Operating Systems: Design and Implementation
Second Edition
操作系统设计与实现
第二版
安德鲁.坦尼鲍姆（Andrew S. Tanenbaum）
阿尔伯特.伍德豪尔（Albert S. Woodhull）

作者简介

安德鲁.坦尼鲍姆分别在麻省理工学院和加州大学伯克利分校获得学士和博士学位。他现任位于荷兰阿姆斯特丹市的Vrije大学计算机科学教授并领导着一个计算机系统研究小组。同时他还任一个研究并行、分布及图像系统的校际研究生院 — 计算机与图像高级学院的院长。

坦尼鲍姆先前的研究领域包括编译器、操作系统、网络和局域分布式系统，他现在的研究主要集中在可扩展到数百万用户的广域分布式系统。对这些课题的研究使他在学报和会议上发表了70余篇论文，并出版了五部专著。

坦尼鲍姆教授同时还主持开发了大量的软件。他是Amsterdam编译工具箱的总设计师，该工具箱被广泛地用来开发可移植的编译器，同时还用于MINIX的开发。他和他的博士研究生及程序员们一起设计了一个基于微内核的高性能分布式操作系统 — Amoeba。现在，以教学和研究为目的的用户可以从Internet上免费获得MINIX 和Amoeba软件。

坦尼鲍姆的许多博士研究生在获得学位后都取得了非常丰硕的成果，这令坦尼鲍姆非常自豪，因为这是他诲人不倦的结果。

坦尼鲍姆教授同时还是ACM的会士、IEEE高级会员、荷兰皇家艺术和科学院院士，他曾获得1994年ACM Karl V. Karlstrom 杰出教育奖和1997年ACM/SIGCSE 计算机科学教育杰出贡献奖。他被列入Internet上的 Who's Who in the World 名单，他在WWW上的主页地址为：http://www.cs.vu.nl/~ast/。

阿尔伯特.伍德豪尔分别在麻省理工学院和华盛顿大学获得学士和博士学位。他进麻省理工学院本来是想成为一名电气工程师，可是后来却成了生物学家。从1973年起他开始在位于麻省Amherst的Hampshire自然科学学院工作。当微型计算机慢慢多起来的时候，作为使用电子检测仪器的生物学家，他开始使用微型计算机。他给学生开设的检测仪器方面的课程逐渐演变为计算机接口和实时程序设计。

伍德豪尔博士对教学和科学技术的发展有浓厚的兴趣，在进入研究生院之前他曾在尼日利亚教过两年中学，近年来他曾几次利用自己的假期在尼加拉瓜教授计算机科学。

他对计算机作为电子系统，以及计算机与其他电子系统的相互配合很感兴趣。他最喜欢讲授的课程有计算机体系结构、汇编语言程序设计、操作系统和计算机通信。他还为开发电子器件及相关软件担当顾问。

在学术之外，伍德豪尔有不少兴趣，包括各种户外运动，业余无线电制作和读书。他还喜欢旅游和学习别国语言。他的WWW主页就存在一台运行MINIX的机器上，地址是：http://minix1.hampshire.edu/asw/。

前 言

多数操作系统教材都重理论而轻实践，本书希望在这二者之间求取较好的平衡。本书详细论述了操作系统的所有基本概念，包括进程、进程间通信、信号量、管程、消息传递、调度算法、输入/输出、死锁、设备驱动程序、存储器管理、页面调度算法、文件系统设计、安全与保护机制等。同时，本书也详细讨论了MINIX — 一个与UNIX兼容的操作系统，并提供了完整的源代码供学习之用。这样的安排使读者不仅学习到理论，而且能够理解它们如何应用在一个实际的操作系统之中。

本书第一版在1987年出版时，曾引发了操作系统课程教学的一场小小的变革。在此之前多数课程都只讲理论。随着MINIX的出现，许多学校开始增加实验环节以使学生了解实际的操作系统是如何运作的。我们认为这种趋势是可取的，并希望通过本书第二版能进一步加强这种趋势。

MINIX在其出现以来的十年间发生了许多变化，最初的代码是为基于8088芯片、256K内存和两个软驱的IBM PC机型编写的，它基于UNIX 版本7。随着时间的推移，MINIX在许多方面有所发展，比如当前版本可运行在众多机型上，从16位实模式的PC机到配有大容量硬盘的奔腾机（32位保护模式），而且它不再基于UNIX版本7，而是基于国际上的POSIX标准（POSIX 1003.1和ISO9945－1）。与此同时，有许多新特征被添加到MINIX中，在我们看来，所增加的特征可能已经太多了，但有些人则认为还不够，这最终导致了LINUX的诞生。MINIX还被移植到许多其他平台上，包括Macintosh、Amiga、Atari和SPARC。本书只涉及MINIX2.0，到目前为止，该版本只能运行于基于80X86的机器，或者可模拟此类CPU的机器，以及SPARC机器。

与第一版相比，第二版有许多变化，原理性部分基本都被修改过，同时增加了大量新内容。最主要的变化是新的基于POSIX的MINIX，以及对其源代码的剖析。另外，每本书都附带一张CD-ROM，它包含了全部MINIX源代码，以及在PC上安装MINIX的说明（见CD-ROM主目录下的README.TXT文件）。

在一台80X86 的PC机上安装MINIX很方便。它需要一个至少30MB的硬盘分区，然后按照CD－ROM上README.TXT文件中的步骤进行即可。在打印README.TXT文件之前，先启动MS-DOS（若运行WINDOWS，则双击MS-DOS图标），然后键入

```
copy  readme.txt  prn
```

即可。该文件也可以用edit、wordpad、notepad等任何可以处理ASCII正文的编辑器进行浏览。

对于没有PC机的学校和个人，有两种解决办法，即CD－ROM上提供的两个模拟程序。一个由Paul Ashton 为SPARC机器编写，它作为用户程序在Solaris上运行，此时MINIX被编译成SPARC上的可执行文件。在这种模式下，MINIX不再是一个操作系统，而只是一个用户程序，所以必须对其底层作一些修改。

另一个模拟程序由Bochs软件公司的Kevin.P.Lawton编写，它解释Intel 80386的指令集以及足以使MINIX运行所需的I/O指令。显然在解释器层次上运行使性能有所下降，但这使得学生更容易进行调试。该模拟程序运行在所有支持M.I.T的X-Window的系统上，更详细的信息请参看CD－ROM上的有关文件。

MINIX仍在继续发展，本书和CD－ROM中的内容仅仅反映了本书出版时的情况，有关MINIX的最新动态请访问MINIX的主页：http://www.cs.vu.nl/~ast/minix.html。MINIX也有USENET中的新闻组：comp.os.minix，读者可以订阅该新闻组。对于仅有Email的读者可通过以下步骤来加入MINIX的邮件用户通信组。给listserv@listserv.nodak.edu 发一封信，其中只需一行字：“subscribe minix-1 <您的完整用户名>”，此后你便会通过E-mail获得很多的信息。

讲授本课程的教师可以从Prentice Hall公司获得一份习题解答手册。从WWW地址http://www.cs.vu.nl/~ast/沿着“Software and supplementary material”链接可以获得一些有用的PostScript文件，其中包含本书中所有的图表，可供需要时使用。

在MINIX的开发项目中我们有幸得到了许多人的帮助。首先要感谢Kees Bot在MINIX标准化和软件发布中所作的大量工作，没有他的帮助，我们不可能完成这件工作。他自己编写了大量的代码（如POSIX终端I/O）并修正了一些数年来一直存在的错误，他还整理了其他的代码。

这些年来Bruce Evans、 Philip Homburg、Will Rose和Michael Temari为MINIX的开发做了大量的工作。 有几百人通过新闻组对MINIX作出了贡献，他们人数众多，所作出的贡献也各不相同，在此谨向他们一并表示感谢。

John Casey、 Dale Grit、 Frans Kashoek等人阅读了本书的部分手稿并提出了宝贵建议，在此向他们表示谢意。

Vrije大学的许多学生测试了CD－ROM中MINIX的 β版本，他们是：Ahmed Batou, Goran Dokic, Peter Gijzel, Thomer Gil, Dennis Grimbergen, Roderick Groesbeek, Wouter haring, Guido Kollerie, Mark Lassche,

Raymond Ris, Frans ter Borg, Alex van Ballegooy, Ries van der Velden, Alexander Wels以及Thomas Zeeman。我们对他们细致的工作和详尽的报告致以衷心的感谢。

阿尔伯特·S·伍德豪尔向他从前的几位学生表示感谢，特别是Hampshire学院的Peter W. Young ，Nacional Autonoma de Nicaragua大学的Maria Isabel Sanchez 和William Puddy Vargas。

最后要向我们的家庭成员表示感谢。Suzanne 已是第十次在我埋头写作时给我支持，对Barbara是第九次，Marvin是第八次，甚至小Bram也是第四次了。他们的支持和爱心对我非常重要。（坦尼鲍姆）

至于阿尔伯特的Barbara，这倒是第一次，假如没有她的支持，耐心和幽默，我们是不可能完成这一工作的，对我的儿子Gordon而言，由于在编写本书时，他大部分时间都不在家中，而是在大学学习，因此是非常幸运的。但是他的理解和关心深深吸引着我从事本书的编写工作，有这样一个儿子是令人非常愉快的。（伍德豪尔）

安德鲁.S.坦尼鲍姆

阿尔伯特.S.伍德豪尔

译　序

坦尼鲍姆教授是国际知名的计算机科学家和教育家。他在操作系统、分布式系统以及计算机网络领域都有很深的造诣。自八十年代以来，他已先后出版了一系列面向大学生和研究生的教材性质的专著，并被世界各国的许多大学广泛采用。这本书就是他的最新专著之一。

操作系统是计算机系统中最核心和最底层的软件，对操作系统的深入学习关系到对整个系统运作机制的全面理解，因此一本好教材也显得愈发重要。本书的英文版出版于1997年，其中涵盖了操作系统课程的所有内容，即传统上的进程管理、存储器管理、文件管理和设备管理。同时其中又包含了许多新内容，如线程、基于消息传递的系统构造模型、日志结构文件系统、安全和保护机制、RAM盘及CD－ROM设备等，而用作例子的CPU则为Interl Pentium。这使得读者一方面能够学习操作系统的经典内容，另一方面又能够了解和跟踪当前的最新技术和研究成果。

本书的另一个特点是基本原理与具体实例，即MINIX紧密结合。第2到第5章的前半部分讲述原理，后半部分则详细地解释这些原理在MINIX的设计和实现中的应用。通过阅读这些部分能够把握MINIX源代码的组织方式，并理解那些很关键或者很难懂的代码。这部分内容非常翔实，有时甚至逐行地解释附录中所列的源程序。对操作系统课程多年的授课经验以及相关的科研工作使我们认识到：详细地剖析一个象MINIX这样的操作系统对于掌握操作系统设计与实现的精髓是大有裨益的。

正因为上述原因，我们真切地感受到将这本书翻译、介绍给国内读者将是一件非常有意义的事，衷心希望我们付出的劳动能对国内的操作系统教学和实践有所帮助和促进。

本书的第一章，第二章，第三章由王鹏翻译，刘福岩和陆宁也参加了部分工作；第四章由朱鹏翻译；第五章由敖青云翻译。全书由尤晋元教授审校并统稿。

在整个翻译过程中，上海交通大学计算机系系统软件研究室的师生给予了许多帮助。并且在计算机系95级本科生的操作系统课程中进行了试用，许多学生提出了很好的建议，在此向他们表示衷心的感谢。

特别要感谢本书的责任编辑邓又强先生，本书的顺利出版与他的辛勤劳动和热情支持是分不开的。

虽然在翻译过程中我们尽力恪守“信，达，雅”的准则，但不当和疏漏之处在所难免，敬请读者提出宝贵建议。

译　者

1998年4月于上海交通大学

第一章　引 言

计算机如果离开了软件将成为一堆废铜烂铁。有了软件，计算机可以对信息进行存储、处理和检索，可以显示多媒体文档、搜索Internet并完成其他工作。计算机软件大致分为两类：系统软件和应用软件。系统软件管理计算机本身及应用程序；应用软件执行用户最终所需要的功能。最基本的系统软件是操作系统（operating system），它控制计算机的所有资源并提供开发应用程序的基础。

现代计算机系统包含一个或多个处理器、若干内存（常称为RAM－随机存取存储器）、磁盘、打印机、网络接口及其他输入/输出设备。编写一个程序来管理所有这些器件以正确地使用它们，即使不考虑优化也是一件很困难的事情。如果每个程序员都必须处理磁盘如何工作，再加上每读一个磁盘块都有几十种因素可能导致操作出错，那么很多程序简直没法写。

许多年以前人们就认识到必须找到某种方法将硬件的复杂性同程序员分离开来。经过不断探索和改进，目前采用的方法是在裸机上加载一层软件来管理整个系统，同时给用户提供一个更容易理解和进行程序设计的接口，这被称为虚拟机（virtual machine）。这样一层软件就是操作系统。

这种处理方式如图1－1所示。底层是硬件，它本身可能包括两层或多层。最低一层是物理器件，包括集成电路芯片、连线、电源、监视器等，它们的构造和工作方式属于电气工程师的范围。

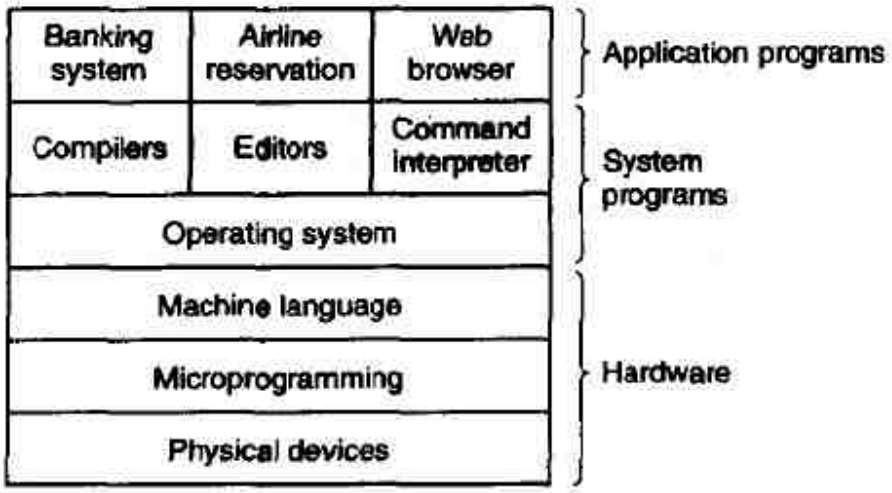

图 1-1　计算机系统由硬件、系统程序和应用程序组成。

接着是微程序（microprogram），通常存放在只读存储器中，它是一层很原始的软件，用来控制设备并向上一层提供一个更清晰的接口。微程序实际上是一个解释器，它先取得机器语言指令，如ADD， MOVE和JUMP等，然后通过一个动作序列来执行这些指令。例如为了执行一条ADD指令，微程序必须先确定运算数据的位置，然后取数，相加，最后存放得数。由微程序解释执行的这一套指令称为机器语言。机器语言并不是硬件的组成部分，但硬件制造商通常在手册中给出机器语言的完整描述，所以许多人将它认作真正的“计算机”。

采用精简指令集计算机（RISC）技术的计算机没有微程序层，其机器指令通过硬件逻辑直接执行。例如Motorola 680X0有微程序，而IBM PowerPC 则没有。

机器语言典型地有50到100条指令，大多数用来完成数据传送、算术运算和数值比较等操作。在这个层次上，通过向特殊的设备寄存器写特定的数值来控制输入/输出设备。例如将磁盘地址、内存地址、读字节数和操作类型（读/写）等值写入特定的寄存器便可完成硬盘读操作。实际操作往往需要更多的参数，而操作完成后的返回状态也非常复杂。进一步而言，对于许多I/O设备，时序在程序设计中的作用非常重要。

操作系统的主要功能之一就是将所有这些复杂性隐藏起来，同时为程序员提供一套更加方便的指令，比如，“从文件中读一个数据块”在概念上比低层的“移动磁头臂，等待旋转延迟”之类的细节来得简单、方便。

在操作系统之上是其他系统软件，包括命令解释器（shell）、窗口系统、编译器、编辑器及类似的独立

于应用的程序。要注意它们本身并不是操作系统的组成部分，尽管它们通常由计算机厂商提供。这一点很重要，操作系统专指在核心态（kernel mode），或称管态（supervisor mode）下运行的软件，它受硬件保护而免遭用户的篡改。编译器和编辑器运行在用户态（user mode）。如果用户不喜欢某一个编译器，他可以自己重写一个，但他却不可以写一个磁盘中断处理程序 — 因为这是操作系统的一部分，而且硬件阻止用户对它进行修改。

系统软件之上是应用软件，这些软件可以是购买的或者是用户自行开发的，它们用来解决特定的问题，如字处理、表格处理、工程计算或者电子游戏等。

1.1 什么是操作系统

多数计算机用户都使用过操作系统，但要精确地给出操作系统的定义却很困难，部分原因是操作系统完成两项相对独立的任务，下面我们逐项进行讨论。

1.1.1 操作系统作为虚拟机

对多数计算机而言，在机器语言一级的体系结构（指令集、存储组织、I/O和总线结构）上编程是很困难的，尤其是输入输出操作。例如考虑使用多数PC机采用的NEC PD765控制器芯片（或功能等价的芯片）来进行软盘I/O操作。PD765有16条命令，它通过向一个设备寄存器装入特定的数据来执行这些命令，命令数据长度从1到9字节不等，其中包括：读写数据、移动磁头臂、格式化磁道、初始化、检测磁盘状态、复位、校准控制器及设备等。

最基本的命令是读数据和写数据。它们均需要13个参数，所有这13个参数被封装在9个字节中。这些参数指定的信息有：欲读取的磁盘块地址、每条磁道的扇区数、物理介质的数据记录格式、扇区间隙、以及对已删除数据地址标识的处理方法等。当磁盘操作结束时，控制器芯片返回23个状态及出错信息，它们被封装在7个字节中。此外，程序员还要注意步进电机的开关状态。如果电机关闭，则在读写数据前要先启动它（有一段较长的加速时间）。还要注意电机不能长时间处于开启状态，否则将损坏软盘，所以程序员必须在较长的启动延迟和可能对软盘造成损坏之间作出折衷。

显然，程序员不想涉及硬件的这些具体细节（也包括硬盘，它与软盘不同，但同样很复杂）。他需要的是一种简单的高度抽象的设备。一种典型的抽象是一张磁盘包含了一组命名的文件，每个文件可以被打开，然后进行读写，最后被关闭。其中的一些细节如数据记录格式、当前步进电机的开启状态等则对用户隐藏。

这种将硬件细节与程序员隔离开来、同时提供一个简洁的命名文件方式的程序，就是操作系统。与磁盘抽象类似，它还隐藏了其他许多低层硬件的特性，包括中断、时钟、存储器等。总之，操作系统提供的每一种抽象都较低层硬件本身更简单、更易用。

从这个角度看，操作系统的作用是为用户提供一台等价的扩展计算机，或称虚拟机，它比低层硬件更容易编程。本书的内容正是详细说明操作系统如何做到这一点。

1.1.2 操作系统作为资源管理器

上述虚拟机模型是一种自顶向下的观点。按照自底向上的观点，操作系统则用来管理一个复杂系统的各个部分。现代计算机都包含处理器、存储器、时钟、磁盘、鼠标、网络接口、激光打印机以及其他许多设备，从这个角度看，操作系统的任务是在相互竞争的程序间有序地控制这些设备的分配。

设想在一台计算机上运行的三个程序同时试图在一台打印机上输出计算结果，那么可能头几行是程序1的输出，接下来几行是程序2的输出，然后又是程序3的输出等等，最终打印结果将是一团糟。操作系统采用将打印输出缓冲到磁盘上的方法可以避免这种混乱。当一个程序结束后，操作系统将暂存在磁盘文件上的输出结果送到打印机，同时其他程序可以继续运行产生新的输出结果，而这些程序并不知道这些输出没有立即送至打印机。

当一台计算机（或网络）有多个用户时，因为用户间可能相互影响，所以管理和保护存储器、I/O设备以及其他设备的需求随之增加。而且用户往往不仅需要共享硬件，还要共享信息（文件、数据库等）。总之，此时操作系统的首要任务是跟踪资源的使用状况、满足资源请求、提高资源利用率、以及协调各程序和用户对资源的使用冲突。

1.2 操作系统发展历史

操作系统经历了一个漫长的发展过程，下面对此进行简要的回顾。由于在历史上操作系统与计算机体系结构存在非常密切的联系，我们将按照计算机的换代历程讲述操作系统的发展状况。

第一台真正的数字计算机是英国数学家Charles Babbage（1792－1871）设计的。Babbage投入毕生精力去建造他的“分析机”，但却没能让它成功地运行起来。因为它是纯机械式的，而当时的技术不可能使分析机的零部件达到他所需要的精度。很显然，分析机没有操作系统。

有趣的是，Babbage认识到他的分析机需要软件，于是他雇佣了一个年轻的女子，英国著名诗人拜伦的女儿Ada Lovelace为他工作。Ada由此成了世界上第一位程序员，Ada程序设计语言就是用她的名字命名。

1.2.1 第一代计算机（1945－1955）：真空管和插板

从Babbage之后一直到二战，数字计算机几乎没有什么进展。在40年代中期，哈佛大学的Howard Aiken、普林斯顿高等研究院的John.Von Neumann（冯.诺依曼）、宾夕法尼亚大学的J.Presper Eckert和William Mauchley、德国电话公司的Konrad Zuse、以及其他一些人都成功地使用真空管建造了计算机。这些机器非常庞大，往往使用数万个真空管，占据几个房间，然而其运算速度却不如现在最便宜的个人计算机。

在计算机出现的早期，每台机器都有一个小组专门来设计、制造、编程、操作和维护。编程全部采用机器语言，通过在一些插板上的硬连线来控制其基本功能，这时没有程序设计语言（甚至没有汇编语言），操作系统更是闻所未闻。机器的使用方式是程序员提前在墙上的机时表上预约一段时间，然后到机房将他的插板插到计算机里，在接下来的几小时里计算自己的题目。这时的计算机很不可靠，因为几万个真空管中任何一个发生故障，计算机就无法运行。这个阶段基本上所有的题目都是数值计算问题，例如计算正弦和余弦函数表。

到50年代早期，出现了穿孔卡片，这时就可以不用插板，而是将程序写在卡片上然后读入计算机，但其他过程则依然如故。

1.2.2 第二代计算机（1955－1965）：晶体管和批处理系统

50年代发明的晶体管极大地改变了计算机的状况。这时的计算机已经很可靠，厂商可以成批地生产计算机并卖给客户，客户可以长时间地使用它来完成一些有用的工作。至此，第一次将设计人员、生产人员、操作员、程序员和维护人员分开。

这个时期计算机安装在空调房间里，有专人操作。由于其价格昂贵，仅有少数大公司、主要的政府部门和大学才买得起。运行一个作业（一个或一组程序）时，程序员首先将程序写在纸上（用FORTRAN或汇编语言），然后用穿孔机制成卡片，最后将这些卡片交给操作员。

计算机运行完当前任务后，其计算结果从打印机上输出，操作员从打印机上取得运算结果并送到输出室，程序员就可从该处取到运算结果，然后，操作员再从卡片上读入另一个任务。如果需要FROTRAN编译器，操作员还要从别处取来读入计算机。当操作员在机房里走来走去时，许多机时被浪费掉了。

由于当时计算机非常昂贵，很自然地人们开始想办法减少机时的浪费，答案就是批处理系统。其思想是：在作业输入室收集到较多的作业，然后用一台相对廉价的计算机（如IBM 1401计算机，它适用于读卡片、拷贝磁带和打印输出，但不适用于作数值运算。）将它们读到磁带上，另外用较昂贵的计算机如IBM 7094来完成真正的计算。该模型如图1－2所示。

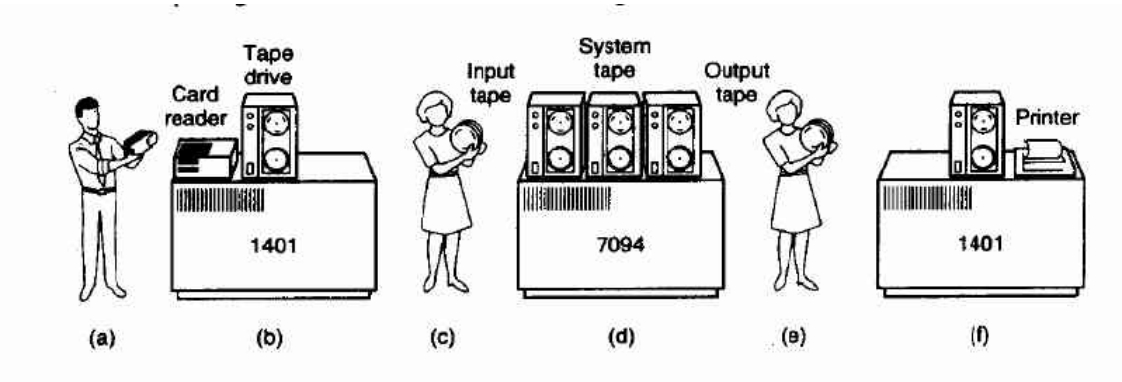

图 1-2　一种早期的批处理系统。(a)程序员将卡片拿到1401机处　(b)1401将批处理作业读到磁带上　(c)操作员将输入带送至7094处　(d)7094进行计算　(e)操作员将输出带送至1401处　(f)1401打印输出

在收集到一批作业之后，输入磁带被送到机房里装到磁带机上。操作员随后装入一个特殊的程序（现代操作系统的前身），它从磁带上将第一个作业读入并运行，其输出写到第一盘输出磁带上，而不是打印出来。每个作业结束后，操作系统自动地读入下一个作业运行。当这一批作业完全结束后，操作员取下输入和输出磁带，将输入磁带换成下一批作业，然后把输出磁带拿到一台1401机器上进行脱机打印。

一个典型的输入作业结构如图1－3所示。它由一张＄JOB卡片开始，该卡标识出所需的最大运行时间（分钟）、计费标识、以及程序员的名字。随后是一张＄FORTRAN卡片，它通知操作系统从系统磁带上装入FROTRAN语言编译器。在此之后是待编译的源程序，然后是＄LOAD卡片，它通知操作系统装入刚编译好的目标程序（编译好的目标程序通常写到暂存磁带上，需要显式装入）。接着是＄RUN卡片，它告诉操作系统运行该程序并使用其后的数据。最后，＄END卡片标识作业结束。这些原始的控制卡片是现代作业控制语言和命令解释器的先驱。

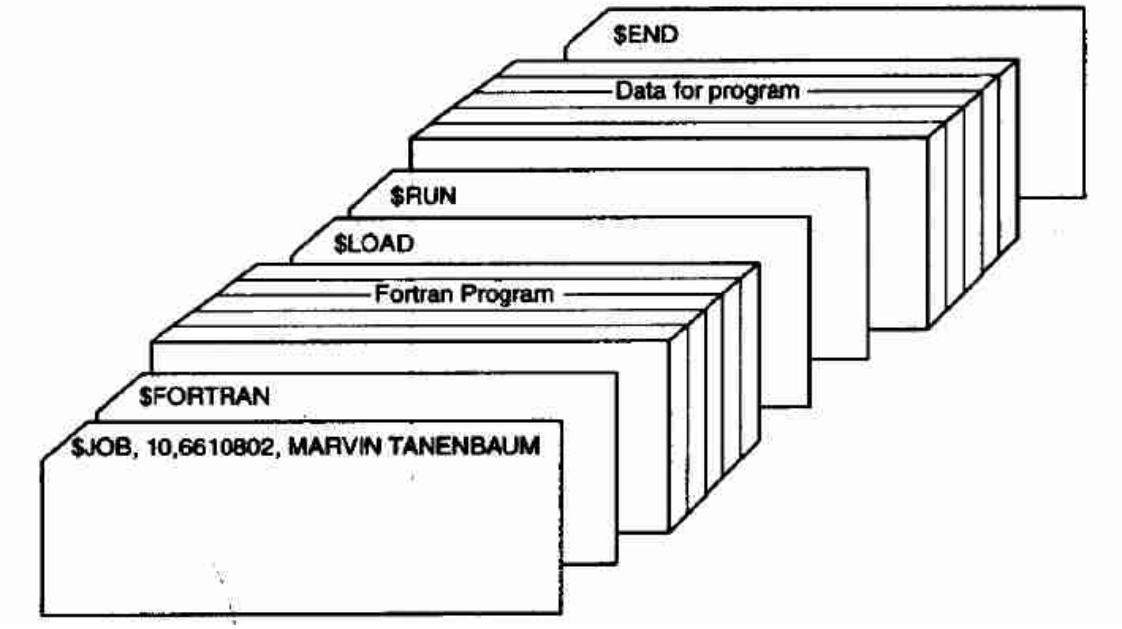

图 1-3　一个典型的FMS作业的结构。

第二代计算机主要用于科学计算，例如解偏微分方程。这些题目大多用FORTRAN语言和汇编语言编程。典型的操作系统是FMS（FORTRAN Monitor System—FORTRAN监控系统）和IBSYS（IBM为7094机配备的操作系统）。

1.2.3 第三代计算机（1965－1980）：集成电路芯片和多道程序

在60年代初期，多数计算机厂商都有两条完全不同并且互不兼容的生产线：一条是面向字的复杂科学计算和工程计算的计算机，如IBM 7094；另一条是面向字符的商用计算机，如IBM 1401，主要被银行和保险公司用于磁带归档和打印服务。

对厂商来说，开发和维护两种完全不同的产品是很昂贵的。同时，许多新的计算机用户开始时只需要一台小计算机，到后来则可能需要一台较大的计算机，而且要求能够更快地执行原有的程序。

IBM公司试图通过引入360系统来解决这两个问题。IBM 360是一个软件兼容的计算机系列，在该系列中，低档机与1401相当，高档机则比7094功能强很多。这些计算机只在价格和性能（最大存储器容量、处理器速度、允许的I/O设备数量等）上有差异。由于所有的计算机都有相同的体系结构和指令集，因此为一种型号机器编写的程序可以在其他所有型号的机器上运行（起码在理论上可行），而且360被设计成既可用于科学计算，又可用于商业计算，这样一个系列的计算机便可以满足所有用户的要求。在随后的几年里，IBM陆续推出了360的后续机型，如用户熟知的370、4300、3080和3090系列。

360是第一种采用集成电路（小规模）芯片的主流机型。与采用分立晶体管制造的第二代计算机相比，其性能价格比有很大提高。360很快就获得了成功，由此其他主要厂商也很快采纳了系列兼容机的思想。这些计算机至今仍在各地的计算中心使用，但其应用正在急剧地萎缩。

“单一家族”思想的最大优点同时也是其最大的缺点。原因是所有软件，包括操作系统，都要能够在所有机器上运行 — 从小的代替1401的机器到用于科学计算的相当于7094的大型机；从只能带很少外部设备的机器到能带很多外设的机器；从商业领域到科学计算领域等，总之，要有效地适用于所有的用途。

IBM无法写出同时满足这些需求相互冲突的软件，其结果是一个庞大的极其复杂的操作系统，它的规模比FMS高大约二到三个数量级。其中包含有数千名程序员写的数百万行汇编语言代码。同时，其中也有成千上万处错误。这就导致IBM不断地发行新版本来更正这些错误，而新版本在改正老错误的同时又引入新错误，所以错误数可能保持大致相同，而不是减少。

OS/360的设计者之一Fred Brooks后来写过一本书（Brooks,1975）来描述他在开发OS/360过程中的经验。这里不可能复述该书的全部内容，不过其封面是一群史前动物陷入一个泥坑不能自拔，Silberschatz和Galvin的著作的封面也表达了同样的观点。

抛开OS/360的庞大和存在的问题，它和其他公司的类似的第三代操作系统的确很好地满足了大多数用户的要求。同时它们也使第二代操作系统缺乏的几项关键技术得到广泛应用。其中最重要的是多道程序（multiprogramming）。在7094机上，若当前作业因等待磁带或其他I/O而暂停时，CPU就只能简单地踏步直至该I/O结束。对于CPU操作密集的科学计算问题，I/O操作较少，因此浪费的时间很少；然而对于商业数据处理，I/O操作等待时间通常占到80－90％，这时必须采取某种措施减少CPU时间的浪费。

经过探索找到的解决办法是将内存分为几个部分，每一部分存放不同的作业，如图1－4所示。当一个作业等待I/O操作完成时，另一个作业可以使用CPU。如果内存中可以存放足够多的作业，则CPU利用率可以接近100％。在主存中同时驻留多个作业需要特殊的硬件来对其进行保护，以避免作业的信息被窃取或受到攻击，幸运的是360及其他第三代计算机都配有此类硬件。

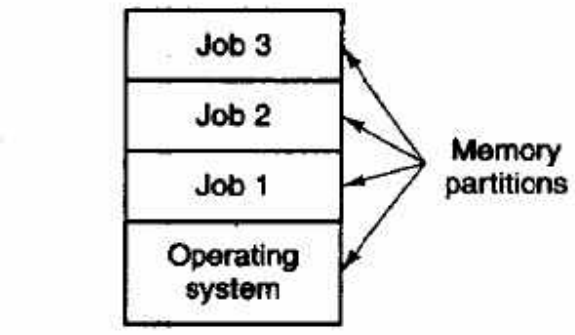

图 1-4　内存中有三个作业的一个多道程序系统。

第三代计算机的另一个新特性是：卡片被拿到机房后能够很快地将一个作业从卡片读入磁盘。于是无论任何时刻当一个作业运行结束，操作系统就能将一个新作业从磁盘读出，装入空出来的内存区域运行，这种技术

叫做spooling（Simultaneous Peripheral Operation On Line — 联机的即时外部设备操作），同时该技术也用于输出。当采用了spooling技术后，就不再需要IBM 1401机，也不必再将磁带搬来搬去。

尽管第三代操作系统很适于大型科学计算和繁忙的商务数据处理，但其实质上仍旧是批处理系统。许多程序员很怀念第一代计算机的使用方式，那时他们可以独占一台机器几个小时，可以即时地调试他们的程序。而对第二代计算机，一个作业从提交到取回运算结果往往长达数小时。更有甚者，一个误用的逗号就会导致编译失败，从而可能浪费程序员半天时间。

程序员们希望很快得到响应，这种需求就导致了分时系统的出现。它实际上是多道程序的一个变种，不同之处只是每个用户都有一个联机终端。在分时系统中，假设有20个用户登录，而其中17个在思考或喝咖啡，则CPU可轮流分配给那三个需要得到服务的作业。由于调试程序的用户常常只发出简短的命令（如编译一个源文件），而很少执行费时的长命令（如将一个上百万条记录的文件排序），所以计算机能够为一些用户提供快速的交互式服务，同时在CPU空闲时还能运行后台的大作业。第一个分时系统CTSS是由M.I.T在一台改装过的7094机上开发成功的（Corbato等，1962），但直到第三代计算机广泛采用了必需的保护硬件之后分时系统才逐渐流行开来。

在CTSS研制成功之后，M.I.T、贝尔实验室和通用电气公司（GE，当时一个主要的计算机制造厂商）决定开发一种“公用计算服务系统” — 一种能够同时支持数百名分时用户的机器。它借鉴于供电系统 — 当你需要电能时，只需将电气设备接到墙上的插座即可。该系统称作MULTICS（MULTiplexed Information and Computing Service），其设计者着眼于建造一台机器来满足整个波士顿所有用户的计算需求。在当时看来，仅30年之后只花几千美元就能买一台计算能力远远超过他们的GE－645的个人计算机的想法完全是科学幻想。

MULTICS引入了计算机领域许多概念的雏形，但其研制难度却超出了所有人的预料。在开发过程中贝尔实验室退出了该项目，通用电气公司也退出了计算机领域。最终MULTICS被成功地应用在M.I.T的实际生产环境以及其他几十个系统中。但“公用计算服务系统”的概念却随着计算机价格的暴跌而被人们遗弃，不过MULTICS对随后的系统却有着巨大的影响，详细请参阅（Corbato etc., 1972; Corbato and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; Saltzer, 1974）。

第三代计算机的另一个主要进展是小型机的崛起，这以1961年DEC的PDP－1作为起点。PDP－1计算机只有4K个18 比特的内存，每台售价120,000美元（不到IBM 7094的5%），该机型非常热销，对于某些非数值的计算，它几乎和7094一样快。PDP－1开辟了一个全新的产业。很快PDP有了一系列机型（与IBM系列机不同，它们互不兼容），其顶峰为PDP－11。

贝尔实验室一位曾参加过MULTICS研制的计算机科学家Ken Thompson，在一台无人使用的PDP－7机器上开发了一个简化的单用户版MULTICS，他的工作导致了后来UNIX操作系统的诞生。UNIX在学术界变得很流行，同时也包括政府部门和许多公司。

有专门的著作讲述UNIX的历史（例如Salus, 1994）。简单地说，由于UNIX的源代码公开，所以许多组织都开发了他们各自的UNIX版本，这些版本互不兼容，所以非常混乱。为了使同一个程序在所有不同的UNIX系统上都能运行，IEEE拟定了一个UNIX的标准，称作POSIX，该标准现在被大多数UNIX支持。POSIX定义了相互兼容的UNIX系统必须支持的一个最小的系统调用接口，实际上，一些其他操作系统现在也支持POSIX接口。

1.2.4 第四代计算机（1980－现在）：个人计算机

随着大规模集成电路的发展，芯片在每平方厘米的硅片上可以集成数千个晶体管，于是个人计算机时代到来了。从体系结构上看，个人计算机与PDP－11并无二致，但就价格而言却相去甚远。通常公司的一个部门或大学里的一个院系配备一台小型机，而个人计算机却使每个人都能拥有自己的计算机。在商业、大学或政府部门使用的功能最强的个人计算机通常称为工作站，它实际上只是大一点的个人计算机，通常工作站之间通过网络互连起来。

随着计算能力越来越容易获得，尤其是具有高品质图形功能的交互式计算的普及，为个人计算机编制软件成为一项重要的产业。此类软件多数对用户很友好，也就是说用户无需掌握太多的计算机知识，而且基本不用怎么学习便能够使用这些软件，这与先前的OS/360　完全不同，OS/360的作业控制语言JCL非常复杂，为了介绍这种语言，已经专门编写了几本书。

在个人计算机和工作站领域有两种主流操作系统：微软的MS－DOS和UNIX。MS－DOS广泛用于IBM PC及其他采用Intel 8OX86芯片的计算机。尽管MS－DOS的最初版本相当简陋，但其后续的版本包含了许多新特性，其中有许多源自于UNIX。MS－DOS的后续产品－Windows起初运行于MS－DOS之上（与其称之为操作系统，不如说它更象一个shell），但从1995年开始发布的Windows95是一个真正可引导的操作系统，因此它不再需要MS－DOS的支持。微软的另一个操作系统是Windows NT，它在某些层次上与Windows95兼容，但其核心则完全重写。

另一种主要的操作系统是UNIX，它在工作站和高档计算机领域（如网络服务器）占据了统治地位，尤其对于采用高性能RISC芯片的计算机。尽管这些计算机通常供一个用户专用，但它们往往具有小型机的计算能力，

所以为它们配备最初为小型机设计的操作系统 — UNIX是很顺理成章的。

从80年代中期开始出现一种有趣的发展趋势，就是运行网络操作系统（network operating systems）和分布式操作系统（distributed operating systems）（Tenenbaum, 1995）的个人计算机网络的崛起。在网络操作系统中，用户知道多台计算机的存在。他能够登录到一台远地机器上并将文件从一台机器拷贝到另一台机器，每台计算机都运行自己本地的操作系统，有自己的本地用户（或多个用户）。

网络操作系统与单处理机的操作系统没有本质区别。它们需要一个网络接口控制器以及一些低层软件来驱动它 ，同时还需要一些程序来进行远程登录和远地文件访问，但这些附加物并未改变操作系统的本质结构。

与之相反，一个分布式操作系统在用户看来就象一个普通的单处理机系统。尽管它实际上由多个处理机组成，但用户不会感知到他们的程序在哪个处理机上运行，或者他们的文件存放在哪里，所有这些均由操作系统自行高效地完成。

真正的分布式操作系统不仅仅是在单机操作系统上增添一小段代码，其原因是分布式系统与集中式系统有本质的区别。例如，分布式系统通常允许一个应用在多台处理器上同时运行，因此需要更复杂的处理器调度算法来获得最大的并行度。

网络中的通信延迟往往导致分布式算法必须能适应信息不完备、信息过时甚至信息不正确的环境。这与单机系统完全不同，对于后者，操作系统掌握整个系统的完备信息。

1.2.5 MINIX 的历史

在UNIX的早期（版本6），源代码可以免费获得并被人们加以广泛的研究。澳大利亚新南威尔士大学的John Lions甚至专门写了一本小册子逐行地解释UNIX源代码（Lions, 1976）。许多大学的操作系统课程就采用这本小册子作为教材。

在AT&T发布版本7时，它开始认识到UNIX的商业价值，于是发布的版本7许可证禁止在课程中研究其源代码以免其商业利益受到损害。许多学校为了遵守该规定，就在课程中略去UNIX的内容而只讲操作系统理论。

不幸的是，只讲理论使学生对实际的操作系统产生一种片面的认识。书本上作为重点讲述的内容，如进程调度算法，实际中并没有那么重要；而实际系统中很重要的内容，如I/O系统和文件系统又因为缺乏理论性而被忽略。

为了扭转这种局面，本书的作者之一坦尼鲍姆决定编写一个在用户看来与UNIX完全兼容，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，它可以被班级和个人用来学习。通过它读者可以剖析一个操作系统，研究其内部如何运作。其名称源于“小UNIX”，因为它非常简洁，一般程度的读者就能够读懂它。

除合法性以外，MINIX与UNIX相比还有另一个优点：它比UNIX晚出现十年，并且其代码采用了一种更加模块化的组织方法。例如，MINIX的文件系统不是操作系统的一部分而是作为一个用户程序运行。另一个不同之处在于UNIX着重效率而MINIX着重可读性（数百页的程序通常认为是可读的），其代码中有数千行注释。

MINIX最初设计成与UNIX版本7兼容。UNIX版本7由于其简洁和优美而被奉为典范，有人甚至认为它不仅超过了其前边的所有版本，而且超过了其后边的所有版本。随着POSIX的出现，MINIX在保持既有的向后兼容性的同时开始向POSIX靠拢。这种渐进的演变在计算机界相当普遍，没有一家厂商希望在引进新系统的同时使客户所有原先的程序作废。本书中介绍的MINIX基于POSIX标准（本书前一版本的MINIX基于UNIX版本7）。

和UNIX一样，MINIX用C语言编写，从而很容易被移植到其他机器上。由于IBM PC机广泛得到使用，所以其最初版本在IBM PC上实现，随后被移植到了Atari、Amiga、 Macintosh和SPARC等平台上。MINIX一直恪守“small is beautiful”的原则，因此最早的版本甚至不需要硬盘就可以运行，这使许多学生都能够达到其硬件要求（这种情况在现在看起来很奇怪，可是在80年代中期MINIX刚出现的时候，硬盘非常昂贵）。随着其功能和规模的增长，MINIX也发展到需要一个硬盘才能运行，但它只要求一个30M字节的分区即可。与之相比，有些商用UNIX系统建议提供多达200M字节的磁盘分区。

对于多数IBM PC用户来说，MINIX运行起来很象UNIX。许多基本程序，如cat、 grep、 ls、make及shell程序都与UNIX中的对应程序有相同的功能。与操作系统本身一样，这些公用程序均由作者和他的学生以及其他人员从头重写。

本书自始至终以MINIX作为例子，然而MINIX的大部分内容，除了代码本身外，均适用于UNIX。其中的许多内容还适用于一些其他的操作系统，在阅读本书时应记住这一点。

有些读者可能对MINIX和LINUX之间的关系感兴趣。在MINIX发布后不久，便出现了一个面向它的USENET新闻组，在数周之内便有多达40000个用户订阅该新闻组。其中的大多数人都想向MINIX中加入一些新特性以使之更大、更有用。每天都有数百人提供自己的建议、思想甚至代码。而MINIX的作者在几年内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，以便于学生理解。人们最终意识到不可能动摇作者的立场，于是一个芬兰学生Linus Torvalds决定编写一个类似MINIX的系统，但是它特征繁多、面向实用而非教学，这就是LINUX。

1.3 操作系统基本概念

操作系统与用户程序的界面由操作系统提供的“扩展指令”集定义。尽管有多种不同的实现方法，这些扩展指令传统上称作“系统调用”。为了真正地理解操作系统的运作机制，有必要仔细研究这个界面。各个操作系统提供的系统调用各不相同（尽管基本概念大致相同）。

本书在讲法上有两种选择：一种是泛泛而笼统地介绍系统调用（如：操作系统有读文件的系统调用）；另一种是选择一个确定的系统，讲述该系统的系统调用（如：MINIX有一条READ系统调用，它有三个参数：一个指定所操作的文件，一个指定将读到的数据存放在何处，最后一个指定读多少字节）。

本书采用后者，这样可以更细致地观察操作系统的内部操作。在1.4节中将详细地介绍UNIX和MINIX都提供的系统调用。为简洁起见，我们只讲述MINIX，多数情况下UNIX系统中对应的系统调用基于POSIX标准。下面先简要地描述一下MINIX以获得一些感性认识，这些内容同样适用于UNIX系统。

MINIX系统调用大致分为两类：与进程有关的系统调用和与文件有关的系统调用。

1.3.1 进程

在MINIX及所有操作系统中，一个重要概念是进程（process）。一个进程本质上是一个程序的执行。每个进程有其自己的地址空间，从0到一个最大值，进程可以读写该空间中的内容。地址空间中包括可执行程序、程序的数据及堆栈。与每个进程相关的还包括一组寄存器、程序计数器、指针和其他硬件寄存器，以及所有其他运行该程序所需要的信息。

进程的概念将在第二章详细讨论，此处希望读者对进程建立一种直观的感觉。为此我们考虑分时系统的工作：分时操作系统周期性地挂起一个进程然后启动运行另一个进程，例如在过去的一秒钟内，第一个进程已运行完分配给它的时间片，则要被暂停。

在上述进程被暂时挂起的情况下，之后的某个时刻它将被再次启动，而且要求再次启动时的状态与先前暂停时完全相同，这就意味着在将其挂起时该进程的所有信息都要被保存下来。例如进程以读方式打开了若干文件，每一个文件都有一个指针来指示当前的读写位置。当该进程被挂起时，这些指针均要被记录下来以保证后面的READ系统调用能够正确地执行。在许多操作系统中，除进程地址空间以外的所有信息均存放在操作系统管理的一张表中，称为进程表。进程表是一个数据结构的数组，当前系统中的每个进程都要占用其中一项。

所以一个（挂起的）进程包括两部分：进程的地址空间 — 称作核心映像（core image），以及对应的进程表项（包含寄存器值及其他信息）。

在与进程管理有关的系统调用中最关键的是完成进程创建和进程终止的系统调用。考虑一个典型的例子。一个命令解释器（shell）进程从终端上读命令，此时，用户刚键入一条命令要求编译一个程序，为此shell必须首先创建一个进程来执行编译程序。当执行编译的进程结束时，它执行一条系统调用来终止自己。

若一个进程能够创建一个或多个进程（称为子进程），而且这些子进程又可以创建子进程，则很容易得到如图1－5所示的进程树。为某项任务而合作的这些相关进程需要通信以同步其操作，这种通信称为进程间通信（interprocess communication），第二章将对此进行详细讨论。

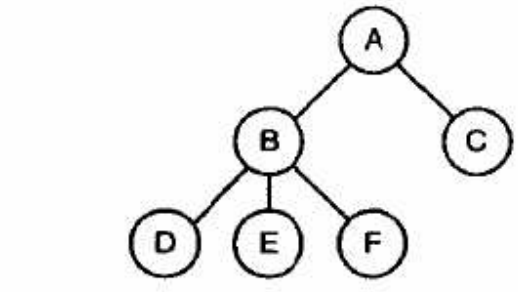

图 1-5 一棵进程树，进程A创建两个子进程B和C，进程B创建子进程D、E和F。

其他系统调用的功能包括：申请更多的内存（或释放不再需要的内存）、等待一个子进程结束、加载并执行另一个程序等。

在某些情况下，需要向一个运行进程传送信息，而该进程并不在信息发送处附近等待接收信息。例如分别在两台机器上的两个进程通过在网络上发送消息进行通信。为了保证一条消息或消息的应答不会丢失，发送者

要求它所在的操作系统在指定的若干秒后给它一个通知，这样如果它尚未收到确认消息就可以进行重发。在设定该时钟后，程序可以继续做其他工作。

经过了指定的秒数后，操作系统向这个进程发送一个信号。接收到信号将导致该进程暂时挂起，将寄存器值保存到堆栈中，并启动一个特殊的信号处理过程，例如重传一条丢失的消息。信号处理程序结束后，进程从收到信号的断点处继续执行。信号是对硬件中断的软件模拟，除了超时以外，还有很多其他原因可以导致产生信号。许多类型的陷入是由硬件检测的，如执行非法指令或非法访问地址，此类操作都将导致向违例进程发送相应的信号。

MINIX中的每一个合法用户都有一个由系统管理员分配的用户标识（uid），MINIX中的每一个进程都记录有启动它的用户的uid。子进程的uid与其父亲相同。系统中有一个特殊的用户 — 超级用户，他拥有特殊的权力，许多保护规则对超级用户无效。在大型系统中，只有系统管理员知道超级用户的口令。但很多普通用户，尤其是学生，总是费尽心机地寻找系统的安全性破绽以使自己无需该口令便成为超级用户。

1.3.2 文件

另一类系统调用面向文件系统。正如前面提到的，操作系统的主要功能之一是屏蔽硬件设备的特殊细节以便为程序员提供一个简洁方便的与设备无关的文件模型。显然，一个操作系统需要有相应的系统调用来创建文件、删除文件、读文件和写文件。在读一个文件之前首先要打开它，在读完之后要关闭它，完成此类功能的系统调用都是存在的。

文件的存放通过目录完成，一个目录下可以存放一组文件。例如一个学生可能给他所选的各门课程创建一个目录，另外设立一个目录存放电子邮件，再一个目录存放其WWW主页。这样就需要用到创建和删除目录的系统调用，以及将一个已经存在的文件放在一个目录中，从一个目录中删除一个文件等。目录项可以是文件或者其他目录，该模型就产生了层次型文件系统，如图1－6所示。

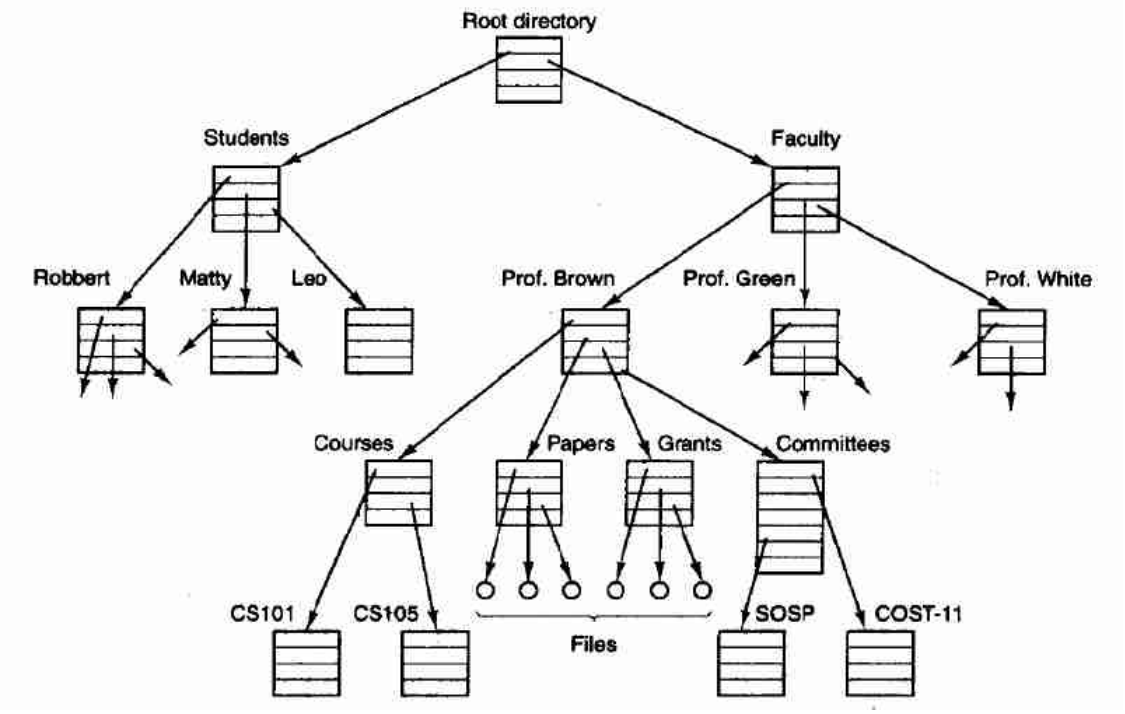

图 1-6　大学里一个系的文件系统。

进程和文件都可以组织成树状结构，但有许多不同之处。进程树的层次一般都不会很深（很少超过三层），而文件层次常多达四层、五层或更多。进程树的层次结构是暂时性的，通常最多存在几分钟，而目录层次则可能长达数年之久。进程和文件在属主及保护方面也是有区别的。典型地，只有父进程可以控制和访问子进程，而对于文件和目录则通常存在一种机制使属主以外的其他用户也可以访问该文件。

目录层次结构中的每一个文件都可以用一个从根目录开始的路径名来确定，这种绝对路径名中包含了从根

目录到该文件的所有中间目录，相互之间由正斜杠隔开。在图1－6中，文件CS101的路径名是/Facaulty/Prof.Brown/Courses/CS101。起始的正斜杠表示这是一个从根目录起始的绝对路径。

进程在任一时刻都有一个当前工作目录，非正斜杠起始的路径名均在此目录开始搜索。例如在图1－6中，若当前工作目录为/Facaulty/Prof.Brown，则使用路径名Courses/CS101与/Facaulty/Prof.Brown/Courses/CS101等效。进程可以通过系统调用改变当前工作目录。

MINIX中的文件和目录通过一个9比特的保护码来进行保护。保护码分成三个3比特的域，分别对应着文件主、同组用户和其他用户。每个域有一位标识读权限，一位标识写权限，一位标识执行权限。如保护码rwxr-x--x表示：文件主可以读、写、执行；同组用户可以读和执行，不能写；其他用户只能执行，不能读写。对目录来说，x表示搜索权限，短横表示不具备相应权限。

在文件读写之前，首先要将其打开，执行打开操作时将检查其访问权限，若访问权限许可，系统将返回一个小的整数，称作文件描述符，供后续操作使用；若访问权限不够则返回一个错误码。

MINIX中另一个重要概念是可安装文件系统。多数个人计算机都配有一个或多个软盘驱动器，为了对此类可移动介质（还包括CD－ROM）提供一种简洁的访问方式，MINIX允许将软盘上的文件系统链接到主文件树上。在图1－7（a）所示的情形中，在调用MOUNT系统调用之前，RAM盘（主存中的一个模拟盘）包含最基本的根文件系统，软驱0的软盘中则包含了另一个文件系统。

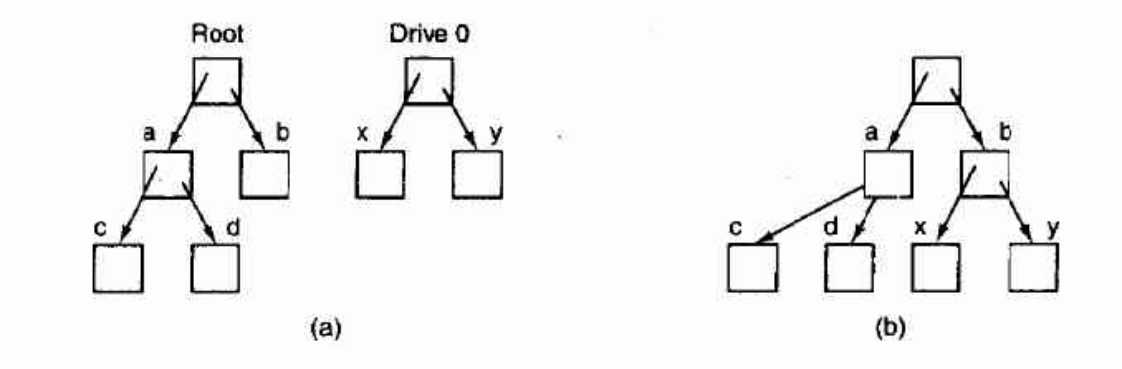

图 1-7 (a)在安装前，驱动器0中的文件不可访问。 (b)安装后，驱动器0中的文件成为文件树的一部分。

然而，软驱0中的文件系统无法访问，因为无法确定其上所存文件的路径名。MINIX不允许用驱动器名或数字作前缀表示的路径名，这种表示方式是设备相关的，操作系统应尽量避免。在MINIX中，使用MOUNT系统调用便可以将软驱0中的文件系统链接到根文件系统的任一目录。在图1－7（b）中软驱上的文件系统被安装到目录b下，这样就可以访问文件/b/x和/b/y。如果目录b下原先存有文件，则在软驱0被安装期间这些文件无法访问，因为目录“/b”现在指向软驱0的根目录。（这种情况通常不会造成很多不便，其原因是文件系统总是被安装在空目录下）

MINIX中另一个重要概念是设备文件（special file）。提供设备文件的目的是使I/O设备使用起来更类似于文件。在这种方式下，设备文件的读写可以使用与普通文件相同的系统调用。设备文件分为两类：块设备文件（block special files）和字符设备文件（character special files）。块设备文件指那些由可以随机存取的数据块组成的设备，如磁盘。打开一个块设备文件，然后读第四个块，程序可以直接访问设备上的第四块而不管其文件系统的格式。类似的，字符设备文件指那些以字符流方式进行操作的设备，如打印机，调制解调器等。

最后了解一下与进程和文件都相关的管道（pipe）。管道是一种用来连接两个进程的虚拟文件，如图1－8所示。当进程A欲向进程B发送数据时，它把管道文件视作输出文件，向其中写数据，进程B则可将管道文件视作输入文件，从中读数据。于是，MINIX中的进程间通信很象普通文件的读写。一个进程判断其输出是普通文件还是管道的唯一方法是调用一条特殊的系统调用。

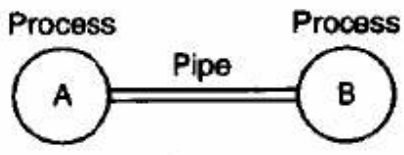

图 1-8 一个管道连接两个进程。

1.3.3 外壳（shell）

MINIX操作系统是指完成系统调用的代码，编辑器，编译器，汇编程序，链接程序以及命令解释器等均不是操作系统的组成部分。下面大概介绍一下MINIX的命令解释器：shell。尽管它本身不是操作系统的一部分，但它体现了操作系统的许多特性并很好地说明了系统调用的具体用法，同时它也是终端用户与操作系统之间的界面。

当用户登录进入系统时，同时将启动一个shell，它以终端作为标准输入和标准输出，它首先显示系统提示符，这通常是一个美元符号，它提示用户shell正在等待接收命令。若用户键入

```
date
```

则shell创建一个子进程并使其运行date程序。在该子进程运行期间，shell等待它结束。当子进程结束后，shell再次显示系统提示符并等待下一个输入。

用户可以将标准输出重定向到一个文件，如：

```
date > file
```

同样地，也可以将标准输入重定向，如：

```
sort < file1 > file2
```

该命令将调用sort程序，从file1中取输入，并将输出送到file2。

通过管道可以将一个程序的输出作为另一个程序的输入，因此

```
cat file1 file2 file3 | sort > /dev/lp
```

将调用cat程序将这三个文件合并，结果送到sort程序按字典序排序，sort的输出又重定向到文件/dev/lp，这正是打印机的设备文件名（一般将所有设备文件都放在/dev/目录下）。

如果在一条命令后加上一个“&”符号，则shell将不等待其结束而直接显示出系统提示符。所以

```
cat file1 file2 file3 | sort > /dev/lp&
```

将启动sort程序作为后台任务执行，这样就可以在本条命令尚未结束时允许用户继续下面的工作。shell还有许多其他有用的特性，由于篇幅所限，不能一一讨论，具体请参阅有关shell的参考书。

1.4 系统调用

现在开始讨论操作系统与应用程序之间的接口－系统调用。尽管此处的讨论基于POSIX标准（国际标准9945－1）和MINIX，但当前多数其他操作系统都有完成相同功能的系统调用，至多在细节上不尽相同。由于系统调用的实现往往与机器有关，而且总是用汇编语言表述，所以为使C程序能够使用系统调用，必须额外提供一个例程库。

为了清楚地了解系统调用的内在机制，我们以READ系统调用为例子。它有三个参数：第一个指定所操作的文件，第二个指定使用的缓冲区，第三个指定要读的字节数。在C程序中调用该系统调用的方法如下：

```
count = read(file, buffer, nbytes);
```

本系统调用将真正读到的字节数返回给count变量，正常情况下这个值与nbytes相等，但当读至文件结尾符时则可能比nbytes小。

若由于参数非法或磁盘操作错导致该系统调用无法执行，则count被置为-1，同时错误码被放在一个全局变量errno中。程序应该经常检查系统调用的返回值以确定其是否正确地执行。

MINIX的全部53条系统调用列于图1－9中，这些系统调用分为六大类。下面将逐个解释每条系统调用的功能。由于个人计算机的资源管理功能非常有限（起码和有许多用户的大型机相比是如此），所以这些系统调用在很大程度上定义了操作系统应提供的主要功能。POSIX标准指定了许多过程，但并未明确规定它们以什么形式出现，是系统调用、库函数、或其他别的形式。在有些情况下，POSIX的过程在MINIX中以库例程形式出现，在其他情况下，鉴于几个例程仅有细微的差别，则将它们都包括在一条系统调用中。

进程管理

```
pid = fork()
pid = waitpid(pid, &statloc, opts)
s = wait(&status)
s = execve(name, argv, envp)
exit(status)
size = brk(addr)
pid = getpid()
```

pid = getpgrp()
pid = setsid()
l = ptrace(req, pid, addr, data)
创建一个与父进程相似的进程
等待一个子进程结束
waitpid的老版本
替换一个进程的核心映像
终止进程的执行并返回状态
设置数据段大小
返回调用进程的标识号
返回调用进程的组号
创建一个新的会话并返回其组标识
用于调试
信号
s = sigaction(sig, &act, &oldact)
s = sigreturn(&context)
s = sigprocmask(how, &set, &old)
s = sigpending(set)
s = kill(pid, sig)
residual = alarm(setconds)
s = pause()
定义对信号的处理操作
从信号返回
检查或修改信号屏蔽码
获得阻塞信号集合
替换信号屏蔽码并使进程挂起
设置时间闹钟
将调用进程挂起直到下一个信号
文件管理
fd = creat(name, mode)
fd = mknod(name, mode, addr)
fd = open(file, how, ...)
s = close(fd)
n = read(fd, buffer, nbytes)
n = write(fd, buffer, nbytes)
pos = lseek(fd, offset, whence)
s = stat(name, &buf)
s = fstat(fd, &buf)
fd = dup(fd)
s = pipe(&fd[0])
s = ioctl(fd, request, argp)
s = access(name, amode)
s = rename(old, new)
s = fcntl(fd, cmd, ...)
创建一个文件
创建普通、设备文件或目录i-节点
打开一个文件进行读、写或读写
关闭一个打开文件
从一个文件读数据到一个缓冲区
从缓冲区将数据写入文件
移动文件指针
获取一个文件的状态信息
获取一个文件的状态信息
为打开文件分配一个新文件描述符
创建一个管道文件

对文件进行特殊操作
检查文件是否可访问
文件改名
文件加锁及其他操作
目录及文件系统管理
s = mkdir(name, mode)
s = rmdir(name)
s = link(name1, name2)
s = unlink(special, name, flag)
s = mount(special, name, flag)
s = umount(special)
s = sync()
s = chdir(dirname)
s = chroot(dirname)
创建一个新目录
删除一个空目录
创建一个新文件name2指向name1
删除一个目录项
安装一个文件系统
卸装一个文件系统
将缓冲的数据块回写到磁盘
改变工作目录
改变根目录
保护
s = chmod(name, mode)
uid = getuid()
gid = getgid()
s = setuid(uid)
s = setgid(gid)
s = chown(name, owner, group)
oldmask = umask(complmode)
改变文件的保护位
获取调用进程的uid
获取调用进程的gid
设置调用进程的uid
设置调用进程的gid
改变文件的属主和组
改变模式屏蔽码
时间管理
seconds = time(&seconds)
s = stime(tp)
s = utime(file, timep)
s = times(buffer)
获取从1970年1月1号以来的时间
设置从1970年1月1号以来的时间
设置文件的最后访问时间
获取到当前所用的用户和系统时间

图 1-9 MINIX的系统调用，返回值s在出错时为 －1， fd为文件描述符，n为字节计数。其他返回码由其字面意思确定。

1.4.1 进程管理系统调用

第一类系统调用用于进程管理。先看fork系统调用，fork是创建进程的唯一途径。它实际上是做一个调用

它的进程的精确拷贝，包括文件描述符，寄存器值等所有内容。调用fork后，原进程和所得的拷贝各自执行，互不相关。在执行fork时，二者所有的对应变量都有相同的值，但由于子进程在创建过程中对父进程的数据作了一个拷贝，所以在此之后，其中任一进程中变量值的改变不会对另一个进程产生任何影响（正文段不可修改，它由父、子进程共享）。正确情况下fork的返回值对子进程为0，对父进程为一个正整数，即子进程的标识号pid。通过该返回值可以将父子进程区分开来。

多数情况下，执行完fork后，子进程需要执行与父进程不同的代码，例如对于一个shell，它首先从终端读取命令，然后创建一个子进程来执行该命令，等待子进程执行完毕，然后再读取下一条命令。为了等待子进程结束，父进程执行一条waitpid系统调用。该系统调用使调用进程阻塞直到子进程中的任一个结束（若将其第一个参数置为－1），也可以等待一指定的子进程结束。waitpid结束时，子进程的终止状态值（正常结束或异常结束，以及正常结束时的返回值）被放在第二个参数指向的地址单元。waitpid有若干选择项可用。waitpid 取代了先前的wait系统调用，wait系统调用虽然已经过时，但目前仍旧提供，其目的是为了保持兼容性。

现在来看shell如何使用fork。当键入一条命令时，shell首先创建一个新进程。该子进程需执行该用户命令，这通过调用exec系统调用实现，exec用其第一个参数指定的可执行文件替换其核心映像，一个高度简化的shell框架如图1－10所示。

```
while (TRUE) {                                  /* 无限循环 */
    read_command(command, parameters);          /* 从终端读取输入 */

    if ( fork() != 0 ) {                        /* 创建子进程 */
        /* Parent code */
        waitpid( -1, &status, 0);               /* 等待子进程退出 */
    } else {
        /* Child code */
        execve(command, parameters, 0);         /* 执行命令 */
    }
}
```

图 1-10 一个简化的shell，在本书中，TRUE被定义为 1。

一般情况下，exec有三个参数：待执行的文件名，指向参数数组的指针和指向环境变量数组的指针。系统提供了若干库例程来简化这些参数的使用，包括execl，execv，execle和execve。本书采用exec来泛指所有这些系统调用。

对于如下一条命令

```
cp file1 file2
```

其功能是为文件file1作一个拷贝file2，在shell创建一个子进程后，子进程执行程序cp，同时向该程序传递执行的参数：源文件名和目标文件名。

cp程序的主函数格式如下：

```
main( argc,  argv,  envp)
```

这里argc是命令行中包括程序名在内的参数个数。对于以上例子，argc为3。

第二个参数argv 是一个指向数组的指针。该数组中第i个因素就是命令行中第i个字符串，此处argv[0]为：“cp”，argv[1]为“file1”，argv[2]为“file2”。

第三个参数envp是一个环境变量指针。环境是由一系列形如“name = value”的字符串组成的，用来将环境信息诸如终端类型，用户主目录等传递给程序。在图1－10中，没有向子进程传递任何环境变量，因此execve的第三个参数为空。

EXEC看起来很复杂，但是你不必为此对其他系统调用也感到担忧，实际上EXEC是最复杂的系统调用，其他系统调用都比它简单得多，例如EXIT系统调用的作用是结束一个进程，它只有一个参数指定其出口值（0~255）。这个值通过wait和waitpid系统调用中的变量status，返回给父进程。status的低字节存放结束状态，0为正常结束，其他值均表示出错。status的高字节包含子进程的出口值（0~255）。例如若父进程执行：

```
n = waitpid( -1, &status, options);
```

则父进程被阻塞直至有一个子进程结束。如果子进程结束时将4作为exit的参数值，则父进程唤醒时n被置为该结束子进程的pid,而status的值为0x0400。

MINIX中进程的存储空间分为三部分：正文段（text segment，即程序代码），数据段（data segment，即变量）和堆栈段（stack segment）。数据段是向上增长而堆栈向下增长，如图1－11所示。在两者之间是空闲区。堆栈的增长随程序的执行自动进行，而数据段的扩展则通过BRK系统调用显式地完成，BRK有一个参数指定数据段的结束地址，它可以比当前值大（数据段扩展），或比当前值小（数据段缩小）。该参数必须小于堆栈指针，否则堆栈和数据段将重叠，这是不允许的。

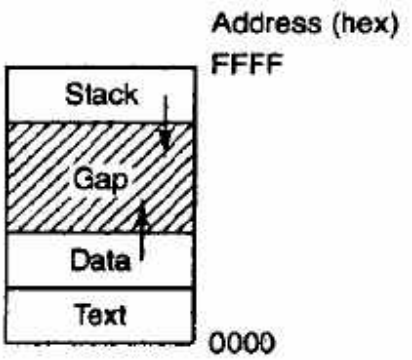

图 1-11 进程有三个段：正文段、数据段和堆栈段。在本例中，所有三个段在同一个地址空间中，但是也支持分离的指令空间和数据空间。

出于为程序员的方便考虑，系统提供了一个库函数sbrk来改变数据段大小，它的参数是数据段的变化量（负数表示数据段缩小）。其原理是跟踪数据段的当前值，（该值由BRK返回）。计算其新的大小，然后申请所需的空间。BRK和sbrk均与实现密切相关，所以不是POSIX的组成部分。

另一个简单的系统调用getpid返回调用进程的进程标识号，注意在调用fork时，只有父进程能够获得子进程的进程标识号。如果子进程要得到自己的进程标识号，它必须使用getpid。同样地，getgrp返回进程的组标识号。SETSID系统调用启动一个新的会话（session），并将进程组的pid设为调用者的pid。会话与POSIX的可选特性－作业控制有关，MINIX不支持作业控制，所以对SETSID不作进一步的讨论。

最后一个系统调用是ptrace，它被调试器用来对被调试程序进行控制，通过ptrace，调试器可以读写被控进程的地址空间并实施其他控制。

1.4.2 信号管理系统调用

尽管大多数进程间通信是计划好的，但同时还需要处理不可预知的通信问题，例如用户使用文本编辑器要求列出一个大文件的全部内容，但随即他认识到该操作并不需要，这时就需要一种方法来中止编辑器的工作。在MINIX中，用户可以通过DEL键作到这一点，按DEL键实际上是向编辑器发送一个信号，编辑器收到此信号即停止打印文件的内容。信号还可用来报告硬件捕获到的特定的陷入，如非法指令或浮点运算溢出，超时也通过信号实现。

当一个信号被发送给一个事先并未声明愿意接收它的进程时，该进程只是简单地被撤销，即被杀死。进程可以用SIGACTION系统调用来声明它准备接收某些类型的信号，并同时提供两个地址：一个是信号处理过程的地址，另一个用于保存该信号的原先处理过程的地址。执行完SIGACTION系统调用后，此进程若接收到相关类型的信号，则先将该进程的当前状态压栈，然后调用指定的信号处理过程。信号处理过程可能很长，它本身也可以调用系统调用，但一般情况下，它往往很短。信号处理过程结束后，它调用SIGRETURN以继续执行被该信号中断的操作，类似于硬件中的中断返回。SIGACTION替换了原先的SIGNAL系统调用，出于兼容性的考虑，SIGNAL仍然提供。

在MINIX中信号可以被阻塞，被阻塞的信号一直被挂起，直到阻塞解除。这段时间内它不被传递，但也不会丢失。SIGPROCMASK系统调用允许一个进程定义其阻塞的信号集，其实现方法是向核心提交一张位图。进程也可以查询当前因阻塞而挂起的信号集，为此可使用系统调用SIGPENDING，它以位图方式返回该信息。最后，SIGSUSPEND系统调用使进程可以原子性地设定一张阻塞信号位图并将其挂起。

除了提供一个函数以捕获一个信号（也就是在接收到该信号时插入执行此函数）外，程序也可以使用常数SIG_IGN来忽略指定类型的信号，或者使用SIG_DFL来恢复缺省的信号处理过程。缺省的处理方式随信号而异，可以是撤销该进程，或者是忽略该信号。为了说明SIG_IGN的用法，请看以下命令，其功能是让shell创建一个后台进程。

```
command &
```

这里希望DEL信号不要对后台进程产生影响，所以shell在执行fork之后，exec之前需要执行

```
sigaction (SIGINT, SIG_IGN, NULL);
```

及

```
sigaction (SIGQUIT, SIG_IGN, NULL);
```

来忽略DEL和quit信号。（quit信号由CTRL-\产生，它与DEL信号的作用基本相同，不同之处在于如果它未被捕获或忽略，则它将产生被撤销进程的核心映像文件）对于前台进程（命令不带&），这些信号不能被忽略。

按DEL键并不是发送信号的唯一途径，使用KILL系统调用可以向另一个进程发送信号（前提条件是两进程有相同的用户标识号，也即无关的进程间不能发送信号）。再看上一个例子，假设一个后台进程已被启动，但随后发现它应被终止，此时SIGINT和SIGQUIT都已被屏蔽，所以只能使用KILL来向它发送一个信号。向后台进程发送信号9（SIGKILL）将撤销该进程，SIGKILL不能被捕获或忽略。

对于许多实时应用，需要在一段指定时间后，中断进程的原有操作，以进行某种其他处理，例如在不可靠的通信线路上重传一个丢失的包。为了处理此类情况，系统提供了ALARM系统调用，ALARM的参数指定一个以秒为单位的时间间隔，一旦该时间段到点就向该进程发送一个SIGALRM信号。在任意时刻一个进程只能设定一个时间闹钟。如果进程先设定一个10秒的时间闹钟，在3秒后又设定一个20秒的时间闹钟，则只有其中一个有效，即在第二个ALARM调用之后20秒会发送一个SIGALRM信号。如果ALARM的参数为0，则所有挂起的SIGALRM信号都被取消。若不捕获SIGALRM信号，则其缺省的处理方法是撤销该进程。

某些情况下进程在信号到达之前不要做任何操作，例如一个测验阅读和理解速度的CAI系统，它先在屏幕上显示一些课文，然后调用ALARM设定在30秒后向自己发送一个信号，以激活程序进行一些处理。当学生阅读课文时，程序无需执行任何操作。它可以采用的一种方法是执行空操作循环以等待时间到，但假如此时系统中还有其他进程运行，这将浪费CPU时间，好的方法是使用PAUSE系统调用，它将挂起调用进程直至信号到来，这段时间里别的进程便可以使用CPU。

1.4.3 文件管理系统调用

许多系统调用与文件系统有关，这里仅讨论对单个文件进行操作的系统调用，下一节将讨论对目录和文件系统进行操作的系统调用。创建新文件要使用CREATE系统调用，其参数指定文件路径名和保护模式，所以

```
fd = creat ("abc", 0751);
```

将创建一个名为abc的文件，其保护模式为0751（在C语言中，开头的0表示一个常数为八进制），它的低9位指明文件主（7表示可读、写、执行），同组用户（5表示可读、执行）及其他用户（1表示只可执行）的操作权限。

CREAT在创建文件的同时还以写方式将其打开，而不管文件模式是什么。CREAT返回的文件描述符fd可用于对该文件执行写操作。如果对一个已经存在的文件进行CREAT操作，在操作权限许可的情况下该文件内容将被破坏，CREAT属于已经过时的系统调用，因为OPEN系统调用也可以创建新文件，但系统仍提供CREAT以保持兼容性。

创建设备文件使用MKNOD，而不是CREAT，典型的用法为：

```
fd = mknod("/dev/ttyc2", 020744, 0x0402);
```

这将创建一个名为“/dev/ttyc2”的文件（二号控制台通常用的文件名），并将其模式代码置为八进制的020744（意为该文件是字符设备文件，保护模式为rwxr--r--），第三个参数的高字节指定其主设备号为4，低字节指定其次设备号为2。主设备号可以取任何值，但名为/dev/ttyc2的文件次设备号应当为2。MKNOD只能被超级用户使用。

读写一个文件之前首先必须用OPEN系统调用将其打开。OPEN的第一个参数指定文件路径名，可使用绝对路径名或相对于当前工作目录的相对路径名；第二个参数指定打开方式O_RDONLY,O_WRONLY或O_RDWR（分别表示只读，只写和可读可写），OPEN返回的文件描述符可用于文件读写。文件在操作完毕后要用CLOSE关闭，这样该文件描述符可供其后的CREAT和OPEN系统调用再次使用。

最常用的系统调用当属READ和WRITE，下面将对READ进行详细讨论，WRITE与它具有相同的参数。

多数程序对文件的读写操作都是顺序进行的，但有些却需要随机地访问文件的任意部分。每个文件都有一个指针指明其当前读写位置。在顺序读写时，该指针通常指向下次要读写的字节。使用LSEEK系统调用可以直接修改文件指针的值，这样随后的READ或WRITE就可在文件的任一位置进行操作，甚至可以超越文件尾。

LSEEK有三个参数：第一个指定文件描述符，第二个指定文件的位置，第三个指明文件位置是相对于文件开头、当前位置、还是文件尾。LSEEK的返回值是文件指针被修改之后的绝对位置。

对每一个文件，MINIX记录了如下信息，包括文件类型（普通文件、设备文件以及目录等），文件大小，最后修改时间等等。程序可以通过STAT和FSTAT系统调用获取这些信息，其不同之处仅在于STAT通过文件名来指定文件，而FSTAT则使用文件描述符，这样FSTAT很适用于已打开的文件，尤其是标准输入和标准输出这类文件名可能不可知的情况。STAT和FSTAT的第二个参数指定一个用来存放所获取信息的数据结构，如图1－12所示。

```
struct stat {
    short     st_dev                /* i-节点所驻留的设备 */
    unsigned short st_ino;          /* i-节点号 */
```

```
        unsigned short st_mode;          /* 模式 */
        short st_nlink;                      /* 链接数 */
        short st_uid;                    /* 用户标识号 */
        short st_gid;                    /* 组标识号 */
        short st_rdev;                       /* 设备文件的主/次设备号 */
        long st_size;                    /* 文件大小 */
        long st_atime;                       /* 最后访问时间 */
        long st_mtime;                       /* 最后修改时间 */
        long st_ctime;                       /* 对i-节点最后修改时间 */
};
```

图 1-12　STAT和FSTAT系统调用所用的存放返回信息的数据结构。在实际代码中，其中的某些数据类型使用符号名。

DUP系统调用常用于对文件描述符的操作，例如一个程序需要关闭标准输出（文件描述符为1），代之以使一个普通文件成为标准输出，随后向标准输出写一些信息，最后恢复原先的状态。为实现这些功能可以先关闭文件描述符1，再打开另一个文件，这时该文件就成为标准输出（设标准输入正被使用），但这样处理无法恢复原先的标准输出。

解决办法是先调用

```
fd = dup(1);
```

该操作将为标准输出分配一个新的文件描述符fd，并使对fd的操作与直接对标准输出的操作完全一样。随后将标准输出关闭，再打开一个新文件，至此该文件将被作为标准输出。当需要恢复原先的标准输出时，先关闭文件描述符1，再执行

```
n = dup(fd);
```

将最小的文件描述符号，即1，定向到fd所指的文件，最后将fd关闭就恢复了最初状态。

DUP系统调用有一个变种，它允许将任一未使用的文件描述符定向到一个指定的打开文件，其调用方法为：

```
dup2 (fd, fd2);
```

此处fd指向一打开的文件，fd2为一个未使用的文件描述符，执行完这条语句后fd2将指向fd所指向的文件。若fd指向标准输入（文件描述符0），fd2为4，则在此调用后描述符0和4都指向标准输入。

如前所述，MINIX中的进程间通信使用管道，若一用户键入

```
cat file1 file2 | sort
```

shell将创建一个管道并将第一个进程的标准输出信息写到管道中，于是第二个进程的标准输入便可以从该管道中读取。PIPE系统调用创建一个管道并返回两个文件描述符，一个用于写，另一个用于读，PIPE的调用格式为：

```
pipe(&fd[0]);
```

这里fd是由两个整数组成的数组，fd[0]存放供读使用的文件描述符，fd[1]存放供写使用的文件描述符。通常典型的用法是在本条语句之后调用一个fork创建一个子进程，然后父进程关掉用于读的文件描述符，子进程关掉用于写的文件描述符（或者相反），这样便可以做到一个进程从管道读数据，另一个向管道写数据。

图1－13提供了一个框架过程，其中创建了两个进程，通过管道将进程1的输出导向进程2（更实际的例子还要进行错误检查，并作参数处理）。其处理过程如下：首先创建一个管道，随后调用fork，将父进程作为管道中的进程1，子进程作为进程2，由于待运行的两个文件process1和process2并不知道它们是管道的一部分，所以必须对文件描述符进行控制以使进程1的标准输出和进程2的标准输入都指向管道。父进程首先关掉从管道读的文件描述符和标准输出，随后执行DUP，这样就使文件描述符1可被用于向管道写。注意DUP总是返回最小的可用文件描述符，在这里即为1。然后程序关闭另一个管道文件描述符。

```
#define STD_INPUT  0                    /* 标准输入的文件描述符 */
#define STD_OUTPUT      1                   /* 标准输出的文件描述符 */

pipeline(process1, process2)
char *process1, *process2;                  /* 指向程序名的指针 */
{
    int  fd[2];
    pipe(&fd[0]);                           /* 创建一个管道 */
    if (fork() != 0) {
        /* 父进程执行如下语句 */
        close(fd[0]);                       /* 进程1不需要从管道读 */
        close(STD_OUTPUT);              /* 准备新的标准输出 */
        dup(fd[1]);                         /* 将标准输出指向fd[1] */
```

```
            close(fd[1]);                       /* 此文件描述符不再需要 */
            execl(process1, process1, 0);
        } else {
            /* 子进程执行如下语句 */
            close(fd[1]);                       /* 进程2不需要向管道写 */
            close(STD_INPUT);           /* 准备新的标准输入 */
            dup(fd[0]);                         /* 将标准输入指向fd[0] */
            close(fd[0]);                       /* 此文件描述符不再需要 */
            execl(process2, process2, 0);
        }
    }
```

图 1-13 建立一个两进程的管道的程序框架。

在EXEC系统调用之后，父进程将保留文件描述符0和2，而文件描述符1则用于向管道中写。子进程的代码与父进程类似。execl的参数被重复，因为其第一个参数是待执行的文件名，而第二个参数是执行文件的第一个参数，对多数程序来说，该参数都是文件名。

下一个系统调用IOCTL适用于所有设备文件，例如它可用于设备驱动程序，SCSI设备驱动程序用它来控制磁带机和CD－ROM。但其主要还是用于字符设备文件，尤其是终端。POSIX定义了许多函数，最终都转化为IOCTL调用。库函数tcgetattr和tcsetattr使用IOCTL来改变终端的模式和各种属性。

终端最常用的模式是Cooked模式，在该模式下删除键和终止键能正常地工作，CTRL_S和 CTRL_Q用来停止和恢复终端输出，CTRL_D为文件结束符，按DEL键产生一个中断信号，而CTRL-\产生一个退出信号并强制进行核心映像转储。

在raw模式下，所有这些功能都被取消，每个字符都被不加处理地送给程序，而且不等一行结束就将从终端读到每一个字符发送给程序。与此不同的是在Cooked模式下，终端输入的数据等到一行结束才送给程序。

Cbreak模式介于上述两者之间，用作编辑的删除键和终止键，以及CTRL_D被屏蔽，但CTRL_S,CTRL_Q,DEL和CTRL_\则仍然有效，与raw模式一样，单个字符不等一行结束就送给程序（如果禁止行内编辑功能，则没必要等待接收到完整的一行，因为用户不可能象在cooked模式下那样改变主意并删除它）。

POSIX并不采用以上所列的术语Cooked, raw 和 Cbreak，POSIX中的正规模式对应于Cooked模式。在正规模式中定义了11个特殊字符，输入也是以行为单位进行。在POSIX的非正规模式中，读数据操作由一个最小接受字符数和一个以0.1秒为单位的时间段决定。POSIX标准有具有很大的灵活性，其中有许多标志可以被设置，使得非正规模式使用起来很象Cooked模式和raw模式。原先的术语：Cooked模式、raw模式和Cbreak模式更具描述性，因此以后仍使用它们。

IOCTL有三个参数，例如调用tcsetattr函数设置终端参数最终将转换成以下调用：

```
ioctl (fd, TCSETS, &termios);
```

第一个参数指定一个文件，第二个参数指定操作类型，第三个参数指定一个POSIX数据结构的地址，其中包含了各种标志及控制字符的数组。除了TCSETS以外，还有一些其他的操作码，其功能包括：推迟对终端参数所作的修改直到全部输出被送出，将未读取的输入信息作废，返回参数的当前值等。

ACCESS系统调用检查对一个文件是否具有某种访问权限，因为有些程序可以用另一用户的用户标识号运行，所以ACCESS非常有用。SETUID机制将在稍后介绍。

RENAME系统调用将文件更名，其参数分别指定老文件名和新文件名。

FCNTL系统调用对文件进行控制，它有些类似IOCTL（这两条系统调用一般被熟练程序员使用）。FCNTL有若干选项，最常用的是文件加锁。使用FCNTL可以对一个文件的一部分加锁和解锁，也可以检测一个文件的某个部分是否被上锁。FCNTL本身不包含任何与锁操作有关的语义，这要由程序员自行定义。

1.4.4 目录管理系统调用

下面介绍对目录和文件系统进行操作的系统调用。首先是MKDIR和RMDIR，分别用来创建和删除空目录。LINK系统调用允许同一个文件按不同路径名出现，一种典型的应用是允许开发小组的几个成员共享一个文件，同时该文件出现在每个人自己的目录下。共享一个文件不同于给每个人一个私有的拷贝，对前者，任何一个人所作的修改都对其他人可见－只存在一个文件；而对后者，所作的修改只有自己可见，而不会更新其他人的拷贝。

图1－14的例子解释了LINK的工作原理，两个用户ast和jim，在他们各自的目录下都有一些文件，如果ast执行以下语句：

```
link ("/usr/jim/memo", "/usr/ast/note" );
```

则jim目录下的文件memo将以文件名note出现在ast的目录下，此后/usr/jim/memo和/usr/ast/note指的是同一个文件。

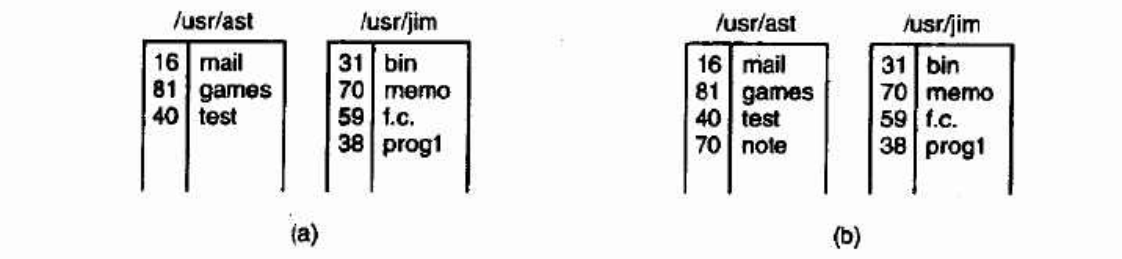

图 1-14 (a)将/usr/jim/memo链接到ast的目录之前的两个目录。 (b)同样两个目录在链接后。

理解LINK的工作原理有助于掌握其功能，MINIX中的每个文件都有一个唯一的数字 — i-节点（i-node）号来标识它。i-节点号是i-节点表的索引值，每个文件有一个i-节点，里面存放有文件主以及该文件所占用的磁盘块等信息。目录实际上也是文件，只是其内容存放的是一些i-节点和文件名的对应信息。在图1－14中，文件mail的i-节点号为16，其他文件都与此类似。LINK所做的只是创建一个新的目录项，它有一个新文件名，但i-节点号则是被链接的文件的i-节点号。图1－14（b）中，两个目录项有相同的i-节点号70，所以它们指向同一个文件。如果两个目录项的任何一个以后用UNLINK系统调用删除，另外一项还存在，则相关文件也继续存在；如果两项都被删除，那么MINIX检测到没有目录项指向该文件（i-节点中包含一个字段，它记录指向该文件的目录项数），则该文件被从磁盘上删除。

如前所述，MOUNT系统调用可将两个文件系统合并成一个。通常情况是先有一个存在于RAM盘上的根文件系统，其中包含有常用命令的可执行文件及其他常用文件。然后用户可以在驱动器0中插入一张存有用户程序的软盘。使用MOUNT系统调用就可以将软盘上的文件系统安装到根文件系统下，如图1－15所示。执行安装操作的典型C语句为：

```
mount ("/dev/fd0", "/mnt", 0);
```

其中第一个参数是软驱的设备文件名，第二个参数指定在文件树中的安装点。

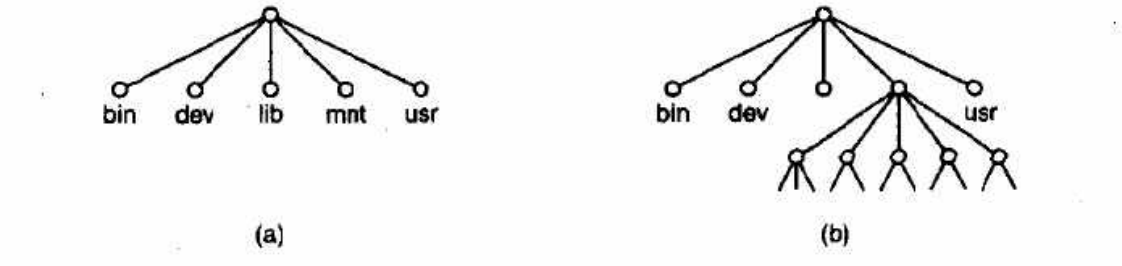

图 1-15 (a)安装前的文件系统。 (b)安装后的文件系统。

执行完MOUNT后，软驱上的文件即可通过路径名访问而与具体的物理设备无关，就是说软盘插入哪一个驱动器都是可以的。使用MOUNT命令，用户可以将可移动介质集成到单一的集成的文件层次结构中而不必关心文件在哪台具体设备上。上例中只涉及到软盘，实际上硬盘，CD－ROM和硬盘的分区（或称次设备）都可以这样安装，当一个文件系统不再需要时，可以用UMOUNT系统调用将其卸装。

MINIX在内存中开辟了一个缓冲区以保存最近经常访问的磁盘数据，这样可以避免再重复地从磁盘上读数据。如果缓冲区中的某数据块被修改，而在其被写回磁盘之前系统发生崩溃，那么文件系统可能被损坏。为避免这种现象，必须周期性地将这些缓冲区中的数据写回磁盘。系统调用SYNC用来将被修改的缓冲区数据写回磁盘。MINIX启动后，一个名为update的程序被启动作为后台进程运转，它每隔30秒执行一次SYNC操作，将更新了的数据写回磁盘。

与目录操作有关的另外两个系统调用是CHDIR和CHROOT，CHDIR改变当前工作目录，CHROOT改变根目录。若先调用

```
chdir ("/usr/ast/test");
```

然后打开文件xyz，则这将打开"/usr/ast/test/xyz"文件。CHROOT的功能与此类似，进程改变其根目录后所有绝对路径名都将从该新的根目录开始。只有超级用户可以执行CHROOT，而且即使超级用户也很少使用CHROOT。

1.4.5 权限管理系统调用

MINIX中每个文件都有一个包含11个比特的保护方式码，其中的9比特标识文件主，同组用户和其他用户的操作权限。CHMOD系统调用可以改变文件的保护方式，例如，要使一个文件对文件主之外的所有用户只读，可以调用：

```
chmod("file", 0644);
```

保护方式码的另外两个比特，02000和04000分别是SETGID位和SETUID位。用户在执行一个SETUID位设置的程序时，在进程运行期间它的有效用户标识被设为文件主的用户标识号uid。该特性常被用来使一般用户可以执行那些通常只有超级用户才能执行的功能。例如只有超级用户才能使用MKNOD来创建设备文件，将MKNOD程序的属主置为超级用户，同时将其保护码设为04755就可以使一般用户能够执行MKNOD，而同时又对一般用户施以严格的限制。

当进程执行一个SETUID或SETGID位设置的文件时，它便获得了与其真实uid或gid不同的有效uid或gid。有时进程需要获得其真实和有效的uid和gid，　　MINIX为此提供了GETUID和GETGID系统调用以获取这些信息。GETUID同时返回真实uid和有效uid，GETGID同时返回真实gid和有效gid。为此，系统提供了四个库例程来获取相应的信息，分别是：getuid, getgid, geteuid, getegid。

一般用户不能改变他们的uid，除非执行一个SETUID的程序。但超级用户则拥有特别的权力：他可以使用SETUID系统调用设置自己的有效和真实uid，也可以使用SETGID设置有效和真实gid，他还可以使用CHOWN改变文件的属主。总之超级用户不受系统保护规则的约束。

在权限管理方面，还有两条系统调用可以被一般用户进程执行。UMASK设置一个内部掩码，它在创建文件时使用。例如执行了

```
umask(022);
```

后，CREAT和MKNOD指定的文件权限码都要与022的反码（755）相与，所得的结果才是最终的保护码，所以如下调用：

```
creat ("file", 0777);
```

将把文件的保护码设置为0755而非0777。由于该掩码被子进程继承，所以如果在登录后shell执行了一次UMASK操作，则此次登录期间所有的用户进程都将受到该掩码的约束，这样就保证不会因为用户的疏忽而导致文件被非法访问。

对于一个文件主为ROOT的SETUID程序，因为其有效用户标识为超级用户，所以它可以访问任何文件。在许多情况下，使程序了解调用此程序的用户对某个文件是否拥有访问权限是很有意义的。

实际上有意义的是检查真正uid是否对文件有访问权限，ACCESS系统调用提供了这样的检查方法，ACCESS的mode参数若为4则检查读权限，为2则检查写权限，为1检查执行权限，而且允许使用这几者的组合。例如若mode为6，则只有当进程的真正uid用户对文件可读可写时再返回成功（0），否则返回-1。当mode为0时仅检查文件是否存在，同时检查从根目录直到该文件的所有目录是否允许搜索。

1.4.6 时间管理系统调用

MINIX有4条用于时间管理的系统调用。TIME系统调用返回当前距1970年1月1日零时的时间，以秒为单位。STIME用来设置系统时间（仅由超级用户执行）。UTIME允许文件主（或超级用户）修改存储在文件i-节点中的时间，例如touch命令就使用UTIME将文件时间设为当前时间。

最后是TIMES系统调用，它返回进程的计账信息，通过它可以知道进程已经占用了多少CPU时间，以及操作系统本身的操作所用的时间（即执行系统调用），同时也返回其所有子进程所用的用户时间与系统时间的总和。

1.5 操作系统结构

至此我们已经了解了操作系统的外部特性（即程序员接口），现在开始观察其内部的组成结构。下面将讨论四种我们都实际尝试过的组织结构，即整体式系统，层次式系统，虚拟机系统和客户－服务器系统。这四种结构并不包括一切，但通过对它们的研究我们将对实际操作系统的设计建立一些基本的概念。

1.5.1 整体式系统

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”，其结构实际就是“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，即它的人口参数和返回值，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，它没有任何程度的隐藏－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）

但即使在整体式系统中，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1－16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：供用户程序用，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1－16（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1－16（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 1-16　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1－17所示。

图 1-17 整体式操作系统的简单结构模型。

1.5.2 层次式系统

图1－17所示的系统进一步通用化就成为层次式系统，即上层软件基于下层软件之上。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1－18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多个进程在单一处理器上运行的细节，总之，第零层提供了CPU基本的多道程序功能。

图 1-18　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、内存、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中操作系统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。

1.5.3 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心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，这如图1－19所示。但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。

图 1-19　带CMS的VM/370结构。

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（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，这就象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。

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都意识到要使老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的另一个缺点是MS－DOS频繁地对中断屏蔽位进行操作，而模拟这些操作是很费时的。

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只是一个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可以运行其自己的操作系统，就象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。

1.5.4 客户/服务器系统

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。为了获取某项服务，比如读文件中的一块，用户进程（客户进程client process）将此请求发送给一个服务器进程（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1－20，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 1-20　客户/服务器模型。

客户－服务器模型的另一个优点是它适用于分布式系统（参阅图1－21）。如果一个客户通过消息传递与服务器通信，客户无需知道这条消息是在本机就地处理还是通过网络送给远地机器上的服务器。在这两种情况下，客户机的处理都是一样的：发送一个请求，收回一个应答。

图 1-21　在一个分布式系统中的客户/服务器模型。

图1－21中所描绘的核心只处理客户与服务器之间的消息传递，这与实际系统不完全符合。某些操作系统功能（如向物理I/O设备寄存器写入命令字）靠用户空间的程序是很难完成的，解决这个问题的方法有两种：一种是设立一个运行于核心态的专用服务器进程，它具有访问硬件的绝对权力，但仍旧通过平常的消息机制与其他进程通信。

另一种方法是在核心中建立一套最少的机制（mechanism），而将策略（policy）留给用户空间中的服务

器进程。例如核心可能将向某特定地址发送的一条消息理解为：取该消息的内容并将其装入某台磁盘的I/O设备寄存器来启动读盘操作。此例中核心甚至不对消息的内容进行合法性检查，而只是将它们机械地拷贝进磁盘设备寄存器（显然这里需要使用某种方案以限制此类消息只能发给授权的进程）。机制与策略分离是一个重要的概念，它在操作系统的许多方面都经常出现。

1.6 各章节内容简介

典型的操作系统由四部分构成：进程管理，I/O设备管理，存储器管理和文件管理。MINIX也分为这样四部分，随后的四章将分别讨论这四个论题，第六章列出了阅读材料和参考文献。

第二章到第五章结构大致相同，首先阐述该论题的基本原理，然后介绍MINIX中的相应内容（同样适用于UNIX）。最后详细地讨论MINIX中的实现。如果读者仅对系统的基本原理感兴趣，他可以略过实现部分的内容而不会引起任何的不连贯。但是如果希望了解一个真实的操作系统(MINIX)是如何工作的，那么应该阅读包括MINIX代码在内的全部内容。

1.7 小结

对操作系统有两种观点：资源管理器观点和扩展的计算机观点。从资源管理器观点看，操作系统的任务是高效地管理整个系统的各个部分；从扩展的计算机观点看，其任务是为用户提供一台比物理计算机更易于使用的虚拟计算机。

操作系统的历史很长，从早期的代替操作员手工操作的系统，到现在的多道程序系统。

任何操作系统的核心都是一套系统调用，系统调用界定了操作系统能完成的功能。对于MINIX，其系统调用分为六大类，第一类与进程创建和终止有关。第二类处理信号。第三类针对文件读写。第四类进行目录管理。第五类对信息进行保护。第六类用于时间管理。

操作系统有若干种构造模型，常见的有整体式模型，层次式模型，虚拟机模型和客户－服务器模型。

习　题

1. 操作系统的两个主要功能是什么？
2. 什么是多道程序？
3. 什么是spooling？你认为将来高档个人计算机会将spooling作为标准特性吗？
4. 在早期的计算机中，每一个字节数据的读写都是CPU直接进行处理的（那时没有DMA－直接存储器访问），这种组织结构对多道程序技术的出现有什么影响？
5. 为什么分时计算没有被第二代计算机广泛采用？
6. 下列哪种指令应该只在核心态下执行？
   1 屏蔽所有中断
   2 读时钟日期
   3 设置时钟日期
   4 改变存储器映像图
7. 请指出个人计算机操作系统与大型主机操作系统的不同之处。
8. 一个MINIX文件的属主uid为12，gid为1，该文件的权限保护码为rwxr-x---，另一用户的uid为6，gid为1，他试图执行该文件，结果如何？
9. 在实际应用中，系统中仅存在一个超级用户将导致许多安全问题，为什么？
10. 客户－服务器模型在分布式系统中很流行，它能够用于单机系统吗？
11. 在分时系统中为什么需要进程表？在个人计算机系统中只存在一个进程，它完全控制整台计算机的使用，在这样的系统中是否还需要进程表？
12. 块设备文件和字符设备文件的本质区别是什么？
13. 在MINIX系统中，用户2对用户1所属的一个文件建立了一个链接，随后用户1删除了此文件，此时用户2访问该文件，其结果如何？
14. 为何CHROOT系统调用仅限于超级用户使用？（提示：信息保护问题）
15. MINIX中为什么设立一个update进程在系统中一直运行？
16. 在什么情况下忽略SIGALRM信号是有意义的？
17. 写一个（组）程序，尝试使用MINIX的所有系统调用。对每一条系统调用，尝试使用不同的参数，包括错误参数，验证这些错误参数是否被检测出来。
18. 写一个类似与图1－10的shell，要求其中包括足够多的代码以便能够对其进行测试。可以加入一些特性如输入，输出的重定向，管道和后台进程等。

第二章　进　程

我们现在开始深入地研究操作系统（特别是MINIX）是如何设计和构造的。 操作系统中最核心的概念是进程：一个对正在运行的程序的抽象。操作系统的其他所有内容都围饶着进程，所以操作系统的设计者（及学生）尽可能早地理解进程是很重要的。

2.1 进程介绍

所有现代的计算机都能同时做几件事情。当一个用户程序正在运行时，计算机还能够同时读盘，并向屏幕或打印机输出正文。 在一个多道程序系统中，CPU由这道程序向那道程序切换，使每道程序运行几十或几百毫秒。 然而严格地说，在一个瞬间，CPU只能运行一道程序。在1秒钟期间，它可能运行多道程序，这样就给用户一种并行的错觉。有时人们所说的伪并行就是指CPU在多道程序之间快速地切换,以此来区分它与多处理机(两个或更多的CPU共享物理存储器）系统真正的硬件并行。人们很难对多个并行的活动进行跟踪。因此，经过多年的努力，操作系统的设计者发展了一种模型（顺序进程），使得并行更容易处理。该模型及其使用正是本章的主题。

2.1.1 进程模型

在该模型中，计算机上所有可运行的软件，通常包括操作系统，被组织成若干顺序进程，简称进程（processes）。一个进程就是一个正在执行的程序，包括程序计数器、寄存器和变量的当前值。从概念上说，每个进程拥有它自己的虚拟CPU。 当然，实际上真正的CPU在各进程之间来回切换。但为了理解这种系统，考虑在（伪）并行情况下运行的进程集，要比我们试图跟踪CPU如何在程序间来回切换简单得多。这种快速的切换称作多道程序，正如在上一章所看到的。

在图2-1（a）中，我们看到在一个多道程序计算机的内存中有四道程序。在图2-1（b）中，我们看到四个进程各自拥有自己的控制流程（即自己的程序计数器），并且每个都独立地运行。在图2-1（C）中，我们看到在观察一段足够长的时间后，所有的进程都有所进展。但在一个给定的瞬间仅有一个进程真正在运行。

图2-1 （a）多道程序中的四道程序。 （b）四个独立、顺序进程的概念模型。 （c）在任意时刻仅有一个程序活跃。

由于CPU在各进程之间来回切换，每个进程执行运算的速度是不确定的，而且当同一进程再次运行其运算速度通常也不可再现。所以，进程的编程绝不能对时序作任何固定的假设。例如考虑一个I/O进程用流式磁带机恢复被备份的文件，它执行一个10000次的空循环以等待磁带机达到正常速度，然后发出命令读取第一个记录。如果CPU决定在空循环期间将处理机调度给其他进程,则磁带机进程可能在第一条记录通过磁头之后还未被再次调度。 当一个进程具有此类严格的实时要求时，也就是一些特定事件一定要在所指定的若干毫秒中发生，那么必须采取特殊措施来保证它们一定在这段时间中发生。然而，通常大多数进程并不受CPU多道程序或其他进程相对速度的影响。

进程和程序之间的区别是很微妙的，但却非常重要。一个类比可以使我们更容易理解这一点。想象一位有一手好厨艺的计算机科学家正在为他的女儿烘制生日蛋糕。他有做生日蛋糕的食谱，厨房里有所需的原料：面粉、鸡蛋、糖、香草汁等等。在这个比喻中，做蛋糕的食谱就是程序（即用适当形式描述的算法），计算机科学家就是处理机（CPU），而做蛋糕的各种原料就是输入数据。进程就是厨师阅读食谱、取来各种原料、以及烘制蛋糕的一系列动作的总和。

现在假设计算机科学家的儿子哭着跑了进来，说他被一只蜜蜂螫了。计算机科学家就记录下他照着食谱做到哪儿了（保存进程的当前状态），然后拿出一本急救手册，按照其中的指示处理螫伤。这里，我们看到处理机从一个进程（做蛋糕）切换到另一个高优先级的进程（实施医疗救治），每个（进程）拥有各自的程序（食谱和急救书）。当蜜蜂螫伤处理完之后，计算机科学家又回来做蛋糕，从他离开时的那一步继续做下去。

这里的关键思想是：一个进程是某种类型的一个活动，它有程序、输入、输出、及状态。单个处理机被若干进程共享，它使用某种调度算法决定何时停止一个进程的工作，并转而为另一个进程提供服务。

进程的层次结构

支持进程概念的操作系统必须提供某种途径来创建所需要的进程。在一些非常简单的系统，或那种设计为仅有一个应用运行的系统（例如，实时地控制一个设备）中，可能在系统启动时，以后所需要的所有进程都已存在。然而在多数系统中，需要有某种方法以便按需创建或撤销进程。在MINIX系统中，进程通过调用FORK系统调用来创建进程，它将创建一个与调用进程相同的进程。子进程同样也能执行FORK，所以有可能形成一棵完整的进程树。在其他操作系统中，也具有若干系统调用，用来创建一个进程、装入它的内存、并使其开始运行。不管系统调用的具体形式如何，系统都需为进程提供一种方法，使其能够创建其他进程。注意，每个进程只有一个父进程，但可以有0个、1个、2个或更多个子进程。

作为进程树如何使用的一个简单例子,让我们来看MINIX在启动时是怎样对其自己进行初始化的。在MINIX的引导映像中有一个称为init的特殊进程，当它开始运行时，它读取一个记录了存在多少个终端的文件，然后为每个终端创建一个新的进程，这些进程等待用户进行登录。如果登录成功，登录进程执行一个shell程序来接受命令。这些命令可能启动更多的进程，依次类推。这样，系统中的所有进程都属于一棵进程树，而init进程则是进程树的根（init 和shell的代码未列在本书中，它们可从其他地方取到）。

进程的状态

尽管每个进程是一个独立的实体，有它自己的程序计数器和内部状态，但进程之间经常需要交互作用。一个进程的输出结果可能作为另一个进程的输入。在shell命令

```
cat  chapter1  chapter2  chapter3 |  grep  tree
```

中，第一个进程运行cat，将三个文件连接并输出。第二个进程运行grep，它从输入中选择所有包含单词“tree”的那些行。根据这两个进程的相对速度（这取决于这两个程序的相对复杂度和各自所分配到的CPU时间），可能发生“grep”准备就绪可以运行，但输入还没有到这种情况。于是它就必须被阻塞直到输入到来。

当一个进程在逻辑上不能继续运行时，它就阻塞，典型的例子是它在等待可以使用的输入。还可能有这样的情况，一个概念上能够运行的进程被迫停止，其原因是操作系统调度另一个进程占用处理机。这两种条件是完全不同的。第一种情况下，挂起是程序自身所固有的（在用户命令行被键入之前，你无法执行它）；第二种情况则是由系统引起的（没有足够的CPU，所以不能使每个进程都有一台它私用的处理机）。在图2-2中我们看到进程三种状态的转换图。这三种状态是：

1 运行态（在该时刻实际占用处理机）

2 就绪态（可运行，因为其他进程正在运行而暂时地被挂起）

3 阻塞态（除非某种外部事件发生，否则不能运行）

图2-2 一个进程可处于运行态、阻塞态、就绪态。图中示出各状态之间的转换。

前两种状态在逻辑上很类似。这两种状态下的进程都希望运行，只是在后者中，暂时没有CPU分配给它。第三种状态与前两种状态不同，该状态的进程不能运行，即使CPU空闲也不行。

这三种状态之间有四种可能的转换关系。转换1在进程发现它不能继续运行下去时发生。在某些系统中，进程需要执行一个系统调用－BLOCK，来进入阻塞状态。在其他系统中，包括MINIX，当一个进程从管道或设备文件（例如终端）读取数据时，如果没有可以使用的输入，则进程自动被阻塞。

转换关系2和3是由进程调度程序引起的，它是操作系统的一部分，进程甚至感知不到调度程序的存在。在系统认为运行进程占用处理机的时间已经过长，决定让其他进程占用处理机时，发生转换2。在系统已让其他进程享有了它们应有的CPU时间而重新轮到该进程来占用处理机时，发生转换3。调度程序的主要内容是决定哪个进程应当运行，及它应运行多长时间。这是很重要的一点，我们将在本节的后边对其进行讨论。已经提出许多算法，它们力图从系统作为一个整体的角度平衡需求和效率之间的竞争，并公平地对待各进程。

当一个进程等待的一个外部事件发生时（例如一些输入到达），发生转换4。如果此时没有其他进程运行，则转换3将立即被触发，该进程便开始运行。否则它将处于就绪态等待CPU空闲。

使用进程模型，我们就易于想象系统内部的操作状况。一些进程运行着一些程序，这些程序执行用户键入的命令，另一些进程是系统的一部分，它们的任务是处理下列一些工作：诸如执行文件服务请求、管理运行磁盘驱动器和磁带机的细节等。当发生一个磁盘中断时，系统作出决定，停止运行当前进程，而转向磁盘进程，该进程在此之前因等待该中断而处于阻塞态。这样，我们可以不再考虑中断，而是考虑用户进程、磁盘进程、终端进程等等。这些进程在等待时总是处于阻塞状态。所等待的事件发生时，则它们被解除阻塞，并成为可被调度运行的进程。

这个观点即引出了图2-3的模型。这里最低层是操作系统的调度程序，在它上面有许多进程。所有关于中断处理、启动和中止进程的具体细节被隐藏在调度程序中。实际上，它是一段非常短小的程序。操作系统的其他部分被简洁地组织成进程形式。图2-3的模型被MINIX使用，但是其中的调度程序应不仅仅被理解为对进程的调度安排，同时也包括中断处理和所有的进程间通信。不过，作为近似描述，它示出了其基本结构。

图2-3 按进程组织的操作系统中最低层处理中断和进程调度，其上是一些顺序进程。

2.1.2进程的实现

为了实现进程模型。 操作系统维持着一张表格（一个结构数组）即进程表(process table)。 每个进程占用一个进程表项。该表项包含了进程的状态、它的程序计数器、栈指针、内存分配状况、打开文件状态、计费和调度信息，以及其他在进程由运行态转到就绪态时必须保存的信息，只有这样才能使进程随后被再次启动，就象从未被中断过一样。

在MINIX中，进程管理、内存管理和文件管理是由系统中的几个独立模块分别处理的，所以进程表被分为几个部分，各模块维护它们各自所需要的那些域。图2-4示出了一些重要的域。与本章有关的域位于第一列，给出其他两列仅仅是为了建立一点概念，即系统的其他部分需要哪些信息。

进程管理
内存管理
文件管理
寄存器
正文段指针
UMASK 屏蔽
程序计数器
数据段指针
根目录
程序状态字
bss段指针

工作目录
栈指针
退出状态
文件描述符
进程状态
信号状态
有效uid
进程开始时间
进程标识号
有效gid
使用的CPU时间
父进程
系统调用参数
子进程的CPU时间
进程组
各种标志位
下次报警时间
真实uid

消息队列指针
有效uid
挂起的信号位
真实gid
进程标识号
有效gid
各种标志位
信号位图
各种标志位
图2-4 MINIX进程表的某些域。

在了解进程表之后，就可以对一台有单个CPU、多个I/O设备的计算机如何维持多个顺序进程的假象作更多的解释。接下来的内容从技术上说是对图2-3中MINIX调度程序如何工作的一个描述，但多数现代的操作系统从本质上来说都差不多。与每类I/O设备（例如软盘、硬盘、定时器、终端）相关的都有一个靠近内存底部的位置，称作中断向量。它包含中断服务程序的入口地址。假设当一个磁盘中断发生时，用户进程3正在运行，则中断硬件将程序计数器、程序状态字、可能还有一个或多个寄存器压入（当前）堆栈，计算机随即跳转到磁盘中断向量所指的地址处。这是硬件做的操作，从这里开始，软件就接管了一切。

中断服务程序的工作从把当前进程全部寄存器值存入进程表项开始。当前进程号及一个指向其表项的指针被保存在全局变量中以便能够快速地找到它们。随后将中断存入的那部分信息从堆栈中删除，并将栈指针指向一个被进程管理者所使用的临时堆栈。一些动作，诸如保存寄存器值和设置栈指针等无法用C语言描述，所以由一个短小的汇编语言例程来完成。当该例程结束后，它调用一个C过程来完成剩下的工作。

MINIX中的进程间通信通过消息完成，所以下一步是构造一条发给磁盘进程的消息，这时磁盘进程正在阻塞并等待该消息。这条消息通知说发生了一条中断，以此将它和那些由用户进程发送的消息加以区分。那些消息发出读磁盘块之类的请求。现在磁盘进程的状态由阻塞转换到就绪，然后，中断服务程序调用调度程序。在MINIX中，不同的进程有不同的优先级，以此向I/O设备服务例程提供比用户进程更好的服务。如果当前磁盘进程是优先级最高的就绪进程，则它将被调度运行。如果被中断进程具有与它相等或更高的优先级， 则它将被再次调度运行，而磁盘进程将只得等待一会儿。

不论哪种情况，被汇编语言中断代码所调用的C过程现在返回，汇编语言代码为新的当前进程装入寄存器值和内存映像并启动它运行。图2-5中总结了中断处理和调度的过程。值得注意的是各系统在细节上略有不同。

图2-5 当一个中断发生后作为操作系统最低层的调度程序的工作步骤。

2.1.3线程

在我们刚才所讨论的传统的进程中，每个进程中只存在一条控制线索和一个程序计数器。但在有些现代操作系统中，提供了对单个进程中多条控制线索的支持。这些控制线索通常被称为线程(threads)，有时也称为轻量进程(lightweight processes)。

在图2－6（a）中我们看到三个传统的进程。每个进程有自己的地址空间和单一的控制线索。与此相反，在图2－6（b）中我们看到一个进程有三条控制线索。尽管这两种情况都有三个线程，但在图2-6（a）中各线程

在不同的地址空间中操作，而在图2－6（b）中所有三个线程共享同一个地址空间。

图2-6（a）三个进程各有一个线程。（b） 一个进程有三个线程。

作为使用多线程的一个例子，我们考虑一个文件服务器进程。它接收读写文件的请求并将所请求的数据送回或者接收更新了的数据。为了提高性能，服务器在内存中维护一个高速缓存，里边存放最近用过的文件，在需要时从该缓存中读数据或向其中写数据。

这种情况就非常适合图2-6（b）中的模型。当一个请求到来时，将它递交给一个线程处理。如果这个线程因等待磁盘传输而中途阻塞，其他线程仍旧可以运行，这样服务器就可以在磁盘I/O进行的同时继续处理新的请求。图2-6（a）的模型就不合适，因为在这里所有的文件服务器线程共享同一块高速缓存非常关键，而图2-6（a）中的三个线程并不共享相同的地址空间，所以无法访问同一块高速缓存。

使用线程的另一个例子是WWW浏览器，例如Netscape和Mosaic。许多Web页面都包含有多幅很小的图像。对Web页面上的每一幅小图像，浏览器都必须与页面的驻留站点建立一条单独的连接以索取该图像。这样就有大量的时间浪费在了建立和释放这些连接上。通过在浏览器内设立多个线程，便可以同时请求传输多幅图像，在多数情况下这样做显著地提高了性能，因为对于小图像，限制因素在于建立连接的时间，而不在于传输线的速率。

当同一地址空间中有多个线程时，图2-4中的几个域就不再针对进程，而是针对线程，于是就需要一张单独的线程表，每个线程占用一项。针对每个线程的信息包括程序计数器、寄存器值及状态。需要程序计数器是因为线程可以象进程一样被挂起和恢复运行。需要寄存器值是因为当线程被挂起时，它的寄存器值必须被保存下来。最后，线程象进程一样，可处于运行，就绪或阻塞态。

在有些系统中，操作系统感知不到线程的存在，换言之，线程完全在用户空间进行管理。例如，当一个线程将被阻塞时，它在停止之前选择并启动它的后继线程。目前有几个用户级的线程软件包用得很普遍，包括POSIX的P－线程和MACH的C－线程软件包。

在另外一些系统中，操作系统知道每个进程中存在多个线程，所以当一个线程阻塞时，操作系统会选择下一个运行的线程，它可能来自同一个进程，或者其他进程。为了进行调度，核心必须有一张线程表，其中列出了系统中所有的线程，这与进程表很类似。

尽管这两个种选择看起来是等价的，但它们的性能相差甚远。在用户空间管理的线程其切换速度比需要核心调用的情况快得多。这一事实有力地支持将线程管理放在用户空间。而另一方面，当线程完全在用户空间管理时，若一个线程阻塞（例如，等待I/O或处理页面故障），则核心将整个进程阻塞，因为它甚至不知道其他线程的存在。这一事实又有力地支持将线程放在核心进行管理。最后的结果是两种系统都被使用，同时还提出了各种混合方案（Anderson et al., 1992）

不论线程是放在核心还是用户空间，它们都带来了一大堆必须解决的问题，而且这些问题相当程度上改变了编程模型。首先来考虑对FORK系统调用的影响。如果父进程有多个线程，那么子进程是否也应该有这些线程？如果不是，那么它可能无法正常工作，因为可能所有的线程都是必不可少的。

然而，如果子进程获得了与父进程一样多的线程，那么当一个线程阻塞在一条READ调用时，比如键盘，这时是否两个线程都阻塞在键盘上？当键入一行内容时，是否两个线程都得到一份拷贝？还是只有父进程得到？还是只有子进程得到？对于打开网络连接也存在同样的问题。

另一类问题与多线程共享许多数据结构有关。若一个线程关闭一个文件而另一个线程正在读该文件，将有什么后果？假设一个线程注意到内存不够并开始申请更多的内存，但此时发生线程切换，新运行的线程也注意到这个问题并再次申请内存。那么这里是申请一次呢，还是两次？在几乎所有设计时未考虑线程的系统中，其库例程（例如申请内存的例程）都不可重入，如果前一个调用尚在激活时进行第二次调用，则必然会引起崩溃。

另一个问题就是错误报告。在UNIX中，一条系统调用执行完之后，其状态将放在一个全局变量errno。如果一个线程执行系统调用，在它读取errno之前，另一个线程也执行一条系统调用并清除了errno原先的值，则情况会怎样？

下面考虑信号。有些信号在逻辑上是针对线程的，另一些则不是。例如，当一个线程调用ALARM，则将产生的信号传递给执行调用的线程是很合理的。当核心能够感知到线程时，通常它可以保证由正确的线程获得该信号。当核心感知不到线程时，线程软件包必须以某种方式跟踪闹钟信号。对于用户级线程还有另一个麻烦：一个进程（例如在UNIX中）某一时刻只能定一个时间闹钟，而若干线程各自独立地调用ALARM，则将发生混乱。

其他信号，例如键盘中断，不是特定于线程的。那么应该由谁来捕捉它们？是一个专门的线程？所有的线程？还是一个新创建的线程？这些方法都存在问题。进一步地，如果一个线程修改了信号处理程序而未通知其他线程，将发生什么情况？

线程引起的最后一个问题是堆栈管理。在许多系统中，当堆栈发生溢出时，核心只是自动地提供更多的堆栈空间。当一个进程有多个线程时，它必须也有多个堆栈。如果核心感知不到所有这些堆栈，则发生堆栈故障时它不能自动地将其扩展，实际上，它可能甚至意识不到内存故障与堆栈增长有关。

这些问题当然并不是不法克服，但它们确实表明仅仅向一个现有系统中引入线程而不进行彻底的重新设计是根本不行的。起码系统调用的语义要重新定义，库例程也要重写。而且所有这些改变必须保持向后兼容，

以保证现存的只有一个线程的进程的可用性。关于线程的其他信息，请参阅（Hauser et al., 1993和Marsh et al., 1991）。

2.2进程间通信

进程经常需要与其他进程通信。例如，在一个shell管道中，第一个进程的输出必须传送到第二个进程，这样沿着管道传递下去。因此在需要通信的进程之间，最好使用一种结构较好的方式，而不要使用中断。在随后几节中，我们就来看进程间通信(IPC)问题。

简单地说，这里有三方面内容。第一方面已经在上面提到过：一个进程如何向另一个进程传送信息。第二方面必须要保证两个或多个进程在涉及临界活动时不会彼此影响（设想两个进程都试图攫取最后100K内存的情况）。第三方面涉及存在依赖关系时进行适当的定序：如果进程A产生数据，进程B打印数据。则B在开始打印之前必须等到A产生了一些数据为止。从下一节开始我们将讨论这三个问题。

2.2.1竞争条件

在有些操作系统中，协作进程可能共享一些彼此都能够读写的公用存储区。它可能在内存中，也可能是一个共享文件；这里共享内存的位置并不影响通信的本质及其带来的问题。为了理解实际中进程间通信如何工作，我们考虑一个简单但很普遍的例子，一个假脱机打印程序。当一个进程需要打印文件时，它将文件名放在一个特殊的打印机假脱机系统（Spooler）目录下。另一个打印机精灵进程周期性地检查否有文件需要打印，若有它就打印之并将该文件名从Spooler目录下删掉。

设想我们的Spooler目录有许多（潜在地有无限多个）槽，编号依次为0，1，2 .....，每个槽存放一个文件名。同时设有两个共享变量，out，指明下一个被打印的文件。in，指向目录中下一个空闲的槽。这两个变量可以被保存在一个所有进程都可访问的两个字的文件中。某一时刻，0到3号槽空闲（其中的文件已经被打印完毕），4到6号槽被占用（其中存有待打印的文件名）。几乎在同一时刻，进程A和进程B都决定将一个文件排队打印，这种情况示于图2-7。

图2-7　两个进程试图在同一时刻访问共享内存。

在Murphy法则1生效时,可能发生以下的情况。进程A读到in的值为7,将它存在一个局部变量next_free_slot中。此时发生一次时钟中断，CPU认为进程A已运行了足够长的时间，决定切换到进程B。进程B也读取in，同样得到值为7，于是它将要打印的文件名存入7号槽，并将in的值更新为8。然后它继续执行其他操作。

最后进程A接着从上次中止的地方再次运行，它检查变量next_free_slot，发现其值为7，于是将打印文件名存入7号槽，这将把进程B存在那里的文件名删掉。然后它将next_free_slot加1，得到值为8，就将8存到变量in中。此时Spooler目录内部是一致的，所以打印机精灵进程将发现不了任何错误，但进程B却永远得不到任何打印输出。类似这样的情况，两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序，就称为竞争条件(race conditions)。调试包含有竞争条件的程序是一件很头痛的事。大多数的运行结果都很好，但在极少数情况下发生一些无法解释的奇怪的事情。

2.2.2临界区

如何避免竞争条件?实际上凡牵涉到共享内存、共享文件、以及共享任何资源的情况都会引发与前边类似的错误。要避免这种错误，关键是要找到某种途径来阻止多于一个的进程同时读写共享的数据。换言之，我们需要的是互斥(mutual exclusion) — 即某种手段以确保当一个进程在使用一个共享变量或文件时，其他进程不能做同样的操作。前述问题的症结就在于，在进程A对共享变量的使用未结束之前进程B就使用它。为实现互斥而选择适当的原语是任何操作系统的主要设计内容之一。也是我们后续章节中要详细加以讨论的。

避免竞争条件的问题也可以用一种抽象的方式进行描述。一个进程的一部分时间做内部计算或其他一些不会引发竞争条件的操作。在某些时候进程可能会访问共享内存或共享文件，或执行其他一些会导致竞争的操作。我们把对共享内存进行访问的程序片段称作临界区(critical region)，或临界段(critical section)。如果我们能够适当地安排使得两个进程不可能同时处于临界区，则就能够避免竞争条件。

尽管这样的要求防止了竞争条件，但它还不能保证使用共享数据的并发进程能够正确和高效地进行操作。对于一个好的解决方案，我们需要以下4个条件：

1 任何两个进程不能同时处于临界区。

2 不应对CPU的速度和数目作任何假设。

3 临界区外的进程不得阻塞其他进程。

4 不得使进程在临界区外无休止地等待。

2.2.3忙等待的互斥

本节将看几种实现互斥的方案。在这些方案中，当一个进程在临界区中更新共享内存时，其他进程将不会进入其临界区，也不会带来任何麻烦。

关中断

这种最简单的方法是使每个进程在进入临界区后先关中断，在离开之前再开中断。中断被关掉后，时钟中断也被屏蔽。CPU只有在发生时钟或其他中断时才会进行进程切换，这样关中断之后CPU将不会被切换到其他进程。于是，一旦进程关中断之后，它就可以检查和修改共享内存，而不必担心其他进程的介入。

这种方案不好，因为把关中断的权力交给用户进程是不明智的。设想一下若一个进程关中断之后不再开中

断，其结果将会如何？系统可能会因此终止。而且，如系统有两个或多个共享内存的处理器，则关中断仅仅对执行本指令的CPU有效，其他CPU仍将继续运行，并可以访问共享内存。

另一方面，对核心来说，当它在更新变量或列表的几条指令期间将中断关掉是很方便的。因为当诸如就绪进程队列之类的数据状态不一致时发生中断，则将导致竞争条件。所以得出结论：关中断对于操作系统本身是一项很有用的技术，但对于用户进程则不是一种合适的通用互斥机制。

锁变量

作为第二种尝试，我们来寻找一种软件解决方案。设想有一个共享（锁）变量，初值为0。当一个进程想进入其临界区时，它首先测试这把锁。如果锁的值为0，则进程将其置为1并进入临界区。若锁已经为1，则进程将等待直到其变成0。于是，0就表示临界区内没有进程，1表示已经有某个进程进入了临界区。

不幸的是，这种想法也含有与Spooler目录一样的纰漏。假设一个进程读锁变量的值并发现它为0，而恰好在它将其置为1之前，另一个进程被调度运行，将锁变量置为1。当第一个进程再次能运行时，它同样也将锁置为1，则此时同时有两个进程处于临界区中。

可能你会想,先读取锁变量,紧挨着在改变其值之前再检查一遍它的值，这样便可以解决问题。但这实际上无济于事。如果第二个进程恰好在第一个进程完成第二次检查之后修改锁变量，则同样还会发生竞争条件。

严格地轮换法

互斥的第三种方法示于图2-8。与本书中几乎所有其他程序一样，这里的程序段用C编写。之所以选择C是由于实际的操作系统普遍用C编写（偶而有用C++），而基本上不用象Modula2或Pascal那样的语言。

```
while（TRUE）{                    while（TRUE）{
     while（turn!=0） /*等待*/;          while（turn!=1） /* 等待 */;
 critical_region（）;                 critical_region（）;
     turn=1;                           turn=0;
     noncritical_region（）;             noncritical_region（）;
}                                  }
      （a）                                    （b）
```

图2-8　临界区问题的一种解法。

在图2-8中，整型变量turn初值为0，它用于跟踪轮到哪个进程进入临界区来检查或更新共享内存。一开始进程0检查turn，发现它是0，于是进入临界区。 进程1同样也发现它是0，于是执行一个等待循环不停地检测它是否变成了1。 持续地检测一个变量直到它具有某一特定值就称作忙等待(busy waiting)。忙等待是应该避免的，因为它浪费CPU时间。只有在有理由预期等待时间很短时才使用忙等待。

当进程0离开临界区时，它将turn置为1，以允许进程1进入其临界区。假设进程1很快便离开了临界区，则这时两个进程都处于临界区之外，turn的值被置为0。现在进程0很快便执行完了其整个循环，它再次执行到非临界区的部分，并将turn置为1。此时，进程0结束了其非临界区的操作并回到循环的开始，但很不幸，这时它不能进入临界区，因为turn的值为1，而进程1还在忙于非临界区的操作。这说明，轮流进入临界区在一个进程比另一个慢很多的情况下并不是一个好办法。

这种情况违反了以上条件3：进程0被一个临界区之外的进程阻塞。再回到Spooler目录的问题，如果我们现在将临界区与读写Spooler目录相联系，则进程0有可能因为进程1在做其他事情而被禁止打印另一个文件。

实际上，该方案要求两个进程严格地轮流进入它们的临界区。例如若用假脱机方式打印文件，那么任何一个进程都不可能在一轮中打印两个文件。尽管该算法的确避免了所有的竞争，但由于它违反了条件3，所以不能作为一个很好的备选方案。

Peterson解决方案

一个荷兰数学家T.Dekker将轮换法和锁变量及警告变量的思想相结合，最早提出了一个不需要严格轮换的软件互斥解法。关于Dekker的算法，请参阅（Dijkstra，1965）。

在1981年，G.L.Peterson发现了一种简单得多的互斥算法。这使得Dekker的方法不再有任何新意。Peterson的算法示于图2-9。该算法由两个用ANSI C写的过程组成。ANSI C要求为所定义和使用的所有函数提供函数原型，但为了节省篇幅，在本例和后边的例子中我们将不给出函数原型。

```
#define FALSE   0
#define TRUE    1
#defiine N       2                    /*进程数*/
int  turn;                            /*轮到谁了?*/
int  interested[N];                   /*所有值初始为0（FALSE）*/

void enter_region（int process）            /*进程号为0或1*/
{
```

```
        int other;                          /*另一个进程的进程号*/
    other=1-process;                    /*另一个进程*/
     interested[process]=TRUE;              /*标识出希望进入临界区*/
     turn=process;                      /*设置标志位*/
     while（turn==process&&interested[other]==TRUE）; /*空语句*/
}
void leave_region（int process）            /*process：即将离开临界区的进程*/
{
      interested[process]=FALSE;        /*标识将离开临界区*/
}
```

图2-9 完成互斥的Peterson方案。

在使用共享变量（即进入其临界区）之前，各进程使用其进程号0或1作为参数来调用enter_region，该调用在需要时将使进程等待，直到能安全地进入。进程在完成对共享变量的操作之后，将调用leave_region，表示操作已完成，若其他进程希望进入临界区，则现在可以进入。

我们来看这个方案如何工作。起初没有任何进程处于临界区，现在进程0调用enter_region，它通过将其数组元素置位和将turn置为0来标识它希望进入临界区。由于进程1并不想进入临界区，所以enter_region很快便返回。如果进程1现在调用enter_region，它将在此处挂起直到interested[0]变成FALSE，该事件只有当进程0调用leave_region退出临界区时才会发生。

现在考虑两个进程几乎同时调用enter_region的情况。它们都将自己的进程号存入turn。但只有后一个被保存进去的进程号才有效，前一个是无效的。假设进程1后存，则turn为1。当两个进程都运行到while语句时，进程0将循环0次并进入临界区，而进程1将不停地循环，并不得进入临界区。

TSL指令

现在来看一种需要硬件支持的方案。许多计算机，特别是那些为多处理机设计的计算机，都有一条指令叫做测试并上锁（TSL）。其工作如下所述：它将一个存储器字读到一个寄存器中，然后在该内存地址上存一个非零值。读数和写数操作保证是不可分割的－即该指令结束之前其他处理机均不允许访问该存储器字。执行TSL指令的CPU将锁住内存总线以禁止其他CPU在本指令结束之前访问内存。

为了使用TSL指令，我们要使用一个共享变量lock来协调对共享内存的访问。当lock为0时，任何进程都可以使用TSL指令将其置为1并读写共享内存。当操作结束时，进程用一条普通的MOVE指令将lock重新置为0。

这条指令如何被用来防止两个进程同时进入临界区呢？解决方案示于图2-10中。其中示出了使用4条指令的汇编语言例程。第一条指令将lock原来的值拷贝到寄存器中并将lock置为1，随后这个原先的值与0相比较。如果它非零，则说明先前已被上锁，则程序将回到开头并再次测试。经过或长或短的一段时间后它将变成0（当前处于临界区中的进程退出临界区时），于是子例程返回，并上锁。清除这个锁很简单，程序只需将0存入lock即可，不需要特殊的指令。

```
enter_region:
    tsl register, lock              |复制lock到寄存器，并将lock置为1
    cmp register, #0                |  lock等于0吗?
    jne enter_region                |如果不等于0，已上锁，再次循环
    ret                             |返回调用程序，进入临界区

leave_region:
    move lock ,  #0                 |置lock为0
    ret                             |返回调用程序
```

图2-10 用TSL指令上锁和清除锁。

现在就有一种很明确的解法了。进程在进入临界区之前先调用enter_region。这将导致忙等待，直到锁空闲为止。随后它获得锁变量并返回。在进程从临界区返回时它调用leave_region，这将把lock置为0。与临界区问题的所有解法一样，进程必须在正确的时间调用enter_region和leave_region，解法才能奏效。如果一个进程有欺诈行为，则互斥将会失败。

2.2.4 睡眠和唤醒

Peterson解法和TSL解法都是正确的，但它们都有忙等待的缺点。这些解法在本质上是这样的：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则进程将踏步等待，直到许可为止。

这种方法不仅浪费CPU时间，还可能引起预料不到的结果。考虑一台计算机有两个进程，H优先级较高，L优先级较低。调度规则规定只要H处于就绪态它就可以运行。在某一时刻，L处于临界区中，此时H变到就绪态准备运行（例如，一条I/O操作结束）。现在H开始忙等待，但由于当H就绪时L不会被调度，也就无法离开临界区，所以H将永远忙等待下去。这种情况有时被称作优先级翻转问题(priority inversion problem)。

现在让我们来看几条进程间通信原语，它们在无法进入临界区时将阻塞，而不是忙等待。最简单的是睡眠（SLEEP）和唤醒（WAKEUP）。SLEEP系统调用将引起调用进程阻塞，即被挂起，直到另一进程将其唤醒。WAKEUP调用有一个参数，即要被唤醒的进程。另一种方法是SLEEP与WAKEUP各有一个参数，即一个用于匹配SLEEP和WAKEUP的内存地址。

生产者 — 消费者问题

作为如何使用这些原语的一个例子，我们考虑生产者－消费者问题（也称作有界缓冲区问题）。两个进程共享一个公共的固定大小的缓冲区。其中的一个，生产者，将信息放入缓冲区；另一个，消费者，从缓冲区中取出信息（该问题也可被推广到m个生产者，n个消费者的情况，但出于简单起见，我们只考虑一个生产者，一个消费者的情况）。

麻烦之处在于当缓冲区已满，而此时生产者还想向其中放入一个新的数据项的情况。解决办法是让生产者睡眠，待消费者从缓冲区中取走一个或多个数据项时再唤醒它。同样地，当消费者试图从缓冲区中取数据而发现缓冲区为空时，它就睡眠，直到生产者向其中放入一些数据时再将其唤醒。

这种方法听起来很简单,但它包含与前边Spooler目录问题一样的竞争条件。为了跟踪缓冲区中的数据项数，我们需要一个变量count。如果缓冲区最多存放N个数据项，则生产者代码将首先检查count是否达到N，若是，则生产者睡眠；否则生产者向缓冲区中放入一个数据项并递增count的值。

消费者的代码与此类似：首先看count是否为0，若是则睡眠；否则从中取走一个数据项并递减count的值。每个进程同时也检测另一个是否应睡眠，若不应睡眠则唤醒之。生产者和消费者的代码示于图2-11。

为了在C语言中表示诸如SLEEP和WAKEUP的系统调用，我们将用对库函数的调用形式来表示。尽管它们不是标准C库的一部分，但在实际上具有这些系统调用的所有系统中都应具有这两种库函数。图中未示出的过程enter_item和remove_item用来记录将数据项放入缓冲区和从缓冲区中取出数据项。

```
# define  N   100                       /* 缓冲区内的槽数 */
int count =0;                           /* 缓冲区内数据项个数 */
void producer（void）
{
    while（TRUE） {                        /* 无限循环 */
        producer_item（）；              /* 产生下一个数据项 */
        if（count==N）sleep（）；        /* 缓冲区满，进入睡眠 */
        enter_item（）；                 /* 将一个数据项放入缓冲区 */
        count=count+1;                  /* 增加缓冲区内数据项个数 */
        if（count==1）wakeup（consumer） /* 缓冲区为空？*/
    }
}
void  consumer（void）
{
    while（TRUE）{                         /* 无限循环 */
           if（count==0）sleep（）；     /* 缓冲区空，进入睡眠 */
           remove_item（）；             /* 从缓冲区中取走一个数据项 */
           count=count-1;                   /* 递减缓冲区中数据项个数 */
           if（count==N-1）wakeup（producer） /* 缓冲区满？*/
           consume_item（）；            /* 打印数据项 */
    }
}
```

图2-11 含有竞争条件的生产者－消费者问题。

现在回到竞争条件的问题。这里有可能会出现竞争条件，其原因是对count的访问未加限制。有可能出现以下情况：缓冲区为空，消费者刚刚读取count的值发现它为0。此时调度程序决定暂停消费者并启动运行生产者。生产者向缓冲区中加入一个数据项，将count加1。现在count的值变成了1。它推断认为由于count刚才为0，所以消费者此时很可能在睡眠，于是生产者调用wakeup来唤醒消费者。

不幸的是，消费者此时在逻辑上并未睡眠，所以唤醒信号丢失。当消费者下次运行时，它将测试先前读到的count值，发现它为0，于是去睡眠。这样生产者迟早会添满整个缓冲区，然后睡眠。这样一来两个进程都将永远睡眠下去。

这里问题的实质在于发给一个（尚）未睡眠进程的唤醒信号丢失了。如果它没有丢失，则一切都很正常。一种快速的弥补方法是修改规则，加上一个唤醒等待位(wakeup waiting bit)。当向一个清醒的进程发送一个唤醒信号时，将该位置位。随后，当进程要睡眠时，如果唤醒等待位为1，则将该位清除，而进程仍然保持清醒。

尽管在本例中唤醒等待位解决了问题，但很容易就可以构造出一些例子，其中有两个或更多的进程，这时一个唤醒等待位就不敷使用。我们可以再打一个补丁，加入第二个唤醒等待位，或者甚至是8个、32个，但原则

上讲这并未解决问题。

2.2.5 信号量

信号量是E. W. Dijkstra在1965年提出的一种方法，它使用一个整型变量来累记唤醒次数，以供以后使用。在他的建议中引入一个新的变量类型，称作信号量(semaphore)。一个信号量的值可以为0，表示没有积累下来的唤醒操作；或者为正值，表示有一个或多个被积累下来的唤醒操作。

Dijkstra建议设两种操作：DOWN和UP（分别为推广后的SLEEP和WAKEUP）。对一信号量执行DOWN操作是检查其值是否大于0。若是则将其值减1（即，用掉一个保存的唤醒信号）并继续。若值为0，则进程将睡眠，而且此时DOWN操作并未结束。检查数值、改变数值、以及可能发生的睡眠操作均作为一个单一的、不可分割的原子操作(atomic action)完成。即保证一旦一个信号量操作开始，则在操作完成或阻塞之前别的进程均不允许访问该信号量。这种原子性对于解决同步问题和避免竞争条件是非常重要的。

UP操作递增信号量的值。如果一个或多个进程在该信号量上睡眠，无法完成一个先前的DOWN操作，则由系统选择其中的一个（例如，随机挑选）并允许其完成它的DOWN操作。于是，对一个有进程在其上睡眠的信号量执行一次UP操作之后，该信号量的值仍旧是0，但在其上睡眠的进程却少了一个。递增信号量的值和唤醒一个进程同样也是不可分割的。不会有进程因执行UP而阻塞，正如在前面的模型中不会有进程因执行wakeup而阻塞一样

在Dijkstra最早的论文中。他使用P和V而不是DOWN和UP，但因为对于不讲荷兰语的读者来说采用什么记号并无大的干系，所以我们将使用DOWN和UP。它们在Algol 68中首次被引入。

用信号量解决生产者 — 消费者问题

图2－12中示出了用信号量解决丢失的唤醒问题。最重要的是它采用一种不可分割的方式来实现。通常是将UP和DOWN作为系统调用实现，而且操作系统只需在执行以下操作时短暂地关掉中断，这些操作包括：检测信号量、修改信号量、以及在需要时使进程睡眠。由于这些动作只需要几条指令，所以关中断不会带来什么副作用。如果使用多个CPU，则每个信号量应由一个锁变量进行保护。通过TSL指令来确保同一时刻只有一个CPU在对信号量进行操作。读者必须搞清楚使用TSL来防止几个CPU同时访问信号量，与生产者或消费者使用忙等待来等待对方腾出或填充缓冲区是完全不同的。信号量操作仅需几个微秒，而生产者或消费者则可能需要任意长的时间。

```
    #define N   100                          /* 缓冲区内槽数 */
typedef int semaphone;                       /* 信号量是一种特殊的整形变量 */
semaphone mutex =1;                        /* 控制对临界区的访问 */
semaphone empty=N;                         /* 记录缓冲区内空的槽数 */
semaphone full=0;                          /* 记录缓冲区内满的槽数 */
void producer（void）
{
    int item;
    while（TRUE）{                         /* TRUE为常数1 */
        produce_item（&item）;             /* 产生一个需放入缓冲区的数据项 */
        down（&empty）;                    /* 递减空槽数 */
        down（&mutex）;                    /* 进入临界区 */
        enter_item（item）;                /* 将一个新数据项放入缓冲区 */
        up（&mutex）;                          /* 离开临界区 */
        up（&full）;                       /* 递增满槽数 */
     }
}
void consumer（void）
{
     int item;
     while（TRUE）{                        /* 无限循环 */
         down（&full）;                        /* 递减满槽数 */
         down（&mutex）;                   /* 进入临界区 */
         remove_item（&item）;             /* 从缓冲区中取走一个数据项 */
         up（&mutex）;                         /* 离开临界区 */
         up（&empty）;                         /* 递增空槽数 */
         consume_item（item）;             /* 对数据项进行操作 */
     }
}
```

图2-12使用信号量的生产者--消费者问题。

该解决方案使用了三个信号量：full用来记录满的缓冲槽数目，empty记录空的缓冲槽总数，mutex

用来确保生产者和消费者不会同时访问缓冲区。full的初值为0，empty的初值为缓冲区内槽的数目，mutex初值为1。两个或多个进程使用的初值为1的信号量保证同时只有一个进程可以进入临界区，它被称作二进制信号量(binary semaphore)。如果每个进程在进入临界区前都执行一个DOWN操作，并在退出时执行一个UP操作，则能够实现互斥。

在有了一些进程间通信原语之后，我们回头再观察一下图2－5中的中断顺序。在使用信号量的系统中，隐藏中断机构的最自然的方法是为每一个I/O设备设置一个信号量，初值设为0。在启动一个I/O设备之后，其管理进程对相关联的信号量执行一个DOWN操作，于是立即被阻塞。当中断到来时，中断处理程序随后对相关联的信号量执行一个UP操作，这将使相关的进程再次成为就绪。该模型中，图2－5中的第6步具体化为在设备的信号量上执行UP操作，所以在第7步中，调度程序将能够运行设备管理程序。当然，如果这时有几个进程就绪，则调度程序下次可以选择一个最为重要的进程来运行。在本章后边我们将看到如何进行调度。

在图2－12的例子中，我们实际上通过两种不同的方式来使用信号量。其间的区别是很重要的。信号量mutex用于互斥。它用于保证任一时刻只有一个进程读写缓冲区和相关的变量。互斥是避免混乱所必需的。

信号量的另一种用途是同步(synchronization)。信号量full和empty用来保证一定的事件顺序发生或不发生。在本例中，它们保证当缓冲区满的时候生产者停止运行，以及当缓冲区空的时候消费者停止运行。这种用法与互斥是不同的。

尽管信号量已经出现了三十多年，人们仍在研究它的应用。作为一个例子，请参阅（Tai 和 Carver，1996）。

2.2.6管程

有了信号量之后，进程间通信看来很容易了，对吗？答案是否定的。仔细看图2－12中向缓冲区放入数据项，以及从中删除数据项之前的DOWN操作。假设将生产者代码中的两个DOWN操作交换一下次序，将使得mutex的值在empty之前被减1，而不是在其之后。如果缓冲区完全是满的，生产者将阻塞，mutex值为0。这样一来，当消费者下次试图访问缓冲区时，它将对mutex的执行一个DOWN操作，由于mutex值为0，则它也将阻塞。两个进程都将永远地阻塞下去，无法做如何有效的工作，这种不幸的状况称作死锁(deadlock)。我们将在第三章中详细地研究死锁。

指出这个问题是为了说明使用信号量时要何等的小心！一处很小的错误将导致很大的麻烦。这就象用汇编语言编程一样，甚至更糟，因为这里出现的错误都是竞争条件、死锁、以及其他不可预测和不可重现的行为。

为了更易于编写正确的程序，Hoare（1974）和Brinch Hansen（1975）提出了一种高级的同步原语，称为管程(monitor)。他们两人提出的方案略有不同，如下面将要描述的。一个管程是一个由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或软件包。进程可在任何需要的时候调用管程中的过程，但它们不能在管程外的过程中直接访问管程内的数据结构。图2-13展示了用一种想象的类Pascal语言描述的管程。

```
monitor example
    integer  i;
    condition c;
    procedure producer（x）;
    。
    。
    。
    end;
    prodecure consumer（x）;
    。
    。
    。
    end;
end monitor;
```

图2-13 一个管程。

管程有一个很重要的特性，使它们能有效地完成互斥：任一时刻管程中只能有一个活跃进程。管程是一种编程语言的构件，所以编译器知道它们很特殊，并可以采用与其他过程调用不同的方法来处理它们。典型的，当一个进程调用管程中的过程时，前几条指令将检查在管程中是否有其他的活跃进程。如果有，调用进程将挂起，直到另一个进程离开管程。如果没有，则调用进程便进入管程。

对进入管程实现互斥由编译器负责，但通常的做法是用一个二进制信号量。因为是由编译器而非程序员来安排互斥，所以出错的可能性要小得多。在任一时刻，写管程的人无需关心编译器是如何实现互斥的。他只需知道将所有的临界区转换成管程中的过程即可，而绝不会有两个进程同时执行临界区中的代码。

尽管如我们上边所看到的，管程提供了一种实现互斥的简便途径，但这还不够。我们还需要一种办法以使得进程在无法继续运行时被阻塞。在生产者－消费者问题中，很容易将针对缓冲区满和缓冲区空的测试放到管程的过程中，但是生产者在发现缓冲区满的时候如何阻塞？

解决方法在于引入条件变量(condition variables)，及相关的两个操作：WAIT和SIGNAL。当一个管程过

程发现它无法继续时（例如，生产者发现缓冲区满），它在某些条件变量上执行WAIT，如full。这个动作引起调用进程阻塞。它也允许另一个先前被挡在管程之外的进程现在进入管程。

另一个进程，如消费者，可以通过对其伙伴正在其上等待的一个条件变量执行SIGNAL来唤醒正在睡眠的伙伴进程。为避免管程中同时有两个活跃进程，我们需要一条规则来通知在SIGNAL之后该怎么办。Hoare建议让新唤醒的进程运行，而挂起另一个进程。Brinch Hansen则建议要求执行SIGNAL的进程必须立即退出管程。换言之，SIGNAL语句只可能作为一个管程过程的最后一条语句。我们将采纳Brinch Hansen的建议，因为它在概念上更简单，并且更容易实现。如果在一个条件变量上正有若干进程等待，则对该条件变量执行SIGNAL操作，调度程序将在其中选择一个使其恢复运行。

条件变量不是计数器，它们并不像信号量那样积累信号供以后使用，所以如果向一个其上没有等待进程的条件变量发送信号，则该信号将丢失。WAIT操作必须在SIGNAL之前。这条规则使得实现简单了许多。实际上这不是一个问题，因为用变量很容易跟踪每个进程的状态。一个原本要执行SIGNAL的进程通过检查这些变量便可以知道该操作是不需要的。

图2-14中用类Pascal语言给出了使用管程解决生产者－消费者问题的解法。

你可能会觉得WAIT和SIGNAL操作看起来很象前面提到的SLEEP和WAKEUP。它们确实很象，但存在一点很关键的区别：SLEEP和WAKEUP之所以失败是因为当一个进程想睡眠时另一个进程试图去唤醒它。使用管程将不会发生这种情况。对管程过程的互斥保证了这样一点：如果管程中的过程发现缓冲区满，它将能够完成WAIT操作而不用担心调度程序可能会在WAIT完成之前切换到消费者进程。消费者进程甚至在WAIT完成且生产者被标志为不可运行之前根本不允许进入管程。

```
monitor ProducerConsumer
   condition full, empty;
   integer  count;
   procedure enter;
   begin
     if count=N then wait（full）;
     enter_item;
     count：=count+1;
     if count=1 then signal（empty）
   end;
   procedure remove;
   begin
     if count=0 then wait（empty）;
     remove_item;
     count：=count-1;
     if count=N-1 then signal（full）;
   end;
   count：=0;
   end monitor;
   procedure producer;
   begin
     while true do
     begin
       produce_item;
       ProducerConsumer。enter
     end
   end;
   procedure cosumer;
   begin
     while true do
     begin
       ProducerConsumer。remove;
       consume_item
     end
   end;
```

图2-14 一个使用管程的生产者－消费者问题解法概要。 在一个时刻仅有一个管程过程活跃，缓冲区有N个槽。

通过临界区互斥的自动化，管程使并行编程比用信号量更容易保证正确性。但它也有缺点。图2-14中之所以使用类Pascal，而不象其他例子那样使用C语言并不是没有原因的。正如我们早先说过的，管程是一个编程语言概念。编译器必须要识别出管程并用某种方式对互斥作出安排。C、Pascal、及多数其他语言都没有管程，所以指望这些编译器来实现互斥规则是不可靠的。实际上，编译器如何知道哪些过程属于管程内部，哪些不属于管程也是一个问题。

上述语言同样也没有信号量，但增加信号量是很容易的：你需要做的就是向库里加入两段短小的汇编程序代码，以执行UP和DOWN系统调用。编译器甚至用不着知道它们的存在。当然，操作系统必须知道信号量，但至少如果你有一个基于信号量的操作系统，你仍旧可以使用C或C++（或者甚至是BASIC，如果你喜欢的话）来写用户程序。如果使用管程，你就需要一种带有管程的语言。有几种语言带有管程，如并发Euclid（Holt，1983），但非常少见。

与管程和信号量有关的另一个问题，是它们都被设计用来解决访问一块公共存储器的一个或多个CPU上的互斥问题。通过将信号量放在共享内存中并用TSL指令来保护它们，我们可以避免竞争。对于一个具有多个CPU、各CPU拥有自己的私有内存、由一个局域网相连的分布式系统，这些原语将失效。得出的结论是：信号量太低级，而管程在少数几种编程语言以外无法使用。同时，这些原语均未提供机器间的信息交换方法，所以我们还需要其他的东西。

2.2.7消息传递

前边提及的其他的东西就是消息传递(message passing)。这种进程间通信方法使用两条原语SEND和RECEIVE。它们象信号量一样，是系统调用；而不象管程那样是语言构件。因此，它们可以很容易地被加入库例程。例如

```
send（destination，&message）；
和
receive（source，&message）；
```

前一个调用向一个给定的目标发送一条消息，后一个调用从一个给定的源（或者是任意源，如果接收者不介意的话）接收一条消息。如果没有消息可用，则接收者可能阻塞直到一条消息到达。或者它也可以立即返回，并带回一个错误码。

消息传递系统的设计要点

消息传递系统面临许多信号量和管程所未涉及的问题和设计难点。特别是对于通信进程位于网络中不同机器的情况。例如，消息有可能被网络丢失。为了防止消息丢失，发送方和接收方可以达成如下一致：一旦信息被接收到，接收方马上回送一条特殊的应答(acknowledgement)消息。如果发送方在一段时间间隔内未收到应答，则进行重发。

现在考虑消息本身被正确地接收，而应答信息丢失的情况。发送者将重发信息， 这样接收者将接收到两次。对于接收者来说，区分新消息和一条重发的老消息是非常重 要的。通常采用在每条原始消息中嵌入一个连续的序号来解决该问题。如果接收者收到一条消息，它具有与前一条消息一样的序号，则它就知道这条消息是重复的，可以忽略。

消息系统还需要解决进程命名的问题，这样才能明确在SEND和RECEIVE调用中所指定的进程。身份认证也是一个问题：客户如何知道它是在与一个真正的文件服务器通信，而不是一个冒充者？

另一方面，对发送者和接收者在同一台机器上的情况，也存在若干设计问题。其中一点是性能。将消息从一个进程拷贝到另一个进段通常比信号量操作和进入一个管程要慢。为了使消息传递变得高效，已经做了许多工作。例如，Cheriton（1986）建议限制信息的大小，使其能装入机器的寄存器中，然后便可以使用寄存器进行消息传递。

用消息传递解决生产者－消费者问题

现在我们来看如何用消息传递而不是共享内存来解决生产者－消费者问题。图2－15中给出了一种解法。我们假设所有的消息都有同样的大小，并且尚未接收到的消息由操作系统自动进行缓冲。在该图中，共使用N 条信息，这就类似于一块共享内存缓冲区中的N个槽。消费者首先将N条空消息发送给生产者。当生产者向消费者传递一个数据项时，它取走一条空消息并送回一条填充了内容的消息。通过这种方式，系统中总的消息数保持不变，所以消息可以存放在预知数量的内存中。

如果生产者的速度比消费者快，则所有的消息最终都将被填满，于是生产者将阻塞以等待消费者取用后返回一条空消息。如果消费者速度快，则正好相反：所有的消息均为空，等待生产者来填充它们，消费者阻塞以等待一条填充过的消息。

```
# deifne N 100                                    /* 缓冲区中的槽数 */
void producer（void）
{
```

```
    int item;
    message m;                              /* 消息缓冲区 */
    while（TRUE）{
          produce_item（&item）;            /* 产生一些数据放入缓冲区 */
          receive（consumer，&m）;            /* 等待一条空消息到达 */
          build_message（&m，item）;          /* 构造一条消息供发送 */
          send（consumer，&m）;               /* 向消费者发送一数据项 */
     }
}
void consumer（void）
{
    int item，i;
    message m;
    for（i=0;i<N;i++）send（producer，&m）;  /* 发送N条空消息 */
    while（TRUE）{
          receive（producer，&m）;          /* 收到一条包含数据的消息 */
          extract_item（&m，&item）;        /* 从消息中析取数据 */
          send（producer，&m）;             /* 回送空消息作为应答 */
          consume_item（item）;             /* 使用数据项进行操作 */
     }
}
```

图2-15 用N条消息的生产者消费者进程。

消息传递可以有许多变体。对于初学者，我们来看如何对消息编址。一种方法是为每个进程分配一个唯一的地址，按进程为消息指定地址。另一种方法是引入一种新的数据结构，称作信箱(mailbox)。一个信箱就是一个用来对一定数量的消息进行缓冲的地方，典型的情况是消息的数量在信箱创建时确定。当使用信箱时，SEND和RECEIVE调用中的地址参数使用信箱，而不是进程。当一个进程试图向一个满的信箱发消息时，它将被挂起，直至信箱内有消息被取走而为新消息腾出空间。

对于生产者－消费者问题，生产者和消费者均应创建足够容纳N条消息的信箱。生产者向消费者信箱发送包含数据的消息，消费者则向生产者信箱发送空消息。当使用信箱时，缓冲机制是很清楚的：目标信箱容纳那些被发送但尚未被目标进程接收的消息。

使用信箱的另一种极端情况是彻底去掉缓冲。采用这种方法时，如果SEND在RECEIVE之前执行，则发送进程被阻塞，直到RECEIVE发生。执行RECEIVE时消息可以直接从发送者拷贝到接收者，不用任何中间缓冲。类似地，如果RECEIVE先被执行，则接收者阻塞直到SEND发生。这种策略常被称为会合(rendezvous)原则。与带有缓冲的消息方案相比，这种方案实现起来更容易一些，但却降低了灵活性，因为发送者和接收者一定要以步步紧接的方式运行。

在MINIX（及UNIX）中用户进程间的通信采用管道，它与信箱在效果上等价。采用信箱的消息系统和管道机制之间的区别实际在于管道没有预先设定消息的边界。换言之，如果一个进程向管道写入10条100字节的消息，而另一个进程从管道中读取1000个字节，则读进程将一次性地获得这所有10条消息。而在一个真正的消息系统中，每个READ操作将只返回一条消息。当然，如果进程能够达成一致：总是从管道中读写固定大小的消息，或者每条消息都以一个特殊字符（如换行符）结束，则不会有任何问题。构成MINIX操作系统的进程之间使用消息大小固定的真正的消息机制进行通信。

2.3 经典IPC问题

操作系统文化中有许多被广为讨论和分析的有趣的问题。以下几节我们将讨论三个较为著名的问题。

2.3.1 哲学家进餐问题

在1965年，Dijkstra提出并解决了一个他称之为哲学家进餐的同步问题。从那时起，每个发明新的同步原语的人都希望通过解决哲学家就餐问题来展示其同步原语的精妙之处。这个问题可以简单地描述如下：五个哲学家围坐在一张圆桌周围，每个哲学家面前都有一碟通心面，由于面条很滑，所以要两把叉子才能夹住。相邻两个碟子之间有一把叉子，餐桌如图2-16所示。

图2-16 哲学家就餐图。

哲学家的生活包括两种活动:即吃饭和思考(这只是一种抽象,即对本问题而言其他活动都无关紧要)。当一个哲学家觉得饿时，他就试图分两次去取他左边和右边的叉子，每次拿一把，但不分次序。如果成功地获得了两把叉子，他就开始吃饭，吃完以后放下叉子继续思考。这里的问题就是：为每一个哲学家写一段程序来描述其行为，要求不能死锁。（要求拿两把叉子是人为规定的，我们也可以将意大利面条换成中国菜，用米饭代替通心面，用筷子代替叉子。）

图2-17给出了最浅显的解法。过程take_fork将一直等到所指定的叉子可用，然后将其取用。不幸的是，这种解法是错误的。设想所有五位哲学家都同时拿起左面的叉子，则他们都拿不到右面的叉子，于是发生死锁。

```
# define  N  5                          /* 哲学家数目 */
void philosopher（int i）               /* i：哲学家号从0到4号 */
{
    while（TRUE）{
      think（）；                       /* 哲学家正在思考 */
      take_fork（i）；                  /* 取左面叉子 */
      take_fork（（i+1）%N）；              /* 取右面叉子；%为取余 */
      eat（）；                         /* 吃面 */
      put_fork（i）；                   /* 放回左面叉子 */
      put_fork（（i+1）%N）；               /* 放回右面叉子 */
   }
 }
```

图2-17 哲学家进餐问题的一种不正确解法。

我们可以将程序修改一下，规定在拿到左叉后，先检查右面的叉子是否可用。如果不可用，则先放下左叉，等一段时间再重复整个过程。但这种解法也是错误的，尽管与前一种的原因不同。可能在某一个瞬间，所有的哲学家都同时启动这个算法，拿起左叉，看到右叉不可用，又都放下左叉，等一会儿，又同时拿起左叉。。。如此这样永远重复下去。对于这种情况，即所有的程序都在运行，但却无法取得进展，就称为饥饿(starvation)。（即使问题不发生在意大利或中国餐馆也被称为饥饿）

现在你可能会想：如果哲学家在拿不到右叉时等待一段随机的时间，而不是等待相同的时间，则发生上述锁步的机会就很小了。这种想法是对的，但在一些应用中人们希望一种完全正确的方案，它不能因为一串靠不住的随机数字而失效（想想核电站中的安全控制系统）。

对图2－17中的算法可进行下列改进，它既不会发生死锁又不会发生饥饿：使用一个二进制信号量对五个think函数之后的语句进行保护。在哲学家开始拿叉子之前，先对信号量mutex执行DOWN。在放回叉子后，再对mutex执行UP。从理论上讲，这种解法是可行的。但从实际角度来看，这里有性能上的局限：同一时刻只能有一位哲学家进餐。而五把叉子实际上允许两位哲学家同时进餐。

图2－18中的解法不仅正确，而且对于任意位哲学家的情况都能获得最大的并行度。其中使用一个数组state来跟踪一个哲学家是在吃饭、思考还是正在试图拿叉子。一个哲学家只有在两个邻居都不在进餐时才允许转移到进餐状态。第i位哲学家的邻居由宏LEFT和RIGHT定义，换言之，若i为2，则LEFT为1，RIGHT为3。

```
 # define N 5                            /* 哲学家数目 */
# define LEFT （i-1+N）%N               /* i的左邻号码 */
# define RIGHT （i+1）%N                /* i的右邻号码 */
# define THINKING 0                     /* 哲学家正在思考 */
# define HUNGRY  1                      /* 哲学家想取得叉子 */
# define EATING   2                     /* 哲学家正在吃面 */
typedef int semaphore;                  /* 信号量是一个特殊的整型变量 */
int state[N];                           /* 记录每个人状态的数组 */
semaphore mutex=1;                      /* 临界区互斥 */
semaphore s[N];                         /* 每个哲学家一个信号量 */
void philosopher（int i）               /* i：哲学家号码，从0到N-1 */
{
  while（TRUE）{                        /* 无限循环 */
      think（）；                       /* 哲学家正在思考 */
      take_forks（i）；                 /* 需要两只叉子，或者阻塞 */
      eat（）；                         /* 进餐 */
      put_forks（i）；                  /* 把两把叉子同时放回桌子 */
  }
}
void take_forks（int i）                /* i：哲学家号码从0到N-1 */
{
  down（&mutex）；                      /* 进入临界区 */
  state[i]=HUNGRY;                      /* 记录下哲学家i饥饿的事实 */
  test（i）；                           /* 试图得到两只叉子 */
```

```
  up（&mutex）;                    /* 离开临界区 */
  down（&s[i]）;                      /* 如果得不到叉子就阻塞 */
}
void put_forks（int I）             /* i：哲学家号码从0到N-1 */
{
       down（&mutex）;                /* 进入临界区 */
  state[i]=THINKING;                /* 哲学家进餐结束 */
  test（LEFT）;                       /* 看一下左邻居现在是否能进餐 */
  test（RIGHT）;                      /* 看一下右邻居现在是否能进餐 */
  up（&mutex）;                       /* 离开临界区 */
}
void  test（i）                     /* i：哲学家号码从0到N-1 */
  {
        if（state[i]==HUNGRY&&state[LEFT]!=EATING
                          &&state[RIGHT]!=EATING）{
           state[i]=EATING;
           up（&s[i]）;
     }
}
```

图2-18 一个哲学家就餐问题的解决方案。

该程序使用了一个信号量数组，每个信号量对应一位哲学家，这样在所需的叉子被占用时，想进餐的哲学家可以阻塞。注意每个进程将过程philosopher作为主代码运行，而其他过程take_forks、put_forks和test只是普通的过程，而非单独的进程。

2.3.2 读者－写者问题

哲学家问题对于多个竞争进程互斥地访问有限资源（如I/O设备）这一类问题的建模十分有用。另一个著名的问题是读者－写者问题（Courtois et al., 1971），它为数据库访问建立了一个模型。例如，设想一个飞机定票系统，其中有许多竞争的进程试图读写其中的数据。多个进程同时读是可以接受的，但如果一个进程正在更新数据库，则所有其他进程都不能访问数据库，即使读操作也不行。这里的问题是如何对读者和写者进行编程？图2-19给出了一种解法。

```
typedef int semaphore;
semaphore mutex=1;                  /* 控制对RC的访问 */
semaphore db=1;                     /* 控制对数据库的访问 */
int rc=0;                           /* 正在读或想要读的进程数 */
void reader（void）
{
  while（TRUE）{                      /* 无限循环 */
     down（&mutex）;                  /* 排斥对RC的访问*/
     rc = rc+1;                     /*又多了一个读者*/
     if（rc == 1）down（&db）;         /*如果这是第一个读者，那么.....*/
     up（&mutex）;                  /*恢复对RC的访问*/
     read_data_base（）;            /*访问数据*/
     down（&mutex）;                /*排斥对RC的访问*/
     rc = rc -1;                    /*读者又少了一个*/
     if（rc==0）up（&db）;          /*如果这是最后一个读者，那么.....*/
     use_data_read（）;             /*非临界区操作*/
    }
 }
 void writer（void）
 {
    while（TRUE）{
      think_up_data（）;              /*非临界区操作*/
      down（&db）;                  /*排斥访问*/
      write_data_base（）;          /*修改数据*/
```

```
        up（&db）；                    /*恢复访问*/
      }
    }
```

图2-19 一个读者与作者问题的解决方案。

该解法中，第一个读者对信号量db执行DOWN。随后的读者只是递增一个计数器rc。当读者离开时，它们递减这个计数器，而最后一个读者则对db执行UP，这样就允许一个阻塞的写者（如果存在的话）可以访问数据库。

这个解法在这里隐含了一条很微妙的规则值得讨论。设想当一个读者在使用数据库时，另一个读者也来访问数据库，由于同时允许多个读者同时进行读操作，所以第二个读者也被允许进入，同理第三个及随后更多的读者都被允许进入。

现在假设一个写者到来，由于写操作是排他的，所以它不能访问数据库，而是被挂起。随后其他的读者到来，这样只要有一个读者活跃，随后而来的读者都被允许访问数据库。这样的结果是只要有读者陆续到来，它们一来就被允许进入，而写者将一直被挂起直到没有一个读者为止。假如每2秒钟来一个读者，而其操作时间为5秒钟，则写者将永远不能访问数据库。

为了防止这种情况，程序可以略作如下改动：当一个读者到来而正有一个写者在等待，则读者被挂在写者后边，而不是立即进入。这样，写者只需等待它到来时就处于活跃状态的读者结束，而不用等那些在它后边到来的读者。这种解法的缺点是并发性较低，从而性能较差。 Courtois等人给出了一个写者优先的解法。详细请参阅他的论文。

2.3.3 理发师睡觉问题

另一个经典的IPC问题发生在理发店里。理发店里有一位理发师、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，则理发师便在理发椅上睡觉，如图2-20所示。当一个顾客到来时，他必须先叫醒理发师，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。如果没有空椅子，他就离开。这里的问题是为理发师和顾客各编写一段程序来描述他们的行为，要求不能带有竞争条件。

图2-20 睡觉的理发师。

我们的解法使用三个信号量：customers，用来记录等候理发的顾客数（不包括正在理发的顾客）；barbers，记录正在等候顾客的理发师数，为0或1；mutex，用于互斥。我们也需要一个变量waiting，它也用于记录等候的顾客数，它实际上是customers的一份拷贝。之所以使用waiting是因为无法读取信号量的当前值。在该解法中，进入理发店的顾客必须先看等候的顾客数，如果少于椅子数，他留下来等，否则他就离开。

我们的解法示于图2-21。当理发师早晨开始工作时，他执行过程barber，这将导致他阻塞在信号量customers上，直到有人到来。这时他将去睡觉，如图2-20所示。

当一个顾客到来时，他执行过程customer，首先获取信号量mutex以进入临界区。如果不久另一位顾客到来，则他只能等到第一位释放了mutex后才能做别的事。顾客随后检查是否等候的顾客少于椅子数，如果不是，他就释放mutex并离开。

如果有一把椅子可坐，则他递增整型变量waiting，随后对信号量customers执行UP，然后唤醒理发师。此时顾客和理发师都处于清醒状态。当顾客释放mutex时，理发师获得mutex，他进行一些准备后开始理发。

当理完发后，顾客退出该过程，并离开理发店。与前边的例子不同，这里顾客不执行循环，因为每个顾客只需理一次发。但理发师必须执行循环以服务下一位顾客。如果有顾客，则为顾客理发，否则就去睡觉。

这里需要指出的是尽管读者－写者问题和理发师问题都不涉及数据传递，但它们仍属于IPC范围，因为它们涉及多进程间的同步。

```
# define CHAIRS   5                  /*为等待的顾客准备的椅子数*/
typedef  int semaphone ;                 /*运用你的想象力*/
semaphone  customers=0;              /*等待服务的顾客数*/
semaphone  barbers=0;                    /*等待顾客的理发师数*/
semaphone  mutex=1;                  /*用于互斥*/
int waiting=0;                           /*等待的顾客（还没理发的）*/
void barber（void）
{
   while（TRUE）{
     down（customers）；             /*如果顾客数是0，则睡眠*/
     down（mutex）；                     /*要求进程等待*/
     waiting=waiting-1;                  /*等待顾客数减1*/
     up（barbers）；                 /*一个理发师现在开始理发了*/
     up（mutex）；                   /*释放等待*/
     cut_hair（）；                      /*理发（非临界区操作）*/
    }
```

```
}
void customers（void）
{
   down（mutex）;                          /*进入临界区*/
   if（waiting<CHAIRS）{                  /*如果没有空的椅子，就离开*/
       waiting=waiting+1;                  /*等待顾客数加1*/
       up（customers）;                   /*如果必要的话，唤醒理发师*/
       up（mutex）;                       /*释放访问等待*/
       down（barbers）;                   /*如果barbers为0，就入睡*/
       get_haircut（）;                   /*坐下等待服务*/
   } else {
         up（mutex）;                     /*店里人满了，走吧*/
   }
}
```

图2-21 理发师问题的一种解法.

2.4进程调度

在前几节的例子中，我们经常遇到两个或多个进程（例如，生产者和消费者）在逻辑上均可以运行的情况。当有多个进程就绪时，操作系统必须决定先运行哪一个。操作系统中作出这种决定的部分称作调度程序(scheduler)，它使用的算法称作调度算法(scheduling algorithm)。

再回到早期以磁带上的卡片映像作为输入的批处理系统时代，那时的调度算法很简单：依次运行磁带上的下一个作业。对于分时系统，则调度算法要复杂一些，因为经常有多个用户等待服务，而且同时可能存在多个批处理流（例如，保险公司理赔）。即使在个人电脑上，也可能有若干用户启动的进程竞争CPU，更不要说还有后台作业，例如网络或收发电子邮件的精灵进程。

在讨论具体的调度算法之前，我们应该考虑一下调度程序要达到的目标。毕竟调度程序关注的是确定策略，而并不提供机制。一个好的调度算法应当考虑很多方面，其中可能有：

1 公平 - 确保每个进程获得合理的CPU份额。
2 有效 - 使CPU百分之百地忙碌。
3 响应时间 - 使交互用户的响应时间尽可能短。
4 周转时间 - 使批处理用户等待输出的时间尽可能短。
5 吞吐量 - 使每小时处理的作业数最多。

对这些目标稍加思考便会发现其中有矛盾之处。为了使交互用户的响应时间最短，应该对批处理作业不予调度（除了凌晨3点到6点，当用户休息时）。然而批处理用户可能不欢迎这种算法，它违背了条件4。可以证明（Kleinrock，1975）任何一个偏向某些类型作业的调度算法必将损害另一些作业。毕竟可利用的CPU时间是有限的。多给一个用户就要少给另一个用户，生活就是这样。

调度程序必须面对的另一个麻烦是每个进程都不一样，而且不可预测。有些进程花费很多时间等待I/O，而另一些进程在允许的条件下将连续使用CPU达几个小时。 当调度程序启动运行某些进程时，它根本不知道进程在阻塞前要运行多久（阻塞可能是因为I/O、信号量、或者其他原因）。为了保证不让进程运行得太久，几乎所有的计算机都内置一个电子定时器或时钟，它将定期地发出中断，通常每秒钟50或60次（亦称作50或60赫兹，简写为Hz）。但在许多计算机上，操作系统能够根据需要将时钟频率设置成任意值。每发生一次时钟中断，操作系统都将运行，并决定当前进程是否应继续运行，还是它已经占用了足够长的CPU时间，应该暂停让其他进程运行。

允许将逻辑上可运行的进程暂时挂起的策略称作可剥夺调度(preemptive scheduling)，这与早期批处理系统使进程运行直到结束的方法正好相反。运行直到结束的调度方式称作非剥夺调度(nonpreemptive scheduling)。正如我们在本章中看到的，进程可在任意时刻被不加警告地挂起，以便让另一个进程运行。这导致了竞争条件以及防止竞争条件的信号量、管程、消息或其他复杂的方法。另一方面，允许一个进程运行它所希望的时间意味着一个计算圆周率小数点后边十亿位的进程将使其他进程永远得不到服务。

所以尽管非剥夺调度算法简单且易于实现，但它通常不适于具有多个竞争用户的通用系统。另一方面，对于专用系统，如一个数据库服务器，主进程在收到请求时启动一个子进程并让其运行直到结束或阻塞则是很合理的。这种系统与通用系统的区别在于数据库系统中的所有进程都处于单个主进程的控制之下，该主进程知道各子进程将做什么，以及将花费的时间。

2.4.1 时间片轮转调度

现在让我们来看具体的调度算法。一种最古老，最简单，最公平且使用最广的算法是时间片调度。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。时间片轮转调度很容易实现。调度程序所要做的就是维护一张就绪进程列表，如图2-22（a）所示，当进程用完它的时间片后，

它被移到队列的末尾，如图2-22（b）所示。

图2-22 时间片轮转调度。 a）就绪进程列表。 （b）进程B用完它的时间片后的就绪进程列表。

时间片轮转调度中唯一有趣的一点是时间片的长度。从一个进程切换到另一个进程是需要一定时间的 - 保存和装入寄存器值及内存映像，更新各种表格和队列等。假如进程切换(process switch) - 有时称为上下文切换(context switch)，需要5毫秒。再假设时间片设为20毫秒。则在做完20毫秒有用的工作之后，CPU将花费5毫秒来进行进程切换。CPU时间的20%被浪费在了管理开销上。

为了提高CPU效率，我们可以将时间片设为500毫秒。这时浪费的时间只有1%。但考虑在一个分时系统中，如果有十个交互用户几乎同时按下回车键，将发生什么情况？这时十个进程被挂在就绪队列中，如果CPU空闲，则立即启动第一个进程，第二个进程在大约1/2秒之后启动，依次类推。假设所有其他进程都用足它们的时间片的话，最后一个不幸的进程不得不等待5秒钟才获得运行机会。多数用户无法忍受一条简短命令要5秒钟才能做出响应。同样的问题在一台支持多道程序的个人计算机上也会发生。

结论可以归结如下：时间片设得太短会导致过多的进程切换，降低了CPU效率；而设得太长又可能引起对短的交互请求的响应变差。将时间片设为100毫秒通常是一个比较合理的折衷。

2.4.2 优先级调度

时间片调度做了一个内在的假设，即所有的进程同等重要. 而拥有和操作多用户计算机系统的人对此常有不同的看法. 在一所大学里，等级顺序可能是教务长、教授、秘书、勤务人员、最后是学生. 这种将外部因素考虑在内的需要就导致了优先级调度. 其基本思想很清楚：每个进程被赋予一个优先级，优先级最高的就绪进程被率先运行.

即使在只有一个用户的PC机上，也可能有多个重要程度不同的进程。例如，一个在后台收发电子邮件的精灵进程应被赋予一个较低的优先级，而在屏幕上实时地播放电影的进程则应赋予较高的优先级。

为了防止高优先级进程无休止地运行下去，调度程序可能在每个时钟滴答（即每一个时钟中断）降低当前进程的优先级。如果这个动作导致其优先级低于次高优先级，则将进行进程切换。或者给每个进程设定一段它能够连续使用CPU的时间片，一旦这段时间用完，则运行次高优先级的进程。

优先级可以为静态或动态。在一台军用计算机上，将军启动的进程优先级为100，上校为90，少校为80，上尉为70，中尉为60，依次类推。或者在一个商业计算中心，高优先级作业每小时费用为100美元，中等优先级每小时75美元，低优先级每小时50美元。UNIX系统中有一条命令nice，它允许用户为了照顾别人而自愿降低其进程的优先级，但从未有人用它。

优先级也可以被系统动态地确定以达到某种目的。例如，有些进程为I/O密集型，其多数时间用来等待I/O结束。当这样的进程需要CPU时，它应被立即分配CPU，以便启动下一个I/O请求，这样就可以在另一个进程计算的同时执行I/O操作。使这类进程长时间等待CPU只会造成它无谓地长时间占用内存。使I/O密集进程获得较好服务的一种简单算法是将其优先级设为1/f，f为该进程上一时间片中计算时间所占的比重。一个在100毫秒中计算只占用2毫秒的进程优先级为50，而计算占用50毫秒的进程优先级则为2。全部时间都在计算的进程则为1。

很方便就可以将一组进程按优先级分成若干类。在各类之间采用优先级调度，而各类进程内部采用时间片轮转调度。图2-23显示了一个有4类优先级的系统，调度算法如下：只要存在优先级为第4类的就绪进程，就按照时间片轮转法使其运行一个时间片，此时不理会较低优先级的进程。若第4类进程为空，则运行第3类进程。若第4、第3类均为空，则按时间片法运行第2类进程。如果对优先级不经常进行调整，则低优先级进程很可能会发生饥饿。

图2-23 一个有四类优先级的调度算法。

2.4.3 多重队列

CTSS（Corbato et al., 1962）是最早使用优先级调度的系统之一。但是CTSS存在进程切换速度太慢的问题，其原因是IBM 7094内存中只放得下一个进程，每次切换都需要将当前进程换出到磁盘，并从磁盘上读入一个新进程。CTSS的设计者很快便认识到为CPU密集的进程设置较长的时间片，比频繁地分给它们很短的时间片要高效（减少交换次数）。另一方面，如前所述，给进程长时间片又会影响响应时间。他们的解决办法是设立优先级类。属于最高级类的进程运行一个时间片，属于次高优先级类的进程运行2个时间片，再次一级运行4个时间片，依次类推。当一个进程用完分配的时间片后，它被移到下一类。

我们看一个例子，有一个进程需要连续计算100个时间片。它最初被分配1个时间片，然后被换出。下次它将获得2个时间片，接下来分别是4、8、16、32和64。当然最后一次它只使用64个时间片中的37个便可以结束工作。该过程需要7次交换（包括最初的装入），而如果采用纯粹的时间片轮转则需要100次交换。而且，随着进程优先级的不断降低，它的运行频度逐渐放慢，从而为短的交互进程让出CPU。

对于那些刚开始运行一段长时间，而后来又需要交互的进程，为了防止其优先级降低过快可以采取这样的策略：只要终端上有回车键按下，则属于该终端的所有进程都被移到最高优先级，这样做的原因是认为此时进程即将需要交互。但可能有一天，一台重载的机器上有几个用户偶然发现，只需坐在那里每过随机的几秒钟敲一下回车键就可以大大地提高响应速度，于是他又告诉所有的朋友。这件事的结论是：实践中行得通比理论上可行要困难得多。

已经有许多其他算法可用来将进程划分为优先级类。例如，在伯克利制造的著名的XDS 940系统中

（Lampson，1968），它有4个优先级类，分别是终端、I/O、短时间片和长时间片。当一个等待终端输入的进程最终被唤醒时，它转到最高优先级类（终端）。当一个等待一块磁盘数据的进程就绪时，它将转到第二类。当进程在时间片用完时仍为就绪时，它被放入第三类。但如果一个进程已经多次用完时间片而从未因终端或其他I/O阻塞，它将被转入最低优先级类。许多其他系统也使用类似的算法，以侧重交互用户和交互进程，而牺牲后台进程。

2.4.4 最短作业优先

上述多数算法都是为交互系统设计的。现在来看一种适用于运行时间可以预知的批作业的调度算法。例如一家保险公司，因为每天都做类似的工作，所以人们可以相当精确地预测一个处理1000起索赔的作业需要多长时间。当输入队列中有若干个同等重要的作业将被启动时，调度程序应使用最短作业优先算法。请看图2-24，这里有4个作业A、B、C、D，运行时间分别为8、4、4、4分钟。若按图中的次序运行，则A的周转时间为8分钟，B为12分钟，C为16分钟，D为20分钟，平均为14分钟。

图2-24 一个最短作业优先调度的例子。

现在考虑使用最短作业优先算法。如图2-24（b）所示。现在周转时间分别为4、8、12和20分钟，平均为11分钟。可以证明最短作业优先是最优的。考虑有4个作业的情况，其运行时间分别为a，b，c，d。 第一个作业在时间a结束，第二个在时间a+b结束，依次类推。平均周转时间为（4a+3b+2c+d）/4，显然a对平均值影响最大，所以它应是最短作业，其次是b，再次是c，最后的d只影响它自己的周转时间。对任意数目作业的情况道理完全一样。

由于最短作业优先常常伴随着最短响应时间，所以如果它同时能够被用于交互进程，那将是非常好的。在某种程度上，它的确可以做到这一点。交互进程通常遵循下列模式：等待命令，然后执行命令。如此不断反复。如果我们将每一条命令的执行看作一个独立的作业，则我们可以通过首先运行最短的作业来使响应时间最短。唯一的问题是如何从当前就绪进程中找出最短的那个。

一种办法是根据进程过去的行为进行推测，并执行估计运行时间最短的那个。假设某终端上每条命令的估计运行时间为T0，现在假设测量到其下一次运行时间为T1，我们可以将这两个值的加权和来改进我们的估计时间，即aT0+（1-a）T1。通过选择a的值，我们可以决定是尽快忘掉老的运行时间，还是在一段长时间内还记住它们。当a=1/2时，我们可以得到如下序列：

T0，T0/2+T1/2，T0/4+T1/4+T2/2，T0/8+T1/8+T2/4+T3/2

我们看到，三轮过后，T0在新的估计值中的占的比重下降到1/8。

这种通过将当前测量值和先前估计值进行加权平均而得到下一个估计值的技术有时称作老化。它适用于许多预测值必须基于先前值的情况。老化算法在a=1/2时特别容易实现，只需将新值加到当前估计值上然后除以2（即右移一位）。

需要指出的是，最短作业优先算法只在所有作业同时可用的情况下才是最优的。作为一个反例，考虑5个作业，A到E，运行时间分别为2，4，1，1，1；到达时间分别为0，0，3，3，3。

最初只有A和B可被选择，因为另外三个作业尚未到达。使用最短作业优先将按照A，B，C，D，E的顺序运行，其平均等待时间为4.6。而按照B，C，D，E，A的顺序则平均等待时间为4.4。

2.4.5 保证调度算法

一种完全不同的调度算法是向用户作出明确的性能保证，然后去实现它。一种很实际并很容易实现的保证是：若你工作时有n个用户登录，则你将获得CPU处理能力的1/n。类似的，在一个有n个进程运行的单用户系统中，若所有的进程都平等，则每个进程将获得1/n的CPU时间。

为了实现其所作的保证，系统必须跟踪各进程自创建以来已使用了多少CPU时间。然后它计算各进程应获得的CPU时间，即自从创建以来的时间除以n。由于各进程实际获得的CPU时间已知，所以很容易计算出真正获得的CPU时间和应获得的CPU时间之比。比率为0.5说明一个进程只获得了应得时间的一半，而比率为2.0则说明它获得了应得时间的2倍。于是该算法随后转向比率最低的进程，直到该进程的比率超过次最低进程为止。

2.4.6 彩票调度算法

尽管向用户作出承诺并履行它是一个好主意，但实现却很困难。不过有另一种算法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法（Waldspurger和Weihl 1994）。

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。当调度程序需作出决策时，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，每次的中奖者获得20毫秒的运行时间。

George Orwell对此解释为：“所有进程都是平等的，而某些进程则需要更多的机会。”更重要的进程被给予更多的额外彩票，以增加其中奖机会。如果共发出100张彩票，而一个进程持有20张，它就有20%的中奖概率，对于长时间运行，它将获得大约20%的CPU时间。彩票算法与优先级调度完全不同，在后者中很难说清楚优先级为40说明了什么，而在前者则很清楚：进程拥有多少彩票份额，它就将获得多少资源。

彩票调度法有几点有趣的特性。例如，如果一个新进程创建并得到一些彩票，则在下次抽奖时，它中奖的机会与其持有的彩票数成正比。换言之，彩票调度的反应非常迅速。

合作进程如果愿意的话可以交换彩票。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

以把所有的彩票都交给服务器进程，以增加其下一次被运行的机会。当服务器进程结束后，它又将彩票交还给客户进程以使其能够再次运行。实际上，在没有客户时，服务器根本不需要彩票。

彩票调度能用来解决其他算法难以解决的问题。例如，一个视频服务器，其中有若干个进程将视频信息传送给各自的客户，但帧频不同。假设其分别需要为10、20和25帧/秒的速度，则通过为这些进程分别分配10、20和25张彩票，它们将自动地按照正确的比例分配CPU资源。

2.4.7 实时调度

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作出适当的反应。一个实例是，计算机用CD-ROM放音乐时从驱动器获得二进制数据，并且必须在很短的时间内将其转换成音乐。如果这中间计算花的时间太长，音乐听起来就会失真。其他实时系统还包括：医院里特护病房的监控系统、飞行器中的自动驾驶仪、以及核反应堆中的安全控制系统等。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是可以容忍的。这两种系统中，实时性的获得是通过将程序分成许多进程，而每个进程的行为都预先可知。这些进程通常生存周期都很短，往往在一秒内便运行结束。当检测到一个外部事件时，调度程序按满足它们最后期限的方式调度这些进程。

实时系统要响应的事件进一步分为周期性（每隔一段固定的时间发生）和非周期性（在不可预测的时间发生）。一个系统可能必须响应多个周期的事件流。根据每个事件需要多长的处理时间，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Pi，其中每个事件需要Ci秒的CPU时间来处理。则只有满足以下条件

$$\sum_{i=1}^{m} \frac{Ci}{Pi} < 1$$

时，才可能处理所有的负载。满足该条件的实时系统称作是可调度的(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100，200和500毫秒。如果事件处理时间分别为50，30和100毫秒，则这个系统是可调度的，因为0.5+0.15+0.2＜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150毫秒，该系统仍将是可调度的。这个运算的隐含条件是上下文切换的开销很小，可以忽略。

实时调度算法可以是动态或静态。前者在运行时作出调度决定，后者在系统启动之前完成所有的调度决策。我们简要地考虑几种实时调度算法。经典的算法是发生率单调算法（Liu和Layland，1973）。该算法事先为每个进程分配一个与事件发生频率成正比的优先级。例如，周期为20毫秒的进程优先级为50，而周期为100毫秒的进程为10。运行时，调度程序总是调度优先级最高的就绪进程，必要时将剥夺当前进程。Liu和Layland证明了该算法是最优的。

另一种流行的实时调度算法是最早截止优先算法。当一个事件发生时，对应的进程被加到就绪队列中。该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次发生的时间。该算法首先运行队首进程，即截止时间最近的那个。

第三种算法首先计算各进程的富余时间，称做作裕度(laxity)。如果一个进程需要运行200毫秒，而它必须在250毫秒内完成，则其裕度为50 毫秒，该算法称作最少裕度法，即选择裕度最少的进程。

尽管在理论上通过使用这三种调度算法中的一种可以将一个通用操作系统转变为一个实时系统，但实际上，通用操作系统的上下文切换开销太大，以至于只对那些时间限制较松的应用才能达到其实时性能要求。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等。

2.4.8 两级调度法

到目前为止，我们或多或少总是假设就绪进程都在内存中。如果没有足够的内存，则某些就绪进程将全部或部分地被放在磁盘上，这种情况对调度有很大影响，因为从盘上读入一个进程运行比单纯在内存中进行进程切换要慢几个数量级。

处理被对换到外存的进程可以使用一种更实际的办法，即使用两级调度。就绪进程的一个子集首先被装入内存，如图2-25（a）所示。调度程序在随后的一段时间里只在这个子集中进行调度。一个高级调度程序周期性地将那些在内存中驻留时间足够长的进程换出，而将那些在磁盘上等候时间过长的进程换入。当这个操作完成后，如图2-25（b）所示，低级调度程序再次在那些驻留在内存中的进程间进行调度。这样，低级调度程序只关心当时在内存中的就绪进程，而高级调度程序则关心将进程在内存和磁盘间来回交换。

图2-25 两级调度程序必须将进程在磁盘和内存间来回移动，并从驻留在内存的进程中进行选择。 三个不同时刻的情况示于（a）、（b）、（c）。

高级调度程序用于决策的标准包括：

1 进程自上次被换入或换出以来的时间

2 进程最近使用的CPU时间
3 进程的大小（小进程不参与高级调度）
4 进程的优先级

这里我们可以再次用到时间片轮转、优先级调度，或其他各种算法。高级和低级调度程序可以使用相同或不同的算法。

2.4.9 策略与机制

截至目前，我们隐含地假设系统中所有进程分属不同的用户，并且相互间竞争CPU。尽管通常是这样，但有时会有这样的情况：一个进程有许多子进程在其控制之下运行。例如，一个数据库管理系统可能有许多子进程，每个子进程可能处理不同的请求，或者每个子进程实现不同的功能（请求的语法分析，访问磁盘等）。主进程完全可能掌握哪个子进程最重要（或最紧迫），哪个最不重要。但不幸的是，以上讨论的调度算法中没有一个从用户进程接受有关的调度决策信息。这就导致调度程序很少能够作出最优的选择。

解决该问题的方法是将调度机制(scheduling mechanism)和调度策略(scheduling policy)分开。亦即将调度算法以某种形式参数化，而参数可以由用户进程来填写。我们再来看数据库的例子。假设核心使用优先级调度算法，但提供一条系统调用，一个进程可以使用它来设置或改变其子进程的优先级。这样尽管父进程本身并不进行调度，但它可以控制子进程如何被调度的细节。在这里，机制位于核心，而策略则由用户进程设定。

2.5 MINIX进程概述

在研究了进程管理、进程间通信以及进程调度的原理之后，我们现在来看这些原理在MINIX中的应用。MINIX与UNIX不同，UNIX的核心是一个不分模块的单块程序，而MINIX本身就是一组进程的集合，它们相互之间、以及与用户进程之间使用进程间通信机制 - 消息传递来进行通信。这种设计使得MINIX的结构更加模块化和灵活。例如，这使得很容易就可以将整个文件系统替换成另一个完全不同的文件系统，而无需重新编译核心。

2.5.1 MINIX的内部结构

开始研究MINIX之前，我们首先大概浏览一下整个系统。MINIX被组织成4层，每一层执行一套定义得很完好的功能。这四层示于图2-26。

图2-26 MINIX的四层结构。

最底层捕获所有的中断和陷入，完成进程调度，并向高层提供一个采用消息进行通信的独立顺序进程模型。该层的代码有两大主要功能。第一是捕获陷入和中断、保存和恢复寄存器、调度以及向高层提供一个独立顺序进程模型。第二是处理消息机制：检查目标进程的合法性、定位物理内存中的发送和接收缓冲区、以及从发送方向接收方拷贝数据。其中中断处理的最底层部分用汇编语言编写，其余部分和其他层次用C语言编写。

第二层包括I/O进程，每类设备都有一个I/O进程。为了将其与普通用户进程相区别，我们称之为任务(tasks)。但任务与进程间的差别微乎其微。在许多系统中I/O任务被称作设备驱动程序(device driver)。这里“任务”和“设备驱动程序”可以换用。每一类设备都需要一个任务，包括磁盘、打印机、终端、网络接口以及时钟。如果有其他I/O设备，则它们也需要相应的任务。有一个任务 - 系统任务有些与众不同，它不对应于任何I/O设备，我们将在下一章对这些任务进行讨论。

第二层的所有任务和第一层的代码链接成一个单一的二进制程序，称作核心(kernel)。某些任务共享公共的子例程，但它们相互之间完全独立，分别进行调度，并采用消息进行通信。从286开始的Intel处理器为每个进程赋予四种特权级中的一种。尽管任务与核心被编译在一起，但当核心和中断处理程序被执行时，它们被赋予较任务更高的特权级。所以真正的核心代码可以访问任一部分内存，以及任一处理器寄存器 - 实质上，核心可以使用系统中任何地方的数据执行任何指令。任务不能执行全部的机器指令，也不能访问所有CPU寄存器或所有的内存。但在为较低特权级的进程执行I/O时，任务可以访问属于这些进程的内存区域。有一个任务 - 系统任务，它并不执行一般意义的I/O，其作用提供某些特定服务，例如当进程本身不允许在不同的内存区域间进行拷贝时，由系统任务执行此操作。当然在不提供多特权级的机器上，例如老式的Intel处理器，无法强制执行这些限制。

第三层包含向用户进程提供有用服务的进程。这些服务器进程在低于核心和任务的特权级上运行，不能直接访问I/O端口。它们也不能访问属于自己的段以外的内存。内存管理器（Memory Manager - MM）负责执行所有牵涉到内存管理的系统调用，如FORK、EXEC和BRK。文件系统（File System - FS）负责执行文件系统的调用，READ、MOUNT和CHDIR。

正如我们在第一章开头所指出的，操作系统做两件事情：管理资源和通过系统调用方式提供扩展的计算机。在MINIX中，资源管理主要在核心（第1、2层）中，系统调用的解释在第三层。文件系统被设计成一个“服务器”，并几乎可以不加修改地移到一台远程机器上。这也适用于内存管理器，尽管远程内存管理器不如远程文件系统那样有用。

第三层也可能存在其他的服务器。图2 - 26显示出那里有一个网络服务器。尽管本书中描述的MINIX不包括网络，但网络的源码却是标准MINIX发布软件的一部分。 系统很容易被重新编译以包括网络。

此处正适于指出尽管服务器是独立的进程，但它们和用户进程有一点不同，即它们在系统启动的同时被启动，而且在系统活跃期间不会终止。此外，尽管从它们禁用的机器指令来看，它们与用户进程运行在相同的

特权级上，但它们的执行优先级比用户进程高。为了加入一个新的服务器，核心必须重新编译。核心的启动代码在用户进程开始运行之前将服务器进程安装在进程表的特权表项中。

最后，第四层包含所有的用户进程 －shell、编译器、编辑器以及用户的a.out程序。一个运行系统通常有一些进程在系统引导时启动，并一直运行，例如，一个精灵程序就是一个周期性运行或总是等待某个事件（例如网络上一个包的到达）的后台进程。从某种意义上说，精灵进程是一个单独启动而作为一个用户进程运行的服务器。但与装入特权进程表项的真正的服务器不同，这些程序无法象内存和文件服务器那样受到核心的特殊对待。

2.5.2 MINIX中的进程管理

MINIX中的进程遵从本章前边所描述的通用进程模型。进程可以创建子进程，子进程又可以创建更多的子进程，这样便构造出一棵进程树。实际上，整个系统中所有的用户进程都属于以init（见图2 － 26）为根节点的一棵进程树。

这种情况是怎样产生的呢? 当计算机开机时,硬件从引导盘上将第一道第一扇区读入内存并从那里开始执行。具体细节根据引导盘是软盘还是硬盘而不同。在软盘上，第一扇区包含了引导程序(bootstrap)。引导程序很小，因为它必须能容纳在一个扇区里。MINIX引导程序装入一个更大的程序boot，由boot装入操作系统。

相比之下，硬盘需要一个中间步骤。硬盘被分成若干分区(partition)，整个硬盘的第一个扇区包括一段小程序和磁盘分区表，通常称为主引导记录。程序部分被执行以读入分区表并选择活跃分区。活跃分区的第一个扇区有一个引导程序，它随后被装入并执行以查找并启动程序boot，这与从软盘引导完全相同。

不管哪种情况，boot将在软盘或硬盘分区上找一个包含多个部分的文件，并将各部分装到内存的适当位置。这些部分包括核心、内存管理器、文件系统以及init－ 第一个用户进程。这个启动过程并不简单。所有那些属于磁盘任务和文件系统范围的操作在这两部分活跃之前都要由boot完成。在后边我们将回到MINIX如何启动这一个主题。现在只需知道一旦装入操作完成，核心便开始运行。

在其初始化阶段，核心先启动各任务，然后是内存管理器、文件系统及所有在第三层运行的服务器。当所有这些都开始运行并完成初始化之后，它们将阻塞，等待执行某种操作。当所有的任务和服务器被阻塞之后，将执行第一个用户进程 －init。init已经位于内存，不过在它启动时其他所有部分均已就位，所以它可以作为一个独立的程序从磁盘上装入。但是，由于init只启动一次，而且不会再从盘上装入，所以最简便的方法是把它与核心、任务和服务器一起包括在系统的映像文件中。

init启动后先读/etc/ttytab文件，该文件列出了所有可能的终端设备。那些可以作为登录终端（在标准MINIX中，只包括控制台）的设备都在/etc/ttytab的getty域有一项。而且init为每个这样的终端创建一个子进程。通常每个子进程都执行文件/usr/bin/getty，打印出一条信息，然后等待输入一个用户名。随后将该用户名作为参数来调用 /usr/bin/login。如果某个终端需要特殊的处理（例如，一条拨号线），则/etc/ttytab可以指定一条命令（如/usr/bin/stty），在执行getty之前执行该命令并对线路进行初始化。

在成功地登录之后，/bin/login执行用户的shell（在/etc/passwd文件中指定，通常是/bin/sh或/usr/bin/ash）。shell等待用户键入命令，并为每条命令创建一个新的进程。采用这种方式时，各个shell都是init的子进程，而用户进程则是init的孙子进程，并且所有的用户进程都是一棵进程树的组成部分。

MINIX用于进程管理的两条最重要的系统调用是FORK和EXEC。FORK是创建一个新进程的唯一途径。EXEC允许一个进程执行一个指定的程序，当一个程序被执行时，将按照文件头中指定的大小为其分配一部分内存。尽管在进程运行期间，数据段、栈段和空闲未使用部分的大小可以不时地改变，但进程分配到的内存总量将保持不变。

一个进程的所有信息被保存在进程表中，进程表划分成核心、内存管理器和文件系统三部分，分别拥有它们各自所需要的那些域。当出现一个新进程（通过FORK），或者一个老进程结束（通过EXIT或信号）时，内存管理器首先更新它那部分进程表，然后向文件系统和核心发送消息，以通知它们进行相应的操作。

2.5.3 MINIX中的进程间通信

MINIX提供了三条原语来发送和接收消息，它们均通过C库例程调用。其中

send（dest，&message）

用来向进程dest发送一条消息，

receive（source，&message）

用来从进程source（或任何地方）接收一条消息，

send_rec（src_dst，&message）

用来发送一条消息，并等待同一个进程的应答。以上调用中第二个参数是消息数据的本地地址。核心中的消息传递机制将消息从发送者拷贝到接收者。应答消息（对于send_rec）将覆盖原先的消息。原则上该核心机制可以替换为另一套机制以实现分布式系统，即在网络上将消息从一台机器拷贝到另一台机器上。但在实践中这很复杂，因为有时消息的内容可能是一个指向大型数据结构的指针，于是分布式系统也必须提供网络上数据本身的拷贝功能。

每个进程或任务都可以从/向同层和下一层中的进程或任务发送和接收消息，用户进程不能直接与I/O任务通信，系统强制地执行这一限制。

当一个进程（作为特例，这里也包括任务）向一个当前未在等待消息的进程发送一条消息时，发送者将阻塞，直到目标进程执行receive。换言之，MINIX使用会合的方法来避免对已发送而未接收到的消息进行缓冲的问题。尽管这没有带缓冲的方案灵活，但事实证明对MINIX来说它已经足够了，而且由于不需要缓冲管理，所以简单许多。

2.5.4 MINIX中的进程调度

中断系统使多道程序操作系统持续不断地工作。当进程请求输入时，它们将阻塞以允许其他进程执行。当输入可用时，当前运行进程被磁盘、键盘或其他硬件中断。时钟也产生中断，这种中断使正在运行的未请求输入的用户进程最终放弃CPU，以使其他进程获得运行的机会。MINIX最底层软件的任务就是通过将中断转换成消息来对其加以隐藏。就进程（以及任务）而言，当一个I/O设备完成一个操作时，它向某些进程发送一条消息，将其唤醒并使之成为就绪。

每当一个进程被中断时，不管中断源是常规的I/O设备还是时钟，都有机会重新确定哪个进程最需要运行机会。当然，在一个进程终止时也要执行该操作，但在类似MINIX这样的系统中，由I/O操作和时钟引起的中断远远多于进程终止的情况。MINIX调度程序使用一个三级排队系统，分别对应于图2-26中的第2、3、4层。任务和服务器一级的进程一直运行直到阻塞，而用户进程则采用时间片轮转调度。任务具有最高优先级，内存管理器和文件管理器次之，用户进程最低。

当调度程序选择一个进程来运行时，它首先检查是否有就绪的任务，如果有一个或多个，则队首的那个将运行。如果没有任务就绪，则检查并运行服务器进程（MM或FS）。若没有合适的服务器进程，则运行一个用户进程。如果没有进程就绪，则选择IDLE进程。这个循环一直执行到下一个中断到来。

在每一个时钟滴答，都将检查当前进程是否是一个运行超过100毫秒的用户进程。如果是，则调用调度程序来查看是否有另一个用户进程在等待CPU，如果发现一个这样的进程，则当前进程被移到队列的末尾，而运行当前的队首进程。任务、内存管理器和文件系统不会被时钟剥夺，不论它们已运行了多久。

2.6 MINIX中进程的实现

现在我们来说明实际代码，先介绍一下将要用到的几个术语。“过程”、“函数”以及“例程”这几个词可以混用。变量名、过程名以及文件名将用斜体，如rw_flag。当一条语句用变量名、过程名或文件名开头时，它将使用大写，但真正的名字都使用小写字母。系统调用使用小一点的大写字母，如READ。

由于这本书和软件是一直在不断演变的，而不是在同一天交付出版，所以在代码引用、打印出的代码清单以及CD－ROM上的软件之间可能会有小的出入。但这种差异一般只影响一、两行代码。本书中印刷出来的源代码已经被简化，从中删去了那些与本书未讨论的选择项相关联的部分。

2.6.1 MINIX源代码的组织

源代码从逻辑上分成两个目录。在一个标准的MINIX系统中，其完整的路径分别为/usr/include/和/usr/src/ （一个缀后的“/”表示这是一个目录）。目录和真实路径可能因系统而异，但一般来说最高层以下的目录结构在所有系统中都是一样的。本书中我们记作 include/ 和 src/。

include/ 目录包含了许多符合POSIX标准的头文件，它又包含三个子目录：

1 sys/ — 包含POSIX头文件

2 minix/ — 包含操作系统使用的头文件

3 ibm/ — 包含IBM PC特有定义的头文件

为了支持对MINIX和在MINIX环境下运行程序的扩展，在CD－ROM上同时提供了放在include/目录下的其他文件和子目录，这些也可在Internet上取到。例如， include/net目录及其子目录 include/net/gen 支持网络扩展。但本书中仅印出了编译MINIX基本系统所需的文件，并且只讨论这些内容。

src/ 目录包含三个重要的子目录，其中包括了操作系统的源代码：

1 kernel/ — 第1层和第2层（进程、消息和驱动程序）

2 mm/ — 内存管理器代码

3 fs/ — 文件系统代码

还有另外三个源代码目录在本书中未列出，也不加讨论。但对于构造一个能够工作的系统，它们是很重要的。

1 src/lib/ — 库例程源代码（如open，read）

2 src/tools/ — init程序源代码，用于启动MINIX

3 src/boot/ — 引导和安装MINIX的代码

MINIX的标准发布软件还包括另外几个源代码目录。操作系统当然要支持在其上运行的命令（程序）。所以有一个很大的命令目录src/commands/，其中包含公用程序（如cat，cp，date，ls，pwd）的源代码。由于MINIX是一个用于教学的操作系统，这意味着对它常常要作修改，所以有一个 src/test/目录包含有一些被设计用来对新编译好的MINIX系统进行完整测试的工具。最后 /src/inet/目录包含了重新编译MINIX以使之支持网络的源代码。

为了方便起见，在通过上下文能够确定完整路径的情况下，将只使用简单的文件名。但必须指出有效文件名可能出现在不只一个目录下。例如存在若干名为const.h的文件，其中定义了系统组成部分所用到的常数。一个目录下的文件将放在一起讨论，这样就不会发生混淆。这些文件在附录A中按照其在教材中讨论的顺序罗列，

这样更易于掌握。在这里若使用一些书签将很有帮助。

同样应该指出的是，附录B包含附录A中描述的所有文件的字母序列表。附录C则包含一个索引，从中可查出MINIX所使用的宏定义、全局变量以及过程的出处。

第1、2层的代码位于 src/kernel/下。本章研究在该目录下的一些文件，它们支持进程管理，这是图2-26中所示的MINIX结构的最低层。该层包括完成以下功能的函数：系统初始化、中断、消息传递以及进程调度。在第3章我们将研究该目录下的其他文件，它们支持图2－26中第2层的不同任务。在第4章我们将研究 src/mm/下的内存管理器。第5章将研究 src/fs/下的文件系统。

编译MINIX时，位于 src/kernel/、src/mm/和src/fs/下的所有源文件都被编译成目标文件。 src/kernel/下的所有目标文件被连接成一个单一的可执行程序 kernel。在src/mm/中的目标文件也被链接成一个可执行程序 mm，fs的情况也一样。通过增加另外的服务器进程可以达到扩展的目的。例如通过修改文件include/minix/config.h便可以将 src/inet/下的文件编译成 inet，从而增加网络支持。另一个可执行程序 init在 src/tools/ 目录下生成。程序 installboot （其源码在 src/boot/ 下）将名字加入上述程序中，通过填充以使其长度变成磁盘扇区大小的整数倍（这样可以在装入这些不同部分时相互独立），并且将它们拼接为一个单独的文件。新文件就是操作系统的二进制映像，并且可被拷贝到根目录、或一张软盘的 /minix/ 目录、或一个硬盘分区上。图2－27示出了这些拼接的文件被分开和装入后的内存布局。当然，具体细节依赖于系统的配置。图中的例子是一个为具有几兆字节内存的机器配置的MINIX系统。这种配置可以分配大量的文件系统缓冲区，但所得的大文件系统无法被放在内存的低端部分，即640K以下的内存。如果大量地缩减缓冲区数目，则有可能使整个系统占用不到640K的内存，还可以为几个用户进程留出空间。

图 2-27 MINIX从磁盘装入内存之后的内存布局。四个单独编译并被链接的部分
（若加上网络则为五部分）划分得很清楚。图中各部分的大小只是大概情况，
具体则依赖依赖于系统配置状况。

很重要的一点是必须认识到MINIX包含三个或更多的独立程序，其间通过消息传递进行通信。 src/fs/ 下的一个名为 panic的过程不会与 src/mm/ 下的 panic 过程冲突，因为它们最终被链入不同的可执行文件。操作系统的三个部分之间仅有的共同的例程是 lib/ 下的若干库例程。这种模块化结构使得很容易做到对文件系统的修改不会影响内存管理器。同时也可以做到从系统中去掉文件系统，并将其放到另一台作为文件服务器的机器上，该文件服务器通过在网络上发送消息来与用户机器通信。

作为MINIX模块性的另一个例子，编译系统时是否带网络支持不会对内存管理器有任何影响，而只会影响核心。因为以太网系统任务及对其他I/O设备的支持，都在核心中编译。当被激活时，网络服务器被集成到MINIX系统成为一个服务器进程。它与内存管理器和文件服务器具有相同的优先级。网络服务器操作牵涉到大量数据的高速传送，这需要比用户进程更高的优先级。不过除了以太网任务外，网络功能也可以被用户级进程执行。网络功能不是传统的操作系统功能，对它的详细讨论超出了本书的范围。后续章节的讨论将基于一个不带网络支持的MINIX系统。

2.6.2 公共头文件

include/ 及其子目录包含了许多文件，其中定义了常量、宏、以及类型。POSIX标准需要其中的许多定义，并且指定了在 include/ 及其子目录 include/sys 下其所需的每一个定义应放在哪一个文件中。这些目录下的文件是头文件（header），或称作包含文件，用后缀 .h 标识。这些文件通过C源程序中的 #include 语句引用，该语句是C语言的特性之一。包含文件使得更容易维护一个大系统。

编译用户程序时可能用到的头文件放在 include/ 目录下；而传统上 include/sys/ 目录放那些编译系统程序和公用程序所用的头文件。这种规定并不是绝对的，而且一个典型的编译过程 — 无论是编译用户程序还是编译操作系统的一部分，将同时包含这两者下边的文件。这里我们要讨论编译标准的MINIX系统所用到的文件，先看 include/ 目录，再看 include/sys/ 目录。下一节将讨论 include/minix/ 和 include/ibm/ 目录下的所有文件。顾名思义，include/ibm/用于IBM类型的机器上的MINIX实现。

首先考虑的是真正通用的头文件，它们不直接被任何C源文件引用，而是被别的头文件包含。主控头文件 src/kernel/kernel.h，src/mm/mm.h 及 src/fs/fs.h 分别对应MINIX的三个主要部分，而这三部分必定包含在MINIX的每一个编译版本中。每个头文件都根据MINIX系统相应部分的需要进行剪裁，但每个文件都含有一个如图2－28所示的起始部分。在本书的其他部分还将讨论这些主控头文件。这里只是强调各不同目录下的头文件是一起使用的。在本节和下一节中我们将提到图2－28中引用的每一个头文件。

```
# include <minix/config.h>                    /* 必须是第一个头文件 */
# include <ansi.h>                            /* 必须是第二个头文件 */
# include <sys/types.h>
# include <minix/const.h>
# include <minix/type.h>
# include <limits.h>
# include <errno.h>
# include <minix/syslib.h>
```

图 2－28 一个主控头文件的一部分，它保证了将那些所有C源文件都需要
的头文件包括在内。

讨论从 include/ 目录下的第一个头文件ansi.h 开始（0000行）。这是编译MINIX系统任何一部分都要处理的第二个文件，第一个文件是 include/minix/config.h。ansi.h 的作用是测试编译器是否符合ISO规定的标准C的要求。标准C也称作ANSI C，因为在获得国际承认之前它最早是由美国国家标准局开发的。一个标准C编译器定义了若干宏，它们可在被编译的程序中测试。__STDC__就是一个这样的宏，而且它被一个标准编译器定义为1，就象是C语言预处理器读入了如下的一行：

```
#define __STDC__   1
```

随当前的MINIX版本发行的编译器遵从标准C，但老版本的MINIX是在该标准被采纳之前开发的。不过仍然可以用一个传统的C编译器（Kernighan & Ritchie）来编译MINIX。MINIX希望能够很容易地被移植到新的平台上，允许使用老的编译器正是该目标的一个部分。如果使用标准C编译器，则将处理在0023到0025行之间的语句

```
# define _ANSI
```

。Ansi.h 使用不同的方式定义了若干宏，这具体依赖于宏 _ANSI 是否定义。

Ansi.h 中最重要的宏是 _PROTOTYPE。它允许用如下形式写一个函数原型：

```
_PROTOTYPE (return_type function_name, ( argument_type argument, . . .) )
```

如果使用一个ANSI标准C编译器，则将由C预处理器将其转换成：

```
return_type function_name( argument_type, argument, . . . )
```

如果使用一个老式（即 Kernighan & Ritchie）编译器，则将其转换成：

```
return_type function_name( )
```

在结束 ansi.h 的讨论前还要提及另一个特性。整个 ansi.h 文件被包括在两行语句

```
#ifndef __ANSI_H
```

和

```
# endif
```

之间。在 #ifndef __ANSI_H 的下面一行，__ANSI_H 紧接着被定义。在一次编译中一个头文件应该仅被包含一次。这种写法保证了一个文件被包含多次时内容的正确性。我们将看到在 include/ 目录下的所用头文件都使用这种技术。

include/ 目录下第二个间接包含在每一个MINIX源文件中的头文件是 limits.h（0100行）。其中定义了许多基本的大小值，既有语言中的数据类型，如整数所占的位数，也有操作系统的限制，如文件名长度。errno.h（0200行）也被所有的主控头文件包含，它包含了系统调用失败时从全局变量errno返回给用户程序的错误码。errno也用来标识一些内部错误，如试图向一个不存在的任务发送消息。在MINIX中，若返回值为负表示它是一个错误码，但在返回用户程序之前必须使它们为正值，这里的技巧在于每个错误码都在类似下面一行语句中定义：

```
# define  EPERM  (-SIGN 1)
```

（0236行）。操作系统各部分的主控头文件定义宏 _SYSTEM，但在一个用户程序被编译后 _SYSTEM 并未定义。如果 _SYSTEM 被定义，则_SIGN 定义为“-”，否则 _SIGN无定义。

下面一组文件没有被包含在所有的主控头文件中，但被MINIX所有三个部分的许多源文件使用。最重要的是 unistd.h （0400行）。该头文件定义了许多常量，其中多数是POSIX需要的。它也包含了许多C函数的原型，其中包括所有用于进行系统调用的C函数原型。 signal.h （0700行）定义了标准信号名，同时包含一些与信号相关的函数原型。随后我们将看到，信号处理牵涉到MINIX的所有部分。

fcntl.h（0900行）采用符号方式定义了文件控制操作使用的许多参数。例如，它允许在 open 调用中使用宏 O_RDONLY 来代替数值0作为参数。尽管该文件主要由文件系统引用，但它的定义在核心和内存管理器中也多次用到。

include/ 中的其他文件不如上述文件用得多。stdlib.h （1000行）定义了大多数C程序，除最简单的以外，编译时要用到的类型、宏和函数原型。尽管在MINIX源码中 stdlib.h 只被核心中少数几个文件引用，它却是编译用户程序时使用最频繁的头文件之一。

正如在第3章的系统任务层将看到的那样，操作系统的控制台和终端接口是很复杂的，因为各种不同类型的硬件必须通过一种标准的方式与操作系统和用户程序交互。头文件 termios.h （1100行）定义了控制终端类型的I/O所用到的常量、宏和函数原型。最重要的数据结构是 termios 结构，它包含的内容有：标识操作模式的标志位、设置输入输出速率的变量、以及放置特殊字符的数组（比如 INTR 和KILL）。termios是POSIX所需要的，同样地，该文件中定义的许多宏和函数原型也是POSIX需要的。

然而，正如POSIX标准始终遵循的那样，它并未提供可能用到的全部内容。在文件的后半部分，第1241行以后，提供了POSIX并不包含的扩展部分。其中有些扩展的意义是很明显的。比如定义57,600或以上的波特率，以及对终端显示窗口的支持。正如任何合理的标准都是开放的那样，POSIX标准并不排斥扩展。但在MINIX环境下想写一个可移植到其他环境的程序时，就必须注意避免使用那些局限于MINIX的特征。这一点很容易作到。在

该文件和其他定义面向MINIX的扩展的文件中，这些扩展是通过以下语句控制的。

```
# ifdef _MINIX
```

如果 _MINIX未被定义，编译器将根本不会感知到MINIX扩展。

include/ 下我们要讨论的最后一个文件是 a.out.h （1400行）。它定义了可执行文件在磁盘上的存储格式，包括启动文件执行的文件头结构和编译器产生的符号表。它只由文件系统引用。

现在来看 include/sys/目录。如图2－28所示，MINIX系统各主要部分的主控头文件在包含 ansi.h 后随即都包含 sys/types.h 文件（1600行）。types.h 中定义了许多MINIX使用的数据类型。使用这里提供的定义可以避免对特定情况下所使用的基本数据结构的理解错误而导致的故障。图2－29示出了其中定义的几种数据类型，以及它们分别在16位和32位处理器上编译时所占位数的差异。注意所有的类型名都以“_t”结尾，这不仅仅是一种习惯，而是POSIX标准的规定。这是“保留后缀”的一个例子，而且“_t”不用于非数据类型的其他任何符号名。

图 2－29　16位和32位系统上一些类型的大小，单位为比特。

尽管并不常用sys/ioctl.h （1800行），它也没有被主控头文件包含，但其中还是定义了许多用于设备控制的宏。它也包含了 IOCTL 系统调用的原型。在 include/termios.h 中提供了函数原型的许多POSIX函数已经取代了老的 ioctl 库函数的许多功能，包括对终端、控制台及类似设备的处理，所以IOCTL系统调用在许多情况下并不直接被程序员调用，但该系统调用仍然是需要的。实际上，控制终端的POSIX函数被库转换成了IOCTL系统调用。而且，当前计算机设备的类型是不断增多的，它们各自需要某种方式的控制。例如在该文件的结尾处有若干以DSPIO开头的操作码，它们就用来控制一个数字信号处理器。实际上，本书中讲述的MINIX和其他版本的MINIX之间的主要差别在于：为了突出本书的意图，我们描述了一个只带有少量I/O设备的MINIX。其余许多设备可被加进来，如网络接口、CD－ROM驱动器，声卡等等。在本文件中它们的控制码被定义为宏。

include/sys 下的若干其他文件在MINIX中得到广泛使用。 sys/sigcontext.h （2000行）定义了一些结构，这些结构用来保存和恢复执行信号处理例程前后正常的系统操作。该文件用于核心和内存管理器。MINIX中提供了对跟踪可执行文件和用调试器分析核心转储（core dump）的支持。 sys/ptrace.h （2200行）定义了使用 PTRACE 系统调用的各种可能操作。 sys/stat.h 文件（2300行）定义了我们在图1－12中所看到的结构，它由 STAT 和FSTAT 系统调用返回。同时其中定义了 stat、fstat 以及其他一些用来对文件进行操作的函数的原型。在文件系统和内存管理器中有几处将引用该文件。

最后两个文件没有上述文件用得多。 sys/dir.h （2400行）定义了MINIX的目录项结构。 dir.h 只被直接引用了一次，但包含它的文件在文件系统中被广为使用。它之所以重要是因为它定义了一个文件名最长有多少个字符。最后 sys/wait.h （2500行）定义了WAIT和WAITPID系统调用所使用的宏，它们本身在内存管理器中实现。

2.6.3 MINIX头文件

include/minix/ 和 include/ibm/ 目录包含了MINIX特有的头文件。 include/minix/ 中的文件对任何平台上的MINIX实现都是需要的（尽管其中有少数定义存在与平台相关的替代物）。include/ibm/ 中定义了IBM 类型机器上的MINIX实现所特有的数据结构和宏。

先从 minix/ 目录开始。在前一节中我们注意到 config.h （2600行）被MINIX所有部分的主控头文件所包含，并且它是编译器实际上处理的第一个文件。在许多情况下，当因硬件差异或对操作系统的使用方式不同而需要对MINIX的配置作修改时，只需要编辑该文件并对系统重新编译即可。用户可设置参数均放在该文件的第一部分。其中第一个是参数MACHINE，根据要在哪个平台上编译MINIX，它的值可以是IBM－PC、SUN－4、MACINTOSH或其他值。MINIX的大部分代码是独立于机器类型的，但一个操作系统总有一部分代码与硬件相关。在书中有少数几处所讨论的代码是对硬件而变的，我们使用的例子是IBM PC 兼容机，它使用32位的高档处理器芯片（80386，80486，奔腾或高能奔腾）。我们将其都看作32位Intel芯片。MINIX也可以在老式的16为字长的IBM PC 上编译，当然相应的机器相关部分要重写。在一台PC上，编译器自行决定MINIX应针对哪种机器进行编译。PC上标准的MINIX编译器是Amsterdam Comiler Kit （ACK）编译器，它通过在 __STDC__之外再定义一个宏 __ACK__来标识自己。同时它还定义一个宏 _EM_WSIZE，以指定目标机的字长（以字节为单位）。在2626到2628行之间，宏 _WORD_SIZE 被赋值为 _EM_WSIZE。在该文件的后边以及其他MINIX源文件中有几处都用到这几个定义，例如第2647到2650行以如下的测试语句开始：

```
# if (MACHINE == IBM_PC && _WORD_SIZE == 4 )
```

然后定义了一个32位系统上文件系统高速缓冲区的大小。

Config.h 文件中的其他定义允许在特殊安装中进行其他需求的定制，例如，其中有一段允许编译MINIX核心时装入各种不同的设备驱动程序。这可能是MINIX源代码中被改动得最多的部分。这一段的开头是：

```
# define  ENABLE_NETWORKING  0
# define  ENABLE_AT_WINI  1
# define  ENABLE_BIOS_WINI  0
```

通过将第1行中的0改成1，可以将MINIX编译成带有网络支持。将ENABLE_AT_WINI定义成0和将ENABLE_BIOS_WINI定义成1，可以删去AT类型（即IDE）的硬盘驱动程序代码而使用PC BIOS来作为硬盘支持。

下一个文件是 const.h （2900行）。通过它可以理解头文件的另一种普遍用法。在这里我们发现一些常量定义，在编译一个新核心时它们不大会改变，但却在许多地方用到。通过这些定义我们可以防止发生变量在多处定义不一致的错误，而且这种错误很难发现。在MINIX源代码目录树中还有另外几个文件也命名为 const.h，但其使用很有限。只在核心中使用的定义都包含在 src/kernel/const.h中。内存管理器将 src/mm/const.h 用于其局部定义。只有那些在MINIX系统一个以上的部分中用到的定义放在 include/minix/const.h 中。

const.h 中有几处定义值得注意。EXTERN被定义为 extern （2906行）。在头文件中定义并被两个以上的文件包含的全局变量被声明为EXTERN类型，形式如下：

```
EXTERN  int  who;
```

如果变量采用以下方式声明，

```
int  who;
```

并被包含于两个以上的文件中，则有些链接程序将会把它认作变量多重定义错误。而且，C语言参考手册（Kernighan & Ritchie, 1980）显式地禁止这种语法形式。

为了避免这类问题，必须在除一处之外的所有其他地方都写成

```
extern  int  who;
```

在包含了 const.h之处EXTERN将被代换成 extern，而在定义该变量之前显式地将EXTERN重新定义为空串，那么这一问题便解决了。实现方法是：在MINIX的每一部分中，将全局定义放在一个指定文件－glo.h 中。例如 src/kernel/glo.h，该文件被间接地包含在每一个编译版本中。在每个 glo.h 中有如下语句：

```
# define  _TABLE
# undef  EXTERN
# define  EXTERN
# endif
```

并且在MINIX每一部分的 table.c 文件中，在＃include 之前都有一行

```
# define  _TABLE
```

这样，当头文件被包含进table.c 并展开成为其中的一部分时， extern 绝对不会出现在该文件中的任何地方（因为在 table.c 中EXTERN被定义为空串），并且全局变量的空间只被预留在一个地方，即 table.o 中。

如果你是C程序设计的新手，对上述内容不太理解，那么其中的细节也并不特别重要。对有些链接程序而言，头文件的多次包含可能会导致一些问题，因为它可能引起被包含变量的多重声明。上述EXTERN用法只是增强MINIX移植性的一种途径，因为这样一来，即使在链接程序不支持变量多重定义的机器上，MINIX也能够被链接成功。

PRIVATE被定义为 static 的同义词。对于那些不会在定义文件以外被引用的过程和数据，它们往往被声明为PRIVATE以使其名字在文件之外不可见。作为一条通用的规则，所有的变量和过程都应被声明成在尽可能局部的范围有效。PUBLIC 被定义为空串，于是如下的声明

```
PUBLIC  void  free_zone ( Dev_t  dev, zone_t numb )
```

经C预处理器处理后成为

```
void  free_zone ( Dev_t  dev, zone_t numb )
```

按照C语言的作用域规则，该语句意味着符号 free_zone 从该文件输出，并能被其他文件使用。PRIVATE和PUBLIC并不是必须的，但使用它们是为了尽量消除C语言作用域规则的副作用（C语言中一个符号名缺省是输出到文件之外，但实际上应该正好相反）。

const.h 的其余部分定义了在整个系统中使用的数值常量。其中有一段专门定义与机器或配置相关的内容。例如在全部的源码中，内存大小的基本单位是click（块）。块的具体大小取决于处理器结构。对于Intel兼容芯片、Motorola 68000、Sun SPARC等体系结构，click定义在2957到2965行。该文件中还包括了宏MAX和 MIN，因此 z=MAX（x, y）的功能是将 x和y中的较大值赋给z。

type.h （3100行）是另一个通过主控头文件被包含在每一个编译版本中的文件。它包括了许多重要的类型定义，以及相关的数量值。该文件中最重要的定义是3135到3146行的message。我们当然可以将message定义为一个包含若干字节的数组，但在编程实践中最好将其处理为一个结构，其中包含有若干不同的可能的消息类型的联合。消息定义了6种格式：mess_1到mess_6。一条消息包括如下的几个部分：m_source域，标识谁发送了该消息；m_type域，标识消息类型（例如，发给时钟任务的GET_TIME）;以及数据域。六种消息类型示于图2－30。该图中第1、第2种看起来很相似，第5、第6种也很相似。这种情况对于在32位的Intel芯片上实现的MINIX成立，但对于在一台int、long和指针类型占有不同空间的机器上则未必。定义6种消息类型将使得更易于在不同的体系结构上重新编译。

图 2-30 MINIX中使用的六种消息类型。消息元素的长度将随机器结构而变化；图中所示的是在32位指针机器，如Pentium (Pro)上的长度。

当需要发送一条消息，而其中包含3个整数和3个指针（或3个整数和2个指针）时，应使用图2－30中的第1种格式。对其他格式依此类推。如何给第1种格式中的第一个整数赋值呢？假设这里的消息为x，则x.m_u指向该

消息结构的联合部分。我们使用x.m_u.m_m1来指向6种格式中的第1种。最后，我们用x.m_u.m_m1.m1i1来指向该结构中的第1个整数，这种使用方法相当麻烦，所以在定义了消息之后，一般都定义一个短一点的名字作为其相应的宏。于是用x.m1_i1来代替x.m_u.m_m1.m1i1。这种短名字的格式都由以下几部分组成：字母m，格式编号，一个下划线，一个或两个类型字母，它用来指明该域是一个整数、指针、长整数、字符、字符串或是函数，以及一个序号，在一条消息中包含多个相同类型的域时用该序号对它们加以区分。

在讨论消息格式时，顺便指出操作系统及其编译器通常对某些事物有共同的“理解”，比如一个结构的布局等。而且这种共识简化了系统的构造。在MINIX的消息中，int域有时用来存放 unsigned数据类型。在某些情况下这会导致溢出，但编码时我们知道，MINIX编译器只机械地将 unsigned 类型和 int 类型之间来回拷贝，而不进行溢出检查。一种更规范的写法是将每一个 int 域写成 int 和 unsigned 的联合。在消息中对long 也同样处理，有些 long 类型的域被用来传送 unsigned long 型数据。这样做是不是有点骗人的感觉？有人也许这么认为，但如果你要将MINIX移植到一个新平台上，很显然必须对消息的确切格式给予足够的注意，而且在这里你已经收到警告，编译器的特性是另一个必须注意的因素。

include/minix 下还有另一个通过主控头文件被广泛使用的文件，即 syslib.h （3000行）。它包含了在操作系统内部调用以访问操作系统其他服务的C库函数原型。本书不详细讨论C库，但其中有许多是标准的，且对任何C编译器都可用。然而，syslib.h 引用的C函数显然是特定于MINIX的；并且，将MINIX移植到一个带有不同编译器的新系统将需要移植这些库函数。幸运的是这并不难，因为这些函数只是抽取出函数调用的参数，将其插入一条消息结构中，然后发送该消息，接着从应答消息中将结果抽取出来。这中间许多库函数的定义仅有十几行甚至更少的C代码。

当一个进程需要执行一条MINIX系统调用时，它向内存管理器（MM）或文件系统（FS）发送一条消息。每条消息中含有所要求的系统调用序号。这些序号在下一个文件 callnr.h （3400行）中定义。

文件 com.h （3500行）主要包含从MM和FS发送到I/O任务的消息中所使用的公共定义。其中也定义了任务序号。为了将其与进程号分开，任务号是负数。该文件还定义了可被发送到每个系统任务的消息类型（函数代码），例如，时钟任务接收代码SET_ALARM（用于设置定时器）、CLOCK_TICK（当时钟中断发生时）、 GET_TIME（取真实时间）、 SET_TIME（设置一天中的当前时间）、 REAL_TIME 的值就是对GET_TIME请求所返回的消息类型。

最后，include/minix/还包含了几个更特殊的头文件。其中有boot.h （3700行），它被核心和文件系统用来定义设备、以及访问从 boot 程序传入系统的参数。另一个是 keymap.h （3800行），它定义了若干结构，这些结构用来实现不同语言所需的字符集对应的特殊键盘布局。它也被那些生成和加载这些表格的程序使用。这里的某些文件，如 partition.h （4000行）只被核心使用，而不被文件系统或内存管理器使用。对于一个支持额外的I/O设备的实现，这里有另外一些类似的文件，它们分别支持其他的设备。这些设备在该目录下的位置需要解释一下。理想情况下，所有的用户程序应该仅仅通过操作系统来访问设备，这些支持外部设备的文件应当放在 src/kernel/ 下。但是系统管理的实际情况要求，某些用户命令能访问系统级的结构，例如对硬盘进行分区的命令。正是为了支持此类公用程序，我们将这些特殊头文件放在 include/ 目录树下。

include/ibm 是这里讨论的最后一个特殊头文件目录，它下面有两个文件包含与 IBM PC系列机相关的信息。一个是 diskparam.h ，它被软盘系统任务使用。尽管该任务属于标准的MINIX版本，但本书不详细讨论其源代码，因为它和硬盘任务很类似。另一个文件是 partition.h （4100行），它定义了IBM 兼容机上使用的硬盘分区表和相关的常量。该文件有助于将MINIX移植到其他硬件平台上。对于不同硬件，include/ibm/partition.h 必须可被替换为通常位于另一个相应目录下的 partition.h 文件。但在include/minix/partition.h 中定义的结构是MINIX内部使用的，它应对各种硬件都保持不变。

2.6.4 进程数据结构和头文件

现在来看 src/kernel/ 下的代码。前两节中的讨论是围绕一个典型的主控头文件进行的，这里我们先来观察核心的主控头文件 kernel.h （4200行）。它一开始先定义了3个宏。第1个 _POSIX_SOURCE是POSIX标准自行定义的一个特征检测宏。所有这类宏都必须以一个下划线“_”开头。定义 _POSIX_SOURCE 的作用是保证所有POSIX要求的符号和那些显式地允许但并不要求的符号将是可见的，同时又隐藏掉任何属于POSIX非官方扩展的附加符号。我们已经提到过接下来的两个定义： 宏_MINIX将为MINIX所定义的扩展而重载_POSIX_SOURCE的作用；而在编译系统代码时，如果要作与用户代码不同的事情，比如改变错误码的符号，则可以对 _SYSTEM宏进行测试。kernel.h随后包含一些 include/ 及其子目录 include/sys/ 和 include/minix/下的其他头文件，其中包括图2－28中的所有文件，这些文件在前两节中已经作过讨论。最后将包含本地目录src/kernel/ 下的另外四个头文件。

这里可以向初学C语言的读者说明 # include 语句如何引用文件名。每个C编译器都有一个查找包含文件的缺省目录。在标准的MINIX中这通常是 /usr/include。当文件名置于一对小于号和大于号（“<...>”）之间时，编译器将到缺省包含目录或其下的一个指定的子目录去找包含文件。当文件名置于双引号（“...”）之间时，首先在当前目录（或一个指定目录）下查找，如果未找到，则再到缺省目录下找。

通过在所有其他核心源文件中写这样一行简单的语句

```
# include  "kernel.h"
```

便可以使所有源文件共享大量重要的定义。由于头文件的包含次序有时非常重要，kernel.h 同时还一次性地保证了这种次序的正确性。这等于将头文件“作对一次，然后便可忽略具体细节”的思想带到了一个新高度。对文件系统和内存管理器也存在类似的主控头文件。

下面继续看 kernel.h 中的四个本地头文件。正如在公共头文件目录 include/minix/ 下有 const.h 和 type.h 一样，在 src/kernel/ 目录下也有 const.h 文件和 type.h 文件。 include/minix/ 下的文件之所以被放在那里是因为系统的许多部分，包括在系统控制下运行的程序都需要它们。 src/kernel/ 下的文件提供了仅为编译核心所需的定义。FS和MM源文件目录也包含有 const.h 和 type.h 文件，同样这些文件定义了仅为系统相应部分使用的常量和类型。在主控头文件中包含的另外两个文件 proto.h 和 glo.h 在主 include/ 目录下没有对应的文件。但我们将发现它有编译FS 和 MM 时使用的对应文件。

const.h （4300行）包含有许多机器相关的值，这些值用于Intel的CPU芯片，但在别的硬件上编译时则可能不同。这些值定义在第4302到4396行之间的语句中，这些语句被包括在以下两条语句之中：

```
# if ( CHIP == INTEL )
```

和

```
# endif
```

当在一种 Intel 芯片上编译MINIX时，宏CHIP 和INTEL 将被定义，并在include/minix/config.h （2763行）中设置成相应的值，于是与机器相关的代码将被编译。当MINIX被移植到基于Motorola 68000的系统时，可以通过加入以下代码

```
# if ( CHIP == M68000 )
```

和

```
# endif
```

同时在 include/minix/config.h 中将定义芯片类型的语句改为

```
# define  CHIP  M68000
```

而实现系统的移植。通过这种方式，MINIX可以处理那些与平台相关的常量和代码。这种语法结构对代码的可读性没有什么好处，因此应当尽量少用。实际上，为了保证可读性，本书中印出的代码已经删节了许多针对68000和其他处理器的代码。但CD－ROM和Internet 上的代码则予以保留。

const.h中有一些定义值得特别的注意。其中有一些是机器相关的，例如重要的中断向量和用在每次中断后复位中断控制器芯片的一些域的值。核心中的每个系统任务都有自己的堆栈，但在处理中断时，将使用一个大小为 K_STACK_BYTES （定义于第4304行）的特殊的堆栈。这也定义在机器相关部分，因为不同的体系结构可能需要不同的堆栈空间。

其他定义是机器无关的，但它们被核心代码的许多部分用到。例如，MINIX进程调度程序有 NQ（3）个优先级队列，分别命名为 TASK_Q（最高优先级）、SERVER_Q（中等优先级）和USER_Q（最低优先级）。使用这三个符号名是为了使源代码更容易理解，但这三个宏所定义的数值则被编译到执行程序中。最后，const.h 的最后一行将 printf 定义为一个宏，其真实值为 printk。这样核心就可以使用核心内部定义的过程将出错信息之类的消息打印在控制台上。这个过程旁路了通常的机制。通常的机制需要从核心传送消息到文件系统，随后又从文件系统到打印机任务。当系统发生故障时，这套机制可能会失效。在这种情况下，我们将看到在一个核心过程panic中调用printf，也即 printk。正如你猜测的那样，panic 检测到严重错误时被调用。

文件 type.h （4500行）定义了几个所有MINIX实现都要用到的原型和结构。tasktab结构定义了tasktab数组中每一项的结构，memory结构（第4513到4516行）定义了唯一确定一片内存区域的两个数值。这里正好可以提到内存管理的几个概念。块是内存管理的基本单位，针对Intel芯片的MINIX中，一个块是256个字节。内存按phys_clicks或vir_clicks进行计量，phys_clicks被核心用来存取系统中任何地方的存储单元；vir_clicks则被核心之外的进程使用。一个vir_clicks存储引用总是与分配给一个特定进程的内存段的基地址相关，而且核心经常需要在这两者之间来回转换。由于一个进程可以使用vir_clicks来引用其全部内存，这使得这种不便被部分地抵消。有人可能会设想使用同一种单位来指定这两种类型内存的大小，但使用vir_clicks更好一点，其原因是使用vir_clicks时将进行检查，以保证不会访问超出当前进程之外的内存地址。这是现代Intel处理器（如奔腾和高能奔腾）保护模式(protected mode)的主要特征之一。由于早期的8086和8088处理器不具备这种特征，造成MINIX早期版本的设计者十分头痛。

type.h还包含几个机器相关的类型定义，如用于Intel处理器上的port_t、segm_t和reg_t类型（第4525到4527行），它们分别用来访问I/O端口、存储器段和CPU寄存器。

数据结构也可能与机器相关。第4537到4558行为Intel处理器定义了stackframe_s结构，它定义了如何将寄存器值保存到堆栈上。这个结构非常重要－在进程被投入运行状态或被脱开运行状态（如图2－2所示）时，它被用来保存和恢复CPU的内部状态。将其定义成可以用汇编语言高速读写的格式，这将减少进程上下文切换的时间。segdesc_s是与Intel处理器结构相关的另一个结构，它是保证进程不会发生内存访问越限机制的一部分。

为了解释不同平台之间的差异，在该文件中保留了几个针对Motorola 68000系列处理器的定义。Intel处理器族的芯片中有些使用16位寄存器，有些使用32位寄存器，所以对于Intel体系结构，基本的reg_t类型是无符号型。对于Motorola处理器reg_t定义为u32_t类型。这些处理器也需要一个stackframe_s结构（第4583到4603

行），但为了使相应的汇编代码操作速度尽量地快，其布局是不一样的。Motorola体系结构根本不需要port_t、segm_t类型，也不需要segdesc_s结构。同时也存在几个专为Motorola体系结构定义的结构，它们没有Intel芯片的对应物。

下一个文件proto.h（第4700行）是我们看到的最长的头文件。所有那些必须在其定义所在文件外被感知的函数的原型都放在proto.h中。它们都使用了前一节中提到的 _PROTOTYPE技术，这样，MINIX核心便既可以使用传统的C编译器（由Kernighan和Richie定义），例如MINIX初始提供的编译器；又可以使用一个现代的ANSI标准C编译器，例如MINIX版本2中所带的编译器。这其中的许多函数原型是与系统相关的，包括中断和异常处理例程以及用汇编语言写的一些函数。本书中未讨论的那些设备驱动程序所使用的函数的原型未被列出。针对Motorola处理器的条件编译代码也已从本文件及后面讨论的文件中删去。

包含在主控头文件中的最后一个核心头文件是glo.h（第5000行）。其中包含了核心的全局变量。宏EXTERN的作用已在文件include/minix/const.h的讨论中说明。它通常被展开成extern。注意glo.h中的许多定义都以该宏开头。当该文件被包含在定义了宏_TABLE的 table.c中时，宏EXTERN的定义被强行取消。将glo.h包含在其他C源文件中将使table.c中的变量定义为核心中的其他模块所感知。held_head 和held_tail（第5013和5014行）是指向一个挂起的中断队列的指针。proc_ptr（第5018行）指向当前进程的进程表项。当发生一条系统调用或中断时，它指明应将寄存器值和处理器状态保存在何处。sig_procs（第5021行）记录那些有信号等待发送给内存管理器进行处理的进程的数目。glo.h中有几项用extern定义，而不是用EXTERN。其中包括sizes — 由引导监控程序填入的一个数组、任务表tasktab、以及任务栈t_stack。最后两个是经初始化过的变量，这是C语言的特征之一。宏EXTERN的使用与C风格的初始化并不兼容，因为一个变量只可被初始化一次。

每个任务在t_stack中有其自己的堆栈。在中断处理期间，核心使用另外一个独立的堆栈，因为只有执行中断处理的汇编语言级的例程访问这一堆栈，因而不需要被全局范围所感知，所以不在这里对它作声明，

还有另外两个核心头文件，虽然不如包含在kernel.h中的那些用得多，但还是用得很广泛的。第一个是proc.h（第5100行）。它将一个进程表项定义为一个结构proc（第5110到5148行）。在proc.h的后边，它将进程表本身定义为proc结构的数组 proc[NR_TASKS＋NR_PROCS]（第5186行）。在C语言中，标识符的这种重用是允许的。宏NR_TASKS定义在include/minix/const.h中（第2953行），而NR_PROCS则定义在include/minix/config.h中。（第2639行）。它们共同决定进程表的大小。NR_PROCS可以被改变以创建一个可以处理大量用户的系统。因为进程表的访问非常频繁，并且计算数组中的一个地址需要用到很慢的乘法操作，所以使用一个指向进程表项的指针数组pproc_addr（第5187行）来加快操作速度。

每个表项都包含有足够的存储空间来保存进程的寄存器值、堆栈指针、状态、存储器映像、堆栈长度、进程标识数、计费信息、时间闹钟以及消息等信息。每个进程表项的第一部分是一个stackframe_s结构。进程被投入运行是通过将其进程表项的地址装入其堆栈指针，然后从该结构中弹出全部的CPU寄存器值而实现的。当发送消息进程由于目标进程未处在等待状态而无法完成一个SEND操作时，它被送到一个队列中，该队列由目标进程的一个p_callerq域（第5137行）指向。于是，当目标进程最终执行RECEIVE操作时，它很容易找到等待向其发送消息的所有进程。p_sendlink域（第5138行）用于将队列中的成员链接起来。

当一个进程执行RECEIVE操作，但是没有任何消息在等待它接收时，该进程将阻塞，同时将它期望接收消息的源进程序号保存在p_getfrom中。消息缓冲区的地址保存在p_messbuf中。每个进程表项的最后三个域是p_nextready、p_pending和p_pendcount（第5143到5145行）。p_nextready用于将进程链接在调度程序队列中，p_pending是一个位图，用于记录那些尚未被传送到内存管理器的信号（因为内存管理器没有在等待一条消息）。p_pendcount域是这些信号的计数值。

p_flags域中的标志位定义了每个表项的状态。如果其中任一位被置位，则进程将无法运行。各种标志被定义和描述在第5154到5160行。如果该表项未被使用，则P_SLOT_FREE被置位。在执行一个FORK操作后，如果子进程的内存映像尚未建立起来，那么NO_MAP将被置位以阻止子进程运行。SENDING和RECEIVING表示该进程被阻塞，其原因是它正在试图发送或接收一条消息。PENDING和SIG_PENDING表示已接收到信号。P_STOP则在调试期间为跟踪提供支持。

提供宏proc_addr（第5179行）是因为在C语言中下标不能为负数。在逻辑上，数组proc应从 －NR_TASKS到＋NR_PROCS。但在C语言中下标必须从0开始，所以proc[0]指向进程表项下标最小的任务，其他也依次类推。为了更便于记录进程表项与进程之间的对应关系，我们可以使用

```
rp = proc_addr(n);
```

将进程n的进程表项地址赋给rp，无论它是正还是负。

bill_ptr（第5191行）指向正在对其CPU使用计费的进程。当一个用户进程调用文件系统，而文件系统正在运行时，proc_ptr（在glo.h中）指向文件系统进程，但是bill_ptr将指向发出该调用的用户进程。因为文件系统使用的CPU时间被作为调用者的系统时间来计费。

两个数组rdy_head和rdy_tail用来维护调度队列。例如，rdy_head[TASK_Q]指向任务队列中的第一个进程。

另一个被许多源文件包含的头文件是protect.h（第5200行）。该文件中几乎所有的内容都与支持保护模式的Intel处理器（80286，80386、80486、奔腾、高能奔腾）体系结构的细节有关。对这些芯片的详细描述不属于本书的范围。简单地说，这些芯片都包含一些指向内存中描述符表(descriptor tables)的内部寄存器。描

述符表定义了系统资源是如何使用的，并防止进程访问属于其他进程的内存区域。此外处理器提供了四种特权级(privilege levels)，MINIX使用其中的三种，它们的符号定义位于第5243到5245行。核心的最中心部分，即运行于中断处理期间的部分和切换进程的部分运行在INTR_PRIVILEGE特权级。在该特权级上，进程可以访问全部的内存空间和CPU的全部寄存器。系统任务运行在TASK_PRIVILEGE特权级。该特权级允许它们访问I/O，但不能使用那些修改特殊寄存器（如指向描述符表的寄存器）值的指令。服务器进程和用户进程运行在USER_PRIVILEGE特权级。运行在该特权级的进程不能执行某些指令，如访问I/O端口、改变内存分配状况，或改变处理器运行级别等等。特权级的概念对于熟悉现代CPU体系结构的读者很容易理解，但对于那些通过汇编语言来学习体系结构，或是只学习过低档微处理器的读者，他们可能从未遇到过此类运行级别的限制。

在核心目录下还有几个其他的头文件，但这里只提及其中两个。第一个是sconst.h（第5400行）。它包含汇编代码所使用的常量。这些常量都是相对进程表项中stackfram_s部分的偏移，其表示方式为汇编器可使用的一种格式。因为汇编代码不由C编译器处理，所以将其放在单独的文件中将更简单。同样，由于这些定义均与机器相关，将其隔离出去将简化MINIX向另一处理器的移植。该处理器将需要另一个版本的sconst.h。注意许多偏移被表示为前一值加上字母W，这里W的值在第5401行被设为字长。这样处理允许同一个文件被编译成16位或32位版本的MINIX。

这里有一个潜在的问题。头文件被假设为允许一个人提供一套正确的定义，随后可以在许多地方使用而不必过多地注意细节。显然，类似sconst.h中的重复定义违反了该原则。当然sconst.h只是一个特例，但在这种情况下，若修改sconst.h 或proc.h，必须保证其间的一致性。

最后一个文件是assert.h（第5500行）。POSIX标准要求必须有assert函数，它可以用来进行运行时测试、终止一个程序同时打印出一条消息。事实上，POSIX要求include/ 目录下必须有一个assert.h 文件。那么为什么此处有另一个版本呢？答案是当用户进程出错时，可以依靠操作系统提供某些服务，比如在控制台上打印一条消息。但如果核心本身出错，则正常的系统资源就未必靠得住。于是核心提供其自己的例程来处理 assert 并打印消息，它独立于通常的系统库中的版本。

在kernel/ 目录下还有几个头文件我们没有讨论过。它们支持I/O系统任务，我们将在下一章中合适的地方描述。但在进入可执行代码之前，我们先看看table.c（第5600行）。其编译生成的目标文件将包含所有的核心数据结构，我们已经看到许多这类数据结构定义在glo.h 和proc.h中。在第5625行定义了宏_TABLE，正好位于 #include语句之前。正如前面所解释的，该定义导致EXTERN被定义为空串，于是为EXTERN之后的所有数据声明分配存储空间。除了 glo.h 和 proc.h中的结构以外，tty.h中定义的由终端任务使用的几个全局变量也都在这里分配存储空间。

除了在头文件中声明的变量之外，还有另外两个地方可以为全局数据分配存储空间。某些定义在table.c中直接进行定义。第5639到5674行为每个任务分配堆栈空间。对每个可选的任务，对应的宏ENABLE_XXX（定义在文件 include/minix/config.h中）用来计算堆栈大小。于是若一个任务未被激活则将不为它分配空间。按照这个规则,各种宏ENABLE_XXX用来确定各个可选的任务是否被表示在tasktab数组中,如前边src/kernel/type.h（第5699到5731行）中所声明的，该数组是由tasktab结构组成的。不论它是系统任务、服务器进程还是用户进程，例如init，对于每一个系统初始化期间启动的进程都对应有一项。数组的下标显式地将任务号映射到相应的启动过程。tasktab确定每一个进程所需的堆栈空间并为每个进程提供一个标识串。将tasktab放在这里而不放在一个头文件中，其原因是前述防止多重声明的EXTERN技术对初始化的变量不起作用，也即任何时候都不能采用如下的写法

```
extern  int  x=3;
```

使用前边的堆栈大小定义可为所有系统任务分配堆栈空间（第5734行）。

虽然已经尽量将所有用户可设置的配置消息单独放在include/minix/config.h中，但是在将tasktab的大小与NR_TASKS相匹配时仍可能会导致错误。在table.c的结尾处使用一个小技巧对这个错误进行检测。方法是在这里声明一个dummy_tasktab，声明的方式是假如发生了前述的错误，则dummy_tasktab的大小是非法的，从而导致编译错误。由于哑数组声明为extern，此处并不为它分配空间（其他地方也不为其分配空间）。因为在代码中任何地方都不会引用到它，所以编译器不会受任何影响。

另一个分配全局空间的地方是在汇编代码文件mpx386.s（第6483行）的末尾。在标号_sizes处的空间分配将在核心数据段的最前边放置一个魔数（籍此来标识一个合法的MINIX核心）。此处通过 .space伪指令来分配其他的空间。汇编语言程序通过这种方式来预留空间使得可以强制地将 _sizes数组在物理上位于核心数据段的开始，这样，boot程序就可以很容易地将数据放在正确的地方。引导监控程序读取这个魔数，如果魔数正确，它将覆盖之并用MINIX系统各部分的空间大小来初始化 _sizes数组。在初始化期间核心将使用这些数据。在启动时，从核心看来，这是一个初始化了的数据区域。但是，核心最终在那里找到的数据在编译时还无法获得。这些数据是在核心被启动之前由引导监控程序补上的。这个过程是不大常见的，通常情况下，不会要求一个程序知道另一个程序的内部结构。从上电开始到操作系统正常运行这段时间是相当不平常的，要求使用不常用的技术。

2.6.5 引导MINIX

现在差不多可以来看执行代码了。但在此之前先来搞清楚MINIX是怎么被装入内存的。当然MINIX是从硬盘上装入的，图 2－31示出了软盘和分过区的硬盘的布局。

图 2－31 引导使用的磁盘结构。 （a）未分区的磁盘，第一扇区就是引导块。
（b）分过区的磁盘，第一个扇区是主引导记录。

当系统启动时，硬件（实际是ROM中的一个程序）读取引导磁盘的第一个扇区，并执行从那里得到的代码。在一个未分区的MINIX软盘上，第一个扇区是一个引导块，由它装入引导程序，如图2－31（a）所示。硬盘是分过区的，第一个扇区上的程序取同样位于第一个扇区中的分区表，并装入和执行活跃分区的第一个扇区，如图2－31（b）所示（通常有且只有一个分区被标识为活跃）。一个MINIX分区与一个未分区的MINIX软盘结构相同，其中有一个装入引导程序的引导块。

真实情况可能较图示的略复杂一些，因为一个分区可能包含子分区。这种情况下分区的第一个扇区就是另外一个包含子分区表的主引导记录。但最终控制权将传到一个引导扇区，即一个不再被进一步细分的设备的第一个扇区。对软盘，其第一个扇区总是一个引导扇区。MINIX确实允许将一张软盘分区，但只有第一个分区可以引导。这张软盘上没有单独的主引导记录，而且也不允许有子分区。这样便可以使用完全相同的方法来安装被分区的和未分区的软盘。软盘分区的主要用途在于它可以方便地将一张安装盘分成一个将拷贝到RAM盘的根映像，及一个在不需要时可被卸下来的可安装部分。这样便可以空出软盘驱动器以继续安装过程。

将MINIX引导扇区写入硬盘时，要对它进行修改，修改的内容是将添加一个扇区号以便在其分区或子分区中找到boot程序。这个添加操作是非常必要的，因为在装入操作系统之前无法通过目录和文件名来定位一个文件。该添加操作以及向硬盘写引导扇区的操作由一个特殊的程序 installboot 完成。boot是MINIX的次级装入程序，它不仅可以装入操作系统，而且作为一个监控程序，它允许用户改变、设置和保存不同的参数。boot从它所在分区的第二个扇区中寻找一套可使用的参数。象标准的UNIX一样，MINIX保留每个硬盘设备的前1K字节作为一个引导块。但其中只有一个512字节的扇区被ROM引导程序或主引导扇区装入，这样另外512字节可以用来保存设置信息。这些信息控制引导操作，并且被传到操作系统本身。缺省的设置显示一个只有一个选择项的菜单，即引导MINIX。但该设置信息可以被改变，以显示一个更复杂的菜单。这样便允许引导其他操作系统（通过装入并执行其他分区的引导扇区）或使用不同的选择项来引导MINIX。缺省设置也可以被修改，以旁路掉该菜单而直接引导MINIX。

boot并不是操作系统的一部分，但它很精巧，可以使用文件系统的数据结构来找到操作系统映像。缺省情况下，boot寻找文件 /minix，若存在 /minix/目录，则查找其下的最新文件。但也可以修改引导参数以查找其他任何名字的文件。这种灵活性是不一般的，而且多数操作系统固定系统映像的文件名。但MINIX是一个与众不同的系统，它鼓励用户对其进行改动并生成新的实验版本。这些都要求进行实验的用户应该有某种方法对多个版本进行选择，因为只有这样才能在一次实验失败后退回到上一个正确的版本。

被boot装入的MINIX映像是这样得到的：先对核心、内存管理器、文件系统和init程序分别进行编译，然后将所产生的各个文件链接而成。其中的每一个独立文件都包含一个 include/a.out.h中定义的很短的头。从这个头所包含的信息中，boot可以确定在可执行代码装入后需要为未初始化数据及各部分初始化数据预留多少空间，据此便可以将下一部分装在合适的地址。前节中提到的_sizes数组也接收一份同样的信息，以便核心可以访问由boot装入的各模块的位置和大小信息。可用于装入引导扇区、boot程序、以及MINIX的内存区域取决于硬件。同样，某些机器体系结构可能需要对可执行代码内部的地址作一些调整以将其校准到程序装入的真正地址。Intel处理器的分段体系结构无需进行该操作。由于装入过程的细节各机器都不相同，而且boot本身不属于操作系统，所以不再对其进行讨论。重要的是操作系统以某种方式被装入内存，一旦装入过程结束，控制便被传递给核心的执行代码。

此外，需要指出操作系统并不仅仅可以从本地硬盘装入。无盘工作站可以通过网络从远地硬盘装入操作系统，当然这要求ROM中具有网络软件。尽管具体细节各不相同，但大体上很类似。ROM代码必须能够从网络上获得一个包含完整的操作系统的文件。如果MINIX通过这种方式装入，则操作系统被装入内存过程中的初始化过程几乎无需修改。当然这需要一个网络服务器以及一个经过修改的，能够从网络访问文件的文件系统。

2.6.6 系统初始化

基于IBM PC类型机器的MINIX若需要与旧的处理器兼容则可被编译成16位模式，若在80386以上的芯片上为了获得更高的性能则可以编译成32位模式。这两种模式使用相同的C源代码，根据编译器本身是16位或32位而产生相应的输出。由编译器本身定义的一个宏确定文件 include/minix/config.h 中宏 _WORD_SIZE 的值。MINIX执行代码的第一部分是用汇编语言写的，并且针对16位或32位编译器必须使用不同的源文件。初始化代码的32位版本位于文件 mpx386.s中。对应的16位系统则位于mpx88.s。这两者同时也都包含对其他低层核心操作的汇编语言支持。16位和32位的选择在文件mpx.s中自动完成。mpx.s很短，其内容示于图2－32。

```
# include <minix/config.h>
# if _WORD_SIZE == 2
# include "mpx88.s"
# else
```

```
# include "mpx386.s"
# endif
```

图 2－32 如何选择不同的汇编语言源文件。

Mpx.s示出了C语言预处理器中＃include 语句的一种非常规用法。通常 # include用来包含头文件，但也可以用来选择源代码的适当部分。使用 ＃if 语句来作到这一点将需要把两个很大的文件mpx88.s 和 mpx386.s放在一个单独的文件中。这样作不仅不实用，而且浪费磁盘空间。因为在一个特定的安装中可能这两个文件中的一个根本就不被用到，而且可以被归档或删除。在随后的讨论中我们将以32位的mpx386.s为例。

由于这是首次接触可执行代码，我们先讲一下在本书中我们是如何对可执行代码进行讨论的。同时理解编译一个大的C程序时使用的多个源文件是很困难的。通常我们一次只讲述一个文件，而且按照这些文件的次序进行。我们从MINIX系统各部分的入口点开始并跟随执行主线前进。当遇到调用一个支撑函数时，将首先简略讲一下调用的意图，但并不对其进行详细的内部描述，而是一直到被调用函数的定义处才详细地对其进行讨论。重要的从属函数定义通常与其调用处放在同一个文件中，其前边是发出调用的高层函数。但小的或通用的函数有时被集中放在单独的文件中。同时，出于移植性的考虑，与机器相关和无关的代码尽量分开放在独立的文件中。组织代码费了很多功夫，实际上在本书的写作过程中为了方便读者重写了许多文件。然而大程序有许多分支，有时理解一个主函数需要读懂它所调用的函数，所以备一些书签并脱开书中的次序，以另一种顺序观察问题有时可能会有帮助。

在讲完代码组织方式之后，现在言归正传。MINIX的启动牵涉到几次控制权转移，它们发生在mpx386.s中的汇编语言例程和start.c及main.c中的C语言例程之间。我们将按执行次序进行讨论，尽管这需要从一个文件跳到另一个文件。

一旦引导进程将操作系统装入内存，控制权便转到标号MINIX（在mpx386.s中，第6051行）。第一条指令是一条跃过几个字节数据的跳转指令。这几个字节数据包括引导监控程序标志（第6054行），该标志由引导监控程序用来标识核心的不同特征，其中最重要的是16位/32位系统标志。引导监控程序总是从16位模式开始，但在需要时将CPU切换到32位模式。这个切换发生在控制权传给MINIX之前。监控程序还建立一个堆栈。汇编代码需要作许多工作，包括：建立一个堆栈框架以为C编译器编译的代码提供适当的环境；拷贝处理器所使用的表格来定义存储器段；以及建立各种处理器寄存器等。待这些工作结束后，初始化过程通过调用（第6109行）C函数cstart继续进行。注意在汇编语言代码中用_cstart引用该函数。这是因为C编译器编译的所有函数在符号表中其名字前都有一个下划线，而且当编译好的分立模块被链接时，链接程序查找这样的名字。由于汇编器并不自动地添加这个下划线，所以汇编语言程序员必须显式地加上这个下划线以保证在目标文件中能够找到相应的符号名。cstart调用另一个例程来初始化全局描述符表(Global Descriptor Table) — 这是Intel 32位处理器实现保护模式的核心数据结构，以及中断描述符表(interrupt Descriptor Table) — 它用来为每种可能的中断类型选择执行代码。在从cstart返回之后，lgdt和lidt指令（第6115和6116行）通过向其对应的寻址寄存器装入相应的值来将这些表格激活。如下指令

```
jmpf  CS_SELECTOR: csinit
```

初看起来象是空操作，因为好象在其所处的地方有一个nop 指令序列一样。它把控制转到当时控制所在的位置，但这是初始化过程的一个重要部分。这个跳转指令强制使用刚刚被初始化的结构。在对处理器寄存器作进一步的操作之后，MINIX以第6131行的一个跳转（不是调用）操作结束，这样便跳转到了核心的main入口点（在main.c中）。在该处mpx386.s中的初始化代码结束。该文件剩余部分的代码用来启动或重启动一个系统任务、进程、中断处理程序或其他的支撑例程。出于效率的原因它们均用汇编语言编写，下一节还将对它们进行讨论。

现在来看高层的C初始化函数。这里最基本的策略是用高层的C代码作尽可能多的操作。正如我们所看到的，这里已经有了两种版本的mpx代码，如果将C代码可能执行的操作都由C代码承担，则将省去两大段汇编代码。cstart做的第一件事（在cstart.c中，第6524行）是调用prot_init来建立CPU的保护机制和中断表。然后将引导参数拷贝到内存的核心部分以及将其转换成数值。它还确定显示器的型号、内存大小、机器类型、处理器操作模式（实模式还是保护模式），以及是否可能返回到引导监控程序等。所有的信息都保存在适当的全局变量中，这是为了使核心代码的所有部分在需要时都能够访问到它们。

Main函数（在main.c中，第6721行）完成初始化，然后开始系统的正常运行。它调用intr_init来配置中断控制硬件。该操作之所以放在这里是因为此前必须知道机器类型，因为完全依赖于硬件，所以该过程放在一个独立文件中。该调用中的参数（1）指示intr_init这是在为MINIX执行初始化，若使用参数（0）则再次初始化硬件，使其回到原始状态。intr_init通过两个步骤来保证在初始化完成之前的任何中断都不会生效。第一步向每个中断控制器芯片中写入一个字节以使其无法响应外部中断。随后，用来访问设备相关的中断处理程序的表格中所有表项都填入一个例程的地址，该例程在收到一个伪中断时将打印出一条信息，它绝对没有任何副作用。其后，在每个I/O任务运行其自己的初始化代码时，这些表项逐个地被重填，以指向相应的中断处理例程。最后，每个系统任务复位中断控制器芯片中的一个二进制位以激活其自己的中断信号输入。

接下来调用mem_init。它初始化一个数组，该数组定义系统中每个可用内存块的地址和大小。和中断硬件的初始化一样，内存初始化的细节是与硬件相关的。而且这里将mem_init分离到一个独立的文件中以保持main函数的平台无关性。

Main函数的主要部分用来建立进程表，这样当调度到第一批任务和进程时，它们的内存映像和寄存器将能够被正确地设置。进程表的所有表项都被标志为空闲；用于加快进程表访问的pproc_addr数组被第6745到6749行的循环进行初始化。第6748行的代码

(pproc_addr + NR_TASKS) [t] = rp;

可被定义为：

pproc_addr [t + NR_TASKS] = rp;

因为在C语言中a[i]只是 *(a+i)的另一种写法。所以如果你对a或i加一个常量将会是同样的效果。对某些C编译器而言，向数组加一个常数将比向下标加一个常数产生稍微快一些的代码。

main函数的大部分，即从第6762到6815行的循环对进程表进行初始化，使其拥有足够的信息来运行系统任务、服务器进程和init。这些进程在启动时都必须存在，而且在整个正常操作过程中任何一个都不会结束。在该循环的开始，rp被设为一个进程表项的地址（第6763行）。因为rp是指向一个结构的指针，所以结构的成员可以通过 rp->p_name来访问，如第6765行所示。在MINIX源代码中广泛使用这种记法。

系统任务当然被编译进与核心相同的文件，关于其堆栈的需求信息放在table.c定义的数组tasktab中。由于任务被编译进核心并且可以调用核心中的任意代码、并可以访问核心中任意处的数据，所以一个独立任务的大小是没有什么意义的，于是每个任务的空间大小域都填入核心本身的长度。sizes数组包含了核心、内存管理器、文件系统和init以块（click）为单位的正文段和数据段大小信息。这些信息在核心开始执行之前由boot添加到核心的数据区，并且在核心看来好象是编译器提供了这些信息。sizes中的前两个数据是核心的正文段大小和数据段大小，接下来是内存管理器的正文段大小和数据段大小，依此类推。如果这四个程序均不使用分开的指令和数据空间，则正文段大小为0，将正文和数据混合在一起作为数据段。对于每个任务都将 sizeindex赋值为0（第6775行），这保证了对于所有任务都取用sizes数组第0个元素（第6783和6784行）的值。第6778行中对sizeindex赋值将给每个服务器进程和init提供它们各自针对sizes数组的下标。

最早的IBM PC机将ROM放在可用内存的高端，在8088 CPU中内存大小的限制是1M字节。现代的PC及其兼容机的内存通常比早期的PC大，但是出于兼容性的考虑，它们仍将ROM放在同样的地址。这样RAM便不再连续，640KB和1MB之间有一块区域是ROM。只要可能，引导监控程序就尽量将服务器进程和init程序装在ROM之上的区域。这主要是为文件系统考虑，因为这样一来就可以将一大块内存作为高速缓存而不会与ROM发生冲突。第6804到6810行之间的条件代码保证了对高端内存的使用被记录在进程表中。

进程表中有两项对应于两个不按通常方式调度的进程，它们分别是IDLE和HARDWARE。IDLE是一个空循环，在系统中无其他进程就绪时就运行它。HARDWARE进程用于计费－它记录中断服务所用的时间。所有其他进程由第6811行的代码放置在适当的队列中。第6811行调用的函数是lock_ready，在对队列进行操作之前它先设置一个锁变量switching，在队列被修改之后则删除该锁。在这一点并不需要上锁和解锁，但这是标准方法，而且没必要为一种只出现一次的情况额外编写代码。

对进程表中的每个表项进行初始化的最后一步是调用alloc_segments。这个过程是系统任务的一部分，但当然此时还没有任务运行，所以在第6814行它是作为一个普通过程调用的。这是一个与机器相关的过程，它将各进程使用的内存段的位置、大小及运行特权级设置到适当的蜮中。对于旧式的不支持保护模式的Intel处理器，它只定义段地址。对一个内存分配方法相异的处理器类型，alloc_segments必须被重写。

一旦对所有任务、服务器和init初始化了进程表，系统基本上就可以运行了。变量bill_ptr标明对哪个进程进行CPU使用的计费，它需要一个初值。该初值在第6818行赋值，此时IDLE是一个合适的选择。随后在调用下一个函数lock_pick_proc时可能会选择其他进程。此时所有的系统任务均已就绪，并且当一个用户进程运行时bill_ptr将被改变。lock_pick_proc的另一项工作是将变量proc_ptr指向下一个运行进程的进程表项。选择下一个运行进程的方法是依次检查系统任务、服务器进程和用户进程队列。这里检查的结果是将proc_ptr指向控制台任务的进程表项，它总是第一个被启动的任务。

最后，main的工作至此结束。在许多C程序中main是一个循环，但在MINIX核心中，它的工作到初始化结束为止。第6822行中对restart的调用将启动第一个任务，控制权从此不再返回到main。

_restart是mpx386.s中的一个汇编语言例程。实际上_restart不是一个完整的函数，它是一个更大的过程的中间入口。我们在下一节将详细讨论该过程。这里仅指出_restart引发一个上下文切换，这样proc_ptr所指向的进程将运行。当 _restart执行了第一次时，我们可以说MINIX正在运行－它在执行一个进程。_Restart被反复地执行，每当系统任务、服务器进程或用户进程放弃运行机会挂起时都要执行_restart，无论挂起原因是等待输入还是在轮到其他进程运行时将控制器转交给它们。

第一个排队的任务（即进程表中占用第0号表项的进程，也即占用最小号表项的进程）总是控制台任务。这样其他任务在启动时便可以用它来报告进度或发生的问题。控制台任务一直运行到因试图接收一条消息而阻塞。然后运行下一个系统任务，一直到它因同样的原因阻塞。最后，所有的任务都将阻塞，这样内存管理器和文件系统便可以运行。这两个进程在首次运行时都进行一些初始化操作，但它们同样也将阻塞。最后init将为每个终端创建一个getty进程，这些getty进程将一直阻塞，直到终端上有键入的输入，在这一点上，第一个用户进行登录。

至此我们已经从三个文件跟踪了MINIX的启动过程，两个使用C语言，一个使用汇编语言。mpx386.s文件中

包含了附加的用于中断处理的代码，这些代码将在下一节中说明。但在继续之前先简短地了解一下这两个C文件中的其余例程。在start.c中剩下的函数有两个：k_atoi（第6594行）用于将字符串转换成一个整数，k_getenv（第6600行）用于在核心的环境中查找数据项，该环境是引导参数的拷贝。这两个函数都是标准函数库例程的简化版，之所以重写它们是为了保持核心的简洁。main.c中只剩下一个过程panic（第6829行），当系统发现无法继续运行下去的故障时将调用它。典型的如无法读取一个很关键的数据块、检测到内部状态不一致、或系统的一部分使用非法参数调用系统的另一部分等。这里对printf的调用实际上是调用printk，这样当正常的进程间通信无法使用时核心仍能够在控制台上输出信息。

2.6.7 MINIX的中断处理

中断硬件的细节与系统相关，但任何系统都必须具有与Intel - 32位CPU功能等价的部件。由硬件设备产生的中断是一些电信号，它们首先由中断控制器进行处理。中断控制器是一片集成电路，它能够检测到许多这类电信号并在处理器的数据总线上为其生成唯一的数据格式。这些数据格式之所以必须唯一是因为：对所有这些设备，处理器本身只能有一个输入，所以它无法辨认需要服务的具体设备。使用32位处理器的PC机通常有两片中断控制器芯片，其中每一个可以处理8个输入，但其中有一片为从片，它的输出线连到主片的一条输入线，这样一共可以挂接15个不同的外部设备，这如图2－33中所示。

图2－33 一台32位Intel PC上的中断处理硬件。

该图中，中断信号出现在右侧的IRQn信号线上。连到CPU INT管脚的连接线通知CPU发生了中断。从CPU发出的INTA（中断应答）信号使负责中断的控制器芯片将数据放在系统数据总线上并通知处理器应执行哪个服务例程。在系统初始化期间，当main调用intr_init时，对中断控制器进行编程。这种编程内容决定了对应于各条输入线的信号将向CPU送出什么样的数据，同时也决定了中断控制器操作所用的其他参数。放在总线上的数据是一个8位（二进制）的数值，它用作对一个表格的索引，该表格最多可包含256项。MINIX的表格中含有56个表项，其中实际用到35项，其余21项保留供MINIX将来扩展使用。在32位的Intel处理器上，这张表中包含中断门描述符，每个中断门描述符是一个含有若干域的8字节结构。

对中断有几种可能的响应方式，在MINIX使用的一种模式中，中断门描述符中最重要的一个域指向服务例程可执行代码段和其中的起始地址。CPU执行被选中的描述符所指向的代码。其结果与执行如下的汇编语言指令完全相同：

int <nnn>

唯一的差别在于，对于硬件中断，<nnn>来自中断控制器芯片的一个寄存器，而不是程序内存中的一条指令。

32位Intel处理器响应中断时的任务切换机制很复杂，改变程序计数器以执行另一个函数只是其中很小的一部分。当CPU在一个进程运行期间接收到一个中断时，它将建立一个新堆栈供中断服务程序使用。该堆栈的位置由任务状态段（Task State Segment - TSS）中的一项决定。整个系统中有这样一个结构，它在cstart调用prot_init时进行初始化，并且在每个进程启动时被修改。其结果是每个中断创建的新堆栈总是从被中断进程的进程表项中stackframe_s结构的结尾处开始。CPU自动地将几个关键寄存器值压入新堆栈，包括用来恢复被中断进程本身堆栈及其程序计数器的那些寄存器。当中断处理例程代码开始运行时，它使用进程表中的这个区域作为其自己的堆栈，同时返回被中断进程所需的大部分信息也已被保存。中断处理例程将其余寄存器的内容压栈，填充stackframe结构，然后切换到一个由核心提供的堆栈以在进行中断服务过程中使用。

中断服务例程的结束操作如下：从核心栈切换回进程表项中的stackfram_s结构（它不必是最后一次中断所创建的那个），显式地弹出非硬件压栈的寄存器值，并执行一条iretd（从中断返回）指令。iretd恢复中断前的状态，恢复被硬件压栈的寄存器，并切换回中断前使用的堆栈。由此可知，中断停止一个进程，而中断服务结束则重启动一个进程。被启动的进程可能不是最近被停止的那个进程。与一般的汇编语言教材中所讲述的较简单的中断机制不同，在中断期间被中断进程的工作堆栈上并不保存任何内容。进一步而言，由于在中断之后堆栈被重新创建于一个已知的位置（由TSS决定），所以多进程的控制得以简化。启动另一个进程所需的是：将堆栈指针指向该进程的stackframe结构，弹出先前显式压栈的寄存器值，然后执行一条iretd指令。

当接收到一个中断时，CPU关掉所有的中断，这保证了进程表项中的stackframe不会溢出。这个过程是自动进行的，但也存在可以关中断和开中断的汇编指令。中断处理例程在切换到位于进程表之外的核心栈之后重新开中断，当然在它切换回进程表中的堆栈之后必须再次关中断。但当它在处理中断时，其他中断可以发生并被处理。CPU跟踪下这些嵌套的中断，并在中断嵌套时使用一种更简单的方法来完成中断服务例程与被中断处理例程之间的来回切换。当正在执行一个中断处理例程（或其他核心代码）时，若接收到新的中断，则并不建立一个新堆栈，CPU把恢复被中断代码所需要的主要寄存器值压入已存在的堆栈。当执行核心代码时若遇到iretd指令，则同样也使用简化的返回机制。作为iretd动作的一部分，处理器通过检查从堆栈弹出的代码段选择符来决定如何处理iretd指令。

前边提到的特权级控制在运行进程和执行核心代码（包括中断服务例程）时对接收到中断作出不同响应。当被中断代码的特权级和中断后即将执行代码的特权级相同时，使用较简单的机制。只有当被中断代码的特权级低于中断服务代码时，才使用较复杂的机制。该机制用到TSS和一个新堆栈。一个代码段的特权级记录在代码段选择符中，并且因为这是中断过程中压栈的内容之一，所以从中断返回时可以检查其内容以决定iretd指令将执行什么操作。当中断服务期间创建一个新堆栈供使用时，硬件检查并保证这个新堆栈起码能够装得下需放在

其中的最小的信息集合。这样就保护了特权级更高的核心代码不会因为调用系统调用的用户进程的堆栈空间不够而崩溃。专门运行多进程操作系统的处理器刻意将这种机制固化在处理器中。

如果你对32位Intel CPU内部工作原理不很清楚，那么这种操作过程可能会使你感到困惑。一般情况下我们尽量避免描述这类细节，但如能理解发生中断和执行iretd指令时所发生的处理细节，则对理解核心如何控制进入和离开图2－2中的“运行”态是非常有帮助的。硬件完成其中许多工作，这极大地简化了程序员的工作，并使得系统更加高效，但却使得通过读软件代码来理解其内部操作变得更加困难。

MINIX核心中只有一小部分感知到硬件中断。这部分代码在mpx386.s中。其中每个中断都有一个入口。从_hwint00到 _hwint07（第6164到6193行）的所有入口点的源代码看起来象是调用hwint_master（第6143行），从 _hwint08到 _hwint15（第6222到6251行）的入口点象是调用hwint_slave（第6199行）。每个入口点好象在该调用中传递一个参数，指明需要服务的设备。实际上这些不是调用，而是宏。由宏hwint_master定义的代码的8份拷贝很类似，只有参数irq不同。同样，hwint_slave的8份拷贝也很类似。这似乎有些浪费，但这些类似的代码非常紧凑。每个宏展开后均不到40字节。在中断服务时速度是最重要的，而且这样作省掉了装入参数、调用子例程和检查参数的开销。

下面将继续把hwint_master作为函数而不是宏来讨论。回忆一下在hwint_master开始执行之前，CPU已经在被中断进程的进程表项的stackframe_s结构中建立了一个新堆栈，而且若干关键寄存器值已经被保存在那里。hwint_master的第一个动作是调用save（第6144行），该子例程将以后重启动被中断进程所需的所有其他寄存器值压栈。save其实可以作为在线代码写成宏的一部分以加快速度，但这样将使宏的大小增加一倍以上，而且save也需要被其他函数使用。我们将要看到save主要围绕堆栈操作。当返回hwint_master时，将使用核心栈而不是进程表项中的stackframe结构。下一步是操作中断控制器以防止从产生当前中断的中断源接收到另一个中断（第6145到6147行）。该操作屏蔽中断控制器对某一特定输入的响应能力；CPU对全部中断的响应能力是在第一次接收到中断信号时继承下来的，此时CPU尚未恢复这种特性。

第6148到6150行的代码将中断控制器复位，然后使CPU再次能够从其他中断源接收中断。然后，第6152行的间接call指令使用当前正被服务的中断编号来对一张与设备相关的低层例程地址表进行索引。它们虽然称作低层例程，但却用C编写。它们执行的典型操作有：为输入设备进行服务，以及将数据传到一个相应任务下次运行时可访问的缓冲区。在从设备相关的低层例程返回之前可能已进行过许多处理操作。

在下一章将看到低层驱动程序代码的示例。但为了理解hwint_master中进行了哪些操作，我们现在指出低层代码可能调用interrupt（在proc.c中，将在下节讨论），并且interrupt将中断转换成一条消息，然后将该消息发向一个系统任务，而该系统任务对激发该中断的设备进行服务。再进一步，对interrupt的调用将调用调度程序，并可能选择该任务作为下一个运行对象。从设备相关代码返回时，处理器响应全部中断的能力再次被第6154行的cli指令屏蔽，并且中断控制器准备在所有中断被再次打开时对激发当前中断的设备作出响应。然后hwint_master以一条ret指令（第6160行）结束。这里看不出使用了什么技巧。如果一个进程被中断，在这里使用的堆栈则是核心栈，而不是启动hwint_master之前由硬件建立的进程表中的堆栈。在本例中，save对堆栈的操作将把 _restart的地址保留在核心栈上。这样的结果是使一个系统任务、或者服务器进程、或者用户进程再次运行。该进程不大会是，实际上基本不会是原先运行的进程。这依赖于特定设备的中断服务例程所产生消息的处理是否改变了进程调度队列。这实际上就是多进程并发执行机制的中心内容。

在结束之前，再讲一下当核心代码正在执行时发生中断的情况，此时正在使用的是核心栈，save将restart1的地址保存在核心栈中。这种情况下，核心先前正在执行的操作将在hwint_master末尾的ret指令之后继续下去。这样中断便可以嵌套，但当全部低层服务例程结束后， _restart将最终执行。而且被中断进程之外的一个进程可能被投入运行。

除了必须将主和从中断控制器全部使能外，hwint_slave（第6199行）很象hwint_master。因为从中断控制器接收到中断将使这两者都被屏蔽。这里有几个细微的汇编语言概念需要说明，首先在第6206行有一句

```
jmp  . + 2
```

该语句指定跳转到紧挨本指令之后的地址。该指令在这里仅仅是为了增加一个小的延迟。最初的IBM PC BIOS设计者认为在连续的I/O指令之间有必要加上延时。尽管并非当前所有的IBM PC兼容机都需要这样处理，我们还是遵循他们的例子。这种细致的技术处理正是为什么有人把硬件设备的编程当作深奥的技巧的原因之一。在6214行有一条条件转移指令，其转移的目标地址为6218行的一条带有数字标号的指令：

```
0:  ret
```

注意该条件转移指令

```
jz  0f
```

并不是象前例那样指定要跳过几个字节。这里的0f并不是一个16进制数，这是MINIX编译器所使用的汇编程序指定一个本地标号的方法。0f表示向前跳转到下一个数字标号0。普通的标号名不允许以数字开头。另外一点有趣但容易混淆之处是同一个标号可以在同一文件的其他地方出现，如在hwint_master的第6160行也出现标号0：。这种情况可能会比表面看上去更复杂，因为这些标号出现在宏中，并且这些宏在汇编器看到该代码之前就被展开。于是汇编器实际上看到的代码中存在16处标号0：。在宏中间声明的标号可能发生的这种增殖情况就是汇编语言提供本地标号的原因。当解析一个本地标号时，汇编器使用指定方向上相匹配的第一个标号，而忽略其在

别处的出现。

现在来看save（第6261行），对它我们已经提到过几次。它的名字指出了它的若干功能之一，即将一个中断进程的上下文保存在CPU提供的堆栈上，该堆栈是进程表中的一个stackframe结构。save用变量_k_reenter来计算和确定中断的嵌套级数。如果当前中断发生时一个进程正在运行，则第6274行的指令

```
mov  esp,  k_stktop
```

切换到核心栈，并且随后的指令将 _restart的地址压栈（第6275行）。否则的话，核心栈已在使用中，则将restart1的地址压栈（第6281行）。不论哪种情况，所使用的堆栈与入口时起作用的堆栈可能不同，而且调用它的例程中的返回地址被压在已压栈寄存器值的下边。这些都造成一条普通的return指令不足以返回到调用者。结束save操作的两个出口点指令

```
jmp  RETADR_P_STACKBASE (eax)
```

（位于第6277和6282行）使用调用save时压进栈的地址。

mpx386.s中的下一个过程是_s_call，它从第6288行开始。在讲述其内部细节之前，先看看它是如何结束的。其结束处没有ret 或jmp指令。在第6315行用cli指令关中断之后，其流程转到_restart继续执行。_s_call是中断处理机制在系统调用中的对应物。在一软件中断，即执行一条int nnn指令后，控制权转到_s_call。对软件中断的处理类似硬件中断，不同之处在于用 int nnn指令中的nnn 作为对中断描述符表的索引，而不是中断控制器芯片提供的一个数字。这样当进入_s_call时，CPU已经切换到了进程表（由TSS提供）中的栈，并且几个寄存器值已经被压入该栈。截止到调用_restart时，对_s_call的调用在经过_restart后最终以一条iretd指令结束，而且正如硬件中断一样，该指令将启动proc_ptr此时指向的进程。图2-34对硬件中断和使用软件中断机制的系统调用作了比较。

图 2-34 (a)硬件中断的处理过程。 (b)系统调用处理过程。

现在来看_s_call的细节。它还有另一个名字_p_s_call，这是16位MINIX版本的残迹，16位版本的MINIX分别具有保护模式和实模式操作例程。在32位版本中，对这两个标号的调用都到达这里。使用系统调用的程序员用C写的函数调用看起来与其他函数调用一样，不论调用的是一个本地定义的函数还是一个C库中的例程。支持一条系统调用的库函数代码构造一条消息，将消息的地址和目标进程标识号装入CPU的寄存器，然后调用一条int SYS386_VECTOR指令。如上所述，其结果是将控制转到 _s_call的起始处，同时几个寄存器值已被压入进程表中的一个堆栈上。

_S_call代码的第一部分看起来象save函数的在线展开，并且将其余必须保存的寄存器值保存下来。正如在save中一样，执行一条

```
mov  esp,  k_stktop
```

指令，这样就切换到了核心栈，同时重新开中断（软件中断与硬件中断的相似之处还包括它们都关中断）。其后将调用 _sys_call，下节中将对其进行讨论。现在只需知道它将构造一条消息，然后发送该消息，这进一步将引发调度程序运行。于是，当_sys_call返回时，proc_ptr有可能指向引发本系统调用的进程之外的另一个进程。在执行到达restart之前，一条cli指令将关中断以保护即将被再次启动进程的stackframe结构。

我们已经看到有几种方法使执行线路到达_restart函数（第6322行）：

1 在系统启动时从main调用它。

2 在硬件中断发生后从hwint_master 或 hwint_slave跳转到它。

3 在系统调用发生后从 _s_call调用它。

在以上所有情况中调用_restart时都要关中断。如果_restart检测到存在未挂起的被服务的中断，这些中断是在处理其他中断期间到达的，则它将调用unhold，这样就允许在任何进程被重新启动之前将其他中断转换为消息。这将暂时地重新开中断，但在unhold返回之前将再次关中断。截至第6333行，下一个将运行的进程已经被选中。这时关中断将保证该进程不会被改变。进程表的结构是精心设计的，它以一个stackframe结构开始，而且该指令

```
mov  esp, (_proc_ptr)
```

使CPU的栈指针寄存器指向stackframe，指令

```
lldt  P_LDT_SEL (esp)
```

随后从stackframe中装入处理器的本地描述符表寄存器。这使得处理器为使用将运行进程的存储器段作好准备。接下来的指令将把该地址保存在将运行进程的进程表项中，下一次中断的堆栈将建立在该表项中。然后，随后的一条指令把这个地址保存在TSS中。当执行核心代码（包括中断服务代码）时发生中断，因为将使用核心栈，所以_restart的第一部分便不再需要。而中断服务结束应允许核心代码继续执行。标号restart1（第6337行）标示出在这种情况下应从何处继续执行。在这一点上，k_reenter被减1，以记录一层可能的嵌套中断已被处理，然后其余指令将处理器恢复到下一进程上一次执行所处的状态。倒数第二条指令修改栈指针，这样使调用save时压栈的地址被忽略。如果最后一重中断是在进程执行时发生的，那么最后一条指令iretd将结束这一系列操作返回到进程接下来应该执行的地方，恢复其剩余的寄存器，包括堆栈段寄存器和栈指针。但是如果遇到经由了restart1的iretd，则使用的核心栈就不是一个stackframe，而是核心栈。而且这种情况也就不再是向一个被中断进程的返回，而是在核心代码执行期间所发生中断的完成。在iretd执行期间，当代码段描述符从堆栈弹出时

CPU将检测到这一点，而且在这种情况下，iretd的全部动作是使核心栈继续保持使用。

关于mpx386.s还有一些内容。除了硬件和软件中断之外，CPU内部的各种错误条件可能激发异常(exception)。异常并不总是坏事，它们可以支持操作系统提供一种服务，例如为进程提供更多的内存，或者将当前换出的内存页面换入等，尽管标准的MINIX并未实现这些服务。但异常发生时不应将其忽略。异常采用和中断相同的处理机制，使用中断描述符表中的描述符。该表中的表项指向16个异常处理入口。这些入口从_divide_error开始，以_copr_error结束，位于mpx386.s尾部的第6350到6412行。根据是否将一个错误码压栈，这些入口分别跳转到exception（第6420行）或errexception（第6431行）。此处汇编代码的处理与我们已经看到的很相似，寄存器被压栈并且调用C例程 _exception（注意其中的下划线）来处理该事件。异常导致的结果各不相同，有的被忽略、有的导致系统崩溃、有的导致向进程发消息。在下一节中将讨论_exception。

还有另外一个入口点象中断一样处理，即 _level0_call（第6458行）。其功能在下一节讨论，在那里我们将讨论它跳转到的代码 _level0_func。该入口点与中断和异常的入口点同放在mpx386.s中是因为它也被int指令所调用。和异常处理例程一样，它也调用save，因此跳转到这里的代码最终也将以一条ret指令结束，而这条ret指令将发展到调用_restart。mpx386.s中最后一个可执行函数是 _idle_task（第6465行），这是一个空循环，在系统中没有就绪进程时则调用它。

最后，在该汇编语言文件的末尾预留了一些数据存储空间。这里定义了两个不同的数据段。在6478行声明的数据段

```
.sect .rom
```

保证该存储空间被分配在核心数据段的最开始处。编译器在这里放置一个魔数以便boot能够验证其装入的文件是一个合法的核心映像。正如在讨论核心数据结构时所讲的，boot随后将用 _sizes数组覆盖这个魔数及其后的空间。这里将为总共有16项的_sizes数组分配足够的空间，以便向MINIX中增加其他的服务器进程。在

```
.sect .bss
```

（第6483行）定义的其他数据存储区，在核心通常未初始化变量区为核心栈和异常处理例程预留空间。服务器进程和普通的用户进程在可执行文件被链接时预留堆栈空间，并在执行时依靠核心来适当地设置堆栈段描述符和堆栈指针，核心也必须为它们自己进行这种处理。

2.6.8 MINIX的进程间通信

MINIX中的进程使用消息进行通信，这里使用到进程会合的原理。当一个进程执行SEND时，核心的最底层检查目标进程是否在等待从发送者（或任一发送者）发来的消息。如果是，则该消息从发送者的缓冲区拷贝到接收者的缓冲区，同时这两个进程都被标记为就绪态。如果目标进程未在等待消息，则发送者被标记为阻塞，并被挂入一个等待将消息发送到接收进程的进程队列中。

当一个进程执行RECEIVE时，核心检查该队列中是否存在向它发送消息的进程。若有，则消息从被阻塞的发送进程拷贝到接收进程，并将两者均标记为就绪；若不存在这样的进程，则接收进程被阻塞，直到一条消息到达。

进程间通信的高层代码在proc.c中。核心的任务是将一个硬件中断或软件中断转换为一条消息，前者由硬件产生，后者则是请求系统服务（即系统调用）的途径。这两者很类似，以至可以用同一个函数处理，但将其分成两个专门的函数会更高效。

首先看interrupt（第6938行）。在接收到一条硬件中断后，相应设备的低层中断服务例程调用该函数。interrupt的功能是将中断转换成向该设备所对应的系统任务发送一条消息，而且通常在调用interrupt之前几乎不进行什么操作。例如，针对硬盘驱动器的整个低层中断处理例程只包含下面三行：

```
w_status = in_byte ( w_wn->base + REG_STATUS);  /* acknowledge interrupt */
interrupt (WINCHESTER);
return 1;
```

假如不需要读取硬盘控制器上的一个I/O端口以获得状态信息，则对interrupt的调用可以放在mpx386.s中，而不是象现在这样放在at_wini.c中。interrupt所做的第一件事是通过变量k_reenter（第6962行）来检查在接收当前中断时是否已经有一个中断正在被处理。若是，则将当前中断排队，同时interrupt返回。当前中断将在以后调用unhold时处理。下一个动作是检查该任务是否正在等待一个中断（第6978到6981行）。如果任务未作好接收中断的准备，则其p_int_blockd标志被置位－后边我们将看到这将使得丢失的中断可能被恢复，并且不发送消息。如果通过了这个测试则消息被发送。从HARDWARE向系统任务发送消息是很简单的，因为任务和核心是编译在同一个文件中的，因此可以访问相同的数据区域。第6989到6992行的代码完成消息的发送，其操作步骤是:在目标任务的消息缓冲区的源和类型域中填入内容,将目标进程的RECEIVING标志复位并将该任务解除阻塞。一旦该消息就绪则目标任务被调度运行。在下一节我们将详细讨论进程调度，在interrupt中第6997到7003行的代码仅仅是一个预览－这是被用来将一个进程排队的过程ready的在线代换。这里的代码很简单，因为从中断产生的消息只发送到系统任务，这样便无需确定要操作的进程队列是三种队列中的哪一个。

proc.c中下一个函数是sys_call。它与interrupt类似：它将一个软件中断（即用来激发一条系统调用的int SYS386_VECTOR指令）转换为一条消息。但由于在这种情况下源和目的的可能范围很宽，并且该调用可能需

要发送、或接收、或既发送又接收一条消息，所以sys_call要做的事情更多。与多数情况一样，这意味着sys_call的代码比较简短，因为它的工作通过调用其他过程来完成。首先调用isoksrc_desk，这是proc.h（第5172行）定义的一个宏，它与proc.h（第5171行）中定义的另一个宏isokprocn协同工作。其功能是检查并保证该消息指定的源进程和目标进程合法。在第7026行一个类似的测试，isuserp（也是proc.h中定义的一个宏）检查该调用是否从一个用户进程发出，若是，则它应该先申请发出一条消息，随后接收一条应答，对用户进程而言，这是唯一一种允许的调用方式。这里所测试的错误是不大会发生的，但这样的测试作起来也很容易，因为它们最终编译成的代码只是对几个小整数进行一下比较。在这样的层次上，操作系统对不大可能发生的错误进行测试是应该提倡的。这段代码在系统活跃期间很可能每秒钟要执行许多次。

最后，如果该调用需要发送一条消息，则调用mini_send（第7031行），而如果需要接收一条消息，则调用mini_rec（第7039行）。这两个函数是MINIX中正常消息传递机制的核心，应加以仔细的研究。

Mini_send（第7045行）有三个参数：调用进程、目标进程、及指向消息所在缓冲区的指针。它进行许多测试。首先它确认用户进程只能向文件系统或内存管理器发消息。在第7060行，用宏isuserp对参数caller_ptr进行测试以确定调用进程是否用户进程;用一个类似的函数issysentn对参数dest进行测试以确定它是文件系统或内存管理器。如果这两者的组合是不允许的，那么mini_send将以出错结束。

下一个检查内容是确认消息的目标进程是一个活跃进程，而不是进程表中的一个空表项（第7062行）。在第7068到7073行mini_send检查该消息是否完全处于用户的数据段、代码段或这两者的间隙中。如不是则返回一个错误码。

接下来对一个可能发生的死锁进行测试。在第7079行上有一个测试，它确保该消息的目标进程没有正在试图向调用进程发送一条反向的消息。

mini_send中最关键的测试是在7088到7090行。这里将检查目标进程是否正阻塞在RECEIVE上，这由进程表项中的p_flags域中的RECEIVING位标志。如果它正在等待，则接着问：“它在等谁？”。如果它正在等待这一条消息的发送者或任一进程，则执行CopyMess来拷贝该消息，然后将接收者的RECEIVING位复位使其解除阻塞。copyMess在第6932行被定义为一个宏，它调用文件klib386.s中的汇编语言例程。

另一方面，如果接收者没有阻塞，或者被阻塞但在等待另一个进程的消息，则执行第7098到7111行的代码以阻塞发送者并将其排队。试图向一个给定目标进程发送消息的所有进程被链在一个链表上，目标进程的p_callerq域指向其队首进程的进程表项。图2－35（a）示出了当进程3无法向进程0发送消息的情况。如果随后进程4也无法向进程0发送消息，则情况如图2－35（b）所示。

图 2－35 试图向进程0发送消息的进程被排队。

当sys_call的参数function为RECEIVE或BOTH时它将调用mini_rec（第6119行）。第7137到7151行的循环遍历所有试图向接收者发送消息的进程，以检查其中是否有可接受的。若发现一个，则消息从发送者拷贝到接收者，随后发送者被解除阻塞，状态变为就绪，并从该队列中删除。

如果未发现合适的发送者，则检查接收进程的p_int_blocked标志是否指明对应该目标进程的一个中断先前被阻塞了（第7154行）。如果是则构造一条消息－因为来自HARDWARE的消息无非就是其源进程域为HARDWARE，其类型域为HARD_INT，这里没必要调用CopyMess。

如果未找到被阻塞的中断，则所期望的消息源和缓冲区地址被保存在其进程表项中，并通过将其RECEIVING位置位来将其标志为阻塞。第7165行对unready的调用将接收者从就绪进程队列中删除。如进程的p_flags中有另外一位置位，那么一个信号可能正被挂起，于是该进程应很快有另一个运行机会以处理该信号，所以不调用unready，以避免阻塞该进程。

mini_rec的倒数第二条语句（第7171和7172行）用来处理核心产生的SIGINT、SIGQUIT和SIGALRM信号。当其中一个发生时，如果内存管理器正在等待一条来自ANY的消息，则向其发送一条消息。否则，该信号被核心记录下来直到内存管理器试图从ANY接收消息为止。该条件的测试在这里进行，并在需要的情况下调用inform来向其告知被挂起的信号。

2.6.9 MINIX的进程调度

MINIX使用与图2－26所示结构密切相关的多级调度算法。在该图中我们看到I/O系统任务位于第2层，服务器进程在第3层，用户进程在第4层。如图2－36所示，调度程序维护三个可运行进程队列，每个队列对应一层。数组rdy_head为每个队列设一项，它指向队首的进程。类似地，rdy_tail数组中的项指向队尾的进程。这两个数组都在proc.h中被定义为EXTERN类型（第5192到5193行）。

图 2－36 调度程序维护三个队列，每个优先级对应一个。

当一个阻塞进程唤醒时，它被追加到其队列的尾部。由于存在rdy_tail队列，该追加操作速度很快。当一个运行中的进程阻塞、或一个就绪进程被一个信号撤销时，该进程从调度程序的队列中删除。只有就绪进程被排队。

有了上述队列结构之后，调度算法就很简单了：找到优先级最高的非空队列，并选择队首进程即可。如果所有队列均为空，则运行IDLE进程。在图2－36中TASK_Q具有最高优先级。调度代码在proc.c中，选择最高优先级队列由pick_proc（第7179）完成。该函数的主要工作是设置proc_ptr。任何影响到选择下一个运行进程的对这些队列的改变都要再次调用pick_proc。无论当前进程在什么时候阻塞，都调用pick_proc来重新调度CPU。

pick_proc很简单。对每个队列都要进行检测。首先是TASK_Q，若其中有一个进程就绪，则pick_proc将设置proc_ptr并立即返回。接着检测SERVER_Q，同样若有进程就绪，则设置proc_ptr并返回。如果USER_Q上有一个就绪进程，则bill_ptr将指向该进程以对其即将使用的CPU时间进行计费（第7198行）。这保证了对于一个即将运行的用户进程，将把系统为它所作的全部工作都计在它的帐上。如果没有如何一个队列具有就绪任务，则第7204行将把计费转到IDLE进程并调度它运行。被选中执行的进程并不仅仅因为被选中而从其所在的队列中删除。

过程ready（第7210行）和unready（第7258行）分别用来将一个可运行进程挂入队列和将一个不再就绪的进程从其队列中删除。如同我们已看到的mini_send和mini_rec均调用ready，它本来也可以被interrupt调用，但为了加快中断处理的速度，在interrupt中将这部分代码写成了在线代码。ready对三个进程队列之一个进行操作，它直接将进程追加到队列的尾部。

unready也对队列进行操作，通常它是将队列头部的进程去掉，因为一个进程只有处于运行状态才可被阻塞。这种情况下，如第7293行的例子所示，unready在返回之前要调用pick_proc。一个未在运行的用户进程若收到一个信号也可能进入非就绪状态，若该进程没在队首，则将遍历整个USER_Q队列来查找它，一旦找到则将其删除。

尽管多数调度决策是在一个进程阻塞或解除阻塞时作出的，但调度仍要考虑到当前用户进程时间片用完的情况。这种情况下，时钟任务调用sched（第7311行）来将USER_Q队首的进程移到队尾。该算法的结果是将用户进程按时间片轮转方式运行。文件系统、内存管理器和I/O任务绝不会被放在队尾，因为它们肯定不会运行得太久。这些进程可以被认为是极其可靠的，而且在完成要做的工作后将阻塞。

在proc.c中还有另外几个进程调度的支撑例程。其中五个：lock_mini_send、lock_pick_proc、lock_ready、lock_unready和lock_sched在调用相关的函数之前用变量switching执行一个加锁操作，在结束时再解锁。proc.c的最后一个函数unhold在讨论mpx386.s中的函数_restart时就已经提到过。它遍历被挂起的中断队列，对其中每个调用interrupt，其目的是在另一个进程被允许运行之前将每一条挂起的中断转换成一条消息。

概括起来，调度算法维护三个优先级队列，分别对应I/O任务、服务器进程和用户进程。最高优先级队列的第一个进程总是被选中执行。系统任务和服务器进程被允许执行到其阻塞为止，但时钟任务监视用户进程所使用的时间。若用户进程的时间片用完，它被挂在其队列的尾部，这样便在相互竞争的用户进程中达到了时间片轮转的调度效果。

2.6.10 与硬件相关的核心支持

有几个C函数与硬件关系极为密切。为了便于将MINIX移植到其他平台，这些函数被隔离出来放在本节讨论的exception.c、i8259.c和protect.c中，而不是和受它们支持的高层代码放在相同的文件中。

exception.c包含异常处理程序exception（第7512行）。该例程被mpx386.s中的异常处理代码的汇编语言部分调用（作为_exception调用）。由用户进程引起的异常被转换成信号。用户自己编写的程序是可能有错的，但由操作系统本身引起的异常则表明发生了严重错误，并产生了不可恢复的故障。数组ex_data（第7522到7540行）确定发生这类错误时应打印的错误信息，或是对每种异常应向用户进程发送的信号。早期的Intel处理器并不能产生所有这些异常，该数组每项的第三个域指出能产生各个异常的最低的处理器型号。这个数组提供了已经实现了MINIX的Intel处理器家族进展的一个有趣的总结。如果严重错误是由所使用的处理器不支持某种中断而引起，则7563行将打出一条建议更换CPU的信息。

i8259.c中的三个函数用于系统初始化过程以对Intel 8259中断控制器芯片进行初始化。intr_init（第7621行）完成中断控制器的初始化，它向控制器的几个端口写数据。有几行代码对由引导参数导出的一个变量进行测试，例如在第7637行对第一个端口写数据以适应不同型号的计算机。在第7638和7644行对参数mine进行测试，并将对MINIX和BIOS ROM均适合的一个数值写入该端口。当推出MINIX时，可调用intr_init来恢复BIOS向量，这样便可以平滑地退回到引导监控程序。Mine选择所使用的模式。要想理解这里的完整细节需要仔细研究8259芯片，所以就不再详细地讲述。我们应指出第7642行对out_byte的调用使主片只接受从片来的输入，而对其他输入均不予响应。同时第7648行的类似操作使从片对其输入不予响应。并且该函数的最后一行预先将下一个函数spurious_irq（第7657行）的地址装入irq_table的每一项中，这保证在真正的处理程序被安装之前所产生的中断将不会产生任何副作用。

i8259.c的最后一个函数是put_irq_handler（第7673行）。在初始化期间那些必须响应中断的任务调用它以将其自己的处理程序地址装入中断表，覆盖掉spurious_irq的地址。

protect.c包含与Intel处理器保护模式相关的例程。全局描述符表（GDT）、本地描述符表（Local Descriptor Tables - LDTs）和中断描述符表都位于内存中，它们提供对系统资源的受保护的访问。GDT和LDT被CPU中特殊的寄存器所指向，GDT表项指向LDT。GDT对所有进程均可用并记录了操作系统使用的内存区域的段描述符。通常对每个进程有一个LDT，它记录了该进程所使用内存区域的段描述符。描述符是一个包含许多内容的8字节结构，但段描述符中最重要的是描述一个内存区域的基址和限长域。IDT也由8字节的描述符构成，其中最重要的部分是当中断被激活时所执行代码的地址。

prot_init（第7767行）由start.c调用以建立GDT（第7828到7845行）。IBM PC BIOS要求它按一种固定的方式排序，所有对它的索引都定义在protect.h中。每个进程LDT的空间都在进程表中分配。每个包含有两个描

述符，分别是代码段和数据段描述符－记住这里讨论的段由硬件定义。这些段与操作系统中的段不同，操作系统中的段将硬件定义的数据段进一步分为数据段和堆栈段。从第7851到7857行，每个LDT的描述符放在GDT中。这些描述符实际上由函数init_dataseg和init_codeseg建立。在进程内存映像改变时LDT中的表项本身被初始化（即执行EXEC系统调用时）。

另一个需要进行初始化的处理器数据结构是任务状态段（TSS），该数据结构在本文件的开头定义（第7725到7753行），并为处理器寄存器和其他任务切换时应保存的信息提供空间。MINIX只使用某些域的信息，这些域定义了当发生中断时在何处建立新堆栈。在第7867行对init_dataseg的调用保证它可以用GDT进行定位。

为了理解在最底层上MINIX是如何工作的，可能最重要的是：理解异常、硬件中断、或是int <nnn> 指令如何激活那些事先写好的服务代码的执行。这通过中断门描述符表完成，数组gate_table（第7786到7818行）由编译器用异常和硬件中断处理例程的地址来初始化，随后它被第7873到7877行的循环用来对中断门描述符的大部分进行初始化，具体通过调用int_gate函数完成。余下的向量SYS_VECTOR、SYS386_VECTOR和LEVEL0_VECTOR需要不同的特权级，它们由该循环后边的代码进行初始化。

有充分的理由解释为什么按照描述符方式来组织数据，其原因源于硬件细节，以及维持先进的处理器与16位的286处理器之间的兼容性这两个方面。幸运的是我们通常可以将这些细节留给Intel处理器的设计者。对其中大部分而言，C语言允许我们回避具体的细节，但在实现一个真正的操作系统时不可避免地要在某些点上面对这些细节。图2－37示出了一种段描述符的内部结构。注意在C程序中可被引用为简单的32位无符号整数的基地址被分成三部分，其中两部分被许多1位、2位或4位的二进制组合隔开。20位的限长被分成16位和4位两部分。限长可被解释为字节数、或者是以4096字节为一页的页面数，这具体依赖于G（粒度）位。其他描述符，例如那些用来指定中断处理方式的描述符，具有不同的结构，但其结构也同样地复杂。第4章将更加详细地讨论这些描述符。

图 2－37 一个Intel处理器的段描述符格式。

protect.c中定义的大部分其他函数，用于将C程序中使用的变量和类似于图2－37所示的供机器使用的格式复杂的描述符数据之间进行转换。init_codeseg（第7889行）和init_dataseg（第7906行）在操作上很类似，都用于将传给它们的参数转换成段描述符。而它们则调用下一个函数sdesc（第7922行）来完成此项工作。sdesc函数的工作就是对图2－37所示的繁琐的结构进行具体处理。在系统初始化期间并不使用init_codeseg和init_dataseg。此外，当启动新进程时，为了给进程分配适当的存储器段，系统任务将调用它们。seg2phys（第7947行）只被start.c调用，它执行sdesc的逆操作，即从一个段描述符中析取出段基地址。int_gate（第7969行）在为中断描述符表构造表项时执行一个类似于init_codeseg和init_dataseg的函数。

protect.c的最后一个函数enable_iop（第7988行）使用了一种不大规范的技巧。我们已经在几个地方指出操作系统的功能之一是对系统资源进行保护，而MINIX使用的方法之一是利用处理器特权级来禁止用户程序执行某些指令。然而MINIX同时也希望能够在小规模的系统上运行，这些系统可能只有一个用户或少许几个相互信任的用户。在这样的系统中，用户写的程序很可能会直接访问I/O端口，例如用于科学实验中的数据采集。文件系统中有一个小秘密－当文件 /dev/mem或 /dev/kmem被打开时，内存管理器将调用enable_iop，它将为执行I/O操作而改变处理器特权级，以允许当前进程执行读写I/O端口的指令。要想描述清楚设置这些函数的意图比描述它们更为复杂。该函数实际上只是将其调用进程stackframe项中一个字的两个比特置位，这个字在进程下次运行时将被装入CPU状态寄存器。因为它只对当前的调用进程有效，所以其他函数没必要撤销该操作。

2.6.11 公用程序和核心库

最后，核心有一个用汇编语言写的支撑函数库，及几个位于文件misc.c中的C实用程序，这些汇编语言函数在编译klib.s时被包含进来。先看汇编语言文件。klib.s（第8000行）是一个与mpx.s类似的短文件，mpx.s的功能是根据WORD_SIZE的定义来选择适当的机器版本。我们将讨论的代码在klib386.s（第8100行）中。其中有近24个实用例程，之所以用汇编语言是出于效率的原因，或是因为它们根本就无法用C来编写。

_monitor（第8166行）使系统能够返回到引导监控程序。在引导监控程序看来，整个MINIX是一个子例程，并且当MINIX启动时，返回引导监控程序的地址被留在其堆栈中。_monitor只需恢复各段选择符和MINIX启动时保存下来的栈指针，然后与从其他子例程同样地返回即可。

下一个函数 _check_mem（第8198行）用于在系统启动时确定内存大小。它对每16个字节构成的单元进行一个很简单的检测，检测有两种模式：将每个比特置成“0”和“1”。

尽管可以使用 _phys_copy（如下所示）来进行消息的拷贝，然而有一个更快的专门过程 _cp_mess（第8243行）被用作此目的，它的调用方式为：

cp_mess (source, src_clicks, src_offset, dest_clicks, dest_offset);

这里source 为发送者的进程号，它被拷贝到接收者缓冲区的m_source域。源和目标地址以一个块号来指定，典型地为包含该缓冲区的段基地址，及一个相对于该块的偏移。这种指定源和目标的方式比 _phys_copy所用的32位地址更高效。

定义 _exit、__exit和 ___exit（第8283到8285行）是因为编译MINIX时可能用到的一些库例程需调用标准C函数exit。从核心退出是一个没有意义的概念，从核心退出后便无处可去。这里所用的解决办法是开中断，然后进入一个无休止的循环。最终一个I/O操作或时钟将发出一条中断，由此恢复正常的系统操作。入口点

___main（第8289行）是处理编译操作的另一种方法。该方法在编译一个用户程序时很合理，但在核心中则没什么用处。它指向一条汇编语言的ret（从子例程返回）指令。

_in_byte（第8300行）、_in_word（第8314行）、_out_byte（第8328行）和_out_word（第8342行）提供了对I/O端口的访问方法。在Intel硬件中I/O端口使用与内存分开的地址空间，并使用与内存访问不同的指令。_port_read（第8359行）、_port_read_byte（第8386行）、_port_write（第8412行）和_port_write_byte（第8439行）完成数据块在I/O端口之间的传送。这种传送主要用于与磁盘相关的数据传递，这种操作必须比其他I/O调用的速度快。面向字节的版本在操作中以8位而不是16位来读数据是为了与老的8位的外部设备兼容。

有时一个系统任务需要临时地关中断，该操作是通过调用_lock（第8462行）完成的。当中断可被重新打开时，该任务调用 _unlock（第8474行）来开中断。所有这些操作是由一条机器指令完成的。与之相比，_enable_irq（第8488行）和 _disable_irq（第8521行）的代码则更为复杂。它们工作在中断控制器芯片的层次来使能和禁止单个的硬件中断。

_phys_copy（第8564行）在C程序中使用如下语句进行调用：

```
phys_copy (source_address, destination_address, bytes);
```

它将物理内存中任意处的一个数据块拷贝到任意的另外一处。其中，两个地址都是绝对地址，也就是地址0确实表示整个地址空间的第一个字节，并且三个参数均为无符号长整数。

接下来的两个小函数专用于Intel处理器。_mem_rdw （第8608行）返回内存中任意处的一个16位的字。其结果用0扩展后放在32位长的eax寄存器中。_reset函数（第8623行）使处理器复位，这通过将一个空指针装入处理器的中断描述符表寄存器，然后执行一个软件中断来实现，它与硬件复位效果一样。

下面两个例程支持视频显示器，它们被控制台任务使用。 _mem_vid_copy（第8643行）将一个字串从核心的内存区域拷贝到视频显示器的存储器中，该字串中包含替换字符码和若干属性字节。_vid_vid_copy（第8696行）完成显示器存储器内部的数据块拷贝。在某种程度上它更复杂一些，因为目的数据块可能与源数据块之间有重叠，并且数据移动的方向非常关键。

文件中最后一个函数是 _level0（第8773行），它允许在需要时任务可拥有最高特权级 — 0级。它用于这样的操作，如复位CPU、或访问PC的ROM BIOS例程。

misc.c中的C语言公用例程具有专门功能。mem_init（第8820行）仅在MINIX首次启动时被main调用。在一台IBM PC兼容机上可能存在两个或三个不连续的内存区域。PC用户称为“常规”内存的最低端内存的大小，以及从PC ROM之上开始的内存区域（“扩展”内存）由BIOS告知引导监控程序，引导监控程序随后又将其作为引导参数传递。该参数由cstart进行解释并在引导时写入low_memsize和 ext_memsize。第三个区域是“影子”内存，BIOS ROM可被拷贝到该区域以提高性能，因为ROM通常比RAM的访问速度慢。由于MINIX一般不使用BIOS，所以mem_init力图定位该片内存并将其加入它的可用内存区中，这是通过调用check_mem来完成的。

下一个例程env_parse（第8865行）也是在启动时使用。引导监控程序可以在引导参数中将形如“DPETH0＝300：10”的任何字符串传给MINIX。env_parse试图找到一个第一个域与其第一个参数env匹配的串，然后析取出所需的域。代码中的注释解释了该函数的用途，主要是为了帮助用户增加一些需要参数的新驱动程序。例中的“DPETH0”用于向支持网络的MINIX传递配置信息。

本章讨论的最后两个例子是bad_assertion（第8935行）和bad_compare（第8947行）。只有在宏DEBUG被定义为TRUE时才对其进行编译。它们支持assert.h中的宏。尽管本书中的代码并不引用这两个例程，但对于那些想构造自己修改了的MINIX的读者，它们可以帮助进行调试。

2.7 小结

为了屏蔽中断的影响，操作系统提供了一个由并发运行的顺序进程所构成的概念模型。进程之间可以使用进程间通信原语来相互通信，比如信号量、管程、消息等。使用这些原语是为了保证任一时刻不会同时有两个进程进入临界区。进程的状态可以是运行、就绪、或阻塞，且当它或另一个进程执行一条进程间通信原语时可能引起状态的改变。

进程间通信原语可用来解决诸如生产者－消费者、哲学家用餐、读者－写者以及理发师睡觉等问题。但即使使用了这些原语，也必须注意避免错误和死锁。目前已经有许多种调度算法，包括时间片轮转、优先级调度、多级队列以及策略驱动的调度等。

MINIX支持进程概念，并提供了用于进程间通信的消息。消息是不加以缓冲的，所以一条SEND操作只有在接收者正在等待它时才成功。同样地，一条RECEIVE操作只有在消息已经可用时才成功。这两种操作的调用进程在操作不成功时都将阻塞。

当中断发生时，核心的最底层将构造一条消息，并将消息发送到与中断设备相关的系统任务。例如，当磁盘任务读写一块数据时，它先向硬盘控制器发出一条命令，然后调用receive并阻塞。当数据准备好之后硬盘控制器将发出一条中断。低层软件随后构造一条发给硬盘任务的消息并将其标志为就绪。当硬盘任务被再次调度运行时，它将获取该消息并进行处理。中断处理例程当然也可以直接进行一些操作，例如时钟中断处理程序更新时间。

中断可能引发任务切换。当一个进程被中断时，该进程的进程表项中将建立一个堆栈，再次启动该进程所需的全部信息都放在这个新堆栈上。通过以下操作可以再次启动任何一个进程：将栈指针指向其进程表项，然

后执行一个指令序列来恢复CPU寄存器值，最后以一条iretd指令结束。调度程序决定堆栈指向哪个进程表项。

核心本身在运行时也可能发生中断。CPU检测到中断后将使用核心栈，而不是进程表中的堆栈。这样便允许中断嵌套。当后来的中断服务例程结束后，在它之前执行的中断服务例程就可以运行直到结束。在所有中断都被处理之后则可以重启动一个进程。

MINIX调度算法使用三个优先级队列。优先级最高的是系统任务；下一个是文件系统、内存管理器及其他服务器进程；最低的是用户进程。用户进程每次运行一小段时间，这样便达到时间片轮转的调度效果。但其他进程则一直运行直到阻塞或被剥夺为止。

习　题

1 假设你正在设计一种先进的计算机体系结构，它使用硬件而不是中断来完成进程切换，则CPU需要哪些信息？请描述用硬件完成进程切换的工作过程

2 目前的计算机上，中断处理程序至少有一小部分用汇编语言编写，为什么？

3 书中认为图2－6（a）的模型不适用于在内存进行高速缓存的文件服务器，为什么？可否使每个进程拥有自己的cache？

4 在使用线程的系统中，是每个线程有一个堆栈还是每个进程有一个堆栈，说明原因。

5 什么是竞争条件？

6 写一个shell程序，其功能是产生一个内容为一个整数序列的文件，要求它先读取文件中的最后一个整数，将其加1，然后将这个新整数追加到文件的末尾。在系统前台和后台同时运行它并使用相同的文件名。在因竞争而造成的故障发生之前它能运行多久？此处的临界区是什么？请对其进行修改以防止发生竞争。

（提示：使用

```
ln  file  file.lock
```

来将数据文件加锁）

7 对于前一问题，如下的语句

```
ln  file  file.lock
```

是一种很有效的加锁机制吗？说明原因。

8 两个进程在一台共享内存的多处理机（即共享同一个存储器）上运行时，图2－8所示的采用变量turn的忙等待方案还奏效吗？

9 现有一台计算机，不具备TEST AND SET LOCK指令，但有一条指令可以按原子操作方式将一个寄存器的值和一个存储器字进行交换，能否利用该指令写一个与图2－10中enter_region类似的例程？

10 给出一个框架，来描述一个可禁止中断的操作系统如何实现信号量。

11 请说明如何只利用二进制信号量和普通的机器指令来实现计数信号量(即可以拥有任意大的数值的信号量)。

12 在 2.2.4节中描述了一个高优先级进程H和低优先级进程L的情况，它最终导致H陷入死循环，若采用时间片调度而不是优先级调度，还能发生这种情况吗？请进行讨论。

13 管程内部的同步机制使用条件变量和两个特殊操作WAIT和SIGNAL，一种更一般的同步方式只有一条原语WAITUNTIL，它以任意的布尔表达式作为参数。例如

```
WAITUNTIL  x<0  or y+z<n
```

这样便不再需要SIGNAL原语。很显然这个方案比 Hoare 和Brinch Hansen的方案更通用，但它从未被采用过，为什么？（提示：从实现考虑）

14 一个快餐厅有四种职员：（1）领班，他们接收顾客点的菜单（2）厨师，准备饭菜（3）打包工，将饭菜装在袋子里（4）出纳员，收钱并将食品袋交给顾客。每个职员可被看作一个进行通信的串行进程，它们采用的进程间通信方式是什么？将该模型与MINIX中的进程加以比较。

15 假设有一消息传递系统使用信箱，一个进程向满信箱发消息或从空信箱收消息都不会阻塞，进程对错误码的处理方式为重试，直到成功为止，这种方案会带来竞争吗？

16 在图2－20所示的哲学家用餐问题的解法中，为什么过程take_forks将状态变量置为HUNGRY？

17 考虑图2－20中的过程put_forks，假设变量state[i]在对test的两次调用之后被置为THINKING，而不是在调用之前。对于3位哲学家的情况这个改动有什么影响？对于100位哲学家的情况呢？

18 从何种类型的进程可以在何时被启动的角度来看，读者－写者问题可以通过几种方式进行形式化。根据优先哪几类进程的不同，请详细地描述该问题的三种变体。对每种变体，请说明当一个读者或写者能够访问数据库时情况将会怎样，以及当一个进程对数据库访问结束后又将会怎样。

19　CDC6600计算机使用一种有趣的称作处理器共享的时间片调度算法，它可以同时处理多达10个I/O进程，每条指令结束后都进行进程切换，这样进程1执行指令1，进程2执行指令2，等等。进程切换由特殊的硬件完成，所以没有开销。如果在没有竞争的条件下一个进程需要T秒钟完成，那么当有n个进程共享处理器时需要多长时间完成？

20 时间片调度程序通常维护一个有所有就绪进程组成的队列，每个进程在队列中出现一次。如果一个进程在队列中出现两次以上，情况将会怎样？你能设想出这种情况出现的原因吗？

21 对某系统进行监测后表明平均每个进程在I/O阻塞之前的运行时间为T。一次进程切换需要的时间为S，这里S

实际上即为开销。对于采用时间片长度为Q的时间片调度法，对以下各种情况给出CPU利用率的计算公式。

(a) Q=∞

(b) Q>T

(c) S<Q<T

(d) Q=S

(e) Q 趋近于0

22 有5个待运行任务，各自预计运行时间分别是9，6，3，5和X。采用哪种运行次序将使平均响应时间最短？（答案依赖于X）

23 有5个批处理任务A到E几乎同时到达一计算中心。其预计运行时间分别为10， 6， 2， 4和8分钟。其优先级（由外部设定）分别为3，5，2，1和4，这里5为最高优先级。对于下列每种调度算法，计算其平均进程周转时间，进程切换开销可忽略。

(a) 时间片轮转

(b) 优先级调度

(c) 先来先服务（按照次序 10， 6， 2， 4， 8）

(d) 最短作业优先

对（a），假设系统具有多道处理能力，每个作业均获得公平的CPU份额，对（b）到（d）假设一时刻只有一个作业运行，直到结束。所有的作业都是完全的CPU密集型作业。

24 在CTSS上运行的一个进程需要30个时间片方能结束，则它需要被换入多少次？包括第1次，即开始运行之前。

25 使用一个参数 a=1/2的老化算法来预测运行时间。从最早到最近的前4次执行时间分别为40，20，40和15毫秒，则下次运行时间预计为多长？

26 一个软实时系统有4个周期性事件，其周期分别为50，100，300和250毫秒。假设其处理分别需要35，20，10和X毫秒，则该系统可调度所允许的X值最大是多少？

27 解释两级调度为什么被广泛采用？

28 MINIX在执行期间维护一个变量proc_ptr，它指向当前进程的进程表项，为什么？

29 MINIX对消息不加缓冲，请解释这样的设计会对时钟和键盘中断带来什么问题？

30 在MINIX中当一条消息被发送给一个睡眠进程时，将调用过程ready来将该进程挂入适当的调度队列中，该过程首先要关中断，为什么？

31 MINIX中的mini_rec过程包含一个循环，请解释为什么需要该循环。

32 MINIX使用图2－23所示的调度方法，其中不同类型的进程有不同的优先级。优先级最低的进程（用户进程）使用时间片调度法，而系统任务和服务器进程则允许一直运行到阻塞。请问优先级最低的进程是否会发生饥饿，为什么？

33 MINIX适用于实时应用吗？例如数据日志记录。若不适用，可对其作什么改动？

34 设你有一个提供信号量的操作系统，请实现一条消息传递系统，写出发送和接收消息的过程。

35 一个主修人类学、辅修计算机科学的学生参加了一个课题，调查非洲狒狒是否能被教会理解死锁。他找到一处很深的峡谷，在上边固定了一根横跨峡谷的绳索，这样狒狒就可以攀住绳索越过峡谷。同一时刻可以有几只狒狒通过，只要它们朝着相同的方向。但如果向东和向西的狒狒同时攀在绳索上则将发生死锁（狒狒将被卡在中间），因为它们无法在吊在峡谷上时从另一只的背上翻过去。如果一只狒狒想越过峡谷，它必须看当前是否有别的狒狒正在逆向通过。使用信号量写一个避免死锁的程序来解决该问题。

36 继续前一问题，但现在要避免饥饿。当一只想向东去的狒狒到了绳索跟前，但发现有别的狒狒正在向西越过峡谷时，它将一直等到绳索可用为止。但在至少有一只狒狒向东越过峡谷之前，不允许再有狒狒开始从东向西越过峡谷。

37 使用管程，而不是信号量来解决哲学家用餐问题。

38 在MINIX核心中增加一些代码来跟踪从进程（任务）i发送到进程（任务）j的消息个数，按下F4键时打印出该矩阵。

39 修改MINIX的调度程序以跟踪每个用户进程最近使用的CPU时间。当没有任务或服务器进程运行时，要求选择使用CPU最少的用户进程运行。

40 重新设计MINIX，使得每个进程的进程表中有一个优先级域可用来对单个进程设置更高或更低的优先级。

41 修改mpx386.s中的宏hwint_master 和hwint_slave，使得当前由save函数执行的操作改为由在线代码执行。这样作使代码增大了多少？你能测出性能提高了多少吗？

第三章　输入/输出系统

操作系统的主要功能之一是控制所有的输入/输出（Input/Output）设备，它必须向设备发布命令，捕获中断并进行错误处理，它还要提供一个设备与系统其余部分之间的简单易用的界面。该界面应对所有设备尽可能地一致（设备无关性）。I/O部分的代码占据了整个操作系统的相当部分。本章研究操作系统如何管理输入/输出。

本章内容安排如下：首先介绍I/O硬件的基本原理，然后概述I/O软件。I/O软件可以分为若干层次，每一

层都有一个定义良好的界面，我们将讨论其中的每一层，以了解各层的功能以及它们如何结合在一起。

随后讨论死锁。我们将给死锁下一个准确的定义，讲述死锁产生的原因，提供对死锁进行分析的两种模型，并讨论几种死锁预防的算法。

接下来简要地浏览一下MINIX中的I/O。在概述之后我们将详细讨论四种I/O设备 — RAM盘、硬盘、时钟和终端。对每一种设备我们都将研究其硬件、软件以及在MINIX中的实现。在本章的结尾，我们将简短地讨论一段MINIX代码，它位于与I/O任务相同的层次但并不执行I/O任务，它为存储器管理和文件系统提供一些服务，例如从用户进程获取若干数据块。

3.1 I/O硬件原理

不同的人对I/O硬件有不同的理解。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－硬件能够接收的命令、它能够完成的功能、以及能报告的各种错误等。本书主要立足于对I/O设备的编程，而并非对硬件的设计、制造和维护，因此我们的重点放在如何对硬件编程，而不是其内部的工作原理。然而，对许多I/O设备编程常常不可避免地牵涉到其内部操作。在以下三节中我们将介绍与I/O设备编程有关的硬件知识。

3.1.1 I/O设备

I/O设备可粗略地分为两类：块设备（block devices）和字符设备（character devices）。块设备将信息存储在可寻址的固定大小的数据块中，通常数据块大小的范围从512字节到32768字节不等。块设备的主要特征是能够独立地读写单个的数据块。磁盘是最常见的块设备。

从严格意义来说，按块编址的设备和无法按块编址的设备之间没有明显的界限。磁盘是公认的按块编址的设备，因为无论当前磁头处于什么位置，总可以定位到其他柱面并等待所需要的数据块旋转到磁头下面。现在设想用一台8mm或DAT磁带机作磁盘备份。磁带上也是固定大小的数据块。如果命令磁带机读第n个数据块，则通过先倒带再前进总能定位到所需要的数据块。该操作可比作磁盘的寻址，只不过更费时罢了。同样，重写磁带中间的某个数据块可能作得到，也可能作不到。即使将磁带用作为随机存取的块设备是可能的，但是通常并不这样使用它们。

另一类是字符设备。与块结构相比，一个字符设备可以发送或接收一个字符流。字符设备无法编址，也不存在任何寻址操作。打印机、网络接口、鼠标（用作指点设备）、老鼠（用于心理学实验）以及多数与磁盘不同的设备均可被视作字符设备。

这种分类方法并不是最好的，有些设备就不适用于这种分类法。例如，时钟是无法寻址的，同时它也不产生或接收字符流。时钟的全部功能就是按照预先定义的时间间隔发出中断。存储映像显示器也不适用该分类模型。但总的来说，使用这种方法可以将控制不同I/O设备的操作系统软件成分隔离开来。例如文件系统仅仅控制抽象的块设备，而将与设备有关的部分留给低层软件，即设备驱动程序（device driver）去处理。

3.1.2 设备控制器

I/O设备通常包含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。电子部分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展槽的印刷电路板。机械部分则是设备本身。

控制器卡上一般都有一个接线器，可以将与设备相连的电缆线接进来。许多控制器可以控制2个、4个甚至8个相同设备。如果控制器和设备之间的接口采用标准接口，如ANSI、IEEE、ISO，或者事实上的标准，则多家厂商都可以制造与该接口匹配的控制器和设备。例如许多厂商生产IDE（集成设备电子器件）接口或SCSI（小型计算机系统接口）接口的硬盘驱动器。

之所以区分控制器和设备本身是因为因为操作系统大多与控制器打交道，而非设备本身。大多数小型计算机的CPU和控制器之间的通信采用图3-1所示的单总线模型。大型主机则采用其他模型，常常是多条总线以及专门用于I/O的计算机，这种专用计算机被称为I/O通道，它能够减轻主CPU的部分工作负担。

图 3-1 一个连接CPU、存储器、控制器和I/O设备的模型。

控制器与设备之间的接口通常是一种很低层次的接口。例如一个磁盘，可能被格式化为每道16个512字节的扇区。实际从驱动器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，最后是一个检查和（亦称作纠错码ECC）。其中的前缀是磁盘格式化时写进磁盘的，里面包含有柱面数和扇区数、扇区大小之类的数据，以及同步信息。

控制器的任务是将这个串行的比特流转换成字节块并在需要时进行纠错。通常该字节块是在控制器中的一个缓冲区中逐个比特汇集而成。在对检查和进行校验证实数据正确之后，该块数据随后被拷贝到主存中。

在同样低的层次上，CRT终端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，最后将显示结果打在屏幕上。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折返信号。如果没有CRT控制器，则操作系统程序员只能自己编写程序来解决此问题。有了CRT控制器后，操作系统只需通过几个参数对控制器进行初始化，输入参数，诸如每行的字符数和每屏的行数，之后控制器将自行控制扫描波束。

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空间的一部分，

这种方案称作内存映像I/O。68OXO系列计算机就采用该方案。其他计算机则使用I/O专用的地址，每个控制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。有些生产IBM PC兼容机的厂商采用与IBM不同的I/O编址方案。除I/O端口外，许多控制器还使用中断来通知CPU它们已作好准备，使其寄存器可以读写。中断首先是一个电信号事件。硬件的中断请求信号线提供了中断控制器芯片的物理输入。这种输入线的数量是有限的：奔腾系列的PC机向I/O设备提供15条可用中断。有些控制器直接制作在主板上，如IBM PC机的键盘控制器。对于那些单独插在主板上的控制器，有时它上面设有一些可用来设置IRQ号的开关或跳线器，以便避免IRQ冲突（对于具有即插即用功能的板子，IRQ号可由软件设定）。中断控制器芯片将每个IRQ输入映射到一个中断向量，通过该中断向量便可以找到相应的中断服务程序。图3-2给出了IBM PC机部分控制器的I/O地址、硬件中断和中断向量号。MINIX采用相同的硬件结构，但其中断向量则与此处给出的MS-DOS的中断向量不同。

图 3-2　一台典型的运行MS-DOS的PC机中若干控制器、I/O地址、硬件中断线和中断向量示例

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IBM PC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。其中许多命令带有参数，这些参数也要装入控制器的寄存器中。某设备控制器接收到一条命令后，CPU可以转向其他工作，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。CPU仍旧是通过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。

3.1.3　存储器直接存取DMA

许多设备，尤其是块设备都支持直接存储器访问，或称DMA（direct memory access）。我们先看一下没有使用DMA时磁盘是怎么读数据的。首先控制器逐个比特地从设备完整地读出一块（一个或多个扇区）数据放入内部缓冲区中。然后计算该块数据的检查和以保证读取正确。接着控制器发出中断信号，操作系统开始逐个字节（或字）地从控制器的设备寄存器中将数据读入内存。由于一次只能读一个字节（或字），所以需要通过一个循环才能将整个数据块读完。

显然，CPU循环地每次从控制器读一个或几个字节是很费时的，为了解决这个问题，人们引入DMA将CPU从这项工作中解脱出来。使用DMA时，CPU除了告诉控制器数据块的地址外，只需告诉控制器两条信息：数据块将存放在内存的地址和要传输的字节数，如图3-3所示。

图 3-3　一次DMA传输完全由控制器完成。

控制器首先从设备中将整个数据块读入内部缓冲区并进行校验，接着它将第一个字节（或字）拷贝至DMA内存地址处，随后它对DMA地址和DMA计数分别增减刚刚传送的字节数。这个过程一直重复下去直到DMA计数变成0。此时设备控制器发出中断信号，通知操作系统数据已读取完毕，在这里操作系统无需再将数据拷贝到主存中，因为数据已经在读的过程中放在那里了。

可能有人会有疑问：为什么控制器从设备读到数据后不立即将其送入内存，而是需要一个内部缓冲区呢？原因是一旦磁盘启动开始读数据，从磁盘读出比特流的速率是恒定的，不论控制器是否作好接收这些比特的准备。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系统总线的控制权。如果由于其他设备的争用而导致总线忙，则只能等待。如果在上一个字还未送入内存之前下一个字到达，控制器只得找另一个地方把它暂存起来。如果总线非常忙，则控制器可能需要大量的信息暂存，而且还要作大量的管理工作。从另一方面来看，如果采用内部缓冲区，则在DMA操作启动之前不需要使用总线，这样控制器就能够设计得很简单，因为DMA到主存的传输对时间要求并不严格（有些旧式控制器的确直接访问主存，而且内部缓冲区设计得很小，但当总线忙碌时，一次传输有可能被迫终止）。

上述两个步骤的缓冲过程对I/O性能有重要影响。当数据被CPU或控制器从控制器传送到内存时，下一个扇区正好通过磁头下边，同时扇区上存储的信息正在送入控制器。简单的控制器不具有同时执行输入和输出的功能，因此当进行内存传送时，当前通过磁头下的那个扇区的数据将丢失掉。

这样的结果是控制器只能隔一个块读取一个数据块 ，为了读完一条完整的磁道，磁盘需要旋转两周，一周读偶数块，一周读奇数块。如果数据通过总线从控制器传到内存所用的时间比从磁盘读数据到控制器的时间长，则有可能读一块数据需跳过两块（或更多）。

有意地跳过一些块以便为控制器腾出时间供其将数据传送到内存的技术称为交叉编址。磁盘在格式化时，数据块编号就考虑到了交叉系数。在图3-4 (a)中，我们看到一个没有交叉编址的8个块的磁盘编址方案。在图3-4 (b)中，我们看到采用单交叉编址的方案。在图3-4 (c)中则采用双交叉编址方案。

图 3-4　(a)不采用交叉编址 (b)单交叉编址 (c)双交叉编址

这种磁盘块编址方案的意图是：使操作系统能够连贯地读出数据块，并同时达到硬件许可的最大速度。如果按图3-4 (a)进行编址，而控制器只能作到隔一个块读取数据的速度，则操作系统为了保证将8个数据块按照从0到7的顺序读出，磁盘就需要旋转8周（当然，若操作系统知道这个问题，则它可以用软件方法来解决，但最好由控制器来解决交叉编址的问题）。

并非所有的计算机都使用DMA。反对意见认为主CPU比DMA控制器快得多，完全可以更快地完成这件工作（当I/O设备的速度不构成瓶颈时）。如果CPU无事可作，而又被迫等待慢速的DMA 控制器，这是完全没有意义的。同时，省去DMA控制器，由CPU完成所有这些工作也能够节省一些成本。

3.2　I/O软件原理

现在来看I/O软件。I/O软件的总体目标很明确。其基本思想是，将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，高层软件则主要是为用户提供一个简洁、规范的界面。随后几节我们将了解这些目标是如何实现的。

3.2.1 I/O软件的目标

I/O软件设计的一个关键概念是没备无关性。其含义就是使程序员写出的软件无需修改便能读出软盘、硬盘以及CD-ROM等不同设备上的文件。用户可以简单地输入如下命令：

```
sort < input > output
```

就能够从各种设备上获得输入，包括软盘、硬盘、或者键盘，同时可以将输出送到各种不同的设备上，如软盘、硬盘、甚至是屏幕。不同设备之间的差异将由操作系统来处理，操作系统会调用不同的设备驱动程序来真正地将数据写到输出设备上。

与设备无关性紧密相关的是统一命名法。一个文件或设备名将简单地只是一个字符串或一个整数，而完全不依赖于设备。在UNIX中，所有的磁盘可以以任何方式集成到文件系统层次结构中去，用户也不必知道哪个各字对应着哪个设备。例如，软盘可以被安装到目录/usr/ast/backup下，这时拷贝一个文件至/usr/ast/backup/monday，将把文件拷到软盘上。使用这种方式，所有的文件和设备的定位都统一通过路径名来实现。

I/O软件的另一个重要方面是错误处理。总的来说，错误应在尽可能接近硬件的地方处理。如果控制器发现一个读错误，它应该尽量进行处理，如果控制器处理不了，则交给设备驱动程序，可能只需重读一次就可以解决问题。许多错误是暂时性的，如磁头被灰尘阻滞导致的读错误，只需重复一次操作便可以消除。只有在低层软件处理不了的情况下才通知高层软件。在许多情况下，低层软件可以自行处理错误而不被高层软件感知。

同样重要的另一个问题是同步（阻塞）－异步（中断驱动）传输。多数物理I/O是异步传输－CPU在启动传输操作后便转向其他工作，直到中断到达。如果I/O操作采用阻塞语义，那么用户编程将简单许多－发出一条READ命令后，程序将自动被挂起，直到数据被读到缓冲区中。使实际上是中断驱动的操作对用户程序具有阻塞语义则属于操作系统的责任。

最后一个概念是专用设备和共享设备。某些设备可同时被多个用户使用，如磁盘，多个用户同时打开多个文件是毫无问题的。另一些设备则在某一时刻只能供一个用户专用，只有当前用户使用完毕，其他用户才能继续使用，如磁带机。两个或多个用户同时向一台磁带机写数据是肯定不行的。专用（非共享）设备带来了许多问题。相应地，操作系统必须能够同时处理这两种设备。

通过将I/O软件组织成以下四个层次，操作系统可以合理、高效地实现以上目标。

1中断处理程序（底层）

2设备驱动程序

3与设备无关的操作系统软件

4用户层软件（高层）

这四个层次与图2-26中所示的四个层次完全对应（这并非偶然）。在随后的章节中我们将自底向上依次进行讨论。本章中我们的重点放在设备驱动程序（第2层），但同时也将涉及其他层次以了解各层之间如何协同工作。

3.2.2 中断处理程序

中断是应该尽量加以屏蔽的一个概念，应该将其放在操作系统的底层进行处理，以便其余部分尽可能少地与之发生联系。屏蔽中断的最好方法是将每一个进行I/O操 作的进程挂起，直至I/O操作结束并发生中断。进程自己阻塞的方法有：执行信号量的DOWN操作、条件变量的WAIT操作；或接收一条消息等等。

当中断发生时，中断处理程序执行相应的操作，以解除相应进程的阻塞状态。在一些系统中是执行信号量的UP操作。在管程中可能是对一个条件变量执行SIGNAL操作，另外一些系统则可能是向阻塞的进程发一条消息，总之其作用是将刚才被阻塞的进程恢复执行。

3.2.3 设备驱动程序

设备驱动程序中包括了所有与设备相关的代码。每个设备驱动程序只处理一种设备，或者一类紧密相关的设备。例如，若系统所支持的不同品牌的所有终端只有很细微的差别，则较好的办法是为所有这些终端提供一个终端驱动程序。另一方面，一个机械式的硬拷贝终端和一个带鼠标的智能化图形终端差别太大，于是只能使用不同的驱动程序。

在本章的前半部分我们了解了设备控制器的功能，知道每个控制器都有一个或多个寄存器来接收命令。设备驱动程序发出这些命令并对其进行检查，因此操作系统中只有硬盘驱动程序才知道磁盘控制器有多少个寄存器，以及它们的用途。驱动程序知道使磁盘正确操作所需要的全部参数，包括扇区、磁道、柱面、磁头、磁头臂的移动、交叉系数、步进电机、磁头定位时间等等。

笼统地说，设备驱动程序的功能是从与设备无关的软件中接收抽象的请求，并执行之。一条典型的请求是读第n块。如果请求到来时驱动程序空闲，则它立即执行该请求。但如果它正在处理另一条请求，则它将该请求挂在一个等待队列中。

执行一条I/O请求的第一步，是将它转换为更具体的形式。例如对磁盘驱动程序，它包含：计算出所请求块的物理地址、检查驱动器电机是否在运转、检测磁头臂是否定位在正确的柱面等等。简言之，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，则驱动程序将数据报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。

3.2.4 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。

图3-5中罗列的功能一般都是由设备无关软件完成的。在MINIX中，多数设备无关软件属于文件系统，位于图2-26的第三层。虽然文件系统属于第5章的内容，这里我们仍需要简单地介绍一下以了解I/O的一些特性。

图 3-5 设备无关I/O软件的功能。

设务无关软件的基本功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。

与命名相关的是保护。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，有些字符设备对字节进行操作(Modem)，另一些则使用比字节大一些的单元（网卡），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，但定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。若设备不可用，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。

3.2.5 用户空间的I/O软件

尽管大部分I/O软件属于操作系统，但是有一小部分是与用户程序链接在一起的库例程，甚至是在核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。但也有一些库例称，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精灵进程（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。而由该精灵进程进行打印，这里只有该精灵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精灵进程，发送文件前先将其放在一特定目录下，而后由网络精灵进程将其取出发送。这种文件传送方式的用途之一是Internet电子邮件系统。Internt通过许多网络将大量的计算机联在一起。当向某人发送E-mail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E-mail系统在操作系统之外运行。

图3-6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 3-6　I/O系统的层次结构及各层的主要功能。

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。

3.3　死锁

计算机系统中有许多独占资源，它们在任一时刻都只能为一个进程使用，如平板式绘图仪、CD-ROM驱动器、CD-ROM记录器、8mm DAT磁带机、照相制版机、以及进程表项等。两个进程同时向打印机输出将导致一片混乱，两个进程同时使用同一进程表项则可能导致系统崩溃。正因为如此，所有的操作系统都具有授权一个进程独立地访问某一资源的能力。

在许多应用中，一个进程需要独占地访问不止一种资源，例如一家公司的业务是在1米宽的平板式绘图仪上绘制精确的美国地图，地图信息从CD-ROM中读出。假设进程A申请到了CD-ROM，稍后进程B申请到了绘图仪，现在进程A申请绘图仪导致阻塞等待,进程B申请CD－ROM也导致阻塞。至此两个进程都被阻塞而且将永远不能解除。这种状态就是死锁，死锁应该尽力加以避免。

除了申请独占设备外，其他情况也可能导致死锁。例如在一个数据库系统中，为了避免竞争，可能将若干记录加锁。若进程A对记录R1加锁，进程B对记录R2加锁，随后两进程又各自试图将对方的记录加锁，这时也将导致死锁。因此，软、硬件资源都有可能导致死锁。

本节将研究死锁如何产生，以及如何加以预防和避免。为方便起见，将以申请物理设备为例，例如，磁带机、CD－ROM和绘图仪，但有关原理和算法同样适用于其余类型的死锁。

3.3.1　资源

进程对设备、文件等获得独占性的访问权时有可能会发生死锁，为了尽可能地通用化，我们将这种需排它使用的对象称为资源。资源可以是硬件设备（如磁带机），或一组信息（如数据库中一个加锁的记录）。计算机中通常有多个资源。有些类型的资源有多个相同的实例，如三台磁带机。当某一资源有若干拷贝时，其中任一个均可用来满足对资源的请求。简言之，资源是在任何时刻只能被单个进程使用的任何对象。

资源有两类：可剥夺式和不可剥夺式。可剥夺式资源可从拥有它的进程处剥夺而没有任何副作用，存储器是一类可剥夺资源。例如，一个系统有512K用户内存和一台打印机。若两个512K内存的进程都要进行打印，进程A申请并得到了打印机，然后开始计算要打印的值。在它未完成计算任务之前，它的时间片用完并被换出。然后进程B开始运行并申请打印机，但未成功。此时有潜在的死锁危险。因为进程A拥有打印机，而进程B占有内存。然而幸运的是通过将进程B换出、进程A换人便可以实现剥夺进程B的内存。于是，进程A继续运行并执行打印任务，然后释放打印机。这个过程将不会发生死锁。

相反地，不可剥夺资源无法在不导致相关计算失败的情况下将其从属主进程处剥夺。若一个进程已开始打印，那么剥夺其占用的打印机并分配给另一进程将使打印结果一片混乱。打印机是不可剥夺的。

总的来说，死锁与不可剥夺资源有关，涉及可剥夺资源的潜在的死锁通过在进程间重新分配资源通常能够化解。所以，我们的重点放在不可剥夺资源。

使用一个资源的事件顺序是：

1 申请资源

2 使用资源

3 释放资源

若申请时资源不可用，则申请进程被迫等待。在一些操作系统中，申请失败的进程自动被阻塞，在资源可用时再被唤醒。在其他系统中，资源申请失败将返回一个错误码，由申请进程等待一段时间后重试。

3.3.2 死锁原理

死锁的定义是：

若一个进程集合中的每一个进程都在等待只能由本集合中的另一个进程才能引发的事件，则这种情况被视为死锁。

由于所有的进程都在等待，所以没有一个进程能够触发那个（些）能够唤醒本集合中另一个进程的事件，于是所有进程都将永远地等待下去。

多数情况下，进程是在等待本集合中另一个进程释放的资源。换言之，每个进程都在等待另一个进程所占有的资源。但因为所有进程都无法运行，因而无法释放资源，于是所有进程都不能被唤醒。至于进程数和占用及申请的资源数并不重要。

死锁的条件

Coffman 等人（1971）总结出了死锁发生的四个条件：

1 互斥条件，每一资源或者被分配给一个进程，或者空闲。

2 保持和等待条件，已分配到了一些资源的进程可以申请新的资源。

3 非剥夺条件，已分配给一进程的资源不可被剥夺，只能被占有它的进程显式地释放。

4 循环等待条件，系统必然有一条由两个或两个以上的进程组成的循环链，链中的每一个进程都在等待相邻进程占用的资源。

以上四个条件是死锁发生的必要条件，只要一条或多条不成立，死锁就不会发生。

死锁模型

Holt（1972）提出了用有向图可以建立以上四个条件的模型。图中有两类节点：圆形表示进程，方形表示资源。从资源节点到进程节点的弧表示该资源已被进程占用。在图 3－7(a)中，资源R当前被进程A占用。

图 3-7 资源分配图。 (a)占用一个资源 (b)请求一个资源 (c)死锁

从进程到资源的弧表示进程当前正申请该资源并处于阻塞等待状态。在图3－7(b)中，进程B正在等待资源S。图3－7(c)示出了死锁状态：进程C等待资源T，资源T被进程D占用，进程D又在等待被进程C占用的资源V。于是两个进程都将等待下去。图中的环表示涉及到这些进程和资源的死锁。本例中的环为C－T－D－V－C。

再看资源图的用法。假设有三个进程A，B，C及三个资源R，S，T。三个进程对资源的申请和释放如图3－8(a)－(c)所示。操作系统可以选择任何一个非阻塞的进程来执行，因此它可以选择A执行直至结束，然后运行B，最后运行C。

图 3－8 一个死锁如何发生和避免的例子。

这个顺序不会导致死锁（因为不存在资源的竞争）。当进程串行执行时，不可能在一个进程等待I/O时，另一个进程占用CPU进行计算。所以严格的串行操作可能不是最优的。另一方面，如果进程完全不执行I/O操作，则最短作业优先调度将优于时间片调度，所以在某些情况下，串行执行可能是最优的。

考虑进程操作包含I/O和计算，则时间片法是一种合理的调度算法。资源申请顺序可能如图3－6（d）所示。若按此顺序执行，则相应的资源图示于图3－8（e）-（j）。在发生请求（4）之后，进程A阻塞等待S，如图3－8（h）所示，随后的两步中B和C也将阻塞，最终导致产生循环和死锁，图3－8(j)中所示。

然而如前所述，操作系统不必局限于按某种特定次序来执行这些进程。对于一个可能导致死锁的资源请求，操作系统可以简单地不批准该请求，并由此将进程挂起（即不予调度）直到安全状态。在图3－8，设操作系统知道可能导致死锁，它可以不把资源S分配给B，于是B被挂起。假设只运行过程A和C，则资源申请和释放过程如图3－8(k)，而非3-8(d)。该过程的资源图示于图 3－8(l)-(q)，其中无死锁发生。

第(q)步执行完后，可将资源S分配给B，因为A已经结束，C已得到它所需要的所有资源，即使B最终因申请T而等待也不会发生死锁，B只是需等C结束而已。

本章后边我们将研究一个详细的算法，以达到对资源进行分配而不会死锁。此处需要明确的是资源图可用作分析一给定的申请/释放序列是否将导致死锁的一种工具。我们只需按申请和释放的顺序一步步地走下来，最后检查其中是否包含有环路。如果有，则发生死锁；反之则无。虽然我们的例子中只涉及一类资源仅包含一个实例，但以上原则完全可以推广到同种资源含有多个实例的情况。

概括而言，处理死锁有四种策略：

1忽略该问题。

2检测死锁并恢复

3谨慎地对资源进行动态分配，避免死锁

4通过破坏上述四个必要条件来预防死锁发生

随后四节将依次讨论这四种方法。

3.3.3 鸵鸟算法

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助，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机的时间后重试。

现假设一个UNIX系统的进程表有100项，有10个进程在执行，每一个都要创造12个子进程。在每个进程创建9个子进程后，进程表项被全部用完。则这10个进程将进入一个无休止的循环：执行FORK，失败，等待一段时间后执行FORK，再失败 ……这实际上是死锁。发生这类事件的概率是很小的，但它的确存在！我们难道会为了消除这种状况就放弃进程、FORK这些概念和方法吗？

打开文件的最大数目受i-节点表大小的限制，所以当i-节点表满时会发生类似的问题。磁盘上的对换空间也是另一种有限的资源。实际上，几乎操作系统中的每一种表格都代表了一种有限的资源。难道我们会因为可能出现类似的死锁而抛掉这一切吗？

UNIX处理这一问题的办法是忽略它，因为大多数用户宁可在极偶然的情况下发生死锁，也不愿限制每个用户只能创建一个进程、只能打开一个文件等等。如果死锁可以不花什么代价就能够解决，则什么问题都没有了。问题是，这种代价很大，而且常常给用户带来许多不便的限制。于是我们不得不在方便性和正确性之间作出折衷。

3.3.4 死锁检测和恢复

第二种技术是死锁检测和恢复，采用这种技术时，系统只需监视资源的申请和释放。每次资源被申请或释放时，资源图随之刷新，同时检测是否存在环路。如果存在一个环，则环中的一个进程被撤销，如果仍不能破除死锁，则撤销另一个进程。如此往复直至环路被破除。

一种更简陋的方法甚至不维护资源图，而是周期性地检测进程是否连续阻塞超过一定时间，如一小时。一旦发现这样的进程则将其撤销。

这种策略常被用于大型主机，尤其是批处理系统。因为在批处理系统中，取消一个进程然后重新启动它通常是可以接受的。但必须注意要将该进程修改了的文件恢复到初始状态，并消除掉它所导致的所有副作用。

3.3.5 死锁预防

第三种策略是对进程施加适当的限制以从根本上消除死锁。Coffman等人（1971）提出的四个条件为解决该问题提供了线索。如果能够保证四个条件中至少有一个不成立，则死锁将不会发生（Havender，1968）。

先考虑互斥使用条件。如果资源不被单一进程独占，则死锁肯定不会发生。然而打印机这样的设备不允许两个进程同时使用，通过借助Spooling技术可以允许多个进程同时产生打印数据。该模型中，唯一真正申请物理打印机的进程是打印精灵进程，由于它决不会申请别的设备，所以不会因打印机而发生死锁。

但并非Spooling技术适用于所有的设备（如进程表）。而且在Spooling时对磁盘空间本身的竞争可能导致死锁。例如两个进程分别占用了可用的Spooling磁盘空间的一半而等待更多的空间，则必将导致死锁。具体地说，如果打印精灵进程被设计为不等全部输出放入Spooling目录下就开始打印，则它可能在打印完第一批数据后空闲几个小时等待剩余数据就位。为了避免这种现象，它一般被设计成等完整的输出文件就绪后才开始打印，这时便可能发生前述的情况，即系统将死锁。

第二个条件似乎更容易解决。只要禁止已拥有资源的进程再申请其他资源便可以消除死锁。一种实现方法是规定所有进程在开始执行前申请所需的全部资源。如果所需的资源全部可用则进行分配，于是它肯定能够运行结束。如果有一个或多个资源正被使用，则不进行分配，进程阻塞。

一个直接的问题是许多进程直到运行时才知道它需要多少资源。另一个问题是这种方案的资源利用率无法优化。例如一个进程先从输入磁带上读取数据，进行一小时的分析，最后将输出写到一条输出磁带上，同时将其在绘图仪上绘出。如果所有资源被提前申请，则有一小时时间输出磁带机和绘图仪无法使用。

另一种方案是当进程申请资源时先暂时释放其当前占用的所有资源，只有对该资源申请成功才收回其原先占用的资源。

消除第三个条件（不可剥夺）比第二个困难。若一个进程已分配到一台打印机正在进行打印，如果因为它需要的绘图仪无法得到而强制性地将打印机剥夺掉，这将导致一片混乱。

消除最后一个条件－循环等待有几种方法。一种是保证每一个进程在任何时刻只能占用一个资源，如果要申请第二个，它必须先释放第一个。若进程正在将一个大文件从磁带机上读出并送到打印机打印，则这种限制是不可接受的。

另一种方法是将所有资源赋予一个全局编号，如图3－9(a)所示。进程申请资源必须按照编号的顺序。进程可以先申请打印机，后申请磁带机，但不可以先申请绘图仪，后申请打印机。

图 3－9 （a）资源被赋予数字序号。（b）一个资源图。

按此规则，资源分配图中肯定不会出现环路。例如图3－9(b)所示的两个进程的例子，只有在A申请资源j，B申请资源i的情况下会发生死锁。设i和j是不同资源，它们将具有不同的数值。若i>j，则A不允许申请j；若i<j，则B不允许申请i。不论哪种情况都不可能发生死锁。

对于多进程的情况，以上逻辑依然成立。对于每一种情况，总有一个被分配的资源是编号最高的。占用该资源的进程不可能申请其他已被占用的各种资源，它或者将执行完毕，或者申请编号更高的资源，而编号更高的资源肯定是可获得的。最终它将结束并释放所有资源，这时另一个占有最高编号资源的进程也可以执行完。依次类推，所有进程都可以执行完毕，因而没有死锁发生。

该算法的一个变种是摈弃必须按升序申请资源的限制，而仅要求不允许进程申请编号比当前所占有资源编号低的资源。若一个进程起初申请9号和10号资源，随后将其释放，它实际上相当于从头开始，所以没有必要阻止它现在申请1号资源。

尽管对资源编号的方法消除了死锁的问题，但几乎找不出一种使每个人都满意的编号次序。当资源包括进程表项、Spooling磁盘空间、加锁的数据库记录及其他抽象资源时，潜在的资源及各种不同用途的数目会变得如此之大，以至于使编号法根本无法使用。

死锁预防的各种方法如图3－10所示。

图 3－10 死锁预防途径小结。

3.3.6 死锁避免

从图3-8中，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。

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算法，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3-11（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最大贷款额，所以他只保留10个单位的资金来为客户服务，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 3－11 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3-11（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止）。图3-11（b）所示的状态是安全的，因为在有2单位资金可用的情况下，银行家可以延迟除Marvin之外的所有请求，这样便可以使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Snganne或者Barbarn的请求，等等。

考虑假如给Barbara另一个她申请的资源，如图3-11（b），则我们得到如图3-11（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。

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3-12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 3－12 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 I2和I5 I6组成的矩形区域，那么最后一定会到达I2和I6的交叉点，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。

多种资源的银行家算法

以上资源轨迹图的方法很被难被扩充到系统中有任意数目的进程、任意种类的资源，并且每种资源有多个实例的情况。但银行家算法可以被推广用来处理这个问题。图3-13示出了其工作原理。

图 3－13 多种资源的银行家算法。

在图3-13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。

最右边的三个向量分别表示总的资源E、已分配资源P，和剩余资源A。由E可知系统中共有6台磁带机，3台绘图仪，4台打印机和2台CD-ROM。由P可知当前已分配了5台磁盘机，3台绘图仪，2台打印机和2台CD-ROM。该向量可通过将左边矩阵的各列相加得到，剩余资源向量可通过从资源总数中减去已分配资源数得到。

检查一个状态是否安全的步骤如下：

1查找右边矩阵中是否有一行，其未被满足的设备数均小于或等于向量A。如果找不到，则系统将死锁，因为任何进程都无法运行结束。

2若找到这样一行，则可以假设它获得所需的资源并运行结束，将该进程标记为结束，并将资源加到向量A上。

3重复以上两步，直到所有的进程都标记为结束。若达到所有进程结束，则状态是安全的，否则将发生死锁。

如果在第1步中同时存在若干进程均符合条件，则不管挑选哪一个运行都没有关系，因为可用资源或者将增多，或者在最坏情况下保持不变。

图3－13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（1 0 0 0），这时将导致死锁。则显然E的请求不能立即满足，必须延迟一段时间。

该算法最早由Dijkstra于1965年发表。从那之后几乎每本操作系统的专著都详细地描述它，许多论文的内容也围绕该算法，但很少有作者指出该算法缺乏实用价值。因为很少有进程能够在运行前就知道其所需资源的最大值，而且进程数不是固定的，往往在不断地变化，况且原本可用的资源也可能突然间变成不可用（如磁带机可能会坏掉）。

总之，死锁预防的方案过于严格，死锁避免的算法又需要无法得到的信息。如果你能想到一种理论上和实际中都适用的通用解法，那么就可以在计算机科学的杂志上发表一篇论文。

对于特殊的应用有许多很好的算法。例如在许多数据库系统中，常常需要将若干记录上锁然后进行更新。当有多个进程同时运行时，有可能发生死锁。

常用的一种解法是两阶段上锁法。第一阶段，进程试图将其所需的全部记录加锁，一次锁一个记录。若成功，则数据进行更新并解锁。若有些记录已被上锁，则它将已上锁的记录解锁并重新开始执行，该解法有点类似提前申请全部资源的方法。

但这种方法不通用，在实时系统和过程控制系统中不能够因为资源不可用而将进程中途终止并重新执行。同样若一个进程已进行过网络消息的读写、更新文件、或其他不宜重复的操作，则将进程重新从头执行是不可接受的。该算法仅适用于那些在第一阶段可以随时停止并重新执行的程序，遗憾的是并非所有的应用都可以按这种方式组织。

3.4 MINIX I/O系统概述

MINIX的I/O结构如图3－6所示。其中，上边四层对应于图2－26中所示的MINIX的四层结构。以下章节中我们将对每一层作大概的介绍重点放在驱动程序。中断处理已在第二章作过介绍，设备无关的I/O将在第五章文件系统中讨论。

3.4.1 MINIX的中断处理程序

许多设备驱动程序在启动一些I/O设备后阻塞，等待某种消息到达。这种消息往往由该设备的中断处理程序产生。另外一些设备驱动程序不启动物理的I/O操作（如从RAM盘中读数据，然后写到一个存储器映像的显示器），也不使用中断，所以也不等待从I/O设备发来的消息。前一章中已详细讨论了中断产生消息以及造成任务切换的机制，我们在这里不再赘述。中断处理程序除了产生一条消息之外，还进行一些最底层的I/O处理工作。我们将在此概要地加以讨论，细节将放在讨论具体设备时叙述。

对于磁盘设备，输入输出通常仅仅是命令设备执行某操作，然后等待该操作结束。大部分工作由磁盘控制器完成，而中断处理程序作的工作很少，硬盘的整个中断处理程序只有3行代码，所作的I/O操作仅仅是从控制器读一个字节从而确定控制器的工作状态。如果所有的中断处理都这么简单，那我们的工作就容易得多了。

然而有时低层中断处理程序有更多的事情可做。消息传递的代价是较高的，所以对于中断本身很频繁、而每次中断所处理的I/O操作量又很少的情况，更好的办法是让中断处理程序作更多的工作，而将向对应的任务例程发送消息推迟到后面某一个中断到来之时，即使相应驱动器任务会有较多工作可做。MINIX的时钟就采用这种方法。并非每一个时钟滴答都有许多事情要作，许多时候除了维护系统时间外几乎无事可做。这种情况下，无需在每次时钟中断时都向时钟任务（系统进程）发送消息。时钟中断处理程序每次递增一个变量pending_ticks，当前时间就是时钟任务上次运行时记录的值加上pending_ticks所得的和。当时钟任务接收到一条消息被唤醒时，它将pending_ticks的值加到它所维护的时钟变量上，并将pending_ticks清零。时钟中断处理程序检查几

个变量，当发现时钟任务确实有工作可干时，例如需要发送时钟闹钟信号或进行进程调度，才向其发送消息，。它也有可能向终端任务发送消息。

终端任务采用另一种稍有不同的方案。它要处理几种不同的硬件，包括键盘和RS－232串口线。这些设备每个都有其自己的中断处理程序。键盘是典型的每次中断I/O操作很少的设备。在PC机上当一个键被按下或释放时发生一个中断，若不考虑同时按下SHIFT和CTRL之类的物殊键，可以认为平均每次中断接收到半个字符。因为终端任务无法对半个字符进行处理，所以必须等相应的信息完全获得之后才能向它发送一条消息。我们将在本章较后部分对此进行详细说明，这里我们只提及键盘中断处理过程完成从键盘读数据的操作并将可以忽略的事件滤掉，比如一个普通键的释放（但诸如SHIFT这样的特殊键是不能被忽略的）。所收到的键码被放在一个队列中供终端任务随后进行处理。

键盘中断处理例程与前述的中断处理例程模型不完全一致，它并不向相关任务发送消息，而是将键码放在一个队列中，然后修改tty_timeout变量，而时钟中断处理程序要读这个变量。当中断并不改变键码队列时，tty_timeout也不变。在下一个时钟滴答，若键码队列已发生变化，则时钟中断处理程序将向终端任务发一条消息。其他终端类型的中断处理程序，例如RS－232串口线等，也采用相同的工作方式。在收到一个字符后终端任务很快就会收到一条消息，但当字符到达速率很快时没有必要为每个字符产生一条消息，可以将若干字符累积起来放在一条消息中处理。而且当终端任务收到一条消息时，它将检查所有的终端设备。

3.4.2 MINIX的设备驱动程序

MINIX中的每一类设备都有一个单独的I/O处理任务（设备驱动程序）。这些驱动程序是完整的进程，每个都有其自己的状态、寄存器、堆栈等。设备驱动程序相互之间进行通信，同时也与文件系统进行通信。这些通信采用与MINIX进程间通信完全一样的消息传递机制。每一个设备驱动程序放在一个单独的源文件里，如时钟驱动程序放在clock.c中，其他如RAM盘、硬盘、软盘等都各自有一个驱动程序源文件，其间的公用例程则放在driver.c 中。这在某种意义上讲是将图3-6中的设备驱动程序层又分为两个子层。这种将与设备相关部分和与设备无关部分分开的方法能够更灵活地处理不同类型的硬件配置。尽管不同类型的磁盘驱动程序共用了一部分代码，但它们各自作为独立的进程运行。

终端驱动程序代码的组织方法与此类似，与硬件无关的代码放在tty.c中，而支持各种设备的代码则分别放在各自独占的文件中，包括内存映像的控制台、键盘、串口线以及虚拟终端等，但对终端而言，是一个单独的进程支持所有这些设备。

对于磁盘和终端这样的设备组，不仅有源文件，同时还有头文件，例如driver.h支持所有的块设备驱动程序，tty.h则支持所有类型的终端设备。

设备驱动程序与其他进程的不同在于设备驱动程序全部被链入核心，所以它们都共享一个公用的地址空间。这样一来，若几个设备驱动程序共用一个过程，则在执行代码中只存在该过程的一个拷贝。

这种设计方法高度地模块化，并保持了较高的效率，这也是与LINUX的几处本质区别之一。MINIX中进程通过向文件系统进程发送消息来读一个文件，而文件系统则向磁盘驱动程序发送一条消息来请求所需的数据块。这一顺序过程示于图3－14(a)。采用消息传递机制，我们可以使系统的各部分按一种标准的方法进行交互，同时将设备驱动程序放在核心空间将使得它们在需要时很容易就访问到进程表以及其他关键数据结构。

图3－14 用户 － 系统通信的两种组织方式。

在UNIX中，进程都有两个部分：一个用户空间部分和一个核心空间部分，如图3－14(b)所示。当执行系统调用时，操作系统以一种特殊的方式从用户空间部分切换到核心空间部分，这种结构是MULTICS的遗留物。在MULTICS中，这种切换是普通的过程调用，而不是像UNIX那样陷入后将用户部分状态保存起来。

UNIX中的设备驱动程序只是能被进程的核心空间部分调用的核心过程。驱动程序需要等待一个中断时，它调用一个核心过程从使自己睡眠，直到某一中断处理程序将它唤醒，注意这里睡眠的只是用户进程自己，因为核心部分和用户部分是一个进程的完全不同的两个实体。

在操作系统的设计者中，关于整体式系统（如UNIX）和按进程构造的系统（如MINIX）的争论永无休止。MINIX的结构更好（更模块化），各部分之间的接口更清晰，也更容易扩展到分布式系统。而UNIX则更高效，因为过程调用比消息传递要快。MINIX被分成许多进程，因为我们相信随着计算机性能的提高，清晰的软件结构更为重要。

本章中将讨论RAM盘、硬盘、时钟和终端的驱动程序。 MINIX实际上也包括软盘驱动程序，但此处不予详细讨论。MINIX发布的软件中包含有其他一些设备的驱动程序，如RS－232串口线、SCS1接口、CD－ROM、以太网卡、以及声卡等等，将MINIX重新编译即可将它们链入系统中。

这些任务与系统中其他部分的接口是一样的：将请求的消息发给相关的任务，消息中的若干域用来存放操作码（例如READ或WRITE）和相关的参数。任务则力图执行收到的操作请求并返回一应答消息。

块设备的请求和应答消息的域结构示于图3－15。请求消息包括：发送或接收数据的缓冲区地址。应答消息包括相应的状态信息，它们供请求的进程验证请求是否被正确地执行。字符设备的域与此类似，但对各个不同的服务可能略有不同。例如发给时钟任务的消息中包含时间，而发给终端任务的消息则包含一数据结构指针，该数据结构中保存终端的各种配置信息，如行编辑键、删除键和抹行符等。

图3－15 从文件系统发送到块设备驱动程序的消息的各个域，以及应答消息的各个域。

各设备驱动程序的任务是接收其他的进程发来的请求并执行之。通常情况下是来自文件系统的请求。所有的块设备进程都是首先获取消息，然后执行和返回应答信息，出于简单起见，这样的处理过程是完全串行的，没有任何多道处理的成分。当发出一个硬件请求时，设备任务进程执行一个RECEIVE操作，指明它只接收中断消息，新来的请求消息将被保持到当前的工作结束为止。终端任务略有不同，因为单个任务要为若干设备服务，所以可以在从串行线上读数据的同时接收新的从健盘输入的请求。但每一个设备都必须执行完上一个请求后才能开始另一个新的请求。

所有块设备驱动程序中的主程序都如图3－16所示。当系统启动时，所有的驱动程序轮流进行内部数据结构的初始化，然后便阻塞等待接收消息。当收到一条消息后，先将调用者标识保存下来，然后开始为请求服务。其中对不同操作将调用不同的过程。当服务结束后，则向调用者发回应答，最后再继续等待下一个请求。

```
message  mess;                                  /* 消息缓冲区 */
void io_task() {
     initialize();                      /* 只在系统初始化期间作一次 */
     while (TRUE) {
          receive(ANY, &mess);               /* 等待一个请求 */
          caller = mess.source;         /* 消息来源 */
          switch(mess.type) {
               case READ:    rcode = dev_read(&mess);  break;
               case WRITE:   rcode = dev_write(&mess);  break;
               /* 这里还有其他分支，如OPEN、CLOSE及IOCTL */
               default:      rcode = ERROR;
     }
     mess.type = TASK_REPLY;
     mess.status = rcode;                       /* 执行结果代码 */
     send(caller, &mess);                       /* 向调用者发送应答消息 */
     }
}
```

图3－16 一个I/O任务主过程的框架。

设备驱动程序中的每一个dev_xxx过程执行一种该驱动程序能做的操作。它返回一个状态码，该状态码包含在应答消息中的REP_STATUS域中。若其值为负，则表示这是一个错误码，为零或正值时指明传输的字节数。该值有可能与请求的字节数不等。如遇到文件尾时，所得到的字节数数就可能小于请求的字节数。在终端上，最大只能返回一行的字符数。

3.4.3 MINIX中与设备无关的I/O软件

在MINIX中所有与设备无关的软件都包含在文件系统进程中。因为I/O系统与文件系统的联系非常紧密，所以也可以合并到一个进程中。文件系统执行的功能示于图3－5中，其中不包括对独占设备的申请和释放。目前的MINIX中不包含对独占设备的支持，但以后在需要时可以很容易地加到相关设备的驱动程序中。

文件系统的工作包括与驱动程序、缓冲、数据块分配等的接口，同时也处理对i-节点、目录、当前被安装文件系统的保护和管理等，文件系统将在第5章详细讨论。

3.4.4 MINIX中用户级I/O软件

本章前边给出的框架在这里也适用，系统提供了必要的库例程来执行系统调用和所有POSIX标准要求的C函数，例如格式化输入和输出函数printf和scanf。标准的MINIX配置包括一个Spooling精灵进程lpd，它处理所有lp命令提交的打印文件。标准的MINIX发行软件包含许多支持网络功能的精灵进程，网络操作需要一些本书中的MINIX不提供的操作系统支持，但MINIX可以很容易被重新编译，从而加入网络服务进程，它与内存管理进程和文件系统进程一样作为用户进程运行。

3.4.5 MINIX的死锁处理

MINIX继承了UNIX的死锁处理办法：仅仅简单地忽略它。尽管将一台标准DAT磁带机装上使用不会带来任何问题，但是MINIX还是不支持任何独占的I/O设备。简言之，死锁可能发生的唯一环节是在使用隐含的共享资源时，例如进程表项、i-节点表项等等，没有一种已知的死锁算法能够解决这种非显式申请资源的问题。

实际上，MINIX在几个地方都采取了非常谨慎的措施来避免死锁发生，主要的一点是文件系统与内存管理程序之间的交互。在执行EXEC系统调用时，内存管理程序通过向文件系统发送消息来读入可执行文件。如果此时文件系统正忙，则内存管理程序将阻塞；如果文件系统随后试图向它发送消息时，文件系统也因此而阻塞，则死锁将发生。

其解决办法是不允许文件系统向内存管理程序发“请求”消息，而只发“应答”消息。这其中有一点小的例外，即在文件系统启动时，它向内存管理器报告RAM盘的大小，而此时内存管理器一定正在等待该消息到达。

不通过操作系统支持也可以对设备和文件加锁，文件名可以当作一个真正的全局变量，其存在与否可以被

所有其他进程感知。在MINIX和多数UNIX系统中都存在一个特殊的目录/usr/spool/locks供进程创建加锁文件，以标志其正在使用的资源。MINIX文件系统也支持POSIX风格的建议锁机制。但是MINIX和POSIX的锁都不是强制性的，这取决于进程的行为，而且没有什么可以阻止一个进程使用另一个进程加锁的资源。这与资源的剥夺并不完全一样，它并不阻止第一个进程继续使用该资源，亦即不存在互斥访问。进程的这种不正常行为可能会搞得一片混乱，但不会导致死锁。

3.5 MINIX中的块设备

下面我们回到本章的重点－设备驱动程序。首先将讨论块设备共性的部分，随后将详细讨论RAM盘、硬盘和软盘。RAM盘是一个好的开端，因为它具有块设备的所有共性，同时又省略了真正的I/O操作－－它只是内存的一个部分。硬盘展示了一个真实的磁盘驱动程序的特征。有人会误认为软盘比硬盘容易一些，但事实上不是这样。我们不打算讨论软盘的完整细节，但将指出其中的复杂之处。

在块设备之后将讨论其他类型的设备驱动程序。时钟是所有系统都必备的，而且与其他设备截然不同，它还有一个有趣的例外：它既不属于块设备，也不属于字符设备。最后是终端驱动程序，它在所有的系统中都很重要，而且是典型的字符设备。

以下每一部分都将描述相关的硬件、驱动程序的有关软件原理、实现的概述、以及代码，这种安排将使这部分内容对那些不愿阅读代码的读者也颇有价值。

3.5.1 MINIX中块设备驱动程序概述

我们提到过所有I/O服务进程的主框架都有类似的结构。MINIX至少有三种设备驱动程序（RAM盘、软盘、硬盘），此外还可以将CD－ROM和一个SCSI设备驱动程序加进来。尽管每一种设备驱动程序都作为独立进程运行，但由于它们都是整个可执行核心代码的组成部分，所以有可能使它们共享相当数量的代码，尤其是实用例程。

块设备驱动程序首先要作一些初始化，RAM盘驱动程序要预留一些内存空间、硬盘驱动程序要确定硬盘参数等等。所有磁盘驱动程序在相应的硬件初始化时都被调用，但随后它们将调用包含一公共主循环的函数，该循环将一直执行下去，不会返回到调用者。主 循环执行的工作是：接收一条消息，执行相应的操作，然后发回一条应答消息。

各驱动程序所公用的主循环并不是每个驱动程序都拥有一份拷贝，在MINIX二进制代码中只存在一份拷贝。它使用的技术是各驱动程序向主循环传递一个参数，它是一个指向一个表的指针，该表中包含了完成各具体操作的函数指针，于是以后就可以间接调用这些函数。这种技术也使得各驱动程序能够共享函数。图3－17示出了主循环的框架。其中如下语句：

```
code  = ( *enty_points -> dev_read)  (& mess);
```

是间接函数调用，尽管各驱动程序执行同一个主循环，但它们调用不同的dev_read函数。不过有些操作，例如CLOSE非常简单，可供多个设备共享。

```
message  mess;                    /* 消息缓冲区 */
void shared_io_task(struct driver_table * entry_points) {
/* 每个任务调用本过程之前先完成初始化 */
     while (TRUE) {
          receive(ANY, &mess);
          caller = mess.source;
          switch(mess.type) {
               case READ:    rcode = ( *entry_points->dev_read) (&mess); break;
               case WRITE:   rcode = ( *entry_points->dev_write)(&mess); break;
               /* 这里还有其他分支，如OPEN、CLOSE及IOCTL */
               default:      rcode = ERROR;
          }
          mess.type = TASK_REPLY;
          mess.status = rcode;        /* 执行结果代码 */
          send(caller, &mess);
     }
}
```

图 3－17 一个使用间接调用的共享I/O任务。

这种共享一个主循环的方法很好地解释了前边提到的进程概念。内存中只有一个执行代码的拷贝，但它同时为多个独立的进程执行，在一给定时刻，各进程可能位于代码的不同点上，但它们使用各自的数据和堆栈。

所有设备驱动程序都有6种可能的操作，它们对应于图3－15所示的消息结构m.m_type域的各种可能值，它们分别是：

1 OPEN
2 CLOSE
3 READ

4 WRITE
5 IOCTL
6 SCATTERED_IO

有编程经验的读者对这些操作可能很熟悉。在设备驱动程序层，大多数操作与同名的系统调用相关。例如READ和WRITE的意义就非常清楚，READ操作将从设备读取一个数据块到调用进程的内存区域；WRITE则正好相反。调用READ时进程在数据传输完成前将一直阻塞；而对于WRITE，操作系统有可能将数据暂存在缓冲区中，随后再真正将其传送到设备，这时WRITE系统调用将很快就返回调用进程。这对调用进程很有利，因为它随后可以再次使用这次暂存了写出数据的缓冲区。OPEN和CLOSE的含义与文件的OPEN和CLOSE类似：OPEN 操作将验证设备是否可用，当不可用时则返回一条错误消息，CLOSE将确保把先前延迟写的数据真正写到设备上。

对IOCTL可能不大熟悉。许多I/O设备都有一些操作参数，经常要对这些参数进行检查，也可能要修改。IOCTL的任务就是做这些工作。常见的一个例子是改变通信线路的传输速率和奇偶校验方式。块设备IOCTL的操作不大常用，在MINIX中，检查和改变磁盘设备的分区是用IOCTL完成的（尽管也可以通过读写数据块完成）。

SCATTERED_IO操作无疑是最少见的，除了个别非常快的磁盘设备（如RAM盘）外，如果每次只请求读写一块，那么其I/O性能很难提高。一个SCATTERED_IO请求允许文件系统读写多个块。对于READ来说，要求多读的块可能并不是由执行该调用的进程所申请的，而只是操作系统试图预测将来对数据的请求。这种请求并不一定由设备驱动程序实现。对每一个块的请求可以被一个标志位修改，它通知驱动程序该请求是可选的。实际上文件系统可以声明：“最好将这些数据全部都拿来，但实际上并非马上就要用”。设备可以据此相机行事。例如软磁盘驱动程序将返回整条磁道上的数据，相当于：“我把这些数据给你，但移动到另一条磁道上操作太费时了，在你需要时再通知我。”

当写数据时，则不存在上述问题。但操作系统可能会暂存许多写请求随后一次写出，这比每次处理单个请求的效率要高。在SCATTERED_IO请求中，申请读写块的请求被排序，这比随机地处理要高效。而且一次调用驱动程序传输多个块减少了需要发送的消息数。

3.5.2 公用块设备驱动程序软件

块设备驱动程序需要的定义都放在driver.h中。其中最重要的是driver结构（9010～9020行）。其中保存了各驱动程序执行具体I/O操作的函数地址。该文件中还定义了device结构（9031～9034行）。其中保存了与分区相关的最主要信息：基地址和长度，它们都以字节为单位。采用这种格式使得对基于内存的设备（RAM盘）无需作任何转换，由此最大程度地提高了响应速度。而对于真正的磁盘，由于有很多因素影响其存取速度，因而转换到扇区地址并不增添很多麻烦。

所有块设备驱动程序共享的主循环及其他函数都在driver.c中，在硬件执行完必要的初始化后，驱动程序都调用driver_task，同时向其传入一个driver结构作为参数。在获得一个供DMA操作使用的缓冲区地址后进入主循环（9156～9199行）。该循环将一直执行下去，不返回到调用进程。

文件系统是唯一向设备驱动任务发送消息的进程。第9165到9175行的switch语句正是对此进行检查。硬件发出的多余中断被忽略，所有错误地发过来的消息将在屏幕上打印一条警告信息。这看起来似乎没什么副作用，但显然发送了错误消息的进程有可能一直阻塞以等待应答。在主循环中的switch 语句中，前三种消息DEV_OPEN、DEV_CLOSE、DEV_IOCTL将使用driver结构传来的地址进行间接调用，而DEV_READ、 DEV_WRITE和SCATTERED_IO消息将直接调用do_rdwt和do_vrdwt。但driver数据结构被switch中的所有调用作为一个参数传递，不论是直接还是间接，所以所有的被调用例程在需要时都可以进一步使用它。

在按消息中的请求进行处理后，根据设备本身的特性需要作一些清理操作。例如对于软盘，可能会启动一个定时器，以便在一段时间内下一个请求未达到的情况下关闭驱动器的电机。间接调用也可被用于此种目的。在清理操作之后，随之构造一个应答消息并传送给调用者（9194到9198行）。

在进入主循环后，各任务执行的第一个操作是调用init_buffer（9205行）。它为DMA操作分配一个缓冲区，所有的驱动程序任务假如执行DMA操作的话，都使用这同一个缓冲区，但有些驱动程序不使用DMA。除了第一次初始化之外的其他的初始化都是多余的，但并没有副作用。编写一段测试代码来确定初始化是否应被跳过将更复杂。

这段初始化操作之所以必要是由于最初的IBM PC机硬件的一个古怪之处所引起的，该PC机硬件要求DMA缓冲区不得越过64K边界。也就是说，一个1K的DMA缓冲区应从64510开始，而不是64514。因为后者正好越过65536的64K边界。

这种讨厌的规则源于IBM PC使用一种老的DMA控制器芯片－Intel 8237A，8237A含有一个16位的计数器。由于DMA使用绝对地址，而不是相对段寄存器的偏移地址，所以需要一个更大的计数器。在老式的只能寻址1M字节的机器上，DMA地址的低16位被装入8237A，高4位则被装入一个4位的锁存器。新一点的机器使用8位的锁存器，并能寻址16M字节。当8237A由0xFFFF变到0x0000时，并不向锁存器进位，导致DMA地址突然减掉了64K。

一个可移植的C程序不能为一数据结构指定绝对的内存地址，所以无法防止编译器将缓冲区放在一个不可用的位置。解决办法是分配一片大小为所需缓冲区2倍的内存buffer（9135行）。并保留一个指针tmp_buf（9136行）供访问这片内存使用。init_buffer首先尝试着将tmp_buf指向buffer的开头；然后检查它在遇到64K边界之前是否能够提供足够的空间。如果不行，则tmp_buf就递增真正所需的空间的字节数。这样在buffer的某一端总

会造成一些地址的浪费，但同时保证了不会由于64K边界的限制而造成缓冲区失败。

更新的IBM PC系列计算机使用更好的DMA控制器，所以这部分代码可以简化。而且如果所用的机器肯定不存在上述问题时，可以只申请一小部分内存便可满足要求。如果确实这样做了，那么现在让我们考虑一下假定在你判断有误的情况下，这个错误发生的有关情况。对于1K的DMA缓冲区，在使用老式的DMA控制器时，出错的概率是64分之一。每次核心代码被修改，导致编译后的核心大小发生变化时，这样的概率同样存在。当这个错误在下一个月或次年发生时，很可能归咎于最后一次修改的代码。类似的不可预知的硬件“特征”使我们可能需要花几周的时间来查找那些隐藏很深的错误（对于那些与本例类似的，在技术手册中从未提到的错误，可能会花更多的时间）。

do_rdwt是driver.c中的下一个函数。它又可能调用三个与设备相关的函数。driver结构中的dr_prepare、dr_schedule和dr_finish分别指向这三个函数。下面我们将按照C语言的记法，用*function_pointer表示function_piinter指向的函数。

在检查请求中的字节计数是正数后，do_rdwt调用*dr_prepare。该操作一定成功，因为只有当OPEN操作指定了一个非法设备时*dr_prepare才会失败。接下来将填充一个标准的iorequest_s结构（在include/minix/type.h中3194行定义）。然后是一个间接调用*dr_schedule。正如我们将要在磁盘硬件一节中即将看到的那样，简单地按照接收到请求的顺序来处理它们是不够的。*dr_schedule的功能就是允许一个特定设备按照最符合其硬件特性的方式来处理请求。这里的间接调用屏蔽了各种设备的差异。对于RAM盘，dr_schedule指向一个例程，由该例程来执行真正的I/O操作，而下一个间接调用*dr_finish什么也不作。对于真正的磁盘，dr_finish指向一个例程，该例程处理自上次*dr_finish以来积累的所有*dr_schedule调用所发出的数据传输请求。但后面将会看到，在有些情况下调用*dr_finish可能并不传送所请求的所有数据。

在执行真正数据传输的例程中，iorequest_s结构中的io_nbytes域将被改变。返回值为负表明出错；返回值为正表明请求的字节数和成功传送的字节数之间的差值。即使没有发生字节传送也并不表示一定是出错，它只是表明到了设备的结尾。当返回主循环时，如果出错，则错误码放在应答消息的REP_STATUS域；若成功则将剩余待传输字节数从初始消息的COUNT域中减去（9249行），结果（真正传送的字节数）放在由driver_task返回的应答消息的REP_STATUS域中。

下一个函数do_vrdwt处理所有散布的I/O请求。请求一个散布I/O的消息使用ADDRESS域来指向一个iorequest_s类型的数据结构数组。其中的每一项包含一次I/O操作所需的所有信息，包括：缓冲区地址、在设备上的偏移、字节数、请求是读还是写等等。一个请求中的全部操作或者都是读，或者都是写，而且将按照它们在设备上的数据块顺序排序。这里需要作的工作不仅仅是do_rdwt所作的简单的读或者写，因为该请求数组必须被拷贝到核心空间。而一旦拷贝完成，将间接地调用*dr_prepare、*dr_schedule和*dr_finish这三个与设备无关的例程。区别在于对*dr_schedule的调用是一个循环，它每收到一个请求就循环一次，或者直到发生错误（9288-9290行）。在循环结束后将调用*dr_finish。然后请求数组被拷贝回原处。数组中每一项的io_nbytes域将被修改以反映传送了多少字节。尽管总数并未在driver_task构造的返回消息中直接返回，但调用例程自己可以从该请求数组中自行计算出来。

在一个散布的I/O读请求中，当最终调用*dr_finish时，并不一定所有调用*dr_schedule的传送请求都已实现。在iorequest_s结构中的io_request域中含有一个标志位，通知设备驱动程序对某一数据块的请求是否是可选的。

在driver.c中的随后几个例程都是为以上操作提供通用的支持。*dr_name用于返回一个设备的名字。对于没有给定名字的设备，no_name函数从任务表中检索设备名。有些设备可能不需要一个特定的服务，如RAM盘在响应DEV_CLOSE请求时并不要求作任何操作，这时就使用do_nop函数，它仅仅根据请求的类别返回不同的码值。接下来的函数nop_finish和nop_cleanup则分别用作那些不需要*dr_finish和*dr_cleanup服务的设备的哑例程。

有些磁盘设备需要延时，例如等待一个软驱的步进电机加速。为此driver.c中包含了提供相应功能的例程clock_mes。clock_mess用来向时钟任务发送消息。该例程的参数包括：需等待的时钟滴答数，以及等待时间到之后需调用函数的地址。

最后，do_diocntl（9364行）执行块设备的DEV_IOCTL请求。DEV_IOCTL请求只能是读（DIOGETP）或写（DIOSETP）分区信息，除此之外均出错。do_diocntl调用设备的*dr_prepare来验证设备是否合法，并得到一个指向device结构的指针。该结构描述了按字节计数的分区基址和大小。对于读请求，它调用设备的*dr_geometry函数来获得该分区的最后柱面号、磁头及扇区信息。

3.5.3 驱动程序库

文件drvlib.h 和drvlib.c包含一些与系统有关的代码，这些代码支持IBM PC及兼容机的磁盘分区。

分区机制允许单个的外存设备被分成若干子设备，它主要用于硬盘，但MINIX还提供对软盘的分区。对磁盘分区的理由是：

1 大容量磁盘单位价格便宜。如果有两个或更多的操作系统在使用不同的文件系统，则将一个大硬盘分区比为各个操作系统安装各自的硬盘更经济。

2 操作系统能够处理的设备的大小可能有限。

3 一个操作系统可能使用两个或更多的文件系统。例如，一个标准的文件系统用于普通文件，同时一个不同结构的文件系统用作虚存的交换区。

4 将一个系统的一部分文件放在一个独立的逻辑设备上可能方便一些。将MINIX根文件系统放在一个小的设备上将使其更便于备份，而且有助于在引导时将其拷贝到一个RAM盘中。

磁盘分区的支持是平台有关的，这种特定性与硬件没有关系。分区支持独立于设备。但如果一台设备上有多个操作系统运行，则它们必须对分区表的格式达成一致。在IBM PC上该标准由MS－DOS的fdisk命令确定。其他操作系统，如MINIX, OS/2以及Linux也使用该命令，以便与MS－DOS共存。当MINIX被移植到另一种机器时，应该使用在这种新硬件上运行的操作系统所采用的分区表格式。所以在MINIX中支持IBM计算机分区的源码部分被放在drvlib.c中，而不是放在driver.c中，目的是使其能更容易地被移植到其他硬件上。

从固件设计人员继承过来的基本数据结构定义在include/ibm/partition.h中，它通过#include 语句被包含在drvlib.h中。其中包含有每个分区的柱面－磁头－扇区格式的信息，以及标识文件系统类型的代码和一个分区是否可引导的标志。对文件系统进行了检查之后，其中多数信息便不再需要。

当第一次打开一个块设备时，将调用partition函数（drvlib.c，9521行）。其参数包括一个driver结构（这样它便可以调用与设备相关的函数）、一个初始的次设备号、以及另一个参数，它标识分区类型是软盘、主分区或次分区。partition函数调用设备相关的*dr_prepare函数来验证设备合法，并获得一个device结构的基地址和大小。然后它调用get-part-table来判断分区表是否存在。若存在则将其读入，如果分区表不存在则操作结束。否则用最初调用时使用的分区编号规则计算第一个分区的次设备号。在使用主分区的情况下，分区表是经过排序的，所以分区的顺序与其他操作系统所使用的相同。

在这里将第二次调用*dr_prepare，这次使用刚计算出来的第一个分区的设备号。如果子设备合法，则对表中所有项执行一个循环，每次检查从设备上的表中读出的值是否超越先前获得的整个设备的基址和范围。如果出现不一致，则内存中的表格被修改以取得一致。这似乎不大合适，但由于分区表可能被不同的操作系统修改，使用另一个操作系统的程序员可能很聪明地试图利用分区表作一些不可预知的事情，或者由于别的原因磁盘上的分区表中可能会有垃圾信息。所以我们更相信使用MINIX计算出来的数值。这里的思想是安全总比出错要好。

还是在这个循环中，对于设备上所有标识为MINIX的分区，递归调用partition函数以获取其子分区信息。如果一个分区被标识为一个扩展分区，则调用drvlib.c中的下一个函数extpartition。

extpartition（9593行）实际上与MINIX无关。所以这里不进行详细讨论。MS－DOS使用扩展分区，它实际上是另外一种创建子分区的机制。为了支持读写MS－DOS文件的MINIX命令，我们需要知道这些子分区。

get-part-table（9642行）调用do_rdwt来从驻留分区表的设备（或子设备）上获取一个扇区。如果它用来获取一个主分区则偏移参数为0；如果是次分区则偏移为非0。它检查分区表的魔数（0xAA55），并根据是否找到了合法的分区表而分别返回真或假。若找到一个分区表，则将其拷贝到由输入参数指定的表地址处。

最后，sort函数（9676行）按最低扇区将分区表中的表项排序。被标识为不包含分区的表项则不参与排序，所以被排在最后，哪怕其最低扇区号为0。这种排序只是简单的冒泡排序，对于一个只有四项的表，没必要使用特别优化的算法。

3.6 RAM盘

现在我们回到各个设备驱动程序，并对其中的几个进行详细研究。我们将研究的第一个是RAM盘驱动程序，利用它可以访问存储器的任何部分，它的主要用途是保留一部分存储器，并象普通磁盘一样来使用它。RAM盘并不提供永久存储，但是一旦文件被拷贝到这一区域，就可以极快的速度进行访问。

MINIX被设计成在仅有一个软盘的计算机上也可以运行，在一个这样的系统中，RAM盘还有一个优点，那就是通过把根文件设备放在RAM盘上，就可以随意地安装和卸下软盘，从而支持了可移动介质。因为不能卸下根设备，所以如果把根文件系统放在软盘上，就不能将文件存在软盘上。另外把根文件系统放到RAM盘可以使系统具有很大的灵活性：软盘和硬盘的任意组合都能安装上去。虽然除了在嵌入系统中使用的计算机外，现在大多数的计算机都有硬盘，但是，在MINIX准备好使用硬盘之前的安装过程中，或者当需要临时使用MINIX而不进行完整的安装时，RAM盘是很有用的。

3.6.1 RAM 盘硬件和软件

RAM盘的思想很简单。块设备是具有两个操作命令的存储介质：即写数据块和读数据块，通常这些数据块存储于旋转存储设备上，例如软盘和硬盘。RAM盘则简单得多，它使用预先分配的主存来存储数据块。RAM盘具有快速存取的优点（没有寻道和旋转延迟），适于存储需要频繁存取的程序和数据。

有些系统支持可安装的文件系统，有些则不支持（例如MS－DOS，WINDOWS），这里简略地指出它们的区别。对于支持可安装文件系统的系统，根设备总是位于固定的位置，可移动的文件系统（即磁盘）可以安装到文件树上从而构成一个统一的文件系统。一旦安装完毕，用户就不必关心一个文件在哪个设备上。

相反地，对于象MS－DOS这样的系统，用户必须指定每个文件的位置，要么象B:\DIR\FILE一样显式指出，要么使用一定的默认值（当前设备、当前目录等等）。在仅有一两个软盘驱动器时，管理的负担还可以承受。但对于拥有几十个磁盘的大型计算机系统而言，要在任何时刻跟踪所有的设备，其负担是无法忍受的。请注意：UNIX

系统可以运行在从IBM－PC、工作站和超级计算机，一直到Cray－2这样的系统；而MS－DOS仅在小系统上运行。图3－18 给出了实现RAM盘的思想。根据为RAM盘分配内存的大小，RAM盘被分成n块，每块的大小和实际磁盘块的大小相同。当驱动程序接收到一条读写一个数据块的消息时，它只计算被请求的块在RAM盘存储区的位置，并读出或写入该块，而不读写软盘或硬盘。数据的传输通过调用一个汇编语言过程来实现，该过程以硬件所能实现的最高速度把数据拷贝到用户程序或从用户程序拷出。

图3－18　　一个RAM盘。

一个RAM盘驱动程序可以支持将存储器中的若干区域当作RAM盘来使用，每个RAM盘用次设备号来区分。一般情况下这些存储区相互分开，但正如下一节我们将看到的，在某些情况下，使它们相互重叠可能会更方便。

3.6.2　MINIX中的RAM盘驱动程序概述

RAM盘驱动程序实际上是将四个紧密相关的RAM盘驱动程序放在一起，传给驱动程序的每条消息都要指定下列次设备之一：

0：/dev/ram　　1：/dev/mem　　2：/dev/kmem　　3：/dev/null

其中第一个文件/dev/ram是真正的RAM盘，驱动程序内部既不指定它的大小，也不指定它的起始地址，而是在MINIX启动时由文件系统确定。缺省情况下将创建一个与根文件系统设备大小相同的RAM盘，以便将根文件系统拷贝到RAM盘。可以通过启动参数来指定一个容量大于根文件系统的RAM盘；如果不把根文件系统拷贝到RAM盘，则在为系统操作留下足够的内存的前提下，RAM盘的容量可以取内存可容纳的任意值。一旦RAM盘大小确定，则系统就寻找一块足够大的内存并将其从内存池中移出，该操作甚至发生在内存管理器开始工作之前。这种策略使得不必重新编译操作系统即可增加或减少RAM盘的容量。

随后的两个次设备分别用于读写物理内存和核心内存。当打开/dev/mem并进行读操作时，读出的是起始于绝对地址零的内容（实模式下的中断向量）。普通用户程序永远不会执行这个操作，但与系统调试有关的程序可能会使用这种功能。打开并对/dev/mem执行写操作将修改中断向量。显然，只有那些对该操作的结果十分清楚的的熟练用户才可非常谨慎地执行这个操作。

设备文件/dev/kmem和/dev/mem相似，只是其第0字节是核心数据存储区的第0字节，而其绝对地址随MINIX核心代码的大小而变。同样地，它也是主要用于调试以及非常特殊的程序。注意这两个次设备覆盖的存储区是重叠的，如果你精确地知道核心在内存中如何放置，则你可以打开/dev/mem文件，将文件指针定位到内核数据区的起点，你会发现这里的数据和从文件/dev/kmem开头读出的数据相同。但如果你重新编译核心并改变其大小，或者如果在MINIX的后续版本中内核被放在内存的其他位置，那么你将不得不把指针定位到不同的位置才能发现和从文件/dev/kmem开头读出的一样的数据。对这两个文件必须采取保护措施以保证超级用户之外的其他用户不能使用它们。

最后一个文件 /dev/null 是一个接收数据并把数据抛弃掉的设备文件。在执行shell命令时，如果程序产生的结果不再需要，则可以使用它，例如：

```
a.out >> /dev/null
```

将执行a.out但丢弃输出结果。RAM盘驱动程序对它采用一种高效的处理方式：将其长度置为零，这样便没有数据对它拷入或拷出。

处理/dev/ram、/dev/mem 和/dev/kmem的代码是相同的，区别仅在于它们各自对应不同的存储区，该存储区由数组ram_origin和ram_limit指示，两数组均由次设备号进行索引。

3.6.3　MINIX中的RAM盘驱动程序实现

和其他的磁盘驱动程序一样，RAM盘驱动程序的主循环在文件driver.c中，和设备相关的支持存储器设备的代码在memory.c中。数组 m_geom（9721行）存放了四个存储器设备的基地址和大小。9733至9743行的driver结构m_dtab定义了将在主循环中执行的存储器设备调用。该表中的四个入口在driver.c中都属于几乎没执行什么操作的例程，这也证实了RAM盘操作并不复杂。主过程mem_task（9749行）调用一个函数来进行一些初始化，然后调用主循环。主循环完成取消息、将其分派到相应的过程、然后发出应答。执行完毕后并不返回到mem_task。

执行读或写操作时主循环执行三个调用：一个准备设备，一个调度 I/O操作，一个结束操作。对于存储器设备，首先调用m_prepare，它检查请求的次设备是否合法，然后返回一个结构的地址，该结构中存放着所请求RAM区的基地址和长度。第二个调用的是 m_schedule（9774行），它执行所有的工作。对于存储器设备，这个函数名起的并不恰当。根据定义，在RAM中任何位置都可以和其他位置一样访问，因此没有必要做任何调度，只有对于有移动臂的磁盘才需要。

RAM盘的操作非常简单和快捷，所以根本不会出现需要延迟一个请求的情况。它所做的第一件事就是清除由离散I/O调用设置的一个比特，该比特用来指示一个操作的完成为可选。在消息中传进来的目标地址指向调用者存储区的一个位置，9792行到9794行的代码将这个地址转换成存储器中的绝对地址，并检查它是否合法。9818行到9820行的代码执行实际的数据传输，它直接把数据从一处拷贝到另一处。

一个存储器设备不需要第三步来结束一个读写操作，在m_dtab中相应的内容为调用nop_finish。

m_do_open（9829行）打开一个存储器设备，其主要任务是调用m_prepare来检查引用的设备是否合法。当引用/dev/mem 和/dev/kmem时，将调用enable_iop（在文件protect.c中）来改变当前CPU的特权级。当访问存储器时，这是不需要的。这是处理另外一个问题的技巧。请注意奔腾型的CPU实现了四个特权级，用户程序运行

在最低特权级。Intel处理器还具有一个在许多其他系统中不存在的体系结构特性：一套独立的寻址I/O端口的指令集。在这类处理器中，I/O端口和存储器是分开处理的。通常，用户进程试图执行一条访问I/O端口的指令将引起一个通用保护异常。然而对于MINIX，存在一些理由应该允许用户编写访问端口的程序，特别是在小系统中。于是使用enable_iop改变CPU的I/O保护级别（IOPL）位从而允许访问端口，其结果是赋予允许打开/dev/mem或/dev/kmem的进程访问I/O端口的特权。在把I/O设备作为存储器来访问的系统结构中，这些设备的rwx位自动地覆盖了对I/O的访问。如果隐藏了这个特性，则可能被认为是安全方面的缺陷，不过现在你已经知道了这一点。如果你计划使用MINIX来控制银行的安全系统，你可能希望重新编译核心以去掉这一函数。

下一个函数m_init（9849行）仅当第一次调用mem_task时被调用一次，它设置/dev/kmem的基地址和长度，并根据MINIX是运行在8088、80286还是80386模式下，将/dev/mem的长度分别设置为1MB、16MB、或4GB－1。这些长度值是MINIX支持的最大长度，和机器上安装的RAM多少无关。

RAM盘支持函数m_ioctl（9874行）中的几个IOCTL操作。MIOCRAMSIZE是文件系统设置RAM盘大小的简便方法。MIOCSPSINFO被文件系统和内存管理器用来将进程表中它们对应部分的地址设置在psinfo表中。实用程序ps使用MIOCSPSINFO来取出这些数据，ps是一个标准的MINIX实用程序，MINIX的微内核结构使它变得很复杂，这是因为这种结构将ps所需要的进程表信息放在几个不同的地方。IOCTL系统功能调用是处理这个问题的一种简便方法。否则每次编译MINIX的新版本时都必须编译一个新版本的ps。

memory.c中的最后一个函数是m_geometry（9934行），存储器设备没有机械驱动器的柱面、磁道和每条磁道上的扇区等几何结构，但出于RAM盘一旦被询问到这些参数的考虑，所以也要模拟提供这些参数。

3.7磁盘

利用RAM盘来介绍磁盘驱动程序很合适（因为它很简单），不过实际磁盘有一些我们尚未接触到的特性。下面几节中我们将首先简单讨论一下磁盘硬件，然后笼统地分析一下磁盘驱动程序，并重点分析MINIX的硬盘驱动程序。我们不详细讨论软盘驱动程序，但是将述及软盘驱动程序和硬盘驱动程序工作方式的不同之处。

3.7.1磁盘硬件

所有实际的磁盘都组织成许多柱面，一个柱面上的磁道数和垂直放置的磁头个数相同。磁道又被分成许多扇区，每条磁道上扇区数目典型的范围是：对软盘每条磁道8至32扇区，在某些硬盘上则可多达几百扇区。最简单的设计是每条磁道具有相同的扇区数，每个扇区包含相同数目的字节。然而，略加思索就可以知道物理上靠近磁盘外边沿的扇区比靠近磁盘内边沿的扇区要长一些。不过读写每个扇区的时间是一样的。很明显在最里面的柱面上的数据密度要高一些，一些磁盘的设计要求读写内部磁道是改变磁头的驱动电流，这由磁盘控制器硬件处理，对用户（或操作系统的实现者）不可见。

内圈磁道和外圈磁道数据密度的不同意味着会牺牲一些磁盘容量，也意味着存在更复杂的系统。有人尝试过设计一种当磁头处于外部磁道时磁盘旋转速度更快的软盘，这样外圈磁道就可以具有更多的扇区，从而增加磁盘的容量。目前安装MINIX的系统还不支持这种磁盘。然而，现代大容量硬盘中外圈磁道具有的扇区数比内圈磁道更多，这就是IDE（Integrated Drive Electrinics）驱动器，它内置的电子器件进行的复杂的处理屏蔽了具体细节，对于操作系统来说，它们仍然呈现出简单的几何结构，每条磁道具有相同的扇区。

驱动器与控制器电子器件和机械硬件同等重要。插到计算机主板上的控制卡的主要元件是一块专用的集成电路，实际上是一个小的微型计算机。硬盘驱动卡的控制线路可能比软盘的还要简单，这是因为硬盘驱动器本身具有一个功能强大的电子器件控制器。对于磁盘驱动程序有重要意义的一个设备特性是：控制器可以同时控制两个或多个驱动器进行寻道，这称为重叠寻道（overlapped seeks）。当控制器和软件等待一个驱动器完成寻道时，控制器可以启动另一个驱动器进行寻道。许多控制器可以在对一个或多个其他驱动器寻道的同时在一个驱动器上进行读写操作，但是软盘控制器不能同时读写两个驱动器（读写数据要求控制器在毫秒级的时间内传输比特流，所以一个传输就基本占用了所有的计算能力）。对于具有集成控制器的硬盘则情况不同，在一个有多个硬盘的系统中，这些驱动器可以同时操作，至少在磁盘和控制器缓冲区之间的数据传输可以并行进行。然而，在控制器和系统内存之间只能同时执行一个传输，同时执行两个或多个操作的能力极大地降低了平均读写时间。

图3－19对用作最初IBM－PC标准存储介质的双面双密度软盘的参数和一个可能在奔腾计算机上发现的典型中等容量硬盘驱动器的参数。MINIX使用1K的数据块，因此，对于这两种磁盘，软件使用的数据块由两个连续的扇区组成，它们总是作为一个单元进行读写。

图3－19　初始IBM－PC 360K软盘和Western Digital WD AC2540 540－MB硬盘的磁盘参数。

在阅读现代硬盘规格时需要注意的一点是驱动程序软件所指定和使用的几何参数可能与物理格式不同。例如，图3－19所描述的硬盘由推荐设置参数指定为1048柱面、16磁头、每条磁道63扇区。磁盘上的电子器件将操作系统提供的逻辑磁头和扇区参数转换成磁盘使用的物理参数。这也是为了兼容旧系统（对老的固件）而采取妥协的另一个例子。最初的IBM－PC的设计者仅为BIOS ROM 中的扇区数域分配了六个比特，这样每条磁道的扇区数超过63的磁盘必须工作在一套模拟的逻辑磁盘参数上。在这种情况下，厂商提供的规格指出实际上有四个磁头，所以如图中所示，实际上每条磁道有252个扇区。因为这类磁盘最外面的磁道比最里面的磁道有更多的扇区，所以这里实际上已经是简化了。图中所描述的磁盘确实有四个物理磁头，但是柱面数比3000略多一些。柱面被分成了十二个组，最内部的磁道有57个扇区，最外部的磁道有105个扇区，这些参数在磁盘的规范中是没有

的。驱动器的电子器件完成了这个转换，用户便无需了解这些细节。

3.7.2 磁盘软件

本节将笼统地浏览一下和磁盘驱动程序有关的内容。首先考虑读写一个磁盘块需要多长时间。需要的时间由下面三个因素决定：

1 寻道时间（将磁头臂移动到相应的柱面所需的时间）。

2 旋转延迟（相应的扇区旋转到磁头下面所需的时间）。

3 实际的数据传输时间。

对于大多数磁盘，和另外两个时间参数相比，寻道时间要大得多，因此减小平均寻道时间将极大地提高系统性能。

磁盘设备可能会出错。所以常将某种错误校验 — 校验和或者循环冗余校验的信息和各扇区的数据同时记录在盘上，甚至磁盘格式化时记录的扇区地址也有校验数据。当检测到出错时，软盘控制器可以报告出错信息，但是必须由软件决定出错后该怎么办。硬盘控制器则通常承担大部分出错处理工作。

尤其对于硬盘，一条磁道上连续扇区的传输是非常快的，因此读取比请求数据更多的数据并把它缓冲在内存中对加速磁盘存取非常有效。

磁盘臂调度算法

如果磁盘驱动程序每次接收一个请求并按照接收顺序执行，即先来先服务（FIFS），则很难优化寻道时间。然而当磁盘负载很重时，可以采用其他策略。很有可能当磁盘臂为一个请求寻道时，其他进程也产生了其他磁盘请求。许多磁盘驱动程序都维护一张表格，按柱面号索引，每一柱面的全部请求用一个链接表链在一起，链表头指针存放在表格相应的表目中。

有了这种数据结构，我们就可以改进先来先服务的调度算法。为了说明如何实现，考虑一个具有四十个柱面的磁盘。假设一个请求到达请求读柱面11上的一个数据块，当对柱面11寻道时，又顺序到达了新请求要求寻道1 、36 、16 、34 、9 和12，则它们被安排进入请求等待表，每一个柱面对应一个单独的链表。图3－20显示了这些请求。

图3－20 最短寻道（SSF）磁盘调度算法。

当前请求（请求柱面11）结束后，磁盘驱动程序需要选择下一个请求。若使用FCFS，则首先选择柱面1，然后是 36 ，依次类推。这个算法中磁盘臂分别需要移动 10 、35 、20 、18 、25和3个柱面，总共需要移动111个柱面。

为了减少寻道时间，也可以总是选择和磁盘臂最接近的柱面请求，对于图3－20所示的请求，被选择请求的顺序如图3－20下方折线所示，依次为12 、9 、16 、1 、34和36。按照这个顺序，磁盘臂分别需要移动1 、3 、7 、15 、33和2个柱面，总共需要移动61个柱面。这个算法 — 最短寻道算法Shortest Seek First（SSF）和FCFS算法相比，把柱面移动数减小了一半。

不幸的是，SSF算法存在一个问题。假设当图3－20所示的请求正在被处理，有其他请求不断到达。例如，磁盘臂移到16柱面以后，到达一个对柱面8的请求，那么它的优先级将比柱面1要高。如果接着又到达了一个对柱面13的请求，磁盘臂将移到柱面13而不是柱面1。对于一个负载很重的磁盘，磁盘臂趋向于大部分时间停留在柱面中部区域，对两端极端区域请求的处理不得不等待，直到由于负载的统计波动使得中部区域没有请求为止。远离柱面中部区域的请求得到的服务很差。这里获得最小响应时间的目标和公平性之间存在冲突。

高层建筑也得进行这种折衷处理，高层建筑中的电梯调度问题和调度磁盘臂很相似。电梯请求不停地到来，随机地呼唤电梯到各个楼层，控制电梯的微处理器能够很容易地跟踪用户按下按钮的顺序，并使用FCFS算法为他们提供服务，也可以使用SSF算法。

然而，大多数的电梯使用另一种算法来调和效率和公平性的冲突。电梯保持按一个方向运动，直到在那个方向上没有更远的请求为止，然后改变方向。这个算法在磁盘领域和电梯领域都被称为电梯算法（elevator algorithm），它需要软件维护一个二进制位：即当前的方向：向上或是向下。当一个请求结束之后，磁盘或电梯的驱动程序检查该位，如果是向上，磁盘臂或电梯舱移至下一个更高的等待请求。如果更高的位置没有请求，就翻转方向位。如果方向位设置为向下，同时存在一个低位置的请求，则移向该位置。

图3－21显示了与图3－20相同的七个请求使用电梯算法的情况。假设初始方向位为向上，则各柱面获得服务的顺序是 12 、16 、34 、36 、9和1，磁盘臂分别移动1 、4 、18 、2 、27和8个柱面，总共移动60个柱面。在该例中，电梯算法比SSF算法稍微好一点，尽管通常它不如SSF。电梯算法的一个优良特性是对任意的一组请求，移动磁盘臂次数的上界是固定的：正好是柱面数的两倍。

对这个算法进行略微改进可以进一步减小响应时间，方法是总是按一个方向移动磁盘臂，处理完编号最高柱面上的请求后，磁盘臂移动到具有读写请求的编号最低的柱面，然后继续向上移动。这实际上等于将最低柱面看作是最高柱面之上的相邻柱面。

一些磁盘控制器提供了供软件检查当前磁头下方扇区号的方法。对于这种磁盘控制器，还可进行另一种优化。如果有两个或多个读写同一柱面的请求正在等待处理，驱动程序可以请求读写下一个要通过磁头的扇区。注意当一个柱面有多条磁道时，因为选择磁头既不需要移动磁盘臂也没有旋转延迟，控制器可以快速选择任意磁头，所以连续的读写请求可以申请不同的磁道。

图3－21 调度磁盘请求的电梯算法。

对于现代的硬盘，其数据传输率比软盘快得多，所以必须采用某种自动高速缓冲机制。典型地，任何读一个扇区的请求将使得这个扇区和同一磁道上的其他扇区都被读进来，读进多少扇区取决于控制器中有多少可用的高速缓冲空间。图3－19中描述的540M磁盘具有64K或 128K的高速缓冲空间。磁盘控制器动态地决定高速缓冲的使用。在最简单的模式中，缓冲空间被分成两部分：一部分用于读，另一部分用于写。

当有多个驱动器时，每个驱动器都有一个单独的请求等待表。一旦任何一个驱动器空闲，则启动一个寻道请求将磁盘臂移到下一个请求的柱面（假设控制器允许重叠寻道）。当前数据传输结束后，将检查是否有一驱动器位于正确的柱面上，如果存在一个或多个这样的驱动器，则启动下一个数据传输。如果没有，则驱动程序对刚结束数据传输操作的驱动器发出新的寻道命令并等待，直到下一次中断到来时检查哪一个磁盘臂首先到达了目标位置。

出错处理

RAM盘不需要考虑寻道和旋转优化：在任意时刻，不做任何物理移动就可以读写所有的数据块。RAM盘比实际磁盘简单的另一个地方是出错处理。RAM盘永远正常工作，而实际磁盘却不能，它们可能会出现各种各样的错误。常见的错误有：

1 程序性错误（例如请求读写不存在的扇区）。

2 暂时性校验和错（例如由磁头上的灰尘引起）。

3 永久性校验和错（例如磁盘块的物理损坏）。

4 寻道出错（例如磁盘臂应定位在第6柱面，但却到了第7柱面）。

5 控制器错（例如控制器拒绝接受命令）。

磁盘驱动程序应尽可能地处理这些错误。

当驱动程序通知控制器对不存在的扇区进行寻道、使用不存在的磁头、或者对不存在的存储器传送数据时，将发生程序性错误。多数控制器检查发送给它们的参数，并在参数非法时给出信息。理论上这些错误不会发生，但是当控制器指出发生了这些错误时，驱动程序如何处理呢 ？对于家用系统，最好的办法是停止运行，打印一条类似“求助于程序员”的消息，以便对错误进行跟踪并加以修正。对于一个在世界上数以千计的地方使用的商用软件产品，这种方法并不合适，也许唯一能做的是将当前磁盘请求以出错结束，并且希望这种错误不会发生得太频繁。

暂时性校验和错因空气中的灰尘进入盘面和磁头之间引起。多数情况下，可以通过重复操作几次来消除这种错误。如果它一直存在，则只能将相应的数据块标志为坏块（bad block），并且不再使用。

避免坏块的一种方法是写一个非常特殊的程序，它将坏块清单作为输入，并小心地创建一个文件，使之包含所有的坏块。一旦这个文件创建完毕，磁盘分配程序就认为这些数据块已被占用并永远不会把它们分配出去。只要不去读坏块文件，就不会发生任何问题。

永远不读坏块文件说起来容易做起来难。许多磁盘采用一次将一条磁道的内容拷贝到后备磁带或磁盘上的方法进行备份。如果使用这种方式，则坏块就会引起麻烦。如果后备程序知道坏块文件的文件名并且不拷贝它，那么按一次拷贝一个文件的方式虽然要慢一些，但可以解决这个问题。

使用坏块文件不能解决的另一个问题是：如何处理必须位于磁盘中固定位置的文件系统数据结构中存在的坏块。几乎每一文件系统至少都有一个数据结构的位置必须是固定的，以便很容易地找到它。在一个可分区的文件系统中，通过重新分区可以避开坏道，但软盘或硬盘开始几个扇区中的永久性错误一般意味着整个磁盘无法使用。

“智能”控制器保留了几条用户程序一般不能使用的磁道。当格式化磁盘的时候，控制器可以确定哪些是坏块，并用保留的磁道代替损坏的磁道。存放坏道和保留磁道之间对应关系的表格保存在控制器的内部存储器和磁盘中。这种替换对驱动程序透明，只是对于精心设计的电梯算法，如果每次请求读写柱面3时，控制器实际使用的是柱面800，那么其性能会大大降低。磁盘记录表面的制造技术已经比过去有所提高，但还不是尽善尽美，不过对用户隐藏缺陷的技术也得到了提高。对于图3－19中所示的硬盘，控制器也管理在使用过程中发生的新错误，并在错误不能恢复时永久地分配一个替换块。对于这样的磁盘，驱动程序软件很少会感觉到坏块的存在。

寻道错由磁盘臂的机械问题引起。控制器在内部跟踪磁盘臂的位置。为了寻道，控制器向磁盘臂电机发出一系列脉冲，每个脉冲移动一个柱面，从而将其移动到新的柱面。当磁盘臂到达目标柱面后，控制器读取实际的柱面号（在驱动器格式化时被写上去），如果磁盘臂处于不正确的位置，就发生寻道错。

大多数的硬盘自动纠正寻道错，但许多软盘控制器（包括IBM — PC）仅设置一个出错位，而把其他工作留给驱动程序。驱动程序处理这个错误的方法是发出一条RECALIBRATE命令，把磁盘臂移到最外面，并将使控制器内部将当前柱面号复位为0，如果不能解决问题，驱动器就必须进行修理了。

正如我们看到的：控制器实际上是一个专用的小计算机，它有完备的软件、变量、缓冲区，偶而还出现故障。有时一个非常的事件序列，例如一个驱动器上的中断和另一个驱动器上的RECALIBRATE命令同时发生可能引发一个故障，致使控制器陷入一个循环或无法接续正在做的工作。控制器设计者通常考虑到最坏的情况，在芯片上提供了一个引脚，当触发该引脚时，强迫控制器忘记当前的工作并自行复位。如果其他努力都无法奏效，磁盘驱动程序可以设置一个比特来触发该信号对控制器复位。如果还不成功，那么驱动程序就只能打印出错信

息并放弃。

每次一道（Track at a time）缓冲

对一个新柱面寻道所花的时间通常比旋转延迟大得多，而且往往比传输时间长很多，也就是说，一旦驱动程序需要把磁盘臂移到某个位置，那么读一个扇区还是读一条磁道都差不多。特别是对于控制器提供旋转检测（rotational sensing）的情况，这种效应就更加明显。因为这时驱动程序能够知道哪个扇区正位于磁头下面，并启动一个对下一扇区的请求，从而使得在一次旋转中读一条磁道成为可能（通常平均花旋转半周的时间加上一个扇区的时间仅仅读一个扇区）。

有些磁盘驱动程序通过维护一个秘密的每次一道的高速缓冲来充分利用这一特性，并且不被独立于设备的软件所感知。如果需要的扇区位于高速缓冲里，则不需要磁盘数据传输。每次一道缓冲的一个缺点（除了软件复杂性和需要缓冲空间）是：从高速缓冲到调用程序之间的数据传输必须由CPU使用一个循环完成，而不是让DMA硬件来做这个工作。

一些控制器将这个过程更进一步，在它们自己内部的存储器中实现每次一道缓冲，而对驱动程序透明，这样在控制器和存储器之间的数据传输就能使用DMA。如果控制器按这种方式工作，就没必要让磁盘驱动程序也做这项工作。注意在控制器和驱动程序中由一个命令读写整条磁道是合适的，但不能放在设备无关软件中，因为它将磁盘看成数据块的线性序列，而不考虑它如何划分成磁道和柱面。

3.7.3 MINIX中的硬盘驱动程序概述

硬盘驱动程序是我们已经看到MINIX不得不处理各种类型硬件的第一部分。在开始讨论驱动程序细节以前，我们简单考虑一下一些硬件的不同可能引起的问题。“IBM— PC”的的确确是一个不同计算机的家族。在不同的家族成员中不仅使用的处理器不同，而且在基本硬件上也有一些很大的差别。家族中最早的成员 — 最初的PC 和PC－XT 使用8位总线，与8088处理器8位的外部接口相配合。下一代 PC－AT 使用16位总线，该总线设计得很巧妙，先前的8位外部设备还可使用，但新的16位的外部设备一般不能用在老的PC－XT系统中。AT总线原先是为使用80286处理器的系统设计，并且许多基于 80386、80486 和奔腾的系统也使用AT总线。但因为这些新的处理器具有32位接口，所以，现在有几种不同的32位总线系统，例如 Intel的PCI总线。

对每一类总线，都有一个家族的I/O适配器（I/O adapters）供插到系统主板上。为某一类总线设计的所有外设都必须与标准兼容，但没有必要和以前的总线兼容。和其他的计算机系统家族一样，在IBM－PC家族中，与总线设计同时配套的还有基本I/O系统（BIOS） ROM中的固件，其目的是在操作系统和硬件特殊性之间建立一个联结二者的桥梁，有些外设在自己的外设板上提供扩充BIOS的ROM芯片。操作系统的实现者面临的困难是：在IBM类型的计算机（当然指早期的）中的BIOS是为MS－DOS操作系统设计的，不支持多道程序设计，运行于16位实模式，这是80X86 CPU家族可以使用的各种操作模式中最普通的一种模式。

于是，为IBM— PC设计新操作系统的实现者面临几个选择：首先，是使用BIOS中的设备驱动程序支持还是重头编写新的驱动程序。由于BIOS在许多方面不适于MINIX的需要，所以在MINIX的初始设计中作出选择并不困难。当然，为了启动，MINIX的引导监控程序仍要使用BIOS实现系统的初始装载 — 无论是从硬盘还是从软盘，实际上这是无法选择的。一旦系统被装入，包括我们的设备驱动程序，我们就可以比BIOS做得更好。

面临的第二项选择是：没有BIOS的支持，我们如何使我们的设备驱动程序适用于不同系统中的各种硬件。为了使讨论更具体，考虑存在至少四个基本类型的硬盘控制器，这些类型我们可以在一个适用于MINIX的系统上找到：最初的8位XT类型控制器、16位AT类型控制器，以及IBM PS/2系列计算机中两种不同类型的控制器。有几种不同的方法解决这个问题。

1 为需要支持的每种硬盘控制器重新编译一个操作系统版本。

2 在核心中编译几个不同的硬盘驱动程序，由核心在启动时自动决定使用哪一个。

3 在核心中编译几个不同的硬盘驱动程序，提供一种方法使用户决定使用哪一个。

正如我们将看到的，这些方法并不相互排斥。

第一种方法从长远来看是最好的。对于使用特定配置的系统，没有必要为从来不使用的驱动程序浪费磁盘和内存空间，然而，对软件发行者而言这却无法接受。准备四套不同的启动盘，并告诉用户如何使用这些盘片非常昂贵也十分困难。因此，其他方案是值得考虑的，至少对于初始安装而言是这样。

第二种方法是由操作系统检测外部设备，通过读写每块卡上的ROM或者写和读I/O端口来识别每一块卡。这在一些系统中可行，但是在IBM类型的系统中却工作得不是很好，因为存在太多的非标准I/O设备。有些情况下，读写I/O端口来对设备进行识别可能激活其他的设备，使其获得控制权而导致系统无法工作，这种方法也使每个设备的启动代码复杂化，并且还是不能工作得很好。使用这种方法的操作系统一般必须提供某种重载机制，典型的如我们在MINIX中使用的机制。

第三种办法是MINIX使用的方法，它允许编译多个驱动程序，其中一个是缺省驱动程序。MINIX引导监控程序允许在启动时读各种引导参数（boot parameters），它们可以手工输入，也可以永久存放于磁盘上。在启动时如果发现引导参数格式为：

```
hd=xt
```

则强制使用XT硬盘驱动程序，如果没有发现hd引导参数，使用缺省驱动程序。

为了减少支持多个硬盘驱动程序所导致的问题，MINIX还做了其他两件事。第一是提供了一个支持在MINIX

和ROM BIOS硬盘之间接口的驱动程序，这个驱动程序几乎可以保证在所有的系统下都可以工作，通过使用启动参数

hd=bios

来选择，不过一般这是可求助的最后一种办法。在80286或更高级的处理器上，MINIX运行于保护模式，但是BIOS永远运行于实模式（8086）。无论何时当BIOS中的例程被调用时，切换出保护模式，重新回到先前模式是很慢的。

MINIX处理驱动程序的另一策略是尽可能推迟初始化工作，这样如果在某些硬件配置中没有硬盘驱动程序可用，则仍然可以从软盘启动，完成一些有用的工作。只要不访问硬盘，MINIX不会有任何问题。在用户友好方面这算不上一个大的突破，但是考虑这种情况：由于我们不恰当地配置了一些从不使用的设备，如果系统引导时所有的驱动程序立即初始化，系统将全部处于停滞状态。把设备初始化推迟到需要的时刻，则当用户试图去解决问题时，系统可以使用任何可以工作的部件继续运行。

另一方面，我们在以困难的方式学习这门课：早期的MINIX版本一启动就试图初始化硬盘，如果没有硬盘，系统就挂起。这是很不幸的。但是，虽然存储能力要受限制，性能也要降低，MINIX完全可以在没有硬盘的系统中顺畅地运行。

在本节和下一节的讨论中，我们选择AT类型的硬盘驱动程序为例，在标准的MINIX发布中这是缺省设备驱动程序。这是一个多功能的设备驱动程序，可以处理从早期80286系统中使用的硬盘控制器到能够处理上吉（G）字节存储量的EIDE（Extended Integrated Drive Electronics — 扩展的集成驱动电子线路）控制器。我们在本节所讨论的硬盘操作的一般内容也适用于系统所支持的其他硬盘驱动程序。

硬盘任务的主循环是我们已经讨论过的相同的共享代码，可以执行六种标准请求。因为在硬盘上进行了分区和子分区，所以DEV_OPEN请求需要执行大量的工作。当设备打开时（第一次被访问时），这些必须首先读进来。一些硬盘控制器也支持CD－ROM驱动器，它们具有可移动介质，并且当执行DEV_OPEN时，必须检查介质是否存在。对于CD－ROM，DEV_CLOSE操作还有以下含义：它需要打开光驱弹出光盘。可移动介质的其他复杂性在软驱中更普遍，所以将在下一节讨论。对于硬盘，DEV_IOCTL操作设置当执行DEV_CLOSE时介质应被弹出的标志。这个特性对CD－ROM也是有用的。正如我们曾经提到过的，它也可以用来读写分区表。

象先前看到的那样，每一个DEV_READ 请求、DEV_WRITE请求和SCATTERED_IO请求都分成三个阶段来处理：准备、调度和结束。和存储器设备不同，在硬盘的调度阶段和完成阶段之间有明确的界限。硬盘驱动程序不使用SSF算法或电梯算法。但进行了一种更为有效的调度：把对连续扇区的请求集中起来。请求一般来源于MINIX文件系统，并且请求许多块，每块1024字节，但驱动程序也能够处理512字节的扇区的任意倍的请求。如果请求的扇区紧挨着前一次请求的扇区，则请求就被加入到一个请求链表中。这个表被组织成一个数组，当它满的时候或者请求的是一个不连续的扇区的时候，就调用结束例程。

在一个简单的DEV_READ和DEV_WRITE请求中，可能会请求不止一块，但是每一个调度例程的调用紧接着一个结束例程的调用，以保证满足当前的请求列表。对于SCATTERED_IO请求，在调用结束例程以前，可能会多次调用调度例程。 如果是对连续的数据块的请求，请求链会扩展到数组满为止。回想在SCATTERED_IO请求中，有一个标志标识对一个特殊块的请求是可选的。硬盘驱动程序象存储器驱动程序一样忽略OPTIONAL标志，传输所有请求的数据。

硬盘设备驱动程序执行的基本调度和当请求连续的数据块时推迟数据传输应被看成是三步调度过程中的第二步。文件系统本身通过使用SCATTERED_IO可以实现类似Teory版的电梯算法。回想在SCATTERED_IO请求中，请求链表按块号排序。第三步调度发生在类似图3－19中描述的现代硬盘控制器中。这些控制器具有“智能”，能够缓冲大量的数据，使用内部的程序算法来按最有效的顺序获取数据，而不是按照请求的到达次序。

3.7.4 MINIX中的硬盘驱动程序实现

微机上使用的小型硬盘有时称为“温彻斯特”硬盘。关于这个名字的来源有几个不同的故事。很明显它是IBM一个开发磁盘技术工程的代号，在这项工程中，读/写磁头飞行于薄薄的空气垫上，当磁盘停止旋转时落在记录介质上。这个名字的一种解释是一个早期的型号有两个数据模块：30M字节固定硬盘和30M字节可移动硬盘。可以设想它使开发人员想起了温彻斯特30 — 30 火枪 — 一个在美国西部传说中的角色。但无论名字来源何处，其基本技术是一样的，尽管今天典型的微计算机上的磁盘比起14英寸磁盘体积要小得多，容量要大的多 — 14英寸磁盘是七十年代开发“温彻斯特”技术时典型的磁盘。

文件wini.c用于对内核的其他部分隐藏实际的硬盘驱动程序。这使我们能够实现前一节讨论的策略：即在一个内核映象中编译几个硬盘驱动程序，并在启动时选择一个。以后可以重新编译用户安装程序使得它仅包含一个用户需要的设备驱动程序。

wini.c包含一个数据定义，hdmap（10013行），它是一个使名字和函数地址相联系的数组。编译程序对该地址初始化，使得需要多少个硬盘驱动程序数组中就有多少项，这标识在文件include/minix/config.h中。函数winchester_task使用该数组，winchester_task是一个在task_tab表中的名字，当内核第一次初始化时使用这个函数。当winchester_task（10040行）函数被调用时，它使用和普通C程序机制相似的内核函数试图寻找一个hd环境变量，读MINIX启动监视程序创建的环境变量，如果hd值没有定义，就使用数组中的第一个元素。否则，在数组中寻找匹配名字，然后间接调用相应的函数。在本节余下的部分，我们将讨论at_winchester_task，它

是发布的标准MINIX中hdmap数组中的第一个元素。

AT风格的驱动程序在at_wini.c（10100行）中，这是一个面向高级设备的复杂驱动程序，有好几页宏定义了控制器寄存器、状态位和命令、数据结构和原型。象其他的块设备驱动程序一样，一个driver结构，w_tab（10274行 到 10284行）被初始化为指向实际完成这项工作的函数指针。这些指针大多数定义在at_wini.c中，但是由于硬盘不需要特殊的清除操作，所以dr_cleanup指向了普通的在driver.c中的nop_cleaanup，和其他不需要特殊清除操作的设备驱动程序共享了该函数。入口函数at_winchester_task（10294行）调用一个由硬件决定的初始化过程，然后调用driver.c中的主循环。这个循环永远运行，把调用分派到driver表中指向的各种函数。

因为我们正在处理实际的机械电子存储设备，所以要做一定量的工作来初始化硬盘驱动器。有关硬盘的各种参数保存于定义在10214行至10230行的数组wini中，作为推迟初始化策略的一部分，由于在必须使用设备以前对设备初始化可能会失败，所以在内核初始化时调用的init_params（10307行）并不做任何访问磁盘的工作。它做的主要工作是把有关磁盘的逻辑配置信息拷贝到wini数组中。这些信息是ROM BIOS从CMOS存储器中提取的，“奔腾”类计算机在这些存储器中存放配置信息。当机器第一次接通电源时，在MINIX第一部分的装载过程开始以前，执行BIOS中的功能。取不出这项信息并不是致命的，如果磁盘是现代化的磁盘，该信息可以从磁盘直接得到。

调用通用的主循环以后，直到试图访问硬盘以前，不做任何事情。然后，收到一个请求DEV_OPEN操作的消息，再间接调用w_do_open函数（10355行）。w_do_open调用w_prepare来确定请求的设备是否合法，然后调用w_identify来确定设备的类型，并初始化在数组wini中的其他一些参数。最后，使用在数组wini中的计数器检测从MINIX启动以来是否第一次打开设备。检测完毕后，计数器加一。如果是第一次DEV_OPEN操作，就调用partition函数。

下一个函数w_prepare（10388行）接收一个设备的次设备号或使用分区的整型参数device，并返回一个指向device结构的指针，指出设备的基址和大小。在C中，使用一个标识符来命名一个结构并不妨碍使用相同的名字来命名一个变量。一个设备是驱动器分区还是子分区可以由次设备号确定。一旦w_prepare完成了它的工作，其他任何读写磁盘的函数都不需关心它们和分区的关系。就象我们看到的那样，当执行DEV _OPEN请求时，将调用w_prepare，该函数也是所有数据传输中使用的准备/调度/完成循环的一个步骤。在这里，把w_count初始化为零是很重要的。

各种软件兼容的AT型磁盘已经使用了很长一段时间，w_identify（10415行）函数不得不把这段时间内引入的各种不同的设计区分开来。第一步是首先检查在所有这一类磁盘控制器都应有的一个地址是否存在一个可读写的端口（10435至10437行）。如果这个条件满足，在中断描述表中安装硬盘中断处理程序地址并开放中断控制器以响应中断。然后向硬盘控制器发ATA_IDENTIFY命令。如果返回结果为OK，就取出各项信息，包括一个标识磁盘模型的字符串以及设备的物理柱面、磁头和扇区参数（报告的“物理”配置可能并不是真正的物理配置，但我们只能接受驱动器报告的参数）。磁盘信息也指出了磁盘是否允许进行线性块访问（Linear Block Addressing — LBA）。如果能，则驱动程序可以忽略柱面、磁头和扇区参数，使用绝对的扇区号访问磁盘从而简化访问磁盘操作。

我们以前已经提到，init_params函数有可能没有从BIOS表中复原逻辑磁盘配置信息，如果确实如此，那么在10469至10477行的代码将试图根据它从驱动器本身读的参数来创建一合适的参数集，其思路是根据原始BIOS数据结构中对应域所允许的位数，把最大的柱面、磁头和扇区数分别取值为 1023、255和63。

如果ATA_IDENTIFY命令失败，可能仅仅意味着磁盘为过时的模型，不支持这个命令。在这种情况下，我们仅仅可以得到以前init_params所读出的逻辑配置值。如果这些值有效，则把它们拷贝到wini中的物理参数域，否则返回出错，磁盘不能使用。

最后，MINIX使用一个u32_t变量来对地址按字节计数。如果柱面数×磁头数×扇区数的计算结果太大的话（10490行），那么必须限制以扇区数表示的驱动程序所能处理的设备的大小。虽然在编写这段程序代码时，在有可能会使用MINIX的机器上，不大可能出现容量超过4G的设备，但经验告诉我们，应该编写能经受这些极限参数测试的软件，而不仅仅是能经受编写软件时设备参数的测试。整个设备驱动程序的基址和大小存放在wini数组中。并调用w_specify函数来向磁盘控制器传递参数，如果有必要可以再调用一次。最后，在控制台上打印设备的名字（由w_name确定）和由w_identify确定的标识符串（如果是现代设备的话）或者打印BIOS报告的柱面磁头扇区数（如果是以前的设备的话）。

w_name（10511行）返回一个指向设备名的字符串指针，它们是“at_hd0”、“at_hd10”或“at_hd15″之一。w_specify（10531行）除了向控制器传参数外，还通过对零柱面寻道来重新校准驱动器（如果是以前的设备的话）。

现在，我们将讨论在完成数据传输请求时被调用的函数。首先调用的是w_prepare，这个函数在前面讨论过，它把变量w_count初始化为零工作是很重要的。在数据传输过程中调用的下一个函数是w_schedule（10567行），它建立起基本参数：包括数据从哪里来、数据到哪里去、传输数据的字节数（必须为扇区大小的倍数，在10584行进行这项检查）、传输数据是读数据还是写数据。在SCATTERED_IO请求中指示的可选数据传输位在向控制器传输的操作中复位（10595行），但是注意该位还被保留在iorequest_s结构的io_request域中。对于硬

盘，驱动程序将试图满足所有的请求，但是就象我们看到的那样，如果出错的话，驱动程序可能会决定不这样做。建立基本参数的最后一件事是检查请求的范围是否超出了设备上的最后一个字节，如果超出的话，就减小请求传输的数据量。这里可以计算出所要读的第一个扇区。

从10602行调度过程开始，如果有等待的请求（通过测试w_count大于零来检测）而且如果下一个要读的扇区和上一个请求的扇区是不连续的，则调用w_finish函数来结束前一个操作。否则，更新w_nextblock变量，在这个变量中存放的是下一个扇区号。执行10611行至10640行的循环把扇区请求加入请求数组中。直到在允许的最大范围以内，到达了允许的请求数目（10614行），该范围保存在变max_count中，我们以后将看到，这样做对我们能够修改该范围是有帮助的。这里再一次可能调用w_finish。

我们已经看到，在函数w_prepare中，有两个地方调用了w_finish。通常w_prepare没有调用w_finish就终止了。但是无论是否在w_prepare中被调用，w_finish（10649行）都要在文件driver.c中的主循环中被调用。如果它已被调用过，那末它什么也不做。所以，在　　10659行需检测这种情况。如果在请求数组中还有请求，就进入w_finish的主要部分。

就象我们所估计的那样，可能会有许多请求，所以w_finish的主要部分是一个在10664和10761行之间的循环。在进入主循环以前，通过预先设置变量r为出错来实施强制重新复位控制器。如果对函数w_specify的调用成功地完成了command结构中的命令，就初始化cdm来执行数据传输。command结构用来把所有需要的参数传给实际控制磁盘控制器的函数。一些控制器使用cdm.precomp来补偿当磁头从外柱面向内柱面移动时，由于磁头下面介质移动的速度不同而引起的磁记录介质性能的不同。对于特定的驱动器，它是永远不变的，许多驱动器都忽略该参数。cdm.count 接收传输的扇区数，使其为8位字节的整数倍，因为驱动器的所有命令和状态状态寄存器都是8位字节。在10675行至10689行之间的代码指定传输的第一个扇区，或者以28位逻辑块号的形式表示（10676行至此0679行），或者以柱面、磁头和扇区的形式表示（10681行至此0688行），在这两种情况下都使用了cdm结构中的同一域。

最后，装入命令本身，并在10692行调用com_out初始化数据传输。如果控制器没有准备好或者在预先设置的时间范围内没有准备好，那么调用com_out可能会失败。在这种情况下，出错计数加一。如果出错计数到达了MAX_ERRORS，则放弃初始化工作，否则在10697行的continue语句使循环从10665行重新执行。

如果控制器接收了在调用com_out时传递的命令，可能过一段时间才能得到数据，因此（假定命令为DEV_READ)在10706行调用了w_intr_wait。以后我们将详细讨论这个函数,但是现在仅说明一下它调用了receive函数，这样磁盘任务将被封锁。

一段时间以后，对w_intr_wait的调用返回，时间的长短取决于是否涉及到寻道。虽然有些控制器支持了DMA，但这个驱动程序并未使用，而是使用了程序I/O方式。如果w_intr_wait正确返回而且没有出错，汇编语言函数port_read将从控制器的数据端口传输SECTOR_SIZE字节的数据到目的地址，该地址位于文件系统块缓冲区中。然后根据成功的数据传输调整各种地址和计数以记录该次成功的数据传输。最后，如果当前请求的字节计数减少为零，就把指向请求数组的指针向下移动以指向下一个请求（10714行）。

在DEV_WRITE命令的情况下，第一部分设置命令参数并把命令送入控制器。在命令参数中除了命令操作码以外和读操作是一样的。但对写操作而言，后续事件的顺序是不同的。首先要等待控制器发信号通知它准备好接收数据（10724行）。wait_for是一个宏，一般很快即可返回。我们在后面还要进一步讨论它。现在我们仅仅说明等待最终会超时，但我们希望很少有长时间的超时。然后使用port_write函数（10729行）把数据由存储器传输到控制器的数据端口，这时要调用w_intr_wait函数，阻塞磁盘任务。当中断到来时，磁盘任务被唤醒并记录薄记（10736至10739行）。

最后，如果在读写时出错，必须对出错进行处理。如果控制器告诉驱动程序由于扇区损坏而出错，那么就没有必要重试了。但对其他类型的出错至少在某种程度上值得重试一次。这里所说的程度由出错计数确定，如果出错计数达到了MAX_ERROR就放弃重试。当达到MAX_ERROR/2时，调用w_need_reset实现当重试时强制重新初始化。不过，如果请求是可选的（由SCATTERED_IO请求确定），不进行重试。

无论w_finish无错终止还是出错终止，变量w_command总要设置为CMD_IDLE。这使得其他的函数可以确定不是因为在试图完成某个操作后由于机械或电子的故障引起出错而产生了一个中断。

通过一组寄存器控制磁盘控制器，在某些系统中，这组寄存器可以被映射为一段存储区，但在IBM兼容的机器中为I/O端口。标准IBM－AT类的硬盘控制器使用的寄存器示于图3－22中。

这是我们第一次遇到I/O硬件，说明一下I/O端口和存储器地址的几点不同对我们是有帮助的。一般地，具有相同I/O地址的输入输出端口并不是相同的寄存器，因此写入某一特定地址的数据不能在以后由读操作取出来。例如，对于图3－22中的最后一个寄存器地址，当对其读时为磁盘控制器状态，当对其写时为向磁盘控制器发命令。一般读写I/O设备寄存器会引起一个独立于数据传输细节动作的发生。AT磁盘控制器中的命令寄存器就是这样的。在使用时，通过把数据写入低编号的寄存器来选择读出或写入的磁盘地址，然后把操作码写入命令寄存器。把操作码写入命令寄存器将启动具体的操作。

图3－22（a）IDE硬盘控制器的控制寄存器。括号中的数是在LBA模式下每一个寄存器选择的逻辑块地址位。（b）驱动器/磁头 寄存器的选择域。

也有这种情况，对于不同的操作模式，寄存器或寄存器中的某些域的使用是不同的。在表中给出的例子中，

向第六寄存器的第六位LBA位写入0或写入1分别选择使用CHS（柱面－磁头－扇区模式）或LBA（线性块寻址）模式。向寄存器3 、4、 5写入的数据或从寄存器3 、4、 5读出的数据以及第六寄存器的低四位数据由于LBA位的不同其解释是不同的。

现在我们通过调用com_out（10771行）来研究以下命令是如何发送到控制器的。在改变任何寄存器的内容以前，通过读状态寄存器来确定控制器不忙。这个工作是通过检查STATUS_BSY位而完成的。在这里速度是重要的。一般磁盘控制器总是准备好的或在很短的时间内即可准备好，所以，采用了忙等待的方法。在10779行调用了waitfor来测试STATUS位，为了提高响应速度，waitfor是一个宏，它在10268行定义。它使所需的测试只执行一次，以避免当磁盘准备好时调用函数产生的开销。在极少数情况下，当需要等待时，就调用w_waitfor，w_waitfor执行循环测试直至条件为真或者预定义的超时时间到，因此，如果控制器准备好，那么将会在最短时间内返回真。如果是暂时性失效，则经过一段时间的延迟后返回真，如果在超过限定时间后还没有准备好，则返回假。当讨论w_waitfor时，我们将进一步讨论超时问题。

一个控制器可以控制多个驱动器，所以一旦确定控制器准备好，通过向控制器写一个字节来选择驱动器、磁头和操作模式（10785行），然后再次调用waitfor。磁盘驱动器执行命令时，有时会失败或不能正常地返回一个出错代码。毕竟驱动器是机械设备，内部有可能发生各种机械故障。所以作为一项保险措施，要向时钟任务发送一条消息以安排一个对唤醒例程的调用。然后通过首先向各种寄存器写入参数再向命令寄存器写入命令代码来发出命令。下一步和接下来的对变量w_command 和w_status的修改部分为临界区，因此通过调用lock 和unlock（10801行至10803行）把这个执行序列括起来，这两个函数用于禁止和打开中断。

下面几个函数很短。我们注意到，当出错计数计到MAX_ERRORS的一半时，w_finish函数调用了w_need_reset函数（10813行）。当等待磁盘中断或准备好时，如果发生超时，也将调用它。w_need_reset的工作仅仅是对wini数组中每个驱动器的state变量做标志，使得下一次访问时强制进行初始化。

w_do_close（10828行）对常规的硬盘几乎不作任何工作。当支持的设备为CD_ROM或其他可移动的设备时，这个例程需要扩展为产生一个命令以打开驱动器或弹出CD，具体如何执行取决于硬件支持。

调用com_simple来向控制器发出不需要传输数据，立即就会结束的一些命令。属于这一类的命令包括取磁盘标识，设置一些参数和重新校准。

当com_out调用时钟任务准备在磁盘控制器失败以后恢复程序运行时，它使用的一个参数是w_timeout（10858行）的地址，为当超时发生时，时钟任务将唤醒该函数。但是一般情况下磁盘将完成所请求的操作，于是当超时发生时，将发现w_command的值为CMP_IDLE，这意味着磁盘完成了它的操作，于是w_timeout终止。如果命令没有完成，并且操作是读请求或是写请求，那么减小I/O请求的大小可能会有帮助。这项工作分两步完成，首先把可以请求的最大扇区数目减小为8，再减小到1，对于所有的超时，打印一条消息，在下一次试图访问磁盘时，调用w_need_reset强制重新初始化所有的驱动器，调用interrupt向磁盘任务发一条消息，并模拟产生一个应该在磁盘操作结束时发生的硬件中断。

当需要复位时，就调用w_reset（10889行）。这个函数利用时钟驱动程序提供的函数milli_delay。经过使驱动器从以前的操作中恢复过来的初始延迟以后，选通磁盘控制器中的某一位，也就是把它提到逻辑电平1一段时间，然后恢复为逻辑电平0。执行这个操作以后，调用w_waitfor给驱动器一段合理的时间以便发信号使其准备好。如果复位不成功，就打印一条消息，返回出错状态，下一步如何做将由调用者决定。

发往磁盘的有关数据传输的命令一般通过产生一个中断而终止，该中断向磁盘任务发送一消息。事实上，当每一个扇区被读出或写入时，都产生一个中断，因此发出一个命令以后，总要调用w_intr_wait（10925行）。接着，w_intr_wait在一个循环中调用receive，忽略掉每条消息的内容，等待一个把w_status设置为非忙的中断。如果接收到了一个这样的消息，就检查请求的状态，这也是一个临界区，因此，调用lock 和unlock来保证不发生新中断，也防止了在相关的每一修改步骤完成以前对status的修改。

我们已经在几个地方见到调用宏waitfor来在磁盘控制器状态寄存器的某一位上执行忙等待。经过初始测试以后，waitfor宏调用w_waitfor（10955行），w_waitfor调用milli_start来启动一个定时器，然后进入一个循环，轮流检测状态寄存器和定时器。如果发生超时，则调用w_need_reset，它设置标志表示下一次请求磁盘服务时，需对磁盘控制器进行复位操作。

w_waitfor使用的TIMEROUT参数在10206行定义为32秒。另一个相似的参数是WAKEUP，其值取为31秒（10193行），时钟任务用它安排唤醒事件。考虑到普通进程在其被迫放弃CPU以前仅仅可以运行100ms，那么这些参数对于忙等待而言，是很长的一段时间，但是，这些数值是基于已公布的AT类计算机硬盘接口标准的，这些标准指出了磁盘旋转到一定的速度所允许的最长时间为31秒，当然实际上，这是最坏情况下的规范，在大多数的系统中，仅仅在刚加电时或在很长时间不活动以后，才需要启动旋转加速。MINIX处于发展之中，当增加支持CD_ROM或其他经常旋转的设备时，可以采用新的处理超时的方法。

w_handler（10976行）是一个中断处理程序。当硬盘任务第一次被激活时，w_identify把这个中断处理程序的地址送入中断描述表中。当磁盘中断发生时，磁盘控制器的状态寄存器被拷贝到w_status中，然后调用内核中的interrupt函数，它重新调度硬盘任务。当这些发生时，由于初始化硬盘操作后，w_intr_wait调用了receive，所以硬盘任务肯定处于阻塞状态。

at_wini.c中的最后一个函数是w_geometry，它返回被选中的硬盘设备的最大柱面数、磁头数和扇区数。

其值为实际值，而不是象RAM盘驱动程序那样构造出来的值。

3.7.5软盘处理

和硬盘驱动程序相比，软盘驱动程序更长，也更复杂。因为软盘的结构和硬盘相比要简单，所以这好象有点不合情理。但是，简单的机构具有一个简单的控制器，因此，操作系统就必须考虑更多的内容。可移动介质这一事实也增加了复杂性。在这一节，我们讨论一些在处理软盘时必须考虑的问题。不过我们将不讨论MINIX软盘驱动器的细节，其重要的部分和硬盘是相似的。

对于软盘驱动程序，我们不必关心的一件事是支持多种控制器类型，而对硬盘，我们是不得不处理的。虽然，初始的IBM— PC不支持当前使用的高密度软盘，但是单一的软件驱动程序可以支持IBM家族中各种计算机上的软盘控制器。这种和硬盘形成鲜明对照的情形可能因为对于软盘而言，没有象硬盘那样提高性能的压力。在计算机系统中，软盘很少作为工作介质，和硬盘相比，其速度和容量太有限了。不过对于新软件的发布和后备，软盘还是很重要的，因此几乎所有的小计算机系统都配备至少一个软驱。

软盘驱动程序不使用SSF算法或电梯算法，它是严格顺序执行的。在接收另一个请求以前，要完成已经接收的请求。在初始MINIX的设计中，由于MINIX是为个人计算机而设计的，大多数时间里仅仅有一个活跃的进程，所以设计者感到一个磁盘请求正在处理时，到达另一个磁盘请求的可能性很小。把各个请求排成一个队列需要显著地增加软件复杂性，但又不会带来什么益处。现在，软盘除了被配有硬盘的系统用来传入传出数据之外很少使用，这使其益处进一步降低。

这就是说，即使在驱动程序软件中没有对请求重新排序的支持，软盘和其他块设备一样也能处理分散I/O请求。而且和硬盘驱动程序一样，软盘驱动程序搜集放在一个数组中的请求，当请求顺序扇区时，就继续搜集这些请求。然而，对软盘驱动程序而言，请求数组比硬盘驱动程序要小，最大值为软盘机上每条磁道的扇区数。另外软盘驱动程序在处理分散I/O时要考虑OPTION标志，如果所有当前的请求是可选的，就不会处理一个新的磁道。

软盘驱动器的简单性使得软盘驱动程序复杂化。廉价的、缓慢的、低容量的软盘驱动器不值得配置硬盘所使用的复杂的集成控制器，因此驱动程序软件就不得不处理一些在硬盘中被隐藏于硬盘驱动器的操作。作为一个由软盘驱动器简单性而引起的复杂性的例子，考虑在寻道过程中如何把读写磁头定位到特定的磁道。没有硬盘会要求驱动器软件显式地调用寻道操作。对于硬盘而言，对程序员可见的柱面、磁头、扇区等几何结构和物理几何结构并无对应关系。事实上物理结构是很复杂的。外部柱面比内部柱面要包含更多的扇区，然而，这对于用户而言是不可见的。作为对磁盘按柱面、磁道、扇区寻址的另一种方法，硬盘可以接收按磁盘上绝对扇区号编址的逻辑块地址（LBA）。即使采用柱面、磁道、扇区编址，由于磁盘的集成控制器计算把磁头移到何处，如果需要执行寻道操作，只要不访问任何不存在扇区，就可以使用任何几何结构。

然而对于软盘，寻道（SEEK）操作需要显式地编程，如果寻道失败，必须提供一个例程来执行重校准（RECALIBRATE）操作，强迫磁头回到零柱面。这使得控制器有可能经过一定的步数把磁头移到需要寻道的位置。对于硬盘也要相同的操作，但硬盘控制器执行了这些操作，而不需要驱动程序软件的指导。

使软盘驱动程序复杂化的一些因素是：

1 可移动介质

2 多种磁盘格式

3 电机控制

一些硬盘控制器也支持可移动介质，CD_ROM驱动器就是一个例子，但是即使没有设备驱动程序软件的支持，驱动器控制器一般也能处理一些复杂的问题。然而对于软盘而言没有这些内嵌的支持，实际上软盘更需要这些支持。软盘最常用的用处是安装软件和后备文件，经常需要从驱动器中取出软盘或把软盘插入驱动器中。把想写入某张软盘的数据写入了另一张软盘是令人伤心的。设备驱动程序应尽它最大的努力来防止这种情况的发生，虽然有时这是不可能的。因为并不是所有的驱动器硬件都支持检测从上次访问磁盘以后驱动器是否被打开过。可移动介质可能引起的另一个问题是如果系统试图访问一个没有插入软盘的驱动器，系统可能挂起。如果能检测驱动器门是否打开就可解决这个问题。然而，并不总是能进行这种检测，所以如果对软盘操作不能在一个合理的时间内终止，就必须提供一些措施实施超时处理和返回出错。

可移动介质可以被其他的介质所代替。对于软盘，有许多不同的格式，MINIX支持3.5寸和5.25寸软盘驱动器。软盘可以按多种格式格式化，从360KB到1.2MB（5.25寸软驱）或1.44MB（3.5寸软驱）。MINIX支持七种不同的软盘格式。对于不同格式引起的问题，有两种解决方法，MINIX支持这两种方法。一种方法是把每一种可能的格式看成一个特殊的驱动器，并为设备提供多个次设备号，MINIX就是这样实现的。在设备目录中，你将会发现十四个不同的设备，从第一个驱动器的/dev/pc0、360KB、5.25寸软盘到第二个驱动器的/dev/ps1、1.44MB、3.5寸软盘。要注意各种不同的组合是很累赘的，因此，MINIX提供了第二种方法。当用第一个/dev/fd0或第二个/dev/fd1第一次访问软驱时，软盘驱动程序测试在驱动器中所访问的软盘以确定其格式。一些格式的柱面比较多，另一些格式每道的扇区数比其他的格式多，通过逐步读更大的磁道和扇区数就可确定软盘的格式。通过一个淘汰过程软盘格式即可确定下来。当然这需要时间。如果软盘上存在损坏的扇区还可能出错。

软驱的最后一个复杂问题是电机控制。如果不旋转，软盘就不能读写。硬盘被设计成连续运行几千小时也不会失效，但是电机永远旋转会使软驱和软盘很快失效。如果访问驱动器时电机未开，就需要发出命令启动电

机，然后在等待半秒后再试图读写数据。开关电机是很慢的，所以MINIX每次使用完驱动器后，都继续使电机开几秒钟，如果在这段时间内再次使用，则把定时器再延长几秒。如果在这段时间内没有使用软驱，那么就关闭电机。

3.8 时钟

由于许多原因，时钟（也称为定时器）对于任何分时操作的系统都是很重要的。它维护每天的时间，并且防止某一进程独占CPU而不让其他实体使用CPU。虽然时钟既不是象磁盘一样的块设备，也不是象终端一样的字符设备，但是时钟软件却可以以设备驱动程序的形式工作。我们对时钟的讨论和前面几节一样，首先一般地讨论一下时钟硬件和软件，然后讨论这些想法如何应用于MINIX。

3.8.1 时钟硬件

在计算机中通常使用两种类型的时钟，两者都和人们使用的钟和表有很大的不同。简单的时钟连接到110或220V的电力线上，以50或60HZ的频率在每个电压周期产生一次中断。

另一种时钟如图3－23中所示，由三个元件组成：晶振、计数器和一个保持寄存器。当把石英晶体进行合适的切削并安装于一定的压力之下时，它会产生非常精确的周期信号。根据所选晶体的不同，其典型范围在5至100M之间。在任何计算机系统中，都至少可以发现一个这样的电路向计算机的各种电路提供同步信号。把这个信号送入计数器并使其递减计数至零，当计数至零时，产生一个中断。

典型的可编程时钟有几种操作模式。在单触发模式（One-shot Mode）中，当一个时钟启动时，它把保持寄存器的值拷贝到计数器中，然后每从晶振来一个脉冲，对计数器值减一，当计数器为零时，产生一个中断，并停止工作直到再次被软件启动为止。在方波模式（Square-wave Mode）中，每次计数至零并引起中断以后，保持寄存器自动拷贝到计数器，整个过程不断重复重复执行。这些周期性的中断称为时钟滴答（Clock Tick）。

图3－23 可编程时钟。

可编程时钟的优点是它的中断频率可以由软件控制。如果使用1M的晶体，那么计数器每微秒接收到一个脉冲，对于十六位寄存器，中断可以编程为按1微秒至65536微秒的间隔而发生。可编程时钟芯片一般包含两至三个可编程时钟，并有许多其他的选项（如向上计数还是向下计数，中断屏蔽等）。

为了防止当计算机掉电时丢失当前时间，大多数的计算机有一个电池供电的时钟，采用使用在数字手表中的低功耗电路。可以在启动时读电池时钟，如果不存在后备时钟，软件可以询问用户当前的日期和时间。对于网络系统，存在一个从远程主机获取时间的协议。在UNIX和MINIX中，时间被转换成从通用标准定时（Universal Coordinated Time （UTC））（格林威治平时 — Greenwich Mean Time）1970年1月1日上午12点开始的时钟滴答数，当然也可以采用其他的时间基准。每一次时钟滴答，实际时间加1。一般实用程序可以手工设置系统时钟或后备时钟，并使其同步。

3.8.2 时钟软件

硬件所做的工作仅仅是按已知时间间隔产生中断。其他和时间有关的工作都必须由软件驱动程序来实现。在不同的操作系统中，时钟驱动程序的任务也不同，但一般包括以下内容：

1 维护日期时间。
2 防止进程的运行时间超出其允许的时间。
3 对CPU使用进行记费。
4 处理用户进程提出的时间闹钟系统调用。
5 对系统某些部分提供监视定时器。
6 支持直方图监视和统计信息搜集。

第一项功能维护日期时间（也称为实际时间）并不困难。它仅需要象我们前面提到的那样，每次时钟滴答增加一次计数器。需要注意的是日期时间的位数。对于60HZ的时钟，32位的计数器刚刚超过两年就会溢出。显然系统无法用32位存储1970年1月1日开始的滴答数来存储实际时间。

有三个方法解决这个问题，第一种方法是使用64位计数器。因为1秒以内需执行多次维护计数器的工作，因此提高了维护时钟的代价。第二种办法使用一个辅助计数器来对滴答计数直至累计一秒为止，因为2 32秒超过136年，这种方法直至22世纪都会工作得很好。

第三种方法按滴答计数，但是相对系统启动时间，而不是相对一个确定的外部时刻。当读后备时钟或用户输入实际时间时，根据当前的时间计算系统启动时间并以一种方便的格式存储于系统中。以后当询问每天的时间时，存储的时间加上计数器中的时间就是当前时间。所有这三种方法都显示在图3－24中。

图3－24维护每天时间的三种方法。

时钟的第二项功能是防止进程运行太长的时间。无论何时启动一个进程，调度程序都应该用一个以时钟滴答计算的进程时间片的值初始化一个计数器。当每次时钟中断时，中断驱动程序把时间片计数器减一，当它为零时，时钟驱动程序调用调度程序来建立其他的进程。

第三项功能为CPU记帐。完成这项工作的最方便的方法是无论何时启动一个进程就启动一个和主系统定时器不同的第二个定时器，当进程暂停时，读出该定时器的内容看一看进程运行了多长时间。为了使记帐正确，当中断发生时，保存第二个定时器的内容，当中断结束时，恢复第二个定时器的内容。

一个不太精确但比较简单的记帐方法是：用一个全局变量维护一个指向当前运行进程在进程表中入口的指

针。在每个时钟滴答，增加当前进程入口表中的一个域的值。通过这种方法，每个时钟滴答都被滴答时刻所运行的相应进程所“支付”。这种方法的一个小问题是如果在一个进程运行过程中，发生了多次中断，即使进程没有做多少工作，该进程也支付了一个完整的滴答。由于对中断期间CPU记帐的合适的方法代价太高，因此从未使用过。

在MI NIX和许多其他的系统中，一个进程可以请求操作系统在一定的间隔后对它报警。报警通常是信号、消息或其他类似的东西。一个需要这些报警的应用是网络。在网络中，对在一段时间内没有应答的包必须重传。另一个应用是计算机辅助教学，当学生在一段时间内没有提供响应时，就告诉他答案。

如果时钟驱动程序有足够的时钟，它可以为每一个请求设置一个时钟。但实际情况不是这样。它必须使用单一物理时钟仿真多个虚拟时钟。一种办法是：设置一张表格和一个变量，在表中保存所有等待定时器的信号时间，变量则给出下一次发送信号的时间。无论何时更新时间，驱动程序都要检查是否到了最近的信号发送时间。如果时间到，就在表格中搜索要发送的下一个信号。

图3－25 用一个时钟模拟多个定时器。

如果需要许多信号，那么象图3－25中所示那样，把所有的时钟请求在一个链表中连接起来并按时间排序，以此模拟多个时钟将更加有效。链表中的每个入口指出前一信号发生以后需要等待多少个滴答才引发下一个信号。在这个例子中，信号发生在 4203、4207、4213、4215、4216。

在图3－25中，下一个中断发生在三个滴答以后。每次滴答，next_signal减一，当它为零时，就引发链表中第一项的信号，并把这一项从链表中移走，然后next_signal设置为链表中第一个元素的值，在这个例子中为4。

注意在时钟中断期间，时钟驱动程序有几件事要做 — 递增实际时间、递减时间片并检查是否为零、CPU记帐、递减报警计数器。因为这些工作每秒都要重复多次，每项工作都必须仔细安排以加快速度。

操作系统的有些部分也需要定时器，这些是所谓的监视定时器（watch-dog Timer）。当研究硬盘驱动程序的时候，我们看到，每次向硬盘控制器发命令时，都要安排唤醒调用，以便当命令执行完全失败时，可以试图进行恢复。我们也注意到软盘驱动程序不得不等待电机加速到一定的速度，并且如果在一段时间内没有任何活动，就关掉电机。一些具有可移动头的打印机可以每秒打印120个字符（8.3ms/字符），但是不能在8.3ms内把打印头返回最左边。当打印回车时，终端驱动程序必须延时。

时钟驱动程序处理监视定时器的机制和用户信号是一样的。唯一的不同是当定时器时间到时，时钟驱动程序将调用一个调用者提供的过程而不是产生一个信号，这个过程是调用者代码的一部分。因为所有的驱动程序处于同一地址空间，因此时钟驱动程序可以以任何方式调用它们。被调用的过程可以做需要做的任何工作，甚至可以引起一个中断，虽然在内核中，中断是不方便的，信号也不存在。这就是为什么要提供监视定时器机制的原因。

我们所列的最后一件事是直方图统计。一些操作系统提供一种机制使用户程序可以要求系统构造程序计数器直方图，以便分析时间在程序的各部分是如何分配的。当有可能统计直方图时，在每一时钟滴答，驱动程序都要检查是否对当前程序进行直方图统计，如果进行直方图统计，则先计算当前程序计数器值所在的地址区段，然后把相应的计数器加一。这种机制也能用来对系统本身进行直方图统计。

3.8.3 MINIX时钟驱动程序概述

MINIX时钟驱动程序包含在文件clock.c中，时钟任务接收六个带有如下所示参数的消息类型：

1. HARD_INT
2. GET_UPTIME
3. GET_TIME
4. SET_TIME（以秒计的新时间）
5. SET_ALARM（进程号，调用过程，延迟）
6. SET_SYN_AL（进程号，延迟）

HARD_INT是当发生时钟中断并有工作可做时发往驱动程序的消息，例如当必须发出一个报警或一个进程已经运行了很长时间的情况。

GET_UPTIME用来取从启动开始后以滴答计数的时间。GET_TIME返回从1970年1月1日上午12：00开始以秒计的当前时间。SET_TIME设置实际时间，它可以被超级用户激活。

时钟驱动程序内部，按图3－24（c）的方法记录时间。当设置时间时，驱动程序计算何时系统被启动，由于驱动程序维护当前的实际时间，它也知道系统已经运行了多少滴答数，所以它可以计算系统是何时启动的。系统在一个变量中存放启动的实际时间，以后当调用GET_TIME时，把滴答计数的当前值转换成秒，并把它和存储的启动时间相加。

SET_ALARM允许一个进程设置一个定时器，该定时器经过一个指定滴答计数的时间间隔后，将引起某些事件的发生。当用户进程执行ALARM调用时，它向存储管理器发一条消息，管理器再把消息发往时钟驱动程序，当报警发生时，时钟驱动程序向存储管理器发回一条消息，然后由它向相关进程发回一个信号。

需要启动监视定时器的任务也使用SET_ALARM，当定时器到达时，就简单地调用提交的过程。但驱动程序并不知道调用的过程做什么。

SET_SYN_AL和SET_ALARM相似，但是用来设置同步闹钟（Synchronous Alarm）。同步闹钟发送一条消息到进程而不是产生一个信号或调用一个过程。同步报警任务负责向各个需要消息的进程分派消息，在后面将仔细讨论同步报警。

时钟任务并没有使用主要的数据结构，但使用了几个变量来保存时间。只有一个全局变量lost_ticks，它定义在文件glo.h中（5031行）。如果未来加入到MINIX系统中的驱动程序，由于屏蔽中断时间太长以至于丢失一个或多个时钟滴答的话，可以使用这个变量来补偿。现在并未使用这个变量，但是如果编写了这样一个设备驱动程序，那么驱动程序可以通过增加lost_ticks来补偿在屏蔽中断期间丢失的时间。

很明显，时钟中断发生的频率很高，对时钟中断的快速处理是很重要的。MINIX通过对大多数的时钟中断进行最少的处理来实现这个目标。一旦接收到中断，处理程序把局部变量ticks设置为lost_ticks+1，然后使用这个值更新统计时间和pending_ticks（11079行），并把lost_ticks复位为零。pending_ticks是一个PRIVATE类型的变量，声明在所有函数定义之外，但仅对定义于clock.c文件中函数可见。另一个PRIVATE类型的变量sched_ticks1用于跟踪执行时间，每次滴答减一，如果报警时间到或允许时间片用完，中断处理程序就向时钟任务发送一条消息。这种方法使得大多数中断处理程序可以立即返回。

当时钟任务接收到任一消息时，它把pending_ticks加到变量realtime（11067行）上，然后置变量pending_ticks为零。变量realtime和变量boot_time（11068行）用以计算每天的当前时间，它们都是PRIVATE类型的变量。所以对于系统的任何部分唯一的取时间的方法是向时钟任务发一消息。虽然在任意瞬间，realtime可能是不精确的，但这种机制能保证当需要的时候，时间永远是精确的。如果你的手表当你看它的时候是准时的，而当你不看它的时候，它却不准时，这有关系吗？

为了处理闹钟，next_alarm记录了下一个信号或监视定时器调用发生的时刻。驱动程序在这里必须很谨慎，这是因为在信号发生以前，请求信号的进程可能已经终止或被杀死。当信号发生时，要检查一下是否需要该信号，如果不需要，也就没有必要处理了。

每一个用户进程只允许有一个未完成的闹钟定时器。当定时器还在工作时，执行一个ALARM调用会终止上一个定时器，因此，一种简便地保存定时器的方法是在每一个进程的进程表项中为定时器保留一个字。对于各个任务，被调用的函数必须在某处存放，所以设置了数组watch_dog。一个相似的数组syn_table用来存储指示对于每一个进程是否等待接收同步闹钟的标志。

时钟驱动程序的整个逻辑和磁盘驱动程序具有相同的模式。主程序是一个无限循环，重复执行取消息，再根据消息类型进行处理，然后发送一个应答（CLOCK_TICK除外）。每种类型的消息由不同的过程处理，它们按我们的标准命名约定命名，即对于从主循环调用的过程，都命名为do_XXX的形式，当然，各个XXX是不同的。顺便说一句，不幸的是，许多连接程序把各个过程名削减为7或8个字符，所以名字do_set_time和do_set_alarm有可能发生冲突，因此后一名字换名为do_setalarm，在整个MINIX中，这个问题都可能发生，一般通过更改名字来解决。

同步闹钟任务（Synchronous Alarm Task）

在这一部分，我们讨论另一类任务：同步闹钟任务（Synchronous Alarm Task）。同步闹钟和闹钟很相似，但是当计数时间到时，它既不发信号，也不调用监视定时器函数，而是向闹钟任务发一条消息。到达的信号或者调用的监视定时器任务可能和当前正在执行的任务部分毫无关系，所以所有这些类型的闹钟都是异步（Asynchronous ）的。与此相反，只有在接收方执行了receive调用后，才接收消息。

同步闹钟机制加入到MINIX中是为了支持网络服务器，它就象存储管理器和文件服务器一样，是作为一个独立的进程运行的。当进程因等待输入而被阻塞时，经常要设置一个等待时间的极限。例如，在网络中，在一定的时间内，可能由于传输失败而没有收到数据包的应答，网络服务器可以在试图接收消息并阻塞以前设置一个同步闹钟。因为同步闹钟是作为一条消息递交的，所以如果没有从网络中收到消息，最后它也将使服务器解除阻塞。如果收到了消息，服务器必须首先复位闹钟，然后检查消息的类型和来源，由此可以确定是数据包到达还是由于超时而解除阻塞。如果是后者，服务器一般通过重新发送最后一个没有确认的包来试图恢复。

同步闹钟比发送信号的闹钟要快，后者需要几条消息和一定量的处理，监视定时器函数也很快，但仅适用于和时钟任务编译进同一空间的任务。当进程在等待消息时，同步闹钟比信号或监视定时器更合适也更简单，而且仅仅需要一些附加操作即可很方便地处理。

时钟中断处理程序

就象前面讨论的，当时钟中断发生时，并不是立即更新realtime。中断例程维护变量pending_ticks计数器，并且完成象把当前的滴答计入到进程中并减小当前的时间片等简单工作，仅当必须完成更复杂的任务时，才向时钟任务发消息。这是和MINIX任务通过消息通信思想的一种妥协，但是它是对消耗CPU时间的一种让步。在一个慢速的机器上，人们发现，和在每个时钟中断向中断任务发消息的工作方式相比，按这种方式工作可以提高15％的速度。

毫秒定时

作为对实际情况的另一种妥协，在clock.c中提供了几个例程用于实现毫秒定时。有许多设备需要不超过一毫秒的延时，使用闹钟和消息传递接口没有办法解决这个问题。在这里提供的函数是被任务直接调用的，使用的是最古老和最简单的I/O技术：轮询法。按最快的速度直接读用于产生时钟中断的计数器，并把计数值转换

为毫秒。调用者重复执行该过程直至经过了所要求的时间。

时钟服务小结

图3－26总结了clock.c提供的各种服务。某些服务是所有的进程都可得到的，其结果通过消息返回。可以从内核或任务发出的函数调用获得时间，这种方法避免了消息机制的开销。用户进程可以请求一个闹钟。其结果是产生了一个信号，任务也可请求一个闹钟，其结果是激活监视定时器函数。这两种机制服务器系统都不能使用，但服务器可以请求同步闹钟。任务或内核可以通过mill_delay函数请求一个延迟，也可以在轮询例程中调用mill_elapsed，例如当等待从端口输入时就是如此。

3.8.4 MINIX时钟驱动程序的实现

当MINIX启动时，将调用所有的驱动程序。大多数驱动程序试图取一条消息并阻塞。时钟驱动程序clock_task（11098行）也是这样做的。但是它首先要调用init_clock来把可编程时钟的频率初始化为60HZ。当收到其他消息时，它把pending_ticks加到real_time上，然后在做其他任何事以前把pend_ticks复位。这个操作可能会和时钟中断发生冲突，因此调用了lock和unlock两个函数来避免冲突（11115行至11118行）。时钟中断处理程序除主循环以外的其他部分和其他的驱动程序是相同的：接收一条消息，调用一个完成所需要工作的函数，然后回送一个应答消息。

图3－26 支持和时间有关服务的时钟代码。

在时钟的每次滴答，并不都调用do_clocktick（11140行），因此其名字并不是其函数功能的准确描述。当中断处理程序确定有一些重要的工作需要做时才调用它。首先检查是否有信号或监视定时器事件到。如果其中之一发生，则检查在进程表中所有的闹钟项。因为时钟滴答并不是每个单独处理的，所以在一遍对进程表的扫描中可能会发现多个闹钟时间到。也有可能接收下一个闹钟的进程已经终止。当发现一个进程其闹钟时刻比当前时间要小，但不为零时，则检查和该进程相对应的在数组watch_dog中的项。在C语言中，数值型的值也具有一个逻辑值，因此如果在watch_dog数组中存放一个有效地址，11161行的测试将返回TRUE，并在11163行间接调用对应的函数。如果发现为空指针（NULL pointer）（在C中表示为数值零），其测试结果将为FALSE，并调用cause_sig来发送一个SIGALRM信号。当需要同步闹钟时，也使用监视器项，在这种情况下，存放的地址为cause_alarm而不是属于特定任务的监视器函数的地址。对于发送一个信号，当然可以存放cause_sig地址，但是然后将必须编写不同的cause_sig函数。该函数不需要参数，它从一个全局变量中取得目标进程号。另外，我们也可以要求所有的监视器进程希望一个它不需要的参数。

在以后的章节里，当我们讨论系统任务时，我们将讨论cause_sig。它的工作是发一条消息到存储管理器。这里需要检查是否存储管理器正在等待该消息。如果确实如此，它发一条消息通知闹钟时间已到。如果存储管理器忙，那么在第一次时将做一个标记来通知它。

当循环检查进程表中每个进程的p_alarming值时，更新next_alarm。在启动循环以前，它被设置为一个很大的数（11151行），然后对于每一个在发送了闹钟或信号后其闹钟值为非零的进程，将其闹钟值和next_alarm进行比较，后者被设置为较小的值（11171行和11172行）。

处理完闹钟以后，do_clockticks继续执行，检查是否到了调度另一进程的时刻。执行时间片的值保存在PRIVATE类型的变量shed_clocktick中，一般在每次时钟中断处理程序中将其减1。然而，在那些do_clocktick被激活的嘀哒中，中断处理程序并不减小它的值，而是让do_clocktick自己执行这项工作并测试结果是否为零（11178行）。当调度新进程时并不对sched_ticks复位（因为允许文件系统和存储管理程序完成这项工作），而是每次SCHED_RATE嘀哒后对其复位。11179行中的比较是用来保证当进程在被剥夺CPU以前确实至少运行了一个完整的调度滴答。

下一个过程do_getuptime（11189行）只有一行，它把realtime的当前值（从启动开始经过的滴答数）存入返回消息的正确域中，任何进程都可以通过这种方式取得已经过的时间。但是，对于访问时间的任务，消息传递的代价是很大的，因此提供了一个相关的函数get_uptime（11200行），它可以被任务直接调用。因为不是通过发送给时钟任务的消息激活其运行，所以它必须把耽误的时钟滴答加到当前的realtime上。在这里，lock和unlock也是必须使用的，它们用来防止当访问pend_ticks时发生时钟中断。

为了取得当前的时间，do_get_time（1121行）根据realtime和boottime（以秒计的从系统启动开始计算的时间）计算当前时间。do_set_time（11270行）则与do_get_time相反，它根据给定的当前时间和从启动开始计数的滴答数为boot_time计算一个新值。

过程do_setalarm（11242行）和do_setsyn_alarm（11269行）非常相似，因此放在一起讨论。二者都从消息中提取参数，一个参数说明要向其发送信号的进程，另一个参数说明等待消息的时间。do_setalarm也需要提取一个调用函数的参数。如果目标进程是用户进程而不是任务，调用函数可以取空指针。在前面我们已经说明在do_clocktick中对该指针进行检测来确定目标进程应当得到一个信号还是调用一个监视定时器函数。这两个函数也计算闹钟需等待的时间（以秒计）并把计算结果设置在返回的消息中。然后二者都调用common_setalarm完成相应的操作。如果是do_setsyn_alarm，传给common_setalarm的函数参数将永远是cause_alarm。

common_setalarm（11291行）完成上面所讨论的两个函数所启动的工作，然后把闹钟时间存放在进程表中，把指向监视定时器过程的指针存在watch_dog数组中（也可能是指向cause_alarm的指针或是空指针），然后它象do_clocktick一样扫描整个进程表寻找下一个闹钟。

cause_alarm（11318行）很简单，它把syn_table数组中对应于同步闹钟目标的项设置为TRUE。如果同步闹钟任务没有处于活跃状态，就发送一条消息唤醒它。

同步闹钟任务的实现

同步闹钟任务syn_alarm_task（11333行）和所有任务的模型是一样的。它首先初始化，然后进入一个循环重复接收或发送消息。初始化工作包括：首先通过设置syn_al_alive为真宣布任务为活跃状态，然后通过设置在syn_table中所有的项为FALSE宣布它无事可作。对于进程表每一项，在syn_table中都有一对应项。通过宣布已完成了工作，开始其外层循环，然后进入内层循环，其中检查在syn_table中每一项，如果发现一项指示等待一个同步闹钟，则首先把该项复位，然后发送一个类型为CLOCK_INT的消息至相应进程，并宣布其工作尚未完成。在外层循环的底部，除非设置了work_done标志，否则将不等待任何新消息。因为cause_alarm直接写入了syn_table中，所以并不需要一个新消息来告诉还需要做其他工作。仅当完成了所有工作以后才需要一条消息将它唤醒。其效果是只要有闹钟需发送就以很快的速度循环处理。

事实上，MINIX的发布版本并不使用这个任务。然而，如果重新编译MINIX来增加网络功能，网络服务器将会使用它，这是因为如果希望接收的包没有接收到，那么需要需要这种机制，以便加快进行超时处理。除了对速度的要求以外，使用这种机制的另一个原因是服务器应当不停地永远工作，而大多数信号的默认动作是撤销目标进程，所以不能向服务器发信号。

时钟中断处理的实现

时钟中断处理程序的设计是做很少的工作（因此使得处理时间很短）和完成较多的工作之间的妥协，后者使得费时的时钟任务的激活不太频繁。它仅仅改变和测试一些变量。clock_handler（11374行）先对系统记帐进行处理。MINIX既记录用户运行时间，也记录系统运行时间。如果时钟滴答时某进程正在运行，则对其用户时间记帐。如果文件系统或存储管理器正在运行，则对系统时间记帐。变量bill_ptr永远指向被调度的最后一个用户进程（不对两个服务器记帐），11447和11448行执行记帐，计算完毕后，把由clock_handler维护的最重要的变量pending_ticks加一（11450行）。为了测试是唤醒tty还是向时钟任务发送一条消息，必须知道实际的时间。但是，由于更新realtme要使用锁机构，所以其代价比较高。为了解决这个问题，中断服务程序计算自己的实际时间，并存放在局部变量now中。有可能其结果暂时是错误的，但结果并不严重。

驱动程序的其他工作取决于各种测试。它需要不停地唤醒终端和打印机。tty_timeout是一个由终端任务维护的全局变量，在这个变量中存放的是终端任务下一次被唤醒的时刻。对于打印机，需要检查几个打印机模块中PRIVATE类型的变量，再调用pr_restart对这些变量进行测试，在打印机挂起这种最坏的情况下，该函数也能很快返回。如果闹钟时间到或到了应该进行任务调度的时候，就执行11455至11458行的测试以激活时钟任务。后面的测试比较复杂，它是三个简单测试的逻辑与。11459行的代码

interrupt（clock）

引起发送一个HARD_INT消息到时钟任务。

当讨论do_clocktick时，我们注意到它对sched_ticks减一并测试是否为零以检查运行时间片是否用完。测试sched_ticks是否等于一是我们前面提到的复杂测试的一部分，如果时钟任务没有被激活，则有必要在中断处理程序中继续对sched_ticks执行减一操作，如果减至零，则对时间片复位。如果这发生了，注意这也是当前进程一个新的时间片的开始，这是通过在11466行把当前bill_ptr的值赋值为prev_ptr来完成的。

时间实用程序

最后，clock.c包含了一些提供各种支持的函数。它们中有许多是和硬件相关的。当把MINIX移植到非Intel的硬件时，必须替换这些程序。我们只讨论它们功能而不详细讨论其内部细节。

时钟任务第一次运行时调用init_clock（11474行），它设置定时器芯片的模式和时间延迟以产生每秒60次的时钟滴答中断。尽管人们在为PC做的广告上看到CPU的速度从初始的IBM－PC的4.77MHZ上升到在当今系统中200MHZ以上。但是，无论MINIX运行在哪种PC模型，初始化定时器的常数TIMER_COUNT都是不变的。每一个IBM兼容机，无论运行多快，都为各种需要基准时间的设备提供了一个14.3MHZ的信号。串行通信线和视频显示都需要这样一个基准。

和init_clock相对应的是clock_stop（11481行），它并不是必须的，而是对MINIX用户有时希望启动另一个操作系统这一事实的让步。它仅仅把定时器芯片的参数复位为默认操作方式，这正是MS－DOS或其他操作系统第一次开始运行时期望ROM BIOS提供的

mill_delay（11502行）是为需要极短延迟的任务而提供的。它用C语言编写，没有引入任何硬件相关性，但是使用了一种人们只有在低级汇编语言中找到的技术。它把计数器初始化为零，然后对其快速轮询直到到达指定的值。在第二章里，我们说过这种忙等待技术一般应该避免，但是，实现的必要性要求不能遵循一般的规则。下一个函数mill_start实现计数器的初始化（11516行），它只是简单地把两个变量清零。通过调用最后一个函数mill_elapse（11529行）实现轮询，它访问定时器硬件。被检查的计数器就是用来对时钟滴答下降计数的那个计数器，因此它可能下溢并在需要的延迟以前被复位为最大值。mill_elapse会纠正这个错误。

3.9终端

每台通用计算机都有一个或多个用来和它通讯的终端。终端有大量不同的型号，需要终端驱动程序来屏蔽这些细节，从而操作系统和用户程序的设备独立性部分对于不同型号的终端不必重新编写。在下面几节，我们

首先一般地讨论终端硬件和软件，然后讨论MINIX软件。

3.9.1终端硬件

从操作系统的观点来看，根据操作系统如何和终端通信，终端被分成三类：第一类为存储映像终端，包括键盘和显示器，二者都直接与计算机相连。第二类为使用RS－232标准串行通讯线，一般还经由调制解调器构成串行接口的终端。第三类为通过网络连接到计算机上的终端。

存储映像终端

图3－27中的第一类终端是存储映像终端。这些终端是计算机整体的一部分。存储映像终端使用称为视频RAM（Video RAM）的特殊存储器，视频RAM是计算机地址空间的一部分，通过和其他地址空间一样的方式对它进行访问。

图3－27 终端类型。

视频存储卡上有一个芯片称为视频控制器（Vedio Controller）。这个芯片从视频RAM中取出字符，产生用于驱动显示器（监视器）的视频信号。监视器产生水平扫描屏幕的电子束在屏幕上划线，典型的屏幕有480至1024行，每行有640至1200点。这些点称为象素（Pixel）。视频控制器调节电子束，决定一个象素是亮的还是黑的。彩色监视器有三个电子束，分别对应红色、绿色和蓝色，它们各自独立调节。

图3－28 存储映像终端直接写入视频RAM。

一个简单的单色显示器可能把一个字符显示为宽度为9个象素，高度为14个象素（包括字符间的空白），共显示25行，每行80个字符。这些显示器有350行扫描线，每行扫描线有720个点，每帧每秒重画45至70次。视频控制器被设计成首先从视频RAM中取80个字符，产生14行扫描线，再取80个字符，再产生14行扫描线，这样一直工作下去。事实上，大多数视频控制器显示每个字符的每行扫描线时，都取一次字符以便在控制器中不需要缓冲。每个字符的9列宽14行高的位模保存在视频控制器的视频ROM中（也可以使用RAM以支持用户字体）。ROM按12位编址，8位来自字符代码，4位指定扫描线。ROM中每个字节的8位控制8个象素，字符间的第九个象素永远为空。因此屏幕上的每行文本需14×80＝1120次存储器访问，也需访问相同次数的字符发生器。

IBM－PC有几种屏幕模式，在最简单的模式中，控制台使用一个字符映射显示器。在图3－29（a）中，我们看到了视频RAM的一部分。在图3－29（b）中屏幕上的每个字符在RAM中占两个字节，低字节是显示字符的ASCII码，高字节为属性字节，用于指定颜色、反显、闪烁等等。在这种模式下，满屏25行80列字符需4000字节的视频RAM。

图3－29（a）IBM单色显示器的视频RAM图象。　（b）和（a） 相对应的屏幕。×为属性字节

除了每个象素是独立控制之外，位映像终端使用相同的原理，对于单显这种最简单的配置，每个象素对应视频RAM中的一位，对于最复杂的配置，每一象素用24位的数来表示，红色、绿色和蓝色各8位。一个24位象素的768×1024的彩色显示器需2MB的RAM来存放图像。

对于存储器映像显示器，键盘是与显示器分开的，它可能通过一个串行口或并行口和计算机相连。对于每一个键动作，产生CPU中断，键盘中断程序通过读I/O口取得键入的字符。

在IBM－PC中，键盘包括一个内嵌的微处理器，通过特殊的串行口和母板上的一个控制芯片通讯。任何时刻击键或释放键，都产生一个中断，而且键盘仅仅提供键码，而不是ASCII码。当击A键时，键码（30）被存放于I/O寄存器。输入字符是大写、小写、CTRL－A、ALT－A、CTRL－ALT－A还是其他的组合则由驱动程序确定。因为驱动程序知道哪些键被按下还没有释放，因此它有足够的信息完成这项工作。虽然键盘接口把全部工作都交给了软件，但这也提供了很大的灵活性。例如，用户程序可能对一个数字是来源于最上面一行的键还是旁边的数字小键盘感兴趣。原则上，驱动程序可以提供这项信息。

RS－232终端

RS－232终端是包括一个键盘和一个显示器的设备，通过一次传输一位的串行口与计算机通讯（参见图3－30）。这些终端使用9针或25针的连接器，其中一针为发送数据，一针为接收数据，一针为地，其他各针可用于各种控制功能，实际上大多数并未使用。为了向RS－232终端发一个字符，计算机必须一次传输一位，在字符前面加一起始位，后接一个或两个终止位为字符定界。可以在终止位前插入一提供基本校验的奇偶位，但是通常仅在与主机系统通讯时才需要这种技术。一般传输速率为9600、19200或38400 bps。RS－232终端通常用于和使用远程计算机通讯，两者之间使用调制解调器及电话线。

图3－30一个RS－232终端通过一条一次传输一位的通讯线路和计算机通讯。计算机和终端完全独立。

计算机和终端在内部都是对整个字符进行操作的，但又必须通过串行线路以一次传输一位的方式来通讯，因此开发了芯片来实现字符到串行口和串行口到字符的转换。它们被称为UART（通用异步收发器 － Universal Asynchronous Receiver Transmitter）。UART通过象示于图3－31那样把RS－232接口板插入总线和计算机相连。RS－232终端正逐渐消失，而由PC和X终端代替，但是它们用于一些老的大型系统中，特别是银行、飞机定票等类似的系统中。

为了显示出一个字符，终端驱动程序把字符写入接口卡，这个字符被缓冲在接口卡中，然后通过UART在串行线上一位一位地发送出去。即使传输速率为38400bps，也需要250个毫秒来发送一个字符。由于传输速率很低，驱动程序一般向RS－232卡输出一个字符后就阻塞，等待字符传输完毕以后产生中断。UART能同时接收另一个字

符。象其名称所指出的，UART可以同时发送和接收字符。当接收到一个字符时一般也产生中断，UART能够缓冲少量的字符。当接收到中断时，终端驱动程序必须检查一个寄存器以确定中断源。一些接口卡有CPU和存储器，并能操纵多条线路，从而承担了许多原先由CPU完成的I/O工作。

正如前边提到的，RS－232终端可以进一步分成几类。最简单的是硬拷贝终端。通过键盘键入的字符传输至主机，主机传出的字符打印在纸上。这些终端已过时并很少见到。

哑CRT终端也按这种方式工作，仅仅用屏幕代替了纸，因为在功能上和硬拷贝是一样的，这些终端常被称为“玻璃tty”（术语tty是TeleType的缩写，TeleType是一个公司，过去该公司是计算机终端企业的先驱；tty的含义为任何终端）。玻璃终端也已过时。

智能终端事实上是微缩的专用计算机，它们有CPU、存储器和软件，软件一般在ROM中。从操作系统的观点，玻璃终端和智能终端的不同在于后者可以理解特殊的转义字符序列，例如通过发送ASCII ESC字符（033）后接各种其他的字符可以把光标移至屏幕上任何位置，在屏幕中间插入文本等等。

X终端

智能终端最高档的一种是包含和主机CPU一样强大的CPU的终端，同时包含几兆字节的主存、键盘和鼠标。这种类型的一种常用的终端就是X终端，在其上运行MIT的X窗口系统。一般X终端通过以太网和主机通讯。

一个X终端是运行X软件的计算机。一些产品只能运行X，其他的为通用计算机，把X作为和其他程序一样的程序来运行。无论哪种方式，X终端都有一个大的位映射屏幕，一般960×1200或更高，黑色 、灰色或彩色，一个完整的键盘，一个鼠标。鼠标一般有三个按钮。

X终端内收集从键盘或鼠标来的输入并接收远程计算机命令的一个程序称为X服务器，它通过网络和运行在远程主机上的X客户通讯。使X服务器运行于终端内，客户程序运行于远程主机上显得很奇怪，但是X服务器的工作是位显示，因此使它靠近用户是有益的。客户和服务器的管理示于图3－31中。

图3－31 M.I.T X窗口系统中的客户和服务器。

X终端的屏幕包含一些窗口，每个窗口都是长方形象素网格形式，在其顶部有一标题条，左边有一滚动条，在右上角有一改变窗口大小的小方框。一个X客户是一个称为窗口管理器（Window Manager）的程序，它的工作是控制在屏幕上创建、删除和移动窗口。为了管理窗口，它向X服务器发一命令，告诉它做什么。这些命令包括划点、划线、划矩形、划多边形、填充矩形、填充多边形等等。

X服务器的工作是调度来源于鼠标、键盘和X客户的输入并更新显示。它要跟踪当前选中了哪个窗口（鼠标所指的），所以它知道键盘输入的内容送给哪个客户。

3.9.2终端软件

键盘和显示器几乎是独立的设备，因此我们分别讨论它们（它们并不是完全独立的，因为键入的字符必须在显示器上打印出来）。在MINIX中，键盘和显示器是同一任务的不同部分，在其他的系统中，它们可能分成不同的驱动程序。

输入软件

键盘驱动程序基本的工作是收集从键盘输入的信息，当用户程序读终端时，把它传输给用户程序。对于键盘驱动程序的设计可以采用两种思想：第一，驱动程序的工作仅仅是接收输入，并且不经任何修改就向上层传递。一个从终端读的程序得到纯粹的ASCII码序列（为用户程序提供键码太原始，也过分依赖于机器）。

这种思想非常适用于象emacs这样复杂的屏幕编辑器的需要。emacs允许用户把任意动作捆绑到任意字符或字符序列上。然而，这意味着如果用户键入了dste而不是date，然后键入三个退格建和ate键来对此进行修正，最后键入一个回车键，那么用户程序将收到键入的全部11个ASCII码。

大多数程序不需要这么多细节，它们仅仅希望得到正确的输入，而不是如何生成它的完整的序列。基于这一点，产生了第二种思想：驱动程序处理行内的编辑，仅仅向用户程序传输正确的一行。第一种思想是面向字符的，第二种思想是面向行的。我们把它们分别称为原始模式（Raw Mode）和熟模式（Cooked Mode）。POSIX标准使用了一个不太形象的术语规范模式（Canonical Mode）来描述面向行的模式。在大多数系统中规范模式指的是定义好的配置。非规范模式（Uncanonical Mode）指的是原始模式。POSIX兼容的系统提供了几个库函数以支持选择哪种模式和改变终端配置。在MINIX系统中，IOCTL系统调用支持这些函数。

键盘驱动程序的第一个任务是收集字符。如果每次击键产生一个中断，驱动程序可以在中断过程中获得字符。如果中断被低层次的软件变成了消息，可以把最新获得的字符放入消息中，也可以放在内存的一个小缓冲中，再由消息告诉驱动程序某件事情发生。如果仅仅把消息送到等待进程，那么就使键盘驱动程序还有机会忙于处理前一字符，因此后一种方法比较安全。

一旦驱动程序接收到了字符，它必须开始处理它。如果键盘传过来的是键码而不是应用软件使用的字符代码，驱动程序必须使用一个表格对其进行转换。并不是所有的IBM“兼容机”都使用标准键码，所以如果驱动程序希望支持这些机器，必须利用不同的表格进行不同的映射。一种简单的办法是在驱动程序中编辑一个表格以进行键盘提供的代码和ASCII码（美国信息交换标准代码）之间的映射，但是这对于非英语的用户是不令人满意的。在不同的国家，键盘的安排是不同的，即使对西半球的大多数人而言，ASCII码集也是不够的。西班牙语、葡萄牙语和法语需要英语中不使用的标点符号和重音字符。为了满足键盘布局的灵活性以提供支持不同语言的需要，许多操作系统提供了可装载的键位表（Keymap）或代码页（Code Page），从而使得选择在键码和传输给

应用程序的代码之间的映射成为可能。这可以在系统启动时实现，也可以在启动后实现。

如果终端工作于规范模式（熟模式），必须保存字符直至累积到一行，这是因为用户以后可能决定删除其中一部分。即使终端工作在原始模式，程序可能还未请求输入，所以为了支持预输入也必须对字符实施缓冲（不允许用户提前输入的系统设计者应该被涂以焦油，用羽毛装饰甚至强迫使用自己的系统）。

一般有两种字符缓冲的方法。对于第一种方法，驱动程序包含一个集中的缓冲池，每个缓冲中大约可存放10个字符。和每个终端相关的是一数据结构，在该结构中除了其他内容外，包含一个缓冲链指针，而缓冲链中的各缓冲存放从该终端上输入的各字符。用户键入的字符越多，用于输入的缓冲就越多，挂在链上的缓冲也越多。当字符传送给用户程序时，把缓冲移走放回缓冲池中。

另一种方法直接在终端数据结构中实施缓冲，没有集中的缓冲池。用户输入一条命令，该命令需运行一段时间（例如编译），因此通常用户再预输入几行。出于安全性的考虑，驱动程序应该为每个终端分配大约200个字节的缓冲。在一个具有100个终端的大规模分时系统中，固定分配20K用于预输入显然太多了，而在集中缓冲池中有5K一般也足够了。另一方面，为每个终端指定一个缓冲使驱动程序比较简单（没有链接表管理），在具有一两个终端的计算机上，用户更偏爱这种方法。图3－32显示了这两种方法的不同。

尽管键盘和显示器在逻辑上是分立的设备，但是许多用户已习惯于在屏幕上看见他们刚刚键入的字符。一些过时的终端把键入的内容自动显示出来，这不仅对于输入口令是很令人讨厌的，而且极大限制了复杂编辑器和应用程序的灵活性。幸运的是，大多数的现代终端在输入时不显示任何内容，而由软件来显示输入的内容，这个过程称为回显（Echoing）。

当用户击键时，程序可能正在写屏幕，这使得回显过程复杂化了。至少，键盘驱动程序要计算出新输入字符的显示位置，使其不被应用程序的输出所覆盖。

在每行80个字符的终端上，键入超过80个字符时，回显也变得复杂了。依赖于应用程序，转入下一行可能是合适的。有些驱动程序通过把一行中第80个字符以后的字符去掉来解决这个问题。

图3－32 （a）中央缓冲池。 （b）每个终端指定的缓冲。

另一个问题是TAB键的处理。大多数键盘都有一个TAB键，但是几乎没有终端能处理TAB键的输出。驱动程序需要既考虑到应用程序的输出，又考虑到回显的输出，从而计算出把光标定位到何处，也要计算回显正确的空格字符数。

现在我们讨论设备等效性问题。从逻辑上说，在每一文本行的末尾，需要一个回车键把光标移到第一列，然后一个换行键进到下一行。需用户在每一行末尾键入回车和换行两键是不合适的（虽然有些终端有产生这两个字符的键，但应用程序仅有50％的可能需要它）。驱动程序通常把输入的内容转换成操作系统使用的内部标准。

如果标准格式仅仅存储换行（MINIX格式），那么回车应被转换为换行。如果内部格式存储回车和换行，那么驱动程序应该在键入回车时生成一换行符，键入换行时生成一回车符。无论在内部如何转换，为了使屏幕能正确更新，终端可能要求回显回车和换行。因为一台大型计算机可能和大量不同的终端相连，需要键盘驱动程序把不同的回车/换行组合转换成内部系统标准并正确处理回显。

一个相应的问题是回车换行的定时问题。在一些终端上，显示一个回车或换行比显示字符或数字的时间长。如果终端内的微处理机不得不拷贝一大块文本来实现滚动一行，换行可能会慢一些。如果机械打印头不得不回到打印纸的左边，回车会慢一些。在这两种情况下，都需要驱动程序向输出流中插入填充字符（Filler Character）（假的空字符）或者等待足够长的时间以使终端跟得上。延迟时间一般与终端的速度有关，例如，4800bps或更低可能不需要延迟，但在9600或更高的速度，需要一个填充字符。如果硬件支持TAB，特别是硬拷贝输出TAB后也需要一个延迟。

当操作在规范模式时，许多输入字符有特殊的含义。图3－33显示了POSIX需要的特殊字符和MINIX识别的附加字符。其默认值都是和程序使用的代码和文本输入不相冲突的控制字符，但如果需要，除最后两个以外，它们都可以由tty命令设置。老版本的UNIX对这些字符的大多数使用不同的默认值。

图3－33在规范模式下特殊处理的字符。

ERASE字符使用户擦除刚刚输入的字符。在MINIX中，这是退格符（CTRL－H）。并不把这个字符插入到字符队列，而是从队列中移走前一字符。为了从屏幕上去掉一个字符，它应顺序回显三个字符：退格、空格和退格。如果前一字符为TAB，把它擦除需要跟踪在TAB前光标在何处。在大多数系统中，退格仅仅擦除当前行的字符，不能擦除回车回到上一行。

当用户在输入的一行中的起始部分发现一错误，删除整行重新输入是很方便的。KILL字符（在MINIX中CTRL－U）删除当前行。MINIX使删除的行从屏幕上消失，但是在一些系统中，由于某些用户喜欢看到从前的一行，因此其处理是在后面放一回车和换行并回显之。所以如何处理KILL字符是一个喜好的问题。关于ERASE字符，一般不可能退回到离开当前行。当删除很多字符时，如果使用了缓冲，那么驱动程序是否把缓冲退还给缓冲池可能值得，也可能不值得。

有时ERASE或KILL字符必须作为普通数据输入，LNEXT字符起转义字符（Escape Character）的作用。在MINIX中，其默认值为CTRL－V。作为一个例子，在老版本的MINIX中，经常使用@作为KILL，但是互连网邮件系统使用形式为linda@cs.washington.edu的地址，对于老习惯感觉比较舒服的人可能会重定义KILL为@，但是在e-mail

地址中需要输入@，这可以通过键入CTRL－V@来实现。CTRL－V自身可以通过键入CTRL－V　CTRL－V来输入。发现CTRL－V以后，驱动程序设置一个标志，宣布对下一个字符不进行特殊处理。LNEXT字符自身不输入到字符序列中。

为了使用户能使屏幕图像停下来不致滚出窗口以外，提供了用户冻结屏幕和以后重启动屏幕的控制码。在MINIX中，分别是STOP（CTRL－S）和START（CTRL－Q），它们不被存储，但用来清除和设置终端数据结构中的一个标志，无论何时试图输出，检查一下这个标志，如果设置，不执行输出。一般一起禁止回显和程序输出。

经常需要杀死一些失去控制的正在调试的程序，使用INTR（DEL）和QUIT（CTRL－\）的目的即在于此。在MINIX中，DEL引起发送一个SIGINT信号到从该终端启动的所有进程。实现DEL很有技巧性，困难在于从驱动程序取得信息，然后送到处理信号的系统部分，但它并没有要求这种信息。CTRL－\和DEL是相似的，但是发送SIGINT信号，如果该信号未被忽略或捕获，则强制进行核心转储。无论键入哪个键，驱动程序应回显一个回车换行，丢掉所有的输入以允许重新启动。INTR的默认值通常为CTRL－C而不是DEL，这是因为许多程序内部把DEL作为编辑的退格建。

另一特殊字符为EOF（CTRL－D），在MINIX中，即使缓冲为空，也使等待从缓冲读的请求得到满足，缓冲中有什么就读出什么。在一行的开始键入CTRL－D使应用程序读出零字节，按惯例这被解释为读到了文件尾，使应用程序就象在输入文件中看到了文件尾一样进行操作。

一些终端驱动程序允许比我们前面讨论的更加令人喜欢的行内编辑。这些终端有特殊的控制字符用于删除一个字、向前或向后跳几个字或字符、回到输入行的行首或行尾等等。把所有这些功能加到终端驱动程序会使它变得很大，而且当使用工作于原始模式的屏幕编辑器时这也是一种浪费。

为了使程序控制终端参数，在标准库中POSIX需要几个函数，最重要的两个函数为tcssetattr和tcgetattr。tcgetattr取回一个如图3－34中显示的数据结构的一份拷贝，该结构为termios，它包含用来改变特殊字符、设置模式和修改终端其他特性的所有信息。程序可以检查当前的设置，当需要时对这些设置进行修改，然后调用tcsetattr把这个结构写回终端任务中。

图3－34 termios结构。在MINIX中 tc_flag_t的类型为short，speed_t的类型为 int ，cc_t的类型为 char 。

POSIX并未规定这些需求是通过库函数实现还是通过系统调用实现，MINIX提供了一个系统调用IOCTL，通过以下的形式调用：

ioctl（descriptor，request，argp）

它用于检查和修改许多设备的配置。用这个调用实现tegetattr和tcsetattr。变量request指出了是读还是写termios结构，在后一种情况下，该请求是立即起作用还是等到当前输出队列中的内容全部输出时才起作用。变量是argp指向一个在调用程序中termios结构的指针。这种在应用程序和驱动程序之间的特殊通讯方式是为了UNIX兼容性而不是为了固有的美感。

顺序讨论一下termios结构。四个标志提供了很大的灵活性。在c_iflag中的各位控制处理输入的各种方法。例如ICRNL位使在输入中的CR字符转换为NL字符，这个标志在MINIX中设置为默认值。c_oflag存放影响输出处理的各位，例如OPOST位允许输出处理。MINIX也设置了该位和ONLCR位，这两位使输出的NL字符转换为CR NL序列。c_cflag是控制标志，MINIX的默认设置使一条线路接收8位字符，而且如果用户在一条线路注销，则使调制解调器挂起。c_lflag标志是局部模式标志域，ECHO位允许回显（登录时可以关闭以提供登录时的安全性）。最重要的位为ICANON位，它使终端工作于规范模式，在关闭ICANON位情况下，还存在几种可能的设置。如果其他的设置都是默认值，则进入常规的cbreak模式。在这种模式下，输入的字符不必等待满行就传给了应用程序，但是INTR、QUIT、START和STOP还起作用。通过在标志中对相应的位复位，所有这些都可以禁止，其结果为常规的原始模式。

各种可以改变的特殊字符，包括那些MINIX扩充的字符，都存放在c_cc数组中，该数组中也存放了两个使用在非规范模式的参数，存储在c_cc[VMIN]中的数值MIN规定了READ调用读出的最少字符数。c_cc[VTIME]中的值TIME设置了这些调用的时间极限。MIN和TIME组合起来的使用示于图3－35，显示的是请求N个字节的调用。当TIME＝0同时MIN＝1时，其行为类似于常规的原始模式。

图3－35　MIN和TIME决定在非规范模式下当调用读时如何返回，N是请求的字节数。

输出软件

输出比输入简单，但是，RS－232终端的驱动程序和存储映像终端的驱动程序基本上是不同的。一般用于RS－232的方法是为每一终端设置缓冲，缓冲可以来源于同时作为输入缓冲的缓冲池，也可以是象输入一样指定的专用缓冲。当程序向终端写时，输出被首先拷贝到缓冲。同样，回显输出也拷贝到缓冲。当所有的输出拷贝到缓冲以后（或缓冲满），输出第一个字符，驱动程序睡眠，当中断发生时输出下一个字符，如此进行下去。

对于存储映像终端，一个更加简单的方案是可行的。需要打印的字符在某一时刻从用户空间中取出，直接放入视频RAM中。对于RS－232终端，每一字符通过传输线送至终端，对于存储映像终端，一些字符需特殊处理，其中包括退格、回车、换行和响铃（CTRL－G）。一个存储映像终端的驱动程序必须在软件中跟踪在视频RAM中当前位置，以便在那里打印可打印字符，然后向前移动输出位置。空格、回车、换行都需要相应地更新位置。

特别是当在屏幕底部行的末尾输出换行时，屏幕必须上滚。为了分析滚动如何实现，看一下图3－29，如

果视频控制器永远从0XB8000开始读RAM，滚动屏幕的唯一办法是拷贝24×80个字符到0XB0000（每个字符需两个字节，这是很费时间的）。

幸运的是，在这里，硬件提供了一些帮助。大多数的视频控制器包含一个寄存器，这个寄存器控制在视频RAM中从何处开始取屏幕上的最顶行的字节，通过设置该寄存器指向0XB00A0而不是0XB0000，以前的第二行移到了最顶行，这个屏幕上滚一行。驱动程序必须做的其他工作仅仅是拷贝新的最底行的内容，当视频控制器指向了RAM的最高端后，它仅仅返回到最低地址并开始继续取数据。

驱动程序必须处理的另一个问题是存储映像终端的光标位置，硬件一般也用一个寄存器来提供帮助，该寄存器告诉光标去向何处。最后还有一个响铃问题，这通过向扬声器送一正弦波或方波来产生。扬声器是计算机中和视频RAM不同的部分。

值得注意的是，存储映像终端驱动程序所面临的许多问题（滚动、响铃等等）RS－232终端中的微处理机也会遇到。从微处理机的观点来看，它是带有存储映像显示器系统中的的主处理机。

屏幕编辑和许多其他的复杂程序需要比仅仅把文本从底部上滚的方式更加复杂的更新屏幕的方式。为了支持这些功能，许多终端驱动程序支持大量的转义序列。虽然许多终端驱动程序支持各别的转义序列集，但是有一个标准使一个系统的软件能适应另一个系统是有益的。美国国家标准委员会（ANSI）定义了一套标准的转义序列，MINIX支持ANSI标准的转义序列的一个子集，该子集示于图3－36，对于许多公共操作，这已经足够了。当驱动程序看到开始转义序列时，它设置一个标志并等待转义序列的其他部分进入系统。当全部进入以后，驱动程序必须在软件中实现转义序列指定的操作。插入和删除文本需要在RAM中移动大量字符。硬件除了滚动和显示光标外不能提供任何帮助。

图3－36 终端输出驱动程序接受的ANSII码转义序列。ESC表示ASCII码转义字符（0X1B）。n，m和s为可选的数值参数。

3.9.3 MINIX中终端驱动程序概述

终端驱动程序包含在四个文件中（如果支持RS－232和伪终端，则为六个文件），它们构成了MINIX中最大的驱动程序。终端驱动程序既处理键盘，也处理显示器，二者都很复杂，同时又包括两个可选择终端，因此终端驱动程序是分成几部分来解释的。和调度程序比较，终端驱动程序要比其大30倍，这对于大多数人而言，一定感到非常吃惊（看一看大量关于操作系统的书籍，这些书籍中关于调度程序的内容相当于把所有I/O的内容加在一起的30倍，这一定会使这些人更吃惊）。

终端驱动程序接收七种消息类型：

1 从终端读（来自FS，它代表用户进程）

2 向终端写（来自FS，它代表用户进程）

3 为IOCTL设置终端参数（来自FS，它代表用户进程）

4 上一次时钟滴答时发生I/O（来自时钟中断）

5 终止上一个请求（当信号发生时，来自文件系统）

6 打开设备

7 关闭设备

除了不需要POSITION域外，读写消息和示于图3－15中的消息具有相同的格式。对于磁盘，程序必须指定它想读那一块，对于终端，却没有这项选择。程序永远取键入的下一个字符，终端不支持寻道。

使用IOCTL系统调用的POSIX函数tcgetattr和tcsetattr检查和修改终端属性（特性），好的编程实践是使用它们和其他在include/termiios.h中的函数，让库函数调用IOCTL系统调用。有一些MINIX需要的控制操作在POSIX中没有提供，装入可选择的键位表就是一个例子。对于这些问题，程序员必须显式地使用IOCTL。

通过IOCTL系统调用发往驱动程序的消息包括一个功能请求代码和一个指针。对于tcsetattr函数，执行的是一个具有TCSETS、TCSETSW或TCSETSF请求类型和一个指向类似图3－34的termios结构指针的IOCTL调用。所有这些调用把当前的属性集换成一个新的属性集，它们区别在于：TCSETS请求立即起作用，TCSETSW请求直到所有的输出被输出后才起作用，TCSETF等待输出完成并抛弃掉还没有读的输入。tcgetattr转换为一个具有TCGETS请求类型的IOCTL调用，并返回调用者一个填入termios结构的消息，因此，可使用该调用检查当前设备的状态。有一些IOCTL调用并不完全对应于POSIX定义的函数，它传输的是另外几种结构的指针，例如，KIOCSMAP请求用于装入一个新键位表，于是需传输的是一个指向keymap_t的指针，这是一个1536字节的结构（128键×6修饰符，每项16位）。图3－43概括了标准的POSIX调用是如何转换成IOCTL系统调用的。

终端驱动程序使用的一个主要的数据结构为tty_table，它是一个tty结构的数组，每个终端一个。标准PC仅有一个键盘和显示器，但MINIX可支持多达8个虚拟终端，这取决于显示适配器卡的存储空间大小。这允许使用控制台的用户登录多次，并且显示输出和键盘输入可从一个用户切换到另一个用户。对于虚拟控制台，按ALT－F2选择第二个用户，ALT－F1返回第一个，也可使用ALT和箭头键。除此之外，串行线路也支持通过RS－232和调制解调器相连的两个远程用户，也支持用户通过网络相连接的伪终端。驱动程序被编写成很容易增加附加终端。在书中的源码中，支持两个虚拟控制台，不支持串行线和伪终端。

在tty_table中的每个tty结构既跟踪输入，也跟踪输出。对于输入，它存放了一个所有已经键入但还没有被程序读出字符的队列，它也存放了请求读但还未接收到字符的信息，以及超时信息，因此如果没有键入字符

也能请求输入，而且可以使任务不永久阻塞。对于输出，它存放了没有完成的写请求的参数，其他的域存放着各种通用变量，例如，前面讨论的termios结构，该结构影响输入输出的许多特性。在tty结构中还有一个指向特殊类设备需要的信息，但在tty_table项中并不是每个设备都需要。例如，控制台驱动程序和硬件有关的部分需要在屏幕上和视频RAM中的当前位置，以及当前显示的属性字节，但对于RS－232传输线，则不需要这些信息。每种类型设备的私有数据结构也与缓冲处于同一位置，该缓冲存放来自中断服务例程的数据。慢速设备，例如键盘不需要象快速设备所需要的那么大的缓冲。

终端输入

为了更好地理解终端驱动程序如何工作，让我们首先看一下在终端上键入的字符是如何从系统到需要它们的程序的。

当用户在控制台上登录时，系统为其创建一个shell，它用/dev/console作为标准输入、标准输出和标准出错。shell启动并试图通过调用库过程read读标准输入，这个过程发送一条包含文件描述符、缓冲地址和字节数的消息到文件系统。这条消息就象图3－37（1）所示的那样。发送消息以后，shell阻塞，等待应答（用户进程仅仅执行SEND_REC原语，这个原语把一个SEND和一个从发往进程的RECEIVE结合起来）。

文件系统取消息，并找到对应文件描述符的i结点，这个i结点对应一个字符设备设备文件/dev/console，并且包含终端的主次设备号，终端的主设备号为4，对控制台，次设备号为0。

文件系统在设备映照图dmap中查找终端任务号。如图3－37（2）所示，向终端任务发一条消息。通常用户至此还没有键入，因此终端驱动程序不能满足这个请求。它立即发回一个应答使文件系统解除阻塞，报告不能得到任何字符，这正如图3－17（3）所示。

图3－37 当还未输入字符时来自终端的读请求。FS是文件系统，TTY是终端任务。当输入字符时，终端的中断处理程序把输入字符排成一个队列，但是在这里由时钟中断处理程序唤醒TTY。

文件系统在tty_table的控制台数据结构中记录一个进程等待终端输入，然后去处理下一个请求。用户shell继续阻塞，直至请求字符到达之后。

当字符最后在键盘上键入以后，它引起两个中断，其中之一是当按下键的时候，另一个是当释放键的时候。这个规则对于象CTRL和SHIFT这样的修饰键也是适用的，这些键本身并不传输任何数据，但每个键仍要引起两个中断。键盘中断为IRQ1，在汇编代码文件mpx386.s中的_hwint01激活kbd_hw_int（13123行），然后调用scan_keyboard（13432行）从键盘硬件取得键码。如果键码是普通字符，并且中断是由按下键而产生的，那么把它放入输入队列ibuf中，但是如果中断是由于释放键而引起，则忽略它。对于两种类型中断，CTRL和SHIFT这类修饰键的代码也放在队列中，但可以通过一个仅仅当键释放时设置的位区分开。注意在这点上，接收和存储在ibuf中的代码并不是ASCII码，它们仅仅是IBM键盘产生的扫描码。然后kbd_hw_int设置一个标志tty_events（tty_table中的有关键盘的部分），调用force_timeout，最后返回。

不象其他的中断服务例程，kbd_hw_int不发送消息唤醒终端任务。在图中虚线（4）表示调用force_timeout，这些不是消息。它们设置对中断服务例程通用的地址空间中的变量tty_timeout。在下一次时钟中断，clock_handler发现tty_timeout指明现在是调用tty_wakeup的时刻，于是它向终端任务发送一条消息（5）。注意虽然tty_wakeup的源码在tty.c中，它却根据时钟中断来运行，因此我们说时钟中断向终端任务发消息。如果输入到达得很快，大量的字符代码按这种方式排成一个队列，这就是为何在图中显示了多次调用force_timeout。

一旦接收到了唤醒消息，终端任务检查每一个终端设备的tty_events标志。对于每一设置了该标志的终端，调用handle_events（12256行）。tty_events标志指出了各种活动（虽然大多数为输入），因此，handle_events永远调用和设备有关的用于读写的函数。对于从键盘输入，将调用kb_read（13165行），该函数跟踪指示CTRL、SHIFT和ALT键的按下和释放的键盘代码，把键盘代码转换为ASCII码。kb_read转而调用in_process（12367行），该函数处理ASCII码，它要考虑特殊字符和可能设置的不同标志，包括规范模式是否有效，虽然一些代码，例如BACK SPACE有其他的功能，但是其一般处理结果为向在tty_table中的控制台输入队列中加入字符。通常in_process还要启动向显示器回显ASCII码。

当输入了足够多的字符时，中断服务程序调用汇编语言例程phys_copy把数据拷贝到shell请求的地址中去。这个操作也不是通过消息传递实现的，正因为如此，在图3－37中通过虚线（6）表示。这样的线显示了若干条，这是因为在用户请求完全满足以前，可能有多个这样的操作。当这个操作最后完成的时候，终端驱动程序向文件系统发一条消息，告诉它工作已完成（7）。文件系统响应这条消息，向shell发一条消息来使其接触阻塞。

怎样才算输入了足够多的字符呢？这取决于终端模式。在规范模式，当收到回车、行结束或文件尾代码时，请求就完成了。为了对输入进行合适的处理，输入的一行不能超过输入队列的大小。在非规范模式，读可以请求大量的字符，in_process可能不得不在返回文件系统一条消息告诉它操作完成以前传输多次字符。

注意终端驱动程序把实际的字符直接从它自己的地址空间中拷贝到shell空间中，它并不首先通过文件系统。对于块I/O，数据确实通过了文件系统，以维护一个最近使用数据块的缓冲。如果请求块正好在缓冲中，请求可以直接由文件系统满足，不需做任何磁盘I/O。

对于终端I/O，缓冲毫无意义，而且文件系统到磁盘驱动程序的请求总可以在几百个毫秒内得到满足，因此使文件系统等待没有什么损失。终端I/O可能需几个小时才能完成，甚至永远不能完成（在规范模式，可能等

待的时间更长，这取决于MIN和TIME的设置），因此，使文件系统阻塞直至终端请求满足是不合适的。

此后，用户可能预键入了一些字符，这些字符在请求以前就已准备好，在这种情况下，事件1、2、6、7和8在读请求以后相继顺序发生，3则不发生。

如果当时钟中断时，终端任务碰巧正在运行，因为它还没有等待消息，因此不能向它发消息。然而，为了在终端任务忙时能平稳地输入输出，要检查几次所有终端设备的tty_events标志，例如，紧接着消息的处理和响应立即进行一次这种检查，于是没有来自时钟的唤醒的帮助，字符也能加入到控制台队列中。如果在终端驱动程序完成它正在做的工作以前，发生了两次或多次时钟中断，所有的字符都存储在ibuf中，并重复设置tty_flags，最后终端任务得到一条消息，其他的消息就丢失了。但是，因为所有的字符都安全地存储在缓冲中，输入并未丢失。甚至有可能在终端任务接收到消息时，输入已经完成，应答也发往了用户进程。

在这种设计中，存在一个当中断例程向忙进程发消息时，如果是在一个无缓冲的消息系统中如何处理的问题。对于大多数的设备，例如磁盘，中断是由于响应驱动程序发出的命令而发生的，因此在任何时刻，仅可能等待一个中断。其自身产生中断的设备是时钟和终端（当允许时，网络也产生）。通过对没有处理的滴答进行计数来处理时钟，因此如果时钟任务没有从时钟中断得到消息，以后的处理会对其补偿。对终端的处理方法是使中断例程在缓冲中积累字符同时增加标志，指示已收到了字符的方法处理终端，如果终端任务正在运行，在其睡眠以前检查这些标志，如果有其他工作要做就推迟睡眠。

终端任务并不被终端中断直接唤醒，因为那样做负担太重。每一次终端中断之后的下一次滴答时钟向终端任务发一个中断。以每分钟100个字的速度，打字员每秒的输入不超过10个字符，即使对于快速的打字员，对于在键盘上输入的每个字符，也会向终端任务发一条消息，虽然有些消息可能丢失。如果在缓冲中没有空间时仍有字符到来，就把该字符丢弃。但是经验显示，对于键盘，32个字符的缓冲就足够了。对于其他输入设备，可能具有更高的数据速率 — 一个连到28800 bps调制解调器的串行口可能比打字员快1000倍或更多。在这样的速度，在两个时钟滴答之间，调制解调器大约收到48个字符，但是考虑到在调制解调器链路上的数据压缩，连到调制解调器上的串行口必须能处理至少两倍的字符。对于串行线，MINIX提供1024个字符的缓冲。

我们有一些遗憾。终端任务没有完全按通用设计原理实现，而是采取了一种折衷方案，但我们使用的方法没有过多地增加软件复杂性，性能也无损失。另一种明显的方法是抛弃会合原则，使系统缓冲所有没有被等待的消息，但这将更复杂，速度也更慢。

实际系统的设计者必须经常面对两种技术间的折衷，一种是采用通用的方案，这种方案在所有时候都很完美，但有时速度较慢；另一种是采用简单技术，一般速度很快，但是在一两种情况下，需要一些技巧以使其正常工作。在给定的情况下，哪种方式最好，经验是唯一真正的指导。Lampson （1984）和brooks（1975）总结了大量设计操作系统的经验，虽然有点过时，但是这些内容还是颇具经典性的。

在结束终端输入概述时，总结一下当终端任务被第一个读请求激活和接收到键盘输入以后被再次激活时发生的事件（参见图3－38）。在第一种情况下，当消息进入终端任务请求从键盘读字符时，主过程tty_task（11817行）调用do_read（11891行）来处理这个请求，如果没有足够多的缓冲字符，do_read把调用参数存入tty_table的键盘项中。

然后它调用in_transfer（12303行）取得任何已在等待的输入，再调用handle_events（12256行），该函数接着调用kb_read（13165行），并再次调用in_transfer试图再取得几个字符。kb_read调用了几个在图3－38中没有显示的例程来完成它的工作，其结果是把立即可得到的都拷贝给用户，如果什么也得不到，就什么也不拷贝。如果in_transfer或handle_events完成了读操作，当所有字符已被传输时向文件系统发一消息，因此文件系统可以使调用者解除阻塞。如果读还未完成（没有字符或无足够多的字符），do_read报告文件系统，告诉它或者挂起初始调用者，或者对对非阻塞请求，取消该读请求。

图3－38的右边总结了键盘中断以后终端任务被唤醒时发生的事件，当键入一个字符时，中断过程kb_hw_int把接收到的字符码放入键盘缓冲，设置一个标志指示控制台设备经历了一个事件，然后安排在下一次时钟滴答时发生超时。时钟任务向终端任务发一条消息，告诉它事件发生了。一旦接收到这条消息，tty_task检查所有设备的事件标志，为每一个标志已设置的设备调用handle_events。如果是键盘，就象接收到初始读请求一样，handle_events调用kb_read和in_transfer。在FS处接到第一条消息后，图右侧所示的事件可能发生若干次，直到有足够的字符能满足do_read所接收的请求为止。如果FS在第一个请求结束之前试图启动同一设备的另一个请求以读取更多字符，则将返回一个错误。当然，各个设备都是独立的；在远程终端上的用户的读请求和控制台上的用户的读请求是分开进行处理的。

图3-38　终端驱动程序的输入处理。树的左分枝表示处理一个读字符请求。右分枝表示向驱动程序发送了一条“字符已被输入”消息。

图3-38中未给出的由kb_read调用的函数包括：把硬件产生的键码（扫描码）转换为ASCII码的map_key，和跟踪修饰键（如SHIFT键）状态的make_break，以及处理各种复杂情况如用户试图用退格覆盖错误输入，其他特殊字符，和不同输入模式下的可用选项等情况的in_process。in_process也调用echo（第12531行），所以键入的字符也会显示在屏幕上。

终端输出

一般来说，控制台输出比终端输入简单，这是因为操作系统拥有控制权，不需要考虑意外的输出请求。另

外，由于MINIX控制台是内存映象显示器，所以向控制台输出就显得特别简单。输出的基本操作是把数据从一个内存区拷贝到另一个内存区，不需要任何中断。另一方面，显示管理的所有细节工作，包括转义序列的处理，都必须由驱动软件处理。和上一节讲述键盘输入时一样，我们将跟踪向控制台发送字符时涉及的每个步骤。我们假设在例子中正被写入的是当前活动显示器；如果考虑虚拟控制台设备，情况将有一点复杂。我们将在以后讨论这种情况。

一个进程在试图显示时一般要调用printf。printf调用WRITE向文件系统发送一条消息。该消息包含一个指向待显示字符序列的指针（不是字符序列本身）。接着文件系统向终端驱动程序发送一条消息，终端驱动程序取出字符并拷贝到视频RAM中。图3-39绘出了输出时涉及到的主要过程。

图 3－39　终端输出中使用的主要过程。虚线表示cons_write把字符直接拷贝到ramqueue。

当终端任务收到一条请求在屏幕上输出的消息时，它调用do_write（第11964行）来存储tty_table中控制台的tty结构参数。然后handle_events（只要tty_events标志被置位，就调用这个函数）被调用。每次调用时这个函数都会调用由它的参数指定的设备的输入和输出例程。在讨论控制台显示时，这意味着任何处于等待状态的键盘输入将被首先处理。在有输入等待时，则待回显字符被加到任何正在等待输出的字符后面。然后执行一个对cons_write的调用（第13729行）。cons_write是一个内存映象显示过程，它使用phys_copy把字符块从用户进程拷贝到局部缓冲区。由于局部缓冲区只有64字节，这个过程和以下各步骤可能要重复若干次。当局部缓冲区满时，每个8位字节被传送到另一个缓冲区ramqueue中。ramqueue是一个16位的字数组，交替间隔地用屏幕属性字节的当前值填充。屏幕属性字节决定了字符的前景色、背景色以及其他属性。如果可能，字符就被直接传送到ramqueue中；但某些字符如控制字符或转义序列中的字符，需要经过特殊处理；当一个字符的屏幕位置超出了屏幕的实际宽度，或者ramqueue已满时，也需要进行特殊处理。在这些情况下就要调用out_char（第13809行）来传送字符并执行各种相应的附加动作。例如，在定位屏幕的最后一行时，如果接收到一个换行符，那么scroll_screen（第13869行）将被调用；而对于一个转义序列，系统将调用parse_escape来处理这些字符。一般地，out_char调用flush（第13951行）把ramqueue中的内容拷贝到视频显示内存，其中使用了一个名为mem_vid_copy的汇编语言例程。在最后一个输出字符被送入ramqueue中后也要调用flush以确保所有要输出的字符都被显示出来。Flush的最终执行结果是使6845视频控制器在正确的位置上显示光标。

从逻辑上讲，从用户进程取出的字符可以在每次循环迭代中逐个写入视频RAM。然而，在Pentium类处理器的保护存储环境中，把在ramqueue中等待的字符用mem_vid_copy成块地拷入视频RAM的效率要高得多。有趣的是，这项技术却是在早期运行在没有存储保护的处理器上的MINIX版本中引入的。早期的mem_vid_copy必须考虑定时问题——老式的视频卡只允许在CRT电子束垂直回扫屏幕为空时写视频存储器，以避免屏幕上出现杂乱无章的“雪花”现象。由于性能牺牲过大，MINIX现在已不再支持这些过时的设备。不过，成块拷贝ramqueue的思想却使现代的MINIX在其他方面得到了益处。

控制台可以使用的视频RAM区由console结构中的c_start和c_limit域限定。光标的当前位置存放在c_column和c_row域中。坐标原点（0，0）位于屏幕的左上角，硬件由这里开始填充屏幕。每次视频扫描从c_org中指定的地址开始，连续扫描80×25个字符（4000个字节）。也就是说，6845芯片从视频RAM的偏移c_org处取出一个字显示在屏幕左上角（0，0）处，由属性字节控制字符的颜色和闪烁等；然后6845取出下一个字显示在（1，0）处。这个过程一直持续到到达（79，0）处，然后从屏幕的第二行（0，1）处重新开始。

计算机刚启动时，屏幕被清空，写入视频RAM的输出字符从视频RAM区的偏移c_start处开始存放；c_org被赋予和c_start相同的初值。这样第一行就显示在屏幕的顶部。当第一行满或out_char检测到一个换行符，需要开始一个新行时，输出被写到偏移量等于c_start加80处。当25行都被写满时，就需要进行卷屏操作。有些程序也需要进行向下卷屏操作。例如一个文本编辑器在光标已位于最顶部并仍然向上运动时，就需要把整个文本块向下移动。

卷屏的方式有两种。在软件卷屏方式下，显示在（0，0）位置上的字符永远被写入视频存储器的第一个位置，c_start指向相对位置为0的字的位置。通过赋予c_org和c_start相同的值使得视频控制芯片从这个位置开始显示。当屏幕需要卷动时，视频RAM中相对位置80处的字，也就是第二行的第一个字符，被拷入相对位置0处；相对位置81处的字被拷入相对位置1处；等等。而扫描顺序没有改变，仍然把视频存储器位置0处的数据显示在屏幕位置（0，0）处。这样屏幕上就产生了向上卷动一行的效果。这种方式下CPU的开销是需要移动80×24=1920个字符。在硬件卷屏方式下，数据本身并不在存储器中移动，而是通过改变视频控制器的起始扫描位置使它从视频RAM中的不同的位置开始显示，比如可以设置为从第80个字开始显示。设置的过程是把c_org的内容加上80，把所得结果保存起来，并把这个值写入视频芯片中相应的寄存器。这种方式要求要么控制器有足够的智能在视频RAM区中卷动，在到达底端（由c_limit指定的地址）时能回卷到视频RAM的起始地址（由c_start指定）读取数据；要么拥有比存储正好一屏内容所必需的80×2000个字更多的视频RAM。老式的显示适配卡显示内存较少，但控制器能够回卷进行硬件卷屏；较新的适配卡一般有比一屏文本的容量大得多的显示内存，但控制器不能回卷。这样，一块拥有32768字节显示内存的适配卡就可以存储204行每行160字节的数据，并且在遇到不能回卷的问题前可以执行179次硬件卷屏。但是最终仍然需要一次内存拷贝操作把最后的24行的数据移到视频内存的位置0处。不管采用那种方法，都要向视频RAM中拷入一空行以保证在屏幕最底部的新行是空行。

如果配置了虚拟控制台，那么一块适配卡上可用的存储器将被各个控制台平均分配，每个控制台设备使用

的范围由各自的c_start和c_limit域设定。配置虚拟控制台将对卷屏产生影响。对于拥有足够的存储器支持虚拟控制台的适配卡，尽管从概念上讲仍然可以使用硬件卷屏，但实际上使用更多的是软件卷屏。每个控制台可用的存储器越少，软件卷屏就使用得越频繁。如果配置了最大数目的虚拟控制台，就达到了极限状态。这时所有的卷屏操作都由软件完成。

相对于视频RAM起始地址的光标位置可以由c_column和c_row计算出来。但单独为光标位置设置一个c_cur域会更快些。当需要显示一个字符时，就把它写入视频RAM的c_cur处，然后更新c_cur和c_column。图3-40总结了console结构中影响当前位置和起始显示原点的域。

图 3－40 console结构中与当前屏幕位置相关的域。

对影响光标位置的字符（如换行符，退格符等）进行处理时需要调整c_column，c_row 和c_cur的值。这些工作由flush在结束前执行一个名为set_6845的调用来完成。这个调用将设置视频控制器中相应的寄存器。

终端驱动程序支持转义序列使得编辑器和其他交互式程序可以用灵活的方式来更新屏幕。所支持的序列是ANSI标准的一个子集，这个子集应该是完备的，使得为其他硬件和操作系统编写的程序能方便地移植到MINIX上。转义序列分为两类：一类不包含可变参数，另一类则可能包含参数。第一类中MINIX支持的唯一的代表是ESC M，它反向搜索屏幕，并且把光标上移一行；如果光标已经位于第一行，就把屏幕下卷一行。另一类转义序列可以包含一个或两个数值参数。这一类中所有的序列都以ESC [开头，其中“[”是控制序列引入符。图3-36给出了ANSI标准定义的可由MINIX识别的转义序列表。

解析转义序列并不简单。在MINIX中有效的转义序列可以只有两个字符，如ESC M；而一个可以接受二个两位数的数值参数的序列则可以有8个字符长，如ESC [20;60H，它使得光标移到第20行，第60列。在一个接受一个参数的序列中，参数可能被省略；而在接受两个参数的序列中，可能会省略一个参数，或者两个都省略掉。当有一个参数被省略或超出了有效范围时，就用默认值替换。默认值等于最低有效值。

以下是构造一个把光标移到屏幕左上角的转义序列的几种方法：

1. ESC [H是可以接受的，因为在没有输入参数时，就用最低有效值代替。
2. ESC [1;1H正确地把光标移动到第1行，第1列处。
3. ESC[1;H和ESC [;1H都省略了一个参数，和第一个例子中一样，用默认　　值为1替换。
4. ESC [0;0H也具有同样的作用，因为每个参数都小于最小有效值，用最小值代替。

这里给出这些例子并不是推荐任意地使用无效参数，而是想说明解析这些序列的代码并不简单。

MINIX使用了一个有限状态自动机解析转义序列。console结构中的c_esc_state变量在正常状态下值为0。当out_char检测到一个ESC字符时，c_esc_state的值被置为1，后面的字符由parse_escape（第13968行）进行处理。如果后继字符是控制序列引入符，就进入状态2；否则认为序列已结束，调用do_escape（第14045行）。在状态2，只要后继字符是数值字符，就把参数计算过程中上一次的参数值（初始值为0）乘以10，加上当前字符的数值作为当前参数值。参数值保存在一个数组中，如果检测到一个分号，处理过程就移到数组的下一个单元中（MINIX中的数组只有两个元素，但原理是一样的）。如果遇到一个不是分号的非数值字符，就判断序列已结束，接着同样调用do_escape。在do_escape的入口处，根据当前字符决定采取什么动作和如何解释参数，是使用默认值还是字符流中输入的参数。图3-48详细地绘出了整个过程。

可装载的键位映射表

IBM PC键盘并不直接产生ASCII码。从标准PC键盘的左上角开始，每个键依次用一个数来标识——“ESC”键为1，“1”键为2，依次类推。这样每个键都被赋予一个数，包括修饰键，如左SHIFT键为42，右SHIFT键为54。当按下一个键时，MINIX把接受到的对应的数作为扫描码。当松开一个键时也产生一个扫描码，但松开时产生的扫描码最高位被置位（即该键对应的数加上128）。这样就可以分辨出一个键被压下和松开。通过检测修饰键是否被压下并且仍未松开可以产生大量的组合键。在一般的场合，两键的组合对于双手打字来说最容易操作，如SHIFT-A或CTRL-D。但在某些特殊场合也使用三键组合（或更多）。比如CTRL-SHIFT-A或众所周知的PC用户用来复位并重新启动系统的CTRL-ALT-DEL组合键。

PC键盘十分复杂，为用户提供了很大的灵活性。一个标准键盘定义了47个普通字符键（26个字母，10个数字，11个标点）。如果我们愿意使用类似CTRL-ALT-SHIFT这样的三键组合，那么我们可以支持有376（8×47）个成员的字符集。但这是在不加任何限制的情况下，现在我们假定不区分左右修饰键，也不使用数字键盘和功能键。实际上，我们并不只限于使用CTRL，ALT和SHIFT作为修饰键；如果需要编写一个支持这种系统的驱动程序，我们可以从普通键集合中去掉一些键，把它们作为修饰键。

支持这种键盘的操作系统根据按下的键和有效的修饰键，用键位映射表（keymap）决定把什么字符码传给程序。MINIX的键位映射表可以看作一个128行，8列的数组，行代表可能的扫描码值（选择这个行数是为了容纳日文键盘；美国和欧洲键盘没有这么多键），各列分别代表无修饰键，SHIFT键，Control加SHIFT键，左ALT键，右ALT键，和ALT与SHIFT键的组合。用这种方式可以产生720个（（128-6）×6）字符编码，能定义一个完备的键盘。这就要求表中的每个表目都是16位长。对于美国键盘，ALT列和ALT2列是一样的。在其他语言的键盘上，ALT2也被称为ALTGR，许多键位映射表使用这个键作为修饰键支持3符号的键。

在编译MINIX内核时包含了一个标准键位映射表（由keyboard.c中的行

```
#include keymaps/us-std.src
```

决定），但也可以用

ioctl(0, KIOCSMAP, keymap)

把其他不同的键位映射表装入到内核地址keymap处。一个完整的键位映射表占用1536个字节（128×6×2）。其他附加的映射表以压缩格式存储。一个程序可以调用genmap来生成一张新的压缩的映射表。在编译时，genmap为某个特定的映射表包含keymap.src代码，这样这张映射表就被编译到genmap中。正常情况下，genmap在编译之后立即执行，把生成的映射表以压缩格式输出到一个文件中；然后二进制的genmap被删除。命令loadkeys读入一个压缩的映射表，把它在内存中展开，然后调用IOCTL把映射表传送到内核存储区。MINIX可以在启动时自动执行loadkeys，也可以由用户在任何时刻调用它。

图 3－41　一个键位映射表源文件中的若干表目。

一个映射表的源代码中定义了一个很大的已初始化的数组。为了节省篇幅，这里不能用源码形式列出一个映射表文件。图3-41用表格形式给出了src/kernel/keymap/us-std.src中若干行的内容，说明了映射表的若干方面。IBM PC键盘上没有键产生值为0的扫描码。扫描码为1，即ESC键对应的表目说明压下SHIFT和CTRL键时返回值不变，但ALT和ESC同时压下时将返回一个不同的值。编译到表中各行里的值是由include/minix/keymap.h中定义的宏决定的：

```
#define C(c)    ((c)&0x1F)          /*Map to control code*/
#define A(c)    ((c) | 0x80)          /*Set eight bit(ALT)*/
#define CA(c)   A(C(c))             /*CTRL_ALT*/
#define L(c)    ((c) | HASCAPS)     /*Add "Caps Lock has effect" attribute*/
```

前三个宏对被引用字符的扫描码中的位进行操作，把生成的所需的扫描码返回给应用程序。最后一个宏在16位值的高字节中设置HASCAPS位。这个标志位指示需要检测大写锁定变量，并且扫描码在返回前可能需要修改。在图中，扫描码为2，13和16的表目说明了典型的数字，标点和字母键是如何被处理的。对扫描码28可以观察到一个不同的特征——ENTER键一般产生扫描码CR（0x0D），这里用C(`M`)表示。由于在UNIX文件中新行符的编码是LF(0x0A),并且有时需要直接输入新行符,所以这个映射表提供了一个CTRL-ENTER组合来产生这个编码,C(`J`)。

扫描码29对应一个修饰键码，并且无论是否还压下了其他什么键它都应该能被识别出来，因此总是返回CTRL。按下功能键不返回普通的ASCII码值。扫描码59对应的行中用符号（在include/minix/keymap.h中定义）给出了F1键和其他修饰键组合时的返回值。这些值是F1: 0x0110，SF1: 0x1010，AF1: 0x810，ASF1: 0x0C10和CF1: 0x0210。表中的最后一个表目对应扫描码127，在映射表数组末尾附近的表目中十分典型。绝大多数在欧洲和美洲使用的键盘没有那么多的键对应所有可能的扫描码，表中的这些表目都用0填充。

可装载字体

早期的PC只能用存储在ROM中的字符点阵在屏幕上显示字符。现代PC系统中使用的显示器的视频显示适配卡上则提供了可以装入定制的字符发生器（character generator）的RAM。MINIX提供了一个IOCTL操作

ioctl(0, TIOCSFON, font)

来支持定制的可装入字符发生器。MINIX支持80行×25列的显示模式，字体文件包含4096个字节。每个字节代表一行8个像素，如果某一位为1，对应的像素就被点亮，映射一个字符需要16个这样的行，即16个字节。不过视频适配卡中使用了32个字节来映射一个字符，用来在MINIX目前还不支持的模式中提供更高的分辨率。MINIX提供了命令loadfont把这些字体文件转换到由IOCTL调用引用的8192字节的font结构中。和键位映射表一样，字体可以在启动时加载，也可以在正常操作中的任何时候加载。不过，任何视频适配器都有一个存储在ROM中的标准字体作为可使用的默认字体。MINIX并不需要把一个字体编译到它本身里，内核中唯一需要的对字体的支持是进行TIOCSFON IOCTL操作的代码。

3.9.4　设备无关终端驱动程序的实现

在这一节中我们将要仔细地研究终端驱动程序的源代码。在研究块设备时，我们已经看到支持几个不同设备的多个任务可以共享一套共同的基本软件。终端设备的情况也是类似的，不同的是这里一个终端任务要支持几个不同类型的终端设备。现在我们从设备无关代码开始。后面几节我们再研究键盘和存储映象控制台显示的设备相关代码。

终端任务数据结构

文件tty.h中包含了在实现终端驱动程序的C文件中使用的定义。这个文件中定义的绝大多数变量都具有前缀tty_。在glo.h中还说明了一个EXTERN变量tty_timeout，由时钟和终端中断处理程序使用。

tty.h中定义的标志O_NOCTTY和O_NONBLOCK在文件include/fcntl.h中已经定义过了。在这里再重写一次是为了避免需要再包含一个文件。devfun_t和devfunarg_t类型用来定义指向函数的指针，以提供和我们在磁盘驱动程序代码的主循环中看到的相类似的间接调用机制。

tty.h中最重要的部分是tty结构的定义（第11614行到11668行）。每个终端设备都有一个这样的结构与之对应（控制台显示和键盘看作一个单一的终端）。tty结构中的第一个变量tty_events被用作一个标志，在某个中断引起变化需要终端任务来处理设备时被置位。在设置这个标志时，系统还对全局变量tty_timeout进行操作以通知时钟中断处理程序在下一个时钟滴答唤醒终端任务。

tty结构中的其余部分分别按照处理输入、输出、状态和关于未完成操作的信息等功能组织在一起。在输入

部分，tty_inhead和tty_intail定义了缓冲接收到的字符的队列。Tty_incount对队列中的字符个数计数，tty_eotct对字符的行数计数，这将在下面解释。除了用来建立间接调用指针的终端初始化例程以外，所有针对特定设备的调用都是间接调用。tty_devread和tty_icancel域中保存了指向执行读和输入取消操作的设备指定代码的指针。tty_min用来和tty_eotct比较，当后者与前者一致时，就认为一个读操作结束。在规范输入模式中，tty_min被置为1，用tty_eotct对输入的行数计数。在非规范输入模式中，用tty_eotct对字符计数，tty_min则由termios结构中的MIN域决定。这样，根据所处的模式，通过比较两个变量可以判断出一行是否准备好或者何时达到最小字符计数。

Tty_time保存定时器的值，这个值决定了终端任务应该何时被时钟中断处理程序唤醒，tty_timenext是用来把活动的tty_time域链接成一个链表的指针。只要设置了定时器，这个链表就被重新排序，因此每次时钟中断处理程序只需要检查表头即可。MINIX可以支持许多远程终端，在任意时刻可能只对几个终端设置了定时器。比起检查tty_table中的每个表目，使用活动定时器链表可以使时钟处理程序的工作简单一些。

由于输出排队是由设备指定代码处理的，所以tty的输出部分没有变量说明，全部由指向写、回显、送中断信号和取消输出等设备指定函数的指针组成。在状态部分，标志tty_reprint、tty_escaped和tty_inhibited说明最后一个字符具有特殊含义；例如，当遇到一个CTRL-V（LNEXT）字符时，tty_escaped被置为1指示下一个字符的任何特殊含义应被忽略。

结构中的下一部分保存了DEV_READ，DEV_WRITE和DEV_IOCTL操作的进度数据。每个操作中包含了两个进程。管理系统调用的服务器（正常时为FS）在tty_incaller（第11644行）中指定。服务器为另一个需要进行I/O操作的进程调用tty任务，这个客户进程在tty_inproc（第11645行）中指定。如图3-37所示，在执行一个READ时，字符直接由终端任务传送到初始调用者的内存空间中的缓冲区。缓冲区由tty_inproc和tty_in_vir定位。接下来的两个变量tty_inleft和tty_incum对仍须传送和已传送的字符计数。一个WRITE系统调用也需要类似的变量集。对于IOCTL操作可能有在请求进程和任务之间的直接数据传送，因此需要定义一个虚拟地址，但不需要标志操作进度的变量。一个IOCTL请求可能被延迟，比如直到当前输出完成；但在时间合适时，请求一次操作即可完成。最后，tty结构中还包含了一些不在其他集中的变量，包括指向在设备级处理DEV_IOCTL和DEV_CLOSE的函数的指针，一个POSIX格式的termios结构，一个支持面向窗口屏幕显示的winsize结构。结构的最后部分提供了存储输入队列本身的数组tty_inbuf。注意这是一个u16_t数组，而不是8位char型字符数组。虽然应用程序和设备使用8位编码的字符，C语言却需要输入函数getchar能使用更大的数据类型以便能返回在所有的256个可能的值之外的符号值EOF。

tty_table，即tty结构的数组，是用EXTERN宏说明的（第11670行）。每个被include/minix/config.h中的定义NR_CONS，NR_RS_LINES和NR_PTYS使能的终端都有一个元素与之对应。在本书讨论的配置中，有两个控制台被使能，但MINIX可以被重新编译以增加到两条串行线和64个伪终端。

tty.h中还有一个EXTERN定义。tty_timelist（第11690行）是由定时器使用的保存tty_time域链表头的指针。许多文件包含了tty.h头文件，tty_table和tty_timelist的存储空间在编译table.h时分配，这与glo.h中定义的EXTERN变量是类似的。

在tty.h的最后定义了两个宏buflen和bufend。终端任务代码经常使用这两个宏把数据拷入和拷出缓冲区。

设备无关终端驱动程序

主要的终端任务和设备无关的支持函数都在tty.c中定义。由于任务支持许多不同的设备，所以在一个特定的调用中需要使用次设备号来辨别那一种设备是被支持的，这些都定义在11760行到11764行中。接下来的是一些宏定义。如果一个终端没有初始化，那么指向该设备的设备指定函数的指针将被C编译器置为0。这样就可以定义一个tty_active宏（第11774行），在发现一个空指针时返回FALSE。当然，如果设备的初始化代码的一部分工作是初始化指针使间接存取成为可能，那么它本身是不能被间接存取的。11777行到11783行定义的条件宏使得对RS-232或伪终端设备的初始化调用在没有配置这些设备时都同样地调用一个空函数。在这个部分do_pty也可以类似地被禁止。这样，如果不需要某些设备，就可以把这些设备对应的代码完全省略掉。

由于每个终端可配置的参数很多，并且在一个网络系统中可能有许多终端，所以第11803到11810行说明了一个termios_defaults结构，并用默认值初始化。当需要初始化或重新初始化一个终端时，就向该终端的tty_table中拷贝一个这样的结构。图3-33中给出了特殊字符的默认值。图3-42给出了各种标志的默认值。接下来的行类似地说明了winsize_defaults结构。它应由C编译器全部初始化为0。这是一个合适的默认动作，因为它表示“窗口大小未知，使用/etc/termcap。”

图 3－42　默认的termios标志值。

终端任务的入口点为tty_task（第11817行）。在进入主循环之前，对每个已配置的终端调用一次tty_init（在第11826行的循环中），然后显示MINIX启动信息（第11829行到11831行）。虽然从源码中看到的是对printf的调用，但在编译时使用了把对printf库例程的调用转化为对printk的调用的宏。printk使用了控制台驱动程序中的一个叫作putk的例程，所以没有涉及到FS。这条消息只发往主控制台显示器，不能被重定向。

第11833行到11884行的主循环原理上与其他任务的主循环是一样的——它接收一条消息，根据消息类型执行一条switch语句调用适当的函数，并产生一条返回消息。但是这里有一点复杂。首先，许多工作是由低层的中断例程完成的，特别是在处理终端输入时。在上一节中我们看到，从键盘输入的各个字符要被接收并缓冲，

而不是为每个字符向终端任务发一条消息。这样，在试图接收一条消息之前，主循环总是扫描整个tty_table，检查每个终端的tp->tty_events标志，如果必要就调用handle_events（第11835行到11837行）来处理未完成的事务。只有在没有需要立刻处理的事件时才产生一个调用来接收消息。如果接收到的消息来自硬件，就执行一条continue语句跳过这一次循环，重新检查事件。

其次，这个任务可能为几个设备服务。如果接收到的消息来自硬件中断，那么通过检查tp->tty_events标志可以确认需要服务的设备。如果消息不是来自硬件中断，那么消息中的TTY_LINE域可以用来确定是由哪个设备发出的。次设备号通过一系列的比较被解码，借助于设备号tp指向tty_table中正确的表目（第11845行到11864行）。如果该设备是一个伪终端，那么就调用do_pty（在pty.h中）并且重新开始主循环。在这种情况下，do_pty生成应答消息。当然，如果伪终端没有被使能，那么对do_pty的调用将使用前面定义的空的宏。人们希望试图存取不存在设备的情况不会发生，但加上一个检查过程要比确保系统中任何地方都不会出错容易得多。在设备不存在或没有配置时，就产生一条ENXIO错误消息，并且控制返回到循环的顶部。

任务的其余部分和我们在其他任务的主循环部分看到的相类似，根据消息的类型进行一个switch选择（第11874行到11883行），调用请求对应的函数，如do_read，do_write等等。每个被调用的函数产生回答消息，而不是把构造消息所需要的信息传回主循环。只有在一个有效的消息类型没有被收到时，主循环的结束部分才产生一条EINVAL错误信息。因为回答消息是在终端任务内部许多不同地方发出的，所以可以调用一个公共的例程tty_reply来处理构造回答消息的细节。

如果tty_task收到的消息是一个有效的消息类型而不是一个中断的结果，也不是来自一个伪终端，那么主循环最后的switch将把它分发给函数do_read，do_write，do_ioctl，do_open，do_close和do_cancel中的一个。每个函数调用的参数是指向tty结构的指针tp和消息的地址。在分别考察这些函数之前，我们要先进行一些一般性的讨论。由于tty_task可能要为多个终端设备服务，所以这些函数必须很快地返回以便主循环能继续响应其他请求。然而，do_read，do_write和do_ioctl可能无法立即处理完所有的请求。为了允许FS为其他调用服务，需要立即响应。如果请求不能立即结束，那么就在回答消息的状态域中返回一个SUSPEND码。这对应于图3-37中的消息（3），它挂起开始调用的进程，同时释放FS。图中消息（7）和（8）在操作结束后被发送。如果请求可以被完全满足，或者发生了一个错误，那么就在返回给FS的消息的状态域中返回传送的字节数或错误码。在这种情况下FS立即向进行原始调用的进程发送一条消息，唤醒该进程。

读终端与读磁盘设备是完全不同的。磁盘驱动程序向磁盘硬件发出一条指令，数据就被逐渐返回，除非发生机械或电子故障。计算机可以在屏幕上显示一条提示信息，但却无法强迫一个人坐在键盘前开始输入。因此，计算机根本无法保证有人坐在那儿。为了能够尽快得到返回，在有输入到达时，do_read（第11891行）一开始就会保存使得请求可以稍后被完成的信息。首先要进行一些错误检查。当设备仍然在等待输入以满足前一个请求，或消息中的参数无效时，将发生一个错误。如果通过了检查，第11911行到11915行就把关于请求的信息拷贝到设备的tp->tty_table表目中合适的域。下一步把tp->tty_inleft设置为所需的字符数非常重要。这个变量用来决定读请求何时得到满足。在规范模式中，tp->tty_inleft随着每个返回的字符逐渐减少，直到接收到一行的结束时，这个值突然减少为0。在非规范模式中，处理有所不同，但在任何情况下，只要调用被满足tp->tty_inleft就被置为0，不论是由于超时还是接收到了要求的最小字符数。当tp->tty_inleft达到0时，一条回答消息被送出。如同我们将要看到的那样，回答消息可以在好几个地方产生。有时还需要检查一个读进程是否仍在等待一个回答，非0时的tp->tty_inleft可以用作这个目的。

在规范模式中，一个终端设备一直处于等待直到已接收到调用中需要的字符数，或达到了一行或一个文件的末尾。第11917行通过检查termios结构中的ICANON位判断终端是否处于规范模式。如果这一位没有置位，就检查termios的MIN和TIME值以决定采取什么动作。

在图3-35中我们可以看到MIN和TIME是怎样相互作用以提供一个读调用可能采取的不同动作。TIME在第11918行中检测。0值对应于图3-35中的左边一列，在这种情况下不需要进一步的检测了。如果TIME不为0，那么就检测MIN。如果为0，就调用settimer启动定时器，即使没有收到一个字节，在延迟一段时间后也将结束DEV_READ请求。这里tp->tty_min被置为1，使得在超时前如果接收到一个或几个字节就立即终止调用。这里还没有对可能的输入进行检查，所以可能有不止一个字符在等待满足请求。在这种情况下，发现输入后将立即返回READ调用中指定个数的字符。如果TIME和MIN都不为0，那么定时器将有不同的含义。在这种情况下，定时器用来作为字符间的定时器。它只在收到第一个字符后启动，并在接收到每个后继字符之后重新启动。tp->tty_eotct在非规范模式下对字符计数，如果在第11931行中为0，那么就还没有接收到字符，并且字节间定时器被禁用。lock和unlock用来保护所有对settimer的调用，防止在settimer运行时发生时钟中断。

在任何情况下，第11941行都调用in_transfer把所有已在输入队列中的字节直接传送到读进程中。接着进行一个handle_events调用，这个调用可能向输入队列中送入更多的数据，然后再次调用in_transfer。这里需要解释一下这个显得重复的调用。虽然到目前为止所有的讨论都从键盘输入的角度来考虑，但do_read是在代码的设备无关部分，也要为通过串行线连接的远程终端的输入服务。上一次的输入可能已经填满了RS-232输入缓冲区从而导致输入被禁止。对in_transfer的第一次调用不能重新启动输入流，但对handle_events的调用可以起到这个作用。由此引起的对in_transfer的第二次调用并没有实质性的作用，主要是为了确保远程终端再次被允许发送。这些调用都可能满足请求并向FS回答消息。tp->tty_inleft被用作一个标志来判断回答是否已发送；

如果在11944行中它仍为0，那么do_read就自己产生并发送回答消息。这些由第11949行到11957行完成。如果原始调用指定了一个不可阻塞读，FS将被通知向原始调用发送一个EAGAIN错误码。如果调用是一个普通的可阻塞读，FS将收到一个SUSPEND码，释放FS但通知它保持原始调用者阻塞。在这种情况下，终端的tp->tty_inrepcode域被置为REVIVE。如果READ稍后被满足，这个值将被放在给FS的回答消息中以指示原始调用被置为睡眠态需要唤醒。

Do_write（第11964行）和do_read很相似，但更简单一些，因为在处理一个WRITE系统调用时需要考虑的情况较少。用和do_read中相类似的检查来检测上一次写是否不在进行以及参数是否有效，然后把请求的参数拷贝到tty结构中。接着调用handle_events，并检查tp->tty_outleft以判断是否完成（第11991行到11992行）。如果是，那么handle_events已经发送了一条回答消息，不需要再做什么。如果没有，那么就产生一条回答消息，消息参数依赖于原始的WRITE调用是否是不可阻塞模式。

图 3－43　POSIX调用和IOCTL操作。

下一个函数do_ioctl（第12012行）很长，但不难理解。do_ioctl的函数体是两条switch语句。第一个确定由请求消息中的指针指向的参数的大小（第12033行到12064行）。如果大小不为0，就检验参数的有效性。这里不能检测内容，但可以检查一个在指定地址处所需大小的结构是否能装入它被指定放置的段。函数的余下部分是根据IOCTL操作请求类型进行选择的switch语句（第12075行到12161行）。不幸的是，用IOCTL调用支持POSIX要求的操作意味着必须构造IOCTL操作的名字而不是复制POSIX要求的名字。一个TCGETS操作为用户的tcgetattr调用服务，并且只是简单地返回终端设备的tp->tty_termios结构的一份拷贝。下面的四个请求类型共享代码。TCSETSW、TCSETSF和TCSETS请求类型对应用户调用POSIX定义的函数tcsetattr，并且都有把一个新的termios结构拷贝到一个终端的tty结构中这样的基本动作。拷贝动作用一个phys_copy调用从用户进程取得数据，接下来在12098行到12099行调用setattr。对于TCSETS调用，拷贝动作立即执行；对于TCSETSW和TCSETSF调用，则可能在输出结束后执行。如果调用tcsetattr时使用了一个修饰符请求把动作延迟到当前输出完毕后执行，并且如果在第12084行中对tp->tty_outleft的检测表明输出没有结束，那么就把请求的参数放在终端的tty结构中供以后处理。tcdrain被翻译为一个TCDRAIN类型的IOCTL调用，它挂起一个程序直到输出结束。如果输出已经结束，那么它不再执行什么动作。如果没有，它也必须把信息保存在tty结构中。

POSIX的tcflush函数根据参数丢弃未读入的输入和/或未发送的输出。它被直接翻译为IOCTL调用，由一个对为所有终端服务的函数tty_icancel和/或由tp->tty_ocancel指向的设备指定函数的调用组成（第12102行到12109行）。tcflow也被类似地直接翻译为一个IOCTL调用。为了挂起或重置输出，它向tp->tty_inhibited中写入一个TRUE或FALSE，然后置位tp->tty_events标志。为了挂起或重置输入，它使用由tp->tty_echo（第12120行到12125行）指向的设备指定回显例程向远程终端发送适当的STOP（正常为CTRL-S）或START（CTRL-Q）码。

余下的由do_ioctl处理的大部分操作都由一行代码调用一个合适的函数处理。对于KIOCSMAP（装入键位映射表）和TIOCSFON（装入字体）操作要进行检验以保证该设备确实为控制台，因为这些操作不应用于其他终端。如果使用了虚拟终端，那么所有的控制台都将使用相同的键位映射表和字体，硬件不允许用其他的方法。窗口大小操作在用户进程和终端任务之间拷贝winsize结构。不过要注意TIOCSWINSZ操作对应的代码下面的注释。当一个进程改变了它的窗口大小时，在某些版本的UNIX下，内核应该向进程组发送一个SIGWINCH信号。POSIX标准中并不要求这个信号。但是任何想要使用这些结构的人应该考虑在这儿添加代码初始化这个信号。

do_ioctl的后两种情况支持POSIX要求的tcgetpgrp和tcsetpgrp函数。这两种情况没有对应的动作，只是永远返回一个错误。这并没有什么毛病。这些函数支持作业控制（job control），能够从键盘上挂起和重启一个进程。POSIX不要求作业控制，在MINIX中也不支持。不过，即使不支持作业控制，POSIX也需要这些函数以保证程序的可移植性。

do_open（第12171行）执行的是一个简单的基本动作——它增加设备的tp->tty_openct变量的值以便能够检验是否被打开。不过，在这之前首先要进行一些检测。对于普通终端，POSIX指定第一个打开终端的进程为会话主导进程。当一个会话领导终止时，将收回该组中的其他进程对该终端的存取权。监控程序需要能够输出出错信息，并且如果错误信息输出没有被重定向到一个文件中，那么它应当被送到一个不能被关闭的显示器上。为此MINIX中存在一个称为/dev/log的设备。在物理上它与dev/console是相同的设备，但它由一个独立的次设备号标识，处理方式也不同。它是一个只写设备，这样如果do_open试图以读方式打开它将返回一个EACCESS错误（第23283行）。do_open执行的另一个检测是测试O_NOCTTY标志。如果该标志没有被置位并且设备不是/dev/log，则该终端成为一个进程组的控制终端。这是通过把调用方的进程号放入tty_table表目的tp->tty_pgrp域完成的。然后，增加tp->tty_openct变量的值并发送回答消息。

一个终端设备可能被打开不止一次，而下一个函数do_close（第12198行）除了减少tp->tty_openct变量的值外不执行其他动作。第12198行的检测防止了该函数试图关闭设备/dev/log。如果这个操作是最后一次关闭操作，那么就调用tp->tty_icancel取消输入。由tp->tty_ocancel和tp->tty_close指向的设备指定例程也被调用。然后tty结构中的各个域被置回他们的默认值并发送回答消息。

最后一类消息由do_cancel处理（第12220行）。当一个试图读或写的被阻塞的进程接收到一个信号时将引起这个动作。这时需要检查三种状态：

1. 终止时进程可能在读。

2. 终止时进程可能在写。
3. 进程可能被tcdrain挂起直到它的输出结束。
对每种情况进行检测，并根据需要调用通用的tp->tty_icancel或由tp->tty_ocancel指向的设备指定例程。在最后一种情况下，唯一要做的就是重置tp->tty_ioreq标志来指示IOCTL操作现在已经完成。最后，tp->tty_events标志被置位并发送一条回答消息。

终端驱动程序支持代码

现在我们已经研究了tty_task中主循环调用的顶层函数，下面将要学习支持他们的代码。我们将从handle_events开始（第12256行）。前面提到，每执行一次主循环，都要检查每个终端设备的tp->tty_events标志，如果需要处理某个特定设备的事件就调用handle_events。do_read和do_write也调用handle_events。这个例程必须工作得很快。它重置tp->tty_events标志，然后调用由指针tp->tty_devread和tp->tty_devwrite（第12279行到12282行）指向的设备指定例程读和写。这些都是被无条件地调用的，因为这里无法检验一次读或写是否会引起标志的置位——这儿有一个设计上的选择，即对每个设备检查两个标志的代价要高于每次一个设备被激活时做两次调用的代价。另外，在大部分时间里从终端接收到的字符是需要回显的，所以两个调用都是需要的。如同在讨论处理do_ioctl调用的tcsetattr时所提到的那样，POSIX可能会推迟对设备的控制操作直到当前的输出结束，所以在调用设备指定函数tty_devwrite后立即处理ioctl操作是一个很好的时机。这些在第12285行中完成：如果有一个等待的控制请求，就调用dev_ioctl。

由于tp->tty_events标志由中断置位，并且字符可能由一个快速设备以很快的速度到达，因此可能出现在对设备指定的读和写例程以及dev_ioctl的调用结束时，另一个中断又使得标志置位的情况。因此将输入信息由缓存中中断例程放置的初始位置移出被赋予较高的优先权。这样只要在循环结束处（第12286行）检测到tp->tty_events标志置位，handle_events就重复地调用设备指定例程。当输入流停止时（也可以是输出，但输入更可能这样重复地请求），就调用in_transfer把字符从输入队列中传送到调用一个读操作的进程的内部缓冲区中。如果传送结果结束了请求，无论是传送了请求的最大字符数还是达到了一行的末尾（在规范模式中），in_transfer本身就发送一条回答消息。如果是这样，那么在返回到handle_events时tp->tty_left的值为0。这里还对传送的字符数是否达到请求的最小数目进行进一步检查并发送一条回答信息。对tp->tty_inleft的检测防止了重复消息的发送。

下一步我们研究in_transfer（第12303行），这个函数把数据从任务存储空间中的输入队列中移动到请求输入的用户进程的缓冲区中。然而，直接拷贝是不行的。输入队列是一个环型缓冲区，并且需要检查字符以判断是否已达到文件末尾，或者检查是否处于规范模式下，传送只持续到一行结束。另外，输入队列中各单元大小为16位，而接收缓冲区是一个8位字符的数组。这样就需要使用一个中间局部缓冲区。字符在放入局部缓冲区时被逐个检查，当缓冲区被填满或者输入队列被移空时，就调用phys_copy把局部缓冲区中的内容移到接收进程的缓冲区中（第12319行到12345行）。

tty结构中有三个变量tp->tty_inleft，tp->tty_eotct和tp->tty_min用于确定in_transfer是否需要完成一些任务，前两个变量用于控制主循环。前面已经提到，tp->tty_inleft一开始被设置为一个READ调用所请求的字符数。正常情况下，每传送一个字符，它就被减一。但是当到达一行的结束时，它会突然减少到0。只要tp->tty_inleft为0，就向读进程发送一条回答信息，因此该变量也可以作为一个标志指示是否已经发送回答消息。所以，在第12314行中，检测到tp->tty_inleft为0可以作为不发送回答消息就中止in_transfer的执行的充分理由。

检测的下一部分比较tp->tty_eotct和tp->tty_min两个变量。在规范模式中，这两个变量与完整的输入行有关；而在非规范模式中，它们与字符有关。只要一个“行中止”字符或一个字节被放入输入队列，就增加tp->tty_eotct的值。这样tp->tty_eotct就可以对已被终端任务接收但还未传给一个读进程的行或字节计数。tp->tty_min则指示了满足一个读请求所必须传送的最小的行数（规范模式）或字符数（非规范模式）。在规范模式中它的值总是为1，在非规范模式中，它可以是从0到MAX_INPUT（MINIX中为255）之间的任何一个值。行12314中的第二部分检测使得在规范模式下如果没有接收到一个整行，in_transfer就立即返回。传送并不马上执行，而是等到一行结束之后，这样队列的内容就可以被修改。例如，如果用户在按下ENTER之前接着输入ERASE或KILL字符，那么将丢弃队列中的内容。在非规范模式中，如果无法取得所需的最小字符数，就立即返回。

在几行之后，tp->tty_inleft和tp->tty_eotct被用来控制in_transfer的主循环。在规范模式中，传送一直持续到队列中不再有完整的行。在非规范模式中，tp->tty_eotct用来对等待的字符计数。tp->tty_min用来控制何时进入循环，但不用来决定何时结束。一旦进入循环，或者传送所有的字符，或者就传送原始调用所需的字符数，这取决于那一个更小。

图 3－44　字符放入输入队列时字符码中的域。

输入队列中的字符大小为16位。实际传给用户进程的是低8位。图3-44显示了高位字节是怎样使用的。三位被用来标志字符是否被转义（CTRL-V）、是否表示文件结束，或是否是表示行结束的几个代码之一。四位用来计数字符回显时占据的屏幕空间。第12322行检查IN_EOF位（图中D）是否被置位。检测是在内循环顶部进行的，因为文件结束符本身不传给用户进程，也不计入字符计数。在传送每个字符时，用一个掩码使高8位为0，只有低8位中的ASCII值被送到局部缓冲区（第12324行）。

标志输入结束的方法不止一种，但一般使用设备指定例程确定接收到的字符是否为换行符，CTRL-D，或者其他这样的字符。in_transfer只需要在第12340行检测IN_EOT位（图3-44中的N）这个标志。如果检测到，就减少tp->tty_eotct的值。在非规范模式中，每个字符在放入输入队列中时都这样被计数，并且每个字符都在这时候被标记IN_EOT位，所以tp->tty_eotct实际上就是队列中未移走的字符的计数。in_transfer的主循环在两种不同的模式下的唯一区别在第12343行。这里如果发现一个字符标志为行中止符，则tp->tty_inleft被置为0，但只有在规范模式下才这样做。这样当控制返回到循环的顶部时，在规范模式中循环在一个行中止符之后终止，但在非规范模式中行中止符被忽略。

当循环中止时，一般会有一个部分被填满的局部缓冲区等待传送（第12347行到12353行）。然后如果tp->tty_inleft达到0，那么就发送一条回答消息。在规范模式下总是这种情况，但如果处于非规范模式并且传送的字符数少于全部的请求，那么将不发送回答。如果你还记得我们在调用in_transfer时所看到的（在do_read和handle_events中），当in_transfer在返回时传送了多于tp->tty_min所指定的字符数时，在in_transfer调用之后的代码将发送一条回答消息，那么你会觉得很迷惑，因为现在就是这种情况。在我们讨论下一个函数时，你将会看到in_transfer为什么不无条件地发送回答消息。这个函数在一个不同的环境中调用in_transfer。

这个函数就是in_process（第12367行）。它由设备指定软件调用来执行对所有输入都要进行的共同处理。它的参数是一个指向tty结构的指针，一个指向待处理的8位字符数组的指针和一个计数器。计数器被返回给调用者。in_process是一个很长的函数，但它的动作并不复杂。它把16位的字符加入到输入队列中，这些字符稍后将由in_transfer处理。

in_transfer中提供了几种不同的处理方法。

1. 普通字符被加入到输入队列中，并扩展为16位。
2. 影响以后处理的字符，通过修改标志位来表示影响，但不放入队列。
3. 控制回显的字符立即生效，不放入队列。
4. 具有特殊标记的字符在放入缓冲区时把编码如EOT位加入它们的高字节。

让我们先来看一下在一个完全正常的环境中，如何从一个设置为标准MINIX默认属性的终端上输入的一条短句中间的一个普通字符，例如“x”（ASCII码0x87）。从输入设备接收到时，这个字符占用了图3-44中的0到7位。在第12385行，如果ISTRIP被置位，那么该字符的最高有效位第7位被重置为0，但MINIX中的默认值是不清除这一位，允许输入全部的8位。不过这不会影响我们的“x”。MINIX的默认值是允许输入的扩充处理，所以能够通过对tp->tty_termios.c_flag（第12388行）中IEXTEN位的检测。但接下来的检测在我们的假设之下不能通过：没有字符转义（第12391行）。输入本身不是字符转义字符（第12397行），并且该输入不是REPRINT字符（第12405行）。

下面几行的检测判断出字符既不是特殊的_POSIX_VDISABLE字符，也不是CR和NL字符。最后，一个正的结果：处于规范模式，这是正常的默认值（第12424行）。然而，我们的“x”既不是ERASE字符，也不是KILL、EOF（CTRL-D）、NL或EOL字符之一，所以直到第12457行，仍然没有对它进行任何处理。这里检测到IXON被默认置位，允许使用STOP（CTRL-S）和START（CTRL-Q）字符，但在接下来的测试中却没有发现匹配。在第12478行发现ISIG位被默认置位，允许使用INTR和QUIT字符，但同样找不到匹配。

实际上，对一个普通字符第一个值得注意的处理发生在第12491行，这里对输入队列是否满进行检测。如果已满，那么字符将在这里被丢弃，因为这时处于规范模式下，用户看不到它回显在屏幕上（continue语句丢弃该字符，因为它重新开始外循环）。然而，由于我们在说明时假设了完全正常的情况，所以我们假定缓冲区还没有满。下一个检测判断是否需要一个特殊非规范模式处理（第12497行），如果失败，就向前跳到第12512行。这里由于tp->tty_termios.c_flag中的ECHO位是默认置位的，所以调用echo显示字符。

最后，在第12515行到12519行，字符被放入输入队列。这时要增加tp->tty_incount的值，但是由于该字符为普通字符，没有EOT位标志，所以tp->tty_eotct不变。

如果刚送入队列的字符填满了队列，那么循环中的最后一行就调用in_transfer。然而，在一般条件下，对于这个例子我们假设in_transfer即使被调用也将不执行任何动作，这是因为（假设队列被正常服务，并且当前一行结束时，前一次输入被接受）tp->tty_eotct为0，tp->tty_min为1，在in_transfer开头的检测（第12314行）使得它立即返回。

在一个普通字符的情况下执行了一遍in_process后，让我们回到in_process的开头看一看在特殊一点的情况下将会如何。首先我们讨论转义字符，它允许一个普通字符把一种特殊影响传递给用户进程。如果转义字符有效，tp->tty_escaped将被置位，当检测到这个标志时（在第12391行），该标志立即被重置，并且IN_ESC位，即图3-44中的V位，被加到当前的字符中。这使得字符在回显时被特殊处理——转义控制字符用“^”加字符的形式显示使它们可见。IN_SEC位也防止字符被识别为其他特殊字符。下面几行处理处理转义字符本身，LNEXT字符（默认为CTRL-V）。当检测到LNEXT字符时，tp->tty_escaped标志被置位，并且调用两次rawecho输出一个“^”后面接一个退格符。这使得用户想起在转义字符有效的键盘上，当后继字符被回显时将覆盖“^”。LNEXT字符是影响后继字符的字符的一个例子（在这里，是紧接着的字符）。它不放在队列中，并且在两次对rawecho的调用后重新启动循环。这两次检测的次序很重要，这使得可以在一行中输入LNEXT两次而把第二个拷贝送给一个进程。

in_process处理的下一个特殊字符是REPRINT字符（CTRL-R）。当in_process发现接下来是一个reprint调用时（第12406行），将重新显示当前的回显输出。REPRINT本身被丢弃，对输入队列没有影响。

考虑每个特殊字符的处理细节是很冗长乏味的，并且in_process的源代码是很简单明了的。我们将只提几个地方。一个是在放入输入队列的16位值的高位使用特殊位使得区分有相似作用的一类字符显得很容易。比如，EOT（CTRL-D），LF和可选的EOL字符（缺省时没有定义）都由EOT位标志，即图3-44中的D位（第12447行到12453行），这使得后面的识别更容易了。最后，我们将说明前面提到的in_transfer的特别行为。虽然我们在前面看到在in_transfer调用中似乎在返回时总是产生一条回答消息，但实际上回答消息并不是每次中止时都产生。回忆当输入队列满时（第12522行），in_process对in_transfer的调用在规范模式下没有作用。但如果在非规范模式中，第12499行将为所有的字符打上EOT标志，这样每个字符都在第12519行中被tp->tty_eotct计数。在返回时，这将引起进入由于在非规范模式中输入队列满而引起的in_transfer调用的主循环。在这种场合当in_transfer终止时不应该发送消息，因为在返回in_process后很可能读入了过多的字符。实际上，虽然在规范模式中单个READ的输入由输入队列的大小限制（MINIX中为255个字符），但是在非规范模式下，一个READ调用必须能够传送POSIX要求的_POSIX_SSIZE_MAX个字符。在MINIX中这个值为32767。

tty.c中的下面几个函数支持字符输入。ECHO（第12531行）对一些字符进行特殊处理，但对大多数字符只是在和输入使用的同一个设备的输出方显示出来。在输入被回显的同时，一个进程可能正好要向同一个设备输出，这时如果用户正试图从键盘退格，就有可能引起混乱。为了处理这种情况，在产生正常的输出时，设备指定输出例程总是把tp->tty_reprint标志设置为TRUE，这样处理退格的函数就能够判断出是否产生了混合输出。由于echo也使用设备输出例程，所以tp->tty_reprint的当前值在回显时由局部变量rp（第12552行到12585行）保存起来。不过，如果刚开始了一个新的输入行，rp将被置为FALSE而不是保持原值以保证tp->tty_reprint在echo终止时被重置。

你可能已经注意到echo将返回一个值，例如，在in_process中的第12512行的调用中：

```
ch = echo(tp,ch)
```

echo返回的值包括回显字符所用的屏幕上的空格数，如果是TAB字符，这个值可以一直到8。这个值被存放在图3-44中的cccc域。普通字符在屏幕上占据一格，但一个控制字符（除TAB，NL，CR或DEL（0x7F）外）回显时，该字符被显示为“^”加一个可打印的ASCII字符的形式，在屏幕上占用两个位置。而另一方面NL或CR不占用空间。当然，实际的回显必须由设备指定例程完成，并且不论何时一个字符必须被送往设备时，就如同在第12580行中对普通字符那样，用tp->tty_echo执行一个间接调用，。

下一个函数rawecho用来旁路由echo进行的特殊处理。它检测ECHO标志是否被置位，如果是，就不经任何特殊处理把字符送往由tp->tty_echo指定的设备指定例程。这里使用了一个局部变量rp以防止rawecho自己对输出例程的调用改变tp->tty_reprint的值。

当in_process发现一个退格符，下一个函数backover（第12607行）将被调用。它对输入队列进行操作，如果在队列中可以回溯，那么就删除队列中的前一个字符。如果队列为空或上一个字符为行中止符，那么就不能回溯。这里也检测在讨论echo和rawecho时提到的tp->tty_reprint标志。如果为TRUE，那么就调用reprint（第12618行）在屏幕上放置一个输出行的纯拷贝。然后查询上一个显示字符的len域（图3-44中的cccc域）来确定需要在显示器上删除多少个字符，并且对每个字符，都由rawecho发出一个退格-空格-退格字符序列从屏幕上删除不需要的字符。

下一个函数是reprint。除了被backover调用外，用户也可以通过按下REPRINT键（CTRL-R）来引用该函数。第12651行到12656行的循环在输入队列中逆向查找最后一个行中止符。如果找到的是位于队列的最末尾的位置，那么reprint什么也不做就返回。否则就回显CTRL-R，即在屏幕上显示两字符序列“^R”，然后移到下一行，重显队列中从上一个行中止符到末尾的部分。

现在我们已经讨论到out_process函数（第12677行）。和in_process一样，它也由设备指定输出例程调用，但要简单一些。它由RS-232和伪终端设备指定输出例程调用，但不能被控制台例程调用。out_process工作在环形缓冲区上，但不从缓冲区中删除字符。它对缓冲区数组所做的唯一变动是如果tp->tty_termios.oflag中的OPOST（使能输出处理）和ONLCR（映射NL为CR-NL）位被置位，就在缓冲区中的NL字符前插入一个CR字符。MINIX中这两位都默认为置位。out_process的任务是保持设备tty结构中的变量tp->tty_position更新。制表符和退格符使得情况要复杂一些。

下一个例程是dev_ioctl（第12763行）。当用TCSADRAIN或TCSAFLUSH选项调用dev_ioctl时，它都支持执行tcdrain函数和tcsetattr函数。在这些情况下，dev_ioctl在输出没有结束时不能立即完成动作，所以延迟IOCTL操作的信息被保存在tty结构的一部分中。无论handle_events何时运行，它都要在调用设备指定输出例程后检查tp->tty_ioreq域，并且如果有操作不能确定，就调用dev_ioctl。dev_ioctl检查tp->tty_outleft判断输出是否结束，如果是，就执行和dev_ioctl在不延迟立即执行时相同的动作。为tcdrain服务的唯一的动作就是重置tp->tty_ioreq域并向FS发送回答消息，通知它唤醒原始调用进程。tcsetattr在TCSAFLUSH选项下调用tty_icancel取消输入。对tcsetattr的两个选项，termios结构被拷贝到设备的tp->tty_termios结构中，termios结构的地址已在对IOCTL的原始调用中被传递。然后调用setattr，接下来和tcdrain一样，通过发送一条回答消息来唤醒被阻塞的原始调用。

下一个过程是setattr（第12789行）。如同我们已经看到的，它由do_ioctl或dev_ioctl调用来改变一个终端设备的属性，而do_close则调用它来把属性重置为默认设置。setattr总是在把一个新的termios结构拷贝到一个设备的tty结构中后被调用，这是因为只拷贝参数是不够的。如果被控制的设备正处于非规范模式，那么第一个动作就是当前输入队列中的所有字符都标上IN_EOF位，如同在处于非规范模式下这些字符刚被输入到队列中时所要做的一样。直接这样做（第12803行到12809行）要比检测字符是否有该位容易一些。因为没有办法可以知道哪个属性刚被改变，哪个还保持原值。

下一个动作是检测MIN和TIME值。在规范模式中tp->tty_min总是为1；这在第12818行置位。在非规范模式中，两个值的组合允许四种不同的操作，如图3-35所示。在第12823行到12825行tp->tty_min首先被设置为tp->tty_termios.cc[VMIN]中传过来的值，然后，如果它为0并且tp->tty_termios.cc[VTIME]不为0，那么它将被修改。

最后，setattr确保如果XON/XOFF控制被禁止，输出就不停止，如果输出速度被设置为0，就发送一个SIGHUP信号，并且执行一个由tp->tty_ioctl指向的设备指定例程的间接调用，来完成只能在设备层完成的工作。

下一个函数tty_reply（第12845行）在前面的讨论中已经被多次提到。它的动作十分简单：构造并发送一条消息。如果因为某种原因回答失败，就会发生混乱。接下来的函数也同样简单。sigchar（第12866行）请求MM发送一个信号。如果NOFLSH标志没有置位，已在排队的输入将被删除——接收到的字符或行的计数被置为0，队列的头指针和尾指针重合。这是默认动作。当即将捕获一个SIGHUP信号时，NOFLSH可以被置位，以在捕获信号后恢复输入和输出。tty_icancel（第12891行）根据sigchar描述的方法无条件地丢弃等待的字符，并且另外再调用由tp->tty_icancel指向的设备指定函数取消可能留在设备本身里或被低层代码缓冲的输入。

对每个设备tty_init（第12905行）在tty_tasks第一次启动时被调用一次。它的设置为默认值。开始时，一个指向tty_devnop的指针和一个什么也不做的空函数被设置到变量tp->tty_icancel、tp->tty_ocancel、tp->tty_ioctl和tp->tty_close中。然后tty_init根据不同类型的终端（控制台，串行线，或伪终端）调用设备指定初始化函数。它们设置间接调用设备指定函数的实际的指针。前面讲过，如果根本就没有配置某一特定类型的设备，那么将生成一个立即返回的宏，这样就不需要为没有配置的设备编译代码了。对scr_init的调用初始化控制台驱动程序，并且也调用键盘的初始化例程。

tty_wakeup（第12929行）虽然短，但在终端任务的功能中却十分重要。只要时钟中断处理程序运行，也就是说，对每个时钟滴答，全局变量tty_timeout（在glo.h第5032行定义）都被检查是否包含小于当前时间的值。如果是，那么就调用tty_wakeup。tty_timeout被终端驱动程序的中断服务例程置为0，所以在每个终端设备中断的下一个时钟滴答唤醒操作都被强迫执行。当一个终端设备为一个非规范模式下的READ调用服务并且如同我们马上就要看到的那样需要设置一个超时，那么tty_timeout也会被settimer改变。当tty_wakeup运行时，它首先把tty_timeout置为TIME_NEVER，一个未来的很远的时间，以禁止下一次唤醒。然后它扫描按最早存入的最先被唤醒排序的定时器值的链表，直到达到比当前时间迟的那个定时器。这就是下一个将要执行的唤醒，它被放入tty_timeout中。tty_wakeup还把该设备的tp->tty_min设置为0，以保证即使没有收到字符下一个读操作也能成功，把设备的tp->tty_events标志置位以保证当终端任务下一次运行时得到处理，并且把该设备从定时器链表中删除。最后，它调用interrupt向任务发送唤醒消息。如同在讨论时钟任务时提到的，tty_wakeup逻辑上是时钟中断服务代码的一部分，因为它只从那儿被调用。

下一个函数settimer（第12958行）设置定时器以确定在非规范模式中何时从一个READ调用中返回。它的调用参数为一个指向tty结构的指针tp，和一个值为TRUE或FALSE的整数on。首先扫描由timelist指向的tty结构链表，寻找与tp参数匹配的已存在的表目。如果找到，就把它从表中删除（第12968行到12973行）。如果settimer被调用来取消设置一个定时器，那么这些就是它所要做的全部工作。如果它被调用来设置一个定时器，那么设备tty结构中的tp->tty_time元素被设置为当前时间加上在设备的termios结构的TIME值中指定的以十分之一秒为单位的增量。然后把匹配表目按排好的次序放入表中。最后，比较刚放入表中的超时和全局变量tty_timeout的值，如果新的超时较早，就用其取代后者。

最后，tty.c中还定义了一个tty_devnop（第12992行），它是一个无操作函数，被间接地定位在一个不需要服务的设备中。我们已经看到在tty_init中tty_devnop被用作在调用设备的初始化例程之前置入各个函数指针的默认值。

3.9.5 键盘驱动程序的实现

现在我们回到支持由一个IBM PC键盘和一个内存映象显示器组成的MINIX控制台的设备无关代码。支持它们的物理设备是完全独立的：在一个标准的桌面系统中显示器使用一块插在底板上的适配卡（至少由半打不同的基本型号），而键盘由设计在母板上的电路支持，通过该电路与键盘内部的一个8位单片机接口。两个子设备需要完全独立的软件支持，即文件keyboard.c和console.c。

操作系统把键盘和控制台看作同一个设备/dev/console的两个部分。如果显示卡上有足够的存储器，就可以把虚拟控制台（virtual console）支持编译进去，这样除/dev/console外，还可以有其他的逻辑设备，如/dev/ttyc1，/dev/ttyc2等。输出只在任何给定的时刻送往显示器，并且对任何被激活的控制台都只有一个键盘用来输入。在逻辑上键盘对于控制台是有用的，但只有两个相对次要的方面说明了这一点。首先，tty_table包含了一个控制台的tty结构，为输入和输出提供了单独的域，例如，keyboard.c和console.c中的函数指针

tty_devread和tty_devwrite在启动时被赋值。不过，只有一个tty_priv域，并且它只指向控制台的数据结构。其次，在进入主循环前，tty_task调用每个逻辑设备一次以进行初始化。为/dev/console调用的例程位于console.c中。从那里调用键盘的初始化代码。不过这种隐含的层次关系也可能被倒过来。在处理I/O设备时我们总是先看输入再看输出，现在我们仍然保持这种模式，在这一节中讨论keyboard.c，而把console.c的留到后面讨论。

和我们看到的大部分源文件一样，keyboard.c的开头也是几条#include语句。不过其中有一条比较特殊。keymaps/us-std.src（在第13014行中包含）不是一个普通的头文件；这是一个C源文件，它使得默认键位映射表被编译为keyboard.o中的一个已初始化的数组。由于篇幅所限，键位映射表源文件没有在本书的末尾列出，但图3-41中示出了一些代表性的表目。在#include语句之后的是定义各种常数的宏。第一组用于和键盘控制器的低层交互。其中许多是在这些交互中有意义的I/O端口地址或位组合。下一组包含了一些特殊键的符号名。由于IBM硬件只支持一个键盘，因此宏kb_addr（第13041行）总是返回一个指向kb_lines数组中第一个元素的指针。下一行中用符号定义了键盘输入缓冲区的大小为KB_IN_BYTES，其值为32。接下来的11个变量用来保存在解释一个按键时所需要记忆的各种状态。它们的使用各不相同。例如，每次当Caps Lock键按下时，capslock标志的值（第13046行）在TRUE和FALSE之间切换。当Shift键被按下时，shift标志被置为TRUE，当Shift键被松开时，该标志被置为FALSE。当接收到一个转义扫描码时，esc变量被置位。在收到后继字符时，它总是被重置。

第13060行到13065行中的kb_s结构用来跟踪输入的扫描码。在这个结构中，扫描码被放在数组ibuf中，该数组为一个环形缓冲区，大小为KB_IN_BYTES。每个控制台说明了这些类型的一个数组kb_lines[NR_CONS]，但实际上只使用了第一个，这是因为kbaddr宏总是被用来确定当前kb_s结构的地址。不过，我们一般使用一个指向该结构的指针来引用kb_lines[0]中的变量，例如kb->ihead，这样我们就可以和处理其他设备的方法保持一致，并且使在文本中的引用与列出的源代码一致。当然，由于有许多数组元素未使用，要浪费一小部分空间。不过，如果有人制造出支持多个键盘的PC，MINIX也能支持，只需要对kbaddr宏进行一些修改。

map_key0（第13084行）被定义为一个宏。它返回一个扫描码对应的ASCII码，忽略修饰符。这等价于键位映射表数组中的第一列。它的兄弟函数是map_key（第13091行），该函数进行扫描码到ASCII码的完全映射，包括处理和普通字符同时按下的（多重）修饰键。

键盘中断服务例程是kbd_hw_int（第13123行），当一个键被压下或松开时被调用。它调用scan_keyboard从键盘控制器芯片获取扫描码。当一个键被松开引起中断时，其扫描码的最高有效位被置位，在这种情况下该键被忽略除非它是修饰键之一。如果中断由按下一个键引起，或松开一个修饰键，则原始的扫描码被放在环形缓冲区中，如果还有空间的话；当前控制台的tp->tty_events标志被置位（第13154行），然后调用force_timeout被调用以保证时钟任务在下一个时钟滴答启动终端任务。图3-45给出了在缓冲区中的一短行字符的扫描码，其中包含两个大写字符，每个的前导是压下一个shift键的扫描码，而后继是松开shift键的扫描码。

图 3－45　在输入缓冲区中的由键盘输入的一行文本的扫描码，下面一行是对应的按下的键。其中，L+，L-，R+和R-分别代表按下和松开左、右Shift键。松开一个键的扫描码比按下同一个键时大128。

当时钟中断发生时，就运行终端任务自身，当发现控制台的tp->tty_events标志被置位时，就使用控制台tty结构中tp->tty_devread域中的指针调用设备指定例程kb_read（第13165行）。kb_read从键盘的环形缓冲区中取出扫描码，并把ASCII码放在它的局部缓冲区中，缓冲区应足够大以保存来自数字键盘的某些扫描码产生的转义序列。然后它调用硬件无关代码中的in_process把字符放入输入队列。第13181行到13183行使用了lock和unlock来防止kb->icount被一个可能在同一时刻到达的键盘中断减少。对make_break的调用以整数形式返回ASCII码。特殊键，例如小键盘和功能键，他们的值大于0xFF。在HOME和INSRT之间的扫描码（0x101到0x10C，在include/minix/keymap.h中定义）由按下数字键盘产生，并且通过numpad_map数组被转换为图3-46所示的三字符转义序列。这些序列接着被传送到in_process中（第13196行到13201行）。更高的扫描码不传给in_process，但要检测ALT-LEFT-ARROW，ALT-RIGHT-ARROW和ALT-F1到ALT-F12这些扫描码，并且如果发现其中之一，就调用select_console切换虚拟控制台。

图3－46　由数字键盘产生的转义编码。普通键的扫描码被转换为ASCII码，而特殊键被转换为大于0xFF的“伪ASCII”码。

make_break（第13222行）把扫描码转换为ASCII码然后更新跟踪修饰键状态的变量。首先，它检测PC用户都知道的用来在MS-DOS下强制重启动的神奇的CTRL-ALT-DEL组合。不过由于希望的是顺序地关机，所以要向init，所有进程的父进程，发送一个SIGABRT信号，而不是试图启动PC BIOS例程。init应该在返回到可以完全重启系统或重启MINIX的启动监控程序之前接收这个信号，把它解释为开始一个按次序关机的进程的命令。当然，希望这个过程每次都能完成是不现实的。大部分的用户清楚在某些地方真正出了问题并且系统已不能正常控制之前不按CTRL-ALT-DEL而突然关机的危险。这时系统可能变得如此的混乱以致按次序向另一个进程发送信号是不可能的。这就是在make_break中为什么有一个静态变量CAP_count的原因。大多数系统崩溃时，中断系统仍能工作，所以键盘输入仍然可以被接收，时钟任务仍可以保持终端任务运行。这里MINIX利用了计算机用户的可预期的行为，即在某些东西看起来不能正常工作时重复地重击键盘。如果试图向init发送SIGABRT失败，并且用户按了CTRL-ALT-DEL两次以上，就直接调用wreboot返回到监控程序而不经过调用init。

make_break的主要部分是不难看懂的。变量make记录扫描码由按下还是放开一个键产生，接着被调用的

map_key把ASCII码返回给ch。接下来是根据ch的一个switch语句（第13248行到13294行）。我们要考虑两种情况，一种是普通键，一种是特殊键。对于普通键没有可以匹配的情况，在缺省情况下也不做任何处理（第13292行），这是因为普通键只在按下和放开的产生（按下）阶段被接收。如果由于某种原因一个普通键在放开时被接收，那么就用-1代替，而调用者kb_read将忽略这个键。而一个特殊键，例如CTRL，会在switch语句中合适的地方被检测出来，对于CTRL是在第13249行。相应的变量，在这种情况下是control，记录make的状态，用-1代替要返回的字符扫描码（被忽略）。对ALT，CALOCK，NLOCK和SLOCK键的处理要复杂一些，但是对所有这些特殊键的效果是相似的：一个变量记录了当前的状态（对只在按下时有效的键）或保持前一个状态（对锁定键）。

另外还要考虑一种情况，就是EXTKEY扫描码和esc变量。它不会和键盘上的ESC键混淆，返回的ASCII码是0x1B。从键盘上按任何一个键或组合键都是无法产生单独的EXTKEY扫描码的；它是PC 键盘的扩展键前缀（extended key prefix），是一个两字节扫描码的第一个字节，它说明了一个不是初始PC扩充键的一部分但具有相同的扫描码的键被按下了。在很多情况下软件对这两个键的处理是等同的。例如，对普通的“/”字符和数字键盘上灰色的“/”键，几乎总是这样的情况，在其他情况下，则可能需要区别这些字符。例如，许多非英语键盘布局对左右ALT键作不同的处理，来支持必须产生不同字符扫描码的键。两个ALT键都产生相同的扫描码（56），但当右ALT按下时前面要加上EXTKEY码。当返回EXTKEY码时，esc标志被置位。在这种情况下，make_break从switch内部返回，在正常返回前旁路了在每种情况下把esc置为0（第13295行）这最后一步。这使得esc只对紧接着收到的扫描码有效。如果你熟悉PC键盘在普通应用下的各种复杂情况，那么你也会对这种情况感到熟悉，但也会有一点奇怪，因为PC BIOS不允许读一个ALT键的扫描码并且为扩展扫描码返回一个不同的值，而MINIX却允许这样做。

set_leds（第13303行）打开或关闭指示PC键盘上的Num Lock、Caps Lock或Scroll Lock键是否被按下的灯。一个控制字节LED_CODE被写到输出端口通知键盘写到端口的下一个字节是指示灯控制字节，三个指示灯的状态用该字节中的三位进行编码。下面两个函数支持这个操作。kb_wait（第13327行）被调用来决定键盘是否已准备好接受一个命令序列；kb_ack（第13343行）被调用来检验该命令是否被响应。这些命令都使用忙等待，即持续地读直到出现一个期望的扫描码。对于处理大多数的I/O操作，这并不是一种值得推荐的技术。但是打开或关闭键盘上的指示灯并不经常发生，所以如果效率稍低一些并不会浪费很多时间。注意这两个函数也都可能会失败，如果发生了这种情况，可以从函数的返回码判断出来。但是设置键盘上的灯并不是很重要的工作，并不值得这两个调用去检查它的返回值，所以set_leds只是直接继续执行。

因为键盘是控制台的一部分，所以它的初始化例程kb_init（第13359行）从console.c的scr_init中被调用，而不是直接从tty.c的tty_init中调用。如果使能了虚拟控制台（即include/minix/config.h中的NR_CONS大于1），则对每个逻辑控制台调用一次kb_init。在第一次之后，kb_init中唯一对于附加控制台必不可少的部分就把kb_read的地址设置到tp->tty_devread中（第13367行），但重复函数的其他部分并不会有害。kb_init的余下部分初始化一些变量，打开键盘上的指示灯，并扫描键盘以确保没有残余的已读入的击键。一切都初始化完毕后，它调用put_irq_handler，然后接着调用enable_irq，所以当一个键按下或松开时将执行kbd_hw_int。

下面三个函数都非常简单。kbd_loadmap（第13392行）几乎微不足道。它由tty.c的do_ioctl调用把一个键位映射表从用户进程空间拷贝到内核空间，覆盖由在keyboard.c开头包含的键位映射表源文件编译成的默认映射表。

func_key（第13405行）从kb_read中调用以判断是否按下了意味着局部处理的特殊键。图3-47总结了这些键和它们的作用。调用的代码可以在几个文件中找到。F1和F2激活dmp.c中的代码，我们将在下一节中讨论。F3激活console.c中的toggle_scroll，也在下一节讨论。而CF7、CF8和CF9扫描码将引起对tty.c中sigchar的调用。当在MINIX中加入网络工作时，一个附加的case，用来检测F5扫描码，被添加进来以显示以太网的状态。除此以外，还有大量的其他扫描码可以用来从控制台启动其他的调试信息或特殊事件。

scan_keyboard（第13432行）在硬件接口层工作，从I/O端口读写字节。从第13440行到13442行的序列通知键盘控制器读入了一个字符，该序列读入一个字节，把它的最高有效位置为1再写一次，再把该位重置为0再重写一次。这样就防止了后继的读操作读入相同的数据。在读键盘时没有状态检测，但在任何情况下都应该没有问题，因为scan_keyboard只在中断时被调用，来自kb_init的异常调用会清除任何不需要的东西。

图 3－47　func_key()检测的功能键。

keyboard.c中的最后一个函数是wreboot（第13450行）。如果是由于系统混乱被引用，它为用户提供了用功能键显示调试信息的机会。第13478行到13487行的循环是使用忙等待的另一个例子。它重复地读键盘直到键入了一个ESC。当然没有人可以断言在崩溃后等待命令重启时需要一种更有效的技术。在循环内部，func_key被调用以提供获得有助于分析崩溃原因的信息的可能。我们将不再讨论返回到引导监控程序的细节。这些细节非常依赖于硬件并且与操作系统没有多少关系。

3.9.6　显示驱动程序的实现

如果有足够的内存，IBM PC显示器可以被配置为几个虚拟终端。这一节中我们将探讨控制台的设备相关代码。我们还要研究使用键盘和显示器低层服务的调试转储例程（debug dump routine）。这些例程支持与控制台使用者有限的交互，即使在MINIX系统的其他部分不工作时。它们还能在系统几乎完全崩溃之后提供有用信息。

对控制台输出到PC内存映象屏幕的依赖于硬件的支持在console.c中。console结构在第13677行到13693行

中定义。在某种意义上，这个结构是tty.c中定义的tty结构的扩展。在初始化时，控制台的tty结构的tp->tty_priv域被赋予一个指向自己的console结构的指针。console结构中的第一项是一个指回对应的tty结构的指针。console结构中的元素对应于一个视频显示器：记录光标的行列位置的变量，显示内存的起始地址和界限，由控制器芯片的基指针指向的内存地址，和光标的当前地址。其他的变量用来管理转义序列。因为字符最初被接收时是8位字节的形式，必须和属性字节组合以16位字的形式传送到视频内存，所以在c_ramqueue中建立了一个待传送的块，这个块是一个足够大的数组，用来存放一整行80列16位的字符-属性对。每个虚拟控制台需要一个console结构，存放在数组cons-table中（第13696行）。和处理tty与kb_s结构一样，我们一般通过一个指针，例如cons->c_tty，来引用console结构中的元素。

cons_write函数的地址存放在每个控制台的tp->tty_devwrite入口处（第13729行）。它只从一个地方被调用，即tty.c的handle_events中。console.c中的其他大部分函数的存在都是为了支持这个函数。当它在一个客户进程进行了一个WRITE调用后被第一次调用时，待输出的数据位于客户进程的缓冲区中，可以用tty结构中的tp->tty_outproc和tp->out_vir域找到。tp->tty_outleft域记录了还有多少字符需要传送，tp->tty_outcum被初始化为0，表示什么还没有传送。这是在进入cons_write时的一般情况，因为在正常情况下，一旦该函数被调用，它就传送原始调用请求的所有数据。不过，如果用户希望进程能慢下来，以便在屏幕上察看数据，那么他可以键入一个STOP（CTRL-S）字符，这将引起tp->tty_inhibited标志置位。cons_write在该标志置位时立即返回，即使WRITE还没有结束。在这种情况下，handle_events将继续调用cons_write，当tp->tty_inhibited最终被重置时，用户通过键入START（CTRL-Q）可以让cons_write继续中断的传送。

cons_write的唯一的参数是一个指向特定控制台tty结构的指针，所以第一件必须做的事就是初始化cons，这是一个指向这个控制台的console结构的指针（第13741行）。然后，因为handle_events总是调用cons_write，所以第一个动作就是检测是否真的有事要做。如果没有就快速返回（第13741行）。接下来就进入了第13751行到13778行的主循环。这个循环在结构上与tty.c中in_transfer的主循环很相似。通过调用phys_copy从客户进程的缓冲区中获得数据，并填入一个可以保存64个字符的局部缓冲区，指向数据源的指针和计数器被更新，然后局部缓冲区中的每个字符连同稍后将由flush传送到屏幕的属性字节一起被传送到cons->c_ramqueue数组中。可以用不止一种方法来执行传送，如图3-39所示。可以调用out_char来传送每个字符，但可以预见如果字符是一个可见字符，没有一个转义序列，没有超过屏幕宽度，并且cons->c_ramqueue数组没有满，那么out_char的特殊服务就变得多余。该字符连同属性字节（由cons->c_attr获得）被直接放入cons->c_ramqueue中，并且增加cons->c_rwords（队列的索引）、cons->c_column（跟踪屏幕上的当前列）和指向缓冲区的指针tbuf。这个把字符直接放入cons->c_ramqueue中的动作对应于图3-39中左边的虚线。如果需要，就调用out_char（第13766行到13777行）。它进行所有的登记，另外还在必要时调用flush最终把字符传送到屏幕存储器。只要tp->tty_outleft指示仍有字符等待传送并且tp->tty_inhibited仍然没有置位，从用户进程缓冲区到局部缓冲区再到队列传送就重复进行。当传送停止时，不论是因为WRITE操作结束还是tp->tty_inhibited被置位，就再次调用flush把队列中余下的字符传送到屏幕存储器。如果操作完成（检测tp->tty_outleft是否为0），就调用tty_reply（第13784行到13785行）发送一条回答消息。

除了从handle_events调用cons_write外，待显示的字符还由终端任务的硬件无关部分的echo和rawecho送往控制台。如果控制台是当前的输出设备，就由tp->tty_echo指针调用下一个函数cons_echo（第13794行）。cons_echo通过调用out_char然后再调用flush完成所有的工作。来自键盘的输入逐个字符地到达，而输入的人希望看到回显没有可察觉的延迟，所以把字符送入输出队列并不能满足要求。

接下来我们讨论out_char（第13809行）。它检测是否有一个转义序列，如果有，就调用parse_escape并立即返回。否则，就进入一个switch语句，检查与特殊情况的匹配：空字符，退格，响铃符，等等。对这些情况的处理大都容易理解。换行和制表符是最复杂的，因为它们涉及到光标在屏幕上的位置变化，并且可能需要卷屏。最后要检测ESC码。如果有，就置位cons->c_esc_state标志（第13871行），对out_char的调用转换为对parse_escape的调用，直到序列结束。最后，对可打印字符进行缺省处理。如果超出了屏幕宽度，则屏幕可能需要卷屏，flush将被调用。在一个字符被放入输出队列前，要检测队列是否已满，如果是，就调用flush。把一个字符放入队列需要进行我们前面在cons_write中看到的记帐。

下一个函数是scroll_screen（第13896行）。scroll_screen处理一般的在屏幕最后一行满时的向上卷屏，以及向下卷屏，在光标定位命令试图把光标移到屏幕最顶行之外时发生。对每个方向的卷动，有三种可能的处理方法。它们要用来支持不同的视频卡。

我们将研究向上卷动的情况。开始时，chars被赋予屏幕尺寸减去一行的值。软件卷屏由一个单独的对vid_vid_copy的调用完成，把chars个字符向内存低端移动，移动的距离是一行中的字符数。vid_vid_copy是可回卷的，即如果请求移动的内存块溢出了赋予视频显示内存的上边界，就从内存块的低端取出溢出部分，把它移动到高于被移到低端部分的地址处，即把整个块看作一个环形数组。这个调用看起来简单，但执行却相当慢。即使vid_vid_copy是定义在klib386.s中的汇编语言例程，这个调用也需要CPU移动3840个字节，就算对于汇编语言也是一项繁重的工作。

软件卷屏永远不会作为默认方法；只有在硬件卷屏不能工作或由于某种原因不能采用时才选择软件方式。一个原因可能是因为希望使用screendump命令把屏幕内存存入一个文件。当使用硬件卷屏时，screendump可能

会产生不可预知的结果，因为屏幕内存的起始地址和显示器可见部分的开始位置可能不一致。

在第13917行中wrap变量被检测，它是一系列复合检测的第一部分。对于较老的支持硬件卷屏的显示器wrap为真，如果检测失败，那么第13921行将进行简单的硬件卷屏，视频控制芯片使用的原始指针被更新为指向显示器左上角显示的第一个字符。如果wrap为FALSE，复合检测将继续测试卷屏操作将要移动的内存块是否溢出了分配给该控制台的内存界限。如果是，就再次调用vid_vid_copy执行一个回卷移动把内存块移动到控制台分配内存的起始处，并且原始指针被修改。如果没有重叠，控制就被传递给旧的视频控制器一直使用的简单硬件卷屏方法。这一步包括调节cons->c_org，然后把新的原点放入控制器芯片对应的寄存器中。这个调用后面还要调用一次，因为有一个清空屏幕最后一行的调用。

向下卷屏的代码与向上卷屏很相似。最后，调用mem_vid_copy清空由new_line指向的屏幕最后（或最顶部）一行。然后调用set_6845把新的原点从cons->c_org中写到对应的寄存器中，flush将确保所有的改动在屏幕上可见。

我们已经好几次提到flush（第13951行）。它使用mem_vid_copy把队列中的字符传送到视频内存中，更新某些变量，并保证行和列的数值是合理的，例如，如果一个转义序列试图把光标移到一个负的列数的位置上，flush将调整它们。最后将计算光标应处的位置，并与cons->c_cur比较。如果它们不一致，并且如果现在正在处理的视频内存属于当前的虚拟控制台，就调用set_6845把控制器的光标寄存器设置为正确的值。

图3-48绘出了对转义序列的处理如何被表示为一个有限状态自动机。这是由parse_escape（第13986行）实现的，如果cons->c_esc_state不为0，它就在

图 3－48　处理转义序列的有限状态机。

out_char的开头被调用。out_char本身检测到一个ESC并且设置cons->c_esc_state为1。当接收到下一个字符时，parse_escape就在指向参数数组起始位置的指针cons->c_esc_intro中放入一个‘\0’，把cons->c_esc_parmv[0]放入cons->c_esc_parmp中，并且参数数组本身都置为0，准备进一步的处理。接着检查ESC后面的第一个字符——有效的值为“[”或“M”。在第一种情况下，“[”被拷贝到cons->c_esc_intro中，状态转换到2。在第二种情况下，do_escape被调用来执行这个动作，并且转义状态被重置为0。如果ESC之后的第一个字符不是有效值中的一个，它将被忽略，后继的字符仍然正常显示。

当检测到一个ESC [序列时，输入的下一个字符由状态2的代码处理。这里有三种可能性。如果该字符是一个数值字符，那么它的值被取出并与由cons->c_esc_parmp所指向的当前位置上的值乘以10相加，初始值为cons->c_esc_parmv[0]（被初始化为0）。转义状态没有变化。这使得可以输入一系列十进制数并累积为一个很大的数值参数，虽然MINIX现在最大能识别的值为80，由把光标移到任意位置的序列使用（第14027行到14029行）。如果该字符为分号，那么指向参数字符串的指针就向前移动，这样后继的数字值可以在第二个参数中累积（第14031行到14033行）。如果想要改变MAX_ESC_PARMS来为参数分配一个大一些的数组，也不需要改变这段代码以在输入额外的参数后累积额外的数字值。最后，如果字符既不是一个数字也不是分号，就调用do_escape。

尽管MINIX能识别的转义序列相对来说并不多，do_escape（第14045行）却是MINIX系统源代码中较长的一个函数，。不过，这段代码并不难理解。在执行了一个对flush的初始调用以保证视频显示被完全更新之后，有一个简单的if选择，这个选择取决于紧接着ESC之后的字符是否是一个特殊的 控制序列引入符。如果不是，那么只有一个有效的动作，即如果该序列是ESC M就把光标上移一行。注意对“M”的检测是在一个switch中的缺省动作中完成，作为有效性检查并等待其他ESC [格式的序列的附加部分。在许多转义序列中检查cons->c_row变量来决定是否需要卷屏这个动作很典型。如果光标已经位于第0行，就用SCROLL_DOWN来调用scroll_screen；否则光标被上移一行。后者由对cons->c_row减一并调用flush完成。如果发现了一个控制序列引入符，那么就执行第14069行在else之后的代码。这里要对MINIX现在唯一能识别的控制序列引入符“[”进行检测。如果序列有效，那么在转义序列中的第一个参数，或在没有输入参数时为0，被赋给value（第14072行）。如果该序列无效，那么除了跳过接下来的switch（第14073行到14272行）并且在从do_escape返回之前把转义状态重置为0外，什么也不做。在我们更感兴趣的序列有效的情况中，将进入switch语句。我们不讨论所有的情况；我们只研究转移序列决定的动作中具有代表性的几种。

开始的五个没有数值参数的序列由IBM PC键盘上的四个箭头键和Home键产生。头两个ESC [A和ESC [B与ESC M很相似，除了它们可以接受一个数值参数并且可以向上或向下移动多于一行的距离，如果参数指定的移动超过了屏幕的界限，它们并不卷动屏幕。在这些情况下，flush捕获移出界限的请求并把移动限制在对应的最后一行或第一行。下面两个序列，ESC [C和ESC [D把光标向左右移动，它们也类似地受到flush的限制。当由箭头键产生时，没有数值变量，这样就进行默认的移动一行或一列。

下一个序列ESC [H可以有两个数值变量，例如，ESC [20;60H。这两个参数指定的是绝对位置而不是相对位置，并且被从1计数的数值转换为从0计数的数值。Home键产生默认的序列（无参数）把光标移动到（1，1）处。

下面的两个序列ESC [sJ和ESC [sK，清除全部屏幕或当前行的一部分，这将取决于输入的参数。每种情况下都计算字符计数。例如，对于ESC [1J，count中保存着从屏幕的开始到光标位置的字符数，这个计数和一个可以是屏幕的起始位置cons->c_org，或当前的光标位置cons->c_cur的位置参数dst，被用作调用mem_vid_copy的参数。用一个参数调用这个过程将使得它用当前背景色填充指定的区域。

下面的四个序列插入或删除在光标处的行或空格，它们的动作不需要详细的解释。最后一种情况ESC [nm

（注意n代表一个数值参数，但“m”是一个文字）对cons->c_attr起作用，即当写到视频存储器时交错存放在字符码之间的属性字节。

下一个函数set_6845（第14280行）被用来更新视频控制芯片。6845具有16位的内部寄存器，一次可以对8位编程，写一个寄存器需要四次I/O端口写操作。Lock和unlock调用用来禁止中断，如果允许序列被中断的话可能会产生问题。图3-49中给出了6845视频控制芯片的部分寄存器。

图 3－49 部分6845寄存器。

beep函数（第14300行）在必须输出CTRL-G字符时被调用。它利用了PC内建的对声音的支持，向扬声器发送方波发出声音。声音的初始化代码大多数是只有汇编程序员才喜爱的I/O端口操作，要注意过程中有一段临界区不能被中断。代码中更令人感兴趣的部分是使用了时钟任务设置闹钟的功能，这可以用来启动一个函数。下一个例程stop_beep（第14329行）的地址被放在送往时钟任务的消息中。它在给定的时间后停止发声并且重置用来防止其他对发声例程的调用的beeping标志。

scr_init（第14343行）由tty_init调用NR_CONS次。每次调用的参数是一个指向tty结构的指针，它是tty_table的一个元素。在第14354行和14355行，line被用作cons_table数组的索引， 它首先被计算出来，然后进行有效性检测，如果有效，就用来初始化指向当前控制台表表目的指针cons。这里可以用指向设备的主tty结构的指针来初始化cons->c_tty，并且tp->tty_priv可以指向该设备的console_t结构。接着，kb_init被调用来初始化键盘，然后建立指向设备指定例程的指针，tp->tty_devwrite指向cons_write，tp->tty_echo指向cons_echo。第14368行到14378行取出CRT控制器基址寄存器的I/O地址，决定视频内存的地址和大小，并根据使用的视频控制器的类型设置wrap标志（用来确定如何卷屏）。第14382行到14384行在全局描述符表中初始化视频内存的段描述符。

接下来是虚拟控制台的初始化。每次调用scr_init时，参数是不同的tp值，这样第14393行到14396行就用不同的line和cons为各虚拟控制台提供可用视频内存的各自的部分。接着每个屏幕被清空，准备好开始，最后控制台0被选为第一个活动控制台。

console.c中余下的例程都很短并且简单，我们将很快地说明一下。putk（第14408行）前面已经提过了。它为任何链接到内核映象里需要服务的代码打印一个字符而不用通过FS。toogle_scroll（第14429行）的动作与它的名称一样，它切换确定使用硬件还是软件卷屏的标志。它还在当前光标位置显示一条信息来确认已选择的模式。cons_stop（第14442行）重新初始化控制台为重启监控程序指定的状态，优先于关机或重启。cons_org0（第14456行）只有在F3键强制切换卷屏模式或准备关机时使用。select_console（第14482行）选择一个虚拟控制台。它用新的索引调用，并调用set_6845两次使视频控制器显示视频内存的正确部分。

最后两个例程非常依赖于硬件。con_loadfont（第14497行）把一个字体装入图形适配器，以支持IOCTL TIOCSFON I/O操作。它调用ga_program（第14540行）执行一系列对I/O端口的写使得一般不可被CPU寻址的视频适配器字体内存变为可见。然后调用phys_copy把字体数据拷贝到这块内存区，并且引用另一个序列把图形适配器设置回一般操作模式。

调试转储

在终端任务中我们最后要讨论的一组过程原来只是为了在调试MINIX时暂时使用的。当不再需要时，它们可以被删除，但很多用户发现保留它们很有用。在修改MINIX时，它们特别有用。

如同我们已经看到的，在kb_read的开始调用func_key来检测用来控制和调试的扫描码。在检测到F1和F2键时调用的转储例程位于dmp.c中。首先，p_dump（第14613行）为所有的进程显示基本的处理信息，包括在按下F1键时显示的内存使用信息。第二，map_dump（第14660行）在按下F2时提供更详细的内存使用信息。proc_name（第14690行）通过检查进程名支持p_dump。

由于这段代码完全包含在内核二进制代码本身之中。并且不作为一个用户进程或任务运行，所以即使在一个严重的系统崩溃之后，它也常常能继续正确地工作。当然，这些例程只能从控制台存取。转储例程提供的信息不能被重定向到一个文件或其他设备中，所以硬拷贝或经由网络连接使用并不是可选项之一。

我们建议向试图扩充MINIX的第一步最好是扩展转储例程以在你想要扩充的方面提供更多的信息。

3.10 MINIX中的系统任务

把文件系统和存储管理器服务进程放在内核以外的一个后果就是有时它们拥有内核需要的信息。不过这种结构禁止了它们把信息写入到内核表中。例如，系统调用FORK由存储管理器处理。当生成一个新进程时，为了能够进行调度，内核必须知道新进程的信息。存储管理器是如何通知内核的呢？

解决这个问题的办法是使用一个内核任务通过消息机制与文件系统和存储管理器通信，而且它对所有内核表拥有存取权。这个任务被称为系统任务，处于图2-26中的第二层，它的作用与我们在本章中研究过的任务相似。唯一的区别在于它不控制任何I/O设备。但是，和I/O任务一样，它实现了一个接口，但在这种情况下不是面向外部世界，而是面向系统中大部分的内部组件。它和I/O任务具有相同的特权，并和它们一起被编译到内核映象中，在这里研究它比在其他章节中更合适。

系统任务接受19种消息，如图3-50所示。系统任务的主程序sys_task（第14837行）与其他任务具有相似的结构。它接收一条消息，分发给合适的服务过程，然后发送一条回答消息。我们将观察每一条消息和它的服务过程。

SYS_FORK消息被存储管理器用来通知内核产生了一个新的进程。内核需要了解这条消息以进行调度。该消息包含进程表内部的对应于父亲和孩子的插槽号，存储管理器和文件系统也有进程表，在三个表中表目k指的都是同一个进程。这样，存储管理器就可以只指定父亲和孩子插槽号，而内核会知道指的是那些进程。

图 3－50　系统任务接收的消息类型。

过程do_fork（第14877行）首先检查存储管理器送给内核的是否是无用的东西。检测使用了一个宏isok-susern，它在proc.h中定义，用来检测父进程和子进程表表目是否有效。在system.c中绝大部分服务过程都要进行类似的检测。这稍有一点偏执狂，但保持内部检查的一致性并没有坏处。do_fork把父进程的进程表表目拷贝到子进程的槽中。这里需要进行一些调整。子进程不受父进程的任何未确定信号的影响，并且子进程不继承父进程的跟踪状态。当然，所有子进程的帐号信息被置为0。

在一个FORK调用之后，存储管理器为子进程分配内存。内核必须知道子进程位于内存何处以在运行子进程时能正确设置段寄存器。SYS_NEWMAP消息允许存储管理器传给内核任何进程的存储映象。该消息也可以在一个BRK系统调用改变了映象之后使用。

这条消息由do_newmap（第14921行）处理，它必须首先把新的映象从存储管理器的地址空间中拷贝出来。映象本身并不包含在消息中因为它太大了。在理论上，存储管理器可以通知内核映象位于地址m处，而m会是一个非法地址。实际上存储管理器并不这样做，但内核仍然要检查。映象被直接拷贝到取得新映象的进程的进程表表目的p_map中。对alloc_segments的调用将从映象中提取出信息并把它装入保存段寄存器的p_reg中。这并不复杂，但细节将依赖于处理器，因此被分离在一个独立的函数中。

SYS_NEWMAP消息更经常地在MINIX系统的普通操作中使用。另一条类似的消息SYS_GETMAP只在文件系统初始启动时使用。这条消息需要从内核向存储管理器反向传送进程映象信息。它由do_getmap（第14957行）执行。两个函数的代码很相似，只是在调用phys_copy时源和目的参数交换了位置。

当一个进程执行EXEC系统调用时，存储管理器为它建立一个新的包含参数和环境的堆栈。它使用SYS_EXEC把堆栈指针传给内核，该消息由do_exec（第14990行）处理。在对一个有效进程的一般检测后，要检查消息中的PROC2域。这里该域被用作指示进程是否被跟踪的标志，与标志一个进程无关。如果正处于跟踪之下，就调用cause_sig发送一个SIGTRAP信号给进程。这里该信号的效果
与一般情况下不一样，正常情况下它将终止一个进程并引起一个内核的转储显示。在存储管理器中，所有送往被跟踪进程的信号除SIGKILL外都被截获并暂停被跟踪的进程以便调试程序能控制它下一步的执行。

EXEC调用会引起一些轻微的不正常。产生该调用的进程向存储管理器发送一条消息并且阻塞自己。在其他的系统调用中，回答消息将释放进程。但EXEC调用没有回答，因为新装入的内核映象不需要回答消息。因此，do_exec在第15009行自己释放进程。下一行用防止可能竞争条件的lock_ready函数使新存储映象准备运行。最后，保存命令字符串使得在用户按下F1键显示所有进程状态时该进程可以被识别。

MINIX中可以用一个EXIT系统调用向存储管理器发送一条消息来退出一个进程，也可以通过一个信号终止一个进程。在两种情况下，存储管理器通过SYS_XIT通知内核。这项工作由do_xit（第15027行）完成，它比你想象的要复杂得多。处理记帐信息的方法很直接。如果有闹钟计时器，可以在它的顶部存入一个0来终止它。因为这个原因时钟任务在一个定时器超时时总是检查是否仍有需要处理的事件。do_xit中比较富于技巧的部分是当进程被终止时它可能处于发送或接收排队状态。第15056行到15076行的代码检测这种可能性。如果在其他进程的消息队列中发现了已存在的进程，那么它将被很小心地删除。

和前一条稍稍复杂一些的消息比起来，SYS_GETSP完全就是微不足道的了。它被存储管理器用来找出某些进程的的当前堆栈指针的值。BRK和SBRK系统调用需要该值来判断数据段和堆栈段是否冲突。这些代码在do_getsp中（第15089行）。

现在我们研究几种只用于文件系统的消息类型之一，SYS_TIMES。实现TIMES系统调用时需要这条消息，它向调用者返回计帐时间。do_times（第15106行）的所有任务就是把请求的时间放入回答消息。对lock和unlock的调用用来防止在存取时间计数器时可能发生的竞争。

存储管理器或文件系统都有发现一个错误以致于无法继续执行操作的可能。例如，如果在首次启动时文件系统发现根设备的超级块有致命的损坏，它将陷入混乱状态并向内核发送一条SYS_ABORT消息。超级用户也可能强迫返回到启动监控程序和/或使用reboot命令调用REBOOT系统调用重新启动。在任何一种情况下，系统任务执行do_abort（第15131行），把指令拷贝到监控程序中，然后调用wreboot完成处理。

信号处理的大部分工作由存储管理器完成，如果信号的发送者有权这样做的话，它检测接收信号的进程是否被允许捕获或忽略信号，以及诸如此类的工作。不过，存储管理器并不能真正引发需要向进程的堆栈中压入一些信息的信号。

POSIX之前的信号处理有一些问题，因为捕获一个信号时恢复了对信号的默认响应。如果需要连续地处理后继信号，那么程序员将不能保证可靠性。信号是异步的，并且第二个信号可能正好在处理被重新使能之前到达。POSIX风格的信号处理解决了这个问题，但付出的代价是较复杂的机制。旧的方法可以通过操作系统向进程的堆栈中推入一些信息来实现，和一个中断推入信息很相似。这样程序员就可以写一个以返回指令结束的处理程序，弹出所需信息来恢复执行。当接收到一个信号时，POSIX保存比上述做法多的信息。然后在进程能继续正在处理的工作前要执行一项另外的工作。这样存储管理器就必须向系统任务发送两条消息来处理一个信号。这样的好

处是信号处理的可靠性高。

当一个信号被送往一个进程时，SYS_SENDSIG消息被送往系统任务。该消息由do_sensig处理（第15157行）。处理POSIX风格信号所需的信息在一个包含处理器寄存器内容的sigcontext结构和一个包含信号如何被进程处理的sigframe结构中。这些结构都需要一些初始化，但do_sensig的基本工作只是把所需的信息放在进程的堆栈中，并调整进程的程序计数器和堆栈指针以便在下次调度程序允许进程执行时将能执行信号处理代码。

当一个POSIX风格的信号处理程序完成了它的工作后，和在旧的处理风格中一样，它不弹出被中断进程将恢复执行的地址。编写处理程序的程序员写一条return指令（或等效的高级语言），但SENDSIG调用对堆栈的操作将引起return执行一个SIGRETURN系统调用。然后存储管理器向系统任务发送一条SYS_SIGRETURN消息。该消息由do_sigreturn处理（第15221行），它把sigcontext结构拷贝回内核空间中，然后恢复进程的寄存器。被中断的进程将在下次调度程序允许它运行时从被中断的地址恢复执行，保留以前建立的任何特殊信号处理。

和本节中讨论的大部分其他调用不一样，SIGRETURN系统调用在POSIX中不是必须的。这是一个MINIX约定，它可以方便地在一个信号处理程序结束时初始化所需的处理。程序员不应该使用这个调用；它不能被其他操作系统识别，并且在任何情况下都不需要明确地提到它。

有些信号来自内核映象内部，或者在它们被送往存储管理器前由内核处理。这其中包括原来来自任务的信号，如来自时钟任务的闹钟，或由终端任务检测到的由按键引起的信号，还有CPU检测到的由异常（比如被0除或非法指令）引起的信号。原来来自文件系统的信号也首先由内核处理。SYS_KILL消息被文件系统用来请求产生一个这样的信号。这个名字可能会让人有一点误解。它与KILL系统调用的处理无关，而是用于普通进程的信号发送。这条消息由do_kill（第15276行）处理，对消息的有效来源进行一般的检查，然后调用cause_sig实际把信号传给进程。起源于内核的信号也由对该函数的调用传送，它通过向存储管理器发送一条KSIG消息来初始化信号。

无论何时存储管理器完成了这些KSIG类型之一的信号处理，它将向系统任务送回一条SYS_ENDSIG消息。该消息由do_endsig(第15294行)处理，它将减少等待信号的计数，如果该计数减少到0，就重置进程的SIG_PENDING位。如果没有其他标志被置位指示进程不能运行，就调用lock_ready允许进程再次运行。

SYS_COPY消息是最常用的一条消息。它被用来允许文件系统和存储管理器从用户进程拷贝信息。

当一个用户进程执行READ调用时，文件系统查看它的缓存中是否有所需的块。如果没有，它就向适当的磁盘任务发送一条消息把该块装入缓存。然后文件系统向系统任务发送一条消息通知它把该块拷贝到用户进程。在最坏的情况下，读一个块需要7条消息；在最好的情况下，需要4条消息。图3-51画出了这两种情况。这些消息是MINIX系统开销的来源，是高度模块化设计的代价。

图 3－51 (a)读一个块的最坏情况需要七条消息。(b)读一个块的最好情况需要四条消息。

在没有保护机制的8088中，可以很容易的欺骗文件系统使它把数据拷贝到调用者的地址空间，但这就违背了设计原则。任何使用这种旧机器的对提高MINIX性能感兴趣的人必须仔细研究这种机制以判断为了获得性能上的提高必须忍受多少不合适的行为。当然，这意外着在具有保护机制的Pentium类机器上这种提高性能的方法是不可能的。

处理SYS_COPY请求是很直接的。它由do_copy（第15316行）完成，由提取消息参数和调用phys_copy以及其他的一些动作组成。

处理消息传递机制效率较低的一种方法是把多个请求包装到一条消息中。SYS_VCOPY消息完成这个工作。这条消息的内容是一个指向指定在内存中移动的多个数据块的向量的指针。函数do_vcopy（第15364行）执行一个循环，提取源和目的地址和块长度并重复调用phys_copy直到所有的拷贝结束。这与磁盘设备基于一个请求处理多个传输的能力是相似的。

系统任务接受的消息类型还有另外几种，大部分都很简单。其中的两个一般在系统启动时使用。文件系统发送一条SYS_GBOOT消息请求启动参数。在include/minix/boot.h中说明的一个结构bparam_s允许MINIX在启动之前可以指定系统配置的各个方面。do_gboot（第15403行）函数完成这个操作，它只是从内存的一部分拷贝到另一部分。存储管理器也在启动时向系统任务发送一系列SYS_MEM消息来请求可用内存块的基址和大小。do_mem（第15424行）处理这个请求。

SYS_UMAP消息被一个非内核进程使用来请求计算给定虚拟地址的物理内存地址。do_umap（第15445行）通过调用umap执行这个动作，而umap是从内核中调用的处理这个转换的函数。

我们讨论的最后一条消息类型是SYS_TRACE，它支持用于调试的PTRACE系统调用。调试不是一个基本的操作系统功能，但操作系统的支持可以使它变得简单。在操作系统的帮助下，一个调试程序可以检查和修改由一个被检查进程使用的内存，以及在被调试程序不运行时存储在进程表中的处理器寄存器的内容。

一般地，一个进程一直运行到被阻塞以等待I/O或用完一定的时间。大部分CPU的设计都提供了使得一个进程可以只执行一条指令的方法，或者通过设置一个断点（breakpoint），使得进程只有在到达某一条指令时才被执行的方法。利用这些功能可以对程序进行仔细的分析。

使用PTRACE可以进行11种操作。其中只有一小部分可以完全由存储管理器完成，对于大部分操作存储管理器向系统任务发送一条SYS_TRACE消息，系统任务再接着调用do_trace（第15467行）。这个函数根据跟踪操作实现了一条switch语句。这些操作一般都很简单。在进程表中的一个的P_STOP位被MINIX用来识别是否处于调试

中，由停止进程的命令（case T_STOP）设置，或者重置以重新启动该进程（case T_RESUME）。调试依赖于硬件的支持，在Intel处理器中是由CPU的标志寄存器中的一位控制的。当该位被置位时，处理器只执行一条指令，然后就产生一个SIGMAP异常。和前面提到的一样，当向一个进程发送一个信号时，存储管理器就停止一个被跟踪的程序。这个TRACEBIT由T_STOP和T_STEP命令操纵。可以用两种方法来设置断点：或者用T_SETINS命令用一条产生一个SIGTRAP的特殊代码来替换一条指令，或者用T_SETUSER命令来修改特殊的断点寄存器。对任何MINIX可以移植的系统，都可以用相似的技术来实现一个调试器，但移植这些函数需要研究特定的硬件。

do_trace执行的大部分命令返回或修改在被跟踪进程的正文或数据空间中的值，或者是它的进程表表目中的值，并且代码的实现是很直接的。允许修改某些寄存器和CPU的标志位实在是太危险了，所以在处理T_SETUSER命令的代码中有许多检查防止发生这种操作。

在system.c的最后是可以在内核的各个部分使用的几个实用过程。当一个任务需要引发一个信号时（例如，时钟任务需要引发一个SIGALRM信号，终端任务需要引发一个SIGINT信号），它就调用cause_sig（第15586行）。这个过程在进程的进程表表目的p_pending域中设置某一位，然后检查存储管理器现在是否在等待来自ANY的消息，即它是否空闲并在等待下一个请求来处理。如果空闲，就调用inform通知存储管理器处理该信号。

如上所述，inform（第15627行）只有在检测到存储管理器不忙时才被调用。除了从cause_sig调用外，只要存储管理器被阻塞并且有未处理的内核信号时，它就从mini_rec中（在proc.c中）被调用。inform建立一个KSIG类型的操作，并把它发送到存储管理器。调用cause_sig的任务或进程在消息被拷贝到存储管理器的接收缓冲区中后立即继续运行。它不等待存储管理器的处理；如果在普通的发送机制中，这将引起发送者被阻塞。不过，在它返回之前，inform调用lock_pick_proc，由它调度存储管理器运行。由于任务具有比服务器更高的优先级，所以存储管理器直到所有的任务被满足之后才被运行。当任务结束时，将进入调度程序。如果存储管理器是优先级最高的可运行进程，那么它将被运行并处理信号。

过程umap（第15658行）是一个通用的把虚拟地址映射为物理地址的过程。如同我们已经注意到的，它由为SYS_UMAP消息服务的do_umap调用。它的参数是一个指向虚拟地址待映射的进程或任务的进程表表目的指针，一个指定正文、数据、或堆栈段的标志，虚拟地址本身，以及一个字节计数器。这个计数器很有用因为umap要检查以确保从虚拟地址开始的全部缓冲区都处于进程的地址空间中。为此，它必须知道缓冲区的大小。字节计数器并不用于映射本身，而只用于这个检查。所有从/向用户进程空间拷贝数据的任务都使用umap计算缓冲区的物理地址。对于设备驱动程序如果能够使用进程号作为参数而不是指向进程表表目的指针就可以得到umap的服务将更为方便。numap（第15697行）完成这个功能。它调用proc_addr来转换它的第一个参数，然后调用umap。

在system.c中定义的最后一个函数是alloc_segments（第15715行）。它由do_newmap调用。在初始化时它也被内核的main例程调用。这个定义十分依赖于硬件。它取出记录在进程表表目中的段赋值并操作Pentium处理器使用的寄存器和描述符以在硬件层支持段保护。

3.11 小结

输入/输出经常被忽视，但它是一个重要的课题。任何操作系统中都有相当重要的部分与I/O操作有关。我们由研究I/O硬件开始，分析了I/O设备和I/O控制器的关系，这是软件必须处理的问题。然后我们研究了I/O软件的四个层次：中断例程，设备驱动程序，设备无关I/O软件，以及在用户空间运行的I/O库和假脱机。

下面我们研究了死锁的问题，以及如何处理它。死锁发生在一组进程都拥有对某些资源的互斥存取权，并且每个进程还要求仍属于该组中另一个进程的资源时。所有进程都被阻塞，没有一个可以再次运行。死锁可以通过结构化系统来使得它不会发生。例如，在任何时刻一个进程只允许拥有一个资源。通过检测每个资源请求是否可能导致死锁（不安全状态）并否决或延迟那些可能引起麻烦的请求也可以避免死锁。

MINIX中的设备驱动程序是作为嵌入在内核中的进程来实现的。我们研究了RAM磁盘驱动程序，硬盘驱动程序，时钟驱动程序，以及终端驱动程序。同步闹钟任务与系统任务不是设备驱动程序但在结构上非常相似。这些任务都有一个主循环取出请求并进行处理，逐渐送回回答消息报告发生的事件。所有的任务都位于相同的地址空间中。RAM磁盘，硬盘，软磁盘驱动器任务都使用一个相同的主循环拷贝，并共享相同的函数。不过，每个任务仍然是独立的进程。几种不同的使用系统控制台，串行线和网络连接的终端都由一个单一的终端任务支持。

设备驱动程序和中断系统有许多不同的关系。能很快完成工作的设备如RAM磁盘和内存映象显示器根本不使用中断。硬盘驱动器任务在任务代码本身中完成了大部分工作，中断处理程序只返回状态信息。时钟中断处理程序本身执行了许多登记操作，只有在处理程序不能处理某项工作时才向时钟任务发送一条消息。键盘中断处理程序缓冲输入并且从不向它对应的任务发送消息，而是改变一个由时钟中断处理程序检测的变量，时钟中断处理程序将向终端任务发送一条消息。

习 题

1 假设芯片技术的发展可以将完整的控制器放在一个便宜的芯片上，其中包含所有的总线存取逻辑。这对图3-1的模型有什么影响？

2 如果一个磁盘控制器没有内部缓冲区，一从磁盘上收到字节就将它们写到内存中，请问这时交叉存取有用吗？试讨论之。

3 根据磁盘的旋转速率和几何尺寸，对于软盘和硬盘，在磁盘本身与控制器的缓冲区间传送的比特速率各为多少？这与其他形式的I/O（串行线和网络）如何进行比较？

4 一个双交叉编址的磁盘如图3-4(c)所示，每磁道有8个512字节的扇区，旋转速率为300rpm，假设磁头臂已被正确地定位，并且把扇区0置于磁头下需要旋转1/2圈，那么它要花多长时间才能依次读完一条磁道上的所有扇区？数据传输率是多少？同样条件下，对于具有相同特性但无交叉编址的磁盘上述问题如何回答？因为交叉编址而使数据传输速率降低多少？

5 多年前用于PDP—11的DM—11终端多路复用器，以七倍于波特率的速度在每一终端线（半双工）上采样来检查输入的数位是0还是1，在线上每次采样时间5.7微秒，则DM—11能够支持多少条1200波特的终端线？

6 一个局域网以如下方式被使用：用户发出一条系统调用，请求将数据包写到网上，然后操作系统将数据拷贝到一个内核缓冲区中，再将数据拷贝到网络控制器板上，当所有数据都安全存储于控制器中时，把它们在网上以10兆位/秒的速率传送，在每一位被发送后一微秒，接收的网络控制器将它保存。当最后一位到达时，目标CPU被中断，内核将新到达的数据包拷贝到内核缓冲区中进行检查。一旦它指明该数据包是发送给哪个用户进程的，内核就将数据拷贝到该用户进程空间。如果我们假设每一个中断及其相关处理过程费时1毫秒，数据包有1024字节（忽略头标），拷贝一个字节费时1微秒，将数据从一个进程注入另一个进程的最大速率是多少？假设发送者一直被阻塞直到接收端的工作完成并且返回一条确认消息。为简便起见，假设返回确认消息的时间可以忽略不计。

7 什么叫设备无关性？

8 以下的工作各在四个I/O软件层的哪一层完成？

(a)为一个磁盘读操作计算磁道、扇区、磁头。

(b)维护一个最近使用的块的缓冲。

(c)向设备寄存器写命令。

(d)检查用户是否有权使用设备。

(e)将二进制整数转换成ASCII码以便打印。

9 为什么打印机的输出文件在打印前通常都假脱机输出在磁盘上？

10 考虑图3-8。假设在步骤(o)中C需要S而不是R，这将会导致死锁吗？如果既要求S也需要R呢？

11 仔细观察图3-11(b)。如果Suzanne再多请求一个单位，将会导致一个安全的状态还是不安全的状态？如果请求来自Marvin而不是Suzanne呢？

12 图3-12中所有的轨迹都是垂直的或是水平的，你能想象出可能出现的对角线轨迹的情况吗？

13 假设图3-13中的进程A请求最后一台磁带驱动器。这会导致死锁吗？

14 一台计算机有6台磁带机被n个进程竞争，每个进程可能需要两台磁带机，那么n为多少时，系统没有死锁的危险？

15 一个系统可以处于既不死锁也不安全的状态吗？如果可以，举出例子；如果不可以，请证明所有状态均处于死锁或安全两种状态之一。

16 一个使用信箱的分布式系统有两条IPC原语：SEND和RECEIVE。后一个原语指定从哪个进程接收，如果该进程没有可用消息，那么即使消息可以等待其他进程。该原语也阻塞。不存在共享资源，但是进程因为其他原因需要经常通信。请讨论可能会发生死锁吗？

17 在一个电子转帐系统中，有数百个相同进程以如下的方式工作：每一进程读取一个输入行，指定一定数目的款项、贷方帐号、借方帐号，然后锁定两个帐号，传送这笔钱，完成后在解锁。由于许多进程并行运行，所以存在这样的危险：锁定x将无法锁定y，因为y已被一个正在等待x的进程锁定。设计一个方案来避免死锁。在没有完成事务处理前不要释放帐号记录。（换言之，锁定一个帐号时如果另一个帐号被锁定就立即释放它的处理方法是不允许的。）

18 银行家算法在一个有m个资源类和n个进程的系统中运行。在m和n都很大的情况下，为检查状态是否稳定而进行的操作次数正比于manb。a和b的值为多少？

19 Cinderella和Prince要离婚，为分割财产，他们商定了以下算法：每天早晨各人发函给对方律师要求财产中的一项。由于邮递信件需要一天的时间，他们商定如果发现在同一天两人要求了同一项财产，第二天他们将发信取消这一要求。他们的财产包括狗Woofer、Woofer的狗屋、金丝雀Tweeter和Tweeter的鸟笼。由于这些动物喜爱它们的房屋，所以又商定任何将动物和它们房屋分开的方案都无效，整个分配从头开始。Cinderella 和Prince都非常想要Woofer。于是他们分别去度假，并且每人都用一台个人电脑处理这一谈判工作。当他们度假回来，发现电脑仍在谈判，为什么？可能发生死锁吗？可能发生饥饿（永远等待）吗？

20 图3-15的消息格式被用来向块设备的驱动程序发送请求消息。如果可以，消息中的哪些字段在字符设备中可以省略？

21 磁盘请求以10、22、20、2、40、6、38柱面的次序到达磁盘驱动器。寻道时每个柱面移动需要6毫秒，计算以下寻道时间：

(a)先到先服务。

(b)下一个最邻近柱面。

(c)电梯算法（起始移动向上）。

所有情况下磁头臂起始都位于柱面20。

22 一个个人电脑销售商在访问阿坶斯特丹西南部的一所大学时，竭力推销他的产品，声称他的公司投入很大的努力使他们的UNIX版本速度非常快。例如，他指出他们的磁盘驱动程序使用电梯算法，并将一个柱面内的多个请求按扇区次序排队。学生Harry Hacker被打动并买了一台。回家后他写了一个程序随机地读分布在磁盘上的10000个块，令他惊讶的是，他测试的结果与先到先服务方式得出的结果一样。商人是否在骗人？
23 一个UNIX进程包含两部分 — 用户部分和核心部分，核心部分类似于子例程还是共行例程（coroutine）？
24 一个电脑的时钟中断处理程序每一滴答要占用2毫秒（包括所有的切换开销），时钟以60Hz的频率运行，那么CPU时间用于时钟处理的部分是多少？
25 教材中给出了两个监视定时器的例子：定时软盘马达的启动和允许硬拷贝终端上使用回车，请给出第三个例子。
26 为什么RS232终端是中断驱动，而内存映象终端不是中断驱动？
27 考虑一个终端如何工作：驱动程序输出一个字符，然后阻塞。当字符被打印完毕后，发生一条中断并且传送一条消息给阻塞的驱动程序，它输出下一个字符然后再次阻塞。如果传送一条消息，输出一个字符，然后阻塞所需时间为4毫秒，那么这一方法在波特率为110的传输线上能很好地工作吗？在4800波特的传输线上呢？
28 一个位映象终端包含1200*800个象素。为了滚动一个窗口，CPU（或者控制器）必须移动上面的文本的所有行，这通过将其所有比特从视频RAM的一个部分拷贝到另一个部分实现。如果一个窗口高66行，宽80个字符，（共5280个字符），每个字符宽8象素，高12象素。假设将字符输出到屏幕需要50微秒，如果以每字节500纳秒的速率进行拷贝，则滚动整个窗口需要多长时间？如果每行都是80个字符，那么终端的波特率是多少？对于每象素占4比特的彩色终端，计算其波特率（假设现在将一个字符输出到屏幕需要200微秒）。
29 操作系统为什么提供转义字符，例如MINIX中的CTRL-V。
30 MINIX驱动程序在接收到一个DEL（SIGKILL）字符后终端当前排队的所有输出，为什么？
31 许多RS232终端具有这样的转义序列：删除当前传输线并将其下边的所有线上移，你认为在终端内部如何实现这一特性？
32 在最初IBM PC的彩色显示器上，除了CRT电子束垂直回扫之后的任何时间向视频RAM中写数据都会导致屏幕上的“雪花”。一个屏幕为25x80字符，每个占用8x8象素。每行的640象素在电子束的一次水平扫描中绘出，包括水平会扫，该过程历时63.6微秒，屏幕每秒钟刷新60次，每次刷新均需要一个垂直回扫以使电子束回到屏幕顶端。该过程中可供写视频RAM的时间占多少？
33 为IBM彩色显示器或其他位映象显示器写一个图形驱动程序，它应该接收如下命令：设置并清除单个象素、移动屏幕上的矩形、以及你认为有趣的其他特性。用户程序与驱动程序的接口方式为打开/dev/graphics文件并向其中写入命令。
34 修改MINIX软盘驱动程序以完成每次一道高速缓冲。
35 实现一个软盘驱动程序使其作为一个字符设备，而非块设备来工作，以便旁路掉文件系统的块高速缓冲。在此方式下，用户可从磁盘读大量数据，这些数据通过DMA方式直接传到用户空间，以大幅度提高性能。该驱动程序主要针对那些需要不通过文件系统，而是机械地从磁盘读取比特的程序，例如文件系统检查工具。
36 实现MINIX中缺少的UNIX PROFIL系统调用。
37 修改终端驱动程序，使得除了一个用来删除前一个字符的特殊键外，另外再有一个键能够删除前一个单词。
38 MINIX系统中已经加入了一种新的支持可移动介质的硬盘设备。该设备在更换介质时必须旋转加速，而且这个时间相当长。预期在系统运行过程中介质更换很频繁。这样at_wini.c文件中的waitfor例程便无法满足需要。请设计一个新的waitfor例程，若所等待的比特模式在1秒钟的忙等待后仍未出现，则进入一个阶段，该阶段下磁盘任务将睡眠1秒钟，测试端口，并回去继续睡眠1秒钟，直到或者出现相应的比特模式，或者到达预先设置的TIMEOUT超时。

第四章 存储器管理

存储器是一种必须仔细管理的重要资源。虽然现在一台普通家用计算机的存储器容量可能是60年代早期全世界最大的计算机IBM 7094的五倍以上，但是程序长度的增长速度和存储器容量的增长一样快。用类似于帕金森定律的话来说： “存储器有多大，程序就会有多长”。在这一章中我们将讨论操作系统是怎样管理存储器的。

在理想的情况下，每个程序员都会喜欢的是无穷大、快速并且内容不易变（即掉电后内容不会丢失）的存储器，同时又希望它是是廉价的。但不幸的是当前的技术没有能够提供这样的存储器，因此大部分的计算机都有一个存储器层次结构，它由少量的非常快速、昂贵、易变的的高速缓存（cache），若干兆字节的中等速度、中等价格、易变的主存储器（RAM），和数百兆或数千兆字节的低速、廉价、不易变的磁盘组成。操作系统的工作就是协调这些存储器的使用。

操作系统中管理存储器的部分称为存储管理器，它的任务是跟踪哪些存储器正在被使用、哪些存储器空闲，在进程需要时为它分配存储器，使用完毕后释放存储器，并且在主存无法容纳所有进程时管理主存和磁盘间的交换。

在这一章中我们将讨论许多不同的存储器管理方案，从非常简单的到高度复杂的。我们从最简单的存储管理系统开始，然后逐步过渡到更加精密的系统。

4.1基本的内存管理

存储管理系统可以分为两类：在运行期间将进程在主存和磁盘之间移动的系统（交换和分页）和不移动的系统。后一种比较简单，因此我们首先讨论。随后在本章的后半部分我们将讨论交换和分页。在本章中读者应该自始至终清醒地认识到：交换和分页在很大程度上是由于缺少足够的主存以同时容纳所有的进程而引入的。随着主存越来越便宜，选择某种存储器管理方案的理由也许会变得过时，除非程序变大的速度比存储器降价的速度还要快。

4.1.1没有交换和分页的单道程序

最简单的存储器管理方案是同一时刻只运行一道程序，应用程序和操作系统共享存储器。这种方案的三个变种如图4-1所示。操作系统可以位于主存最低端的随机存取存储器（RAM）中，如图4-1(a)所示；它也可以位于主存最高端的只读存储器（ROM）中，如图4-1(b)所示；还可以让设备驱动程序位于内存高端的ROM中而让操作系统的其他部分位于低端的RAM中，如图4-1(c)所示。例如小型的MS-DOS系统使用的就是最后一种模型。在IBM PC计算机中，系统位于ROM中的部分称为基本输入输出系统（BIOS）。

图4-1 在一个操作系统一个用户进程时三种简单的内存组织方法，其他方法也是存在的。

当这样组织系统时，同一时刻只能有一个进程在存储器中运行。一旦用户输入了一个命令，操作系统就把需要的程序从磁盘拷贝到存储器中并执行它；在进程运行结束后，操作系统显示出一个提示符并等待新的命令。当收到新的命令时它把新的程序装入存储器，覆盖掉原来的程序。

4.1.2固定分区的多道程序

虽然单道程序常常用于带有简单操作系统的小型计算机上，但我们往往更加希望同时能有多个进程同时运行。在分时系统中，允许多个进程同时在存储器中，这意味着当一个进程因为等待I/O结束而阻塞时，其他的进程可以利用CPU，因而提高了CPU的利用率。即使在个人计算机上，同时运行两个或多个进程也常常是很有用的。

实现多道程序的最容易的办法是把主存简单地划分为n个分区（可能不相等），分区的划分可以在系统启动时手工完成。

当一个作业到达时，可以把它放到能够容纳它的最小的分区的输入队列中。因为在这种方案中分区大小是固定的，一个分区中未被作业使用的空间就白白浪费掉了。图4-2(a)所示的就是这种固定分区、各分区有自己独立的输入队列的系统。

图4-2 (a)各分区具有独立输入队列固定内存分区。(b)仅有单个输入队列的固定内存分区。

把输入作业排成多个队列时，在大分区的队列为空而小分区的队列为满的情况下，其缺点就变得很明显，如图4-2(a)中的分区1和3所示。另一种方法如图4-2(b)所示，只使用一个队列，当一个分区空闲时，选择最靠近队列头可以被该分区容纳的作业装入其中运行。因为我们不希望为了一个小作业而浪费一个大分区，所以另一个策略是搜索整个输入队列找出该分区能容纳的最大的作业，这种算法认为不值得给小作业一个完整的分区，然而通常我们更加期望给小作业（假设是交互作业）最好的服务，而不是最差的。一个解决方法是至少保留一个小分区，这样就不必为了使小作业运行而为其分配大的分区。另一个方法是采用这样一个规则：一个可运行的作业最多只能够被跳过k次。一个作业每被跳过一次就得一分，当它得到k分时它就不能再被跳过了。

这种由操作员在早晨设置好随后就不能再被改变的固定分区的系统曾在IBM大型机的OS/360中使用了许多年，它被称为MFT（具有固定数目任务的多道程序，或OS/MFT），它易于理解也易于实现：输入作业被送入队列排队直到有合适的分区可用，随后作业被装入分区运行直到它结束。现在只有极少数操作系统，如果还有的话，支持这种模式。

重定位和保护

多道程序引入了两个必须解决的问题—重定位和保护。从图4-2可以清楚地看出不同的作业将在不同的地址区间运行。在一个程序被链接时（即把主程序、用户编写的例程、库例程结合到同一个地址空间中），链接器必须知道程序将在主存的什么地址开始运行。

例如，假设程序的第一条指令是调用在链接器产生的二进制文件中绝对地址为100的一个过程。如果程序被装入分区1，这条指令跳转的目的地址将是绝对地址100，这会造成混乱，因为该地址在操作系统的内部。其实真正应该被调用的地址是100K+100。如果程序被装入分区2，它就应该去调用200K+100，等等。这就是重定位问题。一个可能的解决方法是在程序装入主存时直接修改指令，装入分区1的程序在每个地址上加100K，装入分区2的程序在每个地址上加200K，等等。为了在装入时能这样重定位，链接器必须在二进制程序中包含位图或链表，由他们指明那些程序字是需要进行重定位的地址，那些是操作码、常数和其他不能进行重定位的元素。OS/MFT就是这样工作的，一些微机也是这样工作的。

在装入时重定位并没有解决保护问题，一个恶意的程序总可以生成一条新指令并跳转到这条指令执行。因为在这个系统中使用的是绝对地址而不是相对于某个寄存器的地址，没有办法能阻止程序生成读或写主存任何位置的指令。在多用户系统中，我们不希望一个进程读写属于另一个用户的主存空间。

IBM采用的保护360机器的办法是将主存划分为2K字节的块并为每个块分配4位的保护码。PSW中包含一个4位的密钥，若运行进程试图对保护码不同于PSW中密钥的主存进行访问，则由硬件引起一个陷入。因为只有操作系统能够修改保护码和密钥，这种办法能有效地阻止用户进程干涉其他进程或操作系统本身。

另一个既针对重定位又针对保护问题的解决方法是在机器中设置两个专门的寄存器，称为基址和界限寄存器。在一个进程被调度到时，它的分区的起始地址被装入基址寄存器，分区的长度被装入界限寄存器。进程产生的

每一个地址在访问主存前被自动加上基址寄存器的内容，因此如果基址寄存器是100K，不用修改指令，一条 CALL 100 指令就被有效地转换为一条CALL 100K+100指令。指令还被自动地用界限寄存器进行检查以确保他们没有试图访问当前分区以外的地址。基址和界限寄存器受到硬件保护，以防止用户程序修改他们。
CDC 6600—世界上第一台巨型机—使用了这个方案。用于初期IBM PC的Intel 8088 CPU使用了这个方案的一个较弱的版本—有基址寄存器，但没有界限寄存器。从286开始，采用了一种更好的方案。

4.2交换

在批处理系统中，把主存组织为固定的分区是简单而且高效的，每个作业在排到队列头时被装入一个分区，它停留在主存中直到运行完毕。只要有足够的作业能被保持在主存中以使CPU始终处于忙的状态，那么就没有理由使用任何更加复杂的方案。
但在分时系统或面向图形的个人计算机中情形就不同了，有时会没有足够的主存以容纳所有当前活动的进程，多出的进程必须被保存在磁盘上并动态地调入主存运行。
在硬件支持下，有两个通用的内存管理方法可以使用。最简单的策略称为交换（swapping），它把各个进程完整地调入主存，运行一段时间，再放回到磁盘上；另一种策略称为虚拟存储器（virtual memory），它使进程在只有一部分在主存的情况下也能运行。下面我们将先讨论交换系统，在4-3中我们将讨论虚拟存储器。
交换系统的操作如图4-3所示，开始时只有进程A在主存，随后进程B和C被创建或从磁盘上被调入，在图4-3(d)中A结束了或被交换到了磁盘上，然后D进入，接着B离开，最后E进入。
图4-3　内存分配情况随着进程进出内存而变化，阴影表示的区域是未使用的内存。
图4-2所示的固定分区与图4-3所示的可变分区的主要区别是：在后者中分区的数量、位置、大小随着进程的出入是动态变化的，而在前者中他们是固定不变的。这种可变分区不再受固定分区可能太大或太小的约束，从而提高了主存的利用率，但它也使内存的分配、释放和对各个内存块的跟踪更加复杂。
当交换在主存中生成了多个空洞时，可以把所有的进程向下移动至相互靠紧，从而把这些空洞结合成一大块，这种技术称为内存紧缩（memory compaction）。我们通常不进行这个操作，因为它需要大量的CPU时间，例如在一个有32M主存，每微秒可以拷贝16个字节的计算机上把全部内存紧缩一次需要两秒钟。
一个值得关心的问题是在一个进程被创建或换进时应该为它分配多大的内存。如果进程创建时的大小是固定的并且不会改变，那么分配是很简单的：完全根据需要的大小进行分配。
然而如果进程的数据段可以增长，例如在许多程序设计语言中都允许动态地从堆中分配内存，那么进程一旦试图增长时问题就出现了，如果该进程邻接着一个空洞就可以把这个空洞分配给它；然而如果进程邻接的是另一个进程，则需要增长的进程或者不得不被移动到内存中一个足够大的空洞中去，或者必须把一个或多个进程交换出去以生成一个足够大的空洞。如果一个进程不能在内存中增长并且磁盘上的交换区已经满了，那么这个进程必须等待或被杀死。
如果大部分进程在运行时都要增长，那么为了减少进程因为所在的内存区域不够而引起的交换和移动所带来的开销，可以采用的一种方法是：在进程被换进或移动时为其分配一点额外的内存。当然，在进程被换出到磁盘上时应该只交换进程实际使用的内存中的内容，将额外的内存交换出去纯粹是浪费。在图4-4(a)中我们可以看到一个已经为两个进程分配了增长空间的内存配置。
图4-4　(a)为能够增长的数据段预留空间。(b)为能够增长的数据段和堆栈段预留空间。
如果进程有两个可增长的数据部分，例如一个供动态分配和释放的变量使用的作为堆的数据段和一个存放普通局部变量和返回地址的栈段，那么可以使用另一种安排，如图4-4(b)所示。在这个图中我们可以看到所示进程的栈段在进程所占内存的顶端并向下增长，紧接在正文段后面的数据段向上增长，处于这两个段之间的内存，他们都可以使用，如果用完了，则这个进程或者必须被移动到足够大的空洞中，或者交换出内存直到内存中有足够的空间，或者被杀死。

4.2.1使用位图的内存管理

动态分配的内存必须由操作系统管理。一般来说有两种方法跟踪内存的使用情况：位图和自由链表。本节和下一节将逐个讨论这两种方法。
在使用位图方法时，内存被划分为可能小到几个字或大到几千字节的分配单位，每个分配单位对应于位图中的一位，0表示空闲1，表示占用（或者反过来）。图4-5表示出了一个内存片段和对应的位图。
图4-5　(a)一段有五个进程和三个空洞的内存，刻度表示内存分配的单位，阴影表示空闲区域（在位图中用0表示）。(b)对应的位图。(c)用列表表示的同样的信息。
分配单位的大小是一个重要的设计因素。分配单位越小，位图越大，但是即使分配单位只有4个字节大小，32位的内存也只需要位图中的1位，32n位的内存只需要n位的位图，因此位图只占用了1/33的内存。如果分配单位选的比较大，需要的位图就比较小，但是如果进程的大小不是分配单位的整数倍，那么最后一个分配单位中相当数量的内存就可能被浪费掉。
因为位图的大小仅仅取决于内存和分配单位的大小，它提供了一个简单的使用固定大小内存就能对内存使用情况进行跟踪的方法。它的主要问题是当它决定把一个占k个分配单位的进程调入内存时，内存管理器必须搜索位图以找出一串k个连续的0。在位图中查找指定长度的连续0串是一个缓慢的操作（因为串可能跨越字边界）。这是反对使用位图的一个理由。

4.2.2使用链表的内存管理

跟踪内存使用的另一个方法是维持一个已分配和空闲的内存段的链表，这里，一个段或者是一个进程，或者是两个进程间的一个空洞。图4-5(a)的内存可以用图4-5(c)所示的段链表来表示。表中的每一个表项都包含下列内容：指明是空洞(H)还是进程(P)的标志、开始地址、长度、和指向下一个表项的指针。

在这个例子中，段链表是按照地址排序的，这样作的好处是在进程结束或被换出时更新链表十分直观。一个要结束的进程一般有两个邻居（除非它是在内存的最低端或最高端），他们可能是进程也可能是空洞，这导致了图4-6所示的四种组合。在图4-6(a)中更新链表需要把P替换为H；在图4-6(b)和4-6(c)中两个表项被合并成为一个，链表变短了一个表项；在图4-6(d)中三个表项被合并为一个，两个表项被从表中删除。因为结束进程的进程表表项中通常含有指向对应于它的段链表表项的指针，因此这个链表使用双链表可能要比图4-5(c)所示的单链表更方便，这样更易于找到上一个表项以检查是否可以合并。

图4-6　进程X终止时四种与邻居合并的方式。

当进程和空洞按照地址顺序存放在链表中时，好几种算法都可以用来为新创建和换进的进程分配空间，这里我们假设存储管理器知道要分配的内存的大小。最简单的算法是首次适配算法，存储管理器沿着内存段链表搜索直到找到一个足够大的空洞，除非空洞大小和要分配的空间大小刚好一样，否则的话这个空洞将被分为两部分，一部分供进程使用，另一部分是未用的内存。首次适配算法是一种快速的算法，因为它尽可能地少搜索。

首次适配的一个较小变形是下次适配，它的工作方式和首次适配相同，区别是每次找到合适的空洞时都记住当时的位置，在下次寻找空洞时从上次结束的地方开始搜索，而不是每次都从头开始。Bays(1977)的仿真指出下次适配的性能略低于首次适配。

另一个大家熟知的算法是最佳适配算法，它搜索整个链表以找出够用的最小的空洞。最佳适配算法试图找出最接近实际需要的大小的空洞，而不是把一个以后可能会用到的大空洞先使用。

作为首次适配和最佳适配算法的例子，让我们再观察图4-5，假如需要一个大小为2的块，首次适配将分配在位置5的空洞，而最佳适配将分配在位置18的空洞。

由于最佳适配算法每次被调用时都要搜索整个链表，它要比首次适配算法慢，有点出乎意料的是它还会导致比首次适配更多的内存浪费，因为它倾向于生成大量没用的很小的空洞，而总的来说首次适配算法生成的空洞更大。

为了避免最接近适配的空洞会分裂出极小空洞的问题，大家可能会想到最差适配，即总是分配最大的空洞，以使分裂出来的空洞比较大从而可以继续使用，但仿真说明最差适配也同样不是一个好主意。

如果把进程和空洞放在不同链表中，那么这四个算法的速度都能得到提高，这样他们就能只检查空洞而不是进程。但这种分配速度的提高的一个不可避免的代价就是复杂度提高和内存释放速度变慢，因为一个释放的内存段必须从进程链表中删除并插入空洞链表。

如果进程和空洞使用不同的链表，空洞链表可以按照大小排序以提高最佳适配的速度。在最佳适配算法搜索由小到大排列的空洞链表时，当它找到一个合适的空洞时它就知道这个空洞是能容纳这个作业的最小的空洞，因此是最佳的，不需要象在单个链表的情况那样继续进行搜索。当空洞链表按大小排序时，首次适配与最佳适配一样快，而下次适配则毫无意义。

在空洞被保存在不同于进程的链表中时我们可以作一个小小的优化：不用单独的数据结构存放空洞链表，取而代之用空洞自己。每个空洞的第一个字可以是空洞大小，第二个字指向下一个空洞，于是图4-5(c)中三个字加一位(P/H)的那些链表结点就不再需要了。

还有一种分配算法叫做快速适配，它为一些经常被用到长度的空洞设立单独的链表。例如，它可能有一个n个项的表，这个表的第一个项是指向长度为4K的空洞的链表的表头的指针，第二个项是指向长度为8K的空洞的链表的指针，第三个项指向长度12K的空洞链表，等等。象21K这样的空洞既可以放在20K的链表中也可以放在一个专门的存放大小比较特别的空洞的链表中。快速适配算法寻找一个指定大小的空洞是十分迅速的，但它有一个所有将空洞按大小排序的方案所共有的一个缺点，即在一个进程结束或被换出时寻找它的邻接块以查看是否可以合并是非常费时间的。如果不作合并，内存会很快分裂成大量的进程无法使用的小空洞。

4.3 虚拟存储器

许多年以前人们第一次遇到了太大以至于内存容纳不下的程序，通常采取的解决方法是把程序分割成多个叫做覆盖块的片段，覆盖块0首先运行，在它结束时它将调用另一个覆盖块。有一些覆盖系统非常复杂，允许多个覆盖块同时在内存中。覆盖块存放在磁盘上，在需要时由操作系统动态地换入换出。

虽然覆盖块换入换出的实际操作都是由系统完成的，但是必须由程序员把程序分割成片段。把一个大程序分割成小的、模块化的片段是非常费时和枯燥的。不久就有人找到了一个把全部工作都交给计算机的办法。

这个方法(Fotheringham, 1961)被称作虚拟存储器（virtual memory）。虚拟存储器的基本思想是程序、数据、堆栈的总的大小可以超过可用物理存储器的大小，操作系统把程序当前使用的那些部分保留在存储器中，而把其他部分保存在磁盘上。例如一个16M的程序，通过仔细地选择在各个时刻将哪4M内容保留在内存中，并在需要时在内存和磁盘间交换程序的片段，那么就可以在一个4M的机器上运行。

虚拟存储器也可以工作在许多程序的片段同时存放在内存的多道程序系统中。当一个程序等待它的一部分被调入时，它是在等待I/O操作而不能运行，因此CPU可以象在任何其他多道程序系统中一样，交给另一个进程使用。

4.3.1分页

大部分虚拟存储器系统都使用了一种称为分页（paging）的技术，我们这里将讨论它。在任何一台计算机上，都存在一个程序能够产生的内存地址的集合。当程序执行这样一条指令时：

MOVE REG, 1000

它把地址为1000的内存单元的内容复制到REG中（或者反过来，这取决于计算机的型号）。地址可以通过索引、基址寄存器、段寄存器和其他方式产生。

这些由程序产生的地址被称为虚地址（virtual addresses），他们构成了一个虚地址空间（virtual address space）。在没有虚拟存储器的计算机上，虚地址被直接送到内存总线上，使具有同样地址的物理存储器字被读写；而在使用虚拟存储器的情况下，虚地址不是被直接送到内存总线上，而是送到内存管理单元(MMU)，它由一个或一组芯片组成，其功能是把虚地址映射为物理地址，如图4-7所示。

图4-7　MMU的位置和功能。

图4-8是一个非常简单的的例子，它演示这个映射如何工作。在这个例子中，我们有一台可以生成16位地址的计算机，地址变化范围从0到64K，这些地址是虚地址。然而这台计算机只有32K的物理存储器，因此虽然可以编写64K的程序，他们却不能被完全调入内存运行。在磁盘上必须有一个可以大到64K的程序的完整内核映象，以保证程序片段在需要时能被调入。

图4-8　虚地址与物理内存地址之间的关系由页表给出。

虚地址空间被划分成称为页（pages）的单位，在物理存储器中对应的单位称为页框（page frames），页和页框总是同样大小的，在这个例子中是4K，但在现有的系统中常用的页的大小是从512字节到64K。对应于64K的虚地址空间和32K的物理存储器，他们分别包含有16个虚页和8个页框。内存和磁盘之间的传输总是以页为单位的。当程序试图访问地址0时，比如使用这条指令：

MOVE REG, 0

虚地址0将被送往MMU，MMU看到虚地址落在页0范围内（0到4095），根据它的映射这个页对应的是页框2（8192到12287），因此MMU把地址变换为8192并把地址8192送到总线上。内存板对MMU一无所知，它只看到一个对地址8192读或写的请求并且执行它。MMU从而有效地把所有从0到4095的虚地址映射到了8192到12287的物理地址。

同样地，指令：

MOVE REG, 8192

被有效地转换为：

MOVE REG, 24576

因为虚地址8192在虚页2中并且这个虚页被映射到了物理页框6（物理地址从24576到28671）。第三个例子，虚地址20500在虚页5（虚地址20480到24575）距开头20字节处，被映射到物理地址12288 + 20 = 12308。

通过适当地设置MMU，它可以把16个虚页映射到8个页框中的任何一个，但是这种能力本身并没有解决虚地址空间比物理存储器大的问题。在图4-8中我们只有8个物理页框，于是只有8个虚页被映射到了物理存储器中，其他在图中用叉号表示的页没有被映射。在实际的硬件中，每个虚页都有一个Present/absent位指出该页是否被映射了。

在程序试图访问未映射的页时，例如通过下列指令，会发生什么 ？

MOVE REG, 32780

虚页8（从32768开始）的第12个字节所对应的物理地址是什么呢？MMU注意到这个页没有映射（在图中用叉号表示），于是使CPU陷入操作系统，这个陷入称为缺页故障。操作系统找到一个很少使用的页框并把它的内容写入磁盘，随后把需引用的页取到刚才释放的页框中，修改映射，然后重新启动引起陷入的指令。

例如，假设操作系统决定放弃页框1，那么它将把虚页8装入物理地址4K，并对MMU作两处修改：首先，它要标记虚页1为未映射，以使以后任何对虚地址4K到8K的访问都引起陷入；随后把虚页8对应表项的叉号改为1，因此在引起陷入的指令重新启动时，它将把虚地址32780映射为物理地址4108。

现在让我们看看MMU的内部结构以了解它是怎么工作的、以及为什么我们选用的页面大小是2的次幂。在图4-9中我们可以看到一个虚地址的例子，虚地址8196（二进制是0010000000000100）用图4-8所示的MMU的映射进行映射，输入的16位虚地址被分为4位的页号和12位的偏移量，4位的页号可以表示16个页面，12位的偏移可以为一页内的全部4096个字节编址。

图4-9　在16个4K页的情况下MMU的内部操作。

页号用作页表的索引，以得出对应于这个虚页的页框号。如果Present/absent位是0，则将引发一个操作系统的陷入；如果这个位是1，在页表中查到的页框号将被复制到输出寄存器的高三位中，加上输入虚地址中的12位偏移，就构成了15位的物理地址。输出寄存器的内容随即被作为物理地址送到内存总线。

4.3.2页表

从理论上来讲，虚地址到物理地址的映射就象我们刚才描述的那样。虚地址被分成虚页号（高位）和偏移（低位）两部分，虚页号被用做页表的索引以找到该虚页对应的页表项，从页表项中可以找到页框号（如果有的话）。随后页框号被拼接到偏移的高位端，形成送往内存的物理地址。

页表的目的是把虚页映射为页框。从数学角度而言，页表是一个函数，它的参数是虚页号，结果是物理页框号。

通过这个函数可以把虚地址中的虚页域替换成页框域，从而形成物理地址。
在这个简单的描述之外，我们还必须面对两个主要问题：
1. 页表可能会非常大。
2. 地址映射必须十分迅速
第一点是因为现代计算机使用的虚地址最少也有32位，在页长为4K时32位的地址空间有1百万个页，64位的地址空间则更要多得多。虚地址空间中的1百万个页需要有1百万个表项的页表，并且每个进程都需要有自己的页表。
第二点是因为虚地址到物理地址的映射在每次内存访问时都要进行。一条典型的指令有一个操作字，通常还有一个内存操作数，因此每条指令都要进行一次、两次或多次的页表引用。假如执行一条指令需要10纳秒，则页表的查找必须在几个纳秒内完成以避免成为一个主要的瓶颈。
对大而快速的页映射的需要是计算机构造方式的一个重要限制,这个问题不仅在最高档的计算机中是最重要的，在强调性能/价格比的低档机中也是一个重要的问题。在本节和下几节中我们将详细研究页表的设计和已经实际使用的一些解决方法。
最简单的设计（至少在理论上）是使用一个由一组快速的硬件寄存器组成的页表，虚页号作为索引，每个虚页一个表项。在启动一个进程时，操作系统把位于主存中的进程页表装入寄存器，在进程运行期间不用因为页表而访问内存。这一方法的优点是直观并且在映射时不需要访问内存，它的一个缺点是非常昂贵（如果页表很大的话），而且在每次进程切换时都要装入位于主存中的页表，这也会降低性能。
另外一个极端是把页表全部放在主存中，这时，需要的全部硬件仅仅是一个指向页表起始地址的寄存器，在上下文切换时只要重新装入这个寄存器就可以改变内存映射。当然，它的一个严重缺点是在执行每条指令时都需要一次或多次访问内存读取页表项，因此这个这种方法很少以它最单纯的方式使用，下面我们将研究它一些性能要好得多的变种。
多级页表
为了避免把庞大的页表始终保存在内存中,许多计算机使用了多级页表,图4-10是一个简单的例子。在图4-10(a)中，32位的虚地址划分为10位的PT1域、10位的PT2域和12位的偏移。因为偏移是12位，所以页长是4K，页面共有220个。
图4-10　(a)一个有两个页表域的32位地址。(b)二级页表。
多级页表的秘密是避免把所有的页表一直保存在内存中，特别是那些不需要的就不应该保留。比如一个需要12兆字节的进程，最低端是4兆程序正文，后面是4兆数据，顶端是4兆堆栈，在数据顶端上方和堆栈底端之间的上吉（千兆）字节的空洞根本没有使用。
让我们看看图4-10(b)的例子中二级页表是怎样工作的。在左边是顶级页表，它具有1024个表项，对应于10位的PT1域。当一个虚地址被送到MMU时，MMU首先抽出虚地址的PT1域并把它作为访问顶级页表的索引。整个4G虚地址空间（32位）已经被切成1024块，每块4兆字节，这1024个表项中每个表项都表示其中的一块。
通过索引得到的顶级页表项包含有二级页表的地址或页框号，顶级页表的表项0指向程序正文的页表、表项1指向数据的页表、表项1023指向堆栈的页表。其他的表项（用阴影表示的）未用。PT2域被用作访问选定的二级页表的索引以找到该虚页的页框号。
举一个例子，虚地址0x00403004（十进制4,206,596）位于数据部分12,292字节处。它对应的PT1=1、PT2=3、偏移=4。MMU首先用PT1作为索引访问顶级页表得到表项1，它对应的地址范围是4M到8M；随后它用PT2作为索引访问刚刚找到的二级页表并得到表项3,它对应的地址范围是在它的4M块内的12288到16383(即绝对地址4,206,592到4,210,687）。这个表项含有虚地址0x00403004所在页的页框号。如果这个页不在内存中，Present/absent位将是0，一个页面故障将被引发；如果这个页在内存中，从二级页表中得到的页框号将和偏移(4)结合构成物理地址并通过总线送到存储器。
在图4-10中值得注意的是虽然地址空间超过一百万个页，实际上只需要四个页表：顶级页表，0到4M、4M到8M和顶端4M的二级页表。顶级页表中有1021个表项的Present/absent位都被设为0，使得访问他们时强制产生一个页面故障。如果这种情况发生了，操作系统将注意到进程在试图访问一个不期望被访问的地址并采取适当的行动，比如向进程发出一个信号或杀死进程。在这个例子中各种长度我们选择的都是整的数（正文、数据、堆栈都是4M长），并且选择PT1与PT2等长，但在实际中其他的值当然也是可能的。
图4-10的二级页表可以扩充为三级、四级或更多级。更多的级别带来了更多的灵活性，但页表超过三级所带来的复杂性是否值得令人怀疑。
现在让我们把注意力从页表的总体结构转移到单个页表项的细节上来，表项的结构是与机器密切相关的，但不同机器的表项存储的信息大致相似。在图4-11中我们给出了页表项的样本，不同计算机页表项的大小不一样，但32位是一个一般的尺寸。最重要的域是页框号，毕竟页映射的目的是找到这个值；其次是存在/不在(Present/absent)位，这一位是1时这个表项是有效的可以被使用，如果是0，表示这个表项对应的虚页现在不在内存中，访问这一位为0的页会引起一个页面故障。
图4-11　一个典型的页表项。
保护位指出这个页允许什么样的访问。在最简单的形式下这个域只有一位，0表示读写，1表示只读。一个更先进的安排是使用三位，各位分别指出是否允许读、写、执行这个页。

修改(Modified)和引用(Referenced)位跟踪页的使用。在一个页被写入时硬件自动设置修改位，此位在操作系统重新分配页框时是非常有用的，如果一个页已经被修改过（即它是“脏”的），则必须把它写回磁盘，否则只用简单地把它丢弃就可以了，因为它在磁盘上的拷贝仍然是有效的。此位有时也被称为脏(dirty)位，因为这反映了该页的状态。

引用位在该页被引用时设置，不管是读还是写。它的值被用来帮助操作系统在发生页面故障时选择淘汰的页，不在使用的页要比在使用的页更适合于被淘汰。这个位在我们即将讨论的好几个页面替换算法中都将扮演重要的角色。

最后一位可以禁止该页被缓存，这个特性对那些映射到设备寄存器而不是常规内存的页面是非常重要的。假如操作系统正在紧张地循环等待某个I/O设备对它刚发出的命令作出响应，保证硬件是不断地从设备中读取而不是读取一个旧的被缓存的拷贝是非常重要的，通过这个位可以禁止缓存。具有独立的I/O空间而不使用内存映射I/O的机器不需要这个位。

值得注意的是当一个页面不在内存时保存该页的磁盘地址不是页表的一部分，原因很简单，页表只保存硬件把虚地址转换为物理地址时所需要的信息，操作系统在处理页面故障时需要的信息保存在操作系统内部的软件表格中。

4.3.3 TLBs—翻译后援存储器(Translation Lookaside Buffers)

由于页表尺寸较大，因此许多分页方案都只能把它保存在内存中，但这种设计对性能有很大的影响。例如一条把一个寄存器的内容复制到另一个寄存器中指令，在不使用分页时只需要访问内存一次取指令，而在使用分页时需要额外的内存访问去读取页表。因为系统的运行速度一般都是被CPU从内存中取得指令和数据的速率限制的，在每次访问内存时都要去访问两次页表会使机器的性能降低2/3，这样的系统是没有人会愿意用的。

计算机设计者们很早就意识到了这个问题并提出了一个解决方案，这个方案基于这样的观察：大部分程序倾向于对较少的页面进行大量的访问。因此只有一小部分页表项经常被用到，其他的很少被使用。

采取的解决方法是为计算机装备一个不需要经过页表就能把虚地址映射成物理地址的小的硬件设备，这个设备叫做TLB（翻译后援存储器，Translation Lookaside Buffer），有时也叫做相联存储器（associative memory），如图4-12所示。它通常在MMU内部，条目的数量较少，在这个例子中是8个，一般很少超过64个。每个条目包含一个页的信息，主要有虚页号、在页被修改时将被设置的一个位、保护码（读/写/执行允许）、和页所在的物理页框号，这些域和页表中的域是一一对应的，还有另外一位指出这个条目是否有效。

图4-12　一个用于加速分页操作的TLB。

生成图4-12所示的TLB的一个可能的例子是一个正在执行某个循环的进程，该循环跨越虚页19、20、21，因此这些页面的保护码是读和执行。正在使用的主要数据（比如一个正在被处理的数组）在页面129和130，页140包含了数组演算中用到的索引。最后，堆栈在页面860和861。

现在让我们看一看TLB是怎样工作的，当一个虚地址被送到MMU翻译时，硬件首先把它和TLB中的所有条目同时（并行地）进行比较，看它的虚页号是否在TLB中，如果找到了并且这个访问没有违反保护位，它的页框号将直接从TLB中取出而不用去查页表。如果虚页号在TLB中但当前指令试图写一个只读的页面，这时将发生一个保护故障，与直接访问页表时相同。

有趣的是当虚页号不在TLB中时会发生什么？如果MMU发现在TLB中没有命中，它将随即进行一次常规的页表查找，然后从TLB中淘汰一个条目并把它替换为刚刚找到的页表项。因此如果这个页很快会再被用到的话，第二次访问时它就能在TLB中直接找到。在一个TLB条目被淘汰时，被修改的位被复制回在内存中的页表项，其他的值则已经在那里了。当TLB从页表中装入时，所有的域都从内存中取得。

软件TLB管理

直到现在我们一直假设，每个带有页式虚拟存储器的机器都有能被硬件识别的页表外加一个TLB，在这个设计中TLB的管理和TLB故障的处理都完全由MMU硬件完成，只有一个页不在内存时才会陷入操作系统。

在过去，这个假设是对的，但是在一些现代的RISC机中，包括MIPS、Alpha、HP PA，几乎全部的这种页管理工作都是由软件完成的。在这些机器中，TLB条目是由操作系统显式地装入的，在TLB没有命中时，MMU不是到页表中找到并装入需要的页面信息，而是产生一个TLB故障把问题交给操作系统，操作系统必须找到页面，从TLB中淘汰一个条目，装入一个新的，然后重新启动产生故障的指令。当然，所有这些都必须用很少几条指令完成，因为TLB不命中发生的频率远比页面故障大得多。

令人惊奇的是，如果TLB的尺寸取一个合理的较大值（比如64个条目）以减少不命中的频率，那么软件管理的TLB效率可能相当高。这里主要的收益是一个简单得多的MMU，它在CPU芯片上为高速缓存和其他能提高性能的部件让出了相当大的面积。Uhlig等详细地讨论了软件TLB管理(1994)。

人们已经使用了很多方法来提高使用软件管理TLB机器的性能，有一个方法既能减少TLB的不命中率又能减少在TLB不命中确实发生时的开销（Bala等，1994）。为了减少TLB的不命中，操作系统有时可以用它的直觉来指出那些页可能将被使用并把他们预装入TLB中。例如当一个客户进程向位于同一台机器上的服务器进程发出一个RPC请求时，服务器很可能即将运行。知道了这一点，在客户进程因执行RPC陷入时，系统就可以找到服务器的代码、数据、堆栈的页面，并在TLB中提前为他们建立映射，以避免TLB故障的发生。

无论是硬件还是软件，处理TLB不命中的一般方法是对页表执行索引操作找出所引用的页。用软件执行这个搜索

的一个问题是保存页表的页面本身可能就不在TLB中，这将在处理过程中再一次引发一个TLB故障。这种故障可以通过保持一个大的（比如4K）TLB条目的软件高速缓存而得到减少，这个高速缓存保存在固定位置，它的页面总是保持在TLB中，操作系统通过首先检查软件高速缓存可以大大减少TLB不命中的次数。

4.3.4逆向页表

到目前为止，在我们描述的各种类型的传统页表中每个虚页都需要一个页表项。如果地址空间是232字节，每页4096字节，则需要大于一百万的页表项，页表最低限度也有4兆字节长，在较大的系统上这也许是可行的。

然而，随着64位计算机越来越普遍，情况彻底地改变了。如果地址空间是264字节，每页4K，我们需要超过1015字节的空间存放页表，用一百万吉伽字节的内存仅仅来存放页表是绝对不可行的，无论是现在还是未来的几十年内。因此，64位的地址空间需要一种不同的解决方法。

一种解决方法就是逆向页表，在这个设计中物理存储器的每个页框对应一个页表项，而不是虚地址空间中的每个虚页对应一个页表项。在具有64位虚地址、4K大小页面、32M RAM的系统上，一个逆向页表只需要8192个表项，每个表项跟踪谁（进程，虚页）现在在该表项对应的物理页框中。

尽管逆向页表至少在虚地址空间远大于物理存储器时能节省大量的空间，但是它有一个严重的弱点：虚地址到物理地址的变换变得十分困难。当进程n引用虚页p时，硬件不能再用p作为索引查页表得到物理地址，取而代之的是在整个逆向页表中查找表项(n,p)。更为严重的是这种搜索必须在每次内存访问时进行，而不只是发生页面故障时，在每次内存访问时搜索一个8K的表不可能让你的机器能和原来一样快。

摆脱这个困境的方法是使用TLB。如果TLB能包括所有常用的页面，地址转换就能象传统页表一样快。然而，在TLB不命中时必须对逆向页表进行搜索，通过用哈希表作为逆向页表的索引可以使这个搜索是速度相当快。逆向页表现在被用在一些IBM和HP工作站上，随着64位机的普及它将变得更普遍。

处理很大虚存的其他方法可以参阅下列文献：Huck和Hays, 1993; Talluri和Hill, 1994; Talluri等, 1995。

4.4页面替换算法

在发生页面故障时，操作系统必须从内存中选择一个页删除掉以便为即将被调入的页让出空间。如果要被删除的页在内存期间已经被修改过，就必须把它写回磁盘以更新该页在磁盘上的拷贝；如果这个页没有被修改过（例如一个包含程序正文的页），那么它在磁盘上的拷贝已经是最新的了，因此不需要写回，要读入的页直接覆盖被淘汰的页就可以了。

在每次发生页面故障时尽管我们可以随机选择一个页替换，但是选择不常使用的页会使系统性能好得多。如果一个经常使用的页被淘汰掉了，极有可能它很快又要被调回来，引起不必要的额外开销。人们已经在理论上和实践上对页面替换算法进行了深入的研究，下面我们将描述几个最重要的算法。

4.4.1最优页面替换算法

最好的页面替换算法描述非常容易但却不可能实现，它是这样工作的：在页面故障发生的瞬间，有些页在内存中，其中的一个页将被紧接着的下一条指令访问（包含那条指令的哪个页），其他的页可能要到10、100、或1000条指令后才会被访问，每个页都可以用在该页面首次被引用前所要执行的指令数进行标记。

最优页面替换算法只是简单地规定，标记最大的页应该被淘汰掉。如果一个页在八百万条指令内不会被使用，另外一个页在六百万条指令内不会被使用，则淘汰掉前一个，从而把因需要取回这个页发生的页面故障推到将来，越远越好。计算机也象人一样，希望把不愉快的事情尽可能往后拖。

这个算法唯一的一个问题是它是无法实现的，在页面故障发生时，操作系统根本没有办法知道各个页面下一次被访问是什么时候（在短作业优先调度算法中我们曾遇到同样的情况—系统怎么能知道哪个作业是最短的呢？）。当然，通过首先在模拟器上运行程序，跟踪所有页面的引用情况，在第二次运行时利用第一次运行时收集的信息是能够实现最优页面替换算法的。

我们可以用这个办法对可实现算法和最好算法的性能进行对比。如果一个操作系统达到了只比最优算法差百分之一的性能，那么寻找更好的算法的努力最多只能换来百分之一的提高。

为了避免可能的混淆，读者必须清楚这个页面访问情况的记录只涉及到刚刚被测试的一个程序，因此从中导出的页面替换算法也只是针对该程序的。虽然这个方法对评价页面替换算法很有用，但它在实际系统中没有用处。下面我们将研究可以在实际系统中使用的算法。

4.4.2 最近未使用页面替换算法

为了使操作系统能够收集关于哪些页正在被使用和哪些页没有被使用的统计信息，在大部分具有虚拟存储器的计算机中每个页都有两个与其相关的状态位。R在页面被引用（读或写）时设置；M在页面被写入（即被修改）时设置。这些位包含在页表项中，如图4-11所示。这些位一定要在每次内存访问时都进行更新，因此由硬件来设置它们是必不可少的。一旦一个位被设置成1，它就一直保持1直到操作系统用软件把它复位为0。

如果硬件没有这些位，我们可以这样进行模拟：在一个进程启动时，把它所有的页面都标记成不在内存，一旦一个页被访问到就会引发一个页面故障，这时操作系统就可以设置R位（在它的内部表格中）、修改页表项使它指向正确的页、属性为只读、并重新启动指令。如果这个页随后被写入就又会引发一个页面故障，允许操作系统设置这个页的M位并把它的属性改为读写。

R位和M位可以用来构造这样一个简单的换页算法：在一个进程启动时，它的所有页的两个位都由操作系统设置成0，R位被定期地（比如在每次时钟中断时）清零，以把最近没有被访问的页和被访问了的页区别开来。

当发生页面故障时，操作系统检查所有的页面并根据它们当前的R位和M位的值把分为四类：
第0类：没有被访问，没有被修改。
第1类：没有被访问，已被修改。
第2类：已被访问，没有被修改。
第3类：被访问，被修改。
尽管第1类乍看起来似乎是不可能的，但是一个第3类的页在它的R位被时钟中断清除后就成了第1类。时钟中断不清除M位是因为这个信息在决定一个页是否需要写回磁盘时将要用到。
NRU（最近未使用）算法随机地从编号最小的非空类中挑选出一个页淘汰。这个算法隐含着的意思是淘汰一个在最近一时钟周期中（典型时间是20毫秒）没有被访问的已修改页要比淘汰一个被频繁访问的干净页好。NRU吸引人的地方是易于理解和有效地实现，并切它的性能尽管不是最好的，但常常是够用的。
4.4.3先进先出页面替换算法
另一种低开销的页面替换算法是FIFO（先进先出）算法，为了解释它是怎样工作的，让我们设想一个超市，它的货架恰好能展示k种不同的商品。有一天，某家公司介绍了一种新的方便食品—快速、冷冻干燥的可以用微波炉加工的乳酸酪，这个产品非常成功，所以我们有限的超市必须撤掉一种旧商品以便能够存放它。
一种可能的解决方法就是找到超市中销售时间最长的商品撤掉（比如某个120年以前就已经开始卖的商品），理由是已经没有人喜欢它了。这实际上相当于超市有一个按照引入时间排列的所有商品的链表，新的商品加到链表的尾部，链表头上的商品被撤掉。
同样的思想也可以用在页面替换算法中。由操作系统维持一个所有当前在内存中的页的链表，最老的页在头上，最新来的页在表尾。当发生页面故障时淘汰表头的页并把新调入的页加到表尾。当FIFO用在超市时，它可能会淘汰须蜡，但它也可能淘汰掉面粉、盐、或黄油。当它用在计算机上时也会引起同样的问题，由于这个原因，单纯形式的FIFO很少使用。
4.4.4 第二次机会页面替换算法
为了避免FIFO可能会把经常使用的页替换出去的问题，我们可以对它做一个简单的修改，对最老页面的R位进行检查。如果R位是0，那么这个页既老又没用，应该被立刻替换掉；如果是1，就清除这个位，把这个页放到页链表的尾端，修改它的装入时间让它就象刚装入的一样，然后继续搜索。
这个算法的操作叫做第二次机会，如图4-13所示。在图4-13(a)中我们看到页A到H按照抵达内存的时间的顺序保存在链表中。
图4-13　第二次机会算法的操作。(a)页面按先进先出的顺序排列。(b)在时间20发生页面故障并且A的R位已经设置时的页面链。
假设在时间20发生了一个页面故障，这时最老的页是在进程启动的时间—时间0抵达的A。如果A的R位是0，它将被淘汰出内存，或者把它写回磁盘（如果它已经被修改），或者只是简单地放弃（如果它是干净的）；在另一方面，如果R位已经被设置，则A将被放到链表的尾部并且“装入时间”被重新设置为当前时间（20），R位将被清除，对合适页面的搜索将从B开始继续进行。
第二次机会算法所做的是寻找一个从上一次对它检查以来没有引用过的页面，如果 所有的页都被引用了，它就降级为纯粹的FIFO算法。特别地，让我们设想假如图4-13(a)中所有的页的R位都被设置了，操作系统将一个接一个地把各个页移到链表的尾部并清除被移动的页的R位。最后算法又将回到页A，这时它的R位已经被清除了，因此A将被淘汰，所以这个算法总是可以结束的。
4.4.5时钟页面替换算法
尽管第二次机会算法是一个较合理的算法，但它经常要在链表中移动页面，既降低了效率，又是不必要的。一个更好的办法是把所有的页面保存在一个类似钟表面的环形链表中，如图4-14所示，有一个表针指向最老的页面。
在发生页面故障时，算法首先检查表针指向的页面，如果它的R位是0就淘汰掉这个页，并把新页插入这个位置，然后把表针前移一个位置；如果R位是1就清除R位并把表针前移一个位置，重复这个过程直到找到一个R位为0的页为止。了解了这个算法的工作方式我们就不奇怪为什么它被叫做时钟了，它和第二次机会算法的区别仅仅是实现不同。
图4-14　时钟页面替换算法。
4.4.6最久未使用页面替换算法
对最优算法的一个很好的近似是基于这样的观察的：在上面几条指令中被频繁用到的页面极有可能在下面几条指令中也将被频繁用到，反过来说，已经很久没有使用的页面很有可能在未来较长的一段时间内都不会被用到。这个思想向我们提示了一个可以实现的算法：在页面故障发生时，淘汰掉没有使用的时间为最长的页。这个策略叫做LRU（最久未使用）换页。
虽然LRU在理论上是可以实现的，但代价很高。为了完全实现LRU，需要维持一个在内存中的所有页的链表，最近使用的页在表头，最久未使用的页在表尾。困难的是这个链表必须在每次内存访问时都进行更新，在链表中找到被访问的页，将它移动到表头是一个非常费时的操作，即便是用硬件实现也是如此（假设有这样的硬件）。然而，还是有其他一些用特殊硬件实现LRU的方法，我们先考虑一个最简单的。这个方法要求硬件有一个64位计

数器C，它在每条指令执行完后自动加1，每个页表项必须有一个足够容纳这个计数器值的域。在每次内存访问后，当前的C值被存到被访问页的页表项中。当发生页面故障时，操作系统检查页表中所有的计数器的值以找出最小的，这个页就是最久未使用的页。

现在让我们看一看第二个硬件LRU算法，在一个有n个页框的机器中，LRU硬件可以维持一个n×n位的矩阵，开始时所有的位都是0。在页k被访问到时，硬件首先把k行的位都设置成1，再把k列的位都设置成0。在任何时刻，二进制值最小的行就是最久未使用的，第二小的行是下一个最久未使用的，依此类推。这个算法的工作情况我们可以用图4-15所示的实例说明，该实例中有4个页框，他们的引用次序为

0 1 2 3 2 1 0 3 2 3

在页0被访问过后的情况示于图4-15(a)，依此类推。

图4-15　使用矩阵的LRU算法。

4.4.7用软件模拟LRU

虽然前面两种LRU算法在理论上都是可以实现的，但是如果有的话也只有非常少的计算机有这种硬件，因此对为没有这种硬件的机器开发操作系统的设计者而言，他们是没有价值的。我们需要的是一个能用软件实现的解决方案。一种可能的方案就是称作NFU（不常使用）的算法 ，它使每个页都和一个软件计数器相联系，计数器的初值是0。在每次时钟中断时由操作系统对在内存中的页进行扫描，把每个页的R位（它的值是0或1）加到它的计数器上。实际上这个计数器是试图跟踪各个页被访问的频繁程度，当发生页面故障时，计数器值最小的页被选中替换掉。

NFU的主要问题是它从来不忘记任何事情。比如在一个多次扫描编译器中，在第一遍扫描中被频繁用到的页，在程序进入第二遍扫描时计数器值可能仍然很高。实际上，如果第一次扫描的执行时间恰好是各次扫描中最长的，含有以后各次扫描代码的页的计数器可能总是比含有第一次扫描代码的页小，结果是操作系统将删除掉有用的页而不是不再使用的页。

幸运的是只用对NFU做一个小小的修改就能让它相当好地模拟LRU。修改分两部分：第一是计数器在R位被加进来之前右移一位；第二是R位加到计数器的最左端而不是最右端。

修改以后的算法称为老化算法，图4-16解释了它是怎样工作的。假设在第一个时钟周期后页0到页5的R位值分别是1、0、1、0、1、1，换句话说，在时钟周期0到在时钟周期1期间，页0、2、4、5被访问到，他们的R位被设置为1，而其他页的R位仍然是0，对应的六个计数器在经过移位并把R位插入其左端后的值如图4-16(a)所示。图中后面的四列是在下四个时钟周期后的六个计数器的值。

图4-16　用软件模拟LRU的老化算法。图中所示是6个页面在5个时钟周期的情况，5个时钟周期分别由(a)-(e)表示。

在发生页面故障时，计数器值最小的页将被淘汰掉。如果一个页在前面四个周期中都没有被访问过，它的计数器最前面应该有四个连续的0，因此它的值肯定要比在前面三个周期中都没有访问过的页的计数器小。

这个算法在两个方面和LRU不同。让我们看图4-16(e)中的页3和5，他们都连续两个周期没有被访问了，而在两个周期之前的周期中他们都被访问过。根据LRU，如果有一个页必须被淘汰掉，我们应该在这两个页中选一个。然而现在的问题是我们不知道在时钟周期1到时钟周期2期间这两个页中的哪一个后被访问到，在每个时钟周期中只记录一位使我们无法区分一个周期内较早和较晚的访问，我们所能作的只是淘汰掉页3，因为页5在再往前的周期中也被访问过而页3没有。

LRU和老化算法的第二个区别是老化算法的计数器只有有限位,在这个例子中是8位。如果两个页的计数器都是0，我们只能在两个页中随机选一个。在实际中，有可能一个页上次被访问是9个周期以前，另一个页是1000个周期以前，而我们却无法看到这些。在实践中，如果时钟周期是20毫秒，8位一般是够用的。假如一个页已经有160毫秒没有被访问，它可能并不是那么重要。

4.5分页系统中的设计问题

在前面几节中我们解释了分页系统是怎样工作的，并给出了几个基本的页面替换算法，但仅仅知道基本的机制是远远不够的。为了设计一个系统，你必须知道更多的东西以使它工作得好。这两者之间的差别就象知道了怎样移动象棋的各种棋子和成为一个好棋手之间的差别。在下面的章节中我们将讨论为了使分页系统达到较好的性能，操作系统设计者必须仔细考虑的一些其他问题。

4.5.1 工作集模型

在单纯的分页系统中，当进程启动时它的页面没有一个在内存中，在CPU试图取第一条指令时就会引发一个页面故障，使操作系统装入含有第一条指令的页，其他的由访问全局数据和堆栈引起的页面故障通常会紧接着发生。一段时间以后，进程需要的大部分页已经在内存了，进程开始在页面故障较少的情况小运行。这个策略称为请调（demand paging），因为页是在需要时调入，不是预先装入。

尽管我们可以很容易地编写一个测试程序，系统地读一个大的地址空间中的所有页，产生大量的页面故障，使得内存无法把这些页面都容纳下，但幸运的是大部分进程不是这样工作的，他们都表现出一种访问的局部性（locality of reference），意思是在进程运行的任何阶段，它都只访问它的页面中较小的一部分。例如在一个多次扫描编译器中，各次扫描时只引用所有页中的一小部分，并且是不同的部分。

一个进程当前使用的页的集合叫做它的工作集（working set）（Denning, 1968a; Denning, 1980）。如果整

个工作集都在内存中，在进入下一个运行阶段（比如编译器的下一次扫描）之前进程的运行不会引起很多页面故障。如果内存太小无法容纳整个工作集，进程运行将引起大量的页面故障并且速度十分缓慢，因为执行一条指令通常只需要几个纳秒，而从磁盘上读入一个页面一般需要几十个毫秒。一个20毫秒执行一到两条指令的程序不知道需要多长时间才能运行完。一个每隔几条指令就发生一次页面故障的程序被称为是在颠簸（Denning, 1968b）。

在分时系统中，进程经常会被转移到磁盘上（即它所有的页面都被从内存中删除），从而让其他的进程有机会占有CPU，问题是当一个进程被再次调回来以后应该怎样处理？从技术的角度上来看，什么都不用做，这个进程将一直产生页面故障直到它的工作集全部被装入。但是，在每次装入一个进程时都产生20、50甚至100个页面故障是非常慢的，并且因为CPU处理一个页面故障需要几个毫秒的时间，这也浪费了相当多的CPU时间。

因此，许多分页系统都试图跟踪进程的工作集，并保证在进程运行以前它的工作集就已经在内存中了。这样一种方法称为工作集模型（Denning, 1970），它旨在大大减少页面故障。在进程运行之前预先装入页面也叫做预调。

为了实现工作集模型，操作系统必须对哪些页面属于工作集一直保持跟踪。监视这个信息的一个办法就是使用上面讨论过的老化算法，所有计数器的高n位含有1的页都被认为是工作集的成员；如果一个页连续n个时钟周期没有被访问到，它将被从工作集中删除。对于不同的系统，参数n必须通过试验确定，但系统的性能通常对精确的数值不是特别地敏感。

有关工作集的信息可以用来提高时钟算法的性能。本来如果表针指向的页的R位是0，这个页就将被淘汰掉，改进之处是检查这个页是不是当前进程工作集的一部分，如果是就跳过这个页。这个算法叫做wsclock。

4.5.2 局部与全局分配策略

前面的几节中，我们讨论了好几个在发生页面故障时选择被淘汰页的算法。与这个选择相联系的一个主要问题（到目前为止我们一直在小心地回避这个问题）是怎样在相互竞争的可运行进程间分配内存。

让我们看一看图4-17(a)，在图中三个进程A、B、C构成了可运行进程的集合。假如A发生了页面故障，页面替换算法在寻找最久未使用的页时是只考虑分配给A的六个页呢，还是考虑所有在内存中的页？如果只考虑分配给A的页，年龄值最小的页是A5，于是我们将得到图4-17(b)所示的情况。

图4-17　局部与全局替换算法。(a)原先配置。(b)局部页面替换。(c)全局页面替换。

在另一方面，如果我们要淘汰年龄值最小的页而不管它是谁的，那么页B3将被选中，我们将得到图4-17(c)所示的情况。图4-17(b)的算法被称为局部页面替换算法，而图4-17(c) 被称为全局算法。局部算法实际上对应于为每个进程分配固定的内存片段；全局算法在可运行进程之间动态地分配页框，因此分配给各个进程的页框数是随时间变化的。

在通常情况下，尤其是当工作集的大小会在进程的运行期间发生变化时，全局算法工作得比局部算法好。如果使用局部算法，那么即使还有大量的空闲页框存在，工作集的增长会导致颠簸；如果工作集收缩了，局部算法又会浪费内存。在使用全局算法时，系统必须不断地确定应该给各个进程分配多少页框。一个办法是监视由年龄位指出的工作集的大小，但这个办法并不足以阻止颠簸。工作集的大小可以在几微秒内改变，而年龄位只是一个跨越所经过时钟周期数的粗略的量度。

另一种途径是使用一个为进程分配页框的算法。一种办法是定期地确定运行进程的数目并为他们分配相等的份额。例如在有475个空闲（即非操作系统用）页框和10个进程时，每个进程将获得47个页框，剩下的5个放入一个公用池中在发生页面故障时使用。

这个算法貌似公平，但是给一个10K的进程和一个300K的进程分配同样大小的存储器是完全不合理的。取而代之我们可以按照进程大小的比例为他们分配页面，这样300K的进程将得到10K进程的30倍的份额。一个可能比较明智的做法是为每个进程都规定最小的页框数，这样可以使不管多么小的进程都可以运行，例如在某些机器上，一条指令的指令自己、源操作数和目的操作数可能都会跨越页边界，这样的一条指令需要多达六个页面，如果只分配了5个页面，包含这样的指令的程序根本就无法运行。

无论是平均分配还是按比例分配都不是直接处理颠簸问题的，一个更直接的控制颠簸的方法是使用页面故障率(Page Fault Frequency)，或者叫做PFF分配算法。我们曾经讨论过，包括LRU在内的一大类页面替换算法的故障率随着分配的页框数的增加而减少，如图4-18所示。

图4-18　页面故障率是分配的页框数的函数。

虚线A对应的是一个过高的故障率，发生故障的进程将被分配更多的页框以减少故障率；虚线B对应的是一个过低的故障率，我们可以得出结论分配给这个进程的内存太多了，可以收回一些页框。可以看出，PFF试图把页面故障率保持在一个可以接受的范围内。

如果发现内存中进程太多以至不可能使所有进程的故障率都低于A，那么我们就必须从内存中移去某些进程，把他们的页框分给余下的进程或放进一个空闲页框池中供后面发生页面故障时使用。从内存中移去一个进程的决定实际上是一种负载控制，它表明即使在分页系统中交换仍然是需要的，不同的只是现在交换是为了降低潜在的对内存的需求，而不是为了立刻使用收回的内存块。

4.5.3页面大小

页面大小经常是一个操作系统可以选择的参数。例如，即使硬件设计只支持512字节的页，操作系统可以很容易

地通过总是分配两个连续的页把0和1、2和3、4和5等当作1K的页。
选择最优的页面大小需要在几个互相冲突的因素之间进行折衷。首先，随便找一个正文、数据或堆栈段，它不会恰好装满整数个页，在平均的情况下，最后一页中有一半是空的。多余的空间就被浪费掉了，这种浪费叫做内零头（internal fragmentation）。在内存中有n个段、页长为p字节时，np/2字节将被内零头浪费。这个推理要求小的页面。
如果考虑一个程序，它分成8个阶段顺序执行，每阶段需要4K内存，那么另一个要求小页面的原因就很清楚了。如果页面大小是32K，那就必须始终给该进程分配32K内存；如果页面大小是16K，它就只需要16K的；如果页面大小是4K或更小，在任何时刻它只需要4K内存。总的来说，大的页面要比小的页面使更多的没有用的程序保留在内存中。
在另一方面，小的页面意味着程序需要更多的页面，这又意味着更大的页表。一个32K的程序只需要4个8K的页，却需要64个512字节的页。内存与磁盘之间的传输一般是一次一个页，因为大部分时间都花在了寻道和旋转延迟上，传输一个小的页面所用的时间和传输一个大的页面基本上是相同的。装入64个512字节的页可能需要64×15毫秒，而装入4个8K的页可能只需要4×25毫秒。
在一些机器上，每次CPU从一个进程切换到另一个进程时都必须把页表装入硬件寄存器中。在这些机器上页面越小意味着装入页寄存器的时间越长，而且页表占用的空间随着页的减小而增大。
这最后一点可以数学地进行分析，假设平均进程大小是s个字节，页面大小是p个字节，每个页表项需要e个字节，那么进程需要的页数大约是s/p，占用了se/p个字节的页表空间，由于内零头在最后一页浪费的内存是p/2。因此，由页表和内零头损失造成的全部开销是：
开销 = se/p + p/2
在页面比较小时第一项（页表大小）大；在页面比较大时第二项（内零头）大。最优值一定在中间某个地方，通过对p求导并令其等于零，我们得到方程：
-se/p2 + 1/2 = 0
从这个方程我们可以得出最优页面大小的公式（只考虑碎片浪费和页表所需的内存），结果是：
对于s = 128K和每个页表项e = 8个字节，最优页面大小是1448字节。考虑到其他因素（比如磁盘速度），在实际中将使用1K或2K。大部分商品计算机使用的页面尺寸在512字节到64K之间。
4.5.4虚拟存储器界面
到目前为止，我们一直假设虚拟存储器对进程和程序员是透明的，他们所能看到的全部是在一个带有较小的物理存储器的计算机上的一个大的虚地址空间。在许多系统上这是对的，但在一些先进的系统上，程序员对内存映射有一定程度的控制并且能用非传统的方式使用它。在这一节中，我们将简略地介绍其中的少数几个。
给予程序员对内存映射某些控制的一个原因是为了允许两个或多个进程共享同样的存储器。如果进程可以命名它的内存区，一个进程就有可能把自己的一块内存区的名字告诉另一个进程，使另一个进程也可以把它映射进来。两个（或多个）进程共享同样的页面使高带宽的共享成为可能—一个进程写入共享内存区，另一个进程从中读出。
页面共享还可以用来实现高性能的消息传递系统。一般情况下在要传递消息时，数据是被从一个地址空间复制到另一个地址空间，代价相当高。如果进程可以控制他们的页面映射，消息传递可以这样实现：发送消息的进程把包含消息的页面从它的页面映射中删除掉，而由接受进程把映射进来。这里只需要复制页面的名字，而不是复制全部数据。
还有一种先进的内存管理技术是分布共享存储器（distributed shared memory）（Feeley等，1995；Li和Hudak，1989；Zekauskas等，1994）。它的思想是允许在网络上的多个进程共享一套页面，这些页面可以，但不是必须，构成一个共享的单一线形地址空间。当一个进程引用一个当前没有被映射的页时就会引发一个页面故障。页面故障处理器，它可以在内核中也可以在用户空间，随即找到保存这个页面的机器，并向这个机器发消息让它删除对这个页的映射并把它从网络上发送过来。在页面到达后，它将被映射进来，故障进程将被重新启动。
4.6 分段
到目前为止我们讨论的虚拟存储器都是一维的，虚地址从0到最大地址，一个地址跟着另一个地址。对许多问题来说，有两个或多个独立的地址空间可能比只有一个要好得多。比如一个编译器在编译过程中会建立许多表格，其中可能包括：
1. 保存的供打印清单用的源码正文。
2. 符号表，包含变量的名字和属性。
3. 包含整形和浮点常数的表。
4. 语法树，包含程序的语法分析的结果。
5. 编译器内部过程调用使用的堆栈。
前四个表随着编译的进行不断地增长，最后一个在 编译过程中以一种不可预计的方式增长和缩小。在一维存储器中，这五个表只能被分配到虚地址空间中邻接的块中，如图4-19所示。
图4-19 在一维地址空间中，当有多个动态增长的表时，一个表可能会撞上另外一个。
让我们考虑一下如果一个程序有异常大数量的变量时会发生什么。地址空间中分给符号表的块可能会被装满，

但这时其他表中还有大量的空间。编译器当然可以简单地打印出一条信息说由于变量太多编译不能继续进行，但在其他表中还有空间时这样作似乎并不恰当。
另外一种可能的方法就是罗宾汉的劫富济贫，从空闲空间过多的表中拿出一些空间给空间很少的表。这种处理是可以做得到的，但是它和自己管理覆盖一样—在最好的情况下也是一件另人讨厌的事，在最坏的情况下则是一大堆沉闷并且毫无回报的工作。
我们真正需要的是一个能够把程序员从管理表的收缩和扩张的工作中解放出来的办法，就象虚拟存储器使程序员不用再为怎样把程序划分成覆盖块而担心一样。
一个直观并且非常通用的方法是在机器上提供多个互相独立的称为段(Segment)的地址空间，每个段由一个从0到最大的线性地址序列构成。各个段的长度可以是0到允许的最大值之间的任何一个值，不同的段的长度可以不同，而且通常也都不同。段的长度在运行期间可以改变，堆栈段的长度在数据被压入时会增长，在数据被弹出时又会减小。
因为每个段都构成了一个独立的地址空间，他们可以独立地增长或减小而不会影响其他的段。如果一个在某个段中的堆栈需要更多的空间来增长，因为在它的地址空间中没有任何其他东西阻挡，它就可以立刻得到它。段当然有可能被装满，但因为段通常很大，所以这种情况发生的可能性很小。在这种分段的或二维的存储器中指明一个地址必须提供一个段号和一个段内地址。图4-20图示了用于前面讨论过的编译表格的分段内存。
图4-20 分段的内存允许各个表独立地增长或缩减。
我们要强调的是段是一个逻辑实体，程序员知道这一点并把它用作一个逻辑实体。一个段可能容纳一个过程、或者一个数组、或者一个堆栈、或者一些数值变量，但它一般不会同时包含多种不同的内容。
分段的存储器除了能简化对长度经常变动的数据结构的处理以外还有其他一些优点。如果每个过程都位于一个独立的段中并且起始地址是0，把单独编译好的过程链接起来的操作就能大大简化。当组成一个程序的所有过程都被编译和链接好以后，一个对在段n的过程的过程调用将使用由两个部分组成的地址 (n, 0) 转到地址字0（入口点）。
如果随后位于段n的过程被修改并重新进行了编译，即使新版本的程序比老的要大，也不需要对其他的过程进行修改（因为没有修改他们的起始地址）。在一维地址中，过程被一个挨一个紧紧地放在一起，中间没有空隙，结果是修改一个过程的大小会影响其他无关过程的起始地址，而这又需要修改所有引用了被移动过程的过程，以使他们的引用指向这些过程的新地址。在一个有数百个过程的程序中这个操作的开销可能是相当大的。
分段也有助于在几个进程之间共享过程和数据，这方面一个常见的例子就是共享库。运行先进窗口系统的现代工作站常常要把非常大的图形库编译进几乎每个程序。在分段系统中，可以把图形库放到一个单独的段中由各个进程共享，从而不再需要在每个进程的地址空间中都有一份。虽然在纯的分页系统中也可以有共享库，但是它要复杂得多，并且这些系统实际上是通过模拟分段来实现的。
因为段是一个为程序员所知道的逻辑实体，比如一个数组、一个堆栈，不同的段可以有不同种类的保护。一个过程段可以被指明为只允许执行，从而禁止对它的读出和写入；一个浮点数组可以被指明为允许读写但不允许执行，任何试图向这个段内的跳转都将被俘获。这样的保护有助于抓住程序的错误。
你应该尽力理解为什么保护在分段存储器中有意义而在一维的分页存储器中则没有。在分段存储器中用户知道每个段中包含了什么，例如一般来说，一个段中不会既包含一个过程又包含一个堆栈，而是只包含其中的一个或另外一个。正是因为每个段只包含一种类型的对象，这个段就可以有针对这种特定类型的合适的保护。图4-21对分段和分页进行了比较。
考虑的因素
分　　页
分　　段
需要程序员知道正在使用这种技术吗？
不需要
需要
有多少个线性地址空间？
1
多个
全部地址空间可以超过物理存储器大小吗？
可以
可以
数据和过程可以被区别开并被分别保护吗？
不能
能
能够很容易地容纳长度经常变化的表吗？
不能
能

有助于在用户间共享过程吗？
不
是
为什么发明这个技术？
为了得到大的线性地址空间而又不必去买更多的物理存储器
为了允许程序和数据能被划分为逻辑独立的地址空间以帮助实现共享和保护
图4-21 分页与分段的比较
页的内容在某种程度上是随机的，程序员甚至觉察不到分页的事实。尽管在页表的每个表项中放入几位说明对应页的访问权限是可能的，然而为了利用这一点程序员必须对他的地址空间中页的界限保持跟踪，当初正是为了避免这一类的管理工作，人们才发明分页的。在分段系统中，因为用户有所有的段都一直在内存中的幻觉——也就是说他可以就象所有这些段都在内存中一样去访问——他可以分别地保护各个段，不需要关心对他们覆盖的管理工作。
4.6.1 纯分段系统的实现
分段和分页的实现从根本上是不相同的：页是定长的而段不是。图4-22(a)所示的是开始时包含了5个段的物理存储器，现在让我们考虑当段1被淘汰、比它小的段7放进它的位置后会发生什么？这时的内存配置如图4-22(b)所示，在段7与段3之间是一个未用区域，即一个空洞。随后段4被段5代替，如图4－22(c)；段3被段6代替，如图4－22(d)。在系统运行一段时间后内存被划分为许多块，一些包含段，一些包含空洞，这种现象被称为跳棋盘现象或外零头（external fragmentation）。它在空洞上浪费了内存，而这可以通过紧缩来解决，如图4-22(e)所示。
图4-22 (a)-(d)棋盘形碎片的形成。(e)通过紧缩消除棋盘形碎片。
4.6.2 分段和分页结合：MULTICS
如果一个段比较大，把它整个保存在内存中可能是不方便甚至是不可能的，这导致了对它进行分页的想法，这样只有哪些真正需要的页才需要被调进来。有几个著名的系统支持对段进行分页，这一章我们将介绍第一个：MULTICS，在下一章我们将介绍一个更新的：Intel的Pentium。
MULTICS运行在Honeywell 6000计算机和它的一些派生机上，它为每个程序提供了最多218个段（超过250,000个），每个段最长65536个字长（36位）的虚地址空间。为了实现它，MULTICS的设计者决定把每个段看作一个虚拟存储器并对它进行分页，以结合分页的优点（统一的页面大小和在只使用段的一部分时不用把它全部调入内存）和分段的优点（易于编程、模块化、保护和共享）。
每个MULTICS程序都有一个段表，每个段对应一个描述符。因为段表可能会有大于25万个表项，段表自己也是一个段并被分页。一个段描述符包含了一个段是否在内存的标志，只要一个段的任何一部分在内存这个段就被认为是在内存中，并且它的页表也将在内存中。如果一个段在内存中，它的描述符将包含一个18位的指向它的页表的指针〖参看图4-23(a)〗。因为物理地址是24位并且页是按照64字节的边界对齐的（这隐含着页地址的低6位是000000），所以在描述符中只需要18位来存储页表地址。描述符还包含了段大小、保护位和一些其他的条目。图4-23(b)是一个MULTICS段描述符。段在二级存储器中的地址不在段描述符中，它在系统故障处理程序使用的另一个表中。
图4-23 MULTICS的虚拟存储器。(a)描述符段指向页表。(b)一个段描述符，其中的数字是各个域的长度
每个段都是一个普通的虚地址空间，用与本章前面讨论过的不分段的页式存储器相同的方式进行分页。一般的页面大小是1024字节（尽管有一些MULTICS自己使用的段不分页或以64字节为单位进行分页以节省存储器）。
一个MULTICS的地址有两部分构成：段和段内地址。段内地址又进一步分为页号和页内的内存字，如图4-24所示。在进行内存访问时，执行下面的算法。
1. 根据段号找到段描述符。
2. 检查该段的页表是否在内存中，如果在就找到它的位置，如果不在就发出一个段故障。如果访问违反了段的保护要求就发出一个故障（陷入）。
3. 检查所请求虚页的页表项，如果页面不再内存中则发出一个页面故障，如果在内存中就从页表项中取出这个页在主存中的起始地址。
4. 把偏移地址加到页的起始地址上，得到要访问的字在主存的地址。
5. 最后进行读或写操作。
图4-24 34位的MULTICS虚地址。
这个过程如图4-25所示，为了简单起见，我们忽略了描述符段自己也要分页的事实。实际的过程是通过一个寄存器（描述符基址寄存器）找到描述符段的页表，这个页表指向描述符段的页面。一旦找到了所需段的描述符，下面的寻址过程就如图4-25所示。
正如你现在肯定已经猜想到的，如果对于每条指令，上面所述的算法都由操作系统来运行，那么程序就不可能运行得很快。实际上，MULTICS硬件包含了16字高速TLB，对给定的关键字它能并行地搜索所有的表项，如图4-26所示。当一个地址被送到计算机时，寻址硬件首先检查虚地址是不是在TLB中，如果在，就直接从TLB中取得页框号生成要访问的字的实际地址，而不必到描述符段或页表中去查找。

图4－25 两部分组成的MULTICS地址到主存地址的转换。
图4－26 简化的MULTICS TLB。两种页面长度的存在使实际的TLB更加复杂。
TLB中保存着16个最近访问的页。工作集小于TLB容量的程序将随着整个工作集被装入TLB中而逐渐达到稳定，开始高效地运行。如果页不在TLB中，描述符和页表才会被引用以找出页框地址，并更新TLB使它包含这个页，最久未使用的页被淘汰出TLB。年龄域对哪个表项是最久未访问过的保持跟踪。之所以使用TLB是为了并行地和所有表项的段号和页号进行比较。
4.6.3 分段和分页结合：Intel 的 Pentium
Pentium（和Pentium Pro）在许多方面都与MULTICS类似，其中包括既有分段机制又有分页机制。MULTICS有256K个独立的段，每个段最长可以有64K个36位字，Pentium有16K个独立的段，每个段最多可以容纳十亿个32位字。这里虽然段的数目较少，但是大的段尺寸要重要得多，原因是几乎没有程序需要1000个以上的段，但是有很多程序需要一个段能容纳数兆字节的数据。
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，但是同一计算机上所有的程序共享一个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。
为了访问一个段，Pentium程序必须把这个段的选择符(Selector)装入机器的六个段寄存器中的某一个中。在运行过程中，CS寄存器保存代码段的选择符，DS寄存器保存数据段的选择符，其他的段寄存器不太重要。每个选择符是一个16位数，如图4-27所示.
图4-27 一个奔腾的选择符。
选择符中的一位指出这个段是局部的还是全局的（即，它是在LDT中还是在GDT中），其他的13位是LDT或GDT的入口号，因此这些表长度被限制在最多容纳8K个段描述符，还有两位和保护有关，我们将在后面讨论。描述符0是禁止使用的，它可以被安全地装入一个段寄存器中用来表示这个段寄存器目前不可用，如果使用会引起一次陷入。
在选择符被装入段寄存器时，对应的描述符被从LDT或GDT中取出装入微程序寄存器中，以便于快速地访问。一个描述符由8个字节构成，包括段的基地址、长度和其他信息，如图4-28所示。
图4-28 奔腾的代码段描述符，数据段有轻微的差别。
选择符的格式经过了聪明的挑选，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3位被清0；最后LDT或GDT表的地址被加到它上面，得出一个直接指向描述符的指针。例如选择符72指向GDT的第九个入口，它位于GDT＋72。
现在让我们跟踪一个（选择符，偏移）对被转换为物理地址的过程。只要微程序知道哪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）或已被换出，则会发生一次陷入。
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。从逻辑上来说，在描述符中应该简单地有一个32位的域给出段的长度，但实际上只有20位可以使用，因此Pentium采用了一种不同的方案。如果Gbit（粒度）域是0，则Limit域是精确的段的长度，最大1MB；如果是1，Limit域以页而不是字节为单位给出段的长度。Pentium页的长度是固定的4K字节，因此20字节足够最大232字节的段使用。
假设段在内存中并且偏移也在范围内,Pentium接着把描述符中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4-29所示。为了和只有24位基地址的286兼容，Pentium基地址被分为三片分布在描述符的各个位置。实际上，基址域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4-29 （选择符、偏移）到线形地址的转换。
如果分页被禁止（通过一个全局控制寄存器中的一位），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，这可能是因为验证所有的段都互不重叠太麻烦太费时间的缘故。
在另一方面，如果分页被允许，线性地址将被解释成一个虚拟地址并通过页表映射成为物理地址，很象我们前面讲过的例子。这里唯一真正复杂的是在32位虚地址和4K页的情况下，一个段可能包含多达一百万个页，因此Pentium使用了一种两级映射以在段较小时减小页表尺寸。
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。目录中的每个表项都指向一个也包含1024个32位表项的页表，页表项指向页框，这个方案如图4-30所示。
图4-30 线形地址到物理地址的映射。
在图4-30(a)中我们看到线性地址被分为三个域：Dir、Page、和Off。Dir域被作为索引在页目录中找到指向正确的页表的指针；随后Page域被用作索引在页表中找到页框的物理地址；最后，Off被加到页框的地址上得到需要的字节或字的物理地址。
每个页表项是32位，其中20位是页框号，其余的位包含了由硬件设置供操作系统使用的访问位和修改位、保护位、和一些其他有用的位。
每个页表有描述1024个4K页框的表项，因此一个页表可以处理4M的内存。一个小于4M的段将有一个只有一个表项的页目录，这个表项指向一个也是唯一的页表。通过这种方法，短的段的开销只是两个页，而不是一级页表

时的一百万个页表项。
为了避免重复的内存访问，Pentium和MULTICS一样，也有一个TLB把最近使用过的页目录号-页号组合映射为页框的物理地址，只有在当前的组合不在TLB中时图4-30所示的机制才被真正执行并更新TLB。
略微思考一下我们就会发现在使用分页时，让描述符的基址域不为0确实是毫无意义的。基址的全部作用就是使小的偏移量使用页目录中间的表项而不是开头，之所以包含基址域是为了允许纯的分页和与总是禁止分页的286兼容（即，286只有纯的分段，没有分页）。
还有一点值得注意的是如果一个应用程序不需要分段，只需要单个分页的32位地址空间，这也是可以的。这时所有的段寄存器都被设置为同一个选择符，它的描述符基址是0，长度是最大，这样只有一个地址空间被使用，指令偏移将成为线性地址，实际上就是一般的分页。
不管怎么说，我们不得不称赞Pentium的设计者，向他们提出的是互相冲突的目标：实现纯的分页、纯的分段、和段页式管理，还要与286兼容，而他们高效地实现了所有的目标，最终的设计非常简洁干净。
尽管我们已经简单地讨论了Pentium虚拟存储器的全部体系机构，关于保护我们还是值得再说几句的，因为它和虚拟存储器联系很紧密。Pentium的保护机构和虚拟存储器一样非常象MULTICS模型，它支持四个保护级，0级权限最高，3级最低，如图4－31所示。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上，系统中的每个段也有一个级别。
图4-31 奔腾中的保护。
只要程序只使用与它同级的段，一切都会很正常。对更高级别数据的存取是允许的，而对更低级别的数据的存取是非法的并会引起陷入。调用不同级别（更高或更低）的过程是允许的，但是要通过一种被严格控制着的方法。为执行越级调用，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，只有正式指定的入口点可以使用。保护级和调用门的概念来自MULTICS，在那里他们被称为保护环（protection rings）。这个机构的一种典型的应用如图4-31所示。在0级是操作系统内核，处理I/O、存储器管理、和其他关键的操作；在1级是系统调用处理程序，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；在2级是库过程，它可能是由很多正在运行的过程共享的，用户程序可以调用这些过程，读取他们的数据，但是不能修改他们；最后，用户程序运行在级别3上，受到的保护最少。
陷入和中断使用了一种和调用门类似的机构，他们引用的也是描述符而不是绝对地址，这些描述符指向将被执行的特定的过程。图4-28中的Type域用于区别代码段、数据段、和各种门。
4.7 MINIX内存管理概览
MINIX的内存管理是非常简单的：既不分页也不交换。存储管理器保存着一张按照内存地址排列的空洞列表，当由于执行系统调用FORK或EXEC需要内存时，系统将用首次适配算法对空洞列表进行搜索找出一个足够大的空洞。一旦一个程序被装入内存，它将一直保持在原来的位置直到运行结束，它永远不会被换出或移动到内存的其他位置去，为它分配的空间也不会增长或缩小。
对这个方案应该进行一些解释，它是考虑了三个因素后的结果：(1) MINIX是用于个人计算机的，而不是大型的分时系统；(2) 希望MINIX在所有的IBM PC上运行；(3) 希望系统的实现很直观以利于在其他小型计算机上实现。第一个因素意味着一般来说，运行进程的数目将比较少，在典型的情况下内存足够容纳所有的进程，并且还会有一些空余。在这里交换是不需要的，它只会增加系统的复杂性，不要交换使得代码很简单。
让MINIX在所有IBM PC兼容机上运行的愿望也对内存管理的设计有很大的影响。在这个家族中最简单的系统使用8088处理器，它的内存管理结构十分原始，不支持任何形式的虚拟存储器，并且甚至不检查堆栈溢出。这个缺点对进程在内存中布局的方案有很大的影响。这些限制在使用80386，80486或Pentium的处理器时都不再存在，但是使用这些特性将使MINIX和许多目前仍然可以使用或在使用的低端机器不兼容。
对兼容性的考虑也要求内存管理方案尽量简单。如果MINIX使用了分页和分段，要把它移植到没有这些特性的机器上将是十分困难的，如果不是不可能的话。通过对硬件所能提供的支持作最少的假设可以使MINIX能移植的机器数量大大增加。
MINIX另一个不同寻常的方面是它的内存管理实现的方法。它不是内核的一部分，而是由一个运行在用户空间、使用标准的消息机构和内核通讯的管理进程完成的。处于服务器级别的内存管理器的位置如图2-26所示。
把内存管理器移到内核外面是把决策和机构分开的一个例子，哪个进程应该被放在内存中哪个位置的决定（决策）是由内存管理器作出的，而具体的为进程设置内存映像（机构）的操作是由在内核中的系统任务完成的。这个划分使得修改内存管理策略（算法等）比较容易实现，不需要修改操作系统底层。
大部分内存管理器的代码是用来处理和内存管理有关的系统调用的，主要是FORK和EXEC，而不是仅仅操作进程和空洞的表。在下一节中我们将看到内存的布局，在再下一节中我们将鸟瞰内存管理器是怎样处理内存管理系统调用的。
4.7.1 内存布局
简单的MINIX进程使用结合的I和D空间，这时进程所有的部分（正文、数据、和堆栈）共用一个内存块，它是作为一个块申请和释放的。进程也可以被编译为使用分开的I和D空间。为了清楚起见，我们先讨论简单模型的内存分配。使用独立的I和D空间的进程能更有效地利用内存，但利用这个特征的优点会使事情变得复杂。我们将

在勾画完简单的之后讨论复杂的。
在MINIX中有两种情况需要分配内存。第一种是在一个进程执行fork时，为子进程分配所需要的空间；第二种是在一个进程通过EXEC系统调用修改它的内存映象时，老的映象被作为空洞送到空闲表，需要为新的映象分配内存。新的映象可能会在内存中不同于被释放的内存的地方，它的位置取决于在什么地方能找到合适的空洞。在进程因为执行exit或被信号杀死而结束时内存也将被释放。
图4-32示出了这两种分配内存的方法。在图4-32(a)中我们看到在内存中有两个进程，A和B。如果A执行了fork，我们将得到图4-32(b)所示的情况，子进程是A的一个精确拷贝。如果子进程执行了文件C，内存看起来将如图4-32(c)所示，子进程的映象被C取代。
图4-32　内存分配。(a)原来的情况。(b)A执行FORK以后。(c)A的子进程执行了EXEC以后。阴影区域是未用内存。
这个进程是一个普通的I和D结合的进程。
需要注意的一点是用于子进程的老的内存是在用于C的新的内存分配之前被释放的，因此C可以使用子进程的内存。这样，一系列的FORK和EXEC对（比如shell设置一个管道线）将会使所有的进程都是邻接的，中间没有空洞。如果我们在老的内存被释放之前就分配新内存，就有可能产生空洞。
由FORK或EXEC系统调用引起而分配内存时，一定数量的内存将给予新进程。在前一种情况下，分配的数量和父进程的相同；在后一种情况下，内存管理器将分配被执行文件的头部所指明的数量。一旦分配完毕，决不会再给进程分配一丝一毫的内存。
前面所讨论的适用于被编译为具有结合的I和D空间的程序。具有独立的I和D空间的程序可以利用增强方式内存管理的一个叫做共享正文的优点。当这样的一个进程FORK时，只需要分配为新进程做一个堆栈段和数据段拷贝所需数量的内存。父进程和子进程将共享已经由父进程使用的执行代码。当一个这样的进程执行EXEC时，系统将查找进程表看是否有另外一个进程已经在使用需要的执行代码，如果找到了就只为数据和堆栈分配新内存，已经在内存的正文将被共享。共享正文使得进程的结束复杂化了。在一个进程结束时它总是要释放它的数据和堆栈占用的内存，但是只有在搜索了进程表并发现没有其他进程在使用正文段后，才释放正文段所占用的内存。因此如果一个进程在启动时装入了自己的正文，而在结束时它的正文正在被一个或多个其他进程共享，那么这个进程启动时分配的内存就会比结束时释放的多。
图4-33 (a)一个保存在磁盘文件中的程序。(b)一个进程的内部内存布局。在两图中最低的磁盘或内存地址都是在底端，最高的地址都是在顶端。
图4-33所示的是，一个程序怎样作为磁盘文件保存和怎样被转换为MINIX进程的内存布局。磁盘文件的头部包含了进程映象各部分的大小以及总的大小的信息。在具有共同的I和D空间的程序头部，有一个域指出正文和数据部分的总长度，这些部分被直接拷贝到内存映象中。映象中的数据部分增扩了头部bss域指出的数量，扩大的部分被清0，用于未初始化的静态数据。总共分配的内存数量是由文件头中的“total”域说明的。假如一个程序有4K正文段、2K的数据与bss、和1K堆栈，文件头中说明的需要分配的内存总量是4OK，那么在堆栈段和数据段之间的未用内存将是33K。在磁盘上的程序文件中还可以包含一个符号表，它仅用于调试，不装入内存。
如果程序员知道，程序 a.out 的数据和堆栈段增长所需要的内存总共最多是1OK，他可以使用下面的命令：
chmem = 10240 a.out
这个命令将修改文件头，使得在EXEC的时候内存管理器将为这个程序分配比初始的正文和数据段长度的和多10240字节的空间。在上面的例子中，当哪个文件随后被EXEC时将为它分配16K的内存，其中高端的1K用于堆栈，9K是空隙，供数据区或堆栈或者两者共同增长用。
对于使用独立的I和D段的的程序（由文件头中连接器设置的一位指出），文件头中的total域只对结合的数据段和堆栈段有用。一个有4K正文、2K数据、1K堆栈，total域为64K的程序将被分配68K的空间（4K指令空间，64K数据空间），留出61K空间供数据段和堆栈在运行时使用。数据段的界限只有通过BRK系统调用才能修改，BRK所作的全部工作就是检查新的数据段是否超越了当前的堆栈指针，如果没有，就修改一些内部表格以反映这个变动。这个操作涉及的只是当初分配给进程的内存区，操作系统不会分配额外的内存。如果新的数据段闯入了堆栈，这个调用将以失败告终。
采用这个策略是为了能在使用8088处理器的IBM PC机器上运行MINIX。在这种处理器中，硬件不检查堆栈溢出，一个用户程序可以在操作系统不知道的情况下把任意多个字推入堆栈。在具有更先进的内存管理硬件的计算机上，开始时堆栈被分配给一定数量的内存，如果堆栈试图增长超过这个数量就会引起一个操作系统陷入，然后如果可能，系统就为堆栈再分配一些空间。然而在8088上这个陷入不存在，因为堆栈会在不发出警告的情况下快速增长，所以让堆栈段与除了一大块未用内存之外的任何对象相邻都是十分危险的。MINIX的设计使得当它在一个具有更好的内存管理的计算机上实现时，修改MINIX内存管理器是非常直观的。
这里应该指出一个容易混淆的地方，当我们使用“段”这个词时，我们指的是操作系统定义的一个内存区域。Intel的80x86有一套内部的“段寄存器”和（在更高级的处理器上）“段描述符表”，他们为“段”提供硬件支持。Intel硬件设计者的段的概念类似于，但又不总是和MINIX定义和使用的段相同。在本书中所有对段的引用都应该被解释为对由MINIX数据结构所描述的内存区域的引用，在谈到硬件时我们将明确地使用段寄存器或段描述符。
这个警告可以推而广之。硬件设计者经常试图为期望在他们机器上运行的操作系统提供支持，因此用来描述寄

存器和处理器结构其他方面的术语通常会反映对这些特征将来用途的设想。这些特征对操作系统的实现者经常是有用的，但是使用的方法可能和硬件设计者预见的不同，同样的词在描述操作系统的一个方面和底层的硬件时有不同的含义就有可能导致误解。

4．7．2 消息处理

象MINIX所有其他部分一样，内存管理器是消息驱动的。在系统初始化完成之后，内存管理器就进入它的主循环，包括等待消息、执行消息中包含的请求、和发送应答。图4-34中的列表给出了合法的消息类型、他们的输入参数、和在应答消息中送回的值。

图 4-34 与内存管理器通讯时使用的消息类型、输入参数、应答值。

FORK、EXIT、WAIT、WAITPID、BRK、和EXEC很明显地与内存分配和释放紧密相关；KILL、ALARM、PAUSE、SIGACTION、SIGSUSPEND、SIGPENDING、SIGMASK、SIGRETURN等都与信号有关，因为当一个信号杀死进程时将释放进程的内存，所以他们也有可能影响内存；REBOOT对整个操作系统都有影响，但它的第一个任务是以一种受控制的方法发送信号结束所有的进程，因此内存管理器是恰当的对其进行处理之处；七个GET/SET与内存管理根本没有任何关系，他们与文件系统也没有任何关系。但是因为任何一个系统调用都是由两者中的一个处理的，而且文件系统已经足够大了，因此只能把他们放在内存管理器中；PTRACE用于调试，它也是由于同样的原因放到了这里。最后一条消息KSIG不是系统调用，它由内核用来向内存管理器发送SIGINT、SIGQUIT、SIGALRM等源自内核的信号。

系统提供库例程sbrk，但并无系统调用SBRK。该库例程把参数说明的增量或减量加到当前长度上，并以此作为参数调用BRK。同样地，不存在独立的系统调用geteuid和getegid，调用GETUID和GETGID既返回有效标识又返回真实标识。同样地，GETPID返回调用进程及其父进程的pid。

用于消息处理的一个关键数据结构是在table.c中（16515行）说明的call_vec表。它包含了指向处理不同类型消息的过程的指针。当一条消息来到内存管理器时，主循环抽出消息类型并把它放在全局变量mm_call中，随后这个值被作为call_vec的索引以找到处理新到消息的过程。紧接着调用这个过程执行系统调用。它返回的值被放到应答消息中送回调用者，以报告调用是成功还是失败。这个机制与图1-16的类似，差别在于它在用户空间而不是在内核。

4.7.3 内存管理器数据结构和算法

内存管理器有两个关键数据结构：进程表和空洞表。我们将逐个讨论。

在图2-4中我们看到进程管理需要进程表的一些域，内存管理需要另外一些域，文件系统又需要其他一些域。在MINIX中，操作系统的这三个部分都有自己的进程表，每个部分的表仅包含了自己需要的域。为了简单起见，表项是精确对应的，因此内存管理器的表项k和文件系统的表项k对应的是同一个进程。为了保持同步，在进程创建或结束时，这三个部分都要更新他们的表以反映新的情况。

内存管理器的进程表叫做mproc，定义在/usr/src/mm/mproc.h中。它包含了与进程内存分配有关的全部域和一些附加的信息。最重要的域是mp_seg数组，它有三个表项，分别用于正文、数据、和堆栈段。各个表项是一个由虚地址、物理地址、和段长度组成的结构。他们都是用块（click）而不是字节来量度的。块的大小是依赖于实现的，在标准的MINIX中是256字节。所有的段必须开始于块边界并且占据整数个块。

记录内存分配的方法如图4-35所示，该图中的一个进程有3K正文、4K数据、1K的空隙、和随后的2K堆栈，分配的全部内存是10K。假设这个进程没有独立的I和D空间，在图4-35(b)中我们看到这三个段各自的虚地址、物理地址、和长度域。在这个模型中，正文段总是空的，数据段包含了正文和数据。当进程引用虚地址0时，不管是跳转到它还是读它（即，在指令空间还是在数据空间），将使用物理地址0x3200（十进制是200K），这个地址位于块0x320。

图4-35 (a)内存中的一个进程。(b)这个进程在I和D空间非独立时的内存表示。(c)I和D空间独立时的内存表示。

值得注意的是堆栈的起始虚地址取决于分配给进程的内存总量。如果为了提供更大的动态空间（在数据段和堆栈之间更大的间隙）而用chmem命令修改了文件头，那么在下一次文件执行时堆栈将从一个更高的虚地址开始。如果堆栈增长了一块，那么堆栈的表项应该从三元组(0x20, 0x340, 0x8)变成三元组 (0x1F, 0x33F, 0x9)。

8088硬件没有段界限陷入，在32位处理器上，MINIX定义堆栈的方法使得在堆栈段已经覆盖数据段之前不会激发陷入。因此，在下一次BRK系统调用之前不会对堆栈表项做出上面的修改。那时，操作系统会专门读取SP并重新计算段表项。在具有堆栈陷入的机器上，一旦堆栈超出它所在的段，就可以立即更新堆栈段表项。然而在32位Intel处理器上的MINIX并没有这样做，下面下面我们将讨论其原因。

在前面我们提到，硬件设计者的努力并不总是能恰好产生软件设计者所需要的。即使是在Pentium的保护模式下，当堆栈增长超过它的段时MINIX也不会发生陷入。尽管处于保护方式的Intel硬件检测超出段界限（由段描述符定义，例如图4-28中的那个）的内存访问，但是在MINIX中数据段描述符和堆栈段描述符总是一样的。MINIX定义的数据和堆栈各自使用这个空间的一部分，因此两者都可以扩展进入位于他们之间的空隙中。在这里，只有MINIX可以管理这些，因为对硬件来说间隙既是数据段也是堆栈段合法的一部分，CPU根本无法检测涉及间隙的错误。当然，硬件能够检查非常大的错误，例如对超出结合的数据-间隙-堆栈区域的访问。这能够保护一个进程免受另外一个进程错误的影响，但却不足以保护一个进程免受自己错误的影响。

在这里我们作出了一个设计决定。我们意识到在放弃共享的硬件定义的段方面存在不同意见，这种段允许MINIX

动态地重分配间隙区域。另一种选择是用硬件定义不重叠的堆栈和数据段，这样作可以在某种程度上提供更高的安全性，以避免某些错误，但却使得MINIX内存的使用方面更加紧张。任何想评估一下另一种方法的人都可以得到它的源码。

图4-35(c)所示的是4-35(a)的内存布局在具有独立的I和D空间情况下的段表项。这里，正文和数据段的长度都不为零。图4-35(b)和图4-35(c)中的mm_seg数组主要是用于把虚地址映射成物理地址。给定一个虚地址和它所属的空间，那么确定虚地址是否合法（即它是否落在一个段的内部）以及在合法时它对应的物理地址是一件很简单的事情。例如，内核过程umap就为I/O任务和用户空间与内核空间间的拷贝实现这种映射。

图4-36 (a)一个具有独立的I和D空间进程的内存映射，与上图相同。(b)第二个进程启动后的内存布局，他们通过共享正文执行相同的程序映像。(c)第二个进程的内存映射。

一个进程的数据和堆栈区域的内容可能会随着进程的执行而改变，但正文不会改变。几个进程同时执行同一个程序拷贝的情况是很常见的，例如，几个用户可能会同时执行相同的shell，使用共享正文能提高内存的利用率。当EXEC要装入一个程序时，它首先打开保存着该程序磁盘映像的文件并读入文件头，如果进程使用的是独立的I和D空间，系统将搜索每个mproc表项的mm_dev、mm_ino、和mm_ctime域，这些域保存着相应进程正在执行的映像所在的设备、i结点号和修改时间。如果找到了一个正在执行相同程序的进程，那么就没有必要为正文的另一个拷贝分配内存，只要把新进程内存映射的mm_seg[T]部分初始化为指向已经装入正文段的位置，只有数据和堆栈部分被设置成新分配的内存。这个过程如图4-36所示。如果程序使用的是结合的I和D空间或合适的进程没有找到，内存将象图4-35那样分配，新进程的正文和数据将从磁盘上拷贝进来。

除了段信息之外，mproc还保存着进程自己的进程号(pid)和父进程的进程号、用户号(uid)和组号(gid)（真实和有效的）、关于信号的信息、以及在进程已经结束而父进程还没有执行对它的WAIT时的终止状态。

内存管理器另外一个主要的表是空洞表，定义在alloc.h中，它按照内存地址递增的顺序列出了内存中的各个空洞。数据和堆栈段之间的空隙不认为是空洞，他们已经被分配给进程了，所以他们没有被包含在空洞表中。每个空洞表项有三个域：以块为单位的空洞的基地址、以块为单位空洞的长度、和一个指向表中下一个入口的指针。这个表是单向连结的，所以从任何一个空洞开始找到下一个空洞很容易，但是如果要找上一个就必须从头开始搜索直到找到给定的空洞。

用块而不是字节为单位记录段和空洞的各种信息的原因非常简单：在16位状态模式下，内存地址用16位整数记录，在块为256字节时可以支持最大16M的内存；在32位模式下，地址域可以引用高达240字节的地址，即1024G。在空洞表上的主要操作是分配一块指定大小的内存和归还一块已经分配的内存。在分配内存时，首先从最低的地址开始搜索空洞表，直到找到一个足够大的空洞（首次适配）；随后，从空洞中减去段需要的空间，或在极少的大小恰好相等的情况下从空洞表中删除空洞来为段分配空间。这个方案简单快速，但是存在着内零头（最后一个块可能会浪费多达255字节空间，因为分配的总是整数个块）和外零头的问题。

在进程结束并被清理完后，它的数据和堆栈内存区归还给空洞表。如果进程使用的是结合的I和D空间，它的全部内存都将被释放；如果进程使用的是独立的I和D空间并且搜索进程表发现没有其他进程在共享其正文，那么它的正文区也将被释放。因为在共享正文时正文和数据区域未必相邻，所以也可能会归还两块内存。对于每个被归还的内存区域，如果它的任何一侧或者两侧邻接的区域也是空洞，则将他们合并，因此决不会出现邻接的空洞。这样，空洞的数量、位置和大小在系统运行期间将不断变化。一旦所有的用户进程都终止了，那么全部可用内存将又一次可供分配。然而这未必意味着只有一个空洞，因为物理内存可能会被操作系统不可用的区域截断，就象在IBM兼容机中那样只读存储器（ROM）和为I/O传输保留的内存把可用内存分为640K以下和1M以上两块。

4.7.4 FORK、EXIT、和WAIT系统调用

在创建和撤消进程时必须分配或释放内存、必须更新进程表，包括由内核和FS保存的部分。这些操作都由内存管理器协调。进程的创建是由FORK完成的，这时将执行图4-37所示的一系列步骤。

1. 检查进程表是否满了。
2. 试为子进程的数据和堆栈分配内存。
3. 把父进程的数据和堆栈复制到子进程的内存中。
4. 找到一个空闲的进程表项并把父进程的表项复制进去。
5. 在进程表中输入子进程的内存映射。
6. 为子进程选择一个进程号。
7. 告诉内核和文件系统子进程的情况。
8. 向内核报告子进程的内存映射。
9. 向父进程和子进程发送应答信息。

图 4-37 执行FORK系统调用需要执行的步骤。

半途中断FORK调用是困难的也是不方便的，所以为了比较容易地判断进程表是否有空闲位置，内存管理器总是保存着当前存在的进程的数目。如果进程表没有满，就尝试为子进程分配内存，对于具有独立的I和D空间的程序只需要为新的数据和堆栈段分配足够的空间。只要这一步成功了，FORK就保证能成功。随后填写新分配的内存、找到一个进程表项并填写、选择pid、通知系统的其他部分一个新进程已经被创建。

在下列两个事件都已经发生的情况下进程才会完全终止：(1)进程自己已经退出（或已经被一个信号杀死），(2)它的父进程已经执行了WAIT系统调用以观察发生了什么。已经退出或被杀死而它的父进程还没有为它执行WAIT的进程将进入某种挂起状态，有时被称为僵死状态(Zombie State)，这种进程不再参与调度，它的报警时钟被关闭（如果原来是开的），但它仍将留在进程表中。它的内存被释放。僵死是一种临时状态，很少会持续较长的时间，当父进程最后执行WAIT时，将释放进程表项，并通知文件系统和内核。

这样就出现了一个问题，如果要结束进程的父进程已经死了怎么办？如果不采取特殊处理，要结束的进程将永远处于僵死状态。对于这种情况，系统将修改进程表使这种孤儿进程成为init的子进程。在系统启动时，init读取/etc/ttytab文件以得到所有终端的清单，并为每个终端fork出一个注册进程，随后阻塞，等待进程结束。通过这个方法可以使孤儿很快被清理掉。

4.7.5 EXEC系统调用

当从终端输入一条命令被时，shell fork 出一个新进程，然后由它执行请求的命令。本来让一条系统调用一次完成FORK和EXEC两个操作是可以的，但是由于下述理由：“为了易于实现I/O重定向”，他们被作为两条不同的调用提供。在shell执行fork时，如果输入是重定向的，子进程在执行命令前先关闭标准输入再打开新的标准输入，这样新创建的进程将继承重定向过的标准输入，对标准输出的处理方法相同。

EXEC是MINIX中最复杂的系统调用，它用新的内存映像替换当前的内存映像，包括设置新的堆栈。它通过图4-38所示的一系列步骤完成其工作。

1. 检查权限—文件是否可执行？
2. 读取文件头得到各段长度和总长度。
3. 从调用者处取参数和环境。
4. 分配新内存和释放旧内存。
5. 把堆栈复制到新的内存映像中。
6. 把数据（可能还有正文）段复制到新的内存映像中。
7. 检查处理setuid、setgid位。
8. 设置进程表项。
9. 告诉内核进程现在是可运行的。

图 4-38 执行EXEC系统调用需要执行的步骤。

每一步都是由更小的步骤组成的，其中有些可能会失败，例如内存可能不够。执行检测的顺序经过了仔细的选择，以保证直到确定EXEC肯定会成功时才会释放旧的内存映像，以避免出现既不能生成新的内存映像又没有旧的内存映像可以回去的窘迫的情形。通常EXEC 不会返回，但是如果它失败了，调用进程必须再次得到控制，并得到一个错误指示。

对图4-38中的几个步骤应该作更多的注释。首先是是否有足够空间的问题。首先需要检查新进程是否可以共享其他进程的正文内存，以确定它所需的内存大小，然后搜索空洞表以检查在释放老的内存前是否有足够的物理存储器—如果老的内存先被释放了而内存又不够，我们将很难再恢复老的内存映像。

然而，这个测试是过于严格了，它有时会拒绝有些实际上可以成功的EXEC调用。例如，假设执行EXEC系统调用的的进程占用了20K，并且它的正文段没有被其他进程共享。再假设有一个30K的空洞，而新的映像需要50K。通过在释放以前的测试，我们将发现只有30K空间可用，因此会拒绝这个调用。但是如果我们先释放，这个调用也许会成功，取决于新的20K空洞是否与30K的空间相邻因而可以合并。稍复杂一点的实现可以更好地处理这种情况。

如果EXEC后的进程使用独立的I和D空间，则另一个可能的改进是搜索两个空洞，一个用于正文段一个用于数据段，两个段不需要邻接。

一个更微妙的问题是可执行文件是否能放得进虚地址空间。这个问题的原因在于内存是以256字节的块而不是字节为单位分配的。由于整个内存管理都是以块为单位的，每个块必须只属于一个段，不允许出现象一半数据一半堆栈这样的情形。

为了看到这个限制怎样会带来麻烦，请注意16位系统（8086和80286）的地址空间是限制在64K的，它可以被分为256个块。假设一个具有独立I和D空间的程序有40000字节的正文、32770字节的数据和32760字节的堆栈。数据段占用了129个块，其中最后一块只用了一部分，然而整个块还是数据段的一部分。堆栈段是128块。尽管此时数据段和堆栈段需要的字节数之和没有超出虚地址空间，但他们加起来超过了256块，所以并不能一起放入虚地址空间。在理论上，这个问题在所有的块大于1字节的机器上都存在，但是在实际上，因为Pentium系列处理器允许大的（4GB）的段，这个问题很少出现。

另一个重要的问题是如何设置初始堆栈。通常使用库函数execve来调用EXEC，其形式是：

```
execve(name, argv, envp);
```

这里name是指向将被执行的文件名的指针、argv是指向一个指针数组的指针，数组的每个成员指向一个参数、envp是指向一个指针数组的指针，数组的每个成员指向一个环境串。

如果仅仅把上面三个指针放在一条消息中传给内存管理器，而让它自己去取文件名和两个数组来实现EXEC是非常简单的。但是，内存管理器不得不一次一个地取各个参数和串，因为内存管理器无法事先知道各个参数或串

的长度，所以对每个参数或串至少需要向系统任务发一条甚至更多的消息。
为了避免用很多消息读取所有的参数和环境串，MINIX采用了一种完全不同的策略。由execve库过程在它的内部构造完整的初始堆栈，并把它的基地址和长度传给内存管理器。因为在用户空间构造新的堆栈时，对参数和串的引用都是都是局部内存访问，不存在对其他地址空间的访问，所以效率非常高。
图4-39 (a)传给execve的数组。(b)execve构造的栈。(c)经内存管理器重新定位后的栈。(d)在main开始运行时的栈。
为了更清楚地解释这个机制，让我们来看一个例子。当用户向shell键入了
ls -l f.c g.c
以后，shell将解释它并随后调用库过程
execve("/bin/ls", argv, argp);
两个指针数组的内容如图4-39(a)所示。位于shell地址空间中的过程execve现在将构造初始堆栈，这如图4-39(b)所示。这个堆栈在处理EXEC调用的过程中最终将原封不动地复制到内存管理器中。
在堆栈最后被拷贝到用户进程中时，它将不是被放在虚地址0，而是放在分配到的内存空间顶端，而该存储空间的大小是由可执行文件头中的总长度域决定的。例如，让我们随意假设总长度是8192字节，因此程序可以访问的最后一个字节位于地址8191。内存管理器负责在把堆栈放在新的地址时重新定位堆栈中的各个指针。这时的堆栈如图4-39(c)所示。
在EXEC调用结束，程序开始运行时的堆栈与图4-39(c)所示的完全相同，初始栈指针是8136。然而还有一个问题需要处理，被执行文件的主程序可能是这样说明的：
main(argc, argv, envp);
对C编译器来说，main只是一个普通的函数，它并不知道main是特别的，于是认为三个参数将按C的调用习惯（即最后一个参数在最前面）被传递进来，并按此编译访问这三个参数的代码。一个整数和两个指针，这三个参数应该占用返回地址前的三个字，图4-39(c)的堆栈看起来当然不是这个样子。
解决的方法是让程序不从main开始，而是把称为C运行起始过程的一小段汇编例程，crtso，总是被连接到正文地址0，由它首先得到控制。它的任务是把三个字推入堆栈然后用标准的调用指令调用main，这使得在main开始运行时的堆栈如图4-39(d)所示，因此main被欺骗了，认为它是以通常的方式被调用的（实际上这并不是真正的欺骗，它就是那样被调用的）。
图4-40 C运行时初启过程的关键部分。
如果程序员在main结束时忽略了调用exit，在main结束后控制会回到运行起始过程。跟前面一样，编译器只是把main看作一个普通的函数，在最后一个语句后生成通常的从函数中返回的指令。所以，main将返回到它的调用者—C运行起始过程。它随后调用exit退出。32位crtso的大部分代码如图4-40所示。需要指出的一点是省略的代码只是那些装入被压栈的寄存器和设置指示浮点处理器存在或不存在的标志的指令。

4.7.6 BRK系统调用

库过程brk和sbrk用来调整数据段的上限。前者的参数是绝对长度（以字节为单位），并以此调用BRK；后者的参数是相对于当前长度的正的或负的增量，并由此计算出新的数据段长度，然后调用BRK。不存在真的SBRK系统调用。
一个有趣的问题是：“sbrk是怎样跟踪当前的长度以便求出新长度的呢？”答案是有一个变量，brksize，总是保存着当前长度，所以sbrk可以得到这个值。这个变量被初始化为一个编译器生成的给出正文加数据（非独立的I和D）或仅仅数据（独立的I和D）的初始长度的符号。实际上这个名字和这样的符号是否存在都是依赖于编译器的，所以在源文件目录的任何头文件中都找不到它的定义，它被定义在库中，并包含在在文件brksize.s中。它的具体位置依赖于系统，但将在和crtso.s相同的目录中。
执行BRK对内存管理器来说是非常简单的，需要做的全部工作只是检查各个部分是否仍能放进地址空间、调整表格、并通知内核。

4.7.7 信号处理

在第一章中信号被描述成一种把信息传递给一个未必在等待输入的进程的机制。在系统中有一个预定义的信号集合，每个信号都有一个缺省动作—或者杀死接受信号的进程、或者忽略这个信号。如果只有这两种可能，那么信号处理将是很容易理解和实现的。然而，进程可以通过系统调用改变这些响应。进程可以要求忽略任一信号（除了特别的SIGKILL信号），而且进程还可以准备捕获一个信号（再一次除了SIGKILL信号），其方法是要求接收到该信号时激活进程内部的一个信号处理过程而不是执行缺省动作。因此对于程序员来说操作系统处理信号似乎有两个不同的时间：一个准备阶段，这时进程可以修改它对未来信号的响应；和一个响应阶段，这时信号发生了并且相应的动作被执行。动作可能是执行用户编写的信号处理程序，实际上还有第三个阶段。在用户编写的处理程序结束时，一个特殊的系统调用清理并恢复接受信号进程的正常操作。程序员实际上不需要知道这第三个阶段，他编写处理程序就象写其他函数一样，由操作系统负责处理激发与结束处理程序和管理堆栈。
在准备阶段有几个进程可以在任何时间执行以修改它对信号的响应的系统调用。其中最通用的是SIGACTION，用它可以说明进程将忽略某些信号、捕获某些信号（用执行位于进程内部、用户定义的信号处理程序替换缺省处理）、或者恢复某些信号的缺省处理。另一个系统调用SIGPROCMASK可以阻塞一个信号，它使得一个信号被暂时

保存起来，只有当进程在后来某个时候解除对这个信号的阻塞时才会响应这个信号。可以在任何时候使用这些调用，甚至可以在信号处理过程内部。在MINIX中，因为需要的数据结构都在内存管理器中，所以信号处理的准备阶段完全由内存管理器处理。每个进程都有几个sigset_t变量，每个可能的信号由他们中的一个位表示。一个这样的变量定义要忽略的信号集，另一个定义要捕获的信号集，依此类推。每个进程还都有一个sigaction结构的数组，每个信号对应一个数组元素，每个sigaction结构元素都包含两个变量：一个变量保存相应信号指定处理过程的地址，另一个变量sigset_t用于指定在该处理过程执行时应阻塞的信号，保存处理过程地址的域也可以保存指出信号将被忽略或将以缺省的方式处理的特殊值。

在信号发生时，MNIX的多个部分都将被涉及。响应从内存管理器开始，它使用刚刚说明的数据结构，指出哪个进程应该得到这个信号。如果信号应该被捕获，就必须把它传递给目的进程，这需要保存有关进程状态的信息以使进程可以恢复正常运行。这些信息被保存在接收信号进程的堆栈上，因此必须检查是否有足够的堆栈空间。因为这是内存管理器的领地，所以这个检查由它执行，随后内存管理器调用在内核中的一个系统任务把这些信息放到堆栈上。系统任务还处理进程的程序计数器，使进程能够执行信号处理过程的代码。在信号处理过程结束时将执行一个SIGRETURN系统调用，通过这个调用内存管理器和内核共同恢复进程的信号上下文和寄存器，使进程可以恢复正常运行。如果信号不需要捕获，缺省的动作将被执行，这可能涉及调用文件系统生成一个core文件（把进程的映像写到一个文件里，它可能会被调试器检查），或者杀死进程，这涉及到内存管理器、文件系统、内核。最后，因为一个信号可能需要传送给一组进程，所以内存管理器可能使这些动作重复多次。

图4-41 POSIX和MINIX定义的信号。(*)标记的信号依赖于硬件的支持。(M)标记的信号POSIX没有定义，而MINIX为了与老程序兼容定义了。一些过时的名字和同义词没有给出。

MINIX知道的信号定义在/usr/include/signal.h中，这是POSIX标准所要求的一个文件。他们列在图4-41中。所有POSIX需要的信号在MINIX中都定义了，但目前并不是所有的信号都得到了支持。例如，POSIX需要一些与作业控制，即把一个正在运行的程序放到后台和再放回来的能力，有关的信号，MINIX不支持作业控制，但是可能产生这些信号的程序可以移植到MINIX上。这些信号如果产生将被忽略。MINIX还定义一些非POSIX信号和一些为了与旧的源码兼容的POSIX名的同义词。

信号可以通过两种方式产生：由KILL系统调用和由内核产生。由MINIX内核产生的信号总是包括SIGINT、SIGQUIT、SIFALRM，其他的内核信号依赖于硬件支持。例如，在8086/8088处理器上不支持检测非法指令操作码，但286及以上的处理器有这个能力，在他们上面执行非法指令时会发生陷入。这种服务是硬件提供的，操作系统的实现者必须提供产生一个信号的代码以作为对这种陷入的响应，我们在第二章看到在/kernel/exception.c中包含了针对几个不同条件完成上述操作的代码。因此当MINIX运行在286或更高的处理器上时，作为对非法指令的响应它将产生SIGILL信号，但当MINIX运行在8088处理器上时则永远不会产生这个信号。

仅仅是硬件在某种条件下能够陷入并不意味着操作系统实现者能够充分地利用这个能力。例如，在286以上的Intel处理器上有好几种违反内存完整性的操作都会导致异常，位于/kernel/exception.c中的代码把这些异常翻译成SIGSEGV信号。违反硬件定义的堆栈段的界限和违反其他段的界限会产生不同的异常，因为他们可能需要进行不同的处理。然而MINIX使用内存的方式使得硬件对可能发生的所有这些错误都无法检测。硬件为每个段都定义了一个基址和一个长度。硬件定义的数据段基址和MINIX的数据段基址相同，但是硬件定义的数据段界限要比MINIX用软件强制的界限高。换句话说，硬件定义的数据段是在不存在堆栈的情况下，MINIX可能用于数据的最大数量的内存。同样地，硬件定义的堆栈段是在不存在数据区的情况下，MINIX堆栈能用的最大数量的内存。尽管硬件能够检测到某些违法的访问，但是因为对硬件和描述符来说数据区和堆栈区域是重叠的，所以它无法检测堆栈增长进入了数据区域这种最可能发生的堆栈访问违法。

在内核中可以加入一些代码，在每个进程每次得到机会运行后由这些代码检查进程的寄存器，如果检测到违反MINIX定义的数据段或堆栈段完整性的情况，就发出一个SIGSEGV信号。但是这样作是否值得还不清楚，硬件陷入能立刻俘获一个违法访问，而软件检查可能要到执行了几千条指令以后才有机会进行，到了这个时候信号处理过程已经几乎不能为从错误中恢复做什么了。

不论信号来自什么地方，内存管理器都以同样的方式处理。对每个信号欲传向的进程，首先将进行一系列的检查以确定这个信号是否可行。只有当信号发送者是超级用户，或者它的真实或有效用户号等于接受信号进程的真实或有效用户号时，它才能发信号给接受信号的进程。除此之外还有几个条件能阻止一个信号被发送，比如僵死的进程不能接收信号。如果一个进程已经调用SIGACTION忽略了某个信号或调用SIGPROCMASK阻塞了某个信号，那么就不能向它递交信号。阻塞一个信号与忽略不同，接收到的阻塞的信号将被记录下来，当接收进程取消对这个信号的阻塞时就将该信号递交给它。最后，如果接收信号的进程堆栈空间不够它将被杀死。

如果所有的条件都满足，信号将被发送。如果进程没有设置捕获这个信号，那么就不需要向进程传递任何信息。在这种情况下内存管理器将执行这个信号的缺省动作，通常是杀死进程，有时需要产生一个core文件，少数信号的缺省动作是忽略这个信号。在图4-41中标记为“不支持”的信号是POSIX要求定义的，他们在MINIX中将被忽略。

捕获一个信号意味着执行进程自己的信号处理代码，它的地址保存在进程表的sigaction结构中。在第二章中我们看到了一个进程表项中的栈框是怎样在该进程被中断时接收重启它所需的信息的。通过修改将接收信号进程的栈框，可以使得在该进程被允许执行时，运行信号处理程序。通过修改进程在用户空间的堆栈，可以使得在

信号处理程序结束时执行一个SIGRETURN系统调用。这个系统调用永远不会从用户代码中发出，它是通过内核把它的地址放到堆栈上，使得在信号处理程序结束时成为从堆栈中弹出的返回地址而被执行的。SIGRETURN恢复接受信号进程的栈框，使它能够从被信号中断的地方恢复运行。

虽然发送信号的最后一个阶段是由系统进程完成的，但是由于数据是从内存管理器传到内核的，所以在这里总结一下执行的过程是很恰当的。在信号处理过程结束时进程应该象什么都没有发生一样继续执行，因此捕获一个信号需要某种与进程上下文切换很类似的东西，而上下文切换是在一个进程停止运行、另一个进程开始执行时发生的。然而在进程表中只有一个地方，其中可以存放将进程恢复到原来状态所需的所有CPU寄存器的内容。解决这个问题的方法如图4-42所示。图的(a)部分是一个进程在中断发生后刚刚停止运行时的简化的进程堆栈和部分进程表项的视图。在进程挂起时，CPU所有的寄存器都被拷贝到进程表内核部分的进程表项的栈框结构中，因为信号是由不同于接收者的进程或任务产生的，所以这就是信号产生时的情形。

图 4-42 一个进程在处理信号不同阶段的栈（上面），和它在进程表中的栈框（下面）。(a)进程停止运行时的状态。(b)信号处理过程开始运行时的状态。(c)执行SIGRETURN时的状态。(d)SIGRETURN执行完之后的状态。

在准备处理信号的过程中，进程表中的栈框被作为一个sigcontext结构复制到进程自己的堆栈中，从而将它保存起来。随后一个sigframe结构被放到堆栈上,这个结构包含了在信号处理过程结束后SIGRETURN将使用的信息。它还包含了调用SIGRETURN的库过程的地址 ret addr1， 和被中断的进程将来恢复运行的地址 ret addr2。然而，后一个地址在正常运行时是不使用的。

虽然信号处理过程是程序员写的一个普通过程，它却不是通过call指令被调用的。位于进程表栈框中的指令指针（程序计数器）被改变，使得当restart把接收信号的进程投入运行时，将执行信号处理过程。图4-42(b)所示是这个准备工作，已经完成信号处理过程开始运行时的情况。记住，信号处理过程是一个普通的过程，在它结束时 ret addr1 将被弹出，SIGRETURN将被执行。

(c)部分所示是SIGRETURN正在执行时的情况。sigframe的其他部分现在是SIGRETURN局部变量。SIGRETURN的部分工作是调整它自己的堆栈指针使得在它象一个普通函数那样结束时，它将用 ret addr2作为它的返回地址。然而，SIGRETURN实际上并不是这样结束的。它结束得和其他系统调用一样，允许内核中的调度程序决定重启哪个进程。最后，接收信号的进程将被重新调度到，并从存放在进程原来栈框中的地址重新启动。把这个地址放在堆栈上的原因是用户可能想要用调试器跟踪程序，当一个信号处理过程被跟踪时这样做能欺骗调试器并给它一个关于堆栈的合理的解释。在每个阶段，堆栈看起来都象一个普通进程的堆栈，局部变量在返回地址之上。SIGRETURN的实际工作是把各个部分恢复成他们接收信号以前的状态，并进行清理。最重要的是通过使用保存在接收信号进程堆栈中的拷贝，进程表中的栈框被恢复到它原来的状态。SIGRETURN结束时的情况如图4-22(d)所示，从中可以看出等待恢复执行的进程的状态与它刚被中断时的状态相同。

大部分信号缺省动作是杀死接收信号的进程。内存管理器负责处理满足下列条件的各种信号：对他们的缺省动作不是忽略、并且接收进程没有处理、阻塞、或忽略他们。如果父进程正在等待，被杀死的进程将被清理并从进程表中删除；如果父进程没有等待，它将进入僵死状态。对特定的信号（如SIGQUIT），内存管理器还要在进程的当前目录下写一个core文件。

一个信号被发往一个因用READ读一个目前没有输入的终端而被阻塞的进程，这样情形是很常见的。如果进程没有说明这个信号将被捕获，系统将简单地按照通常的方法杀死该进程。但是如果信号被捕获了，这里就出现了一个问题：在信号中断被处理完之后应该怎么办？进程应该继续等待还是执行下一个语句？

MINIX是这样处理的：系统调用将被终止并返回错误代码EINTR，让进程可以知道调用是被一个信号中断的。确定一个进程是否被阻塞并不是一件很简单的事，内存管理器必须请求文件系统进行检查。

POSIX建议可使用这种处理方法，但并不要求，它也允许READ在接到一个信号时返回它在接到信号时已经读到的字节数。返回EINTR使设置一个时钟报警并捕获SIGALRM成为可能，这是一种简单的实现超时的方法，例如，如果用户在指定的时间内没有反应就中止login，并挂断调制解调器。使用同步时钟进程可以用较小的开销做同样的事情，但它是MINIX的发明，没有使用信号移植性好而且只有服务器进程可以使用，用户进程不能使用。

4.7.8 其他系统调用

内存管理器还处理一些更简单的系统调用。库函数getuid和geteuid都调用GETUID系统调用，在它的返回信息中包括两个值。同样地，GETGID系统调用返回真实和有效值，以供getgid和getegid库函数使用。GETPID也以同样的方式工作，它返回进程号和父进程号。SETUID和SETGID可以在一个调用中同时设置真实和有效值。在这一组调用中还有两个调用，GETPGRP和SETSID，前者返回进程组号，后者把它设置为当前的进程号。

PTRACE和REBOOT系统调用也由内存管理器处理，前者支持程序的调试，后者影响到系统的很多方面。把它放在内存管理器中是合适的，因为它的第一个动作就是杀死除了init以外的所有进程，然后调用文件系统和系统任务完成它的工作。

4.8 MINIX中内存管理的实现

在一般地浏览了内存管理器的工作情况以后，让我们看看代码本身。内存管理器完全用C写成，代码是直观的并且包含有很多注释，所以大部分不需要长的或深入的说明。我们将首先简单地看一看头文件，然后是主程序，最后是上面讨论过的各个系统调用组的文件。

4.8.1 头文件和数据结构

内存管理器源程序目录中的几个头文件和内核目录中的文件有相同的名字，这些名字还将在文件系统中再一次见到。这些文件在他们自己的上下文中有相似的函数。设计并行结构的目的是为了易于理解整个MINIX系统的组织。内存管理器还有一些具有独特名字的头文件。与系统的其他部分一样，在内存管理器版本的table.c被编译时，全局变量的存储空间将被保留。这一节我们将看一看所有的头文件和table.c。

与MINIX其他主要部分一样，内存管理器有一个主头文件mm.h（15800行）。每个文件编译时都包含它，它自己则包含了每个目标模块都需要的位于/usr/include和它的子目录中的所有系统范围的头文件。大部分包含在/kernel/kernel.h中的文件也被包含到了这里。内存管理器还需要include/fcntl.h和include/unistd.h中的定义。内存管理器自己版本的const.h、type.h、glo.h也包含在里面。

const.h（15900行）定义了内存管理器用的常量，其中有些是特别为16位机器设置的。其中还包含下列符号定义语句：

```
#define printf printk
```

这使对printf的调用被编译成对函数printk的调用。这个函数与我们在内核中看到的那个类似，定义也是为了同样的原因，使内存管理器能在不请求文件系统帮助的情况下显示错误和调试信息。

type.h目前没有使用，它只是以一个框架的形式存在，使内存管理器和MINIX其他部分有同样的组织。proto.h（16100行）收集了在整个内存管理器内需要的函数原形。

内存管理器的全局变量在glo.h中（16200行）说明。内核中用到的关于EXTERN的技巧也用到了这里，即除了在文件table.c中，EXTERN是一个扩展为extern的普通的宏。而在文件table.c中，它变成一个空串，使得用EXTERN说明的变量被分配空间。

这些变量中的第一个，mp，是指向一个mproc结构的指针，它是某进程表的MM部分，而该进程的系统调用正在被处理。第二个变量，dont_reply，在每一个新的请求到来时被初始化为FALSE，但在调用的执行过程中如果发现没有应答信息应该发送，它就被设置为TRUE。例如成功的EXEC就没有应答发送。第三个变量，proc_in_use，跟踪当前使用了多少个进程表项，使确定FORK是否可行非常简单。

消息缓冲区mm_in和mm_out是分别用于请求和应答消息的。who是当前进程的索引，它与mp的关系是

```
mp = &mproc[who];
```

当一条消息到达的时候，系统调用号被抽取出来放到mm_call中。

三个变量err_code、result2、和res_ptr用来保存在应答消息中返回给调用者的值。其中最重要的是变量err_code，在调用没有错误而结束时，它将被设置为OK。最后两个变量在发生问题时使用。在进程非正常结束时，MINIX把进程的映像写到一个core文件中，core_name定义了这个文件将取的名字，core_sset是一个定义哪些信号应该产生core转储文件的位图。

进程表的内存管理器部分在另一个文件mproc.h中（16300行）。大部分的域都由注释做了恰当的描述。几个域和信号处理有关，mp_ignore、mp_catch、mp_sigmask、mp_sigmask2、和mp_sigpending是位图，每个位代表一个可以送往进程的信号。类型sigset_t是32位整数，因此MINIX可以容易地支持32个信号，但当前只定义了16个，信号1是最低（最右）位。在任何情况下，POSIX都要求提供标准的函数，以添加或删除这些位图表示的信号集成员，因此所有必须的操作都可以在程序员不知道细节的情况下完成。数组mp_sigact对处理信号是非常重要的。对每个信号类型都有一个数组元素，每个元素是一个sigaction结构（定义在 include/signal.h 中），每个sigaction结构由三个域组成：

1. sa_handler域定义信号是按照缺省的方式处理、忽略、还是由专门的处理过程处理。
2. sa_mask域的类型是sigset_t，它定义了在这个信号正在被用户的处理过程处理时，哪些信号将被阻塞。
3. sa_flags是一些信号处理用的标志。

这个数组使得对信号的处理能够有很大的灵活性。

如文件尾部所示，mp_flags域用来保存一些位。这个域是一个无符号整数，在低端CPU上它是16位，在386和以上的CPU上则是32位。因为只用了9位，所以它有足够的扩充空间，即使在8088上也是如此。

进程表中的最后一个域是mp_procargs。在一个新进程启动时，系统将建立一个类似图4-39的堆栈，指向新进程argv数组起始地址的指针被存放在这里，它被ps命令使用。对于图4-39的例子，值8164将被保存在这里，如果在ls活动时执行ps命令，那么它将显示出命令行：

```
ls  -l  f.c  g.c
```

下一个文件是param.h（16400行），它包含了许多用于请求信息中的系统调用参数的宏和四个用于应答消息中域的宏。当下面这一行：

```
k=pid;
```

出现在任何包含了param.h的文件中时，在把它送往编译器之前预处理器将其转换成：

```
k=mm_in.m1_i1;
```

在介绍执行代码之前让我们看一看table.c（16500行），它的编译将为我们在glo.h和mproc.h中看到的各种EXTERN变量和结构保留空间。语句：

```
#define _TABLE
```

使EXTERN成为空串，这与我们在内核代码中看到的方法一样。

table.c另一个重要的成份是数组call_vec（16515行）。当一个请求消息到达时，其中的系统调用号将被取出作为call_vec的索引，以找到处理这个系统调用的过程。不是合法调用的系统调用号都会引起执行no_sys，它只是返回一个错误代码。值得注意的是虽然_PROTOTYPE宏用在了call_vec的定义中，这个定义并不是一个原形定义，而是一个初始化的数组的定义。但是因为它是一个函数的数组，使用_PROTOTYPE宏是使程序与经典C（Kernighan & Ritchie）和标准C都兼容的最简单的办法。

4.8.2 主程序

内存管理器是独立于内核和文件系统编译和连接的，所以它有自己的主程序。它的主程序在内核初始化自己之后被启动。主程序在main.c中（16627行），在通过调用mm_init完成自己的初始化之后，内存管理器进入16636行的循环。在这个循环中，它调用get_work等待到来的请求消息，然后通过call_vec调用某个do_XXX过程执行请求，最后如果需要的话发送应答。这种结构现在应该很熟悉了，它和I/O任务所用的一样。

过程get_work（16663行）和reply（16676行）分别处理实际的接收和发送。

这个文件的最后一个过程是mm_init，它初始化内存管理器。在系统开始运行之后它将不再被使用。16730行对sys_getmap的调用得到核心的内存使用信息，16734到16741的循环为任务和服务器初始化所有的进程表项，随后的几行准备init的进程表项。在16749行内存管理器等待文件系统给它发消息，正如在对MINIX中死锁处理的讨论中提到的，这是唯一的一次文件系统向内存管理器发送请求消息。该消息说明多少内存被用于RAM磁盘。16755行对mem_init的调用通过调用系统任务初始化空洞表。在这以后，正常的内存管理就可以开始了。这个调用还填充total_clicks和free_clicks变量。此后，打印、显示内存总量、核心内存的使用、RAM磁盘的大小和空闲内存情况。在打印完信息之后，向文件系统发送一个应答（16764行），使它能够继续。最后，为了方便ps命令，进程表的内存管理器部分的地址被送给内存任务。

4.8.3 FORK、EXIT、和WAIT的实现

FORK、EXIT、和WAIT系统调用是由文件forkexit.c中的do_fork、do_exit、do_wait过程实现的。过程do_fork（16832行）根据图4-37所示的步骤执行。要注意的是对procs_in_use的第二次调用（16847行）为超级用户保留最后几个进程表项。在计算子进程需要多大的内存时，数据段和堆栈之间的空隙也被包括在内，但不包括正文段。这时或者父进程的正文被共享，或者进程有结合的I和D 空间，它的正文段长度为零。计算完成之后，调用alloc_mem以得到内存。如果成功了，子进程和父进程的基地址将被从块转换为绝对字节，然后调用sys_copy发送消息到系统任务由其完成拷贝工作。

现在在进程表中已经找到了一个空位，早先涉及proc_in_use的测试保证了肯定存在一个空位。然后在其中填入各种内容，首先把父进程的表项复制到这里，然后修改mm_parent、mm_flags、mm_seg、mm_exitstatus、和mm_sigstatus。这些域中有些需要特殊处理，因为子进程不继承跟踪状态，所以mm_flags中的TRACED位被置零。mm_seg域是一个包含用于正文、数据、和堆栈段的元素的数组，如果是具有独立的I和D空间可以共享正文的程序，那么它的正文部分将指向父进程的正文段。

下一步是为子进程指定进程号。变量next_pid跟踪将被指定的下一个进程号。这时在理论上可能会发生下面所述的问题：在把一个进程号，比如20，赋给一个非常长寿的进程后，可能会有30000多个进程被创建和撤消，next_pid可能又再次回到20。指定一个仍然在使用的进程号将是一场灾难（设想随后某个进程向进程20发信号的情形），所以我们要搜索整个进程表以确定将被指定的进程号没有被使用。

对sys_fork和tell_fs的调用分别通知内核和文件系统新进程已经被创建，使他们能够更新他们的进程表。（所有以sys_开头的过程都是向内核中的系统进程发送一条消息请求图3-50中一个服务的库例程。）进程的创建和撤消都是从内存管理器开始，并在结束时传播到内核和文件系统。

对子进程的应答信息是在do_fork的结尾处发送的。对父进程的应答包含有子进程的进程号，它象对普通请求的应答一样是由main中的循环发出的。

下一个由内存管理器处理的系统调用是EXIT。过程do_mm_exit（16912行）接收这个调用，但大部分工作是调用mm_exit（16927行）完成的。这样划分工作是因为mm_exit()也被用来处理被信号终止运行的进程。两者工作相同，但参数不同，所以这样划分是很方便的。

mm_exit做的第一件事情是，如果进程有一个定时器在运行就停止它；其次，通知内核和文件系统这个进程不再可以运行（16949和16950行）。对库例程sys_xit的调用发一条消息给系统任务告诉它标志该进程不可运行，使它不会再被调度到。然后是释放内存。对find_share的调用确定正文段是否正在与另一个进程共享，如果没有，就调用free_mem释放它。紧接着调用同一个过程释放数据和堆栈。确定能否在一次free_mem调用中释放所有的内存是不值得的。如果父进程在等待，mm_exit就调用cleanup释放进程表项。如果父进程没有等待，就让进程进入僵死状态。这由mm_flags中的HANGING位指示。无论进程是被彻底消灭还是进入僵死状态，mm-exit的最后一个动作都是搜索进程表寻找刚才终止进程的子进程（16975行到16982行），如果找到就把它们变成init的子进程。如果init正在等待并且一个子进程进入了HANGING状态，它将调用cleanup处理这个子进程。这个方法可以处理图4-43(a)所示的这类情况。在图中我们看到进程12要结束了，它的父进程7正在等待它，这时cleanup将被调用以除掉12，因此52和53将成为init的子进程，如图4-43(b)所示。现在的情况是已经结束的进程53成为一个正在执行WAIT的进程的子进程，所以它也将被清理。

图4-43　(a)进程12要结束时的情况。(b)进程12结束以后的情况。

当父进程执行WAIT或WAITPID时，控制转移到16992行的过程do_waitpid。两个调用的参数不同，期望的动作也不同，但在17009到17011行通过设置内部变量使do_waitpid能够完成两个调用中任何一个的动作。从17019到17041行的循环扫描整个进程表，检查这个进程是否有子进程，如果有则检查他们是否处于现在可以被清理的僵死状态，如果找到了一个僵死状态的（17026行），do_waitpid将清理掉它并返回。设置dont_reply是因为对父进程的应答是从cleanup内部而不是main中的循环中发出的。如果找到了一个被跟踪的子进程，do_waitpid将送回一个指示这个进程已经停止的应答并返回，这里也同样把dont_reply位设置为真，以阻止main发送第二个应答。

如果执行WAIT的进程没有子进程，它将简单地得到一个错误返回（17053行）；如果它有子进程，但是其中没有一个处于僵死或被跟踪的状态，系统将检查在do_waitpid被调用时是否设置了一个标志父进程不想等待的位。如果没有设置（通常的情况），17047行将设置一个位指示它在等待，父进程将被挂起直到一个子进程结束。

当一个进程已经结束并且它的父进程在等待它时，不管这些事件发生的次序如何，过程cleanup都将被调用执行最后的操作。这时要做的工作不多，把父进程从WAIT或WAITPID中唤醒并给它终止子进程的进程号，和退出及信号状态。这时文件系统已经释放了子进程的内存，内核已经停止了对它的调度，因此现在内核要做的全部工作就是释放子进程在进程表中的表项。

4.8.4 EXEC的实现

EXEC代码按图4-38中的工作步骤编写，它被包含在过程do_exec（17140行）中。在几个简单的合法性检查之后，内存管理器从用户空间取到要执行的文件的名字。在17172行它向文件系统发一个特殊的消息以切换到用户的目录，使刚刚取到的路径被解释为相对于用户而不是内存管理器的工作目录的路径。

如果该文件存在并且可执行，内存管理器将读入其文件头并抽出各段长度。随后从用户空间取来堆栈（17188和17189行）、检查新进程是否能与已经在运行的进程共享正文（17196行）、为新的内存映像分配内存（17199行）、修改指针[参见图4-39(b)和(c)的区别]、和读入正文段（如果需要）和数据段（17221到17226行）。最后，处理setuid和setgid位、更新进程表项、告诉内核它已经结束工作，因此这个进程可以再被调度。

尽管对所有步骤的控制都在do_exec中，但是许多细节都是由exec.c中的许多辅助过程完成的。例如read_header（17272行）不仅读取文件头返回各段长度，它还验证该文件对指定的CPU类型是有效的MINIX执行文件，这是通过在内存管理器被编译时，条件编译适当的测试语句实现的。read_header还验证所有的段能否放入虚地址空间。

过程new_mem（17366行）检查是否有足够的内存用于新的内存映像。如果正文是共享的它就只查找足够容纳数据和堆栈的空洞，否则查找足够容纳结合的正文、数据和堆栈段的单个空洞。这里一个可能的改进是查找两个分开的空洞，一个用于正文，另一个用于数据和堆栈，原因是这两个区域不需要邻接，而在MINIX的早期版本中这是需要的。如果找到了足够的内存，旧的内存将被释放，新的将被分配。如果没有足够的内存，EXEC将失败。在新的内存分配后，new_mem将更新内存图（在mp_seg中），并通过调用库例程sys_newmap把它通知内核。

new_mem余下部分涉及到把bss段、空隙和堆栈段清零（bss段是数据段中包含所有未初始化的全局变量的那一部分）。许多编译器会产生将bss段清零的代码，但这样做使MINIX与不生成这种代码的编译器也能合作。数据段和堆栈段之间的空隙也被清零，因此当数据段被BRK系统调用扩充时，新获得的内存将全部是零。这不仅是为了方便程序员，他可以指望新变量初值为零，在多用户操作系统上这还是一个安全措施，以前占用这块内存的进程可能在其中存放了不该被其他进程看到的数据。

下一个过程是patch_ptr（17465行），它把图4-39(b)指针重新定位成图4-39(c)的形式。这种工作很简单：检查堆栈，找到所有的指针并把基地址加到每个指针上面。

每次EXEC调用过程load_seg（17498行）一次到两次，完成可能需要的装入正文段和总是需要的装入数据段的工作。我们在这里使用了一个技巧，以允许文件系统把整个段直接装入用户空间，而不是一块一块地从文件中读入然后拷贝到用户空间。实际上，这个调用被文件系统以一种略微特殊的方式解码，使它看起来就象是用户进程自己读取一个整段。只有文件系统读例程的头几行知道这里面作了特殊的处理，这个策略使装入略微得到加速。

exec.c的最后一个过程是find_share（7535行），它把要被执行的文件的i结点、设备、和修改时间与现有进程的进行比较，寻找可以共享正文段的进程，这只是对mproc中合适域进行的一个简单搜索。当然，必须忽略为了它进行这次搜索的那个进程。

4.8.5 BRK的实现

正如我们刚刚看到的，MINIX使用的内存模型是相当简单的：每个进程在创建时获得一块用于数据和堆栈的连续的内存块，它永远不会在内存中被移动、永远不会被交换出内存、永远不会增长、也永远不会缩小。唯一能够发生的是数据段可能会从低端吃掉一些空隙，以及堆栈段可能会从高端吃掉一些空隙。在这些情况下，位于break.c中的BRK调用的实现是特别简单的，它首先检查新的大小是否可行，随后更新表格以反映这些变化。

顶层的过程是do_brk（17628行），但大部分工作是由adjust（17661行）完成的。后者检查数据段和堆栈段是否冲突。如果冲突，BRK调用就不能执行，但进程并不是立刻被杀死。在进行测试前，一个安全系数SAFETY_BYTES被加到数据段的顶上，所以虽然可以作出堆栈过度增长的结论，但同时在堆栈中仍会有足够供进程继续运行一小段时间。它取回控制（带着一个错误信息），这样它可以打印出合适的错误信息，然后优美地结束。

值得注意的是SAFETY_BYTES是在过程的中间（17693行）用一个#define语句定义的。这种用法是很不同寻常的，

在通常情况下这种定义出现在文件的开头或独立的头文件中。相关的注释指出，程序员发现确定安全因数的大小是非常困难的，这样定义毫无疑问是为了引起注意，也许还是为了鼓励进一步的实验。
数据段的基址是一个常数，所以如果adjust必须调整数据段，它只需要更新长度域。堆栈从一个固定的终点向下增长，因此如果adjust注意到作为参数传递给它的堆栈指针已经增长超出了堆栈段（到了更低的地址），那么长度和起点都将被更新。
最后一个过程size_ok()（17736行）以块和字节为单位检查段的长度是否能被地址空间容纳下。为了说明为什么它被写成了一个单独的函数，用于16位机器的条件代码被保留在清单中，在用于32位的MINIX上把它作为一个单独的函数是没有什么意义的。它只在两个地方被调用，并且用17765行替换调用将使代码更加紧凑，因为调用要传递几个在32位实现上根本不用的参数。
4.8.6 信号处理的实现
与信号处理有关的系统调用有八个。他们摘要示于图4-44中。这些系统调用和信号自己都是在signal.c中处理的。还有一个系统调用REBOOT，因为它用信号终止所有的进程，所以也是在这个文件中处理的。
系统调用
目的
SIGACTION
修改对未来信号的响应
SIGPROCMASK
改变阻塞信号集合
KILL
向另一个进程发信号
ALARM
经过延迟后向自己发ALRM信号
PAUSE
挂起自己直到收到未来的信号
SIGSUSPEND
改变阻塞信号集合，然后PAUSE
SIGPENDING
检查未处理的（阻塞的）信号
SIGRETURN
在信号处理过程后进行清理
图 4-44 与信号处理有关的系统调用
SIGACTION调用支持sigaction和signal函数，他们使进程能改变自己对信号的响应方式。sigaction是POSIX要求的，大多数情况下它是首选的调用，而signal库函数是标准C要求的，要移植到非POSIX系统上的程序应该用它来写。do_sigaction的代码(17845行)开始于检查信号号码的合法性,和验证这个调用没有试图改变对SIGKILL信号的响应（17851到17852行）（忽略、俘获、或阻塞SIGKILL是不允许的，SIGKILL是用户能够控制自己的进程和管理员能够控制用户的最后手段）。SIGACTION被调用时带有指向一个sigaction结构的指针sig_osa，它接收在调用执行以前的有效的旧的信号处理属性，和另一个这样的结构sig_nsa，它包含了新的属性。
第一步是调用系统任务把当前的属性拷贝到sig_osa所指的结构中。把NULL指针放在sig_osa 中调用SIGACTION，这可以检查旧的信号处理属性而不修改他们，在这种情况下do_sigaction会立刻返回（17860行）。如果sig_nsa不是NULL，定义新的信号动作的结构将被复制到内存管理器的空间中。17867到17877行的代码根据新的动作是忽略信号、使用缺省处理、或者是捕获信号修改位图mp_catch、mp_ignore、和mp_sigpending。尽管这些都是可以用简单的宏实现的直观的位操作，这里还是使用了库函数sigaddset和sigdelset。这些函数是POSIX标准的要求，目的是使使用他们的程序容易移植，甚至是移植到信号数目大于一个整数可用的比特数的系统上。通过使用库函数使MINIX自己能更容易地移植到不同结构的系统上。
最后，进程表内存管理器部分其他与信号有关的域被填入。每个信号都有一个位图sa_mask，它定义了当这个信号的处理程序正在执行时那些信号将被阻塞；每个信号也都有一个指针sa_handler，它包含指向信号处理程序的指针，或指出信号将被忽略或按缺省方式处理的特殊值。在处理程序结束时调用SIGRETURN的库例程的地址保存在mp_sigreturn中，这个地址是内存管理器收到的消息中的一个域。
POSIX允许进程修改自己的信号处理方式，甚至在信号处理过程中也是这样。这可以用来改变在处理一个信号的过程中进程对随后信号的响应，然后再恢复普通的响应。下一组系统调用支持这些信号操作特性。SIGPENDING由do_sigpending处理，它返回位图mp_sigpending，使进程能判断它是否有等待的信号；SIGPROCMASK由do_sigprocmask处理，它返回当前被阻塞的信号集合，它还可以以来改变集合中单个信号的状态或把整个集合替换成一个新的。一个信号被解除阻塞的时刻是一个很好的检查等待信号的时机，这是通过在17927和17933行调用check_pending完成的。do_sigsuspend（17949行）执行SIGSUSPEND系统调用，它挂起一个进程直到收到一个信号为止，与我们在这里讨论过的其他函数一样，它也要操作位图。它还设置mp_flags中的SIGSUSPENDED位，

这就是它为阻止进程运行所作的全部。同时这里也是调用check_pending的一个很好的时机。最后，do_return处理SIGRETURN，它用来从一个用户指定的处理过程中返回，它恢复在进入处理过程时就存在的信号上下文，然后在17980行再次调用check_pending。

有些信号是源自内核的，如SIGINT，这些信号的处理方式与用户进程调用kill产生的信号的处理方式类似。do_kill（17983行）和do_sig（17994行）这两个过程在概念上是类似的。他们都使内存管理器发出一个信号。对KILL的单个调用可能需要向一组进程发送信号，do_kill只是调用check_sig，由它在整个进程表中查找够资格的接收者。当内核的消息到达时将调用do_ksig。在消息中含有一个位图，使得内核可以在一条消息中产生多个信号。与KILL相同，这些信号中的每一个都可能需要传给一组进程。这个位图在18026到18048行的循环中被一次一位地处理。一些内核信号需要特别注意：在有些情况下进程ID被修改以便使信号传递给一组进程（18030行到18033行），如果没有请求过，SIGALRM将被忽略。除了这个例外，每个位集都将导致调用check_sig，这与do_kill中相同。

ALARM系统调用由do_alarm（18056行）控制。它调用下一个函数set_alarm发消息给时钟任务，让它启动计时器。把set_alarm（18067行）作为一个独立的函数是因为在进程结束而计时器仍然打开着时，还要用它关闭计时器。当计时器时间到时，内核通过向内存管理器发一个类型为KSIG的消息来宣布这个事件，正如上面所讨论的，这将使do_ksig运行。如果没有被捕获，SIGALRM的缺省动作是杀死进程。要捕获SIGALRM必须用SIGACTION安装一个处理过程。具有用户指定处理过程的SIGALRM信号完整的事件序列如图4-45所示，这里有三条消息序列。在消息(1)、(2)、(3)中，用户通过向内存管理器发消息执行一个ALARM调用，管理器向时钟发请求，时钟应答；在消息(4)、(5)、(6)中，时钟任务向内存管理器发送告警，内存管理器调用系统任务为执行信号处理过程准备用户进程的堆栈（与图4-42(b)中一样），系统任务应答；消息(7)调用SIGRETURN，它在信号处理过程执行结束时发生。作为回答，内存管理器向系统任务发送消息(8)让它完成清理工作，然后系统任务用消息(9)应答。消息(6)自己并不引起信号处理过程执行，但事件的序列将被保持，因为根据MINIX的优先级调度算法，系统任务作为一个任务将被允许完成它的工作，信号处理过程作为用户进程的一部分只有在系统任务完成它的工作后才会执行。

图4-45 与计时器有关的消息。其中最重要的有：(1)用户执行ALARM。(4)时间到信号被发出。(7)处理过程结束，调用SIGRETURN。详见文中描述。

do_pause处理PAUSE系统调用（18115行）。这里需要作的只是设置一个位并且不作应答，从而使调用者被阻塞，甚至都不需要通知内核，因为它知道调用者被阻塞了。

signal.c中处理的最后一个系统调用是REBOOT（18128行）。这个调用只被超级用户执行的专门程序使用，但它提供了一个重要的功能，它保证所有进程有序地结束，以及在调用内核中的系统任务关机之前，文件系统被同步。进程的终止是通过check_sig向除init之外的所有进程发SIGKILL完成的，这也是REBOOT被包含在这个文件中的原因。

前面我们提到过signal.c中的几个支持函数，现在我们将详细地讨论他们。到目前为止最重要的是sig_proc（18168行），它实际地发送一个信号。首先作一些检查，向死的（18190到18192行）或挂起的（18194到18196行）进程发送信号都是严重问题，会产生一个系统panic。正在被跟踪的进程在接到信号时将被停止（18198到18202行）。如果信号将被忽略，sig_proc的工作在18204行结束。这是一些信号的缺省动作，例如POSIX要求但MINIX不支持的一些信号。如果信号被阻塞，唯一需要作的是在进程的mp_sigpending位图中设置一个位。关键的测试（18213行）是区分已经允许捕获信号的进程和还没有允许的。到此为止，所有其他的特殊情况都已经消除，不能俘获信号的进程将结束。

能被捕获的信号在18214到18249行处理。首先构造一条消息发往内核，它的一些部分是进程表内存管理器部分中信息的拷贝。如果接收信号的进程先前被用SIGSUSPEND挂起，那么在挂起时被保存的信号掩码将被包含在消息中，否则当前进程的掩码将被包含（18213到18217行）。消息中包含的其他条目是接收信号进程空间中的几个地址：信号处理过程的地址、在处理过程结束时将被调用的sigreturn库例程的地址、和当前堆栈指针。

下一步是在进程的堆栈上分配空间。图4-46所示是将被放在堆栈上的结构。把sigcontext部分放在堆栈上保存是为了以后恢复，因为进程表中对应的结构在准备执行信号处理过程时将被修改。sigframe部分提供了信号处理过程的返回地址，和SIGRETURN在处理过程结束时完成进程状态恢复所需的数据。返回地址和堆栈指针实际上并没有被MINIX任何部分使用，他们的作用是在有人用调试器跟踪信号处理过程时愚弄调试器。

图4-46 为信号处理过程作准备而推入栈的sigcontext和sigframe结构。处理器的寄存器组是上下文切换时使用的栈框的一个拷贝。

将被放到接收信号进程的堆栈上的结构是相当大的。18225到18226行的代码为它保留空间，随后调用adjust检查进程堆栈上是否有足够的空间。如果没有，程序将用很少使用的goto语句（18228行到18229行）跳转到标号doterminate，然后杀死进程。

对adjust的调用有一个潜在的问题。在对BRK实现的讨论中我们提到过，如果堆栈与数据段的间隙长度小于SAFETY_BYTES，那么adjust将返回错误。为错误提供一个余量是因为合法性检查只能由软件间歇地进行，在当前的例子中因为处理信号所需的堆栈空间是确切知道的，所以这个错误余量可能是太多了，多出的空间只有信号处理过程需要，而它一般是一个较简单的函数。一些进程可能会因为对adjust的调用失败而毫无必要地终止，

这当然要比让进程在某些时候神秘地失败强，当然更加精细的调节也许是可能的。
如果堆栈上有足够的空间，另外两个标志将被检验。SA_NODEFER标志指出在处理信号时是否阻塞后来的相同类型的信号，SA_RESETHAND标志指出在接到这个信号时信号处理过程是否被复位。（这忠实地模拟了老式的signal调用，尽管这个“特征”经常被认为是老式调用的一个缺陷，支持老式的特征需要同时支持他们的缺陷）。随后用库例程sys_sendsig（18242行）通知内核。最后，指示有一个信号在等待的位被清除，unpause被调用以结束任何进程可能在上面挂起的系统调用。当接收信号的进程下一次运行时，信号处理过程将会执行。
下面我们看一看由标号doterminate（18250行）标记的结束代码，这个标号和goto语句是处理可能的adjust调用失败的最简单的方法。这里被处理的信号由于某种原因不能或不应该被捕获。如果对这个信号适合，处理动作可能包括生成一个core文件，最后总是通过调用mm_exit（18258行）使进程结束。
check_sig（18265行）是内存管理器检查是否有需要发送的信号的地方。调用
kill(0, sig);
使指定的信号被发给调用者组的所有进程（即所有从同一个终端启动的所有进程），源自内核的信号和REBOOT也可能影响多个进程。由于这个原因，check_sig在18288到18318行的循环中扫描整个进程表以找出所有应该接收信号的进程。循环中包括了大量的测试。信号只发给通过了所有测试的进程，发送通过在18315行调用sig_proc完成。
check_pending（18330行）是另一个在我们讨论过的代码中被多次调用的函数。它循环检查do_sigmask、do_sigreturn、do_sigsuspend引用的进程mp_sigpending位图中所有的位，查看被阻塞的信号中是否有已经不再阻塞的。它调用sig_proc发送找到的第一个不再阻塞的等待信号。因为所有信号处理过程最终都将引起do_sigreturn被执行，这样作足以保证把所有挂起的信号最终都发送出去。
过程unpause（18359行）处理发往挂起在PAUSE、WAIT、READ、WRITE或SIGSUSPEND调用上的进程的信号。PAUSE、WAIT和SIGSUSPEND可以通过查询进程表的内存管理器部分检查出来，如果都没有发现就必须让文件系统用它自己的do_unpause函数检查可能的在READ和WRITE上挂起的情况。在各种情况下动作都是相同的：向等待的调用发送一个错误回答，复位对应于进程等待原因的标志位，使进程能够恢复执行处理信号。
这个文件中最后一个过程是dump_core（18402行）。它把核心映像写到磁盘上。核心映像包括一个含有进程各段长度的头、从内核进程表中获得的这个进程的所有状态信息的一个拷贝、和各个段的内存映像。调试器能解释这些信息以帮助程序员确定在程序运行过程中什么发生了错误。写文件的代码是很直观的，但前面章节中提到过的潜在的问题又出现了，而且更难处理。为了保证记录到核心映像上的堆栈段是最新的，在18428行调用了adjust，这个调用可能会因为安全余量的原因失败。dump_core不检查这个调用是否成功，所以无论如何核心映像都将被写入，但是文件中有关堆栈的信息可能会不正确。
4.8.7 其他系统调用的实现
文件getset.c包含了一个过程do_getset（18515行），它执行余下的七个内存管理器调用，如图4-47所示。他们都是如此简单以至不值得为每个写一个单独的过程。GETUID和GETGID调用同时返回真实和有效用户号或组号。
系统调用
描述
GETUID
返回真实和有效用户号
GETGID
返回真实和有效组号
GETPID
返回进程和它父亲的进程号
SETUID
设置调用者的真实和有效用户号
SETGID
设置调用者的真实和有效组号
SETSID
创建新的会话，返回进程号
SETPGRP
返回进程组标识
图4-47 mm/getset.c中支持的系统调用
设置用户号和组号要比读他们稍微难一点，这里必须检查调用者是否有权限设置组号或用户号。如果调用者通过了检查，则必须把新的用户号或组号通知文件系统，因为文件保护依赖于他们。SETSID调用创建一个新的会话，已经是进程组组长的进程不允许执行这个操作，18561行的测试完成这个检查。文件系统完成把一个进程变成没有控制终端的进程组组长的操作。
命令
描述

T_STOP
停止进程
T_OK
允许这个进程被父进程跟踪
T_GETINS
从正文（指令）空间返回值
T_GETDATA
从数据空间返回值
T_GETUSER
从用户进程表返回值
T_SETINS
在正文（指令）空间设置值
T_SETDATA
在数据空间设置值
T_SETUSER
在用户进程表中设置值
T_RESUME
恢复执行
T_EXIT
退出
T_STEP
设置跟踪位
图 4-48 mm/trace.c中支持的调试命令
PTRACE系统调用提供的最小的调试支持包含在文件trace.c中。共有十一个命令可以作为参数传递给PTRACE系统调用，如图4-48所示。在内存管理器中，do_trace处理其中的四个：enable、exit、resume、step。允许和退出跟踪的请求在这里完成，其他的命令都被传递给系统任务，它能够操作进程表的内核部分。这是通过在18669行调用库函数sys_trace完成的。在trace.c尾部定义了两个支持函数，stop_proc用来在被跟踪的进程收到信号时停止它，findproc通过在进程表中搜索要被跟踪的进程支持do_trace。
4.8.8 内存管理器工具
余下的文件中包含了工具例程和表格。文件alloc.c是系统对内存的那些部分在被使用那些部分空闲保持跟踪的地方，它有三个入口：
1. alloc_mem - 请求一块给定大小的内存。
2. free_mem - 归还不再需要的内存
3. max_hole - 计算最大可用空洞的长度
4. mem_init - 在内存管理器开始运行时初始化空闲表。
正如我们前面说过的，alloc_mem（18840行）在按照内存地址排列的空洞表中使用首次适配算法查找。如果找到的块太大，它就取出它需要的，把其他的留在空闲表上并从其原长度中减掉取走的。如果整个空洞都需要，它将调用del_slot（18926行）从空闲表中删除这个空洞的入口。
free_mem的工作是检查一块新释放的内存能否与任何一方的邻居合并，如果能就调用merge（18949行）合并空洞并更新空闲表。
max_hole（18985行）扫描空洞表并返回它找到的最大的条目。mem_init（19005行）构造由所有可用内存组成的初始空闲表。
下一个文件是utility.c，它包含了在内存管理器不同部分的几个杂过程。过程allowed（19120行）检查一个对文件的操作是否允许。比如do_exec需要知道文件是否是可执行的。
过程no_sys（19161行）应该永远不被调用，提供它只是为了处理用户用非法的或不是由内存管理器处理的系统调用号调用内存管理器的情况。
panic（19172行）只有在内存管理器检测到一个它无法恢复的严重错误时才会被调用。它向系统任务报告错误，系统任务紧急停止系统。它不该被轻易调用。
utility.c中最后一个是tell_fs，当内存管理器处理的事件需要通知文件系统时，它构造一条消息并发给文件系统。
尽管与前面的有很大不同，文件putk.c中的两个过程也是工具过程。对printf的调用被一次又一次插入到内存管理器中，主要是为了调试，panic也要调用printf。前面已经说过，printf实际上是一个宏，它被定义为printk，所以对printf的调用不使用把消息送到文件系统的标准I/O库过程。printk调用putk直接与终端任务通讯，这种操作对普通用户是禁止的。在内核代码中我们也见到过一个同名的例程。
4.9 小结

在本章中讨论了一般的和MINIX的内存管理。我们看到最简单的系统根本不用交换和分页，一个程序一旦装入内存就一直保持在那个地方直到结束为止。有些操作系统同一时刻只允许一个进程在内存中，另外一些支持多道程序。

再向前一步是交换。在使用了交换以后，系统可以处理比内存所能容纳的更多的进程，得不到空间的进程将被换出到磁盘上。内存和磁盘上的空闲空间可以用位图或空洞表跟踪。

更先进的计算机通常都有某种形式的虚拟存储器。在最简单的情况下，每个进程的地址空间都被划分为同样大小的称为页的块，它可以被放到内存中任何可用的页框中去。有许多页面替换算法，最著名的两个是第二次机会和老化算法。为了使分页系统很好地工作，仅仅选择一个好的算法是不够的，还需注意确定工作集、内存分配策略、页面大小等问题。

分段有助于处理在运行中要改变大小的数据结构、简化链接和共享，它还有助于为不同的段提供不同的保护。有时分段和分页结合起来构成两维的虚拟存储器，MULTICS系统和Intel的Pentium支持分段和分页。

MINIX的内存管理是很简单的。内存在进程执行FORK或EXEC系统调用时被分配。只要进程还存在，这样分配的内存就永远不会增长或减小。在Intel处理器上MINIX使用的内存模型有两种，小程序可以把指令和数据放到同一个内存段中，大程序使用独立的指令和数据空间（独立的I和D）。具有独立I和D空间的进程可以共享他们内存中的正文部分，所以在执行FORK时只有数据和堆栈的内存是必须分配的。这一点在执行EXEC时也可能成立，其条件是有另外一个进程已经在使用EXEC所需要的正文了。

内存管理器的大部分工作与使用一个空洞表和首次适配算法的跟踪空闲内存的工作没有关系，却是执行与内存管理有关的系统调用。一些系统调用支持POSIX风格的信号，因为大部分信号的缺省动作是结束接受信号的进程，所以在内存管理器中处理他们是比较合适的。所有进程的终止都由它发起。几个与内存没有直接关系的系统调用也由内存管理器处理，因为内存管理器比文件系统小，所以把他们放在这里是最方便的。

习　题

1. 一个计算机系统有足够的空间在它的主存中放四个程序，这些程序有一半的时间在空闲等待I/O操作，多大比例的CPU时间被浪费掉了？

2. 在一个使用交换的系统中，按地址排列的内存中的空洞大小是：1OK，4K，2OK，18K，7K，9K，12K，和15K。对于连续的段请求：

(a) 12K

(b) 1OK

(c) 9K

使用首次适配算法那个空洞将被取出？对最佳适配、最差适配、下次适配回答同样的问题。

3. 物理地址和虚地址之间的区别是什么？

4. 使用图4-8的页表，指出对应于下列虚地址的物理地址：

(a) 20

(b) 4100

(c) 8300

5. Intel 8086处理器不支持虚拟存储器，然而一些公司曾经出售过含有没作任何改动的8086 CPU的分页系统。对他们是如何做到这一点的做一个合理的猜想。（提示：想一想MMU的逻辑位置）

6. 如果一条指令需要1微秒，一个页面故障需要另外n微秒，给出在页面故障每k条指令发生一次时指令的实际执行时间。

7. 一个机器有32位地址空间和8K页面，页表完全用硬件实现，每个入口一个32位字。在进程启动时，页表被以每个字100纳秒的速度从内存拷贝到硬件中。如果每个进程运行100毫秒（包含装入页表的时间），多少比例的CPU时间被用来装入页表？

8. 一个32位地址的计算机使用两级页表，虚地址被分为9位的顶级页表域，11位的二级页表域和偏移，页面长度是多少？在地址空间中一共有多少个个页？

9. 以下是一小段用于一个页面长度512字节的计算机的汇编语言程序。程序位于地址1020，堆栈指针位于8192（堆栈向0增长）。给出这个程序产生的页面引用序列。每个指令占用4个字节（1个字），对指令和数据的引用都应该被包括在引用序列中

```
load word 6144 into register 0
push register 0 onto stack
call procedure at 5120, stacking the return address
substract the immediate constant 16 from stack pointer
compare the actual parameter to immediate constant 4
jump if equal to 5152
```

10. 假设一个32位虚地址被分成a、b、c、d四个域，前三个用于一个三级页表系统，第四个域d是偏移。页面数与这四个域的大小都有关系吗？如果不是，与那些有关、那些无关？

11. 一台计算机的处理器的地址空间有1024个页面，页表保存在内存中。从页表中读取一个字的开销是500纳秒。

为了减小开销，这个计算机使用了TLB，它有32个（虚地址物理地址）对，能在100纳秒内完成查找。把平均开销降到200纳秒需要的命中率是多少？
12. VAX上的TLB没有R位，为什么？
13. 一台机器有48位虚地址和32位物理地址，页面是8K，在页表中需要多少个入口？
14. 一台计算机有4个页框，装入时间、上次访问时间、和每个页的R和M位如下所示（时间以时钟滴答为单位）：

页	装入	上次引用	R	M
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

(a) NRU将替换那个页？
(b) FIFO将替换那个页？
(c) LRU将替换那个页？
(d) 第二次机会将替换那个页？
15. 如果FIFO页面替换算法被用到4个页框和8个页面上，当页框初始为空，引用序列为0172327103时会发生多少次页面故障？如果使用LRU呢？
16. 一台小型计算机有四个页框。在第一个时钟周期时R位是0111（页0是0，其他是1）。在随后的时钟周期中这个值是1011、1010、1101、0010、1010、1100、0001。如果使用带有8位计数器的老化算法，给出最后一个周期后四个计数器的值。
17. 把一个64K的程序从平均寻道时间30毫秒、旋转时间20毫秒、每道32K的磁盘上装入，这需要多长时间？
(a) 在页长为2K时
(b) 在页长为2K时
页随机地分布在磁盘上。
18. PDP-1是最早的分时计算机之一，有4K 18位字的内存。在每个时刻它在内存中保持一个进程。当调度程序决定运行另一个进程时，在内存中的进程将被写到一个换页鼓上，鼓的表面有4K个18位字。鼓可以从任何地址开始读写，你认为为什么要选这个鼓？
19. 一台计算机为每个进程提供65536字节的地址空间，划分为4K的字节的页。一个特定的程序有32768字节的正文、16386字节的数据、和15870字节的堆栈。这个程序能装入地址空间吗？如果页面长度是512字节，能放下吗？别忘了一个页不能同时包含两个不同段的成分。
20. 人们已经观察到在两次页面故障之间执行的指令数与分配给程序的页框数直接成比例。如果可用内存加倍，页故障间的平均间隔也加倍。假设一条普通指令需要1微秒，但是如果发生了页面故障就需要2001微秒（即两毫秒用于处理故障）。如果一个程序运行了60秒，期间发生了15000次页面故障，如果可用内存是原来的两倍这个程序运行需要 多长时间？
21. Frugal计算机公司的一组操作系统设计员正在思考在他们的新操作系统中减少对后备存储器数量的需求的方法，有人建议根本不要把程序正文保存在交换区中，而是在需要的时候直接从二进制文件中调页进来，这种方法有什么问题吗？
22. 解释内零头和外零头的区别。那个发生在分页系统中？那个发生在使用纯的分段的系统中？
23. 当分段和分页同时使用时，就象在MULTICS中那样，首先必须查找段描述符，然后是页描述符。TLB也是这样以两级查找的方式工作的吗？
24. 为什么在MINIX的内存管理模式下必须有一个象chmem这样的程序？
25. 修改MINIX使得僵死进程一旦进入僵死状态就释放它的内存，而不是一直等到它的父进程执行wait。
26. 在当前MINIX的实现中，当执行EXEC系统调用时，内存管理器检查当前是否有足够容纳新的内存映像的空洞，如果没有就拒绝这个调用。一个更好的算法会看在当前的内存映像被释放后是否有足够大的空洞，实现这个算法。
27. 当执行一条EXEC系统调用时，MINIX使用了一条技巧使文件系统立刻读入整个段。设计并实现一个类似的技巧以允许用相似的方式来写核心转储（core dump）。
28. 修改MINIX以实现交换。
29. 4.7.5节中指出对于一条EXEC调用，通过在释放当前进程的内存之前测试足够的“空洞”可以得到一个次最优的实现。请重新对该算法编程以获得更好的性能。
30. 4.8.4节指出最好分别对正文段和数据段搜索“空洞”，请实现这一改进。
31. 请重新设计adjust，以避免被发送信号的进程由于过于严格的堆栈空间检查而被不必要地撤销的问题。

第五章　文件系统

所有的计算机应用程序都要存储信息和检索信息。进程在运行时可以在自己的地址空间中存储一定量的信息。但其存储容量只限于虚拟地址空间的大小，对有些应用程序，这已经足够了；但对于其他的应用程序，例如飞机定票系统、银行系统或者公司记录保存系统，这个存储空间又显得太小了。

在进程的地址空间内保存信息的第二个问题是：当进程终止时，信息随之丢失。对许多应用（例如数据库）而言，它们的信息必须保存几周、几个月、甚至一直保留。信息随着使用进程的终止而丢失是不可接受的。此外，即使系统突然崩溃，这些信息也应该设法保存下来。

第三个问题在于经常有多个进程同时存取信息（或者部分信息）。如果我们在某个进程的地址空间内存有在线电话簿，只有那个进程可以对它进行存取。解决这个问题的方法是使信息本身独立于任何进程。

由此，对于长期的信息存储，我们有以下三个基本要求：

1. 必须能够存储大量的信息。
2. 在使用信息的进程终止时，信息必须保存下来。
3. 多个进程可以并发地存取信息。

解决所有这些问题的常用方法是把信息以一种单元，即通常所说的文件（file）的形式存储在磁盘或其他外部介质上。然后，在需要时进程可以读取这些信息或者写入新的信息。存储在文件中的信息必须是永久性的，也就是说，它不会因进程的创建和终止而受到影响。只有当用户显式地删除它时，文件才会消失。

文件是通过操作系统来管理的。文件的结构以及命名、存取、使用、保护和实现方法都是操作系统设计的主要内容。总体上，操作系统中处理文件的那部分被称为文件系统（file system）。文件系统是本章的主要论题。

从用户的观点来看，文件系统中最重要的方面是文件系统如何呈现在他们面前。即一个文件由什么组成、文件如何命名、如何保护文件，以及对文件可以进行哪些操作等等。而一些细节，比如是用链接表还是用位图来记录空闲存储区，以及在一个逻辑块中有多少个扇区等则无关紧要，虽然这些方面对文件系统设计者相当重要。因此我们把本章分成几节，前两节分别讲述了文件和目录的用户界面，接下来是有关文件系统实现的详细讨论，然后研究文件系统中的安全和保护机制。最后，我们介绍MINIX的文件系统。

5.1 文件

这一节，我们从用户角度来研究文件，即文件是怎么使用的，它有些什么特性。

5.1.1 文件命名

文件是一个抽象机制，它提供了一种把信息保存在磁盘上而且便于以后读取的方法。它必须这样来实现，使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。

抽象机制最重要的特征或许是被管理对象的命名方法。因此，我们从文件命名开始讨论文件系统。在创建一个文件时，进程给出文件名。进程终止后，文件仍然存在，其他进程使用该文件名可以对它进行存取。

各种系统的文件命名规则略有不同，但所有操作系统都允许用一到八个字母组成的字符串作为合法文件名。因而，andrea、bruce和cathy都可以用作文件名。数字和一些特殊字符也可用于文件名之中，所以象2、urgent!和Fig.2-14也经常用作文件名。许多文件系统支持长达255个字符的文件名。

有些文件系统区分大小写，而另一些则不加区分。UNIX属于前一类，MS-DOS属于后者。因此在UNIX系统中，barbara、Barbara、BARBARA、BARbara和BarBaRa等是不同的文件，而在MS-DOS中，他们都表示同一个文件。

许多操作系统支持两部分文件名，两部分之间用句号加以分隔，比如prog.c。在句号后面的部分称作文件扩展名(file extension)，它通常给出了与文件有关的一些信息。在MS-DOS中文件名由1-8个字符和1-3个字符的可选扩展名组成。在UNIX中，如果使用扩展名，则其长度完全由用户决定，甚至一个文件之中可以含两个或多个部分的扩展名。例如，prog.c.Z中，.Z通常表明文件prog.c已经使用Ziv_Lempel压缩算法压缩过。一些常用的文件扩展名及他们的含义如图5-1。

图5-1 一些典型的文件扩展名。

在某些时候，文件扩展名仅仅是一种惯例，并不强迫使用。以file.txt命名的文件可能是文本文件，但这个文件名也主要用于提醒用户，而并不是给计算机传送特别的信息。然而，C编译器可能要求供其编译的文件以.c结尾，否则它将拒绝编译。

当某个程序可以处理几种不同的文件时，这种惯例特别有用。比如C编译器，可以编译、链接多种文件，其中有些是C文件，有些是汇编语言文件。这时扩展名显得很有必要，利用它编译器能够区分哪些是C文件，哪些是汇编文件，哪些是其他文件。

5.1.2 文件结构

文件可以按几种不同方式构成。图5-2给出了三种常用的方式。图5-2(a)中的文件是一个无结构字节序列。事实上，操作系统不知道也不关心文件中有些什么内容，它所见到的都是字节。任何含义都只能加在用户级程序中。UNIX和MS-DOS均使用这一方法。除此之外，WINDOWS 95基本上使用了MS-DOS的文件系统，只是做了一些语法方面的改进(例如长文件名)，因此本章中关于MS-DOS的论述几乎都适用于Windows 95。可是，WINDOWS NT的文件系统完全不同。

图5-2 三种文件。 (a)字节序列。 (b)记录序列。 (c)树。

操作系统把文件看成字节序列提供了很大的灵活性。用户程序可以在文件中加入任何内容，并且以任何方便的形式来命名。这时，处理文件结构的任务由用户完成，操作系统无法提供任何帮助，但也不会设置障碍。对于需要做特殊操作的用户来说，下面的文件结构是非常重要的。

在字节序列结构上的第一步改进见图5-2(b)。在这个模型中，文件是一个固定长度记录的序列，每条记录

都有内部结构。把文件作为记录序列的中心思想是：读操作返回一条记录，而写操作重写或追加一条记录。在过去几年中，当80列的穿孔卡片还在广泛使用的时候，许多操作系统把他们的文件系统建立在由80个字符的记录组成的文件基础之上。这些系统也支持由132个字符的记录组成的文件，这种文件用于行式打印机(当时是132列的行式打印机)。程序以80个字符为单位读取数据，以132个字符为单位输出。当然，最后52个字符有可能都是空格。

CP/M就是一个把文件看作固定长度记录序列的(老式)系统，它使用长度为128个字符的记录。虽然，把文件视作为固定长度的记录序列一度曾成为标准，现在这种想法已经过时了。

文件结构的第三种组成方式如图5-2(c)所示。这种方式中，文件由一棵记录树构成，并非每条记录都具有同样的长度。在记录的固定位置包含一个关键字域。记录树按关键字域进行排序，这便于对特定关键字进行快速查找。

在这种文件结构中，基本操作并不是取"下一条"记录，而是获取具有特定关键字的记录，虽然取"下一条"记录也是可以的。例如就图5-2(c)中的文件zoo而言，用户可以要求系统取关键字为pony的记录，而不必关心记录在文件中的确切位置。此外，在文件中能够添加新记录。这时，由操作系统而不是用户来决定把记录放在文件的什么位置。这种文件结构和UNIX与MS-DOS中使用的无结构字节流显然不同，它广泛使用在一些用于商业数据处理的大型计算机之中。

5.1.3 文件类型

许多操作系统支持几种类型的文件。例如，UNIX和MS-DOS中都有正规文件和目录。UNIX还有字符设备文件和块设备文件。正规文件(regular file)中包含有用户信息。图5-2中的所有文件都是正规文件。目录(directory)是管理文件系统结构的系统文件。我们把目录放在以后研究。字符设备文件(character special file)和输入/输出有关，用于模仿串行I/O设备。例如终端、打印机、网络等等。块设备文件(block special file)则用于模仿磁盘。本章我们主要讨论正规文件。

一般来说，正规文件是ASCII文件或者二进制文件。ASCII文件由多行正文组成。在某些系统中，每行用回车符结束。其他系统则用到换行符。有时，回车符和换行符同时使用。各行不一定要有同样的长度。

ASCII文件的最大优点是可以原样地显示和打印，也可以用通常的文本编辑器进行编辑。此外，如果许多程序都以ASCII文件作为输入和输出，那么我们就很容易把一个程序的输出作为另一个程序的输入，如同shell管道一样。(管道方式来实现进程间通信实现起来并非更容易，但如果采用一种公认的标准来表示它的话，对信息的解释将会更容易，比如用ASCII码。)

除ASCII文件外，另一种正规文件是二进制文件，把二进制文件印在打印机上得到的是一张晦涩难懂的表，里面全是些乱七八糟的字符。二进制文件往往具有一定的内部结构。

例如，在图5-3(a)中，可以看到一个简单的可执行的二进制文件，它取自一个早期版本的UNIX。虽然技术上，这个文件只是一个字节序列，但只有在文件有正确格式时，操作系统才会执行这个文件。这种文件有五段：文件头、正文、数据、重定位位和符号表。文件头以所谓的魔数(magic number)开始，表明该文件是一个可执行文件(防止非此格式文件的偶然运行)。接下来是一些16位的整数，给出了文件不同部分的长度、执行的起始地址和一些标志位。在文件头之后是程序本身的正文和数据，在他们装载到内存中时，使用重定位位进行定位。符号表则用于调试。

图5-3 (a)可执行文件。 (b)存档文件。

二进制文件的第二个例子是一个UNIX存档文件，它由许多编译过但还没有链接的库过程(模块)组成。每个模块以模块头开始，其中给出了模块名、创建日期、拥有者、保护代码和模块长度等等。如同可执行文件一样，模块头中也充斥着二进制数字，把他们在打印机上输出同样毫无意义。

所有的操作系统都必须识别一种文件类型，即他们自己的可执行文件。有些操作系统则可以识别多种文件类型。老式TOPS-20系统甚至可以检查运行文件的创建时间，然后找到相应的源文件，看它在二进制文件生成之后是否被修改过。如果修改过，操作系统自动重新编译这个源文件。按照UNIX的用语，在shell中嵌入了make程序。这时，操作系统要求用户使用强制的文件扩展名，以便确定二进制程序是由哪个源文件生成的。

类似地，当WINDOWS用户双击文件名时，以该文件作为参数，操作系统调用一个合适的程序运行。根据文件扩展名，操作系统可以决定运行哪个程序。

如果用户执行系统设计者未曾预料的操作，这样的强类型文件可能会引起麻烦。举个例子，假设在一个系统中，程序输出文件为dat类型(数据文件)，如果用户编写一个程序格式化器，读取.pas文件，进行转换(例如把该文件转换成标准的行首缩进风格)，再把转换后的文件输出，输出文件具有.dat类型。如果用户想用Pascal编译器来编译这个文件，但因为文件扩展名不符，Pascal编译器将拒绝编译。试图把file.dat拷贝到file.pas也是徒劳，系统会认为这种拷贝是无效的(防止用户误操作)。

尽管这种"用户友好性"对初学者有利，却使得一些有经验的用户大伤脑筋。他们不得不花很大的精力来适应操作系统对合理操作和不合理操作的区分。

5.1.4 文件存取

早期的操作系统只提供了一种文件存取方式：顺序存取(sequential access)。在这些系统中，进程可以从文件开始处顺序读取文件中所有字节或者记录，但不能够略过某些内容，也不能够非顺序读取。顺序存取文

件可以重绕，只要需要，可以多次读取该文件。当存储媒体是磁带，而不是磁盘时，用顺序存取文件是非常方便的。

用磁盘存储文件后，我们可以非顺序地读取文件中的字节或记录，或者根据关键字而不是位置来存取记录。能够以任何顺序读取的文件叫做随机存取文件(random access file)。

对许多应用程序来说，随机存取文件是必不可少的，例如数据库系统。如果一个飞机乘客打来电话，想要预订特定航班的机票。订票程序必须能够直接存取该航班的记录，而不必首先读出成千上万条其他航班的记录。

有两种方法指明从哪儿开始读取文件。第一种是，每次READ操作都给出文件中开始读的位置。另一种方法是，提供一个特殊的SEEK操作来设置当前位置，此后，从这个新的当前位置开始顺序地读取文件。

有些早期的主机操作系统中，文件在创建时，就指定为是顺序存取文件或者随机存取文件。对这两类文件，系统使用不同的存储技术。现代操作系统则不加区分，创建后，所有文件自动成为随机存取文件。

5.1.5 文件属性

每个文件都有文件名和数据。此外，所有操作系统还给文件赋以其他信息，比如，文件创建日期、文件长度等等。我们把这些额外的项称为文件属性(attribute)。不同系统的属性差别很大。图5-4的表中列出了一些可能的属性，但其他的属性也存在。没有一个系统具有全部这些属性，然而，每种属性都在某个系统中使用。

图5-4 一些可能的文件属性。

开始四个属性与文件保护有关，它给出了谁可以存取这个文件，谁不能存取这个文件。存在着各种不同的文件保护方案，其中的一些我们会在以后讨论。在一些系统中，用户必须给出口令才能存取文件，这时，口令也是文件属性之一。

标志是一些位或者短域，用来抑制或者允许某些特定性质。例如隐藏文件使其不出现在文件的显示列表中。存档标志位记录文件是否备份过。备份程序清除该标志位，在文件修改后，操作系统设置存档标志位。这样，备份程序可以区分哪些文件需要备份。临时标志位表示在创建该文件的进程终止后，它被自动删除。

记录长度、关键字位置和关键字长度等域只出现在那些能够用关键字查找记录的文件之中，他们提供了查找关键字所需信息。

各个时间域记录了文件的创建时间、最近存取时间以及最近修改时间等等。他们可用于不同目的。例如，在相应的目标文件生成后修改过的源文件需要重新编译，这些域提供了必要的信息。

当前长度域给出了文件当前的大小。在一些主机操作系统中，文件创建时，需要指明文件最大长度，以便操作系统事先保留一定的存储空间。工作站和个人计算机的操作系统则无需指明这一点。

5.1.6 文件操作

文件用于存储信息便于以后检索。不同系统提供了不同的操作进行存储和检索。下面是一些与文件有关的最常用的系统调用：

1. CREATE。创建没有任何数据的文件。该调用的目的是声明文件存在，并且设置一些属性。

2. DELETE。当文件不再需要时，必须删除它以释放磁盘空间。DELETE系统调用可用于实现这一目的。

3. OPEN。在使用文件之前，进程必须打开文件。OPEN调用的目的是：将文件属性和磁盘地址表载入主存，便于以后系统调用的快速存取。

4. CLOSE。当存取结束后，文件属性和磁盘地址就不再需要了，这时应该关闭文件以释放内部表空间。许多系统限制进程的打开文件数，鼓励用户关闭不再使用的文件。磁盘以块为单位写入，关闭文件可以迫使文件最后一块写回磁盘，尽管这一块可能还没有写满。

5. READ。从文件中读取数据。一般，读出的数据来自当前位置。调用者必须指明需要读取多少数据，并且提供存放这些数据的缓冲区。

6. WRITE。向文件中写入数据，写操作一般也是从当前位置开始。如果当前位置是文件末尾，文件长度增加。如果当前位置在文件中间，则现有数据被重写，并且永远丢失了。

7. APPEND。该调用是WRITE的限制形式，它只能在文件末尾添加数据。只提供最小系统调用集的系统通常没有APPEND。许多系统对同一操作提供了多种实现方法，这些系统中往往有APPEND调用。

8. SEEK。对于随机存取文件，需要指定从哪儿开始读写数据，常用的方法是用SEEK系统调用把当前位置指针指向文件中特定位置。此后，从该位置开始读写数据。

9. GET ATTRIBUTES。进程往往需要读取文件属性，例如，UNIX中make程序常用于管理由多个源文件组成的软件开发项目。在调用make时，检查所有源文件和目标文件的修改时间，安排最小数目的编译，使得所有文件都为最新版本。为实现该目的，需要查找文件的某些属性，如修改时间。

10. SET ATTRIBUTES。有些属性是用户可以设置的，在文件创建之后，能够修改他们。SET ATTRIBUTES系统调用使之成为可能。保护模式信息是一个很明显的例子，大多数标志亦属此类。

11. RENAME。用户常常要改变现有文件的文件名。系统调用RENAME可以实现这一目的。严格说来，这个调用并非必要，因为我们可以先把文件拷贝到新文件名的文件中，然后删除原来的文件。

5.2 目录

为了记录文件，文件系统通常有目录。在许多系统中，目录本身也是文件。本节讨论目录、它的组成、特性和可以进行的操作。

5.2.1 层次目录系统

目录通常包含有许多目录项，每个目录项代表一个文件。图5-5(a)中是一种可能情况，其中每个目录项包含文件名、文件属性和文件数据在磁盘上的地址等等。另一种情况见图5-5(b)，这里，目录项中含有文件名和指向另一个数据结构的指针，文件属性和磁盘地址就放在这个数据结构之中。以上两种系统都得到了广泛的应用。

图5-5 (a)属性放在目录项中。 (b)属性放在其他地方。

在打开文件时，操作系统查找目录，直到找到要打开文件的文件名。然后从目录项或者所指向的数据结构中取得文件属性和磁盘地址，放入内存的相应表中。之后，对该文件的所有引用均使用主存中的信息。

每个系统的目录数目各不相同。最简单的设计方案是维护一个单独的目录，其中包含所有用户的全部文件，如图5-6(a)。如果有很多用户，他们使用相同的文件名(如mail和games)，这时会出现冲突和混乱，使得系统无法正常工作。这一系统模型用在第一代微机操作系统中，而现在已经很少见到了。

图5-6 三种文件系统设计。 (a)所有用户共享一个目录。(b)每个用户拥有一个目录。 (c)每个用户有一棵任意树。图中字母表示目录或文件拥有者。

对于整个系统中使用单独一个目录管理所有文件的想法的改进是，每个用户拥有一个目录[参见图5-6(b)]。这种设计消除了不同用户之间的文件名冲突，但仍然难以使那些有许多文件的用户感到满意。用户常常需要把他们的文件按某种逻辑方式组织起来。例如，一个教授，可能有许多文件一起构成了他正在为某门课程编写的一本书，另一些文件包含了学生交上来的另一门课程的程序，此外，他的第三组文件是他正在构造的高级编译程序编写系统的代码，第四组文件则包含有资助申请表，同时，他还有一些其他的文件，如电子邮件、会议记录、论文、游戏等等。这时，需要某种方法，使得用户可以按自己选择的方式把文件组织起来。

我们需要的是一般的层次结构(即目录树)。使用层次结构，每个用户可以拥有所需的多个目录，以便自然地组织他们的文件。这种方法如图5-6(c)所示。图中，根目录包含有目录A、B、C，他们分别属于不同的用户，其中有两个用户为他们正在做的项目创建了子目录。

5.2.2 路径名

使用目录树来组织文件系统时，需要某种方法指明文件名。通常用到的方法有两种。第一种是，每个文件都赋予一个绝对路径名(absolute path name)，它由从根目录到文件的路径组成。例如，路径/usr/ast/mailbox表示根目录中含有子目录usr，而usr中又包含子目录ast，文件mailbox就放在目录ast下。绝对路径名总是从根目录开始，并且是唯一的。在UNIX中，路径各部分之间用"/"分隔。在MS-DOS中，分隔符是"\"。在MULTICS中是">"。不管使用哪个分隔符，如果路径名的第一个字符是分隔符，那么这个路径就是绝对路径。

另一种文件名是相对路径名(relative path name)。它常和工作目录(也称作当前目录)的概念一起使用。用户可以指定一个目录作为当前的工作目录。这时，所有的路径名，如果不是从根目录开始，都是相对于工作目录的。例如，如果当前的工作目录是/usr/ast，则绝对路径名为/usr/ast/mailbox的文件可以简单地用mailbox来引用。换句话说，如果工作目录是/usr/ast，则UNIX命令

```
cp  /usr/ast/mailbox  /usr/ast/mailbox.bak
```

和

```
cp  mailbox  mailbox.bak
```

具有相同的含义。相对路径往往更加方便，但是，它实现的功能和绝对路径完全一样。

有些程序需要存取某个特定文件，而不管当前的工作目录是什么。这种情况下，应该使用绝对路径名。比如，一个拼写检查程序要读取文件/use/lib/dictionary，而它不知道当前的工作目录，就须使用完整的绝对路径名。无论当前的工作目录是什么，绝对路径名总是可以使用的。

当然，如果这个拼写检查程序需要从目录/usr/lib中读取很多文件，它可以采用另一种方法，即执行一个系统调用，把工作目录切换到/usr/lib，然后只需用dictionary作为open的第一个参数。通过显式地改变工作目录，程序可以知道它在目录树中的确切位置，因此可以使用相对路径名。

大多数系统中，每个进程都有自己的工作目录，因而，在进程改变它的工作目录并退出后，其他的进程不会受到影响，在文件系统中也不会留下改变的痕迹。对进程来说，工作目录的切换是安全的，因此只要需要，它就可以改变当前的工作目录。可是，如果库过程改变了工作目录，在结束前不改为原来的目录，那么程序其他部分可能无法正常运行，因为他们关于当前目录的假设已经无效了。所以，在库过程中，很少改变工作目录，如果非改不可的话，总是在返回之前改回到原来的工作目录。

大多数支持层次目录结构的操作系统，在每个目录中有两个特殊的目录项"."和"..", 通常读作"dot"和"dotdot"。dot指当前目录，dotdot指其父目录。要了解他们是如何使用的，我们可以考虑图5-7中的UNIX文件树。某个进程的工作目录是/usr/ast，它可以使用".."沿树向上到达其父目录/usr。例如，它可以使用shell命令

```
cp  ../lib/dictionary .
```

把文件/usr/lib/dictionary拷贝到自己的目录下。第一个路径告诉系统上溯(到usr目录)，然后向下到达lib目录，找到dictionary文件。

图5-7 一棵UNIX目录树。

第二个参数指定为当前目录。当cp命令用一个目录名(包括".")作为它的第二个参数时，它把所有的文件拷贝到该目录中。当然，对于上述拷贝，更普遍的方法是键入

```
cp  /usr/lib/dictionary  .
```

这里，用户使用"."避免了第二次键入dictionary。

5.2.3 目录操作

相对于文件的系统调用而言，各个系统中用于管理目录的系统调用差别更大。为了让读者对这些系统调用及其工作方式有一个印象，我们下面将给出一个例子(取自UNIX)。

1. CREATE。创建目录。除了目录项"."和".."之外，目录内容为空。目录项"."和".."是系统自动放在目录中的(有时通过mkdir程序)。

2. DELETE。删除目录。只有空目录可以被删除。只含有目录项"."和".."的目录都认为是空目录，这两个目录项是不能被删除的。

3. OPENDIR。目录内容可被读取。例如，为了列出目录中的所有文件，列表程序必须先打开该目录，然后读取其中所有文件的文件名。同打开和读取文件一样，在读目录之前，必须打开目录。

4. CLOSEDIR。读目录结束后，应该关闭该目录以释放内部表空间。

5. READDIR。系统调用READDIR返回打开目录的下一个目录项。以前我们也使用通常的READ系统调用来读目录，但这种方法有一个缺点：程序员必须了解目录的内部结构。相反，不管使用哪一种目录结构，READDIR总是以标准格式返回一个目录项。

6. RENAME。在很多方面，目录和文件相似。文件可以换名，目录亦然。

7. LINK。链接技术允许文件出现在多个目录中。这个系统调用指定一个存在的文件和一个路径名，并建立从文件到路径所指定的名字的链接。这样，同一文件可以在多个目录中出现。

8. UNLINK。删除目录项。如果被解链的文件只出现在一个目录中(正常情况)，它从文件系统中被删除。如果它出现在多个目录中，只删除指定的路径名，其他路径名依然保留下来。在UNIX中，删除文件的系统调用(前面已有论述)实际上就是UNLINK。

以上列出了最主要的系统调用。但还有一些其他调用，比如管理与目录相关的保护信息的系统调用。

5.3 文件系统的实现

现在我们从用户角度转到实现者角度来研究文件系统。用户关心的是文件是如何命名的、可以进行哪些操作、目录树是什么样的以及类似的界面问题。而实现者感兴趣的是文件和目录是如何存储的、磁盘空间是怎样管理的以及如何使系统有效而可靠地工作等等。在下面几节中，我们将研究文件系统的实现中出现的一些问题以及如何来解决这些问题。

5.3.1 实现文件

或许在实现文件存储中最重要的问题是记录各个文件分别用到哪些磁盘块。不同操作系统采用不同的方法。这一节，我们将讨论其中一些方法。

连续分配

最简单的分配方案是把每个文件作为连续数据块存储在磁盘上。因此，在具有1K大小块的磁盘上，50K的文件要分配50个连续的块。这一分配方案有两大优点。首先，简单、容易实现，记录每个文件用到的磁盘块仅需记住一个数字即可，也就是第一块的磁盘地址。其次，性能较好，在一次操作中，就可以从磁盘上读出整个文件。

不幸的是，连续分配方案也有两个很大的不足。首先，除非在文件创建时就知道了文件的最大长度，否则这一方案是行不通的。不知道文件的最大长度，操作系统也就无法确定要保留多少磁盘空间。但是，在那些文件数据一次性写入的系统中，连续分配的优点可以得到充分利用。

第二个不足之处是：该分配方案会造成磁盘碎片。原本可以使用的空间被浪费了。磁盘压缩的代价往往很高，尽管这可以在深夜，当系统空闲的时候进行。

链接表分配

存储文件的第二种方法是为每个文件构造磁盘块的链接表，如图5-8所示。每个块的第一个字用于指向下一块的指针，块的其他部分存放数据。

图5-8 按磁盘块链接表来存储文件。

与连续分配方案不同，这种方法中每个磁盘块都被利用了。不会因为磁盘碎片而浪费存储空间(最后一块的内零头除外)。同样，在目录项中，只需要存放第一块的磁盘地址。文件的其他块可以根据这个地址来查找。

然而，在链接表分配方案中，尽管顺序读取文件非常方便，但是随机存取却相当缓慢。此外，因为指针占去了一些字节，每个磁盘块存储数据的字节数不再是2的幂，虽然这个问题并不足以致命，但它确实降低了系统的运行效率，因为大多数程序都是以长度为2的幂来读写磁盘块的。

使用索引的链接表分配

如果取出每个磁盘块的指针字，把它放在内存的表或索引中，就可以消除上述链接表的两个不足。就图5-8的例子而言，内存中表的内容如图5-9所示。这两个图中，有两个文件。文件A依次使用了磁盘块4、7、2、10和12，文件B依次使用了磁盘块6、3、11和14。利用图5-9的表，我们可以从第4块开始，顺

着链到底，找到文件A的所有磁盘块。同样，从第6块开始，顺着链到底，也能够找出文件B的所有磁盘块。
图5-9 使用内存表的链接表分配。

这样组织的话，整个块都可以存放数据。此外，随机存取也容易得多。要查找文件给定偏移位置，仍然要顺着链进行，但是整个链表都存放在内存中，不需要访问磁盘。同以前的方法一样，不管文件有多大，在目录项中只需记录一个整数(起始块号)，根据它可以找到文件的所有块。MS-DOS就使用这种方法进行磁盘分配。

这种方法的主要缺陷是我们必须把整个链表都存放在内存中。对于大磁盘，例如具有500000个1K大小块(500M)的磁盘，其链表中含有500000项，每项至少3个字节，为了提高查找速度，有时需要4个字节。这样，对于空间或时间的不同优化方案，这张表要占用1.5或2M内存。尽管MS-DOS使用这种分配方案，它在大磁盘上使用大的块(长达32K)，从而避免了使用很大的表。

i- 节点

记录各个文件分别包含哪些磁盘块的最后一个方法是给每个文件赋予一张称为i-节点(索引节点)的小型表，其中列出了文件属性和各块在磁盘上的地址，参见5-10。
图5-10 i-节点。

开始几个磁盘地址存放在i-节点内，因此对于小文件，所需信息均在i-节点中。在打开文件时，这些信息从磁盘读入主存。稍大一些的文件，在i-节点中有一个称为一次间接块的磁盘块的地址，这个磁盘块中含有附加的磁盘地址。如果文件再扩大，可以使用i-节点中的另一个地址，即所谓的二次间接块的地址，二次间接块包含有许多一次间接块的地址，而每个一次间接块又指向几百个数据块。如果这还不够的话，我们也可以使用三次间接块。UNIX中使用了这种分配方案。

5.3.2 实现目录

在读文件前，必须先打开文件。打开文件时，操作系统利用用户给出的路径名找到相应目录项，目录项中提供了查找文件磁盘块所需的信息。由于系统不同，这些信息可能是整个文件的磁盘地址(连续分配方案)、第一个块的块号(对于两种链接表分配方案)或者是i-节点号。无论怎样，目录系统的主要功能是把ASCII文件名映射成查找文件数据所需的信息。

与此密切相关的问题是在哪儿存放文件属性。一种较明显的方法是把文件属性直接存放在目录项中。许多系统确实是这样实现的。对于使用i-节点的系统，还存在另一种可能，即把文件属性存放在i-节点中。以后我们将看到，这种方法比把属性存放到目录项中更为优越。

CP/M中的目录

让我们先从一个特别简单的例子，CP/M(Golden and Pechura，1986)，来研究目录。CP/M的目录项如图5-11中所示。在这个系统中只有一个目录，因此要查找文件名，文件系统所要做的是查找这个唯一的目录。当找到对应的目录项后，也就知道了文件的磁盘块号。同所有文件属性一样，文件的磁盘块号也存放在目录项中。如果文件的磁盘块数多于一个目录项中所能容纳的数目，就为这个文件分配额外的目录项。
图5-11 目录项中包含每个文件的磁盘块号。

图5-11中各个域的含义如下。用户码域记录了文件拥有者。在查找过程中，只检查那些属于当前登录用户的目录项。接下来两个域给出了文件名和扩展名。多于16块的文件占有多个目录项，这时使用范围域。根据这个域，我们可以知道哪个目录项是文件的第一个目录项，哪个是第二个，等等。块数域给出了目录项中16个可能的磁盘块中实际使用的块数。最后16个域包含磁盘块号本身。最后一个块可能没有写满，因而系统无法确切地知道文件的字节数(即它是以磁盘块，而不是字节为单位来记录文件长度的)。

MS-DOS中的目录

我们现在考虑层次目录系统的一些例子。图5-12是一个MS-DOS的目录项。它总共32个字节长，其中包含了文件名、文件属性和第一个磁盘块的块号。根据第一个磁盘块的块号，顺着图5-9中的链，我们可以找到文件的所有块。
图5-12 MS-DOS目录项。

在MS-DOS中，目录可以包含其他目录，从而形成层次文件系统。通常在MS-DOS中，每个应用程序在根目录下创建一个目录，把它的所有文件都放在这个目录下，因此不同的应用程序不会发生冲突。

UNIX中的目录

UNIX中使用的目录结构非常简单，如图5-13所示，每个目录项只包含一个文件名及其i-节点号。有关文件类型、长度、时间、拥有者和磁盘块等所有信息都放在i-节点中。有些UNIX系统有不同的布局，但无论如何，目录项中最终要包含一个ASCII字符串和一个i-节点号。
图5-13 UNIX目录项。

在打开文件时，文件系统必须根据给出的文件名找到它所在的磁盘块。让我们分析如何来查找路径名/usr/ast/mbox。尽管是以UNIX作为例子，但是这个查找算法对所有的层次目录系统基本上都一样。首先文件系统找到根目录。在UNIX中，根目录的i-节点位于磁盘上的固定位置。

然后在根目录中查找路径的第一部分，usr，从而也就获得了文件/usr的i-节点号。因为每个i-节点都位于磁盘的固定位置，所以根据i-节点号找到i-节点是很直接的。利用这个i-节点，文件系统找到目录/usr，并接着查找下一部分ast。当找到ast目录项后，得到目录/usr/ast的i-节点。从而找到目录/usr/ast并在该目录

中查找文件mbox。接着，文件mbox的i-节点被读入内存，并保存在内存中，直至关闭该文件。这个查找过程见图5-14。

图5-14 查找/usr/ast/mbox的过程。

相对路径名的查找也类似。只是，相对路径名从工作目录，而不是根目录开始查找。每个目录在创建时都含有"."和".."项。"."项给出当前目录的i-节点号，".."项给出了其父目录的i-节点号。因而，查找../dick/prog.c的过程仅仅是在工作目录中查找".."项，找到父目录的i-节点号，并在父目录中查找到dick目录。我们不需要特别的机制来处理这些文件名，就目录系统而言，和其他的文件名一样，他们都仅仅是一些ASCII字符。

5.3.3 磁盘空间管理

文件通常存放在磁盘上，所以磁盘空间的管理是系统设计者要考虑的一个主要问题。存储n个字节的文件可以有两种策略：分配n个字节的连续磁盘空间，或者把文件分成许多个(并不一定要)连续的块。在内存管理器系统中，单纯段式和分页也要进行同样的权衡。

按连续字节序列来存储文件有一个明显问题，当文件扩大时，可能需要在磁盘上移动文件。内存中分段也具有同样的问题。不同的是，相对于把文件从磁盘的一个位置移动到另一个位置，段在内存中的移动操作要快得多。由于这个原因，几乎所有的文件系统都把文件分割成固定大小的块来存储，各块不必相邻。

块大小

一旦决定把文件按固定大小的块来存储，就会有一个问题：块大小应为多少呢？根据磁盘组织方式，扇区、磁道和柱面显然都可以作为分配单位。在页式系统中，页面大小也是主要选项之一。

如果分配单位大，比如以柱面为分配单位，这时每个文件，甚至是1个字节的文件，都要占用整个一个柱面。研究(Mullender和Tanenbaum,1984)表明,UNIX环境下的平均文件长度为1K。因而分配32K的柱面将浪费31/32或者说97%的磁盘空间。另一方面，小的分配单位意味着每个文件由许多块组成。每读一个块都要有寻道延迟和旋转延迟，因此，读取由许多小块组成的文件非常缓慢。

举一个例子，假设磁盘每道有32768个字节，其旋转时间为16.67毫秒，平均寻道时间为30毫秒。以毫秒为单位，读取一个k个字节的块所需时间是寻道延迟、旋转延迟和传送时间之和：

30+8.3+(k/32768)*16.67

图5-15的实线显示一个磁盘的数据读取速率与块大小之间的关系。如果我们粗略地假设所有文件都是1K字节(所测文件平均长度)，则磁盘空间的利用率如图5-15中虚线所示。不幸的是，从图中我们看到，随着磁盘空间利用率的提高，读取磁盘数据的速率降低。反之，在数据读取速率提高时，磁盘空间利用率随之下降。时间效率和空间效率本质上相互冲突。

图5-15 实线(左边标度)给出磁盘数据率，虚线(右边标度)给出磁盘空间效率，所有文件均为1K。

常见的折衷办法是把块大小选为512、1K或2K字节。如果在扇区大小为512字节的磁盘上选择1K大小的磁盘块，文件系统通常读写两个连续的扇区，并把他们看成是一个不可分割的单元。不管作了何种选择，我们都必须定期地重新计算块大小。这是因为，同计算机技术的其他方面一样，用户可使用的资源越来越多，但是，他们的需求也越来越高。有系统管理员声称，在他所管理的大学系统中，文件的平均长度近年来缓慢地增长，在1997年，学生的平均文件长度增至12K，而教师的则增至15K。

记录空闲块

一旦选定了块大小，接下来的问题就是如何记录空闲块。如图5-16所示，两种方法得到了广泛使用。第一种使用磁盘块的链接表。每个块中包含尽可能多的空闲磁盘块号。对于1K大小的块和32位的磁盘块号，空闲块链表中每个块包含有255个空闲块块号(我们需要使用一个项存放指向下一块的指针)。200M的磁盘最多需要804个块的空闲链表来存放所有200K个磁盘块的块号。通常情况下，我们使用空闲块存放空闲块链表。

图5-16 (a)将空闲表存放在链接表中。 (b)位图。

空闲磁盘空间管理的另一种方法是使用位图。n个块的磁盘需要n位位图。在位图中，空闲块用1表示，分配块用0表示(或者反之)。200M的磁盘需要有200K位来映照，即只需25个块。毋庸置疑，位图模型所需空间要少于链表模型所需磁盘空间，在链表模型中每个块用到32位，而在位图模型中每块仅需1位。只有在磁盘快满时链接表方案需要的块比位图更少。

如果有足够的内存来存放位图，那么这种方法无疑是非常好的。但是，如果只有一个块的内存可以用于记录空闲磁盘块，并且空闲磁盘块很少，这时选择链接表方案更佳。假设内存中只存放一个块的位图，那么可能在这个位图块中根本找不到空闲块，这时，我们不得不从磁盘上调入其他位图块。而将一个空的链接表块装入内存后，在从磁盘上读取链接表的下一块之前，我们可以进行多达255次分配。

5.3.4 文件系统的可靠性

比起计算机的损坏，文件系统的破坏往往要糟糕得多。如果由于火灾、闪电电流或者一杯咖啡泼在键盘上而弄坏了计算机，确实让人伤透脑筋，而且又要花上一笔钱，但一般来说，更换非常方便。求助于分销商，便宜的个人计算机在短短几个小时之内就可以修复(当然，如果这发生在大学里面，发出购买定单需征得3个委员会的同意，并盖5个公章，总共要花90天的时间，那么情况就不一样了)。

不管是硬件或软件的故障，或者是老鼠咬坏了软盘，如果计算机的文件系统被破坏了，恢复所有信息将是一件困难而又费时的工作，有些时候，这根本是不可能的。对于那些其程序、文档、客户文件、税收记录、数据库、市场计划或者是其他数据丢失的用户来说，这不啻为一次大的灾难。尽管文件系统无法防止设备和媒体的物理损坏，它至少应能保护信息。这一节，我们讨论与文件系统保护有关的一些问题。

正如我们在第三章中所指出的那样，磁盘中可能有坏块。在出厂时，软盘通常是完好无损的，然而在使用过程中，却可能出现坏块。温彻斯特磁盘常常一开始就有坏块：要把它做得完美无缺，成本实在是太高了。事实上，在老式磁盘上往往保留一个扇区作为坏块表，在出厂时其中记录着测试到的坏块。在控制器第一次初始化时，它读取坏块表，选择一个备用块(或磁道)取代有缺陷的块，并把这一信息记录在坏块表中。以后，所有对于坏块的请求都使用备用块。在发现新的坏块时，可以进行低级别的格式化来修改这个表。

制造技术已有了稳步的提高，坏块不再象以前那么普遍了，但还是会出现。我们在第三章说过，现代磁盘驱动器的控制器非常复杂。在这些磁盘上，磁道要比所需的至少多一个扇区，于是在一条磁道上出现坏块就不会造成任何问题，而只需要在读取数据时跳过它。 在每个柱面上还有一些备用的扇区，以便控制器在注意到读写某个扇区的复执次数大于某值时，能够自动将其重新映射至一个备用扇区。因而用户往往不会意识到坏块的存在以及坏块的管理。可是，当一个现代IDE或SCSI磁盘失效时，这往往是很可怕的，因为这时所有的备用块都已用光。SCSI磁盘在重映射磁盘块时， 给出"恢复时出错"信息。如果驱动器发现该错误信息，则将打印出一条消息，如果用户看到这条消息经常出现，就会知道他应该购买一个新磁盘了。

此外，对于坏块问题，还可以用软件方法加以解决，这种方法适合于老式磁盘，它要求用户或文件系统精心地构造一个包含全部坏块的文件。这种技术能将坏块从空闲表中删除，使其不会出现在数据文件之中。只要不对坏块进行读写操作，文件系统就不会出现任何问题。在磁盘备份时，需要注意避免读取这个文件。

备份

即使有再好的处理坏块的策略，我们也需要经常地备份文件。毕竟，在一些关键的数据块损坏之后，自动切换到备用块，无异于亡羊补牢。

备份软盘上的文件系统很简单，只需把整个磁盘拷贝到一张空软盘上。小型温彻斯特磁盘上的文件系统可以转储到磁带上。现代技术中使用了150M的磁带和8G的Exabyte或DAT磁带。

对于大型温彻斯特磁盘(例如10GB)，把整个驱动器的内容转储到磁带上是一件可怕而费时的工作。一种容易实现、但浪费了一半存储空间的策略是，每台计算机备有2个驱动器，每个驱动器都分成两部分：数据区和备份区，每天晚上，驱动器0中的数据被拷贝到驱动器1的备份区，反之，驱动器1中的数据拷贝到驱动器0的备份区，如图5-17。这样，即使有一个驱动器完全损坏了，也不会丢失任何信息。

图5-17 将一个驱动器中的数据备份到另一个驱动器上浪费了一半存储空间。

另一种转储整个文件系统的方法是增量转储(incremental dump)。最简单的增量转储形式是：定期地做一次全量转储，比如每周一次或者每月一次，此后，每天只存储自上次全量转储以后修改过的文件，或者可以采用一种更好的方案，即每天只转储那些自上次增量转储以来修改过的文件。

要实现这一方法，必须在磁盘上保存一张表，记录每个文件的转储时间。转储程序检查磁盘上的每个文件，如果这个文件在上次转储之后修改过，则再次转储，并且把它的"上次转储时间"改为当前时间。如果以月为周期进行转储，这种方法需要31盘日常转储磁带(每日一盘)，和足够多的磁带用于每月一次的全量转储。我们有时也用到其他更复杂但需要磁带更少的方法。

文件系统一致性

影响文件系统可靠性的另一个问题是文件系统的一致性。许多文件系统读取磁盘块，进行修改后，再写回磁盘。如果在修改过的磁盘块全部写回之前，系统崩溃，那么文件系统可能出现不一致。如果一些未被写回的块是i-节点块、目录块或者包含空闲表的磁盘块时，这个问题尤为严重。

为了解决文件系统的不一致问题，许多计算机都带有一个实用程序，检验文件系统的一致性。系统初启时，特别是在崩溃之后重新启动，可以运行该程序。下面我们讲述在UNIX和MINIX中，这个实用程序是如何工作的。在其他系统中，其工作原理类似。这些文件系统检验程序可以独立地检验各个文件系统(磁盘)的一致性。

一致性检查分为两种：块的一致性检查和文件的一致性检查。在检查块的一致性时，该程序建立两张表。每张表中，每块对应有一个计数器，初始值设为0。第一张表的计数器记录了每块在文件中出现的次数，第二张表的计数器记录了每块在空闲块链表(或空闲块位图)中出现的次数。

检验程序读取所有的i-节点，从i-节点开始，可以建立相应文件中使用的所有块的块号表。每当读到一个块号时 ，该块在第一张表中的计数器加1。接着这个程序检查空闲块链表或位图，查找所有未使用的块。每当在空闲表中找到一个块时，它在第二张表中的计数器加1。

如果文件系统一致，则每个块要么在第一张表中为1，要么在第二张表中为1，如图5-18(a)所示。可是系统崩溃后，这两张表可能如图5-18(b)。在图中，磁盘块2不出现在任何一张表中，这时报告块丢失。尽管块丢失不会造成损害，但却浪费了磁盘空间，减少了磁盘容量。解决块丢失问题是很直观的：文件系统检验程序只需要把他们加到空闲表中。

图5-18 文件系统状态： (a)一致 (b)块丢失 (c)空闲表中有重复块 (d)重复数据块

另一种可能出现的情况见图5-18(c)。这里，我们看到，磁盘块4在空闲表中出现了2次(只有在空闲表是一

张真正意义上的链表时，才会出现重复，在位图中，这种情况不会发生。)它的解决方法也是很简单的：我们只需要重新建立空闲表。

最糟糕的情况是，同一个数据块在两个或多个文件中出现，如图5-18(d)中的磁盘块5。如果删除任何一个文件，磁盘块5会加到空闲表中，导致一个磁盘块同时出现在文件和空闲表中。两个文件都删除后，这个磁盘块会在空闲表中出现两次。

文件系统检验程序可以这样来处理，先分配一个空闲块，把磁盘块5中的内容拷贝到空闲块中，然后把它插到其中一个文件之中。这样，文件中的内容未改变(虽然我们几乎可以肯定这些内容是不正确的)，而文件系统的结构保持了一致。这一错误应该报告出来，以便用户检查。

除了检查每个磁盘块外，文件系统检验程序还检查目录系统。这时也要用到一张计数器表，每个计数器对应于一个文件。检验程序从根目录开始，沿着目录树递归下降，检查文件系统中的每个目录。对每个目录中的文件，其i-节点对应的计数器加1(参见图5-13中的目录项布局)。

当全部检查完成后，得到一张表，对应于每个i-节点号，表中给出了指向这个i-节点的目录数目，然后，检验程序把这些数字与存储在文件i-节点中的链接数目相比较。在一致的文件系统中，这两个数目相吻合。但是，有可能出现两种错误，i-节点中的链接数太大或太小。

如果i-节点的链接数大于指向i-节点的目录项个数，这时，即使所有的文件都被删除，文件链接数仍然为非0值，文件i-节点不会被删除。这一错误并不严重，可是却浪费了磁盘空间。我们可以把i-节点中的文件链接数设置成正确的值来改正这一错误。

另一种错误则是一种潜在的灾难。如果两个目录项都链接到同一个文件，但其i-节点的文件链接数只为1，如果删除任何一个目录项，i-节点链接数变为0。文件系统将该i-节点标志为"未使用"，并释放该文件的所有磁盘块。这将导致一个目录指向一个未使用的i-节点，而其磁盘块很可能马上分配给其他文件。同样，纠正方法是把i-节点中的链接数设置为目录项的实际数目。

由于效率的原因，检查磁盘块和检查目录的操作常常结合在一起(即仅需对i-节点扫描一遍)。当然还存在着一些其他的检查方法。例如，目录项有明确的格式：i-节点号和ASCII文件名，如果某个i-节点号大于磁盘中i-节点的总数，显然这个目录被破坏了。

此外，每个i-节点都包含有一个模式项。有些模式是合法的，但值得怀疑，比如0007，它不允许文件主及其所在用户组的成员进行访问，而其他用户却可以读写和执行此文件。这种情况下，我们有必要让系统把其他用户权限高于文件主权限这一情况报告出来。拥有1000个目录项的目录也很可疑。放在用户目录下，但为超级块用户所拥有，并且设置了SETUID位的文件，也可能有安全问题。稍加努力，我们可以列出一长串特殊情况，这些情况尽管合法，但却有必要报告给用户。

以上我们讨论了保护用户文件不因系统崩溃而破坏的问题，某些文件系统也防止用户自身的误操作，如果用户想输入

```
rm  *.o
```

删除所有以.o结尾的文件(编译器生成的目标文件)，但不幸键入的是

```
rm  *  .o
```

(请注意，星号后面有空格)，那么rm命令将删除当前目录中的所有文件，然后报告说不能找到文件.o。在MS-DOS和一些其他系统中，删除文件仅仅是设置目录或i-节点的某一位，标志文件被删去，在实际需要前，磁盘块并不返回到空闲表中，因此，如果用户马上发现了操作错误，他可以运行一个特定的实用程序，恢复被删除的文件。在WINDOWS 95中，被删除的文件放在recycled目录下，便于用户检索，在这些文件从该目录中删除之前，并不回收其存储空间。

5.3.5 文件系统性能

访问磁盘要比访问内存慢得多。在内存中读取一个字往往需要几十纳秒，而从硬盘上读取一个块则需要50多个微秒，此外还需要加上10到20毫秒的寻道时间，然后再等待要读取的扇区移到读写头下面。如果只读一个字，内存访问要比磁盘快100000倍，正因为此，许多文件系统在设计时都尽量减少磁盘访问次数。

减少磁盘访问次数最常用技术是块高速缓存(block cache或者buffer cache)。在这里，高速缓存是一些块，他们逻辑上属于磁盘，但基于性能的考虑而保存在内存中。

在管理高速缓存时，用到了不同的算法。一个常用算法是：检查所有的读请求，看看所需的块是否在高速缓存中。如果在，无须访问磁盘便可进行读操作。如果块不在高速缓存中，首先要把它读到高速缓存，再拷贝到所需的地方。之后，对该块的读写请求都通过高速缓存完成。

如果高速缓存已满，此时要调入新的块，需要把原来的某一块调出高速缓存，如果要调出的块自上次调入以后作过修改，则需要把它写回磁盘。这种情况与分页非常相似，因此，第四章说过的所有的分页算法，例如FIFO算法、第二次机会算法、LRU算法等等都适用于高速缓存。分页和高速缓存的不同之处在于，高速缓存引用相对要少，因此我们可以把所有块按精确的LRU顺序用链表链接起来。

不幸的是，这里有一个使人两难之处。在这种情况下，虽然我们可以使用精确的LRU算法，但它却又带来了问题，这与前一节讨论过的系统崩溃和文件一致性有关。如果一个关键块，比如说i-节点块，读入到高速缓存并作过修改，但是没有写回磁盘，这时，系统崩溃将导致文件系统的不一致。如果我们把i-节点块放在LRU

链表的尾部，在它到达链首并写回磁盘前，可能需要相当长的一段时间。

此外，某些磁盘块，比如两次间接块，在一个短的时间内，很少被引用两次。基于这些考虑，我们需要修改LRU方案，并注意以下两点：

1 是否这一块不久要重新用到?

2 是否这一块关系到文件系统的一致性?

就这两个问题而言，可以把块分为i-节点块、间接块、目录块、全数据块、部分数据块等几类。最近可能不再需要的块放在链表前端，这样，他们的缓冲区很快又可使用。不久又有可能使用的块，比如正在写的部分数据块，放在链表的尾部，以便他们在高速缓存中保存一段较长的时间。

第二个问题与前一个相独立。如果某块关系到文件系统的一致性(除数据块之外，其他块基本上都是这样)，它被修改后，不管是否放在链表尾部，都应该立即写回磁盘。把关键的块迅速写回磁盘，我们大大减少了计算机崩溃后，文件系统被破坏的可能性。

尽管这种方法可以保证文件系统的一致性不受破坏，我们也不希望将数据块放在高速缓存中很久之后才写入磁盘。设想某人用个人计算机编写一本书。即使作者定期要求编辑器把正在编辑的文件写回磁盘，他输入的内容只放在高速缓存中的可能性还是非常大。这时如果系统崩溃，文件系统的结构并不会遭到破坏，但是他这整天的工作都将毁于一旦。

这种情况即使只发生几次，便足以让我们感到不愉快。系统采用两种方法来解决这一问题。UNIX系统中有一个系统调用，SYNC，使所有修改过的块立即写回磁盘。在系统启动时，一个称作update的程序在后台运行，它处于无休止的循环之中，不断执行SYNC调用，在两次调用之间睡眠30秒。这样，即使系统崩溃，丢失的也只是30秒内的工作。

在MS-DOS中，块被修改时，该块同时写回磁盘。如果修改过的块立即写回磁盘，我们称这样的高速缓存为直写高速缓存。和非直写高速缓存相比，直写高速缓存需要更多的磁盘输入/输出。当一个程序欲写满1K的块，每次写一个字符时，我们可以看到这两种方法的区别。UNIX把所有字符保存在高速缓存中，每30秒把这个块写回磁盘，或者当这一块从高速缓存中删除时，写回磁盘。在MS-DOS中，每写入一个字符，访问一次磁盘。当然，大多数程序都有内部缓冲，因此通常情况下，在执行系统调用WRITE时，并不是逐个字符写入的，而是以行或者更大的单位写入。

这两种不同的高速缓存策略的结果是：在UNIX系统中，未调用SYNC就移走(软)磁盘，往往会导致数据丢失，有时还会破坏文件系统。而在MS-DOS中，则不会出现这种情况。之所以选择不同的策略，是因为UNIX是在这样的环境下开发出来的：所有的磁盘都是硬盘，不可移动。而MS-DOS则是从软盘世界发展起来的。随着硬盘成为标准，甚至在小型微机中，UNIX的高速缓存方案由于其更高的效率，必将会成为我们当然的选择。

高速缓存并不是改善文件系统性能的唯一方法，另一种重要技术是把那些有可能顺序存取的块放在一起，最好是同一个柱面上，从而减少磁盘臂的移动次数。当写一个输出文件时，若该文件需要新的块，文件系统就为它分配一块。如果空闲块用位图来记录，并且整个位图放在主存中，我们很容易选择与前一块最近的空闲块。如果采用空闲链表，并且链表有一部分内容存放在磁盘上，找到这样的空闲块就要困难得多了。

然而，即使是采用空闲链表，也可以使用块群技术。这里用到一个小技巧，即使用连续块群，而不是块，来记录磁盘存储区。如果每道由64个扇区组成，每个扇区有512个字节，系统可能使用1K大小的块(2个扇区)，但却按每2个块(4个扇区)为一个单位来分配磁盘存储区。这和2K的块并不一样，它在高速缓存中依然使用1K大小的块，磁盘与内存数据传送也是以1K为单位进行，但在一个不是很空闲的系统上顺序读取文件，其寻道次数可以减少一半，从而大大地改善文件系统的性能。

若考虑到旋转定位，我们可以得到这种方案的一个变体。当分配块时，系统尽量把文件的连续块存放在同一柱面上，但加以交叉以获取最大吞吐率。这样，如果磁盘的旋转延迟为16.67毫秒，并且用户进程需花4毫秒来请求并读取一块数据，则每个数据块的位置应距离前一块至少四分之一磁道。

在使用i-节点或者与i-节点等价结构的系统中，另一个性能瓶颈在于，即使读取一个很短的文件也要访问两次磁盘：一次是读取i-节点，另一次是读取文件块。通常情况下，i-节点的放置如图5-19(a)所示，图中，所有i-节点都靠近磁盘头部，因此i-节点和相应块之间的平均距离是柱面总数的一半，这需要很长的寻道延迟。图5-19 (a)i-节点放在磁盘开始位置。 (b)磁盘分为柱面组，每组有自己的块和i-节点。

一个简单的改进方法是把i-节点放在磁盘中部。这时，在i-节点和第一块之间的平均寻道时间减为原来的一半。另一种想法是：把磁盘分成多个柱面组，每个柱面组有自己的i-节点、数据块和空闲表，参见图5-19(b)(Mckusick et al.，1984)。在创建文件时，可以选取任一个i-节点，然而分配块时，在该i-节点所在的柱面组上进行查找，如果该柱面组中没有空闲的数据块，就查找与之相邻的柱面组。

5.3.6 日志结构的文件系统

技术的改进使得当前的文件系统面临着很大压力。特别地，CPU速度越来越快，磁盘容量不断增大，成本不断降低(但是访问速度没有很大的提高)，内存容量呈指数增长，然而有一个参数没有得到迅速改进，这就是磁盘寻道时间，所有这些因素结合在一起，表明在许多文件系统中正出现着一个瓶颈。Berkely学院设计了一种全新的文件系统，即日志结构的文件系统(log-structured file system)，试图减轻这个问题。本节我们简要地讲述一下LFS的工作原理，要了解更详细的内容，请参见(Rosenblum和Ousterout，1991)。

促成LFS设计的想法是：CPU越来越快，RAM内存越来越大，磁盘高速缓存的容量迅速增加。因此，无需访问磁盘，从文件系统的高速缓存中就可能满足所有读请求。将来大多数磁盘访问是写操作。某些文件系统使用的预读机制，即把数据块在实际需要前调入内存，对文件系统性能的改进不再那么重要了。

更糟糕的是，在大多数文件系统中，写操作都是以小块为单位进行的，效率很差，因为在50微秒的写磁盘之前，需要有10毫秒的寻道延迟和6毫秒的旋转延迟。由于后者的存在，使得磁盘效率还不足1%。

要看看是什么引起小块写，我们考虑在UNIX系统中创建一个新文件。为了写这个文件，必须对目录的i-节点、目录块、文件的i-节点和文件本身执行写操作。尽管这些写操作可以延迟进行，但是延迟写在系统崩溃时，很容易使文件系统产生严重的一致性问题。因此，i-节点一般都是立即写入的。

基于上述推理，LFS的设计人员决定重新设计UNIX的文件系统，他们希望即使在有大量小块随机写的情况下，也能获得磁盘的全部带宽。基本思想是把整个磁盘作为日志。所有写操作都存放在内存的缓冲区中，并定期地收集到一个单独的段中，作为日志尾部的相邻段写回磁盘。因此，每个段都含有i-节点、目录块、数据块等，并且是他们的混合体。在每段起始位置还有一个摘要，给出了该段中的内容。如果段的平均长度大约为1MB，那么几乎所有的磁盘带宽都能利用。

在这一设计中，i-节点依然存在，并且和UNIX中的i-节点具有相同结构。但是这些i-节点并不放在磁盘的固定位置，而是分散在日志之中。一旦找到i-节点，可以用通常的方法找到相应块。毫无疑问，现在查找一个i-节点要困难得多。我们不能象UNIX那样，根据i-节点号通过简单计算来得到i-节点的位置。为了查找i-节点，需要维护一张i-节点映照表，它以i-节点号为下标，其中第i项指向磁盘上第i个i-节点。这张映照表存放在磁盘中，但它同样使用缓存机构，所以在大多时候，最常用的部分将保存在内存中。

我们小结一下LFS的工作方式。所有写的数据开始时都存放在缓冲区中，并且定期地把这些缓冲区中的数据以一个段的形式写到磁盘中，放在日志的尾部。打开一个文件首先要在i-节点映照表中查找该文件的i-节点，一旦找到了i-节点， 也就知道了相应块的地址，所有的块也存放在段中，即日志的某个地方。

如果磁盘容量无限大，上面的描述无疑非常完美。但是，真正的磁盘都有有限的容量，最终日志将占满整个磁盘，这时新段不能被写到日志中。幸好，许多现有段都包含一些不再使用的块。例如，如果文件被重写，其i-节点将指向新块，而原来的块仍然占有以前段的空间。

为了解决这两个问题，LFS中有一个清理工(cleaner)线程循环地浏览和压缩磁盘。它首先读取日志中的第一个段的摘要，找出其中的i-节点和文件。接着查找当前的i-节点映照表，检查i-节点是否还在使用以及文件块是否还在使用。若没有使用，这些信息将被丢弃。还在使用的i-节点和块读入内存中，以便写到下一个段中。原来的段标记为空闲，以便日志用于存放新数据。这样，清理工线程沿日志向前移动，删除旧段，把有效数据读入内存，写入下一新段。这样，磁盘是一个循环的缓冲区，写线程不断地在前面添加新段，而清理工线程不断地从后面删除旧段。

这里，簿记工作比较麻烦，当某文件块写回到一个新段时，先要在日志中找到该文件的i-节点，对其进行修改，然后把它放在内存中，以便写回下一段。i-节点映照也必须进行相应修改以指向新的拷贝。但是总的来说，这种管理是可取的，性能研究表明这种复杂性是值得的。前面提到的论文测试显示，在进行大量小块写的情况下，LFS的性能明显超过了UNIX。而在读数据和大块写数据时，其性能也同于，甚至要好于UNIX。

5.4 安全性

文件系统往往包含有用户非常宝贵的信息。因此，如何保护这些信息不被未授权使用是所有文件系统的一个主要内容。下面几节中，我们讨论与安全和保护有关的一些问题。这些问题既适用于分时系统，也适用于通过局域网连接到共享服务器的个人计算机网络。

5.4.1 安全环境

安全和保护这两个术语经常交替使用。然而，我们有必要区分两类不同的问题，其中一类是确保未被授权用户无法读取或修改某些文件，它包括技术、管理、法律和政治等方面。而另一类则指用于提供安全的特定操作系统机制。为避免混淆，我们用安全性(security)指这些综合的问题，用保护机制(protection machanism)来指用于保护计算机信息的特定操作系统机制。然而，他们之间的界线并不非常明确。我们首先来看看安全性。在这章的后面部分我们再讨论保护机制。

安全性有许多方面，其中较重要的两个方面是数据丢失和入侵者。造成数据丢失的原因往往是：

1. 灾祸：火灾、洪水、地震、战争、暴乱或者是老鼠咬坏磁带或软盘等等。
2. 硬件或软件故障：CPU误操作、不可读取的磁盘或磁带、远程通信故障、程序故障等等。
3. 人的失误：不正确的数据输入、磁带或磁盘安装故障、程序运行错误、磁带或磁盘丢失或者一些其他的错误。

这些数据丢失问题大多数可以通过保存足够的备份而解决，最好是将备份数据放在与源数据相隔较远的地方。

一个有趣的话题是如何对付入侵者。入侵者可以分为两类：消极的入侵者只想读取未授权的文件。积极的入侵者则怀有恶意，他们在未获授权的情况下试图修改文件数据。在设计一个安全的系统时，必须记住要防止哪一类入侵者。下面是较常见的几类入侵：

1. 非技术人员的偶然窥视。大多数人在自己的工作台上都摆放着一台分时系统终端或者联网的个人电脑。不管他们的本性如何，如果我们不设置障碍的话，某些人会读取别人的电子邮件或其他文件。在大多数UNIX系统中，所有文件的缺省属性均为可读的。

2. 入侵者的窥探。学生、系统程序员、操作人员和其他一些技术人员常常以突破局部计算机系统的安全性为个人能力的一次挑战。他们往往都技术娴熟，并且愿意花大量的时间来从事这一努力。
3. 明确的偷窃企图。一些银行程序员试图突入银行系统，从中窃取金钱。他们的阴谋五花八门，包括删除利息据为己有、盗用多年未使用的帐户、甚至敲诈勒索("赶快给我一笔钱，否则我就删除银行所有记录。")
4. 商业或军事间谍活动。间谍活动指的是由竞争对手或者外国政府资助的、正规的、高投入的活动，主要是窃取对方的程序、商业秘密、专利、技术、电路设计和市场计划等等。他们常常窃听对方的信道消息甚至树起天线截获对方计算机的电磁辐射。

我们应该清楚：防止敌对的外国政府窃取军事秘密和防止学生在系统中插入一条有趣的"今日要闻"是完全不同的两码事。人们花在安全和保护上的劳动显然取决于他们所设想的对手是谁。

安全问题的另一方面是隐私权，即保护个人信息不被滥用。这牵涉到许多法律和道德上的问题：政府是否有权为了捕捉X骗子而搜集个人档案呢？这里X可以是"财富"或者是"税收"，这主要取决于你的政策了。警察是否可以对某些人进行搜查来阻止有组织的犯罪活动呢？雇主和保险公司是否有这种权力呢？当这些权力和个人权利相悖时，会出现什么情况呢？所有这些问题都非常重要，但他们都超出了本书的范围。

5.4.2 著名的安全缺陷

正如运输界有泰坦尼克等事故一样，计算机安全专家往往也容易忽略某些细节。在本节中，我们介绍在三种不同的操作系统：UNIX，TENEX和OS/360中发生的一些有趣的安全问题。

UNIX中实用程序lpr用于打印一个文件，在打印完成后，允许用户选择是否删除该文件。在UNIX的早期版本中，任何人都可以使用lpr进行打印，从而他们都可以从系统中删除口令文件。

突破UNIX的另一种方法是将当前目录下的core文件链接到口令文件上，然后，入侵者促使对某个SETUID程序进行core转储。这样，用户可以用一个包含自由选取的字符串文件(即命令参数)来代替口令文件。

在UNIX中，另一个缺陷与命令

```
mkdir  foo
```

有关。mkdir是一个根用户拥有的SETUID程序，它首先使用系统调用MKNOD为目录foo创建一个i-节点，然后把foo的拥有者从有效uid(即根)改为实际uid(用户的uid)。当系统很慢时，用户有可能迅速地删除foo目录的i-节点，而以foo为名字建立一个到口令文件上的链接。如果时机选择得好，上述操作可以在mkdir执行MKNOD之后，执行CHOWN之前完成。这样，执行CHOWN后，用户就变为口令文件的拥有者。在shell脚本中放入一些必要的命令，用户可以反复运行这些命令，直至目的得逞。

TENEX操作系统以前广泛运行在DEC-10机器上，如今它不再使用了，但是由于以下的设计缺陷，它的"名声"将永远留在计算机的安全领域。TENEX支持分页，为了使用户能监控程序的行为，他们可以要求系统在页错误时，调用自己的函数进行相应处理。

TENEX也使用口令来保护文件，要存取一个文件。程序中必须给出正确的口令。操作系统在检测口令时，每次检查一个字符，一旦发现口令错误，立即停止执行。要闯入TENEX，入侵者可以精心地安排口令，如图5-20(a)所示，把第一个字符放在一页的末尾，而剩下部分放在下一页的开头。

图5-20 TENEX口令问题。

接下来，要确保第二页不在内存之中。通过多次引用其他页面，我们可以保证第二页被调出内存。现在，程序可以使用这个精心排列的口令试着打开受害者的文件。如果真实口令中的第一个字符不是A，系统立即停止检查并报告"口令非法"。可是，如果口令确实是以A开始，系统继续读下面的字符，得到页错误，这一错误也会报告给入侵者。

如果口令不是以A开始，入侵者修改口令如图5-20(b)所示，并重复上述整个过程，看看口令是否以B开始。要确定口令的第一个字符，最多只需尝试128次，即检查整个ASCII字符集。

假设第一个字符为F，那么图5－20(c)所示的内存布局可使入侵者检测FA、FB等形式的字符串。采用这种方法，猜出一个n个字符的ASCII口令最多仅需要128n次，而不是128n次。

我们想介绍的最后一个缺陷与OS/360有关。下面的描述略微简单，但保持了缺陷的本质。在该系统中，我们可以启动磁带读操作，并在磁带驱动器将数据传送到用户空间时，继续执行运算。这里我们用到一个小技巧，精心地启动读磁带操作，然后执行一个要求用户输入数据结构的系统调用，例如，文件及其口令。

操作系统首先检查给定文件的口令正确无误，然后返回，再次读取文件名以访问之(它原本可以将文件名在内部保存起来，但是却没有这样做)。可惜，就在系统第二次读取文件名前，文件名被磁带驱动器重写了。这样系统读到的是一个新文件，而这个文件无需用户输入口令。掌握时机可能需要动些脑筋，但这并不困难。事实上，计算机所擅长的无疑便是反复地进行同一操作。

除了这些例子之外，近年来还出现了许多其他的安全问题和攻击方法。人们经常会听到特洛伊木马(Trojan horse)，特洛伊木马是一个貌似无辜并广泛流传的程序，它执行人们意料不到的操作，例如窃取用户数据并用电子邮件传送到某台远距离的计算机上。

在如今工作无保障的年代里，另一个安全问题是逻辑炸弹(logic bomb)。这是公司(当前雇佣)的程序员编写的一段代码，程序员把它悄悄地插入到生产性的操作系统中。只要程序员每天输入口令，则一切正常。如果他突然被解雇，那么第二天，这个逻辑炸弹无法得到口令，它就将爆炸。

一旦爆炸，系统可能清空磁盘、随机地删除文件、对关键程序作难以察觉的修改或者加密一些必需的文件等等。其后，公司不得不在下列两者之间作出艰难的选择：报警(即使过了几个月，也不能保证定罪)或者向勒索者让步,以天文数字的薪水重新雇佣他为"顾问"来纠正这个问题(并且希望他在纠正时不要再放置一个新的逻辑炸弹)。

或许，历史上最大的计算机安全犯罪发生在1988年11月2日。当时康纳尔大学的研究生，Robert Tappan Morris，把一个蠕虫程序放在Internet上，最终使得全世界成千上万台机器陷于崩溃。

这个蠕虫由两个程序组成，即引导程序和真蠕虫程序。引导程序是一个名为l1.c的99行程序，在所攻击的系统中编译、执行。运行时，它连接到它原来所在的机器上，然后下载真蠕虫程序并执行。在设法将自己隐藏起来后，蠕虫接着查阅这台主机的路由表，看看哪些机器连接到这台主机，并试图把引导程序传播到那些机器。

进入某台机器后，蠕虫试着猜出用户的口令。Morris并没有化太多精力做这件事，他的父亲是美国国家安全局---美国政府绝密代码破译机构---的安全专家，十年前在贝尔实验室工作时，曾和Ken Thompson合作写过一篇关于安全问题的经典文章(Morris和Thompson，1979)。Morris只需向他父亲要一份这篇文章的副本，其中每个破译出的口令允许蠕虫在使用这些口令的用户拥有帐号的任何机器上登录。

Morris被捕是因为，有一天，他的一个朋友与纽约时报的计算机报道员John Morkoff交谈，并试图让Morkoff相信这一事件纯属偶然，蠕虫程序毫无恶意以及程序编制者非常难过等等。第二天，这一消息刊登在报纸的显著位置，甚至超过了三天后的总统选举。联邦法院对Morris进行了审判，对其处以10000美元的罚款，3年缓行以及400小时的公众服务。诉讼费可能高达15,000美元。

这一判罚引起了很大的争论，计算机界的许多人认为Morris是一个十分聪明的研究生，他只不过搞了一次恶作剧，蠕虫中没有证据证明他试图窃取或破坏如何东西。而另一些人则认定Morris是一个罪犯，应该被判处监禁。

这一事件导致了计算机紧急事件反应组的建立。CERT为人们提供了一个报告非法入侵的场所，其中，有一群专家专门分析安全问题以及设计纠正方法。这无疑是一次大的进步，但仍然有不足之处。CERT收集有关系统攻击方法及弥补对策的信息，它把这些信息通过Internet传送给成千上万个系统管理员，这就意味着某些坏蛋可以获取这些信息，在漏洞补好之前几个小时(甚至几天)大搞破坏活动。

5.4.3 一般的安全性攻击

上述的缺陷都早已修复了。但是，一般的操作系统还会有其他问题。测试系统安全性的常用方法是雇佣一群专家组成猛虎组(tiger teams)或者叫渗透组(penetration teams),看看他们是否能够闯入系统之中。Hebbard et al.(1980)和他的研究生做了同样的工作。许多年来，这些渗透小组发现在许多方面系统容易出现安全问题。下面我们列出一些常常奏效的攻击方法。当你在设计一个系统时，需要防止类似的攻击：

1. 请求内存页、磁盘空间和磁带并读取其内容。许多系统在分配时并不删除以前的内容，而其中往往有很多其他用户写入的重要信息。
2. 尝试非法的系统调用、或者是带非法参数的合法系统调用、或者是带有合法而不合适的参数的合法系统调用。许多系统很容易引起混淆。
3. 在登录过程中途键入DEL、RUBOUT或BREAK。有些系统中，口令检查程序被终止，认为登录成功。
4. 试图修改保存在用户空间内的、复杂的操作系统结构(如果有的话)。在某些系统(特别是一些大型机)中，在打开文件时，程序创建一个大的数据结构，其中包含有文件名和许多其他参数，并把它传给系统。在读写文件时，系统有时修改这些结构。修改这些域可能造成安全性的严重破坏。
5. 写一段程序来哄骗用户，程序在屏幕上打印"login："。许多用户在登录时键入的用户名和口令被程序详细地记录下来。
6. 仔细查阅手册中关于"请不要做XXX"的说明，然后试验各种变通方法。
7. 说服系统程序员修改系统，使得在用户以你的用户名登录时，跳过某些重要的安全性检查，这种攻击方法称为暗门(trapdoor)。
8. 在其他方法都无效时，渗透者有可能找到计算中心主任的秘书，给他一笔数量可观的贿赂，而这个秘书或许有权访问各种重要的信息，但薪水很低，可千万不要低估了这些职员带来的问题。

这些以及其他一些攻击方法在Linde(1975)中都有讨论。

病毒

一类特殊的攻击是计算机病毒。病毒已经成为许多计算机用户的主要问题。病毒(virus)是一个程序段，它附在合法的程序中，并企图感染其他程序。和蠕虫不同的是，病毒附在现有程序中，而蠕虫本身是一个完整的程序。病毒和蠕虫都试图广泛传播，他们都可能造成严重的后果。

病毒往往是这样流行的：病毒编制者首先写一个有用的程序，比如是MS-DOS下的一个游戏，这个程序中潜伏着病毒。然后，病毒编制者把游戏上载到公告牌系统中，或者以软盘的形式免费或廉价地提供给用户。接着，这个程序被大肆宣扬，许多人下载并使用它。编制病毒并非一件容易的事，因此这些病毒编写者往往是一些很聪明的程序员，他们编写的游戏或者其他程序都非常优秀。

病毒程序启动后，它立即检查硬盘上的所有的二进制文件，检查他们是否已被感染。如果找到一个未被感染的文件，病毒代码被加在文件尾部，并且把第一条指令换成跳转语句，转而执行病毒代码。在病毒代码执行

完毕后，接着执行原来程序的第一条指令，然后跳到第二条指令处继续执行。这样，每次被感染的程序运行时，它总是感染更多的程序。

除了仅仅感染程序之外，病毒程序还可以做一些其他事情。例如删除、修改或者加密文件。有的程序甚至在屏幕上显示一条勒索信息，要该用户送500美元现金到巴拿马的某个邮政局信箱中，否则他就将遭受数据或硬件的巨大损失。

病毒程序还可能感染硬盘的引导区，导致计算机无法启动，这样的病毒可能会要求用户输入口令，而病毒的编制者往往廉价地提供这种口令。

和删除病毒相比，预防病毒要容易得多。最安全的办法是从可信赖的商店里购买精装软件。从公告牌系统中下载软件或者是用软盘拷贝盗版软件无疑是自找麻烦。一些商用杀毒软件也有销售，但是，其中大多数只能查找已知的病毒。

更一般的方法是重新格式化整个硬盘，包括引导区，再安装所有可靠软件，对每个文件计算一个校验码。采用何种算法无关紧要，只是，校验码要有足够多的位(至少32位)。把这些(文件，校验码)对存放在一个安全地方。你可以存放在软盘中或者加密后存放到硬盘上。这样当系统启动时，计算所有文件的校验码并与原来的校验码相比较。任何文件，如果两者的比较结果不一致，都值得怀疑。尽管这种办法无法防止文件被病毒感染，至少它可以较早地发现病毒。

如果使目录中的二进制文件对普通用户不可写，则感染这些文件要困难得多。这种技术使得病毒难以修改二进制文件。虽然它可以用在UNIX中，但对MS-DOS却不适用，因为后者的目录根本不可能设置为不可写。

5.4.4 安全性的设计原则

大多数情况下，病毒发生在桌面系统中。在更大的系统中，会出现其他问题，需要一些其他方法来解决。Saltzer和Schroeder(1975)列出了几个用于指导安全系统设计的一般原则，下面简要地介绍他们的想法(基于MULTICS的经验)：

首先，系统设计必须公开，假设入侵者不知道系统的工作方式无疑是自欺欺人。

其次，缺省属性应该是不可访问。合法访问被拒绝的错误，会比未授权访问被容许出现的错误更快地报告出来。

第三，检查当前权限。 系统不应该先检查权限，确定该访问合法，然后把这个信息保留以备后用。许多系统是在打开文件时检查权限，而不是在以后。 这意味着一个用户打开文件几个星期后，依然有权存取该文件，即使文件主早就修改了文件的权限。

第四，给每个进程赋予一个最小的可能权限。如果一个编辑器只有权存取它所编辑的文件(在编辑器启动时指定)，这时，即使它带有特洛伊木马病毒，也不会造成很大的损害。这个原则蕴含着一个小颗粒度的保护方案。我们后面还会谈到这样的保护方案。

第五，保护机制要力求简单一致、嵌入到系统的底层。要想在不安全的系统上改进安全性几乎是不可能的。同正确性一样，安全性也不是一个附加的特征。

第六，采取的方案必须可以接受。如果用户觉得保护其文件太麻烦，他们根本就不会去保护，然而，在出现问题时他们会不停地抱怨系统设计者，这时对他们说："这是你自己的问题"，用户一般是不会接受的。

5.4.5 用户验证

许多保护方案都假定系统知道用户的身份。当用户登录时，检验其身份的问题叫做用户认证(user authentication)。大多数验证方法都基于以下的检验，即用户知道些什么、用户拥有什么或者用户是什么等等。

口令

使用最广泛的用户认证形式是用户输入口令。口令保护容易理解，也容易实现。在UNIX中口令是这样工作的：登录程序要求用户输入用户名和口令，这时口令立即加密，然后登录程序读取口令文件，通常是许多行的ASCII文件，直至找到包含用户登录名的那一行，如果这行(加密)的口令与计算得到的加密口令相符，则允许登录，否则不予登录。

口令认证也很容易被突破。人们经常可以听到一群高中生，甚至是初中生，借助于他们的家庭电脑，闯入到某大公司或者政府机构的绝密系统中去。一般来说，他们总是想办法猜出用户名和口令组合。

尽管近年来在口令安全方面进行着越来越多的研究，然而经典的工作还是Morris和Thampson(1979)在UNIX系统中所做的。他们搜集并编成了一张可能的口令表，其中包括姓名、街名、城市名、中型字典中的单词(包括反过来拼写的单词)、门牌号码以及很短的随机字符组成的字符串。然后他们使用已有的加密算法对这些可能的密码进行加密，检查任何加密口令是否与他们的表中某一项相匹配。超过86%的口令都在他们的表中出现。

如果口令由7个字符组成，每个字符随机取自95个可打印的ASCII字符，那么我们需要搜索957种组合，约等于7*1013。

即使以每秒加密1000项的速度，也需要2000多年才能建立起一张这样的表。何况，这张表要耗费20亿盘磁带。甚至要求口令包含一个小写字母、一个大写字母、一个特殊字符以及至少7到8个字符，也是对用户自由选择口令的一个极大地改进。

尽管要求每个用户都合理地挑选口令在实际上是做不到的，Morris和Thompson还是提出了一种使得他们自己的攻击方法(事先对大量的口令加密)失效的方法。他们的想法是给每个口令赋予一个n位随机数字。当口令改

变时，随机数随之改变。随机数以明码的形式存放在口令文件中，任何人都可以读取。这时，我们不单单加密口令，而是把口令和随机数连接起来一起加密，将这个加密的结果存储在口令文件中。

设想一个入侵者试图建立一张可能的口令表，对他们进行加密，把这个结果存放在文件f中，以便从中查找加密口令。如果入侵者猜测到口令可能是Marilyn，这时他不仅仅对Marilyn加密，把结果放在文件f中。事实上，他必须加密2n个字符串，如Marilyn0000、Marilyn0001、Marilyn0002等等，并把他们全部放到f中。这种方法把文件f的长度增大了2n倍。UNIX也使用这种方法，其n值为12。这种方法也称为盐渍口令文件。一些版本的UNIX使得口令文件本身不可读取，但提供了一个程序来查找，这样就大大地减慢了破译速度。

虽然这种方法可以防止入侵者预先计算一张大的加密口令表，它却无法保护用户David使用David作口令的情况。一种鼓励用户挑选好的口令的方法是让计算机提供一些建议。有些计算机程序生成容易拼读的、无意义的单词作为口令，例如fotally、garbungy或bipitty(最好其中有一些大写字符和一些特殊字符)。

有的计算机则要求用户定期改变其口令，从而减少口令泄漏可能造成的损失。最极端的方法是使用一次性口令。这种情况下，用户备有一本口令书，其中记载着一长串口令，每次登录时都使用书中下一个口令。如果入侵者偶然破译出口令，也没有什么好处，用户下一次就会使用不同的口令。不过用户需要好好保管他的口令书。

毫无疑问，在用户键入口令时，计算机不应该把敲入的字符显示出来，以防旁边有人窥视。此外，口令决不要以未加密的形式存放，甚至是计算机中心的管理部门也不应该有未加密的口令拷贝。口令未加密就存放起来，简直是自找麻烦。

口令思想的一个变通形式是给每个用户提供一长串的问题，答案以加密形式存放起来。这些问题应该尽量为用户所熟悉，用户根本无需把他们记录下来。下面是一些典型的问题：

1. 谁是Marjolein's的姐姐?
2. 你的小学在哪条街上?
3. Woroboff小姐教哪门课程?

在登录时，计算机随机地选择一条询问用户，并检查用户的回答是否正确。

口令思想的另一个变体是查问-回答。使用查问-回答时，用户在注册时选择好一个算法，比如x2。当用户登录时，计算机显示某个参数，例如7，这时用户需键入49。这种算法可以经常变动，早晚不同、每天不同、并且在不同的终端上也不相同。

物理鉴定

另外一个完全不同的认证方法是检查用户是否有某些"证件"，通常是含有磁条的塑料卡。将卡片插入终端中，系统可以查出卡片所有者。这种方法可以和口令一起使用。因此用户要成功登录，必须(1)拥有卡片，并且(2)知道口令。自动提款机就是这样工作的。

还有一种方法是测量那些难以伪造的物理特征。例如，终端上的指纹或者声波纹读取机可以认证用户身份。(如果计算机不是将给出的指纹与整个数据库相比较，而是根据用户的名字进行查找，则整个过程要快得多。)直接视觉辨认现在还不可行，但终究有一天会实现。

另一种技术是签名分析。用户使用与终端相连的特制笔签名后，计算机与在线存储的已知样本进行比较。更好的方法是不必比较签名，而是比较签名时笔的移动情况，一个优秀的模仿者或许可以模仿你的签名，但是，至于你在在签名时行笔的确切顺序，他就不清楚了。

指长分析也非常实际。当采用这种方法时，每个终端都有一个如图5-21的设备。用户把手插入到这个设备中，他的各个手指的长度被记录下来，并与数据库中存放的数据比较。

图5-21 测量指长的设备。

我们还可以举出更多的例子。但是，下面两个例子说明了一个关键问题。猫和其他的动物会在他们的边界周围撒上一泡尿，这样他们可以很容易彼此区分。假设有人提供能当场进行尿液分析的小设备，然后，每个终端都配置这样一个设备，旁边写着："登录时，请在此排放尿液"。这或许是一个绝对不可突破的系统，但是用户显然难以接受。

下面的方案同样也很难让用户接纳：系统由图钉和小型摄谱仪组成，登录时，用户将拇指按在图钉上，提取一滴血液让摄谱仪分析。至关重要的是，任何一种认证方案都必须为用户所接受。指长测量可能不会引起任何问题，但是，即使是一些非侵入性方法，比如在线存储指纹，许多人也不愿接受。

对策

往往在入侵者闯入并制造极大的损坏之后，计算机装置才会对安全性引起严重关注，他们常常采取很多方法来增加未授权访问的难度。例如，每个用户只允许在指定的某一台终端上登录，并且只能在一周的某几天或者一天中的某几个小时中登录。

拨号电话可按如下方式工作：任何人都可以拨号和登录，但一旦成功登录，系统立即中止连接，并且以某个协商好的号码回呼用户。使用这一方法，入侵者不能从随意一根电话线上进入系统。要进入系统，他只能通过用户的(家庭)电话。并且，不论是否回呼，系统应该至少用10秒钟来检查用户在拨号线上键入的口令。当几次连续的登录失败后，这个时间还要增加，以减少入侵者非法闯入的可能性。在三次登录失败后，电话线被中断10分钟，并通知安全人员。

所有登录信息都应该记录下来。当用户登录时，系统告知他上次登录的时间和终端，以便他检测出可能的非法入侵。

我们还可以设置陷阱来捕获入侵者，例如采用带简单口令的特殊登录名(如登录名guest，口令guest)。当发现任何人以这一名字登录时，立即通知安全专家。其他的陷阱可以是容易发现的操作系统故障或类似的东西，他们是专门为捕获入侵者而设计的。Stoll就曾写过一篇有关陷阱的有趣报道(1989)，利用这陷阱，他捕获了一个潜入某大学计算机系统中窃取军事秘密的间谍。

5.5 保护机制

在前面的几节中，我们看过许多潜在的问题。其中有些是技术性的，有些是非技术性的。下面几节，我们主要讨论操作系统中保护文件和其他信息的一些技术，所有这些技术把策略(谁的数据需要保护，以及禁止谁访问)和机制(如何实现保护)明确地区分开来。策略和机制的分开在(Levin et al.，1975)中讨论。这里，我们的重点主要放在机制，而不是策略上。要了解更详细的材料，请参看(Sandhu，1993)。

某些系统使用一个叫做引用监视器的程序来实现保护。每次访问被保护资源时，系统请求引用监视器检查访问是否合法。 引用监视器查找策略表，作出相应的决定。下面我们介绍引用监视器的运行环境。

5.5.1 保护域

计算机系统中包含有许多有待保护的"对象"，这些对象有可能是硬件(例如CPU、内存段、磁盘驱动器或者打印机)，也有可能是软件(例如进程、文件、数据库或者信号)。

每个对象都有一个唯一的名字，用户通过这个名字来引用对象。并且，对象还有一个有限的、允许进程执行的操作集。例如对文件有READ、WRITE操作，对信号可进行UP、DOWN操作等等。

很明显，我们需要某种方法禁止未授权进程访问对象。此外，保护机制还应该可以限制进程只执行合法操作的子集。比如，进程A有权读取文件F，但不能写F。

为了讨论不同的保护机制，我们有必要引入域的概念。域是(对象，权限)对的集合，每个对标明了一个对象和一个可执行操作的子集，这里权限表示允许执行的操作。

图5-22显示了三个域，并给出了每个域的对象，以及对这些对象允许的操作[读、写、执行]。注意，Printer1同时属于两个域。虽然我们在图中没有画出，但是同一对象可出现在多个域中，并且在不同域中的权限也可以不同。

图5-22 三个保护域。

无论何时，进程都运行在某个保护域中。换句话说，它可以访问某些对象，对每个对象有一定的权限。进程在执行时，可以从一个域切换到另一个域，切换的规则完全独立于系统。

为了使保护域的思想更加具体，我们来看看UNIX。在UNIX中，进程的域用uid和gid来定义，给出一个(uid，gid)对，可以建立一张完整的表，列出所有可以访问的对象(文件，包括用设备文件表示的I/O设备等)以及是否可以读、写、执行这些对象。有相同(uid，gid)对的两个进程有完全相同的访问对象集。具有不同(uid，gid)对的进程可能访问不同的对象集，尽管多数情况下，这些对象集是相互重叠的。

此外，UNIX中每个进程有两个部分：用户部分和内核部分。当进程执行系统调用时，从用户部分切换到内核部分。内核部分和用户部分有不同的访问对象集。例如，内核部分可以存取物理内存中的所有页面，进而访问整个磁盘以及其他保护资源。因此，系统调用导致域切换。

当进程对设置了SETUID或SETGID位的文件执行EXEC操作时，它获得新的有效uid或gid。对于不同的(uid，gid)对，有不同的文件集和有效操作。运行一个有SETUID或SETGID的进程也是域切换，因为这些可用的权限并不相同。

这里会遇到一个重要问题：系统如何记录某个对象属于哪个域呢？至少，你可以想象用一个矩阵来表示，矩阵的行表示域，列表示对象。图5-22对应的矩阵如图5-23所示，给定该矩阵及当前域号，系统可以给出是否能够按特定方式从某指定域中访问对象。

图5-23 一个保护矩阵。

如果意识到域本身也是对象，我们很容易用ENTER操作把域切换包含在矩阵模型中。图5-24重新显示了图5-23，只是增加了三个域作为对象。域1中的进程可以切换到域2，然而，一旦切换后，它不能重新回到域1。在UNIX中，这种情况模拟执行一个SETUID程序。在这个例子中，其他的域切换都不允许。

图5-24 保护矩阵把域作为对象。

5.5.2 存取控制表

实际上，我们很少存储象图5-24中那样的矩阵，这个矩阵非常大并且有很多空项。大多数域都只存取很少的对象，因此存储一个大而空的矩阵浪费了大量的磁盘空间。可是，我们可以使用两种方法：按行或列来存储矩阵，而且只存储非空元素。这两种方法有很大的不同。本节中，我们介绍按列存储。下一节，我们将讨论如何按行存储这个矩阵。

在第一种存储技术中，每个对象被赋予一张排序的列表，其中列出了可以访问该对象的所有域，以及如何进行访问。这张表也称为访问控制表(access control list)，或者叫ACL。如果在UNIX中实现这种技术，我们可以把每个文件的ACL放在一个单独的磁盘块中，并在文件的i-节点中包含这个磁盘块的块号。因为只存储了非空项，全部ACL所需的存储空间要比存储整个矩阵所需的空间少得多。

为了说明ACL的工作原理，我们还是来想象一下它在UNIX中的应用。在UNIX中，域通过(uid，gid)对指定。事实上，ACL在UNIX的作用模型MULTICS中的使用或多或少与我们下面例子中描述的有些类似。因而，这个例子并不是凭空想象的。

假设有四个用户(即uid)Jan,Els,Jelle和Maaike。他们分别属于用户组system,staff,student和student。另外我们还假定一些文件的访问控制表如下：

File0：(Jan，*，RWX)

File1：(Jan，system，RWX)

File2：(Jan，*，RW-)，(Els，staff，R--)，(Maaike，*，R--)

File3：(*，*，R--)

File4：(Jelle，*，---)，(*，*，R--)

括号中的每个ACL项均指定了uid、gid以及对文件允许的访问(读、写、执行)。星号代表所有的uid或gid。这个例子中，File0可以被uid为Jan，gid为任意值的进程读写和执行；File1只能被uid为Jan，gid为system的进程访问。uid等于Jan而gid等于staff的进程可以存取文件File0，但无权访问File1。File2允许uid等于Jan，gid为任意值的进程读写，此外，uid为Els而gid为staff的进程，以及uid等于Maaike而gid为任意值的进程都可以读取该文件。任意进程都能够读取File3。File4比较有趣，只要uid是Jelle的进程都无权访问File4，但是，所有其他进程均可以读取它。利用ACL，我们可以禁止特定的用户或用户组访问某个对象。

UNIX的具体实现方法略有不同。在UNIX中，每个文件都为文件主、文件主所在的用户组以及其他用户提供了三个位rwx，这种方案也属于ACL，只是将其压缩到9位。这是和对象相关联的一张表，给出哪些用户可以存取该对象，以及如何存取该对象。尽管9位的UNIX方案远远没有上述ACL系统那样全面，实际上已经相当适用了，何况，其实现要简单、方便得多。

对象的拥有者随时都能改变对象的ACL。他可以禁止原先允许的访问。唯一问题在于，改变ACL很可能不会影响当前正在使用该对象的用户(即当前打开该文件的用户)。

5.5.3 权限

此外，我们也可以按行来分割矩阵。这时，每个进程都赋予一张该进程可以访问的对象表以及对每个对象允许的操作，即域。这张表称为权限表(capability list)，其中的每一项叫做权限(Dennis和Van Horn，1966；Fabry，1974)。

图5-25是一张典型的权限表。在每个权限中，类型域给出了对象的类型，权限域是一个位图，表明对这种类型的对象允许执行的合法操作，对象域则是指向对象本身的指针(i-节点号)。权限表本身也是对象，可以从其他的权限表中引用，从而很容易实现子域的共享。权限常常按他们在权限表中的位置来引用。某个进程可能要求："在权限2指向的文件中读出1K数据。"这种形式的寻址与UNIX中的文件描述符类似。

图5-25 图5-23中域2的权限表。

显然我们应该防止权限表，或者常常被称作C-表，被用户篡改。有三种方法可用于保护权限表。第一种方法需要带特征位的体系结构。在硬件设计时，每个内存字有一个额外(或特征)位给出了该内存字是否包含有权限。 该特征位不参与算术运算和比较运算，也不用于类似的普通指令中，它只能被运行在内核模式下的程序(即操作系统)修改。

第二种方法是把C-表保存在操作系统内部，进程引用权限时给出权限的槽口号，就象我们上面说到的那样。Hydra(Wulf et al.，1974)就是这样实现的。

第三种方法是把C-表加密后，直接保存在用户空间内。这种方法尤其适用于分布式操作系统。它广泛使用在Amoeba(Tanenbaum et al.，1990)中。

除了依赖于对象的特殊权限，比如读、写和执行，权限往往还有一些适用于所有对象的一般权限。下面就是一般权限的例子：

1. 拷贝权限：为同一对象创建一个新权限。
2. 拷贝对象：创建一个复制对象，使它具有新的权限。
3. 删除权限：从C-表中删除一项，相应对象不受影响。
4. 删除对象：永久地删除对象及其权限。

最后要提的是，在权限系统中，撤销对某对象的访问是很困难的。系统很难找出所有权限并将其收回。事实上，这些权限存储在遍及磁盘的权限表中。一种方法是把每个权限指向一个间接对象，而不是指向对象本身。由于间接对象指向真正对象，系统总能中断他们之间的联系，从而使得权限失效(当一个指向间接对象的权限后来向系统提交时，用户会发现这个间接对象如今指向空对象。)

另一种废除的办法是Amoeba用到的。在Amoeba中，每个对象包含一个很长的随机数，这个随机数同时出现在权限中。在使用权限时，将上述两个数字进行比较。只有在他们一致时才允许操作。对象的拥有者能够修改该对象的随机数，从而使容量失效。这两种方案都不允许选择性撤销，也就是说，它不能只收回某些用户的权限。

5.5.4隐藏通道

尽管使用了访问控制表以及权限表，安全泄漏还常常发生。这一节，我们讨论一个这方面的问题。它是

Lampson(1973)提出的。

Lampson的模型中有三个进程，主要适用于大型分时系统。第一个进程是客户进程，它向第二个进程，即服务器进程，请求执行某项操作。客户进程和服务器进程并不完全相互信任。例如，服务器进程帮助客户进程填写税务单。客户进程担心服务器进程会偷偷地记录下这些数据，然后把这些数据出售给别人。而服务器进程可能怀疑客户进程会窃取其税务程序。

第三个进程是协调进程。协调进程和服务器进程合谋窃取客户进程的秘密数据。协调进程和服务器进程往往属于同一用户。这三个进程显示在图5-26中。我们希望设计一个安全系统，使得服务器进程不可能向协调进程泄漏从客户进程那儿获得的信息。Lampson把它叫做限制问题。

图5-26 (a)客户、服务器和协调进程。 (b)封装后的服务器进程仍然可通过隐藏通道向协调进程泄露信息。

系统设计者的目标是封装或限制服务器进程，使得它无法向协调进程传送信息。使用保护矩阵，我们很容易保证服务器进程与协调进程之间不可能通过文件进行通信，也就是说，服务器进程不可能把数据写到一个协调进程能够读取的文件中。此外，我们还可以保证两个进程不可能通过系统的进程间通信机制实现通信。

不幸的是，他们可以采用其他巧妙的通道进行通信。比如，服务器进程可用下列方式传送二进制位流。为了传送1，在一段时间内，它不停地进行运算。而传送0时，服务器进程睡眠同样的时间段。

然后，协调进程可以仔细地监视它自己的反映时间，从而检测出服务器进程传送的位流。一般说来，服务器进程在传送0时，协调进程的反映速度要比传送1好得多。这种通信通道称为隐藏通道(covert channel)，如图5-26(b)所示。

当然，隐藏通道是一个噪音通道，它含有大量的外部信息。但是通过错误校验码我们可以在噪音通道上可靠地传送信息(例如海明威码，甚至是更复杂的错误校验码)。错误校验码的使用降低了隐藏通道原来就很低的带宽，然而足以泄漏重要信息。显然，任何基于对象和域矩阵的保护模型都无法防止这类泄漏。

调节CPU的使用并不是唯一的隐藏通道。页面率也可以进行调节(很多页面错误表征1，无页面错误表征0)。事实上，几乎所有按时间降低系统性能的方式都可用作隐藏通道。如果系统提供锁文件的方法，服务器进程可以给某些文件加锁以表示1，而释放锁表示0。在有些系统中，进程能够检测出文件的锁状态，尽管它可能没有权力访问该文件。

请求和释放专用资源(磁带驱动器、绘图仪等等)也可用作隐藏通道的信号。在发送1时，服务器进程请求该资源。在发送0时，服务器进程释放该资源。在UNIX中，服务器进程创建一个文件表示传送1，删除该文件表示送0；协调进程用ACCESS系统调用检测该文件是否存在。尽管协调进程无权访问该文件，ACCESS调用依然可行。非常遗憾，还存在着许许多多其他的隐藏通道。

Lampson还提到一种向服务器进程的拥有者泄露信息的办法。假定服务器进程有权通知其拥有者它为客户进程服务的工作量，以便对客户计费。打个比方，如果实际计算出的帐单为100美元，而客户的收入为53K美元，则服务器进程可以把100.53报告给其拥有者。

试图找出所有的隐藏通道是相当困难的，更不要说去堵塞他们了。实际上，我们几乎毫无办法。引入一个随机产生页面错误的进程，或者是利用其他降低系统性能的方法来减少隐藏通道的带宽也根本不可取。

5.6 MINIX文件系统概述

象所有的文件系统一样，MINIX文件系统也必须解决我们上面讨论的问题。它必须为文件分配空间和释放空间、记录磁盘块和空闲空间、提供某种方法以防止文件被未授权使用等等。在本章的剩下部分，我们将仔细研究MINIX的文件系统，看看它是如何实现这些目标的。

在本章的前半部分，为了更具有普遍性，我们反复地引用了UNIX，而不是MINIX操作系统。但两者的外部界面实质上是相同的。现在我们把注意力集中到MINIX文件系统的内部设计上来。要了解UNIX的内核，请参看Thompson(1978)、Bach(1987)、Lions(1996)和Vahalia(1996)。

MINIX文件系统不过是在用户空间中运行的一个大型C程序(图2-26)。读写文件时，用户进程向文件系统发送一条消息，文件系统进行相应处理后，返回结果。实际上，MINIX文件系统可以看成是和调用进程在同一台主机上运行的网络文件服务程序。

如此设计有深刻的含意。一方面，文件系统可以独立于MINIX的其他部分进行修改、调试和测试。另一方面，我们可以很方便地把整个文件系统移植到任何带有C编译器的计算机上，在那台机器上进行编译，并把它作为一台独立的类UNIX远程文件服务器。我们要做的唯一修改是消息的发送和接收方式。在不同的系统中，消息的发送和接收方式往往并不相同。

在下面的几节中，我们将纵观一下文件系统设计中的许多关键领域。我们特别要介绍消息、文件系统布局、位图、i-节点、块高速缓存、目录与路径、文件描述符、文件锁以及设备文件(管道)。在研究了上述内容之后，我们将给出一个简单的例子。通过跟踪用户进程执行READ系统调用的过程，说明各个部分是如何结合在一起运行的。

5.6.1 消息

文件系统接收39种消息请求。除两种外都用于MINIX的系统调用。这两个异常消息是MINIX的其他部分生成的。在用于系统调用的消息中，有31种是从用户进程中接收到的，而另外6种消息用于这样的系统调用，他们首先被内存管理器所处理，然后内存管理器调用文件系统完成一部分工作。文件系统还处理两种异常消息。这些

消息如图5-27所示。

图5-27 文件系统消息。

文件系统的结构基本上与内存管理器和所有的I/O任务一样。文件系统的主循环程序不断地等待消息。当收到消息后，它首先提取消息的类型，然后以其为索引查找文件系统中处理各类消息的过程指针表。随后调用相应过程，进行处理后，返回状态值。文件系统再把回答消息发送给调用进程，然后回到循环的开始，等待下一条消息的到来。

5.6.2 文件系统布局

MINIX文件系统是一个逻辑的、自包含的实体，它含有i-节点、目录和数据块。MINIX文件系统可以存储在任何块设备中，例如软盘或一个硬盘分区。MINIX的文件系统都有相同的布局。图5-28所示是一个128个i-节点和1K块的360K软盘的布局。更大的系统，或者是那些有不同的i-节点数和块大小的系统，也同样由这6部分顺序组成，但是各部分之间的相对大小可能不一样。

图5-28 最简单磁盘的磁盘布局，该磁盘为360K软盘，有128个i-节点，块大小为1K(即两个连续的512字节扇区作为一个块)。

每个文件系统都以引导块开始，引导块中包含有可执行代码。启动计算机时，硬件从引导设备将引导块读入内存，转而执行其代码。引导块代码开始操作系统本身的加载过程。一旦系统启动之后，引导块不再使用。并非每个磁盘驱动器均可用作引导设备，但是为了保持结构的一致，每个块设备都为引导块代码保留一块。这种方法最多不过浪费了一个块。为防止硬件从非启动设备上启动，在将可执行代码写入引导设备的引导块中时，在引导块的已知位置处写入魔数。从一个设备上启动时，硬件(实际上是BIOS代码)首先检测魔数是否存在。若不存在，则拒绝把引导块载入内存，这样可以防止把垃圾用作引导程序。

超级块(super-block)中含有文件系统的布局信息，如图5-29所示，它的主要功能是给出文件系统不同部分的大小。如果给定块大小和i-节点数，我们很容易算出i-节点位图的大小和存放i-节点所需的块数。例如1K的块，每个位图块有1K字节(8K位)，可以记录8192个i-节点的状态(实际上第一块只能处理8191个i-节点，因为0号i-节点并不存在，但我们在位图中也为它保留一位)。10000个i-节点，要用到两个位图块。每个i-节点占64字节，1K的块中可以有16个i-节点。如果有128个可用的i-节点，则需要8个磁盘块来存放。

图5-29 MINIX超级块。

后面我们会详细地解释区段和块之间的区别。但是此时，读者只要了解磁盘存储区可以以区段为单位进行分配，而每个区段可以包含1、2、4、8个，或一般情况下，2n个磁盘块。区段位图按区段，而不是块来记录空闲存储区。MINIX用到的所有标准软盘中，区段大小和块大小是一样的(均为1K)，因此在这些设备上，可以近似把区段看成是块。在本章后面部分详细讨论存储分配之前，在看到"区段"时，只要把它当作"块"就可以了。

可以注意到，每个区段包含的块数并没有存放在超级块中，我们并不需要这一数据。我们存放的是底为2，区段数除以块数所得值的对数。根据它，可以知道从区段转换成块或者从块转换成区段要移位的次数。例如，每个区段中含有8个块，log28=3，因此要找到包含第128块的区段，我们可以把128右移3位，得到16。

区段位图中只包含数据区段(即位图和i-节点用到的块并不在该位图中)。第一个数据区段在位图中用区段1表示，同i-节点位图一样，区段位图中第0位也未使用，因此第一个区段位图块只能映射到8191个区段，以后的每块可以映射到8192个区段。考察一下新格式化磁盘的位图，可以发现i-节点和区段位图中均有2位为1。一位是不存在的0号i-节点和0号区段，而另一位是设备根目录使用的i-节点和区段，在文件系统创建时，根目录自动存在。

我们还可以注意到，超级块中的信息冗余。由于我们有1K空间可用于存放超级块的信息，因此我们可以事先按不同的形式算出所需的信息，而不必在使用时重新计算。例如，磁盘上的第一个数据区段的区段号，可以从块大小、区段大小、i-节点数以及区段数计算得到。但是，直接把它存放在超级块中要方便得多。超级块中的剩下部分总归是要浪费的，我们还不如用它来存储一些有用的数据。

在MINIX启动时，根设备中的超级块被读入内存中，同样，安装其他文件系统时，他们的超级块也读入内存。内存的超级块表中有些域不出现在磁盘上，其中包括指定设备打开方式和字节顺序的标志等，此外，内存的超级块表还含有指向位图第一个空闲位的域，使用这个域可提高访问速度，以及表征超级块所属设备的一个域。

在磁盘用作MINIX文件系统之前，必须具有图5-28所示的结构。实用程序mkfs可用来创建文件系统。我们可以通过象

```
mkfs  /dev/fd1  1440
```

的命令行调用该程序，在驱动器1中的软盘上创建1440个块的空文件系统，这一命令还在超级块中写入魔数，表明该文件系统是一个有效的MINIX文件系统。此外，我们也可以通过一个原型文件来调用mkfs，其中列出要包含在新文件系统中的目录和文件。MINIX文件系统经过改进，有些方面(比如i-节点大小)与早期版本不同，魔数还可以表明创建文件系统的mkfs的版本。从而处理他们之间的不同。MOUNT系统调用检查超级块中魔数和其他信息，可以拒绝安装不是MINIX格式的文件系统，例如MS-DOS的文件系统。

5.6.3 位图

MINIX用两个位图来记录空闲i-节点和空闲区段(见图5-29)。当文件被删除时，很容易算出哪一个位图块

包含了所释放i-节点的相应位，利用通常的高速缓存机制查找该块，一旦找到，相应于被释放i-节点的那一位清0。区段在区段位图中的释放也类似。

逻辑上，在创建文件时，文件系统必须在位图块中查找第一个空闲i-节点，把它分配给这个新创建的文件。然而，超级块在内存的拷贝中有一个域指向第一个空闲i-节点，因此不必进行查找，在该空闲i-节点分配使用后，就需要修改指针，使它指向下一个空闲i-节点，往往是下一个或者较近的一个节点。同样地， i-节点被释放后，检查这个i-节点是否位于第一个空闲i-节点的前面，若是，则需要修改指向第一个空闲i-节点的指针。如果磁盘上的所有i-节点全被使用，查找函数返回0，这也是0号i-节点未使用的原因(即它可以用于表明查找失败)(在mkfs创建新文件时，它把0号i-节点清零，并把位图中的最低位设置为1，防止文件系统分配0号i-节点)。上面所讲的同样也适用于区段位图。逻辑上，申请空间时，需要在区段位图中查找第一个空闲区段，但是超级块的内存拷贝保存了指向第一个空闲区段的指针，因而消除了很多顺序查找位图的麻烦。

有了这些知识后，现在我们可以解释区段和块之间的不同了。使用区段的目的是：确保同一文件的所有磁盘块都位于同一个柱面上，从而改进顺序读取文件的性能。我们采用了一次分配多个块的方法。假设块大小为1K，而区段大小为4K，区段位图中记录了区段，而非块的使用情况。一个20M的磁盘有5K个4K大小的区段，因此其区段位图需要5K位。

文件系统的大多数部分都以块为单位进行操作的。磁盘每次传送一块，高速缓存也按照块进行处理，系统中只有记录物理磁盘地址的一小部分(例如区段位图和i-节点)需要知道区段的存在。

在开发MINIX文件系统的过程中，我们不得不做一些设计上的决策。1985年，当MINIX还处于构思阶段的时候，磁盘容量很小，许多用户只有软盘。在V1文件系统中，我们决定把磁盘地址限制在16位，这样能够把大多数地址存放在间接块中。16位的区段号和1K大小的区段只能寻址64K个区段，从而将磁盘容量限制为64M。那时，这可是一个相当大的容量，我们当时还考虑到，需要扩充磁盘容量时，很容易把区段大小转换为2K或者4K，而不必改动块的大小。16位的区段号还容易使得i-节点的大小保持为32字节。

由于MINIX不断发展，同时大磁盘更加普及，因此有必要对MINIX的文件系统进行修改。许多文件长度小于1K，因此增加块大小意味着浪费磁盘带宽和读写几乎是空的块，也意味着把它存放在高速缓存时浪费了宝贵的主存。原本我们是可以增加区段大小的，但是大的区段浪费了更多的磁盘空间，何况我们还希望保持对小磁盘操作的高效率。当然，另一种合理的解决方案是在不同大小的设备中使用不同大小的区段。

但最终我们决定把磁盘指针的大小增加到32位。这使得在块大小和区段大小均为1K时，MINIX V2版本的文件系统可以处理容量达4TB的设备。作出这种决定的另外一个原因与i-节点的更改有关。由于i-节点中包含的内容增加了，为此我们把i-节点的大小也增加到64个字节。

区段的使用也带来了意料不到的问题。我们用一个简单的例子来阐述，考虑4K的区段和1K的块，假设文件长度为1K，这时文件被分配了一个区段，1K和4K之间的磁盘块中含有垃圾(以前用户的残留数据)，但是这不会对我们造成不利。因为在i-节点中文件大小清楚地标记为1K。事实上，包含垃圾的磁盘块根本不会读到高速缓存中，因为读操作是以块，而不是区段为单位进行的。超出文件尾的读操作总是返回0，不包含任何数据。

假设现在有人将文件指针定在32768字节并写入1个字节数据，文件长度变为32769字节。随后读取1K后面的数据将返回该块以前的内容，从而形成一个很大的安全缺口。

解决这一问题的方法是，执行写操作时，检查写入位置是否超出了文件尾。若超出，则将文件的最后一个区段中所有还未分配的块清空。尽管这种情况很少发生，然而我们的代码必须进行处理，这使得MINIX文件系统更加复杂。

5.6.4 i-节点

MINIX中i-节点的布局如图5-30所示，它几乎与标准UNIX的i-节点相同。磁盘区段指针是一些32位的指针，总共有9个这样的指针：7个直接的，2个间接的。MINIX的i-节点占64个字节，这也同标准UNIX的i-节点一样。有一个未使用的空间可以用于第10个(三次间接)指针，我们在MINIX文件系统的标准版本中还不支持这一指针。MINIX i-节点中的存取时间、修改时间以及i-节点的修改时间都和标准UNIX一样。i-节点的修改时间在大多数文件操作时都要进行修改，只有读文件除外。

图5-30 MINIX i-节点。

在打开文件时，首先要找到文件的i-节点，并把它载入内存的inode表中，直至关闭前，它一直保存在内存中。内存的inode表中有一些域不出现在磁盘上，例如，i-节点所在设备的设备号以及在该设备上的i-节点号，通过这两个值文件系统可以知道在内存中数据修改后，要将这些数据写到何处。每个i-节点还有一个计数器，当文件多次打开时，在内存中只保存一个i-节点拷贝，但是每次打开该文件，计数器加1，每次关闭该文件，计数器减1。只有在计数器减到0时，才将i-节点从inode表中删除。若i-节点自上次调入内存之后被修改过，则要将它写回磁盘。

文件i-节点的主要功能是给出文件数据块所在的位置。前7个区段号就放在i-节点结构之中。对于MINIX标准发行版，区段大小和块大小均为1K，因此小于7K的文件不必使用间接块。如果文件长度超过7K，就要使用间接区段。MINIX中采用了图5-10所示的方案，只是它只用到了一次间接块和两次间接块。若块大小和区段大小均为1K，区段号为32位，则一次间接块含有256项，可以表示1/4M的存储区，两次间接块指向256个一次间接块，因此可以存取长达64M的文件。MINIX文件系统中文件的最大长度为1G，我们可以使用三次间接块或修改区段大

小来存取大于64M的文件。
i- 节点中还包含有模式信息，它给出了文件的类型(正规文件、目录、块设备文件、字符设备文件或管道)以及保护标志、SETUID位和SETGID位。i-节点的链接数目域记录了有多少个目录项指向这个i-节点，因此文件系统知道什么时候该释放文件的存储区。我们不应当把它与打开文件计数器(只出现在内存的inode表中，不在磁盘上)相混淆，后者指出了文件被打开的次数，而且往往是被不同的进程所打开。

5.6.5 块高速缓存

MINIX使用块高速缓存来改进文件系统性能。高速缓存用一个缓冲数组来实现，其中每个缓冲区由包含指针、计数器和标志的头以及用于存放磁盘块的体组成。所有未使用的缓冲区均使用双链表，按最近一次使用时间从近到远的顺序链接起来，这如图5-31所示。

图5-31 块高速缓存使用的链接表。

为了迅速判断某一块是否在内存中，我们使用了哈希表。所有缓冲区，如果它所包含块的哈希代码为k，在哈希表中用第k项指向的单链表链接在一起。哈希函数提取块号低n位作为哈希代码，因此不同设备的块可以出现在同一哈希链之中。每个缓冲区都在其中某个链中。在MINIX启动，初始化文件系统时，所有缓冲区均未使用，并且全部在哈希表第0项指向的单链表中。这时，哈希表其他项均为空指针。但是一旦系统启动完成，缓冲区将从0号链中删除，放到其他链中。

文件系统需要一块时，它调用过程get_block，计算该块的哈希代码，搜索相应链。get_block被调用时，有两个参数：设备号和块号。这两个值与缓冲区链中对应域相比较，如果找到包含这一块的缓冲区，则缓冲区头中标志块使用次数的计数器加1，并返回指向该缓冲区的指针。如果在哈希表中未找到这样的块，我们可以使用LRU链中的第一个缓冲区。LRU链中的缓冲区必然不在使用中，因而它包含的块可以被换出内存，以释放这个缓冲区。

如果选定某一块调出内存，这时需要检查块头中另一个标志，看它在上次读入内存之后是否修改过。若修改过，就重新写回磁盘，然后文件系统向磁盘任务发送一条消息，要求读入新块，之后文件系统被挂起，直到块读入后才继续运行，将指向该块的指针返回给调用进程。

请求块的过程完成任务后，它调用另一个过程put_block释放该块。正常情况下，块在读入后立即被使用并释放。但也有可能在它被释放前出现其他对该块的请求，因此put_block仅仅是将使用计数器减1。当使用计数器减到0时，才将它放到LRU链中。否则，我们就让它保留在那儿。

在put_block的参数中，有一个给出被释放块的类型(例如i-节点、目录、数据)。对于不同类型的块，需要作不同的考虑：

1. 把该块放在LRU链头部还是尾部?
2. 是否把该块(如果修改过)立即写回磁盘?

最近不可能使用的块，比如超级块，放在LRU链的前端。这样，如果下次需要一个空闲缓冲区，可以立即使用之。所有其他块都按真正的LRU方式放在LRU链的尾部。

修改的块只在下面两种情况下写回磁盘：

1. 它到达LRU链前端并被换出。
2. 执行了SYNC系统调用。

SYNC并不遍历LRU链，相反，它查找高速缓存中的缓冲区数组。缓冲区即使未被释放，如果它被修改过，SYNC也可以找到它，并修改它在磁盘上的副本。

但是，有一种情况比较特殊。修改过的超级块要立即写回磁盘。在早期的UNIX版本中，安装文件系统时要修改超级块，我们把修改后的超级块信息立即写回磁盘，可以减少在系统崩溃时文件系统被破坏的可能性。而现在的MINIX版本中，超级块不再被修改，因此那些用于把超级块立即写回磁盘的代码已经过时了。在标准配置中，其他块都不立即写回。但是，若修改系统配置文件include/minix/config.h中ROBUST的缺省定义，然后重新编译文件系统，则可以实现i-节点、目录、间接块和位图块在释放时立即写回磁盘。之所以这样，是为了使文件系统更加健壮；其代价是运行速度下降。这种处理是否有效，我们并不清楚。但是，如果断电，不管丢失的是i-节点块还是数据块，总是件使人头疼的事。

我们还可以看到，块中表明块已修改过的标志是由文件系统中请求和使用该块的过程设置的。get_block和put_block过程只关心对链表的操作,他们不知道文件系统的哪个过程需要该块,更不知道它为什么需要该块。

5.6.6 目录和路径

在文件系统中另一个重要的子系统是目录和路径名的管理。许多系统调用，例如OPEN，都以文件名作参数。实际需要的是文件的i-节点，因此，文件系统需要在目录树中查找文件，找到相应的i-节点。

MINIX目录由包含16个字节目录项的文件组成。目录项的头两个字节构成16位的i-节点号，剩下的14个字节是文件名，这一点和我们在图5-13中见到的传统UNIX目录项相同。为了查找路径/usr/ast/mbox。文件系统首先在根目录中查找usr，然后在/usr中查找ast，最后在/usr/ast中查找mbox。每次只查找路径的一个部分，如图5-14所示。

唯一麻烦的是遇到被安装的文件系统。在MINIX和其他类UNIX系统中，通常配置有一个小的根文件系统，根文件系统中含有用于启动和进行基本维护的文件。此外，配置中还有很多其他文件，包括用户的目录，他们

都存放在另一个设备上，该设备安装至/usr目录下。现在我们可以来介绍安装是如何实现的了。当用户在终端上键入命令

mount /dev/hd2c /usr

时，包含在硬盘第2个分区上的文件系统被安装到根文件系统的/usr目录下。安装前和安装后的文件系统如图5-32所示。

图5-32 (a)根文件系统。 (b)未被安装的文件系统。 (c) (b)的文件系统安装到目录/usr后的效果。

安装成功后，在/usr的i-节点的内存拷贝中设置了一个标志位，表明有设备已安装到这个i-节点上，这就是关键所在。MOUNT调用还把新安装的文件系统的超级块调入内存中的超级块表中，并设置其中两个指针。此外，系统还把被安装文件系统的根目录i-节点也放在内存的inode表中。

在图5-29中，我们看到，被安装文件系统的超级块中有两个域。第一个是i-node-of-the-mounted-file-system域，指向被安装文件系统的根目录i-节点。其次是i-node-mounted-on域，指向该文件系统所安装到的i-节点，在这里是/usr的i-节点。这两个指针把被安装的文件系统和根文件系统链接起来[如图5-32(c)中虚线所示]，使得被安装的文件系统能够正常工作。

在查找路径/usr/ast/f2时，文件系统在/usr的i-节点中发现一个标志，知道它需要在安装到/usr的文件系统的根目录i-节点上继续查找。问题在于，它如何能找到这个i-节点呢？

答案很显然的。系统搜索内存中所有的超级块，直至找到i-node-mounted-on域指向/usr的那一块，它必然是文件系统安装到/usr上的超级块。一旦它找到了这个超级块，便很容易沿着另一个指针找到被安装文件系统的根目录i-节点。现在文件系统可以继续查找了。在图5-32所示的例子中，它在硬盘第2个分区的文件系统的根目录下继续查找ast目录。

5.6.7 文件描述符

文件打开时，文件描述符被返回给用户进程。其后，READ、WRITE调用都使用文件描述符对文件进行操作。这一节，我们介绍一下文件系统中是如何来管理文件描述符的。

象内核和内存管理器一样，文件系统也在其地址空间内维护进程表的部分内容。其中有三个域需要特别注意：前两个是指向根目录i-节点的指针和指向工作目录i-节点的指针。象图5-14中的路径查找总是从其中一个目录开始，它取决于路径是绝对路径还是相对路径，执行CHROOT和CHDIR系统调用可以改变这两个指针，使他们指向新的根目录和新的工作目录。

进程表中的第三个域是以文件描述符号为下标的数组，根据文件描述符可以在该数组中找到适当的文件。粗略地看，我们把数组中的第k个项指向描述符为k的文件的i-节点就够了。毕竟，在文件打开时，只需要把它的i-节点调入内存，并保存在其中直至文件关闭，所以说，这种方法肯定能奏效。

很不幸的是，这种简便方法并不可行。事实上，在MINIX中(同UNIX一样)，文件能够共享。每个文件都有一个32位的数字表征下一个读写的字节位置，这个数字叫做文件位置，每当执行系统调用LSEEK时进行修改。我们可以把问题简单地描述成："文件位置应该如何存放呢？"

第一种可能是把它放在i-节点中。然而，如果两个或更多的进程同时打开同一个文件，他们都应该有自己的文件指针。一个进程的LSEEK操作影响到另一个进程的读操作是不可思议的。因此，文件位置不能存放在i-节点中。

那么把它存放在进程表中又如何呢？为什么我们不使用一个与文件描述符数组相平行的数组来给出文件当前位置呢？这个想法也不可取，但是原因就更微妙了。问题来自FORK系统调用的语义。当生成一个进程时，父进程和子进程必须共享打开文件的指针。

为了更好地理解这个问题，我们考虑一个其输入重定位到某文件的shell脚本。当该shell生成第一个子进程时，其标准输出文件的位置为0 。然后，子进程继承该位置，输出1K数据。在子进程结束之后，共享文件位置应该为1K。

假设shell读取更多的shell脚本，并且生成另一个子进程，第二个子进程必须从shell中继承1K的文件位置，这样，第二个子进程接着在第一个子进程的位置之后输出。如果shell不和子进程共享文件位置，那么第二个子进程可能重写第一个子进程的输出，而不是在第一个子进程的输出之后追加数据。

所以，我们也不能把文件位置放在进程表中，它必须真正地实现共享。在MINIX中，解决这个问题的方法是引入一个新的共享表filp，其中包含了所有的文件位置。filp的使用见图5-33所示。通过真正地共享文件位置，FORK语义能正确地实现了，shell脚本也能正常工作了。

图5-33 父进程与子进程共享文件位置。

尽管filp表中真正必须包含的项是共享文件的位置，我们也可以把指向文件i-节点的指针放在其中。这样，进程表中的文件描述符数组只需包含指向filp项的指针。每个filp项中还包含着文件模式(允许位)、一些表明文件是否以特殊模式打开的标志、以及使用这个filp项的进程计数器。文件系统可以判断最后一个使用它的进程是否终止，若终止，则释放这个filp项。

5.6.8 文件锁

在文件系统的管理中，另外还有一种情况要使用专门的表结构，这就是文件锁。MINIX支持POSIX建议性文

件锁(advisory file locking)的进程间通信机制，允许文件的任何一个或多个部分标记为已锁。MINIX操作系统不强制进行锁操作，但是进程在执行可能与其他进程相冲突的操作前，可以对文件锁进行查询。

我们之所以为锁提供一个单独的表结构，原因与上节讲到的filp表相同。一个进程可以设置多个锁，而一个文件的不同部分可以被不同进程上锁(但是，不同的锁不能重叠)。因此进程表和filp表均不适于记录文件锁。因为文件中可以有多个锁，所以文件的i-节点也不宜用来记录锁。

MINIX用另外一张表，即file_lock表，来记录所有锁。表中每一项都包含锁类型，表明对该文件加的是读锁还是写锁。此外，file_lock各项中还有加锁进程的进程号、指向被锁文件i-节点的指针以及加锁范围的第一个字节和最后一个字节在文件中的偏移。

5.6.9 管道和设备文件

管道和设备文件与普通文件有很大的不同。当进程读写一个磁盘文件时，该操作最多在几百个毫秒内就能完成。最糟糕时，也不过需要2到3次磁盘访问。而读管道时，情况就不同了：如果管道为空，读进程可能要等待另一个进程把数据写到管道中，这可能要几个小时。类似地，当从终端上读取数据时，进程必须等待，直至用户输入。

因此，文件系统通常的规则，即处理请求直至完成，就不再可行了。我们有必要把这些请求挂起，在以后再重新执行。当进程试图从管道中读写数据时，文件系统立即检查管道的状态，看该操作是否能够完成。若能，就等待它执行完成。否则，文件系统把系统调用的参数记录在进程表中，以便在时机到来后继续执行之。

文件系统并不需要采取特殊方法挂起调用进程，它要做的无非是不发送回答信号，让等待回答信号的调用进程自动阻塞。挂起进程后，文件系统回到主循环中等待下一个系统调用。一旦别的进程修改了管道的状态使得挂起进程能够运行结束，则文件系统设置某个标志位。这样，在下一次循环时，文件系统从进程表中提取挂起进程的参数，继续执行前面的系统调用。

终端和其他字符设备文件略有不同。每个设备文件的i-节点中都含有两个数字：主设备号和次设备号。主设备号给出了设备类型(例如RAM盘、软盘、硬盘、终端)，它可用作文件系统表的索引，而文件系统表将主设备号映射到相应的系统任务号(即I/O驱动程序)。事实上，主设备号指定了调用哪个I/O驱动程序。次设备号作为参数传递给驱动程序，指明使用的设备，例如终端2或驱动器1。

在有些情况下，尤其是终端设备中，次设备号包含了一些设备信息。例如，MINIX的主控制台/dev/console的主、次设备号分别为4、0。虚拟控制台也使用同样的驱动程序处理，如/dev/ttyc1(主、次设备号为4、1)和/dev/ttyc2(主、次设备号为4、2)等。串行终端需要使用不同的底层软件，这些设备，即/dev/tty00和/dev/tty01，的设备号为4，16和4，17。类似地，网络终端使用伪终端驱动程序，也用到不同的底层软件。在MINIX中，网络终端设备包括ttyp0、ttyp1等，其设备号为4，128和4，129。这些伪设备每个都有一个与之关联的设备，比如ptyp0、ptyp1等，与他们相应的主、次设备号分别是4，192和4，193等。之所以选择这些数字是为了方便驱动程序调用每组设备所需要的底层函数。对任何人来说都不大可能在一个MINIX系统中配备192台以上的终端。

当进程要从设备文件中读取数据时，文件系统从文件的i-节点中提取主设备号及次设备号，并根据主设备号获得相应的任务号。接着，文件系统向该任务发送一条消息，其中以次设备号、要执行的操作、调用进程的进程号、缓冲区地址和要传送的字节数为参数。消息的格式与图3-15中相同，只是没有用到POSITION项。

如果驱动程序能够立即执行该操作(例如，在终端上已输入了一行)，它把数据从自己的内部缓存拷贝到用户缓冲区中，然后向文件系统发回响应消息，告知执行完毕。文件系统再向用户发一个回答信号。这样，这个调用结束。需要注意的是，驱动程序并不把数据拷贝到文件系统中。从块设备文件中读取的数据要经过块高速缓存，但是从字符设备文件中读取数据则没有这个必要。

如果驱动程序不能进行这个操作，它把消息参数记录在自己的内部表中，立即向文件系统发送一个响应，说明该系统调用无法完成。这种情况同文件系统发现某人试图从空管道中读取数据一样。文件系统记录下进程被挂起的事实，随后等待下一条消息。

在获得足够完成调用的数据后，驱动程序将这些数据传送到被阻塞用户程序的缓冲区中，向文件系统发送消息报告它完成的操作。文件系统再向用户程序发送消息使其解除阻塞，并告知已传送的字节数。

5.6.10 一个例子：READ系统调用

不久我们会看到，文件系统的很多代码都用于系统调用的执行。因此，我们有必要作个小结。我们来看看READ这个最重要的系统调用是如何实现的。

当用户程序执行语句

```
n=read(fd, buffer, nbytes);
```

读取普通文件时，库过程read被调用。它首先创建一条消息，其中包含fd、buffer、nbytes等参数，以及表示READ类型的消息码。然后将这条消息送给文件系统，并阻塞以等待文件系统的响应。文件系统在收到消息后，以消息类型为下标查找过程表，调用相应过程处理读请求。

该过程从消息中提取出文件描述符，由此找到相应的filp项以及要读取文件的i-节点(参见图5-33)。接着，读请求被分成几个段，每段对应一块。例如，如果当前的文件位置为600字节，要读取的数据长度为1K字节。那么，读请求将分成两个部分，分别是从600到1023字节和从1024到1623字节(假定块大小为1K字节)。

对于上述各段，依次检查他们的相关块是否在高速缓存中。如果不在，文件系统选择最久未使用的块，把

它调出内存并收回其缓冲区，如果这一块在上次调入之后修改过，文件系统向磁盘任务发送一条消息，将其写回磁盘，然后，文件系统还要请求磁盘任务将所需的块读入。

如果要读入的块已在高速缓存中，那么文件系统向系统任务发送一条消息，请求它把数据拷贝到用户缓冲区中(即从600到1023字节的数据拷贝到用户缓冲区起始位置,而从1024到1623字节的数据拷贝到从424字节开始的用户缓冲区中)。在拷贝之后，文件系统向用户程序送出响应消息，告知拷贝的字节数。

在用户程序收到响应后，库函数read提取响应代码，作为函数值返回给调用进程。

这里还有额外的一步，其实它并不是READ调用的一部分。 如果对块设备执行的是读操作，并且满足一些其他条件，文件系统在读出数据，送回响应后，将继续读取下一块。顺序读取文件非常普遍，因此可以设想下一次读操作将请求文件的下一块，于是提前做这一操作，当实际需要时，所需的磁盘块就已经在高速缓存中了。

5.7 MINIX文件系统的实现

MINIX的文件系统相对较大(多于100页C代码)，但是非常直观。执行系统调用的请求到达后，进行相应的处理，然后送回回答信号。下面几节中，我们逐个分析文件，指出其中的关键之处，代码本身也含有许多注释以便于读者阅读。

在分析MINIX其他部分的代码时，我们首先分析进程的主循环，接着分析处理不同消息类型的函数。我们按照不同的方式来分析文件系统：首先我们研究主要的子系统(高速缓存管理、i-节点管理等等)，接着我们讨论主循环和各种文件操作的系统调用，之后是关于目录操作系统调用的研究，最后我们介绍其他的系统调用。

5.7.1 头文件和全局变量

同内核和内存管理器一样，文件系统中使用的数据结构和表都定义在头文件之中，有些数据结构放在目录include/及其子目录下的系统头文件中。例如，include/sys/stat.h定义了系统调用向其他程序提供i-节点信息的格式，而include/sys/dir.h中定义了目录项结构。这两个文件是POSIX所需要的。此外，全局配置文件include/minix/config.h中的许多定义也会影响文件系统。例如，ROBUST宏表明是否将重要的文件系统数据结构在修改后立即写回磁盘。NR_BUFS和NR_BUF_HASH则控制块高速缓存的大小。

文件系统头文件

文件系统本身的头文件放在文件系统的源目录/src/fs/中，在看过MINIX系统的其他部分后，许多文件名我们都很熟悉了。文件系统的主要头文件fs.h(19400行)和src/kernel/kernel.h及src/mm/mm.h相似，它包含了文件系统的C源程序中引用的其他头文件，如const.h、type.h、proto.h和glo.h等等。下面我们逐个分析这些头文件。

const.h(19500行)中定义了整个文件系统中使用到的一些常数，例如表长度和标志位等等。MINIX有一定的开发历史，其早期版本有不同的文件系统。现在的MINIX同时支持V1版本和V2版本的文件系统，因此用户可以存取早期MINIX版本写的文件。文件系统超级块中包含有魔数，根据魔数操作系统可以判别文件系统的版本号。常数SUPER_MAGIC和SUPER_V2定义了这些魔数。对于老版本的支持并不只是理论上的研究，常常是软件人员实现新版本时主要考虑的因素。系统设计者必须决定花多大的精力来方便老版本的用户。我们会看到在MINIX文件系统中，有几个地方支持老版本是很麻烦的事。

type.h(19600行)同时定义了V1版本和V2版本的i-节点在磁盘上的结构。V2版本i-节点的长度是V1版本的两倍。V1版本的文件系统是为不带硬盘驱动器和360-KB磁盘的系统而设计的。V2版本中提供了UNIX系统中有的三个时间域。V1版本的i-节点中只有一个时间域，但是可用STAT或FSTAT来模拟，STAT或FSTAT返回一个包含有这三个域的stat结构。支持这两种版本的文件系统稍微有些困难，这一点我们在19616行的注释中提到。老版本的MINIX将gid_t类型定义为八位的值，所以d2_gid必须声明为u16_t类型。

proto.h(19700行)提供了K&R C编译器或ANSI标准C编译器所支持的函数原形。这是一个很长的文件，但并不很重要。只是有一点需要注意：因为文件系统处理多种不同的系统调用，并且由于文件系统的组织方式，不同的do_xxx函数分散在许多文件之中。proto.h按文件进行组织，如果要查阅处理某个特定系统调用的代码，可以在proto.h找到包含这个系统调用的文件。

最后，glo.h(19900行)定义了全局变量。接收和返回消息的缓冲区也定义在这个文件中。这里我们还用到了EXTERN宏，因而这些变量可以为文件系统的各个部分所存取。同MINIX的其他部分一样，在编译文件table.c时，系统会保留一定的存储空间。

进程表的文件系统部分包含在文件fproc.h中(20000行)。fproc数组也用EXTERN宏声明，它含有模式屏蔽码、指向根目录i-节点和工作目录i-节点的指针、文件描述符数组、uid和gid和各个进程的终端号，进程的id以及进程组的id也包含在这里。这些项中有一部分在内核和内存管理器的进程表中重复定义。

在系统调用被中途挂起，例如当从空管道中读取数据时，需要保存调用参数，为此专门使用了几个域。fp_suspended域和fp_revived域实际上只需要一位，然而使用字符时编译器可以生成更好的代码。此外，有一个域为POSIX标准所要求的FD_CLOEXEC位而设置，用于表明在执行EXEC系统调用时应该关闭文件。

现在我们来介绍文件系统中定义其他表的文件。文件buf.h(20100行)定义了块高速缓存，这个结构也用EXTERN声明。buf数组中含有所有的缓冲区，每个缓冲区分为数据部分和头部分。头部分中含有指针、标志和计数器等。数据部分说明为五种类型的联合(20117行)，这样我们可以把块视作为字符数组，或目录等来引用。

如果把缓冲区3作为字符数组，引用其数据部分的正确方法是buf[3].b.b__data。这里，buf[3].b表示整

个联合，我们使用的是其b__data域。尽管正确，但这种方法略显繁琐。因此在20142行定义了宏b_data，这样上述引用可以用buf[3].b_data来表示。请注意，b__data(联合中的域)中含有两个下划线。而为了区别，在b_data(宏)中只使用了一个下划线。访问块的其他宏定义列在20143行至20148行。

缓冲区哈希表buf_hash定义在20150行。哈希表中的每一项指向一个缓冲区列表，初始时，所有列表均为空。文件buf.h末尾的宏(20160行至20166行)定义了不同的块类型。在块使用完毕并返回到高速缓存中时，高速缓存管理进程可以根据这些值决定是否把块放在LRU链的前端或尾部、以及是否将该块立即写回磁盘。WRITE_IMMED位表明块被修改后，应该马上写回磁盘。超级块是文件系统中唯一无条件设置WRITE_IMMED的数据结构。那么MAYBE_WRITE_IMMED是什么意思呢？MAYBE_WRITE_IMMED定义在文件include/minix/config.h中，当ROBUST为真时，它等于WRITE_IMMED，否则等于0，在MINIX的标准配置中，ROBUST定义为0。

文件buf.h的最后一行(20168行)利用文件include/minix/config.h中NR_BUF_HASH的值定义了HASH_MASK。HASH_MASK和块号相与，根据结果，在buf_hash相应项所指向的链表中开始查找该块的缓冲区。

下一个文件dev.h(20200行)定义了dmap表，表本身在table.c中说明并置初值。dmap表的定义不能同时包含在几个文件中，这就是为什么使用dev.h文件的原因。这里，dmap用extern而不是EXTERN声明。dmap表提供了主设备号与相应任务之间的映射。

文件file.h中包含了中间表filp(20300行)(用EXTERN声明)。filp表用于存放文件当前位置及i-节点指针(参见图5-33)，此外，它还给出了打开的文件是否可以读写，以及有多少个文件描述符指向该项等。

文件锁表file_lock(用EXTERN声明)放在文件lock.h(20400行)中。这个数组的长度由NR_LOCKS指定，NR_LOCKS在文件const.h中定义为8。如果想要在MINIX系统上实现一个多用户的数据库，这个数字还要增加。

在文件inode.h(20500行)中定义了i-节点表inode(声明为EXTERN)，用于保存当前使用的i-节点。我们前面说过，在打开文件时，文件的i-节点被读入，并保存在内存中直至文件关闭。inode结构定义了保存在内存中，还未写入到磁盘i-节点上的信息。请注意，这里的i-节点只有一个版本，在将i-节点从磁盘上读入时，即处理了V1版本和V2版本的文件系统的差别。文件系统其他部分不必了解磁盘上文件系统的格式，至少在修改信息被写回磁盘之前是这样。

至此，多数域都是自解释的，然而我们有必要说明一下i-seek。前面提到过，为了优化，在文件系统顺序读取文件时，它会在请求前即将某些块读入到高速缓存中。而随机存取文件没有预读。在执行LSEEK系统调用时，i-seek域设置为禁止预读。

文件param.h(20600行)与内存管理器中的同名文件相似。它为包含参数的消息域定义了名称，因而代码中可以引用buffer来代替m.ml_pl，从消息缓冲区m中选择一个域。

在文件super.h(20700行)中，我们定义了超级块表。系统启动时，根设备的超级块被载入。而安装一个文件系统时，其超级块同样被载入。同其他表一样，super_block表也用EXTERN加以声明。

文件系统存储区分配

本节我们要讨论的最后一个文件并不是头文件，但是同内存管理器一样，在分析了所有的头文件之后，我们有必要介绍文件table.c，它在编译时包含了上述头文件。同文件系统的全局变量以及进程表的文件系统部分一样，我们前面提到的大多数数据结构，如块高速缓存、filp表等等，都用EXTERN宏声明。在编译table.c时即保留一定的存储区，这一点也与MINIX系统的其他部分相似。table.c还包含两个重要的初始化数组。call_vector是指针数组，可用于在主循环中决定过程与系统调用号的对应关系，在内存管理器中，我们也看到过类似的表。

在20914行有一张新表，即dmap表。从0开始，每个主设备对应一行。当设备被打开、关闭和读写时，dmap表提供了执行相应操作所调用的过程名。所有这些过程都放在文件系统的地址空间中，大多数过程什么也不做，也有些过程调用某个任务，请求I/O。另外，dmap表还给出了每个主设备的相应任务号。

要在MINIX中增加一个新的主设备，必须在dmap表中增加一行，说明在设备被打开、关闭和读写时，执行何种操作(如果有的话)。例如，如果在MINIX中增加一个磁带驱动器，当打开磁带驱动器的设备文件时，可以调用表中的过程，检查是否该驱动器还在使用中。为方便用户重新配置时修改该表，我们定义了宏DT进行自动处理(20900行)。

每个可能的主设备在dmap表中均占有一行，用宏的形式写出。有些必需的设备，宏中的enable参数置为1。如果驱动器还未准备或者已经移走，enable参数则置为0。对于可在文件include/minix/config.h中配置的设备，其对应项使用它的允许宏，例如20920行的宏ENABLE_WIN。

5.7.2 表的管理

与块、i-节点、超级块等表结构相关的是包含了管理这些表的过程的文件。这些过程多次为文件系统的其他部分调用，构成了表和文件系统之间的主要界面。因此，我们应该先从表的管理开始来分析文件系统的代码。

块管理

块高速缓存是通过文件cache.h中的过程进行管理的。这个文件包含有9个过程，如图5-34所示。第一个过程get_block(21027行)是文件系统获取数据块的标准方式。当文件系统过程需要读取用户数据块、目录块、超级块或其他任何块时，它调用get_block。在调用时，需要标明设备号和块号。

图5-34 用于块管理的过程。

get_block被调用时，它首先检查块高速缓存，看看请求的块是否在块高速缓存中。若在，返回其指针。

否则必须把它读入块高速缓存。块高速缓存中的块用NR_BUF_HASH个链接表链接在一起。NR_BUF_HASH和NR_BUFS都是可调的参数，后者指定了块高速缓存的大小。这两个参数都放在文件include/minix/config.h中。在本节后面部分我们会提到如何优化块高速缓存和哈希表的大小。HASH_MASK等于NR_BUF_HUSH-1,如果有256个哈希表，则HASH_MASK为255。因此每条链所有块的最后8位都是相同的字符串，分别为00000000、00000001、.....或11111111。

一般时候，get_block都首先要在哈希链中查找块，但有一种情况比较特殊：在读取稀疏文件的一个孔时，不必进行查找。这就是我们在21055行进行检查的原因。下面的两行将bp设置为指向某条链的开始，如果所查找的块在高速缓存中，它必然在这条链中，链的下标是HASH_MASK与块号相与的结果。接下来一行循环查找这条链，检查是否能找到所请求的块。若找到，并且该块未被使用，则从LRU链中删除之，若还在使用，则它根本不会在LRU链中。在21063行，将找到的块的指针返回给调用进程。

如果该块不在哈希链中，它也就不在高速缓存中。因此这时从LRU链中取出最久未使用块，将其缓冲区从原哈希链中删除，因为它将获得一个新的块号，从而属于不同的哈希链。在从原哈希链中删除时，若该块修改过，则在21095行把它写回磁盘。用flushall调用可以把同一设备上所有其他修改过的块都写到磁盘中。当前正在使用的块并不换出内存，事实上他们不在LRU链中。可是我们很难判断一个块是否还将使用，通常情况下在使用完成后，每个块立即用put_block释放。

一旦分配了缓冲区，所有域，包括b-dev，都使用新参数进行修改(21099行至21104行)，然后把该块从磁盘上读入。可是有两种情况我们并不需要读入磁盘块。当get_block以参数only_search调用时，表明这是预读。在预读时，查找一个可用的缓冲区，必要时需将原有内容写回磁盘，为找到的缓冲区赋予一个新的块号，但是其b_dev域置为NO_DEV，表明该块中还没有有效的数据。在讨论rw_scattered函数时，我们会看到它的使用。only_search也可以用来表征需要重写整个块，这时没有必要先把该块读入。在这两种情况下，缓冲区中的参数都被修改，但是忽略读磁盘过程(21107行至21111行)。在读入新块后，get_block将其指针返回给调用进程。

假设文件系统在查找文件名时临时需要一个目录块，它调用get_block请求该目录块，在找到文件名后，调用put_block把该块返回到高速缓存中(21119行)，其缓冲区可以用于满足随后的一个不同块的需要。

put_block把新返回的块插入到LRU链中，在某些情况下，需要把数据写回磁盘。第21144行，根据block_type参数判断把该块放在LRU链的前面还是后面。block_type是调用进程提供的一个标志，它给出了块类型。最近不久可能要用到的块放在LRU链的尾部，以便它在LRU链中可以保存一段时间，而最近很可能不会用到的块放在LRU链的前端，这样它的缓冲区可以重新分配。目前只有超级块按照后面一种方式处理。

在块被放到LRU链后，还要检查该块是否应立即写回磁盘(21172行和21173行)。在标准配置下，只有超级块标记为立即写回，而超级块只有在系统初始化，改变RAM盘大小时才被修改，需要立即写回。这种情况下，修改后的超级块写到RAM盘中，而它不大可能需要再次读入，因此立即写回很少使用。但是，用户可以修改include/minix/config.h文件中的ROBUST宏，使得i-节点、目录块以及关系到文件系统本身正确运行的其他块标记为立即写回。

随着文件的扩大，需要不时地给它分配区段以保存新数据。过程alloc_zone(21180行)可用于分配新区段。它在区段位图中查找空闲区段。如果要分配的区段是文件的第一个区段，那么可以使用超级块中指向设备上第一个可用区段的s_zsearch域，没有必要查找位图。否则，为了使文件的各区段放在一起，需要在区段位图中查找与文件最后一个区段相邻的区段，这可以通过从文件最后一个区段开始搜索位图实现(21203行)。位图中的位号与区段号之间的映射在21215行处理，第1位对应于第一个数据区段。

删除文件时，需要把它的区段返回到空闲区段位图中。free_zone(21222行)用于回收不再使用的区段，它以区段位图和位号为参数，调用free_bit。free_bit也用于回收空闲i-节点，但是后者以i-节点位图为第一个参数。

管理高速缓存需要读写块，为了提供一个简单的磁盘界面，MINIX中定义了rw_block过程(21243行)。rw_block读或写一个块。类似地，rw_inode用于读写i-节点。

本文件中下一个过程是invalidate(21280行)。比如在卸下磁盘时，调用此过程以便从高速缓存中删除属于某个刚卸下的文件系统的所有块。如果未调用该过程，则重新使用该设备(用另外一张软盘)时，文件系统可能把原来的块当成新块使用。

系统调用SYNC调用flush_all(21295行)刷新属于指定设备并修改过的缓冲区，将它们全部写回设备。每次卸下一个设备时，都需调用flash_all。执行时，它将块高速缓存看成线性数组，这样，所有修改过的缓冲区都可以找到，即使是那些当前正在使用，不在LRU链中的缓冲区也可以找到。在高速缓存中的所有缓冲区都扫描一遍，那些属于待刷新的设备以及需要重新写回的缓冲区则被加到dirty指针数组中。这个数组用static声明，所以不会保存在栈中，之后，该数组被传给rw_scattered函数。

rw_scattered函数(21313行)以设备标志符、缓冲区指针数组的指针、数组大小和一个读写标志位为参数。它首先对缓冲区指针数组按块号排序以提高读写操作的效率。只有上面提到的flushall函数以WRITING标志调用rw_scattered。这种情况下，块号的来源很容易理解，他们实际上是一些修改过的数据缓冲区。唯一调用rw_scattered进行读操作的是文件read.c中的rahead函数。现在，我们只要知道：在调用rw_scattered之前，已经以预取方式多次调用过get_block，因此预留了一组缓冲区；这些缓冲区中含有块号，但没有设备参数，不

过这不要紧，因为在调用rw_scattered时会提供该参数。

设备驱动程序对rw_scattered读请求的处理方式与写请求相比有很大的不同。写多个块的请求必定全部按请求处理，然而对读取多个块的请求，不同的驱动程序可能进行不同的处理，这要取决于如何才能最有效。rahead在调用过程rw_scatterred时往往伴随着对一簇块的请求，而这些块可能实际上并不需要，所以最好的响应是尽可能多地获取那些很容易就能读取的块，而不是在设备上四处寻道来读取所有块，因为这可能花费大量的寻道时间。例如，软盘驱动程序在磁道边界处停止，而许多其他驱动程序只读取连续的块。当读操作完成后，rw_scattered过程标志读入的块，在相应缓冲区中填入设备号。

图5-34中最后一个函数是rm_lru(21387行)。这个函数从LRU链中删除一个块，它只由本文件中的get_block调用，因而声明为PRIVATE，使得它对本文件外的过程不可见。

在结束块高速缓存的分析之前，我们谈谈如何优化之。NR_BUF_HASH必须是2的幂。如果它大于NR_BUFS，那么哈希链的平均长度将不足1。若内存可以保存大量的缓冲区，则必然有足够的空间来容纳大量的哈希链，因此通常把NR_BUF_HASH选为大于NR_BUFS的最小的2的幂。本文定义了512个块和1024个哈希表。最佳大小取决于系统的使用情况。根据经验，我们发现将缓冲区数目增加到1024以上，在重新编译MINIX时并不能改进系统性能，因此，1024个缓冲区显然足以容纳在整个编译过程中用到的二进制文件。对于其他一些作业，较小的缓冲区可能就足够了，但是也有可能使用较大的缓冲区能够提高系统性能。

本书所附CD_ROM中，MINIX系统的二进制文件是使用一个小得多的块高速缓存编译而成的，这样它可以在更多的主机上运行。我们希望生成一个可以在只有2MB内存的系统上安装的MINIX发行版本，而用1024个块高速缓存编译而成的系统需要多于2MB的RAM内存。这个二进制文件中还包含有所有可能的硬盘驱动程序以及其他在特殊安装时不会用到的驱动程序。大多数用户宁愿在安装后编辑include/minix/config.h文件，重新编译该系统，从中删除不必要的驱动程序，并尽可能增加块高速缓存的数量。

谈到块高速缓存，我们需要指出：因为在16位的Intel处理器上内存段最大长度为64KB，这使得在这些机器上不可能使用很大的高速缓存。我们可以配置文件系统使它将RAM盘作为辅助高速缓存，存放所有调出主高速缓存的块。这种辅助高速缓存在32位的Intel系统中并不需要，因此我们不加讨论；若允许的话，一个大的主高速缓存将带来很高的性能，然而对于那些文件系统的虚拟地址空间不足以存放很多主高速缓存的机器(例如286)来说，辅助高速缓存非常有效。辅助高速缓存要比常规RAM盘的性能优越。一个高速缓存，如果它所包含的数据至少被用到一次，并且这个高速缓存非常大，那么它将极大地提高系统的性能。在这里，什么才是"足够大"并不能在事先确定，我们只能试着增大高速缓存的数量，观察系统性能是否确实得到了改进，才能确定高速缓存是否足够大了。time命令测量运行一个程序所耗费的时间，可以帮助我们对系统进行优化。

i-节点管理

块高速缓存并不是唯一需要支持过程的表，i-节点表同样需要。很多管理i-节点表的过程与块管理过程相类似，他们都列在图5-35中。

图5-35 用于i-节点管理的过程。

get_inode过程(21534行)与get_block相似。当文件系统任何部分需要一个i-节点时，它调用get_inode过程。get_inode首先搜索inode表，判断所请求的i-节点是否在内存中。若在，则将其使用计数器加1，并且返回其指针。这一搜索过程包含在21546行至21556行的代码之中。如果i-节点不在内存中，则调用rw_inode将它读入。

当一个过程使用完i-节点之后，它调用put_inode过程(21578行)把i-节点返回。put_inode将i-节点的使用计数器i-count减1。如果i-count减到0，表明文件不再使用，此时i-节点可以从inode表中删除，若它被修改过，则需要写回磁盘。

如果i-节点的i_link域为0，表明没有任何目录项指向相应文件，因此它的所有区段均可被释放。注意使用计数器为0和链接数为0是完全不同的两码事，他们的原因不同，所导致的结果也不同。如果i-节点用于管道，这时尽管其链接数目非0，我们也必须释放管道的所有区段。这常常发生在读进程释放它所读取的管道时，只为一个进程保留管道显然是件荒谬的事。

在创建一个新文件时，必须调用alloc_inode(21605行)为其分配i-节点。MINIX允许设备按只读模式安装，因此要检查设备的超级块确保其可写。对i-节点的处理和对区段的处理不同，系统尽量使一个文件的各个区段相邻，而i-节点则无此必要。为了节省查找i-节点位图的时间，我们可以利用超级块中记录第一个空闲i-节点的域。

在获得某个i-节点后，alloc_inode调用get_inode过程将它读入内存的inode表中。接着初始化这个i-节点的各个域。其中一部分初始化工作在21641行至21648行完成，而另一部分则调用wipe_inode过程(21664行)完成。我们之所以这样分工，是因为在文件系统的其他地方还要使用wipe_inode来清除i-节点中的某些域(不是所有域)。

删除文件时，调用free_inode过程(21684行)释放它的i-节点。这个过程只是将i-节点位图中的相应位置0，并且修改超级块中记录第一个空闲i-节点的域。

下面一个函数update_times(21704行)从系统时钟获取时间，并且修改时间域。update_times函数还在STAT和FSTAT两个系统调用中使用，因而用PUBLIC声明。

rw_inode过程(21731行)与rw_block相似，它的任务是从磁盘上取得一个i-节点。其执行过程如下：

1. 计算包含所需i-节点的块。
2. 调用get_block读入该块。
3. 从该块中提取i-节点并把它拷贝到inode表中。
4. 调用put_block返回该块。

rw_inode过程实际上比我们上面列出的更复杂，因此需要一些附加函数。首先，获取当前时间代价很大，因此在需要修改i-节点的时间域时，若i-节点在内存中，则只设置它的i-update域作为标志。在i-节点要写回磁盘时如果发现i-update域非0，这时才调用update_times过程。

其次，MINIX的开发历史也增加了复杂性：原先V1版本文件系统在磁盘上的i-节点具有与V2版本不同的结构。函数old_icopy(21774行)和new_icopy(21821行)实现结构的转换。前者在内存中的i-节点和V1文件系统使用的i-节点之间进行转换，后者在内存中的i-节点和V2文件系统使用的i-节点之间进行转换。这两个函数仅在本文件中调用，所以声明为PRIVATE。每个函数均处理两个方向的转换(从磁盘到内存和从内存到磁盘)。MINIX也在与Intel处理器字节顺序不同的一些系统上实现，每种实现都在磁盘上使用本地字节顺序。超级块中的sp->native域指出了使用的字节顺序。必要时，old_icopy和new_icopy都要调用conv2和conv4来交换字节顺序。

dup_inode(21865行)仅仅增加i-节点的使用计数器，在打开一个已经打开的文件时被调用。第二次打开文件时，并不需要把文件的i-节点从磁盘上读入。

超级块的管理

文件super.c中包含了管理超级块和位图的过程。这个文件中有5个过程，列于图5-36中。

图5-36 用于管理超级块和位图的过程。

当需要一个i-节点或区段时，调用alloc_inode或alloc_zone。如我们在前面所述，这两个过程都调用alloc_bit(21926行)搜索相关位图。这个搜索过程有三个嵌套循环：

1. 最外层，对位图的所有块的循环。
2. 中间层，对块的所有字的循环。
3. 最内层，对字的每一个位的循环。

中间层循环检查当前字各位是否全1。如全1，则这个字中没有空闲的i-节点或区段，这时接着检查下一个字，若找到一个不是全1的字，则进入内层循环，查找空闲位(即0)。如果搜索完所有块后都没有找到0位，表明没有空闲i-节点或区段，因此返回NO_BIT码(0)。这样的搜索过程需要很多处理器时间，但是使用超级块中指向第一个空闲i-节点或第一个空闲区段的域，把它作为origin参数传给alloc_bit过程，可以大大缩短搜索时间。

释放位要比分配位简单得多，它不需要搜索位图。free_bit过程(22003行)首先计算哪一个位图块包含了要释放的位，接着调用get_block将这个位图块读入内存，相应位置0，最后调用put_block把该块写回磁盘。

下面一个过程get_super(22047行)用于在超级块表中搜索特定设备。例如，在安装一个文件系统时，需要检查它是否已经安装，这时可以调用get_super查找该文件系统的设备，如果没有找到，则说明相应的文件系统还未安装。

下面的函数mounted(22067行)仅在关闭块设备时调用。正常情况下，在设备关闭时，它的所有在高速缓存中的数据均被丢弃。可是如果这个设备恰巧是被安装设备，丢弃在高速缓存中的数据是不可取的。mounted函数调用时，以一个指向设备i-节点的指针为参数，若该设备是根设备或是被安装的设备，该函数返回真。

最后，我们看看read_super过程(22088行)。它与rw_block和rw_inode有些类似，但只在读超级块时调用。系统的正常操作中并不需要写超级块。read_super检查所读出的文件系统的版本号，必要时进行相应的转换。这样尽管在磁盘上超级块结构不同，但是内存中超级块的拷贝都是标准结构。

文件描述符的管理

MINIX中还有管理文件描述符和filp表的过程(见图5-33)，这些过程放在文件filedes.c中。创建或打开文件时，需要分配一个空闲的文件描述符和一个空闲的filp项，这时可调用get_fd过程(22216行)。但是分配到的文件描述符和filp项并不标志为使用，因为在确信CREAT和OPEN成功之前，需要进行许多检查。

get_filp过程(22263行)用于检查文件描述符是否在合适的范围之内，若在，返回它的filp指针。

本文件中的最后一个过程是find_filp(22277行)。当进程正向一个断裂管道(即这个管道未被任何进程打开读取)写数据时，调用该过程。find_filp搜索filp表，查找潜在的读进程。若未找到，说明这个管道是断裂管道，写入失败。

文件锁

POSIX记录锁函数如图5-37所示。在FCNTL系统调用中指定F_SETLK或F_SETLKW请求，可以在文件某部分加上读写锁，或者只加写锁。利用F_GETLK请求，我们很容易判断文件的某一部分是否有锁。

图5-37 POSIX的建议性记录锁操作，这些操作通过系统调用FCNTL请求。

文件lock.c中只有两个函数。lock_op(22319行)为FCNTL所调用，调用时给出图5-37中的一个操作码。lock_op检查指定的锁范围是否有效，在设置锁时，不能与现有的锁冲突，在清除锁时，现有的锁不能分成两个部分。清除任何一个锁后，调用本文件中的另一个函数lock_revive(22463行)，这个函数唤醒所有因等待锁而阻塞的进程。我们之所以采取这一折衷方法，是因为要指出哪些进程正在等待释放某个特定的锁，需要编写额

外的代码。那些还在等待加锁文件的进程被唤醒后，会再次阻塞。这种策略假定锁并不经常使用。如果要在MINIX系统上建立一个大型的多用户数据库，那么就需要重新实现文件锁这一部分。

加锁文件被关闭时，也要调用lock_revive过程。这种情况是可能发生的，例如，进程在结束使用加锁文件之前被终止。

5.7.3 主程序

文件系统的主循环包含在文件main.c中，其代码从22537行开始。在结构上，文件系统的主循环和内存管理器以及I/O任务的主循环非常相似。它调用get_work等待下一条消息请求(除非先前在管道或终端上被挂起的进程如今能够执行)。它还设置全局变量who为调用者的进程表槽口号，同时设置另一个全局变量fs_call为待执行的系统调用的编号。

一旦控制回到主循环，设置三个标志：fp指针指向调用进程的进程表项(fproc)，super_user给出了调用进程是否是超级块用户，dont_reply初始时设置为假。然后是关键部分，即执行系统调用的过程。用fs_call作为过程指针数组call_vect的下标选定被调用的过程。

当控制重新回到主循环之后，如果dont-reply被设置，则禁止发送回答信号(例如，进程因试图从空管道中读取数据而阻塞)。否则送回一个回答消息。主循环中最后的一条语句检测文件是否被顺序读取，并且在实际需要前把下一块载入高速缓存中，从而改进系统性能。

get_work过程(22572行)检查是否有以前阻塞的进程已醒，这些进程的优先级高于新收到的消息。只有当文件系统没有内部操作时，才调用系统内核获得消息(22598行)。

在系统调用完成之后，不管成功与否，都要用reply函数(22608行)向调用进程发送回答消息。调用进程有可能已被信号终止，因此从内核中返回的状态码被忽略，在这种情况下，不进行任何处理。

初始化函数

文件main.c的剩下部分由一些仅用于系统初启的函数组成。在文件系统进入主循环之前，它调用fs_init(22625行)进行初始化。而fs_init又调用几个其他函数，初始化块高速缓存、获得启动参数、必要时装载RAM盘以及载入根设备的超级块等等。接下来，fs_init为所有任务和服务器初始化进程表的文件系统部分(22643行至22654行)。最后，fs_init对一些重要常数进行检测，以确保其合理性。为了供ps程序使用，还要向内存任务发送一条消息，其中包含进程表的文件系统部分的地址。

fs_init调用的第一个函数是buf_pool(22679行)，它建立块高速缓存使用的链接表。图5-31显示了块高速缓存的正常状态，其中，所有块都同时链接到LRU链和一个哈希链中。为了更好地理解，我们来看看图5-31的情况是如何发生的。在高速缓存用buf_pool过程初始化后，所有的缓冲区均在LRU链中，并且这些缓冲区都用0号哈希链链接起来，如图5-38(a)所示。当一个缓冲区被请求并在使用时，情况如5-38(b)，这时，我们看到一个块从LRU链中删除，放在另一个不同的哈希链中。正常情况下，块立即被释放，返回到LRU链中。图5-38(c)中显示了该块返回到LRU链后的情况。尽管此时没有被使用，但是如果需要它还可以被存取，并提供相同的数据内容，所以把它保留在哈希链中。在系统运行了一段时间后，差不多所有块都被使用过，随机地分布在不同的哈希链中。这时，LRU链看上去就象图5-31一样。

图5-38 块高速缓存的初始化。 (a)在使用任何缓冲区之前。 (b)在请求一块后。
(c)在该块被释放后。

接下来一个函数是get_boot_parameters(22706行)。它送一条消息到系统任务，向它请求启动参数的一份拷贝，这些参数用在下面的load_ram函数(22722行)中，load_ram为RAM盘分配空间。如果启动参数标明

rootdev=ram

从引导块开始，根设备文件系统被逐块地从设备ramimagedev拷贝到RAM盘中，拷贝时对文件系统的数据结构不作任何解释。若ramsize启动参数较根设备文件系统小，则扩大RAM盘以容纳之。如果ramsize大于根设备文件系统的长度，则分配指定大小的空间，并调整RAM盘文件系统，让它使用所有这些空间(22819行至22825行)。22825行的put_block调用是文件系统中唯一对超级块进行写操作的调用。

若指定的ramsize非零，load_ram为空RAM盘分配空间。这时，因为没有拷贝文件系统结构，该RAM设备不能用作文件系统，除非它已使用mkfs命令进行初始化。除此之外，如果编译的文件系统提供相应支持，这样的RAM盘还可以用作辅助高速缓存。

文件main.c中的最后一个函数是load_super(22832行)。它初始化超级块表，并把根设备的超级块读入内存。

5.7.4 对单个文件的操作

本节我们逐个研究对文件的系统调用(同目录相区别)。我们从打开、创建和关闭文件开始，然后详细讨论文件的读写机制，在这之后，我们再来分析管道及其操作与文件有什么不同。

创建、打开和关闭文件

文件open.c中包含有6个系统调用的代码，这6个系统调用是：CREAT、OPEN、MKNOD、MKDIR、CLOSE和LSEEK。我们首先将CREAT和OPEN结合起来考虑，然后逐个讨论其他的系统调用。

在老版本的UNIX中，CREAT和OPEN调用用于不同的目的。打开一个并不存在的文件会出错，而一个新文件必须用CREAT创建，CREAT也可以把一个存在文件的长度删除为0。可是POSIX系统不再需要两种不同的调用。在

POSIX中，OPEN调用可以用于创建一个新文件和截断一个老文件，因此CREAT调用实现的功能只是OPEN的一个子集，仅用于与原有程序相兼容。处理CREAT和OPEN的过程分别是do_creat(22937行)和do_open(22951行)(文件系统使用了和内存管理器相同的惯例，即系统调用xxx由过程do_xxx执行)。打开或创建一个文件包括三步：

1. 找到i-节点(创建文件时需要进行分配和初始化)。
2. 找到或创建目录项。
3. 为文件建立并返回一个描述符。

CREAT和OPEN都要做两件事：获取文件名、并调用common_open执行这两个调用的共同操作。

common_open过程(22975行)首先验证空闲的文件描述符及空闲的filp项是否存在。若调用函数指定创建新文件(调用时设置O_CREAT位)，则执行22998行的new_node过程。如果目录项存在，new_node将返回指向现有i-节点的指针，否则创建目录项和i-节点。在无法创建i-节点时，设置全局变量err_code。错误码并不总是表示错误，如果new_node发现一个已有的文件，返回的错误码表明文件存在，但是这种错误是可以接受的(23001行)。若O_CREAT位没有设置，需用另一种方法来搜索i-节点，即我们后面要讨论的文件path.c中的eat_path函数。这时要注意的是：如果没有找到i-节点或者创建i-节点失败，在执行到23010行前common_open出错并终止。否则继续执行，对文件描述符赋值，并在filp表中申请一项。之后，如果新文件刚被创建，就跳过23017行到23094行之间的代码。

对于现有文件，文件系统需要检查文件的类型、模式等，从而确定该文件是否能打开。第23018行的forbidden调用首先对rwx位进行一般性检查。若是正规文件，并且调用common_open时设置了O_TRUNC位，则文件长度截为0，再次调用forbidden(23024行)以确保文件可写。如果权限允许，调用wipe_inode和rw_inode以重新初始化i-节点，并将它写回磁盘。其他文件类型(目录、设备文件、命名管道)也需进行适当的检测。对于设备，在23053行(使用dmap结构)调用适当的过程打开它。若是命名管道，则调用pipe_open(23060行)，并进行和管道有关的各种检查。

common_open的代码，同文件系统的其他许多过程一样，含有大量检查各种错误和非法组合的代码，这些代码对于设计一个无错的、健壮的文件系统是必不可少的。如果出现错误，以前分配的文件描述符和filp项被收回，i-节点被释放(23089行至23101行)。在这种情况下，common_open过程返回一个负数以表示出错。若运行成功，返回一个正数，即文件描述符。

现在我们可以更详细地讨论过程new_node(23111行)了。new_node负责分配i-节点，并为CREAT和OPEN把路径名加到文件系统中。它还用在MKNOD及MKDIR两个调用中，这两个调用我们以后会介绍。第23128行的语句分析路径名(即逐个部分查询路径)，直到最后一个目录；后面第三行的advance调用检查最后一个部分能否被打开。

例如，在调用

```
fd=creat("/usr/ast/foobar", 0755);
```

时，last_dir把/usr/ast的i-节点装入内存i-节点表中，返回其指针。如果文件不存在，不久我们就要用到这个i-节点，以便把foobar加入目录中。所有其他增加或删除文件的系统调用，也都先用last_dir打开路径的最后一个目录。

假设new_node发现文件不存在，它就调用23134行的alloc_inode过程分配并载入一个新的i-节点，返回指向该节点的指针。若磁盘上没有空闲i-节点，new_node执行失败，返回NIL_INODE。

如果能够分配i-节点，则继续执行23144行的语句，填入某些域，把它写回磁盘，然后在最后目录下加入文件名(23149行)。这里我们再次看到文件系统需要经常地检查错误，一旦遇到错误，精心地释放它申请到的所有资源，比如i-节点和块。在出现如i-节点耗尽等情况时，如果不取消当前调用的影响，给调用进程返回错误码，而是让MINIX陷于混乱，那么我们的文件系统要简单得多。

我们前面也说到过，对于管道需要作特殊处理。如果一个管道没有读/写对，pipe_open（23176行）过程将挂起调用进程，否则它调用release函数在进程表中查找因该管道而阻塞的进程，并将找到的进程唤醒。

MKNOD调用在过程do_mknod(23205行)中处理，这一过程与do_creat类似，只是前者仅仅创建i-节点并为其分配一个目录项。事实上，大多数工作都是由23217行的new_node过程完成的。如果i-节点已存在，则返回一个错误码。这个错误码与common_open中调用new_node得到的错误码相同。在后面一种情况下，这个错误码可以接受，但是对于前者，错误码被传递给调用进程，以便作相应处理。在此，我们不再逐行分析了。

MKDIR调用在do_mkdir函数(23226行)中处理。同我们这儿讨论的其他系统调用一样，new_node仍然起了重要作用。与文件不同的是，目录通常有链接。此外，目录不可能为空，在创建时，它至少包含两个目录项"."和"..",分别指向当前目录及其父目录。我们将文件链接数限制为不超过LINK_MAX(对于标准的MINIX文件系统，LINK_MAX在文件include/limits.h中定义为127)。因为子目录对父目录的引用也是到父目录的一个链接，do_mkdir首先检查是否可以在父目录中创建另外一次链接(23240行)。若可以，则调用new_node过程。new_node成功执行后，接着创建"."和".."两个目录项(23261行和23262行)。这一过程非常直观，但可能出错(例如磁盘已满)，因此，我们提供很多代码，以便在操作无法完成时，恢复到进程的初始状态。

关闭文件比打开文件要容易得多，它主要是在do_close过程(23286行)中完成的。管道和设备文件需要特别注意。但是对于正规文件，我们只需要减少filp计数器并检查该计数器是否为0。若为0，则调用put_inode过程回收i-节点。最后删除所有的锁，唤醒所有因等待这个文件的锁被释放而挂起的进程。

请注意，回收一个i-节点意味着它在inode表中的计数器减1，这样它将最终从inode表中删除。这个操作与i-节点的释放(即在位图中设置一位表明这个i-节点可用)无关。只有在文件从所有目录中删除时，才要释放i-节点。

文件open.c的最后一个过程是do_lseek(23367行)。进行文件读/写定位时，调用这个过程设置新的文件位置。第23394行禁止预读；顺序存取文件时，不能设置文件位置。

读文件

文件一旦被打开，就可以进行读写。许多函数既可用于读文件，又可以用于写文件。这些函数在文件read.c中。我们首先分析read.c，然后再来分析write.c中专门用于写操作的代码。读写操作有很大的不同，但是也有很多相似之处，因此，do_read过程(23434行)执行时，仅需在调用公共过程read_write时设置READING标志即可。在后面一节我们还会看到，do_write的代码也同样简单。

read_write过程从23443行开始。在23459行至23462行之间有一些特殊的代码，这些代码在文件系统把内存管理器载入用户空间时执行。正常调用的处理从第23464行开始，之后是有效性检查(例如，读取一个只为写而打开的文件)以及变量的初始化。从字符设备文件中读取数据并不经过块高速缓存，因此在第23498行将它们筛选出来。

从23507行到23518行的检测只适用于写操作，它处理文件长度可能超出设备容量的情况，或者超出文件尾写数据从而在文件中产生空洞的异常。我们在MINIX概述一节中说过，一个区段中使用多个块会导致一些问题，需要特别处理。在这里，我们还需要检查管道。

读文件的关键，至少对普通文件来说，是从23530行开始的循环。这个循环把读请求分成若干小块，每个小块都在一个单独的磁盘块中，它从当前位置开始，直到满足下面的条件之一：

1. 所有字节都读完。
2. 遇到一个块边界。
3. 读到文件末尾。

这些规则表明一个小块不可能用到两个磁盘块。图5-39中的三个例子说明如何确定小块的大小，图中小块分别为6、2和1个字节。实际计算由23632行到23641行的代码完成。

图5-39 三个例子表明在10字节文件中如何决定第一个小块的长度，块大小为8个字节，请求的字节数为6，小块用阴影显示。

事实上，读取小块是在rw_chunk中完成的。在rw_chunk执行后，各计数器及指针相应增加，接着进行下一个循环，当循环结束后，文件位置及一些其他变量(例如管道指针)也要作修改。

最后，如果要求预读，则将要读的i-节点和位置存储在全局变量中，以便在向用户送回响应消息后，文件系统能够接着读取下一个磁盘块。大多数情况下，文件系统被阻塞，等待下一个磁盘块，这时，用户程序可以处理刚刚收到的数据。这种安排使得计算和I/O并发执行，从而大大地提高了系统性能。

rw_chunk过程(23613行)以i-节点和文件位置为参数，把他们转换成物理磁盘块号，然后请求传送该块(或其中一部分)到用户空间中。相对文件位置与物理磁盘地址的映射是在read_map函数中进行的。对于普通文件，23637行和23638行的变量b和dev分别包含了物理块号及设备号。在23660行的get_block调用查找块，必要时将它读入内存。

一旦我们获得了该块的指针，就可以调用23670行的函数sys_copy将块中的所需部分传送到用户空间中，随后调用put_block过程释放该块，以便在需要时，将该块换出块高速缓存(在调用get_block申请到某一块后，该块不会出现在LRU队列中，如果块头中的计数器表明该块还在使用，它也不会返回到LRU链中，因而可避免被换出。put_block过程将计数器减1，在它减为0时，将这个块放到LRU队列中。)第23680行的代码表明是否一次写操作会填满块。然而，通过n传递给put_block过程的值并不会影响如何将块放入队列中。所有块一律放在LRU链的尾部。

read_map过程(23689行)检查i-节点，将逻辑文件位置转换为物理块号。对于靠近文件头，在前面7个区段(即那些在i-节点中的区段)中的块，通过简单的计算便可知道要访问哪个区段、哪个块。而对于文件中更后面的块，则需要读取一个甚至多个间接块。

读取间接块时，调用过程rd_indir(23753行)。我们之所以将它作为一个单独的过程，是因为数据在磁盘上的存放格式不同，这主要视文件系统的版本号及编写的硬件而定。如果必要的话，在rd_indir中进行繁琐的转换，使得文件系统的其他部分只见到唯一一种格式的数据。

read_ahead(23786行)将逻辑位置转换成为物理块号，调用get_block确保块在高速缓存中(或将其调入)，之后立即回收该块。read_ahead对块并不进行任何操作，只是增加它即将使用时，已调入内存的机会而已。

请注意，read_ahead仅在main函数中调用。它并不作为READ系统调用的一部分而调用。我们还要意识到，read_ahead 调用是在送回响应之后进行的，所以，用户可以接着进行其他操作，尽管文件系统不得不等待读取下一个磁盘块。

read_ahead本身只是请求另一块，它调用文件read.c中的最后一个函数rahead来完成该操作。

rahead(23805行)的工作原理基于这样的思想：如果稍多一点总比没有强，那么再多一点则更棒。因为磁盘和其他的存储设备在找第一块时要花较长一段时间，而读取相邻块所需时间则较短，稍加努力，我们就可以读取更

多的块。预读请求送给get_block，后者申请块高速缓存以便一次接收多个块。接着调用rw_scattered。我们在前面已经讨论过：当设备驱动程序被rw_scattered调用时，各个驱动程序可以自由地响应那些能够被高效地处理的请求。这听起来很复杂，但对那些从磁盘上读取大量数据的应用，这种复杂的办法可能显著地加快速度。

图5-40显示了在读取文件时用到的一些主要过程之间的关系，特别是他们之间的调用关系。

图5-40 用于读文件的一些过程。

写文件

写文件的代码放在文件write.c中。写文件与读文件类似，do_write过程(24025行)只是在调用read_write时设置WRITING标志。读写文件的主要不同在于写文件需要分配新的磁盘块。write_map(24036行)与read_map相似，只不过它不是在i-节点及其间接块中查找物理块号，而是在其中添加新的物理块号(精确地说，它添加的是区段号，而不是块号)。

write_map的代码很长，它需要处理几种不同情况。如果插入的区段靠近文件头，我们只需要将它插在i-节点中(24058行)。

最糟糕的情况是，文件超过了一次间接块所能处理的长度，这时需要使用二次间接块。接下来，分配一个一次间接块，并将其地址填入二次间接块中。同读文件一样，这时要调用一个独立的过程，wr_indir。如果成功获取二次间接块，而这时磁盘已满，无法分配一次间接块，就需要将前面获得的二次间接块返回，以免破坏位图。

这时候，若我们仅仅让系统陷于混乱并显示严重故障信息，那么MINIX代码就简单许多。然而，从用户观点来看，在磁盘空间不足时，WRITE调用返回错误码，比系统因此崩溃而造成文件系统被破坏要好得多。

wr_indir(24127行)调用conv2或conv4进行必要的数据转换，并将新区段号写入间接块中。注意这个函数的名字，它同其他涉及读写操作的函数的名字一样，字面含义并不一定很确切。真正往磁盘中写入数据是由管理块高速缓存的函数进行的。

文件write.c中的下一个过程是clear_zone(24149行)，它负责清空突然出现在文件中部的某些块。超出文件尾写入数据时，就会出现这种情况。幸好，它并不经常发生。

若需要一个新块，rw_chunk过程调用new_block(24190行)。图5-41是顺序文件扩大时的6个连续阶段。在这个例子中，块大小为1K，而区段大小为2K。

图5-41 (a)-(f)块(区段大小为2K，块大小为1K)的连续分配。

第一次调用new_block时，分配区段12(块24及25)，接下来使用块25，第三次调用时，分配区段20(块40及41)等等。zero_block过程(24243行)清空某个块，删除块中以前的内容，我们上面的描述要比zero_block的实际代码长得多。

管道

在许多方面，管道和普通文件类似。这一节，我们主要讨论他们之间的不同。我们要分析的代码都在文件pipe.c中。

首先，创建管道的方式不同，管道并不由CREAT，而是由PIPE来创建。do_pipe过程(24332行)处理PIPE调用，它为管道分配一个i-节点，并返回两个文件描述符。管道为系统，而不是用户所拥有，并放在指定的管道设备上(在文件include/minix/config.h中配置)。管道数据无需永久保存，所以可把RAM盘用作管道设备。

管道读写与文件的读写也略有不同，管道只有有限的容量。在管道已满时，所有继续向管道写入数据的进程都被挂起。同样，读取空管道的进程也会被挂起。事实上，管道有两个指针：当前位置指针(由读进程使用)及长度指针(由写进程使用)。这两个指针决定读写管道数据的位置。

pipe_check过程(24385行)进行各种检查，以保证对管道的操作能够完成，这些检查都将导致调用进程被挂起。除此之外，pipe_check还调用release过程，判断能否唤醒那些因为没有数据或有过多数据而被挂起的进程。对于正在睡眠的写进程和读进程，唤醒操作分别在24413行和24452行执行。在pipe_check中，还要检测往一个断裂的管道(没有读进程)中写数据的异常。

挂起一个进程是由suspend函数实现的(24463行)。该函数将调用参数保存在进程表中，并将dont_reply标志设为真，从而禁止文件系统的响应消息。

release过程(24490行)检查是否某个因管道而挂起的进程可以继续运行。如果找到这样的进程，它调用revive设置一个标志，以便于主循环处理。这个函数并不是系统调用，我们将它列在图5-27(c)中，是因为它使用了消息传递机制。

文件pipe.c中的最后一个过程是do_unpause(24560行)。在内存管理器试图向某进程发送信号时，它必须检查该进程是否正挂起在一个管道或者设备文件上(这种情况下，它必须用EINTR错误唤醒)。因为内存管理器并不能识别管道和设备文件，它向文件系统发送一条消息询问，这条消息在过程do_unpause中处理。若进程被阻塞，do_unpause将它唤醒。同revive一样，do_unpause有些类似于一个系统调用，尽管它并不是系统调用。

5.7.5 目录和路径

我们前面已经讨论了文件的读写机制，下一步就来看看路径名和目录是如何处理的。

将路径名转换成i-节点

许多系统调用(例如OPEN、UNLINK和MOUNT)都以路径名(即文件名)作为参数。这些系统调用在开始调用前，

大多数都要首先获得指定文件的i-节点。路径名如何转换成i-节点是本小节研究的主要内容。在图5-14中我们已经看过了大致的轮廓。

路径名分析在文件path.c中进行。在path.c中，第一个过程是eat_path(24727行)。该过程接受指向路径名的指针，进行分析，并把它的i-节点调入内存，然后返回指向这个i-节点的指针。具体过程如下：它首先调用last_dir获得最后目录的i-节点，随后调用advance取得路径的最后一个部分。如果搜索失败，比如路径中有一个目录不存在，或者虽然存在但却禁止搜索，那么就返回NIL_INODE。

路径名可以是绝对路径名或相对路径名，可以含有任意多个部分，各部分之间以斜杠分隔。这些问题都在last_dir函数(24754行)中处理。last_dir首先检查路径名的第一个字符以判断其为绝对路径还是相对路径(24771行)。对于绝对路径，rip设置为指向根i-节点的指针，否则rip设置为指向当前工作目录的指针。

现在，last_dir已经知道路径名以及指向某个目录的i-节点，可以在这个i-节点中查找路径的第一部分了。它进入24782行的循环，逐部分对路径名进行分析。当到达尾部时，便返回指向最后目录的指针。

get_name实用过程(24813行)从字符串中提取文件路径名的各个部分。更有趣的是advance过程(24855行)，它以目录指针和一个字符串为参数，在目录中查找该字符串。如果找到，则返回指向相应i-节点的指针。在advance中还要仔细处理如何跨越至安装的文件系统中。

尽管advance处理字符串查找。然而字符串与目录项的比较是在search_dir(24936行)中实现的。search_dir是文件系统中唯一检查目录文件的过程，它含有两个嵌套的循环：一个是针对目录中各块的循环，另一个是针对块中各项的循环。在目录中增加和删除名字时也要调用search_dir过程。图5-42给出了用于查找路径名的一些主要过程之间的关系。

图5-42 用于查路径名的一些过程。

安装文件系统

MOUNT和UMOUNT两个系统调用影响着整个文件系统。利用这两个调用，不同次设备上的文件系统可以链接在一起，形成一个无缝的命名树。正如我们在图5-32中看到的那样，安装是通过把被安装文件系统的根目录i-节点及其超级块读入内存，并且设置超级块中两个指针实现的。其中一个指针指向被安装至的i-节点，另一个指针指向被安装文件系统的根目录i-节点。这两个指针把不同的文件系统链接在一起。

上述两个指针是由文件mount.c中的do_mount函数在25231行和25232行设置的。在指针设置前的两页代码主要检查在安装文件系统时可能出现的各种错误。其中的一些错误如下：

1. 给定的设备文件不是块设备。
2. 设备文件是块设备，但是已经安装。
3. 被安装文件系统具有不正确的魔数。
4. 被安装文件系统无效(例如，没有i-节点)。
5. 要被安装至的文件不存在或者是设备文件。
6. 没有空间以存放被安装文件系统的位图。
7. 没有空间以存放被安装文件系统的超级块。
8. 没有空间以存放被安装文件系统的根目录i-节点。

或许我们不宜在此反复强调，然而在一个实用的操作系统中，确实需要有很大部分的代码用于错误处理。如果某个用户偶尔安装一个错误的软盘，例如每月一次，出现系统崩溃，文件被破坏，他肯定会怀疑系统的可靠性，因而责怪系统设计者，而不会从他自身找原因。

我们这里可以引用托马斯·爱迪生的一段论述。他认为"天才"是1%的灵感加99%的勤奋的结果。一个好的系统和一个平凡的系统的不同之处，并不是调度算法是否优越，而在于是否考虑了所有的细节问题。

卸下文件系统要比安装容易得多，出错的机会也要少。do_unmount过程(25241行)处理卸载。卸载时，关键在于确保没有进程在被卸下的文件系统中具有打开文件或工作目录。这个检查过程是很直观的：我们只需扫描整个i-节点表，检查内存中是否有i-节点属于要卸下的文件系统(根目录i-节点除外)。若有，UMOUNT调用失败。

文件mount.c中的最后一个过程是name_to_dev(25299行)。name_to_dev以一个设备文件的路径名作为参数，获得它的i-节点并从中提取主设备号和次设备号。这些设备号存放在i-节点中原用于存放第一个区段的地方。该位置之所以能够使用，是因为设备文件并没有区段。

链接和解链文件

我们接下来分析文件link.c。link.c用于链接文件和解链文件。同do_mount类似，do_link(25434行)中几乎所有代码均用于错误检查。在调用

link(file_name, link_name);

时，可能发生的错误列举如下：

1. file_name不存在或者不能访问。
2. file_name已经有最大数目的链接。
3. file_name是一个目录(只有超级块用户才可以对目录进行链接)。
4. link_name已经存在。

5. file_name和link_name在不同的设备上。

若无错误发生，则创建一个新的目录项，其文件名为link_name，而i-节点号为文件file_name的i-节点号。在代码中，name1对应于file_name，而name2对应于link_name。新目录项实际上是由do_link在25485行调用的search_dir函数创建的。

解链可以删除文件和目录。系统调用UNLINK和RMDIR的工作都由do_unlink(25504行)来完成。这里，我们依然要进行各种检查；检测文件存在以及目录不是安装点的工作在do_unlink的公共代码中进行，然后根据所支持的系统调用的不同分别执行remove_dir或unlink_file。我们很快还要谈到这些

文件link.c支持的另一个系统调用是RENAME。UNIX用户所熟悉的shell命令mv最终调用了RENAME；其名称还反映了这个系统调用的另一方面：它不仅可以在目录中改变文件名，还能够将文件从一个目录移至另一个目录，这些都是自动完成的。具体工作由do_rename(25563行)实现。在完成这个命令时，需要检测一系列情况，其中有：

1. 源文件必须存在(25578行)。
2. 在目录树中，原路径名不能是新路径名上面的目录(25596行至25613行)。
3. 原路径名和新路径名均不能是"."和".."(25618行)。
4. 他们的父目录必须要在同一个设备上(25622行)。
5. 他们的父目录均可写、可查找，并且都在可写设备上(25625行和25626行)。
6. 原文件名和新文件名都不能是文件系统安装至的目录。

如果新文件名已经存在，我们还要检查另外一些情况，其中最重要的是，我们必须有权删除以新文件名为名字的文件。

在do_rename的代码中，有些例子可以说明我们是如何减少问题出现的可能性的。如果不首先删除原来的文件，将文件名换成一个已经存在的文件名可能导致磁盘满，尽管最终不需要额外的空间，这就是25660行到25666行的代码的目的。25680行是同样道理，在同一目录下创建新文件名之前删除原来的文件，以避免目录申请额外的块。然而，如果新文件名和原文件名位于不同的目录下，所考虑的与此毫不相关。在25685行，我们在删除原文件名前创建一个新文件名(在不同目录下)，因为从系统完整性的观点来看，在系统崩溃时，有两个文件名同时指向一个i-节点，远比有一个节点不被如何目录项所指向的情况要好得多。在执行换名操作时空间耗尽的可能性很小，由此导致系统崩溃的可能性更小，但在这些情况下应付最坏情况并不耗费更多的资源。

文件link.c中剩下的函数为我们上面讨论过的函数提供支持。此外，他们中的第一个函数truncate(25717行)还在文件系统其他几个地方被调用。它在i-节点中查找区段，释放它所找到的所有区段，包括间接块。remove_dir(25777行)进行许多额外的检测，确保目录可被删除，然后调用unlink_file(25818行)。若没有错误，目录项被清空，i-节点中链接数减1。

5.7.6 其他系统调用

我们要讨论的最后一组系统调用涉及状态、目录、保护、时间和其他服务等。

改变目录和文件的状态

文件stadir.c包含四个系统调用的代码，他们是：CHDIR、CHROOT、STAT和FSTAT。在分析last_dir时，我们看到路径查找首先检查第一个字符是否为分隔符，根据结果，将指针指向工作目录或根目录。

从一个工作目录(或根目录)切换到另一个工作目录(或根目录)，只需要修改调用进程的进程表中的相应指针。这一修改在do_chdir(25924行)和do_chroot(25963行)中进行。这两个过程都做一些必要的检查，然后调用change(25978行)打开新目录以取代原来的目录。

在用户进程调用CHDIR时，并不执行do_chdir中的25935行至25951行的代码。它是专门为内存管理器在处理EXEC调用时改变用户目录而编写的。若某用户想在他的工作目录下执行一个文件，比如a.out，对内存管理器来说，切换到这个工作目录要比指出它的位置容易得多。

本文件中的另外两个系统调用，STAT和PSTAT，本质上是相同的，只是指定文件的方式不同：前者给出了路径名，而后者给出的是打开文件的文件描述符。最上层的过程do_stat(26014行)和do_fstat(26035行)都调用stat_inode。在调用之前，do_stat还要打开文件，获得其i-节点。这样，do_stat和do_fstat都向stat_inode传递一个i-节点指针。

stat_inode过程(26051行)从i-节点中提取信息，将它拷贝到缓冲区中。缓冲区很大，无法放在消息中，因而要调用26088行的sys_copy函数将其显式地拷贝到用户空间。

保护

MINIX保护机制使用rwx位。每个文件都有三组rwx位，分别用于文件主、文件主所在的组及其他用户。这些位由系统调用CHMOD设置，而该系统调用由文件protect.c中的do_chmod过程(26124行)执行。在进行一系列有效性检查后，do_chmod最终在26150行改变保护模式。

系统调用CHOWN和CHMOD有些相似，两者都修改某文件内部i-节点的一个域。两者的实现也类似，但是do-chown(26163行)在改变文件拥有者时只能为超级块用户调用。普通用户可以使用这个系统调用改变其文件所在的组。

UMASK系统调用允许用户设置屏蔽标志(保存在进程表中)，该标志将屏蔽此后的CREAT系统调用中的相应

位。完整的实现实际上只需要26209行一条语句，只是该调用需返回其原来的屏蔽值。 这一点使得程序代码量翻了三番(26208行至26210行)。

系统调用ACCESS可由进程用来查找它是否能以某种特定方式访问一个文件(例如读文件)，它在do_access函数(26217行)中实现，do_access取得文件的i-节点，并调用内部过程forbidden(26242行)检查访问是否被禁止。forbidden检查uid、gid以及i-节点的信息。根据这些信息，选取三组rwx中的一组，检查是否允许指定的访问。

read_only(26304行)是一个内部过程，它给出了i-节点参数所在的文件系统是安装为只读还是读写，主要用于防止对安装成只读的文件系统进行写操作。

时间

MINIX中有几个涉及时间的系统调用，他们是UTIME、TIME、STIME和TIMES，都列在图5-43中。尽管这些系统调用中很多与文件无关，但我们把他们都放在文件系统中分析，因为时间信息记录在文件的i-节点中。

图5-43 涉及到时间的四条系统调用。

每个文件都有3个32位数字与其相关联，其中两个记录了文件的最后存取时间和最后修改时间。第三个记录了文件i-节点本身状态最后一次修改的时间，该时间在几乎所有的文件存取时都要修改，唯有READ和EXEC操作例外。这三个时间都保存在i-节点中。利用UTIME系统调用，文件主和超级块用户可以设置文件的存取时间以及修改时间。文件time.c中的do_utime过程(26422行)执行这个系统调用，它取出文件的i-节点，并把时间保存其中。在26450行重置表明需要修改时间的标志，这样，系统就不会每次都花很高代价，去执行不必要的clock_time调用。

实时时钟由内核中的时钟任务来维护，文件系统并不维护实时时钟。因此，获取和设置实时时钟的唯一方法是向时钟任务发送一条消息。这实际上也是do_time和do_stime过程的内容。实时时钟以秒为单位，从1970年1月1日开始计算。

记帐信息亦由内核来维护。在每个时钟周期，它在某个进程中增加一个时钟周期。这个信息可以通过向系统任务发送消息而得到，do_tims过程(26492行)主要执行这一工作。该过程之所以不命名为do_times，是因为大多数C编译器都在外部变量前面加下划线，并且很多链接程序把符号截为8个字符，这使得do_time和do_times无法区分。

其他

文件misc.c中包含了几个系统调用中使用到的过程，这些过程都不宜放在上面说明。系统调用DUP复制文件描述符，换句话说，它创建一个新的文件描述指向其参数所指定的文件。DUP2是DUP的变体。这两个系统调用都在过程do_dup(26632行)中处理，他们现在都不再使用了，在MINIX中仍包含这种功能的目的是支持原来的二进制程序。在C源程序中遇到其中任何一个时，当前版本的MINIX C库函数将调用系统调用FCNTL。

在过程do_fcntl(26670行)中处理的FCNTL是向一个打开文件请求进行某些操作的最佳方式。请求服务时使用图5-44列出的POSIX定义的标志。该过程被调用时，含有文件描述符、请求代码以及特定请求需要的其他附加参数。例如，和原先的调用

```
dup2(fd, fd2);
```

等价的是

```
fcntl(fd, F_DUPFD, fd2);
```

其中几个请求设置或读取标志位，他们的代码很简单，仅有几行。例如，F_SETFD 请求设置一个标志位，使得文件的拥有者进程执行EXEC调用时关闭该文件；F_GETFD 请求则用于检查在调用EXEC时是否要关闭文件；通过F_SETFL和F_GETFL请求可以设置标志位以表明某文件可用于非阻塞模式或者是追加操作。

图5-44 系统调用FCNTL的POSIX请求参数。

do_fcntl还处理文件锁，带有F_GETLK、F_SETLK或F_SETLKW命令的调用被翻译成对lock_op的调用，我们在前面几节中已经讨论过lock_op过程。

下一个系统调用是SYNC，它把自上次读入内存后修改过的所有块和i-节点写回磁盘。这个调用在过程do_sync(26730行)中处理。它搜索所有表，找出修改过的项。i-节点应该首先处理，因为rw_inode把结果保留在块高速缓存中。在所有修改过的i-节点写入到块高速缓存之后，将所有修改过的块都写回磁盘。

系统调用FORK、EXEC、EXIT和SET实际上都是内存管理器调用，但他们运行的结果也同样在这里作些说明。在生成一个进程时，内核、内存管理器以及文件系统都应该知道这一点。这些"系统调用"并不来自用户进程，而是来自内存管理器。do_fork、do_exit和do_set把相关信息记录在进程表的文件系统部分。do_exec搜索所有标志为closed-on-exec的文件，将他们关闭(调用do_close过程)。

本文件中的最后一个函数并不是一个真正的系统调用，但我们将它按系统调用加以处理，这个函数是do_revive(26921行)。一个任务原先无法完成文件系统请求的工作，但是现在完成了该工作(例如为某用户进程提供输入数据)，此时调用本函数。文件系统唤醒用户进程并向它发送响应消息。

5.7.7 I/O设备界面

MINIX中的I/O是向内核任务发送消息来完成的。文件系统与这些任务的接口包含在文件device.c中。在需要设备I/O时，在read_write中调用dev_io(27033行)处理字符设备文件，而在rw_block中调用dev_io中处理块

设备文件。dev_io创建一个标准的消息(见图3-15)并将该消息发送到指定的任务。各个任务在
(*dmap[major].dmap_rw)(task, &dev_mess);
(27056行)中调用。上面的语句通过table.c中dmap数组的指针调用相关函数，这些函数就放在文件device.c中。在dev_io等待任务送回响应时，文件系统也等待。其内部并未实现多道程序机制。但是，通常情况下，这个等待过程非常短(例如50毫秒)。

在打开和关闭设备文件时，根据设备类型的不同，需要进行特别处理。dmap表也用于决定在打开和关闭各种不同的主设备时分别调用哪些函数。对于磁盘设备，不论是软盘、硬盘、或是基于内存的设备，都调用dev_opcl过程(27071行)。代码行
mess_ptr->PROC_NR=fp-fproc;
(27081行)计算调用进程的进程号。实际工作是通过向call_task传递任务号和指向消息的指针完成的。我们在后面还会讨论call_task过程。关闭设备时，也调用dev_opcl。事实上，在本函数层次上，打开设备和关闭设备的唯一不同在于：从call_task返回之后继续做什么。

其他通过dmap结构调用的函数包括用于串行服务的tty_open和tty_close，以及为控制台服务的ctty_open和ctty_close。最后一个函数ctty_close几乎是一个哑函数，它无条件地返回状态OK。

SETSID系统调用要求文件系统完成某些任务，这个调用由过程do_setsid(27164行)来执行。系统调用IOCTL主要放在文件device.c中处理，之所以这样，是因为它和任务界面密切相关。在执行IOCTL时，调用do_ioctl创建一条消息并将其发送给适当的任务。

为了控制终端设备，符合标准POSIX的程序应当使用在头文件terminal.h中定义的一个函数。C库将把这些函数转换为IOCTL调用。非终端设备的许多操作都要用到IOCTL，其中许多已在第三章中介绍。

下面一个函数find_dev(27228行)是本文件中唯一的PRIVATE函数。这是一个小的辅助过程，它从一个完整的设备号中提取出主设备号和次设备号。

大多数设备的读写都在过程call_task(27245行)中进行，call_task调用sendrec将一条消息传送给内存图象中适当的任务。如果这个任务试图唤醒一个进程以响应一个更早的请求，那么这种传送可能会失败。这个进程很可能与当前请求为其而发送的进程不同。若接收到一个不合适的消息，call_task过程将在控制台上显示一条消息。在MINIX的正常操作中，这些消息通常不会出现，但在试图开发一个新的设备驱动程序时可能出现。

设备/dev/tty在物理上并不存在。它只是一个虚拟设备，多用户系统中任何用户都可以引用它，而无需明确指出使用哪一个真实终端。当发送一个引用了/dev/tty的消息时，下一函数call_tty(27311行)找到正确的主、次设备号并在call_task将该消息继续传递下去之前在消息中将设备号替换掉。

本文件中的最后一个函数是no_dev(27337行)，该函数从表中一个设备并不存在的项调用，例如，在一个没有网络支持的机器中访问网络设备时。no_dev返回ENODEV状态，因而防止在访问不存在的设备时系统崩溃。

5.7.8 一般的实用程序

文件系统中还包含有几个一般目的的实用过程，这些实用过程收集在文件utility.c中，在许多地方都要用到这些过程。

第一个过程是clock_time(27428行)，它向时钟任务发送消息查询当前实时时间。因为许多系统调用都以文件名作为参数，所以我们定义了过程fetch_name(27447行)。如果文件名较短，它被包含在用户发送给文件系统的消息之中。若文件名很长，就把指向用户空间中名字的一个指针放在消息中。fetch_name检查这两种情况，获得文件名。

在文件系统中，有两个函数用于处理一般类型的错误。 在文件系统接收到错误的系统调用时调用no_sys函数。而panic则在出现致命性错误时打印一条消息，并要求内核停止操作。

最后两个函数conv2和conv4用于处理Intel和Motorola处理器的不同字节顺序问题。在读写磁盘数据结构，例如i-节点或位图时调用这些过程。创建磁盘的系统所采用的字节顺序记录在超级块中。如果它与本地处理器使用的字节顺序不同，则需要交换顺序。这样，文件系统的其他部分就无需了解磁盘上的字节顺序。

最后一个文件是putk.c，它包含有两个过程，都用来打印消息。由于标准的库过程需要向文件系统发送消息，因此我们不能使用他们。这两个过程向终端任务直接发送消息。在本文件的内存管理器版本中，我们已经看到两个几乎与他们完全一致的函数。

5.8 小结

从外部看来，文件系统是一组文件和目录，以及对文件和目录的操作。文件可以被读写，目录可以被创建和删除，并且可将文件从一个目录移到另一个目录中。大多数现代操作系统都支持层次目录系统，在层次目录系统中，目录中含有子目录，如此循环，直至无穷。

而在内部看，文件系统却迥然不同。文件系统的设计者必须考虑到存储区如何分配以及系统如何记录文件使用了哪些块等等。我们还看到，不同的文件系统具有不同的目录结构。文件系统的可靠性和性能也是一个重要问题。

文件系统的安全和保护对用户和设计者都至关重要。我们讨论了早期系统中的一些安全缺陷以及大多数系统的共同问题。我们还讨论了身份确认、存取控制表、权限以及矩阵模型等等。

最后，我们详细地研究了MINIX文件系统。MINIX文件系统很大，但并不复杂。它从用户进程接收任务请求，

索引过程指针表，接着调用相应过程执行所要求的系统调用。由于其模块结构以及处在核心之外，我们可以将它从MINIX中删除，进行小小的修改后将其用作一个独立的网络文件服务器。

在系统内部，MINIX将数据存放在块高速缓存中，并在顺序存取文件时预读。若高速缓存足够大，在反复存取某些程序，例如编辑时，大多数程序正文都可以在内存中找到。

习　题

1 请给出文件/etc/passwd的5种不同路径名。(提示：考虑目录项"."和"..")

2 在支持顺序文件的系统中常常有文件回绕操作，支持随机存取文件的系统是否也需要该操作？

3 某些操作系统提供系统调用RENAME给文件改名。同样也可以通过将文件拷贝到新文件并删除原文件，而实现文件更名。请问这两种方法有何不同？

4 考虑图5-7中的目录树，如果当前工作目录是/usr/jim，那么相对路径名为../ast/x的文件的绝对路径名是什么？

5 正如书中所提到的，文件的连续分配会导致磁盘碎片。请问这是内零头还是内零头？并将它与前一章的内容作比较。

6 若某操作系统仅支持单一目录，但允许该目录有任意多文件，并且文件名可任意长。请问该操作系统能否模拟一个层次文件系统？如何模拟？

7 空闲磁盘空间可用空闲表或位图来记录。磁盘地址需要D位。某磁盘有B个块，其中F个空闲。在什么条件下，空闲表使用的空间少于位图？设D为16，请确定空闲磁盘空间的百分比。

8 有建议认为每个UNIX文件的第一部分最好和其i-节点放在同一个磁盘块中，这样做有什么好处？

9 文件系统的性能取决于高速缓存的命中率(即在高速缓存中找到所需块的概率)。从高速缓存中读取数据需要1毫秒，而从磁盘上读取需要40毫秒，若命中率为h，给出读取数据所需平均时间的计算公式。并画出h从0到1.0变化时的函数曲线。

10 一软盘有40个柱面，寻道时移过每个柱面花6毫秒。若不采取措施尽量使文件的块在磁盘上紧靠存放，则逻辑上相邻的块平均间隔13个柱面。另一种情况是操作系统尽量将相邻的块放在一起，此时块间的平均距离为2个柱面。设旋转延迟为100毫秒，传输速率为每块25毫秒，则在这两种情况下传输一个100块的文件各需要多长时间？

11 定期紧缩磁盘空间可能带来什么好处？

12 如何修改TENEX，使之避免书中所述的口令问题？

13 在取得学位后，你应聘为某大学计算机中心的主任。该中心刚刚将一种很老的操作系统转移到UNIX。你负责这项工作，在开始工作后十五分钟，你的助手冲进你的办公室，说有些学生已经发现了系统使用的口令加密算法，并将其公布在BBS上。此时你应该作什么？

14 Morris-Thompson保护机制使用n位随机数，可以防止入侵者事先对普通字符串进行加密以破译口令。这种机制对于防止一个学生猜测他机器上的超级块用户口令是否奏效？

15 某计算机系在局域网中连接了许多UNIX机器。任何主机上的用户均可键入如下命令

machine4　who

并使之在主机machine4上运行，而用户不必首先登录到这台远程主机上。这一功能可以由用户的内核将命令和用户uid传送到远程主机上实现。若内核可信(例如，含有硬件保护的分时小型机)，这种方案是否安全？若有些机器是学生的不带硬件保护的PC机，则情况又如何？

16 删除文件时,文件的块常常返回到空闲链中,但不被清除。你认为操作系统是否应该在释放块之前将它清除？请从安全性和性能两方面考虑，并解释各自的影响。

17 我们讨论了三种不同的保护机制：权限、存取控制表以及UNIX的rwx位。对于下面的问题，分别适用于哪些机制。

（a）Ken希望除他的同事以外，任何人都能读取他的文件。

（b）Mitch和Steve希望共享某些秘密文件。

（c）Linda希望公开她的一些文件。

对于UNIX，假设用户被分为教职工、学生、秘书等。

18 考虑下面的保护机制：给每个对象和进程分别赋予一个号码，只有当对象号大于进程号时，进程才可以存取对象。这一机制与书中提到的哪种方案相似？它和这种方案有什么根本不同？

19 在一个用权限保护的系统中，特洛伊木马的攻击能奏效吗？

20 关于i-节点有两种观点，一种认为存储器容量越来越大，价格越来越便宜，所以当打开文件时，直接为i-节点在内存i-节点表中建立一个新拷贝将更快、更方便，没有必要搜索整个i-节点表来判断它是否已经存在。另一方则不同意这种观点。这两种观点哪个对？

21 病毒和蠕虫之间有什么区别？他们各自如何复制？

22 符号链接是间接地指向其他文件和目录的文件。与当前MINIX实现的普通链接不同，符号链接有自己的i-节点，该i-节点指向一个数据块，这个数据块包含有被链接文件的路径，而且符号链接的i-节点使得链接可以有与被链接文件不同的属主和权限。符号链接和它所指向的文件或目录可以位于不同的设备。符号链接不是1990

年POSIX标准的一部分，但可能在将来被加入到POSIX标准中。请在MINIX中实现符号链接。
23 你认为MINIX中定的64M字节的文件大小限制满足不了需要,请使用i-节点中的未用空间实现三次间接来扩展该文件系统。
24 验证设置ROBUST是否会使得文件系统在系统崩溃时更加健壮。这个问题在当前版本的MINIX中还未曾研究，因此两种情况都有可能。考虑修改的块被换出高速缓存，以及修改数据块同时修改i-节点和位图等。
25 filp表的长度当前作为常数NR_FILPS定义在文件fs/const.h中。为了容纳更多的网络用户，你希望增大文件include/minix/config.h中的NR_PROCS。如何把NR_FILPS定义为NR_PROCS的函数？
26 设计一种机制，增加对"外部"文件系统的支持，这样我们可以在MINIX文件系统的目录中安装其他的文件系统，比如MS-DOS的文件系统。
27 假设技术有重大突破，出现了非挥发RAM，即使在掉电时它也能保存数据，而价格和性能等方面也不比常规RAM差。这一改进对文件系统设计有何影响？

第六章　阅读材料和参考文献

前五章中我们已经对操作系统的许多内容作了介绍。本章的目的在于为那些希望对操作系统作进一步研究的读者能提供一些帮助。6.1节罗列了向读者推荐的阅读材料，6.2节按照字母顺序列出了本书中所引用的所有书籍和论文。

除了下面列出的参考文献之外，ACM每年举办的操作系统原理专题研讨会论文集《Proceedings of the n-th ACM Symposium on Operating Systems Principles》，以及IEEE每年举办的分布式计算系统国际会议论文集《Proceedings of the n-th International Conference on Distributed Computing System》都是查阅有关操作系统最新论文的好途径。同样，USENIX操作系统设计和实现专题研讨会也是一个好的信息源。更进一步，《ACM Transactions on Computer Systems》和《Operating Systems Review》这两本学报也常常登载相关的文章。

6.1 推荐的进一步阅读材料

6.1.1 介绍和概论

Brooks, The Mythical Man-Month: Essays on Software Engineering

一本机智、幽默和信息量很大的著作，关于如何避免象某些人那样以一种很困难的方式来编写操作系统，其中有很多好的建议。

Comer, Operation System Design. The Xinu Approach

一本关于Xinu 操作系统的书。Xinu操作系统运行在LSI－11计算机上，它包含有源代码的详细说明，其中包括使用C语言的完整程序。

Corbato, "On Building Systems That Will Fail"

Corbato被誉为分时系统之父，在他接受图灵奖的颁奖演说中，他谈到许多Brooks在其著作《The Mythical Man-Month》中所述及的问题。Corbato的结论是复杂系统将最终失败，而且若想要成功，最重要的就是必须摈弃复杂性，尽力争取设计的简洁和优雅。

Deitel, Operating Systems, 2nd Ed.

一本操作系统的综述性教材。本书在标准内容之外，添加了一些实例，如UNIX、MS－DOS、MVS、OS/2以及Mac OS。

Finkel, An Operating Systems Vade Mecum

关于操作系统的另一本综述性教材。它面向实际应用，写得很好，并且述及到很多本书中讲述的问题。如果想找一本书从不同角度描述同一问题，则它是非常合适的。

IEEE, Information Technology - Portable Operating System Interface (POSIX),
Part 1: System Application Program Interface (API) [C Language]

这是一个标准。其中的有些部分非常易读，尤其是附件B: “合理性和标注”，它对为什么采取这样的方法来解决问题作了许多说明。引用标准的好处之一是，从定义来讲，它是绝对没有错误的。即使是排版过程中导致的一个宏定义名字错，也不能算作错误，因为它是官方认可的标准。

Lampson, "Hints for Computer System Desing"

Buttler Lampson是世界上新型操作系统设计的领导人物之一。他汇集了多年实际经验中的种种启示、建议和指南，并将其集中在这篇睿智和信息量很大的文章中。与Brooks的书一样，这是每一个优秀的系统设计人员的必读材料。

Lewine, POSIX Programmer's Guide

这本书用一种比POSIX标准文档更具可读性的方式描述了POSIX标准，同时还包括一些讨论，如怎样将老的程序转换到POSIX，以及如何在POSIX环境下开发新程序。书中有很多代码实例，包括一些完整的程序。本书还描述了所有POSIX需要的库函数和头文件。

Silberschatz and Galvin, Operating System Concepts, 4th Ed.

这是关于操作系统的另一本教材。它涵盖了进程、存储器管理、文件和分布式系统。其中包含两个实例：UNIX和Mach。封面上画满了恐龙，不知道在九十年代这与操作系统有什么关系。

Stallings, Operating Systems, 2nd Ed.

关于操作系统的又一本教材。它包含了通常操作系统所有的论题，也包括一小部分分布式系统的内容，附录介绍了一些排队理论。

Stevens, Advanced Programming in the UNIX Environment

本书教读者如何使用UNIX系统调用接口和标准C库来编写C程序。所给的例子基于系统V版本4和4.4BSD UNIX。书中详细讨论了这些UNIX实现与POSIX之间的关系。

Switzer, Operating Systems, A Practical Approach.

该书与本书很类似。书中使用伪码对理论概念给出了解释，还包含有关TUNIX的大段C源代码，TUNIX是一个模型化的操作系统。与MINIX不同，TUNIX并不在真实机器上运行，而是在一台虚拟机上运行。在设备驱动程序方面TUNIX没有MINIX那样面向实际，但在其他方面它确实比MINIX走得更远，比如虚拟存储器。

6.1.2 进程

Andrews and Schneider, "Concepts and Notations for Concurrent Programming"

本文探讨了进程和进程间通信，包括忙等待、信号量、管程、消息传递以及其他技术。同时也说明了这些概念如何嵌入在不同的程序设计语言中。

Ben-Ari, Principles of Concurrent Programming

这本小册子专门讨论进程间通信问题，其中有不同的章节讨论互斥性、信号量、管程以及哲学家就餐问题，等等。

Dubois et al. "Synchronization, Coherence, and Event Ordering in Multiprocessors"

这是关于在共享存储器的多处理机系统中的同步问题的指导性材料，但其中的某些思想对单处理机系统和分布存储器系统同样有效。

Silberschatz and Galvi, Operating System Concepts, 4th Ed.

本书的第4到第6章讨论了进程和进程间通信，包括调度、临界区、信号量、管程以及经典进程间通信问题。

6.1.3 输入/输出

Chen et al., "RAID: High Performance Reliable Secondary Storage"

在高端系统中，使用多个磁盘驱动器并行操作以获得高速输入/输出是一个发展潮流。作者就此进行了讨论并从性能、价格和可靠性等方面研究了不同的组织结构。

Coffman et al., "System Deadlocks"

本文简短地介绍了死锁，死锁的产生以及如何预防和检测死锁。

Finkel, An Operating Systems Vade Mecum. 2nd Ed.

本书第5章讨论了输入/输出硬件和设备驱动程序，重点针对终端和磁盘。

Geist and Daniel, "A Continuum of Disk Scheduling Algorithms"

本文给出了一个通用的磁盘臂调度算法，并给出了详细的模拟和实验结果。

Holt, "Some Deadlock Properties of Computer Systems"

本文主要围绕死锁进行讨论。Holt引入了一个可被用于分析某些死锁情况的有向图模型。

IEEE Computer Magazine, March 1994

该期杂志包含8篇关于先进输入/输出系统的文章，其中涉及到仿真、高性能存储器、高速缓存、并行计算机的输入/输出以及多媒体。

Isloor and Marsland, "The Deadlock Problem: An Overview"

这是关于死锁的指导性材料，其中重点讲了数据库系统，并描述了多种模型和算法。

Stevens, "Heuristics for Disk Drive Positioning in 4.3 BSD"

本文详细研究了在Berkeley UNIX中的磁盘性能。正如多数计算机系统一样，实际情况远比理论预测复杂。

Wilkes et al., "The HP AutoRAID Hierarchical Storage System"

RAID（廉价冗余磁盘阵列）是高性能磁盘系统的一项重要进展，在RAID中，由若干小磁盘构成的阵列一起工作，以构造一个高性能系统。本文中作者详细描述了他们在HP实验室里开发的系统。

6.1.4 存储器管理

Denning, "Virtual Memory"

本文是一篇关于虚拟存储器的经典文章，其中论及了许多虚拟存储器的许多方面。Denning是该领域的先躯之一，也正是他创立了工作集概念。

Denning, "Working Sets Past and Present"

本书很好地综述了众多的存储器管理机制和调页算法，书后附有完整的参考文献。

Knuth, The Art of Computer Programming Vol. 1

在本书中，集中讨论和比较了最先适应算法、最佳适应算法和其他算法。

Silberschatz and Galvin, Operating System Concepts, 4th Ed.

本书第8和第9章讨论存储器管理问题，包括对换、调页和分段，其中提到很多页面算法。

6.1.5 文件系统

Denning, "The United States vs. Craig Neidorf"

当一个年轻的“黑客”发现并发表了有关电话网系统的工作原理之后，他被指控为计算机诈骗。本文就讲述了这样一件事情，这中间牵涉到很多基本的问题，包括言论的自由。本文发表后引起了一些非议和反驳。

Hafner and Markoff, Cyberpunk

本书以一名纽约时报计算机记者的口吻讲述了三个杜撰的故事和他自己的记者生涯。，在故事中，年轻的计算机“黑客”闯入了全世界范围的计算机。而这名记者本身曾攻破了Internet蠕虫病毒。

Harbron, File Systems

本书描述文件系统的设计、应用和性能，其中对结构和算法均作了介绍。

McKusick et al., "A Fast File System for UNIX"

UNIX的文件系统在4.2BSD环境下被重新加以实现。本文描述了新文件系统的设计，其中重点放在性能特性上。

Silberschatz and Galvin Operating System Concepts, 4th Ed.

本书第10和11章的内容是文件系统。其中包括文件操作、文件访问方式、一致性语义、目录和保护等，此外还有其他主题的实现。

Stallings, Operating Systems, 2nd Ed.

本书第14章包含很多有关系统环境安全的内容，特别是关于计算机“黑客”、病毒和其他威胁。

6.2 按字母排序的参考文献

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.:
"Scheduler Activations: Effective Kernel Support for the Uer-level Management of Parallelis, " ACM Trans. on Computer Systems, vol. 10, pp. 53-79, Feb. 1992.

ANDRES, G.R., and SCHNEIDER, F.B.:
"Concepts and Notations for Concurrent Programming," Computing Surveys, vol. 15, pp.3-43, March 1983.

BACH, M.J.:
The Design of the UNIX Operating Systems, Englewood Cliffs, NJ: Prentice Hall, 1987.

BALA, K., KAASHOEK, M.F., WEIHL, W.:
"Software Prefetching and Caching for Translation Lookaside Buffers," Proc. First Symp. on Operating System Design and Implementation, USENIX. pp. 243-254, 1994.

BAYS, C.:
"A Comparison of Next-Fit, First-Fit, and Best-Fit," Commun. of the ACM, vol. 20, pp. 191-192, March 1977.

BEN-ARI, M:
Principles of Concurrent Programming, Englewood Cliffs, NJ: Prentice Hall International, 1982.

BRINCH HANSEN, P.:
"The Programming Language Concurrent Pascal," IEEE Trans. on Software Engineering, vol. SE-1, pp. 199-207, June 1975.

BROOKS, F.P., jR.:
The Mythical Man-Month: Essays on Software Engineering, Anniversary edition, Reading, MA: Addison-Wesley, 1996.

CADOW, H.:
OS/360 Job Control Language, Englewood Cliffs, NJ: Prentice Hall, 1970.

CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., and PATTERSON, D.A.:
"RAID: High Performance Reliable Storage," Computing Surveys, vol. 26, pp. 145-185, June 1994.

CHERITON, D.R.:
"An Experiment Using Registers for Fast Message-Based Interprocess Communication," Operating Systems Review, vol. 18, pp.12-20, Oct. 1984.

COFFMAN, E.G., F\ELPHICK, M.J., and SHOSHANI, A.:
"System Deadlocks," Computing Surveys, vol. 3, pp.67-78, June 1971.

COMER, D.:
Operating System Design, The Xinu Approach, Englewood Cliffs, N.J.: Prentice Hall, 1984.

CORBATO, F.J.:
"One Building Systems That Will Fail," Commun. of the ACM, vol. 34, pp. 72-81, June 1991.

CORBATO, F.J., MERWIN-DAGGETT, M., and DALEY, R.C:
"An Experimental Time-Sharing System," Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 335-344, 1962.

CORBATO, F.J., SALTZER, J.H., and CLINGER, C.T.:

"MULTICS-The First Seven Years," Proc. AFIPS Spring Joint Computer Conf., AFIPS, pp.571-583, 1972.
CORBATO, F.J., and VYSSOTSKY, V.A.:
"Introduction and Overview of the MULTICS System," Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 185-196, 1965.
COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.:
"Concurrent Control with Readers and Writers," Commun. of the ACM, vol. 10. 667-668, Oct. 1971.
DALEY, R.C., and DENNIS, J.B.:
"Virtual Memory, Process, and Sharing in MULTICS," Commun. of the ACM, vol. 11, pp. 306-312, May 1968.
DEITEL, H.M.:
Operating Systems, 2nd Ed., Reading, MA: Addison-Wesley, 1990.
DENNING, D.:
"The United states vs. Craig Neidorf," Commun. of the ACM, vol. 34, pp. 22-43, March 1991.
DENNING, P.J.:
"The Working Set Model for Program Behavior," Commun. of the ACM, vol. 11, pp. 323-33, 1968a.
DENNING, P.J.:
"Thrashing: Its Causes and Prevention," Proc. AFIPS National Computer Conf., AFIPS, pp. 915-922, 1968b.
DENNING, P.J.:
"Virtual Memory," Computing Surveys, vol. 2, pp. 153-189, Sept. 1970.
DENNING, P.J.:
"Working Sets Past and Present," IEEE Trans. on Software Engineering, vol. SE-6, pp. 64-84, Jan. 1980.
DENNING, J.B., and VAN HORN, E.C.:
"Programming Semantics for Multiprogrammed Computations," Commun. of the ACM, vol. 9, pp. 143-155, March 1966.
DIJKSTRA, E.W.:
"Co-operating Sequential Processes," in Programming Languages, Genuys, F. (Ed.), London: Academic Press, 1965.
DIJKSTRA, E.W.:
"The Structure of THE Multiprogramming System," Commun. of the ACM, vol. 11, pp. 341-346, May 1968.
DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.:
"Synchronization, Coherence, and Event Ordering in Multiprocessors," IEEE Computer, vol. 21, pp. 9-21, Feb. 1988.
ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J.Jr.:
"Exokernel: An Operating System Architecture for Application-Level Resource Management," Proc. of the Fifteenth Symp. on Operating Systems Principles, ACM, pp. 251-266, 1995.
FABRY, R.S.:
"Capability-Based Addressing," Commun. of the ACM, vol. 17, pp. 403-412, July 1974.
FEELEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEKKATH, C.A.:
"Implementing Global Memory Management in a Workstation CLuster," Proc. of the Fifteenth Symp. on Operating Systems Principles, ACM, pp. 201-212, 1995.
FINKEL, R.A.:
An Operating Systems Vade Mecum, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.
FOTHERINGHAM, J.:
"Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," Commun. of the ACM, vol. 4, pp. 435-436, Oct. 1961.
GEIST, R., and DANIEL, S.:
"A Continuum of Disk Scheduling Algorithms," ACM Trans. on Computer Systems, vol. 5, pp. 77-92, Feb. 1987.
GOLDEN, D., and PECHURA, M.:
"The Structure of Microcomputer File Systems," Commun. of the ACM, vol. 29, pp. 222-230, March 1986.
GRAHAM, R.:
"Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T, Sept. 1970.
HAFNER, K., and MARKOFF, J.:
Cyberpunk, New York: Simon and Schuster, 1991.
HARBRON, T.R.:

File Systems, Englewood Cliffs, NJ: Prentice Hall, 1988.
HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.:
"Using Threads in Interactive Systems: A Case Study," Proc. of the Fourteenth Symp. on Operating Systems Principles, ACM, pp. 94-105, 1993.
HAVENDER, J.W.:
"Avoiding Deadlock in Multitasking Systems," IBM Systems Journal, vol. 7, pp. 74-84, 1968.
HEBBARD, B. et al.:
"A Penetration Analysis of the Michigan Terminal System," Operating Systems Review, vol. 14, pp. 7-20, Jan. 1980.
HOARE, C.A.R.:
"Monitors, An Operating System Structuring Concept," Commun. of the ACM, vol. 17, pp. 549-557, Oct. 1974; Erratum in commun. of the ACM, vol. 18, p. 95, Feb. 1975.
HOLT, R.C:
"Some Deadlock Properties of Computer Systems," Computing Surveys, vol. 4, pp. 179-196, Sept. 1972.
HOLT, R.C:
Concurrent Euclid, The UNIX System, and TUNIS, Reading, MA: Addison-Wesley, 1983.
HUCK, J., and HAYS, J.:
"Architectural Support for Translation Table Management in Large Address Space Machines," Proc. Twentieth Annual Int'l Symp. on Computer Arch., ACM. pp. 39-50, 1993.
IEEE:
Information technology - Protable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], New York: Institute of Electrical and Electronics Engineers, Inc., 1990
ISLOOR, S.S., and MARSLAND, T.A.:
"The Deadlock Problem: An Overview," IEEE Computer, vol. 13, pp. 58-78, Sept. 1980.
KERNIGHAN, B.W., and RITCHIE, D.M.:
The C Programming Language, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.
KLEIN, D.V.:
"Foiling the Cracker: A Survey of, and Improvements to, Password Security," Proc. UNIX Security Workshop II, USENIX, Summer 1990.
KLEINROCK, L.:
Queueing Systems. Vol. 1, New York: John Wiley, 1975.
KNUTH, D.E.:
The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2nd Ed., Reading, MA: Addison - Wesley, 1973.
LAMPSON, B.W.:
"A Scheduling Philosophy for Multiprogramming Systems," Commun. of the ACM, vol. 11, pp. 347-360, May 1968.
LAMPSON, B.W.:
"A Note on the Confinement Problem," Commun. of the ACM, vol. 10, pp. 613-615, Oct. 1973.
LAMPSON, B.W.:
"Hints for Computer System Design," IEEE Software, vol. 1, pp. 11-28, Jan. 1984.
LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J., and WULF, W.A.:
"Policy/Mechanism Separation in Hydra," Proc. of the Fifth Symp. on Operating Systems Principles, ACM, pp. 132-140, 1975.
LEWINE, D.:
POSIX Programmer's Guide, Sebastopol, CA: O'Reilly & Associates, 1991.
LI, K., and HUDAK, P.:
"Memory Coherence in Shared Virtual Memory Systems," ACM Trans. on Computer Systems, vol. 7, pp. 321-359, Nov. 1989.
LINDE, R.R.:
"Operating System Penetration," Proc. AFIPS National Computer Conf., AFIPS, pp. 361-368, 1975.
LIONS, J.:
Lions' Commentary on Unix 6th Edition, with Source Code, San Jose, CA: Peer-to-Peer Communications, 1996.

LIU, C.L., and LAYLAND, J.W.:
"Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, vol. 20, pp. 46-61, Jan. 1973.
MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., and MARKATOS, E.P.:
"First-Class User-Level Threads," Proc. of the Thirteenth Symp. on Operating Systems Principles, ACM, pp. 110-121, 1991.
McKUSICK, M.J., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.:
"A Fast File System for UNIX," ACM Trans. on Computer Systems, vol. 2, pp. 181-197, Aug. 1984.
MORRIS, R., and THOMPSON, K.:
"Password Security: A Case History," Commun. of the ACM, vol. 2, pp. 594-597, Nov. 1979.
MULLENDER, S.J., and TANENBAUM, A.S.:
"Immediate Files," Software - Practice and Experience, vol. 14, pp. 365-368, April 1984.
ORGANICK, E.I.:
The Multics System, Cambridge, MA: M.I.T Press, 1972.
PETERSON, G.L.:
"Myths about the Mutual Exclusion Problem," Information Processing Letters, vol. 12, pp.115-116, June 1981.
ROSENBLUM, M., and OUSTERHOUT, J.K.:
"The Design and Implementation of a Log-Structured File System," Proc. Thirteenth Symp. on Operating System Principles. ACM, pp. 1-15, 1991.
SALTZER, J.H.:
"Protection and Control of Information Sharing in MULTICS," Commun. of the ACM, vol. 17, pp. 388-402, July 1974.
SALTZER, J.H., and SCHROEDER, M.D.:
"The Protection of Information in Computer Systems," Proc. IEEE, vol. 63, pp. 1278-1308, Sept. 1975.
SALUS, P.H.:
"UNIX At 25," Byte, vol. 19, pp. 75-82, Oct. 1994.
SANDHU, R.S.:
"Lattice-Based Access Control Models," Computer, vol. 26, pp. 9-19, Nov. 1993.
SEAWRIGHT, L.H., and MACHINNON, R.A.:
"VM/370 - A Study of Multiplicity and Usefulness," IBM Systems Journal, vol. 18, pp. 4-17, 1979.
SILBERSCHATZ, A., and GALVIN, P.B.:
Operating System Concepts, 4th Ed. Reading, MA: Addison-Wesley, 1994.
STALLINGS, W.:
Operating Systems, 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1995.
STEVENS, W.R.:
Advanced Programming in the UNIX Environment, Reading, MA: Addison-Wesley, 1992.
STEVENS, W.R.:
"Heuristics for Disk Drive Partitioning in 4.3BSD," Computing Systems, vol. 2, pp. 251-274, Summer 1989.
STOLL, C.:
The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage, New York: Doubleday, 1989.
SWITZER, R.W.:
Operating Systems, A Practical Approach, London: Prentice Hall Int'l, 1993.
TAI, K.C., and CARVER, R.H.:
"VP: A New Operation for Semaphores," Operating Systems Review, vol. 30, pp. 5-11, July 1996.
TALLURI, M., and HILL, M.D.:
"Surpassing the TLB Performance of Superpages with Less Operating System Support," Proc. Sixth Int'l Conf. on Architectural Support for Progr. Lang. and Operating Systems, ACM, pp. 171-182, 1994.
TALLURI, M., and HILL, M.D., and KHALIDI, Y.A.:
"A New Page Table for 64-bit Address Spaces," Proc. of the Fifteenth Symp. on Operating Systmes Principles, ACM. pp. 184-200, 1995.
TANENBAUM, A.S.:
Distributed Operating Systems, Englewood Cliffs, NJ: Prentice Hall. 1995.
TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H. VAN, SHARP, G.J., MULLENDER, S.J., JANSEN, J., and

ROSSUM, G.VAN:
"Experiences with the Amoeba Distributed Operating System," Commun. of the ACM. vol. 33, pp. 46-63, Dec. 1990.
TEORY, T.J.:
"Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 1-11, 1972.
THOMPSON, K.:
"UNIX Implementation," Bell System Technical Journal, vol. 57, pp. 1931-1946, July-Aug. 1978.
UHLIG, R., NAGLE, D., STANLEY, T, MUDGE, T., SECREST, S., and BROWN, R:
"Design Tradeoffs for Software-Managed TLBs," ACM Trans. on Computer Systems, vol. 12, pp. 175-205, Aug. 1994.
VAHALIA, U.:
UNIX Internals - The New Frontiers, Upper Saddle river, NJ: Prentice Hall, 1996.
WALDSPURGER, C.A., and WEIHL, W.E.:
"Lottery Scheduling: Flexible Proportional-Share Resource Management," Proc. First Symp. on Operating System Design and Implementation, USENIX, pp. 1-12, 1994.
WILKES, J., GOLDING, R., STAELIN, C, and SULLIVAN, T.:
"The HP AutoRAID Hierarchical Storage System," ACM Trans. on Computer Systems, vol. 14, pp. 108-136, Feb. 1996.
WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK, F.J.:
"HYDRA: The Kernel of a Multiprocessor Operating System," Commun. of the ACM, vol. 17, pp. 337-345, June 1974.
ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.:
"Software Write Detection for a Distributed Shared Memory," Proc. First Symp. on Operating System Design and Implementation, USENIX, pp. 87-100, 1994.
1 若某件事可能出错，则它一定会出错