■教育部高等教育司推荐
■国外优秀信息科学与技术系列教学用书

OPERATING SYSTEM CONCEPTS (Sixth Edition)

操作系统概念 第六版

翻译版

■ Abraham Silberschatz
[美] Peter Baer Galvin 著
Greg Gagne
■ 郑扣根 译

WILEY
高等教育出版社
Higher Education Press

■ 教 育 部 高 等 教 育 司 推 荐
■ 国外优秀信息科学与技术系列教学用书

■ 一流的品质　　■ 优惠的价格

本套教学用书的特点

- 权威性——教育部高等教育司推荐、教育部高等学校信息科学与技术引进教材专家组遴选
- 系统性——覆盖计算机专业主干课程和非计算机专业计算机基础课程
- 先进性——著名计算机专家近两年的最新著作，内容体系先进
- 经济性——价格与国内自编教材相当，是国内引进教材价格最低的

本书讨论了操作系统中的基本概念和算法，并对大量实例（如Linux系统）进行了研究。全书内容共分七部分。第一部分概要解释了操作系统是什么、做什么、是怎样设计与构造的，也解释了操作系统概念是如何发展起来的，操作系统的公共特性是什么。第二部分进程管理描述了作为现代操作系统核心的进程以及并发的概念。第三部分存储管理描述了存储管理的经典结构与算法以及不同的存储管理方案。第四部分I/O系统对I/O进行了深入的讨论，包括I/O系统设计、接口、内部结构与功能等。第五部分分布式系统介绍了分布式系统的一般结构以及连接它们的网络，讨论了分布存取策略、分布式文件系统及分布式系统中同步、通信等机制。第六部分保护与安全介绍了操作系统中对文件、内存、CPU及其他资源进行操作的安全与保护机制。第七部分案例研究，分析与讨论了Linux系统、Windows 2000、Windows XP、FreeBSD、Mach及Nachos等实例。

本书作为操作系统的入门教材，适合所有对操作系统这门学科感兴趣的读者参考，尤其适合高等院校计算机专业及相关专业的学生用做操作系统课程的教材或教学参考书。

ISBN 7-04-013301-6

9 787040 133011 >

定价　55.00元

WILEY

教育部高等教育司推荐
国外优秀信息科学与技术系列教学用书

操作系统概念

（第六版　翻译版）

OPERATING SYSTEM CONCEPTS

(Sixth Edition)

Abraham Silberschatz
[美] Peter Baer Galvin　　著
Greg Gagne
郑扣根　译

高等教育出版社

图字:01－2003－3477 号

Operating System Concepts, Sixth Edition, Simplified Chinese Edition

[美] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne 著,郑扣根 译

本书封面贴有 John Wiley & Sons, Inc.防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

操作系统概念/(美)西尔伯斯查兹(Silberschatz, A.),(美)高尔文(Galvin,P.B.),(美)加尼(Gagne,G.)著;郑扣根译.—北京:高等教育出版社,2004.1(2005 重印)

书名原文:Operating System Concepts,Sixth Edition

ISBN 7－04－013301－6

Ⅰ.操… Ⅱ.①西…②高…③加…④郑… Ⅲ.操作系统 Ⅳ.TP316

中国版本图书馆 CIP 数据核字(2003)第 121589 号

出版发行	高等教育出版社	**购书热线**	010－58581118
社　　址	北京市西城区德外大街 4 号	**免费咨询**	800－810－0598
邮政编码	100011	**网　　址**	http://www.hep.edu.cn
总　　机	010－58581000		http://www.hep.com.cn
经　　销	北京蓝色畅想图书发行有限公司	**网上订购**	http://www.landraco.com
印　　刷	北京外文印刷厂		http://www.landraco.com.cn
开　　本	787×1092　1/16	**版　　次**	2004 年 1 月第 1 版
印　　张	48	**印　　次**	2005 年 11 月第 6 次印刷
字　　数	1 000 000	**定　　价**	55.00 元

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

物料号　13301－00

前　言

20世纪末，以计算机和通信技术为代表的信息科学和技术对世界经济、科技、军事、教育和文化等产生了深刻影响。信息科学技术的迅速普及和应用，带动了世界范围信息产业的蓬勃发展，为许多国家带来了丰厚的回报。

进入21世纪，尤其随着我国加入WTO，信息产业的国际竞争将更加激烈。我国信息产业虽然在20世纪末取得了迅猛发展，但与发达国家相比，甚至与印度、爱尔兰等国家相比，还有很大差距。国家信息化的发展速度和信息产业的国际竞争能力，最终都将取决于信息科学技术人才的质量和数量。引进国外信息科学和技术优秀教材，在有条件的学校推动开展英语授课或双语教学，是教育部为加快培养大批高质量的信息技术人才采取的一项重要举措。

为此，教育部要求由高等教育出版社首先开展信息科学和技术教材的引进试点工作。同时提出了两点要求，一是要高水平，二是要低价格。在高等教育出版社和信息科学技术引进教材专家组的努力下，经过比较短的时间，第一批引进的20多种教材已经陆续出版。这套教材出版后受到了广泛的好评，其中有不少是世界信息科学技术领域著名专家、教授的经典之作和反映信息科学技术最新进展的优秀作品，代表了目前世界信息科学技术教育的一流水平，而且价格也是最优惠的，与国内同类自编教材相当。

这项教材引进工作是在教育部高等教育司和高教社的共同组织下，由国内信息科学技术领域的专家、教授广泛参与，在对大量国外教材进行多次遴选的基础上，参考了国内和国外著名大学相关专业的课程设置进行系统引进的。其中，John Wiley公司出版的贝尔实验室信息科学研究中心副总裁Silberschatz教授的经典著作《操作系统概念》，是我们经过反复谈判，做了很多努力才得以引进的。William Stallings先生曾编写了在美国深受欢迎的信息科学技术系列教材，其中有多种教材获得过美国教材和学术著作者协会颁发的计算机科学与工程教材奖，这批引进教材中就有他的两本著作。留美中国学者Jiawei Han先生的《数据挖掘》是该领域中具有里程碑意义的著作。由达特茅斯学院Thomas Cormen和麻省理工学院、哥伦比亚大学的几

位学者共同编著的经典著作《算法导论》，在经历了11年的锤炼之后于2001年出版了第二版。目前任教于美国Massachusetts大学的James Kurose教授，曾在美国三所高校先后10次获得杰出教师或杰出教学奖，由他主编的《计算机网络》出版后，以其体系新颖、内容先进而倍受欢迎。在努力降低引进教材售价方面，高等教育出版社做了大量和细致的工作。这套引进的教材体现了权威性、系统性、先进性和经济性等特点。

教育部也希望国内和国外的出版商积极参与此项工作，共同促进中国信息技术教育和信息产业的发展。我们在与外商的谈判工作中，不仅要坚定不移地引进国外最优秀的教材，而且还要千方百计地将版权转让费降下来，要让引进教材的价格与国内自编教材相当，让广大教师和学生负担得起。中国的教育市场巨大，外国出版公司和国内出版社要通过扩大发行数量取得效益。

在引进教材的同时，我们还应做好消化吸收，注意学习国外先进的教学思想和教学方法，提高自编教材的水平，使我们的教学和教材在内容体系上，在理论与实践的结合上，在培养学生的动手能力上能有较大的突破和创新。

目前，教育部正在全国35所高校推动示范性软件学院的建设和实施，这也是加快培养信息科学技术人才的重要举措之一。示范性软件学院要立足于培养具有国际竞争力的实用性软件人才，与国外知名高校或著名企业合作办学，以国内外著名IT企业为实践教学基地，聘请国内外知名教授和软件专家授课，还要率先使用引进教材开展教学。

我们希望通过这些举措，能在较短的时间，为我国培养一大批高质量的信息技术人才，提高我国软件人才的国际竞争力，促进我国信息产业的快速发展，加快推动国家信息化进程，进而带动整个国民经济的跨越式发展。

教育部高等教育司

二〇〇二年三月

译 者 序

操作系统是计算机系统的基本组成部分。同样，操作系统课程也是计算机教学的基本组成部分。随着计算机日益广泛的应用，操作系统也正在以惊人的速度发生着变化。如今计算机图书市场上关于操作系统的书非常多，书店中的此类书籍可谓琳琅满目，但真正的好书却凤毛麟角。一本书，能被人引为经典，当然是一本好书。由 John Wiley & Sons 公司出版的贝尔实验室信息科学研究中心副总裁 Silberschatz 教授等人撰写的《操作系统概念(第六版)》就是这样一本经典之作，自第一版问世以来，经历了近 20 年的锤炼，已经成为操作系统教材的一本"圣经"。

该书的影印版是高等教育出版社为配合教育部提出的加快培养大批高质量的信息技术人才的工作所引进的国外优秀信息科学与技术系列教材之一。该书的影印版出版后，受到了广泛的好评，选用本书的多为高等院校研究生院的师生，对其科学性、实用性均给予了高度评价。为了让国人更好地学习和理解书中的知识，并在更广范围内推广使用，高等教育出版社出版了此书的中译本。

作为一本操作系统的经典之作，本书的内容广泛而又重点突出。主要有以下几个特点：

1. 内容全面。全书共分七部分，内容涉及操作系统概念和功能及其设计与构造、进程管理、存储器管理、I/O 系统、分布式系统、保护与安全以及对 Linux、Windows 2000、Windows XP、FreeBSD、Mach 及 Nachos 等实例进行分析与讨论，几乎覆盖了操作系统的各个重要方面。

2. 书中所有提及的原理，都有相应的详细解释，并配有很多实例和插图帮助读者理解，以充实的内容在抽象概念和实际实现之间架设了桥梁。本书讨论了操作系统中的基本概念与算法，提供了大量的实例研究，如 Solaris 2、Linux、MS-DOS、Windows NT、Windows 2000、Windows XP、IBM OS/2 等，为读者深入理解操作系统提供了坚实的理论基础。操作系统本身对许多人来说是枯燥无味的，国人撰书时又常常喜欢将一些浅显的道理深奥化，常给人一头雾水或字典化的感觉。此书却用风趣而智慧的语言讲解许多抽象的概念。

3. 由于该书已连续出版六次，不但每次都对前一次的不足进行了修改，而且还结合当前的技术，增加了最新的内容，因此它的内容和实例并不占老。较之以前的版本，本版本增加了线程、实时操作系统、Windows 2000 等内容。书中所有代码实例均被更新并以 C 语言描述。

4. 此书的写作遵循了循序渐进的原则，结合当今流行的各种操作系统，配有大量的实例和练习，逐步引导读者从一个门外汉变成一个精通操作系统的高手。

整体上看，本书具有内容新、全面、实用、指导性强等特点，不但是从事操作系统应用开

发等专业人士的必备之书，同时也是高等院校相关专业的师生教学的最佳教材。由衷地希望所有读者都能从本书中充分体会到操作系统的精髓，并能在今后的相关工作中游刃有余。

本书的翻译力求忠于作者原意。我们在许多操作系统的专业术语后面的括号中注上了英文原文。这一方面是为了能够方便读者对照理解，为其以后的学习打下基础；另一方面也为了避免以往就存在的不同中文译法带来的歧义，从而节省读者宝贵的时间。

本书由郑扣根教授翻译。在本书的翻译过程中，得到了田稷、冯钢、李祥兵、王晓栋、郑南、方前、李龙连、王万里、徐金星等同志的许多帮助，在此表示深深的谢意。

由于种种原因，书中难免存在错误和不妥之处，恳请读者批评指正。

译 者

2003 年 10 月

原 版 前 言

操作系统是计算机系统的基本组成部分。同样，操作系统课程也是计算机科学教育的基本组成部分。随着计算机在包括从儿童游戏到极为尖端的政府和多国企业的规划工具等领域中的广泛应用，操作系统也正在以惊人的速度发生着变化。然而，操作系统的基本概念仍然是比较清晰的，这些概念是本书所讨论的基础。

本书是一本操作系统导论的教科书，清晰地描述了操作系统的基本概念，适用于本科三、四年级和研究生一年级学生。前提假设读者熟悉数据结构基础、计算机组成和一种高级语言如 C。本书第二章包括了学习操作系统所需的硬件知识。对举例代码，我们主要采用了流行的 C，有时也使用 Java。不过，即使读者没有这些语言的全面知识，也能理解这些算法。

本书所描述的基本概念和算法通常基于用于现代商用操作系统的概念与算法。我们的目的是根据通用操作系统而不是根据特定操作系统来描述这些概念和算法。本书还提供了大量与最通用的操作系统相关的例子，如 Sun Microsystems 的 Solaris 2、Linux；Microsoft 的 MS-DOS、Windows NT、Windows 2000 和 Windows XP；DEC VMS 和 TOPS--20、IBM OS/2 以及 Apple Macintosh 操作系统。

本书不仅直观地描述概念，也包括了重要理论的结论，但省略了正式证明。推荐读物指出了有关研究论文，其中有的首次提出或证明了这些结论，有的是可供进一步阅读的参考材料。有时也使用图和例子来代替证明，以提示我们期望这些问题结果正确的理由。

本 书 内 容

本书共有七大部分：

- **概述**：第一章到第三章解释了操作系统*是什么*、*能做什么*以及它们是如何*设计*与*构造*的。这一部分也解释了操作系统概念是如何发展起来的，操作系统的公共特性是什么，操作系统能为用户做什么，操作系统能为计算机系统操作员做什么。这些描述主要是激励性的、历史性的和解释性的内容。在这些章节中，避免讨论这些问题的内部细节。因此，这部分适合于那些需要学习操作系统是什么而不需要知道其内部算法细节的低年级学生或有关人员。第二章包括一些对学习操作系统来说很重要的硬件知识，对诸如 I/O、DMA 和硬盘操作等硬件知识非常熟悉的读者，可以浏览或跳过这一章。
- **进程管理**：第四章到第八章描述了作为现代操作系统核心的进程以及并发的概念。

进程是系统的工作单元。一个系统由一些并发执行的进程组成，其中有的是操作系统进程（执行系统代码），有的是用户进程（执行用户代码）。这一部分包括进程调度方法、进程间通信、进程同步及死锁处理。这部分还讨论了有关线程的知识。

• **存储管理**：第九章到第十二章主要关于执行期间内存的进程。为了改善 CPU 的利用率及其对用户的响应速度，计算机必须将多个进程同时保存在内存中。存储管理方法有很多，各种算法的有效性与应用情形有关。因为通常情况下内存太小，不能保存所有数据和程序，而且它也不能永久保存数据，所以计算机系统必须提供外存来支持内存。绝大多数现代计算机系统采用磁盘作为主要在线信息（程序和数据）存储媒介。文件系统为在线存储和访问驻留在磁盘上的数据和程序提供了必要的机制。这些章节介绍了存储管理内部所使用的经典算法和结构，并对这些算法进行了深入而实际的描述，包括其特性、优点和缺点。

• **I/O 系统**：第十三章到第十四章描述了与计算机相连的设备和这些设备的多样性。在许多方面，设备是计算机中最慢的部分。由于设备种类如此之多，操作系统需要为应用程序提供大量的功能，以允许它们控制这些设备的各个方面。这部分深入讨论了系统 I/O，包括 I/O 系统设计、接口及系统内部的结构和功能。因为设备是性能瓶颈，所以也讨论了性能问题。另外，还讨论了与外存和第三存储器有关的问题。

• **分布式系统**：第十五章到第十七章讨论了一组不共享内存或时钟的处理器：分布式系统。这类系统允许用户访问系统所维护的各种资源。对共享资源的访问能加快计算，改善数据的可用性和可靠性。这类系统也能为用户提供分布式文件系统：这种系统就是文件服务系统，且其用户、服务器和存储设备位于分布式系统的各个场所。分布式系统必须为进程同步和通信提供各种机制，以处理死锁问题和各种集中式系统未碰到的问题。

• **保护和安全**：第十八章到第十九章解释了为使操作系统中的进程彼此之间不会互相影响，而需要对系统中的进程加以保护。出于保护和安全的目的，采用了这样一种机制：确保只有获得操作系统授权的进程才可以使用相应的文件、内存段、CPU 和其他资源。保护是一种用来控制访问程序、进程或计算机系统资源用户的机制。这种机制必须提供表达控制和实施控制的方法。安全是保护系统所存储的信息（数据和代码）和其他计算机系统的物理资源避免未经授权的访问、恶意破坏和修改以及在意料之外出现的不一致。

• **案例研究**：本书的第二十章到第二十三章和网络上的附录 A 到附录 C 通过描述实际操作系统，融合了本书的概念。这些系统包括 Linux、Windows 2000、Windows XP、FreeBSD、Mach 和 Nachos。之所以选择 Linux 和 FreeBSD 是因为 UNIX 曾经很小，以便于理解，而且不是“玩具”操作系统。它们大多数内部算法的选择主要是基于简单而不是速度和复杂度。可以很容易得到 Linux 和 FreeBSD，因此许多学生都可以访问这些系统。之所以选择 Windows 2000 和 Windows XP 是因为它提供了一个研究现代操作系统的机会，其设计和实现与 UNIX 操作系统有很大不同。本书还选择了 Nachos 系统，该系统允许学生可仔细分析一个操作系统，看看底层如何工作，自己构建部分操作系统，并观察其工作效果。第

二十三章也简要地描述了一些其他有影响的操作系统。

第 六 版

在写第六版时，不但采纳了读者对以前版本的许多改进和建议，而且还加入一些现代操作系统和网络发展的新概念。笔者对绝大多数章节中的内容进行了改写，以反映最新变化，对不再适用的内容做了删除。将以前版本的表达算法的 Pascal 代码改写成现在的 C 代码，还使用了少量的 Java 代码。

笔者对许多章节都做了大量改写和重新组织。最为重要的是，增加了两章，并重新组织了分布式系统的内容。由于网络和分布式系统已渗透到绝大多数操作系统中，因此本书将一些分布式系统的内容如客户机—服务器移出分布式系统，并归并到前面的章节中。

- **第三章，操作系统结构**，现在包括一节专门讨论 Java 虚拟机(Java Virtual Machine，JVM)。
- **第四章，进程**，包括一些小节，以描述套接字和远程程序调用(Remote Procedure Call，RPC)。
- **第五章，线程**，是新增的一章，描述多线程计算机系统。许多现代操作系统现在都允许一个进程包含多个控制线程。
- **第六章**到**第十章**分别是过去的第五章到第九章。
- **第十一章，文件系统接口**，是以前的第十章。笔者对此做了大量修改，包括了分布式文件系统章节(第十六章)的 NFS(Network File System)内容。
- **第十二章**和**第十三章**分别是以前的第十一章和第十二章。在第十三章 I/O 系统中，增加了一节，以描述 STREAM。
- **第十四章，大容量存储器结构**，合并了第十三章和第十四章。
- **第十五章，分布式系统结构**，合并了以前的第十五章和第十六章。
- **第十九章，安全**，是以前的第二十章。
- **第二十章，Linux 系统**，是以前的第二十二章，并经更新，以包括最新发展。
- **第二十一章，Windows 2000**，是新增的一章。
- **第二十二章，Windows XP**，是新增的一章。
- **第二十三章，历史纵览**，是以前的第二十四章。
- **附录 A** 是以前的关于 UNIX 的第二十一章，并经更新，以包括 FreeBSD。
- **附录 B** 涉及 Mach 操作系统。
- **附录 C** 涉及 Nachos 系统。

三个附录通过在线网站提供。

教学辅助材料和网页

本书的网页包括了三个附录、伴随本书的幻灯片(PDF 和 PPF 两种格式)、三个案例研究、最新勘误表以及到作者主页的链接。John Wiley & Sons 公司维护本网页：

http://www.wiley.com/college/silberschatz

为了得到限定的辅助材料，请与本地的 John Wiley & Sons 公司销售代理联系。你可以通过"Find a Rep?"网页(http://www.jsw-edcv.wiley.com/college/findarep)来找到你的代理。

邮 件 列 表

我们提供了一个便于用户进行通信的环境。我们创建了一个邮件列表，包括本书用户及下列地址：os-book@research.bell-labs.com。如果你希望加入该邮件列表，请向 avi@bell-labs.com 发送一个消息，并注明你的姓名、所属机构和电子邮件地址。

建　　议

笔者已设法清除本版中所有错误，但是与操作系统一样，一些隐蔽的错误可能仍然存在。我们非常希望从你那里听到有关所发现的本书中的任何文字错误或疏漏。如果你希望提供建议或提供习题，那么我们也非常高兴。来信请寄 Avi Silberschatz, Vice President, Information Sciences Research Center, MH 2T-310, Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974 (avi@bell-labs.com)。

感　　谢

本书是根据以前版本修改而来的，前三版是与 James Peterson 一起合著的。其他帮助以前版本的包括：Hamid Arabnia, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, P. C. Capon, John Carpenter, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Rasit Eskicioğlu, Hans Flack, Robert Flowler, G. Scott Graham, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Mark Holliday, Richard Kieburtz, Carol Kroll, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Yoichi Muraoka, Jim M. Ng, Banu Ozden, Ed Posnak, Boris Putanec,

Charles Qualline, John Quarterman, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, John Werth 和 J. S. Weston.

我们感谢对本书的改版做出了贡献的以下各位：Bruce Hillyer 审阅和帮助修改了第二章、第十二章、第十三章和第十四章。Mike Reiter 审阅和帮助修改了第十八章。第十四章的部分内容取自 Hillyer 和 Silberschatz 的一篇论文[1996]。第十七章的部分内容取自 Levy 和 Silberschatz 的一篇论文[1990]。第二十章取自 Stephen Tweedie 的未发表的手稿。第二十一章取自 Cliff Martin 的未发表的手稿。第二十二章取自 Dave Probert、Cliff Martin 和 Avi Silberschatz 的未发表的手稿。Cliff Martin 帮助更新了 UNIX 附录，以包括 FreeBSD。Mike Shapiro 审阅了 Solaris 内容，Jim Mauro 回答了多个有关 Solaris 的问题。

我们感谢审阅了本书此版的以下各位：Rida Bazzi (Arizona State University)；Roy Campbell (University of Illinois-Chicago)；Gil Carrick (University of Texas at Arlington)；Richard Guy (UCLA)；Max Hailperin (Gustavus Adolphus College)；Ahmed Kamel (North Dakota State University)；Morty Kwestel (New Jersey Institute of Technology)；Gustavo Rodriguez-Rivera (Purdue University)；Carolyn J. C. Schauble (Colorado State University)；Thomas P. Skinner (Boston University)；Yannis Smaragdakis (Georgia Tech)；Larry L. Wear (California State University, Chico)；James M. Westall (Clemson University)；Yang Xiang (University of Massachusetts)。

我们的编辑，Bill Zobrist 和 Paul Crockett 在准备本书期间给予了专家级指导。他们都得到了 Susannah Barr 的帮助，Susannah Barr 负责本项目的许多具体细节。我们的市场经理是 Katherine Hepburn。高级制作编辑是 Ken Santor。封面制作是 Susan Cyr，封面设计是 Madelyn Lesure。Barbara Heaney 负责管理而 Katie Habib 复印了手稿。校对员是 Katrina Avery(自由职业)；索引员是 Rosemary Simpson(自由职业)。高级制图协调员是 Anna Melhorn。Marilyn Turnamian 帮助生成了图、更新了文字、教师指导书和幻灯片。

最后，我们还希望感谢一些人。Avi 要格外感谢 Krystyna Kwiecien 帮助照顾其母亲，以便他全神贯注地编写本书；Pete 感谢 Harry Kasparian 和其他同事允许他能参加本书的编写而不是其本身的“真正工作”；Greg 感谢在写作本书期间他的孩子所取得的重要成就：Tom，5 岁，学会了阅读；而 Jay，2 岁，学会了讲话。

Abraham Silberschatz, Murray Hill, NJ, 2002

Peter Baer Galvin, Norton, MA, 2002

Greg Gagne, Salt Lake City, UT, 2002

目　录

第一部分　概　述

第二部分 进程管理

第三部分 存储管理

第四部分 I/O系统

第五部分 分布式系统

第六部分 保护与安全

第七部分 案例研究

第一部分　概　　述

*操作系统*是作为计算机硬件和计算机用户之间的中介的程序。操作系统的目的是为用户提供*方便且有效*地执行程序的环境。

首先回顾一下操作系统的发展史：从最初的手工系统(hands on system)，到多道程序设计和分时系统，再到当前手持和实时系统。了解操作系统的发展有助于理解操作系统是做什么的和如何去做的。

操作系统必须确保计算机系统的操作正确。为了防止用户程序干预系统的正确操作，硬件必须提供合适的机制。这里介绍基本的计算机体系结构，以便能编写正确的操作系统。

操作系统为程序和这些程序的用户提供一定的服务，以便于执行这些任务。虽然这些服务随着操作系统不同而有所不同，但是仍然可以识别和研究一些公共类型的服务。

第一章 导 论

操作系统是管理计算机硬件的程序。它还为应用程序提供基础，并且充当计算机硬件和计算机用户的中介。令人惊奇的是操作系统完成这些任务的方式多种多样。大型机操作系统设计的主要目的是为了充分优化硬件的利用率。个人计算机的操作系统是为了能支持复杂游戏、商业应用或位于两者之间的事物。手持计算机的操作系统是为了给用户提供一个可以与计算机方便地交互并执行程序的环境。因此，有的操作系统设计是为了方便，有的设计是为了高效，而有的设计目标是这两者都有。

为了理解操作系统是什么，必须首先了解其发展过程。本章跟踪操作系统的发展：从最初手工系统，到多道程序设计和分时系统，再到个人计算机和手持计算机。也会讨论其他类型的操作系统，如并行的、实时的、嵌入式的系统。随着讨论的不断深入，会发现操作系统的各个部分是对早期计算机系统问题非常自然的解决方式。

1.1 操作系统是什么

操作系统是几乎所有计算机系统的一个重要组成部分。计算机系统可以粗分为四个部分：*硬件*、*操作系统*、*应用程序*和*用户*（图 1.1）。

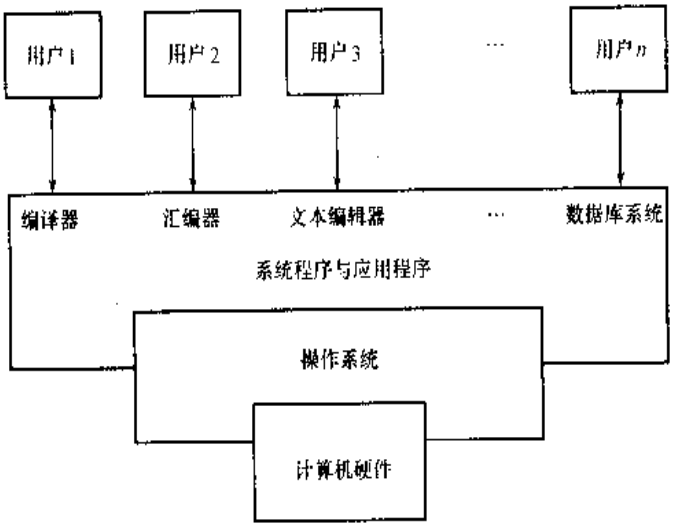


图 1.1 计算机系统组成部分的逻辑图

硬件，如**中央处理单元**(central processing unit，CPU)、**内存**(memory)、**输入/输出设备**(input/output device，I/O device)，提供基本的计算资源。**应用程序**，如字处理程序、电子制表软件、编译器、网络浏览器，规定了按何种方式使用这些资源来解决用户的计算机问题。操作系统控制和协调各用户的应用程序对硬件的使用。

计算机系统的组成部分包括硬件、软件及数据。在计算机系统的操作过程中，操作系统提供了正确使用这些资源的方法。操作系统类似于*政府*。与政府一样，操作系统本身并不能实现任何有用的功能。它只不过提供了一个方便其他程序做有用工作的环境。可以从用户和系统两个观点来研究操作系统。

1.1.1 用户观点

计算机的用户观点根据所使用界面的不同而异。绝大多数计算机用户坐在个人计算机前，个人计算机有显示器、键盘、鼠标和主机。这类系统设计是为了让单个用户单独使用其资源，优化用户所进行的工作(或游戏)。对于这种情况，操作系统的设计目的主要是为了用户**使用方便**，性能是次要的，而且不在乎资源利用率。确切地说，性能对用户来说是非常重要，但是如果系统的绝大部分处于空闲以等待用户相对较慢的I/O操作，那么性能就无关紧要了。

有些用户坐在与**大型机**或**小型机**相连的终端前，其他用户通过其他的终端访问同一计算机。这些用户共享资源并可交换信息。操作系统的设计目的是使**资源利用率**最大化：确保所有的CPU时间、内存和I/O都能得到充分使用，并且确保没有用户使用超过其限额以外的资源。

另一些用户坐在**工作站**前，工作站与其他工作站和服务器相连。这些用户不但可以使用专用的资源，而且可以使用共享资源，如网络和服务器：文件、计算和打印服务器。因此，这类操作系统的设计目的是个人可用性和资源利用率的折中。

近来，许多类型的手持计算机开始成为时尚。绝大多数这些设备为单个用户所独立使用。有的也通过有线或(更为常见)无线与网络相连。由于受电源和接口所限，它们只能执行相对少的远程操作。这类操作系统的设计目的主要是个人可用性，当然如何在有限的电池容量中发挥最大的效用也很重要。

有的计算机几乎没有或根本没有用户观点。例如，在家电和汽车中所使用的嵌入式计算机可能只有一个数值键盘，只能打开和关闭灯来显示状态，而且这些设备及其操作系统通常设计成无需用户干预就能执行。

1.1.2 系统观点

从计算机的角度来看，操作系统是与硬件最为密切的程序。可以将操作系统看做**资源分配器**。计算机系统有许多资源：硬件和软件。这些资源用来解决CPU时间、内存空间、文

件存储空间、I/O设备等问题。操作系统管理这些资源。面对许多甚至冲突的资源请求，操作系统必须决定如何为各个程序和用户分配资源，以便计算机系统能有效而公平地运行。

操作系统的一个稍稍不同的观点是强调控制各种I/O设备和用户程序的需要。操作系统是控制程序。**控制程序**管理用户程序的执行，以防止错误和计算机的使用不当。它特别关心I/O设备的操作和控制。

然而，一般来说，没有一个关于操作系统的充分完整的定义。操作系统之所以存在是因为它们是解决创建可用的计算系统问题的合理途径。计算机系统的基本目的是执行用户程序并能更容易地解决用户问题。为了实现这一目的，构造了计算机硬件。由于仅仅有硬件并不一定容易使用，因此开发了应用程序。这些应用程序需要一些共同操作，如控制I/O设备。这些共同的控制和分配I/O设备资源的功能集合组成了一个软件模块：操作系统。

另外，也没有一个广泛接受的究竟什么属于操作系统的定义。一种简单观点是操作系统包括当你预定一个"操作系统"时零售商所装的所有东西。存储(内存、磁盘和磁带)设备和所包括的功能随系统不同而不同。(系统存储容量通常按GB来计量。(1 KB=1 024 B，1 MB=1024^2 B，1 GB=1024^3 B；但是计算机制造商通常认为1 MB=10^6 B，1 GB=10^9 B))。有的系统只有不到1 MB的空间并且没有全屏编辑器，而其他系统需要数百兆字节空间，并且完全采用图形界面系统。一个比较公认的定义是操作系统是一直运行在计算机上的程序(通常称为**内核**)，其他程序则为应用程序。这一定义是人们通常所采用的。现在，什么组成了操作系统这个问题变得重要了。1998年，美国司法部控告微软公司。简单地说，微软公司将过多的功能加到操作系统中，因此妨碍了其他应用程序开发商的公平竞争。

1.1.3 系统目标

虽然用操作系统"能做什么"比"它是什么"来定义操作系统会更加容易，但是这也有点麻烦。有的操作系统的主要目的是*方便用户*。操作系统之所以存在，是因为它们使得用户的计算变得更加容易。如果分析一下用于个人计算机的操作系统，那么这种观点就很清楚了。

其他操作系统的主要目的是计算机系统的高效执行。大规模、共享、多用户的操作系统就是如此。这些系统比较昂贵，因此需要使得它们尽可能地高效。方便和高效这两个目的有时是互相矛盾的。在过去，高效比方便更为重要(1.2.1小节)。因此，许多操作系统理论主要集中在如何优化计算资源的使用上。随着时间的推移，操作系统也不断发展。例如，UNIX开始主要以键盘和打印机为接口，限制了用户方便性。随着时间的推移，硬件变化了，UNIX也移植到具有更为友好用户的接口的新硬件上。增加的许多**图形用户接口**(全称GUI)使得UNIX不但高效也对用户更为方便。

操作系统的设计是个复杂任务。设计者在设计和实现时面临着诸多权衡，许多人员不但参与创建操作系统，也参与不断修改和更新操作系统。一个操作系统是否满足其设计目

的是值得讨论的，这是由操作系统的不同用户来判断的。

为了知道操作系统是什么和能做什么，现在来研究一下操作系统在过去 45 年内的发展状况。通过跟踪发展，能够识别出操作系统的共同部分，并知道这些系统是如何和为什么发展成现在这样。

操作系统和计算机体系结构相互之间的影响很大。为了方便使用硬件，研究人员开发了操作系统。操作系统的使用者又提出对硬件设计的改进以简化操作系统。通过简短的历史回顾，注意到操作系统中问题的发现往往会导致引入新的硬件功能。

1.2 大型机系统

大型计算机系统(mainframe computer system)是最早的计算机系统，用于处理许多商业和科学应用。在本节，将跟踪大型机系统的发展：从简单的**批处理系统**(只能处理一个应用)到**分时系统**(能允许多个用户同时使用计算机系统)。

1.2.1 批处理系统

早期从终端执行的计算机非常庞大。常用输入设备是卡片阅读机和磁带驱动器。常用输出设备是行式打印机、磁带驱动器和卡片穿孔机。用户不是直接与计算机系统进行交互，而是先准备一个作业，该作业包括程序、数据和一些有关作业性质的控制信息(控制卡)，然后提交给计算机操作员。作业通常是用穿孔卡片来写的。过一段时间(数分钟、数小时或数天)后，出现了输出。这一输出由程序结果和用于调试的最后内存和寄存器内容的信息转储所组成。

这些早期计算机的操作系统相当简单。其主要任务是自动地将控制从一个作业转移到下一个作业。这种操作系统总是驻留在内存中(图 1.2)。

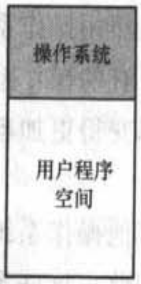


图 1.2 简单批处理系统的内存分布

为了加快处理，操作员按类似需要将作业**分成批次**，并按批或组通过计算机来运行它们。因此，程序员将程序交给操作员后，操作员按类似需求对程序进行排序，当计算机有空时，就按批来运行程序。每个程序的输出再送回给有关的程序员。

在这种执行环境下，CPU 经常空闲，这是因为机械 I/O 设备的速度比电子设备的速度要慢很多。即使一个较慢的 CPU 也会按微秒来运行，每秒能执行数千条指令。另一方面，一个快速的卡片阅读机每分钟可读 1 200 张卡片(或每秒 20 张卡片)。因此，CPU 和 I/O 设备的速度差异为三个数量级甚至更多。当然，随着时间的推移、技术的改善和磁盘的出现会产生更快的 I/O 设备。然而，CPU 的速度增加更快，这样这个差异问题非但没有解决，反而加剧了。

磁盘技术的出现允许操作系统可以将所有作业放在磁盘上，而不必再从卡片阅读机中按顺序读入。由于能直接访问数个作业，操作系统能够进行**作业调度**，以充分有效地使用资源和执行任务。这里简要讨论一些有关作业和 CPU 调度的概念；在第六章将对它们进行深入探讨。

1.2.2 多道程序系统

作业调度最重要的一点是要有多道程序能力。单个用户通常不能使得 CPU 和 I/O 设备一直都忙。**多道程序设计**通过组织作业以使 CPU 总有一个作业可执行，从而提高了 CPU 的利用率。

这种思想如下：操作系统同时将多个作业保存在内存中（图 1.3）。该作业集合是位于作业池中作业集合的子集，这是因为可同时保存在内存中的作业数要比可在作业池中的作业数少。操作系统选择一个位于内存中的作业并开始执行。最终，该作业可能必须等待另一个任务（如 I/O 操作）的完成。对于非多道程序系统，CPU 就会空闲；对于多道程序系统，操作系统会简单地切换到另一个作业并执行。当该作业需要等待时，CPU 会切换到另一个作业，等等。最后，第一个作业完成等待且重新获得 CPU。只要有一个任务可以执行，CPU 就决不会空闲。

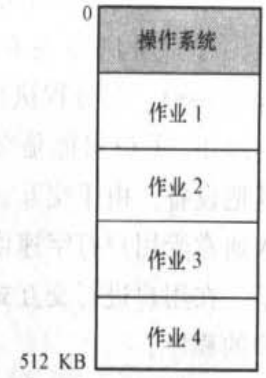


图 1.3 多道程序系统的内存分布

这种思想在日常生活中也很常见。一个律师在一段时间内不只为一个客户工作。当一个案件需要等待审判或需要准备文件时，该律师可以处理另一个案件。如果有足够多的客户，那么他就决不会因没有工作要做而空闲。（空闲律师会成为政客，因此让律师忙碌有一定的社会价值。）

多道程序设计是操作系统必须为用户做出决定的第一个例子。因此多道程序操作系统比较高级。进入系统的所有作业都保存在作业池中。该池由所有驻留在磁盘中需要等待分配内存的作业组成。如果多个作业需要调入内存但没有足够的内存，那么系统必须在这些作业中做出选择。做出这样的决定称为*作业调度*，这将在第六章讨论。操作系统从作业池中选中一个作业，将它调入内存来执行。在内存中同时有多个程序可运行，需要一定形式的内存管理，这将在第九章和第十章讨论。另外，如果有多个作业就绪同时运行，那么系统必须做出选择。做出这样的决定称为*CPU 调度*，这将在第六章讨论。最后，多个并发运行的作业要求操作系统的进程调度、磁盘存储和内存管理等各阶段能够限制作业的互相影响。有关这些的讨论将贯穿本书。

1.2.3 分时系统

多道程序系统、批处理系统提供了一个可以充分利用各种系统资源（例如，CPU、内存、

外设)的环境,但是它没有为用户提供与计算机系统直接交互的能力。**分时系统**(或**多任务**)是多道程序设计的自然延伸。虽然CPU还是通过在作业之间的切换来执行多个作业,但是由于切换非常之快,用户可以在每个程序运行期间与之进行交互。

交互式(或开放式)计算机系统提供用户与系统之间的直接通信。用户通过键盘或鼠标向操作系统或程序直接发出指令,并等待即时结果。相应地,**响应时间**(response time)应该比较短,通常为1 s左右。

分时操作系统允许许多用户同时共享计算机。由于分时系统的每个动作或命令都较短,因而每个用户只要少量CPU时间。随着系统从一个用户快速切换到另一个用户,每个用户会感到整个系统只为自己所用,尽管它事实上为许多用户所共享。

分时操作系统采用CPU调度和多道程序设计,以提供给每个用户一小部分的分时计算机。每个用户在内存中至少有一个程序。装入到内存并执行的程序通常称为**进程**(process)。当进程执行时,它通常只执行较短的一段时间,此时它并未完成或者需要进行I/O操作。I/O可能是交互式的,即输出是给用户的显示器,输入是来自用户的键盘、鼠标或其他设备。由于交互式I/O通常按"人的速度"来运行,因此它需要很长时间完成。例如,输入通常受用户打字速度的限制;每秒7个字符对人来说可能很快,但是对计算机来说相当慢了。在用户进行交互式输入时,操作系统为了不让CPU空闲,会将CPU迅速切换到其他用户的程序。

分时操作系统比多道程序操作系统更为复杂。对于这两种系统,多个作业都必须同时保存在内存中,因此系统必须提供存储管理和保护(第九章)。为了获得合理的响应时间,可能要将作业换入内存或由内存换出到磁盘,磁盘这时作为内存的备用存储。实现这一目的的常用方法是**虚拟内存**(virtual memory),虚拟内存是一种允许一个作业的执行不必完全在内存中进行的技术(第十章)。虚拟内存方案的主要优点是程序可以比**物理内存**大。再者,它将内存抽象成一个庞大且统一的存储阵列,将从用户观点看的**逻辑内存**与真正的物理内存区分开来。这种安排使得程序员不必为内存空间的限制而担心。

分时操作系统也必须提供文件系统(第十一章和第十二章)。文件系统驻留在磁盘组上,因此也必须提供磁盘管理(第十四章)。另外,分时系统要提供并发执行机制,这需要高级CPU调度方案(第六章)。为了确保有序执行,系统必须提供实现作业同步和通信的机制(第七章),它也要确保作业不会进入死锁而无尽地互相等待(第八章)。

虽然分时思想早在1960年就已得到验证,但是由于创建分时系统比较困难和昂贵,所以直到20世纪70年代初才比较常见。虽然有时要做一些批处理,但是现在绝大多数系统都是分时的。相应地,多道程序设计和分时是现代操作系统的主题,也是本书的主题。

1.3 桌面系统

个人计算机(personal computer,PC)出现于20世纪70年代。在开始的十年内,个人计

算机的CPU缺少一些必要功能来保护操作系统不受用户程序的干扰。因此，个人计算机操作系统不是多用户的，也不是多任务的。然而，随着时间的推移，这些操作系统的目标也发生了变化：这些系统的目的不再是最大化CPU和外设的利用率，而是最大化用户方便性和响应速度。这些系统包括运行微软公司Windows和苹果公司Macintosh的PC。微软公司用各种类型的Windows替代MS-DOS操作系统，IBM公司也把MS-DOS升级到了多任务系统OS/2上。Apple Macintosh操作系统也移植到更为高级的硬件，并加上了许多新功能，如虚拟内存和多任务。随着MacOS X的推出，其操作系统核心已改成基于Mach和FreeBSD UNIX，以提高可伸缩性、性能和功能，同时保留了同样丰富的GUI。Linux，一个用于个人计算机的类似于UNIX的操作系统，近来也非常受欢迎。

这些计算机的操作系统都得益于大型机操作系统的研制成果。微型计算机能够很快地采用这些为大型操作系统开发的技术。另一方面，微型计算机的硬件费用很低，所以每人都可拥有一台计算机，从而CPU的利用率也不再是主要问题。所以，有些大型机操作系统的设计决策可能不再适用于小系统。

当然也有一些设计决策仍然适用。例如，文件保护在开始时并不是个人计算机所必需的。然而，现在这些计算机常常通过局域网或因特网与其他计算机相连。当其他计算机和用户能访问某台个人计算机的文件时，文件保护就成为操作系统必不可少的组成部分。如果没有保护，操作系统(如MS-DOS和Macintosh)会很容易地让恶意程序破坏系统上的数据。这些程序可以自我复制，可以通过**worm**或**virus**机制快速传播，会破坏整个公司甚至全世界的网络。高级的分时功能(如保护内存和文件许可)本身并不足以保护系统以避免攻击。近来安全事件一次次地说明了这一点。这些问题将在第十八章和第十九章中加以讨论。

1.4 多处理器系统

绝大多数现代系统都属于单处理器系统，即只有一个主CPU。不过，**多处理器系统**(也称为**并行系统**(parallel system)或**紧耦合系统**(tightly coupled system))的重要性也日益突出。这类系统有多个紧密通信的处理器，它们共享计算机总线、时钟，有时还有内存和外设等。

多处理器系统有三个主要优点：

1. **增加计算量**：通过增加处理器的数量，希望能在更短的时间内做更多的事情。用N个处理器的加速比不是N，而是比N小。当多个处理器工作在同一件事情上时，为了使得各部分能正确工作会产生一定的额外开销。这些开销，加上对共享资源的竞争，会降低因增加了处理器的期望增益。类似地，一组N位程序员在一起紧密地工作并不能完成N倍的单个工作量。

2. **规模经济**：多处理器系统能比多个单处理器系统节省资金，这是因为它们能共享外设、大容量存储和电源供给。当多个程序需要操作同一组数据时，如果将这些数据放在同一磁盘并让多个处理器共享会比用许多有本地磁盘的计算机和多个数据拷贝更为节省。

3. **增加可靠性**：如果将功能正确分布在多个处理器上，那么单个处理器的失灵不会使得整个系统停止，反使其变慢。如果有十个处理器并且其中一个出了故障，那么剩下的九个会分担起故障处理器的那部分工作。因此，整个系统只是比原来慢了10%，而不是完全失败。这种能提供与正常工作的硬件成正比服务的能力称为**功能退化**(graceful degradation)。具有功能退化特征的系统称为**容错系统**(fault tolerant)。

在出错时能继续工作需要一定的机制来对故障进行检测、诊断和纠错(如果可能)。复式系统通过使用硬件和软件的备份来确保在故障时也能继续工作。该系统有两个相同的处理器，每个都有自己的本地内存。处理器通过总线相连。一个处理器为主，另一个处理器为备份。每个进程都有两个拷贝：一个在主处理器上，另一个在备份处理器上。在系统执行的固定检查点，每个作业的状态，包括其内存映像拷贝，都从主机复制到备份机上。如果出现故障，那么就激活备份拷贝并从最近的检查点开始执行。这种解决方案比较昂贵，因为它使用了相当多的备份硬件。

现在最为普遍的多处理器系统使用**对称多处理**(symmetric multiprocessing，SMP)，即每个处理器都运行同一个操作系统的拷贝，这些拷贝根据需要互相通信。有的系统使用**非对称多处理**(asymmetric multiprocessing)，即每个处理器都有各自特定的任务。一个主处理器控制系统；其他处理器或者向主处理器要任务或做预先固定的任务。这种方案称为主－从关系。主处理器为从处理器调度和安排工作。

SMP 意味着所有处理器对等；处理器之间没有主－从关系。每个处理器并发运行一个操作系统拷贝。图 1.4 显示了一个典型的 SMP 结构。一个典型的 SMP 例子是用于 Multimax 计算机的 Encore 版 UNIX。这种计算机可配置成使用数十个处理器，并且都运行 UNIX 拷贝。这种模型的好处是多个处理器可同时运行，如果有 N 个 CPU，那么 N 个进程可以同时运行且并在性能上不会有多大损失。然而，必须仔细控制 I/O，以确保数据到达合适的处理器。另外，由于各 CPU 互相独立，一个可能空闲而另一个可能过载，导致效率低。如果处理器共享一定的数据结构，那么可以避免这种低效率。这种形式的多处理器允许进程和资源(包括内存)在各处理器之间动态共享，能够降低处理器之间的差异。这样的系统需要仔细设计，如第七章所述。目前几乎所有现代操作系统，包括 Windows NT、Solaris、Digital UNIX、OS/2、Linux 等，都支持 SMP。

对称与非对称多处理之间的差异可能是由于硬件也可能是由于软件的原因。特定的硬件可以区分多个处理器，软件也可编写成选择一个处理器为主要的，其他的为次要的。例如，在同样的硬件上，Sun 操作系统 SunOS V4 只提供非对称多处理，而 SunOS V5(Solaris 2)则提供对称多处理。

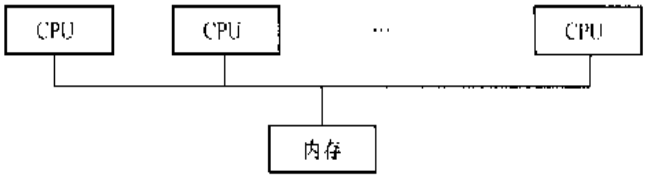


图 1.4 对称多处理体系结构

随着微处理器的功能日益强大且价格低廉，一些附加的操作系统功能可移到从处理器(或后端处理器)。例如，很容易增加一个带有内存的微处理器来管理磁盘系统。该微处理器从主 CPU 收到一系列请求，实现自己的磁盘队列和调度算法。这种方案减轻主 CPU 用于磁盘调度的开销。个人计算机键盘有一个微处理器，以将击键转换成代码并送给主 CPU。事实上，这种形式的微处理器的使用非常普遍，以致于人们并不认为这是多处理了。

1.5 分布式系统

网络，简单地说，就是两个或多个系统之间的通信路径。分布式系统通过网络提供功能。由于可以通信，分布式系统能共享计算任务，并向用户提供丰富的功能。

网络随所使用的协议、节点距离、传输媒介的变换而不同。TCP/IP 是最常用的网络协议，ATM 和其他协议也广泛应用。同样，操作系统对协议的支持也不同。绝大多数操作系统支持 TCP/IP，如 Windows 和 UNIX 操作系统。有的系统只支持专用协议，以满足其需求。对于操作系统而言，一个网络协议只是一个网络接口，如网络适配器，加上管理它的驱动程序以及按通信协议打包数据并发送、解包数据并接收的软件。这些概念将在本书中进行讨论。

网络可根据节点间的距离来划分。**局域网**(local-area network，LAN)位于一个房间、一楼层或一栋楼内。**广域网**(wide-area network，WAN)通常位于楼群之间、城市之间或国家之间。一个全球性的公司可以用 WAN 将其全世界范围内的办公室连起来。这些网络可能运行单个或多个协议。新技术的不断出现带来新型网络。例如，**城域网**(metropolitan-area network，MAN)可以将一个城市内的楼宇连接起来。蓝牙(BlueTooth)设备可以在数英尺的短距离内实现通信，本质上创建了**小域网**(small-area network，SAN)。

架构网络的媒介各具特色。它们包括铜线、光纤电缆和用于卫星、微波碟与广播间无线传输的介质。当计算设备同便携式电话相连时，它们构建形成一个网络。即便是短距离红外线通信也能够用于组建网络。从基本层面上来讲，只要计算机相互通信，它们就使用或构成了一个网络。这些网络在其性能和可靠性上也各有差异。

1.5.1 客户机－服务器系统

随着个人计算机变得更快、更强大和更便宜，设计者开始抛弃中心系统结构。与中心系

统相连的终端开始为个人计算机所取代。相应地，过去为中心系统直接所处理的用户接口功能也日益被个人计算机所取代。因此，今天中心系统成为**服务器系统**(server system)，以满足**客户机系统**(client system)产生的请求。常用的客户机－服务器结构如图 1.5 所示。

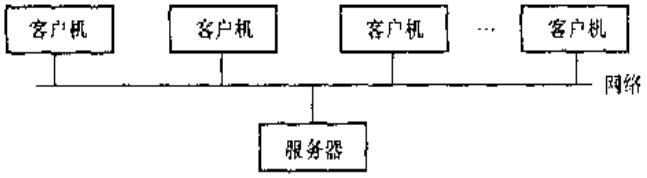


图 1.5 客户机－服务器系统的通用结构

服务器系统可粗分为计算服务器和文件服务器。

• **计算服务器系统**提供了一个接口，以接收用户所发送的执行操作的请求，执行操作，并将操作结果返回给客户机。

• **文件服务器系统**提供文件系统接口，以便客户机能创建、更新、读访问和删除文件。

1.5.2 对等系统

计算机网络的发展，尤其是因特网和万维网的发展，对现代操作系统的发展产生了重大影响。当个人计算机在 20 世纪 70 年代引入时，它们是为个人使用所设计的，并且通常被认为是独立的计算机。随着因特网在 20 世纪 80 年代被人们广泛地用于电子邮件、文件传输和 Gopher(基于菜单驱动的因特网信息查询工具)，许多个人计算机都与计算机网络相连。随着 20 世纪 90 年代中所出现的万维网技术，网络连接已成为计算机系统的基本组成部分。

现在几乎所有个人计算机和工作站都能运行万维网浏览器，以访问万维网的**超链接**文档。操作系统(如 Windows、OS/2、MacOS 和 UNIX)现在都有系统软件(如 TCP/IP 和 PPP)，以便通过局域网或电话线与因特网相连。许多都自带网页浏览器，还有电子邮件、远程登录和文件传输的客户程序和服务程序。

与 1.4 节所讨论的紧耦合系统不一样，这些应用所使用的计算机网络由一组不共享内存和时钟的处理器所组成。这些处理器有各自的本地内存。它们通过各种通信线路，如高速总线或电话线，来互相通信。这些系统通常称为**松耦合系统**(loosely coupled system)(或**分布式系统**(distributed system))。

有的操作系统已进一步采用网络和分布式系统的概念，而不仅是提供网络连接的概念。**网络操作系统**(network operating system)是一种可以提供许多功能(如跨网络文件共享并包括允许不同进程在不同计算机上的交换信息的通信方案)的操作系统。一个运行网络操作系统的计算机虽然知道网络的存在且也能与其他计算机相互通信，但是它与网络上的其他计算机相对独立。分布式操作系统的环境独立性较淡：不同操作系统紧密通信，有一种只有一个操作系统控制网络的假象。在第十五章到第十七章，会深入讨论计算机网络和分布式系统。

1.6 集群系统

与并行系统一样，**集群系统**(clustered system)将多个 CPU 集中起来完成计算任务。然而，集群系统与并行系统不同，它是由两个或多个独立的系统耦合起来的。集群的定义尚未定形，许多商业软件对什么是集群系统及为什么一种形式的集群比另一种好有不同的理解。通常接受的定义是集群计算机共享存储并通过 LAN 网络紧密链接。

集群通常用来提供**高可用性**(high availability)。一层集群软件运行在集群节点之上。每个结点都能监视(通过局域网)一个或多个其他节点。如果被监视的机器失效，那么监视机器能取代存储拥有权，并重新启动在失效机器运行的应用程序。虽然失效机器可以一直停止，但是应用程序的用户和客户机只感觉到很短暂的服务中断。

对于**非对称集群**(asymmetric clustering)，一台机器处于**热备份模式**(hot standby mode)，而另一台运行应用程序。热备份主机(机器)不做什么，只监视现役服务器。如果该服务器失效，那么热备份主机会成为现役服务器。对于**对称集群**(symmetric clustering)，两个或多个主机都运行应用程序，它们互相监视。这种模式因为充分使用了现有硬件，所以更为高效。这要求可以选择多个应用程序来运行。

其他形式的集群有并行集群和 WAN 集群。并行集群允许多个主机访问共享存储上的相同数据。由于绝大多数操作系统不支持多个主机同时访问数据，并行集群通常需要由特定版本的软件和特定发布的应用程序来完成。例如，Oracle Parallel Server(一种版本的 Oracle 数据库)，设计成可运行在并行集群上。每个机器都运行 Oracle，且有一层软件跟踪共享磁盘的访问。每个机器对数据库内的所有数据都可以完全访问。

不管分布式计算如何改善，绝大多数系统并不提供通用分布式文件系统。因此，绝大多数集群不允许对磁盘上的数据进行共享访问。因此，分布式文件系统必须提供对文件的访问控制和加锁，以确保不出现互为矛盾的操作。这种类型的服务通常称为**分布式锁管理器**(distributed lock manager，DLM)。有关通用分布式文件系统的研究正在进行，有的企业如 Sun 微系统公司通告了关于在操作系统内提供 DLM 的路线图。

集群技术发展迅速。集群方向包括全球集群，即世界上(或者 WAN 或到的任何地方)所有计算机组成集群。这些项目目前正在研究和开发。

集群系统的使用和特点会随着存储区域网络(storage-area network，SAN)的流行而进一步扩大，关于 SAN，请参见 14.6.3 小节。SAN 允许多个主机与多个存储单元方便地连接。当前集群通常只有两个或四个主机，这是由连接主机到共享存储单元的复杂性所限制的。

1.7 实时系统

另一种形式的特殊用途操作系统为**实时系统**(real-time system)。当对处理器操作或数据流动有严格时间要求时,就需要使用实时系统;因此,它常用于控制特定应用的设备。传感器将数据送给计算机,计算机必须分析这些数据并可能调整控制,以修正传感器的输入。对科学实验、医学成像系统、工业控制系统和特定显示系统进行控制的系统为实时系统。有些汽车喷油系统、家电控制器和武器系统也属于实时系统。

实时系统有明确和固定的时间约束。处理必须在确定的时间约束内完成,否则系统会失败。例如,如果机器人臂在猛撞进所造的汽车之后才得到停止指令,那么这样就不行了。一个实时系统只有在其时间约束内返回正确结果才是正确工作。可将这一要求与分时系统(只是需要(而不是必须)响应快)或批处理系统(没有任何时间约束)相比较。

实时系统有硬的和软的两种类型。**硬实时系统**(hard real-time system)保证关键任务按时完成。这一目标要求对系统内所有延迟都有限制,包括从获取存储数据到要求操作系统完成任何操作的请求。这一时间约束要求决定了硬实时系统所具有的功能。通常只有少量或根本没有使用任何类型的辅助存储器,数据通常存在短期存储器或 ROM 中。ROM 位于非易失性存储设备上,即使掉电也会保持内容;绝大多数其他类型的内存为易失性的。硬实时系统也没有绝大多数高级操作系统的功能,这是因为这些功能常常将用户与硬件分开,导致难以估计操作所需时间。例如,实时系统是没有虚拟内存的(第十章)。因此,硬实时系统与分时操作系统的操作相矛盾,两者不能混合使用。由于没有一个现代通用操作系统支持硬实时功能,所以在本书中就不讨论这种类型的系统。

另一种限制较弱的实时系统是**软实时系统**(soft real-time system)。对于这类系统,关键实时任务的优先级要高于其他任务的优先级,且在完成之前能保持其高优先级。与硬实时系统一样,需要限制操作系统内核的延迟;实时任务不能无休止地等待内核来运行它。软实时是可以实现的目标,且可以与其他类型的系统相混合。但是,软实时系统比硬实时系统提供更多受限制的功能。由于没有安全时间界限的支持,它们在工业控制和机器人等领域的应用是危险的。但是,软实时系统可应用于一些其他领域,如多媒体、虚拟现实和高级科学研究项目(如深海探测和行星漫游)。这些系统需要高级操作系统功能,而这些功能又是硬实时系统不能支持的。由于软实时功能应用的扩展,正在寻求将其引入绝大多数当前操作系统(包括一些主流的 UNIX)的途径。

在第六章,研究实现操作系统的软实时功能的调度设备。在第十章,描述用于实时计算的存储管理设计。最后,在第二十一章,介绍 Windows 2000 操作系统的实时功能。

1.8 手持系统

手持系统(handheld system)包括**个人数字助理**(personal digital assistant,PDA),如Palm或可与网络如因特网相连的手机。手持系统和应用程序的开发人员面临着许多挑战,绝大多数是由于这些设备的有限尺寸。例如,PDA通常长约5英寸而宽约3英寸,重不到半磅。由于尺寸有限,绝大多数手持设备内存少,处理器速度慢,且屏幕小。下面简要研究一下这些限制。

许多手持设备只有512 KB到8 MB的内存。(而个人计算机或工作站有数百兆字节的内存!)因此,操作系统和应用程序必须有效地管理内存。这包括一旦已分配的内存不再使用就应返回给内存管理器。在第十章会研究虚拟内存,它允许开发人员编写程序时认为系统有比物理内存更多的可用内存。现在,许多手持设备都不使用虚拟内存技术,因此程序开发人员必须在有限物理内存的约束下工作。

手持设备开发人员关心的第二点是设备所使用处理器的速度。绝大多数手持设备处理器的速度通常只有个人计算机处理器速度的几分之一。更快的处理器需要更多电源。在手持设备中使用更快处理器需要使用更大电池,这会导致更加频繁地替换电池(或充电)。为了减小大多数手持设备尺寸,通常使用耗电更小、体积更小、速度更慢的处理器。因此,操作系统和应用程序的设计不能加重处理器的负担。

手持设备程序设计人员所面临的最后一个问题是可使用的显示屏幕较小。虽然家用计算机的显示器可达21英寸,但是手持设备的显示屏不到3平方英寸。常用的任务,如阅读电子邮件或浏览网页,必须在更小的显示器上进行。一种显示网页的方法是**网页剪辑**(web clipping),即在手持设备上只传送和显示一小部分网页。

有些手持设备可使用无线技术,如蓝牙(1.5节),允许远程访问电子邮件和浏览网页。与因特网相连的手机就属于这一类型。然而,许多PDA现在都不能提供无线访问。为了下载数据到这些设备,通常需要先将数据下载到个人计算机或工作站,接着再下载到PDA。有的PDA允许通过红外线链路在PDA之间直接实现数据拷贝。不过,PDA的这些功能限制被其方便性和可携带性所抵消。随着网络功能的增加和其他可选设备(如照相机和MP3播放机)不断扩展其应用,它们的使用会不断地增加。

1.9 功能迁移

总的来说,研究大型机和微型机的操作系统会发现许多大型机所具有的功能已为微型机所采用。同样的概念也适用于各种类型的计算机:大型机、小型机、微型机和手持机。图1.6所描述的许多概念将在本书后面介绍。然而,为了理解现代操作系统,你需要领会功能

迁移主题以及考虑许多操作系统功能的悠久历史。

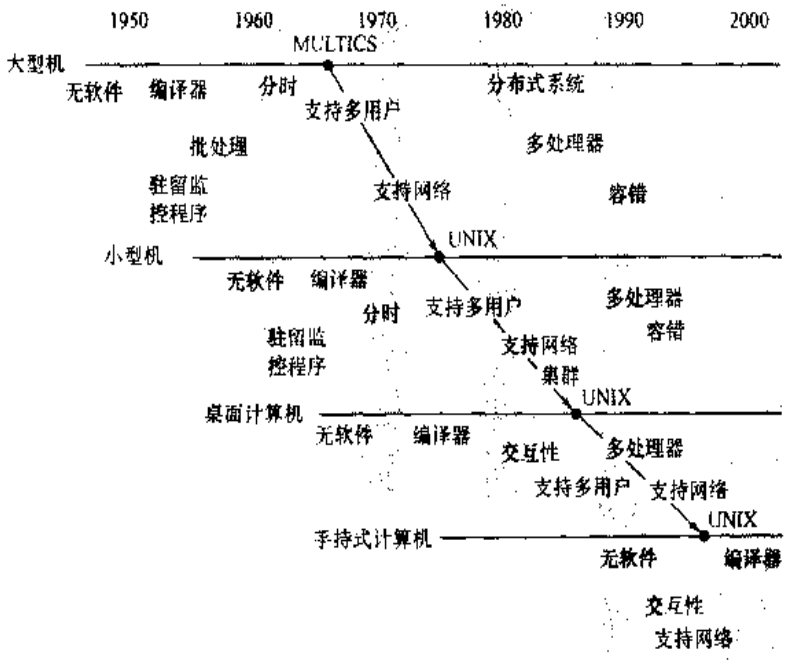


图 1.6 操作系统概念与功能的迁移

这种迁移的一个很好的例子是 MULTICS 操作系统。MULTICS 是在 1965 年到 1970 年期间在美国麻省理工学院(Massachusetts Institute of Technology,MIT)作为计算设施而开发的。它运行于大型复杂的主机上(GE 645)。许多为开发 MULTICS 的思想后来在贝尔实验室(原来 MULTICS 的合作开发者之一)应用于 UNIX 的设计。UNIX 操作系统设计于 1970 年,用于 PDP-11 小型机。在 1980 年左右,UNIX 的许多功能成为微机系统的类 UNIX 操作系统的基础,也为许多现代操作系统(如微软公司 Windows NT、IBM OS/2、Macintosh 操作系统)所吸收。因此,随着时间的推移,为大型机系统而开发的功能已向微型机迁移。

在许多大型操作系统功能不断用于个人计算机的同时,功能更强大、速度更快、更成熟的硬件系统也不断开发出来。**个人工作站**(personal workstation)是大型的个人计算机,如 Sun SPARCstation、HP/Apollo、IBM RS/6000 和支持 Windows NT 或 UNIX 的 Intel Pentium 级系统。许多大学和企业有大量的工作站连成局域网。随着个人计算机硬件和软件的日益成熟,大型机和微型机的分界线已变得模糊了。

1.10 计 算 环 境

在跟踪了操作系统发展过程之后(从无操作系统,到多道程序和分时系统,到个人计算机和手持计算机),下面简单地观察一下这些系统怎样在计算环境设置下使用。

1.10.1 传统计算

随着计算的不断发展成熟，许多传统计算环境的分界已变得模糊。现在考虑一下"典型办公环境"。几年前，这种环境由一些联网的个人计算机组成，服务器提供文件和打印服务，远程访问很不方便，移动功能是通过将笔记本计算机带到用户的一些工作场所来完成的。与主机相连的终端也已在许多公司流行，其远程访问能力和移动性就更差。

现代发展趋势是提供更多方法访问这些环境。网络技术正在扩展传统计算机的边界。公司实现了**入口**(portal)，以访问内部服务器。**网络计算机**(network computer)是可以理解基于网络计算的终端。手持计算机能与个人计算机同步，以允许对公司信息的可移动使用。这些设备可以联入**无线网络**，以使用公司的网络入口(和数不清的其他网络资源)。

在家里，绝大多数用户都有一台计算机通过慢速调制解调器与公司或因特网相连。网络连接曾经很昂贵，现在已经很便宜，这允许在公司或从网络更多地访问更多的数据。这些快速数据连接允许家庭计算机提供网页服务，有自己的网络(打印机、客户端个人计算机和服务器)。有的家庭还有**防火墙**(firewall)来保护家庭内部环境，以避免破坏。这些防火墙几年前价值数千美元，而十年前几乎不存在。

1.10.2 基于 Web 的计算

Web 几乎无处不在，能为多种设备所访问，这是数年前所难以想象的。个人计算机仍然是最为普通的访问设备，而工作站(高端面向图形的个人计算机)、手持 PDA 和手机等也能提供访问。

Web 计算增加了网络的重要性。过去不能联网的设备现在已能提供有线或无线访问。能联网的设备，通过改进网络技术或优化网络实现代码，现在已能提供更快的网络连接。

基于 Web 的计算的实现也导致了新一类设备的出现，如**负载平衡器**，它能在一组相似的服务器之间实现网络连接分配。操作系统如 Windows 95 过去作为 Web 客户机，现在也发展成为 Windows Me 和 Windows 2000，并可作为网络服务器和客户机。通常因为用户需要支持网络驱动，所以 Web 增加了设备的复杂性。

1.10.3 嵌入式计算

嵌入式计算机是现在最为普遍的计算机。它们运行嵌入式实时操作系统。这些设备无处不在，从汽车发动机和制造机器人到 VCR 和微波炉。它们有很特殊的任务。它们所运行的系统比较简单，没有高级功能(如虚拟内存和磁盘)。因此，操作系统只提供了有限的功能。它们通常只有少量或没有用户接口，它们将时间花在监视和管理硬件设备(如汽车发动机和机器人手臂)上。

比如以上所述的防火墙和负载平衡器，有的是通用计算机，运行标准操作系统(如

UNIX)和专门应用程序,以实现其功能。而其他的是具有内嵌专用操作系统的硬件设备,来提供所需要的功能。

嵌入式系统的使用不断增长。这些设备的功能,不管是作为独立单元还是网络或 Web 的组成部分,也肯定会增强。整个住宅可能计算机化,这样一台中央计算机(通用的或嵌入式系统)可以控制加热、照明、报警系统和煮咖啡。Web 访问可以允许户主在其到家之前告诉住宅加热。有一天,冰箱在发现牛奶用光时会通知食品商店送牛奶。

1.11 小 结

操作系统已经发展了 45 年,它有两个主要目的。第一,操作系统试图调度计算活动,以确保计算系统的高性能。第二,操作系统提供了一个环境,以便开发和运行程序。最初,计算机系统只能通过前端控制台来使用。汇编程序、装载程序、链接程序和编译程序这样一些软件改善了系统编程的方便性,但也增加了大量的设置时间。为了减少设置时间,设备雇用了操作员,并将类似作业分批处理。

批处理系统通过使用驻留操作系统允许自动切换作业,进而大大地提高了计算机的整体利用率。计算机不再需要等待人工操作。但是,CPU 的利用率仍然低,这是因为 I/O 设备的速度要比 CPU 的速度慢。慢设备的离线操作提供了一种方法,即在一个 CPU 上使用多个磁带读入器和磁带打印机系统。

为了改善计算机系统的整体性能,开发人员引入了多道程序设计的概念,这样多个作业可以同时位于内存中。CPU 通过在这些作业之间来回切换而增加了 CPU 利用率,也降低了执行作业所需要的总时间。

多道程序设计也允许分时。分时操作系统允许多个(从一个甚至到数百个)用户同时交互地使用一个计算机系统。

个人计算机是微型机;它们与大型机相比,相对比较小且便宜。这些计算机的操作系统在许多方面都得益于大型机操作系统的发展成果。不过,由于单个用户可以独用计算机,因而 CPU 利用率不再是主要问题。因此,有的大型操作系统的设计决策不再适用于这些小系统。其他设计决策,如安全性等,因为个人计算机可以通过网络或 Web 与其他计算机和用户相连,对于微型机和大型机都同样适用。

并行系统有多个紧密通信的 CPU;这些 CPU 共享计算机总线、有时也共享内存和外设。这些系统能提供高计算量和高可靠性。分布式系统允许对分布在各地的主机资源进行共享。集群系统允许多个机器对位于共享存储器的数据进行计算,即使一部分集群成员失败也能正常工作。

硬实时系统常常用于控制专用应用设备。硬实时操作系统具有明确的、固定的时间约束。处理必须在规定的约束内完成,否则系统失败。软实时系统没有严格的时间约束,不支

持最终期限调度。

近来，由于因特网和万维网的影响，现代操作系统的开发也集成了网络浏览器、网络和通信软件的特性。

本书也说明了操作系统开发的逻辑发展是如何受高级功能所需的CPU硬件中内含特性所驱动的。这个趋势可以从个人计算机的发展中看到，性能改善、价格便宜的硬件提供了更好的功能。

习 题 一

1.1 操作系统的两个主要目标是什么？

1.2 列举在一个完全专用机器上运行一个程序所需的四个步骤。

1.3 多道程序设计的主要优点是什么？

1.4 大型机的操作系统和个人计算机的操作系统的主要区别有哪些？

1.5 在多道程序设计和分时环境中，多个用户同时共享一个系统，这种情况会导致多种安全问题。

a. 列出两个此类的问题。

b. 在一个分时机器中，能否确保像在专用机器上一样的安全程度？并解释之。

1.6 列出下列类型操作系统的基本特点。

a. 批处理

b. 交互式

c. 分时

d. 实时

e. 网络

f. 并行式

g. 分布式

h. 集群式

i. 手持式

1.7 本书已经指出操作系统有效使用计算机硬件的必要性。什么时候操作系统可以违反这条原则并“浪费”资源而却又是合适的呢？为什么这样的系统不是真的浪费？

1.8 在什么情况下一个用户使用一个分时系统比使用一台个人计算机或单用户工作站更好？

1.9 描述对称多处理和非对称多处理之间的区别。多处理系统的三个优点和一个缺点是什么？

1.10 对于一个程序员来说，为一个实时环境编写操作系统所必须克服的主要困难是什么？

1.11 考虑操作系统的多种定义，操作系统是否应该包括像网络浏览器和电子邮件程序这样的应用程序？分别从正、反两方面加以论述，来支持你的答案。

1.12 手持计算机中固有的折中属性有哪些？

1.13 设想由两个运行数据库的节点构成的一个计算集群，说出集群软件管理磁盘数据访问的两种方法。论述每种方法的优点和缺点。

推荐读物

分时系统最先是由 Strachey[1959]提出的。最早的分时系统是美国麻省理工学院开发的 CTSS 和 System Development 公司(Schwartz 等[1964]、Schwartz 和 Weissman[1967])开发的 SDC Q-32 系统。其他早期的但更复杂的系统包括美国麻省理工学院(Corbato 和 Vyssotsky[1965])开发的 MULTICS 系统和美国加利福尼亚大学伯克利分校(Lichtenberger 和 Pirtle[1965])开发的 XDS-940 系统,还有 IBM 公司的 TSS/360 系统(Lett 和 Konigsford[1968])。

MS DOS 和个人计算机由 Norton[1988]和 Norton 与 Wilton[1988]加以描述。Apple 公司 Macintosh 硬件与软件的讨论可见 Apple[1987]。OS/2 操作系统在 Microsoft[1989]中加以讨论。更多的 OS/2 的信息可以在 Letwin[1988]、Deitel 和 Kogan[1992]中找到。Solomon 和 Russinovich[2000]论述了微软公司 Windows 2000 操作系统的结构。

Buyya[1999]提供了一个关于集群计算的很好的综述。Ahmed[2000]对近来集群计算的进展进行了讨论。

关于手持设备的论述由 Murray[1998]、Rhodes 和 McKeehan[1999]提供。

有许多关于操作系统的教科书,包括 Milenkovic[1987]、Finkel[1988]、Deitel[1990]、Stallings[2000b]、Nutt[1999]和 Tanenbaum[2001]。

第二章　计算机系统结构

在深入研究系统操作细节之前，需要对计算机系统的结构有一个全面的了解。在本章中，将会研究这一结构的若干不同部分，以复习背景知识。本章主要讨论计算机系统的体系结构，如果您已经理解这些概念，那么就可以浏览或跳过本章。本章首先讨论的问题包括系统启动、I/O 和存储器。

操作系统也必须保证计算机系统的正确运行。为了确保用户程序不干预系统的正常操作，硬件必须提供合适的机制，以确保正确的行为。在本章后面部分，将会描述基本的计算机体系结构，以便编写实用的操作系统。本章的最后概述了网络体系结构。

2.1　计算机系统操作

现代通用计算机系统由 CPU 和若干设备控制器通过共同的总线相连而成，该总线提供了对共享内存的访问(图 2.1)。每个设备控制器负责一种特定类型的设备(例如，磁盘驱动器、音频设备和视频显示器)。CPU 与设备控制器可并发工作，并竞争内存周期。为了确保对共享内存的有序访问，需要提供内存控制器来实现对内存的同步访问功能。

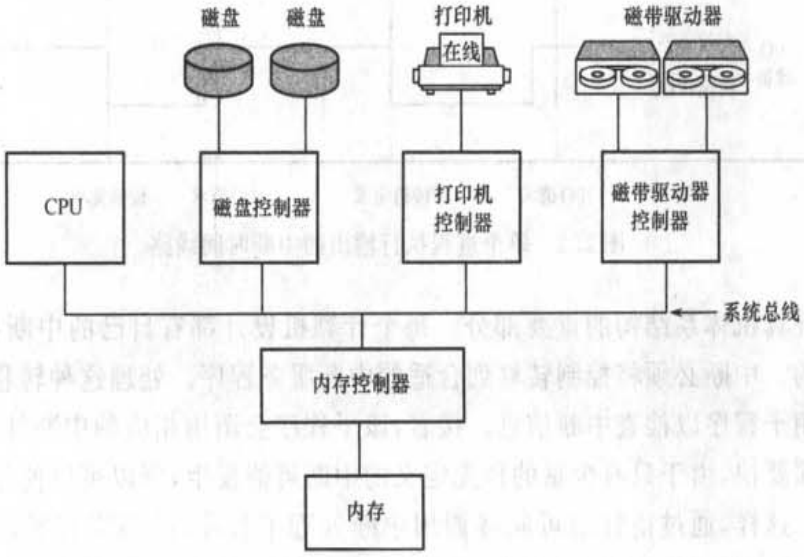


图 2.1　现代计算机系统

为了使计算机开始运行，例如当电源打开或计算机重启时，它需要运行一个初始化程

序。该初始化程序或**引导程序**(bootstrap program)比较简单。它通常位于只读存储器(ROM)中,如计算机硬件内的固件或 EEPROM。它初始化系统的所有部分,从 CPU 寄存器、设备控制器到内存内容。引导程序必须知道如何装入操作系统并开始执行系统。为了完成这一目标,引导程序必须定位操作系统内核并把它装入内存。接着,操作系统开始执行第一个进程如 init,并等待某些事件的发生。

事件的发生通常通过硬件或软件**中断**(interrupt)来表征。硬件可随时通过系统总线向 CPU 发出信号,以触发中断。软件通过执行一种称做**系统调用**(system call)(也称作**监控器调用**(monitor call))的特别操作也能触发中断。

现代操作系统是**中断驱动**(interrupt driven)的。如果没有进程可执行,没有 I/O 设备可服务,没有用户可响应,那么操作系统会安静地歇息并等待某种事件的发生。事件几乎总是通过发生中断或陷阱来表征的。**陷阱**(trap)(或**异常**(exception))是因错误(例如,除以 0 或非法访问内存)或用户程序(以执行操作系统服务)的特定请求所引起的软件生成中断。操作系统的中断驱动特性决定了系统的总体结构。对于每种类型的中断,操作系统都有一段独立的代码决定采取什么动作。中断服务子程序用来负责处理中断。

当 CPU 被中断时,它暂停正在做的事并立即将执行转到固定的位置去。该固定位置通常是中断服务程序开始位置的地址。中断服务程序开始执行,在执行完后,CPU 重新执行被中断的计算。这一操作的时间线路如图 2.2 所示。

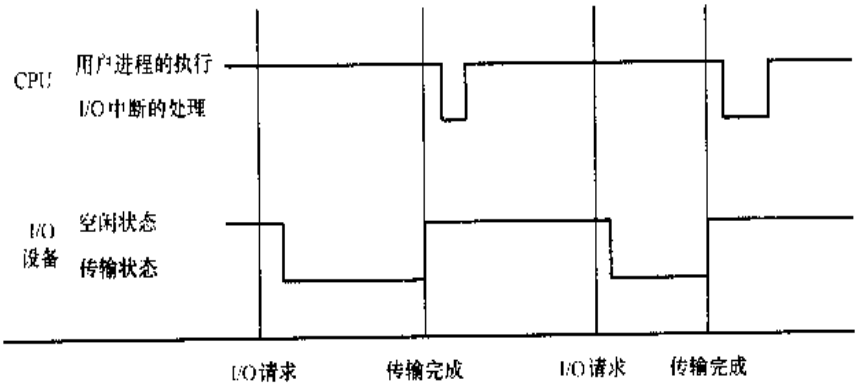


图 2.2 单个进程执行输出的中断时间线路

中断是计算机体系结构的重要部分。每个计算机设计都有自己的中断机制,但是有些功能是共同的。中断必须将控制转移到合适的中断服务程序。处理这种转移的简单方法是调用一个通用子程序以检查中断信息。接着,该子程序会调用相应的中断处理程序。不过,处理中断必须要快,由于只有少量的预先定义的中断可能发生,所以可以使用中断处理子程序的指针表。这样,通过指针表可间接调用中断处理子程序,而不需要通过其他中间子程序。通常,指针表位于低地址内存(头 100 左右的位置)。这些位置包含各种设备的中断服务子程序的地址。这种地址的阵列或**中断向量**(interrupt vector)可通过惟一设备号来索引(对于给定的中断请求),以提供当前中断设备的中断服务子程序的地址。许多操作系统如

MS-DOS 和 UNIX 都采用这种方式来处理中断。

中断体系结构也必须保存被中断指令的地址。许多旧的设计简单地在固定位置(或在可用设备号来索引的地址)中保存中断地址。更为现代的体系结构在系统堆栈中储存返回地址。如果中断处理程序需要修改处理器状态,如修改寄存器的值,它必须明确地保存当前状态并在返回之前恢复该状态。在中断服务之后,保存的返回地址会被装入程序计数器,被中断的计算可以重新开始,就好像中断没有发生过。

根据底层处理器所提供的功能性的不同,可以通过多种方法进行系统调用。不论采取何种形式,都是进程用来请求操作系统执行动作的方法。系统调用通常采用陷入到特定中断向量位置的形式。该陷阱可通过通用的 trap 指令来执行,而有些系统(如 MIPS R2000 族)通过特定 syscall 指令来执行。

2.2 I/O 结 构

如同在 2.1 节所讨论的,通用计算机系统由一个 CPU 和多个设备控制器所组成,它们通过共同的总线连接起来。每个设备控制器负责特定类型的设备。依控制器的不同,可有多个设备与其相连。例如,**小型计算机系统接口**(small computer system interface,SCSI)控制器可有七个或更多的设备与之相连。设备控制器维护一定量的本地缓冲存储器和一组特定用途的寄存器。设备控制器负责在其所控制的外围设备与本地缓冲存储器之间进行数据传递。设备控制器中本地缓冲区的大小取决于该控制器所控制的特定设备。例如,磁盘控制器缓冲区大小与磁盘最小可寻址区域(称为**扇区**(sector),通常为 512 B)的大小一样或是它的倍数。

2.2.1 I/O 中断

为了开始 I/O 操作,CPU 在设备控制器内装入适当的寄存器。相应地,设备控制器检查这些寄存器的内容,以决定采取何种操作。例如,如果控制器发现读请求,那么它开始从设备向其本地缓冲区传输数据。一旦数据传输完成,设备控制器就会通知 CPU 它已完成操作。它通过触发中断来实现这种通信。

这种情况通常是由于用户进程请求 I/O 而发生的。一旦 I/O 开始,就可能有两种行动过程。对于最简单的情况,开始进行 I/O;在 I/O 完成后,控制权返回给用户进程。这种情况称为**同步 I/O**(synchronous I/O)。另一种可能性,称为**异步 I/O**(asynchronous I/O),无需等待 I/O 完成,就将控制权返回给用户程序。接着 I/O 继续进行,同时其他系统操作照常进行(图 2.3)。

等待 I/O 完成可以采用两种方法之一来实现。有的计算机有一种特别指令 wait,以使 CPU 空闲直到下一个中断开始。没有这种指令的机器可能具备等待循环:

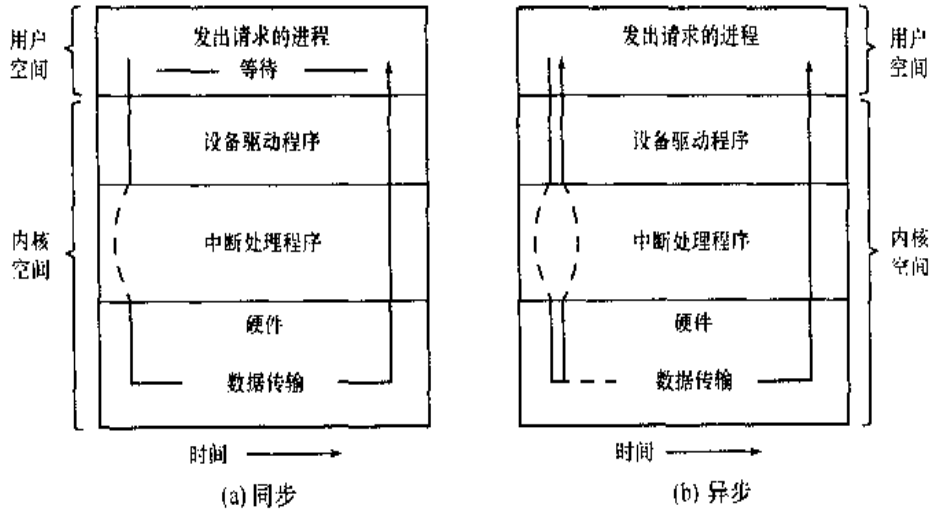


图 2.3 两种 I/O 方式

Loop: jmp *Loop*

这种严格的循环就一直继续，直到中断出现得以将控制权转交到操作系统的其他部分。这样的循环也可能需要轮询那些不支持中断结构的 I/O 设备；这些设备只是在其寄存器之一中设置一个标记，以期望操作系统发现这个标记。

如果 CPU 总是等待 I/O 完成，那么在任何时候最多只能处理一个 I/O 请求。因此，只要发生 I/O 中断，操作系统就肯定知道哪个设备产生了中断。但在另一方面，这种方法排除了多个设备的并发 I/O 操作，也排除了将有用计算与 I/O 相重叠的可能性。

一种更好的选择是开始 I/O 之后，就继续执行其他操作系统或用户程序代码。如果需要，系统调用就要允许用户程序等待 I/O 完成。如果没有用户程序就绪可执行，操作系统也没有其他工作可做，那么像以前一样仍然需要 wait 指令或空闲循环。也需要能够在同一时刻跟踪多个 I/O 请求。为此，操作系统采用了**设备状态表**(device status table)(图 2.4)，在状态表中每个 I/O 设备都有一个条目。每个表条目指明设备的类型、地址和状态(不工作，空闲或繁忙)。如果设备忙于处理一个请求，那么请求类型和其他参数都保存在该设备的表条目中。由于其他进程也可能向同一设备发出请求，所以操作系统也为每个 I/O 设备维持了一个等待队列，该队列就是一个等待请求列表。

当 I/O 设备需要服务时会产生中断。发生中断时，操作系统会首先确定哪个 I/O 设备引起中断。接着它会查找 I/O 设备表，以确定该设备状态，并修改表条目，以反映出现了中断。对于绝大多数设备，中断表示 I/O 请求的完成。如果队列中还有其他请求等待该设备，那么操作系统开始处理下一个请求。

最后，控制权会从 I/O 中断中返回。如果进程正在等待此请求的完成(如设备状态表中所记)，那么现在能将控制权返回到该进程。否则，返回到 I/O 中断前正在做的事：执行用户

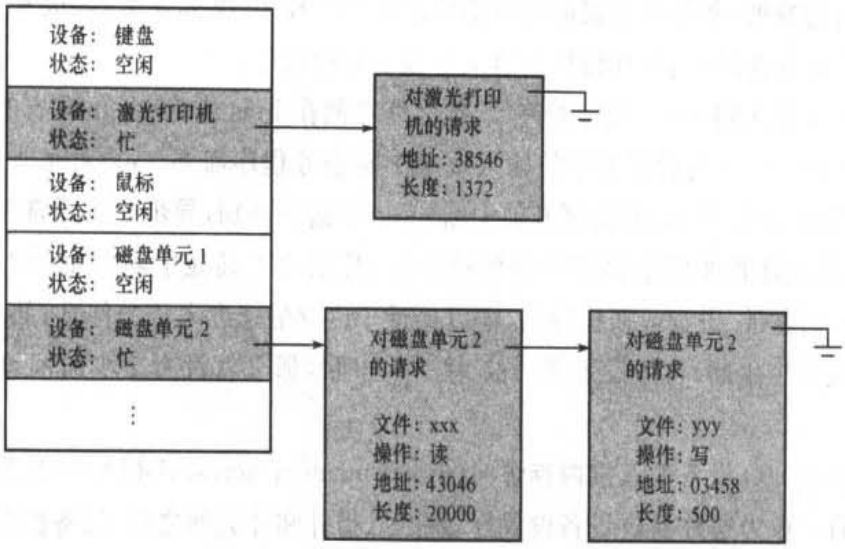


图 2.4 设备状态表

程序(程序开始了 I/O 操作且该操作已完成,但是程序并未等待该操作的完成)或循环执行等待(程序开始了两个或多个 I/O 操作,在等待其中一个完成,但是该中断是来自另一个操作)。对于分时系统,操作系统可切换到另一个就绪可执行的进程。

一个具体输入设备所使用的方法可能会跟上述方法有所不同。许多交互式系统都允许用户在键盘上提前打字:在程序请求数据之前输入数据。这种情况下会有中断产生,以表示字符从终端上到达,但是设备状态块指出没有程序对该设备有输入请求。如果允许提前打字,那么必须使用一块缓冲区以存储提前键入的字符,直到有程序需要它们为止。一般来说,对于每个输入设备都需要一个缓冲区。

异步 I/O 的主要优点是提高系统效率。当 I/O 发生时,系统 CPU 能用来处理或启动其他设备的 I/O。因为 I/O 相对于处理器速度来说执行得较慢,系统需要有效地使用这些设备。在 2.2.2 小节中,将描述另一种改善系统性能的机制。

2.2.2 DMA 结构

在一个简单的终端输入驱动程序中,要从终端读取一行时,所键入的第一个字符会发送给计算机。收到该字符时,与终端线相连的异步通信(或串口)设备会中断 CPU。当来自终端的中断请求到达时,CPU 正准备执行某些指令。(如果 CPU 正在执行某条指令,那么通常保留着中断以等待指令执行完成。)保存被中断指令的地址,将控制权转移到适当设备的中断服务程序。

中断服务程序会保存它所需要使用的所有 CPU 寄存器的内容。它检查所有可能来自最近输入操作的错误条件。接着它从设备处获取字符,并保存在缓冲区中。中断程序还必须调整指针和计数器变量,以确保下一个输入字符能保存在缓冲区的下一个位置。然后,中断程序在内存中设置一个标记,以通知操作系统的其他部分收到了新输入。其他部分负责

处理缓冲区内的数据，并将字符数据传送给请求输入的程序（见 2.5 节）。接着，中断服务程序恢复所保存寄存器的内容，并将控制返交给被中断的指令。

如果字符被键入到 9 600 波特的终端，那么终端能在 1 ms 或 1 000 μs 左右的时间内接受和传送一个字符。一个写得好的字符输入缓冲中断服务程序需要 2 μs 来处理一个字符，而剩下 998 μs 进行 CPU 计算（和处理其他中断）。由于这一差别，异步 I/O 通常是被赋予低优先级的中断，以允许其他更重要的中断先被处理，甚至为了其他中断而抢占当前中断。但是，高速设备，如磁带、磁盘或通信网络，能以接近于内存的速度来传送信息；如果 CPU 需要用 2 μs 来响应一个中断，再假设每隔 4 μs 有一个中断，那么就没有多少时间可用于执行进程。

可用于高速 I/O 设备的**直接内存访问**（direct memory access，DMA）就是为了解决这个问题而设计的。在为这种 I/O 设备设置好缓冲区、指针和计数器之后，设备控制器能在本地缓冲存储器和内存之间直接传送一整块数据，而无需 CPU 的干预。每块只产生了一个中断，而不是像低速设备那样每个字节（或字）产生一个中断。

CPU 的基本操作也一样。用户程序或操作系统本身可能需要请求数据传送。操作系统会从缓冲区池中找到一个缓冲区（一个空缓冲区用于输入，一个满缓冲区用于输出），以进行数据传输（一个缓冲区的大小与设备类型有关，通常为 128 B 到 4 096 B 不等。）接着，称为**设备驱动程序**的操作系统部分设置 DMA 控制器的寄存器，以便使用合适的源地址、目标地址和传送长度。接着，DMA 控制器被指令开始 I/O 操作。在 DMA 控制器执行数据传输时，CPU 空闲可执行其他任务。由于内存通常只能一次传输一个字，所以 DMA 控制器可从 CPU 中“窃取”内存周期。在进行 DMA 传输时，这种周期窃取会减慢 CPU 的运行。当传输已经完成时，DMA 控制器会中断 CPU。

2.3 存储结构

计算机程序必须在内存（又称**随机访问内存**（random-access memory）或 RAM）中，以便于运行。内存是处理器可直接访问的惟一的大容量存储区域（数兆到数千兆字节）。它是用称为**动态随机访问内存**（dynamic random-access memory，DRAM）的半导体技术来实现的，是由一组内存字的阵列组成。每个字都有其地址。通过对特定内存地址执行一系列 load 或 store 指令来实现交互。指令 load 能将内存中的字移到 CPU 的内部寄存器中，而指令 store 能将寄存器的内容移到内存。除了显式使用 load 和 store 外，CPU 可自动从内存中装入指令以执行。

一个典型指令执行周期（在冯·诺依曼（von Neumann）体系结构的系统上执行时），首先从内存中获取指令，并将此指令保存在**指令寄存器**（instruction register）中。接着，指令被解码，并可能导致从内存中获取操作数，并将操作数保存在某个内部寄存器中。在指令完成

对操作数的执行后，其结果可以存回到内存。注意内存单元只看见内存地址流；它并不知道它们是如何产生的（由指令计数器、索引、间接、常量地址等）或它们是什么地址（指令或数据）。相应地，可以忽视程序*如何*产生内存地址。本书只对运行程序所生成的内存地址序列感兴趣。

在理想情况下，人们需要程序和数据都永久地驻留在于以下两种原因，这是不可能的：

1. 内存太小，不能永久地存储所有需要的程序和数据。

2. 内存是*易失性*存储设备，当掉电或有其他原因时会失去所有内容。

因此，绝大多数计算机系统都提供**辅助存储器**（secondary storage）以作为内存的扩充。对辅助存储器的主要要求是它要能够永久性地存储大量的数据。

绝大多数常用辅助存储器设备为**磁盘**（magnetic disk），它能提供对程序和数据的存储。绝大多数程序（网页浏览器、编译器、字处理器、电子制表软件等）被保存在磁盘上，直到要执行时才装入内存。许多程序都使用磁盘作为它们所处理信息的源和目的地。因此，适当的磁盘存储管理对计算机系统来说十分重要，本书将在第十四章中加以讨论。

在广义上，本书所描述的存储结构由寄存器、内存和磁盘所组成，这仅仅是众多可能的存储系统中的一种。除此之外，还有高速缓存、CD-ROM、磁带等。每个存储系统都提供了基本功能，以存储数据或保存数据以便以后提取。各种存储系统的主要差别在于速度、价格、大小和易失性。在2.3.1小节到2.3.3小节中，将描述内存、磁盘和磁带，因为它们代表了所有重要的商业化可用存储设备的通常属性。在第十四章中，将讨论许多特定设备如软盘、硬盘、CD-ROM和DVD的特殊属性。

2.3.1 内存

内存和处理器本身内置寄存器是CPU能直接访问的惟一存储介质。有些机器指令能将内存地址作为参数，但是没有指令能直接使用磁盘地址。因此，正在执行的任何指令和指令所能使用的任何数据必须在这些能直接访问的存储设备中。如果数据不在内存中，那么在CPU能操作它们之前必须先将其移到内存中。

对于I/O，如2.1节所述，每个I/O控制器都有寄存器来保存命令和所要传输的数据。通常，特殊I/O指令允许在这些寄存器和系统内存之间进行数据传输。为了允许对这些I/O设备进行更方便的访问，许多计算机体系结构都提供**内存映射I/O**（memory-mapped I/O）。在这种情况下，内存地址的一块范围被单独分开，并映射到设备寄存器。通过读、写这些内存地址就能实现与设备寄存器之间的数据传输。这种方法对于那些要求快速响应时间的设备（如视频控制器）比较合适。对于IBM个人计算机，每个屏幕位置都被映射到一个内存位置。在屏幕上显示文本几乎如同向适当的内存映射地址写文本一样简单。

内存映射I/O对于其他设备（如连接调制解调器和打印机到计算机的串行端口或并行端口）也很方便。CPU通过对称为I/O端口的设备寄存器进行读和写，与这些设备进行数

据的传输。为了通过内存映射的串行端口发送一长串字节，CPU 会将一个数据字节写入数据寄存器，并设置控制寄存器的相应位，以通知有字节可用了。设备会提取数据字节并清除控制寄存器的相应位，以表示就绪可接收下一个字节。接着，CPU 能够传输下一个字节。如果 CPU 使用轮询方式关注控制位，不断地循环检测设备是否就绪，这种操作方式称为**程序化 I/O**(programmed I/O，PIO)。如果 CPU 不是轮询控制位，而是在设备为下个字节就绪时收到一个中断，那么这种数据传递输为**中断驱动**(interrupt driven)的。

CPU 内置寄存器通常在一个 CPU 时钟周期内可被访问。绝大多数 CPU 能以在一个时钟周期内执行一个或多个操作指令的速度来解码指令并执行有关寄存器内容的简单操作。而内存就不能这样，需要通过内存总线的一个事务来访问。内存访问可能需要多个 CPU 时钟周期来完成，在这种情况下，因为得不到所需要的数据以完成正在执行的指令，处理器通常需要**延迟**。由于经常要访问内存，这种情况是无法忍受的。补救方法是在 CPU 与内存之间增加快速内存。用于解决速度差异的内存缓冲器，称为**高速缓存**(cache)，将在 2.4.1 小节中加以描述。

2.3.2 磁盘

磁盘(magnetic disk)为现代计算机系统提供大容量的外存。从概念上来说，磁盘相对简单(图 2.5)。每个磁盘**片**为扁平圆盘，如同 CD 一样。常用磁盘片的直径从 1.8 英寸到 5.25 英寸不等。每个磁盘片的两面都涂有磁质材料。可以通过在磁片记录来保存信息。

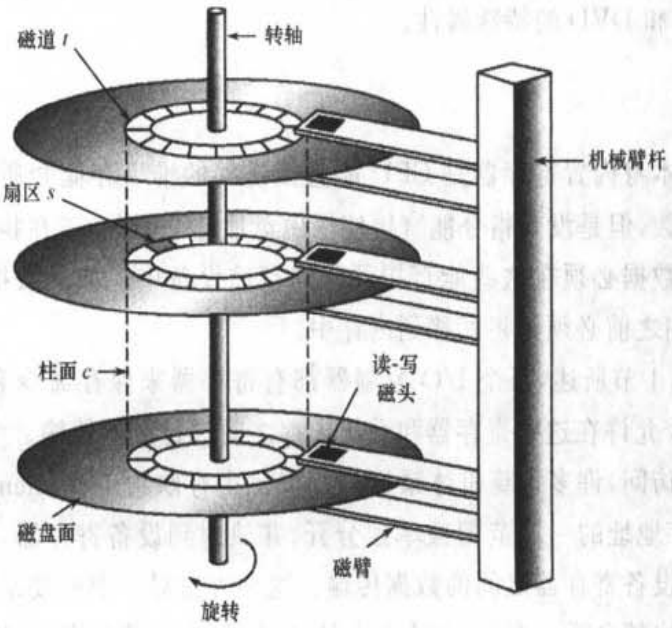


图 2.5 移动磁头的磁盘装置

读－写磁头“飞行”于每个磁盘片的表面之上。磁头与**磁臂**(disk arm)相连，磁臂能将所有磁头作为一个整体而一起移动。磁盘片的表面被逻辑地划分成圆形**磁道**(track)，再进一

步划分为**扇区**(sector)。位于同一磁臂位置的磁道集合形成了**柱面**(cylinder)。每个磁盘驱动器有数千个同心柱面,每个磁道可能包括数百个扇区。常用磁盘驱动器的存储容量是按GB来衡量的。

当磁盘在使用时,驱动器马达会高速旋转磁盘。大多数驱动器每秒可转动60到200圈。磁盘速度有两部分。**传输速率**(transfer rate)是在驱动器和计算机之间的数据传输速率。**定位时间**(positioning time),有时称为**随机访问时间**(random-access time),由**寻道时间**(seek time)(以移动磁臂到所要的柱面)和**旋转等待时间**(rotational latency)(以等待所要的扇区旋转到磁头下)组成。典型的磁盘每秒能传输数兆字节,寻道时间和旋转等待时间为数毫秒。

由于磁头飞行于极薄(数微米)的空气层上,所以磁头有与磁盘表面相接触的危险。虽然磁盘片上涂了一层薄的保护层,但是磁头还是可能损坏磁盘表面。这种现象称为**磁头碰撞**(head crash)。磁头碰撞通常不能修复;整个磁盘必须更换。

磁盘可以**移动**或**撤换**,以便允许根据需要来装入不同的磁盘。可移动磁盘通常由一个磁盘片组成,它保存在塑料盒内,以防止不在磁盘驱动器内时被损坏。**软盘**(floppy disk)是较为便宜的可移动磁盘,它有一个软塑料盒,以保存柔软的磁盘片。软盘驱动器的磁头通常直接与磁盘表面相接触,所以与硬盘驱动器相比,该驱动器所设定的旋转速度较慢,以减少对磁盘表面的损耗。软磁盘的存储容量通常为1 MB左右。还有可移动磁盘与普通硬盘一样工作,其容量是按GB来衡量的。

磁盘驱动器通过一组称为**I/O总线**(I/O bus)的线与计算机相连。有多种可用总线,包括**增强型集成驱动器电子电路**(enhanced integrated drive electronics,EIDE)、**高级技术附件**(advanced technology attachment, ATA)和小型计算机系统接口总线。称为**控制器**(controller)的特殊电子处理器执行总线上的数据传输。**主机控制器**(host controller)是总线上位于计算机端的控制器。**磁盘控制器**(disk controller)位于磁盘驱动器内。为了执行磁盘I/O操作,计算机常常通过内存映射I/O端口(如2.3.1小节所述)在主机控制器上放置一个命令。主机控制器接着通过消息将该命令传送给磁盘控制器,磁盘控制器操纵磁盘驱动器上的硬件以执行命令。磁盘控制器通常有内置缓存。磁盘驱动器的数据传输发生在其缓存和磁盘表面之间,而到主机的数据传输则以更快的电子速度在其缓存和主机控制器之间进行。

2.3.3 磁带

磁带(magnetic tape)曾经是早期的外存媒介。虽然它相对较长久,且能存储大量数据,但是与内存相比其访问速度太慢。另外,磁带随机访问要比磁盘随机访问慢千倍,因此磁带对于作外存而言用途不大。磁带主要用于备份,存储不经常使用的信息,作为系统之间信息传输的一种媒介。

磁带绕在轴上，向前转或向后转并经过读－写磁头。移到磁带的正确位置需要数分钟，但是一旦定位后，磁带驱动器就能以跟磁盘驱动器相似的速度写数据。磁带容量变化很大，它取决于特定磁带驱动器类型。有的磁带能比大磁盘驱动器多存储 2 到 3 倍数据。磁带及其驱动程序通常按其宽度来划分，有 4 mm、8 mm、19 mm、1/4 英寸和 1/2 英寸。

2.4 存储层次

根据速度和价格，可以按层次结构来组织计算机系统的不同类型的存储系统(图 2.6)。层次越高，价格越贵，但是速度越快。随着向层次下面的移动，单个位的价格通常降低，而访问时间通常增加。这种折中是合理的；如果一个给定的存储系统比另一个更快且更便宜，而其他属性一样，那么就没有理由使用更慢且更昂贵的存储系统。事实上，许多早期的存储设备，包括纸带和磁心存储器，之所以现在已经送进博物馆，就是因为磁带和**半导体内存**已变得更快且更便宜。图 2.6 中的上三层存储器可以采用半导体内存技术来构建。

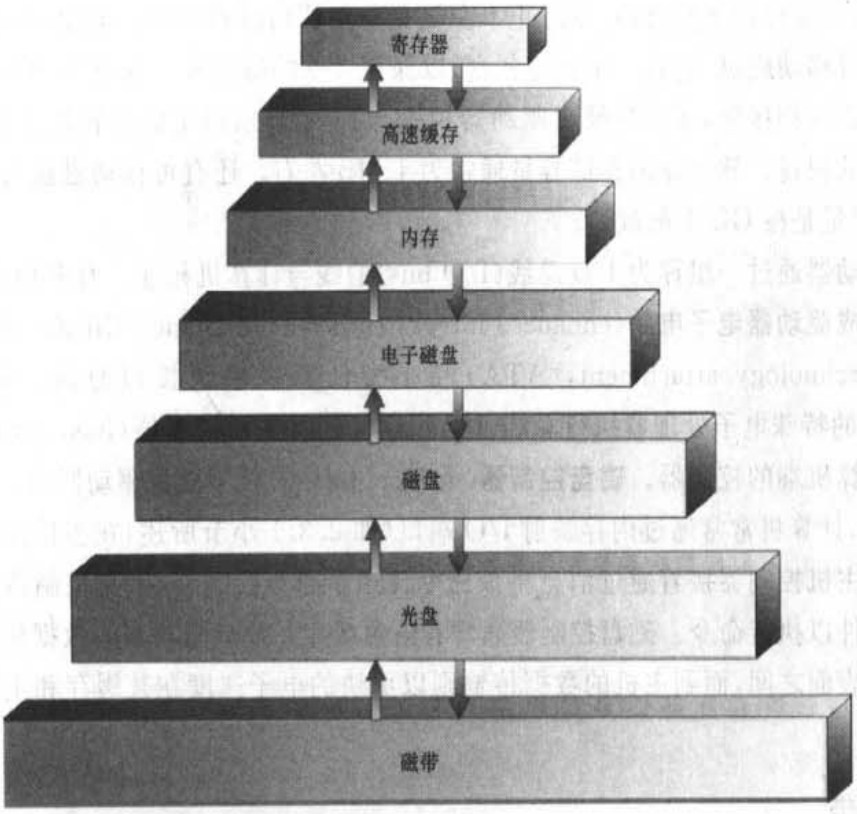


图 2.6 存储设备层次

除了具有不同的速度和价格，各种存储系统还分为易失的和非易失的。当设备断开电源时，**易失存储器**(volatile storage)会丢失其中的内容。如果没有昂贵的电池和发电机后备系统，那么数据必须写到**非易失存储器**(nonvolatile storage)中，以便保存。在图 2.6 所示的

层次结构中，电子磁盘之上的存储系统为易失的，而其下的存储系统为非易失的。**电子磁盘**(electronic disk)可以被设计为易失的或非易失的。在普通操作情况下，电子磁盘将数据保存在一个大的DRAM阵列上，这是易失的。但是，许多电子磁盘设备都有一个隐藏的硬磁盘和电池作为备份电源。当外部电源发生中断时，电子磁盘控制器将数据从RAM复制到磁盘。当外部电源恢复时，控制器将数据复制回RAM中。

一个完整存储系统的设计必须平衡所有这些因素：它只使用必需的昂贵存储器，而提供尽可能多的便宜的、非易失的存储器。对于两个部件间存在较大访问时间或传输速率差别的情况，可通过安装高速缓存来改善性能。

2.4.1 高速缓存技术

高速缓存是计算机系统的重要理论之一。信息通常保存在一个存储系统(如内存)中。当使用它时，它会被临时地复制到更快的存储系统(高速缓存)中。当人们需要特定信息时，首先检查它是否在高速缓存中。如果是，可直接使用高速缓存中的信息；如果否，就使用位于内存系统中的信息，同时将其复制到高速缓存中，以便在不久的将来需要时再次使用。

另外，内部可编程寄存器，如索引寄存器，为内存提供了高速缓存。程序员(或编译程序)实现了寄存器分配和寄存器替换算法，以决定哪些信息应在寄存器中而哪些应在内存中。有的高速缓存完全是由硬件实现的。例如，绝大多数系统都有指令高速缓存，以保存下一个要执行的指令。没有这一高速缓存，CPU将会等待多个时钟周期以便从内存中获取指令。基于类似原因，绝大多数系统在其存储层次结构中有一个或多个高速数据缓存。在本书中不关心纯硬件高速缓存，因为它们不受操作系统所控制。

因为高速缓存大小有限，所以**高速缓存管理**(cache management)的设计工作很重要。对高速缓存大小和置换策略的仔细选择可以使所有访问内容的80%到99%都在高速缓存中，从而极大地提高了性能。关于软件所控制的高速缓存的各种置换算法将在第十章中加以讨论。

内存可看做外存(辅存)的高速缓存，因为外存数据必须先复制到内存才可使用，数据必须已在内存中才可确保移动到外存。文件系统数据永久地驻留在外存上，可以出现在存储层次的许多级别上。在最高层，操作系统可在内存中保存一个文件系统数据的高速缓存。而且，电子RAM磁盘(也称做**固体磁盘**(solid-state disk))也可用做通过文件系统接口访问的高速存储器。外存以磁盘为主。磁盘存储器又可以用磁带或可移动磁盘来备份数据以便不受硬盘损坏导致的数据丢失所影响。有的系统自动将位于外存上的旧文件的数据备份到第三级存储器如磁带塔上，以降低存储费用(参见第十四章)。

存储层次之间的信息移动可以是显式的，也可以是隐式的，这取决于硬件设计和所控制的操作系统软件。例如，高速缓存到CPU和寄存器之间的数据传输通常为硬件功能，无需操作系统干预。另一方面，磁盘到内存的数据传输通常是由操作系统控制的。

2.4.2 一致性与连贯性

对于层次存储结构，同样的数据可能出现在不同层次的存储系统上。例如，假设整数 A 位于文件 B 中且需要加 1，而文件 B 位于磁盘上。加 1 操作这样进行：先发出 I/O 操作，以将 A 所在的盘块拷贝到内存。之后，A 被复制到高速缓存和内部寄存器。这样，A 的拷贝出现在多个地方：磁盘上，内存中，高速缓存中，内部寄存器中(图 2.7)。一旦加法在内部寄存器中执行后，A 的值在不同存储系统中就会不同。只有在 A 的新值从内部寄存器写回磁盘后，A 的值才会变得一样。

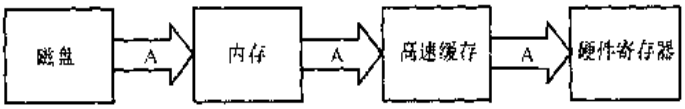


图 2.7 整数 A 从磁盘到寄存器的迁移

对于一次只有一个进程执行的计算环境，这种安排没有困难，因为对整数 A 的访问总是在层次结构的最高层拷贝进行。不过，对于多任务环境，CPU 会在诸多进程之间来回切换，所以需要十分谨慎，以确保：如果多个进程要访问 A 时，那么每个进程都会得到最近已更新的 A 值。

对于多处理器环境，这种情况变得更为复杂，因为每个 CPU 不但要维护自己的内部寄存器，还要有本地高速缓存。对于这种环境，A 的拷贝会同时出现在多个高速缓存中。由于多个 CPU 可并发执行，必须确保在一个高速缓存中对 A 的值所做更新立即反映在所有其他 A 所在的高速缓存中。这种情况称为**高速缓存一致性**(cache coherency)，这通常是硬件问题(在操作系统级别之下处理)。

对于分布式环境，这种情况变得更为复杂。在这种情况下，同一文件的多个拷贝(或复制)会保留在多个分布在不同场所的不同计算机上。由于各个复制可能会被并发访问和更新，所以必须确保当一处的复制被更新时，所有其他复制应尽可能快地加以更新。如第十六章所述，有许多方法可达到这种保证条件。

2.5 硬件保护

早期计算机系统是由程序员操作的单用户系统。当程序员通过终端操作计算机时，可完全控制系统。但是，随着操作系统的发展，这种控制交给了操作系统。早期操作系统称为**常驻监督程序**(resident monitor)。从常驻监督程序开始，操作系统开始执行许多功能，尤其是 I/O，这些过去是由程序员负责的。

另外，为了改善系统使用率，操作系统开始在多个程序之间同时共享系统资源。通过采用假脱机，一个程序能与其他进程的 I/O 并发执行；磁盘能同时为多个进程保存数据。多

道程序设计能同时将多个程序放在内存中。

共享改善了利用率，但与此同时也增加了问题。当系统在没有共享的情况下运行时，程序中的一个错误只会对正在运行的程序引发问题。采用了共享，一个程序中的错误有可能恶意地影响多个进程。

例如，考虑一下简单的批处理操作系统(1.2.1小节)，它只提供了自动作业排序。如果一个程序死循环而不断地读取输入卡片，那么该程序会读取所有数据。除非有事情阻止，它将不断地读入卡片，这些卡片来自下一个作业、再下一个作业，等等。该循环可能阻止了许多作业的正确操作。

对于多道程序设计系统，可能会出现更多的微妙的错误，一个错误程序可能修改程序或另一个程序的数据，甚至常驻监督程序本身。MS-DOS 和 Macintosh OS 都允许这种错误。

如果没有保护来处理这些错误，那么计算机必须只能一次执行一个进程，否则所有输出都值得怀疑。操作系统的合理设计必须确保错误程序(或恶意程序)不会造成其他程序错误执行。

硬件可检测到很多编程错误。这些错误通常由操作系统处理。如果一个用户程序出现某种失败，如试图执行非法指令或者访问不属于自己地址空间的内存，那么硬件会陷入到操作系统。陷阱如同中断一样，能将中断向量的控制转交给操作系统。只要一个程序出现错误，操作系统就必须对这个程序进行异常终止。这种情况的处理代码与用户请求的异常终止的处理代码一样，会给出一个适当的出错信息，程序内存会被转储。内存转储通常被写到文件中，以便用户或程序员能检查它，或许能纠正错误，并重新启动程序。

2.5.1 双重模式操作

为了确保操作正常，必须保护操作系统和所有其他程序及数据使之不受任何故障程序的影响。所有共享资源都需要保护。许多操作系统所采取的方法提供硬件支持，以允许用户区分各种执行模式。

至少需要两重独立的操作**模式：用户模式**(user mode)和**监督程序模式**(monitor mode)(也称为**管理模式**(supervisor mode)、**系统模式**(system mode)或**特权模式**(privileged mode))。一个称为**模式位**(mode bit)的位，增加到计算机硬件，以表示当前模式：监督程序模式(0)或用户模式(1)。有了模式位，可区分为操作系统所执行的任务和为用户所执行的任务。正如大家将要看到的，这种体系结构的改进对于系统操作的其他诸多特性都很有用。

在系统引导时，硬件开始处于监督程序模式。接着，装入操作系统，开始在用户模式下执行用户进程。一旦出现陷阱或中断，硬件会从用户模式切换到监督程序模式(即将模式位状态变为0)。因此，只要操作系统获得了对计算机的控制，它就处于监督程序模式。系统在将控制交权还给用户程序之前总会切换到用户模式(将模式位设置为1)。

双重操作模式为人们提供了保护操作系统和用户程序不受错误用户程序影响的手段。

可以这样实现保护操作：将能引起损害的机器指令作为**特权指令**(privileged instruction)。硬件允许仅在监督程序模式下执行特权指令。如果在用户模式下试图执行特权指令，那么硬件并不执行该指令，而是认为该指令非法，并将其陷入到操作系统。

特权指令概念也为人们提供了一种方法，以便用户能与操作系统进行交互，从而请求系统执行一些只有操作系统才能做的指定任务。每个这样的请求都是由用户调用用来执行特权指令的。这种请求称为系统调用(也称为监督程序调用或操作系统函数调用)，如 2.1 节所述。

当系统调用被执行时，硬件会将它视为软件中断。控制权会从中断向量转交到操作系统的中断服务程序，模式位设置成监督程序模式。系统调用服务程序是操作系统的一部分。监督程序检查中断指令以确定发生了何种系统调用；参数表示用户程序请求什么类型的服务。请求所需要的其他信息可通过寄存器、堆栈或内存(内存地址指针可通过寄存器传送)来传递。监督程序检验参数是否正确和合法，执行请求，然后将控制权返回到系统调用之后的指令。

没有硬件支持的双重模式会在操作系统内产生一些严重缺点。例如，MS-DOS 是为 Intel 8088 体系结构而编写的，它没有模式位，因而没有双重模式。运行错误的用户程序可通过写数据覆盖而清除整个操作系统，多个程序可同时对设备进行写操作，但可能引起灾难性的结果。更为现代、高级的 Intel CPU 版本，如 Pentium，确实提供双重模式操作。因此，更多现代操作系统，如微软公司 Windows 2000 和 IBM OS/2，都利用了这一特征，并为操作系统提供了更强大的保护。

2.5.2 I/O 保护

通过发出非法 I/O 指令、访问操作系统的内存位置或拒绝释放 CPU，用户程序可打断系统的正常操作。可以使用各种机制以确保这种打断不会在系统中发生。

为了防止用户执行非法 I/O，可定义所有 I/O 指令为特权指令。因此，用户不能直接发出 I/O 指令；他们必须通过操作系统来进行。为完全保护 I/O，必须确保用户程序不能在监督程序模式下控制计算机。如果能这样，那么 I/O 操作的保护会打折扣。

考虑一个计算机在用户模式下执行。当出现中断或陷阱时，它会切换到监督程序模式，转到由中断向量决定的地址。如果用户程序在执行时能在中断向量中存入一个新的用户程序的地址，那么新地址就可以改写以前的地址。接着，在相应陷阱或中断出现时，硬件会切换到监督程序模式，并通过(已修改的)中断向量将控制权转到用户程序！用户程序就可以在监督程序模式下获得对计算机的控制。事实上，用户程序有许多其他方法能获得在监督程序模式下对计算机的控制。另外，每天都会发现新错误，这些错误可用来绕过系统保护。这些论题将在第十八章和第十九章中加以讨论。因此，为了执行 I/O，用户程序执行系统调用，以请求操作系统为其执行 I/O(图 2.8)。操作系统在监督程序模式下执行，检查请求是

否合法，并(如果请求合法)执行所请求的 I/O。接着，操作系统返回到用户(程序)。

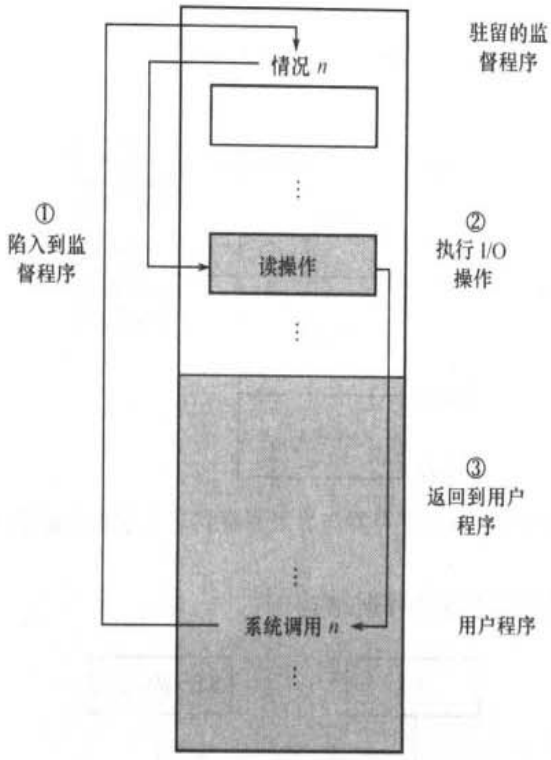


图 2.8 使用系统调用执行 I/O

2.5.3 内存保护

为了确保正确操作，必须保护中断向量以免受用户程序的修改。另外，还必须保护操作系统的中断服务程序以免被修改。即使用户不能获取对计算机的未经授权的控制，对中断服务程序的修改也能中断计算机系统的正常操作、假脱机和缓冲操作。

因此，可以看到至少必须提供对操作系统的中断向量和中断服务程序的内存保护。通常，需要保护操作系统不为用户程序所访问，并保护用户程序不为彼此所访问。这种保护必须由硬件提供。如第九章所述，有许多实现方法。这里只简述一种可能的实现。

为了区分程序的内存空间，需要能确定程序所能访问的合法地址空间，并保护其他内存空间。通过使用两个寄存器(如图 2.9 所示)即**基址寄存器**(base register)和**界限寄存器**(limit register)，可提供这种保护。基址寄存器保存着最小的合法物理内存地址；界限寄存器包含范围的大小。例如，如果基址寄存器和界限寄存器的值分别为 300040 和 120900，那么，程序所能合法访问的地址范围为从 300040 到 420940。

这种保护是由 CPU 硬件完成的：它将用户模式下所生成的每个地址与这些寄存器相比较。在用户模式下运行的程序若试图访问监督程序内存或其他用户的内存，会陷入到监督程序，这种尝试被认为是致命错误(图 2.10)。这种方法防止用户程序(无意地或有意地)

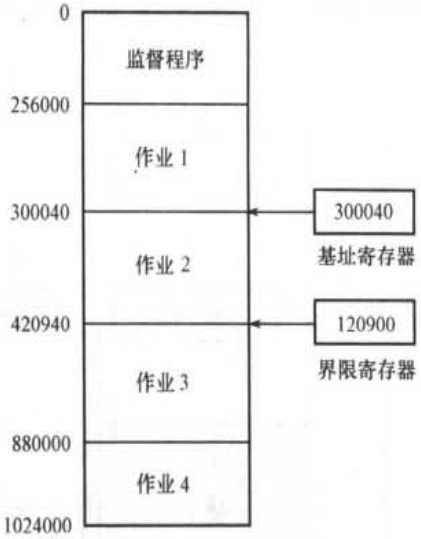


图 2.9 基址寄存器与界限寄存器定义逻辑地址空间

修改操作系统或其他用户的代码或数据结构。

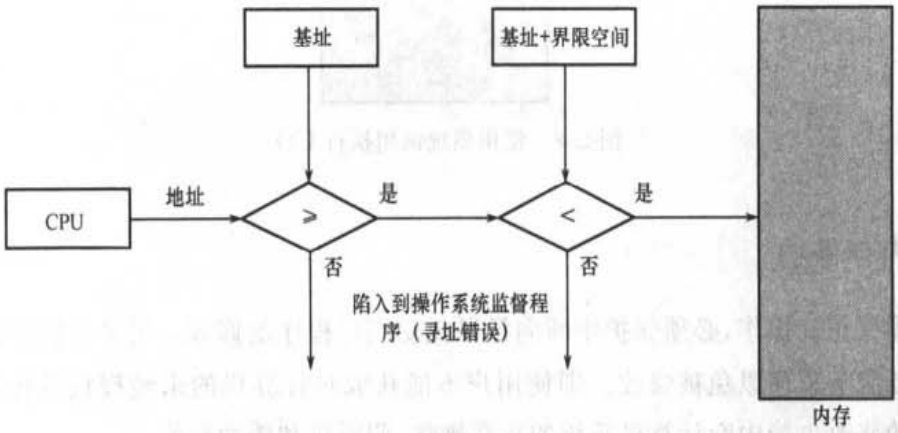


图 2.10 通过基址寄存器与界限寄存器的硬件地址保护

基址寄存器和界限寄存器只能由操作系统来装入，这使用了特殊特权指令。由于特权指令只能在监督程序模式下执行，且只有操作系统才能在监督程序模式下执行，所以只有操作系统才能装入基址寄存器和界限寄存器。这种方法允许监督程序修改这些寄存器的值，但是防止用户程序修改这些寄存器的内容。

操作系统若运行在监督程序模式下，能对监督程序和用户的内存进行无限制的访问。这一条件允许操作系统将用户程序装入用户内存，在出现错误时转储这些程序，访问和修改系统调用的参数，等等。

2.5.4 CPU 保护

除了保护 I/O 和内存外，必须确保操作系统能维持控制。必须防止用户程序陷入死循

环或不调用系统服务，并且不将控制权返回到操作系统。为了实现这一目标，可使用**定时器**(timer)。定时器能设置成在给定时间后中断计算机。时间段可以是固定的(例如，1/60 s)或可变的(例如，1 ms 到 1 s)。**可变定时器**(variable timer)一般通过一个固定速率的时钟和计数器来实现。操作系统设置计数器。对每次时钟周期，计数器都要递减。当计数器的值为 0 时，产生中断。例如，对于 10 bit 的计数器和 1 ms 精度的时钟，可允许从 1 ms 到 1 024 ms的时间间隔内产生中断，时间步长为 1 ms。

操作系统在将控制权交给用户之前，应确保设置好定时器，以便产生中断。如果定时器产生中断，那么控制权会自动交给操作系统，而操作系统可以将中断作为致命错误来处理，也可以给予程序更多的时间。显然，用于修改定时器操作的指令是特权的。

因此，可使用定时器来防止用户程序运行时间过长。一种简单技术是用程序所允许执行的时间量来初始化计数器。例如，能运行 7 min 时间限度的程序可以将计数器设置为 420。定时器每秒钟产生一次中断，计数器相应减 1。只要计数器的值为正，控制就返回到用户程序。当计数器的值为负时，操作系统会中止程序执行，因为它超过了所赋予的时间限制。

定时器更为常用的用途是实现分时系统。最为直接的方法是，将定时器设置为每 N ms 产生一次中断，N 为每个用户程序在下一个用户获得 CPU 控制权之前所允许执行的**时间片**(time slice)。在每个时间片结束时操作系统会被唤起执行各种维护任务，如增加 N 到记录中，以(为了计数目的)表示用户程序迄今为止所执行的时间量。而且操作系统保存寄存器内容、内部变量和缓冲区，并改变其他一些参数，以为下个程序的运行作准备。这一过程称为**关联切换**(context switch)，将在第四章中加以讨论。在关联切换之后，下一个程序从原来停止的位置(在上一次时间片结束时)继续其执行。

定时器的另一个用途是计算当前时间。定时器中断表示一定的时间段已过，这允许操作系统根据某初始时间来计算当前时间。如果每秒产生一次中断，从 1：00 PM 后共有 1 427个中断，那么就可以计算出当前时间为 1：23：47 PM。有的计算机通过这种方式确定当前时间，但是这种计算要仔细进行，以保证准确地记录时间，这是因为中断处理时间(和其他中断未能执行的时间)倾向于引起软件时钟变慢。绝大多数计算机都有一个独立的硬件计时时钟，它独立于操作系统。

2.6 网络结构

有两种基本类型的网络：**局域网**(local-area network，LAN)和**广域网**(wide-area network，WAN)。两者的主要差别在于它们的地理分布方式不一样。局域网由分布在较小地理区域内(如一栋大楼或若干相邻的大楼)的多个处理器组成。而广域网由分布在较大地理区域内(如美国)的一些自治的处理器组成。这些差别意味着通信网络在速度和可靠性方

面的显著差别，它们也反映在分布式操作系统的设计之中。

2.6.1 局域网

局域网，作为大型计算机系统的替代品，出现于20世纪70年代的初期。对于许多企业而言，使用一些小型计算机（每个都有自包容的应用程序）比使用一个大型系统更为经济。由于每台小型计算机可能需要全部的外设（如磁盘和打印机），而且一个企业内部也可能需要一定形式的数据共享，所以很自然地将这些小型系统组成一个网络。

LAN通常设计成为覆盖一个小的地理区域（如一栋大楼或若干相邻的大楼），并通常在办公环境中使用。这类系统的所有站点相对较近，因此与广域网的对应部分相比，其通信链路倾向于具有更高的速度和更低的出错率。需要高质量的（昂贵的）电缆来获得更高的速度和可靠性。在数据网络通信中，仅使用电缆也是可能的。在较长距离内，使用高质量的电缆的代价是巨大的，仅使用电缆也是不允许的。

最为普通的局域网连接是双绞线和光纤电缆。最普通的组成结构是多点访问总线、环形和星形网络。通信速度从1 Mb/s（bit per second）（如AppleTalk、红外线和新型蓝牙本地广播网络）到1 Gb/s（如Gigabit Ethernet）不等。10 Mb/s的速度最为普通，这是**10BaseT Ethernet**的速度。**100BaseT Ethernet**要求更高质量的电缆，但能以100 Mb/s的速度运行，正日渐流行。基于光纤的FDDI网络的使用也在增长。FDDI网络是基于令牌的，能以100 Mb/s的速度运行。

一个典型的LAN可以由许多不同的计算机（从大型机到笔记本或PDA）、各种共享外设（如激光打印机或磁带驱动器）和一个或多个网关（用以提供对其他网络访问的专用处理器）组成（图2.11）。以太网（Ethernet）通常用于构造LAN。以太网网络没有中央控制器；因为是多点访问总线，所以新主机能很容易地被添加到网络上。

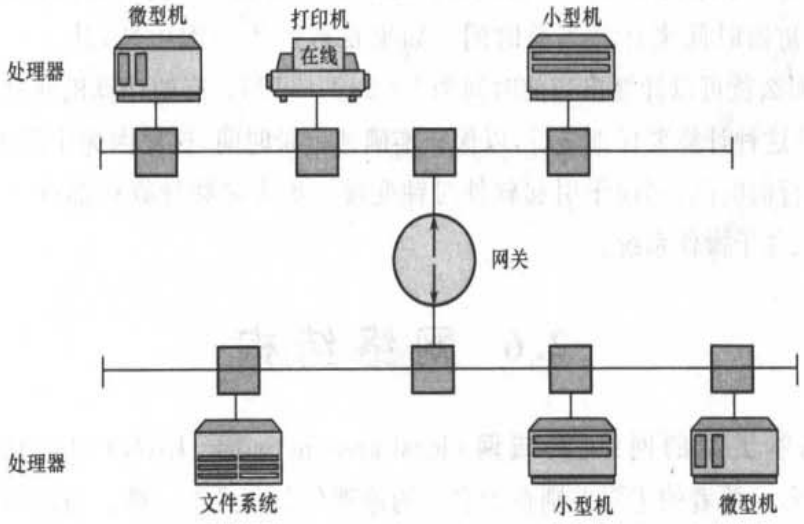


图2.11 局域网

2.6.2 广域网

广域网出现在20世纪60年代的后期，主要作为学术研究项目用于在各站点之间提供高效的通信，允许硬件和软件能被更广泛区域用户方便且经济地共享。第一个设计和开发的WAN是Arpanet。从1968年开始，Arpanet从四个节点的实验网络发展到全世界范围的网络，即因特网，由数百万计算机系统组成。

因为WAN的各个站点在物理上分布于较大的地理区域，所以在默认情况下，通信线路速度相对较慢且不可靠。典型链接为电话线、(专用于数据的)租用线、微波和卫星频道。这些通信链接由特殊的**通信处理器**(communication processor)所控制(图2.12)，它们负责定义各站点通过网络进行通信的接口，还有在各站点之间传输信息。

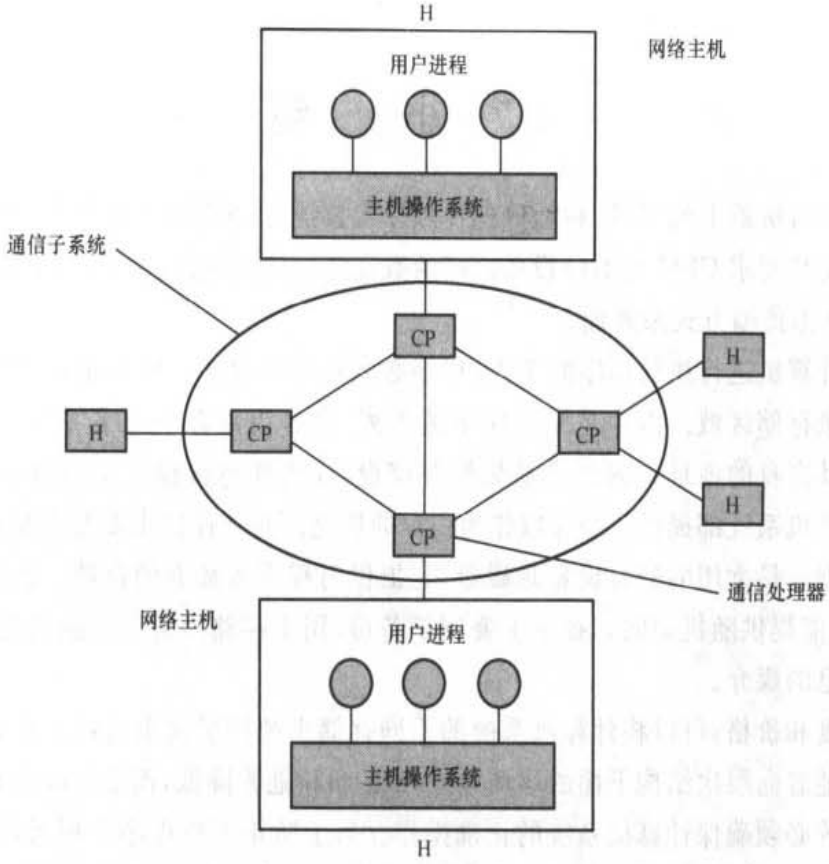


图2.12 广域网的通信处理器

例如，因特网WAN为地理上分布各地的主机提供了互相通信的能力。主机通常在类型、速度、字长和操作系统等方面都不尽相同。主机通常位于LAN上，LAN再通过区域性网络连接到因特网。区域网(如美国东北部的NSFnet)通过**路由器**(router)(15.4.2小节)在内部连接起来，以形成世界范围的网络。网络之间的连接通常采用称做T1的电话系统服务器，它在租用线路上提供1.544 Mb/s的传输速率。对于要求更快因特网访问的场所，可

将多个 T1 合并起来并行工作以提供更多的吞吐量。例如，T3 是由 28 个 T1 所连接组成，其传输速率为 45 Mb/s。路由器控制每条信息通过网络时所采用的路径。路由可以是动态的，以提高通信效率，或是静态的，以降低安全风险，或允许计算通信费用。

还有的正在使用中的 WAN 采用标准电话线作为其主要通信手段。调制解调器是这样一种设备：它能接收来自计算机的数字数据，并将它转换成模拟信号以供电话系统使用。目的端的调制解调器则将模拟信号转换回数字信号，以供目的端接收数据。UNIX 新闻网络和 UUCP 允许系统通过调制解调器按预先设定的时间互相通信，以交换消息。消息接着路由到其他相邻系统，通过这种方式传到网络上的所有主机（公共消息）或传送到其目的地（私有消息）。WAN 通常要比 LAN 慢；其传输速率从 1200 b/s 到 1 Mb/s 间不等。UUCP 现已被 PPP（点到点协议）所取代。PPP 能通过调制解调器连接工作，允许本地计算机完全地连接到因特网上。

2.7 小 结

通过将一台机器上的 CPU 和 I/O 操作加以交替执行，多道程序设计和分时系统改善了性能。这种交替要求 CPU 与 I/O 设备之间的数据传输按照轮询、对 I/O 端口的中断驱动访问或 DMA 数据传输方式来处理。

为了让计算机进行执行程序的工作，程序必须位于内存中。内存是处理器能直接访问的惟一大容量存储区域。内存是字或字节的阵列，容量为数百个千字节到数百个兆字节。每个字都有其自身的地址。内存是易失性存储设备，当断电或没有电源时会失去其内容。绝大多数计算机系统都提供了外存以作为内存的扩充。对外存的主要要求是它要能长久地存储大量数据。最常用的外存设备是磁盘，它提供对程序和数据的存储。磁盘是非易失的存储设备，且能提供随机访问。磁带主要用于备份，用于存储不常使用的信息，也作为系统之间传输信息的媒介。

根据速度和价格，可以将计算机系统的不同存储系统按层次来组织。最高层最为昂贵但也最快。随着向层次结构下面的移动，每一位的价格通常降低，而访问时间通常增加。

操作系统必须确保计算机系统的正确操作。为了防止用户程序干预系统的正常操作，硬件有两种模式：用户模式和监督程序模式。许多指令（如 I/O 指令和停机指令）都是特权的，只能在监督程序模式下执行。操作系统所驻留的内存也必须被加以保护，以防止用户程序修改。定时器防止无穷循环。这些工具（如双模式、特权指令、内存保护、定时器中断）是操作系统所使用的基本内置单元，用以实现正确操作。第三章将继续讨论操作系统所能提供的功能细节。

LAN 和 WAN 是两种基本类型的网络。LAN 通常用昂贵的双绞线或光纤电缆所连接，能允许分布在小地理区域内的处理器进行通信。WAN 通常用电话线、租用线、微波链

路或卫星信道来连接，能允许分布在大地理区域内的处理器进行通信。LAN 通常以高于 100 Mb/s的速率进行传输，而更慢的 WAN 则以 1 200 b/s 到超过 1 Mb/s 的速率进行传输。

习 题 二

2.1 **预取**(prefetching)是一个作业的 I/O 处理与这个作业的计算重叠进行的方法。其概念很简单。当一个读操作完成后，作业将要对数据进行处理时，让输入设备立刻开始进行下一个读操作。这样，CPU 和输入设备都在忙于工作。幸运的话，当作业就绪可处理下一个数据项时，输入设备就应完成那些数据项的读操作。CPU 接着开始处理新近读进来的数据，同时输入设备继续读下面的数据。对于输出可以使用相似的概念。这时，作业产生的数据先被放进一个缓冲区直到输出设备可以接受它们。

比较一下预取和**假脱机**(spooling)操作：一是 CPU 使作业的输入与作业的计算重叠进行，二是作业的输入与其他作业的输出重叠进行。

2.2 作为一个基本的系统保护(安全)形式，监督程序模式和用户模式之间有什么不同？

2.3 陷入与中断之间的区别是什么？各自有什么用途？

2.4 DMA 对哪些类型的操作有用？解释你的回答。

2.5 下面哪些指令应该是特权指令？

a. 设置定时器的值。

b. 读时钟。

c. 清除内存。

d. 关闭中断。

e. 从用户模式切换到监督程序模式。

2.6 一些计算机系统没有提供硬件的特权操作方式，在这样的计算机上能否建立一个安全操作系统？从可以和不可以两方面进行论述。

2.7 一些早期的计算机通过将操作系统放在一个无论是用户作业还是操作系统本身都不能修改的内存区域的方式来保护操作系统。描述你认为这种方式可能产生的两个难点。

2.8 保护操作系统对于确保计算机系统正确运行是很关键的。为了提供保护，开发人员设计了双模式操作、内存保护和计时器。然而，为了提供最大的灵活性，还要对用户设置最小的限制。

下面是一个通常被保护的指令列表，选出必须被保护的最小指令集。

a. 改变到用户模式。

b. 改变到监督程序模式。

c. 读监督程序内存区。

d. 写入监督程序内存区。

e. 从监督程序内存区取一条指令。

f. 开启定时器中断。

g. 关闭定时器中断。

2.9 给出高速缓存有用的两个理由。它们解决何种问题？它们引发何种问题？如果一个高速缓存可以做得和要缓存的设备一样大(如一个和磁盘一样大的缓存)，为什么不做那么大的缓存并且撤销那个设

备?

2.10 写一个不会被恶意或未经调试的用户程序所破坏的操作系统需要硬件的支持。列出编写操作系统的三个这种硬件支持的名字,描述一下它们是怎样在一起应用,以保护操作系统的。

2.11 一些CPU提供了不止两种运行模式,这些多重模式可能的两个用处是什么?

2.12 WAN和LAN之间的主要区别是什么?

2.13 什么样的网络配置最适合下面的这些环境?

a. 宿舍的一层。

b. 一个大学校园。

c. 一个州。

d. 一个国家。

推荐读物

Hennessy和Patterson[1996]全面讨论一般I/O系统、总线和系统结构。Tanenbaum[1990]从详细的硬件级别开始,描述了微型机的体系结构。

Freedman[1983]和Harker等[1981]论述了磁盘技术。Kenville[1982]、Fujitani[1984]、O'Leary和Kitts[1985]、Gait[1988]、Olsen及Kenley[1989]全面讨论了光盘。Pechura和Schoeffler[1983]以及Sarisky[1983]讨论了软盘。

Smith[1982]描述并分析了高速缓存(包括关联存储器),该论著中还包括此问题的大量相关参考文献。

Chi[1982]和Hoagland[1985]论述大容量存储技术。

Tanenbaum[1996]和Halsall[1992]提供了计算机网络的概观。Fortier[1989]给出了关于计算机网络硬件和软件的详细论述。

第三章　操作系统结构

操作系统为执行程序提供环境。从内部结构来说，操作系统的组成变化很大，有很多不同路线的组织方式。设计一个新的操作系统是个大型任务。在开始设计前，必须定义好系统目标。所要设计系统的类型为各种算法和策略的选择提供了依据。

操作系统可以从多个优势角度来研究。第一是通过检验所提供的服务。第二是通过考察为用户和程序员所提供的接口。第三是分解研究系统的各个组成部分及其相互连接。在本章里，将从用户角度、程序员角度和操作系统设计人员角度来分别研究操作系统的三个方面。将研究操作系统提供什么服务、如何提供服务、设计这种操作系统的不同方法论。

3.1　系统组成

只有通过分而治之的方法才能创建像操作系统这样庞大且复杂的系统。所划分的每块应该是完全描述好的系统部分，每块都有仔细定义过的输入、输出和功能。显然，并不是所有系统都是同构的。然而，许多现代操作系统都共同支持拥有在小节 3.1.1 到 3.1.8 小节中所描述的系统组成模块。

3.1.1　进程管理

一个程序的指令如果不被 CPU 执行，那么这个程序什么都不会做。**进程**暂时可以看做是正在执行的程序，但是随着研究的不断深入，进程的定义会更加广泛。分时用户程序如编译程序是进程。由个人计算机上的单个用户所运行的字处理程序是进程。系统任务，比如将输出发送到打印机，也是进程。就目前而言，你可以将进程看做是作业或分时程序；而在后面，进程概念将会更加广泛。在第四章，会提供允许进程创建子进程以并发执行的系统调用。

进程完成其任务需要一定的资源，包括 CPU 时间、内存、文件和 I/O 设备等。这些资源可以在创建进程时赋予进程或在执行进程时分配给进程。除了在创建时进程所得到的各种物理和逻辑资源外，也会接受传输过来的各种初始化数据（或输入）。例如，考虑这样一个进程，其功能是在终端屏幕上显示文件的状态。该进程会得到一个文件名作为输入，会执行适当的指令和系统调用以得到并在终端上显示所要的信息。当进程终止时，操作系统会收回所有可再用的资源。

再次强调一个程序本身并不是进程；一个程序只是一个*被动实体*，如同存储在磁盘上的文件的内容，然而一个进程是*活动实体*，它有一个**程序计数器**(program counter)以指示下一个所要执行的指令。一个进程的执行必须是顺序的。CPU一个个地执行进程中的指令，直到进程终止。再者，在任何时候，最多只有一个指令被以进程的名义执行。因此，虽然两个进程可能与同一个程序相关联，但是这两个进程都有其各自的执行顺序。通常一个程序在运行时可以产生多个进程。

进程是系统的工作单元。这样一个系统由多个进程组成，其中有些是操作系统进程(以执行系统代码)，其余那些是用户进程(以执行用户代码)。所有这些进程通过多路复用其内的CPU才能潜在地并发执行。

操作系统负责下列有关进程管理的活动：

- 创建和删除用户进程和系统进程；
- 暂停和重启进程；
- 提供进程同步机制；
- 提供进程通信机制；
- 提供死锁处理机制。

在第四章到第七章中讨论进程管理技术。

3.1.2 内存管理

正如第一章中所述，内存是现代计算机系统操作的中心。内存是字节或字的一个大的阵列，其大小从数十万到数十亿。每个字或字节都有其自己的地址。内存是可以被CPU和I/O设备所共同快速访问的数据的仓库。中央处理器在获取指令周期内从内存中读取指令，而在获取数据周期内对内存中的数据进行读出或写入。通过DMA所实现的I/O操作也会对内存中的数据进行读出和写入。内存通常是CPU所能直接寻址和访问的惟一大容量存储器。例如，如果CPU需要处理磁盘内的数据，那么这些数据必须首先通过CPU产生的I/O调用传送到内存中。同样，如果CPU需要执行指令，那么这些指令必须在内存中。

如果一个程序要执行，那么它必须先映射成绝对地址并装入内存。随着程序的执行，进程可以通过产生绝对地址来访问内存中的程序指令和数据。最后，程序终止，其内存空间得以释放，并且下一个程序可以被装入并执行。

为改善CPU的利用率和计算机对用户的响应速度，必须在内存中保留多个程序。内存管理方法有很多，不同算法的效能与特定环境有关。对于某一特定系统的内存管理方法的选择取决于诸多因素，尤其是系统的*硬件*设计。每个算法都要求特定的硬件支持。

操作系统负责下列有关内存管理的活动：

- 记录内存的哪部分正在被使用及被谁使用。
- 当内存空间可用时，决定哪些进程可以装入内存。

- 根据需要分配和释放内存空间。

许多内存管理技术将在第九章和第十章中讨论。

3.1.3 文件管理

文件管理是操作系统中最为常见的组成部分。计算机可以在多种类型的物理媒介上存储信息。磁带、磁盘和光盘是最常用的媒介。这些媒介都有自己的特点和物理组织。每种媒介由一个设备来控制，如磁盘驱动器或磁带驱动器等，它们都有自己的特点。这些属性包括访问速度、容量、数据传输速率和访问方法(顺序或随机)。

为了便于使用计算机系统，操作系统提供了统一的逻辑信息存储观点。操作系统对存储设备的物理属性进行了抽象，定义了逻辑存储单元即文件。操作系统将文件映射到物理媒介上，并通过这些存储设备访问这些文件。

文件是由其创建者定义的一组相关信息的集合。通常，文件表示程序(源程序和目标程序)和数据。数据文件可以是数值的、字符的或字符数值的。文件可以没有形式(例如，文本文件)，或有严格的形式(例如，固定域)。文件可由一系列位、字节、行或记录组成，其具体含义由其创建者来定义。文件概念相当广泛。

操作系统通过管理大容量存储媒介(如磁盘和磁带)及控制它们的设备，来实现文件这一抽象概念。而且，文件通常组织成目录，以方便使用。最后，当多个用户访问文件时，需要控制由什么人及按什么方式(例如，读、写、附加)来访问文件。

操作系统负责下列有关文件管理的活动：

- 创建和删除文件；
- 创建和删除目录；
- 提供操作文件和目录的原语；
- 将文件映射到二级存储器(辅存)上；
- 在稳定(非易失的)存储媒介上备份文件。

文件管理技术将在第十一章和第十二章中讨论。

3.1.4 输入/输出系统管理

操作系统的目标之一是为用户隐藏特定硬件设备的特质。例如，在 UNIX 系统中采用 I/O **子系统**从操作系统自身的主体中隐藏了 I/O 设备的特质。I/O 子系统由如下部分组成：

- 包括缓冲器、高速缓存和脱机打印的内存管理部分；
- 一个通用设备驱动程序的接口；
- 用于特定硬件设备的驱动程序。

只有设备驱动程序才知道被指定的设备的特质。

在第二章中讨论了如何用中断处理程序和设备驱动程序来构造有效的I/O子系统。在第十三章中将研究I/O子系统如何与其他系统构件相交互、如何管理设备、如何传输数据和检测I/O的完成。

3.1.5 二级存储管理

计算机系统的主要目标是执行程序。这些程序，加上其所访问的数据，在执行时应位于**内存**(或主存)中。由于内存太小而不能容纳所有数据和程序，再加上掉电后它会失去所拥有的数据，计算机系统必须提供**二级存储器**(secondary storage)，以备份内存。绝大多数现代计算机系统都采用硬盘作为主要在线存储媒介来存储程序和数据。许多程序(如编译程序、汇编程序、排序程序、编辑器和格式化程序等)都存储在硬盘上，要执行时才调入内存，在执行时将硬盘作为处理的来源地和目的地。因此，硬盘存储的正确管理对计算机系统尤为重要。

操作系统负责下列有关硬盘管理的活动：

- 空闲空间管理；
- 存储空间分配；
- 硬盘调度。

由于二级存储器使用频繁，因此必须高效使用。计算机操作的整体速度可能与硬盘子系统的速度和管理该子系统的算法有关。二级存储管理技术将在第十四章中讨论。

3.1.6 联网

分布式系统(distributed system)是一组不共享内存、外设和时钟的处理器的集合。这些处理器都有各自的内存和时钟，处理器之间通过各种通信线路(如高速总线或网络)进行通信。分布式系统的处理器的大小和功能各有差异。它们可包括小型微处理器、工作站、小型机、大型通用计算机系统。

系统中的处理器通过**通信网络**相连，构成连接的方式多种多样。网络可以是完全连接的，也可以是部分连接的。通信网络设计必须考虑消息路由、连接策略、竞争和安全问题等。

分布式系统可以将物理上分开的、可能异构的系统组织成为一个连贯的系统，以便用户能访问系统所维护的各种资源。对共享资源的访问可以使计算加速、功能加强、提高数据可用性以及增强可靠性。操作系统通常将网络访问作为文件访问的一种推广，而将网络细节隐藏在网络接口的设备驱动程序内。创建分布式系统的协议可对系统的可用性和流行程度产生很大影响。万维网(World Wide Web)的创新就是为信息共享而创建的一种新的访问方式。它改进了已有的**文件传送协议**(FTP)和**网络文件系统**(NFS)协议，用户无需登录就可访问远程资源。它定义了一个用于网络服务器和网络浏览器之间的通信的新的协议，**超文本传输协议**(http)。网络浏览器只需要向一个远程机器的网络服务器发送一个信息请

求，信息（文本、图形、与其他信息的连接）就会返回。这种更加方便的使用极大地促进了 http 和网络广泛应用的增长。

本书将在第十五章到第十七章中讨论网络和分布式系统。

3.1.7 保护系统

如果一个计算机系统有许多用户且允许多个进程并发执行，那么各个进程应得到保护，以免受其他进程活动的影响。为此，系统必须提供一定的机制，以确保只有那些已获取操作系统正当授权的进程才可以使用相应的文件、内存段、CPU 和其他资源。

例如，内存寻址硬件确保一个进程只能在其地址空间内执行。定时器确保没有进程可以一直占有 CPU 控制权而不释放它。设备控制寄存器并不能被用户直接访问，这样各种外设的完整性就得以保护。

保护是控制程序、进程或用户访问由计算机系统定义的资源的机制。这种机制必须提供一定的方法以规定所有要进行的控制以及加强控制的方法。

保护通过检测各组成子系统之间接口的潜在错误来改善可靠性。接口错误的早期检测通常能够防止一个健全的子系统受另一个不能正常工作的子系统的破坏。不受保护的资源不能阻止未经授权或不合格用户的使用（或误用）。正如第十八章所要讨论的，一个基于保护的系统能够提供手段以区分已经授权和未经授权的使用。

3.1.8 命令解释系统

操作系统中最为重要的系统程序之一是**命令解释程序**，它是用户和操作系统之间的接口。有的操作系统在其内核部分包括命令解释程序。其他操作系统，如 MS-DOS 和 UNIX，将命令解释程序作为一个特殊程序，当一个作业开始时或用户首次登录（分时系统）时，该程序就会运行。

许多命令通过**控制语句**（control statement）交给操作系统。当一个位于批处理系统的新作业开始时或一个用户首次登录到分时系统时，一个读入和解释控制语句的程序就会自动执行。该程序有时称为**控制卡解释程序**（control-card interpreter）或**命令行解释程序**（command-line interpreter），通常称为**外壳**（shell）。其功能很简单：得到下一个命令语句并执行它。

操作系统经常按所采用的外壳来加以划分，一个用户友好的命令解释程序会使系统更适于某些用户。一种用户友好的接口风格是基于鼠标的窗口和菜单系统，如 Macintosh 和微软公司的 Windows。鼠标可以移动，以定位鼠标指针于屏幕上的图像或**图标**，屏幕上的图标代表程序、文件和系统功能。根据鼠标指针的位置，按一下鼠标按钮可以调用程序、选择文件或目录（称做**文件夹**）或打开包含命令的菜单。更为功能强大、复杂和难学的外壳可能被其他用户所喜欢。对有些外壳，命令是通过键盘输入的，并显示在屏幕或打印机终端

上，回车键表示命令完毕并就绪可执行。MS-DOS 和 UNIX 的外壳就是这样工作的。

命令语句本身处理进程的创建和管理、I/O 操作、辅存管理、内存管理、文件系统访问、保护和联网等。

3.2 操作系统服务

操作系统提供一个环境以执行程序。它向程序和这些程序的用户提供一定的服务。当然，所提供的具体服务随操作系统而不同，但是还是有一些共同特点的。这些操作系统用来方便程序员，使得编程任务更加容易。

• **程序执行**：系统必须能将程序装入内存并运行该程序。程序必须能结束其执行，包括正常或不正常结束（指明错误）。

• **I/O 操作**：运行程序可能需要 I/O。这些 I/O 可能涉及文件或 I/O 设备。对于特定设备，需要特定的功能（如重绕磁带驱动器或清 CRT 屏幕）。为了效率和保护，用户通常不能直接控制 I/O 设备。因此，操作系统必须提供进行 I/O 操作的方法。

• **文件系统操作**：文件系统特别重要。很明显，程序需要读、写文件。程序也需要根据文件名来创建和删除文件。

• **通信**：在许多情况下，一个进程需要与另一个进程交换信息。这种通信主要有两种形式。一种是发生在同一台计算机上运行的两个进程之间；另一种是发生在由计算机网络连接起来的不同计算机上的进程之间。通信可以通过**共享内存**来实现，也可通过**消息交换**技术来实现（对于消息交换，信息包通过操作系统在进程之间移动）。

• **错误检测**：操作系统通常需要知道可能出现的错误。错误可能发生在 CPU 和内存硬件中（如内存错误或电源失败）、I/O 设备中（如磁带奇偶出错、网络连接出错、打印机缺纸）和用户程序中（如算术溢出、试图访问非法内存地址或使用 CPU 时间太长）。对于每种类型的错误，操作系统应该采取适当的行动，以确保正确和一致的计算。

另外，还有一组操作系统函数，不在于帮助用户而是确保系统本身高效运行。多用户系统可以通过在用户间共享计算机资源来提高效率。

• **资源分配**：当多个用户登录到系统上或多个作业同时执行时，系统必须为每个进程分配资源。操作系统管理多种不同类型的资源。有的资源（如 CPU 周期、内存和文件存储器）可能要有特别的分配代码，而其他的资源（如 I/O 设备）可能只需要通用的请求和释放代码。例如，为了确定如何最好地使用 CPU，操作系统需要采用 CPU 调度程序以考虑 CPU 的速度、必须执行的作业、可用的寄存器数和其他因素。也可能有一些程序为一个作业分配磁带驱动器以供使用。一个程序找到一个未用的磁带驱动器并在内部表内加上标记以记录驱动器的新用户。另一程序用来清除表格。这些程序还可以分配绘图仪、调制解调器和其他外设。

• **统计**：人们需要跟踪哪些用户使用了多少和什么类型的计算机资源。这种记录可用于记账(以便让用户交费)，或仅用于建立使用统计数据。使用统计数据对研究人员来讲是有价值的工具，可用于重新配置系统，以提高计算服务能力。

• **保护**：对于保存在多用户计算机系统中的信息，其持有者可能需要控制信息的使用。当多个不连续进程并发执行时，一个进程不能干预另一个进程或操作系统本身。保护涉及到确保控制所有对系统资源的访问。保护系统不受外界侵犯的**安全性**很重要。这种安全从每个用户向系统证明自己(通过利用密码的方法)开始，到允许对资源进行访问。安全也包括保护外部 I/O 设备(如调制解调器和网络适配器)不受非法访问，并记录所有这样的连接，以检测非法闯入的企图。一个要保护和安全的系统中的所有部分都要建立预防。一条链子的坚固程度只能与最弱的链环段一样。

3.3 系统调用

系统调用(system call)提供了进程与操作系统之间的接口。这些调用通常以汇编语言指令的形式提供，它们也常常列在汇编语言开发人员所用的各种手册中。

有的系统允许系统调用直接为高级语言程序所用，这时系统调用通常类似预先定义的函数或子程序调用。它们也可能调用一个特别运行子程序以产生系统调用，或者直接在线生成系统调用。

有些语言(如 C、C++ 和 Perl)已经取代了汇编语言而直接用于系统编程。这些语言允许直接调用系统调用。例如，UNIX 系统调用可以在 C 或C++ 程序中直接调用。现代微软公司 Windows 平台的系统调用是 Win32 **应用程序接口**的一部分，可以为微软公司所有 Windows 平台的编译程序所用。

作为一个使用系统调用的例子，考虑编写一个从一个文件读取数据并复制到另一个文件的简单程序。程序所需要的第一个输入是两个文件的名称：输入文件名和输出文件名。根据操作系统设计的不同，这些名称有许多不同的表示方法。一种方法是程序向用户提问后得到两个文件名。对于交互式系统，这种方法需要一系列的系统调用：先在屏幕上写出提示信息，再从键盘上读取定义两个文件名称的字符。对于基于鼠标的窗口一菜单系统，一个文件名的菜单通常显示在一个窗口中。用户通过鼠标选择源文件名，打开另一个类似窗口可以用来选择目的文件名。

在得到两个文件名后，该程序必须打开输入文件并创建输出文件。每个这样的操作都需要另一个系统调用并可能碰到错误条件。当程序设法打开输入文件时，它可能发现该文件名的文件不存在或者该文件受保护而不能访问。在这些情况下，程序应该在终端上打印出消息(另一系列系统调用)，并且非正常地终止(另一个系统调用)。如果输入文件存在，那么必须创建一个新的输出文件。可能会发现具有同一名称的输出文件已存在。这种情况可

能导致程序中止(一个系统调用),或者可以删除现有文件(另一个系统调用)并创建新的文件(另一个系统调用)。对于交互式系统,另一选择是询问用户(一系列的系统调用以输出提示信息并从键盘读入响应)是否需要替代现有文件或中止程序。

现在两个文件都已设置好,进入循环以从输入文件中读(一个系统调用)并向输出文件中写(另一个系统调用)。每个 read 和 write 都必须返回一些关于各种可能错误的状态信息。对于输入,程序可能发现已经到达文件的结尾,或者在 read 过程中发生了一个硬件失败(如奇偶检验误差)。对于输出,根据输出设备的不同可能出现各种错误(如没有剩余可用磁盘空间、到达磁带物理结尾、打印机没有纸)。

最后,在整个文件复制完成后,程序可以关闭两个文件(另一个系统调用),在终端上写一个消息(更多系统调用),最后正常结束(最后的系统调用)。正如大家所看见的,即便一个简单的程序也会大量使用操作系统。

不过,绝大多数用户绝不会看到这些级别的细节。绝大多数程序设计语言的运行时支持系统(与编译器一起构造的函数库)提供了更加简单的接口。例如,C++ 中的 cout 语句可能编译成对运行时支持程序的一个调用,以发布必需的系统调用,检查错误,最后返回到用户程序。因此,操作系统接口的绝大多数细节被程序设计人员用编译器和运行时支持包隐藏起来。

系统调用根据使用的计算机的不同而有不同的发生方式。通常,需要提供比所需系统调用识别符更多的信息。这些信息的具体类型和数量根据特定操作系统和调用而有所不同。例如,为了获取输入,可能需要指定作为源来使用的文件或设备,和用于缓存将读取的输入的内存区域的地址和长度。当然,设备或文件和长度也可以隐含在调用中。

向操作系统传递参数通常用三种方法。最简单的方法是通过 *寄存器* 来传递参数。不过有时,参数数量会比寄存器多。这时,这些参数通常存放在内存的 *块* 或表中,并将块的地址作为参数传递给寄存器(图 3.1)。Linux 就采用这种方法。参数也可通过程序放在或 *压到* *堆栈* 中,并通过操作系统 *弹出* 堆栈。有的操作系统偏爱块或堆栈的方法,因为这些方法

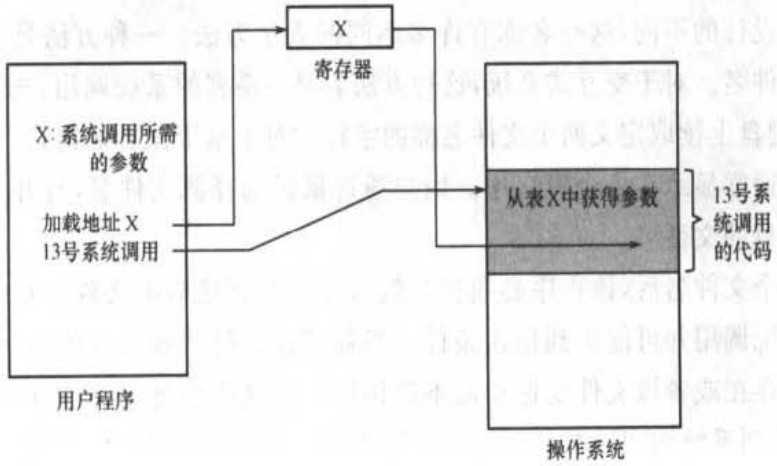


图 3.1 参数作为表传递

并不限制所传递参数的数量或长度。

系统调用大致可分成五大类：**进程控制**、**文件管理**、**设备管理**、**信息维护**和**通信**。在3.3.1小节到3.3.5小节，将简要描述操作系统可能提供的系统调用类型。这些系统调用大多支持后面几章所讨论的有关概念和功能，或被其所支持。图3.2总结了操作系统通常提供的系统调用类型。

- 进程控制
  - ◇ 结束、中止
  - ◇ 装入、执行
  - ◇ 创建进程、终止进程
  - ◇ 取得进程属性、设置进程属性
  - ◇ 等待时间
  - ◇ 等待事件、唤醒事件
  - ◇ 分配和释放内存
- 文件管理
  - ◇ 创建文件、删除文件
  - ◇ 打开、关闭
  - ◇ 读、写、重定位
  - ◇ 取得文件属性、设置文件属性
- 设备管理
  - ◇ 请求设备、释放设备
  - ◇ 读、写、重定位
  - ◇ 取得设备属性、设置设备属性
  - ◇ 逻辑连接或断开设备
- 信息维护
  - ◇ 读取时间或日期、设置时间或日期
  - ◇ 读取系统数据、设置系统数据
  - ◇ 读取进程、文件或设备属性
  - ◇ 设置进程、文件或设备属性
- 通信
  - ◇ 创建、删除通信连接
  - ◇ 发送、接收消息
  - ◇ 传递状态信息
  - ◇ 连接或断开远程设备

图3.2 系统调用的类型

3.3.1 进程控制

运行程序需要能正常或非正常地中断其执行(end或abort)。如果一个系统调用被用来非正常地中断正在执行的程序，或者程序运行碰到问题而引起错误陷阱，那么可能会有内

存信息转储并产生一个错误信息。内存信息转储通常写到磁盘上，可被调试器检查用来确定问题原因。不管是正常还是非正常中止的情况，操作系统都必须将控制权转交给命令解释器。命令解释器接着读取下一个命令。对于交互式系统，命令解释器只不过简单地读取下一个命令；因为假定用户会发出合适的命令以响应任何错误。对于批处理系统，命令解释器通常终止整个作业并继续下一个作业。当出现一个错误的时候，有的系统允许控制卡来指出一个具体的恢复动作。如果程序发现其输入有错并想要非正常地终止，那么它可能也需要定义一个错误级别。更加严重的错误可以用更高级的错误参数来表示。如果将正常终止定义为级别为0的错误，那么可以将正常和非正常终止混合起来。命令解释器或下一个程序能利用错误级别来自动决定下一个动作。

执行一个程序的进程或作业可能需要装入和执行另一个程序。这一特点允许命令解释器来执行一个程序，该命令可通过诸如用户命令、鼠标点击或批处理命令来表示。当装入程序终止时，一个有趣的问题是控制权返回到哪里。这个问题与现有程序是否丢失、保存或允许与新程序继续并发执行有关。

如果新程序终止时控制权返回到现有程序，那么必须保存现有程序的内存映像；因此，事实上已经创建了一个机制，以便一个程序调用另一个程序。如果两个程序并发继续，那么就创建一个新作业或进程，以便多道执行程序。通常，有的系统调用专门用于这一目标（创建进程或提交作业）。

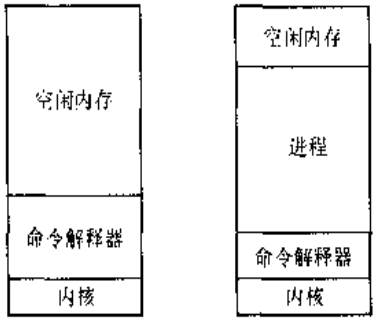
如果创建一个新作业或进程，或者甚至是一组作业或一组进程，那么应该能控制它的执行。这种控制要求能确定和重新设置作业或进程的属性，包括作业的优先级、其最大允许执行时间等（获取进程属性和设置进程属性）。如果发现所创建的作业或进程不正确或不再需要，那么也可能需要能终止它（终止进程）。

创建了新作业或进程之后，可能需要等待其完成执行。也许需要等待一定时间（等待时间）；更有可能，也许需要等待某个事件的出现（等待事件）。当那个事件出现时，作业或进程就应当会唤醒（唤醒事件）。这种类型的系统调用，它处理着并发进程的协同工作，将在第七章深入讨论。

另一组系统调用有助于调试程序。许多系统提供转储内存信息的系统调用。这种作用有助于调试。程序 trace 在执行时能列出所用的每个系统调用指令，但是只有少数几类系统提供。即使微处理器也提供一个称为**单步**的 CPU 模式，这种模式在每个指令运行后 CPU 能执行一个陷阱。该陷阱通常为调试程序所捕获，调试程序是系统程序，主要设计用于帮助程序员发现并纠正错误。

许多操作系统都提供程序的时间表。这表示一个程序在某个位置或某些位置处执行所花费的时间。时间表要求跟踪功能或定期的定时器中断。在每次出现定时器中断时，会记录程序计数器的值。如果有足够频繁的定时器中断，就可得到程序各部分所用时间的统计数据图。

进程和作业控制有许多方面特性和变化，本书将通过例子来解释这些概念。MS-DOS 操作系统是一个单任务系统的例子，它在计算机启动时就调用一个命令解释程序（图 3.3(a)）。由于 MS-DOS 是单任务的，它采用了一个简单方法来运行程序且不创建新进程。它将程序装入内存，并改写它自己的绝大部分以便为新程序提供尽可能多的内存空间（图 3.3(b)）。然后它将指令指针设为指向程序的第一条指令。接着程序运行，或者一个错误会引起陷阱，或者程序执行一个系统调用以终止。不论如何，错误代码会保存在系统内存中以便在后面使用。随后，命令解释程序中尚未改写的一小部分会重新开始执行。其首要任务是从磁盘中重新装入命令解释器的其他部分。当任务完成时，命令解释器会向用户或下一个程序提供可用的错误代码。



(a) 系统启动时　(b) 执行程序时

图 3.3　MS-DOS 执行状态

虽然 MS-DOS 操作系统并不具备一般的多任务功能，但是它确实提供了一个有限的并发执行的方法。TSR 是这样一种程序：通过 **TSR**（terminate and stay resident，终止与驻留）系统调用，它能与中断相连，然后退出。例如，它能与时钟中断相连；将其子程序之一的地址加入到中断处理程序的列表中，当触发系统时钟时调用 TSR 程序。这样，TSR 程序每个时钟段在一秒内可执行多次。TSR 系统调用允许 MS-DOS 为 TSR 保留所占有的空间，这样在命令解释程序重新装入时不会被改写。

FreeBSD 是多任务系统的一个例子。当用户登录到系统时，用户所选择的外壳（或命令解释程序）开始运行。这种外壳类似于 MS-DOS 外壳：接受命令并执行用户所要求的程序。不过，由于 FreeBSD 是多任务系统，命令解释程序在另一个程序执行时可继续运行（图 3.4）。为了启动新进程，外壳执行 fork 系统调用。接着，所选择的程序通过 exec 系统调用装入内存，程序就开始执行。根据命令发出的方式，外壳或者等待进程完成，或者"在后台"运行进程。对于后一种情况，外壳可马上请求下一个命令。当进程在后台运行时，它不能直接接受键盘输入，因为外壳在使用这个资源。因此 I/O 通过文件或通过窗口－菜单接口完成。同时，用户可以随意要求外壳运行其他程序，监视运行进程的进展，改变程序优先级，等等。当进程完成时，它执行 exit 系统调用以终止，并将 0 状态代码或非 0 错误代码返回给调用进程。这一状态（或错误）代码可为外壳或其他程序所使用。进程将在第四章讨论。

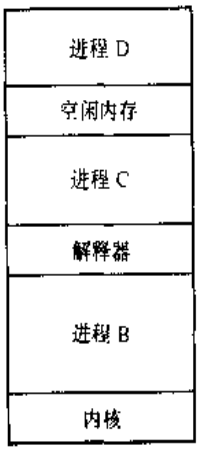


图 3.4　运行多个程序的 UNIX

3.3.2　文件管理

在第十一章和第十二章将深入讨论文件系统。不过，现在可指出一些有关文件的常用

系统调用。

首先要能创建和删除文件。每个系统调用都需要文件名，还可能需要一些文件属性。一旦创建了文件后，就需要打开并使用它。也可能需要读、写或重定位(例如，重绕到或跳到文件末尾)。最后，要关闭文件，以表示不再使用它了。

如果用目录结构来组织文件系统内的文件，那么目录也需要同样的一组操作。另外，不管是文件还是目录，都需要能确定其不同属性的值，或在必要时重新设置其属性。文件属性包括文件名、文件类型、保护模式代码、计账信息等。至少需要两个系统调用(读取文件属性和设置文件属性)完成这一功能。有的操作系统提供更多的调用。

3.3.3 设备管理

程序在运行时可能需要用到其他额外的资源才能继续运行。额外资源可能是更多内存、磁带驱动器、文件访问等。如果有可用资源，那么系统允许请求，控制可返回到用户程序；否则，程序必须等待可用的足够多的资源。

文件可看做抽象或虚拟设备。因此，文件的许多系统调用对设备来说也是需要的。但是，如果系统有多个用户，那么必须请求设备以确保能独自使用它。在使用完设备之后，需要释放它。这些函数类似于文件的打开和关闭(open 和 close)系统调用。

一旦请求了设备(并得到设备)之后，就能如同对普通文件一样，来对设备进行读、写和(可能的)重定位。事实上，I/O 设备和文件非常相似，以至于许多操作系统(如 UNIX 和 MS-DOS)将这两者合并为**文件－设备结构**。这时，I/O 设备可通过特殊文件名来标识。

3.3.4 信息维护

许多系统调用只不过用于在用户程序与操作系统之间传输信息。例如，绝大多数系统都有一个系统调用以返回当前的时间和日期。其他系统调用可返回系统的相关信息，如当前用户数、操作系统的版本号、空闲内存或磁盘空间的大小等。

另外，操作系统维护所有进程的信息，有些系统调用可访问这些信息。一般来说，也有系统调用用于重新设置进程信息(读取进程属性和设置进程属性)。在 4.1.3 小节，会讨论通常需要维护什么信息。

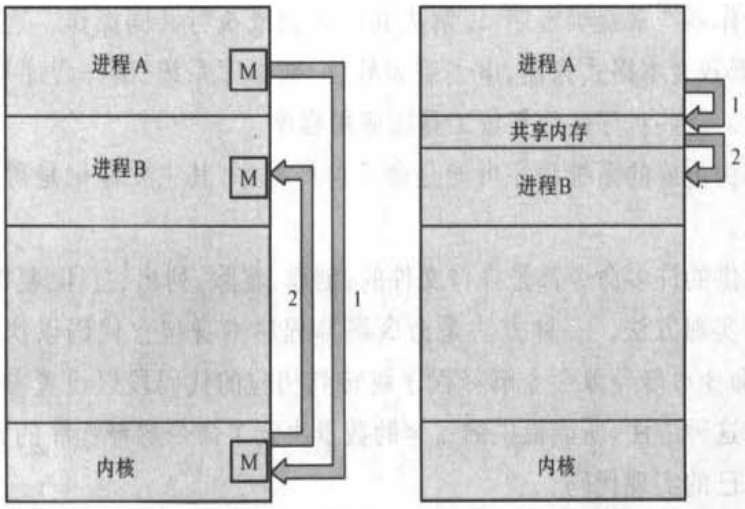
3.3.5 通信

有两种常见的通信模型。对于**消息传递模型**(message-passing model)，信息通过操作系统所提供的进程间通信工具来交换信息。在发生通信前，必须先开通连接。必须知道另一个通信实体的名称，它可能是同一 CPU 上的另一个进程，也可能是通过通信网络相连的另一台计算机上的进程。网络上的每台计算机都有一个如 IP 名的**主机名**，这通常是已知的。类似地，每个进程也有**进程名**，它通常转换成同等标识符，以便操作系统可以引用。系

统调用 get hostid 和 get processid 用于这一转换。这些标识符再传递给文件系统所提供的通用 open 和 close 系统调用，或特别的 open connection 和 close connection 系统调用。这是由系统的通信模型决定的。接受方进程通常通过 accept connection 调用来允许通信发生。能接受连接的大多数进程是针对特殊用途的**后台程序**，这些是为专用目的提供的系统程序。它们执行 wait for connection 调用，当有连接时会被唤醒。通信源，称为 *客户机*，而接收的后台程序，称为 *服务器*，通过 read message 和 write message 系统调用来交换消息。close connection 调用会终止通信。

对于**共享内存模型**(shared-memory model)，进程使用 map memory 系统调用来获得其他进程所拥有的内存区域的访问权。回想一下操作系统通常试图阻止一个进程访问另一个进程的内存。共享内存要求多个进程都同意取消这一限制。这样它们就可以通过读、写数据到共享区域来交换信息。数据的形式和位置是由这些进程来确定的，而不受操作系统所控制。进程还要负责确保它们不会同时向同一个地方写。这些机制将在第七章讨论。本书也会讨论进程模型的一种变形即**线程**，它们缺省就共享内存。线程将在第五章介绍。

这两种方法在操作系统中都常用，有的系统甚至两种都实现了。消息传递在要交换的数据量少时很有用，这是因为不必避免冲突。它也比用于计算机间通信的共享内存更容易实现。共享内存允许通信的最大速度和方便性，这时因为它能在计算机内以内存的速度进行。不过，在保护和同步方面仍存在问题。图 3.5 比较了两种类型的通信模型。



(a) 消息传递　　(b) 共享内存

图 3.5　通信模型

3.4 系统程序

现代系统的另一方面特性是一组系统程序。回想一下图 1.1，它描述了计算机逻辑层

次。最低层是硬件。其上是操作系统，接着是系统程序，最后是应用程序。系统程序提供了一个方便的环境，以开发程序和执行程序。有的只是系统调用的简单用户接口；其他的可能是相当复杂的。它们可分为如下几类：

• **文件管理**：这些程序创建、删除、复制、重新命名、打印、转储、列出和一般操作文件和目录。

• **状态信息**：这些程序仅从系统那里得到日期、时间、可用内存或磁盘空间的数量、用户数或类似状态信息。这些信息经格式化后，再打印到终端或其他输出设备或文件。

• **文件修改**：有多个文本编辑器可用来创建和修改位于磁盘或磁带上的文件的内容。

• **程序语言支持**：常用程序设计语言(如 C、C++、Java、Visual Basic 和 Perl 等)的编译程序、汇编程序、解释程序通常与操作系统一起向用户提供。这些程序有的是需要标价和单独提供的。

• **程序装入和执行**：一旦程序汇编或编译后，它必须装入内存才能执行。系统可能要提供绝对加载程序、重定位加载程序、链接编辑器和覆盖式加载程序。系统还需要高级语言或机器语言的调试程序系统。

• **通信**：这些程序提供了在进程、用户和不同计算机系统之间创建虚拟连接的机制。它们允许用户在互相的屏幕上发送消息、浏览网页、发送电子邮件消息、远程登录或从一台机器向另一台机器传送文件。

绝大多数操作系统都提供程序，以解决共同问题或执行共同操作。这些程序包括网页浏览器、字处理器和文本格式化器、电子制表软件、数据库系统、编译程序编译器、绘图和统计分析包和游戏。这些程序称为**系统工具或应用程序**。

操作系统最为重要的系统程序可能是命令解释程序，其主要作用是得到并执行下一个用户指定的命令。

这一层中提供的许多命令都是操作文件的：创建、删除、列出、打印、复制、执行等。这些命令有两种通用实现方法。一种方法是命令解释程序本身包含代码以执行这些命令。例如，删除文件的命令可能导致命令解释程序跳转到相应的代码段以设置参数和执行适当的系统调用。对于这种方法，所能提供的命令的数量决定了命令解释程序的大小，这是因为每个命令需要它自己的实现代码。

另一种可供选择的方法，为许多操作系统(如 UNIX)所使用，能实现系统程序的绝大多数命令。这样，命令解释程序丝毫不必理解命令；它只要用命令来识别文件，以装入内存并执行。因此，UNIX 删除文件的命令

rm G

会搜索名为 rm 的文件，将该文件装入内存，并用参数 G 来执行。与 rm 命令相关联的功能应完全由文件 rm 的代码所定义。这样，程序员能通过创建合适名称的新文件，以轻松地向系统增加新命令。这种命令解释程序可能很小，在增加新命令时不必改变。

这种命令解释程序的设计方法有些问题。第一，由于执行命令的代码是分开的系统程序，因此操作系统必须提供一种从命令解释程序向系统程序传递参数的机制。这种任务常常比较笨拙，这是因为命令解释程序和系统程序可能不同时在内存中，并且参数列表可能较为昂贵。再者，装入程序和执行程序要比简单地转到当前程序的另一段代码要慢。

另一个问题是参数的解释通常取决于设计系统程序的程序员。因此这些程序是在不同时期由不同程序员编写的，跨程序提供的参数对用户来说显得相似但可能不一致。

因此，绝大多数用户所看到的操作系统是由系统程序而不是实际系统调用定义的。想一下个人计算机的使用。当计算机运行微软公司的 Windows 操作系统时，你可能看到命令行 MS-DOS 外壳或者是图形窗口－菜单接口。两者都使用同一组系统调用，但是系统调用看起来不同且其动作也不同。因而，用户观点与真实系统结构实际上还是有距离的。因此，设计一个有用的友好的用户接口并不是操作系统的直接功能。本书将集中在为用户程序提供足够服务的基本问题上。从操作系统的角度而言，并不区分用户程序和系统程序。

3.5 系统结构

像现代操作系统这样庞大而复杂的系统为了能正常工作并能容易修改，必须认真设计。通常方法是将这一任务分成小模块而不只是一个单块系统。每个这样的模块都应该是定义明确的系统部分，且具有定义明确的输入、输出和功能。在 3.1 节已简要讨论了操作系统的常用模块。本节讨论这些模块如何相互连接起来以组成内核。

3.5.1 简单结构

许多商业系统没有明确定义的结构。通常，这些操作系统以较小、简单且功能有限的系统形式启动，但后来渐渐超过了其原来的范围。MS-DOS 就是一个这样的操作系统。它最初是由几个人设计和实现的，当时这些人并没有想到它会如此受欢迎。由于运行所用的硬件有限，它被编写成利用最小的空间提供最多的功能，因此它并没有被仔细划分成模块。图 3.6 显示了它的结构。

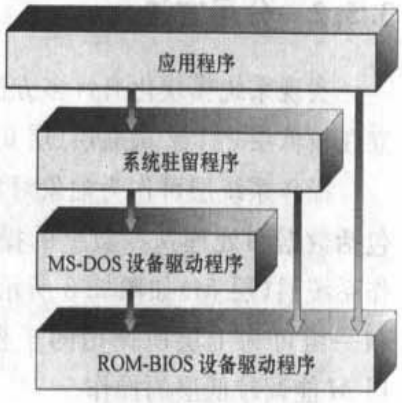


图 3.6 MS-DOS 层次结构

UNIX 是另一个系统，最初受到硬件功能的限制。它由两个独立部分组成：内核和系统程序。内核进一步划分为一系列接口和设备驱动程序，这些年随着 UNIX 的发展这些程序被不断地增加和扩展进来。可以将传统 UNIX 操作系统按分层来研究。如图 3.7 所示，物理硬件之上和系统调用接口之下的所有部分作为内核。内核通过系统调用来提供文件系统、CPU 调度、

内存管理和其他操作系统功能。总的来说，大多数的功能都结合放在这一层。这使得UNIX难以增强，这是因为一部分的改动可能会对其他部分造成不利的影响。

图 3.7 UNIX 系统结构

系统调用定义了UNIX的API；常用的系统程序集合定义了用户接口。程序员和用户接口定义了内核所必须支持的关联。

新版UNIX设计可以充分地使用高级硬件。如果有适当硬件支持，操作系统可以分成比原来MS-DOS或UNIX系统所允许的更小和更合适的模块。这样，操作系统能支持对计算机和使用计算机的应用程序的更多的控制。实现人员能更加自由地改变系统的内部工作和创建模块操作系统。采用自顶向下方法，可先确定总的功能和特征，再划分成构件。这种划分允许程序员隐藏信息；他们在保证子程序外部接口不变和子程序本身执行其宣称任务的前提之下，在认为合适的时候能自由地实现低层子程序。

3.5.2 分层方法

实现系统模块化有许多方法。一种方法是**分层法**，即操作系统分成若干层(级)，每层建立在较低层之上。最底层(层0)是硬件；最高层(层 N)是用户接口。

操作系统层可作为抽象对象来实现，该对象包括数据和处理这些数据的操作。一个典型操作系统层(层 M)如图3.8所示。它由数据结构和一组可为上层所调用的子程序集合所组成。层 M 能调较低层的操作。

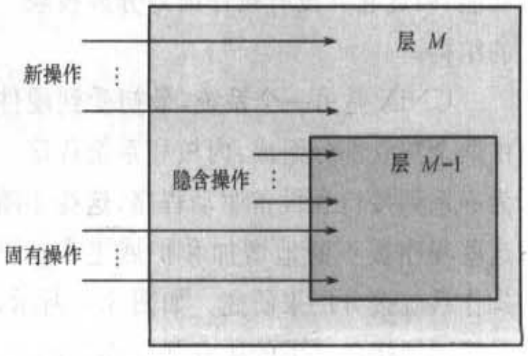


图 3.8 一种操作系统层次

分层法的主要优点是**模块化**。选择了分层，这样每层只能利用较低层的功能(或操作)和服务。这种方法简化了调试和系统验证。第一层能先调试而不需要考虑系统其他部分，因为根据

定义它只使用了基本的硬件(通常认为是正确的)来实现其功能。在第一层调试之后,可以认为它已正确运作,这样可以调试第二层,如此进行。如果在调试特定层时发现了错误,那么错误必然在那一层,这是因为其下的低层都已调试好了。因此,系统分层简化了系统的设计和实现。

每层都是利用较低层所提供的操作来实现的。该层不必知道如何实现这些操作;它只需要知道这些操作做什么。因此,每层为较高层隐藏了一定的数据结构、操作和硬件的存在。

分层法的主要困难涉及到对层的仔细认真的定义,这是因为一层只能使用其下的较低层。例如,虚拟内存算法所使用的磁盘空间的设备驱动程序必须位于内存管理子程序之下,因为内存管理需要能使用磁盘空间。

其他要求并不这么明显。备份存储驱动程序通常在CPU调用程序之上,因为该驱动程序需要等待I/O完成,在这段时间内可以重新调度CPU。不过,对于大型系统,CPU调度程序会有更多关于适合在内存中的活动进程的更多信息。因此,这些信息需要换入和换出内存,从而要求备份存储驱动器程序位于CPU调度程序之下。

分层法实现的最后一个问题是与其他方法相比其效率稍差。例如,当一个用户程序执行I/O操作时,它执行系统调用,并陷入到I/O层;I/O层会调用内存管理层;内存管理层接着调用CPU调度层;最后传递给硬件。在每一层,参数可能被修改,数据可能要传递,等等。每层都为系统调用增加了额外开销;最终结果是系统调用比在非分层系统上要执行更长的时间。

这些限制在近年来引起了分层法的小倒退。现在使用数量更少而功能更多的分层设计,提供了绝大多数模块化代码的优点,同时避免了分层定义和交互的困难问题。例如,OS/2是MS-DOS的后代,增加了多任务和双模式操作及其他一些新特征。由于OS/2设计所增加的复杂性和更为有力的硬件,OS/2系统按更深化的分层形式实现。比较一个MS-DOS结构(图3.6)和OS/2层次结构(图3.9);从系统设计和实现的观点来看,OS/2都有优

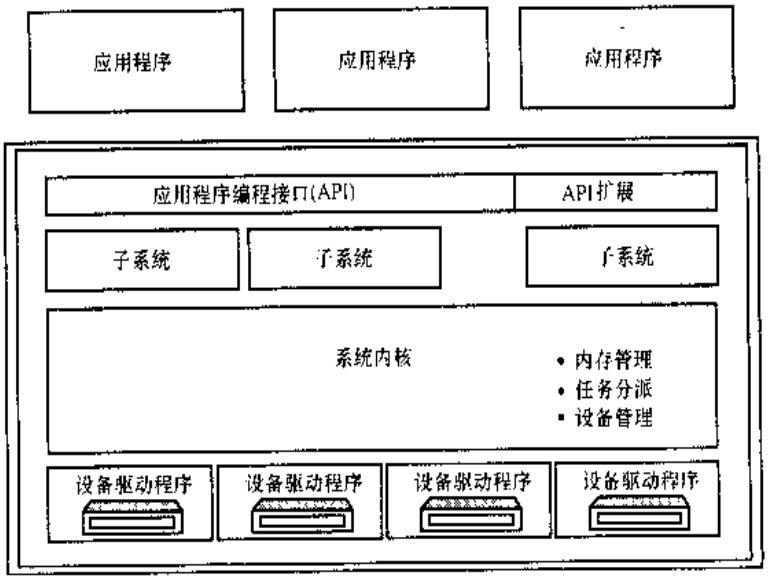


图3.9 OS/2层次结构

势。例如，用户对低层设备的直接访问是不允许的，这使得操作系统能更好地控制硬件，且对每个用户程序所使用的资源有更好的了解。

作为另一个例子，考虑一下 Windows NT 的历史。第一版具有高度的面向分层组织；不过，其性能相对来讲要比 Windows 95 差。Windows NT 4.0 通过将有些层从用户空间移到内核空间和更紧密集成这些层而部分地改善了性能问题。

3.5.3 微内核

随着 UNIX 操作系统的扩充，内核变得更大且更难管理。在 20 世纪 80 年代中期，卡内基－梅隆大学的研究人员开发了一个称为 Mach 的操作系统，该系统采用**微内核**(microkernel)方法来模块化内核。这种方法将所有非基本部分从内核中移走，并将它们当做系统级程序和用户级程序来实现，用这种方法构建操作系统。这一结果是更小的内核。关于哪些服务应保留在内核内，而哪些服务应在用户空间内实现，并没有定论。不过，微内核通常提供最小的进程和内存管理以及通信功能。

微内核的主要功能是提供客户程序和运行在用户空间的各种服务之间进行通信的能力。通信以消息传递形式提供，参见 3.3.5 小节。例如，如果客户程序希望访问一个文件，那么它必须与文件服务器进行交互。客户程序和服务决不会直接交互，而是通过与微内核的消息交换来通信。

微内核方法的好处包括便于扩充操作系统。所有新服务被增加到用户空间中，因而并不需要修改内核。当内核确实需要修改时，所做的改变也会很小，因为微内核本身很小。这样的操作系统很容易从一种硬件平台设计移植到另一种硬件平台设计。由于绝大多数服务是作为用户进程而不是作为内核进程来运行的，因此微内核也就提供了更好的安全性和可靠性。如果一个服务失败，那么操作系统的其他部分并不受影响。

许多现代操作系统使用了微内核方法。Tru64 UNIX(前身是 Digital UNIX)向用户提供了 UNIX 接口，但是它是用 Mach 内核来实现的。Mach 内核将 UNIX 系统调用映射成对适当用户层服务的消息。Apple MacOS X 服务器操作系统也是基于 Mach 内核的。

实时操作系统 QNX 也是基于微内核而设计的。QNX 微内核提供了消息传递和进程调度的服务。它也处理低层网络通信和硬件中断。QNX 所有的其他服务是通过标准进程(以用户模式运行在内核之外)提供的。

Windows NT 采用了混合结构。本书已经提及 Windows NT 体系结构部分地采用了分层法。Windows NT 设计成可以运行各种应用程序，包括 Win32(本地 Windows 应用程序)、OS/2 和 POSIX。它为每种类型的应用程序提供了一个运行在用户空间的服务器。每种应用类型的客户程序也运行在用户空间。内核协调客户应用程序和应用程序服务器之间的消息传递。Windows NT 的客户－服务器结构如图 3.10 所示。

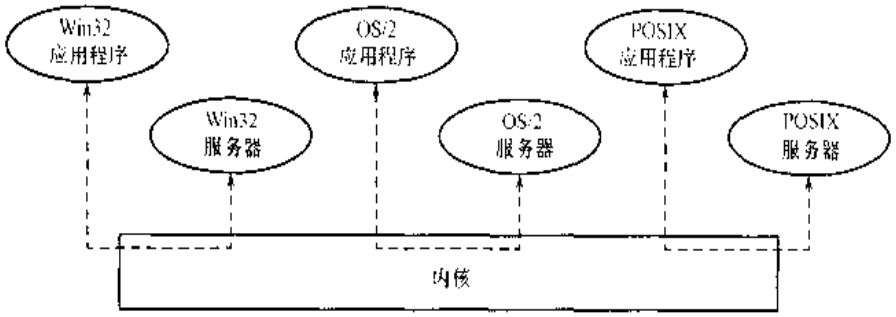


图 3.10 Windows NT 客户—服务器结构

3.6 虚 拟 机

从概念上说，计算机系统是由层次组成的。硬件是这种系统的最低层。运行在上一层的内核利用硬件指令来创建一组系统调用，以供外层使用。内核之上的系统程序因此能使用系统调用或硬件指令，系统程序在许多情况下并不区分这两者。虽然访问方式不同，但是它们都提供供程序使用的功能以创建更高级的功能。因而，系统程序将硬件和系统调用作为同一层来对待。

有的系统进一步扩展了这一策略，允许系统程序能很容易地被应用程序所调用。与以前一样，虽然系统程序比其他子程序的层次要高，但是应用程序还是可以将它们的一切下层当成硬件的一部分而看做一个层次整体。这种分层方法自然而逻辑地延伸为**虚拟机**概念。IBM 系统的 VM 操作系统是虚拟机概念的最好例子，因为 IBM 公司首创了这方面的工作。

通过利用 CPU 调度（第六章）和虚拟内存技术（第十章），操作系统能创建一种幻觉，以至于进程认为有自己的处理器和自己的（虚拟）内存。当然，进程通常还有其他特征（如系统调用和文件系统），这些是纯硬件所不能单独提供的。而虚拟机方法除了提供了与基本硬件*相同*的接口之外并不提供任何额外功能。每个进程都有一个与基本计算机一样的（虚拟）拷贝（图 3.11）。

物理计算机共享资源以创建虚拟机。CPU 调度能共享出 CPU 以造成一种每个用户都有自己的处理器的感觉。假脱机和文件系统能提供虚拟读卡机和虚拟行式打印机。一个普通的用户分时终端提供虚拟机操作员终端的功能。

虚拟机方法的主要困难与磁盘系统有关。假设物理机器有三个磁盘驱动器但是要提供七个虚拟机。显然，它不能为每个虚拟机分配一个磁盘驱动器。注意，虚拟机软件本身需要一定的实际磁盘空间，以提供虚拟内存。解决方法是提供虚拟磁盘，它们除了大小外在各方面都相同，IBM 公司的 VM 操作系统称之为*小型磁盘*。系统通过在物理磁盘上为小型磁盘分配所需要的磁道数来实现每个小型磁盘。显然，所有小型磁盘的大小的总和必须比可用的物理磁盘空间要小。

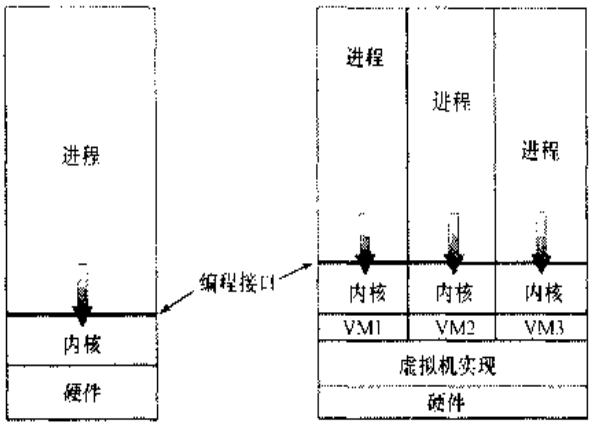


图 3.11 系统模型

因此用户有自己的虚拟机。他们能运行原来机器所具有的任何操作系统或软件包。对 IBM VM 系统而言，用户通常运行 CMS——一种单用户交互式操作系统。虚拟机软件主要与一个物理机器上多道运行多个虚拟机有关，但并不需要考虑任何用户支持软件。这种安排可提供将多用户交互式系统设计分为两个更小部分问题的一种有效的划分。

3.6.1 实现

虽然虚拟机概念有用，但是实现困难。提供与底层机器完全一样的拷贝需要大量的工作。记住底层机器有两种模式：用户模式和监控模式。虚拟机软件能运行在监控模式，因为它是操作系统。虚拟机本身只能运行在用户模式。正如物理机器有两种模式一样，虚拟机也有。因此，必须有虚拟用户模式和虚拟监控模式，这两种模式都运行在物理用户模式。在真实机器上引起从用户模式到监控模式转换的动作（如系统调用或试图执行特许指令）也必须在虚拟机上引起从虚拟用户模式到虚拟监控模式的转换。

这种转换通常能相当容易地完成。例如，当一个以虚拟用户模式在虚拟机上运行的程序执行系统调用时，它会在真实机器上引起一个到虚拟机监控器的转换。当虚拟机监控器获得控制时，它能改变虚拟机的寄存器内容和程序计数器以模拟系统调用的效果。接着它能重新启动虚拟机，注意它现在是执行在虚拟监控模式下。例如，如果虚拟机接着试图从其虚拟读卡机上读，它会执行一个特许的 I/O 指令。因为虚拟机是运行在物理用户模式下，所以这种指令会陷入到虚拟机监控器。虚拟机监控器接着必须模拟 I/O 指令的效果。首先，它找到完成虚拟读卡机的脱机文件。接着，它将对虚拟读卡机的读转换成对脱机操作磁盘文件的读，并将下一个虚拟“卡映像”转换成虚拟机的虚拟内存。最后，它能重新启动虚拟机。虚拟机的状态已改变，好像 I/O 指令已经用真实读卡机在真实的机器下按真实监控模式执行过一样。

当然，主要的差别是时间。虽然真实的 I/O 可能需要 100 ms，但是虚拟 I/O 可能需要更

少的时间(因为脱机操作)或更多时间(因为解释执行)。另外,CPU 在多个虚拟机间被多道编程,也会使得虚拟机以不可预计的方式慢下来。在极端情况下,可能需要模拟所有指令以提供真正的虚拟机。VM 能在 IBM 公司机器上工作,这是因为虚拟机的通常指令能直接在硬件上执行。只有特许指令(主要是用户 I/O 需要)必须被模拟,因而执行更慢。

3.6.2 优点

使用虚拟机有两个主要优点。第一,通过完全保护系统资源,虚拟机提供了一个坚实的安全层。第二,虚拟机允许进行系统开发而不必中断正常的系统操作。

每个虚拟机完全与其他虚拟机相隔离,由于各种系统资源完全被保护,就不存在安全问题。例如,从因特网下载的不信任的应用程序可以运行在独立的虚拟机上。这种环境的一个缺点是没有直接资源共享。提供共享有两种实现方法。第一,可以通过共享小型磁盘。这种方案模拟了共享物理磁盘,但可通过软件实现。采用这种方法可共享文件。第二,可以通过定义一个虚拟机的网络,每台虚拟机通过虚拟通信网络来发送信息。同样,该网络是按物理通信网络来模拟的,但是通过软件实现。

这样的虚拟机系统是用于研究和开发操作系统的很好工具。通常,修改操作系统是一项艰难的任务。因为操作系统程序庞大且复杂,所以改变一处可能会在另一处产生隐匿的错误。操作系统的威力使得这种情况尤为危险。由于操作系统在监控模式下执行,一个指针的错误改变可能会引起足以破坏整个文件系统的错误。因此,有必要仔细检查操作系统的所有改变。

不过,操作系统运行并控制整个机器。因此,必须停止当前系统,暂停其使用以便进行改变和测试。这个周期通常称为**系统开发时间**。由于在这段时间内系统不能被用户使用,因此系统开发时间通常安排在晚上和周末进行,这时系统负荷小。

虚拟机系统能基本上取消这个问题。系统程序员有自己的虚拟机,系统开发可在虚拟机而不是真实的物理机器上进行。正常系统操作无需中断以开发系统。虽然有这些优点,但是这一技术直到近来才得以改进。

作为解决系统兼容性问题的一种方法,虚拟机的应用程度不断增加。例如,许多应用程序可以在运行微软公司 Windows 的基于 Intel CPU 的系统上执行。计算机厂商如 Sun 微系统公司使用其他更快的处理器,但是也希望其客户能运行这些 Windows 应用程序。解决方案是在其本身处理器之上创建虚拟 Intel 机。Windows 程序可在这种环境下运行,其 Intel 指令被转换成本机指令。微软公司 Windows 也运行在这一虚拟机上,这样应用程序可像平常一样执行系统调用。最后结果是一个程序看起来像运行在基于 Intel 的系统上,但是实际上运行在一个不同的处理器上。如果处理器足够快,那么 Windows 程序也会运行得快,尽管每个指令都要转换成若干个本地指令以便执行。类似地,基于 PowerPC 的 Apple Macintosh 包括一个 Motorola 68000 虚拟机,以允许那些为旧的基于 68000 的 Macintosh 而

编写的二进制代码能得以执行。不过，所模拟的机器越复杂，创建一个精确虚拟机也就越困难，且虚拟机运行得也越慢。

最近的一个例子是 Linux 操作系统的成长，虚拟机现在允许 Windows 应用程序运行在基于 Linux 的计算机上。该虚拟机可运行 Windows 应用程序和 Windows 操作系统。

Java 的一个重要特点是它能运行在虚拟机上，因此允许 Java 程序能在任何具有 Java 虚拟机的计算机系统上运行。

3.6.3 Java

Java 是由 Sun 微系统公司于 1995 年底推出的一种很受欢迎的面向对象语言。除了其语言规范和大量 API 库，Java 还提供了 **Java 虚拟机(JVM)**的规范。

Java 对象用 class 结构来描述；Java 程序由一个或多个类组成。对于每个 Java 类，Java 编译器会生成与结构无关的**字节代码**(bytecode)输出文件(.class)，它可运行在任何 JVM 实现上。

JVM 是一个抽象计算机的规范。JVM 由**类加载器**、**类验证器**和 Java 解释器组成，来执行与结构无关的字节代码。类加载器装入 Java 程序和 Java API 的.class 文件，以便为 Java 解释器所执行。在装入类后，验证器会检查类文件是否为有效的 Java 字节代码，且无堆栈的溢出和下溢。它也确保字节代码不进行指针计算，因为这可能会潜在地提供非法内存访问。如果类通过验证，那么就可为 Java 解释器所执行。JVM 通过执行**垃圾收集**(garbage collection，收回对象不再使用的内存并返回给系统)来自动管理内存。为了提高虚拟机中 Java 程序的性能，许多研究集中在垃圾收集算法上。

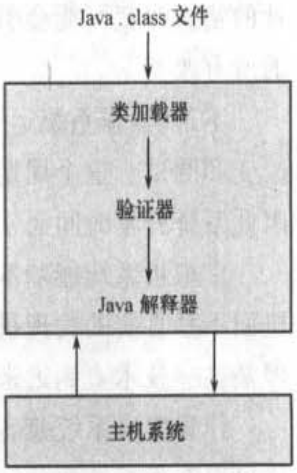


图 3.12 Java 虚拟机

Java 解释程序可能是软件模块，一次只能解释一个字节代码，或者可能是一个 **JIT(just-in-time)**编译器用来将结构无关的字节代码转换成主机的本地机器语言。绝大多数 JVM 实现使用了 JIT 编译器，以增强性能。在其他情况下，解释程序可用硬件来实现，以自然地执行 Java 字节代码。图 3.12 显示了 JVM 的结构。

JVM 有助于开发与结构无关的可移植程序。JVM 的实现对每个系统如 Windows 或 UNIX 来说是明确的，它以标准方式将系统抽象为 Java 程序，提供一个干净的与结构无关的接口。这种接口允许.class文件能在任何根据规范实现了 JVM 的系统上运行。

Java 利用了虚拟机所实现的完整环境。其虚拟机设计提供了一个安全的、高效的、面向对象的、可移植的且与结构无关的平台来运行 Java 程序。

3.7 系统设计与实现

本节讨论一些设计和实现系统时所遇到的问题。虽然对这些设计问题没有完整的解决方案，但是有些方法还是很成功的。

3.7.1 设计目标

系统设计的第一个问题是定义系统的目标和规格。在最高层，系统设计受到硬件选择和系统类型的影响：批处理、分时、单用户、多用户、分布式、实时或通用目标。

除了最高设计层，这些要求可能难以描述。需求可分为两个基本类：*用户*目标和*系统*目标。

用户要求一些明显的系统特性：系统应该方便和容易使用、容易学习、可靠、安全和快速。当然，这些规格说明对于系统设计并不特别有用，因为没有实现这些目标的通用协议。

设计、创建、维护和操作系统的有关人员可以定义另一组类似要求：操作系统应该容易设计、实现和维护；它也应该灵活、可靠、没有错误且高效。同样，这些要求是含糊不清的，并没有通用解决方案。

关于定义操作系统的要求，没有一个惟一的解决方案。多种类型的系统说明了不同要求能形成对不同环境的各种解决方案。例如，MS-DOS 的要求，用于微型计算机的单用户系统，与 MVS 的要求，用于 IBM 大型机的多用户、多访问操作系统，存在实质性差别。

3.7.2 机制与策略

操作系统的规格说明和设计属于高度创造性工作。虽然没有教科书告诉你如何去做，但是一般的**软件工程**原理还是适用的，尤其是适用于操作系统。

一个重要原理是*策略*和*机制*的区分。**机制**(mechanism)决定了*如何来做*；**策略**决定*做什么*。例如，定时器结构(2.5 节)是一种确保 CPU 保护的机制，但是对于特定用户将定时器设置成多长时间是个策略问题。

策略和机制的区分对于灵活性来说很重要。策略可能会随地点或时间而有所改变。在最坏情况下，每次策略改变都可能需要底层机制的改变。系统更需要通用机制。这种策略的改变只需要重定义一些系统参数。例如，对于一个计算机系统，如果一个策略决定是 I/O 密集型程序应该比 CPU 密集型程序有更高的优先级，如果机制是正确分开的且不依赖于策略，那么可以在其他计算机系统上很容易地制定相反的策略。

通过实现一组基本的原始的构造块，基于微内核的操作系统充分利用了机制与策略的区分。这些块几乎与策略无关，允许通过用户创建的内核模块或用户程序本身来增加更高级的机制和策略。另一种极端如 Apple Macintosh 操作系统，该系统完全对机制和策略在

系统内编码，以形成统一的系统风格和感受。所有应用程序都有类似的接口，因为接口本身已在内核中构造。

关于所有资源分配和调度的问题，必须做出策略决定。只要问题是*如何*而不是*什么*，这就必须要由机制决定。

3.7.3 实现

在设计操作系统之后，就必须实现它。传统地，操作系统是用汇编语言来编写的。不过，操作系统现在都是用高级语言如C或C++来编写的。

不是用汇编语言编写的第一个系统可能是用于Burroughs计算机的主控程序(MCP)。MCP是采用一种ALGOL语言编写的，在MIT开发的MULTICS主要是用PL/1来编写的。用于Prime计算机的Primos操作系统是用Fortran语言编写的。UNIX操作系统、OS/2和Windows NT主要是用C编写的。在原来UNIX中，只有900行代码是用汇编语言来编写的，这些主要用于调度程序和设备驱动程序。

使用高级语言或至少是系统实现语言来实现操作系统，可以得到与用高级语言来编写应用程序同样的优点：代码编写更快，更为紧凑，更容易理解和调试。另外，编译技术的改进使得只要通过重新编译就可改善整个操作系统的生成代码。最后，如果用高级语言来编写，从一个硬件到另一个硬件，操作系统更容易**移植**。例如，MS-DOS是用Intel 8088汇编语言编写的。因而，这只能用于Intel类型的CPU。

另一方面，UNIX操作系统，主要是用C来编写的，可用于许多不同的CPU上，如Intel 80x86、Pentium、Motorola 680x0、Ultra SPARC、Compaq Alpha和MIPS RX000等。

对于用高级语言来实现操作系统的反对者，其宣称的主要缺点是减低了速度和增加了存储要求。虽然汇编语言高级程序员能编写更高效更小的子程序，但是现代编译器能对大型程序进行复杂分析并采用高级优化技术以生成优良代码。现代处理器都有很深的流水线和多个功能单元块，它们能处理复杂相关性，这些是人类的有限能力所难以自己处理的，因而可以跟踪细节。

与其他系统一样，操作系统的重要性能改善很可能是由于更好的数据结构和算法，而不是由于优秀的汇编语言代码。另外，虽然操作系统很大，但是只有一小部分代码对于高性能来说是很关键的；内存管理器和CPU调度程序可能是最为关键的子程序。在系统编写完并能正确工作之后，可以找出瓶颈子程序，并用相应的汇编语言子程序来替代。

为了识别瓶颈，必须要能监视系统性能。必须增加代码以计算并显示系统行为的测量度。对于有的系统，操作系统通过生成系统行为的跟踪列表来执行这一任务。所有相关事件的时间和重要参数都记录下来，并写到文件。之后，分析程序能处理日志文件，以决定系统性能，并识别瓶颈和低效率。这些同样的跟踪能作为所建议改进系统模拟的输入。跟踪也有助于帮助人们找出操作系统行为的错误。

另一种方法是实时计算并显示性能测量度。例如，定时器可以触发一个子程序，以存储当前指令指针值。这一结果可作为程序经常使用位置的统计图。这种方法允许系统操作员熟悉系统行为，并实时修改系统操作。

3.8 系统生成

可以为某处的某台机器专门设计、编码和实现操作系统。不过，操作系统通常设计成能运行在一类计算机上，这些计算机位于不同的场所，并具有不同的外设配置。对于某个特定的计算机场所，必须要配置和生成系统，这一过程有时称为**系统生成**(system generation，SYSGEN)。

操作系统通常分布在磁盘或磁带上。为了生成系统，使用一个特殊程序。SYSGEN 程序从给定文件中读取，或询问系统操作员有关硬件系统的特定配置信息，或直接检测硬件以决定有什么部件。下面几类信息必须要确定下来。

- 使用什么 CPU？安装什么选项(扩展指令集，浮点运算操作等)？对于多 CPU 系统，必须描述每个 CPU。
- 有多少可用内存？有的系统通过亲自对内存位置一个个地访问直到出现"非法地址"故障的方法来确定这一值。该过程定义了最后合法地址和可用内存的数量。
- 有什么可用设备？系统需要知道如何访问这些设备(设备号码)、设备中断号、设备类型和模型以及任何特别的设备特点。
- 需要什么操作系统选项或使用什么参数值？这些选项或值可能包括需要使用多少和多大的缓冲区，需要什么类型的 CPU 调度算法，所支持进程的最大数量是多少，等等。

这些信息确定之后，可以有多种方法来使用。对一种极端情况，系统管理员可用这些信息来修改操作系统的源代码拷贝。接着完全编译操作系统。数据说明、初始化、常量和其他一些条件编译，生成了专门适用于所描述系统的操作系统的输出目标代码。

对于另外一种稍微欠定制的层，系统描述可用来创建表，并从预先编译过的库中选择模块。这些模块连接起来，以形成所生成的操作系统。选择允许库包括所有支持 I/O 设备的驱动程序，但是只有所需要的才连接到操作系统。因为系统没有重编译，所以系统生成较快，但是所生成的系统可能过分通用。

对于另外一种极端情况，可以构造完全由表驱动的系统。所有代码都是系统的组成部分，选择发生在执行时而不是在编译或连接时。系统生成只是创建适当的表以描述系统。绝大多数现代操作系统按这种方式来构造。Solaris 在系统安装时(有时在引导时)执行一些系统配置检测。系统管理员可使用配置文件来微调系统的变化，但是硬件支持是自动地由内核配置的。同样，Windows 2000 在安装或引导时也不需要人工配置。在回答了有关磁盘分配和网络配置的基本问题之后，安装程序能自动地检测系统硬件，并安装正确生成的操作

系统。

这些方法的主要差别是所生成系统的大小和通用性，和因硬件配置变化而进行修改的方便性。考虑一下修改系统以支持新需求图形终端或另一个磁盘驱动器的代价。当然，与该代价相对的是这些改变的频繁程度。

在生成操作系统之后，它必须要为硬件所使用。但是硬件如何知道内核在哪里，或者如何装入该内核？装入内核以启动计算机的过程称为**引导**系统。绝大多数计算机系统都有一小块代码，保存在 ROM 中，它称为**引导程序或引导装载程序**。这段代码能定位内核，将它装入该内存，开始执行。有的计算机系统，如个人计算机，采用两步走方式：一个简单引导装载程序从磁盘上取一个更复杂的引导程序，而后者再装入内核。引导系统在 14.3.2 小节中和附录 A 中讨论。

3.9 小 结

操作系统提供若干服务。在最低层，系统调用允许运行程序直接向操作系统发出请求。在较高层，命令解释程序或外壳提供了一种机制，以便用户不必编写程序就能发出请求。命令可以来自文件（在批处理模式执行期间），或者直接来自键盘输入（在交互式模式或分时模式时）。系统程序用来满足许多常用用户请求。

请求类型随请求级别而变化。系统调用级别必须提供基本功能，如进程控制、文件和设备管理。由命令解释程序或系统程序来完成的高级别请求需要转换成一系列的系统调用。系统服务可划分成许多类别：程序控制、状态请求和 I/O 请求。程序出错可作为对服务的一种隐式请求。

在定义了系统服务之后，就可开发操作系统的结构。需要用各种表格来记录一些信息，这些信息定义了计算机系统的状态和系统作业的状态。

设计一个新操作系统是个重大任务。在设计开始之前，要定义好系统目标。它们形成选择各种必要算法和策略的基础。

由于操作系统庞大，所以模块化很重要。按一系列层或采用微内核来设计系统被认为是比较好的技术。虚拟机概念采用了分层方法，并将操作系统内核和硬件都作为硬件来考虑。其他操作系统甚至可以加载在这一虚拟机之上。

实现 JVM 的任何操作系统都能运行所有 Java 程序，因为 JVM 为 Java 程序抽象化了底层系统，以提供与结构无关接口。

在整个操作系统设计周期中，必须仔细区分策略决定和实现细节（或机制）。在后面需要修改策略时，这种区分允许最大限度的灵活性。

现在操作系统几乎都是用系统实现语言或高级语言来编写的。这一特征改善了操作系统的实现、维护和可移植性。为特定机器配置创建操作系统时，必须进行系统生成。

习 题 三

3.1 操作系统关于进程管理的五个主要活动是什么?

3.2 操作系统关于内存管理的三个主要活动是什么?

3.3 操作系统关于二级存储管理的一个主要活动是什么?

3.4 操作系统关于文件管理的五个主要活动是什么?

3.5 命令解释器的用途是什么?为什么它经常与内核是分开的?

3.6 列出操作系统提供的五项服务。说明每种服务如何给用户提供便利。说明在哪些情况下用户级程序不能够提供这些服务。

3.7 系统调用的用途是什么?

3.8 用C或C++编写一个使用系统调用从一个文件中读入数据并复制数据到另外一个文件中的程序。第3.3节描述了一个这样的程序。

3.9 为什么Java提供了从Java程序调用由C或C++编写的本地方法的能力?举出一个本地方法有用的例子。

3.10 系统程序的用途是什么?

3.11 系统设计采用层次化设计的主要优点是什么?

3.12 系统设计采用微内核方法的主要优点是什么?

3.13 操作系统设计员采用虚拟机结构的主要优点是什么?对用户来说主要有什么好处?

3.14 为什么说一个JIT(just-in-time)编译器对执行一个Java程序是有用的?

3.15 为什么机制与策略分离是个令人满意的原则?

3.16 实验性的综合操作系统在内核里有一个汇编器。为了优化系统调用的性能,内核通过在内核空间内汇编程序来缩短系统调用在内核中必须经过的途径。这是一种与分层设计相对立的方法,经过内核的途径在这种设计中被延伸了,使操作系统的构建更加容易。分别从支持和反对的角度来讨论这种综合设计方式对内核设计和系统性能优化的影响。

推荐读物

Dijkstra[1968]提倡操作系统设计的层次化方法。Brinch Hansen[1970]是一个操作系统应作为一个内核,在其上能建立更完整的系统的观点的早期支持者。有关QNX微内核的信息可以在http://www.qnx.com中找到。

第一个提供了虚拟机的操作系统是在IBM 360/67上的CP/67。商业版IBM VM/370操作系统是源自CP/67的版本。关于虚拟机的一般性论述有Hendricks和Hartmann[1979]、MacKinnon[1979]和Schultz[1988]。一个允许Windows应用程序能在Linux上运行的虚拟机请参见http://www.vmware.com。Cheung和Loong[1995]研究了操作系统从微内核到可扩展系统的结构问题。Back等[2000]论述了Java操作系统的设计。

MS-DOS 3.1 版由 Microsoft[1988] 描述。Windows NT 由 Solomon[1998] 描述。Apple Macintosh 操作系统的描述在 Apple[1987]。Berkeley UNIX (BSD) 的描述见 McKusick 等[1996]。标准 AT&T UNIX 系统 V 由 Earhart[1986] 描述。Iacobucci[1988] 给出了一个好的关于 OS/2 的描述。Accetta 等[1986] 介绍了 Mach，而 AIX 则发表在 Loucks 和 Sauer[1987]。Massalin 和 Pu[1989] 论述了试验性的综合操作系统。Bar[2000] 详细介绍了 Linux 内核。

Gosling 等[1996] 和 Lindholm 及 Yellin[1998] 分别描述了 Java 语言和 Java 虚拟机的规范。Venners[1998] 和 Meyer 及 Downing[1997] 描述了 Java 虚拟机的内部工作机制。Jones 和 Lin[1996] 给出了一个完整的垃圾回收算法。更多的有关 Java 的信息见 http://www.javasoft.com。

第二部分　进程管理

进程可以看做是正在执行的程序。进程需要一定的资源(如CPU时间、内存、文件和I/O设备)来完成其任务。这些资源在创建进程或进程执行时分配。

进程在大多数系统中是工作单元。这样的系统由一组进程组成:操作系统进程执行系统代码,用户进程执行用户代码。所有这些进程可以并发执行。

虽然从传统意义上讲,进程运行时只包含一个单独控制线程,但目前大多数现代操作系统支持多线程进程。

操作系统除了负责进程和线程管理外,还负责以下活动:用户进程与系统进程的创建与删除,进程调度,提供进程同步机制、进程通信机制与进程死锁处理机制。

第四章 进 程

早期的计算机系统只允许一次执行一个程序。这种程序对系统有完全的控制，能访问所有系统资源。现代计算机系统允许将多个程序调入内存并发执行。这一进步要求对各种程序提供更严格的控制和更好的划分。这些需求产生了**进程**的概念，即执行中的程序。进程是现代分时系统的工作单元。

操作系统越复杂，就越能为用户做更多的事。虽然操作系统的主要关注点是执行用户程序，但是它也需要照顾各种系统任务（放在内核之外会更好）。因此，系统由一组进程组成：操作系统进程执行系统代码，而用户进程执行用户代码。通过（多个）CPU 在进程之间的切换（多路复用），所有这些进程都有可能并发执行，从而操作系统能使计算机更为高效。

4.1 进程概念

这里讨论操作系统的一个障碍是如何称呼所有这些 CPU 的活动。批处理系统执行*作业*，而分时系统使用*用户程序*或*任务*。即使单用户系统如微软公司 Windows 和 Macintosh OS，也能让用户同时执行多个程序：字处理程序、网页游览器和电子邮件包程序。即使用户一次只能执行一个程序，操作系统也需要支持其内部的程序化活动，如内存管理。所有这些活动在许多方面都相似，因此统称它们为*进程*。

在本书中，时常交换使用*作业*与*进程*这两个概念。虽然笔者自己偏爱*进程*，但是许多操作系统的理论和术语是在操作系统的主要活动被称为*作业处理*期间发展起来的。如果因为*进程*取代了*作业*，而简单地避免使用有关*作业*的常用短语（如*作业调度*），则会令人产生误解。

4.1.1 进程

进程是执行中的程序，这是一种非正式的说法。进程不只是程序代码，程序代码有时称为**文本段**。进程还包括当前活动，通过**程序计数器**的值和处理器寄存器的内容来表示。另外，进程通常还包括进程**堆栈段**（包含临时数据，如方法参数、返回地址和局部变量）和**数据段**（包含全局变量）。

这里强调：程序本身不是进程；程序只是*被动*实体，如存储在磁盘上的文件内容，而进程是*活动*实体，它有一个程序计数器用来表示下一个要执行的指令和相关资源集合。

虽然两个进程可以与同一程序相关，但是它们被当做两个独立的执行序列。例如，多个用户可运行电子邮件程序的不同拷贝，或者同一用户能调用编辑器程序的多个拷贝。这些都是独立的进程，虽然文本段相同，但是数据段不同。通常一个进程在运行时也能产生许多进程。本书会在 4.4 节讨论这些问题。

4.1.2 进程状态

进程在执行时会改变状态。进程**状态**部分地由进程的当前活动所定义。每个进程可能处于下列状态之一：

- **新的**：进程正在被创建。
- **运行**：指令正在被执行。
- **等待**：进程等待一定事件的出现（如 I/O 完成或收到某个信号）。
- **就绪**：进程等待被分配给某个处理器。
- **终止**：进程已完成执行。

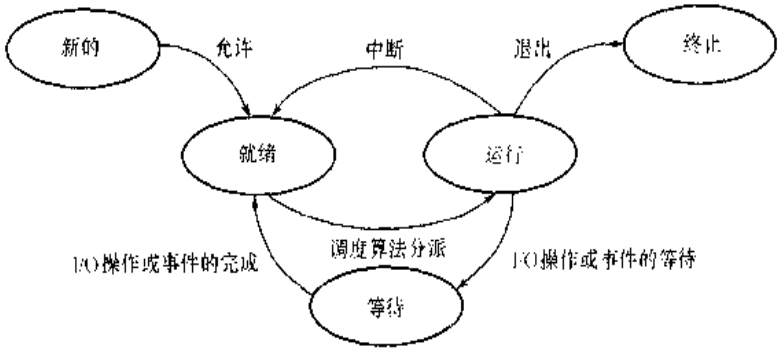


图 4.1 进程状态图

这些状态的名称较随意，且随操作系统的不同而变化。不过，它们所表示的状态可以出现在所有系统上。有的操作系统更为仔细地描述了进程状态。在任何时刻一次只能有一个进程可在任何一个处理器上*运行*，尽管许多进程可能处于*就绪*或*等待*状态。与这些状态相对的状态图如图 4.1 所示。

4.1.3 进程控制块

每个进程在操作系统内用**进程控制块**（process control block，PCB 也称为任务控制块）来表示。图 4.2 给出了一个 PCB 的例子。它包含与特定进程相关的许多信息。

- **进程状态**：状态可包括新的、就绪、运行、等待、停止等。
- **程序计数器**：计数器表示这个进程要执行的下个指令的地址。

图 4.2 进程控制块（PCB）

• CPU **寄存器**：根据计算机体系结构的不同，寄存器的数量和类型也不同。它们包括累加器、索引寄存器、堆栈指针、通用寄存器和其他条件码信息寄存器。与程序计数器一样，这些状态信息在出现中断时也需要被保存，以便进程以后能正确地继续执行(图 4.3)。

• CPU **调度信息**：这类信息包括进程优先级、调度队列的指针和任何其他调度参数。(第六章讨论进程调度。)

• **内存管理信息**：这类信息包括基址寄存器和界限寄存器的值、页表或段表(与操作系统所使用的内存系统有关)(第九章)。

• **计账信息**：这类信息包括 CPU 时间、实际使用时间、时间界限、记账数量、作业或进程数量等。

• I/O **状态信息**：这类信息包括分配给该进程的 I/O 设备列表、打开文件的列表等。

PCB 简单地作为这些信息的仓库，这些信息在进程与进程之间是变化的。

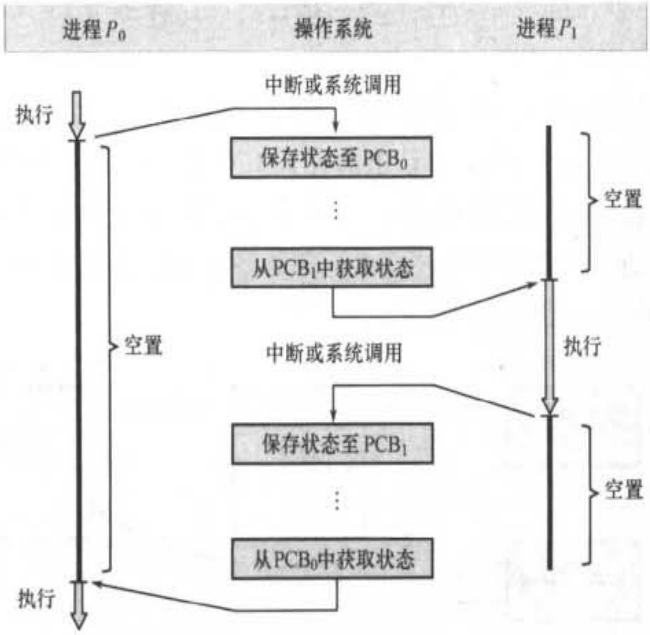


图 4.3 CPU 在进程间的切换图

4.1.4 线程

迄今为止所讨论的进程模型暗示：一个进程是一个只能进行单个执行线程的程序。例如，如果一个进程运行一个字处理器程序，那么只能执行指令的单个线程。这种单一控制线程使得进程一次只能执行一个任务。例如，用户不能在同一进程内，同时输入字符和运行拼写检查。许多现代操作系统扩展了进程概念，以允许进程执行多线程。因此允许进程一次完成多个任务。第五章讨论多线程进程。

4.2 进程调度

多道程序设计的目的是无论任何时候都有进程在运行，从而使 CPU 利用率达到最大化。分时系统的目的是在进程之间频繁切换 CPU，以便用户在程序运行时能与其交互。单处理器系统只能有一个运行进程。如果存在多个进程，那么其他进程需要等待 CPU 空闲并重新调度。

4.2.1 调度队列

进程进入系统时，会被加到**作业队列**中。该队列包括系统中的所有进程。驻留在内存中就绪的等待运行的进程保存在**就绪队列**表上。该队列通常用链表形式来存储，其头节点包括指向链表的第一个和最后一个 PCB 块的指针。可以为每个 PCB 增加一个指针域来指向就绪队列的下一个 PCB。

操作系统也有其他队列。当给进程分配了 CPU 后，它开始执行并最终完成、退出，或被中断，或等待特定的事件发生，如 I/O 请求的完成。对于 I/O 请求的情况，这个请求可能发向专用磁带驱动器或共享设备(如磁盘)。由于系统有许多进程，磁盘可能会忙于其他进程的 I/O 请求，因此该进程可能需要等待磁盘。等待特定 I/O 设备的进程列表称为**设备队列**。每个设备都有自己的设备队列(图 4.4)。

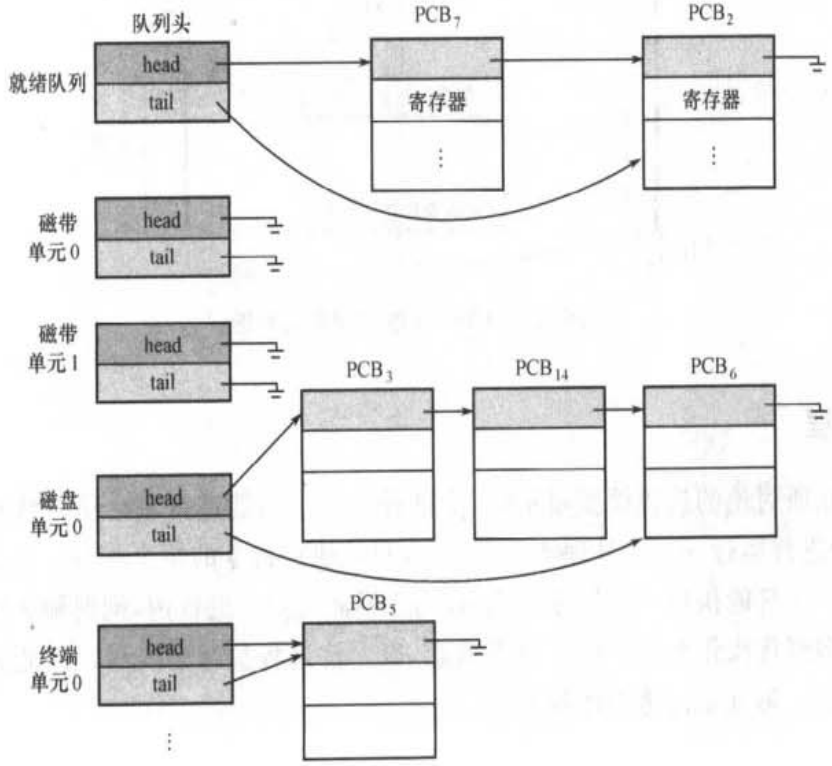


图 4.4 就绪队列和各种 I/O 设备队列

进程调度的常用表示方法是**队列图**，如图 4.5 所示。每个长方形框表示一个队列。有两种队列：就绪队列和一组设备队列。圆形表示为队列服务的资源，箭头表示系统内进程的流向。

新进程开始处于就绪队列。它在就绪队列中等待直到被选中执行(或分派)。当进程分配到 CPU 并执行时，有几种事件可能发生：

- 进程可能发出一个 I/O 请求，并被放到 I/O 队列中。
- 进程可能创建一个新的子进程，并等待其结束。
- 进程可能会由于中断而被强制移出 CPU，并被放回到就绪队列。

对于前两种情况，进程最终从等待状态切换到就绪状态，并放回到就绪队列。进程继续这一循环直到它终止，到时它将从所有队列中被移出，其 PCB 和资源将得以重新分配。

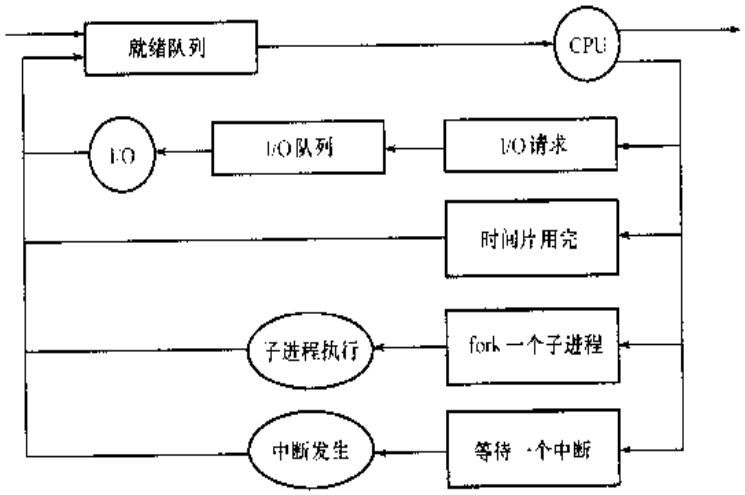


图 4.5 表示进程调度的队列图

4.2.2 调度程序

进程在其生命周期中会在各种调度队列之间迁移。操作系统为了调度的目的，必须按某种方式从这些队列中选择进程。进程选择是由相应的**调度程序**(scheduler)来完成的。

对于批处理系统，通常会提交很多进程，不能够马上都执行。这些进程被放到大容量存储设备上(通常为磁盘)的缓冲池中，保存在那里以便后来执行。**长期调度程序**或**作业调度程序**从该池中选择进程，并将它们装入内存以执行。**短期调度程序**或 CPU **调度程序**从就绪可执行的进程中选择进程，并为其中之一分配 CPU。

这两种调度程序的主要差别是它们执行的频率。短期调度程序必须频繁地为 CPU 选择新进程。进程可能执行数毫秒(ms)就会等待一个 I/O 请求。短期调度程序通常每 100 ms至少执行一次。由于每次执行之间的时间较短，短期调度程序必须要快。如果需要 10 ms来确定执行一个运行 100 ms 的进程，那么 10/(100+10)=9%的 CPU 时间会仅仅用于(或浪费在)调度工作上。

另一方面，长期调度程序执行得并不频繁。在系统内新进程的创建之间可能有数分钟间隔。长期调度程序控制**多道程序设计的程度**，即内存中的进程数量。如果多道程序设计的程度稳定，那么创建进程的平均速度必须等于进程离开系统的平均速度。因此，长期调度程序可能需要在进程离开系统时才被唤起。由于每次执行之间的较长时间间隔，长期调度程序能使用更多时间来选择执行进程。

长期调度程序必须仔细选择。通常，绝大多数进程可分为：I/O 为主或 CPU 为主。I/O **为主的进程**(I/O-bound process)在执行 I/O 方面比执行计算要花费更多的时间。另一方面，CPU **为主的进程**(CPU-bound process)很少产生 I/O 请求，与 I/O 为主的进程相比将更多的时间用在执行计算上。因此，长期调度程序应该选择一个合理的 I/O 为主进程和 CPU 为主进程的**进程组合**。如果所有进程是 I/O 为主的，那么就绪队列将几乎总为空，从而短期调度程序没有什么事情可做。如果所有进程是 CPU 为主的，那么 I/O 等待队列将几乎总为空，从而设备并没有得到使用，因而系统会不平衡。为了得到最好性能，系统需要一个合理的 CPU 为主和 I/O 为主的进程组合。

对于有些系统，长期调度程序可能没有或很小。例如，分时系统如 UNIX 通常没有长期调度程序，只是简单地将所有新进程放在内存中，以供短期调度程序使用。这些系统的稳定性依赖于物理限制(如可用的终端数)或自然人用户的自我调整特性。如果系统性能下降到令人难以接受的程度，那么有的用户就会退出。

有的操作系统如分时系统，可能引入另外的中等程度调度程序。如图 4.6 所示，**中期调度程序**能将进程移出内存(并移出对 CPU 的激烈竞争)，因此降低多道程序设计的程度。之后，进程能被重新调入内存，并从中断处继续执行。这种方案称为**交换**。通过中期调度程序，进程可被换出，并在后来被换入。为了改善进程组合，或者因内存要求的改变引起了过分使用可用内存而需要释放内存，就有必要使用交换。交换将在第九章讨论。

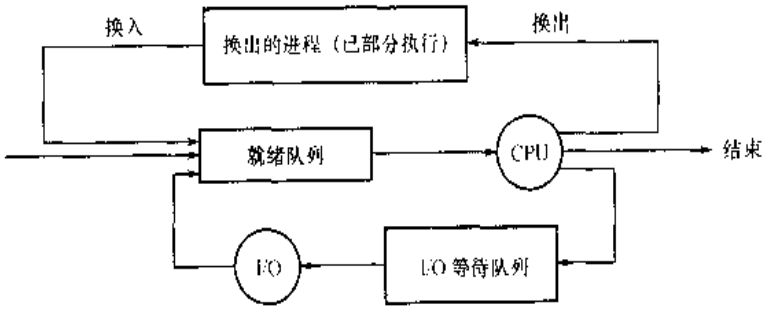


图 4.6 增加了中期调度的队列图

4.2.3 上下文切换

将 CPU 切换到另一个进程需要保存原来进程的状态并装入新进程的保存状态。这一任务称为**上下文切换**(context switch)。进程**关联**是由进程的 PCB 来表示的，它包括 CPU

寄存器的值、进程状态(图 4.1)和内存管理信息等。当发生上下文切换时,内核会将旧进程的关联状态保存在其 PCB 中,然后装入经调度要执行的新进程的已保存的关联状态。上下文切换时间是额外开销,因为切换时系统并不能做什么有用的工作。上下文切换速度因机器而不同,它依赖于内存速度、必须被复制的寄存器的数量、是否有特殊指令(如装入或保存所有寄存器的单个指令)。典型速度为 1 μs 到 1 000 μs。

上下文切换时间与硬件支持密切相关。例如,有的处理器(如 Sun UltraSPARC)提供了多个寄存器组,上下文切换只需要简单地改变当前寄存器组的指针。当然,如果活动进程超过了寄存器组数量,那么系统需要像以前一样在寄存器与内存之间进行数据复制。而且,操作系统越复杂,上下文切换所要做的工作就越多。如第九章所述,高级内存管理技术在各个关联中会要求切换更多的数据。例如,在准备使用下一个任务的空间之前,当前进程的地址空间必须要保存。地址空间如何保存和保存它需要做多少工作,取决于操作系统的内存管理方法。如第五章所述,上下文切换问题可能会成为性能瓶颈,以至于程序员需要使用其他新结构(线程)来尽可能地避免它。

4.3 进程操作

系统内的进程能并发执行,它们必须动态地被创建和删除。因此,操作系统必须提供某种机制(或工具)以创建和终止进程。

4.3.1 进程创建

进程在其执行过程中,能通过系统调用(create-process)创建多个新进程。创建进程称为*父进程*,而新进程称为该进程的*子进程*。这些新进程的每一个可以再创建其他进程,从而形成了进程*树*(图 4.7)。

通常,进程需要一定的资源(如 CPU 时间、内存、文件、I/O 设备),以完成其任务。在一个进程创建子进程时,子进程可以从操作系统那里直接获得资源,也可能只从其父进程资源子集那里获得资源。父进程可能必须在其子进程之间分配资源或共享资源(如内存或文件)。限制子进程只能使用父进程的资源子集能防止通过创建过多的子进程使系统超载。

在进程创建时,除了得到各种物理和逻辑资源外,它还能从父进程那里得到所需的初始化数据(或输入)。例如,考虑一个进程,其功能是在终端屏幕上显示文件(如 F1)的状态。在创建时,它会得到来自父进程的输入如文件 F1 的名称,它能使用该数据执行以获得所需的信息。它也能得到输出设备的名称。有的操作系统将资源传递给子进程。在这类系统上,新进程可得到两个打开文件,即 F1 和终端设备,只需在两者之间传输数据。

当进程创建新进程时,有两种执行可能:

1. 父进程与子进程并发执行。

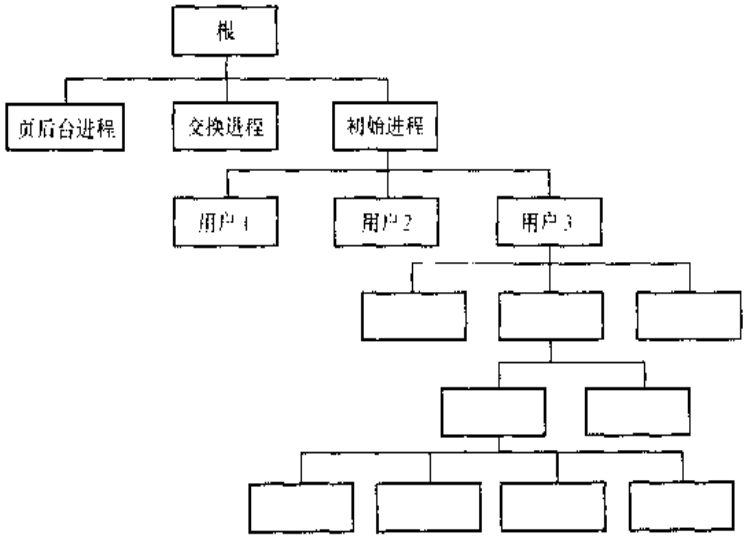


图 4.7 一个典型的 UNIX 系统中的进程树

2. 父进程等待，直到某个或全部子进程执行完毕。

新进程的地址空间也有两种可能：

1. 子进程是父进程的复制品。

2. 子进程装入另一个程序进来。

为了说明这些不同实现，可以来看一下 UNIX 操作系统。在 UNIX 中，每个进程都有一个惟一的整数形式的**进程标识符**(process identifier)。通过系统调用 fork，可创建新进程。新进程由原来进程的地址空间的复制组成。这种机制允许父进程与子进程方便地进行通信。两个进程(父和子)都继续执行位于系统调用 fork 之后的指令，但有一点不同：对于新(子)进程，系统调用 fork 的返回值为 0；而对于父进程，返回值为子进程的(非零)进程标识符。

通常，在系统调用 fork 之后，一个进程会使用系统调用 execlp，以用新程序来取代进程的内存空间。系统调用 execlp 将二进制文件装入内存，消除了原来包含系统调用 execlp 的程序的内存映射，并开始执行。采用这种方式，两进程间能通信，并能按各自的方法执行。父进程就能创建更多子进程，或者如果在子进程运行时没有什么事情可做，那么它采用系统调用 wait 把自己移出就绪队列来等待子进程的终止。如图 4.8 所示的 C 程序说明了上述 UNIX 系统调用。父进程通过执行系统调用 fork 来创建子进程。现在有两个进程运行同一程序。子进程中的 pid 的值为 0；而父进程的 pid 的值大于 0。子进程通过系统调用 execlp 用 UNIX 命令/bin/ls(用来列出目录清单)来覆盖其地址空间。父进程通过系统调用 wait 来等待子进程的完成。当子进程完成时，父进程会从 wait 调用处开始继续，并调用系统调用 exit 来表示结束。

相反，DEC VMS 操作系统创建新的进程，将指定程序装入该进程，并开始执行。微软公司 Windows NT 操作系统支持两种方式：父进程地址空间可以被复制，或者父进程可以提

供程序名让操作系统把它装入新进程的地址空间。

```
#include <stdio.h>
void main (int argc, char *argv[])

int pid;
      /* fork another process */
      pid = fork();
      if (pid < 0) { /* error occurred */
           fprintf(stderr, "Fork Failed");
           exit(-1);
      }
      else if (pid == 0) { /* child process */
           execlp("/bin/ls", "ls", NULL);
      }
      else { /* parent process */
           /* parent will wait for the child to complete */
           wait(NULL);
           printf("Child Complete");
           exit(0);
      }
}
```

图 4.8 fork 另外一个进程的 C 程序

4.3.2 进程终止

当进程完成执行最后的语句并使用系统调用 exit 请求操作系统删除它时，进程终止。这时，进程可以返回数据(输出)到其父进程(通过系统调用 wait)。所有进程资源，包括物理和虚拟内存、打开文件和 I/O 缓冲，会被操作系统所释放。

在其他情况下也会出现终止。进程通过适当的系统调用(如 abort)能终止另一个进程。通常，只有被终止进程的父进程才能执行这一系统调用。否则，用户可能会任意地终止彼此的作业。因此，父进程需要知道其子进程的标识符。因此，当一个进程创建一个新进程时，新创建进程的标识符要传递给父进程。

父进程终止其子进程的原因有很多，如下面一些原因：

• 子进程使用了超过它所分配到的一些资源。这要求父进程有一个检查其子进程的状态的机制。

• 分配给子进程的任务已不再需要。

• 父进程退出，如果父进程终止，那么操作系统不允许子进程继续。对于这类系统，如果一个进程终止(正常或不正常)，那么它的所有子进程也被终止。这种现象，称为**级联终止**

(cascading termination),通常是由操作系统进行的。

为了说明进程执行和终止,考虑一下 UNIX:可以通过系统调用 exit 而终止进程,父进程可以通过系统调用 wait 以等待子进程的终止。系统调用 wait 返回了终止子进程的进程标识符,这样父进程能够知道哪个子进程有可能终止了。如果父进程终止,那么其所有子进程会以 init 进程作为它们新的父进程。因此,子进程仍然有父进程可以收集它们的状态和执行统计。

4.4 进程协作

执行在操作系统内的并发进程可以是独立进程或协作进程。如果一个进程不能影响或被在系统内执行的其他进程所影响,那么该进程是**独立的**。显然,不与其他任何进程共享数据(临时或永久)的进程是独立的。另一方面,如果一个进程能影响或被在系统内执行的其他进程所影响,那么该进程是**协作的**。显然,与其他进程共享数据的进程是协作进程。

人可能需要提供环境以允许进程协作,这有许多理由:

- **信息共享**(information sharing):由于多个用户可能对同样的信息感兴趣(例如,共享文件),所以必须提供环境以允许对这些类型资源的并发访问。
- **加快计算**(computation speedup):如果希望一个特定任务快速运行,那么必须将它分成子任务,每个子任务可以与其他子任务并行执行。如果计算机有多个处理单元(例如 CPU 或 I/O 信道),才可实现这样的加速。
- **模块化**(modularity):可能需要按模块化方式来构造系统,如第三章所讨论的,可将系统功能划分成独立进程或线程。
- **方便**(convenience):单个用户也可能同时执行许多任务。例如,一个用户可能编辑、打印和并行编译。

协作进程的并发执行要求一定机制,以允许进程间互相通信(4.5 节)和同步动作(第七章)。

为了说明协作进程这一概念,现在来研究一下生产者-消费者问题,这是协作进程的通用范例。**生产者进程**产生信息,以供**消费者**进程消费。例如,打印程序产生字符,以供打印机驱动程序所使用。编译器产生的汇编代码供汇编程序使用;汇编程序产生目标模块供装入程序使用。

为了允许生产者进程和消费者进程能并发执行,必须要有一个项目缓冲区来被生产者填充并被消费者所使用。当消费者使用一项时,生产者能产生另一项。生产者和消费者必须是同步的,这样消费者就不会试图使用一个没有生产出来的项。在这种情况下,消费者必须等待一项被产生出来。

无限缓冲(unbounded-buffer)生产者-消费者问题对缓冲区大小没有实际限制。虽然消

费者可能需要等待新项，但是生产者总是在产生新项。**有限缓冲**生产者-消费者问题假设缓冲区大小固定。对于这种情况，如果缓冲区为空，那么消费者必须等待；如果缓冲区为满，那么生产者必须等待。

缓冲区可以由操作系统通过使用**进程间通信**(interprocess-communication，IPC)功能或由应用程序员通过使用共享内存来显式编码。下面给出有限缓冲问题的共享内存解决方案。生产者进程和消费者进程共享如下变量：

```
#define BUFFER_SIZE 10
typedef struct {
  ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

共享缓冲是通过循环数组和两个逻辑指针来实现的：in 和 out。变量 in 指向缓冲区中下一个空位；out 指向缓冲区中的第一个非空位。当 in==out 时，缓冲区为空；当(in+1)%BUFFER_SIZE==out 时，缓冲为满。

生产者进程和消费者进程代码如下。生产者进程有一个局部变量 nextProduced，以存储所产生的新项：

```
while (1) {
 /* produce an item in nextProduced */
 while ( ((in + 1) % BUFFER_SIZE) == out )
   ; /* do nothing */
 buffer[in] = nextProduced;
 in = (in + 1) % BUFFER_SIZE;
}
```

消费者进程有一个局部变量 nextConsumed，以存储所要使用的项：

```
while (1) {
 while ( in == out)
  ; // do nothing

  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  /* consume the item in nextConsumed */
}
```

这种方案最多允许缓冲区同时中有 BUFFER_SIZE－1 个项。关于缓冲区同时能存储 BUFFER_SIZE 个项的解决方法，留做做练习。

在第七章讨论在共享内存环境下，协作进程之间的同步是如何有效地实现的。

4.5 进程间通信

在 4.4 节讨论了协作进程如何能通过共享内存环境来通信。这种方法要求进程共享同一个缓冲池，并且实现缓冲的代码需要应用程序员自己明确编写。实现这种效果的另一种方法是操作系统提供机制，让协作进程能通过进程间通信(IPC)工具来进行彼此间的通信。

IPC 提供一种机制，以允许进程不必通过共同地址空间共享来通信和同步其动作。IPC 在分布式环境中(通信进程可能位于由网络连接起来的不同计算机上)尤其有用。在万维网使用的 chat 程序就是一个例子。

IPC 最好由消息传递系统提供，消息系统有许多定义方式。本节讨论消息传递系统设计的不同问题。

4.5.1 消息传递系统

消息系统的功能是允许进程互相通信而不需要利用共享数据。大家已经知道消息传递可作为微内核中的通信方法(3.5.3 小节)。在这种方案中，服务是通过普通用户进程提供的，即服务在微内核之外进行。用户进程间的通信是通过消息传递完成的。IPC 工具至少提供两个操作：发送消息和接收消息。

由进程发送的消息可以是定长的或变长的。如果只能发送定长消息，那么系统级的实现是简单的。不过，这一限制使得编程任务更加困难。另一方面，变长消息要求更复杂的系统级实现，但是编程任务变得更为简单。

如果进程 P 和 Q 需要通信，那么它们必须互相发送消息和接收消息；它们之间必须要有**通信线路**(communication link)。该线路有多种实现方法。这里并不关心线路的物理实现(如共享内存、硬件总线或网络，参见第十五章)，而只关心其逻辑实现。如下是一些逻辑实现线路和发送/接收操作的方法：

- 直接或间接通信；
- 对称或非对称通信；
- 自动或显式缓冲；
- 复制发送或引用发送；
- 固定大小或可变大小消息。

下面研究这些类型的消息系统。

4.5.2 命名

需要通信的进程必须有一个方法以互相引用。它们可使用直接或间接通信。

1. 直接通信

对于**直接通信**，需要通信的进程必须明确地命名通信的接收者或发送者。采用这种方案，原语 send 和 receive 定义如下：

- send(P,message)：发送消息到进程 P。
- receive(Q, message)：接收来自进程 Q 的消息。

这种方案的通信线路具有如下属性：

- 在需要通信的一对进程之间，自动建立线路。进程只需知道彼此的标识，以便进行通信。
- 一个线路只与两个进程相关。
- 每对进程之间只有一个线路。

这种方案展示了对称寻址，即发送进程和接收进程必须命名对方，以便通信。这种方案的一个变形采用非对称寻址。只要发送者命名接收者，而接收者不需要命名发送者。采用这种方案，原语 send 和 receive 定义如下：

- send(P, message)：发送消息到进程 P。
- receive(id, message)：接收来自任何进程的消息，变量 id 设置成与其通信的进程名称。

对称和非对称寻址方案的共同缺点是限制了结果进程定义的模块化。改变进程的名称可能必须检查所有其他进程定义。所有旧名称的引用都必须被找到，以便修改成为新名称。这种情况从独立编译角度来说是不可取的。

2. 间接通信

对于**间接通信**，消息通过**邮箱**或**端口**来发送和接收。邮箱可以被抽象成一个对象，进程可以向其中存放消息也可从中删除消息。每个邮箱都有一个惟一的标识符。对于这种方案，一个进程能通过不同的邮箱与其他进程进行通信。如果两个进程共享一个邮箱，那么它们可以进行通信。原语 send 和 receive 定义如下：

- send(A, message)：发送一个消息到邮箱 A。
- receive(A, message)：接收来自邮箱 A 的消息。

对于这种方案，通信线路具有如下属性：

- 只要一对进程中的两个成员共享一个邮箱，那么就建立了它们之间的通信线路。
- 一个线路可以与两个或更多的进程相关联。
- 两个通信进程之间可有多个不同的线路，每个线路对应于一个邮箱。

现在假设进程 P_1、P_2 和 P_3 都共享邮箱 A。进程 P_1 发送一个消息到 A，而进程 P_2 和 P_3 都对 A 执行 receive。哪个进程能收到 P_1 所发的消息呢？答案取决于人们所选择的方案：

- 允许一个线路最多只能与两个进程相关联。
- 允许一次最多一个进程执行 receive 操作。

· 允许系统随意选择一个进程以接收消息(即 P_2 或 P_3,而非两者,将接收消息)。系统可以告诉发送者谁是接收者。

邮箱可以为进程或操作系统所拥有。如果邮箱为进程所有(即邮箱是进程地址空间的一部分),那么要区分拥有者(能通过邮箱接收消息)和使用者(只能向邮箱发送消息)。由于每个邮箱都有惟一的拥有者,所以关于谁能接收发到邮箱的消息是没有什么混淆的。当拥有邮箱的进程终止,那么邮箱消失。任何进程后来向该邮箱发送消息,都会得知邮箱不再存在。

另一方面,由操作系统所拥有的邮箱是独立的,并不属于任何特定的进程。因此,操作系统必须提供机制,以允许进程进行如下操作:

· 创建一个新邮箱。

· 通过邮箱发送和接收消息。

· 删除一个邮箱。

创建一个新邮箱的进程缺省为邮箱的拥有者。开始时,拥有者是惟一能通过该邮箱接收消息的进程。不过,通过适当的系统调用,拥有权和接收特权可能传递给其他进程。当然,该规定可能会导致每个邮箱有多个接收者。

4.5.3 同步

进程间的通信可以通过调用原语 send 和 receive 来进行。这些原语的实现有不同的设计选项。消息传递可以是**阻塞**或**非阻塞**,也称为**同步**或**异步**。

· **阻塞** send:发送进程阻塞,直到消息为接收进程或邮箱所接收。

· **非阻塞** send:发送进程发送消息并再继续操作。

· **阻塞** receive:接收者阻塞,直到有消息可用。

· **非阻塞** receive:接收者收到一个有效消息或无效消息。

send 和 receive 的不同组合都有可能。当 send 和 receive 都阻塞时,则在发送者和接收者之间就有一个**集合点**(rendezvous)。

4.5.4 缓冲

不管通信是直接的或是间接的,通信进程所交换的消息都驻留在临时队列中。简单来说,队列实现有三种方法:

· **零容量**:队列的最大长度为 0;因此,线路中不能有任何消息处于等待。对于这种情况,发送者必须阻塞,直到接收者接收到消息。

· **有限容量**:队列的长度为有限的 n;因此,最多只能有 n 个消息驻留其中。如果在发送新消息时队列未满,那么该消息要放在队列中(或者复制消息或者保存消息的指针),且发送者可继续执行而不必等待。不过,线路只有有限容量。如果线路满,发送者必须阻塞直到队列中的空间可用为止。

• **无限容量**：队列长度可以无限；因此，不管多少消息都可在其中等待。发送者从不阻塞。

零容量情况称为没有缓冲的消息系统；其他情况称为自动缓冲。

4.5.5 例子：Mach

作为基于消息的操作系统的例子，考虑一下由美国卡内基－梅隆大学开发的 Mach 操作系统。Mach 内核支持多任务的创建和删除，这里的任务与进程相似，但能有多个控制线程。Mach 的绝大多数通信，包括绝大多数系统调用和所有任务间信息，是通过消息实现的。消息通过邮箱（Mach 称之为*端口*）来发送和接收。

即使系统调用也是通过消息进行的。每个任务在创建时，也创建了两个特别邮箱：**内核**（kernel）邮箱和**通报**（notify）邮箱。内核使用内核邮箱与任务进行通信。内核向通报端口发送事件发生的通知。消息传输只需要三个系统调用。调用 msg_send 向邮箱发送消息。消息可通过 msg_receive 接收。**远程过程调用**（RPC）通过 msg_rpc 执行，它能发送消息并只等待来自发送者的一个返回消息。这样，RPC 模拟了典型的子程序过程调用，但能在系统之间工作。

系统调用 port_allocate 创建新邮箱并为其消息队列分配空间。消息队列的最大尺寸缺省为 8 个消息。创建邮箱是该邮箱的拥有者的任务。拥有者也被赋予了对邮箱的访问权。一次只能有一个任务能拥有邮箱或从邮箱中接收，但是如果需要，这些权利也能发送给其他任务。

邮箱开始时，其消息队列为空。随着消息向邮箱发送，消息被复制到邮箱中。所有消息具有同样的优先权。Mach 确保来自同一发送者的多个消息按照先进先出（FIFO）顺序来排队，但并不确保绝对排序。例如，来自两个发送者的消息可以按任何顺序排队。

消息本身由固定长度的头和可变长的数据部分组成。头部包括消息长度和两个邮箱名称。当发送消息时，一个邮箱名称是消息发送的目的邮箱。通常，发送线程也期待一个回应，发送者的邮箱名称传递到接收任务，接收任务可用它作为"返回地址"，以发回消息。

消息的可变部分是具有类型的数据项的链表。链表内的每一项都有类型、大小和值。消息内所表示的对象类型很重要，因为操作系统定义的对象，如拥有权或接收访问权限、任务状态、内存段，可通过消息发送。

操作 send 和 receive 本身很灵活。例如，当向一个邮箱发送消息时，该邮箱可能已满。如果邮箱未满，消息可复制到邮箱，发送线程继续。如果邮箱已满，发送线程有四个选择：

1. 无限等待直接邮箱有空间为止。
2. 最多等待 n ms。
3. 根本不等待，而立即返回。
4. 暂时缓存消息。即使所要发送到的邮箱已满，还是可以给操作系统一个消息，以便

保持。当消息能被放进邮箱时，通报消息会送回到发送者。对于给定发送线程，在任何时候，只能有一个给已满邮箱等待处理的消息。

最后选项用于服务器任务（如行式打印机的驱动程序）。在处理完请求之后，这些任务可能需要给请求服务的任务发送一个一次性的应答，但即使在客户应答邮箱已满时也必须继续处理其他服务请求。

操作 receive 必须指明从哪个邮箱或邮箱集合来接收消息。一个**邮箱集合**是由任务所声明的、能组合在一起作为一个邮箱以满足任务的目的的一组邮箱。任务中的线程只能从任务具有接收权限的邮箱或邮箱集合中接收消息。系统调用 port_status 能返回给定邮箱的消息数量。接收操作试图从如下两处接收消息：

1. 邮箱集合内的任何邮箱。
2. 特定的（已命名的）邮箱。

如果没有消息等待被接收，那么接收线程可能等待（最多等待 n ms）或不等待。

Mach 系统特别为分布式系统而设计，关于这点将在第十五章到第十七章中讨论，但是它也适用于单处理器系统。消息系统的主要问题是性能差，这是由需要在发送者和邮箱以及邮箱到接收者之间进行的消息复制而引起的。通过使用虚拟内存管理技术（第十章），Mach 消息系统试图避免双重复制操作。其关键就是 Mach 将包含发送者消息的地址空间映射到接收者的地址空间，消息本身并不真正被复制。这种消息管理技术大大地提高了性能，但是只适用于系统内部的消息传递。Mach 操作系统在附加章中有相关讨论，发布在这本书的网站上（http://www.bell-labs.com/topic/books/os-book）。

4.5.6 例子：Windows 2000

Windows 2000 操作系统是现代设计的典型，它采用模块化，以增强功能，并降低了用以实现新特征所需的时间。Windows 2000 支持多个操作环境或子系统，应用程序可通过消息传递机制进行通信。应用程序可作为 Windows 2000 子系统服务器的客户。

Windows 2000 的消息传递工具称做**本地过程调用**（LPC）工具。Windows 2000 的 LPC 在位于同一机器的两进程之间进行通信。它类似于广泛使用的标准 RPC 机制，但是为 Windows 2000 进行了优化并很特殊。与 Mach 一样，Windows 2000 使用了端口对象，以建立和维护两个进程之间的连接。调用子系统的每个客户需要一个通信信道，由端口对象提供且不能继承。Windows 2000 使用两种类型的端口：连接端口和通信端口。它们事实上是相同的，但根据它们如何使用而具有不同名称。连接端口称为*对象*，为所有进程所可见，它们赋予应用程序一种方法来建立通信信道（第二十一章）。这种通信工作如下：

- 客户机打开子系统的连接端口对象的句柄。
- 客户机发送连接请求。
- 服务器创建两个私有通信端口，并返回其中之一的句柄给客户机。

· 客户机和服务器使用相应端口句柄，以发送消息或回调，并等待回答。

Windows 2000 使用三种类型的端口消息传递技术，端口可在客户机建立信道时被指明。最为简单的，用于小消息的，使用端口消息队列作为中间存储，并将消息从一个进程复制到另一个进程。采用这种方法，可发送最多 256 B 的消息。

如果客户机需要发送更大的消息，那么它可通过区段对象（或共享内存）来传递消息。客户机在建立频道时，必须决定它是否需要发送大消息。如果客户机确定它确实需要发送大消息，那么它就要求建立区段对象。同样，如果服务器确定应答会很大，它就建立一个区段对象。为了能使用区段对象，需要发送一个小消息，它包括关于区段的一个指针和大小信息。这种方法比第一种方法更为复杂，但是它避免了数据复制。对于这两种情况，当客户程序或服务器程序不能马上响应请求时，可使用回调机制。回调机制允许它们执行异步消息传递。

4.6 客户机－服务器系统通信

假设一个用户需要访问位于某个服务器上的数据。例如，一个用户需要知道位于服务器 A 上的一个文件的行、字和字符的总数。这种请求由远程服务器 A 处理，它访问文件，计算所需结果，最后将真实数据传送回给用户。

4.6.1 套接字

套接字（socket）可定义为通信的端点。一对通过网络通信的进程需要使用一对套接字，即每个进程各有一个。套接字由 IP 地址和端口号连接组成。通常，套接字采用客户机-服务器结构。服务器通过监听指定端口来等待进来的客户机请求。一旦收到请求，服务器就接受来自客户机套接字的连接，从而完成连接。

服务器实现的特定服务（如 telnet、ftp 和 http）是通过监听熟知端口（telnet 服务器监听端口 23，ftp 服务器监听端口 21，Web 或 http 服务器监听端口 80）进行的。所有低于 1024 的端口都认为是众所周知的，可以用它们来实现标准服务。

当客户机进程发出连接请求时，它被主机赋予一个端口。该端口是大于 1024 的某个任意数。例如，如果 IP 地址为 14.85.5.20 的主机 X 的客户机希望与地址为 161.25.19.8 的网络服务器（监听端口 80）建立连接，主机 X 可能被赋予端口 1625。该连接有一对套接字组成：主机 X 上的（146.86.5.20:1625），网络服务器上的（161.25.19.8:80）。这种情况如图 4.9 所示。根据目的端口号，在主机间传输的数据包可分送

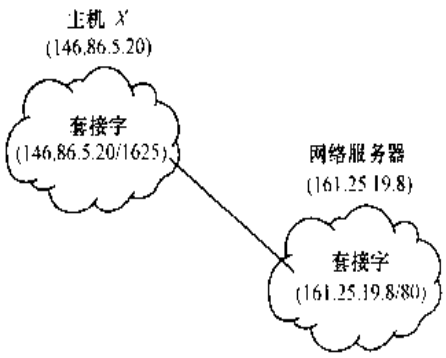


图 4.9 使用套接字通信

给合适的进程。

所有连接必须惟一。因此，如果主机 X 的另一个进程希望与同样的网络服务器建立另一个连接，那么它会被赋予另一个大于 1024 但不等于 1625 的端口号。这确保了所有连接都有惟一的一对套接字。

虽然本书的绝大多数程序例子使用 C，但是笔者使用 Java 来演示套接字，这是因为 Java 提供了一个套接字的简单接口，而且也提供了丰富的网络工具库。如果对用 C 或 C++进行套接字编程感兴趣，可以参考推荐读物。

Java 提供了三种不同类型的套接字。**面向连接(TCP)套接字**是用 Socket 类实现的。**无连接(UDP)套接字**使用了 DatagramSocket 类。第三种类型是 MulticastSocket 类，它是 DatagramSocket 类的子类。多点传送套接字允许数据发送给多个接收者。

作为基于 Java 套接字的例子，现在介绍一个 Java 类，它实现了日期时间服务器。该操作允许客户程序向服务器询问日期、时间。服务器监听端口 5155，不过任何大于 1024 的端口都可以。当收到连接时，服务器会返回日期时间给客户程序。

时间服务器程序如图 4.10 所示。服务器创建了 ServerSocket 以指明它将监听端口号 5155。接着，它通过执行 accept 方法开始监听端口。服务器阻塞在方法 accept 上，以等待客户机请求连接。当接收到连接请求时，accept 会返回一个套接字，以供服务器用来与客户机进行通信。

```
import java.net.*;
import java.io.*;
public class Server
{
   public static void main(String[] args) throws IOException{
        Socket client = null;
        PrintWriter pout = null;
        ServerSocket sock = null;

        try {
              sock = new ServerSocket(5155);
              // now listen for connections

              while (true){

                  client = sock.accept();

                  // we have a connection
                  pout = new PrintWriter(client.getOutputStream(), true);

                  // write the Date to the socket
                  pout.println(new java.util.Date().toString());
```

```
                pout.close();
                client.close();
            }
        }
        catch (IOException ioe){
            System.err.println(ioe);
        }
        finally{
            if (client != null)
                client.close();
            if (sock != null)
                sock.close();
        }
    }
}
```

图 4.10 日期-时间服务器

有关服务器如何与套接字通信的细节如下。首先服务器建立 PrintWriter 对象用来与客户机进行通信。PrintWriter 对象允许服务器通过普通的输出方法 print 和 println 来向套接字进行写操作。服务器进程通过调用方法 println 将日期-时间发送给客户机。一旦完成了将日期-时间写到套接字,服务器就关闭与客户机相连的套接字,并重新监听其他更多的请求。

客户机创建套接字跟服务器监听的端口相连,以便跟服务器进行通信。用如图 4.11 所示的 Java 程序实现了客户机程序。客户机创建了 Socket,并请求与 IP 地址为 127.0.0.1、端口号为 5155 的服务器建立连接。一旦建立了连接,客户机就通过普通流 I/O 语句来对套接字进行读。在得到服务器的日期-时间后,就关闭端口并退出。IP 地址 127.0.0.1 是特殊 IP 地址,称为**本地主机**地址。当计算机引用地址 127.0.0.1 时,它是在引用自己。这一机制允许同一主机上的客户机和服务器能通过 TCP/IP 协议进行通信。IP 地址 127.0.0.1 可以为运行日期-时间服务器的另一个主机的 IP 地址所替代。

```
import java.net.*;
import java.io.*;
public class Client
{
   public static void main(String[] args) throws IOException{
      InputStream in = null;
      BufferedReader bin = null;
      Socket sock = null;
      try
      {
         // make connection to socket
         sock = new Socket("127.0.0.1", 5155);
```

```
            in = sock.getInputStream();
            bin = new BufferedReader(new InputStreamReader(in));
            String line;
            While ( (line = bin.readLine()) ! = null )
                System.out.println(line);
        }
        catch (IOException ioe);
            System.err.println(ioe);
        }
        finally{
            if ( sock ! = null )
                sock.close();
        }
    }
```

图 4.11 客户端

使用套接字通信，虽然普遍和高效，但是它属于较为低层形式的分布式进程通信。一个理由是套接字只允许在通信线程之间交换无结构的字节流。客户机或服务器程序需要负责加上数据的结构。在下面两小节中，介绍可选择的两种高层的通信方法：远程过程调用(RPC)和远程方法调用(RMI)。

4.6.2 远程过程调用

一种最为普遍的远程服务形式是 RPC 方式，这已在 4.5.4 小节中简要介绍了。RPC 设计成抽象过程调用机制，用于通过网络进行连接的系统。它在许多方面都类似于 4.5 节所述的 IPC 机制，并且通常建立在这种系统之上。因为在所处理的环境中，进程在不同系统上执行，所以必须使用基于消息的通信方案来提供远程服务。与 IPC 工具不同，用于 RPC 通信而交换的消息有很好的结构，因此不再仅仅是数据包。这些消息传递给远程系统监听端口的 RPC 后台，它们包含要执行函数的名称和传递给该函数的参数。然后该函数根据请求而执行，任何输出结果通过另一个消息送回给请求者。

*端口*只是一个数字，它包含在消息包的开始处。虽然一个系统通常只有一个网络地址，但是它在这一地址内可以有许多端口号，以区分其所支持的许多网络服务。如果一个远程进程需要服务，那么它就向适当端口发送它的消息。例如，如果一个系统欲允许其他系统能列出其当前用户，那么它可以有一个后台支持这样的 RPC，并监听一个端口如 3027。任何远程系统只要向位于服务器端口 3027 发送一个 RPC 消息，就能得到所需要的信息(即当前用户列表)，数据将通过答复消息收到。

RPC 语义允许客户机调用位于远程主机上的过程，就如同调用本地过程一样。RPC 系统隐藏了必要细节，以允许通信发生。RPC 系统通过在客户机端提供**存根**(stub)来做到这

一点。通常，对于每个独立的远程过程都有一个不同的存根。当客户机调用远程过程时，RPC 系统调用合适的存根，并传递提供给远程过程的参数。该存根定位服务器的端口，并编组参数。参数编组涉及到将参数打包成可通过网络传输的形式。接着，存根使用消息传递向服务器发送一个消息。服务器端的一个类似存根接收到这一消息，并调用服务器上的过程。如果有必要，返回值可通过使用同样的技术，以传回给客户机。

有一个必须处理的事项是关于如何处理客户机和服务器系统机的数据表示的差别。考虑一个 32 bit 整数的表示。有的系统使用高内存地址来存储最重要的字节（称为大端结尾 *big-endian*），而其他系统使用高内存地址来存储最不重要的字节（称为小端结尾 *little-endian*）。为了处理这一问题，许多 RPC 系统都定义了数据的机器无关表示。一种这样的表示称为**外部数据表示（XDR）**。在客户机端，参数编组涉及到将机器相关数据编组成 XDR 来向服务器发送消息。在服务器端，XDR 数据被解除编组，并重新转换成服务器所用的机器相关表示。

RPC 机制在许多网络系统中应用普遍，所以应讨论有关其操作等一些其他事项。其中重要的一个问题就是调用的语义。虽然本地过程调用只有在极端情况下才失败，但是 RPC 由于普通网络错误，可能会失败，或被重复和多次执行。因为人们是通过不可靠的通信线路来传输消息，所以操作系统较容易确保一个消息最多只执行一次，而不是刚好执行一次。因为本地过程调用具有后面的属性，所以绝大多数系统试图也拥有这一功能。它们通过为每个消息附加时间戳的方法来做到。服务器对其所处理的消息，必须有一个完整的或足够长的时间戳历史，以便确保能检测到重复消息。进来的消息，如果其时间戳已在历史上，则被忽略。如何产生这些时间戳将在 17.1 节中讨论。

另一个重要事项是关于服务器与客户机间的通信。对于标准过程调用，在连接、装入或执行时会出现一定形式的捆绑（第九章），如过程调用名称被过程调用的内存地址所代替。RPC 方案要求有一个类似的用于客户机和服务器端口的捆绑，但是客户机如何知道服务器上的端口号呢？没有一个系统拥有其他系统完全的信息，因为它们并不共享内存。有两种常用方法。第一种方法，捆绑信息以固定端口地址形式预先固定。在编译时，RPC 调用有一个相关的固定端口号。一旦程序被编译后，服务器就不能改变请求服务的端口号。第二种方法，捆绑通过集合点机制动态地进行。通常，操作系统在一个固定 RPC 端口上提供集合后台程序（也称为 matchmaker）。客户程序发送一个消息（包括 RPC 的名称）给集合服务程序，以请求它所需要执行的 RPC 端口地址。该端口号返回，RPC 调用可发送至这一端口直到进程终止（或服务器失败）。这种方法需要初始请求的额外开销，但是比第一种方法更灵活。图 4.12 说明了一个交互例子。

RPC 方案可用于实现分布式文件系统（第十六章）。这种系统可通过 RPC 后台程序和客户机来实现。消息发送到服务器的 DFS 端口，以进行应有的文件操作。消息包括要执行的磁盘操作。磁盘操作可能是 read、write、rename、delete 或 status，对应通常的文件相

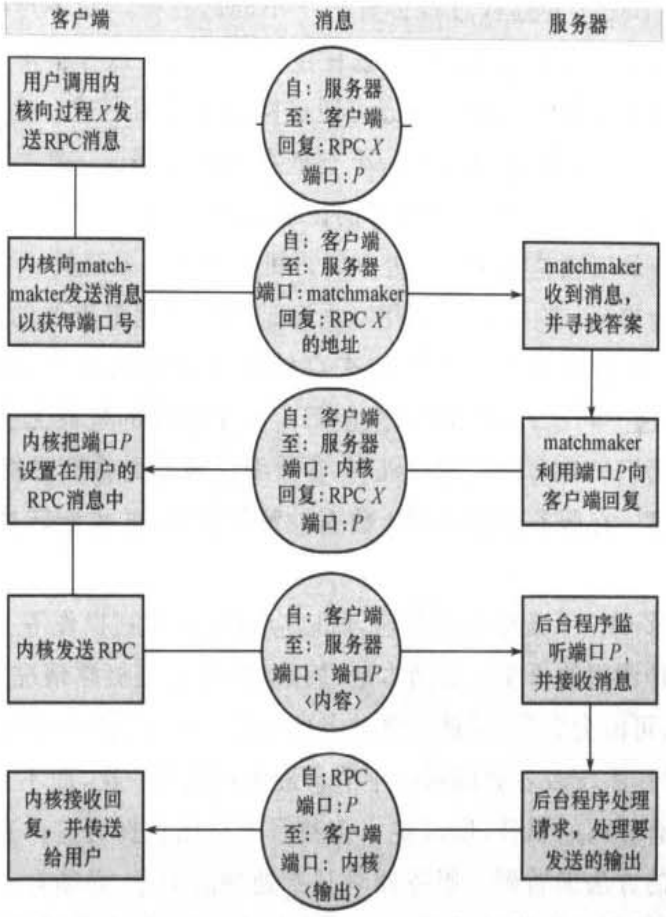


图 4.12 远程过程调用的执行过程

关系统调用。返回消息包括来自该调用(由 DFS 后台程序代表客户机来执行)的任何数据。例如，一个消息可能包括一个上传整个文件到客户机的请求，或被限制为简单块请求。对于后者，当要传输整个文件时，可能需要多个这样的请求。

4.6.3 远程方法调用

远程方法调用(remote method invocation，RMI)是 Java 的一个类似于 RPC 的功能。RMI 允许线程调用远程对象的方法。如果对象位于不同 Java 虚拟机，则被认为是远程的。因此，远程对象可能位于同一计算机的不同 JVM 或位于通过网络连接的主机的 JVM 上。这种情况如图 4.13 所示。RMI 和 RPC 在两方面有根本的不同。第一，RPC 支持子程序编程，即只能调用远程的子程序或函数；RMI 是基于对象的：它支持调用远程对象的方法。第二，在 RPC 中，远程过程的参数是普通数据结构；而 RMI 可以将对象作为参数传递给远程方法。RMI 通过允许 Java 程序调用远程对象的方法，使得用户能够开发分布在网络上的 Java 应用程序。

为了使远程方法对客户机和服务器都透明，RMI 采用存根(stub)和骨干(skeleton)实现

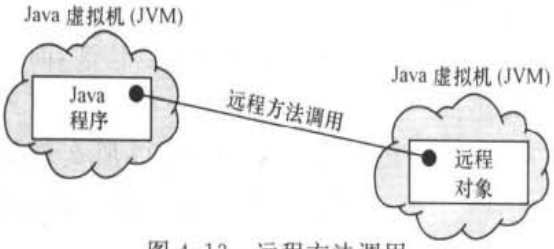


图 4.13 远程方法调用

了远程对象。**存根**是远程对象的代理;它驻留在客户机中。当客户机调用远程方法时,远程对象的存根被调用。这种客户端存根负责创建一个**包**,它有服务器上要调用方法的名称和用于该方法的编排参数。存根会将该包发送给服务器,远程对象的骨干会接收它。**骨干**负责重新编排参数并调用服务器上所要执行的方法。骨干接着编排返回值(或异常,若有)然后打包,并将该包返回给客户机。存根重新编排返回值,并传递给客户机。

下面说明这一过程是如何工作的。假设客户机希望调用远程对象 Server 的一个方法,该方法具有签名 someMethod(Object, Object),并返回 boolean 值。客户机执行如下语句:

```
boolean val = Server.someMethod(A, B);
```

使用参数 A 和 B 对 someMethod 调用了远程对象的存根。存根将参数 A 和 B 以及要在服务器上调用的方法名称一起打包,接着将该包发送给服务器。服务器上的骨干会重新编排参数并调用方法 someMethod。someMethod 的真正实现驻留在服务器上。一旦方法完成,骨干会编排从 someMethod 返回的 boolean 值,并将该值发回给客户机。存根重新编排该返回值,并传递给客户机。这一过程如图 4.14 所示。

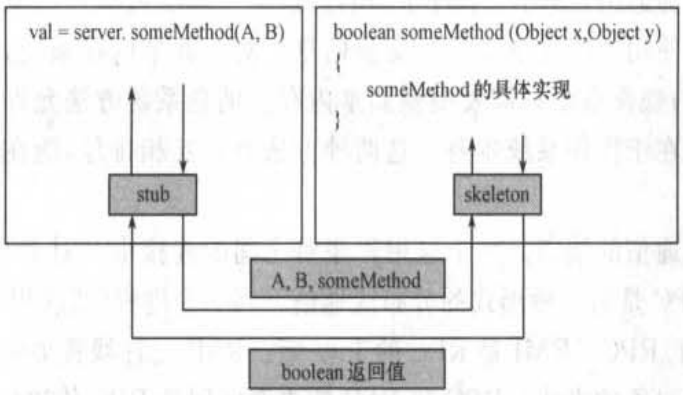


图 4.14 编排参数

幸运地,RMI 提供的抽象程度使得存根和骨干透明,从而允许 Java 开发人员能编写程序,并如同调用本地方法一样地调用分布式方法。不过,你必须理解有关参数传递行为的几个规则:

- 如果编排参数是**本地**(**非远程**)对象,那么通过称为**对象串行化**的技术来复制,以便

传递。不过，如果参数也是远程对象，那么它们通过引用来传递。对于我们的例子，如果 A 是本地对象而 B 是远程对象，那么 A 就被串行化并复制以传递，而 B 通过引用被传递。这能允许服务器远程执行 B 的方法。

• 如果本地对象需要作为参数传递给远程对象，那么就必须实现接口 java.io.Serializable。核心 Java API 中的许多对象都实现了 Serializable，因此允许它们可同 RMI 一起使用。对象串行化允许将对象状态写入字节流。

4.7 小 结

进程是执行中的程序。随着进程的执行，它改变状态。进程状态由进程当前活动所定义。每个进程可处于如下状态之一：新的、就绪、运行、等待或终止。每个进程在操作系统内通过自己的进程控制块(PCB)来表示。

当前不在执行的进程会被放在某个等待队列中。操作系统的两个主要队列类型是 I/O 请求队列和就绪队列。就绪队列包括所有就绪可执行的并等待 CPU 的进程。每个进程都由 PCB 来表示，PCB 链接起来就形成了一个就绪队列。长期(或作业)调度选择进程来竞争 CPU。通常，长期调度会受资源分配考虑严重影响，尤其是内存管理的影响。短期(或 CPU)调度从就绪队列中选择进程。

系统内的进程能并发执行。有许多理由允许并发执行：信息共享，加快计算，模块化和方便性。并发执行需要创建和删除进程的机制。

操作系统中的执行进程可以是独立进程或协作进程。协作进程要有互相通信的机制。主要存在两种互补的通信形式：共享内存和消息系统。共享内存方法要求通信进程共享一些变量。进程通过使用这些共享变量来交换信息。对于共享内存系统，提供通信的责任主要在于应用程序员；操作系统只需要提供共享内存。消息系统方法允许进程交换信息。提供通信的主要责任在于操作系统本身。这两种方法并不互相排斥，能在同一操作系统内同时实现。

套接字定义为通信的端点。一个应用程序对之间的连接由一对套接字组成，通信信道每端各有一个。RPC 是另一种形式的分布式通信。当一个进程(或线程)调用一个远程应用的方法时，就出现了 RPC。RMI 是 RPC 的 Java 版。RMI 允许线程如同调用本地对象的方法一样来调用远程对象的方法。RPC 和 RMI 的主要区别是 RPC 传递给远程过程的数据是按普通数据结构形式的。RMI 允许在远程方法调用中传递对象。

习 题 四

4.1 MS-DOS 没有提供并发执行的方式。论述并发执行给操作系统带来的三个主要复杂化因素。

4.2 论述期短、中期和长期调度之间的区别。

4.3 DECSYSTEM 20 计算机有多个寄存器组，描述一下当新的关联环境已载入到一个寄存器组时进行上下文切换的过程。如果新的关联环境在内存中而不是在寄存器组中，且所有寄存器组都在使用，又必将发生什么事情？

4.4 描述一下内核在两个进程间进行上下文切换的动作。

4.5 下面设计的好处和坏处分别是什么？系统层次和用户层次都要考虑到。

a. 直接和间接通信

b. 对称和非对称通信

c. 自动和显式缓冲

d. 复制发送和引用发送

e. 固定大小和可变大小消息

4.6 第 4.4 节中的正确的生产者－消费者算法在任一时刻只允许装满 $n-1$ 个缓冲区。修改这个算法让它能够充分利用所有的缓冲区。

4.7 考虑使用邮箱的进程间通信方案。

a. 假设进程 P 想等待两条消息，一条来自邮箱 A，一条来自邮箱 B，它应该以怎样的顺序来执行 send 和 receive？

b. 如果进程 P 想等待一条来自邮箱 A 或邮箱 B（或 A 和 B）的消息，它应以怎样的顺序来执行 send 和 receive？

c. 一个 receive 操作使得进程处于等待状态直到邮箱非空。设计一个方案，允许进程等待直到邮箱为空，或解释一下为什么这样的方案不存在。

4.8 写一个基于套接字的算命者服务器。你的程序要创建一个监听指定端口的服务器。当一个客户端接收到一个连接时，服务器要从它的幸运数字数据库中随机选一个幸运数字返回给客户。

推荐读物

Brinch-Hansen[1970]论述了关于 RC 4000 系统的进程间通信。Schlichting 和 Schneider[1982]论述了异步消息传递原语。Bershad 等人[1990]论述了在用户层实现 IPC 机制。

Gray[1997]详细论述了 UNIX 系统的进程间通信。Barrera[1991]和 Vahalia[1996]展示了 Mach 系统的进程间通信。Solomon 和 Russinovich[2000]概述了 Windows 2000 的进程间通信。Stevens[1997]的著作里有关于套接字编程的详细论述。

Birrell 和 Nelson[1984]论述了 RPC 的实现。Shrivastava 和 Panzieri[1982]展示了可靠的 RPC 机制设计。Tay 和 Ananda[1990]综述了 RPC。Stankovic[1982]和 Staunstrup[1982]比较了过程调用和消息传递通信。

Tanenbaum[1996]论述了套接字和 RPC。Waldo[1998]和 Farley[1998]论述了 RPC 和 RMI。Niemeyer 和 Peck[1997]、Horstmann 和 Cornell[1998]对使用 RMI 做了好的介绍。RMI 主页列出了最新的资料(http://www.javasoft.com/products/jdk/rmi)。

第五章 线 程

第四章讨论的进程模型假设进程是一个具有单个控制线程的执行程序。现在许多现代操作系统都提供单个进程包括多个控制线程的特征。本章引入了与多线程计算机系统相关的许多概念，包括有关 **Pthread** API 和 Java 线程的讨论。本节将研究与多线程编程相关的许多事项以及它是如何影响操作系统设计的。最后，将研究一些现代操作系统如何在内核级提供对线程的支持。

5.1 概 述

线程，有时称为**轻量级进程**(lightweight process，LWP)，是 CPU 使用的基本单元；它由线程 ID、程序计数器、寄存器集合和堆栈组成。它与属于同一进程的其他线程共享其代码段、数据段和其他操作系统资源(如打开文件和信号)。一个传统(或**重量级**(heavyweight)进程)只有单个控制线程。如果进程有多个控制线程，那么它能同时做多个任务。图 5.1 说明了传统的单线程进程和现代的多线程进程之间的差别。

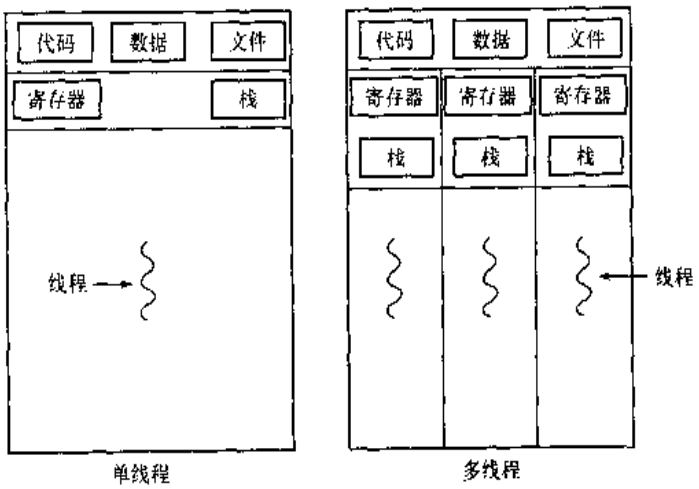


图 5.1 单线程进程和多线程进程

5.1.1 动机

运行在现代桌面个人计算机上的许多软件包都是**多线程**(multithreaded)的。一个应用

程序通常实现得像一个具有多个控制线程的独立进程。网页浏览器可能有一个线程显示图像或文本，另一个线程从网络接收数据。字处理器可能有一个线程用于显示图形，另一个线程用以读入用户的键盘输入，还有第三线程在后台进行拼写和语法检查。

在某些情况下，一个单独的应用程序可能需要执行多个相似任务。例如，网页服务器接收用户关于网页、图像、声音等的请求。一个忙碌的网页服务器可能有多个（可能数百个）客户并发访问它。如果网页服务器作为传统**单个线程**（single-threaded）的进程来执行，那么只能一次对一个客户服务。这样，客户必须等待处理请求的时间可能会很大。

一种解决方法是让服务器作为单个进程运行来接收请求。当服务器收到请求时，它会创建另一个进程以处理请求。事实上，这种进程创建方法在线程流行之前很常用。如上一章所述，进程创建特别麻烦。如果新进程与现有进程执行同样的任务，那么为什么需要所有这些开销呢？如果一个具有多个线程的进程能达到同样目的，那么将更为有效。这种方案要求网页服务器进程是多线程的。服务器创建一个独立线程以监听客户请求；当有请求产生时，服务器不是创建另一个进程而是创建另一个线程以处理请求。

线程在远程过程调用（RPC）系统中也有很重要的作用。回想一下第四章，RPC 通过提供一种类似于普通函数或子程序调用的通信机制，来允许进程间通信。通常，RPC 服务器是多线程的。当一个服务器接收到消息，它通过使用独立线程来处理消息。这允许服务器能处理多个并发请求。

5.1.2 优点

多线程编程具有如下四类主要优点：

1. **响应度高**：如果对一个交互式应用程序采用多线程，即使其部分阻塞或执行较冗长的操作，那么该程序仍能继续执行，从而增加了对用户的响应程度。例如，多线程网页浏览器在用一个线程装入图像时，能通过另一个线程与用户交互。

2. **资源共享**：线程默认共享它们所属进程的内存和资源。代码共享的优点是它能允许一个应用程序在同一地址空间内有多个不同的活动线程。

3. **经济**：进程创建所需要的内存和资源的分配比较昂贵。不过，由于线程能共享它们所属进程的资源，所以线程创建和上下文切换会更为经济。如果要实际地测量进程创建和维护开销（而非线程开销）的差别，则较为困难；但是前者通常要比后者花费更多的时间。对于 Solaris 2，进程创建要比线程创建慢 30 倍，而进程上下文切换要比线程上下文切换慢 5 倍。

4. **多处理器体系结构的利用**：多线程的优点之一是能充分使用多处理器体系结构，以便每个线程能并行运行在不同的处理器上。不管有多少 CPU，单线程进程只能运行在一个 CPU 上。在多 CUP 机器上使用多线程增加了并发功能。对于单处理器体系结构，CPU 通常在线程之间快速移动以创建并行执行的假象，但是实际上在一个时候只能运行一个线程。

5.1.3 用户线程与内核线程

迄今为止只是泛泛地讨论了线程。不过，有两种不同方法可用来提供线程支持：用户层的*用户线程*或*内核层的内核线程*。

• **用户线程**在内核之上支持，并在用户层通过线程库来实现。线程库提供对线程创建、调度和管理的支持而无需内核支持。由于内核并不知道用户级的线程，所以所有线程的创建和调度是在用户空间内进行的，而无需内核干预。因此，用户级线程通常能快速地创建和管理；但是它们也有缺点。例如，如果内核是单线程的，那么任何一个用户级线程若执行阻塞系统调用就会引起整个进程阻塞，即使还有其他线程可以在应用程序内运行。用户级线程库包括 POSIX **Pthread**、Mach **C-thread** 和 Solaris 2 **UI-thread**。

• **内核线程**由操作系统直接支持：内核在其空间内执行线程创建、调度和管理。因为线程管理是由操作系统完成的，所以内核线程的创建和管理通常要慢于用户线程的创建和管理。不过，由于内核管理线程，当一个线程执行阻塞系统调用时，内核能调度应用程序内的另一个线程以便执行。而且，在多处理器环境下，内核能在不同处理器上调度线程。绝大多数当代操作系统，包括 Windows NT、Windows 2000、Solaris 2、BeOS 和 Tru64 UNIX（前身是 Digital Unix），都支持内核线程。

本书将在 5.4 节中讨论 Pthread 作为用户级线程库的例子。还会讨论 Windows 2000（5.6 节）和 Solaris 2（5.5 节）作为具有内核线程支持的操作系统的例子。在 5.7 节中将讨论 Linux 如何提供线程支持（虽然 Linux 并不称这为线程）。

Java 也提供对线程的支持。不过，由于 Java 线程是由 Java 虚拟机（JVM）来创建和管理的，所以它不能简单地判断是属于用户线程还是内核线程的范畴。本书在 5.8 节中讨论 Java 线程。

5.2 多线程模型

许多系统都提供对用户和内核线程的支持，从而有不同的多线程模型。现在看一下三种常用类型的线程实现。

5.2.1 多对一模型

多对一模型（图 5.2）将许多用户级线程映射到一个内核线程。线程管理是在用户空间进行的，因而效率比较高，但是如果一个线程执行了阻塞系统调用，那么整个进程就会阻塞。而且，因为任一时刻只有一个线程能访问内核，多个线程不能并行运行在多处理器上。**Green threads**，Solaris 2 所提供的线程库，就使用了这种模型。另外，在不支持内核级线程的操作系统上所实现的用户级线程库也使用了多对一模型。

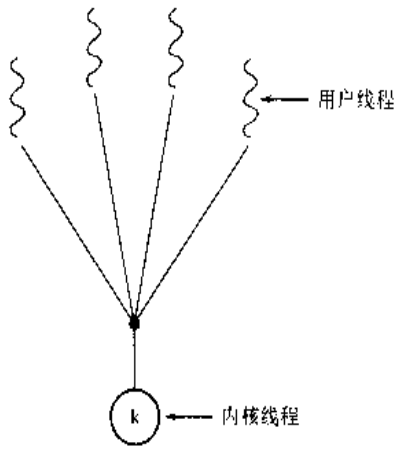


图 5.2 多对一模型

5.2.2 一对一模型

一对一模型(图 5.3)将每个用户线程映射到一个内核线程。该模型在一个线程执行阻塞系统调用时,能允许另一个线程继续执行,所以它提供了比多对一模型更好的并发功能;它也允许多个线程能并行地运行在多处理器系统上。这种模型的惟一缺点是创建一个用户线程就需要创建一个相应的内核线程。由于创建内核线程的开销会影响应用程序的性能,所以这种模型的绝大多数实现限制了系统所支持的线程数量。Windows NT、Windows 2000 和 OS/2 实现了一对一模型。

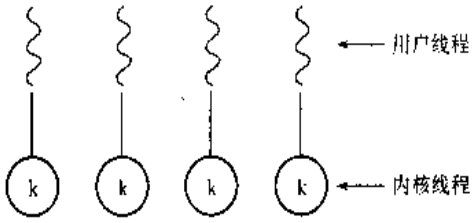


图 5.3 一对一模型

5.2.3 多对多模型

多对多模型(图 5.4)多路复用了许多用户级线程到同样数量或更小数量的内核线程上。内核线程的数量可能与特定应用程序或特定机器有关(位于多处理器上的应用程序可比单处理器上的应用程序分配更多数量的内核线程)。虽然多对一模型允许开发人员随意创建任意多的用户线程,但是由于内核只能一次调度一个线程,所以并不能增加并发性。一对一模型提供了更大的并发性,但是开发人员必须小心不要在应用程序内创建太多的线程(在有些情况下可能受她所能创建的线程数量的限制)。多对多模型没有这两者的缺点:开发人员可创建任意多的必要用户线程,并且相应内核线程能在多处理器系统上并行执行。而且,当

一个线程执行阻塞系统调用时，内核能调度另一个线程来执行。Solaris 2、IRIX、HP-UX 和 Tru64 UNIX 都支持这种模型。

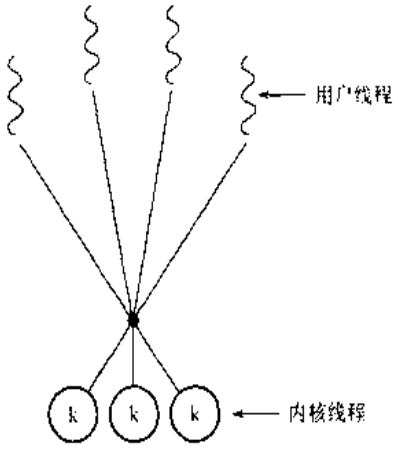


图 5.4 多对多模型

5.3 若干多线程问题

本节将讨论与多线程程序有关的一些问题。

5.3.1 系统调用 fork 和 exec

在第四章讨论了系统调用 fork 如何用于创建独立的、重复的进程。在多线程程序中，系统调用 fork 和 exec 的语义有所改变。如果程序中的一个线程调用 fork，那么新进程会复制所有线程还是新进程只有单个线程？有的 UNIX 系统有两种形式的 fork，一种复制所有线程，另一种只复制调用了系统调用 fork 的线程。系统调用 exec 的工作方式与第四章所述的方式通常相同。即，如果一个线程调用了系统调用 exec，那么 exec 参数所指定的程序会替换整个进程，包括所有线程和 LWP。

fork 的两种形式的使用与应用程序有关。如果调用 fork 之后立即调用 exec，那么复制所有线程就没有必要，因为 exec 参数所指定的程序会替换整个进程。在这种情况下，只复制调用线程比较恰当。不过，如果在 fork 之后另一进程并不调用 exec，那么另一进程就应复制所有线程。

5.3.2 取消

线程取消(thread cancellation)是在线程完成之前来终止线程的任务。例如，如果多个线程并发执行以搜索数据库并且一个线程返回了结果，那么其他线程就可被取消。另一情况是用户按下网页浏览器上的按钮以停止进一步装入网页。通常装入网页是由一个独立线

程进行的。当用户按下*停止*按钮时，装入网页的线程就被取消了。

要被取消的线程通常称为**目标线程**。目标线程的取消可在如下两种情况下发生：

1. **异步取消**(asynchronous cancellation)：一个线程立即终止目标线程。

2. **延迟取消**(deferred cancellation)：目标线程不断地检查它是否应终止，这允许目标线程有机会按着有序方式来终止自己。

如果资源已分配给被要取消的线程或要取消的线程正在更新与其他线程所共享的数据，那么取消在这些情况下就会有困难。对于异步取消而言这会变得尤其麻烦。操作系统通常收回取消线程的系统资源，但是往往并不收回所有资源。因此，异步取消线程并不会使所需的系统资源空闲。

另外，一个线程显示目标线程要被取消时，就会发生延迟取消。不过，只有当目标线程检查以确定它是否应该取消时才会发生取消。这允许一个线程检查它是否是在安全的点被取消。Pthread 称这些点为**取消点**(cancellation point)。

绝大多数操作系统允许进程或线程被异步取消。不过，Pthread API 提供了延迟取消。这意味着实现 Pthread API 的操作系统允许延迟取消。

5.3.3 信号处理

信号在 UNIX 系统中用做通知进程某个特定事件已经发生了。根据来源和被通知信号的事件理由，信号可以被同步或异步接收。不管信号是同步或异步的，所有信号具有同样模式：

1. 信号是由特定事件的发生所产生的。

2. 产生的信号要发送到进程。

3. 一旦发送，信号必须要加以处理。

同步信号的例子包括非法内存访问或被零所除。在这种情况下，如果运行程序执行这些动作中的任何一个，那么就产生信号。同步信号发送到执行操作而产生信号的同一进程(这就是为什么它们被认为是同步的)。

当一个信号是由运行进程之外的事件所产生，那么进程就异步地接收这一信号。这种信号的例子包括用特殊键(比如 Ctrl-C)或定时器时间到期以终止进程。通常，异步信号被发送到另一个进程。

每个信号可能由两种可能的处理程序中的一种来被处理：

1. 缺省信号处理程。

2. 用户定义的信号处理程序。

每个信号都有**缺省信号处理程序**(default signal handler)，这是当处理信号时由内核运行的。这种缺省动作可以用**用户定义的信号处理程序**函数来改写。在这种情况下，用户定义函数被调用用来处理信号，而不是缺省的动作。同步和异步信号可按不同方式被处理。有的信号可以被忽略(如改变窗口大小)；其他的可能要通过终止程序来处理(如非法内存访问)。

单线程程序的信号处理比较直接：信号总是发送给进程。不过，对于多线程程序，发送信号就比较复杂，因为进程可能有多个线程。那么信号应被发送到哪里呢？

通常存在如下选择：

1. 发送信号到信号所应用的线程。
2. 发送信号到进程内的每个线程。
3. 发送信号到进程内的某些线程。
4. 规定一个特定线程以接收进程的所有信号。

发送信号的方法依赖于所产生信号的类型。例如，同步信号需要发送到产生这一信号的线程，而不是进程中的其他线程。不过，对于异步信号，情况就不是那么清楚了。有的异步信号如终止进程的信号（如 Ctrl-C）应该发送到所有线程。有的多线程版 UNIX 允许线程描述它会接收什么信号和拒绝什么信号。因此，有些异步信号只能发送给那些不拒绝它的线程。不过，因为信号只能被处理一次，所以信号通常被发送到进程中不拒绝它的第一个线程。Solaris 2 按第四种方法处理：它在每个进程内创建了一个专门只处理信号的线程。当异步信号被发送到一个进程时，它被发送到该特定线程，进而再将信号传递给第一个不拒绝它的线程。

虽然 Windows 2000 并不明确地提供对信号的支持，但是它们能通过**异步过程调用**（asynchronous procedure call，APC）来模拟。APC 工具允许用户线程指定一个函数以便在用户线程收到特定事件通知时能被调用。正如其名称所表示的，APC 与 UNIX 的异步信号大体相当。不过，UNIX 需要力争解决如何处理多线程环境的信号，而 APC 机制则较为直接，因为 APC 只发送给特定线程而不是进程。

5.3.4 线程池

在 5.1 节中描述了对网络服务器进行多线程编程的情况。在这种情况下，每当服务器收到请求，它就创建一个独立线程以处理请求。虽然创建一个独立线程当然要比创建一个独立进程要好，但是多线程服务器也有一些潜在问题。第一个是关于在处理请求之前用以创建线程的时间，以及线程在完成其工作之后就要被丢弃这一事实。第二个问题更为麻烦：如果允许所有并发请求都通过新线程来处理，那么并没有限制在系统中并发执行的线程的数量。无限制的线程会用尽系统资源如 CPU 时间或内存。对这个问题的解决方法之一是使用**线程池**（thread pool）。

线程池的主要总体思想是在进程开始时创建一定数量的线程，并放入到池中坐以等待工作。当服务器收到请求时，它会唤醒池中的一个线程（如果有可用的线程），并将要处理的请求传递给它。一旦线程完成了它的服务，它会返回到池中再等待更多工作。如果池中没有可用的线程，那么服务器会一直等待直到有空线程为止。

具体来说，线程池有如下优点：

1. 通常用现有线程处理请求要比等待创建新的线程要快。

2. 线程池限制了在任何时候可存在线程的数量。这对那些不能支持大量并发线程的系统尤为重要。

池中线程数量可以通过一些因素如系统中 CPU 的数量、物理内存的大小和并发客户的请求的期望值来启发地设置。更为高级的线程池的体系结构能动态调整池中线程的数量以适应使用情况。这类体系结构提供了较小池的又一个好处，即在系统负荷低时减低内存消耗。

5.3.5 线程特定数据

同属一个进程的线程共享进程数据。事实上，这种数据共享提供了多线程编程的一种好处。不过，在有些情况下每个线程可能需要一定数据的自己的拷贝。这种数据称做**线程特定数据**(thread specific data)。例如，对于事务处理系统，可能需要通过独立线程以处理各个请求。而且，每个事务都有一个惟一标识符。为了让每个线程与其惟一标识符相关联，可以使用线程特定数据。绝大多数线程库，包括 Win32 和 Pthread，都提供了对线程特定数据的一定形式的支持。Java 也提供这种支持。

5.4 Pthread 线程

Pthread 是指 POSIX 标准(IEEE 1003.1c)定义的，它定义线程创建 API 和同步 API。这是线程行为的*规范*，而不是实现。操作系统设计者可以根据意愿来采取任何实现形式。通常，实现 Pthread 规范的库局限于基于 UNIX 的系统如 Solaris 2。Windows 操作系统通常并不支持 Pthread，尽管在公共域中有共享版存在。

在本节引入一些 Pthread API 来作为用户级线程库的例子。之所以称之为用户级线程库，这是由于在由 Pthread API 所创建的线程和任何相关内核线程之间并不存在明显的联系。图 5.5 所示的 C 程序显示一些基本 Pthread API 以构造一个多线程程序。如果您想对 Pthread API 有更详细的了解，请参考本章的推荐读物。

```
#include<pthread.h>
#include<stdio.h>

int sum;/* this data is shared by the thread(s)*/
void runner(void *param);/* the thread */

main(int argc,char *argv[])
{
   pthread_t tid;/* the thread identifier*/
   pthread_attr_t attr;/* set of thread attributes */
   if(argc != 2){
        fprintf(stderr,"usage:a.out <integer value>\n");
```

```
        exit();
    }
    if (atoi(argv[1])<0){
        fprintf(stderr," %d must be >= 0\n",atoi(argv[1]));
        exit();
    }
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* now wait for the thread to exit */
    pthread_join(tid,NULL);
    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int upper = atoi(param);
    int i;
    sum = 0;
    if (upper>0){
        for(i = 1;i<= upper;i++)
            sum += i;
    }
    pthread_exit(0);
}
```

图 5.5 使用 Pthread API 的多线程 C 程序

图 5.5 所示的程序创建了一个独立线程以确定非负整数的累加和。对于 Pthread 程序，独立线程是通过特定函数而开始执行的。在图 5.5 中，这个特定函数是 runner 函数。当程序开始，单个控制线程在 main 开始。在一定的初始化之后，main 创建了第二个线程以在 runner 函数中开始控制。

现在，对该程序作一个更为详细的概述。所有 Pthread 程序都必须包括 pthread.h 头文件。语句 pthread_t tid 定义了将要创建线程的标识符。每个线程都有一组属性包括堆栈大小和调度信息。pthread_attr_t attr 声明代表了线程的属性。可以通过函数调用 pthread_attr_init(&attr)来设置这些属性。因为没有明确地设置任何属性，所以使用了所提供的缺省属性。通过 pthread_create 函数调用可创建一个独立线程。除了传递线程标识符和线程属性外，也传递函数名称(这里为 runner)以便新线程可以开始执行。最后，传递由命令行所提供的整数参数 argv[1]。

程序此时有两个线程：main 的初始线程和通过 runner 函数执行累加和的线程。在创建

了第二个线程之后，main线程通过调用 pthread_join 函数来等待 runner 线程的完成。线程 runner 在调用了函数 pthread_exit 之后就完成了。

5.5 Solaris 2 线 程

Solaris 2 是一种 UNIX 版本，它提供了对内核和用户级线程、SMP、实时调度等的支持。Solaris 2 有一个包括线程创建和管理的 API 库用于支持用户级线程（称为 *UI 线程*），还实现了 Pthread API(5.4 节)。这两个库之间的差别不大，尽管绝大多数程序员现在都选择使用 Pthread 库。Solaris 2 也定义了中间级线程。在用户级线程和内核级线程之间是轻量级进程(LWP)。每个进程至少有一个 LWP。线程库在进程的 LWP 池上多路切换用户级线程，只有当前与 LWP 相关联的用户级线程才能完成工作。其他线程或是阻塞或是等待 LWP 以便在其上执行。

标准内核级线程在内核内执行所有操作。每个 LWP 都有一个内核级线程，有的内核级线程为内核而运行，所以没有相关的 LWP(如处理磁盘请求的线程)。内核级线程是系统内所调度的惟一对象(第六章)。Solaris 2 实现了多对多模型；其整个线程系统如图 5.6 所示。

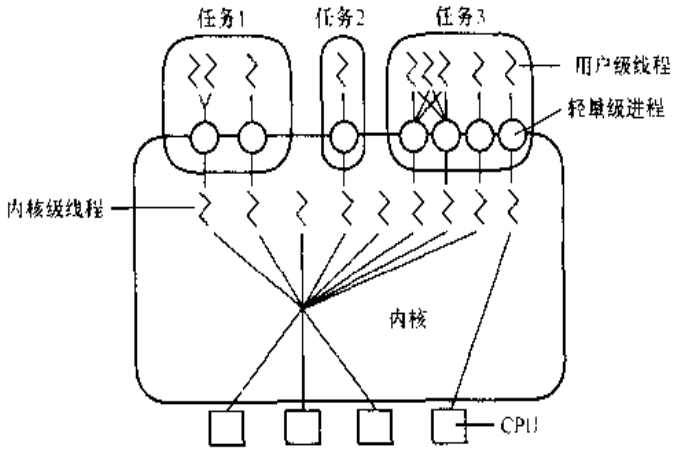


图 5.6 Solaris 2 线程

用户级线程可能是约束的也可能是非约束的。**约束的**用户级线程永远与 LWP 相连。只有该线程能运行在 LWP 上，根据请求该 LWP 可以专门在一个处理器上运行(见图 5.6 最右的线程)。约束线程在许多情况下有用，即需要快速响应时间如实时应用程序。**非约束**线程并不永久地与 LWP 相关联。应用程序的所有非约束线程可以在该应用程序的可用 LWP 池上进行多路复用。线程缺省为非约束的。Solaris 8 也提供了另外一个线程库，它缺省地将所有线程约束到一个相关 LWP 上。

考虑一个工作系统：任何一个进程可能有多个用户级线程。这些用户级线程通过线程库而无需内核干预就可在 LWP 上调度和切换。用户级线程极为高效，这是因为线程创建或

删除无需内核支持，线程库就能完成从一个用户级线程上下文切换到另一个线程。

每个 LWP 只能与一个内核级线程相连，而每个用户级线程是独立于内核的。一个进程可有许多 LWP，但是只有线程需要与内核通信时才需要这些 LWP。例如，对于在系统调用时可能并发阻塞的每个线程都需要一个 LWP。考虑有五个不同文件读请求可能同时发生的情况，此时就需要五个 LWP，因为每个都需要等待内核 I/O 的完成。如果任务只有四个 LWP，那么第五个请求必须等待一个 LWP 从内核中返回。如果只有可被五个 LWP 所用的工作，那么增加第六个 WLP 也不会增加什么。

内核线程是由内核调度程序进行调度的，在系统的(多个)CPU 上执行。如果内核线程阻塞(如等待 I/O 操作的完成)，那么处理器可用于运行其他内核线程。如果阻塞线程是为 LWP 而运行的，那么 LWP 也会阻塞。同理，当前与该 LWP 相连的用户级线程也会阻塞。如果进程有多个 LWP，那么内核可调度另一个 LWP。

线程库动态地调整池中的 LWP 数量以确保应用程序的最佳性能。例如，如果一个进程的所有 LWP 都已被阻塞，并且其他线程能运行，那么线程库会自动创建另一个 LWP 以赋给等待线程。因此，一个程序不会因为缺乏被非阻塞 LWP 而被阻塞。而且，如果 LWP 没有被使用，那么 LWP 是需要维护的昂贵的内核资源。如果 LWP 在很长时间(通常是 5 分钟)内没有被使用，线程库会使 LWP"变老"，并删除它们。

开发人员使用如下数据结构实现了 Solaris 2 上的线程：

• 用户级线程包括线程 ID；寄存器集合(包括程序计数器和堆栈指针)；堆栈；优先权(线程库用于调度目的)。所有这些数据结构都不是内核资源；它们都存在于用户空间内。

• LWP 有它所执行的用户级线程的寄存器集合，还有内存和记账信息。LWP 是内核数据结构，驻留在内核空间上。

• 内核线程只有少量数据结构和一个堆栈。该数据结构包括内存寄存器的拷贝，所关联的 LWP 的指针，优先权和调度信息。

Solaris 2 的每个进程都有许多信息包括在进程控制块(PCB)中，参见 4.1.3 小节。具体来说，每个 Solaris 2 进程都包含一个进程 ID(PID)；内存图；打开文件列表；优先权；与该进程关联的内核线程列表的指针(图 5.7)。

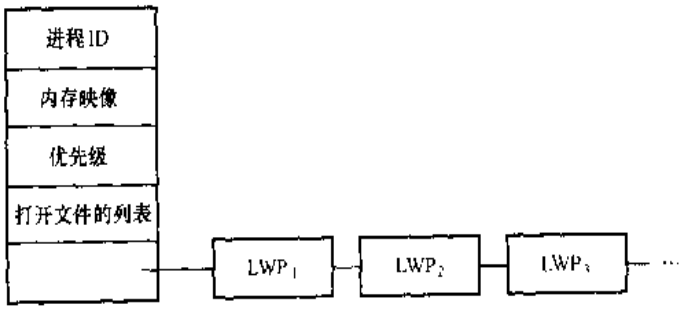


图 5.7 Solaris 2 进程

5.6 Windows 2000 线 程

Windows 2000 实现了 Win32 API。Win32 API 是 Microsoft 操作系统家族(如 Windows 95/98/NT 和 Windows 2000)的主要 API。事实上,本节所讨论的大部分内容也适用于这类操作系统。

一个 Windows 应用程序以独立进程方式运行,每个进程可包括一个或多个线程。Windows 2000 使用了如 5.2.2 小节所述的一对一映射,以将每个用户级线程映射到相关内核线程。不过,Windows 也提供了对 **fiber** 库的支持,该库提供了多对多模型(5.2.3 小节)功能。同属一个进程的每个线程都能访问进程的虚拟地址空间。

线程通常包括如下部分:

- 一个线程 ID 以惟一标识线程。
- 一组寄存器集合以表示处理器状态。
- 一个用户堆栈以便供线程在用户模式下运行时所用。同样地,每个线程也有一个内核堆栈。堆栈供线程在内核模式下运行时所用。
- 一个私有存储区域,为各种运行时库和动态链接库(DLL)所用。

寄存器集合、堆栈和私有存储区域通常称为线程的 *关联*,它与操作系统所运行的硬件的特定体系结构有关。线程的主要数据结构包括:

- ETHREAD(执行线程块)
- KTHREAD(内核线程块)
- TEB(线程执行环境块)

ETHREAD 构件主要包括线程所属进程的指针和线程开始控制的子程序的地址。ETHREAD 还包括相应的 KTHREAD 的指针。

KTHREAD 包括线程的调度和同步信息。另外,KTHREAD 也包括内核堆栈(以供线程在内核模式下运行时所用)和 TEB 的指针。

ETHREAD 和 KTHREAD 完全处于内核空间中;这意味着只有内核才可以访问它们。TEB 是用户空间的数据结构,可供线程在用户模式下运行时访问。TEB 在许多其他域中,包括用户模式堆栈和用于线程特定数据的数组(Windows 称之为 *线程本地存储*)。

5.7 Linux 线 程

Linux 内核在 2.2 版中引入了线程机制。Linux 提供了 fork,这是具有传统进程复制功能的系统调用。Linux 还提供了系统调用 clone,其功能类似于创建一个线程。clone 与 fork 的行为很相似,它不是创建调用进程的复制,而是创建一个独立进程以共享原来调用进

程的地址空间。通过共享父进程的地址空间，clone 任务能像独立线程一样工作。

由于 Linux 内核进程的特定表示方式，所以允许共享地址空间。系统内的每个进程都有一个唯一的内核数据结构。不过，该数据结构并不保存该数据结构中进程本身的数据，而是保存了此数据保存处的数据结构的指针。例如，每个进程的数据结构都包括其他数据结构（如表示打开文件列表、信号处理信息和虚拟内存等）的指针。当调用 fork 时，就创建了新进程，它具有父进程的所有相关数据结构的*拷贝*。当调用 clone 时，也创建了新进程。但是，新进程并不是复制所有数据结构，而是*指向*了父进程的数据结构，从而允许子进程共享父进程的内存和其他进程资源。作为系统调用 clone 的参数，可传递一些标记的集合。这个标记集用来指出父进程有多少内容被子进程共享了。如果没有设置标记，那么就没有共享且 clone 与 fork 一样动作。如果设置了所有五个标记，那么子进程与父进程就共享一切。其他不同标记的组合形成了这两极之间的不同程度的共享。

有趣的是，Linux 并不区分进程和线程。事实上，Linux 在讨论程序内的控制流时，通常称之为*任务*而不是进程或线程。除了克隆进程外，Linux 并不支持多线程、独立数据结构或内核子程序。不过，各种 Pthread 实现可以用于用户级多线程编程。

5.8 Java 线 程

正如大家所看到的，通过使用如 Pthread 的库可以在用户级提供对线程的支持。再者，绝大多数操作系统也在内核级提供对线程的支持。Java 是在语言级提供了线程创建和管理支持功能的为数不多的一种语言。不过，由于线程是由 Java 虚拟机（JVM）而不是由用户级库或内核来管理的，所以很难清楚地将 Java 线程划归为用户级或内核级。在本节，将 Java 线程作为严格用户级或内核级模型的另一种选择。在本节的后面，将讨论 Java 线程是如何映射到底层内核线程的。

所有 Java 程序至少由一个控制线程组成。即使一个只包含 main 方法的简单 Java 程序也是在 JVM 中作为一个单独线程运行的。另外，Java 提供了命令以允许开发人员在程序中创建和管理其他控制线程。

5.8.1 线程创建

明确地创建线程的一种方法是创建一个新类以继承类 Thread，并改写类 Thread 的方法 run。这种方法如图 5.8 所示，这是一个用来确定非负整数累加和的多线程程序的 Java 版本。

```
class Summation extends Thread
{
   public Summation(int n);
       upper = n;
```

```
    public void run(){
        int sum = 0
        if(upper>=0){
            for (int i = 1; i<= upper; i++)
                sum += i;
        }
        System.out.println("The sum of" + upper + "is " + sum);
    }
    private int upper;
}

public class TheradTester;

{
    public static void main(String[] args){
        if(args.length>0){
            if(Integer.parseInt(args[0])<0)
                System.err.println(args[0] + "must be>=0. ");
            else{
                Summation thrd = new Summation(Integer.parseInt(args[0]));
                thrd.start();
            }
        }
        else
            System.err.println("Usage:Summation<integer value>");
        }
    }
```

图 5.8 非负整数累加和的 Java 程序

这一派生类的对象可作为独立控制线程在 JVM 中运行。不过，创建从类 Thread 派生的对象并不特别地创建新线程；只有在调用 start 方法之后，才真正创建了新线程。调用新对象的 start 方法能做如下两件事：

1. 它在 JVM 中分配内存并初始化新线程。

2. 它调用 run 方法，以便线程能合法地为 JVM 所运行。（注意你不能直接调用 run 方法。而是要调用 start 方法，它会为你调用 run 方法。）

当累加程序运行时，JVM 创建了两个线程。第一个是与应用程序相关的线程，该线程通过方法 main 开始执行。第二个线程是通过显式调用 start 方法而创建的线程 Summation。线程 Summation 通过其 run 方法开始执行。线程从此 run 方法中退出时终止运行。

5.8.2 JVM 与主机操作系统

典型的 JVM 实现建立在主机操作系统之上。这种设置允许 JVM 隐藏支撑操作系统的

实现细节，并创建了一个一致的、抽象的环境以允许 Java 程序能在任何支持 JVM 的平台上运行。JVM 规范并没有指出 Java 线程如何映射到支撑操作系统，而是留给特定的 JVM 实现来决定。Windows 95/98/NT 和 Windows 2000 使用了一对一模型；因此，运行在这些操作系统上的 JVM 的每个线程映射到一个内核线程。Solaris 2 开始是用多对一模型（*green thread*）实现了 JVM。不过，从 Solaris 2.6 上的 JVM 1.1 版本开始，它采用了多对多模型来实现。

5.9 小 结

线程是进程内的控制流。多线程进程在同一地址空间内包括多个不同的控制流。多线程程序设计的优点包括对用户响应的改进、进程内的资源共享、经济和利用多处理器体系结构的能力。

用户级线程对程序员来说是可见的，而对内核来说是未知的。用户空间的线程库通常管理用户级线程。操作系统内核支持和管理内核级的线程。通常，用户级线程同内核线程相比，创建和管理速度要更快。有三种类型不同模型将用户线程和内核线程关联起来：多对一模型将许多用户线程映射到一个内核线程；一对一模型将每个用户线程映射到一个相应的内核线程；多对多模型将多个用户线程在同样（或更少）数量的内核线程之间多路复用。

多线程程序为程序员带来了许多挑战，包括系统调用 fork 和 exec 的语义。其他事项包括线程取消、信号处理和线程特定数据。许多现代操作系统提供了对线程的内核支持；这其中有 Windows NT、Windows 2000、Solaris 2 和 Linux。Pthread API 提供了一组函数用于在用户级创建和管理线程。Java 提供了相似的 API 以支持线程。不过，由于 Java 线程是由 JVM，而不是由用户级线程库或内核来管理的，它们不属于用户级线程或内核级线程类。

习 题 五

5.1 举两个多线程程序设计的例子来说明多线程比单线程方案提高性能。

5.2 举两个多线程程序设计的例子来说明多线程不比单线程方案提高性能。

5.3 用户级线程与内核级线程的两个不同点是什么？在什么情况下一种类型比另一种类型更好？

5.4 描述一下内核采用取行动进行内核级线程上下文切换的过程。

5.5 描述一下线程库采取行动进行用户级线程上下文切换的过程。

5.6 当一个线程被创建时使用了哪些资源？这和一个进程被创建时所采用的资源有何不同？

5.7 假设一个操作系统将用户级线程用多对多模型通过 LWP 映射到内核，而且系统允许开发人员创建实时线程，是不是必须将一个实时线程绑定到一个 LWP 上？解释你的答案。

5.8 写一个多线程的 Pthread 或 Java 程序来生成 Fibonacci 序列。这个程序要做到以下几点：用户运行程序时在命令行输入程序所要产生的 Fibonacci 序列的数，然后创建一个独立的线程来生成 Fibonacci

数。

5.9 写一个多线程的 Pthread 或 Java 程序来输出素数。这个程序要做到以下几点：用户运行程序时在命令行输入一个数字，然后程序创建一个独立新的线程来输出小于或等于用户所输入数的所有素数。

推荐读物

Anderson 等[1986]论述了线程性能的问题，在后来的 Anderson 等人[1991]论著中对内核支持的用户级线程的性能进行了评估。Marsh 等[1991]论述了第一级用户级线程。Bershad 等[1990]描述了结合使用线程与 RPC。Draves 等[1991]论述了使用连续性实现操作系统中线程管理与通信。Engleschall[2000]论述了支持用户级线程的一个技巧。Ling 等[2000]的著作中有最佳线程池大小的分析。

IBM OS/2 操作系统是一个运行在个人计算机(Kogan 和 Rawson[1988])上的多线程操作系统。Peacock[1992]论述了 Solaris 2 中文件系统的多线程程序设计。Vahalia[1996]论述了许多版本的 UNIX 线程程序设计问题。Mauro 和 McDougall[2001]描述了 Solaris 2 内核中的线程程序设计的新发展。Zabatta 和 Young[1998]比较了在对称多处理器上的 Windows NT 和 Solaris 2 的线程。

Lewis、Berg[1998]、Sun 和 Kleiman 等[1996]给出了多线程编程的信息，虽然这些资料偏向于 Pthread。Oaks、Wong[1999]、Lea 和 Hartley[1998]论述了 Java 中的多线程程序设计。Solomon[1998]和 Solomon、Russinovich[2000]分别描述了 Windows NT 和 Windows 2000 中怎样实现线程。Beveridge 和 Wiener[1997]论述了用 Win32 进行多线程编程。Pham 和 Garg[1996]描述了用 Windows NT 进行多线程编程。

第六章 CPU 调 度

CPU 调度是多道程序操作系统的基础。通过在进程之间切换 CPU，操作系统可以提高计算机的生产效率。在本章，介绍基本调度概念并展示多个不同 CPU 调度算法。还要研究为特定系统选择算法的问题。

6.1 基本概念

多道程序设计的目标是在任何时候都有一个进程在运行，以使 CPU 使用率最大化。在单处理器系统中，每次只允许一个进程运行；任何其他进程必须等待直到 CPU 空闲且能被调度为止。

多道程序设计的思想较为简单。进程被执行直到它必须等待，通常等待某个 I/O 请求的完成。在一个简单计算机系统中，CPU 就会因此空闲；所有这些等待时间就浪费了。采用了多道程序设计，试图有效地使用这一时间。多个进程可同时处于内存中。当一个进程必须等待时，操作系统会从该进程拿走 CPU 控制权，并将 CPU 交给其他进程。这种方式会继续。

调度是操作系统的基本功能。几乎所有计算机资源在使用前都要被调度。当然，CPU 是最为重要的计算机资源之一。因此，CPU 调度对于操作系统设计来说非常重要。

6.1.1 CPU-I/O 区间周期

CPU 的成功调度依赖于进程的如下观测属性：进程执行由 CPU 执行和 I/O 等待**周期**组成。进程在这两个状态之间切换。进程执行从 **CPU 区间**(CPU burst)开始。在这之后是 **I/O 区间**(I/O burst)，接着是另一个 CPU 区间，接着另一个 I/O 区间，如此进行下去。最终，最后的 CPU 区间通过系统请求终止执行，而不是又一个 I/O 区间(图 6.1)。

图 6.1 CPU 区间和 I/O 区间的交替序列

这些 CPU 区间的长度已经被大量地测量过。虽然它们随着进程和计算机的不同变化很大，但是它们都呈现出类似于图 6.2 所示的频率曲线。该曲线通常表征为指数或超指数形式，

具有大量短 CPU 区间和少量长 CPU 区间。I/O 约束程序通常具有很多短 CPU 区间。CPU 约束程序可能有少量的长 CPU 区间。这种分布能有助于人们选择合适的 CPU 调度算法。

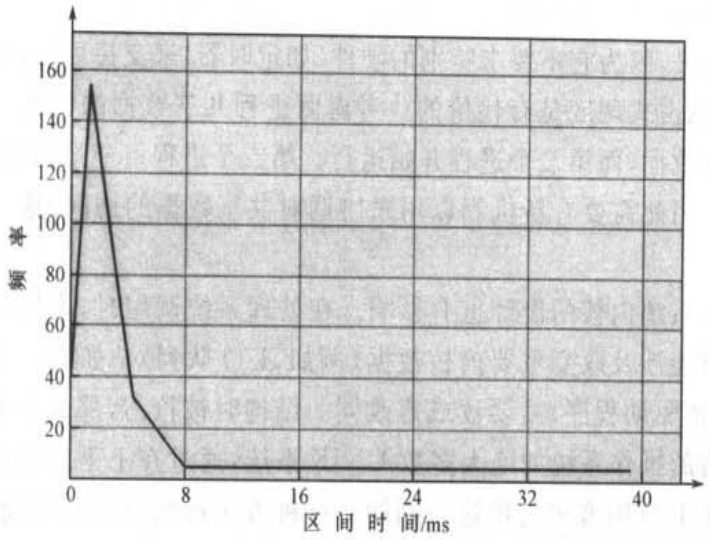


图 6.2 CPU 区间时间直方图

6.1.2 CPU 调度程序

每当 CPU 变为空闲时，操作系统就必须从就绪队列中选择一个进程来执行。进程选择由**短期调度程序**(short-term scheduler，或 CPU 调度程序)执行。调度程序从内存中就绪可执行的进程里选择一个，并为其中之一分配 CPU。

就绪队列不必是先进先出(FIFO)队列。正如研究各种调度算法时将看到的，就绪队列可实现为 FIFO 队列、优先队列、树或仅仅是无序链表。不过，从概念上来说，就绪队列内的所有进程都要排队以等待机会在 CPU 上运行。队列中的记录通常是进程的进程控制块(process control block，PCB)。

6.1.3 可抢占式调度

CPU 调度决策可在如下四种环境下发生：

1. 当一个进程从运行状态切换到等待状态(例如，I/O 请求或调用 wait 以等待一个子进程的终止)。

2. 当一个进程从运行状态切换到就绪状态(例如，当出现中断时)。

3. 当一个进程从等待状态切换到就绪状态(例如，I/O 完成)。

4. 当一个进程终止时。

对于第一和第四两种情况，就调度而言没有选择。一个新进程(如果就绪队列有一个进程存在)必须被选择执行。不过，对于第二和第三两种情况，可以进行选择。

当调度只能发生在第一和第四两种情况时，称调度方案是**非抢占**(nonpreemptive)的；否则，调度方案是**可抢占**(preemptive)的。采用非抢占调度，一旦 CPU 被分配给一个进程，那么该进程会一直使用 CPU 直到进程终止或切换到等待状态时释放 CPU。微软公司 Windows 3.1 和 Apple Macintosh 操作系统都使用这种调度方法。在有的硬件平台上，这是惟一可以使用的方法，因为它不要求特别的硬件(如定时器)来支持可抢占式调度。

遗憾的是，可抢占式调度是有代价的。考虑两进程共享数据的情况。第一个进程被抢占时可能正在更新数据，而第二个进程开始运行。第二个进程可能试图读数据，该数据现在处于不一致状态。因此需要有新机制以用来协调对共享数据的访问；这一问题将在第七章中讨论。

抢占对于操作系统内核的设计也有影响。在处理系统调用时，内核可能忙于为进程而活动。这些活动可能涉及改变重要内核数据(例如，I/O 队列)。如果一个进程在进行这些修改且内核(或设备驱动程序)需要读或修改同一结构时被抢占，那么会有什么结果呢？肯定会导致混乱。有的操作系统如绝大多数 UNIX 系统，通过在上下文切换之前等待系统调用完成或等待发生 I/O 阻塞来处理这一问题。这种方法确保内核结构简单，因为在内核数据结构处于不一致状态时，内核不会抢占一个进程。遗憾的是，这种内核执行模型对实时计算和多处理器的支持较差。这些问题及其解决方案将在 6.4 节和 6.5 节中讨论。

对 UNIX 的情况来说，代码段仍处于危险。因为根据定义中断可以随时发生，而且不能总是被内核所忽视，所以受中断影响的代码段必须加以保护以避免同时使用。操作系统需要在几乎所有时候都能接受中断，否则输入会被丢失或输出会被改写。为了这些代码段不被多个进程同时访问，所以在进入时要禁止中断而在退出时要重新允许中断。遗憾的是，禁止中断和允许中断很花费时间，尤其是对于多处理器系统。为了让系统可有效地*利用*多个 CPU，必须尽量减少中断状态的改变而尽量增加锁的细化程度。例如，对 Linux 的可扩展性是一个挑战。

6.1.4 分派程序

与 CPU 调度功能有关的另一个部分是**分派程序**。分派程序是一个模块，用来将 CPU 的控制交给由短期调度程序所选择的进程。其功能包括：

- 切换上下文
- 切换到用户模式
- 跳转到用户程序的合适位置以重新启动这个程序

分派程序应尽可能快，因为在每次进程切换时都要使用。分派程序停止一个进程而启动另一个进程执行所要花费的时间称为**分派延迟**(dispatch latency)。

6.2 调度准则

不同的CPU调度算法具有不同属性，且可能对某些进程更为有利。为了选择算法以适用于特定情况，必须分析各个算法的属性。

为了比较CPU调度算法，分析员提出了许多准则，用来进行比较的特征对确定最佳算法有很大影响。这些准则包括如下：

- **CPU使用率**：需要使CPU尽可能忙。CPU使用率从0%到100%。对于真实系统，它应从40%（轻负荷系统）到90%（重负荷使用的系统）。
- **吞吐量**：如果CPU忙于执行进程，那么就要评估其工作量。其中一种测量工作量的方法称为**吞吐量**，它指一个时间单元内所完成进程的数量。对于长进程，吞吐量为每小时一个进程；对于短进程事务，吞吐量可能为每秒十个进程。
- **周转时间**：从一个特定进程的角度来看，重要准则是运行该进程需要花费多长时间。从进程提交到进程完成的时间间隔称为周转时间。周转时间是所有时间段之和，包括等待进入内存、在就绪队列中等待、在CPU上执行和I/O执行。
- **等待时间**：CPU调度算法并不影响进程运行和执行I/O的时间量；它只影响进程在就绪队列中等待所花费的时间。等待时间是在就绪队列中等待所花时间之和。
- **响应时间**：对于交互式系统，周转时间并不是最佳准则。通常，进程能相当早就产生某些输出，并能继续计算新结果同时输出以前的结果给用户。因此，另一时间度量是从提交请求到产生第一响应的时间。这种度量称为**响应时间**，是开始响应所需要的时间，而不是输出该响应所需要的时间。周转时间通常受输出设备速度的限制。

人们需要使CPU使用率和吞吐量最大化，而使周转时间、等待时间和响应时间最小化。在绝大多数情况下，要优化平均度量值。不过，在有的情况下，需要优化最小值或最大值，而不是平均值。例如，为了保证所有用户都得到好的服务，可能需要使最大响应时间最小。

对于交互式系统（如分时系统），有的分析人员建议最小化响应时间的**差异**要比最小化平均响应时间更为重要。具有合理的**可预见的**响应时间的系统，比平均来说更快但变化大的系统更为可取。不过，在CPU调度算法如何使差异最小化方面，所做的工作并不多。

在讨论各种CPU调度算法时，需要描述其操作。精确描述需要涉及许多进程，且每个进程有数百个CPU区间和I/O区间的序列。为了简化描述，在本书的例子中，只考虑每个进程有一个CPU区间（以ms计）。本书所比较的量是平均等待时间。更为精确的评价机制将在6.6节中讨论。

6.3 调度算法

CPU调度处理从就绪队列中选择哪个进程并为之分配CPU的问题。在本节，描述一

些现有的 CPU 调度算法。

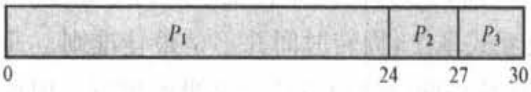
6.3.1 先到先服务调度

显然，最简单的 CPU 调度算法是**先到先服务调度算法**（first-come, first-served, FCFS）。采用这种方案，先请求 CPU 的进程被首先分配到 CPU。FCFS 策略可以用 FIFO 队列来容易地实现。当一个进程进入到就绪队列，其 PCB 被链接到队列的尾部。当 CPU 空闲时，CPU 被分配给位于队列头的进程。接着，该运行进程从队列中被删除。FCFS 调度的代码编写简单且容易理解。

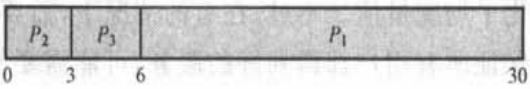
不过，采用 FCFS 策略的平均等待时间通常相当长。考虑如下一组进程，它们在时刻 0 到达，其 CPU 区间时间长度按 ms 计：

进程	区间时间
P_1	24
P_2	3
P_3	3

如果进程按 P_1、P_2、P_3 的顺序到达，且按 FCFS 的顺序被处理，那么得到如下面 Gantt 图所示的结果：



进程 P_1 的等待时间为 0 ms，进程 P_2 的等待时间为 24 ms，进程 P_3 的等待时间为 27 ms。因此，平均等待时间为(0+24+27)/3=17 ms。不过，如果进程按 P_2、P_3、P_1 的顺序到达，那么其结果如下面 Gantt 图所示：



现在平均等待时间为 (6 + 0 +3)/3 = 3 ms。这一减少量很大。因此，采用 FCFS 策略的平均等待时间通常不是最小，且如果进程 CPU 区间时间变化很大，平均等待时间也会变化很大。

另外，考虑 FCFS 调度在动态情况下的性能。假设有一个 CPU 约束进程和许多 I/O 约束进程。随着进程在系统中运行，可能会发生如下情况：CPU 约束进程会得到 CPU 并控制它。在这段时间内，所有其他进程会处理完它们的 I/O 并转移到就绪队列以等待 CPU。当这些进程在就绪队列里等待时，I/O 设备空闲。最终，CPU 约束进程完成其 CPU 区间并移动到 I/O 设备。所有 I/O 约束进程，由于只有很短的 CPU 区间，故很快执行完并移回到 I/O 队列。这时，CPU 保持空闲。之后，CPU 约束进程会移回到就绪队列并被分配到 CPU。再次，所有 I/O 进程会在就绪队列中等待直到 CPU 约束进程的完成。由于所有其他进程都

等待一个大进程释放 CPU，就会产生**护航效果**(convoy effect)。与可能允许较短进程先行相比，这种效果会导致 CPU 和设备的使用率变得更低。

FCFS 调度算法是非抢占的。一旦 CPU 被分配给了一个进程，该进程就会保持 CPU 直到释放 CPU 为止，即程序终止或是请求 I/O。FCFS 算法对于分时系统(每个用户需要定时地得到一定的 CPU 时间)是尤为麻烦的。允许一个进程持有 CPU 的时间过长，将是个严重错误。

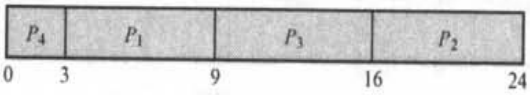
6.3.2 最短作业优先调度

另一个 CPU 调度方法是**最短作业优先调度算法**(shortest-job-first，SJF)。这一算法将每个进程与其下一个 CPU 区间段相关联。当 CPU 为可用时，它会赋给具有最短后续 CPU 区间的进程。如果两个进程具有同样长度的 CPU 区间，那么可以使用 FCFS 调度来处理。注意一个更为适当的表示是*最短下一个 CPU 区间*，这是因为调度的完成是通过检查进程的下一个 CPU 区间的长度，而不是其总长度。可以使用词 SJF，这是因为绝大多数人员和教科书称这种类型的调度策略为 SJF。

作为一个例子，考虑如下一组进程，其 CPU 区间时间长度以 ms 计：

进程	区间时间
P_1	6
P_2	8
P_3	7
P_4	3

采用 SJF 调度，就能根据如下 Gantt 图来调度这些进程：



进程 P_1 的等待时间是 3 ms，进程 P_2 的等待时间为 16 ms，进程 P_3 的等待时间为 9 ms，进程 P_4 的等待时间为 0 ms。因此，平均等待时间为(3 + 16 + 9 +0)/4 = 7 ms。如果使用 FCFS 调度方案，那么平均等待时间为 10.25 ms。

SJF 调度算法可证明为*最佳*，这是因为对于给定的一组进程，SJF 算法的*平均*等待时间最小。通过将短进程移到长进程之前，短进程等待时间的减少大于长进程等待时间的增加。因而，平均等待时间减少了。

SJF 算法的真正困难是如何知道下一个 CPU 请求的长度。对于批处理系统的长期(或作业)调度，可以将用户提交作业时所指定的进程时间极限作为长度。因此，用户有根据地精确估计进程时间，这是因为较低的值可能意味着更快的响应。(过小的值会引起时间极限超出错误，并需要重新提交)。SJF 调度经常用于长期调度。

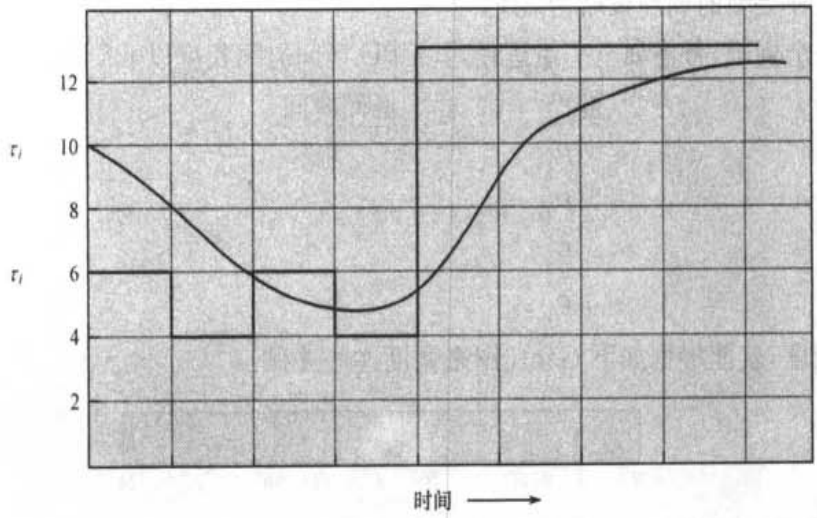
虽然 SJF 算法是最佳的，但是它不能在短期 CPU 调度的层次上加以实现。没有办法知

道下一个 CPU 区间的长度。一种方法是试图近似 SJF 调度。虽然人们不*知道*下一个 CPU 区间的长度，但是可以*预测*它的值。可以认为下一个 CPU 区间的长度与以前的相似。因此，通过计算下一个 CPU 区间长度的近似值，能选择具有最短预测 CPU 区间的进程来运行。

下一个 CPU 区间通常可预测为以前 CPU 区间的测量长度的指数平均。设 t_n 为第 n 个 CPU 区间的长度，设 τ_{n+1} 为下一个 CPU 区间的预测值。因此，对于 α，$0 \leqslant \alpha \leqslant 1$，定义

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

这一公式定义了**指数平均**(exponential average)。t_n 值包括最近的信息；τ_n 存储了过去历史。参数 α 控制了最近和过去历史在预测中的相对加权。如果 $\alpha = 0$，那么 $\tau_{n+1} = \tau_n$，近来历史没有影响(当前情形被认为是暂时的)；如果 $\alpha = 1$，那么 $\tau_{n+1} = t_n$，只有最近 CPU 区间才重要(历史被认为是陈旧的、无关的)。更为常见的是，$\alpha = 1/2$，这样最近历史和过去历史同样重要。初始值 τ_0 可被定义为常量或作为系统的总体平均值。图 6.3 说明了一个指数平均值，其 $\alpha = 1/2$，$\tau_0 = 10$。



CPU区间(t_i)	6	4	6	4	13	13	13	...
"猜测"(τ_i)	10	8	6	6	5	9	11	12	...

图 6.3 下一个 CPU 区间长度的预测

为了便于理解指数平均的行为，可通过替换 τ_n 来扩展 τ_{n+1}，从而得到

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

由于 α 和 $(1-\alpha)$ 都小于或等于 1，所以后面项的权比其前面项的权要小。

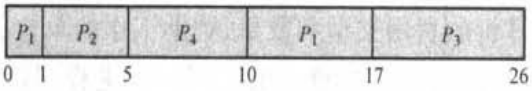
SJF 算法可能是*抢占的*或*非抢占的*。当一个新进程到达就绪队列而以前进程正在执行时，就需要选择。新进程，与当前运行进程所产生的 CPU 区间相比，可能有一个更短的下一个 CPU 区间。可抢占 SJF 算法可能会抢占当前运行进程，而非抢占 SJF 算法会允许当前

运行进程先完成其 CPU 区间。可抢占 SJF 调度有时称为**最短剩余时间优先**(shortest-remaining-time-first)**调度**。

作为例子,考虑如下四个进程,其 CPU 区间时间以 ms 计:

进程	到达时间	区间时间
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

如果进程按所给定的时间到达就绪队列,且需要所给定的区间时间,那么所产生的可抢占 SJF 调度如下面的 Gantt 图所示:



进程 $P1$ 在时刻 0 开始,因为这时队列中只有进程 $P1$。进程 $P2$ 在时刻 1 到达。进程 $P1$ 剩余时间(7 ms)大于进程 $P2$ 所需要的时间(4 ms),因此进程 $P1$ 被抢占,而进程 $P2$ 被调度。对于这个例子,平均等待时间为((10−1) + (1−1) + (17−2) +(5−3))/4 = 26/4 = 6.5 ms。如果使用非抢占 SJF 调度,那么平均等待时间为 7.75 ms。

6.3.3 优先权调度

SJF 算法可作为通用**优先权调度算法**(priority-scheduling algorithm)的一个特例。每个进程都有一个优先权与其关联,具有最高优先权的进程会被分配到 CPU。具有相同优先权的进程按 FCFS 顺序调度。

SJF 算法作为优先权算法,其优先权为(p)下一个(预测的)CPU 区间的倒数。CPU 区间越大,则优先权越小;反之亦然。

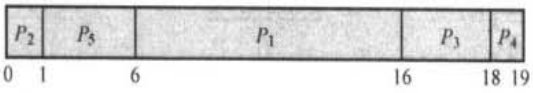
注意这里按照 *高优先权* 和 *低优先权* 来讨论调度。优先权通常为固定区间的数字,如 0 到 7,或 0 到 4 095。不过,对于 0 是最高还是最低的优先权,并没有定论。有的系统有低数字表示低优先权;其他的系统使用低数字表示高优先权。这一差异可导致混淆。在本书中用低数字表示高优先权。

作为例子,考虑下面一组进程,它们在时刻 0 时按顺序 $P1$, $P2$, …, $P5$ 到达,其 CPU 区间时间长度按 ms 计:

进程	到达时间	优先权
P_1	10	3
P_2	1	1
P_3	2	4

P_4	1	5
P_5	5	2

采用优先权调度，会按照下面的 Gantt 图来调度这些进程：



平均等待时间为 8.2ms。

优先权可通过内部或外部方式来定义。内部定义优先权使用一些可测量数据以计算进程优先权。例如，时间极限、内存要求、打开文件的数量和平均 I/O 时间区间与平均 CPU 区间之比都可以用于计算优先权。外部优先权是通过操作系统之外的准则来设置的，如进程重要性、用于支付使用计算机的费用类型和数量、赞助工作的单位、其他(通常为政治)因素。

优先权调度可以是可抢占的或者非抢占的。当一个进程到达就绪队列时，其优先权与当前运行进程的优先权相比较。如果新到达进程的优先权高于当前运行进程的优先权，那么抢占优先权调度算法会抢占 CPU。非抢占优先权调度算法只是将新进程加到就绪队列的头部。

优先权调度算法的一个主要问题是**无穷阻塞**(indefinite blocking)(或**饥饿**(starvation))。就绪可运行但缺乏 CPU 的进程可认为阻塞，等待 CPU。优先权调度算法会使某个低优先权进程无穷等待 CPU 在一个重负载计算机系统中，平稳的高优先权进程流可以阻止低优先权进程获得 CPU。通常，会发生两种情况。要么进程最终能运行(在系统最后为轻负荷时，如星期日凌晨 2 点)，要么计算机系统最终崩溃并失去所有未完成的低优先权进程。(据说，在 1973 年关闭 MIT 的 IBM 7094 时，发现有一个低优先权进程是于 1967 年提交但是一直未运行。)

低优先权进程无穷等待问题的解决方案之一是老化。**老化**(aging)是一种技术，以逐渐增加在系统中等待很长时间的进程的优先权。例如，如果优先权为从 127(低)到 0(高)，那么可以每 15 分钟递减等待进程的优先权。最终即使初始优先权值为 127 的进程会有系统中的最高优先权并能执行。事实上，不超过 32 小时，优先权为 127 的进程会老化为优先权为 0 的进程。

6.3.4 轮转法调度

轮转法(round-robin, RR)**调度算法**是专门为分时系统而设计的。它类似于 FCFS 调度，但是增加了抢占以在进程间切换。定义一个较小时间单元，称为**时间量**(time quantum)或时间片。时间片通常为 10 ms 到 100 ms。就绪队列作为循环队列处理。CPU 调度程序循环就绪队列，为每个进程分配不超过一个时间片间隔的 CPU。

为了实现 RR 调度，将就绪队列实现为进程的 FIFO 队列。新进程增加到就绪队列的尾

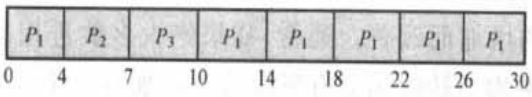
部。CPU 调度程序从就绪队列中选择第一个进程，设置定时器在一个时间片之后中断，最后分派该进程。

接着，有两种情况可能发生。进程可能只需要小于一个时间片的 CPU 区间。对于这种情况，进程本身会自动释放 CPU。调度程序接着会处理就绪队列的下一个进程。另一种情况是，当前运行进程的 CPU 区间比一个时间片要长，定时器会中断并产生操作系统中断。进行上下文切换，该进程会被加入到就绪队列的**尾部**。接着，CPU 调度程序会选择就绪队列中的下一个进程。

不过，采用 RR 策略的平均等待时间通常相当长。考虑如下一组进程，它们在时该 0 时到达，其 CPU 区间长度以 ms 计。

进程	区间时间
P_1	24
P_2	3
P_3	3

如果使用 4 ms 的时间片，那么进程 P_1 会执行最初 4 ms。由于它还需要另外的 20 ms，所以在第一时间片之后它会被抢占，而 CPU 就交给队列中的下一个进程 P_2。由于进程 P_2 不需要 4ms，所以在其时间片用完之前会退出。CPU 接着被交给下一个进程，进程 P_3。在每个进程都得到了一个时间片之后，CPU 又被交给了进程 P_1 以获得更多时间片。因此，RR 调度结果如下：



平均等待时间为 17/3 = 5.66 ms。

对于 RR 调度算法，队列中没有进程被分配超过一个时间片的 CPU 时间。如果进程的 CPU 区间超过了一个时间片，那么该进程会 *被抢占*，而被放回到就绪队列。RR 调度算法是可抢占的。

如果就绪队列中有 n 个进程且时间片为 q，那么每个进程会得到 1/n 的 CPU 时间，每个长度不超过 q 时间单元。每个进程必须等待 CPU 的时间不会超过(n−1)q 个时间单元，直到它的下一个时间片为止。例如，如果有五个进程，且时间片为 20 ms，那么每个进程每 100 ms会得到不超过 20 ms 的时间。

RR 算法的性能很大程度上依赖于时间片的大小。在极端情况下，如果时间片非常大(无限)，那么 RR 策略与 FCFS 策略一样。如果时间片很小(如 1 μs)，那么 RR 方法称为**处理器共享**，(从理论上来说)n 个进程对于用户来说都有它自己的处理器，速度各为真正处理器速度的 1/n。这种方法用在 Control Data Corporation (CDC)的硬件上可以用一组硬件和 10 组寄存器实现 10 个外设处理器。硬件为一组寄存器执行一个指令，然后为下一组执行。这种循环持续进行，形成了 10 个慢处理器而不是 1 个快处理器。(实际上，由于处理器比内

存快很多，而每个指令都要使用内存，所以这些处理器并不比 10 个真正处理器慢很多。）

不过，如果使用软件，那么还必须考虑上下文切换对 RR 调度的影响。假设只有一个具有 10 个时间单元的进程。如果时间片为 12 个时间单元，那么进程在不到一个时间片内就能完成，且没有额外开销。而如果时间片为 6 个时间单元，那么进程需要两个时间片，并产生了一个上下文切换。如果时间片为 1 个时间单元，那么就会有 9 个上下文切换，相应地使进程执行减慢（图 6.4）。

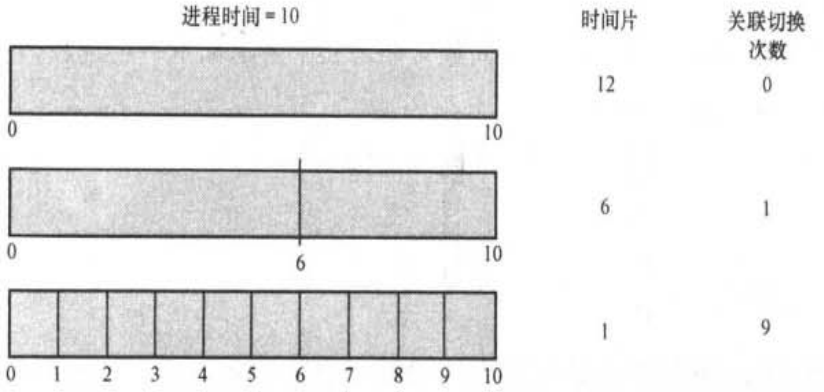
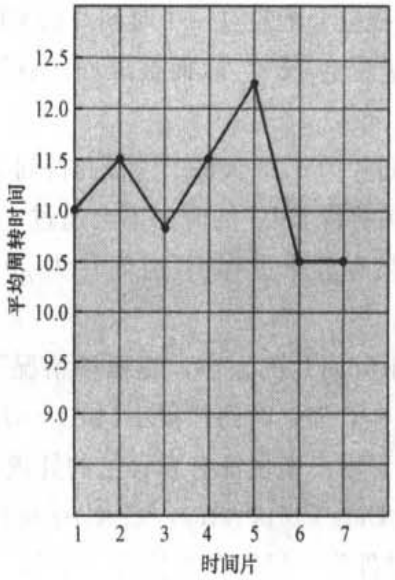


图 6.4 显示小的时间片增加了上下文切换开销

因此，人们希望时间片要比上下文切换时间长。如果上下文切换时间约为时间片的 10%，那么约 10%的 CPU 时间会浪费在上下文切换上。

周转时间也依赖于时间片的大小。正如从图 6.5 中所看到的，这组进程的平均周转时间并未随着时间片大小的增加而改善。通常，如果绝大多数进程能在一个时间片内结束其下一个 CPU 区间，那么平均周转时间会有所改善。例如有三个进程，每个都需要 10 个时间单元，如果时间片为 1 个时间单元，那么平均周转时间为 29。而如果时间片为 10，那么平均



进程	时间
P_1	6
P_2	3
P_3	1
P_4	7

图 6.5 显示周转时间随着时间片大小而改变

周转时间会降为 20。如果再加上上下文切换时间，那么平均周转时间对于较小时间片会增加，这是因为需要更多的上下文切换。

另一方面，如果时间片太大，那么 RR 调度就演变成了 FCFS 调度。根据经验，80%的 CPU 区间应该小于时间片。

6.3.5 多级队列调度

在进程可被容易地分成不同组的情况下，可以使用所建立的另一类调度算法。例如，一个常用划分方法是**前台**(或**交互式**)进程和**后台**(或**批处理**)进程。这两种不同类型的进程具有不同响应时间要求，也有不同调度需要。另外，与后台进程相比前台进程可能要有更高的(或外部定义)优先权。

多级队列调度算法(multilevel queue-scheduling algorithm)将就绪队列分成多个独立队列(图 6.6)。根据进程的某些属性，如内存大小、进程优先权或进程类型，进程会被永久地分配到一个队列。每个队列有自己的调度算法。例如，不同队列可用于前台和后台进程。前台队列可能是用 RR 算法调度，而后台队列可能是用 FCFS 算法调度。

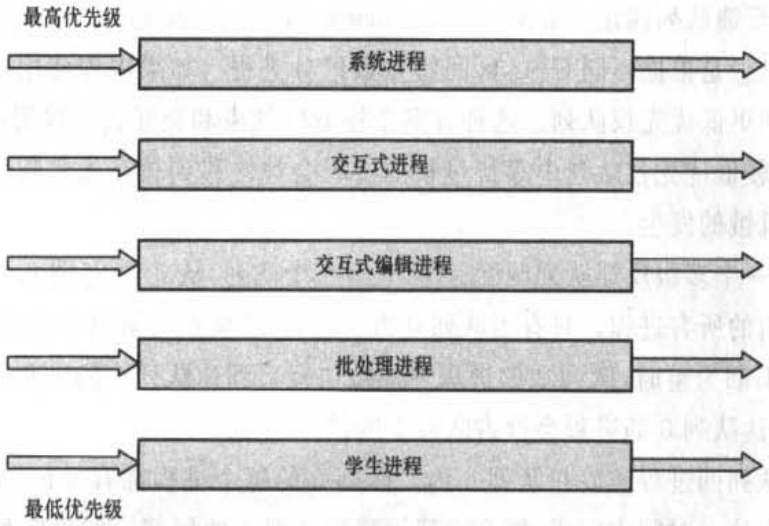


图 6.6 多级队列调度

另外，队列之间必须有调度，通常采用固定优先权可抢占调度来实现。例如，前台队列可以比后台队列具有绝对的优先权。

现在来研究一下具有五个队列的多级队列调度算法的例子：

1. 系统进程
2. 交互式进程
3. 交互式编辑进程
4. 批处理进程
5. 学生进程

每个队列与更低优先权队列相比有绝对的优先权。例如，只有系统进程、交互式进程和交互式编辑进程队列都为空，批处理队列内的进程才可运行。如果在一个批处理进程运行时有一个交互式编辑进程进入就绪队列，那么该批处理进程会被抢占。Solaris 2 使用这个算法的一种形式。

另一种可能是在队列之间划分时间片。每个队列都有一定的 CPU 时间，这可用于调度队列内的不同进程。例如，对于前台—后台队列的例子，前台队列可以有 80%的 CPU 时间用于在进程之间进行 RR 调度，而后台队列可以有 20%的 CPU 时间以采用 FCFS 算法调度其进程。

6.3.6 多级反馈队列调度

对于多级队列调度算法，通常进程进入系统时，被永久地分配到一个队列。进程并不在队列之间移动。例如，如果有独立队列以用于前台和后台进程，进程并不从一个队列移到另一个队列，这是因为进程并不改变前台或后台性质。这种设置的优点是低调度开销，缺点是不够灵活。

然而，**多级反馈队列调度**(multilevel feedback queue scheduling)允许进程在队列之间移动。其主要思想是根据不同 CPU 区间特点以区分进程。如果进程使用过多 CPU 时间，那么它会被移到更低优先权队列。这种方案会将 I/O 约束和交互式进程留在较高优先权队列。类似地，在较低优先权队列中等待过长的进程会被转移到较高优先权队列。这种形式的老化能阻止饥饿的发生。

例如，考虑一个多级反馈队列调度程序，它有三个队列，从 0 到 2(图 6.7)。调度程序首先执行队列 0 内的所有进程。只有当队列 0 为空时，它才能执行队列 1 内的进程。类似地，只有队列 0 和 1 都为空时，队列 2 的进程才能被执行。到达队列 1 的进程会抢占队列 2 的进程。同样，到达队列 0 的进程会抢占队列 1 的进程。

进入就绪队列的进程被放在队列 0 内。队列 0 的每个进程都有 8 ms 的时间片。如果一个进程不能在这一时间内完成，那么它就被移到队列 1 的尾部。如果队列 0 为空，队列 1 头部进程会得到一个 16 ms 的时间片。如果它不能完成，那么它将被抢占，并被放到队列 2 中。只有当队列 0 和 1 为空时，队列 2 内的进程才可根据 FCFS 来运行。

这种调度算法将给那些 CPU 区间不超过 8 ms 的进程以最高优先权。这种进程可以很快地得到 CPU，完成其 CPU 区间，并处理其下一个 I/O 区间。所需超过 8 ms 但不超过 24 ms 的进程也会很快被服务，即使它们的优先权要比较短进程低一点。长进程会自动沉入到队列 2，在队列 0 和 1 不用的 CPU 周期时可按 FCFS 顺序来服务。

通常，多级反馈队列调度程序可由下列参数来定义：

- 队列数量
- 每个队列的调度算法

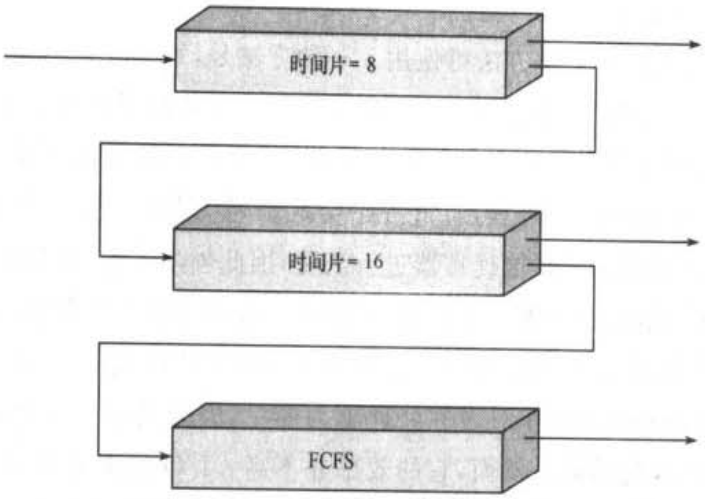


图 6.7 多级反馈队列

- 用以确定进程何时升级到较高优先权队列的方法
- 用以确定进程何时降级到较低优先权队列的方法
- 用以确定进程在需要服务时应进入哪个队列的方法

多级反馈队列调度程序的定义使它成为最通用的 CPU 调度算法。它可被配置以适应特定系统设计。不过,它还需要一些方法来选择参数值以定义最佳的调度程序。虽然多级反馈队列是最通用的方案,但是它也是最复杂的。

6.4 多处理器调度

迄今为止,主要集中讨论了单处理器系统内的 CPU 调度问题。如果有多个 CPU,那么调度问题就相应地更为复杂。已尝试过许多可能的方法,但和单处理器 CPU 调度一样,没有最好的解决方案。接下来,简要讨论多处理器调度的一些有关问题。(多处理器调度的完整介绍超出了本书范围;如需要更多信息,请参见本章推荐读物。)主要讨论处理器功能相同(或**同构**)的系统;任何可用处理器可用于运行队列内的任何进程。而且还假定**通用内存访问**(uniform memory access,UMA)。在第十五章到第十七章,会讨论不同处理器的系统(**异构**系统)。只有为给定处理器指令集编译的程序才能运行在该处理器上。

即使对同构多处理器,也有一些调度限制。考虑一个系统,有一个 I/O 设备与一个处理器通过私有总线相连。希望使用该设备的进程必须被调度以便在该处理器上运行,否则该设备就无法使用。

如果有多个相同处理器可用,那么可进行**负载分配**(load sharing)。有可能为每个处理器提供独立的队列。而在这种情况下,一个具有空队列的处理器会空闲,而另一个处理器会很忙。为了阻止这种情况,可使用一个共同就绪队列。所有进程都进入这一队列,并被调度

到任何可用空闲处理器上。

对于这种情况，有两种调度方法可使用。一种方法是，每个处理器是自我调度的。每个处理器都检查共同就绪队列，并选择一个进程来执行。正如将要在第七章中所看到的，如果有多个处理器试图访问和更新一个共同数据结构，那么每个处理器必须被仔细编程。必须确保两个处理器不能选择同一进程，且进程不会从队列中丢失。另一个方法可以避免这个问题，即选择一个处理器来为其他处理器进行调度，因此创建了主-从结构。

有的系统将这一结构做了进一步拓展，采用单一处理器即主服务器，以处理所有调度决策、I/O 处理和其他系统活动。其他处理器只执行用户代码。这种**非对称多处理**(asymmetric multiprocessing)比对多称处理更为简单，因为只有一个处理器访问系统数据结构，减轻了数据共享的需要。然而，它的效率并不高。I/O 约束进程可能使执行所有 I/O 操作的那个 CPU 成为瓶颈。通常，非对称多处理先在操作系统内实现，后来随着系统的发展而升级为对称多处理。

6.5 实时调度

在第一章，概述了实时操作系统并讨论它们不断增长的重要性。这里，描述用来在通用计算机系统内支持实时计算所需的调度工作，以此继续讨论问题。

实时计算可分成两种类型。**硬实时**(hard real-time)系统需要在保证的时间内完成关键任务。通常，在提交进程时，同时有一条语句告诉系统用来完成或执行 I/O 所需要的时间。接着，调度程序或者允许进程并保证该进程能按时完成，或者因不可能而拒绝请求。这称为**资源预约**(resource reservation)。这一保证要求调度程序确切地知道每个类型操作系统功能需要多长时间来执行，因此保证每个操作有最大时间量。对于具有外存或虚拟内存的系统，这一保证是不可能的；正如将在以后章节所要看到的，这些子系统会引起不可避免的和难以预见的变化时间量以执行特定进程。因此，硬实时系统由运行在专用于关键进程的硬件上的特殊目的软件组成，因而缺乏现代计算机和操作系统的全部功能。

软实时(soft real-time)计算的限制较少。它要求关键进程要比其他较弱进程拥有更高的优先权。虽然为一个分时系统增加软件实时功能会引起资源的不公平分配，且会导致有的进程延迟过长甚至饿死，但是它至少可以实现。其结果是通用目的系统能支持多媒体、高速交互式图形和其他需要支持软实时计算环境才可发挥功效的任务。

实现软实时功能要求仔细设计调度程序和操作系统有关方面。第一，系统必须有优先权调度，且实时进程必须有最高的优先权。实时进程的优先权不能随时间而下降，尽管非实时进程的优先权可以。第二，分派延迟必须小。延迟越小，实时进程在能运行时就可越快开始运行。

确保第一属性的成立相对简单。例如，可以不允许在实时进程在存在进程老化，因而保

证这些进程的优先权不变。然而，确保第二属性就相对复杂。问题是许多操作系统，包括绝大多数 UNIX 版本，在进行上下文切换前，被迫等待系统调用的完成或 I/O 阻塞的发生。这类系统的分派延迟可能很长，因为许多系统调用很复杂且有的 I/O 设备很慢。

为了让分派延迟保持很小，需要允许系统调用是可抢占的。实现这一目标有许多方法。一种方法是在长时间系统调用内插入**抢占点**(preemption point)，以检查高优先权进程是否需要运行。如要，则进行上下文切换。当高优先权进程终止时，所中断的进程继续完成系统调用。抢占点只能放在内核的"安全"位置，这里内核数据结构不是正在被修改。即使有抢占点，分派延迟也可能很大，因为实际只能增加少量的抢占点到内核中。

处理抢占的另一个方法是使整个内核可抢占。为了确保正确操作，所有内核数据结构必须通过各种(在第七章讨论)同步机制的使用来加以保护。采用这种方法，内核总是可被抢占，因为任何正在被修改的内核数据都加以保护，而不会受高优先权进程修改。这是最为有效的(也最为复杂的)应用最广的方法；Solaris 2 就使用它。

但是如果较高优先权进程需要读或修改当前正为另一个较低优先权进程所访问的内核数据，那么会如何呢？高优先权进程需要等待较低优先进权程的完成。这种情况称为**优先权倒置**(priority inversion)。事实上，一系列进程可能都在访问高优先权进程所需要的资源。这个问题可通过**优先权继承协议**(priority-inheritance protocol)来解决，即所有这些进程(即正在访问高优先权进程所需要的资源的进程)继承了高优先权，直到相关资源处理完毕。当它们完成后，它们的优先权进程返回到原来值。

在图 6.8 中，说明了分派延迟的组成。分派延迟**冲突阶段**有两个部分：

1. 抢占运行在内核中的任意进程。
2. 低优先级进程释放高优先级进程所需要的资源。

作为一个例子，如果 Solaris 2 的抢占被禁用，那么分派延迟超过 100 ms。不过，如果启用抢占，那么分派延迟通常会降低到 2 ms。

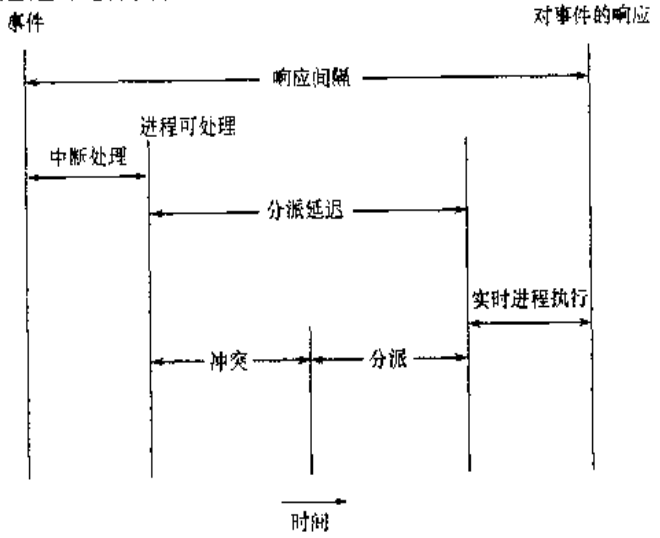


图 6.8 分派延迟

6.6 算法评估

如何选择 CPU 调度算法以用于特定系统？正如在 6.3 节所看到的，调度算法有许多，且各有自己的参数。因此，选择一个算法可能会很困难。

第一个问题是定义用于选择算法的准则。正如在 6.2 节所看到的，准则通常是通过 CPU 使用率、响应时间或吞吐量来定义的。为了选择一个算法，首先必须定义这些度量值的相对重要性。准则可包括如下度量值，如：

- 最大化 CPU 使用率，同时要求最大响应时间为 1 s。
- 最大化吞吐量，以要求周转时间平均地讲与总的执行时间线性成正比。

一旦定义了选择准则，需要评估所考虑的各种算法。在 6.6.1 小节到 6.6.4 小节中讨论不同的评估方法。

6.6.1 确定性建模

一种主要类型的评估方法称为分析评估法。**分析评估法**(analytic evaluation)使用给定算法和系统负荷，产生一个公式或数字，以评估针对该负荷算法的性能。

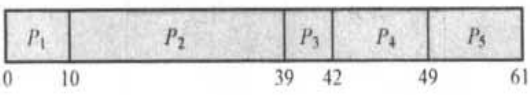
一种类型的分析评估是**确定性建模法**(deterministic modeling)。这种方法采用特定预先确定的负荷，定义在给定负荷下每个算法的性能。

例如，假设有所示的给定负荷。所有五个进程按给定顺序在时刻 0 到达，CPU 区间时间的长度都以 ms 计：

进程	区间时间
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

在这组进程中，主要研究 FCFS、SJF 和 RR(时间片＝10 ms) 调度算法，并判断哪个算法的平均等待时间最小。

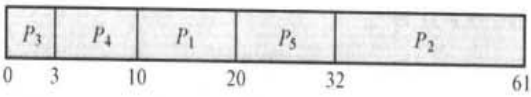
对于 FCFS 算法，按如下执行进程：



进程 P_1 的等待时间是 0 ms，进程 P_2 的等待时间是 10 ms，进程 P_3 的等待时间是 39 ms，进程 P_4 的等待时间是 42 ms 以及进程 P_5 的等待时间是 49 ms。因此，平均等待时间是

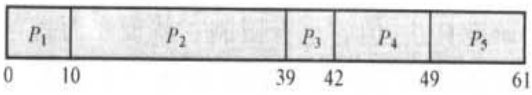
(0+10+39+42+49)/5=28 ms。

对于非抢占 SJF 调度,按如下执行进程:



进程 P_1 的等待时间是 10 ms,进程 P_2 的等待时间是 32 ms,进程 P_3 的等待时间是 0 ms,进程 P_4 的等待时间是 3 ms 以及进程 P_5 的等待时间是 20 ms。因此,平均等待时间是(10+32+0+3+20)/5=13 ms。

对于 RR 算法,按如下执行进程:



进程 P_1 的等待时间是 0 ms,进程 P_2 的等待时间是 32 ms,进程 P_3 的等待时间是 20 ms,进程 P_4 的是 23 ms 以及进程 P_5 的等待时间是 40 ms。因此,平均等待时间是(0+32+2+23+40)/5=23 ms。

可以看到,*在这种情况下*,SJF 调度策略所产生的平均等待时间不到 FCFS 调度中的一半;RR 算法给出了一个中间值。

确定性建模不但简单而且也快。它给出了确切数字,以允许算法被比较。然而,它要求输入为精确数字,而且其答案只适用于这些情况。确定性建模的主要用途在于描述调度算法和提供例子。在有的情况下,可能一次次地运行同样程序,并能精确测量程序的处理要求,那么就能使用确定性建模以选择调度算法。对于一组例子,确定性建模可表示趋势以分别供分析和证明。例如,对于刚才所描述的环境(所有进程及其时间都在时刻 0 时已知),可知 SJF 策略总能产生最小的等待时间。

然而,由于确定性建模通常过分特殊且要求过多精确知识,故用处有限。

6.6.2 排队模型

在许多系统上运行的进程每天都在变化,因此没有静态的进程集合和时间用于确定性建模。然而,CPU 和 I/O 区间的分布是可以确定的。这些分布可以被测量、近似或简单估计。其结果是一个数学公式以描述特定 CPU 区间的分布概率。通常,这种分布是指数的,可用其平均值来描述。类似地,进程到达系统的时间分布,即到达时间分布,也必须给定。

计算机系统可描述为服务器网络。每个服务器都有一个等待进程队列。CPU 是具有就绪队列的服务器,而 I/O 系统具有设备队列。知道了到达率和服务率,就可以计算使用率、平均队列长度、平均等待时间等。这种研究领域称为**排队网络分析**(queueing-network analysis)。

作为一个例子,设 n 为平均队列长度(不包括正在服务的进程),设 W 为队列的平均等

待时间，设 λ 为新进程到达队列的平均到达率(如每秒三个进程)。那么，希望在进程等待的 W 时间内，有 $\lambda \times W$ 新进程到达队列。如果系统处于稳定状态，那么离开队列的进程的数量必定等于到达进程的数量。因此，有

$$n=\lambda \times W$$

这一公式称为 **Little 公式**。Little 公式特别有用因为它适用于任何调度算法和到达分布。

如果已知 Little 公式中的两个变量的值，可以计算第三个变量。例如，若已知(平均)每秒钟有 7 个进程到达，且通常在队列中有 14 个进程，那么可以计算每个进程的平均等待时间为 2 s。

排队分析可用于比较调度算法，但它也有限制。就现在而言，可以处理的算法和分布种类还是比较有限的。复杂算法或分布的数学分析可能难于处理。因此，到达和处理分布通常被定义成不现实的，但数学上易处理的形式。而且往往也需要一些可能并不精确的独立假设。为了能计算一个答案，排队模型通常只是现实系统的近似。因此，计算结果的精确性值得怀疑。

6.6.3 模拟

为了获得对调度算法更为精确的评估，可使用**模拟**(simulation)。模拟涉及对计算机系统模型进行程序设计。通过软件数据结构表示系统主要组成部分。模拟程序有一个变量以代表时钟；当该变量的值增加时，模拟程序会修改系统状态以反映设备、进程和调度程序的活动。随着模拟程序的执行，用以表示算法性能的统计数字可以被收集并打印出来。

驱动模拟的数据可由许多方法产生。最为普通的方法使用随机数生成器，它被编程以根据概率分布生成进程、CPU 区间时间、到达时间、离开时间等。分布可以数学地(统一的、指数的、泊松)或经验地加以定义。如果根据经验定义分布，那么要对所研究的真实系统进行测量。其结果可用来定义实际系统事件的真实分布，该分布可再用来驱动模拟。

然而，由于实际系统前后事件之间有关系，分布驱动模拟可能不够精确。频率分布只表示每个事件发生了多少次；它并不能表示事件的发生顺序。可以采用**跟踪磁带**来纠正这个问题。通过监视实际系统，记录真实事件发生的顺序来建立跟踪磁带(图 6.9)。然后使用这个序列来驱动模拟。跟踪磁带提供了恰基于同一真实输入集合来比较两种算法的很好的方法。这种方法针对给定输入产生了精确的结果。

然而模拟通常需要数小时的计算机时间，所以是昂贵的。为了提供更精确的结果，需要更细致的模拟，但是也需要更多的计算机时间。而且跟踪磁带可能需要大量的存储空间。最后，模拟程序的设计、编码和调试可能是一项大型的工作。

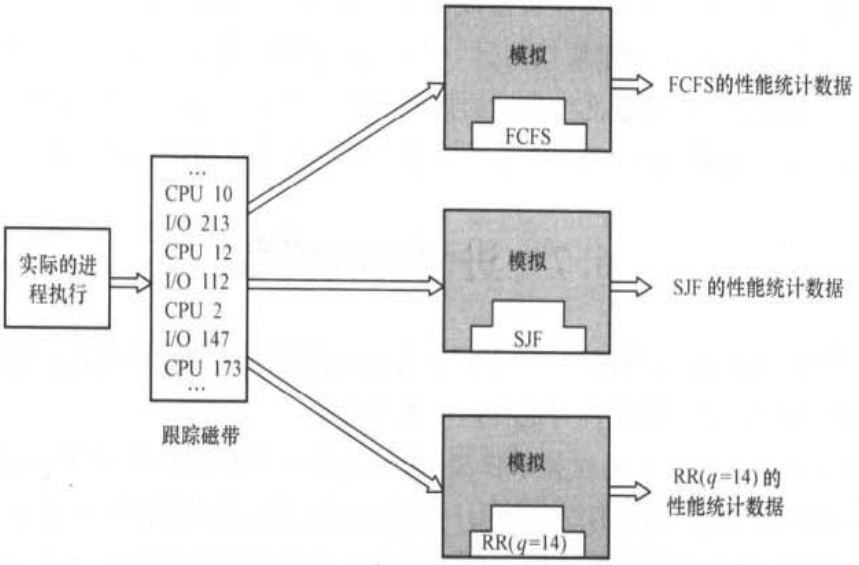


图 6.9 通过模拟来评估 CPU 调度算法

6.6.4 实现

即使是模拟其精确度也是有限的。针对评估调度算法，惟一完全精确的方法是对它进行程序编码，将其放在操作系统内，并观测它如何工作。这一方法将真实算法放进实际系统然后在真实操作条件下对它内进行评估。

主要困难是这种方法的代价。所引起的代价不但包括对算法进行程序编码、修改操作系统以支持该算法和所需的数据结构，而且包括用户对不断变化的操作系统的反应。绝大多数用户并不关心创建更好的操作系统；而只需要能执行进程并使用其结果。一个经常变化的操作系统并不能帮助用户完成他们的工作。这种方法通常用于新计算机系统的安装。例如，一个新的网络工具在真正使用之前，可以先用模拟用户负荷来检验它，以确定此工具中的瓶颈并估算系统能支持多少用户。

任何算法评估的另一个困难是算法所使用的环境会变化。环境变化不但通过常见方式产生，即包括新用户程序的编写和问题类型的变化，而且也包括调度程序性能所引起的结果。如果小进程被赋予优先权，那么用户会将大进程分成小进程的集合。如果交互式进程比非交互式进程更能获得优先权，那么用户就可能切换到交互式进程的使用。

例如，DEC TOPS－20 系统通过观察终端 I/O 的量来自动将进程分成交互式的和非交互式的。如果一个进程在一分钟间隔内没有对终端进行输入或输出，那么该进程就被划分为非交互式的，并移到较低优先权队列。这种策略导致了如下情况。一位程序员修改了其程序，在一分钟不到的常规时间间隔内输出一个任意字符到终端上。虽然该终端输出完全没有意义，但是系统却赋予该程序更高优先权。

最为灵活的调度算法可以为系统管理员或用户所改变。在操作系统构造时、引导时或

运行时，调度程序所使用的变量可以被修改以反映系统所预期的未来应用。灵活调度的需要是反映了机制与策略区分的重要性的另一个例子。例如，如果付薪水的支票需要马上被处理和打印，但是这通常作为较低优先级的批处理作业来完成，那么可暂时地给批处理队列一个较高的优先权。遗憾的是，很少有操作系统允许这种类型的可调整的调度。

6.7 进程调度模型

在本节，讨论 Solaris 2、Windows 2000 和 Linux 操作系统的进程调度。然而，在观察这些不同调度模型之前，首先将线程与进程调度联系起来。

在第五章，介绍了线程与进程关系模型，因此允许一个进程可以有多个控制线程。另外，还区分了*用户级*和*内核级*的线程。用户级线程是由线程库管理的，内核并不知道它们。要在 CPU 上运行，用户级线程最终要映射到一个相关的内核级线程上，尽管这一映射关系可能是间接的且可使用轻量级进程(LWP)。用户级线程和内核级线程的区别之一在于它们是如何调度的。线程库调度用户级线程以在可用的 LWP 上运行，这种方案称为**进程本地调度**(process local scheduling)，即线程调度是在应用程序内部进行的。相反，内核使用**系统全局调度**以决定调度哪个内核线程。本书不详细讨论不同的线程库如何本地调度线程；因为线程调度是软件库的事而不是操作系统要关心的事。这里只讨论全局调度，因为它是由操作系统执行的。

6.7.1 例子：Solaris 2

Solaris 2 采用基于优先级的进程调度。它有四类调度，按优先级分别为实时、系统、分时和交互式。每个类型有不同的优先权和调度算法，尽管分时和交互式是使用同样的调度策略。Solaris 2 调度如图 6.10 所示。

进程开始只有一个 LWP，根据需要能创建多个新的 LWP。每个 LWP 继承了父进程的调度类型和优先权。进程缺省调度类型为分时。分时的调度策略采用多级反馈队列，动态地调整优先级和赋予不同长度的时间片。缺省地，在优先级和时间片之间有反比关系：优先级越高，时间片越小；优先级越低，时间片越大。交互式进程通常有较高的优先级，CPU 约束进程有较低的优先级。这种调度策略对交互式进程有好的响应时间，对 CPU 约束进程有好的吞吐量。Solaris 2.4 在进程调度中引入了交互式类型。交互式类型与分时类型使用同样的调度策略，但是它能给窗口应用程序更高的优先级以提高性能。

Solaris 2 使用系统类来运行内核进程，如调度程序和换页后台服务。一旦创建，系统进程的优先级就不改变。系统类专为内核所使用(在内核模式下运行的用户进程并不属于系统类)。系统类的调度策略是不分时的。属于系统类的线程会一直运行，直到它阻塞或为更高优先级的线程所抢占。

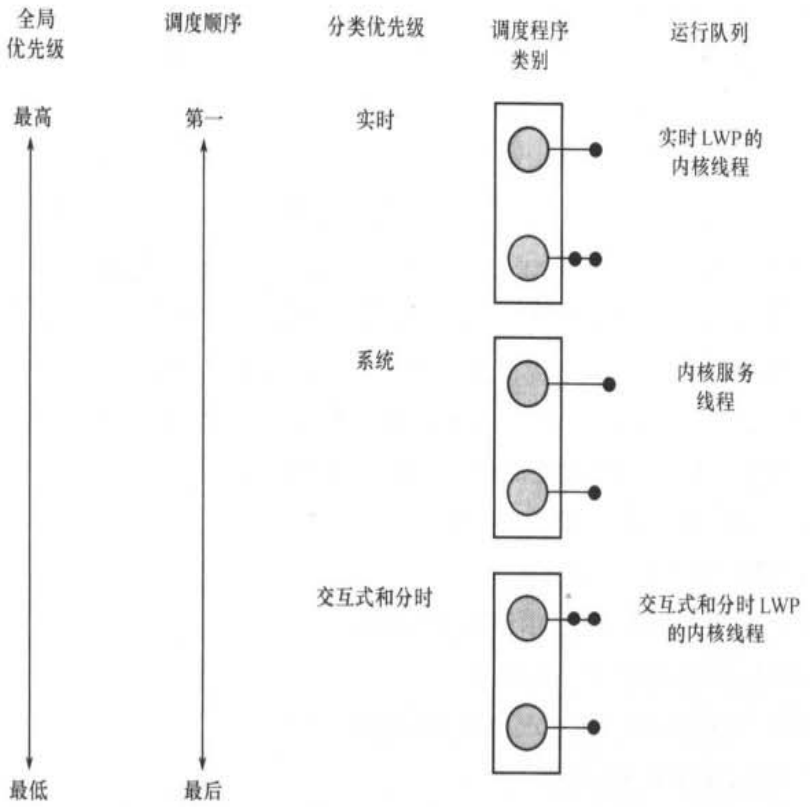


图 6.10 Solaris 2 调度

实时类型的线程在所有类型线程中运行时具有最高优先级。这种安排允许实时进程保证在给定时间段内对系统响应。实时进程能在其他类型的进程之前运行。通常，只有少数进程属于实时类型。

每个调度类型包含一定的优先级别集合。然而，调度程序会将特定类的优先级转换为全局优先级，并从中选择最高全局优先级线程以执行。所选择的线程会在 CPU 上执行，直到下面条件之一发生：

1. 它被阻塞。
2. 它使用了它的时间片(如果不是系统线程)。
3. 它被更高优先级的线程抢占。

如果多个线程具有相同的优先级，那么调度程序采用轮转队列。

6.7.2 例子：Windows 2000

Windows 2000 采用基于优先权的、可抢占调度算法来调度线程。Windows 2000 调度程序确保最高优先级的线程总是运行。Windows 2000 内核中用于处理调度的部分称为分配程序。由分配程序选择运行的线程会一直运行，直到被更高优先级的线程所抢占，或直到它终止，或其时间片已到，或调用了阻塞系统调用如 I/O。如果在较低优先级线程运行时更

高优先级的实时线程变为就绪，那么较低优先级线程就会被抢占。这种抢占使得实时线程在需要访问 CPU 时能获得优先权。然而，Windows 2000 不是硬实时操作系统，因为它不保证实时线程能在某一特定时间限制内开始运行。

使用 32 级优先权方案来确定线程执行的顺序。优先级分为两大类型：**可变类型**(variable class)包括优先级从 1 到 15 的线程，**实时类型**(real-time class)包括优先级从 16 到 31 的线程。(还有一个线程运行在优先级 0，它用于内存管理。)分配程序为每个调度优先级使用一个队列，从高到低审视队列集，直到它发现一个线程就绪可执行。如果没有找到就绪线程，那么分配程序会执行一个称为**空闲线程**(idle thread)的特别线程。

在 Windows 2000 内核和 Win32 API 的数字优先级之间，有一种关系。Win32 API 定义了一个进程可能属于的一些优先级类型。它们包括：

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- IDLE_PRIORITY_CLASS

除了 REALTIME_PRIORITY_CLASS 之外，所有优先级类型都是可变类型优先级，这意味着属于这些类型的线程优先级可以改变。

在每个优先级类型内是相对优先级。相对优先级的值包括：

- TIME_CRITICAL
- HIGHEST
- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- LOWEST
- IDLE

每个线程的优先级是基于它所属的优先级类型和它所属类型的相对优先级。图 6.11 说明了这种关系。每个优先级类型的值出现在顶行。左列包括不同的相对优先级的值。例如，如果一个线程属于 ABOVE_NORMAL_PRIORITY_CLASS 类型，且其相对优先级为 NORMAL，那么该线程的数字优先级为 10。

另外，每个线程在其所属的类型中有一个基础优先级值来表示所属优先级范围。缺省地，基础优先级是那个特定类型中的 NORMAL 相对优先级的值。每个优先级类型的基础优先级是：

- REALTIME_PRIORITY_CLASS——24。

- HIGH_PRIORITY_CLASS — 13。
- ABOVE_NORMAL_PRIORITY_CLASS — 10。
- NORMAL_PRIORITY_CLASS — 8。
- BELOW_NORMAL_PRIORITY_CLASS — 6。
- IDLE_PRIORITY_CLASS — 4。

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

图 6.11 Windows 2000 优先级

进程通常是属于 NORMAL_PRIORITY_CLASS 中的一员，除非父进程是 IDLE_PRIORITY_CLASS 或在创建进程时指定了另一种类型。线程的初始优先级通常为线程所属的进程的基础优先级。

在线程时间片用完时，线程被中断；如果线程属于可变优先级类型，那么它的优先级被降低。然而，优先级决不会降低到基础优先级之下。降低线程优先级似乎限制了计算约束线程的 CPU 消耗。当可变优先级线程从等待操作被释放时，分配程序提升其优先级。提升多少与线程等待什么有关；例如，等待键盘 I/O 的线程会得到较大优先级提升；而等待磁盘操作的线程将得到一般性提升。这种策略能给使用鼠标和窗口的交互式线程更好的响应时间，并能让 I/O 约束线程保持 I/O 设备繁忙，同时也允许计算约束线程在后台使用空闲的 CPU 周期。这种策略为多个分时操作系统包括 UNIX 所采用。另外，与用户交互的当前窗口也会得到优先级提升以加快其响应时间。

当用户运行交互式程序时，系统需要为该进程提供特别好的响应。为此，Windows 2000 对 NORMAL_PRIORITY_CLASS 进程有一个特别调度规则。Windows 2000 区分*前台进程*(在当前屏幕上选择的)和*后台进程*(没有在当前被选择的)。当一个进程进入前台时，Windows 2000 增加其调度时间片的倍数因子，通常为 3。这一增加给了前台进程在其被分时抢占前多三倍的运行时间。

6.7.3 例子：Linux

Linux 提供了两种独立进程调度算法。一种是分时算法，用于在多个进程之间进行公平可抢占调度；另一个是为实时任务而设计的，对这些任务而言绝对优先要比公平更为重

要。在6.5节中，讨论了实时系统必须允许内核被抢占以降低分派延迟的情况。Linux只允许在用户模式下运行的进程被抢占。进程在内核模式下运行时不能被抢占，即使有更高优先级的实时进程可以运行。

各个进程的特性有些是指调度类型，它决定了什么算法应用于这一进程。Linux使用的调度类型是由用于实时计算扩展的POSIX标准(POSIX.4，现称为POSIX.1b)来定义的。

第一种调度类型是用于分时进程的。对于传统的分时进程的，Linux使用了优先权的、**基于信用**度的算法。每个进程都有一定数量的调度信用度；当要选择一个新任务运行时，具有最多信用的进程会被选择。每次出现定时器中断时，当前运行进程会失去一个信用；当其信用为0时，它会被暂停且另一个进程会被选择。

如果没有可运行进程有任何信用，那么Linux重新计算信用，将信用加到系统中的每个进程上(而不仅仅是可运行的进程)。根据如下公式

$$\text{信用} = \frac{\text{信用}}{2} + \text{优先级}$$

这种算法似乎混合了两个因素：进程历史和它的优先级。进程会保留剩余信用的一半(credits)，因为先前的信用重计算操作操作将保留至算法被应用之后，以反映进程近来行为的历史。一直运行的进程会很快用完其信用，而经常需要悬挂的进程会通过多次信用重计算和最终在一次重计算信用后获得较高信用而积聚信用。这样信用系统会自动地赋予交互式进程或I/O约束进程(对这些进程来说，快速响应时间是重要的)更高的优先级(priority)。

在计算新信用时使用进程优先级允许了进程优先级可以进一步细调。后台批处理作业能被给予低优先级；它们比交互式用户的作业自动地收到更少的信用，因此得到了较小百分比的CPU时间，比具有较高优先级的类似作业要少。Linux使用这种优先级系统来实现标准UNIX良好进程优先级机制。Linux实时调度更为简单。Linux实现了两种POSIX.1b所要求的实时调度类型：先到先处理(FCFS)和轮转(RR)(分别在6.3.1小节和6.3.4小节讲述)。对于这两种情况，每个进程除了有调度类型外还有一个优先级。虽然对于分时调度，不同优先级的进程在一定程序上仍然可互相竞争；但是对于实时调度，调度程序总是选择最高优先级的进程来运行。对于同等优先级的进程，Linux选择等待时间最长的进程运行。FCFS和RR调度的主要差别是FCFS进程继续运行直到其退出或阻塞，而RR进程在运行一会儿之后会被抢占而被移到调度队列的尾部，因此同样优先级的轮转进程会自动地会在它们之间分时。

注意Linux实时调度是软实时而不是硬实时的。调度程序能严格保证实时进程的相对优先权，但是内核不能保证一旦实时进程可以运行那么在多久内能让它运行。记住Linux内核代码决不能被用户模式代码所抢占。如果当内核为一个进程而准备好执行系统调用时一个中断到达并唤醒了另一个实时进程，那么实时进程将不得不等待直到当前运行的系统调用完成或阻塞时为止。

6.8 小 结

CPU 调度的任务是从就绪队列中选择一个等待进程，并为其分配 CPU。CPU 由分配程序分配给被选中的进程。

先来先处理(FCFS)调度是最简单的调度算法，但是它会让短进程等待非常长的进程。最短作业优先(SJF)调度被证明为最佳，提供了最短平均等待时间。实现 SJF 调度比较困难，因为预测下一个 CPU 区间的长度很困难。SJF 算法是通用优先权调度算法(将 CPU 简单地分配给具有最高优先权的进程)的特例。优先权和 SJF 调度会产生饥饿。老化技术可阻止饥饿。

时间片轮转(RR)调度对于分时(交互式)系统更为适用。RR 调度让就绪队列的第一个进程使用 CPU 达 q 个时间单元，这里 q 是时间片。在 q 时间单元之后，如果该进程还没有释放 CPU，那么它被抢占且进程被放到就绪队列的尾部。该算法的主要问题是时间片的选择。如果时间片太大，那么 RR 调度就演变成了 FCFS 调度；如果时间片太小，那么因上下文切换而引起的调度开销就变得过大。

FCFS 算法是非抢占的；RR 算法是可抢占的。SJF 和优先级算法可以是可抢占的也可以是非抢占的。

多级队列算法允许多个不同算法用于各种类型的进程。最为常用的是前台交互式队列(使用 RR 调度)和后台批处理队列(使用 FCFS 调度)。多级反馈队列允许进程在队列之间迁移。

因为有多种不同的调度算法可用，所以需要某种方法来从中选择它们。分析方法使用数学分析以确定算法性能。模拟方法通过对“代表性”的进程样例采用调度算法模拟并计算其结果性能来确定性能优劣。

如果操作系统在内核级支持线程，那么必须调度线程而不是进程来执行。Solaris 2 和 Windows 2000 就是这样的系统，这两个系统都采用可抢占的、基于优先级的调度算法，包括支持实时线程，来调度线程。Linux 进程调度程序也使用基于优先级算法，并提供实时支持。这三种操作系统的调度算法通常都偏爱交互式进程而不是批处理进程和 CPU 约束进程。

习 题 六

6.1 一个 CPU 调度算法决定了它所调度的进程的执行顺序。如果在一个处理器上有 n 个进程要被调度，可能有多少种不同的调度算法？给出一个用 n 表示的公式。

6.2 详细说明可抢占式和非抢占式调度之间的差别。说明为什么在计算机中心最好不要使用严格的非抢占式调度。

6.3 考虑下列进程集，进程占用的CPU区间时间长度以毫秒来计算：

进程	区间时间	优先级
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

假设在时刻0进程以 P_1，P_2，P_3，P_4，P_5 的顺序到达。

a. 画出4个Gantt图分别演示用FCFS、SJF、非抢占优先级(数字小代表优先级高)和RR(时间片=1)算法调度时进程的执行过程。

b. 在a里每个进程在每种调度算法下的周转时间是多少？

c. 在a里每个进程在每种调度算法下的等待时间是多少？

d. 在a里结果哪一种调度算法的平均等待时间对所有进程而言最小？

6.4 假设下列进程在所指定的时刻到达等待执行。每个进程将运行所列出的时间量长度。在回答问题时，假设使用非抢占式调度算法，基于选择时你所拥有的信息做出决定。

进程	到达时间	区间时间
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

a. 当使用FCFS调度算法时，这些进程的平均周转时间是多少？

b. 当使用SJF调度算法时，这些进程的平均周转时间是多少？

c. SJF调度算法被认为能提高性能，但是注意在时刻0选择运行进程 P_1 因为无法知道两个更短的进程很快会到来。计算一下如果在第一个时间单元CPU被置为空闲，然后使用SJF调度算法，计算这时的平均周转时间是多少？注意在空闲时，进程 P_1 和 P_2 在等待，所以他们的等待时间可能会增加。这个算法可以被认为是预知(future-knowledge)调度。

6.5 考虑RR调度算法的一个变种，在这个算法里，就绪队列里的项是指向PCB的指针。

a. 如果把两个指针指向就绪队列中的同一个进程，会有什么效果？

b. 这个方案的主要优点和缺点是什么？

c. 如何修改基本的RR调度算法，从而不用两个指针达到同样的效果？

6.6 在多级队列系统的不同层使用不同大小的时间片有什么优点？

6.7 考虑下面的基于动态改变优先级的可抢占式优先权调度算法。大的优先权数代表高优先权。当一个进程在等待CPU时(在就绪队列中，但未执行)，优先权以 α 速率改变；当它运行时，优先权以 β 速率改变。所有的进程在进入就绪队列时被给定优先权为0。参数 α 和 β 可以设定给许多不同的调度算法。

a. $\beta>\alpha>0$ 时所得的是什么算法？

b. $\alpha<\beta<0$ 时所得的是什么算法？

6.8 许多CPU调度算法可以设置参数。例如，RR算法需要一个参数来指定时间片。多级反馈队列需要一些参数来定义队列的数，每一个队列的调度算法，在队列之间移动进程的标准，等等。

这些算法就成了一个算法集合(例如所有时间片的RR算法集合等)。一个算法集合可以包括另一个

(例如 FCFS 算法是一个时间片无限的 RR 算法)。下列各对算法集之间是否有联系,如果有是什么?

a. 优先级和 SJF

b. 多层反馈队列和 FCFS

c. 优先级和 FCFS

d. RR 和 SJF

6.9 假设一种偏好最近使用最少处理器时间的进程的调度算法(在短期 CPU 调度层次),为什么这种算法偏好 I/O 约束程序而又不会让 CPU 约束程序永久饥饿?

6.10 解释下面调度算法对短进程偏好程度上的区别?

a. FCFS

b. RR

c. 多级反馈队列

推荐读物

Lampson[1968]提供了关于调度的一般论述。Kleinrock[1975]、Sauer 和 Chandy 以及 Lazowska 等[1984]的论著中有更多的有关调度理论的正式论述。Ruschizka 和 Fabry[1977]给出了调度的综合的分析。Haldar 和 Subramanian[1991]论述了分时系统中的处理器调度的公平性。

Corbato 等[1962]的论著中描述了最初在 CTSS 系统中实现的反馈队列。Schrage[1967]分析了这种反馈队列系统。Coffman 和 Kleinrock[1968]研究了多级反馈队列的变种。Coffman、Denning[1973]和 Svobodova[1976]给出了其他研究。Vuillemin[1978]给出了一个操纵优先级队列的数据结构。

Anderson 等[1989]论述了线程调度。Jones 和 Schwarz[1980],Tucker 和 Gupta[1989],Zahorjan 和 McCann[1990],Feitelson 和 Rudolph[1990]以及 Leutenegger 和 Vernon[1990]给出了关于多处理器调度的论述。

Liu 和 Layland[1973],Abbot[1984],Jensen 等[1985],Hong 等和 Khanna 等[1992]给出了有关实时系统中的调度的论述。Zhao[1989]编辑了实时的操作系统的一个特殊问题。Eykholt 等[1992]描述了 Solaris 2 中的实时部分。

Henry[1984]、Woodside[1986]、Kay 和 Lauder[1988]论述了公平共享调度程序。

Mauro 和 McDougall[2001],Solomon 和 Russinovich[2000]以及 Bovet 和 Cesati[2001]的论著中分别有关于 Solaris 2、Windows 2000 和 Linux 的调度的详细论述。

第七章 进程同步

协作进程是可以与在系统内执行的其他进程互相影响的进程。互相协作的进程可以直接共享逻辑地址空间(包括代码和数据),或者只通过文件共享数据。前一种情况可通过轻量级进程或线程的使用来实现,参见第五部分。共享数据的并发访问会导致产生数据不一致。在本章,讨论各种确保共享同一逻辑地址空间的协作进程能有序执行并维护数据一致性的机制。

7.1 背 景

在第四章,开发了一种系统模型,它包括若干**协作顺序进程**(cooperating sequential processes),这些进程异步执行且可能共享数据,采用有限缓冲方案(在操作系统中较为典型)说明了这种模型。

再次回到有限缓冲问题的共享内存解决方案,参见 4.4 节所述。正如曾经指出的,解决方案允许在缓冲内同时最多只有 BUFFER_SIZE－1 项。假定要修改这一算法以弥补这个缺陷。一种可能的方法是增加一个整数变量 counter,初始化为 0。每次向缓冲增加一个新项时,counter 就递增;每次从缓冲中移去一项时,counter 就递减。生产者进程代码可修改如下:

```
while (1) {
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
      ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

消费者进程代码可修改如下:

```
while (1) {
    while (counter == 0)
       ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
```

```
    counter--;
    /* consume the item in nextConsumed */
}
```

虽然生产者和消费者程序各自都正确，但是当并发执行时它们可能并不能正确执行。为了便于说明，假设变量 counter 的当前值为 5，且生产者进程和消费者进程并发执行语句“counter++”和“counter--”。根据这两条语句的执行，变量 counter 的值可以是 4、5 或 6！惟一正确的结果应是 counter == 5，如果生产者和消费者独立执行，那么会生成此正确结果。

可说明按如下方式 counter 的值就可能不正确。注意语句“counter++”可按如下方式以机器语言(在一个典型机器上)实现：

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter = $register_1$

其中，$register_1$ 为本地 CPU 寄存器值。类似地，语句“counter--”可按如下方式来实现：

$register_2$ = counter
$register_2$ = $register_2$ - 1
counter = $register_2$

其中，$register_2$ 为本地 CPU 寄存器值。虽然 $register_1$ 和 $register_2$ 可以是同一物理寄存器(如累加器)，但是请记住中断处理程序会保存和恢复该寄存器的值(2.1 节)。

并发执行“counter++”和“counter--”相当于按任意顺序来交替执行上面所表示的较低级别语句(每条高级语句内的顺序被保持)。一种交叉的形式如下：

T_0：*producer* execute $register_1$ = counter	{$register_1$ = 5}
T_1：*producer* execute $register_1$ = $register_1$ + 1	{$register_1$ = 6}
T_2：*consumer* execute $register_2$ = counter	{$register_2$ = 5}
T_3：*consumer* execute $register_2$ = $register_2$ - 1	{$register_2$ = 4}
T_4：*producer* execute counter = $register_1$	{*counter* = 6}
T_5：*consumer* execute counter = $regidter1_2$	{*counter* = 4}

注意这里得到了表示有 4 个满缓冲的不正确的状态“counter == 4”，而事实上有 5 个满缓冲。如果交换 T_4 和 T_5 两条语句的顺序，那么会得到不正确的状态“counter == 6”。

之所以得到了不正确状态，是因为允许两个进程并发操作变量 counter。像这样的情况，即多个进程并发访问和操作同一数据且执行结果与访问发生的特定顺序有关，称为**竞争条件**(race condition)。为了防止上述竞争条件，需要确保一段时间内只有一个进程能操作变量 counter。为了实现这种保证，要求一定形式的进程间同步。这种情况经常出现在操作系统中，因为系统的不同部分操纵资源而人们需要这些变化之间不会互相影响。本章的主

要部分是关于**进程同步**(process synchronization)和**协调**(coordination)的问题。

7.2 临界区域问题

考虑一个系统有 n 个进程$\{P_0, P_1, \cdots, P_{n-1}\}$,每个进程有一个代码段称为**临界区**(critical section),在该区中进程可能改变共同变量、更新一个表、写一个文件等。这种系统的重要特征是当一个进程在临界区内执行,没有其他进程被允许在临界区内执行。因此,进程临界区的执行在时间上 *互斥*。*临界区问题*是设计一个进程能用来协作的协议。每个进程必须请求允许进入其临界区。实现这一请求的代码段称为**进入区**(entry section)。临界区之后可有**退出区**(exit section)。其他代码为**剩余区**(remainder section)。

临界区问题的解答必须满足如下三项要求:

1. **互斥**:如果进程 P_i 在其临界区内执行,那么其他进程都不能在其临界区内执行。

2. **有空让进**:如果没有进程在其临界区内执行且有进程希望进入临界区,那么只有那些不在剩余区内执行的进程能参加决策,以选择谁能下一个进入临界区,且这种选择不能无限推迟。

3. **有限等待**:在一个进程做出进入其临界区的请求到该请求被允许期间,其他进程被允许进入其临界区的次数存在一个上限。

假定每个进程的执行速度不为 0。然而,不能对 n 个进程的**相对速度**作任何假设。

在 7.2.1 小节和 7.2.2 小节中,会给出满足这三项要求的临界区问题的解答。这些解答不需要对任何硬件指令或硬件支持的处理器的数量做假设。然而,假定基本机器语言指令(如 load、store 和 test)是原子执行的。即如果两个这样的指令并发执行,那么结果等同于它们按任何顺序来顺序执行。在这种情况下,如果 load 和 store 并发执行,那么 load 会得到旧值或新值但却不能为两者的组合。

在描述算法时,只定义用于同步目的的变量,并只描述一种典型进程 P_i,它具有图 7.1 所示的通用结构。进入区和退出区被框起来以突出这些代码段的重要性。

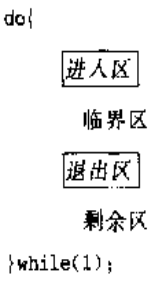


图 7.1 典型进程 P_i 的通用结构

7.2.1 两进程解法

在本节,只讨论同一时间用于两个进程的算法。两个进程标为 P_0 和 P_1。为了方便,当使用 P_i 时,用 P_j 来表示另一个进程;即 $j == 1 - i$。

1. 算法 1

第一种方法是让两个进程共享一个普通整数变量 turn,其初值为 0(或 1)。如果 turn

== i，那么进程 P_i 允许在其临界区内执行。进程 P_i 的结构如图 7.2 所示。

这一解答确保了每个时刻只有一个进程处于临界区域。然而，它并不满足前进要求，这是因为它要求进程在临界区中执行时要严格交替。例如，如果 turn == 0 且 P_1 就绪要进入其临界区，那么尽管 P_0 可能在其剩余区段，P_1 并不能这么做。

2. 算法 2

算法 1 的问题是它没有保留每个进程状态的足够信息；它只记住了哪个进程能进入其临界区。为了解决这一问题，可以用下面的数组来替换变量 turn：

```
boolean flag[2];
```

数组元素初始化为 false。如果 flag[i]为 true，那么该值表示 P_i 准备进入临界区。进程 P_i 的结构如图 7.3 所示。

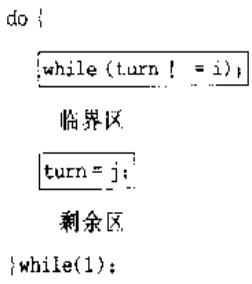


图 7.2 算法 1 中的进程 P_i 的结构

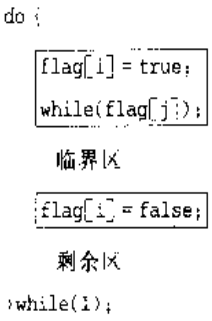


图 7.3 算法 2 中的进程 P_i 的结构

在该算法中，进程 P_i 首先设置 flag[i]为真，以表示进程它准备进入其临界区。接着，P_i 检查并验证进程 P_j 没有准备进入其临界区。如果 P_j 也已就绪，那么 P_i 会等待直到 P_j 表示它不再需要在其临界区内(即，直到 flag[j]为假)。这时，P_i 会进入临界区。在退出临界区时，P_i 会设置 flag[i]为假，以允许其他进程(如果等待)进入其临界区。

这一解答满足了互斥要求，但遗憾的是没有满足前进要求。为了说明这个问题，考虑下面执行顺序：

T_0：P_0 置 flag[0] = true

T_1：P_1 置 flag[1] = true

现在 P_0 和 P_1 在各自的 while 语句内无穷循环。

该算法与两个进程的精确时序有非常密切的关系。这种顺序可出现在如下情况：多个处理器并发执行，或者当执行 T_0 时马上产生了中断(如定时器中断)且 CPU 从一个进程切换到另一个进程。

注意切换设置 flag[i]与检查 flag[j]的值的语句顺序并不能解决问题。事实上，可能出现这样的情况：两个进程可能同时出现在其临界区内，违反了互斥要求。

3. 算法 3

通过结合算法 1 和算法 2 的关键思想，得到了临界区问题的一个正确解答，以满足三个要求。进程共享两个变量：

```
boolean flag[2];
int turn;
```

初时，flag[0] = flag[1] = false，turn 的值无关紧要(但为 0 或 1)。进程 P_i 的结构如图 7.4 所示。

```
do {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);
        临界区
    flag[i] = false;
        剩余区
}while(1);
```

图 7.4 算法 3 中的进程 P_i 的结构

为了进入临界区，进程 P_i 首先设置 flag[i]的值为 true，且设置 turn 的值为 j，从而表示如果另一个进程希望进入临界区，那么它能进入。如果两个进程同时试图进入，那么 turn 会几乎在同时设置成 i 和 j。只有一个赋值语句的结果会保持，另一个也会发生，但会立即被重写。最终 turn 值决定了哪个进程能被允许先进入其临界区。

现在证明这一解答是正确的。需要证明

1. 互斥成立。
2. 前进要求满足。
3. 有限等待要求满足。

为了证明第一点，注意到只有当 flag[j]==false 或者 turn==i 时，进程 P_i 才进入其临界区。而且，注意到如果两个进程同时在其临界区内执行，那么 flag[0] == flag[1] == true。这两点意味着 P_0 和 P_1 不可能成功地同时执行它们的 while 语句，因为 turn 的值只可能为 0 或 1，而不可能同时为两个值。因此，只有一个进程如 P_j，能成功地执行完 while 语句，而进程 P_i 至少必须执行一个附加的语句("turn==j")。而且，由于只要 P_j 在其临界区内，flag[j]==true 和 turn==j 就同时成立。结果是：互斥被满足。

为了证明第 2 点和第 3 点，注意到只要条件 flag[j]==true 和 turn==j 成立，进程 P_i 陷入在 while 循环语句，那么 P_i 就能被阻止进入临界区。如果 P_j 不准备进入临界区，那么 flag[j]==false，P_i 能进入临界区。如果 P_j 已设置 flag[j]为 true 且也在其 while 语句中执行，那么 turn==i 或 turn==j。如果 turn==i，那么 P_i 进入临界区。如果 turn==j，那么 P_j 将进入临界区。然而，当 P_j 退出临界区，它会设置 flag[j]为 false，以允许 P_i 进入其临界区。如果 P_j 重新设置 flag[j]为 true，那么它也必须设置 turn 为 i。因此，由于进程 P_i 执行 while 语句时并不改变变量 turn 的值，所以 P_i 会进入临界区(前进)，且 P_i 最多在 P_j 进入临界区一次后就能进入(有限等待)。

7.2.2 多进程解法

已经证明了算法 3 能解决两个进程的临界区问题。现在，研究一个解决 n 个进程的临

界区问题的算法。该算法称为面包店算法，它的原型是一种用于面包店、冰淇淋店、熟食店、摩托车登记处等必须在混乱中找到顺序的场合中的调度算法。该算法为分布式环境而开发，但是这里只关心与集中式环境有关的算法特性。

在进入商店时，每个客户接收一个号码。具有最小号码的客户先得到服务。然而，面包店算法不能保证两个进程(客户)不会收到同样的号码。在出现相同号码时，具有最小名称的进程先得到服务。即，如果 P_i 和 P_j 收到同样号码，且 $i<j$，那么 P_i 先得到服务。由于进程名称惟一且完全排序，所以这个算法是完全确定的。

共同的数据结构是

```
boolean choosing[n];
int number[n];
```

初时，这些数据结构分别初始化为 false 和 0。为了方便，定义如下符号：

- 若 $a<c$，$(a,b)<(c,d)$，或若 $a==c$ 和 $b<d$ 均成立时，情况亦然。
- $\max(a_0,\cdots,a_{n-1})$ 是数 k，从而 $k\geqslant a_i$ 对 $i=0,\cdots,n-1$ 成立。

采用面包店算法，进程 P_i 的结构如图 7.5 所示。

```
do {
    choosing[i] = true;
    number[i] = max(number[0],number[1],…,number[n-1]) + 1;
    choosing[i] = false;
    for(j = 0;j<n;j++){
        while(choosing[j]);
        while((number[j]!  = 0)&&(number[j],j)<number[i],i,));
    }
        临界区
    number[i] = 0;
        剩余区
}while(1);
```

图 7.5 面包店算法中的进程 P_i 的结构

为了证明面包店算法正确，需要首先证明：如果 P_i 在其临界区内，且 P_k(k! = i) 已经选择了 number[k]! =0，那么(number[i], i) < (number[k], k)。这一算法的证明留做习题 7.3。

有了这一结果，证明互斥满足就简单了。事实上，考虑 P_i 在临界区内且 P_k 试图进入 P_k 临界区。当进程 P_k 执行第二条 while 语句时，j==i，它会发现

- number[i] ! = 0
- (number[i],i) < (number[k], k)

因此，它会继续在 while 语句内循环，直到 P_i 离开其临界区。

如果希望证明前进和有限等待要求得以满足，且算法确保公平，那么只要证明进程按照FCFS基本原则进入临界区就足够了。

7.3 同步硬件

与其他软件一样，硬件特征能简化编程任务且提高系统效率。在本节，介绍一些许多系统都具有的简单硬件指令，并描述如何用它们来有效地解决临界区问题。

对于单处理器环境，临界区问题可简单地加以解决：在修改共享变量时只要禁止中断出现。这样，就能确保当前指令顺序的执行不会被中断。没有其他指令执行，所以共享变量也不会出现意外修改。

然而，在多处理器环境下，这种解决方案是不可行的。在多处理器上因为消息要传递给所有处理器，禁止中断可能很费时。这种消息传递延迟进入每个临界区，导致系统效率降低。而且，该方法影响了系统时钟(如果时钟是通过中断来加以更新的)。

因此，许多系统提供了特殊硬件指令以允许人们能**原子地**(如不可中断单元一样)检查和修改字的内容或交换两个字。可以使用这些特殊指令来相当简单地解决临界区问题。这并不是讨论某个机器的特定指令，而是抽象化这些类型指令的主要思想。

指令 TestAndSet 可以按图 7.6 所示定义。重要特点是该指令能原子地执行。因此，如果两个指令 TestAndSet 同时执行在不同的 CPU 上，那么它们会按任意顺序来顺序执行。

如果机器支持指令 TestAndSet，那么能这样实现互斥：声明一个 Boolean 变量 lock，初始化为 false。进程 P_i 的结构如图 7.7 所示。

```
boolean TestAndSet(boolean &target){
    boolean rv = target;
    target = true;
    return rv;
}
```

图 7.6 TestAndSet 指令的定义

```
do {
    while(TestAndSet(lock));
    临界区
    lock = false;
    剩余区
}while(1);
```

图 7.7 使用 TestAndSet 的互斥实现

指令 Swap，如图 7.8 所示定义，操作两个字的内容；与指令 TestAndSet 一样，它也原子执行。

如果机器支持指令 Swap，那么互斥可按如下方式实现。声明一个全局 Boolean 变量 lock，初始化为 false。另外，每个进程也有一个本地 Boolean 变量 key。进程 P_i 的结构如图 7.9 所示。

这些算法并不满足有限等待要求。这里介绍一个使用指令 TestAndSet 的算法，如图 7.10 所示。该算法满足所有临界区问题的三个要求。共同数据结构如下：

```
void Swap(boolean &a,boolean &b){
  boolean temp = a;
  a = b;
  b = temp;
}
```

图 7.8 Swap 指令的定义

```
do {
  key = true;
  while(key == true)
    Swap(lock,key);
    临界区
  lock = false;
    剩作区
}while(1);
```

图 7.9 使用 Swap 指令的互斥实现

```
boolean waiting[n];
boolean lock;
do {
  waiting[i] = true;
  key = true;
  while(waiting[i] && key)
    key = TestAndSet(lock);
  waiting[i] = false;
    临界区
  j = (i + 1) % n;
  while((j != i) && ! waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = false;
  else
    waiting[j] = false;
    剩余区
}while(1);
```

图 7.10 使用 TestAndSet 的有限等待互斥

这些数据结构均初始化 false。为了证明满足互斥要求，注意到只有 waiting[i] == false 或 key == false 时，进程 P_i 才进入其临界区。只有当 TestAndSet 执行时，key 的值才变为 false。执行 TestAndSet 的第一个进程会发现 key == false；所有其他进程必须等待。只有其他进程离开其临界区时，变量 waiting[i]的值才能变为 false；每次只有一个 waiting[i]被设置为 false，以维护互斥要求。

为了证明满足前进要求，注意到有关互斥的推断也适用。由于进程在退出其临界区时或将 lock 设为 false，或将 waiting[j]设为 false。这两种情况都允许等待进程进入临界区以执行。

为了证明满足有限等待要求，注意到当一个进程退出其临界区时，它会循环地扫描数组 waiting，按循环顺序($i+1$, $i+2$, …, $n-1$, 0, …, $i-1$)，并根据这一顺序而指派在进入区(waiting[j]==true)第一个进程作为下一个进入临界区的进程。因此，任何等待进入临界区的进程只需要等待 $n-1$ 次。然而，对于硬件设计人员，在多处理器上实现原子指令 TestAndSet 并不简单。这种实现会在有关计算机体系结构的书中讨论。

7.4 信 号 量

在 7.3 节中所描述的临界区问题的解决方案不便于推广到更为复杂的问题。为了克服这个困难，可使用称为**信号量**(semaphore)的同步工具。信号量 S 是个整数变量，除了初始化外，它只能通过两个标准原子操作 wait 和 signal 来访问。这些操作原来被称为 P(用于 wait，表测试)和 V(用于 signal，表增加)。wait 的经典定义可用伪代码表示为

```
wait(S) {
  while(S<=0)
    ; // no-op
  S--;
}
```

signal 的经典定义可用伪代码表示为

```
signal(S) {
  S++;
}
```

在 wait 和 signal 操作中，对信号量整数值的修改必须不可分地执行。即当一个进程修改信号量值时，不能有其他进程同时修改同一信号量的值。另外，对于 wait(S)，对 S 的整数值的测试($S\leqslant 0$)和对其可能的修改(S--)，也必须没有中断地执行。本书将在 7.4.2 小节中描述如何实现这些操作；首先，来研究一下如何来使用信号量。

7.4.1 用法

可使用信号量来解决 n 个进程的临界区问题。这 n 个进程共享一个信号量 **mutex**(*mut*ual *ex*clusion)，并初始化为 1。每个进程 P_i 的组织结构如图 7.11 所示。

也可使用信号量来解决各种同步问题。例如，考虑两个正在执行的并发进程：P_1 有语句 S_1 而 P_2 有语句 S_2，假设要求只有在 S_1 执行完之后才执行 S_2，可以很容易地实现这一要

求：让 P_1 和 P_2 共享一个共同信号量 synch，且初始化为 0，在进程 P_1 中插入语句：

S_1；

signal (synch)；

在进程 P_2 中插入语句：

wait(synch)；

S_2；

因为 synch 初始化为 0，P_2 只有在 P_1 已调用 signal(synch)，即 S_1 之后，才会执行 S_2。

```
do {
    wait(mutex);
        临界区
    signal(mutex);
        剩余区
}while(1);
```

图 7.11 使用信号量的互斥实现

7.4.2 实现

7.2 节的互斥解决方案和这里定义的信号量的主要缺点是都要求**忙等待**。当一个进程位于其临界区内时，任何其他试图进入其临界区的进程都必须在其进入代码中连续地循环。这种连续循环在实际多道程序系统中显然是个问题，因为这里只有一个 CPU 为多个进程所共享。忙等待浪费了 CPU 时钟，这本来可有效地为其他进程所使用。这种类型的信号量也称为**自旋锁**(spinlock)(因为进程在等待锁时"自旋")。自旋锁在多处理器系统中是有用的。自旋锁的优点是在进程必须等待一个锁时无需上下文切换，上下文切换可能需要花费相当长的时间。因此，当锁只保留较短时间时，自旋锁就有用了。

为了克服忙等，可修改信号量操作 wait 和 signal 的定义。当一个进程执行 wait 操作时，发现信号量值不为正，则它必须等待。然而，该进程不是忙等而是阻塞自己。阻塞操作将一个进程放入到与信号量相关的等待队列中，且该进程的状态被切换成等待状态。接着，控制被转到 CPU 调度程序，以选择另一个进程来执行。

一个进程阻塞且等待信号量 S，可以在其他进程执行 signal 操作之后被重新执行。该进程的重新执行是通过 wakeup 操作来进行的，该操作将进程从等待状态切换到就绪状态。接着，该进程被放入到就绪队列中。(根据 CPU 调度算法的不同，CPU 有可能会或不会从运行进程切换到刚刚就绪的进程。)

为了实现这样定义的信号量，将信号量定义为如下一个"C"结构：

```
typedef struct {
  int value;
```

```
    struct process *L;
} semaphore;
```

每个信号都有一个整数值和一个进程链表。当一个进程必须等待信号量时，就加入到进程链表上。操作 signal 会从等待进程链表中取一个进程以唤醒。

信号量操作 wait 现在可按如下来定义

```
void wait(semaphore S) {
  S.value--;
  if (S.value < 0) {
     add this process to S.L;
     block();
  }
}
```

信号量操作 signal 现在可按如下来定义

```
void signal(semaphore S) {
  S.value++;
  if (S.value <= 0) {
     remove a process P from S.L;
     wakeup(P);
  }
}
```

操作 block 挂起调用它的进程。操作 wakeup(P)重新启动阻塞进程 P 的执行。这两个操作都是由操作系统作为基本系统调用来提供的。

注意，在具有忙等的信号量经典定义下，信号量的值决不可能为负，但是其实现可能有负的信号量值。如果信号量的值为负，那么其绝对值就是等待该信号量的进程数。出现这种情况是因为操作 wait 实现中递减和测试次序的切换。等待进程的链表可以利用进程控制块 PCB 中的一个链接域来方便地加以实现。每个信号量包括一个整数值和一个 PCB 链表的指针。向链表中添加和删除一个进程以确保有限等待的一种方法可以使用 FIFO 队列，即信号量包括队列的首指针和尾指针。然而，一般来说，链表可以使用*任何*排队策略。信号量的正确使用并不依赖于信号量链表的特定排队机制。

信号量的关键之处是它们原子地执行。必须确保没有两个进程能同时对同一信号量执行操作 wait 和 signal。这种情况属于临界区问题，可通过两种方法来解决。

在单处理器环境下(即只有一个 CPU 存在)，可以在执行操作 wait 和 signal 时简单地禁止中断。这种方案在单处理器环境下能工作，这是因为一旦禁止中断，不同进程指令不会交织在一起。只有当前运行进程执行，直到中断被重新允许和调度器能重新获得控制为止。

在多处理器环境下，禁止中断毫无作用。来自不同进程(运行在不同处理器)的指令可以任意不同方式交织在一起。如果硬件不提供任何特殊指令，那么可以使用临界问题的正

确的软件解决方案以进行(7.2 节),这里临界区包括 wait 和 signal 子程序。

必须承认对于这里的 wait 和 signal 操作的定义,并没有完全取消忙等,而是取消了应用程序进入临界区的忙等。而且,将忙等仅限制在操作 wait 和 signal 的临界区内,这些区比较短(如果适当编码,它们不会超过 10 条指令)。因此,几乎不占用临界区,忙等很少发生,且所需时间很短。对于应用程序,却是一种完全不同的情况,临界区可能很长(数分钟或甚至数小时)或几乎总是占满。这时,忙等极为低效。

7.4.3 死锁与饥饿

具有等待队列的信号量的实现可能导致这样的情况:两个或多个进程无限地等待一个事件,而该事件只能由这些等待进程之一来产生。这里的事件是操作 signal 的执行。当出现这样的状态时,这些进程就称为**死锁**(deadlocked)。

为了加以说明,考虑一个系统由两个进程 P_0 和 P_1 组成,每个进程都访问两个信号量 S 和 Q,这两个信号量的初值均为 1。

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

假设 P_0 执行 wait(S),接着 P_1 执行 wait(Q)。当 P_0 执行 wait(Q)时,它必须等待直到 P_1 执行 signal(Q)。类似地,当 P_1 执行 wait(S),它必须等待直到 P_0 执行 signal(S)。由于这两个 signal 操作都不能执行,那么 P_0 和 P_1 就死锁了。

说一组进程处于死锁状态是指:组内的每个进程都等待一个事件,而该事件只可能由组内的另一个进程产生。这里所关心的主要事件是*资源获取和释放*。然而,如第八章所述,其他类型的事件也能导致死锁。在第八章,将讨论各种机制以处理死锁问题。

与死锁相关的另一个问题是**无限期阻塞**(indefinite blocking)或饥饿(starvation),即进程在信号量内无穷等待的情况。如果对与信号量相关的链表按 LIFO 顺序来添加和删除进程,那么可能会发生无穷阻塞。

7.4.4 二进制信号量

以上各小节所讨论的信号量构架通常称为**计数信号量**,因为其整数值可跨越于一个不受限制的域内。**二进制信号量**(binary semaphore)的值只能为整数值 0 或 1。根据支持硬件的体系结构,二进制信号量可能比计数信号量更容易实现。现在介绍如何用二进制信号量来实现计数信号量。

设 S 为计数信号量。为了用二进制信号量来实现它，需要如下数据结构：

```
binary-semaphore S1, S2;
int C;
```

开始时，S1＝0，S2＝0，整数 C 的值设置为计数信号量 S 的初值。

计数信号量 S 的操作 wait 可实现如下：

```
wait(S1);
C--;
if(C < 0) {
  signal(S1);
  wait(S2);
}
signal(S1);
```

计数信号量 S 的操作 signal 可实现如下：

```
wait(S1);
C++;
if(C <= 0)
  signal(S2);
else
  signal(S1);
```

7.5 经典同步问题

在本节，将介绍若干不同的同步问题以作为大量并发控制问题的例子。这些问题用来测试几乎所有新提出的同步方案。在这里的解决方案中，使用了信号量以处理同步。

7.5.1 有限缓冲问题

*有限缓冲*问题在 7.1 节中介绍；它通常用来说明同步原语的能力。这里，介绍一种该方案的通用结构，而不是只局限于某个特定实现。假定缓冲池有 n 个缓冲项，每个缓冲项能存一个数据项。信号量 mutex 提供了对缓冲池访问的互斥要求，并初始化为 1。**信号量 empty** 和 **full** 分别用来表示空缓冲项和满缓冲项的数量。信号量 empty 初始化为 n；而信号量 full 初始化为 0；

生产者进程的代码如图 7.12 所示；消费者进程的代码如图 7.13 所示。注意生产者和消费者之间的对称性。可以这样来理解代码：生产者为消费者生产满缓冲项，或消费者为生产者生产空缓冲项。

```
do {
    ...
  produce an item in nextp
    ...
  wait(empty);
  wait(mutex);
    ...
  add nextp to buffer
    ...
  signal(mutex);
  signal(full);
}while(1);
```

图 7.12 生产者进程结构

```
do {
  wait(full);
  wait(mutex);
    ...
  remove an item from buffer to nextc
    ...
  signal(mutex);
  signal(empty);
    ...
  consume the item in nextc
    ...
}while(1);
```

图 7.13 消费者进程结构

7.5.2 读者—作者问题

一个数据对象(如文件或记录)可以为多个并发进程所共享。其中有的进程可能只需要读共享对象的内容,而其他进程可能要更新(即读和写)共享对象。为了区分这两种类型的进程,将只对读感兴趣的进程称为**读者**;而其他的则称为**作者**。显然,如果两个读者同时访问共享数据对象,那么不会产生什么不利的结果。然而,如果一个作者和其他一些进程(读者或作者)同时访问共享对象,很可能产生混乱。

为了确保不会产生这样的困难,要求作者对共享对象有完全的访问。这一同步问题称为读者—作者问题。自从它被提出后,就一直用来测试几乎所有新的同步原语。读者—作者问题有多个变种,都与优先级有关。最为简单的,通常称为第一读者—作者问题,要求没有读者需要保持等待除非有一个作者已获得允许以使用共享对象。换句话说,没有读者会因为有一个作者在等待而会等待其他读者的完成。第二读者—作者问题要求,一旦作者就绪,那么作者会尽可能快地执行其写操作。换句话说,如果一个作者等待访问对象,那么不会有新读者开始读操作。

对这两个问题的解答都可能导致饥饿。在第一种情况,作者可能饥饿;在第二种情况,读者可能饥饿。由于这个原因,产生了问题的其他变种。这里介绍一个对第一读者—作者问题的解答。关于读者—作者问题的没有饥饿的解答,请参见推荐读物的有关文献。

对于第一读者—作者问题的解决,读者进程共享以下数据结构:

```
semaphore mutex, wrt;
int readcount;
```

信号量 mutex 和 wrt 初始化为 1;readcount 初始化为 0。信号量 wrt 为读者和作者进程所共用。信号量 mutex 用于确保在更新变量 readcount 时的互斥。变量 readcount 用来

跟踪有多少进程正在读对象。信号量 wrt 供作者作为互斥信号量使用。它为第一个进入临界区和最后一个离开临界区的读者所使用，而不为读者在其他读者处于临界区时进入或离开临界区所使用。

作者进程的代码如图 7.14 所示；读者进程的代码如图 7.15 所示。注意，如果有一个作者在临界区内，且 n 个读者处于等待，那么一个读者在 wrt 上排队，而 $n-1$ 个读者在 mutex 上排队。而且，当一个作者执行 signal(wrt)时，可以重新启动等待读者或一个单独等待作者的执行。这一选择由调度程序去做。

```
wait(wrt);
    ...
  writing is performed
    ...
signal(wrt);
```

图 7.14 作者进程结构

```
wait(mutex);
readcount++;
if (readcount == 1)
  wait(wrt);
signal(mutex);
  ...
  reading is performed
  ...
wait(mutex);
readcount--;
if (readcount == 0)
  signal(wrt);
signal(mutex);
```

图 7.15 读者进程结构

7.5.3 哲学家进餐问题

假设有 5 个哲学家，他们花费一生中的时光思考和吃饭。这些哲学家共用一个圆桌，每个哲学家都有一把椅子。在桌子中央是一碗米饭，在桌子上放着 5 只筷子(图 7.16)。当一个哲学家思考时，他与其他同事不交互。时而，哲学家会感到饥饿，并试图拿起与他相近的两只筷子(他与邻近左、右两人之间的筷子)。一个哲学家一次只能拿起一只筷子。显然，他不能从其他哲学家手里拿走筷子。当一个饥饿的哲学家同时有两只筷子时，他就能不用释放他的筷子而自己吃了。当吃完后，他会放下两只筷子，并再次开始思考。

哲学家进餐问题是个典型的同步问题，这不是因为其本身实际重要性也不是因为计算机科学家不喜欢哲学家，而是一大类并发控制问题的例子。它是需要在多个进程之间分配多个资源且不会出现死锁和饥饿形式的简单表示。

一种简单的解决方法是每只筷子都用一个信号量来表示。一个哲学家通过对信号量执行 wait 操作试图夺取相应的筷子；他会通过对适当信号量执行 signal 操作以释放相应的筷子。因此，共享数据是

```
semaphore chopstick[5];
```

其中，所有 chopstick 的元素被初始化为 1。哲学家 i 的结构如图 7.17 所示。

虽然这一解答确保没有两个相邻哲学家同时进餐，但是这一解答应被丢弃因为它可能会导致死锁。假若这 5 个哲学家同时变得饥饿，且同时拿起他左边的筷子。所有 chopstick 的元素现在均为 0。当每个哲学家试图拿他右边的筷子时，他会永远被延误。

下面列出了多个可能解决死锁问题的方法。在 7.7 节中，会介绍一个不会导致死锁的哲学家进餐问题的解决方案。

- 最多只允许四个哲学家同时坐在桌子上。
- 只有两只筷子都可用时才允许一个哲学家拿起它们(他必须在临界区内拿起两只筷子)。
- 使用非对称解决；即，奇数哲学家先拿起他左边的筷子，接着拿他右边的筷子，而偶数哲学家先拿起他右边的筷子，接着再拿他左边的筷子。

最后，有关哲学家进餐问题的任何满意的解决方案必须确保没有一个哲学家会饿死。没有死锁的解决方案并不能取消饿死的可能性。

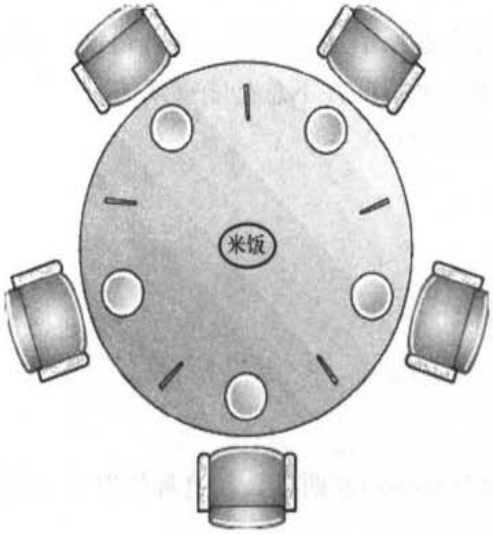


图 7.16 哲学家进餐时的情况

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i + 1) % 5]);
      ...
      eat
      ...
   signal(chopstick[i]);
   signal(chopstick[(i + 1) % 5]);
      ...
      think
      ...
}while(1);
```

图 7.17 哲学家 i 结构

7.6 临界区域

虽然信号量提供了一种方便且有效的机制以处理进程同步，但是其不正确的使用仍然会导致一些定时同步错误并难以检测，因为这些错误只有在一些特定执行顺序情况下才会出现，而这些顺序并不总是出现。

7.1 节介绍 *生产者—消费者* 问题的解答中，在使用计数器时，就出现这样的定时同步错误的例子。在该例子中，定时问题只不过很少出现，而且那时计数器的值看起来似乎合理：

只差 1。然而,这样的解答显然是不能接受的。正是因为这个原因才引入了信号量。

然而,即使采用了信号量,这样的定时错误还是会出现。为了进行说明,回顾一下使用信号量解决临界区问题的解决方案。所有进程共享一个信号量变量 mutex,其初值为 1。每个进程在进入临界区之前必须执行 wait(mutex),之后执行 signal(mutex)。如果这一顺序不被遵守,那么两个进程会同时出现在临界区内。

下面研究一下可能产生的各种困难。注意即使只有一个进程不正确,也会出现这些困难。这种情况可能是诚实的编程错误或不合作的程序员的结果。

• 假设一个进程交换了对信号量 mutex 的 wait 和 signal 操作的执行顺序,从而产生了如下执行情况:

```
signal(mutex);
   ...
  临界区
   ...
wait(mutex);
```

这样,多个进程可能在其临界区内同时执行,进而违反了互斥要求。这种错误只有在多个进程同时在其临界区内执行时才会被发现。注意这种情况并不总是能重现的。

• 假设一个进程用 wait(mutex)替代了 signal(mutex)。即

```
wait(mutex);
   ...
  临界区
   ...
wait(mutex);
```

这样,会发生死锁。

• 假设一个进程省略了 wait(mutex)或 signal(mutex)或两者。在这种情况下,可能会出现死锁,或可能违反互斥。

这些例子说明了当信号量不正确地用来解决临界区问题时,会很容易地产生各种类型的错误。类似问题也会出现在 7.5 节中所讨论的其他同步模型中。

为了处理刚才提出的这些类型的错误,提出了一些高级语言构造。在本节,描述一个基本的高级同步构造,即**临界区域**(有时也称为**条件临界区域**)。在 7.7 节,介绍另一种基本同步构造,即**管程**(monitor)。在介绍这两种构造时,假定进程有一定的局部数据和能对这些数据操作的顺序程序。局部数据只能为同一进程内包容的顺序程序所访问。即一个进程不能直接访问另一进程的局部数据。然而,进程能共享全局数据。

临界区域高级语言同步构造要求:为多个进程所共享的类型为 T 的变量 v,应按如下方式定义:

```
v: shared T;
```

变量 v 只能按如下形式的 region 语句来访问

```
region v when (B) S;
```

这种构造意味着当语句 S 在执行时，没有其他进程能访问变量 v。表达式 B 是布尔逻辑表达式，用来控制对临界区的访问。当进程试图进入临界区时，先计算布尔逻辑表达式 B。如果该表达式的值为真，那么执行语句 S。如果它为假，那么进程就放弃互斥并延迟到 B 为真，并且没有其他进程位于与 v 相关联的区域内。因此，如果如下两个语句

```
region v when (true) S1;
region v when (true) S2;
```

在不同顺序进程中并发执行时，其结果相当于顺序执行"先 S1 后 S2"或"先 S2 后 S1"。

临界区域构造能防止可能由程序员所造成的与临界区问题的信号量解答有关的一些简单问题。注意它并不能必然地消除所有同步问题，而且它只能减少问题的数量。如果错误出现在程序逻辑中，那么重现事件的特定顺序就不会简单了。

临界区域构造能有效地用于解决某些通常类型的同步问题，为了进行说明，来重新编写有限缓冲方法。缓冲空间和其指针可以包含在如下结构中

```
struct buffer {
  item pool[n];
  int count, in, out;
};
```

生产者进程通过执行如下代码将新的项 nextp 加入到共享缓冲：

```
region buffer when (count < n) {
  pool[in] = nextp;
  in = (in + 1) % n;
  count++;
}
```

消费者进程通过执行如下代码以从共享缓冲中移出一项并存在 nextc 中：

```
region buffer when (count > 0) {
  nextc = pool[out];
  out = (out + 1) % n;
  count--;
}
```

下面说明如何用编译器来实现条件临界区域。对于每个共享变量，将有下面的变量与之关联：

```
semaphore mutext, first_delay, second_delay;
int first_count, second_count;
```

信号量 mutex 初始化为 1，信号量 first_delay 和 second_delay 初始化为 0。整数 first_count 和 second_count 初始化为 0。

对临界区的互斥访问是由 mutex 提供的。如果一个进程由于布尔逻辑表达式 B 的值为假而不能进入临界区，那么它最初会等待在信号量 first_delay。在信号量 first_delay 上等待的进程在被允许重新计算布尔逻辑表达式 B 之前，会最终移到信号量 second_delay 上。分别用 first_count 和 second_count 来跟踪在 first_delay 和 second_delay 上等待的进程数量。

当一个进程离开临界区时，它可能会改变某个布尔逻辑表达式 B 的值，以避免其他进程进入临界区。相应地，必须让在 first_delay 和 second_delay 上等待的进程按顺序来检查其布尔逻辑表达式。当一个进程（在跟踪中）检查其布尔条件时，它可能发现后者的值现在重计算为真。在这种情况下，该进程进入其临界区。否则，该进程必须再次在信号量 first_delay 和 second_delay 上等待，如前所述。相应地，对共享变量 x，语句

```
region x when (B) S;
```

可按图 7.18 所示方式来实现。注意这种实现要求在一个进程离开其临界区时，要对每个等待进程的表达式 B 重新计算。如果多个进程被延迟以等待它们的相应布尔逻辑表达式为真时，这种重新计算的额外开销会导致低效代码。可以使用各种优化方法来降低这种额外开销。请参见推荐读物中的有关文献。

```
wait(mutex);
while(! B){
  first_count ++ ;
  if (second_count>0)
    signal(second_delay);
  else
    signal(mutex);
  wait(first_delay);
  first_count - - ;
  second_count ++ ;
  if(first_count>0)
  else
    signal(second_delay)
    signal(first_delay);
  wait(second_delay);
  second_count - - ;
}
S;
if (first_count>0)
  signal(first_delay);
else if(second_count>0)
  signal(second_delay);
else
  signal(mutex);
```

图 7.18 条件区域构造的实现

7.7 管 程

另一高级同步构造是管程类型。一个**管程**（monitor）是由程序员定义的一组操作符来表征的。管程类型的表示包括一组变量的声明（这些变量的值定义了一个类型实例的状态）和实现对这些类型操作的子程序体和函数体。管程的语法如图 7.19 所示。

管程类型的表示不能直接为各个进程所使用。因此，在管程内定义的子程序只能访问位于管程内那些局部声明的变量和其形式参数。类似地，管程的局部变量只能为局部子程序所访问。

管程构造确保一次只有一个进程能在管程内活动。因此，程序员不需要显式地编写同

步限制(图 7.20)。然而,现在所定义的管程构造,还未强大到足以能处理一些同步方案的地步。为此,需要定义一些额外的同步机制。这些机制可由条件(condition)构造来提供。需要编写自己定制的同步方案的程序员能定义一个或多个 *condition* 类型的变量:

```
condition x, y;
```

```
monitor monitor-name
{
    shared variable declarations

    procedure body P1(…){
        …
    }
    procedure body P2(…){
        …
    }
        ⋮
    procedure body Pn(…){
        …
    }
    {
        initialization code
    }
```

图 7.19　管程的语法

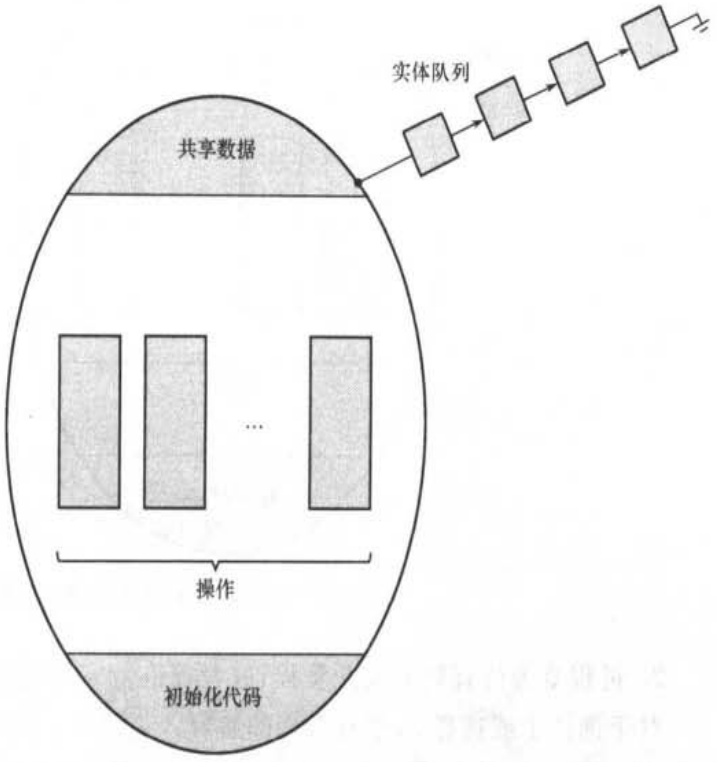


图 7.20　管程的示意图

对条件变量仅有的操作是 wait 和 signal。操作

```
x.wait();
```

意味着调用操作的进程会暂停直到另一进程调用

```
x.signal();
```

操作 x.signal()重新启动一个悬挂的进程。如果没有进程被悬挂,那么操作 signal 就没有作用;即 x 的状态如同没有执行操作一样(图 7.21)。这一操作与信号量相关的操作 signal 不同,后者总能影响到信号量的状态。

现在假设当操作 x.signal()为一个进程 P 所调用时,有一个悬挂进程 Q 与条件变量 x 相关。显然,如果悬挂进程 Q 被允许重启其执行,那么发出信号的进程 P 必须等待。否则,两个进程 P 和 Q 会同时在管程内执行。注意,然而两个进程从概念上来说继续它们的执行。有两种可能性存在:

1. 进程 P 等待直到 Q 离开管程,或者等待另一个条件。

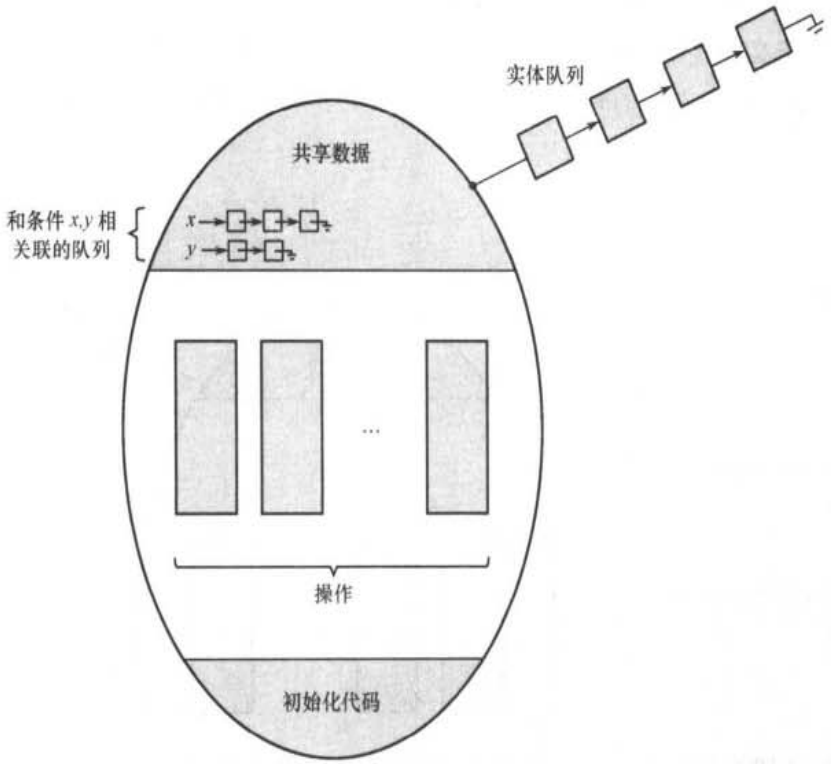


图 7.21 带条件变量的管程

2. 进程 Q 等待直到 P 离开管程,或者等待另一个条件。

对于选择 1 或选择 2,都有合理的解释。由于 P 已经在管程中执行,所以选择 2 似乎更为合理。然而,如果允许进程 P 继续,那么 Q 所等待的"逻辑"条件在 Q 重新启动时可能已不成立。

选择 1 是 Hoare 所提倡的,主要因为前面支持它的理由能直接转换成更为简单和更为优美的证明规则。这两种选择的一个折中为并发 C 语言所采用。当进程 P 执行操作 signal 时,进程 Q 立即重新启动。这种模型比 Hoare 的模型的功能要弱,这是因为一个进程在一个子程序调用内只产生一次信号。

可以通过一个哲学家就餐问题的无死锁解答来说明这些概念。记住一个哲学家只有在两只筷子均可用时才被允许拿起筷子。为了对这个解答编码,需要区分哲学家所处的三种状态。为此,引入如下数据结构:

```
enum {thinking, hungry, eating} state[5];
```

哲学家 i 只有在其两个邻居不就餐时才能将变量 state[i]设置为 eating:(state[(i+4)%5] != eating) 和 (state[(i+1)%5] != eating)。

还需要声明

```
condition self[5];
```

其中,哲学家 i 在饥饿且又不能拿到所需的筷子时可能会延迟自己。

现在可以描述哲学家进餐问题的解答。筷子的分布是由管程 dp 来控制的，管程 dp 的定义如图 7.22 所示。每个哲学家在用餐之前，必须调用操作 pickup。这可能挂起该哲学家进程。在成功完成该操作之后，他就可以进餐。接着，哲学家可调用操作 putdown，并可开始思考。因此，哲学家 i 必须按以下顺序来调用操作 pickup 和 putdown。

```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

很容易看出这一解答确保了相邻两个哲学家不会同时用餐，且不会出现死锁。然而，应注意到哲学家可能会饿死。这里就不再讨论关于这个问题的解答了，而是将它作为练习让您来解决。下面将介绍用信号量来实现管程的一种可能机制。对每个管程，都提供有一个信号量 mutex(初始化为 1)。进程在进入管程之前必须执行 wait(mutex)，在离开管程之后必须执行 signal(mutex)。

因为信号进程必须等待直到重新启动进程或者离开或者等待，所以引入了另一个信号量 next(初始化为 0)以供信号进程挂起自己。还提供了一个整数变量 next_count 以对挂起在 next 上的进程数量进行计数。因此，每个外部子程序 F 会被替换成

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
```

```
monitor dp
{
    enum{thinking,hungry,eating}state[5];
    condition self[5];

    void pickup(int i){
       state[i] = hungry;
       test(i);
       if (state[i] ! = eating)
         self[i].wait();
    }

    void putdown(int i){
       state[i] = thinking;
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    void test(int i){
       if ((state[(i + 4) % 5] ! = eating) &&
         (state[i] = = hungry) &&
         (state[(i + 1) % 5]! = eating)){
            state[i] = eating;
            self[i].signal();
       }
    }

    void init(){
       for (int i = 0;i<5;i++)
          state[i] = thinking;
    }
}
```

图 7.22　哲学家进餐问题的管程解法

```
    signal(mutex);
```

这样，确保了管程内的互斥。

现在介绍如何实现条件变量。对每个条件变量 x，引入信号量 x_sem 和整数变量 x_count，两者均初始化为 0。操作 x.wait 现在可实现如下：

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

操作 x.signal 现在可实现如下：

```
if (x_count>0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

这种实现适用于由 Hoare 和 Brinch Hansen 所给出的管程定义。然而，在有些情况下，这种实现的通用性是不必要的，且有可能在效率上有重大改进。将这个问题作为练习 7.13 留给您来做。

现在讨论管程内进程重新启动的顺序。如果多个进程悬挂在条件 x 上，且某个进程执行了操作 x.signal，那么如何决定下一步应重新启动哪个挂起进程？一个简单解决方案是使用 FCFS 排序，这样等待时间最长的进程先重新启动。然而，在许多情况下，这种简单调度方案是不够的。为此，可使用**条件等待**构造；它具有如下形式

```
x.wait(c)
```

其中，c 是整数表达式，需要在执行操作 wait 时进行计算。c 的值称为**优先号码**(priority number)，会与悬挂进程的名称一起被存储。当执行 x.signal 时，与最小优先号码相关联的进程会被重新启动。

为了说明这种新机制，考虑一个如图 7.23 所示的管程，它用于为多个竞争进程控制对单个资源的分配。每个进程，在请求资源分配时，说明它计划使用资源的最大时间。管程将资源分配给具有最短时间分配请求的进程。

需要访问有关资源的进程必须按如下顺序进行：

```
R.acquire(t);
    ...
```

```
        access the resource;
          ...
      R.release();
```

其中,R 是类型 ResourceAllocation 的实例。

但是,管程概念不能保证以上访问顺序能得到遵守。尤其是:

- 一个进程可能没有先获得资源访问权限就访问资源。
- 一个进程可能在其获得资源访问权限之后就不再释放资源。
- 一个进程可能试图释放一个它从来没有请求的资源。
- 一个进程可能请求同一资源两次(中间没有释放该资源)。

临界区域构造也会碰到同样困难,这些困难与起初鼓励人们开发临界区域和管程构造时的困难的性质相似。以前,必须关注信号量的正确使用。现在,必须关注高级程序员定义操作的正确使用,对此编译器无能为力。

```
monitor ResourceAllocation
{
    boolean busy;
    condition x;

    void acquire(int time){
        if (busy)
            x.wait(time);
        busy = true;
    }

    void release(){
        busy = false;
        x.signal();
    }
    void init(){
        busy = false;
    }
}
```

图 7.23 用来分配单个资源的管程

对以上问题的一种可能的解决办法是将资源访问操作包括在 ResourceAllocation 管程内。然而,这种方法会导致资源调度是根据内置管程的调度算法来完成的,而不是人们自己所编写的调度算法。

为了确保进程遵守适当的顺序,必须检查所有使用管程 ResourceAllocation 和其管理资源的程序。为了确保系统的正确性,有两个条件是必须检查的。第一,用户进程必须总是按正确顺序来对管程进行调用。第二,必须确保一个不合作的进程不能简单地忽略由管程所提供的互斥关口,及在不遵守访问协议的情况下试图直接访问共享资源。只有确保这两个条件,才能保证没有时间依赖性错误会发生,且调度算法不会失败。

虽然这种检查对小的、静态的系统是可能的,但是对于大的或动态的系统是不合理的。这种访问控制问题只能通过由第十八章所讨论的附加机制来解决。

7.8 操作系统同步

下面讨论由 Solaris 和 Windows 2000 操作系统所提供的同步机制。

7.8.1 Solaris 2 中的同步

Solaris 2 操作系统设计成支持实时计算、多线程和多处理器。为了控制访问临界区，Solaris 2 提供了适应性互斥、条件变量、信号量、读－写锁和十字转门。Solaris 2 实现信号量和条件变量的方法与 7.4 节和 7.7 节所述的方法基本相同。在本节，描述适应性互斥、读－写锁和十字转门。

适应性互斥(adaptive mutex)保护访问每个临界数据项。在多处理器系统中，适应性互斥按旋转锁实现的标准信号量而开始。如果数据已加锁，即已在使用，那么适应性互斥有两个选择。如果锁是由正在另一个 CPU 上运行的线程所拥有，那么线程就旋转等待锁成为可用，因为拥有锁的线程可能会很快用完。如果拥有锁的线程现在不处于运行状态，那么线程就阻塞并进入睡眠直到锁释放时被唤醒。当锁不能很快合理释放时，它进行睡眠以避免旋转。由睡眠线程所拥有的锁可能属于这一类。在单处理器系统中，如果另一个线程测试锁，那么持有锁的线程不会旋转，这是因为一时只有一个线程可以运行。因此，在一个单处理器系统中，当线程遇到锁时，它将总是睡眠而非旋转。人们使用适应性互斥方法只保护那些为较短代码段所访问的数据。即，如果锁只持续少于几百条指令，那么就使用互斥。如果代码段比这种情况要长，那么旋转等待就极为低效了。对于较长的代码段，可以使用条件变量和信号量。如果所要的锁已经被占，那么进程就发出等待信号且睡眠。当一个线程释放锁时，它发出一个信号给队列中下一个睡眠线程。线程进入睡眠和唤醒以及相关的上下文切换的额外开销要比在旋转锁上浪费数百条指令的开销要少。

读－写锁用于保护经常被访问但通常只读访问的数据。在这些情况下，读－写锁要比信号量更为有效，因为多个线程可能并发读数据，而信号量只允许顺序访问数据。读－写锁相对来说实现代价较大，因此它们通常只用于很长的代码段。

Solaris 2 使用十字转门以安排等待获取适应锁和读－写锁的线程链表。**十字转门**(turnstile)是个队列结构，队列包含阻塞在锁上的线程。例如，如果一个线程现拥有同步对象的锁，那么所有其他线程试图获取锁时会阻塞并进入该锁的十字转门。当释放该锁时，内核会从十字转门中选择一个线程以作为锁的下一个拥有者。每个同步对象有至少一个线程在此对象的锁上阻塞时，都需要一个独立的十字转门。然而，Solaris 2 不是将每个同步对象与一个十字转门相关联，而是给每个内核线程一个自己的十字转门。用于第一个线程阻塞于同步对象的十字转门成为对象本身的十字转门。以后阻塞于该锁上的线程会被增加到该十字转门。当最初线程最终释放锁时，它会从内核所维护的空闲十字转门列表中获得一个新的十字转门。为了防止**优先级倒置**，十字转门根据**优先级继承协议**(6.5 节)来组织。这意味着如果较低优先级线程现在拥有一个较高优先级线程所阻塞的锁，那么该低优先级线程会暂时继承较高优先级线程的级别。在释放锁之后，线程会返回到它原来的优先级。

注意内核所用的锁机制同样用于用户级线程的实现，因此同样类型的锁在内核内、外都

可用。一个重要实现差别是优先级继承协议。内核阻塞子程序遵守调度程序所使用的内核优先级继承机制，如 6.5 节所述；用户级线程锁机制并不提供这种功能。

因为锁经常被使用，且通常用于关键内核子程序，所以仔细调节其实现和使用能大大地改善性能。为了优化 Solaris 2 的性能，开发人员不断地改善加锁方法。

7.8.2 Windows 2000 中的同步

Windows 2000 操作系统采用了多线程内核，并支持实时应用程序和多处理器。当 Windows 2000 内核访问位于单处理器系统上的全局资源时，它暂时屏蔽所有可能访问该全局资源的中断处理程序的中断。在多处理器系统上，Windows 2000 采用旋转锁来保护对全局资源的访问。与 Solaris 2 一样，内核只使用旋转锁来保护较短代码段。而且，由于效率原因，内核确保拥有旋转锁的线程决不会被抢占。对于内核外线程的同步，Windows 2000 提供了**分配程序对象**(dispatcher object)。采用分配程序对象，线程可根据多种不同机制包括互斥、信号量和事件等来进行同步。共享数据可以通过要求线程获取访问数据的互斥拥有权和使用完后释放拥有权，来加以保护。**事件**是个同步机制，其使用与条件变量相似；即当所要条件出现时会通知等待线程。

分配程序对象可以处于**信号**状态或**非信号**状态。信号状态表示对象可用且线程在获取对象时不会阻塞。非信号状态表示对象不可用且当线程试图获取对象时会阻塞。分配程序对象状态和线程状态之间有一定的关系。当线程阻塞在非信号分配程序对象上时，其状态从就绪转变为等待，且该线程被放到对象的等待队列上。当分配程序对象的状态成为信号时，内核会检查是不是有线程在该对象上等待。如果有，那么内核将改变一个或多个线程的状态，使其从等待状态切换到就绪状态以便重新执行。内核从等待队列中所选择的线程的数量与它们所等待的分配程序对象的类型有关。对于互斥，内核只从等待队列中选择一个线程，因为一个互斥对象只能为单个线程所“拥有”。对于事件对象，内核可选择所有等待事件的线程。

可以采用互斥锁作为例子说明分配程序对象和线程状态。如果一个线程试图获取处于非信号状态的互斥分配程序对象，那么该线程会被挂起，并被放到互斥对象的等待队列上。当互斥移到信号状态(由于另外一个线程释放了该互斥上的锁而引发)，等待该互斥的线程会：

1. 从等待状态移到就绪状态；
2. 获取互斥锁。

7.9 原子事务

临界区的互斥确保了临界区原子地执行。即，如果两个临界区并发执行，那么其结果相

当于它们按某个未知次序顺序执行。虽然这一属性在许多应用领域都有用，但是在许多情况下人们希望确保临界区形成一个逻辑工作单元，要么完全执行，要么一点也不做。一个例子是资金转账，即一个账号要借，另一个账号要贷。显然，为了保持数据一致性，要么同时借和贷，要么不借也不贷。

本节其他部分的内容是关于数据库系统领域。**数据库**关注于数据的存储和提取以及数据的一致性。近来，有一个将数据库系统技术应用于操作系统的热潮。操作系统可看做数据的操作者；因此，也能得益于数据库研究所得的高级技术和模型。例如，操作系统中所使用的用于管理文件的许多特别技术只要使用更为正式的数据库方法便可以更为灵活和强大。在 7.9.2 小节到 7.9.4 小节，会描述这些数据库技术是什么，且它们如何用于操作系统。

7.9.1 系统模型

执行单个逻辑功能的一组指令(或操作)称为**事务**。处理事务的主要问题是不管计算机系统出现什么可能的失败，都要保证事务的原子性。在本节，描述各种确保事务原子性的机制。首先，研究每个时刻只有一个事务能执行的情况。接着，研究多个事务可并发执行的情况。事务是访问且可能更新各种驻留在磁盘文件中的数据项的程序单元。从笔者观点来看，事务只是一系列 read 操作和 write 操作，并以 commit 操作或 abort 操作终止。操作 commit 表示事务已成功终止其执行；操作 abort 表示因各种逻辑错误，事务必须停止其正常执行。已成功完成执行的终止事务称为已**提交**；否则，称为**失败**(aborted)。已提交事务的效果是不能为事务的失败所取消的。

事务也可能因为系统失败而中止其正常执行。无论如何，由于被中止的事务可能已改变了它所访问的各种数据，这些数据的状态与事务在原子执行情况下是不一样的。被中止的事务必须对其所修改的数据的状态不产生任何影响，以便确保原子特性。因此，被中止的事务所访问的数据状态必须恢复到事务刚刚开始执行之前。称这样的事务已经回退(rolled back)。确保这一属性是系统的一部分责任。

为了决定系统如何确保原子性，首先需要识别用于存储事务所访问的各种数据的设备的属性。不同存储媒介的类型可以通过它们的相对速度、容量和对失败的适应性来区分。

- **易失性存储**：驻留在易失性存储上的信息通常在系统崩溃后不能保存。内存和高速缓存就是这种存储的例子。对易失性存储的访问非常快，这是由于内存访问本身的速度和能够直接访问易失性存储内的任何数据项。
- **非易失性存储**：驻留在非易失性存储上的信息通常在系统崩溃后能保存。磁盘和磁带就是这种存储介质的例子。磁盘比内存更为可靠，而磁带比磁盘更为可靠。然而，磁盘和磁带也会出错，从而导致信息遗失。当前，非易失性存储要比易失性存储慢数个数量级，因为磁盘和磁带设备是机电的且要求物理运行以访问数据。

• **稳定存储**:驻留在稳定存储上的信息决不损失("决不"应该不正确,因为从理论上来说这样的保证是不成立的)。为了实现这种存储的近似,需要在多个非易失性存储介质(通常是磁盘)上以独立失败模式复制信息,并按一定的控制方式来更新信息。

这里只关心在不稳定存储上出现信息损失失败的情况下,确保事务原子性。

7.9.2 基于日志的恢复

确保原子性的一种方法是在稳定存储上记录有关事务对其访问的各种数据所做各种修改的描述信息。实现这种形式记录的最为常用方法是**先记日志后操作**。系统在稳定存储上维护一个称为**日志**的数据结构。每个日志记录描述了一个事务写出的单个操作,并具有如下域:

• **事务名称**:执行写操作事务的惟一名称。

• **数据项名称**:所写数据项的惟一名称。

• **旧值**:写操作前的数据项的值。

• **新值**:写操作后的数据项的值。

其他特别日志记录用于记录事务处理的重要事件,如事务开始和事务的提交或放弃。

在事务 T_i 开始执行前,记录<T_i starts>被写到记录。在执行时,T_i 的每个写操作之前都要将适当新记录先写到日志。当 T_i 提交时,记录<T_i commits>被写到记录。

因为日志信息用于构造各种事务所访问数据项的状态,所以在相应日志记录写出到稳定存储之前不能允许真正地更新数据项。因此,要求在执行操作 write(X)之前,对应于 X 的日志记录要先写到稳定存储上。

注意这种系统内在的性能惩罚。对每个逻辑写请求,需要两个物理写。而且,需要更多存储:用于数据本身和变更日志。当数据极为重要且快速出错恢复是必要时,是值得的。

采用日志,系统可处理任何出错,以便不会在非易失性存储上造成信息损失。恢复算法采用两个步骤:

• undo(T_i):事务 T_i 更新的所有数据的值恢复到原来值。

• redo(T_i):事务 T_i 更新的所有数据的值被设置成新值。

由 T_i 所更新的数据集与原来值和新值的集合可以在日志中找到。

操作 undo 和 redo 必须幂等(即,一个操作的多次执行与一次执行有同样结果),以确保正确行为(即使恢复过程是否有差错发生)。

如果事务 T_i 夭折,那么可仅通过执行 undo(T_i)以恢复所更新数据的状态。如果系统出现差错,那么可通过检测日志确定哪些事务需要重做而哪些事务需要撤消来恢复所有更新数据的状态。这种事务分类可按如下方式进行:

• 如果日志包括<T_i starts>记录但没有包括<T_i commits>记录,那么事务 T_i 需要被撤消。

• 如果日志包括<T_i starts>和<T_i commits>记录，那么事务 T_i 需要重做。

7.9.3 检查点

当系统出现了差错，必须检查日志以确定哪些事务需要重做而哪些需要被撤消。从原理上来说，需要搜索整个日志以便做出这些决定。这种方法有两个主要缺点：

1. 搜索进程费时。

2. 绝大多数根据算法需要重做的事务已经如日志记录所说那样更新了数据。虽然重做数据修改并没有什么损坏(因为幂等)，但是它会导致恢复需要较长时间。

为了降低这些类型的额外开销，引入**检测点**(checkpoint)概念。在执行时，系统维护写前日志。另外，系统定期执行检查点并需要执行如下动作：

1. 将当前驻留在易失性存储(通常为内存)上的所有日志记录输出到稳定存储上。

2. 将驻留在易失性存储上的所有修改数据输出到稳定存储上。

3. 在稳定存储上输出一个日志记录<checkpoint>。

日志记录的<checkpoint>记录允许系统简化其恢复过程。考虑一个在检测点之前已提交的事务 T_i。记录<T_i commits>在日志中，出现在记录<checkpoint>之前。T_i 所做任何修改必须在检查点之前或与检测点部分一起，已经写入稳定存储。因此，在恢复时，没有必要对 T_i 进行重做操作。

这一特点允许人们重新调整先前的恢复算法。在出现差错之后，恢复程序检查日志以确定在最近检查点之前开始执行的最近事务 T_i。可以这样查找这种事务：首先向后搜索日志以查找第一个记录<checkpoint>，接着再查找后续记录<T_i start>。

一旦找到事务 T_i，操作 redo 和 undo 只需要应用于事务 T_i 和自 T_i 之后开始执行的所有事务 T_j。下面用集合 T 表示这些事务。日志其他部分可以被忽略。因此，所需恢复操作如下：

• 对于属于 T 的所有事务 T_k，只要记录<T_k commits>出现在日志中，就执行 redo(T_k)。

• 对于属于 T 的所有事务 T_k，只要记录<T_k commits>没有出现在日志中，就执行 undo(T_k)。

7.9.4 并发原子事务

因为每个事务是原子性的，所以事务的并发执行必须相当于这些事务按某任意顺序串行执行。这一属性，称为**串行化**(serializability)，可以通过在临界区域内执行每个事务来实现。即所有这些事务共享一个信号量 *mutex*，其初始值为 0。当事务开始执行时，其第一个动作是执行 wait(*mutex*)。在事务提交或失败之后，它执行 signal(*mutex*)。

虽然这种方案确保了所有并发执行事务的原子性，但是其限制太大。正如大家将要看

到的，在许多情况下，可允许这些事务的执行互相重叠，而又能保证其串行化。有多个不同**并发控制**算法可确保串行化。下面将对此进行讨论。

1. 串行化能力

考虑一个系统，其中有两个数据项 A 和 B 为两个事务 T_0 和 T_1 所读和写。假设这两个事务按先 T_0 后 T_1 的顺序来原子地执行。这个执行顺序，称为一个**调度**，如图 7.24 所示。在图 7.24 所示的调度 1 中，指令执行顺序按时间顺序从上到下，并且 T_0 的指令出现在左列，而 T_1 的指令出现在右列。

每个事务原子执行的调度称为**串行调度**。每个串行调度由不同事务指令的序列组成，其中属于单个事务的指令在调度中出现在一起。因此，对于 n 个事务的集合，共有 $n!$ 个不同的有效的串行调度。每个串行调度都是正确的，因为它相当于各个参与事务按某一任意顺序的原子执行。

如果允许两个事务重叠执行，那么这样的调度就不再是串行的。**非串行调度**不一定意味着其执行结果是不正确的(即，与串行调度是不同的)。为了说明这种情况，需要定义名词冲突操作。考虑一个调度 S，有两个事务 T_i 和 T_j，其操作的执行顺序分别为 O_i 和 O_j。如果 O_i 和 O_j 访问同样数据项并且至少这些操作之一为 write 操作，那么称 O_i 和 O_j 冲突。为了说明冲突操作这一概念，考虑图 7.25 所示的非串行调度 2。T_0 的操作 write(A) 与 T_1 的操作 read(A) 冲突。然而，T_1 的操作 write(A) 与 T_0 的操作 read(B) 并不冲突，因为这两个操作访问不同的数据项。

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

图 7.24 调度 1：T_1 跟着 T_0 的串行调度

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

图 7.25 调度 2：并发串行调度

设 O_i 和 O_j 是调度 S 的顺序操作。如果 O_i 和 O_j 是不同事务的操作，且 O_i 和 O_j 并不冲突，那么可交换 O_i 和 O_j 的顺序以产生新调度 S'。希望 S 跟 S' 是相同的，因为除了 O_i 和 O_j 外(其顺序无所谓)，两个调度的所有其他操作顺序完全相同。

下面通过考虑图 7.25 所示的调度 2 来研究交换这一方法。由于 T_1 的操作 write(A) 与 T_1 的操作 read(B) 不冲突，可交换这两个操作以产生相同的调度。不管初始系统状态如何，两个调度产生了同样的最后系统状态。继续这种非冲突操作的交换过程，可以得到：

- 交换 T_0 的操作 read(B)和 T_1 的操作 read(A)。
- 交换 T_0 的操作 write(B)和 T_1 的操作 write(A)。
- 交换 T_0 的操作 write(B)和 T_1 的操作 read(A)。

这些交换的最后结果是图 7.24 的调度 1，这是个串行调度。因此，可以证明调度 2 相当于一个串行调度。这一结果表示不管初始系统状态如何，调度 2 会产生与某个串行调度相同的最后状态。

如果调度 S 可以通过一系列非冲突操作的交换而转换成串行调度 S'，称调度 S 为**冲突可串行化**(*conflict serializable*)的。因此，调度 2 是冲突可串行化的，因为它可被换成串行调度 1。

2. 加锁协议

确保串行化能力的一种方法是为每个数据项关联一个锁，并要求每个事务遵循加锁协议以控制锁的获取与释放。对数据项加锁有许多模式。在这里，只讨论两种模式。

- **共享**：如果事务 T_i 获得了数据项 Q 的共享模式锁(记为 S)，那么 T_i 可读取这一项，但不能对 Q 进行写操作。
- **排他**：如果事务 T_i 获得了数据项 Q 的排它模式锁(记为 X)，那么 T_i 可读和写 Q。

要求每个事务根据其对数据项 Q 所要进行操作的类型，来按适当模式来请求数据项 Q 的锁。

为了访问数据项 Q，事务 T_i 必须首先按适当模式锁住 Q。如果 Q 当前未被锁上，那么就允许加锁，T_i 就可访问它了。然而，如果数据项 Q 已为某个其他事务所加锁，那么 T_i 可能必须等待。具体地说，假定 T_i 请求 Q 的排它锁。在这种情况下，T_i 必须等待直到 Q 上的锁被释放为止。如果 T_i 请求 Q 的共享锁，而且 Q 已按专用方式加锁，那么 T_i 必须等待。否则，它可得到锁并访问 Q。注意这种方案十分类似于 7.5.2 小节所描述的读—写算法。

一个事务可以释放其以前加在某个数据项上的锁。然而，它在访问该数据项时，必须锁住它。另外，一个事务在最终完成其数据项访问之后，因为可能不需确保串行化能力，所以不是总需要事务马上释放数据项的锁。

确保串行化能力的一种协议是**两阶段加锁协议**(two-phase locking protocol)。这个协议要求每个事务按两个阶段来发出加锁和放锁请求：

- **增长阶段**：事务可获取锁，但不能释放任何锁。
- **收缩阶段**：事务可释放锁，但不能获取任何新锁。

开始时，事务处于增长阶段。事务根据其需要而获取锁。一旦事务释放锁，它就进入收缩阶段，而不再做加锁请求。

两阶段加锁协议确保了冲突串行化(习题 7.23)。然而，它不能确保不会死锁。要注意可能有这样的情况：对于某组事务，存在冲突串行化调度，但却不能通过两阶段加锁协议来实现。然而，为了改善两阶段加锁协议的性能，需要知道有关事务的更多信息或增加某些结

构或对资料集进行重排。

3. 基于时间戳的协议

对于以上所描述的加锁协议，每对冲突事务的顺序是在执行时它们所请求的第一次锁所决定的，这些锁可能涉及不兼容模式。确定串行化顺序的另一方法是事先在事务之间选择一个顺序。这样做的最为常用方法是使用**时间戳排序**(timestamp-ordering)方案。

对于系统内的每个事务 T_i，都为其关联一个惟一固定时间戳，并记为 TS(T_i)。这个时间戳在事务 T_i 开始执行前，由系统赋予。如果一个事务 T_i 已经被赋予了时间戳 TS(T_i)，那么对以后进入系统的新事务 T_j，就有 TS(T_i) <TS(T_j)。有两个简单方法可实现这种方案：

• 采用系统时钟值作为时间戳；即，事务时间戳等于当事务进入系统时的时钟值。这种方法不适用于处于不同系统上的事务，也不适用于不共享时钟的多处理器系统。

• 采用逻辑记数器作为时间戳；即，事务时间戳等于当事务进入系统时的计数器的值。在赋予了新时间戳之后，计数器的值会增加。

事务时间戳决定了串行化的顺序。因此，如果 TS(T_i) <TS(T_j)，那么系统必须确保所产生的调度相当于事务 T_i 在事务 T_j 之前出现的串行化调度。

为了实现这个方案，为每个数据项 Q 关联两个时间戳的值：

• **W-timestamp**(Q)：表示成功执行 write(Q)的任何事务的最大时间戳。

• **R-timestamp**(Q)：表示成功执行 read(Q)的任何事务的最大时间戳。

只要执行指令 read(Q)和 write(Q)时，就更新这些时间戳。

时间戳顺序协议确保任何冲突操作 read 和 write 按时间戳顺序执行。这个协议工作如下：

• 假定事务 T_i 发出 read(Q)：

○ 如果 TS(T_i) <W-timestamp(Q)，那么这个状态表示 T_i 需要读 Q 的值而其值已经被改写。因此，操作 read 被拒绝，T_i 就返回。

○ 如果 TS(T_i) ≥ W-timestamp(Q)，那么就执行操作 read，R-timestamp(Q)就设置成 R-timestamp(Q)和 TS(T_i)的最大值。

• 假定事务 T_i 发出 write(Q)：

○ 如果 TS(T_i) <R-timestamp(Q)，那么这个状态表示 T_i 正在产生的 Q 的值以前被需要，T_i 假定这个值决不会产生。因此，操作 write 被拒绝，T_i 就返回。

○ 如果 TS(T_i) ≥ W-timestamp(Q)，那么这个状态表示 T_i 正在试图写 Q 的陈旧值。因此，操作 write 被拒绝，T_i 就返回。

○ 否则，就执行操作 write。

由于发出操作 read 或 write 而被并发控制算法返回的事务 T_i，被赋予新时间戳并重新开始。

为了举例说明这个协议，考虑图 7.26 所示的调度 3，它具有两个事务 T_2 和 T_3。假定事务在第一个指令之前被赋予一个时间戳。因此，在调度 3 中，$TS(T_2) < TS(T_3)$，在时间戳协议条件下这个调度是可能的。

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

图 7.26 调度 3：时间戳协议下的可能调度

这个执行也可由两阶段加锁协议产生。然而，在两阶段加锁协议下有的调度是可能的而在时间戳协议下却不可能；反之亦然（习题 7.24）。

时间戳排序协议确保冲突串行化能力。这种能力是由于冲突操作按时间戳顺序来处理的。该协议确保避免了死锁，因为没有事务需要等待。

7.10 小 结

对于共享数据的一组协作的顺序进程必须提供互斥。一种解决方法是确保在某个时候只能有一个进程或线程可使用代码的临界区域。在假定只有存储式联锁可用时，可有许多不同算法解决临界区问题。

这些用户编码解决方案的主要缺点是它们都需要忙等待。信号量可克服这个困难。信号量可用于解决各种同步问题，且可高效地加以实现（尤其是在硬件支持原子操作时）。

各种不同的同步问题（如有限缓冲问题、读—写问题和哲学家进餐问题）均很重要，这是因为这些问题是大量并发控制问题的例子。这些问题用于测试几乎所有新提出的同步方案。

操作系统必须提供机制以防止定时出错。人们提出了多个不同语言结构以处理这些问题。临界区域可用于安全、高效地解决互斥和任意同步问题。管程为共享抽象数据类型提供了同步机制。条件变量提供了一个方法供管程程序来阻塞其执行直到其被通知可继续为止。

Solaris 2 是个典型的现代操作系统，它实现了各种锁以支持多任务、多线程（包括实时线程）和多处理。当短代码段保护数据时，它使用适应性互斥。当更长代码段需要访问数据时，可使用条件变量和读—写锁。Solaris 使用十字转门以对等待要求适应性互斥或读—写锁的线程链表进行排序。

Windows 2000 支持实时进程和多处理。当内核试图访问单处理器系统的全局资源时，可以屏蔽访问全局资源的中断。对于多处理器系统，可以通过自旋锁来保护全局资源。在内核之外，通过分配程序的对象来提供同步。分配程序的对象可用做互斥、信号量或事件。事件是一种类型的分配程序的对象，它类似于条件变量。

事务是个程序单元，必须原子化执行；即，要么与其相关的所有操作执行完，要么什么操作都不做。为了确保原子性（即使在系统出错时），可使用写前记录。所有更新都被记录在日志上，而日志被保存在稳定存储上。如果系统出现死机，日志信息可用于恢复更新数据项

的状态，这由操作 undo 和 redo 来完成。为了降低在系统出错时搜索日志的额外开销，可使用检测点方案。

当多个事务重叠执行时，这样的执行可能不再与当这些事物原子执行时相同。为了确保正确执行，必须使用并发控制方案以保证串行化。有各种不同并发控制方案以确保可串行性：或者延迟操作或撤销发布这个操作的事务。最为常用方法是加锁协议和基于时间戳排序算法。

习 题 七

7.1 术语忙等的含义是什么？操作系统里其他种类的等待有哪些？忙等能否完全避免？为什么？

7.2 解释为什么自旋锁对单处理器系统不合适而对多处理器系统合适？

7.3 证明面包店算法(7.2.2 小节)具有以下特性：如果 P_i 在它的临界区，并且 $P_k(k\neq i)$ 已经选择了它的 number[k]!=0，那么(number[i],i)<(number[k],k)。

7.4 第一个著名的正确解决了两个进程的临界区问题的软件方法是 Dekker 设计的。两个进程 P_0 和 P_1 共享以下变量：

```
boolean flag[2];      /* initially false */
int turn;
```

进程 P_i(i==0 或 1)和另一个进程 P_j(j==0 或 1)的结构见图 7.27。

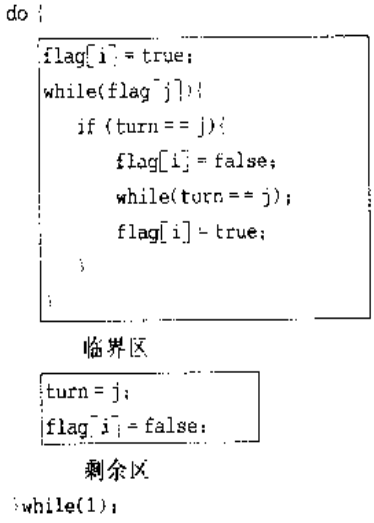


图 7.27 Dekker 算法中的进程 P_i 结构

证明这个算法满足临界区问题的所有三个要求。

7.5 第一个将等待次数降低到 n−1 范围内的正确解决 n 个进程临界区问题的软件解决方法是 *Eisenberg* 和 *McGuire* 设计的。这些进程共享以下变量：

```
enum pstate{idle, want_in, in_cs};
pstate flag[n];
int turn;
```

flag 的所有成员被初始化为 idle；turn 的初始值无关紧要(在 0 和 $n-1$ 之间)。进程 P_i 的结构如图 7.28 所示。

证明这个算法满足临界区问题的所有三个要求。

7.6 在 7.3 节提到经常关闭中断会影响系统的时钟。解释为什么会这样，怎样才能使这种影响最小？

7.7 请说明如果 wait 和 signal 操作不是原子化执行，那么互斥可能是不稳定的。

7.8 **理发店问题**。一个理发店有一间配有 n 个椅子的等待室和一个有理发椅的理发室。如果没有顾客被服务，理发师就去睡觉。如果顾客来时所有的椅子上都有人，那么顾客离去。如果理发师在忙而有空闲的椅子，那么顾客会坐在其中一个空闲的椅子上。如果理发师在睡觉，顾客会摇醒他。写个程序来协调理发师和顾客。

```
do {
    while(1){
        flag[i] = want_in;
        j = turn;
        while(j != i){
            if (flag[j] != idle)
                j = turn;
            else
                j = (j + 1) % n;
        }
        flag[i] = in_cs;
        j = 0;
        while((j<n) && (j == i || flag[j] != in_cs))
            j++;
        if((j>= n) && (turn == i || flag[turn] == idle))break;
    }
    turn = i;
        临界区
    j = (turn + 1) % n;
    while(flag[j] == idle)
        j = (j + 1) % n;
    turn = j;
    flag[i] = idle;
        剩余区
}while(1);
```

图 7.28 Eisenberg 和 McGuire 算法中的进程 P_i 结构

7.9 **抽烟者问题**。假设一个系统有三个*抽烟者*进程和一个*供应*进程。每个抽烟者不停地卷烟并抽掉它。但是要卷起并抽掉一个烟，抽烟者需要有三种材料：烟草、纸和胶水。一个抽烟者有纸，另一有烟草，第三个有胶水。供应进程无限供应所有三种材料，供应者每次将两种材料放到桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉，并给供应者一个信号告诉完成了。供应者就会放另外的两种材料在桌子上，这种过程一直重复。写个程序同步供应者与抽烟者。

7.10 论证管程、条件临界区和信号量都是等同的，在同类型的同步问题中都可以使用。

7.11 写一个内含缓冲的管程，在这个管程中，缓冲区部分被内嵌在管程内部。

7.12 在管程中严格互斥使得题 7.11 中的内含缓冲的管程主要是对小的关键区域合适。

a. 解释为什么这种说法是正确的。

b. 设计一个新的适合大的关键区域的方案。

7.13 假设 signal 语句只能作为一个管程中的最后一条语句出现，可以怎样简化 7.7 节中所描述的实现。

7.14 假设一个系统有进程 $P_1, P_2, \cdots, P_n$，每个进程有一个不同的优先级数。写一个管程来分配三个相同的行式打印机给这些进程，用优先级数来决定分配的顺序。

7.15 一个文件被多个不同进程共享，每个进程有一个不同的数。文件在下面限制条件下可被多个进程同时访问：所有当前访问文件的进程的数的和必须小于 n。写一个管程协调对这个文件的访问。

7.16 假设将管程中的 wait 和 signal 操作替换成一个单一的构件 await(B)，这里 B 是一个普通的布尔表达式，进程执行直到 B 变成真。

a. 用这种方法写一个管程实现读者-写者问题。

b. 解释为什么一般来说这种结构实现的效率不高。

c. 要使这种实现达到高效率需要对 await 语句加上哪些限制？（提示：限制 B 的一般性，参见 Kessels[1977]）

7.17 写一个管程来实现一个*闹钟*，要求它能够通过调用一个程序来让它自身延迟达指定个时间单元(*ticks*)。你可以假设存在一个硬件时钟在规则的间隔时间调用你的管程中的 tick 过程。

7.18 为什么 Solaris 2 实现了多种锁机制？在什么情况下它使用自旋锁、信号量、适应性互斥、条件变量和读-写锁？使用每种机制的原因？十字转门的目的是什么？

7.19 为什么 Solaris 2 和 Windows 2000 都使用自旋锁作为多处理器系统的同步机制而不作为单处理器系统的同步机制。

7.20 就代价而言，易失的、非易失的和稳定的三种存储类型有什么不同？

7.21 检测点机制的用途是什么？应该多长时间执行一次检测？检测的频率会怎样影响：

- 系统性能但没有失败出现？
- 从系统崩溃恢复的时间？
- 从磁盘崩溃恢复的时间？

7.22 解释事务原子性的概念。

7.23 证明两段锁协议能确保冲突的串行执行。

7.24 证明有些调度在两段锁协议下是可行的而在时间戳协议下却不可行，反之亦然。

推荐读物

Dijkstra[1965a]的经典论文里最先论述了关于两个进程的互斥算法 1 和 2。荷兰的数学家 T. Dekker 开发了第一个关于两个进程互斥问题的正确软件解决方法——Dekker 算法（习题 6.3）。Dijkstra[1965a]也论述了这个算法。Peterson[1981]给出了一个关于两个进程互斥问题的简单解决方法（算法 3）。

Dijkstra[1965b]给出了第一个有关 n 个进程互斥问题的解决方案，可是在这个方法里，没有给出一个进程在被允许进入临界区以前必须等待的次数上限。Knuth[1966]给出了第一个有限制的算法，它的限制是 2 的 n 次方。deBruijn[1967]改进了 Knuth 算法，将等待次数减少到 n^2 次。后来 Eisenberg 和 McGuire[1972]（习题 6.4）成功的将次数减少到 $n-1$ 次。Lamport[1974]开发了面包店算法（算法 5）；它也要等待 $n-1$ 次，但它更容易被编程实现和理解。Burns[1978]开发了满足有限制等待要求的硬件解决算法。

Lamport[1986]和 Lamport[1991]给出了关于互斥问题的一般论述。Raynal[1986]给出了一个关于互斥的算法的集合。

Dijkstra[1965a]提出了信号量的概念。Patil[1971]分析了信号量能否解决所有可能的同步

难题的问题。Parnas[1975]指出了 Patil 论述中的缺陷。Kosaraju[1975]继续 Patil 的工作提出了一个不能用 wait 和 signal 操作解决的问题。Lipton[1974]论述了各种同步原语的限制。

本书所叙述的经典进程协同问题是一大类并发控制问题的范例。有界限缓冲问题、哲学家进餐问题和睡眠理发员问题(习题 6.7)是由 Dijkstra[1965a]和 Dijkstra[1971]提出的。Patil[1971]提出了抽烟者问题(习题 6.8)。Courtois 等[1971]提出了读－写者问题。Lamport[1977]论述了并发读－写问题。Lamport[1976]论述了独立进程的同步问题。

Hoare[1972]和 Brinch Hansen[1972]提出了临界区概念。Brinch Hansen[1973]提出了管程的概念。Hoare[1974]给出了管程的完整描述。Kessels[1977]提出了一个允许自动发信号的扩展管程。BenAri[1990]给出了关于并发编程的一般论述。

Mauro 和 McDougall[2001]里有一些关于 Solaris 2 中锁机制实现的细节。注意内核用的锁机制对用户级线程也一样，所以在内核内、外都有同样类型的锁。在 Solomon 和 Russinovich[2000]中可以找到关于 Windows 2000 同步的细节。

Gray 等[1981]第一个介绍了 System R 中的写前记录方案。Eswaran 等[1976]联系他们工作中 System R 的并发控制阐述了串行执行的概念。Eswaran 等[1976]提出了两段锁协议。Reed[1983]提出了以时间戳为基础的并发控制方案。Bernstein 和 Goodman[1980]中讨论了多种不同以时间戳为基础的并发控制算法。

第八章 死 锁

在多道程序设计环境下，多个进程可能竞争一定数量的资源。一个进程申请资源，如果资源不可用，那么进程进入等待状态。如果所申请的资源被其他等待进程占有，那么该等待进程有可能无法改变状态，这种情况称为**死锁**(deadlock)。在第七章，已经在与信号量相关的情况下，讨论了这类情况。

最好的死锁例子可能是 Kansas 立法机构于 20 世纪早期通过的一个法规，其中说到"当两列列车在十字路口逼近时，它们要完全停下来，且在一列列车开走之前另一列列车不能启动。"

在本章，描述操作系统用于预防或处理死锁的各种方法。虽然现在绝大多数操作系统并不提供死锁预防功能，但是这些功能可能很快会加入。随着操作系统的不断发展：如更多数量的进程、多线程程序、更多的系统资源、永久文件和数据库服务的越来越受重视(而不是批处理)，死锁问题会变得更普通。

8.1 系统模型

系统拥有一定数量的资源，分布在若干竞争进程之间。资源分成多种类型，每种类型有相同数量的实例。资源类型的例子有内存空间、CPU 周期、文件、I/O 设备(打印机和磁带驱动器)。如果系统有两个 CPU，那么资源类型 CPU 就有两个实例。类似地，资源类型 *打印机* 可能有 5 个实例。

如果一个进程申请某个资源类型的一个实例，那么分配*任何*这种类型的实例都可满足申请。否则，这些实例就不相同，而资源类型的分类也没有正确定义。例如，一个系统有两台打印机。如果没有人关心哪台打印机打印哪些输出，那么这两台打印机可定义为属于同一资源类型。然而，如果一台打印机在第九层楼上，而另一台在底层楼下，那么第九层楼的用户就不会认为这两台打印机是相同的，这样每个打印机就可能需要定义成属于不同类型。

进程在使用资源前必须申请资源，在使用资源之后必须释放资源。一个进程可能会申请许多资源以便完成其指定的任务。显然，所申请的资源数量不能超过系统所有的资源的总量。换言之，如果系统只有两台打印机，那么进程就不能申请三台打印机。

在正常操作模式下，进程按如下顺序使用资源：

1. **申请**：如果申请不能立即被允许(例如，所申请资源正在为其他进程所使用)，那么申

请进程必须等待直到它获得该资源为止。

2. **使用**：进程对资源进行操作（例如，如果资源是打印机，那么进程就可以在打印机上打印了）。

3. **释放**：进程释放资源。

如第三章所述，资源的申请与释放为系统调用，例如系统调用 request/release device、open/close file、allocate/free memory。其他资源的申请与释放可以通过信号量的 *wait* 与 *signal* 操作来完成。因此，对于每次使用，操作系统会检查以确保使用进程已经申请并获得了资源。系统表记录了每个资源是否空闲或已被分配，如果分配了那么是分配给了哪个进程。如果进程所申请的资源正在为其他进程所使用，那么该进程会被增加到资源的等待队列。

当一组进程的每个进程等待一个事件，而这一事件只能由这一组进程的另一进程引起，那么这组进程就处于死锁状态。这里所关心的主要事件是资源获取和释放。资源可能是物理资源（例如，打印机、磁带驱动器、内存空间和 CPU 周期）或逻辑资源（例如，文件、信号量和监视器）。然而，其他类型事件也会导致死锁（例如，第四章所讨论的 IPC 功能）。

为了说明死锁状态，考虑一个具有三个磁带驱动器的系统。假定有三个进程，每个进程都占用了一个磁带驱动器。如果每个进程现在需要另一个驱动器，那么这三个进程会处于死锁状态。每个进程都在等待事件"磁带驱动器释放"，这只可能由一个等待进程来完成。这个例子说明了涉及同一种资源类型的死锁。

死锁也可能涉及不同资源类型。例如，考虑一个系统，具有一台打印机和一台驱动器。假如进程 P_i 占有磁带驱动器而 P_j 占有打印机。如果 P_i 申请打印机而 P_j 申请磁带驱动器，那么就会出现死锁。

开发多线程应用的程序必须特别关注这个问题：因为多个线程可能因为竞争共享资源而容易产生死锁。

8.2 死锁特点

当出现死锁时，进程不能完成执行，且系统资源被占用，阻止了其他作业开始执行。在讨论各种方法处理死锁问题之前，先讨论一下死锁的特征。

8.2.1 必要条件

如果在一个系统中下面四个条件同时满足，那么会引起死锁。

1. **互斥**：至少有一个资源必须处于非共享模式；即一次只有一个进程使用。如果另一资源申请该资源，那么申请进程必须延迟直到该资源释放为止。

2. **占有并等待**：一个进程必须占有至少一个资源，并等待另一资源，而该资源为其他进

程所占有。

3. **非抢占**:资源不能被抢占;即,只在进程完成其任务之后,才会释放其资源。

4. **循环等待**:有一组进程$\{P_0, P_1, \cdots, P_n\}$,$P_0$ 等待的资源为 P_1 所占有,P_1 等待的资源为 P_2 所占有,P_{n-1} 等待的资源为 P_n 所占有,P_n 等待的资源为 P_0 所占有。

强调所有四个条件必须同时满足才会出现死锁。循环等待条件意味着占有并等待条件,这样四个条件并不完全独立。然而,在 8.4 节中将会看到分开考虑这些条件是有用的。

8.2.2 资源分配图

死锁问题可用称为**系统资源分配图**的有向图进行更为精确地描述。这种图有一个节点的集合 V 和一个边的集合 E 组成。节点集合 V 分成两种类型的节点 $P=\{P_1,P_2,\cdots,P_n\}$(系统活动进程的集合)和 $R=\{R_1,R_2,\cdots,R_m\}$(系统所有资源类型的集合)。

由进程 P_i 到资源类型 R_j 的有向边记为 $P_i \rightarrow R_j$;它表示为进程 P_i 已经申请了资源类型 R_j 的一个实例,并正在等待资源。由资源类型 R_j 到进程 P_i 的有向边记为 $R_j \rightarrow P_i$;它表示资源类型 R_j 的一个实例已经分配给进程 P_i。有向边 $P_i \rightarrow R_j$ 称为申请边;有向边 $R_j \rightarrow P_i$ 称为**分配边**。

在图上,用圆形表示进程 P_i,用矩形表示资源类型 R_j。由于资源类型 R_j 可能有多个实例,所以在矩形中用圆点数表示实例数。注意申请边只指向矩形 R_j,而分配边必须指向矩形内的某个圆点。

当进程 P_i 申请资源类型 R_j 的一个实例时,就在资源分配图中加入一条申请边。当该申请可以得到满足时,那么申请边就立即转换成分配边。当进程不再需要访问资源时,它就释放资源,因此就删除了分配边。

图 8.1 的资源分配图表示了如下情况。

- 集合 P,R 和 E:
  - $P=\{P_1, P_2, P_3\}$
  - $R=\{R_1,R_2,R_3,R_4\}$
  - $E=\{P_1\rightarrow R_1, P_2\rightarrow R_3, R_1\rightarrow P_2, R_2\rightarrow P_2, R_2\rightarrow P_1, R_3\rightarrow P_3\}$
- 资源实例:
  - 资源类型 R_1 有 1 个实例
  - 资源类型 R_2 有 2 个实例
  - 资源类型 R_3 有 1 个实例
  - 资源类型 R_4 有 3 个实例
- 进程状态:
  - 进程 P_1 占有资源类型 R_2 的 1 个实例,等待资源类型 R_1 的 1 个实例。
  - 进程 P_2 占有资源类型 R_1 的 1 个实例和资源类型 R_2 的 1 个实例,等待资源类型 R_3

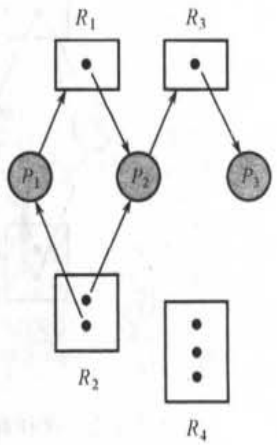


图 8.1 资源分配图

的 1 个实例。

○ 进程 P_3 等待资源类型 R_3 的 1 个实例。

根据资源分配图的定义，可以证明：如果图没有环，那么系统就没有进程死锁。如果图有环，那么可能存在死锁。

如果每个资源类型刚好有一个实例，那么有环就意味着会出现死锁。如果环涉及一组资源类型，而每个类型只有一个实例，那么就出现死锁。环所涉及的进程就死锁。在这种情况下，图中的环就是死锁存在的充分必要条件。

如果每个资源类型有多个实例，那么有环并不意味着已经出现了死锁。在这种情况下，图中的环就是死锁存在的必要条件而不是充分条件。

为了说明这个概念，下面回到图 8.1 所示资源分配图。假设进程 P_3 申请了资源类型 R_2 的一个资源。由于现在没有资源实例可用，所以就增加了有向边 $P_3 \rightarrow R_2$（图 8.2）。这时，在系统中有两个最小环：

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

进程 P_1，P_2 和 P_3 死锁。进程 P_2 等资源类型 R_3，而它又被进程 P_3 占有。另一方面，进程 P_3 等待进程 P_1 或进程 P_2 以释放资源类型 R_2。另外，进程 P_1 等待进程 P_2 释放资源 R_1。

现在考虑图 8.3 所示的资源分配图。在这个例子中，也有一个环

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

然而，没有死锁。注意进程 P_4 可能释放资源类型 R_2 的实例。这个资源可分配给进程 P_3，以打破环。

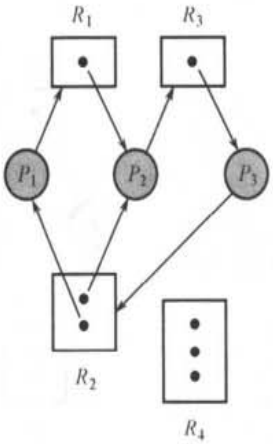


图 8.2 存在死锁的资源分配图

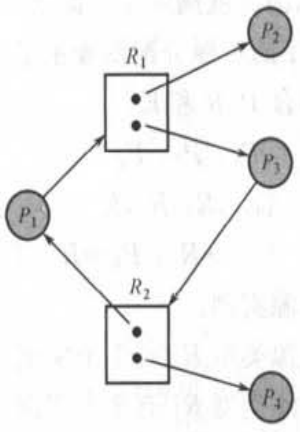


图 8.3 存在环但是没有死锁的资源分配图

总而言之，如果资源分配图没有环，那么系统就不处于死锁状态。另一方面，如果有环，那么系统可能会也可能不会处于死锁状态。在处理死锁问题时，这一点是很重要的。

8.3 死锁处理方法

从原理上来说，有三种方式可处理死锁问题：

- 可使用协议以预防或避免死锁，确保系统决不会进入死锁状态。
- 可允许系统进入死锁状态，然后检测它，并加以恢复。
- 可忽视这个问题，认为死锁不可能在系统内发生。这种方法为绝大多数操作系统如UNIX所使用。

这里将简单描述每种方法。在8.4节到8.7节中，将详细讨论每种方法。为了确保死锁不会发生，系统可以采用死锁预防或死锁避免方案。**死锁预防**是一组方法，以确保至少一个必要条件不成立(8.2.1小节)。这些方法通过限制如何申请资源的方法来预防死锁。在8.4节里，讨论这些方法。

另一方面，**死锁避免**要求操作系统事先得到有关进程申请资源和使用资源的额外信息。有了这些额外信息，可确定：对于一个申请，进程是否应等待。为了确定当前申请是满足还是延迟，系统必须考虑现有可用资源、已分配给每个进程的资源、每个进程将来申请和释放的资源。在8.5节中讨论这些方案。

如果系统不使用死锁预防和死锁避免算法，那么死锁情况可能发生。在这种情况下，系统可提供一个算法来检查系统状态以确定死锁是否发生，而另一个算法从死锁中恢复(如果死锁确实已发生)。在8.6节和8.7节中讨论这些问题。

如果系统不能确保死锁不会发生，且也不提供机制进行死锁检测和恢复，那么可能有这种情况出现：系统处于死锁而又不知道发生了什么。在这种情况下，未检查死锁会导致系统性能下降，因为资源被不能运行的进程所占有，而越来越多进程会因申请资源而进入死锁。最后，整个系统会停止工作，且需要人工重新启动。

虽然这看起来似乎不是个解决死锁问题的可行方法，但是它却为某些操作系统所使用。对于许多系统，死锁很少发生(如一年一次)；因此，与使用频繁的并且开销昂贵的死锁预防、死锁避免和死锁检测与恢复相比，这种方法会更为便宜。而且，在有的情况下，系统处于冻结状态而不是死锁状态。比如，考虑一个实时进程运行于最高优先级(或其他进程运行于非抢占调用程序之下)且不将控制返回给操作系统。因此，系统必须人工地从非死锁状态中恢复，这也是死锁恢复采用的技术。

8.4 死锁预防

如8.2.1小节所述，出现死锁有四个必要条件。只要确保至少一个必要条件不成立，就能预防死锁发生。下面通过详细讨论这四个必要条件来研究这种方法。

8.4.1 互斥

对于非共享资源，必须要有互斥条件。例如，一台打印机不能同时为多个进程所共享。另一方面，共享资源不要求互斥访问，因此不会涉及死锁。共享资源的一个很好的例子是只读文件。如果多个进程试图同时打开只读文件，那么它们能同时获得对只读文件的访问。进程决不需要等待共享资源。然而，通常不能通过否定互斥条件来预防死锁；有的资源本身是非共享的。

8.4.2 占有并等待

为了确保占有并等待条件不会在系统内出现，必须保证：当一个进程申请一个资源时，它不能占有其他资源。一种可以使用的协议是每个进程在执行前申请并获得所有资源。通过要求申请资源的系统调用在所有其他系统调用之前进行，可实现这一要求。

另外一种协议允许进程在没有资源时才可申请资源。一个进程可申请一些资源并使用它们。然而，在它申请其他更多资源之前，它必须释放其现已分配的所有资源。

为了说明这两种协议之间的差别，考虑一个进程，它将数据从磁带驱动器复制到磁盘文件，并对磁盘文件进行排序，再将结果打印到打印机上。如果所有资源必须在进程开始之前申请，那么进程必须一开始就申请磁带驱动器、磁盘文件和打印机。在其整个执行过程中，它会一直占有打印机，尽管它只在结束时才需要打印机。

第二种方法允许进程在开始时只申请磁带驱动器和磁盘文件。它将数据从磁带复制到磁盘，再释放磁带驱动器和磁盘文件。然而，进程必须再申请磁盘文件和打印机。在数据从磁盘文件复制到打印机之后，它就释放这两个资源并终止。

这两种协议有两个主要缺点。第一，**资源利用率**(resource utilization)可能比较低，因为许多资源可能已分配但是很长时间没有被使用。例如，在所给的例子中，只有在可以确认数据始终在磁盘文件上的情况下，才可以释放磁带驱动器和磁盘文件，并再次申请资源磁盘文件和打印机。否则，必须在开始之前申请所有资源。

第二，可能发生饥饿。一个进程如需要多个常用资源可能会永久等待，因为其所需要的资源中至少有一个已分配给其他进程。

8.4.3 非抢占

第三个必要条件是对已分配的资源不能抢占。为了确保这一条件不成立，可使用如下协议。如果一个进程占有资源并申请另一个不能立即分配的资源，那么其现已分配的资源都被抢占。换句话说，这些资源都被隐式地释放了。抢占资源分配到进程所等待的资源的链表上。只有当进程获得其原有资源和所申请的新资源时，进程才可重新执行。

换句话说，如果一个进程申请一些资源，那么首先检查它们是否可用。如果可用，那么

就分配它们。如果不可用，那么就检查这些资源是否已经分配给其他正在等待额外资源的进程。如果是，那么就从等待进程中抢占资源，并分配给申请进程。如果资源不可用或被其他等待进程占有，那么申请进程必须等待。当一个进程处于等待时，其部分资源可以被抢占（但要求其他进程申请它们）。一个进程要重新执行，它必须分配到其所申请的资源，并恢复其在等待时被抢占的资源。

这个协议通常应用于其状态可以保存和恢复的资源，如 CPU 寄存器和内存空间。它不能适用于其他资源如打印机和磁带驱动器。

8.4.4 循环等待

死锁的第四个也是最后一个条件是循环等待。一个确保此条件不成立的方法是对所有资源类型进行完全排序，且要求每个进程按递增顺序来申请资源。

设 $R=\{R_1,R_2,\cdots,R_m\}$ 为资源类型的集合。为每个资源类型分配一个惟一整数，以允许人们比较两个资源以确定其先后顺序。可定义一个函数 $F:R\rightarrow N$，其中 N 是自然数的集合。例如，如果资源类型 R 的集合包括磁带驱动器、磁盘驱动器和打印机，那么函数 F 可以按如下来定义：

$$F(\text{tape drive})=1,$$

$$F(\text{disk drive})=5,$$

$$F(\text{printer})=12。$$

可采用如下协议以预防死锁：每个进程只按递增顺序申请资源。即，一个进程开始可申请任何数量的资源类型 R_i 的实例。之后，当且仅当 $F(R_j)>F(R_i)$ 时，该进程可以申请资源类型 R_j 的实例。如果需要同一资源类型的多个实例，那么对它们必须一起申请。例如，对于以上给定函数，一个进程需要同时使用磁带驱动器和打印机，那么就必须先申请磁带驱动器，再申请打印机。

换句话说，要求：当一个进程申请资源类型实例 R_j 时，它必须先释放所有资源 $R_i(F(R_i)\geqslant F(R_j))$。

如果使用这两个协议，那么循环等待就不可能成立。可以通过反证法来证明这一点。假定有一个循环等待存在。设涉及循环等待的进程集合为 $\{P_0,P_1,\cdots,P_n\}$，其中 P_i 等待一个资源 R_i，而 R_i 又为进程 P_{i+1} 所占有。（对于索引采用模数代数，P_n 等待由 P_0 所占有的资源 R_n）。而且，由于进程 P_{i+1} 占有资源 R_i 而同时申请资源 R_{i+1}，所以对所有 i，必须有 $F(R_i)<F(R_{i+1})$。而这意味着 $F(R_0)<F(R_1)<\cdots<F(R_n)<F(R_0)$。根据传递规则，$F(R_0)<F(R_0)$，这显然是不可能的。因此，不可能有循环等待。

注意函数 F 应该根据系统内资源使用的正常顺序来定义。例如，由于磁带通常在打印机之前使用，所以定义 $F(\text{tape drive})<F(\text{printer})$ 较为合理。

8.5 死锁避免

在 8.4 节所讨论的死锁预防算法中通过限制资源申请的方法来预防死锁。这种限制确保四个必要条件之一不会发生，因此死锁不成立。然而，通过这种方法预防死锁的副作用是设备使用率低和系统吞吐率低。

避免死锁的另一种方法要求有关如何申请资源的附加信息。例如，对于有一台磁带驱动器和一台打印机的系统，可能知道进程 P 先申请磁带驱动器，再申请打印机，之后释放这些资源。另一方面，进程 Q 先申请打印机，再申请磁带驱动器。有了关于每个进程的申请与释放的完全顺序，可决定进程是否因申请而等待。每次申请要求系统考虑现有可用资源、现已分配给每个进程的资源和每个进程将来申请与释放的资源，以决定当前申请是否满足或必须等待从而避免死锁将来发生的可能性。

各种不同的算法在所要求的信息量和信息的类型上有所不同。最为简单和最为有用的模型要求每个进程说明可能需要的每种资源类型实例的最大需求。根据每个进程可能申请的每种资源类型实例的最大需求的事先信息，可以构造一个算法以确保系统决不会进入死锁状态。这种算法定义了**死锁避免**(deadlock-avoidance)方法。死锁避免算法动态地检测资源分配状态以确保循环等待条件不可能成立。资源分配状态是由可用资源和分配资源及进程最大需求所决定的。

8.5.1 安全状态

如果系统能按某个顺序为每个进程分配资源(不超过其最大值)并能避免死锁，那么系统状态就是安全的。更为正式地说，如果存在一个**安全序列**，那么系统处于安全状态。进程顺序$<P_1, P_2, \cdots, P_n>$，如果对于每个 P_i，P_i 申请的资源小于当前可用资源加上所有进程 P_j(其中 $j<i$)所占有的资源，那么这一顺序为安全序列。在这种情况下，进程 P_i 所需要的资源即使不能立即可用，那么 P_i 可等待直到所有 P_j 释放其资源。当它们完成时，P_i 可得到其所需要的所有资源，完成其给定任务，返回其所分配的资源并终止。当 P_i 终止时，P_{i+1} 可得到其所需要的资源，如此进行。如果没有这样的顺序存在，那么系统状态就处于不安全状态。

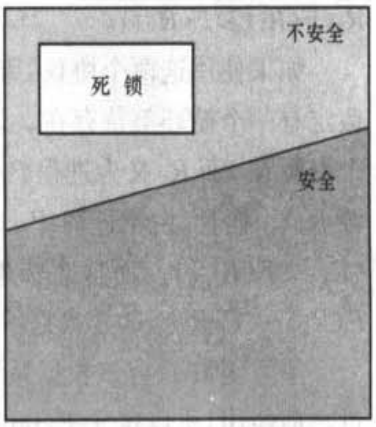


图 8.4 安全、不安全和死锁状态空间

安全状态不是死锁状态。相反，死锁状态是不安全状态。然而，不是所有不安全状态都是死锁状态(图 8.4)。不安全状态可能导致死锁。只要状态为安全，操作系统就能避免不安全(和死锁)状态。在不安全状态下，操作系统不能预防进程申请资源，这将导致死锁发生：进程行为控制了

不安全状态。

为了举例说明，考虑一个系统，有12台磁带驱动器和三个进程 P_0，P_1，P_2。进程 P_0 要求10台磁带驱动器，P_1 最多要求4台磁带驱动器，P_2 最多要求9台磁带驱动器。假定，在时刻 t_0 时，进程 P_0 占有5台磁带驱动器，进程 P_1 占有2台磁带驱动器，进程 P_2 占有2台磁带驱动器。（因此，还有3台磁带驱动器。）

	最大需求	当前需求
P_0	10	5
P_1	4	2
P_2	9	2

在时刻 t_0 时，系统处于安全状态。顺序$<P_1$，P_0，$P_2>$满足安全条件，这是因为进程 P_1 可立即得到其所有磁带驱动器并接着返回它们（系统会有5台磁带驱动器），接着进程 P_0 可得到其所有磁带驱动器并返回它们（这时系统会有10台磁带驱动器），最后进程 P_2 得到其所有磁带驱动器并返回它们（系统会有12台磁带驱动器）。

系统可以从安全状态变化到不安全状态。假定在时刻 t_1 时，进程 P_2 申请并又得到了1台磁带驱动器。系统就不再安全了。这时，只有进程 P_1 可得到其所有磁带驱动器。当其返回这些资源时，系统只有4台磁带驱动器可用。由于进程 P_0 已分配了5台磁带驱动器而其最大需求为10台磁带驱动器，所以它还需要5台磁带驱动器。因为现在不够，所以进程 P_0 必须等待。类似地，进程 P_2 还需要6台磁带驱动器，也必须等待，导致了死锁。

这时的错误在于允许进程 P_2 再获得1台磁带驱动器。如果让 P_2 等待直到其他进程之一完成并释放其资源，那么就避免了死锁。

有了安全状态的概念，可定义避免算法以确保系统不会死锁。其思想是简单地确保系统始终处于安全状态。开始，系统处于安全状态。当进程申请一个可用的资源时，系统必须确定这一资源申请是可以立即分配还是要等待。只有分配后使系统仍处于安全状态，才允许申请。

采用这种方案，如果进程申请一个现已可用的资源，那么它可能必须等待。因此，与没有采用死锁避免算法相比，这种情况下的资源使用率可能更低。

8.5.2 资源分配图算法

如果有一个资源分配系统，每种资源类型只有一个实例，那么8.2.2小节所定义的资源分配图的变形可用于死锁避免。

除了申请边和分配边外，可引入一新类型的边，称为**需求边**。需求边 $P_i \rightarrow R_j$ 表示进程 P_i 可能在将来某个时候申请资源 R_j。这种边类似于同一方向的申请边，但是用虚线表示。当进程 P_i 申请资源 R_j 时，需求边 $P_i \rightarrow R_j$ 变成了申请边。类似地，当进程 P_i 释放 R_j 时，分配边 $R_j \rightarrow P_i$ 变成了需求边。注意系统必须事先要求资源。即，当进程 P_i 开始执行时，所有

需求边必须先处于资源分配图。可放松这个条件，以允许只有在进程 P_i 的所有相关的边都为需求边时才将需求边 $P_i \rightarrow R_j$ 增加到图中。

假设进程 P_i 申请资源 R_j。只有在将申请边 $P_i \rightarrow R_j$ 变成分配边 $R_j \rightarrow P_i$ 而不会导致资源分配图形成环时，才允许申请。注意，通过采用循环检测算法，检测安全性。检测图中是否有环的算法需要 n^2 级的操作，其中 n 是系统的进程数量。

如果没有环存在，那么资源分配会使得系统处于安全状态。如果有环存在，那么分配会导致系统处于不安全状态。因此，进程 P_i 必须等待其资源申请。

为了说明这个算法，考虑图 8.5 所示资源分配图。假设进程 P_2 申请资源 R_2。虽然 R_2 现在可用，但是不能将它分配给 P_2，因为这会创建一个环(如图 8.6)。循环表示系统处于不安全状态。如果 P_1 申请 R_2 且 P_2 申请 R_1，那么死锁会发生。

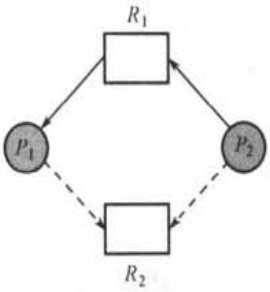


图 8.5 死锁避免的资源分配图

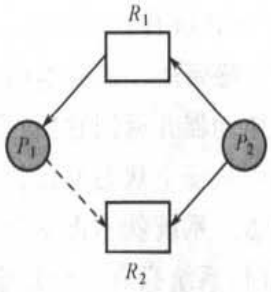


图 8.6 资源分配图的不安全状态

8.5.3 银行家算法

对于每种资源类型有多个实例的资源分配系统，资源分配图算法就不适用了。下面所要描述的死锁避免算法适用于这种系统，但是其效率要比资源分配图方案差。这一算法通常称为银行家算法。该算法如此命名是因为这一算法可用于银行系统，以确保银行决不会分配其现金以致使它不能满足其所有客户的需要。当新进程进入系统时，它必须说明其可能需要的每种资源类型的实例的最大数量。这一数量不可能超过系统资源的总量。当用户申请一组资源时，系统必须确定这些资源的分配是否仍会使系统处于安全状态。如果会，就可分配资源；否则，进程必须等待直到某个其他进程释放足够资源为止。

为了实现银行家算法，必须要有若干数据结构。这些数据结构对资源分配系统的状态进行了编码。设 n 为系统进程的个数，m 为资源类型的种类。需要如下数据结构：

- **Available**：长度为 m 的向量表示每种资源的现有实例的数量。如果 $Available[j]=k$，那么资源类型 R_j 现有 k 个实例。
- **Max**：$n \times m$ 矩阵定义每个进程的最大需求。如果 $Max[i,j]=k$，那么进程 P_i 最多可申请 k 个资源类型 R_j 的实例。
- **Allocation**：$n \times m$ 矩阵定义每个进程现在所分配的各种资源类型的实例数量。如果

$Allocation[i,j]=k$，那么进程 P_i 现在已分配了 k 个资源类型 R_j 的实例。

• **Need**：$n\times m$ 矩阵表示每个进程还需要的剩余的资源。如果 $Need[i,j]=k$，那么进程 P_i 还可能申请 k 个资源类型 R_j 的实例。注意 $Need[i,j]=Max[i,j]-Allocation[i,j]$。这些数据结构的大小和值会随着时间而改变。

为了简化银行家算法的描述，采用一些记号。设 X 和 Y 是长度为 n 的向量。可以说：$X\leqslant Y$ 当且仅当对所有 $i=1,2,\cdots,n$。$X[i]\leqslant Y[i]$。例如，如果 $X=(1,7,3,2)$ 而 $Y=(0,3,2,1)$，那么 $Y\leqslant X$。如果 $Y\leqslant X$ 且 $Y\neq X$，那么 $Y<X$。

可将矩阵 $Allocation$ 和 $Need$ 的每行作为向量，并分别用 $Allocation_i$ 和 $Need_i$ 来表示。向量 $Allocation_i$ 表示分配给进程 P_i 的资源；向量 $Need_i$ 表示进程为完成其任务可能仍然需要申请的额外资源。

1. 安全性算法

确定计算机系统是否处于安全状态的算法分为如下几步：

1. 设 $Work$ 和 $Finish$ 分别是长度为 m 和 n 的向量。按如下方式进行初始化，$Work:=Available$ 且对于 $i=1,2,\cdots,n$，$Finish[i]:=false$。

2. 查找这样的 i 使其满足

   a. $Finish[i]=false$

   b. $Need_i\leqslant Work$

如果没有这样的 i 存在，那么就转到第 4 步。

3. $Work:=Work+Allocation_i$

   $Finish[i]:=true$

返回到第 2 步。

4. 如果对所有 i，$Finish[i]=true$，那么系统处于安全状态。

这个算法可能需要 $m\times n^2$ 数量级的操作以确定状态是否安全。

2. 资源请求算法

设 $Request_i$ 为进程 P_i 的请求向量。如果 $Request_i[j]=k$，那么进程 P_i 需要资源类型 R_j 的实例数量为 k。当进程 P_i 做出资源请求时，会采取如下动作。

1. 如果 $Request_i\leqslant Need_i$，那么转到第 2 步。否则，产生出错条件，这是因为进程 P_i 已超过了其请求。

2. 如果 $Request_i\leqslant Available$，那么转到第 3 步。否则，$P_i$ 必须等待，这是因为没有可用资源。

3. 假定系统可以分配给进程 P_i 所请求的资源，并按如下方式修改状态：

   $Available:=Available-Request_i$；

   $Allocation_i:=Allocation_i+Request_i$；

   $Need_i:=Need_i-Request_i$；

如果所产生的资源分配状态是安全的，那么交易完成且进程 P_i 可分配到其所需要资源。然而，如果新状态不安全，那么进程 P_i 必须等待 *Request*$_i$ 并恢复到原来资源分配状态。

3. 说明性实例

考虑这样一个系统，有 5 个进程 P_0 到 P_4，3 种资源类型 A、B、C。资源类型 A 有 10 个实例，资源类型 B 有 5 个实例，资源类型 C 有 7 个实例。假定在时刻 T_0，系统状态如下：

	Allocation			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

矩阵 *Need* 的内容定义成 *Max*-*Allocation*：

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

可以认为系统现在处于安全状态。事实上，顺序$<P_1, P_3, P_4, P_2, P_0>$满足安全条件。现在假定进程 P_1 再请求一个资源类型 A 和 2 个资源类型 C，这样 *Request*$_1$ = (1, 0, 2)。为了确定这个请求是否立即允许，首先检测 *Request*$_1 \leqslant$ *Available*(即，(1,0,2)$\leqslant$ (3,3, 2))，其值为真。接着假定这个请求可满足，会有如下新状态：

	Allocation			*Need*			*Available*		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

我们必须确定这个状态是否安全。为此，执行安全算法，并找到顺序$<P_1, P_3, P_4, P_2, P_0>$满足安全要求。因此，可以立即允许进程 P_1 的这个请求。

然而，你可以发现当系统处于这一状态时 P_1 的请求(3,3,0)是不能允许的，因为没有这么多资源可用。P_0 的请求(0,2,0)不能允许：虽然有资源可用，但是这会导致系统处于不安全状态。

8.6 死锁检测

如果一个系统既不采用死锁预防算法也不采用死锁避免算法，那么可能会出现死锁。在这种环境下，系统应提供：

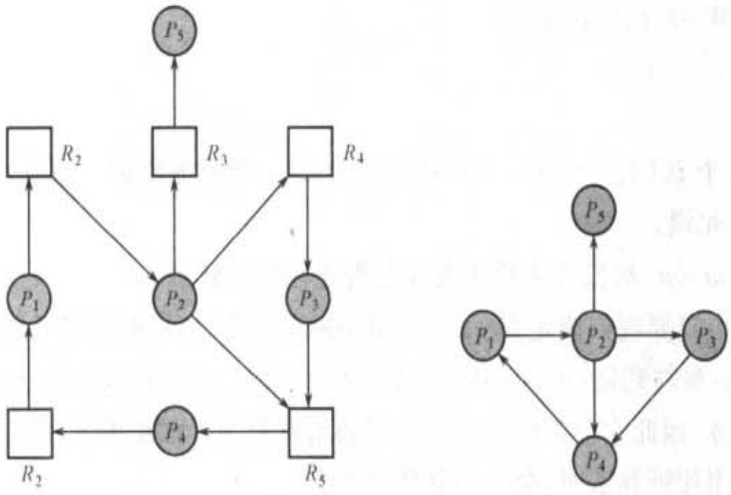
- 一个用来检查系统状态从而确定是否出现了死锁的算法。
- 一个用来从死锁状态中恢复的算法。

在以下讨论中，将针对每种资源类型只有单个实例和每种资源类型可有多个实例这两种情况分别研究这两个算法。不过，现在需要注意到检测恢复方案会有额外开销，这些不但包括维护所需信息和执行检测算法的运行开销，而且也包括死锁恢复所引起的损失。

8.6.1 每种资源类型只有单个实例

如果所有资源类型只有单个实例，那么可以定义这样一个死锁检测算法，该算法使用了资源分配图的一个变种，称为等待图。从资源分配图中，删除所有资源类型节点，合并适当边，就可以得到等待图。

更确切地说，等待图中的由 P_i 到 P_j 的边意味着进程 P_i 等待进程 P_j 释放一个 P_i 所需的资源。等待图有一条由 $P_i \rightarrow P_j$ 的边当且仅当相应的资源分配图中包含两条边 $P_i \rightarrow R_q$ 和 $R_q \rightarrow P_j$，其中 R_q 为资源。例如，图 8.7 显示了资源分配图和相应等待图。



(a) 资源分配图　　(b) 对应的等待图

图 8.7　资源分配图和对应的等待图

与以前一样，系统死锁当且仅当等待图中有一个环。为了检测死锁，系统需要维护等待图，并周期性地调用在图中进行搜索的算法。

从图中检测环的算法需要 n^2 级别操作，其中 n 为图中的节点数。

8.6.2 每种资源类型的多个实例

等待图方案并不适用于每种资源类型可有多个实例的资源分配系统。下面所描述的死锁检测算法适用于这样的系统。该算法使用了一些随时间而变化的数据结构，与银行家算法相似。

- **Available**：长度为 m 的矢量，表示各种资源的可用实例。
- **Allocation**：$n \times m$ 矩阵，表示当前各进程的资源分配情况。
- **Request**：$n \times m$ 矩阵，表示当前各进程的资源请求情况。如果 $Request[i,j]=k$，那么 P_i 现在需要 k 个资源 R_j。

两矢量间≤关系与 8.5.3 小节所定义的一样。为了简化起见，将 *Allocation* 和 *Request* 的行作为矢量，且分别称为 $Allocation_i$ 和 $Request_i$。这里所描述的检测算法为需要完成的所有进程研究各种可能的分配序列。请将本算法与银行家算法做一比较。

1. 设 *Work* 和 *Finish* 分别是长度为 m 和 n 的矢量。初始化 $Work:=Available$。对 $i=1,2,\cdots,n$，如果 $Allocation_i$ 不为 0，则 $Finish[i]:=false$；否则，$Finish[i]:=true$。

2. 找这样的下标，以便同时使

$$Finish[i]=false$$

$$Request_i \leqslant Work$$

如果没有这样的 i，则转到第四步。

3. $Work:=Work+Allocation_i$

$$Finish[i]:=true$$

转到第二步。

4. 如果对某个 $i(1 \leqslant i \leqslant n)$，$Finish[i]=false$，则系统死锁。而且，如果 $Finish[i]=false$，则进程 P_i 死锁。

该算法需要 $m \times n^2$ 级操作来检测系统是否处在死锁状态。

你可能不明白只要我们确定 $Request_i \leqslant Work$（第二步 2b），就收回了进程 P_i 的资源。大家知道 P_i 现在不参与死锁（因 $Request_i \leqslant Work$）。因此，可以乐观地认为 P_i 不再需要更多资源以完成其任务；因此它会返回其现已分配的所有资源。如果假定不正确，那么死锁会稍后发生。下次调用死锁算法时，会检测到死锁状态。

为了举例说明这一算法，考虑这样一个系统，它有 5 个进程 P_0 到 P_4 和 3 个资源类型 A,B,C。资源类型 A 有 7 个实例，资源类型 B 有 2 个实例，资源类型 C 有 6 个实例。假定在时刻 T_0 时，有如下资源分配状态：

	Allocation			*Max*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

可以认为系统现不处于死锁状态。事实上，如果执行检测算法，会找到这样一个序列$<P_0, P_2, P_3, P_1, P_4>$会导致对所有 i，$Finish[i]$ = true。

现在假定进程 P_2 又请求了资源类型 C 的一个实例。这样，*Request* 矩阵修改成如下形式。

	Reqest		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

可以认为现在系统是死锁。虽然可收回进程 P_0 所占有的资源，但是现有资源并不足以满足其他进程的请求。因此，进程 P_1，P_2，P_3 和 P_4 会一起死锁。

8.6.3 应用检测算法

应何时调用检测算法。答案取决于两个因素：

- 死锁可能发生的频率是多少？
- 当死锁发生时，有多少进程会受影响？

如果死锁经常发生，那么就应经常调用检测算法。分配给死锁进程的资源会一直空着，直到死锁被打破。另外，参与死锁循环的进程数量可能会不断增加。

只有当某个进程提出请求且得不到满足时，才会出现死锁。这一请求可能是完成等待进程链的最后请求。在极端情况下，每次请求分配不能立即允许时，就调用死锁检测算法。在这种情况下，不仅能确定哪些进程死锁，而且也能确定哪个特定进程造成了死锁。（而实际上，每个死锁进程都是资源图的环的一个链节，因此，所有进程一起造成了死锁。）如果有许多不同资源类型，那么一个请求可能造成资源图的许多环，每个环由最近请求所完成且由可标识的进程造成。

当然，对于每个请求都调用死锁检测算法会引起相当的计算开销。另一个不太昂贵的

方法只是在一个不频繁的时间间隔里调用检测算法，如每小时一次，或当 CPU 使用率低于 40% 时。（死锁最终会使系统性能下降，并造成 CPU 使用率下降。）如果在不定的时间点调用检测算法，那么资源图会有许多环。通常不能确定死锁进程中是哪些引起了死锁。

8.7 死锁恢复

当死锁检测算法确定死锁已存在，那么可以采取多种措施。一种措施是通知操作员死锁已发生，以便操作人员人工处理死锁。另一种措施是让系统从死锁状态中自动恢复过来。打破死锁有两个方法。一个方法是简单地终止一个或多个进程以打破循环等待。另一个方法是从一个或多个死锁进程那里抢占一个或多个资源。

8.7.1 进程终止

有两个方法通过终止进程以取消死锁。不管采用哪个方法，系统都会收回分配给被终止进程的所有资源。

- **终止所有死锁进程**：这种方法显然终止了死锁循环，但是其代价也大；这些进程可能已计算了很长时间，这些部分计算结果必须放弃，以后可能还要重新计算。
- **一次只终止一个进程直到取消死锁循环为止**：这种方法的开销会相当大，这是因为每次终止一个进程，都必须调用死锁检测算法以确定进程是否仍处于死锁。

终止一个进程并不简单。如果进程正在更新文件，那么终止它会使文件处于不一致状态。类似地，如果进程正在打印文件，那么系统必须将打印机重新设置到正确状态，以便打印下一个文件。

如果采用了部分终止，那么对于给定死锁进程，必须确定终止哪个进程或哪些进程可以打破死锁。这个确定类似于 CPU 调度问题，是个策略选择。该问题基本上是个经济问题；应该终止代价最小的进程。然而“代价最小”并不精确。许多因素都影响着应选择哪个进程，包括：

1. 进程的优先级是什么？
2. 进程已计算了多久，进程在完成其指定任务之前还需要多久？
3. 进程使用了多少什么类型的资源（例如，这些资源是否容易抢占）？
4. 进程需要多少资源以完成？
5. 多少进程需要被终止？
6. 进程是交互的还是批处理的？

8.7.2 资源抢占

通过抢占资源以取消死锁，逐步从进程中抢占资源给其他进程使用，直到死锁环被打破

为止。

如果要求使用抢占来处理死锁，那么有三个问题需要处理：

1. **选择一个牺牲品**：抢占哪些资源和哪个进程？与进程取消一样，必须确定抢占顺序以最小化代价。代价因素包括许多参数如死锁进程所拥有的资源数量，死锁进程到现在为止在其执行过程中所消耗的时间。

2. **回滚**：如果从一个进程那里抢占一个资源，那么应对该进程做些什么安排？显然，该进程不能正常执行；它缺少其所需要的资源。必须将进程回滚到某个安全状态，以便从该状态重启进程。

通常确定一个安全状态并不容易，所以最简单的方法是完全回滚：终止进程并重新执行。然而，更为有效的方法是只将进程回滚到足够打破死锁。另一方面，这种方法要求系统维护有关运行进程状态的更多信息。

3. **饥饿**：如何确保饥饿不会发生？即，如何保证资源不会总是从同一个进程中被抢占。

如果一个系统是基于代价来选择牺牲进程，那么同一进程可能总是被选为牺牲品。结果，这个进程永远不能完成其指定任务，任何实际系统都需要处理这种饥饿情况。显然，必须确保一个进程只能有限次地被选择为牺牲品。最为常用的方法是在代价因素中加上回滚次数。

8.8 小 结

如果两个或更多的进程永久等待某个事件而该事件只能由这些等待进程的某一个引起，那么会出现死锁状态。从原理上来说，有三种方法可以处理死锁：

- 使用一些协议来预防或避免死锁，确保系统永远都不会进到死锁状态。
- 允许系统进入死锁状态，检测死锁，并恢复。
- 忽略所有的问题，并假设系统中永远都不会出现死锁。这种方法被绝大多数的系统所采用，包括 UNIX。

当且仅当系统内的四个必要条件同时成立时（互斥、占有并等待、非抢占、循环等待），才会发生死锁。为了预防死锁，要确保这四个必要条件中的一个不成立。

死锁避免算法要比预防算法的要求低些，只要事先了解进程使用资源的情况即可。例如，银行家算法需要知道每个进程所请求的每种资源的最大数量。采用这种信息，可采用死锁避免算法。

如果是不采用协议以确保死锁不会发生，那么就必须使用检测并恢复方案。必须调用检测算法以确定是否出现了死锁。如果检测到死锁，那么系统必须通过终止某些死锁进程，或通过抢占某些死锁进程的资源从死锁中恢复。

如果系统主要根据代价因素以选择牺牲进程进行回滚，那么可能会出现饥饿，导致所选

择进程永远不能完成其指定任务。

习 题 八

8.1 列出三个与计算机系统环境相关的死锁的例子。

8.2 死锁是否可能只涉及一个进程？为什么？

8.3 人们认为适当的假脱机可以消除死锁。当然，它可以消除读卡机、绘图仪、打印机等上面的竞争。甚至于磁带机都可以假脱机(称为分段执行)，假脱机释放了像 CPU 时间、内存和磁盘空间等这样的资源。一个死锁是否可能占用这些资源？如果可能，这样的死锁是怎样发生的？如果不可能，为什么？哪一种死锁解决方法对于消除这些死锁是最好的(如果可能)？或违反了什么条件(如果不可能)？

8.4 假设有如图 8.8 中所描述的交通死锁。

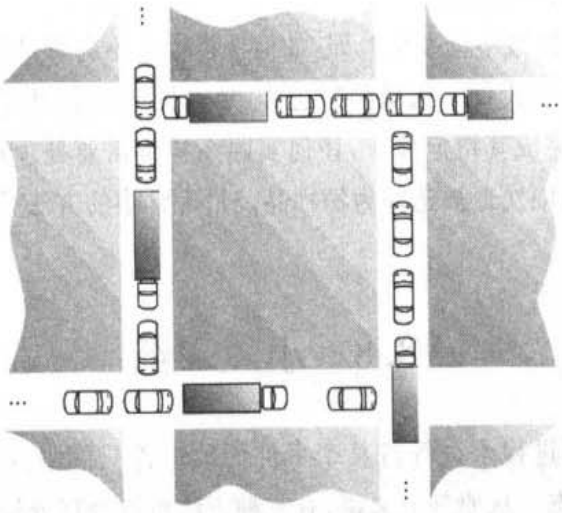


图 8.8 习题 8.4 交通死锁

a. 证明这个例子中实际上包括了死锁发生的四个必要条件。

b. 给出一个简单的规则用来在这个系统中避免死锁。

8.5 假设一个系统处在不安全状态，证明系统里的进程是可能完成它们的执行而不进入死锁状态的。

8.6 在一个真实的计算机系统中，可用的资源和进程命令对资源的要求都不会持续很久(几个月)。资源会损坏或被替换，新的进程会进入和离开系统，新的资源会被购买和添加到系统中。如果用银行家算法控制死锁，下面哪些变化是安全的(不会导致可能的死锁)，并且是在什么情况下发生？

a. 增加 *可用资源*(新的资源被添加到系统)

b. 减少 *可用资源*(资源被从系统中永久性地移出)

c. 增加一个进程的 *Max*(进程需要更多的资源，超过所允许给予的资源)

d. 减少一个进程的 *Max*(进程不再需要那么多资源)

e. 增加进程的数量

f. 减少进程的数量

8.7 证明 8.5.3 小节中的安全算法需要 $m \times n^2$ 数量级操作。

8.8 假设系统中有4个相同类型的资源被3个进程共享。每个进程最多需要2个资源。证明这个系统不会死锁。

8.9 假设一个系统有 m 个资源被 n 个进程共享，进程每次只请求和释放一个资源。证明只要系统符合下面两个条件，就不会发生死锁：

a. 每个进程需要资源的最大值在1到 m 之间。

b. 所有进程需要资源的最大值的和小于 $m+n$。

8.10 假设一个没有死锁预防与避免措施的计算机系统，每月要运行5 000个作业。死锁每月大概出现两次，这时操作必须停止并且大约每个死锁有10个任务要重新运行。每个作业价值2美元(CPU时间)，并且停止的作业往往是在执行到一半时中止。

一个系统程序员估计如在这个系统中加入一个死锁避免算法(像银行家算法)，每个作业增加了大约10%的平均执行时间。现在机器有30%的空闲时间，每月仍然运行5 000个作业，尽管周转时间平均增长了20%。

a. 问支持加入死锁避免算法的论据有哪些？

b. 问反对加入死锁避免算法的论据有哪些？

8.11 将数组的维数减少到1，就可以容易地从一般银行家算法中得到只针对一种资源类型的银行家算法。给出例子说明多资源类型银行家算法方案不可以通过对每种资源类型单独运用单资源类型银行家算法的方法来实现。

8.12 系统能否检测出它的某些进程中正处于饥饿状态？如果你回答"是"，给出理由。回答"不"，解释系统怎样才能处理饥饿问题。

8.13 考虑下面的一个系统在某一时刻的状态：

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

使用银行家算法回答下面问题：

a. *Need* 矩阵的内容是怎样的？

b. 系统是否处于安全状态？

c. 如果从进程 P_1 发来一个请求(0,4,2,0)，这个请求能否立刻被满足？

8.14 考虑下面的资源分配策略。在任何时刻都可以进行资源的请求和释放。如果一个资源的请求因为没有可用的资源而未被满足，那么检查所有等待资源的被阻塞的进程。如果它们有想要的资源，则把资源从它们那里拿出以分配给请求资源的进程。等待进程的资源需求向量增长来包括被取走的资源。

例如，假设一个有三种资源类型的系统，初始的 *可用* 向量为(4,2,2)。如果进程 P_0 要求(2,2,1)，它获得这些资源。如果 P_1 要求(1,0,1)，P_1 获得这些资源。那么，如果 P_0 再要求(0,0,1)，P_0 被阻塞(没有可用的资源)。如果 P_2 现在要求(2,0,0)，它获得可用的一个(1,0,0)和一个已经分配给 P_0 的(P_0 已经被阻塞)。P_0 的已分配向量减少为(1,2,1)，需求向量增加为(1,0,1)。

a. 会出现死锁吗？如果会，举个例子。如果不会，哪个必要条件不可能发生？

b. 不确定阻塞会出现吗？

8.15 假设你已经实现了死锁避免的安全算法并被要求实现死锁检测算法。你能否简单地通过使用安全算法的代码，重新定义 $Max_i = Waiting_i + Allocation_i$，在这里 $Waiting_i$ 是个描述进程 i 等待的资源的向量，$Allocation_i$ 的定义和 8.5 节中一样？解释你的答案。

推荐读物

Dijkstra[1965a]是死锁领域最早和最有影响的贡献者之一。Holt[1972]是第一个像此章内容一样将死锁问题用图论模型进行形式化的人。Holt[1972]论述了饥饿。Hyman[1985]给出了美国堪萨斯州议会死锁的例子。

Havender[1968]曾设计了 IBM OS/360 系统中的资源排序方案，给出了多种死锁预防算法。

Dijkstra[1965a]提出了对单种资源类型的银行家死锁避免算法。Habermann[1969]将此算法扩展到多种资源类型。习题 8.8 和习题 8.9 来自 Holt[1971]。

Coffman 等[1971]给出了 8.6.2 小节中描述的单种资源类型多实例的死锁检测算法。

Bach[1987]叙述了在传统的 UNIX 内核中有多少种死锁处理算法。

第三部分　存储管理

计算机系统的主要用途是执行程序。在执行时，这些程序及其所访问的数据必须在内存里（至少部分是如此）。

为改善 CPU 的使用率和对用户的响应速度，计算机必须在内存中保留多个进程。内存管理方案有很多，以便适应各种不同的需求，每个算法的有效性与特定情况有关。对系统内存管理方案的选择依赖于很多因素，特别是系统的*硬件*设计。每个算法都需要有自己的硬件支持。

由于内存通常太小不足以永久地容纳所有数据和程序，因此计算机系统必须提供次级存储以支持内存。现代计算机系统采用硬盘作为信息（程序和数据）的主要在线存储媒介。文件系统为在线存储和访问驻留在硬盘上的数据提供了一种机制。文件是一组由创建者定义的相关信息的集合。文件通过操作系统映射到物理设备。文件通常组织成目录以方便使用。

第九章　内存管理

在第六章，讨论了CPU如何被一组进程所共享。正是由于CPU调度的结果，才能提高CPU的使用率和计算机对用户的响应速度。但是，为了实现这一性能改进，必须将多个进程保存在内存中；也就是说，必须*共享*内存。

在本章，将讨论各种内存管理的方法。内存管理算法有很多，从简单的裸机方法，到分页和分段策略。各种方法都有其优点和缺点。为特定系统选择内存管理方法依赖于很多因素，特别是系统的*硬件设计*。正如大家将会看到的，尽管现在的设计已经将硬件和操作系统紧密地结合在一起，但是许多算法仍然需要硬件的支持。

9.1　背　　景

正如第一章所述，内存是现代计算机运行的中心。内存由很大一组字或字节所组成，每个字或字节都有它们自己的地址。CPU根据程序计数器PC的值从内存中提取指令。这些指令可能会引起进一步的对特定内存地址的读取和写入。

例如，一个典型指令执行周期，首先从内存中读取指令。接着该指令被解码，且可能需要从内存中读取操作数。在指令对操作数执行后，其结果可能被存回到内存。内存单元只看到地址流，而并不知道这些地址是如何产生的（由指令计数器、索引、间接寻址、实地址等）或它们是什么地址（指令或数据）。相应地，可以忽略内存地址是*如何*由程序产生的。这里只对由运行中的程序所产生的内存地址感兴趣。

9.1.1　地址捆绑

通常，程序以二进制可执行文件形式存储在磁盘上。为了执行，程序应被调入内存并放在进程内。根据所使用的内存管理方案，进程在执行时可以在磁盘和内存之间移动。在磁盘上等待调入内存以便执行的进程形成了**输入队列**。

通常的步骤是从输入队列中选一个进程并装入内存。进程在执行时，会访问内存中的指令和数据。最后，进程终止，其地址空间将被释放。

许多系统允许用户进程放在物理内存的任意位置。因此，虽然计算机的地址空间从00000开始，但用户进程的开始地址不必也是00000。这种组织方式会影响用户程序能够使用的地址空间。在绝大多数情况下，用户程序在执行前，需要经过好几个步骤，其中有的是

可选的(参见图 9.1)。在这些步骤中,地址可能有不同的表示形式。源程序中的地址通常是符号表示(如 count)。编译器通常将这些符号地址捆绑在可重定位的地址(如"从本模块开始的第 14 字节")。链接程序或加载程序再将这些可重定位的地址捆绑成绝对地址(如 74014)。每次捆绑都是从一个地址空间到另一个地址空间的映射。

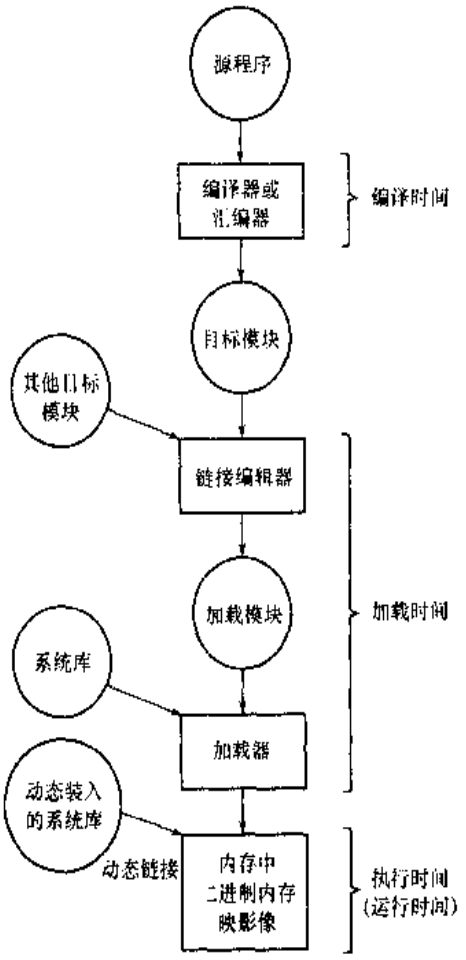


图 9.1 一个用户程序的多步骤处理

通常,将指令与数据捆绑到内存地址可以在以下步骤的任何一步中执行:

- **编译时**:如果在编译时就知道进程将在内存中的驻留地址,那么就可以生成**绝对代码**。例如,如果事先就知道用户进程驻留在内存地址 R 处,那么所生成的编译代码就可以从该位置开始并向后扩展。如果将来开始地址发生变化,那么就必须重新编译代码。MS-DOS 的.COM 格式程序就是在编译时捆绑成绝对代码的。

- **加载时**:如果在编译时并不知道进程将驻留在何处,那么编译器就必须生成**可重定位代码**(relocatable code)。对这种情况,最后捆绑会延迟到加载时才进行。如果开始地址发生变化,只需重新加载用户代码以引入改变值。

- **执行时**:如果进程在执行时可以从一个内存段移到另一个内存段,那么捆绑必须延迟到执行时才进行。正如 9.1.2 小节所述,采用这种方案需要特定硬件才行。绝大多数计算机采用这种方法。

本章的主要部分将描述如何在计算机系统中有效地实现这些捆绑,并将讨论合适的硬件支持。

9.1.2 逻辑地址空间与物理地址空间

CPU 所生成的地址通常称为逻辑地址,而内存单元所看到的地址(即,加载到内存地址寄存器中的地址)通常称为**物理地址**(physical address)。

编译时和加载时的地址捆绑生成相同的逻辑地址和物理地址。但是,执行时的地址捆绑方案导致不同的逻辑地址和物理地址。对这种情况,通常称逻辑地址为**虚拟地址**(virtual address)。在本书中,对逻辑地址和虚拟地址不加区别。由程序所生成的所有逻辑地址的集合称为**逻辑地址空间**;与这些逻辑地址相对应的所有物理地址的集合称为**物理地址空间**。

因此，对于执行时地址捆绑方案，逻辑地址空间与物理地址空间是不同的。

运行时从虚拟地址到物理地址的映射是由称为**内存管理单元**(memory-management unit，MMU)的硬件设备来完成的。正如将在 9.3 节、9.4 节、9.5 节和 9.6 节所要讨论的，有许多可选择的方法能够完成这种映射。这里，用一个简单的 MMU 方案来实现这种映射，这是 2.5.3 小节所描述基址寄存器方案的推广。

如图 9.2 所示，这种方法需要硬件支持，这与 2.4 节所讨论的硬件配置相似。基址寄存器这时称为**重定位寄存器**。用户进程所生成的地址在送交内存之前，都将加上重定位寄存器的值。例如，如果基地址为 14000，那么用户对位置 0 的访问将动态地重定位为位置 14000；对地址 346 的访问将映射为位置 14346。运行于 Intel 80x86 系列 CPU 的 MS-DOS 操作系统在加载和运行进程时，可使用四个重定位地址。

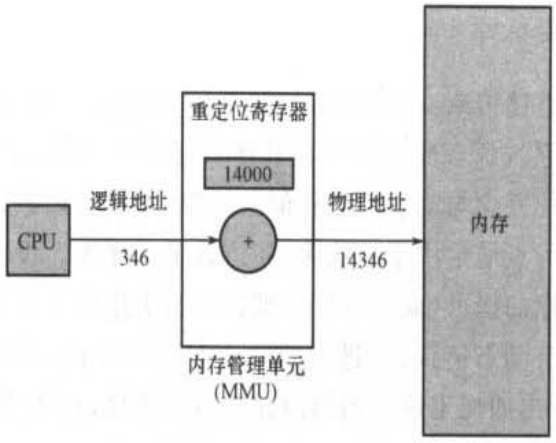


图 9.2　使用重定位寄存器的动态重定位

用户程序决不会看到*真正*的物理地址。程序可以创建一个指向位置 346 的指针，将它保存在内存中，使用它，将它与其他地址进行比较，所有这些都是通过 346 这样一个数来使用。只有当它作为内存地址时(例如，在间接加载和存储时)，它才进行相对于基地址的重定位。用户程序处理*逻辑*地址。内存映射硬件将逻辑地址转变为物理地址。这种运行时捆绑已在 9.1.1 小节中讨论过。所引用的内存地址只有在引用时才最后定位。

现在有两种不同的地址：逻辑地址(范围为 0 到 max)和物理地址(范围为 $R+0$ 到 $R+$ max，其中 R 为基地址)。用户只生成逻辑地址，且认为进程的地址空间为 0 到 max。用户提供逻辑地址，这些地址在使用前必须映射到物理地址。

*逻辑地址空间*捆绑到单独的一个*物理地址空间*这一概念对内存的适当管理至关重要。

9.1.3　动态加载

迄今为止所讨论的是，一个进程的整个程序和数据必须处于内存中，以便执行。进程的大小受内存大小的限制。为了获得更好的内存空间使用率，可以使用**动态加载**(dynamic

loading)。采用动态加载时,一个子程序只有在调用时才被加载。所有子程序都以可重定位的形式保存在磁盘上。主程序装入内存并执行。当一个子程序需要调用另一个子程序时,调用子程序首先检查另一个子程序是否已加载。如果没有,可重定位的链接程序将被用来加载所需要的子程序,并更新程序的地址表以反映这一变化。接着,控制传递给新加载的子程序。

动态加载的优点是不用的子程序决不会被装入内存。如果大多数代码需要用来处理异常情况,如错误处理,那么这种方法特别有用。对这种情况,虽然总体上程序比较大,但是所使用的部分(即加载的部分)可能小很多。

动态加载不需要操作系统提供特别的支持。利用这种方法来设计程序主要是用户的责任。不过,操作系统可以帮助程序员,如提供子程序库以实现动态加载。

9.1.4 动态链接与共享库

图 9.1 也显示了**动态链接库**(dynamically linked library)。有的操作系统只支持**静态链接**(static linking),因而系统语言库的处理与其他目标模块一样,由加载程序合并到二进制程序映象中。动态链接的概念与动态加载相似。这里,不是将加载延迟到运行时,而是将链接延迟到运行时。这一特点通常用于系统库,如语言子程序库。没有这一点,系统上的所有程序都需要一份其语言库的拷贝(或至少那些被程序所引用的子程序)。这一要求浪费磁盘空间和内存空间。如果有动态链接,二进制映象中对每个库程序的引用都有一个存根。存根是一小段代码,用来指出如何定位适当的内存驻留库程序,或如果该程序不在内存时应如何装入库。

当存根执行时,它首先检查所需子程序是否已在内存中。如果不在,就将子程序装入内存。不管如何,存根会用子程序地址来替换自己,并开始执行子程序。因此,下次再执行该子程序代码时,就可以直接进行,而不会因动态链接而产生任何开销。采用这种方案,使用语言库的所有进程只需要一个库代码拷贝就可以了。

动态链接也可用于库更新(如改进版)。一个库可能被新的版本所替代,且使用该库的所有程序会自动使用新的版本。没有动态链接,所有这些程序必须重新链接以便访问新的库。为了不使程序错用新的、不兼容版本的库,程序和库可以包括版本信息。多个版本的库可以都装入内存,程序将通过版本信息来确定使用哪个库拷贝。次要改动保持了同样的版本号,而重要改动增加版本号。因此,只有用新库编译的程序才会受其中的不兼容变化影响。在新库安装之前所链接的其他程序可以继续使用老库。这种系统也称为**共享库**。

与动态加载不一样,动态链接通常需要操作系统的帮助。如果内存中进程是彼此保护的,那么只有操作系统才可以检查所需子程序是否在其他进程内存空间内,或是允许多个进程访问同一内存地址。在 9.4.5 小节讨论分页时,将进一步讨论这一点。

9.1.5 覆盖

为了能让进程比它所分配到的内存空间大，可以使用**覆盖**(overlay)。覆盖的思想是在任何时侯只在内存中保留所需的指令和数据。当需要其他指令时，它们会装入到刚刚不再需要的指令所占用的内存空间内。

作为一个例子，考虑一个 two-pass 汇编程序。在第一遍时，建立符号表；在第二遍时，生成机器码。可以将汇编程序分成第一遍代码、第二遍代码、符号表、由第一遍代码和第二遍代码所共同使用的公共支持程序如下：

第一遍代码	70 KB
第二遍代码	80 KB
符号表	20 KB
公共程序	30 KB

为了将所有代码一次性地装入内存，需要 200 KB 内存空间。如果只有 150 KB，那么就不能运行该进程。不过，注意第一遍代码和第二遍代码不必同时位于内存。因此可以定义两个覆盖：覆盖 A 是符号表、公共程序和第一遍代码；覆盖 B 是符号表、公共程序和第二遍代码。

可以增加覆盖驱动程序(10 KB)，并从内存中的覆盖 A 开始。当完成第一遍时，转到覆盖驱动程序以读入覆盖 B 并重写覆盖 A，接着转到第二遍。覆盖 A 需要 120 KB，而覆盖 B 需要 130 KB(图 9.3)。现在能在 150 KB 内存中执行汇编程序。由于在执行开始前所需要装入的数据较少，因而装入得比较快。不过，由于读入覆盖 B 以替代覆盖 A 的额外 I/O 开销，运行稍慢。

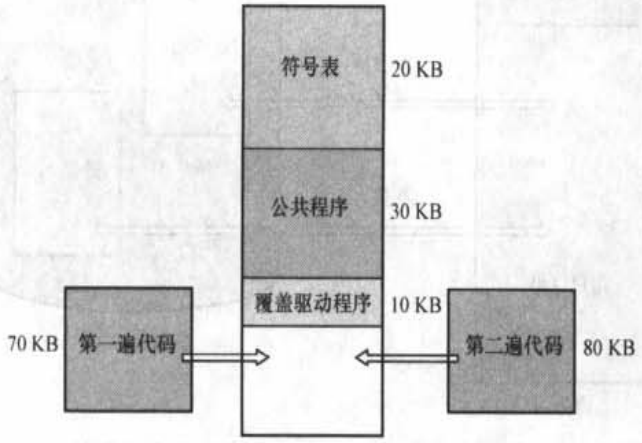


图 9.3　一个 two-pass 汇编程序的覆盖

覆盖 A 代码和覆盖 B 代码作为绝对内存映象存放在磁盘上，覆盖驱动程序根据需要将它们读入内存。为了构造覆盖，需要使用特殊的重定位和链接算法。

与动态加载一样，覆盖不需要操作系统提供特别支持。用户通过简单文件结构，将文件

读入内存、转到内存，并执行所读指令就可以完全实现覆盖。操作系统只不过注意到 I/O 操作比平常多些而已。

另一方面，程序员必须适当地设计和编写覆盖结构。这并不简单，需要对程序结构、代码、数据结构有完全了解。由于程序比较大才需使用覆盖，而小程序无需使用覆盖，因此获得对程序足够且完整的理解可能比较困难。由于这些原因，覆盖的使用通常局限于微处理机和只有有限物理内存且缺乏更先进硬件支持的其他系统。有的微处理机编译系统为程序员提供覆盖支持以简化编程。在有限物理内存运行大程序的自动技术当然更好。

9.2 交　　换

进程需要在内存中以便执行。不过，进程可以暂时从内存中**交换**(swap)出来到**备份存储**上，当需要再执行时再调回到内存中。例如，假如有一个 CPU 调度算法采用轮转法的多道程序环境。当时间片已到，内存管理器开始将刚刚执行过的进程换出，将另一进程换入到刚刚释放的内存空间中(图 9.4)。同时，CPU 调度器可以将时间片分配给其他已在内存中的进程。当每个进程用完时间片，它将与另一进程相交换。在理想情况下，内存管理器可以以足够快的速度交换进程，以便当 CPU 调度器需要调度 CPU 时，总有进程在内存内可以执行。时间片必须足够大以保证交换之间可以进行一定量的计算。

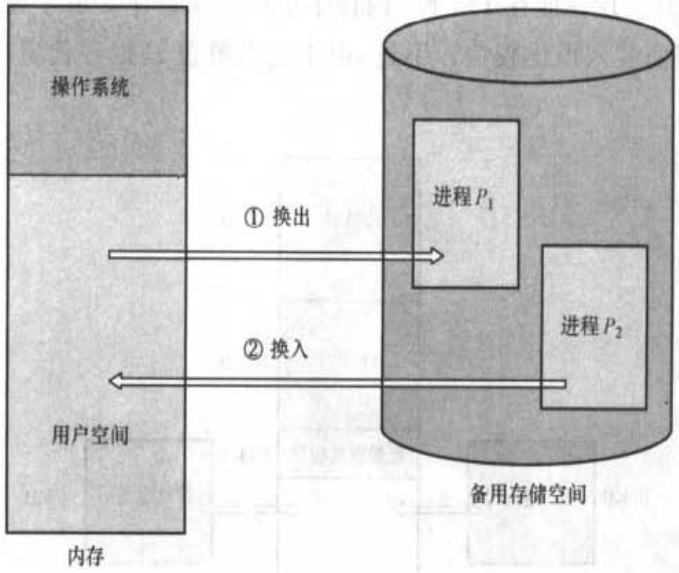


图 9.4　使用磁盘作为备份存储空间的两个进程的交换

这种交换策略的变种被用在基于优先权的调度算法中。如果一个更高优先级进程来了且需要服务，内存管理可以交换出低优先级的进程，以便可以装入和执行更高优先级的进程。当更高优先级进程执行完后，低优先级进程可以交换回内存以继续执行。这种交换有时称为**滚出**(roll out)、**滚进**(roll in)。

通常一个交换出的进程需要交换回它原来所占有的内存空间。这一限制是由地址捆绑方式决定的。如果捆绑是在汇编时或加载时所定的，那么就不可以移动到不同的位置。如果捆绑在运行时确定，由于物理地址是在运行时才确定的，那么进程可以移到不同的地址空间。

交换需要备份存储。备份存储通常是快速磁盘。这必须足够大，以便容纳所有用户的内存映象拷贝，它也必须提供对这些内存映象的直接访问。系统有一个**就绪队列**，它包括在备份存储或在内存中准备运行的所有进程。当 CPU 调度程序决定执行进程时，它调用派遣程序。派遣程序检查队列中的下一进程是否在内存中。如果不在内存中且没有空闲内存空间，派遣程序将一个已在内存中的进程交换出去，并换入所要的进程。然后，它重新装载寄存器，并将控制转交给所选择的进程。交换系统的关联转换时间比较长。为了明确关联切换时间的概念，假设用户进程的大小为 1 MB 且备份存储是传输速度为 5 Mb/s 的标准硬盘。1 MB 进程传入或传出内存的时间为

$$1\ 000\ \text{KB}/5\ 000\ \text{KBs} = 1/5\ \text{s}$$
$$= 200\ \text{ms}$$

假定无需磁头寻址且平均延迟为 8 ms，交换时间为 208 ms。由于需要换出和换入，因此总的交换时间约为 416 ms。

为了有效使用 CPU，需要使每个进程的执行时间比交换时间长。例如，对于轮转法 CPU 调度算法，时间片应比 416 ms 要大。

注意交换时间的主要部分是转移时间。总的转移时间与所交换的内存空间直接成正比。如果有这样一个计算机系统，其内存空间为 128 MB，驻留操作系统为 5 MB，用户的最大空间为 123 MB。但是，许多用户进程可能比这小很多，例如为 1 MB。1 MB 进程的换出需要 208 ms，而 123 MB 的交换需要 24.6 s。因此，知道一个用户进程所真正需要的内存空间，而不是其可能需要的内存空间，是非常有用的。这样，只需要交换出真正使用的内存，以便减少交换时间。为了有效使用这种方法，用户需要告诉系统其内存需求情况。因此，具有动态内存需要的进程要通过系统调用(请求内存和释放内存)来通知操作系统其内存需要变化情况。

交换也受其他因素所限制。如果要换出进程，那么必须确保该进程是完全处于空闲状态的。尤其值得关注的是待处理 I/O。假如要换出一个进程以释放内存，而该进程正在等待 I/O 操作。如果 I/O 异步访问用户内存的 I/O 缓冲区，那么该进程就不能被换出。假定由于设备忙，I/O 操作在排队等待。如果换出进程 P_1 而换入进程 P_2，那么 I/O 操作可能试图使用现在已属于进程 P_2 的内存。对这个问题有两种解决方法，一是不能换出有待处理 I/O 的进程，二是 I/O 操作的执行只能使用操作系统缓冲。仅当换入进程后，才执行操作系统缓冲与进程内存之间的数据转移。

交换不需要或只需要很少磁头移动，这需要进一步解释一下。有关详细情况，将在第十四章涉及有关外存时再讨论。简单地说，交换空间通常作为磁盘的一整块，且独立于文件系统，因此使用就可能很快。

现在，普通交换使用不多。交换需要很多时间，而且提供很少的执行时间，因此这不是一种有效内存管理解决方案。然而，一些交换的变种却在许多系统中得以使用。

一种修正过的交换就在许多 UNIX 版本中得以使用。交换通常不执行，但当有许多进程运行且内存空间吃紧时，交换开始启动。如果系统负荷降低，那么交换就暂停。UNIX 所用的内存管理将在附录 A.6 中讨论。

早期 PC 缺乏高级硬件(或者是能充分利用好硬件的操作系统)来实现高级内存的管理方法，但是通过使用修正过的交换可以运行多个大进程。一个重要例子就是微软公司的 Windows 3.1 操作系统内存中的进程可并发执行。如果需要装入一个进程，但没有足够内存空间，那么一个老的进程就被交换到磁盘上。该操作系统并不支持完全交换，这是因为不是调度器而是用户来决定何时为一个进程而抢占另一进程。换出的进程一直处于换出状态，直到用户再选择执行该进程为止。后来的微软公司操作系统，如 Windows NT 充分利用了个人计算机的高级 MMU 特性。在 9.6 节，将描述许多个人计算机所使用的 Intel 386 CPU 的内存管理硬件。到时候也将描述个人计算机所用的另一种高级操作系统 IBM OS/2 所使用的内存管理方法。

9.3 连续内存分配

内存必须容纳操作系统和各种用户进程。因此应该尽可能有效地分配内存的各个部分。本节将介绍一种常用方法：连续内存分配。

内存通常分为两个区域：一个用于驻留操作系统，另一个用于用户进程。操作系统可以放在低内存也可放在高内存。影响这一决定的主要因素是中断向量的位置。由于中断向量通常位于低内存，因此程序员通常将操作系统也放在低内存。在本书中，只讨论操作系统位于低内存的情况。其他情况的讨论类似。

通常需要将多个进程同时放在内存中。因此需要考虑如何为输入队列中需要调入内存的进程分配内存空间。采用连续内存分配时，每个进程位于一个连续的内存区域。

9.3.1 内存保护

在讨论内存分配前，必须先讨论内存保护问题：保护操作系统不受用户进程所影响，保护用户进程不受其他用户进程所影响。通过采用重定位寄存器(如 9.1.2 小节)和界限寄存器(如 2.5.3 小节)，可以实现这种保护。重定位寄存器含有最小的物理地址值；界限寄存器含有逻辑地址的值(例如，重定位＝100040，界限＝74600)。有了重定位寄存器和界限寄存器，每个逻辑地址必须小于界限寄存器；MMU 动态地将逻辑地址加上重定位寄存器的值后映射成物理地址。映射后的物理地址再送交内存单元(图 9.5)。

当 CPU 调度程序选择一进程执行时，作为上下文切换工作的一部分，派遣程序会用正

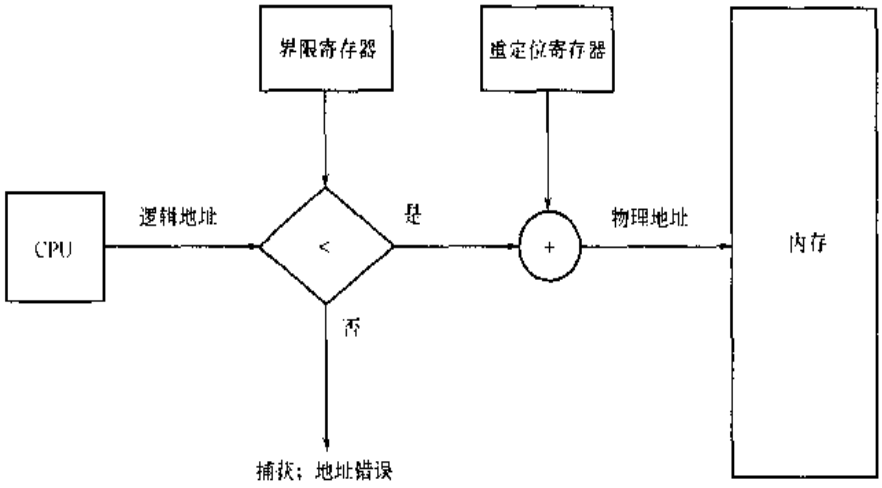


图 9.5 重定位和界限寄存器的硬件支持

确的值来初始化重定位寄存器和界限寄存器。由于 CPU 所产生的每一个地址都需要与寄存器进行核对，所以可以保证操作系统和其他用户程序及数据不受该进程的运行所影响。

重定位寄存器为允许操作系统动态改变提供了一个有效方法。许多情况都需要这一灵活性。例如，操作系统的驱动程序需要代码和缓冲空间。如果某驱动程序（或其他操作系统服务）不常使用，可以不必在内存中保留该代码和数据，这部分空间可以用于其他目的。这类代码有时称为**暂时**（transient）操作系统代码；它们根据需要再调入或调出。因此，使用这种代码可以在程序执行时动态地改变操作系统的大小。

9.3.2 内存分配

现在可以讨论内存分配。最为简单的内存分配方法之一就是将内存分为多个固定大小的**分区**。每个分区只能容纳一个进程。因此，多道程序的程度会受分区数所限制。如果使用这种**多分区方法**，当一个分区空闲时，可以从输入队列中选择一个进程，以调入到空闲分区。当进程终止时，其分区可以被其他进程所使用。这种方法最初为 IBM OS/360 操作系统（称为 MFT）所使用；现在已不再使用。下面所描述的方法是固定分区方案的推广（称为 MVT）；它主要用于批处理环境。这里所描述的许多思想也可用于采用纯分段内存管理的分时操作系统（9.5 节）。

操作系统有一个表用于记录哪些内存可用和哪些内存已用。一开始，所有内存都可用于用户进程，因此能够作为一大块可用内存，称为**孔**（hole）。当有新进程需要内存时，为该进程查找足够大的孔。如果找到，可以为该进程分配所需的内存，未分配的可以下次再用。

随着进程进入系统，它们将其加入到输入队列。操作系统根据所有进程的内存需要和现有可用内存情况来决定哪些进程可分配内存。当进程分配到空间时，它就装入内存，并开始竞争 CPU。当进程终止时，它将释放内存，该内存可以被操作系统分配给输入队列中的其他进程。

在任意时候，有一组可用空间和输入队列。操作系统根据调度算法来对输入队列进行排序。内存不断地分配给进程，直到下一个进程的内存需求不能满足为止，这时没有足够大的可用块(孔)来装入进程。操作系统可以等到有足够大的空间，或者往下扫描输入队列以确定是否有其他内存需求较小的进程可以被满足。

通常，一组不同大小的孔分散在内存中。当新进程需要内存时，系统为该进程查找足够大的孔。如果孔太大，那么就分为两块：一块分配给新进程；另一块还回到孔集合。当进程终止时，它将释放其内存，该内存将还给孔集合。如果新孔与其他孔相邻，那么将这些孔合并成大孔。这时，系统可以检查是否有进程在等内存空间，新合并的内存空间是否满足等待进程。

这种方法是通用**动态存储分配问题**的一种情况(根据一组空闲孔来分配大小为 n 的请求)。这个问题有许多解决方法。查找孔集合以决定哪个孔最适合分配。从一组可用孔中选择一个空闲孔的最为常用方法有**首次适应**(first-fit)、**最佳适应**(best-fit)、**最差适应**(worst-fit)。

- *首次适应*：分配第一个足够大的孔。查找可以从头开始，也可以从上次首次适应结束时开始。一旦找到足够大的空闲孔，就可以停止。
- *最佳适应*：分配最小的足够大的孔。必须查找整个列表，除非列表按大小排序。这种方法可以产生最小剩余孔。
- *最差适应*：分配最大的孔。同样，必须查找整个列表，除非列表按大小排序。这种方法可以产生最大剩余孔，该孔可能比最佳适应产生的较小剩余孔更为有用。

模拟结果显示首次适配和最好适配在执行时间和利用空间方面都好于最差适配。首次适配和最好适配在利用空间方面难分伯仲，但是首次适配要快些。

不过，这些算法都有**外部碎片问题**。随着进程装入和移出内存，自由内存空间被分为小片段。当所有总的内存之和可以满足请求，但并不连续时，就出现了外部碎片问题。该问题可能很严重。在最坏情况下，每两个进程之前就有空闲孔。如果这些内存是一整块，那么可能会再运行多个进程。

在首次适应和最佳适应之间的选择可能会影响外部碎片。(对有的系统，首次适应更好；而对其他系统，最佳适应更好)。另一因素是从空闲孔的哪端开始分配。(上面的，还是下面的。)不管使用哪种算法，外部碎片始终是个问题。

根据内存空间的总的大小和平均进程大小的不同，外部碎片或许次要或许重要。例如，对采用首次适应方法的统计说明，不管使用什么优化，假定有 N 个可分配块，那么可能有 $0.5N$ 个块为外部碎片。即，1/3 的内存可能不能使用。这一特性称为 50%**规则**。

9.3.3 碎片

内存碎片可以是内部的也可以外部的。设想有一个 18 464 B 大小的孔，并采用多分区

分配方案。假如有一个进程需要 18 462 B。如果只准确分配所要求的块，那么还剩下一个 2 B的孔。维护这一小孔的开销要比孔本身大很多。因此，通常将内存以固定大小的块为单元(而不是字节)来分配。采用这种方案，进程所分配的内存可能比所需要的要大。这两个数字之差称为**内部碎片**，这部分内存在分区内，而又不能用。

一种解决外部碎片问题的方法是**紧缩**(compaction)。紧缩的目的是移动内存内容，以便所有空闲空间合并成一整块。紧缩是有一定条件的。如果重定位是静态的，并且在汇编时或装入时进行的，那么就不能紧缩。如果重定位是动态的，是在运行时进行的，那么就能采用紧缩。对于动态重定位，可以首先移动程序和数据，然后再根据新基地址的值来改变基地址寄存器。如果能采用紧缩，还需要评估其开销。最简单的合并算法是简单地将所有进程移到内存的一端；而将所有的孔移到内存的另一端，以生成一个大的空闲块。这种方案比较昂贵。

另一种可能解决外部碎片问题的方法是允许物理地址空间为非连续，这样只要有物理内存就可为进程分配。这种方案有两种实现技术：分页(9.4 节)和分段(9.5 节)。这两种技术也可合并(9.6 节)。

9.4 分 页

分页(paging)内存管理方案允许进程的物理地址空间可以是非连续的。分页避免了将不同大小的内存块备份到交换空间上这样的麻烦问题，前面所述内存管理方案都有这个问题。当位于内存中的代码或数据需要换出时，必须先在备份存储上找到空间。备份存储也有如前面所述与内存相关的碎片问题，只不过访问更慢，因此不适宜采用合并。各种形式的分页，由于比以前所描述的方法更加优越，因此通常为绝大多数操作系统所采用。

传统而言，分页支持一直由硬件来处理的。然而，最近的设计是通过将硬件(尤其是 64 位微处理器)和操作系统相配合来实现分页。

9.4.1 基本方法

物理内存分为固定大小的块，称为**帧**(frame)。逻辑内存也分为同样大小的块，称为**页**。当进程需要执行时，其页从备份存储中调入到可用的内存帧中。备份存储也分为固定大小的块，其大小与内存的帧一样。

分页硬件支持如图 9.6 所示。由 CPU 所生成的每个地址分为两个部分：**页码**(p)和**页偏移**(d)。页号作为**页表**中的索引。页表包含每页所在物理内存的基地址。这些基地址与页偏移的组合就形成了物理地址，就可送交物理单元。内存的页模型如图 9.7 所示。

页大小(与帧大小一样)是由硬件来决定的。页的大小通常为 2 的幂，根据计算机结构的不同，其大小从 512 B 到 16 MB 字节不等。选择页的大小为 2 的幂可以方便地将逻辑地

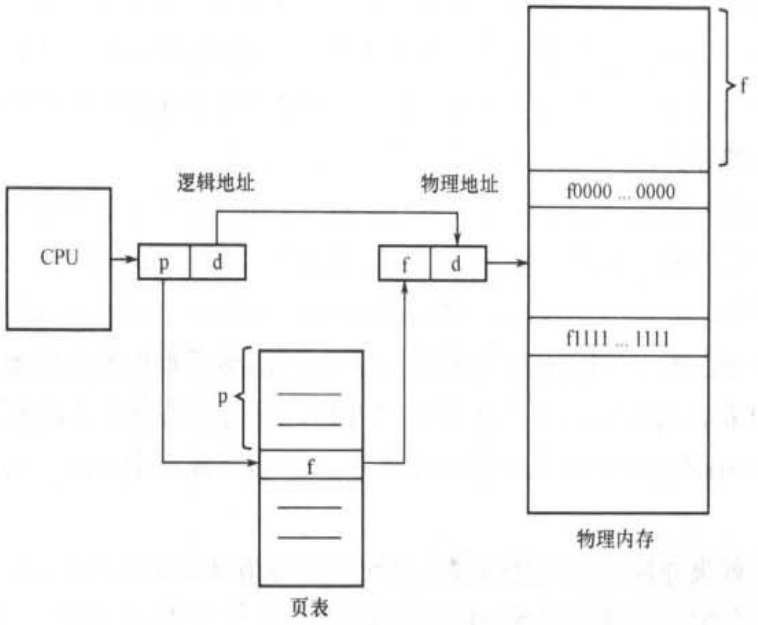


图 9.6 分页的硬件支持

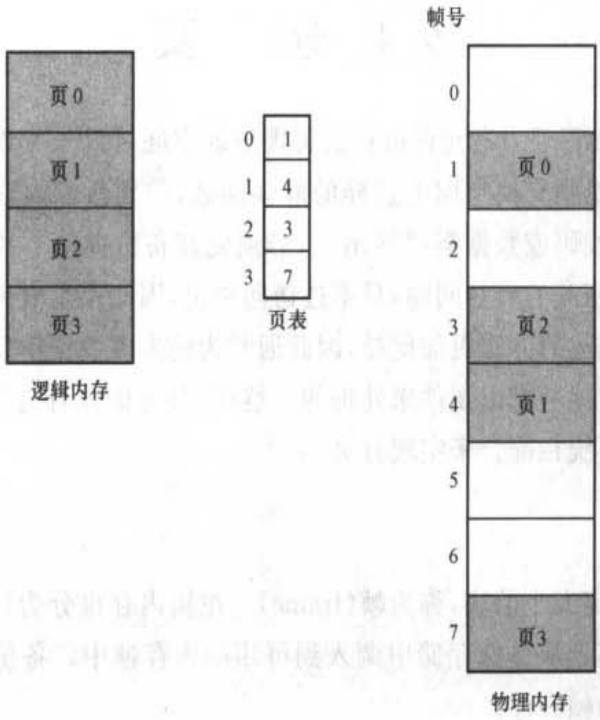
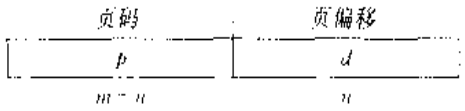


图 9.7 逻辑内存和物理内存的分页模型

址转换为页码和页偏移。如果逻辑地址空间为 2^m，且页大小为 2^n 单元(字节或词)，那么逻辑地址的高 $m-n$ 位表示页号，而低 n 位表示页偏移。这样，逻辑地址就如下所示：



其中，p 作为页表的索引，而 d 作为页的偏移。

举一个具体(极小)例子，考虑一下图 9.8 所示的内存。如果页大小为 4 B，而物理内存为 32 B(8 页)，那么来考虑一下用户观点的内存是如何映射到物理内存的。逻辑地址 0 的页码为 0，页偏移为 0。根据页表，可以查到页码 0 对应为帧 0，因此逻辑地址 0 映射为物理地址 20(=(5×4)+0)。逻辑地址 3 映射物理地址 23(=(5×4)+3)。逻辑地址 4 的页码为 1，页偏移为 0。由于页码 1 对应为帧 6，因此，逻辑地址 4 映射为物理地址 24(=(6×4)+0)。逻辑地址 13 映射为物理地址 9。

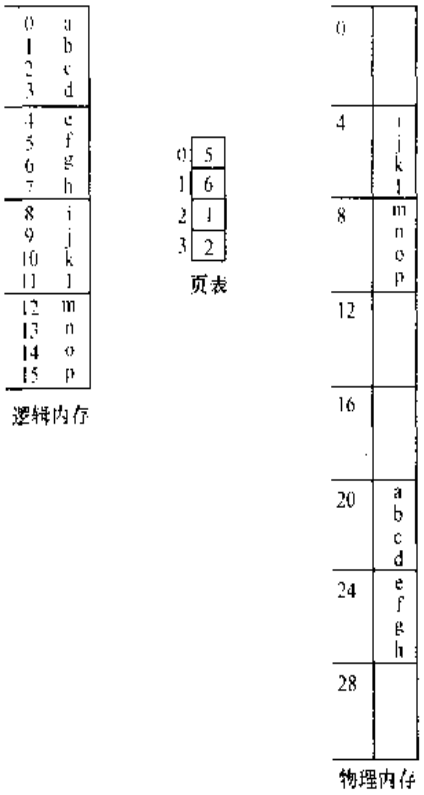


图 9.8 使用 4 B 的页对 32 B 的内存进行分页的例子

读者可能注意到分页也是一种动态重定位。每个逻辑地址由分页硬件捆绑为一定的物理地址。采用分页类似于使用一组基(重定位)地址寄存器，每个基地址对应着一个内存帧。

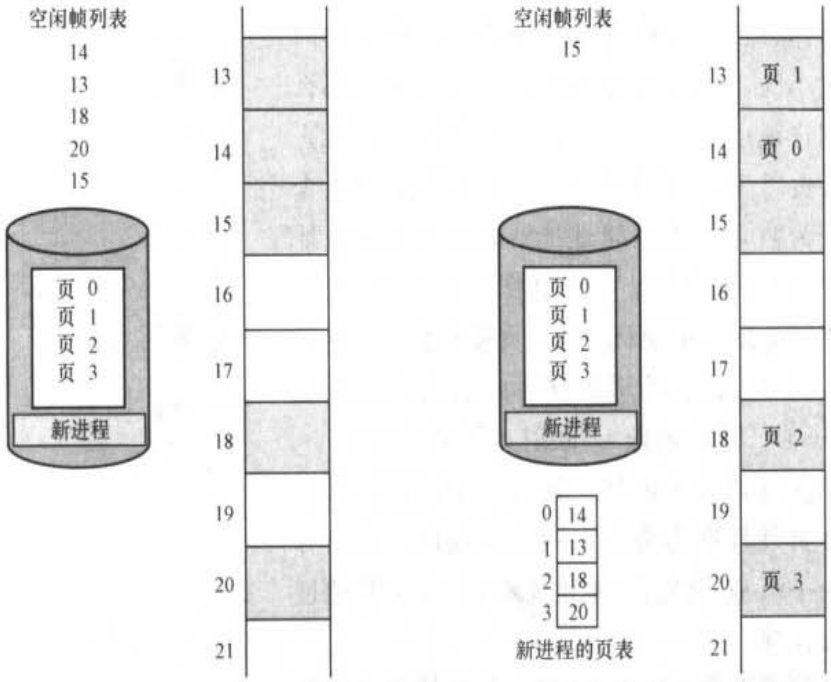
采用分页技术不会产生外部碎片：每个帧都可以分配给需要它的进程。不过，分页有内部碎片。注意分配是以帧为单元进行的。如果进程所要求的内存并不是页的的整数倍，那么最后一个帧就可能用不完。例如，如果页的大小为 2 048 B，一个大小为 72 766 B 的进程需要 35 个页和 1 086 B。该进程会得到 36 个帧，因此有 2 048 − 1 086 = 962 B 的内部碎片。在最坏情况下，一个需要 n 页再加 1 B 的进程，需要分配 $n+1$ 个帧，这样几乎产生整个一个帧的内部碎片。

如果进程大小与页大小无关，那么可以推测每个进程可能有半页的内部碎片。这一结构意味着小一点的页可能好些。不过，页表中的每项也有一定的开销，该开销随着页的增大而降低。而且，磁盘 I/O 操作随着传输量的增大会更为有效(第十四章)。一般来说，随着时间的推移，页的大小也随着进程、数据和内存的不断增大而增大。现在，页的大小通常为 4 KB或 8 KB，有的系统可能支持更大的页。有的 CPU 的内核可能支持多种页大小。例如，Solaris 根据按页所存储的数据，可使用 8 KB 或 4 MB 的大小的页。研究人员正在研究对快速可变的页大小的支持。

每个页表的条目通常为 4 B，不过这是可变的。一个 32 bit 的条目可以指向 2^{32} 个物理帧中的任一个。如果帧为 4 KB，那么 4 B 条目可以访问 2^{44} B(或 16 TB)大小的内存。

当系统需要执行一个进程时，它将检查该进程所需要的页数。因此，如果进程需要 n

页，那么内存中至少应有 n 个帧。如果有，那么就可分配给新进程。进程的第一页装入一个已分配的帧，帧号放入进程的页表中。下一页分配给另一帧，其帧号也放入进程的页表中，等等(图 9.9)。



(a) 在分配前　　(b) 在分配后

图 9.9 空闲帧

分页的一个重要特点是用户观点的内存和实际的物理内存的分离。用户程序将内存作为一整块来处理，而且它只包括一个进程。事实上，一个用户程序与其他程序一起，分布在物理内存上。用户观点的内存和实际的物理内存的差异是通过地址转换硬件来调和的。逻辑地址转变成物理地址。这种映射是用户所不知道的，但是受操作系统所控制。注意用户进程根据定义是不能访问非它所占用的内存的。它无法访问其页表所规定之外的内存，页表只包括进程所拥有的那些页。

由于操作系统管理内存，它必须知道物理内存的分配细节：哪些帧已分配，哪些帧空着，总共有多少帧，等等。这些信息通常保存在称为帧表的数据结构中。在**帧表**(frame table)中，每个条目对应着一个帧，以表示该帧是空闲还是已被占用，如果被占用，是为哪个进程的哪个页所占用。

另外，操作系统必须意识到用户进程是在用户空间内执行，且所有逻辑地址必须映射到物理地址。如果用户执行一个系统调用(例如进行 I/O)，并提供地址作为参数(如一个缓冲)，那么这个地址必须映射成物理地址。操作系统为每个进程维护一个页表的拷贝，就如同它需要维护指令计数器和寄存器的内容一样。当操作系统必须手工将逻辑地址映射成物理地址时，这个拷贝可用做转换。当一个进程可分配到 CPU 时，CPU 派遣程序可以根据该

拷贝来定义硬件页表。因此，页表增加了切换时间。

9.4.2 硬件支持

每个操作系统都有自己的方法来保存页表。绝大多数都为每个进程分配一个页表。页表的指针与其他信息(如指令计数器)一起存入进程控制块中。当分派程序需要启动一个进程时，它必须首先装入用户寄存器，并根据所保存的用户页表来定义正确的硬件页表值。

页表的硬件实现有多种方法。最为简单的一种方法是将页表作为一组专用**寄存器**来实现。这些寄存器应用高速逻辑电路来构造，以便有效地进行分页地址的转换。由于对内存的每次访问都要经过分页表，因此效率很重要。CPU 分派程序在装入其他寄存器时，也需要装入这些寄存器。装入或修改页表寄存器的指令是特权级的，因此只有操作系统才可以修改内存映射图。DEC PDP-11 就是这种类型的结构。它的地址有 16 bit，而页面大小为 8 KB。因此页表有 8 个条目可放在快速寄存器中。

如果页表比较小(例如，256 个条目)，那么页表使用寄存器还是比较合理的。但是，绝大多数当代计算机都允许页表非常大(例如，一百万个条目)。对于这些机器，采用快速寄存器来实现页表就不可行了。因而需要将页表放在内存中，并将**页表基寄存器**(PTBR)指向页表。改变这些页表只需要改变这一寄存器即可，这也大大降低了切换时间。

采用这种方法的问题是访问用户内存位置所需要的时间。如果要访问位置 i，那么必须先用 PTBR 中的值再加上页码 i 的偏移，来查找页表。这一任务需要内存访问。根据所得的帧号，再加上页偏移，就得到了真实物理地址。接着就可以访问内存中所需的位置。采用这种方案，访问一个字节需要两次内存访问(一次用于页表条目，一次用于字节)。这样，内存访问的速度就减半。在绝大多数情况下，这种延迟是无法忍受的。还不如采用交换机制。

对这一问题的标准解决是采用小但专用且快速的硬件缓冲，这种缓冲称为**翻译后备缓冲器**(translation look-aside buffer，TLB)。TLB 是关联的快速内存。TLB 条目由两部分组成：键(标签)和值。当关联内存根据给定值查找时，它会同时与所有键进行比较。如果找到条目，那么就得到相应的值域。这种查找方式比较快，不过硬件也比较昂贵。通常，TLB 的条目数并不多，通常在 64 到 1 024 之间。

TLB 与页表一起按如下方法使用。TLB 只包括页表中的一小部分条目。当 CPU 产生逻辑地址后，其页号提交给 TLB。如果找到页号，那么也就得到了帧号，并可用来访问内存。整个任务与不采用内存映射相比，其时间长度缩短了 10%。

如果页号不在 TLB 中(称为 TLB **失效**)，那么就需要访问页表。当得到帧号后，就可以用它来访问内存(如图 9.10)。同时，将页号和帧号增加到 TLB 中，这样下次再用时就可很快查找到。如果 TLB 中的条目已满，那么操作系统会选择一个来替换。替换策略有很多，从最近最少使用替换(LRU)到随机替换等。另外，有的 TLB 允许某些条目**固定下来**，也就是说它们不会从 TLB 中被替换。通常内核代码的条目是固定下来的。

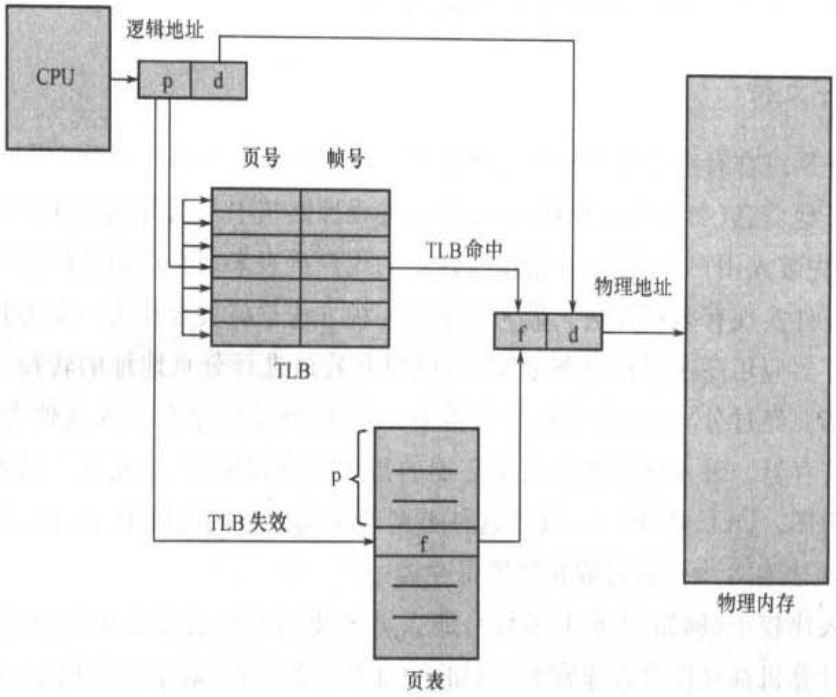


图 9.10 带 TLB 的分页硬件

有的 TLB 在每个 TLB 条目中还保存**地址空间标识符**(address-space identifier, ASID)。ASID 可用来惟一地标识进程,可用来为进程提供地址空间保护。当 TLB 试图解析虚拟页码时,它确保当前运行进程的 ASID 与虚拟页相关的 ASID 相匹配。如果不匹配,那么就作为 TLB 失效。除了提供地址空间保护外,ASID 也允许 TLB 同时包括多个不同进程的条目。如果 TLB 不支持独立的 ASID,每次选择一个页表时(例如,切换进程时),TLB 就必须被**冲刷**(flush)(或删除),以确保下一个进程不会使用错误的地址转换。否则,TLB 中可能有老的条目,这些条目虽然包含有效的虚拟地址,但是却是由前面进程所留下的不正确的或者无效的物理地址(从上一个进程留下来的)。

特定页码在 TLB 中被查找到的百分比称为**命中率**。80%的命中率意味着有 80%的时间可以在 TLB 中找到所需的页号。假如查找 TLB 需要 20 ns,访问内存需要 100 ns,如果访问位于 TLB 中的页码时,那么采用内存映射访问需要 120 ns。如果不能在 TLB 中找到(20 ns),那么必须先访问位于内存中的页表以得到帧码(100 ns),并进而访问内存中所需的字节(100 ns),这总共要花费 220 ns。为了得到**有效内存访问时间**,必须根据概率来对每种情况进行加权。

有效访问时间=0.80×120+0.20×220=140 ns

对于这种情况,现在内存访问速度要慢 40%(从 100 ns 到 140 ns)。

如果命中率为 98%,那么

有效访问时间=0.98×120+0.02×220 = 122 ns

由于提高了命中率,现在内存访问速度只慢 22%。在第十章,将进一步讨论命中率对 TLB 的影响。

9.4.3 保护

在分页环境下，内存保护是通过与每个帧相关联的保护位来实现的。通常，这些位保存在页表中。任何一位都能定义一个页是可读、可写或只可读。每次地址引用都要通过页表来查找正确的帧码，在计算物理地址的同时，可以通过检查保护位来验证有没有对只可读页进行写操作。对只读页进行写操作会向操作系统产生硬件陷阱(或内存保护冲突)。

可以很容易地扩展这一方法以提供更细致的保护。可以创建硬件以提供只读、读写、只执行保护。或者，通过为每种访问情况提供独立保护位，可以实现这些访问的各种组合；非法访问会被操作系统捕捉到。

还有一个位通常与页表中的每一条目相关联：**有效－无效位**。当该位为有效时，该值表示相关的页在进程的逻辑地址空间内，因此是合法(或有效)的页。如果该位为无效时，该值表示相关的页不在进程的逻辑地址空间内。非法地址通过使用有效位会被捕捉。操作系统通过对该位的设置可以允许或不允许对某页的访问。例如，对于 14 位地址空间(0 到 16 383)的系统，有一个程序，其地址空间为 0 到 10 468。如果页的大小为 2 KB，那么得到如图 9.11 所示的页表。页 0、1、2、3、4 和 5 的地址可以通过页表正常映射。然而，如果试图产生页表 6 或 7 中的地址时，就会发现有效位为无效，这样操作系统就会捕捉到这一非法操作(无效地址引用)。

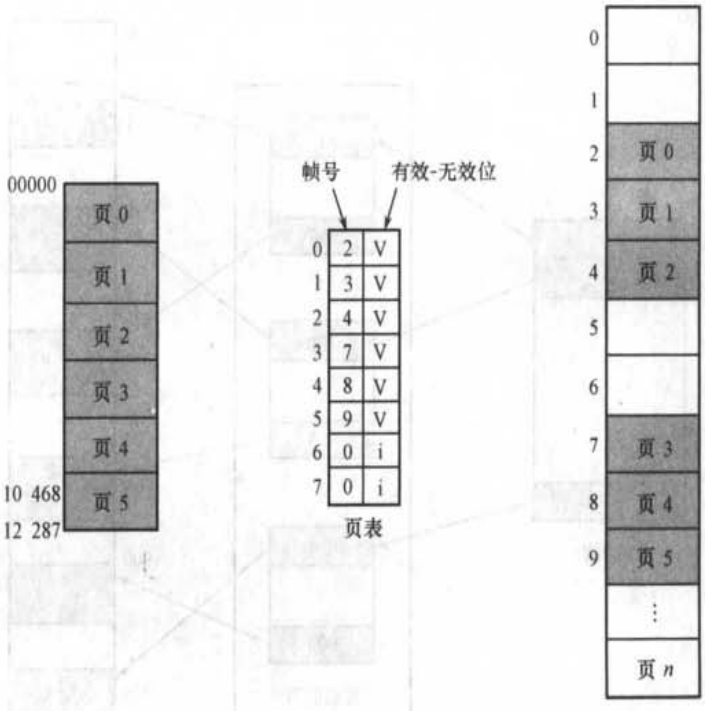


图 9.11　页表中的有效－无效位

由于程序的地址只到 10 468，所以任何超过该地址的引用都是非法的。不过，由于对页

5 的访问是有效的，因此到 12 287 为止的地址都是有效的。只有 12 288 到 16 383 的地址才是无效的。这个问题是由于页大小为 2 KB 的原因，也反映了分页的内部碎片。

一个进程很少会使用其所有的地址空间。事实上，许多进程只使用一小部分可用的地址空间。对这些情况，如果为地址范围内的所有页，都在页表中建立一个条目将是非常浪费的。表中的绝大多数将并不会被使用，但却占用可用的地址空间。有些系统提供硬件如**页表长度寄存器**(PTLR)来表示页表的大小，该寄存器的值可用于检查每个逻辑地址以验证其是否位于进程的有效范围内。如果检测无法通过，会被操作系统捕捉到。

9.4.4 页表结构

在本节，将讨论一些组织页表的常用技术。

1. 层次化分页

绝大多数现代计算机系统支持大逻辑地址空间(2^{32}到2^{64})。在这种情况下，页表本身可以非常大。例如，设想一下具有 32 位逻辑地址空间的计算机系统。如果系统的页大小为 4 KB(2^{12})，那么页表可以拥有一百万条目($2^{32}/2^{12}$)。假设每个条目有 4 B，那么每个进程需要 4 MB 物理地址空间来存储页表本身。显然，人们并人不愿意在内存中连续地分配这个页表。这个问题的一个简单解决是将页表划分为更小部分。完成这种划分有许多方法。

一种方法是使用两层分页算法，就是将页表再分页(图 9.12)。记住对这种 32 位系统的

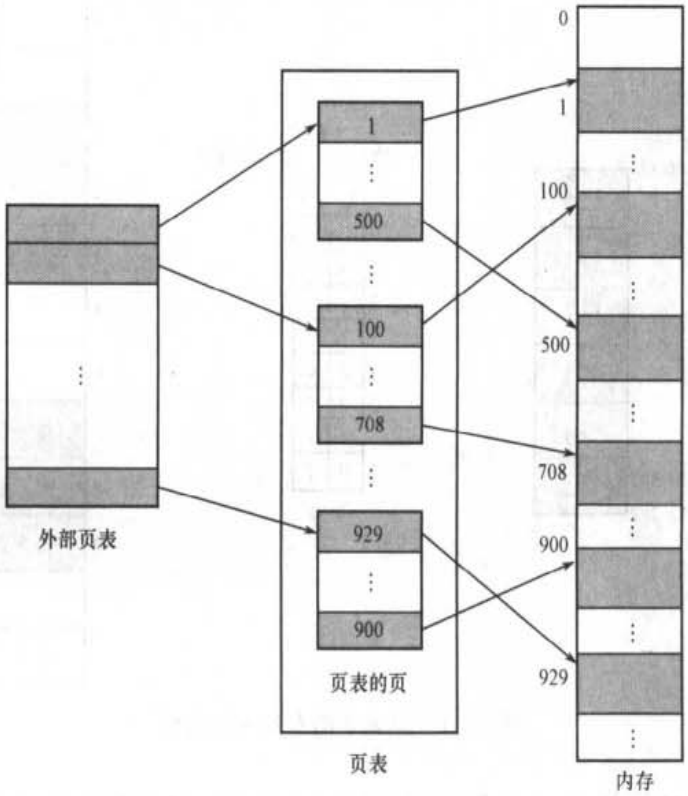


图 9.12 一个两级页表方法

页大小为 4 KB。一个逻辑地址被分为 20 bit 的页号和 12 bit 的页偏移。因为要对页表进行再分页，所以该页号可分为 10 bit 的页号和 10 bit 的页偏移。这样，一个逻辑地址就分为如下形式：

页号		页偏移
p_1	p_2	d
10	10	12

其中 p_1 是用来访问外部页表的索引，而 p_2 是外部页表的页偏移。采用这种结构的地址转换方法如图 9.13 所示。由于地址转换由外向内，这种方案也称为**向前映射页表**。Pentium Ⅱ就采用这种方法。

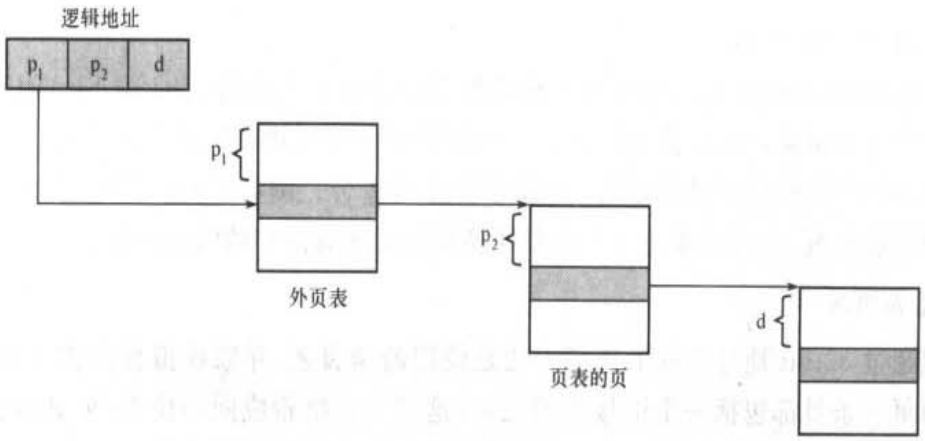


图 9.13　二级 32 位分页体系结构的地址转换

VAX 体系结构也支持一种两层分页的变种。VAX 是 32 位机器，其页大小为 512 B。进程的逻辑地址空间被分为四个区，每个区为 2^{30} B。每个区表示一个进程的逻辑地址空间的不同部分。逻辑地址的前两个高位表示适当的区。中间 21 bit 表示区内的页码，后 9 bit 表示所需页中的偏移。通过按这种方式来分页，操作系统可以只在进程需要时才使用某些分区。VAX 体系结构的地址按如下形式分为：

区	页	偏移
s	p	d
2	21	9

其中 s 表示区号，p 表示页表的索引，而 d 表示页内的偏移。

一个 VAX 进程如果使用一个区，其一层的页表大小为 $2^{21} \times 4$ B=8 MB。为了减少内存使用，VAX 对用户的页表继续进行分页。

对于 64 位的逻辑地址空间，两层分页方案就不再适合。为了说明这一点，假设系统的页大小为 4 KB(2^{12})。这时，页表可由 2^{52} 条目组成。如果使用两层分页，那么内部页表可方便地定为一页长，或包括 2^{10} 个 4 B 的条目。地址形式如下所示：

外页表	内页表	偏移
p_1	p_2	d
42	10	12

外部页表有 2^{42} 条目，或 2^{44} B。避免这样一个大页表的显而易见方法是将外部页表再进一步细分。这种方法也可用于 32 位处理器，能够更加灵活和有效。

可以有很多方式来划分外部表。可以再分外部页表，得到三层分页方案。假设外部表由标准大小的页组成(2^{10} 条目或者 2^{12} B)组成；那么 64 位地址空间仍然是很大的。

第 2 个外页	外页	内页	偏移
p_1	p_2	p_3	d
32	10	10	12

最外部表的大小为 2^{34} B。

下一步是四层分页机制，这时每二级的外部页表被再次细分。32 位 SPARC 体系结构采用了三层分页机制，而 32 位 Motorola 68030 体系结构采用了四层分页机制。

然而，对于 64 位体系结构，层次页表通常并不适合。例如，64 位 UltraSPARC 体系结构可采用 7 层分页，这可以说是一个有效转换逻辑地址的内存访问的极限。

2. 哈希页表

处理超过 32 bit 地址空间的常用方法是使用**哈希页表**，并以虚拟页码作为哈希值。哈希页表的每一条目都包括一个链接组的元素，这些元素哈希成同一位置(要处理碰撞)。每个元素有三个域：(a) 虚拟页码，(b) 所映射的帧码，(c) 指向链表中下一个元素的指针。

该算法工作如下：虚拟地址中的虚拟页码转换到哈希表中。用虚拟页码与链表中的每一个元素的第一个域相比较。如果匹配，那么相应的帧码(第二个域)就用来形成物理地址。如果不匹配，那么就对链表中的下一个域进行比较页码。该方案见图 9.14。

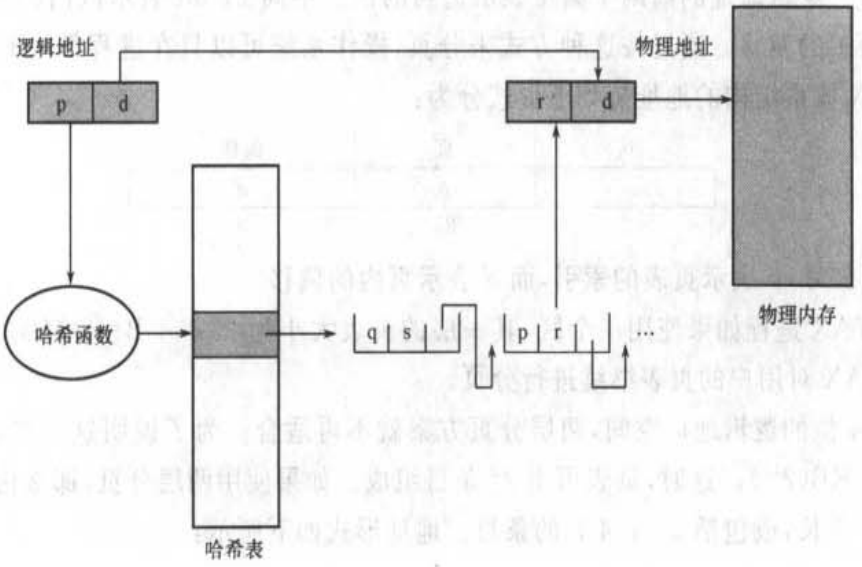


图 9.14 哈希页表

人们提出了这种方法的一个变种，它比较适合 64 位的地址空间。**群集页表**(clustered page table)类似于哈希页表，不过这种哈希页表的每一条目不只包括一页信息而是包括多页(例如 16)。因此，一个页表条目可以存储多个物理帧的映射。群集页表对于稀疏地址空间特别有用，稀疏地址空间中的地址引用不连续，且分散在整个地址空间。

3. 反向页表

通常，每个进程都有一个相关页表。该进程所使用的每个页都在页表中有一项(或者每个虚拟页都有一项，不管后者是否有效)。这种表示方式比较自然，这是因为进程是通过虚拟地址来引用页的。操作系统必须将这种引用转换成物理内存地址。由于页表是按虚拟地址排序的，操作系统能够计算出所对应条目在页表中的位置，并可以直接使用该值。这种方法的缺点之一是每个页表可能有很多项。这些表可能消耗大量物理内存，这些只不过用来跟踪物理内存是如何使用的。

为了解决这个问题，可以使用**反向页表**(inverted page table)。反向页表对于每个真正的内存页或帧才有一个条目。每个条目包含保存在真正内存位置的页的虚拟地址，以及拥有该页的进程的信息。因此，整个系统只有一个页表，对每个物理内存的页只有一条相应的条目。图 9.15 说明了反向页表的工作原理。因为系统只有一个页表，而有多个地址空间映射物理内存，所以反向页表的条目中通常需要一个地址空间标识符，以确保一个特定进程的一个逻辑页可以映射到相应的物理帧。采用反向页表的系统包括 64 位 UltraSPARC 和 PowerPC。

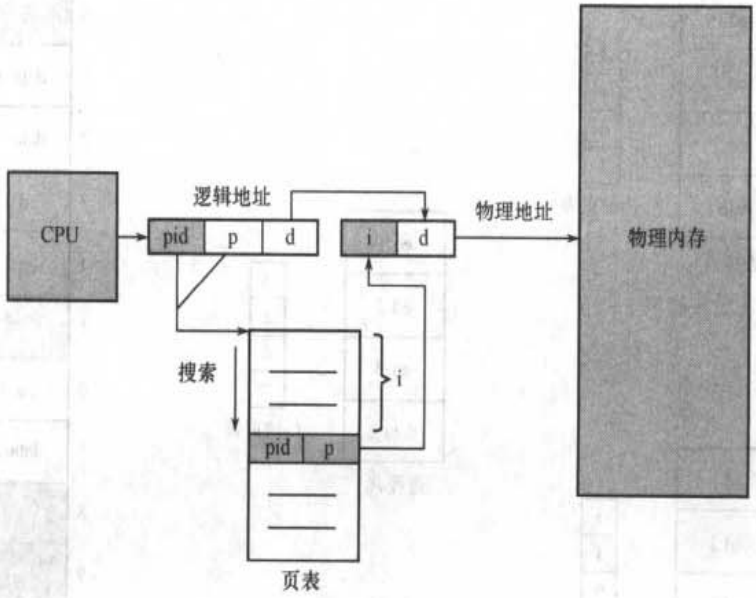


图 9.15　反向页表

为了说明这种方法，描述一种简化的反向页表实现，IBM RT 就使用它。系统的每个地址有一个三元组

<process-id, page-number, offset>

每个反向页表的条目是一对<process-id, page-number>，其中 process-id 用来作为地址空间的标识符。当需要内存引用时，由<process-id, page-number>组成的虚拟地址部分送交内存子系统。通过查找反向页表来寻找匹配。如果匹配找到，例如条目 i，那么就产生了物理地址<i, offset>。如果没有匹配，那么就是试图进行非法地址访问。

虽然这种方案减低了存储每个页表所需要的内存空间，但是当引用页时它增加了查找页表所需要的时间。由于反向页表是按物理地址排序，而查找是根据虚拟地址，因此可能需要查找整个表来寻求匹配。这种查找会花费很长时间。为了解决这一问题，可以使用哈希表来限制查找一个或少数几个条目。当然，每次访问哈希表也增加了一个对子程序的调用，因此，虚拟地址引用至少需要两个内存读：一个查找哈希表条目，另一个查找页表。为了改善性能，记住在访问哈希表时先查找 TLB。

9.4.5 共享页

分页的另一个优点是可以共享共同代码。这一点对分时环境特别重要。设想一下这样的系统，有 40 个用户，每个用户都执行一个文本编辑器。如果文本编辑器有 150 KB 代码段和 50 KB 数据段，需要 8 000 KB 来支持 40 个用户。然而，如果代码是**重入代码**(reentrant code)，那么就可以共享，如图 9.16 所示。这里，编辑器有三页，每页的大小为 50 KB；这么大的页只是用来简化图形而已，这些页可为三个进程所共享。每个进程都有自己的数据页。

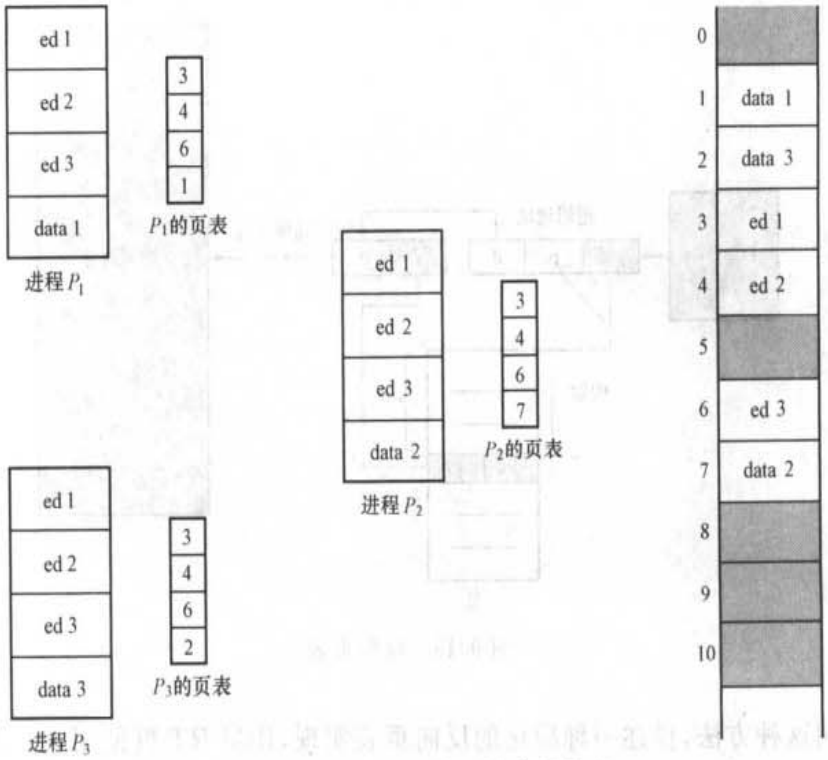


图 9.16 在分页环境下的代码共享

重入代码(或纯代码)是不能自我改变的代码。如果代码是重入的,那么它在执行时决不会改变自己。因此,两个或更多进程可以同时执行代码。每个进程都有自己的寄存器拷贝和保存进程执行所需数据的数据存储拷贝。当然,两个不同进程的数据可以不同。

只需要在物理内存中保存一个编辑器拷贝。每个用户的页表映射到编辑器的同一物理拷贝,而数据页映射到不同的帧。因此,为支持 40 个用户,只需要一个编辑拷贝(150 KB),再加上而 40 个用户数据空间 50 KB。总的需求空间为 2 150 KB,而不是 8000 KB,这是一个重要的节省。

其他常用程序也可能共享,例如,编译器、窗口系统、运行时库、数据库系统等。要共享,代码必须重入。共享代码的只读特点不能只通过正确代码来保证,而需要操作系统来强制实现。一个系统多个进程的内存共享类似于一个任务的多线程地址空间的共享(如第五章所述)。而且,回想一下第四章,已描述过共享内存作为一种进程通信机制。有的操作系统通常共享页来实现共享内存。

采用反向页表的系统在实现代码共享方面有一定困难。共享内存通常是通过多个虚拟页映射到共同的物理地址来实现的(每个共享进程都有一个页)。这种标准方法在这里并不适合,因为每个物理页只有一个虚拟页的条目,因此一个物理页不能有多个共享虚拟地址。

除了允许多个进程共享相同的物理页,按页组织内存也提供许多其他优点。在第十章,将讨论其他优点。

9.5 分　　段

采用分页内存管理有一个不可避免的问题:用户观点的内存和实际内存的分离。用户观点的内存与实际内存不一样。用户观点的内存需要映射到实际内存。该映射允许逻辑内存和物理内存的不同。

9.5.1 基本方法

用户是否会将内存看做为一个线性字节数组,有的包含指令而其他的包含数据?绝大多数人会说不。用户通常会愿意将内存看做为一组不同长度的段的集合,这些段之间并没有一定的顺序(图 9.17)。

想一下你在写程序时是如何考虑程序的。你会认为程序是由主程序加上一些子程序、过程、函数或模块所构成的。还有各种数据结构:表、数组、堆栈、变量等。每个模块或其他数据元素都可通过名称引用。你会说"符号表"、"函数 *sqrt*"、"主函数",而并不关心这些元素所在内存的位置。你不关心符号表是放在函数 *sqrt* 之前还是之后。这些段的长度是不同的,其长度是由这些段在程序中的目的所定义的。段内的元素是由它们距段首的偏移来决定的:程序的第一条语句、符号表的第十七条目、函数 *sqrt* 的第五条指令等。

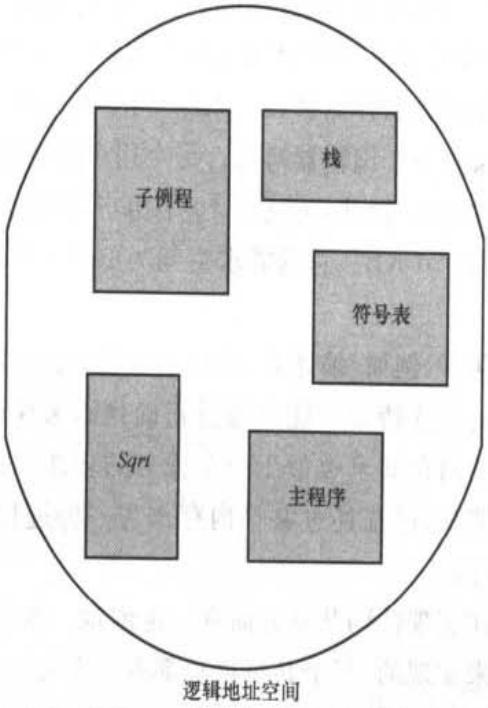


图 9.17 用户对一个程序的看法

分段就是支持这种用户观点的内存管理方案。逻辑地址空间是由一组段组成。每个段都有名称和长度。地址指定了段名称和段内偏移。因此用户通过两个量来指定地址:段名称和偏移。(请将这一方案与分页相比较。在分页中,用户只指定一个地址,该地址通过硬件分为页码和偏移,程序员是看不见这些的。)

为了实现简单起见,段是编号的,是通过段号而不是段名来引用。因此,逻辑地址*由两个元素组成*:

<段号,偏移>

通常,在编译用户程序时,编译器会自动根据输入程序来构造段。一个 Pascal 编译器可能会创建如下不同的段:

1. 全局变量;
2. 过程调用堆栈,以用于保存参数和返回地址;
3. 每个过程或函数的代码部分;
4. 每个过程或函数的局部变量部分。

一个 Fortran 编译器可能会为每个公共块保存段。数组也可作为独立的段。加载程序会装入所有这些段,并为它们分配段号。

9.5.2 硬件

虽然用户现在能够通过二维地址来引用程序中的对象,但是实际物理内存仍然是一维

序列的字节。因此，必须定义一个实现方式，以便将二维的用户定义地址映射为一维物理地址。这个地址是通过**段表**来实现的。段表的每个条目都有段*基地址*和*段界限*。段基地址包含该段在内存中的开始物理地址，而段界限指定该段的长度。

段表的使用如图 9.18 所示。一个逻辑地址由两部分组成：段号 s 和段内的偏移 d。段号用做段表的索引。逻辑地址的偏移 d 应位于 0 和段界限之间。如果不是这样，则会陷入到操作系统中(逻辑地址试图访问段的外面)。如果偏移 d 合法，那么就与基地址相加而得到所需字节在物理内存的地址。因此段表是一组基址和界限寄存器对。

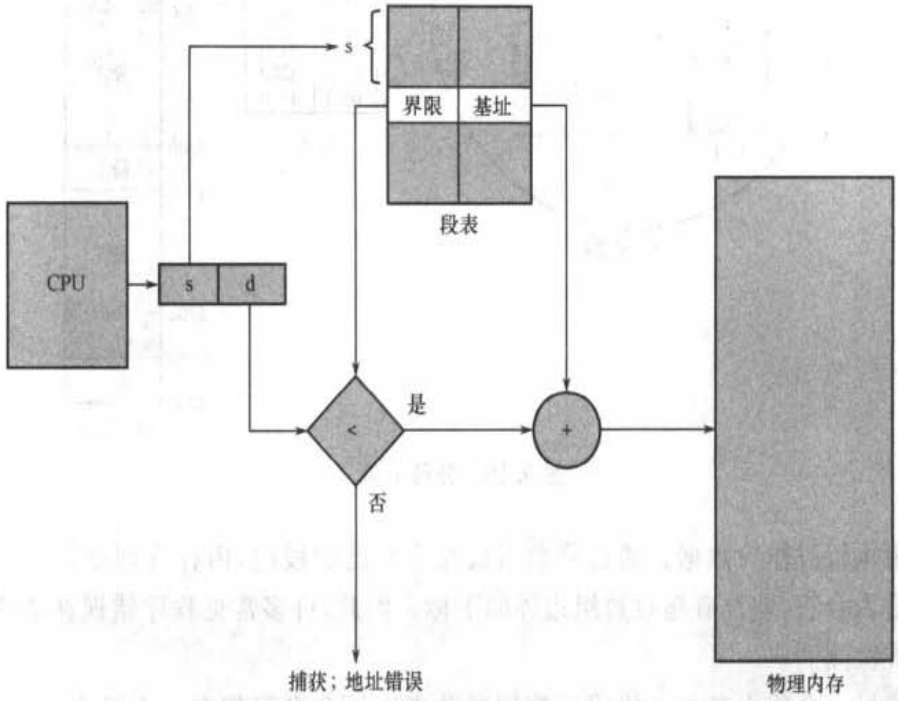


图 9.18　分段硬件

作为一个例子，考虑一下图 9.19 所示的情况。有 5 个段，按 0 到 4 来编号。各段按图所示存储。每个段都在段表中有一个条目，它包括段在物理内存中的开始地址和该段的长度(界限)。例如，段 2 为 400 B 长，开始于位置 4300。因此，对段 2 字节 53 的引用映射成位置 4300+53=4353。对段 3 字节 852 的引用映射成位置 3200+852=4052。段 0 字节 1222 的引用会陷入操作系统，这是由于该段仅为 1 000 B 长。

9.5.3　保护与共享

分段的一个显著优点是可以将段与对其的保护相关联。因为段表示一个有一定语义的程序部分，所以段内的所有内容可能会按同样方式使用。因此，有的段是指令，而有的段是数据。对于现代体系结构，指令不可以自我修改，故指令段可定义为只读或只执行。内存映射硬件会检查与段条目相关联的保护位以防止对内存的非法访问，如试图对只读段进行写

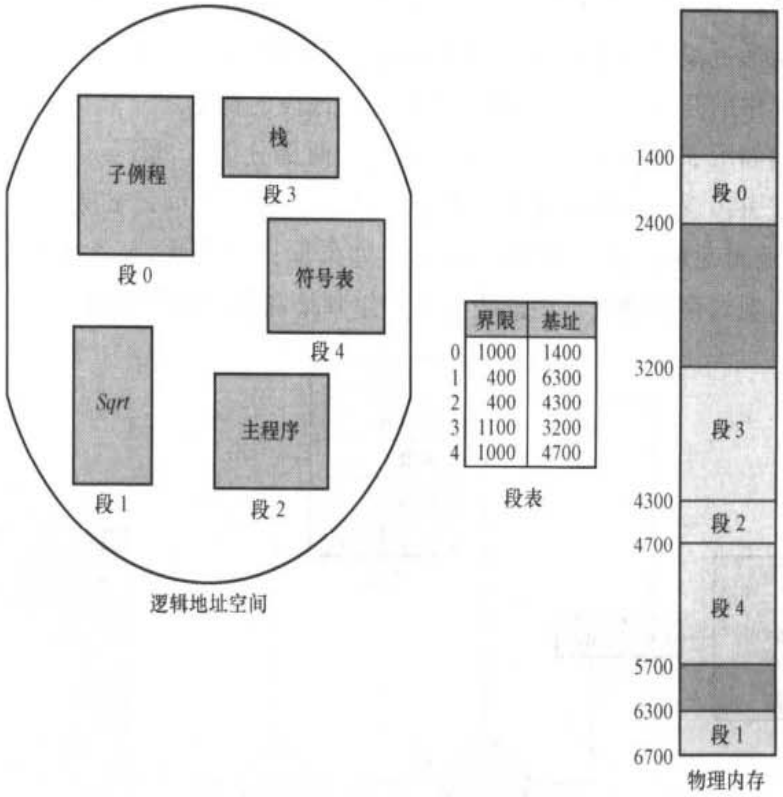


图 9.19 分段的例子

操作或使用执行段作为数据。通过将数组放在它自己的段内，内存管理硬件可以自动检查数组下标是否合法，即没有超过数组边界的下标。因此，许多常见程序错误在造成严重损失之前就已被检测到。

分段的另一个优点是关于代码或数据的共享。每个进程都有一个段表，当该进程被允许使用 CPU 时，派遣程序会定义一个硬件段表。当两个进程的某些条目指向同一物理位置时，就可共享段(图 9.20)。

共享在段级进行。因此，段内所定义的任何信息均可共享。多个段可以共享，因此一个由多个段组成的程序也可共享。

例如，考虑一个分时系统中的文本编辑器的共享。一个完整的编辑器可能很大，由许多段组成。这些段可以为多个用户所共享，从而减少了支持编辑任务所需要的物理内存的空间。不需要 n 个拷贝，而只需一个拷贝。对每个用户，需要一个独立的段来保存局部变量。这些段当然是不能共享的。

也可以只共享部分程序。例如，公用的子程序包如果定义为共享的只读段，那么就可为多个用户所共享。例如，两个 For tran 程序可以使用共同的 *sqrt* 子程序，只要有一个 *sqrt* 子程序的拷贝就可以了。

尽管这种共享看起来简单，但是还是有一些微妙之处的。代码段通常会包括对自己的

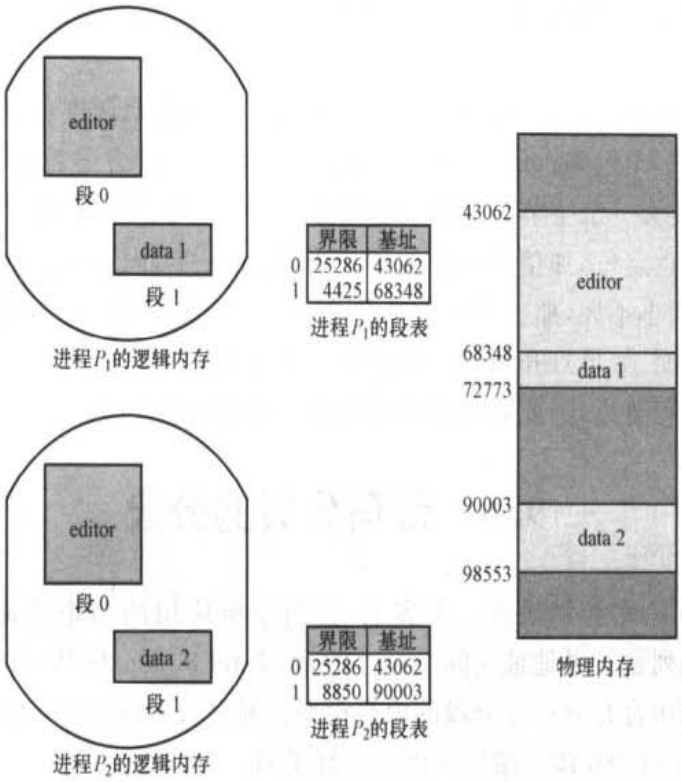


图 9.20　在一个分段内存系统中的段共享

引用。例如，条件转移有一个转移地址，该地址包括段号和偏移。转移地址的编号与代码段的段号一样。如果共享这个段，那么所有共享进程必须使得共享代码段有同样的段号。

例如，如果共享 *sqrt* 子程序，一个进程需要将它作为段 4，而另一个进程需要将它作为 17，那么 *sqrt* 子程序如何来引用自己呢？因为只有一个 *sqrt* 子程序的物理拷贝，对于两个用户来说该子程序必须用同样方法来引用自己：它必须有一个惟一的段号。随着共享段用户的增加，寻找一个可以接受的段号的难度会增加。

不包含物理指针的只读数据段可以用不同段号来共享，这同间接引用（而不是直接引用）自己的代码段一样。例如，条件分支是通过使用当前程序计数器指针的偏移或相对于包含当前段号的寄存器的偏移来作为转移地址，就可避免直接使用当前段号。

9.5.4　碎片

长期调度程序必须为程序的所有段寻找和分配空间。这与分页相似，只不过段是不同长度的，而页是等长的。因此，与不定长分区方案相似，内存分配是一个动态存储分配问题，这可采用最佳适应或首次适应算法。

当所有空闲内存块因太小而不能容纳一个段时，分段就会引起外部碎片。这时，进程可能需要等待直至有更多内存（或有一个大孔）为止，或通过合并来创建一个大孔。由于分段本质上是动态重定向算法，只要需要，就可合并内存。如果 CPU 调度程序因内存分配问题

而必须等待一个进程，那么这也可以（或不可以）查找 CPU 队列以便让更小、更低优先级的进程执行。

对于采用分段方案，外部碎片有多严重？带有合并的长期调度会有用吗？答案取决于平均段的大小。在一个极端，可以定义一个进程为一段。这种方案就成为不定长分区方案。在另一极端，可以定义一个字节为一段。这种方案取消了外部碎片，每个字节都需要一个基地址寄存器来进行重定位，加倍了内存使用。当然，下一步，固定大小的小段就是分页。通常，如果平均段的大小不大，那么外部碎片也会不大。（例如，设想一下将箱子放到车内，它们可能放不进。然而，如果打开箱子而将单个物品放入车内，可能所有的东西都能放进。）由于每个段比整个进程要小，因此它们更有可能装入现有的内存块。

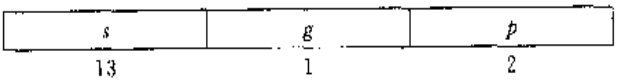
9.6 带有分页的分段

分页和分段各有优点和缺点。事实上，在当今所使用的两个最流行的微处理器中，Motorola 68000 系列是按平地址空间来设计的，而 Intel 80x86 和 Pentium 系列是按分段来设计的。两个都采用合并分页和分段的内存模型。通过合并这两个方法可以取长补短。这种合并可以通过 Intel 386 体系结构来得到最好说明。

IBM OS/2 32 位版本是支持在 Intel 386（和后来）体系结构上的操作系统。Intel 386 采用的内存管理为分段加分页。每个进程的段数的最大值为 16 KB，且每个段最大为 4 GB。页的大小为 4 KB。这里，没有时间完整地描述 386 的内存管理结构，而只能介绍一些主要思想。

进程的逻辑地址空间分为两个部分。第一个部分最多可以由 8 KB 段组成，这部分为私有。第二个部分最多可以由 8 KB 段组成，这部分为所有进程所共享。关于第一个分区的信息保存在**本地描述符表**（LDT）中，而关于第二个分区的信息保存在**全局描述表**（GDT）中。LDT 和 GDT 的每个条目由 8 个字节组成，这包括了一个段的详细信息如基位置和段长等。

逻辑地址是一对（选择器，偏移），选择器是一个 16 位的数：



其中，s 表示段号，g 表示段是在 GDT 还是在 LDT 中，p 表示保护信息。偏移为 32 bit 的数，用来表示字节在段内的位置。

机器有 6 个段寄存器，允许一个进程可以同时访问 6 个段。它有 6 个 8 字节微程序寄存器用来保存相应的描述符。这一缓冲允许 386 不必在每次内存引用时都从内存中读取描述符。

386 的物理地址为 32 bit 长，按如下方式来形成。段寄存器指向 LDT 或 GDT 中的适当条目。段的 0 基地址和界限信息用来产生**线性地址**。首先，界限用来检查地址的合法性。如果地址无效，就产生内存出错，导致陷入操作系统。如果有效，偏移值就与基地址的值相加，产生 32 bit 的线性地址。该地址再转换成物理地址。

如上面所指出的，每个段是分页的，每页长度为 4 KB。因此页表可以有多达一百万个

条目。因为每个条目需要 4 B，所以每个进程可能需要多达 4 MB 的物理地址空间来保存页表。显然，人们不想连续地分配该内存表。386 所使用的解决方法是采用两级分页机制。线性地址由 20 bit 的页号和 12 bit 的页偏移组成。由于对页表进行分页，页表被进一步分为 10 bit 页目录指针和 10 bit 的页表指针。逻辑地址如下：

页号		页偏移
p_1	p_2	d
10	10	12

这种体系结构的地址转换方案类似于图 9.13。Intel 地址转换的详细情况如图 9.21 所示。为了提高物理内存使用的效率，Intel 386 的页表可以交换到磁盘上。这时，页目录条目的无效位可用来表示相应的页表是在内存还是在磁盘上。如果在磁盘上，操作系统可以使用其他的 31 位来表示页表在磁盘上的位置，该页表可根据需要调入内存。

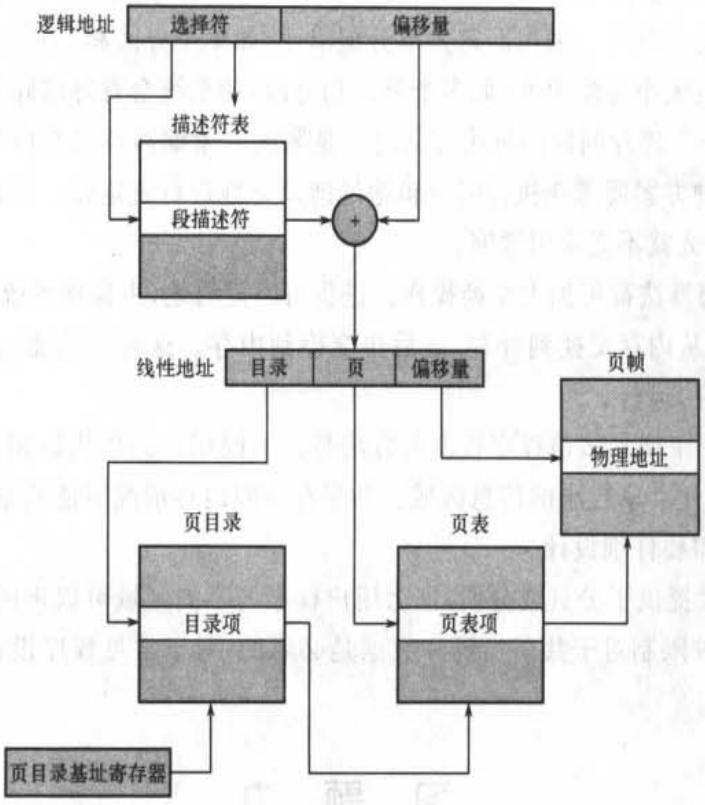


图 9.21　Intel 80386 的地址转换

9.7 小　结

用于多道程序设计的操作系统的内存管理算法，包括从简单的单用户系统的方法到分段加分页的方法。一个特定系统所采用方法的最大决定因素是它所能提供的硬件支持。CPU 所产生的每个内存地址都必须先进行合法性检查，才可映射成为物理地址。检查不能

通过软件来(有效地)实现。因此,受到可用硬件的约束。

本章所讨论的内存管理算法(连续分配、分页、分段及分段加分页)在许多方面都呈现出不同的特点。在比较不同内存管理策略时,需要考虑如下几点:

• **硬件支持**:对单分区和多分区方案,只需要一个基址寄存器或一对基址和界限寄存器就够了;而对于分页和分段,需要映射表以定义地址映射。

• **性能**:随着内存管理算法变得更复杂,逻辑地址到物理地址的映射所需要的时间也更长。对于简单系统,只需要对逻辑地址进行比较和加、减操作(这些较为简单)。如果表能通过快速寄存器来加以实现,那么分页和分段操作也会很快。不过,如果表在内存中,那么用户内存访问就大大地降级。TLB可以用来改善性能。

• **碎片**:如果多道程序的级别很高,那么多道程序系统通常会更有效地执行。对于给定的一组进程,可以通过将更多进程装入内存以增加多道程序的程度。为了完成这一任务,必须降低内存浪费或碎片。采用固定大小分配单元(如单个分区和分页)的系统会有内部碎片问题。采用可变大小分配单元(如多个分区和分段)的系统会有外部碎片问题。

• **重定位**:外部碎片问题的解决方案之一是紧缩。紧缩涉及到在内存中移动程序而不影响到程序。这种方案要求在执行时逻辑地址能动态地进行重定位。如果地址只能在装入时进行重定位,那么就不能采用紧缩。

• **交换**:任何算法都可加上交换操作。进程可以定时地(由操作系统来定,通常由CPU调度策略来决定)从内存交换到外存,之后再交换到内存。这种方法能允许更多进程(它们不能同时装入内存)运行。

• **共享**:另一个增加多道程序程度的方法是让不同用户共享代码和数据。共享通常要求分页或分段,以便共享较小的信息区域。共享在有限内存情况下能共享许多进程,但是共享的程序和数据需要仔细设计。

• **保护**:如果提供了分页或分段,那么用户程序的不同区域可以声明为只可执行、只读或可读可写。这种限制对于共享代码和数据是必要的,对于常见程序设计错误能提供简单的运行时的检查。

习 题 九

9.1 指出逻辑地址和物理地址的两个不同点。

9.2 说明内部碎片与外部碎片的区别。

9.3 描述下列分配算法:

a. 首次适应

b. 最佳适应

c. 最差适应

9.4 当一个进程被换出内存,它将失去使用CPU的能力(至少持续一段时间)。描述另一种进程失去

使用CPU的能力的情形，但在这种情形下进程没有被交换出去。

9.5 如果有内存划分100 KB、500 KB、200 KB、300 KB和600 KB(按顺序)，首次适应、最佳适应与最差适应算法各自将怎样放置大小分别为212 KB、417 KB、112 KB和426 KB(按顺序)的进程？哪一种算法的内存利用率最高？

9.6 假设有一个系统，它的程序可分为两部分：代码和数据。CPU知道它需要指令(取指令)或是数据(取或存数据)。为此，提供了两个基址限制寄存器对：一对用于指令，一对用于数据。指令基址限制寄存器对自动被设置为只读，所以不同的用户可以共享程序。讨论这种方案的优点和缺点。

9.7 为什么页面的大小总是2的幂。

9.8 假设一个有8个1 024字页面的逻辑地址空间，映射到一个有32帧的物理内存。

a. 逻辑地址有多少位？

b. 物理地址有多少位？

9.9 在一个分页系统中，进程不能访问不属于它的内存，为什么？操作系统怎样才能允许访问其他的内存？是否应该，为什么？

9.10 假设一个将页表存放在内存的分页系统。

a. 如果一次内存访问用200 ns，访问一页内存需用多少时间？

b. 如果加入TLB，并且75%的页表引用发生在TLB，内存有效访问时间是多少？(假设在TLB中寻找页表项占用零时间，如果页表项在其中)。

9.11 允许页表中的两个页表项指向内存中的同一个页帧的后果是什么？说明你如何利用这种结果来减少在内存中从一个地方拷贝大量内存到另一个地方的时间。在一个页中更新一些字节会在另一页产生什么后果？

9.12 为什么时常将分段与分页在同一个方案中结合使用？

9.13 描述一种同一个段可以属于两个不同进程的地址空间的机制。

9.14 说明为什么使用分段比纯使用分页更容易共享一个可重入模块。

9.15 在动态链接分段系统中，进程间共享段可以不必使用相同的段号。

a. 详细说明一个允许静态链接且共享段而不要求必须使用同样段号的系统。

b. 描述一个共享页面而不要求使用同样页号的分页方案。

9.16 假设有下面的段表：

段	基址	长度
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

下面逻辑地址的物理地址是多少？

a. 0430

b. 110

c. 2500

d. 3400

c. 4122

9.17 假设 Intel 的地址转换方案如图 9.21 所示。

a. 描述 Intel 80386 将逻辑地址转换成物理地址所采用的所有步骤。

b. 使用这样复杂的地址转换硬件对硬件操作系统有什么好处?

c. 这样的地址转换系统有没有什么缺点? 如果有,有哪些? 如果没有,为什么不是每个制造商都使用这种方案?

9.18 在 IBM/370 中,内存保护是通过关键字的使用来实现的。一个关键字是一个 4 bit 的数。每个 2 KB的内存块有一个与之相关联的关键字(存储关键字)。CPU 也有一个与之相关联的关键字(保护关键字)。一个存操作只有在两个关键字相同或其中一个为零时才允许进行。下面的哪种内存管理方案可以成功地在这种硬件上使用?

a. 裸机

b. 单用户系统

c. 固定进程数的多道程序

d. 可变进程数的多道程序

e. 分页

f. 分段

推荐读物

Knuth[1973](2.5 节)论述了动态存储分配。他还从模拟结果中发现首次适应分配算法比最佳适应算法好。Knuth[1973]讨论了"百分之五十"规则。

分页的概念要归功于 Atlas 系统的设计者们,Kilburn 等[1961]和 Howarth 等[1961]描述了这些。Dennis[1965]论述了分段的概念。GE 645 最先支持分页式分段,在它上面最初实现 MULTICS(Organick[1972])。

Chang 和 Mergen[1988]在关于 IBM RT 的存储管理器的论文中讨论了反向页表。

Jacob 和 Mudge[1997]里有关于地址转换的软件方法。

Smith[1982]描述和分析了高速缓存,包括相联内存。这篇文章中还有关于这个问题的其他参考书目。Hennessy 和 Patterson[1996]讨论了 TLB、cache 和 MMU 的硬件特性。Talluri 等[1995]讨论了 64 位地址空间的页表。Dougan 等[1999]讨论了一种管理 TLB 的技巧。

Motorola[1989a]里描述 Motorola 68000 微处理器家族。Intel[1986]里有关于 80386 的分页硬件信息。Tanenbaum[2001]也讨论了 Intel 80386 的分页。Intel 的出版物 Intel[1989]中也描述了 Intel 80486 的硬件。

Jacob 和 Mudge[1998a]描述了多种体系结构的内存管理,如 Pentium Ⅱ、PowerPC 和 UltraSPARC。

第十章　虚拟内存

在第九章，讨论了计算机系统所使用的各种内存管理策略。所有这些策略都有同样的目的：同时将多个进程保存在内存中以便允许多道程序设计。不过，这些策略都要求在进程执行之前必须的将这个进程放入内存之中。

虚拟内存技术允许进程的执行不必完全在内存中。这种方案的一个很大的优点就是程序可以比物理内存大。而且，虚拟内存将内存抽象成一个巨大的、统一的存储数组，进而将用户看到的逻辑内存与物理内存分开。这种技术允许程序员不受内存存储的限制。虚拟内存也允许进程很容易地共享文件和地址空间，还为创建进程提供了有效的机制。

但是，虚拟内存的实现并不容易，如果使用不当可能会大大地降低性能。本章通过按需分页来讨论虚拟内存，并研究其复杂性和开销。

10.1　背　　景

第九章所介绍的内存管理算法都是基于一个基本要求：执行指令必须在物理内存中。满足这一要求的第一种方法是将整个进程放在内存中。覆盖和动态装入能帮助减轻这一限制，但是它们需要程序员特别小心并且需要一些额外的工作。这一限制似乎是必须的，但是遗憾的是，这使得程序的大小被限制在物理内存的大小之内。

事实上，对真正程序的研究会发现，在许多情况下并不需要将整个程序放到内存中。例如：

- 程序通常有处理异常错误条件的代码。由于这些错误即使有也是很少发生，所以这种代码几乎不执行。
- 数组、链表和表通常分配了比实际所需要更多的内存。声明一个有 100 ×100 个元素的数组，可能实际使用的只是 10×10 个元素。虽然汇编程序系统表可能有 3 000 个符号空间，但是程序平均可能用到的只有不到 200 个符号。
- 程序的某些选项或特点可能很少使用。例如，美国政府计算机上的用于计算预算的子程序只是最近才使用。

即使在需要完整程序时，并不是同时都需要所有的程序（例如，与覆盖相似的情况）。

能够执行只有部分在内存中的程序会有很多好处。

- 程序不再受现有的物理内存空间限制。用户可以为一个巨大的**虚拟地址空间**

(virtual-address space)写程序,简化了编程操作。

• 因为每个用户程序使用了更少的物理内存,所以更多的程序可以同时执行,CPU 使用率也相应地增加,而响应时间或周转时间并不增加。

• 由于装入或交换每个用户程序到内存中所需的 I/O 会更少,用户程序会运行得更快。

因此,运行一个部分在内存中的程序不但有利于系统也有利于用户。

虚拟内存(virtual memory)将用户逻辑内存与物理内存分开。这在现有物理内存有限的情况下,为程序员提供了巨大的虚拟内存(图 10.1)。虚拟内存使编程更加容易,因为程序员不再需要担心有限的物理内存空间或究竟哪些代码需要覆盖;他只需要关注所要解决的问题。采用虚拟内存的系统几乎用不到覆盖。

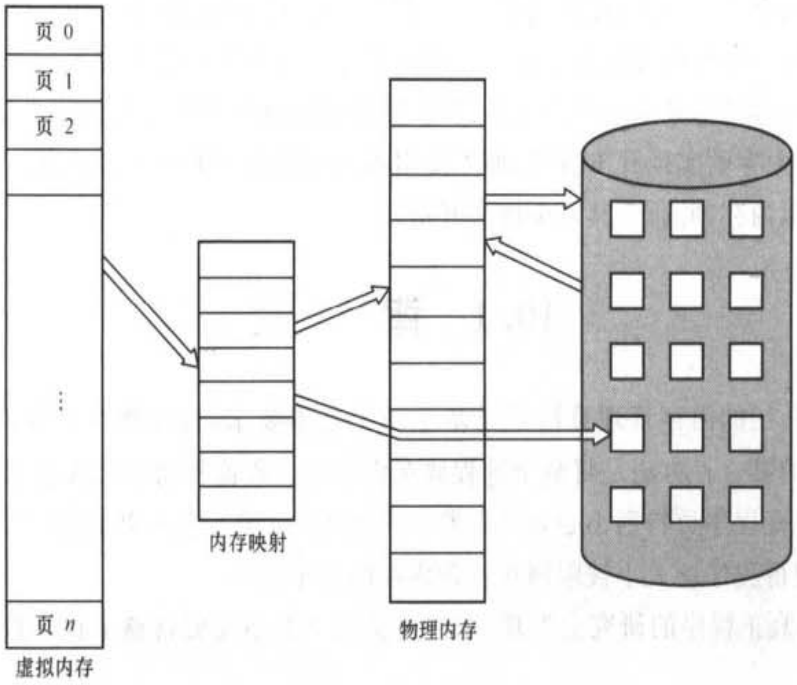


图 10.1 虚拟内存大于物理内存的示意图

除了将逻辑内存与物理内存分开,虚拟内存也允许文件和内存通过共享页而为多个进程所共享。页共享进一步允许在创建进程时的性能改善。

虚拟内存通常采用**请求页面调度**(demand paging)来实现。它也可以在分段系统上实现。多个系统采用了分段加分页的方案,这里段被进一步分页。因此,用户观点是分段,而操作系统可以通过请求页面调度实现这一观点。**请求分段调度**(demand segmentation)也可用来实现虚拟内存。Burroughs 计算机系统使用了请求分段调度。IBM 的 OS/2 操作系统也使用请求分段调度。不过,由于段是不定长的,段置换算法要比页置换算法复杂得多。本书不讨论请求分段调度;请参见相关的推荐读物。

10.2 请求页面调度

请求页面调度系统类似于分页系统加上交换(图 10.2)。进程驻留在次级存储器上(通常为磁盘)。当需要执行进程时,将它换入内存。不过,不是将整个进程换入内存,而是使用 lazy swapper。lazy swapper 只有在需要页时,才将它调入内存。由于将进程看做一系列的页,而不是一个大的连续空间,因此使用"交换"从技术上来讲并不正确。交换程序对整个进程进行操作,而调页程序只是对进程的单个页进行操作。因此,在讨论有关请求页面调度时,需要使用调页程序而不是交换程序。

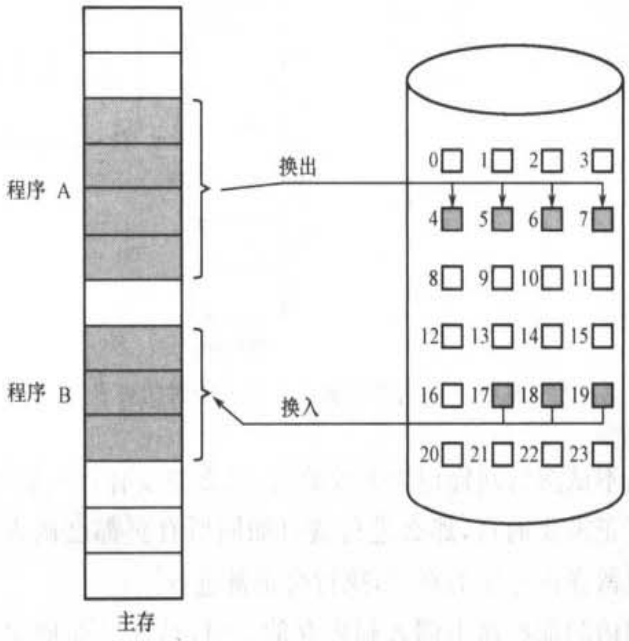


图 10.2 分页的内存与邻接的磁盘空间之间的传递

10.2.1 基本概念

当换入进程时,调页程序推测在该进程再次换出之前会用到哪些页。调页程序不是换入整个进程,而是把那些必需的页调入内存。这样,调页程序就避免了读入那些不使用的页,也减少了交换时间和所需的物理内存空间。

对这种方案,需要一定形式的硬件支持来区分哪些页在内存里,哪些页在磁盘上。在 9.4.4 小节中所描述的有效-无效位可以用于这一目的。不过,现在当该位设置为"有效"时,该值表示相关的页既合法且也在内存中。当该位设置为"无效"时,该值表示相关的页为无效(也就是,不在进程的逻辑地址空间内)或者有效但是在磁盘上。对于调入内存的页,其页表条目的设置与平常一样;但是对于不在内存的页,其页表条目设置为无效,或包含该页在磁盘上的地址。这种情况如图 10.3 所示。

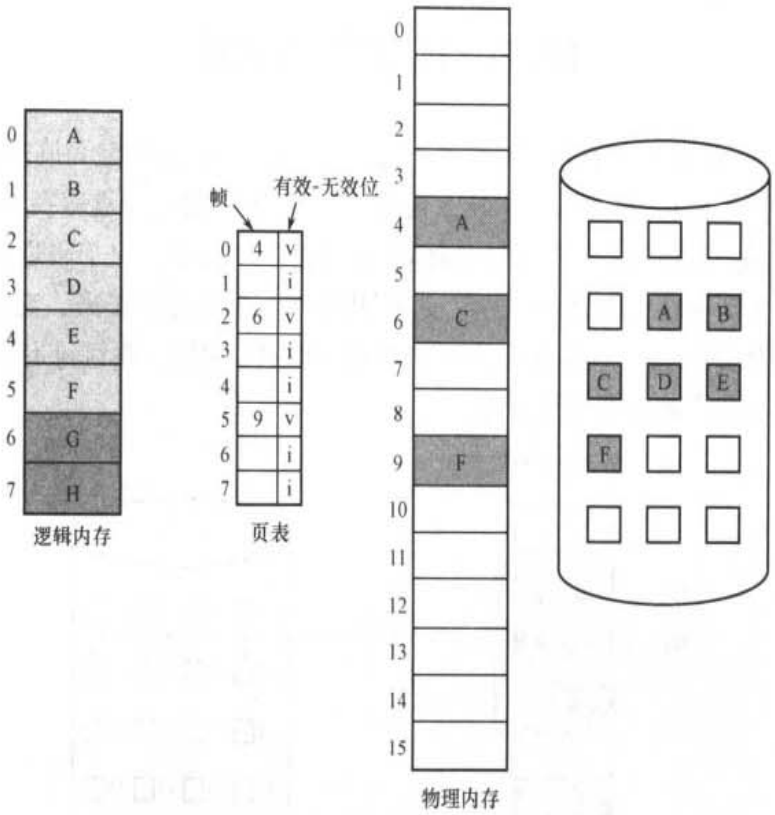


图 10.3 当有些页不在内存中时的页表

注意如果进程从不试图访问标记为无效的页，那么并没有什么影响。因此，如果推测正确并且只调入所有真正需要的页，那么进程就可如同所有页都已调入一样正常运行。当进程执行和访问那些驻留在内存中的页时，执行会正常进行。

但是当进程试图访问那些尚未调入到内存的页时，情况又如何呢？对标记为无效的访问会产生页**错误陷阱**(page-fault trap)。分页硬件，在通过页表转换地址时，会发现已设置了无效位，会陷入操作系统。这种陷阱是由于操作系统未能将所需的页调入内存(以试图减低磁盘传输开销和内存需求)，而不是由于试图使用非法内存地址而引起的无效地址错误(如不正确的数组下标)。必须纠正这一疏忽。处理这种页错误的程序比较简单(图 10.4)：

1. 检查进程的页表(通常与 PCB 一起保存)，以确定该引用是合法还是非法的地址访问。
2. 如果引用非法，那么终止进程。如果引用有效但是尚未调入页面，那么现在应调入。
3. 找到一个空闲帧(例如，从空闲帧链表上取一个)。
4. 调度一个磁盘操作，以便将所需要的页调入刚分配的帧。
5. 当磁盘读操作完成后，修改进程的内部表和页表，以表示该页已在内存中。
6. 重新开始因非法地址陷阱而中断的指令。进程现在能访问所需的页，就好像它似乎总在内存中。

要注意由于出现页错误时，保存了中断进程的状态(寄存器、条件代码、指令计数)，因此

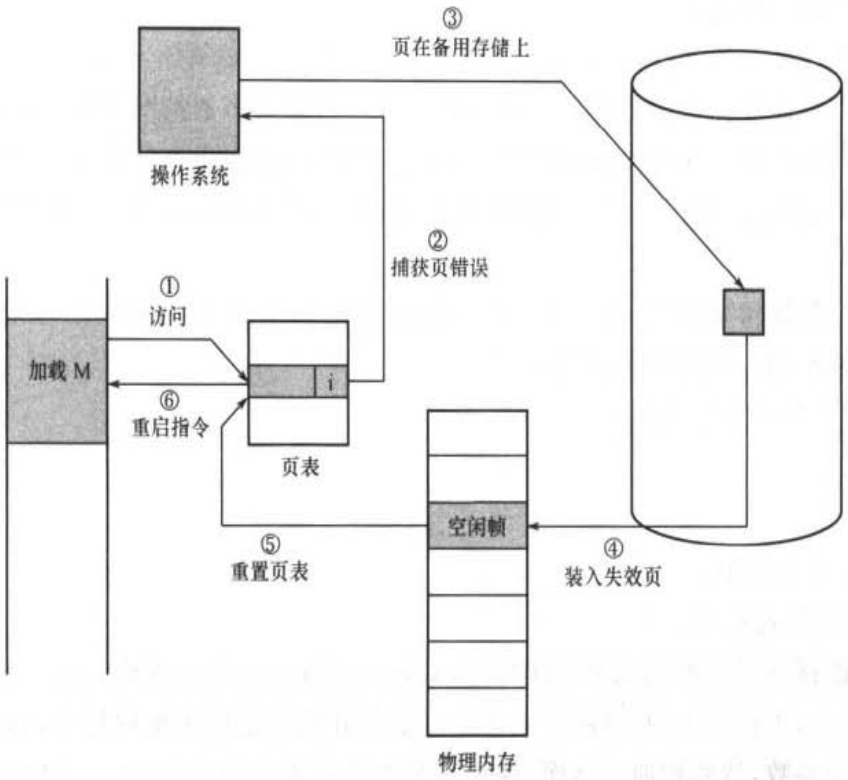


图 10.4 处理页错误的步骤

能按完全相同的位置和地址重新开始执行进程，只不过现在所需要的页已在内存中且可以访问。按这样的方式，虽然进程的有些部分不在内存中，但是仍能执行进程。当进程试图访问不在内存中的位置时，硬件陷入操作系统(页错误)。操作系统读入所需的页到内存中，并重新启动进程，就好像该页已在内存中。

一种极端情况是所有的页都不在内存中，这时就开始执行进程。当操作系统将指令指针指向进程的第一条指令时，由于所在的页并不在内存中，进程立即出现页错误。当页调入内存时，进程继续执行，并不断地出现页错误直到所有所需的页均在内存中。这时，进程可以执行且不出现页错误。这种方案称为**纯粹请求页面调度**(pure demand paging)：只有在需要时才将页调入内存。

从理论上来说，有的程序的单个指令可能访问多个页的内存(一页用于指令，许多页用于数据)，从而一个指令可能产生多个页错误。这种情况会产生令人无法接受的系统性能。幸运的是，对运行进程的分析说明了这种情况是极为少见的。如 10.6.1 小节所述，程序具有**引用的局部性**(locality of reference)，这使得请求页面调度性能较为合理。

支持请求页面调度的硬件与分页和交换的硬件一样：

- **页表**：该表能够通过有效-无效位或保护位的特定值，将条目设为无效。
- **次级存储器**：该次级存储器用来保存不在内存中的页。次级存储器通常是快速磁盘。它通常称为交换设备，用于交换的这部分磁盘称为**交换空间**(swap space)。交换空间的

分配将在第十四章中讨论。

除了这些硬件支持之外，还需要相当多的软件。同时还必须加入额外的体系结构限制。关键一点是需要能够在页面错误后重新运行指令。对绝大多数情况来说，这种要求容易满足。页错误可能出现在任何内存引用中。如果页错误出现在指令获取时，那么可以再次获取指令。如果页错误出现在参数获取时，那么可以再次获取指令，再次译码指令，然后再次获取参数。

作为一个最坏情况的例子，考虑一个三地址的指令，如将 A 和 B 的内容相加，并将结果放在 C 中。执行这一指令的步骤如下：

1. 获取并译码指令(ADD)。
2. 获取 A。
3. 获取 B。
4. 将 A 和 B 相加。
5. 将结果存入 C 中。

如果在保存到 C 中时出现页错误(因为 C 现在不在内存中的页内)，那么必须得到所需的页，将它调入，更正页表，并重新开始指令。重新开始需要再次获取指令，再次译码指令，再次获取两个参数，然后相加。然而，这种重复工作并不会很多(小于一个完整指令)，这种重复只有在出现页错误时才是必须的。

当一个指令可能改变多个不同位置时，会出现严重困难。例如，考虑一个 IBM 系统 360/370 的 MVC 指令(移动字节)，该指令能够将多达 256 B 的块从一处移到另一处(可能重复)。如果任何一块(源或目的)跨越页边界，在移动执行了部分时可能会出现页错误。另外，如果源和目的有交叠，源块可能已经修改，这时并不能简单地再次执行该指令。

这个问题有两种不同的解决方法。一种方案是微码计算并试图访问两块的两端。如果可能出现页错误，那么在这一步出现(在修改之前)。如果知道没有页错误(由于所有页都在内存中)，那么就可以执行移动。另一方案是使用临时寄存器来保存重写位置的值。如果有页错误，那么所有原来的值可在出现页错误之前写回到内存。这一动作在指令开始之前将内存恢复到原来的状态，这样指令就能够重复。

有的机器使用特别地址访问方式，包括自动减少和自动增加方式(例如 PDP-11)，这类机器也有类似的体系结构问题。这些地址访问方式使用一个寄存器作为指针，自动减少或增长指定寄存器的值。自动减少方式在使用寄存器的内容作为操作数地址*前*，自动减少它的值；自动增长方式在使用寄存器的内容作为操作数地址*后*，自动增加它的值。这样，指令

MOV $(R_2)+$, $-(R_3)$

把由寄存器 R_2 指向的地址的内容复制到了由寄存器 R_3 指向的地址。寄存器 R_2 在它作为指针使用后增加了(对一个字来说，是增加了 2，因为 PDP-11 是一个按字节寻址的计算机)；寄存器 R_3 在它作为指针使用后减少了(2)。现在来考虑一下如果在试图存储到由寄存

器 R_3 指向的地址时得到一个错误，那么会发生些什么。为了重新开始指令，必须用开始执行指令前的它们的值来重置这两个寄存器的值。一个解决方法是创建一个特殊状态寄存器来记录寄存器号和在指令执行过程中有变化的寄存器的修改数量。这个状态寄存器允许操作系统撤消引起页错误的执行了一部分的指令的结果。

这些决不是在向一个已有体系中增加分页以允许请求页面调度时产生的仅有体系问题，因为它们还体现了一些困难。分页是加在计算机系统的 CPU 和内存之间的。它应该对用户进程完全透明。这样，人们就假定分页能够应用到任何系统中。虽然这个假定对有一个页错误就代表一个致命错误的非请求页面调度环境来说是正确的，但是对于页错误仅意味着另外一个额外的页需调入内存后进程重新运行的情况来说是不正确的。

10.2.2 请求页面调度的性能

请求页面调度对计算机系统的性能有重要影响。为了说明起见，下面计算一下关于请求页面调度内存的**有效访问时间**。对绝大多数计算机系统而言，内存访问时间(用 ma 表示)的范围为 10 ns 到 200 ns。只要没有出现页错误，那么有效访问时间等于内存访问时间。然而，如果出现页错误，那么就必须先从磁盘中读入相关页，再访问所需要的字。

设 p 为页错误的概率($0 \leqslant p \leqslant 1$)。希望 p 接近于 0；即页错误很少。那么有效访问时间为：

有效访问时间 $= (1-p) \times$ ma $+ p \times$ 页错误时间

为了计算有效访问时间，必须知道处理页错误需要多少时间。页错误会引起如下顺序的动作产生：

1. 陷入到操作系统。
2. 保存用户寄存器和进程状态。
3. 确定中断是否为页错误。
4. 检查页引用是否合法并确定页所在磁盘的位置。
5. 从磁盘读入页到空闲帧中。
   a. 在该磁盘队列中等待，直到处理读请求。
   b. 等待磁盘的寻道和或延迟时间。
   c. 开始将磁盘的页传到空闲帧。
6. 在等待时，将 CPU 分配给其他用户(CPU 调度，可选)。
7. 磁盘中断(以示 I/O 完成)。
8. 保存其他用户的寄存器和进程状态(如果执行了第 6 步)。
9. 确定中断是否来自磁盘。
10. 修正页表和其他表以表示所需页现已在内存中。
11. 等待 CPU 再次分配给本进程。

12. 恢复用户寄存器、进程状态和新页表，再重新执行中断的指令。

以上步骤并非在所有情况下都是必需的。例如，假设在第 6 步，在执行 I/O 时，CPU 分配给另个一个进程。这种安排允许多道程序设计以提高 CPU 使用，但是在执行完 I/O 时也需要额外时间来重新启动页错误处理程序。

无论如何，都有如下三个主要的页错误处理时间：

1. 处理页错误中断。

2. 读入页。

3. 重新启动进程。

第 1 和第 3 个任务可以降低，如编码仔细，可只有数百条指令。这些任务每次可能只有 1 μs 到 100 μs。另一方面，页切换时间可能接近 24 ms。一个典型硬盘的寻道时间为 15 ms，延迟时间为 8 ms，传输时间为 1 ms。因此，总的页错误处理时间可能接近 25 ms，包括硬件和软件时间。而且，要注意这里只考虑了设备处理时间。如果有一个队列的进程在等待设备（其他进程也引起了页错误），那么必须加上等待设备的时间，这又增加了页错误处理时间。

如果设平均页错误处理为 25 ms，内存访问时间为 100 ns，那么有效内存访问时间（以纳秒计）为

$$\begin{aligned}\text{有效访问时间} &= (1-p)\times 100 + p\times 25\ ms\\ &= (1-p)\times 100 + p\times 25\,000\,000\\ &= 100 + 24\,999\,900\,p\end{aligned}$$

由此可以看出有效访问时间与**页错误率**直接有关。如果每 1 000 次访问中有一个页错误，那么有效访问时间为 25 μs。即，计算机会因为采用请求页面调度，而慢 250 倍。如果需要性能降低不超过 10%，那么需要

$$110 > 100 + 25\,000\,000\times p$$

$$10 > 25\,000\,000\times p$$

$$p < 0.000\,000\,4$$

即，为了让因页错误而出现的性能降低可以被接受，那么只能允许每 2 500 000 次访问中出现不到一次的页错误。

对于请求页面调度，降低页错误率是非常重要的。否则，有效访问时间会增加，会显著地降低进程执行。

请求页面调度的另一个重要方面是交换空间的处理和使用。磁盘 I/O 到交换空间通常要比到文件系统要快。这是因为交换空间是按大块来分配的，并不使用文件查找和间接分配方法（第十四章）。因此，如果在进程开始时将整个文件映像复制到交换空间，并从交换空间执行按页调度，那么有可能获得更好的调页效果。另一选择是开始时从文件系统中进行请求页面调度，但是当出现页置换时则将页写入交换空间，这种方法确保只有所需的页才从

文件系统中调入，而以后出现的调页是从交换空间中读入的。

有的系统在使用二进制文件时，试图限制交换空间的使用。对这些文件的请求页面调度是直接从文件系统中进行的。当出现页置换时，这些页只是被重写（因为它们没有被修改）；当再次需要时，再直接从文件系统中读入。采用这种方法，文件系统本身就是备份仓库。然而，对那些与文件无关的页还是需要使用交换空间；这些页包括进程的**堆栈**（stack）和**堆**（heap）。这种技术为许多系统如 Solaris 2 所使用。这种方法看来是个较好的折中；它也用于 BSD UNIX。

10.3 进程创建

在 10.2 节，描述了一个进程如何采用请求页面调度来开始执行。采用这种技术，进程只需要调入包括第一条指令的页就能很快地开始。然而，请求页面调度和虚拟内存也能在进程创建时，提供其他好处。这里，探讨由虚拟内存而带来的两种用于提高进程创建和运行性能的技术。

10.3.1 写时拷贝

当从磁盘中读入文件到内存时，可使用请求页面调度，这些文件可能包括二进制可执行文件。然而，通过采用类似页面共享的技术（如 9.4.5 小节所述），采用系统调用 fork 创建进程的开始阶段可能不需要请求页面调度。这种技术提供了快速进程创建，且使得为新创建进程而必须分配的新页面数量为最小。

回想一下系统调用 fork 是将子进程创建为父进程的复制品。传统地，fork 为子进程创建一个父进程地址空间的拷贝。然而，由于许多子进程在创建之后通常马上会执行系统调用 exec，所以父进程地址空间的复制可能没有必要。因此，可使用一种称为写时复制（copy-on-write）的技术。这种方法允许父进程与子进程开始时共享同一页面。这些页面标记为**写时复制**，即如果任何一个进程需要对页进行写操作，那么就创建一个共享页的拷贝。例如，假设子进程试图修改含有部分栈的页，且操作系统能识别出该页为写时复制页，那么操作系统就会创建该页的一个复制，并将它映射到子进程的地址空间内。这样，子进程会修改其复制的页，而不是父进程的页。采用写时复制技术，很显然只有被进程所修改的页才会复制；所有非修改页可为父进程和子进程所共享。注意只有可能修改的页才需要标记为写时复制。不能修改的页（即包含可执行代码的页）可以为父进程和子进程所共享。写时复制是一种常用技术，为许多操作系统用于复制进程，如 Windows 2000、Linux 和 Solaris 2。

当确定一个页要采用写时复制时，从哪里分配空闲页是很重要的。许多操作系统为这类请求提供了空闲缓冲**池**（pool）。这些空闲页在进程栈或堆必须扩展时可用于分配，或用于管理写时复制页。操作系统通常采用**按需填零**（zero-fill-on-demand）的技术以分配这些

页。按需填零页在需要分配之前先填零，因此清除了以前的页内容。对于写时复制，要复制的页会被复制到已填零的页上。为堆栈或堆所分配的页也类似分配到填零页。

许多 UNIX 版本(包括 Solaris 2)也提供了系统调用 fork 的变种：vfork(**虚拟内存**(virtual memory) fork)。vfork 的操作不同于写时复制的 fork。对于 vfork，父进程挂起，子进程使用父进程的地址空间。由于 vfork 不使用写时复制，因此如果子进程修改父进程地址空间的任何页，那么这些修改过的页在父进程重启时是可见的。所以，vfork 必须小心使用，以确保子进程不修改父进程的地址空间。vfork 主要用于在进程创建后立即调用 exec 的情况。由于没有出现复制页面，vfork 是一种非常有效的进程创建方法，有时用于实现 UNIX 命令行 shell 的接口。

10.3.2 内存映射文件

考虑一下采用标准系统调用 open、read 和 write 并在磁盘上对文件进行一系列的读操作。文件每次访问时都需要一个系统调用和磁盘访问。另外一种方法是，可使用所讨论的虚拟内存技术来将文件 I/O 作为普通内存访问。这种方法称为文件的**内存映射**(memory mapping)，它允许一部分虚拟内存与文件逻辑相关联。文件的内存映射可将一磁盘块映射成内存的一页(或多页)。开始的文件访问按普通请求页面调度来进行，会产生页错误。这样，一页大小的部分文件从文件系统读入物理页(有的系统会一次读入多个页面大小的内容)。以后文件的读、写就按通常的内存访问来处理，从而由于是通过内存的文件操作而不是使用系统调用 read 和 write，简化了文件访问和使用。注意对映射到内存中的文件进行写可能并不会立即写到磁盘上的文件。有的操作系统定期检查文件的内存映射页是否改变，以选择是否更新到物理文件。关闭文件会导致内存映射的数据写回到磁盘，并从进程的虚拟内存中删除。

有的操作系统只能通过特定的系统调用提供内存映射，而通过标准的系统调用处理所有其他文件 I/O。然而，有的系统不管文件是否说明为内存映射，都选择对文件进行内存映射。下面通过 Solaris 2 为例来说明。如果一个文件说明为内存映射(采用系统调用 mmap())，那么 Solaris 2 将该文件映射到进程的地址空间中。然而，如果一个文件采用普通系统调用如 open、read 和 write 来用于打开和访问，那么 Solaris 2 仍然对文件进行内存映射，不过是将其映射到内核空间。无论文件是如何打开，Solaris 2 都将所有文件 I/O 作为内存映射，以允许文件访问在内存中进行。

多个进程可以允许将同一文件映射到各自的虚拟内存中，以允许数据共享。其中任一进程修改虚拟内存中的数据，都会为其他映射相同文件部分的进程所见。根据虚拟内存的相关知识，可以清楚地看到内存映射部分的共享是如何实现的：每个共享进程的虚拟内存表都指向物理内存的同一页，该页有磁盘块的复制。这种内存共享如图 10.5 所示。内存映射系统调用也能支持写时复制功能，允许进程共享只读模式的文件，但也有它们所修改数据的

各自拷贝。为了协调共享数据的访问，有关进程可使用第七章所述的互斥机制。

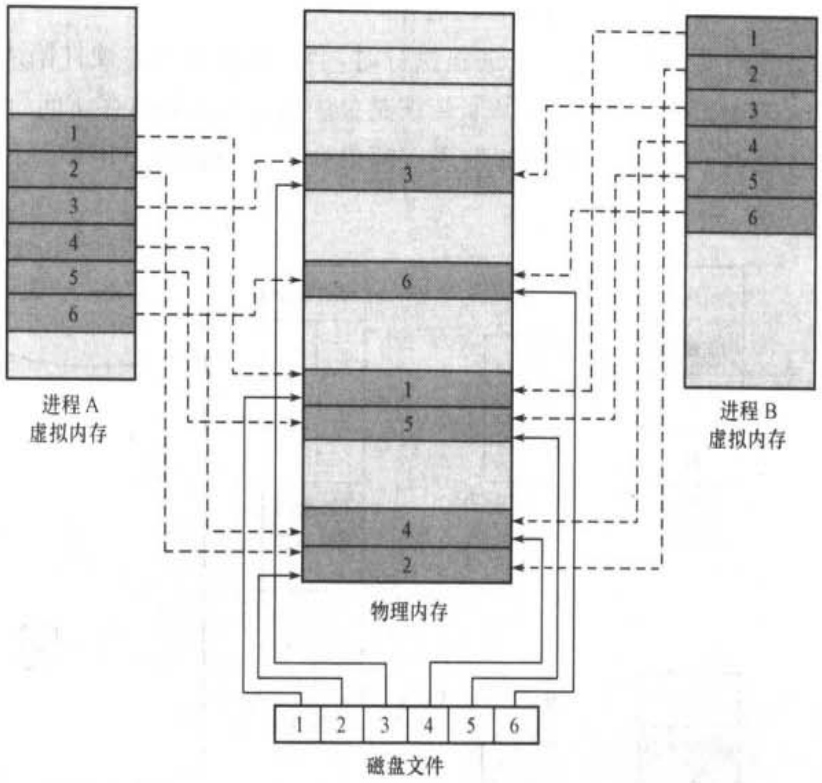


图 10.5 内存映射文件

10.4 页面置换

在迄今为止的讨论中，页错误并不是个严重问题，因为每页最多只会出现一次错误（即在其首次引用时）。这种描述并不严格准确。如果一个进程具有 10 页而事实上只使用其 5 页，那么请求页面调度就节省了用以装入（从不使用的）另 5 页所必需的 I/O。也可以通过运行两倍的进程以增加多道程序的级别。因此，如果有 40 帧，那么可以运行 8 个进程，而不是当每个进程都需要 10 帧（其中 5 个决不使用）时只能运行 4 个进程。

如果增加了多道程序的程度，那么会**过度分配**（over-allocating）帧。如果运行 6 个进程，且每个进程有 10 页大小但事实上只使用其中 5 页，那么就获得了更高的 CPU 利用率和吞吐量，且有 10 帧可用做备用。然而，有可能每个进程，对于特定数据集合，会突然试图使用其所有页，从而产生共需要 60 帧而只有 40 帧可用。虽然这种情况不太可能，但是随着增加多道程序的级别，平均内存使用接近可用的物理内存时，这种情况就很可能发生。（对于这个例子，为什么只运行 6 个进程而不运行 7 个或 8 个进程呢？）

再者，还需要考虑到内存不但用于保存程序页面，用于 I/O 的缓冲也需要使用大量的内存。这种使用会增加内存布置算法的压力。确定多少内存用于分配给 I/O 而多少内存分配

给程序页面是个棘手问题。有的系统为I/O缓冲分配了一定比例的内存；而其他系统允许用户进程和I/O子系统竞争使用所有系统内存。

过度分配会有问题。当一个用户进程执行时，会出现页错误。硬件陷入到操作系统。接着操作系统会检查其内部表以确定该页错误是合法还是非法的内存访问。进而操作系统会确定所需页在磁盘上的位置，但是却发现空帧闲列表上并没有空闲帧：所有帧都在使用(图10.6)。

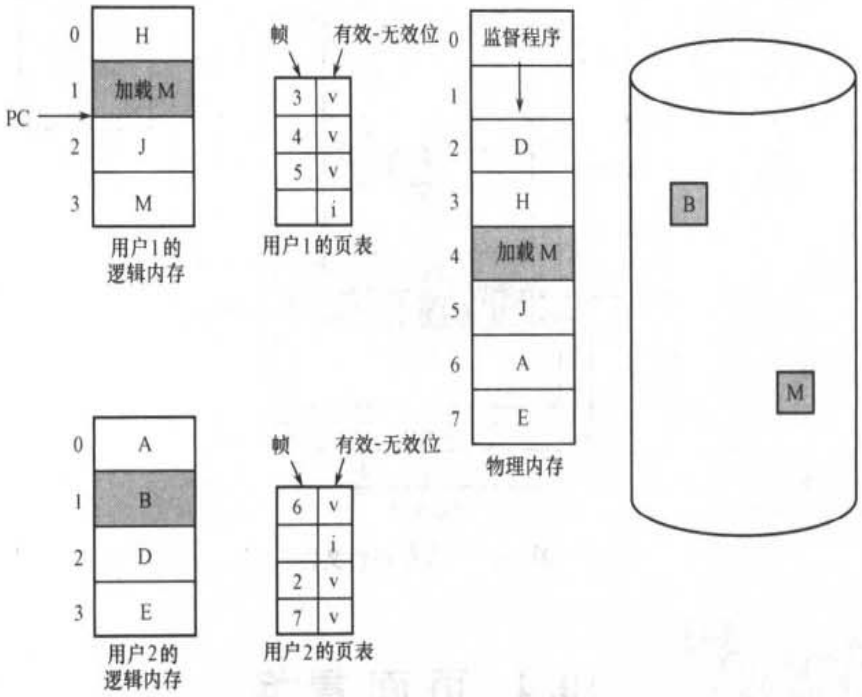


图10.6 需要页置换的情况

这时操作系统会有若干选择。它可以终止用户进程。然而，请求页面调度是操作系统试图改善计算机系统的使用率和吞吐量。用户并不关心其进程是否运行在调页系统上：调页对用户而言应是透明的。因此，这种选项并不是最佳选择。

操作系统也可以交换出一个进程，以释放其所有帧，并降低多道程序的级别。这种选择在有些环境下是好的；将在10.6节中加以讨论。现在，讨论一个更为复杂的可能：**页置换**(page replacement)。

10.4.1 基本方法

页置换采用如下方法。如果没有空闲帧，那么就查找当前不在使用的帧，并使之空闲。可以这样来释放一个帧：将其内容写到交换空间，并改变页表(和所有其他表)以表示该页不在内存中(图10.7)。现在可使用空闲帧来保存进程出错的帧。修改页错误处理程序以包括页置换：

1. 查找所需页在磁盘上的位置。

2. 查找一空闲帧：

a. 如果有空闲帧，那么就使用它。

b. 如果没有空闲帧，那么就使用页置换算法以选择一个“牺牲”帧(victim frame)。

c. 将“牺牲”帧的内容写到磁盘上；改变页表和帧表。

3. 将所需页读入(新)空闲帧；改变页表和帧表。

4. 重启用户进程。

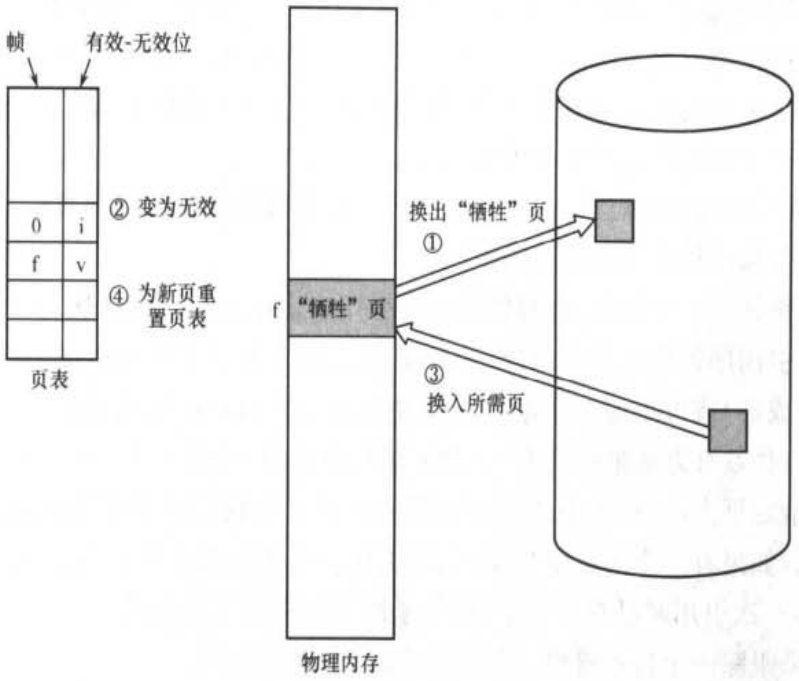


图 10.7 页置换

注意如果没有帧空闲，那么需要采用两个页传输(一个换出，一个换入)。这种情况实际上把页错误处理时间加倍了，而且也相应地增加了有效访问时间。

可通过**修改位**(modify bit)(或**脏位**(dirty bit))来降低额外开销。每页或帧可以有一个修改位通过硬件与之相关联。每当页内的任何字或字节被写入时，硬件就会设置该页的修改位以表示该页已修改。如果修改位已设置，那么就知道自从磁盘读入后该页已发生了修改。在这种情况下，必须要把该页写到磁盘上去。然而，如果修改位没有设置，那么就知道自从磁盘读入后该页并没有发生修改。因此，磁盘上页的拷贝的内容并没有必要(比如用其他页)重写，因此就避免了将内存页写回磁盘上：它已经在那里了。这种技术适用于只读页(例如，二进制代码的页)。这种页不能被修改；因此，如需要可以放弃这些页。这种方案可显著地降低用于处理页错误所需要的时间，因为如果页没有修改，它能降低一半的 I/O 时间。

页置换是请求页面调度的基础。它分开了逻辑内存与物理内存。采用这种机制，小的物理内存能为程序员提供巨大的虚拟内存。对于非请求页面调度，用户地址可映射到物理

地址，所以这两者可以不同。然而，所有进程的页仍然必须在物理内存中。对于请求页面调度，逻辑地址空间的大小不再受物理内存所限制。如果有一个具有 20 页的用户进程，那么可简单地通过请求页面调度用 10 个帧来执行它，如果有必要可以用置换算法来查找空闲帧。如果已修改的页需要被置换，那么其内容会被复制到磁盘上。后来对该页的引用会产生页错误。这时，该页可以再调回内存，有可能会置换进程的其他页。

为实现请求页面调度必须解决两个主要问题：必须开发**帧分配算法**(frame allocation algorithm)和**页置换算法**(page replacement algorithm)。如果在内存中有多个进程，那么必须决定为每个进程各分配多少内存。而且，当需要页置换时，必须选择要置换的帧。设计合适的算法以解决这些问题是个重要任务，因为磁盘 I/O 非常费时。即使请求页面调度方面的很小改进也会对系统性能产生显著的改善。

有许多不同的页置换算法。每个操作系统可能都有其自己的置换算法。如何选择一个置换算法？通常采用最小页错误率的算法。

可以这样来评估一个算法：针对特定内存引用串运行某个置换算法，并计算出页错误的数量。内存的引用序列称为**引用串**(reference string)。可以人工地生成内存引用串(例如，通过随机数生成器)或可跟踪一个给定系统并记录每个内存引用的地址。后一方法产生了大量数据(以每秒数百万地址的速度)。为了降低数据量，可利用以下两个因素。

第一，对给定页大小(页大小通常由硬件或系统来决定)，只需要考虑页码，而不需要完整地址。第二，如果有一个对页 p 的引用，那么任何紧跟着的对页 p 的引用决不会产生页错误。页 p 在第一次引用时已在内存中；任何紧跟着的引用不会出错。

例如，如果跟踪一个特定进程，那么可记录如下地址顺序：

0100，0432，0101，0612，0102，0103，0104，0101，0611，0102，0103，

0104，0101，0610，0102，0103，0104，0101，0609，0102，0105

如果页大小为 100 B，那么就得到如下引用串：

1，4，1，6，1，6，1，6，1，6，1

针对某一特定引用串和页置换算法，为了确定页错误的数量，还需要知道可用帧的数量。显然，随着可用帧数量的增加，则页错误的数量会相应地降少。例如，对于上面的引用串，如果有 3 个或更多的帧，那么只有三个页错误，对每个页的首次引用会产生错误。另一方面，如果只有一个可用帧，那么每个引用都要产生置换，共产生 11 个页错误。通常，人们期待着如图 10.8 所示的曲线。随着帧数量增加，那么页错误数量会降低至最小值。当然，增加物理内存就会增加帧的数量。

为了讨论页置换算法，将采用如下引用串：

7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1，7，0，1

而可用帧的数量为 3。

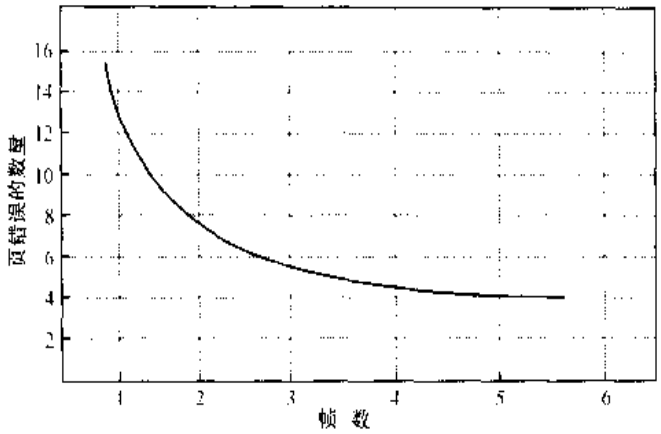


图 10.8 页错误与帧数量关系图

10.4.2 FIFO 页置换

最简单的页置换算法是 FIFO 算法。FIFO 页置换算法为每个页记录着该页调入内存的时间。当必须置换一页时，将选择最旧的页。注意并不需要记录调入一页的确切时间。可以创建一个 FIFO 队列来管理内存中的所有页。置换队列的首页。当需要调入页时，将它加到队列的尾部。

对于样例引用串，三个帧开始为空。开始的三个引用(7,0,1)会引起页错误，会调入这些空帧中。下一个引用(2)置换 7，这是因为页 7 最先调入。由于 0 是下一个引用，但已在内存中，所以对该引用不会出现页错误。对 3 的首次引用导致页 0 被替代，由于它是现在位于内存中的(0,1,2)中最先被调入的页。由于这一替代，下一个对 0 的引用会产生页错误。页 1 被页 0 所替代。该进程按图 10.9 所示的方式继续进行。每次有页错误时，都显示了哪些页在三个帧中。总共有 15 次帧错误。

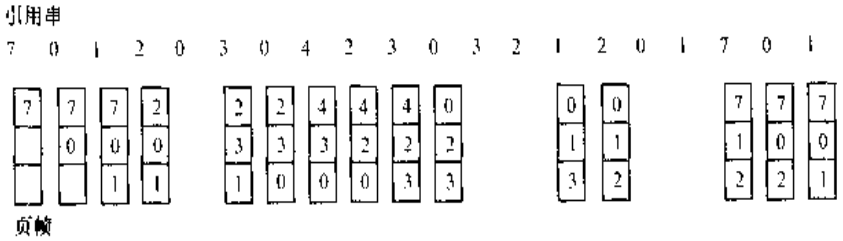


图 10.9 FIFO 页置换算法

FIFO 页置换算法容易理解和实现。但是，其性能并不总是很好。所替代的页可能是很久以前使用的、现已不再使用的初始化模块。另一方面，所替代的页可能包含一个以前初始化的并且不断使用的常用变量。

注意即使选择替代一个活动页，仍然会正常工作。当换出一个活动页以调入一个新页时，一个页错误几乎马上会要求换回活动页。这样某个页会被替代以将活动页调入内存。因此，

一个不好的替代选择增加了页错误频率且减慢了进程执行，但是并不会造成不正确执行。

为了说明与 FIFO 页置换算法相关的可能问题，考虑如下引用串：1,2,3,4,1,2,5,1,2,3,4,5。图 10.10 显示页错误对现有帧数的曲线。注意到对 4 帧的错误数(10)比对 3 帧的错误数(9)还要大。这种最为令人难以置信的结果称为 Belady **异常**(Belady's anomaly)：对有的页置换算法，页错误率可能会随着所分配的帧数的增加而增加。原期望为进程增加内存会改善其性能。在早期研究中，研究人员注意到这种推测并不总是正确的。因此，发现了 Belady 异常。

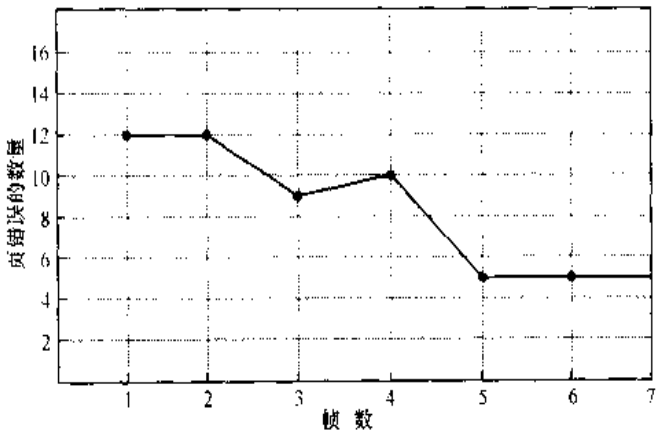


图 10.10 一个采用 FIFO 置换引用串的页错误曲线

10.4.3 最优页置换

Belady 异常发现的结果之一是对**最优页置换算法**(optimal page-replacement algorithm)的搜索。最优页置换算法是所有算法中产生页错误率最低的，且决没有 Belady 异常的问题。这种算法确实存在，它被称为 OPT 或 MIN。它是*置换那些在最长时间中不会被使用的页*。使用这种页置换算法确保对于给定数量的帧会产生最低可能的页错误率。

例如，针对引用串样例，最优页置换算法会产生 9 个页错误，如图 10.11 所示。前 3 个引用会产生错误以填满空闲帧。对页 2 的引用会置换页 7，这是因为页 7 直到第 18 次引用时才使用，而页 0 在第 5 次引用时使用，且页 1 在第 14 次引用时使用。对页 3 的引用，会置换页 1，因为页 1 是位于内存中的 3 个页中最迟引用的页。有 9 个页错误的最优页置换算法要好于有 15 个页错误的 FIFO 算法。(如果忽视前 3 页错误(所有算法均会有的)，那么最优置换要比 FIFO 置换好一倍。)事实上，没有置换算法能只用三个帧且少于 9 个页错误就能处理该引用串。

然而，最优页置换算法难于实现，因为需要引用串的未来知识。(在 6.3.2 小节讨论 SJF CPU 调度，碰到一个类似问题)。因此，最优算法主要用于比较研究。例如，如果知道一个算法不是最优，但是与最优相比最坏不差于 12.3%，平均不差于 4.7%，那么也是很有用的。

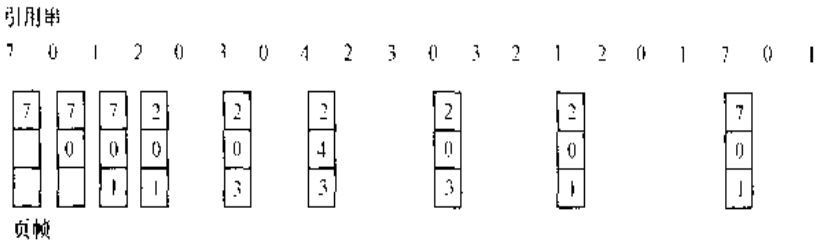


图 10.11 最优页置换算法

10.4.4 LRU 页置换

如果最优算法不可行，那么最优算法的近似算法或许成为可能。FIFO 和 OPT 算法（而不是向后看或向前看）的关键区别在于 FIFO 算法使用的是页调入内存的时间；OPT 算法使用的是页将要使用的时间。如果使用离过去最近作为不远将来的近似，那么可置换最长时间没有使用的页（图 10.12）。这种方法称为**最近最少使用算法**（least-recently-used，LRU）。

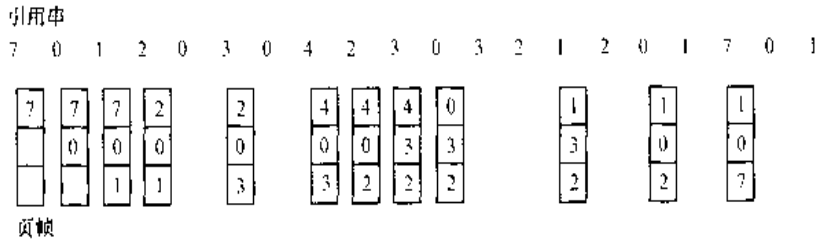


图 10.12 LRU 页置换算法

LRU 置换为每个页关联该页上次使用的时间。当必须置换一页时，LRU 选择最长时间没有使用的页。这种策略为向后看（而不是向前看）最优页置换算法。（奇怪的是，如果 S^R 表示引用串 S 的倒转，那么针对 S 的 OPT 算法的页出错率与针对 S^R 的 OPT 算法的页出错率是一样的。类似地，针对 S 的 LRU 算法的页出错率与针对 S^R 的 LRU 算法的页出错率是一样的。）

对于引用串样例，采用 LRU 算法的结果如图 10.12 所示。LRU 算法产生 12 次错误。注意，前 5 个错误与最优算法一样。然而，当出现对页 4 的引用时，LRU 算法知道页 2 最近最少使用。最近使用的页是页 0，其次是页 3。因此，LRU 算法置换页 2，并不知道页 2 马上要用。接着，当页 2 出错时，LRU 会置换页 3，这是因为位于内存的三个页{0,3,4}中，页 3 最近最少使用。虽然有些问题，有 12 个页错误的 LRU 算法仍然要比有 15 个页错误的 FIFO 置换要好。

LRU 策略经常用做页置换算法，且被认为很好。主要问题是如何实现 LRU 置换。LRU 页置换算法可能需要大量硬件支持。它的问题是为页帧确定一个排序序列，这个序列按页帧上次使用的时间来定义。有两种可行实现：

• **计数器**：最为简单的情况是，为每个页表项关联一个使用时间域，并为CPU增加一个逻辑时钟或计数器。对每次内存引用，计数器都会增加。每次内存引用时，时钟寄存器的内容会复制到相应页所对应页表项的使用时间域内。置换具有最小时间的页。这种方案需要搜索页表以查找LRU页，且每次内存访问都要写入内存（到页表的使用时间域）。在页表改变时（因CPU调度）也必须要保持时间。必须要考虑时钟溢出。

• **堆栈**：实现LRU置换的另一个方法是采用页码堆栈。每当引用一个页，该页就从堆栈中删除并放在顶部。这样，堆栈顶部总是最近使用的页，堆栈底部总是LRU页（图10.13）。由于必须从堆栈的中部删除项，所以该堆栈可实现为具有头指针和尾指针的双向链表。这样，删除一页并放在堆栈顶部最坏情况下只需要改变6个指针。虽说每个更新有些费时，但是置换不需要搜索；尾指针指向堆栈底部，就是LRU页。对于用软件或微代码的LRU置换的实现，这种方法十分合适。

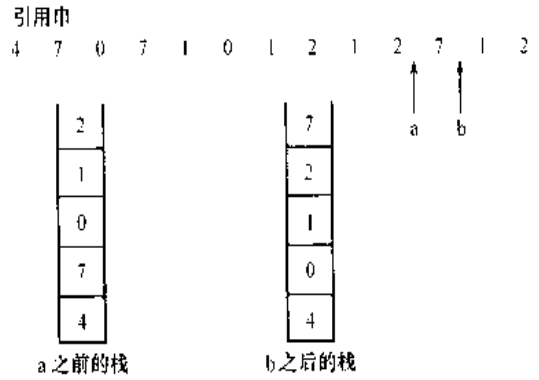


图10.13 用堆栈来记录最近使用的页

最优置换和LRU置换都没有Belady异常。有一类算法，称为**堆栈算法**（stack algorithm），都决不可能有Belady异常。堆栈算法可以证明为：对于帧数为n的内存页集合是对于帧数为$n+1$的内存页集合的子集。对于LRU算法，如果内存页的集合是最近引用的页，那么对于帧的增加，这些n页仍然是最近引用的页，所以也仍然在内存中。

注意，如果只有标准TLB寄存器而没有其他硬件支持，那么这两种LRU实现都是不可能的。每次内存引用，都必须更新时钟域或堆栈。如果对每次引用都采用中断，以允许软件更新这些数据结构，那么它会使内存引用慢至少10倍，进而使用户进程运行慢10倍。几乎没有系统可以容忍如此程度的内存管理的开销。

10.4.5 LRU近似页置换

很少有计算机系统能提供足够的硬件来支持真正LRU页置换算法。有的系统不提供任何支持，因此必须使用其他置换算法（如FIFO算法）。然而，许多系统都通过引用位方式提供一定的支持。页表内的每项都关联着一个**引用位**（reference bit）。每当引用一个页时（无论是对页的字节进行读或写），相应页的引用位就被硬件置位。

开始，操作系统会将所有引用位都清零。随着用户进程的执行，与引用页相关联的引用位被硬件置位（为1）。之后，通过检查引用位，能确定哪些页使用过而哪些页未使用过。虽然不知道顺序，但是知道哪些页用过和哪些页未用过。这种部分排序信息导致了许多近似LRU算法的页置换算法。

1. 附加引用位算法

通过在规定时间间隔里记录引用位，能获得额外顺序信息。可以为位于内存中的每个表中的页保留一个8 bit的字节。在规定时间间隔（如，每100 ms）内，时钟定时器产生中断并将控制转交给操作系统。操作系统把每个页的引用位转移到其8 bit字节的高位，而将其他位向右移，并抛弃最低位。这些8 bit移位寄存器包含着该页在最近8个时间周期内的使用情况。如果移位寄存器含有00000000，那么该页在8个时间周期内没有使用；如果移位寄存器的值为11111111，那么该页在过去每个周期内都至少使用过一次。

具有值为11000100的移位寄存器的页要比值为01110111的页更为最近使用。如果将这8 bit字节作为无符号整数，那么具有最小值的页为LRU页，且可以被置换。注意这些数字并不惟一。可以置换所有具有最小值的页或在这些页之间采用FIFO来选择置换。

当然，历史位的数量可以修改，可以选择（依赖于可用硬件）以尽可能快地更新。在极端情况下，数量可降为0，即只有引用位本身。这种算法称为**第二次机会页置换算法**（second-chance page-replacement algorithm）。

2. 二次机会算法

二次机会置换的基本算法是FIFO置换算法。当要选择一个页时，检查其引用位。如果其值为0，那么就直接置换该页。如果引用位为1，那么就给该页第二次机会，并选择下一个FIFO页。当一个页获得第二次机会时，其引用位清零，且其到达时间设为当前时间。因此，获得第二次机会的页，在所有其他页置换（或获得第二次机会）之前，是不会被置换的。另外，如果一个页经常使用以致于其引用位总是得到设置，那么它就不会被置换。

一种实现二次机会（有时称为时钟）算法的方法是采用循环队列。用一个指针表示下次要置换哪一页。当需要一个帧时，指针向前移动直到找到一个引用位为0的页。在向前移动时，它将清除引用位（图10.14）。一旦找到牺牲页，就置换该页，新页就插入到循环队列的这个位置。注意：在最坏情况下，所有位均已设置，指针会遍历整个循环队列，以便给每个页第二次机会。它将清除所有引用位后再选择页以置换。这样，如果所有位均已设置，那么二次机会置换就变成了FIFO置换。

3. 增强型二次机会算法

通过将引用位和修改位（10.4节）作为一个有序对来考虑，能增强二次机会算法。采用这两个位，有下面四种可能类型：

1. (0,0)最近没有使用且也没有修改——用于置换的最佳页。

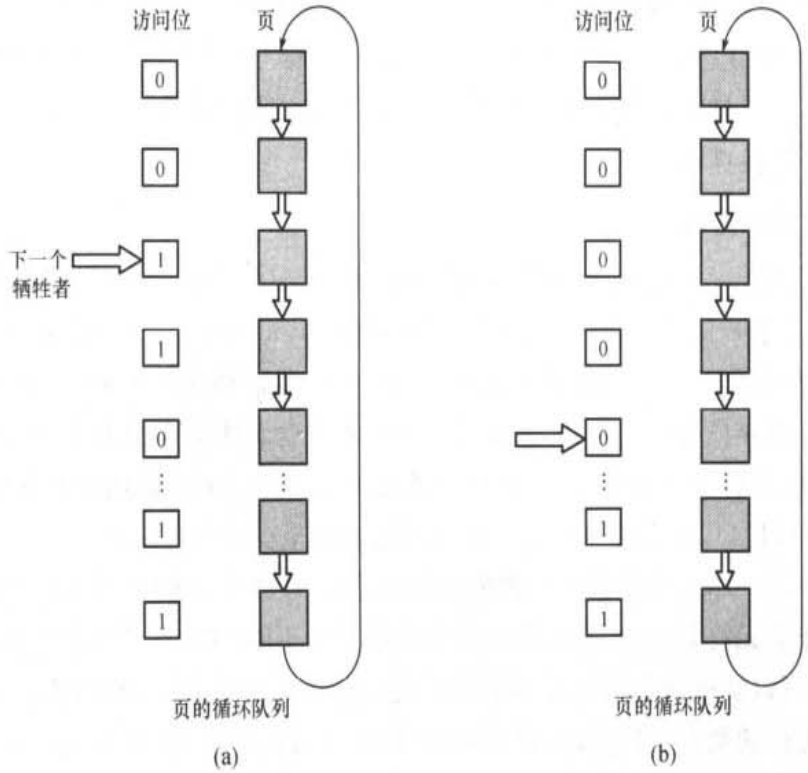


图 10.14 第二次机会(时钟)页置换算法

2. (0,1)最近没有使用但修改过——不是很好,因为在置换之前需要将页写出到磁盘。

3. (1,0)最近使用过但没有修改——它有可能很快又要被使用。

4. (1,1)最近使用过且修改过——它有可能很快又要被使用,且置换之前需要将页写出到磁盘。

当页需要置换时,每个页都属于这四种类型之一。可以使用时钟算法,不是检查所指页的引用位是否设置,而是检查所指页属于哪个类型。置换在最低非空类型中所碰到的页。注意在找到要置换页之前,可能要多次搜索整个循环队列。

这种算法用在了 Macintosh 虚拟内存管理方案中。这种方法与简单时钟算法的主要差别是这里给那些已经修改过的页以更高的级别,从而降低了所需 I/O 的数量。

10.4.6 基于计数的页置换

还有许多其他算法可用于页置换。例如,可以为每个页保留一个用于记录其引用次数的计数器,并可形成如下两个方案。

• **最不经常使用页置换算法**(least frequently used (LFU) page-replacement algorithm)要求置换计数最小的页。这种选择的理由是活动页应该有一个大的引用次数。这种算法会有如下问题:一个页在进程开始时使用得很多,但以后就不再使用。由于其使用过很多次,所以它有较大次数,即使它不再使用仍然会在内存中。解决方法之一是定期地将

次数寄存器右移一位，以形成指数衰减的平均使用次数。

• **最常使用页置换算法**(most frequently used (MFU) page-replacement algorithm)基于如下理论：具有最小次数的页可能刚刚调进来，且还没有使用。

正如你可以想象的，MFU 和 LFU 置换都不常用。这两种算法的实现都很费时，且并不能很好地近似 OPT 置换算法。

10.4.7 页缓冲算法

除了特定置换算法外，还经常采用其他措施。例如，系统通常保留一个空闲帧缓冲池。当出现页错误时，会像以前一样选择一个牺牲帧。然而，在牺牲帧写出之前，所需要的页就从缓冲池中读到空闲内存。这种方法允许进程尽可能快地重启，而无需等待牺牲帧的写出。当牺牲帧在以后写出时，它再加入到空闲池。

这种方法的扩展之一是维护一个已修改页的列表。每当调页设备空闲时，就选择一个修改页以写到磁盘上，接着重新设置其修改位。这种方案增加了当需要选择置换时干净页的概率而不必写出。

另一种修改是保留一个空闲帧的池，但要记住哪些页在哪些帧内。由于当帧写到磁盘上时其内容并没有修改，所以在该帧被重用之前如果需要使用原来的页，那么原来的页可直接从空闲池中取出以直接使用。这时并不需要 I/O。当一个页出错时，先检查所需要页是否在空闲池中。如果不在，那么才必须选择一个空闲帧来读入所需页。

这种技术与 FIFO 置换算法一起，用于 VAX/VMS 系统中。当 FIFO 置换算法错误地置换了一个常用页时，该页会从空闲缓冲池中很快调出，而不需要 I/O。这种空闲缓冲池提供了对相对较差、但却简单的 FIFO 置换算法的弥补。因为早期 VAX 并不正确实现引用位，所以这种方法是必需的。

10.5 帧 分 配

如何在各个进程之间分配一定的空闲内存？如果有 93 个空闲帧和 2 个进程，那么每个进程各得到多少帧？

最为简单的虚拟内存情况是单用户系统。考虑一个单用户系统，其页大小为 1 KB，其总内存为 128 KB。因此，共有 128 帧。操作系统可能使用 35 KB，这样用户进程可以使用 93 帧。如果采用纯请求页面调度，那么所有 93 帧开始都放在空闲链表上。当用户进程开始执行时，它会产生页错误。前 93 页错误会从空闲链表中获得帧。当空闲链表用完后，必须使用页置换算法以从位于内存的 93 个页中选择一个置换为第 94 页，以此类推。当进程终止时，这 93 个帧将再次放在空闲帧链表上。

这种简单策略有许多变种。可要求操作系统从空闲链表上分配其所有缓存和表空间。

当操作系统不再需要这些空间时，它们可用以支持用户调页。也可试图确保空闲链表上任何时候至少有 3 个空闲帧。因此，当出现页错误时，可以将页调入可用的帧。当发生页交换时，可以选择一个页，在用户进程继续执行时将其内容写到磁盘上。

虽然其他变种也有可能，但是其基本策略是清楚的：用户进程会分配到任何空闲帧。

当请求页面调度与多道程序结合在一起，就出现了不同的问题。多道程序同时将两个（或多个）进程放在内存中。

10.5.1 帧的最小数量

帧分配策略受到多方面的限制。所分配的帧不能超过可用帧的数量（除非有页共享）。也必须分配至少最少数量的帧。显然，随着分配给每个进程的帧数量的减少，页错误会增加，从而减慢进程的执行。

除了只分配少量帧会有不好的性能外，还必须要分配最少数量的帧。这一最小数量是由指令集合结构来定义的。记住：当在指令完成之前出现页错误时，该指令必须重新执行。因此，必须要有足够的帧来容纳所有单个指令所引用的页。

例如，考虑这样一个机器，其所有机器指令只有一个内存地址。因此，至少需要一帧用于指令，另一帧用于内存引用。另外，如果允许一级间接引用（例如，一条在 16 页上的 load 指令引用了 0 页上的地址，而这个地址又间接引用了第 23 页），那么每个进程至少需要 3 个帧。想一想如果只有 2 个帧，那么会如何。

帧的最小数量是由给定计算机结构定义的。例如，PDP-11 的移动指令在一定模式下为多个字长，因此指令本身需要 2 帧。另外，它有 2 个操作数，而每个操作数可能是间接引用，因此，共需要 6 个帧。对 IBM 370 而言，最坏情况可能是 MVC 指令。由于该指令是存储到存储器的，它需要 6 B 且可能跨 2 页。要移动的字符的块和要移动到目的的区域也可能都要跨页。这种情况需要 6 个帧（事实上，最坏情况可能是：MVC 指令作为 EXECUTE 指令的参数，后者本身跨两页；这样需要 8 帧）。

最坏情况出现在如下结构的计算机中：它们允许多层的间接（例如，每个 16 bit 的字可能包括一个 15 bit 的地址和 1 bit 的间接标记符。）从理论上来说，一个简单 load 指令可以引用另一个间接地址，而它可能又引用另一个间接地址（在另一页上），而它可能又再次引用另一个间接地址（又在另一页上），等等。因此，在最坏情况下，整个虚拟内存都必须在物理内存中。为了解决这一困难，必须对间接引用加以限制（例如，限制一个指令只能有 16 级的间接引用）。当出现首次间接引用时，计数器设置为 16；对该指令以后的每次间接引用，该计数器都要减 1。如果计数器减为 0，那么出现陷阱（过分间接引用）。这种限制使得每个指令的最大内存引用为 17，因而也要求同样数量的帧。

每个进程帧的最小数量是由体系结构来定义的，而最大数量是由可用物理内存的数量来定义的。在这两者之间，关于帧分配还是有很多选择的。

10.5.2 分配算法

在 n 个进程之间分配 m 个帧的最为容易的方法是给每个进程一个平均值 m/n 帧。例如，如果有 93 个帧和 5 个进程，那么每个进程可得到 18 个帧。剩余的 3 个帧可以放在空闲缓冲池上。这种方案称为**平均分配**(equal allocation)。

另外一种方法是要认识到各个进程需要不同数量的内存。考虑一下这样的系统，其帧大小为 1 KB。如果只有两个进程运行在系统上，且空闲帧数为 62，一个进程是具有 10 KB 的学生进程，另一个进程是具有 127 KB 的交互式数据库，那么给每个进程各 31 个进程帧就没有道理了。学生进程所需要的帧不超过 10 个，因此其他 21 帧就完全浪费了。

为了解决这个问题，可使用比例分配。根据进程大小，而将可用内存分配给每个进程。设进程 p_i 的虚拟内存大小为 s_i，且定义

$$S=\sum s_i$$

这样，如果可用帧的总数为 m，那么进程 p_i 可分配到 a_i 个帧，这里 a_i 近似为

$$a_i=s_i/S\times m$$

当然，必须调整 a_i 以使之成为整数且大于指令集合所需要的帧的最小数量，并使所有帧不超过 m。

采用比例分配，在两个进程之间(一个进程为 10 页，另一个为 127 页)按比例分配 62 帧：一个分配 4 帧，另一个分配 57 帧。这是因为

$$10/137\times 62\approx 4$$

$$127/137\times 62\approx 57$$

这样两个进程根据它们的需要(而不是平均地)获得可用内存。

当然，对于平均和比例分配，每个进程所分配的数量会随着多道程序的级别而有所变化。如果多道程序程度增加，那么每个进程会失去一些帧以提供给新来进程使用。另一方面，如果多道程序程度降低，那么原来分配给离开进程的帧可以分配给剩余进程。

注意，对于平均或比例分配，高优先级进程与低优先级进程一样处理。然而，根据定义，可能要给高优先级更多内存以加快其执行，同时就会损害到低优先级进程。

另一个方法是使用比例分配的策略，但是不根据进程相对大小，而是根据进程优先级或是大小和优先级的组合。

10.5.3 全局分配与局部分配

为各个进程分配帧的另一个重要因素是页置换。当有多个进程竞争帧时，可将页置换算法分为两大类：**全局置换**(global replacement)和**局部置换**(local replacement)。全局置换允许一个进程从所有帧集合中选择一个置取帧，而不管该帧是否已分配给其他进程；一个进程可以从另一个进程中取帧。局部置换要求每个进程仅从其自己的分配帧中进行选择。

例如，考虑这样一个分配方案：允许高优先级进程从低优先级进程中选择帧以便置换。一个进程可以从其他自己的帧中或任何低优先级进程中选择置换帧。这种方法允许高优先级进程增加其帧分配而以损失低优先级进程为代价。

采用局部置换策略，分配给每个进程的帧的数量不变。采用全局置换，一个进程可能从分配给其他进程的帧中选择一个进行置换，因此增加了所分配的帧的数量（假定其他进程不从它这里选择帧来置换）。

全局置换算法的一个问题是进程不能控制其页错误率。一个进程的位于内存的页集合不但取决于进程本身的调页行为，还取决于其他进程的调页行为。因此，相同进程由于外部环境不同可能执行得很不一样（有的执行可能需要 0.5 s，而有的执行可能需要 10.3 s）。局部置换算法就没有这样的问题。采用局部置换算法，进程内存中的页只受该进程本身的调页行为所影响。但是，因为局部置换不能使用其他进程的不常用的内存，所以会阻碍一个进程。因此，全局置换通常会有更好的系统吞吐量，且更为常用。

10.6 系统颠簸

如果低优先级进程所分配的帧数量少于计算机体系结构所要求的最少数量，那么必须暂停进程执行。接着应换出其他所有剩余页，以便使其所有分配的帧空闲。这引入了中程 CPU 调度的换进、换出层。

事实上，研究一下没有“足够”帧的进程。虽然从技术角度而言，可以将所分配帧的数量降低到最小值，但是有一定（更大）数量的帧正在使用。如果进程没有这些帧，那么它会很快出现页错误。这时，必须置换某个页。然而，其所有页都在使用，它置换一个页，但又立刻再次需要这个页。因此，它会不断地产生页错误。进程继续页错误，置换一个页而该页又立即出错且需要立即调进来。

这种频繁的页调度的行为称做**颠簸**（thrashing）。如果一个进程在换页上用的时间要多于执行时间，那么这个进程就在颠簸。

10.6.1 系统颠簸的原因

系统颠簸会导致严重的性能问题。考虑如下情况，这是基于早期调页系统的真实行为。

操作系统在监视 CPU 的利用率。如果 CPU 利用率太低，那么向系统中引入新进程，以增加多道程序的程度。采用全局置换算法；它会置换页而不管这些页是属于哪个进程。现在假设一个进程进入一个新执行阶段，需要更多的帧。它开始页错误，并从其他进程中取帧。然而，这些进程也需要这些页，所以它们也会出现页错误，从而从其他进程中取帧。这些页错误进程必须使用调页设备以换进和换出页。随着它们排队等待换页设备，就绪队列会变空。而进程等待调页设备，CPU 利用率就会降低。

CPU 调度程序发现 CPU 利用率降低，因此会*增加*多道程序的程度。新进程试图从其他运行进程中拿帧，从而引起更多页错误，更长的调页设备的队列。因此，CPU 利用率进一步降低，CPU 调度程序试图再增加多道程序的程度。这样系统颠簸就出现了，系统吞吐量陡降，页错误显著增加。因此，有效访问时间增加。最终因为进程主要忙于调页，系统不能完成一件工作。

这种现象如图 10.15 所示，显示了 CPU 利用率与多道程序程度的关系。随着多道程序程度增加，CPU 利用率（虽然有点慢）增加，直至达到最大值。如果多道程序程度还要继续增加，那么系统颠簸就开始了，且 CPU 利用率急剧下降。这时，为了增加 CPU 利用率和降低系统颠簸，必须*降低*多道程序的程度。

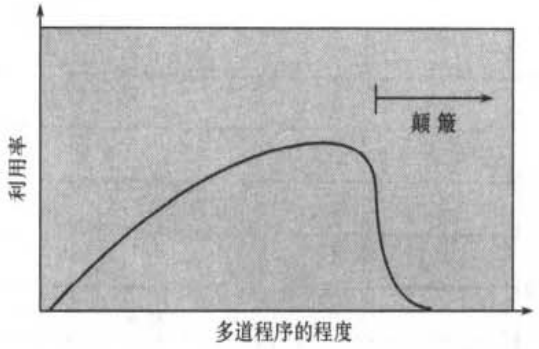


图 10.15 颠簸

通过**局部置换算法**（local replacement algorithm）（或**优先置换算法**（priority replacement algorithm））能限制系统颠簸的结果。采用局部置换，如果一个进程开始颠簸，那么它不能从其他进程取帧，且不能使后者也颠簸。所置换的页必须是进程自己的页。然而，如果进程颠簸，那么绝大多数时间内也会排队来等待调页设备。由于调页设备的更长的平均队列，页错误的平均等待时间也会增加。因此，即使对没有颠簸的进程，其有效访问时间也会增加。

为了防止颠簸，必须提供进程所需的足够多的帧。但是如何知道进程“需要”多少帧呢？可以采用很多技术。工作集合策略（10.6.2 小节）检查进程真正需要多少帧。这种方法定义了进程执行的**局部模型**（locality model）。

局部模型说明，当进程执行时，它从一个局部移向另一个局部。局部是一个经常使用页的集合（图 10.16）。一个程序通常由多个不同局部组成，它们可能重叠。

例如，当一个子程序被调用时，它就定义了一个新局部。在这个局部里，内存引用包括该子程序的指令、其局部变量和全局变量的子集。当该子程序退出时，因为子程序的局部变量和指令现已不再使用，进程离开该局部。可能在后面再次返回该局部。因此，可以看到局部是由程序结构和数据结构来定义的。局部模型说明了所有程序都具有这种基本的内存引用结构。注意局部模型是本书在讨论缓存之后到现在为止还未阐明的原理。如果对任何数

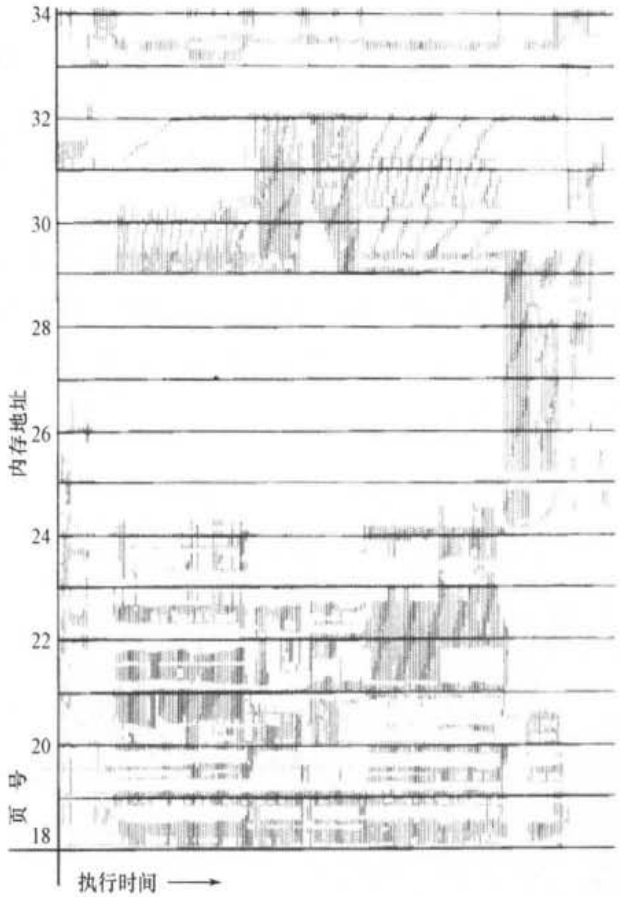


图 10.16 内存引用模式中的局部性

据类型的访问是随机的而没有一定的模式，那么缓存就没有用了。

假设为每个进程都分配了可以满足其当前局部的帧。该进程在其局部内会出现页错误，直到所有页均在内存中；接着它不再会出现页错误直到它改变局部为止。如果分配的帧数少于现有局部的大小，那么进程会颠簸，这是因为它不能将所有经常使用的页放在内存中。

10.6.2 工作集合模型

工作集合模型(working-set model)是基于局部的假设。该模型使用参数 Δ 定义**工作集合窗口**(working-set window)。其思想是检查最近 Δ 个页的引用。这最近 Δ 个引用的页集合称为**工作集合**(working-set)(图 10.17)。如果一个页正在使用中，那么它就在工作集合内。如果它不再使用，那么它会在其上次引用的 Δ 时间单位后从工作集合中删除。因此，工作集合是程序局部的近似。

例如，对于图 10.17 所示的内存引用序列，如果 Δ 为 10 个内存所引用，那么 t_1 时的工作集合为{1,2,5,6,7}。到 t_2 时，工作集合则成为{3,4}。

工作集合精确度与 Δ 的选择有关。如果 Δ 太小，那么它不能包含整个局部；如果 Δ 太大，那么它可能包含多个局部。在最为极端的情况下，如果 Δ 为无穷大，那么工作集合是进

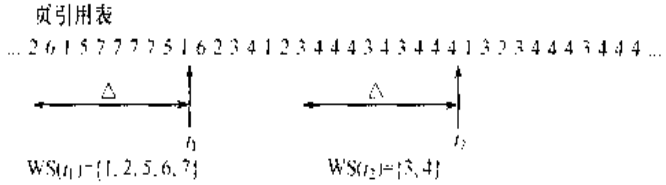


图 10.17 工作集合模型

程执行所碰到的所有页的集合。

最为重要的工作集合的属性是其大小。如果经计算而得到系统内每个进程的工作集合为 WSS_i，那么就得到

$$D=\sum WSS_i$$

其中，D 为总的帧需求量。每个进程都经常要使用位于其工作集合内的页。因此，进程 i 需要 WSS_i 帧。如果总的需求大于可用帧的数量($D>m$)，那么有的进程就得不到足够帧，从而会出现颠簸。

工作集合模型的使用较为简单。操作系统跟踪每个进程的工作集合，并为进程分配大于其工作集合的帧数。如果还有空闲帧，那么可启动另一个进程。如果所有工作集合之和增加以至于超过了可用帧的总数，那么操作系统会选择暂停一个进程。该进程的页被写出，且其帧可分配给其他进程。挂起的进程可以在以后重启。

这种工作集合策略防止了颠簸并尽可能地提高了多道程序的级别。因此，它优化了 CPU 使用率。

工作集合模型的困难是跟踪工作集合。工作集合窗口是移动窗口。在每次引用时，新引用会增加，而最老的引用会失去。如果一个页在工作集合窗口内被引用过，那么它就处于工作集合内。通过固定定时器中断和引用位，能近似工作集合模型。

例如，假设 Δ 为 10 000 个引用，且每 5 000 个引用会产生定时中断。当出现定时中断时，先复制再清除所有页的引用位。因此，当出现页错误时，可以检查当前引用位和位于内存内的两个位，从而确定在过去的 10 000 到 15 000 个引用之间该页是否被引用过。如果没有使用过，那么所有这 3 个位均为 0。只要有 1 个位为 1，那么就可认为处于工作集合中。注意，这种安排并不完全准确，这是因为并不知道在 5 000 个引用的什么位置出现了引用。通过增加历史位的位数和中断频率(例如，10 bit 和每 1 000 个引用就产生中断)，能够降低这一不确定性。然而，处理这些更为经常的中断的时间也会增加。

10.6.3 页错误频率

工作集合模型是成功的，工作集合知识能用于预先调页(10.8.1 小节)，但是用于控制颠簸有点不太灵活。一种更为直接的方法是采用**页错误频率**(page-fault frequency，PFF)策略。

这里的问题是防止颠簸。颠簸具有高的页错误率。因此，要控制页错误率。当页错误

率太高时，知道进程需要更多帧。类似地，如果页错误率太低，那么进程可能有太多的帧。可以为所期望的页错误率设置上限和下限(图 10.18)。如果实际页错误率超过上限，那么为进程分配更多帧；如果实际页错误率低于下限，那么可从该进程中移走帧。因此，能够直接测量和控制页错误率以防止颠簸。

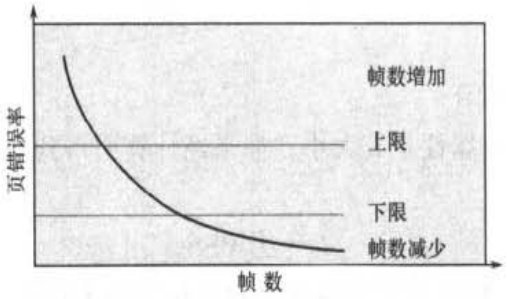


图 10.18 页错误频率

与工作集合策略一样，也可能必须暂停一个进程。如果页错误增加且没有可用帧，那么必须选择一个进程暂停。接着，可将释放的帧分配给那些具有高页错误率的进程。

10.7 操作系统样例

在本节，讨论 Windows NT 和 Solaris 2 如何实现虚拟内存。

10.7.1 Windows NT

Windows NT 采用请求页面调度加上**簇**(clustering)来实现虚拟内存。簇在处理页错误时，不但调入出错的页面，而且还调入出错页周围的页。当一个进程创建时，它会被分配工作集合最小值和最大值。**工作集合最小值**(working-set minimum)是进程在内存中时所保证拥有的页数量的最小值。如果有足够多的内存可用，那么进程可分配更多的页面，直至其最大值。(在有些环境下，进程可允许超过其工作集合的最大值。)虚拟内存管理器维护一个空闲帧的链表。与该链表相关联的是一个阈值，用来表示是否有足够多的可用内存。如果一个进程的页数低于其工作集合最大值，且出现页错误，那么虚拟内存管理器可从该空闲链表上分配帧。如果一个进程的页数已达到其工作集合最大值且出现页错误，那么虚拟内存管理器就采用局部置换以选择置换页。当空闲内存的量低于其阈值，虚拟内存管理器采用**自动工作集合修整**(automatic working-set trimming)以便使该值在其阈值之上。自动工作集合修整工作如下：计算分配给进程的帧数。如果进程分配的帧数大于其最小工作集合，那么虚拟内存管理器从中删除帧直到进程的页数为最小工作集合。一旦有足够多的空闲内存，那么具有最小工作集合页数的进程会从空闲帧中分配帧。

用于确定从哪个工作集合中删除页的算法与操作系统所运行的处理器类型有关。对于

单处理器 x86 系统，Windows NT 使用 10.4.5.2 一节所讨论的时钟算法的变种。对于 Alpha 和多处理器 x86 系统，清除引用位需要使其他处理器的 TLB 内容无效。为了避免这种开销，Windows NT使用了 10.4.2 小节所讨论的 FIFO 算法的一个变种。

10.7.2 Solaris 2

当一个线程产生一次页错误时，内核会从其所维护的空闲链表中为页错误线程分配一个页。因此，操作系统必须维护有足够多的空闲内存空间。与空闲链表相关的一个参数是 *lotsfree*，用于表示调页的阈值。通常将 *lotsfree* 设置为物理内存大小的 1/64。内核每秒钟会检查 4 次看空闲内存是否超过了 *lotsfree*。

如果空闲页数少于 *lotsfree*，那么就启动称为**换页**(pageout)的进程。换页进程采用类似于如 10.4.5.2 一节所述的二次机会算法(也称为**双指针时钟算法**(two-handed-clock algorithm))，工作原理如下。时钟的第一条指针扫描内存的所有页，并清除其引用位。之后，时钟的第二条指针会检查内存页的引用位，以释放那些引用位仍然为 0 的页到空闲链表。

换页算法使用多个参数控制扫描页的速度(称为 *扫描速度*)。扫描速度用每秒页数来表示，其范围从 *slowscan* 到 *fastscan*。当空闲内存低于 *lotsfree* 时，扫描从每秒 *slowscan* 页的速度开始，并根据可以用内存的数量可以增加到每秒 *fastscan* 页。*Slowscan* 的缺省值为每秒 100 页。*fastscan* 通常设置为每秒(*TotalPhysicalPages*)/2，且其最大值为每秒 8 192 次。参见图 10.19(*fastscan* 设置为最大值)。

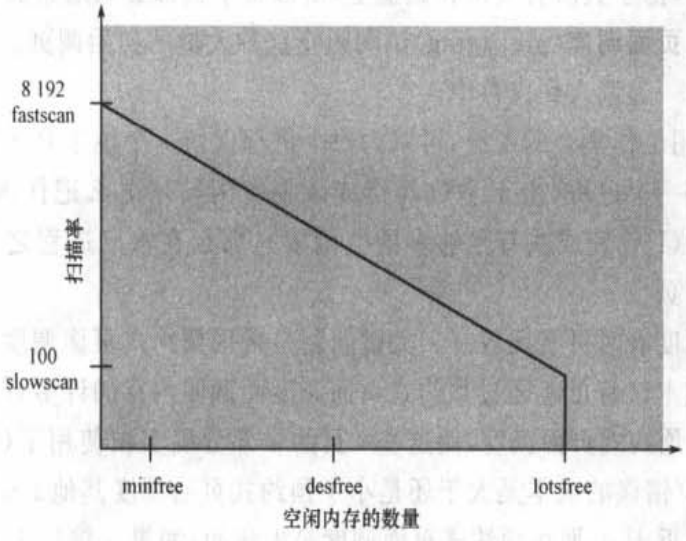


图 10.19 Solaris 2 页扫描器

时钟两针之间的距离(以页数表示)是由系统参数 *handspread* 来确定的。前针清除位和后针检查位的时间与 *scanrate* 和 *handspread* 有关。如果 *scanrate* 为每秒 100 页且 *handspread* 为 1 024 页，那么在前针清除位和后针检查位之间有 10 s。然而，由于对内存系统的需求，每秒

数千页的 *scanrate* 并不是不常见。这意味着前针清除位和后针检查位之间的时间只有数秒。

如上所述，**换页**进程每秒检查内存四次。然而，如果内存低于 *desfree*（图 10.19），那么换页会每秒运行 100 次以保证至少有 *desfree* 个可用空闲页。如果在 30 s 内，换页进程不能使空闲内存的数量的平均值超过 *desfree*，那么内核开始交换进程，以释放分配给该进程的所有页。通常，内核会查找空闲最长时间的进程。最后，如果系统还不能维护空闲内存的数量至少为 *desfree*，那么每次请求新页时会执行换页进程。

最近的 Solaris 2 的版本提供了对调页算法的若干改进。改进之一是识别共享库的页。为多个进程所共享的属于库的页，即使能为扫描程序所需要，也会在扫描时跳过。另一改变是区分分配给进程的页和分配给普通文件的页。这称为**优先权调页**（priority paging），将在 12.6.2 小节中讨论。

10.8 其他考虑

置换算法和分配策略的选择是调页系统的主要问题。然而，还有一些其他问题需要加以考虑。

10.8.1 预约式页面调度

纯请求页面调度系统的一个显著特性是当一个进程开始时会出现大量页错误。这种情况是由于试图将最初局部调入到内存的结果。在其他时候也会出现同样情况。例如，当重启一个换出进程时，由于其所有页都在磁盘上，所以每个页都必须通过其自己的页错误而调入到内存。**预约式页面调度**（prepaging）试图阻止这种大量的初始调页。这种策略就是同时将所需要的所有页一起调入到内存中。

例如，对于采用工作集合的系统，可以为每个进程保留一个位于其工作集合内的页的列表。如果必须暂停一个进程（由于 I/O 等待或缺少空闲帧），那么记住该进程的工作集合。当该进程要重启时（I/O 完成或有足够多的空闲帧），那么在重启进程之前自动调入位于其工作集合内的所有页。

预约式页面调度有时可能比较好。关键问题是采用预约式页面调度的成本是否小于处理相应页错误的成本。有可能通过预约式页面调度而调回内存的许多页可能没有被使用。

假设有 s 页被预约式页面调度，而这些 s 页的 α 部分确实被使用了（$0 \leqslant \alpha \leqslant 1$）。问题是节省的 $s \times \alpha$ 个页错误的成本是大于还是小于预约式页面调度其他 $s \times (1-\alpha)$ 不必要页面的开销。如果 α 接近于 0，那么预约式页面调度是失败的；如果 α 接近于 1，那么预约式页面调度是成功的。

10.8.2 页大小

现有机器的操作系统的设计人员在页大小方面很少有选择。然而，在设计新机器时，必

须对最佳页大小做出决定。正如你所预期的，并不存在单一的最佳页大小，而是有许多因素影响页面大小。页面大小总为 2 的幂，通常从 4 096(2^{12})到 4 194 304(2^{22})字节。

如何选择页大小？一方面是要考虑页表大小。对于给定虚拟内存空间，减低页大小增加了页数量，因此也增加了页表大小。对于 4 MB(2^{22})的虚拟内存，如页大小为 1 024 B，有 4 096 页；而页大小如果为 8 192 B，就只有 512 页。因为每个活动进程必须有其自己的页表，所以大的页是人们想要的。

另一方面，小页可更好地利用内存。如果进程从位置 00000 开始分配内存，且一直继续到它所需要的内存为止，那么它可能不能刚好在页边界处结束。因此，最后页的一部分必须分配(因为页是分配单元)但并未使用(内部碎片)。假定进程大小与页大小无关，那么可以推测：平均来说，每个进程的最后页的一半将会被浪费。对于 512 B 的页，损失为 256 B；对于 8 192 B 的页，损失为 4 096 B。为了减少碎片，需要小页。

另一个问题是页读写所需要的时间。I/O 时间包括寻道、延迟和传输时间。传输时间与传输量(即页大小)成正比，这似乎需要小页。然而，记住如第二章所述寻道和延迟时间远远超过传输时间。对于传输率为 2 MB/s，传输 512 B 需要 0.2 ms。另一方面，寻道时间可能为 20 ms 而延迟时间可能为 8 ms。因此，在总的 I/O 时间(28.2 ms)，1%是用于真正传输的。加倍页的大小使 I/O 时间增加到 28.4 ms。读入页大小为 1 024 B 的单个页需要 28.4 ms，但是读入页大小为 512 B 的两个页需要 56.4 ms。因此，为了最小化 I/O 时间，需要大页

然而，因为局部得以改善，采用小页，总的 I/O 就会降低。较小页允许每个页更精确地匹配程序局部。例如，考虑一个大小为 200 KB 的进程，其一半(100 KB)确实在执行时使用。如果只有一个大页，那么必须调入整个页，从而传输且分配了 200 KB。如果有大小为 1 B 的页，那么可调入确实使用的 100 KB，从而只传输且分配了 100 KB。采用较小页，有更好的**精度**，以允许只处理确实需要的内存。采用较大的页，必须分配且传输不但所需要的，而且还包括其他碰巧在页内的不需要的内容。因此，更小页应用导致更少的 I/O 和更少的总的分配内存。

另一方面，采用 1 B 大小的页，每页会产生页错误。大小为 200 KB 的进程，只使用其一半，如果采用 1 B 大小的页，那么会产生 102 400 个页错误。每个页错误会产生大量的额外开销以处理中断、保存寄存器、置换页、排队等待调页设备和更新表。为了降低页错误数量，需要大页。

还有其他因素需要考虑(如页大小和调页设备的扇区大小的关系)。这个问题没有最佳答案。有的因素需要小页(内部碎片、局部)，而有的需要大页(表大小、I/O 时间)。然而，总的来说是趋向更大的页。事实上，本书的第一版(1983 年)采用 4 096 B 为页大小的上限，这一数值是 1990 年最常用的页大小。然而，现代系统可能使用更大的页。在下节，将继续讨论这个问题。

10.8.3 TLB 范围

在第九章，讨论了 TLB **命中率**(hit ratio)。记住 TLB 命中率是指通过 TLB 而不是页表所进行的虚拟地址转换的百分比。显然，命中率与 TLB 的项目条数有关，增加 TLB 的条数可增加命中率。然而，这种方法的代价并不小，因为用于构造 TLB 的相关内存既昂贵又费电。

与命中率相关的另一类似测量尺度是 TLB **范围**(TLB reach)。TLB 范围指通过 TLB 可访问的内存量，并且等于 TLB 条数与页大小之积。理想情况，一个进程所有的工作集合应位于 TLB 中。否则，该进程因通过页表而不是 TLB 而浪费大量时间以进行地址转换。如果把 TLB 条数加倍，那么可加倍 TLB 范围。然而，对于某些使用大量内存的应用程序，这样做可能不足以存储工作集合。

增加 TLB 范围的另一个方法是增加页的大小或提供多种页大小。如果增加页大小，如从 8 KB 到 32 KB，那么 TLB 将翻两番。然而，对于不需要像 32 KB 这样大小页的应用程序，这会产生碎片。另一选择是，操作系统可使用不同大小的页。例如，UltraSparc Ⅱ支持 8 KB、64 KB、512 KB 和 4 MB 页大小。在这些可用页大小中，Solaris 2 使用了 8 KB 和 4 MB 页大小。对于具有 64 项的 TLB，Solaris 2 的 TLB 范围可从 512 KB(采用 8 KB 页大小)到 256 MB(采用 4 MB 页大小)。对于绝大多数应用程序，8 KB 页大小是足够了，但是 Solaris 2还是用两个 4 MB 大小的页来映射内核代码和数据开始的 4 MB。

然而，提供对多种页的支持要求操作系统而不是硬件来管理 TLB。例如，TLB 项的一个域必须用来表示对应 TLB 项的页大小。用软件而不是硬件来管理 TLB 会影响性能。然而，增加命中率和 TLB 范围也提高了性能。确实，现代趋势是软件管理 TLB 和操作系统提供对多种页大小的支持。UltraSparc、MIPS 和 Alpha 体系结构都采用软件管理 TLB。PowerPC 和 Pentium 采用硬件管理 TLB。

10.8.4 反向页表

在 9.4.3 一节，引入了反向页表的概念。这种形式的页管理能降低为了跟踪虚拟地址到物理地址转换所需的物理内存的数量。通过创建一个表，该表可包括每个物理内存页，且可根据<进程 ID，页码>来索引，从而可以节省内存。

因为它们保留了有关每个物理帧保存了哪个虚拟内存页面的信息，反向页表降低了保存这种信息所需的物理内存。然而，反向页表不再包括进程逻辑地址空间的完整信息，而如果所引用页不在内存中时又需要这种信息。请求页面调度需要用这种信息来处理页面错误。为了提供这种信息，每个进程必须保留一个外部页表。每个这样的表看起来如同传统的进程页表，以包括每个虚拟页所在的位置。

但是，外部页表会影响反向页表的用途吗？由于这些表只有在出现页错误时才需要引

用，所以它们不需要很快得到。事实上，它们本身根据需要可换进或换出内存。不过，可能会出现这样一个页错误(第一次页错误)，以至于在备份仓库上定位虚拟该页的外部页表需要调入内存(第二次页错误)。因此，虚拟内存管理器需要在内核中小心地处理并且在页查找处理时有一个延迟。

10.8.5 程序结构

对用户程序而言，请求页面调度被设计成是透明的。在许多情况下，用户可完全不关心内存的调页特性。然而，在有些情况下，如果用户(或编译器)对请求页面调度有所了解，那么能改善系统性能。

下面研究一个人为的但却有用的例子。假设页大小为 128 字。考虑一个 Java 程序，其主要功能是将 128×128 的两维数组的所有元素清零。其代码如下：

```
int A[][] = new int [128][128];

for (int j = 0; j<A.length; j++)
  for (int i = 0; i<A.length; i++)
    A[i][j] = 0;
```

注意数组是按行存放的。也就是说，数组存储顺序是 A[0][0], A[0][1], …, A[0][127], A[1][0], A[1][1], …, A[127][127]。由于，页大小为 128 字，所以每行需要一页。因此，以上代码只将每页的一个字清零，再将每页的下一个字清零，等等。如果操作系统分配给整个程序的帧数少于 128，那么其执行会产生 128×128=16 384 个页错误。如果将代码改为如下形式

```
int A[][] = new int [128][128];

for (int i = 0; i<A.length; i++)
  for (int j = 0; j<A.length; j++)
    A[i][j] = 0;
```

那么，在开始下页之前，会清除本页的所有字，从而将页错误的数量减低为 128。

数据结构和程序结构的仔细选择能增加局部性，并降低页错误率和工作集合内的页数。堆栈具有良好的局部性，因为访问局限于其顶部。另一方面，哈希表设计成分散引用，从而导致局部性差。当然，引用局部性只是数据结构使用效率的测度之一。其他重要度量要素包括搜索速度、内存引用的总数、所涉及页面的总数。

在后面的阶段，编译器和装入器(loader)对调页都有重要影响。代码和数据的分开和重入代码的生成意味着代码只能读且不能被修改。干净页在置换之前不必调出。装入器能避免子程序跨越页边界，以便使每个子程序完全在单个页内。互相多次调用的页可一起放在同一页内。这种打包是箱柜打包问题的操作研究的一个变形：试图将不同大小的代码段装

入到固定大小的页中以便页间引用最少。这种方法对于大页尤其有用。

程序设计语言对调页也会有影响。例如,C 和 C++经常使用指针,指针趋向于使内存访问随机,因此降低了进程的局部性。有的研究表明面向对象程序的引用局部性也较差。与 C/C++不同,Java 并不使用指针,在虚拟内存系统上,Java 程序比 C 或 C++程序具有更好的局部性。

10.8.6 I/O 互锁

在使用请求页面调度时,有时需要允许某些页在内存中被锁住。这种情况之一是需要对用户(虚拟)内存进行 I/O。I/O 通常通过一个单独的 I/O 处理器完成。例如,一个磁带控制器通常设置为需要传输多少字节和缓冲的内存地址(图 10.20)。当完成传输时,CPU 被中断。

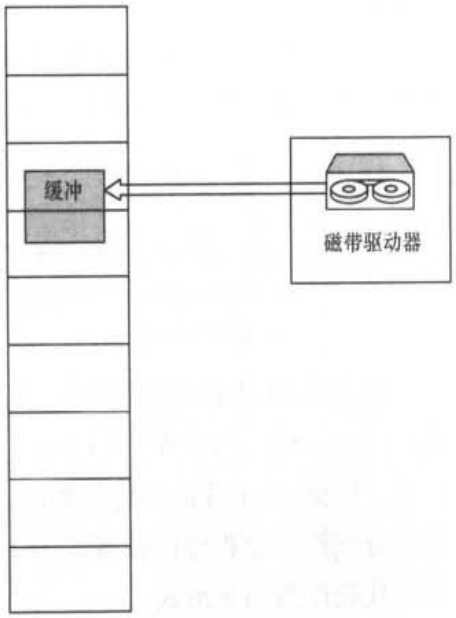


图 10.20 I/O 用到的帧必须要在内存中的原因

必须确保下面的事件序列不会出现:一个进程发出一个 I/O 请求,并被加入到 I/O 设备的等待队列上。同时,CPU 被交给了其他进程。这些进程引起页错误,采用全局置换算法,其中之一置换了等待进程用于内存缓冲的页。这些页被换出。之后,当 I/O 请求移到设备队列的头部时,就针对指定地址进行 I/O。然而,这时该帧已被属于另一个进程的不同页所使用。

对这个问题,通常有两种解决方法。一种解决方法是决不对用户内存进行 I/O。即,I/O 只能在系统内存和 I/O 设备之间进行,数据在系统内存和用户内存之间进行复制。为了向磁带上写一块,必须将该块复制到系统内存中,接着再写到磁带上。

这种额外复制可能会导致令人难以接受的高开销。另一解决方法是允许页锁在内存中。每个帧都有一个锁住位。如果一个帧被锁住,那么它不能被置换。在这种情况下,为了向磁带上写一块,可将包括该块内容的页锁住。而系统能像平常一样继续。锁住页不能被置换。当 I/O 完成时,页就被解锁。

操作系统的有些页或所有页,通常都锁在内存中。绝大多数操作系统不能容忍由内核引起的页错误。想一想页置换子程序引起页错误的结果。

锁住位的另一个用途涉及到普通页置换。考虑如下顺序的事件。一个低优先级进程产生页错误,调页系统会选择一置换帧,并将所需要页读入到内存。为了继续进行,让低优先级进程进入就绪队列并等待 CPU。由于是低优先级进程,所以可能有一段时间不被 CPU

调度程序所选择。在该低优先级进程等待时，一个高优先级进程出现页错误。调页系统寻找置换，发现一个页在内存中，但没有引用或修改；这是由低优先级进程刚才调入的。该页看起来正好置换：它是干净的，并不需要写出，且它很长时间没有使用过。

高优先级进程的页能否置换低优先级进程的页是个策略问题。毕竟，只是为了高优先级进程的利益而延迟了低优先级进程。另一方面，浪费了时间以调入低优先级进程的页。如果决定防止置换刚调入的页直到它至少运行一次，那么可用锁住位来实现这种机制。当一个页选择置换时，其锁住位打开；它一直打开直到出错进程再次分派为止。

然而，采用锁住位有时也有危险：锁住位可能被打开，但从未关闭。如果出现这种情况（例如，由于操作系统的错误），那么锁住帧就不能使用了。Macintosh 操作系统提供了页锁住机制，这是因为它是单用户操作系统，所以过分加锁只会损害加锁用户本身。多用户系统不能过分相信用户。例如，Solaris 2 允许加锁"提示"，但是在空闲帧池太少或单个进程请求在内存中锁住太多页时，能自由地忽略这些提示。

10.8.7 实时处理

本章的讨论主要集中于通过优化内存使用以提供计算机系统的最佳总体使用率。通过为活动数据提供内存而将非活动数据移到磁盘，增加了总体系统吞吐量。然而，单个进程可能会因此受影响，因为执行时可能会引起更多的页错误。

考虑一下第四章所述的实时进程或线程。这类进程需要获得 CPU 控制，并以最小延迟来运行直到结束。虚拟内存与实时计算相违背，因为它在进程执行（将页调入到内存）时引入了不可预测的较长延迟。因此，实时系统几乎从不采用虚拟内存。

对于 Solaris 2，Sun 公司的开发人员希望在一个系统内不但允许分时计算而且也允许实时计算。为了解决页置换问题，Solaris 2 允许进程告诉它哪些页是重要的。除了允许页使用"提示"外，操作系统还允许特权用户来将页锁在内存中。为了允许实时进程具有有限的较低的延迟，这种方法是必要的。如果滥用，那么这种机制会使得所有其他进程不能访问系统。

10.9 小 结

很需要能执行这样一个进程，其逻辑地址空间大于可用物理地址空间。程序可通过覆盖技术来执行这样的进程，但是这通常是个困难的编程任务。虚拟内存允许将大逻辑地址空间映射到小的物理内存。虚拟内存允许极大的进程运行，且提高了多道程序的程度，增加了 CPU 使用率。再者，它使得程序员不必考虑内存可用性。

纯请求页面调度只有在引用页时才调入页。第一次引用就会对操作系统驻留监控程序产生页错误。操作系统检查内部表以确定该页在备份仓库上的位置。接着，它找到空闲帧并从备份仓库中读入页。更新页表以反映这一修改，并重启产生页错误的指令。这种方法

允许一个进程运行，即使其完整内存映像不能同时在内存中。只要页错误率足够低，那么性能就可接受。

可以使用请求页面调度来降低分配给进程的帧数。这种安排能增加多道程序的程度（允许更多进程能同能执行），且（至少从理论上说）增加了系统 CPU 的使用率。即使进程内存需要超过了总的物理内存，也能允许进程执行。这些进程可在虚拟内存中执行。

如果总的内存需求超过了物理内存，那么有可能必须置换内存中的页为空闲帧以便被新页所用。可以使用各种页置换算法。FIFO 页置换算法容易编程，但有 Belady 异常。最优页置换算法需要未来的知识。LRU 页置换能近似最优算法，但是它也很难实现。绝大多数页置换算法如二次机会算法，都是 LRU 置换的近似。

除了页置换算法，还需要帧分配策略。分配可以是固定的，如局部页置换算法，或动态的如全局置换。工作集合模型假定进程在局部内执行。工作集合是位于当前局部所有页的集合。相应地，每个进程应该为其当前工作集合分配足够多的帧。

如果进程没有足够多的内存以用于其工作集合，那么它就会颠簸。为进程提供足够多的内存以避免颠簸可能需要进程交换和调度。

除了要求解决页置换和帧分配的主要问题外，合理设计调页系统还要求考虑页大小、I/O、加锁、预约式页面调度、进程创建、程序结构、颠簸和其他问题。虚拟内存可作为计算机系统存储层次的一层。每层都有其自己的访问时间、大小和代价参数。一个完整的、可用的、混合的虚拟内存系统将在 Mach 一章中加以描述，可以从网站上获取（http://www.bell-labs.com/topic/books/os-book）。

习 题 十

10.1 在什么情况下出现页错误？描述一下在发生页错误时操作系统做了哪些动作？

10.2 假设有个页引用串，它的进程有 m 个帧（初始时全空）。页引用串的长度为 p，里面有 n 个不同的页面数。对各种页面置换算法回答下面这些问题。

a. 发生页错误的次数的下限是多少？

b. 发生页错误的次数的上限是多少？

10.3 某个计算机给它的用户提供了 2^{32} B 的虚拟内存空间，计算机有 2^{14} B 的物理内存。虚拟内存使用页面大小为 4 094 B 的分页机制实现。一个用户进程产生虚拟地址 11123456，现在说明一下系统怎样建立相应的物理地址。区分一下软件操作和硬件操作。

10.4 对于请求调页环境，下面编程技巧和结构哪些“好”？哪些“坏”？为什么？

a. 堆栈

b. 哈希符号表

c. 顺序检索

d. 二分法检索

e. 纯代码

f. 向量操作

g. 间接寻址

10.5 假设有一个请求调页存储器，页表放在寄存器中。处理一个页错误，当有空的帧或被置换的页没有被修改过时要用 8 ms，当被置换的页被修改过时用 20 ms。存储器访问时间为 100 ns。

假设被置换的页中有 70%被修改过，有效访问时间不超过 200 ns 时最大可接受的页错误率是多少？

10.6 考虑下面的页置换算法。按它们的页错误率将这些算法分为从"差"到"完美"5 级刻度范围。将下面的算法分为受 Belady 异常影响和不受影响的两类。

a. LRU 置换算法

b. FIFO 置换算法

c. 最佳置换算法

d. 第二次机会置换算法

10.7 当一个计算机系统中实现了虚拟内存时，它带来一些好处时也付出了一些代价。列出这些好处和代价。可能有时代价会超过好处，解释可采取什么措施确保不会发生这种不平衡。

10.8 一个操作系统支持分页虚拟内存，它使用一个时钟周期为 1 μs 的中央处理器。访问其他页要比访问当前页多花费 1 μs。一页有 1 000 个字，分页设备是一个每分钟 3 000 转的磁鼓并且它的数据传输率为每秒钟 100 万字。从系统中可获得下列统计数据。

- 所有执行指令中有百分之一不对当前页进行访问。
- 在访问其他页的指令中，百分之八十访问一个已经在内存中的页。
- 当要求一个新页时，被置换出去的页百分之五十是修改过的。

假设系统只运行一个进程并且当磁鼓传递数据时处理器空转，计算此系统的有效指令时间。

10.9 假设一个具有下面时间度量利用率的请求调页系统：

CPU 利用率	20%
分页磁盘	97.7%
其他 I/O 设备	5%

说明下面哪一个(可)能提高 CPU 的利用率，为什么？

a. 安装一个更快的 CPU。

b. 安装一个更大的分页磁盘。

c. 提高多道程序设计程度。

d. 降低多道程序设计程度。

e. 安装更多内存。

f. 安装一个更快的硬盘，或对多个硬盘使用多个控制器。

g. 对页面调度算法添加预取页。

h. 增加页面大小。

10.10 假设有二维数组 A：

```
int A[][] = new int[100][100];
```

在一个页面大小为 200 的分页内存系统中，A[0][0]存放在地址 200。一个操作数组 A 的小进程驻存在页面 0(地址 0 到 199)；这样，每条指令都将从页面 0 中获取。

对于 3 个页帧，下面的数组初始化循环将会产生多少个页错误？假设使用 LRU 置换算法，页帧 1 中存

放进程，另外两个初始时为空。

```
a.    for ( int j = 0; j < 100; j++ )
          for (int i = 0; i < 100; i++ )
              A[i][j] = 0;
b.    for ( int i = 0; i < 100; i++ )
          for (int j = 0; j < 100; j++ )
              A[i][j] = 0;
```

10.11 假设有下面页引用序列：

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

下面的页面置换算法会产生多少次页错误？分别假设帧有1,2,3,4,5,6,7个。所有的帧初始时为空，第一个页调入时都会引发一次页错误。

- LRU置换算法
- FIFO置换算法
- 最优置换算法

10.12 假设你想使用一个有引用位的分页算法(像二次机会置换或工作集模型)，但是硬件不支持。大致说一下如果硬件不支持你怎样模拟一个引用位，计算一下这样做的开销。

10.13 你设计了一个你认为是最好的页面置换算法。在一些特殊的测试情形中，出现了Belady异常。那么这个新算法是最优算法吗？为什么？

10.14 假设你的置换策略(在分页系统中)是有规律地检查每个页并将最近一次检测后没有再被引用的页丢弃。与LRU或次二次机会置换算法相比，使用这种策略有哪些好处和坏处？

10.15 分段和分页类似，只不过使用了可变大小的"页"。设计两种以FIFO和LRU页面置换算法为基础的段置换算法。注意，由于段的大小不相同，被选择置换的段也许没有留下足够的连续地址给需要使用的段。分别考虑段不可以被重定位和可以被重定位时的系统策略。

10.16 一个页面置换算法应使发生页错误的次数最小化。怎样才能通过将使用频率高的页平均分配到整个内存而不只是竞争使用少数几个页帧页来达到这种最小化。可以对每个页帧设置一个计数器来记录与此帧相关的页数。那么当置换一个页时，就可以查找计数器值最小的页帧。

a. 基于这种基本思想，定义一个页面置换算法。特别注意的问题：

I. 计数器初始值是多少？

II. 什么时候计数器值增加？

III. 什么时候计数器值减少？

IV. 怎样选择要被置换的页？

b. 设有4个页帧，对于下面引用序列，你的算法会发生多少次页错误？

1,2,3,4,5,3,4,1,6,7,8,7,8,9,7,8,9,5,4,5,4,2

c. 最佳页面置换算法对于4页帧的题b中的引用序列的最小页错误数为多少？

10.17 假设一个请求调页系统具有一个平均访问和传输时间为20 ms的分页磁盘。地址转换是通过在主存中的页表来进行的，每次内存访问时间为1 μs。这样，每个通过页表进行的内存引用都要访问内存两次。为了提高性能，加入一个相关内存，当页表项在相关内存中时，可以减少内存引用的访问次数。

假设80%的访问发生在相关内存中，而且剩下中的10%(总量的2%)会导致页错误。内存的有效访问

时间是多少?

10.18 假设一个请求调页计算机系统的多道程序设计的程度现在固定为4进程。最近对系统做了测试来确定CPU和分页磁盘的利用率,结果是下面的选项之一。对于每种情况,发生了什么?你能够通过增加多道程序设计程度来提高CPU的利用率吗?分页是否有利于提高性能?

a. CPU利用率,13%;磁盘利用率,97%。

b. CPU利用率,87%;磁盘利用率,3%。

c. CPU利用率,13%;磁盘利用率,3%。

10.19 有一个适用于使用基址寄存器和界限寄存器机器的操作系统,但是已经将机器改成使用页表了。能否通过建立页表来模拟基址寄存器和界限寄存器?怎么做,或为什么不可以这么做?

10.20 颠簸的原因是什么?系统怎样检测颠簸?一旦系统检测到颠簸,系统怎样做来消除这个问题?

10.21 写一个程序来实现本章中的FIFO和LRU页面置换算法。首先,产生一个随机的页面引用序列,页面数范围从0到9。将这个随机页面引用序列应用到每个算法并记录所发生的页错误的次数。假设使用请求调页,页帧数范围从1到7分别实现置换算法。

推荐读物

在1960年左右(Kilburn等[1961]),请求调页最先在曼彻斯特大学MUSE计算机上实现,应用于Atlas系统中。另一个早期的请求调页系统是MULTICS,是在GE 645系统上实现的(Organick[1972])。

Belady等[1969]是最早观察到FIFO置换策略会产生现在命名为Belady异常现象的研究者。Mattson等[1970]讲述了堆栈算法不会导致产生Belady异常。

Belady[1966]给出了最优置换算法。Mattson等[1970]证明了它是最优的。Belady的最优算法是针对一个确定分配的;Prieve和Fabry[1976]有一个针对可变分配情况的最优算法。

Carr和Hennessy[1981]论述了改进的时钟算法;这个算法被用在Macintosh的虚拟内存管理设计方法中,Goldman[1989]描述了这个算法。

Denning[1968]开发了工作集模型。Denning[1980]给出了有关工作集模型的论述。

Wulf[1969]设计了管理页错误率的方案,还将这种技术成功地应用到Burroughs B5500计算机系统中。Gupta和Franklin[1978]给出了工作集和页错误频率置换两个方案的性能比较。

Solomon[1998]描述了Windows NT是怎样实现虚拟内存的。Mauro和McDougall[2001]论述了Solaris 2中的虚拟内存。Bovet和Cesati[2001]和McKusick等[1996]分别描述了Linux和BSD中的虚拟内存技术。Iacobucci[1988]描述了OS/2和请求分段的细节。Ganapathy和Schimmel[1998]论述了支持不同大小页面的操作系统。

Intel[1986]中有关于Intel 80386分页硬件的出色论述,Motorola[1989b]中有关于Motorola 68030的硬件的出色论述。Levy和Lipman[1982]论述了VAX/VMS操作系统的虚拟内存管理。Hagmann[1989]给出了关于工作站操作系统和虚拟内存的论述。Jacob和

Mudge[1998b]中有关于虚拟内存在 MIPS、PowerPC 和奔腾体系结构中里实现的比较。一个配套的文章(Jacob 和 Mudge[1998a])中描述了在六个不同体系结构中对实现虚拟内存的必要的硬件支持,包括 UltraSPARC 体系。

第十一章　文件系统接口

对绝大多数用户而言，文件系统是操作系统中最为可见的部分。它提供了在线存储、访问计算机操作系统和所有用户的程序与数据的机制。文件系统由两个不同部分组成：一组*文件*（文件用于存储相关数据）和（一组）*目录*（目录用于组织系统内的文件并提供有关文件的信息）*结构*。有的文件系统还有第三部分——分区，用于在逻辑上或物理上区分不同的目录组合。本章主要讨论文件的各个方面和主要目录结构。另外，本章还讨论各种文件保护的方法，当多个用户访问文件和需要控制哪些用户按什么方式访问文件时，保护是很有必要的。最后，本章还讨论了在多个进程、用户和计算机之间文件共享的语义。

11.1　文件概念

计算机能在多种不同媒介（如磁盘、磁带和光盘）上存储信息。为了方便地使用计算机系统，操作系统提供了信息存储的统一逻辑接口。操作系统对存储设备的各种属性加以抽象并且定义了逻辑存储单元（*文件*），再将文件映射到物理设备上。这些物理设备通常为非易失性的，这样其内容在掉电和系统重启时也会保持一致。

文件是记录在外存上相关信息的具有名称的集合。从用户角度而言，文件是逻辑外存的最小分配单元，即数据除非在文件中，否则不能写到外存。通常，文件表示程序（源形式和目标形式）和数据。数据文件可以是数字的、字符的、字符数字的或二进制的。文件可以是自由形式，如文本文件，也可以是具有严格格式的。通常，文件由位、字节、行或记录组成，其具体意义是由文件创建者和使用者来定义的。因此，文件的概念极为广泛。

文件信息是由其创建者定义的。文件可存储许多不同类型的信息：源程序、目标程序、可执行程序、数字数据、文本、工资记录、图像、声音记录等。文件根据其类型具有一定**结构**。文本文件由行（或页）组成，而行（或页）是由字符组成的。源文件由子程序和函数组成，而它们又是由声明和执行语句组成的。*目标文件*是一系列字节序列，它们按目标系统链接器所能理解的方式组成。*可执行文件*是一系列代码段，以供装入程序调入内存执行。

11.1.1　文件属性

文件是有名称的，以方便用户通过名称对之加以引用。名称通常为字符串，如 example.c。有的系统区分名称中的大、小写字符，而其他的则不加以区分。在文件被命名后，它就独立

于进程、用户、甚至创建它的系统。例如，一个用户可能创建了文件 *example.c*，而另一用户就可通过此名称来编辑该文件。文件拥有者可能将文件写入到软盘、通过电子邮件发送或通过网络拷贝，且在目的系统上它仍然被称为 *example.c*。

文件有一定的属性，这根据系统而有所不同，但是通常都包括如下属性：

- **名称**：文件符号名称是惟一的，按照人们容易读取的形式保存。
- **标识符**：标识文件系统内文件的惟一标签，通常为数字；这是文件的对人而言不可读的名称。
- **类型**：被支持不同类型的文件系统所使用。
- **位置**：该信息为指向设备和设备上文件位置的指针。
- **大小**：文件当前大小(以字节、字或块来计)，该属性也能可包括可允许大小的最大值。
- **保护**：决定谁能读、写、执行等的访问控制信息。
- **时间、日期和用户标识**：文件创建、上次修改和上次访问都可能有该信息。这些数据用于保护、安全和使用跟踪。

所有文件的信息都保存在目录结构中，而目录结构也保存在外存上。通常，目录条目包括文件名称及其惟一标识符。而标识符又定位其他属性信息。一个文件的这些信息可能需要1 KB。在具有许多文件的系统中，目录大小本身可能有数百万字节。因为目录如同文件一样也必须是非易失性的，所以它们必须被存放在设备上，并在需要时分若干次调入内存。

11.1.2 文件操作

文件属于**抽象数据类型**。为了合适地定义文件，需要考虑有关文件的操作。操作系统提供系统调用对文件进行创建、写、读、定位和截短。下面讨论操作系统的6个基本文件操作的实现。这样可以很容易地了解类似操作(如重命名文件)是如何实现的。

- **创建文件**：创建文件有两个必要步骤。第一，必须在文件系统中为文件找到空间。在第十二章会讨论如何为文件分配空间。第二，在目录中为新文件创建一个条目。该目录条目记录了文件名称、在文件系统中的位置以及其他可能的信息。
- **写文件**：为了写文件，执行一个系统调用，它指明文件名称和要写入文件的内容。对于给定文件名称，系统会搜索目录以查找文件位置。系统必须为该文件维护一个写位置的指针。每当发生写操作时，必须更新写指针。
- **读文件**：为了读文件，使用一个系统调用，并指明文件名称和要读入文件块的内存位置。同样，需要搜索目录以找到相关目录项，系统要为该文件维护一个读位置的指针。每当发生读操作时，必须更新读指针。一个进程通常只对一个文件读或写，所以当前操作位置可作为每个进程当前文件位置指针。由于读和写操作都使用同一指针，节省了空间也降低了系统复杂度。
- **在文件内重定位**：为相应条件搜索目录，将当前文件位置设为给定值。重定位不需

要真正读写文件。该文件操作也称为文件寻址。

• **删除文件**：为了删除文件，在目录中搜索给定名称的文件。找到相关目录条目后，释放所有的文件空间以便其他文件使用，并删除相应目录条目。

• **截短文件**：用户可能只需要删除文件内容而保留其属性，而不是强制用户删除文件再创建文件。该函数允许所有文件属性都不变，而只是将其长度设为 0 并释放其空间。

这 6 个基本操作组成了所需文件操作的最小集合。其他常用操作包括向现有文件之后*添加*新信息和重命名现有文件。这些基本操作可以组合起来执行其他文件操作。例如，创建一个文件的*复制*，或复制文件到另一个 I/O 设备如打印机或显示器，可以这样来完成：创建一个新文件，从旧文件读入并写出到新文件。人们还希望有文件操作用于获取和设置文件的各种属性。例如，可能需要文件操作以允许用户确定文件属性如文件长度以及允许用户设置文件属性如文件拥有者。

以上所述的绝大多数文件操作都涉及到为给定文件搜索相关目录条目。为了避免这种不断的搜索操作，许多系统要求在首次使用文件时，使用系统调用 open。操作系统维护一个包含所有打开文件的信息的表(**打开文件表**，open-file table)。当需要一个文件操作时，可通过该表的一个索引指定文件，而不需要搜索。当文件不再使用时，进程可以关闭它，操作系统从打开文件表中删除这一条目。

有的系统在首次引用文件时，会隐式地打开它。在打开文件的作业或程序终止时会自动关闭它。然而，绝大多数操作系统要求程序员在使用文件之前，显式地打开它。操作 open 会根据文件名搜索目录，并将目录条目复制到打开文件表。调用 open 也可接受访问模式参数：创建、只读、读写、添加等。该模式可以根据文件许可位进行检查。如果请求模式获得允许，进程就可打开文件。系统调用 open 通常返回一个指向打开文件表中一个条目的指针。通过使用该指针，而不是真实文件名称，进行所有 I/O 操作，以避免进一步搜索和简化系统调用接口。

对于多用户环境如 UNIX，操作 open 和 close 的实现更加复杂。在这些系统中，多个用户可能同时打开一个文件。通常，操作系统采用两级内部表：单个进程的表和整个系统的表。单个进程表跟踪单个进程打开的所有文件。表内所存是该进程所使用的文件的信息。例如，每个文件的当前文件指针就保存在这里，以表示下一个 read 或 write 调用所影响的文件位置。另外，还包括文件访问权限和记账信息。

单个进程表的每一条目相应地指向整个系统的打开文件表。整个系统表包含进程无关信息如文件在磁盘上的位置、访问日期和文件大小。一旦一个进程打开一个文件，另一个进程执行 open，其结果只不过简单地在其进程打开表中增加一个条目，并指向整个系统表的相应条目。通常，系统打开文件表的每个文件还有一个文件打开计数器 *open count*，以记录多少进程打开了该文件。每个 close 会递减 *count*，当打开计数器为 0 时，表示该文件不再被使用，该文件条目可从系统表中删除。总而言之，每个打开文件有如下关联信息：

• **文件指针**：对于没有将文件偏移量作为系统调用 read 和 write 参数的系统，系统必须跟踪上次读写位置以作为当前文件位置指针。这种指针对打开文件的某个进程来说是惟一的，因此必须与磁盘文件属性分开保存。

• **文件打开计数**：当文件关闭时，操作系统必须重用其打开文件表条目，否则表内的空间会不够用。因为多个进程可能打开一个文件，所以系统在删除打开文件条目之前，必须等待最后一个进程关闭文件。该计数器跟踪打开和关闭的数量，在最后关闭时计数为 0。这时，系统可删除该条目。

• **文件磁盘位置**：绝大多数文件操作要求系统修改文件数据。用于定位磁盘上文件位置的信息保存在内存中以免为每个操作从磁盘中读取该信息。

• **访问权限**：每个进程用一个访问模式打开文件。这种信息保存在单个进程打开文件表中以便操作系统能允许或拒绝以后的 I/O 请求。

有的操作系统提供了措施，以便为多进程的访问而锁住打开文件的部分内容，并可以在多进程之间共享部分文件（采用共享页），还可以将部分文件内容通过内存映射而映射到虚拟内存系统（10.3.2 小节）。

11.1.3 文件类型

当设计文件系统（事实上包括整个操作系统）时，总是要考虑操作系统是否应该识别和支持文件类型。如果操作系统识别文件类型，那么它就能按合理方式对文件进行操作。例如，一个常见错误是用户试图打印一个二进制目标形式的程序。这种尝试通常会产生垃圾，但是如果操作系统已知文件是二进制目标程序那么就能阻止它被打印。

实现文件类型的常用技术是在文件名称内包含类型。名称可分为两部分：名称和*扩展名*，通常用圆点加以分隔（图 11.1）。这样，用户和操作系统仅仅通过文件名称就能确定文件类型是什么。例如，对于 MS-DOS，文件名称可由不超过 8 个字符的主要部分，加上圆点，再加上不超过 3 个字符的扩展部分所组成。例如，只有具有扩展名为 *.com*、*.exe*、*.bat* 的文件才可执行。*.com* 和 *.exe* 是两种形式的二进制可执行文件，而 *.bat* 文件则是字符形式的**批处理文件**（batch file），包括操作系统的命令。虽然 MS-DOS 只识别少量扩展，但是应用程序也可使用扩展名表示其所感兴趣的文件类型。例如，汇编程序认为其源文件具有 *.asm* 扩展名。WordPerfect 字处理器认为其文件具有 *.wp* 扩展名。这些扩展名并不是必需的，所以用户可以不用扩展名（节省打字）来指明文件，应用程序会查找具有给定名称和其所期待的扩展名的文件。因为这些扩展名不是由操作系统所支持的，所以它们只作为能操作它们的应用程序的提示。

另一个使用文件类型的例子来自 TOPS－20 操作系统。如果用户试图执行目标程序，且其源文件自从生成目标文件后已被修改或编辑，那么源文件就会自动被编译。这种功能确保用户只是运行最新的目标文件。否则，用户可能会浪费大量时间来执行旧的目标文件。

为了实现这种功能，操作系统必须能区分目标文件和源文件，检查每个文件所创建或修改的时间，确定源程序的语言（以便使用正确的编译器）。

文件类型	通常文件后缀名	功能
可执行文件	exe，com，bin 或无	就绪可执行机器语言程序
目标文件	obj，o	已编译，机器语言，未链接
源文件	c，cc，java，pas，asm，a	各种语言的源代码
批处理文件	bat，sh	发送给命令解释器的命令
文本文件	txt，doc	文本数据，文档
文字处理文件	wp，tex，rtf，doc	各种文字处理器格式
库文件	lib，a，so，dll	为程序员提供的程序库
打印或视图文件	ps，pdf，jpg	用于打印或视图的 ASCII 或二进制格式的文件
档案文件	arc，zip，tar	相关的几个文件组成在一个文件中，通常为了归档或存储而被压缩
多媒体文件	mpeg，mov，rm	包含音频或 A/V 信息的二进制文件

图 11.1 通常的文件类型

下面考虑 Apple Macintosh 操作系统。对这种系统，每个文件都有类型如 *text* 或 *pict*。每个文件也有一个创建者属性，用来包含创建它的程序名称。这种属性是由操作系统在调用 create 时设置的，因此系统支持并强制其使用。例如，由字处理器创建的文件会用字处理器名称作为其创建者。当用户用鼠标双击表示该文件的图标来打开文件时，就会自动调用字处理器，并将文件装入以便编辑。

UNIX 系统不提供这种功能，因为它采用**幻数**（magic number）（保存在文件的开始部分）大致表示文件类型：可执行程序、批处理文件（shell **脚本**），postscript 文件等。不是所有文件都具有幻数，所以系统特性不能只根据这种信息。UNIX 也不记录文件创建者的名称。UNIX 确实允许文件名称扩展提示来帮助确定文件内容的类型。这些扩展可以被给定应用程序所使用或忽视，这是由应用程序开发者所决定的。

11.1.4 文件结构

文件类型也可用于表示文件的内部结构。如 11.1.3 小节所述，源文件和目标文件具有一定结构，以适应相应处理程序的要求。而且，有些文件必须符合操作系统所要求的结构。例如，操作系统可能要求可执行文件具有特定结构，以便它能确定将文件装入到哪里以及第一个指令的位置是什么。有的操作系统将这种思想扩展到系统所支持的一组文件结构中，采用特殊操作处理具有这些结构的文件。例如，DEC 的 VMS 操作系统有一个可支持三种文件结构定义的文件系统。

以上讨论中的可支持多个文件结构的操作系统不可避免地出现了一个缺点：操作系统会变大。如果操作系统定义了 5 个不同文件结构，那么它需要包含代码以支持这些文件结构。另外，每个文件可能需要定义成操作系统所支持的某一种类型。如果新应用程序要求

信息按操作系统所不支持的结构来组成，那么就会出问题。

例如，假设一个系统支持两种文件类型：文本文件（由回车和换行所分开的 ASCII 字符）和可执行二进制文件。现在，如果我们（作为用户）想要定义加密文件以保护内容不被未经授权的用户读取，那么会发现两种文件类型均不合适。加密文件不是 ASCII 文本行，而且也不是二进制文件。因此，要么必须绕过或错误地使用操作系统文件类型机制，要么放弃加密方案。

有的操作系统强加（和支持）了最少数量的文件结构。这种方法为 UNIX、MS-DOS 和其他操作系统所采用。UNIX 认为每个文件是由 8 位字节序列所组成的；操作系统并不解释这些位。这种方案提供最大程度的灵活性，但是什么也不支持。每个应用程序必须有自己的代码对输入文件进行合适的解释。当然，所有操作系统必须至少支持一种结构，即可执行文件结构，以便能装入和运行程序。

Macintosh 操作系统也支持最少数量的文件结构。它要求每个文件包括两个部分：**资源叉**（source fork）和**数据叉**（data fork）。资源叉包括用户所感兴趣的信息。例如，它包含程序所显示按钮的标签。国外用户可按照自己使用的语言来重新标识这些按钮，Macintosh 操作系统提供工具以允许对资源叉中的数据进行修改。数据叉包括程序代码或数据，即传统的文件内容。为了在 UNIX 或 MS-DOS 上能实现同样任务，程序员必须要修改和重新编译源程序，除非他创建了用户可修改的数据文件。显然，操作系统支持常用结构是有用的，这能节省程序员大量的劳动。结构太少会使编程不够灵活，然而结构太多会使操作系统过大且会使程序混淆。

11.1.5 内部文件结构

从内部而言，定位文件偏移量对操作系统来说可能是比较复杂的。从第二章得知，磁盘系统通常具有明确定义的块大小，这是由扇区大小所决定的。所有磁盘 I/O 是按块（物理记录）来执行的，且所有块都是同样大小。物理记录大小不太可能刚好与所需逻辑记录大小一样长。而且逻辑记录的长度是可变的。对这个问题的常用解决方法是先将若干个逻辑记录打包，再放入物理记录。

例如，UNIX 操作系统定义所有文件为字节流。每个字节可以从其到文件首（或尾）的偏移量来访问。文件通常会自动将字节打包以存入物理磁盘块，或从磁盘块中解包得到字节（如按需要可能为每块 512 B）。

逻辑记录大小、物理块大小和打包技术决定了多少逻辑记录可保存在每个物理块中。打包可由用户应用程序或操作系统来执行。

由于磁盘空间总是按块来分配的，所以文件的最后一块的部分空间通常会有浪费。如果每块为 512 B，一个大小为 1 949 B 的文件会分配到 4 块（2 048 B）；最后 99 B 就浪费了。按块分配所浪费的字节称为内部碎片，块的大小越大，内部碎片也越大。

11.2 访问方法

文件用来存储信息。当使用时，必须访问和读入这些信息。文件信息可按多种方式来进行访问。有的系统只提供了一种文件访问方式。其他系统，如 IBM 的系统，支持多种访问方式，因此为特定应用选择合适的访问方式是个主要的设计问题。

11.2.1 顺序访问

最为简单的访问方式是**顺序访问**。文件信息按顺序，一个记录接着一个记录地加以处理。这种访问模式最为常用，例如，编辑器和编译器通常按这种方式访问文件。

大量的文件操作是读和写。读操作读取下一文件部分，并自动前移文件指针，以跟踪 I/O位置。类似地，写操作会向文件尾部增加内容，相应的文件指针前移到新增加数据之后（新文件结尾）。文件也可重新设置到开始位置，有的系统允许向前或向后跳过 n 个记录，这里某个整数 n，有时只能为 1。顺序访问如图 11.2 所示。顺序访问基于文件的磁带模型，不但适用于顺序访问设备也适用于随机访问设备。

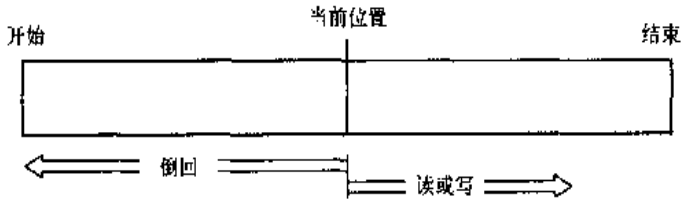


图 11.2 顺序访问文件

11.2.2 直接访问

另一方式是**直接访问**（或**相对访问**）。文件由固定长度的**逻辑记录**组成，以允许程序按任意顺序进行快速读和写。直接访问方式是基于文件的磁盘模型，这是因为磁盘允许对任意文件块进行随机读和写。对直接访问，文件可作为块或记录的编号序列。直接访问文件允许对任意块进行读或写。因此，可读取块 14，再读块 53，最后再写块 7。对于直接访问文件，读写顺序是没有限制的。

直接访问文件可立即访问大量信息，所以极为有用。数据库通常使用这种类型的文件。当有关特定主题的查询到达时，计算哪块包含答案，并直接读取相应块来提供所需信息。

举一个简单的例子，在一个航班订票系统上，可将所有特定航班（如航班 713）的信息，保存在由航班号码所标识的块上。因此，航班 713 的空位数量保存在订票文件的块 713 上。为了存储关于某个更大集合如人的信息，可根据人名计算出一个哈希函数，或者搜索位于内存中的索引以确定需要读和搜索的块。

对于直接访问方式，文件操作必须经过修改从而能将块号作为其参数。因此，有*读 n* 的操作，其中 *n* 是块号，而不是*读下一个*；有*写 n* 的操作，而不是*写下一个*。另外一种方法是像顺序访问一样，保留*读下一个*和*写下一个*，但是增加*定位文件*到 *n*，其中 *n* 是块号。这样，要实现读 *n*，只要*定位到 n* 并再执行*读下一个*。

由用户向操作系统所提供的块号通常为**相对块号**。相对块号是相对于文件开始的索引。因此，文件的第一块的号码是 0，下一块为 1，依此类推，而第一块的真正绝对磁盘地址可能为 14703，下一块为 3192 等。使用相对块号允许操作系统决定该文件放在哪里（称为分配问题），以阻止用户访问不属于其文件的其他文件系统部分。有此系统的相对块号从 0 开始；其他的从 1 开始。

设逻辑记录长度为 L，记录 N 的请求可转换为从文件位置 $L\times(N-1)$ 开始的 L 字节的请求（设第一记录为 $N=1$）。由于逻辑记录为固定大小，所以它也容易读、写和删除记录。

不是所有操作系统都支持文件的顺序访问和直接访问。有的系统只允许顺序文件访问；也有的只允许直接访问。有的系统要求在创建文件时确定究竟是顺序访问还是直接访问，这样的文件只能按照所声明的方式进行访问。然而，对直接访问文件，可容易地模拟顺序访问。可简单地定义一个变量 *cp* 来表示当前位置，以按照图 11.3 所示的方式模拟顺序文件操作。另一方面，在顺序访问文件上，模拟直接访问是极为低效和笨重的。

顺序访问	直接访问的实现
重启	cp=0；
读下一个	read cp；cp=cp+1；
写下一个	write cp；cp=cp+1；

图 11.3 在直接访问文件上模拟顺序访问

11.2.3 其他访问方法

其他访问方式可建立在直接访问方式之上。这些访问通常涉及创建文件索引。**索引**，如同书的索引，包括各块的指针。为了查找文件中的记录，首先搜索索引，再根据指针直接访问文件，以查找所需要的记录。

例如，一个零售价格文件可能列出每个产品的通用商品编码（universal product codes，UPC）及价格。每个记录包括 10 位 UPC 和 6 位价格，所以每个记录为 16 B。如果每个磁盘块有 1 024 B，那么每块存储 64 个记录。一个具有 120 000 个记录的文件可能占有 2 000 块（2 MB）。通过将文件按 UPC 排序，可将索引定义为包括每块的第一个 UPC。该索引有 2 000个条目，每个条目为 10 个数字，共计 20 000 B，因此可保存在内存中。为了查找某一产品的价格，可以（二分）搜索索引。通过这个搜索，可以精确地知道哪个包括所要的记录并访问该块。这种结构允许人们只通过少量 I/O 就能搜索大文件。

对于大文件，索引本身可能太大以致于不能保存在内存中。解决方法之一是为索引文件再创建索引。初级索引文件包括二级索引文件的指针，而二级索引再包括真正指向数据项的指针。

例如，IBM 的索引化顺序访问方法（ISAM）就使用小的主索引文件以指向二级索引的磁盘块。二级索引块再指向实际文件块。该文件按定义的键排序。当查找一特定项时，首先对主索引进行二分搜索，以得到二级索引的块号。读入该块，再通过二分搜索以查找包括所要记录的块。最后，按顺序搜索该块。这样，通过记录关键字最多不超过两次直接访问就可定位记录。图 11.4 显示了一个类似情况，这是由 VMS 索引和相对文件所实现的。

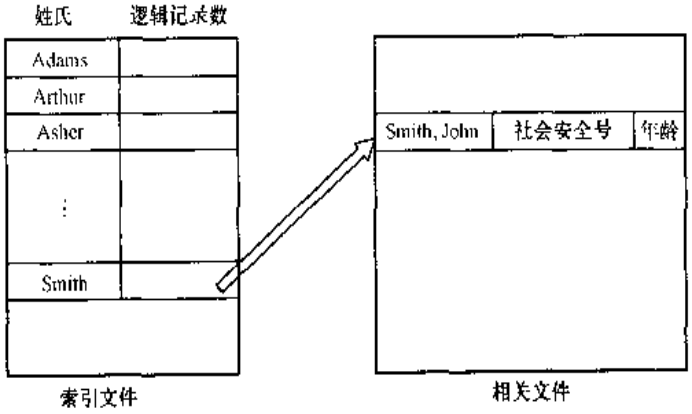


图 11.4 索引文件和相关文件的例子

11.3 目录结构

计算机的文件系统可以非常大。有的系统在数千吉磁盘上保存了数以百万计的文件。为了管理所有这些数据，需要组织它们。这种组织通常分为两部分。第一，磁盘分为一个或多个*分区*，或称为*小型磁盘*（IBM 的说法）或*卷*（PC 或 Macintosh 的说法）。通常，每个系统磁盘至少包括一个分区，这是用来保存文件和目录低层结构。有时，分区在单个磁盘内提供多个独立区域，每个区域可作为独立存储设备；有时，系统允许分区大于一个磁盘，以便将多个磁盘组织成一个逻辑结构。这样，用户只需要关心逻辑目录和文件结构，并可完全忽略为文件分配空间的问题。因为这个原因，分区可作为虚拟磁盘。分区能存储多个操作系统，以允许选择启动和运行一个系统。

第二，每个分区都包含了存储在分区中的文件的信息。这种信息保存在**设备目录或卷内容表**中。设备目录（通常简称为目录）记录分区上所有文件的各种信息，如名称、位置、大小和类型等。图 11.5 显示了典型的文件系统的组成。

目录可看做系统表，它能将文件名称转换成目录条目。如果采用这种观点，那么目录可按许多方式来加以组织。对于目录，需要能够插入条目、删除条目、搜索给定条目、列出所有

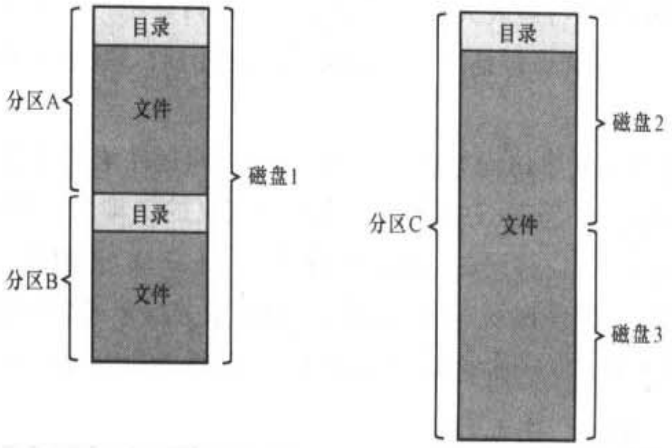


图 11.5 典型的文件系统组成

目录条目。第十二章将讨论用于实现目录结构的各种数据结构。这里,讨论用于定义目录系统的逻辑结构的若干方案。在考虑特定目录结构时,需要记住目录相关操作:

- **搜索文件**:需要能够搜索目录结构以查找特定文件的条目。因为文件具有符号名称,且类似的名称可能表示文件之间的关系,所以要能查找文件名和某个模式相匹配的所有文件。
- **创建文件**:可以创建新文件并增加到目录中。
- **删除文件**:当不再需要文件时,可以从目录中删除它。
- **列出目录**:要能够在一个目录中列出文件,还要在列表中为每个文件列出目录条目的内容。
- **重命名文件**:因为文件名称可向用户表示其内容,当文件内容和用途改变时名称必须改变。重新命名文件也允许改变该文件在目录结构中的位置。
- **跟踪文件系统**:可能希望访问每个目录和每个目录的每个文件。为了可靠,定期备份整个文件系统的内容和结构是个很好的方法。这种备份通常将所有文件复制到磁带上。这种技术提供了备份拷贝以防止系统出错或者当文件不再需要时使用。在这种情况下,文件被复制到磁带上,该文件的原来占用磁盘空间可以释放以供其他文件所用。

在 11.3.1 小节到 11.3.5 小节,将讨论定义目录逻辑结构的常用方案。

11.3.1 单层目录

最简单的目录结构是单层目录。所有文件都包含在同一目录中,其特点是便于支持和理解(图 11.6)

然而,在文件类型增加时或系统有多个用户时,单层目录有严重限制。由于所有文件位于同一目录,它们必须具有惟一名称。如果两个用户称其数据文件为 *test*,那么就违背了惟一名称规则。例如,在一个编程班级中,有 23 个学生将其第 2 个作业称为 *prog2*;而另 11 位则称其为 *assign2*。虽然文件名称通常经过选择以反映文件内容,但是它们的长度通常有限

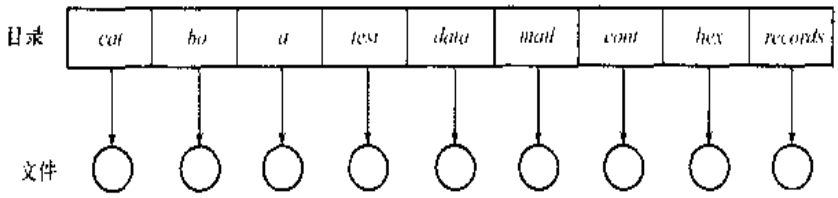


图 11.6 单层目录

制。MS-DOS 操作系统只允许 11 个字符的文件名;UNIX 允许 255 个字符。

随着文件数量增加,单层目录的单个用户会发现难以记住所有文件的名称。通常,一个用户在一个计算机系统上有数百个文件,在另外一个系统上也有同样数量的其他文件。在这种环境上,记住如此之多的文件是令人畏缩的。

11.3.2 双层目录

单层目录通常会在不同用户之间引起文件名称的混淆。标准解决方法是为每个用户创建独立目录。

对于双层目录结构,每个用户都有自己的**用户文件目录**(user file directory,UFD)。每个 UFD 都有相似的结构,但只列出了单个用户的文件。当一个用户作业开始执行或一个用户注册时,就搜索系统的**主文件目录**(master file directory,MFD)。通过用户名或账号可索引 MFD,每个条目指向用户的 UFD(图 11.7)。

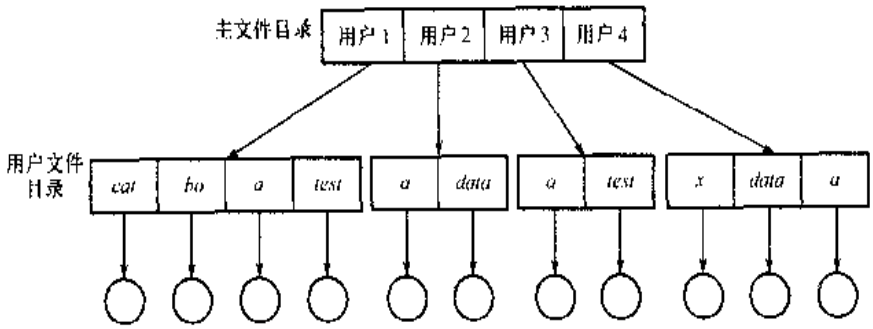


图 11.7 双层目录结构

当一个用户引用特定文件时,只需搜索他自己的 UFD。因此,不同用户可拥有具有相同名称的文件,只要每个 UFD 内的所有文件名称惟一即可。

当一个用户创建文件时,操作系统只搜索该用户的 UFD 以确定具有同样名称的文件是否存在。当删除文件时,操作系统只在局部 UFD 中对其进行搜索;因此,它并不会删除另一用户的具有相同名称的文件。

用户目录本身必须根据需要加以创建和删除。这可通过运行一个特别的系统程序,再加上合适用户名称和账号信息。该程序创建新的 UFD,并在 MFD 中为之增加一项。该程序可能只能由系统管理员执行。用户目录的磁盘空间分配可以采用第十二章所讨论的技术

来处理。

虽然双层目录解决了名称冲突问题，但是它仍有缺点。这种结构有效地对用户加以隔离。这种隔离在用户需要完全独立时是优点，但是在用户需要在某个任务上进行合作和访问其他文件时却是个缺点。有的系统简单地不允许本地用户文件被其他用户所访问。

如果允许访问，那么一个用户必须能够指定另一用户目录内的文件。为了惟一指定位于两层目录内的特定文件，必须给出用户名和文件名。双层目录可作为高度为 2 的树或倒置树。树根为 MFD。其直接后代为 UFD。UFD 的后代为文件本身。文件为树的叶。指定用户名和文件名就在树中，定义了从根(MFD)到叶(指定文件)的路径。因此，用户名和文件名定义了路径名。系统内的每个文件都有路径名。为了惟一指定文件，用户必须知道所要访问的文件的路径名。

例如，如果用户 A 需要访问其自己的名称为 *test* 的文件，那么他可简单地称之为 *test*。然而，为了访问用户 B(其目录名为 *userb*)的名称为 *test* 的文件，他必须称之为 */userb/test*。每个系统都有特定语法以指定不属于用户自己目录内的文件。

指定文件分区还需要额外语法。例如，在 MS-DOS 中，分区用一个字母加上冒号来指定。因此，指定文件可能为 C:\userb\test。有的系统做得还要细，对指定的分区名、目录名和文件名分别加以区分。例如，在 VMS 中，文件 login. com 可能表示为 u:[sst, jdeck] login. com;1，其中 *u* 为分区名，*sst* 为目录名，*jdeck* 为子目录名，*login. com* 为文件名，1 为版本号。其他系统只是简单地将分区名作为目录名。所给的第一个名称为分区名，其他的为目录名和文件名。例如，*/u/pbg/test* 可能表示分区 *u*、目录 *pbg* 和文件 *test*。

这种情况的一个特例是关于系统文件。作为系统一部分的程序如加载器、汇编程序、编译程序、工具程序、库等通常定义为文件。当向操作系统发出适当命令时，这些文件会被加载程序读入，然后开始执行。许多命令解释程序只是将命令作为所要装入和执行的文件名。如果目录系统按现在这样定义，那么程序文件只能在当前 UFD 中搜索。解决方法之一是将系统文件复制到每个 UFD。然而，复制所有系统文件会浪费巨大空间。(如果系统文件要 5 MB，那么 12 个用户需要 5×12=60 MB 以存储系统文件的拷贝。)

标准的解决办法是稍稍修改搜索步骤。定义一个特殊的用户目录，它包括所有系统文件(例如，用户 0)。当给定名称文件需要装入时，操作系统首先会搜索本地 UFD。如果查找到，那么就使用。如果查找不到，那么系统会自动搜索特殊用户目录(它包括系统文件)。当给定一文件时，搜索的一系列目录称为**搜索路径**(search path)。这种方法可以加以扩展，以致于搜索路径可包括没有任何限制的目录链表。这种方法被 UNIX 和 MS-DOS 所使用。

11.3.3 树型目录

一旦理解如何将双层目录作为两层树看待，那么将目录结构扩展为任意高度的树就显得自然了(图 11.8)。这种推广允许用户创建自己的子目录，相应地组织文件。例如，MS-

DOS 系统就是按树组织的。事实上，树是最为常用的目录结构。树有根目录。系统内的每个文件都有惟一路径名。路径名是从根经过所有子目录再到指定文件的路径。

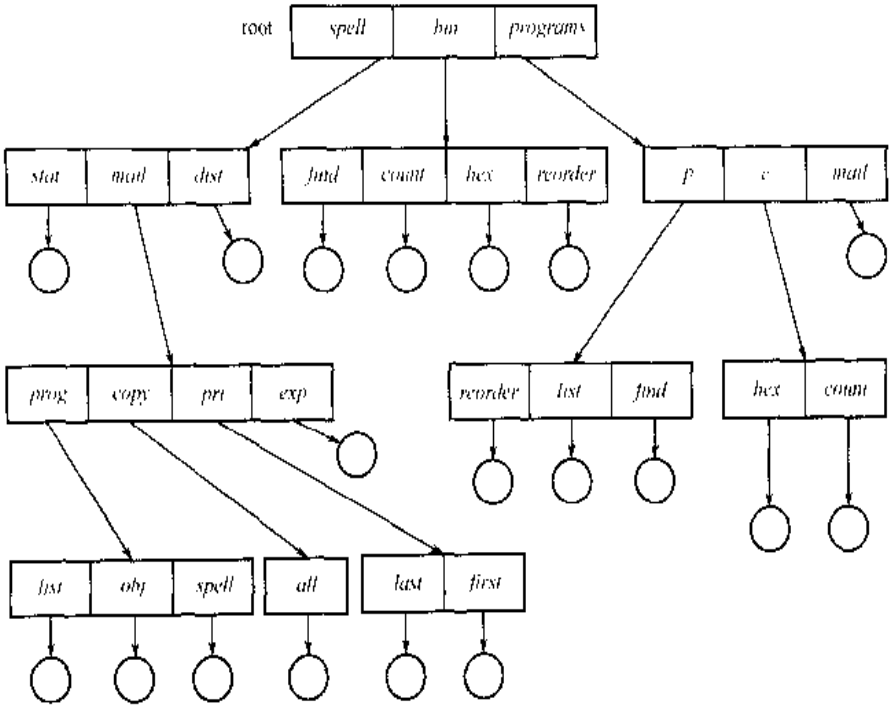


图 11.8 树形目录结构

目录(或子目录)包括一组文件和子目录。一个目录只不过是一个需要按照特定方式访问的文件。所有目录具有同样的内部格式。每个目录条目都用一位来定义其为文件或目录。创建和删除目录条目需要使用特定的系统调用。

在通常情况下，每个用户都有一个当前目录。**当前目录**包括用户当前感兴趣的绝大多数文件。当需要引用一个文件时，就搜索当前目录。如果所需文件不在当前目录中，那么用户必须指定路径名或改变当前目录为包括所需文件的目录。为了改变目录，用户可使用系统调用以重新定义当前目录，该系统调用需要有一个目录名作为参数。这样，用户需要时就可以改变当前目录。从当前改变目录系统调用直到下次改变，系统调用 open 搜索当前目录以打开所指定的文件。

用户的初始当前目录是在用户进程开始时或用户登录时指定的。操作系统搜索记账文件(或其他预先定义的位置)以得到该用户的相关条目(以便于记账)。记账文件有用户初始目录的指针(或名称)。该指针可复制到局部变量以指定用户初始的当前目录。

路径名有两种形式：绝对路径名或相对路径名。**绝对路径名**从根开始并给出路径上的目录名直到所指定的文件。**相对路径名**从当前目录开始定义一个路径。例如在图 11.8 所示树型结构文件系统中，如果当前目录是 *root/spell/mail*，那么相对路径名 *prt/first* 与绝对路径名 *root/spell/mail/prt/first* 指向相对的文件。

允许用户定义其自己的子目录结构可以使其能按一定结构组织文件。这种结构可以是按不同主题来组织文件(例如,可创建一个目录以包括本书的内容);或按不同信息类型来组织文件(例如,目录 *program* 可以包含源程序;而目录 *bin* 可存储所有二进制文件)。

对于树形结构目录,一个有趣的策略决定是如何处理删除目录。如果目录为空,那么可简单地加以删除。然而,假如要删除的目录不为空,而是包括多个文件或子目录,那么可有两个选择。有的系统,如 MS-DOS,如果目录不为空就不能删除。因此,为了删除一个目录,用户必须首先删除其中的内容。如果有子目录存在,那么必须先删除其子目录的内容。这种方案可能会导致大量的工作。

另一种方法,如 UNIX 的 rm 命令,提供了选择:当需要删除一个目录时,所有该目录的文件和子目录也可删除。这两种方法的实现均不难,这是策略的问题。后一种策略更方便,但是更危险,因为用一个命令可以删除整个目录结构。如果错误地使用了这个命令,那么就必须从备份磁带中恢复大量文件和目录。

对于树形结构目录系统,用户除了能访问自己的文件外,还能访问其他用户的文件。例如,用户 B 通过指定路径名能够访问用户 A 的文件。用户 B 可使用绝对路径名或相对路径名。另外,用户 B 可改变其当前目录为用户 A 的目录,而直接用文件名来访问文件。有的系统还允许用户定义其自己的搜索路径。这时,用户 B 可以定义其搜索路径按顺序为:(1) 其本地目录,(2) 系统文件目录,(3) 用户 A 的目录。只要用户 A 的文件不与本地或系统文件相冲突,那么它就可简单地用其名称加以引用。

树形结构目录的文件路径可比两层结构目录的要长。为了让用户不必记住这些长路径就能访问程序,*Macintosh* 操作系统自动搜索可执行程序。它维护一个文件,称为*桌面文件*(Desktop File),用以包括它所看到的所有可执行程序的名称和位置。当系统增加了一个新硬盘或软盘,或网络访问时,操作系统会遍历目录结构,从而查找设备上的可执行程序并记录相关信息。这种机制允许以前所述的双击执行功能。当双击一个文件时,会读入其创建者属性,并搜索桌面文件以匹配查找。一旦查找到,就用所击文件作为输入而启动相应的可执行程序。微软公司 Windows 操作系统系列(95,98,NT,2000)支持扩展的两层目录结构,其设备和目录名用驱动器字母表示。

11.3.4 无环图目录

考虑一下两个程序员在进行一个合作项目。与该项目相关的文件可保存在一个子目录中,以区分两程序员的其他项目和文件。但是,由于两程序员都负责该项目,所以都希望该子目录在他自己的目录内。这种共同子目录应该共享。共享目录或文件可同时位于文件系统的两(或多)处。

树形结构禁止共享文件和目录。**无环图**(acyclic graph)允许目录含有共享子目录和文件(图 11.9)。*同一*文件或子目录可出现在两个不同目录中。无环图是树形结构目录方案的

自然扩展。

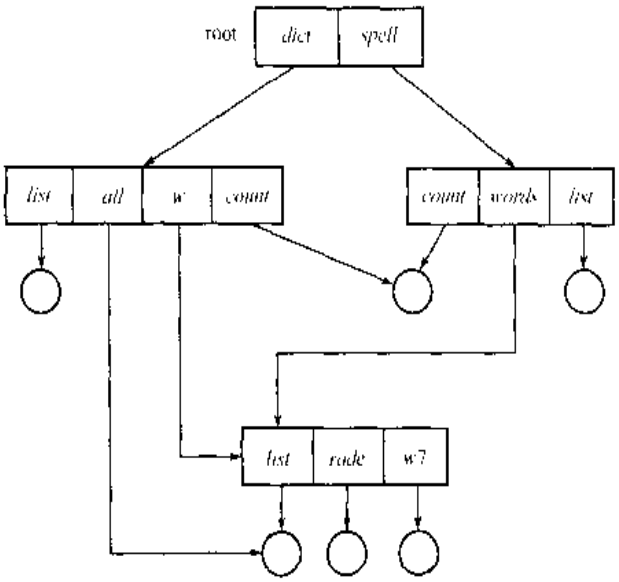


图 11.9 无环图目录结构

共享文件(或目录)不同于文件拷贝。如果有两个拷贝,每个程序员看到的是拷贝而不是原件;但是如果一个程序员改变了文件,那么这一改变不会出现在另一程序员的拷贝中。对于共享文件,只存在一个真正文件,所以任何改变都会为其他用户所见。共享对于子目录尤其重要:由一个用户创建的文件可自动出现在所有共享目录。

当人们作为一个组工作时,所共享的文件可放在一个目录中。所有组员的 UFD 可以将该共享文件目录作为其子目录。即使对于单个用户,也可能会要求有的文件出现在不同子目录中。例如,某个特定项目的程序不但可位于所有程序目录中,也可位于该项目的目录中。

实现共享文件和目录有许多方法。一个为许多 UNIX 系统所采用的常用方法是创建一个称为链接的新目录条目。**链接**实际上是另一文件或目录的指针。例如,链接可用绝对路径或相对路径名称来实现。当需要访问一个文件时,就搜索目录。如果目录条目标记为链接,那么就可获得真正文件(或目录)的名称。链接可以通过使用路径名定位真正的文件来获得解析。链接可通过目录条目格式(或通过特殊类型)而容易地加以标识,它实际上是具有名称的间接指针。在遍历目录树时,操作系统忽略这些链接以维护系统的无环结构。

实现共享文件的另一常用方法是简单地在共享目录中重复所有(被)共享文件信息。因此两个目录条目完全相同。链接显然不同于原来的目录条目;因此,两者是不相同的。然而,重复目录条目会使原来的文件和复制品无法区分。复制目录条目的一个主要问题是在修改文件时要维护一致性。

无环结构目录比树型结构目录更加灵活,但是也更为复杂。有些问题必须仔细考虑。现在一个文件可有多个绝对路径名。这样,不同文件名可能表示同一文件。这类似于程序设计语言的别名问题。如果试图遍历整个文件系统,如查找文件,计算所有文件的统计数

据，复制所有文件到备份存储，那么这个问题就重要了，这是因为人们不希望多次重复地遍历共享目录。

另一问题是删除。分配给共享文件的空间什么时候可删除和重新使用？一种可能是每当用户删除文件时就删除文件，但是这样会留下悬挂指针指向不再存在的文件。更为糟糕的是，这些剩余文件指针可能包括实际磁盘地址，而该空间可能又被其他文件使用，这样悬挂指针就可能指向其他文件的中间部分。

对于采用符号链接实现共享的系统，这种情况较为容易处理。删除链接并不需要影响原文件；而只是删除链接。如果文件条目本身被删除，那么文件空间就释放，并使链接指针无效。可以搜索这些链接并删除它们，但是除非每个文件都保留相关链接列表，否则这种搜索可能会费时。或者，可以暂时不管这些指针，直到试图使用它们为止。到时，可确定由链接所给定名称的文件不再存在，从而不能解析链接名称；这种访问与其他非法文件名一样处理。（在这种情况下，系统设计人员需要仔细考虑如下问题：当一个文件删除，而在使用其符号链接之前，另一个具有同样名称的文件创建了。）对于 UNIX，当删除文件时，其符号链接并不删除，需要由用户认识到原来文件已被删除或替换。微软公司 Windows（所有版本）也使用同样方法。

删除的另一方法是保留文件直到删除其所有引用为止。为了实现这种方法，必须有一种机制来确定最后文件引用已删除。可以为每个文件保留一个引用列表（目录条目或符号链接）。在建立一个目录条目的链接或复制时，需要将新条目增加到文件引用列表。当删除链接或目录条目时，删除列表上的相应条目。当其文件引用列表为空时，就删除文件本身。

这种方法的麻烦是可能会出现可变的、并可能很大的文件引用列表。然而，并不需要保留整个文件列表，只需要保留文件引用的数量。一个新链接或目录条目增加引用计数；删除链接或条目会降低计数。当计数为 0 时，就能删除该文件；对它没有多余的引用。UNIX 操作系统对非符号链接（或**硬链接**）采用了这种方法，即在文件信息块（或 *inode*，参见附录 A.7.2）中保留一个引用计数。通过禁止对目录的多重引用，维护了无环图结构。

为了避免这些问题，有的系统不允许共享目录和链接。例如，对 MS-DOS，目录结构是树结构而不是无环图。

11.3.5 通用图目录

采用无环图结构的一个特别重要的问题是要确保没有环。如果从两层目录开始，并允许用户创建子目录，那么就产生了树结构目录。可以容易地看出对已存在的树结构目录简单地增加新文件和子目录将会保持树结构性质。然而，当对已存在的树结构目录增加链接时，树结构就破坏了，产生了简单的图结构（图 11.10）。

无环图的主要优点是可用简单算法来遍历图并确定是否存在文件引用。主要是因为性能原因，希望避免多次遍历无环图的共享部分。如果搜索一共享子目录以查找特定文件，但

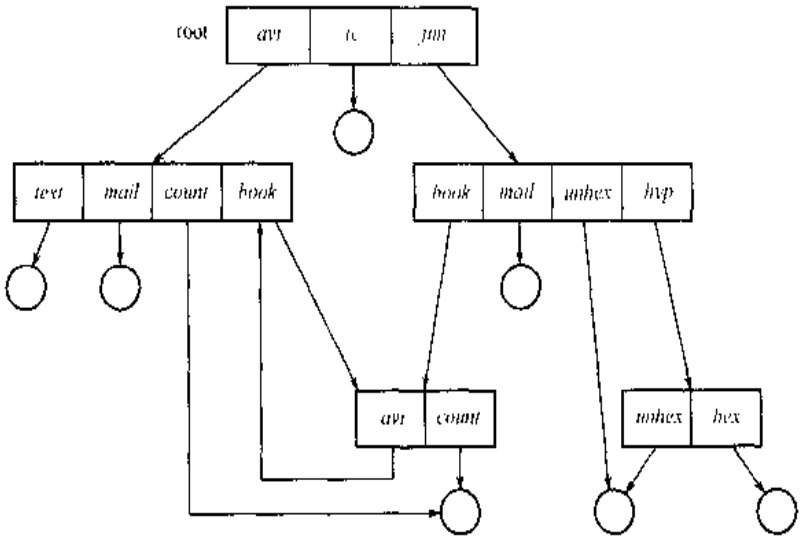


图 11.10 通用图目录

没有找到，那么需要避免再次搜索该子目录；第二次搜索只是浪费时间。

如果在目录中允许有环存在，那么无论是从正确性还是从性能角度而言，同样需要避免多次搜索同一部分。一个设计欠佳的算法可能会无穷地搜索循环而不终止。一个解决办法是可以随意地限制在搜索时所访问目录的次数。

当试图确定一个文件什么时候可删除时，会存在另一个类似的问题。与无环结构目录一样，引用计数为 0 意味着没有对文件或目录的引用，那么文件可删除。然而，当存在环时，即使不可能存在对文件或目录的引用，其引用计数可能不为 0。这种异常是由于在目录中可能存在自我引用的原故。在这种情况下，通常需要使用垃圾收集方案以确定什么时候可删除最后引用，释放其空间。垃圾收集涉及遍历整个文件系统，并标记所有可访问的空间。然后，第二次将所有没有标记的部分收集到空闲空间链表上。（一个类似标记步骤可用于确保只对文件系统内的文件或目录进行一次遍历或搜索）。然而，用于磁盘文件系统的垃圾收集是极为费时的，因此很少使用。

由于图中可能存在环，所以垃圾收集是必要的。因此，无环图结构更加便于使用。问题是如何在创建新链接时要避免环。如何知道什么时候新链接会形成环呢？有些算法可用于检测图中的环；然而，当图位于磁盘上时，这些算法的计算极为费时。处理目录和链接的另一个类似问题是在遍历目录时迂回链接。这样，避免了环，且没有其他开销。

11.4 文件系统安装

如同文件使用前必须要打开，文件系统在被系统上的进程使用之前必须安装（mount）。具体地说，目录结构要建立在多个分区上就必须安装这些分区以使其可用。

安装步骤相对简单。操作系统需要知道设备名称以及在哪里安装文件系统。通常，安

装点(mount point)为空目录,以便于安装文件系统。例如,在 UNIX 中,包括用户主目录的文件系统可安装在*/home*;这样,在访问该文件系统中的目录结构时,只需要在目录名前加上*/home* 如*/home/jane* 即可。将文件系统安装在*/users* 可使路径名*/users/jane* 指向同一目录。

然后,操作系统验证设备是否包含一个有效文件系统。验证是这样进行的:通过设备驱动程序读入设备目录,验证目录是否具有期望的格式。最后,操作系统在其目录结构中记录如下信息:一个文件系统已安装在给定安装点上。这种方案允许操作系统遍历其目录结构,并根据需要可在文件系统之间进行切换。

为了说明文件系统的安装,考虑如图 11.11 所示的文件系统,其中三角形表示所感兴趣的目录子树。图 11.11(a)表示一个已有文件系统;而图 11.11(b)表示一个未安装的驻留在*/device/dsk* 上的文件系统。这时,只有现有文件系统上的文件可被访问。图 11.12 表示把*/device/dsk* 分区安装到*/uesrs* 后的文件系统的情况。如果该分区被拆装,那么文件系统就恢复到如图 11.11 所示的情况。

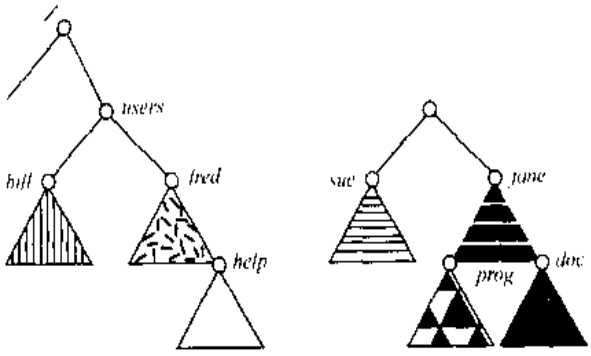


图 11.11 文件系统

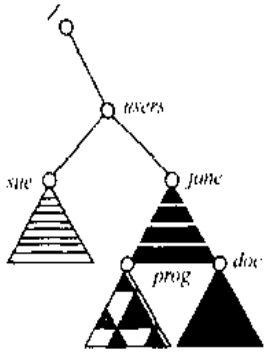


图 11.12 安装点

系统利用语义可以清楚地表达功能。例如,系统可能不允许在包含文件的目录上进行安装,或者使所安装的文件系统在目录中可用,并且使目录中已存的文件不可见直到文件系统被卸载,文件系统的卸载会终止文件系统的使用,并且允许访问目录中原来的文件。另一个例子是,有的系统可允许对同一文件系统在不同的安装点上进行多次重复安装,或者只允许对一个文件系统安装一次。

考虑一个 Macintosh 操作系统的文件系统安装。当系统首次碰到磁盘时(硬盘在启动时查找到,软盘在插入时可被发现),Macintosh 操作系统会查找设备上的文件系统。如果找到,那么就会自动安装在根目录下,并在屏幕上增加一个标有文件系统名称的目录图标。这样,用户按一下图标,就能显示所安装的文件系统。

微软公司 Windows 操作系统系列(95,98,NT,2000)维护一个扩展的两层目录结构,并用驱动器字母表示设备和分区。分区具有通常图结构。特定文件路径形式是 *drive-letter*:\

path\to\file。这些操作系统在启动时自动地发现设备并安装所有文件系统。有的系统，如UNIX，安装命令是显式的。系统配置文件包括一系列设备和安装点，以便在启动时自动安装，也可手动进行其他安装。

文件系统安装将在12.2.2小节和附录A.7.5中进一步讨论。

11.5 文件共享

在前面讨论了文件共享的动机和允许用户共享文件所碰到的一些困难。对于那些希望合作并且希望减少为达到一个计算目标所需要的工作的用户来说，文件共享很有用。因此，不管有什么困难，面向用户的操作系统必须满足共享文件的需要。

在本节，研究文件共享的其他问题。第一是多用户和可能的共享方式的问题。第二，当多用户允许共享文件时，需要将共享扩展到多个文件系统如远程文件系统，这是个挑战。最后，对共享文件，冲突操作可以有多种解释。例如，如果多个用户对一个文件进行写，那么所有写都被允许吗？或者操作系统应该保护用户操作不受其他用户的影响？11.5.3小节将讨论一致性语义。

11.5.1 多用户

当一个操作系统支持多个用户时，文件共享、文件名称和文件保护问题就尤其突出了。对于允许用户共享文件的目录结构，系统必须控制文件共享。系统可缺省地允许一个用户访问其他用户的文件，也可要求一个用户明确授予文件访问权限。下面，将讨论引起访问控制和保护的问题。

为了实现共享和保护，多用户系统必须要比单用户系统维护更多的文件和目录属性。虽然过去有许多方法，但是现在绝大多数系统都采用了文件/目录拥有者（或用户）和组的概念。拥有者可以改变属性、授权访问、拥有文件和目录最高控制权。文件用户组属性用于定义对文件拥有相同访问权限的用户子集。例如，在UNIX系统中，一个文件的拥有者可执行所有操作，文件组的成员只能执行这些操作的子集，而所有其他用户可能只能执行另一操作子集。到底哪些操作组成员能执行，而哪些操作其他成员能执行是由文件拥有者定义的。有关更多权限属性的细节，参见下一节。

绝大多数系统通过管理用户名和相关**用户标识**（user ID，UID）的链表，来实现拥有者属性。对于Windows NT，这称为**安全ID**（Secure ID，SID）。这些数值对于每个用户来说是惟一的。当用户登录到系统，鉴别步骤会确定用户的合适ID。该用户ID与所有该用户的进程和线程相关联。当这些ID需要为用户可读时，它们可通过用户名称链表而转换成用户名。同样，组功能也可通过组名和组ID的系统链表来加以实现。每个用户可属于一个或多个组，这由操作系统设计所定。用户的组ID也会与其所有进程和线程相关联。

一个文件或目录的拥有者 ID 和组 ID 与其他文件属性一起保存。当用户请求文件操作时，用户 ID 可与拥有者属性相比较以确定所请求的是否是文件拥有者。同样，可比较组 ID。比较结果表示可使用哪些权限。这样系统再用这些权限来检查所请求的操作，以决定是允许还是拒绝。

进程的用户信息也可用于其他目的。当一个进程试图与另一进程相交互时，根据操作系统的设计，用户信息可决定交互的结果。例如，一个进程可能试图终止另一个进程，使另一进程后台运行，降低另一进程的优先权。如果两个进程的拥有者相同，那么可能允许这种操作，否则会失败。如果用户为特权用户，那么这种操作可能也被允许。

许多系统有多个局部文件系统，包括单个磁盘的分区或多个磁盘的多个分区。在这种情况下，只要文件系统已安装，那么 ID 检查和权限匹配就简单了。

11.5.2 远程文件系统

网络（第十五章）的出现允许在远程计算机之间进行通信。网络允许在校园范围内或全世界范围内进行资源共享。一个重要共享资源是数据（按文件形式）。随着网络和文件技术的发展，文件共享方式也不断改变。采用第一种实现方式，用户通过程序如 ftp，可实现在机器之间进行文件的人工传输。采用第二种实现方式**分布式文件系统**（DFS），远程目录可从本机上直接访问。采用第三种方法万维网（这有点回到了第一种），可用浏览器获取对远程文件的访问，其单个操作（基本上是 FTP 的包装）用于传输文件。

ftp 可用于匿名访问和鉴别访问。**匿名访问**允许用户在没有远程系统账号的情况下，传输文件。万维网几乎总是采用匿名文件交换。DFS 在访问远程文件的机器和提供文件的机器之间提供了更加紧密的结合。这种结合增加了复杂性，对此将在本节加以讨论。

1. 客户机－服务器模型

远程文件系统允许一台计算机安装一台或多台远程机器上的一个或多个文件系统。在这种情况下，包含文件的机器称为服务器，需要访问文件的机器称为客户机。对于网络机器，客户机－服务器关系是常见的。通常，服务器声明一个资源可为客户机所用，并精确地说明是哪种资源（此时为哪些文件）和哪些客户。文件通常按分区或子目录级别来加以说明。根据客户机－服务器关系的实现，一台服务器可服务多个客户机，而一台客户机可使用多个服务器。

客户标识更为困难。客户可通过其网络名称或其他标识符如 IP 地址来指定，但是这些可能被欺骗（或模仿）。未经验证的客户可能欺骗服务器以使其认为该客户是验证过的，这样它就可获得访问。更为安全的解决方案是客户通过加密密钥向服务器进行安全验证。然而，安全也带来了许多挑战，包括确保客户机和服务器的兼容性（它们必须使用相同加密算法）和安全密钥交换（被截的密钥可允许未经验证客户进行访问）。这些问题本身太过困难，所以绝大多数情况下使用不太安全的验证。对于 UNIX 及其网络文件系统（NFS），验证缺

省是通过客户网络信息进行的。采用这种方案，用户 ID 在客户机和服务器上要匹配。否则，服务器不能确定对文件的访问权限。

考虑这样一个例子，用户 ID 在客户机上为 1000，而在服务器上为 2000。来自客户机并试图访问服务器上特定文件的请求可能不会得到正确处理，这是因为服务器认为用户 ID1000访问文件，而不是真正的用户 ID 2000。根据不正确的验证信息，访问可以允许或拒绝。服务器必须相信客户提供正确的用户 ID。NFS 协议允许多对多关系。即，许多服务器可为多个客户提供文件。事实上，一个机器不但可以对某些 NFS 客户来说是服务器，还可以是其他 NFS 服务器的客户。

一旦安装了远程文件系统，那么文件操作请求会通过网络按照 DFS 协议发送到服务器。通常，一个文件打开请求与其请求的用户 ID 一起发送。然后，服务器应用标准访问检查以确定该用户是否有权限按所请求的模式访问文件。请求可能被允许或拒绝。如果允许，那么文件句柄就返回给客户应用程序，这样该程序就可执行读、写和其他文件操作。当访问完成时，客户会关闭文件。操作系统可采用与本地文件系统安装相同的语义，也可采用不同的语义。

2. 分布式信息系统

为了便于管理客户机－服务器服务，**分布式信息系统**，也称为**分布式命名服务**，用来提供用于远程计算所需的信息的统一访问。**域名系统**(DNS)为整个因特网(包括万维网)提供了主机名称到网络地址的转换。在 DNS 发明和广泛使用之前，包含同样信息的文件是通过电子邮件或 ftp 在网络机器之间进行交流的。这种方法不可扩展。DNS 将在 15.4.1 小节中讨论。

其他分布式信息系统为分布应用提供了用户名称/口令/用户 ID/组 ID。UNIX 系统有很多分布式信息方法。Sun Microsystem 引了入黄页(后来改名为**网络信息服务**(Network Information System，NIS))，绝大多数业界都采用了它。它将用户名、主机名、打印机信息等加以集中管理。然而，它使用了不安全的验证方法，如发送未加密的用户密码以及用 IP 地址来标识主机。SUN 的 NIS+是 NIS 的更为安全的升级，但是也更为复杂且并未得到广泛使用。

对于 Microsoft 网络(CIFS)，网络信息与用户验证信息(用户名和密码)一起，用以创建网络登录，这可以被服务器用来确定是否允许或拒绝对所请求文件系统的访问。要使验证有效，用户名必须在机器之间匹配(如 NFS)。微软公司采用两个分布命名结构为用户提供单一名称空间。旧的技术命名是域。从 Windows 2000 之后采用了称为活动目录的新技术。一旦建立，分布式命名工具可供客户机和服务器使用来验证用户。

现在，业界正在采用**轻量级目录访问协议**(lightweight directory-access protocol，LDAP)作为安全的分布命名机制。事实上，活动目录是基于 LDAP 的。Sun Microsystem 的 Solaris 8 允许 LDAP 用于用户验证和获取系统范围内的信息如打印机等。如果 LDAP

应用成功，那么一个分布 LDAP 目录可用于存储一个企业内的所有计算机的所有用户和资源。这种结果是单一密码签入：用户只需要输入一次验证信息，就可访问企业内的所有计算机。通过将分布于每个系统上的各种文件信息和不同分布信息服务集中起来，也减轻了系统管理的工作负担。

3. 故障模式

本地文件系统可能因各种原因而出错，如包含文件系统的磁盘出错，目录结构或其他磁盘管理信息（总称为**元数据**（metadata））的损坏、磁盘控制器故障、电缆故障或主机适配器故障。用户或系统管理员的错误也可能导致文件丢失，或整个目录或分区被删除。许多这类错误都会引起主机关闭、显示错误条件，需要人工干预以修补。

有的故障不会引起数据丢失或数据可用性丢失。**冗余阵列磁盘机**（redundant arrays of inexpensive disks，RAID）能防止因磁盘故障而引起的数据丢失。RAID 将在 14.5 节中讨论。

远程文件系统具有更多的故障模式。由于网络系统的复杂性和远程机器间所需的交互，会存在更多影响远程文件系统的正确操作的问题。在网络情况下，两个机器间的网络可能被中断。这可能是由于硬件故障或配置错误，或有关主机的网络实现出现问题。虽然有的网络有内置的弹性，包括在主机之间有多个路径，但是还有很多网络没有这种功能。任何一个故障都会中断 DFS 命令的交流。

考虑一个客户在使用远程文件系统。它安装了远程文件系统，并可能打开了源自远程主机的文件；在许多动作中，它可能执行目录查找以打开文件，读、写文件数据和关闭文件。现在，假设网络断开、服务器故障或服务器定期关机等，以致于不可访问远程文件系统。这种情况很常见，所以用户不应该将它作为本地文件系统故障一样来处理。

但是，系统应该终止对故障服务器的所有操作，或者延迟操作直到服务器再次可用为止。这种故障语义是由远程文件系统协议所定义和实现的。终止所有操作会导致用户丢失数据和耐性。绝大多数 DFS 协议强制或允许延迟对远程主机的文件系统操作，以寄希望于远程主机会再次可用。

对于恢复这种故障，在客户机和服务器之间可能需要一定的状态信息。如果服务器故障，那么它必须知道哪些文件系统已输出、哪些已经被远程安装了，哪些文件被打开了。NFS 采用了一种简单方法，以实现无状态 DFS。简单地说，它假定除非已经安装了远程文件，并打开了文件，否则客户不会请求有关文件读或写。NFS 协议携带所有需要的信息，以定位适当的文件并执行所请求的文件操作。同样，它并不跟踪哪个客户安装了远程文件系统，而是假定客户所请求的操作是合法的。这种无状态方法使 NFS 具有弹性并容易实现，但是它并不安全。例如，在没有进行必要的安装请求和许可检查时伪造的读或写请求会被 NFS 服务器允许。

11.5.3 一致性语义

一致性语义是评估文件系统对文件共享支持的一个重要准则。这是描述了多用户同时

访问共享文件时的语义。特别地，这些语义规定了一个用户所修改的数据何时对另一用户可见。这种语义通常是由文件系统代码来实现的。

一致性语义与第七章的进程同步算法直接相关。然而，因为磁盘和网络的巨大延迟和较慢的传输率，这些复杂算法似乎并不适合文件 I/O 操作方面。例如，对远程磁盘执行一个原子操作可能需要多个网络通信或多个磁盘读写或者两者都要。试图实现完整功能集合的系统往往性能欠佳。一个成功实现了复杂共享语义的文件系统是 Andrew 文件系统。

在下面的讨论中，假定一个用户所进行的同一文件的一系列操作(即，读和写)是包括在 open 和 close 操作之间。在 open 和 close 操作之间的这一系列访问称为文件会话。为了说明语义概念，简要介绍几个例子。

11.5.4 UNIX 语义

UNIX 文件系统使用了如下的语义一致性：

- 一个用户对已经打开的文件进行写操作，可以被同时打开同一文件的其他用户看见。
- 有一种共享模式允许用户共享文件当前指针的位置。这样，一个用户向前移动指针会影响其他共享用户。这里，一个文件具有一个映像，它允许来自不同用户的交替访问。

采用 UNIX 语义，一个文件是与单个物理映射相关联，该映射是作为互斥资源访问的。对这种单个映像的竞争会导致进程延迟。

11.5.5 会话语义

AFS 文件系统(Andrew file system)使用了如下语义一致性：

- 一个用户对打开文件的写不能被同时打开同一文件的其他用户所看见。
- 一旦文件关闭，对其修改只能为以后打开的会话所看见。已经打开文件的用户并不能看见这些修改。

采用这种语义，一个文件同时可与(可能不同的)多个物理映射暂时地相关联。因而，多个用户允许对其自己的映像进行并发(没有延迟)的读和写操作。对于访问的调度几乎没有任何限制。

11.5.6 永久共享文件语义

另一方法是针对**永久共享文件**的。一旦一个文件被其创建者声明为共享，它就不能被修改。永久共享文件有两个重要特性：文件名不能重用，且文件内容不可修改。因此，永久文件的名称表示文件内容已固定，而不用于存储可变信息的文件。在分布系统中实现这种语义是简单的，因为共享是有要求的(只读)。

11.6 保 护

当信息保存在计算机系统中，需要保护其安全，使之不受物理损坏（可靠性）和非法访问（保护）。

可靠性通常是由文件备份来提供的。许多计算机都有系统程序，自动地（或通过计算机操作员干预）定期地（一天或一周或一月一次）把文件复制到磁带上。文件系统可能在以下情况下损坏：硬件问题（如读或写错误），电源过高或故障、磁头损坏、灰尘、温度不适和故意破坏等。文件可能被无意删除。文件系统软件错误也会引起文件内容丢失。可靠性将在第十四章中更加详细地加以讨论。

保护有多种方法。对于小的、单用户系统，可以通过移走软盘和将它们锁在抽屉里或文件柜里提供保护。然而，对于多用户系统，则需要其他的机制。

11.6.1 访问类型

文件保护的需要是允许访问的直接结果。如果系统不允许对其他用户的文件进行访问，也就不需要保护了。因此，可以通过禁止访问以提供完全保护。另外，可通过不加保护以提供自由访问。这两种方法都太极端，不适用于普通使用。人们所需要的是**控制访问**。

通过限制可进行的文件访问类型，保护机制可提供控制访问。是否允许访问的决定因素有若干个，其中之一就是所请求的访问类型。以下几种类型的操作都可以加以控制：

- **读**：从文件中读。
- **写**：对文件进行写或重写。
- **执行**：将文件装入内存并执行它。
- **添加**：将新信息添加到文件结尾部分。
- **删除**：删除文件，使其空间用于其他目的。
- **列表清单**：列出文件名称及其属性。

其他操作，如文件的重命名、复制、编辑，也可控制。然而，这些高层功能可以用系统程序调用低层系统调用来加以实现。保护可以只在低层提供。例如，复制文件可利用一系列读请求来简单地实现。这样，具有读访问的用户就可对文件进行复制、打印等。

目前，提出了许多保护机制。每个机制都有其优缺点，适用于特定的应用。小计算机系统（只为少数几个研究成员使用的）不需要提供大型企业级计算机（用于研究、商务和其他人事活动）一样的访问类型。关于保护问题的完整讨论，将在第十八章进行。

11.6.2 访问控制

解决保护问题的最为常用的方法是根据用户身份进行控制。不同用户可能对同一文件

或目录需要不同类型的访问。实现基于身份访问的最为普通的方法是为每个文件和目录增加一个**访问控制列表**(access-control list, ACL),以给定每个用户名及其所允许的访问类型。当一个用户请求访问一个特定文件时,操作系统检查该文件的访问控制列表。如果该用户属于可访问的,那么就允许访问。否则,会出现保护违约,且用户进程不允许访问文件。

这种方法的优点是可以使用复杂的访问方法。访问控制列表的主要问题是其长度。如果允许每个用户都能读文件,那么必须列出所有具有读访问权限的用户。这种技术有两个不好的结果。

- 创建这样的列表可能比较麻烦且很可能没有用处,尤其是事先不知道系统的用户列表。
- 原来固定大小的目录条目,现在必须是可变大小,这会导致更为复杂的空间管理。

这些问题可以通过使用精简的访问列表来解决。

为精简访问列表,许多系统为每个文件采用了三种用户类型:

- **拥有者**:创建文件的用户是拥有者。
- **组**:一组需要共享文件且具有类似访问的用户形成了组或工作组。
- **其他**:系统内的所有其他用户。

最为常用的现代方法是将访问控制列表与更为常用的用户、组和其他成员访问控制方案(前面述及的)一起组合使用。例如,Solaris 2.6 及后来版本缺省使用三种访问类型,但在需要更为仔细的访问控制时可以允许增加访问控制列表。

作为一个例子,考虑一个用户 Sara 在写一本书。她雇了三个研究生(Jim、Dawn 和 Jill)来帮忙。该书的文本保存在名为 book 的目录中。与该目录相关的保护如下:

- Sara 应该能对其中文件执行所有操作。
- Jim、Dawn 和 Jill 应该只能对其中的文件进行读和写;不能允许他们删除文件。
- 所有其他用户应能对其中的文件读但不能写。(Sara 希望尽可能多的用户能读到该书,以便能收到合适反馈。)

为了实现这种保护,必须创建一个新组,称其为 text,并具有三个成员 Jim、Dawn 和 Jill。组 text 的名称必须与目录 book 相关联,且其访问权限必须按照以上所描述的策略进行设置。

现在,假定有一个访问者,Sara 希望允许其暂时访问第一章。该访问者不能被增加到组 *text* 中,因为这样会授予其访问所有章节。因为文件只能在一个组,所以不能向第一章增加另一个组。采用增加访问控制列表功能,访问者可被增加到第一章的访问控制列表。

为了使该方案正常工作,许可和访问权限必须紧密控制。这种控制可以通过多种方式完成。例如,在 UNIX 系统中,只有管理员或超级用户可以创建和修改组。因此,这种控制是由人机交互来完成的。在 VMS 系统中,文件拥有者可创建和修改其列表。访问列表将在 18.4.2 小节中进一步讨论。

采用更为有限的保护分类只需要三个域就可定义保护。每个域通常为一组位，其中每位允许和拒绝相关访问。例如，UNIX 系统定义了三个域以分别用于文件拥有者、组和其他用户。每个域为三个位：rwx，其中 r 控制读访问，w 控制写访问，而 x 控制执行。因此，对于这里的例子，书的三个域如下：对于拥有者 Sara，所有三个位均已设置；对于组 *text*，r 和 w 位设置；而对于其他用户，只有 r 位设置了。

组合方法的困难之一是用户接口。用户必须能区分一个文件是否有可选的 ACL 许可。在 Solaris 例子中，普通许可之后的"+"表示有可选 ACL 许可。如

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

一组独立命令 setfacl 和 getfacl 用来管理 ACL。另一困难是当许可和 ACL 冲突时谁占先。例如，如果 Joe 在一个文件的组中，该组具有读权限，但该文件有一个 ACL 允许 Joe 读和写，那么 Joe 能写吗？Solaris 允许 ACL 许可占先（因为它们更为细致且缺省并不指派）。这遵守一个通常准则：特殊操作应该占先。

11.6.3 其他保护方法

保护问题的另一个解决方案是为每个文件加上密码。正如对计算机系统的访问通常有密码控制一样，对文件的访问也可用密码控制。如果随机选择密码且经常修改，那么这种方案可有效地用于限制文件为少数知道密码的用户所访问。然而，这种方案有诸多缺点。第一，用户需要记住的密码的数量过大，以致这种方案不可行。第二，如果所有文件只使用一个密码，那么它一旦被发现，所有文件就可被访问。有的系统（如 TOPS-20）允许用户为目录而不是文件关联密码，以便解决这个问题。IBM VM/CMS 操作系统允许一个分区有三个密码，以分别用于读、写和多次访问。第三，通常所有文件只使用一个密码。因此，保护要么有要么没有。为了提供更细致的保护，必须使用多个密码。

现在单用户系统（如 MS-DOS 和 Macintosh 操作系统）也提供有限文件保护。这些操作系统在当初设计时，其实忽略了保护问题。然而，由于这些系统现已联网以共享文件和进行通信，所以必须向这些操作系统增加必要的保护机制。在现有的操作系统上增加功能要比在新操作系统上设计功能难。而且，这种更新通常效果欠佳，且不可能无缝。

对于多层目录结构，不仅需要保护单个文件，而且还需要保护子目录内的文件；即需要提供目录保护机制。必须保护的目录操作不同于文件操作。需要控制在一个目录中创建和删除文件。另外，可能需要控制一个用户能够确定一个目录内是否有一个文件存在。有时，关于文件存在和名称的知识本身就很重要。因此，列出目录内容必须是个保护操作。因而，如果一个路径名表示一个目录内的一个文件，那么用户必须允许访问其目录和该文件。对于支持文件有多个路径名的系统（采用无环图结构目录和图结构目录），根据所使用的路径名的不同，一个用户可能对同一个文件，具有不同的访问权限。

11.6.4 例子:UNIX

在UNIX系统中,目录保护类似于文件保护。即,每个子目录都有三个相关域:拥有者、组成和其他,每个域都有三个位rwx。因此,如果一个子目录的相应域的r位已设置,那么一个用户可列出其内容。类似地,如果一个子目录(如*foo*)的相应域的x位已设置,那么用户可改变其当前目录为该目录(*foo*)。

图11.13显示了在UNIX环境下一个目录的列表。第一个域表示文件或目录的权限。第一个字母为d表示子目录。图11.13还列出了文件链接数、拥有者名称、组名称、文件字节数、上次修改时间和文件名称(具有可选扩展部分)。

rw-rw-r-	1 pbg	staff	31200	Sep 3	08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8	09:33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8	09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3	14:13	student-proj/
rw-r--r--	1 pbg	staff	9423	Feb 24	1999	program.c
rwxr-xr-x	1 pbg	staff	20471	Feb 24	2000	program
drwx--x--x	4 pbg	faculty	512	Jul 31	10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29	06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8	09:35	test/

图11.13 目录列表示例

11.7 小 结

文件是由操作系统所定义和实现的抽象数据类型。它是由一系列逻辑记录组成的。逻辑记录可以是字节、行(固定或可变长度)或更为复杂的数据项。操作系统可自己支持各种记录类型或让应用程序提供支持。

操作系统的主要任务是将逻辑文件概念映射到物理存储设备如磁盘或磁带。由于设备的物理记录大小可能与逻辑记录大小不一样,所以可能有必要将多个逻辑记录合并以存入物理记录。同样,这个任务可以由操作系统自己完成或由应用程序来提供。

文件系统的每个设备都有内容的卷表或设备目录,以列出设备上文件的位置。另外,可创建目录以组织文件。多用户系统的单层结构目录会有命名问题,因为每个文件必须具有惟一文件名。双层结构目录通过为每个用户创建独立目录来解决这个问题。每个用户有其自己的目录,包含自己的文件。目录可通过名称列出其文件,目录包括许多文件信息如文件在磁盘上的位置、长度、拥有者、创建时间、上次使用时间等。

双层结构目录的自然扩展是树型结构目录。树型结构目录允许用户创建子目录以组织

其文件。无环结构目录允许共享子目录和文件，但是使得搜索和删除复杂了。通用图结构目录在文件和目录共享方面提供了更为完全的灵活性，但是有时需要采用垃圾收集以恢复未使用的空间。

磁盘分为一个或多个分区，每个分区可包括一个文件系统。文件系统可安装到系统命名结构中，以使其可用。命名方案因操作系统而异。一旦安装，分区内的文件就可使用。文件系统可以卸载以不允许访问或用于维护。

文件共享依赖于系统所提供的语义。文件可有多个读者、多个作者或有限共享。分布式文件系统允许客户机安装来自服务器的分区或目录，只要能从网络访问即可。远程文件系统在可靠性、性能和安全方面有些挑战。分布式信息系统维护用户、主机、访问信息以便客户机和服务器能共享状态信息以管理使用和访问。

由于文件是绝大多数计算机的主要信息存储机制，所以需要文件保护。文件访问可以针对每种访问类型如读、写、执行、添加、删除、列表等分别加以控制。文件保护可以由口令、访问控制列表和其他特定技术来实现。

习题十一

11.1 假设有一个文件系统，它里面的文件被删除后，当连接到该文件的链接依然存在时，文件的磁盘空间会再度被利用。如果一个新的文件被创建在同一个存储区域或具有同样的绝对路径名，这会产生什么问题？如何才能避免这些问题？

11.2 一些系统当一个用户注销或作业中止时会自动删除所有的用户文件，除非用户显式地要求文件被保留。另外的系统保留所有的文件，除非用户显示式删除它们。论述每种方法的相对优点。

11.3 为什么有些系统跟踪文件的类型，而有些系统把它留给用户来做或干脆不实现多种文件类型？哪种系统更“好”？

11.4 类似地，有些系统支持很多文件数据的结构类型，而其他的一些系统仅简单地支持字节流，各自的优点和缺点是什么？

11.5 在文件的属性中记录下创建程序的名字，其优点和缺点是什么？（在 Macintosh 操作系统中就是这样做的。）

11.6 你能否用一个使用任意长文件名的单层目录结构模拟多层目录结果？如果能，解释你是怎样做到的，并将这种方法与多级目录方法进行比较。如果不能，解释什么问题让你不能这么做。如果文件名限制为不超过 7 个字符，你的答案会是怎样？

11.7 解释 open 和 close 操作的目的是什么？

11.8 有些系统当文件第一次被引用时会自动打开文件，当作业结束时关闭文件。论述这种方案与传统的由用户显式地打开和关闭文件的方案相比有什么优点和缺点？

11.9 给出一个文件中的数据应按下面顺序来访问的应用实例：

a. 顺序

b. 随机

11.10 有些系统提供文件共享时只保留文件的一个拷贝，而另外的一些系统则保留多个拷贝，对共享文件的每个用户提供一个拷贝。论述每种方法的相对优点。

11.11 在有些系统中，一个子目录可以被一个授权的用户读和写，就像一个普通的文件。

a. 描述一下可能产生的文件保护问题。

b. 给你在题 a 中列出的每个保护问题提供一个处理的方法。

11.12 假设一个系统支持 5 000 个用户。再假设你想允许这其中 4 990 个用户能够访问一个文件。

a. 你在 UNIX 中怎样实现这种保护方案。

b. 你能设计一个比 UNIX 提供的更有效地实现这种保护的方案吗？

11.13 研究工作者建议，应该为每个用户建立一个**用户控制列表**（指定用户可以访问哪个文件，如何访问），而不是对每个文件建立一个访问列表（指定哪个用户可以访问这个文件，如何访问）。论述两种方案各自的优点。

推荐读物

Grosshans[1986]给出了关于文件系统的总体论述。Golden 和 Pechura[1986]描述了微机文件系统的结构。Silberschatz 等[2001]中有数据库系统和它们的文件结构的充足的论述。

一个多层目录结构第一次在 MULTICS 系统上实现（Organick[1972]）。现在大多数操作系统都实现了多层目录结构，包括 UNIX（Ritchie 和 Thompson[1974]）、Apple Macintosh 操作系统（Apple[1991]）和 MS-DOS（Microsoft[1991]）。

Norton 和 Wilton[1988]里有 MS-DOS 文件系统的描述。Kenah 等[1988]和 DEC[1981]中有关于 VAX/VMS 的描述。Sun Microsystems 设计的网络文件系统（NFS）允许目录结构分布在联网的计算机系统中。Sandberg 等[1985]、Sandberg[1987]和 Sun[1990]里有关于 NFS 的描述。NFS 在第十六章中有详细的描述。Schroeder 等[1985]描述了不变共享文件语义。

Su[1982]最先提出 DNS，自从那时以后已经修改过多次，Mockapetris[1987]增加了许多重要的特性。最近，Eastlake[1999]建议对 DNS 进行安全扩展以使让它拥有安全钥匙。

LDAP，又被称为 X.509，是 X.500 分布式目录协议的派生子集，是由 Yeong 等[1995]设计的，已经在多个操作系统上实现。在文件系统接口领域，尤其是与文件命名和属性方面相关的研究还在进行。例如，Bell 实验室（Lucent Technology）的 Plan 9 操作系统将所有的对象看做是文件系统。这样，要显示系统中进程的列表，用户可以列出*/proc* 目录的内容。类似地，要显示日期时间，用户只需键入文件*/dev/time*。

第十二章　文件系统实现

正如第十一章所述，文件系统提供了在线存储和访问包括数据和程序在内的文件内容的机制。文件系统永久地驻留在外存上，外存设计成可以永久容纳大量数据。本章主要讨论在最为常用的外存即磁盘，如何存储和访问文件的有关问题。讨论各种方法来组织文件，分配磁盘空间，恢复空闲空间，跟踪数据位置，与其他操作系统部分的接口，等等。本章也将讨论性能问题。

12.1　文件系统结构

磁盘提供大量的外存空间来维持文件系统。磁盘的两个特点，使其成为存储多个文件的方便媒介：

1. 可以原地重写；可以从磁盘上读一块，修改该块，并将它写回到原来的位置。

2. 可以直接访问磁盘上的任意一块信息。因此，可以方便地按顺序或随机地访问文件，从一个文件切换到另一个文件只需要简单地移动读写磁头并等待磁盘转动就可以完成。

本书将在第十四章讨论磁盘结构。

为了改善I/O效率，内存与磁盘之间的I/O转移是以块为单位而不是以字节为单位来进行的。每块为一个或多个扇区。根据磁盘驱动器的不同，扇区从32 B到4 096 B不等；通常为512 B。

为了提供对磁盘的高效且便捷的访问，操作系统通过**文件系统**来轻松地存储、定位、提取数据。文件系统有两个不同的设计问题。第一个问题是如何定义文件系统对用户的接口。这个任务涉及到定义文件及其属性、文件所允许的操作、组织文件的目录结构。第二个问题是创建数据结构和算法来将逻辑文件系统映射到物理外存设备上。

文件系统本身通常由许多不同的层组成。图12.1所示的结构是一个分层设计的例子。设计中的每层利用底层的功能来创建新的功能从而为更高层服务。

*I/O控制*为最底层，由**设备驱动程序**和中断处理程序组成，实现内存与磁盘之间的信息转移。设备驱动程序可以作为翻译器。

图12.1　分层设计的文件系统

其输入由高层命令组成如"提取块 123"。其输出由底层的、硬件特定的命令组成，这些命令用于硬件控制器，通过硬件控制器可以使 I/O 设备与系统其他部分相连。设备驱动程序通常在 I/O 控制器的特定位置写入特定位格式来通知控制器在什么位置采取什么动作。设备驱动程序和 I/O 结构将在第十三章中讨论。

基本文件系统只需要向合适的设备驱动程序发送一般命令就可对磁盘上的物理块进行读写。每个块由其磁盘地址来标识(例如，驱动器 1，柱面(cylinder)73，磁道(track)2，扇区(sector)10)。

文件组织模块(file-organization module)知道文件及其逻辑块和物理块。由于知道所使用的文件分配类型和文件的位置，文件组织模块可以将逻辑块地址转换成基本文件系统所用的物理块地址。每个文件的逻辑块按从 0 或 1 到 N 来编码，而包含数据的物理块并不采用逻辑数，因此需要通过翻译来定位块。文件组织模块也包括空闲空间管理器，用来跟踪未分配的块并根据要求提供给文件组织模块。

最后，**逻辑文件系统**管理元数据。元数据包括文件系统的所有结构数据，而不包括实际数据(或文件内容)。逻辑文件系统根据给定符号文件名来管理目录结构，并提供给文件组织模块所需要的信息。逻辑文件系统通过文件控制块来维护文件结构。**文件控制块**(file control block，FCB)包含文件的信息，如拥有者、许可、文件内容的位置。文件系统也负责保护和安全(参见第十一章和第十八章)。

现在已实现了许多文件系统。绝大多数操作系统都支持多个文件系统。例如，绝大多数 CD-ROM 都是按 *High-Sierra* 格式来写的，这一格式是 CD-ROM 制造商所遵循的标准格式。没有这样一个标准，使用 CD-ROM 的系统之间就很难互操作。除了可移动媒介文件系统外，每个操作系统还有一个或多个基于磁盘的文件系统。UNIX 使用 UNIX **文件系统**(UFS)作为基本文件系统。Windows NT 支持磁盘文件系统 FAT、FAT32 和 NTFS(或 Windows NT File System)，还有 CD-ROM、DVD 和软盘文件系统。通过使用分层结构来实现文件系统，可以最大可能地降低重复代码。I/O 控制(有时候基本文件系统的代码)可以为多个文件系统所共用。每个文件系统都有自己的逻辑文件系统和文件组织模块。

12.2 文件系统实现

正如 11.1.2 小节所述，操作系统实现了 open 和 close 系统调用以便进程可以请求对文件内容的访问。在本节，将深入分析用于实现文件系统操作的结构和操作。

12.2.1 概述

实现文件系统要使用多个磁盘和内存结构。虽然这些结构因操作系统和文件系统而异，但是还是有一些通用规律的。在磁盘上，文件系统可能包括如下信息：如何启动所存储

的操作系统、总的块数、空闲块的数目和位置、目录结构以及各个具体文件等。在本章中，会讨论许多这些结构。

磁盘结构包括：

• **引导控制块**(boot control block)包括系统从该分区引导操作系统所需要的信息。如果磁盘没有操作系统，那么这块的内容为空。它通常为分区的第一块。UFS 称之为**引导块**(boot block)；NTFS 称之为**分区引导扇区**(partition boot sector)。

• **分区控制块**(partition control block)包括分区详细信息，如分区的块数、块的大小、空闲块的数量和指针、空闲 FCB 的数量和指针等。UFS 称之为**超级块**(superblock)；而 NTFS 称之为**主控文件表**(Master File Table)。

• 目录结构用来组织文件。

• FCB 包括很多文件信息，如文件许可、拥有者、大小和数据块的位置。UFS 称之为**索引节点**(inode)。NTFS 将这些信息存在主控文件表中，主控文件采用关系数据库结构，每个文件占一行。

内存信息用于文件系统管理和通过缓存来提高性能。这些结构包括：

• 内存分区表包含所有安装分区的信息。

• 内存目录结构用来保存近来访问过的目录信息。(对安装分区的目录，可以包括一个指向分区表的指针。)

• **系统范围的打开文件表**包括每个打开文件的 FCB 拷贝和其他信息。

• **单个进程的打开文件表**包括一个指向系统范围内已打开文件表中合适条目和其他信息的指针。

为了创建一个新文件，应用程序调用逻辑文件系统。逻辑文件系统知道目录结构形式。为了创建一个新文件，它将分配一个新的 FCB，把相应目录读入内存，用新的文件名更新该目录和 FCB，并将结果写回到磁盘。图 12.2 显示了一个典型的 FCB。

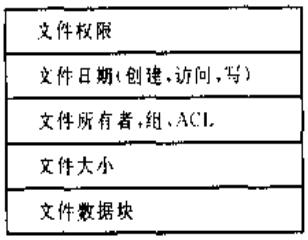


图 12.2 一个典型的文件控制块

有些操作系统如 UNIX 将目录按文件来处理，用一个类型域来表示是否为目录。其他操作系统如 Windows NT 为文件和目录提供了分开的系统调用，对文件和目录采用了不同的处理。不管结构如何，逻辑文件系统都能够调用文件组织模块来将目录 I/O 映射成磁盘块的数目，再进而传递给基本文件系统和 I/O 控制系统。文件组织模块也为文件数据的存储分配了空间。

一旦文件被创建，它就能用于 I/O。不过，首先应打开文件。调用 open 将文件名传给文件系统。当打开文件时，根据给定文件名来搜索目录结构。部分目录结构通常缓存在内存中以加快目录操作。一旦找到文件，其 FCB 就复制到系统范围的打开文件表。该表不但存储 FCB，而且也有打开该文件的进程数量的条目。

接着，在单个进程的打开文件表中会增加一个条目，并通过指针将系统范围的打开文件表的条目同其他域相连。这些其他域可包括文件当前位置的指针（用于读或写操作）和文件打开模式等。调用 open 返回一个指向单个进程的打开文件表中合适条目的指针。所有文件操作都是通过该指针进行的。文件名不必是打开文件表的一部分，因为一旦完成对 FCB 在磁盘上的定位，系统就不再使用文件名了。对于访问打开文件表的索引有各种名称。UNIX 称之为**文件描述符**（file descriptor）；Windows 2000 称之为**文件句柄**（file handle）。因此，只要文件没有被关闭，所有文件操作都是通过打开文件表来进行的。

当一个进程关闭文件，就删除一个相应的单个进程打开文件表的条目，系统范围内打开文件表的打开数也会递减。当打开文件的所有用户都关闭一个文件时，更新的文件信息会复制到磁盘的目录结构中，系统范围的打开文件表的条目也将删除。

在实际中，系统调用 open 会首先搜索系统范围的打开文件表以确定某文件是否已被其他进程所使用。如果是，就在单个进程的打开文件表中创建一项，并指向现有系统范围的打开文件表的相应条目。该算法在文件已打开时，能节省大量开销。

有的系统更加复杂，它们将文件系统作为对其他系统方面的访问接口，如网络。例如，UFS 的系统范围的打开文件表有关于文件和目录的索引节点（inode）和其他信息。它也有关于网络连接和设备的类似信息。

这些结构的缓存也不应忽视。采用这种方案，关于打开文件的所有信息（除了数据外）都可以在内存中。BSD UNIX 系统在使用缓存方面比较典型，哪里能节省磁盘 I/O 哪里就使用缓存。其 85% 的平均缓存命中率说明了这些技术的实现是值得的。BSD UNIX 系统在附录 A 中详细描述。

图 12.3 总结了文件系统实现的操作结构。

12.2.2 分区与安装

磁盘布局因操作系统而异。一个磁盘可以分成多个分区，或者多个磁盘可以组成一个分区。这里，讨论前一种情况，而后一种情况可以作为 RAID 的一种形式，将在 14.5 小节中讨论。

分区可以是“生的”（raw），即没有文件系统，或者“熟的”（cooked）即含有文件系统。**“生”磁盘**（raw disk）用于没有合适文件系统的地方。UNIX 交换空间可以使用生分区，因为它不使用文件系统而是使用其自己的磁盘格式。同样，有的数据库使用生磁盘，格式化它来满足其特定需求。生磁盘也可用于存储 RAID 磁盘系统所需要的信息，如用以表示哪些块已经镜像和哪些块已改变且需要镜像的位图。类似地，生磁盘可包括一个微型数据库，以存储 RAID 配置信息，如哪些磁盘属于 RAID 集合。生磁盘将在 14.3.1 小节中进一步讨论。

引导信息能存在各个分区中。同样，它有自己的格式，因为在引导时系统并没有文件系统设备驱动程序，所以并不能解释文件系统格式。因此，引导信息通常为一组有序块，并作

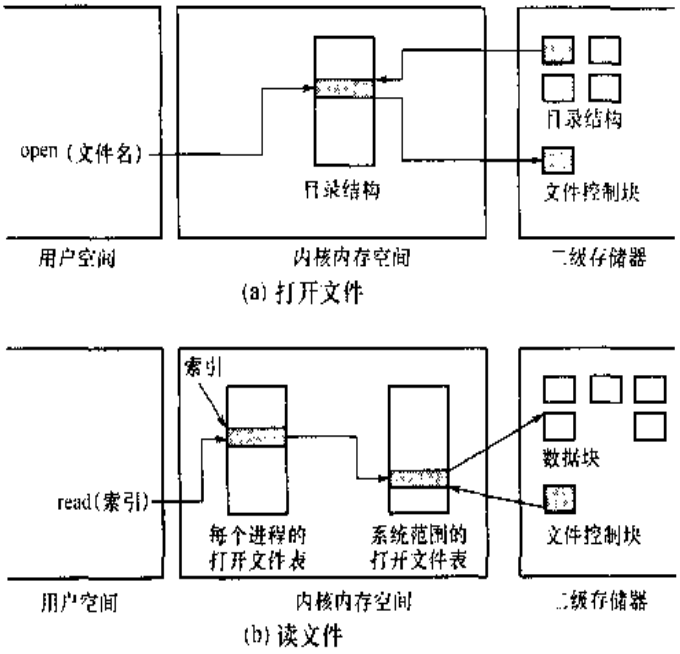


图 12.3 内存中的文件系统结构

为二进制文件读入内存。该二进制文件按预先指定的位置如第一个字节开始执行。引导信息除了包括如何启动一个特定操作系统外，还可以有其他指令。例如，个人计算机和其他系统可以**双引导**。可以把多个操作系统装在这样的系统上。系统如何知道引导哪个？一个能够理解多个文件系统和多个操作系统的引导装入程序可以位于引导区。一旦装入，它可以引导位于磁盘上的一个操作系统。磁盘可以有多个分区，每个分区包含不同类型的文件系统和不同的操作系统。

根分区(root partition)包括操作系统内核或其他系统文件，在引导时装入内存。其他分区根据不同操作系统可以在引导时自动装入或在此之后手动装入。作为成功装入操作的一部分，操作系统会验证设备上的文件系统确实有效。操作系统通过设备驱动程序读入设备目录并验证目录是否有合适的格式。如果为有效格式，那么检验分区一致性，并根据需要自动和手动地加以纠正。最后，操作系统在其位于内存的**装入表**中注明该文件系统已装入和该文件系统的类型。装入功能的细节因操作系统而异。微软公司 Windows 系统将分区装入在独立名称空间中，名称用字母和冒号表示。例如，操作系统为了记录一个文件系统已装在 f_i 上，会在对应 f_i 的设备结构的一个域中加上一个指向该文件系统的指针。当一个进程给定设备字母时，操作系统会查找到合适文件系统的指针，并遍历设备上的目录结构以查找给定的文件和目录。

UNIX 可以将文件系统装在任何目录上。这可以在位于内存的目录的索引节点(inode)上加上一个标记。该标记表示此目录是安装点。一个域指向安装表上的一个条目，以表示哪个设备安装在哪里。该安装表条目包括一个指向位于设备上的文件系统的超级块。这种

方案使得操作系统可以遍历其目录结构，并根据需要切换文件系统。

12.2.3 虚拟文件系统

上一节清楚地说明了现代操作系统必须同时支持多个文件系统类型。现在讨论实现细节。操作系统如何才能把多个文件系统整合成一个目录结构？用户如何在访问文件系统空间时，可以无缝地在文件系统类型之间移动？

实现多个类型文件系统的一个明显但不十分满意的方法是为每个类型编写目录和文件程序。但是，绝大多数操作系统包括 UNIX 都使用面向对象技术来简化、组织和模块化实现过程。使用这些方法允许不同文件系统类型可通过同样结构来实现，这也包括网络文件系统类型如 NFS。用户可以访问位于本地磁盘的多个文件系统类型，甚至位于网络上的文件系统。

采用数据结构和子程序，可以分开基本系统调用的功能和实现细节。因此，文件系统实现包括三个主要层次，如图 12.4 所示。第一层为文件系统接口，包括 open，read，write 和 close 调用及文件描述符。

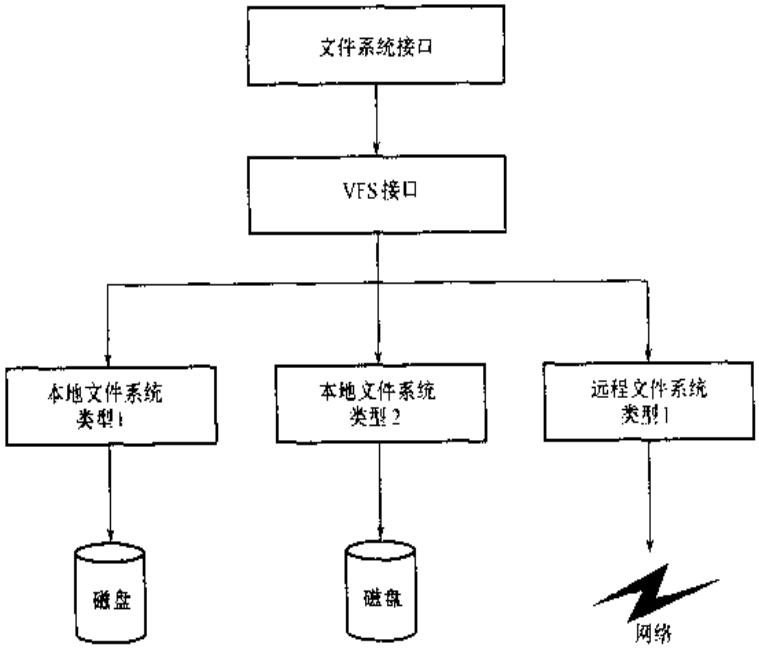


图 12.4 虚拟文件系统示意图

第二层称为**虚拟文件系统**(VFS)层；它有两个目的：

1. VFS 层通过定义一个清晰的 VFS 接口，以将文件系统通用操作和具体实现分开。多个 VFS 接口的实现可以共存在同一台机器上，它允许访问已装在本地的多个类型的文件系统。

2. VFS 是基于称为 **vnode** 的文件表示结构，该结构包括一个数值指定者以表示位于整个网络范围内的惟一文件。该网络范围的惟一性需要用来支持网络文件系统。内核中为每

个活动节点(文件或目录)保存一个 vnode 结构。

因此,VFS 区分本地文件和远程文件,根据文件系统类型可以进一步区分不同本地文件。

VFS 根据文件系统类型调用特定文件类型操作以处理本地请求,通过调用 NFS 协议子程序来处理远程请求。文件句柄可以从相应的 vnode 中构造,并作为参数传递给子程序。结构中最底层是实现文件系统类型或远程文件系统协议的层。VFS 操作的具体说明参见 12.9 节。

12.3 目录实现

目录分配和目录管理算法的选择对文件系统的效率、性能和可靠性有很大影响。因此,你需要理解有关这些算法的优缺点。

12.3.1 线性列表

最为简单的目录实现方法是使用存储文件名和数据块指针的线性列表。目录条目的线性列表需要采用线性搜索来查找特定条目。这种方法编程简单但运行费时。要创建新文件,必须首先搜索目录以确定没有同样名称的文件存在。接着,在目录后增加一个新条目。要删除文件,根据给定文件名搜索目录,接着释放分配给它的空间。要重用目录条目,可以有许多办法。可以将目录条目标记为不再使用(赋予它一个特定的文件,如全为空的名称或为每个条目增加一个使用—非使用位),或者可以将它加到空闲目录条目列表上。第三种方法是将目录的最后一个条目拷贝到空闲位置上,并降低目录长度。链表可以用来减少删除文件的时间。

目录条目的线性列表的真正缺点是查找文件需要线性搜索。目录信息需要经常使用,用户在访问文件时会注意到实现的快慢。事实上,许多操作系统采用软件缓存来存储最近访问过的目录信息。缓存命中避免了不断地从磁盘读取信息。排序列表可以使得二分搜索,并减少平均搜索时间。不过,列表始终需要排序的要求会使得文件的创建和删除复杂化,这是因为可能需要不少的目录信息来保持目录的排序。一个更为复杂的树数据结构,如 B—树,可能更为有用。已排序列表的一个优点是不需要排序步骤就可生成排序目录信息。

12.3.2 哈希表

用于文件目录的另一个数据结构是**哈希表**。采用这种方法时,除了使用线性列表存储目录条目外,还使用了哈希数据结构。哈希表根据文件名得到一个值,并返回一个指向线性列表中元素的指针。因此,它大大地降低目录搜索时间。插入和删除也较简单,不过需要一些预备措施来避免**冲突**(collision)(两个文件名哈希到相同的位置)。哈希表的最大困难是其通常固定的大小和哈希函数对大小的依赖性。

例如,假设使用线性哈希表来存储 64 个条目。哈希函数可以将文件名转换为 0 到 63

的整数，例如采用除以 64 的余数。如果后来设法创建第 65 个文件，那么必须将目录哈希表扩大到 128 个条目。因此，需要一个新的哈希函数来将文件名映射到 0 到 127 的范围，而且必须重新组织现有目录条目以反映它们新的哈希函数值。或者，可以使用 chained-overflow 哈希表。每个哈希条目可以是链表而不是单个值，可以采用向链表增加一项来解决冲突。由于查找一个名称可能需要搜索冲突条目组成的链表，因而查找可能变慢；但是，这比线性搜索整个目录可能还是要快很多。

12.4 分配方法

磁盘的直接访问特点使人们能够灵活地应用文件。在绝大多数情况下，一个磁盘可存储许多文件。主要问题是如何为这些文件分配空间，以便有效地使用磁盘空间和快速地访问文件。常用的磁盘空间分配方法有三个：连续、链接和索引。每种方法都有其优点和缺点。有的系统（如 Data General 公司的用于 Nova 系列计算机的 RDOS 操作系统）对三种方法都支持。但是，更为常见的是一个系统只提供对一种方法的支持。

12.4.1 连续分配

连续分配（contiguous-allocation）方法要求每个文件在磁盘上占有一组连续的块。磁盘地址为磁盘定义了一个线性排序。采用这种排序，假设有一个作业访问磁盘，在访问块 b 后访问块 $b+1$ 通常不需要移动磁头。当需要磁头移动（从一个磁道的最后扇区到下一个磁道的第一扇区），只需要移动一个磁道。因此，用于访问连续分配文件所需要的寻道数最小，在确实需要寻道时所需要的寻道时间也最少。使用连续分配方法的 IBM VM/CMS 操作系统提供很好的性能。

文件的连续分配可以用第一块的磁盘地址和连续块的数量来定义。如果文件有 n 块长并从位置 b 开始，那么该文件将占有块 b，$b+1$，$b+2$，…，$b+n-1$。一个文件的目录条目包括开始块的地址和该文件所分配区域的长度，参见图 12.5。

对一个连续分配文件的访问很容易。要顺序访问，文件系统会记住上次访问过块的磁盘地址，如需要可读入下一块。要直接访问一个从块 b 开始的文件的块 i，可以直接访问块 $b+i$。因此连续分配支持顺序访问和直接访问。

不过，连续分配也有一些问题。一个困难是为新文件找到空间。在 12.5 节中所要描述的空闲空

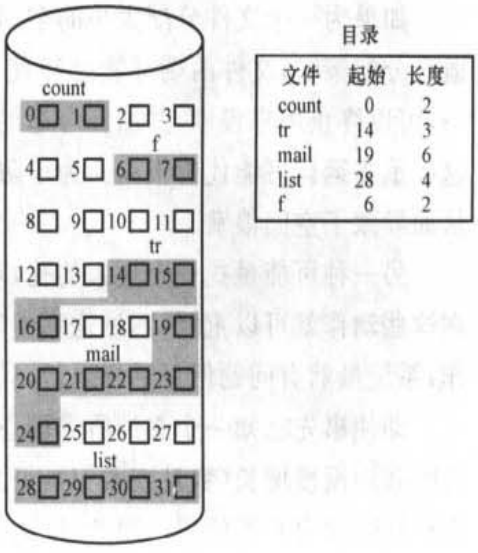


图 12.5 磁盘空间的连续分配

间管理系统的实现决定了如何实现这一任务。虽然可以使用任何管理系统，但是有的系统会比其他的慢。

连续磁盘空间分配问题可以作为在 9.3 节中所述的通用**动态存储分配**(dynamic storage-allocation)问题的一个具体应用，即如何从一个空闲孔列表中寻找一个满足大小为 n 的空间。从一组空闲孔中寻找一个空闲孔的最为常用的策略是首次适合和最优适合。模拟结果显示在时间和空间使用方面，首次适合和最优适合都要比最坏适合更为高效。首次适合和最优适合在空间使用方面不相上下，但是首次适合运行得要快。

这些算法都有**外部碎片**(external fragmentation)问题。随着文件的分配和删除，磁盘空闲空间被分成许多小片。只要空闲空间分成小片，就会存在外部碎片。当最大连续片不能满足需求时就有问题；存储空间分成了许多小孔，但没有一个足够大以存储数据。因磁盘空间的总数和文件平均大小的不同，外部碎片可能是一个小问题也可能是个大问题。

某些老式微型计算机系统对软盘采用了连续分配。为了防止由于外部碎片而导致的大量磁盘空间的浪费，用户必须运行一个重新打包程序，以将整个文件系统复制到另一个软盘或磁盘上。原来的软盘完全变成空的，从而创建了一个大的连续空闲空间。接着，该重新打包程序又对这一大的连续空闲空间采用连续分配方法，以将文件复制回来。这种方案有效地将所有小的空闲空间**合并**(compact)起来，因而解决了碎片问题。这种合并的代价是时间。这种时间代价对于使用连续分配的大硬盘是很严重的，可能需要几小时，因此可能只能一周运行一次。在这一**停机期间**(down time)，不能进行正常操作，因此在生产系统中合并应尽可能地避免。

连续分配的另一个问题是确定一个文件需要多少空间。当创建文件时，需要找到和分配文件所需要的总的空间。创建者(程序和人)又如何知道所创建文件的大小？有时，这个确定比较简单(例如，复制一个现有文件)；然而通常来说，输出文件的大小是比较难估计的。

如果为一个文件分配太小的空间，那么可能会发现文件不能扩展。尤其是采用了最优适合分配策略，文件两端可能已经使用。因此，不能在原地扩大文件。有两种可能性。第一，可以终止用户程序，并加上合适的错误消息。用户必须分配更多空间并再次运行程序。这些重复运行可能代价很高。为了防止这些问题，用户通常会过多地估计所需的磁盘空间，从而导致了空间浪费。

另一种可能是找一个更大的孔，复制文件内容到新空间，释放以前的空间。只要空间存在这些动作就可以重复，不过这耗费时间。当然，在这种情况下，用户无需知道这些具体动作；系统虽然有问题但可继续运行，只不过会越来越慢。

即使事先已知一个文件所需的总的空间，预先分配的效率仍可能很低。一个文件在很长时间内慢慢增长(数月或数年)，仍需要为它最后的大小分配足够空间，虽然其中的大部分在很长时间内并不使用。因此，该文件有大量的内部碎片。

为了降低这些缺点，有的操作系统使用修正的连续分配方案，该方案开始分配一块连续

空间，当空间不够时，另一块被称为**扩展的**连续空间会添加到原来的分配中。这样，文件块的位置就成为开始地址、块数、加上一个指向下一扩展的指针。在有的系统上，文件用户可以设置扩展大小，但如果用户设置不对会影响效率。如果扩展太大，内部碎片可能是个问题；随着不同大小的扩展的分配和删除，外部碎片可能也是个问题。商用 VFS 使用扩展来优化性能。它是标准 UFS 的高性能替代品。

12.4.2 链接分配

链接分配(linked allocation)解决了连续分配的所有问题。采用链接分配，每个文件是磁盘块的链表；磁盘块分布在磁盘的任何地方。目录包括文件第一块的指针和最后一块的指针。例如，一个有 5 块的文件可能从块 9 开始，然后是块 16、块 1、块 10，最后是块 25(图 12.6)。每块都有一个指向下一块的指针。用户不能使用这些指针。因此，如果每块有 512 B，磁盘地址为 4 B，那么用户可以使用 508 B。

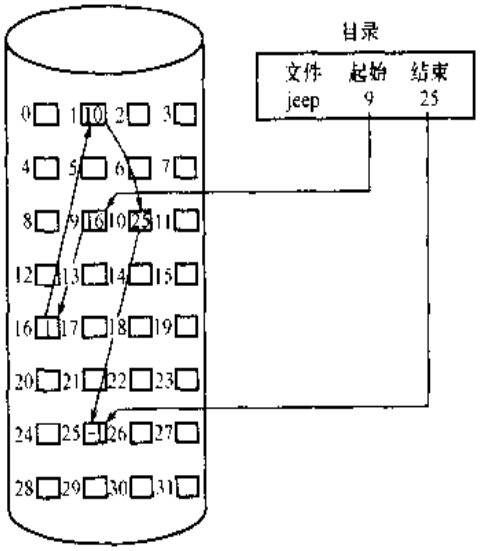


图 12.6 磁盘空间的链接分配

要创建新文件，简单地在目录中增加一个新条目。对于链接分配，每个目录条目都有一个指向文件首块的指针。该指针初始化为 *nil*(链表结束指针值)以表示空文件。大小字段也为 0。要写文件就会通过空闲空间管理系统找到一个空闲块，将该块链接到文件的尾部，以便写入。要读文件，通过块到块的指针，简单地读块。采用链接分配没有外部碎片，空闲空间列表上的任何块可以用来满足请求。当创建文件时，并不需要说明文件大小。只要有空闲块，文件就可以增大。因此，无需合并磁盘空间。

不过，链接分配也有缺点。主要问题是它只能有效地用于文件的顺序访问。要找到文件的第 i 块，必须从文件的开始起，跟着指针，找到第 i 块。对指针的每次访问都需要读磁盘，有时需要磁盘寻道。因此，链接分配不能有效地支持文件的直接访问。

链接分配的另一缺点是指针需要空间。如果指针需要使用 512 字节块中的 4 个字节，那么 0.78% 的磁盘空间将会用于指针，而不是其他信息。因而，每个文件也需要比原来更多的空间。

对这个问题的常用解决方法是将多个块组成**簇**(cluster)，并按簇而不是按块来分配。例如，文件系统可能定义一个簇为 4 块，并以簇为单位来操作。这样，指针所使用的磁盘空间的百分比就更少。这种方法允许逻辑到物理块的映射仍然简单，但是提高磁盘输出(更少磁头移动)，且降低了块分配和空闲列表管理所需要的空间。这种方法的代价是增加了内部

碎片，如果一个簇而不是块没有充分使用，那么就会浪费更多空间。簇可以改善许多算法的磁盘访问时间，因此应用于绝大多数操作系统中。

链接分配的另一个问题是可靠性。由于文件是通过指针链接的，而指针分布在整个磁盘上，想一下如果指针失去或损坏会是什么结果。操作系统软件的 bug 或磁盘硬件的故障可能会导致获得一个错误指针。这种错误可能导致牵连到空闲空间列表，或另一个文件。一个不彻底的解决方案是使用双向链表或在每个块中存上文件名和相对块数；不过，这些方案为每个文件增加了额外开销。

一个采用链接分配方法的变种是**文件分配表**(FAT)的使用。这一简单但有效的磁盘空间分配用于 MS-DOS 和 OS/2 操作系统。每个分区的开始部分用于存储该 FAT 表。每块都在该表中有一项，该表可以通过块号码来索引。FAT 的使用与链表相似。目录条目含有文件首块的块号码。根据块号码索引的 FAT 条目包含文件下一块的块号码。这种链会一直继续到最后一块，该块对应 FAT 条目的值为文件结束值。未使用的块用 0 值来表示。为文件分配一个新的块只要简单地找到第一个值为 0 的 FAT 条目，用新块的地址替换前面文件结束值，用文件结束值替代 0。一个由块 217 618 和 339 组成的文件的 FAT 结构如图 12.7 所示。

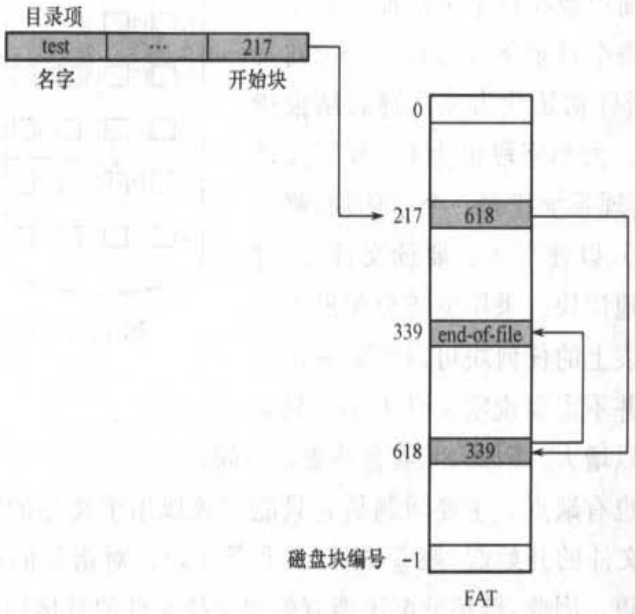


图 12.7 文件分配表

如果不对 FAT 采用缓存，FAT 分配方案可能导致大量的磁头寻道时间。磁头必须移到分区的开始位置以便读入 FAT，寻找所需要块的位置，接着移到块本身的位置。在最坏的情况下，每块都需要两次移动。优点是改善了随机访问时间，因为通过读入 FAT 信息，磁头能找到任何块的位置。

12.4.3 索引分配

链接分配解决了连续分配的外部碎片和大小声明问题。但是，如果不用 FAT，那么链接分配就不能有效支持直接访问，这是因为块指针与块一起分布整个磁盘，且必须按顺序读取。**索引分配**(indexed allocation)解决了这个问题，它把所有指针放在一起：**索引块**。

每个文件都有其空闲的索引块，这是一个磁盘块地址的数组。索引块的第 i 个条目指向文件的第 i 个块。目录条目包括索引块的地址(图 12.8)。要读第 i 块，通过索引块的第 i 个条目的指针来查找和读入所需的块。这一方法类似于第九章所描述的分页方案。

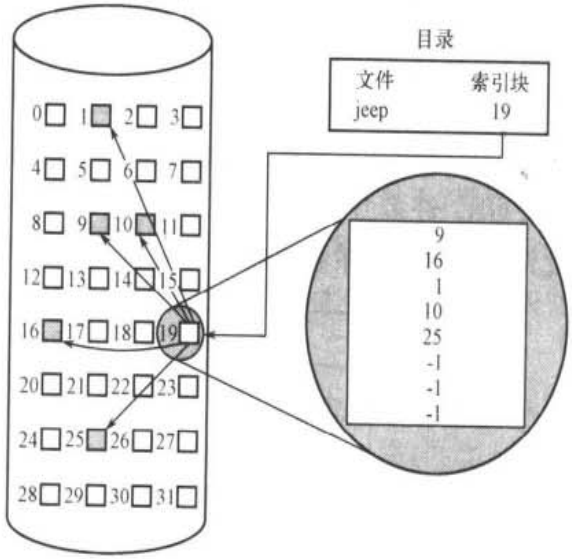


图 12.8 磁盘空间的索引分配

当要创建文件时，索引块的所有指针都设为 *nil*。当首次写入第 i 块时，先从空闲空间管理器中得到一块，再将其地址写到索引块的第 i 个条目。

索引分配支持直接访问，且没有外部碎片问题，这是因为磁盘上的任一块都可满足更多空间的要求。

索引分配会浪费空间。索引块指针的开销通常要比链接分配的指针开销要大。设想一下一般情况，每个文件只有一个块或两块长。采用链接分配，每块只浪费一个指针(总的为一个或两个指针)。采用索引分配，不管只有一个或两个指针为非空，都必须分配一个完整的索引块。

这也提出了一个问题：索引块应为多大？每个文件必须有一个索引块，因此需要索引块尽可能地小。不过，如果索引块太小，那么它不能为大文件存储足够多的指针。因此，必须采取一定机制来处理这个问题：

- **链接方案**：一个索引块通常为一个磁盘块。因此，它本身能直接读写。为了处理大文件，可以将多个索引块链接起来。例如，一个索引块可以包括一个含有文件名的头部和一组前 100 磁盘块的地址。下一个地址(索引块的最后一个词)为 nil(对于小文件)或指向另一个索引块(大文件)。

- **多层索引**：链接表示的一个变种是用第一层索引块指向一组第二层的索引块，第二层索引块再指向文件块。为了访问一块，操作系统通过第一层索引查找第二层索引，再用第二层索引查找所需的数据块。这种方法根据最大文件大小的要求，可以继续到第三层或第四层。对于有 4 096 B 的块，能在索引块中存入 1 024 个 4 B 的指针。两层索引允许

1 048 576个数据块，这允许最大文件为 4 GB。

• **组合方案**：另一个方案用于 UFS，将索引块的前 15 个指针存在文件的 inode 中。这其中的前 12 个指针指向**直接块**；即，它们包括了能存储文件数据的块的地址。因此，(不超过 12 块的)小文件不需要其他的索引块。如果块大小为 4 KB，那么不超过 48 KB 的数据可以直接访问。其他 3 个指针指向**间接块**。第一个间接块指针为**一级间接块**的地址。一级间接块为索引块，它包含的不是数据，而是那些包含数据的块的地址。接着是一个**二级间接块**指针，它包含了一个块的地址，而这个块中的地址指向了一些块，这些块中又包含了指向真实数据块的指针。最后一个指针为**三级间接块**的地址。采用这种方法，一个文件的块数可以超过许多操作系统所使用的 4 B 的文件指针所能访问的空间。32 bit 指针只能访问2^{32} B，或 4 GB。许多 UNIX 如 Solaris 和 IBM AIX 现在支持高达 64 bit 的文件指针。这样的指针允许文件和文件系统为数千吉字节。图 12.9 显示了一个 inode。

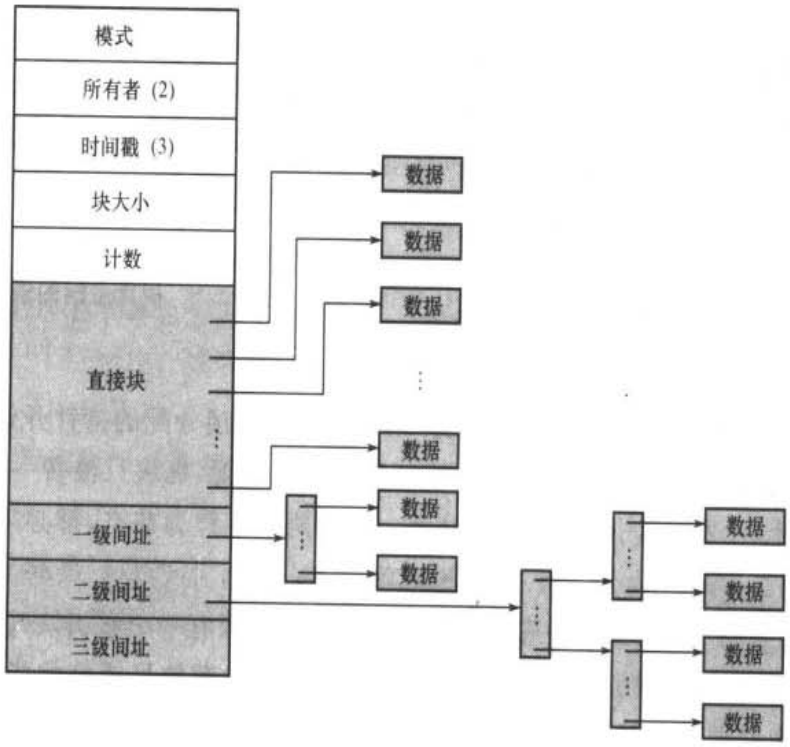


图 12.9 UNIX 的 inode

索引分配方案与链接分配一样在性能方面有欠缺。尤其是，虽然索引块可以缓存在内存中，但是数据块必须分布在整个分区上。

12.4.4 性能

本书所讨论的分配方法在存储效率和数据块访问时间上各有特点。这两个特性是为实现操作系统而选择合适算法时的重要依据。

在选择分配方法之前，需要确定系统将如何被使用。一个主要为顺序访问的系统应该

与一个主要为随机访问的系统采用不同的方法。不管什么类型的访问，连续分配需要访问一次就能得到磁盘块。由于能将文件的开始地址放在内存中，可以马上计算出第 i 块的磁盘地址（或下一块），并能直接读。

对于链接分配，也能将下一块的地址放在内存中，并可以直接读取。对于顺序访问，这种方法还可以；但对于直接访问，对第 i 块的访问可能需要读 i 次磁盘。这一问题也说明了为什么链接分配不适用于需要直接访问的应用程序。

因此，有的系统通过使用连续分配以支持文件的直接访问，通过链接分配以支持文件的顺序访问。对于这些系统，所使用的访问类型必须在文件创建时加以说明。用于顺序访问的文件可以链接分配，但不能用于直接访问。用于直接访问的文件可以连续分配，且能支持直接访问和顺序访问，但是在创建时，必须说明其最大文件大小。在这种情况下，操作系统必须有合适的数据结构和算法来支持这两种分配方式。文件可以从一种类型转换成另一种类型：创建一个所需的类型，将原来文件的内容复制过来，删除原来文件，重新命名新文件。

索引分配更为复杂。如果索引块已在内存中，那么可以直接访问。不过，将索引块保存在内存中需要相当大的空间。如果内存空间不够，那么可能必须先读入索引块，再读入所需的数据块。对于两级索引，可能需要读两次索引块。对于一个非常巨大的文件，访问文件结束附近的块需要读入所有索引块以跟踪指针链，最后才能读入所需要的数据块。因此，索引分配的性能依赖于索引结构、文件大小以及所需块的位置。

有的系统将连续分配和索引分配组合起来：对小文件（只有 3 或 4 块）采用连续分配；当文件大时，自动切换到索引分配。由于绝大多数文件都较小，小文件的连续分配的效率又高，所以平均性能还是很好的。

例如，Sun 公司的 UNIX 操作系统版本在 1991 年修改过，以改善文件系统分配算法的性能。性能测试表示在一个典型工作站（12 MIPS SPARC 工作站）最大磁盘吞吐量使用了 CPU 的 50%，产生了 1.5 MB/s 的磁盘带宽。为了改善性能，Sun 做了改进，只要可能就按大小为 56 KB 的簇来分配空间。（56 KB 是当时 Sun 工作站 DMA 传输的最大能力。）这种分配减少了外部碎片、因页寻道和延迟时间。另外，也优化了读磁盘程序以方便读这些大簇。索引节点（inode）没有改变。这些改进，加上使用了向前读和后释放，降低了 25% 的 CPU 使用，大大地提高了磁盘输出量。

可能有许多其他优化在使用。由于 CPU 和磁盘速度的不等，就是花费操作系统数千条指令以节省一些磁头移动都是值得的。再者，随着时间的推移，这种不等程度会增加，甚至花费操作系统数十万条指令来优化磁头移动，也是值得的。

12.5 空闲空间管理

因为磁盘空间有限，所以如果可能，需要将删除文件的空间用于新文件。（只写一次的光

盘仅允许向任何扇区写一次，因而不可能重新使用。）为了记录空闲磁盘空间，系统需要维护一个**空闲空间链表**（free-space list）。空闲空间链表记录了所有空闲磁盘空间，即未分配给文件或目录的空间。当创建文件时，搜索空闲空间链表以得到所需要的空间，并分配给新文件。这些空间会从空闲空间链表中删除。当删除文件时，其磁盘空间会增加到空闲空间表上。空闲空间链表虽然称为链表，但不一定实现为链表，这一点随着后面的讨论将会清楚。

12.5.1 位向量

通常，空闲空间表实现为**位图**（bit map）或**位向量**（bit vector）。每块用一位表示。如果一块为空闲，那么其位为 1；如果一块已分配，那么其位为 0。

例如，假设有一个磁盘，其块 2、3、4、5、8、9、10、11、12、13、17、18、25、26 和 27 为空闲，其他块为已分配。那么，空闲空间位图如下：

001111001111110001100000011100000…

这种方法的主要优点是查找磁盘上第一空闲块和 n 个连续空闲块时相对简单和高效。确实，许多计算机都有位操作指令，能有效地用于这一目的。例如，从 80386 开始的 Intel 系列和从 68020 开始的 Motorola 系列都有能返回一个字中第一个值为 1 的位的偏移的指令。事实上，Apple Macintosh 操作系统就使用位向量方法来分配磁盘空间。当要查找第一空闲块，Macintosh 操作系统会按顺序检查位图的每个字以检查其是否为 0，因为一个值为 0 的字表示其对应的所有块都已分配。再对第一个值为非 0 的字进行搜索值为 1 的位偏移，该偏移对应着第一个空闲块。该块号码的计算如下：

（一个字的位数）×（值为 0 的字数）+第一个值为 1 的位的偏移

这里，再次看到硬件特性简化了软件功能。不过，如果整个位向量都要保存在内存中（并时而写入到磁盘用于恢复的需要），那么位向量的效率就不高。对于微机上的小磁盘，完全保存在内存中是有可能的，但对于大的计算机就不行了。对于一个每块为 512 B、容量为 1.3 GB 的磁盘，可能需要 332 KB 来存储位向量，以便跟踪空闲空间。如果采用按合并 4 个扇区为一个簇，那么该数字会变为每个磁盘需要 83 KB 的内存。

12.5.2 链表

空闲空间管理的另一种方法是将所有空闲磁盘块用链表连接起来，并将指向第一空闲块的指针保存在磁盘的特殊位置，同时也缓存在内存中。第一块包含一个下一空闲磁盘块的指针，如此继续下去。对上一个例子（12.5.1 小节），有一个指向块 2（第一个空闲块）的指针。块 2 包含一个指向块 3 的指针，块 3 指向块 4，块 4 指向块 5，块 5 指向块 8，等等（图 12.10）。不过，这种方案的效率不高；要遍历整个表时，需要读入每一块，这需要大量的 I/O 时间。好在遍历整个表并不是一个经常操作。通常，操作系统只不过简单地需要一个空闲块以分配给一个文件，所以分配空闲表的第一块就可以了。FAT 方法将空闲块的计算结合

到分配数据结构中，不再需要另外的方法。

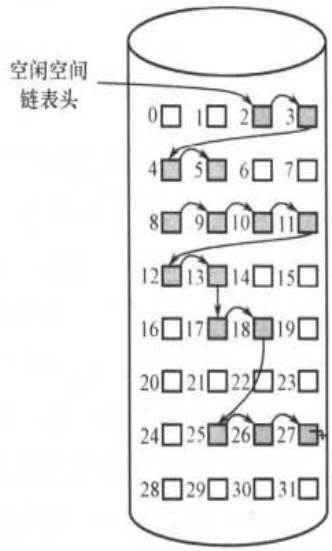


图 12.10 采用链接方式的磁盘空闲空间链表

12.5.3 组

对空闲链表的一个改进是将 n 个空闲块的地址存在第一个空闲块中。这些块中的前 $n-1$ 个确实为空。而最后一块包含另外 n 个空闲块的地址，如此继续。这种实现的重要性在于大量空闲块的地址可以很块地被找到，这一点有别于标准链表方法。

12.5.4 计数

另外一种方法是利用了这样一个事实：通常，有多个连续块需要同时分配或释放，尤其是在使用连续分配和采用簇时更是如此。因此，不是记录 n 个空闲块的地址，可以记录第一块的地址和紧跟第一块的连续的空闲块的数量 n。这样，空闲空间表的每个条目包括磁盘地址和数量。虽然每个条目会比原来需要更多空间，但是表的总长度会更短，这是因为连续块的数量常常大于 1。

12.6 效率与性能

既然已经讨论了块分配和目录管理方法，那么就可进一步考虑它们对性能和磁盘使用效率的影响。由于磁盘是计算机主要部件中最慢的部分，所以磁盘常成为系统性能的瓶颈。在本节，将讨论各种技术，以改善外存的效率和性能。

12.6.1 效率

磁盘空间的有效使用主要取决于所使用的磁盘分配和目录管理算法。例如，UNIX 索

引节点(inode)预先分配在分区上。即使一个"空"磁盘也会有一定百分比的空间用于存储索引节点。而由于预先分配索引节点并将其分布在整个分区上,从而改善了文件系统的性能。这种性能改善是由于UNIX所采用的分配和空闲空间算法所带来的,这些算法试图将文件数据与其索引节点信息存放在一起,从而降低了寻道时间。

作为另一个例子,再研究一下12.4节所讨论的簇技术,这有利于文件查找和文件传输但以内部碎片为代价。为了降低这种碎片,BSD UNIX根据文件大小而调节簇大小。当大簇能填满时,就用大簇;对小文件和文件最后簇,就使用小簇。这种系统将在附录A中描述。

保留在文件目录条目(或索引节点)内的数据类型也需要加以考虑。通常,要记录"最近写日期"以提供给用户,如用于确定是否需要对给定文件备份。有的系统也记录"最近访问日期",以便用户能确定最后一次读文件在什么时候。由于保留了这个信息,每当读文件时,目录结构的一个域就必须被更新。这种更新要求将相应块读入内存,修改相应部分,再将该块写回至磁盘,这是因为磁盘操作是以块(或簇)为单位来进行的。因此,每次文件打开以读取时,其目录条目也必须读入和写出。对于经常访问的文件,这种要求是低效的。因此,在设计文件系统时,必须平衡其好处和性能代价。通常,与文件相关的所有数据项都必须加以研究,以考虑其对效率和性能的影响。

作为一个例子,考虑用于访问数据的指针大小是如何影响效率的。绝大多数系统在其整个操作系统中,使用16位或32位指针。这些指针大小将文件大小限制为2^{16}(64 KB)或2^{32}(4 GB)。有的系统实现了64位指针以将限制提高到2^{64} B,这确实是个巨大的数字。然而,64位指针需要更多空间以便存储,相应地,分配和空闲空间管理方法(链表、索引等)也使用更多磁盘空间。

选择指针大小(或操作系统内的其他任何固定分配大小)的困难之一是需要考虑技术变化的影响。考虑一下早期的IBM PC XT只有10 MB硬盘,其MS-DOS文件系统只支持32 MB。(每个FAT条目为12 bit,指向大小为8 KB的簇)随着磁盘容量不断增加,更大磁盘必须分成32 MB的分区,这是因为该文件系统只能跟踪32 MB以内的磁盘块。随着超过100 MB容量硬盘的普及,MS-DOS的磁盘数据结构和算法必须加以修改以支持更大文件系统。(每个FAT条目先扩展到16位,后来又到32位)。最初的文件系统是基于效率设计的。然而,随着MS-DOS V4的出现,数百万计算机用户必须很不方便地切换到新的、更大的文件系统。

作为另一个例子,来看一下Sun公司Solaris操作系统的发展。最初,许多数据结构都是固定大小的,在系统启动时分配的。这些结构包括进程表和打开文件表。当进程表已满时,就不能再创建进程。当文件表已满时,就不能再打开文件。从而,系统不能向用户提供服务。如果要增加这些表格的大小,就只能重新编译内核并重新启动。从Solaris 2发布以来,几乎所有内核结构都是动态分配的,取消了对系统性能的这些人为限制。当然,管理这些表格的算法也更加复杂,而且操作系统也有点慢(由于必须动态地分配和释放这些表条

目),但是为了更为通用的功能,这种代价也是值得的,也是经常需要付出的。

12.6.2 性能

即使选择了基本文件系统,仍然能够从多方面来改善性能。如第二章所述,绝大多数磁盘控制器都有本地内存以作为板载高速缓存,它足够大,能同时存储整个磁道。寻道之后,整个磁道就可以从磁头所处的扇区开始,读入到磁盘缓存(以缓解延迟时间)。接着,磁盘控制器可将所请求的扇区传给操作系统。在数据块从磁盘控制器调入到内存之后,操作系统就可缓存它。

有的系统有一块独立内存用做**磁盘缓存**,位于其中的块假设马上需要使用。其他系统采用**页缓存**(page cache)来缓存文件数据。页缓存使用虚拟内存技术,以将文件数据作为页而不是面向文件系统的块来缓存。采用虚拟地址来缓存文件数据,与采用物理磁盘块来缓存相比,更为高效。许多系统,包括 Solaris、新版 Linux、Windows NT/2000,都使用页缓存来缓存进程页和文件数据。这称为**统一虚拟内存**(unified virtual memory)。Solaris 使用块缓存和页缓存。块缓存用于保存文件系统元数据(如索引节点)而页缓存用于所有文件系统数据。

有的 UNIX 版本提供了**统一缓冲缓存**(unified buffer cache)。考虑一下文件打开和访问的两种方法。一种方法是使用内存映射(10.3.2 小节),另一种方法是使用标准系统调用 read 和 write。如果没有统一缓存,那么情况会如图 12.11 所示。在这种情况下,标准系统调用 read 和 write 会经过缓冲缓存。内存映射调用需要使用两个缓存:页缓存和缓冲缓存。内存映射先从文件系统中读入磁盘块并存放在缓冲缓存中。因为虚拟内存不能直接与缓冲缓存进行交流,所以缓冲缓存内的文件必须复制到页缓存中。这种情况称为**双缓存**(double caching),需要两次缓存文件数据。这不但浪费内存,而且也浪费重要 CPU 和 I/O 时间(需要用于在系统存储之间进行数据移动)。而且,这两种缓存之间的不一致性也会导致破坏文件。通过提供统一缓冲缓存,内存映射和 read 和 write 系统调用都使用同样的页缓存。这避免了双重缓存且也允许用虚拟内存系统来管理文件数据。这种统一缓冲缓存如图

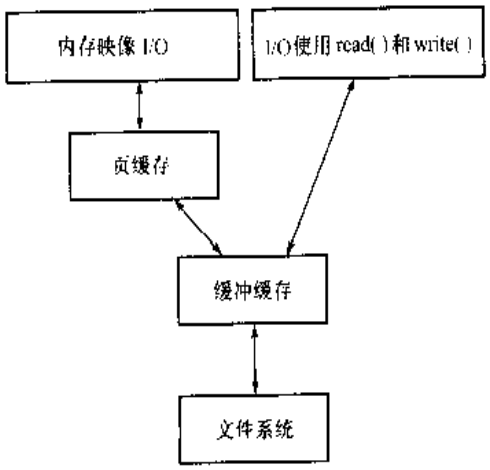


图 12.11 无统一缓冲缓存的 I/O

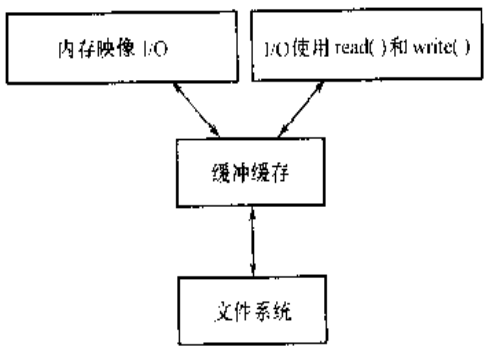


图 12.12 采用了统一缓冲缓存的 I/O

12.12 所示。

不管是缓存磁盘块还是页，LRU 似乎都是一个用于块或页替换的、合理且通用的算法。然而，Solaris 页缓存算法演变说明了选择算法的困难。Solaris 允许进程和页缓存共享未使用内存。在 Solaris 2.5.1 之前，为进程或页缓存而分配页是没有区别的。因此，执行许多 I/O 操作的系统会为页缓存而使用绝大多数可用内存。由于很高频率的 I/O，当空闲内存变少时，页扫描程序（10.7.2 小节）会从进程而不是从页缓存中收回页。Solaris 2.6 和 Solaris 7 可选择地实现了优先调页，即页扫描程序赋予进程页更高的级别（而不是页缓存）。Solaris 8 在进程页和文件系统页缓存之间增加了固定限制，以阻止一方将另一方赶出内存。

页缓存、文件系统和磁盘驱动程序有着有趣的联系。当数据写到磁盘文件时，页先放在缓存中，并且磁盘驱动程序会根据磁盘地址对输出队列进行排序。这两个操作允许磁盘驱动程序最小化磁头寻道和优化写数据。除非要求同步写，否则进程写磁盘只是写入缓存，系统在方便时异步地将数据写到磁盘中。因而，用户觉得写非常快。当从磁盘中读入数据时，块 I/O 系统会执行一定的提前读；结果是，写比读更加接近于异步。因此，通过文件系统输出到磁盘，通常要比从磁盘读入更加快；这与直觉相反。

同步写按磁盘子系统接收顺序来进行；写并不缓存。因此，调用子程序必须等待数据写到磁盘驱动器后，再继续。绝大多数时间进行的是**异步写**。对于异步写，将数据先存在缓存后就返回。有些操作如元数据写可以是同步写。操作系统经常允许系统调用 open 包括一个标记，以表示允许进程请求执行同步写。例如，数据库的原子事务使用这种操作，以确保数据按给定顺序存入稳定存储中。

有的系统根据文件访问类型，采用不同替换算法，以优化页缓存。按顺序所进行的文件的读入或写出就不应采用 LRU 页替换，因为最近使用的页最后才使用或根本不用。因此，顺序访问可以通过采用称为马上释放和预先读取的技术，来加以优化。**马上释放**（free-behind）是在一旦请求下一页时，就马上从缓存中删除上一页。以前的页不可能再次使用，且浪费缓冲空间。采用**预先读取**（read-ahead），所请求的页和之后的一些页可一起读入并缓存。这些可能在本页处理之后就要被请求。从磁盘中一次性地读入这些数据并加以缓存节省了大量时间。对于多道程序系统，控制器上的磁盘缓存并不能代替这种需要，这是因为从磁道缓存到内存的许多小传输的延迟和开销。

使用内存改善性能的另一方法通常用于个人计算机。留出一块内存作为**虚拟磁盘**（或 RAM **磁盘**）。在这种情况下，RAM 磁盘设备驱动程序接受所有标准磁盘操作，但是在内存中而不是在磁盘上执行这些操作。所有磁盘操作都可在 RAM 盘上执行，除了速度飞快外，用户并不会感到其他差别。但是，RAM 磁盘只能用于临时存储，因为掉电或重启通常会删除所有数据。通常，可以存放临时文件如中间编译结果。

RAM 磁盘和磁盘缓存的区别是 RAM 盘上的内容是由用户所完全控制的，而磁盘缓存是由操作系统所控制的。例如，RAM 盘可能一直为空，直到用户（或程序）在其上创建文件

时为止。图 12.13 显示了系统内的可能存在的缓存位置。

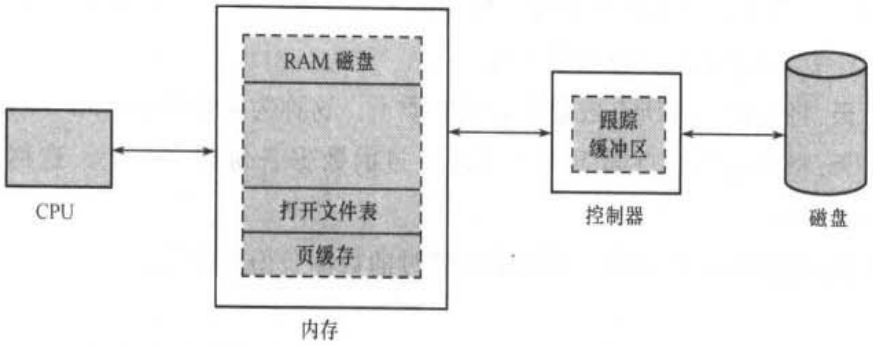


图 12.13　不同的磁盘缓存位置

12.7 恢　复

由于文件和目录可保存在内存和磁盘上，所以必须注意确保系统失败不会引起数据丢失和数据的不一致性。

12.7.1 一致性检查

正如 12.3 节所述，部分目录信息保存在内存(或缓存)中以加快访问。因为缓存目录信息写到磁盘并不是马上进行的，所以内存目录信息通常要比相应磁盘信息更新。

考虑一下计算机崩溃所产生的可能影响。在这种情况下，打开文件表通常丢失了，同样与打开文件相关的目录修改也丢失了。这种事件会导致文件系统处于不一致状态：有的文件真实状态与目录结构所记录的不一样。通常，一个检查和纠正磁盘不一致的特殊程序需要在启动时运行。

一致性检查程序(consistency checker)将目录结构数据与磁盘数据块相比较，并试图纠正所发现的不一致。分配算法和空闲空间管理算法决定了检查程序能发现什么类型的问题，及其如何成功地纠正问题。例如，如果使用链接分配，那么每一块都指向下一块，这样整个文件可以从数据块来重建，并可重建目录结构。对于采用索引分配系统，目录条目的损坏可能就严重了，因为数据块之间并没有什么联系。因此，UNIX 在读时对目录条目进行缓存，而在执行导致空间分配或其他元数据变更的操作时同步地更新目录条目。

12.7.2 备份与恢复

由于磁盘有时会出错，所以必须注意确保数据不会永远丢失。为此，可以利用系统程序将磁盘数据**备份**到另一存储设备如软盘、磁带或光盘。恢复单个文件或整个磁盘时，只需要从备份中加以**恢复**就可以了。

为了降低复制量，可利用每个文件的目录条目信息。例如，如果备份程序知道一个文件上次何时备份，且目录内该文件上次写日期表明该文件从此备份以来并未改变，那么该文件不必再复制。一个典型备份计划可以这样。

- **第 1 天**：将磁盘上的所有数据备份到介质上。这称为**完全备份**(full backup)。
- **第 2 天**：将磁盘上的自第 1 天以来改变过的数据备份到介质上。这称为**增量备份**(incremental backup)。
- **第 3 天**：将磁盘上的自第 2 天以来改变过的数据备份到介质上。

⋮

- **第 N 天**：将磁盘上的自第 $N-1$ 天以来改变过的数据备份到介质上。再返回到第 1 天。

新一轮可以将备份写在以前或新的备份介质集合上。这样，可从完全备份上开始恢复，并根据增量备份不断更新。当然，N 越大，完整恢复所读入的磁带或磁盘的数量就越多。这种备份周期的一个额外优点是：可以恢复任何一个在一个周期内所偶然删除的文件，只要通过从前一天备份中恢复删除文件就可以了。周期长度是由所需备份介质数量和恢复多少天的数据所决定的。

用户可能在文件损坏好久以后，才发现数据不见或损坏。因此，通常需要不时地进行完全备份以永远保存，而不是重新使用这些介质。将这些永久备份与普通备份分开保存是个好的方法，这可避免危险(如失火)会损坏计算机和所有备份。如果备份周期重新使用介质，那么必须注意不要过多次地使用介质：介质可能会磨损以致不能从备份中恢复数据。

12.8 基于日志结构的文件系统

在计算机科学里，通常许多算法和技术会从其原来的应用而扩展到其他应用领域。7.9.2 小节所描述的数据库基于日志恢复算法就属于这种情况。这些日志算法已成功地应用到一致性检查问题。这种实现称为**面向事务的基于日志**(log-based transaction-oriented)的文件系统。

回想一下磁盘文件系统数据结构如目录结构、空闲指针、空闲 FCB 指针，可能因系统崩溃而不一致。在操作系统采用基于日志的技术之前，通常要适当地修改这些结构。一个典型操作如文件创建，可能涉及修改文件系统内的许多结构。修改目录结构，分配 FCB，分配数据块，减少这些块的空闲计数。这些修改可能因系统崩溃而中断，从而产生了数据的不一致。例如，空闲计数可能表示 FCB 已分配，但是目录结构还没有指向该 FCB。这样该 FCB 可能就丢失了，除非使用一致性检查程序。

允许数据结构损坏再通过恢复来修补会有许多问题。一个问题是一致性可能无法修补。一致性检查可能不能恢复结构，从而导致文件甚至整个目录失去。一致性检查可能需

要人工干预以解决冲突，如果没有人在场那么这是不方便的。系统可能直到操作员告诉系统如何做前一直都不能使用。一致性检查也浪费系统时间。数千占字节的数据检查可能需要数小时。

这个问题的解决可采用基于日志的恢复技术以更新文件系统元数据。NTFS 和 VFS 都使用这种方法，Solaris 7 及其后版本也为 UFS 增加了这种选项。事实上，这已为许多操作系统所采用。

简单地说，所有元数据都按顺序写到日志上。执行一个特殊任务的一组操作称为**事务**(transaction)。这些修改一旦写到这个日志上之后，就可认为已经提交，系统调用就可返回到用户进程，以允许它继续执行。同时，这些日志条目再对真实文件系统结构进行重放。随着修改的进行，可不断地更新一个指针以表示哪些操作已完成和哪些仍然没有完成。当一个完整提交事务已完成，那么就可从日志文件中删除(日志文件事实上是个环形缓冲)。日志可能是文件系统的一个独立部分，或在另一个磁盘上。采用分开读/写磁头可以减少磁头竞争和寻道时间，因此会更有效但也更复杂。

如果系统崩溃，日志文件可能有零个或多个事务。这些事务虽然已经由操作系统所提交，但是还没有(对文件系统)完成，所以必须要完成。根据指针可以执行事务直到该工作完成，文件系统结构保持一致。惟一可能出现的问题是一个事务被中断。即，在系统崩溃之前，它还没有被提交。这些事务所做的文件系统修改必须撤消，以恢复文件系统的一致性。这种恢复只是在崩溃时才需要，从而避免了与一致性检查有关的所有问题。

对磁盘元数据更新采用日志的另一好处是，这些更新比直接在磁盘上进行要快。这种改善原因是顺序 I/O 比随机 I/O 的性能要好。低效率的同步随机元数据写被转换成较高效的同步顺序写(到基于日志文件系统的记录区域)。这些修改再通过随机写而异步回放到适当数据结构。总的结果是面向元数据操作(如文件创建和文件删除)性能的提高。

12.9 NFS

网络文件系统已经普及。它们通常与客户系统的总体目录结构和接口相集成。NFS 是个很好的、广泛使用的、实现很好的网络文件系统的例子。这里，用它作为例子来讨论网络文件系统的实现细节。

NFS 是用于通过 LAN(或甚至 WAN)访问远程文件的软件系统的实现和规范。NFS 是 ONC+的一部分，ONC+为绝大多数 UNIX 厂商和一些个人计算机操作系统所支持。这里所描述的实现是关于 Solaris 操作系统的，这是基于 UNIX SVR4 的改进版，可运行在 SUN 工作站和其他硬件上。它采用了 TCP 或 UDP/IP 协议(根据互连网络而定)。在这里有关 NFS 的描述中，规范和实现交织在一起。在讨论细节时，参考 Sun 实现；在讨论一般原理时，就针对规范。

12.9.1 概述

NFS将一组互连工作站作为具有独立文件系统的机器组合。目的是允许透明地(根据显式请求)共享这些文件系统。共享可在两个对等机器之间进行,而不需要专门服务器。为了确保机器独立,远程文件系统的共享只影响客户机而不是其他机器。

为了使一台特定机器 *M*1 透明地访问远程目录,客户机必须先执行安装操作。通过安装操作可将远程目录安装到本地文件系统的目录上。一旦完成安装,远程目录就与本地文件系统有机集成起来,从而取代了原来本地目录下的内容。本地目录就成为新安装目录的根的名称。远程目录作为安装参数按非透明方式提供;远程目录位置(或主机)必须提供。然而,在此之后,*M*1 的用户就可完全透明地访问远程目录。

为了描述文件系统安装,考虑一下图 12.14 所示的文件系统,其中三角形表示所感兴趣的目录子树。图中有三个机器的 *U*、*S*1 和 *S*2 的三个独立文件系统。这时,每台机器只可访问本地文件系统。图 12.15 说明了将 *S*1:/*usr*/*shared* 安装到 *U*:/*usr*/*local* 的情况,该图说明了机器 *U* 的用户所看到的文件系统。注意这些用户在安装完成后可采用/*usr*/*local*/*dir*1 来访问目录 *dir*1 内的任何文件。原来机器上的/*usr*/*local* 的内容不再可见。

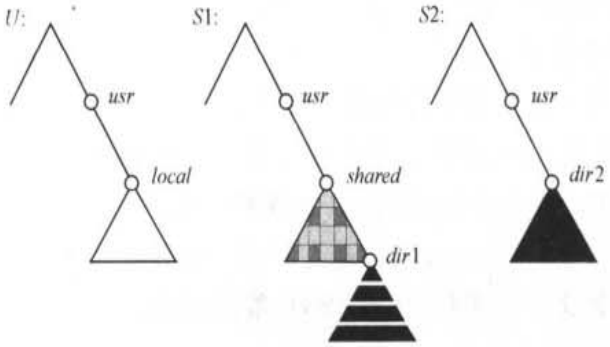
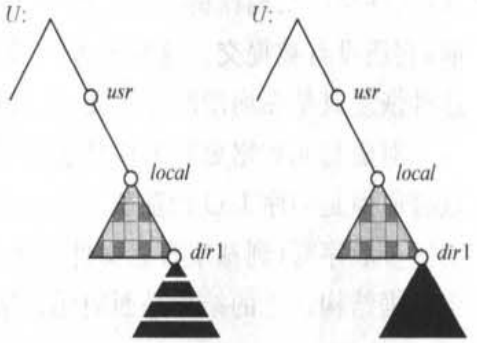


图 12.14 三个独立的文件系统



(a) 一般安装 (b) 级联安装

图 12.15 NFS 中的安装

根据访问权限,任何文件系统或文件系统内的任何目录,均可远程安装在本地目录上。无盘工作站甚至可从服务器安装其根目录。

有的 NFS 实现也允许通过串联安装。即一个文件系统可安装在另一个远程安装的文件系统上。一个机器只受其自己安装所影响。

通过安装远程文件系统,客户不能获得对碰巧安装在以前文件系统上的其他文件系统的访问。因此,安装机器不具有转移性。在图 12.15(b)中,通过继续前面的例子说明了串联安装。该图说明了安装 *S*2:/*usr*/*dir*2 到 *U*:/*usr*/*local*/*dir*1 的结果。*U* 的用户可使用前缀/*usr*/*local*/*dir*1 访问 *dir*2 上的内容。如果共享文件系统安装在网络上所有计算机的用户主目录上,那么用户可登录到任何工作站且获得其主环境。这种属性允许**用户移动性**(user mobility)。

NFS 设计目标之一是允许在不同机器、操作系统和网络结构的异构环境中工作。NFS 规范与这些无关，因此允许其他实现。这种独立性是因为在两种独立实现接口之间采用 RPC 和 XDR 而带来的。因此，系统由异构机器和文件系统(满足 NFS 接口要求)所组成，不同类型文件系统可以本地或远程安装。

NFS 规范区分两种服务：一是由安装机制所提供的服务，二是真正远程文件访问服务。相应地，有两种协议用于这两种服务：一是安装协议，另一是远程文件访问协议即 ***NFS*** 协议。协议是用 RPC 来表示的。这些 RPC 是用于实现透明远程文件访问的基础。

12.9.2 安装协议

安装协议(mount protocol)在客户机和服务器之间建立初始逻辑连接。在 Sun 的实现中，每台机器都有一个服务进程，独立于内核，执行协议功能。

安装操作包括要安装的远程目录的名称和存储它的服务器的名称。安装请求映射到相应 RPC，并传送到在特定服务器上运行的安装服务程序。服务器维护一个**输出列表**(export list)，在 Solaris 上为*/etc/dfs/dfstab*，该文件只能由超级用户编辑，它列出了哪些本地文件系统已经输出以便安装，及允许安装它们的机器名称。该文件也可能包括访问权限如只读。为了简化维护输出列表和安装表，可采用分布命名方案来存储这些信息，以供客户所使用。

输出文件系统的任何目录可以为授权机器所远程安装。因此，组成单元是目录。当服务器收到满足其输出链表的安装请求时，它就返回客户一个**文件句柄**(file handle)，为进一步访问安装文件系统内的文件所用。该文件句柄包括服务器需要区分其所存储的单个文件的所有信息。对于 UNIX，它由文件系统标识符和索引节点所组成，以标识安装文件系统内的确切安装目录。

服务器也维护一个客户机及相应于当前安装目录的列表。该列表主要用于管理目的，如通知所有客户服务器将要关机。通过安装协议影响服务器状态的惟一方法是增加和删除列表内的条目。

通常，一个系统有一个静态安装预配置，这是在启动时建立的(在 Solaris 中，为*/etc/vfstab*)，然而，这种安排可修改。除了真正安装步骤外，安装协议还包括几个其他程序如卸载和返回输出列表。

12.9.3 NFS 协议

NFS 协议提供了一组 RPC 以供远程文件操作。这些程序包括下列操作：

- 搜索目录内的文件
- 读一组目录条目
- 操作链接和目录
- 访问文件属性

· 读和写文件

只有在远程目录的句柄已经建立了之后，才可以进行这些操作。

操作 open 和 close 的省略是故意的。NFS 服务器的一个显著特点是无状态。服务器并不维护客户每一步的访问信息。服务器端没有像 UNIX 中的打开文件表和文件结构。因此，每个请求必须提供完整参数，包括惟一文件标识符和用于特定操作的文件内的绝对偏移。这种设计较为坚固，不需要采用特别措施以从崩溃状态中恢复过来。为此，文件操作必须是幂等的。每个 NFS 请求都有一个序列号，允许服务器确定一个请求是否重复和缺少。

上面所提到的客户列表的维护似乎违反了服务器的无状态属性。然而，这种列表对客户机和服务器的正确运行并不重要，因此它不需要从服务器崩溃中加以恢复。因而，它可能包括不一致的数据，这只作为一个提示。

无状态服务器和 RPC 同步的结果是：修改数据（包括间接和状态块）必须在返回结果给客户之前，提交到服务器磁盘上。即，客户机可缓存写数据块，但是当其将数据发送到服务器时，它假定这些数据已到达服务器磁盘。服务器必须同步写所有 NFS 数据。因此，服务器崩溃和恢复对客户来说是不可见的；服务器为客户所管理的所有数据块要完好。由于没有缓存，因而性能损失是巨大的。可能采用其自己的非易失性缓存（由电池备份的内存）以提高性能。磁盘控制器在写已存储在非易失性缓存上之后就确认磁盘写。这样主机看到极快的同步写。这些块即使在系统崩溃后仍然完好，并定期地写到磁盘上。

单个 NFS 写程序可确保是原子的，不会与同一文件的其他写调用相混杂。然而，NFS 协议并不提供并发控制机制。一个系统调用 write 可能分成多个 RPC 写，因为每个 NFS 写或读只能包括 8 KB 的数据而 UDP 包限制为 1 500 个字节。因此，两个用户对同一远程文件的写可能会导致数据相混杂。由于锁管理本身是有状态的，所以 NFS 外的服务必须提供加锁（Solaris 就这么做）。建议用户在 NFS 之外采用机制以协调对共享文件的访问。

NFS 通过 VFS 与操作系统有机地集成。为了演示结构，下面来跟踪对一个已打开远程文件的操作是如何进行的（参见图 12.16）。客户通过普通系统调用来启动操作。操作系统层将这个调用映射成相应 V 节点的 VFS 操作。VFS 层发现这个文件是远程的，因此调用适当 NFS 程序。一个 RPC 调用发送到远程的 NFS 服务层。该调用又插入到远程系统的 VFS 层，远程系统发现这是本地的并调用适当文件系统操作。计算结果再按这个步骤返回到客户机。这种结构的特点是客户机和服务器是相等的；因此一个机器可以是客户机或服务器，或两个都是。

服务器的实际服务是由多个内核进程执行的，以提供轻量级进程（或线程）机制的临时代替。

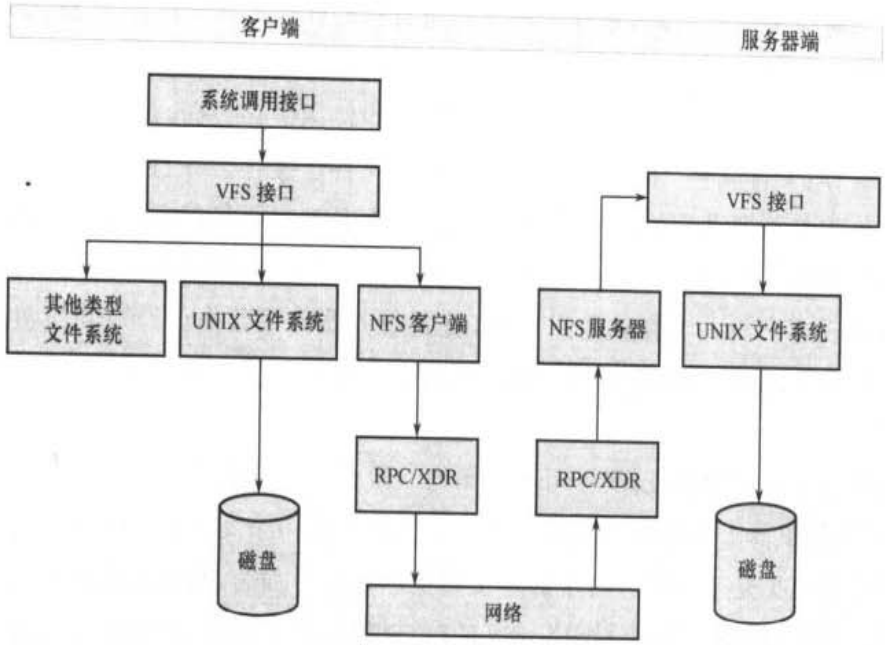


图 12.16 NFS体系结构示意图

12.9.4 路径名转换

路径名转换(path-name translation)将路径分成组件名称,并为每个组件名称和目录虚拟节点执行独立 NFS lookup 查找。一旦碰到安装点,每个组成查找就对服务器发送一个独立 RPC。这个低效率的路径转换方案是需要的,因为每个客户都有其惟一的逻辑名称空间,这是由其所执行的安装决定的。在碰到安装点时,如果发送给服务器一个路径名并接收一个目标虚拟节点,那么也许会更为高效。然而,服务器可能并不知道某个特定客户有另一个安装点。

所以为了加快查找,客户端的路径转换缓存可保存远程目录名称的虚拟节点。这种缓存加快了具有同样初始路径名称文件的引用。当从服务器所返回的属性与缓存内的属性不匹配时,目录缓存就要加以更新。

记住有的 NFS 实现允许在一个已经远程安装的文件系统上再安装另一个远程文件系统(串联安装)。然而,服务器并不作为一个客户机和另一个服务器的中介。因此,客户机必须通过直接安装所需要的目录,与第二个客户机建立客户机－服务器连接。当客户机有串联安装时,路径转换可能涉及到多个服务。然而,每个组件查找都是在原客户机和某个服务器之间进行的。因此,当客户机查找一个目录且服务器在其上安装了文件系统时,客户机看到下面的目录,而不是安装目录。

12.9.5 远程操作

除了打开和关闭文件,在普通 UNIX 文件操作系统调用和 NFS 协议 RPC 之间,几乎有

一个一对一的对应关系。因此，远程文件操作可直接转换成相应 RPC。从概念上来说，NFS 坚持远程服务形式，但是实际上采用了缓冲和缓存技术以提高性能。事实上，文件块和文件属性是由 RPC 获取并在本地缓存的。将来远程操作使用缓存数据，并受一致性的限制。

有两种缓存：文件属性（索引节点信息）缓存和文件块缓存。在打开文件时，内核检查远程服务器以确定是否要获取信息或使缓存信息重新生效。只有在缓存属性为最新时，才使用相应文件数据块。缓存属性缺省时在 60 s 后丢弃。在服务器和客户机之前，采用提前读和延迟写技术。客户机并不释放延迟写块，直到服务器确认数据已经写到磁盘上。与使用 Sprite 的系统相反，延迟写是保留的，不管文件是否按冲突模式而并发打开。因此，没有保留 UNIX 语义。

系统的性能要求使得很难保持 NFS 语义的一致性。一个机器上一些新的文件已经创建了 30 秒钟而其他机器上可能仍不知道。在一个机器上写一个文件对于其他打开并在读这个文件的机器是否可见是未知的。对于那个文件的新的打开而言，只能看到已经提交到服务器的改变内容。因此，NFS 不提供 UNIX 语义的严格模仿，也不提供 Andrew 会话语义。尽管有这些缺点，其实用和高效仍使它成为当前使用最广泛的多数厂家支持的分布式文件系统。

12.10 小 结

文件系统持久驻留在外存上，外存设计成可以持久地容纳大量数据。最常用外存介质是磁盘。

物理磁盘可分成区，以控制介质的使用和允许在同一磁盘上支持多个可能不同的文件系统。这些文件系统安装在逻辑文件系统结构上，然后才可以使用。文件系统通常按层结构或模块结构来加以实现。低层处理存储设备的物理属性。高层处理符号文件名和文件逻辑属性。中间层将逻辑文件概念映射到物理设备属性。

每个文件系统类型都有其结构和算法。VFS 层允许上层统一地处理每个文件系统类型。即使远程文件系统也能集成到文件系统目录结构中，通过 VFS 接口采用标准系统调用进行操作。

文件在磁盘上有三种不同空间分配方法：连续的、链接的或索引分配。连续分配有外部碎片问题。链接分配的直接访问效率低。索引分配可能因其索引块而浪费一定空间。连续分配可采用区域来扩展，以增加灵活性和降低外部碎片。索引分配可按簇（为多个块）来进行，以增加吞吐量和降低所需索引条目的数量。按大簇的索引分配与采用区域的连续分配相似。

空闲空间分配方法也影响磁盘使用效率、文件系统性能、外存可靠性。所使用的方法包括位向量和链表。优化方法包括组合、计数和 FAT（将链表放在一个连续区域内）。

目录管理程序必须考虑效率、性能和可靠性。哈希表是最为常用的方法；它快速且高

效。然而，表损坏和系统崩溃可能导致目录信息与磁盘内容不一致。一致性检查程序，如UNIX的系统程序fsck，或MS-DOS的chkdsk，可用来修补损坏。操作系统备份工具允许磁盘数据复制到磁带，恢复数据甚至整个磁盘（因硬件失败、操作系统错误或用户错误）。

网络文件系统（如NFS）使用客户机－服务器方法允许用户访问远程机器的文件和目录，就好像本地文件系统一样。客户机上的系统调用转换成网络协议，再转换成服务器的文件系统操作。网络和多客户访问在数据一致性和性能方面增加了挑战。

由于文件系统在系统操作中的重要位置，其性能和可靠性十分关键。日志结构和缓存等技术可帮助改善性能，日志结构和RAID可提高可靠性。

习题十二

12.1 假设一个文件有100个块，并且FCB（及索引块，当是索引分配时）已经在内存中。对于下列条件，计算读一个块对于连续、链接和索引（一级）分配方法各需要多少次磁盘I/O操作。在连续分配方法中，假设增长时起始端没有空间、末端有空间。假设要增加的块信息存放在内存中。

a. 在起始端增加块

b. 在中部增加块

c. 在末端增加块

d. 从起始端删除块

e. 从中部删除块

f. 从末端删除块

12.2 假设有一个系统，它的空闲空间保存在空闲空间链表中。

a. 假设指向空闲链表的指针丢失了，系统能不能重建空闲空间链表？为什么？

b. 设计一个方案以确保发生内存错误时总不会丢失链表指针。

12.3 如果一个系统允许一个文件系统同时安装到超过一个位置时会产生什么问题？

12.4 为什么文件分配的位图必须保存在大容量存储器中，而不是主存中？

12.5 假设一个系统支持连续分配、链接分配和索引分配，应用什么标准决定哪个方法最适用于一个特定文件？

12.6 设想一个在磁盘上的文件系统的逻辑块和物理块的大小都为512 B。假设每个文件的信息已经在内存中。对三种分配方法（连续分配、链接分配和索引分配），分别回答下面的问题：

a. 逻辑地址到物理地址的映射在系统中是怎样进行的？（对于索引分配，假设文件总是小于512块长。）

b. 假设现在处在逻辑块10（最后访问的块是块10），现在想访问逻辑块4，那么必须从磁盘上读多少个物理块？

12.7 连续分配的一个问题就是用户必须给每个文件预分配足够的空间。如果一个文件增长超过了分配给它的空间，就必须采取特殊的措施。一个解决方法是定义一个有一个初始连续区域（有一个指定大小）的文件结构。如果这个区域满了，操作系统自动将溢出的部分链接到初始的连续区域。如果溢出区满了，就分配另一个溢出区。将这种实现方法与标准连续分配和链接分配进行比较。

12.8 一个存储设备上的存储碎片可以通过信息再压缩来消除。典型的磁盘设备没有重定位或基址寄存器(像内存被压缩时用的一样),怎样才能重定位文件呢?给出为什么文件再压缩和重定位常常被避免使用的三个原因。

12.9 高速缓存是怎样提高性能的?如果它们那么有用为什么系统不用更多或更大的高速缓存?

12.10 在什么情况下,把内存用做一个 RAM 磁盘比用做高速缓存更为有用?

12.11 操作系统动态分配它的内部表格对用户有什么好处?操作系统这样做有什么损失?

12.12 解释为什么记录元数据更新能确保文件系统能从崩溃中恢复回来?

12.13 解释 VFS 层是怎样让操作系统容易地支持多种类型的文件系统。

12.14 设想下面的备份方法:

- **第一天**:将所有文件从磁盘拷贝到备份介质。
- **第二天**:将从第一天开始变化的文件拷贝到另一介质。
- **第三天**:将从第一天开始变化的文件拷贝到另一介质。

与 12.7.2 小节中的方法不同,并非将所有从第一次备份后改变的文件都拷贝。与 12.7.2 小节中的方法相比有什么优点?有什么缺点?恢复操作是更简单了还是更复杂了?为什么?

推荐读物

Apple[1987]和 Apple[1991]中有关于 Apple 公司 Macintosh 磁盘空间管理方案的论述。Norton 和 Wilton[1988]中解释了 MS-DOS FAT 文件系统。Iacobucci[1988]里有关于 OS/2 的描述。这些操作系统分别使用 Motorola MC68000 系列(Motorola[1989a])和 Intel 8086(Intel 1985b、Intel 1985a、Intel[1986]、Intel 1990)CPU。Deitel[1990]中描述了 IBM 公司的分配方法。McKusick 等[1996]详细论述了 BSD UNIX 系统的内部机制。McVoy 和 Kleiman[1991]给出了 SunOS 中对这些方法的优化。

Koch[1987]论述了以伙伴系统为基础的磁盘文件分配。Larson 和 Kajla[1984]论述了一个保证访问一次就检索到的文件组织方法。

McKeon[1985]和 Smith[1985]论述了磁盘缓冲。Nelson 等[1988]描述了实验性 Sprite 操作系统中的高速缓冲。Chi[1982]和 Hoagland[1985]给出了关于大容量存储技术的综合论述。Folk 和 Zoellick[1987]综述了文件结构。Silvers[2000]论述了 NetBSD 操作系统的页缓冲实现。

Sandberg 等[1985]、Sandberg[1987]、Sun[1990]和 Callaghan[2000]论述了网络文件系统(NFS)。Vahalia[1996]、Mauro 和 McDougall[2001]中描述了 NFS 和 UNIX 文件系统(UFS)。Solomon[1998]中描述了 Windows NT 文件系统、NTFS。Bovet 和 Cesati[2001]中描述了 Linux 中使用的 Ext2 文件系统。

第四部分 I/O 系 统

与计算机相连的设备在许多方面都呈现出不同。有的设备在一定时间内传输一个字符或一块字符。有的按顺序或随机访问。有的按同步或异步传输数据。有的专用或共享。有的只读或可读可写。它们速度差异很大。在许多方面它们总是计算机中最慢的部分。

由于这些差异，操作系统需要向应用程序提供各种功能，以便于控制设备的各个方面。操作系统 I/O 子系统的关键目标之一就是向系统的其他部分提供尽可能简单的接口。本书首先描述 I/O 设备的各种差异以及操作系统控制方式。接着，讨论用于次级或三级存储的更为复杂的 I/O 设备，并且解释操作系统必须给予它们的特别关注。

第十三章 I/O 系 统

计算机有两个主要任务：I/O 操作与计算处理。在许多情况下，主要任务是 I/O 操作而计算处理只是附带的。例如，当浏览网页或编辑文件时，主要兴趣在于读取或输入信息，而不是计算答案。

操作系统在计算机 I/O 方面的作用是管理和控制 I/O 操作和 I/O 设备。虽然有关问题也出现在其他章节中，但是这里将把所有部分都组合起来以描述一幅完整的 I/O 图。首先描述 I/O 硬件的基本特点，这是因为硬件接口本身会对操作系统的内部功能有所要求。然后，讨论操作系统所提供的 I/O 服务及其为应用程序所提供的接口。接着，解释操作系统如何缩小硬件接口与应用接口之间的差距。本章也将讨论 UNIX System V 的流机制，这个机制能使应用程序将设备代码动态地组合起来。最后，讨论 I/O 性能问题及用来提高 I/O 性能的操作系统设计原则。

13.1 概 述

对与计算机相连设备的控制是操作系统设计者的主要任务之一。因为 I/O 设备在其功能与速度方面存在很大差异（设想一下鼠标、硬盘及 CD-ROM 光盘塔），所以需要采用多种方法来控制设备。这些方法形成了*I/O 子系统*的核心，该子系统使内核其他部分不必涉及复杂的 I/O 设备的管理。

I/O 设备技术呈现两个相矛盾的趋势。一方面，可以看到硬件与软件接口的日益增长的标准化。这一趋势有助于人们将设备集成到现有计算机和操作系统。另一方面，可以看到 I/O 设备日益增长的多样性。有的新设备与以前的设备区别很大以至于很难集成到计算机和操作系统中。这种困难需要共同运用硬件和软件技术来解决。I/O 设备的基本要素如端口、总线及设备控制器均适用于许多不同的 I/O 设备。为了隐藏不同设备的细节与特点，操作系统内核设计成使用设备驱动程序模块的结构。**设备驱动程序**为 I/O 子系统提供了统一接口，正如系统调用为应用程序与操作系统之间提供了统一的标准接口。

13.2 I/O 硬 件

计算机使用很多种设备。绝大多数设备都属于存储设备（磁盘、磁带）、传输设备（网卡、

调制解调器)和人机交互设备(屏幕、键盘、鼠标)。其他设备则比较特别,例如控制军用战斗机或航天飞机的设备。对这类飞机,只需要通过操纵杆向飞行计算机提供输入,计算机就会送出命令控制马达从而改变方向、振动、猛冲等。

尽管 I/O 设备存在令人难以置信的差异,但只需要通过少数几个概念就可以理解如何连上设备和如何用软件来控制硬件。

设备与计算机系统的通信可以通过电缆甚至空气来传送信息。设备与计算机通信有一个连接点(或**端口**),例如,串行端口。如果一个或多个设备使用一组共同的线,那么这种连接则称为总线。**总线**(bus)是一组线和一组严格定义的可以描述在线上传输信息的协议。用电子学的话来说,信息是通过在线上的具有一定时序的电压模式来传递的。如果设备 A 通过电缆连接到 B 上,B 又通过电缆连接到 C 上,C 通过电缆连接到计算机上,那么这种方式称为**链环**(daisy chain)。链环常常按总线方式工作。

总线在计算机体系结构中使用很广。图 13.1 显示了一个典型的 PC 总线结构。该图显示了一个 PCI **总线**(最为常用的 PC 系统总线)用以连接处理机-内存子系统与快速设备,**扩展总线**(expansion bus)用于连接串行、并行端口和相对较慢的设备如键盘。在该图的右上角,四块硬盘一起连到与 SCSI 控制器相连的 SCSI 总线。

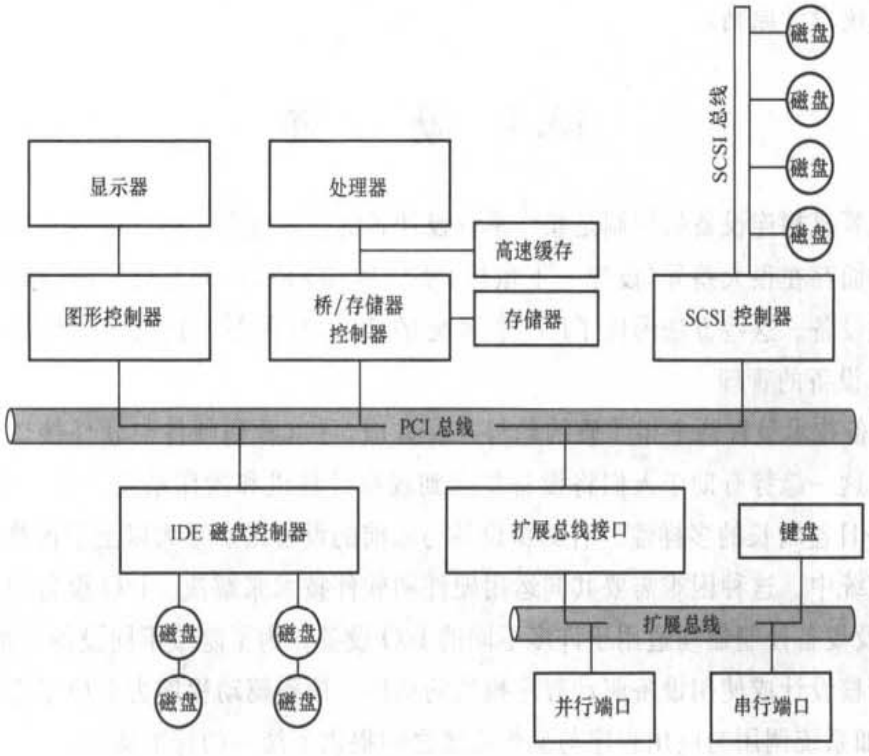


图 13.1 一个典型的 PC 总线结构

控制器(controller)是用于操作端口、总线或设备的一组电子器件。串口控制器是简单的设备控制器。它是计算机上的一块芯片或部分芯片,用以控制串口线上的信号。相比较

而言，SCSI 总线就不简单。由于 SCSI 协议的复杂，SCSI 总线控制器常常实现为与计算机相连接的独立的线路板或**主机适配器**(host adapter)。该适配器通常有处理器、微码及一定的私有内存以便能处理 SCSI 协议信息。有的设备有内置的控制器。如果观察一下磁盘，那么就会看到贴着磁盘一侧的线路板。该板就是磁盘控制器。它在磁盘边的部分实现了某种连接协议(如 SCSI 或 IDE)。它有微码和处理器以做许多任务，如坏簇映射、提前获取、缓冲和高速缓存。

那么处理器如何向控制器发送命令和数据以完成 I/O 传输？简单的回答是控制器有一个或多个用于数据和控制信号的寄存器。处理器通过读写这些寄存器的位组合来与控制器通信。这种通信的一种方法是通过使用特殊 I/O 指令来描述向 I/O 端口地址传输一个字节或字。I/O 指令触发总线线路来选择合适设备并将位信息传入或传出设备寄存器。另外，设备控制器也可支持**内存映射** I/O。这时，设备控制寄存器映射到处理器的地址空间。处理器执行 I/O 请求是通过标准数据传输指令来完成对设备控制器的读写。

有的系统使用两种技术。例如，个人计算机使用 I/O 指令来控制一些设备，而使用内存映射 I/O 指令来控制其他设备。图 13.2 显示了常用的个人计算机 I/O 端口地址。图形控制器不但有 I/O 端口以完成基本控制操作，而且也有一个较大的内存映射区域以支持屏幕内容。图形控制器可以根据图形内存内容来生成屏幕图像。这种技术使用简单。而且向图形内存中写入数百万字节要比执行数百万条指令快得多。但是向内存映射 I/O 控制器写入的简便性也存在一个缺点。因为软件出错的常见类型之一就是通过一个错误指针向一个不该写的内存区域写数据，所以内存映射设备寄存器容易受到意外修改。当然，内存保护可以减低风险。

I/O 地址范围(十六进制)	设备
000～00F	DMA 控制器
020～021	中断控制器
040～043	定时器
200～20F	游戏控制器
2F8～2FF	串行端口(辅助)
320～32F	硬盘控制器
378～37F	并行端口
3D0～3DF	图形控制器
3F0～3F7	磁盘驱动控制器
3F8～3FF	串行端口(主要)

图 13.2 个人计算机中的设备 I/O 端口位置(部分)

I/O 端口通常有四种寄存器，即*状态*、*控制*、*数据输入*与*数据输出*寄存器。

• *状态*寄存器包含一些主机可读取的位信息。这些位信息指示各种状态如当前任务是否完成，数据输入寄存器中是否有数据可以读取，是否出现设备故障等。

• *控制*寄存器可以被主机用来向设备发送命令或改变设备状态。例如，串口控制寄存器中的一位选择全工通信或单工通信，另一位控制是否奇偶校验检查，第三位设置字长为 7

或 8 位，其他位选择串口通信所支持的速度。

- *数据输入*寄存器被主机读取数据。
- *数据输出*寄存器被主机写入数据以发送数据。

数据寄存器通常为 1 至 4 个字节。有的控制器有 FIFO 芯片，可用来保留多个输入或输出数据以扩展控制器的能力而超过数据传递的大小。FIFO 芯片可以保留少量突发数据直到设备或主机来接收数据。

13.2.1 轮询(polling)

主机与控制器之间交互的完成协议可能很复杂，但基本握手概念则比较简单。可以通过举例来解释握手。假定有两个位来协调控制器与主机之间的生产者与消费者的关系。控制器通过*状态*寄存器的忙位(busy bit)来显示其状态。(记住置位就是将 1 写到位中而*清位*就是将 0 写到位中)。控制器工作忙时就置忙位；而可以接收下一命令时就清忙位。主机通过*命令*寄存器中命令*就绪*位来表示其意愿。当主机有命令需要控制器执行时，就置*命令就绪*位。例如，当主机需要通过端口来写出数据时，主机与控制器之间握手协调如下：

1. 主机不断地读取忙位，直到该位被清除。
2. 主机设置*命令*寄存器中的写位并向*数据输出*寄存器中写入一个字节。
3. 主机设置*命令就绪*位。
4. 当控制器注意到*命令就绪*位已被设置，则设置忙位。
5. 控制器读取命令寄存器，并看到写入命令。它从*数据输出*寄存器中读取一字节，并向设备执行 I/O 操作。
6. 控制器清除*命令就绪*位，清除状态寄存器的*故障*位以表示设备 I/O 成功，清除忙位以表示完成。

输出每个字节，都要执行以上循环。

在步骤 1 中，主机处于**忙等待**或**轮询**：在该循环中，不断地读取*状态*寄存器直到忙位被清除。如果控制器和设备都比较快，那么这种方法比较合理。但如果等待时间长，那么主机可能应该切换到另一任务。但是，主机又怎样知道控制器何时变为空闲？对有些设备，主机应很快地处理设备，否则数据会丢失。例如，当数据来自串口或键盘，且主机等待太长再来读取数据时，那么串口或键盘控制器上的小缓冲器可能会溢出，数据会丢失。

对许多计算机体系结构，轮询设备只要使用三种 CPU 指令就足够了：*读取*设备寄存器，*逻辑 AND* 以提取状态位，根据是否为 0 进行跳转。很明显，基本轮询操作的效率还是很高的。但是如果不断地重复轮询，主机很少发现有准备好的设备，同时其他需要使用处理器处理的工作又不能完成，则效率就会变差。这时，如果让设备准备好时再通知处理器而不是由 CPU 轮询外设 I/O 是否已完成，那么效率就会更高。能使外设通知 CPU 的硬件机制称为**中断**。

13.2.2 中断

基本中断机制工作如下。CPU 硬件有一条**中断请求线**(interrupt request line,IRL)。CPU 在执行完每条指令后,都将判断 IRL。当 CPU 检测到已经有控制器通过中断请求线发送了信号,CPU 将保留少量状态如当前指令位置,并且跳转到内存特定位置的**中断处理程序**(interrupt-handler)。中断处理程序判断中断原因,进行必要的处理,执行从中断返回(return from interrupt)指令以便使 CPU 返回中断以前的执行状态。可以说:设备控制器通过中断请求线发送*信号*而*引起*(raise)中断,CPU *捕获*(catch)中断并*派遣*(dispatch)到中断处理程序,中断处理程序通过处理设备来*清除*(clear)中断。图 13.3 总结中断驱动 I/O 循环。

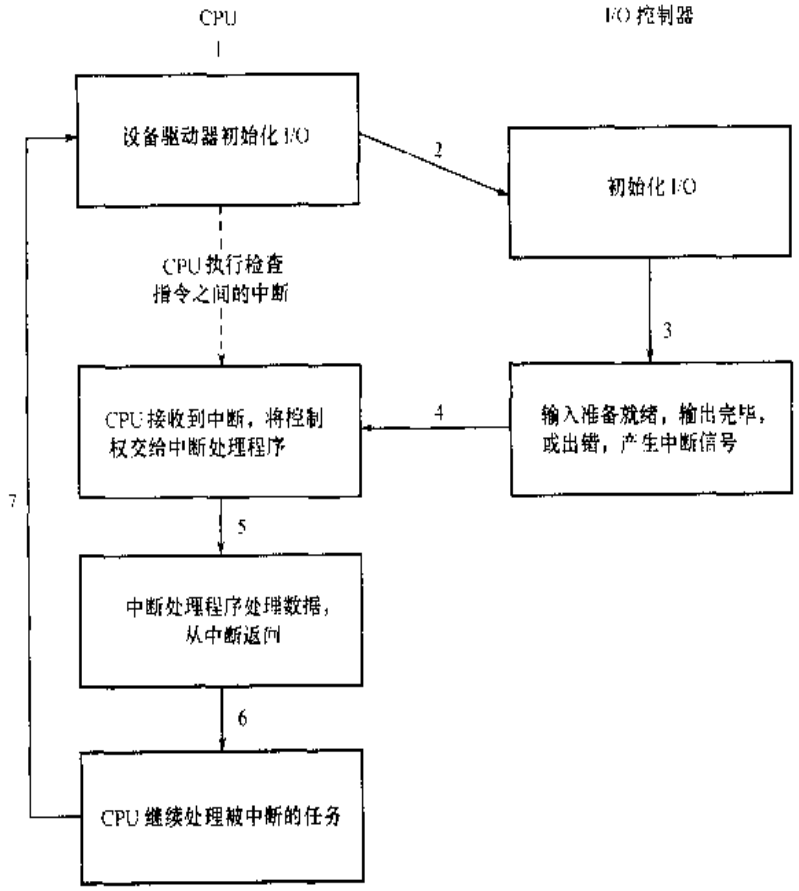


图 13.3 采用中断驱动的 I/O 循环周期

这一基本中断机制可以使 CPU 响应异步事件,例如设备控制器处于就绪状态。对于现代操作系统,需要更为复杂的中断处理特性。第一,需要在做关键处理时,能够延迟中断处理。第二,需要更为有效地将中断派遣到合适的中断处理程序,而不是检查所有设备以决定哪个设备引起中断。第三,需要多级中断,这样操作系统能区分高优先级或低优先级的中断,能根据紧迫性的程度来响应。对现代硬件,这三个特性是由 CPU 与**中断控制器**硬件提供的。

绝大多数 CPU 有两个中断请求。一个是**非屏蔽中断**,主要用来处理如不可恢复内存错

误等事件。另一个是**可屏蔽中断**,这可以由 CPU 在执行关键的不可中断的指令序列前加以屏蔽。可屏蔽中断可以被设备控制器用来请求服务。

中断机制接受一个**地址**,以用来从一小集合内选择特定的中断处理程序。对绝大多数体系机构,这个地址是一个称为**中断向量**(interrupt vector)的偏移量。该向量包含了特殊中断处理程序的内存地址。向量中断机制的目的是用来减少对单个中断处理的需要,这些中断处理搜索所有可能的中断源以决定哪个中断需要服务。事实上,计算机设备常常要比向量内的地址多。解决这一问题的常用方法之一就是中断链接技术,即中断向量内的每个元素都指向中断向量队列的头。当有中断发生时,相应链表上的所有中断处理程序都将一一调用,直到发现可以处理请求的为止。这种结构是在大型中断向量表的大开销与派遣到单个中断向量表的低效率之间的一个折中。

图 13.4 显示了 Intel Pentium 中断向量的设计。从 0 到 31 的事件为非屏蔽中断,用来表示各种错误条件信号。从 32 到 255 的事件为可屏蔽中断,用于设备所产生的中断。

向量数	描述
0	除出错
1	调试异常
2	空中断
3	断点
4	INTO—检测溢出
5	边界范围异常
6	非法操作码
7	设备不可用
8	两次出错
9	协处理器段越界(保留)
10	非法任务状态段
11	段不存在
12	堆栈出错
13	通用保护
14	页出错
15	(Intel 保留,不用)
16	浮点错误
17	对准检测
18	机器校验
19—31	(Intel 保留,不用)
32—255	可屏蔽中断

图 13.4 Intel Pentium 处理器的事件向量表

中断机制也实现了**中断优先级**(interrupt priority)。该中断机制能使 CPU 延迟处理低优先级中断而不屏蔽所有中断,这也可以让高优先级中断抢占低优先级中断处理。

现代操作系统可以与中断机制进行多种方式的交互。在启动时,操作系统探查硬件总线以发现哪些设备是存在的,而且将相应中断处理程序安装到中断向量中。在 I/O 时,各种设备控制器如果准备好服务会触发中断。这些中断表示输出已完成,或输入数据已有,或已检测到错误。中断机制也用来处理各种**异常**,如被零除,访问一个受保护的或不存在的内存地址,企图从用户态执行一个特权指令。触发中断的事件有一个共同特点:它们都是会导致 CPU 去执行一个紧迫的自我独立的程序的事件。

对于能够保存少量处理器的状态并能调用内核中的特殊程序的高效硬件,操作系统也有其他用途。例如,许多操作系统使用中断机制进行虚拟内存分页。页错误是异常,引起中断。该中断会暂停当前进程并跳转到内核的页错误处理程序。该处理程序保存进程状态,将所中断的进程加到等待队列中,进行页面缓存管理,安排一个 I/O 操作来获取所需页面,安排另一个进程恢复执行,并从中断返回。

另一个例子是系统调用的实现。*系统调用*是应用程序用来调用内核服务的函数。应用

程序检查应用程序所给的参数，建立一个数据结构将应用参数传递给内核，并执行一个称为**软中断**(software interrupt)的特殊指令(或陷阱)。该指令有一参数用来标识所需的内核服务。当系统调用执行陷阱指令时，中断硬件会保存用户态的状态，切换到内核模式，分派到实现请求服务的内核程序。陷阱所赋予的中断优先级要比设备中断优先级低；为应用程序而执行的系统调用与在FIFO队列溢出并失去数据之前处理设备控制器相比，后者更为紧迫。

中断也可以用来管理内核的控制流。例如，考虑一下完成磁盘读所需的处理。其中一步是从内核空间中将数据拷贝到用户缓冲。这个拷贝耗费时间但并不紧迫，不应该阻塞其他更高优先级的中断处理。另一步是为该磁盘驱动器启动相应的下一个I/O。这一步有更高的优先级：如果要使磁盘使用高效，必须在完成一个I/O操作之后马上启动另一个I/O操作。因此，一对中断处理程序实现了完成磁盘读的内核代码。高优先级处理记录了I/O状态，清除了设备中断，启动了下一个I/O操作，引起一个低优先级中断来完成任务。后来，当CPU没有更高优先级的工作时，将会处理低优先级中断。相应的处理将会把数据从内核缓冲拷贝到用户空间并调用进程调度程序将应用加到就绪队列，以完成用户级的I/O操作。

多线程的内核体系结构非常适合实现多优先级中断，并确保中断处理的优先级要高于内核后台处理和用户程序的优先级。可以用Solaris内核来说明这一点。Solaris的中断处理是作为线程来执行的。一定范围的高优先级保留给这些线程。这些优先级使得中断处理程序的优先级高于应用程序和内核管理的优先级，并且实现了中断处理程序之间的优先级关系。该优先级使得Solaris线程调度器用高优先级中断处理程序抢占低优先级中断处理程序，多线程实现允许多处理器硬件能同时执行多个中断处理程序。在附录A和第二十一章，将分别讨论UNIX和Windows NT的中断体系结构。

总而言之，中断在现代操作系统中用来处理异步事件和为内核的管理态程序设置陷阱。为了能使最紧迫工作先做，现代计算机都使用中断优先级。设备控制器、硬件错误、系统调用都可以引起中断并触发内核程序。因为中断大量地用于时间敏感的处理，所以高性能系统需要高效中断处理。

13.2.3 直接内存访问

对于需要做大量传输的设备，例如磁盘驱动器，如果使用昂贵的通用处理器来观察状态位并按字节来向控制器送入数据(Programmed I/O，PIO)，那么就浪费了。许多计算机为了避免用PIO(编程I/O)而增加CPU的负担，将一部分任务下放给一个的专用处理器，这称为**DMA**(direct-memory access)控制器。在开始DMA传输时，主机向内存中写入DMA命令块。该块包括传输的源地址指针、传输的目的地指针、传输的字节数。CPU在将该命令块的地址写入到DMA控制器中后，就继续其他工作。DMA控制器则继续去直接操作内存总线，无需主CPU的帮助即可以将地址放到总线以开始传输。一个简单的DMA控制器已经是个人计算机的标准部件。一般来说，采用总线控制I/O的主板都拥有它们自己的高速

DMA 硬件。

DMA 控制器与设备控制器之间的握手通过一对称为 DMA-request 和 DMA-acknowledge 的线来进行。当有数据需要传输时，设备控制器就通过 DMA-request 线发送信号。该信号会导致 DMA 控制器抓住内存总线，并在内存地址总线上放上所需地址，并通过 DMA-acknowledge 线发送信号。当设备控制器收到 DMA-acknowledge 信号时，就可以向内存传输数据，并清除 DMA-request 请求信号。

当整个传输完成后，DMA 控制器中断 CPU。图 13.5 描述了这一过程。当 DMA 控制器抓住内存总线时，CPU 会暂时不能访问主内存，但可以访问一级或二级高速缓存中的数据项。虽然**周期挪用**(cycle stealing)可能减少 CPU 计算，但是将数据传输工作交给 DMA 控制器常常能改善系统总体性能。有的计算机体系结构的 DMA 使用物理内存地址，而有的使用直接虚拟内存访问(direct virtual-memory access，DVMA)，这里所使用的虚拟内存地址需要经过虚拟到物理的地址转换。DVMA 可以直接实现两个内存映射设备之间的传输，而无需 CPU 的干涉或使用主内存。

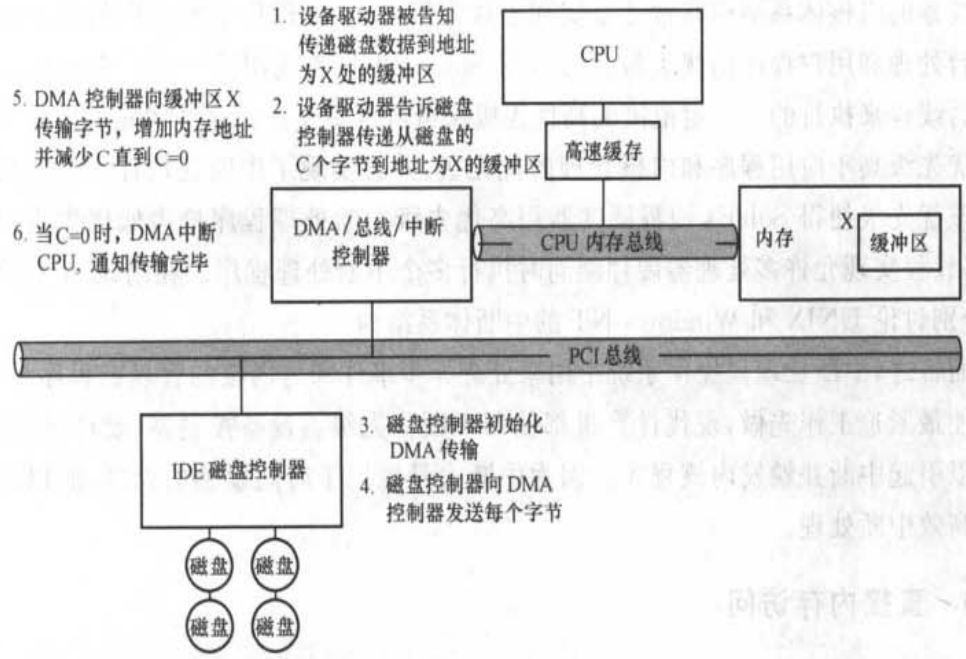


图 13.5 DMA 传输中的步骤

对于保护模式内核，操作系统通常不允许进程直接向设备发送命令。该规定保护数据以免违反访问控制，并保护系统不因设备控制器的错误使用而导致系统崩溃。操作系统有一些函数，这些函数可以被具有足够特权的进程用来访问低层硬件的底层操作。对于没有内存保护的内核，进程可以直接访问设备控制器。该直接访问可以得到高性能，这是因为它避免了内核间通信、关联切换及内核软件的分层。不过，这也妨碍了系统的安全与稳定。通用操作系统的发展趋势是保护内存和设备，这样系统可以预防错误或恶意应用程序的破坏。

虽然从电子硬件设计师角度来考虑I/O硬件是十分复杂的，但是刚才所描述的概念就足以使人理解操作系统I/O方面的许多问题。下面复习一下主要概念：

- 总线
- 控制器
- I/O端口及其寄存器
- 主机与设备控制器之间的握手关系
- 通过轮询检测或中断的握手执行
- 将大量传输下交给DMA控制器

在13.2节中，举例说明了发生在设备控制器与主机之间的握手。事实上，大量的外设为操作系统实现者提出了一个问题。每种设备都有其自己的功能，控制位定义与主机交互的协议，这些都不相同。如何设计操作系统以使得新的外设可以直接加到计算机上而不必重写操作系统？再者，由于设备种类繁多，操作系统如何提供一个统一、方便的应用程序接口？

13.3 I/O应用接口

本节讨论操作系统的组织技术与接口，以便I/O设备可以按统一的标准方式来对待。例如，将解释应用程序如何打开磁盘上的文件而不必知道是什么磁盘，及新磁盘和其他设备是如何增加到计算机中而不必中断操作系统。

与其他复杂软件工程问题一样，这里的方式包括抽象、包装与软件分层。具体地说，可以从详细而不同的I/O设备中抽象出一些通用类型。每个通用类型都可以通过一组标准函数(即**接口**)来访问。具体的差别被内核模块(称为设备驱动程序)所封装，这些设备驱动程序一方面可以定制以适合各种设备，另一方面也提供了一组标准接口。图13.6说明了内核中与I/O相关的部分是如何按软件层来组织的。

设备驱动程序层的作用是为内核I/O子系统隐藏设备控制器之间的差异，就如同I/O系统调用通用类型包装了设备行为，为应用程序隐藏了硬件差异。将I/O子系统与硬件分离简化了操作系统开发人员的任务，这也有利于硬件制造商。他们可以设计新的设备并使其与现有主机控制器接口相兼容(如SCSI-2)，或为流行操作系统编写新的设备驱动器以与新硬件交互。这样，新的外设可以与计算机相连而无需等待操作系统商家开发支持代码。

对设备硬件制造商不利的是每种操作系统都有其自己的设备驱动接口标准。一个特定设备可能带有多种设备驱动器，例如，MS-DOS驱动器、Windows 95/98驱动器、Windows NT/2000驱动器、Solaris驱动器。如图13.7，设备在许多方面都有很大差异：

- **字符流或块**：字符流设备按一个字节一个字节地传输，而块设备以块为单位进行传输。
- **顺序或随机访问**：顺序设备按其故有的固定顺序来传输数据，而随机访问设备的用户可以让设备寻找到任一数据存储位置。

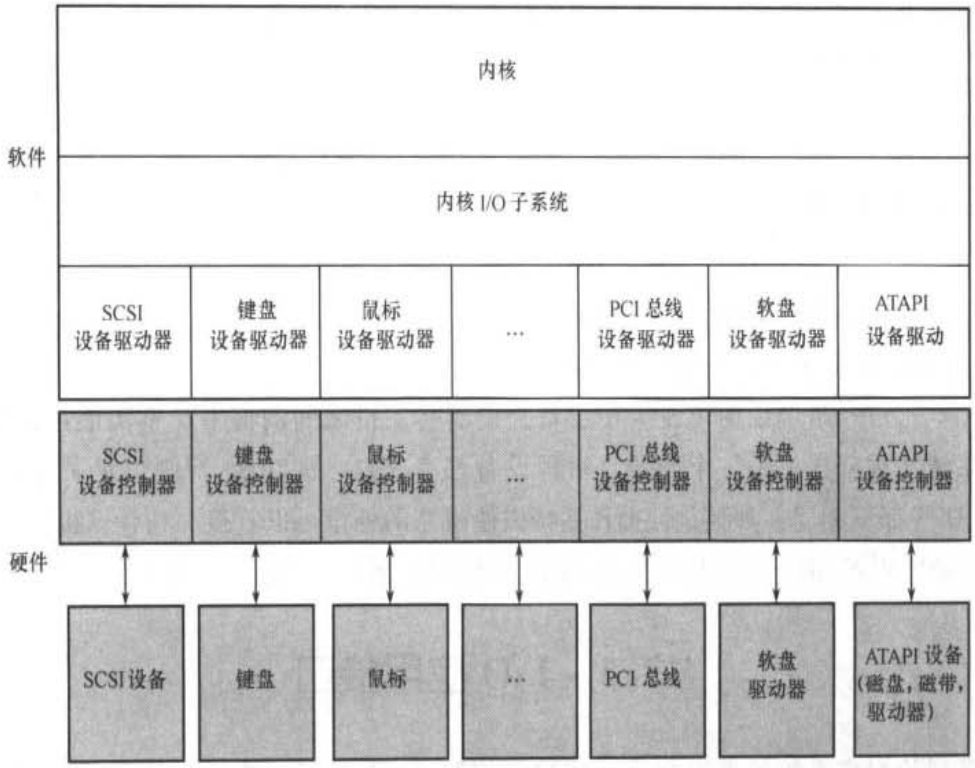


图 13.6 内核 I/O 结构

- **同步或异步**:同步设备按一定响应时间来进行数据传输,而异步设备呈现的是无规则或不可预测的响应时间。
- **共享或专用**:共享设备可以被多个进程或线程并发使用,而专用设备则不能。
- **操作速度**:设备速度从每秒几个字节到每秒数吉字节。
- **读写、只读、只写**:有的设备能读能写,而其他的只支持单向数据操作。

方面	差异	例子
数据传输模式	字符 块	终端 磁盘
访问方法	顺序 随机	调制解调器 CD-ROM
传输调度	同步 异步	磁带 键盘
共享	专用 可共享	磁带 键盘
设备速度	延迟 寻道时间 传输速率 操作之间的延迟	
I/O 方向	只读 只写 读-写	CD-ROM 图形控制器 磁盘

图 13.7 I/O 设备的特点

对应用程序访问而言，许多差别都被操作系统所隐藏，设备也分为少量的传统类型。由此产生的设备访问方式也证明十分有用并被广泛应用。虽然具体系统调用与操作系统有关，但是设备类型比较标准。主要访问方式包括块 I/O、字符流 I/O、内存映射文件访问与网络套接字。操作系统也提供特殊系统调用以访问一些其他设备，如时钟和定时器。有的操作系统为图形显示器、视频与音频设备提供一组系统调用。

绝大多数操作系统存在**后门**，这允许应用程序将任何命令透明地传递到设备控制器。对 UNIX，这个系统调用是 ioctl(I/O control)。系统调用 ioctl 能使应用程序访问由设备驱动程序所实现的一切功能，而无需再设计新的系统调用。系统调用 ioctl 有 3 个参数。第一个是文件描述符，用来将应用程序与设备驱动程序连接起来。第二个是整数，用来选择设备驱动程序所实现的命令。第三个是内存中一个数据结构的指针，这使得应用程序和驱动程序能传输任何必要的命令信息或数据。

13.3.1 块与字符设备

块设备接口规定了访问磁盘驱动器和其他基于块设备所需的各个方面。通常设备应明白 read 和 write 命令，如为随机访问设备则也应有 seek 命令来描述下次传输哪个块。应用程序通常通过文件系统接口访问设备。操作系统本身和特殊应用程序如数据库管理系统，可能更加倾向于将块设备当做一个简单的线性块数组来访问。这种访问方式有时称为**原始 I/O**。可以看到 read、write 和 seek 描述了块存储设备的基本特点，这样应用程序就不必关注这些设备的低层差别。

内存映射文件访问是建立在块设备驱动程序之上的。内存映射接口不是提供 read 和 write 操作，而是提供通过内存中的字节数组来访问磁盘存储。将文件映射到内存的系统调用返回了一个字符数组的虚拟内存地址，该字符数组包含了文件的一个拷贝。实际数据传输在需要时才执行，以满足内存映像的访问。因为传输采用了与虚拟内存访问相同的机制，所以内存映射 I/O 高效。内存映射也有益于程序员：访问内存映射文件如内存读写一样简单。提供虚拟内存的操作系统常常为内核服务提供映射接口。例如，为了执行程序，操作系统将可执行程序映射到内存中，并切换到可执行程序的入口地址。内存映射也通常被内核用来访问磁盘交换空间。

键盘是一种可以通过**字符流**接口访问的设备。这类接口的基本系统调用使得应用程序可以 get 或 put 字符。在此接口之上，可以构造库以提供具有缓冲和编辑功能的按行访问（例如，当用户键入了一个退格键，之前的字符可以从输入流中删除）。这种访问方式对有些输入很方便，如键盘、鼠标、调制解调器，这些设备自发地提供输入数据，也就是说，应用程序无法预计这些输入。这种访问方式也有助于输出设备如打印机、声卡，这些设备很适合于线性字节流的概念。

13.3.2 网络设备

由于网络 I/O 的性能和访问特点与磁盘 I/O 相比有很大差别，绝大多数操作系统所提供的网络 I/O 接口也不同于磁盘的 read-write-seek 接口。许多操作系统所提供的接口是网络**套接字**接口，如 UNIX 和 Windows NT。

想一下墙上的电源插座：任何电器用具都可以插入。同样，套接字接口的系统调用可以让应用程序创建一个套接字，连接本地套接字和远程地址（将本地应用程序与由远程应用程序创建的套接字相连），监听要与本地套接字相连的远程应用程序，通过连接发送和接收数据。为支持服务器的实现，套接字接口还提供了 select 函数，以管理一组套接字。调用 select 可以得知哪个套接字已有接收数据需要处理，哪个套接字已有空间可以接收数据以便发送。使用 select 就不必再使用轮询和忙等待来处理网络 I/O。这些函数封装了基本的网络功能，大大地加快了使用网络硬件和网络协议的分布应用程序的创建。

进程间通信和网络通信的许多其他方式也已实现。例如，Windows NT 提供了一个访问网络硬件的接口，也提供了访问网络协议的接口。UNIX 在网络通信方面有着长久的得到实践检验的良好技术，如半双工管道、全双工管道、全双工流、消息队列和套接字。关于 UNIX 网络编程的有关信息参见附录 A.9。

13.3.3 时钟与定时器

许多计算机都有硬件时钟和定时器以提供如下三个基本函数：

- 获取当前时间
- 获取已经逝去的时间
- 设置定时器以在 T 时触发操作 X。

这些函数被操作系统和时间敏感应用程序大量使用。不过，实现这些函数的系统调用并没有在操作系统之间实现标准化。

测量逝去时间和触发操作的硬件称为**可编程间隔定时器**（programmable interval timer）。它可被设置为等待一定的时间，然后触发中断。它也可以设置成做一次或重复多次以产生定时中断。调度程序可以使用这种机制来产生中断以抢占时间片用完的进程。磁盘I/O子系统用它来定时清除已改变的缓冲区，网络子系统用它来定时取消一些由于网络拥塞或故障而太慢的操作。操作系统也为用户进程提供了使用定时器的接口。操作系统通过模拟虚拟时钟而支持比定时器硬件信道数更多的定时器请求。为此，内核（或定时器驱动程序）需要维护一组由内核请求和用户请求所需要的定时器链表，该表按时间先后顺序排序。内核为最早时间设置定时器。当定时器触发中断时，内核通知请求者，并用下次最早时间重新设置定时器。

对于许多计算机，由硬件时钟产生的中断约在每秒 18 次到 60 次。这种分辨率相对粗

糙，因为现代计算机每秒可执行数百万条指令。触发器的精度受到定时器的粗糙分辨率和维护虚拟时钟的开销的限制。如果定时器用来维持系统时钟，那么系统时钟就会偏移。对绝大多数计算机，硬件时钟是由高频率时钟计数器来构造的。对于有的计算机，计数器的值可通过设备寄存器来读取，这可作为高精度的时钟。虽然这种时钟不产生中断，但它可以提供时间间隔的精确测量。

13.3.4 阻塞与非阻塞 I/O

系统调用接口的另一方面同阻塞与非阻塞(异步)I/O 的选择有关。当应用程序发出一个**阻塞**系统调用时，应用程序的执行就被中止。应用程序将会从操作系统的运行队列移到等待队列中去。在系统调用完成后，应用程序就移回到运行队列，在适合的时候继续执行并能收到通过系统调用返回的值。由 I/O 设备执行的物理动作常常是异步的：其执行时间是可变的或不可预计的。然而，绝大多数操作系统应用程序接口阻塞系统调用，这是因为阻塞应用代码比非阻塞应用代码更容易理解。

有的用户级进程需要使用**非阻塞 I/O**。一个例子是用户接口，用来接收键盘和鼠标输入而同时又要处理并在屏幕上显示数据。另一个例子是一个视频应用程序，用来从磁盘文件上读取帧同时解压缩并在显示器上显示输出。

应用程序开发可以重叠 I/O 执行的方法是编写多线程应用程序。有的线程执行阻塞系统调用，而其他的继续执行。Solaris 开发人员使用这种技术来实现用户级的异步 I/O 库，使得应用程序开发人员不必考虑这一任务。有的操作系统提供非阻塞系统调用。一个非阻塞调用并不使应用程序的执行停顿过长。而它会很快返回，其返回值表示已经传输了多少字节。

除了非阻塞系统调用外，还有异步系统调用。异步调用不必等待 I/O 完成就可立即返回。应用程序继续执行其代码。在将来 I/O 完成时可以通知应用程序，通知方式可以是设置应用程序地址空间内的某个变量，或通过触发信号或软件中断，或应用程序执行流程之外的某个回调函数。非阻塞与异步系统调用的差别是非阻塞 read 调用会马上返回，其所读取的数据可以等于或少于所要求的，或为零。异步 read 调用所要求的传输应完整地执行，其具体执行可以是将来某个特定时间。

一个很好的非阻塞例子是用于网络套接字的 select 系统调用。该系统调用有一个参数以描述最大等待时间。如果设置为 0，应用程序可以轮流检测网络活动而无需等待。但是使用 select 也有额外开销，这是因为 select 调用只检查是否可能进行 I/O。对于数据传输，在 select 之后，还需要使用 read 或 write 命令。在 Mach 中，有这种方法的变种，即阻塞多读调用。通过这一系统调用可以对多个设备描述进行读，只要一个完成就返回。

13.4 I/O 内核子系统

内核提供了许多与 I/O 有关的服务。许多服务如调度、缓冲、高速缓冲、假脱机、设备预

留及错误处理是由内核 I/O 子系统提供的，并建立在硬件和设备驱动程序结构之上。

13.4.1 I/O 调度

调度一组 I/O 请求就是确定一个好的顺序来执行这些请求。应用程序所发布的系统调用的顺序并不一定总是最佳选择。调度能改善系统整体性能，能在进程之间公平地共享设备访问，能减少 I/O 完成所需要的平均等待时间。这里是一个简单的例子以说明这种情况。假设磁头位于磁盘开始处，三个应用程序向该磁盘发布阻塞读调用。应用程序 1 需要磁盘结束部分的块，应用程序 2 需要磁盘开始部分的块，应用程序 3 需要磁盘中间部分的块。操作系统如果按照 2、3、1 的顺序处理，则可以减低磁头所需移动的距离。按这种方法来重新安排服务顺序就是 I/O 调度的核心。

操作系统开发人员通过为每个设备维护一个请求队列来实现调度。当一个应用程序执行阻塞 I/O 系统调用时，该请求就加到相应设备的队列上。I/O 调度重新安排队列顺序以改善系统总体效率和应用程序的平均响应时间。操作系统可以试图公平，这样没有应用程序会得到特别不良的服务；也可以给予那些对延迟很敏感的请求比较优先的服务。例如，虚拟内存子系统的请求会比应用程序的请求要优先。磁盘 I/O 的若干调度算法在第 14.2 节中做详细描述。

I/O 子系统改善计算机效率的一种方法是进行 I/O 操作调度。另一种方法是使用主存或磁盘上的存储空间的技术如缓冲、高速缓冲、假脱机。

13.4.2 缓冲

缓冲区是用来保存在两设备之间或在设备和应用程序之间所传输数据的内存区域。采用缓冲有三个理由。一个理由是处理数据流的生产者与消费者之间的速度差异。例如，假如从调制解调器收到一个文件，并保存到硬盘上。调制解调器大约比硬盘慢数千倍。这样，可以在内存中创建缓冲区以累积从调制解调器处收到的字节。当整个缓冲区填满时，就可以通过一次操作将缓冲区写入到磁盘中。由于写磁盘并不即时而且调制解调器需要一个空间继续保存输入数据，因而需要两个缓冲区。当调制解调器填满第一个缓冲区后，就可以请求写磁盘。接着调制解调器开始填写第二个缓冲区，而这时第一个缓冲正被写入磁盘。等到调制解调器写满第二个缓冲区时，第一个缓冲区也应已写入磁盘，因此调制解调器可以切换到第一个缓冲区而磁盘可以写第二个缓冲区。这种**双缓冲**将生产者与消费者进行分离，因而缓和两者之间的时序要求。关于需要缓和可以参见图 13.8，该图列出了常用计算机硬件设备速度的巨大差异。

缓冲的第二个用途是协调传输数据大小不一致的设备。这种不一致在计算机网络中特别常见，缓冲常常用来处理消息的分段和重组。在发送端，一个大消息分成若干小网络包。这些包通过网络传输，接收端将它们放在重组缓冲区内以生成完整的源数据镜像。

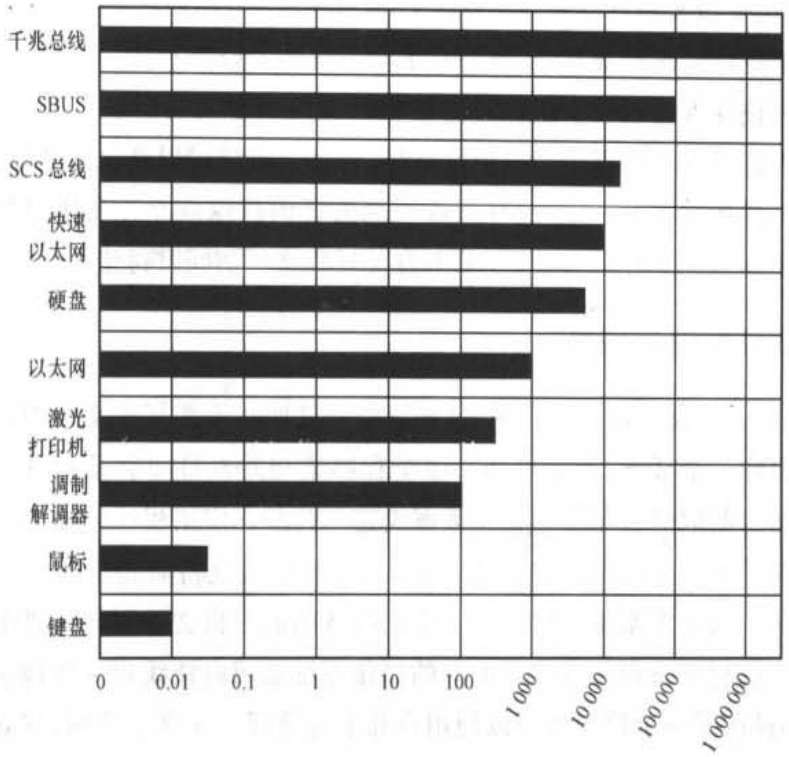


图 13.8 Sun Enterprise 6000 的设备传输率(对数形式)

缓冲的第三个用途是应用程序 I/O 的拷贝语义。一个例子可以阐明“拷贝语义”的含义。假如某应用程序需要将缓冲区内的数据写入到磁盘。它可以调用 write 系统调用,并给出缓冲区的指针和表示所写字节数量的整数。当系统调用返回时,如果应用程序改变了缓冲区中的内容,那么会如何呢? 根据**“拷贝语义”**,操作系统保证要写入磁盘的数据就是 write 系统调用发生时的版本,而无需顾虑应用程序缓冲区随后发生的变化。一个简单方法就是操作系统在 write 系统调用返回到应用程序之前将应用程序缓冲区复制到内核缓冲区中。磁盘写会从内核缓冲中执行,这样后来应用程序缓冲区的改变就没有影响。操作系统常常使用内核缓冲和应用程序数据空间之间的数据复制,尽管这会有一定的开销但是获得了简洁的语义。类似地,通过使用虚拟内存映射和写复制保护也可能会提供更为高效的结果。

13.4.3 高速缓存

高速缓存(cache)是可以保留数据拷贝的高速内存。高速缓存拷贝的访问要比原始数据访问更为高效。例如,正在运行的进程的指令既存储在磁盘上,也存储在物理内存上,也被复制到 CPU 的二级和一级高速缓存中。缓冲与高速缓存的差别是缓冲只保留数据仅有的一个现存拷贝,而根据定义高速缓存只是提供了一个驻留在其他地方的数据的一个高速拷贝。

高速缓存和缓冲是两个不同的功能,但有时一块内存区域也可以同时用于两个目的。

例如，为了维护拷贝语义和有效调度磁盘I/O，操作系统在内存中开辟缓冲区来保留磁盘数据。这些缓冲也可用做高速缓存，以改善对某些文件的I/O操作效率，这些文件可被多个程序所共享或者被快速地写入和重读。当内核收到I/O请求时，内核首先检查高速缓存以确定相应文件的内容是否在内存中。如果是，物理磁盘I/O就可以避免或延迟。而且，几秒内的磁盘写也可以累加在缓冲区中，这样就收集到大量的传输以允许高速的写调度。为了改善I/O效率而延迟写的策略将在16.3节中有关远程文件访问时讨论。

13.4.4 假脱机与设备预留

spool是用来保存设备输出的缓冲，这些设备如打印机不能接收交叉的数据流。虽然打印机只能一次打印一个任务，但是可能有多个程序希望并发打印而又不将其输出混合在一起。操作系统通过截取对打印机的输出来解决这一问题。应用程序的输出先是假脱机到一个独立的磁盘文件上。当应用程序完成打印时，假脱机系统将对相应的待送打印机的假脱机文件进行排队。假脱机系统一次拷贝一个已排队的假脱机文件到打印机上。有的操作系统采用系统服务进程来管理假脱机。而有的操作系统采用内核线程来处理假脱机。不管怎样，操作系统都提供了一个控制接口以便用户和系统管理员来显示队列，删除那些尚未打印的而不再需要的任务，当打印机工作时暂停打印，等等。

有的设备(如磁带和打印机)不能有效地多路复用多个并发应用程序的I/O请求。假脱机是一种操作系统可以用来协调并发输出的方法。处理并发设备访问的另一个方法是提供协调所需要的工具。有的操作系统(包括VMS)提供对专用设备访问的支持，如允许进程分配一个空闲设备以及不再需要时再释放该设备。而有的操作系统则对这种设备的打开文件句柄有所限制。许多操作系统都对允许进程的互斥访问提供了功能。例如，Windows NT提供的系统调用可以等到设备对象有用为止。有些操作系统的系统调用open也可有一个参数以表示其他并发线程所允许的访问类型。对这些系统，应用程序需要自己来避免死锁。

13.4.5 错误处理

采用内存保护的操作系统可以预防许多硬件和应用程序的错误，这样就不会因为小的机械失灵导致完全的系统崩溃。设备和I/O传输的出错有多种方式，有的短暂如网络过载，有的永久如磁盘控制器缺陷。操作系统可以对短暂出错进行弥补。例如，磁盘read出错可以导致read重试，网络send出错可以导致resend(如果协议允许)。但是，如果某个重要系统组件出现了永久错误，那么操作系统就不可能从中恢复。

作为一个规则，I/O系统调用通常返回一位调用状态信息，以表示成功或失败。对于UNIX操作系统，一个名为errno的额外全局变量用来表示出错代码，约有100个，以表示失败的一般原因(例如，参数超过范围、坏指针、文件未打开)。与此相对照，有的硬件能提供很详细的出错信息，虽然操作系统并不将这些信息传递给应用程序。例如，SCSI设备的失

败可以通过 SCSI 协议按表达失败一般信息的 **sense key**，或按表达失败类型如错误命令参数或自检失败的 **additional sense key**，或按表达更详细信息如哪个命令参数出错或哪个硬件子系统自检失败的 **additional sense-code qualifier** 来报告。再者，许多 SCSI 设备都维护一个出错日志信息以便主机查询，不过这一功能实际很少使用。

13.4.6 内核数据结构

内核需要保存 I/O 组件使用的状态信息，可以通过若干内核数据结构如文件打开表等来完成，参见 12.1 节。内核使用许多类似的结构来跟踪网络连接、字符设备通信和其他I/O活动等。

UNIX 提供对若干实体如用户文件、原设备和进程地址空间的文件系统访问。虽然所有实体都支持 read 操作，但是语义不同。例如，读一个用户文件时，内核需要先检查一下缓存，然后再决定是否执行磁盘 I/O。读一个原始磁盘时，内核需要确保所请求的大小是磁盘扇区大小的倍数而且与扇区边界对齐。读一个进程映像时，内核只需要从内存中读取数据。UNIX 通过面向对象技术采用统一结构来包装这些区别。打开文件的记录包括一个分派表，该表含有与文件类型相对应的程序的指针，参见图 13.9。

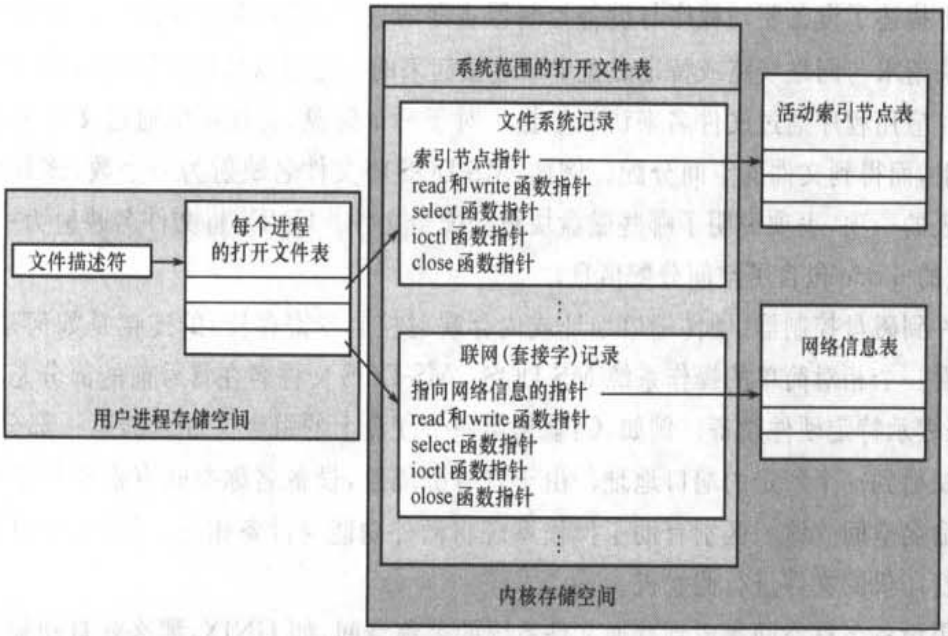


图 13.9 UNIX 的 I/O 内核结构

有的操作系统更为广泛地使用了面向对象方法。例如，Windows NT 的 I/O 采用消息传递的实现。一个 I/O 请求首先转换成一条消息，而后再通过内核传递给 I/O 管理器，再到设备驱动程序，以下每一步都可能改变消息内容。对于输出，消息内容包括要写的数据。对于输入，消息包括接收数据的缓冲。消息传递，与采用共享数据结构的子程序调用技术相比，会增加开销，但是它能简化 I/O 系统的结构和设计并增加灵活性。

总而言之，I/O 子系统协调为应用程序和内核内部所提供的许多服务的集合。I/O 子系统负责：

- 文件和目录的命名空间的管理
- 文件和目录的访问控制
- 操作控制(例如，调制解调器不能使用 seek)
- 文件系统空间分配
- 设备分配
- 缓冲、高速缓存和假脱机
- I/O 调度
- 设备状态监控、错误处理、失败恢复
- 设备驱动程序的配置和初始化

I/O 子系统的上层通过设备驱动程序所提供的统一接口来访问设备。

13.5 把 I/O 操作转换成硬件操作

以上描述了设备驱动程序与设备控制器之间的握手，但尚未解释操作系统是如何将应用程序的请求与网络线路或特定磁盘扇区连接起来的。这里以从磁盘读文件为例来考虑这一问题。应用程序通过文件名来访问数据。对于一个磁盘，文件系统通过文件名从文件目录的映射，而得到文件的空间分配。例如，MS-DOS 将文件名映射为一个数，该数显示了文件访问表的一项，表项说明了哪些磁盘块被分配给文件。UNIX 将文件名映射为一个 inode 号，相应的 inode 包含了空间分配信息。

文件到磁盘控制器(硬件端口地址或内存映射控制器寄存器)的连接是如何建立的呢？首先来看一个相对简单的操作系统 MS-DOS。MS-DOS 文件名在冒号前的部分是一个字符串，用来表示特定硬件设备。例如，C:就是第一个硬盘上的每个文件名的第一部分；C:通过设备表映射到一个特定的端口地址。由于冒号分隔符，设备名称空间有别于每个设备内文件系统命名空间。这一区别有助于操作系统将额外功能与设备相连。例如，可以容易地对输出到打印机的文件进行假脱机。

如果设备名称空间集成到普通文件系统的名称空间，如 UNIX，那么就自动提供了普通文件系统名称服务。如果文件系统提供对所有文件名称进行所有权和访问控制，那么设备就有所有权和访问控制。由于文件保存在设备上，这种接口提供对 I/O 系统的双层访问。名称能用来访问设备，也能用来访问存储在设备上的文件。

UNIX 通过普通文件系统命名空间来给设备命名。与 MS-DOS 文件名称不一样(即有冒号分隔符)，UNIX 路径名并不区别设备部分。事实上，路径名中没有设备名称。UNIX 有一个**装配表**(mount table)用来将路径名的前缀与特定设备名称相连。为了解决路径名，

UNIX 检查装配表内的名称以找到最长的匹配前缀；装配表内的相应项就给出了设备名称。该设备名称在文件系统命名空间内也有一名称。当 UNIX 在文件系统目录结构内查找该名称时，得到的不是 inode 号，而是设备号<主，次>。主设备号表示设备驱动程序，用来处理该设备的 I/O。次设备号传递给设备驱动程序以查找设备表。设备表内的相应条目给出设备控制器的端口地址或内存映射地址。

现代操作系统通过对一个请求与物理设备控制器路径之间多级表查找可以获得巨大的灵活性。应用程序与驱动程序之间的请求传递机制是通用的。因此，不必重新编译内核也能为计算机引入新设备和新驱动程序。事实上，有的操作系统能够按需调入设备驱动程序。导入时，系统首先检测硬件总线以确定有哪些设备，接着操作系统就马上或等首次 I/O 请求时装入所需驱动程序。

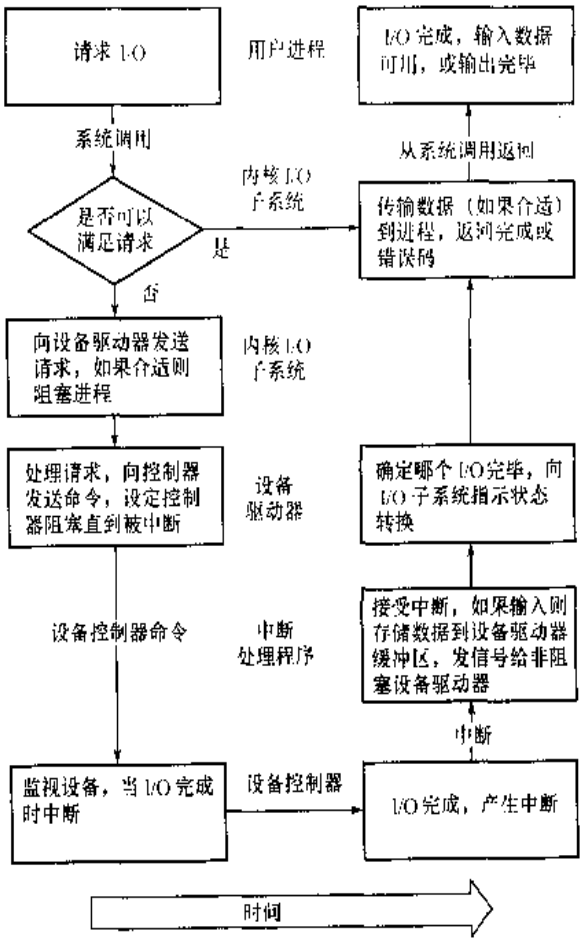


图 13.10 I/O 请求的生命周期

现在，描述阻塞读请求的典型周期，参见图 13.10。该图说明了 I/O 操作需要很多步骤，这也消耗大量 CPU 周期。

1. 一进程对已打开文件的文件描述符调用阻塞 read 系统调用。

2. 内核系统调用代码检查参数是否正确。对于输入，如果数据已在缓冲中，那么就将该数据返回给进程并完成 I/O 请求。

3. 否则，就需要执行物理 I/O 请求。这时，该进程会从运行队列移到设备的等待队列上，并调度 I/O 请求。最后 I/O 子系统对设备驱动程序发出请求。根据操作系统的不同，该请求可能通过子程序调用或内核消息传递。

4. 调用驱动程序分配内核缓冲空间以接收数据，并调度 I/O。最后，设备驱动程序通过写入设备控制器寄存器来对设备控制器发送命令。

5. 设备控制器控制设备硬件以执行数据传输。

6. 驱动程序可以轮流检测状态和数据，或通过设置 DMA 将数据传入到内核内存。假定 DMA 控制器管理传输，当传输完成后会产生中断。

7. 合适的中断处理程序通过中断向量表收到中断，保存必要的数据，通知内核设备驱

动程序，从中断返回。

8. 设备驱动程序接收到信号，确定 I/O 请求是否完成，确定请求状态，并通知内核 I/O 子系统请求已完成。

9. 内核将数据或返回代码传递给请求进程的地址空间，将进程从等待队列移到就绪队列。

10. 将进程移到运行队列使该进程不再阻塞。当调度给该进程分配 CPU 时，该进程就继续在系统调用完成后执行。

13.6 流

UNIX V 系统有一个有趣的机制，称为 **STREAM**(流)，它能让应用程序动态地组合驱动程序代码流水线。流是在设备驱动程序和用户级进程之间的全工连接。它包括与用户相连的**流开始**、控制设备的**驱动程序结尾**、位于这两者之间的若干个**流模块**。流开始、驱动程序结尾和流模块都有一对队列：读队列和写队列。队列之间的数据传输使用消息传递。图 13.11 显示了一个 STREAM 结构。

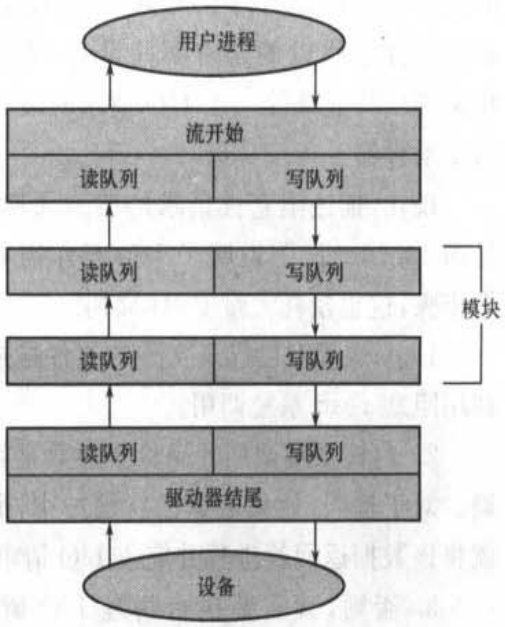


图 13.11 流的结构

提供了 STREAM 处理功能的模块可以通过使用 ioctl 系统调用*增加*到流上。例如，一个进程能通过流打开一个串口设备，并能增加一个模块来处理输入编辑。因为相邻模块队列之间可以交换消息，所以一个模块的队列可能会溢出相邻的模块。为了防止这种事情发生，队列可以支持**流控制**。如没有流控制，则队列接收所有消息并马上而不加以缓冲地传给邻近模块的队列。如有流控制，则队列会缓冲消息而且如没有足够缓冲空间就不会接收消息。流控制可以通过交换相邻队列之间的控制消息来实现。

用户进程采用 write 或 putmsg 系统调用来将数据写入到设备。系统调用 write 将原始数据写入到流中，而 putmsg 允许用户进程描述消息。不管用户进程使用何种系统调用，流开始将数据拷贝到消息中并递交给下一模块。消息不断地拷贝一直到交给驱动程序结尾和设备。类似地，用户进程采用 read 和 getmsg 系统调用来从流开始处读取数据。如果使用 read，流开始从相邻队列得到消息，并将普通数据(无结构的字节流)返回给进程。如果使用 getmsg，将消息返回给进程。

除非用户进程与流开始直接通信，流 I/O 是异步的(或无阻塞的)。当对流写时，如果下

一队列使用流控制而且需要等待空间拷贝消息，那么用户进程会阻塞。同样，当对流读时，如果需要等数据，那么用户进程会阻塞。

流结尾与流开始和模块相似，也有读和写队列。然而，驱动程序端必须响应中断，比如如果网络上有帧就绪需要被读取时会触发中断。与流开始可以阻塞不一样，它不能将消息拷贝到下一队列，而必须处理所有输入数据。驱动程序也可以支持流控制。然而，如果设备缓冲已满，那么设备通常需要扔掉输入数据。考虑一个网卡，其输入缓冲已满。该网卡必须扔掉后面的消息直到有足够的缓冲来存储输入消息。

使用流的好处是流可以提供一个框架以便模块化地递增地编写设备驱动程序和网络协议。

模块可以为不同的 STREAM 以及不同的设备所使用。例如网络模块可以被以太网和令牌网所共用。而且，不只是将字符设备的输入数据作为非结构化的数据流，STREAM 还支持消息边界和模块之间的控制信息。流在 UNIX 或其变种中得到广泛使用，是较好的编写协议和设备驱动程序的方法。例如，在 UNIX V 和 Solaris 中，采用流来实现套接字机制。

13.7 性　能

I/O 是影响系统性能的重要因素之一。执行设备驱动程序代码以及随着进程阻塞变化而公平且高效地调度进程都增加了 CPU 的负荷。由此而产生的关联切换也增加了 CPU 及其硬件高速缓存的负担。I/O 会暴露内核中断机制的任何效率缺陷，它也会使得用于控制器和物理内存之间的数据拷贝以及内核缓冲和应用程序数据空间之间的数据拷贝时的内存总线的负荷过重。很好地处理这些需求是操作系统设计师所要关心的问题之一。

虽然现代计算机每秒能处理数千个中断，但是中断处理仍然是相对较为费时的任务：每个中断都会导致系统改变状态，执行中断处理，再恢复状态。如果程序控制 I/O 所需要忙等待的计算机周期并不多，那么程序控制 I/O 可能比中断驱动 I/O 更为有效。I/O 完成通常会释放阻塞进程，进而会产生关联切换的全部开销。

网络交通也能导致更高的关联切换率。例如，设想一下有一个从一台计算机到另一台计算机的远程登录。本地计算机上所键入的每一个字节都要传送到另一台计算机。在本地计算机上，每键入一个字符，都会产生一个中断，该字符通过中断处理程序传给设备驱动程序，再传给内核，最后到用户进程。用户进程执行一个网络 I/O 调用来将该字符送到远程计算机。该字符流入本地内核，通过构造网络包的网络层，再到网络设备驱动程序。网络设备驱动程序将该包送交网络控制器以发送该字符并产生中断。该中断传回到内核以完成网络 I/O 系统调用。

这时，远程系统的网络硬件收到包，并产生中断。该字符被网络协议解包后再送给网络服务程序。网络服务程序确定与哪个远程登录会话有关，进而将该包交给合适会话的子服务程序。整个流程有许多关联切换和状态变化，参见图 13.12。通常，接收者会将该字符送

回给发送者；这种方式会使工作量加倍。

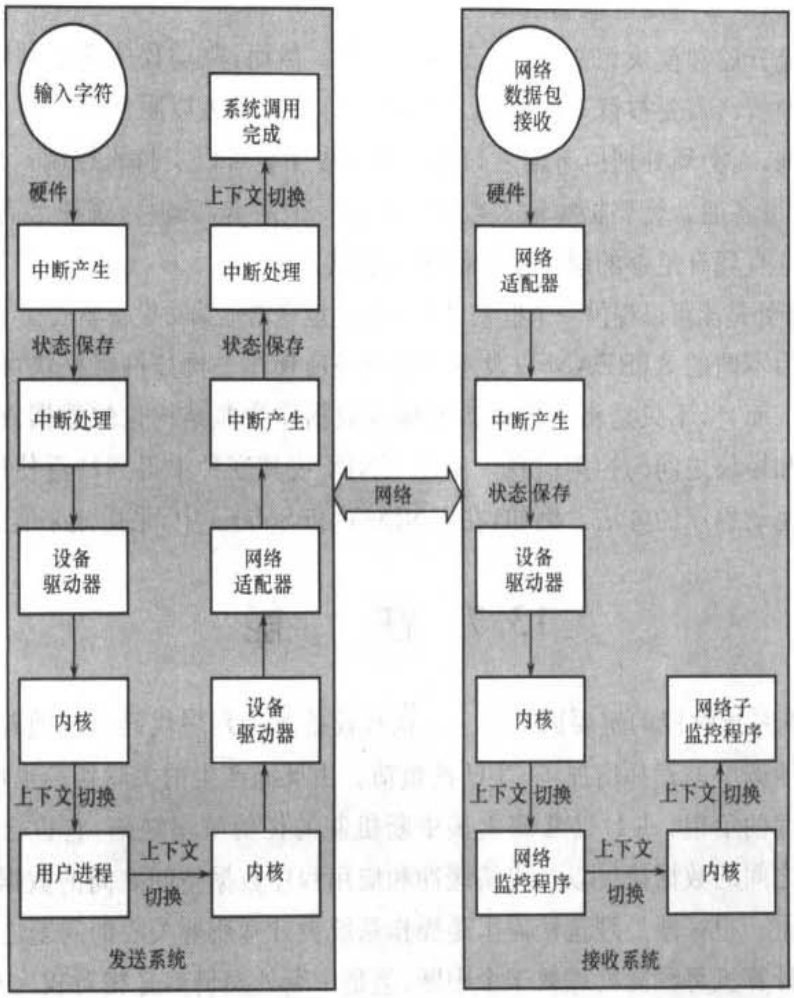


图 13.12 计算机之间的通信

Solaris 开发人员利用内核线程重新实现了 **telnet** 服务程序，消除了在服务程序与内核之间移动字符所涉及的关联切换。Sun 公司估计这一改进会使得服务器上网络登录数量从几百增加到几千。

其他系统为终端 I/O 使用了独立的**前端处理器**，以减轻主 CPU 的中断负担。例如，**终端集中器**可以将数百个远程终端多路复用成大型计算机上的一个端口。**I/O 通道**是大型机和高端系统所使用的专用 CPU。I/O 通道的任务是为主 CPU 承担 I/O 工作。主要思想是通道保持数据平稳流动，而主 CPU 就可以处理数据。与小型计算机所使用的设备控制器和 DMA 控制器一样，通道可以处理更多的通用和复杂的程序，这样通道就可用来调节工作负载。

为了改善 I/O 效率，可以采用一些原则：

- 减少关联切换的次数。
- 减少设备和应用程序之间传递数据时在内存中的数据拷贝次数。
- 通过使用大传输、智能控制器及轮流检测，来减少中断频率。

• 通过采用 DMA 智能控制器和通道来为主 CPU 承担简单数据拷贝，以增加并发。

• 将处理原语移入硬件，允许控制器内的操作与 CPU 和总线内的操作并发。

• 平衡 CPU、内存子系统、总线和 I/O 的性能，这是因为任何一处的过载都会引起其他部分空闲。

设备复杂性的差异很大。例如，鼠标比较简单。鼠标移动和按钮单击转换为数值，该值从硬件传递过来，经过鼠标驱动程序，到应用程序。相反，NT 磁盘设备驱动程序所提供的功能复杂。它不但管理单个磁盘也能实现 RAID 阵列(参见 14.5 节)。为了这样做，它将应用程序的读写请求转变为一组协调的 I/O 操作。而且，它也实现了高难度出错处理和数据恢复算法，采用很多步骤来优化磁盘操作，这是因为二级存储性能对系统的整体性能影响很大。

I/O 功能到底应在哪里实现呢？是硬件层还是设备驱动程序或是应用程序软件？有时，可以观察到如图 13.13 所示的发展过程。

• 起初，会在应用程序层实现试验性的 I/O 算法，这是因为应用程序灵活且应用程序错误不会使系统崩溃。再者，由于在应用程序级开发代码，所以避免了因代码修改而需要的重新启动或重新装载设备驱动程序。然而，由于关联切换，应用程序不能充分利用内部内核数据结构和内核功能(如高效内核消息传递、多线程和锁定等)，应用程序级的实现可能效率较低。

• 如果应用程序算法已验证了其价值，那么就可以在内核级重新实现。这能改善其性能，但由于操作系统是一个复杂且庞大的软件系统，开发工作可能会更为困难。而且，内核实现必须经过完全调试以避免数据损坏和系统崩溃。

• 高性能可以通过设备或控制器的硬件来专门实现。硬件实现的不利因素包括做进一步改进或除去错误比较困难和昂贵，增加了开发时间(数月而不是数天)且灵活性降低。例如，即使内核有关于负荷的特定信息可以使得内核改善 I/O 性能，硬件 RAID 控制器也不能帮助内核来对各个块的读和写有任何影响。

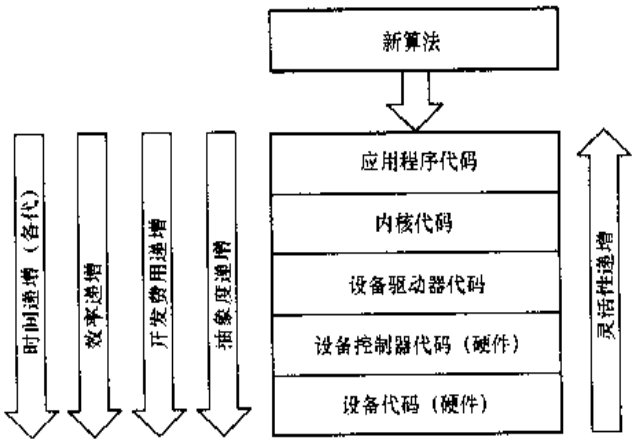


图 13.13　设备功能的发展

13.8 小 结

I/O 有关硬件的基本要素是总线、设备控制器和设备本身。设备与内存间的数据移动工作是由 CPU 按程序控制 I/O 来完成的，或转交给 DMA 控制器。控制设备的内核模块称为设备驱动程序。为应用程序所用的系统调用设计为处理若干基本类型的硬件，包括块设备、字符设备、内存映射文件、网络套接字、可编程间隔定时器。系统调用通常会使发布命令的进程阻塞，但是非阻塞和异步调用可以为内核自己所使用，也可为不需要等待 I/O 完成的应用程序所使用。

内核 I/O 子系统提供了若干服务。其中有 I/O 调度、缓冲、假脱机、出错处理和设备预留。另一个服务是名称转换，即在硬件设备和应用程序所用和符号文件名之间建立连接。这包括多次映射，可以将字符文件名称映射到特定设备驱动程序和设备地址，然后到 I/O 端口和总线控制器的物理地址。这一映射可以包含在文件系统名称空间内，如同 UNIX，也可以出现在独立的设备名称空间内，如 MS-DOS。

STREAM 是使得设备驱动程序可以重用和更容易使用的一种实现方法。通过 STREAM，驱动程序可以堆叠，数据可以按单向和双向来进行传输和处理。

由于物理设备和应用程序之间的多层软件，I/O 系统调用所用的 CPU 周期较多。这些层意味着多种开销，如穿过内核保护边界的关联切换，用于 I/O 设备的信号和中断处理，用于内核缓冲和应用程序空间之间的数据拷贝所需要的 CPU 和内存系统的负荷。

习题十三

13.1 针对将功能放在设备控制器而不是内核的情况，陈述三个优点和三个缺点。

13.2 考虑以下单用户计算机的 I/O 情况：

a. 用于图形用户界面的鼠标

b. 多任务操作系统的磁带驱动器(假定并没有设备预分配)

c. 包含用户文件的磁盘驱动器

d. 能直接与总线相连且可以通过内存映射 I/O 来访问的图形卡

针对这些 I/O 情况，您将如何设计操作系统来使用缓冲、假脱机、高速缓存或多种技术的组合？您将使用轮询检测 I/O 或是中断驱动 I/O？请给出选做选择的理由。

13.3 13.2 节所述的握手例子使用了两个位：忙位和命令就绪位。是否能用一位来实现这一握手？如果能，请描述该协议。如果不能，请说明为什么一位是不够的。

13.4 描述三种适合使用阻塞 I/O 的情况。描述三种适合使用非阻塞 I/O 的情况。为什么不能只实现非阻塞 I/O 而让进程忙等直至其设备就绪。

13.5 为什么一个系统会使用中断驱动 I/O 来管理单个串口，而使用轮询 I/O 来管理前端处理器如终端集中器？

13.6 如果处理器需要在为完成I/O前重复多次忙等循环，那么轮询I/O完成的检查就会浪费大量CPU周期。但是如果I/O设备已就绪等待服务，那么轮询可能比捕捉中断和分派中断更为有效。描述一种混合策略以便将I/O设备服务的轮询、睡眠和中断组合起来。对这三种(纯轮询、纯中断、混合方法)中的每一种策略，描述它比其他策略更为有效的计算环境。

13.7 UNIX通过操作共享内核数据结构来协调内核I/O组件的活动，而Windows NT却是通过内核I/O组件间的面向对象消息传递来完成的。请说出每种方法的三个优点和三个缺点。

13.8 DMA如何增加系统并发率？DMA为什么会增加硬件设计的复杂性？

13.9 用伪代码写出虚拟时钟的实现，包括内核和应用程序定时器请求的排队和管理。假定硬件提供三个定时器通道。

13.10 随着CPU速度的增加，按比例增加系统总线和设备的速度很重要，这是为什么？

13.11 描述STREAM驱动程序和STREAM模块的区别。

推荐读物

Vahalia[1996]对UNIX的I/O和网络做了很好的概述。Leffler等[1989]详细说明了用于BSD UNIX的I/O结构和方法。Milenkovic[1987]讨论了I/O方法和实现的复杂性。Stevens[1992]探讨了UNIX的各种进程间通信和网络协议的使用和编程。Brain[1996]说明Windows NT应用程序接口。Tanenbaum和Woodhull[1997]描述了MINIX OS的I/O实现。Custer[1994]对NT I/O消息传递的实现做了详细描述。

关于硬件级I/O处理和内存映射功能的细节，最好资料是处理器参考手册(Motorola[1993]和Intel[1995])。Hennessy与Patterson[1996]描述了多处理器系统和高速缓存一致性问题。Tanenbaum[1990]描述了硬件I/O的低层设计，Sargent和Shoemaker[1995]提供了针对低层PC硬件与软件的程序员指南。IBM[1983]给出了IBM PC设备I/O的地址图。1994年3月IEEE Computer专门讨论了高级I/O硬件和软件。Rago[1993]提供了一个关于流的很好的讨论。

第十四章　大容量存储器结构

文件系统从逻辑上来看可分为三个部分。第十一章讨论了用户和程序员所使用的文件系统接口。第十二章描述了操作系统实现该接口所使用的数据结构和算法。本章将讨论文件系统的最低层：次级和第三级存储结构。首先讨论为改善性能而调度磁盘 I/O 顺序的磁盘调度算法。接着，讨论磁盘格式化以及启动块、坏块和交换空间的管理。本章研究次级存储结构，包括磁盘的可靠性和稳定存储的实现。最后，简单描述第三级存储设备及操作系统使用第三级存储所引起的问题。

14.1　磁盘结构

磁盘为现代计算机系统提供了大容量的次级存储。磁带曾经用做次级存储媒介，但是由于其访问时间比磁盘要长，所以不再用做次级存储。现在，磁带主要用做备份、存储很少使用的信息、在计算机之间传输信息和存储磁盘系统所不能存储的海量数据。磁带将在 14.8 节中描述。

现代磁盘驱动器可以看做一个一维的**逻辑块**的数组，逻辑块是最小的传输单位。逻辑块的大小通常为 512 B，虽然有的磁盘可以通过**低级格式化**来选择不同逻辑块大小如 1 024 B。该选项参见 14.3.1 小节。

一维逻辑块数组按顺序映射到磁盘的扇区。扇区 0 是最外面柱面的第一个磁道第一个扇区。该映射是先按磁道内扇区顺序，再按柱面内磁道顺序，再按从外到内的柱面顺序来排序的。

通过映射，至少从理论上能将逻辑块号转换为由磁盘内的柱面号、柱面内的磁道号、磁道内的扇区号所组成的老式磁盘地址。事实上，执行如下转换并不容易，这有两个理由。第一，绝大多数磁盘都有一些缺陷扇区，因此映射必须用磁盘上的其他空闲扇区来替代这些缺陷扇区。第二，对于有些磁盘，每个磁道的扇区数并不是常量。对使用**常量线性速度**(constant linear velocity)的媒介，每个磁道的位密度是均匀的。磁道离磁盘中心越远，其长度越长，因而也能容纳更多的扇区。从外到内时，每个磁道的扇区数也会减少。外部磁道的扇区数通常比内部磁道的扇区数多 40%。随着磁头由外移到内，驱动器会增加速度以保持磁头下的数据率恒定。这种方法用于 CD-ROM 和 DVD-ROM 驱动器。另外，磁盘转动速度可以保持不变，但是由内磁道到外磁道的位密度不断降低以保持数据率不变。这种方法

称为**恒定圆角速度**(constant angular velocity,CAV)。

随着磁盘技术的不断改善,每个磁道的扇区数不断增加。磁盘外部常常有数百个扇区。类似地,每个磁盘的柱面数也在不断增加;大磁盘有数万个柱面。

14.2 磁盘调度

操作系统的任务之一就是有效地使用硬件。对磁盘驱动器,满足这一要求意味着要有较快的访问速度和较宽的磁盘带宽。访问时间包括两个主要部分(参见 2.3.2 小节)。**寻道时间**是磁臂将磁头移动到包含目标扇区的柱面的时间。**旋转延迟**是磁盘需要将目标扇区转动到磁头下的时间。**磁盘带宽**是所传递的总的字节数除以从服务请求开始到最后传递结束时的总时间。可以通过使用好的访问顺序来调度磁盘 I/O 请求,提高访问速度和带宽。

如第二章所述,每当一个进程需要对磁盘进行 I/O 操作,它就向操作系统发出一个系统调用。该调用请求指定了一些信息:

- 操作是输入还是输出
- 所传递的磁盘地址是什么
- 所传递的内存地址是什么
- 所传递的字节数是多少

如果所需的磁盘驱动器和控制器为空闲,那么该请求会马上处理。如果磁盘驱动器或控制器为忙,那么任何新的服务请求都会加到该磁盘驱动器的待处理请求队列上。对于有多个进程的多道程序设计系统,磁盘队列可能有多个待处理请求。因此,当完成一个请求时,操作系统可以选择处理哪个待处理的请求。

14.2.1 FCFS 调度

最简单的磁盘调度形式当然是先来先服务算法(FCFS)。这种算法本身比较公平,但是它通常不提供最快的服务。例如,有一个磁盘队列,其 I/O 对各个柱面上块的请求顺序如下:

98,183,37,122,14,124,65,67

如果磁头开始位于 53,那么它将从 53 移到 98,接着再到 183,37,122,14,124,65,最后到 67,总的磁头移动为 640 柱面。图 14.1 显示了这种调度。

从 122 到 14 再到 124 的大摆动显示了这种调度的问题。如果对柱面 37 和 14

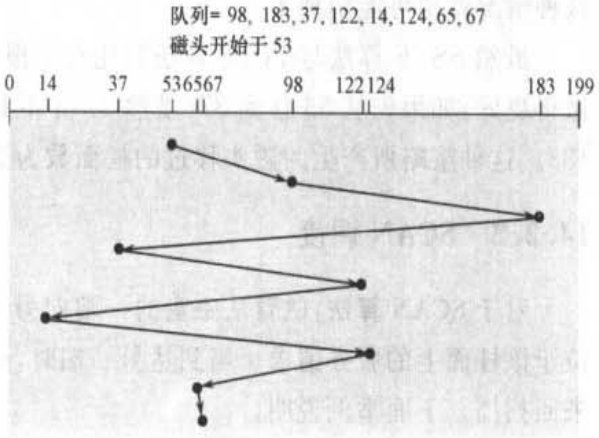


图 14.1 FCFS 磁盘调度

的请求一起处理，不管是在 122 和 124 之前或之后，总的磁头移动会大大地减少，且性能也会因此得以改善。

14.2.2 SSTF 调度

在将磁头移到远处以处理其他请求之前，先处理靠近当前磁头位置的请求可能较为合理。这个假设是**最短寻道时间优先算法**(shortest-seek-time-first，SSTF)的基础。SSTF 算法从当前磁头位置选择最短寻道时间的请求。由于寻道时间随着磁头所经过的柱面数而增加，SSTF 选择与当前磁头位置最近的待处理请求。

对于请求队列的例子，与开始磁头位置(53)最近的请求是位于柱面 65。当位于柱面 65，下个最近请求位于柱面 67。从柱面 67，由于柱面 37 比 98 还要近，所以下次处理 37。如此继续进行，会处理位于柱面 14，接着 98、122、124，最后 183 上的请求(图 14.2)。这种调度算法所产生的磁头移动为 236 柱面，约为 FCFS 调度算法所产生的磁头移动数量的三分之一强一点。这种算法大大提高了性能。

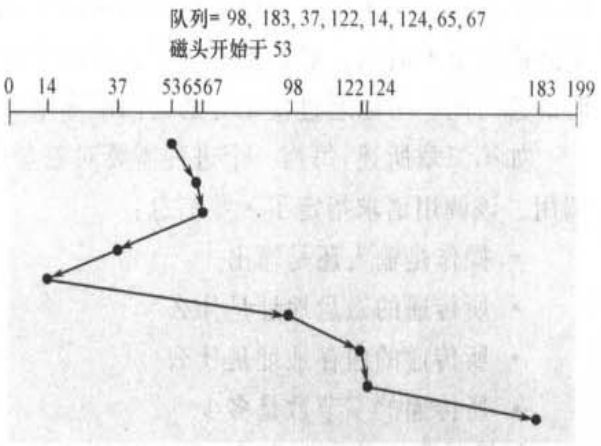


图 14.2 SSTF 磁盘调度

SSTF 调度基本上是一种最短作业优先(SJF)调度，与 SJF 调度一样，它可能会导致一些请求得不到服务。记住请求可能会随时到达。假如一个队列中有两个请求，分别为对柱面 14 和 186。当处理来自 14 的请求时，另一个靠近 14 的请求来了。这个新的请求会在下次处理，这样位于 186 上的请求要等待。当处理该请求时，另一个靠近 14 的请求可能会来。从理论上来说，相近一些的请求会连续不断地到达，这样位于 186 上的请求可能永远得不到服务。如果待处理请求队列比较长，那么这种情况就很可能出现了。

虽然 SSTF 算法与 FCFS 算法相比有了很大改善，但并不是最佳。对于这种例子，可以做得更好：如果先从 53 移到 37(虽然 37 并不是最近)，再到 14，再到 65、67、98、122、124 和 183。这种策略所产生的磁头移过的柱面数为 208。

14.2.3 SCAN 调度

对于 **SCAN 算法**，磁臂从磁盘的一端向另一端移动，同时当磁头移过每个柱面时，处理位于该柱面上的服务请求。当到达另一端时，磁头改变移动方向，处理继续。磁头在磁盘上来回扫描。下面举例说明。

在应用 SCAN 算法来调度位于柱面 98、183、37、122、14、124、65 和 67 上的请求之

前，不但需要知道磁头的当前位置，而且还需要知道磁头的移动方向。如果磁头朝 0 方向移动，那么磁头会先处理服务 37，再处理 14。在柱面 0 时，磁头会掉转方向，朝磁盘的另一端移动，并处理位于柱面 65、67、98、122、124 和 183 上的请求(图 14.3)。如果一个请求刚好在磁头之前加入到队列，那么它将几乎马上得到处理；如果一个请求刚好在磁头之后加入到队列，那么它必须等待磁头到达磁盘的另一端，掉转方向，并返回。

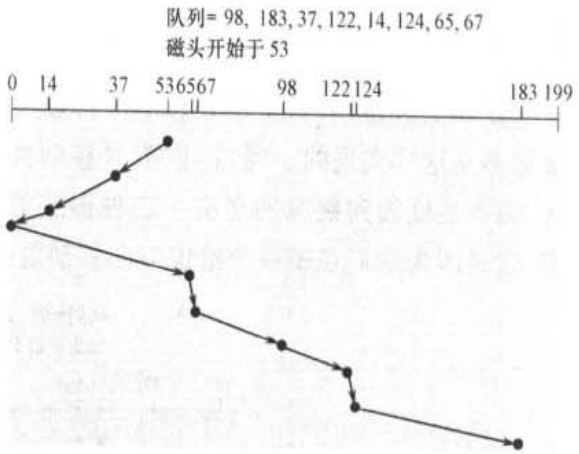


图 14.3 SCAN 磁盘调度

SCAN 算法有时称为**电梯算法**，这是因为磁头像大楼中的电梯一样，先处理所有向上的请求，再反过来处理另一方向的服务请求。

假设磁盘服务请求均匀地分布在各个柱面上，下面来研究当磁头移到磁盘一端并掉转方向时请求的分布情况。这时，紧靠磁头之前的请求只有少数，因为这些柱面上的请求刚刚处理过。而在磁盘的另一端的请求密度却最大。这些请求的等待时间最长，那么为什么不首先去那里处理呢？这就是下一算法的思想。

14.2.4 C-SCAN 调度

C-SCAN 调度是 SCAN 调度的变种，主要提供一个更为均匀的等待时间。与 SCAN 一样，C-SCAN 将磁头从磁盘一端移到磁盘的另一端，随着移动而不断地处理请求。不过，当磁头移到另一端时，它会马上返回到磁盘开始，返回时并不处理请求(参见图 14.4)。C-SCAN 调度算法基本上将柱面当做一个环链，以将最后柱面和第一柱面相连。

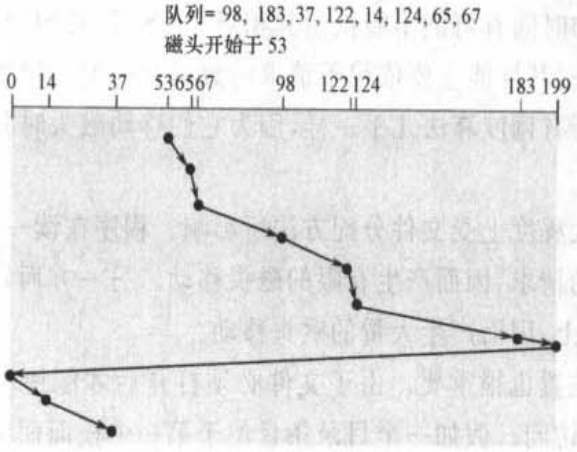


图 14.4 C-SCAN 磁盘调度

14.2.5 LOOK 调度

正如上所述，SCAN 和 C-SCAN 将磁头在整个磁盘宽度内进行移动。事实上，这两个算法都不是这样实现的。通常，磁头只移动到一个方向上最远的请求为止。接着，它马上回头，而不是继续到磁盘的尽头。这种形式的 SCAN 和 C-SCAN 称为 **LOOK** 和 **C-LOOK 调度**，这是因为它们在朝一个给定方向移动前会 *look* 是否有请求(图 14.5)。

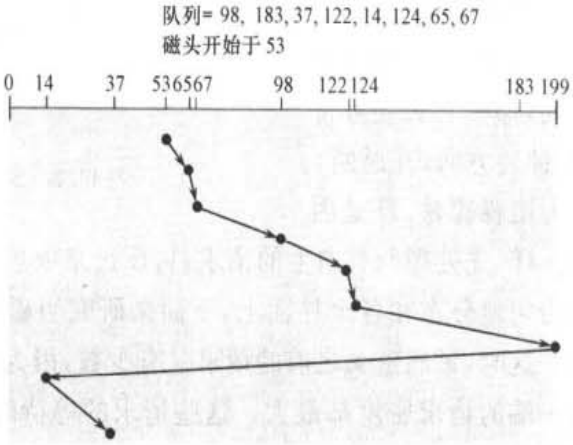


图 14.5 C-LOOK 磁盘调度

14.2.6 磁盘调度算法的选择

现在有如此之多的磁盘调度算法，如何选择最佳的呢？SSTF 较为普遍且很有吸引力，因为它比 FCFS 的性能要好。SCAN 和 C-SCAN 对于磁盘负荷较大的系统会执行得更好，这是因为它不可能产生饿死问题。对于一个特定请求队列，能定义一个最佳的执行顺序，但是查找最佳调度所需的时间有可能不能抵消其相对于 SSTF 或 SCAN 的节省。

对于任何调度算法，其性能主要依赖于请求的数量和类型。例如，假设队列通常只有一个待处理请求。那么所有调度算法几乎一样，因为它们移动磁头时只有一种选择：它们几乎都与 FCFS 调度一样。

磁盘服务请求很大程度上受文件分配方法所影响。程序在读一个连续分配文件时会产生数个在磁盘上相近的请求，因而产生有限的磁头移动。另一方面，链接或索引文件可能会有许多块，分散在磁盘上，因而产生大量的磁头移动。

目录和索引块的位置也很重要。由于文件必须打开后才能使用，打开文件要求搜索目录结构，目录会被经常访问。假如一个目录条目位于第一个柱面而文件数据位于最后柱面。对于这种情况，磁头必须移过整个磁盘宽度。如果目录条目位于中央柱面，那么磁头只需要移过不到一半的磁盘。在内存中缓存目录和索引块有助于降低磁头移动，尤其是对于读请求。

由于这些复杂因素，磁盘调度算法应作为一个操作系统的独立模块，这样如果有必要，

它可以替换成另一个不同的算法。SSTF 或 LOOK 是比较合理的缺省算法。

这里所描述的调度算法只考虑了寻道距离。对于现代磁盘，旋转等待几乎与平均寻道时间一样。但是操作系统比较难以调度来改善旋转等待，这是因为现代磁盘并不透露逻辑块的物理位置。磁盘制造商通过在磁盘驱动器的控制器中加上磁盘调度算法来缓解这个问题。如果操作系统向控制器发送一批请求，那么控制器可以对这些请求进行排队和调度，以改善寻道时间和旋转等待。如果只需要考虑 I/O 性能，那么操作系统会很乐意将磁盘调度的责任交给磁盘硬件。不过，事实上操作系统对请求服务顺序还有其他限制。例如，按需分页比 I/O 的优先级高。如果缓存将要用完空闲页那么写就比读更重要。而且，可能需要保存写的顺序以使文件系统更加灵活，从而免受经常崩溃之苦。假如操作系统分配了一页给一个文件，应用程序已将数据写入到页中但操作系统还没有在磁盘上更新修改 inode 和空闲空间列表。为了处理这些要求，操作系统需要选择自己的磁盘调度算法，将请求按批次（或一个一个地，对于有的 I/O 类型）交给磁盘控制器。

14.3 磁盘管理

操作系统在磁盘管理方面还有其他的内容。这里讨论磁盘初始化、从磁盘引导、坏块恢复。

14.3.1 磁盘格式化

一个新的磁盘是一个空白板：它只是一些含有磁性记录材料的盘子。在磁盘能存储数据之前，它必须分成扇区以便磁盘控制器能读和写。这个过程称为低级格式化（或**物理格式化**）。低级格式化为磁盘的每个扇区采用特别的数据结构。每个扇区的数据结构通常由头、数据区域（通常为 512 B 大小）和尾部组成。头部和尾部包含了一些磁盘控制器所使用的信息，如扇区号码和**纠错代码**（error-correcting code，ECC）。当控制器在正常 I/O 时写入一个扇区的数据，ECC 会用一个根据磁盘数据计算出来的值进行更新。当读入一个扇区时，ECC 值会重新计算，并与原来存储的值相比较。如果这两个值不一样，那么这可能表示扇区的数据区已损坏和磁盘扇区变坏（14.3.3 小节）。ECC 是*纠错*代码，这是因为它有足够多的信息，如果只有少数几个数据损坏，控制器能利用 ECC 计算出哪些数据已改变并计算出它们的正确值。控制器在读或写磁盘时会自动处理 ECC。

绝大多数硬盘在工厂时作为制造过程的一部分就已低级格式化。这一格式化使得制造商能测试磁盘和初始化从逻辑块码到磁盘上无损扇区的映射。对许多硬盘，当通知磁盘控制器低级格式化磁盘时，也能选择在头部和尾部之间留下多长的数据区。通常有几种选择，如 256 B、512 B 和 1 024 B 等。用一个较大扇区去低级格式化磁盘意味着每个磁道上的扇区数会比较少，每个磁道上的头部和尾部信息也会比较少，因此增加了用户数据的可用空间。有的操作系统只能处理 512 B 大小的扇区。

为了使用磁盘存储文件，操作系统还需要将自己的数据结构记录在磁盘上。这分为两步。第一步是将磁盘分为由一个或多个柱面组成的**分区**。操作系统可以将每个分区作为一个独立的磁盘。例如，一个分区可以用来存储操作系统的可执行代码，而其他分区用来存储用户数据。在分区之后，第二步是**逻辑格式化**（创建文件系统）。在这一步，操作系统将初始的文件系统数据结构存储到磁盘上。这些数据结构包括空闲和已分配的空间以及一个初始为空的目录。

有的操作系统允许特别程序将磁盘分区作为一个大逻辑块的顺序数组，而没有任何文件系统数据结构。该数组有时称为生磁盘（raw disk），对该数组的 I/O 称为生I/O（raw I/O）。例如，有的数据库系统比较喜欢生 I/O，因为它能控制每条数据库记录所存储的精确磁盘位置。生 I/O 绕过了所有文件系统服务，如缓冲、文件锁、提前获取、空间分配、文件名和目录。通过在生磁盘分区上实现自己的特定目录存储服务，某些应用程序的效率可能会更高，但是绝大多数应用程序在使用普通文件服务时则会执行得更好。

14.3.2 引导块

为了让计算机开始运行，如当打开电源时或重启时，它需要运行一个初始化程序。该初始化自举程序应该很简单。它初始化系统的各个方面，从 CPU 寄存器到设备控制器和内存，接着启动操作系统。为此，自举程序应找到磁盘上的操作系统内核，装入内存，并转到起始地址，从而开始操作系统的执行。

对绝大多数计算机，自举程序保存在**只读存储器**（ROM）中。这一位置较为方便，由于 ROM 不需要初始化且位于固定位置，这便于处理器在打开电源或重启时开始执行。而且，由于 ROM 是只读的，所以不会受计算机病毒的影响。问题是改变这种自举代码需要改变 ROM 硬件芯片。因此，绝大多数系统只在启动 ROM 中保留一个很小的自举装入程序，其作用是进一步从磁盘上调入更为完整的自举程序。这一更为完整的自举程序可以容易地进行修改：新版本可写到磁盘上。更为完整的自举程序保存在磁盘的启动块上，启动块位于磁盘的固定位置。拥有启动分区的磁盘称为**启动磁盘**，或**系统磁盘**。

图 14.6 MS-DOS 的磁盘布局

启动 ROM 中的代码引导磁盘控制器将启动块读入到内存（这时尚未装入设备驱动程序），并开始执行代码。完整自举程序比启动 ROM 内的自举程序更加复杂；它能从磁盘非固定位置中装入整个操作系统，并开始运行。即便如此，完整自举程序仍可能很小。例如，MS-DOS 的启动程序只使用一个 512 B 大小的块（图 14.6）。

14.3.3 坏块

由于磁盘有移动部件并且容错能力小(磁头在磁盘表面上飞行),所以容易出问题。有时问题严重,必须替换磁盘,其内容就要从备份媒介上恢复到新磁盘上。但更经常遇到的问题是,一个或多个扇区坏掉。绝大多数磁盘从生产厂家里出来时就有**坏扇区**。根据所使用的磁盘和控制器,对这些块有多种处理方式。

对于简单磁盘如使用 IDE 控制器的磁盘,坏扇区可手工处理。例如,MS-DOS format 命令执行逻辑格式化,其中,它将扫描磁盘以查找坏扇区。如果 format 找到坏扇区,那么它就在相应 FAT 条目中写上特殊值以通知分配程序不要使用该块。如果在正常使用中块变坏,那么就必须人工地运行一个特殊程序如 chkdsk 来搜索坏扇区,并像前面一样将它们锁在一边。坏扇区的数据通常会丢失。

更为复杂的磁盘,如用于高端计算机、绝大多数工作站和服务器上的 SCSI 磁盘,对坏块的处理就更加聪明。其控制器维护一个磁盘坏块链表。该链表在出厂前进行低级格式化时就初始化了,并在磁盘的整个使用过程中不断更新。低级格式化将一些块放在一边作为备用,对此操作系统并不知道。控制器可以用备用块来逻辑地替代坏块。这种方案称为**扇区备用**或**转寄**。

一个典型的坏扇区事务处理可能如下:

- 操作系统试图访问逻辑块 87。
- 控制器计算 ECC 的值,发现该块是坏的。它将此结果通知操作系统。
- 下次操作系统重启时,可以运行一个特殊程序以告诉 SCSI 控制器用备用块替代坏块。
- 之后,每当系统试图访问逻辑块 87 时,这一请求就转换成控制器所替代的扇区的地址。

这种控制器所引起的间接性可能会使操作系统的磁盘调度算法无效。为此,绝大多数磁盘在格式化时为每个柱面都留了少量的备用块,还保留了一个备用柱面。当坏块需要重新映射时,控制器就尽可能使用同一柱面的备用扇区。

作为扇区备用的另一方案,有的控制器采用**扇区滑动**来替换坏扇区。这里有一个例子:假定逻辑块 17 变坏,而第一个可用的备用块在扇区 202 之后。那么,扇区滑动就将所有从 17 到 202 的扇区向下滑动一个扇区。即,扇区 202 复制到备用扇区,201 到 202,200 到 201 等等直到扇区 18 复制到扇区 19。这样滑动扇区使得扇区 18 为空,这样可将扇区 17 映射到其中。

坏块替代通常并不是完全自动的进程,这是因为坏区中的数据通常会丢失。因此,当使用了坏块的文件必须修补时(如从备份磁带中恢复),通常需要人工干预。

14.4 交换空间管理

交换空间管理是操作系统的另一底层任务。虚拟内存使用磁盘空间作为内存的扩充。

由于磁盘访问比内存访问要慢很多，所以使用交换空间会严重影响系统性能。交换空间设计和实现的主要目的是为虚拟内存提供最佳吞吐量。这里讨论如何使用交换空间、交换在磁盘上的什么位置以及交换空间该如何管理。

14.4.1 交换空间的使用

不同操作系统根据所实现的内存管理算法，可按不同方式来使用交换空间。例如，实现交换的系统可以将交换空间用于保存整个进程映像，包括代码段和数据段。换页系统也可能只用交换空间来存储换出内存的页。系统所需交换空间的量因此会受以下因素影响：物理内存的多少、所支持虚拟内存的多少、内存使用方式等。它可以是数兆到数吉的磁盘空间。

有的操作系统（如 UNIX）允许使用多个交换空间。这些交换空间通常位于不同磁盘上，这样因换页和交换所引起的 I/O 系统的负荷可分散在各个系统 I/O 设备上。

注意对交换空间数量的高估要比低估更为安全。这是因为如果系统使用完了交换空间，那么可能会中断进程或使整个系统死机。高估只是浪费了一些空间（本可用于存储文件），但并没有造成什么损害。

14.4.2 交换空间位置

交换空间可有两个位置：交换空间在普通文件系统上加以创建，或者是在一个独立的磁盘上进行分区。如果交换空间是文件系统内的一个简单大文件，那么普通文件系统程序就可用来创建它、命名它并为它分配空间。这种方式虽然实现简单但是效率较低。遍历目录结构和磁盘分配数据结构需要时间和（可能）过多磁盘访问。外部碎片可能会通过在读写进程镜像时强制多次寻道，从而大大地增加了交换时间。通过将块位置信息缓存在物理内存中以及采用特殊工具为交换文件分配物理上连续块等技术，可以改善性能；但是遍历文件系统数据结构的开销仍然存在。

另外一种方法是，交换空间可以创建在独立的磁盘分区上。这里不需要文件系统和目录结构。而只需要一个独立交换空间存储管理器以分配和释放块。这种管理器可使用适当算法来优化速度，而不是优化存储效率。内部碎片可能会增加，但还是可以接受的，这是因为交换空间内的数据的存储时间通常要比文件系统的文件的存储时间短很多，且交换空间的使用更为经常。这种方法在磁盘分区时创建一定量的交换空间。增加更多交换空间可能需要重新进行磁盘分区（涉及移动或删除和利用备份以恢复文件系统），或在其他地方增加另外的交换空间。

有的操作系统较为灵活，可以使用分区空间和文件系统空间进行交换。Solaris 2 就是这样的操作系统。由于策略和实现的分开，系统管理员可决定使用何种类型。权衡取决于文件系统分配和管理的方便以及生分区交换的性能。

14.4.3 交换空间管理:例子

为了说明交换空间管理所使用的方法,下面研究 UNIX 交换和调页的发展。正如附录 A 所详细描述的,UNIX 开始只实现了交换,以便在连续磁盘区域和内存之间进行整个进程的复制。后来随着调页硬件的出现,UNIX 发展成混合使用交换和换页。

对于 4.3 BSD,当创建进程时,也为之分配交换空间。这一空间足够大,能够容纳进程中的**代码页**或**代码段**和**数据段**。预先分配所有所需空间可防止进程在运行时用完交换空间。当进程开始时,其代码以页形式从文件系统中调入。这些页在需要时可写出到交换空间,并从交换空间中读回,这样只需要为每个代码页读文件系统一次即可。数据段的页从文件系统中读入或创建(如果没有初始化),并按需要写入到交换空间和从交换空间中读回。一种优化(例如,当两个用户使用同一编辑器时)是具有相同代码页的进程可共享这些页(包括在物理内存和在交换空间中)。

内核对每个进程采用两个**交换表**来跟踪交换空间使用。代码段是固定大小的,因此其交换空间是按 512 KB 区域来分配的,除了最后一块是按 1 KB 增量来容纳剩余页(图 14.7)。

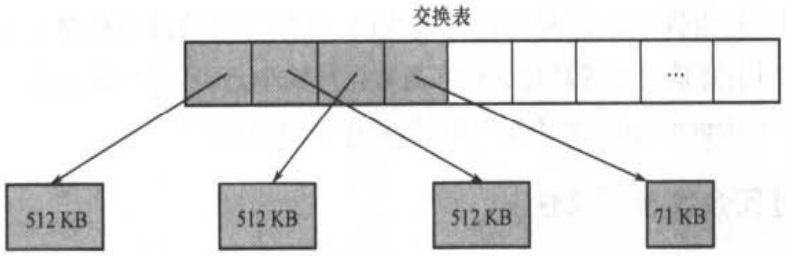


图 14.7 4.3BSD 系统的代码段交换表

数据段交换表较为复杂,这是因为数据段会随时间而增长。表是固定大小的,但可包括可变大小块的地址。对给定索引 i,交换条目 i 所指块的大小为 $2^i \times 16$ KB,最大为 2 MB。这种数据结构如图 14.8 所示。(块大小最小值和最大值是可变的,可在系统重启时加以调整。)当一个进程试图增长其数据段以超出其在交换空间中所分配的最后一块时,操作系统会分配另一块,其大小是前一块的两倍。这种方法使得小进程只使用小块。它也降低了碎片。可以很快找到大进程的块,交换空间也较小。

对于 Solaris 1(SunOS 4),设计人员对标准 UNIX 方法做了修改,以改善效率和反映技术变化。当一个进程执行时,代码段从文件系统中调入,在内存中加以访问,如果需要换出时就丢弃。从文件系统中再次读入一页要比将它保存在交换空间中再从中读入更为高效。

Solaris 2 又做了更多修改。最大修改是 Solaris 2 只有在一页被强制换出物理内存时(而不是在首次创建虚拟内存页时)才分配交换空间。这一修改提高了现代计算机的性能,因为它们比旧的系统有更多物理内存,所以换页少。

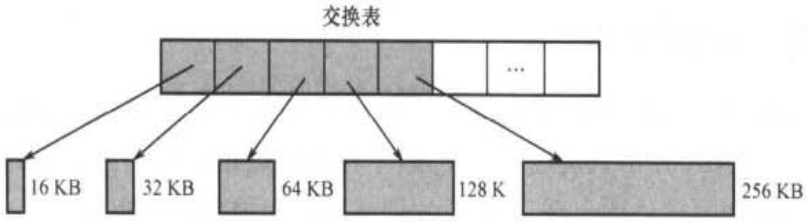


图 14.8 4.3BSD 系统的数据段交换表

14.5 RAID 结 构

随着磁盘驱动器不断地变得更小、更便宜，如今在一台计算机系统上装载大量磁盘从经济上来说已经可行了。

一个系统拥有了大量磁盘，它就有机会改善数据读或写的速度（因为磁盘操作可并行进行）。而且，这种设置也使得系统有机会改善数据存储的可靠性，因为可在多个磁盘上存储冗余信息。因此，一个磁盘损坏并不会导致数据丢失。这里的多种磁盘组织技术，通常统称为 RAID 技术，用于提高性能和可靠性。

过去，RAID 是由许多小的便宜磁盘组成的，可作为大的昂贵磁盘的有效替代品。现在，RAID 的使用主要是因为其高可靠性和高数据传输率，而不是经济原因。因此，RAID 中的 I 表示“独立（independent）”而不是“便宜（inexpensive）”。

14.5.1 通过冗余改善可靠性

首先研究可靠性。在 N 个磁盘中有一个磁盘出错的概率要比某个特定磁盘出错的概率高得多。假如单个磁盘的**平均故障出现时间**为 100 000 小时。那么在 100 个磁盘中有一个磁盘变坏的平均时间则为 100 000/100＝1 000 小时或 41.66 天，这并不长！如果只存储数据的一个拷贝，那么一个磁盘出错会导致大量数据损坏，这样高的数据损坏率是难以接受的。

可靠性问题的解决方法是引入**冗余**。我们存储额外信息，这是平常所不需要的，但在磁盘出错时可以用来重新修补损坏信息。因此，即使磁盘损坏，数据也不会损坏。

最为简单（但最为昂贵）的引入冗余的方法是复制每个磁盘。这种技术称为**镜像**（或**影子**）。因此每个逻辑磁盘由两个物理磁盘组成，每次写都要在两个磁盘上进行。如果一个磁盘损坏，那么可从另一个磁盘中恢复。只有在第一个损坏磁盘没有替换之前而第二个磁盘又出错，那么数据才会丢失。

镜像磁盘出错（这里指数据丢失）的平均时间取决于两个因素：单个磁盘出错的平均时间，以及**修补平均时间**（用于替换损坏磁盘并恢复其中数据）。假定两个磁盘出错是相对**独立**的；即，一个磁盘出错与另一磁盘出错之间没有关联。因此，如果一个磁盘出错的平均时间为 100 000 小时，且修补平均时间为 10 小时，那么镜像磁盘系统的**数据丢失的平均时间**为

$100\ 000^2/(2\times10) = 500\times10^5$ 小时或 57 000 年。

需要注意磁盘出错独立性假设并不有效。电源掉电和自然灾害如地震、火灾、水灾都可能导致两个磁盘同时损坏。而且，成批生产磁盘的制造缺陷会引起相关出错。随着磁盘老化，出错概率增加，也增加了在替代第一个磁盘时第二个磁盘出错的概率。当然，尽管存在这些因素，镜像磁盘系统仍然能够比单个磁盘系统提供更高的可靠性。

电源掉电是个特别值得关注的问题，因为它们比自然灾害更为经常。然而，由于磁盘镜像，如果对两个磁盘都进行写同样块，且在数据完全写完之前电源掉电，那么这两块可能处于不一致。这个问题的解决方法是先写一个拷贝，再写下一个，这样两个之一总是一致的。当在电源掉电之后重启时，需要采取额外动作，以便从不完全写状态中恢复。

14.5.2 通过并行处理改善性能

现在考虑多磁盘并行访问的益处。对于磁盘镜像，读请求处理的速度可以加倍，这是因为读请求可以发送给任一磁盘（只要两个磁盘都能工作，情况几乎总是这样）。每个读的传输率与单个磁盘系统一样，但是单位时间内读数量加倍了。

对于多个磁盘，通过在多个磁盘上分散数据，能够改善传输率。最简单的形式是，**数据分散**是在多个磁盘上分散每个字节的各个位；这种分散称为**位级分散**。例如，如果有 8 个磁盘，可将每个字节的位 i 写到磁盘 i 上。这 8 个磁盘可作为单个磁盘使用，其扇区为正常扇区的 8 倍，更为重要的是它具有 8 倍的传输率。对于这种结构，每个磁盘都参与每次访问（读或写），这样每秒所能处理的访问数与单个磁盘一样，但每次访问可在同样时间内与单个磁盘系统读 8 倍的数据相同。

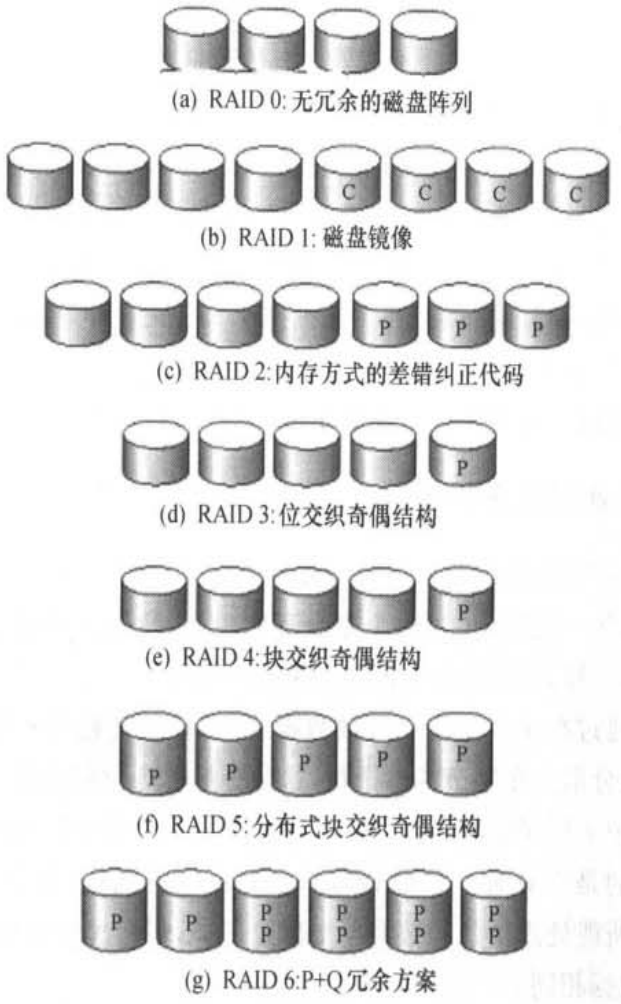
位级分散可扩展到其他数量的磁盘，只要该数量为 8 的倍数或能除以 8。例如，如果有 4 个磁盘，每个字节的位 i 和位 $4+i$ 可存在磁盘 i 上。另外，分散不必在字节的位级上进行：例如，对于**块级分散**，一个文件的块可分散在多个磁盘上；对于 n 个磁盘，一个文件的块 i 可存在磁盘 $(i \bmod n)+1$。其他分散级别，如扇区字节和块的扇区也是可能的。

总之，磁盘系统并行访问有两个主要目的：

1. 通过负荷平衡，增加了多个小访问（即页访问）的吞吐量。
2. 降低大访问的响应时间。

14.5.3 RAID 级别

镜像提供高可靠性，但很昂贵。分散提供了高数据传输率，但并未改善可靠性。通过磁盘分散和“奇偶”位（下面将要讨论）可以提供多种方案以在低代价下提供冗余。这些方案有不同的性价折中，可分成不同级别，称为 **RAID 级别**。这里讨论各种级别；图 14.9 描述了这些结构（图中，P 表示差错纠正位，C 表示数据的第二拷贝）。在图中所描述的各种情况中，4 个磁盘用于存储数据，其他磁盘用于存储冗余信息以便从差错中恢复。



(a) RAID 0:无冗余的磁盘阵列

(b) RAID 1:磁盘镜像

(c) RAID 2:内存方式的差错纠正代码

(d) RAID 3:位交织奇偶结构

(e) RAID 4:块交织奇偶结构

(f) RAID 5:分布式块交织奇偶结构

(g) RAID 6:P+Q冗余方案

图 14.9 RAID 的级别

• **RAID 级别 0**:RAID 级别 0 指按块级别分散的磁盘阵列,但没有冗余(如镜像或奇偶位)。图 14.9(a)显示了大小为 4 的磁盘阵列。

• **RAID 级别 1**:RAID 级别 1 指磁盘镜像。图 14.9(b)所示为一个镜像组织,可容纳 4 个磁盘的数据。

• **RAID 级别 2**:RAID 级别 2 也称为**内存方式的差错纠正代码结构**。内存系统一直实现了基于奇偶位的错误检测。内存系统的每个字节都有一个相关奇偶位,以记录字节中 1 位的个数是偶数(parity=0)或是奇数(parity=1)。如果字节的 1 个位损坏(或是 1 变成 0 或是 0 变成 1),那么字节的奇偶也改变,因此与所存储的奇偶位就不匹配。类似地,如果所存储的奇偶位损坏了,那么它就与所计算的奇偶位不匹配。因此,单个位差错可为内存系统所检测。差错纠正方案存储两个或多个额外位,当单个位出错时可用来重新构造数据。ECC 的思想可直接用于将字节分散在磁盘上的磁盘阵列。例如,每个字节的第 1 位可存在磁盘 1 上,第 2 位可存在磁盘 2 上,如此进行直到第 8 位可存在磁盘 8 上,而差错纠正代码存在其他磁盘上。图 14.9(c)显示了这种方案,其中标为 P 的磁盘存储了差错纠正位。如

果一个磁盘出错，那么可从其他磁盘中读取字节的其他位和相关差错纠正位，以重新构造损坏数据。图 14.9(c)显示了大小为 4 的阵列；注意对于 4 个磁盘的数据，RAID 级别 2 只用了 3 个额外磁盘，而 RAID 级别 1 则需要用 4 个额外磁盘。

• **RAID 级别 3**：RAID 级别 3 或基于**位交织奇偶结构**对级别 2 做了改进：与内存系统不同，磁盘控制器能检测到一个扇区是否正确读取，这样单个奇偶位就可用于差错检测和差错纠正。这种方案如下。如果一个扇区损坏，那么知道是哪个扇区，通过计算其他磁盘扇区相应位的奇偶值可得出所损坏位是 1 还是 0。如果其他位的奇偶值等于存储奇偶值，那么缺少位为 0；否则，就为 1。RAID 级别 3 与级别 2 一样好，但在额外磁盘数量方面要更便宜（它只有一个额外磁盘），这样级别 2 在实际中并不使用。这种方案如图 14.9(d)所示。

RAID 级别 3 与级别 1 相比有两个优点。多个普通磁盘只需要一个奇偶磁盘，而级别 1 的每个磁盘都需要相应镜像磁盘，因此降低了额外存储。由于采用 N 路分散数据，字节的读和写分布在多个磁盘上，所以采用 N 路分散后，单个块的读或写传输速度是采用 RAID 级别 1 的 N 倍。另一方面，RAID 级别 3 的每秒 I/O 次数将较小，这是因为每个磁盘都要参与每次的 I/O 请求。RAID 3 的另一性能问题（其他奇偶检验 RAID 级别也有）是需要计算和写奇偶。与其他非奇偶检验 RAID 相比，这种额外开销会导致写更慢。为了缓解这种性能损失，计算机 RAID 存储阵列包括一个具有专用奇偶计算硬件的控制器。这将 CPU 奇偶计算转移到 RAID 阵列。这种阵列也有**非易失性随机存储器**（NVRAM），以在计算奇偶时存储块，并缓存从控制器到磁盘的数据。这些措施使得奇偶 RAID 与非奇偶 RAID 几乎一样快。事实上，缓存的奇偶 RAID 可比非缓存的非奇偶 RAID 要快。

• **RAID 级别 4**：RAID 级别 4 或**块交织奇偶结构**采用与 RAID 0 一样的块级分散，另外在一个独立磁盘上保存其他 N 个磁盘相应块的奇偶块。这种方案如图 14.9(e)所示。如果一个磁盘出错，那么奇偶块可以与其他磁盘的相应块一起用于恢复出错磁盘的块。

读块只访问一个磁盘，可以允许处理其他磁盘的请求。因此，每个访问的数据传输速度较慢，但是多个读访问可以并行处理，产生了更高的总的 I/O 速度。大读的数据传输速度高，这是因为所有磁盘可以并行读；大量数据的写操作传输速度也高，这是因为数据和奇偶可以并行写。

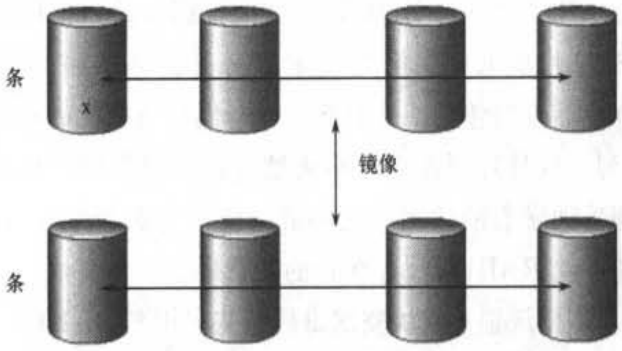
另一方面，小量的独立写不能并行进行。一块的写必须访问块所存的磁盘和奇偶磁盘，因为奇偶块必须要更新。而且，奇偶块的旧值和待写块的旧值必须读入以便计算新值的奇偶。这称为**读—改—写**。因此，单个写需要 4 个磁盘访问：两个读入旧块，两个写入新块。

• **RAID 级别 5**：RAID 级别 5 或**块交织分布式奇偶结构**。不同于级别 4，它是将数据和奇偶分布在所有 $N+1$ 块磁盘上，而不是将数据存在 N 个磁盘上而奇偶存在单个磁盘上。对于每一块，一个磁盘存储奇偶，而其他的存储数据。例如，对于 5 个磁盘的阵列，第 n 块的奇偶保存在磁盘 $(n \bmod 5)+1$ 上；其他 4 个磁盘的 n 块保存该块的真正数据。这种方案如图 14.9(f)所示，其中 P 分布在所有磁盘上。奇偶块不能保存在同一磁盘上，这是因为一个

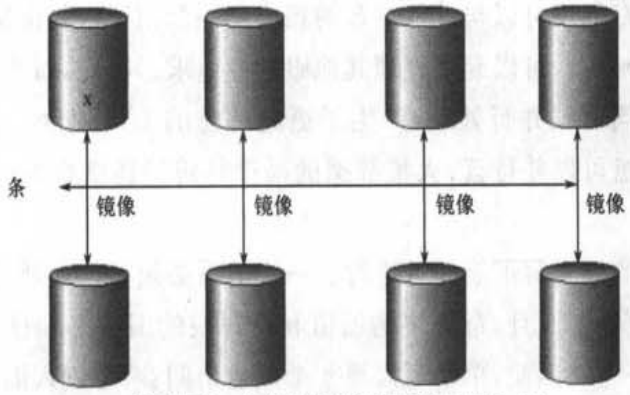
磁盘出错会导致所有数据及奇偶丢失，因而无法恢复。通过将奇偶分布在所有磁盘上，RAID 5 避免了 RAID 4 方案的对单个奇偶磁盘的过度使用。

• **RAID 级别 6**：RAID 级别 6，也称为 **P+Q 冗余方案**，与 RAID 级别 5 很类似，但是保存了额外冗余信息以防止多个磁盘出错。不是使用奇偶校验，而是使用了差错纠正码如 **Reed-Solomon 码**。在如图 14.9(g)所示方案中，每 4 个位的数据使用了 2 个位的冗余数据，而不是像级别 5 那样的一个奇偶位，这样系统可以忍受两个磁盘出错。

• **RAID 级别 0+1**：RAID 级别 0+1 指 RAID 级别 0 和级别 1 的组合。RAID 0 提供了性能，而 RAID 1 提供了可靠性。通常，它比 RAID 5 有更好的性能。对于性能和可靠性都重要的环境，这更为常用。然而，这使用于存储的磁盘数量加倍，所以也更为昂贵。对于 RAID 0+1，一组磁盘被分散成条，每一条再镜像到另一条。另一当前正在商业化的 RAID 选择是 RAID 1+0，即磁盘先镜像，再分散。这种 RAID 比 RAID 0+1 有一些理论上的优点。例如，如果 RAID 0+1 中的一个磁盘出错，那么整个条就不能访问，虽然所有其他条可用。对于 RAID 1+0，如果单个磁盘不可用，但其镜像仍如其他磁盘一样可用(图 14.10)。



(a) 只有一个磁盘故障情况下的 RAID 0+1



(b) 只有一个磁盘故障情况下的 RAID 1+0

图 14.10 RAID 0+1 和 1+0

最后，要注意到对这里所描述的基本 RAID 方案有许多建议变种。因此，对于不同 RAID 级别的精确定义存在一定的混淆。

14.5.4 RAID 级别的选择

如果一个磁盘损坏，那么重建其数据的时间可能很长且随所使用的 RAID 级别有所变化。对于 RAID 级别 1，重建较为容易，因为可从另一磁盘上复制数据；对于其他级别，需要读入所有阵列中的其他磁盘以重建出错磁盘上的数据。在需要连续提供数据时，如高性能或交互式数据库系统，RAID 系统的重建性能可能是个重要因素。而且，重建性能影响平均失败时间。

RAID 级别 0 用于高性能应用但数据损失并不关键。RAID 级别 1 用于需要高可靠性和快速恢复的应用。RAID 0+1 和 1+0 用于性能和可靠性都重要的地点，例如对于小型数据库。由于 RAID 1 的高空间开销，RAID 5 通常用于存储量大的数据。级别 6 并不为许多 RAID 实现所支持，但它应该能比 RAID 5 提供更高的可靠性。

RAID 系统设计人员还必须做出其他几个决定。例如，一个组应有多少磁盘？每个奇偶位保护多少位？如果一个阵列有更多磁盘，那么数据传输率就更高，但是系统就更昂贵。如果一个奇偶位保护更多的位，那么因奇偶位所造成的额外开销就更低，但是在一个磁盘出错需要替换之前而出现另一磁盘错误的概率就会增加，这会导致数据损失。

绝大多数 RAID 实现的另一方面是使用热备份磁盘。**热备份**不用于存储数据，但配置成替换出错磁盘。例如，当一个磁盘出错时，热备份可用于重新构造镜像磁盘。这样，RAID 级别就自动重新建立，而无需等待替换磁盘。分配更多热备份允许更多磁盘出错而无需人工干预。

14.5.5 扩展

RAID 概念可扩展到其他存储设备，包括磁带阵列甚至在无线系统上的数据广播。当用于磁带阵列时，即使磁带阵列中的一个磁带出现损坏，仍然可以利用 RAID 结构恢复数据。当用于数据广播时，一块数据可分成小单元，并和奇偶单元一起广播；如果一个单元因某种原因不能收到，那么它可通过其他单元收到。通常，磁带驱动机器人包括多个磁带驱动器，可将数据分散在所有驱动器上以增加吞吐量和降低备份时间。

14.6 磁盘附属

计算机访问磁盘存储有两种方式。一种方式是通过 I/O 端口（或**主机附属存储**（host-attached storage)），小系统常采用这种方式。另一方式是通过分布式文件系统的远程主机；这称为**网络附属存储**（network-attached storage)。

14.6.1 主机附属存储

主机附属存储是通过本地 I/O 端口访问的存储。这些端口可由多种技术提供。典型的

桌面个人计算机使用I/O总线结构，如IDE或ATA。这种结构允许每条I/O总线支持最多两个端口。高端工作站和服务器通常采用更为复杂的I/O结构，如SCSI或FC(fibre channel)。

SCSI是个总线结构。其物理媒介通常为带状电缆，具有大量电线(通常50或68)。SCSI协议在一根总线上可支持16个设备。通常这些设备包括主机的一个控制器卡(**SCSI引导器**)和15个存储设备(**SCSI目标**)。SCSI磁盘是个典型SCSI目标，但是协议只提供访问8个**逻辑单元**的能力。逻辑单元地址的典型使用是向RAID阵列的成员或可移动媒介库的成员(如CD组向媒介切换机制或一个驱动器，发送命令)发送命令。

FC是个高速串行结构。该结构可在光纤上或4芯铜线上运行。它有两种方式。一是大的交换结构，具有24位地址空间。这种方式可望在将来流行，是**存储区域网络**(SANs)的基础。由于大地址空间和通信的交换特性，多主机和存储设备可以附属到光纤上，允许更灵活的I/O通信。另一种是**裁定循环**(FC-AL)，可以访问126个设备(驱动器和控制器)。

很多类型的存储设备可用于主机附属存储。它们包括硬盘驱动器、RAID阵列、CD、DVD和磁带驱动器。

向主机附加存储设备发出数据传输的I/O命令是针对特定存储单元的逻辑数据块的读和写(例如总线ID、SCSI ID和目标逻辑单元)。

14.6.2 网络附属存储

网络附属存储(network-attached storage，NAS)设备是专用存储系统，通过数据网络访问(图14.11)。客户通过远程进程调用接口如UNIX系统的NFS或Windows系统的CIFS。远程进程调用(RPC)可通过IP网络(通常为向客户传输所有数据的局域网LAN)的TCP或UDP来进行。网络存储单元通常用RAID阵列加上实现RPC接口的软件来实现。将NAS作为另一存储访问协议就更为容易。例如，采用NAS的系统不采用SCSI设备驱动程序和SCSI协议来访问存储，而是使用TCP/IP上的RPC。

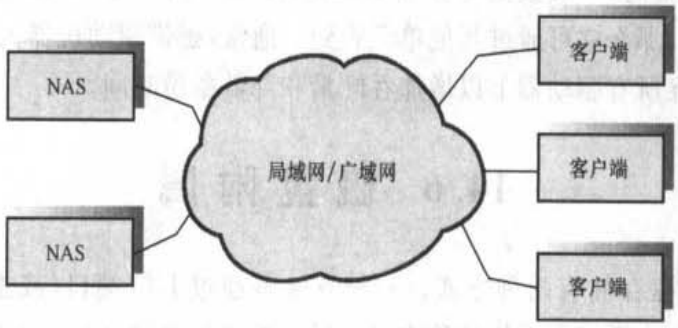


图14.11 网络附加存储

网络附加存储为LAN上的所有计算机提供了一个共享存储池的方便方法，其命名和访问与主机附加存储一样方便。然而，与直接附加存储相比，这种方法似乎效率更低，性能更差。

14.6.3 存储区域网络

网络附属存储系统的缺点之一是存储 I/O 操作需要使用数据网络的带宽，因此增加了网络通信延迟。这一问题对于大客户－服务器环境可能尤为明显，客户与服务器间的通信和存储设备与服务器间的通信互相竞争。

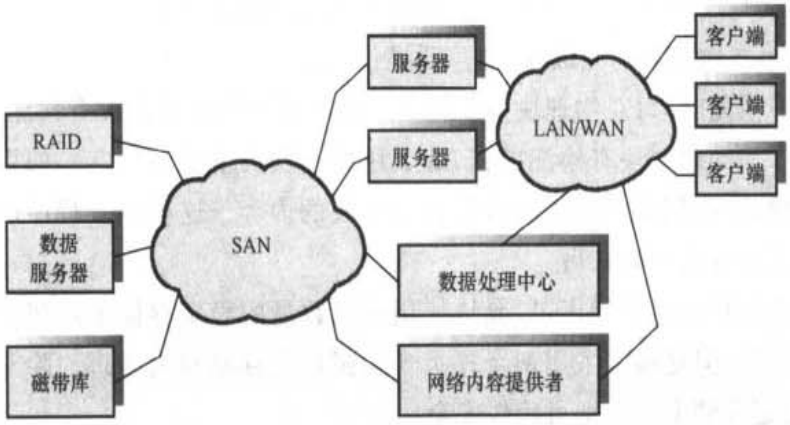


图 14.12 存储区域网络

存储区域网络(storage-area network，SAN)是服务器与存储单元之间的专门网络(采用存储协议而非网络协议)，它与连接服务器和客户机的 LAN 或 WAN 不同(图 14.12)。多个主机和多个存储阵列可以附加在同一 SAN 上，存储可以动态地附加在主机上。一个例子，如果一个主机缺少磁盘空间，那么可以配置 SAN 为该主机提供更多存储。在 2001 年，已有很多专门的 SAN 系统，但 SAN 部件尚未标准化或缺少互操作性。2001 年的绝大多数 SAN 是基于光纤循环或光纤通道交换网络。另外一种新出现的方式是通过 IP 网络架构如 Gigabit Ethernet 的存储。另一种可能方案是称为 Infiniband 的专门 SAN 结构，它在服务器和存储单元之间提供对高速互连网络的硬件和软件支持。

14.7 稳定存储实现

在第七章，讨论了预写式日志(write-ahead log)，它要求使用稳定存储。根据定义，存储在稳定存储上的数据是永远不会丢失的。为了实现这种存储，需要在多个具有独立出错模式的存储设备(通常为磁盘)上重复所需信息。需要协调用于更新的写操作，以确保更新时所发生的差错不会使所有拷贝处于损坏状态，而且当恢复数据时，能强制使得所有数据处于一致和正确状态(即使在恢复时出现差错)。下面讨论如何满足这些要求。

磁盘写可能有三种情况：

- **成功完成**：数据正确写到磁盘上。
- **部分差错**：在传输中出现差错，这样有些扇区写上了新数据，而在差错发现时正在写

的扇区已损坏。

• **完全差错**:在磁盘开始写之前发生差错,这样磁盘原来数据值没有变化。

我们要求:若在写一块时发生差错,系统应检测到,并调用恢复程序使数据块恢复到一致状态。为此,系统必须为每个逻辑块维护两个物理拷贝。输出操作可按如下执行。

1. 将信息写到第一物理块上。
2. 当第一次写成功完成时,再将同样信息写到第二物理块上。
3. 只有在第二次写成功完成时,才声明操作完成。

在从差错中恢复时,每对物理块都要检查。如果两个块相同且没有检测到差错,那么无需采取任何动作。如果一块有检测差错,那么用另一块的值来替代它。如果两块没有检测差错但内容不同,那么用第二块的内容替代第一块的内容。这种恢复程序确保对稳定存储的写要么完全成功要么一点未做。

可以很容易地扩展这种程序,以允许使用任意数量的稳定存储的块拷贝。虽然大量拷贝可降低出错概率,但是通常采用两个拷贝来模拟稳定存储较为合理。除非差错损坏了所有拷贝,否则稳定存储上的数据可确保安全。

因为等待磁盘写(异步 I/O)操作的完成是费时的,许多存储阵列增加了 NVRAM 作为缓存。因为这种内存是非易失性的(通常它用电池作为单元电源的后备电源),所以可以相信它能够存储磁盘上的数据。因此,这些 NVRAM 也作为稳定存储的一部分。对它进行写要比对磁盘进行写快很多,这样大大地提高了性能。

14.8 第三级存储结构

如果一个 VCR 里面只有一个磁带且你不能取出或替换,或者一个录音机或 CD 播放机里面只有一个封好的媒介,那么你会买它吗?当然不会。你希望使用 VCR 或 CD 播放机,且它能使用许多相对便宜的磁带或磁盘。同样对于计算机,只带一个驱动器而使用许多便宜存储器可降低总体开销。

14.8.1 第三级存储设备

低价格是第三级存储的主要特征。因此,事实上,第三级存储是用**可移动媒介**制造的。最为普通的可移动媒介的例子有软盘、CD-ROM 和磁带;还有许多其他类型的第三级存储设备。

1. 可移动磁盘

可移动磁盘是一种第三级类型存储。软盘是可移动磁盘的一个例子。它们由薄而灵活的盘片加上磁性涂料和保护性塑料盒所制成。虽然普通软盘只能存储约 1 MB 的空间,但是相似技术可用于制造可容纳 1 GB 的可移动磁盘。可移动磁盘与硬盘几乎一样,但是其记

录层更容易因刮擦而受损。

磁光盘是另一种可移动磁盘。它将数据记录在涂有磁性材料的硬盘片上，但是记录技术与磁盘并不相同。与磁头相比，磁光头飞行时离表面更高，而且磁材料上加盖了较厚的塑料或玻璃的保护层。这种安排使磁光盘更能抵抗磁头碰撞。

驱动器有一线圈，以产生磁场；在室温下，这种磁场太大太弱以致于不能磁化磁盘上的一位。为了写一位，磁盘磁头在磁盘表面闪现一下激光束。该激光对准了待写的小斑点。激光加热该斑点，以使其易于磁化。这样，大而弱的磁场就能记录一位。

由于磁光头距离磁盘表面太高，以致于不能像硬盘的磁头那样检测到较小的磁盘磁场。所以，驱动器读取一位的方法是采用了一种称为**Kerr效应**的激光属性。当一束激光从磁场反弹回来时，根据磁场方向，激光束极性可能是顺时针或逆时针。这种转向就是磁头读取一位的方法。

另外一种类型的可移动磁盘是**光盘**。这些盘根本不使用磁。它们使用特殊材料，可以被激光所改变以出现一些相对明亮或相对暗一些的点。一种光盘技术的例子是相位变化盘。

相位变化盘涂有一种材料，它可变成晶体或无组织状态。晶体状态更加透明，因此在穿过相位变化材料并反弹回来时，激光束会更亮。相位变化驱动器使用三种不同强度的激光：低强度以读取数据、中强度通过溶化并将材料变成晶体状态以删除数据、高强度溶化材料使其成为无组织状态以便写数据。这种技术的最为常用例子是可记录CD-RW和DVD-RW。

这里描述的各种盘可多次使用。它们称为**读写盘**(read-write disk)。相反，**一次写多次读的盘**(Write-Once, Read-Many-times, WORM)属于另外一类。一种古老的制造WORM的方法是将一个铝薄膜盘片夹在两个玻璃或塑料盘片之间。当写一位时，驱动器使用激光在铝薄膜上烧一小孔。由于这种烧技术是不可逆的，所以盘上的任何扇区只能写一次。虽然可通过处处烧孔以删除WORM盘上的所有数据，但是事实上并不可能改变磁盘数据，这是因为孔只可以增加，与每一扇区相关的ECC码可以用来检测这种孔的增加。WORM盘是可靠的、经久的，因为金属层安全地夹在两保护玻璃或塑料盘片之间，且磁场并不能损坏数据。更新的一次写技术将数据记录在高分子膜上而不是铝上；这种材料吸收光以形成标志。这种技术用于可记录CD-R和DVD-R上。

一次写盘如CD-ROM和DVD，从生产厂家里出来就有数据了。它们使用了与WORM盘相似的技术(不过位是压上的而不是烧上的)，它们非常耐用。

绝大多数可移动盘要比非移动盘更慢。与旋转和寻道时间一样，写过程会更慢。

2. 磁带

磁带是另一种类型的可移动介质。一般而言，磁带通常能比磁盘或光盘容纳更多的数据。磁带驱动器和磁盘驱动器具有相似的传输速率。但是磁带随机访问要比磁盘寻道时间慢很多，这是因为磁带驱动器需要倒带或快进操作，这可能需要数秒钟或数分钟。

虽然一个典型的磁带驱动器要比典型磁盘驱动器更为昂贵，但是由于磁带的价格要比

相同容量的磁盘的价格更低。所以对于不需要快速随机访问的情况，磁带更为经济。磁带通常用于保存磁盘数据的备份。它们也用于大型超级计算中心保存科学研究或大型企业所使用的海量数据。

有的磁带能比磁盘驱动器保存更多的数据；磁带表面区域要比磁盘表面区域大很多。磁带容量可望进一步提高，这是因为磁带技术的**面密度**（或每平方英寸的位数）要比磁盘低很多。

大型磁带装置通常使用磁带机器人以在磁带驱动器和磁带库的存储位之间移动磁带。这些机器允许计算机对大量磁带进行自动访问。

机器人磁带库可以降低数据存储的总开销。有一段时间不需要使用的磁盘驻留文件可以**存档**到磁带上，磁带的每吉字节的价格更低；如果将来需要该文件，那么计算机可将它**调回**到磁盘以便经常使用。机器人磁带库有时称为**近线**存储，这是因为它位于高性能的**在线磁盘**和低价格的**离线磁带**（位于保存房间的架子上）之间。

3. 未来技术

在将来，其他存储技术可能会更重要。一种有希望的存储技术是**全息照相存储器**，它使用激光在特殊介质上存储全息照片。可以将黑白照片作为两维的像素数组。每个像素表示一位：黑表示 0，白表示 1。一个照片可以存储数百万位的数据。全息的所有位可以在激光一闪之间传输完，所以数据速率相当高。随着技术的不断改进，全息照相存储器可能会具有商业价值。

另一种热门研究的存储技术是基于微电子机械系统（Micro-Electronic Mechanical System，MEMS）。其思想是将电子芯片制造技术应用于制造小的数据存储机器。一种建议是制造 10 000 小磁头的阵列，该阵列之上是一平方厘米的磁性存储材料。当存储材料在磁头之后纵向移动时，每个磁头就读取材料上的自己的线性磁道。存储材料也可侧向地轻轻移动，以便所有磁头读取其下一磁道。虽然这种技术是否会成功有待观察，但是它可能会提供非易失性数据存储技术，其速度大于磁盘而价格低于半导体 DRAM。

不管存储介质是可移动磁盘、DVD 或磁带，操作系统需要提供多种功能以使用这些数据存储的可移动介质。这些功能将在 14.8.2 小节中讨论。

14.8.2 操作系统作业

操作系统的两个主要任务是管理物理设备和为应用程序提供一个虚拟机器的抽象。在本章将看到：对于磁盘，操作系统提供了两种抽象。一是生设备（raw device），即只是数据块的阵列。另一是文件系统。对于磁盘上的文件系统，操作系统会对来自多个应用程序的交叉请求进行排队和调度。现在讨论当存储介质是可移动的时，操作系统如何处理。

1. 应用接口

绝大多数操作系统几乎完全如同处理固定盘一样地处理可移动磁盘。当空盘插入到驱

动器中(或安装时),空盘必须格式化,并进而创建一个空文件系统。这种文件系统可如同硬盘一样地使用。

磁带通常采用不同处理。操作系统通常将磁带作为存储媒介。应用程序并不打开磁带上的一个文件;而是打开整个磁带以作为生设备。通常,磁带驱动器就专门作该应用程序所使用,直到它退出或关闭磁带设备。这种排他性有一定的道理,这是因为磁带随机访问可能需要花数十秒或甚至数分钟,因此不同应用程序可能引起的交织的磁带随机访问很可能会产生抖动现象。

当磁带驱动器作为生设备时,操作系统就不必提供文件系统服务。应用程序必须决定如何使用块数组。例如,将硬盘内容备份到磁带的程序,可能在磁带开头保存文件名列表和文件大小,并按顺序将文件数据复制到磁带上。

不难看出这种方式使用磁带会出现一些问题。由于每个应用程序自己决定如何组织磁带的规则,所以一个装满数据的磁带通常只能为创建它的应用程序所使用。例如,即使知道一个备份磁带包括文件名和文件大小的列表及按顺序存储的文件数据,还是会发现难以使用该磁带。文件名称到底如何存储?文件大小是二进制形式还是 ASCII?文件是一块一个,还是按字节串存储?人们甚至并不知道磁带的块大小,这是因为在写一块时可以进行选择。

对于磁盘驱动器,基本操作为 read、write 和 seek。另一方面,对于磁带,有一组不同的基本操作。不是 seek,磁带驱动器有 locate 操作。磁带 locate 操作比磁盘 seek 操作更为精确,因为它能定位磁带到某个特定逻辑导体,而不是整个磁道。定位到块 0 与重新缠绕磁带一样。

对于绝大多数磁带驱动器,可以定位到已经写到磁带上的任一块。然而,对于部分满的磁带,不可能定位到超过已写区域的空闲空间,因为绝大多数磁带驱动器管理其物理空间的方式是不同于磁盘驱动器的。对于磁盘驱动器,扇区有固定大小,在写数据之前格式化进程必须用来将空扇区放在其最后位置。绝大多数磁带有可变的块大小,每块的大小是在写该块时确定的,坏块会跳过,然后再写一块。这种操作解释了为什么不可能定位到已写区域之外的空闲空间:因为还没有确定逻辑块的位置和数量。

绝大多数磁带驱动器有一个 read position 操作以返回磁头所处的逻辑块号。许多磁带驱动器也支持 space 操作以用于相对定位。例如,操作 space −2 可以向后移动两个逻辑块。

对于绝大多数磁带驱动器,写一块具有副作用:即会删除写位置之后的所有内容。事实上,这种副作用意味着绝大多数磁带驱动器是只附加设备,因为更新磁带中央的某一块会实际上删除之后的所有内容。磁带驱动器在写一块时通过放上 EOT(end of tape,磁带尾部)标记以实现这种附加。驱动器不允许定位到 EOT 之后,但是能定位到 EOT 并接着开始写。这样做改写了原来的 EOT,并将它放在刚刚新写块之后。

从原理上来说,在磁带上可以实现一个文件系统。但是许多文件系统的数据结构和算法会与磁盘所用的不一样,这是因为磁带的只附加属性。

2. 文件命名

操作系统需要处理的另一个问题是如何命名可移动介质上的文件。对于固定磁盘，命名并不难。在个人计算机上，文件名由设备驱动器字母加上路径名组成。在 UNIX 上，文件名并不包括设备名，但是安装表使得操作系统能确定一个文件位于哪个驱动器上。但是如果磁盘是可移动的，那么知道某个驱动器过去某时包含一个盘并不意味着知道如何找到文件。如果世界上的每个可移动盘都有不同序列号，那么可移动盘可以用序列号作为前缀，但是为了确保不可能有两个相同序列号，那么每个序列号的长度应为 12 位数字。如果需要记住 12 位数字的序列号，那么谁能记住文件名？

当人们需要在一台计算机上向可移动磁盘上写数据而在另一台计算机上使用时，问题会更加困难。如果两个机器是同样类型且具有同样类型的可移动驱动器，那么这一困难只是知道可移动盘的内容和数据分布。但是如果机器或驱动器不同，那么会引起许多其他问题。即使驱动器兼容，那么不同计算机可能按不同顺序来存储字节，可能使用不同编码以分别存储二进制数字和字母(如个人计算机的 ASCII 与大型机的 EBCDIC)。

现代操作系统通常对可移动媒介的命名空间问题并不加以解决，而是让应用程序和用户来决定如何访问和解释这些数据。幸运的是，有些类型的可移动介质已经标准化，以致于所有计算机按同样方式进行使用。CD 就是一个例子。音乐 CD 具有统一格式，可为任何驱动器所使用。数据 CD 只有少数几种不同的格式，所以驱动器和操作系统驱动程序可以处理所有这些格式。DVD 格式也已标准化。

3. 层次存储管理

自动光盘塔(robotic jukebox)能使计算机切换磁带或光盘驱动器内的可移动盘，而无需人工干预。这种技术的两个用途是备份和层次化存储系统。将自动光盘塔用做备份较简单：当一个可移动盘已满时，计算机会让**光盘塔**自动切换到另一可移动盘。有的自动存储塔可以容纳数十个驱动器和数百个可移动盘，其机器臂负责磁带到驱动器的移动。

层次存储系统扩展了存储层次，使其不但包括内存和外存(即磁盘)还包括可移动存储。可移动存储通常采用磁带或可移动盘塔形式来实现。这种存储层次大、便宜但可能更慢。

虽然虚拟内存系统可直接扩展到第三层次的存储器，但是事实上这种扩展很少实现。其理由是从塔中获取数据要花费数十秒甚至数分钟，如此之久的延迟对于按需调页和其他虚拟内存来说是无法忍受的。

可移动存储通常用来扩展文件系统。小且经常使用的文件可以留在磁盘上，但大而旧且不常使用的文件可以备份到塔。对有的文件备份系统，文件目录条目仍继续存在，但是文件内容并不在外存上。如果应用程序试图打开文件，那么系统调用 open 会阻塞直到文件内容从可移动存储中调入为止。当内容再次从磁盘上可用时，操作 open 会将控制返回到应用程序，以便使用数据的磁盘驻留拷贝。层次存储管理已在普通分时系统如 TOPS-20 中实现，该操作系统在 20 世纪 70 年代末运行在 DEC 公司的小型计算机上。现在，HSM 通常用

于超级计算机中心和其他大型企业，以处理海量数据。

14.8.3 性能

与操作系统的其他组成部分一样，第三级存储性能的最为重要的三个指标是速度、可靠性和价格。

1. 速度

第三级存储的速度有两个方面：带宽和延迟。通常按每秒多少字节来测量带宽。**持续带宽**是一个大传输的平均数据速率，即字节数量被传输时间所除。**有效带宽**计算整个时间内（包括寻道或定位时间、盘片切换时间等）的平均值。从本质上来说，持续带宽为数据真正流动时的数据速率，有效带宽为驱动器所提供的总体数据速率。驱动器的带宽通常指持续带宽。

对于可移动磁盘，带宽可从 0.25 MB/s 到每秒数兆字节。对于磁带，带宽可从 0.25 MB/s 到超过 30 MB/s。最快的磁带驱动器要比可移动磁盘驱动器快很多。

速度的另一方面是**访问延迟**。对于这一性能参数，磁盘要比磁带快：磁盘存储基本上是二维的，所有数据位一经打开就可访问。磁盘访问简单地移动磁头到给定柱面，等待旋转延迟，这可能不到 5 s。相反，磁带存储是三维的。在任何时候，只有一小部分磁带可以被磁头所访问，而绝大多数数据位位于卷轴的数百或数千层磁带之下。磁带的随机访问要求缠绕磁带，直到所选块位于磁头之下，这可能需要数百秒或数千秒。所以通常说：磁带的随机访问要比磁盘的随机访问慢数千倍。

如果使用了光盘塔，那么访问延迟就更大了。为了换一个可移动磁盘，驱动器必须停止旋转，接着光盘塔必须切换盘片，驱动器必须再开始旋转。这种操作可能需要数秒，约比磁盘慢一百倍之多。因此，光盘塔切换盘片导致相当高的性能损失。

对于磁带，机器臂时间与磁盘一样。但是当切换磁带时，原磁带通常必须重新缠绕以便弹出，这一操作可能需要 4 min 之久。并且，在装入新磁带之后，需要许多时间以便驱动器校准磁带和准备 I/O。虽然慢磁带可能需要 1 min 或 2 min 的切换时间，但是与磁带的随机访问时间相比，这一时间并不多。

因此，可以这样说，磁盘塔的延迟为数十秒，而磁带塔的延迟为数百秒；切换磁带费时，但切换磁盘并不费时。但是也有例外：有的较为昂贵的磁带塔可在 30 s 内，进行重新缠绕、弹出、装新磁带、快速转到所需位置。

如果只关心盘片塔驱动器，那么带宽和延迟似乎不错。但是，如果关注盘片时，那么就有一个可怕的瓶颈。首先考虑带宽。与固定磁盘相比，移动库的带宽与存储容量之比并不好。读取一大磁盘上所存储的所有数据可能需要 1 小时。读取一个大的磁带库所存储的所有数据可能需要数年。同样，访问延迟也不好。为了便于说明，如果一个磁盘有 100 个请求队列，那么平均等待时间为 1 s。如果一个磁带库有 100 个请求队列，那么平均等待时间可

能超过 1 小时。第三级存储的低价格主要是由于大量便宜磁带可共享少量昂贵驱动器。由于库只能满足相对小数量的每小时的 I/O 请求，所以可移动库最适用于不常使用数据的存储。

2. 可靠性

虽然人们经常认为高性能意味着高速度，但是另一重要性能是可靠性。如果试图读取数据但因驱动器或媒介出差错又不能读到，那么事实上访问时间可能无限长而带宽无限小。因此，可移动存储媒介的可靠性是非常重要的。

可移动磁盘与固定磁盘相比，其可靠性要差，因为它更容易受到外界环境的影响，如灰尘、温度和湿度的较大变化，机械力如震动和弯曲。光盘通常被认为非常可靠，因为存储数据位的层由透明塑料和玻璃所保护。磁带的可靠性变化很大，且与驱动器有关。有的便宜驱动器只能使用一盘磁带数十次；而有的却能数百万次地使用同一磁带。与磁盘相比，磁带驱动器的磁头是个弱点。磁盘磁头在介质表面上飞行；而磁带磁头是与磁带相接触的。磁带擦碰会在数千或数万小时之后磨坏磁头。

总之，认为固定磁盘驱动器可能比可移动磁盘或磁带驱动器更为可靠，光盘可能比磁盘或磁带更为可靠。但是固定磁盘也有一个缺点。硬盘的磁头损坏通常会损坏数据，而磁带或光盘驱动器的出错并不会损坏数据盒。

3. 价格

存储价格是另一重要因素。这里有一个利用可移动媒介来降低总的存储价格的实际例子。假设一个 X GB 磁盘的价格为 $200；其中，$190 是用于包装、马达和控制器；$10 是用于磁盘片的。这样，磁盘的价格为 $200/$X$ GB。现在，假设采用可移动磁盘。对于一个驱动器和 10 盒盘，总价格为 $190 + $100，容量为 10$X$ GB，这样存储价格为 $29/$X$ GB。即使制造可移动盘可能更贵，可移动存储的每占字节的价格可能仍比硬盘要低很多，因为驱动器的费用为许多可移动盘所平摊。

图 14.13、图 14.14 和图 14.15 分别显示了 DRAM、硬磁盘和磁带驱动器的每兆字节的价格趋势。图中的价格是 BYTE 杂志和 PC Magazine 杂志的每年年底的广告最低价格。这些价格反映了杂志的读者的个人计算机市场，与小型或大型计算机相比价格要低。对磁带，这里的价格是针对有一个磁带的驱动器的价格。当一个磁带驱动器用多个磁带时，那么磁带存储的总价格就会变得更低，因为一个磁带价格只是一个驱动器价格的一小部分而已。然而，对于包括数千盒磁带的磁带库，存储价格主要是这些磁带的价格。在 2001 年，磁带的每占字节价格约为 $2。

DRAM 的价格波动很大。在 1981 年到 2000 年间，看到三次大幅度降价（约于 1981 年、1989 年和 1996 年），这是由于过度生产所造成的。也看到了两个时期的价格上涨（约于 1987 年和 1993 年），这是由于市场缺货所造成的。对于硬盘，总的来说价格稳步下跌，但从 1992 年以来下跌速度似乎增加了。到 1997 年为止，磁带驱动器的价格也稳步下跌。自从 1997 年以来，低廉磁带驱动器的价格停止了下跌，但是中等磁带技术（如 DAT/DDS）仍旧继

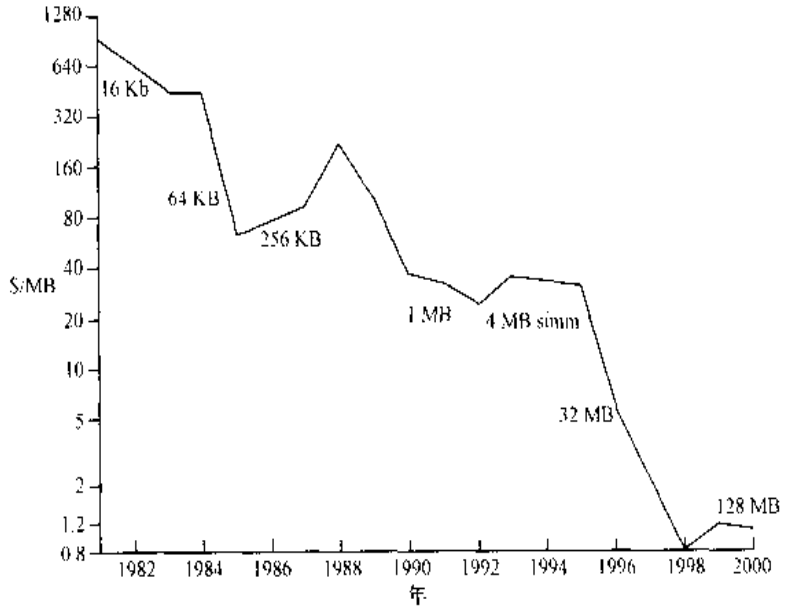


图 14.13 1981 年到 2000 年 DRAM 价格(每兆字节)

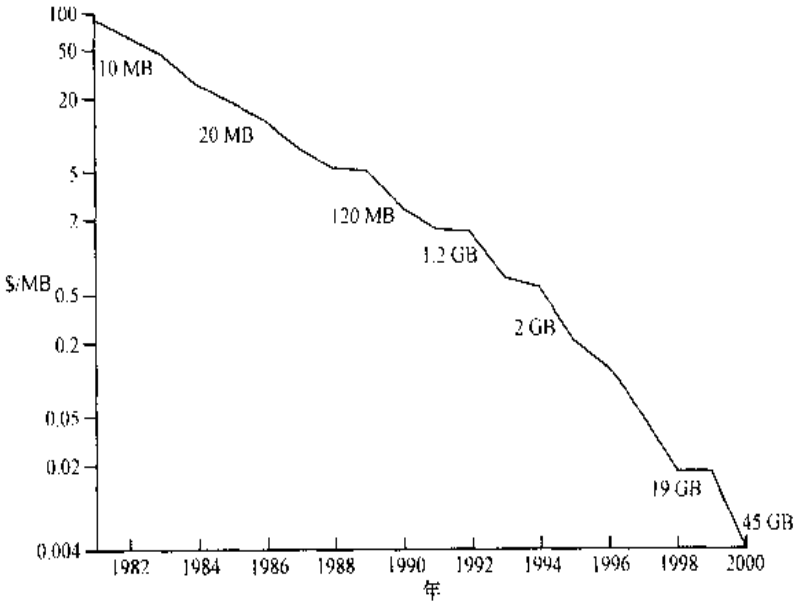


图 14.14 1981 年到 2000 年硬磁盘价格(每兆字节)

续下跌,现在已接近低廉磁带驱动器的价格。磁带驱动器的价格在 1984 年之前是没有的,这是因为 BYTE 杂志是面向个人计算机用户的,在 1984 年之前个人计算机通常不使用磁带驱动器。

通过比较这些图,可以看到磁盘存储价格比 DRAM 和磁带下跌更快。

在过去的二十年内,磁盘每兆字节的价格已经下跌了四个数量级,而相应内存的价格下跌三个数量级。现在内存要比磁盘贵 100 倍左右。

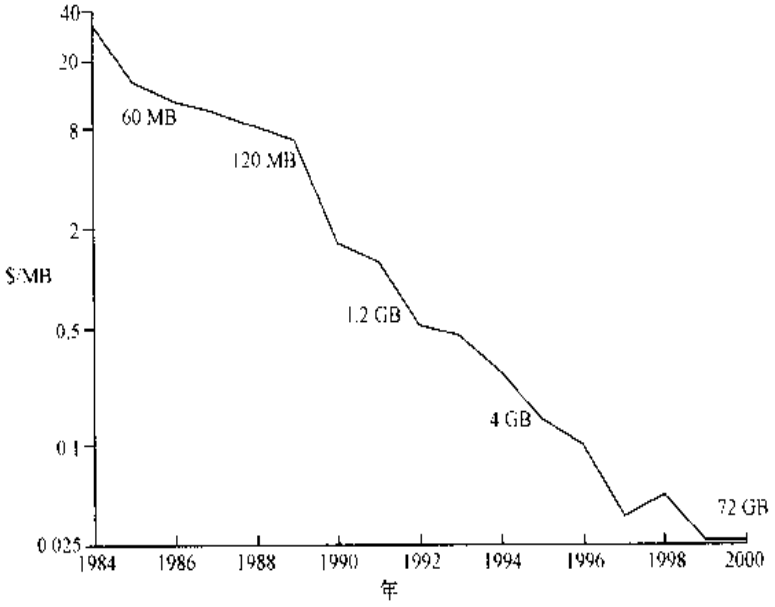


图 14.15 1984 年到 2000 年磁带价格(每兆字节)

每兆字节磁盘驱动器的价格与磁带驱动器的价格相比,下跌更快。事实上,磁盘驱动器的每兆字节的价格已接近没有磁带驱动器的磁带盒的价格。因此,小型或中型磁带库要比具有同样容量的磁盘具有更高的价格。磁盘价格的下跌使得第三存储几乎没用了:人们不再拥有这样的第三存储,其价格远远低于磁盘。看来第三存储的再次兴起必须要等待革命性的技术突破。现在,磁带存储通常只用于备份磁盘驱动器以及存储超过磁盘容量的海量数据。

14.9 小 结

磁盘驱动器是绝大多数计算机的主要外存 I/O 设备。磁盘 I/O 的请求主要由文件系统和虚拟内存系统所产生。每个请求以逻辑块号的形式,指定所引用的磁盘的地址。

磁盘调度算法可改善有效带宽、平均响应时间、响应时间差异。许多算法如 SSTF、SCAN、C-SCAN、LOOK 和 C-LOOK 通过磁盘队列的重排来改善这些指标。

性能可因外部碎片而降低。有些系统提供工具,以扫描文件系统,进而确定碎片文件;它们可移动块以降低碎片。对一个严重变成碎片的文件系统进行碎片整理可显著地改善性能,但是在整理碎片时,系统性能也会受些影响。复杂文件系统如 UNIX 的 FFS(Fast File System,快速文件系统)采用了许多措施以控制空间分配所引起的碎片,这样就不需要对磁盘进行重新组织。

操作系统管理磁盘块。首先,必须低级格式化磁盘从而在原来硬件上创建扇区,新磁盘通常已经低级格式化。接着,对磁盘进行分区,创建文件系统和分配启动块以存储系统的启

动程序。最后，当一块损坏时，系统必须提供一种方法以便不再使用该坏块，或用另一备份块来逻辑替代它。

因为有效交换空间对于提高性能十分关键，系统通常绕过文件系统，而直接使用低级磁盘访问以进行调页。有的系统将生磁盘分区用做交换空间，也有的系统使用文件系统内的一个文件作为交换空间。其他系统提供两种选择，以允许用户或系统管理员做出决定。

写前日志方案要求使用稳定存储。为了实现这种存储，需要在多个、具有不同差错模式的非易失性存储设备(通常是磁盘)上重复所需信息。也需要按一定控制方式来更新信息以确保能在数据传输出错或恢复错误之后能恢复数据。

由于大系统要求大量存储，所以经常通过 RAID 算法以使磁盘冗余。这些算法允许多个磁盘用于一个给定操作，即使在磁盘出错时也允许继续运行，甚至恢复数据。RAID 算法分成不同级别，每个级别都有不同的可靠性和数据传输速度的组合。

磁盘可通过两种方式与计算机系统相连：(1) 通过主机的本地 I/O 端口或(2) 通过网络连接如存储区域网络。

第三级存储包括磁盘和磁带驱动器，它们使用可移动媒介。可以使用许多不同技术，包括磁带、可移动磁盘、磁光盘和光盘。

对于可移动磁盘，操作系统通常提供文件系统接口的全部功能，包括空间管理和请求队列调度。对于许多操作系统，可移动媒介上的文件名称由驱动器名称和该驱动器内的文件名称组成。与采用一个名称以标识特定媒介相比，这种方法较简单但比较容易混淆。

对于磁带，操作系统通常只提供原始接口。对于光盘塔，许多操作系统都没有内置支持。光盘塔支持可通过驱动程序或专用于备份或 HSM 的应用程序来提供。

性能有三个重要方面：带宽、延迟和可靠性。磁盘和磁带有各种不同带宽，但是磁带随机访问延迟与磁盘相比要慢很多。光盘塔的盘片切换也相对较慢。因为光盘塔与盘片相比，有较低的驱动器速率，所以从光盘塔中读入大量数据需要大量时间。光介质(通过透明涂层保护敏感数据)要比磁介质(将磁材料暴露在外以致于可能更容易受到物理损坏)更耐用。

习题十四

14.1 除了 FCFS，没有其他的磁盘调度算法是真正公平的(可能会出现饥饿)。

a. 说明为什么这个断言是真。

b. 描述一个方法，修改像 SCAN 这样的算法以确保公平性。

c. 说明为什么在分时系统中公平性是一个重要的目标。

d. 给出三个以上例子，在这些情况下操作系统在服务 I/O 请求时做到"不公平"很重要。

14.2 假设一个磁盘驱动器有 5 000 个柱面，从 0 到 4 999。驱动器正在为柱面 143 的一个请求提供服务，且前面的一个服务请求是在柱面 125。按 FIFO 顺序，即将到来的请求队列是

86,1 470,913,1 774,948,1 509,1 022,1 750,130

从现在磁头位置开始，按照下面的磁盘调度算法，要满足队列中即将到来的请求要求磁头总的移动距离（按柱面数计）是多少？

a. FCFS

b. SSTF

c. SCAN

d. LOOK

e. C-SCAN

f. C-LOOK

14.3 基础物理中说：当一个物体在不变加速度 a 的情况下，距离 d 与时间 t 的关系可以用 $d=\frac{1}{2}at^2$ 来表示。假设在一次磁盘寻道中，像题 14.2 中一样，在开始一半，磁头以一个不变加速度加速，而在后一半，磁头以同一加速度减速。假设磁盘完成一个临近柱面的寻道需要 1 ms，一次寻道 5 000 柱面需要 18 ms。

a. 寻道的距离是磁头移动经过的柱面数，说明为什么寻道时间和寻道距离的平方根成正比。

b. 写一个寻道时间是寻道距离的函数的等式。这个等式应是这样的形式 $t=x+y\sqrt{L}$，t 是以毫秒为单位的时间，L 是以柱面数表示的寻道距离。

c. 计算题 14.2 中各种调度算法的总寻道时间。比较哪一种调度最快（有最小的总寻道时间）。

d. “加速百分比”是节省下的时间除以原先要用的时间。最快的调度算法与 FCFS 相比后的“加速百分比”是多少？

14.4 假设题 14.3 中的磁盘以 7 200 RPM 速度转动。

a. 磁盘驱动的平均旋转延迟时间是多少？

b. 在 a 中算出的时间里，可以寻道多少距离？

14.5 题 14.3 中的加速寻道是一种典型的硬盘驱动器的方式。与此相比，软盘（和 20 世纪 80 年代中期以前生产的许多硬盘）的寻道方式则是在一个固定的速率。假设题 14.3 中的磁盘使用固定速率的寻道方式，而不是固定加速度的寻道方式，所以寻道时间的形式为 $t=x+yL$，t 是以毫秒为单位的时间，L 是寻道距离。假设寻道临近的柱面需要 1 ms，每增加一个柱面需多用 0.5 ms。

a. 写出寻道时间是寻道距离的函数的等式。

b. 使用 a 中的寻道时间函数，计算题 14.2 中的各调度方法的总的寻道时间。你的答案和题 14.3c 是否一样？解释为什么一样或不一样。

c. 在这种情况下，最快调度算法超过 FCFS 的“加速百分比”是多少？

14.6 使用 SCAN 和 C-SCAN 磁盘调度算法，为磁盘调度写一个 Java 程序。

14.7 假设对于同样均衡分发的请求，比较 C-SCAN 和 SCAN 调度的性能。考虑平均响应时间（从请求到达时刻到请求的服务完成之间的时间）、响应时间的变化程度和有效带宽。问性能对于相关的寻道时间和旋转延迟的依赖如何？

14.8 除了 FCFS 调度外，其他的磁盘调度算法在单用户环境中是否有用，为什么？

14.9 解释为什么 SSTF 调度算法，同使用最内或最外的柱面相比，有喜欢使用中间柱面的倾向。

14.10 请求往往不是均衡分发的。例如，包含文件系统 FAT 或索引结点的柱面比仅包含文件内容的柱面的访问频率要高。假设你知道 50% 的请求都是对一小部分固定数目柱面的。

a. 对这种情况，本章讨论的调度算法中有没有哪些性能特别好？为什么？

b. 设计一个磁盘调度算法，利用此磁盘上的"热点"，提供更好的性能。

c. 文件系统一般是通过一个间接表找到数据块的，像 DOS 中的 FAT 或 UNIX 中的索引节点。描述一个或更多的利用此类间接表来提高磁盘性能的方法。

14.11 为什么在磁盘调度中一般不考虑旋转延迟？你怎样修改 SSTF、SCAN 和 C-SCAN 算法来让它们包含延迟优化？

14.12 使用 RAM 磁盘将会对你选择磁盘调度算法有什么影响？你需要考虑哪些因素？如果文件系统将最近用过的块保存在一个主存的缓冲中，对硬盘调度是否也要有同样的考虑？

14.13 在一个多任务环境中，为什么在系统中磁盘和控制器间平衡文件系统的 I/O 很重要？

14.14 从文件系统中重读代码页与使用交换空间保存它们的折中方法是什么？

14.15 有没有方法能实现真正稳定的存储？为什么？

14.16 硬盘驱动的可靠程度通常用一个叫做*故障间平均时间*(MTBF)的术语来量化地描述。虽然这个量叫做"时间"，MTBF 实际上用每次故障的驱动器小时数来测量。

a. 如果一个系统包含 1 000 个磁盘驱动器，每个磁盘驱动器的 MTBF 为 750 000 小时。下面的关于这个特大容量磁盘多长时间会出现一次错误的描述，哪一个最好：千年一次，百年一次，十年一次，一年一次，一个月一次，一个星期一次，一天一次，一小时一次，一分钟一次或一秒钟一次？

b. 死亡率统计显示，平均每个美国居民在 20 岁到 21 岁之间死亡的几率是 1:1 000，推断一个 20 岁的人的 MTBF 小时数。将这个小时数转化成年，这个 MTBF 告诉你这个 20 岁的人预期的寿命是多少？

c. 制造商保证某种磁盘驱动器模型的 MTBF 是一百万个小时。你能由此推断出这些驱动器的保修的年数是多少吗？

14.17 术语*高速宽带 SCSI-Ⅱ*表示操纵数据包在主机与设备之间移动时传输率在每秒 20 MB 的一条 SCSI 总线。假设一个高速宽带 SCSI-Ⅱ磁盘驱动的转速为 7 200 RPM，一个扇区大小为 512 B，每个磁道有 160 个扇区。

a. 估计这个驱动器支持的传输速度是每秒多少兆字节。

b. 假设这个驱动器有 7 000 个柱面，每个柱面 20 个磁道，一次磁头转换(从一磁盘片到另一个)需要 0.5 ms，到临近柱面的寻道时间为 2 ms。使用这些附加的信息对巨量数据传输所支持的传输速度做一个精确的估计。

c. 假设此驱动器的平均寻道时间为 8 ms，估计一下每秒 I/O 和对于一个读取分散在整个磁盘的单独扇区的随机访问工作负荷的有效传输速度是多少。

d. 计算随机访问每秒 I/O 和对于 I/O 大小为 4 KB、8 KB 和 64 KB 的传输速度。

e. 如果队列中有多个请求，像 SCAN 这样的调度算法能够减少平均寻道距离。假设一个正在读取8 KB页的随机访问工作负载，平均队列长度为 10，并且调度算法将平均寻道时间减少到 3 ms。计算每秒I/O和此驱动器的有效传输速度。

14.18 SCSI 总线可以连接一个以上的驱动器。特别地，一个高速宽带 SCSI-Ⅱ(题 14.17)总线最多可以连接 15 个磁盘驱动器。还设这个总线的速度为 20 MB/s。在任何时刻，在总线上只有一个包在某个硬盘内部的缓存与主机之间传递。然而，当其他盘在总线上传输数据包的时候，磁盘可以移动磁臂，并且在其他盘在总线上传输数据包的时候，磁盘也可以在它内部的缓存与磁碟之间传输数据。参考你在题 14.17 中算出的各种工作负载的传输速度，讨论一个高速宽带 SCSI-Ⅱ总线可以有效地使用多少个磁盘。

14.19 通过扇区保留(sector sparing)或扇区滑动(sector slipping),将坏的块重映射可能会影响性能。假设在题14.17中的驱动器总共有100个坏的扇区随机分布在磁盘中,并且每个坏的扇区被映射到一个保留的扇区,这个保留的扇区在不同的磁道,但在同一个柱面。对于一个队列长度为1(也就是说调度算法的选择不需要考虑)读取8 KB的随机访问工作负荷,估算每秒的I/O数和有效传输速度。一个坏的扇区对性能的影响是什么?

14.20 讨论扇区保留(sector sparing)和扇区滑动(sector slipping)各自的优点和缺点。

14.21 操作系统一般把可移动磁盘作为一个共享的文件系统,但对于磁带驱动器,却一次只允许一个应用使用。给出三个原因来解释为什么对待磁盘和磁带采取不同方式。描述一下操作系统如果支持对磁带塔的共享文件系统访问需要增加的新的特性。要共享磁带塔的应用程序是否需要特殊的属性?或他们是否可以像使用本地磁盘文件一样使用磁带中的文件?为什么?

14.22 在一个磁盘塔中,打开文件的数目大于磁盘塔中的驱动器数目的后果是什么?

14.23 如果磁带和磁盘有相同的存储区域密度,对价格与性能有什么影响?

14.24 如果硬磁盘最终和磁带的每吉字节的花费变成一样,磁带会不会过时?还是否需要它们?为什么?

14.25 你可以简单地通过对一个1 000 GB的由多个磁盘组成的存储系统和由第三级存储组成的存储系统的花费和性能进行比较来评估它们。假设每个磁盘容量为100 GB,花费为$1 000,传输速度为5 MB/S,平均访问延迟为15 ms。假设一个磁带库花费为每吉字节$10,传输速度为10 MB/s,平均访问延迟为20/s。计算一个纯磁盘系统的总花费、最大总数据传输速度和平均等待时间。如果你对工作负载做了任何假设,描述清楚并证明它们。现在,假设5%的数据是经常用到的,所以它们必须存在磁盘上,其他的95%存在磁带库中。进一步假设磁盘系统处理95%的请求,磁带库处理5%的请求。这个分层存储系统的总花费、最大总数据传输速度和平均等待时间分别是多少?

14.26 时常说磁带是顺序访问媒介,而磁盘是随机访问媒介。事实上,一个存储设备是否适合随机访问依赖于传输数据的大小。术语流量传输速度(streaming transfer rate)表示除去访问延迟的正在传输的数据传输速度。相应的有效传输速度(effective transfer rate)是总的字节数与总的秒数的比率,包括像访问延迟这样的开销。

假设在一个计算机中,二级高速缓存的访问延迟为8 ns,流量传输速度为800 MB/s。主存的访问延迟为60 ns,流量传输速度为每秒80 MB/s。磁盘的访问延迟为15 ms,流量传输速度为5 MB/s。磁带驱动的访问延迟为60 s,流量传输速度为每秒2 MB/s。

a. 随机访问导致设备的有效传输速度下降,因为在访问时间里没有数据被传输。对于所说的这个磁盘,如果按照平均的访问度,对于512 B、8 KB、1 MB和16 MB的流量传输的有效传输速度是多少?

b. 设备的利用率是它的有效传输速度与流量传输速度的比率。按照随机访问传输a中的4个不同大小的数据,分别计算磁盘驱动器的利用率。

c. 假设利用率超过25%是可以接受的。使用所给的性能数据,计算利用率可以被接受时的磁盘最小传输量。

d. 完成下面的句子:当传输量超过____字节时,磁盘是一种随机访问设备,当小于这个传输量时是顺序访问设备。

e. 计算高速缓存、内存和磁带的可以接受的利用率下的最小传输量。

f. 什么时候磁带是随机访问设备?什么时候它是顺序访问设备?

14.27 设想一个全息存储设备已经被发明出来。假设全息存储设备的花费为$10 000,它的平均访问时间为40 ms。假设它使用了一个价值$100的CD大小的盒子。这个盒子存储了40 000个图像,每个图像是分辨率为6 000×6 000像素(每个像素占1个bit)大小的方形黑白图像。再假设这个设备可以在1 μs的时间内读或写一张图。回答下面的问题。

a. 这个设备用在什么方面比较好?

b. 这个设备对计算机系统I/O的性能有什么影响?

c. 发明了这种设备将使哪些存储设备,如果有,变成过时?

14.28 假设一个5.25英寸的单面光盘的存储密度为每平方英寸1 Gbit,假设一个磁带的存储密度为每平方英寸20 Mbit,磁带的宽度为0.5英寸,长度为1 800英尺。计算这两种存储设备的存储容量的估计值。假设存在一个"光带",具有和磁带一样的物理尺寸,但存储密度和光盘一样。这个"光带"能够存储多少数据?如果磁带的价格为$25,那么对于"光带"来说适于销售的价格是多少?

14.29 假设可以认为1 KB是1 024 B,1 MB是$1 024^2$ B,1 GB是1024^3 B。这种级数一直到terabyte、petabyte、exabyte($1 024^6$)。在未来十年,许多新兴科学项目计划能够记录和存储几个exabyte的数据。要回答下面的问题,你需要做一些合理假设,说出你的假设。

a. 存储4 exabyte的数据需要多少磁盘?

b. 存储4 exabyte的数据需要多少磁带?

c. 存储4 exabyte的数据需要多少光带(题14.28)?

d. 存储4 exabyte的数据需要多少全息存储设备(题14.27)?

e. 每一种选择需要占用多少立方英尺存储空间?

14.30 讨论操作系统如何为磁带文件系统维护一个空闲空间链表。假设磁带技术是悬挂式的,而且使用EOT标志和14.8.2.4一节中的locate、space和read position命令。

推荐读物

Patterson等[1988]给出了有关独立磁盘冗余阵列(RAID)的论述,Chen等[1994]给出了详细的综述。Katz等[1980]论述了针对高性能计算的磁盘系统体系结构。Teorey和Pinkerton[1972]给出了早期的关于磁盘调度算法的比较性分析。他们使用模拟器模拟一个寻道时间对于经过的柱面是线性相关的磁盘。对于这个磁盘,LOOK对于长度小于140的队列是个好的选择,C-LOOK在队列长度大于100时是好的选择。King[1990]描述了通过在磁盘空闲时移动磁臂来减少寻道时间的一些方法。Seltzer等[1990]描述了除寻道时间外还要考虑旋转延迟的磁盘调度算法。Worthington等[1994]讨论了磁盘性能,还证明了缺陷管理的可忽略的性能后果。Ruemmler和Wilkes[1991]、Akyurek和Salem[1993]研究了合理地放置热数据可以改进寻道时间。Ruemmler和Wilkes[1994]描述了一个对于现代磁盘驱动器的精确性能模型。Worthington等[1995]指出怎样设置一些像存储区结构之类的磁盘的低级属性,Schindler和Gregory[1999]给出了有关这些工作更进一步的论述。

工作负载的I/O数据的多少和随机性对于磁盘性能有相当大的影响。Ousterhout

等[1985]和 Ruemmler 以及 Wilkes[1993]总结了许多工作负载的有趣特性，包括很多文件都很小，新创建的文件会很快被删除，很多被打开读的文件全部是按顺序读取的，多数查找是很短的。McKusick 等[1984]描述了 Berkeley 快速文件系统，这个文件系统使用了许多复杂的技术来对很多不同的工作负载取得良好的性能。McVoy 和 Kleiman[1991]论述了对于基本 FFS 的进一步改进。Quinlan[1991]描述了如何在一个带有磁盘缓存的 WORM 存储上实现文件系统。Richard[1990]论述了第三级存储的文件系统方法。Maher 等[1994]给出了关于分布式文件系统和第三级存储整合的概述。

分级存储的思想已经被研究了四分之一多世纪。例如，Mattson 等[1970]的一篇论文描述了一个预测分级存储性能的数学方法。Atl[1995]描述了商业操作系统中的可移动存储设备。Miller 和 Katz[1993]描述了在超级计算机环境中的第三级存储访问的特性。Benjamin[1990]给出了 NASA 的 EOSDIS 项目的大容量存储要求的概述。

全息存储技术是 Psaltis 和 Mok[1995]一篇文章的题目。Sincerbox[1994]汇总了 1963 年以来有关全息存储的论文。Asthana 和 Finkelstein[1995]描述了许多新兴存储技术，包括全息存储、光带和电子陷阱等。Toigo[2000]给出了一个关于现代磁盘技术和许多有潜力的未来存储技术的深度论述。

第五部分　分布式系统

分布式系统是一组不共享内存和时钟的处理器的集合。也就是说，每个处理器都有其自己的内存，处理器之间的通信通过局域网或广域网进行。分布式系统内处理器的大小和功能不尽相同，它们可能包括小的掌上型实时设备、个人计算机、工作站以及大的计算机系统。

分布式系统的优点在于用户可以访问由系统所维护的资源，进而可以提高计算速度、数据可用性及数据可靠性。分布式文件系统是文件服务系统，其用户、服务器、存储设备等分散在各处。因此，服务活动必须通过网络实现，用多个且相互独立的存储设备代替单一集中式数据存储。

由于系统是分布式的，它必须提供处理同步和通信的机制，以处理死锁和集中式系统中未曾遇到过的错误。

第十五章　分布式系统结构

分布式系统是不共享内存和时钟的一组处理器的集合，即每个处理器都有其自己的内存。处理器之间的通信可通过各种通信网络加以实现，如高速总线或电话线。在这一章中将讨论分布式系统的一般结构以及连接它们的网络，并且把它们和前面所研究的集中式系统相比较，对比二者在操作系统设计上的主要不同之处。第十六章和第十七章将给出详细的论述。

15.1　背　　景

分布式系统(distributed system)是通过**通信网络**(communication network)而松散连接的一组处理器的集合。从分布式系统的某一个特定处理器的角度来看，其他处理器及其资源都是**远程的**(remote)，而其本身资源则是本地的(local)。

分布式系统的处理器在大小和功能上不尽相同，可包括小的微处理器、工作站、小型机或大型通用计算机系统。有许多名称被用来称呼这些处理器，如*站点*(site)、*节点*(node)、*计算机*(computer)、*机器*(machine)或*主机*(host)，具体用哪一个必须取决于上下文环境。通常主要用*站点*来表示一个机器的位置，而用*主机*来表示某个站点的一个特定系统。一般地，某个站点的某个特定系统，即*服务器*，拥有位于其他站点的另一个主机，即*客户机*(或用户)所需要的资源。分布式系统的目的在于为这些资源共享提供一个高效的、方便的环境。图15.1显示了一个分布式系统。

15.1.1　分布式系统的优点

为什么需要建立分布式系统？主要有四个方面的原因：*资源共享*、*加速计算*、*可靠性*和*通信*。本节将逐一予以简单介绍。

1. 资源共享

如果许多站点(拥有不同的能力)相互连接，那么其中某个站点的用户就可以使用其他站点的可用资源。例如，站点A的某个用户可能正在使用站点B的一台激光打印机，同时，站点B的用户可以访问驻留在站点A上的文件。一般说来，分布式系统的**资源共享**(resource sharing)提供了诸如远程站点文件的共享、分布式数据库的信息处理、远程站点文件的打印、指定远程硬件设备的使用(如一个高速阵列处理器)和其他操作的执行。

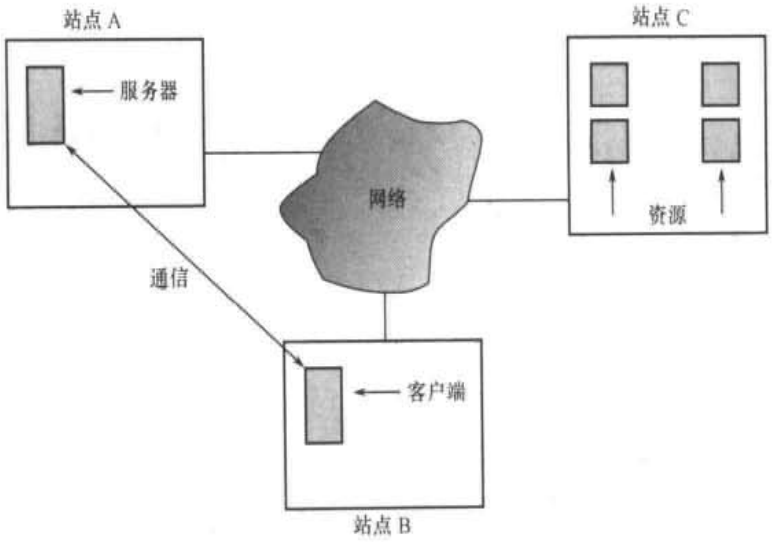


图 15.1 分布式系统

2. 加快计算速度

如果可以将一个特定的计算分化成能够并发运行的子运算，并且分布式系统允许将这些子运算分布到不同的站点，那么这些子运算可以并发地运行，因此**加快了计算速度**(computation speedup)。另外，如果某个站点在超负荷地工作，那么其中一部分可被移到其他站点，以减轻其负荷。这种作业移动被称为**负载共享**(load sharing)。然而，目前在商用系统中，自动负荷共享(在分布式系统中自动地移动作业)尚未得到普遍使用。

3. 可靠性

在分布式系统中，如果一个站点出错，其余站点可以继续工作，因此，相比较以往的系统而言，分布式系统具有更强的可靠性。如果系统由多个大的自动安装站点(如通用计算机)组成，那么其中的某一个出现故障并不影响其他成员。而另一方面，如果系统由小的机器组成，它们中的每一个都担负一些关键的系统操作(如终端字符的I/O或文件系统)，则单个机器的错误将会终止整个系统的操作。一般地，如果系统具有足够的冗余(包括硬件和数据)，即使某些站点出现故障，系统也会继续运作下去。

分布式系统必须能检测到站点故障，并采取适当的措施恢复故障，当然系统必须不再使用该站点的服务。此外，如果发生故障的站点的操作可以被其他站点接管，那么系统必须保证操作转移的正确。最后，当发生故障的站点恢复后，必须有一个能使其顺利结合到系统中的机制。如同将在第十六章和第十七章看到的，这些动作会带来一些问题，其可能的解决方法有多种。

4. 通信

当许多站点通过通信网络相互连接在一起时，不同站点的用户可以有机会交换信息。

在较低的层次上，系统间的**消息**传递类似于4.5节所述的单个计算机的消息系统。对于给定报文传递的情况，所有独立系统的高级功能都可扩展到分布式系统。这些操作包括文件传输、注册、发邮件、网络浏览以及远程程序调用(RPC)。

分布式系统的优点在于这些操作可以在彼此相隔很远的情况下执行。例如，两个地理位置分开的人员可以协同进行同一个项目。通过传输项目文件，登录彼此的远程系统以运行程序，交换邮件以协调工作，用户可以尽可能地缩小远程工作本身的局限性。

分布式系统的优点使得企业趋于**减小规模**(downsizing)，许多公司用工作站或个人计算机组成的网络替代大型机。通过这种方式，许多公司可获得更好的性能价格比、更强的资源布置、更灵活的设备扩充、更好的用户界面以及更方便的维护。

显然，一个操作系统被设计成通过消息系统通信的一组进程集合要比非消息传递的系统更易扩展成一个分布式系统。例如，MS-DOS不易与网络结合，就是因为它的内核是基于中断的且缺少对消息传递的支持。

15.1.2 分布式操作系统的类型

本节讨论两种基于网络的通用操作系统。网络操作系统往往易于实现，但与能够提供更多功能的分布式操作系统相比，它更难为用户所访问和使用。

1. 网络操作系统

网络操作系统(network operating system)为那些了解机器多样性的用户提供一个环境，通过登录适当的远程机器或从远程机器传送数据到其自己机器的方式，来访问远程资源。

远程登录

网络操作系统的一个重要功能是允许用户**远程登录**(remote login)到另一台计算机上。因特网为此提供了telnet工具。为了举例说明这个工具，假设一个在Brown大学的用户希望在地址为cs.utexas.edu且位置为Texas大学的一台计算机上进行计算。为此，该用户必须有那台机器的有效账户。进行远程登录时，用户发出如下命令：

```
telnet cs.utexas.edu
```

该命令使得在位于Brown大学的本地机与位于*cs.utexas.edu*的计算机之间形成连接。建立连接后，网络软件创建一个透明的、双向的连接，使用户键入的所有字符都被送到*cs.utexas.edu*的一个进程上，而该进程的所有输出都被送回用户。远程机器的进程询问的登录名和口令，获得正确信息后，该进程即作为用户的代理，而用户可以像本地用户所能做的那样在远程机器上进行计算。

远程文件传输

网络操作系统的另一个主要功能是提供一种机制以便从一台机器到另一台机器进行远

程文件传输(remote file transfer)。在此环境下,每个计算机保持它自己的文件系统。如果某个站点(如 *cs. brown. edu*)的用户希望访问位于另一位置(如 *cs. utexas. edu*)的计算机上的一个文件,则该文件必须被明确地从位于 Texas 的计算机拷贝到位于 Brown 的计算机上。

因特网为这种传输提供了**文件传输协议**(file transfer protocol,FTP)。假设位于 *cs. brown. edu* 的用户想将 *cs. utexas. edu* 上的 *paper. tex* 文件拷贝到自己机器上的 *my-paper. tex* 文件上,用户必须先调用 FTP 程序,通过执行

ftp cs. utexas. edu

该程序就会询问用户的登录名和口令,一旦收到正确信息,用户还需进入文件 *paper. tex* 所在的子目录,然后执行下面的命令拷贝文件:

get paper. tex my paper. tex

在此方法中,文件的位置不会传给用户,用户必须准确地知道每个文件的位置。另外,不会发生真正的文件共享,因为用户只能够将文件从一个站点拷贝到另一个站点,因此,可能存在同一文件的多个拷贝,从而导致浪费空间。此外,如果这些拷贝被修改,不同的拷贝将会不一致。

在这里的例子中,位于 Brown 大学的用户必须有登录 *cs. utexas. edu* 的许可,FTP 还提供一种方式,以允许那些没有 Texas 计算机账户的用户进行远程文件拷贝。这种远程拷贝是通过**匿名 FTP**(anonymous FTP)的方法完成的。该方法工作如下:被拷贝的文件(此为 *paper. tex*)必须放在一个特定的具有允许公共用户读取的子目录下(如 *ftp*),希望拷贝该文件的用户像以前一样使用 ftp 命令。当需要用户提供登录名时,用户键入 *anonymous* 以及一个随意的口令。

一旦匿名登录完成,系统必须小心保证这种部分授权的用户不能访问所有文件。一般来说,这些用户只被允许访问 *anonymous* 用户目录下的文件,该目录下的任何文件可被任何匿名用户访问,这些用户要服从文件所在机器的文件保护机制的制约。当然,匿名用户不能访问该目录之外的任何文件。

FTP 的实现机制与 telnet 类似,远程站点的一个服务程序负责监视系统 FTP 端口的连接请求。当执行登录身份验证后,用户就被允许执行远程命令。与 telnet 服务程序允许用户执行任何命令不同,FTP 服务程序只对预先设定的一组与文件相关的命令有反应。这些命令集包括:

- **get**:从远程机器传送文件到本地机器。
- **put**:将本地机器上的文件传送到远程机器。
- **ls**:列出远程机器当前目录下的文件。

另外,还有些命令允许改变传输模式(如二进制或 ASCII 文件)和决定连接状态。

无论是 telnet 还是 FTP 都需要用户改变语句表达。FTP 需要用户知道如何使用与一般操作系统命令完全不同的命令集,telnet 需要一点小的变化:用户必须知道远程系统的正

确命令。例如，一个 UNIX 用户远程登录到一台 VMS 机器上，他必须在会话期间使用 VMS 命令。如果他们不需要使用不同的命令集，那么无疑使用这些工具将更为方便。分布式操作系统就是设计用来改善这个问题的。

2. 分布式操作系统

对于分布式操作系统，用户可以如同访问本地资源一样来访问远程资源，从一个站点到另一站点的数据和程序迁移由分布式操作系统所控制。

数据迁移

假设 A 站点的用户想访问 B 站点的数据（如一个文件），系统可采用两种基本的方法传送数据。其中一种实现**数据迁移**（data migration）的方法是将整个文件传给 A，然后，所有对此文件的访问都是本地的。当用户不再需要访问文件时，文件的一个拷贝（如果它已被修改）被传回 B。即使对一个很大的文件进行一个细微的改变，所有的数据都必须被传回。这种机制可被视为一个自动 FTP 系统。这种方法曾被用于在第十六章要讨论的 AFS（Andrew file system）中，但它被证实效率太差。

另一种方法是只将对当前任务实际*所需*的文件部分传到站点 A，如果接下来需要另一部分，将进行另一次传送。当用户不再需要访问文件时，所有对文件的改变都会传回到站点 B（注意它与按需分页的相似处）。作为 AFS 的改进，Sun 微系统公司的网络文件系统（NFS）协议就使用这种方法（第十六章）。微软的 SMB 协议（运行在 TCP/IP 或 NETBUI 协议之上）同样允许在网络上共享文件。SMB 将在 21.6.1 小节中加以讨论。

显然，如果仅仅访问一个大文件中很小的一部分，后一种方法更好。如果要访问文件的大部分，那么拷贝整个文件效率更高。

在两种方法中，数据迁移包含了比从一个站点传送数据到另一个站点更多的内容。如果相关的两个站点不直接兼容，系统还必须完成不同的数据转换（例如，它们使用不同的字符编码或是采用不同的位数或顺序来表示整数）。

计算迁移

在某些情况下，可能要在系统之间传递计算而不是数据，这种方法叫**计算迁移**（computation migration）。例如，设想有这样一项作业：它需要访问不同站点的不同大文件，以获得这些文件的汇总。也许这样做会更有效率：分别在文件所在的站点访问这些文件，并把处理的结果返回给发起此项作业的站点。通常，如果数据传送的时间比执行远程命令的时间还长，则应使用远程命令。

这种计算可用不同的方法来实现。假设进程 P 想访问站点 A 的一个文件，文件的访问在站点 A 被执行，它可通过一个 RPC 开始。RPC 利用一个**数据报协议**（因特网上的 UDP）来执行位于远程系统的一个程序（4.6.2 小节），进程 P 调用站点 A 的一个预先设定的程序，

该程序正确地执行，然后将结果返回给 P。

另一种方法是，进程 P 可向站点 A 发送一条消息，之后站点 A 的操作系统将创建一个新的进程 Q，Q 的职能就是执行所指派的任务。当进程 Q 完成后，通过消息系统将所需的结果传回 P。在这样的设计中，进程 P 可以与进程 Q 并发地执行任务，事实上，可以有多个进程在多个站点上并发地运行。

这两种方法可以用来访问驻留在不同站点上的多个文件。一个 RPC 可能会导致调用另一个 RPC，甚至产生一个到另一站点的消息传递。类似地，进程 Q 可能在它的执行过程中向另一站点发送消息，继而又产生另一个进程，而该进程既可能向 Q 传回一个消息，也可能重复这个循环。

进程迁移

进程迁移是计算迁移的一个逻辑扩展。当一个进程被提交执行时，并不总是在它开始提交的站点上执行，进程的全部或部分可能在不同的站点上执行。该设计可能基于如下考虑：

- **负荷平衡**(load balancing)：进程(或子进程)可能被分散在网络上，从而平均工作负荷。
- **计算加速**(computation speedup)：如果单个进程可以被分成能在不同站点上并发运行的多个子进程，那么总进程的周转时间也许会缩短。
- **硬件偏好**(hardware preference)：进程可能具有某些特征使得它更适合在某些特定的处理器上执行(如矩阵求逆在一个数组处理器上，而不是在一个微处理器上)。
- **软件偏好**(software preference)：进程可能需要只在特定站点上才可用的软件，而该软件不能迁移，或它的迁移要比进程迁移昂贵。
- **数据存取**(data access)：就像在计算迁移那样，如果计算所使用的数据非常多，远程执行进程可能比传递所有的数据到本地更为有效。

可以使用两种互补的技术在一个计算机网络中迁移进程。在第一种方法中，系统设法隐藏进程已经从客户端移走的事实。该方法所具备的优点是，用户不必为实现迁移而显式地编程。当不需要用户输入来帮助远程执行程序时，该方法通常用于在同类型的系统中获取负荷平衡或加速计算。

另一种方法允许(或需要)用户清楚地指明进程应如何迁移。该方法通常用于迁移进程是为了满足某种硬件偏好或软件偏好。

你可能已经认识到 Web 具有许多分布式计算环境的特征。首先，它提供数据迁移(在一个 Web 服务器和 Web 客户端之间)。其次，它也提供计算迁移，例如，一个 Web 客户机可能触发一个 Web 服务器上的数据库操作。最后，运用 Java，它可以提供一种进程迁移形式：Java applets 从服务器传送到它们被执行的客户机上。一个网络操作系统提供了这些功能的大部分，但分布式操作系统使它们融合得更好，并且更容易使用。这样产生的结果就是一个强大而易用的工具——这也是万维网高速增长的原因之一。

15.1.3 阶段性小结

既然现在已经知道了建立分布式系统的原因和分布式操作系统的几种类型，接下来就可以学习实现这样一个系统所需的一些基本要素。本章余下部分将研究整个系统的最底层：计算机网络。第十六章将研究分布式文件系统，即一个典型的分时文件系统的分布式实现（多个用户共享文件和存储资源），文件被物理地分散到分布式系统的不同站点。第十七章将讨论分布式操作系统协调工作所需的方法。

15.2 拓 扑 结 构

系统内的站点可用多种方法物理连接，每种连接方式都有其优点和缺点。可按下面的标准来比较这些结构之间的差异：

- **安装成本**：物理连接系统站点的成本。
- **通信成本**：从站点 A 发送消息到站点 B 的时间和费用。
- **有效性**：不管连接或站点是否出错，数据能被访问的程度。

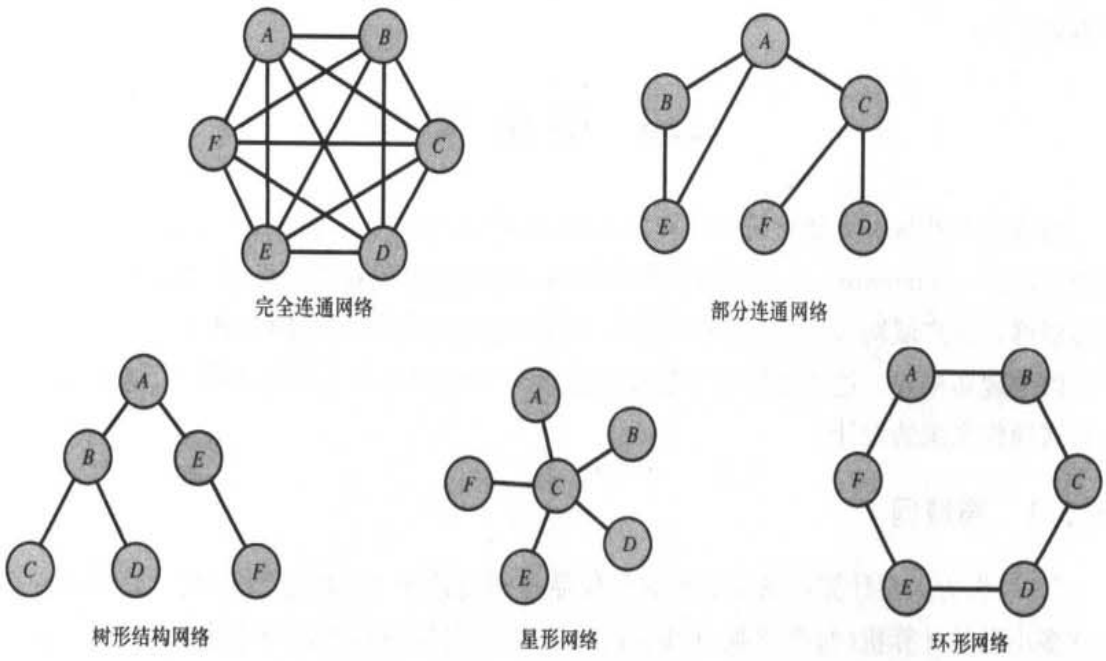


图 15.2 网络拓扑结构

图 15.2 描绘了几种不同的拓扑结构，图中用节点来表示站点，用从节点 A 到 B 的边来表示两个站点间的一条直接通信链接。在一个完全连接的网络中，每个站点都与其他所有站点直接相连。然而，连接数按站点数平方地增长，这将导致巨大的安装成本。因此，完全连接网络在大型系统中是不现实的。

在一个**部分连接网络**(partially connected network)中,直接连接存在于一些(但不是全部)站点之间,因此,这种结构的安装成本要比完全连接网络低。当然,如果站点 A 和 B 之间不直接相连,从一个站点发送消息到另一站点就必须通过一系列的通信链路,这将导致较高的通信成本。

如果通信链接出现故障,被传送的消息必须被重新发送。在某些情况下,可能会找到另一条路线,这样消息才能到达目的地。但在另一些情况下,故障将导致某些站点间无法连接。一个系统如果已被分成两个(或多个)相互之间没有任何链接的子系统,那么就称这些子系统为分区。根据这个定义,一个子系统可由一个单节点组成。

不同的部分连接网络类型包括树形结构网络、环形网络、星形网络,如图 15.2 所示。它们具有不同的故障特征、安装成本和通信成本。树状网络的安装成本相对较低,然而该结构的一个链接故障将导致该网络被分割。对环形网络结构,发生分割至少要有两个链接故障。因此,环状网络比树形网络更具有适用性。但由于消息可能不得不通过大量的链接,所以它的通信成本较高。对于星形网络结构,单个链接的故障将导致网络被分割,但其分开的部分是单个站点,此类分割可视为单个站点的故障。由于每个站点与其他站点至多存在两个链接,星形网络同样具有较低的通信成本。然而,中心站点的故障将导致系统的所有站点都变为无法连接。

15.3 网络类型

网络的两种基本类型为局域网和广域网,它们之间的不同主要在于地域分布范围。**局域网**(local area network,LAN)由分布在较小地域范围内的处理器组成,如单幢楼或一些紧邻的楼群。而**广域网**(wide area network,WAN)则是由分布在大的地域范围内(如中国)的自治的处理器构成。这些区别主要反映在速度和网络通信可靠性的不同上,这些都会影响分布式操作系统的设计。

15.3.1 局域网

LAN 作为大型计算机系统框架的替代品最早出现于 20 世纪 70 年代。许多企业发现用许多小型的计算机(每个都拥有其自己独立的应用程序),要比单个大的系统更为经济。由于每个小计算机可能需要一个完整的外围设备(如磁盘或打印机),且由于在一个企业中可能需要某种形式的数据共享,于是很自然地将这些小计算机系统连接起来就形成了网络。

局域网通常被设计成用于覆盖小的区域(如单个建筑物或一些紧邻的楼群)且一般用于办公环境。系统的所有站点之间相隔很近,故它们的通信链接相对于广域网而言具有高速度和低错误率的优点。为了获得这样的速度和可靠性,需要高质量(同时也是昂贵的)的电缆。这种网络可专用于网络数据交换。对于超长距离,使用高品质电缆的费用非常昂贵,一

般不使用这种专用电缆。

局域网最常用的链接方式是双绞线和光纤。最常见的结构是多路访问总线、环型网和星形网络。通信速度从 1 Mb/s，如 AppleTalk 和红外网，到 1 Gb/s，如千兆位以太网。10 Mb/s最为常用，*10 BaseT Ethernet* 就使用此速率。*100 BaseT Ethernet* 需要更高品质的电缆，但它可以达到 100 Mb/s，并日渐流行。**基于光纤的 FDDI 网**(optical-fiber-based FDDI networking)正在增加它的市场份额，它基于令牌且传输速度超过 100 Mb/s。

一个典型的局域网可由许多计算机，从大型机到膝上型电脑或 PDA，各种共享外围设备(如激光打印机或磁盘)以及一个或多个支持访问其他网络的网关(一种特别的处理器)所组成(如图 15.3 所示)。以太网的设计通常用来构建局域网，在一个以太网中没有中心控制器，由于是多路访问总线，新的主机可以很容易地加入到网络中来。

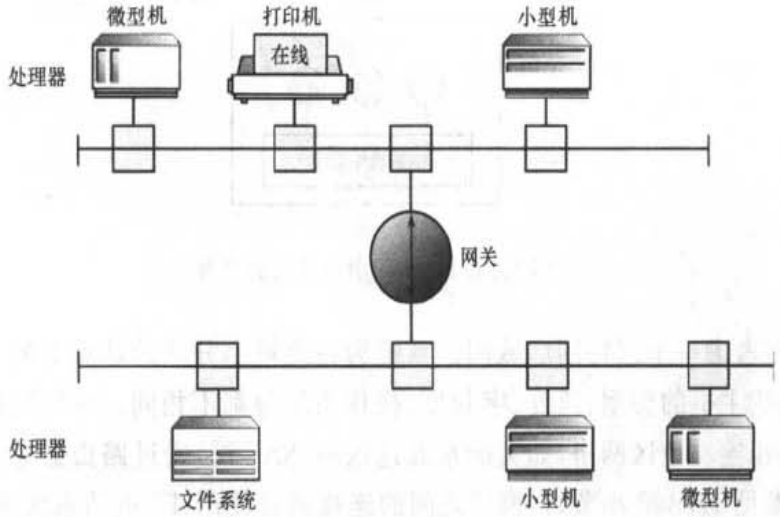


图 15.3 局域网

15.3.2 广域网

广域网出现于 20 世纪 60 年代，当时主要作为一个学术研究的项目，希望能在站点之间提供有效的通信，从而允许一个大的用户团体方便且经济地共享硬件和软件。设计和开发的第一个广域网是 **Arpanet**，Arpanet 的研究工作从 1968 年开始，已经从一个四站点的实验性网络发展为一个世界范围的网络，即包含了数百万计算机的因特网。一些国际性的私有商业网络在混合通信点之间提供通信，如公司下属机构的办公室。这些网络为它们的客户访问大范围的硬件和软件资源提供支持。

由于广域网站点分布在很大的地域上，通信也就相对较慢并且不可靠。典型的链接方式包括电话线、微波和卫星频道。这些通信链接由特殊的**通信处理器**(communication processor)所控制(图 15.4)，通信处理器负责定义网络站点间的通信接口以及在不同站点

之间的信息传输。

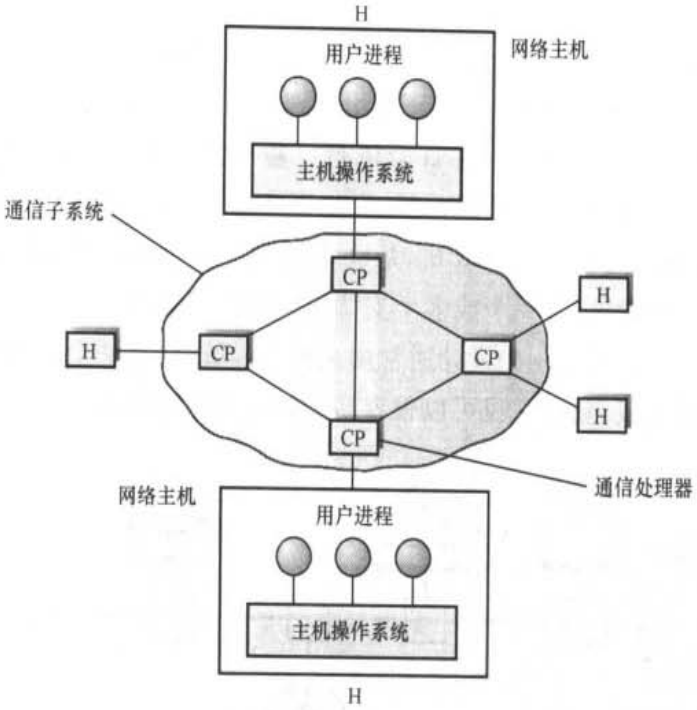


图 15.4 广域网中的通信处理器

举个例子,考虑一下因特网广域网。系统为在地域上分开的站点上的主机提供相互通信的功能。这些主机的类型、速度、字长度、操作系统等都不相同。局域网的主机通过地区网络与因特网相连。地区网络,如美国东北地区的 NSF 网,通过**路由器**相互连接形成世界范围的网络(参见 15.4.2 小节)。网络之间的连接通常使用 T1 电话系统业务,它在租用线路上可以提供 1.544 Mb/s 的传输速率。对于那些需要更快访问的站点,将 T1 聚集为多 T1 单元然后并行工作可以提供更多的吞吐量。例如,T3 就是由 28 个 T1 组成的,具有 45Mb/s 的传输速度。路由器控制了每条消息通过网络的路径,路由选择既可以是动态的,以增加通信效率,也可以是静态的,以减少安全风险或允许计算通信费用。

其他广域网使用电话线作为主要通信方法。**调制解调器**(modem)用来接收来自计算机端的数字信号,并将它转为用于电话系统的模拟信号,目的端的调制解调器将模拟信号转为数字信号,然后目的端接收这些数字信号。**UNIX 新闻网络**(UNIX news network,UUCP)允许系统彼此之间按预定时间通过调制解调器交换消息,然后消息被发送到其他邻近的系统,或者传播到网络的所有主机(公共信息),或者传送到它们的目的地(私有信息)。广域网通常比局域网慢,它们的传输速度从 1 200 b/s 到 1 Mb/s。

UUCP 很快被**点到点协议**(Point to Point Protocol,PPP)所代替。PPP 是通过调制解调器工作的 IP 协议的一种形式,它使得家用计算机可以与因特网完全连接。

15.4 通　信

已经讨论了网络的物理特性，接下来研究其内部工作方式。通信网络的设计必须考虑五个基本的问题：

- **命名和名字解析**(naming and name resolution)：两个进程如何定位以便进行通信？
- **路由策略**(routing strategies)：消息如何通过网络被发送？
- **包策略**(packet strategies)：包是被单独发送还是以一系列顺序发送？
- **连接策略**(connection strategies)：两个进程如何发送一系列消息？
- **线路竞争**(contention)：假设网络是一个共享的资源，如何解决冲突需求？

从 15.4.1 到 15.4.5 小节，将详细讨论上述问题。

15.4.1 命名和名字解析

网络通信的第一个组成部分是网络的系统命名。对于分别位于站点 A 和站点 B 的两个想交换信息的进程来说，它们必须能够区分对方。在一个计算机系统中，每个进程都有一个进程标识，消息可用进程标识标注地址。由于网络系统并不共享内存，开始时它们对目标进程的主机一无所知，甚至不知道其他进程的存在。

为了解决这个问题，远程系统的进程通常用<主机名，标识符>来标识，其中*主机名*是网络内的惟一名称，*标识符*可以是进程标识符或该主机的其他惟一号码。为了方便用户区分使用，*主机名称*通常用字符标识，而不是用数字。例如，站点 A 有名为 *homer*、*marge*、*bart* 和 *lisa* 的主机，无疑 *bart* 比 *12814831100* 更易记住。

名字对人而言是很方便的，但对于机器，数字则更加快速和简单。因此，必须有一种机制来将主机名**解析**(resolve)，从而能够把目标系统描述成联网硬件的主机 ID。这个解析机制类似于在程序编辑、连接、加载和执行过程中的名称－地址绑定(第九章)。主机名称有两种可能的模式。第一种，每个主机都有一个数据文件，它包含所有网络能访问到的其他主机的名字和地址(类似于编译时的绑定)。该模式的问题在于对网络增加和删除一个主机需要更新所有主机的数据文件。另一种就是将信息分布在网络系统中，而网络必须有一种协议来分布和检索这些信息。这个设计类似于执行时的绑定。第一种方法是因特网初期采用的方法，随着因特网的发展，它变得无法继续维持，所以现在采用第二种方法，即域名系统(domain name system，DNS)。

DNS 规定了主机的命名结构，包括名字到地址的解析。因特网的主机用一个由多部分组成的名称来进行逻辑编址。命名是从地址的最特殊部分到最一般部分，每个部分用句点分开。例如，*bob.cs.brown.edu* 指的是 Brown 大学计算机系一台名为 *bob* 的计算机。通常，系统解析地址时以相反的顺序检查主机名的组成部分。每个部分都有一个**名称服务器**—

仅仅是系统的一个进程——它接收一个名称，并返回负责该名称的名称服务器的地址。最后一步，连接该主机的名称服务器并返回一个主机 ID。在这里的例子中，对于 *bob.cs.brown.edu*，下面的步骤是系统 A 的一个进程请求与 *bob.cs.brown.edu* 通信的结果：

1. 系统 A 的内核向名称服务器发出一个关于 *edu* 域的请求，寻找负责 *brown.edu* 的名称服务器的地址。*edu* 域的名称服务器必须在一个已知地址的机器上，以便能向它发送请求（其他的顶级域名包括用于商业站点的 com、用于政府机构的 org 以及国家来确定的系统）。

2. *edu* 名称服务器返回 *brown.edu* 名称服务器的主机地址。

3. 系统 A 的内核向该地址的名称服务器查找 *cs.brown.edu*。

4. 返回一个地址，并向该地址发出查找 *bob.cs.brown.edu* 的请求，最后，返回该主机因特网地址（Internet address）的主机 ID（如 128.148.31.100）。

此协议似乎效率不高，但在每个名称服务器上可以保留本地缓存以加快速度。例如，edu 名称服务器在它的缓存中可能有 *brown.edu*，告知系统 A 它能解析名称的两个部分，并返回一个指向 *cs.brown.edu* 名称服务器的指针。当然，当名称服务器被移动或地址改变时，必须更新缓存的内容。事实上，这种服务非常重要，人们在协议中添加了许多优化和安全措施。考虑一下如果首级 *edu* 名字服务器崩溃后将会发生什么？很可能没有任何 *edu* 主机的地址能被解析出来，这使得它们根本无法被访问！解决的办法就是用第二个即备份名称服务器来复制主名称服务器的内容。

在引入域名服务器之前，因特网上所有的主机都需要有一个包含网络的每个主机名和其地址的文件拷贝。该文件的所有改变都必须在一个站点上注册登记（SRI-NIC 主机），所有的主机必须定时地从 SRI-NIC 主机上拷贝更新过的文件，以便能够与新系统联系或找到地址已更改的主机。而采用域名服务后，每个名称服务器站点负责更新该域的主机信息。例如，Brown 大学的任何主机的改变是 *brown.edu* 名称服务器的责任，而不再需要在其他任何地方公布。由于 *brown.edu* 是直接连接的，DNS 查询将自动地检索到更新信息。域内可能有自治的子域，以进一步分解主机名称和主机 ID 的更改责任。

通常，操作系统负责从它的进程中接收目的地址为＜主机名，标识号＞的消息，并将消息传送到适当的主机，然后目标主机的内核负责将消息传送给用此标识号命名的进程。此交换决不简单，15.4.4 小节将讨论它。

15.4.2 路由策略

当站点 A 的一个进程想与站点 B 的一个进程通信时，消息是如何传送的呢？如果 A 与 B 之间只有一条物理路径（如星形或树状结构的网络），必须通过该路径来传送消息。如果 A 到 B 有多条物理路径，则存在不同的路由选择。每个站点拥有一张**路由表**，它描述发送一条消息到其他站点的可选路径。该表可能还包括各条路径通信的速度和成本的信息，必要时，可以手动或通过交换路由信息的程序来更新这些信息。最为通用的三种路由策略是**固**

定路由、虚拟路由和动态路由：

• **固定路由**(fixed routing)：从 A 到 B 的路径是预先指定且不变的，除非出现硬件错误使该路径不能用。通常选择的是最短路径，以使通信成本最低。

• **虚拟路由**(virtual routing)：从 A 到 B 的路径在一个**会话**期间内是固定的，不同的会话期涉及的从 A 到 B 的消息可以有不同的路径。一个会话期既可以短到仅传送一个文件，也可长到如远程登录。

• **动态路由** (dynamic routing)：从站点 A 传送消息到站点 B 的路径仅在具体传送某个消息时选用。由于该选择决定是动态的，不同的消息可能被分配不同的路径。站点 A 可能做出的一个选择是，将消息传送到站点 C，转而再传送到站点 D，等等。最终，有一个站点必须将消息传送给站点 B。通常，一个站点采用最少时间的连接向另一站点发送消息。

这可以权衡三种设计方案。固定路由无法适应连接故障或负载改变。即，如果 A 与 B 之间已经建立了一路径，消息就必须由此路径传送，即使该路径产生故障或比其他可能的路径更繁忙。可以用虚拟路由来部分地纠正这个问题，也可用动态路由完全地避免此类问题。固定路由和虚拟路由保证从 A 到 B 的消息按它们所发送的顺序来传送，而用动态路由则可能出现顺序颠倒的情况。可以通过在每条消息上附加一个序列号来解决这个问题。

动态路由的建立和运行最为复杂，但在复杂环境中它是管理路由的最好方法。UNIX 既为简单网络的主机提供固定路由选择，也为复杂网络环境提供动态路由选择，甚至可以将两者混用。在一个站点中，主机可能仅仅需要知道如何到达连接本地网络与其他网络(如因特网或商业网)的系统，这样的一个节点被称为**网关**(gateway)。这些单个主机有一条固定的路由到网关，而网关本身使用动态路由到达网络的任何其他主机。

路由器是计算机网络负责路由的实体。路由器可以是一台具有路由软件的主机，或一个特殊的设备。无论采用哪种方式，一个路由器必须至少有两个网络连接，否则它将无处发送消息。路由器决定是否要把某个消息从接受到它的网络上传送到其他某个连接到这个路由器的网络。它通过分析此消息的目的网络地址来做出决定。路由器检查它的路由表来决定目标主机的位置，或者至少是目标主机所在的网络。在静态路由情况下，路由表的改变只能通过手工更新来完成(一个新的文件被加载到路由器上)。在动态路由情况下，可在路由器之间使用**路由协议**来通知它们网络的变化并允许它们自动更新自己的路由表。网关和路由器都是专用的硬件设备。它们运行固定的代码，而不是通用操作系统(其上可运行网络应用)。

15.4.3 分组策略

消息通常具有不同的长度。为了简化系统设计，通常使用称为**分组**(packet)、**帧**(frame)或**数据报**(datagram)的长度固定的消息来实现通信。只包含一个分组的通信可以通过用**无连接**(connectionless)的方式发送到它的目的地来实现。一个无连接的消息可能是**不可靠**的(unreliable)，此时发送方不能保证，也无法得知此包是否到达了目的地。另一种选择是使用

可靠的分组，此时通常有一个从目的地返回的分组来表明分组已到达(当然，返回的分组可能在路上丢失)。如果一个消息太长而不能装入一个分组中，或如果分组需要在两方之间来回流动，那么需要建立一个允许可靠地交换多个分组的连接。

15.4.4 连接策略

一旦消息能够到达它的目的地，进程可建立**通信会话**(communication session)来交换信息。需要通过网络通信的一对进程可以用多种方式连接，最常用的三种方法是**电路交换**、**消息交换**和**分组交换**：

- **电路交换**(circuit switching)：如果两个进程需要通信，则在它们之间建立一个永久的物理链接。在通信会话期间该链接被分配，任何其他进程在此期间都不能使用此连接(即便那两个进程某段时间未进行通信)。这种设计类似于电话系统所使用的技术。一旦一条通信线路对两个用户开放(用户 A 和 B)，任何人也不能使用这条线路，直到通信被明确终止(例如，用户之一挂断)。
- **消息交换**(message switching)：如果两个进程需要通信，则在传送一条消息期间建立一个暂时的链接。物理链接根据需要被动态地分配给通信者，且只分配给很短的时间。每条消息是一组带有系统信息的数据，如来源、目的以及纠错码(error correction code，ECC)，以允许通信网络将消息正确地传送到目的地。这种设计类似于邮局系统。每封信被认为是包含目的地址和来源(返回)地址的一条消息。多条消息(来源于不同的用户)可以在同一个链接上传送。
- **分组交换**(packet switching)：一个逻辑消息可能不得不被分成许多分组，每个分组可以被分别传送到它的目的地，因此在分组中除数据之外还必须包括一个源地址和目的地址。每个分组可以经由网络的不同路径，但当它们到达目的地后必须重新组合在一起。

这三种设计有着明显的折中。电路交换需要一定的建立时间，并且可能浪费网络带宽，但运送每个消息时只需很少的系统开销。而对于消息交换和分组交换，则只需很少的建立时间，但需要更多的开销。同样在分组交换中，每个消息被分为若干个分组，之后再被重新组合。分组交换是数据网络最常用的方法，因为这种方式最好地利用了网络带宽，而且数据在分成分组后被分别传送，继而在目的地重新组合并非无益。当然，在分解一个声音信号(假定一次电话通信)时，如果处理不当将可能引起大的混乱。

15.4.5 竞争

根据网络拓扑，一个链接可以将计算机网络的两个以上的站点联结起来，因此多个站点可能需要同时在一个链接上传输信息。这种情况主要发生在环状网络和多路总线结构网络中。因此，传输的信息可能变得混乱而必须被丢弃。发生这个问题时站点需要能被通知到，以便它们能重新发送报文。如果不采用特别的规定，这种情况可能被重复，从而导致性能下

降。现在已经开发了几个避免重复冲突的技术，包括**冲突检测**、**令牌传递**和**报文槽**。

• **CSMA/CD**：在通过一个连接传输报文(message)之前，站点必须侦听以确定是否有另一个报文正在此链路上传输，这种技术被称为**载波侦听多路存取**(carrier sense with multiple access，CSMA)。如果链路空闲，站点可以开始传输。否则，它必须等待(并继续侦听)，直到链路空闲。如果两个或更多站点正好同时开始传输(每个站点都认为没有其他站点使用链路)，则它们必须记录一个**冲突检测**(collision detection，CD)，并停止传输。每个站点将在随机的时间间隔后重新尝试。当站点 A 通过一条链路开始传输，它必须持续不断地侦听与其他站点间是否存在报文冲突。此方法的主要问题在于当系统非常忙时，可能发生许多冲突，从而导致性能下降。不过，CSMA/CD 已经成功地用在以太网这个最常用的网络系统上。(以太网协议由 IEEE 802.3 标准定义。)为了限制冲突数，可以限制每个以太网的主机数。加入更多的主机到一个拥挤的网络中可能导致非常低的网络吞吐量。当系统变快时，它们能在单个时间段发送更多的分组。因此需要适当减少每个以太网段的系统数量，以保证合理的网络性能。

• **令牌传递**(token passing)：一种独特的称为令牌的报文，持续不断地在系统(通常为环型结构)中循环。需要传输报文的站点必须等待，直到令牌到达，然后它从环中获取令牌，并开始传送消息。当站点完成它的报文传输，它重新传送令牌。此操作反过来允许另一站点接收和传输令牌，并开始它的报文传输。如果令牌丢失，系统必须检测到此丢失并产生一个新的令牌。它们通常通过一次**选举**，产生一个惟一的站点生成新的令牌。稍后在 17.6 节中，将介绍一种选举算法。IBM 和 HP/Apollo 系统采用了令牌传递方法。令牌传递网络的优点在于性能稳定。如果增加新的系统到网络中，可能会延长系统等待令牌的时间，但不会引起大的性能下降，而在以太网中却会如此。当然在轻负荷的网络上，由于系统可随时发送报文，以太网更有效。

• **报文槽**(message slot)：许多固定长度的报文槽连续不断地在系统(通常为环型结构)中循环，每个槽可以拥有一个固定大小的报文和控制信息(如来源和目的地，槽是空还是满)。准备传输的站点必须等待空槽的到达，然后将报文加入槽中，并设置相应的控制信息。带着报文的槽在网络上继续移动，当它到达一个站点后，该站点检查它的控制信息，以确定槽中是否包含给它的报文。如果没有，该站点重新循环槽和报文。否则，它移除报文，重置控制信息以表明此槽为空。站点可以利用此槽发送它自己的报文或释放此槽。由于槽只能包含固定长度的报文，所以一个逻辑报文可能不得不分成几个小的分组，每个分组在单独的槽中被发送。此方法已被 Cambridge 数字通信环网(Digital Communication Ring)所采用。

15.5 通信协议

当设计一个通信网络时，由于网络环境很慢并且容易出错，所以必须处理好协调同步通

信操作的问题。此外，网络的系统必须在一个或一组能够支持诸如确定主机名、网络的主机定位、建立连接等操作的协议上取得一致。可以通过将此问题分为多个层次来简化设计问题(以及相关的实现问题)。系统的每一层与其他系统上的同等层通信，每一层可以有它自己的协议。协议可以通过硬件或软件加以实现。例如，图 15.5 表明了两个计算机之间的逻辑通信，其中最底的三层用硬件实现。依据国际标准化组织(ISO)的标准，其分层有一定的描述方式。

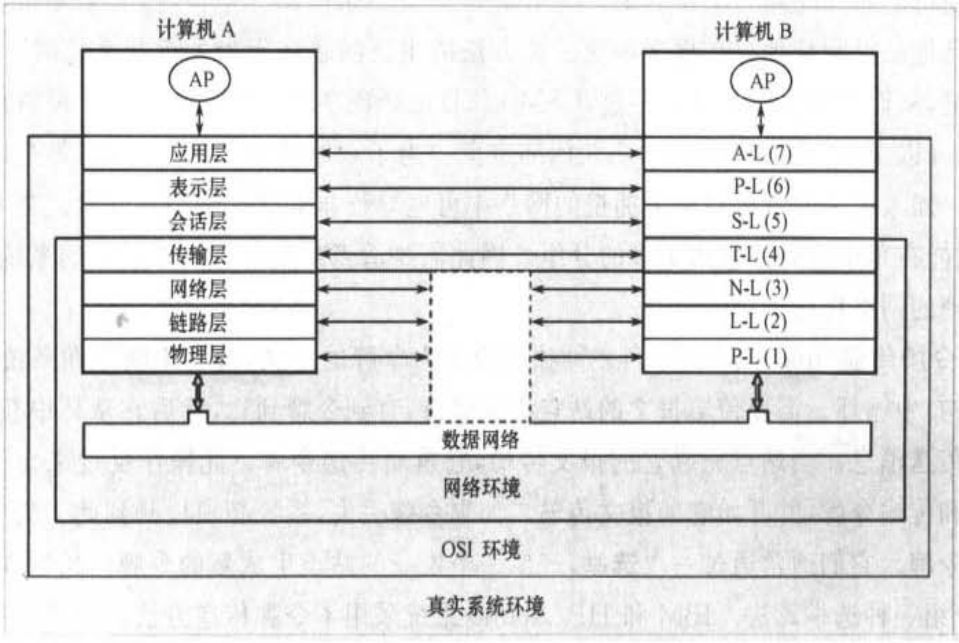


图 15.5 通过 ISO 网络模型通信的两台计算机

1. **物理层**(physical layer)：物理层负责物理传输比特流的机械和电气方面的具体细节。在物理层，通信系统必须在二进制 0 和 1 的电气表示上取得一致，以使当数据作为电信号流传送时，接收方能正确地将数据解释为二进制数据。该层通过网络设备硬件加以实现。

2. **数据链路层**(data-link layer)：数据链路层负责处理*帧*，或分组中的某些固定长度的部分，包括对物理层的错误检测和恢复。

3. **网络层**(network layer)：网络层负责提供连接和通信网络的分组路由，包括处理待发分组的地址，解析输入分组的地址，并维护路由信息以正确地响应负荷级别的改变。路由器工作在该层。

4. **运输层**(transport layer)：传输层负责提供底层对网络的访问以及客户机之间的报文传输，包括将报文分为分组、维护分组顺序、控制流并产生物理地址。

5. **会话层**(session layer)：会话层负责实现会话，或进程与进程之间的通信协议。通常，这些协议是远程登录以及文件和邮件传输的实际通信方式。

6. **表示层**(presentation layer)：表示层负责解决网络的不同站点的不同形式，包括字符

转换，以及半双工和全双工模式(字符 echoing)。

7. **应用层**(application layer)：应用层负责与用户直接交互，处理文件传输、远程登录协议、电子邮件以及分布式数据库设计。

图 15.6 概述了 **ISO 协议栈**(protocol stack)，这是一组互相配合的协议，它们描述了数据的物理流动。在逻辑上，协议的每一层与其他系统的同层通信。但在物理上，一个报文从应用层或更高的层出发，依次通过每一个更低的层，每一层可以修改此报文，包括添加报文头部。最后，报文到达数据网络层，并被转换成一个或多个分组来传输(图 15.7)。目的系统的数据链路层接收这些数据，在报文沿着协议栈上行的过程中，被分析、修改、剥去头部。最后到达应用层以供接收进程使用。

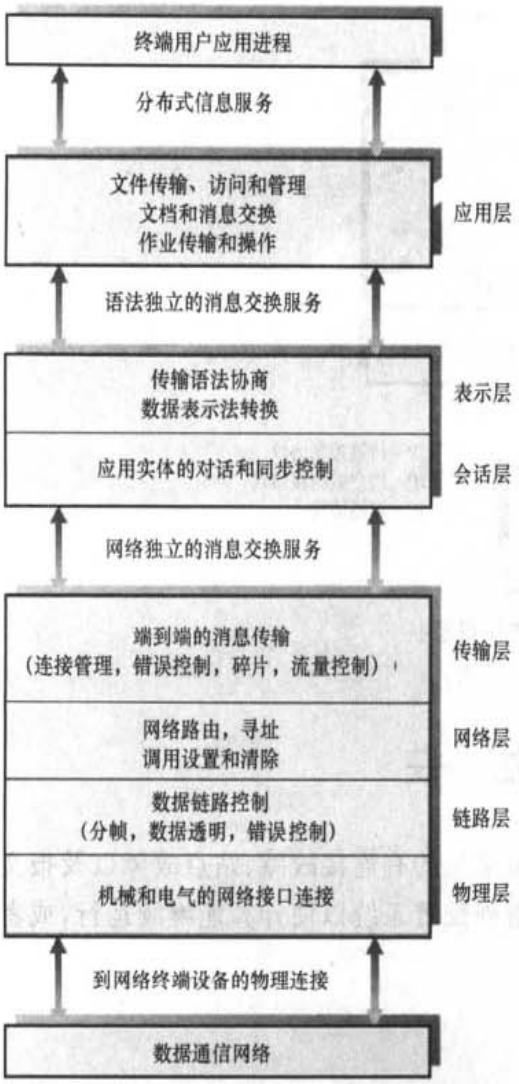


图 15.6 ISO 协议层

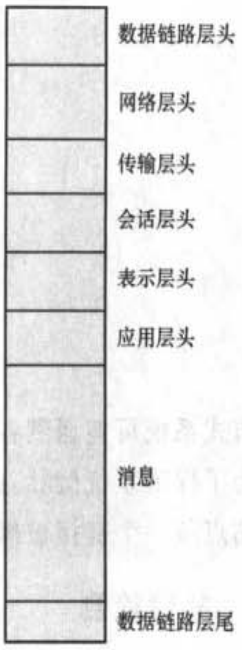


图 15.7 ISO 网络报文

ISO 模型使一些早期网络协议的工作趋于正式，但它在 20 世纪 70 年代末才得到发展，并未被推广使用。ISO 的部分基础被 UNIX 系统经久的且广泛使用的协议栈开发和采用

(先用于 Arpanet,后来发展为因特网)。

绝大多数因特网站点仍然使用 **Internet 协议**,通常被称为 **IP 协议**。服务通过无连接的**用户数据报协议**(UDP)和面向连接的**传输控制协议/Internet 协议**(TCP/IP)在 IP 层之上加以实现,TCP/IP 协议栈比 ISO 模型层数少。理论上,由于它每层组合了好几个功能,比 ISO 网络更难以实现,但是也更为有效。图 15.8 表示了 TCP/IP 模型和 ISO 模型的对应关系。IP 协议负责通过 TCP/IP 因特网传输 IP 数据报,即信息的基本单元。TCP 使用 IP 以在两个进程之间进行信息流的可靠传送。UDP/IP 是一个不可靠、无连接的传输协议,它使用 IP 来传送分组,但增加错误修正码以及一个协议**端口地址**来明确分组所指定的远程系统的进程。

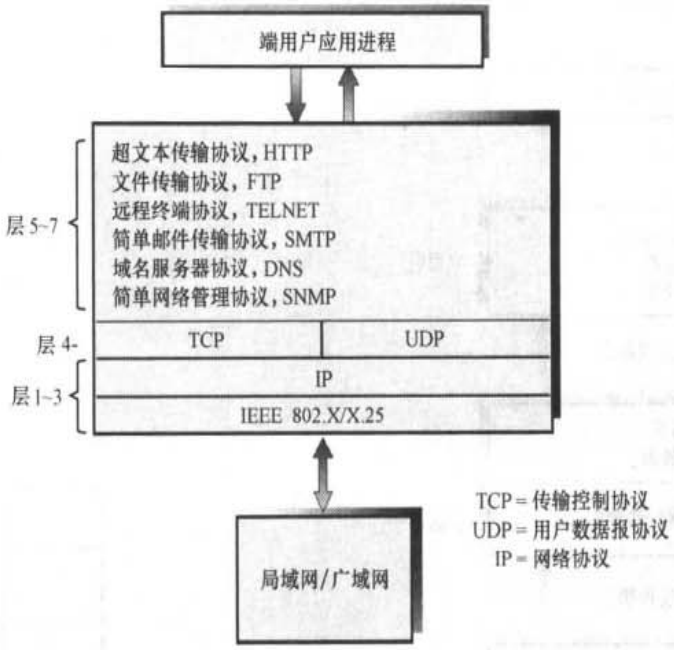


图 15.8 TCP/IP 的协议层

15.6 健 壮 性

分布式系统可能遇到各种类型的硬件故障。最常见的有连接故障、站点故障以及报文丢失。为了保证系统健壮,必须**检测到**任何错误,**重新配置**系统以使计算能继续运行,或者当一个站点或一个链接被修复后得以复原。

15.6.1 故障检测

对于无共享内存的环境,通常无法区分链接故障、站点故障和报文丢失,只能检测到故障发生,而无法确定是哪种类型。所以一旦检测到一个故障,必须根据特定的应用采取适当的措施。

可以使用**握手**(handshaking)过程来检测链接或站点故障。假设站点 A 和 B 之间有一条直接的物理链接,在固定时间间隔内,两个站点彼此发送一条 *I-am-up* 报文。如果站点 A 在预先设定的时间内未收到此消息,那么它可以假定 B 已出错,A 和 B 之间的链接出现故障,或来自于 B 的报文被丢失。此时,站点 A 有两个选择。它可以等待另一个时间间隔来接收 B 的 *I-am-up* 报文,或者也可以发送一条 *Are-you-up*? 的报文给 B。

如果站点 A 未接收到一条 *I-am-up* 报文或任何针对它询问的回答,上述过程可以重复进行。站点 A 所能明确的结论仅为发生了某种类型的故障。

通过另一条路由(如果存在的话)向 B 发送一条 *Are-you-up*? 消息,站点 A 可以设法区分连接故障或站点故障。如果当 B 收到此消息后,它立即明确响应。这个响应告诉 A,B 已准备好,出现的故障应是它们之间的直接链接故障。由于事先不知道报文从 A 到 B 并返回需要多少时间,必须使用一个**超时方案**(time-out scheme)。在 A 发送 *Are-you-up*? 消息的时候,它指定一个时间间隔用来等待 B 的响应。如果 A 在时间间隔内收到回应消息,它就可以确定 B 已准备好。如果没有收到(即发生超时),那么 A 只能推断可能发生了一个或多个以下情况:

- 站点 B 已停机;
- A 和 B 之间的直接链接(如果有)已停止;
- A 到 B 之间的可选路径已停止;
- 报文已丢失。

然而,站点 A 并不能确定究竟发生哪种情况。

15.6.2 重构

假定站点 A 通过前面描述的机制发现了故障,它必须启动一个程序,该程序将重新配置系统并继续正常的操作模式。

- 如果 A 与 B 之间的直接链接出现故障,此信息必须被广播到系统的每个站点,以使各个路由表能依此更新。
- 如果系统相信一个站点出错(由于不再能到达该站点),那么系统的每个站点必须被通知到,使它们不再试图使用出错站点的服务。如果是一个负责某些活动(如死锁检测)的中心协调者的站点失效,那么需要进行一次新协调者的选举。类似地,如果失效站点是逻辑环中的一部分,那么必须构建一个新的逻辑环。注意,如果站点未失效(即它已准备好,但不能到达),那么可能会出现不愿看到的情况:有两个站点同为协调者。当网络被断开时,两个协调者(每个负责它们各自的区域)可能启动竞争行动。例如,如果协调者负责实现互斥,可能看到两个进程同时在临界区执行的情形。

15.6.3 故障恢复

当一个出错的链接或站点被修复后,它必须能够与系统重新整合起来。

· 假设 A 和 B 之间的链接出现故障。当它被修复后，A 和 B 都必须被通知到。可以通过 15.6.1 小节所介绍的持续的握手程序来完成此通知。

· 假设站点 B 失效。当它被修复后，它必须通知所有其他站点。然后站点 B 可能不得不接收来自其他站点的信息以更新它的本地表，例如，它可能需要路由表、已坏的站点列表或未发送的消息和邮件等信息。如果站点未失效，而是简单地不能到达，那么也需要这些信息。

15.7 设计事项

把处理器和存储设备的多样性设计成对用户透明并不是一件简单任务。理想的情况是，分布式系统在用户看来就像一个传统的集中式系统。一个透明的分布式系统的用户界面不应区分本地和远程资源，即用户能像在本地一样地访问远程分布式系统，分布式系统应该负责查找资源以及安排适当的交互操作。

透明的另一方面体现在用户的灵活性上。它应该允许用户登录到系统中的任意机器，而不是强迫用户使用特定的机器。一个透明的分布式系统通过在任何登录机器上安插用户个人环境(例如 home 目录)来促进用户灵活性。美国卡内基－梅隆大学的 Andrew 文件系统和美国麻省理工学院的 Athena 项目都较多地提供此功能。NFS 则较小地提供此项功能。

*容错性*这个术语被广泛地使用。系统应该能在一定程度上容忍通信故障、机器故障、存储设备崩溃和存储介质的损耗。尽管出现了这些问题，一个**容错系统**(fault-tolerant system)应该能够继续运行，当然性能或功能可能会有所下降，而且下降程度应该与导致它的故障成合理的比例。少数部件出错的就中断的系统当然不具备容错性。遗憾的是，容错很难实现。大多数商用系统只提供有限的容错能力。例如，DEC/VAX 集群系统允许多个计算机共享一组磁盘。如果一个系统崩溃，用户仍然可以从其他系统访问到所需信息。当然，如果磁盘出现故障，所有系统都将无法访问。不过即使在这种情况下，RAID 可以保证数据的继续访问(参见 14.5 节)。

可扩展性(scalability)指的是系统可以适应日益增长的负荷的能力。系统只有有限的资源，在不断增长的负荷下，系统可能变得完全饱和。例如，对于一个文件系统，当服务器 CPU 按高利用率运转或磁盘几乎全满时，饱和就发生了。可扩展性是一个相对的特性，但它可被准确地测量。一个可扩展的系统面对日益增长的负荷，比一个不可扩展的系统要表现得更好。首先，其性能下降更合理；其次，其资源更慢地达到饱和。即便一个非常完美的设计也不可能适应不断增长的负荷需要。增加新的资源可能会解决这种问题，但它可能给其他资源产生附加的间接负荷(例如，增加新的机器到一个分布式系统中可能阻塞网络，并增加服务量)。更糟糕的是，扩展系统可能导致昂贵的设计修改。一个可扩展的系统应该具有可扩展的潜力，在分布式系统中，这是一种相当重要的能力，因为通过增加新机器或连接

两个网络来扩展网络是很平常的事。简单地说，一个可扩展的设计应能承受高服务负荷，适应用户群的增长，能够简单地整合新增的资源。

容错和可扩展性是彼此相关的。一个负荷严重的部件可能瘫痪，就像出故障部件一样。同样，从故障部件将负荷移到备份部件可能使后者饱和。通常，备用资源对保证可靠性和处理好高峰负荷是必要的。由于资源的多样性，分布式系统的一个内在的优点是拥有容错和可扩展的潜力。当然，不合适的设计可能会掩盖这种潜力，所以容错和可扩展的设计要考虑体现控制和数据的分布性。

大型分布式系统在很大程度上仍处于理论状态。没有普适的方法能保证系统的可扩展性。这就很好理解为何现行的设计不是可扩展的。这里讨论几个会引起问题的设计，然后再讨论可能的解决方案，所有这些讨论都是针对可扩展性的。

设计大型系统的一个原则是系统的任何部件的服务请求应由一个独立于系统的节点数的常量来限制。对于任何服务机制，如果其负荷要求与其系统大小成比例，那么一旦系统超过一定的规模，该机制必定成为阻碍。增加更多的资源并不能缓解此问题，因为此机制的性能限制了系统的增长。

在建立可扩展(以及容错)的系统时不应使用中央控制方案和中心资源的方法。集中式的实例有中心身份验证服务器、中心命名服务器以及中心文件服务器。集中式是构成系统的机器间功能不对称的一种形式。理想的选择是一种功能对称结构，即所有的机器在系统操作上具有平等地位，因此每台机器具有一定程度上的自治性。实际上，遵从这样一个法则是行不通的。例如，由于工作站依赖于一个中央磁盘，设立无盘化计算机就违反了功能对称。然而，自治和对称是人们所希望达到的重要目标。

对称和自治构造的一个实际近似是使用**集群**，整个系统被分成半自治的簇的集合。一个**簇**包括一组计算机以及一个专门的簇服务器，每个簇服务器应当在大部分时间内满足其辖内机器的请求，以使跨簇的资源查询相对很少。当然，此方法依赖于资源查询定位能力和适当安置部件单元的能力。如果簇被很好地平衡，即主管服务器足够满足所有簇的需求，那么它可被用做构建模块来扩大系统。

在任何服务的设计中，如何决定服务器的进程结构是一个主要的问题。当数百个激活的客户需要同时服务时，服务器就需要能在高峰时期有效地运作。单进程服务器显然不是一个好的选择，只要一个请求需要磁盘I/O，整个服务将会被阻塞。给每个客户机以一个进程是一个更好的选择，但必须考虑到进程之间频繁的关联切换的花费。由于所有的服务器进程需要共享信息，相关的问题也会发生。

对服务器体系结构的最好解决方案是运用第五章所介绍的轻量级进程或线程。用一组轻量级进程表示的抽象概念与一组共享资源的多线程控制其实是类似的。通常，一个轻量级进程并不与一个特定的客户机绑定，而是服务于不同客户机的单个请求。线程的设计可以是抢占的或非抢占的，如果线程允许运行到结束(非抢占)，则它们的共享数据不需要被明

确地保护。否则，必须使用一个明确的加锁机制。当然，如果需要服务器是可扩展的，那么某些形式的轻量级进程方案是必需的。

15.8 实例：连网

现在回到 15.4.1 小节提出的名称解析问题，考察在因特网上利用 TCP/IP 协议栈是如何操作的以及如何在不同以太网主机之间传送分组。

在一个 TCP/IP 网络中，每一个主机都有一个名字和一个相应的 32 位 Internet 号码（或者说主机 ID），这两个字符串都必须是惟一的，且为了便于管理名字空间，它们是分段的。名字是分级的（如 15.4.1 小节所述），它描述了主机名，然后是同主机相关的组织名。主机 ID 被分成一个网络号和一个主机号，分开的程度取决于网络的大小。一旦网络管理员分配了一个网络号码，那么该网络的站点就可以自由分配主机号码。

发送系统检测其路由表以确定一个路由器，从而向其发送分组。路由器使用主机 ID 的网络部分将分组从其来源网络传送到目的网络，然后目的系统接收分组。分组可以是一个完整的消息，也可以只是消息的一个部分，这样在消息被重新集成和传递到 TCP/UDP 层，并被目的进程接收之前，有很多分组存在。

现在已经了解了分组是如何从它的源网络转移到目的网络的。在一个网络中，一个分组是如何从发送方（如主机或路由器）到接收方的呢？每一个以太网设备都有一个为寻址而分配的惟一的字节号码，被称为**媒介访问控制**（MAC）**地址**（medium access control address）。局域网的两个设备可使用该号码进行通信。如果一个系统需要发送数据到另一个系统，内核生成一个包含目的系统 IP 地址的**地址解析协议**（Address Resolution Protocol，ARP）分组。该分组被**广播**通知到以太网的所有其他系统。广播使用一个特殊的网络地址（通常是最大的地址）来通知所有主机应该接收和处理这个分组。广播不是利用网关来进行重发，所以只有局域网的系统才能接收它们。只有 IP 地址与 ARP 请求的 IP 地址相符的系统才做出响应，并将它的 MAC 地址返回给发出询问的系统。为提高效率，主机将 IP-MAC 地址对存储在其内部表中。缓存内的项目是有时限的，如果在给定的时间内不需要访问该项目，它最终会从缓存中删除。这样，从网络中移除的主机最终会被遗忘。为了提供更好的性能，对使用频繁的主机的 ARP 项可以固化在 ARP 缓存中。

一旦一个以太网设备发布了其主机 ID 和地址，就可以开始通信。进程可以指定通信的主机名，内核获取名字，并使用 DNS 查找来确定目的主机 Internet 号码。信息从应用层通过软件层，最后被传送到硬件层。在硬件层，分组（或多个分组）在它的头部添加以太网地址，并在尾部添加检测分组损坏的**校验和**（图 15.9）。分组通过以太网设备置于网络上，其数据部分可能包含部分或全部原始报文数据，但它也可能包含一些组成报文的上层头部。换言之，原始报文的所有部分都必须从源发送到目的地，802.3 层（数据链路层）上的所有头部

都作为以太网分组中的数据。

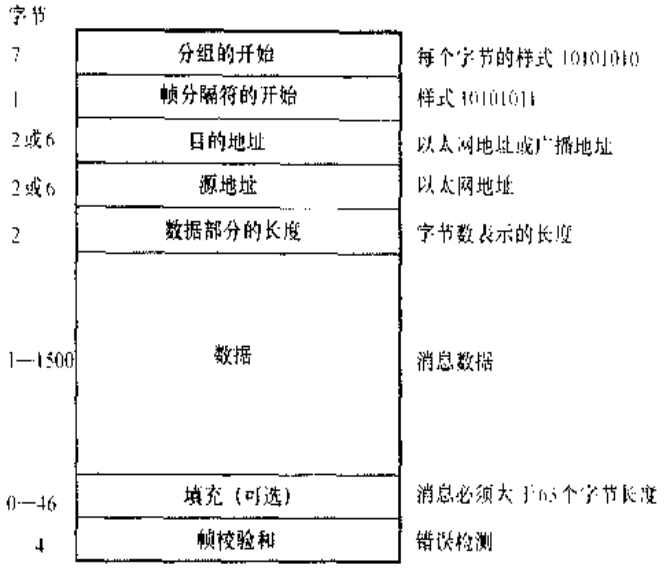


图 15.9 一个以太网的分组

如果目的地与源地在同一局域网内，系统可以查询其 ARP 缓存，以找到主机的以太网地址，然后将分组置于电缆上。目的端以太网设备则可发现分组的地址，并读取分组，再将它上传给协议栈。

如果目的系统与源系统不在同一网络上，源系统在其网络上查找适当的路由器并将分组发送给它。然后路由器沿着 WAN 转发包，直到到达它的目的网络。连接目的网络的路由器检查其 ARP 缓存，查找目的主机的以太网号码，并将包发送到主机。尽管在转发过程中，因为要不断使用下一路由器的以太网地址，所以数据链路层的头部会发生改变，分组的其他头部变化很少，直到分组被接收且被协议栈处理，并最终通过内核传送到接收进程。

15.9 小 结

分布式系统是不共享存储器或时钟的一组处理器的集合。每个处理器都有其本地存储器，处理器之间的通信通过不同的通信线路，如高速总线或电话线进行。分布式系统处理器的大小和功能都不尽相同。它们可能包括小的微处理器、工作站、小型机以及大的通用计算机系统。

系统的处理器通过一个通信网络连接起来，该网络可以用许多方式配置。网络可以完全或部分连接，可以是树型、星形、环型网以及多路总线结构。通信网络的设计必须考虑到路由策略和连接策略，必须解决线路竞争和安全问题。

一个分布式系统为用户提供访问系统资源的功能，可通过数据迁移、计算迁移或进程迁移来提供共享资源的访问。

分布式系统主要有两种类型：局域网和广域网。两者之间的主要区别在于它们的地域分布的不同。局域网由分布在较小地域范围的处理器组成，如一幢大楼或几个相邻的大楼。广域网则由分布在一个大的地域区域（如中国）内的自治的处理器组成。

协议栈，就像网络层模型规定的那样，处理消息数据，给它增加信息以保证到达它的目的地。名称系统，如域名服务器（DNS）用来将主机名译为网络地址，而另一个协议（如APR）则需用来将网络号码转换为网络设备的地址（如一个以太网地址）。如果系统位于不同的网络，还需用路由器将分组从源网络发送到目的网络。

一个分布式系统可能会遇到各种类型的硬件故障。为了使一个分布式系统具有容错能力，它必须能够检测到硬件故障并且重新配置系统。当故障修复后，必须重新配置系统。

习题十五

15.1 比较各种网络的拓扑结构的可靠性。

15.2 为什么大部分 WAN 都采用部分连接的拓扑结构？

15.3 WAN 和 LAN 之间的主要区别是什么？

15.4 以下环境中最适合采用哪种网络配置？

a. 宿舍楼的某层

b. 一所大学校园

c. 一个州

d. 一个国家

15.5 尽管 ISO 的模型把网络分成了 7 个功能层，但是大部分计算机系统在具体实现网络时还是采用了较少的几层，为什么这样做？这样使用少数几层会出现什么样的问题？

15.6 解释为什么把以太网段的系统速度提高一倍可能会导致网络性能的下降？用什么方法可以改进这种情况？

15.7 在什么情况下令牌传递网要比以太网效率高？

15.8 让网关在网络之间转发分组的优点和缺点各是什么？

15.9 网关和路由器使用专门的硬件设备有什么好处？如果选用通用计算机会有什么劣势？

15.10 在何种情况下使用名称服务器要比使用静态主机表更好？使用名称服务器的问题或复杂之处在哪里？你会采用什么方法来降低为了满足名称转换请求而产生的网络服务器访问量？

15.11 原始的 HTTP 协议采用 TCP/IP 作为其下层网络协议。对于每一页、每一幅图像、每一个 applet，会创建、使用和销毁一个单独的 TCP 会话。因为建立和销毁 TCP/IP 的开销很大，这种实现方法会产生性能方面的问题。那么是否使用 UDP 而非 TCP 就是一个很好的选择呢？你还有什么别的方法来提高 HTTP 的性能吗？

15.12 地址解析协议的作用是什么？为什么这样做要比每个主机自己分析分组以决定分组的目的地址要好呢？令牌网需要这样的协议吗？试做解释。

15.13 把计算机网络设计成对用户透明有什么优点和缺点？

15.14 设计者在实现一个网络透明的系统之前应该解决的两个重要问题是什么？

15.15 在由异构主机构成的网络中进行进程迁移通常是不可能的，因为考虑到操作系统以及体系结构间的不同。假设主机体系结构不同，试描述在以下两种情况下如何进行进程迁移？

a. 相同操作系统

b. 不同操作系统

15.16 为了构造一个健壮的分布式系统，必须知道哪些故障可能发生。

a. 列举分布式系统可能出现的三种主要故障。

b. 描述哪些故障项在集中式系统中也可能出现。

15.17 总是有必要知道发出的报文是否已经安全到达目的地吗？如果你回答"是"，请陈述理由，如果你回答"否"，请给出恰当的例子。

15.18 试描述一个算法，如何在环中的进程出现故障后重构一个逻辑环？

15.19 考虑一个拥有两个站点 A、B 的分布式系统，站点 A 是否有能力区分以下几种情况：

a. B 停机。

b. A 和 B 之间的链路出现故障。

c. B 的负荷严重超载，反应时间是正常时的 100 倍。

你的回答和分布式系统的故障恢复之间有什么联系吗？

推荐读物

Tanenbaum[1996]、Stallings[2000a]以及 Kurose 和 Ross[2001]讨论了通用计算机网络概览。Williams[2001]从计算机系统结构的观点描述了计算机网络。

Quarterman 和 Hoskins[1986]讨论了 Internet 和其他几种网络。Comer[1999]和 Comer[2000]描述了 Internet 及其协议。关于 TCP/IP，可以参考 Stevens[1994]和 Stevens[1995]。UNIX 网络编程可以充分参考 Stevens[1997]和 Stevens[1998]。

在 Popek 和 Walker[1985]（Locus 系统）、Cheriton 和 Zwaenepoel[1983]（V 内核）、Ousterhout 等[1988]（Sprite 网络操作系统），Balkovich 等[1985]（Athena 项目）、Tanenbaum 等[1990]和 Mullender 等[1990]（Amoeba 分布式操作系统）中，有关于分布式操作系统结构的讨论。有关 Amoeba 和 Sprite 之间的比较参考 Douglis 等[1991]。Mullender[1993]讨论了很多有关分布式计算的论题。

有关负荷平衡和负荷共享在 Harchol-Balter 和 Downey[1997]以及 Vee 和 Hsu[2000]中有所讨论。Harish 和 Owens[1999]讨论了 DNS 服务器的负荷平衡问题。关于进程迁移的讨论参见 Han 和 Ghosh[1998]以及 Milo 等[2000]。Artsy[1989]中探讨了有关进程迁移的一个特别问题。

关于分布式共享计算环境中的空闲工作站共享的方案在 Nichols[1987]、Mutka 和 Livny 以及 Litzkow 等[1988]中有所讨论。

Birman 和 Joseph[1987]讨论了发生故障时的可靠通信。Satyanarayanan[1989]中讨论了如何在大型分布式系统中整合安全性。

第十六章　分布式文件系统

前一章讨论了网络体系结构以及系统之间传送消息所需的底层协议，本章讨论该基本结构的一个应用。**分布式文件系统**(distributed file system，DFS)是一个文件系统典型分时模式的分布式实现，该系统中有多个用户共享文件和存储资源(第十二章)。分布式文件系统的目的是为了支持当文件被物理地分散在一个分布式系统中时的同类型共享。

本章将讨论一个 DFS 的设计和实现方法。首先要讨论的是 DFS 常用概念，然后通过分析一个有影响力的 DFS——Andrew 文件系统(AFS)来解释这些概念。

16.1　背　　景

分布式系统是一个通过通信网络相互连接的松散结合的机器集合。用*机器*(machine)这个术语来表示大型机或工作站。从分布式系统中的某个特定的机器来看，其他的机器和它们各自的资源都是*远程的*，而该机器本身的资源是*本地的*。

在这一章，术语 DFS 通常表示分布式文件系统，而不是商用的 Transarc DFS 产品，后者表示为 *Transarc DFS*。

为了解释 DFS 的结构，需要定义这些术语：服务(*service*)、服务器(*server*)和客户(*client*)。**服务**是运行在一个或多个机器上的软件实体，它为某个事先未知的客户提供某种类型的功能。**服务器**是运行在单个机器上的服务软件。**客户**是指能通过一组操作来调用某个服务的进程，这些操作构成了**客户接口**。有时，底层的接口是为机器间交互而定义的，所以也称为**机器间的接口**。

现在，可以说文件系统是用来为客户提供文件服务的，一个文件服务的客户接口由一系列简单的文件原语操作组成，如产生一个文件、删除一个文件、读文件、写文件等。文件服务控制的主要硬件是一组本地辅助存储设备(通常是磁盘)，文件就存储在其中，并根据客户的请求从中重新取回。

一个 DFS 的客户机、服务器和存储设备都分散在分布式系统的机器上。因此，服务活动必须在网络上进行，系统有多个独立的存储设备，而不是单一的集中式数据存储。你会发现，DFS 的具体结构和实现可能会有所不同。在某些结构中，服务器运行在专用的机器上，而在另一些结构中，一台机器既可用做服务器又可当做客户机。DFS 可作为分布式操作系统的一部分来实现，或者作为一层专门用来管理传统操作系统和文件系统之间通信的软件

层来实现。DFS 与众不同的特征在于系统中的客户机和服务器具有多样性和自治性。

在理想的情况下，对于客户机而言，DFS 最好能表现得如同一个传统的集中式文件系统，它的服务器和存储设备的多样性和分散性应该被隐藏起来，即 DFS 的客户接口不应区分本地和远程文件。这就需要 DFS 负责文件定位和安排数据传输。通过将用户环境(即主目录)带到任何一个用户登录的地方，**透明的** DFS 促进了用户的灵活性。

衡量 DFS 最重要的性能指标是满足服务请求所需的时间。在传统的系统中，此时间包括磁盘存取时间和少量的 CPU 处理时间。而在 DFS 中，由于分布式结构的原因，远程访问需要额外的开销。这些额外的开销包括传递请求到服务器所需的时间、通过网络获取响应并返回给客户机的响应时间。除信息的传送外，每一方面上还有运行通信协议软件的 CPU 开销。DFS 的性能可以视为 DFS 透明性的另一方面，理想的 DFS 的性能应该和传统的文件系统性能差距不大。

DFS 管理一系列分散的存储设备，这是 DFS 的关键特征。DFS 管理的所有存储空间由不同的、远程的、小的存储空间所组成。通常，这些连续的存储空间对应于文件集。一个**部件单元**是能存储在单个机器上的最小文件集，它独立于其他单元。属于相同部件单元的所有文件必须驻留在相同的位置。

16.2 命名和透明性

命名是在逻辑对象和物理对象之间建立的映射。例如，用户处理用文件名表示的逻辑数据对象，而系统操作需要管理的是存储在磁道上的物理数据区。通常，用户用一个文本名字来关联一个文件，这个名字被映射到一个更低层的数字标识，而它又继续映射到磁盘。这个多级映射提供给用户一个文件的抽象，它隐藏了文件如何存储以及存储在磁盘何处的细节。

在一个透明的 DFS 中，文件抽象还增加了一条要求：即文件被存储在网络中的何处。在传统的文件系统中，名字映射范围是磁盘中的某个地址，在 DFS 中，此范围扩展到文件所在的特定机器。将文件抽象处理的概念再进一步引申将引发**文件复制**的可能性。给定一个文件名，映射返回一系列该文件复本的位置。在这种抽象概念中，多个复本和它们的位置都是被隐藏的。

16.2.1 命名结构

关于 DFS 中的名字映射，需要区分两个概念：

1. **位置透明性**：文件名字不揭示任何有关文件物理存储位置的线索。
2. **位置独立性**：当文件的物理存储位置改变时，不需要改变文件名。

由于处于不同等级的文件具有不同的名字(如用户级的文本名字和系统级的数字标识)，两个定义都是与命名等级相对而言的。位置独立的命名方案是一种动态的映射，因为

它能在不同的时间把同样的文件名映射到不同的位置。因此，位置独立性是比位置透明性更强的性能。

实际上，现在大多数的 DFS 为用户级的名字提供一个静态的、位置透明的映射，这些系统不支持**文件移动**，即不能自动改变文件位置。因此，位置独立性的概念与这些系统无关。文件与一组磁盘块永久地相关联。文件和磁盘可以手动地在机器间移动，但文件移动意味着一个自动的、由操作系统引发的动作。只有 AFS 和少数实验性的文件系统支持位置独立性和文件活动性。AFS 支持文件活动的目的是为了管理。一个协议支持 AFS 部件单元的移动以满足高级用户的请求，而不需要改变相应文件的用户级名字或低级别名字。

可以通过下面几个方面来进一步区分位置独立性和静态位置透明性：

· 正如位置独立性所表现的，数据与位置的分离提供了更好的文件抽象。一个文件名应体现出文件的大多数重要属性，即它的内容而不是位置。位置独立性文件可被视为未关联到某个特定的存储位置的逻辑数据容器。如果仅支持位置透明性，文件名仍然表明了一个特定的（虽然是隐藏的）物理磁盘块集合。

· 静态位置透明性为用户提供一个方便的共享数据的方法。用户可通过位置透明的方式简单命名文件来共享远程文件，就好像这些文件在本地一样。然而，存储空间的共享就比较麻烦，因为逻辑名仍然静态地关联在存储设备上。位置独立性促进了存储空间共享，同时也包括数据对象的共享。当文件能被移动时，整个系统范围内的存储空间就像一个单个的虚拟资源一样，这样的一个优点在于具备了平衡地跨系统使用磁盘的能力。

· 位置独立性将命名级别从存储器体系和计算机间结构中分开。相反，如果使用了静态位置透明性（尽管名字是透明的），容易暴露部件单元和机器间的交流。机器以一种类似于命名结构的方式来配置。这种配置可能会过分地限制系统的体系结构，并与其他事项冲突。管理根目录的服务器是一个使用命名层次结构的例子，且与分散的指导思想相矛盾。

一旦完成名字和位置的分离，客户机就可以访问驻留在远程服务器上的文件。事实上，这些客户机可以是**无盘化**的，依靠服务器提供所有的文件，包括操作系统内核，但需要特殊的协议来引导程序。考虑一下无盘工作站获取内核的问题。无盘工作站没有内核，所以它不能用 DFS 代码来获得内核，而是调用一个存储在客户机的只读存储器（ROM）上的特殊引导协议，使其初始化网络，并从一个固定位置获得一个特定的文件（内核或引导代码）。一旦内核通过网络被拷贝过来并加载，它的 DFS 使得所有其他的操作系统文件都有效。无盘化客户机的优点很多，包括低价格（由于每台机器不需磁盘）和方便性（当一个操作系统更新后，只需修改服务器，而不需改变所有的客户机）。它的缺点在于增加了引导协议的复杂性，及由于使用网络而不是本地磁盘而引起的性能下降。

现在的流行趋势是使用具有本地磁盘的客户机。磁盘驱动器在容量上快速增长，而价格在下降，每年都出现新一代的产品。但网络就并非如此，它每一代需要发展 5 年至 10 年。总而言之，系统比网络增长得更快，因此有必要限制网络访问，从而改善系统吞吐量。

16.2.2 命名方案

在DFS中主要有三种命名设计方法。最简单的方法是结合主机名和本地名以对文件进行命名，它保证了在整个系统范围中只有惟一的名字。例如，在Ibis中，文件用"*主机：本地名*"(*host:local-name*)来惟一地标识文件，其中本地名(*local name*)是类似于UNIX的路径。这种命名方法既不是位置透明的，也不是位置独立的。不过，无论是本地或远程文件，文件操作都是相同的。DFS被构造为孤立的部件单元的集合，这些部件单元完全是传统的文件系统。在此第一种方法中，部件单元仍然是孤立的，尽管它提供了访问远程文件的方法。本书中不再进一步考虑这种设计方案。

第二种方法在Sun公司的网络文件系统(NFS)中得到广泛应用。NFS是ONC+的文件系统组成部件，ONC+是很多UNIX厂商支持的网络包。NFS提供将远程目录添加到本地目录的方法，使其目录树看起来一致。早期的NFS版本只允许先前已被加载的远程目录被透明地访问。随着**自动加载**的出现，加载可根据需要进行，它基于一个加载点表和一些文件结构名。因为每台机器可以附加不同的远程目录到它的目录树上，所以尽管这种整合是有限且不统一的，部件还是被整合在一起以支持透明共享，所得的结构是通用的。通常，它是具有共享子树的UNIX树集。

第三种方法，能够得到组件文件系统的完整集成。系统中的所有文件都使用单个的全局名字结构。理想的情况是，这种复杂的文件系统结构与传统的文件系统是**同构**(isomorphic)的。而实际上，许多特殊的文件(例如，UNIX设备文件和特别的机器二进制目录)很难达到此目的。为了评价这些命名结构，可以考虑它们的**管理复杂度**。最复杂且最难维护的结构是NFS结构，因为任何远程目录都能附加到本地目录树的任何地方，所产生的系统是毫无结构性可言的。如果一个服务器出错而不能用，一些在不同机器上的任意目录集也会变得不可用。此外，由于有一个单独的信任机制控制着哪台机器可以附加哪个目录到它的目录树上，用户可能可以在一个客户机上访问一个远程目录树，但不能在另一客户机上进行。

16.2.3 实现技术

透明性命名的实现需要提供从文件名到相关位置映射的能力。为保持映射可管理，必须将文件集聚集到部件单元中，并在一个部件单元的基础上提供映射，而不是在单个文件的基础上进行操作。此聚集也适用于管理目的。类似于UNIX的系统使用等级目录树来提供名字—位置的映射，并将文件递归地聚集到目录中。

为了提高关键映射信息的可用性，可以采用复制、本地缓存的方法。正如本书所讲，位置独立性意味着映射会随时间而改变，因此，复制映射虽然简单，却导致信息不可能一致更新。解决的方法是引进底层的、**位置独立文件标识**(location-independent file identifier)。文本文件名被映射到低层的文件标识，该标识表明了文件属于哪个部件单元。这些标识仍然

是位置独立的，它们能自由地被复制和存储，不需要移动部件单元使其无效。当然，这不可避免地需要一种简单却能保持更新一致的二级映射来将部件单元映射到具体位置上。利用这种底层的、位置独立的标识来实现类似 UNIX 的目录，使整个层在部件单元移动时不变，改变的只是部件单元位置的映射。

实现这些低层标识的常用方法是使用结构化的名字，这些名字是位串形式的，通常有两个部分。第一部分表明了文件属于哪个部件单元，第二部分表明了单元中特指的文件。可能会存在具有更多的部分的变体。结构化名字的不变性在于，名字的各个部分仅在其余部分的上下文环境中总是惟一的。通过避免重用仍在使用的名字，或增加足够多的位（此方法用于 NFS），或通过使用时间戳作为其名字的一部分（如在 Apollo 中所用），可以获得所有时间内的惟一性。另一种方法是运用位置透明性系统，如 Ibis，通过再增加一个层次的抽象来产生一个位置独立的命名方法。

将文件聚集在部件单元中以及低层的位置独立文件标识符的应用在 AFS 中得以例证。

16.3 远程文件访问

现有一个用户请求访问远程文件。假设存储该文件的服务器用命名方案来定位，就必须产生满足用户远程访问请求的实际数据传输。

完成此传输的方法之一是通过一个**远程服务机制**（remote-service mechanism），由此，访问请求被送到服务器，服务器完成此访问，产生的结果被转送回用户。执行远程服务的最常用的方法之一是第四章中所讲的远程过程调用（RPC）。传统文件系统中的磁盘访问方法和 DFS 中的远程服务方法之间很相似：使用远程服务方法类似于对每个访问请求完成一次磁盘访问。

可以用高速缓存方式来保证远程服务机制所期待的性能。在传统的文件系统中，缓存的基本原理在于减少磁盘的 I/O（从而提高性能），而在 DFS 中，目的在于既要减少网络通信量也要减少磁盘 I/O。下面讨论在 DFS 中缓存的实现以及与基本的远程服务的对比。

16.3.1 基本的缓存设计

高速缓存的概念很简单。如果满足访问请求所需的数据尚未缓存，则这些数据的一个拷贝从服务器传到客户机系统，访问在缓存的拷贝上完成。此思想是在缓存中保留最近访问的磁盘块，从而使对同样信息的重复访问可以本地化处理，不再需要额外的网络通信。可以采用某种替换策略（如最少最近使用）来保持缓存大小。在对服务器的访问和通信之间没有直接的通信。文件仍用存储在服务器上的一个主拷贝来标识，但文件的拷贝（或文件的部分）被分散到不同的缓存中。当一个存储的拷贝被更改后，需要考虑到主拷贝的改变，以保持相关的语义一致性。如何保持缓存拷贝与主拷贝之间的一致性是 16.3.4 小节所要讨论

的**缓存一致性问题**。DFS的高速缓存可像**网络虚拟存储器**那样调用：它类似于按需分页的虚拟内存，不过备份存储通常不是本地磁盘，而是远程服务器。

DFS中缓存的数据粒度可以从文件的若干块到整个文件之间变化。通常，对于单个访问，存储的数据多于所需的数据，这样可以使较多的访问通过缓存的数据来完成。这个过程很像磁盘先读(12.6.2小节所述)。AFS将文件存储在大存储块中(64 KB)。本章提到的其他系统支持根据客户需求缓存单个存储块。增加存储单元就增加命中率，但也增加了失效代价，因为每个失效都需要传输更多的数据，同时也增加了潜在的一致性问题。选择存储单元涉及到考虑诸如网络传输单元和RPC协议服务单元(如果用到RPC协议)之类的参数。网络传输单元(对以太网而言，是一个分组)大约有1.5 KB，因此大的存储数据单元需要分组后进行传输，然后在接收方进行重新集成。

块的大小和整个缓存的大小对于块缓存设计非常重要。在UNIX系统中，块的大小一般为4 KB或8 KB。而对于大的缓存(如1 MB)，大一点的块(大于8 KB)更有利。但对于小的缓存来说，由于大块易导致缓存中只有很少的块，从而产生较低的命中率，故大的块对于小的缓存无益。

16.3.2 缓存位置

缓存的数据应存放在何处呢？磁盘缓存比主存储器具备一个明显的优点：它们是可靠的。如果缓存是易变的，存储数据的更改将在系统崩溃中丢失。然而，如果缓存数据存储在磁盘上，在恢复期间它们仍将保留在磁盘上，不过那时已没有必要再来读取它们了。当然，另一方面，主存储缓存具有自身的几个优点：

- 主存储缓存允许工作站无盘化。
- 从主存储缓存中可以比从磁盘缓存上更快地访问数据。
- 目前的技术趋势是朝着更大、更便宜的主存储器发展，获得的性能加速将会超过磁盘缓存的优点。
- 不管用户缓存位于何处，服务器缓存(用于加快磁盘I/O)将在主存储器中。如果在用户机器上也用主存储缓存，可以为服务器和用户建立一个单缓存机制。

许多远程访问的实现可被视为缓存和远程服务的混合。例如，在NFS中，该实现基于远程服务，但为提高性能将客户机方和服务器方的存储器缓存加大。另一方面，Sprite的实现基于缓存，但在某些环境中采用远程服务方法。因此，为了评价这两种方法，将评价每种方法被强调的程度。

NFS协议和大多数实现都不提供磁盘缓存。最近Solaris的NFS实现(Solaris 2.6及更高版本)包括一个客户机磁盘缓存可选项，即*cachefs*文件系统。一旦客户机从服务器上读取文件块，它便将它们存储在内存和磁盘上。如果内存拷贝被刷新，或者即使系统重启，就访问磁盘缓存。如果所需的文件块既不在内存中，也不在*cachefs*磁盘缓存上，一个RPC

被送到服务器，以重新得到文件块，且该块被写入磁盘缓存以及客户机所用的内存缓存中。

16.3.3 缓存更新策略

用于将更改的数据块写回服务器主拷贝的策略对系统的性能和可靠性具有关键性的影响。最简单的方法是，一旦有数据被放置在缓存中，就将它们写到磁盘上。**直写策略**(write-through policy)是可靠的：当一个客户机系统崩溃后，几乎没有数据丢失。然而，此方法需要每次写访问等待，直到信息被送到服务器，所以它导致较差的写性能。直写的缓存等同于使用远程服务来写访问，及利用缓存进行只读访问。

另一种方法是**延迟写策略**(delayed-write policy)，它对主拷贝的更新是延迟的。更新被写到缓存，稍后才被写到服务器。相对于直写方法，此方法有两个优点。第一，由于是写到缓存，所以写访问完成得更快。第二，数据可在被写回之前被重写，此时只有最后的更新需要写。不足的是，只要用户机崩溃，未写的数据就会丢失，所以延迟写方法存在可靠性问题。

由于更改的数据块被刷新到服务器的时机不同，延迟写方法还有一些变种。一种选择是，当一个数据块从客户机缓存中被逐出后刷新它。此选择可能会得到较好的性能，但有些数据块在写回服务器之前可能会在客户机的缓存上停留很长一段时间。将此方法与直写方法折中，在固定的时间间隔扫描缓存，刷新自最近扫描后已被修改的数据块，就像 UNIX 扫描它的本地缓存一样。Sprite 使用此方法时的时间间隔为 30 s。NFS 也使用此方法，但在刷新缓存期间，一旦一次写操作被提交给服务器，那么写入必须在它被认为完成之前到达服务器的磁盘。NFS 用不同的方式处理元数据(目录数据和文件属性数据)，任何一种元数据的改变都同步提交给服务器。因此，这可以使得当客户机或服务器崩溃时避免文件丢失和目录结构恶化。

对具有 *cachefs* 的 NFS 而言，为保持所有拷贝一致而在写入服务器的同时也要写到本地磁盘的缓存区。因此，具有 *cachefs* 的 NFS 在对 *cachefs* 缓存命中的读请求情况下，性能改善要优于普通的 NFS；但对于存在缓存失效的读或写请求，则性能要差。对所有的缓存来讲，具有高缓存命中率以获取性能是非常重要的。

然而，延迟写的另一种变化是在文件关闭时将数据写回服务器。AFS 使用这种**写关闭策略**(write-on-close policy)。在文件打开时间很短或很少被修改的情况下，此方法并未有效地减轻网络通信。另外，写关闭方法在文件被直写时需要关闭进程来延迟，从而降低了延迟写方法的性能优势。对于要打开很长时间或修改频繁的文件，如果延迟写需要经常刷新，该方法的性能显然优于前者。

16.3.4 一致性

客户机需要判断一个数据的本地缓存拷贝与主拷贝(然后才能使用)是否一致的问题。如果客户机确定它的缓存数据已过时，那么这些缓存数据就不能再提供数据访问服务，而需

要缓存该数据的一个最新拷贝。下面两种方法用来验证缓存数据的有效性：

1. **客户机发起的方法**(client initiated approach)：客户机发起一次有效性检查，它与服务器联系，并检查本地数据与主拷贝是否一致。有效性检查的频率是此方法的关键，并决定其所产生的一致性语义。频率范围可以从每次访问前都进行一次检查到只对一个文件的第一次访问进行一次检查(主要是在文件打开的时候)。相对于访问立即被缓存服务而言，伴随有效性检查的访问就被延迟了。作为选择，可在某个固定的时间间隔开始一次检查。有效性检查既可装载在网络上，也可装载在服务器上，这取决于它的频率。

2. **服务器发起的方法**(server-initiated approach)：服务器为每个客户机记录它缓存的文件(或文件的一部分)。当服务器检测到一个潜在的不一致时，它必须有所反应。当两个不同的处于竞争状态的客户机缓存一个文件时，就会发生潜在的不一致性。如果实现UNIX语义(11.5.3小节)，可以通过让服务器扮演一个积极的角色来解决这个潜在的不一致性。无论文件何时被打开，服务器都必须被告知，包括每一次打开后将要采取的模式(读或写)。根据通知，当服务器检测到一个文件在冲突状态下被同时打开，那么服务器使此文件的缓存失效。实际上，使缓存失效将导致转换到一个远程服务操作模式。

16.3.5 高速缓存和远程服务的对比

本质上，在缓存和远程服务之间的选择存在一个潜在的性能加强与简易性降低的矛盾，以下列举这两种方法的优、缺点来做一比较：

• 当使用缓存时，本地缓存能有效地处理许多远程访问。在文件访问模式中利用局部性原理使得缓存性能更好。因此，大多数远程访问就像在本地进行一样快捷。更进一步，只是偶尔与服务器连接，而不是每次访问都需如此，从而减少了服务器负载和网络通信，并且扩充的潜力也得到了提高。相反地，当使用远程服务方法时，每次远程访问都要跨越网络进行，结果可能产生网络阻塞、服务器超载，性能会如何也就显而易见了。

• 网络总开销在传输大块数据时(就像在缓存中那样)要比对个别请求的一系列响应传输时(就像在远程服务方法那样)低。而且，如果服务器知道请求总是针对大的、连续的数据段，而不是对随机的数据块，那么服务器上的磁盘访问程序可以被更好地优化。

• Cache一致性问题是缓存技术的主要困难。对不常写入的访问模式，缓存技术是很有效的。但当写入频繁时，用来克服一致性问题的机制反而导致了大量诸如性能、网络流量以及服务器负荷的开销。

• 因此缓存技术应在有本地磁盘或大主存的机器上实现，从而得益于此。在无盘、小容量存储器的机器上的远程访问应通过远程服务方法进行。

• 在缓存技术中，由于数据在服务器和客户机之间整体传输，而不是响应一个特殊的文件操作需求，因此机器间的界面不同于上级的用户界面。另一方面，远程服务方式不过是本地文件系统界面在网络上的扩展，因此，机器间的界面反映了本地用户文件系统界面。

16.4 有状态服务和无状态服务

对服务器端的信息处理有两种方法：一是服务器跟踪被每个客户机所访问的每个文件，二是服务器不必了解数据块的用途而直接提供客户机请求的数据块。

有状态的文件服务(stateful file service)的典型方案如下：客户机在访问一个文件之前必须完成一个打开操作。服务器从其磁盘上取出有关文件的信息并存入内存，将一个对客户机和文件而言惟一的连接标识号提供给客户机。(在 UNIX 系统中，服务器取出 inode，并提供给客户机一个文件描述符，它可用做 inode 的内部核心表的索引。)此标识号也被用于以后的访问，直到会话结束。有状态服务的特征体现在会话期间客户机和服务器之间的连接。无论是关闭文件，还是用一个垃圾收集机制，服务器必须收回曾被客户机使用的不再有效的主存空间。在一个有状态服务方法中容错的关键点在于服务器保存了客户机在其主存上的活动信息。AFS 是一个有状态的服务。

无状态的文件服务器(stateless file server)通过使每个请求自立而避免状态信息。即每个请求完全地标识文件和在文件中的位置(为了读和写访问)。服务器不需要在主存中维持一个打开文件列表，尽管它常常为了提高效率而这样做。此外，也没有必要通过打开和关闭操作来建立或终止一个连接。它们都是多余的，因为每个文件操作都是自立的，并不被视为会话的一部分。一个客户机进程可能会打开一个文件，但此打开动作不会引起一个远程消息的发送。读和写将作为远程消息发生(或缓存的查找)。最后由客户端产生的关闭操作将再次只引发一个本地操作。NFS 是一个无状态文件服务。

相对于无状态的服务，有状态的服务的优点在于增强了性能。文件信息在主存中得到缓存，可以通过连接标识符很容易地访问到，故节省了磁盘存取。此外，有状态的服务知道一个文件是否对随后的访问打开，因而可事先读下一个块。无状态的服务就不能这样了，因为它不知道客户机请求的目的。

当考虑在一次服务活动中发生崩溃时，有状态服务和无状态服务的区别变得更加明显。有状态服务器在崩溃中丢失它所有易失的状态。此服务器的完全恢复涉及到修复该状态，通常用一个基于与客户机对话的恢复协议来完成。不完全恢复则需要在崩溃发生时停止正在进行中的操作。另一个问题是由客户机出错引起的。服务器需要知道这些错误，以收回已分配的用于记录发生崩溃的客户机进程状态的空间。此现象有时被称为 **orphan 检测和排除**。

无状态的服务则避免了这些问题，因为一个新的服务器能毫无困难地对一个自主的请求做出响应。因此，服务器故障和恢复的影响几乎不受注意。在客户机看来，一个慢的服务器和一个正在恢复的服务器是没有区别的。如果未收到响应，客户机继续重新发送它的请求。

使用健壮的无状态服务的代价是较长的请求报文和较慢的请求处理，因为没有核心中的信息来加快处理。此外，无状态的服务不得不在 DFS 的设计上增加附加的约束。首先，由于每个请求都标识目标文件，需要使用一个统一的、系统范围内的、低层的命名方法。对每个请求进行远程到本地的名字转换将使得对请求的处理更慢。其次，由于客户机重新发送文件操作请求，这些操作必须是幂等的，即如果连续执行几次，每个操作效果相同且返回同样的结果。自主性的读和写访问是幂等的，只要它们使用完全字节计数来表明它们所访问的文件中的位置，而不依靠偏移量(就像在 UNIX 中的读和写系统调用那样)。然而，必须注意，在实现破坏性操作(如删除文件)时，也应该使这些操作是幂等的。

在某些环境中，必须使用有状态的服务。如果服务器采用服务器发起的缓存确认方法，因为它要维护一个记录来表示某个文件被某个客户机缓存，就无法提供无状态的服务。

UNIX 使用文件描述符和隐式的偏移量，因此是有状态的服务。服务器必须维护一张从文件描述符到 inode 的映射表，且必须保存文件中的当前偏移量。这也就是为什么使用无状态服务的 NFS 不使用文件描述符并在每次访问中包括一个显式的偏移量。

16.5 文件复制

不同机器上的复制文件对于提高有效性而言是有用的冗余。多机器复制也有益于性能：选择邻近的一个复制品来服务一个访问请求将使服务时间更短。

复制设计的基本条件是同一文件的不同复本应驻留在彼此故障独立的机器上。即一个复本的有效性不受其余复本有效性的影响。此条件意味着复制管理是位置非透明的。必须提供一种机制来将一个复本放在一个特定的机器上。

对用户隐藏复制的细节是必要的。将一个复制的文件名映射到一个特定的复本上是命名设计的任务。复本的存在对高层是不可见的。在低层，复本必须用不同的更低层的名字区分开来。另一个透明性条件是需要在高层提供复制控制。复制控制包括复制程度和复本放置的确定。在某些环境下，可能希望将这些细节暴露给用户。例如，Locus 提供给用户和系统管理员控制复制机制。

与复制相关的主要问题是它们的更新。从用户的角度来看，一个文件的复本表示的是同样的逻辑实体，因此对任何复本的更新应影响到所有其他的复本。更明确地讲，当对文件复本的访问被视为对复本的逻辑文件的虚拟访问时，必须保持相关的一致性语义。如果一致性并不是最主要的，那么就不得不在有效性和其他性能方面做出牺牲。在容错范围内的权衡下，有两种选择，要么选择不惜任何代价保持一致性，从而产生不确定的数据块的可能性，要么选择在某些(希望是很少)灾难性错误的情况下为了保证系统的继续运行而牺牲一致性。例如，Locus 广泛地使用复本，在网络中断时为了保证文件读写操作的有效而牺牲了一致性。

Ibis 使用的是主拷贝方法的一种变体，名字映射域为一对<*主复本标识，本地复本标识*>（<*primary-replica-identifier*, *local-replica-identifier*>）。如果不存在本地复本，则使用一个特殊值，因此映射是相对于机器的。如果本地复本是主复本，则标识对中包含两个相同的标识。Ibis 支持按需复制，这是一个类似于整体文件缓存的自动复本控制方法。使用此方法，读取一个非本地复本就使其被本地缓存，从而产生一个新的非主复本。更新只在主拷贝上完成，并通过发送适当的消息使所有其他复本变为不可用。但这无法保证所有非主复本的原子性和连续无效性。因此，一个旧的复本有可能被认为是有效的。为了满足远程写访问，可以将主拷贝移植到发出请求的机器上。

16.6 一个实例：AFS

Andrew 是美国卡耐基－梅隆大学(Carnegie Mellon University)在 1983 年开始设计并实现的分布式计算环境。Andrew 文件系统(AFS)在其环境中的客户机上建立了基本的信息共享机制。Transarc 公司继续发展了 AFS，之后被 IBM 公司收购。IBM 公司由此生产了一些 AFS 商业工具。后来 AFS 被选做与工业联合的 DFS，产生了 *Transarc* DFS，它是 OSF 组织的分布式计算环境(DCE)的一部分。

2000 年，IBM 公司的 Transarc 实验室宣布 AFS 将成为一个在 IBM 公司公共许可证下的开放源代码产品。Transarc DFS 继而发展为一个商用产品。许多 UNIX 厂商以及微软公司都宣布支持 DCE 系统。研究继续进行，以使其成为一个跨平台、能被普遍接受的 DFS。AFS 和 Transarc DFS 非常相似，所以在此讨论 AFS，除非特别提到 Transarc DFS。

AFS 试图寻求解决简单的 DFS(如 NFS)问题的方法，它被认为是最有特色的非实验性的 DFS。它的特点体现在统一的名字空间、位置独立的文件共享、Cache 一致性的客户端缓存技术和通过 Kerberos 的安全认证。它还包括用复本形式的服务器端缓存技术，及在源服务不能使用时通过复本自动替换来获得高有效性。AFS 最强大的特点在于它的容量：Andrew 系统可跨越超过 5 000 个工作站。在全世界，介于 AFS 和 Transarc DFS 之间有成百的实现系统。

16.6.1 概述

AFS 对待*客户机*(有时指的是*工作站*)和专用*服务器*是不同的。本来服务器和客户机只能运行 4.2BSD UNIX，但 AFS 已经移植到了许多操作系统上。客户机和服务器通过 LAN 和 WAN 相连接。

客户机用分空间的文件名来表示：一个**本地名字空间**和一个**共享名字空间**。专用服务器，通常的称法是在它们所运行的软件名后加*副*(*Vice*)字，将名字空间以一种同质、相似、位置透明的文件层次的形式呈现给客户机。本地名字空间是一个工作站的根文件系统，共享

的名字空间由此传下去。工作站运行 *Virtue* 协议来与 *Vice* 通信,并被要求用本地磁盘来存储它们的本地名字空间。服务器共同负责共享名字空间的存储和管理。本地名字空间是很小的,对每个工作站都有所不同,它包括自治操作和优秀性能所必须的系统程序。同样,本地名字空间是临时文件或是工作站所拥有的文件,这些文件为隐蔽起见,明确地希望存储在本地。

从更细的粒度来看,客户机和服务器是用 WAN 相互连接的簇结构,每一簇由 LAN 上的一组工作站构成,Vice 的一个代表被称为**簇服务器**(cluster server),通过路由器与 WAN 相连。分解成簇的目的主要在于解决规模问题。为了得到优异的性能,工作站应在大部分时间内使用本簇上的服务器,从而相对地减少跨簇的文件访问。

文件系统体系结构设计也基于规模的考虑。基本的探索方式是服务器将工作卸载给客户机,因为经验表明服务器的 CPU 速度是系统的瓶颈。按此方式,远程文件操作所采用的关键机制是使用大数据块(64 KB)形式的缓存文件,这种方式减少了文件打开的等待延迟,使得读写操作被定向到对缓存拷贝的操作,而不需再频繁地访问服务器。

简要地,下面是 AFS 设计中几个附加的问题:

- **客户机灵活性**:客户能从任何工作站访问位于共享名字空间中的任何文件。当从非常用工作站来访问文件时,可能会出现由于需要缓存文件而导致的最初性能下降。
- **安全性**:由于没有客户机程序在 Vice 机上执行,所以 Vice 接口被认为是可靠性的边界。认证和安全传输的功能是作为基于连接的通信包的一部分来提供的,它们基于 RPC 实现。经过互斥认证后,Vice 服务器和客户机通过加密消息进行通信,加密过程通过硬件或软件(更慢)来完成。客户机和组的信息保存在一个受保护的数据库中,这个数据库在每个服务器都有备份。
- **保护**:AFS 提供了**存取列表**来保护目录和常规的 UNIX 文件。此存取列表包含允许以及不允许访问某个目录的用户信息。因此通过这种设计,可以很容易指定,如除了 Jim 外所有人都可以访问某目录。AFS 支持的访问类型有读、写、查询、插入、管理、锁定和删除。
- **异质性**:对 Vice 定义一个清晰的接口是整合各种不同的工作站硬件和操作系统的关键,因此异质性是有益的,比如本地/bin 目录下的某些文件是指向驻留在 Vice 上的机器特定的可执行文件的符号连接。

16.6.2 共享名字空间

AFS 的共享名字空间由称为**卷**(volume)的单元组成,AFS 的卷通常是很小的部件单元。一般来说,它们与单个客户机上的文件相关。很少有卷驻留在单个磁盘分区上,它们可以在大小上缩放(到一定的限度)。从概念上讲,卷通过一个类似于 UNIX 安装机制的方法聚合在一起。当然,二者的粒度差别是很大的,这是由于在 UNIX 只能安装一个完整的磁盘分区(含有文件系统)。卷是一个重要的管理单元,在识别和定位一个文件时作用很大。

Vice 文件或目录用一个称为 fid 的低层标识符来识别。每个 AFS 目录项把一个名字组件映射为一个 fid。一个 fid 有 94 bit 长，它有三个等长的部分：一个卷号、一个 v 节点(*vnod*)号和一个*惟一号*(*uniquifier*)。**vnod** 号被用做一个数组的索引，该数组包含单个卷中所有文件的索引节点(inode)。**惟一号**允许 v 节点号的重用，从而保持数据结构的紧凑。fid 是位置透明的，因此，文件从服务器到服务器的移动不会使缓存的目录内容无效。

位置信息保存在一个**卷位置数据库**的卷基上，这个数据库在每个服务器上得到备份。客户机可以通过查询此数据库识别每个卷在系统中的位置。把文件集合成卷使得数据库的大小保持在一个合适的范围内。

为了平衡可用磁盘空间和服务器的利用率，卷需要在磁盘分区和服务器之间移动。当一个卷被送到它的新位置，它的原始服务器留下了临时的查找信息，故位置数据库不需要同步更新。在此卷被传送期间，原来的服务器仍能处理更新，随后送到新的服务器上。在某些方面卷暂时不可用。然后，新的卷在新的站点上再次可用。卷移动操作是原子操作，无论哪个服务器崩溃，操作都会被放弃。

整卷粒度大小的只读复制用于支持 Vice 名字空间中较高层中的系统可执行文件和不常更新的文件。卷位置数据库指定了包含一个卷的一个读写拷贝的服务器和一系列只读复制站点。

16.6.3 文件操作和一致性语义

AFS 的基本体系原则是整体地缓冲来自服务器的文件。因此，一个客户端工作站只在打开和关闭文件期间与 Vice 服务器相互作用，甚至此交互作用并不总是必需的。读和写文件不会引起远程交互(与远程服务方法相反)。关键的区别在于性能上有很大的差别，对文件操作语义也如此。

各个工作站的操作系统截取文件系统调用，并将它们交给其上的一个客户级的进程。这个被称为 *Venus* 的进程缓存来自于 Vice 的文件(当它们被打开时)，并在它们被关闭时，将修改的文件副本存回它们原来所在的服务器。Venus 可能只在文件打开或关闭时与 Vice 联系，一个文件的个别字节的读和写将会绕开 Venus，直接在存储的拷贝上完成。这样产生的结果是，在某些站点上的写不能为另一些站点立即可见。

高速缓存为将来打开缓存文件而被充分利用。Venus 假定缓存项(文件或目录)是有效的，除非另有说明。因此，当一个文件打开以使存储的副本生效时，Venus 不需要与 Vice 联系。有一种被称为**回调**(callback)的支持此方法的机制可以显著地减少服务器接收的缓存确认请求数量。它按如下方式工作：当一个客户机存储一个文件或目录时，服务器更新用于记录此存储的状态信息，称该客户机对该文件有一次回调。服务器在允许另一客户机修改此文件之前通知该客户机。此时，假设服务器移除前一个客户机的文件上的回调。只有当文件具有一次回调时，客户机才可以使用缓存的文件来打开文件。如果客户机在修改一个

文件之后关闭它,所有存储此文件的其他客户机则丧失它们的回调。因此,当这些客户机随后打开该文件时,它们不得不从服务器那里获取一个新的版本。

文件的读和写直接通过内核完成,而不需要 Venus 对存储的副本的干预。当文件被关闭时,Venus 重获控制,并且如果文件已被局部修改,那么它在相应的服务器上更新此文件。因此,只有在服务器打开那些不在缓存区或丧失它们的回调以及关闭局部修改的文件时,Venus 和 Vice 才有接触的机会。

AFS 主要是执行会话语义。仅有的例外是除了简单的读和写外的文件操作(如目录级的保护变化),操作完成之后在网络上立即随处可见到它们。

尽管采用回调机制,仍然可能出现少量的缓存确认通信,通常用来代替由于机器或网络故障引起的回调丢失。当一个工作站被重新启动时,Venus 怀疑所有的缓存文件和目录,然后它为所有这种记录的第一次使用产生一个缓冲区确认请求。

回调机制强制每个服务器维护回调信息,每个客户机维护有效性信息。如果服务器维护的回调信息量过多,服务器可以暂停回调,并通过单方面通知客户机以及解除它们缓存文件的有效性来收回一些存储空间。如果 Venus 维护的回调状态与服务器维护的相应状态相比,变得失去同步,可能会导致某些不一致性。

为了解释路径一名字,Venus 还存储目录内容和符号连接。路径名中的每一部分都可获取,如果它还未被存储,或者如果客户机没有对它的回调,则可以为之建立一个回调。查找通过 Venus 在获取的使用 fids 的目录来进行。没有请求从一个服务器被送到另一个服务器。在一个路径一名字遍历的结束,所有的中间目录和目标文件都被存储到具有回调的缓存区中。将来对此文件的打开调用根本不需要涉及网络通信,除非在此路径名字的部分上回调被破坏。

缓存策略仅有的例外是对目录的修改,为了统一,此目录直接在负责该目录的服务器上生成。对此问题,Vice 接口具有很好定义的操作。Venus 在它的缓存拷贝上反映了变化,以避免再次取此目录。

16.6.4 实现

客户机进程与一个具有常用系统调用集的 UNIX 内核相衔接。内核被少许修改,以检测对相应操作中 Vice 文件的访问,并将请求转给工作站上客户机级的 Venus 进程。

Venus 如前所述,一个组件接着一个组件地实现路径一名字解释。它有一个联系卷与服务器位置的映射缓存,以避免服务器查询一个已经知道的卷位置。如果某个卷未在此缓存区出现,Venus 与任何已经建立连接的服务器联系,请求位置信息,并将此信息加入映射缓存区中。除非 Venus 与服务器之间已经存在连接,否则它将建立一个新的连接,然后利用此连接获取文件或目录。建立连接对身份认证和安全是必要的。当一个目标文件被发现和存储后,本地磁盘上产生一个副本。然后 Venus 返回到内核,此内核打开缓存的副本并返回

它对客户机进程的处理。

UNIX 文件系统被 AFS 的服务器和客户机用做低层存储系统。客户机缓存是一个工作站磁盘上的本地目录。此目录中包括的是文件，它们的名字是缓存进入的占位符。Venus 和服务器进程通过后者的索引节点(inode)直接访问 UNIX 文件，以避免昂贵的路径名到索引节点的转换程序。由于内部的索引节点接口对客户机级进程是不可见的(Venus 和服务器进程都是客户机级进程)，需要增加一个相应的附加的系统调用。DFS 使用它自己的日志文件系统来改善性能和 UFS 上的可靠性。

Venus 管理了两个不同的缓存：一个是关于状态的，另一个是关于数据的。它使用简单的最近最少使用(LRU)算法来保持它们中每一个的大小限制。当一个文件从缓存中被刷新，Venus 通知相应的服务器移除对此文件的回调。状态缓存被维持在虚拟存储器中，以允许 stat(文件状态返回)系统调用的快速服务。数据缓存驻留在本地磁盘，但 UNIX 的 I/O 缓冲机制在内存中实行一些对 Venus 透明的磁盘区缓存。

每个文件服务器上的单个客户机级进程对客户机的所有请求进行服务。此进程使用一个非占先的轻量级进程包来并发地为许多客户请求服务。RPC 包与轻量级进程包整合，从而允许文件服务器并发地为每个轻量级进程生成或服务一个 RPC。RPC 建立在低层的数据报抽象顶端，整个文件传输被作为这些 RPC 调用的副作用来实现。每个客户机存在一个 RPC，但对这些连接没有一个轻量级进程的优先绑定。取而代之的是，一个轻量级进程库在所有的连接上为客户机请求服务。单个多线程服务器进程的使用允许存储对服务请求所需的数据结构。另一方面，单个服务器进程的崩溃存在是此特定服务器瘫痪的惨重的后果。

16.7 小 结

DFS 是这样一个文件服务系统，它的客户机、服务器和存储设备都分散在一个分布式系统的站点上。服务活动必须相应地在网络上进行；有多个独立的存储设备取代了单个集中式数据仓库。

理想地，DFS 应将它的客户机视为一个传统的集中式文件系统。它的服务器和存储设备的多样性和分散性应该透明，即 DFS 的客户机接口对远程和本地文件应无区别。DFS 负责确定文件位置和安排数据的传送。一个透明的 DFS 通过将客户机环境变换到客户机所登录的那个站点而方便了客户机的移动性。

在 DFS 中有几种命名设计方法。最简单的方法是，文件用它们的主机名和本地名的组合来命名，这保证了系统范围内名字的惟一性。另一种方法在 NFS 中广泛使用，它提供一种方法，用于将远程目录附加到本地目录上，以提供相同的目录树。

访问远程文件的请求通常由两种互补的方法来处理。使用远程服务方法，访问请求被传送到服务器，服务器执行访问，并将结果返回给客户机。使用缓存技术，如果满足访问请

求所需要的数据尚未在缓存中，就将这些数据的一个拷贝从服务器送到客户机，访问在缓存的拷贝上执行。这种思想是在高速缓存中保留最近被访问的磁盘块，使得对相同信息的重复访问能在本地进行，而不再需要额外的网络通信。一种替换策略被用来保证高速缓存的大小，保持缓存拷贝与主文件一致属于缓存一致性问题。

处理服务器方信息有两种方法。或是服务器跟踪客户机访问的每个文件，或在客户机发出请求但不需要它们的用途信息时，简单地提供数据块。二者分别是有状态服务和无状态服务。

在不同机器上的文件拷贝是改善有效性的一个有用冗余。由于选择一个近的复本来为一个访问请求进行服务将花费更少的服务时间，多机拷贝也有益于性能改善。

AFS 是一个更有特色的 DFS。它是 Transarc DFS 开放源码的先例。它的特征在于位置独立性和位置透明性，还加以大量的语义一致性。缓存和拷贝被用来改善性能。

习题十六

16.1 DFS 与集中式系统中的文件系统相比有哪些优点？

16.2 本章中所讨论的 DFS 例子中哪一个对于处理大的、多客户的数据库应用最有效？为什么？

16.3 在什么情况下客户喜欢一个位置透明的 DFS？在什么情况下客户喜欢位置独立的 DFS？解释为什么客户会有这些偏好？

16.4 对于一个在完全可靠的网络上运行的系统，你会选择一个什么样的分布式系统？

16.5 对比和比较在客户端系统本地运行，以及在服务器上远程运行的磁盘块高速缓存技术。

16.6 像 Apollo Domain 那样，将对象映射到虚拟内存中有哪些好处？有哪些缺陷？

16.7 描述 AFS 和 NFS 之间的基本不同点（第十二章）。

推荐读物

Davcev 和 Burkhard[1985] 描述了关于备份文件的一致性和恢复控制。Brereton[1986] 和 Purdin 等[1987] 描述了 UNIX 环境中的备份文件管理。Wah[1984] 论述了分布式计算机系统中的文件布局问题。Svobodova[1984] 里给出了关于集中式文件服务器的一个详细综述。

Callaghan[2000] 展示了 Sun 的网络文件系统（NFS）。Morris 等[1986]、Howard 等[1988] 和 Satyanarayanan[1990] 论述了 AFS 系统。关于 Transarc AFS 和 DFS 的信息可以在 http://www.transarc.ibm.com/DFS 中找到。

还有很多有趣的 DFS 在本书中没有详细论述，包括 UNIX United、Sprite 和 Locus。Brownbridge 等[1982] 描述了 UNIX United。Popek 和 Walker[1985] 论述了 Locus 系统。Ousterhout 等[1988] 和 Nelson 等[1988] 描述了 Sprite 系统。

第十七章　分布式协调

第七章描述了允许进程同步工作的各种机制，还讨论了在事务独立执行或与其他事务并发执行时，保证事务原子属性的方法。第八章描述了操作系统可用来处理死锁问题的多种方法。这一章将研究如何把集中式同步机制扩展到分布式环境中，及在分布式系统中处理死锁的方法。

17.1　事件排序

在集中式系统中，由于系统只有单个公共存储器和时钟，因而总能确定两个事件发生的顺序。有许多应用需要人们确定顺序，如在一个资源分配方案中，指定一个资源必须在得到它的许可后才能被使用。但在分布式系统中，由于没有公共的存储器，也没有公共的时钟，因此，有时不能判断两个事件谁先发生。在分布式系统中，事前(*happened-before*)关系仅仅是事件的一个部分排序关系。由于决定整体排序的能力在许多应用中非常关键，因此提出一种分布式算法来将事前关系扩充为系统中的所有事件的一致性整体排序。

17.1.1　事前关系

由于只考虑连续进程，在单个进程中执行的所有事件都是有序的。同样，根据因果关系，消息只有在它被发送之后才能被接收。因此，可以如下(假设发送和接收一条消息组成一个事件)在一组事件上定义事前关系(用→来表示)：

1. 如果 A 和 B 是同一进程中的事件，并且 A 在 B 之前执行，则 A→B。
2. 如果 A 是某个进程中发送消息的事件，B 是另一进程中接收此消息的事件，则 A→B。
3. 如果 A→B，且 B→C，则 A→C。

一个事件不能在它自己之前发生，所以"→"关系是一个非反射的部分排序。

如果事件 A 和 B 不是"→"关系(即 A 不在 B 之前发生，B 也不在 A 之前发生)，则称这两个事件是**并发**执行的。此时，无论哪个事件都不能以因果关系影响另一个。然而，如果 A→B，则事件 A 可能会影响到事件 B。

如图 17.1 所示，用时空图最能说明并发和事前的定义。横坐标表示空间(即不同的进程)，纵坐标表示时间。有标记的垂直线表示进程(或处理器)，圆点标记表示事件，波浪线表示从一个进程发送到另一进程的消息。在此图中，当且仅当 A 和 B 或 B 和 A 之间不存在路

径，则称事件 A 和 B 是并发的。

例如，分析一下图 17.1。一些具有*事前*关系的事件为：

$p_0 \rightarrow q_2$，

$r_0 \rightarrow q_1$，

$q_3 \rightarrow r_1$，

$p_1 \rightarrow q_4$（由于 $p_1 \rightarrow q_2$，且 $q_2 \rightarrow q_4$）

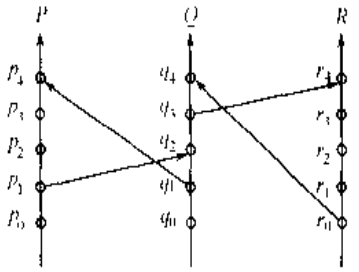


图 17.1 三个并发进程的相对时间

系统中一些具有并发关系的事件有：

q_0 和 p_2，

r_0 和 q_3，

r_3 和 p_3，

q_3 和 p_3

无法确定两个并发事件如 q_0 和 p_2 中哪个先发生。然而，由于它们之间并不相互影响（因为其中一个没有办法知道另一个是否已发生），所以哪个先发生并不重要。重要的只是那些在意两个并发事件先后顺序的进程同意按某种顺序排序。

17.1.2 实现

为了确定事件 A 是在事件 B 之前发生的，需要一个公共时钟，同时需要一个完全同步的时钟集。由于这些在分布式系统中都不现实，所以必须定义一个*事前*关系，而不用物理时钟。

将每个系统事件与一个**时间戳**相联系，然后可以定义**全局排序**的必要条件：对每一对事件 A 和 B，如果 A→B，则 A 的时间戳小于 B 的时间戳。（下面将看到相反的不必为真。）

如何在一个分布式系统中执行全局排序的必要条件呢？在每个进程 P_i 中定义一个**逻辑时钟** LC_i，逻辑时钟可以通过一个简单的计数器来实现，它在进程中执行的任意两个连续事件之间增加。由于逻辑时钟具有**单调增加值**，它赋予每个事件一个惟一的号码，如果在进程 P_i 中，事件 A 在事件 B 之前发生，则 $LC_i(A) < LC_i(B)$。一个事件的时间戳是该事件的逻辑时钟值。此方法保证了对同一进程中的任意两个事件满足全局排序的必要条件。

不幸的是，此方法并不能保证跨进程时满足全局排序的必要条件。为了说明这个问题，考虑两个相互通信的进程 P_1 和 P_2。假设 P_1 发送一条消息给 P_2（事件 A），其中 $LC_1(A) = 200$，并且 P_2 接收到消息（事件 B），$LC_2(B) = 195$（由于 P_2 的处理器比 P_1 的慢，它的逻辑时钟也就较慢）。由于 A→B，但 A 的时间戳比 B 的时间戳大，故这种情况违反必要性条件。

为了解决此类问题，要求当进程接收到一个时间戳大于它的逻辑时钟当前值的消息时，增加它的逻辑时钟。特别是，如果进程 P_i 接收一个消息（事件 B），其时间戳为 t 且 $LC_i(B) \leqslant t$，它应增加它的逻辑时钟为 $LC_i(B) = t + 1$。因此，在这里的例子中，当 P_2 接收到来自于 P_1 的消息时，它将增加它的逻辑时钟为 $LC_2(B) = 201$。

最后，为了实现全部排序，根据时间戳排序方法，只需遵守如下规则，如果两个事件 A 和 B 的时间戳相同，则事件是并发的。在这种情况下，可以使用进程号来打破僵局，产生一个全局排序。17.4.2 小节将介绍时间戳的使用。

17.2 互 斥

本节将提出若干在分布式环境中实现互斥的算法。假设系统包括 n 个进程，其中每个进程驻留在不同的处理器上。为简化讨论，还假设进程被编上惟一的编号，从 1 到 n，且在进程与处理器之间存在一一对应关系（即每个进程都有它自己的处理器）。

17.2.1 集中式算法

在提供互斥的集中式算法中，系统中的某个进程被选为进入临界区的协调者。每个想调用互斥的进程发送一条请求（*request*）消息给协调者，当此进程接收到一条来自于协调者的应答（reply）消息后，它就可以开始进入它的临界区。而在退出它的临界区后，该进程发送一条*释放*（release）消息给协调者并继续进行下去。

在接收一条*请求*消息时，协调者检测是否有其他进程在临界区中。如果没有进程在临界区，协调者马上返回一条*应答*消息。否则，该请求被排队等待。当协调者接收到一条释放消息后，它从队列中移出一条请求消息（根据一些时序安排算法），并发送一条*应答*消息给请求进程。

很显然该算法保证了互斥。另外，如果协调者的时序安排策略公平（例如先来先服务方法（FCFS）），则不会发生饥饿。每次进入临界区，该方法需要三条消息：*请求消息*，*应答消息*和*释放消息*。

如果协调者进程出错，必须有一个新的进程来取代它的位置。在 17.6 节中，将介绍选举一个新的惟一协调者的一些算法。一旦选举出一个新的协调者，它必须调查系统中所有的进程，以重构请求队列。一旦该队列构造好，计算就可以继续进行。

17.2.2 完全分布式的算法

如果要将决策分布到整个系统，解决方法将会复杂很多。在此提出一个基于 17.1 节中所介绍的事件排序方法的算法。

当进程 P_i 想进入临界区时，它产生一个新的时间戳 TS，并发送一条*请求*消息（P_i，TS）给系统中的所有其他进程（包括它自己）。当这些进程接收到请求消息时，它们可能马上回复（即发回一条*应答*消息给 P_i），或者推迟发送一个应答消息（因为它自己已经在其临界区中）。如果这个进程接收到系统中所有其他进程的应答消息，则它可以进入临界区，并对到来的请求进行排队且延迟它们。在退出临界区后，该进程向所有被它延迟的请求发送*应答*

消息。

进程 P_i 是否立即回应一条请求消息(P_j, TS),基于如下三个因素:

1. 如果进程 P_i 已经在它的临界区,则它推迟对 P_j 的应答。

2. 如果进程 P_i 不想进入它的临界区,则它立即发送应答给 P_j。

3. 如果进程 P_i 想进入它的临界区却还未进入,则它比较自己的请求时间戳和进程 P_j 产生的请求时间戳 TS,如果它自己的请求时间戳大于 TS,则它马上发送一条应答消息给 P_j(P_j 先请求),否则,应答被推迟。

该算法体现了下面的特征:

• 获得了互斥。

• 保证了从死锁中获得自由。

• 由于进入临界区是根据时间戳排序,保证了从饥饿中得到解放。时间戳排序保证进程按 FCFS 次序服务。

• 每次进入临界区所发送消息的数量为 $2\times(n-1)$,这个数量是当各进程独立且并发地执行时,每次进入临界区所需消息的最小值。

为了说明算法如何工作,考虑一个包含进程 P_1、P_2 和 P_3 的系统,其中假设进程 P_1 和 P_3 想进入它们的临界区。进程 P_3 发送一条请求消息(P_3,时间戳=4)给 P_1 和 P_2,与此同时进程 P_1 发送一条请求消息(P_1,时间戳=10)给 P_2 和 P_3,时间戳 4 和 10 从 17.1 节中描述的逻辑时钟中得到。当进程 P_2 接收到这些请求消息后,它立即回应。当进程 P_1 接收到 P_3 的请求消息后,它立即回应,因为它自己请求消息的时间戳(10)大于 P_3 的时间戳(4)。当进程 P_3 接收到进程 P_1 的请求消息后,由于它的时间戳(4)小于 P_1 的时间戳(10),它推迟回应。在接收来自于 P_1 和 P_2 的应答后,进程 P_3 可以进入它的临界区。而在退出临界区后,进程 P_3 发送一个应答给进程 P_1,然后进程 P_1 可以进入临界区。

由于此方法需要系统中所有进程参与,它产生三个额外后果:

1. 进程需要知道系统中所有其他进程的标识。当一个新的进程加入到参与互斥算法的进程组中时,必须采取如下措施:

a. 进程必须接收组中所有其他进程的进程名。

b. 新的进程名必须散布到组内所有其他进程中。

这个任务不像它看起来的那么小,因为当新进程加入到进程组中时,某些请求和应答消息可能在系统中循环。有兴趣的读者可以参考推荐书目以了解更多细节。

2. 如果一个进程出错,整个算法就会崩溃。可以通过连续监控系统中所有进程的状态来解决此问题。如果一个进程出错,则所有其他进程都被通知到,以使它们不再发送请求消息给出错的进程。当一个进程恢复后,它必须启动允许它重新加入进程组的程序。

3. 未进入其临界区的进程必须经常暂停,以保证其他想进入临界区的进程。因此该协议适合一组小的、稳定合作的进程。

17.2.3 令牌传递算法

另一种提供互斥的算法是在系统的进程之间循环传递一个令牌。**令牌**(token)是在系统中传递的一种特殊的消息。只有令牌的持有者才有权进入临界区。由于只有一个令牌，因此一次只有一个进程能进入临界区。

系统中的进程被*逻辑地*组织成一个**环结构**，而实际的物理通信网络则不必为环状。只要进程与另一进程相连，就可以形成一个逻辑环。为了实现互斥，在环中传递令牌。当一个进程得到此令牌，它保管令牌，并可以进入它的临界区。当此进程退出临界区，令牌又将被传递。如果得到令牌的进程不想进入临界区，它将令牌传递给它的邻居。此算法类似于第七章所讲的算法 1，只是用令牌代替了一个共享变量。

如果环是单向的，则可保证不会产生饥饿。实现互斥所需的消息数量会有所变化，在高争夺时(即每个进程都想进入临界区)每次进入需一条消息，在低争夺时(即没有进程想进入临界区)需要无穷的消息。

必须考虑到两种类型的错误。第一，如果令牌丢失，必须通过一次选举来产生新的令牌。第二，如果一个进程出错，必须建立一个新的逻辑环。在 17.6 节中，提出一个选举算法，当然还有其他算法。重构环的算法将在题 17.6 中留给读者。

17.3 原 子 性

第七章介绍了原子事务的概念，它是一个必须**原子**执行的程序单元。即或者所有与它相关的操作都执行完，或者没有一个被完成。当处理分布式系统时，保证事务的原子特性要比在集中式系统中更为复杂。困难可能源于有几个站点参与了单个事务的执行，这些站点中的一个出错，或连接这些站点的通信失败，都可能导致计算错误。

分布式系统中的**事务协调者**的作用在于保证分布式系统中事务执行的原子性。每个站点都有它自己的本地事务协调者，负责协调所有始于该站点的事务的执行。对每一件这样的事务，协调者负责如下工作：

- 启动事务的执行。
- 将事务分成若干子事务，并将这些子事务分布到合适的站点去执行。
- 协调事务的结束，它可能导致事务被提交到所有的站点，或在所有站点终止。

假设每个本地站点维护一个恢复用的日志。

17.3.1 两阶段提交协议

为了保证原子性，执行事务 T 所涉及的所有站点必须在执行的最终结果上取得一致。T 必须提交给所有站点，或在所有站点终止。为了保证该特性，T 的事务协调者必须执行一

个**提交协议**。最简单且使用最广泛的协议是**两阶段提交**(2PC)协议,下面就讨论它。

假设 T 是站点 S_i 发起的一个事务,并假设站点 S_i 的协调者为 C_i,当 T 执行完成后——即当执行 T 的所有站点通知 C_i,说 T 已完成——然后 C_i 启动 2PC 协议。

• **第一阶段**:C_i 将记录<prepare T>加到日志中,并将记录存入稳定的存储器中。然后发送一条 *prepare*(T)消息给所有执行 T 的站点。在接收到这样一条消息后,站点上的事务管理者决定是否提交它的 T 部分。如果回答是"不",添加一条<no T>记录到日志中,然后通过发送一条 *abort*(T)消息给 C_i 来做出反应。如果回答是"是",则添加一条<ready T>记录到日志中,然后它将所有符合 T 的日志记录存入稳定的存储器中。事务管理者用一条 *ready*(T)消息回答 C_i。

• **第二阶段**:当 C_i 接收到所有其他站点对其所发消息 *prepare*(T)的响应时,或者当消息被送出后已有一段预先指定的时间间隔流逝时,C_i 可以确定是否提交或终止事务 T。如果 C_i 从所有参与的站点处接收到 *ready*(T)消息,则事务 T 可被提交。否则,事务 T 必须被终止。根据此裁决,或者在日志中加入记录<*commit* T>,或者加入记录<*abort* T>,且被强制写入稳定存储器上。此时,事务的命运是未知的。随后,协调者或者发送消息<*commit* T>,或发送消息<*abort* T>给所有参与的站点。当一个站点接收到此消息后,它将消息记录到日志中。

执行 T 的一个站点在它发送 *ready*(T) 消息给协调者之前的任何时候都可以无条件地终止 T。*ready*(T) 消息实际上是一个站点许诺以遵循协调者的命令来提交 T 或终止 T。一个站点能做出此承诺的惟一情形是所需的信息已存入到稳定的存储器中。否则,如果站点在发送 T 准备好之后崩溃,它就不可能实现它的承诺。

由于提交一件事务需要全体一致,一旦至少有一个站点用 *abort*(T)做出反应,那么 T 的命运就成为未知的了。由于协调者站点 Si 是执行 T 的站点之一,因此协调者可以单方面地终止 T。当协调者将结论(究竟是提交还是终止)写入日志,并强制存入稳定存储器中时,对 T 的最后裁决才确定。在某些 2PC 协议的实现方式中,一个站点在两阶段协议的最后发送一条应答 T 消息给协调者,当协调者接收到所有站点的应答 T 消息时,它将记录<complete T>加到日志中。

17.3.2 2PC 中的错误处理

现在来仔细研究 2PC 如何对各种类型的错误做出反应。正如将要看到的,2PC 协议的一个主要缺点在于协调者的错误可能导致阻塞,使得是否提交或终止 T 的决定被延迟,直到 C_i 恢复。

1. 一个参与站点的出错

当一个参与的站点 S_k 从一次出错中恢复后,它必须检查它的日志以决定那些当错误发生时正在执行的事务的命运。假设 T 是这样一个事务,考虑如下的可能:

• 日志包含一条<commit T>记录。这种情况下，站点执行 redo(T)。

• 日志包含一条<abort T>记录。这种情况下，站点执行 undo(T)。

• 日志包含一条<ready T>记录。这种情况下，站点必须查阅 C_i 来决定 T 的命运。如果 C_i 已准备好，它通知 S_k 究竟 T 是提交还是终止。前一种情况下，执行 redo(T)；后一种情况下，执行 undo(T)。如果 C_i 停机，S_k 必须从其他站点发现 T 的命运，这通过向系统中所有站点发送一条 *query-status*(T)消息来完成。当某个站点接收到此消息后，它必须查阅它的日志来决定是否 T 已在那里执行，如果是，再决定是提交还是终止 T，然后通知 S_k 此结果。如果没有站点具有适当的信息(即是否提交或终止 T)，则 S_k 既不能提交也不能终止 T。那么关于 T 的决定被推迟，直至 S_k 能够获得所需的信息。因此 S_k 必须定时向其他站点重发 *query-status*(T)消息，直到一个包含所需信息的站点恢复。C_i 所驻留的站点总是会包含所需的信息。

• 日志没有包含关于 T 的控制记录(终止、提交、准备)。缺乏控制记录意味着 S_k 在响应来自于 C_i 的 prepare T 消息之前出错。由于 S_k 的出错阻碍这种响应的发送，在本书的算法中，C_i 必须终止 T。因此，S_k 必须执行 undo(T)。

2. 协调者出错

如果协调者在执行事务 T 提交协议的过程中出错，那么需要由参与的站点决定 T 的命运。大家将会看到，在某些情况下，参与的站点不能决定是否提交或终止 T，故这些站点必须等待出错的协调者恢复。

• 如果一个活动站点在其日志中包含一条<commit T>记录，则 T 必须被提交。

• 如果一个活动站点在其日志中包含一条<abort T>记录，则 T 必须被终止。

• 如果有些活动站点在其日志中未包含一条<ready T>记录，则出错的协调者 C_i 不能决定是否提交 T。这是因为一个在其日志中没有<ready T>记录的站点不会向 C_i 发送一条 ready(T)消息，因而可以得出此结论。但是，协调者可以决定终止 T 并不提交 T。由于不需要等待 C_i 恢复，它更可能选择终止 T。

• 如果上述情况都没有，则所有活动站点必须在其日志中具有一条<ready T>记录，但没有额外的控制记录(如<commit T>或<abort T>)。由于协调者出错，所以在协调者恢复之前，无法确定是否已经做出了决定，或者决定是什么。因此，活动站点必须等待 C_i 恢复。由于 T 的命运仍未知，T 可能继续占有资源。例如，如果使用了加锁，T 可能仍旧保留了活动站点上数据的锁。这种情况是不符合需要的，因为 C_i 重新生效可能要数小时或数天。其间其他事务也可能被迫等待 T。结果是数据不仅在出错的站点(C_i)上无法使用，在活动的站点上也如此。随着故障时间的增长，无法使用的数据量会不断增加。由于在站点 C_i 恢复之前 T 被锁住，这种情况被称为封锁(*blocking*)问题。

3. 网络出错

当一个链接出错时，所有通过该链接的进程中的消息都不能完整地到达它们的目的地。

从连接到该链接的站点来看，其他站点仿佛都出错了。因此，前面所讨论的方法也可用于此。

当许多链接出错时，网络可能会被断开。此时存在两种可能。协调者以及它所有的参与者可能在同一分区上，此时链接失败并不影响提交协议。另一种情况是，协调者与它的参与者可能属于几个分区，此时参与者和协调者之间的消息丢失，故要注意减少链接失败的情况发生。

17.4 并发控制

在这一节中，将说明第七章中所讨论的并发控制算法经修改后是如何用于分布式环境的。

分布式数据库系统的事务管理者管理访问存储在本地站点的数据的事务，这些事务或是一个本地事务(即一个只在该站点执行的事务)，或是全局事务(即在几个站点执行的事务)的一部分。每个事务管理者负责维护一个用于恢复的日志，并参与适当的并发控制方案，以协调在此站点上所执行事务的并发执行。正如将要看到的，第七章所讲的并发算法需要做些修改以适应事务的分布性。

17.4.1 加锁协议

第七章所讲的两段加锁协议可以用于分布式环境，需要改变的仅仅是锁管理者的实现方式。在这一节提出几种设计方法，第一种方法处理不允许数据复制的情况，其他方法适用于在多个站点复制数据的更为通用的情况。如第七章所述，假设**共享锁模式**和**排它锁模式**的存在。

1. 非复制方法

如果系统中没有数据被复制，7.9 节所描述的加锁方法可以这样使用：每个站点维持一个锁管理者，它的功能是管理对存储在站点中的数据的加锁和解锁请求。当一个事务希望在站点 S_i 对数据项 Q 加锁，它简单地发送一条消息给站点 S_i 的锁管理者以请求加锁(以某种加锁方式)。如果数据项 Q 以不一致的方式加锁，则请求被延迟，直到请求被批准。一旦锁管理者认为请求可以被批准，则发回一条消息给初始者，以表明加锁请求已被批准。

此方法具有易于实现的优点。它需要两个消息传递来处理加锁请求，另一个消息传递来处理解锁请求。然而，死锁处理更为复杂。由于加锁和解锁请求不在一个站点上生成，第八章所介绍的各种死锁处理算法必须要修改，这些修改将在 17.5 节中讨论。

2. 单协调者方法

在单协调者方法中，系统维护驻留在单个选定的站点(如 S_i)上的单个锁管理者。所有的加锁和解锁请求都在站点 S_i 上生成。当某事务需要对一数据项加锁时，它发送一个加锁请求给 S_i，锁管理者决定是否同意立即加锁。如果同意，它发回一条消息给加锁请求者。否则，请求被延迟，直到被同意，此时发送一条消息给加锁请求者。事务可以从任意一个拥有

该数据项拷贝的站点上读取该数据项。在写操作的情况下，所有具有该数据项拷贝的站点都要涉及写操作。

此方法具有以下优点：

- **易实现**：此方法只需两个消息来处理加锁请求以及另一个消息来处理解锁请求。
- **易进行死锁处理**：由于所有的加锁和解锁请求都在一个站点上进行，第八章所介绍的死锁处理算法可直接应用于此系统。

此方法也有以下缺点：

- **瓶颈**：站点 S_i 成为瓶颈，因为所有请求都要在此处理。
- **脆弱性**：如果站点 S_i 出错，使得并发控制丢失。此时要么必须停止处理，要么必须使用恢复方案。

通过**多协调者方法**可以综合上述优点和缺点，其中锁管理者功能被分布到多个站点上。

每个锁管理者管理数据项子集的加锁和解锁请求，每个锁管理者驻留在不同的站点上。这种分布减少了协调者的瓶颈程度，但由于加锁和解锁请求不在一个站点上进行，它增加了死锁处理的复杂性。

3. 多数协议

多数协议是对前面所讲的非复制数据方法的修改。系统在每个站点维护一个锁管理者，每个管理者控制存储在站点上的所有数据或其拷贝的加锁。当某事务希望对一个在 n 个不同的站点上拥有拷贝的数据项 Q 加锁时，该事务必须对超过半数的站点发送一个加锁请求。每个锁管理者决定是否能立即加锁（只要它开始关注这个消息）。跟前面一样，应答会被延迟，直到请求被批准。直到事务成功地获得对 Q 拷贝的多数加锁，它才在 Q 上进行操作。

此方法以一种分散的方式来处理拷贝数据，从而避免了集中控制的缺点。然而它仍然有自己的缺点：

- **实现**：多数协议比前面几种方法实现起来复杂得多。它需要 $2(n/2+1)$ 个消息来处理加锁请求，还要 $(n/2+1)$ 个消息来处理解锁请求。
- **死锁处理**：由于加锁和解锁请求不在一个站点上进行，必须修改死锁处理算法（参见 17.5 节）。此外，即便只有一个数据项被加锁，也可能发生死锁。为了说明此问题，考虑一个具有 4 个站点且完全复制的系统。假设事务 T_1 和 T_2 想以排它方式对数据项 Q 加锁，事务 T_1 可能在站点 S_1 和 S_3 上对 Q 加锁成功，而事务 T_2 可能在站点 S_2 和 S_4 上对 Q 加锁成功，然后每个均需等待第三次加锁，故而产生了死锁。

4. 偏倚协议

偏倚协议（biased protocol）基于类似于多数协议的模式。不同之处在于对共享锁的请求比对排它锁的请求得到了更便利的处理。系统在每个站点维护一个锁管理者，每个锁管理者管理所有存储在此站点数据项的锁。**共享锁和排它锁**以不同的方式处理：

• **共享锁**:当某事务需要对数据项 Q 加锁时,它简单地从一个包含 Q 拷贝的站点上的锁管理者那里请求对 Q 加锁。

• **排它锁**:当一个事务需要对数据项 Q 加锁时,它从所有包含 Q 拷贝的站点上的锁管理者那里请求对 Q 加锁。

如前面一样,对该请求的应答会被延迟,直到它被批准。

此方法具有比多数协议更少读操作开销的优点。因为通常情况下都是读的频率大大高于写的频率,因此这个优点在此显得尤为重要。但是,它的缺点是出现了额外的写开销。并且,偏倚协议与多数协议同样存在处理死锁复杂的缺点。

5. 主拷贝

在数据复制中,可以选择某个拷贝做为主拷贝。因此,对于每个数据项 Q,Q 的主拷贝必须准确驻留在一个站点上,该站点被称为 Q 的主站点。

当一个事务需要对一数据项 Q 加锁时,它请求在 Q 的主站点上加锁。同前面一样,该请求的应答被延迟,直到它被批准。

因此,主拷贝使复制数据的并发控制与非复制数据的并发控制类似。此处理方法允许简单地实现,但如果 Q 的主站点出错,即使存在其他可以访问的包含此拷贝的站点,Q 也变得不可访问。

17.4.2 时间戳

7.9 节中介绍的时间戳方法的主要思想是每个事务被赋予一个*惟一的*时间戳,用它来决定顺序。因此在将集中式方法扩充到分布式方法时的第一个任务,就是要开发一个能够产生惟一的时间戳的方案。此前所讨论的协议可以直接应用于非复制环境中。

1. 惟一时间戳的产生

有两个主要的方法用来生成惟一的时间戳,一个是集中式,另一个是分布式。在集中式方法中,选择一个站点来分派时间戳。该站点可以使用逻辑计数器或它自己的时钟来完成任务。

在分布式方法中,每个站点利用逻辑计数器或它自己的时钟来生成一个惟一的本地时间戳。全局的惟一时间戳则通过将此本地惟一时间戳与站点标识相连接来获得,该站点标识也必须惟一(图 17.2)。连接的顺序非常重要!在低位中使用站点标识来保证某个站点产生的全局时间戳并不总是大于某些其他站点的时间戳。可将生成惟一时间戳的这种技术与 17.1.2 小节中生成惟一名字的方法进行对比。

如果一个站点产生本地时间戳的速度比其他站点快,则仍然存在问题。此时,速度快的站点的计数器比其他站点的大。因此,所有产生于速度快的站点的时间戳将大于其他站点生成的时间戳。此时需要有一个机制来保证在整个系统中公平地产生时间戳。为了完成公平时间戳的产生,在每个站点 S_i 中定义一个逻辑时钟(LC_i),它产生惟一的本地时间戳(参

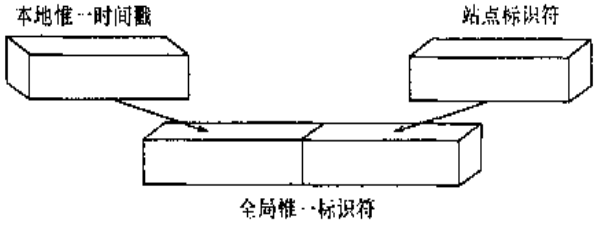


图 17.2 惟一时间戳的产生

见 17.1.2 小节)。为了保证不同的逻辑时钟同步,当具有时间戳$<x, y>$的事务 T_i 访问此站点,其中 x 比 LC_i 的当前值大时,需要站点 S_i 调快它的逻辑时钟。此时,站点 S_i 将它的逻辑时钟调快为 $x+1$。

如果用系统时钟来产生时间戳,假如没有一个站点的系统时钟跑快了或跑慢了,则时间戳是公平的。但由于时钟并不完全精确,因此必须用一个类似于逻辑时钟的技术来保证不会有时钟比其他时钟变得过于超前或过于滞后。

2. 时间戳排序方法

7.9 节中介绍的基本时间戳方法可以直接扩展到分布式系统中。如同在集中式系统中如果没有机制用来防止事务读取一个未提交的数据项,可能导致层叠式回滚。为了消除层叠式回滚,可以把 7.9 节中介绍的基本时间戳方法与 17.3 节中介绍的 2PC 协议相结合,从而保证不会出现层叠回滚的串行能力。将此算法留给读者去开发。

刚才介绍的基本时间戳方法还存在着一些问题,即事务间的冲突通过回滚而非等待来解决。为了减轻此问题,可以缓冲各种读和写操作(即延迟它们),直到得到保证这些操作不会引起中止。如果存在一个事务 T_j,它将执行一个 write(x)操作,但还未开始,且 $TS(T_j)<TS(T_i)$,则事务 T_i 的一个 read(x)操作必须被延迟。类似地,如果存在一个事务 T_j,它将执行一个 write(x)操作或一个 read(x)操作,且 $TS(T_j)<TS(T_i)$,则事务 T_i 的一个 write(x)操作必须被延迟。可用不同的方法来保证此特性。一种被称为**保守的时间戳排序方法**需要每个站点为所有那些将要在站点上执行但为了防止上述问题而必须被延迟的读写请求分别维护一个读写队列。在此不介绍此方法,而将算法留给读者去做。

17.5 死锁处理

第八章所讲的死锁预防、死锁避免以及死锁检测算法可以扩展用于分布式系统。下面描述几个分布式算法。

17.5.1 死锁预防

第八章所讲的死锁预防和死锁避免算法在经过适当的修改后可用于分布式系统。例

如，只需对系统资源简单地定义一个全局排序，就可以利用资源排序的死锁预防技术。即整个系统中所有的资源都被赋予惟一的编号，只有当进程当前未占用编号大于 i 的资源时，才可以请求编号为 i 的资源(在任何处理器上)。类似地，可以在分布式系统中用银行家算法，通过指定系统中的某个进程(银行家)作为维护所需信息的进程来实现银行家算法，每个资源请求必须通过银行家引导。

这两种方法可用在分布式环境中处理死锁问题。全局资源排序的死锁预防设计的实现简单，开销很少。银行家算法的实现也较简单，但它可能需要较多的开销。由于出入银行家的消息数量可能很大，故银行家可能成为瓶颈。因此，银行家算法在分布式系统中可能不太实用。

在这一节中，提出一种新的死锁预防方法，该方法基于资源抢占的时间戳排序方法。尽管此方法能处理任何分布式系统中可能出现的死锁，为了简单起见，仅考虑每种资源类型只有单个实例的情况。

为了控制抢占，给每个进程赋予一个优先权号，这些编号用来决定进程 P_i 是否应等待进程 P_j。例如，如果 P_i 比 P_j 有更高的优先权，可以让 P_i 等待 P_j，否则 P_i 应回滚(roll back)。该方法防止了死锁，因为对等待关系图中的每条边 $P_i \rightarrow P_j$，P_i 比 P_j 具有更高的优先权，因此不会形成回路。

此方法的一个问题在于可能产生饥饿，一些优先权特别低的进程可能永远被回滚。可以用时间戳来避免此问题发生。系统中的每个进程在它生成时被赋予一个惟一的时间戳。下面提出了两个相互补充的使用时间戳的死锁预防方法：

1. **等待－死亡方法**(wait-die)：该方法基于非抢占技术。当进程 P_i 请求一个正被 P_j 占用的资源时，只有当 P_i 的时间戳小于 P_j 时，允许 P_i 等待(即 P_i 比 P_j 老)。否则，P_i 被回滚(死亡)。例如，假设进程 P_1、P_2 和 P_3 的时间戳分别为 5、10 和 15，如果 P_1 请求一个由 P_2 占有的资源，P_1 将等待；如果 P_3 请求一个由 P_2 占有的资源，则 P_3 将被回滚。

2. **伤害－等待方法**(wound-wait)：该方法基于抢占技术，是等待－死亡系统的对等。当进程 P_i 请求一个正被 P_j 占用的资源时，只有当 P_i 的时间戳大于 P_j 时，允许 P_i 等待(即 P_i 比 P_j 年轻)。否则，P_j 被回滚(P_j 被 P_i 伤害)。回到上一个例子，在进程 P_1、P_2 和 P_3 中，如果 P_1 请求一个由 P_2 占有的资源，则 P_2 的资源将被抢占，P_2 被回滚；如果 P_3 请求一个由 P_2 占有的资源，则 P_3 等待。

假定一个进程被回滚，不再被赋予新的时间戳，则上面两种方法都能避免饥饿。由于时间戳总是不断增大的，被回滚的进程将最终有一个最小的时间戳，这样，它将不再被回滚。然而在操作时，这两种方法有许多不同：

1. 在等待－死亡方法中，老的进程必须等待一个年轻的进程来释放它的资源。因此，进程越老，它越趋于等待。相反，在伤害－等待方法中，一个老的进程永远不会等待一个年轻的进程。

2. 在等待－死亡方法中，如果进程 P_i 由于请求由进程 P_j 占据的资源而死去并被回滚时，则进程 P_i 在被重启后，可能会重新发出同样的排序请求。如果资源仍然被 P_j 占据，P_i 将再次死去。因此，P_i 在得到所需的资源前可能死几次。将这些事件与它们在伤害－等待方法中对比，由于 P_j 请求一个 P_i 占据的资源，进程 P_i 将被伤害并被回滚。当 P_i 被重新启动，请求一个正在被 P_j 占据的资源时，P_i 等待。因此，在伤害－等待方法中将发生更少的回滚。

两种方法都存在的主要问题是可能发生不必要的回滚。

17.5.2 死锁检测

死锁预防算法即使在没有死锁发生时也可能抢占资源。可以利用死锁检测算法，来避免不必要的抢占。构建一张描述资源状态的等待关系图，由于假设每种类型只有单个资源，等待资源图中的一个回路表示一次死锁。

分布式系统中的主要问题是如何维护等待关系图。通过描述几个通用的处理此问题的技术来予以说明。这些方法需要每个站点维持一张*本地*等待关系图，图中的节点对应于所有进程(本地的以及非本地的)，它们正在占用或请求本地的资源。例如，图 17.3 中有一个包括两个站点的系统，每个站点维护它的本地等待关系图，注意进程 P_2 和 P_3 在两个图中均出现，表明这些进程已在向两个站点请求资源。

这些本地等待关系图被本地进程和资源以常用的方式构建起来。当站点 S_1 中的进程 P_i 需要站点 S_2 中的 P_j 占据的资源时，一个请求消息由 P_i 发送给站点 S_2，边 $P_i \rightarrow P_j$ 被加入站点 S_2 的本地等待关系图。

显然，如果任何本地等待关系图存在回路，就发生了死锁。另一方面，即使一些本地等待关系图中没有回路也并不表明没有死锁发生。为了说明这个问题，考虑图 17.3 描绘的系统。每个等待关系图都是无环的，但系统中还是存在死锁。为了证明死锁并未发生，必须证实所有的本地图的**合并**是无环的。但事实上由图 17.3 中的两个资源图得到的合并图(图 17.4)就包含一条回路，这意味着系统处于死锁状态。

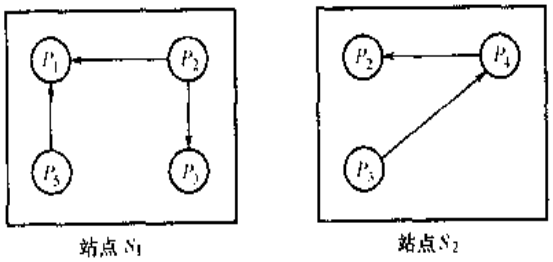


图 17.3 两张本地等待关系图

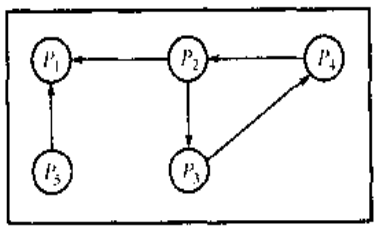


图 17.4 图 17.3 的全局等待关系图

在分布式系统中有许多方法被用来构建等待关系图。下面将介绍几种常用的方法。

1. 集中式方法

在集中式方法中，全局资源图作为所有本地等待关系图的并集，由一个专门的进程维护：**死锁检测协调者**。由于系统中存在通信延迟，必须区分两种类型的资源图。*真实*图及时描述了在任何情况下真实但未知的系统状态，就好像一个无所不知的观察者所观察到的那样。*构建*图是在协调者的算法执行期间产生的一个近似。构建图必须保证不管何时调用检测算法，报告的结果都是正确的。所谓*正确*，意味着：

- 如果存在一个死锁，它被正确地报告；
- 如果报告了死锁，系统事实上已处于死锁状态。

正如将要证明的，构造这样一个正确的算法是不容易的。

等待关系图可以在以下三个不同的时间及时得到构造：

1. 当从本地等待关系图中加入一条边或删除一条边时；
2. 周期性地，当在一个等待关系图中发生了一些变化时；
3. 当死锁检测协调者需要调用回路检测算法时。

先考虑第一种选择。只要从本地等待关系图中加入一条边或删除一条边，本地站点必须发送一条消息给协调者以通知它这个修改。协调者接收到此消息后，更新它的全局等待关系图。另一个选择（选择2）是一个站点可以周期性地在一条消息中发送许多这样的变化。回到前面的例子，协调者进程将维护一个如图17.4所示的全局等待关系图。当站点 S_2 加入一条边 $P_3 \rightarrow P_4$ 到它的本地等待关系图中时，它也发送一条消息给协调者。类似地，当站点 S_1 由于 P_1 释放 P_2 所请求的资源而删除边 $P_2 \rightarrow P_1$ 时，也发送一条相应的消息给协调者。

当调用死锁检测算法时，协调者搜索它的全局图。如果发现一条回路，挑选一个*牺牲者*并使之回滚。协调者必须把这一情况通知所有站点，这些站点反过来回滚牺牲者进程。

注意，在这种方法（第一种选择）中，可能发生不必要的回滚，它产生于如下两种情形：

1. 全局等待关系图中可能存在**错误的回路**。为了说明这一点，考虑一个图17.5所示的系统快照。假设 P_2 释放它在站点 S_1 所占的资源，导致删除站点 S_1 上的边 $P_1 \rightarrow P_2$。然后进程 P_2 请求由站点 S_2 上的 P_3 所占据的资源，并导致在站点 S_2 上增加边 $P_2 \rightarrow P_3$。如果在站点 S_2 上*增加*边 $P_2 \rightarrow P_3$ 的消息要比从站点 S_1 上*删除*边 $P_1 \rightarrow P_2$ 的消息提前到达，那么当协调者在*加入*之后（*删除*之前），可能会发现错误的回路 $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$。此时尽管没有发生死锁也可能启动死锁恢复。

2. 当一个死锁事实上已发生，并准备选取一个牺牲者，但与此同时一个进程因为某种与死锁不相关的原因被中止（如进程超过了分配给它的时间），这将会产生不必要的回滚。例如，假设图17.3中站点 S_1 决定终止 P_2，同时协调者已经发现一条回路，并选 P_3 为一个牺牲者，P_2 和 P_3 现在都回滚，尽管只有 P_2 需要回滚。选用 P_2 和 P_4 也存在同样的内在问题。

现在提出第三个选择，这个集中式死锁检测算法检测实际发生的所有死锁，不检测虚假死锁。为了避免报告虚假死锁，需要为来自于不同站点的请求添加惟一的标识（或时间戳）。

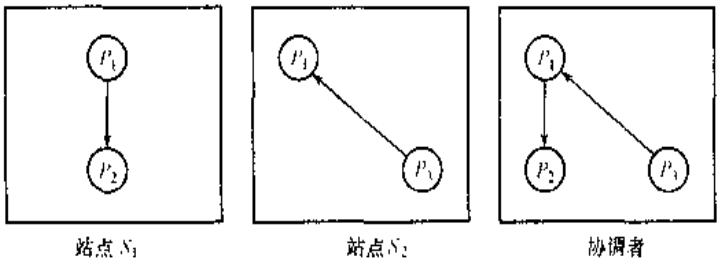


图 17.5 本地和全局等待关系图

当站点 S_1 上的进程 P_i 请求一个位于站点 S_2 上的进程 P_j 的资源时，即发送一条具有时间戳 TS 的请求消息。标为 TS 的边 $P_i \rightarrow P_j$ 被加入 S_1 的本地等待关系图中。只有当站点 S_2 接收到此消息且不立即批准请求资源时，该边才能加入 S_2 的本地等待关系图。在同一站点中，P_i 对 P_j 的请求用一般的方式处理，$P_i \rightarrow P_j$ 的边上不需要附加时间戳。死锁检测算法如下进行：

1. 控制者向系统中的所有站点发送一条开始消息。

2. 接收到此消息后，站点将它的本地等待关系图发给协调者。每个图包含关于站点的真实图状态的所有本地信息。该图反映了站点的瞬间状态，但并不涉及其他站点的同步。

3. 当控制者从每个站点收到反馈信息，它按如下方法构建图：

a. 图中每一点表示系统中的每一个进程。

b. 当且仅当其中一个等待关系图中存在边 $P_i \rightarrow P_j$，或在多于一个的等待关系图中存在具有时间戳 TS 标识的边 $P_i \rightarrow P_j$ 时，此图中存在边 $P_i \rightarrow P_j$。

可以说，如果构建图中包含一个回路，则系统处于死锁状态；如果构建图不包含一个回路，则当由于协调者发出开始消息(第一步中)而调用死锁算法时，系统未处于死锁状态。

2. 完全分布式方法

在**完全分布式死锁检测算法中**，所有的控制者同等地分担死锁检测的任务。在此方法中，每个站点根据系统的动态性能，构建一个等待关系图来表示全局图的一部分。它的思想是，如果存在一个死锁，则(至少)在一个局部图中存在一个回路。在此介绍一个包括在所有站点上构建局部图的算法。

每个站点维护它自己本地的等待关系图。此方法中的等待关系图与先前所讲的有所不同：将一个附加节点 P_{ex} 加到图中。如果 P_i 是在等待另一站点上的、由任意进程控制的数据项，则图中存在一条弧 $P_i \rightarrow P_{ex}$。类似地，如果另一站点上的一个进程在等待获取当前由此本地站点上的 P_j 进程控制的资源时，则图中存在一条弧 $P_{ex} \rightarrow P_j$。

为了说明这种情况，考虑图 17.3 中的两个本地等待关系图。两个图中增加的节点 P_{ex} 生成了图 17.6 所示的本地等待关系图。

如果一个本地等待关系图包含一条不涉及节点 P_{ex} 的回路，则系统处于死锁状态。然而，如果存在包含节点 P_{ex} 的回路，则意味着*有可能*死锁。为了确定是否存在死锁，必须调用

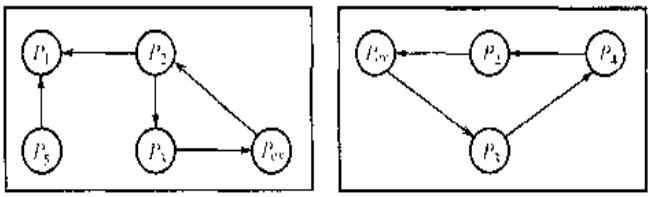


图 17.6 图 17.3 上增加的本地等待关系图

一个分布式死锁检测算法。

假设在站点 S_i，本地等待关系图包含一条涉及节点 P_{ex} 的回路，此回路的形式必为：

$$P_{ex}\rightarrow P_{k_1}\rightarrow P_{k_2}\rightarrow \cdots \rightarrow P_{k_n}\rightarrow P_{ex}$$

它表明站点 S_i 上的进程 P_{k_n} 在等待获取其他站点，如 S_j 上的一个数据项。在发现此回路时，站点 S_i 发送一条死锁检测消息给站点 S_j，该消息包含了回路信息。

当站点 S_j 接收到此死锁检测消息时，它用新的信息来更新它的本地等待关系图，然后寻找新的不涉及节点 P_{ex} 的回路的构建等待关系图。如果存在这样一个回路关系图，则发现一个死锁，且调用相应的死锁恢复方法。如果发现一条涉及节点 P_{ex} 的回路，则 S_j 发送一条死锁检测消息给适当的站点，如 S_k。随后，站点 S_k 重复此过程。因此，经过有限次循环，要么发现一个死锁，要么死锁检测计算停止。

为了说明此过程，分析一下图 17.6 中的本地等待关系图。假设站点 S_1 发现了回路

$$P_{ex}\rightarrow P_2\rightarrow P_3\rightarrow P_{ex}$$

由于 P_3 在等待获得站点 S_2 上的一个数据项，一条描述回路的死锁检测消息从 S_1 传送到 S_2。当站点 S_2 收到此消息，则更新它的本地等待关系图，得到如图 17.7 所示的等待关系图。该图包含回路

$$P_2\rightarrow P_3\rightarrow P_4\rightarrow P_2$$

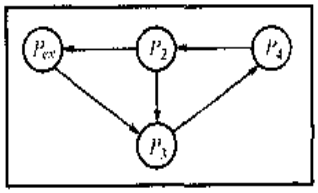


图 17.7 图 17.6 中站点 S_2 增加的本地等待关系图

它不包含节点 P_{ex}，因此，系统处于死锁状态，必须调用相应的恢复方法。

注意，如果站点 S_2 首先在本地等待关系图中发现回路，并发送死锁检测消息给站点 S_1，则结果是相同的。在最坏的情况下，两个站点将同时发现回路，并发送两条死锁检测消息：一个由 S_1 发往 S_2，另一个由 S_2 发往 S_1。这种情况导致了在更新两个本地等待关系图以及在两个图中寻找回路时产生了不必要的消息传送和开销。

为了减少消息通信量，给每个进程 P_i 一个惟一标识，在此用 $ID(P_i)$ 表示。当站点 S_k 发现它的本地等待关系图中包含一条涉及节点 P_{ex} 的回路

$$P_{ex}\rightarrow P_{k_1}\rightarrow P_{k_2}\rightarrow \cdots \rightarrow P_{k_n}\rightarrow P_{ex}$$

只有当 $ID(P_{k_n})<ID(P_{k_1})$ 时，它才发送一条死锁检测消息给另一站点。否则，站点 S_k 继续正常执行，而将启动死锁检测算法的责任留给其他站点。

为了说明此方法，再次分析图 17.6 中所示的由 S_1 和 S_2 维护的等待关系图。假设

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4)$$

假设两个站点同时发现本地回路。S_1 中的回路形式为

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}$$

由于 $ID(P_3) > ID(P_2)$，站点 S_1 不发送死锁检测消息给站点 S_2。

站点 S_2 中的回路形式为

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}$$

由于 $ID(P_2) < ID(P_3)$，站点 S_2 发送死锁检测消息给站点 S_1，S_1 接收到此消息后，更新它的本地等待图。然后站点 S_1 在图中寻找一条回路并发现系统处于死锁状态。

17.6 选举算法

正如在 17.3 节中所指出的，许多分布式算法使用一个协调者进程来完成系统中其他进程所需的功能。这些功能包括实施互斥、为死锁检测维护一个全局等待关系图、替换一个丢失的令牌或控制系统中的输入/输出设备。如果协调者由于它所在站点的出错而出现错误，只需通过重新启动其他站点上一个新的协调者备份，系统就可以继续执行。决定何处可以重新启动协调者的新备份的算法称为**选举算法**(election algorithms)。

选举算法假设一个惟一的优先权号与系统中每个活动进程相联系，为了简化符号，假定进程 P_i 的优先权号为 i。为了简化讨论，假定进程和站点之间是一一对应关系，故两者都被称为进程。协调者总是具有最大优先权号的进程。然后，当一个协调者出错时，算法必须选举出具有最大优先权号的活动进程，该优先权号必须送到系统中的每一个活动进程。此外，算法还必须为一个已恢复的进程提供一个机制来认出当前的协调者。

在这一节，介绍两种不同构造的分布式系统的选举算法实例。第一种算法适用于每个进程都能向所有其他进程发送消息的系统，第二种算法适用于组织为环状的系统(逻辑上或物理上)。两种算法每次选举都要发送 n^2 个消息，其中 n 为系统中的进程数。假定一个出现错误的进程知道在出错处进行恢复，因此可以采取适当的措施来重新加入活动进程集。

17.6.1 Bully 算法

假设进程 P_i 发送一个请求，它在一个时间间隔 T 内未被协调者响应。此时，假设协调者出现错误，并且 P_i 试图选举自己作为新的协调者。该任务通过下面的算法来完成。

进程 P_i 向所有具有更高进程号的进程发送一条选举消息，然后在时间间隔 T 内等待这些进程的响应。

如果在 T 时间间隔内没有收到任何响应，P_i 就假定所有进程号高于 i 的进程出现错误，并选它自己作为新的协调者。进程 P_i 重新启动一个新的协调者备份，并发送一条消息

告知所有优先权号小于 i 的活动进程，现在 P_i 是协调者。

然而，如果收到了回答，P_i 在一个时间间隔 T' 内，等待接收一条消息，并告诉它一个具有更高优先权号的进程已被选举（某些其他进程选举它自己为协调者，应在时间间隔 T' 内报告此结果）。如果在时间 T' 内没有发送消息，那么就假定具有更高优先权号的进程出现错误，进程 P_i 应重新启动算法。

如果 P_i 不是协调者，则在执行期间的任何时候，P_i 可以从 P_j 接收下面两条消息之一：

1. P_j 是新的协调者（$j>i$），且进程 P_i 记录该信息。

2. P_j 启动一次选举（$j<i$），这时如果 P_i 还没有启动这样的选举进程，则 P_i 发送一条响应消息给 P_j 并开始它自己的选举算法。

完成此算法的进程具有最高的进程号，并被选为协调者。它将它的进程号发送给其他所有进程号比它小的活动进程。一个出错的进程恢复之后，它马上开始执行同样的算法。如果没有更高进程号的活动进程，即使现在有一个进程号比它小的活动协调者，恢复的进程也强迫所有其他进程号小于它的进程让它成为协调者进程。由于此原因，该算法被称为**bully 算法**。

下面通过一个简单的例子来演示该算法的实施，该例子包括进程 P_1 到进程 P_4：

1. 所有进程都是活动的，P_4 是协调者进程。

2. P_1 和 P_4 出现错误。P_2 通过发送一个请求，并在时间 T 内未收到回答来检测到 P_4 出现错误。然后 P_2 通过发送一个请求给 P_3 来开始它自己的选举算法。

3. P_3 收到请求，并响应 P_2，通过发送一个请求给 P_4 来开始它自己的选举算法。

4. P_2 收到 P_3 的响应，开始等待一个时间间隔 T'。

5. P_4 在时间 T 内未响应，故 P_3 选举它自己为新的协调者，并发送进程号 3 给 P_1 和 P_2（其中 P_1 由于出现错误而未接收到）。

6. 接着，当 P_1 恢复后，它发送一条选举请求给 P_2、P_3 和 P_4。

7. P_2 和 P_3 响应 P_1，并开始它们自己的选举算法，根据前面的事件，P_3 将再次被选举。

8. 最后，P_1 恢复并通知 P_1、P_2 和 P_3，它是当前的协调者。（由于 P_4 是系统中最大进程号的进程，所以它发出不选举的请求。）

17.6.2 环算法

环算法（ring algorithm）假设连接是无方向的，并且进程将消息发送给它们右边的邻居。算法所用的主要数据结构是**活动列表**（active list），它包含算法结束时系统中所有活动进程的优先权号，每个进程维护它自己的活动列表。该算法工作如下：

1. 如果进程 P_i 检测到一个协调者出错，它生成一个新的初始值为空的活动列表，然后发送一条消息 $elect(i)$ 给它右边的邻居，并将进程号 i 加到它的活动列表中。

2. 如果进程 P_i 从它左边的进程收到一条消息 $elect(j)$，它必须用如下三种方式之一进

行反应：

a. 如果这是它见到或发送的第一条 *elect* 消息，P_i 生成一个具有 i 和 j 的新活动列表，然后发送消息 $elect(i)$，后面紧跟消息 $elect(j)$。

b. 如果 $i \neq j$，即接收的消息不包含 P_i 的进程号，则 P_i 将 j 加到它的活动列表中，并将消息转发到它的右邻居。

c. 如果 $i=j$，即 P_i 接收到消息 $elect(i)$，然后 P_i 的活动列表包含系统中所有活动进程的进程号，进程 P_i 现在就能确定活动列表中最大的进程号并用此标识新的协调者进程。

该算法并不指明一个恢复进程如何确定当前协调者进程的进程号。一种解决方法是要求一个恢复进程发送一条询问消息，该消息沿环向前传送到当前的协调者，该协调者又继续发送一条包含它的进程号的应答信息。

17.7 达成一致

为了使系统可靠，需要一种机制以允许一组进程在某个公共值上达成一致。这样的一致可能由于几个原因而不会发生。第一，通信介质可能出错，导致消息丢失或垃圾消息。第二，进程自己可能出错，导致不可预知的进程行为。此时，最希望的是进程能以一种干净的方式停止它们的执行，而不要偏离正常执行模式。最坏的情况是，进程可能发送垃圾信息或不正确信息给其他进程，或甚至与其他出错的进程一起来试图摧毁整个系统。

此问题用**拜占庭将军问题**(Byzantine generals problem)来表达。拜占庭军队有几个师，每个师由其自己的将军指挥，去包围敌人的阵地。几个将军必须在是否拂晓时进攻敌人的问题上达成一致。因为如果只由其中几个师去进攻将导致失败，故所有将军取得一致是非常关键的。这些师在地理位置上分布在不同地点，将军们之间的通信只能通过穿梭于阵地之间的信使。下面两个原因可能导致将军们不能达成一致：

1. 信使可能被敌人抓住而不能传递消息。此情况与计算机系统中的不可靠通信相符，将在 17.7.1 小节中进一步讨论。

2. 将军中可能有叛徒，试图阻止忠诚的将军取得一致。此情况与计算机系统中的进程出错相符，将在 17.7.2 小节中进一步讨论。

17.7.1 不可靠通信

假设进程以一种干净的方式出错，且通信介质是不可靠的。假设站点 S_1 上的进程 P_i 已经向站点 S_2 上的进程 P_j 发送了一个消息，现在它需要知道 P_j 是否接收到消息，以便能决定如何继续进行计算。例如，如果 P_j 收到消息，P_i 可以决定计算函数 foo，或者如果 P_j 未收到消息(由于某些硬件原因)，则决定计算函数 boo。

为了检测错误，可以使用一个类似于 15.6.1 小节所讲的**超时算法**。当 P_i 发出一条消

息时，它同时指定了一个时间间隔，在此期间它将等待一条来自于 P_j 的应答消息。P_j 接收到消息后，马上发一条应答消息给 P_i。如果 P_i 在规定时间间隔内接收到应答消息，它能确切地得出结论 P_j 已收到它的消息。然而，如果一次超时发生，则 P_i 需要再次发送消息并等待一个应答。这个过程持续到 P_i 要么等到应答消息，要么由系统通知 S_2 已停止。第一种情况中，它将计算 S，后一种情况中，它将计算 F。注意，如果只有两个可行的选择，P_i 必须等到它被告知其中一种情况发生。

现在假设 P_j 也需要知道 P_i 是否已收到它的应答消息，以便决定如何进行计算。例如，P_j 可能只有在确认 P_i 收到它的应答时，才想计算 foo。换言之，当且仅当 P_i 和 P_j 达成一致时，它们才会计算 foo。事实证明，当有错误出现时，将不可能完成任务。更为准确地讲，在分布式系统中，P_i 和 P_j 在它们各自的状态下不可能完全达成一致。

现在来证明这个结论。假设存在一个最小序列的消息传输，这些消息被发送后，两个进程都同意计算 foo。假设 m' 为 P_i 发送给 P_j 的最后消息，由于 P_i 不知道它的消息是否到达 P_j（消息可能因为错误而丢失），P_i 将不管消息投递的结果，执行 foo。因此，m' 可以从系列中被删除，而不会影响这个决定。因此，这个原来的系列不是最小的，与假设相矛盾，从而证明了不存在这样的序列。因此进程永远都不能确保两个都计算 foo。

17.7.2 故障处理

假定通信介质是可靠的，但进程会以不可预知的方式产生错误。考虑一个具有 n 个进程的系统，其中不超过 m 个进程产生故障。假设每个进程 P_i 具有私有值 V_i，希望设计一个算法允许非故障进程 P_i 构建向量 $X_i=(A_{i,1},A_{i,2},\cdots,A_{i,n})$ 满足下面的条件：

1. 如果 P_j 是非故障进程，则 $A_{i,j}=V_j$。

2. 如果 P_i 和 P_j 两个都是非故障进程，则 $X_i=X_j$。

有几种解决此问题的方法，它们都有以下特点：

1. 当且仅当 $n\geqslant 3\times m+1$ 时，可以设计一个正确的算法。

2. 达到一致的最坏延迟与 $m+1$ 个消息延迟成正比。

3. 需要达到一致的消息数量很大，没有任何单个进程是可信赖的，因此所有的进程必须集合它们所有的信息并做出自己的决定。

在这里并非要提出一个可能会很复杂的一般性算法，而是提出一个当 $m=1$、$n=4$ 的简单算法，该算法需要两轮的信息交换：

1. 每个进程发送它的私有值到其他三个进程。

2. 每个进程将它在第一轮中获得的信息发送给所有其他进程。

一个故障进程显然可能拒绝发送消息。此时，一个非故障进程可以选择一个任意值并假装该值由故障进程发送。

一旦这两轮完成，一个非故障进程 P_i 能如下构建它的向量 $X_i=(A_{i,1},A_{i,2},A_{i,3},A_{i,4})$：

1. $A_{i,i} = V_i$。

2. 对于 $j \neq i$ 的情况，如果进程 P_i 报告的三个值中至少两个值相同，那么这个多数值被用做 $A_{i,j}$ 的值，否则，一个缺省值 *nil* 被用来赋予 $A_{i,j}$。

17.8 小 结

在一个没有公共存储器和公共时钟的分布式系统中，有时无法确定两个事件发生的准确顺序。事先关系只是分布式系统中的一个部分排序方法。时间戳可用来提供分布式系统中的一致性事件排序。

分布式环境中的互斥可用几种方式实现。在集中式方法中，系统中的一个进程被选来以协调进入临界区的活动。在完全分布式的方法中，在整个系统的范围内做出决定。一个可应用于环状网络的分布式算法是令牌传递算法。

为了保证原子性，事务 T 执行的所有站点必须在执行的最终结果上达成一致。T 或者提交给所有的站点，或在所有的站点终止。为保证这一特性，T 的事务协调者必须执行一个提交协议，使用最广的提交协议是 2PC 协议。

用于集中式系统的多种并发协议经修改后可用于分布式环境。在加锁协议中，所需改变的只是锁管理者实现的方式。在时间戳和确认设计中，所需改变的只是开发一个产生惟一全局时间戳的机制，该机制能将本地时间戳与站点标识相连接，也可当一个更大时间戳的消息到达时，更新自己的本地时钟。

分布式系统中处理死锁的主要方法是死锁检测。主要的问题是决定如何维护等待关系图。组织等待关系图的方法包括集中式方法和完全分布式方法。

有些分布式算法需要用到协调者。如果由于协调者所在的站点错误而引起协调者错误，系统可以通过重新启动其他站点上的协调者备份来继续执行。这是通过维护一个备份协调者以准备在协调者错误时启用而做到的。另一个方法是在协调者产生错误后选举一个新的协调者，决定新的协调者备份在哪里重启的算法称为选举算法。Bully 算法和环算法是两种能用来在产生错误时选举一个新的协调者的算法。

习题十七

17.1 讨论本书给出的产生全局惟一时间戳的两种方法的优点和缺点。

17.2 假设你的公司正在建设一个计算机网络，要求你来编写一个实现分布式互斥的算法，你会使用哪一种方案？说出理由。

17.3 为什么在一个分布式环境中检测死锁比在集中式环境中的代价高得多？

17.4 假设你的公司正在建设一个计算机网络，要求你来开发一个解决死锁问题的方案。

a. 你会使用死锁检测方案还是死锁预防方案？

b. 如果你使用死锁预防方案，你会使用哪一种？为什么？

c. 如果你使用死锁检测方案，你会使用哪一种？为什么？

17.5 考虑下面的层次化死锁检测算法，在这个算法中，全局等待关系图分布在一些用树结构组织起来的*控制器*中。每个非叶的控制器维护一个等待关系图，图中包含了它的子树中控制器的图的相关信息。例如，S_A、S_B 和 S_C 是控制器，S_C 是 S_A 和 S_B 的最近的相先（S_C 必须惟一，因为所处理的是树）。假设结点 T_i 在控制器 S_A 和 S_B 的本地等待关系图中出现，那么 T_i 必须也出现在

- 控制器 S_C
- S_C 到 S_A 的路径中的每个控制器
- S_C 到 S_B 的路径中的每个控制器

的本地等待关系图中。另外，如果 T_i 和 T_j 在控制器 S_D 的等待关系图中出现，并且在 S_D 的某个孩子的等待关系图中存在一条 T_i 到 T_j 的路径，那么在 S_D 的等待关系图中必然存在一个 $T_i \rightarrow T_j$ 的边。

证明，如果在任何的图中存在一个环，则系统就会被死锁。

17.6 从本章给出的算法中导出一个效率更高的针对双向环的选择算法。对于 n 个进程，需要多少个消息？

17.7 假设在 2PC 的一个事务中出现了一个错误，针对每个可能的错误，解释 2PC 怎样在发生错误时仍确保事务的原子性。

推荐读物

Lamport[1978b]开发了将系统中 *happened-before* 关系扩展成所有事件一致排序的分布式算法。

Lamport[1978b]也开发了第一个在分布式环境中实现互斥的通用算法。Lamport 的方案对每个关键区条目要求 $3\times(n-1)$ 个消息。后来，Ricart 和 Agrawala[1981]设计一个只需要 $2\times(n-1)$ 个消息的分布式算法。17.2.2 小节中给出了他们的算法。Mackawa[1985]给出了一个分布式互斥的平方根算法。Lann[1977]设计了 17.2.3 小节中的针对环结构系统的令牌传递算法。Carvalho 和 Roucairol[1983]论述了计算机网络中的互斥问题。Agrawal 和 Abbadi[1991]给出了一个有效且容错的分布式互斥的解决方案。Raynal[1991]给出了一个简单的分布式互斥算法的分类。

Reed 和 Kanodia[1979]（共享内存环境）、Lamport[1978b]、Lamport[1978a]和 Schneider[1982]（完全分离的进程）论述了分布式同步问题。Chang[1980]给出了一个哲学家晚餐问题的分布式解决方案。

Lampson 和 Sturgis[1976]以及 Gray[1978]设计了 2PC 协议。Mohan 和 Lindsay[1983]讨论了 2PC 的两个修改版，称做假设提交和假设放弃，通过依据事务的结果定义默认的假设来减少 2PC 的开销。

Gray[1981]、Traiger 等[1982]以及和 Spector 和 Schwarz[1983]给出了处理在分布式数据库中实现事务概念问题的论文。Bernstein 等[1987]提供了关于分布式并发控制的全面论述。

Rosenkrantz 等[1978]发表了时间戳分布式死锁预防算法。Obermarck[1982]设计了 17.5.2 小节中的完全的分布式死锁检测方案。Menasce 和 Muntz[1979]中有习题 17.3 中的层次式死锁检测方案。Knapp[1987]和 Singhal[1989]提供了一个关于在分布式系统中进行死锁检测的调研报告。

Lamport 等[1982]和 Pease 等[1980]论述了 Byzantine generals 问题。Garcia-Molina[1982]给出了 bully 算法。Lann[1977]编写了针对环结构系统的选择算法。

第六部分　保护与安全

保护机制通过限制用户的文件访问类型许可提供受控制访问。此外，保护必须确保只有那些从操作系统获得了恰当授权的进程才可以操作内存段、处理器和其他资源。

由一个控制程序、进程或用户对计算机系统资源的访问机制提供保护。这个机制必须为强加的控制提供一种规格说明方法和一种强制执行方法。

安全要确保系统用户的验证，以保护系统的物理资源和系统存储的信息（包括数据和代码）的整体性。安全系统要防止未授权的系统访问、恶意地破坏或更改数据以及意外地引入不一致性。

第十八章 保 护

操作系统中的进程必须加以保护，使其免受其他进程活动的干扰。为此，系统采用了各种机制确保只有从操作系统中获得了恰当授权的进程才可以操作相应的文件、内存段、处理器和其他资源。

保护是指一种控制程序、进程或用户对计算机系统资源的访问的机制。这个机制必须为强加控制提供一种规格说明方法和一种强制执行方法。要区分保护和安全，安全是对系统完整性和系统数据安全的可信度的衡量。安全保障是一个比保护广泛得多的主题，将会在第十九章讨论这个主题。

18.1 保护目标

计算机系统越来越复杂，应用日益广泛，保护系统完整性的需求也随之增长。保护最初是多作业操作系统的附属产物，以便不受信任的用户可以共享一个公有的逻辑命名空间（比如文件目录），或者公有的物理命名空间（比如内存）。现代的保护观念提高了所有使用共享资源的复杂系统的可靠性。

要提供保护基于如下理由：首先，需要防止用户有意地、恶意地违反访问约束；另外一个普遍适用的理由是，需要确保系统中活动的程序组件只以同规定的策略一致的方式使用系统资源。

通过检测组件子系统接口的潜在错误，保护能够提高可靠性。早期检测接口错误通常能防止已经发生故障的子系统影响其他健康的子系统。一个未受保护的资源无法抵御未授权或不合格用户的访问（或误用）。面向保护的系统会提供辨别授权使用和未授权使用的方法。

保护在一个计算机系统中扮演的角色是：为加强资源使用的控制策略提供一种机制。可以通过各种途径建立这些策略。有些已经固化在系统设计中，有些会在系统管理中阐明。还有一些由个人用户定义，以保护他们自己的文件和程序。一个保护系统必须有一定的弹性，能够强制执行多种可向它声明的策略。

资源使用的策略可能会随应用改变，而且可能会随时间改变。基于以上原因，安全不再只是操作系统设计者所要关心的问题。应用程序员同样需要使用保护机制，保护应用子系统所创建和支持的资源，防止它们被误用。这一章描述的是操作系统需要提供的保护机制，应用程序设计者可以将这些保护机制应用到自己的保护软件中。

策略和机制不一样。机制决定*怎样做*，策略决定*做什么*。就弹性而言，分离策略和机制是很重要的。策略可能会随着位置和时间变化。最坏的情况是，策略中的每个变化都可能要求底层的机制做相应的变化。具有一般性的机制更可取，因为一个策略的变化可能只需要改动一些系统的参数或表格。

18.2 保 护 域

一个计算机系统是进程和对象的集合。*对象*分为**硬件对象**(如处理器、内存段、打印机、磁盘和磁带驱动器)和**软件对象**(如文件、程序和信号)。每个对象都有一个惟一的名字，这个惟一的名字将它所对应的对象和系统中的其他对象区分开来。用户只能通过定义好的有意义的操作来访问对象。对象是基本抽象数据类型。

这些操作可能要取决于对象。例如，处理器用于执行指令，对内存段的操作是读和写，而 CD-ROM 和 DVD-ROM 只能实现读取，磁带驱动器可以进行读、写和回转操作。数据文件可以被创建、打开、读、写、关闭和删除；程序文件可以被读、写、执行和删除。

进程只能访问那些已经获得了授权的资源。而且，在任何时候，进程只能访问完成现阶段的任务所需要的资源。这第二个要求通常被称为需要则知道(need-to-know)原则。一个有缺陷的进程可能会引起系统错误，而需要则知道原则可以有效地限制该类错误的发生次数。例如，如果进程 p 调用进程 A，在这个过程中，进程 A 只能访问它自己的变量和进程 p 传递给它的正式参数；它不能访问进程 p 的所有变量。同样道理，如果进程 p 调用一个编译器来编译一个特殊的文件，编译器也不能任意访问所有文件，而只能访问所有文件中一个定义好的子集(如源文件、名单文件以及其他)。相反地，编译器也有用做统计或优化目的私有文件，进程 p 也不能访问这些文件。

18.2.1 域结构

为了方便研究这个策略，假定一个进程只在一个**保护域**(protection domain)内操作，该保护域指定了进程可以访问的资源。每个域定义了一个集合。集合的元素是对象和运用于集合中每一个对象上的操作的类型。在一个对象上执行一个操作的权限是一种**访问权限**。一个域是一个访问权限的集合，每一个访问权限是一个有序对<*对象名*，*权利集合*>。例如，如果域 D 中有访问权限<*文件* F，{读，写}>，那么一个在域 D 中执行的进程就能读写文件 F；然而，进程不能对文件 F 执行任何其他的操作。

域之间允许存在交集；它们可以共享访问权限。例如，在图 18.1 中，有三个保护域：D_1、D_2，和 D_3。访问权限<O_4，{print}>是由 D_2 和 D_3 共享的，也就是说，运行在 D_2 或 D_3 上的任意一个进程都有打印对象 O_4 的权限。请注意，一个进程只有运行在 D_1 上时才能读写对象 O_1。另一方面，只有域 D_3 中的进程才能执行对象 O_1。

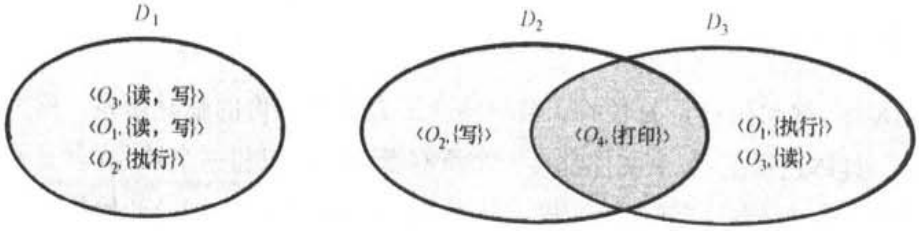


图 18.1 有三个保护域的系统

一个域和一个进程之间的关联可以是**静态**的，也可以是**动态**的。如果一个进程可获得的资源集合在进程的生命期固定不变，那么这种关联是静态的。创建动态保护域要比创建静态保护域复杂，这是预料中的事。

如果进程和域之间的关联固定不变，并且不想违反需要则知道原则，那么必须保证存在一个能够改变域的内容的机制。一个进程可能会有两个不同的执行阶段。例如，它可能在一个阶段需要读访问，在另一个阶段需要写访问。如果域是静态的，那么必须在域的定义中同时包含读访问和写访问。然而，在两个阶段中，这种安排都提供了过多的权限，因为在只需要写权限的阶段还拥有读权限，反过来也一样。因此，这里违反了需要则知道原则。必须允许修改域的内容，以便域能够随时反映所必须的最少的访问权限。

如果关联是动态的，则必须提供一个允许进程在域之间切换的机制。可能还是希望允许修改域的内容。如果不能更改一个域的内容，可以通过其他途径达到同样的效果。可以根据修改后的内容创建一个新域，然后在想更改域的内容时切换到这个新域。

一个域可以通过以下几种不同的途径来实现：

• 每个*用户*是一个域。这种情形下，可以访问的对象集取决于用户的身份。**域切换**动作在更换用户时发生——一般的情形是一个用户退出，另外一个用户登入。

• 每个*进程*是一个域。这种情形下，对象集的访问取决于进程的身份。当一个进程发送消息给另外一个进程时，由域切换给出响应，然后等待响应。

• 每个*过程*是一个域。在这种情形下，可以访问的对象集对应这个过程中所定义的局部变量。域切换动作发生于过程调用发生时。

在 18.3 节将讨论域切换的细节。

考虑一下操作系统执行的标准双模式(监控－用户模式)模型。当一个进程在监控模式下执行时，它可以执行特权指令并完全控制计算机系统。另一方面，如果进程在用户模式下执行，它只能调用非特权指令；结果，它只能在它自己预先定义好的内存空间执行。这两种模式保护了操作系统(在监控域执行)，使其免受用户进程(在用户域执行)的干扰。在一个多作业操作系统中，仅有两个保护域是不够的，因为还需要保证多个用户间互不干扰。因此，在这种情况下，需要一种更精巧的策略。接下来就举例解释一下这种策略，看看这两个极具影响力的操作系统——UNIX 和 MULTICS 是怎样实现这些原理的。

18.2.2 举例:UNIX

在 UNIX 下,域和用户是关联的。域切换会配合用户身份的临时切换。这个变动由文件系统完成。具体过程是:每个文件都有一个所有者身份标识和一个域位(就是通常所说的*设置用户 ID 位*(setuid bit))与它相关联。当一个用户(*用户 ID 为 A*)开始执行一个属于 B 的文件时,如果此时 B 的关联域位是*关闭*的,那么该进程的*用户 ID* 会被设置成 A;如果这个设置用户 ID 位是*开启*的,那么该进程的用户 ID 应该设置为文件的所有者:B。如果进程退出,这个临时的用户 ID 的变动也就随之结束。

在以用户 ID 为域定义的操作系统中,还采用了一些其他的方法来实现域切换操作,因为几乎所有的系统都需要提供这样一种机制。当需要给普通用户群体提供使用特权的便利时,需要用到这种机制。例如,用户不用自己写网络程序,也同样能访问网络。在这种情形下,UNIX 系统的做法是,将网络程序的设置用户 ID 位设置成开启状态,程序运行时就会改动用户 ID。用户 ID 会变成拥有网络访问特权的的用户(如 *root*,最强大的用户)。这个方法中存在一个问题:如果一个用户成功地以用户 ID *root* 创建了一个文件并且让它的设置用户 ID 位处于*开启*状态,那么这个用户可能会变成 *root* 用户,从此他可以对系统执行任意操作。在附录 A 中将更深入地讨论设置用户 ID 这个机制。

这个方法在其他操作系统中的相对应的做法是将特权程序存放在一个特殊的目录下。运行这个特殊目录下的任意一个程序时,操作系统都会将程序的用户 ID 更改成与 *root* 等价的用户 ID 或者拥有这个目录的用户 ID。这种方法将所有这样的程序存放在同一个位置,这样就解决了秘密设置用户 ID 程序的设置用户 ID 问题。然而,跟 UNIX 采取的方法相比,这个方法缺少弹性。

系统只要简单地禁止更改用户 ID,就会变得更具约束力,也更加安全。在这些实例中,必须采用特殊技术为用户提供访问特权的便利。例如,一个**监护进程**可能在系统启动时就开始以特殊用户 ID 运行。然后用户运行一个单独的程序,当他们需要使用这些特权便利时就向监护进程发送请求。操作系统 TOPS-20 采用的就是这种方法。

任意系统在写特权程序时都必须分外小心。任何一个疏忽大意都可能让系统完全丧失保护。一般来说,这些程序是企图入侵系统的用户首先要攻击的对象;遗憾的是,这种攻击的成功几率很高。例如,很多 UNIX 系统的安全都因为设置用户 ID 这个特性而被破坏了。将在第十九章讨论安全。

18.2.3 举例:MULTICS

MULTICS 系统将保护域组织成一个环状继承结构。每个环对应一个单独的域(图 18.2)。这些环按顺序用数字 0~7 编号。D_i 和 D_j $(0 \leqslant i, j \leqslant 7)$ 为任意两个域:如果 $j < i$,那么 D_i 是 D_j 的一个子集。也就是说,在 D_j 中运行的进程比在 D_i 中运行的进程拥有更多特

权。一个在 D_0 中执行的进程拥有最多特权。如果只存在两个环，那么这个策略就等价于监控－用户执行模式，监控对应 D_0，而用户对应 D_1。

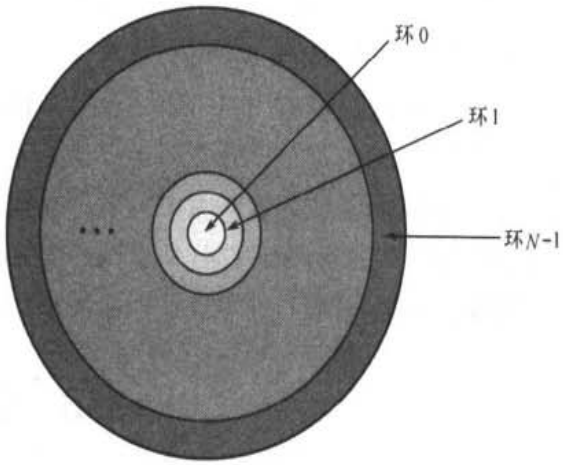


图 18.2 MULTICS 的环结构

MULTICS 有一个分段的地址空间，每个段是一个文件。每个段和这 8 个环中的一个相关联。一个段描述包含一个标识环编号的条目。此外，它还有 3 个访问位用来控制读、写和执行。段和环之间的关联是一个策略决策，本书不涉及这部分内容。每个进程都有一个“*当前环编号*”计数器和它关联，标识该进程目前所在的域。当 $j<i$ 时，如果一个进程在环 i 中执行，那么它不能访问和环 j 相关联的段。然而，如果 $k\geqslant i$，那么它可以访问和环 k 相关联的段。访问类型由和段相关联的访问位决定。

在 MULTICS 下，当进程调用一个不在同一个环的过程时，它必须跨越环，此时域切换动作就会发生。很显然，这种切换必须在受控方式下完成；否则，一个进程可以在环 0 中开始执行，这样的话系统无法提供任何保护。为了实现受控的环切换，需要修改段描述中关于环的部分，具体包括以下三个方面：

- **访问对**：一个有序整数对 (b_1, b_2)，并且 $b_1 \leqslant b_2$。
- **限制**：存在一个整数 b_3 满足条件：$b_3 > b_2$。
- **条目列表**：标识可能调用段的条目点（或门）。

如果一个在环 i 上执行的进程调用一个访问对为 (b_1, b_2) 的过程或者段，如果 $b_1 \leqslant i \leqslant b_2$ 成立，那么这个调用是合法的，并且进程的当前环的编号仍然是 i。否则，进程会向操作系统抛出一个陷阱，处理的情形如下：

- 如果 $i<b_1$，那么允许访问，因为现在是要迁移到一个特权更少的环（或者域）。然而，如果传递的参数涉及编号更低的环里边的段，那么必须把这些段拷贝到一个被调用过程可以访问到的位置上。
- 如果 $i>b_2$，那么只有当 $b_3 \leqslant i$ 成立时才允许调用，并且调用被定向到条目列表中一个指定的条目点上。这个策略允许一个只有有限访问权限的进程调用一个在编号较低的环中的、比调用者拥有更多访问权限的过程，但是这种调用是在一个谨慎控制的方式下进行的。

环（或继承）结构的主要不足之处在于它不允许强制执行需要则知道策略。特别地，如果要使得一个对象在域 D_j 中必定可以访问，而在域 D_i 中必定不可以访问，那么要满足条件 $j<i$。这个要求意味着每个在 D_i 中可以访问的段在 D_j 中都可以访问。

跟当前的操作系统相比，MULTICS 的保护系统通常比较复杂，并且效率较低。如果保护接口让系统的使用变得复杂，或者会大幅度降低系统的性能，那么就必须在使用安全和系统目的之间慎重选择。比如，有一台被学校用来处理学生成绩、同时被学生用来完成课程作业的计算机，你想在这台计算机上安装一个复杂的保护系统。对于一台用做大量处理数据的计算机来说，类似的保护系统是不合适的，因为此时性能是最重要的。笔者倾向于从保护策略中分离出机制，允许同样的系统根据用户的需求采用或繁或简的保护。为了要将机制从策略中分离，需要一些更普通的保护模型。

18.3 访问矩阵

可以将保护模型抽象为一个矩阵，称之为**访问矩阵**(access matrix)，矩阵的行代表域，矩阵的列代表对象。矩阵的每个条目是一个访问集合。由于列明确地定义了对象，可以在访问权限中删除对象名称。访问条目(i,j)定义了在域 D_i 中执行的进程在调用对象 O_j 时被允许执行的操作的集合。

如图 18.3 所示，该访问矩阵中有 4 个域和 4 个对象，其中 3 个对象都是文件(F_1，F_2，F_3)，剩下的一个是激光打印机。进程在域 D_1 中执行时，它可以读文件 F_1 和 F_3。进程在域 D_4 中执行时拥有和在域 D_1 中执行时一样的特权，但除此之外，它还可以写文件 F_1 和 F_3。请注意，只有在域 D_2 中执行的进程才可以访问激光打印机。

域 \ 对象	F_1	F_2	F_3	打印机
D_1	读		读	
D_2				打印
D_3		读	执行	
D_4	读、写		读、写	

图 18.3 访问矩阵

访问矩阵策略提供了一个指定多样化策略的机制。这个机制包括两个方面的内容，一是实现访问矩阵，二是确保维持在提纲中提及的语义属性。具体地说，必须确保在域 D_i 中执行的进程只能访问在行 i 中指定的对象，就跟访问矩阵条目定义的一样。

访问矩阵可以实现保护相关的策略决策。这个策略决策包括条目(i,j)中应当包含哪些权限。还必须确定每个进程执行时所在的域。最后的这条策略通常由操作系统决定。

通常由用户决定访问矩阵条目的内容。当用户创建一个新的对象 O_j 时，列 O_j 就被添

加到访问矩阵，并根据创建者的指示恰当地初始化条目。用户可能会根据需要在列 j 中添加一些权限，并在别的条目中另外添加一些权限。

前文曾提到进程和域之间的静态和动态关联，访问矩阵为它们提供了一种定义和实现严格控制的机制。当需要将一个进程从一个域切换到另外一个域时，其实是在一个对象（域）上执行一个操作（切换）。可以将域作为对象添加到访问矩阵，这样就可以控制域切换了。同样道理，当更改访问矩阵的内容时，也是在对象——访问矩阵上执行某项操作。可以将访问矩阵本身作为一个对象，这样就可以控制这些变化了。实际上，因为矩阵对象中的每个条目都可能被单独修改，必须考虑将访问矩阵中的每个条目当做一个对象来保护。

现在，只需要考虑这些新对象上的可能操作，并决定进程应该如何执行这些操作。

进程必须能够在域之间切换。当且仅当访问权限 switch∈access(i,j) 时，才允许从域 D_i 到域 D_j 的切换发生。因此，在图 18.4 中，一个在域 D_2 中执行的进程可以切换到域 D_3 或域 D_4。一个域 D_4 中的进程可以切换到域 D_1，而域 D_1 中的进程可以切换到域 D_2。

域 \ 对象	F_1	F_2	F_3	激光打印机	D_1	D_2	D_3	D_4
D_1	读		读			切换		
D_2				打印			切换	切换
D_3		读	执行					
D_4	读、写		读、写		切换			

图 18.4 将域作为对象的图 18.3 的访问矩阵

要为访问矩阵的条目内容提供受控更改还需要三个额外的操作：拷贝（copy）、所有者（owner）和控制（control）。

从访问矩阵的一个域（行）中拷贝一个访问权限到另外一个域，这种权限用附加在访问权限后面的“*”标记。*拷贝*权限只允许在定义了该权限的列内拷贝访问权限。例如，在图 18.5(a)中，一个在域 D_2 中执行的进程可以将读操作拷贝到任意与文件 F_2 相关联的条目。因此，图 18.5(a)中的访问矩阵可以被修改为图 18.5(b)中的访问矩阵。

这种策略有两种变形：

1. 将一个权限从访问对(i,j)拷贝到访问对(k,j)，然后将这个权限从 access(i,j)中删除；这种行为并不是拷贝一个权限，而是*迁移*一个权限。

2. *拷贝*权限的传播是受限制的。也就是说，从 access(i,j)拷贝权限 R^* 到 access(k,j)，实际上被创建的权限是 R（而不是 R^*）。一个在域 D_k 中执行的进程不可以进一步拷贝权限 R。

域 \ 对象	F_1	F_2	F_3
D_1	执行		写*
D_2	执行	读*	执行
D_3	执行		

(a)

域 \ 对象	F_1	F_2	F_3
D_1	执行		写*
D_2	执行	读*	执行
D_3	执行	读	

(b)

图 18.5 带拷贝权限的访问矩阵

一个系统也许会只选择这三种拷贝权限的方法中的一种，也许会同时提供三种方法，将它们标记为三种分离的权限：*拷贝*、*迁移*和*受限的拷贝*。

同样需要提供这样一种机制，它允许添加新的权限或者删除已有权限。由*所有者*权限控制这些操作。如果 access(i,j)包含*所有者*权限，那么一个在域 D_i 执行的进程就可以添加和删除列 j 中的任意一个权限。例如，在图 18.6(a)中，域 D_1 是 F_1 的所有者，因此域 D_1 可以在列 F_1 中任意添加或删除合法权限。同样道理，域 D_2 是 F_2 和 F_3 的所有者，因此可以在这两列中任意添加或删除合法权限。因此，图 18.6(a)中的访问矩阵可以被修改为图 18.6(b)中的访问矩阵。

域 \ 对象	F_1	F_2	F_3
D_1	所有者 执行		写
D_2		读* 所有者	读* 所有者 写*
D_3	执行		

(a)

域 \ 对象	F_1	F_2	F_3
D_1	所有者 执行		
D_2		所有者 读* 写*	读* 所有者 写*
D_3		写	写

(b)

图 18.6 有所有者权限的访问矩阵

有了*拷贝*和*所有者*权限之后，进程就可以在一个列中修改条目。还需要一个允许在行中修改条目的机制。*控制*权限只适用于域对象。如果 access(i,j)包含*控制*权限，那么在域 D_i 中执行的进程可以从行 j 中删除任意一个访问权限。例如，假设在图 18.4 中，access(D_2,D_4)中包含*控制*权限。那么，一个在域 D_2 执行的进程就可以修改域 D_4，如图 18.7 所示。

*拷贝*和*所有者*权限提供了一个限制访问权限传播的机制。然而，它们并没有提供一个合适的工具用来防止信息传播(或泄露)。有一个问题被称为**限制问题**，它说的是要确保一个对象最初持有的信息不能迁移到对象的执行环境之外。这个问题其实是一个不可解问题。

这些域和访问矩阵上的操作的重要之处不在于它们自身，重要的是它们展现了访问矩阵模型的权限，满足了实现和控制动态保护需求。可以在访问矩阵模型中动态地创建并包

域 \ 对象	F_1	F_2	F_3	激光打印机	D_1	D_2	D_3	D_4
D_1	读		读			切换		
D_2				打印			切换	切换控制
D_3		读	执行					
D_4	写		写		切换			

图 18.7 修改图 18.4 中的访问矩阵

含新的对象和域。这里只讨论基本机制，具体需要访问哪些域、需要通过怎样的途径创建对象等，这些问题都是系统设计者和用户需要考虑并做出决策的。

18.4 访问矩阵的实现

怎样才能有效地实现访问矩阵呢？一般情况下，访问矩阵是一个稀疏矩阵；也就是说大多数条目都是空的。虽说表示稀疏矩阵的数据结构技术已经很成熟了，但由于保护的使用方式的特殊性，这些技术在这里并不是很合适。

18.4.1 全局表

实现访问矩阵的最简单的方式是采用一个全局表，这个表是一个有序三元关系<*域*，*对象*，*权限集合*>的集合。任何时候，当一个操作在域 D_i 中作用于对象 O_j 时，系统就在全局表中查找三元关系$<D_i, O_j, R_k>$，查找条件是 $M \in R_k$。如果找到了这个三元关系，那么操作可以继续，否则，将会产生一个异常或者错误。这种实现有一些缺陷。这张表通常很大，以至于无法将整张表存放在内存中，所以需要额外的 I/O 开销。通常会采用虚拟内存技术来管理这张表。此外，也很难利用对象和域的奇特的分组优势。例如，如果要让每个用户都能读取一个特定对象，那么这个对象必须在每个域都有一个单独的条目。

18.4.2 对象的访问列表

访问矩阵中的每个列都可以被实现成一个对象的访问列表，具体描述见 11.6.2 节。显然，可以抛弃那些空的条目。最后，每个对象的列表由有序对<*域*，*权限集*>组成，这些有序对为该对象定义了所有有着非空访问权限集合的域。

可以对这个方法作一些扩充，定义一个列表和一个缺省的访问权限集合。当用户想在域 D_i 中对对象 O_j 执行操作 M 时，系统开始在访问权限列表中为对象 O_j 查找条目$<D_i, R_k>$，查

找条件是 $M \in R_i$。如果查找成功，那么操作可以继续；否则查找缺省的集合。如果 M 在缺省集合中，那么允许访问。否则，禁止访问并引发异常状态。为了提高效率，通常是在查找缺省集合失败之后才查找访问列表。

18.4.3 域的权限列表

前文中将访问矩阵的列和对象相关联，由此设计了访问列表，除此之外，还可以将每个行关联到它的域。域的**权限列表**由对象以及允许作用于这些对象上的操作组成。一个对象通常用它自己的物理名称或者地址表示，被称为**权限**。如果要对对象 O_j 执行操作 M，由进程来执行这个操作，参数为对象 O_j 的权限（或指针）。对权限的简单**拥有**意味着访问是被允许的。

权限列表被关联到一个域，但在该域中执行的进程并不能直接地访问它。更确切地说，权限列表本身是一个被保护的对象，由操作系统持有，并被用户以间接的方式访问。基于权限的保护依赖于以下事实：即决不允许这些权限迁移到任意一个用户进程可以直接访问的地址空间。如果所有的权限都是安全的，那么它们所保护的对象也是安全的，所有未经授权的用户都不能访问这个对象。

权限最初被定位为一种安全指针，目的是满足多作业计算机系统对资源保护的需求。这个内在保护指针的想法（从系统用户的角度来看）为扩展到应用层次的保护提供了一个很好的基础。

为了提供内在保护，必须将权限和其他的对象区分开，并用一个抽象机器来解释权限，这个抽象机器必须能够运行高级语言程序。通常采用以下两种途径来区分权限和其他数据：

- 每个对象都有一个标识自身类型（权限/可访问数据）的**标志**。应用程序本身不能直接访问这些标志。可能会用硬件或者防火墙支持来强制执行这些约束。虽然一个位足以区分容量和其他对象，但通常会使用多个位。这样扩展之后，硬件就可以将所有的对象都标记上类型信息。这样，硬件就可以区分整数、浮点数、指针、布尔变量、字符、指令、权限和未初始化的标识值。
- 另一种选择是：将一个程序所关联的地址空间分裂成两部分。一部分存放着程序的普通数据和指令，程序可以直接访问这个部分。另一部分存放着权限列表，只有操作系统才可以访问。以段组织的内存空间（见 9.5 节）对这种方式提供了很好的支持。

业内已经开发了一些基于权限的保护系统；18.6 节有简要描述。操作系统 Mach 也采用了基于权限的保护，附录 B 对此有描述。

18.4.4 锁—钥匙机制

锁—钥匙机制（lock-key scheme）是访问列表和权限列表之间的一个折中。每个对象都有一个由具有惟一性的位模式组成的列表，称为锁。类似地，每个域都有一个由具有惟一性

的位模式组成的列表，称为钥匙。仅当域拥有一把打开那个对象的某把锁的钥匙时，在该域执行的进程才可以访问相应的对象。

与权限列表的情形一样，操作系统代替域管理域的钥匙列表。用户不能直接地检查或者修改钥匙列表。

18.4.5 比较

访问列表直接和用户需求挂钩。当用户创建一个对象时，他可以指定哪些域可以访问该对象，可以对对象执行哪些操作。然而，一个特定域的访问权限信息不是局部的，因此判定每个域的访问权限集合是非常困难的。此外，每次访问对象都要检查，需要查找访问列表。对于一个庞大的并且有着很长的访问列表的系统来说，这种查找是相当费时的。

权限列表不是直接和用户需求挂钩的，然而它们对一个指定进程的局部信息而言非常有用。要访问的进程必须为访问准备一个权限。然后，保护系统只需要确认权限的合法性。然而，撤销权限的操作可能会失败。

锁－钥匙机制是上文所提及的两种机制的折中。这个机制可以同时兼有有效性和弹性，这取决于钥匙的长度。钥匙可以在域之间自由地传递。此外，只需要一个简单技术，即更改与对象相关联的锁当中的部分锁，就可以有效地撤销访问特权。

绝大多数系统采用的是访问列表和权限的结合体。当一个进程首次访问一个对象时，系统搜索访问列表。如果访问被拒绝，则会产生一个异常状态。否则，创建一个权限并将它附加给进程。额外的授权使用权限快速地表明允许访问。最后一次访问结束后，权限被销毁。MULTICS 和 CAL 系统都采用这种策略；这两个系统都同时采用访问列表和权限列表。

例如，考虑这样一个文件系统，该系统的每一个文件都有一个关联的访问列表。当一个进程打开一个文件时，系统通过搜寻目录结构查找文件，检查访问许可，并分配缓冲区。所有这些信息都被记录到文件表中和该进程相关的那个条目。这个操作会返回一个在表中指向这个新打开的文件的索引。所有对该文件的操作都由指向该文件表的索引的详细描述产生。然后这个文件表中的条目指向文件和分配给文件的缓冲区。文件关闭后，这项文件条目会被删除。因为文件表是由操作系统维持的，用户不可能意外地损坏它。因此，用户只能访问已经打开的文件。由于在打开文件时要检查访问权限，这一行为提供了有效的保护。UNIX 系统采用了这种策略。

每次访问时，仍然必须检查访问权限，并且文件表的条目有一个仅被允许执行的操作设置的权限。如果打开文件用于读操作，那么就在文件表的条目存放一个读访问的权限。如果企图写该文件，系统会通过比较要执行的操作和文件表条目的权限来判断操作有没有违反保护。

18.5 访问权限的撤回

在动态保护系统中，有时需要将不同用户共享的访问权限撤回给对象。撤回对象访问权限时可能会引发各种问题：

• **即时撤回还是延时撤回**：是立刻撤回访问权限，还是延迟到某个时机再撤回？如果是延迟撤回，那么能找到执行撤回动作的恰当时机吗？

• **选择性的还是一般性的**：当撤回一个对象的访问权限时，这个撤回动作是针对拥有这个对象的访问权限的所有用户呢，还是有选择地指定一个用户群体执行撤回操作？

• **部分的还是所有的**：可以只撤回同一个对象相关联的部分权限吗？还是必须撤回同该对象相关联的所有权限？

• **暂时的还是永久的**：访问权限可以被永久地撤回吗（就是说，撤回的访问权限永远不会再被使用）？还是说可以在撤回访问之后再次获得访问权限？

如果采用访问列表策略，那么撤回是个很简单的问题。首先在访问列表中搜索要撤回的访问权限，然后从列表中删除这些权限。访问列表策略中的撤回是立刻执行的，可以是一般性的也可以是选择性的，可以是所有的也可以是部分的，可以是永久的也可以是暂时的。

对于权限来说，撤回问题就麻烦许多。权限分布在整个系统中，在撤回之前必须先找到它们。实现权限撤回的策略如下：

• **重新获得**：系统将周期性地从每个域中删除权限。当一个进程要使用一个权限时，它可能会发现所需的权限已经被删除了。然后进程会尝试重新申请这个权限。如果访问已经被撤回的话，进程将无法重新获得权限。

• **折回指针**：每个对象都维护着一个指针列表，这些指针指向与该对象关联的所有权限。需要执行撤回操作时，系统就追踪这些指针，根据需要更改权限。MULTICS 系统采用了这一策略。尽管这个策略执行时要消耗很多资源，还是有许多系统采用它。

• **间接**：权限指针是间接地指向对象的。每个权限指针都指向一张全局表中的一个具有惟一性的条目，而该条目转而指向对象。撤回操作分两步执行：首先在那张全局表中查找对应的条目，找到后删除该条目。如果有访问想要访问该权限，将会发现它指向表中一个不合法的条目。因为权限和表条目都拥有那个对象的独一无二的名字，所以其他权限很容易重用那个表条目。权限所对应的对象和表条目必须匹配。CAL 系统采用了这种策略。这种策略不允许选择性地撤回。

• **钥匙**：钥匙是一个可以关联每个权限的独一无二的位模式。这个钥匙是在创建权限的时候定义的，拥有这个权限的进程不能更改和检查它。关联到每个对象的**主钥匙**可以通过设置钥匙操作进行定义和修改。创建一个权限时，主钥匙的当前值是和权限相关的。当权限被使用时，系统会比较权限的值和主钥匙的值。如果这两个值匹配，操作可以继续；否

则就产生一个异常状态。撤回操作通过设置钥匙操作将主钥匙的值替换成一个新的值，这样，之前与该对象相关联的所有权限都失效了。这个策略不允许选择性地撤回，因为每一个对象只有一个主钥匙和它相关联。如果一个对象和一个钥匙列表关联，那么就可以实现选择性撤回了。最后，可以将所有的钥匙组合成一个全局的钥匙表。权限只有在它的钥匙和表中的某个钥匙匹配时才是合法的。在这种情形下，只要从表中删除匹配的钥匙，就能成功地实现撤回。在这种策略下，一个钥匙可以和多个对象关联，并且多个钥匙也可以关联到一个对象，这提供了最大限度的弹性。

在基于钥匙的策略中，不应让所有用户都有执行定义钥匙、将钥匙插入到列表、从列表中删除钥匙等操作的权限。特别地，应该只有对象拥有者才能为对象设置钥匙，这也是合理的。然而，这个选择是一个策略决策，操作系统可以实现它但不能定义它。

18.6 基于权限的系统

这一节，来看两个基于权限的保护系统。这些系统在复杂性和实现策略的类型方面有所区别。这两者的应用都不是很广泛，但很有意义，它们为保护理论的研究提供了良好的基础。

18.6.1 举例：Hydra

Hydra是一个基于权限的保护系统，它有很大的灵活性。这个系统提供了一个固定的访问权限集合，集合中包含了大多数可能用到的访问权限，系统知道这些访问权限并负责解释它们。这些访问权限包括基本的访问形式，如读权限、写权限或者操作一个内存段的权限。此外，系统为用户提供了一些让用户声明额外权限（除系统提供的权限外的权限）的方法。只有用户自己的程序才可以解释用户定义的权限，但是系统为这些权限的使用提供了访问保护，同时也包括系统定义的权限。这些便利使得保护技术的发展上了一个很大的台阶。

对对象的操作是由程序定义的。这些实现操作的程序本身也是某种形式的对象，权限采取间接的方式访问这些特殊对象。如果系统要处理用户定义的类型的话，那么它首先要确认这些用户定义程序的名称。当系统被告知一个对象的定义时，对这个类型的操作的名称就成了一个**辅助权限**。辅助权限在权限中可以被描述为一个类型实例。一个进程如果要对某个类型的对象执行某项操作，那么它所持有的那个对象的权限必须包含该操作在它的辅助权限当中被调用的那个操作的名称。这个限制导致访问权限间出现不公平现象，实例和实例间，进程和进程间都存在差异。

Hydra还提供了**权限扩展**。这个策略允许一个程序证明为在操作一个特定类型的正式参数上是*值得信赖*的，主要是保障了所有拥有执行这个程序权限的进程的利益。值得信赖的程序所持有的权限独立于，并且可能超出负责调用它的进程所持有的权限。然而，不能认为这样一个程序在全局范围内都是值得信赖的，并且不能将这个信赖扩展到任意一个可能

被一个进程执行的过程或者程序段。

扩展允许实现过程访问一个抽象数据类型的各种表示方式。例如，一个进程持有一个类型对象 A，那么这个权限可能包含一个调用某些操作 P 的辅助权限，但不会包含所谓的核心权限，比如读、写、执行代表 A 的段。这样的权限给进程提供了一个间接访问 A 的表示式的途径，但仅限于某些特定目的。

从另一方面看，如果一个进程调用一个对对象 A 的操作 P，当控制被传递给 P 的代码体时，可能会对访问 A 的权限做适当的扩展。这个扩展可能有必要让 P 拥有访问代表 A 的存储段的权限，以实现 P 在抽象数据类型上定义的操作。虽然调用进程不能直接访问 A 的段，P 的代码体却可能可以直接读、写 A 的段。在 P 返回时，A 的权限被重新设回到原来的值，即未扩展前的状态。一个进程所持有的访问受保护段的权限需要动态更新，这种情形是很常见的，这取决于要完成的任务。动态调整权限的目的在于保证程序员定义的抽象的一致性。可以在一个抽象数据类型的声明处显式地向操作系统 Hydra 声明权限扩展。

当用户将一个对象作为一个参数传递给一个过程时，可能需要保证这个过程不会修改该对象。实现这个约束并不难，只要传递一个没有修改权限的访问权限就可以了。然而，如果扩展发生的话，就有可能恢复修改权限。因此，可能会无法保障用户的保护需求。当然，一般地说，用户可以相信一个过程会正确地完成它的任务。然而，由于各种软/硬件错误的存在，这个假设不可能总是成立的。Hydra 通过约束扩展解决了这个问题。

Hydra 的过程调用机制被设计成一个直接解决双向怀疑子系统问题的方案。以下是这个问题的定义。假设提供了这样一个程序，它可以作为一项服务同时被几个用户调用。当用户调用这个服务程序时，他们是要冒风险的：程序可能发生错误；可能毁坏给定的数据；可能保留一些稍后要在别处用到的数据的访问权限。同样地，服务程序会有一些私有文件，负责调用的用户程序不能直接访问这些文件。Hydra 提供了直接处理这个问题的机制。

一个 Hydra 的子系统建立在它的保护核心之上，并且可能需要保护它自己的组件。子系统通过调用一个核心定义的原型集合来与核心沟通，这些原型定义了对系统资源的访问权限。用户进程使用这些资源的策略可以由子系统设计者来定义，但他们被强制使用权限系统提供的标准访问保护。

在阅读参考手册、熟悉系统的特性之后，程序员可以直接利用保护系统。Hydra 提供了一个庞大的由系统定义的过程库，用户程序可以调用这些过程。在使用 Hydra 系统时，用户可以将系统过程的调用显式地合并到他自己的程序中，他也可以使用连接到 Hydra 的程序转换器。

18.6.2 举例：剑桥 CAP 系统

剑桥 CAP 系统的设计采用了另外一种面向权限的保护方法。CAP 的权限系统比较简单，远不如 Hydra 强大。然而，最近的调查显示：它同样可以为用户定义的对象提供安全保

护。CAP 有两种权限。普通的那种被称为**数据权限**。它被用来为对象提供访问权限，但只限于标准的读、写和执行与对象相关连的私有存储段等权限。由 CAP 机器的微代码负责解释数据权限。

第二种权限被称为**软件权限**，CAP 的微代码只为这种权限提供保护，但不负责解释。它的解释由一个受保护的过程负责，这个过程可能会被应用程序员写入子系统。一个特定类型的权限扩展是和一个受保护的过程相关联的。当执行这样一个过程的代码体时，进程会临时获得读写这个软件权限内容的权限。这个特定类型的权限扩展等价于一个权限的封闭和非封闭原型的实现。当然，这种特权是从属于权限的类型检查的，确保只有特定抽象类型的软件权限才可以被传递给任意一个这样的过程。通用的信任被存放在 CAP 机器的微代码中。

软件权限的解释由子系统全权负责，由它包含的受保护的过程来完成。这个策略允许实现多样化的保护策略。虽然一个程序员可以定义自己的受保护过程，但是不能保证系统整体的安全性。基本的保护系统不允许一个未经检验的、用户定义的受保护过程去访问任何不属于该过程所处保护环境的存储段。一个不安全的受保护过程所能带来的最严重的后果是：该过程负责的子系统发生保护故障。

CAP 系统的设计者已经意识到软件权限应用所蕴含的巨大商机，接着要做的是根据抽象资源的需求恰当地规范和实现保护策略。然而，如果一个子系统设计者要利用这些工具，仅仅阅读参考手册是不够的，这一点和 Hydra 系统不太一样。系统不提供过程库，他必须学习保护的规律和技术。

18.7 基于语言的保护

就现有计算机系统提供的保护而言，通常是由操作系统核心来完成的，操作系统核心扮演一个代理检查和核准一个访问受保护资源的请求的角色。综合的访问核查是一个巨大的却又无法避免的潜在开销来源，要么为它提供硬件支持以减少每次核查的开销，要么降低对保护的要求。如果已有的支持机制已经限制实现保护策略的弹性，或者所保护环境超出了所必需的确保更好操作效率的话，那么很难同时满足所有目标。

操作系统日益复杂了，特别是要提供更高层次的用户接口，保护的目标也随之精练了许多。保护系统的设计者现在开始借用源于程序语言的概念，特别是抽象数据类型和对象这两个概念。保护系统的考虑现在已不再局限于确认要访问的资源，它还考虑要访问的功能性质。在最近的保护系统中，对要调用的功能的考虑已经延伸到系统定义的功能集合之外，如标准的文件访问方法，还包含了用户定义的功能。

资源使用的策略也会发生变化，这取决于应用，它们也可能会随时间变化。基于以上原因，保护不再只是操作系统设计者要考虑的问题。它应该成为一个应用程序设计者的工具，

这样应用子系统的资源就可以免受外界干扰和错误的影响。

18.7.1 基于编译程序的强制

程序语言开始进入了视线。要指定对系统共享资源的访问控制权限，只需要为资源做一个声明陈述。只要扩展语言的用法，这种陈述就可以集成到语言。当保护和数据使用方法一起被声明时，子系统的设计者可以指定它的保护需求，还有它对系统中其他资源的需求。应该在写程序时直接指定以上内容，并且采用程序本身使用的语言。这种方法有明显的优势：

1. 保护需求只要简单声明就可以了，不需要调用一系列操作系统的过程。
2. 保护需求可以用特定操作系统提供的工具独立地声明。
3. 子系统的设计者不需要提供强制执行的方法。
4. 访问特权和数据类型的语言概念有着密切联系，因此它的声明符号很自然。

编程语言实现可以提供各种技术来强制执行保护，但所有这些都会在一定程度上依赖于底层机器和它的操作系统的支持。例如，假设采用某种语言来产生在剑桥 CAP 系统上运行的代码。在这个系统中，底层机器上的内存引用是通过一个权限间接发生的。这个约束时刻防止所有进程访问自身保护环境之外的资源。然而，一个程序可能会强加一些专断的约束，限制某些资源在一个特定代码段的执行期间的使用方式。采用 CAP 提供的软件权限，可以很轻松地实现这些约束。软件权限会实现那些在语言中指定的保护策略，一个语言实现要提供标准的保护过程来解释这些软件权限。这个方法将策略指定这件事交付给程序员，同时不再要求他们实现强制的保护。

虽然这种系统提供的安全核心不如 Hydra 或者 CAP 强大，但还是提供了足够的机制来实现程序语言给定的保护规则。主要的差别在于这个保护的安全性不如保护核心提供的那样强大，原因是这种机制必须更多地依赖系统运转状态。编译器可以区分确定会发生保护违例和不会发生保护违例的参考内容，并将它们区别对待。这种方式的保护提供的安全要依赖于这样一个假设：这个编译器产生的代码在执行前和执行期间不会被修改。

同主要由编译器提供的强制相比，仅基于核心的强制的相对优势是什么？

• **安全**：和编译器的保护一检查代码生成所提供的强制相比，核心提供的强制给保护系统自身提供了更强的安全性。在一个支持编译器的策略中，安全要依赖于以下几个方面：翻译器的正确性，一些底层的内存管理机制，它保护那些段以免受已编译代码运行所在段的影响，还有程序装载时的源文件的安全性。这其中的一些因素也可以应用到支持软件的保护核心中，但核心可能被固定存储到物理存储段中，并且可能要从一个指定的文件中装载，由于这些限制，应用要打折扣。在一个带标记的权限系统中，所有的地址计算要么由硬件完成，要么由一个固定的微程序完成，再大的安全都是可能的。对于硬件或系统软件的故障可能引发的保护侵犯，支持硬件的保护也有着比较强的免疫力。

• **灵活性**：虽然保护核心为系统提供了足够的工具来为系统自身的策略提供强制，但它在实现用户自定义的策略时，在灵活性方面有一些局限性。对编程语言来说，可以由实现根据实际需要声明保护策略并提供强制执行。如果某种语言不能提供足够的灵活性，那么可以扩展这种语言，或者做一些替换动作，这种做法在系统服务方面引发的混乱可能会比修改操作系统核心引发的混乱小一些。

• **效率**：如果硬件直接支持保护的强制执行，那么效率是最高的。然而需要软件支持的保护，基于语言的强制执行有着自己的优势：静态访问强制可以在编译期间离线验证。而且，一个智能编译器会根据特定的需要裁剪强制机制，可以避免原本不可或缺的核心调用的固定开销。

总之，在编程语言中指定保护，使资源的分配和使用策略的高层描述成为可能。当没有自动支持硬件的检查时，语言实现可以为保护强制提供软件。此外，它可以解释保护规格，产生面向硬件和操作系统提供的保护系统的调用。

为应用程序提供保护的一种方法是通过使用软件权限，它可以被作为计算的一个对象。这蕴含着这样一个观念：某些特定的编程组件可以拥有创建和检查这些软件权限的特权。创建权限的程序可能可以执行这样一个原子操作：密封一个数据结构，使得任何未持有密封和解封特权的程序组件都不能访问该数据结构。这些程序组件可以拷贝该数据结构的内容，或将它的地址传递给其他的程序组件，但不能访问它的内容。引入这些软件权限的目的是为编程语言引入一个保护机制。这个概念的惟一缺陷是：密封和解封操作的使用需要用一个过程式的方法来指定保护。要让应用程序员能够使用保护的话，一个非过程式的或者声明式的符号表示法是一个更可取的方法。

要将权限分配给用户进程的系统资源，需要的是一个安全、动态的访问一控制机制。要想对系统整体的可靠性有所贡献，访问控制机制在使用时必须保证安全。要有实用性的话，它还应该提供合理的效率。这个要求促进了一些语言结构的发展，这些语言结构允许程序员在使用指定的被管理资源时声明各种限制。这些结构为三种功能提供机制：

1. 在客户进程中安全而有效地分配权限：特别地，机制要确保这一点，即一个用户进程只有在获得该资源的一个权限时才能使用这个被管理资源。

2. 指定一个特定进程在分配资源时可能调用的操作的类型（例如，一个文件的读者只能读文件，一个文件的写者只能写文件）；不需要将同样的权限集合赋给所有的用户进程；除非有访问控制机制的授权，否则进程不能扩大自己的访问权限集合。

3. 指定一个特定进程调用某项资源的各种操作时要遵循的顺序（例如，文件只有先打开才能读）；两个进程可以在调用分配到的资源的操作顺序上有不同的约束。

将保护概念结合到编程语言中，作为一个系统设计的实用工具，尚处于萌芽阶段。有着分布式体系结构并且数据安全性需求日益迫切的新系统的设计者可能要开始更多地考虑保护问题。然后，恰当地表述保护需求的语言符号表示法的重要性可能会引起更广泛的关注。

18.7.2 Java 2 的保护

Java 是来自 Sun Microsystem 公司的面向对象编程语言。Java 程序由**类**组成；每个类是一个数据域和操作该数据域的函数(叫做**方法**)的集合。Java 虚拟机会响应创建类实例的请求：装载这个类。Java 最为新颖、实用的特性之一是：它支持从一个网络中动态地装载不可信任的类；支持在同一个 JVM 中执行互不信任的类。

鉴于 Java 的这些能力，保护成为一个首先要考虑的问题。在同一个 JVM 中运行的类可以有不同的来源，也可以有不同的可信度。因此，JVM 进程级别的强制保护是不够的。显然，是否允许一个打开文件的请求取决于请求打开文件的类。操作系统不具备这方面的信息。

因此，由 JVM 做这些保护决定。JVM 装载一个类时，它会给该类分配一个保护域，这个保护域给了类各种有关保护的许可。类被分配到哪一个保护域，这取决于类装载自哪个 URL 和类文件上的数字签名。一个可配置的策略文件决定了要分配给相应的域的许可。例如，装载来自一个可信赖的服务器的类可能被分配到允许这些类访问用户目录下的文件的保护域；而装载自不可信赖的服务器的类可能没有任何文件访问许可。

JVM 要决定由哪个类为一个访问受保护资源的请求负责，这不是一件简单的事。访问常常要间接地通过系统库或者其他的类来完成。例如，试想一下一个不允许打开网络链接的类。它可能会调用系统库来请求装载一个 URL 的内容。JVM 必须决定是否为这个请求打开一个网络链接。然而，应该根据哪个类来决定是否允许这个链接，是应用程序还是系统库？

Java 2 中采用的哲学是：要求库类显式允许装载被请求的 URL 的网络链接。更普遍的做法是：如果要访问一个受保护资源，调用序列中引发该请求的方法必须显式地断言访问该资源的特权。这样的话，这个方法就要为这个请求负责；由此推测，它会承担起所有必要的检查以确保该请求的安全。当然，并不是任何方法都可以断言一个特权；只有当方法所属的类运行在一个域本身可以执行这些特权的保护域上时，这个方法才可以这样做。

这个实现方法被称为**栈检查**(stack inspection)。JVM 中的每个线程都有一个相关联的栈，栈里存放的是该线程的正在进行中的方法调用。当方法的调用者可能不可靠时，它会在一个实施特权的块内执行一个访问请求来直接或间接地访问一个受保护的资源。当方法进入这个实施特权的块时，与该方法相对应的栈帧就会被做上相应的注释以表明该事实的发生。然后，这个块的内容就会被执行。如果紧接着又有一个来自该方法本身或者它的调用者的访问受保护资源的请求，那么系统会调用一个许可检测来检查栈，以决定是否许可这个请求。该检查在调用线程的栈上检查栈帧，从最近添加的帧开始，一直到最老的那个帧。如果首先发现一个栈帧有执行特权的注释，那么许可检测就立即返回，并允许访问。如果首先发现一个基于该方法所属类的保护域的栈帧是不允许访问该栈帧的，那么许可检测会抛出一个异常。如果栈检查在检查完该栈之后，没有发现以上两种类型的栈帧，那么根据具体实

现来决定是否允许访问。

栈检查如图 18.8 所示。图中，*在不可信任的 Java 小应用程序*的保护域内，一个类的 gui 方法执行了两个操作，首先是一个 get 操作，然后是一个 open 操作。前者是在*URL 装载器*保护域中的某个类的 get 方法的调用，它被允许打开与域 *lucent.com* 中的站点的会话，特别是一个代理服务器 *proxy.lucent.com*，该代理服务器用来重新获得 URL。因此，这个不可信任的应用小程序的 get 调用会成功：在网络库中，许可检测发现 get 方法的栈帧，而 get 方法在一个执行特权块完成它的打开操作。然而，因为许可检测在遇上 gui 方法的栈帧前不会找到执行特权的注释，所以一个不可信任的应用小程序的打开方法的调用会引发一个异常。

保护域	不可信任的java小应用程序	URL加载器	联网
套接字许可:	没有	*.lucent.com:80, 连接	任意

类:	gui
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged{
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission(a,connect);
			connect(a);
			...

图 18.8　栈检查

当然，由于要执行栈检查，所以必须禁止一个程序在它自己的栈帧上修改注释，或者执行其他的栈检查操作。这是 Java 和许多其他语言之间的最重要的差别之一。一个 Java 程序不能直接访问内存。确切地说，它只能操作一个它拥有引用的对象。引用是不可伪造的，而操作通过明确的接口完成。通过一个复杂的装载期和运行期检查集合，强制程序遵从了本段前文所述规定。最终，由于一个对象无法得到一个指向自身的栈或者保护系统的其他组件的引用，所以它无法操作自己的运行期栈。

更常见的是，Java 的装载期和运行期检查会对类强制要求类型安全。类型安全确保类不能将整数当做指针，不能在数组的范围之外访问数组，不能以一种直接的方式访问内存。其实，一个程序只能通过对象所属类中定义的方法来访问一个对象。这让类达到了很好的封装效果，并保护自身的数据和方法免受装载在同一个 JVM 上的其他类的干扰，这是 Java 保护的基础，例如，一个变量可以被定义为私有的，这样就只有该变量所属类本身可以访问它；或者声明为被保护的，这样就只有该变量所属类本身、继承自该类的子类和同一个包中的类可以访问该变量。类型安全确保这些约束会被强制执行。

18.8 小　结

计算机系统包含许多对象，系统必须保护这些对象，防止它们被误用。对象可以是硬件

(如内存、CPU时间或I/O设备),也可以是软件(如文件、程序、抽象数据类型)。一个访问权限是在一个对象上执行某项操作的许可。一个域是一个访问权限集合。进程在域中执行,并且可以使用域中的任意访问权限来访问、操作对象。在一个进程的生命期中,可以将它绑定到一个保护域,也可以允许它从一个保护域切换到另一个保护域。

访问矩阵是一个普通的保护模型。不需要给系统或用户强加特定的保护策略,访问矩阵就可以为保护提供一种机制。策略和机制的分离是设计上的一个重要发展。

访问矩阵是稀疏矩阵。通常,它要么被实现为一系列关联到每个对象的访问列表,要么被实现为一系列关联到每个域的权限列表。如果将域和访问矩阵本身看做对象,就可以在访问矩阵模型中实现动态保护。将访问权限从一个动态保护模型中删除,就这方面,将访问列表策略和权限列表策略做一个比较,通常用前者要比用后者更容易实现。

实际的系统有着更多的限制,并且通常只为文件提供保护。UNIX是一个代表,对每一个文件,它都分别为文件拥有者、组和普通公共用户提供了读、写和执行等权限保护。MULTICS中,除了文件访问外,还采用了一个环结构。Hydra、剑桥CAP系统和Mach是权限系统,这些权限系统已经将保护的范围扩展到用户定义的软件对象。

和操作系统相比,基于语言的保护为请求和特权提供了更细致的颗粒化仲裁。例如,一个单独的Java JVM可以同时运行若干个线程,每个线程都在一个不同的保护类中。它通过复杂的栈检查和语言的类型安全来强制执行资源请求。

习题十八

18.1 权限列表和访问列表之间的主要差别是什么?

18.2 一个Burroughs B7000/B6000 MCP文件被标记为敏感数据。当这样一个文件被删除时,系统会用一些随机位覆盖它所占用的存储区域。请问采取这种策略的目的何在。

18.3 在一个环保护系统中,0层拥有最大的对象访问权限,而n层(大于0)拥有较少的访问权限。在环结构中,一个程序在一个特定层的访问权限被看做一个权限集合。取任意一个对象,一个处于j层的域的权限和一个处于i层的域的权限之间的关系是什么(假设$j>i$)?

18.4 有这样一个计算机系统,学生只能在10 P. M. 至6 A. M. 的时段内玩"电脑游戏",教师只能在5 P. M. 至8 A. M. 的时段内玩游戏,而计算机中心的成员可以在任意时间玩游戏。设计一个能有效实现该策略的算法。

18.5 系统RC 4000(和其他系统)定义了一个进程树,这样一个进程的所有后代只能由它们的祖先提供资源(或对象)和访问权限。因此,如果一个祖先没有权限做某事,那么它的任何子孙都决不可能拥有做这件事的权限。这棵树的根是操作系统,它拥有做所有事的权限。假设将访问权限集合表示为一个访问矩阵A。$A(x,y)$定义了进程x对对象y的访问权限。如果x是z的一个后代,那么对于任意的对象y,$A(x,y)$和$A(z,y)$的关系是什么?

18.6 要在一个计算机系统中实现高效的权限操作需要哪些硬件特性的支持?能不能把这些特性应用到内存保护中?

18.7 考虑这样一个计算环境，系统中的每一个进程和每一个对象都有一个惟一的数字与之对应。假如只有在满足条件 $n>m$ 时，才允许编号为 n 的进程访问编号为 m 的对象。那么这个保护结构的类型是什么？

18.8 如果一个共享栈被用来传递参数，那么会引发什么样的保护问题？

18.9 试考虑这样一种计算环境，一个进程只能访问一个对象 n 次。请设计一个算法来实现这个策略。

18.10 如果访问某对象的所有权限都被删除了，那么就再也不能访问这个对象。同时，对象也应该被删除，而且它所占用的空间应该归还给系统。请为该策略设计一个有效的实现方案。

18.11 什么是需要则知道策略？为什么对于一个保护系统来说，遵循这个策略很重要？

18.12 为什么保护一个只允许用户执行自己的 I/O 操作的系统是很困难的？

18.13 权限列表通常被保存在用户自己的地址空间内。系统是怎样确保用户不能修改列表的内容的？

18.14 如果允许一个 Java 程序直接修改它自身的栈帧的注释，那么 Java 的保护模型会付出什么样的代价？

推荐读物

Lampson[1969]和 Lampson[1971]设计了域与对象间的保护的访问矩阵模型。Popek[1974]、Saltzer 和 Schroeder[1975]提供了关于保护问题的优秀的综述。Harrison 等[1976]使用了这个模型的一个正式版并让它们能够证明一个保护系统的数学属性。

权限的概念是从 Iliffe 和 Jodeit 的 *codewords* 演化而来的，它们是在美国 Rice 大学的计算机上实现的(Iliffe 和 Jodeit[1962])。术语*权限*是由 Dennis 和 Horn[1966]引入的。

Wulf 等[1981]描述了 Hydra 系统。Needham 和 Walker[1977]描述了 CAP 系统。Organick[1972]论述了 MULTICS 环保护系统。

Redell 和 Fabry[1974]、Cohen 和 Jefferson[1975]、Ekanadham 和 Bernstein[1979]论述了恢复问题。设计者 Hydra(Levin 等[1975])提倡规则与机制分离的原则。Lampson[1973]首先论述了制约问题，Lipner[1975]进一步论证了它。

Morris[1973]最先使用高级语言来表示访问控制，他提出 18.7 节中论述的 seal 和 unseal 操作的使用。Kieburtz 和 Silberschatz[1978]、Kieburtz 和 Silberschatz[1983]、McGraw 和 Andrews[1979]提出了用来处理一般动态资源管理策略的不同语言构造。Jones 和 Liskov[1978]设想怎样将一个静态访问控制方案整合到一个编程语言中让它支持抽象数据类型。

在 Wallach 等[1997]和 Gong 等[1997]中可以找到关于栈检查的更详细的分析，包括其他方法与 Java 安全性的比较。

第十九章 安 全

在第十八章讨论了保护。保护严格来讲是个内部问题：应该以一种怎样的方式，提供对存储在计算机系统中的程序和数据的受控访问？相对来说，**安全**除了需要一个适当的保护系统外，还需要考虑系统运行的外部环境。如果操作者的控制台被暴露在未经授权的人员面前或者可以轻易地将文件从计算机系统的磁带和磁盘上转移到一个未受保护的系统中，那么内部保护是无用的。这些安全问题不是操作系统问题，而是管理问题。

要保护系统内存储的信息（数据和代码）和计算机系统的物理资源，防止它们被未经授权者访问，被恶意地更改或破坏，及被意外地引入不一致问题。这一章要考察信息被误用或者被有意引入不一致问题的可能途径。然后，提出一些有效机制，以防止这些问题的出现。

19.1 安全问题

在第十八章中，讨论了那些操作系统能够提供的机制（借助一些硬件支持），它们使用户得以保护属于他们的资源（通常为程序和数据）。只有在用户遵守资源的使用和访问规则时，这些机制才能很好地发挥作用。只有用户可以在各种情况下按计划使用和访问资源时，才会认为系统是**安全**的。遗憾的是，无法做到百分之百的安全。然而，必须有这样一些机制，它们能够将出现安全缺口的情况控制在一个极小的概率范围内，要比一般标准小得多。

系统的安全违例（或误用）可以分为有意的和意外的。防止意外误用比防止恶意破坏要容易得多。恶意破坏的形式有以下几种：

- 未经授权的读数据操作（或者信息盗用）
- 未经授权的修改数据操作
- 未经授权的数据破坏
- 阻止系统的正当使用（或者拒绝服务）

要在系统中完全杜绝恶意滥用的现象是不可能的，不过可以采取措施，使得恶意破坏的代价异常地高，这样使未经授权者打消绝大多数（甚至全部）未经授权访问系统资源的企图。

要保护系统，必须在4个层次上采取安全机制：

1. **物理**：必须采取物理措施保护计算机系统的站点，防止入侵者强行地或者秘密地入侵。
2. **人**：必须谨慎筛选用户，减少授权用户授予入侵者访问权限的机会。
3. **网络**：现代系统中的许多计算机数据都在私人租用的线、共享的线（如因特网和电话

线)上传播。中途截取这些数据的危害和入侵计算机的危害是等同的。这些连接中断的起因可能是一个远程的拒绝服务攻击,结果会降低系统的使用率和可信度。

4. **操作系统**:操作系统必须防止自身遭受意外的或者有意的安全破坏。

如果要确保操作系统的安全,那么必须保证前两层的安全。如果一个高层次的安全(物理或人)中存在弱点的话,它会欺骗低层次的安全(操作系统)措施。

在许多应用中,确保计算机系统的安全是一件值得努力的事。存有工资表或者其他金融数据的大型商业系统很容易引起盗贼的兴趣。不道德的竞争者可能会对存储着合作数据的系统产生兴趣。不管是因为意外还是因为受骗,数据丢失都会严重影响企业的运作。

另一方面,系统硬件必须为安全措施的实现提供保护(第十八章)。例如,MS-DOS 和 Macintosh OS 之所以没有提供什么安全措施,就是因为当时它们的硬件设计没有提供内存或者 I/O 保护。现在硬件已经发展成熟,能够提供充分的保护;操作系统设计者开始争相添加安全措施。遗憾的是,向一个已经在使用的系统添加一种特性,要比在构造系统之前就设计并实现一种特性相对而言困难得多。后来的那些操作系统,如 Windows NT,都是从设计时就开始考虑提供安全特性。

在本章接下来的章节中,将在操作系统的层次上讲述安全。物理和人这两层的安全虽然重要,但它们远远超出了本文的范围。操作系统内和操作系统间的安全的实现途径有很多:从访问系统时用到的密码到系统内同时运行的进程间的隔离等。文件系统也提供了一定程度的保护。

19.2 用户验证

操作系统的一个主要的安全问题就是**验证**(authentication)。这种方式依赖于识别当前执行的程序和进程的能力。这种能力转而依赖于识别每个系统用户的能力。一个用户通常要识别自己。应该怎样确定一个用户的身份是否真实呢?通常,验证基于以下三个条款中的一条或者几条:用户持有的物品(一个钥匙或者卡),用户的信息(一个用户鉴别和密码),或用户的特征属性(指纹、视网膜模型或者签名)。

19.2.1 密码

验证用户身份的最常用的方法就是使用**密码**。当用户使用用户 ID 或者账户名识别自己时,系统会要求用户输入密码。当用户提供的密码和系统存储的密码匹配时,系统就认为该用户是合法用户。

缺乏更完善的保护策略时,计算机系统经常用密码来保护对象。密码可以被看做钥匙或者权限的特例。例如,一个密码可以被关联到每个资源(例如文件)。用户在产生使用资源的请求时,必须向系统提供密码。如果密码是正确的,系统就允许访问。不同的密码可能

会关联到不同的访问权限。例如，对文件执行读操作、附加操作和更新操作时可能会使用不同的密码。

19.2.2 密码脆弱的一面

密码易于理解和使用，因而应用得特别多。很遗憾，密码可能会被猜中，或被意外地泄露，或被监听，或者被一个授权用户非法传递给一个未授权用户，在以下章节中讨论这些问题。

有两种常见的猜测密码的方法。第一种是用户经常使用一些明显的信息（如他们宠物或者配偶的名字）作为密码，而且入侵者（人或程序）掌握了目标用户的有关信息。另一种方法是使用暴力：尝试枚举，或者可能的字母、数字和标点的组合，直到找到密码。只要重复试验，很容易破解出位数较少的密码。例如，一个由十进制数组成的长度为 4 位的密码只有 10 000种可能。平均 5 000 次命中一个密码。一个每千分之一秒可以试验一个密码的程序，只需要 5 秒就可以猜出一个 4 位的纯数字密码。如果一个系统提供长度较长的密码，允许在密码中区分字母的大小写，允许在密码中使用各种标点符号和数字，那么枚举方法通常会遭到失败。当然，前提是用户必须充分使用大密码空间，例如，不应该只在密码中使用小写字母。

前文提到了由于泄露而导致密码安全失败的问题，这可能是由可见的或者电子的监视引起的。在用户登录时，入侵者的视线可以越过用户的肩膀偷窥用户密码（*偷窥*），或者通过键盘上各个键的磨损程度猜测密码。另外还有一种途径：任何人，只要他拥有计算机所常驻的网络的访问权限，他就可以神不知鬼不觉地添加一个网络监视器，通过该监视器就可以监视网络中传输的数据（*嗅探*）、受保护用户的 ID 和密码。但是只要系统将包含密码的数据流加密，就可以轻松解决这个问题。

如果将密码记录在一个可能丢失或者被偷窥到的地方，泄露问题就会变得特别严重。正如将要看到的，一些系统强迫用户使用很难记忆的或者是很长的密码。如果用户实在无法在大脑里记住这些密码，那么他就得将密码记录下来，这比允许使用简单密码的系统更不安全。

最后一个削弱密码安全性的途径是非法传递，这是人性的弱点导致的恶果。绝大多数计算机安装都有一个强迫用户共享账号的规则。实现这个规则有时是为了方便账号管理，但这种做法常常不利于安全。例如，如果几个用户共享一个用户 ID，如果那个用户 ID 出现了安全问题，那么就无从知道此时是哪个用户在使用那个 ID，甚至无法确认该用户是否是授权用户。如果一个 ID 只归一个用户使用，关于该账号的使用状况，自然是直接询问账号拥有者了。有时候，用户为了帮助朋友或者避开账号管理，有意破坏账号共享规则，这种行为会导致系统被未授权者访问，而有些访问者是有破坏意图的。

密码可以由系统产生，也可以让用户选择。系统产生的密码通常不容易记住，因此用户可能会将密码记录下来。而用户选择的密码很容易被猜中（例如，用户名或喜爱的车）。在一些站点中，管理员会不时地检查用户密码，如果用户密码太短或者太简单，他就会提醒用

户重新选择密码。一些系统为密码设定了有效期，强制用户定期更新密码(例如，每三个月更换一次)。这个方法并不保险，因为用户可以只准备两个密码，每次都在这两个密码之间切换。为了解决这个问题，一些系统会记录用户使用过的密码。例如，系统会记录每个用户最近使用的 n 个密码。

系统还可以采用简单密码机制的变种。例如，可以频繁地更换密码。在极端的情况下，每次会话都更改密码。每次会话的结束时都要选择(或者由系统选择或者由用户)一个新的密码，新密码将在下次会话时使用。这样一来，即便密码被误用了，也只会被误用一次。当合法用户在下次会话中使用一个当时合法的密码时，他会发现有人曾经入侵过安全系统。当场就可以采取措施修复被破坏的安全。

19.2.3 密码加密

以上种种的密码安全方法，都有一个难以解决的问题，就是怎样在计算机中秘密地存储密码。用户输入密码时要用系统存储的密码来验证用户密码，又要秘密地存放密码，系统怎样才能做到两者兼顾呢？UNIX 系统会对密码进行**加密**(encryption)，这样就不再需要秘密地保存密码列表。每个用户都有一个密码。系统包含一个极其复杂的函数，设计者希望该函数不可逆，而计算函数值却非常简单。也就是说，给定一个值 x，很容易计算出函数值 $f(x)$。给定一个函数值 $f(x)$，却不可能计算出 x 的值。用这个函数加密所有的密码。系统只保存已经加密的密码。即便入侵者看到了存储的已加密密码，因为他不可能从已经加密的密码计算出加密前的密码，因此他不可能得到密码。因此，没有必要秘密存放密码的文件。函数 $f(x)$通常是一个经过精密设计和测试的**加密算法**，将在 19.7.2 小节中讨论它。

这个方法的缺陷是系统不再全权控制密码。虽然密码已经加密了，但是任何有密码文件备份的人，都可以快速运行密码加密时采用的加密规则，例如给一本字典里的所有词加密，并且将加密结果和密码文件中的密码做比较。如果用户选择的密码恰好是该字典里的一个词，那么这个密码就被破译了。在足够快的计算机上，或者是慢速计算机的集群上，这样的比较只需要几个小时就够了。因为 UNIX 采用的是一个著名的加密算法，因此黑客可能会有一些以前已经破译出来的"用户密码－已加密密码"对，以便快速找到密码。因此，新版本的 UNIX 将加密过的密码保存在一个只有**超级用户**才能读取的文件中。将用户提供的密码和系统存储的密码做比较的程序首先要执行设置用户 ID 的操作，将用户 ID 设置为 root，root 用户可以读取这个文件，而别的用户不行。

UNIX 采用的密码方法的另外一个缺陷是：许多 UNIX 系统只注重密码的前 8 个字符，因此用户要充分利用可用的密码空间。为了避免被字典破解法猜中，一些系统不允许用户使用字典中的词做密码。这里有一个产生安全密码的好方法：选取一个容易记忆的短语，采用它的每个词的首字母，区分大小写字母，并夹带数字或者标点。例如，短语"My mother's name is Katherine."可以产生密码"MmnisK.!"。这个密码很难破译，却很好记。

19.2.4 一次性密码

为了避免密码被嗅探或偷窥，系统可以使用**配对密码**集合。一个会话开始时，系统随机选择并提供一个密码对的一部分，用户必须提供另外一个部分。在这种系统中，由系统挑战用户，并且要求用户提供正确的答案来响应挑战。

可以将这种方法扩展为使用一个算法作为密码。例如，算法可以是一个整数函数。系统选择一个随机的整数并向用户展示该整数。用户用这个整数函数计算系统提供的整数，并将正确结果回复给系统。系统自身也用函数计算该整数。如果两者的结果匹配，就允许访问。

用算法做密码的方法是不怕泄露的；也就是说，用户输入一个密码，任何一个截取该密码的实体都无法重用该密码。在这个方法变形中，系统和用户共享一个秘密。这个秘密从不在可能会发生泄露的媒体中传播。实际上，这个秘密和一个共享的种子一起，被用来做一个函数的输入。一个**种子**是一个随机数或者是一个由字母和数字组成的序列。这个种子来自计算机的验证挑战。这个秘密和这个种子都被用做函数 f(*秘密*,*种子*)的输入。这个函数的结果被当做密码传送给计算机。因为计算机同样知道这个秘密和这个种子，它可以执行同样的计算。如果结果匹配，那么用户通过验证。下一次需要验证用户时，系统会产生另外一个种子，接下来的步骤是一样的。但这一次的秘密肯定跟以前的不一样。

在这个**一次性密码**(one-time password)系统中，每个实例的密码都不一样。任何人，如果他在一个会话中抓获了一个密码，并且在另外一个会话中使用该密码，他必定会失败。一次性密码可以避免由密码泄露而引起的错误验证。一次性密码系统在商业中的应用，如SecureID，采用的是硬件计算器。绝大多数都采用信用卡的样子，但是会带键盘和显示屏。有些用当前时间做随机种子。用户通过键盘输入共享的秘密，也被称为**个人身份号码**(personal identification number，PIN)。显示屏显示一次性密码。如果同时使用一个一次性密码和一个PIN，那么就是一种**双因素验证**(two-factor authentication)。这种情况下需要两种不同类型的组件。双因素验证提供的验证保护比单因素验证提供的要好得多。

另外一种一次性密码的变形是使用一本**密码书**，或者是说**一次性缓冲器**，它是一个单独使用的密码的列表。这种方法中，按顺序采用列表中的密码，每个密码只使用一次，用过的密码被立即删除或者标记为无效密码。通常使用的一次性口令系统的一次性密码的来源要么是一个软件计算器，要么是一本基于这些计算的密码书。

19.2.5 生物测定学

用做验证的密码有许多变种。Palm或者手掌阅读器通常会保护物理访问，如对一个数据中心的访问。这些阅读器从它们的手掌阅读器的缓冲器中读取信息，并将这些信息和存储的参数匹配。这些参数可能包含一个温度图，还有手指长度、手指宽度和指纹模型。这些

设备由于占用的空间太大同时代价也太高，而不适合在普通的计算机验证中使用。

指纹阅读器的精度已经足够高了，价位也可以接受了，将来它的使用会更加普遍。这些设备读取使用者的指纹模型，并将它们转换成一个数字序列。随着时间的推移，它们会存储一个数字序列集合，调整手指在阅读垫的位置和其他的一些因素。然后软件就能扫描垫子上的一个手指，并将它和存储的序列比较，根据比较结果决定垫子上的手指是否和存储器中的相同。当然，多用户系统可以存储用户描述文件，扫描器可以区分这些用户。同时要求一个密码和一个用户名、指纹扫描的系统会提供一个精度很高的双因素验证策略。如果在传输过程中加密这些信息，系统将能够抵御欺骗和二次攻击。

19.3 程序威胁

如果一个用户使用另一个用户编写的程序，那么可能会发生误用或者其他一些不可预期的行为。在 19.3.1、19.3.2 和 19.3.3 三个小节中，将会描述可能产生这些行为的一般方法：特洛伊木马、后门、栈和缓冲溢出。

19.3.1 特洛伊木马

许多系统都提供了这样一种机制：允许程序作者之外的用户运行程序。如果这些程序运行时所在的域提供了用户的访问权限，那么其他的用户就可能误用这些权限。例如，一个文本编辑器，它可能允许根据关键字搜索要编辑的文件。如果有文件被搜索到，就会将整个文件的内容拷贝到一个文本编辑器的创建者可以访问到的特定区域中。一个误用自身环境的代码段被称为**特洛伊木马**(Trojan horse)。长的搜索路径，如 UNIX 系统中常用的那些，常常会加剧特洛伊木马问题。给定一个模糊的程序名时，搜索路径通常会列出要搜索的目录的集合。在该路径中搜索目标文件，然后执行该文件。搜索路径中的所有目录都必须是安全的，否则就可能会有特洛伊木马进入搜索路径，并且被附带执行。

例如，在搜索路径中使用字符“.”。字符“.”告诉 shell 在搜索中包含当前目录。因此，如果一个用户在他的搜索路径中包含字符“.”，而且已经将他的当前目录设置到一个朋友的目录下，并且输入了一个普通的系统命令，结果这个命令就有可能在那个朋友的目录下执行。程序将在朋友的域中运行，程序可以做朋友能够做的任何事情，包括删除朋友的文件。

特洛伊木马问题有一个变种，就是模仿一个登录程序。一个可靠用户开始在一个终端登录系统，稍后被告知输错了密码。他重新试了一遍正确的密码，然后登录成功。这期间发生了什么事呢？入侵者运行在这个终端的登录模拟器偷走了这个用户的验证钥匙和密码。这个登录模拟器保存好密码，发出一个登录错误的消息，然后退出；然后用户收到一个真正的登录提示信息。如果操作系统在一个互动会话的终点给出一个用法消息，或者是采用一个不可捕捉的键系列，如 Windows NT 中使用的 Ctrl+Alt+Del，都可以击败模拟登录程序

的攻击。

19.3.2 后门

程序和系统的设计者都可能在软件中留一个只有他自己能使用的漏洞。电影"*战争游戏*"讲述的就是这种安全攻击(或**后门**,trap door)。例如,代码可能会检查特定的用户 ID 或者密码,让这个特定的用户避开正常的安全程序。故事中的程序员因为盗用了银行的大笔存款被捕了;他们采取的手段是:在代码中隐藏漏洞,不时地将一笔交易额的 0.5% 划到他们自己的账户上。想想一个大的银行的交易量,就知道这个账户的款额很可能累积成一个天文数字。

可以在一个编译器中包含一个巧妙的后门。不管被编译的源代码怎样,这个编译器都会在产生标准的目标代码的同时,另外产生一个后门。这种行为所带来的后果是极其可怕的,因为检查程序的源代码发现不了任何问题。只有编译器的源代码中才有问题的相关信息。后门是一个很棘手的问题;经常为了查出一个后门,需要分析一个系统的所有组件的源代码。大家都知道,软件系统可能由上百万行代码组成,不可能经常分析整个系统。

19.3.3 栈和缓冲区溢出

一个来自系统外的攻击者,通过网络或者电话线连接,为了要访问目标系统,他最可能采用的攻击方式就是利用栈或缓冲区溢出。系统的授权用户也可能使用这种攻击方式谋求**特权升级**(privilege escalation),即获得系统许可的特权范围之外的特权。

从根本上说,这种攻击就是利用程序中的一个错误。这个错误可能是一个拙劣的程序中的一个很简单的例子:程序员没有对输入域进行边界检查。在这种情况下,攻击者向程序发送比预期的数目多得多的数据。通过试验和错误,或者通过检查被攻击程序的源代码,如果有的话,攻击者将会发现程序的弱点,并自己编写一个程序来完成以下几项任务:

1. 使一个输入域、命令行的参数或者一个输入缓冲溢出,如在一个网络监护程序中,直到写入栈中。

2. 用步骤 3 中装载的攻击代码的地址覆写栈内当前的返回地址。

3. 为紧接的栈空间写一段代码,这段代码包括了攻击者想要执行的命令,例如,产生一个 shell。

这个攻击程序的执行结果是一个根 shell 或者其他特权命令的执行。

例如,如果一个网页表格要求用户向其中一栏输入一个用户名,攻击者就可以发送一个用户名,并添加额外的字符,目的是使缓冲区溢出并且写栈,往栈中添加一个新的返回地址,还有攻击者想要执行的代码。当读缓冲的子程序从读缓冲的操作中返回时,返回地址是攻击代码的地址,紧接着就开始执行攻击代码。

利用缓冲区溢出的攻击特别致命,因为它可以只攻击一个系统,也可以在允许连接的信

道间流窜。这样的攻击可能发生在用做连接机器的协议间，因此很难检测和防止。他们甚至能绕过防火墙的安全措施。

这个问题有一个解决办法：给 CPU 添加一个特性，让它禁止执行存放在栈内的代码。最新版本的 Sun 的 SPARC 芯片有这项设置，最近版本的 Solaris 激活了这项设置。还可以修改处理溢出的子程序的返回地址，但如果返回地址和将要执行的代码都在栈内，就会产生一个异常，程序由于发生错误而中止。

19.4 系统威胁

绝大多数操作系统都为进程提供了产生其他进程的方法。在这样一个环境下，很容易产生操作系统资源和用户文件被误用的情形。导致这种误用情形的两种最常见的方法就是蠕虫和病毒。

19.4.1 蠕虫

蠕虫(worm)是一个利用**繁殖**(spawn)机制破坏系统性能的进程。蠕虫大量产生自身的拷贝，耗尽系统资源，甚至可能让其他进程都停止使用系统。在计算机网络中，蠕虫的影响特别大，因为他们会在系统间复制自己，藉此使整个系统瘫痪。曾经有过这样的事情，于 1988 年发生在因特网的 UNIX 系统中，导致了上百万美元的损失。

在 1988 年 11 月 2 日左右，美国康奈尔大学的一年级研究生 Robert Tappan Morris, Jr. 在若干台连到因特网上的主机上释放了一个蠕虫程序。这次攻击的目标是运行版本 4 BSD UNIX系统系列的 Sun Microsystem 的 Sun 3 工作站和 VAX 计算机，蠕虫以极快的速度向远处传播；在释放后的短短几个小时内，蠕虫就已经将系统资源消耗到足以让感染的机器崩溃的程度。

尽管 Robert Morris 设计这种自我复制的程序的目的是重复拷贝和扩散，UNIX 网络环境的一些特性也为蠕虫在系统中繁殖提供了便利。Morris 最初感染的那台主机很可能是开放的，外边的用户可以访问它。在那台主机上，蠕虫程序发现了 UNIX 操作系统安全程序的缺陷，并且利用 UNIX 在简化本地网络资源共享方面的工具，获得了数以千计的连接站点的未授权访问。接下来将简要讨论 Morris 采用的攻击方法。

这个蠕虫程序由两部分组成：一个**挂钩**(grappling hook)(也被称为**引导**(bootstrap)或**向量**(vector))程序和一个主程序。名为 ll.c 的挂钩程序由 99 行已编译的 c 代码组成，并且在每台已访问到的机器上运行。新感染系统上的挂钩的创建工作一旦完成，挂钩就开始连接到它原来的那台机器上，并从那台机器上拷贝一份主程序到*新感染的系统*中(图 19.1)。主程序继续搜索新感染的系统能够轻易连接上的那些机器。在这些活动中，Morris 发掘了 UNIX 网络的功能，rsh，用于执行简单的远程任务。通过建立特殊文件，文件中列出了“主

机一登录名"对,列表中的用户不再需要在每次访问远程账户时都输入密码。蠕虫在这些特殊文件中搜索那些不需要密码也允许远程登录的站点。在远程 shell 创建的地方上传蠕虫程序并开始新一轮的攻击。

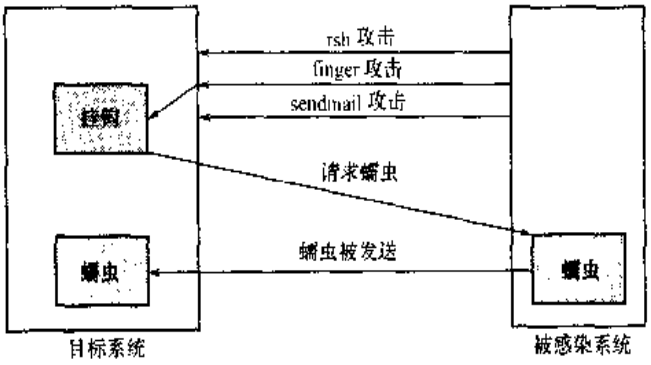


图 19.1 Morris 的因特网蠕虫

基于蠕虫的感染方法有三种,通过远程访问攻击是其中的一种。另外两种感染方法利用了 UNIX *finger* 和 *sendmail* 程序中的操作系统错误。*finger* 像电话目录一样工作;命令

finger 用户名@主机名

返回一个人的真实姓名和登录名,还有用户提供的其他一些信息,比如办公室和家庭地址、电话号码、调查计划或者座右铭。*finger* 在 BSD 站点上是作为一个后台程序运行的,并响应来自因特网的询问。恶意入侵利用的漏洞包括不检查溢出边界就读取输入数据。代码会执行一个**缓冲溢出攻击**。Morris 的程序用一个精心设计的长度为 536 B 的字符串询问 *finger*,超出分配给输入的缓冲,并覆写栈帧。*finger* 不是返回到它在 Morris 调用之前的正常的主程序中,这个监护进程 *finger* 被定向到一个存储在栈内的入侵的 536 B 长的字符串中。要执行的新进程 */bin/sh*,如果成功的话,会给蠕虫一个被攻击机器上的远程 shell。

Sendmail 中发掘的 bug 也包含可以被恶意入侵利用的监护进程。*Sendmail* 在一个网络环境中传送电子邮件。这个工具的调试代码允许测试者识别并显示出邮件系统的状态。对系统管理员而言,这个调试选项是很有用的,它通常是一个后台进程。Morris 在他的攻击武器库中包含了一个对调试的调用,这个调用没有像正常的测试那样指定用户地址,而是发出了一个用来邮寄和执行一个挂钩程序的拷贝的命令集。

一旦被安置好,蠕虫的主程序就开始系统地探测用户密码。开始时它只是试探那些不需要密码或者密码由账号一用户一用户名组合而成的情况,然后将结果和一个有着 432 个惯用密码的内部字典做比较,然后执行最后一个步骤:将标准 UNIX 在线字典中的每个词都当做可能密码测试一遍。这个精巧高效的密码破解算法三步曲使得蠕虫可以进一步访问被感染系统中的其他账户。然后蠕虫开始在这些最近破解的账号中搜索 *rsh* 数据文件。蠕虫会尝试每个 *rsh* 入口,然后获得该账号在远程系统上的访问权限。

每一次新的攻击,蠕虫程序都会搜寻已经激活的自我拷贝。如果它找到一个,那么新的

拷贝就退出，但如果新的拷贝是第 7 个实例，那么新的拷贝也会保留在目标机器上。每次重复瞄准一台目标机器时，蠕虫都会选择退出，因此要将它们检测出来确实很不容易。允许第 7 份拷贝继续感染(可能以此来挫败通过用假的蠕虫来引诱以阻止扩散的努力)，正是蠕虫程序的这个策略制造了因特网上的 Sun 和 VAX 系统的大规模感染。

UNIX 网络环境中有助于蠕虫繁殖的那个特性同样也会阻碍蠕虫感染进程。电子通信的便利，将源文件和二进制文件拷贝到远程机器的机制，及同时访问源代码和专家之间的协作，这些都有利于迅速找到解决方案。在第二天的傍晚之前，11 月 3 日，中止这个入侵程序的方法就已经通过因特网送达到各个系统管理员。几天之内，为那个安全漏洞定制的软件包就被开发出来了。

Morris 释放蠕虫的动机是什么？他的行为已经被定性为既是一次无害的恶作剧又是一次严重的犯罪攻击。从启动攻击的复杂性判断，蠕虫的释放和它选择的传播范围肯定是有意的。蠕虫精心设计了一些步骤，来掩盖自身踪迹和击退阻碍它传播的力量。然而，程序没有包含破坏和摧毁受感染系统的代码。作者显然具备编写这些代码的技能；实际上，在引导程序中已经包含了用来传播特洛伊木马或者病毒程序的数据结构(19.4.2 小节)。程序的行为引起了人们的关注，但无从推测作者的动机。出乎意料的是它所导致的法律后果：联邦法庭将 Morris 处以 3 年监禁，400 小时的公益劳务，和 10 000 的罚款。而 Morris 的诉讼费可能超过了 $100 000。

19.4.2 病毒

计算机攻击的另外一种形式是**病毒**。蠕虫和病毒的目的都是扩散到别的程序中并在系统中搞破坏，如修改和毁坏文件、导致系统崩溃和程序出错。但是，蠕虫是一个独立而完整的程序，而病毒是一个内嵌到合法程序中的代码段。对于计算机用户，特别是微机系统的用户来说，病毒是个大问题。一般地说，多用户计算机中，操作系统不能写可执行程序，因此病毒对多用户计算机的危害要小一些。即便病毒感染了一个程序，因为系统的其他方面有很好的保护措施，病毒的影响仍然是很有限的。单用户的系统没有这些保护措施，病毒就可以自由发挥。

病毒的传播通常有两种途径：一是用户从公共计算机中下载带病毒的程序；二是交换已感染的磁盘。例如，1992 年 2 月，美国康奈尔大学的学生设计了三个内嵌病毒的 Macintosh 游戏程序，并通过因特网将这些游戏发布到全世界的软件包中。威尔士的一个数学教授在下载游戏时发现了这些病毒，他的计算机系统的反病毒程序也向他发出了病毒报警。当时有大约 200 人已经下载了这些游戏。虽然这些病毒不会摧毁数据，但它会扩散到应用文件中，耽搁程序执行和引发程序错误。这些游戏是用一个康奈尔的电子邮件账号邮寄的，因此很快就追踪到了作者。纽约官方以计算机干扰的罪名逮捕了这两个学生。

这里还有一个小故事，美国加利福尼亚州一个被妻子抛弃的程序员给了他的前妻一个

磁盘，这个磁盘是用来引导出故障的计算机的。这个磁盘中存有一个病毒，该病毒会删除系统中的所有文件。这个丈夫被官方逮捕，并被没收了所有财产。

最近几年，病毒经常通过因特网（Office 文件的交换来传播，例如微软公司的 Word 文档。这些文档可能包含所谓的宏（或 Visual Basic 程序），Office 套件（Word、PowerPoint、Excel）会自动地执行这些宏。因为这些程序在用户自己的账号下运行，这些宏可以随心所欲地运行（如任意地删除用户文件）。

有时，一些引人注目的传媒事件会预告即将到来的病毒感染。例如，米开朗奇罗病毒，会在这位文艺复兴时期的艺术家的第 517 个生日那天，也就是 1992 年 3 月 6 日，删除已感染硬盘上的所有文件。由于很多病毒信息已经公开，绝大多数站点都会在病毒发作前发现并摧毁病毒，因此病毒的危害极小甚至没有危害。这些情况既预报了病毒问题，同时也警醒了一般公众。反病毒软件的销路极好。但绝大多数商业软件包都只对特定的、已知的病毒有效。它们在系统的所有软件中搜索构成病毒的特定指令模式。当它们找到一个已知模式时，就将这些指令删除，使程序免受感染。这些商业软件包在目录中列出了数以百计的目标病毒。病毒和反病毒软件都越来越复杂。针对反病毒软件的基本模式匹配方针，一些病毒会在感染其他软件时改变自身形态。反过来，反病毒软件也开始抛弃原有的基本模式匹配方针，转而采用家族模式匹配方针确认病毒。

要使计算机免受病毒感染，最好的方法是预防，或者是实行**安全计算**。购买未拆封的软件、少使用免费软件和盗版软件、少交换磁盘是预防感染的最佳途径。然而，即便是合法软件的新拷贝也不是百分之百安全的。曾经有过这样的例子，一个软件公司的雇员因为对公司不满，就让软件程序的母盘感染上病毒，结果给软件供应商造成了经济损失。要对付宏病毒的话，可以在交换 Word 文档时采用另一种被称为**富文本格式**（rich text format，RTF）的文件格式。与原来的 Word 格式相比，RTF 没有附加宏的能力。

另外还有一个防御方法：不要打开来自未知用户的邮件的附件。很遗憾，历史表明：在人们发现一个弱点的同一时刻，已经有人在利用这个弱点发动攻击了。例如，2000 年的爱虫病毒就是假装成一个来自朋友的情书，在世界范围内广为传播。一旦打开 Visual Basic 脚本写的附件，病毒就开始将自己发送给用户地址簿中的前几个用户。幸运的是，除了阻塞邮件系统、占用用户邮箱的空间，这种病毒没有什么大的危害。前文曾提到这样一条建议：不要打开来自未知用户的邮件附件。但爱虫病毒的行为否定了这条建议。这里有一条更中肯的建议：不要打开任何包含可执行代码的附件。一些公司强制执行该方针：公司的邮件系统服务器会删除所有进入公司内部网的附件。

另外有一项安全防护措施，虽然不能防止感染，但能在早期就发现病毒。用户必须在开始时完全格式化硬盘，特别是启动扇区，它经常是病毒的攻击目标。只上载安全的软件，并为每个文件计算校验和。未经授权的访问不能访问校验和列表。系统每次重启时，都有一个程序重新计算校验和，并将新的计算结果和原来的校验和列表做比较；如果发现不一致，

系统就会给出一个可能感染病毒的警告。

因为病毒和蠕虫经常在系统间活动，而不是程序间、进程间或者用户间，它们引发的经常是安全问题，而不是保护问题。

19.4.3 拒绝服务

最后一种攻击方式，**拒绝服务**(denial of service)，它不会从系统中窃取信息或者盗用资源，但它会阻止系统或者设备的合法使用。例如，一个入侵者可能会删除系统中的所有文件。绝大多数拒绝服务攻击的都是攻击者以前没有进入过的系统。实际上，对一个系统发动拒绝服务攻击通常要比入侵一台机器要容易得多。

这些攻击大都是面向网络的。它们分为两大类。第一种攻击，会占用许多系统设备资源，却不做任何有用的工作。例如，在用户点击网页时，一个 Java 小程序被下载到本机上，紧接着开始执行，并占用所有可用的 CPU 资源。

第二种情况是破坏网络设备。已经有过好几例拒绝服务攻击成功地攻击大网站的案例。它们是滥用 TCP/IP 的一些基本功能的结果。例如，如果攻击者发送标准协议的开始部分，说"我想要开始一个 TCP 连接"，但接着不再发送标准协议的剩余部分"现在这个连接结束了"，可能会有几个部分开始的 TCP 会话，它们将耗尽系统所有的网络资源，阻止了后来的合法 TCP 连接。这些攻击可能会持续几个小时或者几天，部分地或者完全阻止合法用户使用目标设备。目前只能在网络层截断这些攻击，除非操作系统升级，增加操作系统的强壮性。

一般地说，很难预防拒绝服务攻击。这些攻击和常规操作采用同样的机制。很难断定是正常使用突然增多还是攻击导致了系统减速。

19.5 保证系统与设备的安全

保证系统与设备的安全和入侵检测密切相关(19.6 节)。要确保系统安全、要检测出已经发生的安全攻击，需要两种技术协同工作。

定期扫描系统、发现系统的安全漏洞，是提高系统安全性的好办法。这些扫描可以在系统相对空闲时进行，这样它们的影响就会比日志的影响小一些。这样的扫描可以检查系统的以下几个方面：

- 短的或者容易猜中的密码
- 未授权的特权程序，如设置用户 ID"setuid"的程序
- 系统目录中的未授权程序
- 意外的长时间运行的进程
- 用户和系统目录下不恰当的目录保护

- 对系统数据文件(如密码文件、设备驱动甚至操作系统核心本身)的不恰当保护
- 程序搜索路径中的不安全入口(如 19.3.1 小节中讨论的特洛伊木马)
- 通过校验和的值发现的系统程序的改变
- 意料之外的或者隐蔽的网络监护程序

安全扫描发现的问题可以自动修复,也可以报告给系统管理员。

联网的和不联网的计算机相比,前者更容易受到安全攻击。除了来自已知访问节点集合的攻击,比如直接连接的终端,还要面对来自一个庞大而未知的访问节点集合的攻击,这是一个极其严重的潜在的安全问题。即便是通过调制解调器连接到电话线的系统也要比不联网的计算机面临更多的危险。

实际上,美国官方认为,系统的安全性和系统最远能到达的连接的安全性是等同的。例如,一个顶级机密的系统,只有同楼的同样顶级机密的系统能访问到它。只要这个原本封闭的环境和外界有联系,而不管是什么形式的联系,这个系统就丧失了它的顶级机密等级。一些政府设备采用了最高级别的安全警戒。当终端处于空闲状态时,就将连接终端和这个安全计算机的插头锁在办公室的保险箱里。用户如果要访问计算机,他必须知道一个物理锁的组合,还有这台计算机本身的验证信息。

遗憾的是,对于系统管理员和专职计算机安全维护员来说,要将一台计算机锁在一个房间里并拒绝所有的远程访问,这几乎是一件不可能的事。例如,因特网连接着数以百万计的计算机。对许多公司和个人来说,因特网责任重大,是一个不可或缺的资源宝藏。如果将因特网看做一个*俱乐部*,那么就像任意一个有着上百万成员的俱乐部一样,它里面会有许多好成员,同时也会有一些坏成员。坏成员有很多可以在因特网上使用工具,他们试图访问一些互联的计算机,就像 Morris 借助蠕虫采取的行动一样。

怎样将可靠的计算机安全地连接到一个不可靠的网络呢?其中的一条解决途径是使用防火墙来分离可靠和不可靠的系统。**防火墙**(firewall)是一台夹在可靠系统和不可靠系统之间的计算机或者路由器。它限制这两个**安全域**之间的网络访问,并且监控和记录所有的连接。它还会根据源地址或者目的地址、源端口或者目的端口或者连接的方向来限制连接。例如,**网页服务**器通过 http(超文本传输协议)和网页浏览器连接。因此,防火墙可以这样控制:只允许所有防火墙外部的主机到防火墙内部的网页服务器的 http 连接可以通过防火墙。Morris 的因特网蠕虫入侵计算机时使用的是 finger 协议,因此,finger 协议无法通过这道防火墙。

实际上,一道防火墙可以将一个网络分离成几个域。一个通用的实现方法是:将因特网作为一个不可靠的域;将一个被称为**非军事区**(demilitarized zone, DMZ)的半可靠和半安全网络作为另外一个域;将一个由公司的计算机组成的局域网作为一个第三域(图 19.2)。允许这两类连接:从因特网到 DMZ 计算机的连接,从公司计算机到因特网的连接;禁止以下两类连接:从因特网或者 DMZ 到公司计算机的连接。以下为可选项:可能允许 DMZ 和一台

或多台公司计算机之间的受控连接。例如，一个位于 DMZ 的网页服务器可能需要查询一个位于公司网络的数据库服务器。使用防火墙的话，就会对所有的访问进行管理，并且任意一台基于允许通过防火墙的协议的被入侵 DMZ 系统，仍然无法访问公司网络中的计算机。

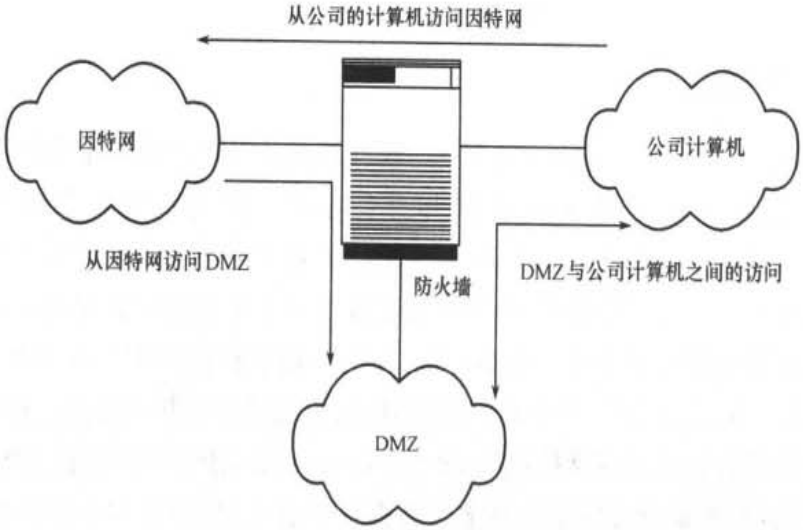


图 19.2 通过防火墙实现域分离的网络安全

当然，防火墙本身必须安全可靠，必须能够抵御攻击，否则它保护连接的能力就会被削弱。防火墙无法防止**隧道攻击**(tunnel)或者在防火墙所允许的协议或连接内传播的攻击。防火墙允许 http 连接，它不能防止包含在 http 连接内容中的攻击，因此防火墙无法阻止攻击者利用缓冲区溢出攻击网页服务器。同理，拒绝服务攻击可以像攻击一台普通机器那样攻击一道防火墙。防火墙还有一个弱点，即**欺骗**(spoofing)，一台未授权主机满足一定的授权标准时，就可以伪装成一台授权主机。例如，如果有这样一条防火墙规则：根据主机的 IP 决定是否允许来自该主机的连接，那么一台主机只要成功抢占授权主机的 IP，就可以通过防火墙发送包。

19.6 入侵检测

入侵检测(intrusion detection)，从这个名称来看，它主要是检测对计算机系统的入侵企图或已经成功的入侵，并且启动对入侵的恰当的响应。入侵检测会碰到的一系列技术标准问题。以下内容是部分技术标准：

- 检测的时机：实时的(当入侵发生时)或者仅在事件发生后。
- 检测入侵活动时需要检测的输入类型：可以包括用户 shell 命令，进程的系统调用，还有网络包的头部分或者内容。有时候根据几个相同源头的相关信息就可以检测出某种入侵。
- 响应能力的范围。简单的响应只需要向管理员报告潜在的入侵；或者中止潜在的入侵活动，例如杀死明显参与入侵活动的进程。如果是复杂的响应，系统可能会透明地将入侵

活动转向一个**陷阱**(trap),即向入侵者暴露一个错误资源,目的是监控攻击并且获得攻击的相关信息;在攻击者眼中,这个资源是真实的。

入侵检测的设计空间有着极大的自由度,现在已经有许多种入侵检测的解决方案了,统称为**入侵检测系统**(intrusion-detection system, IDS)。

19.6.1 入侵的组成

实际上,很难定义一个合适的入侵规范。现在流行的是两种并不十分严谨的方法,自动的IDS一般都会满足于在这两者间做出一个选择。第一种被称为**基于签名的检测**(signature-based detection),它在系统输入和网络传输中检测预示着攻击的特定行为模式(或**签名**)。这里举一个简单的例子,让系统监视登录一个账号屡次失败的行为,因为这是一个有人在猜测该账号密码的征兆。再举一例,在网络包中扫描以UNIX系统为目标的字符串*/etc/passwd/*。第三例,即一个病毒检测软件,它扫描的是已知病毒的二进制码。

第二种方法通常被称为**异常检测**(anomaly detection),涉及在计算机系统中检测异常行为的技术。当然,并不是所有的系统异常行为都预示着入侵,但通常会做这样一个假设:入侵通常会在系统中引入异常行为。异常检测的一个例子,即监视一个监护进程的系统调用,来检测它的系统调用行为是否偏离了正常模式,这可能预示着在监护进程中已经发生了一个破坏它行为的缓冲区溢出。另外一例是通过监视shell命令来检测一个给定用户使用的异常命令,或者检测一个用户的异常登录时间,这些可能预示着攻击者已经得到了该账号的访问权限。

基于签名的检测和异常检测可以看做同一个硬币的两个面:基于签名的检测试图描述危险行为并在这些危险行为发生时能够检测出来;而异常检测试图描述正常行为并且在这些行为之外的行为发生时可以检测出来。

这些不同的方法让IDS有了很多大不相同的属性。特别是,异常检测可以检测出以前不知道的入侵方法,而基于签名的检测只能辨别模式已知的攻击。如果某种攻击方式在签名产生的时候并没有被预测到,那么它可以从基于签名的检测中逃脱。病毒检测软件商都知道这个问题,因此,在一个新的病毒产生并被手动检测到之后,他们必须频繁地更新签名。

并不是说异常检测一定会超越基于签名的检测。实际上,采用异常检测的系统面临着一个重大的挑战:精确设定系统正常行为的基准点。如果在测定系统基准点时系统已经被入侵过,那么这个正常行为的基准点就可能包含入侵行为。即便基准点的测定是在系统干净时进行的,没有受到入侵行为的影响,仍然很难给出一个完美的基准点,因为它必须是正常行为的一个完整描述。否则,错误警告的数目将会多得让人难以忍受。

为了说明错误警告发生频率过高的影响,请看由几十台UNIX工作站组成的安装过程,为了进行入侵检测,这里将记录安全相关的所有事件。这样一个小小的安装过程每天都可能轻易地产生上百万条审计记录。然而,只有一两条值得管理员去做一番调查研究。做一

个乐观的假设：每十条审计记录中反映一次这样的攻击，就可以用下面这个公式粗略地计算出审计记录真实反映的入侵行为的发生比率：

$$\frac{2\ \frac{\text{入侵行为}}{\text{天}} \times 10\ \frac{\text{记录}}{\text{入侵}}}{10^6\ \frac{\text{记录}}{\text{天}}} = 0.000\ 02$$

把这个解释为"入侵记录发生的可能性"，可以用 $P(I)$ 表示这个公式；也就是说，事件 I 反映真实入侵行为的记录的发生次数。因为 $P(I)=0.000\ 02$，还知道 $P(\neg I)=1-P(I)=0.999\ 98$。现在，让 A 表示这个引发警报的 *IDS* 事件。一个精确的 *IDS* 应该会让 $P(I|A)$ 和 $P(\neg I|\neg A)$ 都极大化，也就是说，警报预示着有入侵，没有警报预示着没有入侵。现在把注意力集中到 $P(I|A)$，可以用**贝叶斯理论**(Bayes' theorem)计算出它的值：

$$P(I \mid A) = \frac{P(I).P(A \mid I)}{P(I).P(A \mid I) + P(\neg I).P(A \mid \neg I)}$$

$$= \frac{0.000\ 02.P(A \mid \neg I)}{0.000\ 02.P(A \mid \neg I) + 0.999\ 98.P(A \mid \neg I)}$$

现在来看错误警报比率 $P(A|\neg I)$ 对 $P(I|A)$ 的影响。即便是在一个真实情形中算是很好的的警报比率 $P(A|I)=0.8$，一个似乎不错的错误报警比率 $P(A|\neg I)=0.000\ 1$ 产生 $P(I|A)\approx 0.14$。也就是说，6 到 7 个警报预示一个真实的入侵。如果一个安全管理员要调查系统的每个警报，那么这个错误警报比率是极其浪费人力的，这也会使得管理员很快就放弃对警报的调查。

这个例子反映了一个普遍的 IDS 原理：要想有实用性的话，IDS 必须提供一个极低的错误警报比率。要充分测定正常的系统行为的基准点并不容易，因此，对异常检测系统而言，要达到一个足够低的错误报警比率是一个很大的挑战。然而，研究者们仍然在努力提高异常检测的可用性。

19.6.2 审计和记录

审计跟踪处理(audit-trail processing)是一个常用的入侵检测方法。在这个方法中，把与安全相关的事件都记录到一个审计跟踪表中，然后将记录和攻击签名匹配(在基于签名的检测中)，或者用来做异常行为分析(在异常检测中)。在 UNIX 系统中，syslog 和 swatch 组成了一个简单的程序对，负责创建和分析审计跟踪，并启动响应。程序 syslog 负责创建审计跟踪，并为安全相关消息提供发送工具。在 syslog 根据已有的入侵迹象创建审计跟踪和启动响应时，程序 swatch 就将简单地基于签名的检测应用到 syslog 审计跟踪中。

当 UNIX 系统安装 syslog 之后，系统每次启动时都会创建一个叫做 syslogd 的监护进程。进程 syslogd 等待来自各种不同来源的消息，并根据配置文件 syslog.conf 的指示将这些消息发送出去(通常在*/etc/*目录下)。只要缺省地包含日志驱动流*/dev/log*，进程 syslogd 就能接收各种来源的消息，这些来源包括分布在网络中的其他机器。文件

syslog.conf由一些对组成，每个对包含一个选择器域和一个行为域。由选择器域识别应用相应行为的消息。可以应用到消息的行为类别包括发送消息到一个文件，到一个用户列表，到其他机器上的 syslogd 监护进程，或者通过一个 UNIX 管道将消息发送到另外一个程序。这样，syslogd 就提供了一个集中发送各种安全相关消息的工具。

2001 年 4 月发布的 swatch 程序，可以从 http://www.stanford.edu/~atkins/swatch/ 下载。它处理由监护进程 syslogd 发送给它的消息并且在检测到某些模式时完成启动动作。没必要将这些消息直接发送给 swatch，可以将它们写到一个 swatch 要读取和分析的文件中。程序 swatch 将每条审计消息和配置文件中的正则表达式进行匹配。对于正则表达式来说，如果匹配的话，相应要执行的动作也会被同时列出。允许执行的操作有：允许在 swatch 的控制终端显示一个批处理命令行，唤醒 swatch 的控制终端，通过电子邮件或者 write 命令发送一条消息到一个用户列表，或者执行一个带参数程序，参数由该审计消息提供。

swatch 配置文件中列出的每个正则表达式都可以在基于签名的入侵检测中被用做一个原始签名。例如，一个有用的签名可能会和任意包含"permission denied"的字符串匹配，并且指示将一个时钟信号定向到 swatch 的控制终端。然而，swatch 支持的信号都是非常初级的，只能应用于每次只有一个审计信息的情况。由多个活动组成的攻击签名通常由几条审计信息交叉表示。这种方法通常无法检测这种信号。商业化的审计跟踪过程工具提供的过程能力可能会丰富些。

19.6.3 Tripwire

这里有一个简单异常检测工具的例子，被称为 Tripwire **文件系统**完整性检测工具，由美国普渡大学为 UNIX 设计。Tripwire 执行的前提是：许多指令导致系统的目录和文件有了不正常改动。例如，为了以后能够轻易获得访问权限，攻击者可能会更改系统的*/etc/passwd* 文件。入侵者有可能修改系统程序：比如插入特洛伊木马的拷贝，或者在用户 shell 搜索路径中的目录中插入新的程序。为了掩盖踪迹，入侵者可能会删除系统的日志文件。Tripwire 是一个监视文件系统的工具，它监控增加、删除和修改文件等操作，并提醒系统管理员注意这些变化。

Tripwire 的操作由**配置文件** tw.config 控制，该文件中列举了需要监控修改、删除和添加操作的目录和文件。该配置文件中的每个入口都包含一个选择屏蔽字，它指定了那些要监控修改的文件属性。例如，有这样一个选择屏蔽字：它指定监控文件的访问许可，但忽略文件的访问时间。此外，选择屏蔽字还可以指定监控文件内容的改变，或者更具体地说对文件内容应用一个预定义哈希函数而得到的结果的改变也可以被监控。为此，Tripwire 提供了几个抗冲突的哈希函数。如果一个哈希函数是抗冲突的，那么给定一个结果 $f(x)$，就不可能在计算另外一个文件 x' 时得到结果 $f(x')$ 满足 $f(x')=f(x)$。因此，就监控修改而言，监控文件的哈希结果和监控文件本身的效果是一样的，但是，存储文件的哈希结果会比存储

文件本身的拷贝节省多得多的空间。

最初运行时，Tripwire 以文件 tw.config 为输入，并为每个文件或目录计算一个签名，该签名由文件或目录的受监控的属性(索引节点属性和哈希值)组成。这些签名存储在一个数据库中。以后运行时，Tripwire 同时输入文件 tw.config 和先前存储的数据库，为 tw.config中提及的文件和目录重新计算签名，并将计算结果和数据库中的相应数据(存在的话)做比较。需要向管理员汇报的事件包括以下几类：受监控文件或目录的签名与数据库中存储的不同(修改过的文件)，受监控文件或目录的签名在数据库中并不存在(添加的文件)，数据库中的签名所对应的文件或目录已经不再存在(删除的文件)。

虽然 Tripwire 适用于很多种攻击，但是它确实也有它的局限性。首先，要防止未授权用户修改 Tripwire 程序和相关文件，特别是数据库文件。因此，要将 Tripwire 极其相关文件存储在防篡改的介质中，比如写保护的磁盘，或者严格控制访问可靠的服务器。但这样做也会带来一些不便，在合法更新文件和目录之后，更新数据库的操作就不那么方便了。第二个局限是：一些安全相关的文件被假定要随时更新，比如系统日志文件，但 Tripwire 不能区分授权修改和未授权修改。因此，如果攻击修改了一个正常情况下也会更改的系统日志，它将有可能从 Tripwire 的检测能力中逃脱。在这种情况下，Tripwire 能做的顶多就是检测一些显然存在不一致性的情况(例如，如果日志文件收缩了)。Tripwire 有商业版本，也有免费版本。

19.6.4 系统调用监控

最近有一个比较投机的异常检测方式，叫做系统调用监控。这个方法监控的是系统调用进程，在一个进程偏离预期的系统调用行为时进行实时检测。程序会根据程序中可能的执行路径的集合隐含地定义它需要的系统调用的序列。这个方法就是对程序的这一行为实施杠杆作用。在庞大而复杂的程序中，系统调用序列的集合是巨大的，并且难以确定，也不可能显式地存储它。然而，这个方法在设计时采用了一种简化形式来描述“常用”的系统调用行为，并将实际的系统调用和这个简化描述做比较，根据比较结果来检测异常行为。这种方法可以检测出企图接管进程的攻击，例如，通过在程序中利用缓冲区溢出的缺陷，迫使进程中止执行原来的代码，转而执行攻击者的代码。如果攻击者的代码包含一个异常的系统调用序列，那么这个入侵检测系统应该可以把它检测出来并采取相应措施。

这里有一例该方法的应用，是由美国新墨西哥州大学针对 UNIX 进程设计的。在这个例子中，程序的“常用”的系统调用序列是通过在各种输入上运行程序后得出的，包括实际的用户输入，也包括专门为了从程序中抽取一系列行为而设计的输入。然后记录所产生(产生时忽略参数)的系统调用序列。例如，有这样一个序列：

```
open,read,mmap,mmap,open,getrlimit,mmap,close
```

这个序列可以组装成一个表，用于指出对于每个系统调用，有哪些系统调用可以跟它的

距离为 1、2 等，直到 k 的位置后面。例如，选定 $k=3$。现在检查一下处于第一位的 open，可以看到 read 跟随在距离为 1 的位置上，而 mmap 跟随在距离为 2 和 3 的位置上。同理，mmap 跟随在 read 后面距离为 1 和 2 的位置上，而 open 跟随在距离为 3 的位置上。对于每个系统调用由此类推，得出如图 19.3 所示的表格。

系统调用	距离 =1	距离 =2	距离 =3
open	read gertlimit	mmap	mmap close
read	mmap	mmap	open
mmap	mmap open close	open gertlimit	gertlimit mmap
gertlimit	mmap	close	
close			

图 19.3 从系统调用序列中得出的数据结构

然后存储这张表。以后执行这个程序时，就将系统调用序列和这张表做比较，寻找差异。例如，如果观察到以下调用序列(即，有一个 mmap 被 open 替换了)：

```
open,read,mmap,open,open,getrlimit,mmap,close
```

系统就会注意到以下差异：在 open 后面距离为 3 的位置上跟随着一个 open 调用；read 后面距离为 2 的位置上跟随着一个 open 调用；open 后面距离为 1 的位置上跟随着一个 open 调用；open 后面距离为 2 的位置上跟随着一个 getrlimit 调用。IDS 是否将这次行为视为一次入侵，这可能取决于观察到的差异的数目，也许是占总的可能差异的比率，也可能取决于特定的系统调用的数目。

目前主要将这种方法及其类似方法应用到根进程中，如 sendmail，目的是检测各种入侵。根进程通常只有有限的几个行为，并且相对稳定，因此这种方法很适合它们。此外，由于这些进程的特权地位，由于在可以接受网络连接的网络服务环境中，攻击者可以从远方轻易地刺探到这些进程，因此它们经常会成为攻击的目标。

系统调用监控技术并不是完美的，如果攻击者精心设计攻击，使其对系统调用序列的扰动不足以惊动 IDS，那么这次攻击就会从系统调用监控中逃脱。比如，攻击者可以拷贝一份要攻击的目标软件，并进行一系列入侵测试，直至找到一种只修改很小一部分系统调用序列的攻击方式。如果将系统调用的参数考虑进去，扩展这些检测技术，将会获得额外的安全性。

19.7 密码系统

在一台孤立的计算机中，操作系统控制着这台计算机中的所有连接信道，因此它能够可靠地确定所有进程间连接的发送者和接收者。一台计算机在网络中，情形就大不一样了。

一台网络中的计算机从网线中接收位流，它没有及时可靠的方法来确定发送这些位的机器或者应用程序。同样道理，计算机将位流发送到网络，也不知道最后谁会接收到这些位。

通常是根据网络地址来推断网络消息的潜在发送者和接收者。到达时，网络包会携带一个源地址，比如一个 IP 地址。当计算机要发送消息时，它会通过指定一个目的地址来指定想要的接收者。然而，在强调安全的应用领域，如果认为可以根据包的源地址或者目的地址确定包的发送者或者接收者的话，那么是自找麻烦。一台计算机可能会用一个伪造的源地址发送一条消息，并且，除了目的地址指定的计算机之外，可能另有几台计算机也会收到那个包。例如，通往目的地的路途中的所有路由都会收到这个包。那么，在无法信任请求中指定的来源时，操作系统该如何决定是否允许该请求写文件呢？当操作系统无法确定谁会收到它通过网络发送的文件内容时，它该如何为该文件提供读保护呢？

通常认为，要建立包的源地址和目的地址可靠的网络，无论网络的范围大小如何，其实都是不可行的。因此，惟一可行的替换办法是：减少对网络可靠性的依赖。这正是密码术要做的事。简要地说，**密码术**是用来限制一条消息的潜在发送者和接收者的。现代密码术是基于那些被叫做**钥匙**的秘密。那些钥匙被有选择地分布到网络中，用于加工消息。在密码术的帮助下，消息的接收者就能够确定消息是否来自某台持有特定钥匙的计算机，该钥匙是消息的来源。同样道理，发送者可以加密它的消息，这样只有拥有特定钥匙的计算机才能破解该消息；在这种情况下，钥匙成了*目的地*。和网络地址不同，攻击者无法从由钥匙产生的消息中推测出钥匙，也无法从其他公共信息中推测出钥匙。因此，在限制消息的发送者和接收者方面，密码术提供的方法要可靠得多。

19.7.1 验证

验证就是限制消息的潜在发送者的集合。验证算法使得消息的接收者确定消息是否自持有特定钥匙的计算机。更精确地说，验证算法由以下组件构成：

- 一个钥匙集合 K。
- 一个消息集合 M。
- 一个验证集合 A。
- 一个函数 $S:K\to(M\to A)$。也就是说，任意一个 $k\in K$，$S(k)$是一个根据消息产生验证者的函数。S 和 $S(k)$，对于 k 取任意合法值，都必须是高效可计算函数。
- 一个函数 $V:K\to(M\times A\to\{\text{true},\text{false}\})$。也就是说，任意一个 $k\in K$，$V(k)$是一个验证消息的验证者函数。V 和 $V(k)$，对于 k 取任意值，都必须是高效可计算函数。

验证算法必须具备的临界属性是：对于一条消息 m，一台计算机能够产生一个属于 A 的验证者 a，并且只有在计算机拥有 $S(k)$时 $V(k)(m,a)$的值才为 true。因此，持有 $S(k)$的计算机都能够产生消息的验证者，这样其他持有 $V(k)$的计算机就能够验证这些验证者。然而，如果一台计算机没有 $S(k)$，那么它就无法产生可以用 $V(k)$验证的消息验证者。验证者

通常是暴露的(例如它们跟消息一起被发送到网络上),因此要确保攻击者无法根据验证者推测出 $S(k)$。

验证算法有两种。在**消息验证码**(message authentication code, MAC)中,对 $V(k)$的认知和对 $S(k)$的认知是等同的,可以从其中任意一个推算出另外一个。因此,保护 $V(k)$和保护 $S(k)$同等重要。这里有一个简单的 MAC 例子,定义 $S(k)(m)=f(k,m)$,其中 f 是一个单向函数(即无法从函数的结果推算出函数的第一个参数),并且函数 f 的第二个参数是抗冲突的(也就是说,无法找到同时满足条件 $f(k,m)=f(k,m')$和条件 $m'\neq m$ 的值 m')。请注意,计算 $S(k)$和 $V(k)$时都需要用到 k;也就是说,如果能够计算出其中一个,必定能够计算出另一个。

第二种主要的验证算法是**数字签名**(digital signature)算法,所产生的验证者被称为数字签名。在数字签名算法中,无法从 $V(k)$推算出 $S(k)$,特别是 V,它是一个单向函数。因此,没有必要将 $V(k)$保密,实际上可以放心地将它散布到公共的网络中。因此,将 $V(k)$称做**公钥**(public key),相反地将 $S(k)$(或者单独的 k)称为密钥(private key)。将在这里描述一个数字签名算法,一个以设计者的名字命名的算法,RSA 算法。在 RSA 的策略中,钥匙 k 是一个有序对(d,N),N 是两个很大的随机素数 p 和 q(例如 p 和 q 都有 512 bit)的产物。签名算法是 $S(<d,N>)(m)=f(m)^d \bmod N$,其中 f 是一个抗冲突函数。验证算法是 $V(<d,N>)(m,a)\equiv(a^e \bmod N=f(m))$,其中 e 满足条件 $e^d \bmod (p-1)(q-1)$。普遍认为,持有 $V(<d,N>)$(即持有$<e,N>$而非 d)的计算机不可能推算出 $S(<d,N>)$(也就是 d)。

19.7.2 加密

加密是一种约束消息的可能接收者的方法。加密是验证的补充,为了强调加密算法,现在将这两者放在一个同等重要的地位。在加密算法的帮助下,消息的发送者可以做到只让那些拥有特定钥匙的计算机读取该消息。更精确地说,加密算法由以下组件构成:

- 一个钥匙集合 K。
- 一个消息集合 M。
- 一个密文集合 C。
- 一个函数 $E:K\rightarrow(M\rightarrow C)$。也就是说,对任意 $k\in K$,$E(k)$是一个根据消息产生密文的函数。E 和任意 k 的 $E(k)$都必须是高效可计算函数。
- 一个函数 $D:K\rightarrow(C\rightarrow M)$。也就是说,对任意 $k\in K$,$D(k)$是一个根据密文推算消息的函数。D 和任意 k 的 $D(k)$都必须是高效可计算函数。

加密算法必须提供的基本属性是:给定一个密文 $c\in C$,计算机只有在拥有 $D(k)$时才能计算出满足条件 $E(k)(m)=c$ 的 m。因此,持有 $D(k)$的计算机可以将密文解密,得到相应的明文。没有 $D(k)$的计算机都无法解密密文。密文通常是暴露的,因此要尽力保证无法从密文推算出 $D(k)$。

主要的加密算法也有两种。第一种叫做**对称加密算法**(symmetric encryption algorithm),可以从 $D(k)$推算出 $E(k)$,也可以从 $E(k)$推算出 $D(k)$。因此,要对 $E(k)$和 $D(k)$实施同等程度的安全保护。在过去的20年中,美国民用领域最常用的对称加密算法是**数据加密标准**(data encryption standard, DES),该算法已经被(美国)国家标准和技术协会(NIST)采用。然而,到2001年4月为止,DES在许多应用领域中都被认为是不安全的,因为它采用的钥匙的长度为56 bit,许多拥有中等计算资源的计算机可以用穷尽法搜索出正确的钥匙。因此,NIST采用了一种新的加密算法来替代DES,该算法被称为**高级加密标准**(advanced encryption standard, AES)。从2003年开始将会选择AES。

在**非对称加密算法**(asymmetric encryption algorithm)中,无法从 $E(k)$推算出 $D(k)$,因此不需要为 $E(k)$提供保密措施,实际上完全可以将 $E(k)$散布到公共网络中。$E(k)$是公共钥匙,$D(k)$(或者仅仅是 k)是私有钥匙。有趣的是,RSA签名算法的底层机制也可以用来产生一个非对称加密算法。同样地,钥匙 k 是一个有序对(d,N),N 是两个巨大的、随机选定的素数 p 和 q 的产物。加密算法是 $E(d,N)(m)=m^e \bmod N$,其中 e、d 和 N 要满足的约束条件跟前文提到的一样。解密算法是 $D(<d,N>)(c)=c^d \bmod N$。

19.7.3 举例:SSL

SSL 3.0是一个确保两台计算机之间安全连接的密码协议,也就是说,这两台计算机中的任意一台都可以将消息的发送者和接收者限定为另一台计算机。它是保证网页浏览器和网页服务器安全连接的标准协议,因此它很可能是当今因特网中应用最为广泛的密码协议。SSL是一个很复杂的协议,它有许多选项。这里来看一个SSL的简单变种,而且只是一个非常简单非常抽象的形式,因为本节的目的在于分析SSL对加密的基本原理的使用情况。

SSL协议的初始化是由一个安全连接到服务器的客户端完成的。在这个协议之前,假定服务器 s 已经从另外一个被称为**身份认证**(certification authority, CA)的机构中获得了一个**证书**(certificate),该证书用符号 cert_s 表示。这个证书的内容如下:

- 服务器的各种属性 attrs,例如它的惟一的*特别的*名字,还有它的*普通的*(DNS)名字。
- 一个为服务器设计的公共加密算法 $E(k_s)$。
- 一个有效区间 interval,即证书有效的时间区间。
- 一个由CA根据以上信息提供的数字签名 a,也就是说 $a=S(k_{CA})(<\text{attrs},E(k_s),\text{interval}>)$

在这个协议之前,假定客户端已经为CA获得了公共验证算法 $V(k_{CA})$。在申请网页服务时,用户的浏览器已经从持有某个身份认证的验证算法的卖主中取得了算法。用户可以根据自己的需要往身份认证中添加或删除验证算法。

当客户端 c 连接 s 时,它会向服务器发送一个28 B的随机值 n_c。服务器用它自己的值 n_s,加上它的证书 cert_s,来响应客户端。客户端确认 $V(k_{CA})(<\text{attrs},E(k_s),\text{interval}>,a)=\text{true}$,

并确认当前时间在有效区间 interval 内。如果两个条件都满足的话，接着客户端就会产生一个随机的 46 B 的**预先控制秘密**(premaster secret)pms 并向服务器发送 cpms=$E(k_s)$(pms)。服务器恢复 pms=$D(k_s)$(cpms)。现在客户端和服务器都有了 n_c、n_s 和 pms，它们可以各自计算出一个 48 B 的共享**控制秘密**(master secret)ms=$f(n_c, n_s, \text{pms})$，其中 f 是一个单向、抗冲突函数。因为只有服务器和客户端知道 pms，因此只有它们可以计算出 ms。此外，ms 对 n_c 和 n_s 的依赖保证了 ms 会是一个*新鲜*的值，也就是说，之前协议运行时肯定没有使用过当前这个 ms 值。此时，客户端和服务器都开始根据 ms 计算以下这些值：

- 用来加密客户端到服务器消息的一个对称加密钥匙 k_c^{crypt}。
- 用来加密服务器到客户端消息的一个对称加密钥匙 k_s^{crypt}。
- 用来根据客户端到服务器消息产生验证者的一个 MAC 产生钥匙 k_c^{mac}。
- 用来根据服务器到客户端消息产生验证者的一个 MAC 产生钥匙 k_s^{mac}。

客户端要向服务器发送消息 m 时，它发送的是：

$$c=E(k_s^{\text{crypt}})(<m, S(k_c^{\text{mac}})(m)>)$$

服务器端在接收 c 时，它将会恢复以下内容：

$$<m, a>=D(k_c^{\text{crypt}})(c)$$

如果 $V(k_c^{\text{mac}})(m, a)=\text{true}$，服务器就接受消息 m。同样道理，服务器向客户端发送消息 m 时，它发送的是：

$$c=E(k_s^{\text{crypt}})(<m, S(k_s^{\text{mac}})(m)>)$$

客户端恢复以下内容：

$$<m, a>=D(k_s^{\text{crypt}})(c)$$

如果 $V(k_s^{\text{mac}})(m, a)=\text{true}$，客户端就接受消息 m。

采用这个协议后，服务器就能够将消息接收者限制为那个产生 pms 的客户端，并将它接收的消息的发送者也限制为同一个客户端。同样道理，客户端也能够将消息的接收者和发送者限制为知道 $S(k_s)$的那个群体。在许多应用领域，比如那些网页事务，客户端都需要验证究竟谁知道 $S(k_s)$。这是证书的用途之一，特别是 attrs 域，它包含了客户端用来确定身份的信息，比如它所连接的服务器的域名。在那些服务器也需要知道客户端信息的应用领域，SSL 提供了一个选项，通过它客户端可以向服务器发送一个证书。

19.7.4 密码术的使用

通常采用层的方式组织网络协议，每一层都是低一级的相邻层的客户。也就是说，当一个协议产生了一条消息，要将这条消息发送到另外一台机器上的对等协议时，它的做法是：将消息传递给网络协议栈中比它低一级的协议，通过它传送给另外一台机器上的对等协议。例如，在一个采用 IP 协议的网络中，TCP 协议(*传输层*协议)就是 IP 协议(网络层协议)的客户；为了将 TCP 包传送给 TCP 连接的另一端的 TCP 对等体，首先要将 TCP 包向下传递给 IP 协议。IP

协议在 IP 包中封装 TCP 包，然后，同样道理，IP 包被向下传递给数据链路层，然后通过网络传递给 IP 协议在目的计算机中的对等体。之后这个 IP 协议对等体将 TCP 包向上传递给那台机器的 TCP 对等体。总而言之，OSI 参考模型已经在数据网络中得到了广泛的应用。一般比较好的网络书都会详细地描述 OSI 参考模型。

密码术可以插入到这个模型的任意层。SSL 协议(19.7.3 小节)中在传输层提供安全。标准化的网络层安全(或 **IPSec**)中定义的 IP 包格式中允许插入验证者、及对数据包内容进行加密。**虚拟私有网络**(**virtual private network**)开始普遍采用 IPSec。其他一些协议也开始在应用中发展起来。

在协议栈的哪个位置插入密码保护最好呢？一般来说，这个问题没有确定的答案。一方面，如果将保护放在协议栈中一个较低位置的话，就会有更多的协议受到保护。比如，因为 IP 包封装了 TCP 包，加密 IP 包的时候(例如使用 IPSec)就会将封装的 TCP 包的内容隐藏起来。同样道理，IP 包的验证者在检测时也会检测 TCP 的头信息的修改情况。

另一方面，如果将保护放在协议栈中一个较低位置的话，恐怕就无法为处于较高层的协议提供足够的保护。例如，一台运行 IPSec 的应用程序服务器可以验证有请求的客户端。然而，如果要验证客户端的用户的话，可能就需要另外用一个应用层的协议，比如，让用户输入一个密码。

19.8 计算机安全分类

美国国防部的可靠计算机系统评价标准部门将系统安全分成 4 类：A、B、C 和 D。D 类是最低一级的分类，即安全性最小的那类。D 类仅由一个类组成。当系统无法达到其他 3 类的要求时就使用 D 类。例如，MS-DOS 和 Windows 3.1 都是属于 D 类的。

C 类，就是紧接着 D 类的下一个安全层，它用的是审计能力，为用户及其行为提供任意保护和责任。C 类分为两层：C1 和 C2。C1 类的系统组合了若干种形式的控制，用户可以借助它来保护私有信息，并保护自己的数据不被其他用户意外地读取或者破坏。在 C1 环境中，合作用户在同一个敏感层次上访问数据。大部分 UNIX 版本都属于 C1 类。

在一个计算机系统(软件、硬件、防火墙)内，正确地强制执行一个安全方针的所有保护系统的总和，被称为**可靠的计算机基础**(trusted computer base，TCB)。C1 系统的 TCB 控制着用户和文件间的访问，它采用的方式是：允许用户指定和控制指定的个人或定义的组对对象的共享。此外，在用户开始任何一项需要 TCB 仲裁的活动之前，TCB 会要求用户先进行自我识别。这个识别由一个受保护的机制或者密码来完成；TCB 会保护验证数据，使其免遭未授权用户的非法访问。

在 C1 类系统的基础上，C2 类系统增加了一个个体层的访问控制。例如，可以将一个文件的访问权限指定到一个单一个体的层次上。此外，在个体识别的基础上，系统管理员可以选择

性地审计任意一个或多个用户的活动。TCB 系统还会保护自己的代码和数据结构。此外，如果之前有一个用户生成了一些信息，而且已经将存储对象释放并返回给系统，则别的用户还是无法访问到那些信息。UNIX 的一些特别的安全版本被证明是属于 C2 层的。

B 类强制保护系统拥有 C2 类系统的所有属性，此外，它们还在每个对象上贴了敏感标签。B1 类的 TCB 维持了系统中每个对象的安全标签；这个标签用于属于强制访问控制的决策。例如，一个在机密层的用户不能访问一个处于更敏感的安全层的文件。在每个可供人读取的输出中，TCB 都会在页眉和页脚标注敏感层次。除了正规的用户名一密码验证信息之外，TCB 还维持着个体用户的清除和授权，并且至少提供两层安全。这些层是等级制的，也就是说，只要一个对象所处的安全层次不比某用户的高，该用户就可以访问它。例如，一个处于秘密层的用户，他不需要别的访问控制权限，就已经可以访问一个处于机密层的文件。进程间通过使用不同的地址空间来实现隔离。

B2 类的系统为每个系统资源（如存储对象）扩展了敏感标签。系统为每个物理设备设置了最小和最大安全层次，系统用这两个极值来强制执行设备所处物理环境强加在设备上的限制。此外，B2 系统还支持转换信道，支持对那些利用转换信道的事件的审计。

给定一个对象，有些用户和组不具备访问该对象的权限，B3 系统可以创建一个指示这些用户和组的访问控制列表。TCB 还包括一个监控机制，监控对象为那些预示着违反安全策略的事件。该机制会向安全管理员通报该事件，如果有必要的话，它还会以一种破坏性最小的方式中止该事件。

最高层的分类是 A。A1 类系统在功能上等价于一个 B3 系统体系，但采用的是正式的设计规则和检验技术，这让系统有了一个高度的保证，TCB 已经正确实现该保证。或许可以由可靠的人，以一种可靠的工具来设计比 A1 类更安全的系统。

使用 TCB 只能保证系统可以强制执行一个安全策略的各个方面；但 TCB 系统不会指定安全策略的内容。通常，一个给定的计算环境会开发一种安全策略以用于**认证**（certification），并且会有一个被安全机构**认可**的计划，如国家计算机安全中心。某些计算环境可能需要别的认证，比如由 TEMPEST 的认证，该认证的目的在于防御电子窃听。例如，一个 TEMPEST 认证的系统可能会将一些终端屏蔽起来，目的是防止电磁泄露。这层屏蔽确保该终端显示的信息不会被位于屏蔽层之外的仪器探测到。

19.9 例子：Windows NT

微软公司 Windows NT 是一个比较新的操作系统，它支持一系列的安全特性和安全层次；它的安全层次的范围从最小限度的安全开始，直到官方标准的 C2 层的安全。NT 中缺省的安全层是最小限度的安全，不过系统管理员可以将系统的安全层次配置成需要的层次。*C2config.exe* 是一个很好用的程序，管理员可以用它来帮助选择需要的安全设置。这个小节

中,要检查 Windows NT 中用来实现安全功能的特性。要了解更多关于 Windows NT 的信息和背景知识的话,可以去看第二十一章的内容。

NT 的安全模型是基于用户账号这个概念的。NT 允许创建任意个用户账号,并允许以任意方式组织这些账号。可以根据需要允许或者拒绝访问系统对象的请求。系统通过一个具有惟一性的安全 ID 来识别用户。当用户登录时,NT 会为用户创建一个包含安全 ID 的**安全访问标记**(security access token),为用户所在组创建安全 ID,及一个由用户拥有的特权组成的列表。用户特权有哪些呢?备份文件和目录,关机,以互动方式登录,更改系统时钟,等等,都是特权。NT 代表用户运行的每个进程都会得到一份访问标记的拷贝。无论何时,当用户或者代表用户的进程要访问对象时,系统都会用访问标记中的安全 ID 来允许或者拒绝它们对系统对象的访问。尽管 NT 的模块化设计已经提供了自定义的验证包,系统通常还是通过用户名和密码来验证一个账号。例如,一个视网膜(或眼睛)扫描仪可以用来鉴别用户的身份是否与实际相符。

NT 采用了**主题**(subject)这样一个概念,以确保用户运行的程序对系统的访问权限是系统授权给用户的访问权限的子集。一个**主题**由用户的访问标记和代表用户运行的程序组成,它被用来跟踪和管理用户运行的每个程序的各种许可。NT 运行时用的是"客户端-服务器"模型,因此采用了两类主题来控制访问。这里有一个**简单主题**的例子,即大多数用户在登录后都要执行的那一个程序。系统根据用户的安全访问标记给这个简单的主题分配一个**安全关联**(security context)。服务器主题被实现为一个受保护服务器的进程,在代替客户端执行任务时,那个受保护的服务器用的是客户端的安全关联。这种允许一个进程接纳另外一个进程的安全属性的技术通常被称为**扮演**(impersonation)。

从 19.5 节讨论的内容中,得出这样一个结论:审计是一个很有用的安全技术。NT 中安装了审计技术,并且允许监控多个通用的安全线程。以下是一些可以用来跟踪线程的审计的例子:登入登出事件失败的审计,其目的是检测随机密码的破解;登入登出成功的审计,其目的是检测异常时段的登录活动;对可执行文件的写访问成功/失败审计,其目的是跟踪病毒的爆发;对文件访问成功/失败的审计,其检测对敏感文件的访问。

NT 中用**安全描述符**(security descriptor)来描述对象的安全属性。安全描述符包括对象持有者的安全 ID,一个仅用于 POSIX 子系统的组安全 ID,一个任意的访问控制列表,它识别被允许或不被允许访问该对象的用户和组,一个系统访问控制列表,它控制系统产生的审计消息。例如,文件 *foo.bar* 的安全描述符可能有持有者 avi 和下面这个任意的访问控制列表:

- avi——所有访问权限
- 组 cs——访问权限读、写
- 用户 cliff——没有权限

此外,它可能还会有一个系统访问一控制列表,内容是每个用户写权限的审计。一个访问一控制列表由访问一控制入口组成,这些入口包括,个体的安全 ID,一个在对象上定义了所有可能行为的访问屏蔽字,屏蔽字中每个行为的值为 AccessAllowed 或 AccessDenied。NT 中文

件的访问权限有： ReadData、 WriteData、 AppendData、 Execute、 ReadExtendedAttribute、 WriteExtendedAttribute、ReadAttributes 和 WriteAttributes。我们可以来看看它是怎样控制对对象的访问的。

NT 将对象分为两类：容器类对象和非容器类对象。**容器类对象**(container object)，如目录，可以在逻辑上包含其他对象。NT 中的缺省做法是：如果在一个容器类对象内创建一个新对象，这个新对象将会从父对象继承许可权限。如果用户将一个文件从一个目录拷贝到一个新目录，那么文件将会继承目标目录的许可权限。**非容器类对象**(noncontainer object)不会继承别的许可权限。

然而，如果修改一个目录的某个许可权限，它的文件和子目录的相应的许可权限并不会自动改变；如果需要的话，用户可以显式地更改许可权限。同样地，当用户将一个文件移到一个新目录下时，文件当前的许可权限也会跟着文件移动。

此外，系统管理员可以全天禁止或在一天中的部分时间禁止系统的打印机打印，并且借助 NT 性能监视器帮助侦察查找问题。NT 中的某些特性提供了一个很好的安全计算环境。在缺省情况下，并不是所有这些特性都是处于激活状态的，这样做的目的是提供一个接近于个人计算机用户习惯的环境。在一个真实的多用户环境中，系统管理员应该利用 NT 提供的特性规划并实现一个良好安全计划。

19.10 小 结

保护是一个内部问题。安全要同时考虑计算机系统和环境——人，建筑物，商务，贵重物品和威胁——使用计算机系统的环境。

要很好地保护计算机系统中存储的数据，防止它们被未授权者访问，被恶意地损坏或更改，被意外地引入不一致性。和防止数据被恶意访问相比，防止数据意外地丢失一致性要容易得多。完全杜绝恶意滥用计算机存储的数据的现象是不现实的，不过可以采取一些措施，让罪犯付出足够高的代价，这样来阻止绝大多数甚至全部未授权访问。

尽管计算机系统已经提供了各种各样的授权机制，但还是无法为高敏感数据提供足够的保护。在这种情况下，可以考虑将数据加密。除非阅读者知道如何解密密文，否则它无法读取已加密的数据。

除了标准的用户名和密码保护之外，还有一些别的验证方法。一次性密码每次都会改变发送的数据，这样可以免重复攻击。双因素验证要求有两种形式的验证，如带有一个激活 PIN 的硬件计算器。随着生物器件的发展，这些方法大大地降低了伪验证的成功概率。

有几种既可以攻击个体计算机，也可以攻击群体计算机的攻击方式。病毒和蠕虫可以自我繁殖，有时会感染数千台计算机。攻击者可以利用栈和缓冲的溢出改变自己的系统访问权限层次。拒绝服务攻击会阻碍对目标系统的合法使用。

有几种防止或者检测意外安全事故的方法。包括入侵检测系统、系统事件的审计和日志记录，系统软件变动的检测和系统调用监测。密码术既可以用在系统内，也可以用在系统间，可以有效地防止黑客中途截取信息。

习题十九

19.1 别的用户可以通过各种途径得知密码。试问有没有一种简单的方法来检测是否发生密码泄露事件？请解释你的答案。

19.2 如果将所有密码的列表保存在操作系统内部，在这种情况下，如果用户设法看到了这个列表的话，密码保护就失效了。请提供一个可以避免这个问题的策略。（提示：采用不同的内部和外部表示方法。）

19.3 UNIX 系统增加了一个尚处于实验阶段的特性：即允许用户将程序 watchdog 连接到一个文件，这样一来，无论何时，只要有程序请求访问这个文件，watchdog 就会被调用。然后由 watchdog 允许或拒绝对该文件的访问请求。请从安全的角度出发，分别讨论使用 watchdog 的两个好处和两个坏处。

19.4 UNIX 系统中的程序 COPS 在一个给定的系统中扫描可能存在的安全漏洞，并向用户报告可能存在的问题。使用这种安全系统的两种潜在的危险是什么？怎样限制或者消除这些问题呢？

19.5 讨论一种连接到因特网的系统的管理员可以用来限制或消除蠕虫带来的危害的系统设计方法，这种方法的缺点是什么？

19.6 你是赞成还是反对官方对 Robert Morris, Jr. 的审判 —— 因为他创建并传播了因特网 t 蠕虫。

19.7 为一个银行计算机系统列举 6 种安全措施，并陈述列表中的每个条款与物理环境、人或者操作系统安全的关系。

19.8 对计算机系统中存储的数据进行加密的两种好处是什么？

推荐读物

Hsiao 等[1979]、Landwehr[1981]、Denning[1982]、Pfleeger[1989]和 Russell 等[1991]给出了关于安全性的一般论述。Lobel[1986]的书也有一般的论述。

Rushby[1981]和 Silverman[1983]论述了安全系统的设计与验证问题。Schell[1983]描述了对于多处理器微机的一个安全内核。Rushby 和 Randell[1983]描述了一个分布式安全系统。

Morris 和 Thompson[1979]论述了密码安全问题。Morshedian[1986]给出了防止偷盗密码的方法。Lamport[1981]考虑了在非安全通信中的密码授权问题。Seely[1989]论述了密码破解问题。Lehmann[1987]和 Reid[1987]论述了计算机非法闯入的问题。

Grampp 和 Morris[1984]、Wood 和 Kochan[1985]、Farrow[1986b]、Farrow[1986a]、Filipski 和 Hanko[1986]，Hecht 等[1988]、Kramer[1988]、Garfinkel 和 Spafford[1991]给出了关于 UNIX 安全性问题的讨论。Bershad 和 Pinkerton[1988]给出了对 BSDUNIX 的 watchdog 的扩展。Farmer 在美国普渡大学写了对 UNIX 的 COPS 安全性扫描包。用户可以用 ftp 程序通过因特网在 ftp.uu.net 主机的/pub/security/cops 目录获得。

Spafford[1989]给出了一个关于因特网 t 蠕虫的详细的技术性讨论。Spafford 和其他三篇关于因特网蠕虫的文章在 Communications of the ACM(Volume 32, Number 6, June 1989)上发表。

Diffie 和 Hellman[1976]、Diffie 和 Hellman[1979]是最早提出使用公开密钥方案的研究者。19.7.2 中的以公开密钥方案为基础的算法是由 Rivest 等[1978]开发的。Lempel[1979]、Simmons[1979]、Gifford[1982]、Denning[1982]和 Ahituv 等[1987]研究了计算机系统中的加密技术应用。Akl[1983]、Davies[1983]、Denning[1983]和 Denning[1984]提供了关于数字信号保护问题的论述。

美国联邦政府当然关心安全性问题。*国防部信任计算机系统评估标准* DoD[1985]，也被称做*桔黄皮书*，描述了一系列安全级别和每个级别的计算机操作系统要满足的特性。读它是理解安全性问题的一个好的起点。*Microsoft Windows NT Workstation Resource Kit*(Microsoft[1996])描述了 NT 的安全模型以及如何使用这个模型。

Rivest 等[1978]中展现了 RSA 算法。关于 NIST 的 AES 活动的信息可以在 http://www.nist.gov/aes/找到。在这个站点上还可以找到其他的美国加密技术标准。更加完整的关于 SSL 3.0 的论述可以在 http://home.netscape.com/eng/ssl3/找到。在 1999 年，SSL 3.0 经过轻微的修改并以 TLS 的名字呈现在 IETF 请求注解(RFC)上。

19.6 节中演示错误警告次数对 IDS 效果的影响的例子是以 Axelsson[1999]为基础的。Hansen 和 Atkins[1993]中可以找到 swatch 程序和 syslog 一起使用的完整的描述。19.6.3 小节中的关于 Tripwire 的描述是根据 Kim 和 Spafford[1993]的内容书写的。19.6.4 小节中的描述和例子来自于 Forrest 等[1996]的内容。

第七部分　案例研究

现在可通过描述真实操作系统而对本书前面所述的概念进行整合。两个这样的系统将被详尽描述——Linux 和 Windows 2000。选择 Linux 进行讲解有几个原因：它很流行，它是免费可用的，它具备了完整特征 UNIX 系统的代表性特征。这对于学习操作系统的学生而言提供了一个机会去读、进而修改真实操作系统源代码。

本部分还详尽描述了 Windows 2000。微软公司新推出的这个操作系统正在逐步获得广泛用，这不仅体现在单机市场，而且也体现在工作组服务器市场。之所以选择 Windows 2000，是因为它为我们研究现代操作系统提供了一个机会，其设计和实现方法与 UNIX 大为不同。

另外，本部分还简要论述了其他极具影响的操作系统。笔者对所展示的内容进行了排序，以突出各系统间的相似点和不同点；它并不是严格按照时间顺序展现的，并未反应系统的相对重要性。

最后，提供了其他三个系统的在线资料。Free BSD 是另一种 UNIX 系统。然而，尽管 Linux 采用了从多个 UNIX 系统中结合特性的方法，Free BSD 是基于 UNIX 的 BSD 模型的。如同 Linux 一样，Free BSD 源代码是免费可用的。Mach 操作系统是一个现代操作系统，提供了与 BSD UNIX 的兼容性。Nachos 系统是一个指导型操作系统，旨在作为本科或研究生一年级课程的项目。它允许学生自己创建操作系统中的重要片断，并观察他们工作的效果。

第二十章 Linux 系 统

附录 A 讨论了 4.3BSD 操作系统的内部工作方式。BSD 是一种类似于 UNIX 的系统，而 Linux 则是近几年来颇受欢迎的另一个类似于 UNIX 的系统。这一章将回顾 Linux 的历史及发展过程，同时还将描述 Linux 系统呈现给用户与程序员的接口，而这些接口很大程度上归功于 UNIX 的传统。还将讨论 Linux 运行这些接口的内部方式。然而，从一开始 Linux 就被设计成尽可能多地运行多种标准 UNIX 应用工具，所以它与现存的 UNIX 执行方式具有共同特性。在此不再重复附录 A 中对 UNIX 的基本描述。

Linux 是一种发展很快的操作系统。本章将专门描述 1999 年 1 月发布的 Linux 2.2 版内核。

20.1 发展历程

Linux 看起来与其他的 UNIX 系统非常相似；事实上，UNIX 的兼容性已成为 Linux 项目的主要设计目标。但是 Linux 毕竟比大多数 UNIX 系统年轻得多。它开始于 1991 年，当时由一个名为 Linus Torvalds 的芬兰学生编写并命名为 **Linux**。这个很小但功能完整的内核可运行在 80386 处理器上。在 Intel 的 PC 兼容的 CPU 系列中，80386 是第一个真正的 32 位处理器。

在 Linux 的发展早期，其源代码可以在因特网上免费得到。结果，Linux 的历史成为由来自世界各地的许多使用者合作开发的过程之一，相应地这种情况在因特网的历史上也是独一无二的。最初内核只能部分地执行 UNIX 系统服务程序的一个小子集，但现在 Linux 系统已发展到囊括了大多数 UNIX 功能。

早期 Linux 发展围绕的中心就是操作系统内核——核心(core)，它是一种特权执行程序，其主要功能是管理所有的系统资源，与计算机的硬件直接交互。当然，要实现一个完整的操作系统，所需的内核要比这大得多。区分好 Linux 内核与 Linux 系统很有意义。**Linux 内核**(Linux kernel)是由 Linux 这个团体从零开始开发的一个完全原创的软件。而 **Linux 系统**(Linux system)，正如今天所知道的，它包括很多部分，一些是从零开始编写出来的，一些是从其他的开发方案中借鉴的，而还有一些是在和其他团队的合作当中实现的。

虽然基本的 Linux 系统是应用程序与用户编程的标准环境，但是它不强制性地将任何标准方法与其实现的功能结合为一个整体。当 Linux 日趋成熟时，在 Linux 系统之上出现

了另一个功能层面的需求。一个 **Linux** 版本(Linux distribution)包括了所有的 Linux 系统的标准部分,加上一套能简化初始安装和 Linux 系列升级并且能管理在系统上安装和卸载软件包的工具。现代版本一般也包括了一些工具,如文件系统管理,用户账户的创建与管理,网络的管理,等等。

20.1.1 Linux 内核

第一个 Linux 内核 0.01 版于 1991 年 5 月 14 日发布。它不具有网络功能,只能在 80386 系列 Intel 处理器和个人计算机硬件上运行。并且对设备驱动的支持非常有限。虚拟内存子系统也相当简单并且不支持内存映射文件;然而这个早期的版本甚至也支持写时复制(copy-on-write)共享页面。惟一支持文件系统的是 Minix 文件系统——第一个 Linux 内核就是在 Minix 平台上交叉开发而来的。然而内核却能通过保护地址空间来执行正确的 UNIX 进程。

下一个具有里程碑意义的版本是 Linux 1.0,于 1994 年 3 月 14 日发布。此版本使 Linux 内核在快速发展的 3 年内达到了顶峰。也许其最大的特点是网络:1.0 版支持 UNIX 的标准 TCP/IP 协议,同时也支持网络编程的 BSD 套接字接口。设备驱动器支持的增加使得系统能在以太网(使用 PPP 或者 SLIP 协议)、在串行线或者在调制解调器上运行 IP。

1.0 内核同时也包括一个新的、更为强大的文件系统,不再受原先 Minix 文件系统的限制,并支持一系列高性能磁盘访问的 SCSI 控制器。开发者扩展虚拟内存子系统从而支持用于交换文件的分页技术和任意文件的内存映射(但是 1.0 版只执行只读内存映射)。

这个版本也实现了对一系列外部硬件设备的支持。虽然受限于 Intel PC 平台,但是其所支持的硬件已发展到包括软驱、CD-ROM 设备还有声卡、一系列的鼠标和国际标准键盘。这个内核还能为没有 80387 协处理器的 80386 的用户提供浮点仿真,同时能实现系统五(System V)的具有 UNIX 风格的**进程间通信**(IPC),包括共享内存、信号量机制和消息队列。同时还提供进行动态地可装入和卸载内核模块的简单支持。

从这个意义上讲,开发始于 1.1 版内核系列,但是无数针对 1.0 版的缺陷修正(bug-fix)补丁被陆续地出版。这种模式被采纳作为 Linux 内核标准编号方式的规范:版本末尾是奇数如 1.1、1.3 或者 2.1 编号的内核是**开发内核**(development kernel);而编号以偶数结尾的版本比较稳定,即**产品核心**(production kernel)。对稳定内核的更新仅仅是修正而已,而开发内核则很可能包含更新的和相关的还未测试的功能。

1995 年 3 月,1.2 版本的内核发布了。此版本和 1.0 版一样几乎也没提供功能上的进展,但是它确实支持了更为广泛的硬件,包含新的 PCI 硬件总线体系结构。开发者增加了另外针对个人计算机的新特征——支持 80386 CPU 的虚拟 8086 模式,进而允许对个人计算机进行 DOS 操作系统的仿真。他们更新了网络协议栈,为 IPX 协议提供支持,包含了记账和防火墙功能技术使得 IP 实现更加完善。

1.2 版的内核是最后的仅适用于个人计算机的 Linux 内核。1.2 版 Linux 的源版本包含了部分对 SPARC、Alpha 和 MIPS 的 CPU 的支持，但是直到 1.2 版的稳定版本发布后这些体系结构才开始完全地整合在一起。

Linux 1.2 版把注意力集中在更广泛的硬件支持与现有功能的更彻底的实现上。那时很多新功能正在开发，但是把新代码整合到内核源代码却被推迟到了稳定内核 1.2 版发布以后。结果，1.3 版有大量的新功能增加到了内核上。

此项工作最终于 1996 年 6 月在 Linux 2.0 版中发布，这个版本出现了较大的版本数量的增量需求，这是因为开发了两大主要新功能：支持多体系结构，包括一个完全的 64 位 Alpha 端口和支持多处理器体系结构。基于 Linux 2.0 的版本可应用于 Motorola 68000 系列处理器和 Sun 公司的 SPARC 系统。Linux 的一个派生版本既可运行在 Mach 微内核上，也可运行在个人计算机和 PowerMac 系统上。

2.0 版的变化并不仅限于此。内存管理代码得到实质性的改进，从而为文件系统数据提供了一个统一的高速缓冲存储器，它独立于数据块设备。这个改变的结果是内核大大增强了文件系统和虚拟内存的性能。文件系统高速缓存技术第一次延伸到了网络文件系统，并且也支持了可写存储映射区域。

2.0 版内核同时也大大改进了 TCP/IP 的性能，并且加上了许多新的网络协议，包括 AppleTalk、AX.25 业余无线电网络和 ISDN 的支持。它还增加了加载远程 Netware 和 SMB 网络卷的功能。

为处理可加载模块之间的依赖性和按需自动加载模块，在 2.0 版中另一个主要的改进是支持内部核心线程。运行时的内核动态配置通过一种新型的、标准化的配置界面得到了很大改进，另外出现了与此不相关的新特征，包括文件系统限额和兼容性 POSIX 分时进程调度。

1999 年 1 月 Linux 2.2 版发布，它保留了 2.0 版新增的改进功能，又增加了 UltraSPARC 系统端口。更加灵活的防火墙技术、更好的路由选择、通信量管理、支持 TCP 大窗口及可选肯定应答的运用大大提高了网络性能。Acorn、Apple 和 NT 磁盘现在已可读，NFS 得以改善，一种 NFS 后台程序的内核模式也被加了进来。信号量处理、中断和一些输入/输出，现已锁定于一个比以前更好的层面上，进而提高了 SMP 的性能。

20.1.2 Linux 系统

从很多方面来讲，Linux 内核形成了 Linux 工程的核心，但是其他组成部分构成了完整的 Linux 操作系统。而 Linux 内核完全由针对 Linux 工程的代码拼凑起来，很多其所支持的并构成 Linux 系统的软件并不是其专有的，而是与许多其他 UNIX 操作系统所共有的。特别值得一提的是，Linux 使用的许多工具是 Berkeley 的 BSD 操作系统、MIT 的 X Window 系统和免费软件基金的 GNU 项目的一部分。

这种共享的工具在两种方向上工作。Linux 主系统库源于 GNU 项目，但是 Linux 团体通过对地址省略、低效率、故障等缺点的修改而大大改进了这些系统库。像 **GNU C 编译器**(GNU C Compiler 即 gcc)那样的其他的高质量的系统组成部分已经直接运用于 Linux 之中。Linux 下的网络管理工具从 4.3BSD 最先开发的代码中派生而来。但是更多最新的 BSD 系统，例如 FreeBSD，反过来从 Linux 系统中借用代码，比如 Intel 浮点仿真数学库、PC 声卡设备驱动器。

Linux 系统作为一个整体是通过一个松散的网络来维护的，这个松散的网络得益于通过因特网的开发者的合作，每个小组或个人负责维护某个特定的部件。一些公众 FTP 文档站点充当了这些部件事实上的标准储藏室。**文件系统层标准**(File System Hierarchy Standard)文档也是由 Linux 团体维护的，可以作为对各种系统组成保持兼容性的一种手段。这个标准详细说明了一个标准 Linux 文件系统的全部规划，它决定配置文件、库、系统二进制和运行时间数据文件应该被保存在哪个目录名称下。

20.1.3 Linux 版本

理论上讲，任何人只要从 FTP 站点上取得最新版本的必要系统部件，就能编译安装一个 Linux 系统。在 Linux 的早期发展阶段，这种操作是 Linux 用户必须执行的。然而，随着 Linux 的日趋成熟，不管是个人还是团体都试图通过提供一套标准化的、预编译的程序包来减轻这些工作带来的痛苦，从而使之易于安装。

这些组合，或者版本，所包括的远远不止一个最基本的 Linux 系统。它们典型地包括了附加的系统安装和管理程序，及许多与 UNIX 共有的预编译和预安装工具包，如消息服务器、Web 浏览器、文本处理和编辑工具，甚至还有游戏。

第一个版本通过简单地提供一种解开所有文件到特定位置的方式进行管理。然而，高级的程序包管理是现代版本最重要的贡献之一。今天的 Linux 版本包括了一个程序包跟踪数据库，它能轻松地安装、更新、移动程序包。

在 Linux 的早期阶段，SLS 版本是第一个 Linux 程序包的集合，并且是公认的一个完全版本。虽然 SLS 版本可作为一个简单的实体进行安装，但是它缺少现在 Linux 版本所具有的程序包管理工具。**Slackware** 版本在整体质量上表现出巨大的进步，尽管程序包的管理仍然很差；它是在 Linux 团体中安装最广泛的版本之一。

自从有了 Slackware 版本，许许多多商业的和非商业的 Linux 版本变得随手可得。**Red Hat** 和 **Debian** 两种版本分别来自一家商业的 Linux 支持公司和自由软件 Linux 团体。其他的 Linux 商业支持版本包括 **Caldera**、**Craftworks** 和 **WorkGroup Solutions**。Linux 的德文版导致产生另外许多种专用的德文版本，包括了来自 **SuSE** 与 **Unifix** 的版本。现在有众多的 Linux 流通版本，在此无法一一列出。Linux 各种版本之间都兼容。RPM 包文件格式被大多数的版本所使用，或至少被支持。采用这种格式的商业应用软件可被安装运行于任何支

持 RPM 文件的版本上。

20.1.4 Linux 许可

Linux 内核必须获得 GNU 通用公共许可(GPL)后发布,其条款由自由软件基金会制定。Linux 并不是公共域的软件:**公共域**(public domain)意味着作者放弃软件的版权,但是 Linux 代码的版权仍为代码的各位作者所有。Linux 是免费软件,也就是说人们可以随意拷贝、修改和使用它,并且能毫无约束分发他们自己的拷贝。

Linux 许可证条款的主营含义在于任何人可以使用 Linux,或者创建自己的 Linux 派生系统(合法的使用),但不能拥有这个派生产品的所有权。在 GPL 下发行的软件不得以二进制产品形式进行发布。如果你发布的软件只要包括任何 GPL 的部分,根据 GPL,你必须随着二进制产品的发行也提供源代码。(这个限制并不是禁止制造或者销售二进制软件产品,任何获得二进制产品的人都有机会获得源代码,代价是合理的版本费用。)

20.2 设计原理

总体设计上,Linux 类似于任何其他的传统设计,是非微内核的 UNIX 实现。它是一个多用户、多任务的系统,拥有一整套与 UNIX 兼容的工具。Linux 的文件系统追随传统的 UNIX 语义学,而且完整地实现了标准的 UNIX 网络模型。Linux 的内部设计细节深受操作系统发展历史的影响。

虽然 Linux 运行于多种平台,但它唯独是从个人计算机的体系结构上发展过来的。大量的早期开发是由个人爱好者实现的,而不是靠组织良好的开发和研究团队,所以从一开始 Linux 就试图从有限的资源中开发尽可能多的功能。如今,Linux 欢快地运行在数以百兆字节内存和以数十亿字节计的磁盘空间的多处理计算机上,但它依然能运行在 4 MB 以下的 RAM 上。

随着个人计算机的日益强大,内存和硬盘的日趋便宜,最初低要求的 Linux 内核开始逐渐扩展实现更多的 UNIX 功能。速度与效率仍然是重要的设计目标,但是当前的很多基于 Linux 的工作却集中在第三个主要的设计目标上:标准化。由于 UNIX 的不同实现而付出的代价之一便是,为一个版本写的源代码未必能在另一个版本上正确编译或运行。甚至当相同的系统调用出现在两个不同的 UNIX 系统上时,其实现过程也未必完全一样。POSIX 标准包含一套规范,详细说明了操作系统执行的不同方面。还有一些关于一般操作系统和延伸部分如进程线程与实时操作的 POSIX 文档。Linux 根据对应的 POSIX 文档进行设计,至少两种 Linux 版本已取得官方的 POSIX 认证。

Linux 给程序员和用户提供了标准的接口,所以任何熟悉 UNIX 的人都不会对此感到陌生。在此不再细说 Linux 环境下的这些接口。关于程序员接口(A.3 部分)和用户界面

(A.4 部分)部分也将很好地应用于 UNIX。然而由于疏忽,Linux 程序设计接口追随 SVR4 版的 UNIX 语义,而不跟随 BSD 的风格。一个单独的程序库集合也能在 BSD 语义下有效运行,尽管这两种语义是极为不同的。

在 UNIX 世界还存在许多其他的标准,但是 Linux 对这些标准的完全认证却进展缓慢,因为它们通常是收费的,并且认证一个操作系统是否遵照大多数标准的相关费用是庞大的。然而提供一个广泛的软件基础对任何操作系统来说都是十分重要的,因此执行标准是 Linux 开发的一个主要目标,即使其尚未得到正式的认证。除了基本的 POSIX 标准外,当前 Linux 还支持 POSIX 线程扩展和用于实时进程控制的 POSIX 扩展子集。

20.2.1 Linux 系统的组件

Linux 系统由三种主要的代码部分组成,符合大多数传统的 UNIX 实现:

1. **内核**:内核负责维护操作系统的重要抽象,包括虚拟内存和进程等。

2. **系统库**:系统程序库定义了一系列函数,由此应用程序能够和内核进行交互,并且实现大多数系统功能而无需使用拥有完全特权的内核代码。

3. **系统实用程序**:系统实用程序是指那些独立的,特别是管理任务的程序。有些系统使用程序只在系统初始化和配置时调用一次;其他诸如 UNIX 术语中的后台程序(daemons),将永久性运行,处理任务如响应即将到来的网络连接,接受来自终端的登录请求,或刷新日志文件,等等。

图 20.1 说明了组成一个完整的 Linux 系统的各种组件。这里最主要的区别在于内核与其他非内核的所有部分。所有的内核代码都在处理器的特许模式下运行,并能访问计算机的所有物理资源。Linux 称这个特权模式为**内核模式**(kernel mode),相当于 2.5.1 小节所描述的监控模式。在 Linux 环境下,任何用户模式的代码都未被加入到内核中。而任何操作系统支持的、无需以内核模式运行的代码都放入系统库内。

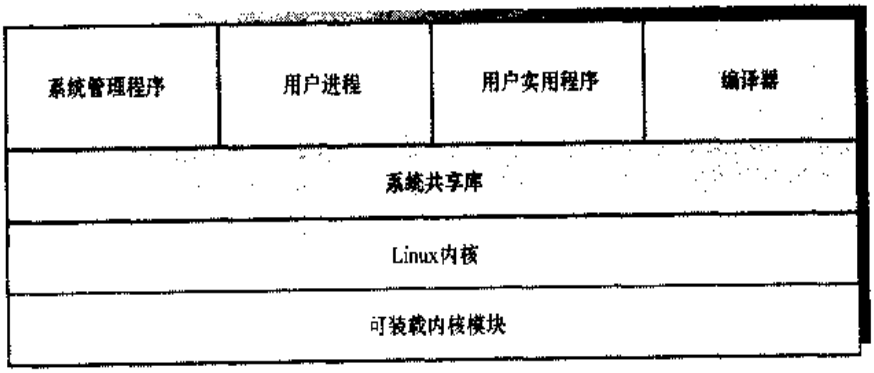


图 20.1 Linux 系统的组件

虽然各种现代操作系统在内核中都采用了消息传递体系结构,但是 Linux 还是保留了 UNIX 的历史模型:内核被创建成单一的、整体的二进制形式。最主要的原因是为了提高性

能;因为所有的内核代码和数据结构被保存在一个单一的地址空间,当一个进程调用一个操作系统的功能或者产生硬件中断时就没有必要进行前后关联转换。不但是核心调度和虚拟内存代码占据这个地址空间;而且所有的内核,包括所有的设备驱动程序、文件系统和网络代码都在这个地址空间内。

不要仅仅因为所有的内核共享同一个区域就认为没有模块作用域。通过同样的方式,用户应用程序可在运行时调入共享库,以引入所需要的代码,所以 Linux 内核能在运行时动态装载或卸载模块。内核不必预先知道哪个模块可能会被调入——模块是完全独立可装入的组件。

Linux 内核组成了 Linux 操作系统的核心部分。它提供了运行进程必需的所有功能,并且它还提供进行判断和保护访问硬件资源的系统服务。内核实现了所有要求与一个操作系统相匹配的功能。然而,从它自己的角度看,Linux 内核提供的操作系统却一点也不像 UNIX 系统。它失去了许多 UNIX 的外部特征,并且也没有 UNIX 应用程序所期望出现的必要格式。内核并不直接维护对应用程序透明的系统接口。事实上,应用程序调用的是系统库,接着系统库在必要时也调用操作系统服务。

系统库提供了许多类型的功能。在其最简单层,它们允许应用程序提出内核系统服务请求。实现系统调用涉及到从非特权用户模式到特权内核模式的控制转移。这种转移的细节在各个体系结构之间是不同的。系统库收集系统调用的参数,如果必要的话,以特殊形式编排这些参数来完成系统调用。

程序库还应该提供更复杂的基础系统调用的形式。譬如,C 语言的缓冲文件处理函数全都在系统库中执行,提供比基本的内核系统调用更高级的文件 I/O 控制。这些库也提供与系统调用毫无关系的程序,例如排序算法、数学函数以及字符串处理程序。所有支持 UNIX 和 POSIX 应用程序运行的必需功能都在系统库里被执行。

Linux 系统包含了种类广泛的用户模式的程序——既有系统应用程序又有用户应用程序。系统应用程序包括所有使系统初始化的必要程序,如配置网络设备或者调入内核模块。持续运行服务器程序也算是系统应用程序,这些程序主要处理用户注册请求、输入网络连接及打印机队列。

并不是所有的标准应用程序都为关键的系统管理功能服务。UNIX 用户环境包含了大量处理简单日常任务的标准应用程序,譬如列出目录,删除和移动文件,或者是显示文件内容。更为复杂的应用程序可进行文本处理功能,如对原文数据进行分类或对输入文本执行模式搜索。这些应用程序形成了一个用户在任何 UNIX 操作系统里都能看到的标准工具集;虽然这些应用程序不能执行任何操作系统功能,但它们却是 Linux 系统很重要的一个组成部分。

20.3 内核模块

Linux 内核能够根据需要装载或卸载任意内核代码段。这些可装载的内核模块运行于特权内核模式，所以这些内核模块能够完全访问其所运行的计算机的硬件。从理论上讲，对内核模块的权限并没有限制；典型的例子是，模块可能执行一个设备驱动程序，或一个文件系统，或是一个网络协议。

有几方面的原因可以说明内核模块用起来很方便。Linux 的源代码是开放的，因此任何想编写内核代码的人都能修改内核并且对之进行编译，重启后便能装入那些新功能；然而，当你正在开发一个新的驱动程序时，你也不得不反复进行重新编译、重新连接、重新装入整个内核这样繁琐的工作。如果使用内核模块，你就无需做一个新的内核来测试新的驱动程序——因为驱动程序在其自身基础上编译并且能被装载到正运行的内核中。当然，一旦一个新的驱动程序被写出来，就该将它作为一个模块发布，这样其他的用户无需重建内核也能从中受益。

接下来的这个观点具有另外的含义。因为 Linux 内核在 GPL 许可下，当加入了具有所有权的组件时，它就不能发布，除非那些新组件也遵守 GPL 而发布，并且它们的源代码根据需要就可得到。内核模块接口允许第三方组织进行编写和发布，但按照其条款，设备驱动或者文件系统在 GPL 下是不能发布的。

内核模块允许 Linux 系统由一个很小的标准内核构成，而不必包括额外的驱动程序。任何用户需要的设备驱动程序可以在系统启动时被显式地装入，或者系统按需自动装入或卸载。例如，当加载 CD 时就必须装入 CD-ROM 驱动，当 CD 从文件系统中移去后 CD-ROM 驱动程序就从内存中卸载下来。

Linux 环境下模块支持体现在以下三个方面：

1. **模块管理**(module management)允许该模块被导入内存并能与内核的其他模块进行通信。

2. **驱动程序注册**(driver registration)允许该模块告诉其他模块一个新的驱动程序已经可以使用。

3. **冲突—解决机制**(conflict-resolution mechanism)允许不同的设备驱动器保留硬件资源，并且保护那些被其他驱动器突发使用的资源。

20.3.1 模块管理

装入一个模块不仅仅只是将二进制的内容导入内核存储空间中。系统必须将模块引用到的内核调用的内核符号或入口点更新为内核寻址空间的正确位置。Linux 将模块的导入工作分为两个部分来处理：管理内核存储空间中的模块代码；处理被允许引用的符号。

Linux内核中有一个内部符号表。在后期的编译过程中标识符表并不包含内核所定义的所有符号：符号必须被内核显式地输出。被输出的符号组成了一个定义明确的接口，通过这个接口，模块能与内核进行交互。

虽然内核函数输出的符号需要程序员进行明确的请求，但是将这些符号引入到模块中却不用其他特别的工作。编写模块的程序员只需使用标准的C语言外部链接：在最终生成二进制文件时，编译器只是简单地将模块中使用到的外部符号设置为未解析。所有需要被解析的符号都能在内核符号表中找到，在当前运行的内核中的符号地址被替换成模块代码。至此，模块便被装入内核中。如果系统查询内核符号表时无法解析模块中的符号引用，那么该模块就会被拒绝。

装载模块分两个阶段执行。首先，模块装载程序为模块向内核申请预定一个连续的虚拟内核存储空间。内核返回分配好的内存地址，装入程序就能利用地址重定位模块代码得到正确的装载地址，随后的系统调用把模块和新模块需要输出的任何标识符表传递给内核。现在模块逐字地复制到先前分配的地址中，内核符号表和可能被那些尚未转入的模块所使用的新符号都被更新完毕。

最后的模块管理组件是模块请求程序。内核定义了一个能与模块管理程序连接的通信接口。这种连接建立后，当某个进程向尚未安装的设备驱动程序、文件系统或网络服务程序请求时，内核将通知管理程序并让管理程序装入所需的服务程序。一旦模块被装入内核，原先的服务请求也就完成了。管理程序进程经常询问内核动态装入的模块是否仍在使用，并将不再使用的模块卸载。

20.3.2 驱动程序注册

一旦模块装好以后，除非内核的其他部分知道它能够提供什么新功能，否则它只不过是一个孤立的内存块。内核将其所知的所有驱动程序都保留在动态表中，同时提供了一套允许任何时候从这些表中增加或删除驱动的程序。当装入模块之后，内核确保它能调用所需模块的启动程序，并且能在模块卸载之前调用模块清除程序。这些程序负责注册模块的功能。

一个模块能注册多种类型的驱动程序，一个单一模块可以注册任一或所有这一类型的驱动程序，也可以注册一个以上的驱动程序。例如，设备驱动程序可能同时需要注册两个单独访问设备的机制。注册表包含以下几项：

- **设备驱动程序**：这些驱动程序包括字符设备（诸如打印机、终端、鼠标）、区块设备（所有磁盘驱动器）及网络接口设备。
- **文件系统**：文件系统可能只是运行Linux的虚拟文件系统调用程序。它可能实现文件在磁盘中的存储格式，但它也可能是一个网络文件系统，例如NFS文件系统，或者内容根据需要经常更新的虚拟文件系统，例如Linux的PROC文件系统。
- **网络协议**：模块可能实现诸如IPX这样完整的网络协议，或者可能只是简单实现网

络防火墙的一套新的分组过滤规则。

• **二进制格式**：这种格式描述了识别和装载新类型可执行文件的方法。

另外，模块可以在 sysctl 中注册一组新入口和/proc 表，从而允许动态配置此模块（20.7.3 小节）。

20.3.3 冲突解决方案

商业版 UNIX 通常出售给硬件供应商在他们自己的硬件上运行。单一供应商解决方案的一个好处在于软件供应商非常清楚硬件可能采用什么样的配置。另一方面，IBM PC 有大量的硬件配置，并附有大量相应的设备驱动程序，例如网卡、SCSI 控制器和显示适配器。当支持模块设备驱动时，管理硬件配置的问题变得越来越严重，因为现在这些设备的变化越来越快。

Linux 系统规定了一种重要的冲突解决机制，这有助于仲裁对某一硬件资源的访问。其目的在于：

• 防止模块在访问硬件资源时发生冲突。

• 防止现存的设备驱动器互相干扰，即**自动检测**(autoprobe)设备配置的设备驱动器问题。

• 解决由于多个设备试图访问同一硬件时所产生的冲突问题；例如，并行打印机驱动程序和并行 IP(PLIP)驱动程序可能同时访问打印机端口。

最后，内核维护分配的硬件资源表。个人计算机只有有限的 I/O 端口（硬件 I/O 地址空间中的地址）、信号中断线和 DMA 通道；当任一设备驱动想要访问这些资源时，必须先在内核数据库中预定这些资源。这样的要求附带地允许系统管理员给定任一端口便能决定哪些资源被那个驱动程序申请了。

模块利用这种机制可以提前预定它所希望使用的任一硬件资源，如果资源不存在，或者在使用当中，预定将被取消，然后由模块决定该如何处理。它可能无法进行初始化，并且由于不能继续进行而被要求卸载；或者可能会继续进行，选择其他硬件资源。

20.4 进 程 管 理

在操作系统中，进程的一项基本内容就是用户请求服务。为了与其他 UNIX 系统相互兼容，Linux 必须使用一种类似于其他 UNIX 版本的进程模式，然而 Linux 在一些关键部分与 UNIX 有所不同。这一部分回顾 A.3.2 部分的传统 UNIX 进程模式，并介绍 Linux 自己的线程模式。

20.4.1 Fork/Exec 进程模型

UNIX 进程管理的基本原理是把创建进程与运行一个新程序这两个截然不同的操作分

开。一个新进程由系统调用 fork 产生，而新的一个程序通过调用 execve 来运行。这是两个完全不同的函数。由 fork 创建的新进程不需要运行新程序——新创建的子进程仅仅是继续执行与父进程相同的程序。同样，运行一个新程序不需要创建新进程；任何进程可以随时调用 execve。当前运行的程序立刻被中断，新的程序作为进程内容开始执行。

模块具有极其简单的优点。在运行此程序的系统调用过程中，不需要详细说明新程序环境中的每一个细节，新程序只运行于他们所存在的环境。如果父进程想要修改新程序运行的环境，它可以派生一个新进程，然后在子进程中继续运行原先的程序，在最终新程序运行前使用系统调用来修改子进程。

在 UNIX 环境下，进程包含了所有系统跟踪程序对应的关联信息。在 Linux 环境下，将这些内容分为几个特定的部分。在一般情况下，进程属性可分为三组：进程特征、环境和关联。

1. 进程特征

进程特征主要由以下几项组成：

• **进程** ID(PID)：每个进程都有惟一的标识符。当某一应用程序通知、修改或等待其他进程时，操作系统便使用 PID 来识别进程。另外，标识符可设置该进程与进程组以及登录会话的关系(举个典型的例子，单一的用户命令可派生出一系列呈树状的进程)。

• **认证**(credential)：每个进程必须拥有一个相关的用户 ID 和一个或多个用户组 ID(用户组在 11.6.2 小节中进行过讨论)，这些 ID 决定进程访问系统资源和文件的权限。

• **特性**(personality)：进程的特性不是在传统的 UNIX 系统里就可以找到的，但在 Linux 系统里每个进程都拥有相应的特征标识符，它能对某特定的语义系统调用程序做稍稍的修改。它的主要功能是被仿真库用来要求系统调用程序需与某一 UNIX 的风格相匹配。

大多数的进程标识符受到进程自身控制的限制。如果进程要重启一个新的组或会话，进程的组和会话标识符就会相应地发生改变。在适当的安全检查情况下，进程自身的认证也可改变。然而直到进程结束，它的主 ID 也是不能更改的，并且是惟一的。

2. 进程环境

进程环境从父进程继承而来，由两个非终结向量组成：即参数向量和环境向量。**参数向量**(argument vector)只是简单罗列用于调用运行程序的命令行参数，一般都以程序名开始。环境向量(environment vector)则是以表的形式罗列"NAME=VALUE"对，它把被命名的环境变量和任意原文的值联系起来。环境向量并不占据内核存储空间，而是作为进程栈顶部的第一项数据被保存于进程自己的用户模式地址空间。

当一个进程创建时，其参数向量与环境向量是无法选择的：新的子进程继承了其父进程的环境。然而，当一个新的程序被调用时则会建立一个新的环境。调用 execve 时，进程必须为新的程序提供环境。内核把这些环境变量传递给下一个程序，该程序对进程当前环境进行替换。否则，内核将无视这些环境和命令行变量——对它们的描述将完整地留给用户模式的程序库和应用程序。

进程间传递环境变量和子进程对这些变量的继承，都为用户模式系统软件组件传递信息提供了十分灵活的途径。各个重要的环境变量与系统软件中某个部分相对应。例如，TERM 变量用来命名连接用户登录会话终端的类型。许多程序利用这个变量来决定如何执行显示用户操作。例如，移动光标或者是滚动文本区域。具有多语言支持的程序利用 LANG 变量来决定用哪种语言来显示其系统信息。

环境变量机制不是为整个系统，而是为每一个进程裁剪出一个略有不同的操作系统环境。用户可选择他们自己的语言或编辑器。

3. 进程关联

进程特征和环境属性通常随着进程的创建而建立，在进程退出前都不会改变。如果需要的话，进程会有选择地改变它自身的某方面，或者改变它的环境。从另一方面来说，进程关联是程序运行在某一时刻的一种状态，是经常改变的。

• **调度关联**：进程关联中最重要的部分是它的调度关联：进程挂起和重启的信息。这种信息包括了所有进程寄存器的备份。浮点寄存器被单独保存并且只在需要用时才恢复，所以不引用浮点算法的进程就不会保存这些寄存器。进程关联同时也包括了关于调度优先级和等待被传给进程信号的信息。调度关联的关键部分是进程的内核栈：内核存储器的一个独立的区域，由内核模式代码专用。进程运行时可能用到的系统调用和中断都将使用此栈。

• **记账**：内核保留了最近被进程所使用的资源的相关信息，并且记录了进程在其生存期内所使用全部资源的情况。

• **文件表**：文件表是一个指向内核文件结构的指针数组。当调用文件 I/O 时，进程根据索引访问此表。

• **文件系统关联**：文件表列出了正在打开的文件，而文件系统关联适是用于申请打开新文件。用于搜索新文件的那些当前根目录和缺省目录就被保存在这里。

• **信号处理程序表**：UNIX 系统能将异步信号传给进程以响应不同的外部事件。信号处理程序表在进程地址空间中定义了当特殊信号到达时所要调用的程序。

• **虚拟内存关联**：虚拟内存关联描述了进程私有地址空间的全部内容，将在 20.6 节讨论。

20.4.2 进程与线程

大多数现代操作系统都同时支持进程与线程。虽然两者之间确切的不同之处经常在实现时发生改变，但是还是要做一下如下区分：*进程*表现为单一程序的执行过程，而*线程*表现为在运行某一程序的某一进程内，独立地、并发地执行关联。

任何两个单独的进程都拥有它们自己的独立地址空间，尽管有时候它们利用共享内存来共用虚拟内存中的某些(但不是全部)内容。相反，同一进程的两个线程将共享*相同的*地址空间，而不是*相似的*地址空间。因为线程都在同一地址空间运行，所以一旦某一线程对虚拟空间做任何改变，其他的线程就会马上发现。

线程可通过几种方法来实现。在操作系统内核里,线程可以作为进程的对象来实现,或者它可以作为一个完全独立的实体。它根本不能在内核中实现——线程只有在内核提供的时钟中断帮助下,通过应用程序或者程序库才能真正实现。

Linux 内核只是简单处理进程与线程之间的差别:它准确利用了二者之间在内部表示方式上的共同之处。线程只不过是一个能与其父进程共享相同地址空间的新进程。进程和线程的区别在于通过系统调用 clone 创建线程。由 fork 创建的新进程拥有它自己全新的进程关联环境,而 clone 创建的新进程则拥有它自己的标识符但允许共享其父进程的数据结构。

因为在主进程数据结构中 Linux 不包含进程的整个关联环境,所以这种差别完全可以实现。然而在独立的子关联中,它包含关联环境。进程的文件系统关联、文件描述表、信号处理表及虚拟内存关联被保存在独立的数据结构中。进程数据结构只是简单地包含了指向这些结构的指针,因此通过指向合适的同一子关联,许多进程可以轻而易举地共享这些子关联的任何内容。

clone 系统调用通过参数得知哪个子关联需要复制,哪个需要共享,什么时候要创建新的进程。新的进程赋给新的标识符和新的调度关联。然而,根据传递的参数,也可能创建新的子关联数据结构,并初始化为其父进程的一个拷贝,或者建立新的进程,与父进程同时使用同一关联数据结构。fork 系统调用只不过是拷贝所有子关联,无任何共享的 clone 特例。利用 clone 给出的一种应用程序,能非常细致精确地控制两个线程之间所要共享的内容。

POSIX 工作组已经定义了程序接口,在 POSIX.1c 标准中有详细说明,它允许应用程序运行多线程。Linux 系统库支持两种不同的机制,它们以不同方式实现同一标准。应用程序可以选择使用 Linux 的基于用户模式的线程包或者基于内核的线程包。用户模式线程库避免了线程间交互时内核调度和内核系统调用的消耗,但它仍局限于所有线程同时运行在单一进程的情况。内核支持线程库利用 clone 系统调用来实现相同的程序接口,但是因为创建了多个调度关联,它能让应用程序将线程运行在多个处理器上(如果程序运行在多处理器系统上),也允许多线程同步运行内核系统调用。

20.5 调 度

调度是操作系统通过分配 CPU 的时间来区分任务的一项工作。通常,认为调度就是运行或中断进程,但是对于 Linux 来说调度还有另一方面的任务,那就是,运行多种内核任务。内核任务包含运行进程所要求的任务和代表设备驱动器在内部执行的任务。

20.5.1 内核同步

内核调度的方式与执行进程调度的方式是完全不同的。内核模式执行的程序发出请求

有两种方式。正在运行的程序都可以请求操作系统服务，无论是显式的或者是隐式的（例如，页面故障的发生）系统调用。另外，设备驱动程序能产生硬件中断，从而导致 CPU 启动执行一个内核定义的操作来处理这个中断。

摆在内核面前的问题是这些任务可能都试图访问同一个内部数据结构。如果一个内核任务在访问某一数据结构时要执行一个中断服务程序，在防止数据损坏的情况下，该中断服务程序就不能访问或者修改同样的数据。这与临界区的想法相关：代码可访问共享数据但不允许并行访问。

结果，内核同步涉及的内容远远超出了进程调度本身。需要有一个结构允许内核的临界区能在不被其他临界区中断的情况下运行。

对于这个问题，Linux 解决方案的第一部分是使得普通内核代码非抢占。通常情况下，内核收到时钟中断后，它就调用进程调度程序，这样它就能挂起当前运行的进程，并继续运行另一进程（任何 UNIX 系统都具备的时间分配）。但是，若进程在执行内核系统的服务程序时收到时钟中断，则重新调度就不能立即执行。事实上，内核的 need_resched 标志就是用来告诉内核在系统调用结束后运行调度程序并且控制进程返回用户模式。

内核的某段代码一旦开始运行，那在下列行为开始前它是惟一正在运行的内核代码：

- 中断
- 页错误
- 调度程序本身的内核代码调用

如果它们本身包含了临界区，中断就有问题，时钟中断不会直接导致进程的重新调度；它们只是请求稍后执行重新调度，因此任何输入的中断都不会影响非中断内核代码的执行顺序。一旦完成了中断服务程序，执行程序将返回到中断发生时正在运行的同一内核代码那里。

页错误是一个潜在的问题，如果内核程序试图读或写用户内存，将会导致页错误，进而需要磁盘 I/O 的完成，这时正在运行的进程将被挂起，直到 I/O 完成为止。与此相似的是，如果系统调用服务程序调用了调度程序，而此调度程序是处于内核模式，那么无论是对调度代码显式地直接调用，还是隐式地通过调用一个函数来等待 I/O 系统完成，进程都会被挂起并且进行重新调度。当进程能够重新运行时，它将以内核模式继续执行，在调用调度程序之后继续执行指令。

内核代码假定决不会被其他进程抢占，也不需要特别对临界区进行保护。惟一的要求是临界区不能包含用户内存参数，或者是等待 I/O 系统的完成。

Linux 使用的第二种保护方法是对出现于中断服务器程序中的临界区的应用。其基本的工具是处理器的中断控制硬件。通过在临界区禁止出现中断的方法，内核确保在没有并行访问共享数据结构的风险下得以继续进程。

对于禁止中断的行为也要付出代价。在大多数硬件体系中，中断的有效和失效指令都

很昂贵。而且，只要中断保持失效，所有的 I/O 都被挂起，任何等待服务的设备驱动程序将不得不等候中断恢复使用，因此这将降低性能。Linux 内核使用了一种同步体系结构，在中断有效的情况下，允许临界区在他们的整个持续期间运行。这种功能在网络代码中特别有用：在网络设备驱动器中出现的中断标志着一个完整分组的到达，且在中断服务程序中将执行大量分解、路由、转发分组的代码。

Linux 为实现这个体系结构，把中断服务程序一分为二：上半部（top-half）与下半部（bottom-half）。**上半部**是一种普通的中断服务程序，运行时递归中断被禁止，中断的高级优先权可以中断程序。但是同样的或较低的优先级的中断就没有这种功能。服务程序的**下半部**在所有中断有效时，通过一个微型的调度程序运行，这样就保证了下半部的中断服务程序不会被它们自己中断。无论什么时候中断服务程序存在，下半部的调度程序就会被自动调用。

这样的区分意味着内核在不被中断干扰的前提下，内核就能完成任何与中断相关的复杂的处理过程。如果另外的中断出现时下半部正在执行中，那么中断就会向下半部发出执行请求，但是执行要往后推迟直到当前运行结束。每一个下半部的执行程序都有可能被上半部（的执行请求）所中断，但是不会被相似的下半部所中断。

上半部、下半部体系是通过“在执行常规的、前台内核代码时使被选用的下半部失效”这样一种机制来完成的。内核通过这个系统可以轻易地为临界区编码。中断处理程序也可为它们的临界区在下半部进行编码，当前台内核想进入一个临界区，它能禁止任何相关下半部，以防其他的临界区来中断它。在临界区最后，内核恢复下半部并运行下半部的任务，而这些任务在临界区期间被上半部的中断服务程序要求排队等候。

图 20.2 概述了内核内部中断保护的各个层。每一层都可能被运行于上一层的代码所中断，但是不会被运行在同一层或下一层的代码所中断；用户模式的代码例外，当一个分时调度中断发生时，用户进程总是被另外的进程所抢占。

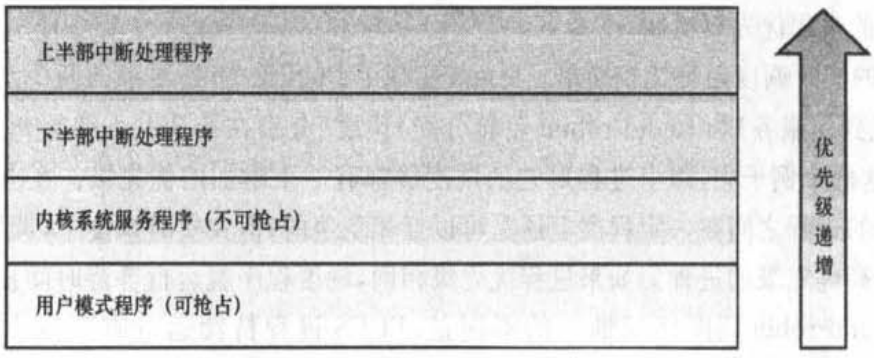


图 20.2　中断保护层

20.5.2 进程调度

一旦内核到达重新调度点时——如发生重新调度中断，或者一个运行的内核进程阻塞后正在等待某个唤醒信号——它必须决定下一步运行什么进程。Linux 有两个独立的进程调度算法。一个是多进程中的公平抢占调度的分时算法；另一个是为实时任务设计的绝对优先权比公平更为重要的算法。

进程特性是一个调度类，它定义了应该采用哪种算法处理进程。POSIX 附加标准定义了 Linux 使用的实时计算调度类(即 POSIX.4，也就是现在的 POSIX.1b)。

对于分时进程，Linux 使用的一种基于信用量(credit-based)的优先级算法，每个进程处理一定的调度信用量(credit)；当必须选择运行新任务时，就会选择信用量最大的进程。时钟中断发生时，当前正在运行的进程就会减少一个信用量。当它的信用量变成零时，系统将此进程悬挂并且选择其他的进程。

如果可运行的进程的信用量(credit)都为零，Linux 将重新设定(recredit)，增加系统中每一个进程的信用量，而不仅仅是针对可运行的进程，(增加的方法)根据下面的公式：

$$\text{信用量} = \frac{\text{信用量}}{2} + \text{优先级}$$

这个算法取决于两个因素——进程的历史和优先级。进程包含的二分之一的信用量是因为先前重新设置信用量的操作在采用算法之后被保留，它保留了进程当前操作的记录。经常处于运行状态的进程会很快耗尽其信用量，但经常被挂起的进程将通过多次的信用量重设和累加，最后以较高的信用量结束。这种信用量系统自动给交互式进程或者是 I/O 进程以高优先级，因为这些进程的快速响应时间显得非常重要。

在计算新的信用量时，进程优先级的使用是有一定规则的。后台批处理应该赋予较低的优先级；它们所得到的信用量也比用户交互式任务少，因此其所分配到的 CPU 时间比例也比有较高优先权的类似工作小。Linux 系统就是利用这样一个优先级系统来实现标准 UNIX 完美的进程优先权机制。

Linux 的实时调度还是比较简单。Linux 实现了 POSIX.1b 所要求的两个实时调度类，即 FCFS(先到先服务)和 round-robin(轮转分配)调度(分别在 6.3.1 小节与 6.3.4 小节中介绍)。在这两个例子中，每个进程对它的调度级都有一个附加的优先级。在分时调度中，不同优先权的进程之间在一定程度上还是可以互相竞争的；但在实时调度中，调度程序始终运行具有最高优先级的进程。如果进程优先级相同，调度程序就运行等待时间最长的进程。FCFS 与 round-robin 调度之间惟一的不同是，FCFS 进程持续运行直到退出与堵塞；而 round-robin 进程运行一定时间后被抢占，并且被放置在调度等待队列的最后，因此在 round-robin 下，具有同等优先权的进程之间自动分时。

Linux 的实时调度是软实时(而不是硬实时)。调度程序严格保证实时进程之间相对的优先级。实时进程一旦变为可以运行时，内核却不能保证多久以后才能调度它。记住

Linux 内核代码从来不会被用户模式的代码所抢占。如果发生中断，势必要唤醒一个实时的进程，而内核却已经执行了另一个进程的系统调用，那么该实时进程就必须等待当前运行的系统调用结束或者堵塞。

20.5.3 对称多处理技术

Linux 2.0 内核是第一个支持**对称多处理机**(symmetric multiprocessor, SMP)硬件的内核。进程与线程可以在各个处理器上并行运行。然而为保护内核的非抢占式同步要求，必须在内核中对 SMP 的实现加以限制：同一时间只允许一个处理器执行内核模式的代码。SMP 使用了一个简单的自旋锁就实现了上述限制。自旋锁不会引起计算限制任务之类的问题，但是频繁使用内核的任务将严重地成为系统瓶颈。

Linux 2.2 内核通过把一个内核自旋锁分为多个锁的方法使 SMP 的实现更具有可升级性，每个锁保护内核数据结构的小子集不被其他进程访问。目前，信号处理、中断和一些 I/O 程序使用多重锁从而允许多处理器同时运行内核代码。内核 2.3(Linux 2.4)似乎完全摈弃了单一内核自旋锁。

20.6 内存管理

Linux 中的内存管理分为两部分。第一部分处理分配和释放物理内存：分页、分页组和小内存块。第二部分处理虚拟内存，就是内存被映射到正在运行的进程的地址空间上。

先描述这两部分，然后研究新程序的可加载部分导入进程的虚拟内存中(对应于 exec 系统调用)的机制。

20.6.1 物理内存管理

Linux 内核中基本的物理内存管理器是**页面分配程序**(page allocator)。此分配程序负责分配和释放所有的物理页面，并且它能根据要求分配连续的物理页面，分配程序运用**伙伴堆算法**(buddy-heap algorithm)跟踪可用的物理页面，伙伴堆分配程序把可分配内存的毗邻单元配成对(名字由此而来)，每个可分配内存区域都有一个与之毗邻的伙伴，无论何时这两个已分配的内存区被一起释放，它们将合并成更大的区域。这个更大的区域也有伙伴，它们也能合并成一个区域。如现有的小空间不能满足内存空间请求的话，那么系统将相对大一点的内存单元分为两个部分去满足这个请求。各个链表用来记录大小合适的空闲内存区域。Linux 系统中，这种机制下允许分配的最小尺寸是一个物理页面。图 20.3 就是伙伴堆分配的例子，一个 4 KB 的区域等待分配，但是可利用的最小区域是 16 KB。这 16 KB 区域递归分解，直到满足要分配的大小。

最终，Linux 内核的所有内存分配以静态或动态的方式存在。静态分配指的是驱动程

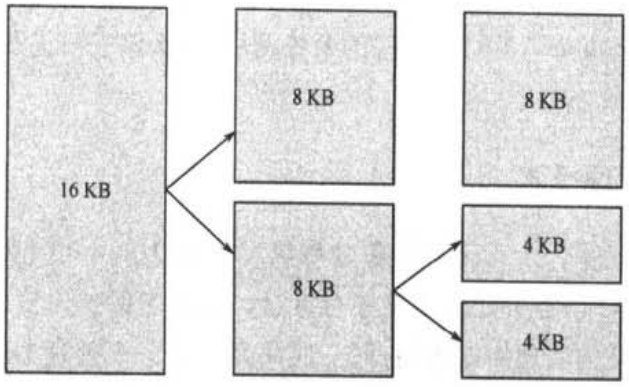


图 20.3 伙伴堆中内存的分割

序在系统启动时获得的内存连续空间，动态分配指的是通过页面分配程序获得的内存空间。然而内核函数并不是一定要使用基本的页面分配程序来获得内存空间。有些特殊的内存管理子系统优先使用基本的页面分配器来管理自己的内存池。20.6.2 小节将描述其中最重要的部分——虚拟内存系统；kmalloc 变长分配程序；内核的两个常驻数据高速缓存器，缓冲高速缓存器，页面高速缓存器。

Linux 操作系统的许多组件都需要分配完整的页面，但需要的内存块通常比较小。内核为任意大小的请求提供附加的分配器，所请求的大小预先不知道，可能只有几个字节，而不是整个页面。与 C 语言的 malloc 函数相似，**kmalloc** 服务按需分配完整的页面，然后把它们分成较小的几个部分。内核保留了 kmalloc 服务使用的整套页面表，表上所有的页面都被分割成大小特定的几块。分配内存包括设计出适当的列表，取出表上第一片可利用的，或者是分配一个新的页面并分割开来。

页面和 kmalloc 分配程序由中断确保安全性。需要申请内存空间的函数向分配函数发出了请求优先级。中断程序利用原子优先级确保请求被响应或者请求由于没有内存而立即失败。相反地，普通用户进程请求内存分配则寻找释放现有的内存，除非存在可用的内存，否则进程将在此延迟。DMA 内存请求时可能特别使用分配优先级，因为在诸如个人计算机的体系结构中，并非所有的物理内存页面都支持 DMA 的某些请求。

kmalloc 系统要求的内存区域将被永远分配出去直到被显式地释放。内存不足时，**kmalloc** 系统无法重定位或者收回这些区域。

有自己物理页面管理的其他三个子系统紧密相关。它们是内存缓高速缓存(buffer cache)、页面高速缓存(page cache)和虚拟内存系统。**内存缓冲高速缓存**(buffer cache)是内核的主要 cache，它用于面向块的设备(诸如磁盘驱动器)；同时也是实现与这些设备进行 I/O 操作的主要机制。Linux 本地基于磁盘文件系统和 NFS 网络文件系统都使用页面 cache。页面 cache 缓冲了所有文件内容的页面，甚至是块设备；它也能缓冲网络数据。*虚拟内存系统*管理着每个进程虚拟地址空间的内容。

这三个系统相互紧密联系。将一个页面的数据读入页面，cache需要临时通过内存缓冲cache，如果进程把文件映射到它的地址空间，那么页面cache中的页面也可以被映射到虚拟内存系统上。内核对物理内存的页面都有引用计数，这样被两个或者两个以上的子系统所共享的页面当不再使用时便被释放。

20.6.2 虚拟内存

Linux虚拟内存系统负责保持地址空间对于每个进程都是可视的。它根据需要创建虚拟内存的页面，并管理从磁盘装入页面，或者是按照要求将页面交换到磁盘上。在Linux系统下，虚拟内存管理程序对进程地址空间有两种不同的观点：作为一组独立的区域，或作为一组页面。

地址空间的第一种是逻辑视图，它描述了虚拟内存系统接收到关于地址空间布局的指令。在这种视图中，地址空间由一组不重叠的区域组成，每个区域都是连续的、页面对齐的地址空间子集。每个区域内部使用 **vm_area_struct** 结构来定义区域的属性，包括了进程的读、写和执行许可以及任何与区域相关的文件信息。每一个地址空间的区域都被连接到平衡二叉树，这样就可以快速查找任何与虚拟地址相关的区域。

内核还有第二种地址空间的物理视图。视图存储在进程的硬件页表中。页表入口决定虚拟内存每个页面的确切位置，而不管它是否位于磁盘中还是在物理内存中。当进程要访问当前并不存在于页表中的页面时，一组程序便从内核中调用软件中断处理程序来管理物理视图。在地址空间描述中每个 **vm_area_struct** 结构包含了指向函数表的字段，它能为任意给定的虚拟内存区域执行关键页面管理函数。所有读写无效页面的请求都被分派到 **vm_area_struct** 函数表的适当处理程序中去，这样中心内存管理程序就无需知道内存区域管理每个可能类型的细节。

1. 虚拟内存区域

Linux执行多种虚拟内存区域。第一种代表某类虚拟内存的属性是区域后备存储：这种存储描述了页面来自何处。大多数的内存区域要么通过文档备份，要么什么都不需通过而直接备份(这是虚拟内存中最简单的备份方式)。这样的区域叫做**按需填零内存**(demand-zero memory)：当进程试图把一个页面读入该区域，它只是简单地返回一个填满零的内存页面。

文件备份区域充当了部分文件的观察点，无论什么时候进程从这个区域访问页面，页面表存有对应的页面地址，该页面存在于内核页面cache中，并且对应于文件偏移位置。物理内存的同一页面被页面cache和进程页表同时使用，因此文件系统对文件的任何改变会马上被映射到地址空间，并被所有进程发现。任意数量的进程可以映射同一文件区域，它们也可以由于某种原因终止使用相同的物理内存页面。

通过对写操作的反应也定义了虚拟内存区域。进程地址空间的映射区既是私有的也可以是共享的，如果进程对私有映射区进行写操作，那么分页程序就会发现必须使用写时复制

(copy-on-write),从而使得这些变化只是局限于该进程。另一方面,对共享域的写(操作)导致了映射到那个区域对象的更新,因此,这些变化马上被其他正在映射该对象的进程发现。

2. 虚拟地址空间的生存周期

内核需要在下述两种情况下创建新的虚拟地址空间:进程通过系统调用 exec 运行新的程序;由 fork 系统调用创建新的进程。第一种情况比较简单,当新的程序被执行时,系统就给予进程一个新的、完全空闲的虚拟地址空间。它取决于用虚拟内存区域把程序装载到地址空间的子程序。

第二种情况,用 fork 创建新进程将创建一个完全的现存进程虚拟地址空间的备份。内核复制父进程的 **vm_area_struct** 描述符,然后为子进程创建一组新的页表。父进程页表直接拷贝到子进程中,每个页面的引用计数也随之递增;这样,在派生(fork)后,父进程与子进程共享它们地址空间内存的同一物理页面。

当复制操作程序到达虚拟内存的私有区域时,就会产生另外一种特殊的情况。在这个区域中,父进程写入的任意页面都是私有的,无论是父进程还是子进程对这些页面的更改都不必在其他进程地址空间做出相应的修改。当区域页面入口被复制时,就把它们设定为只读并且标记为写时复制(copy-on-write)。只要两个进程都不修改这些页面,那么它们就共享物理内存页面。然而,其中任何一个进程想要修改一个写时复制页面,就会检查此页面的引用次数。如果页面仍然是共享的,进程就把页面内容复制到一个崭新的物理页面中,然后使用它的复制页。这种机制确保了私有数据页面在随时可能的进程之间共享;只有在万不得已的时候才会复制页面。

3. 交换与分页

虚拟内存系统的一项重要任务是,在需要时将内存页面从物理内存重定位到磁盘。早期的 UNIX 系统是通过立即交换整个进程内容的方式来实现重定位的,但是现代版本的 UNIX 更依赖于内存分页(技术)——虚拟内存的单个页面在物理内存与磁盘之间的移动。Linux 并不实现进程整体交换技术,它使用的是较新的内存分页机制。

分页系统可分为两部分。第一部分,**策略算法**(policy algorithm)决定哪个页面写到磁盘上以及什么时候进行写操作。第二部分,**分页机制**(paging mechanism)在必要时把页面数据转移和交换到物理内存。

Linux **页换出策略**(pageout policy)使用的是标准时钟(或第二次机会)算法(在 10.4.5.2 节中有详细描述)的修订版。Linux 使用多轮时钟,每个页面都有年龄(age)设置,每做一次轮回就调整一次年龄。年龄能衡量页面是否依然年轻,或者说最近页面的活跃程度。经常被访问的页面,它的年龄值比较高,而很少被访问的页面,年龄值将随着每次轮回逐渐降到零。年龄值将使分页程序根据 LFU(最近最少使用原则)选择被换出的页面。

分页技术机制支持对专用交换设备和分区以及标准文件的分页,尽管由于文件系统的额外消耗使得文件交换明显减慢。根据使用过的块(每次留存在物理内存)的位图,交换设

备分配的块总是保存在物理内存中。分配程序使用 next-fit 算法写页面，保持磁盘块的连续运行，从而提高了性能。通过现代处理器上的页表功能，分配程序记录了页面被换到磁盘上的情况。页表项的 page-not-present 位被设定后，允许其他的页表项填充索引，以此识别写出页表的地方。

4. 内核虚拟内存

Linux 为其内部保留了一个不变的、结构独立的、每个进程虚拟地址空间的区域。映射到这些内核页面的页表项都设置保护标志。这样当进程在用户模式下运行时，不能看见或者修改这些页面。这个内核虚拟存储区域包含两个区。第一部分是静态区域，它包含的系统中指向每个有效物理存储页面的页表。当运行内核代码时，产生了物理地址到虚拟地址之间的简单转换。内核核心，加上一般页面分配程序分配的所有页面，都驻存在这个区域。

剩下来的内核保留部分地址空间并没有特别的用途。在此地址范围的页表项可以被内核修改并按需指向其他内存区域。内核提供了一对允许进程使用这个虚拟内存的工具。函数 vmalloc 分配了一些随机的内存物理页面，然后把它们映射到一个内核虚拟内存区域，使得大量的毗邻存储块得以分配，甚至当前没有足够的相邻空闲内存空间来满足请求。函数 vremap 把一连串的虚拟地址映射到指向内存映射 I/O 设备驱动器所使用的内存区域。

20.6.3 用户程序的执行与装载

Linux 内核执行用户程序是通过 exec 系统调用来启动的。该调用命令内核在当前进程中运行新的程序，将新程序的初始关联环境完全覆盖进程当前执行关联环境。系统服务的第一项工作便是确认系统调用的进程对正在执行的文件具有操作权。一旦检查通过，内核调用装载子程序以便开始运行程序。装载子程序不必把程序文件的内容装入到物理内存中，但是至少需要建立程序到虚拟内存的映射。

Linux 没有装入新程序的单独程序。相反，Linux 保存了一个可能的装载程序函数表，当调用 exec 时，表中的每个函数都有机会试着装入给定的文件。保留这个装入程序的初衷是，在 1.0 版和 2.0 版的内核之间，Linux 的二进制文件标准格式改变了。老版本的 Linux 内核支持 a.out 格式的二进制文件——一普遍存在于老版本 UNIX 系统的一种相对简单的格式。较新的 Linux 系统使用的是较现代的 **ELF** 格式，已被大多数现代 UNIX 支持。ELF 格式比 a.out 格式有更多的优点，包括灵活性和可扩展性：新的部分增加了 ELF 二进制(例如，增加了调试信息)而不会使装载程序迷惑。通过允许多种装载程序，在一个系统里面 Linux 很容易支持 ELF 和 a.out 两个格式的文件。

在 20.6.3.1 和 20.6.3.2 中，将单独着重讨论 ELF 格式二进制文件的装入和运行。装入 a.out 二进制文件更简单，但操作起来还是比较相似。

1. 程序映射到内存

在 Linux 环境下，二进制装入程序并不把二进制文件装入到物理内存，而是把二进制文

件的页面映射到虚拟内存区域。只有在程序要访问某一页面时，页错误才会导致页面装入到物理内存。

内核二进制装入程序必须建立初始的内存映射。一个 ELF 格式的二进制文件由一个文件头和几个页面对齐部分组成。ELF 装入程序将读入的文件头和各个文件部分映射到虚拟内存的独立区域中。

图 20.4 列出了由 ELF 装入程序建立的典型内存区域布局。在预留区域中，地址空间的一端是内核，虚拟内存的私有区域禁止标准用户模式程序的访问。剩下来的虚拟内存可被应用程序使用，它可利用内核的内存映射函数建立部分文件映射或者是应用程序数据可用区域。

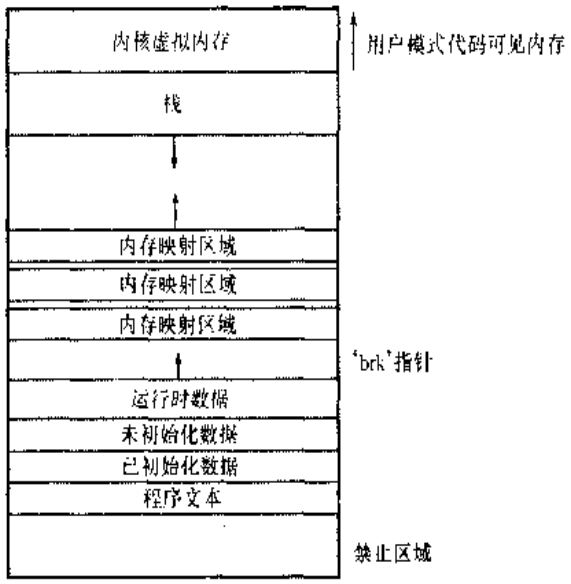


图 20.4 ELF 程序内存布局

装入程序的任务是建立初始的内存映射，使得执行程序得以开始。有待初始化的区域还包括栈、程序文本和数据区域。

栈建立在用户模式的虚拟内存的顶部；它随地址减小而减少。栈在调用 exec 时保存程序的参数及环境变量。其他区域建立在靠近虚拟内存的底端。包含程序代码或只读数据的部分二进制文件部分将作为写保护区映射到内存中。随后映射已初始化的可写数据；然后所有未初始化的数据被映射到私有的按需填零区域。

除了这些固定大小(fixed-sized)区域，就是变长(variable-sized)区域。在变长区域里，程序可根据需要扩展，在运行时间内得到分配的数据。每个进程都有一个 brk 指针，它指向这个数据区域的当前范围，进程也可以利用一次系统调用来扩大或者缩小它们的 brk 指针区域。

一旦这些映射已经被建立，装入程序就利用 ELF 头部记录的开始点来对进程的程序计数器寄存器进行初始化，这时进程可以被调度。

2. 静态链接与动态链接

一旦程序装入并开始运行，二进制文件的所有必要的内容都要加载到进程的虚拟地址空间中。然而，大多数程序也需要从系统库中运行函数，这些库函数也需要加载。举个最简单的例子，当程序员编译一个应用程序，所需的库函数被直接嵌入到程序的可执行二进制文件。这种程序静态链接到它的程序库，而且它们一旦被加载，与之相链接的静态可执行程序便马上开始运行。

静态链接的主要缺点是每个生成的程序都必须包含相同的公用系统库函数的拷贝。如

果考虑物理内存与磁盘空间的使用，把系统库一次性加载到内存显得更加有效。动态连接就能够允许只加载一次。

在用户模式下，Linux是通过一个特殊的链接库来实现动态链接的。每个动态链接的程序都包含一个小的静态链接函数，当程序开始运行时，该函数就被调用。静态函数只是将链接库映射到内存并运行函数包含的代码。链接程序库读入一个动态库表，是根据来自程序库的对程序、变量名和函数的需要，通过读取包含在ELF二进制段的信息(来实现)。然后将程序库转换到虚拟内存的中间，把参数变成程序库中的符号。这些共享的程序在内存的何处实行映射并没有多大关系。它们将被编译成能在内存任何地址运行的**位置独立的代码**(PLC)。

20.7 文件系统

Linux保留了UNIX的标准文件系统模型。在UNIX中，文件不一定要储存在磁盘或者从远程服务器上通过网络取得。然而UNIX文件有能力处理输入或者输出数据流。设备驱动器可以作为文件、进程通信通道或者网络连接出现，它看起来也像用户文件。

Linux通过将所有单个文件类型的实现细节隐藏到软件框架，来处理所有这些类型的文件，这一过程都产生在虚拟文件系统(VFS)。

20.7.1 虚拟文件系统

Linux VFS是根据面向对象原则来设计的。它有两个部分：文件对象看起来像是什么的一组定义；用来处理那些目标的一层软件。VFS定义的三个主要的对象类型是**inode-object**和**file-object**结构(描述单个文件)以及**文件系统目标**(描述整个文件系统)。

VFS为这三个对象类型定义了一组必须由这种结构实现的操作，每个对象类型都包含一个指向函数表的指针。此函数表罗列了为特殊目标执行操作的事实函数的地址。这样VFS软件层通过从目标函数表调用适当的函数就能执行在其中一个目标上的操作，而不需要提前知道要处理什么样的目标。VFS不知道或者不关心inode描述的是网络文件、网络插座还是目录文件。文件读数据(read data)操作的合适函数通常都在函数表的同一位置，VFS软件层调用此函数时并不关心数据的读入方式。

文件系统对象表述了一组相连接的文件，这些文件形成了一个自包含的目录级别。操作系统内核为每个以文件系统形式加载的磁盘设备和当前连接的网络文件系统保留一个文件系统对象。文件系统对象的主要作用是访问inode。VFS通过惟一对(pair)来定义每个inode。它通过询问文件系统对象所返回的inode号，找到inode所对应的特定inode号。

文件对象和inode对象是用于访问文件的机制。inode对象描述的是整个文件，而文件对象描述的则是文件中要访问的数据位置。进程在未获得指向inode文件对象的情况下不能访问inode的数据内容。文件目标跟踪进程对文件当前的读或写位置，跟踪顺序文件的

I/O 。当文件打开时，它还要记住进程是否请求写许可，并跟踪进程的活动，如果有必要执行自适应 read-ahead。进程发出请求，提前提取文件数据到内存，改善性能。

文件对象属于一个典型的进程，但是 inode 对象则不然。甚至当文件不再被任何进程使用，文件的 inode 对象还是为了提高性能而被 VFS 缓冲（前提是不久后文件被再次使用）。所有缓冲的文件数据都与文件 inode 对象表连接。inode 还保留了每个文件的标准信息，诸如，文件所有者、文件大小以及最近修改的时间等。

目录文件处理起来和其他的文件有一些细微的差别。UNIX 程序设计接口定义了一系列目录操作方式，例如创建、删除以及对目录中文件的重命名。和读写数据必须首先打开要操作的文件不同，对这些目录操作的系统调用不需要用户打开相关的文件。所以，VFS 是在 inode 对象（而不是文件对象）中定义这些目录的操作方法。

20.7.2 Linux ext2fs 文件系统

因为历史原因，Linux 使用的标准磁盘文件系统称为 ext2fs。Linux 最初设计使用 Minix 兼容的文件系统，目的是便于和 Minix 开发系统进行数据交换。但是这种文件系统严重局限于 14 字符的文件名，并且文件不能超过 64 MB。Minix 文件系统被一种新的文件系统所取代，此新系统叫做**扩展文件系统**（extfs）。后来这个文件系统为要提高性能和升级性又重新设计，并增加一些缺少的功能，这就成了**第二版的扩展文件系统**（即 ext2fs）。

ext2fs 与 BSD Fast File System（ffs）存在很多共性。它使用相似的机制定位某个特定文件的数据块，整个文件系统使用三级间接块存储数据块指针。在 ffs 中，目录文件和标准文件一样存储在磁盘上，虽然它们的内容大相径庭。每个目录文件块由项目连接表组成，每个项目包含了项目内容长度、文件名、项目访问的 inode 对象的数量。

ext2fs 与 ffs 之间的主要区别在于磁盘分配策略。在 ffs 系统中，磁盘给文件分配了大小为 8 KB 的块，再把块分成 1 KB 的几个碎片（fragment），用来存储小文件或者是在文件末尾填充部分文件块。相反，ext2fs 根本就不使用碎片，而是在更小的单元里进行分配。虽然 ext2fs 也支持 2 KB 和 4 KB 的块，但块的缺省大小是 1 KB。

为了保持高性能，操作系统必须尽可能通过群集物理上相邻的 I/O 请求，从而在较大程序块中执行 I/O 。群集减少了由设备驱动器、磁盘和磁盘控制器硬件各自请求所引起的消耗。1 KB 的 I/O 请求实在太小以至于无法保证良好的性能，因此 ext2fs 使用分配策略将文件的逻辑相邻块放置到磁盘的相邻块中。这样它仅用一次操作就可以为多个磁盘块提交 I/O 请求。

ext2fs 分配策略有两个部分。像 ffs 一样，一个 ext2fs 文件系统被分割成多个**块组**（block group）。ffs 也使用类似**柱面组**（cylinder group）的概念——每个组都对应一个物理磁盘柱面。然而现代磁盘驱动技术根据磁盘的不同密度来压缩扇区（依赖于不同的柱面大小和磁头离磁盘中心的距离）。所以固定长度的柱面组不需要相应的磁盘结构。

ext2fs 分配文件必须首先为该文件选择块组。对于数据块，它试图选择同样的块组作为分配文件 inode。对于非目录文件，它选择与文件上级目录相同的块组进行 inode 分配。目录文件并不放在一起，而是散布在整个可用的块组中。这些策略被设计成在相同的块组中保存互相关联的信息，同时也在磁盘的块组中分散磁盘的负荷，从而减少磁盘碎片。

在块组中，ext2fs 尽可能使分配保持物理连续，并且尽可能减少碎片。它保留了块组中所有空闲块的位图。当为新文件分配第一个块时，便从块组开始位置查找一个空闲块；当扩展一个文件时，它就从最近分配给文件块的位置开始搜索。搜索分两个阶段。第一阶段，它在位图中搜索一个完整的空字节；如果没有找到，它就寻找任意空位(free bit)。搜索空字节的目的是在可能的地方把至少八个大块的磁盘空间分配出去。

一旦一个空闲块被确认，系统继续向后搜索直到遇见一个已被分配的块。如果在位图中发现一个空字节，则向后搜索防止 ext2fs 在先前非零字节中最近分配的块与搜索到的零字节之间产生孔。一旦八位或者是字节搜索到了下一个要分配的块，ext2fs 就向前扩大分配到八块为止，并对文件预先分配这些特定的块。这样的预先分配在交叉存取的写操作过程中有助于减少碎片，同样也减少了因为同时分配多个块所引起的 CPU 磁盘分配消耗。在关闭一个文件后，预先分配的块返回到空地址位图当中。

图 20.5 说明了分配策略。每行代表分配位图中一系列设定的和未设定的位(bit)，显示了磁盘上已使用的和空闲的块。在第一个例子中，如果在开始检索的块附近能找到任意足够的空闲块，那么不管它们有多么破碎也要进行分配。如果块之间连在一起并且可能无需磁盘定向就能都被读出，那么磁盘碎片能得到部分补偿。从长远看来，一旦磁盘上大块空闲

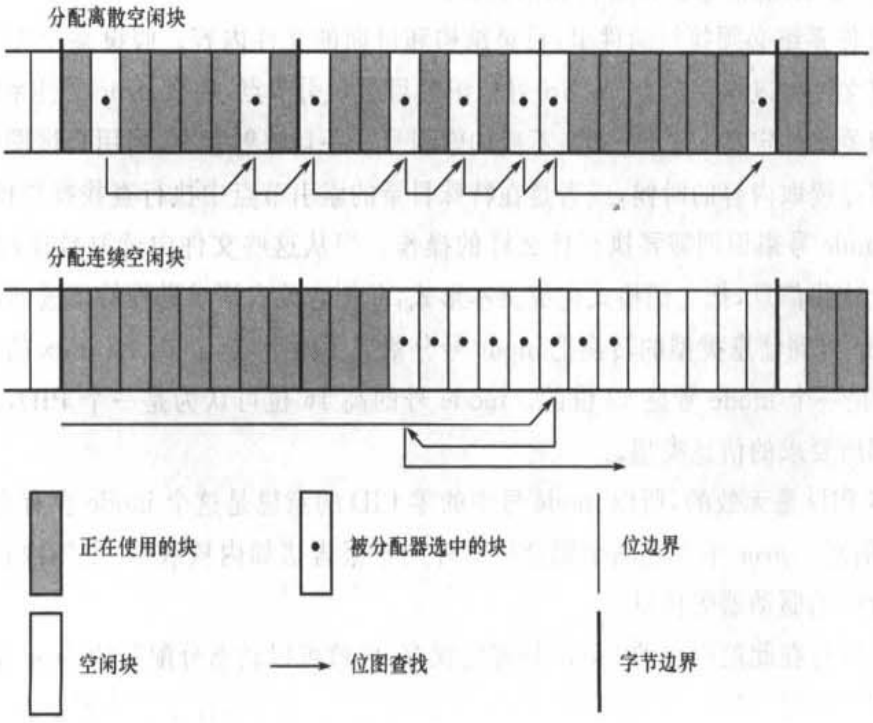


图 20.5 ext2fs 块分配策略

区域不足时，把磁盘碎片全部分配到一个文件要比把它们分配到各个文件好得多。在第二个例子中，无法立即在附近发现空闲块，因此在位图中查找整个的空闲字节。如果把字节当做一个整体进行分配，最终只是在空闲区域(free space)前新建了一个碎片区域，所以在分配前对此做了备份并且将要分配的资源进行刷新，然后执行默认的八块分配。

20.7.3 Linux Proc 文件系统

Linux VFS 十分灵活，它甚至可能实现数据毫不连续的文件系统，并为其他的功能提供一个简单的接口。Linux **进程文件系统**(process file system)，就是所说的 proc 文件系统，其内容实际上没有在任何地方存储，但还是根据用户文件 I/O 请求的需要来计算。

proc 文件系统并不只限于 Linux。SVR4 UNIX 把 proc 文件系统作为一个有效的接口提供给内核进程的调试支持程序：文件系统的每个子目录对应的不是磁盘上的某一目录，而是当前系统运行的进程。文件系统表显示的是每个进程一个目录，并且 ASCII 十进制表述的目录名是进程惟一的**进程标识符**(PID)。

Linux 通过在根目录下增加临时目录和文本文件数量的方式大大扩展了 proc 文件系统。这些新增项对应于内核和相关装入驱动器的各种统计数值。proc 文件系统作为纯文本文件(标准 UNIX 用户环境给进程提供了强大的工具)为程序访问这些信息提供了一种方法。例如，过去传统的 UNIX ps 命令(对象是所有运行进程状态的列表)作为从内核虚拟内存直接读取进程状态的一个特权进程被实现。在 Linux 下，此命令是一个完整的非特权程序，它只是从 *proc* 的信息进行解析和格式化。

proc 文件系统必须执行两件事：目录结构和里面的文件内容。假设某个 UNIX 文件被定义为一组文件和以索引节点(inode)号标识的目录索引节点，那么 *proc* 文件系统必须为每个目录和相关文件定义一个惟一的、不变的索引号。一旦映射存在，当用户试图从一个特殊文件索引节点读取内容的时候，或者是在特殊目录的索引节点中执行查找操作的时候，它可利用这个 inode 号来识别需要执行什么样的操作。当从这些文件中读取数据时，*proc* 文件系统将组合这些信息，把它们格式化成文本形式，并把它放入请求进程的读缓冲区。

从 inode 号到信息类型的转换把 inode 号分割成了两个域(field)。Linux 的一个 PID 有 16 位宽，但是一个 inode 号是 32 位的。inode 号的高 16 位可认为是一个 PID，剩下的位用来定义进程所要求的信息类型。

一个零 PID 是无效的，所以 inode 号中的零 PID 的意思是这个 inode 含有全局(而非特定进程的)信息。*proc* 中存在的个别全局文件用于报告诸如内核版本、空闲内存、性能状态和当前运行中的驱动器等信息。

并不是所有在此范围内的 inode 号都被保存：内核可以动态分配新的 proc 索引节点(该索引节点动态映射、保存一个已分配 inode 号的位图)。它还保存了已注册全局 *proc* 文件系统项的树型数据结构：每一项都包含有文件的索引节点号码、文件名、存取许可和用于产生

文件内容的特定函数。驱动程序能随时注册和注销此树型结构中的表项，而其特别部分(即出现于/*proc*/*sys* 目录下)是为内核变量保留的。一组能读写这些变量的普通的句柄处理此树型结构的文件，因此系统管理员能够在 ASCII 十进制下通过把所要求的新变量写到适当文件的方式来简单调整内核参数变量。

为了能够在应用程序中有效访问这些变量，可以通过 sysctl 这个特殊的系统调用使用/*proc*/*sys* 子树。sysctl 采用二进制，而不是文本格式对这些变量进行读写操作，它也无需消耗文件系统资源。函数 sysctl 不是最好的工具，它对 proc 动态项目树进行写操作只是决定应用程序将要引用哪一个变量而已。

20.8 输入与输出

对用户来说，Linux 系统的 I/O 系统与其他 UNIX 系统的 I/O 非常相似，从某种意义上说，所有的设备驱动器都以常规文件方式出现。用户可以向设备打开一个访问通道，就像打开任何其他文件一样——在文件系统中，设备是作为对象出现的。系统管理员可以在文件系统中创建特殊文件，该文件指向设备驱动器，用户打开这样的一个文件就可以对指向的设备进行读写操作。文件保护系统决定用户所能访问的文件。通过使用普通的文件保护系统，管理员就可以给每个设备设置访问权限。

Linux 把所有的设备分成三类：块设备、字符设备和网络设备。图 20.6 列出了设备驱动系统的总体结构。**块设备**(block device)包括了允许自由访问完全独立的、固定大小数据块的所有设备，比如硬盘、软驱和 CD-ROM。块设备典型地用来存储文件系统，但是直接访问某一块设备也是允许的，这样程序可以创建和修复设备所包含的文件系统。应用程序也可以按照意愿直接访问这些块设备。例如，数据库应用程序使用它自己的磁盘数据组织方式，而不是使用通用的文件系统。

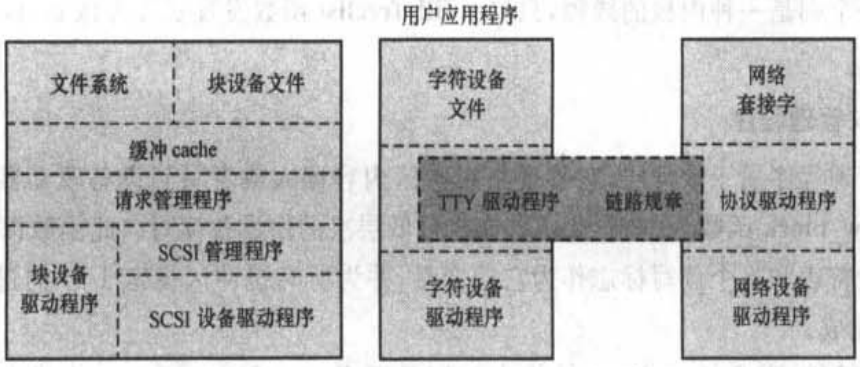


图 20.6 设备驱动器块结构

字符设备包括除了网络设备以外的其他设备。字符设备不需要支持普通文件的所有功能。例如，扬声器设备也可以写入数据，但不支持数据的读出。类似地，磁带设备支持在某

一位置寻找文件，但不需要类似于鼠标的定点设备的支持。

网络设备与块设备和字符设备有所区别。用户不能直接把数据转换到网络设备，必须通过打开内核网络子系统的连接进行间接通信。将在 20.10 节单独讨论网络设备接口。

20.8.1 块设备

块设备为系统的所有磁盘设备提供了一个主要接口。对磁盘来说，性能显得尤为重要，块设备系统必须提供确保尽可能快速的访问磁盘的功能。这种功能通过**块缓冲 cache**(block buffer cache)和**请求管理程序**(request manager)这两个系统部件实现。

1. 块缓冲 cache

Linux **块缓冲 cache** 主要有两个用途：它既是运行中 I/O 的缓冲池，又是完成后 I/O 的 cache。块缓冲 cache 由两部分组成。首先，缓冲区本身是一组由内核的主存池直接分配的、大小动态变化的页面。每个页面被分割成了许多大小相等的缓冲区。其次，cache 中的每个缓冲区都包含一组相应的缓冲区描述符——*buffer_heads*。

buffer_heads 包含内核保存的关于缓冲区的所有信息。其主要信息便是缓冲区的识别。每个缓冲区都通过以下三种情况进行识别：块设备属于哪个缓冲区，块设备数据的偏移量，缓冲区的大小。缓冲区还维护一些列表，例如用于干净的、脏的、锁定的以及空闲缓冲的列表。缓冲区通过文件系统放置(例如，删除文件的时候)或者是通过一个 **refill_freelist** 的程序方式输入空闲列表，此程序内核需要更多的缓冲区时可随时调用。内核填充空闲列表的方式是通过扩大缓冲池或者循环使用现有的缓冲区，通常是根据可利用的空闲内存是否足够(来判断是否需要填充)。最后，不在空闲列表的每个缓冲区由哈希函数根据其设备号和块号索引，并且与一个相应的哈希查找列表相连接。

内核缓冲器管理自动将脏缓冲器写回到磁盘，并由两个后台程序协助。其中一个只是在正常的时间间隔内被唤醒，并且缓冲区被更改的时间大于某一特定的时间间隔时，将数据写回。另一个则是一种内核的线程，只要 **refill_freelist** 函数发现脏缓冲区 cache 的比例太高就被唤醒。

2. 请求管理程序

请求管理程序属于软件层，它管理将缓冲区内容读入或者写到设备驱动器。请求系统主要以 **ll_rw_block** 函数为中心，此函数能执行低层次的块设备读写。此函数以 **buffer_head** 缓冲区描述符表和一个读写标志作为它的变量，并为所有缓冲区设定 I/O 的进度。它不等待 I/O 的完成。

典型的 I/O 请求在 request 结构中被记录下来。一个 request 结构代表一个典型的 I/O 请求(在单个块设备上读写连续范围的扇区)。在操作中可能不止一个缓冲区，所以 **request** 包含了一个指向列表第一项的 *bufffer_heads* 指针。

每个块设备驱动程序都有一个单独的请求表，并由单向电梯(C-SCAN)算法来调度这些

请求。单项电梯算法使用从每个设备表中插入或移动请求的顺序。系统以启动扇区号的增长顺序来排列保存请求表。当一个请求被设备驱动接受时，并不马上从列表中移走，只有当I/O完成后该请求才能被移走。即使在运行请求之前新的请求插入表中，驱动程序仍然继续表上的下一个请求。

当产生一个新的I/O请求时，请求管理程序就试图合并每个设备表的请求。从设备的消耗上看，传递一个单一的大请求要比传递许多小请求更有效。只要产生初始的I/O请求，在这样一个合并的请求中所有的*buffer_heads*都被锁定。当设备驱动程序处理请求时，单个*buffer_heads*组成请求集合，一次解开一个锁，一个进程等待一个缓冲并不需要请求所有的其他缓冲都解开。这个因素特别重要，因为它能确保read-ahead能够有效地被执行。

请求管理器最后一个特点是可以完全绕过缓冲cache发出I/O请求。低级页面I/O函数，**brw_page**，新建临时*buffer_heads*集标明了存储页面的内容，目的是提交I/O请求到请求管理器。然而，这些临时的*buffer_heads*并没有连接到缓冲缓存器，一旦页面上的最后的缓冲完成了它的I/O，整个页面就被解锁，*buffer_heads*也随之释放。

只要是函数调用者处理独立的数据存储，或者只要知道数据不会再被缓冲，内核就使用绕过cache(cache-bypassing)这个机制。页面cache用这种方式填充它的页面，从而避免不必要的、相同数据同时在页面cache和缓冲缓存器之间存储。虚拟存储系统在执行I/O交换设备时同样要绕过cache。

20.8.2 字符设备

字符设备驱动程序可以是任何不能对固定数据块进行随机访问的设备。任何字符设备驱动程序注册Linux内核的同时必须也要注册一组函数，这些函数执行驱动程序能处理的文件I/O操作。内核执行大多数没有经过预处理的对字符设备文件的读写请求，但是只简单地把有问题的请求传递给设备，并让设备来处理这个请求。

字符设备驱动程序的特殊子集是实现终端设备的一个主要例外。内核通过一组*tty_struct*结构保存这些驱动程序的标准接口。每个结构都提供缓冲区和来自终端设备数据流上的流控制，也提供了链路规程对应的数据。

链路规程(line discipline)是一个对来自终端设备信息的解释程序。最普通的链路规程是tty规程，它把终端数据流粘贴在正在运行的用户进程的标准输入/输出流上，这些进程才得以和用户终端进行直接交流。因为多个进程同时运行的情况使得此项工作变得复杂，tty链路规程负责连接和释放来自与之相连接的各个进程终端的输入与输出，就像进程被用户唤醒和悬挂。

实现其他的链路规程与用户进程的I/O无关。PPP和SLIP网络协议是通过一个终端设备(诸如串行链路)对网络连接进行编码。这些协议在UNIX下以驱动程序形式实现，一端是作为链路规程出现在终端，另一端是作为一个网络设备驱动程序出现在网络系统。当

一个终端激活其中的一个链路规程后，终端上的任一数据将被直接发送到正确的网络设备驱动程序。

20.9 进程间通信

UNIX 为进程之间的通信提供了一个丰富的环境。通信可以只是让别的进程知道事件已经发生，也可以从一个进程到另一个进程之间传送数据。

20.9.1 同步与信号

通知进程事件已发生的标准 UNIX 机制是**信号**(signal)。信号可以在任意进程之间互相发送。信号被发往另一用户拥有的进程是有条件限制的。然而，可以使用一些受限信号，但它们不能传送信息；惟一的情况是进程可以使用已存在的信号。其他的进程不需要产生信号。内核内部也会产生信号。例如，当数据到达某一网络通道时，它向服务进程发送一个信号，或者当子进程终止时向父进程发送信号，或者是定时器到时。

Linux 内核内部进程在内核模式下运行时并不使用信号进行通信；如果一个内核模式的进程想要产生一个事件，通常它不使用信号来接收该事件的通知。事实上，关于内核中异步事件的通信要通过使用调度状态和 wait_queue 结构。这些机制可让内核模式的进程之间能够相互通知相关事件，它们也可由设备驱动器或者网络系统产生。只要是进程需要等待完成某事件，它就会自动置身于一个与之相关的 wait_queue 队列，并告诉调度程序它不再是一个合法执行，一旦事件完成后，它将唤醒等待队列的每个进程。这个程序使得多个进程等待单个事件。例如，几个进程都要从磁盘读文件，一旦数据被成功读入内存，那么它们都将被唤醒。

虽然信号已成为进程间异步通信的主要机制，但是 Linux 也执行 UNIX V 版的信号量机制，进程可以很容易地同等待信号一样等待信号量。但是信号量有两个优点：大量的信号量可在多个独立进程间共享而对多信号量的操作可以原子地执行。内部标准的 Linux 等待队列(wait_queue)机制通过信号量实现进程同步的通信。

20.9.2 进程间数据传输

Linux 进程间传输数据有多种机制，标准的 UNIX **管道**(pipe)机制允许子进程从其父进程继承一个通信通道；写进管道一端的数据可以在另一端读出。在 Linux 下，管道仅仅是以 inode 的另一类型(形式)出现在虚拟文件系统软件，每个管道都有一对 wait_queue 使得读写程序同步。UNIX 同时也定义了一组网络设备，这些设备既可对本地的进程，又可对远程的进程发送数据流。网络的内容在 20.10 节中讲述。

进程间共享数据还有其他两种方法。首先，共享内存提供了传送大小不等的数据的一

种极为快速的方法：任何被一个进程写到共享区域的数据可以立即被其他已把该区域映射到地址空间的进程读出。共享内存的主要缺点是，就它自身而言，不支持同步操作：进程从不过问操作系统某一个共享的内存是否被写进，在写操作发生之前也不悬挂执行程序。把共享内存和其他进程间通信机制（提供失去的同步）一起使用，共享内存就变得特别强大。

Linux 中共享内存区域是一个不变的对象，可被进程创建和删除。该对象被当做只是一个很小的独立地址空间一样处理。Linux 的分页算法可以挑选页面到磁盘的共享内存页面，就好像它们换出进程数据页面。共享内存对象充当了共享内存区域的后备存储器，正像文件可以充当内存映射内存区域的后备存储器一样。当一个文件被映射到虚拟地址空间区域，那么任何发生的页错误都将导致合适的文件页面被映射到虚拟区域。相似地，共享内存映射直接将页错误从某一不变的"共享内存"目标中转换页面。和文件一样，尽管当前没有进程把它们映射到虚拟内存，共享内存对象依然记住它们的内容。

20.10 网络结构

网络是 Linux 的关键功能，Linux 既支持标准的因特网协议（用于大多数的 UNIX-to-UNIX 通信），也能实现本系统到其他非 UNIX 操作系统的许多协议。Linux 最初主要运行在个人计算机上，而不是大型工作站或是服务器系统，它支持多种用于个人计算机网络的协议，如 AppleTalk 和 IPX。

Linux 内核中的内部网络通常通过以下三个软件层实现：

1. Socket 接口
2. 协议驱动器
3. 网络设备驱动器

用户应用程序通过 socket 接口执行所有的请求。该接口类似于 BSD 4.3 socket 层，因此任何使用 Berkeley sockets 的程序不用改变源代码就能在 Linux 上运行。这个接口已在 4.6.1 小节讲到。BSD socket 接口十分通用，它能描述范围广泛的网络协议的网络地址。这个 Linux 使用的单一接口不仅限于标准 BSD 系统中实现的协议，而且适用于所有被系统所支持的协议。

下一个软件层就是协议栈，在组织形式上它与 BSD 结构有相似之处。不管是来自某一应用 socket 还是来自网络设备驱动器，只要这些任意的网络数据到达这个层，就会在标识符上做记号，并说明它们包含哪种网络协议。如果有必要，协议之间也可以进行通信。例如，在因特网协议集中，单个协议管理着路由、错误报告、丢失数据的可靠再传输。

协议层可以改写数据包，新建数据包，把数据包分割成碎片或者重新装配，或者是简单地抛弃刚输入的数据。最后，一旦它处理完数据包，如果数据要送往一个本地连接就把它们往上传递到 socket 接口，或者如果数据包需要远程传送的话，就往下传送到某一设备驱动

器。协议层决定把数据包发送到哪个 socket 或者哪个设备。

网络栈层之间的所有通信是通过传递单一的 **skbuff** 结构来实现的。一个 skbuff 结构包含有一套指向单个相邻内存区域的指针，它描述了构建网络数据包的缓冲区。skbuff 中的有效数据在该缓冲区的开始阶段不需要启动，也不需要运行结束。网络代码可以将数据添加到包的任何一端，或者是在数据包的两端修改数据，只要结果符合 skbuff 结构。对于现代微处理机容量显得特别重要，CPU 速度的改进已大大超出了主存储器的性能；skbuff 体系结构在处理包头部和校验和方面也有灵活性，从而避免了不必要的数据备份。

Linux 网络系统协议中最重要的是因特网协议（简称 IP）组，这个协议组由大量独立的协议组成。IP 协议可以在网络上任意地方的两台不同主机之间实现路由。在路由协议的顶端是 UDP、TCP 和 ICMP 协议。UDP 协议在主机间传送着随机的单个数据包。TCP 协议实现主机间可靠连接的同时，必须保证数据包的有序传递和丢失数据的自动重新传输。ICMP 协议用于传输主机间各种错误信息和状态信息。

数据包（skbuff）在到达网络栈协议之前最好标上一个显示与之相关协议的内部标识符。不同网络设备驱动器在其传播介质上使用不同的协议类型编码方式。这样，输入数据的协议识别必须在设备驱动器中完成。设备驱动器使用一个已知网络协议识别符的哈希（hash）表来查找合适的协议，并把数据包传输给协议，新的协议可作为内核可装入模式加载到哈希数据表。

输入的 IP 数据包被传输到 IP 驱动器。该层的工作就是实现路由：它决定数据包传送地址，并把它传输到合适的内部协议驱动器（在本地传输）；或者是把它放入一个已选好的网络设备驱动器队伍，并把它传送到另外的主机。它通过两个表实现路由方案：持续传送的信息库（FIB）和存储当前路由方案的高速缓存器。FIB 包含了路由配置信息，并描述了基于特定目的地址或者是描述多目的地的多点路由。FIB 由一组目的地址检索的哈希表组成；描述大多数特定路由的列表通常被第一个搜索到。对此表的成功查找（后的数据）被加入到 route-caching 表中，此表以特定目的地址存储路由；cache 中不存储通配符，因此查找起来很快。在固定时间段内没有命中，路由 cache 中的项目就会被终止。

在各个阶段，IP 软件将数据包传送到**防火墙管理**（firewall management）的一个独立代码区，通常出于安全的考虑，根据随机标准有选择地进行过滤数据包。防火墙管理程序保留了大量的**防火墙链**（firewall chain），并使得任意链与 skbuff 相匹配。防火墙链有其特定的用途：首先用于传输数据包，其次是用于向该主机输入的数据包，最后是用于在该主机中产生的数据。每个防火墙链都是一个规则有序的表，其中的一个规则将描述大量可能的防火墙方案功能的一个以及一些与之相匹配的随机数据。

通过 IP 驱动器实现的其他两个功能是大数据包的拆卸和重新组装。如果输出的数据包太大而不能加入到某个设备的队列中，那么只要把它分解成小一点的**碎片**（fragment），并把它们都放到驱动器的队列中。当到达主机时，这些碎片必须被重新组合。IP 驱动器为每

一个等待重组的碎片都保留了一个 ipfrag 对象，并且在组合时为每一个碎片保留了一个 ipq。输入的碎片必须和某个已知的 ipq 相匹配。如果找到一个匹配的碎片，就把它放入对应的 ipq 中；否则，就要新建一个 ipq。一旦最后一个碎片到达 ipq，就组成了一个保存完整的新数据包的 skbuff，这些数据包将被传回 IP 驱动器。

目的地为该主机的 IP 匹配的数据包将被传送到其他一个协议驱动器上。UDP 和 TCP 协议用同一种方式把数据包和源 socket、目标 socket 联系起来：只有它的源地址和目标地址、源端口号及目标端口号才能识别每个相连接的 socket。哈希表以这四个"地址一端口"值(address-port)来查找关键字，并与 socket 表连接。TCP 协议必须处理不可靠连接，因此它保留了有关的有序表：无确认输出数据包在超时后的重发表，丢失的数据到达时向 socket 描述输入无序数据包的表。

20.11 安 全

Linux 的安全模型与典型的 UNIX 安全机制密切相关。安全关系可分为两组：

1. **认证**：首先在保证有登录权限的情况下，才能访问系统。

2. **访问控制**：提供一种核查机制，检查用户是否有权访问某一目标，并且能根据需要禁止用户对某一目标的访问。

20.11.1 认证

UNIX 中的认证是通过使用公用可读的口令文件这种特别方式实现的。用户的口令与一个随机的"salt"值相结合，由单向传输函数进行编码，并将结果存储于口令文件中。使用单向传输函数意味着无法从口令文件中推导出原始的口令，除非是重试和发生错误。当用户给系统提交口令时，保存在口令文件中的 salt 值与口令实现重新结合并通过同样的单向传输方式。如果结果与口令文件中的内容相符，那么该口令就被接受通过。

历史上 UNIX 实现这种机制存在着几个问题。口令限于八个字符，可能的 salt 值太小，导致攻击者可以轻易地把通用口令字典和每个可能的 salt 值组合起来，并在口令文件中匹配一个或若干个口令，从而非法访问任意账号，结果将导致泄密。扩展后的口令机制文件中加密后的口令不再可读，并且允许更长的口令，或者说使用更安全方法对口令进行编码。其他引入的认证机制限制了允许用户连接系统或者是向网络中所有相关系统版本认证信息的次数。

UNIX 供应商开发出一种解决此类问题的安全机制。**可插入的鉴别模块**(pluggable authentication module，PAM)系统是基于一个任意系统组件(该组件用于认证用户)都能使用的共享库。在 Linux 下就可实现该系统。正如系统范围配置文件中描述的那样，PAM 根据需要装入认证模块。如果以后要加入新的认证机制，配置文件中也会加入这个机制，并且

所有的系统组件很快便能使用它。PAM 模块能指定鉴别的方法、账户限制、安装会话功能，或者是口令更改功能(因此，用户改变他们的口令时，所有必要的鉴别机制就要立即更新)。

20.11.2 访问控制

UNIX 系统下(包括 Linux 系统)的访问控制是通过使用惟一的数字标识符来实现的。用户标识符(简称 uid)识别的是某一个单个用户或者是一组访问权限。组标识符(gid)是用来识别若干个用户权限的一种特殊的标识符。

访问控制应用于系统中的多种对象。标准的访问控制机制维护着系统中每个可使用的文件，另外，其他的诸如共享内存段、信号量等共享对象使用的是同样的访问系统。

UNIX 系统中，用户和用户组访问控制下的每个对象都有一个对应的 uid 和 gid。用户进程也同样拥有一个单独的 uid，但却拥有若干个 gid。如果进程的 uid 与对象的 uid 匹配，那么进程对该目标就拥有**用户权限**(user rights)或者是说拥有**所有者权限**(owner rights)；如果 uid 不匹配但是进程 gid 中的任意一项与对象的 gid 相匹配，那么就授予**组权限**(group rights)，否则，进程拥有 **world 权限**。

Linux 是通过赋予对象一**种保护掩码**(protection mask)的方式来实现访问控制，该保护掩码将具体描述在有所有者、组和 world 访问的情况下，允许对进程访问的方式(读、写、执行等)。这样，对象的所有者就可以对文件进行完全的读、写、执行等访问；特定组中的其他用户可以进行读访问但不能进行写访问；而其他用户则没有任何访问权。

惟一的例外是特许的**根**(root)uid。带有这种特殊 uid 的进程绕过了通常情况下的访问检查，并允许自动访问系统中的任意对象。这样的进程允许进行某些特许的操作，如读取任意的物理内存，或者是打开保留的网络 socket。在这个机制下，内核就能防止普通用户访问这些资源：大多数重要的内部内核资源毫无疑问归根 uid 所有。

Linux 执行标准的 UNIX setuid 机制(在 A.3.2 节中有介绍)。在该机制下，运行有特许权的程序不同于那些用户运行的程序：比如 lpr 程序(给打印机队列提交工作的程序)能访问系统的打印队列而运行该程序的普通用户却不行。在进程的*真实*和*有效*的 uid 之间，UNIX 执行 setuid 是有所区别的：用户运行的程序属于真实 uid；文件拥有者的属于有效 uid。

在 Linux 下，有两种方式可增强这个机制。第一种是，Linux 实现 POSIX 中描述的 saved use-id 机制，在该机制下，进程可以反复地丢失、重新获得它的有效 uid。为了安全原因，程序需要在一个安全的模式下进行它的大多数操作，它将放弃 setuid 状态所给予的特许权，但是希望用它的特许权来执行选择过的操作。标准 UNIX 通过交换真实 uid 和有效 uid 实现这个功能。系统保存了先前的有效 uid，但是程序的真实 uid 并不与用户运行程序的 uid 相符。保存过的 uid 允许某一进程对它的真实 uid 设置有效 uid，然后返回到它的有效 uid 的原值，而无需随时修改它的真实 uid。

Linux 提供的第二种加强方式是，添加承认有效 uid 权限子集的进程属性。访问文件的权限得到许可后方能使用 **fsuid** 和 **fsgid** 的进程属性，并且每当有效 uid 和 gid 被设置后，都要设置该属性。然而 fsuid 和 fsgid 也可脱离有效的 id 进行单独设置。这样进程可代表其他用户不通过验证而使用别的方式访问文件。比较特殊的是，服务器进程可利用这个机制服务于某一特定用户的文件，而不用担心该用户杀死或者悬挂此进程。

Linux 提供的另外一个机制在现代版的 UNIX 中得到越来越普遍的应用，那就是两个程序之间权限的灵活传递机制。当一个本地网络 socket 在系统的任意两个进程间建立时，任一进程都有可能给另外一个进程中的其中一个开放文件发送一个文件描述符；而其他的进程则接收到来自同一文件的一个复制文件描述符。这个机制下，客户可以从一个挑选过的单个文件访问到某些服务器进程，不需要授权进程的任何特许权。如，打印服务器没有必要读取用户提交的某一项新打印任务中的所有文件，打印客户只要把服务器文件描述符传递给需要打印的任意文件，拒绝服务器访问用户其他文件中的任何一项。

20.12 小 结

Linux 是一个基于 UNIX 标准的现代开放式操作系统。它能高效而稳定地运行在普通计算机硬件上，也能运行于其他各种平台上。它提供了一个能与标准 UNIX 系统兼容的程序接口和用户接口，并能运行大量 UNIX 应用程序，包括在数量上日益增长的商业软件。

Linux 并不是在真空中发展而成的。一个完整的 Linux 系统包括了许多独立于 Linux 而开发出来的组件。Linux 操作系统的核心是原先的版本，但它支持很多已有的自由软件，从而导致了一个 UNIX 完全兼容的操作系统从商业代码中解放出来。

因为性能的原因，Linux 内核以传统的单块内核形式实现，但是在设计方面它能实现模块化，大多数的驱动器在运行时间片内都能自行装载和卸载模块。

Linux 是一个多用户系统，根据分时系统调度程序在进程和运行中的多进程间提供保护。刚创建的进程可与父进程共享部分选择过的执行环境，并且支持多线程编程。系统 V 版机制（即消息队列、信号量和共享内存）和 BSD 的网络接口都支持进程间通信。通过网络接口可以同时访问多个网络协议。

对用户而言，文件系统呈现的是一个遵循 UNIX 语义的分级目录树。在其内部，Linux 使用了抽象层的概念来管理多个不同的文件系统，并且支持面向设备的、网络化的和虚拟的文件系统。面向设备的文件系统通过两个高速缓存器来访问磁盘存储器：数据被存储在与虚拟内存系统一体化的页面高速缓存器中；元数据被存储在缓冲器中（一种独立的高速缓存器，由物理磁盘块索引而成）。

存储器管理系统通过页面共享和写时复制（copy on-write）最大程度地减少与不同进程共享数据的复制。当首次引用页面时是根据需要装入的，根据 LFU 算法如果需要备份存

储，则把页面调出来回到后备存储器。

习题二十

20.1 Linux运行于多种硬件平台。对于不同的处理器和内存管理体系结构而言，Linux开发者必须采取什么样的措施来保证系统是活动的，并减少体系结构特殊化的内核代码数量？

20.2 系统中加入驱动器时，动态的可装入的内核模块显得很有灵活。请问它们有哪些缺点？在什么情况下，内核将被编译成一个单一的二进制文件？什么时候把它们分割成模块比较好些？请解释答案。

20.3 多线程技术是普遍使用的程序设计技术。请描述实现线程的三种方式。并解释这些方法怎样和Linux克隆机制做比较的。比起使用克隆而言，每个可供选择的机制什么时候用好一些？什么时候差一些？

20.4 跟克隆一个线程的代价相比，通过创建和调度一个进程要引发什么样的额外成本？

20.5 Linux调度程序执行软(soft)实时调度。特定的实时程序设计任务中遗漏了哪些必要的功能？怎样向内核中加入这些功能？

20.6 Linux内核不运行从内核存储器中调页。在内核设计方面，这种限制会有什么影响？这个设计决策中有哪两个优点和哪两个缺点？

20.7 在Linux中，共享程序库执行操作系统中多种重要的操作。不使用内核的这种功能性的优点是什么？有什么缺点吗？请加以解释。

20.8 程序库的动态连接(或者是共享)与静态连接相比有哪三个优点？在哪两种情况下静态连接更好一些？

20.9 作为一种机制，单一计算机上进程之间要进行通信数据，请比较网络socket接口的使用与共享存储器的使用。请列举出每种方法的两个优点。分别在什么情况下，选用哪一个将好一些。

20.10 基于磁盘数据的旋转位置，UNIX系统过去使用"磁盘布局"最优化，但是，在现代的实施过程中(包括Linux)，只是优化了顺序数据访问。他们为什么要这样做？顺序访问利用了硬件特性中哪方面的优点？为什么旋转优化不再那么有用处？

20.11 Linux的源代码在因特网上或者是从CD-ROM供应商那里可以普遍地免费得到。对Linux系统的安全性来说，这种可用性暗示有哪三个方面的意义？

推荐读物

Linux系统是因特网的产物，因而有关Linux的大多数文档都可以通过因特网以某种方式获得。以下是一些关键站点，涉及了最有用的信息：

- Linux交叉引用页 http://lxr.linux.no/ 包含有当前Linux内核的列表，可以通过Web浏览，并且可以完全交叉引用。
- http://www.linuxhq.com/上的Linux-HQ包含了大量有关Linux 2.x内核的信息。这个站点也包括了多数Linux发行版本的主页链接，还有一些主要的邮件列表档案。
- http://sunsite.unc.edu/linux/ 的Linux文档项目(LDP)列出了许多有关Linux的

书籍，以原始资料的形式作为 Linux 文档项目的一部分。这个项目也包括了 Linux *How-to* 指南，这个指南包含了一些有关 Linux 的提示和技巧。

• *The Kernel Hackers' Guide* 是基于因特网的普通内核内部结构的指南，位于 http://www.redhat.com:8080/HyperNews/get/khg.html，且此站点内容不断得到扩充。

另外，有许多致力于 Linux 的邮件列表。最重要的列表由邮件列表管理者维护，可以通过电子邮件地址 majordomo@vger.rutgers.edu 访问。如果想知道如何访问邮件列表服务器，如何订阅列表，给这个地址发一封电子邮件，正文只要一句"help"就可以了。

最近的一本描述 Linux 内核细节的书是由 Beck 等[1998]写的 *Linux Kernel Internals*。要进一步地深入阅读一般的 UNIX，可以从 Vahalia 的 *Unix Internals: The New Frontiers* (Vahalia[1996])开始。

最后，Linux 系统自身也可以在因特网上获得。完全的 Linux 发布版本可以从有关公司的主页上获得，因特网上一些地方的 Linux 社区也保存有当前系统组件的文档。最主要的如下：

• ftp://tsx-11.mit.edu/pub/linux/
• ftp://sunsite.unc.edu/pub/Linux/
• ftp://linux.kernel.org/pub/linux/

第二十一章　Windows 2000

微软公司 Windows 2000 操作系统是用于 Intel 公司推出的奔腾及其后续微处理器的 32 位抢占式多任务操作系统。它是继 Windows NT 后的又一个操作系统。在此之前称之为 Windows NT 5.0 版。该系统的重要(开发)目标是可移植性、安全性、符合可移动操作系统界面(POSIX 或 IEEE Std. 1003.1)规范、支持多处理机、系统的可扩展性、国际支持、与 MS-DOS 和 Microsoft Windows 的应用程序兼容。在这一章,将讨论系统的几个重要目标和系统分层体系结构。正是系统分层结构使得文件系统、网络和程序设计界面更易于使用。

21.1 历　　史

20 世纪 80 年代中期,微软公司和 IBM 公司曾经合作开发 OS/2 操作系统,该系统是用汇编语言编写的,适用于 Intel 80286 系统的单处理机。1988 年,微软公司决定重新开发一种叫做"新技术"(或者称为 NT)的可移植操作系统,同时支持 OS/2 和 POSIX 的"应用程序接口"(简称 API)。1988 年 10 月,DEC VAX/VMS 操作系统的设计师——Dave Cutler 受聘,并签订了构建这个新操作系统的合同。

刚开始,NT 开发组打算使用 OS/2 的 API 作为它的本地环境,但是在开发过程中,Windows NT 改为使用 32 位的 Windows API ,这反映了 Windows 3.0 受到欢迎。NT 的第一个版本是 Windows NT 3.1 和 Windows NT 3.1 高级服务器。(当时,16 位的 Windows 的版本是 3.1。)Windows NT 4.0 版采用了 Windows 95 用户界面并且加入了因特网 Web 服务器和 Web 浏览器软件。另外,为了提高性能,用户界面的例程和图形代码被移植到了内核,但是产生了副作用,降低了系统的可靠性。虽然以前的 NT 版本移植到了其他微处理器体系结构中,但是由于市场的因素 Windows 2000 停止了这种做法。这样,现在所说的**可移植性**指的是 Intel 体系结构系统中的可移植性。Windows 2000 使用的是微内核体系结构(如 Mach),在对其他部件没有大影响的情况下,操作系统的某一个部件的功能可以得到增强。加上终端服务器,Windows 2000 是一个多用户的操作系统。

Windows 2000 是在 2000 年发布的,而且有重大的改变。它加进了一个基于 X.500 的目录服务程序,改善了网络支持,支持即插即用设备,支持分层存储器的新型文件系统和分布式文件系统,同时支持更多的处理器和存储器。

Windows 2000 有四个版本。专业版适用于桌上型电脑的使用,其他的三个是服务器版

本，即服务器、高级服务器和数据资料处理中心服务器。它们之间的区别主要在于支持处理机和存储器的数量不同。它们使用同样的内核和操作系统代码，但是 Windows 2000 的服务器和高级服务器版本是为“客户服务器”应用程序配置的，可以在 NetWare 和 Microsoft LANS 网上充当应用程序的各种服务器。Windows 2000 的数据资料处理中心服务器现在可以支持多达 32 个处理机和 64 GB 的 RAM。

1996 年，售出的 Windows NT 服务器许可证比全部版本的 UNIX 许可证还多。有趣的是，Windows 2000 的代码库拥有三千万个代码行。与这个大小相比，Windows NT 4.0 的代码库则拥有大约一千八百万行代码。

21.2 设计原则

微软公司发布的 Windows 2000 的设计目标包括系统的可扩展性、可移植性、可靠性、兼容、高性能和国际支持。

可扩展性指的是操作系统能跟上先进计算技术的发展的能力。开发者通过分层结构实现 Windows 2000，这样便于修改升级。Windows 2000 执行体运行在内核或保护模式下，并提供基本系统服务。在执行体之上，多个服务器子系统运行于用户模式下。其中包括模拟不同操作系统的**环境子系统**(environmental subsystems)。因此，为 MS-DOS、Microsoft Windows、POSIX 所编写的程序可运行于 Windows 2000 的适当子系统中。(关于环境子系统的更多信息请参见 21.4。)由于模块化结构，可以在不影响执行体的情况下，增加其他环境子系统。另外，Windows 2000 在 I/O 系统中加入了可加载驱动程序，这样新文件系统、新 I/O 设备类型、新网络可以在系统运行时加入到系统中。Windows 2000 与 Mach 操作系统一样采用了 C/S 结构，并支持由 OSF 所定义的 RPC 形式的分布式处理。

如果一个操作系统能从一个硬件结构移到另一个硬件结构，且只需要少量修改，那么这种操作系统就称为**可移植**的。Windows 2000 被设计成可移植的。和 UNIX 操作系统一样，Windows 2000 的主要部分都是用 C/C++ 编写的。所有处理器相关代码都被封装在一个叫**硬件抽象层**(hardware-abstract layer，HAL)的 DLL 里。DLL 文件可以映射到进程地址空间，以便使 DLL 的函数为进程所用。Windows 2000 内核的上层依赖于 HAL 接口，而不是底层硬件，从而进一步提高了可移植性。HAL 直接操作硬件，从而使得 Windows 2000 的其他部分与硬件差异相分开。

可靠性是指处理出错情况的能力，包括操作系统自我保护和使其用户免受问题软件和恶意软件的攻击。Windows 2000 通过对虚拟内存实行硬件保护和对操作系统资源实行软件保护的保护机制来达到抵制故障和攻击的目的。Windows 2000 还有一种本地文件系统，即 NTFS 文件系统，它能在系统崩溃之后自动从多种文件系统错误中恢复。Windows NT 4.0 版获得美国政府的 C－2 安全认证，在防止问题软件攻击级别中属中等层次。当前

Windows 2000 也正在接受政府的这项认证评估。要了解更多的关于安全分类的信息请参见 19.8 节。

Windows 2000 对符合 IEEE 1003.1(POSIX)标准的应用程序提供资源层兼容。这样，无需修改源代码就能使它们在 Windows 2000 上编译运行。另外，通过运行以前提到的环境子系统，Windows 2000 能运行许多已编译程序的可执行二进制程序，这些编译后的程序运行在 Intel X86 体系结构上的 MS-DOS、16 位 Windows、OS/2、LAN Manager 和 32 位 Windows 系统。这些环境子系统支持多种文件系统，包括 MS-DOS FAT 文件系统、OS/2 的 HPFS 文件系统、ISO9660 CD 文件系统以及 NTFS 文件系统。然而，Windows 2000 的二进制兼容性并不十分完美。例如，在 MS-DOS 中，应用程序能直接访问硬件端口。因为考虑到可靠性和安全性的因素，Windows 2000 禁止这样的访问。

Windows 2000 的设计能提供良好的**性能**。组成 Windows 2000 的子系统间通过本地过程调用(简称 LPC)进行有效通信，LPC 能提供高性能的信息传递。Windows 2000 子系统的非内核线程可以被优先级更高的线程抢占。这样，系统可以快速响应外部事件。另外，Windows 2000 设计有对称多处理技术：在一台多处理器计算机上，多个线程能同时运行。与 UNIX 相比，当前 Windows 2000 的规模有限。自 2000 年末开始，Windows 2000 支持 32 个 CPU，而 Solaris 达到 64 个。先前的 NT 版本只支持 8 个处理器。

Windows 2000 设计成**国际使用**，它通过**国家语言支持**(national language support，NLS)**API** 提供不同的地域支持。NLS API 提供专门的程序对日期、时间和各国货币进行格式化。专门的字符串比较用于解释不同的字符集。Windows 2000 使用 UNICODE 字符编码；Windows 2000 通过在操作前把字符转换成 UNICODE 的方式(8 位到 16 位的转换)支持 ANSI 字符。

21.3 系统组成

Windows 2000 结构是分层模块化的结构，如图 21.1 所示。主要层次包括 HAL、内核和可执行体，这些均运行于保护模式下；及一组运行于用户模式下的子系统。用户模式子系统分成两类。一类为环境子系统，以模拟不同操作系统；另一类为**保护子系统**，以提供安全功能。这种结构的主要优点是模块之间的互操作简单。本节将讨论这些层次和子系统。

21.3.1 硬件抽象层

硬件抽象层(hardware-abstraction layer，HAL)是一软件层，用来为操作系统的上层隐藏硬件差异，以提高 Windows 2000 的可移植性。HAL 有一个可为内核调度程序、可执行体和设计驱动程序所使用的虚拟机接口。这种方法的一个优点是每个设备驱动程序只需要一个版本，它可运行于各种硬件平台，而无需移植驱动程序。同时 HAL 支持对称处理。为了

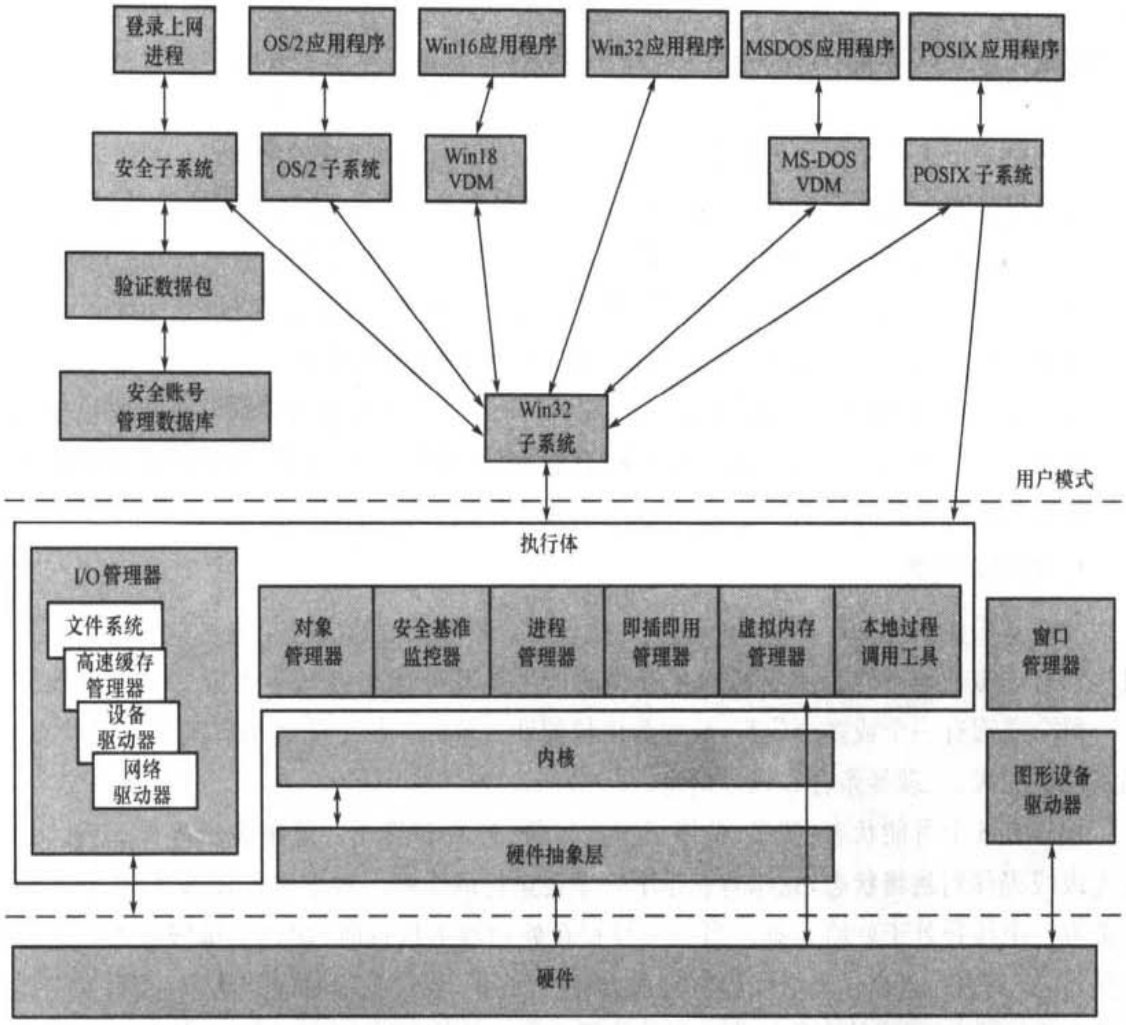


图 21.1 Windows 2000 模块化结构图

提高性能，I/O 驱动器(还有 Windows 2000 的图形驱动器)可以直接访问硬件。

21.3.2 内核

Windows 2000 的内核为执行体和子系统提供了基础。内核驻留于内存，其执行不会被抢占。它有四个主要的责任：线程调度，中断和异常处理程序，低层处理器同步和掉电后的恢复。

内核是面向对象的。Windows 2000 的**对象类型**是系统定义的数据类型，它具有一定的属性(数据值)和一组方法(例如，函数或操作)。**对象**是对象类型的实例。内核通过使用一组对象(属性存储内核数据而方法执行内核活动)执行其工作。

内核使用两组对象。第一组由调度程序对象组成。**调度程序对象**(dispatcher object)控制系统的调度和同步。这些对象包括事件、mutant、mutex、信号量、线程和定时器。**事件对象**用来记录事件的发生，并使事件与某一动作同步。对象 **mutant** 规定了在拥有者概念下的

内核模式和用户模式的互斥。对象 **mutex** 提供了只能在内核模式下使用的无死锁互斥。**信号量对象**作为计数器或门，控制访问某个资源的线程数量。**线程对象**是由内核调度程序调度的实体，它与进程对象相关联(进程对象有虚拟地址空间)。**定时器对象**用来跟踪时间，发出时间到信号以表示操作过长或需要中断或需要调度一个周期性的活动。

第二组内核对象由**控制对象**(control object)组成。这些对象包括非同步的过程调用、中断、电源通知、电源状态、进程和 profile 对象。系统使用非同步程序调用(asynchronous procedure call, APC)中断正在运行的线程并调用一个程序。**中断对象**(interrupt object)把中断服务程序与中断资源绑定；在掉电后，系统使用**电源通知对象**(power-notify object)自动调用一个程序，**电源状态对象**(power-status object)检查电源是否错误。进程对象描述虚拟地址空间和需要执行与某一进程相关联的线程组的控制信息；最后，系统使用 **profile 对象**来测量代码块所耗费的时间。

1. 线程与调度

与许多操作系统一样，Windows 2000 也使用进程和线程概念来描述可执行代码。每个进程都有虚拟地址空间和用于初始化进程的信息如基础优先级和单个或多个处理器的亲合。每个进程有一个或多个线程，线程为内核的调度单元。每个线程都有其自己的状态，包括实际优先级、处理器亲合和账号信息。

线程有 6 个可能状态：就绪、备用、运行、等待、过渡和终止。**就绪**表示等待运行。最高优先级线程移到**就绪状态**，意味着它是下一个要运行的线程。对于多处理器系统，每个处理器都有一个线程处于就绪状态。当一个线程在处理器上执行时，就处于**运行状态**。它会一直运行，直到它为更高优先级线程所抢占，或线程终止，或其分配时间片结束，或阻塞于调度对象如表示 I/O 完成的事件。当线程等待将要发出的信号时(比如 I/O 系统任务完成时)，就处于**等待状态**。新线程等待执行所需资源时，就处于**过渡状态**。当线程完成执行时，就进入了**终止状态**。

调度程序采用了 32 级优先权方案以确定线程执行顺序。优先级分成两类：可变类和实时类。可变类包括 0 到 15 优先级的线程，实时类包括 16 到 31 优先级的线程。调度程序为每个优先级都采用了队列，并从高到低遍历队列集合，直到找到可以运行的线程为止。如果某个线程具有特定处理器亲合而该处理器不可用，那么调度程序跳过它，继续查找能在可用处理器上运行的线程。如果找不到就绪线程，那么调度程序就执行一个称为空闲线程的特殊线程。

当线程的时间片用完时，该线程就被中断；如果被抢占线程属于可变优先级类，那么其优先级会降低。但优先权不会低于基础优先级。降低线程优先级往往能限制计算约束(compute-bound)线程的 CPU 消耗。当可变优先级线程从等待状态释放出来时，调度程序会加大其优先级。加大的量取决于线程所等待的设备；例如，等待键盘的线程会得到较大程度的增加；而等待磁盘的线程会得到中等程度的增加。这种策略能够给予使用鼠标和窗口

的交互线程更高的优先级，从而能让 I/O 为主（I/O bound）线程保证 I/O 设备一直忙，也能让计算约束线程使用后台的空闲 CPU 周期。这种策略为许多分时操作系统如 UNIX 所使用。另外，与用户活动 GUI 窗口相关的线程会得到优先级提升，从而增加其响应速度。

当线程进入以下状态就会发生调度：就绪或等待状态，线程终止状态，应用程序改变线程的优先级以及处理器亲合。如果在较低优先级线程运行时有一个更高优先级的实时线程进入就绪状态，那么较低优先级线程就被抢占。在需要时，这种抢占能给予实时线程更高优先级以使用 CPU。Windows 2000 不是实时操作系统，因为它不能保证实时线程在某一特定时间限制内可以开始执行。

2. 异常与中断

内核调度程序也为由硬件或软件所产生的异常和中断提供了陷阱处理。Windows 2000 提供了多个与体系结构无关的异常，包括：内存访问违例、整数上溢、浮点数上溢或下溢、整数被 0 除、浮点被 0 除、非法指令、数据不对齐、特权指令、读页错误、访问违例、超出换页文件量、调试中断点和单步调试。

陷阱处理能处理简单异常。更为复杂的异常处理是由内核异常调度程序所执行的。**异常调度程序**创建一个包括异常理由的异常记录，并查找一个异常处理程序处理它。

当异常在内核模式下发生时，异常调度程序只不过简单地调用一个程序以定位异常处理程序。如果没有找到，那么就出现了一个致命系统错误，这时用户会看到声名狼藉的"蓝屏死"（表示系统出错）。

对于用户模式进程，异常处理就更为复杂，因为环境子系统（例如 POSIX 系统）会为由它所创建的每个进程设置一个调试端口和异常端口。如果调试端口已注册，那么异常处理程序会向该端口发送一个异常。如果调试端口没有找到或者不处理所收到的异常，那么调度程序试图找到一个合适的异常处理程序。如果找不到一个异常处理程序，那么调试程序会再次用来捕捉调试错误。如果调试程序没有运行，那么会向进程的异常端口发送一个消息，以便让环境子系统能有机会转换这一异常。例如，POSIX 环境将 Windows 2000 异常消息转换成 POSIX 信号，再发送给引起异常的线程。最后，如果这些都不行，那么内核就简单地终止包含有引起异常的线程的进程。

内核的中断调度程序通过调用中断处理程序 ISR（比如由调备驱动程序所提供）或内核处理子程序处理中断。中断由中断对象表示，中断对象包括处理中断所需要的所有信息。采用中断对象便于让中断处理程序与中断相关联，而不需要直接访问中断硬件。

不同处理器结构，如 Intel 或 DEC Alpha，有不同类型和数量的中断。为了可移植性，中断调度程序将这些硬件中断映射到一个标准集合。中断根据优先级高低划分并按优先级顺序处理。Windows 2000 有 32 个中断级别（IRQL，Interrupt ReQuest Level）。其中 8 个为内核所保留使用；其他 24 个通过 HAL 表示硬件中断（虽然绝大多数 x86 系统只用了 16 个）。Windows 2000 中断按图 21.2 来定义。内核使用中断调度表（interrupt dispatch

table)将一个中断级别绑定到一个服务例程。在多处理器计算机上,Windows 2000 为每个处理器保留了独立的**中断调度表**,每个处理器的 IRQL 可以独立设置以屏蔽中断。所有等于或低于处理器 IRQL 的中断会阻塞,直到 IRQL 被内核级线程降低。Windows 2000 充分利用了这一属性采用软件中断来执行系统功能如分派线程调度、处理定时器、支持异步操作。

中断级别	中断类型
31	机器检验或总线出错
30	电源出错
29	处理器间通知(请求另外一个处理器执行;例如,分派一个进程或更新 TLB)
28	时钟(用于跟踪时间)
27	简档
3—26	传统 PC IRQ 硬件中断
2	分派和延迟的过程调用(DPC)(内核)
1	异步过程调用(APC)
0	被动

图 21.2 Windows 2000 的中断请求级别

内核使用调度中断来控制线程关联切换。当内核处于运行状态时,将处理器的 IRQL 级别提高到高于调度程序的级别。当内核确定需要一次线程调度时,内核就产生一次调度中断,但该中断在内核完成现有工作并降低 IRQL 之前被阻塞。此时,调度程序就会选择运行一个线程处理调度中断。

当内核最终决定必须执行某些系统函数时(并不是马上),它将排列**延迟过程调用**(deferred procedure call,DPC)对象所包含的函数地址,并产生一个 DPC 中断。当进程的 IRQL 足够低时,就执行 DPC 对象。DPC 中断的 IRQL 通常都高于用户线程的 IRQL,因此 DPC 会中断用户线程的执行。为了避免问题的出现,DPC 受到限制,因而相当简单。它们不能修改一个线程的存储,不能新建、获取或等待对象;不能调用系统服务程序或者是产生页错误。

3. 低层处理器同步

内核的第三个职责是提供低层处理器同步。APC 机制与 DPC 制相似,但是更为通用。APC 制使得进程得以建立一个过程调用来防范将来可能发生的崩溃。例如,许多系统服务都能以参数形式接受用户模式的程序。一个用户线程将使用非同步系统调用并提供一个 APC,而不调用同步系统调用(同步系统调用将阻塞线程,直到完成系统调用)。当系统服务程序结束时,用户线程就被中断并自发地去运行 APC。

虽然只在线程宣布自己处于**警报**(alertable)状态时才执行用户模式的 APC,但 APC 可以放入系统线程队列和用户线程队列中的任何一个。APC 的功能比 DPC 强大,因为 APC 能获取和等待对象,产生分页错误和调用系统服务。因为 APC 在对象线程的地址空间中运行,所以 Windows 2000 执行体广泛使用 APC 进行输入/输出处理。

Windows 2000 可以在对称多处理机上运行，因此内核必须防止两个线程同时修改共享的数据结构。内核使用驻留在全局存储器当中的 spinlock 来实现多处理机之间的互斥。如果一个进程试图获取一个 spinlock 时，那么处理器中所有的活动都将停止，并且拥有 spinlock 的线程不能被抢占。所以进程可以尽快地结束并释放该锁。

4. 掉电后恢复

内核的第四个也是最后一个职责就是提供掉电后恢复功能。拥有次高级优先权的掉电中断，一旦发现掉电便通知操作系统。电源通知对象为设备驱动器提供了注册程序的方法，它将在电源恢复时被调用。该对象确认该设备已设置成恢复时的正确状态。对电源备份系统来说，电源状态对象很有用。在驱动程序进行临界操作之前，它会检查电源状态对象以确定电源有没有发生故障。如果驱动程序确定电源没有故障，它将增加该处理机的 IRQL 直到电源出现故障，完成这个操作后，就重新设置 IRQL。这一系列的行为阻塞了电源故障中断，直到临界操作结束。

21.3.3 执行体

Windows 2000 执行体提供一组环境子系统所使用的服务。服务可分成如下几组：对象管理器、虚拟存储器管理器、进程管理器、本地程序调用工具、I/O 管理器、安全引用监视器。

1. 对象管理器

Windows 2000 作为一个面向对象的系统对其所有的服务程序及实体都使用对象。对象的例子包括目录对象、符号连接对象、信号量对象、事件对象、进程和线程对象、端口对象以及文件对象。**对象管理器**(object manager)的工作就是监督管理所有对象的使用。当一个线程想要使用一个对象时，它就调用对象管理器的 open 方法，取得一个对象句柄。**句柄**(handle)是所有对象类型的标准化接口。就像文件句柄，对象句柄是一个进程惟一的标识符，利用它可以访问及操作一个系统资源。

因为对象管理器是生成对象句柄的惟一实体，所以理所当然地在此检查安全性。例如，当某个进程试图打开一个对象时，对象管理器就要检查这个进程是否有权访问此对象。对象管理器也可以增加配额，例如一个进程最多可以分配的内存。

对象管理器可以跟踪进程使用的每个对象。在每个对象的头部包含拥有操作该对象句柄的进程计数。如果是一个临时对象，那么当计数器等于零时，管理器将从名称空间中删除该对象。Windows 2000 经常使用指针(而不是句柄)来访问各种对象，对象管理器也包含了一个引用计数，当 Windows 2000 访问对象时该计数就递增，不需要访问时该计数就递减。当一个临时对象的引用计数趋于零时，就从内存中删除该对象。固定对象表示物理实体，如磁盘驱动器，当参考计数和开发句柄计数器趋于零时，不会被删除。

操作对象有一组标准方法：create，open，close，delete，query name，parse 和 security。现在解释一下后面的三个方法。

- 当线程拥有某一对象的句柄，但需要知道该对象的名称时就调用 query name 这一方法。
- 对象管理器利用 parse 方法来查找给定对象名字的对象。
- 当进程打开或者是改变对象的保护时需要调用 security 方法。

Windows 2000 执行体允许任何对象都拥有**名称**(name)。名称空间是全局的，因此一个进程可能创建一个已命名的对象，而另一个进程会打开该对象的一个句柄，并与该进程共享。打开一个已命名对象的进程可以请求查找，可以区分大小写，也可不区分大小写。

名称可以是永久的也可以是临时的。永久名称表示一个实体，如磁盘驱动器，甚至在没有进程访问的情况下也会保留下来。暂时名称只在进程拥有对象的句柄时才存在。

虽然名称空间在网络上并不是直接看得见的，但是对象管理器的解析方法可以用于帮助访问另外系统的已命名对象。当进程试图打开一个驻留在远程计算机上的对象时，对象管理器调用 parse 方法，然后该方法就会请求一个网络重定向器来查找对象。

对象名称的结构如 MS-DOS 和 UNIX 中的文件路径名。**目录对象**(directory object)包含该目录中所有对象的名字，目录就是由目录对象表示的。对象名字空间会随着**对象定义域**(object domain)的增加而增大，对象定义域是独立的对象组合。对象域的典型例子就是软磁盘和硬盘驱动器。当软盘插入到系统中就很容易明白名字空间是怎样扩展的。软盘拥有自己的名字空间，它与现存的名字空间相连接。

UNIX 文件系统有**符号连接**(symbolic link)，因此多个别名或者昵称可以指向同一个文件。与之相似的是，Windows 2000 也执行**符号连接对象**(symbolic link object)。Windows 2000 应用符号连接的一种方法把驱动器名映射为标准的 MS-DOS 驱动器字符。驱动器字符只是符号连接，可以根据用户的偏好重新映射。

进程取得句柄可以通过创建一个对象，或者通过打开现存的对象，或者通过接收来自另外一个进程的复制句柄，或者通过从其父进程中继承句柄，这种方法与 UNIX 进程取得文件描述符很相似。这些进程保存于进程的**对象表**，每个对象表项含有对象访问权并说明是否需要由**子进程**继承。当进程停止时，Windows 2000 自动关闭所有进程已打开的句柄。

当登录进程验证一个用户时，也为用户进程附加了访问标记。该访问标记包括各种信息如安全 ID、组 ID、特权、主组和缺省访问控制链表。用户可访问的服务和对象是由这些属性决定的。

在 Windows 2000 中，每个对象都用访问控制列表(包括安全 ID 和允许访问权限)加以保护。当一个线程试图访问一个对象时，系统将线程访问标记的安全 ID 与对象的访问控制列表进行比较，以确定是否允许该访问。Windows 2000 的内部服务程序使用的是指针绕过访问检查，而不是打开对象句柄。

总之，对象的创建者决定适合该对象的访问控制列表。如果没有明确的对象，将从创建者对象中继承一个，或者是从用户的访问标准对象中选取一个故障表。

访问标志里有一个字段控制审核对象。正在审核的操作带着用户标识被记录到系统的审

核日志中，审计字段可以监视该条记录，从而能发现入侵系统或者是访问受保护对象的企图。

2. 虚拟内存管理器

Windows 2000 执行体的虚拟内存部分就是**虚拟内存**(VM)**管理器**(virtual-memory manager)。VM 管理器的设计假定底层硬件支持虚拟到物理的映射、调页机制、多处理器系统的透明的缓存一致性，允许多个页表条目映射到同一物理帧。Windows 2000 的 VM 管理器应用一个基于页面的管理方案，该方案采用 4 KB 大小的页面。赋给某一进程但不在物理存储器中的数据页面被保持在磁盘上的**页面调度文件**(paging file)中。

VM 管理器使用的是 32 位地址，因此每个进程拥有一个 4 GB 的虚拟地址空间。高地址的 2 GB 对于所有进程都是一样的，Windows 2000 只在内核模式下使用这 2 GB，低地址的 2 GB 对于每个进程是不同的，用户模式的线程和内核模式的线程都可以访问。要指明的是，Windows 2000 的特定配置只有 1 GB 留给操作系统专用，允许进程使用 3 GB 的地址空间。

Windows 2000 VM 管理器采用了两步过程来分配用户内存。第一步保留进程虚拟地址空间的一部分。第二步通过指派虚拟内存空间(物理内存或调页文件的空间)来提交分配。Windows 2000 通过对提交内存加上限额以限制进程所使用的虚拟内存的大小。当进程不再使用而释放虚拟内存时，进程会解除内存。因为内存是由对象表示的，当一个进程(父进程)创建另一个进程(子进程)时，父进程能够访问子进程的虚拟内存。环境子系统就是这样管理它们客户进程的存储器的。在性能方面，VM 管理器允许具有优先权的进程在物理存储器中锁定挑选过的页面，这样确保了该页面不会被换到页面调度文件中去。

两个进程之间可以通过同一内存对象的句柄共享内存，但是这种方法效率很差，因为在每个进程访问该对象之前必须提交对象的整个内存空间。Windows 2000 提供了另一种办法，称为**地域对象**(section object)，以此表示共享内存的一个块。取得一个地域对象的句柄后，进程就可以映射内存所需要的部分，这个部分叫做**视图**(view)。在视图机制中，进程能够访问一个因过大而不适合进程页面调度文件规模的对象。系统可以使用视图来访问对象的地址空间，每次一个对象。

进程可按许多方式控制共享内存段对象的使用。共享段的最大值可以加以约束。共享段可通过磁盘空间加以扩展如系统调页文件或普通文件(**内存映射文件**)。共享段可以有个基础，即在所有试图访问它的进程中，以同一虚拟地址出现。最后，共享段中的保护页可以设成只读、读写、读写执行、只可执行、无访问、写时拷贝。最后两个保护设置需要加以解释。

- 访问一个保护页面会导致异常，但也可以利用异常，举个例子，可用来检查出错程序是否在阵列的末端之外循环。
- 写时拷贝机制为 VM 管理器增加了物理内存的使用效率。当两个进程需要各自的对象拷贝时，VM 管理器将一个共享拷贝映射到虚拟内存并激活内存区域的写时拷贝属性。如果一个进程试图修改写时拷贝页中的数据，那么 VM 管理器就为该进程复制一个私有拷贝。

Windows 2000 中虚拟地址转换用到多种数据结构。每个进程拥有一个**页面目录**

(page-directory),它包含了 1 024 个、大小为 4 B 的**页目录条目**(page-directory entry)。很有代表性的是,页面目录是专用的,但是情况需要时,也可以在进程间共享。每个页面目录项指向一个含有 1 024 个**页表项**(PTE)的页表,每个页面表项的大小为 4 B。每个 PTE 指向物理内存中的一个 4 KB 的页帧(page frame)。一个进程所有页面表的大小总额达到 4 MB,因此 VM 管理器在需要时将把这些表交换到磁盘。请参见图 21.3 中此结构的图表。

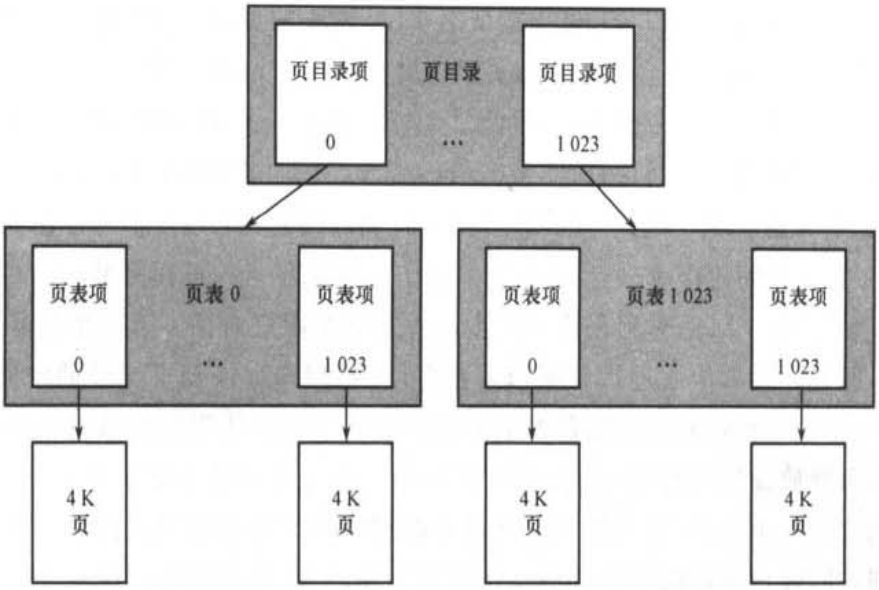


图 21.3 虚拟内存层次结构

10 位值可能表示 0 到 1 023 的所有值。因此,10 位值可以选择页目录或页表的任一条目。这个属性用于将虚拟地址指针转换成物理内存的字节地址。32 位虚拟内存地址分成三个部分,如图 21.4 所示。虚拟地址的头 10 位用低页目录的索引。这一地址选择一个页目录条目(PDE),它包括了页表的物理页帧。内存管理单元(MMU)采用了虚拟地址的后 10 位,方便从页表中选择一个 PTE。该 PTE 指定了物理内存的页帧。虚拟地址的剩余 12 位作为页帧中特定字节的偏移。MMU 通过将 PTE 的 20 位与虚拟内存的低 12 位相合并,就得到了一个指针以指向物理内存的特定字节。因此,32 位 PTE 有 12 位可用来描述物理页的状态。奔腾 PTE 保留 3 位专供操作系统使用。剩下的位描述页面是否脏的、访问过的、只读的、写通、内核模式或者是有效的;这样它们在存储器中描述页面状态。关于页面调度方案的所有内容请参见 9.4 节。

物理页可以有 6 个状态:有效、空闲、清零、备用、修改、坏和过渡。*有效*页为活动进程所使用。*空闲*页是没有被 PTE 引用的页。*清零*页是已清零的、可满足调页清零需要的空闲页。*修改*页则已经被进程所写,在分配给其他进程之前需要发送到磁盘。*备用*页上的信息已经保存在磁盘上。这些可能是没有修改过的页,已经写到磁盘上的修改过的页,或为改善局部性而提前读入的页。最后是*坏*页,其不可用是因为硬件已检测到错误。

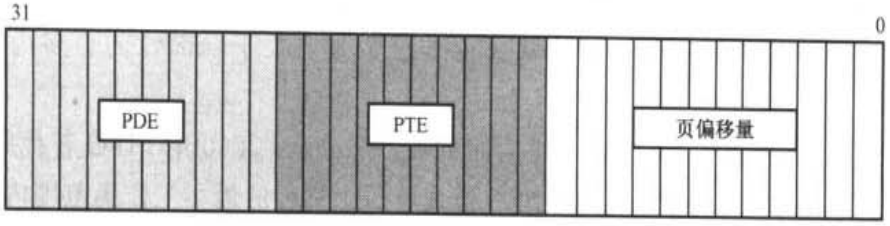


图 21.4 虚拟地址到物理地址之间的转换

页文件 PTE 的真正结构如图 21.5 所示。PTE 包括 5 位用于页保护、20 位用于页文件偏移、4 位用于选择调页文件和 3 位以描述页状态。页文件 PTE 标记成对 MMU 为无效虚拟地址。因为可执行代码和内存映射文件已有一个磁盘拷贝，所以它们在调页文件中不需要空间。如果这样的一个页不在物理内存中，那么 PTE 结构就会按如下所述：最重要位用于表示页保护，后 28 个位用于索引系统数据结构以表示一个文件和在这个文件中的页偏移，低 3 位表示页状态。

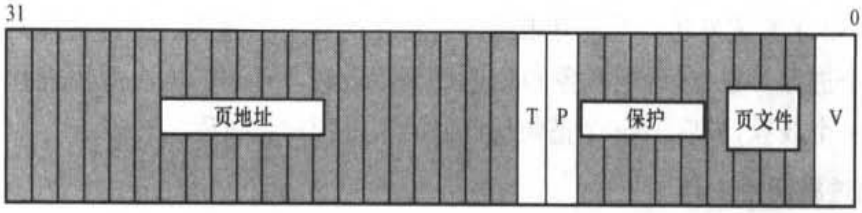


图 21.5 页文件页表项

如果每个进程拥有各自所有的一组页表，那么在进程间共享一个页面是非常困难的，因为每个进程拥有它自己的、适合页帧的 PTE。当一个共享页面在物理存储器中出现错误时，就不得不在 PTE 里保存物理地址，而 PTE 是属于共享页面的每个进程的。在这些 PTE 中的保护位和页面状态位都需要经常的设置与更新。为避免这些问题的出现，Windows 2000 使用了一个间接的办法。因为每个页面都是共享的，进程都有一个指向**原型页表项**，而不是指向页帧的 PTE。原型页表项含有页帧地址和保护位以及状态位。这样，进程第一次访问共享页面就产生一次页错误。第一次访问之后，就可以用正常的方式执行更多次的访问。如果页面有只读的标记，VM 管理器执行一次写拷贝，而进程就不再有效拥有页面。共享页面不会出现在页面文件中，但是可以在文件系统中查找到。

VM 管理器跟踪**页帧数据库**(page-frame database)内的所有物理内存的帧。系统内的每个物理内存帧都有一个条目。条目指向 PTE，而 PTE 又指向帧，这样 VM 管理器可维护页的状态。未被有效 PTE 所引用的帧根据页类型如清零、修改或空闲形成链表。

页错误发生时，VM 管理器页面丢失而发生错误，并把该页面放入空闲表的第一个帧。但它并不能停留在那里，研究发现内存引用线程趋向于拥有一个**局部性**特性：当一个页面正在使用时，相邻的页面很可能在不久的将来就被使用。(考虑到阵列循环，取出形成适合线

程的可执行代码的顺序指令。)因为局部性,VM 管理器导致了某一页错误,也将导致相邻的几个页面的故障。相邻的故障会减少页错误的总数量。关于局部性的更多内容,请参见 10.6.1 小节。

如果空闲页面表(清单)上没有可使用的页帧,Windows 2000 用 FIFO 替代原则以取回超过其最小工作集合的页。Windows 2000 监视着每个进程的最小工作集范围内的故障,并相应地调整工作集的大小。值得注意的是,Windows 2000 下启动某一进程时,就赋予该进程一个 30 个页面大小的故障工作集。Windows 2000 运用从进程中窃取有效页面的办法对该页面的大小进行周期性的测试。如果在不产生页错误进行窃取页面的话,进程继续执行,进程工作集就会减少 1,并把该页面加入到空闲表中。

3. 进程管理器

Windows 2000 进程管理器提供了创建、删除和使用进程、线程和作业的服务。它没有关于进程的父子关系和层次关系的概念;这些细节由拥有这些进程的环境子系统处理。

在 Win32 环境中,进程创建的一个例子如下。当 Win32 应用程序调用 CreateProcess 时,就向 Win32 子系统发送一个消息来通知它正在创建一个进程。进程管理器调用对象管理器新建一个进程对象,然后把对象句柄返回到 Win32, Win32 再次调用进程管理器重新为进程创建一个线程,最后 Win32 把句柄返回给新进程和新线程。

4. 本地过程调用工具

在单个机器上的客户与服务进程之间,操作系统采用 LPC 工具以传递请求和结果。尤其是,LPC 用来从各种 Windows 2000 子系统中请求服务。LPC 在许多方面类似于 RPC 机制(RPC 机制为许多操作系统用来通过网络进行分布处理),但是 LPC 为单个系统使用而进行了优化。

LPC 是消息传递机制。服务器进程发布一个全局可见的连接端口对象。当一个客户需要子系统的服务时,它便打开一个子系统连接端口对象的一个句柄,并对该端口发送一个连接请求。服务器建立一个通道,并返回一个句柄给客户。该通道有一对私有通信端口:其中一个用于服务器到客户消息,另一个用于客户机到服务器消息。通信频道支持回调机制,这样客户机和服务器在等待回答时也可接受请求。

建立一个 LPC 通道时,必须指定下面三种信息传递方法。

1. 第一个技术适用于小消息(不到数百字节)。在这种情况下,端口消息队列用做中间存储,消息从一个进程复制到另一个进程。

2. 第二个技术用于更大消息。在这种情况下,每个通道要创建一个共享内存区间对象。通过消息队列端口发送的消息(包括指针和大小信息)来引用区间对象。这避免了复制大消息。发送者将消息放入共享区间、接收者可直接看到。

3. 第三种办法称为**快速 LPC**,用于 Win32 子系统的图形显示部分。客户机发出要求使用快速 LPC 的请求时,服务器就建立三个对象:处理请求的专用服务器线程,64 KB 的共享

段对象以及事件对对象。*事件对对象*是一个同步对象，客户线程向 Win32 服务器拷贝完一条信息之后，Win32 子系统可用它来提供通知，反之亦然。LPC 的传输是在区域对象中进行的，而同步则是通过事件对对象实现的。LPC 有许多的优点。区域对象可以删除信息拷贝，因此它代表一个共享存储器的区域。事件对对象取消了采用端口对象传递包括指针和长度的消息的额外开销。专门服务器线程取消确定哪个客户线程调用服务器的额外开销，这是由于每个客户线程只有一个服务器线程。最后，内核给予这些专门服务器线程优先调度以改善性能。但它的缺点是，快速 LPC 比使用其他两种方法占用了更多的资源，因此 Win32 子系统使用快速 LPC 只适用于 Window 管理器和图形设备接口。

5. I/O 管理器

I/O 管理器(I/O manager)负责文件系统、设备驱动程序和网络驱动程序。它跟踪装入了何种设备驱动程序、过滤驱动程序和文件系统，也管理 I/O 请求的缓冲。它与 VM 管理器一起提供内存映射文件 I/O，控制 Windows 2000 的缓存管理器，从而处理整个 I/O 系统的缓存。I/O 管理器支持同步与异步的操作，规定了驱动器的间歇时间，也规定了一个驱动器访问另一个驱动器的机制。

I/O 管理器能把接收到的请求转换为一种标准格式，称为 **I/O 请求包**(IRP)。然后把 IRP 交给正确的驱动器进行处理。当操作完成时，I/O 管理器从最近执行过的操作驱动器中接收到 IRP 后就完成此请求。

在许多操作系统中，高速缓存是由文件系统执行的。然而，Windows 2000 却规定了一个集中式的高速缓存工具。高速缓存管理器规定的高速缓存服务程序适合 I/O 管理器控制下的所有组件，并且与 VM 管理器紧密合作。根据系统中可使用的空闲内存的多少，高速缓存的大小始终处于动态的变化当中。再次调用 2 GB 的地址空间组成了系统区域供所有进程使用。VM 管理器把超过一半的空间分配给了系统的高速缓存器。高速缓存管理器把文件映射到这个地址空间，并占用了 VM 管理器的容量来处理文件 I/O。

缓存按 256 KB 的块来划分。每个缓存块可以容纳文件的一个视图(即内存映射区域)。每个缓存块按 VACB 来描述，**VACB**(virtual-address control block)存储了视图的虚拟地址和文件偏移，及使用视图的进程数量。VACB 驻留在由缓存管理器维护的一个数组中。

对于每个打开文件，缓存管理器维护一个独立 VACB 索引数组以描述整个文件的缓存。这一数组对每个 256 KB 的文件块都有一个条目；这样，一个 2 MB 的小文件就可能有一个 8 条目的 VACB 索引数组。如果那部分文件存于缓存中，那么 VACB 索引数组的条目指向 VACB；否则，就为空。

当 I/O 管理器收到一个文件的用户级读请求时，I/O 管理器就发送一个 IRP 给设备驱动程序堆栈(文件所驻留的)。文件系统试图查找缓存管理器所请求的数据(除非请求显式要求非缓存的读)。缓存管理器计算哪个文件的 VACB 索引数组的条目对应于请求的字节偏移。该条目或指向缓存的视图，或无效。如果有效，那么缓存管理器分配一个缓存块(和

VACB 数组的相应条目),并将该视图映射到缓存块中。缓存管理器接着试图将映射文件的数据复制到调用者的缓存。如果拷贝成功,那么操作就完成。如果拷贝失败(由于页出错),那么就会引起 VM 管理器向 I/O 管理器发送一个非缓存的读请求。I/O 管理器沿着驱动程序堆栈向下发送另一请求,这次请求*调页*操作迂回缓存管理器,且从文件直接读入数据并存入为缓存管理器所分配的页。在完成之后,VACB 就指向该页。现存于缓存的数据被复制到调用者缓冲,原来 I/O 请求就完成了。图 21.6 显示了这些操作的概图。当可能时,对缓存文件的同步操作、缓冲、无锁定 I/O 以及 I/O 都按照**快 I/O 机制**(fast I/O mechanism)来处理。该机制只是把数据直接拷贝到 cache 页面(或者是从 cache 页面直接拷贝)并利用 cache 管理器来实现所需求的 I/O 。

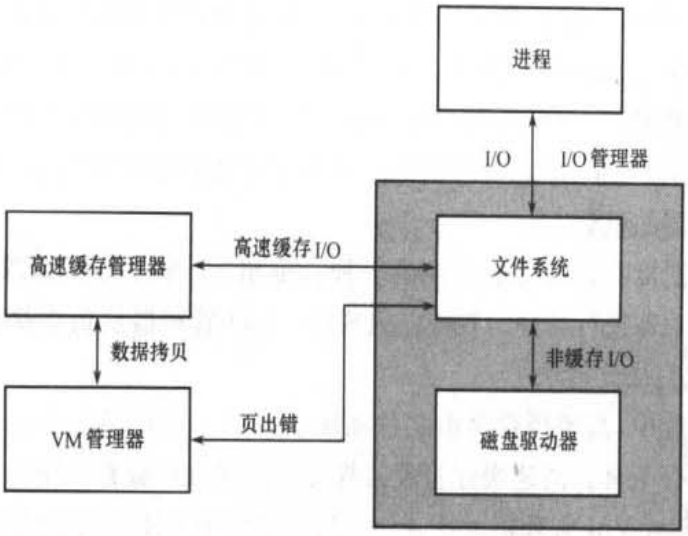


图 21.6 文件 I/O

内核级的读操作和用户级差不多,只不过数据可直接在缓存中处理,而不必复制到用户空间的缓冲中。为使用文件系统元数据(描述文件系统的数据结构),内核使用缓存管理器的映射接口读入元数据。为了修改元数据,文件系统使用缓存管理器的钉住接口。**钉住**(pinning)一页会将该页锁在物理内存帧中,这样 VM 管理器就不能移动或调出该页。在更新元数据之后,文件系统会请求缓存管理不再钉住页。修改页标记为脏页,以便 VM 管理器将该页冲刷到磁盘。元数据保存在普通文件中。

为了改善性能,缓存管理保留一个读请求的较小历史记录,并尝试预测将来的请求。如果缓存管理器发现了以前三个读请求的模式,如顺序向前或向后访问,就提前将数据读入到缓存(在应用程序下个请求前)。这样,应用程序会发现其数据已在缓存中,而不必等待磁盘 I/O 完成。Win32 API 中的 OpenFile 和 CreateFile 函数可以传递参数 FILE_FLAG_SEQUENTIAL_SCAN 标记,这是一个对缓存管理器的提示以便提前访问 192 KB。通常,Windows 2000 按 64 KB 或者 16 页执行 I/O 操作;因此,这一提前读是普通量的三倍。

缓存管理器还负责通知 VM 管理器以冲刷缓存的内容。缓存管理器缺省行为是回写缓存:它先将操作进行积累,然后每 4、5 秒唤起写线程。当需要直写缓存,进程在打开文件时可设置标记,或者进程可调用显式缓存刷新函数。

快写进程可能会用完所有空闲页,而此时缓存写线程还未被唤醒将页刷新到磁盘。缓存写通过如下方法以防止进程用淹没系统。当空闲缓存内存量变低时,缓存管理器会暂时阻塞进程试图写数据,并唤起缓存写线程以将页刷新到磁盘。如果快写进程实际是个网络文件系统的网络重定向器,阻塞太久会引起网络传输超时重传。这种重传浪费网络带宽。为防止这种浪费,网络重定向器可告诉缓存管理器不要在缓存中积累大块的数据。

因为网络系统需要在磁盘和网络接口之间移动数据,缓存管理器还提供了 DMA 接口用于直接移动数据。直接移动数据避免了通过中间缓存的数据复制。

6. 安全引用监控器

Windows 2000 面向对象的性质有助于采用统一机制,对所有用户可访问的实体进行运行时访问确认和审计检查。当进程打开对象句柄时,**安全引用监控器**(Security Reference Monitor,SRM)就检查进程安全标记和对象访问控制链表,从而确定进程是否有必需的权限。

7. 即插即用(PnP)管理器

操作系统使用**即插即用管理器**(plug and-play manager, PnP)识别和适应硬件配置的变化。为了应用 PnP 技术,设备及其驱动程序都必须支持 PnP 标准。在系统运行时,PnP 管理器自动识别安装设备并检测设备的变化。它会跟踪设备所使用的资源及其可能要使用的资源,也负责装入适当驱动程序。硬件资源(主要为中断和 I/O 内存区域)管理的目的之一是确定一个硬件配置,以便所有设备都能正常工作。例如,如果设备 B 可能使用中断 5 而设备 A 可使用 5 或 7,那么 PnP 管理将 5 赋给 B 而将 7 赋给 A。对以前版本,用户可能需要撤下设备 A 并重新配置使用中断 7,然后再安装设备 B。因此,在安装新硬件之前,用户必须研究系统资源,查找或记住哪些设备使用哪些硬件资源。PCMCIA 卡和 USB 设备的大量流行也要求支持动态资源配置。

PnP 管理器按如下方式处理这一动态配置。首先,它从总线驱动程序那里获得设备列表(如 PCI、USB)。它装入已安装的驱动程序(如果需要可安装),并对每个设备的适当驱动程序发送一个 add-device 请求。PnP 管理器计算出最佳资源分配,向每个设备驱动程序发送一个 start-device 命令以及为该设备分配的资源。如果一个设备需要重新配置,那么 PnP 管理器会发送一个 query-stop 命令,用于询问驱动程序能否暂停设备。如果驱动程序可暂停设备,那么就完成所有未决操作,并阻止新操作开始执行。接着,PnP 管理器发送一个 stop 命令;然后它可用另一个 start-device 命令来重新配置设备。PnP 管理器也支持其他命令,如 query-remove。在用户准备弹出 PCCARD 设备时,可使用这一命令,其执行过程类似于 query-stop。当设备出错或者更为可能的是用户没有停止就直接移去 PCCARD 时,就使用 surprise-remove 命令。命令 remove 要求驱动程序停止使用设备并释放该设备所分

配的所有资源。

21.4 环境子系统

环境子系统是用户模式进程，该进程建立在 Windows 2000 执行体服务程序之上，(这样)Windows 2000 才得以运行为其他操作系统开发的程序，包括 16 位的 Windows 操作系统、MS-DOS 系统、POSIX 系统以及基于字符的 16 位 OS/2 系统应用程序，每个环境子系统提供一个 API 或者是应用环境。

Windows 2000 把 Win32 子系统作为主要操作环境，并在此基础上运行所有进程。当执行一个应用程序时，Win32 子系统就通过 VM 管理器载入应用程序的可执行代码。内存管理器给 Win32 发回一个状态，告知可执行代码的类型。如果不是 Win32 本身可执行的代码，那么 Win32 将核查合适环境子系统是否已在运行。如果子系统尚未运行，就把它作为用户态进程启动。然后，该子系统会控制其应用程序的运行。

环境子系统利用 Windows 2000 LPC 程序获取内核进程的服务，这种方法增加了 Windows 2000 的强壮性。在调用实际的内核例程之前，要核查传送给一次系统调用的参数的正确性。Windows 2000 禁止应用程序与不同环境下的 API 例程混合使用。例如，一个 Win32 的应用程序不能调用一个 POSIX 例程。

因为每个子系统是作为一个独立的用户模式进程运行的，一个子系统出现崩溃并会不影响其他的子系统。但是 Win32 是个例外，它提供所有的键盘、鼠标和图形显示功能。如果它不能提供这些功能，系统就失效了。

Win32 环境把应用程序分为两类：要么是基于图形的，要么是基于字符的，该环境中一个*基于字符的应用程序*是一个(认为是)交互输出到基于字符的窗口。Win32 把基于字符的程序输出结果转换成命令窗口的图形表示。这种转换并不难：当需要调用输出程序时，环境子系统调用 Win32 程序以显示文本。因为 Win32 环境为所有基于字符的窗口执行这一功能，所以可通过剪贴板在窗口之间传送屏幕文本。这种转换不但适用于 MS-DOS 程序，而且也适用于 POSIX 命令行程序。

21.4.1 MS-DOS 环境

MS-DOS 环境没有其他的 Windows 2000 环境子系统那么复杂。它由一个称为虚拟 DOS 机(virtual DOS machine，VDM)的 Win32 应用程序提供。由于 VDM 只是一个用户态进程，它的分页和调度与其他的 Windows 2000 线程一样。VDM 的**指令执行单元**(instruction-execution unit)能执行或者模仿 Intel 486 的指令。VDM 还提供程序模仿 MS-DOS ROM BIOS 和"int 21"软件中断服务程序，并拥有用于屏幕、键盘以及通信接口的虚拟设备驱动程序。VDM 基于 MS-DOS 5.0 源代码；它给予应用程序的内存空间至少有 620 KB。

Windows 2000 命令 shell 新建的窗口看起来很像 MS-DOS 环境。可以运行 16 位和 32 位的可执行体。运行一个 MS-DOS 环境时，命令 shell 就启动一个进程执行该项任务。

如果 Windows 2000 运行于一个 x86 处理器上，那么 MS-DOS 图形程序将以全屏模式运行，而字符程序可以运行于全屏或者是一个窗口。如果 Windows 2000 是运行在不同的处理器特性结构中，那么所有的 MS-DOS 的应用程序都会在 Windows 下运行。有些 MS-DOS 程序直接访问磁盘硬件，但是因为磁盘是受限制的（用于保护文件系统），所以就不能运行 Windows 2000。总的来说，直接访问硬件的 MS-DOS 应用程序不能在 Windows 2000 下运行。

MS-DOS 不是多任务环境，有些应用程序会独占整个 CPU。例如，通过使用过忙的循环体将导致执行过程中时间的延迟或者是暂停。Windows 2000 调度程序中的优先机制会发现这种推延并自动取消 CPU 的耗费，这导致类似应用程序不能正常运行。

21.4.2 16 位 Windows 环境

Win16 执行程序环境是由一个 VDM 提供的，该 VDM 合并了称做关于*窗口的窗口*（Windows on Windows）的额外软件。这一软件提供了 Windows 3.1 内核例程，用于 Windows 管理器和 GDI 子例程的 stub 例程。stub 例程可以调用适当的 Win32 子例程，转换，*形实转换*（thunk），16 位地址转换成 32 位地址。依赖于 16 位 Windows 管理器或 GDI 内部结构的应用程序可能无法运行，这是因为底层 Win32 实现当然是有别于真正的 16 位 Windows。

Windows on Windows 可以和 Windows 2000 的其他进程进行多任务的操作，但是它在很多方面都像 Widows 3.1。一次只能运行一个 Win16 应用程序，所有程序都是单线程的，驻留在同一地址空间，共享同一输入队列。这些功能意味着应用程序停止接收输入将阻断所有其他的 Win16 应用程序。就像 Windows 3.x 一样，而且一个 Win16 应用程序会因为破坏地址空间而毁损其他的 Win16 应用程序。然而，多个 Win16 环境能共存，则要通过从命令行调用 start/separate win16application 命令（才能实现）。

21.4.3 Win32 环境

Windows 2000 的主要子系统是 Win32 子系统。它运行 Win32 的应用程序，并管理着所有的键盘、鼠标和屏幕 I/O。因为它控制环境，所以设计得非常健壮。Win32 的几个特点都有助于提高这种健壮性。与 Win16 环境不同，每个 Win32 进程都有它自己的输入队列，窗口管理器把系统的所有输入项调度到适当的进程输入队列，因而一个失效的进程不会阻塞向其他进程的输入操作。Windows 2000 内核也提供抢占式多任务，它可以让用户终止已经失效或者是不再需要的应用程序。在使用之前 Win32 检查所有对象，以防止可能因为应用程序试图使用一个失效的和错误的句柄而导致崩溃。Win32 子系统可以在使用对象前核

实句柄所指向的对象类型。管理器一直保持着引用计数，以防对象在使用中被删除，防止对象在删除后被使用。

21.4.4 POSIX 子系统

POSIX 子系统被设计成能运行遵循 POSIX.1 标准的应用程序。该标准基于 UNIX 模型。POSIX 应用程序能通过 Win32 子系统和其他的 POSIX 应用程序启动。POSIX 应用程序用的是 POSIX 子系统服务器 PSXSS.EXE、POSIX 动态连接库 PSXDLL.DLL 以及 POSIX 控制台会话管理器 POSIX.EXE。

虽然 POSIX 标准不能指定打印技术，但是 POSIX 应用程序可以通过 Windows 2000 重定向器机制透明地使用打印机。POSIX 应用程序拥有访问 Windows 2000 系统中所有文件系统的能力。POSIX 环境在目录树方面执行类似 UNIX 的许可(标准)。POSIX 子系统不支持某些 Win32 的程序，包括内存映象文件、网络、图形以及动态数据交换。

21.4.5 OS/2 子系统

虽然 Windows 2000 在刚开始时准备提供一个强壮的 OS/2 操作环境，但是 Microsoft Windows 的成功改变了初衷；在 Windows 2000 的早期开发阶段，Windows 环境是它的默认环境。接下来，Windows 2000 提供了只限于 OS/2 环境子系统的工具。OS/2 1.X 基于字符的应用程序只能在 Intel x86 上的 Windows 2000 下运行。通过使用 MS-DOS 环境，实时模式的 OS/2 应用程序可以在所有的平台运行。同时拥有 MS-DOS 和 OS/2 对偶码的受限制程序，在 OS/2 环境不失效的情况时可以在该环境下运行。

21.4.6 登录和安全子系统

在访问 Windows 2000 的对象之前，用户必须通过登录子系统的鉴别。为了通过鉴别，用户必须拥有一个账号以及提供与账号相对应的口令。

安全子系统用发给访问令牌的方式来代表系统用户。它通常调用一个**鉴别程序包**(authentication package)并利用来自登录子系统或者是网络服务器的信息来执行鉴别操作。通常，鉴别程序包只是查阅本地数据库中的账户信息并核查确认口令的正确性。接着，安全子系统再为用户 ID 生成访问令牌，其中包含了适当的优先权、配额限制和组 ID。不管什么时候用户试图访问系统中的一个对象，如打开一个指向对象的句柄，访问令牌被传递给安全引用监视器以核查优先权与配额。Windows 2000 域的缺省鉴别程序包是 Kerberos。

21.5 文件系统

在历史上，MS-DOS 系统使用的是文件分配表(简称 FAT)文件系统。16 位的 FAT 文

件系统有很多缺点，如内部文件碎片，2 GB 的大小限制以及缺少文件的访问保护。32 位 FAT 文件系统已经解决了大小和碎片的问题，但是与现代的文件系统相比，它的性能和功能依然很弱。NTFS 文件系统就好多了。它设计有很多的功能，包括文件恢复、安全、容错、超大文件和文件系统、多数据流、UNICODE 名以及文件压缩。在兼容性方面，Windows 2000 能支持 FAT 和 OS/2 HPFS 文件系统。

21.5.1 内部布局

NTFS 文件系统中的基本实体是卷。卷是由 Windows 2000 的磁盘管理工具基于磁盘的逻辑分区表创建的。卷可能占据一个磁盘的一部分，也可能占据着整个磁盘，或者是横跨几个磁盘。

NTFS 不与磁盘的单个扇区打交道，而是采用**簇**(cluster)作为磁盘分配的单元。一个簇的(大小)包括 2 的乘方个磁盘扇区。一个 NTFS 文件系统在格式化时给簇配置大小。一个缺省簇的大小，对于不超过 512 MB 的卷为一个扇区，对于不超过 3GB 的卷为 1 KB，对于不超过 2 GB 的卷为 2 KB，而对于更大的卷为 4 KB。这种簇大小比 16 位 FAT 文件系统的簇要小很多，小尺寸降低了内部碎片的数量。例如，考虑一个有 16 000 个文件的 1.6 GB 的磁盘。如果你使用的是一个 FAT16 的文件系统，因为簇的大小是 32 KB，内部碎片可能会达到 400 MB。在 NTFS 系统下，当保存同样数量的文件时只会丢失 17 MB。

NTFS **把逻辑簇号码**(logical cluster numbers)(简称 LCN)用做磁盘地址。按照磁盘的开头到末端的顺序给每个簇编号。利用此方案，通过以簇的大小乘以 LCN 所得的结果，系统便能计算出一个物理磁盘的字节偏移量。

NTFS 中的文件并不是像 MS-DOS 或者是 UNIX 系统中认为的简单的字节流；相反，它是一个由多种**属性**组成的结构化的对象。一个文件的每个属性都是一个独立的字节流，这种字节流可以新建、删除、读取和写出。有一些标准属性是适用于所有的文件的，包括文件名(或名称，如果文件有别名如 MS-DOS 的短名)、创建时间以及说明访问控制的安全描述符。其他的一些属性特定于某种类型的文件。例如，Macintosh 文件就有两个数据属性，分别为资源 fork 和数据 fork。每个目录都具有索引目录中文件的属性。总的来说，可以根据需要加进某些属性，也可以用一种 *file-name:attribute* 术语去访问这些属性。大多数的数据文件都有一个 *unnamed* 的数据属性，它包含了所有该类文件的数据。对于文件查询操作如运行 dir 命令，NTFS 只返回了未命名属性的大小。比较清楚的是，属性大小不定。

NTFS 的每个文件都是保存在一个特殊文件数组中的一个或多个记录，这个文件叫做**主文件表**(简称 MFT)。一个记录的大小在文件系统被创建时就确定了；它的范围大小在 1 KB～4 KB 之间。一些小的属性存放在 MFT 记录当中，并被叫做**常驻属性**(resident attribute)。如未命名的巨大的数据，属于超级属性，叫做**非常驻属性**，被保存在一个或多个连续的磁盘盘区中，而指向每个盘区的每个可寻址指针则保存在 MFT 记录中。至于一个极

小的文件，甚至数据属性也完全符合 MFT 记录。如果一个文件具有多种属性，或者它是高度分散的，那么就需要许多的指针来指向所有的存储片，MFT 中的一个记录可能就不够大了。在这种情况下，文件由一个**基本文件记录**来描述，基本文件记录也是一条记录，它包含了指向溢出记录（拥有额外的指针和属性）的指针。

NTFS 卷的每个文件都有一个惟一的 ID，称为**文件引用**（file reference）。文件引用是一个 64 位的值，它由一个 48 位的文件号码和一个 16 位的序列号码组成。文件号码就是 MFT 中描述文件的记录号码（也就是，数组插槽）。每次使用一个 MFT 的记录时，其序列号递增。这种顺序号码允许 NTFS 执行内部一致性检测，如在 MFT 项用于新文件之后发现一个删除文件的失效引用。

与在 MS-DOS 和 UNIX 系统中一样，NTFS 的名称空间是根据目录层构成的。每个目录使用一个称为 **B+树**的数据结构，用于保存该目录内的所有文件名称的一个索引。B+树能消除重新组织树的耗费，并具有这样的特性：从树根到树叶的每个路径等长。目录的**索引根**（index root）包括 B+树的顶层。对于一个超大目录，这个顶层包含了磁盘的延伸区，而延伸区保存了此树的剩余部分。目录的每项都包含文件的名称和引用以及更新过的时间信息拷贝、从 MFT 的文件常驻属性中获取的文件大小。这些信息的副本被保存在目录中，因此就有能力生成一个目录列表，上面有全部的文件名、文件大小以及从目录本身就可获得更新的时间，所以就没有必要为了每个文件把所有的这些属性从 MFT 的各项中集中起来。

NTFS 卷的元数据均保存在文件中。第一个文件是 MFT。第二个文件，用于 MFT 文件遭破坏时的恢复，包括了 MFT 头 16 项的一个副本。下面的一些文件也很特殊。它们是日志文件、卷文件、属性定义表、根目录、位图文件、引导文件以及坏簇文件。*日志文件*记录文件系统更新的所有元数据。**卷文件**（volume file）包含卷名，卷格式化后的 NTFS 版本以及显示已被损坏的卷而需要连续查验的一个位。**属性定义表**（attribute-definition table）显示卷中用的是哪种属性类型，对它们中的每个要执行什么样的操作。**根目录**（root directory）是文件系统层中的顶级目录。**位图文件**（bitmap file）显示卷中的哪几个簇被分配给了文件以及哪几个是空闲的。**导入文件**包含了 Windows 2000 的启动代码，并且必须放在一个特殊的磁盘位置以便简单 ROM 导入设备的装入程序能很容易查找到。导入文件还包含了 MFT 的物理地址。最后，**坏簇文件**（bad-cluster file）跟踪卷内的任意坏区；NTFS 使用这个记录以进行错误恢复。

21.5.2 恢复

在许多的简单文件系统中，非正常掉电会导致文件系统的数据结构遭到极其严重的破坏甚至把整个卷都搞乱了。许多版本的 UNIX 在磁盘上保存了冗余的元数据，它们利用 fsck 程序去检查所有文件系统数据结构，得以从崩溃的系统中恢复过来，并把它们强行保存到一个相容的状态。恢复这些数据经常会碰到删除了损坏的文件和释放了数据簇，而用户

的数据已被写进了这些簇但尚未完全地记录到文件系统的元数据结构中。这种核查是一个很缓慢的过程，并且会丢失相当数量的数据。

NTFS对于文件系统的健壮性采取了一种截然不同的方法。在NTFS系统中，所有的文件系统数据结构的更新都在内部事务中执行。在改动一个数据结构之前，事务会写入一个包含redo和undo信息的日志记录；在改变完数据结构之后，事务会在日志上写进一条提交记录，表示该事务已操作成功。在一次崩溃之后，系统通过处理日志记录把文件系统的数据结构恢复到了一个相容的状态。首先是重复对已提交事务的操作，然后撤销对在系统崩溃之前未成功提交的事务的操作。一个检查点记录会被周期性地(通常是5 s)写进日志中去。系统要从一次崩溃中恢复时不需要在检查点之前的日志记录。日志文件会被废除，因此日志文件不会无限制地增加。待系统启动之后，首先访问的便是一个NTFS的卷，NTFS会自动执行文件系统的恢复。

此方案并不能保证一次崩溃之后的所有用户文件还能保持正确，它只能确保文件系统的数据结构(元数据文件)没有被损坏，并能反映崩溃之前某些业已存在的一致状态。把事务方案延伸到用户文件也不是没有可能的，只是系统开销会削弱文件系统的性能。

日志被保存在卷初的第三个元数据文件中。它是在文件系统格式化时用一个固定的最大范围创建的。它有两个区域：一个是**存入区域**(logging area)，它是日志记录的一个循环队列，一个是**重启区域**(restart area)，它包含关联信息；就像存入区域的位置，是恢复期间NTFS启动读取操作的地方。事实上，重启区域有它相关信息的两个副本，如果一个副本在崩溃中被损坏，那么恢复还是可能的。

记录功能是由Windows 2000 **日志文件服务**(logging file service)所提供的。除了写日志记录和执行恢复操作外，日志服务还跟踪日志文件的空闲空间。如果空闲空间太少，那么日志文件服务会暂停进行的交易，NTFS会暂停所有新I/O操作。在所进行操作完成之后，NTFS会调用缓存管理器以写出所有数据，接着重新设置日志文件并执行待进行的交易。

21.5.3 安全

NTFS卷的安全性是从Windows 2000的对象模型中派生出来的。每个NTFS文件都引用一个安全描述符，它包含文件所有者的访问令牌以及一个访问控制列表，用于规定访问该文件的用户所具有的访问特权。

21.5.4 卷管理及容错

FtDisk是Windows 2000的容错磁盘驱动程序。当安装时，它提供了许多种方法把多个磁盘驱动器联结成一个逻辑卷，以便于提高性能、计算效率或是可靠性。

组合多个磁盘的一种方法是把它们进行逻辑连接以形成一个超大的逻辑卷，如图21.7所示。在Windows 2000中，这个逻辑卷被称为**卷集**(volume set)，它由0～32个物理分区组

成。一个卷集包含有一个 NTFS 卷;已保存于文件系统中的数据在没有被扰乱的情况下,可以扩展该卷集。扩展 NTFS 卷中的位图元数据只是去覆盖新增的空间。NTFS 连续使用同一个 LCN 机制,一个单一的物理磁盘,FtDisk 驱动程序满足了从一个逻辑卷偏移到一个特殊磁盘中的偏移的映射。

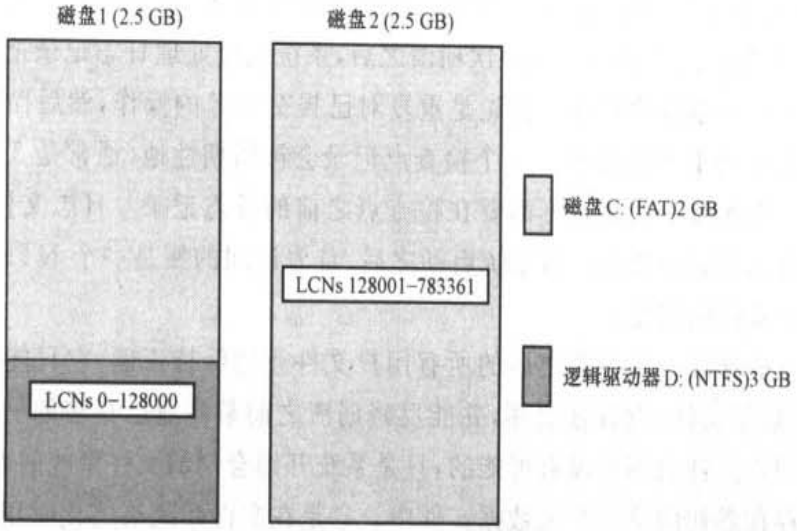


图 21.7 两个驱动器上的卷集

合并多个物理分区的另一个方法是用循环法交替使用它们的区块以形成所谓的**条集**(stripe set),如图 21.8 所示。这种方案称为 RAID 级 0,或者是**磁盘分条法**(disk stripe),FtDisk 使用的是一个 64 KB 大小的条。逻辑卷的第一个 64 KB 保存在第一个物理分区,第二个 64 KB 的逻辑卷保存在第二个物理分区,依此类推,直到每个分区都分配了 64 KB 的空间。然后,分配绕回到第一个磁盘,分配第二个 64 KB 的区块。条集形成了一个巨大的逻辑卷,但是物理布局可以改善 I/O 的带宽,对于一个巨大的 I/O,所有的磁盘可以并行转移。关于 RAID 的更多的内容,请查看 14.5 节。

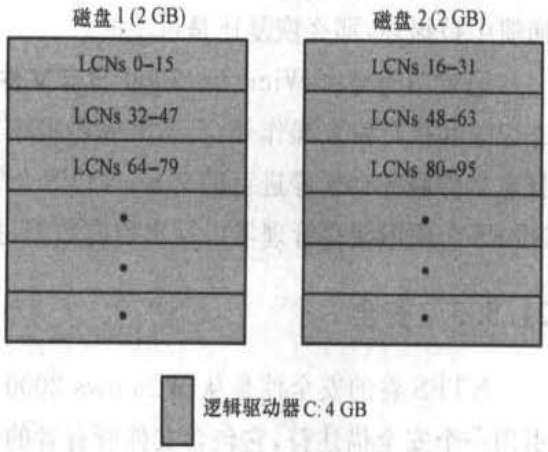


图 21.8 在两个驱动器上的条集

这个思路的一个变通是**奇偶条集**(stripe set with parity),如图 21.9 所示。这个方案也被称为 RAID 5 级。如果条集拥有 8 个磁盘,那么 7 个数据条分别位于 7 个独立的磁盘,而奇偶条位于第 8 个磁盘上。奇偶条含有数据条的基于字节的异或(exclusive or)。如果这 8 个条中的任意一个遭到毁损,系统将通过计算异或并根据剩余 7 个奇偶条的数据而重新构建数据。重建数据的能力使得磁盘阵列在磁盘遭遇错误时丢失数据的可能性大大减少。我

们注意到，一个数据条的更新同样要重新计算奇偶条。同时向 7 个不同的数据条(写进数据)的 7 个写操作同样也需要更新 7 个奇偶条。如果奇偶条都在同一个磁盘，那么该磁盘将执行 7 次的数据磁盘输入/输出装入(操作)。为避免建立瓶颈，通过给它们赋循环值，把奇偶条分摊在所有的磁盘上。用奇偶校验建立一个条集，至少需要独立于不同磁盘的大小相同的 3 个分区。

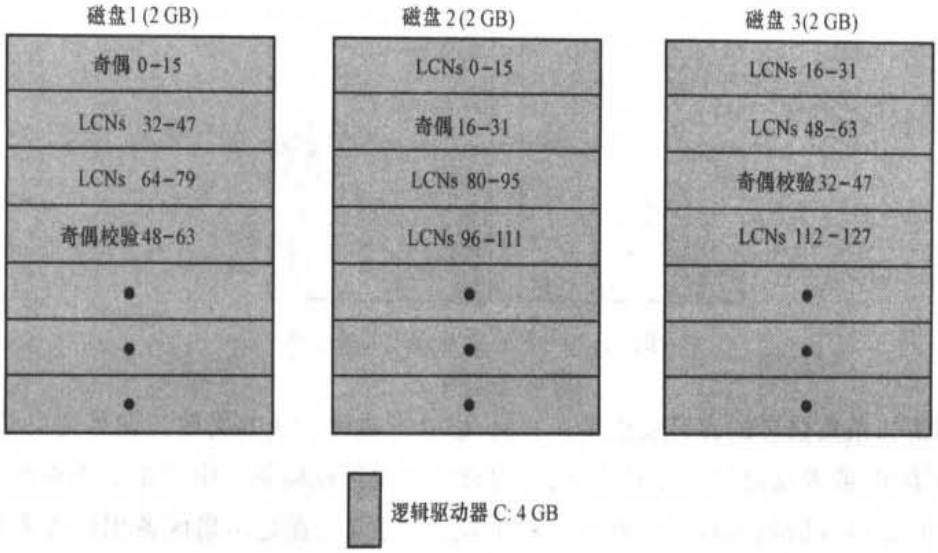


图 21.9 三个驱动器上的带奇偶校验条集

一个更强壮方案是**磁盘镜像**(disk mirror)，或者叫做 RAID 1 级，如图 21.10 所示。一个**镜像集**(mirror set)由两个分别位于不同磁盘的大小相同的分区组成，这样它们的数据内容是完全相同的。当一个应用程序把数据写进一个镜像集时，FtDisk 就把数据同时写入了两个分区。如果一个分区写入失败，FtDisk 还有一个拷贝安全地存放于镜像中。镜像集同样可以提高性能，因为读请求会在两个镜像之间进行分割，给予每个镜像一半的工作量。为了防止磁盘控制器的失效，可以把一个镜像集的两个磁盘放在两个独立的磁盘控制器上，这种安排叫做**双工集**(duplex set)。

为处理变坏的磁盘扇区，FtDisk 使用了一种硬件技术，称为扇区备用法，而 NTFS 使用的是一种软件技术，称为簇重新映射法。扇区备用法(sector sparing)是一种由许多磁盘驱动器提供的硬件能力。当一个磁盘驱动器被格式化时，它在逻辑区块号码与磁盘的好扇区之间建立一个映射。它同时也保留了未被映射的扇区作为备用。如果一个扇区失效了，FtDisk 就命令磁盘驱动器替代一个备用扇区。**簇重新映射法**(cluster remapping)是文件系统执行的一种软件方法。如果一个磁盘区块变坏了，NTFS 通过改变 MFT 中任一受影响的指针，用一个不同的未分配的区块来置换它。NTFS 还会做上一条标记，以注明损坏区块不会将这些扇区分配给任何文件。

一个磁盘区块损坏时，通常的结果是数据的丢失。但是扇区备用法与簇重新映射法也

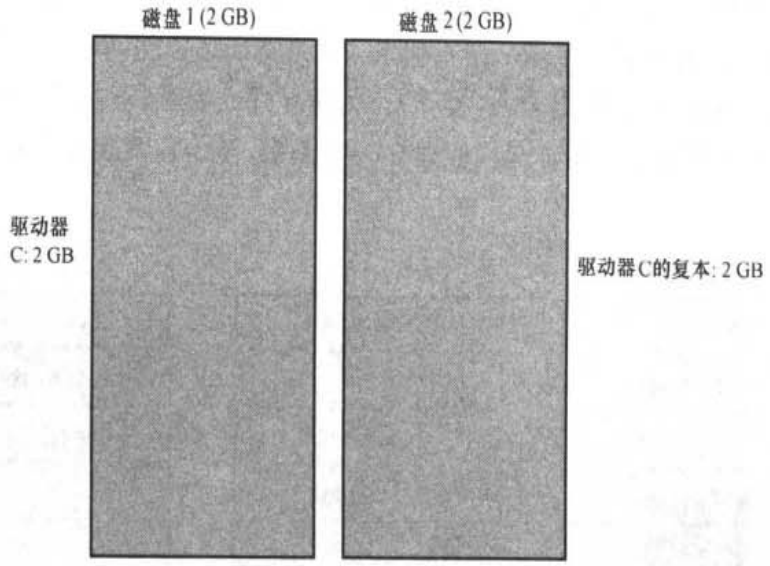


图 21.10 两个驱动器上的镜像集

可能会和诸如条集这样的容错卷连接在一起，以处理磁盘区块的失效。如果读失败，系统会通过读取镜像，或者是通过计算异或，或是用奇偶校验法校验条集中的奇偶来重新构建丢失的数据。重新构建的数据被存放到一个新的位置，而该位置是用扇区备用法或者是簇重新映射法取得的。

21.5.5 压缩技术

NTFS可以对目录中个别文件或者是所有数据文件进行数据压缩。在压缩一个文件之前，NTFS首先把文件的数据分成几个**压缩单元**，压缩单元是16个相邻簇的区块。当写进每个压缩单元时，就应用一种数据压缩算法。如果结果符合的要比16簇少，那么就保存这个压缩版本。要进行读时，NTFS能确定已被压缩的数据：如果是压缩过的，被保存的压缩单元的长度一定小于16簇。为了提高性能，当读取连续的压缩单元时，NTFS会在应用程序发出请求之前提前提取并解压缩。

对于稀疏文件或者是绝大部分都是零的文件，NTFS使用另外一种方法来节省空间。由于未被写过而含有所有零的文件簇，实际上并没有被分配和保存在磁盘上。而是以间隙的形式保存在文件MFT项的各个虚拟簇的序列号码中。读一个文件时，如果它找到了虚拟簇号码间的间隙，NTFS只是用零填充调用程序的缓冲区分区。UNIX使用的也是这种方法。

21.5.6 再解析点

再解析点(reparse point)是文件系统中一个新的功能，访问它们时，能有效地返回一个错误代码，然后再解析数据会告诉I/O管理器去做什么。

安装点是Windows 2000另一个新增加的功能。它们是再解析点的一种形式。与

UNIX 系统不一样的是，以前的 Windows 版本没有提供逻辑分区连接方法，每个分区都被指定了一个驱动器字母，这样与其他的任何分区就有很明显的区别。这意味着，在其他的事件当中，如果加进了一个文件系统，就需要改动目录结构以增加空间。安装点允许在另一个磁盘中新建一个新的卷，并把旧的数据移植到新卷中，然后在原来位置安装新卷。该数据还可被装好的程序重新利用，因此该数据会像从前一样出现在同样的位置。安装点操作是作为一个再解析点来完成的，而该再解析点还带有包含真实卷名的再解析数据。

远程存储器服务(remote storage service)程序也使用再解析点，一个文件移植到脱机的存储器时，原始文件数据就被一个含有关于文件所在位置信息的再解析点替换掉了。以后访问该文件时，就可以重新找到，文件中的数据也替换了再解析点。关于存储器分层的更多内容，请参见 14.8.1 节。

21.6 网　络

Windows 2000 同时支持对等网络与客户/服务器网络，还拥有网络管理程序。Windows 2000中的网络组件有数据传送、进程间通信、网络中的文件共享以及向远程打印机发送打印任务。

为描述 Windows 2000 网络，将引用两个内部网络接口，它们是**网络设备接口规范**(network device interface specification，NDIS)和**传输驱动程序接口**(translation driver interface，TDI)。NDIS 接口是由微软公司和 3Com 公司于 1989 年合作开发的，主要是为了把网络适配器从传输协议中分离开，这样两者可以在相互不影响的情况下得到改变。NDIS 是驻留在 OSI 模型的"数据连接控制"与"媒介访问控制"层之间的接口；它使得许多协议能够在许多不同的网络适配器上进行操作。就 OSI 模型来说，TDI 是位于传输层(第四层)和会话层(第五层)之间的接口，此接口允许任意会话层的组件使用任何可利用的传输机制。(与 UNIX 中<数据>流有很相似的原因。)TDI 同时支持基于连接与非连接的传输，具有发送任何类型数据的功能。

21.6.1 协议

Windows 2000 执行的传输协议与驱动程序一样。这些驱动程序可以从系统中动态地装入或者是卸载，但是在实际应用中，一次改变之后系统通常需要重新启动。Windows 2000 提供好几种协议。

服务器消息块(Server Message Block，SMB)首先用于 MS-DOS 3.1。系统可以使用这一协议通过网络发送 I/O 请求。SMB 协议有四种消息类型。消息 Session Control 命令用于开始和结束服务器端共享资源的重定向连接。重定向器采用消息 File 访问服务器端的文件。系统使用消息 Printer 向远程打印机队列发送数据并接收状态信息，消息 Message 用

于与另一工作站进行通信。

网络基本 I/O 系统(Network Basic Input/Output System,NetBIOS)是用于网络的硬件抽象接口,类似于用于运行 MS-DOS 的 PC 硬件抽象接口 BIOS。NetBIOS 设计于 20 世纪 80 年代早期,现已成为标准网络编程接口。NetBIOS 用于建立在网络之上的逻辑名,建立网络逻辑名之间的逻辑连接或会话,支持通过 NetBIOS 或 SMB 请求的可靠会话数据传输。

网络 BIOS 扩展接口(NetBIOS extended user interface,NetBEUI)是由 IBM 公司于 1985 年引入的,用于支持不超过 254 台机器的简单而高效的网络协议。这是 Windows 95 对等网络和 Windows for Workgroups 的缺省协议。Windows 2000 在与这些网络进行资源共享时,使用 NetBEUI。NetBEUI 限制很多,它使用计算机的真实名称作为地址,而且不支持路由。

TCP/IP 协议系列已变成事实上的标准网络内部结构,它受到广泛的支持。Windows 2000 通过 TCP/IP 连接多种操作系统和硬件平台。Windows 2000 TCP/IP 程序包中包括了 SNMP(Simple Network-Management Protocol)、DHCP(Dynamic Host-Configuration Protocol)、WINS(Windows Internet Names Service)以及 NetBIOS 支持。

点到点隧道协议(point-to-point tunneling protocol,PPTP)由 Windows 2000 提供,用于在运行 Windows 2000 服务器的远程访问服务器模块和与因特网相连的客户机之间进行通信。远程访问服务器可以对连接上所发送的数据进行加密,它们支持多协议的通过因特网的**虚拟私有网络**(virtual private network, VPN)。

Novell Netware 协议(SPX/IPX)广泛地用于 PC LAN。Windows 2000 的 NWLink 协议将 NetBIOS 与 Netware 网络相连。与重定向器(如 Microsoft 的 Netware 客户服务或 Novell 的用于 Windows 的 Netware 客户)一起,这个协议允许 Windows 2000 客户机与 NetWare 服务器相连。

Windows 2000 通过**数据链路控制协议**(data-link control protocol,DLC)访问与网络直接连接的 IBM 主机和 HP 打印机。这个协议不会被 Windows 2000 另作它用。

AppleTalk 协议是由 Apple 所设计的,允许 Macintosh 计算机共享文件的低成本连接。如果位于网络的 Windows 2000 服务器运行用于 Macintosh 包的 Windows 服务,那么 Windows 2000系统可通过 AppleTalk 与 Macintosh 计算机共享文件和打印机。

21.6.2 分布式处理机制

虽然 Windows 2000 不是分布式操作系统,但是它确实支持分布式应用。Windows 2000 的分布式处理支持的机制包括 NetBIOS、命名管道、邮件槽(mailslot)、Windows Socket、RPC 和网络动态数据交换(NetDDE)。

对于 Windows 2000,NetBIOS 应用程序可通过网络采用 NetBEUI、NWLink 或 TCP/IP 进行通信。

命名管道(name pipe)是面向连接的消息机制。命名管道最初设计成网络 NetBIOS 连接的高层接口。一个进程也可以使用命名管道与同一台机器的另一个进程进行通信。由于命名管道通过文件系统接口加以访问,所以用于文件对象的安全机制也适用于命名管道。

命名管道的名称有一个称为**统一命名习惯**(uniform naming convention,UNC)的格式。UNC 名称看起来像普通远程文件名称。UNC 名称的格式为\\server_name\share_name\x\y\z,其中 server_name 标识网络服务器;share_name 标识可为网络用户所用的网络资源如目录、文件、命名管道和打印机等;\x\y\z 部分表示普通文件路径名。

邮件槽(mailslot)是一种无连接的消息传递机制。当通过网络使用时,邮件槽是不可靠的。即发送给邮件槽的消息可能会丢失,导致要接收它的用户不能接收到。邮件槽用于广播应用程序如查找网络组件;也可用于 Windows 2000 计算机浏览器服务。

Winsock 是 Window 2000 的套接字 API。它是一个会话层接口,与大多数的 UNIX 套接字兼容,并且与 Windows 2000 的某些扩展部分兼容。它提供了对许多不同传输协议(具有不同寻址方式)的一个标准接口,这样 Winsock 应用程序可运行于任何与 Winsock 相兼容的协议堆栈。

远程子程序调用(remote procedure call,RPC)是个客户机—服务器通信机制,允许一台机器上的一个应用程序调用另一台机器上的子程序。客户进程调用本地子程序——即**存根子程序**(stub routine),将参数打包成一个消息,并通过网络发送给一个特定服务器进程。接着客户端存根子程序就阻塞。同时,服务器解开消息,调用子程序,并将结果打包成消息,再传递给客户存根。客户存根不再阻塞,接收消息,解开 RPC 的结果,并返回给客户进程。这种参数打包有时称为**编组**(marshalling)。

Windows 2000 RPC 机制遵守广泛使用的用于 RPC 消息的分布式计算环境的标准。RPC 标准非常详细。它通过规定 RPC 消息的标准数据格式,而隐藏了计算机体系结构的差异如二进制大小、计算机字的字节和位的顺序。

Windows 2000 可通过 NetBIOS 或 TCP/IP 网络的 Winsock 或 LAN Manager 网络的命名管道等,来发送 RPC 消息。以前所讨论的 LPC 与 RPC 相似,不过在同一计算机的两进程之间的消息传递是通过 LPC 消息进行的。

可以通过手工编写代码并按标准形式编排和传递参数,解开参数并执行远程子程序,编排和传递返回结果,解开结果并返回给调用程序,但是这非常麻烦且容易出错。然而,幸运的是,根据参数和返回结果的简单描述,可以自动地生成这部分代码。

Windows 2000 提供了**微软接口定义语言**(Microsoft Interface Definition Language)描述远程子程序的名称、参数和结果。这种语言的编译器可生成头文件以描述远程子程序的存根和参数与返回值消息的数据类型。它也为客户端存根子程序和服务器端的解开和分派、生成源程序。当链接应用程序时,会包括存根子程序。当应用程序执行 RPC 存根,所生成的代码处理其他部分。

组件对象模型(component object model,COM)是为 Windows 开发的进程间通信机制。COM 对象提供了明确定义的操作对象数据的接口。例如,COM 是 Microsoft 的**对象链接和嵌入**(object linking and embedding,OLE)的基础,它可用于在 Word 文档中嵌入电子表格。Windows 2000 有一个通过 RPC 运行于网络之上的称为 DCOM 的分布式扩展,提供一种开发分布式应用程序的透明方式。

21.6.3 重定向器与服务器

对于 Windows 2000,只要远程计算机运行 MS-NET 服务器(Windows 2000 或 Windows for Workgroups 所提供的),应用程序可使用 Windows 2000 I/O API 如同位于本地一样访问远程计算机的文件。**重定向器**(redirector)是客户端对象,用于转寄对远程文件的 I/O 请求,再为服务器所处理。基于性能和安全原因,重定向器与服务器运行于内核模式。

下面将更详细地说明访问一个远程文件所发生的情况:

- 应用程序调用 I/O 管理器,请求以标准的 UNC 格式根据文件名打开某个文件。
- I/O 管理器构建一个 I/O 请求程序包,就像 21.3.3.5 节所描述的一样。
- I/O 管理器知道这是个远程文件的访问,因此调用一个称为**多通用命名标准程序**(multiple universal naming convention,MUP)的驱动器。
- MUP 把 I/O 请求程序包异步地发送给所有注册的重定向器。
- 符合请求的重定向器会响应 MUP。为避免未来向所有的重定向器询问同样的问题,MUP 利用一个高速缓存来记取能够处理该文件的重定向器。
- 重定向器把网络请求发送到远程系统。
- 远程系统网络驱动程序接收到请求,并把它传输到服务器的驱动程序。
- 服务器驱动程序把请求传递给适当的本地文件系统驱动程序。
- 调用适当的设备驱动程序去访问数据。
- 把结果返回服务器驱动程序,它把数据发回到请求的重定向器。然后重定向器通过 I/O 管理器把数据返回到调用程序。

对于使用 Win32 网络 API 而不是 UNC 服务的应用程序也有类似过程。不过,所使用的模块是多提供者的路由而不是 MUP。

为了可移植性,重定向器与服务器采用基于网络传输的 TDI API。请求本身用更为高层的协议表示,缺省为 21.6.1 小节所述的 SMP 协议。重定向器列表由系统注册表数据库维护。

21.6.4 域

许多网络环境都由用户自然组合而成,如学校计算机实验室的学生,或某企业一个部门的雇员。通常,需要让组内的所有成员都能够访问组内各台计算机的共享资源。为了管理

这种组的全局访问权限，Windows 2000 采用了域的概念。以前，这些域与 DNS(将因特网名称转换成 IP 地址)没有任何关系。然而现在它们却紧密地联系在一起。具体地说，Windows 2000的一个域是共享共同安全策略和用户数据库的一组 Windows 2000 工作站和服务器。由于 Windows 2000 现在使用 Kerberos 协议以用于信任和验证，所以 Windows 2000 的域与 Kerberos 领域完全一样。NT 的以前版本采用了主域控制器和备份域控制器的概念；现在所有域内的服务器都是域控制器。另外，Windows 2000 采用了基于 DNS 的分层方法，允许在层次结构中上下传递信任。这种方法降低了用于 n 个域的信任数量，从 $n\times(n-1)$ 到 $O(n)$。域内工作站信任域控制器会提供关于每个用户访问权限的正确信息(通过用户访问标记)。不管域控制器如何说，所有用户保留限制对其工作站访问的能力。

因为一个企业可以有许多部门，而一个学校可有多个班级，所以通常需要管理一个组织的多个域。一个**域树**(domain tree)为一个连续的 DNS 命名层次。例如，*bell-labs.com* 可以为树根，而 *research.bell-labs.com*(表示域 research)和 *pez.bell-labs.com*(表示域 pez)为孩子。一个**森林**为一组非连续的集合。一个例子是树 bell-labs.com 和/或 lucent.com。然而，一个**森林**可只由一个域树组成。

信任关系可以在域之间按三种方式来建立：单向的、传递的和交叉的。NT 4.0 以前的版本只允许单向信任。**单向信任**(one-way trust)正如其名称所指的：域 A 被告之它可信任域 B。然而，除非配置了这种信任关系，否则域 B 并不信任 A。对于**传递信任**(transitive trust)则是，如果 A 信任 B 且 B 信任 C，那么 A、B、C 互相信任，因为传递信任缺省为双向的。传递信任缺省用于树内的新域，也可配置成用于森林内的域。第三种类型，**交叉信任**(cross-link trust)，可用于减少验证流量。假如域 A 和 B 为叶节点，而域 A 内的用户使用域 B 的资源。如果使用标准传递信任，那么验证请求必须传递到这两个叶节点的共同祖先；但是如果 A 和 B 已建立了交叉连接，那么验证可直接发送给另一节点。

21.6.5 TCP/IP 网络中的名称解析

对于 IP 网络，**名称解析**可将计算机名称转换成 IP 地址，如将 *www.bell-labs.com* 转换成 135.104.1.14。Windows 2000 为名称解析提供了多种方法，包括 WINS(Windows Internet Name Service)、广播名称解析、DNS、文件 host 和文件 LMHOSTS。绝大多数方法被许多操作系统所使用，但这里只讨论 WINS。

对于 WINS，两台或多台 WINS 服务器维护名称—IP 地址捆绑的动态数据库，而客户软件询问服务器。至少使用两台服务器，这样即使一台服务出差错也不会中断 WINS 服务，另外也可将名称解析负荷分布在多台机器上。

WINS 使用了动态主机配置协议(dynamic host-configuration protocol，DHCP)。DHCP 可自动地更新地址配置 WINS 数据库，而无需用户或管理员的干预。当 DHCP 客户机开始时，会广播消息 discover。每个收到消息的 DHCP 服务器会用消息 offer 给以应

答，它包括了客户所需要的 IP 地址和配置信息。客户选择一个配置，并向所选择的 DHCP 服务器发送一个请求消息。DHCP 服务器用之前所给的 IP 地址和配置信息及地址租赁，来加以响应。这个租赁允许客户有权在给定时间内使用这个 IP 地址。当租赁时间用完一半时，客户会试图续租这个地址。如果不能续租，那么客户必须得到一个新地址。

21.7 程序接口

Win32 API 是 Windows 2000 功能的基本接口。本节描述 Win32 API 的五个主要方面：访问内核对象、进程间对象的共享、进程管理、进程间通信和内存管理。

21.7.1 访问内核对象

Windows 2000 内核为应用程序提供了许多服务。应用程序通过处理内核对象来取得这些服务。若进程要访问名为 XXX 的一个内核对象，就要调用 CreatXXX 函数来打开一个指向 XXX 的句柄。这个句柄对该进程来说是惟一的。如果 Creat 函数调用失败，将返回一个“0”值，或者是返回一个名为 INVALID_HANDLE_VALUE 的特殊常量，这都取决于打开了哪个目标。进程通过调用 CloseHandle 函数可以关闭任何一个句柄。运行对象的进程计数减少到 0 时系统将删除这个对象。

Windows 2000 提供三种方法用以共享进程间对象。第一种办法是，子进程继承了一个指向对象的句柄。当父进程调用 CreatXXX 函数时，父进程将 SECURITIES_ATTRIBUTES 结构的 bInheritHandle 字段设定为 TRUE。该字段能新建一个可继承的句柄。然后，新建的子进程把 TRUE 值传递到 CreatProcess 函数的 bInheritHandle 变量。图 21.11 展示了一个代码例子，它用于创建一个信号量句柄并为子进程所继承。

```
...
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa,1,1,NULL);
char command_line[132];
ostrstream ostring(command_line,sizeof(command_line));
ostring<< a_semaphore <<ends;
CreateProcess("another_process.exe",command_line,
                NULL,NULL,TRUE,...);
...
```

图 21.11 通过继承句柄使得子进程能共享一个对象的代码

假定子进程知道共享的句柄，那么父进程与子进程通过共享对象可以成功地进行进程间的通信。在图 21.11 中，子进程从第一命令行参数中取得句柄的值，然后与父进程共享信

号量。

共享对象的第二种方法是在对象创建时进程给该对象命名，以便于第二个进程打开已命名的对象。这种方法有两个缺点。第一个缺点是，Windows 2000 不提供任何方法来检查选定的名称对象是否已经存在。第二个缺点是，对象名空间是全局的，不需要确定对象类型。例如，需要两个截然不同对象时，两个程序都可以新建一个名为 pipe 的对象。

命名对象的优点是不相关进程也可以很容易地进行共享。第一个进程调用 CreatXXX 函数并在 lpszName 参数中提供一个名字。如图 21.12 所示，第二个进程通过用同样的名字以调用 OpenXXX(或者是 CreatXXX)，可取得一个共享该目标的句柄。

```
//process A
...
Handle a_semaphore = CreateSemaphore(NULL,1,1,"MySEM1");
...
//process B
...
Handle b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS
                                   FALSE,"MySEM1");
...
```

图 21.12 通过名称查找来共享一个对象的代码

共享对象的第三个方法是通过调用 DuplicateHandle 函数。这种方法要求一些其他的进程间通信方法来传递重复句柄。在进程中给定一个指向某一进程的句柄及句柄的值，另一个进程就能取得同一对象的一个句柄，这样就可共享了，如图 21.13 中的例子所示。

```
...
//process A wants to give process B access to a semaphore
//process A
Handle a_semaphore = CreateSemaphore(NULL,1,1,NULL);
//send the value of the semaphore to process B
//using a message or shared memory
...
//process B
Handle process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
                                process_id_of_A);
Handle b_semaphore;
DuplicateHandle(process_a,a_semaphore,
    GetCurrentProcess(),&b_semaphore,
    0,FALSE,DUPLICATE_SAME_ACCESS);
//use b_semaphore to access the semaphore
...
```

图 21.13 通过传递一个句柄来共享一个对象的代码

21.7.2 进程管理

在 Windows 2000 系统中，进程是应用程序的一个执行实例，而线程是一个可由操作系统调度的代码单元。这样，一个进程中包含一个或者是若干个线程。当一些其他的进程调用 CreatProcess 例程时，某个进程就被启动了。该例程装入进程所使用的所有动态连接库，并新建一个**主线程**(primary thread)。CreatThread 函数可创建附加线程。每个线程建立时都有自己的栈，除非在调用 CreatThread 函数时加以特别规定，否则栈的默认值是 1 MB。因为有些 C 运行时函数在静态变量中保存状态值，如 errno 函数，一个多线程的应用程序需要防止非同步访问。包装函数 beginthreadex 提供了恰当的同步。

装入进程地址空间的每个动态连接库或可执行文件通过一个实例句柄加以标识。**实例句柄**(instance handle)的值实际上是装入文件所处的虚拟地址。应用程序把模块的名字传递到 GetModuleHandle，便能够取得指向该模块的句柄。如果把 NULL 传递过去作为名字的话，就会返回进程的所在地址。最低的 64 MB 的地址空间是不会使用的，因此一个企图使用 NULL 指针的错误程序会得到一个访问违例的警告。

Win32 环境的优先权基于 Windows 2000 的调度模型，但并不是所有的优先值都会被选用。Win32 使用了 4 个优先级：IDLE_PRIORITY_CLASS(优先级为 4)，NORMAL_PRIORITY_CLASS (优先级为 8)，HIGH_PRIORITY_CLASS(优先级为 13) 以及 REALTIME_PRIORITY_CLASS (优先级为 24)。除非其父进程是属于 IDLE_PRIORITY_CLASS，或者是调用 CreatProcess 函数时已经指定了其他的优先权级，进程通常属于 NORMAL_PRIORITY_CLASS。一个进程的优先权级可以用 SetPriorityClass 函数来改变，或者由传递给 START 命令的参数改变。例如，命令 START /REALTIME cbserver. exe 可以按 REALTIME_PRIORITY_CLASS 运行 cbserver 程序。请注意只有拥有*增长调度优先级*特权的用户才能把进程移到 REALTIME_PRIORITY_CLASS 中。管理员和重要用户缺省地拥有这样的权利。

当用户运行一个交互式程序时，系统需要为该进程提供特别好的性能。为此，Windows 2000有一套适用于 NORMAL_PRIORITY_CLASS 进程的特殊调度规则。Windows 2000 区分两类进程：前台进程(即在屏幕上当前正在选用的)与后台进程(当前没有选用的)。当一个进程移到了前台进程中，Windows 2000 就成倍地增加调度量 ——经常是 3 倍。(这种倍数通过系统控制面板的性能选项可以改变。)在分时抢占发生之前，这种增长使得前台进程运行的时间延长 3 倍(在分时抢占之前)。

线程按其所属类型所确定的最初优先级而开始。线程优先权可以通过 SetThreadPriority 函数改变。该函数带有一个参数，以表示一个优先权与其类型的基本优先权的相互关系。

- THREAD_PRIORITY_LOWEST：base −2
- THREAD_PRIORITY_BELOW_NORMAL：base −1

- THREAD_PRIORITY_NORMAL: base +0
- THREAD_PRIORITY_ABOVE_NOMAL: base +1
- THREAD_PRIORITY_HIGHEST: base +2

另外的两个设计也可用于调整优先权。可以回想一下 21.3.2 小节中提到的内核的两个优先级:16－31 是实时级,0－15 是变量优先级。THREAD_PRIORITY_IDLE 设定 16 为实时线程的优先权,而 1 为变量优先线程的优先权。THREAD_PRIORITY_TIME_CRITICAL 设定了 31 为实时线程的优先权,而 15 为变量优先线程的优先权。

正如在 21.3.2 小节中所讨论的一样,内核动态调整一个线程的优先权依赖于该线程是 I/O 限制还是 CPU 限制。Win32 API 提供了一种使这种调整失效的方法,就是通过 SetProcessPriorityBoost 和 SetThreadPriorityBoost 函数。

线程也可在**悬挂状态**(suspended state)中加以创建:在其他进程通过调用 ResumeThread 函数之前,该线程不能执行。而 SuspendThread 函数则相反。这些函数设定一个计数器,如果一个线程被暂停过两次,那么在它可以运行之前必须恢复(resume)两次。

为了使对进程间共享的对象的访问同步,内核提供了同步对象,如信号量和互斥。另外,线程同步可以通过 WaitForSingleObject 或者是 WaitForMultipleObjects 函数来实现。Win32 API 的另一个同步的方法是临界区。临界区是一个同步代码区域,在每个时间片只能有一个线程执行。线程通过调用 InitializeCriticalSection 函数创建临界区。应用程序在进入临界区之前必须调用 EnterCriticalSection 函数,并且在退出临界区之时调用 LeaveCriticalSection 函数。这两个例程能确保在多个线程试图并行进入临界区的情况下,在任一时候只允许处理一个线程,而其他线程等候在 EnterCriticalSection 例程。临界区机制比内核同步对象处理起来要快一些,这是因为它直到首次碰到临界区时才分配内核对象。

fiber 是用户态代码,它根据用户定义的调度算法进行调度。一个进程里面可能拥有多个 fiber,就像它拥有多个线程一样。线程与 fiber 之间主要的区别在于线程可以并发执行,但是在同一时间只允许执行一个 fiber,甚至是多处理机的硬件(也是如此)。这种机制被包括在 Windows 2000 里是有助于移植那些旧版本的 UNIX 应用程序,这些程序当时是专为 fiber 执行程序模型所编写的。

系统通过调用 ConvertThreadToFiber 或 CreatFiber 来创建 fiber。这两个函数的主要区别是 CreatFiber 函数在创建了 fiber 之后并没有开始执行。如要开始执行的话,应用程序必须调用 SwitchToFiber 函数。应用程序通过调用 DeleteFiber 函数可终止一个 fiber。

21.7.3 进程间通信

Win32 应用程序进行进程间通信的一种方法是通过共享内核对象。另一种方法是通过传递消息,这是一种特别受 Windows GUI 应用程序欢迎的方法。

一个线程要向另一个线程或者是一个窗口发送一条消息可以通过调用函数的方法，这些函数分别是：PostMessage，PostThreadMessage，SendMessage，SendThreadMessage，SendMessageCallback。提交(posting)一条消息的与发送(sending)一条消息的区别在于提交例程是非同步的：它们会立即返回，而调用线程事实上都不知道什么时候发送消息。传递例程是同步的，在发送和处理完消息之前它们会阻塞调用程序。

另外线程在发送一条消息时，也可以发送附带消息的数据。进程拥有各自独立的地址空间，所以数据必须复制到目的地。系统通过调用 SendMessage 函数来复制数据，通过调用此函数来发送一条 WM_COPYDATA 类型的信息，并附带一个 COPYDATASTRUCT 数据结构。该数据结构含有要传输数据的长度与地址。消息一旦发送，Windows 2000 就把数据复制到一个新的内存区块，并给予接收进程新区块的虚拟地址。

与 16 位的 Windows 环境不同，每个 Win32 线程都拥有它自己的输入队列，可以从该队列接收信息。(所有输入都以消息的形式接收。)这种结构比起 16 位的 Windows 的共享输入队列更加可靠，因为有了独立的队列，一个出错应用程序不会阻止其他应用程序的输入。如果一个 Win32 的应用程序不能调用 GetMessage 函数来处理位于其输入队列上的事件，那么这个队列将被填满，在 5 s 之后，系统将给该程序标上“没有响应”的符号。

21.7.4 内存管理

Win32 API 为应用程序使用内存提供了以下几种方法：虚拟内存，内存映射文件，堆，及线程本地存储器。

应用程序调用 VirtualAlloc 函数保留或者提交虚拟内存，调用 VirtualFree 函数回收或者释放内存。这些函数使得应用程序可以指定在内存的什么地方分配虚拟地址。它们能够在多个内存页面范围内进行操作，而且已分配区域的启动地址必须大于 0x10000。请参见图 21.14 中显示的函数的例子。

```
...
//allocate 16 MB at the top of our address space
void * buf = VirtualAlloc(0,0x1000000, MEM_RESERVE|MEM_TOP_DOWN,
                          PAGE_READWRITE);
//commit the upper 8 MB of the allocated space
VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT, PAGE_READWRITE);
//do some stuff with it
//decommit
VirtualFree(buf + 0x800000,0x800000, MEM_DECOMMIT);
//release all the allocated address space
VirtualFree(buf,0, MEM_RELEASE);
...
```

图 21.14 分配虚拟内存的代码段

进程可以通过调用 VirtualLock 函数把一些已提交的页面锁定在物理内存。进程可以

锁定的页面数量最大可达 30，除非是进程首先调用了 SetProcessWorkingSetSize 函数增加了最大的工作集大小。

应用程序使用内存的另一种方法是通过内存把一个文件映射到它的地址空间。内存映射对于两个进程共享内存也不失为一种简便的方法：两个进程都把同一文件映射到它们的虚拟内存。内存映射是一个多级进程，正像你能参考到的图 21.15 中的例子。

```
...
//open the file or create it if it does not exit
HANDLE hfile = CreateFile("somefile",GENERIC_READ|GENERIC_WRITE,
  FILE_SHARE_READ|FILE_SHARE_WRITE,NULL,OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL,
  NULL);
//create the file mapping 8MB in size
HANDLE hmap = CreateFileMapping(hfile,PAGE_READWRITE,
  SEC_COMMIT,0,0x800000,"SHM_1");
//get a view to the space mapped
void *buf = MapViewOfFile(hmap,FILE_MAP_ALL_ACCESS,0,0,0×800000);
//do some stuff with it
//unmap the file
UnmapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);...
```

图 21.15 一个文件的内存映像的代码段

如果进程想映射一些地址空间只是为了与另一个进程共享某一内存区域，那么就不需要什么文件。进程可以用一个 0xffffffff 的文件句柄和一个特定的大小来调用 CreatFileMapping 函数。文件映射对象的结果可以通过以下方式来共享：继承，名字查找，复制。

应用程序使用内存的第三种方法是堆。Win32 环境中的堆只是保留地址空间的一个区域。当一个 Win32 的进程被初始化时，就新建了一个带有 1 MB 的**缺省堆**(default heap)。因为许多 Win32 函数使用缺省堆，所以对堆的访问是同步的以防止堆空间的分配数据结构被并发的多进程破坏。Win32 提供了几个堆管理函数，这样进程就可以分配和管理一个私有堆。这些函数是 HeapCreate，HeapAlloc，HeapRealloc，HeapSize，HeapFree，及 HeapDestroy。Win32 API 还提供了 HeapLock 和 HeapUnlock 函数，这样线程就可以对堆进行互斥访问。与 Virtuallock 函数不一样的是，这些函数只能执行同步操作；它们不能把页面锁定到物理内存当中。

应用程序使用内存的第四种方法是线程本地存储器机制。那些依赖于全局或者静态数据的函数在一个多线程环境下通常不能正确地工作。例如，C 运行期函数 strtok 在分析一个字符串的同时，使用一个静态变量来跟踪它的当前位置。为使两个并发线程正确执行 strtok 函数，它们需要分解 *current position* 变量。线程本地存储器机制分配全局存储器是基于每一个线程。它同时提供了创建线程本地存储器的动态与静态的两种方法。动态方法请参见图 21.16。

```
//reserve a slot for a variable
DWORD var_index = TlsAlloc();
//set it to some value
TlsSetValue(var_index,10);
//get the value back
int var = TlsGetValue(var_index);
//release the index
TlsFree(var_index);
```

图 21.16 动态线程本地存储代码

为使用一个线程本地的静态变量，应用程序应该声明如下的变量来确保每个线程都拥有它自己的私有拷贝：

```
__declspec(thread) DWORD cur_pos = 0;
```

21.8 小 结

微软公司设计的 Windows 2000 是一个具有可扩展性、可移植性的操作系统，是一个可以利用新技术和硬件优势的操作系统。Windows 2000 支持多种操作环境和对称多处理技术。提供基本服务的内核对象的使用、对客户服务计算的支持，使得 Windows 2000 可以支持多种广泛的应用环境。例如，Windows 2000 可以运行为 MS-DOS、Win16、Windows 95、Windows 2000以及(或者是)POSIX 编译的程序。它提供了虚拟内存、完整的高速缓存技术以及抢占式调度方法。Windows 2000 提供了比以往 Microsoft 操作系统提供的更为健壮的安全模式，并包含了国际化的功能。Windows 2000 运行在类型广泛的计算机上，因此用户可以选择和更新硬件来匹配他们的预算及性能需求，而不需要改变他们运行的应用程序。

习题二十一

21.1 在 Windows NT 中为什么把图形代码从用户态转移到核心态会降低系统的可靠性？这种降级违反了 Windows NT 的哪一个原始的设计目标？

21.2 Windows 2000 VM 管理器分配内存使用的是一个二级进程。请确定几种方法中哪一种是有用的。

21.3 请论述 Windows 2000 中使用的特殊页表结构的一些优点和缺点。

21.4 页面错误可能发生在访问(a) 一个虚拟内存；(b) 一个共享的虚拟地址，请问页面错误的最大数量分别是多少？大多数处理器提供什么硬件机制来减少数量？

21.5 Windows 2000 中一个原型页表项的用途是什么？

21.6 高速缓存管理器必须采取什么步骤把数据复制到高速缓存中并从高速缓存中拷贝出来。

21.7 在 VDM 中运行 16 位的 Windows 应用程序涉及到的主要问题是什么？请标出 Windows 2000

为这些问题的每一个所选用的解决方案。对于每个方案，至少要列出一个缺点。

21.8 Windows 2000 在运行一个使用 64 位地址空间的进程时需要做什么样的改动？

21.9 Windows 2000 拥有一个集中式高速缓存管理器。这种高速缓存管理器的优点和缺点是什么？

21.10 Windows 2000 应用了一个"包驱动"的 I/O 系统。请论述为什么赞成或者是反对 I/O 的这个包驱动方案。

21.11 假设有一个 1 TB 的主存储器数据库。你可以使用 Windows 2000 中的哪种机制去访问这个数据库？

推荐读物

Solomon 和 Russinovich[2000]概述了 Windows 2000 的有关内容，并对系统内部以及系统组件的技术细节进行了较详细的描述。Tate[2000]是一份关于 Windows 2000 使用的很好的参考手册。The Microsoft Windows 2000 Server Resource Kit(Microsoft)[2000b]是关于 Windows 2000 使用和部署的六卷本帮助手册。The Microsoft Developer Network Library (Microsoft-2000a])每个季度都会出版，它为 Windows 2000 以及其他 Microsoft 产品提供了丰富的信息资源。

Iseminger[2000]为 Windows 2000 活动目录提供了一份很好的参考。Richter[1997]详细讨论了如何利用 Win32 API 进行程序设计的问题。Silberschatz 等[2001]中有关于 B+树的一个很好的讨论。

第二十二章　Windows XP

微软公司 Windows XP 操作系统是 32/64 位抢占式多任务操作系统，可用于 AMD K6/K7，Intel IA32/IA64 及其后代微处理器。Windows XP 不但是作为 Windows NT/2000 的后一代，还可取代 Windows 95/98 操作系统。系统的主要目的是安全性、可靠性、使用方便、Windows 和 POSIX 应用程序的兼容性、高性能、可扩展性、可移植性和国际支持。这里将讨论 Windows XP 的主要目的、使其使用十分方便的层次结构、文件系统、网络和编程接口。

22.1 历　　史

在 20 世纪 80 年代中期，微软公司和 IBM 公司合作开发 OS/2 操作系统。它是用汇编语言编写的，可用于 Intel 80286 单处理器系统。在 1988 年，微软公司决定重新开始开发一个"新技术"（New Technology，NT）可移植操作系统，并使其同时支持 OS/2 和 POSIX 应用程序接口（Application Programming Interface，API）。在 1988 年 10 月，Dave Cutler，DEC VAX/VMS 操作系统的设计师，被请来负责开发这一新操作系统。

最初，NT 开发小组将 OS/2 API 作为基本环境，但是在开发过程中，由于 Windows 3.0 的流行，NT 转而采用 32 位的 Windows API（或 Win32 API）。NT 的第一版为 Windows NT 3.1 和 Windows NT 3.1 高级服务器（当时，16 位的 Windows 版本是 3.1）。Windows NT 4.0 版采用了 Windows 95 用户接口并集成了因特网 WWW 服务器和浏览器软件。另外，用户接口子程序和所有图形代码都移到内核中以提高性能，但也降低了可靠性。虽然以前 NT 版本移植到了其他微处理器结构，但是出于商业原因，Windows 2000 于 2000 年 2 月发布，停止了对非 Intel（及其兼容产品）的支持。Windows 2000 在 Windows NT 上做了重要改变。它增加了活动目录（基于 X.500 的目录服务）、更好的网络功能和笔记本支持、即插即用设备的支持，分布式文件系统和更多处理器与内存的支持。

在 2001 年 10 月，作为 Windows 2000 桌面操作系统的升级和 Windows 95/98 的替代产品，发布了 Windows XP。在 2002 年，发布 Windows XP 的服务器版（后称为 Windows. NET Server）。Windows XP 更新了图形用户接口，使其充分利用了最近硬件的发展技术，并具有容易使用的特征。另外也增加了许多特性以修补应用程序和操作系统本身的问题。与 Windows 2000 相比，Windows XP 提供了更好的网络与设备支持（包括零配置无线、即时消息、流媒体、数字图像/视频）、对桌面处理器和大规模多处理器的更好的性能改善、更高的可

靠性和安全性。

Windows XP 采用 C/S 体系结构，以实现多种操作系统环境如 Win32 和 POSIX，这些子系统为用户级进程。子系统结构允许改善操作系统环境，而并不影响其他环境的兼容性。

Windows XP 是多用户操作系统，并通过分布式服务或通过 Windows 终端服务器的多个 GUI 实例，以支持并发访问。Windows XP 的服务器版支持来自 Windows 桌面系统的并发终端服务器会话。终端服务器的桌面版多路复用了键盘、鼠标和监视器，为每个登录用户提供虚拟会话终端。这种特性，称为快速用户切换，允许用户互相抢占计算机终端而不必退出。

Windows XP 是 Windows 支持 64 位的第一版本。NT 文件系统(NTFS)和许多 Win32 API 在所有合适的地方都使用了 64 位，以便支持更大的地址。

Windows XP 桌面版有两个版本。Windows XP Professional 是高级用户工作和家用的主要操作系统。Windows XP Personal 是针对从 Windows 95/98 迁移过来的普通用户，它提供了更好的可靠性和使用方便性，但没有提供一些高级特性以便与活动目录无缝联结，也不能运行 POSIX 应用程序。

Windows .Net Server 系列采用了与 Windows XP 桌面版本相同的内核，但是增加了许多功能如 WWW 服务器、打印服务器、支持集群系统和支持大的数据中心机器。大数据中心机器采用了 32 个 IA32 处理器和 64 GB 的内存，或 64 个 IA32 处理器和 128 GB 的内存。

22.2 设计原则

Microsoft 关于 Windows XP 的设计目标包括安全性、可靠性、Windows 和 POSIX 应用程序的兼容性、高性能、可扩展性、可移植性和国际支持。

22.2.1 安全性

Windows XP 安全性目的要求超出了 Windows NT 4.0 满足美国政府的 C－2 安全认证的要求，C－2 安全认证表示对有缺陷软件和恶意攻击的中等级别的保护。广泛代码检查与测试和高级自动分析工具一起用来鉴别和研究潜在的安全隐患。

22.2.2 可靠性

Windows 2000 是微软公司当时所发布的最为可靠与稳定的操作系统。这种可靠性是由于源代码的成熟、深入高强度的系统测试和驱动程序内严重错误的自动检测。Windows XP 的可靠性要求更为严格。Microsoft 采用了人工和自动的代码检查，查到了超过 63 000 行未被测试所发现的有可疑之处的代码，并检查这些区域以确认代码确实是正确的。

Windows XP 扩展了驱动程序的验证来捕捉更为细微的错误，改善了捕捉用户级代码的编程错误的工具，并使第三方应用程序、驱动程序和设备接受更为严格的验证过程。而

且，Windows XP 增加了新工具以监视计算机的正常运行包括用户遇到问题之前先下载补丁。Windows XP 的可靠性还包括如下方面的改进，如更容易使用的 GUI、更为简单的菜单、更容易处理常见任务。

22.2.3 Windows 和 Posix 应用的兼容性

Windows XP 不仅是 Windows 2000 的升级，而且是 Windows 95/98 的替代品。Windows 2000主要关注商业应用程序的兼容性。Windows XP 要求更为广泛的兼容性，如支持可运行于 Windows 95/98 的用户应用程序。应用程序兼容性实现困难，因为每个应用程序检查特定版本的 Windows，可能依赖于特定 API 的实现，也可能有以前版本的 Windows 所隐藏的一些错误，还有其他依赖性。

Windows XP 在应用程序与 Win32 API 之间引入了兼容层。该层使得 Windows XP 看起来与原来版本的 Windows 一样。Windows XP，与 Windows NT 一样，也支持 16 位应用程序，这是由将 16 位 API 调用转换成 32 位 API 调用的 thunking layer 来实现的。同样，64 位版本的 Windows XP 也提供了 thunking layer，将 32 位 API 调用转换成 64 位 API 调用。Windows XP的 Posix 支持也改进了。现在有一个新版本的 POSIX 子系统，称为 Interix。绝大多数当今的 UNIX 兼容软件可以在 Interix 下不加修改就能编译和运行。

22.2.4 高性能

Windows XP 也被设计用来提高桌面系统（主要受 I/O 性能所限）、服务器系统（CPU 通常为瓶颈）和多线程和多处理器环境（加锁与缓存管理对于可伸缩性尤为重要）的性能。对于 Windows XP，高性能目标的重要性与日俱增。在发布 Windows 2000 之际，采用 Compaq 硬件，Windows 2000 和 SQL 2000 的组合达到了当时最高的 TPC－C 号码。

为了满足性能要求，NT 采用了各种技术如异步 I/O 、优化的网络协议（如优化分布数据的加锁、请求的批处理）、基于内核的图形、高级文件系统数据的缓存。通过设计内存管理和同步算法以充分考虑缓存流水线与多处理器的性能。

Windowx XP 通过降低关键函数的代码路径长度、采用更好算法与数据结构、采用 NUMA（Non-Uniform Memory Access）内存加色、实现扩展更好的加锁协议如排队自旋锁，以进一步提高性能。新的加锁协议有助于降低系统总线周期、无锁链表和队列、原子读－写－改操作的使用和其他高级加锁技术。

组成 Windows XP 的各子系统可通过 LPC 来互相通信，以提高消息传递性能。除了在内核调度程序内执行外，Windows XP 的各子系统的线程可以为更高优先级线程所抢占。因此，系统能更快地响应外界事件。另外，Windows XP 为对称多处理而设计；在多处理器计算机上，多个线程可同时运行。

22.2.5 可扩展性

可扩展性是操作系统利用计算机技术的程度。为了利用这些技术，开发人员采用了分层结构以实现 Windows XP。Windows XP 执行体运行在内核或保护模式下，并提供基本系统服务。在执行体之上，多个服务器子系统运行于用户模式下。其中包括模拟不同操作系统的环境子系统。因此，为 MS-DOS、Microsoft Windows、POSIX 所编写的程序可运行于 Windows XP的适当子系统中。由于模块化结构，可以在不影响执行体的情况下，增加其他环境子系统。另外，Windows XP 在 I/O 系统中加入了可加载驱动程序，这样新文件系统、新 I/O 设备类型、新网络可以在系统运行时加入到系统中。Windows XP 与 Mach 操作系统一样采用了 C/S 结构，并支持由 OSF 所定义的 RPC 形式的分布式处理。

22.2.6 可移植性

如果一个操作系统能从一个硬件结构移到另一个硬件结构，且只需要少量修改，那么这种操作系统就称为可移植的。与 UNIX 操作系统一样，Windows XP 的主要部分是用C/C++来编写的。绝大多数处理器相关代码通过 DLL 而加以分开，通常称为硬件抽象层(hardware-abstraction layer，HAL)。DLL 文件可以映射到进程地址空间，以便使 DLL 的函数为进程所用。Windows XP 内核的上层依赖于 HAL 接口，而不是底层硬件，从而进一步提高了可移植性。HAL 直接操作硬件，从而使得 Windows XP 的其他部分与在运行于各平台的硬件差异相分开。

虽然由于市场原因 Windows 2000 在发布时只支持 Intel IA32 兼容产品，但是在发布之前，Windows 2000 也在 DEC Alpha 平台加以测试来确保其可移植性。微软公司认识到多平台开发和测试的重要性，因为事实上，兼容性的维护是一个要么用或要么不用的问题。

22.2.7 国际支持

Windows XP 被设计成为国际化的或能在多国使用的操作系统。它通过 NLS API 以支持多种语言。NLS API 提供了专用函数以格式化日期、时间和货币，以便与各国习惯相一致。字符串操作考虑了不同字符集。UNICODE 是 Windows XP 所自带的字符集。Windows XP 通过先将 ANSI 字符转换成 UNICODE 字符再加以处理(8 位到 16 位转换)来支持 ANSI 字符。系统文本串保存在资源文件中，可以进行替换以支持不同语言的本地化。Windows XP 可以同时支持多种语言，这对于使用多国语言的企业或个人来说是重要的。

22.3 系统组成

Windows XP 结构是分层的模块化的结构，如图 22.1 所示。主要层次包括 HAL、内核

和可执行体，这些均运行于保护模式下；一组子系统和服务运行于用户模式下。用户模式子系统分成两类。一类为环境子系统，以模拟不同操作系统；另一类为保护子系统，以提供安全功能。这种结构的主要优点是模块之间的互操作简单。本节将讨论这些层次和子系统。

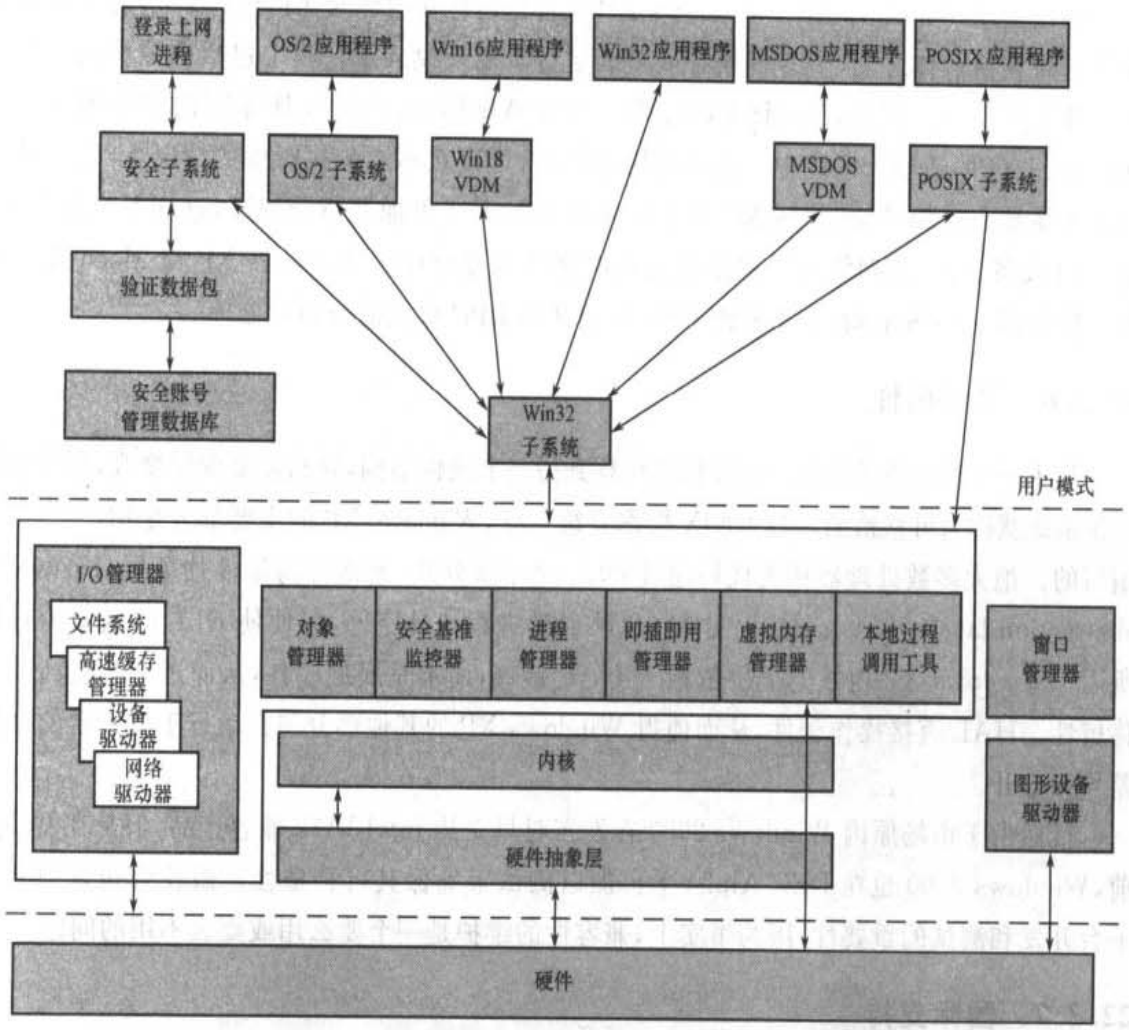


图 22.1 Windows XP 模块化结构

22.3.1 硬件抽象层

HAL 是一个软件层，用来为操作系统的上层隐藏硬件差异，以提高 Windows XP 的可移植性。HAL 有一虚拟机接口，可为内核调度程序、可执行体和设计驱动程序所使用。这种方法的一个优点是每个设备驱动程序只需要一个版本，即它可运行于各种硬件平台，而无需移植驱动程序。HAL 也支持对称多重处理。设备驱动程序映射设备并直接访问它们，但是映射内存的管理、配置 I/O 总线、设置 DMA 和处理母板等有关细节，都是由 HAL 接口提供的。

22.3.2 内核

Windows XP 内核为执行体和子系统提供了基础。内核驻留于内存，其执行不会被抢占。它有 4 个主要责任：线程调度、中断和异常处理、低级处理器同步、掉电后恢复。

内核是面向对象的。Windows 2000 的对象类型是系统定义的数据类型，它具有一些属性（数据值）和一组方法（例如，函数或操作）。对象是对象类型的实例。内核通过使用一组对象（属性存储内核数据而方法执行内核活动）执行其工作。

1. 内核调度程序

内核调度程序为执行体和各子系统提供了基础。绝大多数调度程序不会调出内存，其执行也不会被抢占。其主要责任包括线程调度、同步原语实现、定时器管理、软件中断（异步与延迟程序调用）和异常处理。

2. 线程和调度

与许多操作系统一样，Windows XP 也采用了进程和线程概念以描述执行代码。每个进程都有虚拟地址空间和用于初始化线程的信息，如基础优先级和单个或多个处理器的亲合度。每个进程有单个或多个线程，每个线程是由内核所调度的可执行单元。每个线程都有其自己的调度状态，包括实际优先级、处理器亲合和 CPU 使用信息。

线程有 6 个可能状态：就绪、备用、运行、等待、过渡和终止。就绪表示等待运行。最高优先级线程移到就绪状态，意味着它是下一个要运行的线程。对于多处理器系统，每个进程都有一个线程处于备用状态。当一个线程在处理器上执行时，就处于运行状态。它会一直运行，直到它为更高优先级线程所抢占，然后终止，其分配时间片结束，或阻塞于调度对象如表示 I/O 完成的事件。当线程等待调度对象时，就处于等待状态。新线程等待执行所需资源时，就处于过渡状态。当线程完成执行时，就进入了终止状态。

调度程序采用了 32 级优先权方案以确定线程执行顺序。优先级分成两类：可变类和实时类。可变类包括 0 到 15 优先级的线程，实时类包括 16 到 31 优先级的线程。调度程序为每个优先级都采用了队列，并从高到低遍历队列集合，直到找到可以运行的线程为止。如果某个线程具有特定处理器亲合而该处理器不可用，那么调度程序跳过它，继续查找能在可用处理器运行的线程。如果找不到就绪线程，那么调度程序就执行一个称为空闲线程的特殊线程。

当一个线程的时间片用完时，时钟中断会向处理器产生一个时间片结束延迟过程调用（deffered procedure call，DPC），以便重新调度处理器。如果被抢占线程属于可变优先级类，那么其优先级会降低。但是，优先级不会低于基本优先级。降低线程优先级往往能限制偏于计算的线程的 CPU 消耗。当可变优先级线程从等待状态释放出来时，调度程序会加大其优先级。加大量取决于线程所等待的设备；例如，等待键盘的线程会得到较大程度的增加；而等待磁盘的线程会得到中等程度的增加。这种策略能够给予使用鼠标和窗口的交互

线程更高的优先级，从而能让 I/O 约束线程使得 I/O 设备一直忙，能让偏于计算的线程使用后台的空闲 CPU 周期。这种策略为许多分时操作系统如 UNIX 所使用。另外，与用户活动 GUI 窗口相关的线程会得到优先级提升以增加其响应速度。

当线程进入就绪或等待状态，或线程终止，或应用程序改变线程的优先级或处理器亲合，那么就会出现调度。如果在较低优先级线程运行时有一个更高优先级的实时线程变为就绪，那么较低优先级线程就被抢占。这种抢占给予实时线程在需要时更高优先的 CPU 使用。Windows XP 不是实时操作系统，因为它不能保证实时线程在某一特定时间限制内可以开始执行。

3. 同步原语的实现

操作系统的关键数据结构是作为对象管理的，它们使用共同的工具来分配、引用计数和安全。调度对象控制系统内的调度和同步。这些对象例子包括事件、变异、人工干预、信号量、进程、线程与定时器。事件对象用来记录事件发生和与某一动作相同步。通告事件用信号通知所有等待线程，同步事件用信号通知一个等待线程。对象变异提供了内核模式或用户模式的带有拥有者概念的互斥。对象人工干预，只能在内核模式下使用，提供了没有死锁的互斥。信号量对象作为计数器或门来控制访问某个资源的线程数量。线程对象是由内核调度程序调度的实体，它与进程对象相关联(进程对象有虚拟地址空间)。定时器对象用来跟踪时间，发出超时信号以表示操作过长或需要中断或需要调度一个周期性的活动。

许多调度对象可以通过在用户模式下用 open 所返回的句柄来访问。用户模式代码轮询或等待句柄，以便与其他线程和操作系统同步(参见 22.7.1 小节)。

4. 软件中断：异步过程调用

调度程序实现了两种类型的软件中断：异步过程调用(APC)和延迟过程调用。异步过程调用可以进入执行线程以调用一个过程。APC 可用来执行新线程、终止进程和发送通知以表示异步 I/O 已完成。APC 在特定线程上等待，允许系统在进程关联环境中执行系统和用户代码。

5. 软件中断：延迟过程调用

延迟过程调用(DPC)用来延迟中断处理。在处理完所有阻塞设备中断进程之后，中断服务程序(ISR)通过加上 DPC 来调度剩余处理。调度程序以与设备中断相比较低的优先级来调度软件中断，所以 DPC 不会阻塞其他 ISR。

除了延迟设备中断处理外，调度程序采用 DPC 处理定时器，并在调度时间片段之后抢占执行。

DPC 的执行防止线程在当前处理器调用，也让 APC 不发出 I/O 完成信号。这样做是为了使 DPC 不需要过多时间来完成。另外一种方法是，调度程序维护一个工作线程池。ISR 和 DPC 将有关项加到工作线程上。DPC 子程序有不能出现页错误、不能调用系统服务

和采用可能会阻塞执行的其他动做等的限制。与 APC 不一样，DPC 子程序并不对进程所执行的处理器作任何假设。

6. 异常与中断

内核调度程序也为由硬件或软件所产生的异常和中断提供了陷阱处理。Windows XP 提供了多个与体系结构无关的异常，包括：内存访问违例、整数上溢、浮点数上溢或下溢，整数被 0 除、浮点数被 0 除、非法指令、数据不对齐、特权指令、读页错误、访问违例、超出换页文件量、调试中断点和调试单步。

陷阱处理程序处理简单的异常。更为复杂的异常处理是由内核异常调度程序来执行的。异常调度程序创建一个包括异常原因的异常记录，并查找一个异常处理程序来处理它。

当异常在内核模式下发生时，异常调度程序只不过简单地调用一程序来定位异常处理程序。如果没有找到，那么就出现了一个致命系统错误，这时用户会看到表示系统出错的声名狼籍的“蓝屏死机”。

对于用户模式进程，异常处理就更为复杂，因为环境子系统（POSIX）会为由它所创建的每个进程设置一个调试端口和异常端口。如果调试端口已注册，那么异常处理程序会向该端口发送一个异常。如果调试端口没有找到或者不处理所收到的异常，那么调度程序试图查找一个合适的异常处理程序。如果找不到一个异常处理程序，那么调试程序会再次用来捕捉调试错误。如果调试程序没有运行，那么会向进程的异常端口发送一个消息，让环境子系统能有机会转换这一异常。例如，POSIX 环境将 Windows XP 异常消息转换成 POSIX 信号，再发送给引起异常的线程。最后，如果这些都不行，那么内核就简单地终止包含有引起异常的线程的进程。

内核的中断调度程序通过调用中断服务程序 ISR（由调备驱动程序所提供）或内核处理子程序处理中断。中断是由中断对象表示的，中断对象包括处理中断所需要的所有信息。采用中断对象便于让中断服务程序与中断相关联，而不需要直接访问中断硬件。

不同处理器结构，如 Intel 或 DEC Alpha，有不同类型和数量的中断。为了可移植性，中断调度程序将这些硬件中断映射到一个标准集合。中断有优先级高低之分并按优先级顺序处理。Windows XP 有 32 个中断请求级别（Interrupt ReQuest Level，IRQL）。其中 8 个为内核所保留使用；其他 24 个通过 HAL 表示硬件中断（虽然绝大多数 IA32 系统只用了 16 个）。Windows XP 中断按图 22.2 来定义。

内核采用了**中断分配表**（interrput dispatch table），以将每个中断级别与处理程序相关联。对于多处理器计算机，Windows XP 为每个处理器保留了独立的中断向量表，每个处理器的 IRQL 可以独立设置以屏蔽中断。所有等于或低于处理器 IRQL 的中断会被阻塞，直到 IRQL 被内核级线程或从中断处理中返回的 ISR 降低为止。Windows XP 充分利用了这一属性，并采用软件中断，以发送 APC 和 DPC，执行系统功能如将线程与 I/O 完成相同步，以启动线程调度或处理定时器。

中断级别	中断类型
31	机器检验或总线出错
30	掉电
29	处理器间通知(请求另外一个处理器执行；例如，分派一个进程或更新TLB)
28	时钟(用于跟踪时间)
27	简档
3—26	传统PC IRQ硬件中断
2	分派和延期的过程调用(DPC)(内核)
1	异步过程调用(APC)
0	被动

图 22.2 Windows XP 中断请求级别

22.3.3 执行体

Windows XP 执行体提供一组服务，以供环境子系统所使用。服务可分成如下几组：对象管理器、虚拟内存管理器、进程管理器、本地过程调用工具、I/O 管理器、安全引用监视器、即插即用和安全管理器、注册、启动。

1. 对象管理器

Windows XP 采用了一组通用接口(供用户模式程序使用)以管理内核模式实体。Windows XP 称这些实体为*对象*(object)。管理实体的可执行体部分称为**对象管理器**(object manager)。每个进程都有一个对象表用来保存跟踪进程所使用的对象的信息。用户模式代码通过称为句柄的隐性值(可从许多 API 得到)来访问这些对象。对象句柄的创建也包括复制已有句柄(可能来自同一进程或可能来自不同进程。)

对象的例子有信号量、mutex、事件、进程和线程。这些称为可*调度对象*。线程能在内核调度程序中阻塞，并等待直到某个对象接收到信号。进程、线程和虚拟内存 API 使用进程和线程句柄来区别所操作的进程或线程。其他对象例子包括文件、区间、端口和各种内部 I/O 对象。文件对象用来维护文件和设备的打开状态。区间用来映射文件。打开文件使用文件对象描述。本地通信端点是由端口对象实现的。

对象管理器维护 Windows XP 内部名称空间。与 UNIX 不同(UNIX 采用了文件系统的系统名称空间)，Windows XP 采用了抽象名称空间，并将文件系统连接为设备。

对象管理器提供了接口，用于定义对象类型和对象实例，将名称转换成对象，维护抽象名称(通过内部目录和符号链接)和管理对象创建与删除。对象通常在内核模式下采用引用计数来管理，而在用户模式下采用句柄来处理。然而，有的内核模式部件也使用了与用户模式相同的 API，因此也用句柄来操作对象。如果句柄在当前进程生存期之后还需要存在，那么它就标记为内核句柄，并保存在系统进程的对象表中。抽象命名空间在重启的时候将不保留，但是可以通过保存在注册表中的配置信息、即插即用设备的发现和系统组成的对象创

建等来进行构造。

Windows XP 执行体允许给予任一对象一个**名称**。一个进程 P_1 可以创建一个命名对象，而另一进程 P_2 可以打开这一对象的句柄，从而能与进程 P1 进行共享。进程也能通过复制句柄以共享对象，这时对象不必命名。

名称可以是暂时的也可以是永久的。永久名称表示一个实体如磁盘驱动器，这种实体即使在没有进程访问它的时候也能存在。暂时名称只有在进程有对象句柄时才存在。

对象名称与 MS DOS 和 UNIX 的文件路径名一样组织。名称空间目录由目录对象表示，以包括目录内所有对象的名称。对象名称空间可以通过增加设备对象(以表示包含文件系统的卷)来加以扩展。

对象由一组虚函数来操作，每个对象类型提供了这些函数的实现：create，open，close，delete，query name，parse 和 security。后三个函数需要解释一下。

- 当一个线程有一个对象的引用，但需要知道其名称时，可以调用 query name。
- 根据对象名称，对象管理用 parse 来搜索相应对象。
- 调用 security 以对所有对象操作进行安全检查，如进程打开或关闭一个对象时，对安全描述符进行修改，复制一个对象句柄。

过程 parse 是用来扩展抽象名称空间来包括文件。将路径转换成文件对象是从抽象名称空间的根开始的，路径名部分是由字符'\'而不是 UNIX 的'/'分隔的。每个部分可从名称空间的当前分析目录下加以查找。名称空间的内部节点可以是目录或符号链接。如果找到叶对象而没有剩余的路径名，就返回叶对象。否则，对剩余路径名，再调用叶对象分析过程。

分析过程只能用于少数对象，如属于 Windows GUI 的、配置管理器(注册表)和表示文件系统的设备对象。

用于设备对象类型的分析过程分配一个文件对象，并初始化用于文件系统的打开或创建 I/O 操作。如果成功，文件对象的域将被填充来描述文件。

总之，文件路径名用于遍历对象管理器名称空间，以将原来绝对路径名转换成(设备对象，相对路径名)对。这个数据对再通过 I/O 管理器传递给文件系统，用来填充文件对象。文件对象本身没有名称，但可通过句柄来引用。

UNIX 文件系统有**符号链接**(symbolic link)，从而允许同一文件具有多个别名。由 Windows XP 对象管理器所实现的**符号链接对象**(symbolic link object)用于抽象名称空间，但不能使用文件系统的文件别名。即使如此，符号链接仍然非常有用。它们可以用来组织名称空间，类似于 UNIX 设备目录的结构。驱动器字母是符号链接，可以被再次映射，方便用户或管理员。

驱动器字母是这样一种情况，Windows XP 的抽象名称空间不是全局的。每个登录用户都有自己的驱动器字母集合，以便避免互相干扰。另一方面，终端服务器会话共享同一会话的所有进程。BaseNamedObjects 包括由绝大多数应用程序建创的命名对象。

虽然名称空间不能通过网络直接可见，但是对象管理的 parse 方法可用于帮助访问位于远程系统的命名对象。当一个进程试图打开位于远程计算机上的对象时，对象管理器调用对应于网络重定向设备对象的分析方法。这将导致 I/O 操作，从而通过网络访问文件。

对象是**对象类型**的实例。对象类型指定实例如何分配、数据域的定义、用于所有对象的虚函数标准集合的实现。这些函数实现的操作包括将名称映射到对象、关闭、删除以及安全检查。

对象管理器为每个对象保存两个计数。指针计数是对对象不同引用的个数。引用对象的保护模式代码必须有一个对象引用以确保对象在使用时不会被删除。句柄计数是引用对象的句柄表条目的个数。每个句柄也反映在引用个数中。

当关闭对象句柄时，会调用对象关闭子程序。对于文件对象，这种调用会引起 I/O 管理器在关闭最后一个句柄时进行清除操作。清除操作通知文件系统，某文件已不再为用户模式所访问，所以它可以删除共享限制、区域锁和其他与相应打开子程序相关的状态。

每个用于关闭的句柄从指针计数中删除一个引用，但是内部系统部件可能仍有其他引用。当删除最后引用时，删除子程序会让 I/O 管理器向文件系统发送一个关闭文件对象的操作。这会引起文件系统释放为该文件对象而分配的任何内部数据结构。

在删除临时对象的过程完成时，就从内存中清除对象。通过请求对象管理器保持一个额外对象引用，可以让对象成为永久对象（即直到系统重启之前）。因此即使当对象管理器之外的最后引用删除时，永久对象也不会被删除。当永久对象再次成为暂时之际，对象管理器就删除引用。如果这是最后引用，那么就删除对象。永久对象很少，主要用于设备、驱动器字母映射和目录与符号链接对象。

对象管理器(object manager)的工作是管理所有对象的使用。当一个线程需要使用一个对象时，它会调用对象管理器的 open 方法，以获得对象的引用。如果通过用户模式 API 打开对象，那么就会将引用插入进程的对象表中，并返回一个句柄。

进程获得句柄有很多方法：创建对象，打开现存对象，从另一引用接受一个复制句柄，或从父进程继承句柄（类似于 UNIX 进程获得文件描述符）。这些句柄都保存在进程**对象表**中。对象表的每个条目包括对象的访问权限和句柄能否为**子进程**所继承的状态。当一个进程终止时，Windows XP 自动关闭进程的所有打开句柄。

句柄(handle)是所有各种对象的标准接口。与 UNIX 的文件描述符一样，一个对象句柄是相对于一个进程而言的惟一标识符，它表示访问和操作一个系统资源的能力。句柄可以在一个进程内或多个进程之间进行复制。当创建子进程或实现不属于进程执行关联时，可在进程之间进行句柄复制。

由于对象管理器是生成对象句柄的惟一实体，所以很自然它也进行安全性检查。在进程试图打开对象时，对象管理器会检查该进程是否有访问该对象的权限。对象管理器也可强制限额，如通过对所有引用对象所占用内存进行计量，当累加计量超过进程限额时拒绝更

多分配内存，可限制进程使用的最大内存。

当登录进程验证一个用户时，也为用户进程附加了访问标记。该访问标记包括各种信息如安全 ID、组 ID、特权、主组和缺省访问控制链表。用户可访问的服务和对象是由这些属性决定的。

控制访问标记与进行访问的每个线程相关联。通常线程没有标记，并缺省为进程标记，但是服务通常需要标记来为客户执行代码。Windows XP 允许线程用客户标记，暂时扮演角色。因此，线程标记不必与进程标记一样。

在 Windows XP 中，每个对象都用访问控制链表(包括安全 ID 和允许访问权限)加以保护。当一个线程试图访问一个对象时，系统将线程访问标记的安全 ID 与对象的访问控制链表进行比较，以确定是否允许该访问。只有在打开对象时，才进行检查，所以在打开之后，就不可能否认访问。在内核模式下运行的操作系统部件可绕开访问检查，因为内核模式是认为可以信赖的。内核模式代码必须避免安全弱点，如在一般进程中创建用户模式对象后而使安全检查失效。

通常，对象创建者决定对象的访问控制列表。如果没有特别指定，那么对象类型的打开子程序就会缺省地选择一个，或者从用户访问标记对象中获得一个缺省列表。

访问标记有一个域用于控制对象访问的审计。审计过的操作和其用户 ID 会一起记录到系统安全日志上。系统管理员监视这一日志以发现是否有闯入和访问保护对象的企图。

2. 虚拟内存管理器

虚拟内存管理器(virtual-memory manager，VM)是管理虚拟地址空间、物理内存分配和调页的执行体部分。VM 管理器的设计假定底层硬件支持虚拟到物理的映射、调页机制以及多处理器系统透明的缓存一致性，并且允许多个页表条目映射到同一物理帧。Windows XP 的 VM 管理器采用了基于页面的管理方案，在 IA32 兼容处理器上页面大小为 4 KB，而在 IA64 兼容处理器上页面大小为 8 KB。分配给进程的数据页如不在物理内存中，那么可能在磁盘的**调页文件**或直接映射到本地或远程文件系统的普通文件。页也可标记为按需置零。

对于 IA32 处理器，每个进程有一个 4 GB 的虚拟地址空间。上部 2 GB 对所有进程都是相同的，被内核模式的 Windows XP 用来访问操作系统代码和数据结构。每个进程所特有的内核模式区域的重要部分有**页表映射**(page table self-map)、**超空间**(hyperspace)和**会话空间**(session space)。硬件采用物理页帧来引用进程的页表。VM 管理器将页表映射到进程地址空间的一个 4 MB 区域，所以它们可以通过虚拟地址进行访问。超空间将当前进程的工作集合信息映射到内核模式的地址空间。

会话空间用于同一终端服务器会话的所有进程而不是系统的所有进程之间，共享 win32k 和其他会话有关的驱动程序。下部 2 GB 为每个进程所特有，可以被用户模式和内核模式线程所访问。有的 Windows XP 的配置只为操作系统保留了 1 GB 的空间，允许进程有 3 GB 的地址空间。按 3 GB 模式运行系统大大地降低缓存在内核中的数据量。然而，对

于需要自己管理 I/O 的大规模应用程序如 SQL 数据库，还是值得牺牲一定缓存以换取大用户模式地址空间的优点。

Windows XP VM 管理器采用了分两步来分配用户内存。第一步保留进程虚拟地址空间的一部分。第二步通过赋值虚拟内存空间(物理内存或调页文件的空间)来提交分配。Windows XP 通过对提交内存加上限额以限制进程所使用的虚拟内存。当进程不再使用虚拟内存时，进程会释放内存以供其他进程使用。用来保留虚拟地址空间和提交虚拟内存的 API 使用进程对象的句柄作为参数。这允许一个进程控制另一进程的虚拟内存。环境子系统就是这样管理其客户进程的内存的。

为提高性能，VM 管理器允许特权进程锁住物理内存中的某些选定的页，以确保这些页不会调出到调页文件中。进程也可分配生物理内存，并将它映射到其虚拟地址空间。IA32 处理器的 PAE(Physical Address Extension，物理地址扩展)特征可以使系统的物理内存达到 64 GB。这种内存不能同时映射到进程的地址空间，但是 Windows XP 通过 AWE (Address Windowing Extension)API 使用，这些 API 分配物理内存并将进程的地址空间的虚拟地址的区域映射到物理内存的一部分。AWE 功能主要用于非常大的应用程序如 SQL 数据库。

Windows XP 通过定义**区域对象**(section object)来实现共享内存。在获得区域对象句柄之后，进程将其所需要内存映射到其地址空间。这部分称为**视图**。通过每次一个区域，进程重定义对象视图从而获得对整个对象的访问。

进程可按许多方式控制共享内存区域对象的使用。区域的最大值可以加以约束。区域可通过磁盘空间加以扩展如系统调页文件或普通文件(**内存映射文件**)。区域可以有个基础，意味着区域在所有试图访问它的进程中，以同一虚拟地址出现。最后，区域中的保护页可以设成只读、读写、读写执行、只可执行、无访问和写时复制。最后两个保护设置需要加以解释。

• 无访问页如果被访问会产生异常。例如，异常用来检查出错程序是否超过了一个数组的结尾。用户模式内存分配器和设备验证器所使用的特别内核分配器可以配置成：将每次分配映射到页尾加上无访问页，以便检测缓冲溢出。

• 写时复制机制为 VM 管理器增加了物理内存的使用效率。当两个进程需要各自的对象拷贝时，VM 管理器将一个共享拷贝映射到虚拟内存并激活内存区域的写时复制属性。如果一个进程试图修改写时复制页中的数据，那么 VM 管理就为该进程复制一个私有拷贝。

Windows XP 虚拟地址转换采用了多级页表。对于 IA32 处理器，如果没有使用 PAE，那么每个进程都有一个**页目录**(page directory)，包括 1 024 个大小为 4 B 的页目录条目(page directory entry，PDE)。每个 PDE 指向一个页，包括 1 024 个大小为 4 B 的**页表条目**(page table entry，PTE)。每个 PTE 指向一个 4 KB 的物理内存的**页帧**(page frame)。一个进程总页表大小为 4 MB，因此 VM 管理器可以根据需要换出单个页表。关于这种结构，请

参见图 22.3。

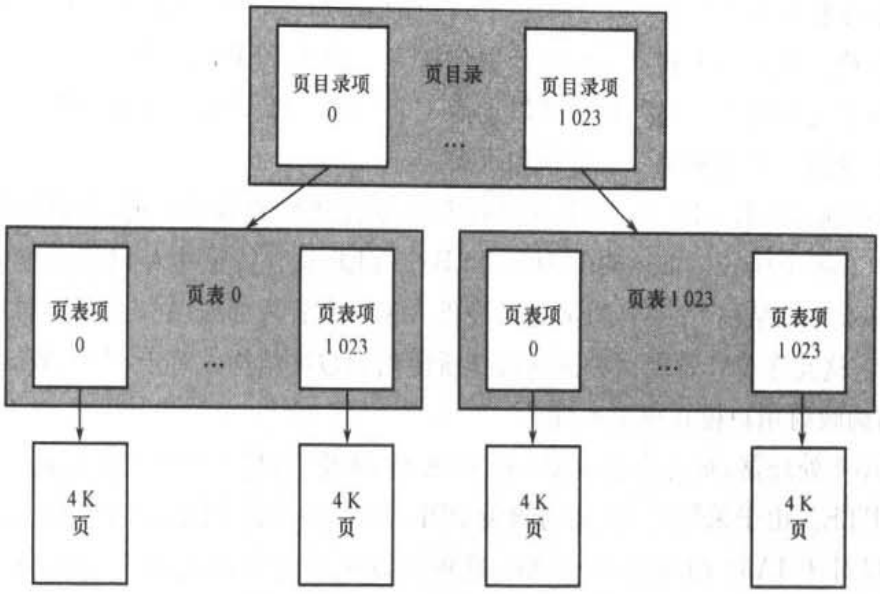


图 22.3 页表层次

页目录和页表是通过物理地址由硬件来引用的。为了提高性能，VM 管理器自己将页目录和页表映射到一个 4 MB 的虚拟地址的区域。自我映射允许 VM 管理器将虚拟地址转换成相应的 PDE 和 PTE 而无需额外内存访问。当改变进程关联环境时，只需要改变一个页目录条目以映射新进程的页表。出于许多理由，硬件要求每个页目录和页表都只占有一个页。因此，单个页所能容纳的 PDE 或 PTE 的数量决定了虚拟地址如何转换。

下面描述了在 IA32 兼容处理器(没有使能 PAE)上，虚拟地址如何转换成物理地址。10 位值可能表示 0 到 1 023 的所有值。因此，10 位值可以选择页目录或页表的任一条目。这个属性用于将虚拟地址指针转换成物理内存的字节地址。32 位虚拟内存地址分成三个部分，如图 22.4 所示。虚拟地址的头 10 位用做页目录的索引。这一地址选择一个页目录条目(PDE)，它包括了页表的物理页帧。内存管理单元(MMU)采用了虚拟地址的后 10 位以从页表中选择一个 PTE。该 PTE 指定了物理内存的页帧。虚拟地址的剩余 12 位作为页帧中特定字节的偏移。MMU 通过将 PTE 的 20 位与虚拟内存的低 12 位相合并，就得到了一个指针以指向物理内存的特定字节。因此，32 位 PTE 有 12 位可用来描述物理页的状态。IA32 硬件将 3 个位留给操作系统专用。其余位表示该页是否访问过或写过、缓存属性、访

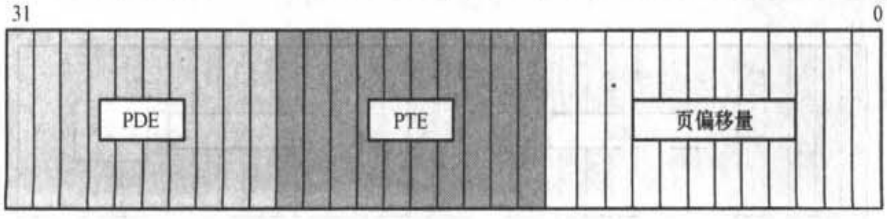


图 22.4 IA32 虚拟地址和物理地址之间的转换

问模式、页是否全局和 PTE 是否有效。

IA32 处理器如果加上 PAE，则能使用 64 位的 PDE 和 PTE 来表示更大的 24 位页帧号码域。因此，第二级页目录和页表只能分别包括 512 PDE 和 PTE。为了提供 4 GB 的虚拟地址空间，则需要使用另一层的页目录以包括 4 个 PDE。32 位虚拟地址转换用 2 位表示顶级目录索引，使用 9 位表示第二页目录和页表。

为了避免通过查找 PDE 和 PTE 来转换每个虚拟地址的额外开销，处理器使用了**翻译后备缓冲器**(translation-lookaside buffer，TLB)。TLB 包含快速缓存以便将虚拟页映射到 PTE。与 IA32 体系结构不一样(TLB 是由硬件 MMU 来管理的)，IA64 调用软件陷阱子程序以进行转换。这允许 VM 管理器灵活地选择所使用的数据结构。对于 IA64，Windows XP 采用了三级结构映射用户模式虚拟地址。

对于 IA64 处理器，页大小是 8 KB，但 PTE 有 64 位，所以一个页只能包括 1 024(10 位)个 PDE 或 PTE。由于采用了 10 位的顶级 PDE，10 位的二级 PDE，10 位的页表和 13 位的页偏移，所以对于 IA64 的 Windows XP，进程的虚拟地址空间的用户部被分为 8 TB(43 位)。当前版本的 Windows XP 的 8 TB 限制小于 IA64 处理器的能力，但是这是在需要处理 TLB 未击中的内存引用数量和所支持的用户地址空间大小之间的一个平衡。

物理页可以有 6 个状态：有效、空闲、清零、备用、修改、坏和过渡。*有效*页为活动进程所使用。*空闲*页是没有被 PTE 引用的页。*清零*页是已清零的、可满足调页清零需要的空闲页。*修改*页则已经被进程所写，在分配给其他进程之前需要发送到磁盘。*备用*页上的信息已经保存在磁盘上。这些可能是没有修改过的页，已经写到磁盘上的修改过的页，或为改善局部性而提前读入的页。*坏*页不可用是因为硬件已检测到错误。最后，*过渡*页是正在将磁盘数据读入到物理内存中的分配帧。

当 PTE 有效位等于 0 时，VM 管理器定义其他位的格式。无效页可有一些状态，用 PTE 的位来表示。页文件的页如果没有调入，就标记为按需清零。文件通过区间对象映射编码区间对象的指针。已写到页文件的页包括足够信息来查找磁盘上的页，等等。

页文件 PTE 的真正结构如图 22.5 所示。PTE 包括 5 位用于页保护、20 位用于页文件偏移、4 位用于选择调页文件和 3 位以描述页状态。页文件 PTE 标记成对 MMU 为无效虚拟地址。因为可执行代码和内存映射文件已有一个磁盘拷贝，所以它们在调页文件中不需要空间。如果这样的一个页不在物理内存中，那么 PTE 结构就会按如下所述：最重要的位用于表示页保护，后 28 个位用于索引系统数据结构以表示一个文件和在这个文件中的页偏

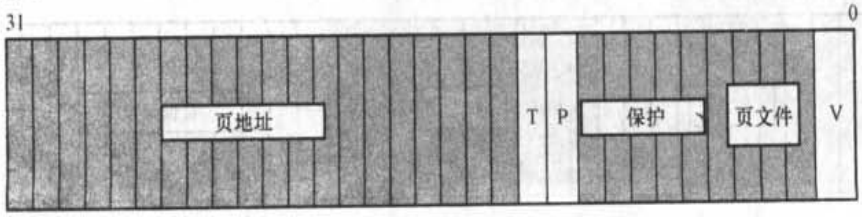


图 22.5 页文件的页表项(有效位为 0)

移，低 3 位表示页状态。

无效虚拟地址也可有一些状态，以供调页算法使用。当从一个进程工作集合中删除一页时，它就移到修改链表（以便写磁盘）或直接写到备用链表。如果写到备用链表而尚未成为空闲页，那么在再次需要时，该页可重新使用而不必从磁盘中读入。当可能时，VM 管理器使用空闲 CPU 周期来把空闲链表上的页清零，并把它们移到清零链表。过渡页已被分配了物理页，在 PTE 标记为有效之前等待调页 I/O 的完成。

Windows XP 使用区间对象来描述在进程之间共享的页。每个进程都有其自己的虚拟页表，但是区间对象也包括一组页表以表示主（或原型）PTE。当进程页表的 PTE 标记为有效时，它指向包括页的物理帧。当共享页标记为无效时，PTE 改为指向与区间对象关联的原型 PTE。

与区间对象相关的页表在创建时或整理时，是虚拟的。惟一需要的原型 PTE 是用来描述那些有当前映射视图的页。这大大地改善了性能，且允许有效使用内核虚拟地址。

原型 PTE 包括帧地址以及保护和状态位。因此，进程首次访问共享页会产生页错误。在首次访问之后，之后的访问按通常方式进行。如果一个进程对 PTE 中标记为只读的页进行写时复制，那么 VM 管理器就复制一个页，标记为可写，这样该进程事实上就不再有共享页了。共享页决不会出现在页文件中，但可在文件系统中找到。

VM 管理器跟踪**页帧数据库**（page-frame database）内的所有物理内存的帧。系统内的每个物理内存帧都有一个条目。条目指向 PTE，而 PTE 又指向帧，这样 VM 管理器可维护页的状态。未被有效 PTE 所引用的帧根据页类型如清零、修改或空闲而形成链表。

如果共享物理页标记为对某个进程有效，那么该页就不能从内存中删除。VM 管理器跟踪帧数据内的每个页的有效 PTE 的个数。当数量为零时，物理页就可在其内容写到磁盘上后（如果标记为脏页）重新使用。

当出现页错误时，VM 管理器会找一个物理页以容纳数据。对于按需清零页，第一选择是找一个已清零的页。如果这样的页没有，那么就从空闲链表或备用链表中选择一页，并在使用前进行清零。如果出错页已经标记为转变，那么它可能正在从磁盘中读入，或正在取消映射或整理且仍然在备用或修改链表上。该线程或者等待 I/O 完成或者（在后一种情况）重新从适当链表中收回页。

否则，必须发出 I/O 从页文件或文件系统中读入页。VM 管理器试图从空闲链表或备用链表中分配可用页。位于修改链表的页不能使用，直到其写回磁盘并转到备用链表上。如果没有页可用，那么线程就阻塞，直到工作集合管理器调整内存的页，或其他进程取消映射物理内存的某个页。

Windows XP 为每个进程采用 FIFO 替代策略以取回超过其最小工作集合的页。Windows XP跟踪处于最小工作集合的每个进程的页出错。当进程开始时，它赋给了 50 页的缺省工作集合。VM 管理器根据时间会替代或调整进程的工作集合。页生存期是由没有

PTE 时发生的调整周期的次数来决定的。根据页 PTE 的修改位是否已设置，调整页可移到备用或修改链表中。

VM 管理器不只调入马上需要的页。研究发现线程的内存引用往往有局部性属性；当使用一个页，可能不久也要使用其相邻页(想一下对数组进行迭代或顺序获取线程可执行代码)。由于局部性，当 VM 管理器调入页时，它也调入一些相邻页。这种提前获取往往降低了页错误的总的数量。写也合并在一起以降低独立 I/O 操作的数量。

除了管理提交内存，VM 管理器还管理每个进程的保留内存或虚拟地址空间。每个进程都有一个相关的张开树以描述使用虚拟地址的区域和用途是什么。这允许 VM 管理器按需要调入页表。如果出错地址的 PTE 不存在，那么 VM 管理器会搜索进程的 VAD(virtual address descriptor，虚拟地址描述符)树以查找地址，使用这一信息以填充缺少 PTE，并获取该页。在有的情况下，页表的页本身就不存在，必须由 VM 管理器透明地分配并初始化它。

3. 进程管理器

Windows XP 进程管理器提供了创建、删除和使用进程、线程和作业等服务。它没有关于父子关系或进程层次的知识；这些细节由拥有这些进程的环境子系统处理。进程管理器也不涉及进程调度，只是在进程和线程创建时设置其优先级和亲合力。线程调度在内核调度程序进行。

进程包括一个或多个线程。进程本身可组成称为**作业对象**的大单元，作业对象允许对 CPU 使用、工作集合大小和处理器亲合力加以限制，来同时控制多个进程。作业对象用来管理大数据中心机器。

在 Win32 环境中，进程创建的一个例子如下。当 Win32 应用程序调用 CreateProcess 时，就向 Win32 子系统发送一个消息来通知它正在创建一个进程。原来进程内的 CreateProcess 就调用 NT 执行体内进程管理器的 API，从而真正创建进程。进程管理器调用对象管理器创建进程对象，返回对象句柄给 Win32。Win32 再次调用进程管理器为该进程创建一个线程，并返回新进程和线程的句柄。

用于操作虚拟内存、线程和复制句柄的 Windows XP API 会使用进程句柄作为参数，以便子系统可以为新进程执行操作，而不需要在新进程的关联环境中直接执行。一旦创建了新进程，就创建最初线程，并向线程发送一个 APC，以促使用户模式装入程序开始执行。该装入程序为 ntdll.dll，这是个连接库，自动地映射到每个新创建的进程中。Windows XP 也支持 UNIX fork()风格的创建进程，以便支持 POSIX 环境子系统。虽然 Win32 环境从客户进程中调用进程管理器，但是 POSIX 采用 Windows XP API 的交叉进程性质从子系统进程中创建新进程。

进程管理器也实现对线程的 APC 的排队和发送。系统用 APC 来开始线程执行，完成 I/O、终止线程和进程及加上调试程序。用户模式代码也可对线程的 APC 进行排队以发送信号形式的通知。为了支持 POSIX，进程管理器提供 API 向线程发送警告，通过系统调用

释放锁。

进程管理器的调试程序支持包括暂停和重启线程，并且从暂停模式开始创建线程。还有其他进程 API 可以读取或设置线程寄存器关联环境，及访问另一进程的虚拟内存。

线程可在当前进程中创建；线程也可注入另一进程中。在执行体内，现有线程可暂时附加到另一进程。这种方法可以为工作线程所使用，这样工作线程可按照需要工作请求的关联环境来执行。

进程管理器也支持假冒。在具有属于某个用户的安全标记的进程中执行的线程，可设置属于另一用户的线程有关标记。这一功能对 C/S 计算模型是基本的，服务器需要代替不同安全 ID 的客户而执行。

4. 本地过程调用工具

Windows XP 实现采用了客户机－服务器模型。环境子系统是服务器，以实现特定操作系统的个性。客户机－服务器模型除了用于实现环境子系统，还实现了各种操作系统服务。安全性管理、假脱机打印、WWW 服务、网络文件系统、即插即用和许多其他特征都采用这种模型实现。为了降低内存占用，多个服务通常组织成少数几个进程，这些进程再利用用户模式线程池功能来共享线程和等待消息（参见 22.3.3.3 一节）。

在单个机器上的客户与服务进程之间，操作系统采用 LPC 工具来传递请求和结果。尤其是，LPC 用来从各种 Windows XP 子系统中请求服务。LPC 在许多方面类似于 RPC 机制（RPC 机制为许多操作系统用来通过网络进行分布式处理），但是 LPC 为单个系统使用而进行了优化。Windows XP 的 OSF RPC 的单机实现通常采用 LPC 作为传输机制。

LPC 是消息传递机制。服务器进程发布一个全局可见的连接端口对象。当一个客户需要子系统的服务时，它便打开一个子系统连接端口对象的一个句柄，并对该端口发送一个连接请求。服务器请求一个频道，并返回一个句柄给客户机。该频道有一对私有通信端口：其中一个用于服务器到客户机消息，另一个用于客户机到服务器消息。通信频道支持回调机制，这样客户机和服务器在等待回答时也可接受请求。

当创建 LPC 频道时，必须指定三种消息传递技术。

1. 第一个技术适用于小消息（不到数百字节）。在这种情况下，端口消息队列用做中间存储，消息从一个进程复制到另一个进程。

2. 第二个技术用于更大消息。在这种情况下，每个通道要创建一个共享内存区间对象。通过消息队列端口发送的消息（包括指针和大小信息）来引用区间对象。这避免了复制大消息。发送者将消息放入共享区间，接收者可直接看到。

3. LPC 消息传递的第三个技术使用 API 以对进程的地址空间进行直接读/写。LPC 提供函数和同步技术以便服务器访问客户中的数据。

Win32 窗口管理器使用自己形式的消息传递，这与执行体 LPC 工具相独立。当客户机请求连接使用窗口管理器消息，服务器建立 3 个对象：专门服务器线程处理请求、64 KB 区

间对象和事件对对象。*事件对对象*(event-pair object)是同步对象,当客户线程已经复制一个消息给 Win32 服务器或情况相反时,Win32 子系统将用它来提供通知。区间对象传递消息,事件对对象执行同步。窗口管理器消息传递有多个优点。区间对象消除消息复制,这是由于它表示共享内存区域。事件对对象消除了采用端口对象传递包括指针和长度的消息的额外开销。专门服务器线程消除了确定哪个客户线程调用服务器的额外开销,这是由于每个客户线程只有一个服务器线程。最后,内核给予这些专门服务器线程优先调度以改善性能。

5. I/O 管理器

I/O 管理器(I/O manager)负责文件系统、设备驱动程序和网络驱动程序。它跟踪装入了何种设备驱动程序、过滤驱动程序和文件系统,也管理 I/O 请求的缓冲。它与 VM 管理器一起提供内存映射文件I/O,控制 Windows XP 的缓存管理器,从而处理整个 I/O 系统的缓存。I/O 管理器主要是异步的。同步 I/O 通过显式等待 I/O 操作的完成而提供。I/O 管理器提供多种模型的异步 I/O 完成包括设置事件、向发起线程发送 API 以及允许从多个线程中选择单个线程来处理 I/O 完成的 I/O 完成端口。

每个设备的驱动程序按列表安排(称为驱动器或 I/O 堆栈)。I/O 管理器将其收到的请求转换成标准形式称为 **IRP**(I/O Request Packet, I/O 请求包)。接着它将 IRP 转递给堆栈中的第一个设备驱动程序处理。每个设备驱动程序处理完 IRP 之后,它会调用 I/O 管理器以便再传递给堆栈中的下一个驱动程序,或者完成对 IRP 的操作(如果所有处理都已完成)。

I/O 完成可能出现在不同于原来 I/O 请求的关联环境中。例如,如果驱动程序执行其 I/O 操作的部分并被强制阻塞的时间过长,那么它可以将 IRP 排队到工作线程,方便在系统关联中继续处理。在原来的线程中,驱动程序返回一个状态来表示 I/O 请求正在处理,以便线程可与 I/O 操作并行执行。IRP 也可在中断处理程序中处理,并在任意关联环境中完成。因为有的最后处理可能需要在发起 I/O 的关联中进行,I/O 管理器使用 APC 在发起线程的上下文中进行最后 I/O 完成处理。

设备堆栈模型非常灵活。当构造设备驱动程序时,各种驱动程序有机会将自己作为**过滤驱动程序**(filter-driver)嵌入进来。过滤驱动程序有机会检查且可能修改每个 I/O 完成。安装管理、分区管理和磁盘分条/镜像都是通过过滤驱动程序(在堆栈的文件系统之下执行)实现功能的例子。文件系统过滤驱动程序在文件系统之上执行,已用于实现许多功能如层次存储管理、远程启动的单个文件和动态格式转换。第三方也使用文件系统过滤驱动程序来实现病毒检测。

Windows XP 的设备驱动程序是按 WDM(Windows Driver Model, Windows 驱动程序模型)规范来编写的。这个模型列出设备驱动程序的要求,包括如何对层过滤驱动程序进行过滤,共享公用代码以处理电源和即插即用请求,构造正确的取消逻辑,等等。

由于 WDM 非常复杂,为新硬件设备编写完整 WDM 设备驱动程序可能需要大量工作。不过,由于端口/微端口模型并不是必要的。对于相似类型的设备,如声卡、SCSI 设备、以太

网控制器，这些同一类型的设备可以共享该类的公共驱动程序称为**端口驱动程序**。端口驱动程序实现了一个类型的标准操作，然而可调用设备特定程序（称为**微端口驱动程序**（miniport driver））以实现设备特定功能。

6. 缓存管理器

对许多操作系统，缓存是由文件系统实现的。但是，Windows XP 提供了中央控制的缓存工具。**缓存管理器**（cache manager）与 VM 管理器密切工作，从而为 I/O 管理器控制的所有设备提供缓存服务。Windows XP 缓存是基于文件和生块（raw block）的。

根据系统现有空闲内存的多少，缓存大小动态改变。记住进程地址空间的上部 2 GB 由系统区域组成；它在所有进程的关联环境中可用。VM 管理器将不到一半的空间分配给系统缓存。缓存管理器将文件映射到这个地址空间，使用虚拟管理器的能力来处理文件 I/O。

缓存按 256 KB 的块来划分。每个缓存块可以容纳文件的一个视图（即内存映射区域）。每个缓存块按 VACB 来描述，**VACB**（virtual-address control block）存储了视图的虚拟地址和文件偏移，及使用视图的进程数量。VACB 驻留在由缓存管理器维护的一个数组中。

对于每个打开文件，缓存管理器维护一个独立 VACB 索引数组以描述整个文件的缓存。这一数组对每个 256 KB 的文件块都有一个条目；这样，一个 2 MB 的小文件就可能有一个 8 条目的 VACB 索引数组。如果那部分文件存于缓存中，那么 VACB 索引数组的条目指向 VACB；否则，就为空。当 I/O 管理器收到一个文件的用户级读请求时，I/O 管理器就发送一个 IRP 给设备驱动程序堆栈（文件所驻留的）。文件系统试图查找缓存管理器所请求的数据（除非请求显式要求非缓存的读）。缓存管理器计算哪个文件的 VACB 索引数组的条目对应于请求的字节偏移。该条目或指向缓存的视图，或无效。如果有效，那么缓存管理器分配一个缓存块（和 VACB 数组的相应条目），并将该视图映射到缓存块中。缓存管理器接着试图将映射文件的数据复制到调用者的缓存。如果拷贝成功，那么操作就完成。

如果拷贝失败（由于页出错），那么就会引起 VM 管理器向 I/O 管理器发送一个非缓存的读请求。I/O 管理器沿着驱动程序堆栈向下发送另一请求，这次的请求调页操作绕过缓存管理器，数据直接从文件读入并存入为缓存管理器所分配的页。在完成之后，VACB 就指向这一页。现存于缓存的数据被复制到调用者缓冲，原来 I/O 请求就完成了。图 22.6 显示了这些操作的概观。

当可能时，对缓存文件的同步操作 I/O 按照**快 I/O 机制**（fast I/O mechanism）来处理。这种机制与普通基于 IRP 的 I/O 相并行，直接通过调用进入驱动程序堆栈而不是传递 IRP。由于没有涉及 IRP，操作不会阻塞过长时间，不能在工作线程上排队。因此，当操作系统到达文件系统，它调用缓存管理器，如果信息不在缓存中操作就会失败。I/O 管理器接着试图采用普通 IRP 路径操作。

基于内核级的读操作相似，只不过数据可直接在缓存中处理，而不必复制到用户空间的缓冲中。为使用文件系统元数据（描述文件系统的数据结构），内核使用缓存管理器的映射

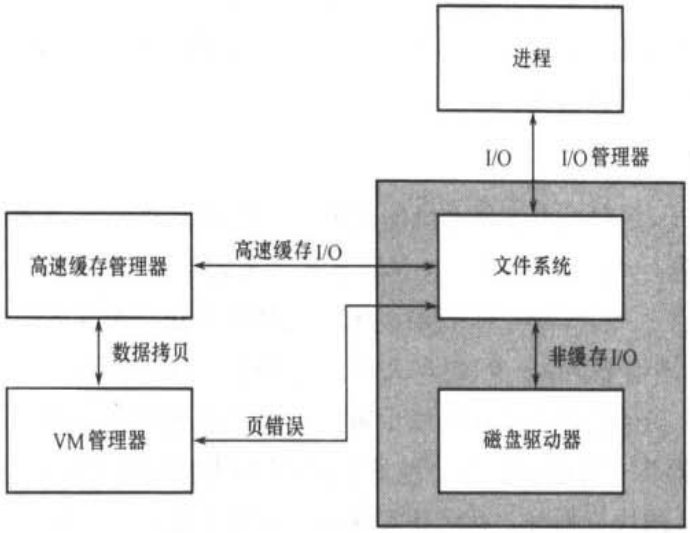


图 22.6 文件 I/O

接口读入元数据。为了修改元数据,文件系统使用缓存管理器钉住接口。**钉住**(pinning)一页会将该页锁在物理内存帧中,这样 VM 管理器就不能移动或调出该页。在更新元数据之后,文件系统会请求缓存管理不再钉住页。修改页标记为脏页,以便 VM 管理器将该页清出到磁盘。元数据保存在普通文件中。

为了改善性能,缓存管理保留一个读请求的较小记录,从这一记录可以预测将来请求。如果缓存管理器发现了以前三个读请求的模式,如顺序向前或向后访问,那么这会提前将数据读入到缓存(在下个请求为应用程序提交以前)。这样,应用程序会发现其数据已在缓存中,而不必等待磁盘 I/O 完成。Win32 API 中的 OpenFile 和 CreateFile 函数可以传递参数 FILE_FLAG_SEQUENTIAL_SCAN 标记,这是一个对缓存管理器的提示以便提前访问 192 KB。通常,Windows XP 按 64 KB 或者 16 页来执行 I/O 操作;因此,这一提前读是普通量的三倍。

缓存管理器也负责通知 VM 管理器以清出缓存的内容。缓存管理器缺省行为是写回缓存:它先将 4 s 或 5 s 内就能完成的操作进行积累,然后再唤起写线程。当需要写通缓存,进程在打开文件时可设置标记,或者进程可调用显式缓存清出函数。

快写进程可能会用完所有空闲页,而此时缓存写线程还未被唤醒将页清出到磁盘。缓存写通过如下方法以防止进程淹没系统。当空闲缓存内存量变低时,缓存管理器会暂时阻塞进程试图写数据,并唤起缓存写线程以将页清出到磁盘。如果快写进程实际是个网络文件系统的网络重定向器,阻塞太久会引起网络传输超时重传。这种重传会浪费网络带宽。为了防止这种浪费,网络重定向器可让缓存管理器限制缓存内的写积压。

因为网络系统需要在磁盘和网络接口之间移动数据,缓存管理器也提供了 DMA 接口用于直接移动数据。直接移动数据避免了通过中间缓存的数据复制。

7. 安全引用监控器

对象管理器对系统实体的集中管理有助于采用统一机制，对所有用户可访问的实体进行运行时访问确认和审计检查。当进程打开对象句柄时，**安全引用监控器**(Security Reference Monitor，SRM)就检查进程安全标记和对象访问控制链表，从而确定进程是否有必需的权限。

SRM 也负责控制安全标记的特权。用户需要特权来执行文件系统的备份或恢复操作，或作为管理员跳过某些检查，调试程序等。标记也可标记为在某些特权方面被加以限制，这样它们就不能访问可被绝大多数用户所使用的对象。限制标记主要用来限制不可靠代码执行所带来的损坏。

SRM 的另一责任是记录安全审计事件。安全级 C2 要求系统有能力检测和记录对系统资源的访问企图，以便能更容易地跟踪未被授权的访问。由于 SRM 负责进行访问检查，所以它在安全事件记录中产生了大量的审计记录。

8. 即插即用与电源管理器

操作系统通过使用**即插即用管理器**(plug-and-play manager，PnP)识别和适应硬件配置的变化。为了应用 PnP 技术，设备及其驱动程序必须都要支持 PnP 标准。在系统运行时，PnP 管理器自动识别安装设备并检测设备的变化。它会跟踪设备所使用的资源及其可能要使用的资源，也负责装入适当驱动程序。硬件资源(主要为中断和 I/O 内存区域)管理的目的之一是确定一个硬件配置，以便所有设备都能正常工作。

例如，如果设备 B 可能使用中断 5 而设备 A 可使用 5 或 7，那么 PnP 管理将 5 赋给 B 而将 7 赋给 A。对于以前版本，用户可能需要撤下设备 A 并重新配置使用中断 7，然后再安装设备 B。因此，在安装新硬件之前，用户必须研究系统资源，查找或记住哪些设备使用哪些硬件资源。PCMCIA 卡、USB、IEEE1394、Infiniband 和其他热插拔设备的大量流行也要求支持动态资源配置。

PnP 管理器按如下方式处理这一动态配置。首先，它从总线驱动程序那里获得设备链表(如 PCI、USB)。它装入所安装的驱动程序(如果需要可安装)，并对每个设备的适当驱动程序发送一个 add-device 请求。PnP 管理器算出最佳资源分配，向每个设备驱动程序发送一个 start-device 请求及其设备的分配资源。如果一个设备需要重新配置，那么 PnP 管理器会发送一个 query-stop 请求，用于请求驱动程序暂停设备。如果驱动程序可暂停设备，那么就完成所有未决操作，并阻止新操作开始执行。接着，PnP 管理器发送一个 stop 请求；然后它可用另一个 start-device 请求来重新配置设备。

PnP 管理器也支持其他请求，如 query-remove。在用户准备弹出 PCCARD 设备时，可使用这一请求，其执行过程类似于 query-stop。当设备出错或者更为可能的是用户没有停止就直接移去 PCCARD 时，就使用 surprise-remove 请求。请求 remove 告诉驱动程序停止使用设备并释放该设备所分配的所有资源。

Windows XP 支持高级电源管理。虽然这些功能可用于家用系统，降低电源消耗，但是其主要应用是为使用方便和扩展膝上型电脑的电源寿命。系统及其单个设备在不使用时可进入低耗电模式(备用或睡眠模式)，这样电池主要用于物理内存数据的保持。当网络有数据包要接收，或调制解调器的电话响，或用户打开膝上型电脑或按下软电源按钮时，系统便可进入正常模式。Windows XP 将物理内存内容存入磁盘，并关机进入休眠；在执行需要继续时，可将系统恢复过来。

系统还支持另外一些策略以降低电源消耗。在 CPU 空闲时，Windows XP 不是让处理器处于忙循环，而让系统进入一个需要更少电源消耗的状态。如果 CPU 使用过低，Windows XP 会降低 CPU 时钟速度，从而节省了大量电源。

9. 注册表

Windows XP 将其配置信息保存在称为**注册表**(registry)的内部数据库中。注册数据称为 **hive**。对于系统信息、缺省用户选项、软件安全和安全等，都有各自的 hive。由于**系统 hive** 的信息需要用来启动系统，所以注册表管理器是作为执行体的部件来实现的。

每次系统成功启动时，它会将系统 hive 保存为 *last-known-good*。如果用户安装了软件如设备驱动程序而导致所产生的系统 hive 不能用于启动，那么用户通常可用 last-known-good 配置来启动。

因安装第三方应用程序和驱动程序而损坏了系统 hive 是比较常见的，为此 Windows XP 有一部件称为**系统恢复**(system restore)，以用来周期性地检测 hive 和其他软件状态如设备驱动程序和配置文件，这样当系统启动时如不能像以前一样正常工作，那么系统可恢复到以前的工作状态。

10. 启动

Windows XP PC 的启动在硬件电源打开时开始，BIOS 开始从 ROM 中执行。BIOS 找到**系统设备**(system device)以便用于启动，从磁盘开头装入并执行启动导入程序。该装入程序对文件系统格式足够地知道，从系统设备的根目录中装入 NTLDR 程序。NTLDR 用于确定哪个设备包含有操作系统。接着，NTLDR 从**引导设备**(boot device)中装入 HAL 库、内核和系统 hive。根据系统 hive，它得知哪些设备驱动程序需要用来启动系统(*启动驱动程序*)并加以装入。最后，NTLDR 开始执行内核。

内核初始化系统并创建两个进程。**系统进程**包括所有内部工作线程，决不会按用户模式运行。第一个用户模式进程为 SMSS，类似于 UNIX 的 INIT 进程。SMSS 进一步系统初始化，包括建立调页文件和装入设备驱动程序，并创建 WINLOGON 和 CSRSS 进程。CSRSS 是 Win32 的子系统。WINLOGON 启动系统其余部分，包括 LSASS 安全子系统和需要运行系统的其他服务。

通过提前装入上次系统启动的磁盘上的文件，系统可优化启动过程。启动磁盘访问模式也可用于重新布局磁盘的系统文件以降低所需要的 I/O 操作的数量。启动系统所需要的

过程可以通过将多个服务组成一个进程来进一步降低。所有这些方法都有助于降低系统启动时间。

另一方面，Windows XP 睡眠和冬眠能力允许用户关掉电源，并在停机处很快重启，这样系统启动时间就不如原来那么重要了。

22.4 环境子系统

环境子系统是建立在 Windows XP 本身的执行体的服务之上的用户态进程，它能让 Windows XP 运行为其他操作系统开发的程序，包括 16 位的 Windows 操作系统、MS-DOS 系统、POSIX 系统。每个环境子系统提供独立的应用程序环境。

Windows XP 把 Win32 子系统作为主要操作环境，因此它用来启动所有进程。当执行一个应用程序时，Win32 子系统就通过 VM 管理器并装入应用程序的可执行代码。内存管理器给 Win32 发回一个状态，告知可执行代码的类型。如果不是 Win32 本身可执行的代码，那么 Win32 将核查合适环境子系统是否已在运行。如果子系统尚未运行，就把它作为用户态进程来启动。然后，该子系统会控制其应用程序的运行。

环境子系统利用 LPC 工具为客户进程提供操作系统服务。Windows XP 体系结构不允许应用程序混合使用不同环境的 API 程序。例如，一个 Win32 的应用程序不能调用一个 POSIX 例程，这是因为一个应用程序只能有一个环境子系统与其相关联。

因为每个子系统是作为一个独立的用户态进程运行的，一个子系统出现崩溃并会不影响其他的子系统。Win32 例外，它提供所有的键盘、鼠标和图形显示能力。如果它不能提供这些功能，系统就失效了。

Win32 环境把应用程序分为两类：要么是基于图形的，要么是基于字符的，该环境中一个*基于字符的应用程序*是一个(认为是)交互输出的基于字符的窗口。Win32 把基于字符的程序输出结果转换成命令窗口的图形表示。这种转换并不难：当需要调用输出程序时，环境子系统调用 Win32 程序以显示文本。因为 Win32 环境为所有基于字符的窗口执行这一功能，所以可在窗口和剪贴板之间转换屏幕文本。这种转换不但适用于 MS-DOS 程序，而且也适用于 POSIX 命令行程序。

22.4.1 MS-DOS 环境

MS-DOS 环境没有其他的 Windows XP 环境子系统那么复杂。它由一个称为**虚拟 DOS 机**(virtual DOS machine，VDM)的 Win32 应用程序提供。由于 VDM 只是一个用户态进程，它的分页和调度与其他的 Windows XP 一样。VDM 的**指令执行单元**(instruction-execution unit)能执行或者模仿 Intel 486 的指令。VDM 还提供程序模仿 MS-DOS ROM BIOS 和"int 21"软件中断服务程序，并拥有用于屏幕、键盘以及通信接口的虚拟设备驱动程

序。VDM 基于 MS-DOS 5.0 源代码;它给予应用程序的内存空间至少有 620 KB。

Windows XP 命令 shell 新建的窗口看起来很像 MS-DOS 环境。可以运行 16 位和 32 位的可执行体。运行一个 MS-DOS 环境时,命令 shell 就启动一个进程执行该项任务。

如果 Windows XP 运行于一个 IA32 兼容处理机,MS-DOS 图形程序将以全屏模式运行,而字符程序可以运行于全屏或者是一个窗口。并不是所有 MS-DOS 应用程序都可运行在 VDM 之下。例如,有些 MS-DOS 程序直接访问磁盘硬件,但是因为磁盘是受限制的(用于保护文件系统),所以就不能运行 Windows XP。总的来说,直接访问硬件的 MS-DOS 应用程序不能在 Windows XP 下运行。

MS-DOS 不是多任务环境,有些应用程序会独占整个 CPU。例如,通过使用过忙的循环体将导致执行过程中时间的推延或者是暂停。Windows XP 调度程序中的优先机制会发现这种推延并自动取消 CPU 的耗费,这导致类似应用程序不能正常运行。

22.4.2 16 位 Windows 环境

Win16 执行程序环境是由一个 VDM 提供的,该 VDM 合并了称做*关于窗口的窗口*(Windows on Windows,用于 16 位应用程序的 WOW32)的额外软件。这一软件提供了 Windows 3.1内核例程,用于 Windows 管理器和 GDI 子例程的 stub 例程。stub 例程可以调用适当的 Win32 子例程转换或*形实转换*(thunking)16 位地址到 32 位地址。依赖于 16 位 Windows 管理器或 GDI 内部结构的应用程序可能无法运行,这是因为底层 Win32 实现当然是有别于真正的 16 位 Windows。

WOW32 可以和其他 Windows XP 的应用程序一起进行多任务工作,但是它在很多方面都像 Widows 3.1:一次只能运行一个 Win16 应用程序,所有程序都是单一线程,驻留在同一地址空间,共享同一输入队列。这些功能意味着应用程序停止接收输入时,将阻塞所有其他的 Win16 应用程序,就像 Windows 3.x 一样。而且一个 Win16 应用程序会因为破坏地址空间而毁损其他的 Win16 应用程序。然而,多个 Win16 环境能共存,这可通过从命令行调用 *start /separate win16application* 命令来实现。

还有少量的 16 位应用程序需要继续运行于 Windows XP 之上,但是其中一些需要共同的安装程序。因此,WOW32 环境需要继续存在,因为有的 32 位应用程序没有它就不能安装在 Windows XP 之上。

22.4.3 IA64 的 32 位 Windows 环境

IA64 的 Windows 环境使用 64 位地址和 IA64 指令集合。为了在这种环境上执行 IA32 程序要求一个辅助层以将 32 位 Win32 调用转换成相应 64 位调用,这与 IA32 之上的 16 位应用程序所要求的一样。因此 64 位 Windows 支持 WOW64 环境。32 位和 64 位 Windows 的实现基本相同,IA64 提供了 IA32 指令的直接执行,这样 WOW64 提供了比 WOW32 更好

的兼容性。

22.4.4 Win32 环境

Windows XP 的主要子系统是 Win32 子系统。它运行 Win32 的应用程序，并管理着所有的键盘、鼠标和屏幕 I/O。因为它控制环境，所以设计得非常健壮。Win32 的几个特点都有助于提高这种健壮性。与 Win16 环境不同，每个 Win32 进程都有它自己的输入队列，窗口管理器把系统的所有输入项调度到适当的进程输入队列，因而一个失效的进程不会阻塞向其他进程的输入操作。

Windows XP 内核也提供抢占式多任务，它可以让用户终止已经失效或者是不再需要的应用程序。在使用之前 Win32 也可检查所有对象，以防止可能因为应用程序试图利用一个失效的和错误的句柄而导致崩溃。Win32 子系统可以在使用对象前核实句柄所指向的对象类型。管理器一直保持着引用计数，以防对象在使用中被删除，也防止对象在删除后被利用。

为了实现与 Windows 95/98 系统的更高层的兼容性，Windows XP 允许用户通过一个特定**薄层**(shim layer)来运行单个应用程序，该薄层可以修改 Win32 API，来使原来的应用程序表现得更加好。例如，有的应用程序需要使用特定版本的系统，而在新系统上却不能运行。通常应用程序有些错误，这些只有在以后的系统实现中才可被发现。例如，在释放内存之后使用它可能会导致内存破坏(如果堆所使用的内存顺序发生改变)，或者应用程序假定某个子程序返回哪些错误或者地址的有效位的数量。采用 Windows 95/98 薄层运行程序可使系统提供与 Windows 95/98 更加接近的行为，但这也会导致性能降低及与其他应用程序的互操作性降低。

22.4.5 POSIX 子系统

POSIX 子系统被设计成为能运行遵循 POSIX.1 标准的 POSIX 应用程序。该标准基于 UNIX 模型。POSIX 应用程序能用 Win32 子系统或其他的 POSIX 应用程序启动。POSIX 应用程序使用 POSIX 子系统的服务器 PSXSS.EXE，POSIX 动态链接库 PSXDLL.DLL 以及 POSIX 控制台会话管理器 POSIX.EXE。

虽然 POSIX 标准并不指定打印技术，但是 POSIX 应用程序还是通过 Windows XP 转向器机制透明地使用打印机。POSIX 应用程序拥有访问 Windows XP 系统中所有文件系统的能力。POSIX 环境在目录树方面实施类似 UNIX 的权限。

由于发布日期的原因，Windows XP 的 POSIX 系统并没有与整个系统一起发布，但是可以另外购买以用于专业桌面系统和服务器。与原来 NT 的版本相比，它提供了与 UNIX 应用程序更高级的兼容性。对于现有常用 UNIX 应用程序，绝大多数可以不加修改地在最新版本的 Interix 上加以编译和运行。

22.4.6 登录与安全子系统

在访问 Windows XP 的对象之前，用户必须通过登录子系统的认证。WINLOGON 负责响应安全关注顺序(Ctrl-Alt-Del)。安全关注顺序是个请求机制，用于防止应用程序被特洛伊木马使用。只有 WINLOGON 才可截获这个顺序以便显示登录屏幕、改变口令和锁住屏幕。为了通过确认，用户必须拥有一个账号以及提供与账号相对应的口令。另外，用户可通过智能卡和个人识别码来登录。

本地安全子系统是一个进程，它能生成在系统中表示用户的访问令牌。它通常调用一个**授权程序包**(authentication package)并利用来自登录子系统或者是网络服务器的信息来执行授权操作。通常，授权程序包只是查阅本地数据库中的账户信息并核查确认口令的正确性。接着，安全子系统再为用户 ID 生成访问令牌，其中包含了适当的优先权、配额限制和组 ID。不管什么时候用户试图访问系统中的一个对象，例如打开一个指向对象的句柄，访问令牌都是先传递给核查优先权与配额的安全引用监视器。Windows XP 域的缺省鉴别程序包是 Kerberos。LSASS 也负责实现安全策略如加强型密码、认证用户并执行数据和密码的加密。

22.5 文件系统

在历史上，MS-DOS 系统使用的是文件分配表(简称 FAT)文件系统。16 位的 FAT 文件系统有很多缺点，包括内部文件碎片，2 GB 的大小限制以及缺少文件的访问保护。32 位 FAT 文件系统已经解决了大小和片断的问题，但是与现代的文件系统相比较，它的性能和功能还是很弱。NTFS 文件系统就好多了。它设计有很多的功能，包括文件恢复、安全、容错、超大文件和文件系统、多种数据流、UNICODE 名称、稀疏文件、加密、日志记录、卷隐藏拷贝和文件压缩。

Windows XP 仍然继续使用 FAT16 读取软盘和其他可移动媒介。尽管 NTFS 有许多优点，FAT32 因其与 Windows 95/98 的所用媒介的互操作性而继续起着重要作用。Windows XP 支持用于 CD 和 DVD 媒介的常用格式的其他文件系统类型。

22.5.1 NTFS 内部布局

NTFS 文件系统的基本实体是卷。卷是由 Windows XP 的逻辑磁盘管理器应用程序所创建的，它是基于逻辑磁盘分区的。一个卷可能占据一个磁盘的一部分，也可能占据着整个的磁盘，或者是横跨几个磁盘。

NTFS 不与磁盘的单个扇区打交道，而是采用簇(cluster)来作为磁盘分配的单元。一个簇的是由一些磁盘扇区组成的，扇区数是 2 的幂。一个 NTFS 文件系统在格式化时就设

置了簇的大小。一个缺省簇的大小，对于不超过 512 MB 的卷为一个扇区，对于不超过 1 GB 的卷为 1 KB，对于不超过 2 GB 的卷为 2 KB，而对于更大的卷为 4 KB。这种簇大小比 16 位 FAT 文件系统的簇要小很多，小尺寸也降低了内部碎片的数量。例如，考虑一个 1.6 GB 的磁盘有 16000 个文件。如果你使用的是一个 FAT16 的文件系统，内部碎片可能会达到 400 MB，因为簇的大小是 32 KB。在 NTFS 系统下，当保存同样数量的文件时只会丢失 17 MB。

NTFS 把**逻辑簇号码**(logical cluster numbers，LCN)用做磁盘地址。按照磁盘的开端到末端的顺序，给每个簇编号。利用这种方案，根据簇的大小乘以 LCN 所得的结果，系统能计算出一个物理磁盘偏移量(按字节)。

NTFS 中的文件并不是 MS-DOS 或者是 UNIX 系统的简单的字节流；相反，它是一个结构化的对象，由多种**属性**组成。一个文件的每个属性是一个独立的字节流，这种字节流可以新建、删除、读取和写出。有一些属性是标准的，适用于所有的文件，包括文件名(或名称，如果文件有别名如 MS-DOS 的短名)、创建时间以及说明访问控制的安全描述符。用户数据保存在*数据属性*中。

绝大多数传统数据文件都有一个无名的数据属性，用于包含文件所有的数据。然而，另外数据流可以用显式名称来创建。例如，保存在 Windows XP 服务器上的 Macintosh 文件就将资源分支作为命名数据流。COM 的 IProp 接口采用命名数据流以存储普通文件属性，包括小图像。一般来说，属性可能按需要增加，并通过语法 file-name:attribute 来加以访问。对于文件查询操作如运行 dir 命令，NTFS 只返回了未命名属性的大小。

NTFS 的每个文件都保存在一个特殊文件数组中的一个或多个记录中，这个文件叫做主文件表(master file table，MFT)。一个记录的大小在文件系统被创建时就确定了；它的范围大小在 1 KB～4 KB 之间。一些小的属性存放在 MFT 记录当中，并被叫做**常驻属性**(resident attribute)。如未命名的巨大的数据，属于超级属性，叫做**非常驻属性**(nonresident attribute)，被保存在一个或多个连续的磁盘盘区中，而指向每个盘区的每个可寻址指针则保存在 MFT 记录中。至于极小的文件，甚至数据属性也完全符合 MFT 记录。如果一个文件具有多种属性，或者它是高度分散的，那么就需要许多的指针来指向所有的存储片，MFT 中的一个记录可能就不够大了。在这种情况下，文件由一个**基本文件记录**(base file record)来描述，基本文件记录也是一条记录，它包含了指向溢出记录(拥有额外的指针和属性)的指针。

NTFS 卷的每个文件都有一个惟一的 ID，称为**文件引用**(file reference)。文件引用是一个 64 位的值，它由一个 48 位的文件号码和一个 16 位的序列号码组成。文件号码就是 MFT 中描述文件的记录号码(也就是，数组插槽)。每次使用一个 MFT 的记录时，其序列号递增。这种顺序号码允许 NTFS 执行内部一致性检测，如在 MFT 项用于新文件之后发现一个删除文件的失效引用。

1. NTFS B+树

与在 MS-DOS 和 UNIX 系统中一样，NTFS 的名称空间是根据目录层构成的。每个目

录使用一个称为 **B+树**的数据结构，用于保存该目录内的所有文件名称的一个索引。B^+ 树能消除重新组织树的耗费，并具有这样的特性：从树根到树叶的每个路径等长。目录的**索引根**(index root)包括 B+树的顶层。对于一个超大目录，这个顶层包含了磁盘的延伸区，而延伸区保存了此树的剩余部分。目录的每项都包含文件的名称和引用，及更新过的时间信息拷贝、从 MFT 的文件常驻属性中获取的文件大小。这些信息的副本被保存在目录中，因此就有能力生成一个目录列表，上面有全部的文件名、文件大小以及从目录本身就可获得更新的时间，所以就没有必要为了每个文件把所有的这些属性从 MFT 的各项中集中起来。

2. NTFS 元数据

NTFS 卷的元数据均保存在文件中。第一个文件是 MFT。第二个文件，用于 MFT 文件遭破坏时的恢复，包括了 MFT 前 16 项的一个副本。下面的一些文件也很特殊。它们是日志文件、卷文件、属性定义表、根目录、位图文件、引导文件以及坏簇文件。**日志文件**记录文件系统更新的所有元数据。**卷文件**(volume file)包含卷名、卷格式化后的 NTFS 版本以及显示已被损坏的卷而需要连续查验的一个位。**属性定义表**(attribute-definition table)显示卷中用的是哪种属性类型，对它们中的每个要执行什么样的操作。

根目录是文件系统层中的顶级目录。**位图文件**(bitmap file)显示卷中的哪几个簇被分配给了文件以及哪几个是空闲的。**引导文件**(boot file)包含了 Windows XP 的启动代码，并且必须放在一个特殊的磁盘位置以便简单 ROM 导入设备的装入程序能很容易查找到。导入文件还包含了 MFT 的物理地址。最后，**坏簇文件**(bad-cluster file)跟踪卷内的任意坏区；NTFS 使用这个记录以进行错误恢复。

22.5.2 恢复

对许多简单文件系统，非正常时间掉电会导致文件系统的数据结构遭到极其严重的破坏，甚至于能把整个卷都搞乱。许多版本的 UNIX 在磁盘上保存了多余的元数据，它们利用 fsck 程序去检查所有文件系统数据结构，得以从崩溃的系统中恢复过来，并把它们强行保存到一个相容的状态。恢复这些数据经常会删除损坏了的文件和释放数据簇(而用户的数据已被写进了这些簇但尚未完全地记录到文件系统的元数据结构中)。这种核查是一个很缓慢的过程，并且会丢失相当数量的数据。

1. NTFS 日志文件

NTFS 为文件系统的健壮性采取了一种不同的方法。对于 NTFS，所有文件系统数据结构的更新都在事务中执行。在改动一个数据结构之前，事务会写入一个包含 redo 和 undo 信息的日志记录；在改变完数据结构之后，事务会在日志上写进一个提交记录，表示该事项已操作成功。

在崩溃后，系统通过处理日志记录把文件系统的数据结构恢复到一致状态。首先重复已提交事务的操作，然后撤销对在系统崩溃之前未成功提交事务的操作。一个检查点记录

会被周期性地(通常是 5 s)写进日志中去。系统为了从崩溃中恢复过来,并不需要检查点之前的日志记录。这些记录可以删除,所以日志文件不会无限增加。在系统启动之后的首次访问一个 NTFS 卷时,NTFS 会自动执行文件系统的恢复。

此方案并不能保证崩溃之后的所有用户文件还能保持正确,它只能确保文件系统的数据结构(元数据文件)没有被损坏,并能反映崩溃之前的已存在的某个一致状态。将事务方案延伸到用户文件是有可能的,微软公司将来可能会这么做。

日志保存在卷初的第三个元数据文件当中。它是在文件系统格式化时,用一个固定的最大尺寸来创建的。它有两个区域:一个是**记录区域**(logging area),它是日志记录的一个循环队列;另一个是**重启区域**(restart area),它包含关联环境信息,如记录区域的位置(恢复期间 NTFS 启动读取操作的地方)。事实上,重启区域有两个相关信息的副本,如果一个副本在崩溃中被损坏,那么恢复还是可能的。

记录功能是由 Windows XP **日志文件服务**(logging file service)所提供的。除了写日志记录和执行恢复操作外,日志服务还跟踪日志文件的空闲空间。如果空闲空间太少,那么日志文件服务会暂停进行的事务,NTFS 会暂停所有新 I/O 操作。在所进行的操作完成之后,NTFS 会调用缓存管理器以写出所有数据,接着重新设置日志文件并执行待进行的事务。

22.5.3 安全

NTFS 卷的安全性是从 Windows XP 的对象模型中派生出来的。每个 NTFS 文件都引用一个安全描述符,它包含文件所有者的访问令牌以及一个访问控制列表,用于规定访问该文件的用户所具有的访问特权。

在正常情况下,NTFS 对遍历目录文件名并不要求许可权限。然而,为了与 POSIX 相兼容,则可以启动这些检查。遍历检查本身更加复杂,这是因为现代文件名的解析并不使用前缀匹配而是使用一个逐个打开目录名来进行的。

22.5.4 卷管理和容错

FtDisk 是 Windows XP 的容错磁盘驱动程序。在安装后,它提供了多种方法把多个磁盘驱动器组成一个逻辑卷,以便于提高性能、计算效率或可靠性。

1. 卷集合

合并多个磁盘的一种方法是把它们在逻辑上组织成一个超大的逻辑卷,如图 22.7 所示。在 Windows XP 中,这个逻辑卷被称为一个**卷集**(volume set),它由 0～32 个物理分区组成。一个卷集包含有一个 NTFS 卷;已保存于文件系统中的数据在没有被扰乱的情况下,可以扩展该卷集。扩展 NTFS 卷的位图元数据只是覆盖新增的空间。NTFS 连续使用同一 LCN 机制,如同用于单一物理磁盘的一样。FtDisk 驱动程序提供了从一个逻辑卷偏移到一个特定磁盘偏移的映射。

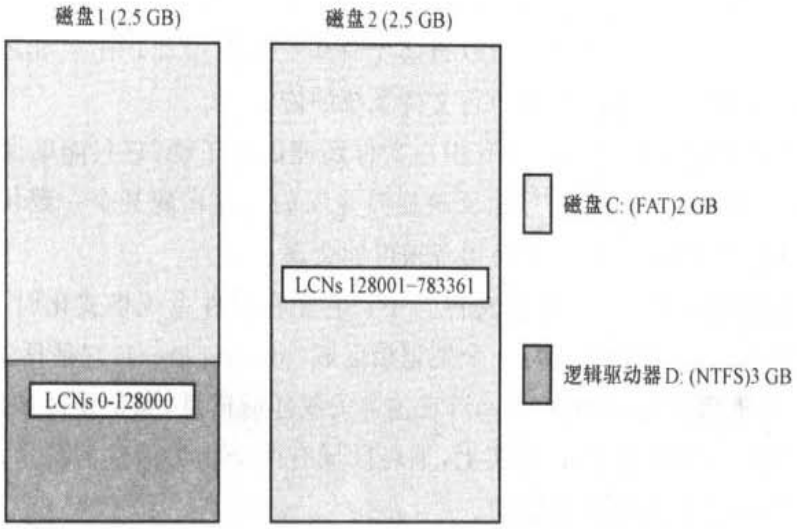


图 22.7 两个驱动器上的卷集

2. 条集合

合并多个物理分区的另一个方法是用循环法交替使用它们的区块以形成所谓的**条集**(stripe set),如图 22.8 所示。这种方案称为 RAID0,或者是**磁盘分条法**(disk stripe),FtDisk 使用的是一个 64 KB 大小的条。逻辑卷的第一个 64 KB 保存在第一个物理分区,第二个 64 KB 的逻辑卷保存在第二个物理分区,依此类推,直到每个分区都分配了 64 KB 的空间。然后,分配绕回到第一个磁盘,分配第二个 64 KB 的区块。条集形成了一个巨大的逻辑卷,但是物理布局可以改善 I/O 的带宽,对于一个巨大的 I/O ,所有的磁盘可以并行转移。

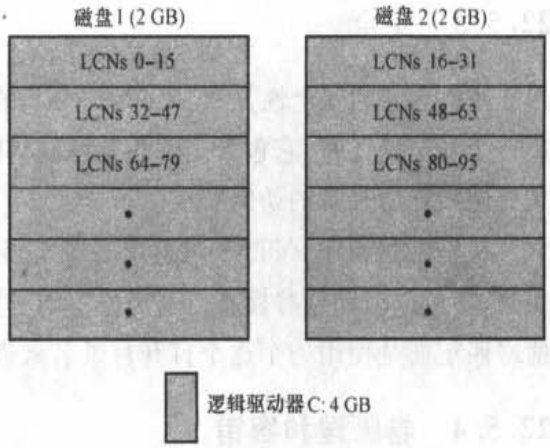


图 22.8 两个驱动器上的条集

3. 带有奇偶的条集合

这个思路的一个变通是**奇偶条集**(stripe set with parity),如图 22.9 所示。这个方案也被称为 RAID 5。如果条集拥有 8 个磁盘,那么 7 个数据条分别位于 7 个独立的磁盘,而奇偶条位于第 8 个磁盘上。奇偶条含有数据条的基于字节的异或(exclusive or)。如果这 8 个条中的任意一个遭到毁损,系统将通过计算异或并根据剩余 7 个奇偶条的数据而重新构建数据。重建数据的能力使得磁盘阵列在磁盘遭遇错误时丢失数据的可能大大减少。

可以注意到,一个数据条的更新同样要重新计算奇偶条。同时向 7 个不同的数据条(写进数据)进行的 7 个写操作同样也需要更新 7 个奇偶条。如果奇偶条都在同一个磁盘,那么

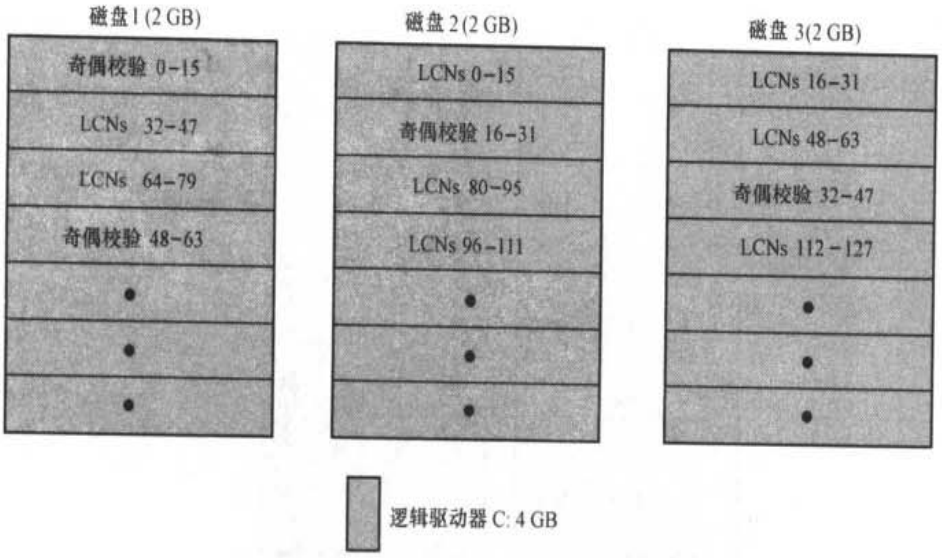


图 22.9 三个驱动器上的带奇偶校验的条集合

该磁盘将执行 7 次的数据磁盘输入/输出装入(操作)。为了避免建立瓶颈,通过给它们赋循环值,把奇偶条分摊在所有的磁盘上。用奇偶校验建立一个条集,至少需要独立于不同磁盘的大小相同的 3 个分区。

4. 磁盘镜像、镜像集合、RAID 和双工集合

一个更强壮方案是**磁盘镜像**(disk mirror),或者叫做 RAID 1 级,如图 22.10 所示。一个**镜像集**(mirror set)由两个分别位于不同磁盘的大小相同的分区组成,这样它们的数据内容是完全相同的。当一个应用程序把数据写进一个镜像集时,FtDisk 就把数据同时写入了两个分区。如果一个分区写入失败,FtDisk 还有一个拷贝安全地存放于镜像中。镜像集同样可以提高性能,因为读请求会在两个镜像之间进行分割,给予每个镜像一半的工作量。为防护磁盘控制器的失效,可以把一个镜像集的两个磁盘放在两个独立的磁盘控制器上,这种安排叫做双工集(duplex set)。

5. 扇区备用与簇重映射

为处理变坏的磁盘扇区,FtDisk 使用了一种硬件方法,称为**扇区备用法**,而 NTFS 使用的是一种簇重新映射法。扇区备用法(sector sparing)是一种由许多磁盘驱动器提供的硬件能力。当一个磁盘驱动器被格式化时,它在逻辑区块号码与磁盘的好扇区之间建立一个映射。它同时也保留了未被映射的扇区作为备用。如果一个扇区失效了,FtDisk 就命令磁盘驱动器替代一个备用扇区。**簇重新映射法**(cluster remapping)是文件系统执行的一种软件方法。如果一个磁盘区块变坏了,NTFS 通过改变 MFT 中任一受影响的指针,将用一个不同的未分配的区块来置换它。NTFS 还会做上一条标记,以注明损坏区块不会再分配给任何文件。

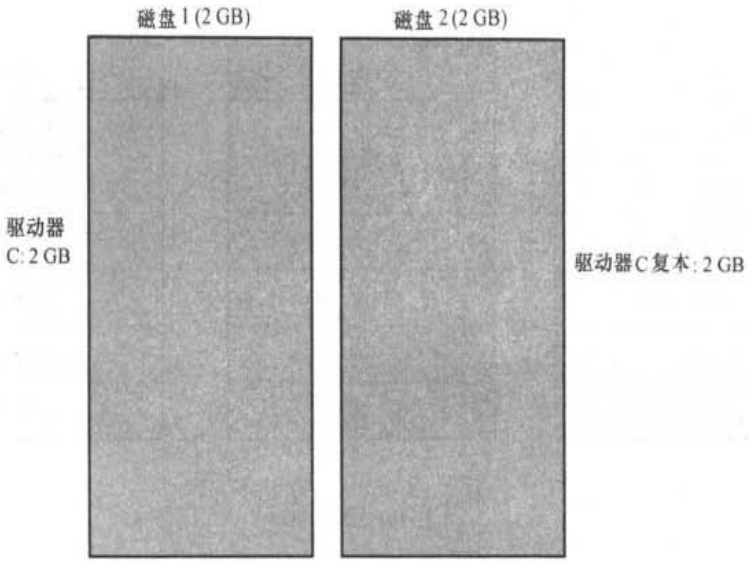


图 22.10 两个驱动器上的镜像集

一个磁盘区块损坏时,通常的结果是数据的丢失。但是扇区备用法与簇重新映射法也可能会和诸如条集这样的容错卷连接在一起,以处理磁盘区块的失效。如果读失败,系统会通过读取镜像,或者是通过计算异或,或是用奇偶校验法校验条集中的奇偶来重新构建丢失的数据。重新构建的数据被存放到一个新的位置,而该位置是用扇区备用法或者是簇重新映射法取得的。

22.5.5 压缩与加密

NTFS 可以对目录中的个别文件或者是所有数据文件进行数据压缩。在压缩一个文件之前,NTFS 首先把文件的数据分成几个**压缩单元**,压缩单元是 16 个相邻簇的区块。当写进每个压缩单元时,就应用一种数据压缩算法。如果结果符合的要比 16 簇少,就要保存这个压缩版本。要进行读时,NTFS 能确定已被压缩的数据:如果是压缩过的,被保存的压缩单元的长度一定小于 16 簇。为了提高性能,当读取连续的压缩单元时,NTFS 会在应用程序发出请求之前提前提取并解压缩。

对于稀疏文件或者是包含多数为零的文件,NTFS 使用另外一种方法来节省空间。因为含有所有为零的文件簇未被写过,所以实际上并没有被分配和保存在磁盘上。而是以间隙的形式保存在文件 MFT 项的各个虚拟簇的序列号码中。读一个文件时,如果它找到了虚拟簇号码间的间隙,NTFS 只是用零填充调用程序缓冲区的分区。UNIX 用的也是这种方法。

22.5.6 安装点

安装点是 NTFS 目录特有的一种符号链接。它们提供了更加灵活的机制,以便管理员不仅仅提供符号链接(如驱动器字母),而且还能更好地组织磁盘卷。安装点以符号链接的

形式实现，与其相关联的数据包含其真正的文件名。最终，安装点可完全取代驱动器字母，虽然这一转换过程会比较长，因为许多应用程序使用驱动器字母方案。

22.5.7 改变日志

NTFS使用一个日志描述对文件系统所做的所有修改。用户模式服务可接受这种日志改变的通知，并能确定哪些文件发生了改变。内容索引服务采用这一功能以确定哪些文件需要再次索引。文件复制技术通过这一功能以确定哪些文件需要通过网络加以复制。

22.5.8 卷影子拷贝

Windows XP实现了将一个卷变成一致状态并创建一个影子拷贝(以用于恢复到一致状态)的功能。对一个卷做一个影子拷贝(shadow copy)采用了一种形式的写时复制，在影子拷贝创建之后所修改的数据中隐藏着原来数据的内容。为了实现卷的一致状态，要求应用程序加以合作，因为系统并不知道应用程序所使用的数据是否处于一致，以便安全地重新执行应用程序。

Windows XP服务器版本使用影子拷贝以有效地维护文件服务器的文件的旧版本。这允许用户看到文件服务器所保存的以前版本的文档。用户使用这个特性，可在不使用备用磁带的情况下以恢复偶然删除的文件或只是检查文件的以前版本。

22.6 网　　络

Windows XP同时支持对等网络与客户－服务器网络，还具有网络管理的程序。Windows XP中的网络组件有数据传输、进程间通信、网络文件共享以及向远程打印机发送打印任务。

22.6.1 网络接口

为描述Windows XP网络，将引用两个内部网络接口，它们是**网络设备接口规范**(Network Device Interface Specification，NDIS)和**传送驱动器接口**(Transport Driver Interface，TDI)。NDIS接口由微软公司和3Com公司于1989年合作开发而成，主要是为了把网络适配器从传输协议中分立开来，这样两者可以在相互不影响的情况下得到改变。NDIS驻留在OSI模型的"数据连接控制"与"媒介访问控制"层之间的接口；它使得许多协议能够在许多不同的网络适配器中间进行操作。就OSI模型来说，TDI是位于传输层(第四层)和会话层(第五层)之间的接口，此接口允许任意会话层的组件能够使用任何可利用的传输机制。(与UNIX中<数据>流有很相似的原因。)TDI同时支持基于连接与非连接的传输，具有发送任何类型数据的功能。

22.6.2 协议

Windows XP 通过驱动程序来实现传输协议。这些驱动程序可动态地装入或卸载，虽然有时需要在改变之后重启系统。Windows XP 有多个网络协议。

1. 服务器消息块

服务器消息块(Server Message Block，SMB)首先用于 MS-DOS 3.1。系统可能使用这一协议通过网络发送 I/O 请求。协议 SMB 有 4 种消息类型。消息 Session control 命令用于开始和结束服务器端共享资源的重定向连接。转向器采用消息 File 访问服务器端的文件。系统使用消息 Printer 向远程打印机队列发送数据并接收状态信息，消息 Message 用于与另一工作站进行通信。SMP 协议是作为 **CIFS**(Common Internet File System)发布的，现在被多个操作系统所支持。

2. 网络基本 I/O 系统

网络基本 I/O 系统(Network Basic Input/Output System，NetBIOS)是用于网络的硬件抽象接口，类似于用于运行 MS-DOS 的 PC 硬件抽象接口 BIOS。NetBIOS，设计于 20 世纪 80 年代的早期，现已成为标准网络编程接口。NetBIOS 用于建立在网络之上的逻辑名，建立网络逻辑名之间的逻辑连接或会话，支持通过 NetBIOS 或 SMB 请求的可靠会话数据传输。

3. 网络 BIOS 扩展用户接口

网络 BIOS 扩展接口(NetBIOS extended user interface，NetBEUI)是由 IBM 公司于 1985 年引入的，用于支持不超过 254 台机器的简单而高效网络协议。这是 Windows 95 对等网络和 Windows for Workgroups 的缺省协议。Windows XP 在与这些网络进行资源共享时，使用 NetBEUI。NetBEUI 限制很多，它使用计算机的真实名称作为地址，而且不支持路由。

4. 传输控制协议/因特网协议

传输控制协议/因特网协议(Transmission Control Protocol/Internet Protocol，TCP/IP)用于因特网，现已成为事实上的标准网络体系结构。Windows XP 采用 TCP/IP 与各种不同的操作系统和硬件平台相连。Windows XP TCP/IP 包括简单网络管理协议(simple network management protocol，SNMP)、动态主机配置协议(dynamic host configuration protocol，DHCP)、Windows Internet 名称服务(Windows Internet name service，WINS)和 NetBIOS 支持。

5. 点到点隧道协议

点到点隧道协议(point-to-point tunneling protocol，PPTP)由 Windows XP 提供，用于在运行 Windows XP 服务器的远程访问服务器模块和在因特网相连的客户机之间进行通

信。远程访问服务器可以对连接上所发送的数据进行加密，它们支持多协议的通过因特网的**虚拟私有网络**(virtual private network，VPN)。

6. Novell Netware 协议

Novell Netware 协议(SPX 传输层上的 IPX 数据报服务)广泛地用于 PC LAN。Windows XP 的 NWLink 协议将 NetBIOS 与 Netware 网络相连。与转向器(如微软分司用于 Netware 的客户服务或 Novell 的用于 Windows 的 Netware 客户)一起，这个协议允许 Windows XP 客户机与 NetWare 服务器相连。

7. 基于 Web 的创建与版本协议

基于 Web 的创建与版本协议(Web Distributed Authoring and Versioning，WebDAV)是基于 HTTP 的协议，以用于通过网络协作创作。Windows XP 将 WebDAV 转向器建立到文件系统。通过在文件系统中直接建立 WebDAV 支持，它可与其他特点如加密一起工作。这样个人文件在公共地方也能安全存储。

8. AppleTalk 协议

AppleTalk 协议是由 Apple 所设计的，允许 Macintosh 计算机共享文件的低成本连接。如果位于网络的 Windows XP 服务器运行用于 Macintosh 包的 Windows 服务，那么 Windows XP系统可通过 AppleTalk 与 Macintosh 计算机共享文件和打印机。

22.6.3 分布式处理机制

虽然 Windows XP 不是分布式操作系统，但是它确实支持分布式应用。Windows XP 的分布式处理所支持的机制包括 NetBIOS、命名管道、mailslot、Windows Socket、RPC 和网络动态数据交换(NetDDE)。

1. Net BIOS

对于 Windows XP，NetBIOS 应用程序可通过网络采用 NetBEUI、NWLink 或 TCP/IP 进行通信。

2. 命名管道

命名管道(name pipe)是面向连接的消息机制。命名管道最初设计成网络 NetBIOS 连接的高层接口。一个进程也可以使用命名管道与同一台机器的另一个进程进行通信。由于命名管道通过文件系统接口加以访问，所以用于文件对象的安全机制也适用于命名管道。

命名管道的名称有一个称为**统一命名习惯**(uniform naming convention，UNC)的格式。UNC 名称看起来像普通远程文件名称。UNC 名称的格式为*server_name**share_name*\x\y\z，其中 server_name 标识网络服务器；share_name 标识可为网络用户所用的网络资源如目录、文件、命名管道和打印机等；\x\y\z 部分表示普通文件路径名。

3. mailsot

mailslot 是无连接的消息传递机制。当通过网络使用时，mailslot 是不可靠的。即发送给 mailslot 的消息可能会丢失，以致于要接收它的用户不能接收到。mailslot 用于广播应用程序如查找网络组件；它们也为 Windows 计算机浏览服务所使用。

4. Winsock

Winsock 是 Windows XP 套接字 API。Winsock 是会话层接口，与 UNIX 套接字大部分兼容，但是增加了一些 Windows XP 的扩展。它提供了对许多不同传输协议（具有不同寻址方式）的一个标准接口，这样 Winsock 应用程序可运行于任何与 Winsock 相兼容的协议堆栈。

5. 远程过程调用

远程过程调用（remote procedure call，RPC）是一个客户机—服务器通信机制，允许一台机器的一个应用程序调用另一台机器上的过程。客户进程调用本地过程——即**存根子程序**（stub routine），将参数打包成一个消息，并通过网络发送给一个特定服务器进程。接着客户端存根子程序就阻塞。同时，服务器解开消息，调用这个过程，并将结果打包成消息，再传递给客户存根。客户存根不再阻塞，接收消息，揭开 RPC 的结果，并返回给客户进程。这种参数打包有时称为**编组**（marshalling）。Windows XP RPC 机制遵守广泛使用的用于 RPC 消息的分布式计算环境的标准。RPC 标准非常详细。它通过规定 RPC 消息的标准数据格式，而隐藏了计算机体系结构的差异如二进制大小、计算机字的字节和位的顺序。

Windows XP 可通过 NetBIOS 或 TCP/IP 网络的 Winsock 或 LAN Manager 网络的命名管道等，来发送 RPC 消息。以前所讨论的 LPC 与 RPC 相似，不过在同一计算机的两进程之间的消息传递是通过 LPC 消息进行的。

6. Microsoft 接口定义语言

可以通过手工编写代码并按标准形式编排和传递参数，解开参数并执行远程过程，编排和传递返回结果，解开结果并返回给调用程序，但是这非常麻烦且容易出错。然而，幸运的是，根据参数和返回结果的简单描述，可以自动地生成这部分代码。

Windows XP 提供了**微软接口定义语言**（Microsoft Interface Definition Language）描述远程过程的名称、参数和结果。这种语言的编译器可生成头文件以描述远程过程的存根和参数与返回值消息的数据类型。它也为客户端存根子程序和服务器端的解开程序和分派程序，以生成源程序。当链接应用程序时，会包括存根过程。当应用程序执行 RPC 存根，所生成的代码处理其他部分。

7. 组件对象模型

组件对象模型（component object model，COM）是 Windows 所开发的进程间通信的机制。COM 对象提供了明确定义的接口以操作对象的数据。例如，COM 是 Microsoft 的**对象链接和嵌入**（object linking and embedding，OLE）的基础，它可用于在 Word 文档中嵌入电

子表格。Windows XP 有一个称为 **DCOM** 的分布式扩展，可通过 RPC 运行于网络之上，以提供一种透明方式而开发分布式应用程序。

22.6.4 重定向器与服务器

对于 Windows XP，应用程序可使用 Windows XP I/O API 访问远程计算机的文件如同位于本地一样，只要远程计算机运行 CIFS 服务器（如 Windows XP 或日期 Windows 系统所提供的）。**重定向器**是客户端对象，可转寄对远程文件的 I/O 请求，再为服务器所处理。基于性能和安全原因，重定向器与服务器运行于内核模式。

更为具体地，远程文件的访问按如下方式进行。

- 应用程序调用 I/O 管理器以请求打开一个文件，其文件名采用标准 UNC 格式。
- I/O 管理器建立 I/O 请求包，如 22.3.3.5 一节所述。
- I/O 管理器知道这是个远程文件的访问，因此调用一个称为 **MUP**（multiple universal naming convention）的驱动器。
- MUP 向所有注册重定向器异步发送 I/O 请求包。
- 可满足请求的重定向器响应 MUP。为了避免将来再次用同样问题来询问所有重向器，MUP 采用缓存记录哪个重向器可处理这个文件。
- 重定向器向远程系统发送网络请求。
- 远程系统网络驱动程序接收请求，并传递给服务器驱动器。
- 服务器驱动程序将这个请求转交给合适的本地文件系统驱动程序。
- 调用适当设备驱动程序以访问数据。
- 结果返回给服务驱动程序，再将数据发回给请求重定向器。接着重定向器通过 I/O 管理器将数据返回给调用应用程序。

对于使用 Win32 网络 API 而不是 UNC 服务的应用程序，也有类似过程。不过，所使用的模块是多提供者的路由而不是 MUP。

为了可移植性，重定向器与服务器采用基于网络传输的 TDI API。请求本身用更为高层的协议表示，这缺省为 22.6.2 小节所述的 SMP 协议。重定向器列表是由系统注册数据库所维护的。

1. 分布式文件系统

UNC 名称并不总是便于使用，因为多个文件服务器可用于提供同样内容的文件而 UNC 名称显式地包括了服务器名称。Windows XP 支持**分布式文件系统**（Distributed File System，DFS）协议，从而允许网络管理员采用单一分布名称空间而用多个服务器来提供文件服务。

2. 目录重定向与客户端缓存

为了使得经常切换计算机的企业用户更好地使用计算机，Windows XP 允许管理员为

用户建立**漫游 profile**，以在服务器上保存用户的偏好和其他设置。这样**目录重定向**可自动地在服务器上存储用户文档和其他文件。但是当一台计算机不再连到网络上(如位于飞机上的便携型电脑)，就出现问题了。为了让用户离线访问其重定向文件，Windows XP 采用**客户端缓存**(client-side caching，CSC)。当在线时，CSC 在客户机上保存服务器文件的拷贝以提高性能。当文件改变时，再送回到文件。如果计算机不再连接时，那么使用可用文件，只不过文件更新延迟到计算机再次与网络相连时。

22.6.5 域

许多网络环境都由用户自然组合而成，如学校计算机实验室的学生，或某企业某部分的雇员。通常，需要让组内的所有成员能够访问组内各台计算机的共享资源。为了管理这种组的全局访问权限，Windows XP 采用了域概念。以前，这些域与 DNS(将因特网名称转换成 IP 地址)没有任何关系。然而，现在它们却紧密联系在一起。

具体地说，Windows XP 的一个域是共享共同安全策略和用户数据库的一组 Windows XP 工作站和服务器。由于 Windows XP 现在将 Kerberos 协议用于信任和验证，所以 Windows XP 的域与 Kerberos 领域完全一样。NT 的以前版本采用了主和备份域控制器的概念；现在所有域内的服务器都是域控制器。另外，Windows XP 采用了基于 DNS 的分层方法，允许在层次结构中上下传递信任。这种方法降低了用于 n 个域的信任数量，从 $n\times(n-1)$ 到 $O(n)$。域内工作站信任域控制器会提供关于每个用户访问权限的正确信息(通过用户访问标记)。不管域控制器如何说，所有用户保留限制对其工作站访问的能力。

1. 域树与森林

因为一个企业可以有许多部门，而一个学校可有多个班级，所以通常需要管理一个组织的多个域。一个**域树**(domain tree)为一个连续 DNS 命名层次。例如，*bell-labs.com* 可以为树根，而 *research.bell-labs.com*(表示域 research)和 pez.bell-labs.com(表示域 pez)为孩子。一个森林为一组非连续的集合。一个例子是树 bell-labs.com 和/或 lucent.com。然而，一个**森林**可只由一个树组成。

2. 信任关系

信任关系可以在域之间按三种方式进行建立：单向的、传递的和交叉的。NT 5.0 以前的版本只允许单向信任。**单向信任**(one-way trust)正如其名称所指的：域 A 被告之它可信任域 B。然而，域 B 并不信任 A 除非配置了这种信任关系。对于**传递信任**(transitive trust)则是，如果 A 信任 B 且 B 信任 C，那么 A、B、C 互相信任，因为传递信任缺省为双向的。传递信任缺省用于树内的新域，也可配置成用于森林内的域。第三种类型，**交叉信任**(cross-link trust)，可用于减少验证流量。假如域 A 和 B 为叶节点，而域 A 内的用户使用域 B 的资源。如果使用标准传递信任，那么验证请求必须传递到这两个叶节点的共同祖先；但是如果 A 和 B 已建立了交叉连接，那么验证可直接发送给另一节点。

22.6.6 活动目录

活动目录是 Windows XP 的**轻量级目录访问协议**(lightweight directory access protocol,LDAP)服务的实现。活动目录存储关于域的拓扑信息,为基于域的用户和组保留账号及其口令,提供有关**组策略**和 **intellimirror** 的基于域的存储。

管理员使用组策略为桌面偏好和软件建立标准。对许多企业信息技术组,统一显著地降低了计算代价。Intellimirror 与组策略一起使用,以指定什么软件为什么类型用户所使用,甚至可根据需要从企业服务器自动安装软件。

22.6.7 TCP/IP 网络的名称解析

对于 IP 网络,**名称解析**可将计算机名称转换成 IP 地址,如将 *www.bell-labs.com* 转换成 135.104.1.14。Windows XP 为名称解析提供了多种方法,包括 WINS(Windows Internet Name Service)、广播名称解析、DNS、文件 host 和文件 LMHOSTS。绝大多数这些方法被许多操作系统所使用,但这里只讨论 WINS。

对于 WINS,两台或多台 WINS 服务器维护名称-IP 地址捆绑的动态数据库,而客户软件询问服务器。至少使用两台服务器,这样即使一台服务出差错也不会中断 WINS 服务,另外也可将名称解析负荷分布在多台机器上。

1. 动态主机配置协议

WINS 使用了动态主机配置协议(dynamic host-configuration protocol,DHCP)。DHCP 可自动地更新地址配置 WINS 数据库,而无需用户或管理员的干预。当 DHCP 客户机开始时,会广播消息 discover。每个收到消息的 DHCP 服务器会用消息 offer 加以应答,它包括了客户所需要的 IP 地址和配置信息。客户选择一个配置,并向所选择的 DHCP 服务器发送一个请求消息。DHCP 服务器用之前所给的 IP 地址和配置信息及地址**租赁**,来加以响应。这个租赁允许客户有权在给定时间内使用这个 IP 地址。当租赁时间用完一半时,客户会试图续租这个地址。如果不能续租,那么客户必须得到一个新地址。

22.7 程序接口

Win32 API 是 Windows XP 功能的基本接口。本节描述 Win32 API 的五个主要方面:内核对象的访问、进程间对象的共享、进程管理、进程间通信和内存管理。

22.7.1 内核对象访问

Windows XP 内核提供了应用程序能使用的许多服务。应用程序通过处理内核对象来取得这些服务。进程访问名为 XXX 的一个内核对象,可通过调用 CreatXXX 函数来打开一个

指向 XXX 的句柄。如果 Creat 函数调用失败，将返回一个"0"值，或者是返回一个名为 INVALID_HANDLE_VALUE 的特殊常量，这都取决于打开了哪个目标。进程通过调用 CloseHandle 函数可以关闭任何一个句柄。如果运行对象的进程计数掉到 0 时系统将删除这个对象。

22.7.2 进程间的对象共享

Windows XP 提供三种方法，用以共享进程间对象。第一种办法是，子进程继承了一个指向对象的句柄。当父进程调用 CreatXXX 函数时，父进程将 SECURITIES_ATTRIBUTES 结构的 bInheritHandle 字段设定为 TRUE。该字段能新建一个可遗传的句柄。然后，新建的子进程把 TRUE 值传递到 CreatProcess 函数的 bInheritHandle 变量。图 22.11 展示了一个代码例子，它用于创建一个信号量句柄并为子进程所继承。

```
...
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa,1,1,NULL);
char command_line[132];
ostrstream ostring(command_line,sizeof(command_line));
ostring<< a_semaphore <<ends;
CreateProcess("another_process.exe",command_line,
                NULL,NULL,TRUE,...);
...
```

图 22.11 通过继承句柄使得了进程能共享一个对象的代码

假定子进程知道共享的句柄，那么父进程与子进程通过共享对象可以成功地进行进程间的通信。在图 22.11 中，子进程从第一命令行参数中取得句柄的值，然后与父进程共享信号量。

共享对象的第二种方法是，在对象创建时，进程给该对象命名，以便于第二个进程打开已命名的对象。这种方法有两个缺点。第一个缺点是，Windows XP 不提供任何方法来检查选定的名称对象是否已经存在。第二个缺点是，对象名空间是全局的，不需要确定对象类型。例如，需要两个截然不同对象时，两个程序可以新建一个名为 pipe 的对象。

命名对象的优点是不相关进程也可以很容易地进行共享。第一个进程调用 CreatXXX 函数并在 lpszName 参数中提供一个名字。如图 22.12 所示，第二个进程通过用同样的名字以调用 OpenXXX(或者是 CreatXXX)，可取得一个共享该目标的句柄。

共享对象的第三个方法是通过调用 DuplicateHandle 函数。这种方法要求一些其他的进程间通信方法来传递复制句柄。在进程中给定一个指向某一进程的句柄及句柄的值，另一个进程就能取得同一对象的一个句柄，这样就可以共享了，如图 22.13 中例子所示。

```
//process A
...
Handle a_semaphore = CreateSemaphore(NULL,1,1,"MySEM1");
...
//process B
...
Handle b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
                     FALSE,"MySEM1");
...
```

图 22.12 通过名称查找共享一个对象的代码

```
...
//process A wants to give process B access to a semaphore
//process A
Handle a_semaphore = CreateSemaphore(NULL,1,1,NULL);
//send the value of the semaphore to process B
//using a message or shared memory
...
//process B
Handle process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
                   process_id_of_A);
Handle b_semaphore;
DuplicateHandle(process_a,a_semaphore,
  GetCurrentProcess(),&b_semaphore,
  0,FALSE,DUPLICATE_SAME_ACCESS);
//use b_semaphore to access the semaphore
...
```

图 22.13 通过传递一个句柄来共享一个对象的代码

22.7.3 进程管理

在 Windows XP 系统中，**进程**是应用程序的执行实例，而**线程**是一个可由操作系统调度的代码单元。因此，一个进程能包含一个或者若干个线程。当某个其他进程调用 CreatProcess 例程时，一个新进程就被启动了。该例程装入进程所使用的任一动态连接库，并新建一个**主线程**(primary thread)。CreatThread 函数可创建附加线程。每个线程都是用它自己的栈建立的，除非在调用 CreatThread 函数时加以特别规定，否则栈的默认值是 1 MB。因为有些 C 运行时间函数保留了静态变量，如 errno 函数。一个多线程的应用程序需要以非同步访问加以保护。包装函数 beginthreadex 提供了适当的同步。

1. 实例句柄

装入进程地址空间的每个动态链接库或者是可执行文件通过一个实例句柄加以标识。**实例句柄**(instance handle)的值实际上是装入文件所处的虚拟地址。应用程序通过把模块的名字传递到 GetModuleHandle，便能够取得指向某一模块的句柄。如果把 NULL 传递过去作为名字的话，就会返回进程的所在地址。最低的 64 MB 的地址空间是不会使用的，因此一个企图使用 NULL 指针的错误程序会得到一个访问违例(的警告)。

Win32 环境的优先权基于 Windows XP 的调度模型，但并不是所有的优先值都会被选用。Win32 使用了 4 个优先级：IDLE_PRIORITY_CLASS(优先级为 4)，NORMAL_PRIORITY_CLASS(优先级为 8)，HIGH_PRIORITY_CLASS(优先级为 13) 以及 REALTIME_PRIORITY_CLASS(优先级为 24)。进程通常属于 NORMAL_PRIORITY_CLASS，除非其父进程是属于 IDLE_PRIORITY_CLASS，或者是调用 CreatProcess 函数时已经指定了其他的优先权级。一个进程的优先权级可以用 SetPriorityClass 函数来改变，或者由传送到 START 命令的变量来改变。例如，命

令 START /REALTIME cbserver.exe 可以按 REALTIME_PRIORITY_CLASS 运行 cbserver 程序。请注意只有拥有增长调度优先级特权的用户才能把进程移到 REALTIME_PRIORITY_CLASS 中。管理员和重要用户缺省地拥有这样的权利。

2. 调度规则

当用户运行一个交互式程序时，系统需要为该进程提供特别好的性能。因为这个原因，Windows XP 有一套适用于 NORMAL_PRIORITY_CLASS 进程的特殊调度规则。Windows XP 区分两类进程：前台进程(即在屏幕上当前正在选用的)与后台进程(当前没有选用的)。当一个进程移到了前台进程中，Windows XP 就成倍地增加调度量——经常是 3 倍。(这种倍数通过系统控制面板的性能选项可以改变。)在分时抢占发生之前，这种增长使得前台进程运行的时间延长 3 倍(在分时抢占之前)。

3. 线程优先级别

线程按其所属类型确定的最初优先级而开始。线程优先权可以通过 SetThreadPriority 函数改变。该函数带有一个参数，以表示一个优先权与其类型的基本优先权的相互关系，

- THREAD_PRIORITY_LOWEST：base -2
- THREAD_PRIORITY_BELOW_NORMAL：base - 1
- THREAD_PRIORITY_NORMAL：base +0
- THREAD_PRIORITY_ABOVE_NOMAL：base +1
- THREAD_PRIORITY_HIGHEST：base +2

另外的两个指定也可用于调整优先权。可以回想一下 22.3.2.1 一节所提到的内核的两个优先级：16－31 是实时级，0－15 是变量优先级。THREAD_PRIORITY_IDLE 设定实时线程的优先级为 16，而可变优先级线程的优先级为 1。THREAD_PRIORITY_TIME_CRITICAL 设定实时线程的优先级为 31，而可变优先级线程的优先级为 15。

正如在 22.3.2.1 一节所讨论的一样，内核动态根据该线程是 I/O 限制还是 CPU 限制，调整一个线程的优先权。Win32 API 提供了一种使这种调整失效的方法：SetProcessPriorityBoost 和 SetThreadPriorityBoost 函数。

4. 线程同步

线程也可在**悬挂状态**(suspended state)中加以创建：在其他进程通过调用 ResumeThread 函数之前，该线程不能执行。而 SuspendThread 函数则相反。这些函数设定一个计数器，如果一个线程被暂停过两次，那么在它可以运行之前必须恢复(resume)两次。为了对共享对象的线程并发访问进行同步，内核提供了同步对象，如信号量和互斥。

另外，线程同步可以通过 WaitForSingleObject 或者是 WaitForMultipleObjects 函数来实现。Win32 API 的另一个同步的方法是临界区。临界区是一个同步代码区域，在每个时间片只能有一个线程执行。线程通过调用 InitializeCriticalSection 函数创建临界区。

应用程序在进入临界区之前必须调用 EnterCriticalSection 函数，并且在退出临界区之时调用 LeaveCriticalSection 函数。这两个函数能确保在多个线程试图并行进入临界区的情况下，在任一时候只允许处理一个线程，而其他线程等候在 EnterCriticalSection 例程。临界区机制比内核同步对象处理起来要快一些，这是因为它直到首次碰到临界区时才分配内核对象。

5. fiber

fiber 是用户态代码，它根据用户定义的调度算法进行调度。一个进程里面可能拥有多个 fiber，就像它拥有多个线程一样。线程与 fiber 之间主要的区别在于线程可以并发执行，但是在同一时间只允许执行一个 fiber，甚至是在多处理机的硬件上(也是如此)。这种机制被包括在 Windows XP 里是有助于移植那些旧版本的 UNIX 应用程序，这些程序当时是专为 fiber 执行程序模型所编写的。

系统通过调用 ConvertThreadFiber 或 CreatFiber 来创建 fiber。这两个函数的主要区别是 CreatFiber 函数在创建了 fiber 之后并没有开始执行。如要开始执行的话，应用程序必须调用 SwitchToFiber 函数。应用程序通过调用 DeleteFiber 函数可终止一个 fiber。

对于执行少量工作的应用程序和服务，重复创建或删除线程可能代价昂贵。线程池为用户模式程序提供了三种服务：可以提交工作工作请求的队列(通过 QueueUserWorkItem API)、可用于为可等待句柄捆绑回调的 API(RegisterWaitForSingleObject)、为超时捆绑回调的 API(CreateTimerQueue 和 CreateTimerQueueTimer)。

线程的目的是提高性能。线程相对比较昂贵，不管有多少线程，一个处理器在某一时刻只能执行一个线程。线程池试图通过延迟工作请求(为多个请求重新使用每个线程)并提供足够线程来有效利用机器的 CPU，并降低未完成线程。

等待和定时回调 API 通过使用更少线程(而不是为处理每个可等待句柄或超时都使用一个线程)，以允许线程池进一步降低进程内的线程数量。

22.7.4 进程间通信

Win32 应用程序进行进程间通信的一种方法是通过共享内核对象。另一种方法是通过传递消息，这是一种特别受 Windows GUI 应用程序欢迎的方法。一个线程要向另一个线程或者是一个窗口发送一条消息，可以通过调用 PostMessage、PostThreadMessage、SendMessage、SendThreadMessage 或 SendMessageCallback 来完成。投递(posting)一条信息与发送(sending)一条信息的区别在于投递例程是异步的：它们会立即返回，且调用线程事实上并不知道什么时候真正发送信息。发送例程是同步的，在发送和处理完信息之前它们会阻塞调用程序。

另外，在发送一条信息时，线程也可以发送附带数据的信息。因为进程拥有各自独立的地址空间，所以数据必须复制。系统通过调用 SendMessage 函数来复制数据。通过调用此

函数来发送一条 WM_COPYDATA 类型的信息，并附带一个 COPYDATASTRUCT 数据结构。该数据结构含有要传输数据的长度与地址。信息一旦发送，Windows XP 就把数据复制到一个新的内存区块，并给予新区块以虚拟地址以方便接收进程。

1. 单个输入队列

每个 Win32 线程都拥有它自己的输入队列，这与 16 位的 Windows 环境不同，可以从此处接收到信息。(所有输入都要通过消息来接收。)这种结构比起 16 位的 Windows 的共享输入队列更加可靠，因为有了独立的队列，一个出错应用程序不会阻止其他应用程序的输入。如果一个 Win32 的应用程序不能调用 GetMessage 函数来处理位于其输入队列上的事件，那么整个队列将被填满，在 5 s 之后，系统将给该程序标上"没有响应"的符号。

22.7.5 内存管理

Win32 API 为应用程序使用内存提供了多种方法：虚拟内存，内存映象文件，堆，及线程本地存储。

1. 虚拟内存

应用程序调用 VirtualAlloc 函数保留或者是提交虚拟内存，调用 VirtualFree 函数回收或者释放内存。这些函数允许应用程序指定在内存的什么地方分配虚拟地址。它们按页面大小为单位进行操作，而且分配区域的开始地址必须大于 0x10000。请参见图 22.14 函数例子。

```
...
//allocate 16 MB at the top of our address space
void * buf = VirtualAlloc(0,0x1000000, MEM_RESERVE|MEM_TOP_DOWN,
            PAGE_READWRITE);
//commit the upper 8 MB of the allocated space
VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT, PAGE_READWRITE);
//do some stuff with it
//decommit
VirtualFree(buf + 0x800000,0x800000, MEM_DECOMMIT);
//release all the allocated address space
VirtualFree(buf,0, MEM_RELEASE);
...
```

图 22.14 分配虚拟内存的代码段

进程可以通过调用 VirtualLock 函数把一些已提交的页面锁定在物理内存。单个进程可以锁定的页面数量最大可达 30，除非进程首先调用了 SetProcessWorkingSetSize 函数增加了最大工作集的大小。

2. 内存映射文件

应用程序使用内存的另一种方法是通过把一个文件映射到其地址空间。内存映射对于

两个进程共享内存来说，也不失为一种简便的方法：两个进程都把同一文件映射到它们的虚拟内存中。内存映象是一个多步骤的过程，正如图 22.15 所示的那样。

```
...
//open the file or create it if it does not exit
HANDLE hfile = CreateFile("somefile",GENERIC_READ|GENERIC_WRITE,
   FILE_SHARE_READ|FILE_SHARE_WRITE,NULL,OPEN_ALWAYS,
   FILE_ATTRIBUTE_NORMAL,
   NULL);
//create the file mapping 8MB in size
HANDLE hmap = CreateFileMapping(hfile,PAGE_READWRITE,
   SEC_COMMIT,0,0x800000,"SHM_1");
//get a view to the space mapped
void *buf = MapViewOfFile(hmap,FILE_MAP_ALL_ACCESS,0,0,0×800000);
//do some stuff with it
//unmap the file
UnmapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);...
```

图 22.15 文件的内存映射代码段

如果一个进程想要映射一些地址空间，且只是与另一个进程共享某一内存区域，那么就不需要什么文件。进程可以用一个 0xffffffff 的文件句柄和一个特定的大小，来调用 CreatFileMapping 函数。所生成的文件映射对象可以通过以下方式来共享：继承，名字查找，复制。

3. 堆

应用程序使用内存的第三种方法是堆。Win32 环境的堆是保留地址空间的一个区域。当一个 Win32 的进程在初始化时，就创建了一个带有 1 MB 的**缺省堆**(default heap)。因为许多 Win32 函数应用了缺省堆，所以堆的访问需要同步，以保护堆空间分配数据结构不会由于多个线程的并发更新而损坏。

Win32 提供了几个**堆管理**函数，这样进程就可以分配和管理一个私用堆。这些函数是 HeapCreate，HeapAlloc，HeapRealloc，HeapSize，HeapFree 以及 HeapDestroy。Win32 API 还提供了 HeapLock 和 HeapUnlock 函数使线程对堆可以互斥访问。与 VirtualLock 函数不一样的是，这些函数只能执行同步操作；它们不能把页面锁定到物理内存当中。

4. 线程局部存储

应用程序使用内存的第四种方法是线程本地存储机制。那些依赖于全局的或者是静态的数据的函数在一个多线程环境下通常不能正常工作。例如，C 运行时间函数 strtok 在分析一个字符串的时，采用一个静态变量来跟踪它的当前位置。为了使两个当前并发线程正确执行 strtok 函数，它们需要有各自的当前位置变量。线程本地存储机制允许根据每个线

程分配全局存储。它同时提供了创建线程本地存储的动态与静态的两种方法。对于动态方法请参见图 22.16。

```
//reserve a slot for a variable
DWORD var_index = TlsAlloc();
//set it to some value
TlsSetValue(var_index,10);
//get the value back
int var = TlsGetValue(var_index);
//release the index
TlsFree(var_index);
```

图 22.16 动态线程本地存储代码

为使用一个线程本地静态变量，应用程序按如下方式来声明变量，以确保每个线程都拥有它自己的私有拷贝：

```
--declspec (thread) DWORD cur_pos=0;
```

22.8 小 结

微软公司设计的 Windows XP 是一个具有可扩展性、可移植性的操作系统，也是一个利用了新技术和硬件优势的操作系统。Windows XP 支持多种操作环境和对称多处理技术，包括 32 位和 64 位处理器和 NUMA 计算机。Windows XP 使用内核对象提供基本服务，支持客户机－服务器计算机，进而支持多种广泛的应用环境。例如，Windows XP 可以运行为 MS-DOS、Win16、Windows 95、Windows XP 以及（或者是）POSIX 编译的程序。它提供了虚拟内存、集成高速缓存以及抢占式调度。Windows XP 提供了比以往微软公司的操作系统更为健壮的安全模式，并包含了国际化的功能。Windows XP 可在大量不同类型的计算机上运行，因此用户可以选择和更新硬件来匹配他们的预算及性能需求，而不需要改变他们运行的应用程序。

习题二十二

22.1 Windows XP 是个什么类型的操作系统？描述其两个主要特征。

22.2 列出 Windows XP 的设计目标。详细讨论两个。

22.3 描述 Windows XP 系统的引导过程。

22.4 描述 Windows XP 的体系结构的三个主要体系结构层次。

22.5 对象管理器的作用是什么？

22.6 句柄是什么，进程如何得到句柄？

22.7 描述虚拟内存管理器的管理方案。VM 管理器如何提高性能？

22.8 进程管理器提供什么类型的服务？什么是本地过程调用？

22.9 I/O 管理器的责任是什么？

22.10 管理 Windows XP 缓存的是什么？缓存是如何管理的？

22.11 Windows XP 提供什么用户模式进程，以允许 Windows XP 运行为其他操作系统所开发的程序？描述两个这样的子系统。

22.12 Windows XP 支持什么类型的网络？Windows XP 如何实现传输协议？描述两个网络协议。

22.13 NTFS 名称空间是如何组织的？描述之。

22.14 NTFS 如何处理数据结构？NTFS 如何从系统崩溃中恢复过来？在进行恢复之后什么要得以确保？

22.15 什么是进程，Windows XP 是如何管理它的？

22.16 Windows XP 如何分配用户内存？

22.17 描述应用程序可以通过使用 Win32 API 使用内存的方法。

推荐读物

Solomon 和 Russinovich[2000]概述了 Windows XP 并相当详细地描述了系统内部和组件的技术细节。Tate[2000]是关于使用 Windows XP 的很好参考。Microsoft Windows XP Server Resource Kit(Microsoft[2000b])共有 6 卷，对于使用和部署 Windows XP 非常有帮助。Microsoft Developer Network Library(Microsoft[2000a])每季度出版一次。它包括了有关 Windows XP 和微软公司其他产品的大量有用信息。

Iseminger[2000]是关于 Windows XP 活动目录的很好参考。Richter[1997]详细讨论了如何使用 Win32 API 来编写程序。Silberschatz 等[2001]中含有关于 B+树的很好的论述。

第二十三章　历史纵览

在第一章，简要地介绍了操作系统发展的历史渊源。这样的概述缺少细节的东西，因为当时还没有介绍操作系统的最基本的概念（如 CPU 调度、内存管理、进程等）。然而现在你已了解这些基本的概念，这样，就可以分析这些概念是如何应用在几个较老但却非常有影响的操作系统上的。像 XDS-940 或者 THE 系统这样的系统属于“一个是一类”（one-of-a-kind）系统，其他的诸如 OS/360 都被广泛应用的。介绍的顺序突出了系统的相似点和不同点，并不是严格按照年代或重要性的顺序排列。认真学习过这些操作系统的同学想必很熟悉它们。

当描述早期的一些系统时，列出了一些参考文献以供进一步阅读。无论是技术内容还是其风格，系统设计者所写的论文都非常重要。

23.1　早期系统

早期的计算机外形巨大，从控制台运行。程序员，同时也是计算机系统的操作员，写了程序之后，直接从操作员的控制台对该程序进行操作。首先，把程序以人工方式从前端面板进行开关操作（一次一个指令），将纸带或者穿孔卡片装入内存。然后，按下适当的按键设定开始地址并开始执行该程序。当程序运行时，程序员或者是操作员通过控制台的显示灯来监控程序的执行。如果发现错误，程序员就可以暂停该程序，检查存储器和寄存器的内容，并直接从控制台调试该程序。结果被打印输出，或者是在纸带和卡上打孔，以便于以后打印。

随着时间的推移，人们开发了一些额外的软件和硬件。读卡机、行式打印机、磁带变得很普遍。汇编程序、装入程序、连接程序的设计成功大大减轻了程序设计的负担。创建了通用函数库，可以把通用函数复制到一个新的程序，而不用重新再写一遍，提供了软件可重用性。

执行输入/输出的程序相当重要。每个新的 I/O 设备都有自己的特性，要求进行仔细的程序设计。一种叫做设备驱动程序的特殊的子程序，就是为每个 I/O 设备而写的。设备驱动程序知道如何使用一个具体设备的缓冲器、标记、寄存器、控制位和状态位。每种类型的设备都有自己的驱动程序。一个简单的任务，比如从纸带读出器读取字符，就会涉及到特定设备的一系列复杂操作。设备驱动程序并不是每次都需要写一些必要的代码，而只需从库中使用它即可。

后来，出现了 FORTRAN、COBOL 和其他语言的编译器，编译器使编程变得更加容易，

但是计算机的操作却变得更复杂了。例如，为了执行一个 FORTRAN 语言程序，程序员首先需要把 FORTRAN 编译器装到计算机上。编译器通常保留在磁带上，因此必须把专门的磁带装到一个磁带驱动器上。程序通过读卡机读出，并写到另一个磁带上。FORTRAN 编译器生成汇编语言输出，它需要进行汇编编译。这个过程需要汇编程序装入其他的磁带，还需要连接汇编程序的输出以支持库程序。最后，程序的二进制结果格式就可以执行了。和以前一样，可以把它装入到内存并从控制台对其进行调试。

运行一个作业包含相当长的**启动时间**(set-up time)。每个作业由许多独立的步骤组成：

1. 装入 FORTRAN 编译程序带
2. 运行编译程序
3. 卸载编译程序带
4. 装入汇编程序带
5. 运行汇编程序
6. 卸载汇编程序带
7. 装入目标程序
8. 运行目标程序

如果在任意一步中发生错误，则程序员或者是操作员不得不重新开始启动。每个作业步骤可能都包括装入和卸载磁带、纸带和穿孔卡片。

作业的启动时间确实是个问题。当装入磁带时，或者程序员在操作控制台时，CPU 处于空闲状态。应该记住的是，在早期计算机很少并且很昂贵。一台计算机要花费数以百万美元，而且不包括运作费用，诸如电源、冷却、程序员等。这样，计算机时间就变得非常宝贵，机主希望尽可能多地使用计算机。他们需要和从其投资中得到尽可能高的**利用率**。

此方案具有两重性。首先，要雇用一个专业的计算机操作员。程序员就不再操作机器。当一个作业完成时，操作员就开始下一个。因为操作员比程序员更具有装带的经验，从而减少了安装时间。程序员提供所有需要的卡或带，及如何运行该项作业的一个简短描述。当然，操作员不能在控制台对一个不正确的程序进行调试，因为操作员不懂程序。因此，万一程序出错，就会占据一堆存储器和寄存器，程序员不得不从中进行调试。清除存储器和寄存器可以让操作员立即继续进行下一个作业，但留给程序员的是更棘手的调试问题。

第二，有相似要求的作业可以捆绑在一起，作为一个组在计算机上运行，以减少安装时间。例如，假设操作员分别收到一个 FORTRAN 作业、一个 COBOL 作业和另外一个 FORTRAN 作业。如果她以那样的顺序运行，那么她将不得不先安装 FORTRAN(如装入编译程序带等)，然后安装 COBOL，再安装 FORTRAN。但是，如果她把两个 FORTRAN 作业当做一个批处理来运行的话，那么她只需要安装一次 FORTRAN，节省了操作时间。

但是还存在问题。例如，当一个作业结束时，操作员必须发布通知说该作业已停止(通过观察控制台)，确定*为什么*停止(是正常终止还是非正常终止)，(需要时)清理存储器和寄

存器，为下一个作业装入合适的设备，并重启计算机。在作业转换过程中，CPU 是闲置的。

为克服时间闲置的问题，人们开发了**自动作业定序方法**；利用这个技术，创建了第一个基本操作系统。一个被称为**常驻监督程序**的小程序，它被创建用于在作业之间的自动地传递控制（见图 23.1）。常驻监督程序总是在存储器里（或者说是**常驻**）。

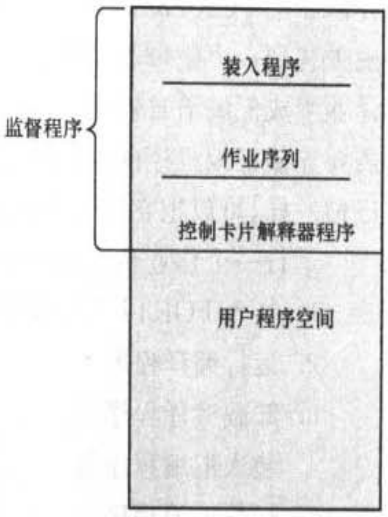


图 23.1 常驻监督程序的内存布局

计算机一打开，常驻监督程序就被激活了，并把控制传递给了程序。当程序终止时，就把控制返回给常驻监督程序，然后它又继续用于下一个程序。这样，常驻监督程序就可以在程序之间和作业之间自动排序。

但是常驻监督程序又是如何知道应该执行哪个程序呢？以前，操作员手头都会有一份简单说明书，说明什么样的程序将在什么样的数据上运行。**控制卡片**的引入使得可以直接向监督程序提供这些信息。这个思路很简单：除了一个作业的程序或数据之外，程序员还添加了控制卡片，控制卡片包含了指示程序运行的对常驻监督程序的指令。例如，假设一个正常的用户程序需要运行下面三个程序中的一个：FORTRAN 编译程序（FTN）、汇编程序（ASM）和用户程序（RUN）。对于每个程序可以使用一个不同的控制卡片：

$FTN——执行 FORTRAN 编译程序

$ASM——执行汇编程序

$RUN——执行用户程序

以上的卡片将会告诉常驻监督程序运行哪一个程序。

可以使用另外的两个控制卡片来界定每个作业的界限。

$JOB——作业的第一个卡片。

$END——作业的最后个卡片。

这两个卡片对程序员分配机器资源很有用处。参数可用于定义作业名字，账号管理等。其他类型的控制卡片可以定义为其他的功能，如请求操作员装载或者是卸载带等。

控制卡片有一个问题，就是如何来区分数据卡片和程序卡片。通常的方法是通过卡片上的某一个特殊字符或格式来标识它们。很多系统都在第一栏利用美元号（$）来标识一类控制卡片。其他的使用不同的代码。IBM 公司的作业控制语言（简称 JCL）在前面两列使用斜杠标志（//）。图 23.2 中的例子显示了一个简单的批处理系统卡片组构造。

常驻监督程序有这样几个明确的部分：

- **控制卡片解释程序**。负责在执行时读出和执行卡片上的指令。
- **装入程序**。它由控制卡片解释程序激活，以不断地把系统程序和应用程序装入内存。

• **设备驱动程序**。它既可以被控制卡片解释程序调用也可以被装入程序调用，以使系统的 I/O 设备执行 I/O 操作。系统程序和应用程序经常被连接到相同的设备驱动程序，这样保持了操作的连贯性，同时节省内存空间和程序设计的时间。

这些批处理系统用处非常大。常驻监督程序提供了如控制卡片展示那样的自动作业排序方法。当一个控制卡片显示要运行某一程序时，监督程序就把程序装入到内存并传递控制权给它。当程序运行结束时，它就把控制权传回监督程序，这样监督程序就可以读取下一个控制卡片，并装入适当的程序，等等。在所有的控制卡片得到适合该项作业的解释之前这样的循环会不断重复。然后，监督程序就自动继续下一项作业。

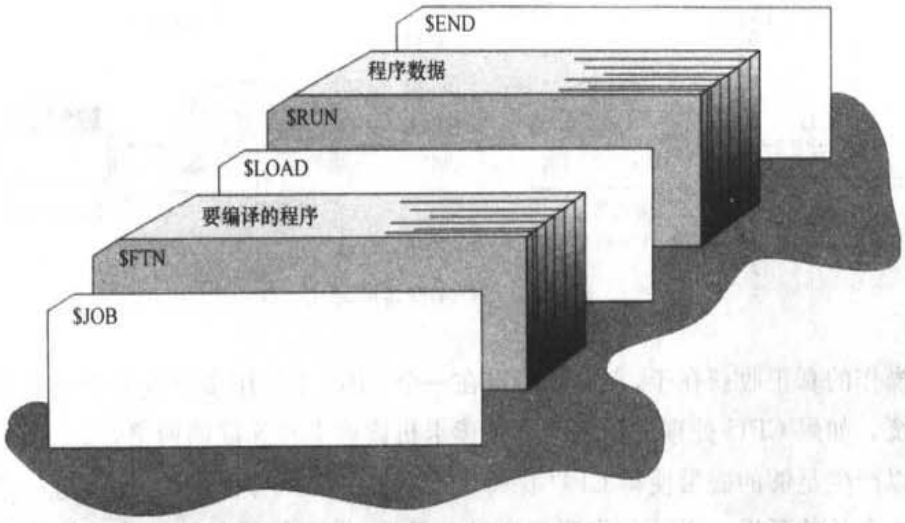


图 23.2 一个简单批处理系统的卡片板

用自动作业排序法对批处理系统做一个转换可以提高性能。这个问题非常简单，即人与计算机比较起来，人要慢得多。因此，通过操作系统软件来替代人工操作就很有必要。自动作业排序法消除了人工的安装时间和作业排序。

正如 1.2.1 小节所指出的一样，甚至使用了这样的安排，CPU 还是经常闲置。问题在于机械 I/O 设备的速度，机械 I/O 设备本身就比电子设备来得慢。即使是一个较慢的 CPU 也可以运行在毫秒的范围内，每秒执行成千上万计的指令。另一方面，一个快速读卡器每分钟可以读取 1 200 张卡，即每秒钟读取 20 张卡。这样，CPU 及其 I/O 设备之间在速度上的差别将会是三个以上的数量级或者是更多。当然，随着时间的推移，技术的进步导致了更加快速的 I/O 设备的出现。不幸的是，CPU 的速度增长得越快，问题非但没有得到解决，反而是恶化了。

一个普通的方法是把速度比较慢的读卡机(输入设备)和行式打印机(输出设备)用磁带设备替换掉。20 世纪 50 年代末期和 60 年代早期，大多数的计算机系统都是批处理系统，它们从读卡机中读取并写到行式打印机或者卡片穿孔机中。并不是让 CPU 直接从卡片中读取，而是首先需要通过一个单独的设备把卡片复制到一条磁带上。当磁带足够满时，就把它

卸下并转载到计算机。当需要输入一个卡片到某一程序，就要从磁带上读取相当的记录。类似地，把输出结果写到磁带，可以以后再打印磁带内容。读卡机和行式打印机都是*脱机*操作，而不是通过主机(见图 23.3)。

脱机操作的主要优点是主机不再受读卡机和行式打印机速度的制约，而是受限于快得多的磁带设备的速度。对所有 I/O 使用磁带的这种技术可以应用在任何类似的设备上(如读卡机，卡片穿孔机，绘图仪，纸带，打印机。)

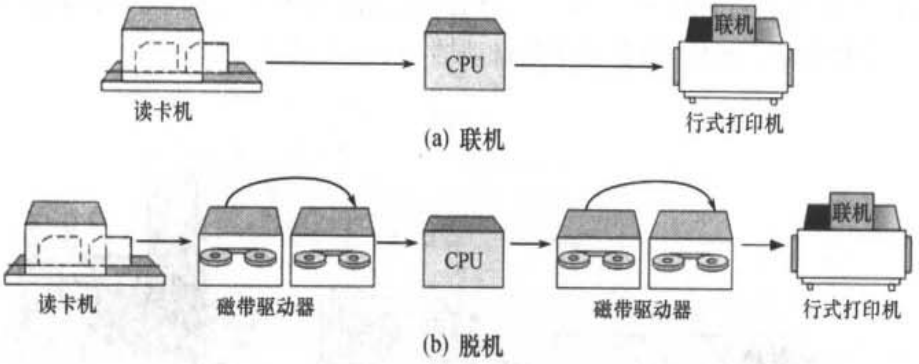


图 23.3 I/O 设备的操作

脱机操作的真正收获在于，它使得可以在一个 CPU 上应用多个读卡机到磁带和磁带到打印机系统。如果 CPU 处理的输入速度是读卡机读取卡片速度的两倍，那么两个读卡机同时工作可以产生足够的磁带使得 CPU 保持忙碌状态。另一方面，现在要运行一个特殊作业时就出现一个长的延迟。必须首先读到磁带。然后，必须等到有足够的其他作业被读到磁带上来"填满"它。磁带必须重绕、卸载、手工装载到 CPU，并把它安装到一个空带驱动器上。当然，对于批处理系统来说是很合理的。许多相似的作业在导入计算机之前可以被成批地放到磁带上。

虽然作业的脱机准备会持续一些时间，但是在大多数系统中可以很快地被替代。磁盘系统变得垂手可得，并大大改进了脱机操作。磁带系统的问题在于，当 CPU 对磁带的一端进行读操作时，读卡器就不能在磁带的另一端写。因为磁带从本质上讲是一种**顺序存取设备**，所以在倒带和读整个磁带之前必须先写。磁盘系统消除了这个问题，它属于一种**随机存取设备**。因为磁头可以从磁盘的一个区域滑动到另一个区域，磁盘可以快速地从磁盘上正在被读卡机使用的用于存储新卡片的区域，转移到 CPU 读取下一张卡片所需要的位置上。

在磁盘系统中，卡片直接从读卡机读到磁盘上。卡片映像的位置被记录在由操作系统保管的表格中。当执行某一作业时，操作系统就可以通过从磁盘读取来满足该作业的读卡机输入请求。同样地，当作业要求打印机输出一行时，该行就被复制到一个系统缓冲区中并被写到磁盘。当完成作业时，实际上已经打印出了输出结果。这种处理形式被称为**假脱机**(spooling)(见图 23.4)；spooling 是 simultaneous peripheral operation on-line 的缩写。从本质上讲，假脱机利用磁盘作为一个巨大的缓冲区，以便于在输入设备上尽可能地提前读

取，也便于存储输出文件直到输出设备能够接收它们。

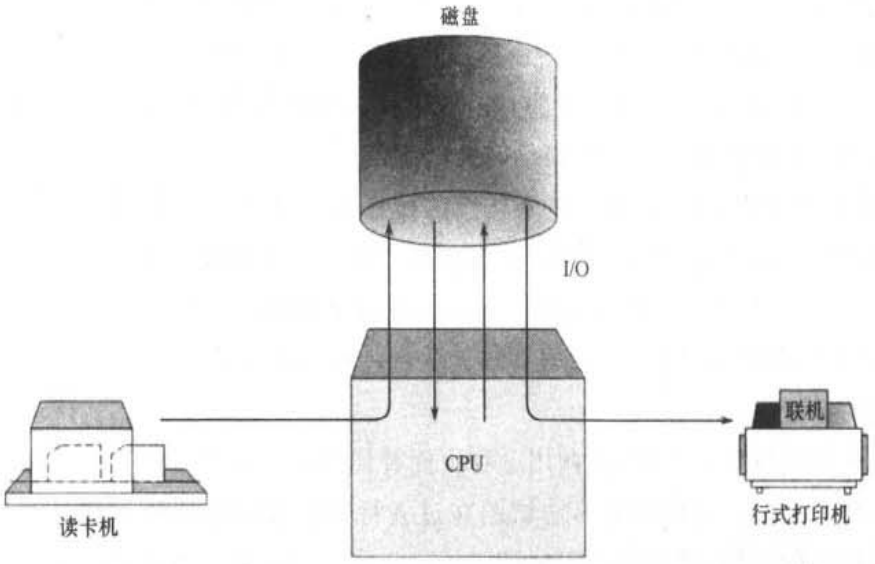


图 23.4 假脱机

假脱机技术也用于在远程站点处理数据。CPU 通过通信路径把数据传送到某一远程打印机(或者是从远程读卡机接收一项完整的输入作业)。远程处理以它自己的速度完成，不需要 CPU 的干涉。当处理过程结束时需要通知 CPU，这样才可以对下一批数据进行假脱机操作。

假脱机技术可以把一项作业的 I/O 与其他多项作业的计算交叠起来进行。即使是一个简单的系统，在打印不同作业的输出结果时，spooler 可能在读取另一项作业的输入。在此期间，还执行其他的某项作业(或者是多项作业)，从磁盘读取他们的“卡片”，并把它们的输出行“打印”到磁盘。

假脱机技术在系统性能方面具有一个直接有益的作用。用一些磁盘空间和表、一项作业的计算就可以与其他多项作业的输入、输出交叠进行。这样，假脱机技术就可以保持 CPU 和 I/O 设备以更高的速率运行。

假脱机技术很自然地导致了多道程序设计技术，它是所有现代操作系统的基础。

23.2 Atlas

Atlas 操作系统诞生于 20 世纪 50 年代末和 60 年代早期英格兰的曼彻斯特大学。它的很多在当时是很新奇的基本功能已成为了现代操作系统的标准部分。设备驱动程序是该系统的主要部分。另外，通过一组被称为*额外代码*(extra codes)的特殊指令增加了系统调用。

Atlas 是具有假脱机技术的批处理操作系统。假脱机技术使得系统根据外围设备的有效性来调度作业成为可能，外围设备有磁带机、纸带读卡机、纸带穿孔机、行式打印机、卡片

阅读机、卡片穿孔机。

然而 Atlas 最显著的功能是它的内存管理。**核心存储器**(core memory)在当时很新也很贵。像 IBM 650 的许多计算机都使用磁鼓作为最基本的存储器。虽然 Atlas 系统把磁鼓作为它的主存储器,但是它拥有少量的核心存储器用做磁鼓的一个高速缓存器。按需调页技术用于在核心存储器和磁鼓之间自动转换信息。

Atlas 系统曾使用一台 48 位字的英国产的计算机。地址是 24 位的,但以十进制方式编码,这样只允许一百万字的寻址空间。在当时这已算是一个超级大的地址空间了。Atlas 的物理内存是一个 98 KB 的字磁鼓以及一个 16 KB 的字内核。内存被分割成多个 512 字的页面,规定了物理内存的 32 帧。32 个寄存器的联想内存实现了从一个虚拟地址到一个物理地址之间的映射。

如果某一页面发生了故障,就调用一种页面替代算法。存储器帧通常是空的,所以磁鼓转换器可以马上启动。页面替代算法试图在过去行为的基础上预测未来的内存访问行为。当一个帧被访问的时候,该帧的引用位被置位。每隔 1 024 条指令就把引用位读入内存,并且将保留这些位的最后 32 个值。这种做法用来定义时间,把最近的参考时间点定义为 t_1,把最后的两个参考时间之间的间隔定义为 t_2。选择用于替换的页面遵循如下的规则:

1. 任意页面如果有 $t_1 > t_2+1$,则认定不再使用该页面。
2. 如果所有的页面都有 $t_1 \leqslant t_2$,那么把该页面用 $t_2 - t_1$ 的值最大的那个页面替换。

页面替代算法假定程序能够循环访问内存。如果最后两次引用的时间间隔为 t_2,那么另外一次引用应该在 t_2 时间单元后。如果引用没有出现($t_1 > t_2$),由此可以假定该页面不再被使用,并替换该页面。如果所有的页面都在使用,那么在最长时间里不需要使用的页面就被替换掉。而下一个引用时间就应是 $t_2 - t_1$。

23.3 XDS-940

XDS-940 操作系统(Lichtenberger 和 Pirtle [1965])设计于美国加利福尼亚大学伯克利分校。和 Atlas 操作系统一样,它在内存管理中使用了分页技术,不同的是,XDS-940 是一个分时操作系统。

分页技术只用于重定位;在按需调页中并没有用到。任何用户进程的虚拟内存都只有 16 KB 的字,而物理内存则有 64 KB 的字。每个页面是 2 KB 字。页表存放在寄存器中。因为物理内存比虚拟内存大,所以同时可容纳多个用户进程。当页面包含了只读的可重入代码后,通过共享页面可以增加用户数量。进程存放在磁鼓里,根据需要从内存中调进和调出。

XDS-940 系统是从改进过的 XDS-930 系统中构建起来的。它对基本计算机做了一些典型的改变,使得操作系统可以编写得更加合适。增加了用户监督模式。某些指令,像 I/O 和 Halt,被定义为享有特权。操作系统限制在用户模式下尝试执行一个特权指令。

在用户模式指令集中加入了一条系统调用指令。这条指令用于创建新资源，如文件，使得操作系统可以管理物理资源。例如，文件被分配在磁鼓中 256 字的物理块中。位图(bit map)用于管理空闲的磁鼓区块。每个文件都有一个索引块，它指向实际的数据块。索引块是链接在一起的。

XDS-940 同时也规定了允许进程新建、启动、暂停以及消除子进程的系统调用。任何一个程序员都可以构建一个系统进程。分离的进程可以共享内存以实现通信和同步。进程的创建过程定义了一个树型结构，进程是树的根节点而它的子进程则是树下面的枝节。每个子进程可以依次创建更多的子进程。

23.4 THE

THE 操作系统(Dijkstra[1968]，McKeag 和 Wilson[1976])设计于荷兰 Eindhoven 的 Technische Hogeschool。它是一个批处理系统，运行在一台荷兰产计算机 EL X8 上，带有 27 位字的 32 KB。此系统因它清晰的设计而著名，特别是它的层结构，及使用了一组引入信号量以用于同步的并发进程。

但是，与 XDS-940 不一样的是，THE 系统中的进程集合是静态的。操作系统本身被设计为一组协同进程。另外，创建了 5 个用户进程担任活动代理，用于编译、执行以及打印用户程序。当完成一项作业时，该进程将返回到输入队列中去选择另一作业。

系统使用基于优先权的 CPU 调度算法。优先权在每隔两秒钟就要重新计算一次，并且与最近所使用的 CPU 时间成反比(在最后的 8 s 至 10 s)。此方案给了 I/O 约束的进程和新进程以更高的优先权。

内存管理会因为缺少硬件支持而受到限制。然而，由于系统受到限制，并且只能用 Algol 写用户程序，因此使用了一种软件页面调度方法。Algol 编译程序自动生成对系统程序的调用，这些调用保证了所请求的信息都在内存中，如有必要则可交换。备份存储器是一个 512 KB 字的磁鼓。使用了一个 512 字的页面和带 LRU 页面替换策略。

另外，THE 系统的主要关注点是死锁控制。银行家算法提供了避免死锁的办法。

与 THE 系统紧密关联的是 Venus 系统(Liskov[1972])。Venus 系统也采用一种分层结构的设计，使用信号量机制来同步进程。该设计中的低层部分是用微码来实现的，但是却提供了一个更快速的系统。内存管理被改成使用段页式存储器。系统同时也是作为一个分时系统来设计的，而不仅仅是一个批处理系统。

23.5 RC 4000

RC 4000 系统，与 THE 系统一样，基本上是以它的设计概念而出名的。它是由

Regnecentralen 特别是 Brinch-Hansen(Brinch-Hansen[1970,1973])为丹麦产的 RC 4000 计算机而设计。其目标并不是设计一个批处理系统或一个分时系统,也不是其他的特殊的系统。它的真正的目的是想建立一个操作系统的核心程序,或者是内核,在此基础上可以构建一个完整的操作系统。这样,系统结构就被分层了,只提供了较低的层—— 内核。

内核支持多个并发进程的集合。采用了轮转法的 CPU 调度器来支持进程。虽然进程可以共享内存,但是主要的通信和同步机制还是采用由内核提供的**消息系统**。进程间的通信是通过交换长度固定为 8 个字的消息。所有的消息保存在从一个公共的缓冲池中取出的各个缓冲器中。当不再需要使用某个消息缓冲器时,则把它返回到公共缓冲池中。

消息队列是每个进程相关联的。它包含了所有已发往进程但进程还未接收到的消息。消息以 FIFO 的方式从队列中移出。系统支持 4 个原语操作,这些操作是自动执行的:

- **send-message** (**in** *receiver*, **in** *message*, **out** *buffer*)
- **wait-message** (**out** *sender*, **out** *message*, **out** *buffer*)
- **send-answer** (**out** *result* , **in** *message*, **in** *buffer*)
- **wait-answer** (**out** *result*, **out** *message*, **in** *buffer*)

最后的两步操作允许进程一次交换多条消息。

这些原语要求进程以 FIFO 的方式处理消息队列,同时,当有其他进程在处理它们的消息时进程会自动阻塞起来。为消除这些限制,开发者提供了两条额外的通信原语,这样进程就可以以任意的方式来等待下一个报文的到来或者是回应、服务报文队列。

- **wait-event** (**in** *previous-buffer*, **out** *next-buffer* , **out** *result*)
- get-event (out *buffer*)

I/O 设备也是作为进程来处理的。设备驱动程序是一些代码,这些代码可以把设备中断和寄存器转换到消息中,这样,通过向终端发送消息的方式进程就可以写某一终端。设备驱动程序将会接收到消息,并把字符输出到终端。一个输入字符会导致系统中断,并会传输到设备驱动程序上。设备驱动程序将从输入的字符中创建一个消息并把它发送到一个等待的进程中。

23.6 CTSS

CTSS(Compatible Time-Sharing System)系统设计于美国麻省理工学院,当时是作为一个实验性质的分时系统。它在一台 IBM 7090 计算机上实现,最终可以同时支持多达 32 个交互式用户。系统向用户提供一组交互式的命令,这样他们就能通过终端来处理文件、编译和运行程序。

这台 7090 计算机有一个 32 KB 的内存,由 36 位字组成。监督程序用去了 5 KB 字,留下 27 KB 给用户。用户的内存映象在内存与一个快速磁鼓之间进行交换。CPU 调度使用

多级反馈队列算法。第 i 级的时间总额为 $2\times i$ 个时间单位。如果程序在一个时间总额里没有完成它的 CPU 脉冲，它将被转移到队列的下一层，得到两倍的时间。位于最高级的（拥有最短的时间总额）程序会先执行。一个程序的最初级别是由它的大小决定的，时间总额至少应该和交换时间等长。

CTSS 取得了极大的成功，到 1972 年时已被广泛应用。虽然它也有局限性，但是，它成功地展示了分时是一种简便而实用的计算模式。CTSS 带来的一种结果是分时系统的快速发展。另一个结果是 MULTICS 的发展。

23.7 MULTICS

MULTICS 操作系统（Corbato 和 Vyssotsky[1963]，Organick[1972]）设计于美国麻省理工学院，它是 CTSS 的一个自然扩展的产品。CTSS 和其他早期分时系统是如此的成功，以至于产生了一种继续快速开发一个更大和更好系统的强烈需求。当出现了大型计算机后，CTSS 的设计者们就开始创建一个分时的公用程序。可以像电力一样提供计算服务。大型的计算机系统通过电话线连接到整个城市的家庭和办公室的终端上。操作系统将是一个连续运行的分时系统，有一个可以共享程序和数据的、巨大的文件系统。

MULTICS 是由一个来自美国麻省理工学院的团队、GE 公司（后来它把计算机部门卖给了 Honeywell 公司）和贝尔实验室（1969 年退出该项目组）共同设计的。基本的 GE 635 计算机主要通过额外的段页式存储硬件被改进为一个新的计算机系统，被称为 GE 645。

虚拟地址由一个 18 位的段号和一个 16 位字的偏移量组成。段被分成大小为 1 KB 字的多个页面。使用了二次机会页面置换算法。

分段式虚拟地址空间被合并到了文件系统中，每段就是一个文件。段是根据文件名来寻址的。文件系统本身就是一个多级的树型结构，允许用户建立他们自己的子目录结构。

和 CTSS 一样，MULTICS 运用了多级反馈队列进行 CPU 调度。通过与每个文件相关联的访问列表，及对执行进程的保护组环来实现保护。该系统几乎全部是用 PL/1 语言编写，有 300 000 行代码。它被扩展到一个多处理机系统后，可以在系统继续运行的情况下从服务中抽出一个 CPU 用于维护。

23.8 OS/360

操作系统发展过程中战线最长的无疑是 IBM 公司计算机上的操作系统。早期的计算机，如 IBM 7090 和 7094，是普通 I/O 子程序发展的主要代表。接下来是常驻监督程序、优先权指令，内存保护以及简单的批处理。这些系统是在不同的场合单独开发出来的。结果，IBM 公司要面临许许多多不同的计算机、不同的语言以及不同的系统软件。

IBM/360 就是为了改变这种状况而设计的。IBM/360 当初的设计目标是一系列的计算机，包含从小型商务计算机到超大型科研用计算机的整个领域。这些系统只需要一组软件，都使用同样的操作系统：OS/360（Mealy 等[1966]）。这个举措是想减少 IBM 公司的维护问题，并使得用户可以把程序和应用程序从一个 IBM 系统向另一个 IBM 系统自由移动。

遗憾的是，OS/360 试图为人们做任何事，但结果所做的任务没有一项是特别好的。文件系统包含了定义每个文件类型的类型域，为固定长度和非固定长度的记录文件、分块和不分块的文件定义了不同的文件类型。在该系统中使用了连续分配的方法，这样用户不得不去猜每个输出文件的大小。作业控制语言（JCL）给每个可能的选项都加上了参数，使得普通用户不能理解。

存储器管理程序受体系结构的牵制。虽然用上了基址寄存器寻址方式，但是程序还是可以访问和修改基址寄存器，因此 CPU 就生成了绝对地址。这种安排防止了动态的重定位。在装入时程序只限定在物理内存中。两个独立版本的操作系统产生了：OS/MFT 运用了固定区域和 OS/MVT 运用了变化的区域。

该系统是由成千上万的程序员用汇编语言写成的，结果有数百万行的代码。操作系统本身的代码和表格就需要大量内存。操作系统本身的开销常常耗费了全部 CPU 时钟周期的一半。在过去的几年里开发了新的版本，旨在增加新的功能和纠错。然而，纠正了一个错误经常导致系统中某些远程部分出现另一个错误。因此系统中经常会有一定数量已知的错误。

OS/360 在 IBM 370 体系结构中增加了虚拟内存。底层的硬件提供了一个段页式的虚拟内存。OS 的新版本对这个硬件有多种不同的用法。OS/VS1 建立了一个很大的虚拟地址空间，并在虚拟内存中运行 OS/MFT。这样，操作系统本身也就被分页了，还有用户程序。OS/VS2 的第一个版本是在虚拟内存中运行 OS/MVT 的。最后，OS/VS2 的第二个版本（现称为 MVS）为每个用户提供他自己的虚拟内存。

MVS 基本上还是一个批处理操作系统。CTSS 系统是在 IBM 7094 上运行的，但是美国麻省理工学院认为 IBM 7094 的后续产品 360 的地址空间，对 MULTICS 系统来说实在太小了，因此他们改变了供应商。然后，IBM 公司决定创建自己的分时系统，即 TSS/360（Lett 和 Konigsford[1968]）。和 MULTICS 系统一样，TSS/360 应该是一个巨大的分时公用系统。但是基本的 360 体系结构被修改成 67 模型以提供虚拟内存。有几个站点在 TSS/360 之前购买了 360/67 。

然而 TSS/360 被延误了，在 TSS/360 出来之前，就有其他的分时系统被开发出来以作为临时系统。OS/360 中加入了一个分时选项（TSO），IBM 公司的剑桥科学中心开发了 CMS 系统作为一个单用户系统，以及 CP/67 提供了在其上运行的虚拟机（Meyer 和 Seawright[1970]，Parmelee 等[1972]）。

TSS/360 终于面世了，但它却失败了。它太庞大而且太慢。结果，没有一个站点从临时

系统转向 TSS/360 系统。现在，IBM 系统上的分时大部分或者由 MVS 下的 TSO 提供，或者由 CP/67(又称做 VM)下的 CMS 提供。

TSS/360 和 MULTICS 系统到底出了什么问题？问题的一部分在于这些高级系统太大、太复杂而不容易理解。另外一个问题是，对通过分时可以从一个大型的远程计算机上获得计算能力的这样一个假设。现在看来，大多数的计算只需要在小型的单机(即个人计算机)来完成，而不是由试图要为用户做任何事情的超大型的、远程分时系统来完成。

23.9 Mach

Mach 操作系统的祖先是 Accent 操作系统，该系统是在美国卡内基·梅隆大学研制成功的 Rashid 和 Robertson[1981]。Mach 的通信系统和哲学理念是从 Accent 中衍生出来的，但是系统的许多其他的重要部分(如虚拟内存系统、任务和线程管理)是从 scratch (Rashid[1986]，Tevanian 等[1987b]和 Accetta[1986])中发展而来的。Tevanian 等[1987a]和 Black[1990]详细描述了 Mach 的调度程序。Mach 早期的版本中存在的共享内存和内存分页技术是 Tevanian 等[1987b]提出的。

Mach 操作系统是根据下面三个重要的目标来设计的：

- 模仿了 4.3BSD UNIX 系统，因此 UNIX 系统中的可执行文件可以在 Mach 系统正确地运行。
- 是一个现代的操作系统，可以支持多种内存模型、并行和分布式计算。
- 有一个比 4.3BSD 更简单以及更容易修改的内核。

从 BSD UNIX 系统开始，Mach 的发展走过了一条不断创新的路程。Mach 代码起初是在 4.2BSD 内核中发展起来的，当完成 Mach 组件时，BSD 的内核组件就逐渐被 Mach 组件代替。当 BSD 组件更新至 4.3BSD 时就得到了 Mach 的组件。到了 1986 年，虚拟内存和通信子系统已运行在 DEC VAX 系列计算机上了，包括 VAX 的多处理机版本。紧接着，就出现了 IBM RT/PC 的版本和 SUN 3 工作站的版本。1987 年，Encore Multimax and Sequent Balance 多处理机的版本也完成了，包括任务和线程支持以及第一个官方的系统版本，分别为版本 0 和版本 1。有了版本 2 之后，通过在内核里包含多数的 BSD 代码，Mach 实现了与相关 BSD 系统的兼容。Mach 的一些新特性和新功能使得它的内核要比相应的 BSD 内核大。Mach 3 把 BSD 代码移到了内核之外，剩下了一个小得多的微内核。系统只执行内核中基本的 Mach 功能。所有 UNIX 相关的代码都放到用户态的服务程序中执行。将 UNIX 相关的代码排除在内核之外，这样就可以用其他的操作系统来替代 BSD，或者是在微内核的顶部同时执行多个操作系统的接口。除了 BSD 系统之外，用户态的执行也朝着 DOS 系统、Macintosh 操作系统和 OSF/1 系统的方向发展。此方法与虚拟机的概念相似，但是，虚拟机是由软件来定义的(Mach 内核接口)，而不是硬件。从第 3 版开始，Mach 在许多种类的系统

中都可以用，包括单处理机 SUN、Intel、IBM、DEC 机器和多处理机 DEC、Sequent 和 Encore 系统。

当开放软件基金会(简称 OSF)在 1989 年宣布将使用 Mach 2.5 作为它新的基本操作系统(OSF/1)时，Mach 引起了前沿产业的注意。最初的 OSF/1 版本出现在一年之后，现在已完善为 UNIX System V，第 4 版，即在 UNIX 国际会员中入选的操作系统。OSF 会员包括许多重要的技术公司，如 IBM、DEC 还有 HP。OSF 后来改变了方向，只有 DEC UNIX 还是基于 Mach 内核的。

Mach 2.5 也是 NeXT 工作站的操作系统的基础。它是苹果电脑公司有名的 Steve Jobs 的智慧的结晶。UNIX 在发展过程中并不关注多处理器技术，与之相反，Mach 加入了多处理器技术以支持整个系统。从共享内存系统到进程间无内存可共享的系统，它的多处理器技术支持都是非常灵活的。Mach 使用轻量级进程，在一个任务(或者是地址空间)中以多线程方式执行，从而支持多处理器技术和并行计算技术。作为惟一通信方式的消息的广泛应用保证了保护机制的彻底性和有效性。通过集成消息到虚拟内存，Mach 也能保证消息可以被有效地处理。最后，通过让虚拟内存使用消息与管理后备存储的后台程序进行通信，Mach 在设计和实现这些**内存目标管理**的任务上提供了很大的灵活性。通过提供低级的，或者是说原始的系统调用，Mach 缩小了内核的大小同时允许了用户级的操作系统仿真，很像 IBM 公司的虚拟机系统。

《操作系统概念》的以前版本中专门设有 Mach 系统的章节。出现在第四版的这一章在网上也可浏览到。(http://www.bell-labs.com/topic/books/os-book/Mach.ps)。

23.10 其他系统

当然，除了以上介绍过的几种操作系统外，还有另外的操作系统，而且它们中的大多数具有一些有趣的特性。Burroughs 计算机家族的 MCP 操作系统是第一个用系统程序设计语言编写的系统。它支持分段和多 CPU。用于 CDC 6600 的 SCOPE 操作系统也是一个多 CPU 的系统。多进程的协同和同步的设计是出乎意料的好。Tenex 是早期用于 PDP－10 上的一个按需调用系统，PDP－10 对后来的分时系统有巨大影响，如 DEC－20 的 TOPS－20。适用于 VAX 的 VMS 操作系统是基于 RSX 操作系统的，因为 PDP－11 CP/M 是适用于 8 位微机的最普遍的操作系统，今天这个系统已很少存在了。MS-DOS 是在 16 位微机上应用最普遍的操作系统。**图形用户接口**(Graphical User Interface，GUI)使得计算机用起来更加方便，也越来越受到欢迎。Macintosh 操作系统和 Microsoft Windows 是这个领域的两大权威。

参考文献

[**Abbot 1984**] C. Abbot, "Intervention Schedules for Real-Time Programming", *IEEE Transactions on Software Engineering*, Volume SE-10, Number 3(1984), pages 268～274.

[**Accetta et al. 1986**] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. T. Jr. , and M. Young, "Mach: A New Kernel Foundation for Unix Development", *Proceedings of the Summer USENIX Conference*(1986), pages 93～112.

[**Agrawal and Abbadi 1991**] D. P. Agrawal and A. E. Abbadi, "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, Volume 9, Number 1(1991), pages 1～20

[**Ahituv et al. 1987**] N. Ahituv, Y. Lapid, and S. Neumann, "Processing Encrypted Data", *Communications of the ACM*, Volume 30, Number 9(1987), pages 777～780.

[**Ahmed 2000**] I. Ahmed, "Cluster Computing: A Glance at Recent Events", *IEEE Concurrency*, Volume 8, Number 1(2000).

[**Akl 1983**] S. G. Akl, "Digital Signatures: A Tutorial Survey", *Computer*, Volume 16, Number 2(1983), pages 15～24.

[**Akyurek and Salem 1993**] S. Akyurek and K. Salem, "Adaptive Block Rearrangement", *Proceedings of the International Conference on Data Engineering* (1993), pages 182～189.

[**Alt 1993**] H. Alt, "Removable Media in Solaris", *Proceedings of the Winter USENIX Conference* (1993), pages 281～287.

[**Anderson et al. 1989**] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for SharedMemory Multiprocessors", *IEEE Transactions on Computers*, Volume 38, Number 12(1989), pages 1631～1644.

[**Anderson et al. 1991**] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the Userlevel Management of Parallelism", *Proceedings of the ACM Symposium on Operating Systems Principles*(1991), pages 95～109

[**Apple 1987**] *Apple Technical Introduction to the Macintosh Family*, Addison Wesley(1987).

[**Apple 1991**] *Inside Macintosh*, *Volume VI*. Addison-Wesley(1991).

[**Artsy 1989**] Y. Artsy, "Designing a Process Migration Facility: The Charlotte Experience", *Computer*, Volume 22, Number 9(1989), pages 47～56.

[**Asthana and Finkelstein 1995**] P. Asthana and B. Finkelstein, "Superdense Optical Storage", *IEEE Spectrum*, Volume 32, Number 8(1995), pages 25～31.

[**Axelsson 1999**] S. Axelsson, "The Base-Rate Fallacy and Its Implications for Intrusion Detection",

Proceedings of the ACM Conference on Computer and Communication Security (1999), pages 1~7.

[**Bach 1987**] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall (1987).

[**Back et al. 2000**] G. Back, P. Tullman, L. Stoller, W. C. Hsieh, and J. Lepreau, "Thechniques for the Design of Java Operating Systems", *Techniques for the Design of Java Operating Systems* (2000).

[**Balkovich et al. 1985**] E. Balkovich, S. R. Lerman, and R. P. Parmelee, "Computing in Higher Education: The Athena Experience", *Communications of the ACM*, Volume 28, Number 11 (1985), pages 1214~1224.

[**Bar 2000**] M. Bar, *Linux Internals*, McGraw-Hill (2000).

[**Barrera 1991**] J. S. Barrera, "A Fast Mach Network IPC Implementation", *Proceedings of the USENIX Mach Symposium* (1991), pages 1~12.

[**Beck et al. 1998**] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, *Second Edition*, Addison-Wesley (1998).

[**Belady 1966**] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer", *IBM Systems Journal*, Volume 5, Number 2 (1966), pages 78~101.

[**Belady et al. 1969**] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine", *Communications of the ACM*, Volume 12, Number 6 (1969), pages 349~353.

[**Ben-Ari 1990**] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall (1990).

[**Benjamin 1990**] C. D. Benjamin, "The Role of Optical Storage Technology for NASA", *Proceedings, Storage and Retrieval Systems and Applications* (1990), pages 10~17.

[**Benstein and Goodman 1980**] P. A. Bernst and N. Goodman, "Time-Stamp-Based Algorithms for Concurrency Control in Distributed Database Systems", *Proceedings of the International Conference on Very Large Databases* (1980), pages 285~300.

[**Bernstein et al. 1987**] A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, (1987).

[**Bershad and Pinkerton 1988**] B. N. Bershad and C. B. Pinkerton, "Watchdogs: Extending the UInix File System", *Proceedings of the Winter USENIX Conference* (1988).

[**Bershad et al. 1990**] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight Remote Procedure Call", *ACM Transactions on Computer Systems*, Volume 8, Number 1 (1990), pages 37~55.

[**Beveridge and Wiener 1997**] J. Beveridge and R. Wiener, *Mutlithreading Applications in Win32*, Addison-Wesley (1997).

[**Birman and Joseph 1987**] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transactions on Computer Systems*, Volume 5, Number 1 (1987).

[**Birrell and Nelson 1984**] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACm Transactions on Computer Systems*, Volume 2, Number 1 (1984), pages 39~59.

[**Black 1990**] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating

System", *Computer*, Volume 23, Number 5(1990), pages 35～43.

[**Bobrow et al. 1972**] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing for the PDP-10", *Communications of the ACM*, Volume 15, Number 3(1972).

[**Bovet and Cesati 2001**] D. P. Bovet and M. Cesati, *Understanding The Linux Kernel*, O'Reilly & Associates(2001).

[**Brain 1996**] M. Brain, *Win32 System Services, Second Edition*, Prentice Hall (1996).

[**Brereton 1986**] O. P. Brereton, "Management of Replicated Files in a UNIX Environment", *Software-Practice and Experience*, Volume 16, (1986), pages 771～780.

[**Brinch-Hansen 1970**] P. Brinch-Hansen, "The Nucleus of a Multiprogramming System", *Communications of the ACM*, Volume 13, Number 4(1970), pages 238～241 and 250.

[**Brinch-Hansen 1972**] P. Brinch-Hansen, "Structured Multiprogramming", *Communications of the ACM*, Volume 15, Number 7(1972), pages 574～578.

[**Brinch-Hansen 1973**] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall(1973).

[**Brownbridge et al. 1982**] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!", *Software-Practice and Experience*, Volume 12, Number 12(1982), pages 1147～1162

[**Burns 1978**] E. Burns, "Mutual Exclusion with Linear Waiting Using Bsinary Shared Variables", *SIGACT News*, Volume 10, Number 2(1978), pages 42～47.

[**Buyya 1999**] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*, Prentice Hall (1999).

[**Callaghan 2000**] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

[**Carr and Hennessy 1981**] W. R. Carr and J. L. Hennessy, "WSClock-A Simple and Effective Algorithm for Virtual Memory Management", *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 87～95.

[**Carvalho and Roucairol 1983**] O. S. Carvalho and G. Roucairol, "On Mutual Exclusion in Computer Networks", *Communications of the ACM*, Volume 26, Number 2(1983), pages 146～147.

[**Chang 1980**] E. Chang, "N-Philosophers: An Exercise in Distributed Control", *Computer Networks*, Volume 4, Number 2 (1980), pages 71～76

[**Chang and Mergen 1988**] A. Chang and M. F. Mergen, "801 Storage: Architecture and Programming", *ACM Transactions on Computer Systems*, Volume 6, Number 1(1988), pages 28～50.

[**Chen et al. 1994**] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Survey*, Volume 26, Number 2(1994), pages 145～185.

[**Cheriton and zwaenepoel 1983**] D. R. Cheriton and W. Z. Zwaenepoel, "The Distributed V Kernel and Its Performance for Diskless Workstations", *Proceedings of the ACM Symposium on Operating Systems Principles*(1983), pages 129～140.

[**Cheung and Loong 1995**] W. H. Cheung and A. H. S. Loong, "Exploring Issues of Operating Systems Structuring: From Microkernel to Extensible Systems", *Operating Systems Review*, Volume 29(1995),

pages 4～16

[**Chi 1982**] C. S. Chi, "Advances in Computer Mass Storage Technology", *Computer*, Volume 15, Number 5 (1982), pages 60～74.

[**Coffman and Denning 1973**] E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice Hall (1973).

[**Coffman and Kleinrock 1968**] E. G. Coffman and L. Kleinrock, "Feedback Queuing Models for Time-Shared Systems", *Communications of the ACM*, Volume 11, Number 4(1968), pages 549～576

[**Coffman et al. 1971**] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System Deadlocks", *Computing Surveys*, Volume 3, Number 2(1971), pages 67～78

[**Cohen and Jefferson 1975**] E. S. Cohen and D. Jefferson, "Protection in the Hydra Operating System", *Proceedings of the ACM Symposium on Operating Systems Principles*(1975), pages 141～160

[**Comer 1999**] D. Comer, *Internetworking with TCP/IP, Volume II, Third Edition*, Prentice Hall(1999).

[**Comer 2000**] D. Comer, *Internetworking with TCP/IP, Volume I, Third Edition*, Prentice Hall(1999).

[**Corbato and Vyssotsky 1965**] F. J. Corbato and V. A. Vyssotsky, "Introduction and Overview of the MULTICS System", *Proceedings of the AFIPS Fall Joint Computer Conference*(1965), pages 185～196.

[**Corbato et al. 1962**] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-Sharing System", *Proceedings of the AFIPS Fall Joint Computer Conference*(1962), pages 335～344

[**Courtois et al. 1971**] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'", *Communications of the ACM*, Volume 14, Number 10(1971), pages 667～668.

[**Custer 1994**] H. Custer, *Inside the Windows NT File System*, Microsoft Press (1994).

[**Davcev and Burkhard 1985**] D. Davcev and W. A. Burkhard, "Consistency and Recovery Control for Replicated Files", *Proceedings of the ACM Symposium on Operating Systems Principles*(1985), pages 87～96.

[**Davies 1983**] D. W. Davies, "Applying the RSA Digital Signature to Electronic Mail", *Computer*, Volume 16, Number 2(1983), pages 55～62.

[**deBruijn 1967**] N. G. deBruijn, "Additional Comments on a Problem in Concurrent Programming and Control", *Communications of the ACM*, Volume 10, Number 3(1967), pages 137～138.

[**DEC 1981**] DEC, *VAX Architecture Handbook*, Digital Equipment Corporation (1981)

[**Deitel 1990**] H. M. Deitel, *An Introduction to Operating Systems, Second Edition*, Addison-Wesley (1990).

[**Deitel and Kogan 1992**] H. M. Deitel and M. S. Kogan, *The Design of OS/2*, Addison-Wesley(1992).

[**Denning 1968**] P. J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, Volume 11, Number 5(1968), pages 323～333.

[**Denning 1980**] P. J. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, Volume SE-6, Number 1(1980), pages 64～84.

[**Denning 1982**] D. E. Denning, *Cryptography and Data Security*, Addison Wesley(1982).

[**Denning 1983**] D. E. Denning, "Protecting Public Keys and Signature Keys", *Computer*, Volume 16,

Number 2(1983), pages 27～35.

[**Denning 1984**] D. E. Denning, "Digital Signatures with RSA and Other Public Key Cryptosystems", *Communications of the ACM*, Volume 27, Number 4(1984), pages 388～392.

[**Dennis 1965**] J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems", *Communications of the ACM*, Volume 8, Number 4(1965), pages 589～602.

[**Dennis and Horn 1966**] J. B. Dennis and E. C. V. Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM*, Volume 9, Number 3(1966), pages 143～155.

[**Diffie and Hellman 1976**] W. Diffie and M. E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Volume 22, Number 6(1976), pages 644～654.

[**Diffie and Hellman 1979**] W. Diffie and M. E. Hellman, "Privacy and Authentication", *Proceedings of the IEEE*(1979), pages 397～427.

[**Dijkstra 19655a**] E. W. Dijkstra, "Cooperating Sequential Processes", *Technical Report, Technological University, Eindhoven, the Netherlands*(1965), pages 43～112.

[**Dijkstra 1965b**] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Volume 8, Number 9(1965), pages 569.

[**Dijkstra 1968**] E. W. Dijkstra, "The Structure of the THE Multiprogramming System", *Communications of the ACM*, Volume 11, Number 5(1968), pages 341～346.

[**Dijkstra 1971**] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes", *Acta Informatica*, Volume 1, Number 2(1971), pages 115～138.

[**DoD 1985**] *Trusted Computer System Evaluation Criteria*, Department of Defense (1985).

[**Dougan et al. 1999**] C. Dougan, P. Mackerras, and V. Yodaiken, "Optimizing the Idle Task and Other MMU Tricks", *Proceedings of the Third Symposium on Operating System Design and Implementation*(1999).

[**Douglis et al. 1991**] F. Douglis, M. F. Kaashoek, and A. S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite", *Computing Systems*, Volume 4(1991).

[**Douglis et al. 1991**] F. Douglis, M. F. Kaashoek, and A. S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite", *Computing Systems*, Volume 4, (1991).

[**Draves et al. 1991**] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems", *Proceedings of the ACM Symposium on Operating Systems Principles*(1991), pages 122～136.

[**Earhart 1986**] S. V. Earhart, editor, *UNIX Programmer's Manual*, Holt, Rinehart and Winston(1986).

[**Eastlake 1999**] D. Eastlake, "Domain Name System Security Extensions", *Network Working Group, Request for Comments*: 2535(1999).

[**Eisenberg and McGuire 1972**] M. A. Eisenberg and M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem", *Communications of the ACM*, Volume 15, Number 11 (1972), pages 999.

[**Ekanadham and Bernstein 1979**] K. Ekanadham and A. J. Bernstein, "Conditional Capabilities", *IEEE Transactions on Software Engineering*, Volume SE-5, Nember 5(1979), pages 458～464.

[**Engelschall 2000**] R. Engelschall, "Portable Multithreading: The Signal Stack Trick For User-Space Thread Creation", *Proceedings of the 2000 USENIX Annual Technical Conference* (2000).

[**Eswaran et al. 1976**] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Volume 19, Number 11 (1976), pages 624～633.

[**Eykholt et al. 1992**] J. R. Eykholt, S. R. Kleiman, S. Barton, S. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing: Multithreading the SunOS Kernel", *Proceedings of the Summer USENIX Conference* (1992), pages 11～18.

[**Farley 1998**] J. Farley, *Java Distributed Computing*, O'Reilly & Associates (1998).

[**Farrow 1986a**] R. Farrow, "security for Superusers, or How to Break the UNIX System", *UNIX World* (May 1986), pages 65～70.

[**Farrow 1986b**] R. Farrow, "Security Issues and Strategies for Users", *UNIX World* (April 1986), pages 65～71.

[**Feitelson and Rudolph 1990**] D. Feitelson and L. Rudolph, "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control", *Proceedings of the International Conference on Parallel Processing* (1990).

[**Filipski and Hanko 1986**] A. Filipski and J. Hanko, "Making UNIX Secure", *Byte* (April 1986), pages 113～128.

[**Finkel 1988**] R. A. Finkel, *Operating Systemes Vade Mecum*, *Second Edition*, Prentice Hall (1988).

[**Folk and Zoellick 1987**] M. J. Folk and B. Zoellick, *File Structures*, Addison-Wesley (1987).

[**Forrest et al. 1996**] S. Forrest, S. A. Hofmeyr, and T. A. Longstaff, "a Sense of Self for UNIX Processes", *Proceedings of the IEEE Symposium on Security and Privacy* (1996), pages 120～128.

[**Fortier 1989**] P. J. Fortier, *Handbook of LAN Technology*, McGraw-Hill (1989).

[**FreeBSD 1999**] FreeBSD, *FreeBSD Handbook*, The FreeBSD Documentation Project (1999).

[**Freedman 1983**] D. H. Freedman, "Searching for Denser Disks", *Infosystems* (1983), pages 56.

[**Fujitani 1984**] L. Fujitani, "Laser Optical Disk: The Coming Revolution in OnLine Storage", *Communications of the ACM*, Volume 27, Number 6 (1984), pages 546～554.

[**Gait 1988**] J. Gait, "The Optical File Cabinet: A Random-Access File System for Write-On Optical Disks", *Computer*, Volume 21, Number 6 (1988).

[**Ganapathy and Schimmel 1998**] N. Ganapathy and C. Schimmel, "General Purpose Operating System Support for Multiple Page Sizes", *Proceedings of the USENIX Technical Conference* (1998).

[**Garcia-Molina 1982**] H. Garcia-Molina, "Elections in Distributed Computing Systems", *IEE Transactions on Computers*, Volume C-31, Number 1 (1982).

[**Garfinkel and Spafford 1991**] S. Garfinkel and G. Spafford, *Practical UNIX Security*, O'Reilly & Associates (1991).

[**Gifford 1982**] D. K. Gifford, "Gryptographic Scaling for Information Secrecy and Authentication", *Communications of the ACM*, Volume 25, Number 4 (1982), pages 274～286.

[**Golden and Pechura 1986**] D. Golden and M. Pechura, "The Structure of Microcomputer File Systems",

Communications of the ACM, Volume 29, Number 3(1986), pages 222～230.

[**Goldman 1989**] P. Goldman, "Mac VM Revealed", *Byte*(September 1989).

[**Gong et al. 1997**] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", *Proceedings of the USENIX Symposium on Internet Technologies and Systems*(1997).

[**Gosling et al. 1996**] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley (1996).

[**Grampp and Morris 1984**] F. T. Grampp and R. H. Morris, "UNIX OperatingSystem Security", *AT&T Bell Laboratories Technical Journal*, Volume 63, (1984), pages 1649～1647.

[**Gray 1978**] J. N. Gray, "Notes on Data Base Operating Systems", in [Bayer et al. 1978](1978), pages 393～481.

[**Gray 1981**] J. N. Gray, "The Transaction Concept: Virtues and Limitations", *Proceedings of the International Conference on Very Large Databases*(1981), pages(144～154).

[**Gray 1997**] J. Gray, *Interprocess Communications in UNIX*, Prentice Hall(1997).

[**Gray et al. 1981**] J. N. Gray, P. R. McJones, and M. Blasgen, "The Recovery Manager of the System R Database Manager", *ACM Computing Survey*, Volume 13, Number 2(1981), pages 223～242.

[**Grosshans 1986**] D. Grosshans, *File Systems Design and Implementation*, Prentice Hall(1986).

[**Gupta and Franklin 1978**] R. K. Gupta and M. A. Franklin, "Working Set and Page Fault Frequencyu Replacement Algorithms: A Performance Comparison", *IEEE Transactions on Computers*, Volume C-27, Number 8(1978), pages 706～712.

[**Habermann 1969**] A. N. Habermann, "Prevention of System Deadlocks", *Communications of the ACM*, Volume 12, Number 7(1969), pages 373～377, 385.

[**Hagmann 1989**] R. Hagmann, "Comments on Workstation Operating Systems and Virtual Memory", *Proceedings of the Workshop on Workstation Operating Systems*(1989).

[**Haldar and Subramanian 1991**] S. Haldar and D. Subramanian, "Fairness in Processor Scheduling in Time Sharing Systems", *Operating Systems Review*(January 1991).

[**Halsall 1992**] F. Halsall, *Data Communications, Computer Networks and Open Systems*, Addison-Wesley (1992).

[**Han and Ghosh 1998**] K. Han and S. Ghosh, "A Comparative Analysis of Virtual Versus Physical Process-Migration Strategies for Distributed Modeling and Simulation of Mobile computing networks", *Wireless Networks*, Volume 4, Number 5(1998), pages 365～378.

[**Hansen and Atkins 1993**] S. E. Hansen and E. T. Atkins, "Automated System Monitoring and Notification With Swatch", *Proceedings of the USENIX Systems Administration Conference*(1993).

[**Harchol-Balter and Downey 1997**] M. Harchol-Balter and A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *ACM Transactions on Computer Systems*, Volume 15, Number 3(1997), pages 253～285.

[**Harish and Owens 1999**] V. C. Harish and B. Owens, "Dynamic Load Balancing DNS", *Linux Journal*, Volume 1999, Number 64(1999).

[**Harker et al. 1981**] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana, and L. G. Taft, "A Quarter Century of Disk File Innovation", *IBM Journal of Research and Development*, Volume 25, Number 5 (1981), pages 677～689.

[**Harrison et al. 1976**] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in Operating Systems", *Communications of the ACM*, Volume 19, Number 8(1976), pages 461～471.

[**Hartley 1998**] S. Hartley, *Concurrent Programming: The Java Programming Language*, Oxford University Press(1998).

[**Havender 1968**] J. W. Havender, "Avoiding Deadlock in Multitasking Systems", *IBM Systems Journal*, *Volume* 7, Number 2(1968), pages 74～84.

[**Hecht et al. 1988**] M. S. Hecht, A. Johri, R. Aditham, and T. J. Wei, "Experience Adding C2 Security Features to UNIX", *Proceedings of the Summer USENIX Conference*(1988), pages 133～146.

[**Hendricks and Hartmann 1979**] E. C. Hendricks and T. C. Hartmann, "Evolution of a Virtual Machine Subsystem", *IBM Systems Journal*, Volume 18, Number 1(1979), pages 111～142

[**Hennessy and Patterson 1996**] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers(1996).

[**Henry 1984**] G. Henry, "The Fair Share Scheduler", *AT&T Bell Laboratories Technical Journal* (1984).

[**Hoagland 1985**] A. S. Hoagland, "Information Storage Technology-A Look at the Future", *Computer*, Volume 18, Number 7(1985), pages 60～68.

[**Hoare 1972**] C. A. R. Hoare, "Towards a Theory of Parallel Programming", in[**Hoare and Perrott 1972**] (1972), pages 61～71.

[**Hoare 1974**] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Volume 17, Number 10(1974), pages 549～557.

[**Holt 1971**] R. C. Holt, "Comments on Prevention of System Deadlocks", *Communications of the ACM*, Volume 14, Number 1(1971), pages 36～38.

[**Holt 1972**] R. C. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Volume 4, Number 3(1972), pages 179～196.

[**Hong et al. 1989**] J. Hong, X. Tan, and D. Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System", *IEEE Transactions on Computers*, Volume 38, Number 12(1989), pages 1736～1744.

[**Horstmann and Cornell 1998**] C. Horstmann and G. Cornell, *Core Java* 1.2, *Volume I: Fundamentals*, *Second Edition*, Prentice-Hall (1998).

[**Howard et al. 1988**] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, Volume 6, Number 1(1988), pages 55～81.

[**Howarth et al. 1961**] D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part II: User's Description", *Computer Journal*, Volume 4, Number 3(1961), pages 226～229.

[**Hsiao et al. 1979**] D. K. Hsiao, D. S. Kerr, and S. E. Madnick, *Computer Security*, Academic Press(1979).

[**Hyman 1985**] D. Hyman, *The Columbus Chicken Statute and More Bonehead Legislation*, S. Greene Press (1985).

[**Iacobucci 1988**] E. Iacobucci, *OS/2 Programmer's Guide*. Osborne McGraw-Hill (1988).

[**IBM 1983**] *Technical Reference*. IBM Corporation(1983).

[**Iliffe and Jodeit 1962**] J. K. Iliffe and J. G. Jodeit, "A Dynamic Storage Allocation System", *Computer Journal*, Volume 5, Number 3(1962), pages 200~209.

[**Intel 1985a**] *iAPX 286 Programmer's Reference Manual*. Intel Corporation (1985).

[**Intel 1985b**] *iAPX 86/88, 186/188 User's Manual Programmer's Reference*. Intel Corporation(1985).

[**Intel 1986**] *iAPX 386 Programmer's Reference Manual*. Intel Corporation(1986).

[**Intel 1989**] *i486 Microprocessor*. Intel Corporation (1989).

[**Intel 1990**] *i486 Microprocessor Programmer's Reference Manual*. Intel Corporation(1990).

[**Intel 1993**] *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*. Intel Corporation. (1993).

[**Iseminger 2000**] D. Iseminger, *Active Directory Services for Microsoft Windows 2000. Technical Reference*, Micrososft Press(2000).

[**Jacob and Mudge 1997**] B. Jacob and T. Mudge, "Software-Managed Address Translation", *Proceedings of the Third International Symposium on High Performance Computer Architecture and Implementation*(1997).

[**Jacob and Mudge 1998a**] B. Jacob and T. Mudge, "Software-Managed Address Translation", *Proceedings of the Third International Symposium on High Performance Computer Architecture and Implementation*(1997).

[**Jacob and Mudge 1998a**] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors", *IEEE Micro Magazine*, Volume 18, (1998), pages 60~75

[**Jacob and Mudge 1998b**] B. Jacob and T. Mudge, "Virtual Memory: Issues of Implementation", *IEEE Computer Magazine*, Volume 31, (1998), pages 33~43.

[**Jensen et al. 1985**] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", *Proceedings of the IEEE Real-Time Systems Symposium*(1985), pages 112~122.

[**Jones and Lin 1996**] R. Jones and R. Lin, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons (1996).

[**Jones and Liskov 1978**] A. K. Jones and B. H. Liskov, "A Language Extension for Expressing Constraints on Data Access", *Communications of the ACM*, Volume 21, Number 5(1978), pages 358~367.

[**Jones and Schwarz 1980**] A. K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems—A Status Report", *Computing Surveys*, Volume 12, Number 2(1980), pages 121~165

[**Katz et al. 1989**] R. H. Katz, G. A. Gibson, and D. A. Patterson, "Disk System Architectures for High Performance Computing", *Proceedings of the IEEE*(1989).

[**Kay and Lauder 1988**] J. Kay and P. Lauder, "A Fair Share Scheduler", *Communications of the ACM*, Volume 31, Number 1(1988), pages 44-55.

[**Kenah et al. 1988**] L. J. Kenah, R. E. Goldenberg, and S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press(1988).

[**Kenville 1982**] R. F. Kenville, "Optical Disk Data Storage", *Computer*, Volume 15, Number 7(1982), pages 21～26.

[**Kessels 1977**] J. L. W. Kessels, "Optical Disk Data Storage", *Computer*, Volume 15, Number 7(1982), pages 21～26

[**Kessels 1977**] J. L. W. Kessels, "An Alternative to Event Queues for Synchronization in Monitors", *Communications of the ACM*, Volume 20, Number 7(1977), pages 500～503.

[**Khanna et al. 1992**] S. Khanna, M. Sebree, and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0", *Proceedings of the Winter USENIX Conference*(1992), pages 375～390.

[**Kieburtz and Silberschatz 1978**] R. B. Kieburtz and A. Silberschatz, "Capability Managers", *IEEE Transactions on Software Engineering*, Volume SE-4, Number 6(1978), pages 467～477.

[**Kieburtz and Silberschatz 1983**] R. B. Kieburtz and A. Silberschatz, "Access Right Expressions", *ACM Transactions on Programming Languages and Systems*, Volume 5, Number 1(1983), pages 78～96.

[**Kilburn et al. 1961**] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part I: Internal Organization", *Computer Journal*, Volume 4, Number 3(1961), pages 222～225.

[**Kim and Spafford 1993**] G. H. Kim and E. H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker", *Technical Report, Purdue University*(1983).

[**King 1990**] R. P. King, "Disk Arm Movement in Anticipation of Future Requests", *ACM Transactions on Computer Systems*, Volume 8, Number 3(1990), pages 214～229.

[**Kleiman et al. 1996**] S. Kleiman, D. Shah, and B. Smaalders, *Programming with Threads*, Sunsoft Press (1996).

[**Kleinrock 1975**] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications* Wiley-Interscience (1975).

[**Knapp 1987**] E. Knapp, "Deadlock Detection in Distributed Databases", *Computing Surveys*, Volume 19, Number 4(1987), pages 303～328.

[**Knuth 1966**] D. E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control", *Communications of the ACM*, Volume 9, Number 5(1966), pages 321～322.

[**Knuth 1973**] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition*, Addison-Wesley(1973).

[**Koch 1987**] P. D. L. Koch, "Disk File Allocation Based on the Buddy System", *ACM Transactions on Computer Systems*, Volume 5, Number 4(1987), pages 352～370

[**Kogan and Rawson 1988**] M. S. Kogan and F. L. Rawson, "The Design of Operating System/2", *IBM Systems Journal*, Volume 27, Number 2(1988), pages 90～104.

[**Kosaraju 1973**] S. Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets", *Operating Systems Review*, Volume 7, Number 4(1973), pages 122～126.

[**Kramer 1988**] S. M. Kramer, "Retaining SUID Programs in a Secure UNIX", *Proceedings of the Summer*

USENIX Conference(1988), pages 107～118.

[**Kurose and Ross 2001**] J. Kurose and K. Ross, *Computer Networking—A TopDown Approach Featuring the Internet*, Addison-Wesley(2001).

[**Lamport 1974**] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem", *Communications of the ACM*, Volume 17, Number 8(1974), pages 453～455.

[**Lamport 1976**] L. Lamport, "Synchjronization of Independent Processes", *Acta Informatica*, Volume 7, Number 1(1976), pages 15～34.

[**Lamport 1977**] L. Lamport, "Concurrent Reading and Writing", *Communications of the ACM*, Volume 20, Number 11(1977), pages 806～811.

[**Lamport 1978a**] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems", *Computer Networks*, Volume 2, Number 2(1978), pages 95～114.

[**Lamport 1978b**] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, Volume 21, Number 7(1978), pages 558～565.

[**Lamport 1981**] L. Lamport, "Password Authentication with Insecure Communications", *Communications of the ACM*, Volume 24, Number 11(1981), pages 770～772.

[**Lamport 1986**] L. Lamport, "The Mutual Exclusion Problem", *Communications of the ACM*, Volume 33, Number 2(1986), pages 313～348.

[**Lamport 1991**] L. Lamport, "The Mutual Exclusion Problem Has Been Solved", *Communications of the ACM*, Volume 34, Number 1(1991), pages 110.

[**Lamport et al. 1982**] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 3(1982), pages 382～401.

[**Lampson 1968**] B. W. Lampson, "A Scheduling Philosophy for Multiprocessing Systems", *Communications of the ACM*, Volume 11, Number 5(1968), pages 347～360.

[**Lampson 1969**] B. W. Lampson, "Dynamic Protection Structures", *Proceedings of the AFIPS Fall Joint Computer Conference*(1969), pages 27～38.

[**Lampson 1971**] B. W. Lampson, "Protection", *Proceedings of the Fifth Annual Princeton Conference on Information Systems Science* (1971), pages 437～443.

[**Lampson 1973**] B. W. Lampson, "A Note on the Confinement Problem", *Communications of the ACM*, Volume 10, Number 16(1973), pages 613～615.

[**Lampson and Sturgis 1976**] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System", *Technical Report*, *Xerox Research Center* (1976).

[**Landwehr 1981**] C. E. Landwehr, "Formal Models of Computer Security", *Computing Surveys*, Volume 13, Number 3(1981), pages 247～278.

[**Lann 1977**] G. L. Lann, "Distributed Systems—Toward a Formal Approach", *Proceedings of the IFIP Congress*(1977), pages 155～160

[**Larson and Kajla 1984**] P. Larson and A. Kajla, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access", *Communications of the ACM*, Volume 27, Number 7(1984), pages 670～677.

[**Lazowska et al. 1984**] E. D. Lazowksa, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance*, Prentice Hall(1984).

[**Lea 2000**] D. Lea, *Concurrent Programming in Java*, *Second Edition*, Addison Wesley(2000).

[**Leffler et al. 1989**] S. J. Leffler, M. K. McKusick, M. J. Kerels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley (1989).

[**Lehmann 1987**] F. Lehmann, "Computer Break-Ins", *Communications of the ACM*, Volume 30, Number 7 (1987), pages 584～585.

[**Lempel 1979**] A. Lempel, "Cryptology in Transition", *Computing Suroveys*, Volume 11, Number 4 (1979), pages 286～303.

[**Lett and Konigsford 1968**] A. L. Lett and W. L. Konigsford, "TSS/360: A Time-Shared Operating System", *Proceedings of the AFIPS Fall Joint Computer Conference*(1968), pages 15～28.

[**Letwin 1988**] G. Letwin, *Inside* OS/2, Microsoft Press(1988).

[**Leutenegger and Vernon 1990**] S. Leutenegger and M. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", *Proceedings of the Conference on Measurement and Modeling of Computer Systems*(1990).

[**Levin et al. 1975**] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf, "Policy/Mechanism Separation in Hydra", *Proceedings of the ACM Symposium on Operating Systems Principles*(1975), pages 132～140.

[**Levy and Lipman 1982**] H. M. Levy and P. H. Lipman, "Virtual Memory Management in the VAX/VMS Operating System", *Computer*, Volume 15, Number 3(1982), pages 35～41.

[**Lewis and Berg 1988**] B. Lewis and D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press(1988).

[**Lichtenberger and Pirtle 1965**] W. W. Lichtenberger and M. W. Pirtle, "A Facility for Experimentation in Man-Machine Interaction", *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 589～598.

[**Lindholm and Yellin 1998**] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley (1998).

[**Ling et al. 2000**] Y. Ling, T. Mullen, and X. Lin, "Analysis of Optimal Thread Pool Size", *Operating System Review*, Volume 34, Number 2(2000).

[**Lipner 1975**] S. Lipner, "A Comment on the Confinement Problem", *Operating System Review*, Volume 9, Number 5(1975), pages 192～196.

[**Lipton 1974**] R. Lipton, "On Synchronization Primitive Systems", *PhD. Thesis*, *Carnegie-Mellon University*(1974).

[**Liskov 1972**] B. H. Liskov, "The Design of the Venus Operating System", *Communications of the ACM*, Volume 15, Number 3(1972), pages 144～149.

[**Litzkow et al. 1988**] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations", *Proceedings of the IEEE International Conference on Distributed Computing Systems* (1988), pages 104～111.

[**Liu and Layland 1973**] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Communications of the ACM*, Volume 20, Number 1(1973), pages 46～61.

[**Lobel 1986**] J. Lobel, *Foiling the System Breakers: Computer Security and Access Control*, McGraw-Hill (1986).

[**Loucks and Sauer 1987**] L. K. Loucks and C. H. Sauer, "Advanced Interactive Executive (AIX) Operating System Overview", *IBM Systems Journal*, Volume 26, Number 4(1987), pages 326～345.

[**MacKinnon 1979**] R. A. Mackinnon, "The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware and Other Virtual Machines", *IBM Systems Journal*, Volume 18, Number 1(1979), pages 18～46.

[**Maekawa 1985**] M. Maekawa, "A Square Root Algorithm for Mutual Exclusion in Decentralized Systems", *ACM Transactions on Computer Systems*, Volume 3, Number 2(1985), pages 145～159.

[**Maher et al. 1994**] C. Maher, J. S. Goldick, C. Kerby, and B. Zumach, "The Integration of Distributed File Systems and Mass Storage Systems", *Proceedings of the IEEE Symposium on Mass Storage Systems* (1994), pages 27～31.

[**Marsh et al. 1991**] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-Class User-Level Threads", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), pages 110～121.

[**Massalin and Pu 1989**] H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel", *Proceedings of The ACM Symposium on Operating Systems Principles* (1989), pages 191～200.

[**Mattson et al. 1970**] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies", *IBM Systems Journal*, Volume 9, Number 2(1970), pages 78～117.

[**Mauro and McDougall 2001**] J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall(2001).

[**McGraw and Andrews 1979**] J. R. McGraw and G. R. Andrews, "Access Control in Parallel Programs", *IEEE Transactions on Software Engineering*, Volume SE-5, Number 1(1979), pages 1～9.

[**McKeag and Wilson 1976**] R. M. McKeag and R. Wilson, *Studies in Operating Systems*, Academic Press (1976).

[**McKeon 1985**] B. McKeon, "An Algorithm for Disk Caching with Limited Memory", *Byte*, Volume 10, Number 9 (1985), pages 129～138.

[**McKusick et al. 1984**] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Volume 2, Number 3(1984), pages 181～197.

[**McKusick et al. 1996**] M. K. McKusick, K. Bostic, and M. J. Karels, *The Design and Implementation of the 4.4 BSD UNIX Operating System*, John Wiley and Sons(1996).

[**McVoy and Kleiman 1991**] L. W. McVoy and S. R. Kleiman, "Extent-like Performance from a UNIX File System", *Proceedings of the Winter USENIX Conference* (1991), pages 33～44.

[**Mealy et al. 1966**] G. H. Mealy, B. I. Witt, and W. A. Clark, "The Functional Structure of OS/360", *IBM Systems Journal*, Volume 5, Number 1(1966).

[**Menasce and Muntz 1979**] D. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering*, Volume SE-5, Number 3(1979), pages 195~202.

[**Meyer and Downing 1997**] J. Meyer and T. Downing, *Java Virtual Machine*, O'Reilly and Associates (1997).

[**Meyer and Seawright 1970**] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System", *IBM Systems Journal*, Volume 9, Number 3(1970), pages 199~218.

[**Microsoft 1986**] *Microsoft MS-DOS User's Reference and Microsoft MS-DOS Programmer's Reference*. Microsoft Press(1986).

[**Microsoft 1989**] *Microsoft Operating System/2 Programmer's Reference*, 3 *Volumes*. Microsoft Press (1989).

[**Microsoft 1991**] *Microsoft MS-DOS User's Guide and Reference*. Microsoft Press (1991).

[**Microsoft 1996**] *Microsoft Windows NT Workstation Resource Kit*. Microsoft Press(1996).

[**Microsoft 2000a**] *Microsoft Developer Network Development Library*. Microsoft Press(2000).

[**Microsoft 2000b**] *Microsoft Windows* 2000 *Server Resource Kit*. Microsoft Press (2000).

[**Milenkovic 1987**] M. Milenkovic, *Operating Systems: Concepts and Design*, McGraw-Hill(1987).

[**Miller and Katz 1993**] E. L. Miller and R. H. Katz, "An Analysis of File Migration in a UNIX Supercomputing Environment". *Proceedings of the Winter USENIX Conference* (1993), pages 421~434.

[**Milo et al. 2000**] D. Milo, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration", *ACM Computing Survey*, Volume 32, Number 3(2000), pages 241~299.

[**Mockapetris 1987**] P. Mockapetris, "Domain Names-Concepts and Facilities", *Network Working Group*, *Request for Comments*: 1034(1987).

[**Mohan and Lindsay 1983**] C. Mohan and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", *Proceedings of the ACM Symposium on Principles of Database Systems*(1983).

[**Morris 1973**] J. H. Morris, "Protection in Programming Languages", *Communications of the ACM*, Volume 16, Number 1(1973), pages 15~21.

[**Morris and Thompson 1979**] R. Morris and K. Thompson, "Password Security: A Case History", *Communication of the ACM*, Volume 22, Number 11(1979), pages 594~597.

[**Morris et al. 1986**] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, Volume 29, Number 3(1986), pages 184~201.

[**Morshedian 1986**] D. Morshedian, "How to Fight Password Pirates", *Computer*, Volume 19, Number 1 (1986).

[**Motorola 1989a**] *MC68000 Family Reference*, *Second Edition*, Prentice Hall(1989).

[**Motorola 1989b**] *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, *Second Edition*, Prentice Hall(1989).

[**Motorola 1993**] *PowerPC 601 RISC Microprocessor User's Manual*. Motorola Inc. (1993).

[**Mullender 1993**] S. Mullender, editor, *Distributed Systems*, *Second Edition*, ACM Press(1993).

[**Mullender et al. 1990**] S. J. Mullender, G. V. Rossumand, A. S. Tanenbaum, R. van Renesse, and H. V. Staveren, "Amoeba: A Distributed Operating System for the 1990s", *Computer*, Volume 23, Number 5 (1990), pages 44～53.

[**Murray 1998**] J. Murray, *Inside Microsoft Windows CE*, Microsoft Press(1998).

[**Mutka and Livny 1987**] M. W. Mutka and M. Livny, "Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network", *Proceedings of the IEEE International Conference on Distributed Computing Systems*(1987).

[**Needham and Walker 1977**] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and Its Protection System", *Proceedings of the Sixth Symposium on Operating System Principles*(1977), pages 1～10.

[**Nelson et al. 1988**] M. Nelson, B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, Volume 6, Number 1(1988), pages 134～154.

[**Nichols 1987**] D. A. Nichols, "Using Idle Workstations in a Shared Computing Environment", *Proceedings of the ACM Symposium on Operating Systems Principles*(1987), pages 5～12.

[**Niemeyer and Peck 1997**] P. Niemeyer and J. Peck, *Exploring Java*, *Second Edition*, O'Reilly & Associates(1997).

[**Norton 1986**] P. Norton, *Inside the IBM PC*, Brady Books(1986).

[**Norton and Wilton 1988**] P. Norton and R. Wilton, *The New Peter Norton Programmer's Guide to the IBM PC&PS/2*, Microsoft Press(1988).

[**Nutt 1999**] G. Nutt, *Operating Systems: A Modern Perpective*, *Second Edition*, Addison-Wesley (1999).

[**Oaks and Wong 1999**] S. Oaks and H. Wong, *Java Threads*, *Second Edition*, O'Reilly & Associates (1999).

[**Obermarck 1982**] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Volume 7, Number 2(1982), pages 187～208.

[**O'Leary and Kitts 1985**] B. T. O'Leary and D. L. Kitts, "Optical Device for a Mass Storage System", *Computer*, Volume 18, Number 7(1985).

[**Olsen and Kenley 1989**] R. P. Olsen and G. Kenley, "Virtual Optical Disks Solve the On-Line Storage Crunch", *Computer Design*, Volume 28, Number 1(1989), pages 93～96.

[**Organick 1972**] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).

[**Ousterhout et al. 1985**] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1985), pages 15～24.

[**Ousterhout et al. 1988**] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System", *Computer*, Volume 21, Number 2(1988), pages 23～36.

[**Parmelee et al. 1972**] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. Hatfield, "virtual Storage and Virtual Machine Concepts", *IMB Systems Journal*, Volume 11, Number 2(1972), pages 99～130.

[**Parnas 1975**] D. L. Parnas, "On a Solution to the Cigarette Smokers' Problem Without Conditional Statements", *Communications of the ACM*, Volume 18, Number 3(1975), pages 181～183.

[**Patil 1971**] S. Patil, "Limitations and Capabilities of Dijstra's Semaphore Primitives for Corrdination Among Processes", *Technical Report*, *MIT*(1971).

[**Patterson et al. 1988**] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1988).

[**Peacock 1992**] J. K. Peacock, "File System Multithreading in System V Release 4 MP", *Proceedings of the Summer USENIX Conference* (1992), pages 19～29.

[**Pease et al. 1980**] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Communications of the ACM*, Volume 27, Number 2(1980), pages 228～234.

[**Pechura and Schoeffler 1983**] M. A. Pechura and J. D. Schoeffler, "Estimating File Access Time of Floppy Disks", *Communications of the ACM*, Volume 26, Number 10(1983), pages 754～763

[**Peterson 1981**] G. L. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, *Volume* 12, *Number* 3(1981).

[**Pfleeger 1989**] C. Pfleeger, *Security in Computing*, Prentice Hall(1989).

[**Pham and Garg 1996**] T. Pham and P. Garg, *Multithreaded Programming with Windows* NT, Prentice Hall(1996).

[**Popek 1974**] G. J. Popek, "Protection Structures", *Computer*, Volume 7, Number 6(1974), pages 22～33.

[**Popek and Walker 1985**] G. Popek and B. Walker, editors, *The LOCUS Distributed System Architecture*, MIT Press(1985).

[**Prieve and Fabry 1976**] B. G. Prieve and R. S. Fabry, "VMIN-An Optimal Variable Space Page-Replacement Algorithm", *Communication of the ACM*, Volume 19, Number 5(1976), pages 295～297.

[**Psaltis and Mok 1995**] D. Psaltis and F. Mok, "Holographic Memories", *Scientific American*, Volume 273, Number 5(1995), pages 70～76.

[**Purdin et al. 1987**] T. D. M. Purdin, R. D. Schlichting, and G. R. Andrews, "A File Replication Facility for Berkeley UNIX", *Software--Practice and Experience*, Volume 17, (1987), pages 923～940.

[**Quarterman and Hoskins 1986**] J. S. Quarterman and H. C. Hoskins, "Notable Computer Networks", *Communications of the ACM*, Volume 29, Number 10(1986), pages 932～971.

[**Quinlan 1991**] S. Quinlan, "A Cached WORM", *Software—Practice and Experience*, Volume 21, Number 12(1991), pages 1289～1299.

[**Rago 1993**] S. Rago, *UNIX System V Network Programming*, Addison-Wesley(1993).

[**Rashid 1986**] R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society, Fall Joint Commputer Conference* (1986).

[**Rashid and Robertson 1981**] R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the ACM Symposium on Operating System Principles* (1981).

[**Raynal 1986**] M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press(1986).

[**Raynal 1991**] M. Raynal, "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms", *Operating Systems Review*, Volume 25, Number 1(1991), pages 47～50.

[**Redell and Fabry 1974**] D. D. Redell and R. S. Fabry, "Selective Revocation of Capabilities", *Proceedings of the IRIA International Workshop on Protection in Operating Systems*(1974), pages 197～210.

[**Reed 1983**] D. P. Reed, "Implementing Atomic Actions on Decentralized Data", *ACM Transaction on Computer Systems*, Volume 1, Number 1(1983), pages 3～23.

[**Reed and Kanodia 1979**] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequences", *Communications of the ACM*, Volume 22, Number 2(1979), pages 115～123.

[**Reid 1987**] B. Reid, "Reflections on Some Recent Widespread Computer Break-Ins", *Communications of the ACM*, Volume 30, Number 2(1987), pages 103～105.

[**Rhodes and McKeehan 1999**] N. Rhodes and J. McKeehan, *Understanding The Linux Kernel*, O'Reilly & Associates (1999).

[**Ricart and Agrawala 1981**] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networkk, "*Communication of the ACM*, Volume 24, Number 1(1981), pages 9～17

[**Richards 1990**] A. E. Richards, "A File System Approach for Integrating Removable Media Devices and Jukeboxes", *Optical Information Systems*, Volume 10, Number 5(1990), pages 270～274.

[**Richter 1997**] J. Richter, *Advanced windows*, Microsoft Press(1997).

[**Ritchin and Thompson 1974**] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *Communication of the ACM*, Volume 17, Number 7(1974), pages 365～375.

[**Rivest et al. 1978**] R. L. Rivest, A. Shamir, and L. Adleman, "On Digital Signatures and Public Key Cryptosystems", *Communications of the ACM*, Volume 21, Number 2(1978), pages 120～126.

[**Rosenkrantz et al. 1978**] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "System Level Concurrency Control for Distributed Database Systems", *ACM Transactions on Database Systems*, Volume 3, Number 2(1978), pages 178～198.

[**Ruemmler and Wilkes 1991**] C. Ruemmler and J. Wilkes, "Disk Shuffling", *Technical Report*, *Hewlett-Packard Laboratories*(1991).

[**Ruemmler and Wilkes 1993**] C. Ruemmler and J. Wilkes, "Unix Disk Access Patterns", *Proceedings of the Winter USENIOX Conference*(1993), pages 405～420.

[**Ruemmler and Wilkes 1994**] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling", *Computer*, Volume 27, Number 3(1994), pages 17～29.

[**Ruschizka and Fabry 1977**] M. Ruschizka and R. S. Fabry, "A Unifying Approach to Scheduling", *Communications of the ACM*, Volume 20, Number 7(1977), pages 469～477.

[**Rushby 1981**] J. M. Rushby, "Design and Verification of Secure Systems", *Proceedings of the* ACM *Symposium on Operating Systems Priniciples* (1981), pages 12～21.

[**Rushby and Randell 1983**] J. Rushby and B. Randell, "A Distributed Secure System", *Computer*, Volume 16, Number 27(1983), pages 55. ~67

[**Russell and G. T. Gangemi 1991**] D. Russell and S. G. T. Gangemi, *Computer Security Basics*, O'Reilly & Associates(1991).

[**Saltzer and Schroeder 1975**] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems", *Proceedings of the IEEE*(1975), pages 1278~1308.

[**Sandberg 1987**] R. Sandberg, *The Sun Network File System: Design, Implementation and Experience*, Sun Microsystems (1987).

[**Sandberg et al. 1985**] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem", *Proceedings of the Summer USENIX Conference* (1985), pages 119~130.

[**Sargent and Shoemaker 1995**] M. Sargent and R. Shoemaker, *The Personal Computer from the Inside Out, Thiord Edition*, Addison-Wesley (1995).

[**Sarisky 1983**] L. Sarisky, "Will Removable Hard Disks Replace the Floppy?", *Byte*(1983), pages 110~117.

[**Satyanarayanan 1989**] M. Satyanarayanan, "Integrating Security in a Large Distributed System", *ACM Transactions on Computer Systems*, Volume 7 (1989), pages 247~280.

[**Satyanarayanan 1990**] M. Satyanarayanan, "Scalable, Secure and Highly Available Distributed File Access", *Computer*, Volume 23, Number 5(1990), pages 9~21

[**Sauer and Chandy 1981**] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling*, Prentice Hall(1981).

[**Schell 1983**] R. R. Schell, "A Security Kernel for a Multiprocessor Microcomputer", *Computer*(1983), pages 47~53.

[**Schindler and Gregory 1999**] J. Schindler and G. Gregory, "Automated Disk Drive Characterization", *Technical Report, Carnegie-Mellon University*(1999).

[**Schlichting and Schneider 1982**] R. D. Schlichting and F. B. Schneider, "Understanding and Using Asynchronous Message Passing Primitives", *Proceedings of the Symposium on Principles of Distributed Computing*(1982), pages 141~147.

[**Schneider 1982**] F. B. Schneider, "Synchronization in Distributed Programs", *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 2(1982), pages 125~148.

[**Schrage 1967**] L. E. Schrage, "The Queue M/G/I with Feedback to Lower Priority Queues", *Management Science*, Volume 13(1967), pages 466~474.

[**Schroeder et al. 1985**] M. D. Schroeder, D. K. Gifford, and R. M. Needham, "A Caching File System for a Programmer's Workstation", *Proceedings of the ACM Symposium on Operating Systems Principles* (1985), pages 25~32.

[**Schultz 1988**] B. Schultz, "VM: The Crossroads of Operating Systems", *Datamation*, Volume 34, Number 14(1988), pages 79~84.

[**Schwartz and Weissman 1967**] J. I. Schwartz and C. Weissman, "The SDC Time-Sharing System

Revisited", *Proceedings of the ACM National Meeting* (1967), pages 263～271.

[**Schwartz et al. 1964**] J. I. Schwartz, E. G. Coffman, and C. Weissman, "A General Purpose Time-Sharing System", *Proceedings of the AFIPS Spring Joint Computer Conference* (1964), pages 397～411.

[**Seely 1989**] D. Seely, "Password Cracking: A Game of Wits", *Communications of the ACM*, Volume 32, Number 6(1989), pages 700～704.

[**Seltzer et al. 1990**] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited", *Proceedings of the Winter USENIX Conference* (1990), pages 313～323.

[**Shrivastava and Panzieri 1982**] S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism", *IEEE Transaction on Computers*, Volume C-31, Number 7(1982), pages 692～697.

[**Silberschatz et al. 2001**] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts, Fourth Edition*, McGraw Hill(2001).

[**Silverman 1983**] J. M. Silverman, "Reflections on the Verification of the Security of an Operating System Kernel", *Proceedings of the ACM Symposium on Operating Systems Principles* (1983), pages 143～154.

[**Silvers 2000**] C. Silvers, "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD", *USENIX Annual Technical Conference -FREENIX Track* (2000).

[**Simmons 1979**] G. J. Simmons, "Symmetric and Asymmetric Encryption", *Computing Surveys*, Volume 11, Number 4(1979), pages 304～330.

[**Sincerbox 1994**] G. T. Sincerbox, editor, *Selected Papers on Holographic Storage Number MS 95, SPIE Milestone Series*, Optical Engineering Press(1994).

[**Singhal 1989**] M. Singhal, "Deadlock Detection in Distributed Systems", *Computer*, Volume 22, Number 11(1989), pages 37～48.

[**Smith 1982**] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Volume 14, Number 3(1982), pages 473～530.

[**Smith 1985**] A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems*, Volume 3, Number 3(1985), pages 161～203.

[**Solomon 1998**] D. A. Solomon, *Inside Windows NT, Second Edition*, Microsoft Press(1998).

[**Solomon and Russinovich 2000**] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000, Third Edition*, Microsoft Press(2000).

[**Spafford 1989**] E. H. Spafford, "The Internet Worm: Crisis and Aftermath", *Communications of the ACM*, Volume 32, Number 6(1989), pages 678～687.

[**Spector and Schwarz 1983**] A. Z. Spector and P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", *ACM SIGOPS Operating Systems Review*, Volume 17, Number 2(1983), pages 18～35.

[**Stallings 2000a**] W. Stallings, *Local and Metropolitan Area Networks*, Prentice Hall(2000).

[**Stallings 2000b**] W. Stallings, *Operating Systems, Fourth Edition*, Prentice Hall (2000).

[**Stankovic 1982**] J. S. Stankovic, "Software Communication Mechanisms: Procedure Calls Versus

Messages", *Computer*, Volume 15, Number 4(1982).

[**Staunstrup 1982**] J. Staunstrup, "Message Passing Communication Versus Procedure Call Communication", *Software—Practice and Experience*, Volume 12, Number 3(1982), pages 223~234.

[**Stevens 1992**] R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley(1992).

[**Stevens 1994**] R. Stevens, *TCP/IP Illustrated Volume 1: The Protocols*, Addison-Wesley(1994).

[**Stevens 1995**] R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley(1995).

[**Stevens 1997**] W. R. Stevens, *UNIX Network Programming—Volume I*, Prentice Hall(1997).

[**Stevens 1998**] W. R. Stevens, *UNIX Network Programming-Volume II*, Prentice Hall(1998).

[**Strachey 1959**] C. Strachey, "Time Sharing in Large Fast Computers", *Proceedings of the International Conference on Information Processing*(1959), pages 336~341.

[**Su 1982**] Z. Su, "A Distributed System for Internet Name Service", *Network Working Group, Request for Comments*: 830(1982).

[**Sun 1990**] *Network Programming Guide*, Sun Microsystems(1990).

[**Sun 1995**] *Solaris Multithreaded Programming Guide*. Sunsoft Press(1995).

[**Svobodova 1976**] L. Svobodova, *Computer Performance Measurement and Evaluation*, Elsevier North Holland(1976).

[**Svobodova 1984**] L. Svobodova, "File Servers for Network-Based Distributed Systems", *ACM Computing Survey*, Volume 16, Number 4(1984), pages 353~398.

[**Talluri et al. 1995**] M. Talluri, M. D. Hill, and Y. A. Khalidi, "A New Page Table for 64-bit Address Spaces", *Proceedings of the ACM Symposium on Operating Systems Principles* (1995).

[**Tanenbaum 1990**] A. S. Tanenbaum, *Structured Computer Organization, Third Edition*, Prentice Hall (1990).

[**Tanenbaum 1996**] A. S. Tanenbaum, *Computer Networks, Third Edition*, Prentice Hall(1996).

[**Tanenbaum 2001**] A. S. Tanenbaum, *Modern Operating Systgems*, Prentice Hall(2001).

[**Tanenbaum and Woodhull 1997**] A. S. Tanenbaum and A. S. Woodhull, *Operating System Design and Implementation, Second Edition*, Prentice Hall(1997).

[**Tanenbaum et al. 1990**] A. S. Tanenbaum, R. van Renesse, H. V. Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. V. Rossum, "Experiences with the Amoeba Distributed Operating System", *Communications of the ACM*, Volume 33, Number 12(1990), pages 46~63.

[**Tate 2000**] S. Tate, *Windows 2000 Essential Reference*, New Riders (2000).

[**Tay and Ananda 1990**] B. H. Tay and A. L. Ananda, "A Survey of Remote Procedure Calls", *Operating Systems Review*, Volume 24, Number 3(1990), pages 68~79.

[**Teorey and Pinkerton 1972**] T. J. Teorey and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies", *Communications of the ACM*, Volume 15, Number 3(1972), pages 177~184.

[**Tevanian et al. 1987a**] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the Summer USENIX Conference*(1987).

[**Tevanian et al. 1987b**] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W.

Bolosky, and R. Sanzi, "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach", *Technical Report, CarnegieMellon University* (1987).

[**Tevanian et al. 1989**] A. Tevanian, Jr., and B. Smith, "Mach: The Model for Future Unix", Byte (1989).

[**Toigo 2000**] J. Toigo, "Avoiding a Data Crunch", *Scientific American*, Volume 282, Number 5 (2000), pages 58~74.

[**Traiger et al. 1982**] I. L. Traiger, J. N. Gray, C. A. Galtieri, and B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems", *ACM Transactions on Database Systems*, Volume 7, Number 3 (1982), pages 323~342.

[**Tucker and Gupta 1989**] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proceedings of the ACM Symposium on Operating Systems Principles* (1989).

[**Vahalia 1996**] U. Vahalia, Unix Internals: The New Frontiers, Prentice Hall (1996).

[**Vee and Hsu 2000**] V. Vee and W. Hsu, "Locality-Preserving Load-Balancing Mechanisms for Synchronous Simulations on Shared-Memory Multiprocessors", *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation* (2000), pages 131~138.

[**Venners 1998**] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill (1998).

[**Vuillemin 1978**] A. Vuillemin, "A Data Structure for Manipulating Priority Queues", *Communications of the ACM*, Volume 21, Number 4 (1978), pages 309~315.

[**Wah 1984**] B. W. Wah, "File Placement on Distributed Computer Systems", *Computer*, Volume 17, Number 1 (1984), pages 23~32.

[**Waldo 1988**] J. Waldo, "OO Systems", *IEEE Concurrency*, Volume 16, Number 3 (1988).

[**Wallach et al. 1987**] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten, "Extensible Security Architectures for Java", *Proceedings of the ACM Symposium on Operating Systems Principles* (1987).

[**Williams 2001**] R. Williams, *Computer Systems Architecture-A Networking Approach*, Addison-Wesley (2001).

[**Wood and Kochan 1985**] P. Wood and S. Kochan, *UNIX System Security*, Hayden (1985).

[**Woodside 1986**] C. Woodside, "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers", *IEEE Transactions on Software Engineering*, Volume SE 12, Number 10 (1986), pages 1041~1048.

[**Worthington et al. 1994**] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling Algorithms for Modern Disk Drives", *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (1994), pages 241~251

[**Worthington et al. 1995**] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters", *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (1995), pages 146~156.

[**Wulf 1969**] W. A. Wulf, "Performance Monitors for Multiprogramming Systems", *Proceedings of the ACM Symposium on Operating Systems Principles* (1969), pages 175~181.

[**Wulf et al. 1981**] W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer*

System, *McGraw-Hill* (1981).

[**Yeong et al. 1995**] W. Yeong, T. Howes, and S. Kille, "Lightweight Directory Access Protocol", *Network Working Group*, *Request for Comments*: 1777 (1995).

[**Zabatta and Young 1998**] F. Zabatta and K. Young, "A Thread Performance Comparison: Windows NT and Solaris on a Symmetric Multiprocessor", *Proceedings of the 2nd USENIX Windows NT Symposium* (1998).

[**Zahorjan and McCann 1990**] J. Zahorjan and C. McCann, "Processor Scheduling in Shared-Memory Multiprocessors", *Proceedings of the Conference on Measurement and Modeling of Computer Systems* (1990).

[**Zhao 1989**] W. Zhao, editor, *Special Issue on Real-Time Operating Systems*, ACM (1989).

原版相关内容引用表

Figure 3.9: From Jaccobucci, *OS/2 Programmer's Guide*, © 1988, McGraw-Hill, Inc., New York, New York. Figure 1.7, p. 20. Reprinted with Permission of the publisher.

Figure 6.8: From Khanna/Sebree/Zolnowsky, "Realtime Scheduling in SunOS 5.0", Proceedings of Winter USENIX, January 1992, San Francisco, California. Derived with Permission of the authors.

Figure 6.10 adapted with permission from Sun Microsystems, Inc.

Figure 9.21: From 80386 *Programmer's Reference Manual*, Figure 5-12, p. 5-12. Reprinted by permission of Intel Corporation, Copyright/Intel Corporation 1986.

Figure 10.16: From *IBM Systems Journal*, Vol. 10, No. 3, © 1971, International Business Machines Corporation. Reprinted by permission of IBM Corporation.

Figure 12.9: From Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, © 1989 by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 7.6, p. 196. Reprinted with permission of the publisher.

Figure 13.4: From *Pentium Processor User's Manual: Architecture and Programming Manual*, Volume 3, Copyright 1993. Reprinted by permission of Intel Corporation.

Figures 15.5, 15.6, and 15.8: From Halsall, *Data Communications, Computer Networks, and Open Systems, Third Edition*, © 1992, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 1.9, p. 14, Figure 1.10, p. 15, and Figure 1.11, p. 18. Reprinted with permission of the publisher.

Sections of chapter 7 and 17 from Silberschatz/Korth, *Database System Concepts, Third Edition*, Copyright 1997, McGraw-Hill, Inc., New York, New York. Section 13.5, p. 451～454, 14.1.1, p. 471-742, 14.1.3, p. 476-479, 14.2, p. 482-485, 15.2.1, p. 512-513, 15.4, p. 517-518, 15.4.3, p. 523-524, 18.7, p. 613-617, 18.8, p. 617-622. Reprinted with permission of the publisher.

Figure A.1: From Quarterman/Wilhelm, *UNIX, POSIX and Open Systems: The Open Standards Puzzle*, © 1993, by Addison-Wesley Publishing Co., Inc. Reading, Massachusetts. Figure 2.1, p. 31. Reprinted with permission of the publisher.

Timeline information for the back end papers was assembled: From a variety of sources, which include "The History of Electronic Computing", compiled and edited by MarcRettig, Association for Computing Machinery, Inc. (ACM), New York, New York, and Shedroff/Hutto/Fromm, Understanding Computers, © 1992, Vivid Publishing, distributed by SYBEX, San Francisco, California.

英汉对照表

B

C

cylinders 柱面
term description 术语描述

D

daemon process 守护进程
protection mechanism use 保护机制应用
term description 术语描述
daisy chain 链环
term description 术语描述
dangling pointers 悬挂指针
as shared file issue 作为共享文件问题
data 数据
See Also file(s); file system(s); I/O(input/output)
access 访问,存取
compression, in Windows 2000 file system 压缩,Windows 2000 文件系统(的)
consistency, as database system concern 一致性,作为数据库系统关注点
data-link layer, ISO network model 数据链路层,ISO 网络模型
migration, in distributed operating systems 转换,分布式操作系统(的)
nonreplicated, distributed system concurrency control 不可复制的,分布式系统并发控制
section 区
sharing 共享
striping, as parallelism technique in RAID systems 分散读写模式,作为 RAID 系统并行技术
thread-specific 线程独享
term description 术语描述
transfer mode, as device design dimension 传输模式,作为设备设计(考虑)维度
database(s) 数据库
raw disk use 生磁盘应用
techniques, operating system use of 技术,(的)操作系统应用
datagram(s) 数据报
as fixed-length communication network message 作为固定长度通信网络消息
protocol, computation migration use 协议,计算移动应用
TCP transmittal TCP 传输
UDP transmittal UDP 传输
DCE(distributed computing environment) 分布式计算环境
Transarc DFS part of (的)Transarc DFS 部分
DHCP(dynamic host-configuration protocol) 动态主机配置协议
with Windows XP
deadlock(s) 死锁
See Also process(es); synchronization
avoidance 避免
in fully distributed mutual exclusion algorithm 完全分布式互斥算法(的)
monitors use for (的)监控程序应用
term description 术语描述
characterization of (的)特征
detection 检测
in distributed system concurrency control 分布式系统并发控制(的)
handling, in distributed systems 处理,分布式系统(的)
methods for handling 处理方法
necessary conditions for (的)必要条件
prevention 预防
recovery 恢复
starvation and 饥饿(与)
term description 术语描述
Debian distribution Debian 版本
Linux distribution package Linux 版本包
DEC
See Digital Equipment Corporation
DEC VMS
See VMS operating system
deferred 延期

E

F

as remote file sharing mechanism 作为远程文件共享机制
remote file transfer use 远程文件传输应用
WWW improvement on (的)万维网改进
fully-connected network 完全连接网络
fully-distributed approach 完全分布式方法
to deadlock detection, in distributed systems 死锁检测(的),分布式系统(的)
to mutual exclusion, in a distributed systems 互斥(的),分布式系统(的)

G

Gantt chart Gantt 图
scheduling algorithm results display 调度算法结果显示
garbage collection 垃圾收集
See Also memory, management
file deletion use 文件删除应用
JVM use JVM 应用
gateway 网关
communication network use 通信网络应用
GDT(global descriptor table) 全局描述符表
term description 术语描述
general graph directories 通用图目录
Gigabit Ethernet 吉级以太网
storage area network use 存储区域网络应用
GNU project GNU 项目
General Public License, Linux use 通用公共许可,Linux 应用
Linux use of components from (的)组件的 Linux 应用
GPL (General Public License) 通用公共许可
Linux use Linux 应用
graceful degradation 优雅降阶
term description 术语描述
graph(s) 图
acyclic-graph directories 非循环图目录
general graph directories 通用图目录
system resource allocation 系统资源分配
wait-for, deadlock detection use 等待,死锁检测应用
grappling hook 挂钩
worm use 蠕虫应用
group 组
as access control classification 作为访问控制分类
indentifiers, file sharing management use 标识符,文件共享管理应用
grouping 分组
as free space list variation 作为空闲空间列表变化
growing phase 增长阶段
of two-phase locking protocol 二阶段加锁协议(的)

H

HAL (hardware-abstraction layer) 硬件抽象层
in Windows 2000
handheld systems 手持系统
handles 句柄
file 文件
in Windows 2000 object manager Windows 2000 对象管理器
handprints 手印
in security mechanisms 安全机制(的)
hands-on computer systems 手持计算机系统
term description 术语描述
handshaking 握手
DMA and device controller DMA 与设备控制器
as failure detection mechanism, in distributed systems 作为失败检测机制,分布式系统(中的)
term description 术语描述
handspread 扫描窗口
in Solaris 2 virtual memory implementation Solaris 2 虚拟内存实现(的)
happened-before relation 前已发生关系

L

M

N

O

P

Q

S

T

U

V