT 表中占用一个唯一的 16 位项。这个 16 位的 FAT 表目类似于位示图可以用来标识一个簇是否被占用，进一步，它是通过存储文件的下一个存储簇的编号来表示当前簇已经被占用，因此 FAT 表在作为位示图的同时还用来存储了每个连接物理文件的指针，我们不得不佩服它在设计上的精巧性。

FAT 文件系统在小容量磁盘下工作的很好，但是随着计算机外存储设备容量的迅速扩展，它出现了明显的不适应。不难看出，FAT 文件系统最多可以容纳 2^{16} 或 65536 个簇，由于 FAT 文件系统为自己保留了一些表目，事实上，单个 FAT 卷被限制为 65518 的簇。如果一个簇包括 16 个扇区的话，单个 FAT 卷的容量小于 1GB，显然如果继续扩展簇中包含扇区数，文件系统的零头/碎片将很多，浪费很大。

从 Windows95 OSR2(Windows97)开始，FAT 表被扩展到 32 位，从而形成了 FAT32 文件系统。显然，FAT32 文件系统解决了 FAT16 在文件系统容量上的问题，它可以支持 2 GB 以上的大硬盘分区，但是由于 FAT 表的大幅度扩充，造成了文件系统处理效率的下降，这一点广大使用者都明显地感觉得到。Windows98 操作系统也支持 FAT32，但与其同期的 Windows NT 则不支持 FAT32。基于 NT 构建的 Windows 2000 则支持 FAT32。

在扩充 FAT 文件系统的同时，Microsoft 的另一个操作系统产品 Windows NT 则开始提供一个全新的文件系统 NTFS。NTFS 除了克服 FAT16 在容量上的局限和 FAT32 在容量上的不足外，另外一个出发点是立足于设计一个服务器端适用的文件系统，显然，作为一个客户端计算机的文件系统， FAT 不适合于需要可恢复性、安全性、数据冗余和容错的非常重要的应程序。为了有效地支持服务器系统，Windows 2000 在 NT4 的基础上进一步扩充了 NTFS，这些扩展需要将 NT4 的 NTFS4 分区转化为一个已更改的在盘格式，这种格式被称为 NTFS 5。NTFS 具有以下的特性：

- 可恢复性：NTFS 提供了基于事务处理模式的文件系统恢复，并支持对重要文件系统信息的冗余存储，从而满足了用于可靠的数据存储和数据访问的要求。
- 安全性：NTFS 利用操作系统提供的对象模式和安全描述体来实现数据安全性。在 Windows2000 中，安全描述体(访问控制表或 ACL)只需存储一次就可在多个文件中引用，从而进一步节省磁盘空间。
- 文件加密：在 Windows2000 中，加密文件系统 EFS 与 NTFS 机密集成，允许在 NTFS 卷上存储加密文件。
- 数据冗余和容错：NTFS 借助于分层驱动程序模式提供容错磁盘，RAID 技术允许借助于磁盘镜像技术，或通过奇偶校验和跨磁盘写入来实现数据冗余和容错。
- 大磁盘和大文件：NTFS 采用 64 位分配簇，从而大大扩充了磁盘卷容量和文件长度。
- 多数据流：在 NTFS 中，每一个与文件有关的信息单元，如文件名、所有者、时间标记、数据内容，都可以作为文件对象的一个属性，并由一个流（stream）——简单的字节队列组成。
- 基于 Unicode 的文件名：NTFS 采用 16 位的 Unicode 字符来存储文件名、目录和卷，适用于各个国家与地区，每个文件名可以长达 255 个字符，并可以包括 Unicode 字符、空格和多个句点。
- 通用的索引机制：NTFS 的体系结构被组织成允许在一个磁盘卷中索引文件属性，从而可以有效地定位匹配各种标准文件。在 Windows 2000 中，这种索引机制被扩展到其他属性，如对象 ID、对属性(例如基于 OLE 上的复合文件)的本地支持，包括对这些属性的一般索引支持。属性作为 NTFS 流在本地存储，允许快速查询。
- 动态添加卷磁盘空间：在 Windows2000 中，增加了不需要重新引导就可以向 NTFS 卷中添加磁盘空间的功能。
- 动态坏簇重映射：可加载的 NTFS 容错驱动程序可以动态地恢复和保存坏扇区中的数据。
- 磁盘配额：在 Windows2000 中，NTFS 可以针对每个用户指定磁盘配额，从而提供限制使用磁盘存储器的能力。
- 稀疏文件：在 Windows2000 中，用户能够创建文件，并且在扩展这些文件时不需要分配磁盘空间就能将这些文件扩展为更大。另外，磁盘的分配将推迟至指定写入操作之后。
- 压缩技术：在 Windows2000 中，避免解压和再压缩在整个网络中传递的压缩文件数据，减少了服务器的 CPU 开销。
- 分布式链接跟踪：在 Windows2000 中，NTFS 支持文件或目录的唯一 ID 号的创建和指定，并

保留文件或目录的 ID 号。通过使用唯一的 ID 号，从而实现分布式链接跟踪。这一功能将改进当前的文件引用存储方式(例如，在 OLE 链接或桌面快捷方式中)。重命名目标文件的过程将中断与该文件的链接。重命名一个目录将中断所有此目录中的文件链接及此目录下所有文件和目录的链接。

- POSIX 支持：如支持区分大小写的文件名、链接命令、POSIX 时间标记等。在 Windows2000 中，还允许实现符号链接的重解析点，仲裁文件系统卷的装配点和远程存储“分层存储管理(HSM)”。

Windows 2000 还提供分布式文件服务。分布式文件系统(DFS)是用于 Windows 2000 服务器上的一个网络服务器组件，最初它是作为一个扩展层发售给 NT4 的，但是在功能上受到很多限制，在 Windows 2000，这些限制得到了修正。DFS 能够使用户更加容易地找到和管理网上的数据。使用 DFS，可以更加容易地创建一个单目录树，该目录树包括多文件服务器和组、部门或企业中的文件共享。另外，DFS 可以给予用户一个单一目录，这一目录能够覆盖大量文件服务器和文件共享，使用户能够很方便地通过“浏览”网络去找到所需要的数据和文件。浏览 DFS 目录是很容易的，因为不论文件服务器或文件共享的名称如何，系统都能够将 DFS 子目录指定为逻辑的、描述性的名称。

6.7.2 NTFS 的实现层次

在 Windows2000 中，NTFS 及其它文件系统都结合在 I/O 管理器中，采用分层的设备驱动程序实现的。

如图 6-15 所示，在 Windows2000 执行体的 I/O 管理器部分，包括了一组在核心态运行的可加载的与 NTFS 相关的设备驱动程序。这些驱动程序是分层次实现的，它们通过调用 I/O 管理器传递 I/O 请求给另外一个驱动程序，依靠 I/O 管理器作为媒介允许

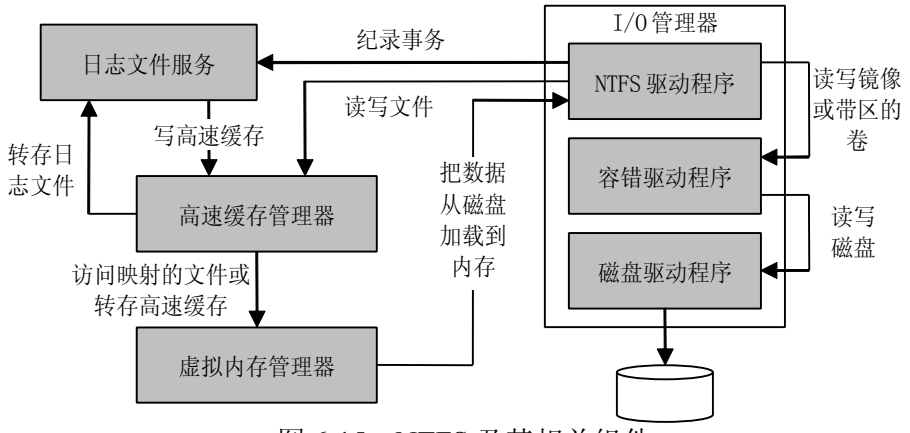


图 6-15　NTFS 及其相关组件

每个驱动程序保持独立，以便可以被加载或卸载而不影响其他驱动程序。

另外，图 6-15 还给出了 NTFS 驱动程序和与文件系统紧密相关的三个其他执行体的关系。

日志文件服务(LFS)是为维护磁盘写入的日志而提供服务的 NTFS 的一部分。此日志文件用于在系统失败时恢复 NTFS 的已格式化的卷。

高速缓存管理器是 Windows 2000 的执行体组件，它为 NTFS 以及包括网络文件系统驱动程序(服务器和重定向程序)的其他文件系统驱动程序提供系统范围的高速缓存服务。Windows2000 的所有文件系统通过把高速缓存文件映射到虚拟内存，然后访问虚拟内存来访问它们。为此，高速缓存管理器提供了一个特定的文件系统接口给 Windows2000 虚拟内存管理器。当程序试图访问没有加载到高速缓存的文件的一部分时(高速缓存遗漏)，内存管理器调用 NTFS 来访问磁盘驱动器并从磁盘上获得文件的内容。高速缓存管理器通过使用它的“延迟书写器”(lazy writer)来优化磁盘 I/O。延迟书写器是一组系统线程，它在后台活动，调用内存管理器来刷新高速缓存的内容到磁盘上(异步磁盘写入)。

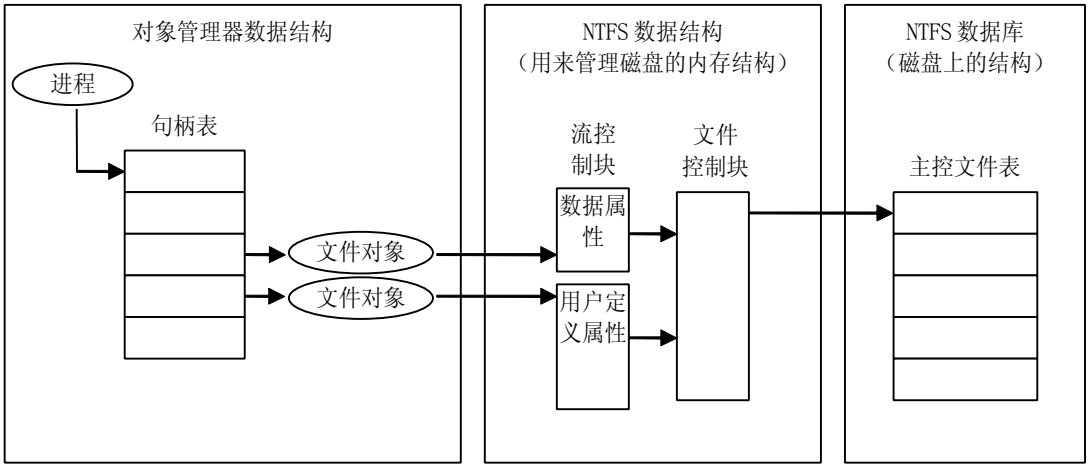
NTFS 通过实现把文件作为对象来参与 Windows2000 对象模式。这种实现方法允许文件被对象管理器共享和保护，对象管理器是管理所有执行体级别对象的 Windows2000 组件。应用程序创建和访问文件同对待其他 Windows2000 对象一样——依靠对象句柄。当 I/O 请求到达 NTFS 时，Windows 2000 对象管理器和安全系统已经验证该调用进程有权以它试图访问的方式来访问文件对象。安全系统把调用程序的访问令牌同文件对象的访问控制列表中的项进行比较。I/O 管理器也将文件句柄转换为指向文件对象的指针。NTFS 使用文件对象中的信息来访问磁盘上的文件。

图 6-16 显示了将文件句柄链接到文件系统在磁盘上结构的数据结构。

当 I/O 系统调用 NTFS 时，该句柄已经被转换成指向文件对象的指针。然后，NTFS 跟随几个指针从文件对象中得到文件在磁盘上的位置。一个代表打开文件系统服务的单一调用的文件对象，指向用于调用程序试图读取或写入文件属性的流控制块（Stream Control Block，SCB）。进程已同时打开数据属性和文件的用户定义的属性。SCB 代表单个的文件属性并包含有关怎样在文件中查找指定的属性的某些信息。文件的所有 SCB 都指向一个称作文件控制块（File Control Block，FCB）的公共数据结构。FCB 包括一个指针(实际上是一个文件引用，它指向基于磁盘的主控文件表（或 MFT）中文件的记录。

图 6-16　NTFS 数据结构

6.7.3 NTFS 在磁盘上的结构



物理磁盘可以组织成一个或多个卷。卷与磁盘逻辑分区有关，它随着 NTFS 格式化磁盘或磁盘的一部分而创建，其中镜像卷和容错卷可能跨越多个磁盘。NTFS 将分别处理每一个卷，同 FAT 一样，NTFS 的基本分配单位是簇，它包含整数个物理扇区。

NTFS 卷中存放的所有数据都包含在一个 NTFS 元数据文件中，包括定位和恢复文件的数据结构、引导程序数据和记录整个卷分配状态的位图。

主控文件表 MFT 是 NTFS 卷结构的中心。它使用文件记录数组来实现的。NTFS 忽略簇的大小，每个文件记录的大小都被固定为 1KB。从逻辑上讲，卷中的每个文件在 MFT 上都有一行，其中还包括 MFT 自己的一行。除了 MFT 以外，每个 NTFS 卷还包括一组“元数据文件”，其中包含用于实现文件系统结构的信息。每一个这样的 NTFS 元数据文件都有一个以美元符号($)开头的名称，虽然该符号是隐藏的。例如，MFT 的文件名为$MFT。NTFS 卷中的其余文件是正常的用户文件和目录，如图 6-17 所示。

通常情况下，每个 MFT 记录与不同的文件相对应。然而，如果一个文件有很多属性或分散成很多碎片，就可能需要不止一个文件记录。在此情况下，存放其他文件记录的位置的第一个记录就叫作“基文件记录”。

当 NTFS 首次访问某个卷时，它必须“装配”该卷——也就是说准备使用它。要装配该卷，NTFS 会查看引导文件，找到 MFT 的物理磁盘地址。MFT 自己的文件记录是表中的第一项；第二个文件记录指向位于磁盘中间的称作“MFT 镜像”的文件(文件名为$MFTMirr)，该文件包含有 MFT 前面几行的副本。如果因某种原因 MFT 文件不能读取时，这种 MFT 的部分副本就用于定位元数据文件。

图 6-17　MFT 中 NTFS 元数据文件的文件记录

一旦 NTFS 找到 MFT 的文件记录，它就从文件记录的数据属性中获得虚拟簇号 VCN 到逻辑簇号 LCN 映射信息，将其解压缩并存储在内存中。这个映射信息告诉 NTFS 组成 MFT 的运行构成放在磁盘的什么地方。然后，NTFS 再解压缩几个元数据文件的 MFT 记录，并打开这些文件。接着，NTFS 执行它的文件系统恢复操作。最后，NTFS 打开剩余的元数据文件。现在用户就可以访问该卷了。

系统运行时，NTFS 会向另一个重要的元数据文件——日志文件(log fi1e)(文件名为$LogFile)写入信息。NTFS 使用日志文件记录所有影响 NTFS 卷结构的操作，包括文件的创建或改变目录结构的任何命令，例如复制。日志文件被用来在系统失败后恢复 NTFS 卷。

MFT 中的另一项是为根目录(即\)保留的。它的文件记录包含一个存放于 NTFS 目录结构根部的文件和目录索引。当第一次请求 NTFS 打开一个文件时，它开始在根目录的文件记录中搜索这个文件。打开文件之后，NTFS 存储文件的 MFT 文件引用，以便当它在以后读写该文件时可以直接访问此文件的 MFT 记录。

NTFS 把卷的分配状态记录在位图文件(bitmap file)(文件名为$Bitmap)中。用于该位图文件的数据属性包含一个位图，它们中的每一位代表卷中的一簇，标识该簇是空闲的还是已被分配给了一个文件。

另一个重要的系统文件，引导文件(bootfile)(文件名为$Boot)，存储 Windows2000 的引导程序代码，为了引导系统，引导程序代码必须位于特定的磁盘地址。然而，在格式化期间，Format 实用程序通过为这个区域创建一个文件记录将它定义为一个文件。创建引导文件使得 NTFS 坚持将磁盘上的所有事物都看成文件的原则。此引导文件以及 NTFS 元数据文件可以通过应用于所有 Windows 2000 对象的安全描述体被分别地保护。使用这个“磁盘上的所有事物均为文件”模式也意味着虽然引导文件目前正被保护而不能编辑，但引导程序还是可以通过一般的文件 I/O 来修改。

NTFS 还保留了一个记录磁盘卷中所有损坏位置的“坏簇文件”(bad-cluster)(文件名为$BadClus)和一个“卷文件”(volume fi1e)(文件名为$volume)，卷文件包含卷名、被格式化的卷的 NTFS 版本和一个位，当设置此位时表明磁盘已经损坏，必须用 Chkdsk 实用程序来恢复。

最后，NTFS 保持一个包含属性定义表(attribute definition table)的文件(名为$AttrDef)，它定义了卷中支持的属性类型，并指出它们是否可以被索引，在系统恢复操作中是否可以恢复。

NTFS 卷中的文件是通过称为“文件引用”的 64 位值来标识的。文件引用由文件号和顺序号组成。文件号与文件在 MFT 中的文件记录的位置减 1 相对应(如果文件有多个文件记录，则对应于基文件记录的位置减 1)。文件引用的顺序号在每次重复使用 MFT 文件记录的位置时会被增加，它使得 NTFS 能完成内部的一致性检查。

NTFS 将文件作为许多属性/值对的集合来存储，其中的一个就是它包含的数据(称为未命名的数据属性)。组成文件的其他属性包括文件名、时间标记、安全描述体以及可能附加的命名数据属性。每个文件的属性在文件中以单独的字节流存储。严格地讲，NTFS 不读取也不写入文件——它只是读取和写入属性流。NTFS 提供了这些属性操作：创建、删除、读取(字节范围)以及写入(字节范围)。读取和写入服务一般是对文件的未命名属性的操作。然而，调用程序可以通过使用已命名的数据流句法来指定不同的数据属性。

NTFS 和 FAT 文件系统的文件名长度在 255 个字符以内。文件名可以包括 Unicode 字符、空格和多个句点。并可以映射到 DOS 和 POSIX 的名字空间。

如果文件很小，那么它所有的属性和值(例如，它的数据)就放在文件记录中。当属性值直接存放在 MFT 中时，该属性叫作常驻属性(resident attribute)。

每种属性以一个标准头开始，在头中包含有关属性信息和 NTFS 用一般方法管理属性所需的信息。这个头总是常驻的，它记录了属性值是常驻的还是非常驻的。对于常驻属性，头中还包含从头到属性值的偏移量和属性值长度

当一个属性值直接存放在 MFT 中时，NTFS 访问该值的时间将大大缩短。取代了原有的在表中查找一个文件，然后读出连续分配的单元以找到文件的数据(例如，像 FAT 文件系统那样)，NTFS 只需访问磁盘一次，就可立即重新得到数据。

当然，许多文件和目录不能压缩成 1KB 固定大小的 MFT 记录。如果一个特定的属性，例如文件的数据属性，由于它太大而不能包含在 MFT 文件记录中，则 NTFS 将在磁盘上分配一个与 MFT 分开的 2KB 的区域(对于具有 4KB 或更大簇的卷来说是 4KB)。这个区域叫作一个运行(run)(或者叫作一个区域(extent))，它用来存储属性值(例如，文件的数据)。如果属性值以后增加了(例如，用户向文件中追加了数据)，则 NTFS 将为额外的数据分配另一个运行。值存储在运行中而不是在 MFT 中的属性叫作非常驻属性(nonresident attributes)。文件系统决定了一个特定的属性是常驻的还是非常驻的；数据所在的位置对于要进行的访问操作来说是透明的。

当一个属性是非常驻属性时，对于大文件可能是数据属性，它的头包含 NTFS 需要在磁盘上查找属性值所必需的有关信息。

在标准属性中，只有可以增长的属性才可以是非常驻的。对于文件，可增长的属性是安全描述体、数据和属性列表。标准信息和文件名是常驻的。

一个大目录也可能包括非常驻属性(或属性的一部分)，当 MFT 文件记录没有足够的空间来存储构成大目录的文件索引。部分索引被存储在索引根属性中，其余的索引存储在叫作“索引缓冲区”(index buffers)的非常驻运行中。索引根、索引分配以及位图属性在此均使用简化形式表示。标准信息和文件名属性总

是常驻的。对目录来说，头和至少部分索引根属性值也是常驻的。

习题

1. 叙述下列术语的定义并说明它们之间的关系。卷、块、记录、文件。
2. 什么是记录的成组和分解操作?采用这种技术有什么优点?
3. 列举文件系统面向用户的主要功能。
4. 假定磁带的记录密度为每英寸 800 字符，每一逻辑记录长为 160 字符，块间隙为 0.6 英寸，现有 1000 个逻辑记录需要存储。分别计算不成组操作和块因子为 5 时，磁带介质的利用率。物理记录至少为多大时，才不致于浪费超过 50%的磁带存储空间?
5. 什么是文件的逻辑结构?它有哪几种组织方式?
6. 什么是文件的物理结构?它有哪几种组织方式?
7. 叙述各种文件物理组织方式的主要优缺点。
8. 什么是记录键?它有何用处?
9. 连接文件的连接字可以如下定义：

   连接字的内容为(上一块的地址)(下一块的地址)

   首块连接字内容为(下一块的地址)

   末块连接字内容为(上.一块的地址)

   其中，是模 2 按位加。试述这种连接字的主要特点。
10. 磁带卷上记录了若干文件，假定当前磁头停在第 j 个文件的文件头标前，现要按名读出文件 i，试给出读出文件的步骤。
11. Unix 采用空闲块成组连接的方式，试返出申请和归还一块的工作流程。
12. 文件系统提供的主要文件操作有哪些?叙述各自的主要功能。
13. 简述 Windows 2000 NTFS 文件系统的主要特点和优点。
14. 简述 Windows 2000 NTFS 文件系统的实现。
15. 假定令 B-物理块长、R=逻辑记录长、F=块因子。在下列两种情况下，给出计算 F 的公式：

    (1) 定长记录：一个块中有整数个逻辑记录

    (2) 变长记录：逻辑记录长可变，但逻辑记录不允许跨块
16. 为了快速访问，又易于更新，当数据为以下形式时，你选用何种文件组织方式。

    (1) 不经常更新，经常随机访问；

    (2) 经常更新，经常按一定顺序访问；

    (3) 经常更新，经常随机访问；
17. 一个操作系统采用树形结构的文件系统，但限制了树的深度，如只能有 3 层，这个限制对用户有何影响?这种文件系统如何设计?
18. 一些系统允许用户同时访问文件的一个拷贝来实现共享，另一些系统为每个用户提供一个共享文件拷贝，试讨论各自的缺点。
19. 试比较文件系统和主存存储空间分配的异同。
20. 对目录管理的主要要求是什么?
21. 目前采用广泛的是哪种目录结构?它有什么优点?
22. 试述 Hash 检索方法的优点和缺点?
23. 在 Hash 检索方法中，如何解决冲突?
24. 试说明树型目录结构中线性检索法的检索过程?

25. 有一计算系统采用图来理空闲盘块，设盘块大小为 1KB，现若申请 2 个盘块，试说明盘块分配的具体过程或画出分配的流程图。
26. 何谓管道连接?试举例说明。
27. 什么叫‘按名存取’?文件系统是如何实现按名存取文件的?
28. .文件目录在何时建立?它在文件管理中起什么作用?
29. 某操作系统的磁盘文件空间共有 500 块，试画出相应的位示图，并给出申请/归还一块的工作流程。
30. 若两个用户共享一个文件系统，用户甲使用到文件 A、B、C、D、E；用户乙要用到文件 A、D、E、F。已知用户甲的文件 A 与用户乙的文件 A 实际上不是同一文件；甲、乙两用户的文件正是同一文件。试设计一种文件系统组织方案，使得甲、乙两用户能共享该文件系统又不致造成混乱。

CH7 操作系统安全性

影响计算机系统安全性的因素很多。首先，操作系统是一个共享资源系统，支持多用户同时共享一套计算机系统的资源，有资源共享就需要有资源保护，涉及到种种安全性问题；其次，随着计算机网络的迅速发展，客户机要访问服务器，一台计算机要传送数据给另一台计算机，于是就需要有网络安全和数据信息的保护；另外，在应用系统中，主要依赖数据厍来存储大量信息，它是各个部门十分重要的一种资源，数据库中的数据会被广泛应用，特别是在网络环境中的数据库，这就提出了信息系统——数据库的安全性问题；最后计算机安全性中的一个特殊问题是计算机病毒，需要采用措施预防、发现、解除它。上述计算机安全性问题大部份要求操作系统来保证，所以操作系统的安全性是计算机系统安全性的基础。

按照 ISO 通过的“信息技术安全评价通用准则” 关于操作系统、数据库这类系统的安全等级从低到高分为七个级别：

- D　最低安全性
- C1　自主存取控制
- C2　较完善的自主存取控制、审计
- B1　强制存取控制
- B2　良好的结构化设计、形式化安全模型
- B3　全面的访问控制、可信恢复
- A1　形式化认证

目前流行的几个操作系统的安全性分别为：DOS：D 级；Windos NT 和 Saloris：C2 级；OSF/1：B1 级；Unix Ware 2.1：B2 级。

7.1 安全性概述

所有的文件都储存于计算机的共享设备上。这就意味着某人的文件存在着被别人读写的潜在可能性。有时这恰是人们所想达到的：把信息存于共享文件中让用户共用。但在其他情况下，用户又希望私有某些信息。那么操作系统如何建立一个用户可选择性的把信息保存为私有的或公有的环境呢？这就是操作系统中的保护和安全性功能要做的任务。而要在一个连接着其他计算机的网络中保密信息就更困难了。

现代计算机可支持多用户，或通过一台单机的时空复用或利用网络进入其他计算机。各组织机构依赖于计算机来储存各种各样的信息。诸如储存他们的工作状态以帮助他们管理好自己的组织，以及储存他们的财产和机密信息。计算机本身对该组织也代表了一种重要的资源，因此仅计算机的使用就是该组织涉及的开销。所以，该组织必须同时保护他们的计算机和他们的信息不被未授权用户使用，正如他们想保护他们其他诸如建筑、设备、财政基金等其他资源一样。

软件的易变性带来了一个保护资源不被未授权用户使用的困境。理论上，在计算机界中，软件的功能仅受限于在这些软件上的智力投资。这说明存在编写可破坏任何操作系统安全方案的复杂软件的可能。对操作系统设计者来说，挑战在于现在就建立可预防任何未来软件侵入的安全方案。一般来讲，如果没有硬件的特殊支持很难做到这一点。

计算机网络加剧了这种问题，因为它允许在不同地点的人们进入异地的计算机。象关上门这些物理方法根本构不成任何侵入计算机的阻碍。

从信息安全性的角度出发，安全性的实现包括下面几个层次。

7.1.1 操作系统的分级安全管理

7.1.1.1 系统级安全管理

系统级安全管理的任务是不允许未经核准的用户进入系统，从而也就防止了他人非法使用系统的资源。主要采用的手段有：

- 注册　系统设置一张注册表，登录了注册用户名和口令等信息，使系统管理员能掌握进入系统的用户的情况，并保证用户各在系统中的唯一性。
- 登录　　用户每次使用时，都要进行登录，通过核对用户名和口令，核查该用户的合法性。同时也可根据用户占用资源情况进行收费。

口令很容易泄密，可要求用户定期修改口令，以进一步保证系统的安全性。

7.1.1.2 用户级安全管理

用户级安全管理，是为了给用户文件分配文件“访问权限”而设计的。用户对文件访问权限的大小，是根据用户分类、需求和文件属性来分配的。例如，Unix 中，将用户分成三类：文件主、授权用户和一般用户。

已经在系统中登录过的用户都具有指定的文件访问权限，访问权限决定了用户对哪些文件能执行哪

些操作。当对某用户赋予其访问指定目录的权限时，他便具有了对该目录下的所有子目录和文件的访问权。通常，对文件可以定义的访问权限有：建立、删除、打开、读、写、查询和修改。

7.1.1.3 文件级安全管理

文件级安全性是通过系统管理员或文件主对文件属性的设置，来控制用户对文件的访问。通常可对文件置以下属性：执行、隐含、修改、索引、只读、写 、共享等。

7.1.2 通信网络安全管理

就信息存储、处理和传输三个主要操作而言，信息在传输过程中受到的安全威胁最大。计算机网络的实体防护最为薄弱，利用通信或远程终端作案易于实现，因此计算机通信网络的安全性备受关注。对网络安全的威胁主要表现在：非授权访问、冒充合法用户、破坏数据完整性、干扰系统正常运行、利用网络传播病毒、线路窃听等方面。

由于网络的安全比独立计算机系统复杂得多，因而网络操作系统必须采用多种安全措施和手段，其主要有：

- 用户身份验证和对等实体鉴别：远程录入用户的口令应当加密，密钥必须每次变更以防被人截获后冒名顶替。网络环境下，一个用户向另一个用户发送数据，发主必须鉴别收方是否确定是他要发给信息的人，收方也必须判别所发来的信息是否确定是由发送者个人发来，这就是对等体鉴别。
- 访问控制：除了网络中主机上要有存取访问控制外，应当将访问控制扩展到通信子网，应对哪些网络用户可访问哪些本地资源，以及哪些本地用户可访问哪些网络资源进行控制。
- 数据完整性：防止信息的非法重发，以及传送过程中的篡改、替换、删除等，要保证数据由一台主机送出，经网络链络到达另一台主机时完全相同。
- 加密：加密后的信息即使被人截取，也不易被人读懂和了解，这是存取访问控制的补充手段。
- 防抵赖：防止收发信息双方抵赖纠纷。收方收到信息，他要确保发方不能否认曾向他发过信息，并要确保发方不否认收方收到的信息是未被篡改过的原样信息。发方也会要求收方不能在收到信息后抵赖或否认。
- 审计：审计用户对本地主机的使用，还应审计网络运行情况。

通常网络系统安全保障的实现方法分两大类：一类是以防火墙技术为代表的防卫型网络安全保障系统。它是通过对网络作拓扑结果和服务类型上进行隔离，在网络边界上建立相应的网络通信监控系统，来达到保障网络安全的目的。实现防火墙所用的主要技术有：数据包过滤、应用网关和代理服务器(Proxy Server)等。另一类是建立在数据加密和用户授权确认机制上的开放型网络安全保障系统。这类技术的特征是利用数据加密技术来保护网络系统中包括用户数据在内的所有数据流，只有指定用户或网络设备才能解译加密数据，从而在不对网络环境作特殊要求的前提下从根本上解决网络安全性问题。数据加密技术可分为三类：对称型加密、不对称型加密和不可逆加密。对称型加密使用单个密钥对数据进行加密或解密，计算量小、加密效率高，但密钥管理困难，使用成本高，保安性能差。不对称加密也称公用密钥算法，它采用公用和私有二个密钥，只有二者搭配使用才能完成加解密过程。不可逆加密算法的特征是加密过程不需要密钥，经过加密的数据无法被解密，只有同样的输入数据，经过同样的不可逆加密算法才能得到相同的加密数据。但其加密计算工作量大，仅适用于数据量有限的场合。

7.1.3 数据库安全管理

又称数据库安全管理，数据库内汇集了大量应用信息，并被广泛使用，在网络环境下，给数据库的安全性带来许多新问题。

信息安全是指保护信息以防止未授权者对信息的恶意访问、泄漏、修改和破坏，从而导致信息的不可靠或被破坏。它可以用 CIA 来表示，机密性（Confidentiality）定义了哪些系统资源不能被未授权用户访问；完整性（Integrity）决定了信息不能被未授权的来源所替代或遭到改变和破坏；可用性（Availability）防止非法独占资源，每当用户需要并有权访问时，总能访问到所需的信息资源。

信息系统的安全性可用 4A 的完善程度来衡量，即用户身份验证（Authentication）、授权（Authorization）、审计（Audit）和保证（Assurance）。用户身份验证是指在用户获取信息、访问系统资源之前对其身份的标识进行确定和验证，以保证用户自身的合法性；授权是指使不同的用户能用各自的权限合法地访问他们可使用的信息及系统资源；审计是对各种安全性事件的检查、跟踪和记录；保证的作用是在意外故障乃至灾难中信息资源不被破坏与丢失。

7.1.4 预防、发现和消除计算机病毒

计算机病毒是一个能够通过修改程序，并把自身的复制品包括在内去“传染” 其它程序的一种程序。自从 1983 年，美国专家发现并验证存在计算机病毒(Computer Virus) 后，计算机病毒已在全世界蔓延，发现的病毒有数千种，给人类社会、政治、经济等各方面带来巨大危害。计算机病毒的出现并非偶然，它是计算机技术和以计算机为核心的社会信息化进程发展到一定阶段的产物；同时在相当大的程度上反

映了当今计算机系统的脆弱性。除了在技术上需要研究和改进计算机系统的抗病毒能力外，各国政府正在制定法规，把编制和散布计算机病毒定为犯罪行为。

计算病毒具有破坏性、隐蔽性、传染性、和表现性等特性。计算机病毒按其寄生方式可分成源码病毒、入侵病毒、外壳病毒、系统病毒等四种。计算机病毒的防治不外乎三个方面：一是病毒的预防，指采取措施保护传染对象不受病毒的传染；二是病毒的发现，指不能有效预防病毒入侵时，应该尽早根据计算机系统中产生的种种蛛丝马迹发现病毒的存在，以便消除它；三是病毒的消除，有专门的杀毒工具，如 Vsafe、MSAV、Kill 等，用来杀毒和解毒，使系统恢复正常。

7.2 安全性和保护的基本机制

7.2.1 策略与机制

一个组织的安全策略定义了一组用于授权使用其计算机及信息资源的规则。例如，某组织只允许财务部门的职员使用存储了财务信息的计算机。那么，如何在一个暂时的计算机系统中定义和执行组织的安全策略呢？计算机保护机制是实施组织安全策略的工具，同型号的计算机（操作系统相同）可拥有不同的安全策略，即使他们采用了相同的保护机制。怎样的保护机制才适合支持策略？什么才是合理的定义问题的方法以达到其可被域内的计算机软硬件解决的目的？这些都是关系到计算机系统保护和安全问题的例子。

在操作系统中要强调机制和策略的区别。机制是用于为实现任何其他不同种类策略提供工具的组件的集合。而策略则是一则特定的指定实现某项确定目标方法的策略。例如，除了交换信息外，某种特殊的通讯策略不允许两个进程分享资源。支持此策略的通讯机制就需要支持消息传送，可能是通过把一个进程地址空间内的信息复制到另一进程的方法。在另一种情况下，某必须的页面机制可能支持不同的替换算法策略。一个系统的安全策略制订了对本组织人员和非本组织人员资源的共享方式。机制是系统提供用于强制执行策略的特定步骤和工具。

举个例子，某学院的计算机系可能有这样一条策略：即本科生实验室中的计算机只能给已注册的计算机班的本科生使用。支持此策略的机制将需要用户的学生证和计算机系的班级列表来实行策略。此机制还必须由其他诸如授权验证、实验室的设备来补充完成。建立一条精确的策略是困难的，因为它需要既制订一套准确的软件，又制订一套无任何漏洞的“法令”以控制人类用户的活动。

根据经验（不考虑执行性能），当需要保证执行它们预期中的函数时，策略就被执行了。如果机制不能被确保按定义工作，那将不能依赖此机制来实行策略。在现代操作系统中，安全的保护机制之可以在操作系统内执行。

策略由计算机管理员选定，一般都是在操作系统和其保护机制设计及实现很久以后。理论上，安全策略可通过在用户空间内定义它而决定，虽然许多操作系统流于函数应用层。操作系统的很小一部分用于实现机制，而其他部分---系统软件、应用软件决定了策略。结果造成了通常的保护和安全方面的研究先依赖于操作系统机制的设计，然后才是系统设计者和管理员选定的策略。

保护机制实现了身份鉴别函数来使策略可验证作为某个实体的远程用户或计算机是否确实是其宣称的那样。保护机制还被用于检查某实体是否拥有访问某些资源的权限。

7.2.2 身份鉴别机制

身份鉴别机制是大多数保护机制的基础。身份鉴别分为内部和外部身份鉴别两种。外部身份鉴别涉及验证某用户是否是其宣称的。例如，某用户用一个用户名登录了某系统，此系统的基本外部身份鉴别机制将进行检查以证实此用户的登录确实是预想中拥有此用户名的用户。最简单的外部验证是赋予每个账号一个口令，账号可能是广为人知的，例如，它可能被用作一个电子邮件地址，而口令则对使用此账号的人员保密，此口令作为一个实体只能被此账号的拥有者或系统管理员改变。操作系统机制支持这种验证来确保不存在通过某些隐蔽的方式绕过验证机制的可能。

内部身份鉴别机制确保某进程不能表现为除了它自身以外的进程。若没有内部验证，某用户可以创建一个看上去属于另一用户的进程。从而，即使是最高效的外部验证机制也会因为把这个用户的伪造进程看成另一个合法用户的进程而被轻易地绕过。

让我们来考察一下在 UNIX 工作站网络中创建一套安全机制的困难性。开始，分时系统的主机被放置在安全可靠的机房中，用户通过与主机及控制台无物理连接的通讯线路登录到主机上。这种情形在工作站进入 UNIX 系统后有了变化。今天的工作站物理上是放在用户的工作区，操作台已不是在安全区域的独立终端而是物理显示器上的一个窗口。早期分时系统的物理安全性在今天的 UNIX 工作站中以不复存在了。工作站网络一般是由一个中心组织管理，而工作站的“拥有者”是普通用户，他用逻辑意义上的操作台不享有任何特殊管理权限地登录主机。通过给系统管理员提供权限登录号，整个系统可实现远程管理而不许本地用户修改系统文件。然而，设想用户关上了机器的电源然后又打开它，在这种情况下，传统 UNIX 工作站设计成了在单用户模式下启动而控制台在根模式，因此，任何想拥有根权限的用户都

可以用开关电源使机器处于单用户模式，随心所欲地切换许可，然后启动操作系统到多用户模式而得到根权限。这个缺陷很快得到证实并通过使机器必须以多用户模式启动而得到补救。这个例子说明了人们不能依赖于简单的诸如“此操作系统以被证实是可靠的，那么此系统亦即可靠”这类假设。

7.2.3 授权机制

授权机制确认用户或进程只有在策略许可某种使用时才能够使用计算机的实体（例如资源）。授权机制依赖于安全的验证机制的存在。图 1 显示了经典的计算机系统授权访问的地点。当一个用户试图访问计算机时，外部访问授权机制首先验证用户的身份，然后再检查其是否拥有使用本计算机的权限。

对交互式计算机，通常的做法是为系统保存一个所有授权用户的登录账号纪录。一个可证明他（她）身份的用户便有权使用计算机，正如图 7-1 所示，登录进某机器的用户若正试图登录另一台远程机器的话，他必须先被授权使用本地机器。远程机器可能会使用另一种形式的授权进程。

一旦某用户被授权使用某机器，此机器的操作系统将代表该用户分配一个执行进程。在登录验证完毕后，用户将可自由使用命令行解释器（shell）进程来使用任意的资源。例如，用户可试图编辑系统的口令文件。如图 7-1 所示，每个使用目标资源的进程必须得到内部授权机制的授权。内外部授权机制是不一样的：外部机制授权用户进入计算机，而内部机制只在交互过程开始时才开始活跃起来。不过，内部机制在每当用户输入一条指令后又会被暂时地唤醒，这是因为大多数指令被存储在文件中，当某个指令执行时，通常伴随文件的读写。

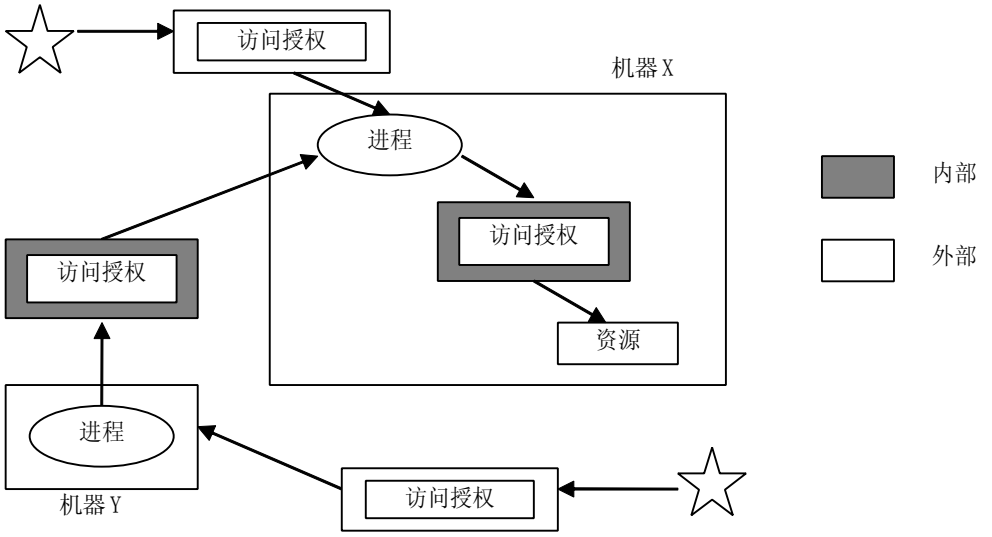


图 7-1　访问授权

7.2.4 加密

加密是将信息编码成像密文一样难解形式的技术，在现代计算机系统中，加密的重要性在迅速增加。在通过网络互连的计算机系统中，想要提供一种信息不可达的机制是困难的。因此，信息被加密成若不解密则其信息内容就不可见的形式。加密的关键处在要能高效地建立从根本上不可能被未授权用户解密的加密算法。后面我们将对现代加密手段进行详细描述。

7.3 身份鉴别

计算机安全是一个复杂的问题，他包括了处于操作系统之外的管理策略、道德问题及物理设施的安全。因为计算机安全受威胁的天性，保护系统的方法甚至比保护该系统的软硬件复杂的多。

保护机制可以或者依赖于计算机间的物理隔离，或者依赖于阻止远程用户用远程网络进入计算机的逻辑隔离。大体上，本章不准备讨论物理安全问题，除了假设某未授权用户能够通过使用某种直接（或间接）的设备连入系统来建立某种逻辑通路的现象。注意：要在一个失控的外部环境中解决问题是很困难的。在这种情形下，入侵者的行为根本不遵循任何特定的“游戏规则”。

7.3.1 用户身份鉴别

操作系统是当用户初始化一次系统会话进行交互的代理。操作系统需要验证这些用户确实是他们所

宣称的。这就是众所周知的用户身份鉴别。如果某系统能明白地正确验证其用户，那么许多保护方面的问题早就被解决了。尽管没有任何商业系统是据此假设设计的。如果这是可能的，某用户的行动都将完全由他本人承担责任，并且任何其他用户都不能假装成此用户。而一般来讲，这种确定的验证是不可能的。

用户标识符和口令的组合被广泛用于操作系统来进行用户身份鉴别。操作系统还可以使用附加手段来确认某用户是否是其宣称的。这些手段也许会涉及到类似于银行允许用户通过电话传送资金这一技术。用户可能会被要求提供除密码外的附加信息，这些视策略中授予用户的权限而定。现代操作系统甚至会采用指纹或眼球扫描技术等手段。

最简单的伪装事例是多人合用一个合法的用户名及口令。系统现在不能区分这些伪装成一个人的人们，所以它不能为他们提供安全保证。另一更严重的伪装发生在某授权用户把用户名和口令遗留在公共地点或用户名早就为人所知，而口令又被被人轻易猜出的情况下。一旦知道远程系统某用户登录名，某计算机就可利用此用户名与另一台计算机初始化一段对话模拟出一次合法的远程登录对话，然后，伪装的计算机与验证进程建立起连接，然后，迅速地、系统性地对已知用户名尝试多个不同的口令。当验证进程检测到反复的密码提供错误时，它可侦测到这种闯入尝试从而在这种错误达到一定次数时终止这次连接。但是，伪装的机器稍后将可重新建立并继续搜寻此登录名的密码。

7.3.2 网络中的身份鉴别

诸如用来传递电子邮件连接的文件传输机制也可被用于侵入机器。文件传输要求某计算机能够将信息传递到另一个文件系统的文件空间中，发送的计算机在存储文件之前必须准备接收将被存到其文件系统的任意文件。文件传输端口通常配合一则授权机制来证实发送方计算机拥有存储文件的权限。在许多系统中，这种授权机制并不实用，因为文件传输中的扩展验证增加了文件传输中的开销。

现代网络授权机制在侦测接收文件是否含有病毒和蠕虫方面显得尤为重要。病毒和蠕虫的不特点在于它们进入机器后的不同反应。病毒是一个隐藏于其他模块的软件模块，通过伪装成修错或升级补丁替换某个已存模块，病毒可植根于一个文件系统中。它也可能在运行免费游戏或其他软件时被装载，这个隐藏文件将执行它所要做的任务。但它也会执行一些隐蔽的函数，譬如留下给入侵者日后使用的隐藏漏洞或植入破坏系统资源的程序。近几年来，病毒已成为软件业中显著的一部分，特别是因为计算在两方面的进化：首先是软盘在个人计算机中的广泛使用，软盘是病毒的理想载体，主要是因为它的容量和运行它的程序；其次，互联网成为了病毒多产的温床，主要原因是它提供了广泛的邮件、新闻组、网页及免费软件。如今，已生产出很多明确地以侦测（尽可能地）病毒的存在及移除病毒为目标的产品。

蠕虫不同于病毒之处在于它是一个活动的入侵实体，它可能以文件形式进入某机器，但以后它将自我独立地运行。一旦某个含蠕虫的文件进入到文件系统中，蠕虫就找到进程管理器中的漏洞以便执行它自己。举个例子，有一个著名的蠕虫程序---“莫里斯蠕虫”，它诞生的目的是通过利用 finger 命令侵入 UNIX 系统。

在 UNIX 系统中，命令“finger name@host 打印出一组标准的关于某用户身份证明信息的概要，其中的姓名部分可能是在口令文件中用户名的一部分。因此，假使某人知道该用户的证实姓名，他就很容易找出该用户的登录名。概要的主机部分允许 finger 指令连接到远程主机上以执行 finger 指令去寻找该用户的信息。Finger 指令还打印其他诸如.plan 文件等在标准文件中寻找到的信息。出于对其他用户的礼貌，用户一般将他们的个人信息存放在.plan 文件中。但这些文件亦可被用做猜测口令的基石。虽然 finger 指令对使用电子邮件程序来寻找登录名来说是无用的工具，它仍可作为一个被用于收集入侵信息的工具的一个例子。

莫里斯蠕虫在远程机器上用一个对 finger 程序的字符串数组来讲过长的名字执行 finger 指令，结果破坏了 finger 的后台运行栈，导致当后台完成此命令后，在为输入的 finger 调用服务之前，后台再也不能返回它正在执行的程序。取而代之地，它转移开去为蠕虫运行一小段唤醒远程 shell 的代码。这样，蠕虫就控制了被入侵机器的一个进程，从而使其能够利用各种资源引起导致机器性能低下的可观的破坏。尽管存在病毒、蠕虫及其他隐藏模块，商业系统仍在使用基本的账号和口令法来完成外部验证。通过仔细分析有运行病毒、蠕虫和猜测口令的暗示行为可使这些系统得到加强。不幸的是，这种外部验证机制必须为每个账号采用所有可能的入侵模型并逐一检查之---这是一项可怕的工作。同时，审核登录尝试的路线可用于侦测归纳性的入侵尝试，虽然对阻止破坏来讲可能太迟了。

7.3.3 Kerberos 网络身份鉴别

Kerberos 是一套可用于验证一个用不可靠网络中的计算机进入另一台计算机的网络协议。就是说 Kerberos 假定在网络中流动的信息传输时是可干预的，更进一步，Kerberos 并不假定在两台计算机上的操作系统是足够安全的。此项技术于 20 世纪 80 年代在麻省理工学院开发并在今日广泛使用。

在 Kerberos 中，它假设在一台计算机（客户机）上的进程利用网络通信希望占有另一台计算机（服务器）上进程的服务。Kerberos 提供一台身份鉴别服务器及协议来允许客户机和服务器传送验证消息到特定会话段中的协助进程中。在协议中遵守以下步骤（图示见图 7-2）：

步骤 1： 客户机向身份鉴别服务器请求服务器进程的证书.

步骤 2： 身份鉴别服务器把一张后来用用户密钥加密过的令牌和一个会话密钥当作证书返回。后面将更详细地介绍加密，现在只要知道那张令牌和会话密钥（及其组合）只有客户机才能读。令牌中有含有客户机证明和用服务器密钥加密的会话密钥的复件的域，这意味着能够解释该令牌的域的进程只能是服务器。

步骤 3： 当客户机获得证书后，它将令牌和会话密钥解密，同时保留一份会话密钥的拷贝以便它可验证来自服务器的信息。

步骤 4： 客户机然后发送一份加密域不变的令牌复件给服务器。

步骤 5： 服务器对令牌的复件进行解密以便获取一份客户机证明和会话密钥的安全复件。

在此协议中，身份鉴别服务器必须是可信的，因为它拥有一份客户机证明的安全复件，它知道如何进行只有客户机才能解密的信息加密方法，还可以创建一个独特的会话密钥来代表客户机和服务器间的会话。因为身份鉴别服务器能够为客户机和服务器加密信息，它可提供给客户机一个“容器”---含有客户机不可读但可传送给服务器的信息的证书。这类似于拥有一张背面印有代表账号数字的条形码的信用卡，你事实上根本不能从条形码中读出什么，但当你把此卡提交给一台自动柜员机（服务器）时，柜员机可从条形码中读出你的账号。信用卡就是一张拥有加密账号数字的令牌（虽然它没有会话密钥）。

当完成步骤 1 到 5 后，客户机和服务器进程都拥有一份他们当作可信身份鉴别服务器的加密过的信息的会话密钥的拷贝。网络上的入侵者若不知道如何对其进行解密就将不能读出这些加密信息。他也不能改变信息来生成欺骗性的证明文件。另外，服务器拥有一份可信的客户机证明的拷贝以便当客户机向服务器发送消息时，服务器能够验证用户的身份和会话密钥。

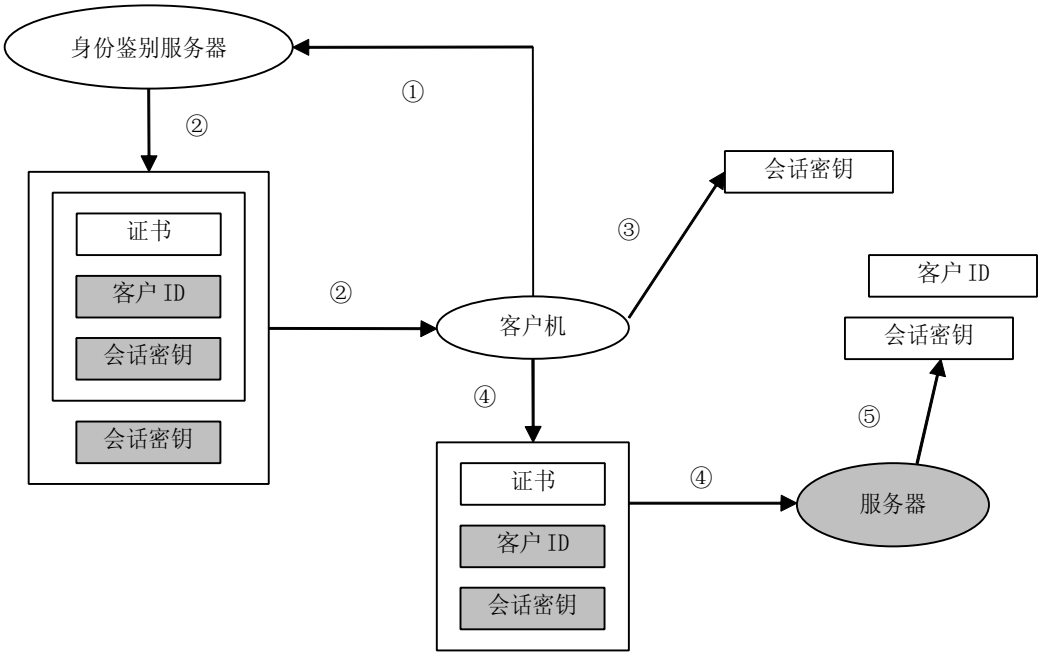


图 7-2 Kerberos 验证密钥分配

7.4 内部访问授权

内部访问授权是管理资源共享任务的一部分。在这里，目标是为了保护进程的资源不被其他的行为改变。设想进程 A 拥有资源 W，X，Y，Z，如图 7-3 所示，这其中的某些资源由其他进程共享：例如，进程 B 对 W 有读取权限，而 C 对 W 有写的权限，B 对 Y 刻读取，而 C 对 Y 无任何权限，C 对 X 可读，B 对 X 无任何权限，并且 A 对 Z 有私有的所有权限。

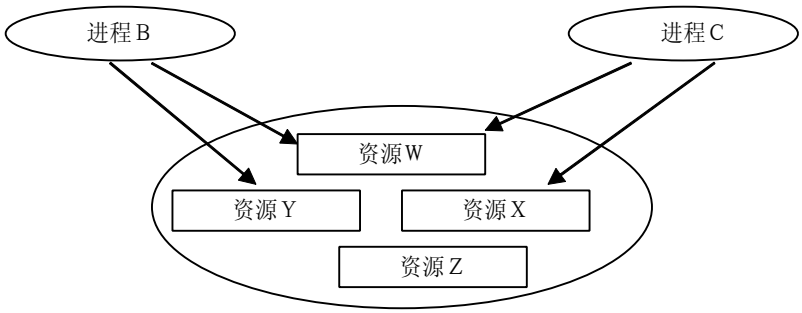


图 7-3　受限资源共享

研究者依其特点对保护问题进行了如下所列的分类，并试图在此环境中设计策略和机制来阐述它们。

- 共享参数：假如其他进程能够无区别地在某进程的地址空间内改变参数，那么，该进程的资源策略将受到破坏。例如，假设某进程在其他进程的地址空间内调用过程，而被调用过程改变了送入该过程的参数，故当调用者再次获得控制时，在它的地址空间内的变量已被调用的过 程改变了。
- 限制：限制是参数共享的归纳性结果。假设某进程希望把分散的信息集中到某特定环中，挑战在于包含资源的所有权限以使它们不能把资源传播到指定的进程集合之外。
- 分配权限：在一个保护系统中，可以允许一个进程为另一个进程提供使用其资源的特定权限。在某些情况下，第一个进程应能在任何时刻取消这些权限，权限只能暂时由一个进程赋予另 一个进程。如果一个进程为另一个进程提供权限就可能产生一个敏感的问题：接收进程及收到 的权限送给了其他并没有得到资源所有者知道或许可的进程，在未得到资源所有者的明确许可下，有些保护系统是不允许这种授权繁殖的现象的。
- 特洛伊木马：特洛伊木马问题是一种特殊的权限分配问题的例子——某服务程序由一个使用本身权限的客户机进程使用。假设服务器程序代表自己利用客户机的进程权限去获取资源就 被叫做特洛伊木马。

7.4.1 资源保护模型

一般来讲，一个系统拥有积极的和消极的两部分。积极的部分，例如进程或线程，代表了用户的行为；消极的部分，类似于资源，在保护系统中叫做对象（这里的保护对象不同于“面向对象编程”中的对象，之所以用对象这个字眼是因为其流行于保护学说中。在下面要讨论的保护模型中，进程按权限要求它做的去访问对象。

在现代操作系统中，一个进程在不同时刻，依赖于其当前所做的任务，对某对象有不同的权限。例如，一个执行系统调用的进程拥有访问操作系统的权限，它一般也拥有用户所有的权限。举例来说，当使用系统表和操作系统资源时，UNIX 系统在一个二进制文件中用 SetUID 位来允许此文件暂时性地拥有超级用户权限。在任何给定的时刻，某进程拥有的特定的一套权限都遵从于其所在的保护域。因此任何关于使用某进程的决定的对象必须包括此进程执行的所在保护域的因素。一个主体是一个运行在特定保护域的进程。主体 A 可能是作为一个可执行程序的进程，而主体 Y 可能是作为一个系统调用的同一进程。“访问”的含义是不能概括一个进程是如何控制另一个进程的，这意味着对象的集合包括系统内所有的消极因素及所有的主体。现在，基本的保护模型可描述为用于指定主体和对象的动态联系的系统主体、对象和机制。

一个保护系统由一套指定保护策略的对象、主体和规则构成，它体现了通过系统保护状态定义的主体的可访问性。系统要保证为每次对象的调用都检查一下保护状态，如图 7-4 中的 X 通过主体 S 进行的检查。内部保护状态只有通过一套执行了外部安全策略的规则才能被改变。

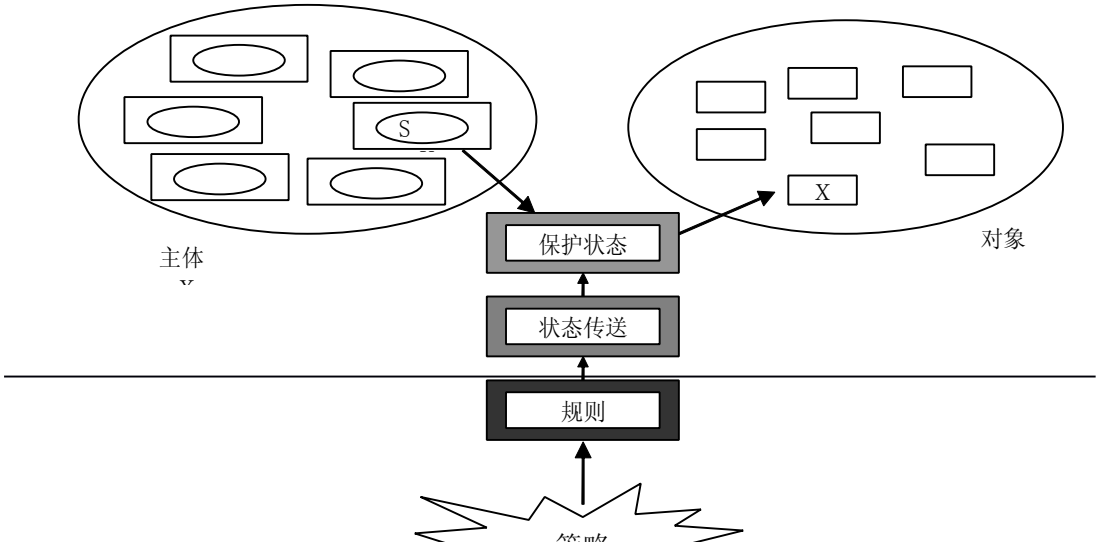


图 7-4　一个保护系统

保护状态可抽象化为访问矩阵。在访问矩阵 A 中，用行代表主体，用列代表对象，所有主体也是对象，因为进程需要能对其他进程实施控制。A[S，X]中的每个入口是一个描述对象 S 对对象 X 的访问权的集合。每次访问包含以下步骤（见图 7-5）。

步骤 1：　主体对对象初始化类型 α 使用。

步骤 2：　保护系统验证 S 并代表 S 产生（S，α，X），由于身份由系统提供，这个主体不能伪造主体身份。

步骤 3：　对象 X 的检查器查询 A[S，X]，如果 α ∈A[S，X]则访问有效，若 α！∈A[S，X]则访问无效。

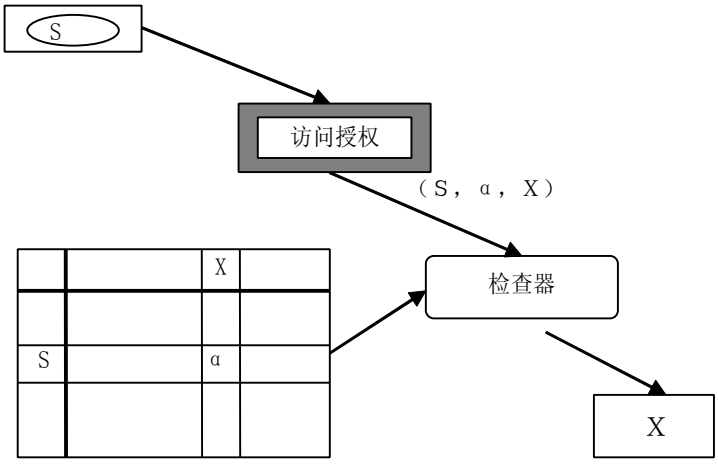


图 7-5.保护状态说明

访问矩阵保护机制可用于执行许多不同的安全策略。例如，假设某简单系统是如下构成的：

subjects = {S1，S2，S3}

objects = subjects ∪ {F1，F2，D1，D2}

这里 F1 和 F2 表示文件，D1、D2 表示设备。图 8-6 是一个代表了一个系统保护状态的访问矩阵，每个主体对其自身有控制权。S1 对 S2 有阻止、唤醒和占有的特权，对 S3 有占有的特权。文件 F1 可被 S1 进行读*和写*。S2 是 F1 的所有者，S3 对 F1 有删除权。

举例来讲，如果 S2 试图更新对 F2 的访问，则它先初始化这个访问，然后，导致保护系统建立一张表（S2，update F2）的纪录，这个纪录被送给 F2 的检查器，检查器查询 A[S2，F2]，因为此更新存在于 A[S2，F2]中，所以这个访问是有效的，于是此主体被许可更新此文件对象。若 S2 试图对 F2 执行运行的访问，则它先初始化这个访问，再由保护系统产生一个表（S2，execute F2）的纪录，这个纪录被送给 F2 的检查器检查 A[S2，F2]，因为运行权不在 A[S2，F2]中，故这个访问无效，同时这个侵犯被报告给操作系统。

	S1	S2	S3	F1	F2	D1	D2
S1	控制	阻塞 唤醒 占有	控制 占有	读* 写*		查找	占有

S2	控制	结束	占有	更新	占有	查找*
S3		控制	删除	执行 占有		

图 7-6　保护状态

7.4.2 保护状态的改变

保护系统用策略规则来控制用于切换保护状态的手段。就是说，可通过选用在矩阵中出现的访问类型和定义一套保护状态转换规则来定义策略。作为例子，我们在这里用一套来自 Graham 和 Denning 的规则来阐明可传送于主体间的权限。

例如，在图 7-7 中显示的规则实现了一则特定的保护策略。它们是用图 7-6 中所示的访问类型来定义的。在图中，S0 试图通过执行改变访问矩阵入口 A[S,X]的命令来改变保护状态。例如，S0 试图批准 S3 对 D2 的读的访问权，仅当此指令的所有者属于 A[S0，X]时，此指令才可执行。这导致读的访问权加入到 A[S3,D2]中。这个安全策略的例子的目的是为了提出伪装、共享参数和限制问题的。

规则	S0 的命令	权限	效果
1	transfer{α｜α*} to{S,X}	α*∈A[S0,X]	A[S,X]=A[S,X]∪{α｜α*}
2	grant{α｜α*} to {S,X}	owner∈A[S0,X]	A[S,X]=A[S,X]∪{α｜α*}
3	delete α from {S,X}	control∈A[S0,S] or owner∈A[S0,S]	A[S,X]=A[S,X]-{α}

图 7-7　策略规则

在策略规则中，符号*被称为拷贝标志，若某进程 S0 对 X 有使用权且拷贝标志被设为 α 对 X 可访问，则它可传送使用权 α 给对象 X 再给另一个进程 S。在图 9-6 中，S1 可为 F1 通过传输读和读*权给 S2 或 S3 来改变保护状态，因为 S1 的读取权上有拷贝标志.对规则 2，假如 S0 占有对象 X，不管有没有拷贝标志，S0 主体都可批准给对象 X 权限。

拷贝标志及规则的设计目标是防止未知的主体间的权限繁殖，拷贝标志可从一个主体传送给另一个主体，或者可传送给一个拷贝标志已清除的主体。这里的传输规则必须是非破坏性的拷贝规则。另一种策略可能要求传输是破坏性的，即当一个非所有者主体向另一个主体传输访问权后，第一个主体就失去了它自己的访问权。这样的策略可用于严密地保护所有者主体的权限。删除规则被用于唤醒来自于另一个主体的对象，在一个主体想删除一个对象之前，它或者必须控制失去访问权的主体或者成为此对象的占有者。相应于这套规则暗指的策略，若某主体是另一个主体的占有者，那么它对那个主体也拥有控制权。这个例子阐述了决定某机制是否足够实现某一大类策略及此机制和策略的结合确实能达成一个可接受的解决方案的问题的基本的复杂性。而且，它也显示了怎样利用规则来建立起预想中的策略。

Graham、Denning[1972]指出了在图 7-7 中的规则（和其他一些规则）定义了一个可用于解决一些本章开头提到的问题的保护系统：

- 伪装：此模型要求该实现能阻止某主体伪装成另一主体。验证模块可复杂到任何安全策略所要达到的，在此模块验证完此主体后，它为 S 产生出一个不可伪造的标记来执行一个到 X 的 α 访问，然后把它送入 X 的检查器。这就杜绝了伪装。
- 限制和分配权限：对解决限制问题的模块和规则来说，它们必须提供一种机制，在此机制下，权限被限制于指定的主体集合里。拷贝标志限制了传输过程中的权限繁殖，同时，所有权成为批准权限的前提。然而，还有很多值得考虑的敏感问题。能够限制主体繁殖权限和信息的想法是值得期待的。因为读的访问提供了复制信息的能力，在确信其不含有权限和信息的条件下，允许一个不可信的子系统提供服务是困难的。通常，这需要保证可疑的主体是无记忆性的---即它不具备保存信息或将信息泄漏给其他主体的能力。这意味着只要考虑程序的行为就可完全解决限制问题。若不可信主体不能证实为无记忆性的，限制问题就无法解决。
- 参数共享：我们可通过只允许不可信主体对对象的间接访问仔细地检查参数共享。某所有者主体可创建一个“门卫”主体来保护对象不受不可信主体的有害访问。本质上，这导致用户把访问权控制委托给了“门卫”。最终用户只需要用户有对“门卫”主体的访问权就可获得对对象的明确的占有权。所有者可在任何时刻取消不可信任子系统对“门卫”的访问，而“门卫”可对不可信主体的每次访问进行验证。
- 特洛伊木马：特洛伊木马问题的产生是因为某进程假定另一进程的权限在代表它运行。本模型把两个使用相同权限的进程区别为不同的主体，本模型使利用一套恰当的规则来解决特洛伊木马问题成为可能---不同于我们在上面例子中的解决方法。然而，许多规则集可能会为特定策

略解决问题但并不能保证独立于策略的解决方案。（作为对比，上面的伪装的解决方案就是通过选择某些独立于策略的规则得到的。）

7.4.3 保护机制的开销

指出引入保护机制带来的管理开支会严重影响系统的运行性能是值得的。基本保护模型要求每次资源的使用之前必须通过一个检查器，操作系统设计者必须考虑在存在保护机制的情况下，运行性能上的开销是否值得。在某些信息必须保密的环境中（如关于某公司财政中心或涉及国家防御方面的信息），性能上的开销也许算不了什么，这些信息必须受到保护，否则，计算机系统根本起不了什么益处。尽管如此，对操作系统设计者来讲，面临的挑战就是要设计可能的如下面讨论的最高效的机制。

7.5 内部授权的实现

上面讨论的保护模型描述了一套可用于解决各种安全问题的逻辑组件。什么样的实现对此模型来讲是开销高效的呢？如何才能高效地实现访问矩阵？在虚拟存储系统中，实现一个和理论模型一模一样工作的系统是开销颇巨的，所以实现只能是大致地实现模型所确定的行为。本节讨论这类实现策略。

一般的保护机制建立在保存保护状态的方案之上，通过查询这些状态来使运行中的访问生效及改变状态。要实现机制需考虑以下几点：

- 访问矩阵并不是保护状态的唯一可能的代表，但它是大多数实现的基石。
- 访问矩阵必须存在某种安全的中间存储媒体上并只可被选定的进程进行读写。
- 此设计的目标是通过保护检查序列化所有的访问，这种序列化将确认当前的保护状态可用于使此次访问生效。
- 保护机制应能够通过某个主体来验证每个要求的来源而不是通过过程调用接收作为一个参数的主体身份。
- 要实现规则，检查器必须是一个受保护的进程。不得存在其他主体通过和检查器和解并确定传输机制（如通过共享资源）的可能。

下面将讨论在不同域中进程是如何具有不同的访问权，以及保护检查器和访问矩阵的实现。

7.5.1 保护域与状态隔离

保护系统程序或用户程序不受其他用户程序的破坏，采用的方法是对计算机系统设置不同工作状态或者说处理器具有不同的工作模式，采用两态模式时常称：管态和目态。运行在管态下的程序比运行在目态下的程序有更多的访问权——例如对内存的使用权和运行扩展指令集的权限，从而限制用户使用容易造成系统混乱的那些机器指令，达到保护系统程序或其他用户程序的目的。有的计算机处理器可工作在多种状态下，也有的提供了保护环设施，都能更好地进行状态隔离。

这种两层次的域可概念推广成一个含有 N 个在保护学中被叫做域环结构的同心圆的集合。环结构通过最早出现在[Organic 1972]中的 Multics 系统构架的概括进行描述。假设保护系统有 N 个保护环构成，在这些环中，R_0 到 R_s 支持操作系统域，而 R_{s+1} 到 R_{n-1} 则被应用程序使用。则 i<j 表示 Ri 比 Rj 拥有更多的权限。内核（保护学用语）中最关键的部分在环 R_0 中运行。其次重要的操作系统层运行在 R_1 中，以此类推。最安全的用户程序层运行在环 R_{s+1} 中，越次要安全的程序运行在越往外的环中。在此模型中，硬件超级用户模式一般当软件运行在编号最低的环（可能只能是 R_0，像在 Multics 系统中）中时才会被使用。操作系统的这部分是被最仔细设计和实现的，由此推测也是被证明是无错的。

在一个环中运行的程序存在于一个分配到那个环上的文件中。保护机制提供了一种进程可安全地改变域的手段——交叉环。若在 R_i 中的某文件正在执行，那么这个进程可无需特定许可地调用在 R_j（j>=i）中的任何过程，因为这个调用代表了一个向低保护的域的调用。但是，当某个进程调用某个外部的环，操作系统机制必须确保将被作为内部环引用的返回值和参数引用是被许可的。内部环调用可通过仅是让外部环软件从一个环门卫（一个进入的检查器）进入内部环来实现。每次内部环的交叉试图会引起内部授权机制来确认此次调用有效，例如，通过中断 R_0 的一部分来唤醒到目的环的门卫。

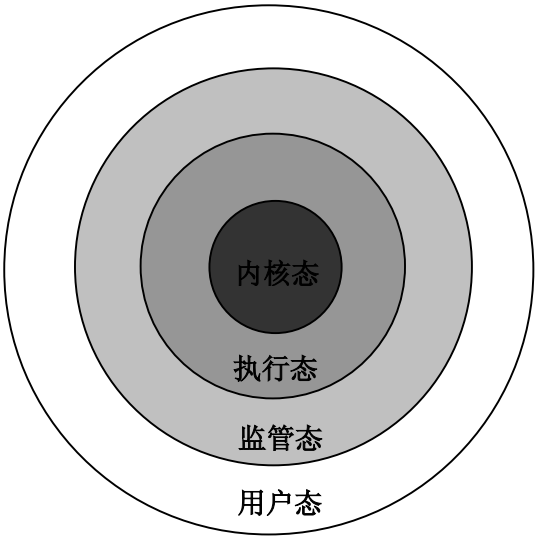
一旦某个进程进行了一个向内的调用，它就改变了域；就是说，它变成了一个不同的主体。明显地，当一个进程调用了一个内部环的进程，目标函数保存在内部环中的一个独特的文件中。一种可选的看法是操作系统暂时地放大了此进程的权限以使该进程可运行在内部环中的进程。当该进程返回到外部环中时，它将再变换它的域并回到它原先的权限拥有。

一般的环结构并不需要支持内部环数据访问，只要支持过程调用。内部环中的数据只能用一个相应于内部环的过程来访问，很像某抽象数据类型只允许通过公共接口对其进行引用。

环结构被用于当今的计算机体系结构中。例如，Intel 80386 微处理器采用了一种和我们上面的讨论有相似之处的四层结构。在 Intel 的例子中有三层指令集。层 2 和 3 的指令是普通的应用程序指令集，尽管操作系统代码的非关键性的部分也是运行在层 2 的。层 1 的指令包括输入输出指令。层 0 的指令通过使用系统操作全局描述表来操纵内存段，它也执行设备环境的转换。这种结构及它的后继产品（80486 及奔腾处理器）意图在于让层 0 支持内存段操作，而把输入输出操作交给某低安全级的层处理---某一编号较大的环。操作系统的主体在层 2 中操作，这里它的段可受到环结构的保护。

图 7-8　VAX/VMS 操作系统保护环

VAX/VMS 操作系统利用了处理器的四种模式构成保护环来加强对系统资源的保护和共享。这些处



理器模式决定了：指令执行特权、即处理器这时可执行的指令和存储访问特权、即当前指令可以存取的虚拟内存的位置。如图 7-8 所示四种模式：

- 内核(Kernel)态：执行 VMS 操作系统的内核，包括内存管理、中断处理、I/O 操作等。
- 执行(Executive)态：执行操作系统的各种系统调用，如文件操作等。
- 监管(Supervisor)态：执行操作系统其余系统调用，如应答用户请求。
- 用户(User)态：执行用户程序；执行诸如编译、编辑、连接、和排错等各种实用程序。

处于较低特权状态下执行的进程常常需要调用在更高特权状态下执行的例程，例如，一用户程序需要操作系统的某种服务，使用访管指令可获得这种调用，该指令会引起中断，从而将控制转交给处于高特权状态下的例程。执行返回指令可以通过正常或异常中断返回到断点。

采用保护环时具有单向调用关系，如果在域 Dj 中欲调用 Di 中的例程，则必然有 j>i。

7.5.2 空间隔离

现代操作系统常常为不同作业分配不同的地址空间，避免相互干扰。在多道程序环境中，空间隔离十分重要，它能使每个用户进程能正确运行。如果一个进程错误地写信息到另一个进程的地址空间，会导致严重的后果。

每个用户进程的内存空间可以通过虚拟存储技术来实现内存保护，分页、分段或段页式，可提供有效的内存隔离。这时操作系统能确保每个页面或每个段只被其所属进程或授权进程访问。

隔离技术能保证系统程序和用户程序的安全性，操作系统中还会采用各种技术实现其它资源的保护。

7.5.3 访问矩阵的实现

访问矩阵的可通过以下几种方法中的任一来实现。对大多数主体和对象的集合，矩阵是稀疏的，因为大多数对象只会有几个主体进行访问，而大多数主体也只会访问少数对象。这暗示高效的实现应使用入口索引表而不是将矩阵存在一个矩形数组中。例如，如果{α}是逻辑存储在 A[Si，Xj]中的字符串集合，这个索引表就可容纳表单（Si，Xj，{α}）的入口。这是解决稀疏矩阵的普遍的均衡方法。就是说，此索引表的长度正比于矩阵的入口数---密集矩阵则会产生出冗长的索引表。

另一可用的方法是把矩阵转化为列向量，每个向量存储为受保护对象的权限的列表。现在，在任何 Si 想访问对象的时刻，对象的检查器只需简单地查询列表。对给定的对象这种向量被叫做访问控制列表（ACL）。

类似地，也可以联系主体对访问矩阵的行进行压缩。一旦某主体初始化一个访问，保护系统检查列表看此主体是否拥有权限（权能）访问指定的对象。分配访问权给主体就像给某事件令牌一样。如果此

主体在它的列表中没有此权能，那么它可能连这个对象的名都不知道。如果访问矩阵以此种方式存储，它就被叫做一个权能列表。访问控制列表和权能将在下文详细讨论。Kerberos 令牌就是权能的一个例子。

早期的系统试图用锁和钥匙的方法提供保护机制。这种途径将在下面的小节中描述。这是一种含有某些 ACL 和权能列表属性的简单的机制。在更仔细地考察这些访问矩阵的完全实现之前，先让我们来回顾一下锁和钥匙的途径。

7.5.3.1 内存锁和钥匙

20 世纪 70 年代以来，为实现内存对象的保护检查器人们投入了相当大的努力。早期的保护检查器不支持完整的访问矩阵的通则，作为代替，它们使用了一种较弱的 ACL 形式。

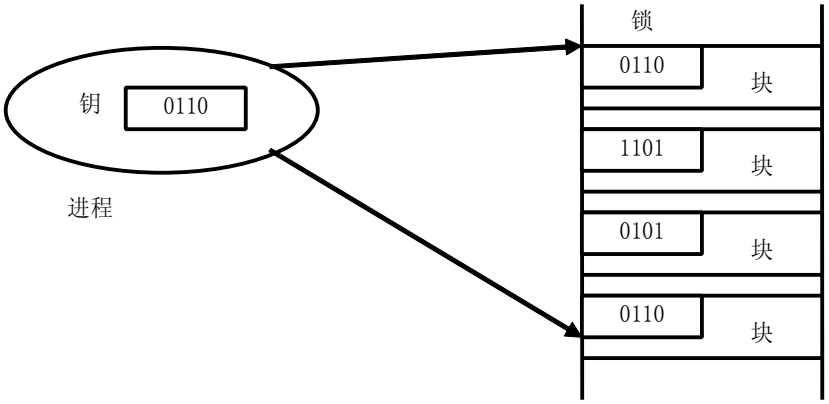


图 7-9　内存锁

某可分配的内存单元可以是一个字，一个分区，一张页面或一个段。在 20 世纪 60 年代，机器有时在 h 字节的可分配块上使用内存锁。假定每个块可分配一个 k 位的锁的值且每个进程的描述符都包含一个 k 位的钥匙设置（见图 7-9）。当一个内存块被分配给一个进程时，就设定一个和进程钥匙一样的锁。进程钥和内存锁只有系统处于超级用户模式时才能设定。在每次内存访问时，硬件通过保存钥匙的值到 CPU 的寄存器中和为内存设定一个锁寄存器对此次访问进行检查。因此这些内存锁可看作访问控制列表。每当在它块内的单词被引用，锁寄存器就伴随锁被加载了。如果锁和钥匙是一样的，那么此次访问被许可；否则，此次访问企图就引起一个送给操作系统的中断。在 20 世纪 60 年代的计算机中，h=16，k=4。这就意味着要么限制多道程序的道数小于 16，要么重复使用锁和钥匙以使某进程的钥匙能够同时打开另一进程在内存上的锁。

这种途径既简单在使用中又有效。但是，它不能分辨不同种类的访问，也不允许操作系统的或用户进程的共享，因为锁和钥匙必须一模一样。此共享问题可通过保留特殊模式给特殊使用而解决；例如，某种钥匙模式可能是在超级用户模式下任何主体的“主人钥匙”。相对应的锁模式可能表示内存可“无保护地”由任何主体访问。超级用户访问和共享于是被实现了，虽然这是一个弱共享机制。

7.5.3.2 访问控制列表

访问控制列表以普遍的形式已经使用了很多年。在这种途径下，资源管理者为每种资源实行一个 ACL。在大多数应用中，主体只有在它打开或分配资源（而不是每次访问）时才会受到检查。如果该主体及它的访问类型不在 ACL 中，那么打开或分配资源的行为就失败了。

UNIX 文件保护机制是 ACL 途径著名的应用。每个 UNIX 用户用一个用户身份证（UID）进行验证。每个用户也可从属于不同的用户组，由组的身份证（GID）注明。某进程的 UID 和 GID 是该进程描述符的一部分，意味着当这些进程想访问某个文件时系统程序可方便地进行检查。UNIX 用十个字符描述了访问相应文件和目录所需的权限。开头的“d”字符表示此项目是一个目录，“-”字符表示其是一个文件。后面的 9 各字符三个一组地进行解释，第一组表示文件所有者对此文件的使用权限。第二组描述了该文件所属组成员对此文件的所持有的权限，第三组描述了其他所有用户拥有的许可（它被叫做“通用”许可位）。如果某个三元组在第一个位置上有一个“r”字符，相应的用户对此文件或目录就有读取的许可；“-”表示该用户不具备读的许可。第二个位置的“w”代表写的许可，第三个位置的“x”代表运行的许可。

当某进程在执行一个可信软件模块时，每个文件也可拥有用于暂时提高此进程的权限的一个 setUID 标志位。当一个进程执行从一个 setUID 位为真的文件中加载的程序时，只要它在执行该文件中的程序，就假定它就是该文件的 UID 的进程。

Windows NT 内核的最底层通过内涵一套完全的 ACL 机制[Solomon，1998]来支持安全操作。内核的主体部分通过用户空间组件指定的保护策略来对每次对象的访问进行检查。无论什么时候，任何线程对系统进行调用，进行访问处理的内核部分提交给验证机制一份关于访问企图的描述。验证机制确定线程的身份和访问类型，然后验证此线程是否被许可访问它想访问的对象（通过从 ACL 中得到的信息）。

7.5.3.3 权能

权能（capability）是一个独特的，通用的代表在一个系统中对某对象的访问权的名字。因此，权能可能是一张 Kerberos 令牌，也可能是“对盘 k 上 i 扇区的读取访问”和“对进程 j 的地址空间的虚拟地址的写入访问”。在保护模型的设备环境中，在访问矩阵中的一列是由主体 S 控制形式（对象或 α 访问）的权能的列表。权能被用于今日的操作系统来实现保护机制和生成主体间的地址空间。

权能充当两种职能。首先，它为在一个很大的地址空间里的资源提供地址。其次，拥有权能代表了主体对描述对象的授权。后者是基于权能的系统的关键点（回想一下 Kerberos 的例子）。就是说，当一个主体获得某权能时，验证就发生了。一旦某权能被发布，就没有必要让运行检查器和访问矩阵去检查每次访问。

权能的使用暗示了如下必然的属性：

- 权能使用的值必须从一个巨大的名字空间衍生。这是因为一个使用权能的系统将需要许多不同的权能实例来代表所有主体到所有对象的所有访问。
- 权能必须是独特的并且一旦被分配再也不能被重用。这防止了通过无意地给一个主体提供了原来对某对象的访问而引起的权能的“再生利用”现象。
- 权能必须和似是而非的名字区别开来。例如，系统不能把普通的整型数字或指针和权能弄混。

有两种基本的实现权能的途径。它们或者完全地在操作系统地址空间之内实现，或者硬件可实现对权能的特殊支持。作为一个实用的途径，权能有时是通过提供一个很大的空间然后从中随机地提取而得到。但这种途径并不能保证绝对的保护；权能不能被保证是独特的虽然在很大的可能下它们确实是的。

某权能在操作系统中可表示为一种类型的标量。概念上它是形式

```
struct capability{
    type tag;
    long addr;
}
```

的一个纪录。

若 c 是一个权能，那么它的 tag 域—c.tag 就被设成拥有该权能的值；而它的 address 域---c.addr 就是该权能的通用地址。某对象只有当它拥有一个 c.addr 指出访问类型所要资源“进入点”的权能才能访问另一个对象。如果本来就有一个对象检查器，那就只需要验证 c.tag 被设定拥有权能的值以能够保证此访问有效。

每个主体都要能获得和使用权能，但不允许它们创建权能。若主体的权能都维持在操作系统的空间内，主体就不能不受操作系统干预地创建一个类似于权能的数据结构。然而，主体可不受操作系统支配地利用它的权能去访问对象。既然这样，那么权能就不会被操作系统独占地管理，数据结构中的 tag 域就可被省去，因为权能的使用已经暗示了它的类型。例如，在一个分段的虚拟存储系统中，所有的权能被存在一个主体的权能段内。

标记可通过把每个 tag 域联系到每个内存单元的方法在硬件上实现。早期的 Burroghs 计算机结构就是使用硬件权能的例子。单词中的 Tag 域可通过超级用户模式指令设成 *capability* 或 *other*。在对象保护检查器中，tag 域只可被超级用户模式指令进行读取。

Mach 的操作系统提供了内核级的权能。Mach 是 Rochester Intelligent Gateway（RIG）和 Accent 操作系统的后继产品。三者采用了不同的权能使用方法。IPC 机制中权能的使用是一个如何在现代操作系统中使用权能的好例子[Accetta，et al. 1986]。

Mach IPC 建立在消息和端口上。消息是一种数据结构，而端口是一个在内核中实现的通讯频道并用于从其他线程接收消息。每个端口接收不同特定类型的消息。若某线程试图挂起另一个线程，它把挂起消息送到那个线程的端口，然后，它就拥有了一个可控制第二个线程的权能。

端口是在被使用之前必须请求和分配的受保护的内核对象。若某线程知道一个端口，它就有权向那个端口发送消息，而接收者就会尊重这些消息。再使用一个端口之前，内核必须分配这个端口。端口等同于权能。若某线程拥有权能，它就可向目的端口发送消息；若没有就不能向端口发送信息。

7.6 密码学

不可避免地，即使在一个安全的操作系统内，重要的信息也会暂时性地脱离保护，例如，当诸如访问矩阵入口或权能的信息从系统的一个部分传送到另一个部分时。在 Kerberos 验证策略中我们已给出了例子。加密技术可用于将明文转化为密文来保护暴露在未受保护媒体上的原文。我们定义一个加密函数 encrypt 和一个解密函数 decrypt，这里

```
decrypt（key’，encrypt（key，clearText））=clearText
```

我们用 encrypt 和一个密钥 key 把明文加密成密文。另一个不同的密钥---key’，decrypt 用它将密文转化为明文。

有两种建立加密、解密机制的机制。一种建立的设计是加密和解密的细节未知，以至此实现的机制

的一部分是保密的。另一种建立的设计是机制是公开的，但密钥是未知且难以伪造的。对第一种情况，函数 encrypt 和 decrypt 很复杂，以至无法猜出转换是如何完成的。对第二种情况，密钥是复杂的，以至无法猜出。

有一类对称的加密技术，它的加密密钥和解密密钥是一样的。这种加密在同时对信息进行加解密的可信子系统中是有用的。例如，某操作系统用户验证系统可用此技术来存储口令。当一个用户定义一个口令后，操作系统用它私有的密钥把它加密进一个口令对象中。在验证时，操作系统用它的密钥解码口令对象的入口来和用户提供的口令进行比较。UNIX 系统就是用这种途径来保护它的口令的。

在由不同主体实行加解密的情况下就需要使用非对称的加密和解密。依赖于主体的权限，每个主体都有它自己的适合加解密的密钥。在一般的保护策略中就需要这种加密手段。

公开密钥加密法就是非对称加密算法的实例。在这里，一个密钥---Ks 是公开的，另一个密钥---Kp 是秘密的。如果明文是用 Ks 加密的，则它可被 Kp 解密，反之亦然。Ks 是某用户秘密持有的，而 Kp 对该用户想通信的任何人都是公开的。那么，任何知道 Kp 的用户都可以给拥有 Ks 的用户安全地发送他们用 Kp 加密过的信息。（如果此机制是正确的，那么，除了拥有 Ks 的用户，没有人能够解密用 Ks 加密过的信息。）

在这种途径下，Ks 的拥有者可建立任何其他人都可证实其真实性的信息。信息的建立者用 Ks 加密信息，然后，其他任何人可用 Kp 来看它是否能解密出信息。如果可以，说明信息是可信的。

某用户可通过如下的步骤创建可验证的、私有的通信：

- 信息由发送者签名并用标记 Ks 加密。
- 加密好的签名和信息再用接收者的公共密钥加密。

完美隐私保护（Pretty Good Privacy）是一个 Zimmerman[1994]发明的流行的公共密钥加密系统。在 PGP 中，公共密钥包括持有者的电邮地址，一旦这个密钥被创建，它就符号化了。私有密钥包括身份验证和创建时间以及私有的密钥字符和一个口令。密钥被存于一个含有所有者身份、这套密钥创建时间及定义此密钥所使用的资料的密钥证书中。公共密钥证书包括生成公共密钥的资料及可靠的包含生成私有密钥的资料的密钥证书。某用户可将几个那样的公共密钥和安全的密钥证书保存在公共的和可靠的密码环中。

邮件文摘（Message digest）是一个 128 位的要在某个在不安全的网络传送的消息的 “加密的强单向 hash 函数”[Zimmerman，1994]。一般地，要伪造一个邮件文摘是不可能的。一旦一个邮件文摘被计算出来，此消息的一个签名便通过用私有密钥加密这个邮件文摘衍生出来。通过生成一个含有内部的 64 位的密码 ID、一个签名及此签名生成的时间戳的头可签署一份秘密文档。接收者用密码 ID 或者检索发送者的公共密钥（从接收者自己的公共密钥环中），假如它正在验证消息的话；或者从它的私有密钥环中检索它自己的私有密钥，假如它正在解密信息的话。

7.7 实例研究：Windows2000 的安全性

7.7.1 Windows2000 安全性概述

Windows2000 提供了一组全面的、可配置的安全性服务，这些服务达到了美国政府用于受托操作系统的国防部 C2 级要求。1995 年，两个独立配置的 Windows NT Server 和 Windows NT Worksbtion 3.5 正式得到美国国家计算机安全中心(United States National Computer Security Center，NCSC)的 C2 级认证。(详细信息请参见 http://www.radium.ncsc.mil)。1996 年，Windows NT Server 和 Windows NT Workstation 3.51 的独立和网络配置都通过了英国信息技术安全评估和认证(UK Information Technology Security Evaluation and Certification，ITSEC)委员会的 F-C2/E3 级认证。这个评价与美国的 C2 级评价等同。(关于 ITSEC 的详细信息，请参见 http://www.itsec.gov.uk)。目前，Windows NT 4.0 和 Windows2000 正在接受美国 NCSC 和 ITSEC 的评测。

以下是安全性服务及其需要的基本特征：

- 安全登录机构：要求在允许用户访问系统之前，输入唯一的登录标识符和密码来标识自己。
- 谨慎访问控制：允许资源的所有者决定哪些用户可以访问资源和他们可以如何处理这些资源。所有者可以授权给某个用户或一组用户，允许他们进行各种访问。
- 安全审核：提供检测和记录与安全性有关的任何创建、访问或删除系统资源的事件或尝试的能力。登录标识符记录所有用户的身份，这样便于跟踪任何执行非法操作的用户。
- 内存保护：防止非法进程访问其他进程的专用虚拟内存。另外，Windows2000 保证当物理内存页面分配给某个用户进程时，这一页中绝对不含有其他进程的脏数据。

Windows2000 通过它的安全性子系统和相关组件来达到这些需求，并引进了一系列安全性术语，例如活动目录、组织单元、用户、组、域、安全 ID、访问控制列表、访问令牌、用户权限和安全审核。与 Windows NT4 比较，为适应分布式安全性的需要，Windows2000 对安全性模型进行了相当的扩展。简单地说，这些增强包括：

- 活动目录：为大域提供了可升级的、灵活的账号管理，允许精确地访问控制和管理委托。

- Kerberos 5 身份验证协议：它是一种成熟的作为网络身份验证默认协议的 Internet 安全性标准，为交互式操作身份验证和使用公共密钥证书的身份验证提供了基础。
- 基于 Secure Sockets Layer 3.0 的安全通道。
- CryptoAPI 2.0：提供了公共网络数据完整性和保密性的传送工业标准协议。

7.7.2 Windows2000 安全性系统组件

实现 Windows2000 的安全性系统的一些组件和数据库如下：

- 安全引用监视器(SRM)：是 WindowsNT 执行体(NTOSKRNL.EXE)的一个组件，该组件负责执行对对象的安全访问的检查、处理权限(用户权限)和产生任何的结果安全审核消息。
- 本地安全权限(LSA)服务器：是一个运行映像 LSASS．EXE 的用户态进程，它负责本地系统安全性规则(例如允许用户登录到机器的规则、密码规则、授予用户和组的权限列表以及系统安全性审核设置)、用户身份验证以及向“事件日志”发送安全性审核消息。
- LSA 策略数据库：是一个包含了系统安全性规则设置的数据库。该数据库被保存在注册表中的 HKEY-LOCAL-MACHINE\security 下。它包含了这样一些信息：哪些域被信任用于认证登录企图；哪些用户可以访问系统以及怎样访问(交互、网络和服务登录方式)；谁被赋予了哪些权限；以及执行的安全性审核的种类。
- 安全账号管理器服务器：是一组负责管理数据库的子例程，这个数据库包含定义在本地机器上或用于域(如果系统是域控制器)的用户名和组。SAM 在 LSASS 进程的描述表中运行。
- SAM 数据库：是一个包含定义用户和组以及它们的密码和属性的数据库。该数据库被保存在 HKEY-LOCAL-MACHINE\SAM 下的注册表中。
- 默认身份认证包：是一个被称为 MSV1_0．DLL 的动态链接库(DLL)，在进行 WIndows 身份验证的 LSASS 进程的描述表中运行。这个 DLL 负责检查给定的用户名和密码是否和 SAM 数据库中指定的相匹配，如果匹配，返回该用户的信息。
- 登录进程：是一个运行 WINLOGON.EXE 的用户态进程，它负责搜寻用户名和密码，将它们发送给 LSA 用以验证它们，并在用户会话中创建初始化进程。
- 网络登录服务：是一个响应网络登录请求的 SERVICES.EXE 进程内部的用户态服务。身份验证同本地登录一样，是通过把它们发送到 LSASS 进程来验证的。

7.7.3 Windows2000 保护对象

保护对象是谨慎访问控制和审核的基本要素。Windows2000 上可以被保护的对象包括文件、设备、邮件槽、已命名的和未命名的管道、进程、线程、事件、互斥体、信号量、可等待定时器、访问令牌、窗口站、桌面、网络共享、服务、注册表键和打印机。

因为被导出到用户态的系统资源(和以后需要的安全性有效权限)是作为对象来实现的，因此 Windows2000 对象管理器就成为执行安全访问检查的关键关口。要控制谁可以处理对象，安全系统就必须首先明确每个用户的标识。之所以需要确认用户标识，是因为 Windows2000 在访问任何系统资源之前都要进行身份验证登录。当一个线程打开某对象的句柄时，对象管理器和安全系统就会使用调用者的安全标识来决定是否将申请的句柄授予调用者。

以下各节从两个角度检查对象保护：控制哪些用户可以访问哪些对象和识别用户的安全信息。

7.7.3.1 安全描述体和访问控制

所有可安全的对象在它们被创建时都将被分配“安全描述体”(security descriptor)。安全描述体控制哪些用户可以对访问的对象做什么，它包含下列主要属性：

- 所有者 SID：所有者的安全 ID。
- 组 SID：用于对象主要组的 SID(只有 POSIX 使用)。
- 谨慎访问控制列表(DACL)：指定谁可以对访问的对象做什么。
- 系统访问控制列表(SACL)：指定哪些用户的哪些操作应登录到安全审核日志中。

访问控制列表(ACL)包括一个 ACL 头和零个或多个“访问控制项”(ACE)结构。具有零个 ACE 的 ACL 被称为“空 ACL”，表示没有用户可以访问该对象。

在 DACL 中，每个 ACE 都包含一个安全标识和访问掩码。DACL 中可能存在两种类型的 ACE：访问允许和访问拒绝。正如您所设想的那样，访问允许 ACE 授予用户访问权，而访问拒绝 ACE 拒绝在访问掩码中指定的访问权力。由各个 ACE 授予的访问权力的集合就构成了由 ACL 授予的一组访问权力。如果安全描述体中没有 DACL，则每个用户就拥有对象的完全访问权。另一方面，如果 DACL 为空(零个 ACE)，就没有用户可以访问对象。

系统 ACL 只包含一种类型的 ACE，称为系统审核 ACE，用来指明特定用户或组在对象上进行的应得到审核的操作(审核信息存储在系统审核日志中)。成功和不成功的尝试都可以被审核。如果系统 ACL 为空，则对象不会被审核(本章稍后将描述安全审核)。

1）分配 AGL

要确定分配给新对象的 ACL，安全系统将应用三种互斥的规则之一，步骤如下：

(1)如果调用者在创建对象时明确提供了一个安全描述体，则安全系统将把该描述体应用到对象中。

(2)如果调用者没有提供安全描述体，而对象有名称，则安全系统将在存储新对象名称的目录中查看安全描述体。一些对象目录的 ACE 可以被指定为可继承的，表示它们可以应用于在对象目录中创建的新对象上。如果存在可继承的 ACE，安全系统将它们编入 ACI，并与新对象连接。(单独的标志表明 ACE 只能被容器对象继承，而不可能被非容器对象继承。)

(3)如果以上两种情况都没有出现，安全系统会从调用者访问令牌中检索默认的 ACL，并将其应用到新对象。操作系统的几个子系统有它们在创建对象时分配的硬性编码 DACL(例如，服务、LSA 和 SAM 对象)。

2）决定访问

对对象的有效访问有两种算法：

- 其一是确定允许访问对象的最大权限(可以使用 WIN32 的 GetEffectiveRightsFromAcl 函数)。
- 其二是确定是否允许一个特定的所希望的访问(可以使用 WIN32 的 AccessCheck、AccessCheckByType 和 TrusteeAccessToObject 函数)。

第一种算法通过检查 ACL 中的项来生成授予访问掩码和拒绝访问掩码，步骤如下：

(1)如果对象没有 DACI，对象将不会、被保护，安全系统将授予所有的访问权力。

(2)如果调用者具有所有权特权，安全系统将在检查 DACL 之前授予它写入访问权力。

(3)如果调用者是对象的所有者，则被授予读取控制和写入 DACL 的访问权力。

(4)对于每一个访问拒绝的 ACE，如果其中包含与调用者的访问令牌相匹配的 SID，则 ACE 的访问掩码会被添加到拒绝访问掩码上。

(5)对于每一个访问允许的 ACE，如果其中包含与调用者的访问令牌相匹配的 SID，除非访问已被拒绝，否则 ACE 的访问掩码被添加到被计算的授予访问掩码上。

在 DACL 中所有的项检查完之后，经过计算的授予访问掩码作为允许访问对象的最大权限返回到调用者。

第二种算法依据调用者的访问令牌来确定是否授予所申请的指定访问权限。WIN32[中处理可安全对象的每一个打开的函数都有一个参数用来指定希望的访问掩码。要确定调用者是否具有访问权力，请执行下列步骤：

(1)如果对象没有 DACL，对象将不会被保护，安全系统会授予所希望的访问权力。

(2)如果调用者具有所有眯权，安全系统会在检查 DACL 之前授予它写入访问权力。如果写入访问权力是唯一请求的访问权力，则安全系统把它授予调用者。

(3)如果调用考是对象的所有者，就将被授予读取控制和写入 DACL 访问权力。如果只申请了这两个权力，则不检查 DACL 就可以授予访问权力。

(4)DACL 中的每个 ACE 都会被从头至尾检查一遍。如果 ACE 中的 SID 与调用者的访问令牌(无论是首选 SID 还是组 SID)中的“启用”(enabled)SID 匹配(SID 可以被启用或禁用)，则将处理 ACE。如果它是一个访问允许 ACE，则 ACE 中的访问掩码内的权力将被授予调用者；如果授予了所有申请的访问权力，则访问检查将继续。如果它是一个访问拒绝 ACE，任何申请的访问权力都在拒绝访问权力范围内，则对对象的访问会被拒绝。

(5)如果 DACL 已检查完毕，而一些被请求的访问权限没有被授予，则访问会被拒绝。

两种访问有效性算法都依赖于访问拒绝 ACE 被放置在访问允许 ACE 之前。

在 Windows2000 中由于引入了对象指定的 ACE 并且自动继承，所以 ACE 的顺序序变得更加复杂了。非继承的 ACE 置于继承的 ACE 之前。在继承和非继承的 ACE 中，依据 ACE 的类型来排列次序：应用于对象自身的访问拒绝 ACE，应用于对象的子对象的访问拒绝 ACE，接下来是应用于对象自身的访问允许 ACE，然后是应用于对象的于对象的访问允许 ACE。

因为在进程每次使用句柄时，安全系统都处理 DACI 是缺乏效率的，所以这种检查只在打开句柄时进行，并不是每次使用句柄时都进行。而且需要记住的是由于核心态代码使用指针而不是句柄去访问对象：所以操作系统使用对象时并不进行访问检查。换句话说，在安全性方面，Windows2000 是完全“信任”它自己的。

一旦进程成功地打开一个句柄，安全系统也不能取消已授予的访问权力，即使对象的 DACL 改变了。这就要求每次使用句柄时都要进行彻底的安全检查，而不是仅在最初创建句柄时才做这样的检查。把已经授予的访问权力直接存储在句柄中将显著地提高性能，特别是对那些具有较长 DACL 的对象。

7.7.3.2 访问令牌与模仿

“访问令牌”是一个包含进程或线程安全标识的数据结构：安全 ID(SID)、用户所属组的列表以及启用和禁用的特权列表。由于访问令牌被输出到用户态，所以使用 WIN32 中的一个函数就可以创建和处理它们。在内部，核心态访问令牌结构是一个对象，是由对象管理器分配的由执行体进程块或线程块指向的对象。您可以使用 Pview 实用工具和内核调试器来检查访问令牌对象。

每个进程都从它的创建进程继承了一个首选访问令牌。在登录时，LSASS 进程验证用户名称及口令

是否与保存在 SAM 中的一致。如果一致，则将一个访问令牌返回到 WinLogon，WinLogon 然后将该访问令牌分配到用户会话中的初始进程。接下来，在用户会话中创建的进程就继承了这个访问令牌。您也可使用 WIN32 中 LogonUser 函数生成一个访问令牌然后使用该令牌调用 WIN32 CreateProcessAsUser 函数来创建一个带有一个特定访问令牌的进程。

单个线程也可以有自己的访问令牌——如果它们在“模仿”客户。这就使得线程具有不同于进程的访问令牌。例如，服务器进程典型地模仿客户进程，这样服务器进程(它在运行时可能具有管理权力)就可以使用客户的安全配置文件而不是自己的安全配置文件来代表客户执行操作。当连接到服务器时，通过指定“服务安全特性”(Security quality of service，SQOS)，客户进程可以限制服务器进程模仿的级别。

默认情况下，除非一个线程使用 WIN32 ImpersonnateSelf 函数来请求访问令牌，否则该线程不会有自己的访问令牌，这个函数复制进程最初的访问令牌并将它分配给线程。一旦线程具有了自己的访问令牌，它就可以使用 WIN32 四个模仿函数之一来承担代表线程将要操作的客户的安全令牌。这四个函数分别是：RpcImpersonateClient、DdelmpersonateClient、ImpersonateNamedPipeClient 和 ImpersonateLoggedOnUser。如果正在使用安全支持提供程序接口，那么模仿客户访问令牌的另一种方法就是使用 ImpersonateSecurityContext 函数。

许多系统进程在名为 SYSTEM 的特殊访问令牌下运行。这个账号同 SAM 中的“管理员”账号不同，虽然它有类似的特权。在 SYSTEM 访问令牌下运行的进程有一些限制。例如，它没有域认证，这意味着在访问网络资源时，将受到限制或无权进行访问。另外，它也不能与其他非 SYSTEM 用户进程共享对象，除非使用 DACL(DACI 允许一个或一组用户访问对象)或 NULL DACL(允许所有用户访问对象)来创建这些对象。

7.7.4 Windows2000 安全审核

对象管理器可以生成审核事件作为访问检查的结果，而用户使用有效的 WIN32 函　数可直接生成这些审核事件。核心态代码通常只允许生成一个审核事件。但是，调用　审核系统服务的进程必须具有 SeAuditPrlvilege 特权才能成功地生成审核记录。这项要　求防止了恶意的用户态程序“淹没”“安全日志”。

本地系统的审核规则控制对审核一个特殊类型安全事件的决定。本地安全规则调　用的审核规则是本地系统上 LSA 维护的安全规则的一部分。LSA 向 SRM 发送消息以　通知它系统初始化时的审核规则和规则更改的时间。LSA 负责接收来自 SRM 的审核记　录，对它们进行编辑并将记录发送到“事件日志”中。LSA(而不是 SRM)发送这些记录，因为它添加了恰当的细节，例如更完全地识别被审核的进程所需的信息。

SRM 经连接到 LSA 的 IPC 发送这些审核事件。“事件记录器”将审核事件写入“安全日志”中。除了由 SRM 传递的审核事件之外，LSA 和 SAM 二者都产生 lSA 直接发送到“事件记录器”的审核记录。

当接收到审核记录后，它们被放到队列中以被发送到 LSA——它们不会被成批提交。可以使用两种方式中的一个从 SRM 中把审核记录移到安全子系统。如果审核记录较小(小于最大的 LPC 消息)，那么它就被作为一条 LPC 消息发送。审核记录从 SRM 的地址空间复制到 LSASS 进程的地址空间。如果审核记录较大，SRM 使用共享内存、使 LSASS 可以使用该消息，并在 LPC 消息中简单地传送一个指针。

7.7.5 Windows2000 登录过程

登录是通过登录进程(Winlogon)、ISA、一个或多个身份验证包和 SAM 的相互作用发生的。身份验证包是执行身份验证检查的 DLLo MSVl0 是一个用于交互式登录的身份验证包。

WinLogon 是一个受托进程，负责管理与安全性相关的用户相互作用。它协调登录，在登录时启动用户外壳，处理注销和管理各种与安全性相关的其他操作，包括登录时输入口令、更改口令以及锁定和解锁工作站。W1nLogon 进程必须确保与安全性相关的操作对任何其他活动的进程是不可见的。例如，W1nLogon 保证非受托进程在进行这些操作中的一种时不能控制桌面并由此获得访问口令。

WinLogon 是从键盘截取登录请求的唯一进程。它将调用 LSA 来确认试图登录的用户。如果用户被确认，那么该登录进程就会代表用户激活一个登录外壳。

登录进程的认证和身份验证都是在名为 GINA(图形认证和身份验证)的可替换 DLL 中实现的。标准 Windows 2000 GINA.DLL，MSGINA.DLL 实现了默认的 Windows 登录接口。但是，开发者们可以使用他们自己的 GINA.DLL 来实现其他的认证和身份验证机制，从而取代标准的 Windows 用户名／口令的方法。另外，WinLogon 还可以加载其他的网络供应商的 DLL 来进行二级身份验证。该功能能够使多个网络供应商在正常登录过程中同时收集所有的标识和认证信息。

7.7.5.1 WinLogon初始化

系统初始化过程中，在激活任何用户应用程序之前，W1nLogon 将进行一些特定的步骤以确保一旦系统为用户做好准备，它能够控制工作站：

- 创建并打开一个窗口站以代表键盘、鼠标和监视器。WinLogon 为窗口站创建一个安全描述体，该站有且只有一个只包含 WinLogon SID 的 ACE。这个唯一的安全描述体确保没有其他进程可

以访问该工作站，除非得到 WinLogon 的明确许可。

- 创建并打开三个桌面：应用程序桌面、WinLogon 桌面和屏幕保护程序桌面。在 W1nLogon 桌面上创建安全性以便只有 W1nLogon 可以访问该桌面。其他两个桌面允许 Winlogon 和用户访问。这种安排意味着任何情况下 WlnLogon 桌面都是被激活的，其他的进程不能访问与该桌面相关的任何激活的代码或数据。WindOws2000 利用该特性来保护包括口令、锁定和解锁桌面的安全操作。
- 建立与 LSA 的 LPC 连接。该连接将用于在登录、注销和口令操作期间交换信息，这些都通过调用 LsaRegisterLogonProcess 来完成。
- 调用 LsaLookupAuthenticationPackage 来获得与 MSVl_0 相关的 ID，它将在试图登录时用于身份验证操作。

然后，WinLogon 执行特定的 Windows 操作来设置窗口环境：

- 用它随后创建的窗口初始化并注册一个与 WinLogon 程序相关的窗口等级的数据结构。
- 用刚创建的窗口注册与之相关的安全注意序列(SAS)热键序列，保证在用户输入 SAS 时，WinLogon 的窗口程序能够被调用。
- 注册该窗口以便在用户注销或屏幕保护程序时间到时的时候能调用与该窗口相关的程序。WIN32 子系统检查以验证请求通知的进程是 WinLogon 进程。

一旦在初始化过程中创建了 WinLogon 桌面，那么该桌面就成为活动桌面。当 WinLogon 桌面激活时，它总是被锁定的。只有 WinLogon 解锁才能切换到应用程序桌面或屏幕保护程序桌面(只有 WinLogon 进程才能锁定或解锁桌面)。

7.7.5.2 用户登录步骤

当用户按 SAS（或按下）时，登录就开始了。在按了 SAS 以后 WinLogon 切换到安全桌面并提示输入用户名称和口令。WinLogon 也为这个用户创建了一个唯一的本地组，并将桌面的这个实例(键盘、屏幕和鼠标)分配给该用户。WinLogon 把这个组作为 LsaL。80nUset 调用的一部分传送到 LSA。如果用户成功地登录，该组将包括在登录进程令牌中——这是保护访问桌面的一步。例如，其他用户登录到不同系统的相同账号，由于第二个用户不在第一个用户组中，所以第二个用户不能写入第一个用户的桌面。

在输入了用户名和口令后，W1nLogon·就调用 LSA，传递登录信息并指定哪一个用于身份验证的包来接收登录信息(如前所述，MSVl—0 执行 W1ndows NT 认证，系统上所有的身份验证包都被定义在 HKLM＼System＼CunentC0n 加 1Set＼Contro 比 sa 目录下的注册表中)。LSA 调用基于这些信息的身份验证包来传送登录信息。

MSVl0 身份验证包获取用户名称和口令信息并向 SAM 发送请求来检索账号信息，包括口令、用户所属的组和任何账号限制。MSVl—O 首先检查账号限制，例如允许访问的时间或访问类型。如果用户因为 SAM 数据库中的限制而不能登录，那么该登录就会失败，并且 MSV1_0 给 lSA 返回一个失败状态。

然后，MSVl0 对比存储在 SAM 中的口令和文件名。如果信息匹配，MSV1_0 生成一个唯一的用于登录会话(调用登录用户 ID 或 LuID)的标识符，并通过调用与会话的唯一标识符相关的 LSA 来创建登录会话，传递用户最终创建访问令牌所需的信息。(一个访问令牌包含用户的 SD、组的 SID 以及用户配置文件信息，如宿主目录。)

接下来，LSA 查看本地规则数据库来了解允许该用户做的访问——交互式、网络或服务进程。如果请求的登录与允许的访问不匹配，那么登录企图将被终止。LSA 通过清除它的所有数据结构来删除最近创建的登录会话，然后向 WmLogon 返回失败信息，接着仍 nLogon 向用户显示相应的消息。如果请求的访问被允许，LSA 会附加某些其他的安全圆(例如“每人”、“交互式”等等)。然后它检查它的数据库来了解这个用户所拥有 ID 的所有被授予的特权，并将这些特权添加到该用户的访问令牌中。

当 LSA 已经得到所有必要的信息后，它将调用执行体来创建访问令牌。执行体为交互式登录或服务登录创建一个首选访问令牌，为网络登录创建一个模仿令牌。在成功地创建了访问令牌以后， LSA 将复制令牌，创建一个可以被传送到 WinLogon 的句柄，然后关闭它自己的句柄。如果需要的话，将审核该登录操作。此时，LSA 把成功信息连同由 MSVl—0 返回的一个访问令牌句柄、登录会话的 LUID 和配置文件信息(如果有的话)返回给 WinLogon。

7.7.6 Windows2000 的活动目录

活动目录是 Windows2000 中最重要的新特性之一。它将大大简化与执行和管理 Windows2000 大型网络有关的任务，同时，它也将改善用户与网络资源间的交互性。

活动目录存储了有关网络上所有资源的信息，它使开发者、管理员和用户可以很容易地找到和使用这些信息。活动目录提供一组单一、一致和开放的接口，用于执行通用的管理任务，如在分布式计算环境中添加新的用户，管理打印机和定位资源等。活动目录数据模型有很多概念类似于 X.500。目录控制代表各种资源的对象，这些资源由属性描述。能够存储于目录中的对象范围在方案(schema)中定义。对于每一个对象类，方案定义了类中的一个实体必须具备的属性、该类能够具有的附加属性以及能够成为当前对象类的父对象的对象类。这一目录结构具有以下主要特性：

- 灵活的分级结构

- 有效的多主机复制
- 粒状安全授权
- 新对象类和属性的可扩展存储
- 通过轻量目录访问协议(LDAP)版本 3 支持实现的基于标准的相互操作性
- 每一个存储中可达到上百万对象
- 集成动态域名系统(DNS)服务器
- 可编程类存储

对于用户、管理员和应用程序代码，使用活动目录更容易在网络上找到资源。用户能够在“查找”实用程序上从“目录”选项中寻找资源，或者在“网上邻居”上从“目录”中浏览资源。应用程序代码也能够从任何程序设计语言中，使用已定义好的 API 集合，搜索或浏览资源。

可编程性和扩展性是活动目录的重要功能。开发者和管理员能够在不需考虑已安装的目录服务的情况下，处理目录服务接口的单一集合，这一编程接口称为“活动目录服务接口”(ADSI)，可通过任何语言来访问。用户也可以使用 LDAP API 访问目录。定义在 RFC l823 中的 LDAP C API，是供 C 语言程序员使用的低级接口。

活动目录也是改进分布式系统安全性的重要基础。

7.7.7 分布式安全性扩展

Windows2000 的分布式安全性，是以公用密钥密码系统为基础的，具有简化域管理、改进性能、集成 Intemet 安全技术等很多新特性。这里介绍 Windows2000 分布式安全服务的一些重要特征。

- 活动目录对所有域安全策略和账号信息提供存储。为多域控制器(以前称为“备份域控制器”)提供账号信息的复制和可用性的活动目录可用于远程管理。其他域控制器上活动目录的多主机复制自动得到更新和同步。
- 对于用户、组和计算机账号信息，活动目录支持多级分层树状名称空间。账号按组织单元分组，而不是由 Windows NT 早期版本提供的呆板的域账号名称空间。域之间信任关系的管理，是通过整个域树间的信任传递得到简化的。
- 创建和管理用户或组账号的管理员权限可以委派给组织单元级。访问权限可由用户对象上授权的单独属性授予。例如，一个特定的个人或组有权重新设置密码，但不能修改其他账号信息。
- Windows2000 安全性包括以 Internet 标准安全协议为基础的新身份验证，Internet 标准安全协议包括用于分布式安全协议的 Kerberos 5 和传输层安全性(TLS)。此外，为获得兼容性，还包括对 Windows NT LAN Manager 身份验证协议的支持。
- 安全通道安全协议的实施，以公用密钥证明的形式，通过映射用户证书到现有的 Windows 2000 账号中，支持客户身份验证。不论用户使用共享的秘密身份验证，还是公用密钥安全性，都可使用公用管理工具管理账号信息和访问控制。
- 除了密码，Windows2000 支持用于相互作用登录的智能卡的选择使用。智能卡支持密码系统，以及对于私人密钥与证书的安全存储，这些私人密钥与证书使强大的身份验证从桌面到 WindowsNT 域都得到实现。
- Windows NT 为那些给其雇员和商业伙伴发行 X.509 版本 3 证书的组织提供 Microsoft 证书服务器。Windows 2000 引入了 CryptoAPI 证书管理 API 和处理公用密钥证书的模块，其中包括由商业证书裁定、第三方证书裁定或者包括在 Windows2000 中的 Microsoft 证书服务器发布的标准格式证书。系统管理员定义在其环境中委托哪些证书裁定，这些证书可以被客户身份验证及访问资源所接受。
- 对于那些没有 Windows 2000 账号的用户，可以使用公用密钥证书进行身份验证，并将其映射到现有的 Windows NT 账号中。已定义的 Windows2000 账号访问权限决定了外部用户能够在系统上使用的资源。使用公用密钥证书的客户机身份验证，以可信赖的证书发放机构的证书为基础，允许 Windows2000 对外部用户进行身份验证。
- 为了管理私人密钥/公用密钥对和访问以 Internet 为基础的资源，Windows 2000 用户拥有易于使用的工具和通用接口对话框。个人安全证书的存储使用以磁盘为基础的安全存储，可以很容易地实现与 Microsoft 提议的工业标准协议和个人信息交换的传输。同时，操作系统也集成了对智能卡设备的支持。

7.7.8 Windows2000 的文件加密

Windows2000 中的加密文件系统(EFS)允许 NTFS 卷上的加密文件的存储。EFS 专门负责安全性方面的问题，这些安全问题是由允许用户从不带有访问检查(如 www.winternals. com 中的 NtRecover)的 NTFS 卷中访问文件的工具而引起的。利用 EFS，NTFS 文件中的数据在磁盘上加密。所用的加密技术以公用密钥为基础，作为一个被集成的系统服务运行，由此使得该技术易于管理，不易受到攻击，对于用户非常透明。如果一个试图访问已加密 NTFS 文件的用户拥有此文件的私人密钥，那么该用户将能够打开这个文件，并作为一个正常的文档进行工作。系统将直接拒绝没有私人密钥的用户的访问。

EFS 与 NTFS 紧密集成。EFS 的驱动程序组件以核心态运行，使用非页交换区存储文件密钥，确保这些文件密钥不会将其变成页面调度文件。以下是组成 EFS 的关键组件：

- Win32API 这些 API 提供程序设计接口，这些接口用于加密纯文本文件、解密或恢复密码文本文件、导入和导出已加密文件(事先并不对它们进行解密)。
- EFS 驱动程序 EFS 驱动程序被分层在 NTFS 的顶部。EFS 驱动程序通过与 EFS 服务进行通信，请求文件密钥、DDF、DRF 和其他密钥管理服务。它将这些信息传递给 EFS 文件系统运行时库(FSRTL)，用来透明地执行各种文件系统操作(打开、读取、写入和附加)。
- FSRTIJ 标注 FSRTL 是 EFS 驱动程序内部的一个模块，EFS 驱动程序通过 执行 NTFS 标注，在已加密的文件和目录上处理各种文件系统操作(如读取、写入和打开)，当系统从磁盘写入或读取时，EFS 也通过该方式进行加密、解密和恢复文件数据的操作。尽管 EFS 驱动程序和 FSRTL 作为单一组件被执行，但它们仍然不能直接通信。它们使用 NTFS 文件控制标注机制相互传递信息，这样可以确保 NTFS 的参与者存在于所有的文件操作中。使用文件控制机制实现的操作包括以文件属性写入 EFS 属性数据(数据解密字段[DDF]和数据恢复区字段[DRF])，将在 EFS 服务中已计算的文件密钥传给 FSRTL，这样该密钥就能够安装在打开的文件描述表中。在磁盘文件数据的读写上，使用这一文件描述表，可以透明地加密和解密。
- EFS 服务 EFS 服务是安全子系统的一部分。它在本地安全裁定服务器和核心态安全参考监控器之间使用现有的 LPC 通信端口与 EFS 驱动程序进行通信。在用户态中，EFS 服务接口连同 CVptoAPI 提供文件密钥并生成 DDF 和 DRF。

文件加密可以使用任何对称加密算法。EFS 第一版将采用 DES(数据加密标准)作为加密算法。以后的版本将允许改变加密方案。

7.7.9 安全配置编辑程序

安全方面的另一个关键增强是新的安全配置编辑程序。这一编辑程序的主要目标是为以 Windows2000 为基础的单个站点提供系统安全管理。这一编辑程序允许管理员配置和分析系统安全策略，如：用户登录系统的方式和时间、密码策略、整个系统的对象安全性、审核设置、域策略等。使用该编辑程序，也能够在用户、文件、目录、服务以及注册表上更改安全设置。

为满足 Windows2000 中的安全性分析需要，安全配置编辑程序将提供在宏水平上的分析。这一编辑程序被设计成提供与安全相关的所有系统方面的信息。对于整个信息技术(1T)基础构件，安全管理员们可以查看这些信息，并执行安全风险管理。

就所涉及的系统组件和可能要求的更改等级而言，在以 Windows2000 为基础的网络中，配置安全性的过程是复杂而精细的。安全配置编辑程序允许管理员定义很多配置设置，并使它们在后台生效。运用此工具，能使配置任务分组和自动化：为了配置一组机器，配置任务不再需要多次反复按键并重复访问大量的不同应用程序。

安全配置编辑程序的设计目标并不是想替代用于处理系统安全的不同方面的系统工具——如用户管理器、服务器管理器、访问控制表编辑器等。其目标在于，通过定义一个能够解释标准配置模板并在后台自动执行所请求的操作的引擎，来实现这些系统工具。在任何必要的时候，管理员都能够继续使用现有的工具更改个人安全设置。

7.8 实例研究：UnixWare 2.1/ES 操作系统

UnixWare 2.1/ES 操作系统是具有高安全性的开放式操作系统，安全等级为 B2 级。它的安全措施和功能主要有：

- 标识与鉴别：系统中的每个用户都识置了一个安全级范围，表示用户的安全等级，系统除进行身份和口令的判别外，还进行安全级判别，以保证进入系统的陶户具有合法的身份标识和安全级别。
- 审计：用于监视和记录系统中有关安全性的活动。可以有选择地设置哪些用户、哪些操作(或系统调用)、对哪些敏感资源的访问需要审计。这些事件的活动就会在系统中留下痕迹，事件的类型、用户的身份、操作的时间、参数和状态等构成一个审记记录记入审记日志。通过检查审记日志可以发现有无危害安全性的活动。
- 自主存取控制：用于实现按用户意愿的存取控制。用户可以说明其私有资源允许系统中哪个或哪些用户以何种权限进行共享。系统中的每个文件、消息队列、信号量集、共享存储区、目录、和管道都可具有一个存取控制表，说明允许系统中的用户对该资源的存取方式。
- 强制存取控制：提供了基于信息机密性的存取控制方法，用于将系统中的用户和信息进行分级别、分类别管理，强制限制信息的共享和流动，使不同级别和类别的用户只能访问到与其相关的、指定范围的信息，从根本上防止信息的泄密和乱访问现象。
- 设备安全性：周于控制文件卷、打印机、终端等设备 I/O 信息的安全级范围。
- 特权管理：让系统中每个用户和进程只具有完成其任务的最数特权。系统中不再有超级用户，

而设若干系统管理员/操作员共同管理系统，他们每个只有部分特权且相互间有所约束。

- 可信通路：这种机制为用户提供一种可信登录方式，防止窃取合法用户的口令以登录到系统中。
- 隐通道处理：用于堵塞隐通道或降低隐通道的带宽，并审记其使用情况。
- 网络安全：实现安全系统之间及安全系统与非安全系统之间的网络互通，可进行网络身份认证、控制进入、防火墙、网络审记等功能。

CH8 网络与分布式操作系统

伴随着计算机技术和通信技术的发展，当今已经进入网络化时代。对计算机网络和分布式计算的支持是现代操作系统必不可少的组成部分。目前，所有操作系统都具有网络操作系统的特征，全面支持接入计算机局域网和 Internet 网，并且有很多操作系统也具备分布式系统的特征，在不同的程度上支持分布式计算。分布式系统是一组松散耦合的计算机系统通过通信网络连接起来的，因此网络技术是分布式计算的首要的前提技术，两者的区别在于：

- 计算机网络系统。用户在通信或资源共享时必须知道网络上的计算机及资源位置，即访问资源时需要通过以下两种方法之一：1）登录到相应的远程计算机中；2）与显式指定的远程计算机直接传输数据。
- 分布式计算系统。用户在通信或资源共享时并不知道多台计算机的存在，队员成资源的访问就像访问本机资源一样。

本章首先简介计算机网络、网络操作系统和分布时操作系统，然后分别介绍通信技术、远程文件系统技术和分布式协同技术。

8.1 计算机网络

网络是硬件、系统软件和应用软件的组合体。网络设备本身对计算机没有太大的意义。但是，通过网络设备将多台计算机或通信子网连接起来可以形成一个通信环境。在这个通信环境下，运行于不同机器上的进程可以实现相互通信。为了使网络更有效，可以将网络中实现的功能分出一定的层次。ISO OSI 结构模型就提供了实现这些功能的框架。

本节首先讨论：现代计算机网络如何从传统的使用串行接口和调制解调器实现的计算机通信，发展到由复杂软件支持的网络。在介绍 ISO OSI 模型后，我们将回顾网络硬件的特性。然后介绍网络中由操作系统实现的部分。最后是计算的客户机/服务器模式，以此说明，网络应用的需求驱使操作系统支持网络。

8.1.1 从计算机通信到网络

现代网络是基于电信技术的。一个串行接口可以把二进制字节发送到外设，也可以从外设接收二进制字节。计算机之间的通信就利用了串行口的这一特性。如图 8-1 所示，将调制解调器接在计算机的串行通信口上，一台机器的调制解调器把串行口控制器送来的数字信号转化为与普通电信网络中信号相类似的信号并送出，再由某台远程计算机的调制解调器把这种信号转化为原数字信号并送到与它对应的控制器。这样远程计算机上的程序就可以获得需要的数据。

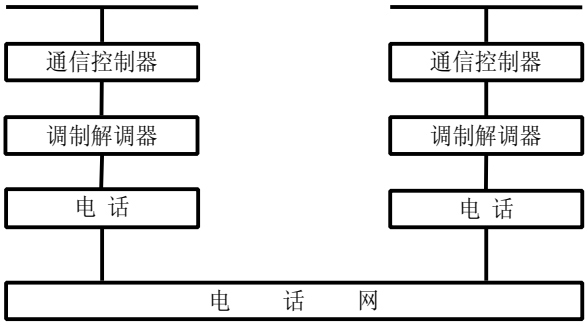


图 8-1　用电话网连接计算机

这种点到点、面向特性的通信技术是现代网络的基础。任何一台附有调制解调器和电话线路的计算机都可以和另一台带有调制解调器和电话线路的计算机建立连接。一旦电话连接建立，两台计算机之间可以通过一组预先设定的网络通信协议（或者只是简单协议）规定交换信息的语法和语义，从而进行信息的交换。

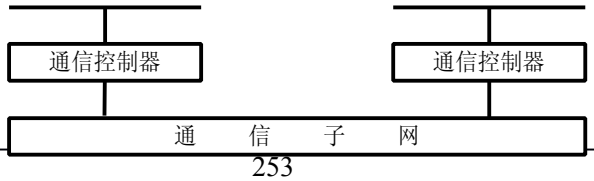


图 8-2　用数据通信子网连接计算机

现代网络控制器需要遵循多种协议的要求，以特定结构的字节流与 I/O 设备进行交互。如今，大多数网络协议在串行网络中使用信息块发送或接收信息。这种信息块被称为“包”。如图 8-2 所示，计算机网络技术将原来的调制解调器、电话连接和电话转换系统的实现方式替换为专门的数据通信子网实现方式。数据通信子网也允许发送方将“包”传递到子网中的任何一台主机。

和其他设备一样，网络控制器的软/硬件接口由设备驱动程序和管理器的硬件来实现。当然，网络管理器需要和通信子网，而不是和其他的本地设备（如存储器或终端设备）相连。

不难看出，计算机网络事实上是通过通信子网把物理位置上分散的多台计算机和设备连接起来，并通过网络通信协议、网络操作系统和网络应用程序向用户提供网络上的通信、设备共享和信息共享。

为了协调收发双方的行为，需要利用通信协议实现许多功能。如，一台机器如何探知另一台机器发来的信号；浮点数传输时应该使用什么格式；等等。

可以将文件系统和网络协议作一些比较。文件系统中的某些功能可由应用软件实现，在网络协议中也有一些功能可以由应用来确定。在文件系统中，一些基本的模块是由硬件和操作系统实现的；同样在网络协议中的关键部分也是由硬件和操作系统实现的。

一个网络操作系统将向用户提供在网络上的进行通信、设备共享和信息共享的手段，当然用户必须知道网络上存在多台计算机，并通过远程登录或与远程主机显式传输数据的方式进行。

8.1.2 通信子网

通信子网是一个逻辑概念。以它为媒体，与网络相连的某台主机可以发送一个字节块（通常称为“包”）到网上的任一台主机。主机的进程先将要发送的信息组织为逻辑上的“包”（“包”中附有地址作为发送的目的地），然后将“包”写入通信子网。作为目的地的主机上的进程从通信子网中读取“包”中字节，从而获得需要的信息。这样的通信子网被称为多点包网络(multidrop packet network)。

多点包网络的目标是提供一种有效的方法来实现比电话网更快更廉价的网络。它提供多种功能：

- 设计并完成有效的“转换结构”-- 一种在与网络相连的主机之间传送信号的中介形式；
- 将“包”转换为物理网络中要求的格式；
- 将从物理网络中得到的信息转换回“包”；
- 将“包”直接发往目标主机（称为包寻址）。

以太网是运用得较广泛的多点包网络。在以太网中，字节以串行的方式通过共享的通信媒体。1980 年，以太网初次建成。当时的物理媒体是一个同轴电缆（类似于电视电缆），现在则是两股扭在一起的电线（类似于电话线）。软件用从以太网控制器读写大小不同的“包”的方式使用以太网。在一次写操作中，网络控制器接收到“包”的一个拷贝并将之放在缓存中，由网络设备将其内容逐位转换。现在，当主机间物理连接的电缆长度为 1 公里时，以太网的传输速度已经达到 1Gbps。

8.1.3 网络通信协议

70 年代后期，研究者们开始使用网络。他们很快发现在通信子网中，不同的应用需要不同的功能来支持，这意味着子网需要支持的功能比预期的更多。同时，他们发现有许多附加的功能可以在很大程度上改善原来的信号发送过程。这些附加功能有：

- 控制信息传输率；
- 实现位于不同网络中的主机间的通信；
- 允许“包”文件像流式文件一样，在包的流中包含字节流；
- 确保信息在网络传输时的完整性；
- 确保信息在网络传输时的安全性；
- 为在网络中传输的 IPC 进程提供一个标准的行为模式；
- 适于传输文件；
- 允许用户由本机访问网上另一台主机；
- 将网上某台主机上的过程作为本机的过程运行；
- 按照不同机器的要求转换信息表达方式（不同计算机对字的规定不同，有的是 2 字节/字，有的是 4 字节/字）。

此外，不同的应用对网络功能有不同的需要，有的要求实现上述所有的功能，有的只要求实现部分功能。这自然让人想到引入层次结构的概念，将这些功能分为若干层次，而每个层次则是一些功能的集合。层次之间存在偏序关系（可以定义层次 L1 高于 L2 当且仅当 L2 对应的功能集合是 L1 的功能集合的

子集）。那么，基于较高层次上的操作应用可以使用任何低层次中实现的功能。在网络中，一个层次中的功能的实现对应于一个网络通信协议。基于同一个协议的软件之间可以直接实现通信。

协议并不是一个新的概念。程序员需要使用某种协议在它的程序间实现通信。例如，程序通过函数名调用函数；在函数原型中明确规定了参数的传递方式，即参数个数和参数类型。在读写文件时也需要遵循某种协议。将数据写入文件时必须按照该文件的组织方式写入，这样，在从文件读取数据时就可以按照既定的组织方式正确读取。

网络通信基于两个独立的进程。这两个进程必须同时存在且彼此有通信的要求。但是，仅满足这两个条件不能实现成功的通信，这两个进程还必须遵循相同的语法和语义规范。假设进程 P1 要将某个文件传给进程 P2。首先，P1 和 P2 要实现同步，也就是说，当 P1 将文件发往 P2 时，P2 已经准备好从网络中接收文件。接下来考虑对传送的内容的要求。数据单元要如何传送？有可以用的数据格式吗？可以有多种数据传输格式吗？如果计算机对浮点数的表达方式不一致，如何确定接收方收到的字节可以被解释为正确的数据？为解决诸如此类的问题，我们要求通信双方遵循同一个协议，由该协议规定通信中的语法和语义。

文件传输只是通信的一部分。进程之间有交换信息和请求远程服务等需要。针对不同的应用的有不同的网络协议。随着网络的发展，逐渐形成国际标准——ISO OSI 结构(OSI 即 Open System Interconnection，开放式系统互联)。这个模型规定了协议的大体内容和部分细节要求。以下详细介绍 ISO OSI 结构。

8.1.4 ISO OSI 网络结构模型

ISO OSI 结构模型是现代网络中定义网络协议的主要模型。作为标准结构，该模型已被大多数的开发者和使用者接受，当然，实现的细节可能有所不同。因此，有必要考察模型的发展史，以便理解它为什么有今天的形式。

在 60 年代后期，随着网络技术的发展，分布式计算开始在一些应用领域取得成效。1975 年，为了支持美国在国防上的应用需要，美国国防部建立一个远程网络体系—ARPA 网。同时，在欧洲，支持商业应用的 X.25 网成为一种较为可行的技术。

到了 70 年代后期，ISO 开始推广关于网络通信的 OSI 结构模型的第一个草案。该草案对 X.25 网产生巨大影响。而在美国，ARPA 网已经成为网络发展的主导力量。

1980 年，以太网在商业领域证明了局域网的可行性。X.25 和 ARPA 网都是针对数百里范围的通信的；以太网则适合于一公里以内的局域网。ARPA 网可以根据网络的使用情况以可变速率传输信息；以太网在本地的传输速度可达 3 Mbps，在商业应用中可达 10 Mbps。现在，以太网的传输速度已经达到 100 Mbps。

IBM 系统网络结构(System Network Architecture，简称 SNA)也是 70 年代时一种主要的商业计算模型。X.25、ARPA 网和以太网的协议是公开的，而 SNA 协议却是非公开的。SNA 的重要性在于它使用更加可行得技术建立数据网络。到了 1980 年，SNA 引入令牌环局域网，作为开放式环境下除了以太网以外的另一种选择。

在 1980 年，DEC、施乐和 Intel 联合发布了商业以太网。他们改进 ARPA 网的中间层的协议来支持他们的底层局域网。几乎是同一时间，IBM 发布了它的令牌环局域网。IEEE 标准委员会融合以太网和令牌环形成一个可靠的商业局域网标准，即 ISO OSI 模型。IEEE 802 作为一个草案，在 80 年代成为局域网的标准。虽然以太网和令牌环是不能直接相连的，但是在 IEEE 802 标准中，以太网和令牌环都是允许的。也就是说，一个基本结构模型，如 IEEE 802，可以有两种不同的实现。一个基于 IEEE 802 的网络可以用以太网来实现，而另一个则可以用令牌环，两者不能直接相连。在 90 年代早期，IEEE 802 草案成为局域网通信的标准 ISO 8802。

由此可见，80 年代网络技术的发展有三条主线(见图 3)。ISO OSI 模型奠定了标准网络协议的基础。从本质上看，它是从 X.25 网发展起来，受到 ARPA 网的影响。现在，它已经是网络中占统治地位的协议模型。由于对 ARPA 网中间层协议进行改进(尤指 TCP,UDP 和 IP)，以太网很快获得成功。当它成为 IEEE 草案标准时，令牌环是它的主要竞争对手。经过数年的竞争，IEEE 802 委员会将以太网(IEEE 802.3),令牌总线(IEEE 802.4)和令牌环(IEEE 802.4)融合，形成一个单一的草案标准。ISO 8802 就是 IEEE 802 的国际版本。

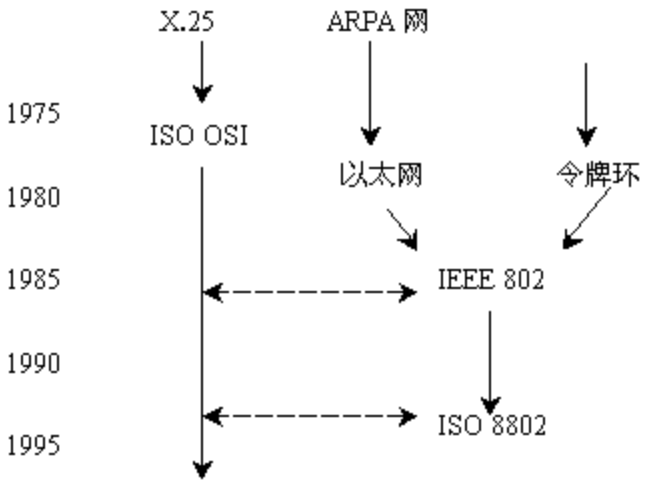


图 8-3　网络的发展

值得惋惜的是，TCP/UDP/IP 协议没有按照 ISO OSI 模型的要求去做。随着以太网和令牌网在商业领域取得的巨大成功，比较可行的办法是或者修改 IEEE 802，或者改动 ISI OSI，以便 IEEE 802 网络可以符合国际标准。图 8-3 中的虚线就是开发者们将两者合二为一的尝试。直到今天，最终用户联盟仍然强烈要求将 ISO OSI 模型的框架和 TCP/UDP/IP 的使用联合为一个完整的部分，因为很多成功的网络产品都使用 ARPA 网络协议。从技术角度看，虽然 TCP/UDP/IP 组织经常利用 ISO OSI 模型作为网络策略的一部分，但是他们并不想完全照搬该模型。

ISO OSI 结构模型定义一个功能集合。其中的功能针对所有的网络通信方式。接着，它将这些功能划分出层次结构。每个特殊的网络应用，比如文件传输，都有各自的协议集合，称为协议栈。协议栈按照不同网络层次的要求将网络协议分类。根据 ISO OSI 结构模型，协议栈可以分为以下七个层次：

1)物理层。物理层位于模型的最底层。在 ISO OSI 模型中由物理层定义字节是如何编码和传输到另一台机器的。RS-232 异步串行通信协议用于连接计算机主机和终端设备、调制解调器或打印机。尽管一般将它作为网络协议，RS-232 更像一个物理层协议。而以太网中，载波检测和冲突检测都属于物理层网络协议。

2)数据链路层。建立在物理层的基础上。在以太网和令牌网中，数据链路层与物理层相连，由硬件实现。数据链路层上的协议定义了帧—“包”在数据链路层中的格式。帧结构中包括帧头，数据块和帧尾。在数据链路层上，网络中主机间以交换帧的方式通信。

3)网络层。网络层为通信节点创建极大的地址空间，称为英特网地址空间。这为网络的使用提供方便，从而有助于网络的发展，最终导致英特网的出现。在英特网中，每个节点就是一个网络。而信息则是以“包”的形式在英特网中传输。网络层总是作为操作系统的一部分来实现。

4)传输层。传输层在网络层的基础上提供各种应用接口,包括利用块、字节流和记录流通信。传输层一般由操作系统实现，有时也会由系统软件实现其中的某些部分。

5)会话层。会话层提供特殊的进程间通信的方法，从而扩展了传输层的功能。例如，关于网络消息的协议或远程进程的协议就是在会话层实现的。可以说，会话层是面向应用的。

6)表示层。表示层定义数据的提取和表现。

7)应用层。针对 ISO OSI 模型下分布式计算的应用软件。因为应用层试图面向所有的应用领域，所以在应用层没有必须遵循的标准。在 ISO OSI 模型中，它的存在只是为了说明应用程序在该层次结构中位置。

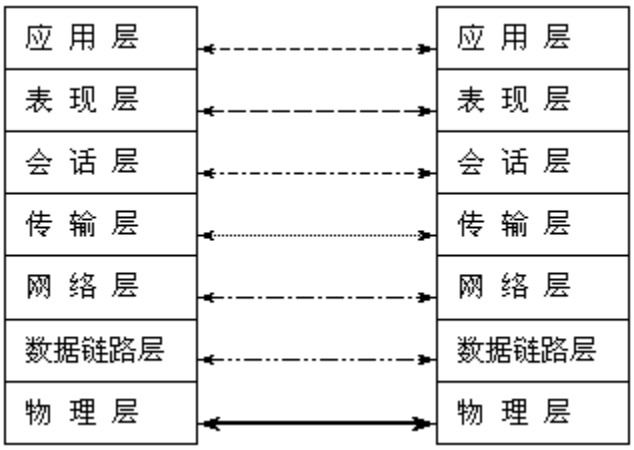


图 8-4　ISO OSI 模型下两个通信进程间的联系

ISO OSI 模型中的层次是将通信时需要的各种功能抽象化后得到得分层结构。每个进程都有各自的协议栈，其中包含上述结构中为实现本次网络通信所需要的协议。图 8-4 说明了在 ISO OSI 模型下，两个通信进程间的联系。对于这两个进程而言，只有物理层是真正用来传输信息的。数据链路层在逻辑上将信息以帧的形式发往接收方。实际上，它把帧转换为物理层所支持的格式，然后把转换后的“帧”送到物理层，由物理层将其发送至接收方的物理层，再由接收方的数据链路层将其转换回帧格式。由此可见，在两个数据链路层之间存在着同位通信。在图 4 中，用粗实线强调信息在两台主机间传输时，仅在物理层实现物理意义上的传输。在其余的层次有逻辑意义上的通信。

在网络层，“包”的交换要经过以下步骤：先把网络层意义的“包”转换为数据链路层中的帧，再将帧转换为物理层支持的格式，由物理层将信息发送至接收方。接受方先把信息转回帧格式，再由帧格式转换为“包”。在同位通信中，传输层、会话层、表现层和应用层都采用这种技术。应用层使用建立在表现层接口上的抽象机制，如远程文件服务器接口，远程过程调用接口等等。

ISO OSI 模型作为最主要的模型，反映各个网络及网络产品的生产厂家之间达成的共识。该模型的构架已被使用多年，但是具体的协议却在不断发展中。例如，ARPA 网的 IP 协议不是标准的一部分，但它是 ISO OSI 模型中网络层的主要实现途径。由于 TCP(以及 UDP)对传输层的适应和 IP 对网络层的支持，传输层及传输层以下各个层次的接口总可以保持不变。

现代操作系统必须支持各种数据链路层和物理层网络控制器，以及网络层和传输层的实现。

可以看到，在 ISO OSI 协议栈中，ARPA 网协议是如何使用的。TCP 和 UDP 相当于传输层上的协议，IP 则是网络层协议。ISO OSI 传输层接口（Transport Layer Interface，简称 TLI）提供的功能往往是 TCP 也可提供的。如图 8-5 所示，抽象的 ISO OSI 协议栈可以由 ISO OSI 会话层协议、TCP、IP 和以太网来实现。在任一种实现中，ISO OSI 会话层的使用者将获得同样的功能支持，与具体的实现无关。当然，以太网的数据链路层的实现不能直接和 X.25 数据链路层的实现相连，但是，利用以太网的网络层软件可以直接在抽象的协议栈上使用。

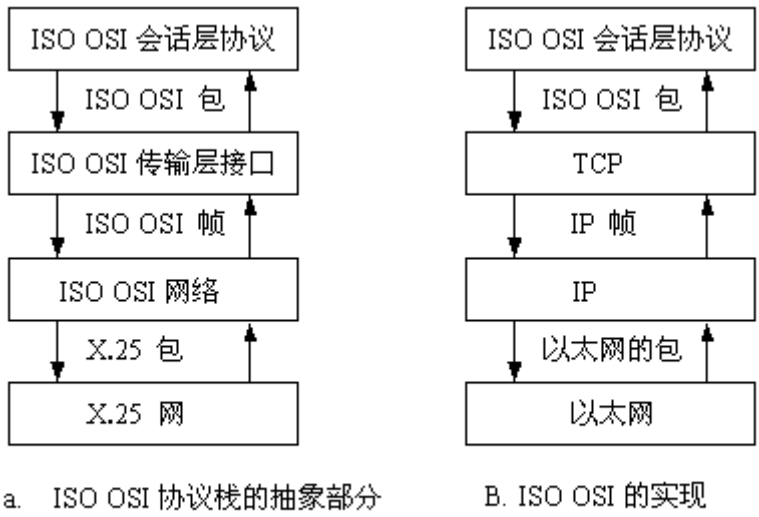


图 8-5　在 ISO OSI 协议栈中使用 TCP/IP

8.2 远程文件系统

本节通过内存文件抽象来考虑操作系统如何利用网络访问网络中计算机的二级存储设备。目的是帮助大家获得以网络为基础的计算机模型的整套看法以及考虑各种操作系统策略是如何在网络的计算机中分布文件系统使一个应用程序在一个远程文件的使用就象是本地计算机一样。第一步策略允许文件管理者使用远程机器上的存储设备。第二步策略要求文件管理的功能分布在本地和远程计算机之间。第三步策略是让操作系统透明的从远程机器上拷贝文件，保证多个拷贝的一致性，当文件被关闭时把拷贝还原到初始的那台机器。远程文件技术是由操作系统提供的分布式计算的基础支持。

8.2.1 通过网络共享信息

网络技术最明显的应用就是让操作系统使得在一台机器上的一个进程能够使用另一台机器的一级或二级内存。操作系统抽象二级内存的使用使得内存能够通过使用虚拟内存模型来被访问，同时期的操作系统应该允许进程从通过网络可到达的远程计算机的内存读出或写入数据。

冯.诺依曼计算机通过给各自提供明确的、不同的接口来区别一级和二级内存。一级内存接口由计算机结构自身定义，并由地址空间组织来扩展。虚拟内存系统在需要时通过把信息载入到物理一级内存，来使用一级内存接口提供到二级内存存储设备的途径。

有两种二级存储接口，一个通过存储设备实现，另一个通过文件系统实现。存储设备接口没有被广泛使用，除了作为文件系统的一层。文件系统接口被应用程序用来访问二级内存。

进程如何通过远程的一级和二级内存相互作用？即使在基本的冯.诺依曼结构中，一个进程必须使用分离的接口来访问一级和二级内存。在这些情况下，一个网络环境的自然目标是使一个进程能够通过本地的一级内存接口访问远程的一级内存，通过使用本地的二级内存接口访问远程的二级内存。理想情况下，一个进程应该能够将一个内存地址写到一级内存接口中去，而使得这个地址被存储在另一台机器的相联系的一级内存中。它同样应该能够读取一个远程文件，好象这个文件被存储在本地文件系统中。

在 APPA 网成为一个可成立的网络后，软件被发展来实现远程文件引用的基本结构框远程文件架。最早的系统便利使得一个用户能够在网络上从一台主机到另一台主机拷贝文件，通过使用明确的命令。远程文件技术的发展使得现在能够通过使用本地文件管理接口来访问远程文件。到 1990 年，操作系统研究者已经开始努力工作，以能够通过使用一级内存接口来实现各种远程一级内存。随着这条操作系统对通过网络访问内存支持的道路，这里主要考虑支持远程文件访问的操作系统技术。

8.2.1.1 显式的文件拷贝系统

文件是进程间共享的典型的单位。信息可以被一个进程创造，被写到一个文件中，然后被其他进程使用。信息也可以通过在网络中显式的从一台机器拷贝文件到另一台以实现在机器间的共享。在典型的广域网联结系统中，例如 APPA 网，显式的文件拷贝操作是实现机器间共享的最普通的机制。当本地机器上的一个进程需要远程计算机中某个进程的信息时，远程进程将信息写入某个文件，然后用户显式的把文件从远程计算机拷贝到本地机器。

显式操作在一个常规的文件管理器中被用作以一个简单的 shell 命令在一个系统的目录中把文件从一个位置拷贝到另一个位置。最早的网络系统软件提供了一个便利使得用户可以把一个文件从一台机器的

文件系统中拷贝到另一个。可以通过使用操作系统的点对点连结功能或传输层软件与远程机器相连接，然后让一个在远程机器上的代理进程取回目标文件的一份拷贝并把它写入连接来实现。当连接的本地端把请求文件拷贝操作的信息发送出去后，它就在连接中等待代理进程开始通过连接传送文件的拷贝，本地端接收拷贝并把它存储在本地文件系统中备用。APPA 网的 ftp 命令和 UNIX 中的 uucp 命令分别反映了使用网络和通信的便利。

在异构的网络中还有一些其他的手工的文件传输包（异构网络就是一个有不同类型主机的网络）。ISOOSI 文件传输、访问和管理协议是 ISOOSI 手工文件传输的标准机制；telnet 和 ftp 在同时期的计算机网络中也同样被广泛的应用。

1）APPA 网文件传输协议

文件传输协议可以从一个用户界面激活，例如 UNIX 系统中的 ftp 程序，来使用 TCP/IP 在一个网络的主机间拷贝文件。ftp 程序是 FTP 的一个用户界面，每一个支持 FTP 的主机开始一个服务进程来接收服务请求。当一个应用进程试图使用 FTP 时，它在本地机器上执行 ftp 客户代码。

FTP 为文件传输使用一个控制连接和一个数据连接。服务器希望初始服务请求到达已知的 21 号端口，这样当一个客户希望使用 FTP 时，它的 TCP 连接请求被定向在<.net,host,21>。当连接被打开，客户端和服务器使用控制连接来交换接下来的命令和控制响应。

当一个文件被拷贝和客户端取得一个目录文件列表时，数据在客户端和服务器之间传输。每次客户端声明一个命令来引起这种数据传输，一个数据连接被打开来容纳传输。在一个 FTP 会话期间，数据连接可能被打开、关闭多次，依赖于命令的属性。

2）UNIX uucp 命令

uucp（unixtounixcopy）是一个使用串行连接设备和拨号 modem 来交换文件的程序；uucp 的用户界面是 uucp。Uucp 使用一个由机器系统管理员定义的路由表来调用另一台机器上的 uucp 程序，来往返拷贝文件。

UNIX 的 uucp 命令与本地的 cp 命令有相同的表现形式，除了拷贝源或目的地是一个远程机器的文件名。远程文件的格式是这样的：

System_name_1!system_name_2!…system_name_N!n_pathname

每一个 system_name 是存储在 uucp 路由表中的一个 UNIX 机器。如果远程文件名中有多个 system_name，它们定义了一条通过 system_name_1 到 system_name_2 一直到 system_name_N 的路径，n_pathname 定义了文件在机器 system_name_N 的文件系统中的位置。Uucp 操作在每一个系统中都被证明，这样 uucp 通常用来从一个远程主机读取文件。如果试图写一个文件到一台远程主机，保护设置可能要求用户只是登录到远程机器然后执行一个 uucp 到本地机器上来。

Uucp 已经被在现代化的网络环境下执行的新命令所取代，包括 rcp 和 rdist。在这些命令中，通过提供一个 DNS 主机以及那台机器上的绝对路径名来访问一个文件。

8.2.1.2 隐式的文件共享

前面描述的那一类显式文件传输要求另一个界面，或者至少普通文件界面的扩展，正如 FTP 和 uucp 的例子所示。隐式文件共享更好的利用了现有的文件和二级存储器接口。

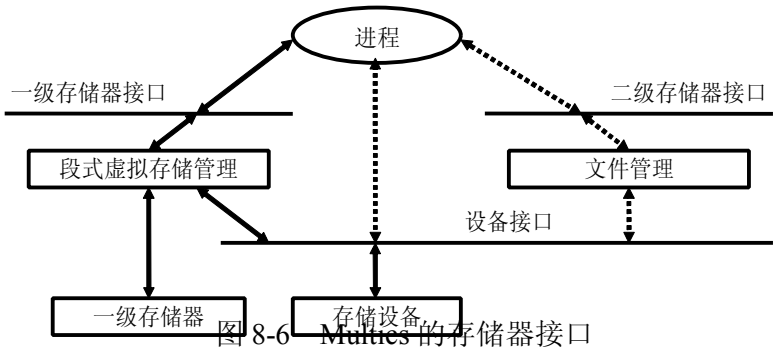


图 8-6 Multics 的存储器接口

分离的一级和二级存储器接口，以及隐含的分离的地址空间，反映了一级和二级存储器在内存访问时间上的巨大不同。二级存储器的访问速度通常要比一级存储的速度慢上几个数量级。当然，如果一级存储器采用虚拟存储器技术，两种接口访问存储在存储设备上的信息的速度差不多是相同的。然而，虚拟内存的实现将要处理从虚拟设备和向虚拟设备恢复信息的细节，并要保存一级存储器中正在被访问的信息。如图 8-6 所示，Multics 的存储器接口使得一个程序员能够使用一个扩展的一级存储器接口来访问所有的一级和二级存储器。一个一级存储器地址由一个段名和一个段偏移组成，段名可以被映射成文件名。操作系统中的内存管理器访问时自动加载段名，并把物理地址绑定到一级存储器接口中生成的两部分地址中。所有存储在存储设备上的信息能以段名和偏移访问（这些系统也支持一个文件接口来访问二级存储设备，因此文件和设备接口在图中被解释成冗余接口）。

理想情况下，当一个存储在远程站点的信息被一个进程访问时，对应的程序可以使用两个常用的存

储器接口中的一个。使用已有的接口是远程文件的主要特征，因为这一过程使用二级存储器接口来访问存储在远程机器上的文件。图 8-7 反映了通过远程服务器重新使用二级存储器接口的总的设计策略：服务器是一个远程磁盘设备，它通过一个与本地磁盘驱动器类似的接口来提供一个虚拟磁盘驱动器（一个安装在远程机器上的物理磁盘）。远程服务器也可以是一个作为本地文件系统扩展的一个远程文件，并使用一个新的内部的远程文件接口。两种选择都可以被使用在同时期的计算机中。接下来讨论究竟选择哪种进程的标准。

8.2.1.3 远程存储接口

设备驱动接口主要是由系统软件而不是应用软件所使用。根据图 8-7 的内容，它表示应用程序访问存在于本地文件接口上的本地存储设备。根据一致性的原则，远程磁盘和文件接口也使用本地文件接口，与远程存储访问有很小或没有区别。它们通过重新实现本地文件管理器来使得它既能够以通常的方式访问本地文件，又能够通过使用远程服务接口来访问远程存储设备。

低级的文件系统接口通过远程文件服务比在接口有着更为复杂功能的文件系统容易实现。文件被表示成通过逻辑读写头访问的字节流。文件的组织和状态保存在文件描述符中。文件上的基本操作是打开、关闭、读、写和查找。

因为低级的文件接口被用来控制本地和远程文件，对一个远程文件的访问被说成是透明的。然而，关于透明性有一点注意：当本地和远程文件的文件操作命令相同时，文件名来标识它是本地的或是远程的。因此，透明性应用在接口的功能而不是对一个应用可提供的文件的实际路径名。通过利用 UNIX 格式的装入命令，远程文件名可以被显示成本地文件名。这意味着文件的位置对于程序员和用户来说是完全透明的，而系统管理员需要通过合适的装入命令来解决这个问题。

远程磁盘一直定位的透明性，意味着对包含文件的远程存储设备的访问是完全由文件系统处理的。从应用程序的角度，对远程磁盘的访问和对本地磁盘的访问是一样的，因为都使用本地文件接口。在极端情况下，本地机器可以不装备磁盘，意味着每次磁盘或文件访问都是到一个远程机器的。

例如，每当一个交互式用户运行程序时，它被从一个远程机器的磁盘上载入。当编译器编译一个程序时，源程序从远程机器的磁盘载入，可重定位的模块被写回远程机器的磁盘。远程磁盘的位置由系统管理员确定,从逻辑上看，远程访问对于用户来说是很透明的，即使是关于名称。用户可以预测的远程磁盘访问的主要时机在于被要求访问在一个远程硬盘上实现的文件。虽然这也可能因为本地磁盘、远程磁盘和网络的相对速度也变得透明。

8.2.1.4 任务的分布

使用本地文件接口得隐式共享文件通常认为系统实现如图 8-7 所描述的逻辑过程。此图确认了一个本地服务器接口，这个接口从功能上说可以运用到使用网络控制器和协议的远程机器上去。图 8-8 是对图 8-7 里显示的流程的阐述，尤其在一个客户机服务器结构中，应用程序在客户机上执行并且远程文件设备作为服务器使用。客户机上的操作系统实现了为在远程访问模块下实现远程文件的部分功能。实际上，这个远程服务器的本地部分是远程文件服务器的重要部分。其方式就是使得本地文件管理者和没有远程文件功能的服务器尽量一致。这就意味着，一个不使用远程文件的配置不需要使用和远程访问结合在一起的功能和空间。因此，远程访问模块是操作系统的一部分，这个 OS 负责实现远程文件管理，网络的交互反应，传送和其他使用远程文件必须的特征。但是，它会允许一个应用过程不需要了解远程访问的细节就可以轻易的激活文件服务器的客户部分。

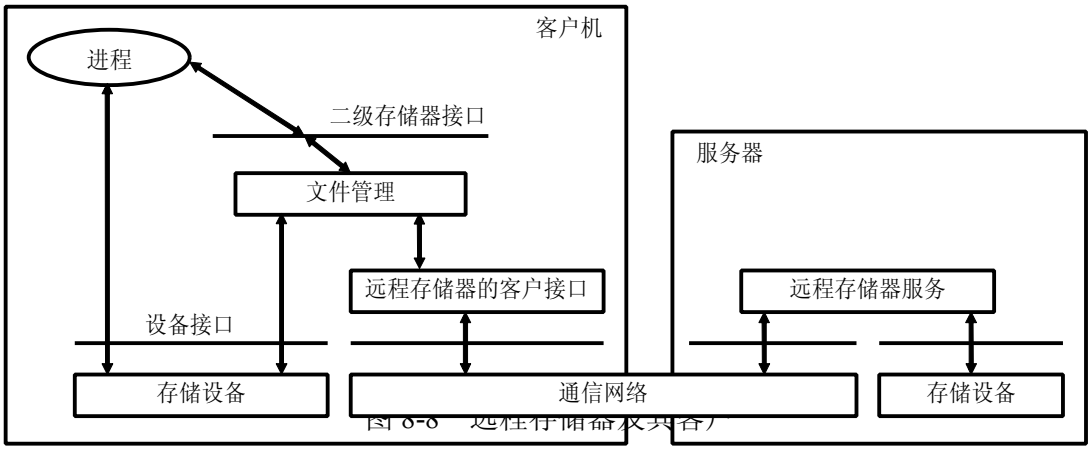


图 8-8 远程存储器及其客户

在服务器上的远程存储模块从客户远程访问模块中接收存储命令，在其存储设备上实现这个操作，然后把这个结果返回给客户远程访问模块。在这个管理过程中存在的最主要的问题是在功能上应当怎样在客户机器的远程访问模块和服务器的远程存储模块上划分普通的文件系统。一旦划分功能的策略被决定下来，那么在设计中就必须回答其他问题，包括如下一些。一个网络协议怎样才能最好的适应于这个已分配的文件管理器。一个已分配的文件管理器可以提供一个可接受的功能层。在这样一个框架中怎样确保网络和系统的可依赖性。这些问题构成了在远程文件系统中最主要的设计问题的特征。

在过去的 15 年里，在分割文件系统功能中，三个基本策略被广泛应用。

一、使得服务器实现远程磁盘功能，这意味着客户服务器接口和一个拥有磁盘驱动器的本地文件系统所具有的接口类似。在这种情况下，一个客户机器将磁盘块读出并写入磁盘驱动器。所有的文件系统功能只有一小部分在服务器中实现。例如，磁盘服务器不知道一个文件是什么，甚至不知道文件中有什么块。

二、将较大的功能部分分配到服务器以达到这样的目的，例如，远程文件服务器可以在文件描述的信息的基础上满足文件要求。这个客户服务器接口是一个不同于任何在普通本地文件系统中使用的接口的内置接口。这样不同结果的产生是因为一个传统的文件管理器在本质上分为两块模块。一个在客户远程访问模块中实现；另一个在服务器的远程存储器中实现。

三、在客户机和服务器上重复运用文件管理器，以使文件被隐式复制。当文件打开时，一个远程访问操作就从远程存储中获得了一份文件的本地复制本。在这个文件缓存方式中，文件服务器是个完整的文件系统，这个系统在完全文件操作的层次上提供服务，如同复制和删除一个文件。文件缓存系统必须明示的和为客户和管理文件制作的各种复制件的地址保持联系，以使它们的内容能保持一致。信息缓存的想法和一致性问题是一个相等的问题，尽管保证一致性是为了全部文件而不是为了一个信息块。因为在信息存储器特征的巨大差异，在文件缓存中保证一致性技术方面必须和那些使用于缓存块的区别开来。

8.2.2 远程磁盘系统

在 19 世纪 80 年代，磁盘驱动器的成本在工作站的硬件设备中是主要成本。今天，磁盘驱动器的成本骤降，以至于它们在建设工作站时再也构不成什么问题。19 世纪 80 年代的磁盘驱动器还制造热量和噪音，有时候，在办公室环境中，如果将它们直接附在工作站上，这些热量和噪音使得磁盘驱动器若人生厌。两个原理鼓励支持远程磁盘驱动器的操作系统的发展。工作站和网络设计在一起，但没有本地磁盘。在网络中，一个或多个服务器和一个或多个快速大型的磁盘驱动器设计在一起。没有磁盘的工作站使用服务器的磁盘作为它们的第二存储器。当无盘工作站不再是一个节约成本的解决方法（已被和使用X协议的服务器交互反应的无盘 X 终端取代）。它提供一个在分配电脑技术进程中重要的步骤。但是，最近网络电脑的新趋势（有着网络浏览器和虚拟机解释器）标志了向无盘工作站的回归。远程文件服务器解释了在一个客户机服务器适用中的流行层次的许多重要方面。该层次必须得到现代操作系统的支持。但是远程磁盘技术并不象 1990 年那样广泛使用。它对于理解这些系统的基本原理和和设计比学习远程 文件系统更为重要。

图 8-9 解释了在客户机工作站和远程磁盘服务器之间的功能组织。为达到远程访问的目的，客户机器和本地虚拟磁盘驱动器结合在一起取代传统的本地磁盘驱动器，而不是提供一个新的远程文件管理器。VDD 的软件接口在某种程度上和一个本地磁盘驱动器有同样的功能，它可以允许文件系统（其他客户 软件）引用抽象存储设施——远程磁盘应用（RDA）。抽象磁盘可以像一个本地磁盘那样被读出和写入，但是其他等级较低的磁盘管理如格式化、分区、读取 boot 纪录等通常不能在客户机器上操作。远程磁盘应用从 VDD 那里接受命令，又把这些命令运用到服务器磁盘驱动器，然后将结果返回给客户 VDD。请注意，文件管理器必须区分本地和远程磁盘的访问。因此它必须包括为普通本地文件访问的本地磁盘地址和为远程文件访问的虚拟磁盘地址。

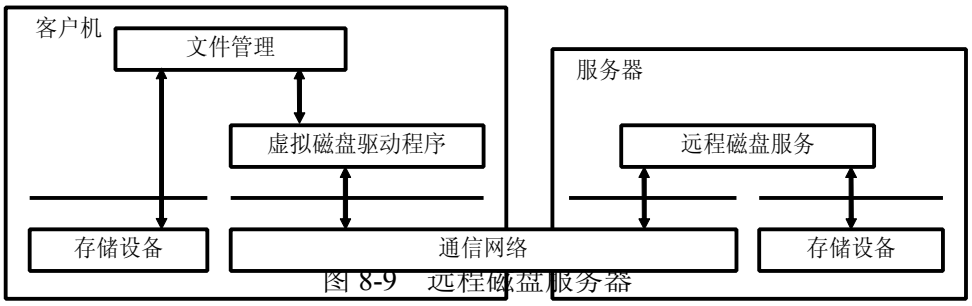


图 8-9　远程磁盘服务器

就像一个物理磁盘有一个设备地址和一系列逻辑块地址一样，远程磁盘有一个传输层地址<net host port>和一系列逻辑块地址。因此，VDD 必须引用使用网络名义上的设施的服务器。当磁盘服务器开始运作时，使用传输层设施将服务器地址和在网络命名服务器上的有名称的注册绑定起来就可实现上述过程。系统管理器将如此构建客户机以使得它可以轻易的使用已决定的名称从一个名称服务器中取回磁盘服务器传输层地址。然后 VDD 可以开始和磁盘服务器直接交流，就像在 FTP 例子中进行的那样。

8.2.2.1 远程磁盘操作

VDD 将每一个磁盘命令都压缩到一个网络包，然后传送它以代替在磁盘服务器机器上的 RDA。在收到的命令的基础上，RDA 以网络包中将磁盘命令解压缩，并且发送请求到它的本地磁盘上。例如，一个读命令包括命令和虚拟磁盘地址。RDA 将虚拟磁盘地址翻译到本地磁盘地址，并且发生一个本地磁盘读的操作，同样的，一个写操作包括命令、虚拟磁盘块和虚拟磁盘地址。当磁盘服务器的本地磁盘操作完成，RDA 将操作的结果——一个写过程完成的通知或在读的情况下，一个磁盘块压缩，并且将其传送 到

客户机上的 VDD。客户机 VDD 将结果解压缩并且返回到本地应用，这就是通常所说的客户文件系统。

既然磁盘是以块为基础的设备，传送单位为磁盘扇区。设想一下，一个扇区与单个网络包完全适合，和一小部分为命令和地址的多余空间。然后使用单个网络包在客户机和服务器之间传送磁盘扇区就成为可能。这个观察结果支持一个数据层次服务器的使用，尤其是在服务器发送更高层次的操作而不是从事一些有联系的协议的情况下。

远程磁盘的结构很简单，但是它面临着潜在的可靠性问题。设想一下 LANS，相对于电脑总线而言可靠性相当低，那么它们在传送磁盘访问请求和回应时足够可靠吗？尤其是使用网络层协议。如果客户正在使用时，磁盘服务器出故障，客户机会不会永久阻塞或者丢失磁盘存储的数据。这些问题将在下面予以讨论。

8.2.2.2 性能的考虑

性能对于远程磁盘服务器是一个重要标准，对于一个在本地磁盘访问时期具有竞争力远程磁盘服务器而言，客户必须能够传送命令给它，并且使它在其磁盘驱动器上执行一个 I/O 操作。服务器必须将结果通过网络返回给客户，然后必须在和本地磁盘访问相同的时间内将结果返回给应用。这个说明很明显依赖于在两种磁盘的访问时间上有明显区别。一种磁盘是和客户机构建在一起的低级磁盘，另一种是和共享服务器构建在一起的高级终端磁盘。这种在低级和高级磁盘驱动器之间的访问时间差别只是 3 或 4 个要素中的一个，但在 1980 年它是 10 个或更多要素中的一个。什么时候一个远程磁盘驱动器足够的块？当我们比较本地软盘和服务器上的高速硬盘的传输速度时，远程磁盘访问可能比本地磁盘访问更快。和硬盘技术相比较，软盘因为循环所需要和耽误时间，其速度是如此之慢，以至于通过比较，网络被轻易打败。

如果客户机有一个较慢的硬盘而服务器有一个较快的硬盘，经验表明，对远程磁盘服务器的访问时间依然是足够证明它们在电脑环境中的使用是合理的，例如，在 19 世纪 7 年代末的远程磁盘服务器中，本地磁盘操作需要 60 毫秒的时间传送一个块，而一个快速硬盘仅需 25 毫秒。在当时的文献中记载的实验看来，和与高速磁盘构建在一起的服务器相连的客户机需要 48 毫秒/块的速度传送，如果仅有一个客户在使用服务器的话[swinebartetal.1979](在这些实验中，客户机和服务器与一个标准以太操作以 3Mbps 的速度联系在一起)。但是，当 2 个客户使用服务器时，访问时间增加到 76 毫秒，当 3 个客户共享磁盘服务器时，需要 100 毫秒。这些实验表明性能的瓶颈在服务器而不是网络。

在另一个测量的研究中，研究者发现远程磁盘服务器仍然是花费成本的[Lazowskaetal.1986]这篇论文引用的实验又一次认为服务器而不是网络是性能的瓶颈。

在以显式的告知在网络层上传输的情况为代价的基础上，TCP 提供了可靠的连接。UDP 比 TCP 快，因为 UDP 并不包括任何确保可靠传输的机制。出于性能的考虑，远程硬盘服务器被设计成可以使用数 据或在某些情况下，使用生的网络层包。这种方式很明确的依赖于客户机和服务器，通过特定的高层次的机制处理可靠性问题的能力。

在远程磁盘环境中，因为路线的缘故，网络传输不允许在因特网上徘徊。因此，远程磁盘系统要求磁盘服务器和客户连接到普通的局域网。这就意味着网络根本不需要使用路线功能。相反，一个客户 -服务器交互反应的高度专业的协议可以在数据连接层直接执行。因此，图表/包/框架性能开始接近在物理网络中可能有的性能入口。

在本地文件系统中，块缓冲被用来重复和 CPU 时间一起的磁盘 I/O 时间。在远程磁盘服务器中，客户可以通过提前阅读缓存块。但是服务器不能这样做，因为它并没有组织文件的概念。客户方的缓存在远程磁盘系统中可以被用来作为性能的增强。

8.2.2.3 可靠性

假设一个和服务器上的物理磁盘扇区一样大小的虚拟磁盘扇区和一个包完全契合。然后，可靠性和两个问题有关。第一、保证磁盘命令最终被服务器执行。第二、使得客户和服务器的操作尽可能的和谐，当两者因为磁盘请求相碰撞时。

网络层确定了最大尺寸的包的内容的可靠传送。这个保证独立于基础数据连接层的框架大小。这就意味着网络包可以被分裂并在网络传送层和接受层之间重新构建。但是全部的可能会丢失。（在网络结构中，包可以另一种命令发送，而是象它们将要通过一系列关口那样被传送。但是，出于性能的考虑，结构还是受到限制，以至于客户和服务器必须在相同的 LAN 上）可靠性的焦点确保了系统正常工作，即使是在包丢失的情况下。

1）可靠的命令执行

本地磁盘命令仅限于小部分操作，块的读写，跟踪搜索，数据命令和另一些生僻的磁盘命令。例如开始和停止驱动器启动，低层次的命令。如驱动器启动控制。不是通过远程磁盘接口完成的。既然对于物理磁盘，只有服务器能够有此层次的控制。同样的，支持另外一些生僻的磁盘命令也不必要。因此，VDD 只有在传送读、写、搜索命令时才需要。

在普通操作中，客户发出一个读或写命令，然后等待服务器的反应，要么和读命令完成的内容在一起，要么和完成读命令的磁盘块返回一起。假使一个客户发出一个读命令，然后要么在发送到服务器之前包含命令的包丢失，要么在服务器完成读操作后的结果丢失。从客户的角度看，这两种情况中命令都没成功。客户该怎么做呢？通常的 VDD 协议时当命令发出时，使一个到计时器开始工作。当时间用完而

结果还未返回，VDD 认为读命令失败，并且重新发出对服务器的命令。

当时间界满时，又有三种情况必须考虑：

一、当第一个命令从未到达服务器，忽视命令是最正确的事情。

假设第一个操作到达了服务器，但结果在网络中丢失，第二个读操作不能影响服务器的正确操作，但可允许用户恢复丢失的包。在这种情况下，读操作被认为是幂等的。这意味着该命令可以重复执行，并产生相同的结果就象仅应用一次那样。（假如我们不考虑高层次问题如另一个客户在第一个读命令和第二个读命令之间写块）

二、假如服务器超负荷以至于当客户时间用完重新发出命令，服务器对第一个读命令作出反应，一段时间后又对第二次操作作出反应。在这种情况下，第二次读操作并不带来任何逻辑危害，尽管可能增加已超负荷工作的服务器的负担。协议的客户端必须准备从命令中放弃这种迟来的结果。

关于这些情况，有两个关键点。第一，幂等操作不能对服务器造成伤害。第二，如果客户机不能处理对重复请求的多重反应，只有客户机受到损害（但是，客户机可以计算它重复了多少次命令，然后有准备的处理有限数量的反应）。

一个磁盘写命令也是幂等的。如果命令包在传达到服务器之前丢失了。VDD 会时间用尽，重新发出命令。如果命令传达到服务器，并且信息被写入到磁盘，但是，内容丢失了，则第二次写命令会导致 RDA 将同样的信息重新写入磁盘扇区。多次写入将导致多重内容被传送给用户。这些内容会比较容易处理。

一个加快读出或写入到下一个更高数量路径的命令不是幂等的。假设头在 track 50，然后执行，头将会移动到 track 51，而第二次执行则会转移到 track 52。

三、假设客户-服务器接口如此设计以使客户发出的所有命令都是幂等命令，并且都有内容，然后，假设网络最终会发出一对适合的命令和内容包，系统将保证命令会被执行。在获得这种可靠性的过程中的关键问题使命令的幂等特征。如果命令不是幂等的，此方式将无法工作。

关于内容，一些启发式证据可以被用来减少他们的明式交流的次数。在写命令的状态下，内容必须是明示的。但在读的状态下，内容可以根据用户得到的读的结果推断。这个方法，像许多其他客户-服务器协议一样，依赖于客户使用倒计时器去重新发送命令，如果没有在合理时间内接受到内容的话。

2）在服务器崩溃后恢复磁盘

假设在磁盘上读或写的过程中服务器崩溃，例如，在 VDD 打开客户机器上的文件后，如果服务器再也不能从崩溃中恢复，客户机器将不能完成它的工作。在现代系统中，磁盘将最终会恢复并试图去回应任何在其恢复之前没能回应的请求。因为客户采用的倒计时策略，客户在服务器崩后会持续发送命令。（一些设计计算连续失败的次数，当次达到一定程度时，放弃这项命令。原则上客户可以在无限延长的时间内持续发送命令，直到服务器从崩溃中恢复过来）服务器怎样知道哪些命令是在崩溃之前受到却没得到满足的呢？当崩溃时，服务器怎么如何恢复正在执行的命令？如果文件描述符号不一致，或磁盘引导块不一致，内容会不会被破坏呢？

如果远程磁盘服务器采用的方式，大部分难题可以忽略不计。这种方式的基础是无状态服务器的理念。一个远程磁盘服务器不需要保持和文件有关的状态就象物理磁盘不需要保持和存储在其中的文件相关的状态。磁盘服务器只是简单的读写块，并不具有包含在块中的连接知识。在客户系统里，文件描述符号完全由文件系统解释。因此，当一个文件被打开时，它的文件描述符号可以通过一个或多个读块操作从磁盘服务器中读出。这个服务器不需要知道客户已取回了文件描述符号。当文件系统跨越了块列表，它将根据文件描述符号的客户软件翻译进行读写块操作。又一次，磁盘服务器不能够即使时横跨文件块列表。

当块被假如或从文件中删除，客户的文件系统从磁盘服务器中读出适合的块，然后使用他们。稍后，客户根据缓存策略，将块回写到磁盘服务器中。结果是，如果当文件打开，服务器崩溃，然后当它恢复时，为它没任何状态以便使其和在恢复之前的未决定的操作一致。服务器不需要知道在它恢复之前发送了哪些命令，既然，各自的客户最终时间用尽然后在恢复后重新发送命令。

关于磁盘服务器必须处理的恢复的主要问题和当它崩溃时正在进行的操作有关。如果磁盘扇区包括将要进行写操作，，弱国服务器开始对磁盘进行实际的写操作，那此操作必须在崩溃之前完成。否则，磁盘扇区会包含一些新的旧的信息。当然，这同样是对本地磁盘操作的要求。如果本地磁盘在写操作过程中失败，那么按规律无论是本地磁盘还是远程磁盘扇区的信息都会丢失，如果服务器没有开始实际的写操作，那么服务器命令可以被客户重复并且不带来任何损害。为了让客户知道操做是否完成，服务器必须知道每一次操作。如果在规定的时间间隔中客户没收到任何通知，客户只需简单的重发命令。

8.2.2.4 远程磁盘的未来

远程磁盘服务器是具有吸引力的，因为设计能够从网络和服务器崩溃中恢复的服务器还是比较容易的。但是，这种服务器因为分割工作的特征会有性能缺陷。考虑到一个操作如在系统中的文件定位。在这个系统中块和使用列联表联系在一起。即使在本地磁盘系统中，这也是一个很耗时的操作，既然它要求文件管理器将每一个磁盘块从文件头（或当前位置）读入到包含目的地的磁盘块中。在远程磁盘系统每一次读块都会导致网络从服务器传送到客户。如果服务器被设计成具有文件描述符号知识，然后实质的网络交通将可以通过发送寻找命令服务器被删除。此命令使得服务器可以目标块定位。这无需通过和客户的交互反应即可实现。但是，这种方式改变磁盘服务器的特征，使其具有文件服务器特征。

磁盘技术从本质上排除远程磁盘作为可实行的商业产品。关于这篇论文，一个具有 10 毫秒访问时间的 20GB 磁盘驱动器可以在零售市场上用几百美购得。无盘工作站的经济和性能方面的诱因再也不存在了。尽管如此，和 15 年前相比，为方便不同客户使用，再共享服务器上保持文件备份得优点已相当明显。这些要求结合在一起，将人们得兴趣从远程磁盘服务器转移到远程文件技术。

8.2.3 远程文件系统

远程文件系统提供和本地文件系统（这个系统可以通过远程磁盘实现）相同的接口给应用程序。在命名计划方面，它们可能和本地系统有所区别（取决于使用技术的数量）。在客户与服务器之间从功能上分配文件系统的方式上，远程文件系统和远程磁盘有所不同。如前所述，通过减少在客户和服务器之间的信息流动的数量和频率，文件服务器可以设计成从根本上减少网络阻塞。成本使客户和服务器必须各自包含文件系统操作的部分状态，然后相应的协调各自的活动。一个文件服务器是一个操作系统如何使用已分配的软件提供网络服务的具体例证。

8.2.3.1 通常的结构

一个本地文件在磁盘设施上被当作相关磁盘块的集合一样实行。这之间的关系可以当作列联表中 指明第一个块，通过定位已索引的集合的块，通过间接定位块，如在 UNIX INODES 中，文件描述符号给磁盘块提供路径图。文件服务器设计目标是通过利用文件执行命令的过程中的文件结构知识增强磁盘服务器的性能。

如图 8-10 所示，远程文件系统划分为客户部分和服务器部分。客户部分实现和服务器部份交互反应必须的操作。不太明显的一点是，本地文件管理器必须有所变化去适应它现有的对本地文件的访问并能够使用远程文件服务。客户机器的应用软件使用普遍的模块和本地文件系统以及远程文件系统交互反应。在操作是应用于本地还是远程文件的基础上，普通的部分将文件系统请求传送到本地或远程文件系统的客户部分。

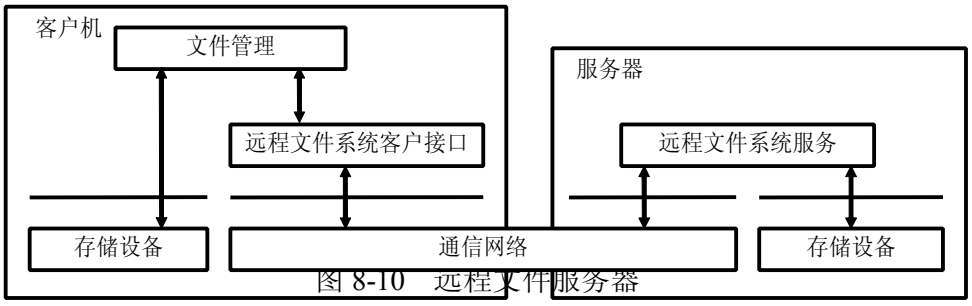


图 8-10　远程文件服务器

为了便于讨论，或者因为它反映了大多数系统中的可靠性，假设文件系低层次的比特流文件。这就意味着文件系统实现的操作和打开、关闭、、读、写、搜索以及 IOCTL 的可能形式对应。这儿有一个本地文系统的 I/O 的简要概括。打开命令使得文件描述符号可以下载到磁盘服务器的主要内存，然后使得文件比特流处于随时读出或写入的状态。一个读或写操作将现在的读/写文件位置应用到比特流中，并且可能造成块被读或写，这依赖于过去的读或写以及文件系统缓冲的内容。部分操作是为了文件系统压缩或解压缩必须的块。关闭的操作使得完全输出的缓冲（那些是满的并且等待写入设施的）和文件描述符号回写到磁盘。

逻辑文件系统在功能上可以分裂为几个不同部分。例如，服务器必须使用文件描述符号实现块管理和缓冲。客户将要处理读或写头管理，压缩或解压缩，以及额外的缓冲。这种方式是有吸引力的，因为它使得大部分块列表管理的系都压缩载服务器中，同时，块到流的翻译的细节被压缩到客户机器。客户必须保持文件描述符号现有的备份，以使得客户可以执行确认，保持而后读写比特流享关的现有状态部分和管理锁定。相反的，服务器也需要知道文件位置从哪里引用比特流以至于它可以读或写要求的块。服务器还必须更进一步的知道文件块的位置在哪里。一个可选择的方式是试图保持客户的文件块管理，而使得服务器实现块缓冲。这种方式同样要求客户和服务器必须同时保持打开文件的状态。通常说来，客户和服务器必须共享文件描述符号的信息，既然文件系统的不同部分使用了文件描述符合的不同信息。当功能得到分配，客户和服务器必须重复文件描述符号。

在所有的远程文件服务器方式中，一个打开命令被输送到服务器。服务器然后取回文件描述符号，保持一份备份，并且传送另一备份到客户的文件管理系统。这样文件描述符号多余的备份存在于远程磁盘，远程服务器和客户机器上。

关于设计远程文件系统有两个独立于特殊分割功能的问题必须决定。

- 既然客户和服务器都缓冲磁盘块，什么是最有效的结合系统的缓冲策略？在一个或另一个地方缓冲有无优点还是两边都必须使用缓冲？
- 幂等操作和无状态服务器能够被用来实现一个简单的崩溃恢复策略，但提高了网络中的信息流量。由于文件描述符必须保存在客户机和服务器中，这一策略如何才能被应用到文件服务器中？如果使用的话，对性能会有怎样的影响？

8.2.3.2 块缓存

在逻辑上，当一个远程文件的读操作被发送到客户端，它被打包并被发给服务器。服务器对消息进行解码，从磁盘上读出块，然后将块返回给客户机。本地文件系统中的讨论指出，文件的连续特性强烈鼓励缓冲以使设备与 CPU 的操作能够重叠。这能使性能得到很大提高（使得进程的执行时间减少）。在一个远程文件服务器中，缓冲能够用来通过在客户机和服务器中都实行缓冲，以使网络传输的操作与在服务器上的磁盘访问能够重叠。在读的这边，服务器在磁盘上提前读，并缓冲客户机要访问的块。客户机执行时，它请求读服务器的缓冲区，并按请求它们的客户机应用程序期望的顺序访问。缓冲写的信息按照反的方向进行。

严格的提前读、延迟写缓冲策略可以通过在技术中加入一个技巧层来使得它更为有效。对一个文件的数据访问趋向于遵循局部化模式，正如以前在对虚拟内存的研究中发现的。那就是，当一个块中的某个物体被访问后，连续文件语义表明接下来的字节也将被访问。在很多应用程序中，数据访问有局部性，虽然一个块中的数据可能不遵循纯粹的连续模式而被重复的访问。

局部性的存在表明，当一个块被拷贝到内存的缓冲区时，它可能被重复访问，就像在虚拟内存系统中一个页面被重复访问一样。这样很容易在缓冲方案中加入一个替代策略，以使一个块一旦被客户机读取或者写入后不会被移走，除非替换策略如 LRU 指明它必须被移走。这样一个策略替换了原先的提前读、延迟写的缓冲语义。一旦缓冲技术开始考虑一个替换策略，它也能够通过把这一策略应用到文件的大的或者小的块中来改变拷贝到客户机的信息数量。大的块趋向于赢得局限性，因为它们在客户机的缓冲区中保持了大量的字节。然而，当一个新块必须被载入时，载入的时间将会很多。通过称这一技术为“block caching”以使它与单纯的缓冲技术区分开来。

远程文件系统允许多个客户在同一时间访问同一个文件。在本地文件系统中，如果两个或以上的进程同时打开一个文件进行写操作，那么在合乎逻辑的写存在后，每一个单独的写操作都会写回磁盘。在缓存机制中，一个块在被写回服务器磁盘前可能会在客户机上保存一段时间。假设两个进程已经打开一个文件进行写操作，并且都把同样的快缓存到客户机上。现在，当一个进程在客户机上写入快时，写的结果将不会将不会为另一个客户机所感知。这就是所说的块存一致性问题。这与多进程共享内存中的缓存一致性是类似的。

操作系统怎样才能解决块缓存一致性问题？一些文件服务器只支持“连续写共享”。在这种情况下，当一个进程对某个文件有写的权利时，其他多个进程不可将该文件打开。如果一个文件被作为本地磁盘写入并关闭，缓冲区将被清洗，接下来任何的打开操作将在由 write 操作生成的数据上进行。接下来的写共享保证了对先前打开的文件的缓慢缓存不会与对文件的新的打开命令产生冲突。解决从其他客户机回写块这个更为困难的问题可以这样，如果在回写完成之前有新的打开操作到达，则强迫缓慢的写者更新服务器的磁盘映像。

“并行写共享”是一个更为灵活的策略，多个客户机可以打开同样的文件进行读或写。在这种情况下，新写入的数据必须以一种及时的方式传播给客户机访问者。并行写共享是通过在有任何一个客户机打开一个共享文件进行写操作时禁止缓存来实现的。

The Sprite Network File System——一个由加利福尼亚大学伯克利分校开发的 UNIX 兼容文件服务器——采取更为进取的策略进行缓存以获得更好的性能。他被设计了用来利用关于 UNIX 文件的两个经验观察结果。首先，高性能本地文件系统广泛利用在用户级进程和磁盘之间的缓冲区。第二，用作缓冲的物理内存数量对性能有很大影响。

Sprite 被仔细设计了来通过缓存技术增强性能：

- 既然它利用了客户端缓存技术，它必须花额外的代价来保证文件的缓存拷贝之间的一致性。
- 它为每个缓存使用动态空间定位，其中缓存定位策略与虚拟内存机制是类似的。

Sprite 使用延时回写策略，当一个客户机写入它的缓存时。客户机不是立刻向服务器刷新缓存信息并经过服务器缓存到磁盘上，write 在服务器有空闲时间或一个适当的时间间隔过去以后执行。这允许客户机不必等待服务器的写操作完成就可完成写操作。它有时当数据刚写就被删掉的情况下可以节约写操作。例如，一个编译器可能会在传递 1 和 2 之间创建一个中间文件，然后当编译完成后删除文件。文本编辑者同样试图保持临时文件一小段时间。在延迟回写方法中，缓存的信息可能还没有被写回磁盘。Sprite 在回写操作中使用一个 30 秒的延迟，然后在另外 30 秒中真正完成这个操作。

Sprite 的开发者报道说，他们进行的文件系统的实验实现了很高的性能表现。使用缓存的客户机的速度比不使用缓存的客户机的速度提高了 10%到 40%。无盘客户机与有一个类似基准集的磁盘的工作站相比，慢了不到 12%。根据实验中观察到的应用，开发者推测对于一个运行平均程序的标准配置客户机来说，一个服务器应该能够处理 50 个以上的客户机。

8.2.3.3 崩溃恢复

在本地文件系统中，缓冲带来另一隐患。如果当信息被缓冲时，磁盘或机器崩溃，尤其是文件描述符，信息将会丢失。在远程文件系统中这种危险更大，因为不仅磁盘或机器会崩溃，而且信息可能由于网络的不可靠而丢失。如果一个服务器崩溃后恢复过来，它必须能够确定在崩溃时受到影响的每个对话的全部状态。崩溃恢复是设计中需要重点考虑的。

一些文件服务器在设计时更多的考虑操作的可靠性，而另一些更多的考虑性能这就需要结合复杂的

崩溃恢复算法。两种设计都试图实现高可靠性和高性能，但侧重点各有不同。

1）面向恢复的文件服务器

设计一个无状态的服务器（在崩溃后简化恢复）的策略可以从使用在用作远程文件服务器的磁盘服务器的技术中拓展得到。在一个无状态的文件服务器中，文件描述符总是被保存在客户机上，当客户机请求一个操作时，总是将与操作有关的描述符部分的拷贝传送给服务器。在执行 open 操作时，服务器取回文件描述符，然后在传递给客户机之前做一份拷贝。在 open 操作被执行时，服务器的文件描述符中的信息是正确的。假设服务器只使用文件描述符中的内容作为所有后续操作的“暗示”。在客户机的文件描述符的拷贝保持着文件的真实状态。这表明服务器可以使用文件描述符来执行任何操作以增强性能，如缓冲。然而，服务器不允许执行任何准确性依赖于它在文件描述符中准确性的操作，如果服务器在执行操作如块管理之前需要知道文件描述符的状态，它必须通过请求获得文件描述符对应部分当时的拷贝。客户机一直会通过一个请求引起服务器执行这样的操作。每个这样的请求都包含文件描述符的正确的值。服务器从不允许在没有客户机的指示或首先获得客户机执行操作的允许之前执行正确性依赖于文件描述符状态的操作。

接下来，由客户机调用的操作必须都是幂等的。对于低级的文件系统来说，这不难保证，因为这些操作是关闭，打开，读取，写入和查找。读和写的语义与磁盘读和写的语义是一样的，并能被做成幂等的。打开是幂等的，因为它不需要改变文件描述符就可以被重复。类似的，关闭是幂等的，因为它引起缓冲区刷新到磁盘上，文件描述符被写回到磁盘。

这种文件服务器设计允许系统使用更快的网络协议而不是那些需要可靠连接的。例如，这种类型的远程文件服务器可以使用 UDP 或者一个网络的网络层中的未加工的包接口。

The Sun Network File System (NFS)是最著名的面向崩溃恢复的文件系统。NFS 自从二十世纪八十年代面市以来受到广泛的欢迎[Sandberg et al., 1985]。Sun 的目的是为了支持异构的文件系统，即使在客户机中，使得通过一个简单的崩溃恢复机制实现合理的性能和工业强度的可靠性。NFS 变得如此普遍，以至它的基本特性成为网络协议，也称为 NFS。它面市 15 年后，它已有 3.0 版本，并仍在商用远程文件服务中处于统治地位。NFS 与其他的文件服务有着一样的总体组织（图 8-10），修改后如图 8-11 所示。基本的 UNIX 文件系统接口被一个虚拟文件系统，VFS，客户机与服务器核心中的接口所代替。VFS 实现标准的本地 UNIX 文件接口，它能被传送给一个常规的 UNIX 本地文件管理器。它也可以用来在客户机上把 UNIX 文件操作转换成用作其他文件管理器上的文件操作，这样 VFS 能够，例如，从一个不同的操作系统使用文件管理器(这被包含在允许 NFS 在 pc 上实现，尤其使得 VFS 能够产生 DOS 文件管理命令)。要被应用在远程文件上的操作被传送给 NFS 的客户端部分。

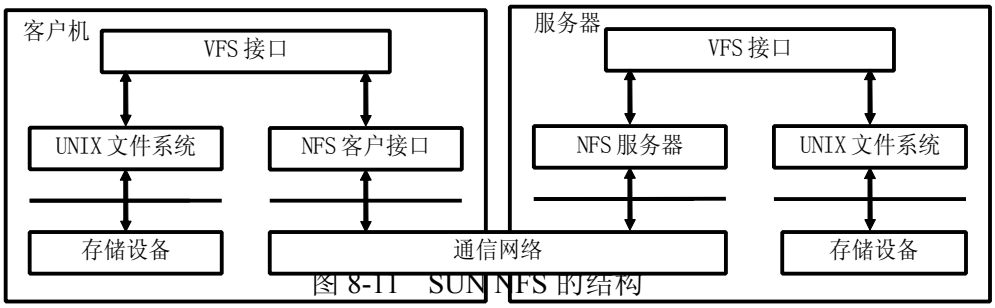


图 8-11　SUN NFS 的结构

一个 NFS 的文件体系可以是异构的，因此体系的不同部分是通过不同的操作系统文件管理器来实现的。系统被设计来为体系中的每个子树使用适当的文件管理器。为了实现这个，任何本地系统的文件接口，如 UNIX 的文件接口能够在 VFS 的顶部被实现。VFS 模块能够被用来使文件管理器接口与文件管理器实现相匹配。但是更重要的是，它也能够传递一个标准命令集合给一个远程服务器。这些命令使用一个称为 vnode 的抽象文件描述符，并在客户机与服务器之间建立一个点对点的协议，称为 NFS 协议。Vnode 捕捉文件管理器和在文件体系中的子树类型的一致性。

一个 NFS 服务器管理在网络文件体系中的子树。如果客户机需要使用子树中的文件，它把子树装入自己的体系中。然后 NFS 的客户机和服务器部分使用协议来协调客户机的 VFS 接口的和 NFS 服务器的操作。在极端的情况下，一个无盘客户机可以通过把 NFS 当作远程磁盘服务器使用来把服务器的 root 文件系统装入。NFS 的部分协议被设计用来实现和协调文件系统的装入。

NFS 协议也处理 NFS 客户机和服务器之间的数据传输。它是协议的一部分，这协议指导服务器为个别的文件和目录根据来自客户机应用程序和 NFS 的客户端的命令操纵信息节点。NFS 协议意图允许服务器变得无状态，这样大大简化崩溃恢复。每个服务器命令是自动的，要么运行完成，要么对服务器数据不产生影响。这通过在客户机保持完整文件描述符和在服务器保留描述符的一个备份来实现， NFS 协议使用一个 LOOKUP 命令而不是一个打开命令来做这个。每当客户机发出要求状态必须是正确的命令时，它把描述符中对应的部分拷贝到命令消息中，并把它和命令一起传递给服务器。从而，NFS 使用数据报而不是请求一个连接协议，客户机或服务器都崩溃都不会影响其他的操作。

NFS 协议应用在 NFS 的客户机和服务器模块之间，都是在内核实现的。NFS 的第一个版本使用 UDP,

而版本 2 和 3 使用 UDP 或 IP 上的 TCP（NFS 在英特网上工作时）。关于协议的其他细节可以可以在 Stevens [1994, ch. 29]中找到。SUN 的微系统文档描述了实现。

2）面向性能的文件服务器

面向恢复的方法由于需要在很多操作中传递文件描述符的一个拷贝而受到批评。另一种可选的方法是允许客户机和服务器分布文件描述符，然后使用其他的方法分布的文件状态始终是一致的并能在客户机或服务器崩溃时被重建。客户机和服务器的任务将得到相当的简化，如果连接两者的网络是可靠的。所以这种方法通常使用一个传输层协议如 TCP。

客户机按照比特流的操作执行文件输入输出命令，而服务器处理比特流与磁盘块之间的转化。这样，服务器保持比特流的文件状态。例如，客户机发出一个写操作，要将一块比特流写入文件。命令并不要求正好适于一个包的大小，因为字节块的大小是可以任意的。因为客户机含有部分文件描述符而服务器处于空闲，当客户机发出写操作，它将以下的任一操作：（1）它将假设服务器受到了命令和数据并完成了写操作；或（2）它不会把块的任何部分放入比特流，命令随后失败。

客户机和服务器都含有部分文件描述符，包含文件和记录锁，如果它们被使用的话。这样，如果当客户机打开一个文件时服务器崩溃了，服务器需要恢复每一个打开文件的描述符状态。否则，例如，客户机可能认为一个文件被锁了，而服务器恢复描述符指示文件被解锁了。客户机必须探测服务器是什么时候崩溃的。虚拟流程机制如 TCP 将通知客户机如果服务器崩溃的话。当客户机发现服务器崩溃后，它将当前分布在它和服务器间的描述符的状态保存起来。当服务器指示它已经恢复时，被延缓打开的文件被恢复，远程文件操作继续执行。应用程序可以由客户机文件管理器通知，并能被允许做其他的事情，虽然在默认情况下，它们仅仅被阻塞，等待服务器恢复。

服务器被假设为易变的，所以它在设计时特别注意对恢复的支持。首先，每一个打开文件的描述符被存储在稳定的存储器中，以使它在文件打开时，如果服务器崩溃，能够被取回。当客户机请求服务器执行一个操作时，文件和描述符的状态在服务器开始执行操作之前被存储。如果服务器在执行中崩溃，文件和文件描述符的初始状态在恢复中被使用。当服务器完成一个操作后，它更新在稳定存储器中的文件描述符，并把变化存储到文件中。

稳定的存储器比较难以实现，使得它从不出错[Larnpson and Sturgis,1979]。大部分稳定存储器的实现是“几乎一直是正确的”，但它们有时会出错。大致的思想是建立一个关键的硬件级的部分——运行至完成即使机器断电的物理活动块——来保证信息被写入存储设备。接下来，稳定存储器内容的两份拷贝被保存下来，大概在不同的设备上，为了防止一个设备出错毁坏稳定存储器。当一个稳定存储器的写操作存在时，关键部分把写的内容压缩到稳定存储器的所有拷贝中去。如果机器在关键部分出错，一个恢复进程必须能够发现机器在关键部分出错了。它然后比较两份拷贝，如果第一份正在被写，那就把它抛弃并使用第二份。这与机器在写操作存在前出错的情况是一致的。如果第二份正在被写，第一份被作为稳定存储器的内容来使用。如果崩溃是在写的中间出现的，第一份拷贝将被使用。

很少有商用操作系统实现稳定存储器。它们依赖于后备电源系统以使机器能够在出现断电后运行几毫秒。在电源断电时产生一个中断，系统然后使用后备电池执行断电中断。待决的代码完成任何的写操作以使机器不会在写操作上出错。这种方法对于磁盘设备出错同样是敏感的，然而由于磁盘头的崩溃将毁坏稳定存储器的单个拷贝，从而使得恢复成为不可能。

8.2.4 文件级缓存

第 2 小节讲了远程磁盘如何才能支持共享。这一主题在第 3 小节对远程文件服务器的讨论中得到了扩展。文件缓存是在文件服务器中的缓存的一个逻辑上的极端。它是文件共享的第三级。它允许整个的文件被自动的从服务器拷贝到客户机，然后当客户机关闭文件时返回给服务器。这样客户机通常含有一个小的磁盘。文件级缓存通过消除个别磁盘块或文件块在网络上的更新来提高整体的性能。所有的变化在再次写回服务器前被做成本地的备份。文件级缓存并不防止一致性问题；然而，问题现在存在于文件间的基础上。这一领域的文献把文件描述为可变的或不可变的。一个不可变的文件在它被拷贝到客户机之后不可以被改变。可变的文件可以被改变，并且服务器被要求管理这些变化。

处理文件中变化的一个基本的方法是要求所有的文件成为不可变的。当客户机取回文件的一个拷贝，它被认为是将不会改变的。这对很多文件来说很适；但它不允许文件被更新。文件的改变被通过对文件不同版本的支持来解决。如果一个客户机取回一个文件，然后需要修改它，客户机创建一个包含有必要变化的新的文件版本。这个更新过的文件在被关闭时能够被写回到服务器。如果在第一个客户机取回文件拷贝期间没有其他的客户机取回文件的拷贝，那么这个新的版本将成为文件版本的默认值。

在另一方面，假设两个客户机取得文件的同一个版本，并把它更新、写回服务器。服务器将给每一个指定一个唯一的版本号。“不一致”的文件被允许作为同一个文件的不同版本而共存在服务器上。服务器为不同的操作使用不同的版本。注意“操作” 在这种文件服务中有着与在通常文件系统中的不同的含义。例如，一个文件的移去操作移走一个文件的最老的版本，而打开操作使用文件的最新版本。

一致性机制要求客户机与服务器各自保存文件系统的状态。这就强迫服务器每当文件在服务器单元被写时使用回叫（callback），并结合一个崩溃恢复机制。

8.2.5 目录系统及其实现

目录系统一般使用分层结构，因为这一方法是与人们在手工文件系统使用的结构技术一致的。远程文件系统拓展任一单个主机的分层文件结构以适合一个主机的网络。网络自身同样使用分层组织结构。例如，一个网络层的 internet 地址是一个有着无名的根的三层子树——可能是特定网络的名称。根节点的子节点是 internet 中的网络，网络的子节点是依附在网络中的主机。远程文件系统为了标识位于网络中 不同机器的文件而拓展了每一个主机的文件名层次结构。

8.2.5.1 文件名

文件可能通过两种主要的途径在远程服务器上被访问：超路径名和远程安装。超路径名是首先应用在网络文件服务器中的，大部分的系统已经发展成为远程安装技术，因为它提供命名和定位的透明性。

1）超路径名

超路径名拓展了普通的分层路径名称，包含了“在根节点上的层次”。Super 级的名称是从名称的平的空间中取出的机器名称。例如，一组远程文件服务器通过下面类型的超路径名来标识文件：

goober:/usr/gjn/book/chap16

机器名是 goober，在 goober 上的文件的绝对路径名是：

/usr/gjn/book/chap16

另一种选择名称方案建立在这样的算法上：“在机器的根开始，向上一级，然后就是从那里开始的路径。”在这种命名方案中的文件有以下的形式：

/. ./goober/usr/gjn/book/chap16

超路径命名使得应用软件区分本地文件和远程文件，因为远程文件名有超路径符号。一个支持这种命名形式的远程文件系统将使用机器名，并根据机器使用的语法从绝对路径名中分析它，来标识主机。它接着将使用机器名如 goober 来查找正在使用命名服务的机器的网络位置。文件服务器在客户机上实 现的部分将配合在远程机器上的服务进程来访问目标文件。

2）远程安装

远程安装方法有着更为广泛的应用，因为它支持命名和定位的透明性。它从用于可移动媒体磁盘驱动器的 UNIX 安装操作发展而来。本地的安装操作允许管理员连接一个文件系统，如在可移动磁盘上的，到本机的文件系统上。被装入的文件系统取代本地文件系统的一个节点。接着路径名可以在两个文件系统上被拓展，只要子树文件系统保持被装入。远程安装命令通过允许位于远程主机的子树可以被装入到本地文件系统来拓展安装命令。因此，在本地文件系统中的路径名可以通过网络拓展到另一个文件系统，要求后者当时是被远程装入的。在图 13 中所示的远程安装命令把 S 机器中的 s_root 目录装入到 R 机器中的安装点 mt_pt。这样，机器 R 中的/usr/gjn/mt_pt/zip 与访问 S 机器时的/m_sys/s_root/zip 指向同一个文件。这种方法使得每个机器上的进程看到网络文件系统的不同构形，虽然本地和远程文件名有着同样的格式。这种不同透视图的一个结果就是绝对路径名不可以在不知道远程安装构形的情况下在处于不同机器的进程间传递。例如，在机器 R 上的进程 Pi 在消息中传递一个网络范围的绝对路径名给位于机器 S 的进程 pj，pj 将无法在不改变绝对路径名的情况下访问文件。

对于一个将被远程装入或者从客户机访问的文件系统，名称可能需要在全局名称空间声明一下，如通过在名称服务器注册。远程文件系统通常在包含名称服务器的网络配置下运行。如，一个域（或一个大型网络的一部分）包含一个中心名称服务器来注册在域中声明了的文件系统。文件服务器可能不支持跨域的远程文件操作。客户机在名称服务器中查找一个文件系统名，然后在客户机文件名空间远程的装入文件系统。在客户机和服务器之间完成第一次远程安装时，一个虚拟回路在两台机器间被建立。在多路传输连接时，后来的远程安装使用已存在的虚拟回路。

8.2.5.2 打开文件

参照在一个树形目录结构中打开一个文件的步骤，树中的路径名指定一个用来从中搜索目标文件的文件描述符的目录序列。一次路径搜索可能导致文件系统读取非常多的磁盘块，因为每一级目录通常包含至少两个设备读操作。当这些操作通过网络在一个远程磁盘上执行，产生的加载和网络传输的延时将会非常大。

正如在磁盘服务器讨论中的结论，这是远程文件系统的基本原理。而远程磁盘简单而且相对有效，在有些例子中，客户机需要在每次读之间做一些计算相对少的读操作，如在一个块中查找指针。如果这些操作能够在服务器中实现，将会产生一个性能上的总体提高，由于减少了网络中的延时。远程文件系统与远程磁盘系统的区别在于，文件和目录系统的部分语义不仅在客户机实现，还在服务器中实现。服务器提供可以从每个客户机访问的共享文件。服务器可以提供并发控制和文件保护来作为文件服务的一部分。

当进程打开一个位于远程文件服务器上的文件，可能导致一系列的相对复杂并耗时的步骤。总体上说，打开操作引起对路径名中的每个目录的一系列的搜索。在搜索的每一级，都有可能遇到远程安装点。当遇到一个远程安装点时，后来的目录搜索应该交由远程机器中的文件服务器来完成打开命令。例如，一个网络文件系统被组织成这样，S 机器中的 s_root 目录被远程安装在机器 R 的安装点 mt_pt。机器 T 的目录 bin 被远程安装在机器 S 的安装点 zip。如果一个位于机器 R 的进程使用路径名**/usr/**

gjn/mt_pt/zip/xpres 来打开文件，打开请求首先在机器 R 执行，直到搜索遇到远程安装点 mt_pt。机器 R 然后把路径搜索传递给机器 S，它继续在目录 s_root 中搜索直到遇到远程安装点 zip。机器 S 然后把路径遍历传递给机器 T 中的文件系统，从目录 bin 开始，打开文件 xpres。远程文件系统中的绝对路径名可能会产生大量的不同文件系统上的分布式进程。

在 UNIX 文件系统中，一个成功的文件打开操作使文件描述符被载入到客户机的内存中。大部分系统也是这样要求的，因为文件的当前状态被保存在描述符中。如果两个不同的客户机进程打开同一个文件，每个进程将含有打开文件的描述符的缓冲版本。依赖于操作系统的策略，两个系统可能都允许同时打开文件进行写操作。这在 UNIX 的许多版本中都是可实现的。这种情况表明了通过存储锁来控制对当前可能被多重访问的文件的访问的必要性。一些远程文件系统提供了上锁的机制，而其他的如 sun NFS 的早期版本则不支持。

8.3 分布式计算

远程文件系统是第一个充分利用高速网络的系统。文件作为一种有组织的信息载体，最初是为了便于批量计算而设计的。文件管理接口和存储模型的使用包含了分布式程序的形式。现在的操作系统大多使用其它的方式以便支持网络环境下的计算。

本节介绍操作系统中支持分布式计算的重要技术。一般说来，分布式计算需要由新的、专门的网络协议支持，那么我们就不得不涉及高层协议。同时，由于需要文件和设备支持分布式计算，本地文件管理也不得不随之改变。

本节首先简单介绍分布式系统。然后讨论分布式环境中进程管理的特性，然后是操作系统如何支持网络环境下的信息传送。以往，人们把消息作为分布式计算的基础——以过程的调用与返回的形式，这导致远程过程调用范型的发展。本节将介绍现代分布式计算中广泛运用的远程过程调用的工作原理。假定一台机器上的进程可以使用另一台机器上的存储空间（当然，两台机器在同一网络中）。这一章还要讨论网络环境中使用的分布式存储管理的技术。

8.3.1 分布式系统概述

8.3.1.1 分布式系统的设计目标

分布式系统的主要设计目标有四个：通信、资源共享、分布计算和可靠性。

分布式系统的通信既包括底层的消息传递机制又包括高层的通信方法，如远程过程调用。同网络系统相比，通信应该更加具有效率和透明性。在分布式系统中，本机上的进程间通信和网络通信的方式应该是一致的，只有这样才能有效实现分布计算。另外，网络通信的效率也是非常重要的，如果一个系统的网络通信延迟太大，实现分布式系统从效率和实用性而言就无从谈起了。

资源共享是指用户在系统不同的位置都可以有效透明地使用分布式系统的资源，无论这以资源是属于本机的还是属于分布式系统中的其他计算机，如一个用户可以在位置 A 可以使用未置 B 的打印机，另一个用户也可以在位置 B 访问存储在位置 C 的文件。使用的透明性是分布式系统与网络操作系统最本质的区别，用户对本地资源的使用和对远程资源的使用应该是一致的。从理论上来说，分布式系统应该管理系统中所有计算机的所有资源，用户使用分布式系统就像使用一台拥有多个处理器、多个独立存储器和多个外围设备的计算机系统一样。在目前的实现层次，商用的分布式系统一般提供共享远程的文件、打印机和其他硬件设备、分布式数据库中的数据、以及其他一些资源（如处理器和内存）。

分布式计算是分布式系统的关键特征之一。一个计算任务可以被分解成多个子计算任务并被合理地调度到分布式系统中的多个计算机上去并行执行。分布并行是一个并行计算的一个重要研究方向，它可以有效的达到加速计算的目的。另外，当一个计算机上的计算任务过多的时候，分布式系统应该允许把其中的一些任务自动迁移到分布式系统中的其他计算机上去运行，这一特征称为负载共享。

可靠性和鲁棒性式分布式系统另一个主要特征。一旦分布式系统的一个节点崩溃时，它不会影响到分布式系统中的其他节点，也不会影响到整个分布式系统的运行。这就要求分布式系统中的所有计算机是无主从的，分布式操作系统和分布式数据库系统的软件功能也是分布的，这样一个节点出现了故障，其他节点和这些节点上的系统软件就可以去弥补这一故障，从而使整个系统呈现出坚定性和健壮性。

8.3.1.2 分布式操作系统的实现考虑

1）数据迁移

在分布式系统中，一旦一个用户需要在机器 A 访问机器 B 上的文件，可以通过两种方式实现。一种方法是先把文件从机器 B 拷贝到机器 A，再在机器 A 上使用该文件，如果该文件被修改，使用完毕后要把该文件从机器 A 传回机器 B，Andrew 文件系统采用的自动 FTP 系统就是这个方法的一个典型例子。显然，这种方法对于大文件来说存在着很大问题，一旦数据被修改，整个文件都需要被反复传递，这将影响到系统效率。

另外一种实现方法是仅仅把需要访问的数据从机器 B 传到机器 A，Sun 的 NFS 和 Microsoft 的 NETBUI 都采用了这种方法。显然后一种方法的效率比较高。

另外，对设备的共享以及对数据库中数据的共享也是一种数据迁移。

2）计算迁移

在一些情况下，在机器之间迁移计算式非常重要的。如，一个计算需要处理保存在另一台机器上的大批量文件数据，从效率角度出发，与其把异地数据迁移到本地来执行，不如在远程执行这个计算。

计算迁移可以有多种不同的执行方式。如果进程 P 需要访问在远程机器 A 上的文件，它可以通过 RPC 调用机器 A 上的例行程序来处理文件，并把处理后的结果传给自己；它也可以发送一个消息给机器 A，机器 A 上的操作系统将创建一个进程 Q 来处理这个文件，进程 Q 处理完毕后再把结果传递回进程 A。注意，在第二种方式中进程 P 和 Q 是在不同的机器上并发执行的。

3）进程迁移

对计算迁移的扩展是进程迁移。一个进程开始执行之后，可以被迁移到其他机器上去运行。分布式系统在迁移进程时，既可以把整个进程也可以把进程的一部分迁移到其他机器上执行。

实现进程迁移由两种基本技术。采用第一种技术时，用户不必要关心迁移的发生，也不需要在编程时考虑迁移，操作系统将根据效率、负载平衡和分布并行计算的需要自动在多个节点之间迁移进程，这种方法非常适用于同构系统，对于异构系统来说则必须提供中间执行代码。第二种技术要求用显式定义进程如何进行迁移，这一方法能够更好的适应不同的硬件与软件环境。

4）分布式共享内存

内存中数据的迁移需要分布式共享内存技术，这以技术的实现方式类式于虚拟存储器技术，主要的难度在于数据通信的速度。

5）鲁棒性的实现

一个分布式系统应该能够更好的适应各种类型的软硬件故障，为了做到这一点，一旦一个故障发生，系统应该能够及时检测，然后重构系统以重建计算环境，并在故障解决后进行恢复。

故障检测的主要方法是通过一个 handshaking 过程，每个一个时间片就去检测一下该节点或该软件模块是否响应，如果没有得到回答信息，则认为故障发生。一旦故障发生或关机时，可以通过广播通知系统中的所有节点，及时调整和修改路由表，标记该节点为故障节点，以后系统将不再请求故障节点服务。同样一旦故障恢复或机器报到，同样可以通过广播通知各个节点并调整和修改路由表。

8.3.2 分布式进程管理

现代软件在分布式和并行计算方面有了飞速的发展。这种分布式计算需要两方面的支持：共享可分配存储空间的多处理器模式；提供抽象多处理器的计算机网络。这种“多计算机”模式的目标是：为设计和执行分布式计算提供简单而有效的环境。这样，应用计算中的并行度可以获得较稳定的投入/产出比。尽管操作系统的开发者们仍然有很多问题要解决——我们接下来将讨论一些这样的问题，不可否认，操作系统技术已经有了巨大的进步。

8.3.2.1 任务分割

计算机网络中的某一台计算机总是可以使用传统的顺序进程方式来完成一次应用处理。对操作系统的挑战是允许多个自由进程同时在单处理器机器上执行。网络提供了一种机会，可以将计算任务分割为多个逻辑单元，由网络中不同的计算机同时执行。理想状态下，一个单处理器状态下执行时间为 T1 秒的计算任务，在拥有五台或五台以上的网络环境中的执行时间为 T5 秒,T5=T1/5。我们称这个计算任务在网络环境中执行的加速度为 5。由于在管理、同步和通信上需要消耗时间，在 N 台计算机组成的网络中，不可能达到 N 的加速度，虽然这是分布式计算的目标。在应用程序员看来，这个问题的实际意义是将一个串行的计算分割为多个单元，每个单元可以同时执行从而减少执行时间。如果这种分割是“完美”的，而且在管理、同步和通信上也没有时间消耗，那么加速度可以达到 N。

除了共享文件的能力，完全由操作系统实现的远程文件系统还是一种尝试。它将文件管理的工作分割，由两台机器联合完成。远程文件系统中，这种分割实际上增加了进程的执行时间。因此，远程文件的优势仅在于文件共享。很自然会有这样的疑问：将程序分割后并行执行是否总是可以减少执行时间？答案显然是对。当然，分割必须很小心。如果分割的程度很大（远程文件系统只分割成两部分），加速度就较为明显。

一个程序员应如何分割串行的应用程序才能使其有明显的加速度？应用程序员需要了解计算中使用的算法的执行方式。首先，地址空间被分为两部分：一部分包括执行应用函数需要的软件；另一部分是数据。这种分割方案对应图 8-12 中的第一次变换。

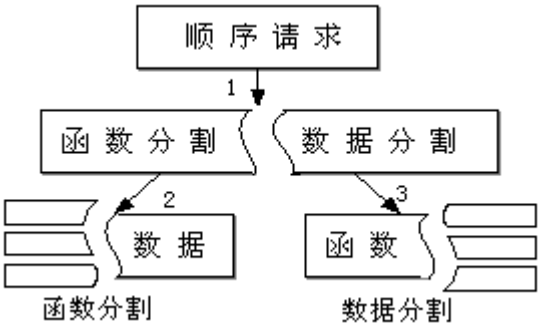


图 8-12　分解一个顺序计算

在函数-数据基础上进行分割的有两条思路。图 1 中的转换 2 表示将数据放在一块地址空间而把函数部分分割后作为独立的进程在各个处理器上执行。这种分割方式称为函数分割。在函数分割中，需要将函数分割为相对独立的片断，然后把所有的数据传送到各个片断。如果计算中包含条件控制流，那么传送的某些数据就需要遵循特殊的要求。不管如何，通过将函数片断并行执行就可以获得并行效果。因为文件管理比函数计算更小，函数被分割成块后可以以块为单位同时执行、交互、或是进行文件管理。

图 8-12 中的转换 3 是除函数分割外的另一种选择，称为数据分割。它将数据分为独立的单元，由有序函数的数个拷贝进行处理。因为计算被复制到 N 台计算机上，每台机器都分配需要处理的数据的一部分。N 台计算机上的 N 个进程都是在各自的地址空间中运行。于是，可以有 N 个不同的进程同时执行，每个进程负责处理全部数据的一部分。各个进程的数据都被载入各自的地址空间。有的时候，数据分割比函数分割更简单，因为一个顺序程序的完全拷贝总是可以简单的交给 N 台机器上的 N 个进程来执行。但实际上这是很难做到的：如果一个进程需要它的数据块以外的数据（也就是另一地址空间中的数据）来执行该函数，那么，数据分割就变得异常复杂。举例来说，若矩阵 A 与矩阵 B 相乘得到矩阵 C，那么矩阵 C 中 C[i,j]是由矩阵 A 的行与矩阵 B 的列点乘得到的。如果由网络中主机 R 上的某个进程来计算结果矩阵 C 中第 i 行的所有值，那它需要读取矩阵 B 中所有列的数据并反复读取矩阵 A 中第 i 行的数据。同样地，如果由网络中另一台主机 S 上的某个进程计算 C 中第 k 行的数据，它需要矩阵 B 中所有列的数据和矩阵 A 中第 k 行的数据。这要求一种不太可能实现的数据分割（如果不重复数据），因为主机 R 和 S 上都需要有矩阵 B 的数据。

8.3.2.2 支持分割计算

操作系统可以提供通用工具来支持数据分割策略和函数分割策略，还是必须为每一种方式提供特殊的工具？现代操作系统可以支持分布式计算，但它们没有简单采取这两种分割策略中的任何一种，而是灵活运用了两者。将来的操作系统可能更会偏向于使用其中的某一种，但现有的策略还不足以引起这样大的偏向。这也被认为是现在的操作系统的缺点之一。

分布式计算需要什么样的支持呢？在进程管理部分，通常需要考虑以下几点：

1）创建/撤销：当一次计算开始时，一般由一个单独的进程决定运行时是否需要创建和使用其他进程。操作系统必须提供一种较为灵活的机制，允许进程创建或撤销其他机器上的子进程。

2）调度：在一些分布式环境中，一个进程执行的位置是由调度程序不是进程来决定的。当一个进程在这样的环境中准备执行时，调度程序在网络中查找可以执行该程序的位置，试图实现自动地交替执行。

3）同步：传统的同步方式依赖于共享存储区。在网络中，操作系统使用消息机制代替共享内存来实现同步。

4）死锁控制：死锁检测算法根据系统资源的分配情况来决定是否存在死锁。因为，网络中资源的集合包括网络中所有机器上的所有资源，并且，检测网络中所有机器的在任一时刻的状态是十分困难的。所以，网络环境下分布式的死锁检测就成了一大难题。本书将不对此问题作进一步的讨论。

8.3.2.3 常规进程管理

顺序操作系统提供一种调用方式来创建进程，可以在本机创建和父进程相同的子进程。比如，在 UNIX 中的 fork 系统调用。在客户机/服务器的计算方式下，并没有可以直接使用的在另一台机器上动态创建进程的系统调用。当某个分布式的程序执行时，服务器端进程首先在远程机器上运行。然后在客户端手动启动客户端进程。这是商业操作系统中使用的进程创建方式。

具有划时代意义的是，有些操作系统（如 Java）使用比进程更小的单元（对象或线程）来实现对组成部分的动态执行。在网络中任一台机器的用户可以启动一个独立的应用（如 Web 浏览器）来提供执行框架。于是，一台机器上的进程可以和远程机器上的执行框架相互通信，从而执行某些计算单元（如 Java 中的 applets）。在该环境下，控制总是遵循传统的客户机/服务器方式。也就是说，在网络中设置一台服务器，由用户启动执行框架来执行计算单元并与服务器交互，以实现特殊应用程序的分布式计算。

这是现代操作系统中仍需不断完善的领域。由于它在程序模式中的巨大影响，关于它的研究涉及操作系统和程序设计语言方面。

8.3.2.4 调度

在分布式环境下。有两种主要的调度方式：

- 直接调度：由应用程序的程序员负责确定计算单元的执行位置。
- 透明调度：应用程序的进程最初作为单进程在一台机器上运行。当计算单元被创建并准备运行时，由本机上的调度程序负责和其他的调度程序交互，以决定该计算单元的最佳执行位置。

客户机/服务器模式属于直接调度模式。服务器位于网络中的某个既定位置；客户机可以位于网络中的任何地方。在这种情况下，服务器或客户机上的计算单元和该机上的普通进程一样执行。缺省状况下，由计算机的多进程调度协议提供需要的服务。

用户线程，如 C 线程和 POSIX 线程，为应用程序员提供更多的控制：允许他们将分配给父进程的 CPU 时间转给某个线程——这是由应用程序决定的。然而，线程调度并不总是能在机器间使用的，除非操作系统支持进程/线程的转化。

在透明调度中，由应用程序创建在分布式环境中执行时计算的可调度单元。创建可调度单元时无须考虑各个单元的实际执行位置。网络中，各台机器上的调度程序可以相互通信。这样，当一个可调度的计算单元处于就绪状态时，它将被传送到某台选定的计算机并在该机上执行。一般说来，一旦计算单元被安排给某台计算机，就由该机完成它的全部计算。根据多进程调度的协议，计算单元将和本机上的其它单元共享 CPU 时间。透明度最初的目的是让操作系统支持网络应用—对线程和对象透明。

8.3.2.5 进程的并行

由于计算的可调度单元在不同的机器上创建并执行，操作系统必须提供有效的方法在需要时同步它们的运行。这一领域的发展有两个主要方向：

- 直接同步：程序员利用操作系统提供的机制在需要时同步计算单元的执行。
- 事务处理和并行控制：由服务器负责同步。因此，问题集中于服务器上操作单元的效果，而与客户端请求无关。

1）同步

信号量和管程机制都把需要加锁的变量储存在共享存储区中。进程间通过操作单元对变量的不断测试和设置来实现同步。虽然设置“信号量服务器”是可以实现的，但如果利用传输层协议在服务器端测试或设置信号量，其时间消耗是在本机上读/写变量的数千倍。因此，分布式系统中必须使用更好的同步方法。

以网络为基础的分布式系统中使用消息机制实现同步。这一思想的萌芽出现于 70 年代后期。例如在 1978 年，Lamport 提出：运行在不同机器上的进程可以通过设定统一的网络时钟以实现同步。在每一个进程，消息与同步事件相联系。消息发送方为每个消息盖上“时间戳”，由“时间戳”确定事件发生的时间。在这种情况下，若是机器 A 的时钟比机器 B 慢，那么，可能认为机器 A 上的事件 a 比机器 B 上的事件 b 更晚发生，虽然实际上事件 a 比事件 b 早发生。Reed 和 Kanodia 在 1979 年提出一种试验性的系统，可以像信号量一样实现同步的效果而无需共享内存。

2）事务处理

在很多情况下，事务处理可以获得与同步一样的效果。由于相关消息的交换和消息流的发送，分布式组成成分之间的交互可以变得十分复杂。这里的相关消息流称为事务——它是一连串的指令，这些指令或者全被执行，或者全都不执行。一个事务形成一个由微操作和组成成分间的交互操作构成的特殊集合。例如，一个计算机控制的飞机导航系统可能需要许多微操作以改变飞机的航行方向。这些操作或许要调整飞机引擎速度、飞机副翼和飞行姿势。需要进行的调整的多少依赖于飞机当时的状态等。显然，这些操作需要按照事务的方式执行——或者全部执行，或者全都不执行。

进程 Pi	进程 Pj
… …	… …
Send (server,update,k,3)	Send (server,update,k,5)
Send (server,update,k,6)	Send (server,update,k,8)
Send (server,update,k,2)	Send (server,update,k,4)
Send (server,update,k,8)	Send (server,update,k,6)
… …	… …

图 8-13　修改一个含多个域的纪录

以软件系统举例。假设一个服务器程序中有一个包含 n 个域的记录的集合，该集合可以由若干客户端进程刷新。那么，当多个客户端进程同时试图修改同一个记录的不同域时就会出现问题。假定进程 Pi 按照 3、6、2、8 的顺序修改记录 k 中相应的域，同时，进程 Pj 试图按照 5、8、4、6 的顺序修改记录 k 的域。那么就存在两条客户端执行顺序，如图 8-13 所示。

在服务器方，一种可能的顺序是：Pi 修改了域 3、6 后，Pj 修改域 5、8、4 和 6，然后再由 Pi 修改

域 2、8。这时服务器上的记录 k 中域 6 的信息是 Pj 修改的，而域 8 的信息却是 Pi 的。在某些应用中，这或许是可以接受的，但更多的时候这将带来灾难——比如域 6 代表一个人的名字而域 8 代表地址。

事务处理的思想是将一串顺序操作看成一个操作命令来执行；事务一旦执行，这个操作序列将全部执行，或者是等待以后执行该事务。在上一个例子中，这就意味着进程 Pi 或 Pj 执行完毕后，另一个进程才能开始执行。

事务处理在分布式数据库中有广泛的应用。因为很自然地，在分布式数据库中，包含多个域的记录有可能被多个不同的操作修改。另外，在容易崩溃的系统或一旦崩溃将会带来灾难性后果的系统中，事务处理也是很有用的。因为，如果服务器在一个事务处理的中途崩溃，那么有可能某个记录只修改了几个域而留下不正确的数据。

程序员使用标记来定义一个事务。可以在操作序列的头尾分别加上标记，以说明这些操作属于同一个事务。当服务器端检测到事务开始标记，它可以选择执行其后的操作命令，也可以将其放置一边而不执行任何一条命令。一旦它选择执行操作命令，它就必须执行接下来所有命令，直到检测到事务结束标志。这时，它才真正提交该操作序列中所有操作的执行结果，修改服务器上信息的状态。如果这次提交不能全部完成，它就退出本次事务处理。在这种情况下，它会把服务器上的信息恢复为事务处理前的状态。客户端也可以退出某个事务，不过需要服务器端的控制。

在操作系统使用事务处理机制时，有很多机会可以实现进程的并发执行。例如，远程文件系统在页面调用或文件调用时可以使用事务处理机制，因为信息的移动可能需要在任一时刻修改客户端或服务器端各方面的信息。

在事务处理时，需要利用快照为相关资源在事务处理之前的状态保留一个副本。事务中的操作可以在这个副本或资源本身上执行。一旦操作失败，就可以利用副本将相关资源的状态恢复。如果这个事务准备开始时另一个事务已经开始，那么状态的保存就得十分小心，确保第一个事务提交后的效果被保存下来。如果某次事务被中断，就需要把资源的状态恢复为事务执行前的版本，否则就将修改后的版本视为正式版本，然后可以把剩下来的那个释放掉。

使用事务处理机制可能会引起死锁。服务器上执行可能引发死锁的事务时，若是该事务似乎不在执行，可以执行检测算法察看死锁是否存在。由于服务器端程序可以主动退出事务，当死锁出现时可以完全解除死锁的影响。当然，CPU 时间是不能挽回了。

3）并发控制

并发控制技术使得系统允许一组进程插入某个事务集合中共享资源。这样的结果相当于每个进程都对事务执行中相关的资源给予额外的控制。由此，并发控制保证了一组事务在逻辑上的顺序执行，即使事务中的操作可能被中断。

由服务器端程序给资源加锁是实现并发控制最简单的办法。一旦事务修改了资源的某一部分，服务器端程序将在该事务执行期间锁住这个资源。那么，在该事务执行完毕之前，其它进程要修改这个资源的任何一部分都是不可能的。

两个状态的加锁协议可以保证正确且有序地获得一组事务的执行结果而不会引起死锁。在第一个状态下，事务对完成该事务所需要的所有资源加锁，而不释放任何资源。第二个状态则相反，可以释放锁，但不能加任何锁。简化的方法是：在事务开始的时候申请对所有需要使用的资源加锁，到事务结束的时候再统一释放资源。

在不加选择地使用锁时会出现两个与资源加锁的粒度和死锁有关的问题：

如果资源是指某个文件，那么加锁的对象是一个物理页，一个逻辑的文件块，还是整个文件？这引起极大的争议。需要在锁的数量和并发控制的消耗之间找到平衡点。

若是一个事务在对系统资源的某些部分加锁后又申请该资源其它部分，就可能引发死锁。当然，采用上面提到的简化后的两个状态加锁方法可以避免这种情况。也可以采用其他方法避免死锁：给资源设定顺序，要求申请或释放锁时都遵循这个顺序；采用死锁检测算法；将资源设为可抢占式资源。

并发控制需要逻辑上设定一个加锁管理器。如果资源在网络中是分散的，管理器还需要获取相应节点的状态。因为网络通信的速度受到通信环境的影响，分布式环境下加锁方式的并发控制往往把加锁粒度设定得比较大。同时，可以在每个事务上加盖“时间戳”，这样就可以确定各个事务的先后顺序。

另一个难题和一般的同步问题一样。也就是说，如果事务来源于不同机器，他们的“时间戳”必须遵循某一个统一的时间，而不能简单使用本机时钟。一般说来，加锁机制中使用“时间戳”可以表明事务的先后顺序，在运行时解决冲突。

8.3.3 分布式环境下消息传递

消息传递机制是网络环境下分布式计算的基础。 高性能的应用领域总是向程序员提供网络消息传递的接口，即使这个接口同时被操作系统用来实现分布式环境中的其他部分。当然，程序员需要了解一个新的信息共享接口：IPC（interprocess communication）。

接下来说明 IPC 是如何成为根本的网络协议的。简单地说，消息就是由一个进程发送给另一个接受进程的信息块。它服务于以下两个目标：

- 通过消息机制，让一个进程和另一个进程共享信息；

- 实现消息发送方和接收方的同步。

在接收方需要设置一个“邮箱”，在逻辑上没有收到消息之前缓存不断接收到的信息。收发双方间可以是同步或异步的。在同步操作中，发送方在消息安全地被送到接收方的“邮箱”之前，一直处于等待状态；在异步操作中，发送方把消息送出后继续执行，不必等待察看消息是否真的送入接收方的“邮箱”。接收方可以是阻塞或非阻塞的。第一种情况下，接收方从“邮箱”读取消息时，接收方在消息可读（即消息已完全发送到“邮箱”）前一直等待，不执行其它操作。在非阻塞的接收操作中，无论接消息是否可读，接收方总处于执行状态。

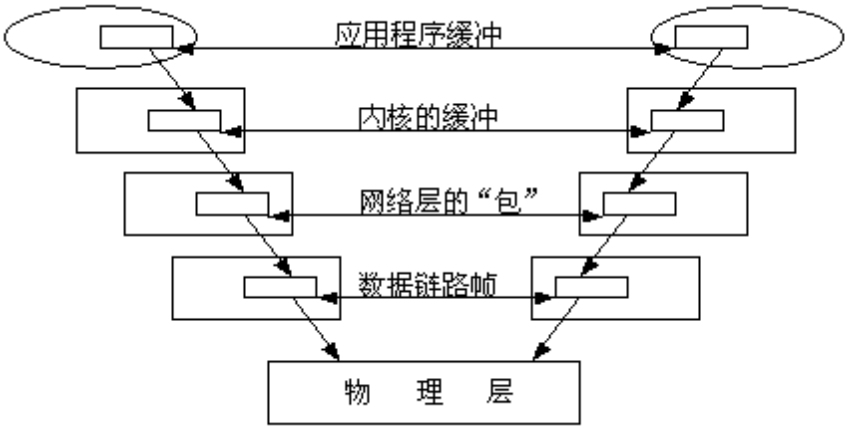


图 8-14　消息的复制

在网络体系中，为了把信息放入远程计算机上某个进程的地址空间，往往需要将信息的内容复制多次。图 8-14 在逻辑上简要列出一次发送信息的操作需要进行的复制。首先，发送方将需要发送的信息放入内部的缓冲池。然后把信息复制并放入操作系统的地址空间，这样才能把它复制放入另一个进程的“邮箱”。假设使用了网络层协议，需要把信息再次拷贝，放入“包”。接着，是把信息从“包”拷贝放入数据链路层的帧（通常放在控制器的某个缓存器）。最后还要再复制一次——把信息放入物理网络中传输。在接受方也要进行类似的操作。这样，一次信息传递至少要进行八次逻辑上的复制操作，这还不包括传输层协议和邮箱管理中可能需要的复制。

复制操作限制了以网络为基础的消息传递机制的性能。这为操作系统的设计者带来新的挑战：如何减少这种复制操作的次数。现代操作系统在实现消息传递机制时尽量减少不必要的复制。这首先要求操作系统提供应用层上的消息传递接口，让系统软件和应用程序设计员直接使用。然后，由操作系统根据它的设计策略和通信时使用到的网络协议来优化该接口的实现。以下，我们将关注网络消息传递接口。

8.3.3.1 消息传递接口

许多应用软件在传输层协议上使用消息传递接口。应用程序专家经常质问系统设计员为什么仅提供底层的发送-接收机制来支持分布式计算，而将分散在各处的信息都交由应用程序设计员负责。此外，应用程序设计员不得不使用发送-接收的操作来同步计算的各个操作部分。他们中有的人希望使用更高层次上的分布式模型——通常是分布式存储器；有的人赞成用面向应用的消息传递接口，以便获得需要的性能。

PVM（并行虚拟机）就是一种被广泛使用的消息传递接口。作为一个平台，它的优势在于可以方便地使用消息发送环境，而这个环境可以在不同的机器上实现。那么，程序员可以在同一个网络的一组不同机器上装上 PVM，然后就能利用 TCP/UDP 的实现来支持 PVM 库的程序。于是可以编写多机运行的分布式计算的应用，并且编写时不必关心各台机器上传输层协议的实现细节。

PVM 通常不由操作系统实现，而是作为用户层的一个程序库来实现的。它广泛使用 TCP 和 UDP 来支持 PVM 消息。有些组织，如 Control Data Corporation，已经在操作系统中实现 PVM，这样可以避免图 3 中的性能消耗。

PVM 提供很小进程管理能力来创建和管理 PVM 进程。PVM 任务是在并行虚拟机（PVM）上执行的有序单元。每个 PVM 任务都要调用 pvm_mytid 函数以便和并行虚拟机相连；该函数返回一个任务标识号。另外，可以使用 pvm_gettid 函数获得其他任务的标志号；使用 pvm_spawn 创建另一个任务；使用 pvm_exit 结束任务本身。可以把一些任务设定为兄弟关系，这需要使用 pvm_joingroup 把它们加入同一个逻辑组；从逻辑组中删除某个任务需要使用 pvm_lvgroup。

PVM 中有同步调用，包括传统信号 V 和等待调用 P。PVM 库中使用 TCP/IP 实现等同于信号量的机制——这儿的“信号量”共享一块存储区。

PVM 消息中包含规定类型的数据。发送方使用 pvm_initsend 函数初始化消息缓冲池。规定类型的数

据以压缩的方式放入消息缓冲，在以下的例子中 pvm_pkint 可以将整数加入消息。接收方使用 pvm_upkint 从消息缓冲池取出数据。一旦发送方的任务将缓冲池填满，它就把缓冲交给另一个任务。这过程中，pvm_send，pvm_multicast 或 pvm_broadcast 中都需要使用任务标识号。由 pvm_rcv 来接收消息，接收后需要先把消息放在某个缓冲中并对数据解压、放在本地的变量中。

下面的程序是 PVM3.3 版本中的片断。SPMD 是一个分布式计算的模型，就是用多个进程在多数据流（“MD”）上执行同一个过程（“SP”）。它使用的是数据分割策略。每台主机执行图中的代码在主机间传递令牌。

PVM 主程序中的 SPMD 算法

```
#define NPROC 4
#include <sys/types.h>
#include “pvm3.h”
main(){
int mytid;              /* my task id */
int tid[NPROC];      /* array of task id */
int me;                  /* my process number */
int i;
mytid = pvm_mytid();    /* enroll in pvm */
/* join a group ; if first in the group ,create other tasks */
me = pvm_joingroup(“foo”);
if (me == 0)
     pvm_spawn(“spmd”,(char**)0, 0,“”, NPROC-1, &tids[1]);
/* wait for everyone to startup before proceeding.                  */
pvm_barrier(“foo”,NPROC);
/* ---------------------------------------------------------- */
dowork(me,NPROC);
/* program finished leave group and exit pvm                          */
pvm_lvgroup(“foo”);
pvm_exit();
exit(1);
}
```

PVM 中的 dowork 函数

```
dowork(int me, int nproc){
int token;
int src,dest;
int count = 1;
int stride = 1;
int msgtag = 4;
/* Determine neighbors in the ring      */
src = pvm_gettid(“foo”, me-1);
dest= pvm_gettid(“foo”, me+1);
if (me == 0) src = pvm_gettid(“foo”,NPROC-1);
if (me == NPROC-1) dest = pvm_gettid(“foo”,0);
if (me == 0){
token = dest;
pvm_initsent(PvmDataDefault);
pvm_pkint(&token, count, stride);
pvm_send(dest, msgtag);
pvm_recv(src, msgtag);
printf(“token ring done\n”);
  }else{
pvm_recv(src, msgtag);
pvm_upkint(&token, count, stride);
pvm_initsend(PvmDataDefault);
pvm_pkint(&token, count, stride);
pvm_send(dest, msgtag);
}
}
```

8.3.3.2 计算范型

客户机/服务器模式是一种用来描述分布式计算的范型（或说是组织模式）。任何特定的应用都可以使用这个范型将计算分解成两个部分：以被动方式工作的服务器端和以主动方式工作的客户端。客户端根据应用的需要输入数据并调用服务器端程序。

上面提到的 PVM 就描述了一个精简的范型：它在多个数据流上执行单个程序。这些组织模式是分布式应用程序领域非常重要的部分。在应用领域使用网络十年之后，一种比较流行的范型已经出现：客户端进程将消息发往服务器后保持等待状态，直到它收到服务器发来的消息，标明服务器已经执行完。例如，在远程磁盘系统中，客户端进程向服务器发出读请求后等待，直到磁盘服务器将读的结果返回。在服务器端，服务器进程一直保持等待状态，直到它收到客户端发来的请求。当它收到这样的请求时，它执行并将结果返回相应的客户机端（或者发回消息表示请求已经执行完毕）。在该范型中，控制流的模式和本机过程调用的控制流是相同的。也就是说，当一个程序调用某个函数时，该函数执行完成后返回，然后调用者可以继续执行。这种范型被反复使用，自然引发了这样的想法：提供专门的操作系统来支持远程过程调用的范型。

8.3.4 远程过程调用

几十年以来，程序员在顺序程序中使用过程来实现模块化计算。现在，专业的程序员接受培训，设计能在公用程序接口的后面加密数据及实现功能的模块。分布式计算使得计算更为复杂，而模块化设计可以减少其复杂程度。在分布式的方式中，程序的执行需要调度的介入，执行位置对程序员透明。这就是说，如果程序员希望充分利用底层的适合分布式计算的硬件，他们就需要在新的环境下编写应用程序。然而，还没有哪种分布式计算环境成为“标准”的环境。因为，在选择最佳的计算分割方式的问题还有诸多争议，同时也缺乏标准分布式计算环境的出现。现在提倡使用多种分布式环境，包括 Java 模型，OSF DCE 和以面向对象为基础的 CORBA 实现。远程过程调用（简称 RPC）模型是对网络中顺序计算环境的主要扩展。在 Java，DCE 和 CORBA 中都有类似的机制。

8.3.4.1 RPC的工作原理

RPC 是作为网络协议的一个集合来实现的。它允许一个过程调用不同机器上的另一个过程，调用时要给出参数的拷贝。因此，RPC 在与调用进程不同的地址空间执行。RPC 是一种进程间通信的模式：初始的程序执行发消息的操作后立即执行“读”操作，而该“读”操作是等待状态的。接收方的程序在收到“调用者”进程发来的消息之前，也一直处于“读”的等待状态。接收到消息后，它执行相应的操作并将结果返回给相应的“调用者”进程。在“调用者”进程的角度看， 这只是进行了一次普通的过程调用。

图 6 中简要给出这种控制流/同步范型。在传统的过程调用中，主程序将需要的各个参数放入堆栈并调用过程。然后停止主程序，转而执行过程。首先，需要从堆栈中取出调用方给出的参数；然后，执行过程；最后，将结果放入堆栈并返回。

远程过程调用需要穿越两个不同进程的地址空间，在图 6(b)中表示为 theClient 和 rpcServer。theClient 的进程在它的地址空间中执行主程序；rpcServer 进程在它的地址空间执行远程过程。调用时，theClient 将参数打包放入消息体，后面附加有被调用过程的名字，然后将消息送至 rpcServer。theClient 将消息送出后，执行接收命令，等待远程过程调用返回结果。这和顺序调用中的控制流十分相似。

在传统方式或 RPC 范型中，调用方在被调用过程执行时都是空闲等待的。rpcServer 进程在它可以执行时是处于接收的等待状态的。当调用消息到达时，它解开包、取出参数；在传统方式中，就是从堆栈取参数。然后，rpcServer 根据过程名调用相应的过程。一旦过程执行完毕，它将返回至 rpcServer 的主程序。由这个主程序将需要返回的参数打包，通知 theClient 这次调用已经完成。

RPC 机制使得两个进程可以使用传统过程调用中的控制流模式来实现相互交互。RPC 机制允许程序员编写调用过程和被调用过程。然后，在一个进程中执行调用者过程，在另一台机器的远程进程中执行被调用的进程。程序员不需要了解任何关于消息传送或网络的知识。从图 8-15 看来，RPC 模式是由发送和接收的消息组成的有一定结构的集合。因此，通常认为它是高层的网络协议。

的值。绝大多数的远程过程调用都不支持出传址方式。

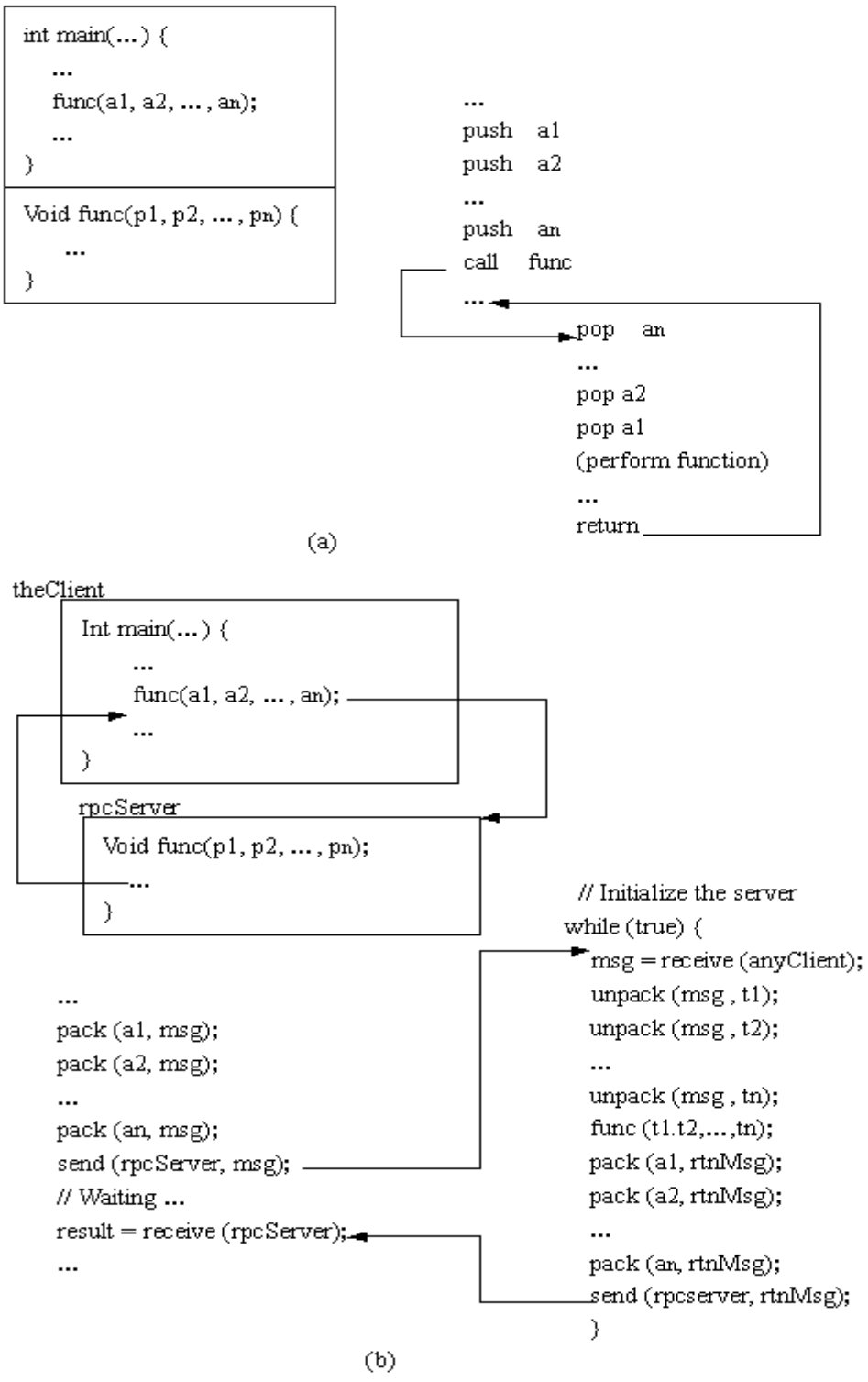


图 8-15　远程过程调用中的同步

8.3.4.2 RPC的实现

在高级编程语言中，远程过程调用应该和本地过程调用有一样的语法结构。虽然一个过程调用在远程和本地的情况下要得到完全相同解释可能比较困难，但是应尽量使它得到相同的语义解释。比如，使用传址方式的参数就会使过程的本地调用和远程调用获得不一样的语义解释，因为传址方式允许改变调用方程序变量的值。在远程过程调用中，需要为传址方式的参数增添额外的传送任务：从 rpcServer 端发送消息，在 theClient 端接收消息——这样就可以在 theClient 的地址空间中修改相应变量远程过程调用的接收方需要在与调用方相同的环境中执行。在传统的环境中，一个过程可以设置和改变全局变量。要想

在远程过程调用中实现相同的效果，同样需要在 theClient 和 rpcServer 之间增加专门的通信。一般说来，不可能在被调用过程的地址空间创建调用方的动态堆栈。

1）通用组织结构

RPC 的实现一般采用图 8-16 所示的通用结构。客户机执行 theClient 进程，它包括客户端应用的代码、clientStub 和传送机制。服务器端有 rpcServer 进程，它包括传送机制、server Stub。

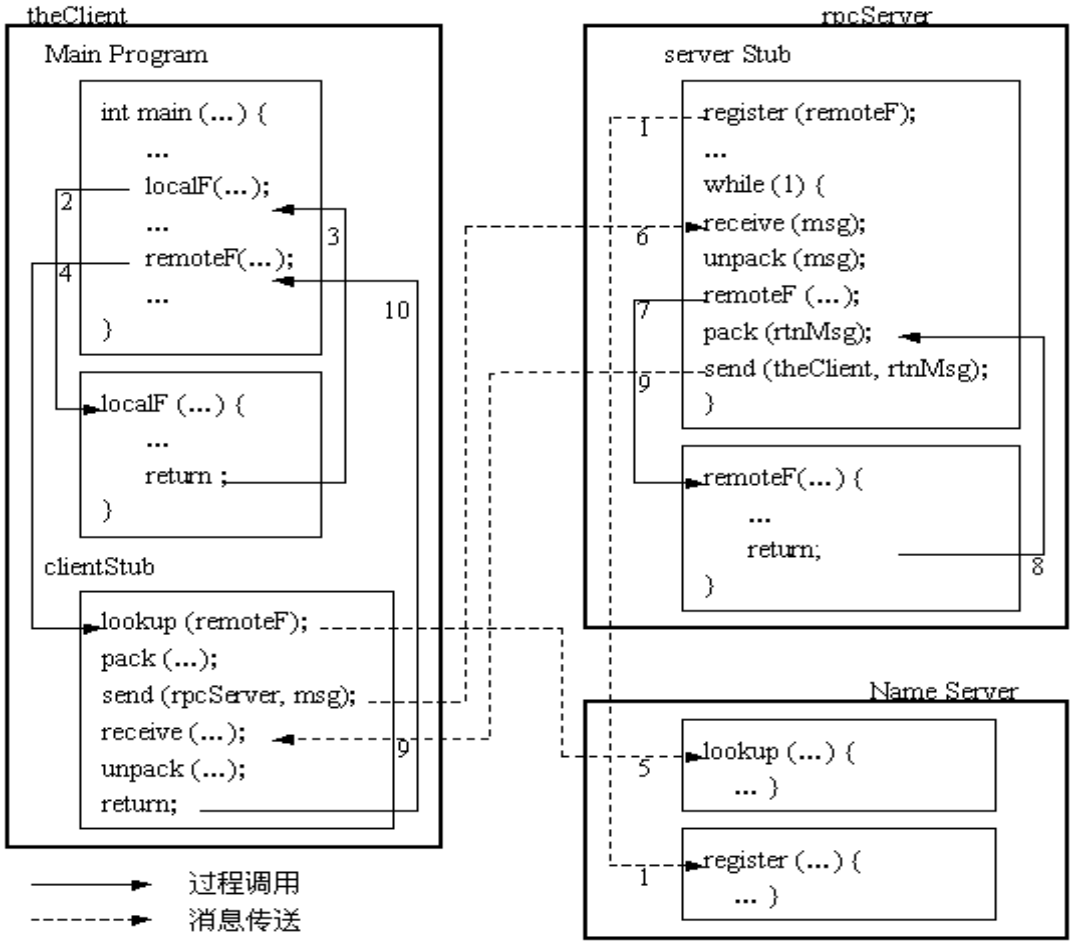


图 8-16　远程过程调用的实现

主程序和远程过程在服务器端的实现。由 clientStub 将本地过程调用转换为客户端远程过程调用协议支持的格式；由 server Stub 实现服务器端的远程过程调用协议。如图所示，theClient 执行远程过程调用时调用 clientStub 的代码。clientStub 利用 Name Server 为本机定位并执行 rpcServer 上的进程。在 Name Serve 响应查找要求前，rpcServer 进程必须已经为该远程过程注册一个通用的名字，本例中是 remoteF。只有第一次调用时需要查找。因为查找之后，客户机和服务器间将建立一个连接，随后的调用都可以利用这个现有的连接。接下来，clientStub 将参数压缩打包放入消息，并将消息送至 rpcServer 进程的 server Stub。因为一个 server Stub 可能同时为多个不同得远程过程服务，所以让接收到的消息来选择远程过程。server Stub 完成这次调用后，将结果作为参数压缩打包并返回给 theClient 进程。同时，theClient 进程处于等待状态，等待这次调用的完成。当它接收到返回的消息时，它把返回的参数解压打开并送至主程序。

2）根据本地调用为远程调用建模

使用什么机制来区分本地过程和远程过程？什么时候、如何让调用者知道远程过程服务器？需要编写客户端程序，使得远程过程调用和本地调用有同样的语义。如果过程是远程的，那么远程过程调用（RPC）机制就会要求程序员在编译、链接或与运行时区分本地过程和 RPC。如果在编译时区分，程序员需要为 RPC 使用与本地过程调用不同的接口。例如，一个远程过程可能有如下的形式：

callRemote (remoteF，a1，a2，…，aN，…) ；

此处，callRemote 是一个被链接在调用程序的地址空间的本地过程。由它的参数来确定远程过程的名字（此处是 remoteF）以及远程过程需要的其他参数。

假如程序员选择在链接时区分远程过程和本地过程，那么系统对两种过程使用相同的格式。编译器不需区分两种过程，而是自动为远程过程调用 clientStub。链接编译器被用来确保外部引用。为解决外部

引用的问题，需要一些类似于库联接信息的信息来区别远程过程和本地过程，最少只需用符号提示某个过程是远程过程。

通常的选择是在运行时把远程过程区别开。这需要与动态段落绑定相似的动态绑定支持。编译器和链接编译器都不能解决外部引用的问题，所以引用在运行时绑定。这种后期绑定要求静态绑定机制在编译的代码中留下足够的信息，以便运行时的系统可以实现外部引用。

3）远程过程定位

虽然辨识远程过程的方法很多，但都要求调用程序可以将执行远程过程的服务器定位。同样，这种定位信息也可以在编译、链接或运行时指定。不论用哪种方法，调用程序都要在调用进程的地址空间产生一个地址映射至<net, host, remoteProcedurePort>（标志着远程过程的执行位置）。如果该位置在编译时确定，那么当然，远程过程调用也在编译时被辨别出来。此时，远程过程调用需要额外的参数来标志远程过程服务器的位置。例如：

callRemote (remoteF，a1，a2，…，aN，…，internetLocation) ；

其中，internetLocation 是一个地址名，像<net, host, remoteProcedurePort>一样，标志远程过程的服务器进程的执行位置。

链接时刻的定位只有当远程过程调用在编译或链接时被辨识才有意义。同样，这种静态绑定的方式在概念上与编译时刻绑定相同。在链接时刻，远程过程服务器由外部的符号定义来定位；就程序的执行而言，这是静态的。

远程过程在网络中的定位采用动态绑定，这是通常使用的方法。图 7 中使用的就是动态绑定。ClientStub 作为中介，与调用程序是静态链接的。而本章前面曾经提到，ClientStub 利用运行时刻的信息来查找远程过程服务器的位置并与之建立联接。当第一次执行远程过程调用时，ClientStub 查询命名法则为远程过程服务器定位。在随后的调用中，远程过程服务器作为已知信息使用，无需再次查找定位。

在动态绑定中，ClientStub 是在过程编译的时候确定它与该过程的关系的。一旦调用进程确认某个过程是远程过程，它就会把自己和 ClientStub 绑定。例如，在链接时很容易地将 ClientStub 静态绑定到某个调用程序，而位置则动态确定。

4）stub 的产生

ClientStub 是如何自动产生的？现代程序设计语言提供过程接口模块来规定过程的调用顺序。ANSI C 或 C++中的函数原型就是过程接口的例子。用模块来实现某个过程称为输出；在模块中使用该过程称为输入。接口模块由过程名和参数唯一确定，因此有足够的信息来产生 ClientStub。Stub 编译器可以使用接口模块调用本地传输机制，从而实现调用和返回时的内部转换，以及参数的压缩打包（以便加入网络消息中）。在运行时刻，ClientStub 使用“名”服务器（NameServer）为远程过程服务器定位，然后才与服务器交换消息。当调用远程过程时，若是服务器的位置已经在地址空间绑定，ClientStub 就可以利用消息的发送和接收来模拟本地调用。它将参数压缩打包并送至 serversStub。然后，ClientStub 或说是客户端进程开始等待，直到远程过程调用完成。

5）网络支持

传输机制实现了网络消息传送。由于可靠性的要求，现行的实现方式在特殊目的的远程过程调用协议中使用自寻址数据包协议。操作系统设计员证明这样的实现是正确的，因为远程过程调用协议并不需要使用 TCP 提供的一般虚拟通信。当收发双方是确定的 clientStub 和 serverStub 时，可以很容易地为之度身定制一组协议。

在服务器端，输出过程的每个模块必须准备好接收远程调用。这需要服务器端包含一个代理调用进程，即 serverStub，来接收由客户端 clientStub 发来的调用请求并且调用本地过程。利用接口模块和实现过程时在其地址空间中指明的出口，可以实现 serverStub。在一般情况下，服务器在“名”服务器中注册每个过程，因此允许 clientStub 在运行时刻为过程定位。注册时要把过程名加入“名”服务器，并将某个内部标志符映射至该过程。

在调用时，clientStub 将调用的参数压缩打包后放入消息中，然后将消息送至“名”服务器指定的某个网络端口。接着，服务器的传输部分把消息送到 serverStub，由 serverStub 把其中的参数解压，找到相对应的过程并调用它。当过程返回时，先由 serverStub 将执行结果压缩打包并返回给那个 clientStub。再由 clientStub 将结果解压后返回给调用者。

在这种机制下，传值方式的参数比较容易处理。传址方式或传名方式的参数就比较难处理了，因为它们使用时需要对参数求值并传给服务器。对此，不同远程过程包有不同的实现途径。然而，不管哪种实现方法都需要在 clientStub 和 serverStub 之间进行网络通信。

远程过程调用在不同机器的分布式进程中十分实用，但它不支持并行计算。当调用一个远程过程时，调用者在远程过程执行时处于阻塞等待状态。在实现远程过程调用时，性能方面主要考虑执行时间的问题。因为远程过程调用提供后期绑定，所以应尽可能的减少它的性能损耗。即使这样，远程过程仍然广泛运用于现代分布式应用。因为，它实现了传统计算模型而几乎不需要了解分布式机制或策略的知识。

8.3.5 分布式内存管理

与进程面向存储系统的一般接口不同，图 8-17 中的接口包括操作系统对远程磁盘访问和远程文件访问的支持。

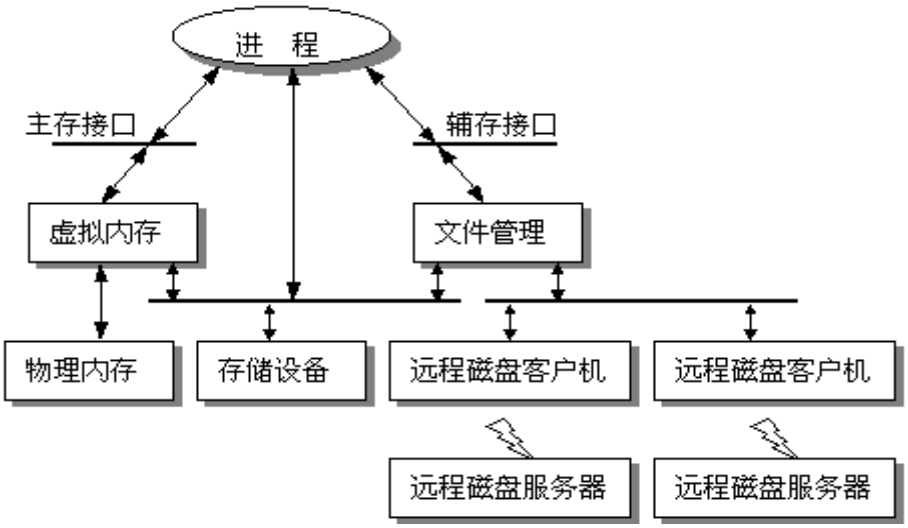


图 8-17　存储系统的标准接口

操作系统设计员使用不同的模型来处理远程内存：增加一个新的接口以便程序直接使用远程主机的主存（见图 8-18）。

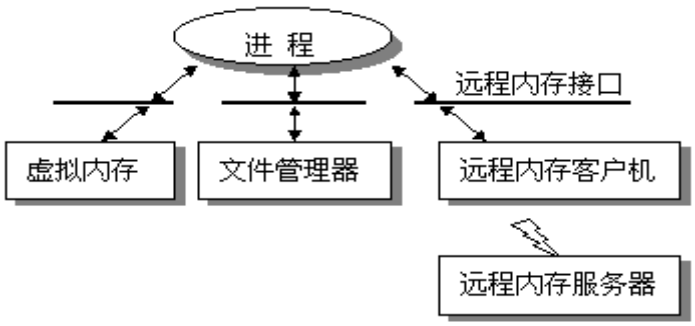


图 8-18　远程内存的接口

在图 8-19 中，分布式对象是分布式共享内存的特殊例子；一些操作系统设计员使用单独的本地对象接口（如 CORBA 中），而另一些同时提供本地和远程对象接口，这样本地对象的性能就不会由于使用通用协议而受到影响。图 11 中简要说明分布式虚拟内存在网络中如何扩展本机系统的页面调度系统。页面调度服务器管理辅存，这样丢失的页面可以在服务器上找回，而不是当页面出错时从本地磁盘上找回。

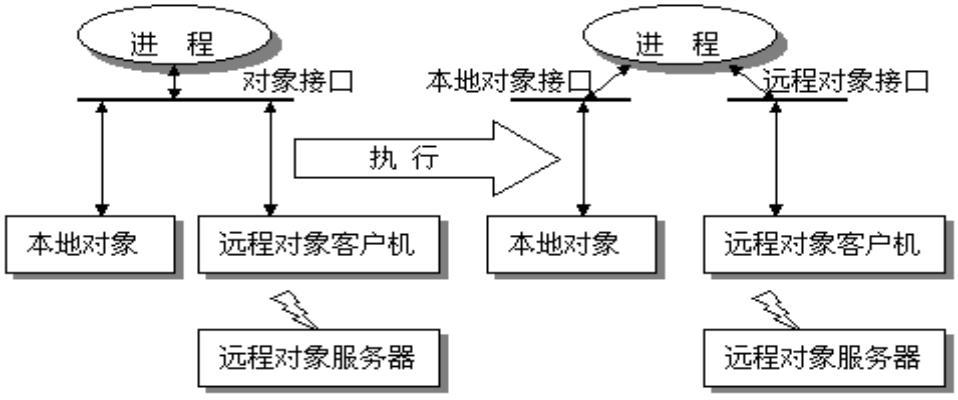


图 8-19　分布式对象

有多种选择来代表远程地址空间。对程序员而言，哪一种是表达分布式内存地址空间的最佳方法？分布式内存是共享内存，因为它是由多台机器上的两个或两个以上进程使用的。因此，在提供远程存取的同时，必须确保共享。与面向文件的接口相比，分布式内存的典型出现是主存块的集合。“块”的说法是从程序设计语言发展而来的。“块”可能是数据结构中的对象，或者是动态分配的内存。相连机器上的操作系统分配内存空间，并将共享的内存块映射到应用进程的地址空间。

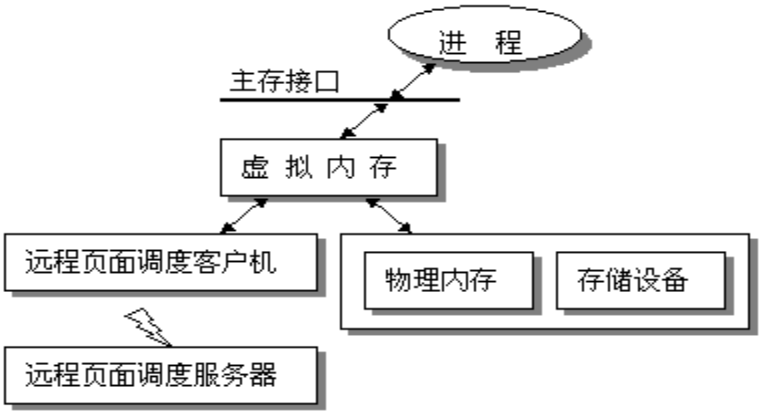


图 8-20　分布式虚拟内存

有两种类型的结构可以实现分布式内存：

- 多计算机：多处理器模式也可以说是“多计算机”，因为它使用多个不同的处理器，且每台处理器都可以访问机器的全部内存。“多计算机”中，各台机器对内存的存取是不均匀的（称为不均匀内存存取，nonuniform memory access），因为不同位置的内存的存取时间与处理器有关。
- 计算机网络：它提供逻辑上的共享内存接口。这个接口使用以信息包为基础的网络来支持对内存块的存取。

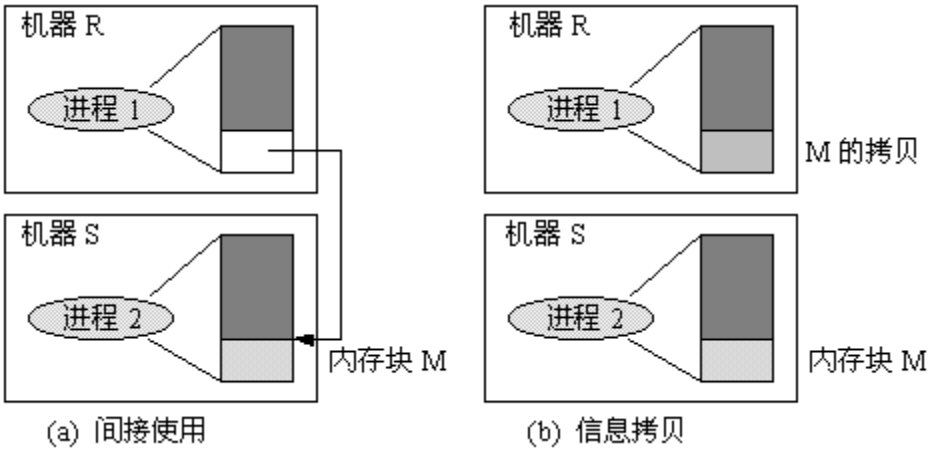


图 8-21 共享内存块

图 8-21 中说明了两种常用的分布式内存设计的方法。在图 12(a)中，机器 S 将内存块 M 分配为共享内存，由机器 R 的进程 1 和机器 S 的进程 2 共享。当进程 1 涉及到分布式内存时，相应的存取要求转化为机器 S 上的一个服务请求。于是，作为分布式内存客户端的机器 R 和服务器端的机器 S 之间就有通信要求，这种通信使用相应的网络协议。

图 8-21(b)给出另一种实现途径。机器 R 在进程 1 的地址空间内复制内存块 M。于是，进程 2 对内存 M 的存取在本机上执行，而进程 1 对内存 M 的存取也是在进程所在的机器 R 上进行。显然，当进程 1 对内存 M 进行“写”操作时出现问题：机器 S 和 R 上内存块 M 的拷贝不一致。因此，分布式内存系统必须提供某种机制，确保内存块拷贝的一致性。

设计分布式内存系统时，需要考虑如下的基本问题：

- 内存接口：网络方式下，需要为内存提供一个单独的接口吗？或者，利用现有的主存接口存取远程内存？
- 位置透明度：关于远程地址空间中的位置，一个进程可以了解到什么程度？

- 共享单元：在地址空间中，共享单元应该是什么？例如某个数据结构、页面、段，或者其他的单元。
- 命名管理：信息在输入输出时需要对共享单元命名（以便唯一标志该单元）。这又该怎么处理，或者说，它应该遵循什么规则？
- 执行效率：假设有两个进程，并且它们的地址空间位于不同的机器。那么，采用哪种方法来存取远程共享内存效率更高呢？

至今，研究者们对这些问题的最佳解决方案还没有达成共识，对分布式内存实现途径的探讨依然十分热烈。以下将给出分布式内存的三种实现：远程内存，分布式虚拟内存和分布式对象。

8.3.5.1 远程内存

远程内存是一种更广范围的实现途径。在这里，分布式内存的存取需要使用一个与普通的主存接口或文件接口不同的接口。远程内存接口是从传统的冯·诺依曼计算机的计算模型扩展而来的。例如，程序员直接把程序的某些部分映射到相应进程的共享内存。这需要在扩展程序设计语言的同时，由程序员在编译时确定共享数据结构。比如，可以定义一个数据结构并将它标记为共享的。

1）内存接口

设计内存接口时，会遇到以下两个比较有争议的问题：

- 如何定义内存为远程的？也就是说，如何将远程内存映射到一个进程的地址空间？
- 在内存定义为远程的之后，如何对之进行读/写操作？

内存接口设计是一个不断发展的领域。它需要新的使用远程内存的程序模型发展并被大众接受。而现在的操作系统中对此还没有占主导地位的实现途径。这儿给出两种比较著名的实现方法作为例子：POSIX 系统调用关于共享内存的扩展接口；Linda 程序设计语言的扩展。

（1）POSIX 共享内存接口

POSIX 系统调用接口定义一个共享内存的扩展接口。不同机器上的多个进程使用这个接口来共享这些机器上的内存。这和商业性的共享内存多处理器中使用的内存接口很相似。从概念上说，它可以运用于支持远程内存的计算机网络。

使用系统调用 shmget 可以创建一个共享内存段。这个调用返回一个整型变量给调用它的进程，以便唯一标识它所创建的共享段。一旦创建共享段，其他进程可以使用系统调用 shmat 与共享段绑定。这个系统调用返回整型标志符。进程可以使用 shmdt 来移动共享段。由此可见，使用共享内存段可以方便地在进程间传送整型标识符，而不需要特别的规定。系统调用 shmctl 有较多用途，它可以把特殊命令传给内核。例如，一个进程可以通过 shmctl 巧妙利用共享内存段中的某些部分，或是将共享段加锁或开锁，又或者是取消共享段。

关于内存接口设计中的问题，这种接口从历史悠久的 malloc 的用法中获得灵感：内存被引入进程地址空间，由 shmget 返回一个整数作为该内存块的“全局名字”。而 shmat 返回一个无类型(void)的指针，指向共享内存段的首地址。于是，可以像 C 程序中利用 malloc 获得的内存一样，通过指针使用内存。

总之，POSIX 系统调用接口的方法就是增加一个外在的远程内存接口。

（2）Linda 程序设计语言

Linda 程序设计语言采用和 POSIX 类型的接口完全相反的做法。它试图以直接扩展传统程序模型的方法来使用远程内存。在 Linda 中，数据以数组形式存储，使用时通过“存取关键字”和数组的各个域相联系。与传统的读写操作不同，为了能够灵活操纵内存，Linda 提供一组操作：可以从数组空间中读取、添加或移除数组。数组空间中的信息可以通过移除、更新或替换的方式加以修改。

例如，一个名为 studentCount 的整数以 course 作为它的关键字。那么，可以通过以下形式的代码段来更新它:

```
tmp = read ( course=3753 )；
++tmp.studentCount；
write ( tmp，course=3753 )；
```

虽然这种实现方法看起来很笨拙，但是它提供了一个便于编译的应用程序接口，可以在网络中存取内存。

另外，使用数组的更新操作可以直接实现数据读写的并行。和文件接口一样，这些内存接口的扩展接口与程序设计语言相独立。有的研究者，如 Carriero 和 Gelernter，认为 Linda 定义了一种程序设计语言，而不是一个操作系统。特别是，可以用 Linda 将多种顺序程序设计语言扩展为并行程序设计语言。

2）内存单元的大小

由于远程内存是设计时指定的，自然，内存单元的大小也由程序员直接定义，而不是系统设定的。例如，在 Linda 中，数组被定义为共享单元，数组的大小也就是共享单元的大小。

3）位置透明度

远程内存可以位于网络中的任何位置，所以系统必须有能力在地址空间中为远程内存定位。纯粹的位置透明意味着需要设立全局地址空间，以便完全隐去某个地址的物理位置。进程使用本地地址空间中的某个“名”来访问远程内存，这个“名”指向某个全局地址。全局地址静态或动态地与恰当的网路地址相绑定。

例如，若是在传输地址< net, host, port >中分配远程内存，进程必须在给定的服务器地址中指出内存块及偏移量。假设服务器管理一个以上的内存块，本地“名”空间中的简单的地址格式变为如下形式：

<< net, host, port >, block, offset >

Linda 中，对共享的全局地址空间使用已结合的存取操作来实现位置透明。程序可以把一个数组写 入全局数组空间，而无需关心内存在网络中的物理位置。

作为选择，系统可以设计成在编译、装载或运行时绑定。编译时刻绑定本质上需要地址完全可见并预先嵌入软件中。因此，很少有系统采用这种方法。装载时刻的绑定中，网络位置由链接编辑器在处理外部引用时确定。这意味着，使用者需要在程序链接和装载时就给出远程内存的位置。由于运行时刻绑定更为灵活，这种方法也很少使用。在运行时刻的绑定中，程序在编译时给出足够的信息，利用某个“名”服务器在第一次使用远程内存时绑定该内存块的位置。随后对该内存块访问可以重复使用已有的绑定。运行时刻的绑定需要系统提供相应的机制，相当于将段名映射到某个段的机制。

因为共享内存的实现比较明了，可以将特殊的语义运用于共享内存中的数据存储。数据的一致性需要另行规定，这里明确不保证共享内存中的数据的一致。于是，一致性问题交由程序员负责，而不再是系统。在需要时，程序员可以加入同步机制。

也可以引入新的抽象数据类型语义学来迎合特殊应用领域的需要。例如，可以为针对计算范型特殊的机制提供单生产者-多消费者模型。由生产者向共享内存写入信息；信息的拷贝被“消费”，也就是逻辑上从内存中移出数据。

通用远程内存系统的缺点主要和接口的透明度有关。分布式内存是特殊的，它向程序员提供最大限度的灵活性。远程内存和本地内存必须区别对待。分布式虚拟内存使用虚拟内存接口来解决这个问题。这个接口不需要引入特殊语法或太多的特殊语义。

8.3.5.2 分布式虚拟内存

共享远程内存本身就有额外的抽象的一方面。即使它在物理上位于远程机器上并且存取时需要使用网络协议，它也被当作本地内存来使用。虚拟内存把它本身的内存映射机制作为操作中固有的一部分，用来实现分布式内存。

虚拟内存的使用和物理内存不同，因为每个虚拟地址在使用物理内存模块之前，被映射为一个物理地址。在分布式虚拟内存中，分布式虚拟地址被映射到一块共享虚拟地址空间。如图 8-22 所示，若是它装载于本机的主存中，共享单元地址直接确定为目标内存的位置；或者，确定为全局的辅存中的位置。

当进程地址空间的私有部分的页面被装载时，它是从进程所在机器辅存映像中得到的。当映射到虚拟地址空间的共享部分的页面被装载时，它是从共享的全局辅存的位置装载的。图 8-22 中，共享的全局辅存位于独立的一台机器上。有争议的是：共享页面的使用需要一个服务进程，这个服务进程应该位于本机还是远程机器？

在程序员看来，分布式虚拟内存是十分吸引人的，因为它在分配和使用时可以像本地虚拟内存一样处理。而内存管理员认为，构造虚拟地址空间需要解决普通网络中的命名和透明性的问题。

全局辅存服务器在适当的“名”服务器上注册它可以提供的辅存，这样，客户机才可能在运行时刻与服务器绑定。可以有多种方法实现绑定。最简单的方法是由程序员调用初始化程序，确定适当的“名”服务器。初始化过程查找页面调度服务器的传输层地址，<net, host, port>，和它建立联接，然后才允许进程开始执行。当本机的内存管理器检测到页面丢失的错误时，首先要判断该页面是分布式内存还是本机的虚拟内存。如果是分布式内存，可以利用已经建立的联接找回丢失的页面并将它放入本机主存。

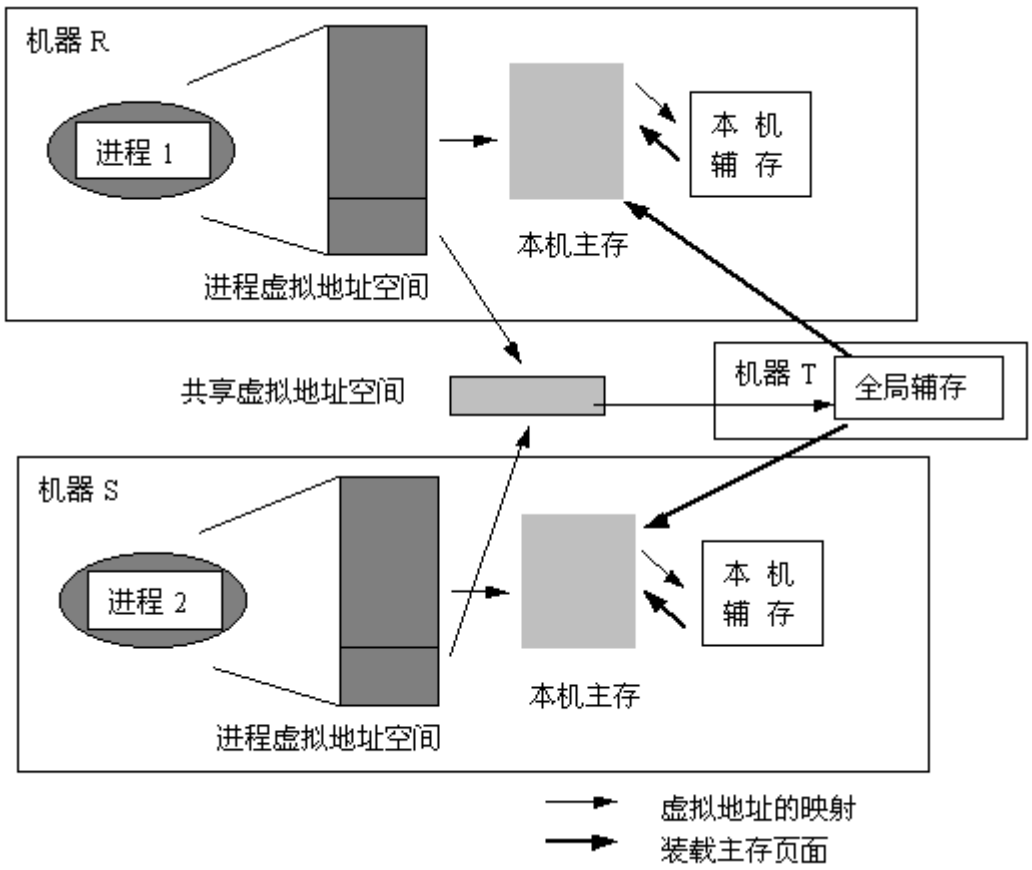


图 8-22　分布式虚拟内存

页面在各台机器的主存上可能会有多个拷贝。这些拷贝如何与全局页面调度服务器上的页面映像保持一致？集中或分布式的算法都需要面对这个问题。使用页面缓存的方法来解决一致性问题需要采用页面同步和页面所有权策略。页面同步利用失效的方法保证一致性：当进程“写”缓存页面时，所有使用该页面的其他进程全部“失效”。具体而言，每个页面都由它的“所有者”处理器来“写”页面(“所有者”指最近“写”页面的进程)。一次“写”的错误使得该页面所有的缓存拷贝失效；对某个处理器而言就是引发一次错误，需要修改对这个页面拷贝的访问。这样，该处理器获得页面的所有权，可以执行“写”操作。

动态的“所有权”可以利用集中或分布式的内存管理器来实现。集中式的内存管理的实现方式需要在网络中的某个处理器上设置一个内存管理器。由这个管理器保存全局信息，如现有的存取权限、“所有者”和加锁的信息。这很容易实现，但是也容易成为系统瓶颈。

分布式内存管理器采用这样的策略：共享页面的集合被静态分割并分配给不同的管理器。由客户机负责使用恰当的服务器来满足页面分配和交换的请求。作为选择，客户机也可以不获取服务器的位置，而是通过广播协议来决定“所有权”。

分布式虚拟内存的实现还有待研究，但试验性的实现结果已经足以鼓舞人心。它可能会在分布式存储的设计中占主导地位，因为应用程序会渐渐放弃传统的 I/O 模式转而与它紧密结合。

8.3.5.3 分布式对象

面向对象的程序设计已经比较流行的一种定义计算的模型，因为它本身将消息作为各个组件间交互的唯一方法。这个模型最大的优点也就在于这种直接的消息传送方式。使用面向对象程序设计语言来编写应用程序需要有这样的观点：数据被封装在对象中；以消息的方式实现交互。这个模型的缺点是：面向对象程序设计语言的语义依赖于单一地址空间上的顺序操作，而这样做妨碍了它在分布式环境中的执行。例如，C++在定义参数如何传送到成员函数时，其语义解释建立在单线程和单一地址空间的基础上。

可以建立软件系统来扩展语义，为分布式计算定义更合适的面向对象程序设计语言。在这些系统中，每个对象有自己的地址空间，所有对象的联合组成计算的地址空间。因此，对象名作为可用的“名”提供给共享的“名”空间。如果在网络中对对象名加以管理，那么面向对象的模型就是实现分布式计算的一种可行的办法，同时也就实现了分布式内存。

由于对象可以小到一个整数、也可以大到一幅图像，操作系统很难有效地管理它。而对于分布式内存而言，对象在网络中的移动成为难题。因为它希望对象的移动可以带来下面的效果：当某个对象频繁

使用另一个对象时，这两个对象装载在同一台机器上。因此，对象的移动是实现分布式对象的首要问题。与负载平衡有相似要求的同时，对象的移动并不仅仅是移动地址空间的某些部分，还要在保证在全局地址空间中可以使用对象名。

Emerald 是分布式对象系统的一个例子。它是程序设计语言和系统的结合体，支持对象的移动。Emerald 的设计者们认为：不同大小的对象在实现时也应该有所区别；但对象的接口应该是一致的（参见图 10）。于是，Emerald 为本地和远程的对象提供统一接口。由编译器辨别两者，并允许在运行时再把大的对象移入。Emerald 将全局对象与本地对象区别开来，为它产生一个对象描述符以便和全局名字相结合。如果对象是远程的，描述符中将包含足够的信息，使得本地的消息可以送至该对象。由于对象是动态的、可移入的，描述符中还必须包含参照计数，当对象解除分配时，可以凭借它恢复描述符空间。全局对象的地址可能潜在于多台机器上，这依赖于系统跟踪对象移动的能力。

8.3.6 分布式系统小结

本节介绍了操作系统支持分布式计算的技术。这里，操作系统必须提供基本的工具，允许在网络环境下使用进程管理器；还需要直接扩展本机操作系统的功能以实现同步。这就引入了事务的概念，可以为网络中的事件建立先后顺序，而不再依赖传统的共享内存的同步机制。

消息是支持分布式计算的常用方法。可惜的是，以消息为基础的程序设计并没有被广泛运用。最近几年在科技设计领域有这样的发展趋势：使用消息来支持高速计算。虽然许多专家赞成用更好的实现方法来替代消息传送机制，但现代高速计算仍然到处使用消息传送机制。

远程过程调用是支持客户机/服务器计算模式的中流砥柱。现代远程文件服务器(如 SUN NFS),Windows 系统（如 X windows），以及其他的分布式服务器都使用到远程过程调用。商业系统早在十年前就提供了远程过程调用的程序设计方式，因此许多商业应用软件中用到远程过程调用。如今，远程过程调用已经成为实现分布式应用的重要工具。

操作系统的发展渐渐出现这样的趋势：由操作系统提供分布式内存。通过建立专门的远程内存接口、或是在现有的主存接口下加入分布式虚拟内存模块，可以方便地将分布式内存加入计算环境。由于分布式虚拟内存的实现方式使用现有的接口，它更受到应用程序员的欢迎，也更为可行。远程内存的实现十分普遍，它向程序员提供更多数据分配方面的控制，并且为各种应用领域提供更广阔的发展前景。由此看来，分布式虚拟内存必将成为主流趋势。

CH9 操作系统结构

9.1 操作系统设计目标

随着软件大型化、复杂化，软件设计，特别是操作系统设计呈现出以下特征：一是复杂程度高，表现在程序庞大、接口复杂、并行度高；二是生成周期长，从提出要求明确规范起，经结构设计、模块设计、编码调试，直至整理文档，软件投入运行，需要许多年才能完成；三是正确性难保证，一个大型操作系统有数十万、甚至数百万行指令；参加研制的人员有数十、数百，工作量之大，复杂程度之高可想而知。一个操作系统即使开发完成，仍然是无生命的，必须要开发该系统下运行 的大量应用程序，待应用程序开发问世后，用户还必须通过文件、培训及实践去学会操作和使用。这意味着用户拥有并使用的是 10 年或 20 年以前的操作系统技术。当一个操作系统投放市场后，硬件技术却又在迈进，多 CPU 的计算机出现，计算机的处理器更快、内存更大、外设种类更多，使用多种文件系统，采用多种网络协议，于是操作系统设计和开发者们就要急急忙忙扩展已有系统以利用新的硬件性能。因而，现代操作系统在设计时必须要努力追求以下设计目标：

1、正确性。随着计算机应用的广泛深入，要求操作系统提供越来越强的功能，导致正确性难以保证。引起操作系统不正确的因素很多，其中最重要的是并发性、共享性和随机性。并发性使得系统交替地执行用户的指令序列；共享性使得进程或模块之间产生直接或间接的制约关系和交互作用；随机性表现在中断发生的随机性、用户作业到达系统的时刻及作业类型的随机性。因此，必须对操作系统的结构进行研究，一个良好结构的操作系统不仅能保证正确性而且易于验证。

2、高效性。操作系统自身的开销，如占用的内存和辅存空间，占用的 CPU 时间等，对系统的效率有很大影响，减少系统开销，就能提高整个系统的效率。特别是操作系统的常驻内存部分，如短程调度、中断处理、I/O 控制等，它们均处于频繁活动状态，是影响系统效率的关键所在，所以更要精心设计。

3、可扩充性

操作系统进入运行维护期后，不可避免地要进行改进，其中的一种改进是扩充新功能。例如，支持新的输入设备 CD-ROM 读入器；增加支持新的网络协议的通信能力；或支持新的软件技术，如图形用户界面及面向对象的编程环境。

为了保证操作系统代码的完整性，现代操作系统的设计均采用以下思想：构造一个提供主要操作系统功能的系统内核，在此基础上开发服务器，以提供操作系统的其它功能，包括完整的 API。操作系统的核心保持不变，而服务器可以修改、增加，这一设计思想最早由 Garnegie-Mellon 大学的 Richard Rashid 博士等在开发 Mac 操作系统时采用。后来称这种操作系统结构为微内核结构。

Windows 和 OS/2 的设计均借用了上述设计思想，它们由特权执行体（核心态）及一系列被称为保护子系统的非特权服务器（用户态）组成。通常操作系统仅在核心态执行，应用程序除了调用操作服务外，仅在用户态执行。但在 Windows 等操作系统中，保护子系统像应用程序一样在用户态运行，这种结构使保护子系统在不影响系统完整性的情况下被修改或增加。

此外，现代操作系统的设计中还采纳了许多其它技术以保证其可扩充性：

(1) 模块化结构。操作系统的执行体包含了一系列单个部件，它们仅通过功能接口相互交互，新的部件可用模块化的方式添加到执行体中，通过调用现有部件提供的接口来完成其任务。

(2) 用对象来代表子系统资源。引用对象来代表系统资源，使系统内的资源易于统一管理，使系统的可扩充性、可维护性好。

(3) 可加载驱动程序。现代操作系统支持运行需要时才装入驱动程序，新的文件系统、新的 I/O 设备及新的网络协议，可通过写一个文件系统驱动程序、设备驱动程序或传输驱动程序并将其装进系统得到支持。

(4) 远程过程调用（RPC）机制。这种机制允许应用程序调用远程服务，而无需考虑这些服务在网络中的位置。新的服务可被添加到网络上的任何计算机中，而且能立即为网络上其它计算机中的应用程序使用。

4、可移植性

可移植性与可扩充性是密切相关的。可扩充性使操作系统很容易被增强，而可移植性使整个操作系统以尽可能少的改动移植到一个具有不同处理器或不同配置的计算机上。硬件的发展很快，Intel 80386 和 80486 芯片与许多流行的 CPU，一道被称为复杂指令集计算机（CISC），其特点是具有大量机器指令，每条指令都很精细有用，Intel 及其他制造商已经在 Intel 的 CISC 技术基础上，在 80 年代中期，开发了另一类被称作精简指令集计算机（RISC）的 CPU。RISC 芯片与 CISC 芯片的主要区别在于 RISC 芯片只提供少量简单的机器指令，大大简化了指令系统，可以在高频率时钟下运行且执行速度非常快。在 CISC 与 RISC 的竞争中，使操作系统开发者们意识到要充分利用这些硬件特性，就要为 90 年代开发研制新的操作系统，它易于移植，能轻易地从一种硬件平台移到另一种硬件平台。

写一个易于移植的操作系统如同写任何可移植代码一样，必须遵循某种规则，尽可能多的代码用所

要移植到的全部计算机上都有的语言书写，这意味着，应该用高级语言来编写代码，最好是已经标准化了的高级语言。

其次，应考虑要将软件移植到什么样的硬件环境中，不同硬件对操作系统提出 了不同的限制。例如，在 32 位寻址的机器上建立的操作系统不能被移植到 16 位寻址的机器上。

第三，尽可能减少或消除直接访问硬件的代码数量是很重要的，硬件依赖性有多种，如直接操作寄存器及其它硬件结构等。同时，与硬件有关的代码总是不可避免的，应该将这些代码孤立地放在一些容易找到的模块中，而不应分散在操作系统的各处。例如，可将一种与硬件有关的结构隐藏在模块中，操作系统的其它模块通过调用这一模块来控制硬件。当操作系统移植，仅有封装硬件特性的模块需要改变。

为了达到可移植性，现代操作系统设计时，还包括以下特性：

(1) 采用 C 语言或 C++语言编码。开发者们选中 C 或 C++语言的原因在于它有国际标准，并且，C 编译器及软件开发工具已被广泛使用。汇编语言仅被用于系统中那些必须直接与硬件交互的部分（如中断处理）和那些要求最优速度的部分（如多精度运算）。

(2) 处理器分立。操作系统的某些低层部分必须存取与 CPU 有关的数据结构及寄存器，执行这些操作的代码被封装在一些模块中，更换 CPU 时，只需更换这些模块。

(3) 平台分立。与硬件平台有关的代码封装在被称作硬件抽象层（HAL）的动态连接库，使得高层代码从一个平台移到另一个平台时无需改动。

5、可靠性

首先，要求操作系统是坚固的，对软件出错或硬件故障均有可预计的反映；其次，操作系统应主动地保护自身及其用户程序遭到有意或无意的破坏。

异常处理是一种捕获出错情况并予以统一处理的方法。这是现代操作系统抵御软件及硬件错误的主要手段。每当异常情况发生时，系统捕获这个事件，异常处理代码被激活以响应和处理该事件，确保没有未查到的错误会侵犯用户程序和操作系统本身。

可靠性还由操作系统的其它特性来进一步增强：

(1) 采用模块化设计。系统各个部件通过预定接口交互，例如，可把内存管理全部删除，而用新的具有相同接口的内存管理替换。

(2) 新型文件系统。能从各种磁盘错误下恢复数据，包括出现在关键盘区的错误，它采用冗余存储及基于事务的方案来保存数据，确保可恢复性。

(3) 虚拟存储器。虚存至实存的信息映射由系统来控制，可防止一个用户去读或写另一个用户占用的内存。

6、可伸缩性

操作系统被设计成一个可伸缩的、多道处理系统，使用户能在单 CPU 计算机与多 CPU 计算机上运行同一个应用程序。在最佳的情况下，用户可以全速同时运行若干个应用程序，计算若干个应用程序可通过将其工作分配到不同 CPU 上而提高性能。

7、分布计算

随着个人计算机的普及，计算方式有了很大改变，以前使用单一主干机的场合，现在十分便宜的微机成了标准工具。增强的网络功能允许较小的计算机相互通信，共享网络中的硬软件资源。为了适应这种变化，现代操作系统中应装入网络功能，而且还应提供一种工具，使应用程序能分布在多个计算机系统上工作。

8、认证的安全性

操作系统提供各种安全机制，如用户登录、资源配额及对象保护。美国政府还为政府应用程序规定了计算机安全准则，达到某种政府批准的安全级的操作系统可参与竞争。安全性准则又定义了一些必要的能力，如保护一个用户的资源不受他人侵犯，建立资源配额以防止一个用户得到全部系统资源。

美国政府认证的安全级从 D 级（最不严格）到 A 级（最严格），其中 B 和 C 又分别有几个子级。如 Windows NT 的设计目标是 C2 级，意味着“自由决定的保护并通过引入监视功能对用户及其行为负责”，即是该系统的资源拥有者有权决定谁能存取资源，而操作系统能检测出何时数据被存取及被谁存取。

9、POSIX 承诺

在 80 年代中期，美国政府部门开始定义 POSIX 为政府计算合约的认准标准。POSIX 虽然是被不确切地定义为“基于 Unix 的可移植操作系统界面”的首字母缩略语，其实它代表了 Unix 类型的操作系统界面的国际标准集。POSIX 标准鼓励制造商实现兼容的 Unix 风格界面，以使编程者容易将其应用程序从一个系统移到另一个系统。为满足政府的 POSIX 认证要求，NT 将设计成提供一个可选的 POSIX 应用程序执行环境。Minix 和 Linux 操作系统都符合 POSIX 标准。

9.2 操作系统的构件

操作系统的结构有两层含义，一是操作系统程序的数据结构和控制结构；二是组成操作系统程序的构件过程和方法。我们把组成操作系统程序的单位称作操作系统的构件。剖析现代操作系统，其构成操作系统的基本单元除内核之外，主要还有进程、线程、类程和管程。采用不同的构件和构造方法可组成

不同结构的操作系统。

9.2.1 内核

现代操作系统中大都采用了进程的概念，为了解决系统的并发性、共享性和随机性，使进程能协调地工作，单靠计算机硬件提供的功能是十分不够的。例如，进程调度工作目前就不能用硬件来实现；而进程自己调度自己也是困难的。所以，系统必须有一个部分能对硬件处理器及有关资源进行首次改造，以便给进程的执行提供良好的环境。这个部分就是操作系统的内核。

由于操作系统设计的目标和环境不同，内核的大小和功能有很大差别。有些设计希望把内核的常驻部分限制在较少的主存空间内，有些则希望内核具有较多的功能。因而，操作系统的一个基本设计问题内核的功能。

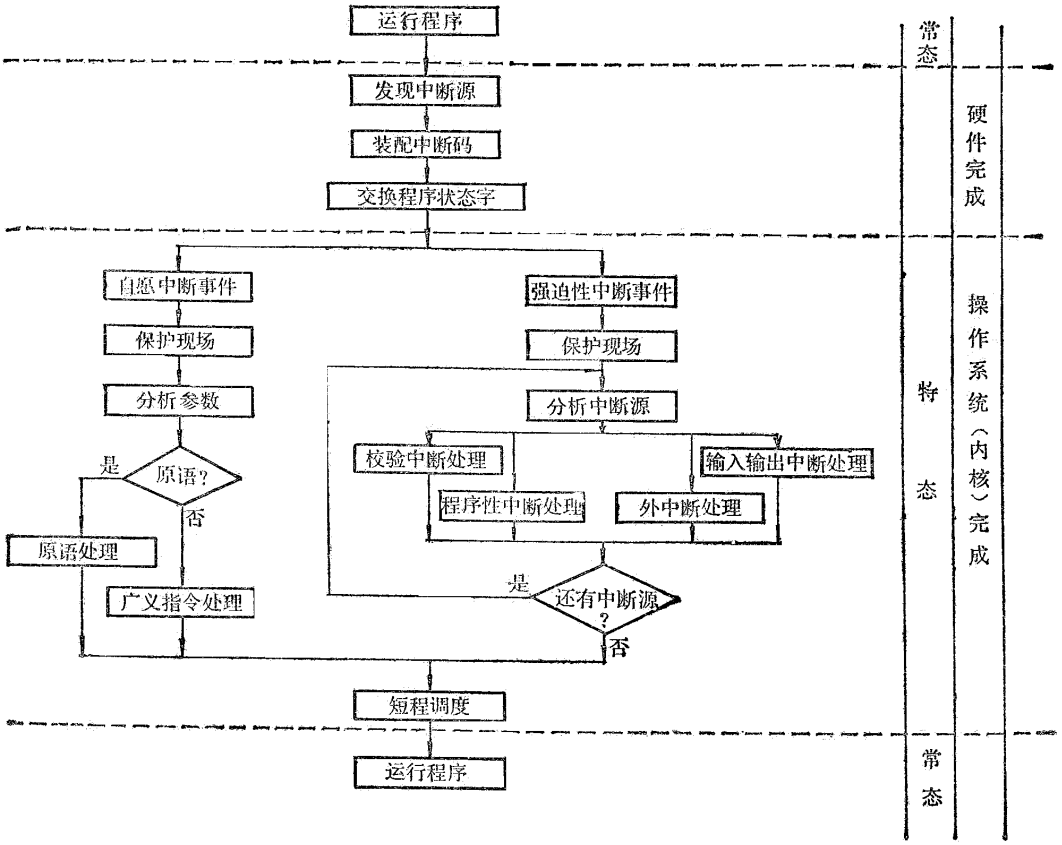


图 9-1　内核的处理流程

一般而言，内核提供以下三方面功能：

(1) 中断处理　当中断事件产生时，先由内核的中断处理例行程序接受并进行原则处理。它分析中断事件的类型和性质，进行必要的状态修改，然后交给进程或模块去处理。例如，产生外围设备结束中断事件时，内核首先分析是否正常结束，如果是正常结束，那么，就应释放等待该外围传输的进程。又如当操作员请求从控制台输入命令时，内核将把这一任务转交给命令管理进程或模块去处理。

(2) 短程调度　主要职能是分配处理器。当发生了一个事件之后，可能一个进程要让出处理器，而另一个进程又要获得处理器。短程调度按照一定策略管理处理器的转让，以及完成保护和恢复现场的工作。

(3) 原语管理　原语是内核中一个完整的过程。为了协调进程并发工作和共享资源，同步原语是必不可少的，此外还有其它原语，如启动外围设备工作的启动原语，若启动不成功则请求启动者应等待，显然，这个启动过程应该是完整的，否则在成为等待状态时，可能外围设备已经空闲。

内核的执行有以下属性：

(1) 内核是由中断驱动的。只有当发生中断事件后由硬件交换 PSW 才引出操作系统的内核进程中断处理，且在处理完中断事件后内核自行退出。如图所示。

(2) 内核的执行是连续的，在内核运行期间不能插入内核以外的程序执行，因而，能保证在一个连续的时间间隔内完成任务。

(3) 内核在屏蔽中断状态下执行，在处理某个中断时，为避免中断的嵌套可能引起的错误，必须屏蔽该级中断。有时为处理简单，把其它一些中断也暂时屏蔽了。

(4) 内核可以使用特权指令，现代计算机都提供常态和特态等多种机器工作状态，有一类指令称特权指令，只允许在特态下使用，例如，输入输出、状态修改、存储管理等。规定这类指令只允许内核使用，可防止系统出现混乱。

内核是操作系统对裸机的第一次改造，内核和裸机组成了一台虚拟机，进程或模块就在这台虚拟机上运行，它比裸机的功能更大，具有以下特性：

(1) 虚拟机没有中断，因而进程或模块的设计者不再需要有硬件中断的概念，进程或模块执行中无需处理中断。

(2) 虚拟机为每个进程提供了一台虚拟处理器，每个进程就好象在各自的私有处理器上顺序地进行，实现了多个进程的并发执行。

(3) 虚拟机为进程或模块提供了功能较强的指令系统，即它们能够使用机器非特权指令，广义指令和原语所组成的新的指令系统。

分　类		工作状态	提供功能	可用指令
操作系统	内核	管态	原语	特权指令 非特权指令
	系统进程	目态	广义指令	原语 非特权指令
用户进程	编译系统	目态	语句	广义指令 非特权指令
	用户作业	目态		语句

操作系统扩充了计算机功能

为了保证系统的有效性和灵活性，设计内核应遵循少而精的原则。如果内核功能过强，则一方面在修改系统时，可能牵动内核，另一方面它占用的主存量和执行时间都会增大，且屏蔽中断的时间过长也会影响系统效率。因而，设计内核时应注意：中断处理要简单；调度算法要有效；原语要灵活有力，数量适当。这样就可以做到修改系统时，尽少改动内核，执行时中断屏蔽时间短。

近年来又提出了微内核（Micro Kernel）的概念，有关这部分内容，稍后作介绍。

9.2.2 进程

进程是并发程序设计的一个工具，并发程序设计支撑了多道程序设计，由于进程能确切、动态地刻划计算机系统内部的并发性，更好地解决系统资源的共享性，所以，在操作系统的发展史上，进程概念被较早地引入了系统。它在操作系统理论研究和设计实现上均发挥了重要作用。采用进程概念使得操作系统结构变得清晰，主要表现在：一是一个进程到另一个进程的控制转移由进程调度机构统一管理，不能杂乱无章，随意进行；二是进程之间的信号发送、消息传递和同步互斥由通信及同步机制完成，从而进程无法有意或无意破坏它进程的数据。因此，每个进程相对独立，相互隔离，提高了系统的安全性和可靠性；三是进程结构较好刻画了系统的并发性，动态地描述出系统的执行过程，因而，具有进程结构的操作系统，结构清晰、整齐划一，可维护性好。

9.2.3 线程

早期，进程是操作系统中资源分配以及系统调度的独立单位。由于每个进程拥有自己独立的存储空间和运行环境，进程和进程之间并发性粒度较粗，通信和切换的开销相当大。要更好地发挥硬件提供的能力（如多 CPU），要实现复杂的各种并发应用，单靠进程是无能为力的，于是，近年来开始流行多线程（结构）进程（Multithreaded process），亦叫多线程。

在一个多线程环境中，进程是系统进行保护和资源分配的单位，而线程则是进程中一条执行路径，每个进程中允许有多个并行执行的路径，而线程才是系统进行调度的单位。

在一个进程中包含有多个可并发执行的控制流，而不是把多个控制流一一分散在多个进程中，这是并发多线程程序设计与并发多进程程序设计的主要不同之处。

9.2.4 管程

管程是管理共享资源的机制，对管程的调用表示对共享资源的请求与释放。管程可以被多个进程或管程嵌套调用，但它们只能互斥地访问管程。管程应包含条件变量，当条件不满足时，可以通过对条件变量做操作使调用进程延迟，直到另一个进程调用管程过程并执行一个释放操作为止。

9.2.5 类程

类程是管理私有资源的，对类程的调用表示对私有资源的操作。它仅能被进程及起源于同一进程的其它类程或管程嵌套调用链所调用。其本身也可以调用其它类程或管程。类程可以看作子程序概念的扩充，但一个类程可以包含多个过程，不像子程序仅仅一个。

采用进程、管程、类程的操作系统中，进程在执行过程中若请求使用共享资源，则可以调用管程；若要控制私有资源操作，可以调用类程，这样便于使用高级程序设计语言来书写操作系统。1975 年，汉森使用这一方法就成功地在 PDP 11/45 机上实现了：单用户操作系统 Solo、处理小作业的作业流系统和过程控制实时调度系统等三个层次管程结构的操作系统。

9.3 操作系统结构概述

操作系统是一种大型、复杂的并发系统，为了研制操作系统，首先必须研究它的结构，在操作系统的发展过程中，产生了多种系统结构。下面将讨论四种都实际尝试过的组织结构：整体式系统、层次式系统、虚拟机系统和客户—服务器系统。

9.4 整体式结构

整体式系统是最常用的组织方式，但常被人们形容为“一锅粥”，其结构其实就是“无结构”。整个操作系统是一堆过程的集合，每个过程都可以调用任意其他过程。使用这种技术时，系统中的每一过程都有一个定义完好的接口，即它的入口参数和返回值，而且相互间的调用不受约束。

在整体式系统中，为了构造最终的目标操作系统程序，开发人员首先将一些独立的过程进行编译，然后用链接程序将其链接在一起成为一个单独的目标程序。从信息隐藏的观点看，它没有任何程序的隐藏——每个过程都对其他过程可见。（与此相对的是将系统分成若干个模块，信息被隐藏在这些模块内部，在外部只允许从预先定好的调用点访问这些模块）

但即使在整体系统中，也存在一些程度很低的结构化。操作系统提供的服务（系统调用）的调用过程是这样的：先将参数放入预先确定的寄存器或堆栈中，然后执行一条特殊的陷入指令，即访管指令或核心调用（kernel call）指令。

这条指令将机器由用户态切换到核心态，并将控制转到操作系统。该过程如图 12-2 所示。多数 CPU 有两种状态：核心态：供操作系统使用，该状态下可以执行机器的所有指令；用户态：该状态下 I/O 操作和某些其他操作不能执行。

操作系统随后检查该调用的参数以确定应执行哪条系统调用，这如图 9-2 中②所示。然后，操作系统查一张系统调用表，其中记录了每条系统调用的执行过程，这一步操作如图中③所示，它确定了将调用的服务过程。当系统调用结束后，控制又返回给用户程序（第④步），于是继续执行系统调用后面的语句。

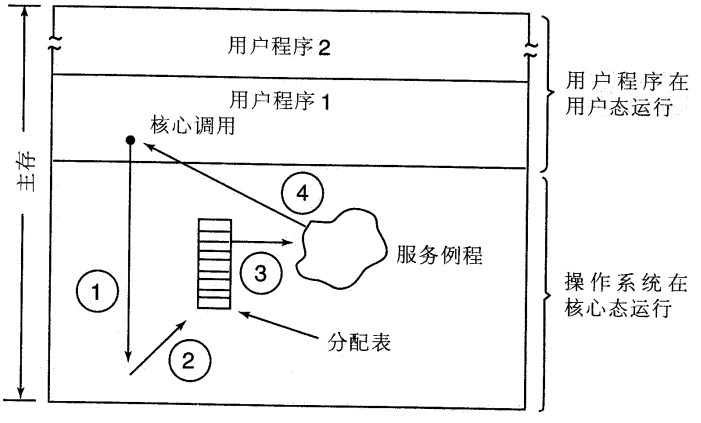


图 9-2　系统调用的操作过程：①用户程序陷入核心，②操作系统确定所请求的服务编号，③操作系统调用服务过程，④控制返回到用户程序

这种组织方式提出了操作系统的一种基本结构：

(1) 一个用来调用被请求服务例程的主程序。

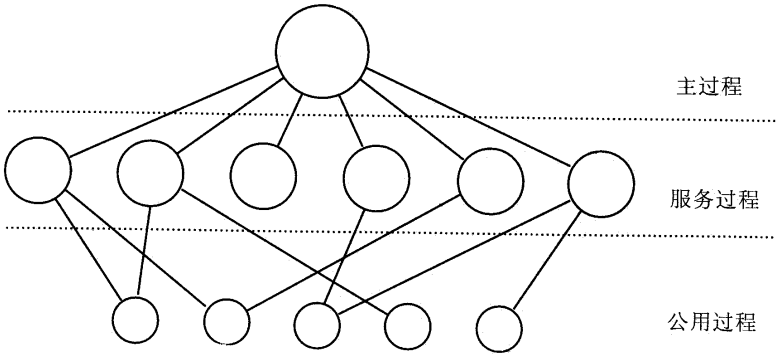
(2) 一套执行系统调用的服务例程。

(3) 一套支持服务例程的实用过程。

在这种模型中，每一条系统调用都由一个服务例程完成；一组实用过程用来完成若干服务例程都需要用到的功能，如从用户程序获取数据等，这种将各种过程分为三层的模型如图 9-3 所示。

图 9-3　整体式操作系统的简单结构模型

Unix 的主要特点是短小精悍，简易有效，易扩充，易移植，为什么它会得到如此的赞誉？主要是因



为采用了模块化结构，Unix 操作系统所具有的特点，恰是模块接口结构的优点。

首先，由于模块接口结构是把整个操作系统划分为若干具有一定独立功能的模块，而模块又可细分为子模块。规定好各模块、子模块之间的接口关系后，就可由多人分头去完成它们的程序设计工作，随后逐一连接起来，形成一个完整的系统，这就可缩短操作系统的开发周期。

第二，由于各模块之间可以直接通过名字调用，能随意地利用别的模块所提供的功能，所以整个系统结构紧密，效率高；

第三，由于各模块都具有一定的独立功能，所以这种结构如同搭积木一般，便于根据不同的环境，选择不同的模块，生成满足不同要求的操作系统。这个过程通常称作系统生成；

第四，正因为这种结构如同积木一样，因此也便于扩充。当需要增加系统功能时，只要将新模块连入，即可达到扩充的目的。

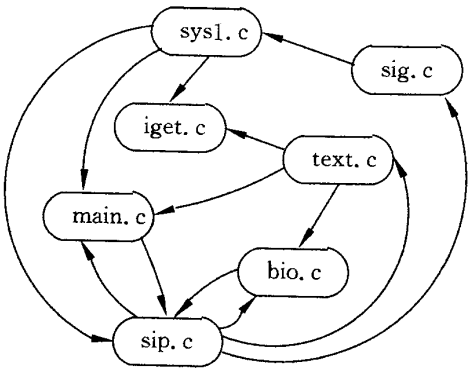


图 9-4　Unix 部分文件间的复杂调用关系

然而，模块接口结构也存在很多的问题，很多优点的背后也恰好隐藏了缺点。

第一，没有一个清晰的结构。在模拟接口结构中，系统短小精悍，各模块能随意调用，因此在它们之间形成了非常复杂的调用关系，互相依赖，毫无次序。举例来讲，在 Unix 操作系统的核心中，每个文件内的程序可以任意调用其它文件内的程序，下图给出了其中七个文件之间的调用关系。可以看出，里面包含了很多的循环调用。如文件 sys1.c 要调用文件 main.c 中的程序；文件 main.c 要调用文件 sip.c 中的程序；sip.c 要调用文件 sig.c 中的程序；最后，sig.c 又要调用文件 sys1.c 中的程序。这样一种复杂的调用，使得它们之间关系扑朔迷离，难以理解。

第二，系统正确性难以保证。由于模块接口结构之间关系紧密，因此某一模块的错误会造成很大的影响面，错误在一个地方表现出来，但根源可能在别处，难以查找和消除。

第三，系统维护不便利，由于模块间的关系密切，因此某一模块功能的变更或扩充，都将可能引起接口的改变，而接口的改变又会影响到与此有关的各模块的变动，这种牵一发而动全身的态势，会给系统的扩充，移植等工作带来麻烦。

9.5 层次式结构

9.5.1 层次式结构概述

为了能让操作系统的结构更加清晰，使其具有较高的可靠性，较强的适应性，易于扩充和移植，在模拟接口结构和进程结构的基础上又产生了层次结构的操作系统。所谓层次结构，即是把操作系统划分为内核和若干模块（或进程）组成，这些模块（或进程）排列成若干层，各层之间只能是单向依赖关系，不构成循环，如图 12-5 所示。

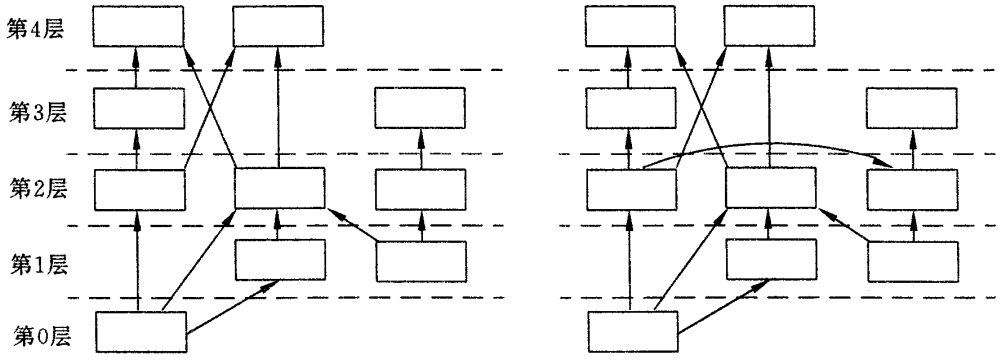


图 9-5　层次结构

层次结构可以有全序和半序之分。如果各层之间是单向依赖的，并且每层中的诸模块（或进程）之间也保持独立，没有联系，则这种层次结构被称为是全序的，如图 9-5（a）所示。如果各层之间是单向依赖的，但在某些层内允许有相互调用或通信的关系，则这种层次结构为半序的，如图 9-6（b）的第二层所示。

层次结构是如此构造起来的，从裸机 A_0 开始，在它上面添加一层软件，使机器的功能得以扩充，形成了一台功能比原来机器要强的虚拟机 A_1。又从 A_1 出发，在它上面添加一层新的软件，把 A_1 改造成功能更强的虚拟机 A_2。就这样“添加——扩充——再添加”，由底向上地增设软件层，每一层都在原来虚拟机的基础上扩充了原有的功能，于是最后实现一台具有所需操作系统各项功能的虚拟机。

9.5.2 分层的原则

在用层次结构构造操作系统时，目前还没有一个明确固定的分层方法，只能给出若干原则，供划分层中模块（或进程）时参考。

(1) 应该把与机器硬件有关的程序模块放在最底层，以便起到把其它层与硬件隔离开的作用。在操作系统中，中断处理、设备启动、时钟等反映了机器的特征，因此都应该放在离硬件尽可能近的这层之中，这样安排也有利于操作系统的移植，因为只需把这层的内容按新机器硬件的特性加以改变后，其它层内容都可以基本不动。

(2) 对于用户来讲，可能需要不同的操作方式，譬如可以选取批处理方式，联机作业控制方式，或实时控制方式。为了能使一个操作系统从一种操作方式改变或扩充到另一种操作方式，在分层时就应把反映系统外特性的软件放在最外层，这样改变或扩充时，只涉及到对外层的修改。

(3) 应该尽量按照实现操作系统命令时模块间的调用次序或按进程间单向发送信息的顺序来分层。这样，最上层接受来自用户的操作系统命令，随之根据功能需要逐层往下调用（或传递消息），自然而有序。譬如，文件管理时要调用到设备管理，因此文件管理诸模块（或进程）应该放在设备管理诸模块（或进程）的外层；作业调度程序控制用户程序执行时，要调用文件管理中的功能，因此作业调度模块（或进程）应该放在文件管理模块（或进程）的外层。

(4) 为进程的正常运行创造环境和提供条件的内核（CPU 调度、进程控制和通信机构等）应该尽可能放在底层，以保证它们运行时的高效率。

下图给出了一个完整操作系统的简略分层结构图。层次结构 OS 按此模型构造的第一个操作系统是 E. W. Dijkstra 和他的学生在荷兰的 Eindhoven 技术学院开发的 THE 系统（1968 年）。THE 系统是为荷兰制造的 Electrologica X8 计算机（内存为 32K 个 27 比特的字）配备的一个简单的批处理系统。

该系统分为六层，如图 9-6 所示。第零层进行处理器分配，当发生中断或时钟到达期限时由该层软件进行进程切换。在第零层之上有若干个顺序进程运行，编写这些进程时就不用再考虑我个进程在单一

处理器上运行的细节。换句话说，第零层提供了 CPU 基本的多道程序功能。

层　次	功　　能
5	操作员
4	用户程序
3	输入/输出管理
2	操作员—进程通信
1	内存和磁鼓管理
0	处理器分配和多道程序

图 9-6　THE 操作系统的结构

第一层进行内存管理，它为进程分配内存空间，当内存用完时则会在用作对换的 512K 字的磁鼓上分配空间。在第一层之上，进程不用再考虑它是在内存还是在磁鼓上，因为第一层软件保证在需要访问某一页面时，它必定在内存中。

第二层软件处理进程与操作员控制台之间的通信。在第二层之上，则可认为每个进程都有它自己的操作员控制台。第三层软件管理 I/O 设备和相关的信息流缓冲。在第三层之上，每个进程都与适当抽象了的设备打交道而不必考虑物理设备的细节。第四层是用户程序层，用户程序在此不必考虑进程、内容、控制台和 I/O 设备等环节。系统操作员进程位于第五层。

MULTICS 对层次化概念进行了更进一步的通用化，它不采用层而是由许多同心的环构成，内层的环比外层的环有更高的特权级。当外层环的过程调用内层环的过程时，它必须执行一条类似系统调用的 TRAP 指令，TRAP 指令执行前要进行严格的参数合法性检查。尽管在 MULTICS 中操作是各个用户进程地址空间的一部分，硬件仍然能够对单个进程（实际上是内存中的一个段）的读、写和执行权限进行保护。

实际上 THE 分层方案只是在设计上提供了一些方便，因为系统的各个部分最终仍然被链接成一个完整的单个目标程序，而在 MULTICS 中，上述环形方案在运行中是实际存在的且由硬件实现。环形方案的一个优点是它很容易被扩展，以构造用户子系统。例如在一个系统中，教授可以写一个程序来检查学生编写的程序并打分，将教授的程序放在第 n 个环中运行，而将学生的程序放在第 n+1 个环中运行，则学生无法篡改教授给出的成绩。

9.5.3 对层次结构的分析

层次结构的最大优点是把整体问题局部化。由于把复杂的操作系统依照一定的原则分解成若干单向依赖的层次，因此整个系统中的接口量相比前两种结构要少且简单，整个系统的正确必可通过各层的正确性来保证，从而使系统的正确性大大提高。

层次结构展现在人们面前显得井井有序，清晰明朗。这种结构有利于系统的维护和扩充，也便于移植。然而，层次结构是分层单向依赖的，因此系统花费在通信上的开销较大，且效率不高。

9.6 虚拟机系统

OS/360 的最早版本是纯粹的批处理系统，然而许多 360 的用户希望使用分时系统，于是 IBM 公司和另外的一些研究小组决定开发一个分时系统。随后 IBM 提供了一套分时系统 TSS/360，但它非常庞大，运行缓慢，几乎没有什么人用。该系统在花费了约五千万美元的研制费用后最终被弃之不用（Graham，1970）。但 IBM 设在麻省剑桥的一个研究中小开发了一个完全不同的系统，最终被 IBM 用作为产品。该系统目前仍然在 IBM 的大型主机上广泛使用。

该系统最初命名为 CP/CMS，后来改为 VM/370（Seawright and MacKinnon， 1979）。它基于如下的思想：一个分时系统应该提供以下特性：(1) 多道程序，(2) 一个具有比裸机更方便、界面扩展的计算机。VM/370 的主旨在于将此二者彻底地隔离开来。

该系统的核心称作虚拟机监控程序，它在裸机上运行并具备多道程序功能。它向上层提供了若干台虚拟机，如图 9-7 所示。与其他操作系统不同的是：这些虚拟机不是那种具有文件等良好特征的扩展计算机，而仅仅是裸机硬件的精确复制。它包含有：核心态/用户态，I/O 功能，中断，以及真实硬件具有的全部内容。

因为每台虚拟机都与裸机完全一样，所以每台虚拟机可以运行裸机能够运行的任何操作系统。不同的虚拟机可以运行不同的操作系统而且往往如此。某些虚拟机运行 OS/360 的后续版本作批处理或事务处理，而同时另一些运行一个单用户交互系统供分时用户使用，该系统称作 CMS（Conversational Monitor System，会话监控系统）。

当 CMS 上的程序执行一条系统调用时，该系统调用陷入其自己的虚拟机的操作系统，而不是

VM/370，这就像在真正的计算机上一样。CMS 然后发出正常的硬件 I/O 指令来执行该系统调用。这些 I/O 指令被 VM/370 捕获，随后 VM/370 执行这些指令，作为对真实硬件模拟的一部分。通过将多道程序功能和提供虚拟机分开，它们各自都更简单、更灵活和易于维护。

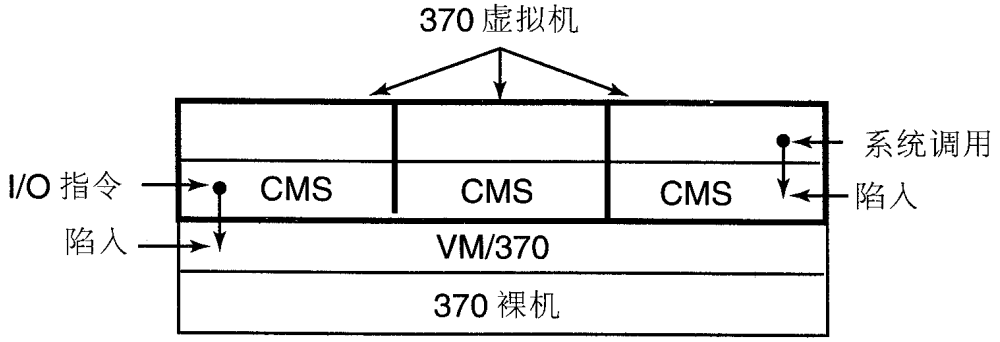


图 9-7　带 CMS 的 VM/370 结构

现在虚拟机的思想被广泛采用：例如在奔腾 CPU（或其他 Intel 的 32 位 CPU）上运行老的 MS-DOS 程序。在设计奔腾芯片的硬件和软件时，Intel 和 Microsoft 都意识到要使软件能够在新硬件上运行，于是 Intel 在奔腾芯片上提供了一个虚拟 8086 模式。在此模式下，奔腾机就象一台 8086 计算机一样，包括 1M 字节内的 16 位寻址方式。

虚拟 8086 模式被 MS-Windows、OS/2 及其他操作系统用于运行 MS-DOS 程序。程序在虚拟 8086 模式下启动，执行一般的指令时它们在裸机上运行。但是，当一个程序试图陷入系统来执行一条系统调用时，或者试图执行受保护的 I/O 操作时，将发生一条虚拟机监控程序的陷入。

此时有两种设计方法：第一种，MS-DOS 本身被装入虚拟 8086 模式的地址空间，于是虚拟机仅仅将该陷入传回给 MS-DOS。这种处理与在真正的 8086 上运行是一样的，当 MS-DOS 后来试图自行执行 I/O 操作时，该操作被捕获而由虚拟机监控程序完成。

另一种方法是虚拟机监控程序仅仅捕获第一条陷入并自己执行 I/O 操作，因为它知道所有的 MS-DOS 系统调用，并且由此知道每条陷入企图执行什么操作。这种方法不如第一种方法纯，因为它仅仅正确地模拟了 MS-DOS，而不包括其他操作系统，相比之下，第一种方法可以正确地模拟其他操作系统。但另一方面，这种方法很快，因为它不再需要启动 MS-DOS 来执行 I/O 操作。在虚拟 8086 模式中真正地运行 MS-DOS 另一个缺点是 MS-DOS 频繁地对中断屏蔽进行操作，而模拟这种操作是很费时的。

需要注意的是上述方法都不同于 VM/370，因为它们模拟的并不是完整的奔腾硬件而是一个 8086。在 VM/370 系统中，可以在虚拟机上运行 VM/370 本身，而在奔腾系统中，不可能在虚拟 8086 上运行 Windows，因为 Windows 不能在 8086 芯片上运行。其最低版本也需要在 80286 上运行，然而奔腾芯片不提供对 80286 的模拟。

在 VM/370 中，每个用户进程获得真实计算机的一个精确拷贝。在奔腾芯片的虚拟 8086 模式中，每个用户进程获得的是另一套硬件（8086）的精确拷贝。进一步而言，M. I. T 的研究人员构造了一个系统，其中每个用户都获得真实计算机的一个拷贝，只是占用的资源是全部资源的一部分（Engler et al.， 1995）。于是一台虚拟机可能占用磁盘的第 0 块到 1023 块，另一台可能占用 1024 到 2047 块等等。

在核心态下运行的最底层软件是一个称作“外核”（exokernel）的程序，其任务是为虚拟机分配资源并确保资源的使用不会发生冲突。每台用户层的虚拟机以运行其自己的操作系统，就像 VM/370 和奔腾的虚拟 8086 一样。不同在于他们各自只能使用分配给它的那部分资源。

“外核”方案的优点在于它省去了一个映射层。在别的设计方案中，每台虚拟机认为它有自己独立的磁盘，编号从 0 到最大，于是虚拟机监控程序必须维护一张表来完成磁盘地址的映射（也包括其他资源）。有了“外核”之后，这种映射就不再需要了。外核只需记录每台虚拟机被分配的资源。这种方法的另一个好处是以较少的开销将多道程序（在外核中）与用户操作系统代码（在用户空间）分离开来，因为外核需做的工作是使各虚拟机互不干扰。

9.7 客户/服务器结构（微内核结构）

9.7.1 微内核(Microkernel)技术

近来微内核的概念得到了广泛的关注。尽管不同的操作系统开发者对微内核有着不同的解释，微内核可以看作是一个小型的操作系统核心，它为操作系统的扩展提供了基于模块扩充的基础。

微内核的流行是由于 Mach 操作系统成功地应用了该技术。这种技术提供了高度的灵活性和模块化。Windows NT 是另一个成功地使用微内核技术的操作系统，除了模块化之外它还取得了很好的可移植性，NT 的微内核包括一组紧凑的子系统，基于此在各种平台实现 NT 操作系统就变得十分容易。目前大量产品均自称基于微内核开发，可以预见在不久的将来这一技术将在微机、工作站和服务器操作系统中得到广泛使用。

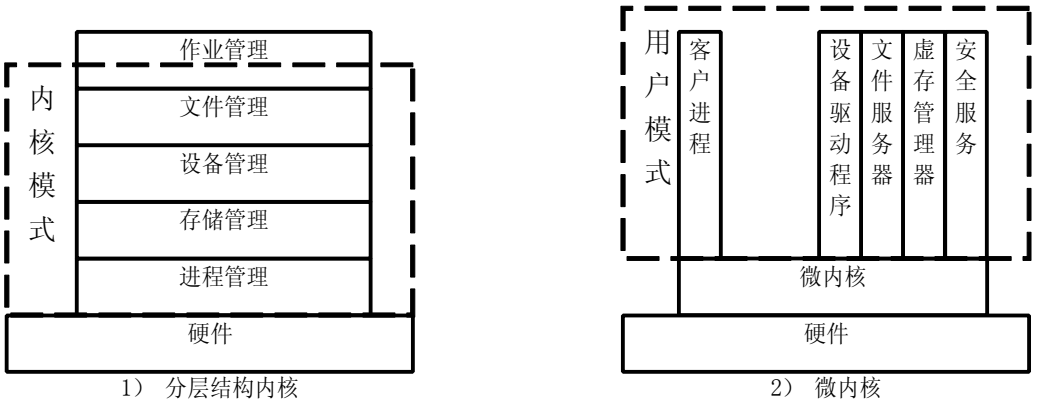
9.7.2 微内核结构概述

早期操作系统很少考虑结构，实现时采用过程调用的方式，系统缺乏结构性，过于庞大，如 OS/360 由 5000 个程序员做了五年，有 100 万行源程序；Multics 则包括 2000 万行。

以后，模块化程序设计技术被引进来解决大型软件的开发，于是出现了分层操作系统结构，操作系统被划分为进程管理、存储管理、设备管理、文件管理、…、等等层次，它们一般都处于操作系统内核，很少分布在用户模式下。但是由于层与层之间是按功能划分的，互相之间关系密切，因此操作系统的扩充和裁减变得十分困难；并且由于许多交互卡在相邻层之间进行，还影响到系统的安全性。

微内核基本思想是：内核中仅存放那些最基本的核心操作系统功能。其它服务和应用则建立在微内核之外，在用户模式下运行。尽管那些功能应该放在内核内实现，那些服务应该放在内核外实现，在不同的操作系统设计中未必一样，但事实上过去在操作系统内和中的许多服务，现在已经成为了与内核交互或相互之间交互的外部子系统，这些服务主要包括：设备驱动程序、文件系统、虚存管理器、窗口系统和安全服务。

如图 9-8 所示，分层结构操作系统的内核很大，互相之间调用关系复杂。微内核结构则把大量的操作系统功能放到内核外实现，这些外部的操作系统构件是作为服务过程来实现的，它们之间的信息相互均借助微内核提供的消息传送机制实现。这样，微内核起消息交换功能，它验证消息，在构件构件之间传送消息，并授权存取硬件。例如，当一个应用程序要打开一个文件，它就传送一个消息给文件系统服务器；当它希望建立一个进程或线程，就送一个消息给进程服务器；每个服务器都可以传送消息给另外的服务器，或者调用在内核中的原语功能。这是一种可以运行在单计算机中的 C/S 结构。



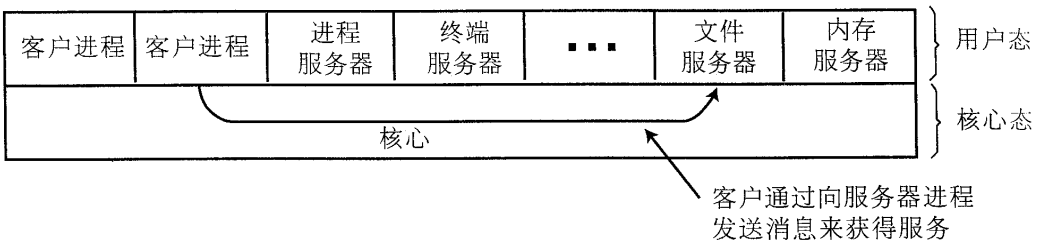
1） 分层结构内核　　2） 微内核

图 9-8　分层结构内核和微内核结构对比

举例来说，为了获取某项服务，比如读文件中的一块，用户进程（现称客户进程，client process）将此请求发送给文件服务器进程（server process），服务器进程随后完成此操作并将回答信息送回。

该模型示于图 9-9 中，核心的全部工作是处理客户与服务器间的通信。操作系统被分割成许多部分，每一部分只处理一方面的功能，如文件服务、进程服务、终端服务或存储器服务。这样每一部分变得更小、更易于管理。而且，由于所有服务器以用户进程的形式运行，而不是运行在核心态，所以它们不直接访问硬件。这样处理的结果是：假如在文件服务器中发生错误，文件服务器可能崩溃，但不会导致整个系统的崩溃。

图 9-9　客户/服务器模型



9.7.3 微内核结构的优点

微内核结构的优点主要有：

一致性接口：微内核结构对进程的请求提供了一致性接口（uniform interface），进程不必区别内核级服务或用户级服务，因为所有这些服务均借助消息传送机制提供。

可扩充性：任何操作系统都要增加目前设计中没有的特性，如开发的新硬件设备和新软件技术。微内核结构具有可扩充性，它允许增加新服务：以及在相同功能范围中提供多种可选服务，例如，对磁盘上的多种文件组织方法，每一种可以作为一个用户级进程来实现，而并不是在内核中实现多种文件服务。因而，用户可以从多种文件服务中选出一种最适合其需要的服务。每次修改时，新的或修改过的服务的影响被限制在系统的子集内。修改并不需要建立一个新的内核。

适用性：与可扩充性相关的是适用性，采用微内核技术时，不仅可以把新特性加入操作系统，而且可以把已有的特性能被抽象成一个较小的、更有效的实现。微内核操作系统并不是一个小的系统，事实上这种结构允许它扩充广泛的特性。并不是每一种特性都是需要的，如高安全性保障或分布式计算。如果实质的功能可以被任选，基本产品将能适合于广泛的用户。

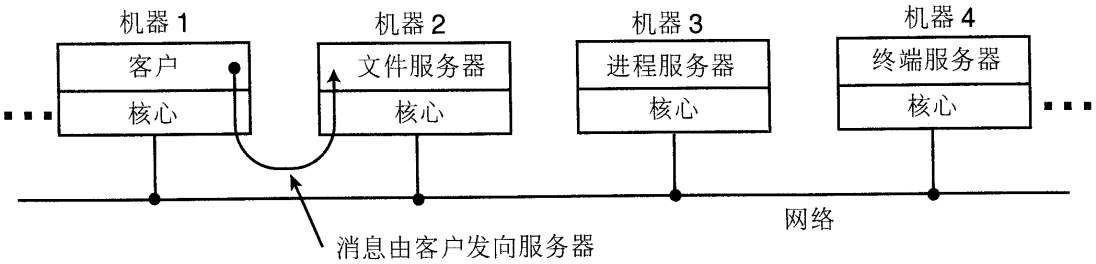
可移植性：随着各种各样的硬件平台的出现，可移植性称为操作系统的一个有吸引力的特性。在微内核结构中，所有与特定 CPU 有关的代码均在内核中，因而把系统移植到一个新 CPU 上所作修改较小。

可靠性：对大型软件产品，较困难的是确保它的可靠性，虽然模块化设计对可靠性有益，但从微内核结构中可以得到更多的好处。销的微内核化代码容易进行测试，小的 API 接口的使用提高了给内核之外的操作系统服务生成高质量代码的机会。

支持分布式系统：微内核提供了对分布式系统的支撑，包括通过分布操作提供的 cluster 控制。当消息从一个客户机发送给服务器进程时，消息必须包含一个请求服务的标识，如果配置了一个分布式系统（即一个 cluster），所有进程和服务均有唯一标识，并且在微内核级存在一个单一的系统映象。进程可以传送一个消息，而不必知道目标服务进程驻留在那台机器上。

图 9-10　在一个分布式系统中的客户/服务器模型

支持面向对象的操作系统（OOOS）：微内核结构能在一个 OOOS 环境中工作得很好，OO 方法能为



设计微内核以及模块化的扩充操作系统提供指导。许多微内核设计时采用了 OO 技术，其中，有一种方法是使用构件。另外一些系统，如 NT 操作系统，并不完全依赖于面向对象的技术，但在微内核设计时结合 OO 原理。

9.7.4 微内核的性能

性能问题是微内核一个潜在缺点，发送消息和建立消息需要化费一定的时间代价，同直接调用单个服务相比接收消息和生成回答都要多化费时间。

性能与微内核的大小和功能直接有关。一种可行的方法是扩充微内核的功能，减少用户 _——内核模式切换的次数和进程地址空间切换的次数，但这直接提高了微内核的设计代价，损失了微内核在小型接口定义和适应性方面的优点。

另一解决方法是把微内核做得更小，通过合理的设计，一个非常小的微内核提高性能、灵活性及可靠性。典型的第一代微内核结构大小约由 300KB 的代码和 140 个系统调用接口。一个小的二代微内核的例子是 L4。它有 12KB 代码和 7 个系统调用。对这些系统的试验表明它们的工作性能要优于采用传统分层结构的 Unix。

9.7.5 微内核的设计

目前存在着多种不同微内核，它们之间规模和功能差别很大，并且不存在一个必须遵守的规则来规定微内核应提供什么功能或基于什么结构实现。因此在本节中我们只讨论一个最小化的微内核所应提供的功能与服务。

微内核必须包括那些直接依赖于硬件的功能，以及支撑操作系统用户模式应用程序和服务所需的功能，这些功能可概括为：存储管理、进程通信（IPC）、I/O 和中断管理。

9.7.5.1 基本的存储管理

为了实现进程级的保护，微内核必须控制地址空间的硬件设施。内核负有把每个虚页面映射到物理页框的责任，而大量的存储管理，包括进程地址空间之间的相互保护、页面淘汰算法、其它页逻辑，都能在内核之外来实现。例如，由一个内核之外的虚存模块来决定页的调入和替换，内核负责映射这些页到主存的具体物理地址空间。

页面调度和虚存管理可以在内核外执行，图 9-11 给出了建立在内核外的页面管理程序，例如：当一个应用中的一个线程进引用了一个不在主存中的页面时，缺页中断发生，执行陷入到内核；内核传送一个消息给页面管理程序，指明要引用的页面；页面管理程序决定装入页面，并预先分配一个页框；页面管理程序和内核必须交互以映射分页管理的逻辑页面操作到物理存储空间；当页面调入后，页面管理程序发送一个恢复消息给应用程序。

图 9-11　建立在内核外的页面管理程序

这种技术能不必调用内核（功能），而让一个非内核进程来映射文件和数据库到用户地址空间，并且与应用相关的内存共享策略也能在内核之外来实现。

这里介绍三个基本的微内核操作，以支持内核外部的页面管理和虚存管理：

转让（Grant）：一个地址空间（进程）的拥有者能够转让它的一些页面给其它进程使用。执行了 这个操作之后，内核将从转让进程的地址空间中移去这些页面，并分配给被转让的的进程。

映射(Map)：一个进程可以映射它的任何一个页面到另一进程的地址空间中。执行了这个操作之后，在两个进程间建立了共享存储区，两个进程均可以存取这些页面。

刷新（Flush）：一个进程能再次申请已经被转让或映射给其它进程的任何页面。

一开始，内核定义所有的物理内存作为让一个基本系统进程控制的单一地址空间，当建立新进程时，原有的总地址空间中的页面能被转让或映射给新进程，这种模式能支撑同时执行的多重虚拟存储管理。

9.7.5.2 进程间通信

在微内核操作系统中，进程通信和线程通信的基本形式是消息。一个消息包括了头标（header）和消息体，头标指明了发送和接受消息的进程，消息体包含直接数据，即一个数据块指针和进程的有关控制信息。进程间通信基于进程之间相关联的端口（Ports），一个端口是一个特定进程的消息队列，与端口相关的是一张能力表，记录了可以与这个进程通信的进程，端口的标识和能力由内核维护。一个进程通过发送一消息给内核以指明新端口的性能，以实现对自己的访问。

9.7.5.3 I/O和中断管理

对于微内核结构，如同消息一样来处理硬件中断和包含 I/O 端口到地址空间中是可能的。

微内核能够发现中断，但不能处理它，微内核将生成一个消息传送给用户层中处理有关中断的进程。因而，当中断出现时，一个特别用户进程被指派到中断（信号），内核维护这一映射。转换中断为消息必须由内核做，但内核民间不包含与特定设备的中断处理。

可以把硬件看作为一组具有唯一线程标识的线程，并且可以发消息给地址空间中的有关软件线程，接收线程决定是否消息来自于一个中断并且确定中断的类型，这类用户级代码的通用结构如下：

```
driver thread;
do
      wait For (mhg，  sender);
      if sender = my_hardware_interrupt
      then    read/writer I/O ports; reset hardware interrut
      else  …
      endif
enddo
```

微内核是一个小型的操作系统核心，提供了模块扩充的基础，由于 Mac 操作系统的使用使微内核结构开始流行了。这种方法提供系统高度的灵活性、模块性和可移植性，已经在微机、工作站和服务器操

作系统中得到广泛使用。

早期，采用模块调用结构，系统庞大，接口复杂，缺乏良结构，如 OS/360 由 5000 个程序员干了五年，写出了 100 万行源码；Multics 的源码更是扩展到 2000 万行。层次式结构的操作系统，分层困难，通信开销大；VM/370 将传统操作系统的大部分代码（实现扩展的计算机）分离出来放在更高的层次上，即 CMS，由此使系统得以简化。但 VM/370 本身仍然非常复杂，因为模拟许多虚拟的 370 硬件不是一件简单的事情（尤其是还想作得高效时）。

现代操作系统的一个趋势是将这种把代码移到更高层次的思想进一步发展，从操作系统中去掉尽可能多的东西，而只留一个最小的核心。通常的方法是将大多数操作系统功能由用户进程来实现。过去已成为操作系统传统的许多服务，现在成了与微内核交互的外部子系统。微内核中包括的功能主要有：设备驱动、中断管理、安全服务和提供原语，用于支撑文件系统、虚存管理等功能。

微内核结构用水平型代替传统的垂直型结构操作系统。

9.8 实例研究：Windows2000 的系统结构

9.8.1 Windows2000 系统结构的设计目标

Windows2000 系统结构的设计需求包括：

- 提供一个真正 32 位抢占式可重入的虚拟内存操作系统。
- 能够在多种硬件体系结构上运行。
- 能够支持 SMP 结构和 CLUSTER 结构。
- 优秀的分布式计算平台，既可以作为网络客户，又可以作为网络服务器。
- 支持 FAT、FAT32、NTFS 和 CDFS 等多种文件系统。
- 可以运行多数 16 位的 DOS 程序和 Windows 3.1 程序。
- 符合政府对支持 POSIX 1003.1 的要求。
- 支持政府和企业对操作系统安全性的要求。
- 支持 Unicode，适应对全球市场的需要。

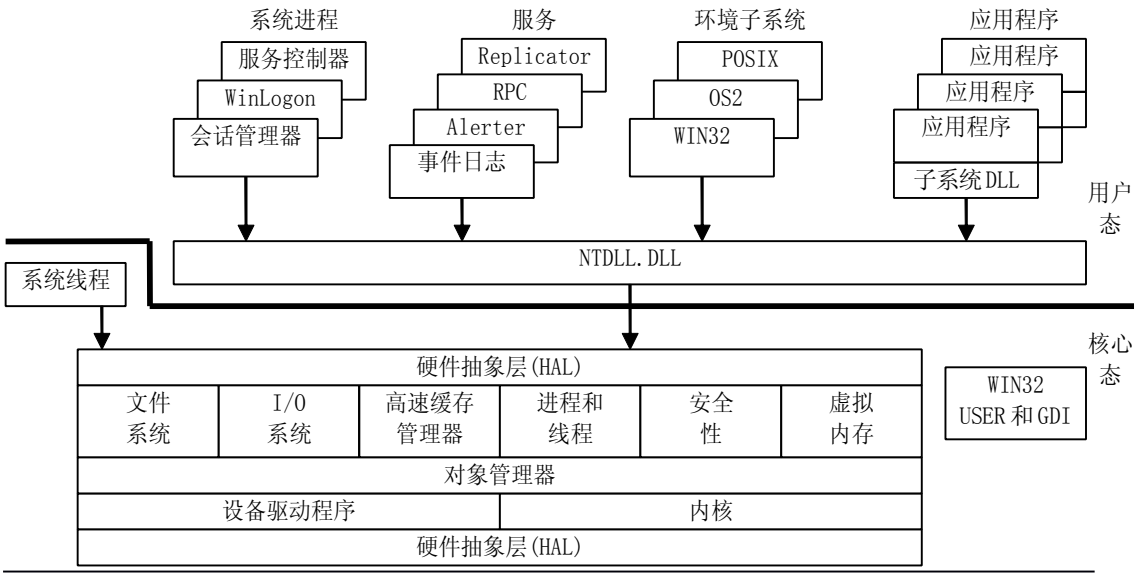
为创建符合上述需求的系统，Windows2000 系统结构的应达到如下的设计目标：

- 可扩充性：当市场需求变化时，代码必须易于扩充和改动。
- 可移植性：能够在多种体系结构中运行，并相对简单地移入新体系结构。
- 可靠性与坚固性：能够防止内部故障和外部侵扰造成的损害。
- 兼容性：与 DOS、Windows 的旧版本兼容，并和一些其他的操作系统如 Unix、OS2 和 Netware 相互操作。
- 性能：能够达到较高的效率。

为此，Windows2000 的设计者们认为：

- 采用整体式或层次式的操作系统体系结构是不恰当的，它们在可扩充性和可移植性方面效果不好。
- 采用类似于 Mach 的微内核结构也是不恰当的，纯的微内核设计只涉及最小内核，其他服务都运行在用户态，它的运算费用太高，在商业上不适用。

图 9-12　Windows2000 的系统结构



因此，Windows2000 的系统结构在纯微内核结构的基础上做了一些扩展，它融合了层次式结构和纯微内核结构的特点。对操作系统性能影响很大的组件在内核下运行，而其他一些功能则在内核外实现。如图 9-12 简单说明 Windows2000 的系统结构

在内核状态下运行的组件包括：内存管理器、高速缓存管理器、对象及安全性管理器、网络协议、文件系统、窗口和图形系统、及其所有的进程和线程管理。在核心态情况下，组件可以和硬件交互，也可以相互交互，不会引起描述表切换核模式转变。所有这些内核组件都受到保护，用户态程序不能直接访问操作系统特权部分的代码核数据，这样就不会被错误的应用程序侵扰。

除应用程序外，在用户状态下运行的还有系统进程、服务和环境子系统，他们提供一定的操作系统服务。

以下讨论 Windows2000 的关键系统组件。

9.8.2 Windows2000 的关键系统组件

1）硬件抽象层 HAL

Windows2000 的设计要点是在多种硬件平台上的可移植性，硬件抽象层 HAL 是实现可移植性的关键部分。HAL 是一个可加载的核心态模块 HAL.DLL，它为运行在 Windows2000 上的硬件平台低级接口，以隐藏各种与硬件有关的细节，例如 I/O、中断控制器、多处理器通信机制等。

2）设备驱动程序

设备驱动程序是可加载的核心态模块，通常以 SYS 为扩展名，它们是 I/O 系统和相关硬件之间的接口。Windows2000 的设备驱动程序不直接操作硬件，而是调用 HAL 的某些部分作为与硬件的接口。设备驱动程序包括以下几类：

- 硬件设备驱动程序：操作硬件（使用 HAL）读写物理设备或网络。
- 文件系统驱动程序：接受面向文件的 I/O 请求，并把它们转化为对特定设备的 I/O 请求。
- 过滤器驱动程序：截取 I/O 并在传递 I/O 到下一层之前执行某些增值处理，如磁盘镜像、加密。
- 网络重定向程序和服务器：一类文件系统驱动程序，传输远程 I/O 请求。

3）内核

内核执行操作系统最基本的操作，决定操作系统如何使用处理器并确保慎重使用它们。内核和执行体都存在于 NTOSKRNL.EXE，其中内核是最低层。内核提供如下一些函数：

- 线程安排和调度
- 陷阱处理和异常调度
- 中断处理和调度。
- 多处理器同步
- 提供由执行体使用的基本内核对象

内核与执行体在某些方面有所不同，它永远运行在核心态，也不能被其他正在运行的线程中断，为保持效率，代码短小紧凑、不进行堆栈和传递参数检查。

内核为严格定义的、可预测的操作系统基本要素和机制提供最低级的基础，允许执行的高级组件执行它们需要执行的操作。内核通过执行操作系统机制和避免制定策略而使其自身与执行体的其他部分分开。内核除了执行线程安排和调度外，几乎将所有的策略制定留给了执行体。

在内核以外，执行体代表了作为对象的线程和其他可共享的资源。这些对象需要一些策略开销，例如处理它们的对象句柄、保护它们的安全检查以及在它们被创建时扣除的资源配额。在内核中则要除去这种开销，因为内核执行了一组称作“内核对象”的简单对象，这些内核对象帮助内核控制中央处理并支持执行体对象的创建。大多数执行体级的对象都封装了一个或多个内核对象及其内核定义属性。

一个称作“控制对象”的内核对象集为控制各种操作系统功能建立了语义。这个对象集包括内核进程对象、APC 对象、延迟过程调用(DPC)对象和几个由 I/O 系统使用的对象，例如中断对象。

另一个称作“调度程序对象”的内核对象集合负责同步性能并改变或影响线程调度。调度程序对象包括内核线程、互斥体(在内部称作“变异体”)、事件、内核事件对、信号量、定时器和可等待定时器。执行体使用内核函数创建内核对象的实例，使用它们并构造为用户态提供的更复杂的对象。

内核的另外一个重要功能就是把执行体和设备驱动程序从 W1ndows 2000 支持的在硬件体系结构之间的变更中提取或隔离开来。这个工作包括处理功能之间的变更，例如中断处理、异常情况调度和多处理器同步。

即使对于这些与硬件有关的函数，内核的设计也是尽可能使公用代码的数量达到最大。内核支持一组在整个体系结构上可移植和在整个体系结构上语义完全相同的接　口。大多数实现这种可移植接口的代码在整个体系结构上完全相同。

然而，一些接口的实现因体系结构而异。或者说某些接口的一部分是由体系结构　特定的代码实现的。可以在任何机器上调用那些独立于体系结构的接口。不管代码是　否随体系结构而异，这些接口的

语义总是保持不变。一些内核接口(例如转锁例程)实际上是在 HAL 中实现的，因为其实现在同一体系结构族内可能因系统而异。

4）执行体

Windows NT 执行体是 NTOSKRNL.EXE 的上层(内核是其下层)。执行体包括五种类型的函数：

- 从用户态被导出并且可以调用的函数。这些函数的接口在 NTDLL.DLL 中。通过 WIN32 API 或一些其他的环境子系统可以对它们进行访问。
- 从用户态被导出并且可以调用的函数，但当前通过任何文档化的子系统函数都不能使用。这种例子包括 LPC 和各种查询函数，例如 NtQuerylnformationxxx，以及专用函数，例如 NtCreatePagingFile 等。
- 只能从在 Windows 2000 DDK 中已经导出并且文档化的核心态调用的函数。
- 在核心态组件之间调用的但没有文档化的函数。例如在执行体内部使用的内部支持例程。
- 组件内部的函数。

执行体包含下列重要的组件：

- 进程和线程管理器：创建及中止进程和线程。对进程和线程的基本支持在 Windows 2000 内核中实现；执行体给这些低级对象添加附加语义和功能。
- 虚拟内存管理器：实现“虚拟内存”。虚拟内存是一种内存管理模式，它为每个进程提供了一个大的专用地址空间，同时保护每个进程的地址空间不被其他进程占用。内存管理器也为高速缓存管理器提供基本的支持。
- 安全引用监视器：在本地计算机上执行安全策略。它保护了操作系统资源，执行运行时对象的保护和监视。
- I/O 系统：执行独立于设备的输入／输出，并为进一步处理调度适当的设备驱动程序。
- 高速缓存管理器：通过将最近引用的磁盘数据驻留在主内存中来提高文件 I/O 的性能，并且通过在把更新数据发送到磁盘之前将它们在内存中保持一个短的时间来延缓磁盘的写操作，这样就可以实现快速访问。正如您将看到的那样，它是通过使用内存管理器对映射文件的支持来做到这一点的。

另外，执行体还包括四组主要的支持函数，它们由上面列出的执行体组件使用。其中大约有三分之一的支持函数在 DDK 中已经文档化，因为设备驱动程序也使用这些支持函数。这四类支持函数包括：

- 对象管理器：创建、管理以及删除 Windows 2000 执行体对象和用于代表操作系统资源的抽象数据类型，例如进程、线程和各种同步对象。
- LPC 机制：在同一台计算机上的客户进程和服务器进程之间传递信息。LPC 是一个灵活的、经过优化的远程过程调用 RPC 版本，RPC 是一种通过网络在客户和服务器进程之间传递信息的工业标准通信机制。
- 一组广泛的公用“运行时库”函数，例如字符串处理、算术运算、数据类型转换和安全结构处理。
- 执行体支持例程：例如系统内存分配(页交换区和非页交换区)、互锁内　　存访问和两种特殊类型的同步对象：资源和快速互斥体。

5）NTDLL.DLL

NTDLL.DLL 是一个特殊的系统支持库，主要用于子系统动态链接库。NTDLL.DLL 包含两种类型的函数：

- 作为 Windows2000 执行体系统服务的系统服务调度占位程序。
- 子系统、子系统动态链接库、及其他本机映像使用的内部支持函数。

第一组函数提供了可以从用户态调用的作为 Windows NT 执行体系统服务的接口。这里有 200 多种这样的函数，例如 NtCreateFile、NtSetEvent 等。正如前面提到的那样，这些函数的大部分功能都可以通过 WIN32 API 访问。

对于这些函数中的每个函数，NTDLL 都包含一个有相同名称的入口点。在函数内的代码含有体系结构专用的指令，它能够产生一个进入核心态的转换以调用系统服务调度程序(稍后将在本章中详细解释)。在进行一些验证后，系统服务调度程序将调用包含在 NTOSKRNL.EXE 内的实代码的实际的核心态系统服务。

NTDLL 也包含许多支持函数，例如映像加载程序(此函数使用 Ldr 启动)、堆管理器和 WIN32 子系统进程通信函数(此函数使用 Csr 启动)以及通用运行时库例程(此函数使用 Rtl 启动)。它还包含用户态异步过程调用(APC)调度器和异常调度器。

6）系统进程

系统进程包括：

- Idle 进程：系统空闲进程，进程 ID 为 0。对于每个 CPU，Idle 进程都包括一个相应的线程，

用来统计空闲 CPU 时间。它不运行在真正的用户态。因此，由不同系统显示实用程序显示名称随实用程序的不同而不同，如任务管理器(Task Manager)中为 System Idle 进程，进程状态(PSTAT.EXE)和进程查看器(PVIEWER.EXE)中为 Idle 进程，进程分析器(PVIEW.EXE)、任务列表(TLIST.EXE)、快速切片(QSLICE.EXE)中为 System 进程.

- System 进程和 System 线程：系统进程的 ID 为 2，是一种特殊类型的只运行在核心态的 System 线程的宿主进程。它用于执行加载于系统空间中的代码，进程本身没有地址空间，必须从系统内存堆中动态分配。
- 会话管理器 SMSS.EXE：是第一个再系统中创建的用户态进程，用于执行一些关键的系统初始化步骤，包括创建 LPC 端口对象和两个线程、设置系统环境变量、加载部分系统程序、启动 WIN32 子系统进程（必要时包括 POSIX 和 OS2 子系统进程）和 WinLogon 进程等。在执行完初始化步骤后，SMSS 中的主线程将永远等待 CSRSS 和 WinLogon 进程句柄。另外，SMSS 还可作为应用程序和调试器之间的开关和监视器。
- WIN32 子系统 CSRSS.EXE：WIN32 子系统的核心部分。
- 登录进程 WinLogon.EXE：用于处理用户登录和注销。
- 本地安全身份身份鉴别服务器进程 LSASS.EXE：接收来自于 WinLogon 进程的身份验证请求，并调用一个适当的身份验证包执行实际验证。
- 服务控制器 SERVICES.EXE 及其相关服务器进程：启动并管理一系列服务进程。

7）服务

服务类似于 Unix 的守护进程，如“事件日志”和“调度”等服务，许多附加的服务器应用程序，例如 Microsoft SQL Server 和 Microsoft ExChange Sewer，也包含了作为 Windows2000 服务运行的组件。

8）环境子系统

环境子系统向用户应用程序展示本地操作系统服务，提供操作系统“环境”或个性。Windows2000 带有三个环境子系统：WIN32、POSIX 和 OS/2 1.2。

9）用户应用程序和子系统动态连接库

用户应用程序可以是 WIN32、Windows 3.1、MS-DOS、POSIX 或 OS/2 1.2 五种类型之一。

在图 9-12 中，请注意“子系统动态链接库”框在“用户应用程序”框之下。在 Windows2000 中，用户应用程序不能直接调用本地 Windows 2000 操作系统服务，但它们能通过一个或多个“子系统动态链接库”调用。子系统动态链接库的作用是将文档化函数转换为适当的非文档化的 Windows 2000 系统服务调用。该转换可能会给正在为用户应用程序提供服务的环境子系统进程发送消息，也可能不会。