操作系统原理与设计

CH1　概论

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，软件能充分发挥和扩充硬件功能，完成各种系统及应用任务，两者互相依存、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。

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。

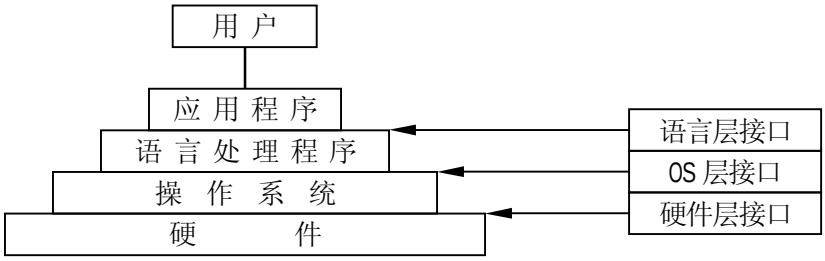


图 1-1 计算机系统层次结构

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，为编译系统设计者提供了有力支撑。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。应用程序要求操作系统提供的共性服务大致有：

- 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。当程序执行出现异常时，应能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法实现 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，系统将提供信息保护机制。
- 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。

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。
- 提高系统效率——OS 应该使计算机系统的资源充分利用，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。

1.1.1 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。它在硬件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。

1.1.2 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。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。例如，操作系统决定将处理器在何时、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？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，最大限度地实现各类资源的共享，从而，使计算机系统的效率有很大提高。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。

1.1.3 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上加上一层又一层的软件来管理整个计算机系统，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。如果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。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。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。由于操作系统包含了许多层软件，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机器（Virtual Machine）。

1.2 操作系统的形成和发展

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：

（1）人工把源程序用穿孔机穿制在卡片或纸带上；

（2）将准备好的汇编解释程序或编译系统装入计算机；

（3）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；

（4）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；

（5）通过引导程序把装在输入机上的目标程序读入计算机；

（6）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；

（7）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。

这种工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%；I/O 计算机速度提高到每秒 10 万次，此时，作业出运行时间仅需 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。

1.2.2 管理程序阶段

早期的计算机系统非常昂贵，为了充分利用计算机时间，减少空闲等待，缩短作业的准备和建立时间，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。

在管理程序的控制下，可以自动控制和处理作业流，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。

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，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，这比早期批处理系统提高了处理能力。

管理程序 Monitor 的内存组织如图 1-2 所示，它的主要功能有：

- 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：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 1-2 管理程序内存组织

1.2.3 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。

1、操作系统实现了计算机操作的自动化。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。

2、资源管理水平有了提高，实现了外国设备的联机同时操作，进一步提高了计算机资源的利用率。

3、存储管理功能得到加强，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。

4、开始支持分时操作，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。

5、文件管理功能有改进，数据库系统开始出现。

6、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

促使操作系统不断发展的主要动力有以下几个方面：

1、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。

2、计算体系结构不断发展。硬件的改进导致 OS 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。

3、提高计算机系统的资源利用率的需要。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。

4、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。

5、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，便出现了实时操作系统；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。

1.2.5 操作系统的发展

至今，已经设计了许多优秀的或成功运行的操作。例如，Atlas（英国，Manchester 大学）、XDS-940（美国，加利福尼亚 Berkeley 分校）、Multics（美国，MIT）、OS/370（美国，IBM）、Unix（美国，贝尔实验室）、VM/370（美国，IBM）、MCP（美国，Broughs 公司）和 TENEX（美国，Dec 公司）、Scope（美国，CDC 公司）等等。

从操作系统的形成和发展可以看出，它是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起来的，进入 80 年代以后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：（1）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。（2）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。（3）高性能　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，导致计算机系统性能大大提高。（4）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，下面简介微型机上最具代表性的操作系统：

- DOS：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 80 年代。
- Windows：Microsoft 公司开发的图形化用户界面操作环境 Windows，1990 年宣布的 Windows 3.0 首获成功，1992 年发布的 Windows 3.1 在全世界流行，此后又推出了 Windows NT， Windows 95，Windows 97、Windows98 和 Windows2000 能独立在硬件上运行，是真正的新型操作系统。微软还推出嵌入式 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。
- OS/2：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、图形界面、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。
- 微机 Unix：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 年开发成功，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。目前在微机上运行的 Unix 类操作系统中比较有名的有 XENIX 和 Unix SVR4。
- Macintosh：美国 Apple 公司推出的 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。
- MINIX：荷兰 Vrije 大学计算机系教授 Andrew S. Tanenbavm 开发的一个与 Unix 兼容，然而内核全新的学习型操作系统，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。
- Linux：芬兰的一个学生 Linvs Torvalds 编写的一个类似 Minix 的系统，但是它功能繁多，面向实用而非教学。Linux 是一个充满生机，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，是唯一

能与 Unix 与 Windows 较量和抗衡的操作系统。

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：

- V-Kernel，美国 Stanford 大学
- Meglos，美国 Bell 实验室。
- MACH，美国卡内基梅隆大学。

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

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 流行操作系统简介

1.3.1 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 80 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。

1.3.2 Windows 操作系统

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，1990 年宣布 Windows 3.0，对原来系统做了彻底改造；1992 年 4 月 Windows 3.1 发布后在全世界流行，此后又推出了 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。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。

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。

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。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，即多个任务可基于多个线程对称地分布到各个 CPU 上工作，从而大大提高了系统性能。
- 支持抢先的多任务处理。
- 32 位页式授权存储管理。
- 具有容错功能，C2 安全级。
- 可移植性好，既可在 Intel x86 也可在 MIPS RISC 平台上运行。
- 集成网络计算，支持 LAN Manager, 为其他网络产品提供接口。
- 支持世界范围字符集。

4、Windows 2000

Windows2000 是在 Window NT 基础上修改和扩充而成的。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。

1.3.3 OS/2 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、图形界面、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。

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。

1.3.4 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 年开发成功，1972 年用 C 语言改写，使得 Unix 具有高度易读性、可移植性。Unix 取得成功的最重要原因是系统的开放性，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，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，SCO 公司的 Unix SCO 以及 AT&T 公司自己的 Unix SystemV 等。

1.3.5 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。

1.3.6 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。

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/

1.3.7 Linux 操作系统

Linux 的产生和发展将在下一节介绍。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，是唯一能与 Unix 与 Windows 较量和抗衡的操作系统，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，可用于各种 PC 机和工作站；(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；(9) 提供 GUI；(10) 支持并行处理和实时处理，能充分发挥硬件性能；(11) 在 Linux 平台上开发软件成本低。

Linux 具有强大功能，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羔；可仅供个人使用，应用软件应有尽有；此外，Linux 的 Win 仿真 Win3.1、Win95，能够使用 Word7.0 等大部分程序；Linux 下有图形接口 X-Windows，有多种窗口管理器，fvwm-95 提供 Win95 的界面；Linux 的 CxTerm、Chdrv 可处理中文；GNU C++是最好的 C++编译器之一，Fortraan、PASCAL 等都有 Linux 版本。

1.3.8 MACH 操作系统

MACH 是由 CMU 于八十年代中期开始研制的一个并行分布式操作系统，其设计目标是支持松散、紧耦合多处理机体系结构(Loosely-coupled Distributed Memory Multiprocessor) ，以及多处理器与传统单机形成的网络环境。MACH 是采用微内核(Micro-Kernel) 技术实现操作系统的典范。它包括了一个很小的内核和建立在内核之上的各种服务模块。内核中支持以下概念的操作：任务(task) 、线程(thread) 、端口(port) 、消息(message) 、内存对象(memory object) 。内核之上的服务模块有：命名服务器(name server) 、网络服务器(network server) 、内存对象服务器(memory object server) 、处理器服务器(CPU server) 、Unix 务器(Unix server) 等。MACH 具有内核可精减、可移植性好、网络透明性、方便灵活、效率高、实时性好等优点，影响了许多其他操作系统

的设计。

1.4 操作系统的分类

1.4.1 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。

1.4.2 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（Tine 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。

分时的思想于 1959 年由 MIT 正式提出，并在 1961 年开发出了第一个分时系统 CTSS，成功地运行在 IBM 7090/7094 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：若干个终用户同时联机用计算机，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。
- 及时性：终端用户的立即型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地相关。
- 交互性：人机交互，联机工作，使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，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。

1.4.3 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、全安保密，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，由它们反映和控制某些外部事件，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。

1.4.4 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。

为了信息传送和资源共享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）。它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。
- 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。
- 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。

1.4.5 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。
- 系统中的计算机没有主次之分，即没有整个系统的主机，也没有受控于主机的从机。
- 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，一般的网络是不能满足这个条件的，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。

从上面叙述可以看出，分布式系统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；
- 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、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。

1.4.6 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，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、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。

1.4.7 自由软件和 Linux 操作系统

自由软件（Free Software）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 级是指可对软件的自由修改；自由之 2 级是指对软件的自由使用；自由之 3 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‘素取型’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。

GNU 的含义是 GNU is not Unix 的意思，是 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，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。

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，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。

Linus 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。除了操作系统通常应具备的功能外，它还具有许多新功能，如，异种通信联网、符合 POSIX 标准，支标多达 32 种文件系统、支持并行处理和实时处理，强大的远程管理功能等等。

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。

1.5 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。

1.5.1 处理机管理

处理器管理的第一项工作是处理中断事件，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。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
- 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。

1.5.2 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动；还能够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。

1.5.3 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，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

1.5.4 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

1.5.5 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个友好的用户接口，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。

1.6 操作系统提供的用户接口

操作系统向用户提供的用户接口可以分成三种：

1.6.1 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。在完成命令规定的任务后，控制返回，等待用户进入下一条命令，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。

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，人们可以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或处理不同的数据文件，大大增强了命令接口的处理能力。

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名字和参数，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越的工具。

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。

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。

1.6.2 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。

1.6.3 程序接口——系统调用

程序接又称应用编程接口 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。国际标准化组织给出的系统调用的国际标准 POSIX 9945—1，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，因而只能在汇编语言编程的程序中，才能直接使用系统调用；而在高级语言中，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。最新推出席的一些操作系统，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，可直按使用系统调用。

系统调用按功能大致可以分成：进程管理、进程控制、文件操作、内存管理、资源申请、系统控制等类别。

1.7 操作系统的主要特性和需要解决的主要问题

1.7.1 操作系统的主要特性

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；I/O 设备和 CPU 计算同时进行；内存中同时有多个作业被启动执行，这些都是并发性活动的例子。由此引发了一系统的问题：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？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。

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，以及数据都是临界资源，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。

3、异步性（Asynchronism)

操作系统的第三个特点是异步性，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。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。

1.7.2 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、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、I/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 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 技术，可以用一类物理 I/O 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，每个用户都从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，提供虚拟存储器，使得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，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。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。

因为每台虚拟机一个裸机完全一样，所以，同一台裸机上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。

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。

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，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。

CH2　　操作系统的运行环境

操作系统是管理、调度系统资源，方便用户使用的程序的集合。任何一个程序在计算机上运行是要有一定条件的，或者说要有一定的运行支撑环境，例如要有处理机执行、主存储器暂存信息、输入输出设备进行交互和有关的系统软件的支撑。而操作系统作为计算机系统的管理程序，为了实现其预定的各种管理功能，同样需要有一定的条件，或称之为运行环境来支持其工作。

计算机硬件系统主要包括处理器、主存储器、输入输出设备和通信系统。操作系统是对硬件系统的首次扩充，它的运行直接依赖于系统的硬件环境，与硬件的关系尤为密切，同时它全面管理和调度系统资源，与硬件的接口很多且较为分散。本章讨论操作系统对其运行环境的全面要求，主要着眼点在于操作系统各类管理技术均要用到的基本硬件技术和概念，而与操作系统几大管理功能密切相关的具体硬件条件则放到以后各章详细讨论。

2.1　　中央处理器

2.1.1　单机系统和多机系统

程序最终是要在处理器上执行的，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。

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：

- 内存缓冲寄存器（MBR）：又称通用寄存器，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。
- 内存地址寄存器（MAR）：它们可以被系统程序和用户程序直接使用，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（I/OAR）：用于指定 I/O 设备。
- I/O 缓冲寄存器（I/OBR）：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（PC）和指令寄存器（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。

2.1.3　程序状态字寄存器

为了方便操作系统理论研究和各种算法的讨论，我们经常提及程序状态字 PSW（Program Status Word）。程序状态字 PSW 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。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三部份内容：

（1）程序基本状态。

- 程序计数器：指明下一条执行的指令
- 条件码：表示指令执行的结果状态。
- 处理器状态位：指明当前的处理器状态。

（2）中断码。保存程序执行时当前发生的中断事件。

（3）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 360/370 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 包括内容有：系统屏蔽位 8 位，保护键 4 位，PSW 基本/扩充格式、开/关中断、运行/等待、目态/特态各 1 位，中断码 16 位，条件码 2 位，程序屏蔽 4 位和指令地址 24 位。

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。

2.1.4 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。

机器指令可以分为以下四类：

- 数据处理类指令：用于执行算术和逻辑运算。
- 控制类指令：用于改变执行序列。
- 寄存器数据交换类指令：用于在处理器和存储器之间交换数据。
- I/O 类指令：用于和外围设备交换数据。

2.1.5 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，取 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）。

2.1.6 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。

以 Pentium 的保护机制为例，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。
- 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 操作系统只使用了 0 级和 3 级。

2.2 中断技术

2.2.1 中断的概念

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，只要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。

2.2.2 中断源

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。

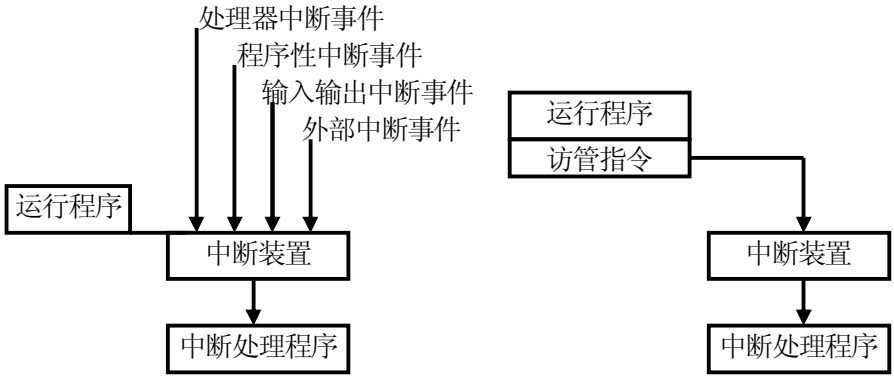


图 2-1　两类中断事件

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，例如要求操作系统协助启动外围设备工作。图 2-1 表示出了上述的两类中断事件。

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好，然后将处理中断程序的程序状态字送入处理器的程序状态字寄存器，从而就引出了处理中断的程序。

处理中断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、即引出处理程序)。

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固定单元，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

1、中断处理程序

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、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送主存储器保存起来，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：

- 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。

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

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。

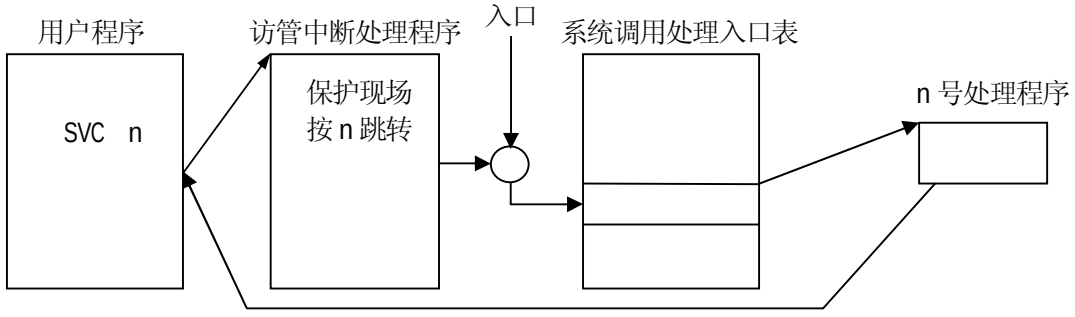


图 2-2　系统调用的共性处理流程

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。如图 2-2 所示，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

用户可以使用广义指令向操作系统提出启动外围设备的要求；控制程序的要求(例如，中断续元处理后的返回)；申请主存资源、外围设备的要求等等。一般说，广义指令分成：文件操作类、作业管理类、进程控制类、资源申请类等。

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，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，这将在以后的章节中讨论。

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

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，中断装置特技预定的顺序来响应，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。

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顺序。

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。

3、多个中断事件的处理

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的处理　期间。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，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，一般不超过三级为宜，因为过多级的‘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。

2.2.6　实例研究：Windows 2000 的中断处理

1、Windows 2000的中断处理概述

在 Pentium 和 Windows 的中断实现中，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。

图 2-3 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，它将在纪录机器状态（机器状态将在另一个中断或异常发生时抹去）时，暂时禁用中断。它将创建一个“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；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。

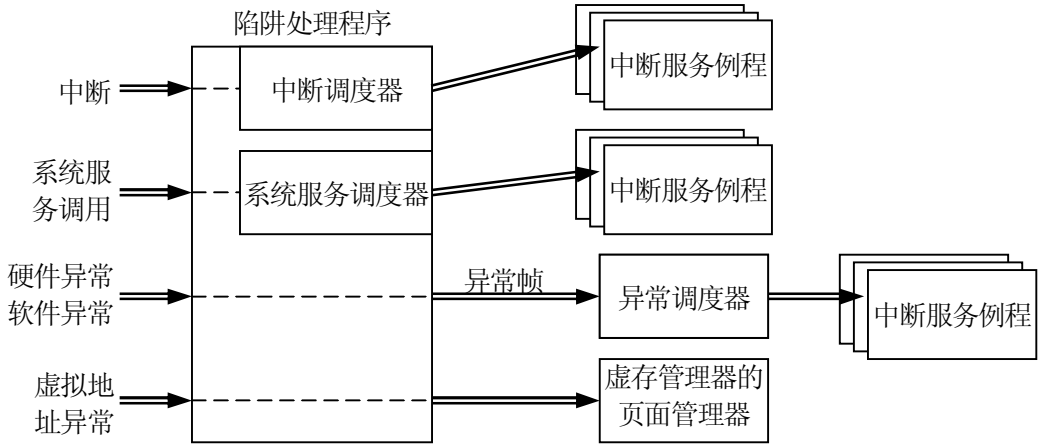


图 2-3　Windows2000 的陷阱调度

2、Windows 2000的中断调度

1）中断类型和优先级

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 的标准集上。这一组内核维护的 IRQL 时可以移植的，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的服务。

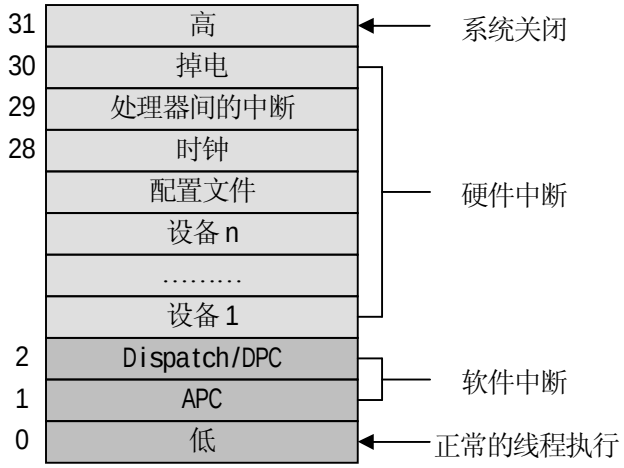


图 2-4　x86 体系结构 Windows 的中断请求级

图 2-4 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，抵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。

在 Windows2000 中，每一个处理器都可以识别不同的中断号和中断类型。具体来说，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 2-5 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，处理器不会向应该中断，直到一个正在执行的线程降低了 IRQL。

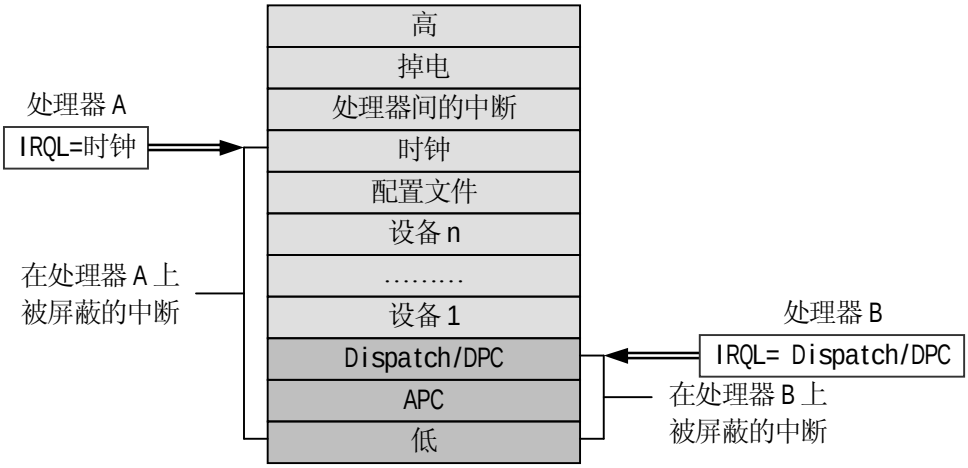


图 2-5　Windows 的中断屏蔽

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 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 2-6 所示。

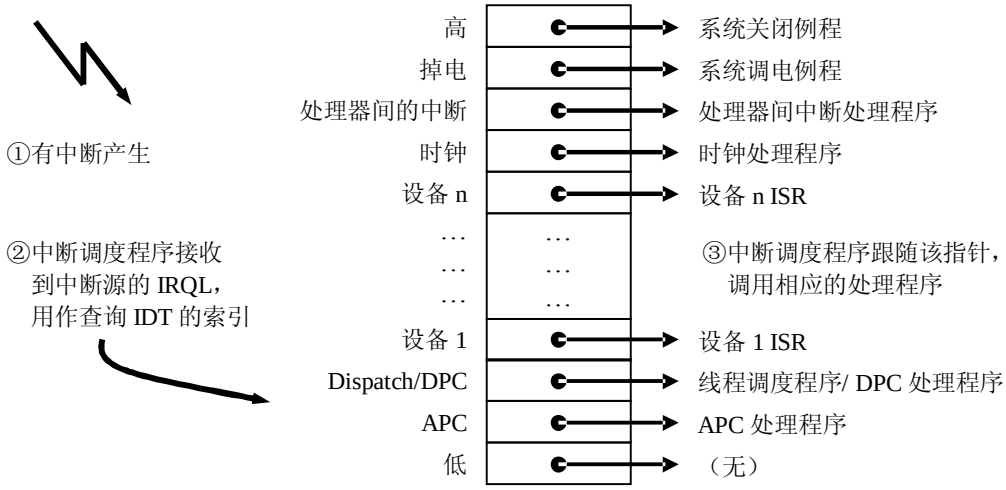


图 2-6　Windows 的中断服务

在 x86 系统中，IDT 是处理器控制区 PCR 指向的硬件结构。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、定点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。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）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，有时内核再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 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 是执行系统任务的延迟函数，该任务比当前任务次要，从而他们可能不立即执行。DPC 位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和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 2-7 描述了 DPC 的处理。

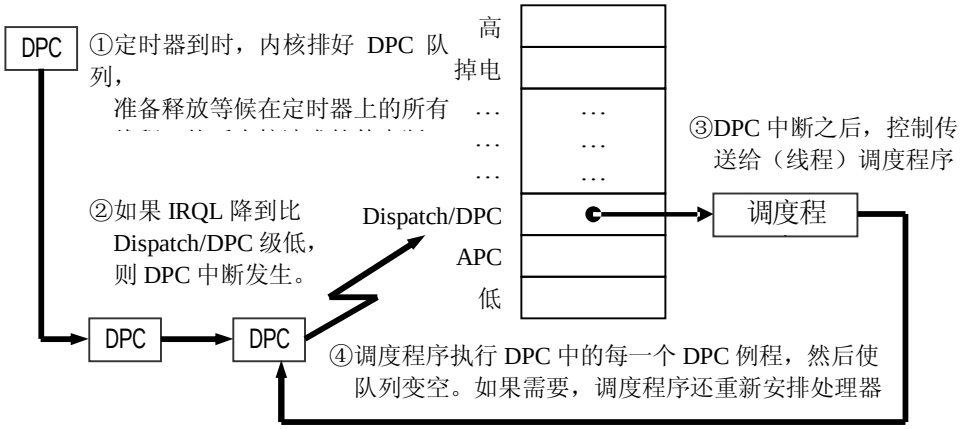


图 2-7　提交 DPC

（2）异步过程调用（APC）中断

异步过程调用（APC）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、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，展开堆栈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。

内核俘获核处理某些对用户透明的异常，如调试断点异常。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。

3、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。如图 2-8 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。

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。

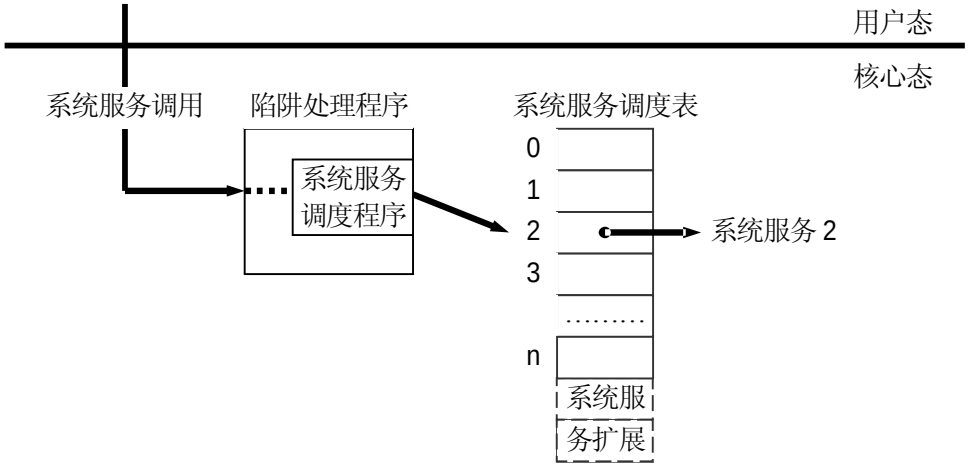


图 2-8　Windows200 的系统服务异常

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。

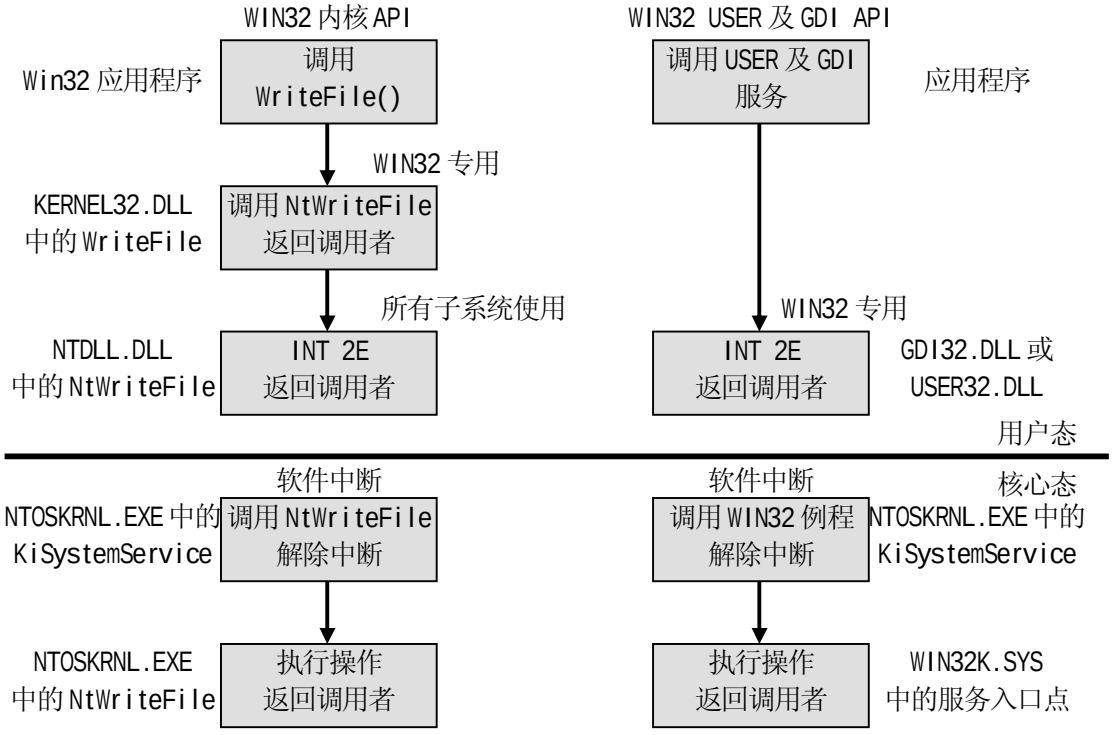


图 2-9　Windows200 的系统服务调度

如图 2-9 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。

2.3 主存储器

2.3.1 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 2-10 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。

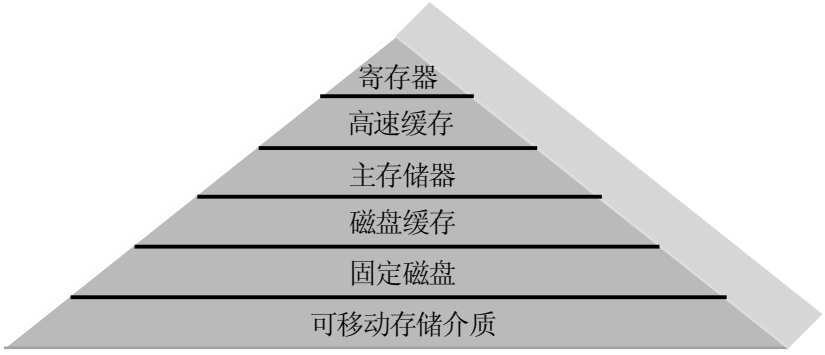


图 2-10　计算机系统存储器的层次

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他处理时间，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：寄存器操作数的使用，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。

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存储容量大的多的存储空间。

2.3.2 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，我们把程序在内存种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。

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。

2.4 输入输出系统

2.4.1 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，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。

2.4.2 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。

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。早期计算机和微机往往采用这种方式，中央处理机的大量时间用在等待输入输出的循环检测上，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。

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。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队 1 秒。若为外围设备增加缓冲寄存器存放数据，则可大大减少中断次数。中央处理器在外围设备与缓冲寄存器交换信息期间可执行其它指令。例如行式打印机、卡片机、字符显示器等均配置数据缓冲寄存器，提高了中央处理器和外围设备并行工作的程度。

3、DMA方式

在 DMA(直接主存存取)方式中，I/O 控制器有更强的功能，它不仅设有中断机构，而且，还增加了 DMA 控制机构。在 DMA 控制器的控制下，它采用‘偷窃’总线控制权的方法，让设备和主存之间成批交换数据，而不必由 CPU 干予。这样可减轻 CPU 的负担，因每次传送数据时，不必进入中断系统；只要 CPU 暂停几个周期，从而使 I/O 数据的速度也大大提高。

目前，在小型、微型机中的快速设备均采用这种方式，DMA 的操作全部由硬件实现，不影响 CPU 寄存器的状态。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。

4、通道方式

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。上例中，中央处理器只要用一条启动指令，就可通知输入机输入 1000 个字符，一秒钟之后发生中断再去处理。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。如果按在字节多路通道上，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，它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。

CH3　进程与线程

3.1　多道程序设计

3.1.1　多道程序设计的概念

多道程序设计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进行计算的方法。

为什么要采用多道程序设计技术呢?因为，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 3-1 所示，不难看出在这个计算过程中，处理器的利用率为：

$$52 / (78 + 52 + 20) \approx 35\%$$

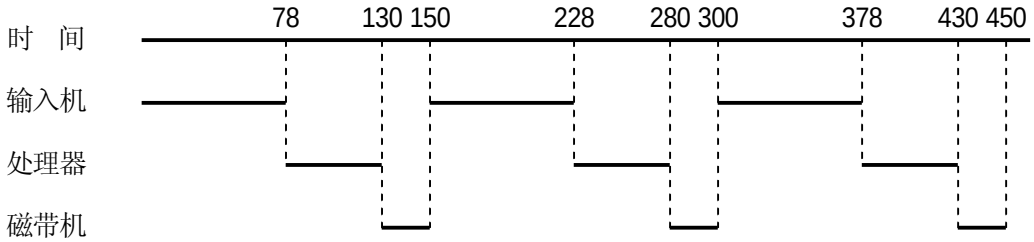


图 3-1　单道算题运行时处理器的使用效率

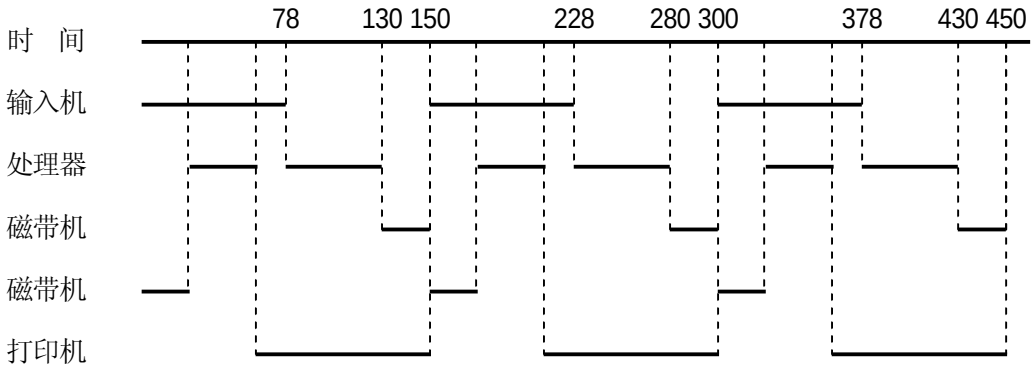


图 3-2　两道算题运行时处理器的使用效率

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 72 毫秒时间才能输入完毕。

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，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。不难算出，此时处理器的利用率为：

（52+42）/ 150 ≈ 63%

从上面的分析可以看出，让几道程序同时进入计算比一道道串行地进行效率要高，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法后，可以提高处理器的效率，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，采用多道程序设计方法时只要大约 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。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具体要求也影响道数的多寡。

3.1.2 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。

1、存储保护与程序浮动

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，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。

关于存储保护和程序浮动的实现技术可以有多种办法，在有关存储管理的章节里

将作具体介绍。

2、处理器的管理和调度

在多道程序设计系统里，如果仅配置一个处理器，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 D 传输完毕的等待状态，当 D 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 D 传输结束事件，而 D 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。

负责管理一道程序占有或释放处理器的软件称为处理器调度，有关它的算法和实现将在后面叙述。

3、资源的管理和调度

在多道程序设计系统里，系统的资源为几道程序所共享，上节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。

3.2 顺序性与并发性

3.2.1 程序执行的顺序性

程序执行的顺序性是指每个程序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。
- 程序环境的封闭性。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。
- 程序执行结果的确定性。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，最大的缺点是导致计算机系统效率不高。

3.2.2 程序执行的并发性

程序的并发性是指一组程序在执行时间上是重叠的。所谓执行在时间上是重叠的，是指执行一个程序的第一条指令是在执行另一个程序的最后一条指令完成之前开始。例如：有两个程序 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 两个程序而言，它们的执行顺序分别为 al，a2，a3 和 bl，b2，b3，这是程序执行的顺序性。然而，这两个程序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 两个程序的执行是并发的。

程序并发执行时可能是无关的，也可能是交往的。无关性是指并发执行的程序分别在不同的数据集合上操作，所以一个程序的执行与其它并发执行的程序的进展无关，即一个程序的执行不会改变另一个正在执行程序的变量值。然而，程序并发执行时可能是交往的，它们共享某些变量，所以一个程序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。

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(1) { input，process，output }来 编制的，所以这个程序只能顺序地执行。图 3-3(a)是串行输入-处理-输出的示意图，这时系统的效率是相当低的。

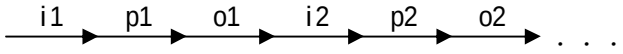


图 3-3(a) 串行工作

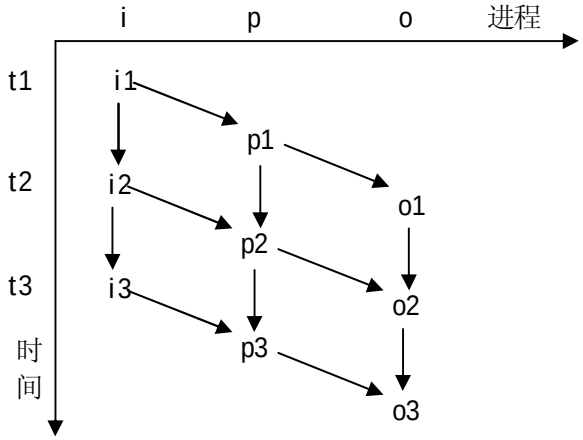


图 3-3(b) 并行工作

如果把这个问题的程序分成三部分：

while(1) { input，send }

while(1) { receive，process，send }

while(1) { 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 500 个字符，将读入字符送缓冲区 1。

模块 2：循环执行：处理缓冲区 1 中 500 个字符，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。

从图 3-3(b)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 3-4 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＋52＋20）

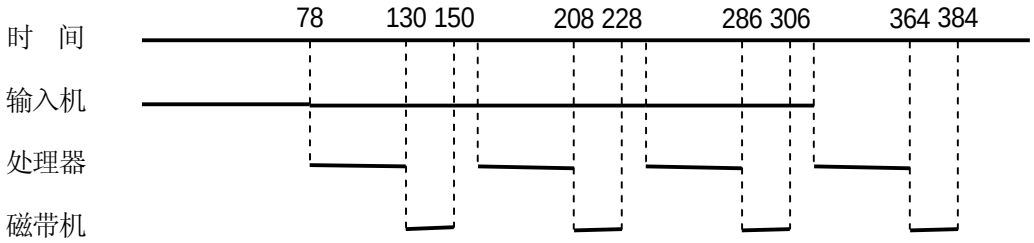


图 3-4 采用并发程序设计时处理器的使用效率

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。

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，通过共享这些共享变量或其他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 识备，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，提高系统的效率。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。

3.3 进程的基本概念

3.3.1 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。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并和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 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，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。
- (制约性) 并发进程之间存在着制约性，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。
- (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 P1，P2，…，Pm 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 P1 执行了 n_1 条指令后让出处理器给 P2，P2 执行了 n_2 条指令后让出处理器给 P3，…，Pm 执行了 n_m 条指令后让出处理器给 P1，…。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“并发性”。

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 点时需要将信息记在磁盘上，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，这时可让编译程序 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?称它为在 B 点等待磁盘传输，还是称它正在从 A 点开始执行?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，而进程 P 乙正在从 A 点开始执行。

3.3.2 进程的状态和转换

1、三态模型

为了便于管理进程，一般来说我们可按进程在执行过程中的不同状况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 3-5 表示进程的状态转换。

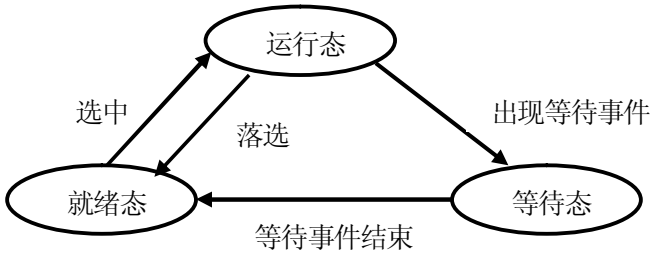


图 3-5　进程的状态转换

运行状态的进程将由于出现等待事件而进入的等待状态，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；等待外设传输；等待人工干预。
- 等待态—→就绪态：资源得到满足；外设传输结束；人工干预完成。
- 运行态—→就绪态：运行时间到；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。

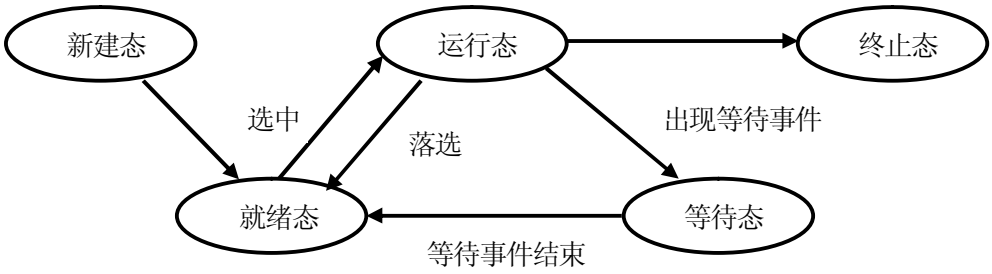
2、五态模型

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 3-6 给出了进程五态模型及其转换。

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，然后是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。

类似的，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，操作系统建删除该进程。

图 3-6　进程五态模型及其转换

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：但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。

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：

- 操作系统中的进程 0 均处于等待状态，处理器空闲，此时需要把一些进程对换出去，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。
- 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。
- 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 3-7 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，也可以决定把等待态进程将被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将被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，原则上时不需要把它调入内存的。但是在下面一种情况下，这一状态变化时可能的。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，它需要抢占了 CPU，，而此时主存空间有不够，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。

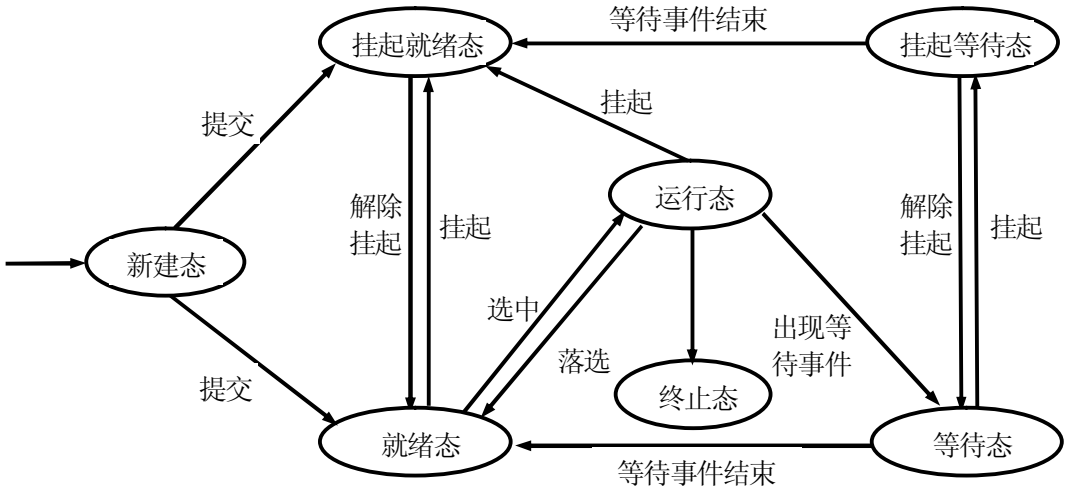


图 3-7　具有挂起功能的进程状态

不难看出，我们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。

3.3.3 进程的描述

1、操作系统的控制结构

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，通常是通过构造了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，存储控制表，I/O 控制表，文件控制表和进程控制表。

- 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。
- 进程控制表用来管理进程及其相关信息。有关进程控制结构以及进程的控制将在下面详细介绍。

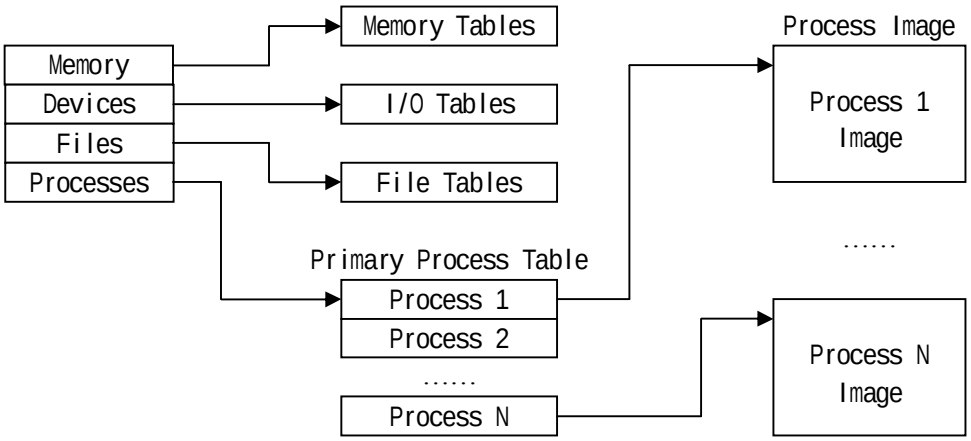


图 3-8　操作系统控制表的通用结构

图 3-8 给出了操作系统控制表的通用结构，虽然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。

最后要指出的是，图 3-8 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、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？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、状态信息和控制信息。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 3-9 所示：

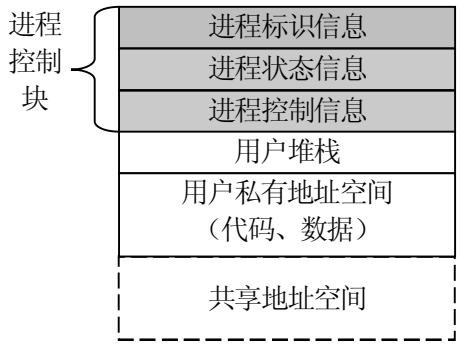


图 3-9　用户进程在虚拟内存中的组织

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，简称 PCB 或 TCB) ，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，如状态、等待事件或等待原因、优先级、采用的进程调度算法、队列指引元等；2）进程间通信相关信息，如消息队列指针、信号量；3）进程在二级存储器内的地址；4）资源的占用和使用信息，如已进程占用 CPU 的时间、进程已执行的时间总和；5）进程特权信息，如在内存访问和处理器状态方面的特权。6) 资源清单，包括进程所需全部资源、已经分得的资源。

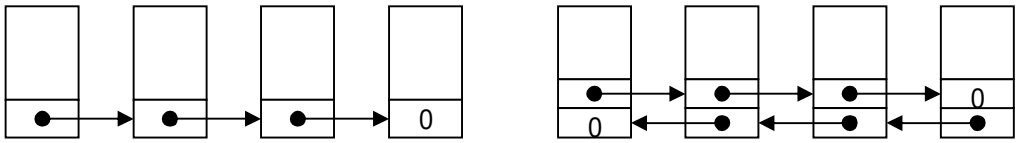
进程控制块是操作系统中最为重要的数据结构，每个进程控制块包含了操作系统管理所需的所有进程信息，进程控制块的集合事实上定义了一个操作系统的当前状态。进程控制块使用或修改权仅属于操作系统程序，包括调度程序、资源分配程序、中断处理程序、性能监视和分析程序等。有了进程控制块进程才能被调度执行，因而，进

程控制块可看作是一个虚的 CPU。

4、进程队列

一般说来，处于同一状态(例如就绪态)的所有进程控制块是连接在一起的。这样的数据结构称为进程队列，简称队列。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，此进程就连接到与该设备相关的等待态队列中去。

在一个队列中，连接进程控制块的方法可以是多样的，常用的是单向连接和双向连接。单向连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。双向连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。这两种连接方式如图 3-10(a)和(b)所示。



(a) 单向连接　　(b) 双向连接

图 3-10　进程控制块的连接

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 3-11 给出了操作系统的队列管理和状态转换示意图。

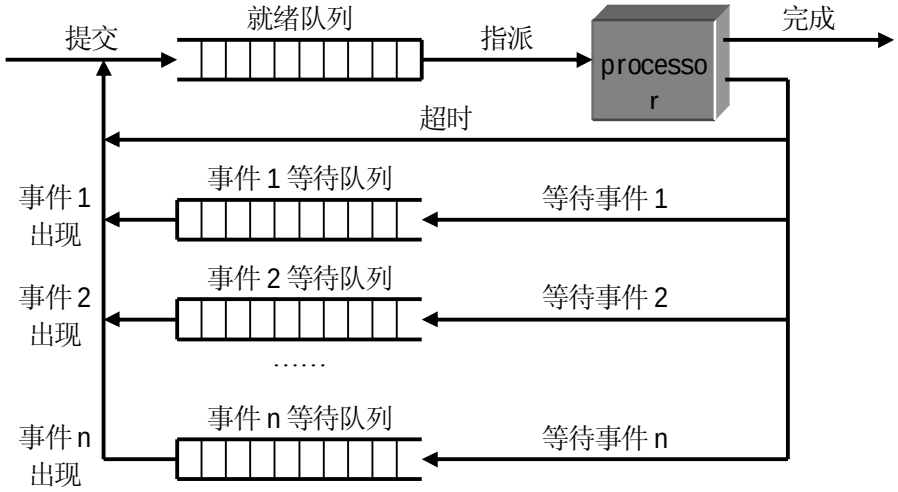


图 3-11　给出了操作系统的队列管理和状态转换示意图

现在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。假设采用双向连接，如图 3-10(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。

3.3.4 进程的控制

通常操作系统提供了若干基本操作以管理和控制进程，称之为进程控制原语。常用的进程控制原语有：

- 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。
- 改变优先数原语。
- 调度进程原语。

通过这一组原语，操作系统就可以有效地控制和管理进程。本节讨论进程控制，有关进程调度和中级调度的进程控制将在以后各章进一步讨论。

1、进程的创建

每一个进程都有生命期，即从创建到消亡。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。
- 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。
- 通知操作系统的某些模块，如记账程序、性能监控程序。

2、进程的切换

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，以便另外一个进程运行。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。
- 根据被选中进程恢复处理器现场。

为了进一步说明进程切换，我们来讨论一下模式切换。当中断发生的时候，中断正在执行的用户程序，把用户状态切换到内核状态，去执行操作系统程序以获得服务，这就是一次模式切换。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程序。

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。

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，系统调用撤销原语终止进程：
- 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，应先撤销元的所有子孙进程，以防它们脱离控制；
- 撤销进程出队，将它的 PCB 归还到 PCB 池。

3.3.5 进程管理的实现模型

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的是一个程序集合，哪么它是如何来管理进程的呢？它本身是不是一个进程呢？不同的操作系统有着不同的处理方法，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 3-12 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。

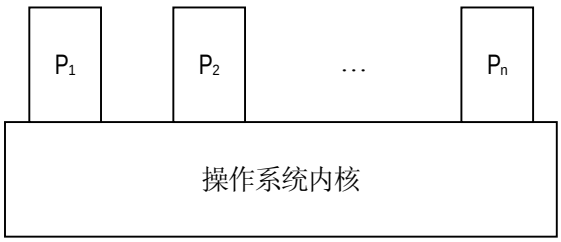


图 3-12　非进程内核模型

在这种情况下，进程执行的概念仅仅是用户程序和用户地址空间，操作系统代码作为一个分离实体在内核模式下运行。

2、用户进程模型

小型机和微型机操作系统往往采用用户进程模型，如 Unix、Windows、Windows-NT 等。如图 3-13（a）所示，在用户进程模型中，大部分操作系统程序组织成一组例行程序供用户程序调用其功能，并在用户进程的上下文环境中执行。

图 3-13（b）给出了用户进程模型中的进程映像，它既包括进程控制块、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。

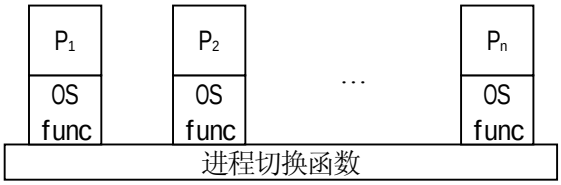


图 3-13（a） 用户进程模型

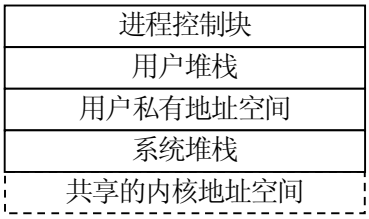


图 3-13（b） 用户进程模型的进程映像

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，但是进程切换并没有发生，该用户进程依然在执行。

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当前进程被中断的用户程序。在这种情况下，不必要通过进程切换就可以中断用户程序来调用操作系统例行程序。

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 3-13（a）中，我们把它在逻辑上分离出来。

3、基于进程的实现模型

基于进程的实现模型把操作系统程序组织成一组进程。如图 3-14 所示，主要的操作系统内核功能被组织在一组分离的进程内实现，这组进程在内核模式下运行，而进程切换例行程序的执行仍然在进程之外。

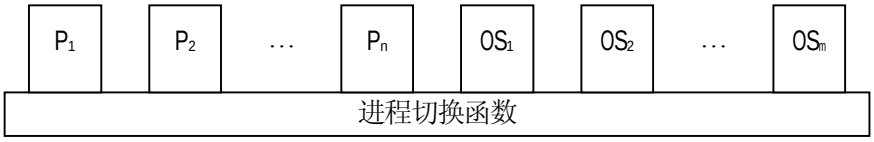


图 3-14 基于进程的实现模型

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，模块之间居于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里的进程，有利于操作系统的实现、配置和扩充。最后这一结构在多处理器和多计算机的环境下非常有效，有利于系统效率的提高。

3.3.6 实例研究——Unix SVR4 的进程管理

Unix SVR4 的进程管理的实现采用用户进程模型，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、中断和例外将引起模式切换。

1、Unix SVR4的进程状态

图 3-15 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 3-13 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。

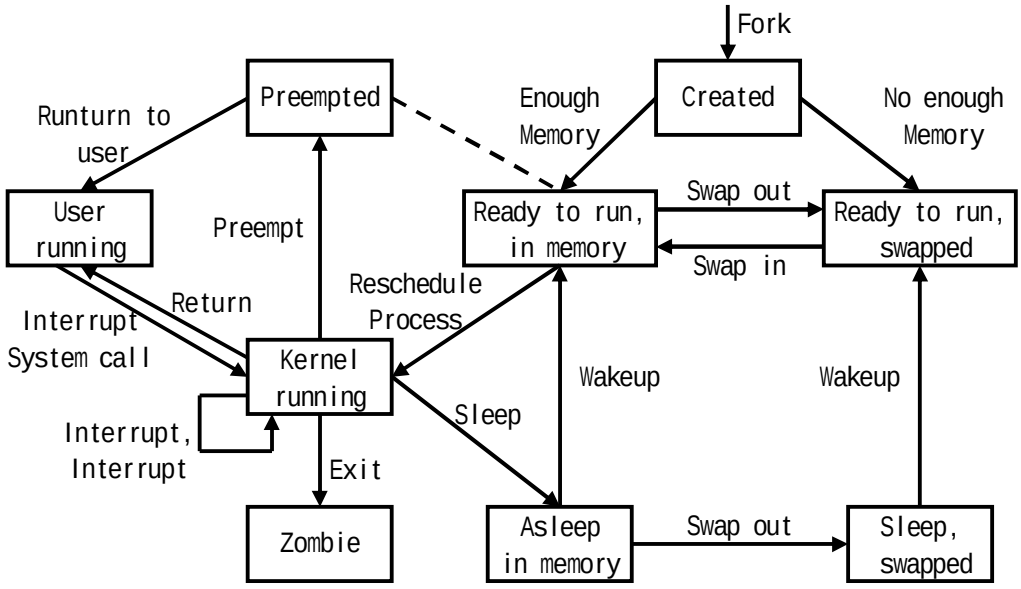


图 3-15　Unix SVR4 进程状态及其转换

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：就绪状态，被对换出内存。
- Created：新建状态。
- Zombie：终止状态。

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、Unix SVR4的进程描述

Unix SVR4 的进程映像包括用户级上下文、寄存器上下文和系统级上下文三个部

分。用户级上下文包括用户程序、用户数据、用户堆栈、共享内存等部分。寄存器上下文包括程序计数器、处理器状态寄存器、用户/系统堆栈指针、通用寄存器等。系统级上下文包括进程表入口、用户区域、进程区域表、系统堆栈等。

每个进程包括一个进程表（Process Table Entry），它由操作系统维护，包含了内核随时可以访问到的进程控制信息，因此始终保存在主存储器中。进程表的数据项包括进程标识符、父进程标识符、进程的用户标识符、当前进程状态、等待的事件、未处理信号、调度优先数、是否在内存的指示位、进程的大小、指向 U-area 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。

用户区域（U-area）包含了进程在运行时内核所要知道的附加进程控制信息。用户区域的数据项包括进程表指针、用户标识符、在用户模式和系统模式下运行的时间、信号处理数组、登录的终端、出错记录、系统调用返回值、I/O 参数、文件参数、文件句柄表、进程大小和写文件大小的限制、进程所创建文件的属性设置。

进程区域表（Per Process Region Table）由存储管理系统使用，它定义了实在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。

系统堆栈（Kernel Stack）是进程在内核模式下运行时使用的，用来保存过程调用和中断访问时用到的地址和参数。

3、Unix SVR4的进程创建

Unix SVR4 通过系统调用 fork()来创建一个进程，当进程调用 fork()后，操作系统执行下面的操作：

- 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，如何来区别父子进程呢?事实上，执行 fork()调用后，子进程的返回值是 0，而父进程的返回值不为 0，这样就可以通过程序控制流在父子进程中各自执行需要的操作。

3.4 线程的基本概念

3.4.1 引入多线程技术的必要性

在传统的操作系统中，进程是系统进行资源分配的单位，诸如按进程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资源，I/O CH、I/O DV 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进程（Single Threaded Process）。这种单线程结构的进程已不能适应当今计算机技术的迅猛发展。

下面从并行技术的发展、网络技术的发展、软件技术的发展和并发程序设计效率多个侧面来论述在操作系统中改造单线程结构进程和引入多线程机制的必要性。

1、并行技术的发展

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。

目前 cluster 和 SMP 得到了广泛的应用，成为并行处理技术的热点。值得注意的是，单线程结构进程从管理、通信和并发粒度多个方面都很难满足并行处理的要求。

2、网络技术的发展

近二十年以来，计算机网络技术得到了迅猛的发展，这里不考虑网络基础通信设施的进展，单从网络操作系统和分布式操作系统的角度来考虑，单线程结构进程就难以满足要求。

分布式操作系统是一个无主从的、透明的资源管理系统，也是一个松散耦合的 MIMD 系统。从资源管理看，每一类资源的管理都有可能分布在各个独立的节点上，需要广泛的协同和频繁的通信，从分布并行看，要求操作系统合理使用网络上的可计算资源，更好地发挥多处理器的能力，把许多任务或同一任务的不同子任务分配到网络不同结点的处理器上去同时运行，所有这一些都需要改良单线程结构进程，提高各个协作子任务之间的协作、切换和通信效率。

客户/服务器计算是九十年代网络计算的最大热点，对于基于服务器的客户/服务器计算（如 SQL 服务器），要求采用更加高效的并行或并发解决方案来提高服务器处理效率；对于基于客户的客户/服务器计算（如页服务器），虽然服务器的瓶颈效应大大改善，但服务器一方除了页面读写外还需要进行锁管理和锁协商，客户一方则面临着如何快速解决计算逻辑问题，另外还要考虑锁协商和页回调，这些也要求一种更加实用和有效的并行或并发解决方案；类似的问题同样存在与协作式的客户/服务器计算和基于中间件的客户/服务器计算中。

3、软件技术的发展

从软件技术的发展来看，系统软件和应用软件均有了很大的进展，要求人们设计出多事件并行处理的软件系统，如：操作系统中的并行文件操作，数据库中的多用户事务处理，窗口子系统中的多个相关子窗口操作，实时系统中多个外部事件的同时响应，网络中多个客户共享网络服务器任务等等。这些多事件并行处理的软件系统当然要求提高并行或并发处理的效率，而单线程结构进程对此是无能为力的。

4、并发程序设计的要求

在传统的操作系统中，往往采用多进程并发程序设计来解决并行技术、网络技术和软件技术发展带来的要求，即创建并执行多个进程，按一定策略来调度和执行各个进程，以最大限度地利用计算机系统中的各种各样的资源。这一方式当然是可行的，但关键在于并行和并发的效率问题，采用这一方式来实现复杂的并发系统时，会出现以下的缺点：

- 进程切换的开销大，频繁的进程调度将耗费大量时间。
- 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过程，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很难体现效率。

综上所述，要求操作系统改进进程结构，提供新的机制，使得很多应用能够按照需求，在同一进程中设计出多条的控制流，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）。

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 2.x、Mach 2.6、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。

3.4.2 多线程环境中的进程与线程

本节讨论在多线程环境中进程与线程的概念。

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，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。进程运行时，处理器的寄存器由进程控制，而进程不运行时，这些寄存器的内容将被保护。所以，进程和进程之间的关系比较疏远，相对独立，进程管理的开销大，进程间通信效率低效。单线程进程的内存布局如图 3-16 所示：

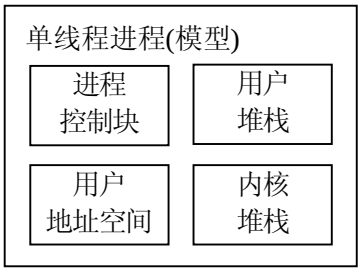


图 3-16　单线程进程的内存布局

使用单线程进程进行并发程序设计称为并发多进程程序设计，采用这种方式时，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。为解决这一问题，我们来研究一下进程的运行，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，如图 3-17 所示：

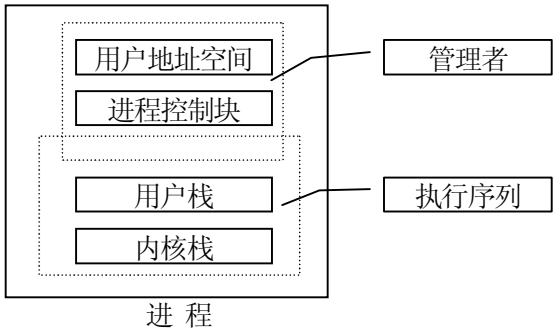


图 3-17　单线程进程的运行

设想是否可以把进程的管理和执行相分离，如图 3-18 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有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。

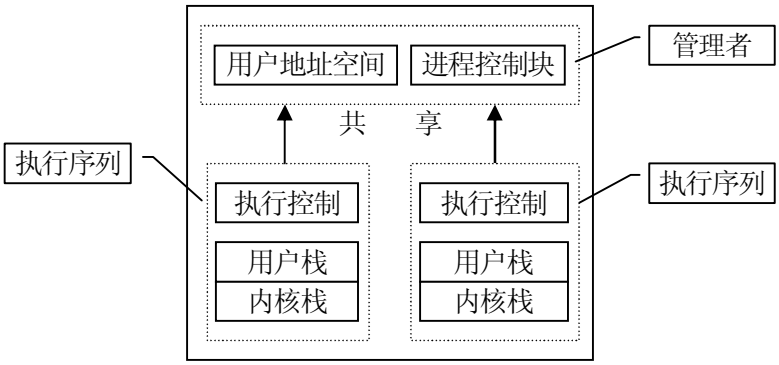


图 3-18　管理和执行相分离的进程模型

多线程进程的内存布局如图 3-19 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，而每个线程则存在独立堆栈，以及包含寄存器信息、优先级、其它有关状态信息的线程控制块。线程之间的关系较为密切，一个进程中的所有线程共享其拥有的状态和资源。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。

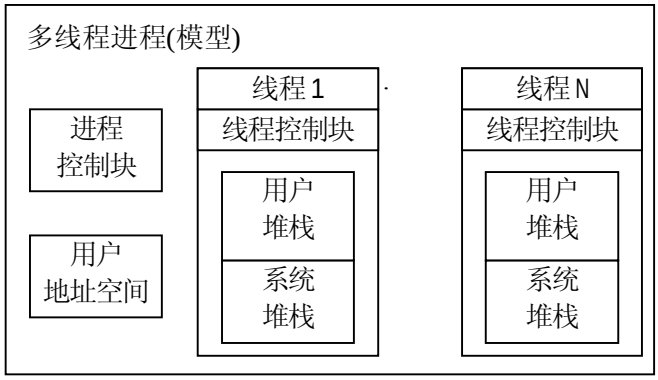


图 3-19　多线程进程的内存布局

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 CPU、进程、文件和资源(CH、DV)等的存取保护机制。

2、多线程环境中的线程概念

线程则是指进程中的一条执行路径（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。线程具有：

- 一个线程执行状态（运行、就绪、…）

- 有一个受保护的线程上下文，当线程不运行时，用于存储现场信息
- 一个独立的程序指令计数器
- 一个执行堆栈
- 一个容纳局部变量的静态存储器

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或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。
- 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。

线程的内存布局如图 3-20 所示，多线程进程的组成包括线程和空间。空间是完成一个程序的运行所需占有和管理的内存空间，它封装了对进程的管理，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。线程封装了并发（concurrency），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。线程主动地访问空间。

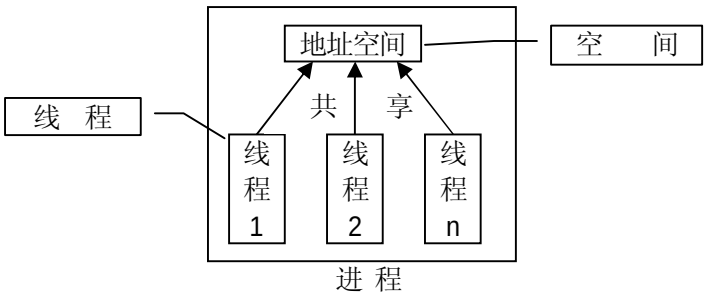


图 3-20　线程的内存布局

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。

当处于运行状态的线程阻塞时，对于某些线程实现机制，所在进程也转换为阻塞状态，即使这个进程存在另外一个处于就绪状态的线程；对于另一些线程实现机制，如果存在另外一个处于就绪状态的线程，则调度该线程处于运行状态，否则进程才转换为阻塞状态。

与线程状态变化有关的线程操作有 4 个：

- 孵化（Spawn）：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创建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）：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）。处理器现在就可以转向执行其它就绪线程。
- 活化（Unblock）：当被阻塞线程等待的事件发生时，线程变成就绪态。

- 结束（Finish）：当一个线程完成时，便回收它的寄存器和堆栈。

4、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，而且必须调用内核功能，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，这对许多并发应用来说是不方便的。而对多线程技术来说，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，故此时，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。

5、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。

3.4.3 线程的实现

从实现的角度看，可以分成二类线程：用户级线程（ULT）(如，Java 和 Informix)和内核级线程（KLT）(如，OS/2 和 Macintosh)。后者可归结为内核支撑线程或轻量进程（LWP）。也有一些系统(如，Solaris)提供了混合式策略，同时支持两种线程实现。图 3-21 给出了各种线程实现方法。

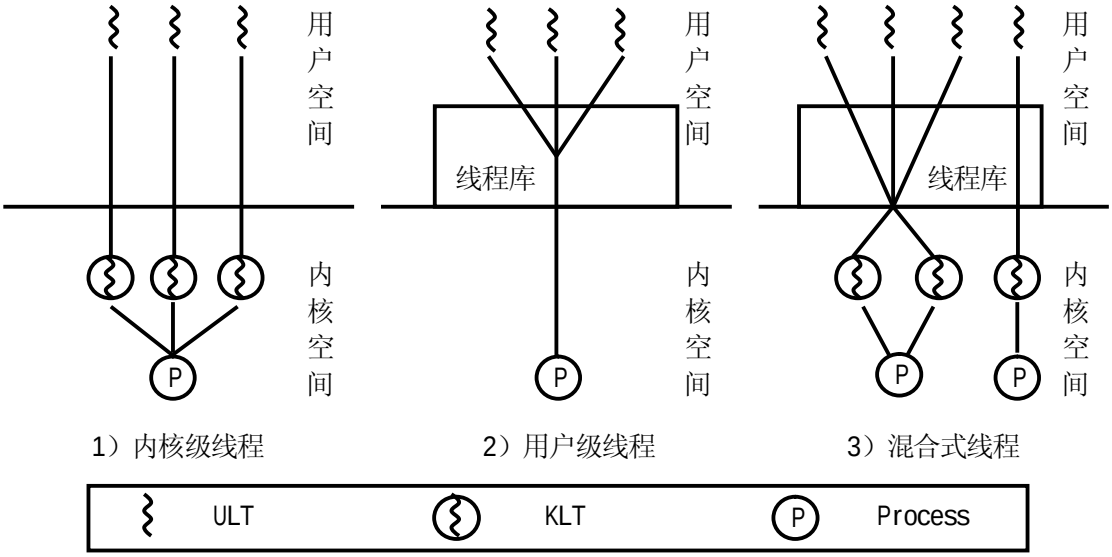


图 3-21　线程实现方法

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。ＫＬＴ专门提供了一个应用程序设计接口（API），供开发者使用，应用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，并且这个应用的所有线程均在一个进程中支持。内核要为整个进程及进程中的单个线程维护现场信息。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的优点，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；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。

这一方法还有一个显著的特点，内核例行程序本身也能被多线程化。

ＫＬＴ的主要缺点是：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内核的模式切换，开销较大。

2、用户级线程

纯 ULT 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/毁灭线程的代码、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状态（contexts）的代码。

一个应用程序开始运行时只包括一个线程，并且首先运行这个线程。这个应用和它的线程被分配到由内核管理的一个进程。当应用进程处于运行状态时，通过调用线程库中的“孵化实用程序”，应用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“孵化实用程序”， 线程库为新线程创建一个的数据结构，然后使用某种调度算法，把控制权则传送给进程中处于就绪态的一个线程。当控制权传送到库时，当前线程中的现场信息被保存，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、封锁...）。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 动作，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。

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。

上述两种情况中，当内核切换控制权返回到进程 B 时，执行便恢复执行线程 3。注意到当正在执行的线程库中的代码时，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，因而，进程并不要切换到内核方式来做线程管理。这就节省了二种方式切换的开销。
- 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，根据应用需要裁剪调度算法。
- 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。

和 KLT 比较，ULT 有二个明显的缺点：

- 在典型的操作系统中，大多数系统调用将被阻塞，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道程序设计的能力，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 实用程序，由 jacketing 程序来检查资源使用情况，以决定是否调用系统调用或传递控制权给另一个

线程。

3、混合式策略

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，线程创建是完全在用户空间做的，单应用的多个 ULT 映射成一些 KLT。程序员可按应用和机器调整 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二者优点，并舍去它们的缺点。

3.4.4 实例研究：JAVA 语言中的线程

1、JAVA语言线程控制的基本概念

为了进一步利用现代计算机硬件平台卓越的处理能力，JAVA 语言引进了原本在操作系统中才存在的线程概念。

对于现代应用软件来说，多线程并发程序设计时必须的。例如，一个图形用户界面 GUI 应用，在后台进行计算（可能是真正的计算，也可能只是屏幕输出）的同时，，要对用户的输入作出反应。加入通过多进程并发程序设计来完成该功能，需要多次中断后台计算，频繁在两个进程之间切换，这样做开销大且笨拙危险，因为后台计算可能不允许中断。有了线程的支持，这个应用就可以分成两个控制线程：处理 GUI 的用户事件线程和进行后台计算的线程，两者既可以在一个 CPU 上并发执行，大大降低线程切换的开销；又可以在两个处理器上并行执行。

JAVA 语言提供一个多线程系统，在这个多线程系统中，每个线程均被赋予一个优先级，优先级决定了线程获得 CPU 被调度执行的优先程度。优先级高的线程可在更短的时间段内获得更多的执行时间，优先级低的线程则正好相反。线理的优先级如果完全相等，将被依据“先来先服务”的原则进行调度.

线程要能够运行，必须通过一个“调度程序”（schduler〉来进行调度。调度实际上指的就是分配处理器资源．多线程系统中，处理器资源是按时间片分配的，获得处理器资源的线程只能在规定的时间片内执行，一旦时间片使用完毕，就必须把处理器让给另一处于等待状态的线程。

Java 支持一种“抢占式”（preemptive）调度方式。因此，为使低优先级线程能够有机会运行，较高优先级线程可以不时进入“睡眠”（sleep）状态。进入睡眠状态的线程必须在被唤醒之后才能继续运行。

Java 的运行时环境中，线程是从相同内存空间开始运行的，它们共享类变量和方法的访问权。这就产生了同步问题。同步的基本思想是避免多个线程访问同一个资源。Java 是纯面向对象的语言，它的资源都是以对象的形式表现的。因此，Java 的同步机制的作用就是力图避免对“对象”的访问冲突．Java 语言中，同步的实现在很大程度上取决于编程人员，编程人员可决定一个类中哪些方法或代码段需同步执行，Java 提供了一种类似信号量的东西，即 monitor 来控制对象的访问同步。同步执行的代码段要访问某个共享对象．必须拥有该对象的 monitor。

线程组用以管理线程。每个线程均属于某一线程组，一个线程组可包含多个线程或其他线程组，由此而形成线程间的一种层次关系。一个线程可访问其自身的线程组，但无法访问该线程组上面的线程组。线程组的相关操作被封装在类 ThreadGroup 中。

2、Java的线程状态

Java 的线程从产生到消失，可分 5 个状态，如图 3-22 所示：

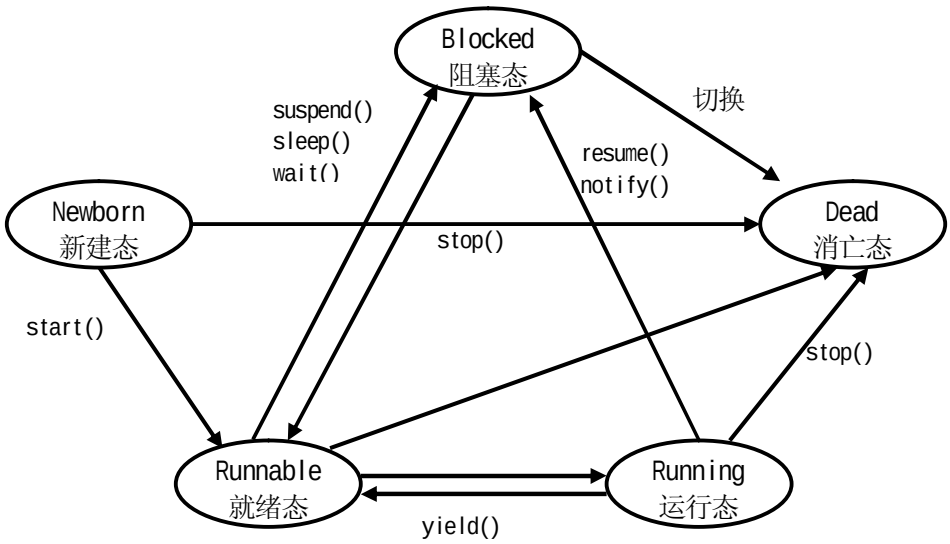


图 3-22　Java 的线程状态

1）Newborn

线程在已被创建但尚未执行这段时间内，处于一个特殊的新建状态。这时，线程对象已被分配内存空间，其私有数据已被初始化，但该线程还未被调度。此时线程对象可通过 start（）方法调度，或者利用 stop（）方法杀死。新创建的线程一旦被调度，就将切换到就绪状态.

2）Runnable

即线程的就绪状态，表示线程正等待处理器资源，随时可被调用执行。处于就绪状态的线程事实上已被调度，也就是说，它们已经被放到某一队列等待执行。处于就绪状态的线程何时可真正执行，取决于线程优先级以及队列的当前状况。线程的优先级如果相同，将遵循“先来先服务”的调度原则。

线程依据自身优先级进入等待对列的相应位置。某些系统线程具有最高优先级、这些最高优先级线程一旦进入就绪状态，将抢占当前正在执行的线程的处理器资源，当前线程只能重新在等待队列寻找自己的位置。某些具有最高优先级的线程执行完自己的任务之后，将睡眠一段时间，等待被某一事件唤醒。一旦被唤醒，这些线程就又开始抢占处理器资源。这些最高优先级线程通常被用来执行一些关键性任务，如屏幕显示。

低优先级线程需等待更长的时间才能有机会运行。由于系统本身无法中止高优先级线程的执行，因此，如果你的程序中用到了优先级校高的线程对象，那么最好不时让这些线程放弃对处理器资源的控制权，以使其他线程能够有机会运行。

3）Running

即线程的运行状态。该线程已经拥有了对处理器的控制权，其代码目前正在运行．这个线程将一直运行直到运行完毕，除非运行过程的控制权被一优先级更高的线程强占。

综合起来，线程在如下 3 种情形之下将释放对处理器的控制权：

主动或被动地释放对处理器资源的控制权。这时，该线程必须再次进入等待队列，等待其他优先级高或相等线程执行完毕。

睡眠一段确定的时间，不进入等待队列。这段确定的时间到期后，重新开始运行。等持某一事件唤醒自己。

4）Blocked

即线程的阻塞状态。如果一个线程处于堵塞状态，那么暂时这个线程将无法进入就绪队列。处于堵塞状态的线程通常必须由某些事件才能唤醒．至于是何种事件，则取决于堵塞发生的原因，处于睡眠中的线程必须被堵塞一段固定的时间；被挂起、或处于消息等待状态的线程则必须由一外来事件唤醒。

5）Dead

即线程的消亡状态。表示线程已退出运行状态，并且不再进入就绪队列。其中原因可能是线程已执行完毕〈正常结束〉，也可能是该线程被另一线程所强行中断（kill）。

3、Java的多线程实现机制

Java 的多线程主要体现在两个方面：系统级和语言缀．本节主要阐述 Java 在语言级对多线程的支持，其中将具体介绍两个重要的类：Thread 和 Runnable。Java 实现多线程的两种方式分别：

- 创建类 Thread 的子类
- 根据 Runnable 接口创建一个类

1）创建类 Thread 的子类

Java 用于实现多线程的方法之一是创建类 Thread 的子类，并重载 run（）方法。下面我们通过一实例来阐述这种方式的具体实现过程。

在我们要介绍的这个例子中，我们将在后台计算质数，并把计算结果显示于屏幕。首先我们给出完整的源程序清单：

［源程序清单，runPrime.java］

```
class primeThread extends Thread
{
    public void run()
    {
    int number = 3;
        int mod = 0;
    boolean flag = true
while (true) {
            loop: for (int i = 0; i < number; i++) {
                if ((mod = number % i) == 0) {
                    flag = false;
                    break loop;
                }
            }
            if (flag) {
                System.out.println(number);
            flag = true;
            }
            number++;
```

```
                flag = true;
                try { sleep(5); } catch(InterruptedException e) { return; }
            }
        }
    }
    class runPrimes
    {
        public static void main(String args[])
        {
            primeThread getPrimes = new primeThread();
            primeThread.start();
            while (primeThread.isAlive()) {
                System.out.println("Counting the prime number ... \n");
                try { Thread.sleep(5); } catch(InterruptedException e) {return;}
            }
        }
    }
```

这个程序用到了两个类 primeThread 和 runPrimes。第 2 个类用于调用第 1 个类，它的作用是实现一个 main()方法。这里起主要作用的是类 primeThread。

类 primeThread 是类 Thread 的一个子类。随后我们重载了 Thread 类的 run()方法，在方法中设置了一个无穷循环，不断寻找质数，并打印输出找到的质数。

在类 runPrimes 中，我们首先创建了类 primeThread 的一个实例 getPrimes。这个线程实例一旦被创建，马上进入 Newborn 状态。这个新线程被赋予和主线程同等的优先级。随后我们利用了一个 start()方法．这个方法由 Thread 继承而来，作用是把线程状态由 Newborn 转换成 Runnable。start()方法将调用线程的 run 方法()。线程一旦被启动.程序将马上返回到调用线程一一main()方法，不断显示系统正在运行的提示，直至出现一个异常情况。

这时，程序一共有了两个线程，第一个是我们从命令行启动的线程，这是系统创建的主线程，第二个则是计算质数的线程，等待被调用执行。

接下来程序进入一个循环，只要对象 primeThread 的方法 isAlived()返回 true，循环就将一直执行。isAlive()是类 java.lang.Thread 的成员方法，只要线程对象的 run()方法仍在运行，isAlive()就将返回 true，否则返回 false。当线程仍然处于执行状态时，屏幕上将不断显示出“Counting the prime number...”字样。

这段程序中还必须注意两点：首先是线程方法 Thread.sleep(5)的调用；其次是异常情况 InterruptedException 的捕捉。方法 Thread.sleep()用于使当前正处于运行状态的线程睡眠一段确定的时间，该线程将暂时进入堵塞状态。就绪队列中的下一个线程将获得对处理器的控制权。由于程序的两个线程处于同等优先级，因此，为使两个线程均能有机会运行，我们必须让当前正处运行状态的钱程主动让出对处理器的控制权。这其实有两种方法都可以达到我们的目的：一是调用 sleep()方法，其次是调用线程类的静态方法 yield()。yield()方法直接把处理器资源让给下一个等待线程，sleep()则可以指定线程被堵塞的时间片，这段时间片结束之后，线程才能恢复“就绪”状态，进入就绪队列。或许你已经注意到，线程类 primeThread 中，我们也调用了一个 sleep〈）方

法，这主要是为了让主线程随后可以有机会再次运行。InterruptedException 则是当一个线程被另一线程中断时，必定被抛出的一个异常情况。sleep()方法必将导致这个异常情况。同时这个异常情况又不属于运行时异常倩况，故必须设置一异常情况句柄，对其进行捕捉。否则．Java 编译器将给出一警告错误。

2）实现 Runnable 接口

创建线程的另一方式是通过实现一个 Runnable 接口，以这种方式创建的类可通过把类实例传递给一线程对象而在一线程体内运行。采用这种方式实现的前面求质数的程序如下所示：

```
[源程序清单，runPrime2.java]
class primeRunnable extends Runnable
{
    public void run()
    {
    int number = 3;
        int mod = 0;
    boolean flag = true
while (true) {
            loop: for (int i = 0; i < number; i++) {
                if ((mod = number % i) == 0) {
                    flag = false;
                    break loop;
                }
            }
            if (flag) {
                System.out.println(number);
            flag = true;
            }
            number++;
            flag = true;
            try { sleep(5); } catch(InterruptedException e) { return; }
        }
    }
}
class runPrimes
{
    public static void main(String args[])
    {
        primeRunnable getPrimes = new primeRunnable();
        new Thread（getPrimes）.start();
        while (primeThread.isAlive()) {
            System.out.println("Counting the prime number ... \n");
```

```
                try { Thread.sleep(5); } catch(InterruptedException e) {return;}
            }
        }
    }
```

3）Java 的 Thread 类简介

Java 的 Thread 类构造函数如下：

public Thread() 以这种方式创建线程，必须重载新线程的 run()方法，在其中定义自己的操作
public Thread(String threadName) 同第一种方式，只不过这里同时指定了线程名
public Thread(Runnable targetObject) 以这种方式创建线程，run()方法从目标对象创建
public Thread(Runnable targetObject， String threadName) 同第三种方式，但这里同时指定了线程名

前面例子中我们曾使用过其中的两个构造函数，它们实际上表示了创建线程的两种方式。如果创建线程时未指定线程名，则可在以后用线程的 setName()方法指定。

Thread 类定义了 3 个与线程优先级有关的实例变量，如下所示：

- public final static int MIN_PRIORITY
- public final static int MAX_PRIORITY
- public final static int NORM_PRIORITY

MIN_PRIORITY 表示可能的最小优先级，也就是说，在为线钱程指定优先级时。所指定的优先级不得小于 MIN_PRIORITY。同理，MAXPRI_PRIORITY 表示可能的最大优先级，为线程指定的优先级不得大于此值。NORM_PRPRITY 表示线程优先级的缺省值。

前面例子中我们已经介绍了 Thread 的若干方法，如 start()和 run()等．这些方法可以说是线程对象最为重要的几个方法。除此之外，Thread 还提供了用于指定或获取线程名、确定线程状态、控制线程执行顺序等的多个方法。下面是真中的一些方法：

- currentThread(): 返回当前正处于运行状态的线程
- getName(): 获取线程名
- getPriority(): 获取线程优先级
- isAlive(): 确定线程是否处于活动状态
- isDaemon(): 确认线程是否为守护线程
- resume(): 恢复被挂起线程的执行
- setName(): 为线程指定一个名字
- SetPriority(): 设定线程优先级
- setDaemon():把线程设为守护线程
- Start(): 启动线程的执行，立即返回调用线程，如线程原先已被启动，抛出 InterruptedException 异常情况
- suspend(): 挂起线程
- sleep(int millseconds): 使线程睡眠指定的毫秒数.
- yield(): 退出运行状态，直接进入就绪队列。

4、Java的线程同步

这里我们简要介绍 Java 中同步的实现。

1）synchronized 关键字

Java 的同步是通过对需要同步的方法或代码段进行标记来实现的，标记的方法是利用 synchronized 关键字．下面代码行表示方法 testData()必须被同步:

```
synchronized int testData()
{
    ...             // 需要进行同步的代码
}
```

对于使用 synchronized 关键字声明的方法，系统将为其设置一特殊的内部标记，名为 monitor。这个标记起着信号量的作用，每当调用该方法时 Java 的运行时条统都将进行检查，确认此标记的状态。看看相应代码是否已被调用。如果没有，系统将把这个内部标记授予调用对象，方法可以运行。运行结束后，标记被释放。标记来被释放之前，任何其他对象通常不得调用此方法.

带 synchronized 关键字的代码段的运行机制稍有不同．这时代码段的声明方式为，

```
synchronized(locked_object)
{
    ...             // 需要进行同步的代码
}
```

此时，所指定的对象将拥有 monitor 标记。要调用这段代码，必须获取此对象的 monitor 标记。一旦我们进入这段代码，在我们退出该代码段并释放 monitor 之前，其他线程就将不得进入。

2）wait()和 notify()

进入带有 synchronized 修饰的方法或代码段之后。有时我们需要暂时释放 monitor 标记，以使其他线程能够调用相应代码．这可以利用 wait()方法加以实现．调用 wait()方法，既可以使线程处于无条件等待状态，也可以使之等待一段确定的时间．这时线程将帮放 monitor 标记。

线程进入等待状态之后，其他线程可通过调用相应对象的 notify()方法来使之恢复执行，重新获得 monitor 标记。这时，线程将从 wait()语句的下一条语句开始执行。

下面我们给出对象 locked object 用以使线程处于等待或用于唤醒线程的一些方法：

`public final void wait(long millseconds)throws InterruptedException` 使线程进入等待状态，这时线程将等待给定的毫秒数，除非被其他线程所唤醒。

`public final void wait(long millseconds, int nanoseconds)`
`throws InterruptedException`
使线程进入等待状态，这时线程将等待确定的时间，除非被其他线程所唤醒。

`public final void wait() throws InterruptedException` 使线程进入等待状态，此时线程将一直处于等待状态，直至被其他线程所唤醒。
`public final void notify()` 唤醒一个同一对象内处于等待状态的线程.
`public final void notifyAll()` 唤醒同一对象内处于等待状态的所有线程

3.5 实例研究：Solaris 的进程与线程

3.5.1 Solaris 中的进程与线程概念

Solaris 采用了 4 个分离的与进程和线程有关的概念：

- 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。
- 轻量级进程（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。
- 内核线程（Kernel Threads）：即 KLT，它们时能被调度和指派到系统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，特别适用于解决那些逻辑并行性问题；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 3-23 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。

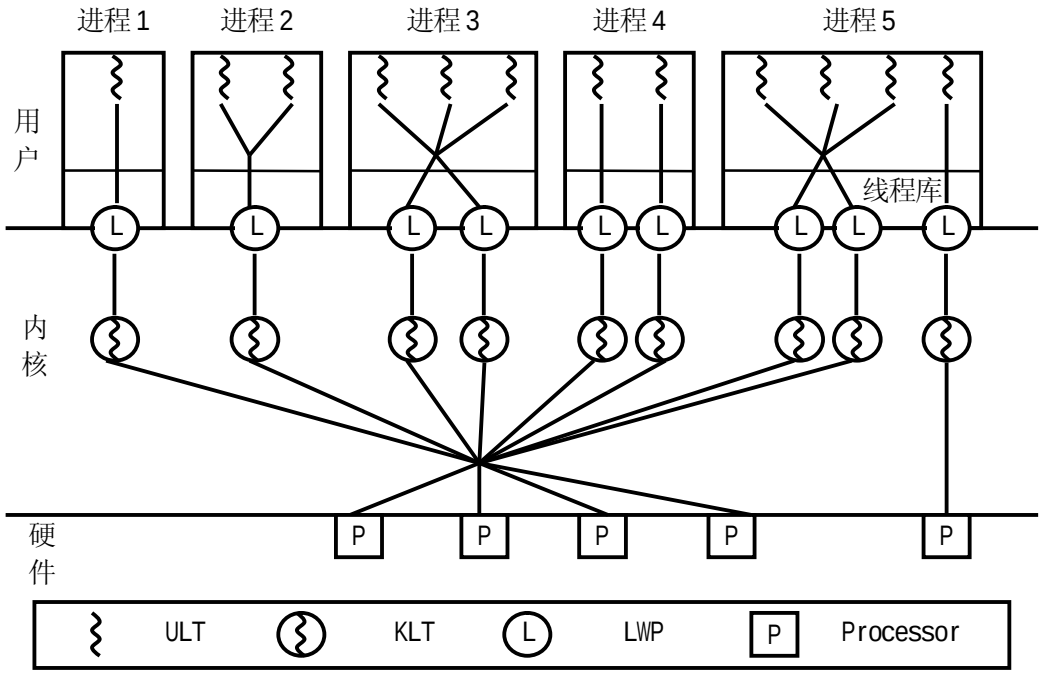


图 3-23　Solaris 线程的使用

3.5.2　Solaris 的进程结构

Solaris 中，进程包括多个轻进程，进程结构如图 3-24 所示：

LWP 的数据结构包括：

- 轻进程标识符标识了轻进程
- 优先数定义了轻进程执行的优先数
- 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进程让出处理器时的现场信息
- 轻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

用户级线程的数据结构包括：

- 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

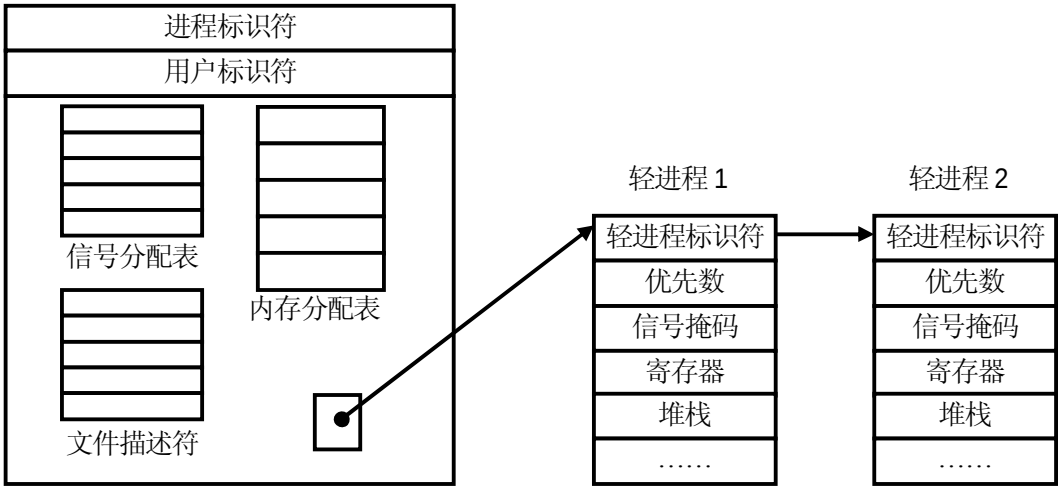


图 3-24　Solaris 的进程结构

3.5.3　Solaris 的线程状态

图 3-25 简单说明了 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 是目前分配到 LWP 上，并且在对应内核线程执行时，它被执行的。出现事件会导致 ULT 离开活跃态，让我们考虑一个活跃的 ULT，名为 T1，下面的事件可能发生：

- 同步：T1 调用了一条并发原语与其它线程协调活动。T1 进入睡眠态，当同步条件满足后，再把 T1 放入可运行队列。
- 挂起：任何线程（包括 T1）可导致 T1 挂起并进入停止状态，T1 保止停止态直到别的线程发出一个继续请求，再把 T1 移入可运行队列。
- 剥夺：一个活跃线程（T1 或其它）执行了引起另一个更高优先级的线程 T2 变为可运行状态，如果处于活跃状态的线程 T1 优先级较低，它就要被剥夺并放入可运行态，而 T2 被分配到可用的 LWP。
- 让位：若 T1 执行了 thr_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（T2），它与 T1 具有相同的优先级，如果是这样，把 T1 放入可运行队列，将 T2 被分配到可用 LWP 上，否则 T1 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 3-25 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，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。

绑定了线程之后，ULT 和 LWP 的关系只有轻微的不同。例如，当一个因 ULT 等待一个事件而进入睡眠状态之后，相应的 LWP 也停止运行。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。

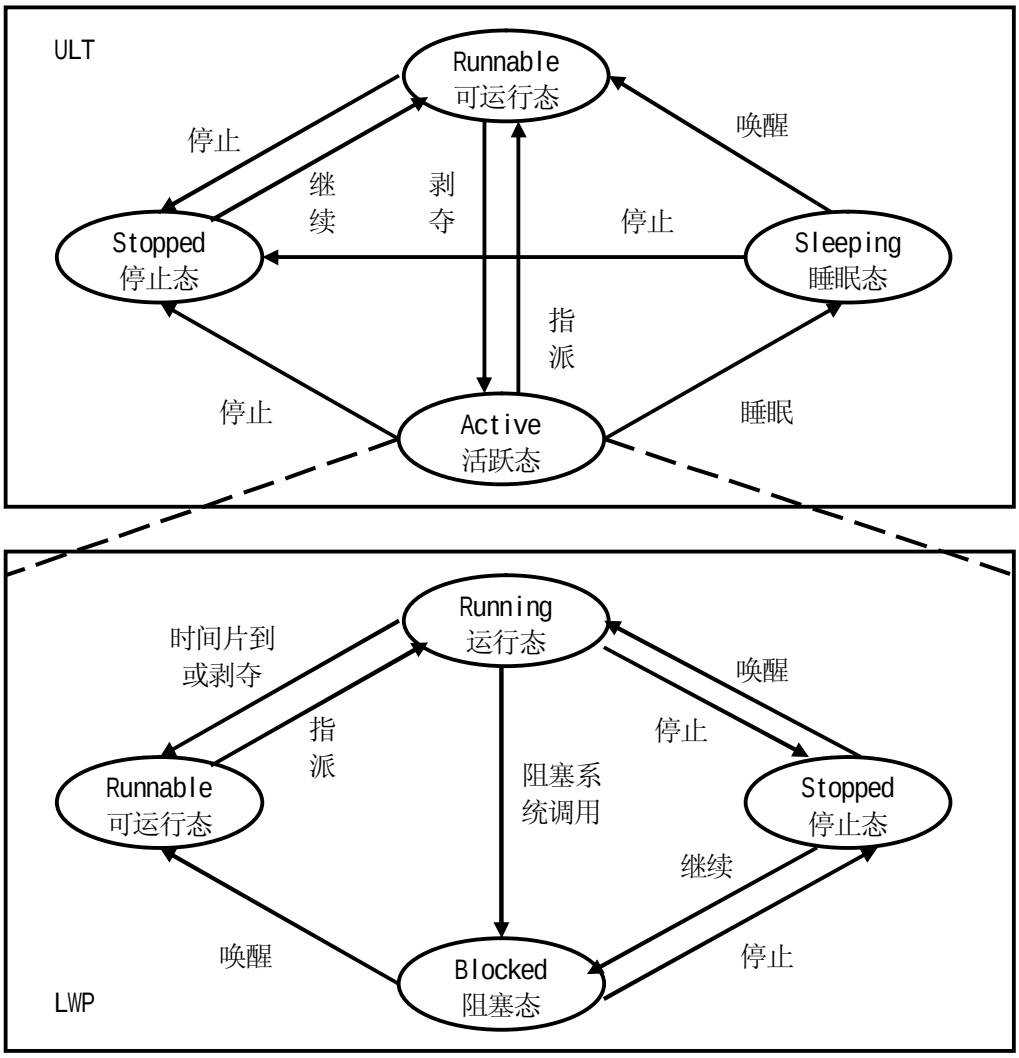


图 3-25　ULT 和 LWP 的状态

3.5.4　Solaris 的线程程序设计接口

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

3.6 实例研究：Windows 2000 的进程与线程

3.6.1 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的调度单位，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。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 3-26 所示：

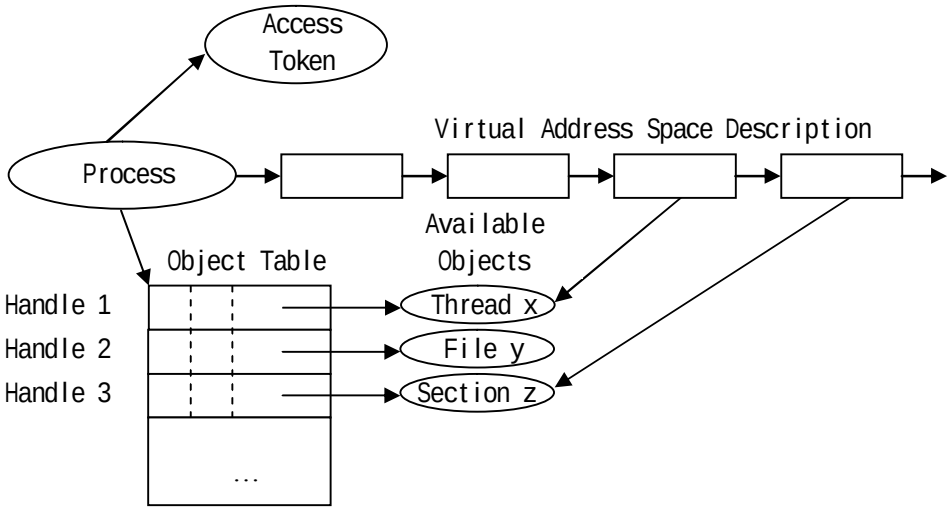


图 3-26 进程以及控制和使用的资源

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能够改变自己的属性，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的块，进程不能直接修改它，必须依靠为进程提供内存分配服务的虚存管理程序。

进程区包括一个对象表，其中包括了这个进程的其它对象的句柄。上图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。

3.6.2 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。图 3-27 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。

内核进程块
进程标识符
父进程标识符
退出状态
创建和退出次数
指向下一个进程的指引元
配额块
内存管理信息
异常端口
调试程序端口
主访问令牌指引元
局柄表指引元
进程环境块
映像文件名
映像基址
进程特权级
WIN32 进程块指引元

图 3-27　EPROCESS 块的结构

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

3.6.3 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。

每一个进程都由一个执行体线程（ETHREAD）块表示。图 3-28 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。

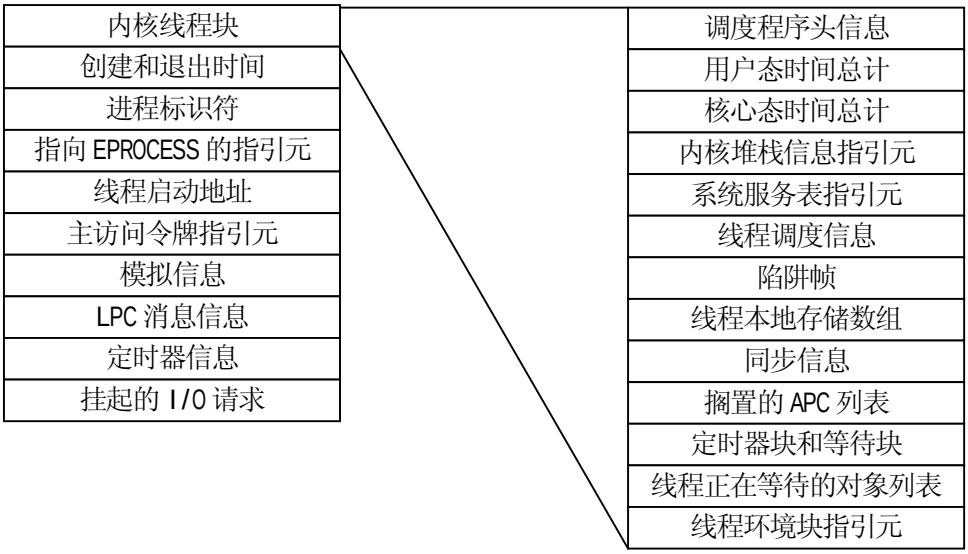


图 3-28　ETHRED 块的结构

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。

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 3-29 所示，它可能处于以下 6 个状态之一：

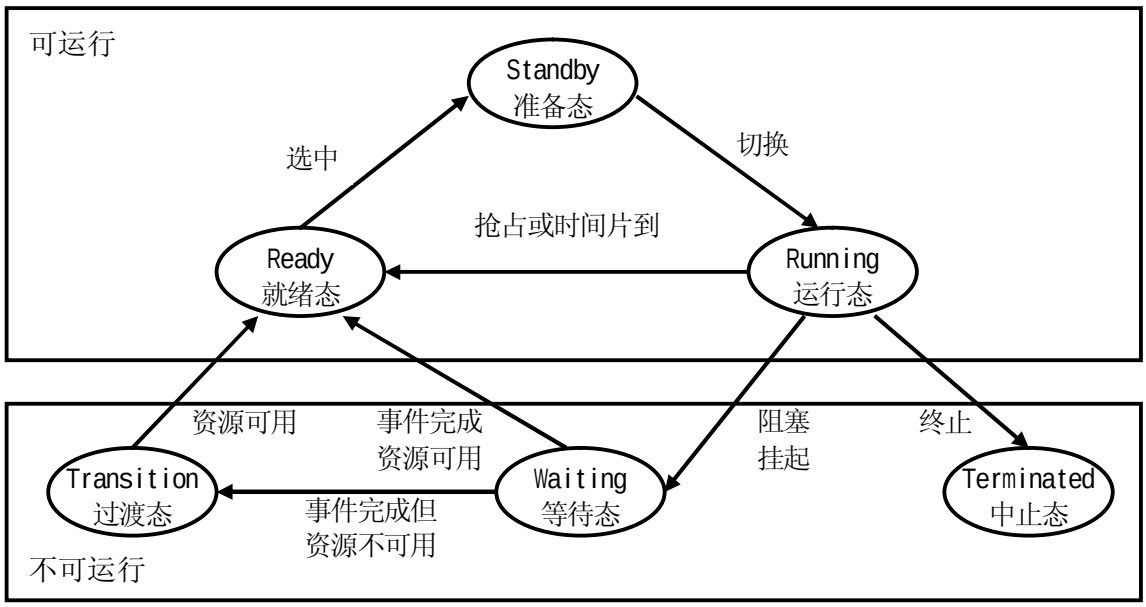


图 3-29　Windows2000 的线程状态

- 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。

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。

3.6.4 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。

CH4　处理机调度

4.1　处理机调度的类型

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。不同的操作系统采用的调度算法各不相同，常用处理机调度的层次作分类，下面讨论处理机调度。

4.1.1　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 4-1 所示，处理机调度可以分为以下三个级别：

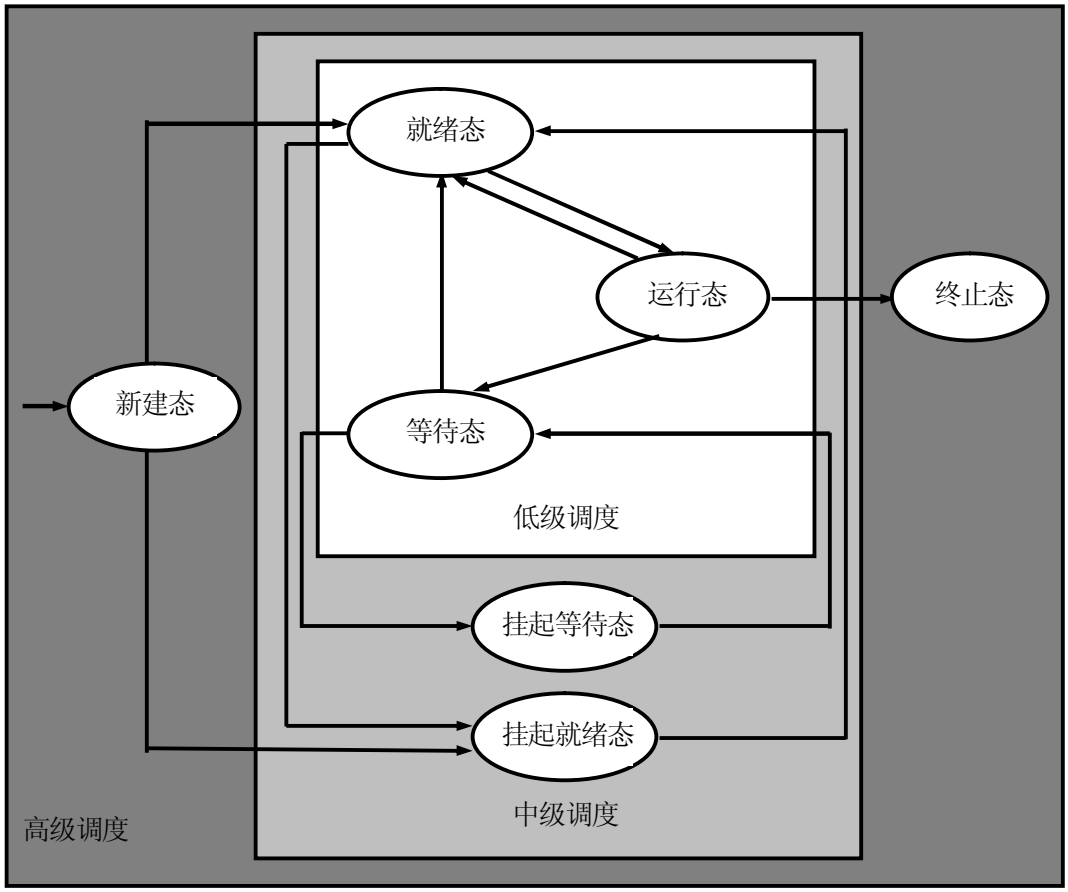


图 4-1　调度的层次

- 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能获得处理器，并将处理机出让给它进行工作。

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 4-2 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。

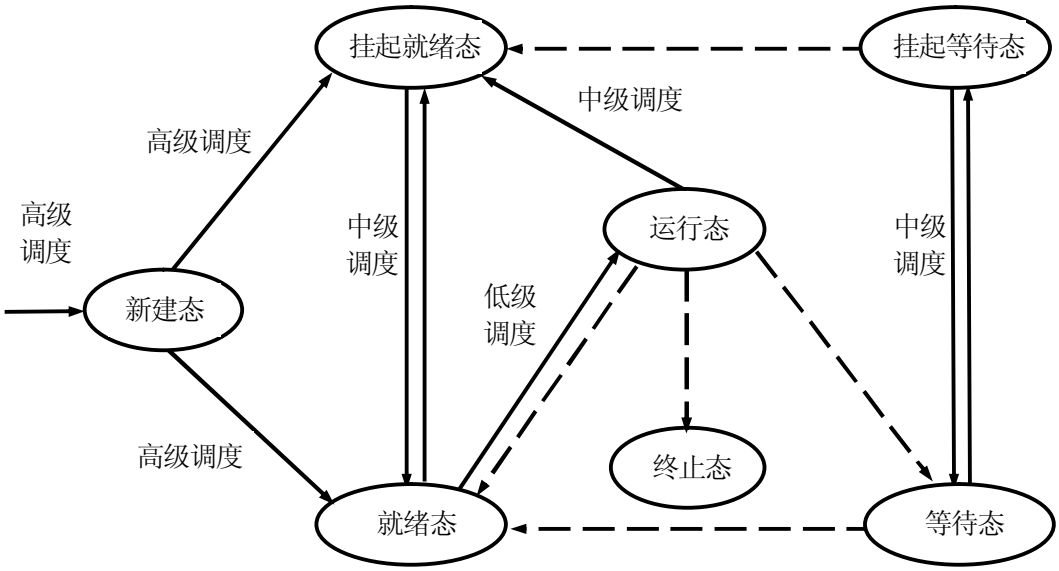


图 4-2 处理器调度与进程状态转换

4.1.2 高级调度

对于分时操作系统来说，如图 4-2 所时，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。

在多道批处理操作系统中，高级调度又称为作业调度。作业是用户要求计算机系统完成的一项相对独立的工作，它包括若干个作业步，每个作业步又可以转化为一个或几个可执行的进程。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 4-3 给出了批处理操作系统的调度模型。关于批处理系统作业调度的详细情况参见本章第二节。

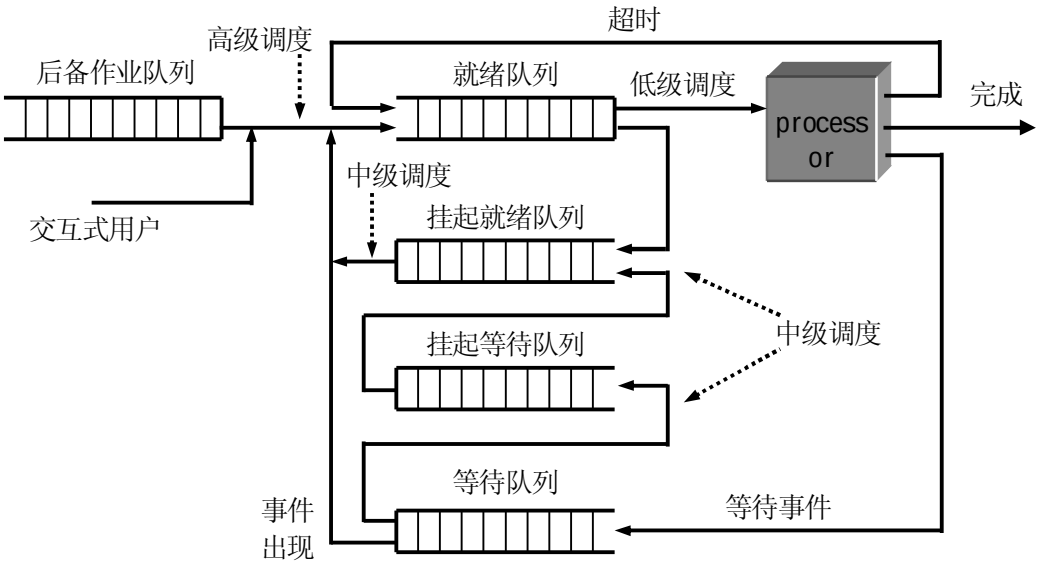


图 4-3　批处理操作系统的调度模型

4.1.3　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，以平顺系统的操作。有关中级调度的详细内容参见第三章中有关“进程的挂起”的小节。

4.1.4　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其分配给就绪进程。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。有关低级调度的详细情况参见本章第三节。

4.1.5　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小，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的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 ts，完成时刻是 tf，那么该作业的周转时间 ti 为：

$$ti = tf - ts$$

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。

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

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

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

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。

4.2 批处理作业的管理与调度

4.2.1 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，它提供一个作业说明语言，用户使用作业说明语言书写作业说明书来规定如何控制作业的执行。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）。一旦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：

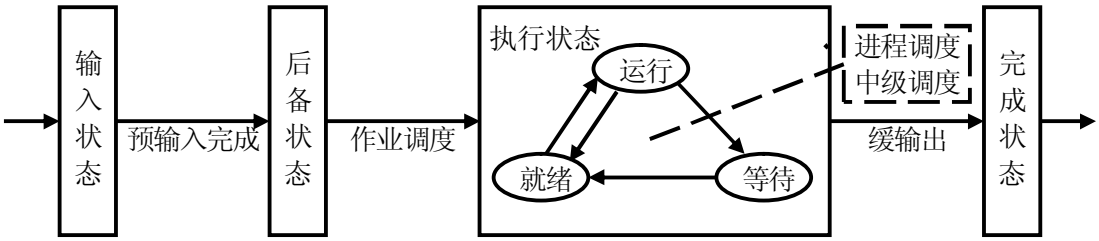
- 输入状态：此时作业的信息正在从输入设备上预输入。
- 后备状态：此时作业预输入结束但尚未被选中执行。
- 执行状态：作业已经被选中并构成进程去竞争处理器资源以获得运行。
- 完成状态：作业已经运行结束，正在等待缓输出。

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了一个用户进程。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 4-4。

图 4-4　作业调度与进程调度的关系



4.2.2　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 Ti 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 Ti 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=(T1+T2+…+Tn)/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。

4.2.3　作业调度算法

1、先来先服务算法

先来先服务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、最短作业优先算法

这种算法是以进入系统的作业所提出的计算时间为标准，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。

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问，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，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

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，后面将会给读者介绍 Unix 的动态优先数法调度。

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。

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。

4.3 进程调度

4.3.1 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。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。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。

4.3.2 进程调度算法

处理器的调度策略很多；现介绍如下几种：

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、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转换工作。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。—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，时间片长度固定，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系统效率和响应时间两方面考虑。

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，这祥有利于用户；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。

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。

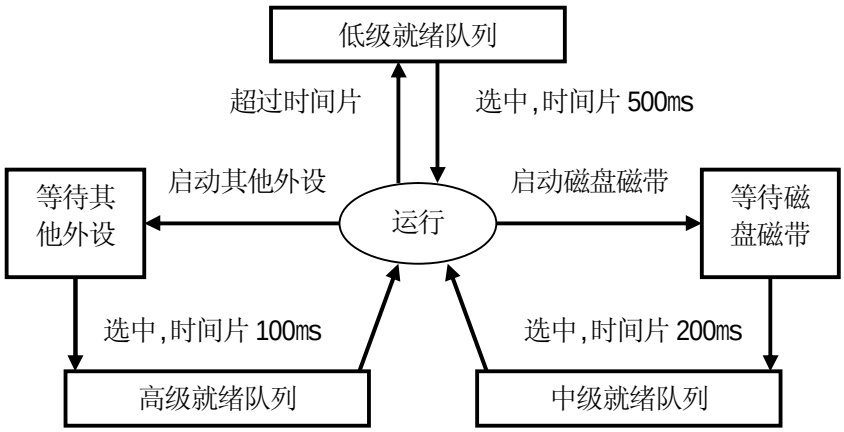


图 4-5　一个三级调度策略

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 4-5 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。

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：如你工作时有 n 个用户的登录，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。

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。

4.3.3 实时调度

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、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。

4.3.4 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。

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、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。

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）均分负载调度算法

均分负载（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。

4.3.5 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

4.3.6 实例研究——Unix SVR4 调度算法

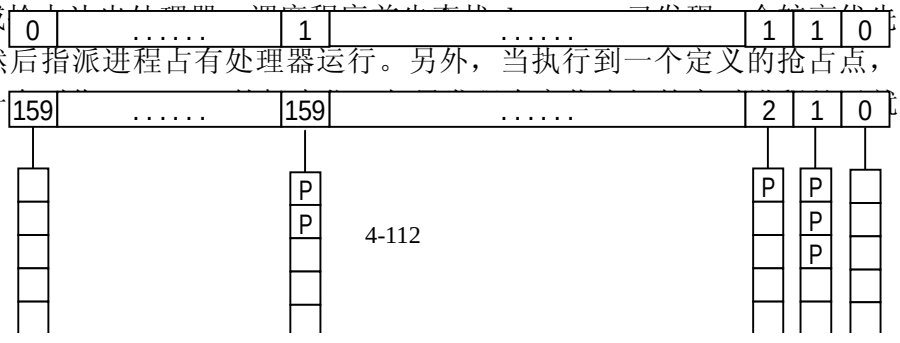
Unix SVR4 的调度算法同传统 Unix 相比有了较大变动，其设计目的是优先考虑实时进程，次优先考虑内核模式进程，最后考虑用户模式进程。Unix SVR4 对调度算法的主要修改包括：

- 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 4-6，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、时间片或抢占而放弃处理器，调度程序首先查找 dqactmap 以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，内核程序将查看 dispq 查一



态，就执行抢占。

图 4-6　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。

4.3.7　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。

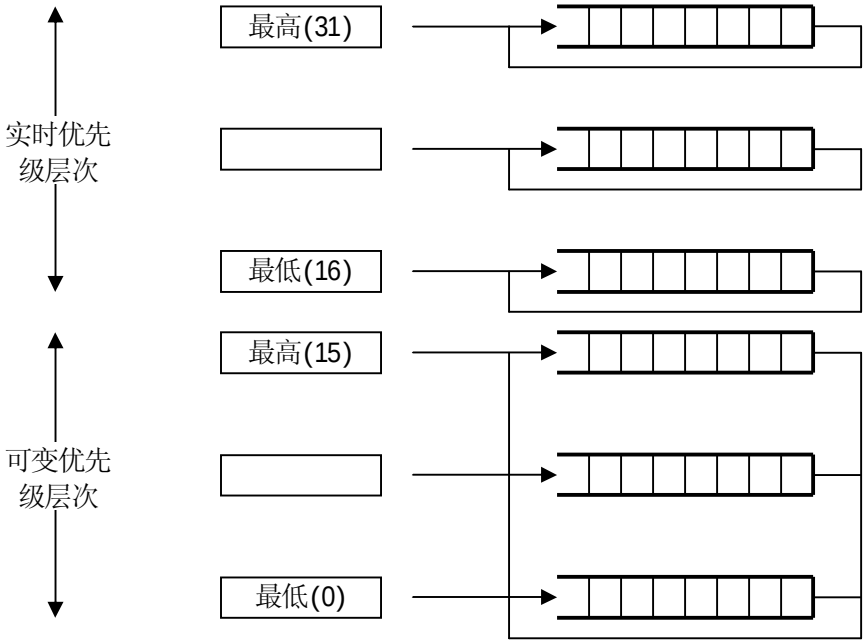


图 4-7　Windows NT 的线程优先级

如图 4-7 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。

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。

CH5　　并发进程

5.1　并发进程

5.1.1　顺序性和并发性

在第三章中我们已经引入了进程这个概念，本章将对进程的属性和管理作进一步的讨论。

一个进程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。一组进程的并发性是指这些进程的执行在时间上是重叠的，把这些可交叉执行的进程称为并发进程。所谓进程的执行在时间上是重叠的，是指一个进程的第一个操作是在另一个进程的最后一个操作完成之前开始。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，进程久：和 B 的执行顺序分别为 al，a2，a3 和 bl，b2，b3，这是进程的顺序性。然而，如果这两个进程在单处理器上交叉执行，如执行序列为 al，bl，a2，b2，a3，b3 或 a1，bl，a2，b2，b3，a3 等，则说进程 A 和进程 B 是并发的。

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) U R(p2)&W(p1) U 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。

5.1.2　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。

5.1.3 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的硬设备、变量、表格、链表、文件等资源时，导致处理出错。

由于进程间相互竞争资源时并不交换信息，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式协作。

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。

5.2 临界区管理

5.2.1 互斥和临界区

例 l 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”， 共享变量代表的资源叫“临界资源”。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是分散在各有关进程的程序中，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。

5.2.2 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。

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(insidd)置成 true 以封锁进程 P2(P1)进入临界区，直至 P1(P2)退出脑临界区，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 */
process P1
Begin
    while inside2 do [ ];
    inside1 := true;
    临界区;
    inside1 := false;
end;
process P2
begin
    while inside1 do [ ];
    inside2 = true;
    临界区;
    inside2 := false;
end;
```

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 */
process P1
begin
    inside1 := true;
    while inside2 do [ ];
    临界区;
    inside1 := false;
end;
process P2
begin
    inside2 := true;
    while inside1 do [ ];
    临界区;
    inside2 := false;
end;
```

5.2.3 实现临界区管理的软件方法

1、Dekker算法

Dekker 算法能保证进程互斥地进入临界区，用一个指针 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：

```
turn:integer;
turn := 1;
process P1
begin
    while turn = 2 do [ ];
    临界区;
    turn = 2;
end;
process P2
begin
```

```
    while turn = 1 do [ ];
    临界区;
    turn = 1;
end;
```

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。

但是，这种实现方法强制了两个进程以交替次序进入临界区。当 P1(P2)进入一次临界区后，一定让 P2(P1)去进入临界区。如果这时 r(P1)不想执行，则 Pl(P2)执行时也不可能再次进入临界 区，而必须等待 P2(P1)在临界区执行一次后才可以再次进入临界区。这种以交替次序进入临界区的管理方法使共享资源的使用效率不高。

2、Peterson算法

为了克服 Dekker 算法中对共享资源必须交替使用的限制，可采用 Peteron 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：

```
inside1,inside2:boolean;
turn:integer;
turn := 1;
inside1 := false;    /* P1 不在其临界区内 */
inside2 := false;    /* P2 不在其临界区内 */
process P1
begin
    inside1 := true;
    turn := 2;
    while (inside2 and turn=2)
         do begin end;
    临界区;
    inside1 := false;
end;
process P2
begin
    inside2 := true;
    turn := 1;
    while (inside1 and turn=1)
        do begin end;
    临界区;
    inside2 := false;
```

```
end;
```

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。

5.2.4 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。

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。

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。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、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;
```

5.3 信号量与 PV 操作

5.3.1 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对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，于是去睡觉。这样生产者迟早会添满缓冲区，然后去睡觉，形成了进程都永远睡觉。

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 P、V，管程和消息传递。

5.3.2 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赛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，而原语是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“发信号” 和“等信号” 的头字母，此外还用的符号有：wait 和 signal；up 和 down；sleep 和 wakeup 等)。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。

下面讨论整型信号量，这时 P 操作原语和 V 操作原语定义如下：

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。

- P(s)；当信号量 s 大于 0 时，把信号量 s 减去 l，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；  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。记录型信号量和 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 l，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;
          Q: 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);          /* 若信号量小于 0，则执行 P(s)的进程
                                           调用 W(s) 进行自我封锁，被置成等
                                           待信号量 s 的状态，进入信号量队列 Q*/

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);    /* 若信号量小于等于 0，则从信号量 s
                                           队列 Q 中释放一个等待信号量 s 的进程*/
end;
```

其中 W(s)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)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。信号量 s 的初值可定义为 0，l 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。

仅能取值 0 和 1 的信号量称二元信号量，可以证明它与一般的记录型信号量有同等的表达能力，作为练习，读者可以自行写出二元信号量上操作的 P 和 V 原语的定义。

5.3.3 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。同 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;
    s : semaphore;
    s := 1;

cobegin
process Pi
    var Xi:integer;
begin
    L1:
    按旅客定票要求找到A[j]；
    P(s)
    Xi := A[j];
    if Xi>=1
    then begin
       Xi:=Xi-1; A[j]:=Xi;
       V(s); 输出一张票；
    end;
    else begin
       V(s); 输出票已售完；
    end;
    goto L1;
 end;
coend;
```

```
Var A : ARRAY[1..m] OF integer;
   s : ARRAY[1..m] OF emaphore;
   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 l。假设进程 Tl

首先调用 P 操作，则 P 操作过程把信号量 mutex 减 l，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 只有同时买通个航班的机票时才会发生于时间有关的错误，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。

5.3.4 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 */
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;
```

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*/


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;
```

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。

5.3.5 记录型信号量解决读者-写者问题

读者与写者问题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。如果考虑写者优先问题，即写者欲工作，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。程序将作如下修改，引入计数器 rc 和 wc 对读进程和写进程计数，mutex 是用于互斥的信号量，R 和 W 表示允许读和写的信号量。

```
var rc, wc : integer:=0;
    R, W, smutex: semaphore;
    rc := 0; wc := 0; R := 0; W := 0; s := 1;
Procedure read;
Begin
P(mutex);
if wc>0 then begin V(s); P(R); P(s);
rc := rc + 1;
V(R);
V(mutex);
读文件;
P(mutex);
rc := rc - 1;
if rc = 0 then V(W);
V(mutex);
end;
procedure write;
begin
P(mutex);
wc := wc + 1;
if rc>0 or wc>1 then V(s); P(W); P(s);
V(mutex);
写文件;
P(s);
wc := wc - 1;
if wc>0 then V(W); else V(R);
V(mutex);
end;
```

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，所有其它线程的读写操作必须等待。

5.3.6 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;
    customers, barbers，mutex : semaphore;
    customers := 0; barbers := 0; waiting := 0; mutex :=
1;
Procedure barber;
begin
  P(cutomers);
  P(mutex);
  waiting := waiting – 1;
  V(barbers);
  V(mutex);
  理发;
end;
procedure customer
begin
  P(mutex);
  if waiting < 椅子数
begin
  waiting := waiting + 1;
  V(customers);
  P(barbers);
End
  V(mutex);
end;
```

5.3.7 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 操作解放决生产者-消费者问题的算法描述如下：

```
var B : arrary [0,...k-1] of item;
   sput:semaphore:=k;            /* 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;           /* 互斥信号量
   sput := k;                   /* 缓冲区内允许放入产品数 */
   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
```

5.3.8 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   begin
    else
        进程进入第一个迁到的满足 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;
      L:semaphore:=rn;
      W:semaphore:=1;
begin
        cobegin
          process reader
            begin
              repeat
                SP(L,1,1);
                SP(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。

5.4 管程

5.4.1 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，难以防止无意地违返同步操作，而且容易造成程序设计的错误。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。

汉森(BrinchHansen)和霍尔(Hoare)提出了一个新的同步机制——管程。把系统中的资源用数据抽象地表示出来，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。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。

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 5-1 所示。

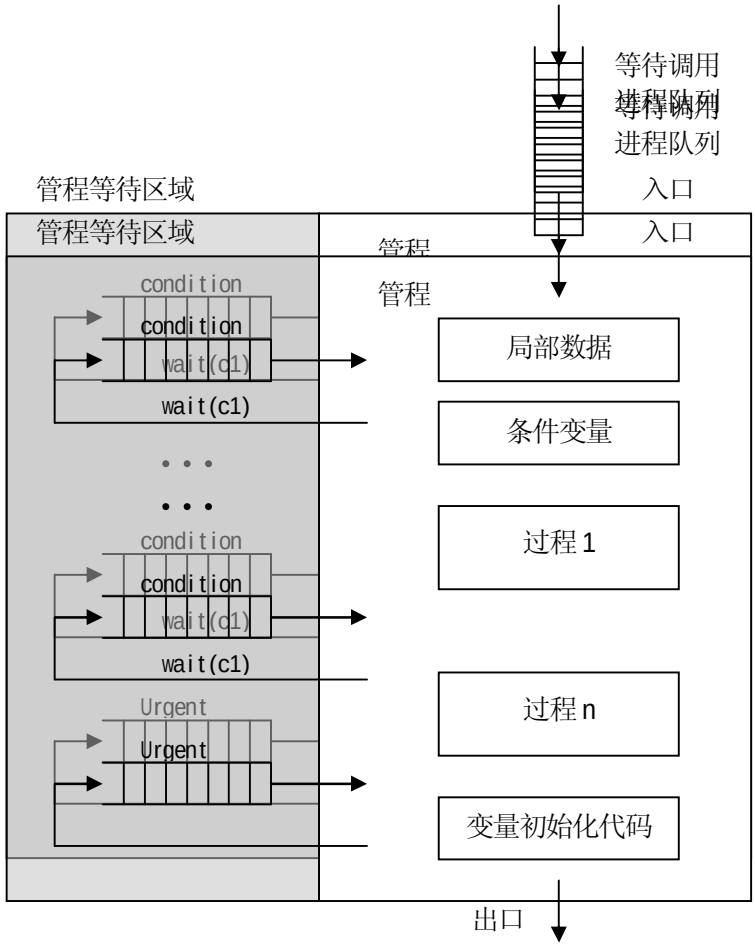


图 5-1　管程的结构示意图

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 signa1，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。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，它仅仅起到维护等待进程队列的作用，当不存在等待条件变量的进程时，signal 操作等于是空操作。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；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队列和从等待资源的队列中释放出来。当执行 wait 之后，相应的进程被置成等待状态，同时开放管程，允许调用管程中其它过程。当执行 signal 之后，指定队列中的一个进程被释放，如果被指定的队列中没有进程，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，造成两个过程同时执行。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 退出售程，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。

5.4.2 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，re1ease。

- 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 re1ease 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;
```

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(var fruit:(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;
```

5.4.3 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,var x-count :integer,var 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,var   x-count:integer,var
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;
```

5.5 消息传递

5.5.1 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。

消息传递机制需要提供两条原语 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问对数据的共享。

5.5.2 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。

1、直接通信方式

在直接通信方式下，企图发送或接收消息的每个进程必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)
      else    begin
        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)
      else    begin
        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}
```

5.5.3 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 5-2 所示：

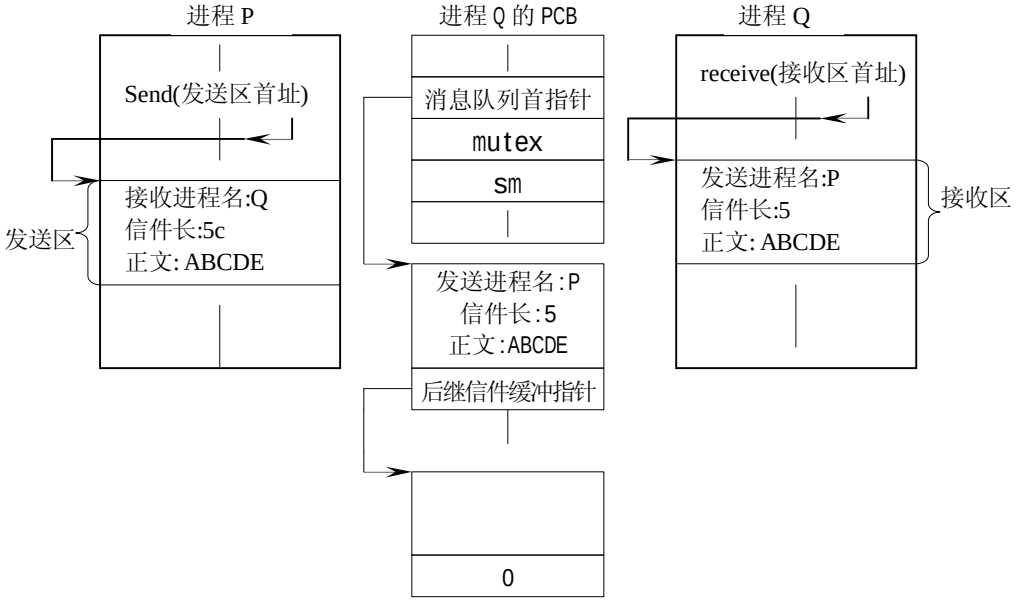


图 5-2　消息缓冲通信

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。

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;
```

5.5.4 管道和套接字

管道（pipeline）是 Unix 和 C 语言的传统通信方式。套接字（socket）起源于 Unix BSD 版本，目前已经被 Unix 和 Windows 操作系统广泛采用，并支持 TCP/IP 协议，即支持本机的进程间通信，也支持网络级的进程间通信。

管道和套接字都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。

CH6　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存储器。存储器可分成主存储器（简称主存）和辅助存储器（简称辅存）两类，本章讨论主存储器空间的管理，有关辅助存储器空间的管理可参见第四节。

主存储器的存储空间一般分中两部分：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。

6.1　存储管理的功能

计算机系统采用多道程序设计技术后，往往要在主存储器中同时存放多个作业的程序，而这些程序在主存储器中的位置是不能预先知道的，所以用户在编写程序时不能使用绝对地址。现代计算机的指令中地址部分所指示的地址通常是地址，逻辑地址可从 0 开始编号。用户按逻辑地址编写程序。当要把程序装入计算机时，首先，操作系统要为其分配一个合适的主存空间。由于逻辑地址经常与分配到的主存空间的绝对地址不一致，而处理器执行指令是按绝对地址进行的，所以必须把逻辑地址转换成绝对地址才能得到信息的真实存放处。把逻辑地址转换成绝对地址的工作称地址转换。

多个作业共享主存储器时，必须对主存储器中的程序和数据进行保护，并进行合理有效地调动，以达到充分发挥主存储器的效率。

为方便用户编制程序，使用户编写程序时不受主存储器实际容量的限制，可以采用一定的技术“扩充”主存储器容量，可使用得到比实际容量大的主存空间。

总之，存储管理的目的是要尽可能地方便用户和提高主存储器的效率。具体地说，存储管理有下面四个功能。

6.1.1　主存储器空间的分配和去配

要把作业装入主存时，必须按照规定的方式向操作系统提出申请，由存储管理进行具体分配。存储管理设置一张表格记录存储空间的分配情况，根据申请者的要求按一定的策略分析存储空间的使用情况找出足够的空闲区域，分配情况不能满足申请要求时，则让申请者处于等待主存资源的状态，直到有足够的主存空间时再分配给他。

当主存储器中某个作业撤离或主动归还主存资源时，存储管理要收回它所占用的全部或部分存储空间，使它们成为空闲区域（也叫自由区），这时也要修改表格的有关项。收回存储区域的工作也称“去配”。

6.1.2　主存储器空间的共享

主存储器空间的共享为了提高主存空间的利率效率，所谓主存储器空间共享有两方面的含义：

- 共享主存储器资源。采用多道程序设计技术使若干个程序同时进入主存储器，各自占用一定数量的存储空间，共同使用一个主存储器。
- 共享主存储器的某些区域。若干个作业有共同的程序段或数据时，可将这些共同的程序段或数据存放在某个存储区域内，各作业执行时都可访问它

们。

6.1.3 存储保护

主存储器中不仅有系统程序，而且还有若干道用户作业的程序。为了避免主存中的若干道程序相互干扰，必须对主存中的程序和数据进行保护。通常由硬件提供保护功能，软件配合实现。当要访问主存某一单元时，由硬件检查是否允许访问，若允许则执行，否则产生中断，由操作系统进行相应的处理。

最基本的保护措施是规定各道程序只能访问属于它的那些区域或存取公共区域中的信息，不过对公共的访问应该加以限制，一般说，一个程序执行时可能有下列三种情况：

- 对属于自己主存区域中的信息既可读又可写；
- 对公共区域中允许共享的信息或获得可使用的别的用户的信息，可读而不准修改；
- 对未获得授权使用的信息，既不可读又不可写。

对于不同结构的主存储器，采用的保护方法是各不相同的，在以后各节将作介绍。

6.1.4 主存储器空间的扩充

在计算机硬件的支撑下，软硬件协作，可把磁盘等辅助存储器作为主存储器的扩充部分来使用。当一个大型的程序要装入主存时，可先把其中的一部分装入主存储器，其余部分存放在磁盘上，如果程序执行中需用不在主存中的信息时，由操作系统采用覆盖技术将其调入主存，这样，用户编制程序时不需考虑实际主存空间的容量，就他能使用一个足够的主存储器一样。这种主存空间的扩充大大地方便了用户，使他们在编制程序量可免去考虑繁杂的覆盖问题。

6.2 连续存储空间管理

6.2.1 重定位

在采用多道程序设计技术的计算机系统中，往往要在主存器中时存放多个用户作业。为了方便用户都可以认为自己作业的程序和数据是存放在主存储器中从“0”单元开始的一组连续地址空间中，我们把这组地址空间称为“逻辑地址”空间。当用户把作业交给计算机系统执行时，存储管理就为其分配一个合适的主存空间，这个分配到的主存空间可能是从“N”单元开始的一个连续地址空间，称为“绝对地址”空间。由于逻辑地址经常与分到的主存空间的绝对地址不一致，而且对于每个用户的逻辑地址空间在主存储器中也没有一个固定的绝对地址空间与对应。因此，不能根据逻辑地址直接到主存储器中去取信息，但处理器执行指令是按绝对地址进行的。为保证使用的正确执行，必须根据分配到的主存区域存放指令和数据的逻辑地址转换成绝对地址。把逻辑地址转换成绝对地址的工作称“地址转换”或“重定位”。

重定位方式可以分成静态定位和动态定位两种。

- 静态定位：在装入一个作业时，把该作业中程序的指令地址和数据地址全部转换成绝对地址。由于地址转换工作是在程序执行前集中一次完成的，所以在程序执行过程中就不需再进行地址转换工作。这种定位方式称静态定位。

- 动态定位：动态定位是靠硬件的地址转换机构来实现的。硬件设置一个定位寄存器，当存储管理为程序分配了一个主存区域后，装入程序直接把程序装入到所分配的区域中，然后把该主存区域的起始地址存入定位寄存器中。在程序执行过程中动态地进行地址转换，只需将逻辑地址与定位寄存器中的值相加就可得到绝对地址。这种定位方式是在指令过程中进行的，所以称动态定位。

采用动态定位可实现程序在主存中移动。在程序执行过程中，若需要改变程序所占的主存区域时，则把程序移到一个新的区域后，只要改变定位寄存器的内容，该程序仍可正确执行，但采用静态定位时，程序执行过程中是不能移动的。

6.2.2 单连续存储管理

单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。

单连续存储管理的地址转换多采用静态定位，如图 6-1 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。

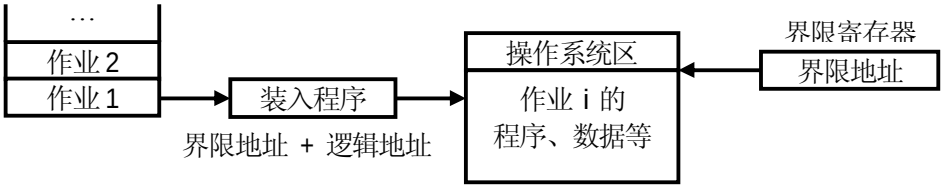


图 6-1　采用静态重定位的单连续存储管理

单连续存储管理的地址转换也可以采用动态定位，如图 6-2 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。

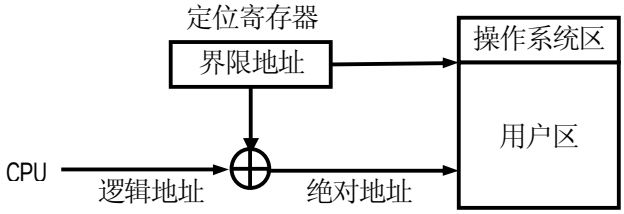


图 6-2　采用动态重定位的单连续存储管理

由于单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，都是一个作业独占存储区域，这就可能降低存储空间的利用率；
- 计算机的外围设备利用不高。

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 等等均采用单连续存储管理。

6.2.3 固定分区存储管理

固定分区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 6-3 所示。

操作系统区（8K）
用户分区 1（8K）
用户分区 2（16K）
用户分区 3（16K）
用户分区 4（16K）
用户分区 5（32K）
用户分区 6（32K）

图 6-3　固定分区存储管理示意

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 6-4 所示。页表

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	16K	0
5	64K	32K	Job2
6	96K	32K	0

图 6-4　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 6-4 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 6-3，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，且应归还所分析的区域。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 6-5 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，从而实现存储保护。

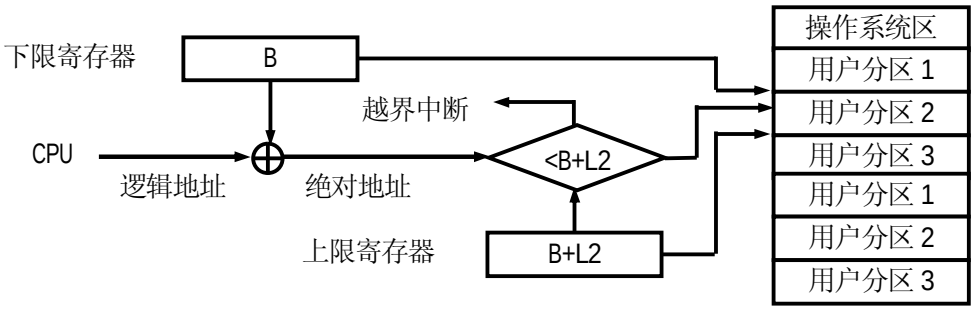


图 6-5　固定分区存储管理的地址转换和存储保护

采用固定分区存储管理，主存空间的利用不高，例如图 6-4 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，所以这种分配方式存储空间利用率不高，然而这种方法简单，因此，对于程序大小和出现频繁次数已知的情形，还是合适的。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。

6.2.4　可变分区存储管理

1、主存空间的分配和去配

可变分区方式是按作业的大小来划分分区。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。

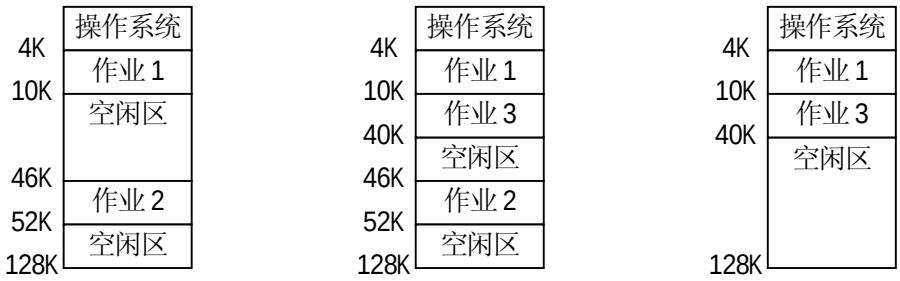


图 6-6　可变分区存储管理的主存分配示例

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。当一作主行结束撤离时，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。采用采用可变分区方式的主存分配示例如图 6-6。

从图 6-6 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 已分配区的情况表，另一张是未分配区的情况表，如图 6-7。

分区号	起始地址	长度	标志
1	4K	6K	Job1
2	46K	6K	Job2

（a）已分配区情况表

分区号	起始地址	长度	标志
1	10K	36K	未分配
2	52K	76K	未分配

（b）未分配区情况表

图 6-7　可变分区存储管理的主存分配表

图 6-7 的两张表的内容是按图 6-6 最左边的情况填写的，当要装入长度为 30K 的作业时，从未分配情况表中可找一个足够容纳它的长度 36K 的人．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，应从已分配区情况表中找一个空栏目登记作业 3 占用的起址、长度，同时修改未分配区情况表中空闲区的长度和起址。当作业撤离时则已分配区情况表中的相应状态改成“空”，而将收回的分区登记到未分配情况表中，若有相邻空宋区则将其连成一片后登记。由于分区的个数不定，所以表格应组织成链表。

常用的可变分区管理方式的分配算法有：

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，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，这给收回分区带来一些麻烦，每次收回一个分区后，必须搜索未分配区表来确定它在表格中的位置且要移动表格中的登记 。

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区为止。按这种方法，在收回一个分区时也必须对表格重新排列。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。

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，当作业执行是硬件完成地址转换。硬件

设置两个专门的特权寄存器：基址寄存器和限长寄存器。基址寄存器存放分配给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的长度。

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 6-8。

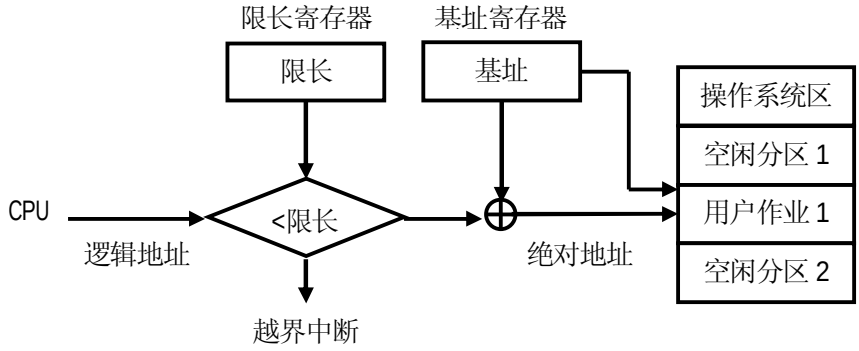


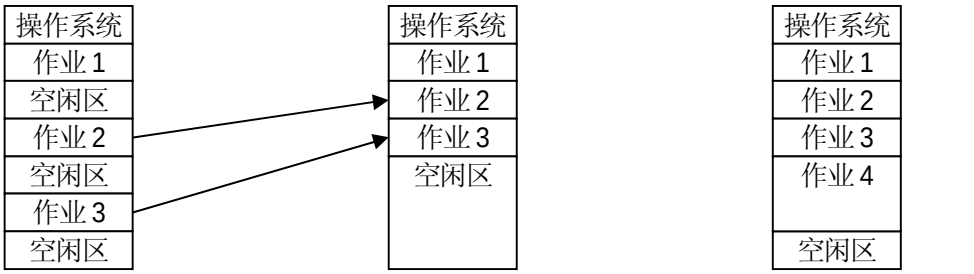
图 6-8　可变分区存储管理的地址转换和存储保护

当地逻辑地址小于限长值时，则逻辑地址加基址寄存器值就可得绝对值地址；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，仍只需一对基址/限长寄存器的。作业在执行过程中出现等待等，操作系统把基址/限长寄存器的内容随同该作业的其它信息，如 PSW，通用寄存器等一起保存起来。当作业被选中执行时，则把选中作业的基址/限长值再送入基址/限长寄存器。

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 6-9。



(a) 原主存分配情况　　(b) 移动主存中的作业　　(c) 装入作业 4

图 6-9　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的。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区域准备接待。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，主存提供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：

$$(1/2 + 1/3 + \cdots + 1/n) \cdot V$$

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。

6.3　分页式存储管理

6.3.1　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连续的存储区域，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。

分页存储器将主存分成大小相等的许多区，每个区称为一块，与此对应，编制程序的逻辑地址也分成页，页的大小与块的大小相等。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。由地址结构自然就决定了页面的大小，也就确定了主存分块的大小。

在进行存储分配时，总是以块为单位进行分配，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，当作业被分散存放后，如何保证作业的正确执行呢？

首先，进行存储分配时，应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包括一张作业表，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 6-10：

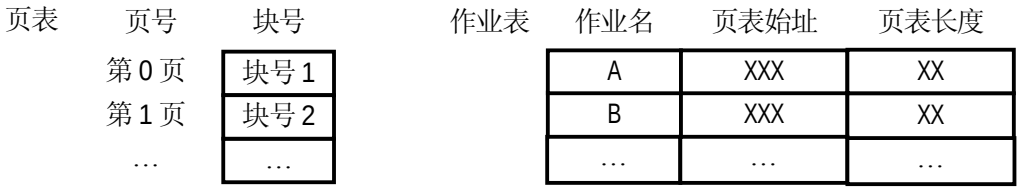


图 6-10　页表和作业表的一般格式

然后，借助于硬件的地址转换结构，在作业执行过程中按页动态定位。调度程序在选择作业后，从作业表中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就可以找到相应的页表，再按照逻辑地址中的页号查页表，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确切的地址进行存取。

图 6-11 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。

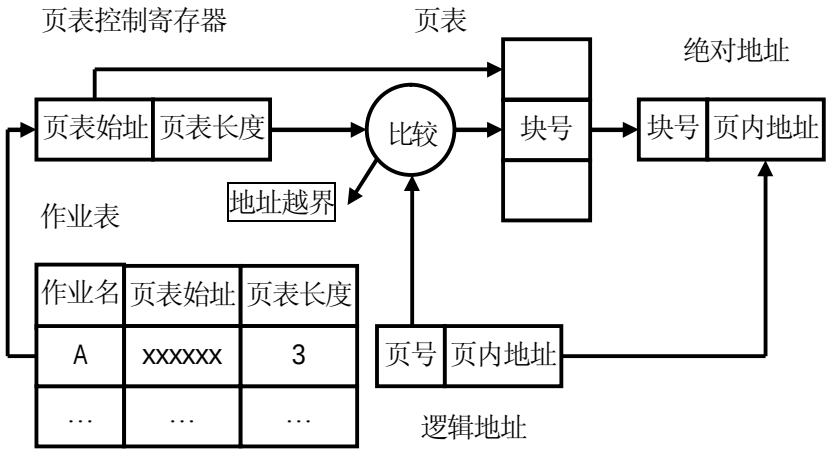


图 6-11　页式存储管理的地址转换和存储保护

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。

6.3.2 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的专门区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，降低了存取速度。

为了提高存取速度，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小于主存的，但造价高，故一般都是小容量的，例如 8-16 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”，即总是淘汰最先登记的那一页。

采用相存储器的方法后，便利地址转换时间大大下降。假定访问主存的时间为 200 毫微秒，访问相联存储器的时间为 40 毫微秒，相联存储器为 16 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：

（200＋40）×90%＋（200＋200）×10%＝256 毫微秒

比两次访问主存的时间 200 毫微秒×2＝400 毫微秒下降了 36%。

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。

6.3.3 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可分成若干块，主存分配以块为单位。可用一张主存分配来来记录已分配的块和尚未分配的块以及当前有多少空闲块。最简单的办法可用一张位示图来指出主存分配情况。例如主存的可分配区被分成 256 块，则可用字长 16 位的 16 个安来构成主存分配表，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，用另一字节记录

当前空闲块数，如图 6-12。

0/1	0/1	0/1	0/1	… …	0/1
0/1	0/1	0/1	0/1	… …	0/1
0/1	0/1	0/1	0/1	… …	0/1
0/1	0/1	0/1	0/1	… …	0/1
0/1	0/1	0/1	0/1	… …	0/1
…	…	…	…	… …	…
空闲块数					

图 6-12　位示图

进行主存分配时，先查空闲块数能否满足作业要求，若不能满足作业不能装入；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，发还主存时，则根据归还的块号，计算机出在位示图中的位置，将占标志清成“0”，归还块数加入到空闲块数中。

6.3.4　页的共享和保护

分页存储器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。现假定有一个共享 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，要想向只块写入信息则指令停止，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证键，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为“0”，规定程序状态中键为“0”时也不进行核对。

6.4 分段式存储管理

6.4.1 程序的分段结构

在前面讨论的几种存储管理方法中，假设用户程序都是从 0 开始编址的一个单一连续的逻辑地址空间。事实上，程序还可以有一种分段结构，如图 6-13 所示，一个程序由若干段组成的，例如由一个主存程序段、若干子程序数组段和工作区段所组成，每个段都从“0”开始编址，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）调用一段的功能，程序在编译或汇编时给每个段名再定义一个段号。

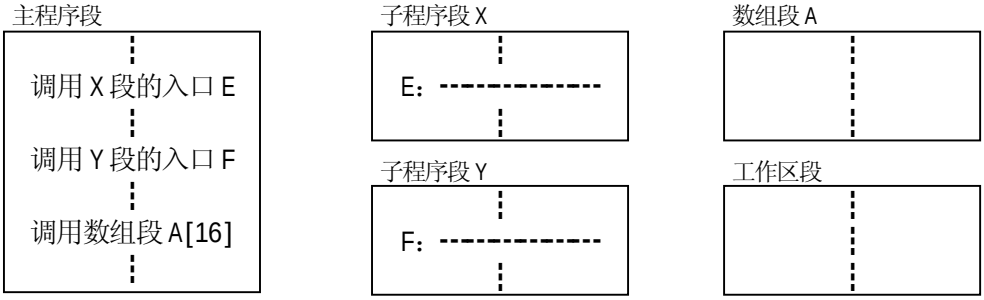


图 6-13　程序的分段结构

6.4.2 分段式存储管理的基本思想

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逻辑地址：

段　号	单　元　号

在分页式存储管理中，页的划分——即逻辑地址如何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，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 6-14：

段表

段号	始址	始址
第 0 页	XXX	XXX
第 1 页	XXX	XXX
…	…	…

作业表

作业名	段表始址	段表长度
A	XXX	XX
B	XXX	XX
…	…	…

图 6-14　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 6-15。

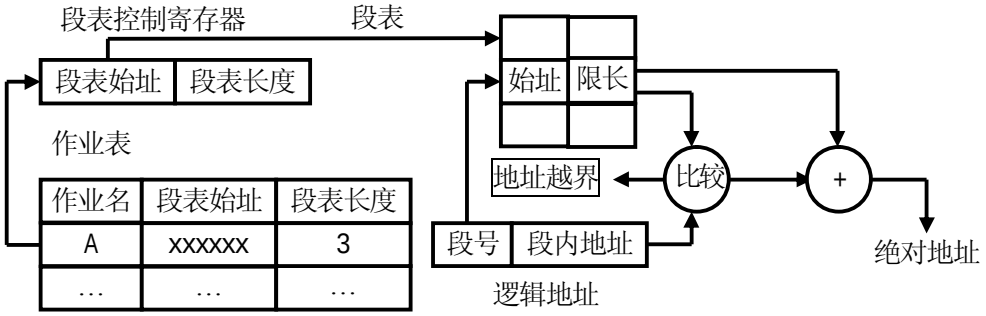


图 6-15　分段式存储管理的地址转换和存储保护

6.4.3　段页式存储管理

分段式存储管理也可以在分页式存储管理的基础上实现，就是段页式存储管理。段页式存储管理的控制结构更为复杂，包括作业表、段表和页表三级结构，以及更为复杂的地址转换和存储保护方法，实现的代价也较大。有兴趣的同学可以自己考虑其实现方法，或参见其它文献。

6.4.4　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。

6.4.5　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。

6.5　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。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这些信息的，甚至有些部分在作业执行的整个过程中都不会被使用到。于是，提出了这样的问题：能否不把作业的全部信息同时装入主存储器，而是将其中一部分先装入主存储器，另一部分暂时存放在辅助存储器中，待用到这些信息时，再把它们装到主存储器中。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，作业也能执行。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”。

虚拟存储器实际上是为扩大主存而采用的一种设计技巧。虚拟存储器的容量由计算机的地址的地址结构和辅助存储器的容量决定，与实际的主存储器的容量无关。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护作业的正确运行？回答是肯定的，这只要对程序的执行进行分析就可以发现。

第一、程序往往包含若干个循环，运行时一里某部分的指被执行或某部分的数据被访问后，常常公多次执行或访问这个部分。第二，程序在每次运行中在一段时间运行中，往往是集中访问一区域中的数据。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。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。在装入部分信息的情况下，只要调度很好，不仅可以正确运行，而且能提高系统效率。

虚拟存储器是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：如何决定当前应该是哪些信息在主存中？如果作业要访问的信息不在主存时那怎么办？如何发现当前所用信息不在主存？如何实现虚地址到实地址的转换？这些问题将在后面各节中详细讨论。

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两个分时系统 MULTICS（Multiplexed Information and computing service）和 IBM 系列中得到实现。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。

6.6 分页式虚拟存储系统

6.6.1 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或磁鼓这一类辅助存储器中，当作业被调度投入行时，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时，再把它们装入。

首先，应该指出哪些页已经在主存，哪些而还没有装入主存。这只需将页表进行修改，如图 6-16 所示：

页号	驻留标志	块号	辅存地址	调出标志
第 0 页	---	---	---	---
第 1 页	---	---	---	---
---	---	---	---	---

图 6-16　页式虚拟存储管理的页表扩展

驻留标志位用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，此时可以根据辅存地址知道该页在辅助存储器中的地址。调出标志可以包括修改位（Modified）、引用位（Renferenced）、禁止缓存位和访问位，用来跟踪页的使用。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。引用位则在该页被引用时设置，无论是都或写，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的。访问位则限定了该页允许什么样的访问，读、写和执行。

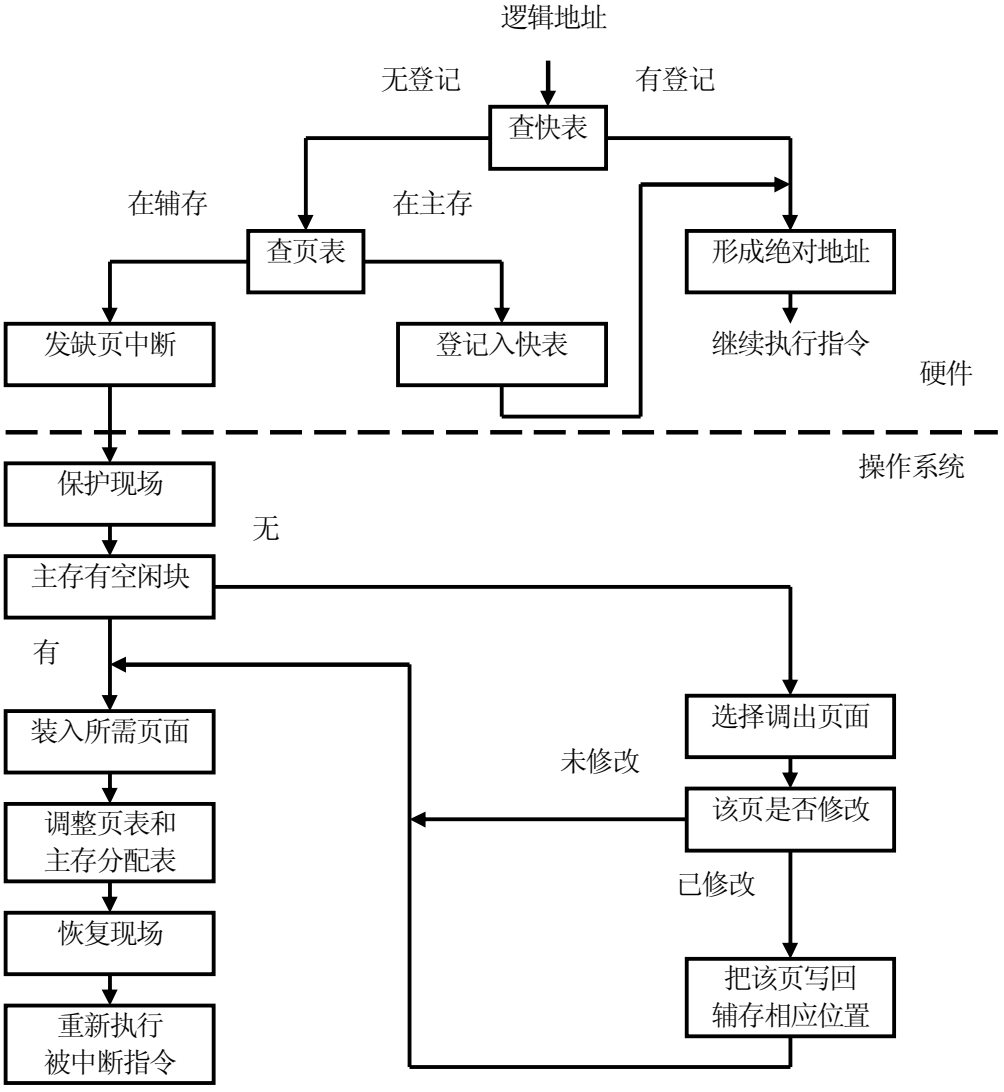


图 6-17　缺页中断处理流程

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，则按前节中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页找出且装入主存，在页表中填上它占用的块号且修改标志。若主存已没有空闲块，则必须先调出已在主存中的某一页，再将所需的页装入，对页表和主存分配表作相应的修改。见图 6-17。

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，来指出对应页是否修改过，若访问某页时执行的是写指令，则置上已修改标志。若被选择的需要调出的页，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，而已修改过的页调出时必须将该页写回到辅助存储器中。

如果需调出的页正在与外围设备交换信息，那么该页暂量不能调出，这时要么另选一页调出，要么等待该页与外围设备信息交换结束后再调出。同样可用标志来指出某页是否在与外围设备交换信息。

在分页式虚拟存储系统中，由于作业的诸页是根据请求页被装入主存的，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 MOLTICS 以及 UNIVAC 系列 70/64 的 VMS 等都采用分页式虚拟存储系统。

6.6.2 页面调度

实现虚拟存储器能给用户提供一个容量很大的存储器，但当主存人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，这个工作称页面调度。所以，页面调度算法实际上就是用来确定应该淘汰哪页的算法。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在来回调度上。这种现象叫做“抖动”，又称“颠簸”，一个好的调度算法应减少和避免抖动现象。

为了衡量调度算法的优劣，我们考虑在固定空间的前提下来讨论各种页面调度算法。这一类算法是假定每道作业都给固定数时的主存空间，即每道作业占用的主存块数不允许页面调度算法加以改变。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），即最多只能容纳 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：

- 分配给作业的主存块数。分配给作业的主存块数多，则缺页中断就低，反之，缺页中断率就高。
- 页面的大小，如果划分的页面大，则缺页中断就低，否则缺页中断率就高。
- 页面调度算法。
- 作业本身的程序编制方法。程序编制的方法不同，对缺页中断的次数有很大影响。

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化程度密切相关。程序的局部化有两种：时间局部化和空间局部化。

时间局部化是指一旦某个位置——数据指令——被访问了，它常常很快又要再次被访问。这可借助于循环、经常用到的变量和也要用到。这可以尽量采用顺序的指令串、线性的数据结构（例如，数组或常用的变量彼此相近放置）来实现。

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高的局部化程度。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页很大。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，在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 块空间时才让它进入主存执行，这样可以使系统获得高效率。

6.6.3 页面调度算法

一个理想的调度算法是：当要调入一页而必须淘汰一页旧页时，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（OPT）。

Belady 算法时一种理想化的页面调度算法，下面分别介绍几个比较典型的实用化的页面调度算法。

1、随机调度算法

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。这种调度算法很简单，但效率低，一般不采用。

2、先进先出调度算法（FIFO）

先进先出调度算法是一种低开销的页面调度算法，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) 表示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，如果某一矾要不断地和经常地被使用，但它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。

3、最近最少用调度算法（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属性来考虑的，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。

第二种模拟方法则是为每个页设置一个多位的寄存器 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。有关工作机模型的讨论参见下节。

4、时钟页面调度算法(CLOCK Policy)

如果在 FIFO 算法的基础上，结合 LRU 算法的第一种模拟方法进行改造，就形成了第二次机会页面调度算法。该算法的基本思想是：首先采用 FIFO 算法选择最先进入内存的页，如果该页最近没有被访问过（R 位为 0），则将其淘汰，否则把它加入对列尾部，继续根据 FIFO 算法选择下一页。

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 FIFO 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，这就是时钟页面调度算法的得名。图 6-18 给出了时钟页面调度算法的示意，当发

生缺页中断时，算法首先检查表针指向的页面，如果 R 位为 0，则将其淘汰，并把新页插入这个位置，然后表针前移一个位置；否则清除 R 位，表针前移一个位置，重复上述过程直到找到一个 R 位为 0 的页面将其淘汰。

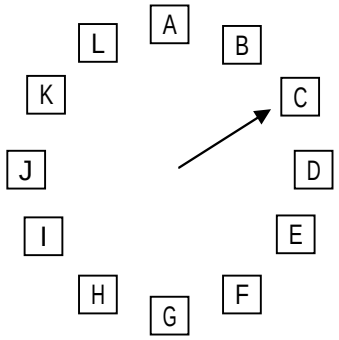


图 6- 18　时钟页面调度算法示意

如果在 FIFO 算法的基础上，结合 LRU 算法第二种模拟方法（老化算法）进行改造，则构成了改进的时钟页面调度算法，又称 wsclock 算法。

5、最不常用页面先调度算法（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，但代价太高的而且选择多大的 t 最适宜也是难题。

6.6.4　页式虚拟存储系统的几个设计问题

1、对页面大小的概略讨论

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，如果页面大小较小，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 1K 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。

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 6-19 所示。

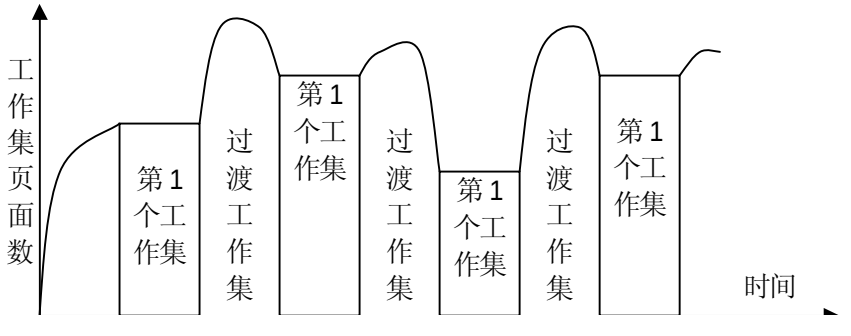


图 6-19　工作集的演变过程

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。

通常，用 W(t, w)表示从时刻 t-w 到时刻 t 之间所访问的不同页面的集合，这就是作业在时刻 t 时的工作集。用 Nw(t, w) 表示工作集中所包含的页面数。如果系统能随 Nw(t, w)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。

通过工作集来管理主存的分配和使用，从理论上来说是一件非常好的事情，但工作集算法的实现有很多困难，目前仍然在研究和试验中。

3、局部和全局分配

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。当出现一次缺页中断时，页面调度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。

如果页面调度算法的作用范围是整个系统，称为全局页面调度算法。它可以在可运行进程之间动态地分配页架。

如果页面调度算法的作用范围局限于本进程，称为局部页面调度算法。它实际上需要为每个进程分配固定的内存片断。

在通常情况下，尤其是工作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内存区存在，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。

比较简单的方法是根据进程数平均分配内存，这对于大程序是不合理的。也可以根据进程数和进程大小按比例分配内存，但是缺页中断率的大小，或者说系统的颠簸也可能与之无关。

一种更好的解决方法是 PEF 算法，它根据缺页中断率直接控制内存分配。具体方

法是设置两个缺页中断率阀值 A 和 B。A 是较高的缺页中断率阀值，当一个进程在某段时间内的缺页中断率高于 A，就可以考虑给它分配更多的内存以降低缺页中断的次数。B 是较低的缺页中断率阀值，当一个进程在某段时间内的缺页中断率低于 B，就可以考虑收回分配给它的多余内存。如果发现一直不能使内存中的所有进程的缺页中断率低于 A，就应该从内存中暂时移出一些进程，以便让出部分内存，降低整个系统的缺页中断率，这事实上是一种负载控制，即中级调度。

6.7 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 6-20：

段号	特征	存取权限	扩充位	标志	主存始址	限长	辅存始址

图 6-20　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 6-21。

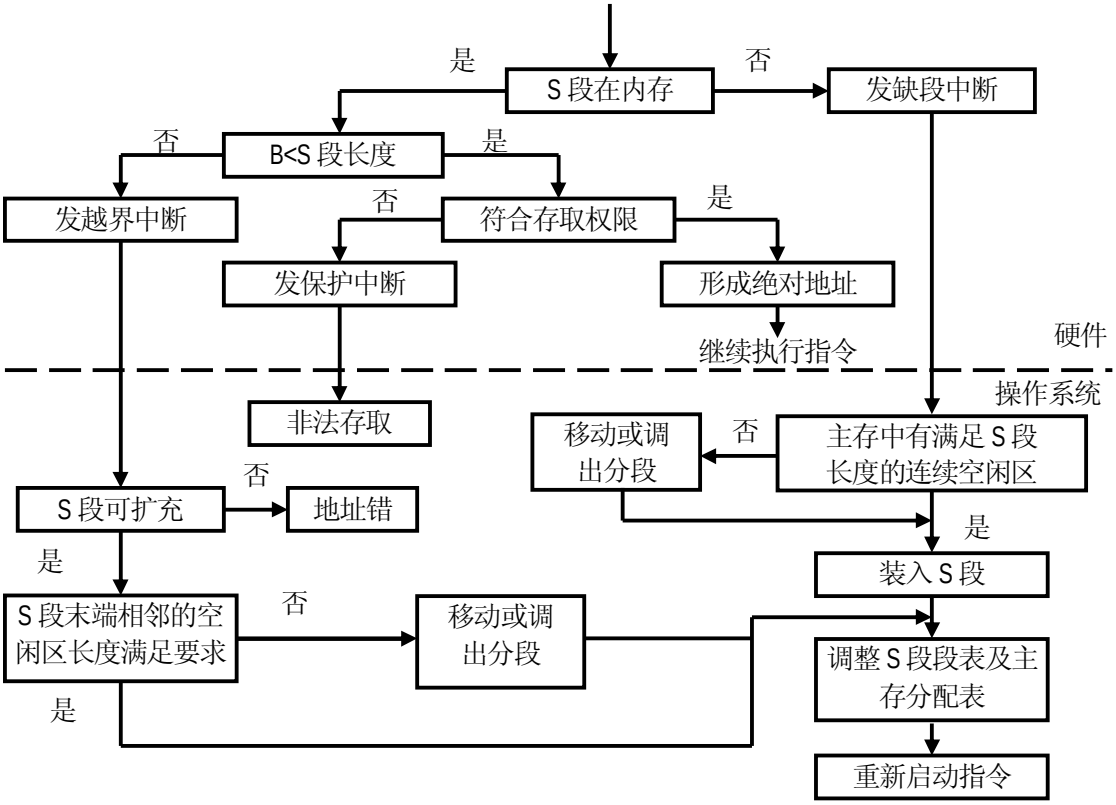


图 6-21　分段式存储管理的地址转换和存储保护

6.8　实例研究：Intel 的 Pentium

Intel 的 Pentium 和 Pentium Pro 既有分段机制又有分页机制，它包括 16K 个独立的段，每个段最多可以容纳 10 亿个 32 位字。Pentium 虚拟存储器的核心是两张表：LDT（局部描述符表）和 GDT（全局描述符表）。每个程序都有自己的 LDT，但是所有计算机上的所有程序共享一个 GDT。LDT 描述局部于每个程序的段，包括代码、数据、堆栈等，GDT 描述系统段，包括操作系统自己。

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，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 6-22 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。

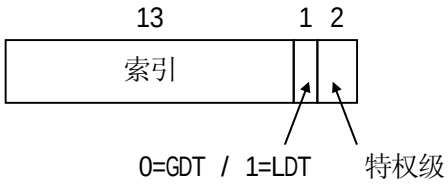


图 6-22　Pentium 的选择符

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。一个描述符由 8 各自结构称，包括段的基址、长度和其他信息，如图 6-23 所示。

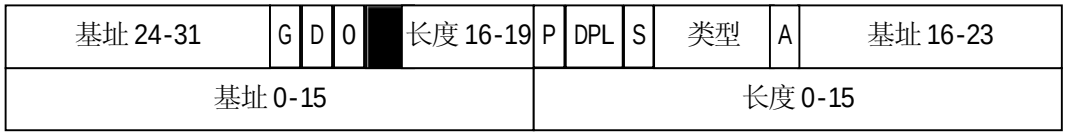


图 6-23　Pentium 的代码段描述符，数据段描述符略有区别

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 $2^{20}\times 4KB=2^{32}$ 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）。
- 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

选择符的格式经过了精心的挑选，定位描述符十分方便。首先根据第二位选择 LDT 或 GDT；随后选择符被拷贝进一个内部寄存器中并且它的低三位被清零；最后 LDT 或 GDT 表的起始地址被加到它上面，得到一个直接指向描述符的指针。例如选择符 72 指向 GDT 的第 9 个入口，它位于 GDT+72。

现在让我们跟踪一个（选择符，偏移）被转换为物理地址的过程。只要微程序直到哪一个段寄存器正在被使用，它就能从内部寄存器中找到对应于这个选择符的完整的描述符，如果段不存在（选择符为 0），就会发生一次陷入（中断）；如果段已经被换出，就会发生一次陷入（缺段中断）。

它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）。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所谓的线性地址（linear address），如图 6-24 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。

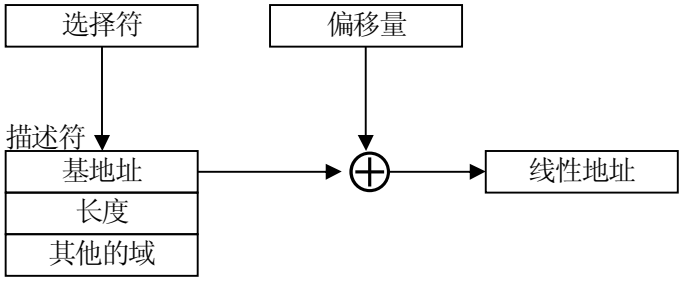


图 6-24 选择符，偏移被转换为线性地址的过程

如果通过全局控制寄存器中的 1 位禁止分页的话，线性地址就被解释为物理地址并被送往存储器用于读写。因此在分页被禁止时，我们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。顺便提一句，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。

在另一方面，如果分页被允许，线性地址将被解释成一个虚地址并通过页表映射成为物理地址。这里唯一真正复杂的是在 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 使用了一种两级映射以在段较小的情况下减少页表的尺寸。

每个运行程序都有一个由 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，页表项指向页架（块），这个方案如图 6-25 和图 6-26 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架的物理地址；最后，Offset 被加到页架的地址上得到需要的字节或字的物理地址。

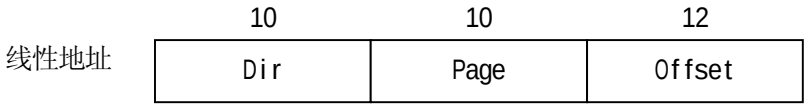


图 6-25　允许分页时线性地址的组成

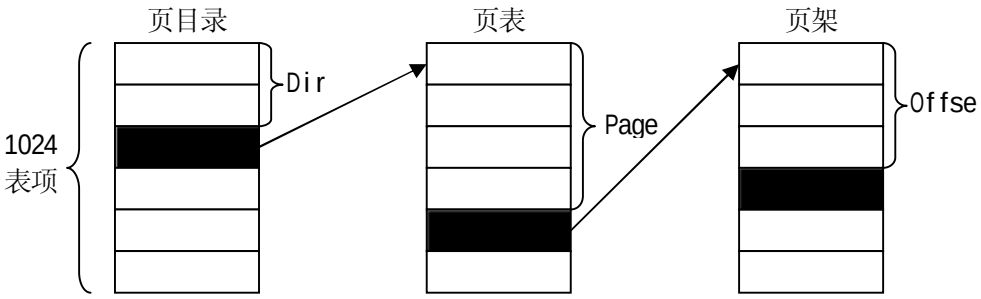


图 6-26　线性地址转换为物理地址的过程

如图 6-27 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 6-27　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。

下面讨论一下 Pentium 的保护机制。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。
- 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。

CH7　设备管理

7.1　设备管理的基本功能

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。为了方便地使用各种外围设备。

以上两类外围设备的物理特性各不相同，因此，操作系统对它们的管理也有很大差别。为了使这些设备在用户面前具有统一的格式和一致的面貌，对于存储型设备，信息以文件为单位存取；对于输入输出设备，信息以文件为单位输入输出。这样，用户可以通过“按名存取”文件实现对外围设备的访问，而不必考虑直接控制外围设备时应做的许多繁琐工作。操作系统除了要提供文件系统外，还必须要有实现对外围设备上文件信息的物理存取和设备控制的功能。操作系统中完成这一功能的程序就是设备管理。

设备管理要达到的主要目标是：提供统一界面、方便用户使用，发挥系统的并行性，提高 I/O 设备使用效率。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。

7.2　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已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。

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，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。

7.2.1 I/O 设备的分类

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。

7.2.2 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。

- 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。早期计算机和微机往往采用这种方式，中央处理机的大量时间用在等待输入输出的循环检测上，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。
- 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。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。若为外围设备增加缓冲寄存器存放数据，则可大大减少中断次数。中央处理器在外围设备与缓冲寄存器交换信息期间可执行其它指令。例如行式打印机、卡片机、字符显示器等均配置数据缓冲寄存器，提高了中央处理器和外围设备并行工作的程度。
- DMA 方式：在直接主存存取方式中，I/O 控制器有更强的功能，它不仅设有中断机构，而且，还增加了 DMA 控制机构。在 DMA 控制器的控制下，它采用‘偷窃’总线控制权的方法，让设备和主存之间成批交换数据，而不必由 CPU 干予。这样可减轻 CPU 的负担，因每次传送数据时，不必进入中断系统；只要 CPU 暂停几个周期，从而，使 I/O 数据的速度也大大提高。目前，在小型、微型机中的快速设备均采用这种方式，DMA 的操作全部由硬件实现，不影响 CPU 寄存器的状态。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。

- 通道技术：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。

7.2.3 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。

有些控制器之间做在计算机主板上，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 7-1 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。

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 7-1　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。

7.2.4　直接主存存取 DMA

当设备控制器不能直接访问主存时，控制器将逐个比特地从设备读出一批数据放入内部缓冲区中，然后进行校验；接着控制器发出中断信号，操作系统开始逐个字节地从控制器寄存器中把数据读入内存，注意每次只能读一个字节，所以读入数据的过程是一个循环。

为了解决这一问题，引入了直接主存存取（direct memory access, DMA）技术。使用 DMA 后，CPU 除了告诉控制器数据块的外存地址外，还需告诉控制器数据块的内存地址和要传输的字节数。控制器首先从设备中把整个数据读入内部缓冲区中并进行校验；接着把第一个字节/字拷贝到内存区，然后对 DMA 地址和 DMA 计数分别进行增减（刚刚传送的字节数）；前一过程将不断重复直到 DMA 计数为 0，即所用数据转入内存；最后设备控制器发出中断信号，通知操作系统数据已经读取完毕。显然，采用了 DMA 技术后，操作系统无需再把数据拷贝到内存，这将大大提高 CPU 与外围设备的并行度。

可能有人会提出疑问：为什么控制器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。如果上一个子还未送入内存前另一个字到达，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。

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。

7.2.5 输入输出处理器——通道

对于大型计算机系统来说，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。上例中，中央处理器只要用一条启动指令，就可通知输入机输入 1000 个字符，一秒钟之后发生中断再去处理。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，如图 7-2 所示。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。

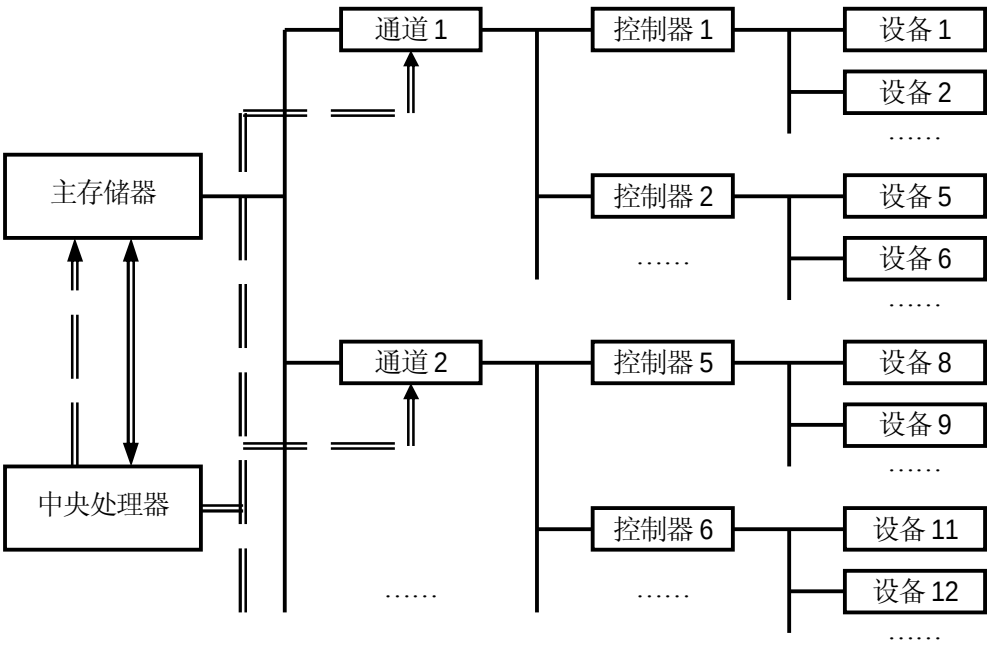


图 7-2　通道方式的计算机系统结构

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、磁鼓和磁盘快速设备。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。如果按在字节多路通道上，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，它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。

7.3 I/O 软件原理

I/O 软件的总体设计目标很明确。其基本思想是：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。

7.3.1 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。

7.3.2 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。

7.3.3 与硬件无关的 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。

7.3.4 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 7-3 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。

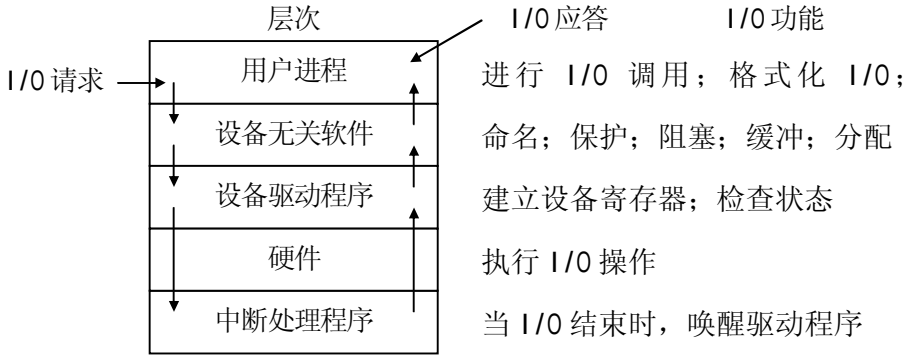


图 7-3　I/O 系统的层次及其功能

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。

7.4　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。

7.4.1　I/O 指令

操作系统通过 I/O 指令要求通道或设备为其服务，常用的 I/O 指令包括：启动 I/O（SIO）、查询 I/O（TIO）、查询通道（TCH）、停止 I/O（HIO）、停止设备（HDV）等。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。下面以启动 I/O（SIO）为例，说明 I/O 指令的含义和用法：

为了启动输入输出，必须给出通道号，设备号及通道程序首地址。

例如，指令

SIO　X　00E

规定了 0 号通道，0E 号设备，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。

SIO 指令执行后，如果条件码为 0，表示设备已被启动成功，通道从通道地址字单元取得通道程序首地址开始工作，CPU 可以执行下一条指令。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。

7.4.2 通道命令和通道程序

通道根据通道命令（CCW）内容控制外围设备执行输入输出操作。通道命令由操作码（常称为命令码）、数据主存地址、交换字节个数和标志码等组成。若干条通道命令规定了设备的一种操作，而通道程序规定了外围设备所应执行操作及顺序。

通道命令的命令码字段规定了外围设备所执行的操作。通道命令码分成三类：数据传输类（读、反读、写、取状态），通道转移类（转移），设备控制类（随设备不同而异）。

对于数据传输型通道命令，通道命令的数据主存地址字段规定了本通道命令指定的主存数据地址，而传送字节数 字段指出这个存区域的大小。对于“读”通道命令，这个区域用来存放输出到个围设备读出的信息；对于“写”通道命令，这个区域用来存放输出到外围设备的信息；

对于设备控制通道命令，通道命令的数据主存地址字段用来存放从外围设备取出的或送给外围设备的控制信息。

对于通道转移命令，通道命令的数据主存地址字段用来指定转移地址。

通道命令的标志码字段用来定义通道程序的连接方式或标志通道命令的特点。标志码的主要作用是：1）定义无链，表示本条通道命令是通道程序的最后一条，执行完本条后，通道程序便结束。2）定义命令链，表示本命令的操作已是最后一条，后面还有通道命令但为其它命令码。3）定义数据链，表示下一条通道命令将延用本条通道命令码，但由下一条通道命令规定一个新的主存区域。4）标志在通道命令执行过程中，禁止发出长度错误指示。5）标志在“读”型操作实现假读功能。6）标志请求程序进程中断，通道程序执行这条指令时，导致一个外围设备中断请求，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。

7.4.3 通道启动和 I/O 操作过程

输入输出操作中要使用两个固定存储单元：通道地址字（CAW）和通道状态字（CSW）。通道地址字是存放在主存固定单元的控制字，用来存放通道程序的主存首地址。通道状态字保存在主存的另一个固定单元，是通道向操作系统报告情况的汇集。通道利用 CSW 可以提供通道和外围设备执行输入输出指令的情况，以及外围设备和通道识别的信号：如通道结束、设备结束、注意、状态修正、忙，以及各种错误信号。CSW 构成应该包括：通道命令地址、设备状态、通道状态、剩余传送字节数等。

启动通道和 I/O 设备操作的过程简述如下：

- 第一步：CPU 使用启动 I/O 指令启动输入输出，通道首先判断通道的状态，是否可以使用，并据此形成条件码，此后 CPU 按条件码执行下条指令。若通道可用，则 CPU 传送本次设备地址，I/O 操作开始。
- 第二步：通道从主存固定单元取 CAW，根据该地址从主存取得第一条通道命令，通道执行 CCW 命令，将 I/O 地址传送控制器，同时，向它发出读、写或控制命令。
- 第三步：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；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，当通道程序执行结束时，由通道向 CPU 发结束中断，请求干予。

由上可见：CPU 向通道发出输入输出指令，命令通道工作，通道根据其状态形成

条件码作为回答。这一通信过程发生在操作开始期，CPU 根据条件码作就可决定转移方向。

7.5 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予盾，以及协调逻辑记录大小与物理记录大小不一致的问题，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，在输出区数据时，只有在系统还来不及腾空缓空缓冲而进程又要从中读取数据时，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。

7.5.1 单缓冲

单缓冲是让输入和输出设备共用一个缓冲区的缓冲技术。通常，输入设备和输出设备以串行方式工作，当输入设备数据输入缓冲区时，输出设备不工作，处于等待状态；反之，当输出设备从缓冲区取数据输出时，输入设备不能工作，应当等待。一般说，每次读写操作都要转入进程调度，所以，采用单缓冲技术，I/O 设备并行性差。系统效率低。

7.5.2 双缓冲

双缓冲是为输入输出 设备分配两个缓冲区的缓冲技术。在输入数据时，首先填满缓冲区 1。进程从缓冲区 1 提取数据使用的同时，输入设备填充缓冲区 2。当缓冲区 1 空出时，进程又可以从缓冲区 2 获得数据。同时，输入设备又可以填充缓冲区 1 。两个缓冲区交替使用，使 CPU 和 I/O 设备的并行进行性进一步提高， 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。

双缓冲与输入和输出设备相连时，可如下进工作：首先、让输入设备填满缓冲区 1，启动输出设备输出缓冲区 1 的数据，并同时启动输入设备填缓冲区 2。当缓冲区 2 填满后，并且缓冲区 1 已输入出空，又可以让输出设备输出缓冲区 2 的数据，同时，让输入设备填充缓冲区 1。这样就能使 I/O 设备充分地并行工作，也能进一步减少进程调度次数。

7.5.3 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配的情况仍不能适应。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，常常采用多缓冲技术。

操作系统从自由主存区域中分配一组缓冲区组成多组冲，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，都采用多缓冲技术，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 6 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。

7.6 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。本节讨论与驱动调度有关的技术。

7.6.1 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。

7.6.2 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。

7.6.3 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。

7.6.4 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，“调度电梯”算法就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法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，所以性能较差。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。

2）“扫描”算法。磁盘臂移动时，一个沿一个方向移动，扫过所有的柱面，然后再向相反方向移动回来。

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陬请求的等待时间不至太长。

4）“单向扫描”算法。这是为适应极大量存取请求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。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”仅适应不断大批量输入输出存取请求磁道上存放记录数量较大情况。

上在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以达到高的吞叶量。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。

7.7 设备分配

7.7.1 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，该作业也就不能投稿运行。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：用户怀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。

7.7.2 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。

7.8 虚拟设备

7.8.1 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，例如，从一台读卡机上输入了作业 J1 的全部数据后，虽然，作业 J1 尚未执行完，但这台读卡机已可用来输入作业 J2 的数据，实现了设备的共享。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。

7.8.2 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 7-4 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”。

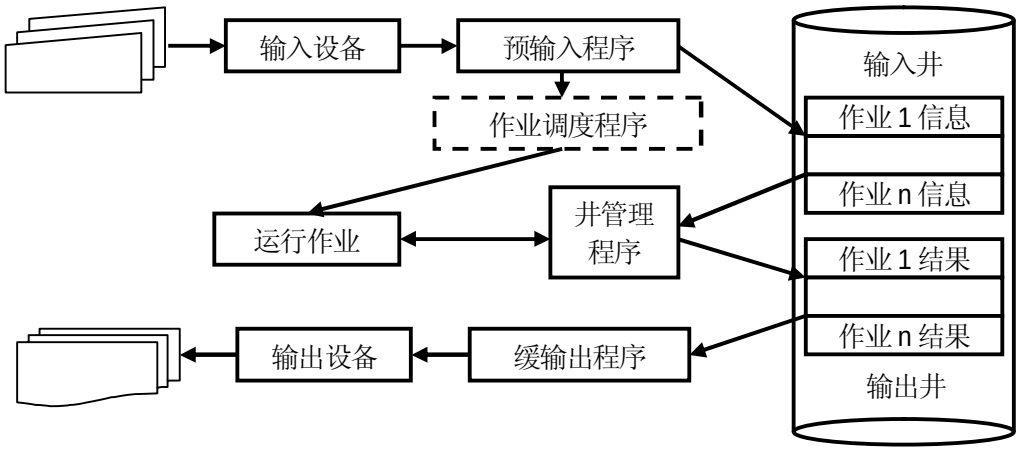


图 7-4　斯普林系统的组成和结构

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。

存放放井在文件称井文件，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，以后各尬用指针连起来，读出 n 块后，由连接指针就可找到第 m＋ 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放的井文件空间，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。

7.9 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。随后我们介绍 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。

7.9.1 Windows NT4 的设备管理

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。

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 7-5 说明了这种基本结构。

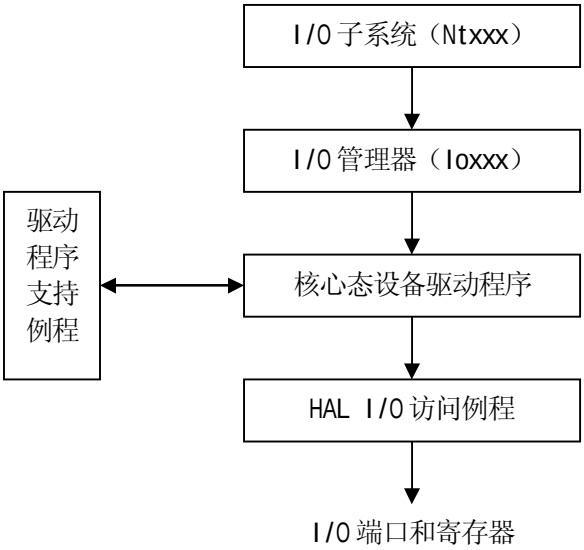


图 7-5　I/O 系统结构

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。

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 7-6 给出了核心态设备驱动程序之间的关系。

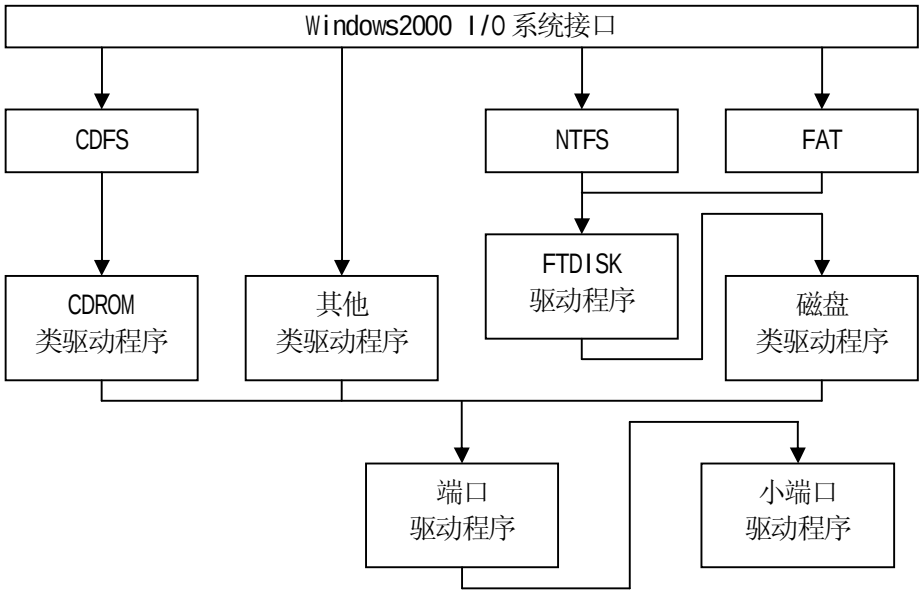


图 7-6　核心态设备驱动程序之间的关系

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。

7.9.2 Windows 2000 设备管理的扩展

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。

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。

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。

4、即插即用结构

Windows2000 的即插即用组件如图 7-7 所示。

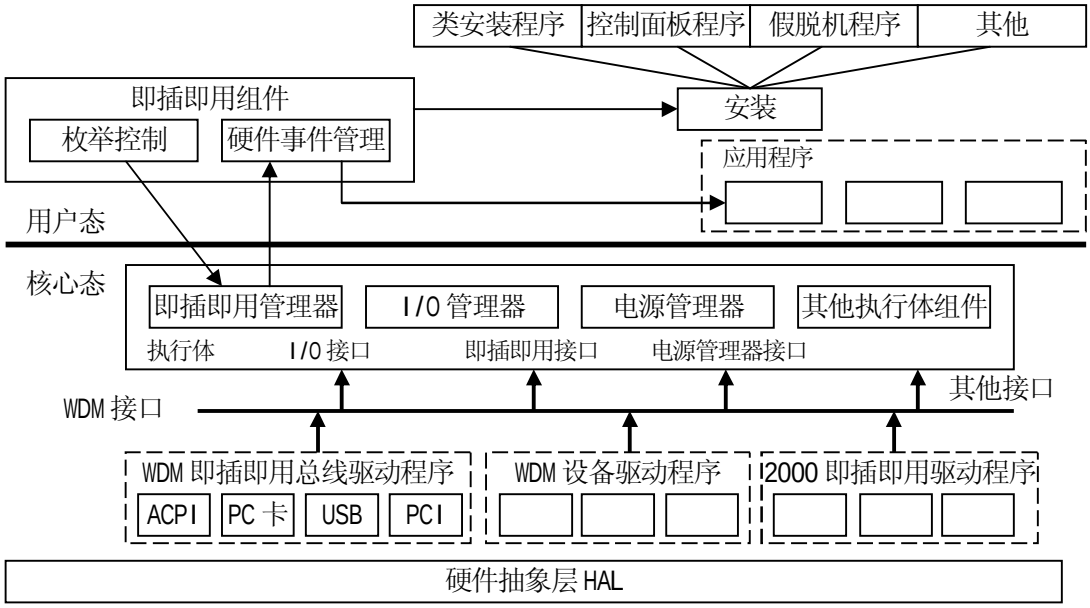


图 7-7　Windows2000 的即插即用结构

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。

CH8　文件管理

8.1　文件系统概述

8.1.1　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。

文件是在逻辑上具有完整意义的信息的集合，它有一个名字以供识别。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：如按用途可人成：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，硬件失效而可能造成文件信息破坏痢，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，又减少了传送文件的交换时间，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。

8.1.2　文件系统及其功能

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，文件系统向用户的功能是：

- 文件的按名存取
- 建立文件目录
- 实现从逻辑文件到物理文件的转换
- 分配文件的存储空间
- 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。

8.2 文件

本节将从用户角度来研究文件，即文件是如何使用的，它有些什么特性。

8.2.1 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、位置以及磁盘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，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；操作系统也将根据该文件名对文件进行保护。

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-98。也有很多操作系统的文件命名需要区分大小写，如 Unix。

8.2.2 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：

- 正规文件：即我们前面所讨论的存储在外存储设备上的数据文件。

- 目录文件：及管理文件系统结构的系统文件。
- 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，正规文件包括 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。

8.2.3 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。

8.2.4 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。

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。

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。

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。

8.2.5 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。
- 文件定位：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。
- 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。

8.3 文件目录

8.3.1 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-ON。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-ON，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。

索引节点的内容解释如下。

- 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[0]-di-add 文件最近被访问的时间。
- 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。

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。

8.3.2 一级目录结构

如图 8-1 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。

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

会出现混光淆而无法实现‘按名存取’。如果人为地限制文件名命规则，对用户来说又极不方便；二是难于实现文件共享，如果允许不同使用户不同文件名 来共享文件具有不同的名字，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。

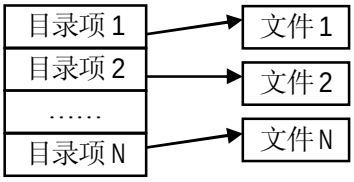


图 8-1　一级目录结构示意图

8.3.3　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 8-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。

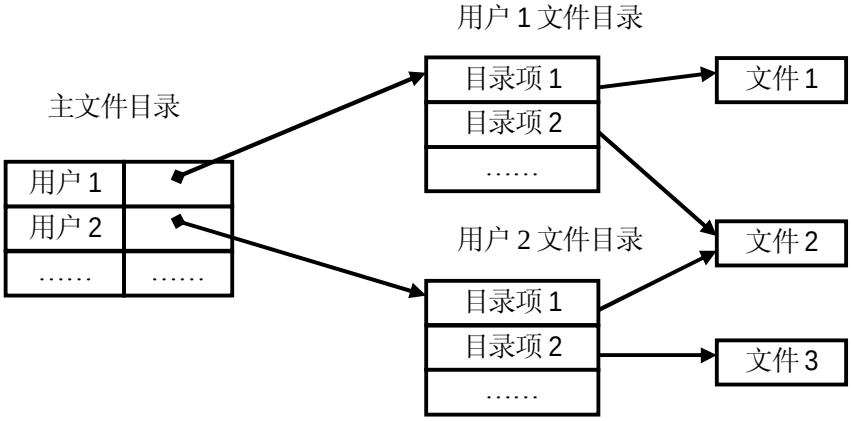


图 8-2　二级目录结构示意图

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。

8.3.4　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 8-3 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。

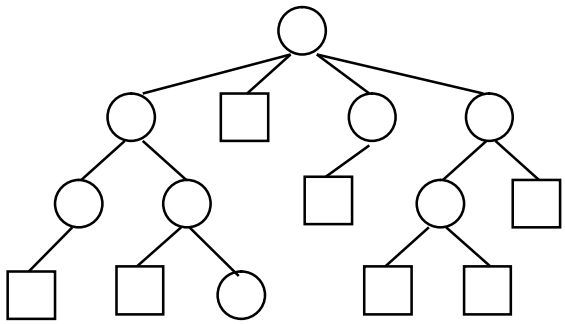


图 8-3　树形目录结构示意图

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。

8.4　文件组织与数据存储

8.4.1　文件的存储

1、卷和块

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。

文件的存储结构密切依赖于存储设备的物理特性，下面介绍两类文件存储设备。

2、顺序存取存储设备的信息安排

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 8-4 所示。

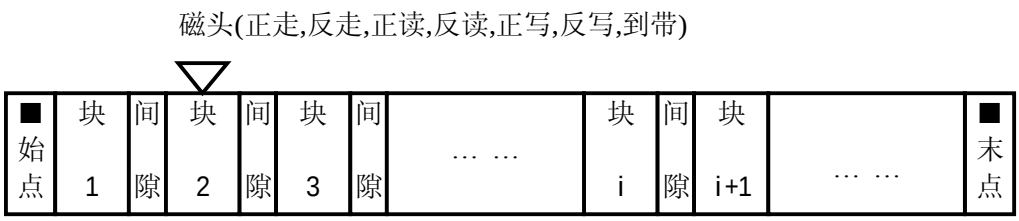


图 8-4　磁带的存储示意图

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。

3、直接存取存储设备的信息安排

磁盘是一种直接存储设备，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 8-5 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。

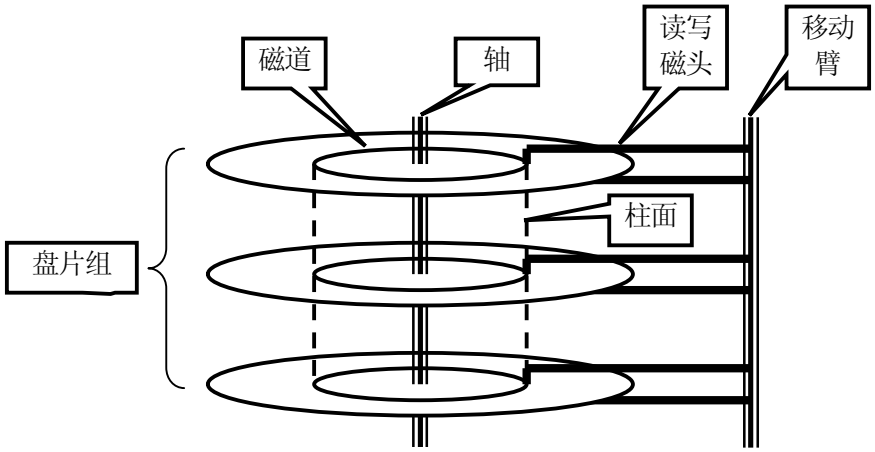


图 8-5　磁带的存储示意图

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。

8.4.2 文件的逻辑结构

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。

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 8-6 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。

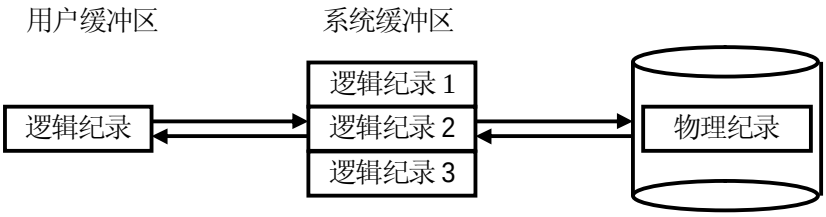


图 8-6　记录成组和分解的处理过程

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。

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 8-7 表明了三种记录之间的关系。

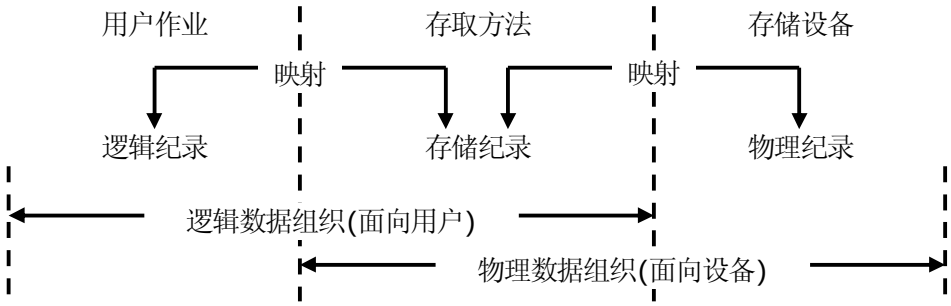


图 8-7　逻辑记录、存储记录和物理记录之间的关系

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。

8.4.3 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。

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。

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 8-8 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。

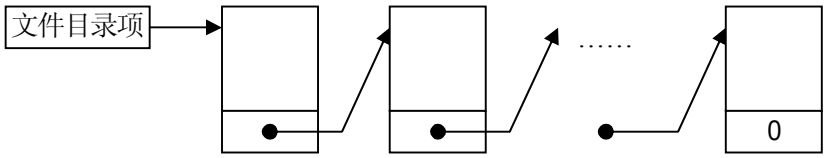


图 8-8　连接文件结构示意图

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。

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。

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 8-9 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。

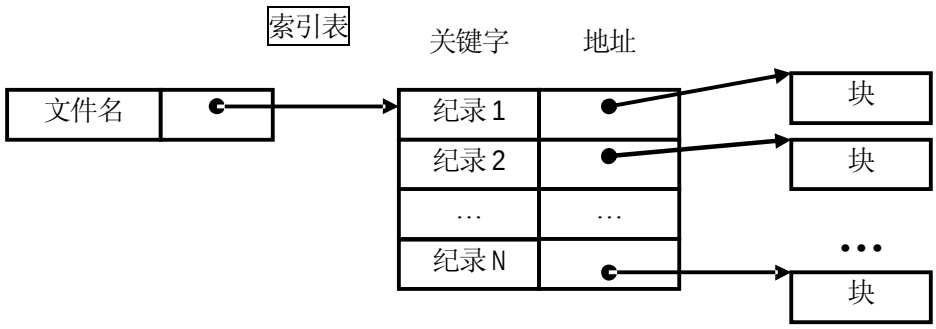


图 8-9　索引文件结构示意图

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。

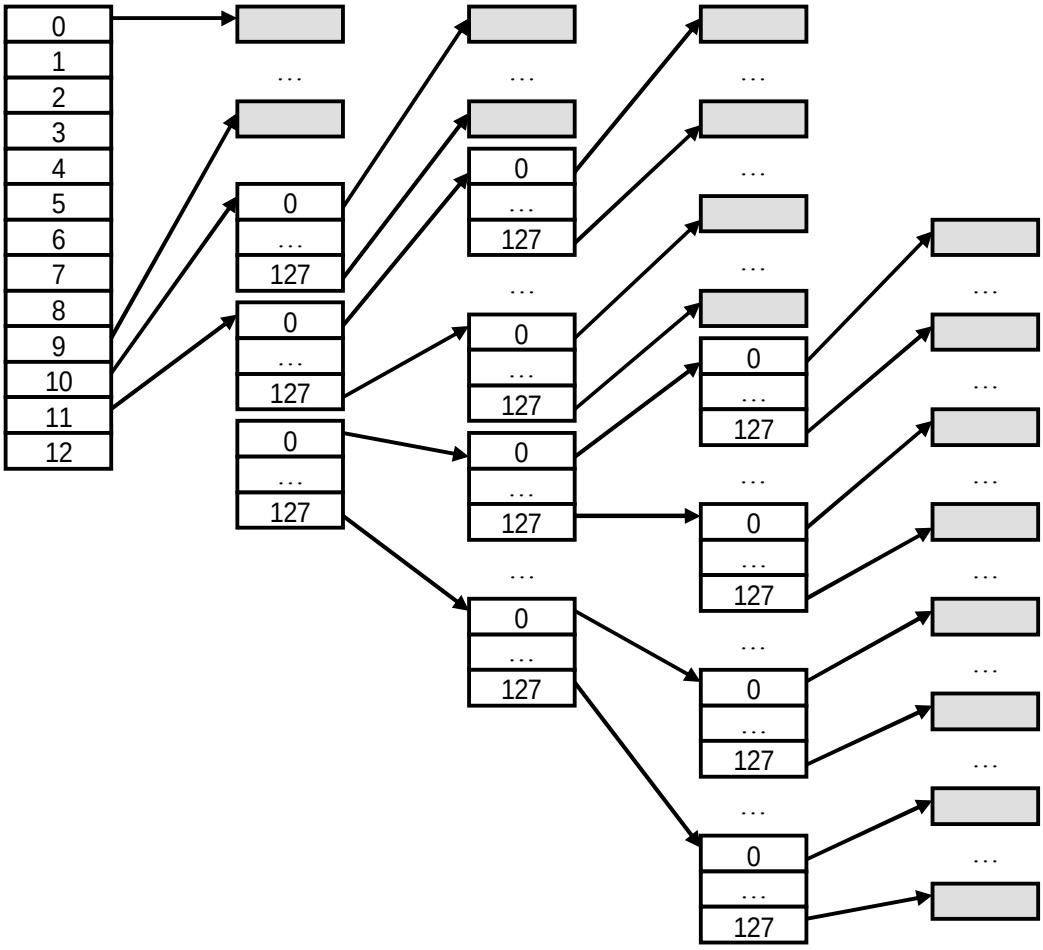


图 8-10　Unix 的多重索引结构

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。

Unix 操作系统的多重索引结构稍有不同。如图 8-10 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 Unix 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。

8.5 文件的保护和保密

8.5.1 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 8-11 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 8-11　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 8-12 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 8-12　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 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。

8.5.2 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。

8.6 文件系统其他功能的实现

8.6.1 目录的查找和打开文件表

当进程要访问文件资源时，首先必须打开文件，把文件目录项种的有关信息调入打开文件表。每个进程都会有一个打开文件表，连接在进程控制块上，存放已经打开的文件的有关信息。在很多操作系统中，这个打开文件表中的表目是有限的（一般默认为 16 项左右，有的操作系统允许调整这一数目），也就是说一个进程最多同时打开规定多个文件，当要继续打开新的文件时，有的操作系统将禁止，有的操作系统将自动关闭最先打开的文件。几乎所有的操作系统都默认打开文件表的前三项被标准输入、标准输出和标准错误输出三个设备文件占用。

如何打开一个文件，其主要工作就是目录查找，即找到相应的目录项将其信息调入打开文件表，以北访问文件时使用。‘按名存取’文件实质上就是系统根据用户提供的文件名来搜索各级文件目录，直到找到该文件。一级目录结构采用顺序查找法，依次扫描文件目录中的目录项，将目录项中名字与欲查找的文件名相比较。树形目录结构根据树形结构逐级查起，为了节省时间也可以从‘当前目录’查起。

Unix 的文件系统就采用这种方式，例如，要查找用户定义的一个文件 user.h, 根据客观存储安排，其文件路径名为/user/include/user.h 。第一个/表示从根目录查起，找出根目录中的各目录项与 usr 相比较，直到找到目录 usr；再取出 usr 中各个目录项与 inclde 相比较，依次下去，直至找到 user.h 文件。为了提高查找速度，可以设置“当前目录”，即把工作目录从根目录或某级子目录改变到离查找目标较近的那个目录，从那儿开始查找。在上例中工作目录可以改为 user/usr.h，就可能非常快地找到 usr.h。

现代操作系统都设置有改变工作目录命令（Change Directory)这给用户在查找文件时提供了很大方便。为了加快查找目录的速度，还可以采用其它办法。如当目录表项是按键的顺序编排的，则可以采用‘二分查找法’。另一种常用办法是‘杂凑法’，把每个文件名经过变换文件名作函数的变换就获得目录表表项。

在 Unix 系统中，索引节点是文件系统处理和管理文件的主要数据结构，它们通常存在文件存储器中。为了加快文件访问速度，主存中开辟了一个索引节点缓冲区——活动索引节点表，它能暂存 100 个索引节点项。活动索引节点表的主要内容从外存的索引节点中复制，此外，它还加进了一些自身的控制信息，如活动文件标志 I-Flags，

节点所在设备名， I-degv 节点在辅存索引节点区的编 I-no，节点访问计数 I-count，及读出的上次块号， I-lasrt 等。

活动索引节点是 Unix 文件系统的一种宝贵资源，经常进行寻找与释放活动 I 节点操作，其实现过程大致如下：

1、I 节点的寻找（iget）

该过程根据所提供的参数设备号（dev）和节点号（I-no）,先检索活动 I 节点表，若能找到所需节点，则将其访问计数（I-conut）加 I，表示要求共享；若别进程正在使用此节点（ILOCK）放此 I 节点（ILOCK=0）时，才被‘唤醒’，从而，获得该 I 节点。同时也设置锁标志（ILOCK=1），以防其它进程插入。

若活动 I 节点表中找不到所需节点，表示该 I 节点还没有进程使用它，需要按设备名和 I 节点号，把含该 I 节点信息的物理块读入内存，然后，把 I 节点项有关内容复制活动 I 节点表的第一个空闲项里，并且，将访问计数（I-conu）加 I，至此得到了所需 I 节点。

2、i 节点释放（ippui）

该过程通常把 I 节点的共享计数（1-conut）减 I；若没有用户使用（I-count=0），也没有连接用户（I-nlike=0）,则删除该文件；否则说明还有用户需要使用（I-nlike=0），删除该文件；否则说明还有用户需要使用（I-nlike=0）,尽管当前没有使用者（I-conut=0）.这时，不能删除该文件，而且若已被修改过，还要将它复制回外存的盘区。同时，I 节点号送入空闲索引节点栈，标志文件卷上相索引节点为空闲（I-mode=0）。

有了文件目录、索引节点和活动索引节点，在执行打开文件操作时，系统就可以快速、有效地进行目录查索了，假如要查找路径名为 usr/include/usr.h 的文件，系统初启时，把根目录文件的(I-no=1)复制到内存活动 I 索引节点表中，并用针 rootddr 指向它。

- 第一步：从根目录查起，根据 rootddr 指向的活动 I 索引节点中的 I-addr[ ]，把对应物理块中的根目录信息读到设备缓冲区中，文件路径名的第一分量 usr 依次与缓冲区中每个目录项比较，必要时可读入多个物理块，直至找到 usr 为止。
- 第二步：找到 usr 后，再根据这个目录项的 I-on，把相应的 I 索引节点取到活动 I 索引节点表中。然后，再由这个活动 I 节点的 I-addr[ ]来重复上述过程，直至找到 include，并把其对应的索引节点复制到内存活动 I 索引节点表中。
- 第三步：依次重复上述步骤，直至找到 user.h 把它所对应的索引节点复制到活动 I 索引节点表中，用一个指针 IP 指向它，以备读写文件之用。

8.6.2 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。

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。

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；
- 若打开的是共亨文件，则应有相应处理，如使用共亨文件的用户数加 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。

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。

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。

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。

8.6.3 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。

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。

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。

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。

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。

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。

8.6.4 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。

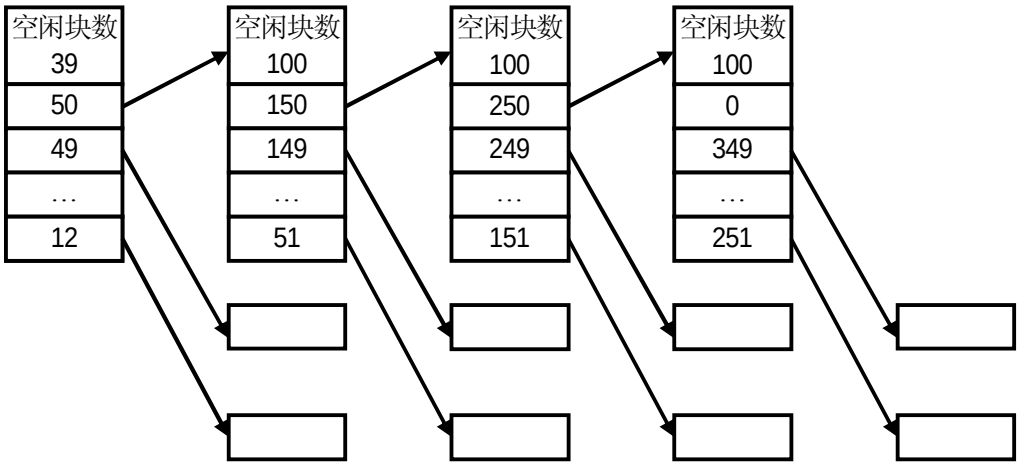
下面介绍常用的几种具体辅存空间管理方法。

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。

2、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。下面以 Unix 操作系统大量的空闲块。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 ç è (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 8-13　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。

3、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。

8.7　实例研究：Windows 2000 文件系统

8.7.1　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。

8.7.2 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 8-14 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。

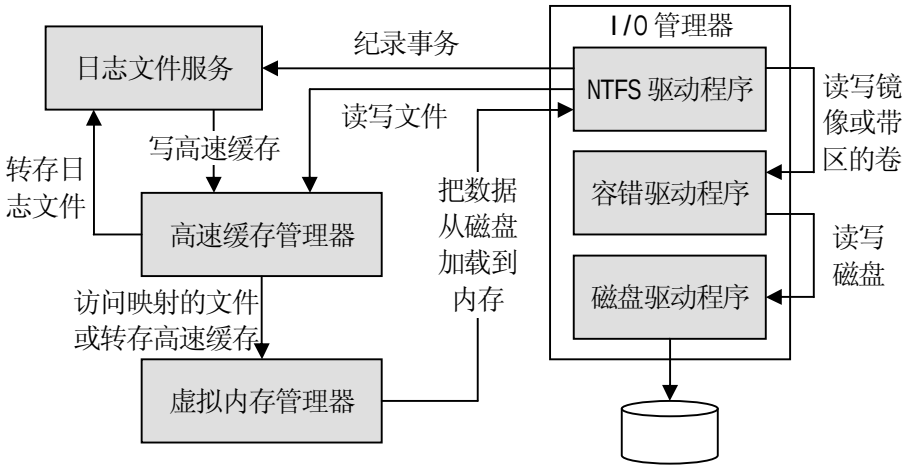


图 8-14　NTFS 及其相关组件

另外，图 8-14 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。

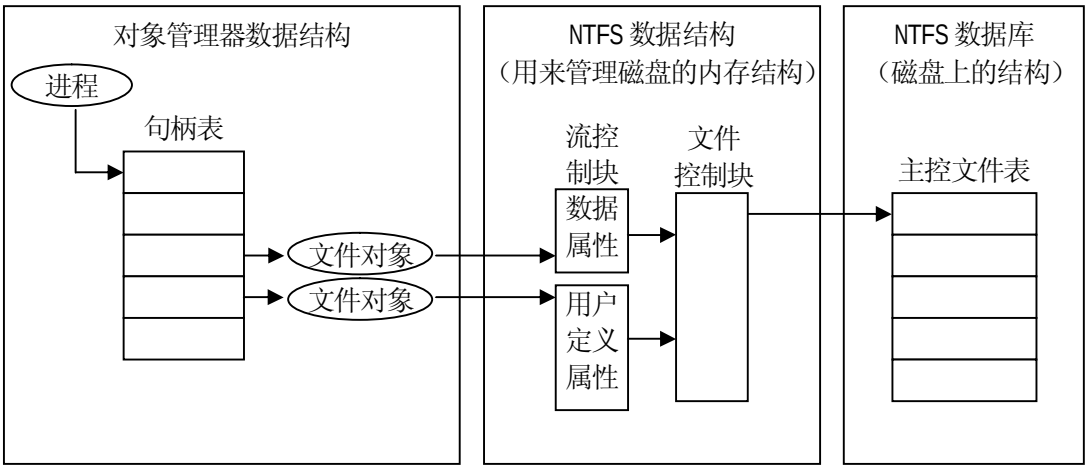
图 8-15 显示了将文件句柄链接到文件系统在磁盘上结构的数据结构。

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 8-15　NTFS 数据结构

8.7.3　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。

MFT
MFT 副本
日志文件
卷文件
属性定义表
根目录
位图文件
引导文件
坏簇文件
……
用户文件和目录 ……

图 8-16　MFT 中 NTFS 元数据文件的文件记录

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 8-16 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。

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。

CH9　操作系统安全性

9.1　安全性概述

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。

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。

下面讨论操作系统以及计算机系统中保证系统安全性的主要手段。

9.2　隔离

9.2.1　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。不同状态下只能使用不同机器指令，限制用户使用容易造成系统混乱的那些机器指令，从而，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。

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 9-1 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。

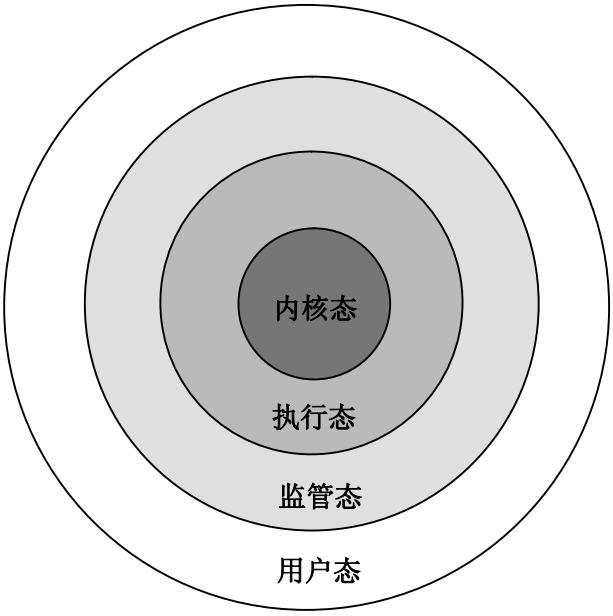


图 9-1　VAX/VMS 操作系统保护环

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。

9.2.2　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。

9.3　分级安全管理

9.3.1　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。

9.3.2　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。

9.3.3　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。

9.4　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。

9.5 信息安全管理

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。

9.6 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。

9.7 实例研究：Windows2000 的安全性

9.7.1 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。

9.7.2 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。

9.7.3 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。

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。

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、DdeImpersonateClient、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。

9.7.4 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。

9.7.5 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。

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)。

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。

9.7.6 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问的交互性。

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。

9.7.7 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。

9.7.8 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。

9.7.9 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。

9.8 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。

CH10　死锁

10.1　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次资源，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 工享。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
请示打印机	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 O P 占用了读机，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，它又请示读卡机，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等，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上资源，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 • 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而引起的死锁

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。

10.2　死锁的定义

死锁可能是由于竞争资源产生，也可能是由于程序设计的错误所造成的，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、等待外围设备舆．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：该组中每一个进程都在等待被另一个进程所占有的、不能抢占的资源。例如，n 个进程 P1、P2……Pn 是，Pi（i=…n）因为申请不到资源 Ri 而处于等待状态，而 Ri 又被 Pi+1（i=1,…n-1）占有，Rn 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。

10.3 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。

10.4 死锁的防止

10.4.1 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： 1 ）进程互斥使用资源。任一时刻一个资源仅为一个进程独占，若另一个进程请示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。 2 ）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。 3 ）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。 4 ）存放在一个循环等链，其中，每一个进程分别自己等待它前一个进程所持有的资源。

只要能破坏这四个必要条件一，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，但有许多资源往往是不能同时访问的，所以这种做法行不通。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，而不适用于所有资源。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。

10.4.2 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，因而被许多操作系统采用。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。

10.4.3 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：

$$r1, r2\cdots\cdots, rm$$

规定如何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不会发生死锁。事实上，若在时刻 t1，进程 P1 处于记录均等资源 rk1 的状态，则 rk1 必为另一进程假定是 P2 所占用。若 P2 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 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 必为前面的进程 P1 占用。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：

$$k2 < k3 < \cdots < kn$$

但是，进程 Pn 是占有 rkn 申请 rk1，那么，必定有：

$$kn < k1$$

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。例如，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。

10.5 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。本节我们将讨论几种死锁避免的方法。

10.5.1 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，他向一群客户分别承诺了一定的贷款额度。在图 10-1（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。这里将客户比作进程，贷款比作设备，银行家比作操作系统。

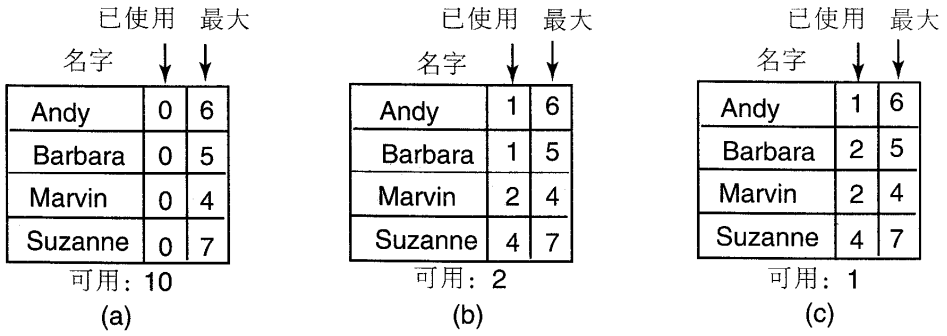


图 10-1 三种资源分配状态 (a) 安全 (b) 安全 (c) 不安全

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 10-1（b）所示。客户已获得的贷款（已分配的磁带机）和可用的最大数额贷款称为与资源分配相关的系统状态。

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上）。图 10-1（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 10-1（b），则我们得到如图 10-1（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，检查如果满足它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。

10.5.2 资源轨迹图

以上算法描述了一种资源的情况（例如仅有磁带机或仅有打印机，而不是多种资源）。在图 10-2 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。

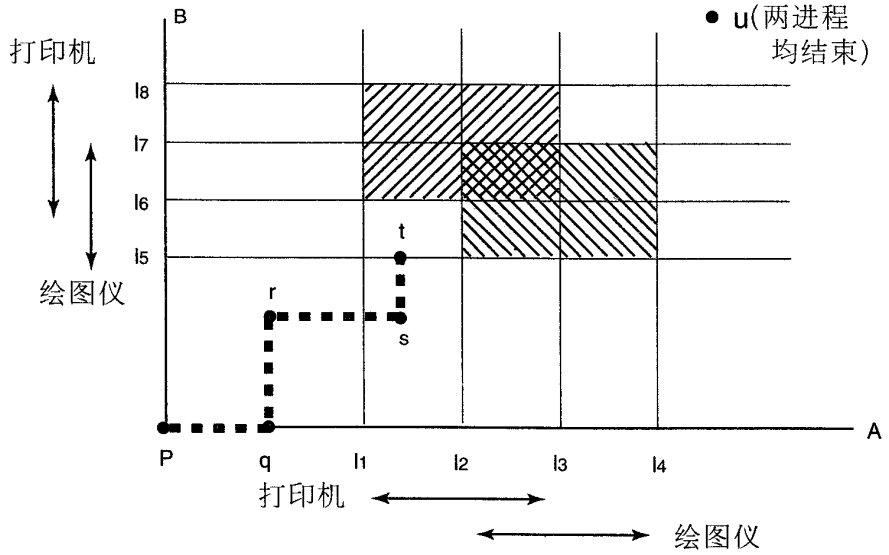


图 10-2　两个进程的资源轨迹图

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。

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，此时就发生死锁。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。

10.5.3 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 9-3 示出了其工作原理。

在图 10-3 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 10-3 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。

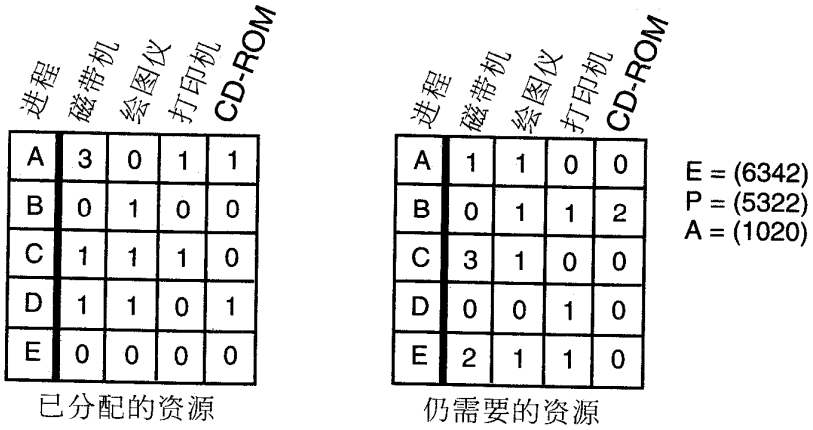


图 10-3　多种资源的银行家算法

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 9-3 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有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。

10.6 死锁的检测和恢复

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限制，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。

实行这种检测的一种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

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PI等待 Pk 所占的资源且Pk等待Pj所占的资源时取值为 1，也就是说 Pi 与Pj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。

当死锁被检测到后，最简单的办法是结束所有进程的执行，并重新启动操作系统；第二个办法是结束所有卷入死锁的进程的执行；第三个办法是一次结束卷入死锁的一个进程的执行，然后在每次结束后再调用检测程序，直到死锁消失；第四个办法是重新启动卷入死锁的进程中抢占资源，再这些资源指派给卷入死锁的其余进程之一，然后恢复执行；第六种办法是在理论上应用的，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。

若采用重新启动进程执行的办法，那么恢复工作应包含必须重启动一个或多个进程，心脏从哪一点开始重启动。一种最简单的办法是从头开始启动，但这样的代价是相当大的。有许多操作系统在进程执行过程中，但这样的代价是相当大的。有许多操作系统在进程执行过程中，对它设置校验点，这样，当该进程需要重启动时，就可以从这个校验点开始重执行。设置校验点不仅为了从死涣中恢复，更主要的是为了从错误中恢复，这对执行时间较长的作业和在实时系统中最很必要的。

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。

10.7 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，……，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，…，Rr

和 m 个无关的资源组：

M1，M2，…，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 … U Mm U R1 U R2 U … 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。

CH11　实时任务管理

11.1　实时操作系统概述

实时操作系统是操作系统的一个重要分支，是实时控制计算机软、硬件系统的核心。它随着实时多任务计算机系统软件的形成而诞生，随着实时多任务系统要求的提高而发展。以数字计算机为中心的实时多任务操作系统已经在工业、交通、能源、银行、科学研究和科学试验、国防等各个领域发挥了极其重要的作用。

11.1.1　实时操作系统的基本概念

实时操作系统 RTOS（REAL-TIME OPERATING SYSTEM）是操作系统的一个重要分支。它应属于操作系统的研究范畴。实时操作系统与通用操作系统有共同的一面，但在功能、性能、安全保密及环境适应能力等方面，还有其独特的一面。

实时操作系统是指具有实时特性，能支持实时控制系统工作的操作系统，它可将系统中的各种设备有机地联系在一起并控制它们完成既定的任务。

实时操作系统的首要任务是利用一切可利用的资源完成实时控制任务，其次才着眼于提高计算机系统的使用效率。

实时操作系统的一个重要特点就是要满足对时间的限制和要求。在实时系统中，时间就是生命，这与通用操作系统有显著的差别。

除个别系统外，实时操作系统都就是多道程序的操作系统。

11.1.2　实时操作系统的基本术语

1) 系统：一个系统是一个完整而相对独立的整体，具有特定的功能，并由某些相互作用或相互依赖的分系统组成。例如：实时操作系统、软件系统或计算机系统等。

2) 任务（TASK）与进程（PROCESS）：在实时操作系统中，任务与进程的含义相同。它们既是用户程序的基本单位，也是系统程序的基本单位，甚至包括操作系统本身在内。

3) 任务换道时间（CONTEXT-SWITCHING TIME）：操作系统处理任务换道所需要的时间称为任务换道时间。它是指从任务停止工作的时刻开始到另一任务开始工作的时刻为止所历经的时间。因为任务换道操作受各种条件及偶然因素的制约，故任务换道时间不一定是恒定的，即使在同一台机器上，对同一任务的换道也是如此。

4) 中断延迟（INTERRUPT LATENCY）：从计算机接收到中断信号的时刻开始到操作系统响应该中断、完成换道操作特效药转入相应的处理程序为止，这段时间就称为中断延迟。在具体系统中，中断延迟这个量变化范围较大，由于情况复杂，因此测量也比较困难。在实时操作系统中，中断延迟是一个重要的系统参数。

5) 系统响应时间（SYSTEM REPONSE TIME）：系统响应时间是指从向系统发出处理要求起到系统给出某些应答信号为止所经历的时间。系统响应时间主要包括：输入信息及排队等待时间，信息处理及等待时间。

6) 同步（SYNCHRONIZATION）

所谓同步是指系统严格按某种信号或预定规律而顺序地工作。例如，同步 I/O 操作是指用户程序发出读/写请求后，一直要等到本次 I/O 操作执行完，程序才能继续执

行 I/O 请求（即该读/写请求）后的指令。

7) 磁盘存取时间（DISK ACCESS TIME）：从系统发出访问磁盘记录的请求开始，到从磁盘读出记录并开始向主机传送为止，这段时间称为磁盘的读出时间，从主机发出磁盘写记录的请求开始，到记录写入磁盘并向主机给出回答信号为止，这段时间称为磁盘的写入时间。磁盘的读出时间和写入时间统称为存取时间，其中包括磁头定位时间、旋转等待时间和读出/写入时间等。

11.1.3 实时数字控制系统

实时数字控制系统是实时操作系统的主要生存环境和工作环境，实时操作系统控制实时系统的各种设备共同完成任务。

实时控制系统属于控制系统的研究范畴。从功能而言，控制系统可被定义为对能量或其他媒体流量等进行调节的装置。

实时控制系统是一种能接收数据、加工处理并可将处理结果及时予以反馈的环境控制系统。

实时控制系统由以下四大部分构成：

- 数字采集。它用来收集、接收或录入系统工作所必要的信息，或进行信号检测；
- 加工处理。它对收集、接收或录入的信息（包括信号检测的结果）进行加工处理，得出控制系统工作所必要的参数或作出决定，然后进行输出、记录或显示；
- 操作控制。它根据加工处理设备所输出的信息（包括输出信号）采取适当措施或动作，以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接收设备，以便系统根据反馈情况进一步采取措施，达到控制的预期目的。

实时控制系统的工作过程见图 11-1。

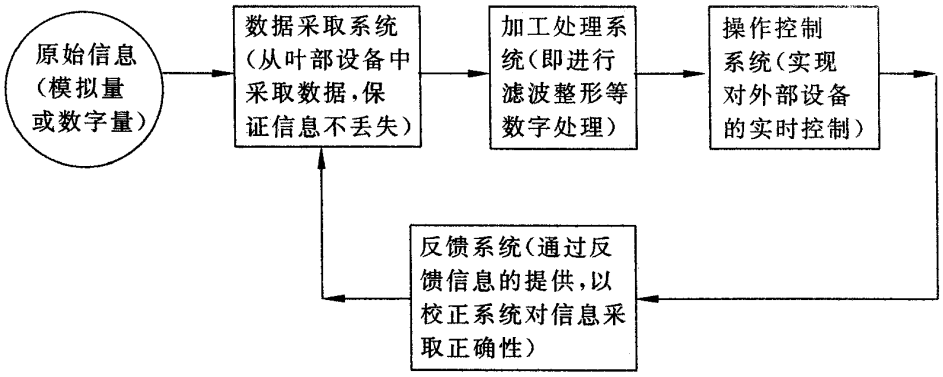


图 11-1　实时控制系统的工作过程

实时数字控制系统是电子数字计算机和实时控制系统相结合的系统。实时数字控制系统具有两大特征：数字控制和实时工作。

数字控制的意义是：实时数字控制系统以电子数字计算机为中心环节，全系统的信息都以数字化的形式输入到计算机中。如果系统中有一些参数是以模拟信号（电压、电流等）的形式给出的，那就要经过模拟-数字（A/D）转换器，将模拟量变换成数字

量。计算机根据当前最新的输入数据，参考历史记录及各种客观规律和约束条件，加以综合和优化处理。这种处理是以数字处理、逻辑处理或其他数字信号处理为基础的。

实时数字控制系统的分为以下几类：

(1) 实时控制系统（Real-Time Control System）这种系统接收由环境发来（或采集）的数据并加以实时处理，再将处理结果迅速地反馈给该环境，从而控制该环境工作。这种系统还可细化为过程控制和操作控制两类。

过程控制是指在给定的系统响应时间的条件下，实时控制计算机根据采样数据和数字（或物理）模型进行处理，实现系统的最优化控制。它又可进一步分连续控制和顺序控制两种。

其一为连续控制，从被控制对象直接读取数据，将其与希望或额定的数据进行比较，将比较所产生的输出数据直接送给被控制的对象，力图达到理想的状态或结果。

其二为顺序控制，把一个大的目标分解成多个小目标，每步的控制工作皆实现一个小目标，各步控制目标严格按顺序进行，从而实现总的控制目标。

操作控制是指环境和计算机之间不要求自动反馈处理，但需要人工干预，计算机只用于改善人工操作（如加快速度、提高精度等）。例如飞机、火车的订票系统。宾馆的事务管理系统，银行终端服务系统及邮电服务系统等，都属操作控制系统之列。这些系统本身要求操作人员和计算机之间必须进行通信，但对系统响应时间不做苛刻的要求，只要恰如其分即可。

(2) 数据采集系统（Data Acquisition System）

这是一种从环境中实时录取数据但不要求实时处理并输出处理结果的系统。在大多数情况下，数据采集工作是实时控制系统工作的一部分，但它本身有独立的意义。如石油勘探地震数据录取系统、火箭或航空发动机试验数据采集系统，以及气象观测或气象数据采集系统等。

(3) 指挥和管理系统

不利用计算机进行管理控制而只是利用计算机为指挥员和管理员提供判断信息。

11.1.4 实时操作系统的特点

实时操作系统的特点可以从性能、功能等不同的侧面进行分析研究，我们这里仅从性能方面加以探讨。

1) 系统的正确性

系统的正确性是系统存在的基础，因此是首要的和必须保证的。

2) 系统的实时性

系统的实时性是由系统响应时间的长短来标志的。系统响应时间应该有一个上限，即系统的每次响应时间都不会超过该数值。

实时操作系统应能保证在规定的响应时间内响应并处理异步事件的请求。

3) 高度的可靠性

实时操作系统的可靠性包括两个方面的内容：一是在正常情况下系统正确地连续工作妈连续性；二是在异常情况下系统能及时正确处置，保证完成任务或最重要的任务，即系统的可保护性。

系统工作的连续性是指在规定的一段时间内系统始终处于良好的工作状态。为实现这一目标提出了很多方法，如采用容错技术或采用多机系统等。

4) 系统的安全性

安全性的含义是系统本身有防护信息泄露和防止信息破坏的能力。系统安全首先是信息安全。为此，应将系统中的信息分类，不同类的信息可以分别采用不同的安全措施。

系统内部信息的安全性可通过备份和后援存储的方法实现。

5) 任务处理的随机性

绝大部分实时操作系统都有及时处理随机事件的能力。在任一时刻，系统都不应丢失信息和降低实时性。

在研究和设计实时操作系统时，要充分估计随机事件出现的密集度，并定出系统的额定负载和极限负载。操作系统在具体处理这些可能同时出现的随机事件时，应根据其轻重缓急按一定算法定出其俦级，然后依次排队处理。

在额定负载下运行时，实时控制计算机系统一定要保证系统资源至少有 20%的余量，以应付随机事件的出现。

6) 输入/输出的复杂性

在实时控制系统中，输入/输出设备数量可能较多，类型各异，因而相应的计算机通道和接口可能有同步型的、异步型的和通用或特殊型的，它们传输速度可从每秒几个二进位到每秒几万个二进位数，甚至更高。

7) 友好的人-机界面

人与计算机之间常常存在着一种相互依赖的关系，并处在一个统一体中。主动性和创造性的工作往往依靠人来完成，而大量重复的信息处理则依靠机器。人机结合的渠道就是人-机界面。

8) 易读性和易移植性

在实时操作系统和实时系统应用软件的开发过程中，程序的易读性是关键问题之一。软件在开发完成后，最经常、最重要的工作就是使用和维护。这些工作的难易往往与易读性紧密相关，并决定了系统的命运。

为了保证实时操作系统的易读性和易移植性，写操作系统时要尽量采用高级语言。

11.1.5 实时任务

一个作业中可以并行执行的程序段，而这些程序段一般具有实时特性，称为实时任务。

1、标识号

用户作业可包含多个并行执行的任务，任务标识号用来标识任务，起到任务名的作用。它的取值范围为 0~255，两个任务不能有相同的非零标识号，但唯独标识号为零的任务可以有多个。

2、实时任务的状态

(1) 运行状态：任务获得处理机，正在运行。

(2) 就绪状态：只要获得处理机，任务即可运行。

(3) 挂起状态：由于某种原因，任务即使得到处理机也无法继续执行，只有待此种原因撤销后，任务才有获得处理机的权利。

(4) 潜伏状态：任务建立之前和撤销以后的状态。

3、实时任务的优先级

任务的状态是调度时决定任务能否被选中投入运行的主要依据。就绪任务取得处理机的先后次序是由任务的优先级来规定的，任务的优先级反映了它想获得其执行结

果的急切程度。一般情况下，任务调度程序总是挑选优先级最高的任务运行。任务优先级范围为 0~255，其中 0 优先级为最高，不同任务可以有相同的优先级。在创建一个任务时，必须指定其优先级。若未指定，系统就自动取当前任务的优先级为新任务的优先级。

4、任务控制块

每个任务都包含了一组程序，不同的任务其内部操作各不相同。从任务系统来看，主要关心的不是每一个任务的具体功能和内部操作，而是任务的外部特性，如任务的名字、当前状态和优先级等。依据这些描写任务外部特性的信息，任务系统实现对任务的控制和管理。任务的所有外部特性信息集中在一起，便称为任务控制块 TCB（Task Control Block）。

TCB 是表征任务存在的唯一实体。当任务系统收回了任务的 TCB 后，任务就被撤销了，当建立一个新任务时，任务系统为它分配一个空的 TCB，并填入此任务有关的外部特性信息，如启动地址、任务标识号、优先级等，这时任务便存在于系统之中。同样，任务系统对于各个任务进行调度、控制和管理，也都是根据每个任务的任务控制块进行的。总之，任务控制块是任务的标志，任务系统根据任务控制块而感知任务的存在，各种任务调用命令的操作对象就是任务的任务控制块。

任务控制块 TCB 也是用户任务与操作系统交换信息的通信区。当用户任务向系统申请系统资源，如内存储区、输入输出设备时，需将有关申请请求的内容填入任务控制块 TCB 之中。然后再提交给系统，而系统在完成用户提出的某些申请或操作后，也是将有关返回信息送入任务的任务控制块，而由任务从任务控制块中获取信息。

任务控制块的主要内容包括用户任务的运行现场信息，任务本身的外部特性信息和管理任务控制块的有关信息三部分。其内容如下表所示。

任务控制块 TCB

类　　别	主　　要　　内　　容
现场信息	保存用户任务运行现场信息中各寄存器或累加器的内容 保存用户任务程序计数器的内容
任务外部特征信息	用户任务标识号 用户任务状态 用户任务优先级
管理控制信息	任务调用命令字 任务与系统通信区 任务控制块连接字 其　　它

5、任务控制块队列

用户作业内的任务数由用户指定，系统装配程序根据任务数来分配任务控制块空间。系统一般将处地就绪、运行和挂起状态的所有任务的 TCB 借助于 TCB 中的连接字串成一个链，称为就绪队列或活动链。该队列的首地址一般存于用户作业的作业控制表中。而队列中正在执行的任务 TCB，也称当前活动 TCB。

各任务 TCB 是按任务的优先级先高后低的顺序排列的，相同优先级的 TCB 则按任务建立先后次序排列。所有空 TCB 一般也串成一条链，称为空队列或静态链。

任务控制队列如图 11-2 所示。

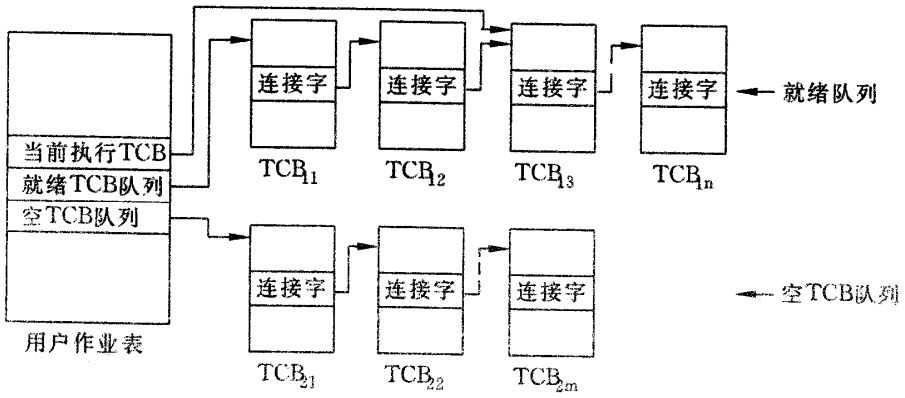


图 11-2　任务控制队列

11.2　实时任务的设计

实时操作系统应包括以下基本功能：实时任务之间的同步与通信、定时任务和延迟任务的管理、用户及系统实时 I/O 管理、以及多任务调度。除任务调度外，各种功能均是以任务调用命令形式提供给用户使用的，本节重点介绍实时任务之间的同步通信与定时任务和延迟任务的处理。

11.2.1　实时任务之间的同步通信

从概念上讲，组成作业的任务具有逻辑独立性，而且能并行执行。但是有时也要求它们协同运行。这就是说，某些任务之间在关键点上可能需要互相等待或交换信息，这种诸任务间相互制约的等待和交换信息称为任务之间的通信。除此之外，多任务系统还允许任务与系统控制台交换数据，操作员通过控制台灵活地使用有关通信命令来查询和改变任务状态，执行某些任务或者按一定周期来执行某些任务。

1、实时任务之间的通信

多任务系统提供发送和接收两条任务命令来实现两个任务之间交换一个非零信息字的通信，这里预先规定了一个公共通信字。一个任务将信息送入此通信字，另一个任务从此字中取走信息。若两个任务要交换一批信息，那么可把信息存放区的首地址作为传送的非零信息字，不过事先得规定信息存放区长度。

在多任务系统之间的通信可分为同步通信和异步通信两种方式。

异步通信具有五种基本功能：置通信请求，取通信请求，置回答，取回答，多次回答。同步通信的功能是单一的，发出通信请求后立等通信回答，只有收到通信回答后本次通信才告结束，即当发送信息字的任务正在将信息送入公共通信字中时，不允许接收信息字的任务去取信息字。在传送任务结束后，发送信息一方应该给接受一方发一信号，之后接受方才可以去取通信字内的信息，但在这之前接受方必须等待。另一方面，在接受一方未取走公共信息字中的信息前，传送一方不得将新信息送入通信字，以免破坏另一方尚未取走的信息，此时也要求同步。这样，传送一方和接受一方为了完成一次信息接送工作，必须有两次同步。

2、多任务通信的特点

在研究实时多任务操作系统的通信机构时，应注意实时控制任务之间通信的特点。同步通信和异步通信两种手段都应该有，但绝大多数情况都应该应用异步通信。这是因为，同步通信万一用不好，会带来实时性降低、甚至死锁的问题。多任务通信的特点是：

- 严格的时间—空间关系：任务的执行周期和它应处理的输入或采样信息、输出或控制信息在时间上要对应，甚至在输入/输出缓冲区的位置上也要对应。
- 适应不同执行周期任务的需要：相互通信的诸任务可能有相同或不同的执行周期，而且各有不同的执行优先级，因而要受其约束。
- 通信关系一次建立长期有效：实时控制任务执行的周期性，带来通信的周期性。周期性的通信最好一次请求建立关系长期有效，直到任务要求解除通信关系或任务执行完为止。
- 数据共享：几个实时任务可能要求数据共享，而且实时共享。

3、通信操作

两个任务 A 和 B 之间通信是这样进行的：设 A 为发送信息一方，B 为接收一方，在通信中，A、B 两任务发命令先后次序是不重要的。若 A 先发传送命令，则它将信息送入通信字后继续运行（或者根据不同传送命令要求将自己挂起），B 任务发接收命令后从通信字中取走信息（或任务 A 因为等待传送工作结束而处于挂起状态，则解挂任务 A）。若任务 B 先发接收命令，当它发现公共通信字为 0 时，表明任务 A 的信息尚未送到，于是将自己挂起等待 A 任务的信息，A 任务发送命令后发觉 B 任务已在等待，则直接将信息送给 B，且解挂 B。此时信息不再送入公共通信字。在 A 任务向通信字中传送信息时，如果公共通信字为非零，则表明上次发送的信息尚未取走作命令出错处理。

任务之间的通信命令有四条：

- 发送命令：此命令应提供两个参数，一个指定公共通信字地址，另一个存取非零信息字，发送信息的任务在发完命令后处于就绪状态。若接收信息的任务在发此命令前已被挂起，那么，命令处理解挂接收任务并且将信息送给优先级最高的等待任务。若接受方并未挂起，命令处理程序将信息字送入公共通信字。
- 发送并等待命令：此命令的两个参数发送命令，它与发送命令的区别在于：利用此命令可实现两个任务之间的同步。即若命令处理程序发现接收一方未发接收命令，则将信息字送入其通信字后，将当前任务变为挂起状态，等候接收任务发接收命令，当接收任务已发出接收命令时，发送方和接收方都变为就绪状态，控制转到任务调度程序。
- 用户设备中断服务程序使用的应该传送命令：除了系统设备外，操作系统还应该能管理用户自己定义的设备。例如用户自己配备的汉字显示器就是用户设备。用户设备中断服务程序使用本命令向用户任务传送数据。该命令功能与发送命令相似，区别在于该命令处理结束后仍继续执行用户设备中断服务程序。
- 接收命令：该命令的两个参数发送命令相同。命令处理程序若发现公共通信字为 0，则挂起发此命令的任务。当信息已在公共通信字内时，则取走

信息并且将公共通信字置 0。以便通知发送一方信息已经接收。或发现发送一方等待传输结束而挂起时，命令处理程序将解挂发送任务。

11.2.2 定时任务和延迟任务

1、定时任务

多任务系统具有实时处理功能，主要表现在当一个用户作业包含某些控制实时终端设备的任务时，系统能对外部信号作出及时响应，并使之与实时终端设备有关的任务能按一定时间关系和逻辑关系协调工作。这样的任务有下述特性：

- 任务需要按用户规定时间建立；
- 任务需要周期性地重复执行。

我们称这类任务为定时任务，实时任务将它与时间有关的信息记录在用户任务排队表 UTQT（USER TASK QUEUE TABLE）中。譬如 UTQT 可以包含 10 个字，其结构如表中所示。

用户任务排队表（UTQT）

控制字缩写符	意　　义
QPC	定时任务的启动地址
QNUM	重复执行次数，每执行一次，此数减 1
QTOV	当程序存储在磁盘中时，它给出了有关地址信息
QSH	任务起始建立时间(时)
QSMS	任务起始建立时间(秒)，任务每执行一次，QSH 和 QSMS 加上周期，形成新的起始时间
QPRJ	左字节为任务标识号，右字节为任务优先级
QRR	任务重复执行的周期（秒）
QTLNK	定时任务连接字，借助此字，定时任务的 UTQT 串成一列队链。称为定时任务排队链。

2、延迟任务

1) 延迟任务链

系统允许用户任务将自己挂起若干时间后再运行，这类将自己挂起若干时间推迟执行的任务称为延迟任务。

一个作业中所有延迟任务的 TCB 串成一个链，链中各延迟任务 TCB 的排队原则为：

① 根据任务延迟时间由小到大顺序排列；

② 延迟时间相同的任务的 TCB 的排列次序取决于任务请求排队时间的先后。

延迟任务排队链中的 TCB 内记录着有关此任务延迟时间的信息，具体为本任务延迟时间减去链中前一个任务延迟时间。

因此除了链首 TCB 中记录了相应任务的延迟时间外，其余 TCB 中记录的都是此任务相对于链上前一任务的延迟时间增量，延迟时间以脉冲数为单位。

2) 延迟任务的唤醒

用户任务规定了挂起时间将自己推迟执行，它的 TCB 即被挂到延迟链上，那么延迟链上的任务由谁来解挂呢？

时钟中断程序负责唤醒延迟链上的到时任务，将它们解挂，并从延迟链上撤下使之处于就绪状态。当多任务调度程序再次扫视就绪队列时，该任务就有可能被选中，

重新投入运行。时钟中断程序唤醒延迟任务的过程如下：它依次扫描延迟链，每当发现一个到时的延迟任务时，就将该任务的 TCB 从延迟链上撤下，并解挂此任务。

3、有关的任务命令

系统提供了一组任务命令，以便用户实现对任务的控制和管理，定时任务是通过建立定时任务命令建立起来的。

此命令的功能是将定时任务的 UTQT（用户任务排队表）插入到排队链上去，定时任务的排队原则是：

① 当用户任务排阶表中的起始建立时间迟于当前系统时间时，UTQT 按时间先后次序排队，相同起始建立时间的任务按优先级高低次序排列，优先级相同的任务按参加排队先后次序排队。

② 当任务起始建立时间早于当前系统时间时，此任务即为过时任务，它的 UTQT 挂在队尾，并按过时时间由多到少顺序排队。

建立定时任务命令仅仅解决了 UTQT 的排队问题，任务的建立是由建立程序来实现的。当系统时间发生变化且处理机控制转到实时处理程序时，建立程序由定时任务排列开始检查每一个 UTQT，凡是满足条件“上次扫视排队链时间<任务起始时间≤当前系统时间”的定时任务称为到时任务。建立程序为到时任务分配一个 TCB，并在其中填入此任务的有关信息（启动地址、标识号和优先级等），将这个任务控制块插到就绪队列上同优先级任务 TCB 的末尾。这样，定时任务就被建立且呈就绪状态，当多任务调度程序再次扫描就绪队列时，它就可能被选中投入运行。

定时任务建立程序还负责修改到时任务的 UTQT，这包括下述操作：

① UTQT 中重复执行次数控制字 QNUM 减 1；

② 修改任务起始时间，将 QSH 和 QSMS 另上任务重复执行周期 QRR，形成新的起始时间；

③ 若经过悠扬后，QNUM 不为 0，则按新建立的起始时间重新安排 UTQT 在排列链中的位置；否则将该 UTQT 从排队链上撤销，每撤销一个 UTQT，排队计数减 1。

从以上处理过程可见：用建立定时任务命令建立的定时任务的起始时间并不是中央处理机开始运行这个任务的实际时间，这个起始时间只不过是建立程序分配给定时任务一个 TCB，并且将控制块插入就绪队列的时间（当这个 TCB 被多任务调度程序选中时，任务才开始被执行）。同样，定时任务重复执行的周期也不是中央处理机运行此任务的重复周期，而是定时任务得到一个新的 TCB 从而重新得到调度机会的周期。因此，为了保证定时任务尽可能地按原定周期被中央处理机重复执行，用户应给予任务最高优先级。这样，当它的 TCB 被插入就绪队列时，就有希望放在链首，容易被多任务调度程序选中。

注意，就绪队列上定时任务的 TCB 在此任务执行一次后并不会自动撤销，用户必须使用相应的任务命令来释放它。

11.3 实时系统的实现

11.3.1 实时任务调度

从实时操作系统的设计观点出发，对实时任务进行调度的目的有二：一是保证各任务都能按其要求执行；二是充分发挥全部资源的系统效率，即利用系统的现有配置，使其资源的利用率尽量提高。这里所说的系统资源不仅包括计算机系统的系统资源，

还包括实时控制系统的其他设备和资源等。在这种控制系统中，第一个目的更为重要，而在非实时系统中，第二个目的更重要。

任务调度要解决的主要问题是在什么时候扫视就绪队列选择一个任务投入运行和选择哪一个任务，这一功能是由多任务调度程序完成的。在发生下列事件时，多任务调度程序扫视就绪队列，以挑选一个任务投入运行：

(1) 任何一个任务的状态发生变化；

(2) 任务优先级发生变化；

(3) 操作系统又将中央处理机控制权交给任务调度程序。

多任务系统采用最高优先级第一的调度算法。任务调度程序从就绪队列链首开始扫视，选择就绪任务中优先级最高的任务投入运行，由于就绪队列中 TCB 是按优先级高低顺序排列的，因此被扫视的第一个就绪任务必定有最高的优先级。为了使相同优先级的就绪任务有轮流得到处理机控制权的机会，多任务系统对它们采取了循环调度的策略，即将选中投入运行任务的 TCB 从就绪队列上原来位置移到同一优先级 TCB 的最末尾。这样，当任务调度程序下次扫视就绪队列时，此任务的 TCB 已成为同优先级任务 TCB 中的最后一个，从而也最后得到处理机。

本程序是多任务系统处理模块的核心，其程序实现流程图见图 11-3。

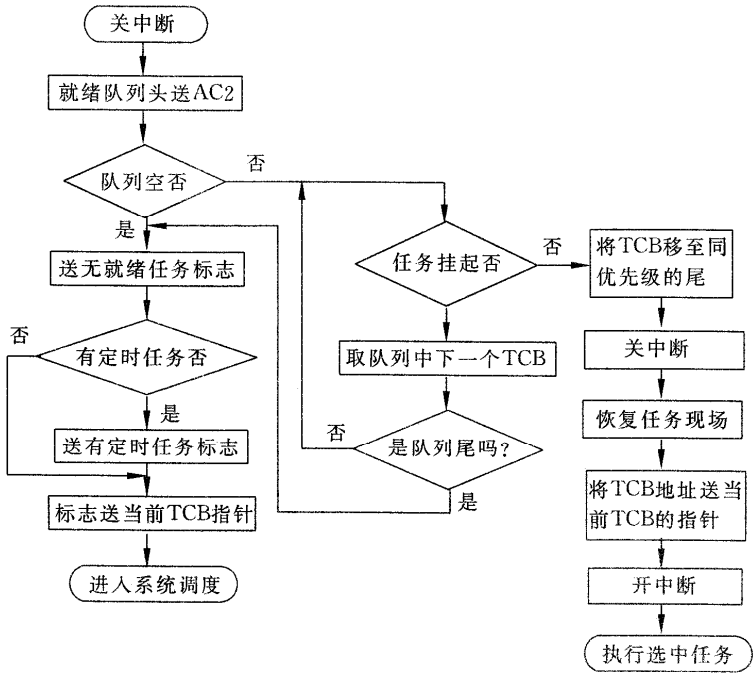


图 11-3　程序实现流程图

该程序在封锁中断状态下工作。因为就绪队列上 TCB 是程序的操作对象，各控制块中信息不允许在调度进行过程中发生变化，故进入程序立即封锁中断，以保证多任务调度程序执行过程中不会被打断。

程序依次扫描就绪队列上各 TCB，若发现处于就绪状态的任务，根据同优先级任务循环调度原则，选出当前的活动 TCB，调度程序负责恢复投入运行任务的现场，这包括有关状态字、指令计数器和各寄存器内容等。

11.3.2 实时任务命令的实现

在实时操作系统中，提供了多任务系统，实时任务命令根据对象的不同分为任务调用命令和系统调用命令。

任务调用命令是为用户编制实时多任务操作系统所提供的各种命令。它是以命令字及参数的形式给出的。

任务调用命令概括地说分为五类；改变任务状态命令、任务间信息通信命令、用户实时钟和系统实时钟命令、定时任务命令和实时设备命令。

1、改变任务状态命令

改变任务状态命令分为任务的建立命令、任务的撤销命令、改变任务优先级命令、挂起命令、任务的解挂命令、延迟命令等几类。

- 任务的建立命令（TASK）：主要完成在多任务系统中建立一个新任务，在整个用户作业输入到系统后，系统认为只有一个任务（首任务），其他任务必须通过首任务建立新的任务。通过任务建立命令时，可建立一个实时多任务系统。
- 撤销任务命令（KILL）：就是把使用 KILL 的任务变为潜伏状态。KILL 是撤销当前的任务。AKILL 是撤销当前任务及撤销参数所指定的优先级。
- 改变任务的优先级（PRI）：即动态更改任务优先级。
- 任务的挂起命令（SUSP）：强制性地通过挂起命令把用户作业挂起，挂起后，永远不能成为就绪状态，不能运行。在实时系统中用户作业可以自行挂起。
- 任务的解挂命令（ARIDY）：解挂已挂起的命令使之变为就绪状态。参数中要给出优先级。
- 任务延迟命令（DELAY）：将当前执行的任务推迟一定的时间以后再变为预备状态。

2、任务间信息通信的命令

用户任务之间信息通信的命令主要完成两个任务信息的通信，并可实现任务之间的同步。

其中包括以下几类：

- 传送信息命令（XMT）：当前正在执行的任务把一个非 0 的信息字插入指令的内存地址中。发出传送命令信息字的任务变为就绪状态。XMT 必须与 REC 配合用。
- 接收信息命令（REC）：将其它的任务传送到指定地址的内容取出，同时将其内容清 0。在接收信息时，首先看发送信息地址内容是否为 0，若不为 0，取走，REC 变为就绪状态；若为 0，则把 REC 任务挂起等待。当发传送命令时，看是否等待，有则解挂，使 REC 变为就绪状态。
- 同步传送信息命令（XMTW）：完成信息传送状态后，不是变为就绪状态，而是变为挂起等待状态，REC 需检查 XMTW 是否挂起，若挂起，使之变为就绪状态。所谓同步状态，指同时处于就绪状态而被处理机调度。

3、用户实时钟和系统实时钟命令

用户实时钟：受系统实时钟驱动时钟，可按用户指定间隔提供中断。用户钟中断处理程序是由用户作业自己编制的。

系统实时钟：指计算机内部硬件的时钟，每一周期中断一次。

用户实时钟和系统实时钟命令包括：定义用户实时钟命令（DUCLK）、取消用户实时命令（RUCLK）、检测实时钟频率（GRHZ）。

4、定时任务命令（QTSK）

就是使一个任务按用户的要求，按一定周期建立任务并重复执行该任务。

其参数要给出定时要求：

- 任务本身的信息：任务的标识号、任务优先级、任务起始地址等。
- 定时信息：执行任务次数，第一次执行的时间、间隔时间。

5、实时设备命令

实时设备命令就是实现对实时设备的管理为用户提供的接口命令。例如：登陆用户设备（IDEF）、删除用户设备（IRMV）。

11.3.3 建立任务命令程序实现流程

建立任务命令用来建立新的实时任务。该命令需要三个参数：新任务标识号和优先级、任务启动地址及要传送给新任务的信息。

建立任务命令程序实现流程图如图 11-4 所示。

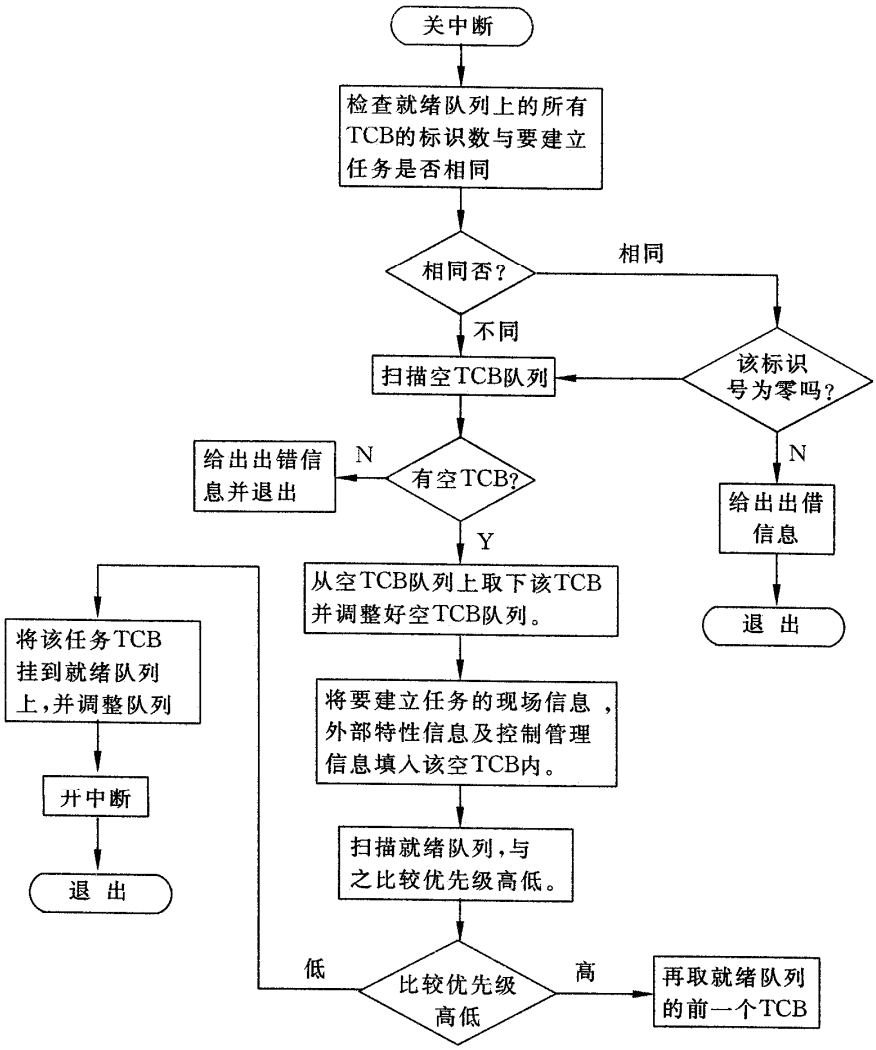


图 11-4　建立任务命令实现流程图

CH12　操作系统结构

12.1　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。

12.2 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。

12.2.1 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。

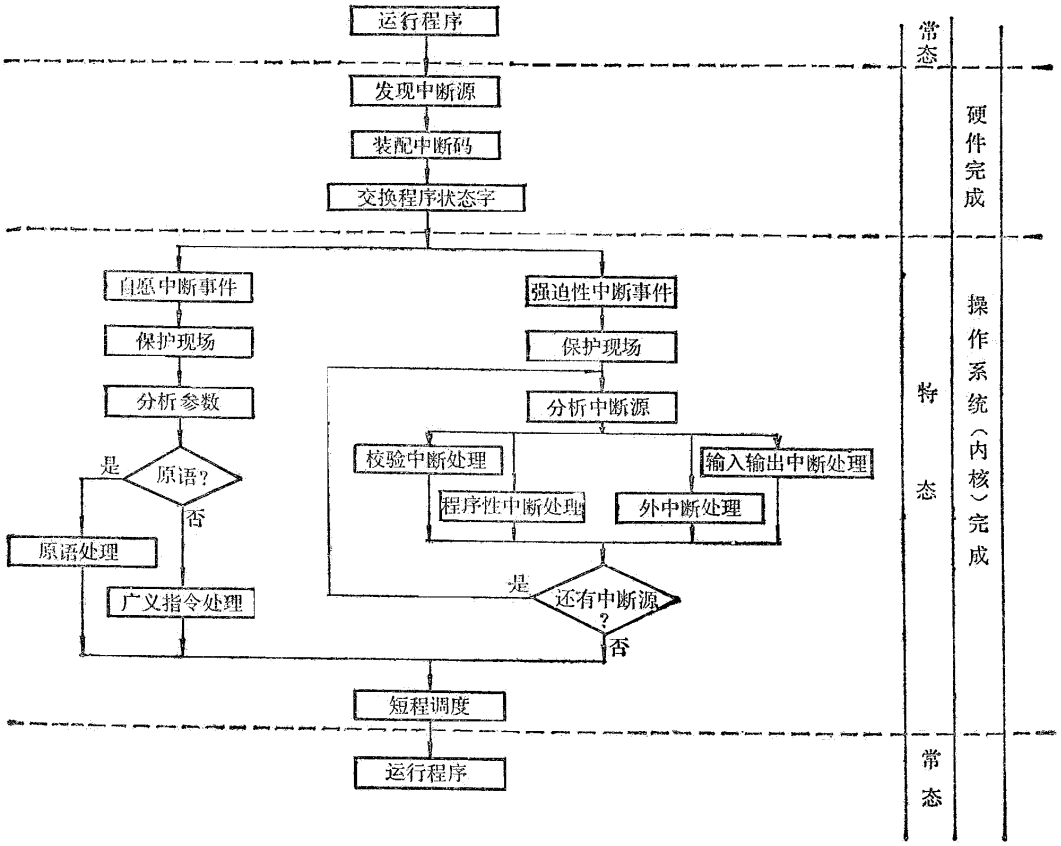


图 12-1　内核的处理流程

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。

12.2.2　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。

12.2.3 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。

12.2.4 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。

12.2.5 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。

12.3 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。

12.4 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 12-2 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。

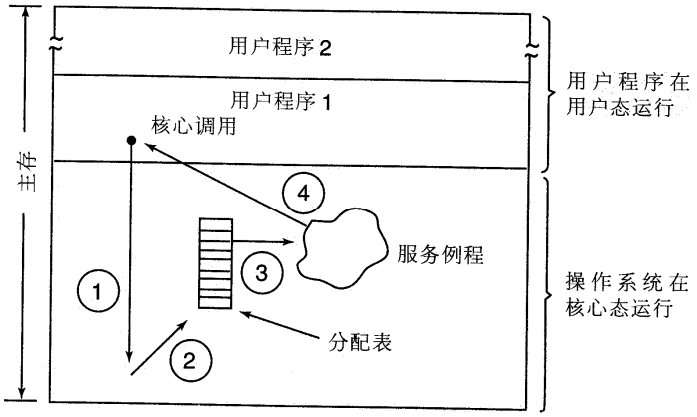


图 12-2　系统调用的操作过程：①用户程序陷入核心，②操作系统确定所请求的服务编号，③操作系统调用服务过程，④控制返回到用户程序

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 12-3 所示。

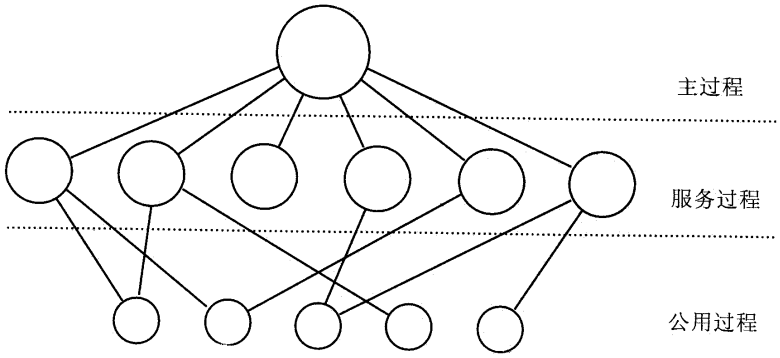


图 12-3　整体式操作系统的简单结构模型

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。

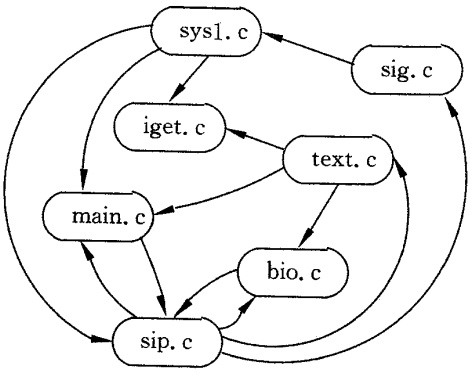


图 12-4　Unix 部分文件间的复杂调用关系

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。

12.5 层次式结构

12.5.1 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。

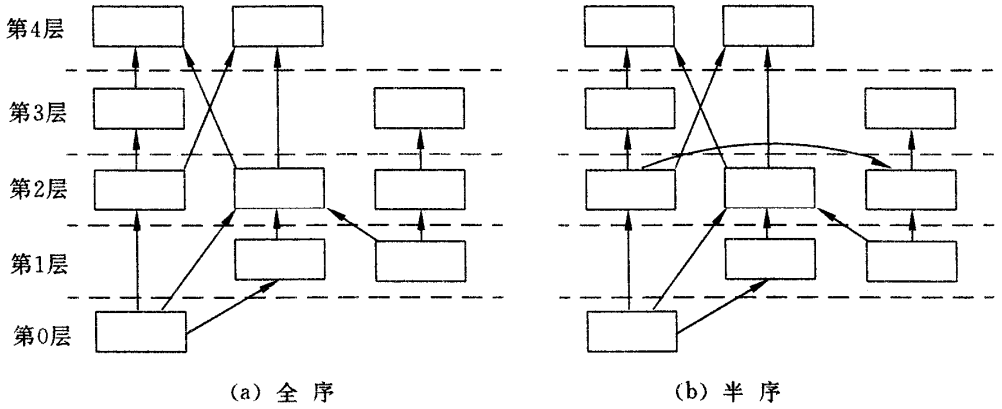


图 12-5 层次结构

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 12-5（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 12-6（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。

12.5.2 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 12-6 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 12-6　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。

12.5.3 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。

12.6 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 12-7 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。

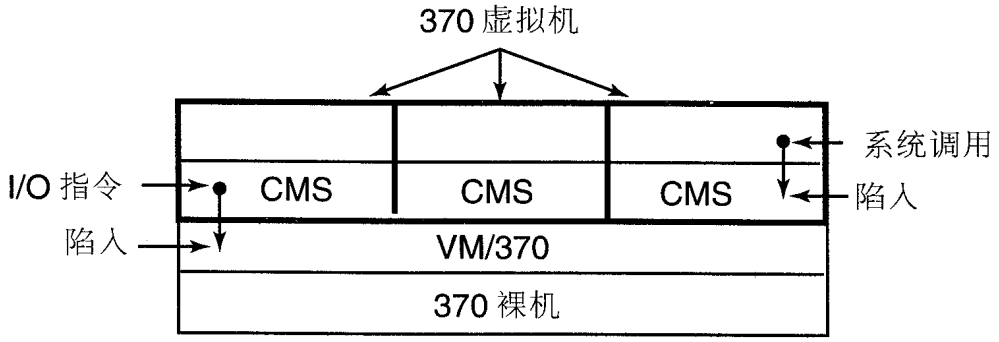


图 12-7　带 CMS 的 VM/370 结构

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。

12.7 客户/服务器结构（微内核结构）

12.7.1 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。

12.7.2 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 12-8 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。

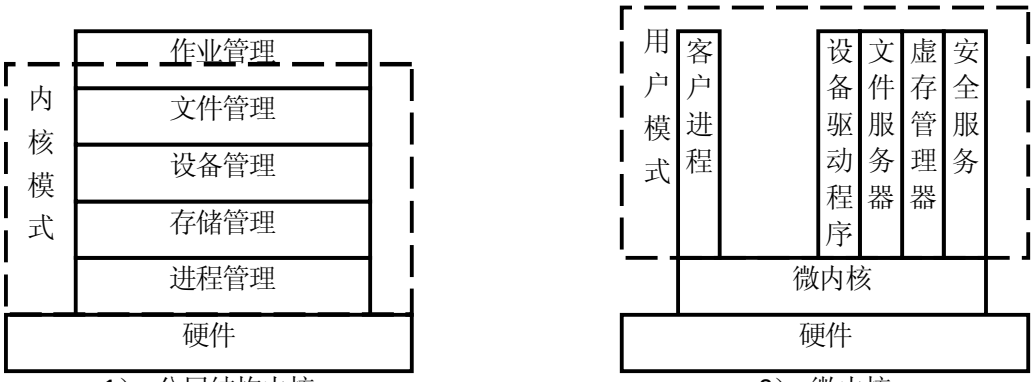


图 12-8　分层结构内核和微内核结构对比

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 12-9 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。

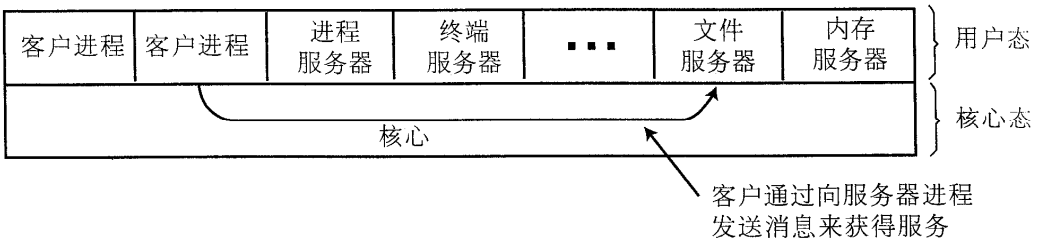


图 12-9　客户/服务器模型

12.7.3　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。

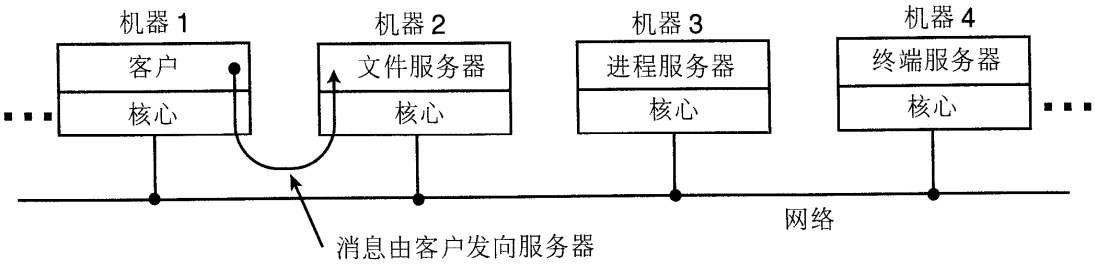


图 12-10　在一个分布式系统中的客户/服务器模型

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。

12.7.4　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。

12.7.5 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。

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 12-11 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 12-11　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。

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。

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。

12.8 实例研究：Windows2000 的系统结构

12.8.1 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。

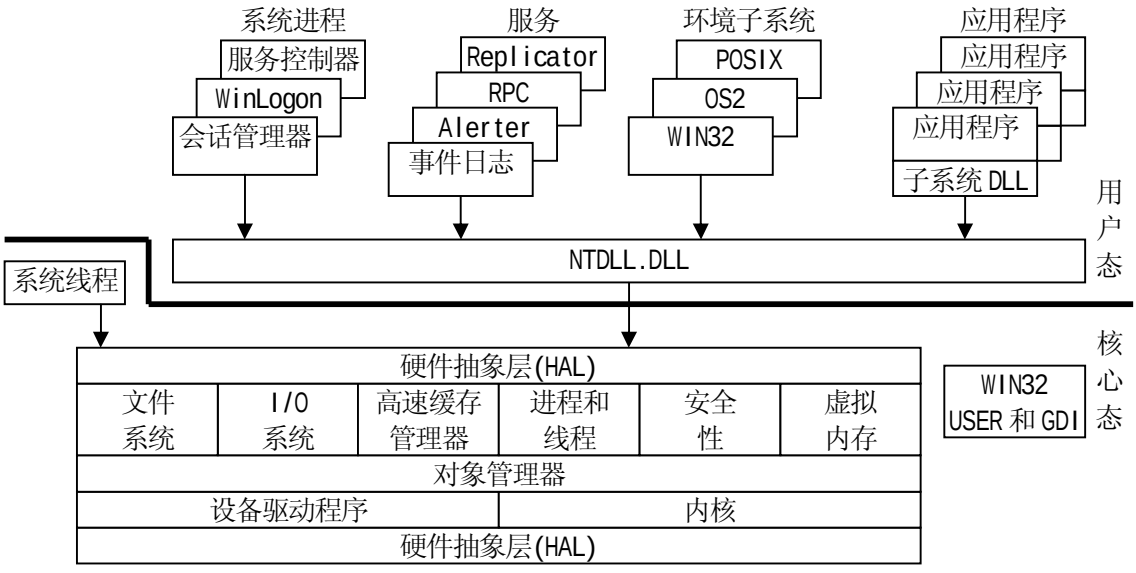


图 12-12　Windows2000 的系统结构

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 12-12 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。

12.8.2 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。
- 本地安全身份验证服务器进程 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 12-12 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。

CH13　操作系统 UNIX

13.1　UNIX 操作系统概述

13.1.1 UNIX 系统的结构

UNIX 是一个通用的、多用户分时、交互型的操作系统，是美国贝尔实验室的汤普逊（K. Thompson）和里奇（D. M. Ritchie）于 70 年代初共同研制成功的。在经历了开发、发展、不断完善和广泛应用的过程之后，影响日益扩大。目前，UNIX 系统已经成为世界上最为著名的分时操作系统之一，我们这里所介绍的，是以 PDP-11/40 机上的第六版为依据。UNIX 系统的结构，可如图 13-1 所示，下面对此做些说明。

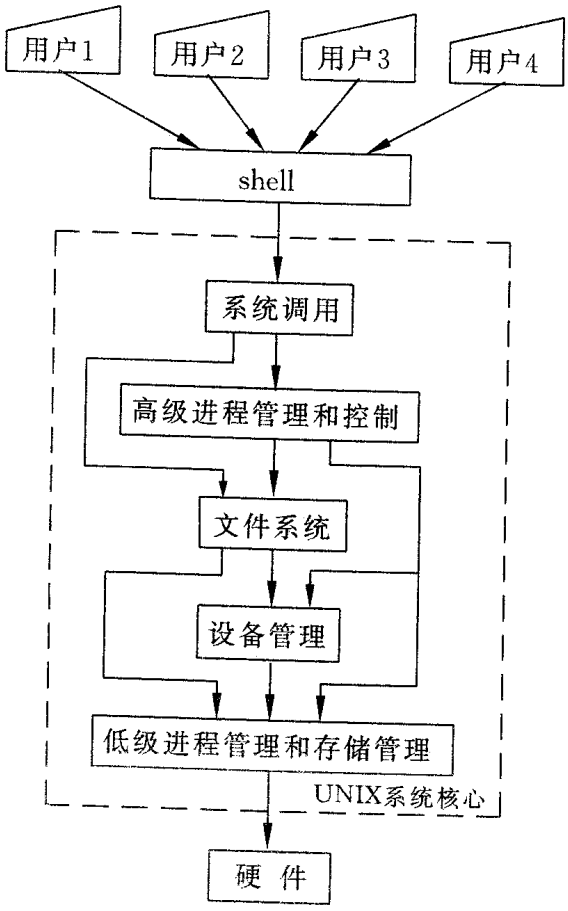


图 13-1　UNIX 系统结构示意图

(1) UNIX 系统基本上可分成核心与核外程序两部分，通常所说的 UNIX 操作系统多是指核心程序，它大致由存储管理、进程管理、设备管理和文件系统管理等几部分组成。进程管理还可以进一步分成高级进程管理和低级进程管理两部分。前者主要包括：进程创建、终止；进程间的高级通信；进程在内、外存之间的转移；信号机构和进程间的跟踪控制等。后者主要包括：进程调度；进程间的低级通信机构等。

(2) 由于 UNIX 产生于七十年代，当时才刚刚开始进行结构程序设计方法的研究，

因此 UNIX 系统的结构并不是很理想的，它的整个核心属于模块结构，模块间的关系极为复杂。不过由于设计者具有的丰富经验，悉心研究和精心构思，使得核心程序不光代码紧凑，系统效率高，而且对模块间的关系做了较好的处置和安排，所以从模块间的调用关系上看，除某些模块外，大多数模块之间仍然可以划分成一定的层次。譬如从图 1 可以看出，文件系统的大部分模块都在设备管理的上面，而设备管理又处在存储管理之上。

(3) UNIX 操作系统源程序由三大部分组成：

- C 语言程序文件；
- C 语言全局变量和符号常数文件；
- 汇编语言程序文件。

每个 C 语言程序文件包括若干个子程序，这些文件都可以独立编译，UNIX 操作系统所具有的功能基本上由这些文件反映出来。C 语言全局变量和符号常数文件中包含有 UNIX 操作系统使用的重要数据结构的说明，它与 C 语言程序文件的区别在于它不能单独编译，只能和 C 语言文件一起编译。汇编语言程序文件大约有 1000 条左右的汇编语句，大部分反映了与机器硬件直接关联的功能，也有的是为追求系统效率而设置的。

(4) 如果把 UNIX 核心看作是分层的，那么核心的最外层是“系统调用”。系统调用命令是 UNIX 向用户提供的第一种界面——面向用户程序的界面，即用户在编写程序时可以使用的界面，这是用户程序获得操作系统为其服务的必由和唯一的途径。

很多操作系统也都为用户提供系统调用这种界面，但这只能做到汇编语言这一级上（即用户在编写汇编语言源程序时，可在源程序中使用操作系统提供的系统调用命令）。但是 UNIX 不仅在汇编语言，而且也在程序设计语言 C 中向用户提供这种界面，用户可以把系统调用命令视为 C 语言的一部分去加以应用，去编写自己的 C 语言源程序。

(5) UNIX 是一个多用户分时、交互式的操作系统，用户会经常通过终端发出命令与系统进行交往：一方面控制自己的作业在系统中的运行，另一方面也可对系统采取的某些动作作出及时地响应。这种要求操作系统进行某种工作的通信语言，在 UNIX 系统中形成了向用户提供的第二种界面——面向用户终端的界面，即命令语言 shell。UNIX 系统提供的 shell，不仅是一种键盘命令语言，而且也含有许多高级语言所具备的复杂的控制结构与变量运算能力。因此，用户除了能使用 shell 在终端上与 UNIX 系统会话，直接参与管理和控制计算机工作，还可以把它做为一种程序设计语言，用来编写出所谓的 shell 过程，以文件形式存入系统，需要时经 shell 调入内存，按用户进程的形式执行。一般地，把 shell 提供的界面简称为用户界面，把系统调用命令提供的界面简称为系统调用。

13.1.2 UNIX 系统的运行描述

如图 13-2，在 UNIX 系统生成后，系统的所有程序都是以文件的形式存放在磁盘上，系统开始初启，引导程序把系统的核心（UNIX 操作系统）放入内存空间的低地址部分，随之初启程序为系统建立起进程 0。

进程 0 是系统调度进程，其主要任务是负责把在盘上的准备运行的所有进程换入内存。

进程 1 是系统初启时由进程 0 创建的。随之，它为每一个终端建立一个相应的 shell

进程，这些进程都执行 shell 命令解释程序。每个终端的 shell 进程都等待用户敲入命令，在接到命令后，相应进程的 shell 解释程序就开始工作。它根据命令名去查找存放在文件系统中的这个命令文件，并将它调入内存，建立一个子进程来执行这个命令，以完成 shell 命令的功能。这个子进程执行完后就被撤销，回到相应 shell 进程，以接收下一个 shell 命令。

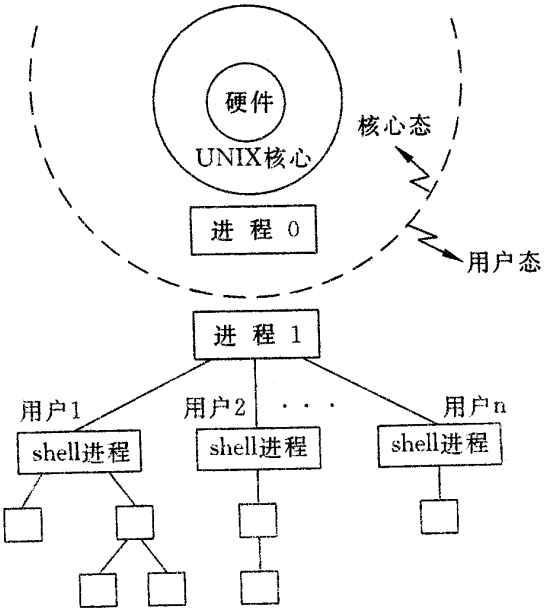


图 13-2　UNIX 系统运行描述图

这里要注意两点：第一，进程 0 和进程 1 是在系统初启时在核心态下直接创建的两个进程，它们与 UNIX 系统中的其它进程不同，其它进程都是通过系统调用命令 fork 而建立起来的。UNIX 操作系统能否正常运行，在很大程度上取决于进程和进程 1 的工作。第二，在一般的操作系统中，命令解释语言都是作为操作系统的一部分而常驻内存的（譬如 MS-DOS 的 COMMAND.COM 文件），但 UNIX 的 shell 却采用了独特的处理方法，它把实现 shell 命令的程序都以文件的形式存放在文件系统中，每一个终端的 shell 进程根据用户键入的命令，找到相应的命令程序，并为之建立一个子程序在用户态下运行。这样的好处，不光使 UNIX 操作系统显得短小精悍，节省内存，而且能很方便地扩充和更改 shell 的功能，使得用户能根据自己的需要，灵活地构造出与 UNIX 系统的用户界面。

13.2　UNIX 操作系统的进程管理

13.2.1 UNIX 操作系统的进程

1、进程映像与UNIX进程

从前面进程的基本概念可知，描述一个进程应由程序、数据集合以及进程控制块 PCB 三者来完成。

程序——表示该进程所要完成的操作，允许多个进程共享一个程序，这时的程序应编制成为“可重入”的结构，即程序的代码在执行过程中不能被修改。

数据集合——这是程序运行时所需要的数据以及工作区（如栈区等），是附属于

该进程的私有物。

进程控制块——表示进程存在的唯一标志，它记录下有关进程的重要信息。譬如进程的状态，保留进程运行暂停时刻的各通用寄存器，控制寄存器，程序状态字的当前值等。

可以看出，在整个进行的生命周期里，这三部分的瞬息变化如同电影中的一幅幅镜头似地，描绘出了该进程的执行过程。因此，UNIX 中就称这三部分为进程映像，把进程定义为是“进程映像的执行”。

2、UNIX进程的两种运行状态

在有的操作系统里，把执行操作系统程序的进程称作系统进程，譬如作业调度进程、存储分配进程等；而把执行用户程序的进程称为用户进程。UNIX 对此有不同的处理：在 UNIX 里，进程既可以执行操作系统程序，也可以执行用户程序，操作系统程序的任务是管理资源，控制系统中的各种活动，用户程序则应在操作系统的管理和控制下，完成自己的任务。

为了能区分开一个进程在两种不同环境下的运行情况，UNIX 采取的措施为：

第一，让操作系统程序和用户程序构成各自的虚拟地址空间，运行时两者在内存占有不同的存储区域，使用两套不同的映射寄存器组，来分别实现它们的虚拟地址空间到内存地址空间的映射。

第二，为进程设置了两种运行状态：核心态和用户态，这实际上也是处理机的两种工作状态，体现在处理机状态字 PS 里（图 13-3）。

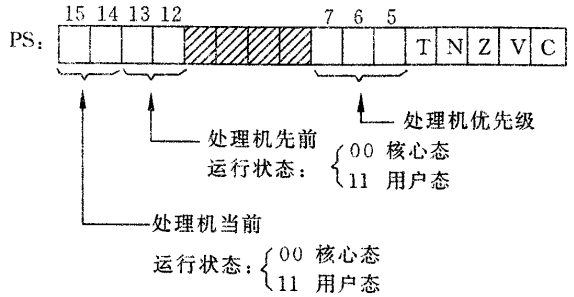


图 13-3　PDP-11 机的处理机状态字 PS

所谓核心态，即表示进程在执行操作系统程序，此时它的活动地址空间为核心空间；所谓用户态，即表示进程在执行用户程序，此时它的活动地址空间为用户态空间。这两种运行态将按照需要在一定时机进行转换。

3、UNIX进程的映像

UNIX 把进程映像分成：进程控制块、数据段、以及共享正文段三大部分，下面加以分别介绍。

1) 进程控制块

UNIX 的进程控制块由基本控制块 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，因此为了节省内存空间，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在一起。下面行介绍 proc 结构。

系统中维持一张 proc 表它有 50 个表目，每个表目为一个 proc 结构，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。每个进程对应的 proc 结构，包含有 22 个字节，用 C 语言表示，下面列出其中最常用的几项加以说明。

```
struct proc
{   char p-stat;   进程状态
    char p-flag;   进程特征
    char p-pri;   进程优先数
    char p-sig;   软中断号（送至进程的信号数）
    char p-uid;   进程所属的用户标识数，指向终端用户的 tty
    char p-time;   进程在内存或外存的驻留时间
    char p-cpu;   用于调度优先数计算
    char p-nice;   用于调度优先数计算
    int p-ttyp;   指向对应终端的 tty 结构
    int p-pid;   进程标识数
    int p-ppid;   父进程的标识数
    int p-addr;   进程数据段起始地址（在内存或盘上）
    int p-size;   数据段大小
    int p-wchan;   标识进程睡眠的原因
    int * p-textp;   指向本文的文本结构
}   proc [NPROC]
```

其中，

p-stat　表示进程状态码

NULL 0　表示进程结构尚未分配任何进程

SSLEEP 1　进程处于高优先级睡眠状态

SWAIT 2　进程处于低优先级睡眠状态

SRUN 3　逻辑上可能运行，已获得处理机为运行态，否则为就绪态，等待处理机

SIDL 4　父进程正在创建子进程

SZOMB 5　表示进程终止态，等待父进程最后消灭它（使成为 NULL）

SSTOP 6　等待父进程发送跟踪命令

p-flag　进程特征标志码

SLOAD 01　表示进程映象已在内存中

SSYS 02　该进程是调度进程、不能换出内存

SLOCK 04　表示本进程映象不能换出内存

SSWAP 010　表示本进程映象不能换出内存

STRC 020　子进程已发出跟踪请求

SWTED 040　另一跟踪标志

进程诸状态之间的转换，如图 13-4 所示。

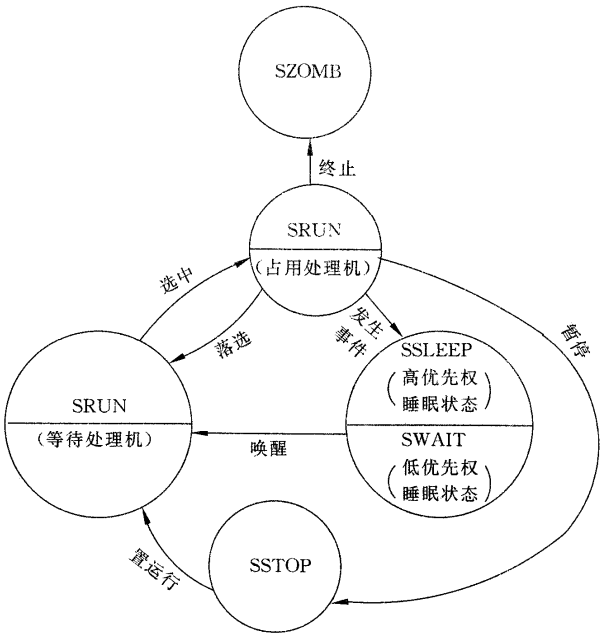


图 13-4　UNIX 的进程状态变迁图

(1) 运行状态 SRUN

一个进程占有处理机时就处于运行状态。这时其 ppda 的起址为核心态第七页 $KISA_6$ 所指，即 proc [i].p-addr 就是 $KISA_6$ 所指的地址。

(2) 准备就绪状态

一个进程如等待占用处理机，则处于准备就绪状态，其 p-stat 也为 SRUN，它与运行状态的主要区别是，第一，它的 ppda 的起始地址（或 p-addr）尚未在 $KISA_{60}$。第二，可能进程映象尚未在内存，这时 SLOAD 尚未被置位，调度程度 sched ( )按进程优先级从高到低将所有准备就绪进程调入内存。

(3) 睡眠状态

一个运行进程由于下列原因之一，放弃处理机而进入睡眠状态。

① 进程使用系统调用 wait ( )或 sleep ( )，自身主动停止，以实现进程间的同步或进程自身定时间歇。

② 进程在运行过程中要求使用某种系统资源，由于系统资源不足，不能立即满足其要求，它被迫进入睡眠状态，等待其它进程释放这类资源。

③ 保护程序运行的临界区。例如，当进程 A 对进程通信文件 pipe 进行读写时，要对其它进程封锁。若此时有进程 B 企图使用 pipe 时，要迫使进程 B 进入睡眠状态。

④ 等待 I/O 操作结束。

⑤ 调度进程，即进程 0 在一次调度结束后进入睡眠状态。

进入睡眠状态的进程，系统按其睡眠的原因，赋予它当被唤醒时所具有的调度优先数分别有高低优先级睡眠状态 SSLEEP 和 SWAIT。

(4) 停止状态 SSTOP

这是一种特殊的睡眠状态，被用于跟踪机构，表明子进程已接收信号等待父进程对它进行跟踪处理。由 stop ( )程序把其 p-stat 置成 SSTOP。以后由 setrun ( )程序将停止状态进程转入就绪状态。

(5) 等待善后处理状态 SZOMB

一个进程用系统调用 exit ( )自我终止，并未立即放弃 proc 结构，相应的 user 结构也暂留起来。这时 p-stat 被置成 SZOMB 状态，并唤醒其父进程对它进行善后处理。这是一个进程在消亡之前，等待父进程对其作某些处理的一个短暂状态。

• P-flag 进程标志

进程标志实际上是对进程状态的进一步说明，它只有一个字节，利用该字节中各位的 0 和 1 取值，表示进程的映像是否在内存、是否可交换到磁盘的对换区上去，等等。共有六个标志位（图 13-5），一个进程的标志可以是这六种标志的组合：

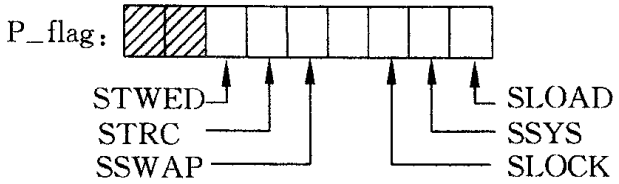


图 13-5　UNIX 的进程标志字节

SLOAD：该位为 1，表示本进程的映像全在内存中；

SSYS：该位为 1，表示本进程为进程 0，应常驻内存；

SLOCK：该位为 1，表示本进程映像因某种原因不能调出内存；

SSWAP：对换标志；

STRC：该位为 1，表示本进程已向父进程提出跟踪请求；

STWED：该位为 1，表示父进程已接受了进程跟踪的请求。

• P-pri 进程优先数

进程优先数表示一个进程的优先权，其值越小，则该进程的优先权越高。优先数的取值范围为-100~+127。

• P-time 进程映像调入内存后，在内存的驻留时间，或进程映像调出到盘对换区后，在盘对换区的驻留时间。

• P-addr 进程数据段的起始地址，也就是除正文段外，进程非常驻内存部分的起始地址（ppda）。

• P-size 数据段的长度。

• P-wchan 标明处于睡眠状态的进程的睡眠原因。

• P-textp 是一个指向 text 结构指针，在那里存放着与该进程正文段有关的信息。

• P-cpu 记录进程使用处理机的程序，值越大意味着进程使用处理机的程度越高。

2) 数据段

UNIX 进程映像中的数据段由如下三部分顺序排列组成（图 13-6）：

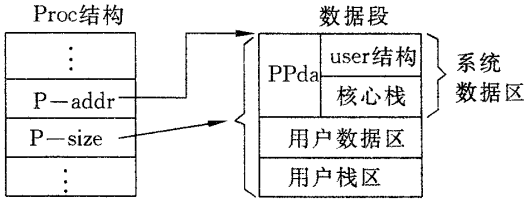


图 13-6　UNIX 的数据段

USER 数据结构的 C 语言表示如下：

```
struct user
{
    int u-rsav [2];     保存 r5, r6，用于程序的正常或非正常返回
    int u-fsav [2];    用于保留浮点寄存器
    char u-segflg;    I/O 目的地址在用户空间或系统空间的标志
    char u-error;    返回出错号码
    char u-uid;    有效用户标识数
    char u-ruid;    有效用户小组标识数
    char u-rgid;    实际用户小组标识数
    int u-procp;    指向本进程 proc 结构的指针
    char * u-base;    指向读写缓冲存储器的起始地址
    char * u-count;    用来存放读写字节数 nbytes
    char * u-offset[2];    在文件中进行读、写的起始位置
    int * u-cdir;    当前工作目录文件的 inode 的指针
    char u-dbuf [DIRSIZ];    用于存放当前工作文件的路径名 DIRSIZ=14
    char * u-dirp;    指向当前目录文件名的指针，路径名最多 14 个字符
    struct {构成文件的目录项，由文件所在节点的节点号 ino 和文件的路径
        int u-ino;    名组成
        char u-name [DIRSIZ];
        } u-dent;
    int * u-pdir;    指向父目录项 i 节点的指针
    int u-uisa [16];    本进程用户空间地址寄存器样本，由 16 个组成
    int u-uisd [16];    本进程用户空间说明寄存器样本，由 16 个组成
    int u-ofile [NOFILE];     用户打开文件表，由指向系统打开文件表的表项指针
组成，NOFILE=15，共有 15 项
    int u-arg [5];    用于存放当前系统调用所带的参数
    int u-tsize;    本进程的文本段的块数
    int u-dsize;    本进程的数据段的块数
    int u-ssize;    本进程用户栈的块数
    int u-sep;    指令空间 I（文本段）和数据空间 D（数据段）是否分离的标志
    int u-qsav [2];    保存 r5, r6
    int ussav [2];    保存 r5, r6
    int u-signal [NSIG];    软中断处理程序入口表，NSIG=20，共 20 项
    int u-utime;    本进程用户态运行时间
    int u-stime;    本进程核心态运行时间
    int u-cutime [2];    子进程用户态运行时间之和
    int u-cstime [2];    子进程核心态运行时间之和
    int * u-aro    中断保留区中保留 R0 的栈内单元的地址
    int u-prof [4];    用于保存程序直方图计算的参数
    char u-intflg;    表示系统调用执行完成与否的标志
} u;
```

•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。

• 用户数据区，这时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。

对于数据段，做如下几点说明：

第一，在 user 结构里，总共包含 35 项内容，涉及到现场保护、内存管理、系统调用、文件管理、文件读写、时间信息、进程映像位置等方面。譬如，u-procp 是指向本进程基本控制块 proc 的指针，于是 u-procp 与 proc 结构中的 p-addr 就形成了互相勾链。又譬如在进程的 user 结构里劈了两个数组，一个为 uisa，一个为 uisd，这们是该进程用户态虚地址空间八个页面的相对虚、实地址映照表。再譬如，在 user 结构里有一个整形数组 u-ofile[15],共有 15 个元素，称作为该进程的打开文件表，进程每打开一个文件，就在此表里做一登记，所以每个进程可同时最多打开十五个文件。

第二，一个进程的 proc 结构是常驻内存的，当该进程运行时，进程映像的其余部分也都全部在内存。但当处理机转去运行别的进程时，为了节省内存，就可能会把该进程映像的绝大部分换出送到磁盘上去。由于进程运行的程序可能是共享的，这种共享正文段此时就不能交换出去。所以一个进程的数据段不能包含共享正文段，它是进程映像的非常驻内存部分。在 UNIX 里，进程映像的非常驻内存部分在内、外存之间的对换操作是经常进行的，磁盘上被开辟专门存放进程非常驻内存部分的那部分区域，在 UNIX 里称为盘对换区。

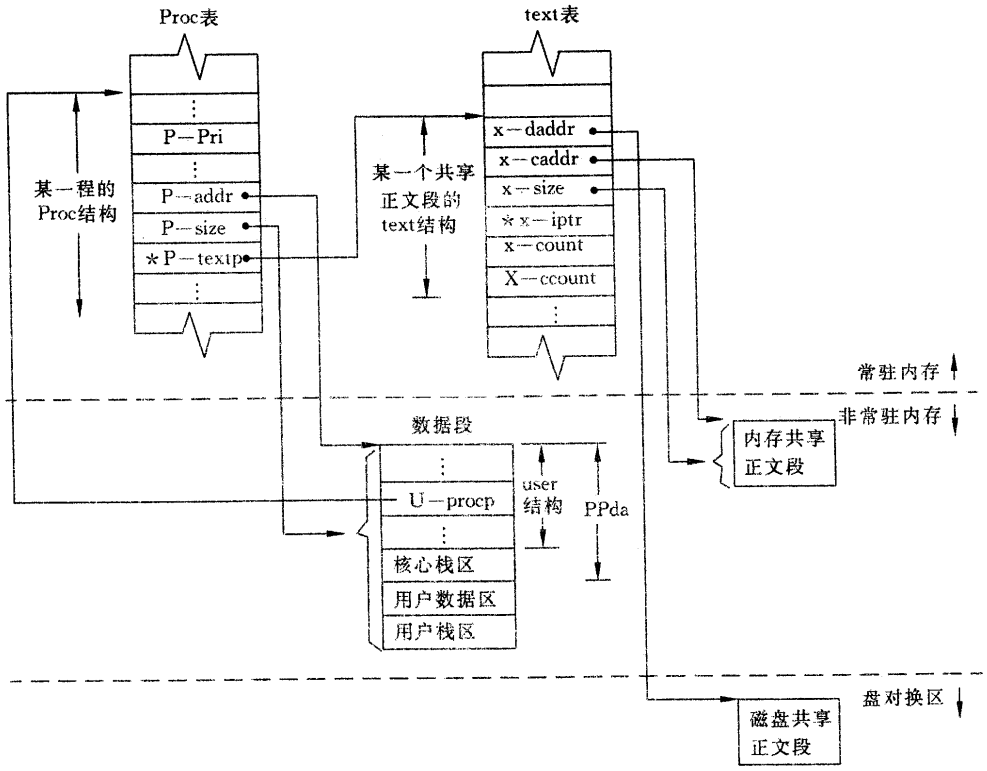


图 13-7　UNIX 进程映像的基本结构

第三，在实行内、外存之间的进程映像对换时，被对换的非常驻内存部分应是一个整体，即要调出则一起调出，要调入则一起调入，这样管理起来比较便利。

3) 共享正文段

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，那么它的进程映像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像全部在内存，那么这个共享正文段就不得调出去。

由于共享正文段在进程映像中的特殊性，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（位置、尺寸、共享的进程数等），有时也把这种结构称为正文段控制（信息）块。text 结构的内容如下所列：

x-daddr：正文段在盘对换区中的起址；

x-caddr：已被装入内存的正文段在内存中的起址；

x-size：正文段的长度；

x-count：共享本正文段的所有进程数；

x-ccount：共享本正文段、且进程映像全部在内存的进程数；

x-iptr：指向正文段所有文件的；节点的指针。

要说明的是，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 进程映像的三个组成部分中，proc、user 和 text 这三个数据结构是最为重要的角色，图 7 反映了这三者之间的基本关系。

13.2.2 UNIX 操作系统的进程调度

1、进程切换调度及其时机

UNIX 系统中的进程 0 主要是由两个程序组成：一是进程切换调度程序 swtch，一是进程映像传送程序 sched。swtch 使运行进程释放处理机，然后选择另一个进程占用处理机，所以 UNIX 中是通过进程切换调度程序 swtch 来实现对 CPU 的具体分配而完成进程调度任务的。

为了处理机分配给一个进程使用，swtch 来查看整个 proc 表，在处于就绪状态（SRUN）且进程映像在内存（SLOAD）的进程里，挑选出一个优先数最小（即优先权最高）的进程来投入运行。如果找不到满足条件的进程，则进程 0 将处于等待中断状态，一旦发生中断请求，处理机响应并处理后，进程 0 就会再次去搜索 proc 表。这种作法，能使系统以最快的速度找到可运行的进程，提高 CPU 的利用率。

进程将在以下两类情况下进行切换调度：

(1) 当前占用处理机的进程不再具备继续占用处理机的条件而自愿让出处理机。譬如它已完成了任务而进入 SZOMB 状态；它因等待某事件的发生而进入 SSLEEP 或 SWAIT 状态；它申请内存资源得不到满足，只能先将自己调出到盘对换区上，等内存空间够用时再调入，这时该进程仍然处于 SRUN 状态，但不带 SLOAD 标志；如此等等。

(2) 当系统中出现比运行进程更适宜占用处理机的进程时，强迫现运行进程让出处理机。譬如，由于某事件的出现使处于 SSLEEP 或 SWAIT 状态的进程变为 SRUN 状态，且优先权比现运行进程的高，这时就要强迫现运行进程让出处理机；又如中断发生时，现运行进程在用户态下工作，则中断处理结束后，要看一下是否有因中断而出现了具有更高优先权的进程，如果出现，就要强迫现运行进程（它被中断打断）让出处理机；如此等等。

2、进程切换调度算法

由上可知，UNIX 在进程切换调度时使用的是优先数算法，即根据一个进程的优先数大小来确定它是否能获取处理机，优先数越小，获得处理机的权利越高。

UNIX 中进程的优先数是动态变化的，确定一个进程优先数，有设置和计算两种方法。设置方法用于当一个进程从运行状态变为睡眠状态时，系统将根据进程不同的睡眠原因，赋予它们不同的优先数（存放在其 proc 结构里的 p-pri 里），这个优先数在进程唤醒后发挥作用。如果赋给的优先数为负值，则这是所谓的高优先权睡眠状态，这时进程 proc 中的 p-stat 被设置为 SSLEEP；否则是所谓的低优先权睡眠状态，这时进程 proc 的 p-stat 被设置为 SWAIT（参见图 9-10）。计算方法是使用如下公式来算出进程的优先数 p-pri：

$$p\text{-}pri=\min\{127, (p\text{-}cpu/16+PUSER+P\text{-}nice)\}$$

其中 p-nice 是一个用户可以通过系统调用命令来设置的量（记录在进程 proc 结构里），各种用户可按执行任务的紧急程度来给相应进程赋上不同的 p-nice 值；PUSER 为常数 100；P-CPU 在进程的 proc 结构里已见过，它反映了进程使用处理机的程度。进程优先数 p-pri 在数值 127 和(p-cpu/16+PUSER+p-nice)两者中取其小的值。

这时最关键的是 p-cpu，它按照如下办法来改变：系统时钟中断处理程序每 20ms 做一次，每做一次就将现运行进程的 p-cpu 加 1。每到 1 秒时它就依次检查系统中所有进程的 p-cpu 一次，如果被查进程的 p-cpu<10，这表时该进程在这一秒内所使用处理机的时间不超过 200ms，于是把这个 p-cpu 置为 0；如果被查进程的 p-cpu>10，这表明该进程在这一秒内所使用处理机的时间超过 200ms，于是把 p-cpu 减 10。这种改变办法的效果是：

(1) 如果一个进程连续占用处理机较长时间，那么它的 p-cpu 值就增大，从而计算出的 p-pri 也增大，于是优先数下降。当实行进程切换调度时，这种进程被调度到的可能性就减少。

(2) 如果一个进程长时间未被调用到或者虽频繁调度到，但每次占用的时间都小于 200ms，那么 p-cpu 就较小甚至为 0，从而计算出的 p-pri 也较小，于是优先权上升。当实行进程切换调度时，这种进程被调度到的可能性就增加。

这种称之谓“负反馈”的效果（见图 13-8），就能使用户态下运行的进程能有较为均衡地获取 CPU 时间的权利。

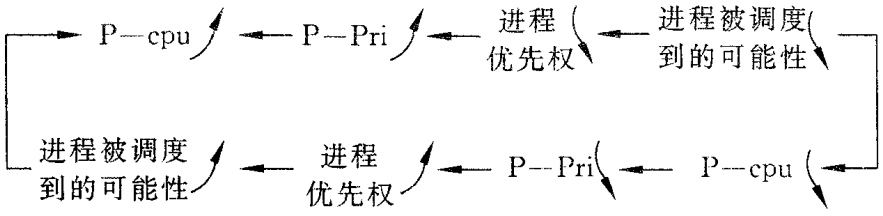


图 13-8　p-cpu 变化带来的负反馈效果

UNIX 系统计算优先数的时机有二：一是对所有优先数大于 100 的进程，系统每秒钟重新计算一次它的优先数；二是每次系统调用命令处理完毕之时，就重新对现进程的优先数进行计算。

3、进程映像的调入和调出

进程切换调度程序 swtch 选择运行进程的条件之一是该进程的映像要全部在内存。因此，把进程映像在内存和盘对换区之间进程调入和调出的传送工作，为切换调度创造了前提条件，也就为 swcth 提供了尽可能多的候选者。如前所提及的，这一调入、调出的传送任务是由进程 0 中的进程映像传送程序 sched 来完成。

1) 谁是调入者？当一个就绪进程的映像不在内存（即处于 SRUN 状态、但 SLOAD 示被设置）时，只有把它的进程映像调入内存才能让它真正参与对 CPU 的竞争。每个进程 proc 结构中的 p-time 记录了该进程调入内存或调出到磁盘对换区后所经历的时间，sched 程序根据现在进程映像在盘对换区里的就绪进程的 p-time，依照先长后短的原则逐一把它们的进程映像调入内存，直到内存没有跷的空闲区时为止。

2) 谁是调出者？如果 sched 程序实行调入时，内存根本无空闲可以接纳调入者，那么就应先实施调出某一个进程映像的非常驻内存部分，以腾出空间让更具运行条件的进程调入。

在 UNIX 中，有两种类型的进程映像不允许调出内存，一种是 proc 结构里 p-flag 标志字节中的 SSYS 被置位的进程，这是系统进程，整个系统中只有进程 0 才具备这一条件；另一种是标志字节中的 SLOCK 被置位的进程，它表示该进程因为某种原因而要求将其进程映像暂时保留在内存中。除这两类进程外，其余进程映像在内存的进程全是调出的候选者，其调出的先后顺序是：

- 调出低优先权睡眠状态进程（SWAIT）和暂停状态进程（SSTOP）；
- 调出高优先权睡眠状态进程（SSLEEP）；
- 按在内存中驻留时间的长短，调出就绪状态进程（SRUN）。

13.2.3 UNIX 操作系统的进程通信

UNIX 操作系统为进程之间的通信提供了三种工具，这三种工具或使用的场合不同，谨、或完成的功能有异：

(1) 基本通信工具 sleep 和 wakeup，适用于解决操作系统中程序间的同步和互斥；

(2) 利用“软中断”实现同一用户诸进程之间少量信息的传输和同步；

(3) 文件系统提供的 pipe 工具，实现同一用户两个进程之间大量信息的传输。

1、基本通信工具sleep和wakeup

在 UNIX 操作系统的程序实现过程中，有很多的同步和互斥问题需要解决，这里没有使用惯常的信号量和 P、V 操作，而是根据不同情况做了不同的处理。不过在一旦判定需要转入睡眠去取得同步或互斥时，就统一地调用 sleep 程序，使调用进程进入睡眠状态，在睡眠原因消失以后，就统一地调用 wakeup 程序，以便将有关进程唤醒。

2、“软中断”——进程间的简单通信方式

在 UNIX 系统里，设置有系统调用命令 Signal 和 kill，用于信号的设置和发送之用。

1) signal (sig, func)——信号设置

其功能是建立起信号 sig 与功能 func 之间的关系，即若收到了 sig 信号，则去执行由 func 指出的相应动作。

2) kill (pid, sig)——信号发送

其功能是将信号 sig 送给同一用户的一个或几个进程：如果 pid 为非 0，则它就是接收信号的进程的标识数；如果 pid 为 0，则要将信号 sig 送给所有进程。

当一个进程接获某一信号后，处理方式与硬中断非常类似，即它立即暂停自己正在执行的程序，转去执行该信号事先规定的信号处理程序 func，完成后再返回原先正在执行的程序（图 13-9），所不同的只是信号的设置、检查都是由软件实施的，所以被称作“软中断”。

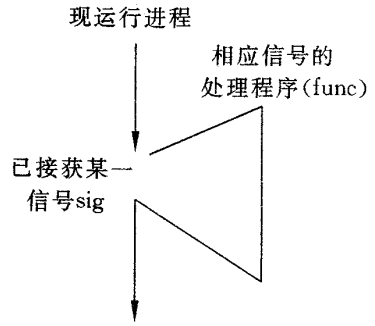


图 13-9 “软中断”处理示意图

进程运行时，会在下列地点或时刻检查是否收到了什么信号：

(1) 在系统执行各类中断处理程序（除 I/O 中断）的末尾，检查被中断的进程是否接收到了信号。

(2) 现运行进程执行系统调用处理程序过程中，可能因为种种原因调用到 sleep 程序，在要进入低优先权睡眠状态之前和被唤醒之后，都要检查是否接收到了信号。

(3) 若时钟中断了在用户态下运行的进程，则每隔一秒钟，时钟中断处理程序就去检测现运行进程是否接到了信号。

3、pipe——进程间大量的信息通信

利用系统提供的系统调用命令 pipe，在文件系统的支持下，就可以为同一用户的两个进程传送大量的信息。

通过系统调用命令 pipe，可以建立起一个 pipe 通信机构，它由两个文件组成，一个在 pipe 机构中只能读，也就是说它是该通信机构中的信息接收端；另一个在 pipe 机构中只能写，也就是说它是该通信机构的信息发送端。这样，发布 pipe 命令的进程可以通过信息接收端进行写操作，而另一进程可以通过信息发送进行读操作。UNIX 规定一次最多可以读/写 4096 个字节的信息。若写的内容超出这个数量，写进程就要在一次写 4096 字节后睡眠等待，在读进程取走这部分信息之后，唤醒写进程继续写。

13.2.4 UNIX 中的进程控制

1、UNIX启动及进程树形成

由系统引导程序，把 UNIX 核心装到内存的最低部分，系统进入启动程序 start。系统首先建立整个系统的调度进程 proc [0]，接着建立进程 proc [1]，由它通过系统调用“exec ( )”执行一个叫做“/etc/init”的文件，为每个终端生成一个子进程，系统等待用户在终端上注册。用户在终端上输入命令，每个命令都对应于一个可执行文件。Shell 命令解释程序解释此命令，找到相应的可执行命令文件，用系统调用 fork ( )创建子进程，由此子进程调用系统调用 exec ( )执行此命令文件；父进程（即终端进程）处于等待状态，子进程执行文件结束，调用系统调用 exit ( )自我终止，唤醒父进程，

善后处理，并再次发出提示信号，准备接收下一命令。

此过程如图 13-10 所示。

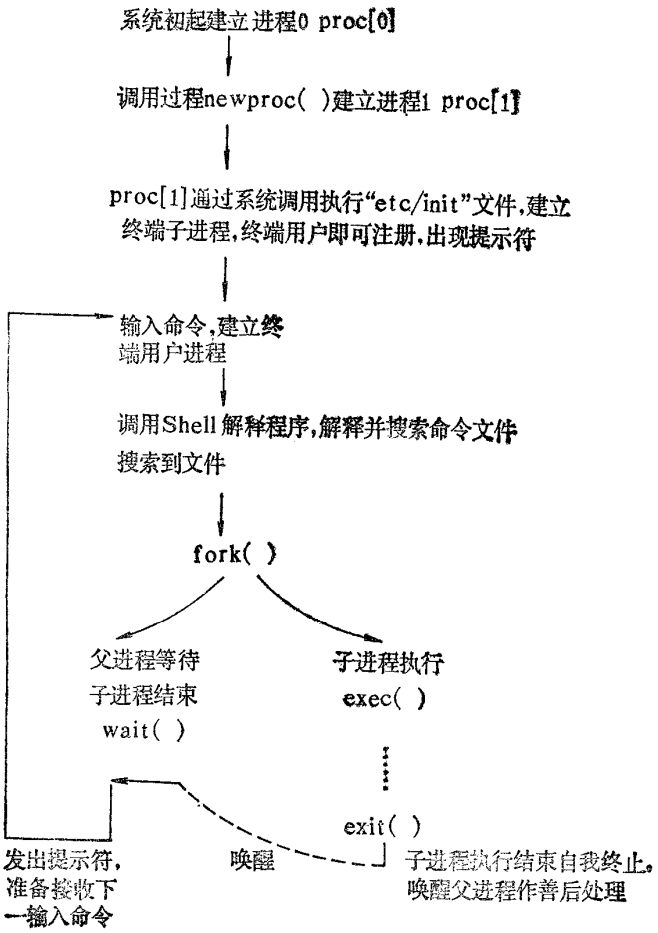


图 13-10　UNIX 进程树形成过程

在这里要指出：

- proc [0]是整个系统的调度进程，其功能是执行程序段 sched (　)不断把外存中的准备就绪的进程调入内存。proc [0]一旦存在，就不会消亡，要么在执行调度功能，要么处于睡眠状态。
- proc [1]是整个系统中除 proc [0]以外的所有进程的祖先。由它产生各终端子进程，子进程又可产生子进程，等等。形成一进程树。

2、进程控制

这一小节要讨论的是关于进程的创建，执行和自我终止的问题，它由三个基本的系统调用 fork (　), exec (　), exit (　)来实现。

fork (　)的功能是创建一个子进程。所创建的子进程是 fork (　)调用者进程（即父进程）的复制品，即它的进程映象，除了进程的标识数物进程特性有关的一些参数以外，与父进程相同。并与父进程共享文本段和打开的文件。当子进程要运行时，通过系统调用 exec (　)按指定文件中的文本段和数据段取代自己的映象，并按指定的参数执行文本段，这相当于一个动态链接。因此 exec (　)的功能是改变进程映象使之执行一个指定的（新的）目的文件。exit (　)系统调用的功能是进程的自我终止，释放资源并通知父进程。此时，它处于 SZOMB 状态，等待父进程作善后处理。父进程用系统

调用 wait ( )等待一个子进程的终止或进入暂停状态 SSTOP。如果在使用 wait ( )之前已经有一个子进程结束了，则对它作简短的善后处理：主要是释放子进程占用的 proc 结构，使之成为空闲项，把子进程的有关时间统计项 u-utime, u-cutime, u-stime, u-cstime 分别加到父进程的 u-cutime, u-cstime 上去。子进程在请求父进程跟踪时，处于暂停状态 SSTOP。

fork ( ), exec ( ), wait ( ), exit ( )这一组系统调用，在 Shell 命令执行过程中使用的情况如图 11-10 所示。它们也可在一般程序中使用，其形式往往为：

```
if (fork ( ))
    { 父进程程序实体
    }
else { 子进程程序实体
    }
```

父进程程序实体一般包含有 wait ( ) 以等待子进程结束，子进程一般以 exit ( ) 自行结束。下面给出一个具体例子：

```
/ * Example for system calls fork, wait, exit*/
main ( ) {
    int i;
    int l;
    int j;
    if (fork ( ))
    {
        i =wait ( );
        l =getpid ( );
        printf (“ I AM PARENT PROCESS, ID num. %d\n”, l);
        printf (“The child process, ID number %d, is finished. \n”, i);
    }
    else {
        printf (“ I am child process. \n”);
        j =getpid ( );
        printf (“ j =$d\n”, j);
        exit ( );
        }
}
```

其运行结果为

```
192
I am child process.
j =122
I AM PARENT PROCESS, ID num. 121
The child process, ID number 122, is finished.
```

13.3 UNIX 操作系统的存储管理

13.3.1 存储管理部件

PDP-13/40 的字长为 16 位，因此可直接寻址的范围为 64K 字节。这就是说，如果没有任何硬件的支持，每个进程可用的最大程序地址空间为 64K 字节。然而处理机的总线寻址能力为 18 位，即内存的物理地址空间是 256K 字节。为了能把 16 位的字节地址映射到 18 位的字节地址上，硬件必须配置存储管理部件，它成为 UNIX 操作系统存储管理的基础。

1、虚地址

在存储管理部件的支持下，16 位的字节地址不再被直接解释为物理地址，而被看作是一个虚地址，虚地址的组成如下图 13-11 所示。

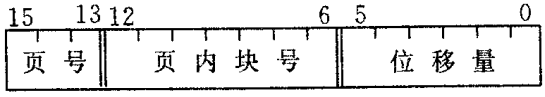


图 13-11　虚地址的形式

把 64K 字节的虚地址空间分成 8 页，每页分成 128 块，每块为 64 个字节，于是每页最多可达到 8K 字节。由于这里的页面大小是可变的，故可称为“可变长的页式管理”。要注意的是，为了充分利用内存空间，系统是按虚页实际的页长来分配内存空间的。

2、活动页寄存器（APR）

虚存页在内存中的实际位置，需要有一张映象表加以反映，这个映象表在 PDP-11 中是由两组活动页寄存器组成的，一组用于核心态，一组用于用户态。每组有 8 个 32 位长的寄存器，每一个又一分为二：一个是页地址寄存器（PAR），一个是页说明寄存器（PDR），如图 13-12 所示。

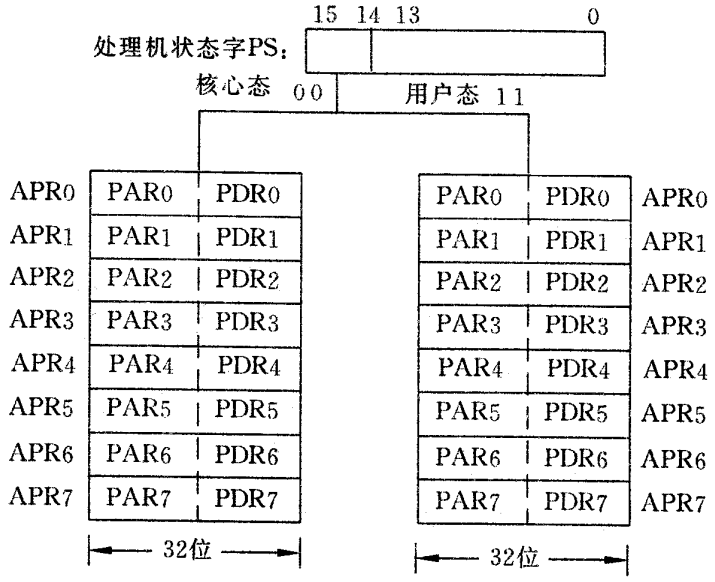


图 13-12　活动页寄存器

1) 页地址寄存器（PAR）

16 位的页地址寄存器如图 9-18（a）所示。PDP-11 中，把内存按 64 个字节为单位划分成块，共有 4096=2^{12} 块，编号为 0~（2^{12}—1），内存分配以块为单位。

据此，每个 PAR 只使用其低 12 位，用来记录虚地址空间相应各页在内存的起始块号。

2) 页说明寄存器（PDR）

16 位的页说明寄存器如图 13-13（b）所示，在它里面记录着虚地址空间相应各页的存取权限、扩展方向以及该页的长度。

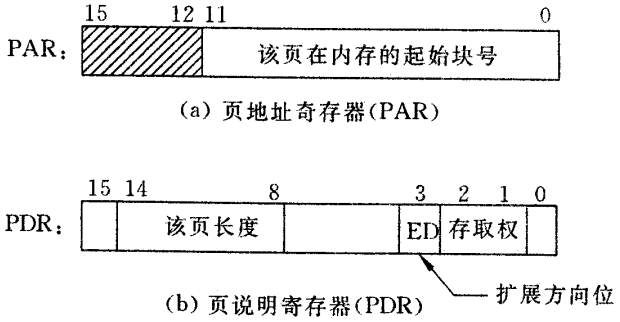


图 13-13　活动页寄存器的组成

- PDR 寄存器的第 1、2 位表明对相应页的存取权限；
- PDR 寄存器的第 3 位（ED）表明该页的扩展方向。当一个虚页实际长度小于 128 内存块时，由 ED 这一位说明该页空白部分在低地址一边还是在高地址一边。这样，如果该虚页要扩展，就可以确定向何方向延伸。具体地说，ED=0，表示空白部分在高地址一边，扩展时就向高地址方向延伸；ED=1，表示空白部分在低地址一边，扩展时就向低地址方向延伸。通常，程序占用页的 ED=0，堆栈占用页的 ED=1。
- PDR 寄存器的第 8~14 位为该虚页的实际长度，用含多少块来表示。由于一虚页最长为 128 块，故由七个二进制位表示。

3、虚——实地址的转换

在得到一个形如图 13-14 所示的虚存空间地址之后，按照如下过程就可实现从虚地址到实地址的转换：

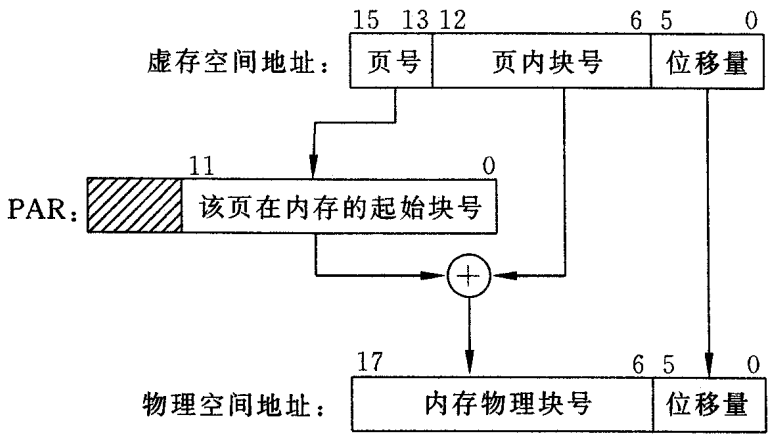


图 13-14　虚地址到实地址的转换示意图

第一步，根据处理机状态字 PS 的第 14、15 位，确定处理机现在所处的运行状态，由此选择相应的活动页寄存器组；

第二步，由虚地址中的页号得到该组中相应的页地址寄存器（PAR）主页说明寄存器（PDR）；

第三步，由页地址寄存器中的“本页在内存起始块号”和虚地址中的“页内块号”构成内存中的物理块号，并由页说明寄存器检查该块号的合法性（如是否越界等）；

第四步，由物理块号和虚地址中的“位移量”构成这个虚地址所对应的物理地址。

13.3.2 UNIX 进程映像的虚、实地址空间

UNIX 的存储管理是建立在上述 PDP-11 存储管理部件基础上的，为此它为每一个进程提供两个虚存，一个是核心态下的虚存，它使用该态下的那组活动页寄存器来实现其虚、实地址的映射；一个是用户态下的虚存，它使用该态下的那组活动页寄存器来实现其虚、实地址的映射。这两个状态下的虚存，最大都是 64K 字节。

1、进程在核心态下的虚地址空间

在核心态下，进程运行的是操作系统程序，要使用核心栈区，要用到该进程的 user 结构，此态下的虚地址空间的组成如图 13-15 所示。

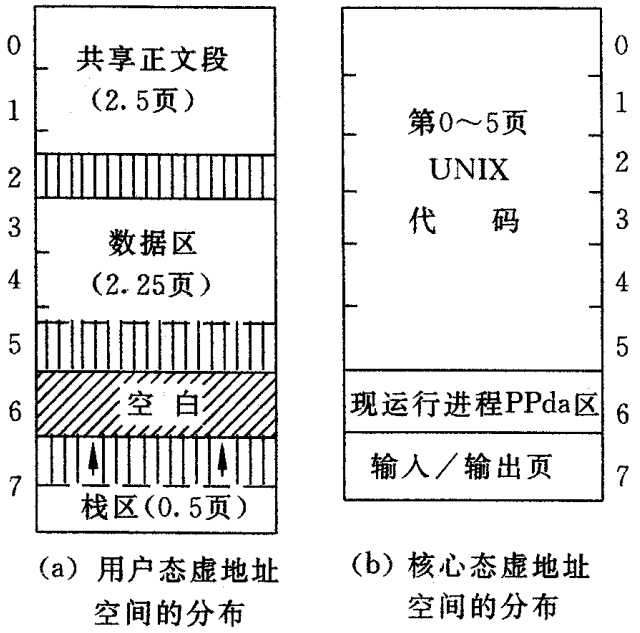


图 13-15　UNIX 进程的虚地址空间

其中，0~5 页存放 UNIX 代码，这里面包括诸如常驻内存的进程 proc 结构；第 7 页存放与系统输入、输出相关的内容，称输入、输出页；第 6 页放现运行进程的 ppda 区，即 user 结构和核心栈，实际只用 1024 个字节。对所有进程来说，第 0~5 页和第 7 页共七页的内容都是相同的。

2、进程在用户态下的虚地址空间

在用户态下，进程运行时涉及共享正文段，数据区（里面包括非共享的程序），以及用户栈区，此态下的虚地址空间的安排原则为：

(1) 由低往高安排先共享正文段，再数据区；由高往低放置栈区；

(2) 以页为分配单位，不满一页按页找齐。如图 14（a）所示，若共享正文段长度

为 2.5 页，则在虚地址空间中占据 0~2 页；数据区为 2.25 页，则在虚地址空间中紧接共享正文段占据 3~5 页；栈区为 0.5 页，则占据虚地址空间的第 7 页，且向低地址方向延伸。这时虚空间的第 6 页为空白，未被占用。

3、进程映像在内存中的实际分布

虽然每个进程都有一个核心态的虚地址空间，但它们中的第 0~5 以及第 7 页的内容是完全一样的。此外，即使是每个进程的用户态虚拟空间，也不能按其分配情况原封不动地照搬到内存中去，否则会造成内存空间的巨大浪费。因此，各个进程映像在内存的真实分布情况是这样的（见图 13-16）；

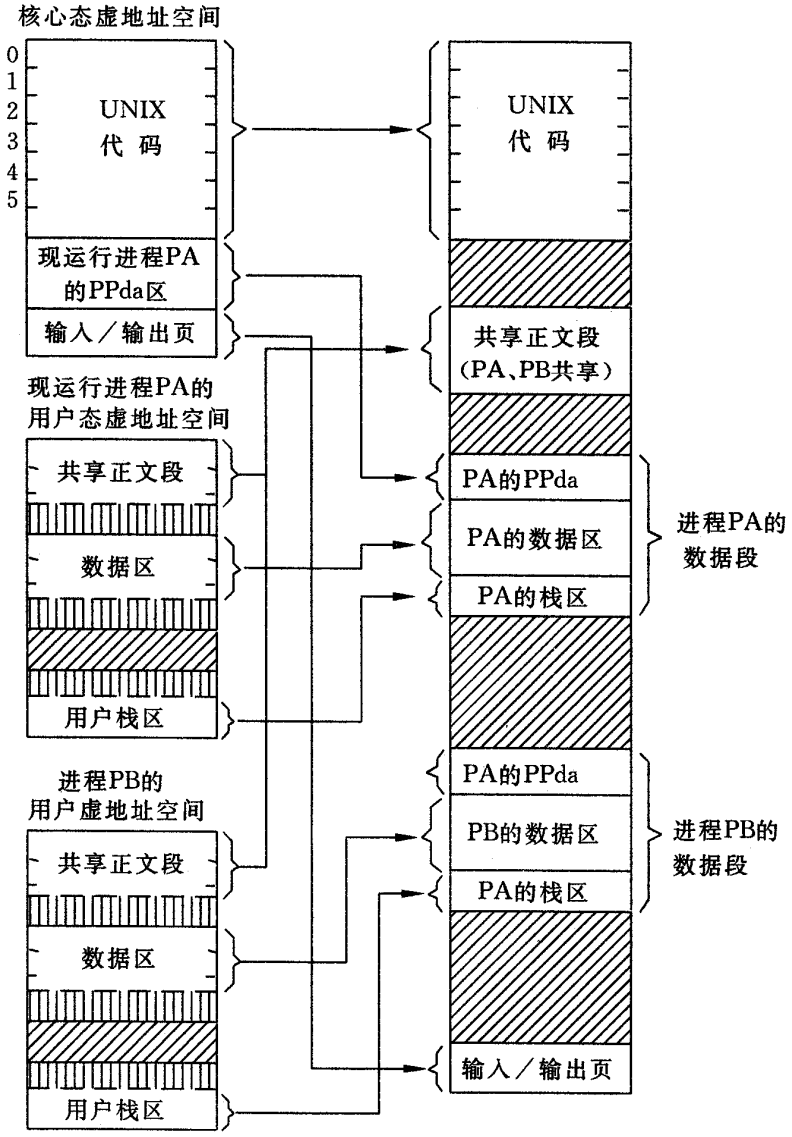


图 13-16　进程映像在内存中的实际分布

(1) 把核心态虚地址空间的 0~5 页装在内存 0 地址开始的低地址处，常驻内存；

(2) 把核心态虚地址空间的第 7 页装在内存最高地址处的 8K 空间里，常驻内存；

(3) 根据当前存储分配情况，以内存块为单位把共享正文段放在一个连续区内；

(4) 根据当前存储分配情况，以内存块为单位把核心态虚地址空间中的 1024 字节的 ppda 内容、用户态虚地址空间中的数据区、以及用户栈区放在一个连续区内，构成

进程映像的数据段。由于这恰是进程的非常驻内存进程映像，所以这种存放有利于它们对盘对换区的调入和调出；

(5) 将进程映像全部在内存的 进程的共享正文段的起始块号送入相应 text 结构的 x-caddr 保存，将数据段的起始块号送入 proc 结构的 p-addr 保存。

(6) 如果现在调度到进程 pa 运行，则第一，要把该进程的 ppda 地址（在 proc 结构的 p-addr 里）送入核心态活动页寄存器组的 APR6 里；第二，根据共享正文段起址 x-caddr、数据段起址 p-addr、以及 user 里数组 uisa 和 uisd 记录的相对虚、实地址映照表内容，得到该进程用户态虚地址空间八个页的实际虚、实地址映照表，并送入用户态活动页寄存器的 APR0~APR7 保持。至此，就能保证该进程的正常运转了。

13.3.3 UNIX 存储管理及分配释放策略

UNIX 存储管理的对象为两类：一是内存中除了 UNIX 代码以及输入、输出页所占区域之外的部分；一是在磁盘上专门开辟的对换区。它们实际上都是为了存放进程映像非常驻内存部分的，因此对这两部分资源，UNIX 采用了同样的分配和释放算法，差别只是在于对于内存，分配和释放以内存块大小为单位，每个内存块为 64 个字节；对于盘对换区，分配和释放以磁盘块为单位，每个磁盘块为 512 个字节。

1、可用存储区表

为了管理内存和盘对换区的可用存储区，UNIX 系统设置了内存可用存储区表 coremap 和盘对换区可用存储区表 swapmap，每个表均含有 50 个表目，每个表目由两部分内容组成：

- m-size 记录了一个空闲内存或磁盘可用区域的大小；
- m-addr 记录了一个空闲内存或磁盘可用区域的起始地址。

因此，每个表目反映了一个内存或磁盘可用区域的情况。

这两个表按照可用存储区起始地址由小到大进程登记，未使用的表目用 m-size 为 0 来标识成为空白项，它们显然都集中在表的后半部分。为了便于管理，表中最后一个表目项必须为空白项。

2、存储区的分配——malloc

UNIX 在接到一个存储申请时，采用最先适应策略来进行存储空闲区的分配。方法是从相应可用存储区表的第一个表目开始，用申请的尺寸与表目中的 m-size 比较，第一个大于或等于申请尺寸的 m-size 即为所求。在按单位分配后，要对可用存储区表进行调整，如果尺寸与所申请量恰好相等，则表中随后各表目都要随之上移，整个分配工作是由 malloc 程序实行和完成的。

3、存储区的释放——mfree

在对存储区进行释放时，先根据释放存储区的起始地址，在相应可用存储区表中找到它的插入位置，然后根据插入位置处的表目以及前一个表目的信息，判别被释放的存储区是否可以与它们合并成一个可用存储区。随之根据不同情况调整相应可用存储区表，以完成释放工作。整个释放工作是由 mfree 程序完成的。

这里要注意的是，存储区的分配和释放都要在相应的可用存储区表上进行操作。也就是说，两个可用存储区表 coremap 和 swapmap 是分配和释放的共享变量，所以在 malloc 和 mfree 两个程序里，涉及到对 coremap 和 swapmap 操作的那部分程序段就是临界段，这两部分的执行必须要保证互斥。

13.4 UNIX 操作系统的文件管理

13.4.1 UNIX 文件系统的基本工作原理

1、UNIX文件的逻辑结构及分类

从总体上看，文件的逻辑结构可以分成两种形式，一是记录式文件，一是无结构的流式文件。记录式文件是一种有结构的文件，这种文件由记录组成，每个记录又包含若干数据项，它们彼此之间具有一定的关系。如果文件中每个记录的长度都相等，则称为定长记录文件，否则称为变长记录文件。所谓无结构的流式文件，即是把文件视为一个无内部结构的字符流。UNIX 系统中文件的逻辑结构就是采用这种形式，并把它们分为：

- 一般文件：这是通常意义下的磁盘文件，系统总把它视为无结构、无记录概念的字符流，文件的长度可以动态地增长。作为用户，可以根据自己的需要，在处理过程中把它们构造成有结构的文件。系统软件、用户自己编写的各种原程序以及可以执行的二进制目标程序等都属此列。
- 目录文件：由文件的目录组成的文件称为目录文件。一般来讲，文件的目录应该包括：文件名、文件长度、文件在存储设备上的物理位置、文件类型、存取权限、管理信息等内容。在 UNIX 里，为了加快文件目录的搜索速度，便于实施文件共享，而把这些内容划分为两部分：一部分称为该文件的文件控制块（或索引节点）inode，它包含了文件的长度、物理位置、文件组、文件类型、存取权限、共享信息、管理住处等内容；另一部分仍称为该文件的目录，它只含文件名以及相应 inode 节点的编号（见图 13-17）。因此，UNIX 的目录文件虽也是由文件的目录组成，但相比之下要比通常所说的目录文件简单许多。
- 特殊文件：在 UNIX 系统中，把块存储设备（如磁盘）和字符设备（如终端机）都视为文件。但为了与前两类文件有区别，故统称它们为特殊文件。考虑到检查和处理方便起见，系统把所有特殊文件都放在名为“dev”的目录文件中。例如，行式打印机特别文件可写成/dev/lp，终端机特别文件可写成/dev/con (con=console)。

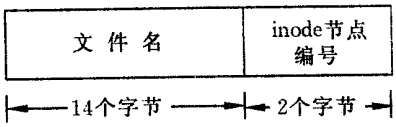


图 13-17　UNIX 文件目录的格式

2、基本文件系统及可装卸的子文件系统

UNIX 文件系统可分成基本文件系统和可装卸的子文件系统两部分（见图 13-18）。

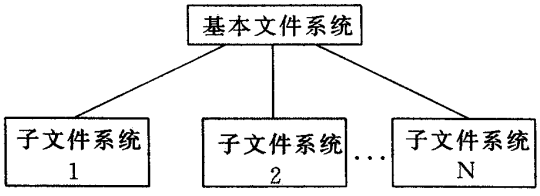


图 13-18　UNIX 文件系统的基本结构

1) 基本文件系统

基本文件系统固定在根存储设备上，是整个文件系统的基础。通常把硬盘做为根存储设备，系统一旦启动运行，基本文件系统就不能脱卸。

2) 可装卸的子文件系统

存储在可装卸存储介质（如软盘）上的文件系统为可装卸的子文件系统，它可以随时更换。每个用户都可以把自己的文件存放在软盘上，使用时插入软件驱动器，然后通过系统调用命令将其与基本文件系统勾连在一起，也可以用系统调用命令使子文件系统与基本文件系统脱勾。

UNIX 文件系统的这样一种结构，使其使用灵活，便于扩充和更改。

3) 文件系统的目录结构

在 UNIX 文件系统里，基本文件系统和子文件系统都独立采用树型带勾连的目录结构。所谓树型，即它们各自都有一个根目录文件，在根目录文件中所列的文件，可以是一个目录文件，也可以是一个一般文件或特殊文件。这样一层层地发展下去，就形成了一个通常意义下的树型文件目录结构。在这种结构下，叶节点为一般文件或特殊文件，中间节点为目录文件。图 13-19 是 UNIX 文件系统的目录结构图，我们以方框代表目录文件，圆圈代表一般文件或特殊文件。

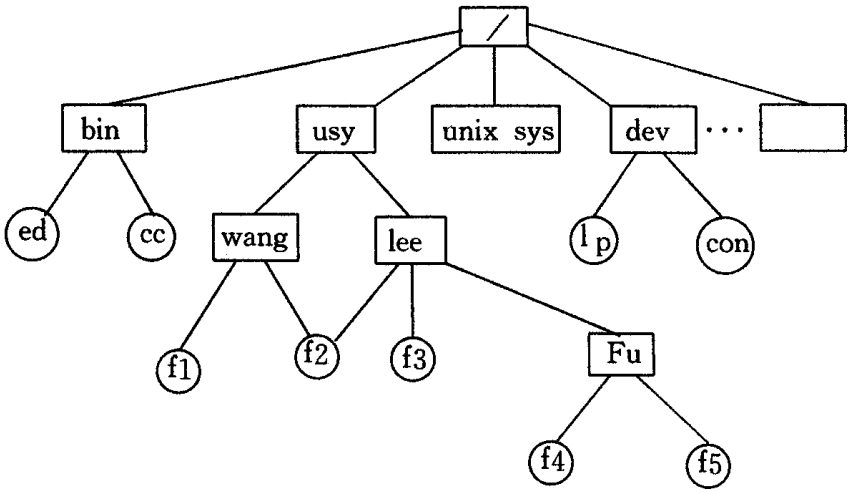


图 13-19　UNIX 文件系统的目录结构

目录文件中的每一个文件都有一个文件名，最大长度为 14 个字符（即 14 个字节）。

UNIX 在树型结构的基础上增加交叉连接部分，以达到文件共享的目的。在 UNIX 系统中，是通过文件的 inode 节点来实现文件共享勾连的，并且只允许勾连到代表一般文件的叶节点上去。由图 13-19 可知，wang 和 lee 共享文件 f2。

还要注意的是，在基本文件系统和子文件系统之间不能使用这种方式来共享文件。

13.4.2 UNIX 文件系统的数据结构综述

在 UNIX 文件系统的实施过程中，涉及到多种数据结构。有一类数据结构用于对文件的静态管理，因此都分布在文件所在的存储设备上，它们包括外存文件控制块 inod、目录，以及存储资源管理信息块 filsys 三种；另一类数据结构用于文件打开时的管理，因此都出现在内存，它们包括内存文件控制块 inode、打开文件控制块 file、以及进程打开文件表三种。下表分别简介之。

1、外存文件控制块inode

由前知，文件存储设备上的每一个文件，都有一个文件控制块 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，这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 inode 结构理用 32 个字节，共九项内容，反映出一个文件的如下信息：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。下面给出与我们讲述有关的三项。

(1) i-mode　文件的各种属性

用各位来表示文件的不同属性，譬如此索引节点是否被占用，此索引节点对应的是何种类型的文件，以及对该文件的存取权限等。

(2) i-nlink　文件勾连数

这项表明在文件目录结构中，该文件具有的路径名个数。由于在 UNIX 中，只允许对叶节点产生交叉连接，因此一个文件系统从根出发到任何中间节点（目录文件），只能存在一条路径，但从根出发到叶节点则可能出现多条路径。于是，对应于某个一般文件的 inode 中的 i-nlink，反映了对该文件的共享情况。

(3) i-addr[8]　存放文件所在物理块号的基本索引表

UNIX 文件的物理结构，根据文件规模的不同而采取索引结构或多级索引结构。i-addr[8]是一个有八个元素的数组，它构成了文件逻辑块号和物理块号的基本对照表。如果文件规模超出八个物理块，则可在它的基础上扩展成一级间接、二级间接的索引表，详情见后面关于 UNIX 文件物理　结构部分的讲述。

2、目录和目录文件

1) 目录

UNIX 中的每个文件都有一个目录项，其格式如图 9-22 所示，目录项中记录了文件的名字以及该文件对应的外存 inode 的编号。文件名是一个文件的外部标识，而这个文件的外存 inode 编号，则是它的内部标识。可以看出，文件的目录项建立起了文件内、外部标识之间的对应关系：根据文件名找到它的目录项，由目录项的外存 inode 编号找到文件控制块 inode，从而获得该文件的信息。

2) 目录文件

UNIX 视每张目录表为一目录文件。作为一个文件，它有自己的名字以及对应的外存 inode。要注意的是，每个文件系统（基本的或子文件系统）都有一个根目录文件，它的外存 inode 总是放于文件存储设备上 inode 区中的第一个，于是保证很容易从它出发，到达树型目录结构上的任一节点。另外还要注意的是，由于每一个目录项需要 16 个字节的存储空间，每个盘块的容量为 512 字节，因此存放目录文件的盘块中，每一盘块可以存放 32 个文件的目录。有了这些，UNIX 文件目录的树型结构可以细化成如图 13-20 所示。

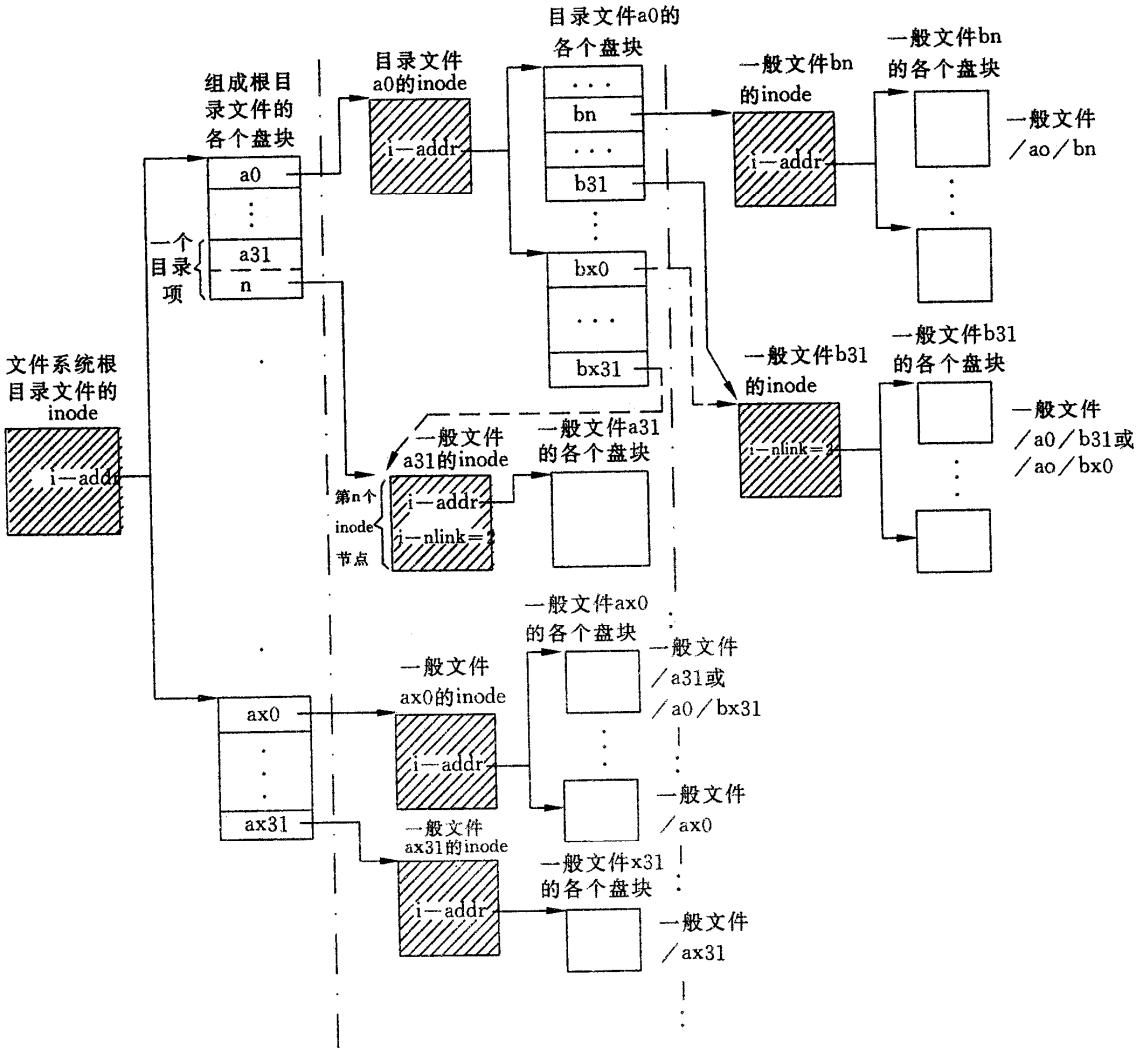


图 13-20　UNIX 树型目录结构的细化

3) 文件目录中的勾连

为了实现文件共享，UNIX 允许对一般文件节点实行交叉连接，这称为勾连。它是通过在同一文件系统中的两个不同目录项里填入同一个外存 inode 节点编号来实现的，图 9-25 中的虚线处反映的正是这种勾连。譬如说，一般文件的原有路径名为/a31，则在目录文件中就应有一目录项，其文件名为 a31，对应的外存 inode 编号为 n。如果再给它另起一个路径名/a0/bx31，则在目录文件 a0 中应该新设置一个目录项，它的文件名部分填入 bx31，它的外存 inode 编号仍填为 n。另外，在编号为 n 的外存 inode 里，将 i-nlink 的值加 1。于是文件系统中就有两个目录项同时指向这一个 inode，实现了对文件 a31 的不同名共享。

3、存储资源管理信息块

文件系统所在存储设备上的存储资源，主要有两个用途，一是用来存放外存文件控制块 inode，一是用来存放文件信息或扩展的地址索引表(当文件是大型的巨型的时，就需要这样做)。为了能对盘块的使用情况加以管理，UNIX 将这些管理信息集中放在

一个数据结构——存储资源管理信息块 filsys 中。filsys 总是固定在 1#盘块上，这一盘块通常称作该文件系统的管理块。这样，整个磁盘空间的安排就如图 13-21 所示。

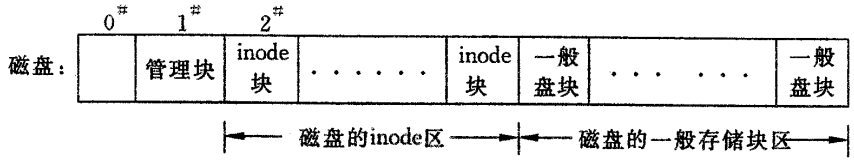


图 13-21　文件系统磁盘存储区的分布图

要注意的是，为了使用方便，基本文件系统以及与基本文件系统装连上的子文件系统都要申请一个缓冲存储区，将自己的管理块 filsys 复制到里面，以便内存中有它们的副本。

由前知，每一个文件的 inode 节点占用 32 个字节，因此每一个 inode 块包含 16 个文件控制块。这些 inode 顺序编号，一个文件占用了某 inode，则其编号就成为这个文件的内部标识，第 1 号 inode 是专门用于根目录文件的。

数据结构 filsys 共有 12 项内容，下面给出与我们讲述有关的六项。

(1) s-isize　inode 区占用的盘块数；

(2) s-fsize　盘块总数；

(3) s-nfree　直接管理（也就是 s-free[100]指向）的空闲块数；

(4) s-free[100]　空闲块索引表

(5) s-ninode　直接管理的空闲 inode 节点数；

(6) s-sinode[100]　空闲 inode 节点索引表。

至于如何通过 filsys 来对空闲 inode 和空闲盘进行具体管理，详情见后面关于 UNIX 文件系统资源管理综述部分。

4、内存文件控制块inode和内存inode表

外存 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，于是它就要不断来回于内、外存之间，这当然是极不经济的。为此，UNIX 在系统占用的内存区里开辟了一张表——内存 inode 表（或活动文件控制块表、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文件控制块 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 的大部分内存拷贝到这个内存 inode 中，随之就使用这个内存 inode 来控制磁盘上的文件。在最后一个用户关闭此文件后，内存 inode 的内容被写到外存 inode，然后释放以供它用。

内存 inode 的结构基本上与外存 inode 相同，仅略有增减。增加的有关项目有：

- i-count：内存 inode 访问计数。若为 0，表示此节点为空闲，某文件被打开时，其内存 inode 里的此项就加 1。只有所有用户都关闭了此文件，以使 i-count 为 0 后，这个文件才被真正关闭。
- i-number：与此内存 inode 相对应的外存 inode 编号。

5、打开文件控制块file和file表

一个文件可以被同一进程或不同进程、用同一路径名不同路径名、具有同一特性或不同特性（读、写、执行）同时打开。做为内存 inode，基本上只包含文件的静态信息（如文件的物理结构、勾连情况、存取权限等），没有记录下一个文件被打开时的动态信息。为此，UNIX 又在系统占用的内存区里开辟一张表）——打开文件控制块表，简称 file 表，共有 100 个表目，每一个表目称做一个打开文件控制块 file。主要内容有

- f-flag 打开文件的特性，指明对文件的读、写要求
- f-count 共享该 file 的进程数
- f-inode 指向对应于此打开文件的内存 inode

当打开一个文件时，都要在该表里申请分配一个 file，形成该打开文件的控制块。

6、进程打开文件表

在进程的 user 结构里，设有一个整型数组 u-ofile[15]，它被称为该进程的打开文件表。在进程打开一个文件时，分配到一个打开文件控制块 file，就把这个控制块的地址填入 u-ofile[ ]的一个元素里。于是，这个打开文件就和它在 u-ofile[ ]中占据的位置形成一个对应关系，这个位置就是打开文件号。图 13-22 给出了 UNIX 文件系统中各数据结构之间的关系。

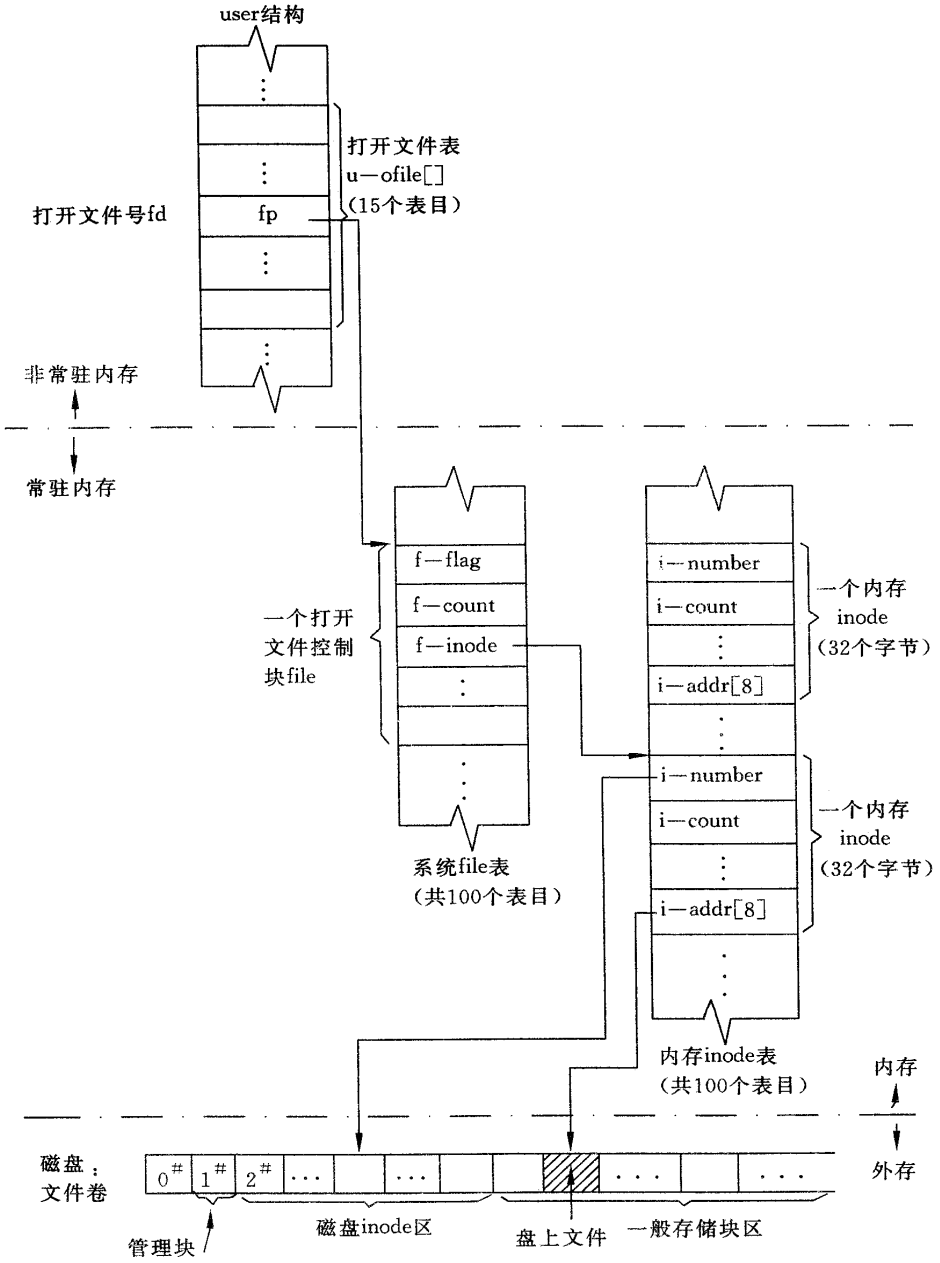


图 13-22　UNIX 文件系统各数据结构间的关系

当某进程欲打开一个文件时，首先在自己 user 结构的打开文件表里申请一个打开

文件号 fd（即找到一个 u-ofile[ ]中的空表目），随之在系统打开文件控制块表里申请一个 file 结构，且将其位置 fp 填入 u-ofile 表目中。然后在内存 inode 表里找到该文件的内存 inode，或申请一个内存 inode，并将位置信息填入相应 file 中的 f-inode, i-count 加 1。这样，如果该文件的内存 inode 原先就在内存 inode 表中，那么现在这个进程则又一次通过同一路径名不同路径名将它打开。由于它们都有自己的 file 结构，所以它们对这个文件可以持有不同的操作要求，拥有各自的读、写指针，从而形成了通过不同的 file 结构，使用一个内存 inode 的共享形式。这种共享在 file 结构里的 f-count 都为 1，但因大家都指向同一个内存 inode，故内存 inode 里面的 i-count 则大于 1（有几个 file 结构指向它，它的 i-count 就为几），图 13-23 描述了这一情形。

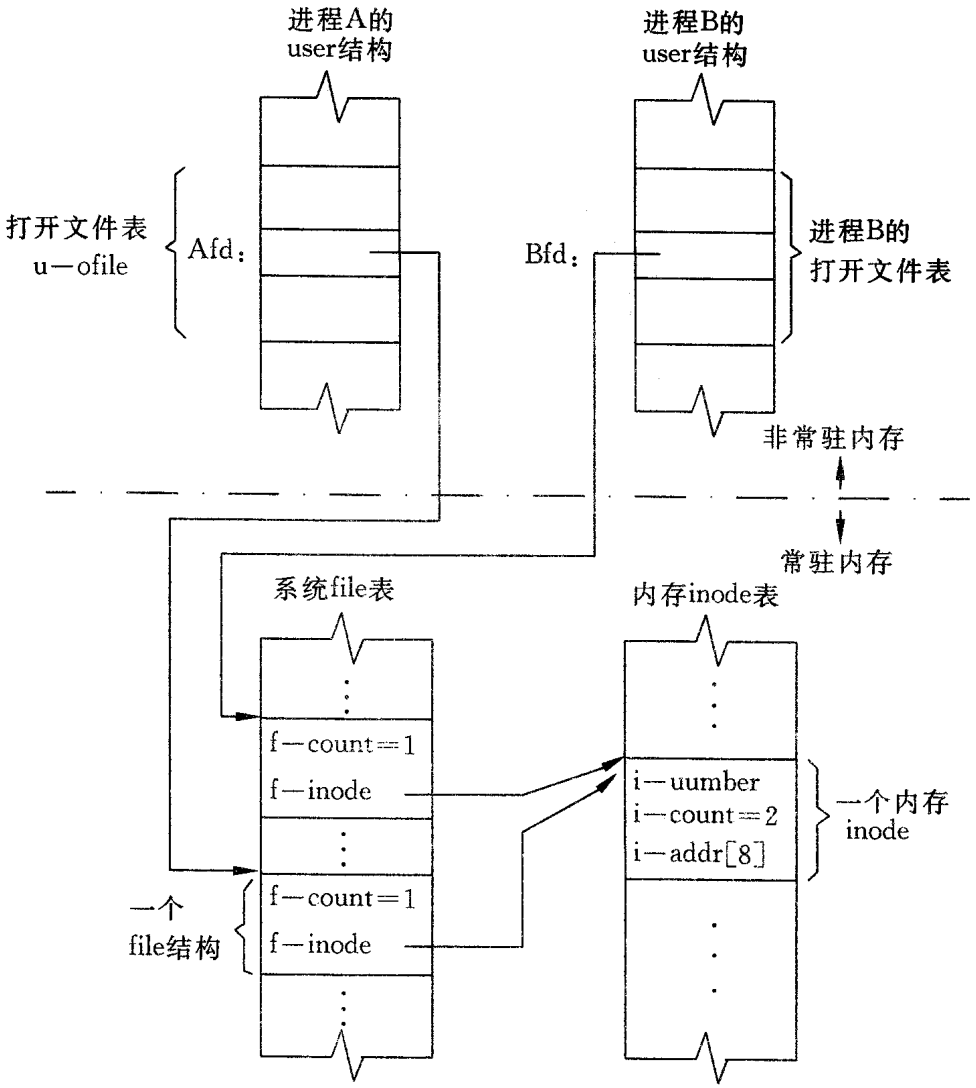


图 13-23　不同进程共享打开文件示意图

共享打开文件的另一种情形是由父进程创建子进程引起的。在 UNIX 中，当父进程创建一个子进程时，先要继承父进程的 u-ofile 表的全部内容。这样它和父进程使用同一个 file 结构，对这个文件有相同的操作要求和读写指针。所以，这种共享打开文件表现为通过共享同一个 file 结构来体现，图 13-24 描述了这一情形。由此可知，在 UNIX 里提供了两种文件共享的方式，第一种是在目录结构里通过勾连，对同一文件提

供不同路径名，以达到能够异名共享的目的；第二种是在打开文件结构里，通过共享同一个 file 结构或共享同一个内存 inode 而实现对打开文件的共享。

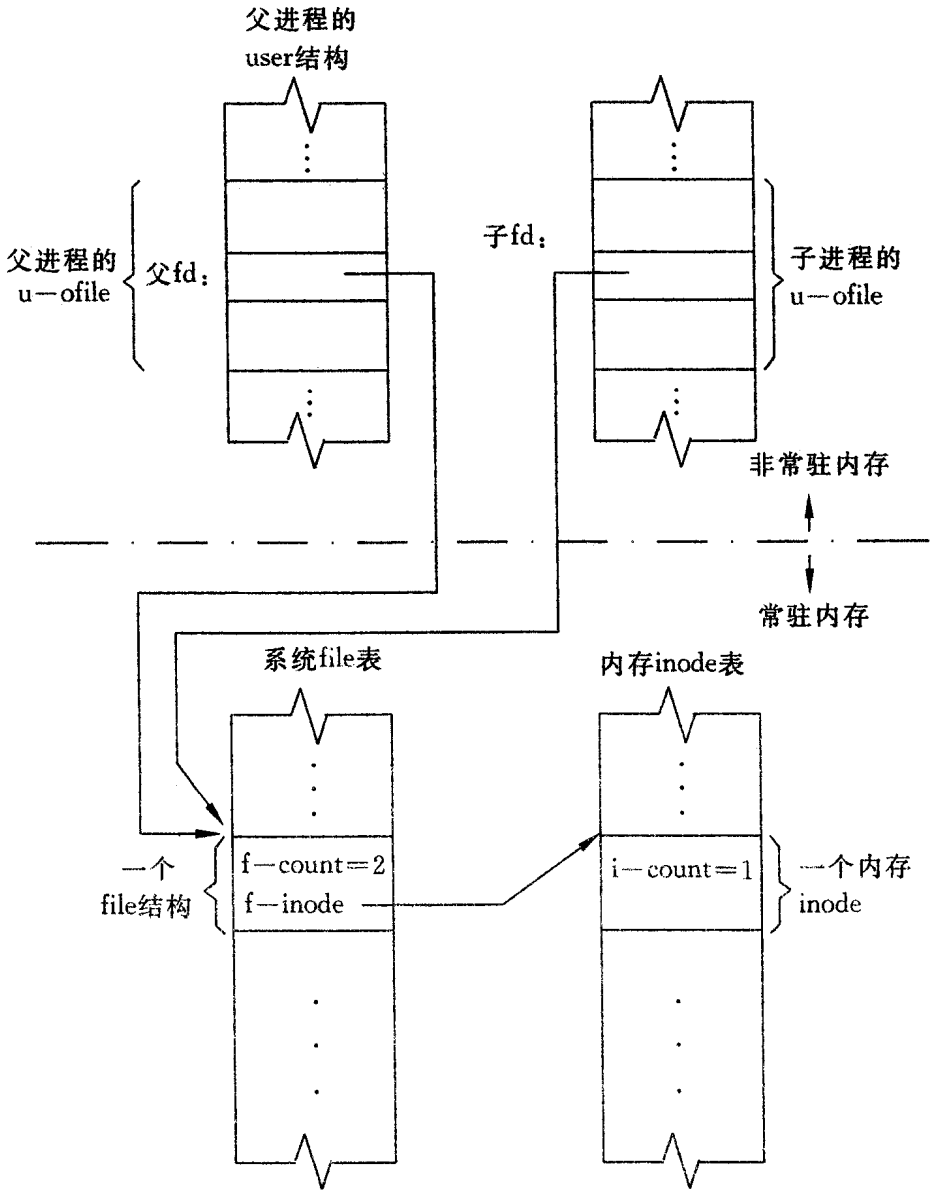


图 13-24　父、子进程共享打开文件示意图

13.4.3 UNIX 文件的物理结构

前面已经提及，UNIX 文件采用的是索引式物理结构，并且也知道有关文件物理位置的索引信息是由该文件外存 inode 中的八个数组元素 i-addr[8]给出的。然而 UNIX 在具体的实施中，并没有受一般索引结构的限制，它把自己的文件根据长度的不同分为小型文件、大型文件、以及巨型文件三种，分别通过一级索引、二级索引和三级索引来加以实现。

1、小型文件的索引结构

文件存储设备以盘块为单位进行存取，每块 512 个字节。当文件长度在 1~8 个盘

块之间时，称为小型文件，数组 i-addr[ ]就是通常意义下的地址索引表，它里面的内容就是文件在盘中的物理块号。因此，小型文件是通过 i-addr[ ]的一级索引而找一盘文件的。

2、大型文件的索引结构

当文件长度在 9~7×256 个盘块之间时，称为大型文件，此时的数组 i-addr 只使用七个元素 i-addr[0]~ i-addr[6]，形成一个间接索引表，每一个指向一个盘块，它们才是真正的地址索引表。由于一个盘块包含 256 个字，所以通过这七盘块的索引，可最多得到 7×256 个盘块。由此可见，大型文件是通过二级索引而找到盘文件的。

3、巨型文件的索引结构

当文件长度在（7×256+1）~（7×256+256×256）个盘块之间时，称为巨型文件，这时 i-addr 前七个元素的作用不变，而把 i-addr[7]用来进行扩充。即把它指向的盘块作为间接索引表，再指向的 256 个盘块才形成真正的地址索引表，所以巨型文件有一部分是通过三级索引而找到盘文件的。

13.4.4 UNIX 文件系统的资源管理综述

为了实施文件系统，需要涉及众多的资源。综前述，这些资源有如下几种：系统打开文件控制块表（file）、系统内存 inode 表，用户打开文件表 u-ofile、外存 inode 区、以及外存一般存储块区。对于前三种资源的管理比较简单，都是采用线性搜索分配法，这里着重介绍后两种资源的管理方法。有关这两种资源的管理信息，都集中放在文件存储设备的存储资源管理信息块 filsys 中（它总是固定存放在 1#盘块内，并且在内存中都有各自的副本）。

1、外存inode区的管理

在 filsys 里，s-isize 记录了 inode 区所占用的盘块数。由于每个 inode 占用 32 个字节，因此可以得知在该 inode 区里共有多少个 inode 节点。由于存储设备上创建一个文件就需要有一个 inode 节点与之对应，删除一个文件时，它占用的 inode 节点就被系统收回，所以 inode 区中空闲 inode 的数量是动态变化的。

系统按照如下规定来实现对空闲 inode 的管理：

(1) 在 filsys 里，开辟一个空闲 inode 索引表：s-inode[100]。它是一个具有 100 个元素的数里，每个元素可指向一个空闲 inode，这里系统直接管理的空闲 inode。至于当前该数组里究竟含有多少个空闲 inode，则由 filsys 里的 s-ninode 加以记录。

(2) 把 s-inode[ ]视为一个栈来使用。按照 C 语言的约定，数组下标总是从 0 开始，所以 s-nnode 的值恰好是一个可以使用的索引表目的下标。当需要分配 inode 时，如果 s-ninode 不为 0，则将 s-inode[--s-ninode]里指示的 inode 节点分配出去；如果释放回一个 inode 节点，则把该节点指针送入 s-inode[s-ninode++]中。

(3) 如果 s-inode[ ]已无直接管理的空闲区了（s-ninode=0），则搜索 inode 区，将找到的空闲 inode 依次登入，直至表满或搜索完整个 inode 区。如果 s-inode[ ]已经直接管理了 100 个空闲 inode，则对再释放的 inode 不作任何处理，让这个空闲的 inode 散布在 inode 区里。

2、外存一般存储块区的管理

由上述可知，系统对文件存储设备上 inode 区里的空闲 inode，通过 s-ninode 和 s-inode[100]只直接管理最多 100 个空闲 inode，置其它空闲 inode 而暂时不顾。

在 filsys 里，对一般存储块区也开辟了两个项目：

• s-free[100]　空闲块索引表

• s-nfree　直接管理的空闲块数目

形式上，它们与 s-inode[100]、s-ninode 相似，但实际上却采用了不同的管理方法——分组链接法。

1) “分组链接”法的基本思想

把一般存储块区里的空闲块归成若干组，第一组为 99 个空闲块，从第二组起均为 100 个空闲块，最后一组可能会不足 100 个空闲块。为了进行链接，规定每个后组的第一个空闲块的开头 101 个字用来记录其前组的空闲块数目以及每一空闲块的位置。由于最后一组不可能还有后组，故将最后一组的有关信息存放在管理块 filsys 的 s-nfree 和 s-free[100]中。于是就形成了如图 13-25 所示的空闲块分组链式索引（那里假定最后一组共有 52 个空闲块）。

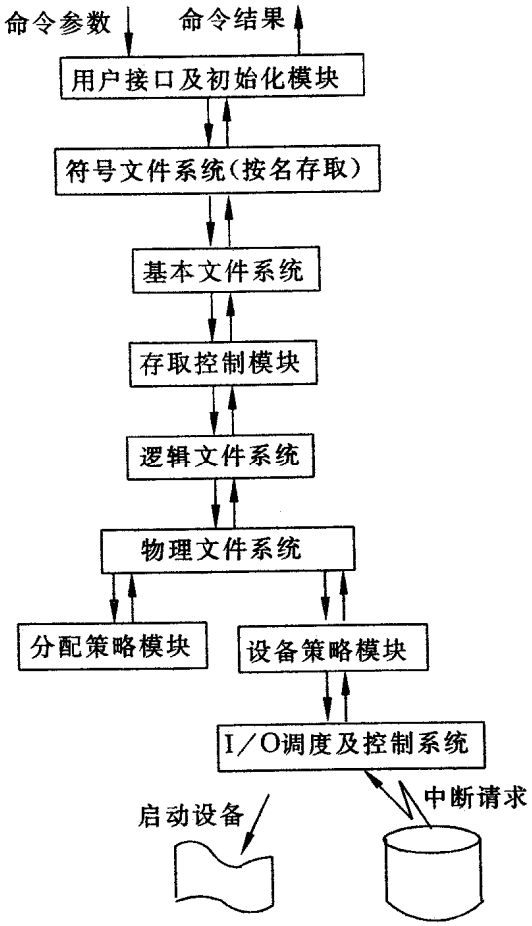


图 13-25　文件存储设备空闲块分组链式索引

2) 空闲块的分配

分配总是在 filsys 中的空闲块索引表 s-free[ ]中进行，并把它视为一个栈结构来使用。如前面空闲 inode 管理中所说，这里的 f-nfree 之值也恰是下个可以使用的索引表目的下标。因此，当 s-nfree 不为 0 时，总是把 s-free[--s-nfree]里指示的空闲块分配出去。这里要注意的是，如果在——s-nfree 操作后 s-nfree 为 0，则表明是把 s-free[0]中指示的当前这组的最后一个空闲盘块分配出去，而这一盘块里却存放有它前面一组空闲块的信息。因此在把它分配出去之前，应该先把它前面的 101 个字中的内容移入 s-nfree 和索引表 s-free[100]中。

另外还要注意的是，第二组的 n-free=99, free[0]=0。这两个值是判定空闲块全部分配完的重要信息。设想现在已把第三组及其后面各组管理的空闲块都分配出去了，于是 filsys 中的 s-nfree=99, s-free[0]=0, s-free[1]~s-free[99]将指向第一组的 99 个空闲块。若把这 99 个空闲块也全都分配出去了，那么此时 s-nfree=1, s-free[0]=0。这时如果再提出空闲块的申请，按上面的办法去做时，s-nfree 减 1 为 0，于是可以发现 s-ree[0]=0。这就表明已到空闲块的分组链尾，没有空闲块可分配了。

3) 空闲块的释放

释放时的动作总是把释放的空闲块位置登入索引表的 s-free[s-nfree++]表目中。这里要注意的是，如果在登入前发现索引表已满，那么意味着在此前已收集到了 100 个空闲块，可以形成一个新的链组。而现在释放的空闲块是下一组的开始，因此应先把 filsys 里 s-nfree 的值和索引表 s-free[100]中的内容复写到新释放的这一空闲块的前 101 个字中，然后将此释放块的地址填入 filsys 的 s-free[0]中，置 s-nfree=1。

13.4.5 子文件系统的装卸

在 UNIX 中，每个用户都可以建立自己的子文件系统（或称文件卷），并可以把它装配到基本文件系统上去而融为一棵更大的树。不用时，也可以把装配上去的文件卷完整地拆卸下来，因此 UNIX 的文件系统显得十分灵活。

子文件系统与基本文件系统的这种装配连接，是通过所谓的装配块来进行的，它在这两者之间架起了联系的桥梁。一旦完成了这种装配连接，装配块所起的作用对用户而言是完全透明的。

1、装配块及装配块表

装配块是系统专为子文件系统装配到基本文件系统上去而开设的数据结构，它共有三项内容：

(1) m-dev　子文件系统所在存储设备号；

(2) m-bufp　指向子文件系统管理块 filsys 在内存的副本；

(3) m-inodp　指向基本文件系统中对应于子文件系统根目录文件的内存 inode。

这样的装配块总共有五个，形成了装配块表 mount[5]: mount[0]~mount[4]。在系统初启时，总是把 mount[0]用于基本文件系统，将根设备的信息登在其中，也就是说，mount[0]中的 m-dev 记录根设备号，m-bufp 指向根设备上 1#盘块中管理块 filsys 在内存的副本，m-inodp 指向与根设备上 inode 区的第一个 inode 相对应的内存 inode。余下的四个装配块 mount[1]~mount[5]可用于子文件系统的装配，所以 UNIX 最多可以允许同时装配四个文件卷于基本文件系统之上。

2、子文件系统的装配

子文件系统通过装配块而与基本文件系统连接，可以划分为两部分：一是装配块与基本文件系统的连接，一是装配块是子文件系统的连接。

1) 与基本文件系统的连接

对于一个文件系统而言，最关键的是得到它的根目录文件，因为从它出发，就可以得到该文件系统的全部目录结构。因此，要把一个子文件系统装配到基本文件系统上去，最重要的是在基本文件系统连接处的目录文件中设置一个目录项，该目录项对应于与其装接的子文件系统名，并指向一个 inode，这个 inode 对应于一个内存 inode。把内存 inode 的 i-flag 设置成“已装配”标志，说明它是用来连接某个子文件的。然后在装配块表中获得一个装配体，使其 m-inodp 指向这个内存 inode。这样，装配块就与

基本文件系统连接上了。

2) 与子文件系统的连接

把子文件系统的管理块 filsys 复制到某个缓冲存储区中，用装配块里的 m-bufp 指向它，另外把子文件系统所在设备号填入到 m-dev，这样就实现了装配块与子文件系统的连接。

5、文件搜索过程

若给出了某个文件的路径全名（即从基本文件系统根目录出发），开始的搜索一切如常。只是当遇到该子文件系统名对应的内存 inode 时，发现它的 i-flag 标志有“已装配”信息，表明它是用来与苛子系统装接的，于是转到装配块表，去从中寻找出哪一个装配块的 m-inodp 也是指向这个内存 inode。找到的装配块正是用来与子文件系统连接的那块。根据装配块中的 m-dev 可找到子文件系统所在的设备，从而获得其根目录文件的 inode。这样，由这个 inode 出发就可得到子文件系统所在的设备，从而获得其根目录文件的 inode。这样，由这个 inode 出发就可得到子文件系统的全部目录结构。通过使用由 m-bufp 指出的 filsys，可以对该子文件系统的物理资源加以管理。图 13-26 给出了示意图

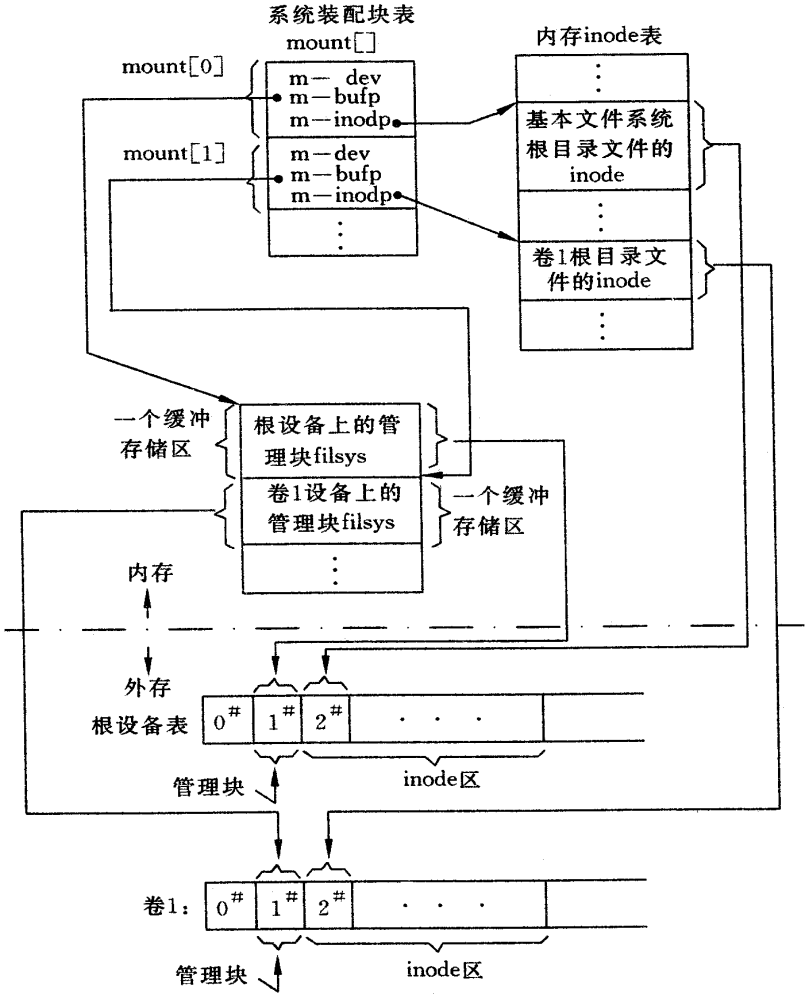


图 13-26　利用装配块连接卷 1 的示意图

13.4.6 文件的共享

所谓文件共享，就是不同用户或不同进程共同使用同一个文件，文件共享有时不仅为不同用户完成共同的任务所必须，而且还可以节省大量的外存空间，减少由于文件复制而增加的访问外存次数。

文件共享可以有多种形式。在 UNIX 系统中，允许多个用户静态地共享，或动态地共享同一个文件，下面就介绍 UNIX 系统的这两种共享文件方式。

1、文件的静态共享

在 UNIX 系统中，两个或多个用户可以通过对文件连接达到共享一个文件的目的。比如图 7.10 中，user 下的 sun 共享 liu 的文件 d2。共享时，可以用原名，也可以换另外的名字，如 D2，来访问同一文件。由于这种共享关系不管用户是否正在使用系统，其文件的连接关系，都是存在的，因此，我们称它为静态共享。用文件连接来代替文件的复制可以提高文件资源的利用率，节省文件的物理存储空间。

为了实现文件的连接，只要把不同目录的索引节点编号，指定为同一文件的索引节点即可。但为了反映共享同一文件的用户数，在每个索引节点中设立了一个变量 i_link。当文件第一次创建时，i_nlink 等于“1”。以后，每次连接时就把 i_nlink 加“1”。当用户每次删除文件（对于文件主）或解除连接（对其它用户）时，就把该计数值减“1”，直到发现其结果为零时，才释放文件的物理存储空间，从而真正删除这个文件。

2、文件的动态共享

所谓文件的动态共享，就是系统中不同的用户进程或同一用户的不同进程并发地访问同一文件。这种共享关系只有当用户进程存在时才可能存在，一旦用户的进程消亡，其共享关系也就自动消失。

在 UNIX 系统中，文件打开以后，对应的索引节点就在活动索引节点表中，活动索引节点表是整个系统公用的。为了使用户掌握他们当前使用文件的情况，系统在各个进程的 user 结构中设立了一个用户打开文件表，并通过它与各自打开文件的活动索引节点相联系。

由于 UNIX 系统中的文件是无结构的字符流序列，因此文件的每次读写都要由一个读/写位移指针指出要读写的位置。现在的问题是：若一个文件可以为多个进程所共享，那么应让多个进程共用同一个读/写位移，还是应让各个进程具有各自的读写位移呢？下面分两种情况进行讨论。

在 UNIX 系统中，进程可以动态地创建，被创建的进程完全继承了父进程的一切资源，包括 user 结构中用户打开文件表中指出的打开文件。通常同一用户的父、子进程往往要协同完成同一任务，若使用同一读/写位移，那么一个进程改变它时，另一个进程能够感觉到它的变化。这样，使父、子进程更容易同步地对文件进行操作。因此，该位移指针宜放在相应文件的活动索引节点中。此时，当用系统调用 fork 建立子进程时，父进程的 user 结构被复制到子进程的 user 结构之中，使两个进程的打开文件表同指向同一活动的索引节点，达到共享同一位移指针的目的。

另一方面，若一个文件为两个以上的用户所共享，必然是每个用户都希望能独立地读、写这个文件，彼此不相干扰。显然，这时不能只设置一个读写位移指针，而必须为每个用户进程分别设置一个读、写位移指针。因此，位移指针应放在每个进程用户打开文件表的表目中。这样，当一个进程读、写文件，并修改位移指针时，另一个进程的位移指针不会随之改变，从而使两个进程能独立地访问同一文件。

可见，在上述两种动态共享方式中，对读写位移指针的设置位置和数目是不同的。

为了解决这一矛盾，同时满足这两种共享要求，系统又设置了一个“系统打开文件表”。该表共有 100 个表目，为整个系统所公用。在该表的每个表目中，除了含有读写位移指针 f_offset 之外，还有文件的访问计数 f_count，读写标志 f_flag 和指向对应文件的活动索引节点指针 f_inode，其中 f_count 指出使用同一系统打开文件表目的进程数目，它是系统打开文件表目资源能否释放的标志。f_flag 指出打开文件是为了读还是写，它为进行访问权限验证之用，这一点以后还要进一步介绍。用户打开文件表、系统打开文件表和活动索引节点表之间的关系如图 13-27 所示。

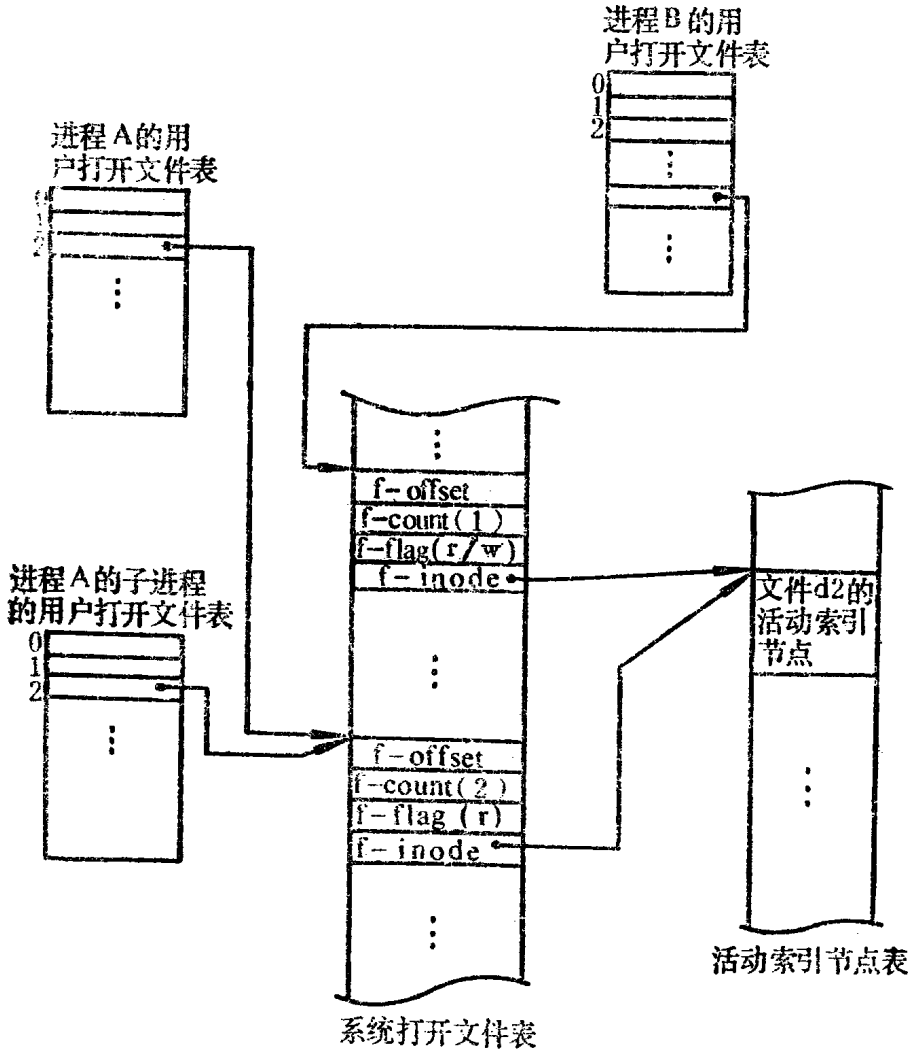


图 13-27　用户打开文件表、系统打开文件表和活动索引节点表之间的关系

图中，进程 A 和它的子进程通过同一个系统打开文件表表目共享文件 d2，因而其 f_count 为“2”。而进程 B 自己独立使用一个系统打开文件表表目，因而其 f_count 为“1”，但它却通过同一个活动索引节点与进程 A 及其子进程共享文件 d2。图中用户打开文件表表目的序号称为文件描述字，以后进程就是通过文件描述字对相应的文件进行读、写操作。下面看一下系统打开文件表目的申请和释放过程。

当一个进程要求打开一个文件时，系统首先要为它申请一个系统打开文件表表目，并建立该表目与相应文件活动索引节点的联系，然后把用户打开文件表表目中的一个空闲项指向它。如果在这之后，用户进程通过系统调用 fork 建立一个子进程，系统自

动把用户打开文件表目所指的系统打开文件表目中 f_count 加“1”。反之，当一个进程关闭一个文件时，系统不能简单地释放系统打开文件表目，而必须首先判断 f_count 的值是否大于 1。如果大于 1，说明还有进程共享相应的系统打开文件表目，此时只须把 f=count 减 1 即可。由上述过程可知，进程之间到底采用哪一种方式动态地共享同一文件，主要是由执行“fork”和“打开”系统调用而打开同一名字的文件，则系统分别为它所分配不同的系统打开文件表目，并指向同一活动索引节点，这时，这两个进程独立地使用各自的读、写位移量指针。但如果进程在打开一个文件之后，又用 fork 系统调用建立了一个子进程，由于子进程的用户打开文件表表目是由父进程复制得到的，那么它自然会指向父进程使用的系统打开文件表表目，因此这两个进程就共用同一个位移量指针来共享一个文件。

13.4.7 文件卷的动态安装

在一般的文件系统中，总的文件存储空间是不能动态变化的。而在 UNIX 系统中，则可以动态地装卸文件卷。

在系统初启时，系统中只有一个安装的文件卷，即所谓根文件卷，其上的文件是保证系统正常运行的最小文件集。如汇编语言和 C 语言编译程序、文件卷构造程序、终端命令解释程序等等。这个根文件卷在系统运行过程中是不可卸下的，所以它是系统的基本部分。

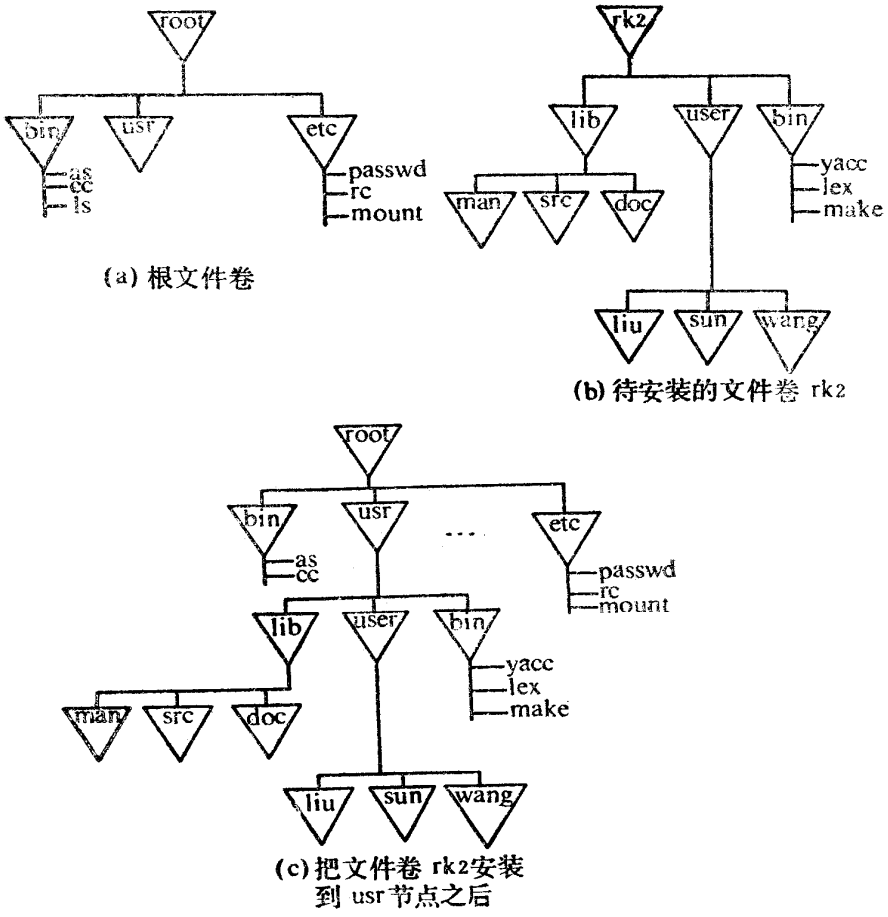


图 28　文件卷的安装

其它的文件卷可以根据需要，作为子系统动态地安装到一个“已存”文件卷的某

一个节点上。通常，这个节点是为文件卷安装而特地创建的一个空目录。而“已存”文件卷是指根文件卷，或者是已安装的别的文件卷。因为每个文件卷本身有一个完整独立的目录树结构，所以文件卷的安装就如同树的“嫁接”一样，使安装的文件卷目录树结构与原有的文件卷树形结构融为一体。这一过程如图 28 所示。

当然，也可以把“安装”上去的文件卷整个地“拆卸”下来，从而恢复“安装”前的状态。

文件卷的装卸不同于存储介质（盘、带）的装卸，因为文件卷卸是从逻辑意义上说的，与文件卷所在的存储介质本身的装卸有本质的不同。即使物理介质本身在工作，但若其上的文件卷没有安装好，系统也无法存取其中的信息。

一旦文件卷安装好，就要把这个文件卷的专用块复制到内存专用块中。以后，当文件操作要进行资源分配和释放时，就要使用这个内存区。因此，即使多个文件卷组成了一个统一的树型结构，但分配和释放资源仍然是各自独立进行的。

文件卷的这种动态装卸有以下优点：

(1) 可以随时扩充文件系统的内存空间。例如用户又买了一个磁盘时，可以把它构造之后安装到文件树结构中。

(2) 可以加强对文件的保护。为了防止一个文件卷上的信息受到有意或无意的破坏，可以把暂时不用的文件卷卸下，使用户不能访问其上的信息。

13.4.8 UNIX 文件操作的系统调用

到目前为止，我们已经介绍了文件的组织管理及文件系统的结构。在本节中，我们通过 UNIX 系统的使用，来进一步加深理解前述的概念和技术。这些系统调用主要包括文件的打开和关闭，文件的创建和删除，文件的连接和解除连接，文件的读和写，以及文件的随机访问。用户可以通过文件系统提供的系统调用在其程序中对文件进行上述操作。另外，有关文件操作的实用程序，实际上也是通过上述系统调用来实现的。

1、文件的创建和删除

当文件还不存在时，即文件还未在文件卷中登记时，需要创建；或者文件原来已经存在，有时需要重新创建。注意这同文件的打开是完全不同概念。文件打开是指当文件已经存在，需要使用时先执行打开，以便建立用户进程与文件的联系。相应地，文件不再有人需要，则要删除之，以便腾出存储空间。这同文件“关闭”也是完全不同的概念。文件关闭是指用户暂时不使用文件时，通过关闭操作切断用户进程与文件的联系。

1）文件的创建

文件创建首先是要求文件系统为新的文件建立一个新目录项和相应的索引节点，以便随后的写操作为这个新文件输入信息。该系统调用的 C 语言格式为：

```
int fd, mode;
char * filenamep;
fd = creat (filenamep, mode);
```

其中参数 filenamep 是指向要创建的文件路径名字符串指针；参数 mode 是文件创建者提出的该文件应该具有的存取权限。在文件成功地创建之后，这个权限就记录在相应索引节点的 i_mode 之中。变量 fd 中的内容是当创建成功之后，系统返回给用户的文件描述字，即用户打开文件表项的编号。由此可见，creat 兼有文件的“打开”功能，于是随后的文件写系统调用，就可使用这个 fd 进行操作。

例如，用户文件的路径名是/usr/lib/d2，则用户可用如下的 C 语言程序调用 creat：

```
char * dp;
int fdlib, fmode;
de = "/usr/lib/d2";
fmode = 0775;
fdlib = creat (dp, fmode);
```

或用更简单的方式

```
int fdlib;
fdlib = creat ("/usr/lib/d2", 0775);
```

下面简述这一系统调用的执行过程，这里假定文件是首次创建，即在执行之前，文件还未存在：

① 首先为新文件 d2 分配索引节点和活动索引节点，并把索引节点编号与文件分量名 d2 组成一个新的目录项，记到目录/usr/lib 中。在这一过程中，需要执行以前介绍过的目录检索程序。

② 在文件 d2 所对应的活动索引节点中置初值，包括把存取权限 i_mode 置为 0775，连接计数 i_nlink 置为“1”等等。

③ 为文件分配用户打开文件表项和系统打开文件表项，置系统打开文件表项的初值。包括在 f_flag 中置“写”标志，读写位移 f_offset 清“0”等等。然后，把用户打开文件表项，系统打开文件表项及 d2 所对应的活动索引节点用指针连接起来，最后把用户打开文件表项的序号，即文件描述字返回给调用者。

由于在上述步骤中，也执行了文件“打开”功能，因此在以后操作中，不用再执行“打开”操作。

2) 文件的删除

删除的主要任务是把指定文件从所在的目录文件中除去。如果没有连接的用户，即如果在执行删除之前 i_link 为“1”，还要把这个文件占用的存储空间释放。文件删除系统调用的形式为：unlink (filenamep)；

其中参数与 creat 中的意义相同。在执行删除时，必须要求用户对该文件具有“写”操作权。这一系统财用的处理步骤比较简单，读者可自己设想一下它的执行步骤。

2、文件的连接和解除连接

1) 文件的连接

在文件共享一节中，已介绍文件连接的意义，它的调用方式为：

```
chat * oldnamep, * newnamep;
link (oldnamep, newnamep);
```

其中 oldnamep 和 newnamep 分别为指向已存在文件名字符串和文件别　名 字 符串的指针。这一系统调用的执行步骤如下：

① 检索目录　找到 oldnamep 所指向文件的索引节点编号。

② 再次检索目录　找到 newnamep 所指文件的父目录文件，并把已存文件的索引节点编号与别名构成一个目录项，记入到该目录中去。

③ 把已存文件索引节点的连接计数 i_nlink 加“1”。

从上述过程可知，所谓连接，实际上是共享已存文件的索引节点。

2) 文件的解除连接

其调用形式与文件删除相同：unlink (namep)；

实际上解除连接与文件删除执行的是同一系统调用程序。删除文件是从文件主角度讲的，而解除文件连接是从共享文件的其它用户角度讲的。不论删除还是解除连接，都要除去目录项，把 i_nlink 减“1”，不过，只有当 i_nlink 减为“0”时，才真正删除文件。

3、文件的打开和关闭

文件在使用前，必须先“打开”，以建立用户进程与文件的联系。其主要任务是把文件夹的索引节点复制到活动索引节点表中，以便加速文件夹的访问。

另一方面，活动索引节点表的大小，受到内存容量的限制，这就要求用户一旦当前不再对文件操作时，应及时释放相应的活动索引节点，以便让其它进程使用。这就是“关闭”文件的主要功能。

1）文件的打开

其调用方式为：int fd, mode;

char * filenamep;

fd = open (filenamep, mode);

其中 mode 是打开的方式，它表时打开后的操作要求，如读（0）、写（1）或又读又写（2）。其余参数的意义与 creat 中的相同。open 的执行过程如下：

① 检索目录：一般来说，要求打开的文件应该是已经创建的文件，因此它应该在文件目录中登记，否则就算错。在检索到指定文件之后，就把它的索引节点复制到活动索引节点表中。

② 把参数 mode 提出的打开方式与活动索引节点中在创建文件时记录的文件访问权限相比较，如果非法，则这次打开失败。

③ 当“打开”合法时，为文件分配用户打开文件表项和系统打开文件表项，并为系统打开文件表项设置初值。然后通过指针建立这些表项与活动索引节点之间的联系。在完成上述工作之后，把文件描述字，即用户打开文件表中相应文件表项的序号返回给调用者。

需要指出，如果在执行这一调用之前，别的用户已打开了同一文件，则活动索引节点表中已有了这个文件的索引节点，于是在执行这一调用时，不用执行第（1）步中复制索引节点的工作。而仅把活动索引节点中点的计数器 i_count 加“1”即可。这里 i_count 反映了通过不同的系统打开文件表项来共享同一活动索引节点的进程组数目。它是以后执行文件关闭时，活动索引节点能否释放的依据。例如，图 7-13 的情况，活动索引节点中的 i_count 应为“2”。

2) 文件的关闭

文件使用完毕，就应该执行 close6 系统调用把它关闭，从而切断用户进程与文件之间的联系。其调用方式为：

int fd;

close (fd);

显然，要关闭的文件应该是已经打开的，所以文件描述字 fd 一定存在。close 的执行过程如下：

① 根据 fd 找到用户打开文件表项，继而找到系统打开文件表项。把用户打开文件表项释放。

② 把对应的系统打开文件表项中的 f_count 减“1”，如果不为“0”，说明进程族中还有子程序正在共享这一系统打开文件表项，所以不用释放系统打开文件表项，

而直接返回；否则释放这个系统打开文件表项，并找到与之连接的活动索引节点。

③ 把上述活动索引节点中的 i_count 减“1”，若不为“0”，表明还有其它用户进程正在使用该文件，所以不用释放该活动索引节点而直接返回，否则在把该活动索引节点中的内容复制回文件卷的相应索引节点之后，释放该活动索引节点。

由上述过程可见，f_count 和 i_count 分别反映了进程动态地共享一个文件的两种方式。前者反映了不同进程通过同一个系统打开文件表项共享一个文件的情况；而后者反映了不同的进程或进程族通过同一个活动索引节噗共享一个文件的情况。通过这两种方式，进程之间既可以用相同的位移指针 f_offset，也可以用不同的 f_offset 来动态地共享同一个文件。

4、文件的读和写

文件的读和写是文件夹的最基本操作。“读”是指文件的内容读入到用户进程的变量区中，“写”是指把用户进程变量区中的信息写入到文件存储区中。从文件的什么逻辑位置读入数据，或把数据写入文件的什么逻辑位置均由系统打开文件表中的 f_offset 决定。

1) 读文件

该系统调用的形式为：

```
int nr, fd, count;
char buf [    ]
nr = read (fd, buf, count);
```

这里 fd 是打开系统调用执行之后返回给用户程序的文件描述字；buf 是读出信息应送入的用户内存区首地址；count 是本次要求传送的字节数，nr 是这个系统调用执行之后返回的实际读入字节数。即使在正常情况下，nr 指出的字节数也可能小于 count 要求的字节数。例如，一旦读到文件末尾时，系统调用就返回，而不管是否已读了用户要求的字节数。如果文件的位移时指针已指向文件末尾，又使用了 read 系统调用，则返回“0”值。

假定我们通过打开系统调用打开了文件/usr/lib/d2，与它有关的用户打开文件表项，系统打开文件表项和活动索引节点见图 29 所示的关系。

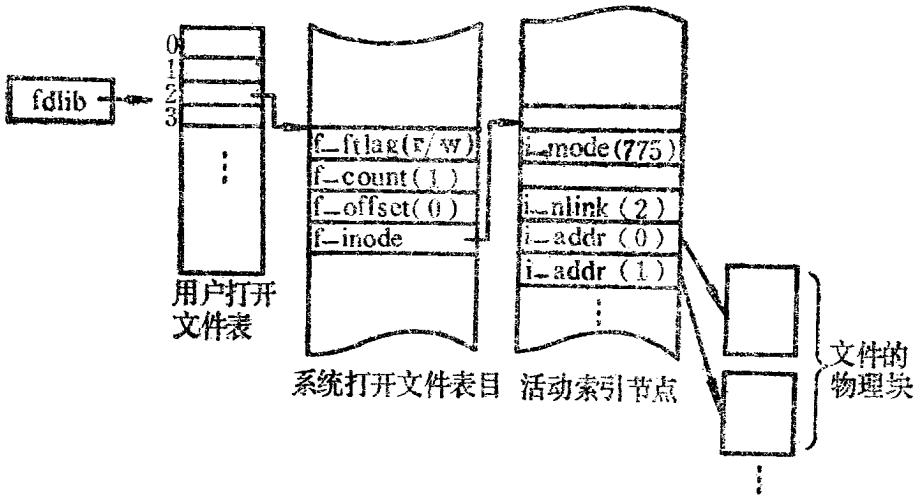


图 29 用户打开文件表项，系统打开文件表项和活动索引节点的关系

现要求读文件 d2 的 1500 个字符到指针 bufp 指向的用户内存区中，number 用来存放实际传送的字节数，则可按如下方式调用 read：

number = read (fdlib, bufp, 1500);

在执行 read 系统调用的过程中，系统首先根据 f_flag 中记录的信息，检查读操作的合法性，如果合法，则根据当前位移量 f_offset 的值，要读出的字节数，以及活动索引节点中 i_addr 指出的文件物理块存放地址，把相应的物理块读到块设备缓冲区中，然后再送到 bufp 指向的用户内存区中。由此可见，在执行 read 的过程中，一定要用到块设备管理中的读程序。

2) 写文件

该系统调用的形式为：

nw = write (fd, buf, count);

其中，fd, count 和 nw 的意义类似于 read，只是 buf 是信息传送的源地址，即把 buf 所指向的用户内存区中的信息，写入到文件存储区中。只要情况正常（中间无差错），nw 一定与 count 相等。

5、文件的随机存取

在文件初次“打开”时，文件的位移量 f_offset 总是清为零。如果不特别指明，以后的文件读写操作总是根据 offset 的当前值，顺序地读写文件。为了支持文件的随机访问，文件系统提供了系统调用 lseek，它允许用户在读、写文件之前，事先改变 f_offset 的指向。这一系统调用的形式为：

long lseek;

long offset;

int whence, fd;

lseek (fd, offset, whence);

其中，文件描述字 fd 必须指向一个用读或写方式打开的文件，当 whence 是“0”时，则 f_offset 被置为 offset，当 whence 是“1”时，则 f_offset 被置为文件当前位置加上 offset。

13.5 UNIX 操作系统的设备管理

在 UNIX 系统中，外部设备按其信息组织和传送单位的不同而被分成两类，一是存储设备，主要代表为磁盘。由于在这类设备上存储的信息，在物理往往是按盘块为单位组织和传输的，所以称作为块设备；一是输入/输出设备，主要代表为终端键盘和行式打印机。由于在这类设备上的信息往往是以字符为单位组织和传输的，因此称作为字符设备。

为了能提高 CPU 和块设备工作的并行度，为了能解决 CPU 和字符设备之间的速度不匹配问题，UNIX 在设备管理时，采用了完善的缓冲技术。

本节主要讲述 UNIX 采用的各种缓冲技术以及对缓冲的管理。至于 I/O 管理，由于涉及具体设备，琐碎且繁杂，加之篇幅所限，就省略不介绍了。

UNIX 系统的设备管理具有下述特点：

1、文件和外部设备的一致性。所有设备都作为文件来处理——一种特殊文件。它们驻留在目录/dev 下，每一类设备都有一个文件名，对每一类设备除了具体的输入/输出处理外，其余处理都和普通文件一样。特定设备的特性都放到了各自的设备驱动程序里，用户无须知道有关的细节。因此使用 I/O 设备如同读写一个普通文件，可同样使用读写系统调用：

read (fd, buffer, count)

write (fd, buffer, count)

2、从设备管理的层次结构上，块设备管理和字符设备管理是相似的。尽管这两类设备在功能上和物理上都有很大的差别，但在管理控制方法、数据结构和层次结构上都相似。管理上可以分三个层次：(1) 低层都设置了一个公用的缓冲池及相应的管理；(2) 中间层是设备驱动程序，都设置了设备表和设备开关表；(3) 高层是设备管理与文件系统的接口，包含有读、写、中断处理过程等。

3、读写文件除了一般的读写方式外，还提供了“提前读”、“异步写”和“延迟写”方式。从盘中读文件时，通常都是顺序地读出一系列逻辑块中的信息，但为了加快信息读出，可采用“提前读”方式。这时，一方面把指定的“当前块”中的信息读入内存，同时把“当前块”的下一块的信息提前读出，为下次读提前作好准备，从而提高了读出速度。在写的过程中，写过程须等待传送完成才能返回，而在“异步写”时，进程不必等待传送完成即可返回，这样提高了运行进程和磁盘 I/O 传送的并行程度。采用“延迟写”方式是为了尽量减少不必要的 I/O 操作，节省磁盘空间。因为当前要写的信息，可能以后还要用，那么就让它驻留在缓冲区内，延迟一段时间再写入磁盘，如果此时有进程要读该信息，就不必从盘中去读，而是直接从缓冲区中去读，这样就减少了对该文件的磁盘 I/O 操作。

4、尽可能用公用代码处理设备的相似部分。例如磁盘可以用作一般的文件存贮器，又可作为进程对换的外存区（swap 区），还可以作为非块设备的字符设备。在使用中，把内存地址、盘块号、传送字节数、读写特性等各自事先处理好，然后再去调用公共处理程序。又如，打印机、终端等字符设备有公共的处理字符的 I/O 加工程序，只要花很少代码去处理它们之间的差别就行。这样不仅使整个设备处理代码大为减少，也为以后增加设备打下良好的基础，因为只需增加与增加设备特别有关的部分代码，其它则可利用已有的公用代码。

13.5.1 块设备管理的数据结构

1、缓冲存储区与缓存控制块

1) 缓冲存储区

为了便于内存和块设备之间信息的传输，在内存设置了十五个缓冲存储区（以下简称缓存），每个缓存为 514 个字节大小；其中 512 个字节恰能放入一个盘块的有效信息，另外两个字节供它用。这十五个缓存构成了一个缓冲池。

缓存主要用于文件信息的读、写。在磁盘和处理机之间设置这个缓冲池，就能发挥它们之间的并行工作能力，协调速度不匹配的矛盾。

2) 缓存控制块 buf

缓存只是用来存放有效信息的，为了对缓存的使用情况加以管理，为了能记录具体的输入/输出要求，系统相应地设置了十五个缓存控制块 buf。每一个 buf 对应于一个缓存，它们在物理上并不相邻，而是在 buf 里有指针记录下对应缓存的起始地址。此外在 buf 中还有关于相应缓存的使用情况以及队列指针等信息。因此缓存控制块 buf 和它所对应的缓冲存储区之间的关系，恰如进程控制块 PCB 与进程之间的关系似地，系统运行过程中感知的是 buf。正是 buf 在各种队列中排队，得到一个 buf，就会获得它所对应的缓存及其使用情况。

每一个 buf 结构有十二项内容，buf 是一个结构类型的数据。其形式如下：

struct buf

```
{
   int b_flags;      各种标记
   struct buf * b_forw;设备缓冲区（b）链向前指针
   struct buf * b_back;      设备缓冲区（b）链向后指针
   struct buf * av_forw;     空闲缓冲区（av）链向前指针
   struct buf * av_back;     空闲缓冲区（av）链向后指针
   int b_dev; 与缓冲区相连的设备号
   int b_wcount;  传送字数
   char * b_addr;  缓冲区内存地址低 16 位
   char * b_xmen;缓冲区内存地址高 6 位
   char * b_blkno;      块设备上的物理块号
   char b_error;
   char * b_resid;
} buf [NBUF];
/*flags*/各种标记如下：
# define B_WRITE 0  写标记
# define B_READ 01  读标记
# define B_DONE 02  传送结束
# define B_ERROR 04      传送有错
# define B_BUSY 010缓冲区正被使用不在 av 链上
# define B_PHYS 020
# define B_MAP 040
# define B_WANTED 0100      有进程正在等待使用该缓冲区
# define B_RELOC 0200
# define B_ASYNC 0400  异步 I/O 操作，无需等待
# define B_DELWRI 01000      延迟写
```

• b-addr 和 b-xmen

这是存放对应信息内存起址的地方，b-addr 为低 16 位，b-xmen 为高 16 位。

• b-dev

这是存储设备的设备号。设备号由主设备号 d-major 和次设备号 d-minor 组成。主设备号是设备的类型号，位于 b-dev 的高节字处，次设备号是设备在同类设备中的编号，位于 b-dev 的低字节处。

• b-blkno

这是存放块设备上物理块序号的地方。

由 buf 中的(1)~(3)信息就建立起了一个缓存和某块设备上的一个盘块之间的对应关系。

• b-wcount

这是输入/输出请求传输的字数，如果是传送一盘块，则是 256。

• b-error

这是输入/输出出错时返回的出错信息。

• b-resid

这是由于输入/输出出错而未能传输的剩余字数。

• b-flags

这是标志字。一是用来标明相应缓存的使用情况，二是反映输入/输出方式，主要有：

B-WRITE 0 写（将缓存中的信息写到盘块上去）
B-READ 01 读（将盘块中的信息送到缓存中）
B-DONE 02 I/O 操作结束
B-ERROR 04 I/O 因出错而中止
B-BUSY 010 相应缓存正在使用
B-WANTED 0100 有进程正在等待使用该 buf 管理的缓存，请 B-BUS 时，唤醒此进程
B-ASYNC 0400 异步 I/O，不需等待其结束
B-DELWR 11000 延迟写，只有在相应缓存要移作它用时，才将其内容写到盘块上

• b-forw 和 b-back

当 buf 位于某个块设备的设备 buf 队列时，b-forw 为队列的前向指针，b-back 为队列的后向指针。

• av-forw 和 av-back

当 buf 空闲时，它将排在自由 buf 队列，av-forw 为队列的前向指针，av-back 为队列的后向指针。由于 buf 不仅反映了缓存使用的有关信息，也记录了具体的输入/输出要求和执行结果（和上面的 b-wcount, b-error, b-reisid, b-flags）。因此对某一块设备的 I/O 请求，也可以通过它体现出来，即用 buf 构成对一个设备的输入/输出请求队列。由于一个 buf 排在设备输入/输出请求队列时，绝对不可能排在自由 buf 队列里，所以 buf 中的 av-form 和 av-back 也借用来形成设备输入/输出请求队列的前向、后向指针。

2、块设备表devtab

UNIX 为各类设备（也就是 b-major 不同的设备）设置了块设备表，以便对该类设备的输入/输出请求队列以及与该类设备相关的设备 buf 队列进行管理。

每一类块设备有一张块设备表，每个 devtab 结构有六项内容，其结构为：

```
struct devtab
{
char d_active;      相应设备已被启动进行 I/O 操作标记
char d_errcnt;      I/O 操作出错记数，对盘而言，错误次数小于 10
    次，重执行 I/O 操作
struct buf * b_forw;    该设备缓冲器队列的头指针
struct buf * b_back;    该设备缓冲器队列的尾指针
struct buf * d_actf;    该设备 I/O 缓冲器队列的头指针
struct buf * d_actl;    该设备 I/O 缓冲器队列的尾指针
}
```

列置如下：

(1) d-active

该类设备的忙/闲标志。

(2) d-errcnt

出错重试次数。对磁盘而言，若出错次数≤10，则重复执行该 I/O 请求。

(3) b-forw 和 b-back

该类设备的 buf 队列的前向指针和后向指针。

(4) d-actf 和 d-actl

该类设备的输入/输出请求队更的首指针和尾指针。

3、块设备开关表

为了组织设备的输入、输出，各类块设备都有各自的一套子程序，例如打开、关闭、启动等子程序。为了能方便地寻找到这些子程序的入口，系统为每类设备设置了一数据结构，用于存放这些子程序入口，这个数据结构被称为设备开关，起名 bdevsw。每类块设备的设备开关集中在一起，就称作块设备开关表。块设备开关表也称为处理程序入口表，其结构如下：

```
struct bdevsw {
        int ( * d_open) (    );     打开子程序入口地址
        int ( * d_close) (    );    关闭子程序入口地址
        int ( * d_strategy) (    ); 启动子程序入口地址
        int * d_tab;     对应设备表地址
} bdesw [    ]
```

bdevsw 结构共有四项内容，列置如下：

(1) d-open 打开子程序的入口地址；

(2) d-close 关闭子程序的入口地址；

(3) d-strategy 启动子程序的入口地址；

(4) d-tab 该块设备的设备表地址。

譬如，由于磁盘没有“打开”和“关闭”子程序，故在相应位置处填入&nulldev，在启动程序位置填上&rkstrategy，在设备控制表处填上&rktab。

在系统生成时，根据设备配置的情况形成如下开关表：

```
int ( * bdevsw [    ]) (    )
{
& nulldev, & nulldev, & rkstrategy, & rktab, /*rk*/
& nodev, & nodev, & nodve, 0, /*rr*/
& nodev, & nodev, & nodve, 0, /*rf*/
& nodev, & nodev, & nodve, 0, /*tm*/
& nodev, & nodev, & nodve, 0, /*tc*/
& nodev, & nodev, & nodve, 0, /*hs*/
& nodev, & nodev, & nodve, 0, /*hp*/
& nodev, & nodev, & nodve, 0, /*ht*/
0
}
```

这里，&表示指向变量或函数（子程序）的地址。

& nulldev 表示无此种设备（空操作）。

& nodev 表示无此类设备。

一般 UNIX 系统中只配备了一台块设备即磁盘 rk，其打开子程序和关闭子程序是空操作，它的启动子程序是 rkstrategy，对应设备表为 rktab。如果用户需增加块设备，则根据所增加设备的特性填上相应各项。

13.5.2 缓存控制块 buf 的各种队列

由于 buf 记录了与缓存有关的各种管理信息（包括具体的 I/O 要求），所以系统总是通过管理 buf 来使用缓存的。

1、自由buf队列和自由buf队列控制块bfreelist

一个可被分配它用的 buf，一定位于自由 buf 队列，这意味着该 buf 所对应的缓存现在是自由的，此种 buf 标志字 b-flags 中的 B-BUSY 标志均被清除。

每个处于自由 buf 队列中的 buf 结构相互使用 av-forw 和 av-back 形成双向勾链。

为了能真正管理起自由 buf 队列，显然需要对队列的首、尾加以管理，这将由自由 buf 队列控制块 bfreelist 来完成。为了整齐化一，系统让它也具有和一般 buf 相同的结构，只是用它里面的 av-forw 做自由 buf 队列的首指针，而 av-back 做自由 buf 队列的尾指针。于是，自由 buf 队列可如图 13-30 所示。

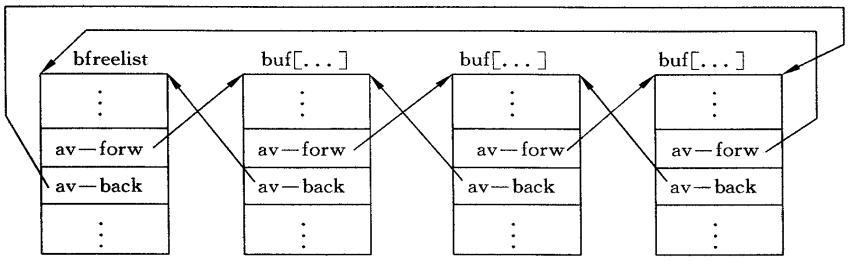


图 13-30　自由 buf 队列

对自由 buf 队列，系统采用 FIFO 管理算法，一个缓存被释放时，相应于它的 buf 就被送入自由 buf 队列之尾；当请求一个缓存时，就从自由 buf 队列之首摘下一个 buf，它所对应的缓存即被分配出去。

2、输入/输出请求队列

通过缓存进行输入/输出时，具体的 I/O 请求全都包含在 buf 结构中，它们是：操作类型（读或写）、信息源和目的地的地址、以及数据的传输量等。因此，buf 既是缓存的控制块，又是 I/O 的请求块。也就是说，向主设备号相同的各设备提出的 I/O 请求所构成的输入/输出队列也由 buf 组成。由于位于 I/O 请求队列的 buf 肯定不会出现在自由 buf 队列中，故在 I/O 请求队列里的 buf 就借用 av-forw 来构成一个单向链，该队列的队首、队尾指针是该类设备的块设备表中的 d-actf 和 d-actl。于是某块设备的输入/输出请求队列如图 13-31 所示。

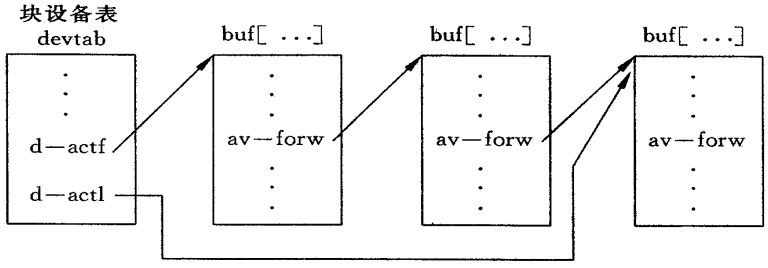


图 13-31　块设备的 I/O 请求队列

对于块设备的 I/O 请求队列，系统仍采用 FIFO 管理算法。要注意的是，系统中也有不通过缓存进行的输入和输出。回忆一下 UNIX 存储管理，在那里曾提及在磁盘上专门开辟了进程映像对换区，用以存放调出的进程映像非常内驻内存部分。这部分存

储资源归 UNIX 存储管理部分进行分配和释放，但它提出的输入/输入却要由设备管理部分来负责完成。由于它的 I/O 请求是指把对换区盘块中的内容送到内存的用户区指定处，与缓存没有关系，也以为了能提交这种不通过缓存进行的 I/O 请求，系统专门开设了一个名为 swbuf 的 I/O 请求块。swbuf 的结构与 buf 结构一样，它也排列在块设备的 I/O 请求队列中，使用 av-forw 与它的前、后 buf 勾链。不同的是它里面的 b-addr 和 b-xmen 给出的不是哪个缓存的内存起址，而是分配给它的内存区起址。顺便再强调一下，swbmf 和 bfreelist 是使用 buf 结构的二种特殊情况，它们和缓存没有关系。

3、设备buf队列

整个系统的缓存资源是有限的，为了实现对它的充分共享，UNIX 做了如下考虑：

第一，一旦某缓存被释放，相应的 buf 就应立即进入自由 buf 队列，以作它用。

第二，一个已经位于自由队列的 buf，在尚未移作它用之前，它里面的信息仍保持着缓存与某设备上盘块的对应关系。若此时系统根据需要能够退回来按原状继续使用它，那么就可省去重复的、且十分耗费时间的设备 I/O 操作过程，从而提高文件系统的工作效率。为了能方便地做这种“进”（移作它用）或“退”（继续按原状使用），UNIX 在自由 buf 队列和 I/O 请求队列之外，又设置了一种新队列：设备 buf 队列。

每类块设备都有一个设备 buf 队列，队首和队尾分别是相应块设备表 devtab 中的 b-forw 和 b-back。队列中的 buf 结构用其指针 b-forw 和 b-back 进行双向勾链。一个缓存被分配用来对某类块设备的某一盘块进行读、写时，与之对应的 buf 就进入了该类块设备的设备 buf 队列。即使在完成了 I/O 请求而被释放后(这时进入了自由 buf 队列)，它仍然呆在这个设备 buf 队列里，除非再次分配用于别类块设备时止。

每当一个 buf 进入设备 buf 队列时，总是在该队列之首进行插入。

这里要注意的一个问题是，在系统初启时，每一个 buf 都没有分给任何实际的块设备，为此，系统增设了一个想象的设备 NODEV，其首、尾指针就用 bfreelist 中的 b-forw 和 b-back。于是 NODEV 队列和某类设备的设备 buf 队列如图 13-32 的所示。

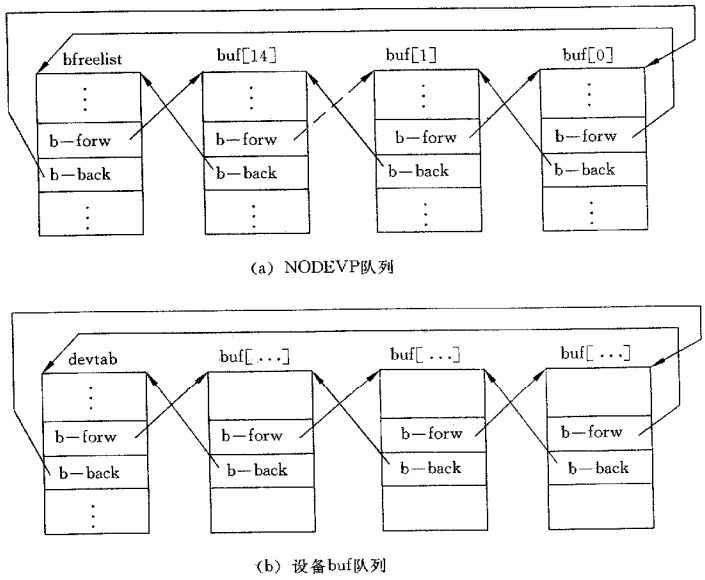


图 13-32　设备 buf 队列的初始态和演变

由于一个 buf 建立起某类设备的一个盘块与一个缓存之间的联系之后，它就一直呆在该类设备的设备 buf 队列里，直到这个缓存重分它用，因此对于一个 buf 来讲，任何时候都处在上述三个队列的两个之中：在自由 buf 队列和设备 buf 队列(包括 NODEV

队列的情形），或在 I/O 请求队列和设备 buf 队列。

13.5.3 字符设备管理的数据结构

1、字符缓存和自由字符缓存队列

为字符设备设置字符缓冲的主要出发点是解决它们和 CPU 之间的速度不匹配问题。由于字符设备的共同特点是：工作速度慢，一次输入/输出请求的字符数量少且不固定，因此系统为字符设备设置了一个使用比较灵活、占用内存较小的缓冲机构。

系统在内存总共设置了 100 个字符缓存 cblock，它们形成一个能够共享的缓总池 cfree。每个字符缓存为 8 个字节，前两个字节为字符缓存指针 c-next，后六个字节为该缓存的信息区 info。

未被使用的字符缓存通过各自的 c-next 勾链成一个单向的自由字符缓存队列，其队首由一个名为 cfreelist 的指针指出，处于队列之尾的字符缓存的 c-next 取值 null，图 13-33 给出了自由字符缓存队列的示意。

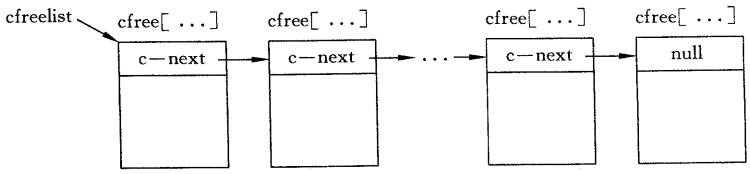


图 13-33　自由字符缓存队列

系统视自由字符缓存队列为一栈，分配和释放均在队首进行。这里要注意的是，由于对字符缓存的管理比较简单，因而没有必要再设置专门的缓存控制块。

2、字符设备表和I/O字符缓存队列

和块设备类似，每个字符设备也都有一张设备表。但由于各类字符设备的特性判别很大，不可能设置统一形式的设备表。譬如说在 UNIX 系统里，纸带机的设备表仅三项内容，而终端机的设备表则含有十三项内容。

但它们共同之处是都包含有输入/输出字符缓存队列的控制块，该控制块均为 clist 结构，里面含三项内容。

(1) c-cc 队列中可用字符计数；

(2) c-cf 指向队列中的第一个字符；

(3) c-cl 指向队列的最后一个字符。

于是，一个字符设备的输入/输出字符缓存队列的一般形式如图 13-34 所示。

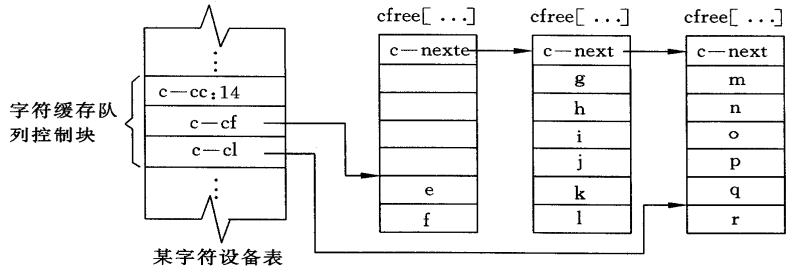


图 13-34　I/O 字符缓存队列

由此可知，在整个系统中存在着多个 I/O 字符缓存队列。如果某一字符设备既有输入功能，又有输出功能，则它的设备表中就会有两个字符缓存队列控制块，一个控制输入字符缓存队列，一个控制输入字符缓存队列。

3、字符设备开关表

每个字符设备也有自己的设备开关，这些开关集合在一起，就形成字符设备开关表 cdevsw。开关表中的每一表目与一个字符设备相对应，是该设备的开关。它包含五项内容，前四项分别是打开、关闭、读和写子程序的入口地址，第五项是专门为终端设置的，为设置并获得终端机特性子程序的入口地址，其余字符设备均没有此项。

13.5.4 字符缓存的管理

为了从字符缓存中取字符或往字符缓存中送字符时能够较为容易地判断出是否要释放字符缓存或申请字符缓存，UNIX 系统在对 cfreelist 进行初始化时，采取了一定的措施，使每一个字符缓存的起始地址的最后三位为 0。这实际上只需要确保第一个字符缓存 cfree[0]的起始地址最后三位为 0 即可，因为每个字符缓存都由 8 个字节构成，这样排列下来，每个字符缓存起始地址的最后三位就都为 0 了（这也是每个字符缓存长度取为 8 个字节的原因之一）。

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问题。

以图 34 为基础，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，这时图 13-34 就成为图 13-35 所示。

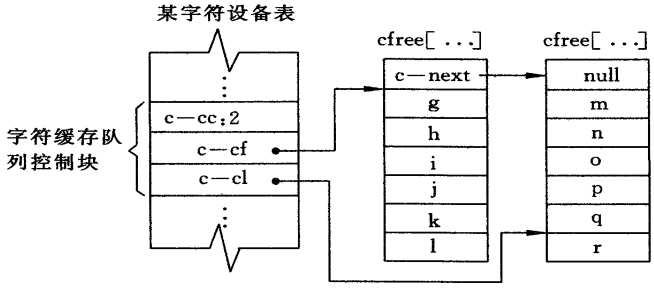


图 13-35　取出 e、f 字符后的 I/O 字符缓存队列

若在图 13-35 的基础上，根据 c-cl 指点往字符缓存里送字符。显然，当 c-cl 加 2 而获取下一存放字符的位置时，会发现该地址的最后三位为 0，这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。于是去 cfreelist 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。此时图 35 就成为图 13-36 所示。

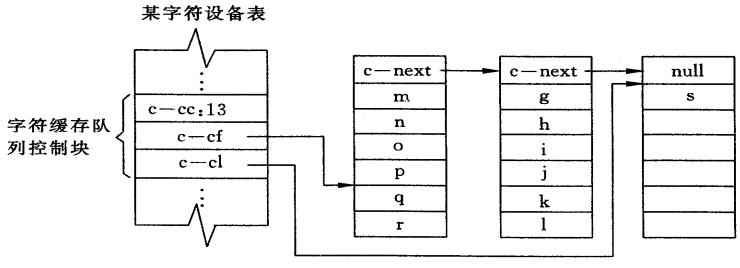


图 13-36　送入 S 后的 I/O 字符缓存队列

13.5.5 块设备的读写操作

通过缓冲技术，块设备的读写有三种方式、五个过程：

1、一般读、写方式，有二个过程：

bread (dev, blkno)——一般读过程，把物理块中的信息读入缓冲区

bwrite (bp)——一般写过程、把 bp 指定的缓冲区中的信息写到磁盘上，要等待写完成。

图 13-37、13-38 给出了一般读写方式的流程图。

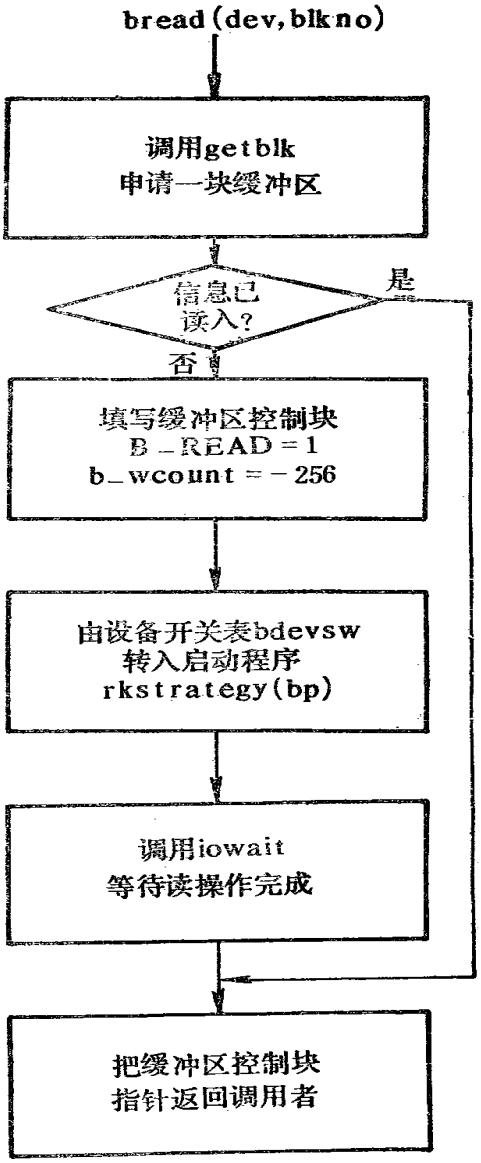


图 11-37　bread (dev, blkno) 流程

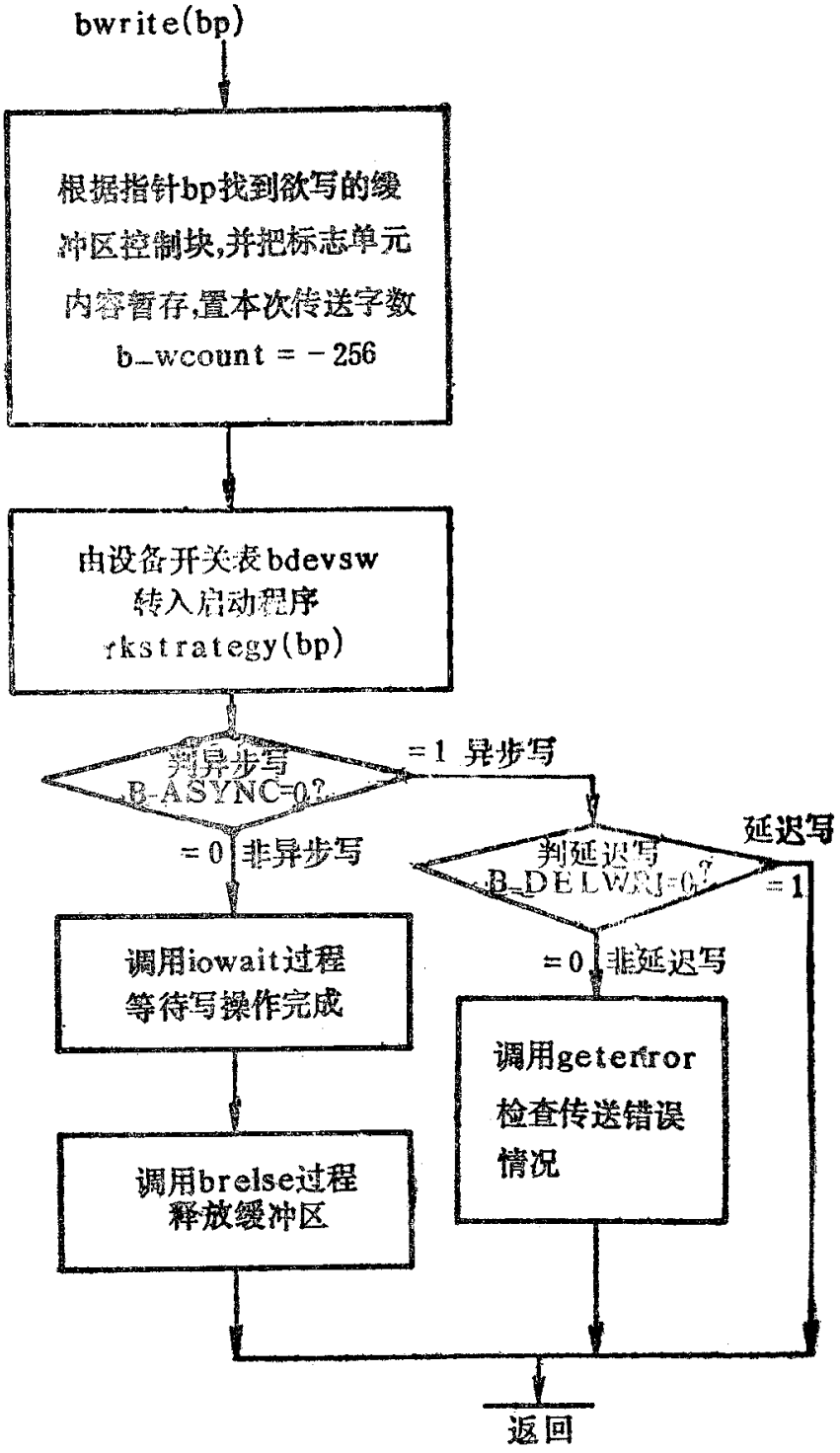


图 13-38　bwrite (bp) 流程图

下面介绍在上述流程中所调用到的几个过程：

1）rkstrategy (bp)

这一过程的功能是把 bp 所指定的缓冲区排列在磁盘设备缓冲区 I/O 队列末尾。若设备不忙，则启动磁盘，传送该队列中第一个缓冲区的内容。

过程的流程图如图 13-39 所示。

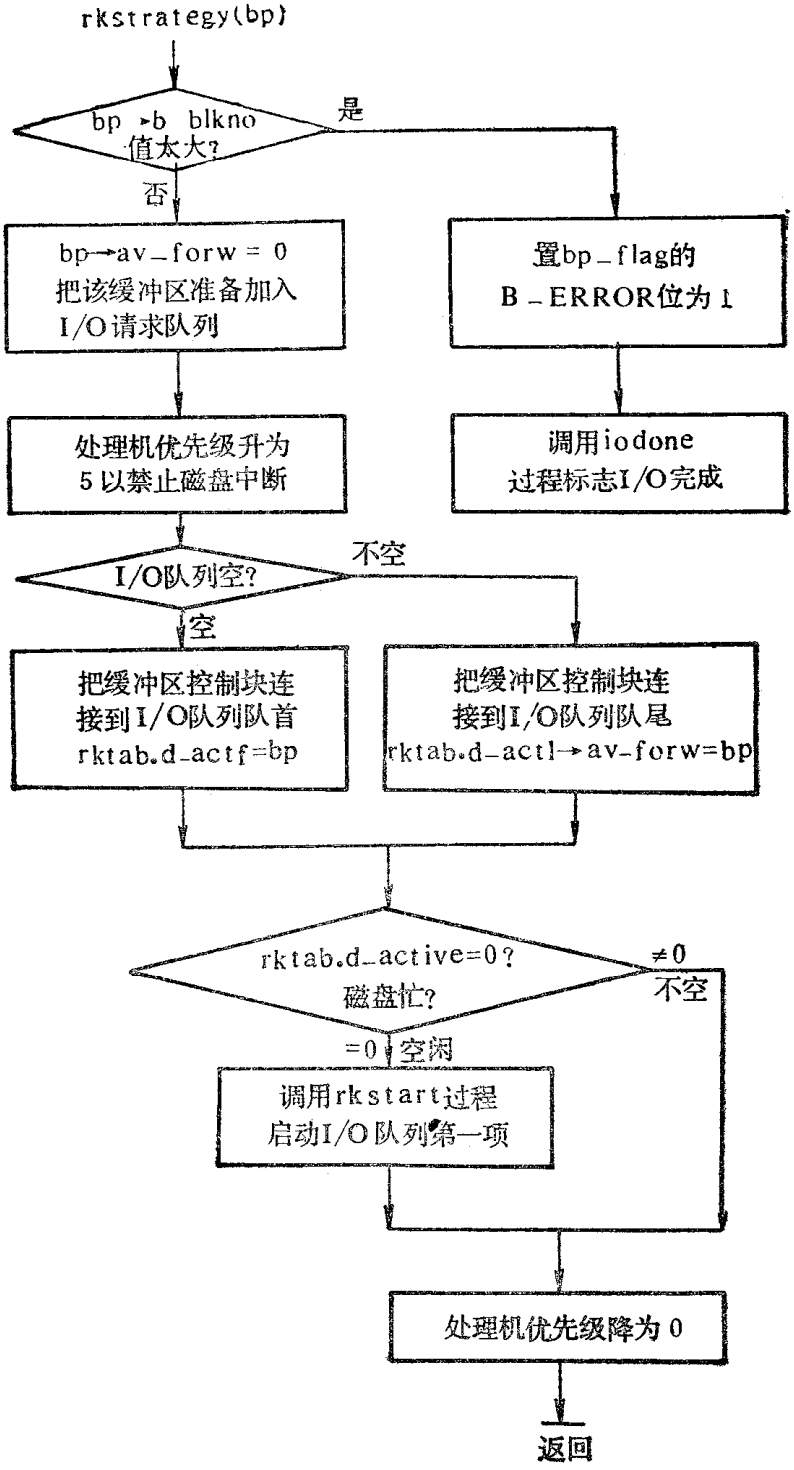


图 13-39　rkstrategy (bp) 流程图

2）rkstart (　)

它由 rkstrategy (bp)调用，共主要功能是从磁盘设备缓冲区 I/O 队列中取出第一个缓冲区控制块，调用 devstart (bp, devloc,devblk, hbcom)过程，启动磁盘进行 I/O 操作。主要流程图 13-40 所示。

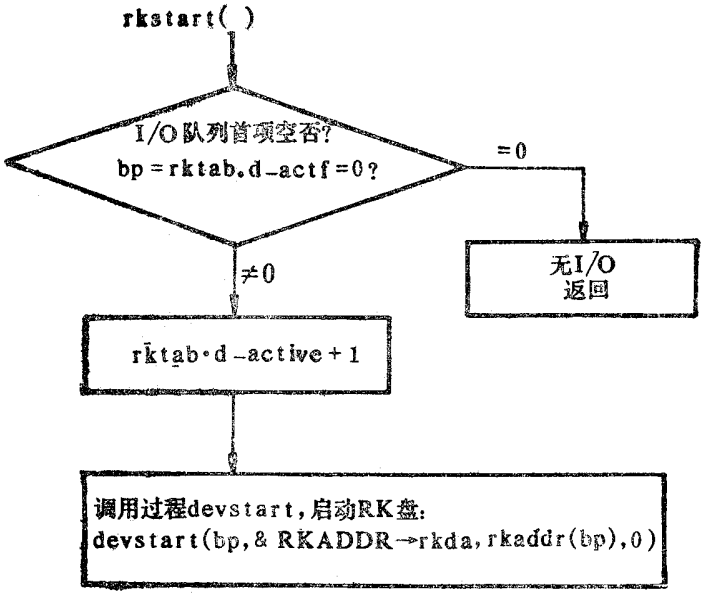


图 13-40　rkstart (　) 流程图

3）devstart (bp, devloc, devblk, hbcom)

此过程由 rkstart (　)调用。功能是装配和填写各个磁盘寄存器，启动磁盘。其中的四个参数的意义为：

bp——要传送信息的缓冲区首部地址，缓冲区首部中包含着本次传送的所有控制信息。

devloc——给出要启动的设备的寄存器组织的起始地址。

devblk——指定的外设所要求的块设备地址格式。

hbcom——命令码高位。

下面我们以 RK 磁盘为例加以说明，但这个过程不仅仅适用于 RK 磁盘。

RK 磁盘控制器的寄存器组由六个寄存器组成：

RKDS　存放驱动器状态的寄存器

RKER　存放传送出错信息的寄存器

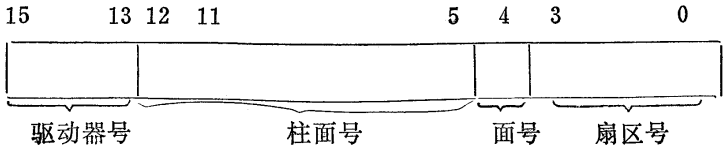
RKCS　存放控制状态信息的寄存器

RKWC　存放传送字数的补码的寄存器

RKBA　存放内存数据地址的寄存器

RKDA　存放磁盘数据地址的寄存器

在每次磁盘传送之前，驱动程序必须按下面的规定填写：



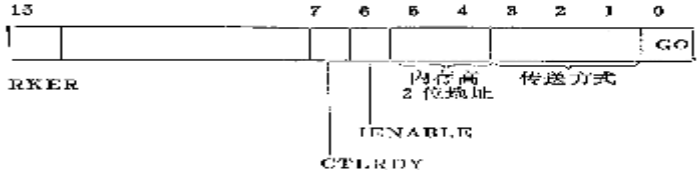
RDKA 为：

RKBA 要填入传送数据在内存的地址的低 16 位。

RKWC 传送字数的补码，如同缓冲区首部中的 b_wcount。

RKCS 是控制状态寄存器，其上信息一旦填入就立即启动磁盘。

第 0 位（GO）：此位置 1 则启动磁盘实现第 1~3 位指出的传送。规定如下：



第 321 位　　传送方式

000　　控制复位 RESET

001　　从内存写到磁盘上

010　　从磁盘读到内存中

第 5-4 位：内存传送地址共占用 18 位，RKBA 中存放低 16 位，高 2 位存放于此。

第 6 位（IENABLE）：此位为 1 表示开放中断。

第 7 位（CTLRDY）由硬件设置，为 1 表示设备已准备好接收下一传输命令。

第 15 位（RKER）由硬件设置，表示传送结果，如为 1，则传送出错。

磁盘传送完成之后，其中包含有传送错误与否的信息，由 RKDS 和 RKER 中的信息表明。

图 13-41 给出了 devstart (　)的流程图。

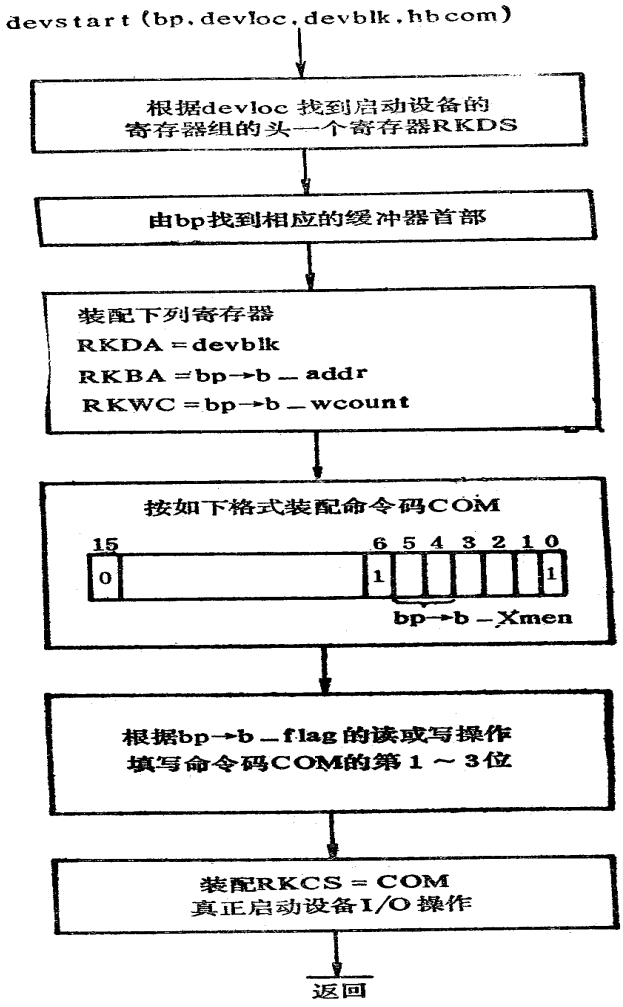


图 13-41 给出了 devstart (　)的流程图。

2、“提前读”和“延迟写”方式，有二个过程：

breada (adev, blkno, rablkno)——提前读，先把指定物理块 blkno 中的信息读入缓冲区，接着再把“提前”要读的块 rablkno 的信息读出。

bdwrite (bp)——延迟写，系统只把信息写到指定缓冲区，并把该缓冲区写标记 B-DELWRI 置 1，但并不启动磁盘把它写到盘上，待以后有进程申请缓冲区，且申请到该缓冲区并判该缓冲区是“延迟写”，则调用 bwrite 过程，以异步写方式将该缓冲区信息写到磁盘上。

设置延迟写是为了减少不必要的 I/O 重复和节省磁盘空间。有些情况下，这些输出的信息可能不久又要将它读入内存。在延迟写方式下，只要将其它进程未申请到该缓冲区，其上的信息便暂时不会写入磁盘，以后需要再使用这些信息时，就可直接从该缓冲区读入，而不必从盘上去读，从而避免了把信息输出后又马上要读入的 I/O 反复操作。另外，在信息较短的情况下，为了节省磁盘空间，一个磁盘块上可装几个这种短信息，因此相应的缓冲区在未写满信息时，并不把它立刻写入磁盘块中，而只是将该缓冲区又挂到 av 链上，当下次又有短信息要写入盘时，则从 av 链上摘下此缓冲区，写入新的短信息，然后又挂到 av 链上，如此反复，直到该缓冲区写满或有其他进程申请此缓冲区时，才启动磁盘写入。这样，既避免了过于频繁的 I/O 操作，也节省了磁盘空间。

异步写 bawrite (bp) ：异步写过程与一般写过程基本相同，但它不必等待传送完成即可返回。bawrite (bp)调用者进程先把异步写标志位 B_ASYNC 置 1，然后调用 bwrite (bp)并立即返回。这样，进程一方面在进行其他操作，同时在作写操作，因而提高了进程运行和 I/O 操作的并行度。

13.5.6 磁盘中断处理

当磁盘 I/O 传送结束，产生 I/O 结束中断，CPU 响应中断，进入中断处理。它首先检查相应设备表中的 d_active 项是否为 1，即磁盘是否真正被启动，若未被启动，则不作中断处理而立即返回。若是在启动状态中，取出其缓冲区 I/O 队列的第一缓冲控制块，置 d_active = 0，表明本次传送结束。然后根据状态寄存器 RKCS 的第 15 位是否为“1”，判传送出错否。若出错，则打印出错信息（由 RDER 和 RKDS 的内容决定）。由于磁盘设备传送出错几率很高，因此 UNIX 中并不是一出错便立即停止传送，而是重复执行 10 次传送（最多 10 次），若仍出错，那么才置本次缓冲区传送错误标志 B_ERROR，放弃本缓冲区的 I/O 操作，转向缓冲区 I/O 队列的下一个缓冲区信息的传送。中断处理 rkintr ( )的流程图如图 13-42 所示。

上面我们已比较详细地介绍了块设备的管理，目的是希望对块设备管理了解得更具体一些，以供用户自行增加某种设备时参考，为了让读者对块设备的读写过程有一完整的概念，我们给出读过程的源程序及其流程图。

```
bread (dev, blkno)
{
        register struct buf * rbp
        rbp = getblk (dev, blkno);
        if (rbp->b_flags &    B_DONE)
              return (rbp);
        rbp->b_flags = | B_READ;
```

```
        rbp->b_wcount = -256;
        (*bdevsw [dev.d-major]. d_strategy) (rbp);
        iowait (rbp);
        return (rbp);
}
```

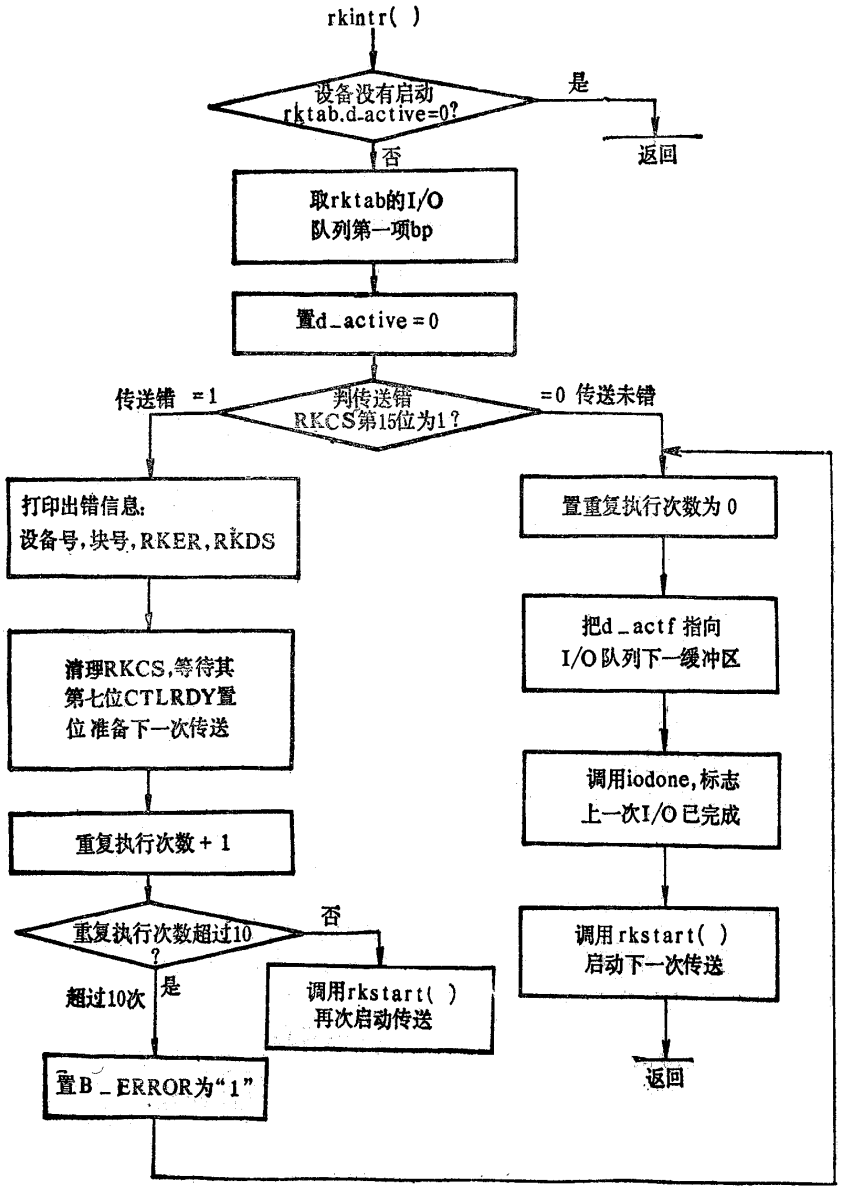


图 13-42　rkintr (　) 的流程图

它的功能是从块设备 dev 上将一指定的字符块 blkno 上的内容读到缓冲存贮器。由源程序可知，这一读过程由下列步骤构成：

1、根据 dev, blkno 在 dev 设备的缓冲区队列中找到指定的缓冲区，由于其内容就是 bread (　)所需的，不必再进行读操作，而直接返回缓冲区的指针。

2、在 dev 的缓冲区队列中找不到相应的缓冲区，则要从空闲缓冲区队列中分配一个缓冲区。

3、把所得的缓冲区挂到设备 dev 的 I/O 缓冲区队列，并进行读操作。如 I/O 缓冲区队列中无其他的 I/O 操作，则立即读取该块。否则，先读其他的请求读取的信息块。每读一块，将由中断处理程序来启动执行下一块的读取。图 13-43 给出了它的流程图。

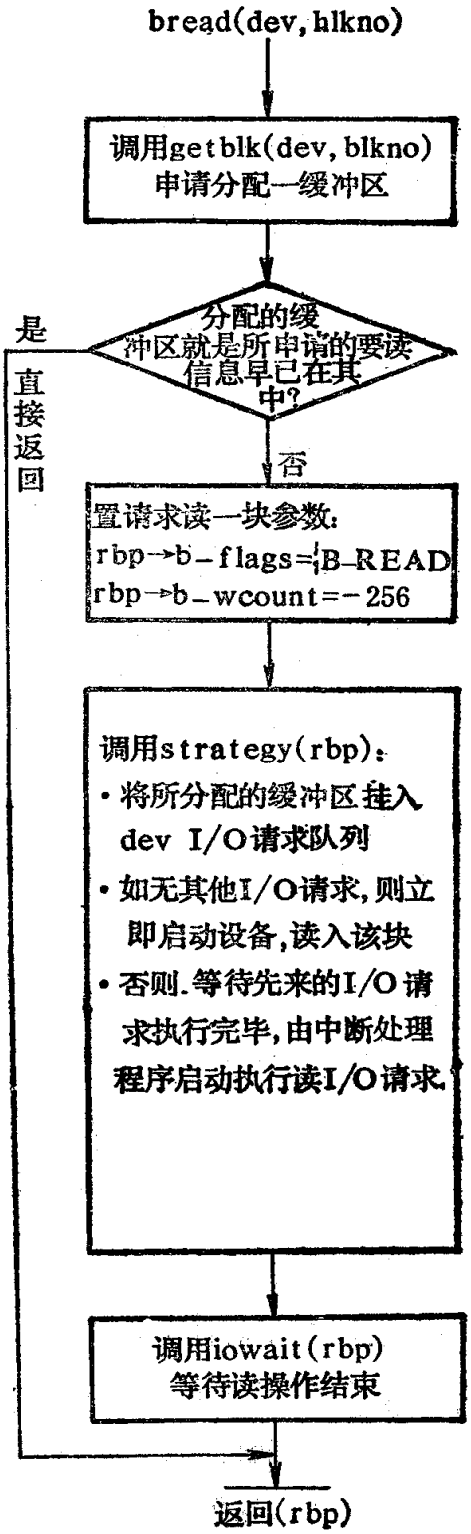


图 13-43　bread (dev, dlkno) 流程图

13.5.7 块设备 I/O 操作与文件读写关系

下面进一步说明块设备读操作 bread (dev, blkno)与文件系统中的读文件系统调用 read (fd, base, count)的关系。

读文件 read (fd, base, count)的过程如下（参见图 13-44）：

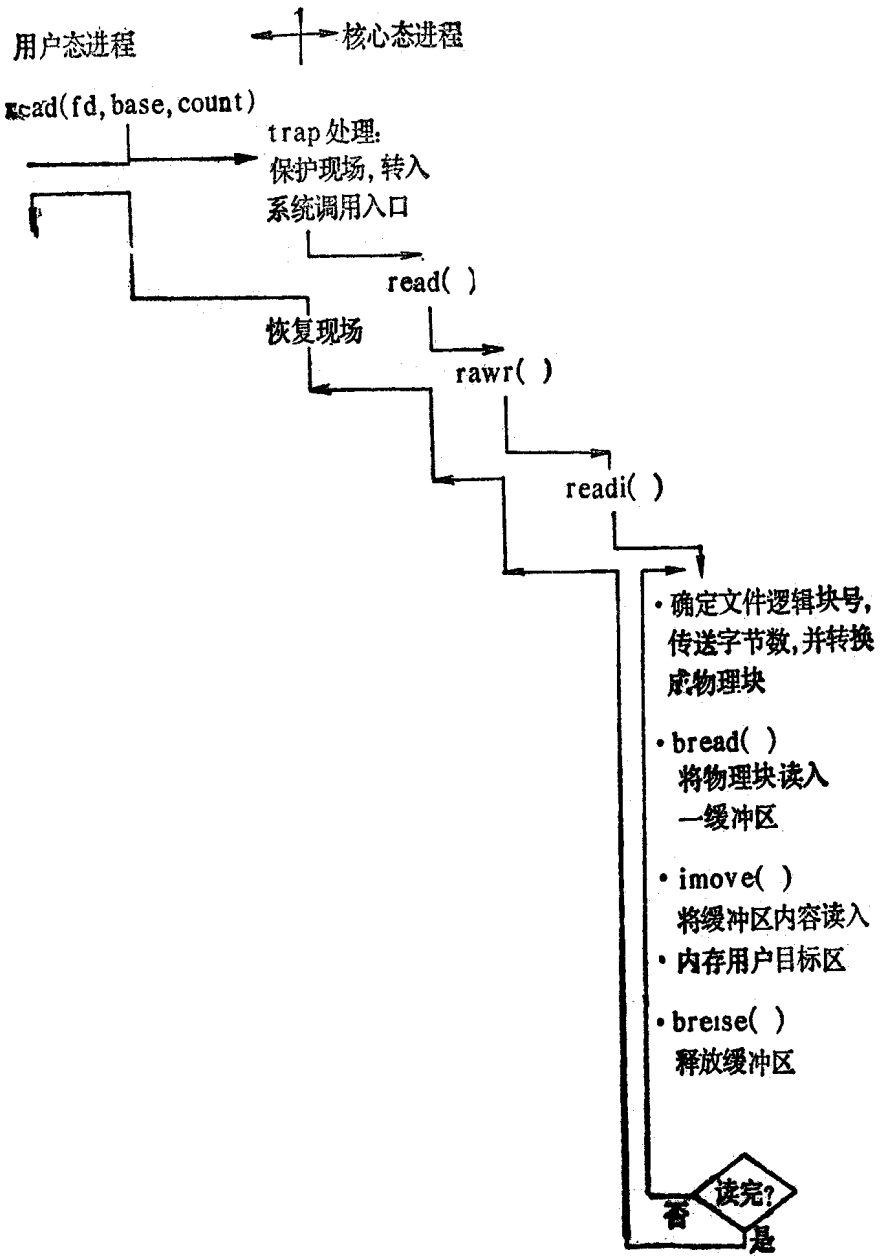


图 13-44　read (fd, base, count) 工作流程

1、用户程序请求操作系统为其服务，读取一文件。通过 trap 处理进入读文件系统调用入口 read (　)。这时进程由用户态进入核心态。

2、read (　)调用 rdwr (FREAD)，而由 rdwr (FREAD)执行；

根据文件描述符 fd，通过用户打开文件表项确定系统打开文件表项及内存活动 i 节点，并确认读或写操作的合法性，置有关工作单元初值，调用 readi (　)。

3、readi (　)执行：

(1) 确定是块设备文件，还是字符设备文件，是后者则通过字符设备开关表转到特别文件处理。是块设备文件则转下述的读处理。

(2) 由读写位移 u. u_off_set 得到文件逻辑块号，本次实际传送字节数。并调用映象处理程序 bmap ( )把逻辑块号转换成物理块号。

(3) 确定一般方式读还是提前读，调用 bread ( )或 bread ( )，执行读取一块到缓冲区。

(4) 调用 imove ( )程序，把已读入缓冲区的信息移至内存，并准备读取下一块。

(5) 调用 brelse ( )，释放缓冲区。

(6) 全部文件块读完或出现错误时，则返回；否则继续读下一块文件信息。

由上可知，块设备的读（写也一样）操作（即 I/O 操作），就是文件读或写系统调用的一个内部过程。面向用户的只是文件读写的系统调用，设备的 I/O 操作对用户则是完全透明的。

13.6 UNIX 命令语言 shell

由图 13-1 示出的 UNIX 系统结构图可知，命令语言 shell 是用户与 UNIX 系统交往的界面。虽然 shell 是一种命令语言，但它不仅仅是一些简单有效的、能为直接运行程序提供便利的交互命令，它也是一种命令级程序设计语言，能够用其来编制复杂的命令程序。因此，UNIX 的 shell 是命令语言、命令级程序设计语言及两者的解释程序的总称，形成了 UNIX 系统的重要特点之一。

13.6.1 shell 命令语言

1、简单命令

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，arg1、arg2、… agrn 是该命令执行时需要携带的参数，命令名和各参数之间用空格分开。简单命令可以没有参数，有些参数是可选项。例如：

```
pwd
```

是一个简单命令，意思是打印出工作目录（或称当前目录）的名字，它没有带任何参数。又如：

```
cp disarm/ * newlit
```

也是一个简单命令，意思是把子目录 disarm 里的全部文件拷贝到子目录 newlit 里，它带有两个参数 disarm/ *和 newlit。下面列出一些简单命令及含义。

命令名	含　义
cat	连接并显示文件
cc	调用 C 编译程序编译文件
cd	改变工作目录
cp	拷贝文件
date	设置日期和时间
ed	进行文本编辑
kill	终止一个进程
lpr	在行式打印机上打印文件

ls	列出目录内容
mkdir	建立一个目录
ps	显示进程的状态信息
pwd	打印工作目录名
rm	删除文件
rmdir	删除目录
who	列出使用系统的用户名

这不是简单命令的全部，只是通过这几个命令，你可以悟出 shell 命令语言中简单命令的各种功能。简单命令中的参数是用来向所执行的程序传递信息用的，如果参数中有任选项，则应列在命令名的后面。在 UNIX 系统里，习惯上在可选参数的前面都加上一个连字符“—”。譬如说，单纯的 ls 命令是列出目录内所含文件的名字和类型，但若带有任选项“-1”，即 ls-1，则除列出文件名和类型外，还要列出诸如文件主名，各类用户的存取权限，文件长度等更为详细的信息。

2、后台命令——&

有时你要运行一个需花很长时间才能结束的程序，在它运行期间并不需要从终端做什么输入，输出在最后产生。于是你只得耐心地等待它，直到运行完毕。shell 有一个特殊的功能：在启动一个这样的程序之后，可以随它去做，而你则继续输入你的 shell 命令。我们称前面的那个程序是在后台运行，你随之输入的 shell 命令则是运行在前台，后台进程和前台进程同时在做。

为了实现这一功能，只需你在某个 shell 命令之后放上一个“&”符号（之间应有一个空格）。这样，shell 就在后台开始做这一命令（执行这个程序），同时允许你输入别的命令。

例如，你打算运行一个叫做 acct 的程序，它要花费较长时间，这时你可以键入命令

```
acct &
```

于是 shell 就在后台启动 acct 运行，并显示出 acct 这个进程的标识数。随后，你就可以在终端键入别的 shell 命令了。譬如说，你想在编译 prog1.c 的同时，对 prog2.c 进行编辑，那么你可以发如下命令：

```
cc prog1.c & ED prog2.c
```

shell 在接到这一命令序列后，先调用 C 的编译程序，让它在后台对 prog1.c 进行编译，并打印出执行这一任务的进行标识数。然后就转入前台，来对 prog2.c 进行编辑。很明显，这种前、后台的工作方式，大大提高了系统和用户的工作效率。

3、输入、输出重定向

计算机终端是计算机和用户之间的基本通信设备，如果不做特别说明，shell 命令的输入总是发往终端显示器，输入总是来自终端键盘，因此终端键盘和显示器是 shell 命令的标准输入和标准输出。例如，PS、LS、SHO、PWD 等命令程序都是使用标准输出为你提供它们的信息，而交互程序（如 shell 和编辑程序）则从标准输入读得你的各种命令，并通过标准输入告知它们对命令的响应。

1) 输入重定向

由于标准输入和标准输出是由 shell 建立起来的，因此可以通过 shell 来重新指定标准输入和标准输出。这种输入、输出重定向，是 UNIX 系统最重要的特性之一。

例如，一般地输入命令 PS，则 PS 程序就把进程状态信息在标准输入——你的终

端上示出，然而，若输入命令：

```
ps>file1
```

那么结果就和上面略有不同。PS 程序虽然仍是把进程状态信息写到标准输出上去，但由于后面的特殊符号“>file1”，shell 就把标准输出与名为 file1 的一个普通磁盘文件连接起来，进程状态住处被写入了文件 file1，终端屏幕上看不到什么输出。

“>”是一个特殊的 shell 符号，它意味着命令的输入将被送往它后面指出的一般文件。要注意的是，如果 file1 文件里原先就有信息，那么执行 PS>file1 之后，就会对它进行重写，以致原有内容丢失。如果你希望把输出送到这个文件的末尾而不造成原先信息的丢失，那么可发命令：

```
ps>>file1
```

2) 输出重定向

标准输入也可以被重新指定，它使用特殊的 shell 符号“<”，意味着输入源来自于符号右端给出的一般文件。

例如，wc 是一个统计文件中有的字符数、字数和行数的简单命令，如果只发 WC，那么它就统计工作目录中文件里的字数、字符数和行数，但如果发：

```
wc<file2
```

则输入源就成为 file2，wc 将去统计 file2 中的字符数、字数、以及行数，结果在屏幕上显示。

输入和输出可以同时重定向，例如：

```
wc<file2>file1
```

则意味着把 WC 统计出的 file2 中的字符数、字数和行数送入文件 file1 保存，而不在屏幕上加以显示。

4、管道命令——“|”

管道（pipe）把一个程序（命令）的标准输出连接到另一个程序（命令）的标准输入上。管道与前面所讲的输入/输出重新定向不同，输入重定向是把程序的输入写到指定的文件上，输入重定向是从指定文件获取输入的内容，而管道则直接地把一个程序的输入作为另一个程序的输入，中间无需通过文件来介绍。例如，下面的一组 shell 命令：

```
cat file1 file2>file3
wc <file3
rm file3
```

其功能是先由命令 CAT，把文件 file1、file2 连接起来输入到文件 file3，然后用命令 WC 统计出文件 file3 所包含的字符数、字数以及行数，在屏幕上加以显示，最后用命令 RM 将文件 file3 删除。这里文件 file3 实际上是为了统计而产生的一个临时文件，它是命令 CAT 的输出，又是命令 WC 的输入。这种情形恰能利用管道命令，以便一次性地完成上述功能，省去不必要的临时性文件。上面的 shell 命令组的功能可以借助于管道命令：

```
cat file1  file2 | wc
```

来完成。由此可见，管道命令可以使命令更为简捷、便当。

5、简单命令、管道、命令表

我们已经知道，简单命令是由命令名及其参数构成，它是 UNIX 的 shell 中最为基本的单位，还有两个基本的单位就是管线和命令表。

所谓一个管线，它或是一个简单的命令，或是由管道符“|”装配起来的一简单命令组。下面每一行都是一个管线：

```
ls  /bin
who  |wc-1
ps
```

所谓命令表，是管线序列的意思。例如上面的三个管线就构成一个命令表，该表里的元素（管线）位于不同的行中，用回车换行符隔开。

下面的命令表与上面给出的等价：

```
ls /bin; who | wc-1; ps
```

在这种命令表里，元素之间用分号隔开。

命令表是 UNIX 系统中的一种基本结构，一个命令表可以简单到就只是一条单个命令，也可以复杂到你希望得到的形式，一个命令表返回的值是表中最后一个管线的出口状态。应该理解简单命令、管线和命令表三者之间的不同之处：

(1) 简单命令执行一个程序；

(2) 管线是由管道符连接起来的简单命令序列，最简单的管线即是一条简单命令；

(3) 命令表是管线序列，最简单的命令表即是单独一个管线（也就可能是一条简单命令）。

6、通配符

通配符的概念大家已经非常清楚，在 shell 命令参数里的文件名中也广泛地采用通配符，以方便命令等的键入。在 UNIX 里，以下各种通配符用来控制文件名的生成：

*	匹配任何字符串
?	匹配任何单个字符
[	引入一个字符组
]	结束一个字符组
-	指定一个字符范围

星号“*”和问号“?”使用起来很简便。星号将匹配任何字符串，包括空串在内，因此*.c 代表以 c 为扩展名的所有文件名，它将与诸如“.c”、“a.c”或“aaa.c”的文件名匹配，但不能与文件名“a.ca”匹配；问号匹配任何单个字符，于是，??.c 将能与诸如“ab.c”或“77.c”的文件名匹配，而不能与文件名“a.c”、“abc.c”或“bc.cc”匹配；方括号和连接符用来形成一个字符组，组中的字符被括在方括号内。例如 abc[aeiou]将与以字符串 abc 开头、以单个元音结束的文件名匹配。连接符可用在方括号里，以指定字符的范围。例如 def[0-9]将与前三个字符为 def、第四个字符（且是最后一个字符）是数字的文件名匹配。

在 shell 中，诸如“*”、“?”等字符已失去了它原有的含义而成为具有特殊意义的字符，在 UNIX 中称它们为元字符，“>”、“,”、“|”等都属于元字符。如果在某种场合下需要去除它们的特殊含义而按一般字符对等，那么就应该在这一元字符前加一个反斜杠“\”即可。例如，*就是通常意义下的星号，而不再是通配符。要注意的是，出现在两个单引号之间的字符串中若包含元字符，一律按一般字符处理。

13.6.2 shell 程序设计语言

UNIX 系统的 shell，既是一个交互命令的解释程序，也是一个命令级程序设计语言的解释程序。某些计算机系统具有简单而有效的交互命令解释程序，但却缺少编制

精巧命令程序的能力；另一些计算机系统灵便的命令级程序设计语言，虽然功能完备有力，但却未能提供程序运行的简便手段。UNIX 则把这两种能力全部归入了 shell，成为 shell 的一大特色。

前面提及的 shell 的典型交互式应用，例如简单命令（“ls”），文件名生成手段（“ls *.doc”），I/O 重定向（“ls>myfile”）以及管线（“ls|wc-1”）等，功能不仅强，而且非常有用，但它们仅是 shell 能力的很小一部分。

1、执行一个shell程序

把 shell 命令序列存放在一个文件里，就构成了一个 shell 程序或一个命令文件。通常，当该文件里只包含 shell 命令的简单序列时，就使用命令文件这一术语；当文件里对命令有更为复杂的安排（如使用了 shell 的条件命令等）时，就使用 shell 程序这一术语。

执行一个 shell 程序的最一般的形式是：

```
sh programname arg1 arg2 … argn
```

这里，sh 为命令名，programname 为所要执行的 shell 程序名，arg1、arg2、…、argn 则是 shell 程序执行时需要的参数。

2、shell变量

shell 程序设计语言用变量来存放值，shell 变量只能存放字符串，使用一个赋值命令就可为一个变量置值。如：

```
ux = u.UNIX
```

这个赋值命令把值 u.UNIX 赋给了名为 un 的 shell 变量。

一个 shell 变量的名字必须以一个字母打头，然后可以包含字母、数字或下划线。由于 ux 是一个变量，因此对它施行另一个赋值：

```
un = PCDOS
```

就会立即将其值加以变更。

在你要使用存放在一个变量里的值时，必须在该 shell 变量名前放上一个‘$’符号，该符号告知 shell，随后的名字涉及的是一个变量而不是一个文件。

3、shell程序中的控制语句

做为一种程序设计语言，shell 中提供了各种程序结构的控制语句，它们是：

- if 语句；
- while 语句；
- for 语句；
- case 语句；
- break 和 continue 语句等等。

这里不可能把 shell 做为一种程序设计语言时所具有的功能全都讲述出来，但通过粗略的介绍就可以知道，shell 作为一种程序设计语言，其功能确实是很强的。