

■ 教育部 高等教育司 推荐
■ 国外优秀信息科学与技术系列教学用书

OPERATING SYSTEM CONCEPTS (Seventh Edition)

操作系统概念

第七版

翻译版

Abraham Silberschatz

■ [美] Peter Baer Galvin 著

Greg Gagne

■ 郑扣根 译



高等教育出版社
Higher Education Press

教育部高等教育司推荐
国外优秀信息科学与技术系列教学用书

操作系统概念

Caozuo Xitong Gainian

(第七版 翻译版)

OPERATING SYSTEM CONCEPTS

(Seventh Edition)

Abraham Silberschatz

[美] Peter Baer Galvin 著

Greg Gagne

郑扣根 译



高等教育出版社·北京
HIGHER EDUCATION PRESS BEIJING

图字：01-2006-5425 号

Operating System Concepts, Seventh Edition, Simplified Chinese Edition

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

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

Copyright © 2005 John Wiley & Sons, Inc.

All Rights Reserved.

AUTHORIZED TRANSLATION OF THE EDITION PUBLISHED BY JOHN WILEY & SONS, New York, Chichester, Brisbane, Singapore AND Toronto. No part of this book may be reproduced in any form without the written permission of John Wiley & Sons inc.

For sale and distribution in the People's Republic of China exclusively(except Taiwan, Hong Kong SAR and Macao SAR).

仅限于中华人民共和国境内(但不允许在中国香港、澳门特别行政区和中国台湾地区)销售发行。

图书在版编目(CIP)数据

操作系统概念: 第七版: 翻译版 / (美)西尔伯查茨 (Silberschatz, A.), (美)高尔文 (Galvin, P.B.), (美)加根 (Gagne, G.) 著; 郑扣根译. —北京: 高等教育出版社, 2010.1

书名原文: Operating System Concepts, Seventh Edition
ISBN 978-7-04-028341-9

I. 操… II. ①西… ②高… ③加… ④郑… 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(2009)第 224835 号

| | | | |
|------|----------------|------|---|
| 出版发行 | 高等教育出版社 | 购书热线 | 010-58581118 |
| 社 址 | 北京市西城区德外大街 4 号 | 咨询电话 | 400-810-0598 |
| 邮政编码 | 100120 | 网 址 | http://www.hep.edu.cn |
| 总 机 | 010-58581000 | | http://www.hep.com.cn |
| 经 销 | 蓝色畅想图书发行有限公司 | 网上订购 | http://www.landaco.com |
| 印 刷 | 北京民族印务有限责任公司 | | http://www.landaco.com.cn |
| | | 畅想教育 | http://www.widedu.com |
| 开 本 | 787×1092 1/16 | 版 次 | 2010 年 1 月第 1 版 |
| 印 张 | 52 | 印 次 | 2010 年 1 月第 1 次印刷 |
| 字 数 | 1 096 000 | 定 价 | 60.00 元 |

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

版权所有 侵权必究

物料号 28341-00

序

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

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

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

这项教材引进工作是在教育部高等教育司和高等教育出版社的共同组织下，由国内信息科学技术领域的专家、教授广泛参与，在对大量国外教材进行多次遴选的基础上，参考了国内和国外著名大学相关专业的课程设置进行系统引进的。其中，John Wiley 公司出版的贝尔实验室信息科学研究中心副总裁 Silberschatz 教授的经典著作《操作系统概念》，是我们经过反复谈判，做了很多努力才得以引进的。William Stallings 先生曾编写了在美国深受欢迎的信息科学技术系列教材，其中有多种教材获得过美国教材和学术著作者协会颁发的计算机科学与工程教材奖，这批引进教材中就有他的两本著作。留美中国学者 Jiawei Han 先生的《数据挖掘》是该领域中具有里程碑意义的著作。由达特茅斯学院 Thomas Cormen 和麻省理工学院、哥伦比亚大学的几位学者共同编著的经典著作《算法导论》，在经历了 11 年的锤炼之后于 2001 年出版了第二版。目前任教于美国 Massachusetts 大学的 James Kurose 教授，曾在美国三所高校先后 10 次获得杰出教师或杰出教学奖，由他主编的《计算机网络》出版后，以其体系新颖、内容先进而备受欢迎。在努力降低引进教材售价方面，高等教育出版社做了大量和细致的工作。

作。这套引进的教材体现了权威性、系统性、先进性和经济性等特点。

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

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

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

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

教育部高等教育司

二〇〇二年三月

译者序

操作系统对学习计算机的人来说早已不是什么陌生的字眼，作为计算机系统的基本组成部分，它正在以惊人的速度发生着变化；而同样作为计算机专业教学的基本组成部分的操作系统课程，也在随之发生许多改变。书店中的此类书籍可谓琳琅满目，但真正的好书却凤毛麟角。一本书，能被人誉为经典，当然是一本好书。由 John Wiley 公司出版的美国耶鲁大学计算机科学系主任 Silberschatz 教授等编写的《操作系统概念》（第七版）就是这样一本经典之作，自第一版问世以来，经历了 20 余年的锤炼，已经成为操作系统教材的一本“圣经”。相信本系列书的每一位读者都和我一样，从接触到它的某一版本开始，便将之作为学习操作系统的不二之选，不断地收藏和学习它的每个更新版本，仔细品读，并从中获益匪浅。

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

作为一本操作系统的经典之作，除了传承本书之前版本的优点之外，本版主要有以下几个特点：

1. 内容全面。全书共分八部分，内容涉及操作系统的主要部件以及基本的计算机组成结构、进程管理、内存管理、存储管理、保护与安全、分布式系统、专用系统，以及对 Linux、Windows XP 等实例进行分析与讨论的案例研究，覆盖了操作系统的各个重要方面。

2. 书中所有提及的原理都有相应的详细解释，并配有很多实例和插图帮助读者理解，以充实的内容在抽象概念和实际实现之间架设了桥梁。本书讨论了操作系统中的基本概念与算法，提供了大量与特定操作系统相关的例子，如 Sun Microsystems 的 Solaris，Linux，Mach，Microsoft 的 MS-DOS、Windows NT、Windows 2000 和 Windows XP，DEC VMS 和 TOPS-20，IBM OS/2，以及 Apple Mac OS X。为读者深入浅出地学习和理解操作系统提供了坚实的理论基础，用风趣而智慧的语言讲解许多抽象的概念。

3. 内容新颖。由于该书已连续出版七次，每次更新都对前一次的不足进行了修改，而且还结合当前的技术，增加了最新的内容。与第六版相比，此版本不但采纳了读者对以前版本的许多评论和建议，而且还加入一些快速发展的操作系统和网络领域的新概念，对绝大多数章节的内容进行了改写以反映最新的变化，对不再适用的内容作了删除。本书的

大多数示例程序是用 C 语言写的,如果您对 Java 编程环境更为熟悉,建议您阅读本书的 Java 版本(《操作系统概念——Java 实现》(第七版))。

整体上看,本书具有内容新颖、全面,实用性、指导性强等特点,不但是从事操作系统应用开发等专业人士的必备之书,同时也是高等院校相关专业的师生教学的最佳教材。由衷地希望所有读者都能从本书中充分体会到操作系统的精髓,并能在今后的相关工作中游刃有余。

本书的翻译力求忠实于原著。我们在许多操作系统的专业术语后面注明了英文原文。这一方面是为了方便读者能对照理解,为其以后的学习打下基础;另一方面也为了避免因不同中文译法带来的歧义,从而节省读者宝贵的时间。

本书由郑扣根教授翻译。在本书的翻译过程中,得到了姜富强、杨蕾婧、孙莹莹、田稷等同志的许多帮助,在此表示深深的谢意。同时也非常感谢高等教育出版社的编辑们给予我们耐心的等待和支持。

由于种种原因,书中难免存在错误和不妥之处,恳请读者批评指正。

译者

2009 年 12 月

前 言

操作系统是计算机系统的基本组成部分。同样，“操作系统”课程也是计算机教学的基本组成部分。随着计算机在众多领域（从儿童游戏到极为尖端的政府和跨国公司使用的规划工具等）得到广泛应用，操作系统也正在以惊人的速度发展。然而，操作系统的基本概念仍然是比较清晰的，这些概念是本书所讨论的基础。

本书是一本操作系统导论的教科书，适用于大学三、四年级和研究生一年级学生。我们也希望它对相关工程技术人员也有所帮助。本书清晰地描述了操作系统的基本概念。作为阅读本书的前提，我们假设读者熟悉基本的数据结构、计算机组成和一种高级语言，例如 C。本书第 1 章介绍了学习操作系统所需的硬件知识。示例代码主要采用 C 语言编写，有时也使用 Java 语言编写。不过，即使读者没有这些语言的全面知识，也能理解这些算法。

本书不仅直观地描述了概念，同时还给出了重要理论的结论，但省略了形式化的证明。推荐读物指出了相关研究论文，其中有的论文首次提出并证明了这些结论，有的资料是可供进一步阅读的材料。本书还通过图和例子来代替证明，以说明我们所关注的结论的正确性。

本书所描述的基本概念和算法通常是基于既有的商用操作系统，我们的目的是根据通用操作系统而不是特定操作系统来描述这些概念和算法。本书还提供了大量最通用和最具有创造性的操作系统的例子，如 Sun Microsystems 的 Solaris, Linux, Mach, Microsoft 的 MS-DOS、Windows NT、Windows 2000 和 Windows XP, DEC VMS 和 TOPS-20, IBM OS/2, 以及 Apple Mac OS X 操作系统。

在本书中，当以 Windows XP 作为示例操作系统时，表示同时适用于 Windows XP 和 Windows 2000 两个系统。如果某项特性只存在于 Windows XP 中而 Windows 2000 没有，那么我们将明确说明；反之亦然。

本书结构

本书的结构是根据笔者多年来讲授“操作系统”课程的经验来安排的，同时也参考了本书评审专家提供的反馈，以及以前版本的读者提交的意见。此外，本书内容还符合 IEEE 计算机学会和计算机协会（ACM）联合工作组发布的《计算机学科教程 2001（Computing Curricula 2001）》为讲授“操作系统”课程提出的建议。

在本书的支持网页上，提供了几个教学大纲样本，可在讲授“操作系统导论”和“高

级操作系统”课程中使用。一般情况下，建议读者按章节顺序阅读本书，因为这样可以最全面地研究操作系统。不过，读者也可以依据教学大纲样本选择不同的顺序阅读各章（或各章的每个小节）。

本书内容

本书共有八个部分：

- **概述。**第 1 章和第 2 章解释了操作系统是什么，能做什么，以及它们是如何设计与构造的。这一部分讨论了操作系统的公共特性，操作系统为用户提供的服务，操作系统为计算机系统操作员提供的功能。这些描述主要是激励性和解释性的内容。在这两章中，我们避免讨论这些问题的内部实现细节。因此，这部分适合于那些需要了解操作系统而不需要知道其内部算法细节的低年级学生或有关人员。

- **进程管理。**第 3 章到第 7 章描述了进程和并发的概念，这是现代操作系统的核心。进程是系统的工作单元。一个系统由一组并发执行的进程组成，其中部分是系统进程（执行系统代码），其余是用户进程（执行用户代码）。这一部分包括进程调度、进程间通信、进程同步及死锁处理的方法。这部分还讨论了有关线程的知识。

- **内存管理。**第 8 章和第 9 章主要讨论进程执行过程中的内存管理问题。为了改善 CPU 的利用率及其对用户的响应速度，计算机必须将多个进程同时保存在内存中。内存管理的方案有很多，这反映了有多种途径可进行内存管理，特定算法的有效性与具体应用情形有关。

- **存储管理。**第 10 章到第 13 章描述了现代计算机系统如何处理文件系统、大容量存储和 I/O。文件系统为磁盘上的数据和程序提供了在线存储和访问机制。这些章节描述了存储管理内部的经典算法和结构。这部分内容有助于深入理解这些算法，例如算法的特性、优点和缺点。由于连接到计算机的 I/O 设备种类如此之多，操作系统需要为应用程序提供大量的功能以允许它们全面控制这些设备工作。这部分深入讨论了系统 I/O，包括 I/O 系统设计、接口及系统内部的结构和功能。在很多方面，I/O 设备也是计算机中速度最慢的主要组成部件。因为设备是性能瓶颈，所以这部分也讨论了性能问题。另外，这部分还讨论了与二级存储和三级存储有关的问题。

- **保护与安全。**第 14 章和第 15 章讨论了为使系统中的进程彼此之间不会相互影响，如何对系统中的进程加以保护。出于保护和安全的目的，我们采用了这样一种机制：只有获得操作系统授权的进程才可以使用相应的文件、内存、CPU 和其他资源。保护是一种用来控制程序、进程和用户计算机系统资源访问的机制，这种机制必须提供指定控制和实施控制的方法。安全机制保护系统所存储的信息（数据和代码）和计算机的物理资源，从而避免未经授权的访问、恶意破坏和修改，以及意外地引入不一致。

- **分布式系统。**第 16 章到第 18 章讨论了一组不共享内存或时钟的处理器——分布式系统。这类系统允许用户访问系统所维护的各种资源，从而提高计算速度，改善数据的可用性和可靠性。这类系统也为用户提供了分布式文件系统，分布式文件系统是一种用户、服务器和存储设备分散于分布式系统不同位置的文件服务系统。分布式系统必须提供各种机制来处理进程同步和通信问题，以及处理死锁问题和各种集中式系统所未曾遇到的各种故障。

- **特殊用途系统。**第 19 章和第 20 章论述了用于特殊用途的系统，包括实时系统和多媒体系统。这些系统有不同于本书其余部分所关注的那些通用系统的特殊需求。实时系统可能不仅要求计算结果“正确”，而且要求结果要在特定期限内产生。多媒体系统要求服务质量保证，确保多媒体数据在特定时间段内传送到客户端。

- **案例研究。**本书的第 21 章到第 23 章通过描述实际操作系统，将本书描述的概念融合起来。这些系统包括 Linux、Windows XP、FreeBSD、Mach 和 Windows 2000。我们选择 Linux 和 FreeBSD，是因为 Linux 虽小但足以用于理解操作系统的内涵，而且不是“玩具”操作系统。它们内部算法的选择主要是基于其简单的特性而不是速度和复杂性。在计算机科学系，通常可以很容易得到 Linux 和 FreeBSD 系统，因此许多学生都可以接触到这些系统。我们选择 Windows XP 和 Windows 2000，是因为它为研究在设计和实现上与 UNIX 有很大不同的现代操作系统提供了机会。第 23 章也简要地描述了其他一些有影响的操作系统。

操作系统环境

本书使用了许多实际的操作系统来解释操作系统的基本概念。并且，我们特别关注了微软的操作系统家族（包括 Windows NT、Windows 2000 和 Windows XP）和 UNIX 的各种版本（包括 Solaris、BSD 和 Mac OS X）。我们还用了大量的篇幅来描述 Linux 2.6 版内核，这是写作本书时该系统的最新内核。

本书还提供了几个用 C 和 Java 语言编写的示例程序。这些程序需要运行在以下编程环境中：

- **Windows 系统。**Windows 系统的主要编程环境是 Win32 API（应用程序接口）。Win32 API 为管理进程、线程、内存和外部设备提供了完整的函数集合。我们提供了几个 C 程序来演示 Win32 API 的使用。示例程序均在 Windows 2000 和 Windows XP 系统上测试过。

- **POSIX。**POSIX（可移植操作系统接口）代表了一组基于 UNIX 操作系统的标准接口。虽然 Windows XP 和 Windows 2000 也可以运行部分 POSIX 程序，但我们主要针对 UNIX 和 Linux 操作系统来描述 POSIX。POSIX 兼容的系统必须实现 POSIX 核心标准（POSIX.1），Linux、Solaris 和 Mac OS X 都是 POSIX 兼容系统。POSIX 还定义了核心标准的一些扩展，

包括实时扩展 (POSIX1.b) 和线程库扩展 (POSIX1.c, 又称为 Pthread)。我们提供了 POSIX 基本 API、Pthread 和实时扩展的演示程序, 这些程序用 C 语言编写。这些演示程序在 Debian Linux 2.4 和 2.6、Mac OS X 以及 Solaris 9 系统上测试过, 测试用的编译器是 gcc 3.3。

- **Java。**Java 是一种广泛使用的编程语言, 它有丰富的 API, 并对线程创建和管理有内建语言支持。Java 程序可以运行在支持 Java 虚拟机的任何操作系统上。我们用 Java 程序展示了许多操作系统和网络概念, 这些程序在 Java 1.4 虚拟机上测试过。

选择这三种编程环境的原因是, 我们认为它们代表了最流行的两个操作系统模型 Windows 和 UNIX/Linux, 以及广泛使用的 Java 环境。大多数示例程序是用 C 语言写的, 期望读者习惯这种编程语言; 对 C 语言和 Java 语言都熟悉的读者可以很容易地理解本书提供的大多数示例程序。

有时我们会使用三种编程环境来分别解释一个概念, 例如线程创建。这让读者在解决相同任务的时候可以比较这三个不同的库。在其他情况下, 可能仅用其中一种来解释某个概念。例如使用 POSIX API 来解释共享内存, 使用 Java API 来解释 TCP/IP 下的 Socket 编程。

第 七 版

在编写本书时, 我们不但采纳了读者对以前版本的许多评论和建议, 而且还加入了一些快速发展的操作系统和网络领域的新概念。我们对绝大多数章节的内容进行了改写以反映最新的变化, 对不再适用的内容做了删除。

我们对许多章节都做了大量改写和重新组织。最为重要的是, 我们完全重新组织了第 1 章和第 2 章的内容, 并增加了两章来描述特殊用途系统 (实时嵌入式系统和多媒体系统)。因为保护和安全在操作系统中越来越流行, 所以把这部分内容提前了, 并且大量地更新和扩展了对安全的讨论。

下面简要介绍本书各章的主要变化。

- **第 1 章 导论**, 已经全部修改。之前的版本中, 本章给出了操作系统发展历史的概述。新的第 1 章概述了操作系统的主要部件, 以及基本的计算机组成结构。

- **第 2 章 操作系统结构**, 是以前第 3 章的修订版, 它有很多新增内容, 包括对系统调用和操作系统结构的更深入的论述, 另外还包括对虚拟机的重要更新。

- **第 3 章 进程**, 是以前的第 4 章。它新增了在 Linux 中如何表示进程和使用 POSIX 和 Win32 API 来说明进程创建。通过一个 POSIX 系统中共享内存 API 的示例程序, 增强了对共享内存的描述。

- **第 4 章 线程**, 是以前的第 5 章。本章增强了对线程库的论述, 包括 POSIX、Win32 API 和 Java 线程库, 并更新了 Linux 线程的内容。

• **第 5 章 CPU 调度**，是以以前的第 6 章。本章对多处理器系统的调度问题有很多新的讨论，包括处理器亲和性和负载平衡算法。新增“线程调度”一节，包括 Pthread 和 Solaris 中表驱动调度的更新内容。Linux 调度的小节已修订，反映了 Linux 2.6 内核中的调度器。

• **第 6 章 进程同步**，是以以前的第 7 章。由于现代处理器不能保证双进程算法的正确执行，因此删除了双进程算法的内容，现在只讨论 Peterson 解法。本章还新增了对 Linux 内核和 Pthread API 中的同步的描述。

• **第 7 章 死锁**，是以以前的第 8 章。新增内容包括一个多线程 Pthread 程序的死锁例子。

• **第 8 章 内存管理**，是以以前的第 9 章。本章不再涉及覆盖 (overlay)。此外，分段部分做了很大修改，包括加强了对 Pentium 系统中分段的论述和 Linux 中如何设计分段系统的论述。

• **第 9 章 虚拟内存**，是以以前的第 10 章。本章扩展了对虚拟内存和内存映射文件的论述，提供了一个通过内存映射文件实现共享内存的示例程序，该示例程序使用 Win32 API 编写。更新了对内存管理硬件细节的描述。新增的小节描述了在内核中使用 Buddy 算法和 slab 分配器来分配内存。

• **第 10 章 文件系统接口**，是以以前的第 11 章。本章增加了 Windows XP ACL 的例子。

• **第 11 章 文件系统实现**，是以以前的第 12 章。新增 WAFL 文件系统的全面描述和对 Sun 的 ZFS 文件系统的讨论。

• **第 12 章 大容量存储器的结构**，是以以前的第 14 章。新增现代存储阵列的内容，包括新的 RAID 技术和特性，如精简配置。

• **第 13 章 I/O 输入系统**，是对以前第 13 章的更新，增加了新内容。

• **第 14 章 保护**，是对以前第 18 章的更新，新增最小权限原则。

• **第 15 章 安全**，是以以前的第 19 章。本章进行了很大的修改，更新了所有小节。新增一个缓冲区溢出的完整例子，扩展了威胁、加密和安全工具的内容。

• **第 16 章到第 18 章**，是以以前的第 15 章到第 17 章，增加了新内容。

• **第 19 章 实时系统**，是全新的一章，集中研究实时和嵌入式计算机系统，它们的要求不同于传统系统。本章概述了实时计算机系统，并描述了如何构建操作系统以满足这些系统的严格的时间期限。

• **第 20 章 多媒体系统**，是全新的一章，详述了相对较新的多媒体系统领域的发展。多媒体数据不同于常规数据，因为多媒体数据（如视频中的帧）必须按照特定时间限制完成传输（流化）。本章探索这些要求如何影响操作系统的设计。

• **第 21 章 Linux 系统**，是以以前的第 20 章，更新反映了 Linux 2.6 内核的改变，Linux 2.6 内核是本书编写时最新的 Linux 内核。

• **第 22 章 Windows XP**，已更新。

- **第 23 章** 有影响的操作系统，已更新。

以前的第 21 章（Windows 2000）已改成现在的附录 C。与本书之前的版本一样，附录在线提供。

编程练习和项目

为了巩固本书所介绍的概念，我们提供几个新的编程练习和项目，它们使用 POSIX、Win32 API 或 Java。我们新增加了 15 个以上的编程练习，强化了进程、线程、共享内存、进程同步和网络。此外，还增加了几个编程项目，它们比标准编程练习更复杂。这些项目包括给 Linux 内核增加一个系统调用、使用 `fork()` 系统调用创建一个 UNIX Shell、创建一个多线程矩阵应用程序以及使用共享内存的生产者-消费者问题。

教学辅助材料和网页

本书的网页包括很多资料，如与本书配套的一套幻灯片、课程教学大纲样本、所有的 C 和 Java 的源代码和最新勘误表。网页也包括本书的三个案例研究附录和分布式通信附录。网址是：

<http://www.os-book.com>

本版新增一个称为“学生解答手册” (Student Solutions Manual) 的书面补充材料，它包括一些问题和练习的答案（本书没有这些答案），应该有助于学生理解本书的概念。读者可以在 John Wiley 的网站上购买该书面材料，在 <http://www.wiley.com/college/silberschatz> 上选择学生解答手册的链接。

为了得到受限制的资源，如本书习题的解答指导，请与本地的 John Wiley & Sons 销售代理联系。注意，这些资源只对使用本书的教师可用。读者可以通过“Find a Rep?” 网页 (<http://www.jsw-edcv.wiley.com/college/findarep>) 来找到当地的代理。

邮 件 列 表

我们现在改用邮件系统来方便使用本书的用户联系。如果你希望使用这项功能，请访问下面的网址，按步骤订阅：

<http://mailman.cs.yale.edu/mailman/listinfo/os-book-list>

邮件列表系统提供了很多便利，如存档信息以及几个订阅选项，包括只订阅摘要和网页。要发送消息到该列表，可以发 E-mail 到：

os-book-list@cs.yale.edu

基于邮件内容，我们将个人单独回复或转发邮件到邮件列表的每个人。列表是受控的，因此读者不会收到垃圾邮件。

使用本书作为教材的学生，不应使用列表询问习题的答案，我们也不会提供答案。

建 议

我们已尽量消除本版中的所有错误，但是与操作系统一样，可能仍然存在一些隐藏的误差。我们非常希望收到您所发现的任何本书的文字错误或遗漏。

如果您希望提供改进建议或提供习题，那么我们也非常高兴。请发送邮件至 `os-book@cs.yale.edu`。

致 谢

本书是根据以前版本修改而来的，前三版是与 James Peterson 合著的。帮助完成以前版本的人还有：Hamid Arabnia、Rida Bazzi、Randy Bentson、David Black、Joseph Boykin、Jeff Brumfield、Gael Buckley、Roy Campbell、P.C.Capon、John Carpenter、Gil Carrick、Thomas Casavant、Ajoy Kumar Datta、Joe Deck、Sudarshan K.Dhall、Thomas Doepfner、Caleb Drake、M.Racsit Eskicioglu、Hans Flack、Robert Fowler、G.Scott Graham、Richard Guy、Max Hailperin、Rebecca Hartman、Wayne Hathaway、Christopher Haynes、Bruce Hillyer、Mark Holliday、Ahmed Kamel、Richard Kiebertz、Carol Kroll、Morty Kwestel、Thomas LeBlanc、John Leggett、Jerrold Leichter、Ted Leung、Gary Lippman、Carolyn Miller、Michael Molloy、Yoichi Muraoka、Jim M.Ng、Banu Ozden、Ed Posnak、Boris Putanec、Charles Qualline、John Quarterman、Mike Reiter、Gustavo RodriguezRivera、Carolyn J.C.Schauble、Thomas P.Skinner、Yannis Smaragdakis、Jesse St.Laurent、John Stankovic、Adam Stauffer、Steven Stepanek、Hal Stern、Louis Stevens、Pete Thomas、David Umbaugh、Steve Vinoski、TommyWagner、Larry L.Wear、John Werth、James M.Westall、J.S.Weston 和 Yang Xiang。

第 12 章的部分内容取自 Hillyer 和 Silberschatz^[1996]的一篇论文。第 17 章的部分内容取自 Levy 和 Silberschatz^[1990]的一篇论文。第 21 章取自 Stephen Tweedie 未发表的手稿。第 2 章取自 Dave Probert、Cliff Martin 和 Avi Silberschatz 未发表的手稿。附录 C 取自 Cliff Martin 未发表的手稿。Cliff Martin 还帮助更新了 UNIX 附录以描述 FreeBSD。Mike Shapiro、Alan Cantrill 和 Jim Mauro 回答了多个有关 Solaris 的问题。Josh Dees 和 Rob Reynolds 对 .NET 的讨论做出了贡献。美国佛蒙特州 Winooski 市 St. Michael's College 的 John Ho 提供了设计和增强 UNIX Shell 接口的项目。

本版有很多新的习题和相应的解答是由 Arvind Krishnamurthy 提供的。

我们感谢审阅本版的以下各位: Bart Childs、Don Heller、Dean Hougen Michael Huangs、Morty Kewstel、Euripides Montagne 和 John Sterling。

顾问编辑 Bill Zobrist 和 Paul Crockett 在我们写作本书期间给予了专家级指导。他们的助理 Simon Durkin 管理了许多细节以保证本书顺利完成。高级制作编辑是 Ken Santor。封面制作是 Susan Cyr, 封面设计是 Madelyn Lesure。Beverly Peavler 负责了编辑审稿。校对是 Katrina Avery (自由职业), 索引制作是 Rosemary Simpson (自由职业)。Marilyn Turnamin 帮助生成了图和演示幻灯片。

最后, 我们还希望感谢一些人。Avi 开始了他人生的新篇章, 重新回到学术界, 并和 Valerie 开始了新生活, 这使他可以全心地写作本书。Pete 感谢他的家人、朋友和同事在项目期间的支持和理解。Greg 感谢家人一直以来的关心和支持。他还要特别感谢 Peter Ormsby, 他不管多忙总是首先询问“写作进行得怎么样了?”。

Abraham Silberschatz, New Haven, CT, 2004

Peter Baer Galvin, Burlington, MA, 2004

Greg Gagne, Salt Lake City, UT, 2004

目 录

第一部分 概 述

| | | | |
|----------------|----|-------------------|----|
| 第 1 章 导论 | 3 | 1.12.2 客户机-服务器计算 | 29 |
| 1.1 操作系统做什么 | 3 | 1.12.3 对等计算 | 29 |
| 1.1.1 用户视角 | 4 | 1.12.4 基于 Web 的计算 | 30 |
| 1.1.2 系统视角 | 5 | 1.13 小结 | 30 |
| 1.1.3 定义操作系统 | 5 | 习题 | 32 |
| 1.2 计算机系统组织 | 6 | 文献注记 | 33 |
| 1.2.1 计算机系统操作 | 6 | 第 2 章 操作系统结构 | 34 |
| 1.2.2 存储结构 | 7 | 2.1 操作系统服务 | 34 |
| 1.2.3 I/O 结构 | 9 | 2.2 操作系统的用户界面 | 36 |
| 1.3 计算机系统体系结构 | 10 | 2.2.1 命令解释程序 | 36 |
| 1.3.1 单处理器系统 | 11 | 2.2.2 图形用户界面 | 36 |
| 1.3.2 多处理器系统 | 11 | 2.3 系统调用 | 37 |
| 1.3.3 集群系统 | 13 | 2.4 系统调用类型 | 41 |
| 1.4 操作系统结构 | 14 | 2.4.1 进程控制 | 41 |
| 1.5 操作系统操作 | 16 | 2.4.2 文件管理 | 45 |
| 1.5.1 双重模式操作 | 16 | 2.4.3 设备管理 | 46 |
| 1.5.2 定时器 | 18 | 2.4.4 信息维护 | 46 |
| 1.6 进程管理 | 18 | 2.4.5 通信 | 46 |
| 1.7 内存管理 | 19 | 2.5 系统程序 | 47 |
| 1.8 存储管理 | 20 | 2.6 操作系统设计和实现 | 48 |
| 1.8.1 文件系统管理 | 20 | 2.6.1 设计目标 | 48 |
| 1.8.2 大容量存储器管理 | 21 | 2.6.2 机制与策略 | 49 |
| 1.8.3 高速缓存 | 21 | 2.6.3 实现 | 49 |
| 1.8.4 I/O 系统 | 23 | 2.7 操作系统结构 | 50 |
| 1.9 保护和安全 | 23 | 2.7.1 简单结构 | 50 |
| 1.10 分布式系统 | 25 | 2.7.2 分层方法 | 52 |
| 1.11 专用系统 | 26 | 2.7.3 微内核 | 53 |
| 1.11.1 实时嵌入式系统 | 26 | 2.7.4 模块 | 54 |
| 1.11.2 多媒体系统 | 27 | 2.8 虚拟机 | 55 |
| 1.11.3 手持系统 | 27 | 2.8.1 实现 | 56 |
| 1.12 计算环境 | 28 | 2.8.2 优点 | 57 |
| 1.12.1 传统计算 | 28 | 2.8.3 实例 | 57 |

| | | | |
|-----------------------------|-----|---------------------------------|-----|
| 2.9 系统生成 | 60 | 习题 | 63 |
| 2.10 系统启动 | 61 | 项目: 向 Linux 内核增加一个系统调用 | 64 |
| 2.11 小结 | 62 | 文献注记 | 67 |
| 第二部分 进 程 管 理 | | | |
| 第 3 章 进 程 | 71 | 4.2.2 一对一模型 | 114 |
| 3.1 进程概念 | 71 | 4.2.3 多对多模型 | 114 |
| 3.1.1 进程 | 71 | 4.3 线程库 | 115 |
| 3.1.2 进程状态 | 72 | 4.3.1 Pthread | 116 |
| 3.1.3 进程控制块 | 73 | 4.3.2 Win32 线程 | 117 |
| 3.1.4 线程 | 74 | 4.3.3 Java 线程 | 119 |
| 3.2 进程调度 | 75 | 4.4 多线程问题 | 121 |
| 3.2.1 调度队列 | 75 | 4.4.1 系统调用 fork()和 exec() | 121 |
| 3.2.2 调度程序 | 77 | 4.4.2 取消 | 122 |
| 3.2.3 上下文切换 | 78 | 4.4.3 信号处理 | 122 |
| 3.3 进程操作 | 79 | 4.4.4 线程池 | 123 |
| 3.3.1 进程创建 | 79 | 4.4.5 线程特定数据 | 125 |
| 3.3.2 进程终止 | 84 | 4.4.6 调度程序激活 | 125 |
| 3.4 进程间通信 | 84 | 4.5 操作系统实例 | 126 |
| 3.4.1 共享内存系统 | 86 | 4.5.1 Windows XP 线程 | 126 |
| 3.4.2 消息传递系统 | 87 | 4.5.2 Linux 线程 | 127 |
| 3.5 IPC 系统的实例 | 90 | 4.6 小结 | 128 |
| 3.5.1 实例: POSIX 共享内存 | 90 | 习题 | 129 |
| 3.5.2 实例: Mach | 91 | 项目: 矩阵乘法 | 130 |
| 3.5.3 实例: Windows XP | 94 | 文献注记 | 133 |
| 3.6 客户机-服务器系统通信 | 95 | 第 5 章 CPU 调度 | 134 |
| 3.6.1 Socket | 95 | 5.1 基本概念 | 134 |
| 3.6.2 远程过程调用 | 98 | 5.1.1 CPU-I/O 区间周期 | 134 |
| 3.6.3 远程方法调用 | 101 | 5.1.2 CPU 调度程序 | 135 |
| 3.7 小结 | 102 | 5.1.3 抢占调度 | 135 |
| 习题 | 103 | 5.1.4 分派程序 | 136 |
| 项目: UNIX Shell 和历史特点 | 106 | 5.2 调度准则 | 137 |
| 文献注记 | 110 | 5.3 调度算法 | 138 |
| 第 4 章 线 程 | 111 | 5.3.1 先到先服务调度 | 138 |
| 4.1 概述 | 111 | 5.3.2 最短作业优先调度 | 139 |
| 4.1.1 动机 | 112 | 5.3.3 优先级调度 | 141 |
| 4.1.2 优点 | 112 | 5.3.4 轮转法调度 | 142 |
| 4.2 多线程模型 | 113 | 5.3.5 多级队列调度 | 145 |
| 4.2.1 多对一模型 | 113 | 5.3.6 多级反馈队列调度 | 146 |

| | | | |
|-------------------------|-----|---------------------|-----|
| 5.4 多处理器调度 | 147 | 6.8.2 Windows XP 同步 | 189 |
| 5.4.1 多处理器调度的方法 | 148 | 6.8.3 Linux 同步 | 190 |
| 5.4.2 处理器亲和性 | 148 | 6.8.4 Pthread 同步 | 191 |
| 5.4.3 负载均衡 | 148 | 6.9 原子事务 | 191 |
| 5.4.4 对称多线程 | 149 | 6.9.1 系统模型 | 192 |
| 5.5 线程调度 | 150 | 6.9.2 基于日志的恢复 | 192 |
| 5.5.1 竞争范围 | 150 | 6.9.3 检查点 | 193 |
| 5.5.2 Pthread 调度 | 150 | 6.9.4 并发原子操作 | 194 |
| 5.6 操作系统实例 | 152 | 6.10 小结 | 198 |
| 5.6.1 实例: Solaris 调度 | 152 | 习题 | 198 |
| 5.6.2 实例: Windows XP 调度 | 154 | 项目: 生产者-消费者问题 | 202 |
| 5.6.3 实例: Linux 调度 | 156 | 文献注记 | 207 |
| 5.7 算法评估 | 158 | 第 7 章 死锁 | 209 |
| 5.7.1 确定模型 | 158 | 7.1 系统模型 | 209 |
| 5.7.2 排队模型 | 160 | 7.2 死锁特征 | 210 |
| 5.7.3 模拟 | 160 | 7.2.1 必要条件 | 211 |
| 5.7.4 实现 | 161 | 7.2.2 资源分配图 | 212 |
| 5.8 小结 | 162 | 7.3 死锁处理方法 | 214 |
| 习题 | 163 | 7.4 死锁预防 | 215 |
| 文献注记 | 165 | 7.4.1 互斥 | 216 |
| 第 6 章 进程同步 | 166 | 7.4.2 占有并等待 | 216 |
| 6.1 背景 | 166 | 7.4.3 非抢占 | 216 |
| 6.2 临界区问题 | 168 | 7.4.4 循环等待 | 217 |
| 6.3 Peterson 算法 | 169 | 7.5 死锁避免 | 218 |
| 6.4 硬件同步 | 170 | 7.5.1 安全状态 | 218 |
| 6.5 信号量 | 173 | 7.5.2 资源分配图算法 | 220 |
| 6.5.1 用法 | 173 | 7.5.3 银行家算法 | 220 |
| 6.5.2 实现 | 174 | 7.6 死锁检测 | 224 |
| 6.5.3 死锁与饥饿 | 176 | 7.6.1 每种资源类型只有单个实例 | 224 |
| 6.6 经典同步问题 | 176 | 7.6.2 每种资源类型可有多个实例 | 225 |
| 6.6.1 有限缓冲问题 | 177 | 7.6.3 应用检测算法 | 226 |
| 6.6.2 读者-写者问题 | 177 | 7.7 死锁恢复 | 227 |
| 6.6.3 哲学家进餐问题 | 179 | 7.7.1 进程终止 | 227 |
| 6.7 管程 | 180 | 7.7.2 资源抢占 | 227 |
| 6.7.1 使用 | 181 | 7.8 小结 | 228 |
| 6.7.2 哲学家进餐问题的管程解决方案 | 183 | 习题 | 229 |
| 6.7.3 基于信号量的管程实现 | 183 | 文献注记 | 231 |
| 6.7.4 管程内的进程重启 | 185 | | |
| 6.8 同步实例 | 187 | | |
| 6.8.1 Solaris 同步 | 187 | | |

第三部分 内存管理

| | | | |
|--------------------------|-----|------------------------|-----|
| 第 8 章 内存管理 | 235 | 9.4 页面置换 | 280 |
| 8.1 背景 | 235 | 9.4.1 基本页置换 | 281 |
| 8.1.1 基本硬件 | 235 | 9.4.2 FIFO 页置换 | 284 |
| 8.1.2 地址绑定 | 237 | 9.4.3 最优置换 | 285 |
| 8.1.3 逻辑地址空间与物理地址空间 | 239 | 9.4.4 LRU 页置换 | 286 |
| 8.1.4 动态加载 | 240 | 9.4.5 近似 LRU 页置换 | 287 |
| 8.1.5 动态链接与共享库 | 240 | 9.4.6 基于计数的页置换 | 289 |
| 8.2 交换 | 241 | 9.4.7 页缓冲算法 | 290 |
| 8.3 连续内存分配 | 243 | 9.4.8 应用程序与页置换 | 290 |
| 8.3.1 内存映射与保护 | 243 | 9.5 帧分配 | 291 |
| 8.3.2 内存分配 | 244 | 9.5.1 帧的最少数量 | 291 |
| 8.3.3 碎片 | 245 | 9.5.2 分配算法 | 292 |
| 8.4 分页 | 246 | 9.5.3 全局分配与局部分配 | 293 |
| 8.4.1 基本方法 | 246 | 9.6 系统颠簸 | 293 |
| 8.4.2 硬件支持 | 250 | 9.6.1 系统颠簸的原因 | 294 |
| 8.4.3 保护 | 252 | 9.6.2 工作集模型 | 296 |
| 8.4.4 共享页 | 254 | 9.6.3 页错误频率 | 297 |
| 8.5 页表结构 | 255 | 9.7 内存映射文件 | 298 |
| 8.5.1 层次页表 | 255 | 9.7.1 基本机制 | 299 |
| 8.5.2 哈希页表 | 257 | 9.7.2 Win32 API 中的共享内存 | 300 |
| 8.5.3 反向页表 | 258 | 9.7.3 内存映射 I/O | 303 |
| 8.6 分段 | 260 | 9.8 内核内存的分配 | 303 |
| 8.6.1 基本方法 | 260 | 9.8.1 Buddy 系统 | 303 |
| 8.6.2 硬件 | 261 | 9.8.2 slab 分配 | 304 |
| 8.7 实例: Intel Pentium | 263 | 9.9 其他考虑 | 306 |
| 8.7.1 Pentium 分段 | 263 | 9.9.1 预调页 | 306 |
| 8.7.2 Pentium 分页 | 264 | 9.9.2 页大小 | 306 |
| 8.7.3 Pentium 系统上的 Linux | 264 | 9.9.3 TLB 范围 | 308 |
| 8.8 小结 | 266 | 9.9.4 反向页表 | 308 |
| 习题 | 267 | 9.9.5 程序结构 | 309 |
| 文献注记 | 269 | 9.9.6 I/O 互锁 | 310 |
| 第 9 章 虚拟内存 | 270 | 9.10 操作系统实例 | 311 |
| 9.1 背景 | 270 | 9.10.1 Windows XP | 311 |
| 9.2 按需调页 | 272 | 9.10.2 Solaris | 312 |
| 9.2.1 基本概念 | 273 | 9.11 小结 | 313 |
| 9.2.2 按需调页的性能 | 276 | 习题 | 314 |
| 9.3 写时复制 | 278 | 文献注记 | 317 |

第四部分 存 储 管 理

| | | | |
|----------------------------|-----|----------------------------------|-----|
| 第 10 章 文件系统接口 | 321 | 11.3 目录实现 | 360 |
| 10.1 文件概念 | 321 | 11.3.1 线性列表 | 360 |
| 10.1.1 文件属性 | 322 | 11.3.2 哈希表 | 361 |
| 10.1.2 文件操作 | 322 | 11.4 分配方法 | 361 |
| 10.1.3 文件类型 | 326 | 11.4.1 连续分配 | 361 |
| 10.1.4 文件结构 | 327 | 11.4.2 链接分配 | 363 |
| 10.1.5 内部文件结构 | 328 | 11.4.3 索引分配 | 366 |
| 10.2 访问方法 | 329 | 11.4.4 性能 | 368 |
| 10.2.1 顺序访问 | 329 | 11.5 空闲空间管理 | 369 |
| 10.2.2 直接访问 | 329 | 11.5.1 位向量 | 369 |
| 10.2.3 其他访问方式 | 330 | 11.5.2 链表 | 370 |
| 10.3 目录结构 | 331 | 11.5.3 组 | 370 |
| 10.3.1 存储结构 | 332 | 11.5.4 计数 | 371 |
| 10.3.2 目录概述 | 332 | 11.6 效率与性能 | 371 |
| 10.3.3 单层结构目录 | 333 | 11.6.1 效率 | 371 |
| 10.3.4 双层结构目录 | 333 | 11.6.2 性能 | 372 |
| 10.3.5 树状结构目录 | 335 | 11.7 恢复 | 374 |
| 10.3.6 无环图目录 | 337 | 11.7.1 一致性检查 | 374 |
| 10.3.7 通用图目录 | 339 | 11.7.2 备份和恢复 | 375 |
| 10.4 文件系统安装 | 340 | 11.8 基于日志结构的文件系统 | 376 |
| 10.5 文件共享 | 342 | 11.9 NFS | 377 |
| 10.5.1 多用户 | 342 | 11.9.1 概述 | 377 |
| 10.5.2 远程文件系统 | 342 | 11.9.2 安装协议 | 379 |
| 10.5.3 一致性语义 | 345 | 11.9.3 NFS 协议 | 379 |
| 10.6 保护 | 346 | 11.9.4 路径名转换 | 380 |
| 10.6.1 访问类型 | 346 | 11.9.5 远程操作 | 381 |
| 10.6.2 访问控制 | 347 | 11.10 实例: WAFL 文件系统 | 382 |
| 10.6.3 其他保护方式 | 349 | 11.11 小结 | 384 |
| 10.7 小结 | 350 | 习题 | 384 |
| 习题 | 351 | 文献注记 | 385 |
| 文献注记 | 352 | 第 12 章 大容量存储器的结构 | 387 |
| 第 11 章 文件系统实现 | 353 | 12.1 大容量存储器结构简介 | 387 |
| 11.1 文件系统结构 | 353 | 12.1.1 磁盘 | 387 |
| 11.2 文件系统实现 | 355 | 12.1.2 磁带 | 389 |
| 11.2.1 概述 | 355 | 12.2 磁盘结构 | 389 |
| 11.2.2 分区与安装 | 357 | 12.3 磁盘附属 | 390 |
| 11.2.3 虚拟文件系统 | 358 | 12.3.1 主机附属存储 | 390 |

| | |
|-------------------|-----|
| 12.3.2 网络附属存储 | 391 |
| 12.3.3 存储区域网络 | 391 |
| 12.4 磁盘调度 | 392 |
| 12.4.1 FCFS 调度 | 393 |
| 12.4.2 SSTF 调度 | 393 |
| 12.4.3 SCAN 调度 | 394 |
| 12.4.4 C-SCAN 调度 | 395 |
| 12.4.5 LOOK 调度 | 396 |
| 12.4.6 磁盘调度算法的选择 | 396 |
| 12.5 磁盘管理 | 397 |
| 12.5.1 磁盘格式化 | 397 |
| 12.5.2 引导块 | 398 |
| 12.5.3 坏块 | 400 |
| 12.6 交换空间管理 | 401 |
| 12.6.1 交换空间的使用 | 401 |
| 12.6.2 交换空间位置 | 401 |
| 12.6.3 实例: 交换空间管理 | 402 |
| 12.7 RAID 结构 | 403 |
| 12.7.1 通过冗余改善可靠性 | 403 |
| 12.7.2 通过并行处理改善性能 | 404 |
| 12.7.3 RAID 级别 | 405 |
| 12.7.4 RAID 级别的选择 | 409 |
| 12.7.5 扩展 | 409 |
| 12.7.6 RAID 的问题 | 410 |
| 12.8 稳定存储实现 | 410 |
| 12.9 三级存储结构 | 411 |
| 12.9.1 三级存储设备 | 411 |
| 12.9.2 操作系统支持 | 413 |
| 12.9.3 性能 | 416 |
| 12.10 小结 | 419 |

| | |
|----------------------|-----|
| 习题 | 421 |
| 文献注记 | 423 |
| 第 13 章 I/O 输入系统 | 425 |
| 13.1 概述 | 425 |
| 13.2 I/O 硬件 | 426 |
| 13.2.1 轮询 | 428 |
| 13.2.2 中断 | 429 |
| 13.2.3 直接内存访问 | 432 |
| 13.2.4 I/O 硬件小结 | 434 |
| 13.3 I/O 应用接口 | 434 |
| 13.3.1 块与字符设备 | 436 |
| 13.3.2 网络设备 | 437 |
| 13.3.3 时钟与定时器 | 437 |
| 13.3.4 阻塞与非阻塞 I/O | 438 |
| 13.4 I/O 内核子系统 | 439 |
| 13.4.1 I/O 调度 | 439 |
| 13.4.2 缓冲 | 440 |
| 13.4.3 高速缓存 | 442 |
| 13.4.4 假脱机与设备预留 | 442 |
| 13.4.5 错误处理 | 443 |
| 13.4.6 I/O 保护 | 443 |
| 13.4.7 内核数据结构 | 444 |
| 13.4.8 内核 I/O 子系统小结 | 445 |
| 13.5 把 I/O 操作转换成硬件操作 | 445 |
| 13.6 流 | 448 |
| 13.7 性能 | 449 |
| 13.8 小结 | 452 |
| 习题 | 453 |
| 文献注记 | 453 |

第五部分 保护与安全

| | |
|--------------------|-----|
| 第 14 章 保护 | 457 |
| 14.1 保护目标 | 457 |
| 14.2 保护原则 | 458 |
| 14.3 保护域 | 459 |
| 14.3.1 域结构 | 459 |
| 14.3.2 实例: UNIX | 461 |
| 14.3.3 实例: MULTICS | 462 |

| | |
|----------------|-----|
| 14.4 访问矩阵 | 463 |
| 14.5 访问矩阵的实现 | 467 |
| 14.5.1 全局表 | 467 |
| 14.5.2 对象的访问列表 | 467 |
| 14.5.3 域的权限列表 | 467 |
| 14.5.4 锁-钥匙机制 | 468 |
| 14.5.5 比较 | 468 |
| 14.6 访问控制 | 469 |

| | | | |
|-----------------------------|------------|-----------------------------|------------|
| 14.7 访问权限的撤回 | 470 | 15.4 作为安全工具的密码术 | 496 |
| 14.8 面向权限的系统 | 472 | 15.4.1 加密 | 497 |
| 14.8.1 实例: Hydra | 472 | 15.4.2 密码术的实现 | 503 |
| 14.8.2 实例: 剑桥 CAP 系统 | 473 | 15.4.3 实例: SSL | 504 |
| 14.9 基于语言的保护 | 474 | 15.5 用户验证 | 506 |
| 14.9.1 基于编译程序的强制 | 474 | 15.5.1 密码 | 506 |
| 14.9.2 Java 的保护 | 476 | 15.5.2 密码脆弱的一面 | 506 |
| 14.10 小结 | 478 | 15.5.3 密码加密 | 508 |
| 习题 | 479 | 15.5.4 一次性密码 | 508 |
| 文献注记 | 480 | 15.5.5 生物测定学 | 509 |
| 第 15 章 安全 | 481 | 15.6 实现安全防护 | 510 |
| 15.1 安全问题 | 481 | 15.6.1 安全策略 | 510 |
| 15.2 程序威胁 | 484 | 15.6.2 脆弱性评估 | 510 |
| 15.2.1 特洛伊木马 | 484 | 15.6.3 入侵检测 | 512 |
| 15.2.2 后门 | 485 | 15.6.4 病毒防护 | 514 |
| 15.2.3 逻辑炸弹 | 486 | 15.6.5 审计、会计和日志 | 516 |
| 15.2.4 栈和缓冲区溢出 | 486 | 15.7 保护系统和网络的防火墙 | 516 |
| 15.2.5 病毒 | 489 | 15.8 计算机安全分类 | 518 |
| 15.3 系统和网络威胁 | 492 | 15.9 实例: Windows XP | 519 |
| 15.3.1 蠕虫 | 492 | 15.10 小结 | 521 |
| 15.3.2 端口扫描 | 495 | 习题 | 521 |
| 15.3.3 拒绝服务 | 495 | 文献注记 | 522 |
| 第六部分 分布式系统 | | | |
| 第 16 章 分布式系统结构 | 527 | 16.5.3 包策略 | 540 |
| 16.1 动机 | 527 | 16.5.4 连接策略 | 541 |
| 16.1.1 资源共享 | 527 | 16.5.5 竞争 | 541 |
| 16.1.2 加快计算速度 | 528 | 16.6 通信协议 | 542 |
| 16.1.3 可靠性 | 528 | 16.7 健壮性 | 545 |
| 16.1.4 通信 | 529 | 16.7.1 故障检测 | 545 |
| 16.2 分布式操作系统的类型 | 529 | 16.7.2 重构 | 545 |
| 16.2.1 网络操作系统 | 529 | 16.7.3 故障恢复 | 546 |
| 16.2.2 分布式操作系统 | 531 | 16.8 设计事项 | 546 |
| 16.3 网络结构 | 533 | 16.9 实例: 连网 | 548 |
| 16.3.1 局域网 | 533 | 16.10 小结 | 550 |
| 16.3.2 广域网 | 534 | 习题 | 550 |
| 16.4 网络拓扑结构 | 535 | 文献注记 | 551 |
| 16.5 通信结构 | 537 | 第 17 章 分布式文件系统 | 552 |
| 16.5.1 命名和名字解析 | 537 | 17.1 背景 | 552 |
| 16.5.2 路由策略 | 539 | 17.2 命名和透明性 | 553 |

| | | | |
|---------------------------|-----|-------------------------|-----|
| 17.2.1 命名结构 | 554 | 18.2 互斥 | 573 |
| 17.2.2 命名方案 | 555 | 18.2.1 集中式算法 | 573 |
| 17.2.3 实现技术 | 556 | 18.2.2 完全分布式的算法 | 574 |
| 17.3 远程文件访问 | 556 | 18.2.3 令牌传递算法 | 575 |
| 17.3.1 基本的缓存设计 | 557 | 18.3 原子性 | 575 |
| 17.3.2 缓存位置 | 557 | 18.3.1 两阶段提交协议 | 576 |
| 17.3.3 缓存更新策略 | 558 | 18.3.2 2PC 中的错误处理 | 577 |
| 17.3.4 一致性 | 559 | 18.4 并发控制 | 578 |
| 17.3.5 高速缓存和远程服务的比较 | 560 | 18.4.1 加锁协议 | 578 |
| 17.4 有状态服务和无状态服务 | 561 | 18.4.2 时间戳 | 580 |
| 17.5 文件复制 | 563 | 18.5 死锁处理 | 582 |
| 17.6 实例: AFS | 563 | 18.5.1 死锁预防和避免 | 582 |
| 17.6.1 概述 | 564 | 18.5.2 死锁检测 | 583 |
| 17.6.2 共享名称空间 | 565 | 18.6 选举算法 | 587 |
| 17.6.3 文件操作和一致性语义 | 566 | 18.6.1 Bully 算法 | 588 |
| 17.6.4 实现 | 567 | 18.6.2 环算法 | 589 |
| 17.7 小结 | 568 | 18.7 达成一致 | 590 |
| 习题 | 569 | 18.7.1 不可靠通信 | 590 |
| 文献注记 | 569 | 18.7.2 出错进程 | 591 |
| 第 18 章 分布式协调 | 571 | 18.8 小结 | 591 |
| 18.1 事件排序 | 571 | 习题 | 592 |
| 18.1.1 事前关系 | 571 | 文献注记 | 593 |
| 18.1.2 实现 | 572 | | |

第七部分 特殊用途系统

| | | | |
|---------------------------|-----|--------------------------|-----|
| 第 19 章 实时系统 | 597 | 19.7 小结 | 612 |
| 19.1 概述 | 597 | 习题 | 613 |
| 19.2 系统特性 | 598 | 文献注记 | 613 |
| 19.3 实时内核特性 | 599 | 第 20 章 多媒体系统 | 614 |
| 19.4 实现实时操作系统 | 601 | 20.1 什么是多媒体 | 614 |
| 19.4.1 基于优先级的调度 | 602 | 20.1.1 媒体传送 | 614 |
| 19.4.2 抢占式内核 | 602 | 20.1.2 多媒体系统的特点 | 616 |
| 19.4.3 最小化延迟 | 602 | 20.1.3 操作系统问题 | 616 |
| 19.5 实时 CPU 调度 | 605 | 20.2 压缩 | 616 |
| 19.5.1 单调速率调度 | 605 | 20.3 多媒体内核的要求 | 618 |
| 19.5.2 最早截止期限优先调度算法 | 607 | 20.4 CPU 调度 | 620 |
| 19.5.3 按比例分享调度 | 608 | 20.5 磁盘调度 | 620 |
| 19.5.4 Pthread 调度 | 608 | 20.5.1 最早期限优先调度 | 621 |
| 19.6 VxWorks 5.x | 610 | 20.5.2 SCAN-EDF 调度 | 621 |

| | | | |
|--------------------|-----|-------------|-----|
| 20.6 网络管理 | 622 | 20.7.2 接纳控制 | 626 |
| 20.6.1 单播和多播 | 623 | 20.8 小结 | 628 |
| 20.6.2 实时流协议 | 623 | 习题 | 628 |
| 20.7 实例: CineBlitz | 626 | 文献注记 | 629 |
| 20.7.1 磁盘调度 | 626 | | |

第八部分 案例研究

| | | | |
|---------------------------|-----|---------------------------------|-----|
| 第 21 章 Linux 系统 | 633 | 21.9.2 进程间数据传输 | 666 |
| 21.1 Linux 发展历程 | 633 | 21.10 网络结构 | 666 |
| 21.1.1 Linux 内核 | 634 | 21.11 安全 | 668 |
| 21.1.2 Linux 系统 | 636 | 21.11.1 认证 | 669 |
| 21.1.3 Linux 发行版 | 636 | 21.11.2 访问控制 | 669 |
| 21.1.4 Linux 许可 | 637 | 21.12 小结 | 670 |
| 21.2 设计原则 | 637 | 习题 | 671 |
| 21.3 内核模块 | 640 | 文献注记 | 672 |
| 21.3.1 模块管理 | 640 | 第 22 章 Windows XP | 673 |
| 21.3.2 驱动程序注册 | 641 | 22.1 历史 | 673 |
| 21.3.3 冲突解决 | 642 | 22.2 设计原则 | 674 |
| 21.4 进程管理 | 642 | 22.2.1 安全性 | 675 |
| 21.4.1 fork()和 exec()进程模式 | 642 | 22.2.2 可靠性 | 675 |
| 21.4.2 进程与线程 | 644 | 22.2.3 Windows 和 POSIX 应用程序的兼容性 | 675 |
| 21.5 调度 | 645 | 22.2.4 高性能 | 676 |
| 21.5.1 进程调度 | 645 | 22.2.5 可扩展性 | 676 |
| 21.5.2 内核同步 | 647 | 22.2.6 可移植性 | 676 |
| 21.5.3 对称多处理技术 | 649 | 22.2.7 国际支持 | 677 |
| 21.6 内存管理 | 650 | 22.3 系统组件 | 677 |
| 21.6.1 物理内存管理 | 650 | 22.3.1 硬件抽象层 | 677 |
| 21.6.2 虚拟内存 | 653 | 22.3.2 内核 | 678 |
| 21.6.3 执行与装入用户程序 | 655 | 22.3.3 执行体 | 682 |
| 21.7 文件系统 | 657 | 22.4 环境子系统 | 697 |
| 21.7.1 虚拟文件系统 | 658 | 22.4.1 MS-DOS 环境 | 698 |
| 21.7.2 Linux ext2fs 文件系统 | 659 | 22.4.2 16 位 Windows 环境 | 699 |
| 21.7.3 日志 | 661 | 22.4.3 IA64 上的 32 位 Windows 环境 | 699 |
| 21.7.4 Linux 进程文件系统 | 662 | 22.4.4 Win32 环境 | 699 |
| 21.8 输入与输出 | 663 | 22.4.5 POSIX 子系统 | 700 |
| 21.8.1 块设备 | 664 | 22.4.6 登录与安全子系统 | 700 |
| 21.8.2 字符设备 | 665 | 22.5 文件系统 | 701 |
| 21.9 进程间通信 | 665 | 22.5.1 NTFS 内部布局 | 701 |
| 21.9.1 同步与信号 | 665 | | |

| | | |
|--------|----------------|-----|
| 22.5.2 | 恢复 | 703 |
| 22.5.3 | 安全 | 704 |
| 22.5.4 | 卷管理和容错 | 704 |
| 22.5.5 | 压缩与加密 | 707 |
| 22.5.6 | 安装点 | 708 |
| 22.5.7 | 变更日志 | 708 |
| 22.5.8 | 卷影子副本 | 708 |
| 22.6 | 网络 | 708 |
| 22.6.1 | 网络接口 | 709 |
| 22.6.2 | 协议 | 709 |
| 22.6.3 | 分布式处理机制 | 710 |
| 22.6.4 | 重定向器与服务器 | 712 |
| 22.6.5 | 域 | 713 |
| 22.6.6 | 活动目录 | 714 |
| 22.6.7 | TCP/IP 网络的名称解析 | 714 |
| 22.7 | 程序接口 | 715 |
| 22.7.1 | 访问内核对象 | 715 |
| 22.7.2 | 进程间共享对象 | 715 |
| 22.7.3 | 进程管理 | 717 |
| 22.7.4 | 进程间通信 | 719 |
| 22.7.5 | 内存管理 | 720 |
| 22.8 | 小结 | 722 |

| | |
|-----------------|-----|
| 习题 | 722 |
| 文献注记 | 723 |
| 第 23 章 有影响的操作系统 | 724 |
| 23.1 早期系统 | 724 |
| 23.1.1 专用计算机系统 | 724 |
| 23.1.2 共享计算机系统 | 725 |
| 23.1.3 I/O 叠加 | 728 |
| 23.2 Atlas | 730 |
| 23.3 XDS-940 | 731 |
| 23.4 THE | 731 |
| 23.5 RC4000 | 732 |
| 23.6 CTSS | 733 |
| 23.7 MULTICS | 733 |
| 23.8 IBM OS/360 | 734 |
| 23.9 Mach | 735 |
| 23.10 其他系统 | 737 |
| 习题 | 737 |
| 参考文献 | 738 |
| 原版相关内容引用表 | 763 |
| 英汉名词对照表 | 764 |

第一部分 概述

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

操作系统是管理计算机硬件的软件。硬件必须提供合适的机制来保证计算机系统的正确运行，以及确保系统不受用户程序干扰正常运行。

根据操作系统不同的组织方式，它们内部各不相同。设计一个新的操作系统是主要的任务。在设计开始之前明确所设计系统的目标是非常重要的。这些目标构成了选择不同算法和策略的基础。

因为操作系统庞大而复杂，因此它必须被分块构造。每一块都是系统中明确定义的一部分，具有严格定义的输入、输出和功能。

第1章 导 论

操作系统是管理计算机硬件的程序，它还为应用程序提供基础，并且充当计算机硬件和计算机用户的中介。令人惊奇的是操作系统完成这些任务的方式多种多样。大型机的操作系统设计的主要目的是为了充分优化硬件的使用率，个人计算机的操作系统是为了能支持从复杂游戏到商业应用的各种事物，手持计算机的操作系统是为了给用户提供一个可以与计算机方便地交互并执行程序的环境。因此，有的操作系统设计是为了方便，有的设计是为了高效，而有的设计目标则是兼而有之。

在研究计算机操作系统的细节之前，首先需要了解系统结构的知识。本章从讨论系统启动、I/O 和存储的基本功能开始，并讨论能编写一个可用操作系统的基本计算机体系。

由于操作系统非常庞大且复杂，必须逐个部分地生成。每一部分都必须是构造好的系统的一部分，并严格定义了输入、输出和功能。本章提供了操作系统主要部件梗概。

本章目标

- 提供对操作系统主要部件的浏览。
- 提供基本的计算机系统体系结构的概述。

1.1 操作系统做什么

本章通过了解操作系统在计算机系统中所扮的角色开始讨论。操作系统是几乎所有计算机系统的一个重要部分。计算机系统可以大致分为 4 个组成部分：*计算机硬件、操作系统、系统程序与应用程序*和*用户*（见图 1.1）。

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

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

为了更加全面地理解操作系统所担当的角色，接下来从两个视角探索操作系统：即从

用户的视角和系统的视角来研究。

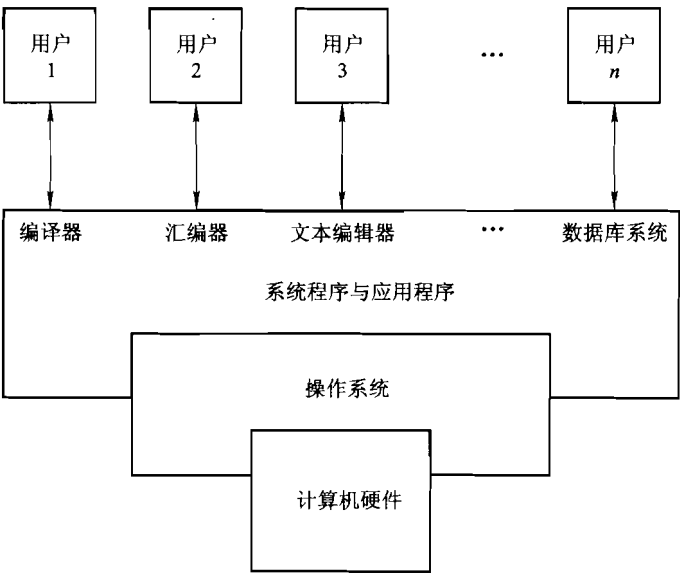


图 1.1 计算机系统组成部分的逻辑图

1.1.1 用户视角

计算机的用户观点因所使用接口的不同而异。绝大多数计算机用户坐在一台这样的 PC 前，PC 由显示器、键盘、鼠标和主机组成。这类系统设计是为了让单个用户单独使用其资源，其目的是优化用户所进行的工作（或游戏）。对于这种情况，操作系统的设计目的是为了用户使用方便，性能是次要的，而且不在乎资源使用率——如何共享硬件和软件资源。性能对用户来说非常重要，而不是资源使用率，这种系统主要为了优化单用户的情况。

在某些情况下，有些用户坐在与大型机或小型机相连的终端前，其他用户通过其他的终端访问同一计算机。这些用户共享资源并可交换信息。操作系统设计为资源使用做了优化：确保所有的 CPU 时间、内存和 I/O 都能得到充分使用，并且确保没有用户使用超出其权限以外的资源。

在另一些情况下，其他用户坐在工作站前，工作站与其他工作站和服务器相连。这些用户不但可以使用专用的资源，而且可以使用共享资源，如网络和服务器及文件、计算和打印服务器。因此，这类操作系统的设计目的是个人使用性能和资源利用率的折中。

近来，各种手持计算机开始成为时尚。绝大多数这些设备为单个用户所独立使用。有的也通过有线或（更为常见）无线与网络相连。由于受电源、速度和接口所限，它们只能执行相对较少的远程操作。绝大多数这类操作系统的设计目的是为了个人使用，当然

如何在有限的电池容量中发挥最大的效用也很重要。

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

1.1.2 系统视角

从计算机的角度来看，操作系统是与硬件最为密切的程序。本节中，可以将操作系统看做资源分配器。计算机系统可能有許多资源，用来解决 CPU 时间、内存空间、文件存储空间、I/O 设备等问题。操作系统管理这些资源。面对许多甚至冲突的资源请求，操作系统必须决定如何为各个程序和用户分配资源，以便计算机系统能有效而公平地运行。众所周知，资源分配对多用户访问主机或微型计算机特别重要。

操作系统的一个稍稍不同的观点是强调控制各种 I/O 设备和用户程序的需要。操作系统是控制程序。控制程序管理用户程序的执行以防止计算机资源的错误使用或使用不当。它特别关注 I/O 设备的操作和控制。

1.1.3 定义操作系统

读者已经从用户的视角和系统的视角了解了操作系统，但是，可以定义操作系统是什么吗？一般来说，目前没有一个关于操作系统的十分完整的定义。操作系统之所以存在，是因为它们提供了解决创建可用的计算机系统问题的合理途径。计算机系统的基本目的是执行用户程序并能更容易地解决用户问题。为实现这一目的，构造了计算机硬件。由于仅仅有硬件并不一定容易使用，因此开发了应用程序。这些应用程序需要一些共同操作，如控制 I/O 设备。这些共同的控制和分配 I/O 设备资源的功能集合组成了一个软件模块：操作系统。

另外，也没有一个广泛接受的究竟什么属于操作系统的定义。一种简单观点是操作系统包括当你预定一个“操作系统”时零售商所装的所有东西。当然，包括的特性随系统不同而变化很大。有的系统只有不到 1 MB 的空间甚至没有全屏编辑器，而其他系统则需要数百 MB 空间并且完全采用图形窗口系统（1 KB = 1 024 B，1 MB = 1 024² B，1 GB = 1 024³ B；但是计算机制造商通常认为 1 MB=10⁶ B，1 GB=10⁹ B）。一个比较公认的定义是，操作系统是一直运行在计算机上的程序（通常称为内核），其他程序则为系统程序和应用程序。这一定义是人们通常所采用的。

现在，什么组成了操作系统这个问题变得越来越重要了。1998 年，美国司法部控告微软公司将过多的功能加到操作系统中，因此妨碍了其他应用程序开发者的公平竞争。例如，将 Web 浏览器作为操作系统的一个整体部分，结果，微软公司因独占使用其操作系统以限制竞争受到处罚。

1.2 计算机系统组织

在研究计算机系统如何操作的细节之前，需要对计算机系统的结构有一个全面的了解。本章将研究这一结构的若干方面以复习背景知识。本章主要讨论计算机的系统结构，如果您已经理解这些概念，那么就可以浏览或跳过本章。

1.2.1 计算机系统操作

现代通用计算机系统由一个或多个 CPU 和若干设备控制器通过共同的总线相连而成，该总线提供了对共享内存的访问（见图 1.2）。每个设备控制器负责一种特定类型的设备（如磁盘驱动器、音频设备、视频显示器）。CPU 与设备控制器可以并发工作，并竞争内存周期。为了确保对共享内存的有序访问，需要内存控制器来协调对内存的访问。

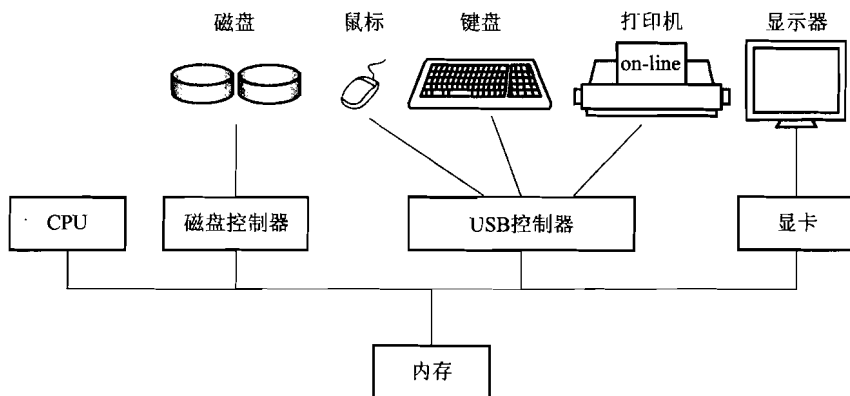


图 1.2 现代计算机系统

当打开电源或重启时，计算机开始运行，它需要运行一个初始化程序。该初始化程序或**引导程序**（bootstrap program）比较简单，通常位于 ROM 或 EEPROM 中，称为计算机硬件中的固件。它初始化系统中的所有部分，包括 CPU 寄存器、设备控制器和内存内容。引导程序必须知道如何装入操作系统并开始执行系统。为了完成这一目标，引导程序必须定位操作系统内核并把它装入内存。接着，操作系统开始执行第一个进程如 `init`，并等待事件的发生。

事件的发生通常通过硬件或软件**中断**（interrupt）来表示。硬件可随时通过系统总线向 CPU 发出信号，以触发中断。软件通过执行特别操作如**系统调用**（system call）（也称为**监视器调用**（monitor call））也能触发中断。

当 CPU 中断时，它暂停正在做的事并立即转到固定的位置去继续执行。该固定位置通

常是中断服务程序开始位置的地址。中断服务程序开始执行，在执行完后，CPU 重新执行被中断的计算。这一操作的时间线路如图 1.3 所示。

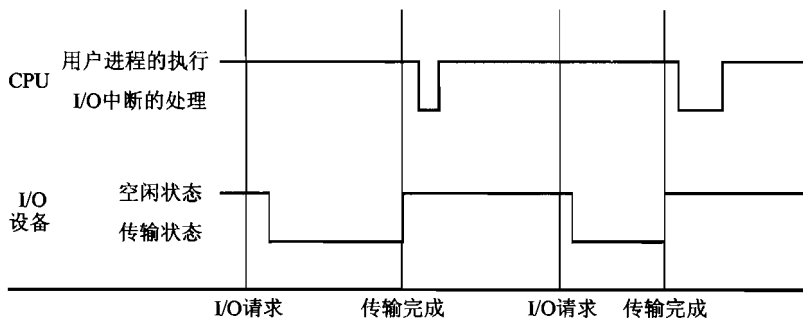


图 1.3 单个进程执行输出的中断时间线路

中断是计算机结构的重要部分。每个计算机设计都有自己的中断机制，但是有些功能是共同的。中断必须将控制转移到合适的中断处理程序。处理转移的简单方法是调用一个通用子程序以检查中断信息。接着，该子程序会调用相应的中断处理程序。不过，处理中断要快，由于只有少量的预先定义的中断，所以可使用中断处理子程序的指针表。这样通过指针表可间接调用中断处理子程序，而不需要通过其他中间子程序。通常，指针表位于低地址内存（前 100 左右的位置）。这些位置包含各种设备的中断处理子程序的地址。这种地址的数组或中断向量（interrupt vector）可通过唯一设备号来索引（对于给定的中断请求），以提供设备的中断处理子程序的地址。许多操作系统如 Windows 或 UNIX 都采用这种方式来处理中断。

中断体系结构也保存被中断指令的地址。许多旧的设计简单地在固定位置中（或在可用设备号来索引的地址中）保存中断地址。更为现代的结构将返回系统堆栈中的地址。如果中断处理程序需要修改处理器状态，如修改寄存器的值，它必须明确地保存当前状态并在返回之前恢复该状态。在处理中断之后，保存的返回地址会装入程序计数器，被中断的计算可以重新开始，就好像中断没有发生过一样。

1.2.2 存储结构

计算机程序必须在内存（或随机访问内存（random access memory, **RAM**））中以便于运行。内存是处理器可以直接访问的唯一的容量存储区域（数兆到数千兆字节）。它通常是用被称为动态随机访问内存（dynamic random access memory, **DRAM**）的半导体技术来实现的，是一组内存字的数组，每个字都有其地址。通过对特定内存地址执行一系列 load 或 store 指令来实现交互。指令 load 能将内存中的字移到 CPU 的寄存器中，而指令 store

能将寄存器的内容移到内存。除了显式使用 `load` 和 `store` 外, CPU 可自动从内存中装入指令来执行。

一个典型指令执行周期(在冯·诺依曼体系结构上执行时)首先从内存中获取指令,并保存在**指令寄存器(instruction register)**中。接着,指令被解码,并可能导致从内存中获取操作数或将操作数保存在内部寄存器中。在指令完成对操作数的执行后,其结果可以存回到内存。注意内存单元只看见内存地址流,它并不知道它们是如何产生的(通过指令计数器、索引、间接、常量地址等),或它们是什么地址(指令或数据)。相应地,可忽视程序如何产生内存地址,只对程序运行所生成的地址序列感兴趣。

理想情况下,程序和数据都永久地驻留在内存中。由于如下原因,这是不可能的:

- ① 内存太小,不能永久地存储所有需要的程序和数据。
- ② 内存是**易失性**存储设备,当掉电时会失去所有内容。

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

最为常用的辅存设备为**磁盘(magnetic disk)**,它能存储程序和数据。绝大多数程序(网页浏览器、编译器、字处理器、电子制表软件等)保存在磁盘上,直到要执行时才装入到内存。许多程序都使用磁盘作为它们所处理信息的来源和目的。因此,适当的管理磁盘存储对计算机系统来说十分重要,这将在第 12 章中加以讨论。

上面描述的存储结构由寄存器、内存和磁盘组成,这些仅仅是一种存储系统。除此之外,还有高速缓存、CD-ROM、磁带等。每个存储系统都提供了基本功能以存储数据,或保存数据以便日后提取。各种存储系统的主要差别是速度、价格、大小和易失性。

根据速度和价格,可以按层次结构来组织计算机系统的不同类型的存储系统(图 1.4)。层次越高,价格越贵,但是速度越快。随着层次降低,单个位的价格通常降低,而访问时间通常增加。这种折中是合理的:如果一个给定的存储系统比另一个更快更便宜,而其他属性一样,那么就没有理由再使用更慢更昂贵的存储器。事实上,许多早期存储设备,如纸带和磁芯存储器,之所以现在已经送进博物馆,就是因为磁带和半导体内存已变得更快更便宜。图 1.4 中的上面四层存储通常采用半导体内存技术。

除了不同的速度和价格,存储系统还分为易失的和非易失的。当没有电源时,正如前面所讲,**易失存储(volatile storage)**会丢失其中的内容。如果没有昂贵的电池和发电机后备系统,那么数据必须写到**非易失存储(nonvolatile storage)**中以便保护。在图 1.4 所示的层次结构中,**电子磁盘(electronic disk)**之上的存储系统为易失的,而之下的为非易失的。电子磁盘可以被设计为易失的或者非易失的。在普通操作情况下,电子磁盘将数据保存在一个大的 **DRAM** 数组上,这是易失的。但是,许多电子磁盘设备都有一个隐藏的磁盘和电池作为备份电源。当外部电源发生中断时,电子磁盘控制器将数据从 **RAM** 复制到磁盘。当外部电源恢复时,控制器将数据复制回 **RAM** 中。另一种电子磁盘是闪存,它在照相机、

PDA 和机器人中使用得很广泛，并越来越多地作为通用计算机上的可移动的存储设备。闪存比 DRAM 慢，但不需要电源来保存它的内容。另一种非易失存储器是 **NVRAM**，即具有备用电池的 DRAM。这种存储可以像 DRAM 那样快，但其存储时限是有时间限制的。

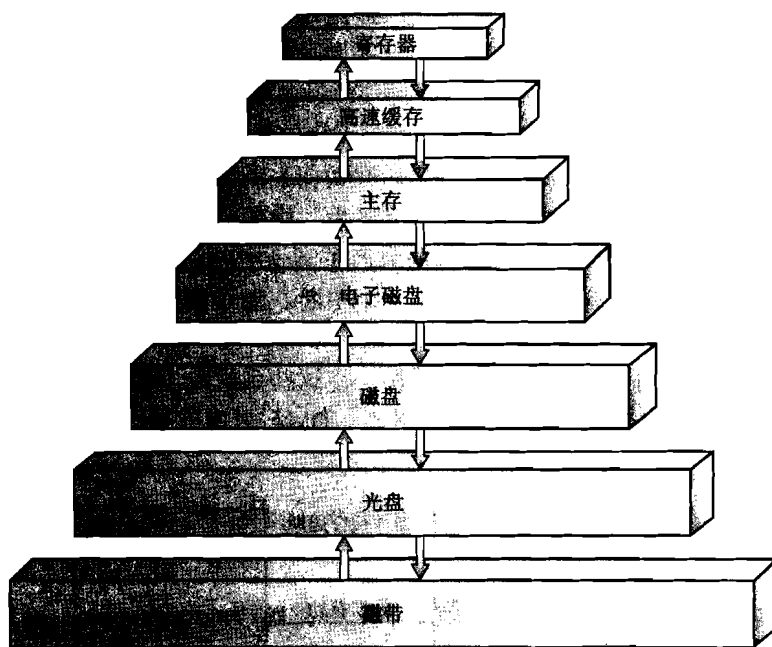


图 1.4 存储设备层次

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

1.2.3 I/O 结构

在计算机中，存储器只是众多 I/O 设备中的一种，操作系统的大部分代码用来进行 I/O 管理，这既是因为它对系统可靠性和性能的十分重要，也是因为设备变化的特性。因此，在此仅讨论 I/O 的概况。

通用计算机系统由一个 CPU 和多个设备控制器组成，它们通过共同的总线连接起来。每个设备控制器负责特定类型的设备，可有多个设备与其相连。例如，**SCSI** (small computer system interface) 控制器可有 7 个或更多的设备与之相连。设备控制器维护一定量的本地缓冲存储和一组特定用途的寄存器。设备控制器负责在其所控制的外部设备与本地缓冲存储之间进行数据传递。通常，操作系统为每个设备控制器提供一个设备驱动程序。这些设

备驱动程序理解设备控制器，并提供一个设备与其余操作系统的统一接口。

为了开始 I/O 操作，设备驱动程序在设备控制器中装载适当的寄存器。相应地，设备控制器检查这些寄存器的内容以决定采取什么操作（如从键盘中读取一个字符）。控制器开始从设备向其本地缓冲区传输数据。一旦完成数据传输，设备控制器就会通过中断通知设备驱动程序它已完成操作。然后，设备驱动程序返回对操作系统的控制，如果是一个读操作，可能将数据或数据的指针返回。对其他操作，设备驱动程序返回状态信息。

这种 I/O 中断驱动适合移动少量数据，但对大块的数据移动，如磁盘 I/O，就会带来超载问题。**DMA**（direct memory access，直接内存访问）就是为了解决这个问题而设计的。在为这种 I/O 设备设置好缓冲、指针和计数器之后，设备控制器能在本地缓冲和内存之间传送一整块数据，而无需 CPU 的干预。每块只产生一个中断，来告知设备驱动程序操作已完成，而不是像低速设备那样每个字节产生一个中断。当设备控制器在执行这些操作时，CPU 可去完成其他工作。

一些高端的系统采用交换而不是总线结构。在这些系统中，多个部件可以与其他部件并发对话，而不是在公共总线上争夺周期。此时，DMA 更为有效。图 1.5 表示了计算机系统所有部件的互相作用。

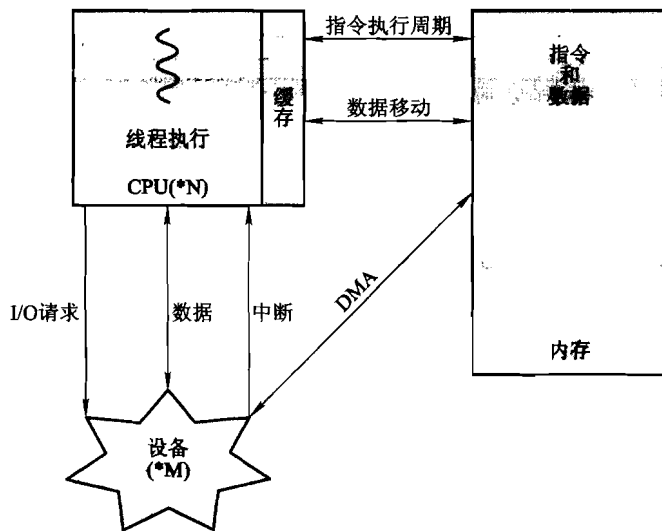


图 1.5 现代计算机系统工作模式

1.3 计算机系统体系结构

1.2 小节介绍了一个典型的计算机系统的通用结构。计算机系统可能通过许多不同的

途径组织，可以大致通过其采用的通用处理器的数量来分类。

1.3.1 单处理器系统

绝大多数系统采用单处理器。单处理器系统的种类可能令人惊讶，从 PDA 到大型机都有。在单处理器系统中，有一个主 CPU 能够执行一个通用指令集，包括来自于用户进程的指令。绝大多数系统还包括其他特定目的的处理器，它们可能以专用设备处理器（比如磁盘、键盘、图形控制器）的形式出现；在大型机上，它们可能以通用处理器的形式出现，比如在系统部件间快速移动数据的 I/O 处理器。

所有这些专用处理器运行一个受限的指令集，并不运行用户进程。有时它们由操作系统管理，此时操作系统将接下来的任务信息发给它们，并监控它们的状态。例如，磁盘控制器微处理器接收来自于主 CPU 的一系列请求，执行它们自己的磁盘队列和调度算法。这种安排克服了主 CPU 的磁盘调度超载问题。PC 在其键盘上用一个小微处理器来将击键转换为代码，并发送给 CPU。在其他系统或环境中，专用处理器被构建成硬件的低级部件。操作系统不能与这些部件通信，后者独立地做自己的工作。使用专用处理器很常见，并不会将一个单处理器系统变成多处理器系统。如果只有一个通用 CPU，系统则为单处理器系统。

1.3.2 多处理器系统

虽然绝大多数系统都属于单处理器系统，多处理器系统（也称为并行系统（parallel system）或紧耦合系统（tightly coupled system））的重要性也日益突出。这类系统有多个紧密通信的 CPU，它们共享计算机总线，有时还有时钟、内存和外设等。

多处理器系统有三个主要优点：

① **增加吞吐量**：通过增加处理器的数量，希望能在更短的时间内做更多的事情。用 N 个处理器的加速比不是 N ，而是比 N 小。当多个 CPU 在同一件事情上时，为了使得各部分能正确工作，会产生一定的额外开销。这些开销，加上对共享资源的竞争，会降低因增加了 CPU 的期望增益。这与一组 N 位程序员在一起紧密地工作，并不能完成 N 倍的单个程序员的工作量类似。

② **规模经济**：多处理器系统比单个处理器系统能节省资金，这是因为它们能共享外设、大容量存储和电源供给。当多个程序需要操作同样的数据集时，如果将这些数据放在同一磁盘上并让多处理器共享，将比用许多有本地磁盘的计算机和多个数据复制更为节省。

③ **增加可靠性**：如果将功能分布在多个处理器上，那么单个处理器的失灵将不会使得整个系统停止，只会使它变慢。如果有 10 个处理器而其中一个出了故障，那么剩下的 9 个会分担起故障处理器的那部分工作。因此，整个系统只是比原来慢了 10%，而不是停止运行。

在许多应用中，计算机系统不断增加的可靠性是很关键的。这种能提供与正常工作的硬件成正比的的服务的能力被称为**适度退化**（graceful degradation），有些系统超出适度退化的能力被称为**容错**（fault tolerant），因为它们能忍受单个部件的错误并继续操作。容错需要一定的机制来对故障进行检测、诊断和纠错（如果可能）。HP NonStop 系统（即先前的 Tandem）通过使用冗余的硬件和软件来确保在故障时也能继续工作。该系统具有多对 CPU，它们同步工作。每一对处理器都各自执行自己的指令并比较结果。如果结果不一样，则其中一个 CPU 出错，此时两个皆停止。然后执行的进程被送到另一对 CPU 中，刚才出错的指令重新开始执行。这种方法比较昂贵，因为它用到了专用硬件和相当多的冗余硬件。

现在使用的多处理器系统主要有两种类型。有的系统使用**非对称多处理**（asymmetric multiprocessing），即每个处理器都有各自特定的任务。一个主处理器控制系统，其他处理器或者向主处理器要任务或做预先定义的任务。这种方案称为主—从关系。主处理器调度从处理器并安排工作。

现在最为普遍的多处理器系统使用**对称多处理**（symmetric multiprocessing, SMP），每个处理器都要完成操作系统中的所有任务。SMP 意味着所有处理器对等，处理器之间没有主—从关系。图 1.6 显示了一个典型的 SMP 结构。一个典型的 SMP 例子是 Solaris，一个由 Sun Microsystems 设计的商用 UNIX。一个 Solaris 系统可配置成使用数十个处理器，并且都运行 Solaris。这种模型的好处是如果有 N 个 CPU，那么 N 个进程可以同时运行且并不影响性能。然而，必须仔细控制 I/O 以确保数据到达合适的处理器。另外，由于各 CPU 互相独立，一个可能空闲而另一个可能过载，导致效率低。如果处理器共享一定的数据结构，那么可以避免这种低效率。这种形式的多处理器允许进程和资源（包括内存）在各处理器之间动态共享，能够降低处理器之间的差异。这样的系统需要仔细设计，如第 6 章所述。目前几乎所有现代操作系统，包括 Windows、Windows XP、Mac OS X 和 Linux 等，都支持 SMP。

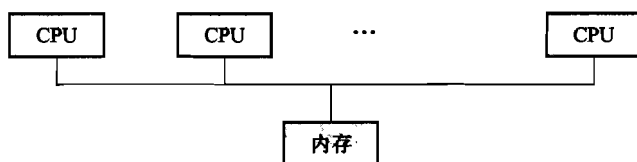


图 1.6 对称多处理体系结构

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

最新的 CPU 设计趋势是将多个计算机**内核 (core)** 设计到单个芯片上。它们实际上是多处理器芯片。现在双芯片正在成为主流, 而 N 芯片则在高级系统中越来越常用。除考虑体系结构, 如缓存、内存及总线竞争外, 这些多核 CPU 对操作系统而言就像是 N 个标准处理器。

最后, 最近开发的**刀片服务器 (blade server)** 将多处理器板、I/O 板和网络板全部置于同一底板上。它和传统多处理器系统的不同在于, 每个刀片处理器独立启动并运行各自的操作系统。有些刀片服务器板也是多处理器的, 从而模糊了计算机类型的划分。实际上, 这些服务器包括了多个独立的多处理器系统。

1.3.3 集群系统

多 CPU 系统的另一种类型是**集群系统 (clustered system)**。与多处理器系统一样, 集群系统将多个 CPU 集中起来完成计算任务。然而, 集群系统与多处理器系统不同, 它是由两个或多个独立的系统耦合起来的。**集群**的定义目前尚未定形, 许多商业软件对什么是集群系统及为什么一种形式的集群比另一种好有不同的理解。较为常用的定义是集群计算机共享存储并通过局域网络连接 (如 1.10 节所述) 或更快的内部连接 (如 InfiniBand)。

集群通常用来提供**高可用性 (high availability)** 服务, 这意味着即使集群中的一个或多个系统出错, 服务仍然继续。高可用性通常通过在系统中增加一定的冗余来获取。集群软件运行在集群节点之上, 每个都能监视 (通过局域网) 一个或多个其他节点。如果被监视的机器失效, 那么监视机器能取代存储拥有权, 并重新启动在失效机器上运行的应用程序。应用程序的用户和客户机只感觉到很短暂的中断。

集群可以是对称的, 也可以是非对称的。**非对称集群 (asymmetric clustering)** 中, 一台机器处于**热备份模式 (hot standby mode)**, 而另一台运行应用程序。热备份主机 (机器) 只监视活动服务器。如果该服务器失效, 那么热备份主机会成为现行服务器。对于**对称集群 (symmetric clustering)**, 两个或多个主机都运行应用程序, 它们互相监视。这种模式因为充分使用了现有硬件, 所以更为高效。这要求具有多个应用程序可供运行。

其他形式的集群有并行集群和 WAN 集群 (如 1.10 小节所述)。并行集群允许多个主机访问共享存储上的相同数据。由于绝大多数操作系统不支持多个主机同时访问数据, 并行集群通常需要由专门软件 and 应用程序来完成。例如, Oracle Parallel Server 是一种可运行在并行集群上的 Oracle 数据库版本。每个机器都运行 Oracle, 且有软件跟踪共享磁盘的访问。每个机器对数据库内的所有数据都可以完全访问。为了提供这种对数据的共享访问, 系统必须提供对文件的访问控制和加锁, 以确保不出现冲突操作。有些集群技术中包括了这种通常称为**分布式锁管理器 (distributed lock manager, DLM)** 的服务。

集群技术发展迅速。有些集群产品支持几十个系统, 即使集群中的节点之间间隔几英里。其中许多发展可能随着 **SAN (storage-area network)** 的流行而进一步扩大, 关于 SAN,

请参见 12.3.3 小节，它允许很多系统附有存储池。如果应用程序和数据存储在 SAN 中，集群软件可以分配应用程序在任何附着在 SAN 上的主机上运行。如果主机出错，可以用其他主机取代。在数据库集群中，数十个主机可以共享相同的数据库，从而大大地提升了性能和可用性。

1.4 操作系统结构

前面已经讨论了计算机系统的组织和体系，现在将要讨论操作系统了。操作系统提供执行程序的环境。从内部讲，操作系统的组成变化非常大，因为它们通过许多不同的线路组织。但也有许多共有特点，本节将会涉及。

操作系统最重要的一点是要有多道程序处理能力。单个用户通常不能总是使得 CPU 和 I/O 设备都忙。多道程序设计通过组织作业（编码或数据）使 CPU 总有一个作业可执行，从而提高了 CPU 的利用率。

这种思想如下：操作系统同时将多个任务保存在内存中（见图 1.7）。该作业集可以是作业池中作业集的子集（作业池中包括所有进入系统的作业），这是因为可同时保存在内存中的作业数要比可在作业池中的作业数少。操作系统选择一个位于内存中的作业并开始执行。最终，该作业可能必须等待另一个任务如 I/O 操作的完成。对于非多道程序系统，CPU 就会空闲；对于多道程序系统，CPU 会简单地切换到另一个作业并执行。当该作业需要等待时，CPU 会切换到另一个作业。最后，第一个作业完成等待且重新获得 CPU。只要有一个任务可以执行，CPU 就决不会空闲。

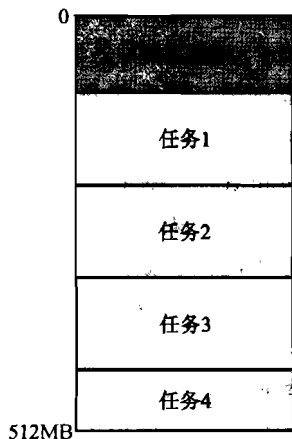


图 1.7 多道程序系统的内存分布

这种思想在日常生活中也常见。例如，一个律师在一段时间内不只为一个客户工作。

当一个案件需要等待审判或需要准备文件时，该律师可以处理另一个案件。如果有足够多的客户，那么他就决不会因没有工作要做而空闲（空闲的律师会成为政客，因此让律师忙碌有积极的社会意义）。

多道程序系统提供了一个可以充分使用各种系统资源（如 CPU、内存、外设）的环境，但是它们没有提供与计算机系统直接交互的能力。**分时系统（或多任务）**是多道程序设计的延伸。在分时系统中，虽然 CPU 还是通过在作业之间的切换来执行多个作业，但是由于切换频率很高，用户可以在程序运行期间与之进行交互。

共享需要一种**交互计算机系统**，它能提供用户与系统之间的直接通信。用户通过输入设备，如键盘或鼠标，向操作系统或程序直接发出指令，并等待输出设备立即出来的结果。相应地，**响应时间（response time）**应该比较短，通常小于 1 秒。

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

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

分时和多道程序设计需要在存储器中同时保存有几个作业。通常由于主存较小而不能容纳太多作业，所以这些作业刚开始存储在磁盘的**作业池（job pool）**中。该池由所有驻留在磁盘中需要等待分配内存的作业组成。如果多个作业需要调入内存但没有足够的内存，那么系统必须在这些作业中做出选择，这样的决策被称为**作业调度（job scheduling）**，这将在第 5 章介绍。当操作系统从作业池中选中一个作业，就将它调入内存来执行。在内存中同时有多个程序可运行，需要一定形式的内存管理，这将在第 8 章和第 9 章讨论。另外，如果有多个任务同时需要执行，那么系统必须做出选择，这样的选择称为**CPU 调度（CPU scheduling）**，这将在第 5 章讨论。最后，多个并发执行的作业需要操作系统在各方面限制进程的互相影响，如进程调度、磁盘存储和内存管理，这些将贯穿本书。

在分时操作系统中，操作系统必须保证合理的响应时间，这有时需要通过交换来得到。交换时进程被换入内存或由内存换出到磁盘。实现这一目的更常用方法是使用**虚拟内存（virtual memory）**，虚拟内存允许将一个执行的作业不完全放在内存中（第 9 章）。虚拟内存的主要优点是程序可以比**物理内存（physical memory）**大。再者，它将内存抽象成一个庞大且统一的存储数组，将用户所理解的**逻辑内存（logical memory）**与真正的物理内存区

分开来。这种安排使得程序员不必为内存空间的限制而担心。

分时操作系统也必须提供文件系统（参见第 10 章和第 11 章）。文件系统驻留在一组磁盘上，因此也必须提供磁盘管理（参见第 12 章）。另外，分时系统要提供一种保护资源以防不当使用的机制（参见第 14 章）。为了确保有序执行，系统必须提供实现作业同步和通信（参见第 6 章）的机制，它也要确保作业不会进入死锁，进而无尽地互相等待（参见第 7 章）。

1.5 操作系统操作

如前所述，现代操作系统是由中断驱动的。如果没有进程要执行，没有 I/O 设备要服务，也没有用户请求要响应，操作系统将会静静地等待某件事件的发生。事件总是由中断或陷阱引起。陷阱（或异常）是一种软件中断，源于出错（如除数为零或无效的存储访问），或源于用户程序的一个特别请求（完成操作系统服务）。这种操作系统的中断特性定义了系统的通用结构。对每一种中断，操作系统中不同的代码段决定了将要采取的动作。中断服务程序被用来处理中断。

由于操作系统和用户共享了计算机系统的硬件和软件，必须保证用户程序中的一个出错仅影响正在运行的程序。采用共享，许多进程可能会受到一个程序中的一个漏洞（bug）的不利影响。例如，如果一个进程陷入死循环，那么这个死循环可能会阻止很多其他进程的正确操作。在多道程序设计中可能会发生更为微妙的错误，如一个错误的程序可能修改另一个程序、另一程序的数据，甚至操作系统本身。

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

1.5.1 双重模式操作

为了确保操作系统的正常执行，必须区分操作系统代码和用户定义代码的执行。许多操作系统所采取的方法是提供硬件支持以允许区分各种执行模式。

至少需要两种独立的操作模式：用户模式（user mode）和监督程序模式（monitor mode）（也称为管理模式（supervisor mode）、系统模式（system mode）或特权模式（privileged mode））。在计算机硬件中增加一个称为模式位（mode bit）的位以表示当前模式：监督程序模式（0）和用户模式（1）。有了模式位，就可区分为操作系统所执行的任务和为用户所执行的任务。当计算机系统表示用户应用程序正在执行，系统处于用户模式。然而，当用户应用程序需要操作系统的服务（通过系统调用），它必须从用户模式转换过来执行请求，如图 1.8 所示。正如所将看到的，这种结构改进对于许多系统操作都很有用。

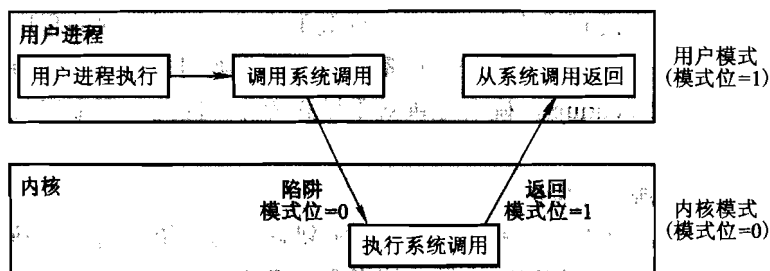


图 1.8 用户模式到内核模式的转换

系统引导时，硬件开始处于内核模式。接着，装入操作系统，开始在用户模式下执行用户进程。一旦出现陷阱或中断，硬件会从用户模式切换到内核模式（即将模式位设为 0）。因此，只要操作系统获得了对计算机的控制，它就处于内核模式。系统在将控制交还给用户程序时会切换到用户模式（将模式位设为 1）。

双重模式操作提供了保护操作系统和用户程序不受错误用户程序影响的手段。其实现方法为：将能引起损害的机器指令作为**特权指令**（privileged instruction）。如果在用户模式下试图执行特权指令，那么硬件并不执行该指令，而是认为该指令非法，并将其以陷阱的形式通知操作系统。

转换到用户模式就是一个特权指令，其他的例子包括 I/O 控制、定时器管理和中断管理。在本书中，我们还将看到许多其他的特权指令。

现在可以了解一下计算机系统中的指令执行的生命周期问题。最初的控制发生在操作系统中，在此指令以内核模式来执行。当控制权转到一个用户应用程序后，模式变为用户模式。最后，通过中断、陷阱或系统调用将控制权返回给操作系统。

系统调用为用户程序请求操作系统代表用户程序完成预留给操作系统的任务提供了方法。系统调用可以采用多种途径，具体采用哪种途径取决于由下层处理器提供的功能。不管哪种途径，它都是一种进程请求操作系统执行动作的方法。系统调用通常采用陷阱到中断向量中的一个指定位置的方式。该陷阱可以由普通 trap 指令来执行，尽管有些系统（如 MIPS R2000 系列）具有专门的 syscall 指令。

当系统调用被执行时，硬件会将它作为软件中断。控制权会通过中断向量转交到操作系统的中断处理程序，模式位设置成内核模式。系统调用服务程序是操作系统的一部分。内核检查中断指令以确定发生了什么系统调用；参数表示用户程序请求什么类型的服务。请求所需要的其他信息可通过寄存器、堆栈或内存（内存的指针可传递给寄存器）来传递。内核检验参数是否正确和合法，再执行请求，然后将控制返回到系统调用之后的指令。2.3 小节将详细介绍系统调用。

缺乏硬件支持的双重模式会在操作系统内产生一些缺点。例如，MS-DOS 是为 Intel

8088 体系结构而编写，它没有模式位，因而没有双重模式。运行错误的程序可通过写数据来清除整个操作系统，多个程序可同时对设备进行写操作则可能引起灾难性的结果。Intel CPU 的最近的版本，如 Pentium，确实提供双重模式操作。因此，更多现代操作系统，如 Microsoft Windows 2000、Windows XP、Linux 和 Solaris 的 x86 系统，都利用了这一特征，并为操作系统提供了更强大的保护。

一旦硬件保护到位，硬件可检测到违反模式的错误。这些错误通常由操作系统处理。如果一个用户程序出现失败，如试图执行非法指令或者访问不属于自己地址空间的内存，那么硬件会向操作系统发出陷阱信号。陷阱如同中断一样，能通过中断向量将控制转交给操作系统。只要一个程序出现错误，操作系统就必须对它进行异常终止。这种情况的处理代码与用户请求的异常终止的处理代码一样，会给出一个适当的出错信息，程序内存会被转储。内存信息转储通常写到文件以便用户或程序员能检查它，纠正错误，并重新启动程序。

1.5.2 定时器

必须确保操作系统能维持对 CPU 的控制，也必须防止用户程序陷入死循环或不调用系统服务，并且不将控制权返回到操作系统。为了实现这一目标，可使用**定时器**（timer）。可将定时器设置为在给定时间后中断计算机，时间段可以是固定的（例如 1/60 s）或可变的（例如，1 ms~1 s）。**可变定时器**（variable timer）一般通过一个固定速率的时钟和计数器来实现。操作系统设置计数器。每经过一个时钟周期，计数器都要递减。当计数器的值为 0 时，产生中断。例如，对于 10 位的计数器和 1 ms 精度的时钟，可允许在 1~1 024 ms 的时间间隔内产生中断，时间步长为 1 ms。

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

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

1.6 进 程 管 理

程序在未被 CPU 执行之前不会做任何事。如前面提到过的，处于执行中的程序被称为进程。分时用户程序，如编译器，就是一个进程。由 PC 上的个人用户所运行的字处理程序是一个进程。系统任务，如将输出发送到打印机也可以是一个进程（或至少是其中的一

部分)。现在, 可以将进程视为作业或分时程序, 但以后, 进程的概念将更为广泛。正如将在第 3 章所要学习的, 将提供允许进程创建子进程以并发执行的系统调用。

进程需要一定的资源 (包括 CPU 时间、内存、文件、I/O 设备) 以完成其任务。这些资源可以在进程创建时分配给进程, 也可以在执行进程时分配给进程。除了在创建时得到各种物理和逻辑资源外, 进程还可以接受传输过来的各种初始化数据 (输入)。例如, 考虑这样一个进程, 它的功能是在终端或者屏幕上显示文件状态。该进程会得到一个文件名作为输入, 并且执行适当的指令和系统调用以得到和显示终端所需的信息。当进程中止时, 操作系统将收回所有可再用的资源。

需要强调的是, 程序本身并不是进程, 程序是被动的实体, 如同存储在磁盘上的文件内容, 而进程是一个活动的实体。单线程进程具有一个**程序计数器**来明确下一个执行的指令 (第 4 章将涉及线程问题)。这样一个进程的执行必须是连续的。CPU 一个接着一个地执行进程的指令, 直至进程终止。再者, 在任何时候, 最多只有一个指令代表进程被执行。因此, 尽管两个进程可能与同一个程序相关联, 然而这两个进程都有其各自的执行顺序。多线程进程具有多个程序计数器, 每一个指向下一个给定线程要执行的指令。

进程是系统工作的单元。系统由多个进程组成, 其中一些是操作系统进程 (执行系统代码), 其余的是用户进程 (执行用户代码)。所有这些进程可以潜在地并发执行, 如通过在单 CPU 上采用 CPU 复用来实现。

操作系统负责下述与进程管理相关的活动:

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

从第 3 章到第 6 章, 将讨论进程管理技术。

1.7 内存管理

正如 1.2.2 小节所讨论的, 内存是现代计算机系统操作的中心。内存是一个大的字节或字的数组, 其大小从数十万到数十亿。每个字节或字都有其自己的地址。内存是可以被 CPU 和 I/O 设备所共同快速访问的数据仓库。中央处理器在获取指令周期时从内存中读取指令, 而在获取数据周期时对内存内的数据进行读出或写入 (在冯·诺依曼结构中)。内存通常是 CPU 所能直接寻址和访问的唯一大容量存储器。例如, 如果 CPU 需要处理磁盘内的数据, 那么这些数据必须首先通过 CPU 生成的 I/O 调用传送到内存中。同样, 如果 CPU 需要执行指令, 那么这些指令必须在内存中。

如果一个程序要执行，那么它必须先变换成绝对地址并装入内存。随着程序的执行，进程可以通过产生绝对地址来访问内存中的程序指令和数据。最后，程序终止，其内存空间得以释放，并且下一程序可以装入并得以执行。

为改善 CPU 的利用率和计算机对用户的响应速度，通用计算机必须在内存中保留多个程序，从而产生对内存管理的需要。内存管理有多种不同的方案。这些方案反映出各种各样的方法，所有特定算法的有效率取决于特定环境。对于某一特定系统的内存管理方法的选择，必须考虑许多因素——尤其是系统的硬件设计。每个算法都要求特定的硬件支持。

操作系统负责下列有关内存管理的活动：

- 记录内存的哪部分正在被使用及被谁使用。
- 当有内存空间时，决定哪些进程可以装入内存。
- 根据需要分配和释放内存空间。

内存管理技术将在第 8 章和第 9 章中讨论。

1.8 存 储 管 理

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

1.8.1 文件系统管理

文件管理是操作系统最为常见的组成部分。计算机可以在多种类型的物理介质上存储信息。磁带、磁盘和光盘是最常用的介质。这些介质都有自己的特点和物理组织。每种介质通过一个设备来控制，如磁盘驱动器或磁带驱动器等，它们都有自己的特点。这些属性包括访问速度、容量、数据传输率和访问方法（顺序或随机）。

文件是由其创建者定义的一组相关信息的集合。通常，文件表示程序（源程序和目标程序）和数据。数据文件可以是数值的、字符的、字符数值或二进制的。文件可以没有格式（例如文本文件），也可以有严格的格式（例如固定域）。显然，文件概念相当广泛。

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

操作系统负责下列有关文件管理的活动：

- 创建和删除文件。
- 创建和删除目录来组织文件。
- 提供操作文件和目录的原语。

- 将文件映射到二级存储上。
- 在稳定存储介质上备份文件。

文件管理技术将在第 10 章和第 11 章讨论。

1.8.2 大容量存储器管理

如前所述，由于内存太小不能容纳所有数据和程序，再加上掉电会失去所有数据，计算机系统必须提供二级存储器（secondary storage）以备份内存。绝大多数现代计算机系统都采用硬盘作为主要在线存储介质来存储程序和数据。许多程序，如编译程序、汇编程序、字处理器、编辑器和格式化程序等，都存储在硬盘上，要执行时才调入内存，在执行时将硬盘作为处理的来源地和目的地。因此，硬盘的适当管理对计算机系统尤为重要。操作系统负责下列有关硬盘管理的活动：

- 空闲空间管理。
- 存储空间分配。
- 硬盘调度。

由于二级存储器使用频繁，因此必须高效。计算机操作的最终速度可能与硬盘子系统的速度和管理该子系统的算法有关。

但是，有时也使用许多比二级存储更慢、价格更低的存储器（有时有更高的容量），如磁盘数据的备份、很少使用的数据、长期档案存储。磁带驱动器及其磁带、CD/DVD 驱动器及光盘就是典型的三级存储（tertiary storage）设备。这些介质（磁带和光盘）格式包括 WORM（一次写，多次读）和 RW（读-写）。

三级存储对系统性能并不是关键，但也必须管理好。有些操作系统对之加以管理，而另一些则将三级存储管理交给应用程序管理。有些操作系统提供的功能包括安装和卸载设备介质、为进程互斥使用分配和释放设备，以及将数据从二级存储器上迁移到三级存储器上。

二级和三级存储管理技术将在第 12 章中讨论。

1.8.3 高速缓存

高速缓存是计算机系统的重要概念之一。信息通常保存在一个存储系统中（如内存）。当使用它时，它会被临时地复制到更快的存储系统——高速缓存中。当需要特定信息时，首先检查它是否在高速缓存中。如果是，可直接使用高速缓存中的信息；否则，使用位于内存中的信息，同时将其复制到高速缓存中以便下次再用。

另外，内部可编程寄存器（如索引寄存器）为内存提供了高速缓存。程序员（或编译程序）使用寄存器分配和替换算法以决定哪些信息应在寄存器中而哪些应在内存中。有的高速缓存完全是由硬件实现的。例如，绝大多数系统都有指令高速缓存以保存下一个要执

行的指令。没有这一高速缓存, CPU 将会等待多个时钟周期以便从内存中获取指令。基于类似原因, 绝大多数系统在其存储层次结构中有一个或多个高速缓存。本书并不关心单独的硬件高速缓存, 因为它们不受操作系统所控制。

由于高速缓存大小有限, 所以高速缓存管理 (cache management) 的设计很重要。对高速缓存大小和置换策略的仔细选择可以极大地提高性能。图 1.9 比较了一个大型工作站与小服务器上的存储性能, 展示了对高速缓存的需求。关于软件控制的高速缓存的各种置换算法将在第 9 章中加以讨论。

| 级别 | 1 | 2 | 3 | 4 |
|----------|------------------|-----------------|-------------|-----------|
| 名称 | 寄存器 | 高速缓存 | 内存 | 磁盘存储 |
| 典型大小 | <1 KB | >16 MB | >16 GB | >100 GB |
| 实现技术 | 带多个端口的定制内存, CMOS | 片上或片下 CMOS SRAM | CMOS DRAM | 磁盘 |
| 访问时间(ns) | 0.25~0.5 | 0.5~25 | 80~250 | 5 000 000 |
| 带宽(Mbps) | 20 000~100 000 | 5 000~10 000 | 1 000~5 000 | 20~150 |
| 受控于 | 编译器 | 硬件 | 操作系统 | 操作系统 |
| 底层支持者 | 高速缓存 | 内存 | 磁盘 | CD或磁带 |

图 1.9 不同级别存储器的性能

内存可用做外存的高速缓存, 因为外存数据必须先复制到内存才可使用, 数据在移至外存保存前也必须保存在内存中。永久地驻留在外存上的文件系统数据, 可以出现在存储系统的许多层次上。在最高层, 操作系统可在内存中保存一个文件系统数据的高速缓存。而且, 电子 RAM 磁盘 (固体磁盘 (solid-state disk)) 也可用做通过文件系统接口访问的高速存储。外存以磁盘为主。磁盘存储又可以用磁带或可移动磁盘来备份数据以免受磁盘损坏的影响。有的系统自动地将位于磁盘上的旧文件的数据备份到三级存储, 如磁带塔上, 以降低存储费用 (参见第 12 章)。

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

对于层次存储结构, 同样的数据可能出现在不同层次的存储系统上。例如, 整数 A 位于文件 B 中且需要加 1, 而文件 B 位于磁盘上。加 1 操作这样进行: 先发出 I/O 操作以将 A 所在的磁盘块调入内存。之后, A 被复制到高速缓存和硬件寄存器。这样, A 的副本出现在许多地方: 磁盘上, 内存中, 高速缓存中, 硬件寄存器中 (见图 1.10)。一旦加法在内部寄存器中执行后, A 的值在不同存储系统中会不同。只有在 A 的新值从内部寄存器写回磁盘时, A 的值才会一样。

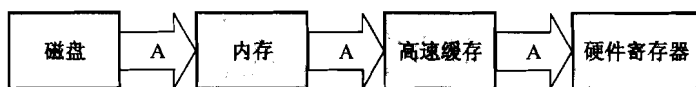


图 1.10 整数 A 从磁盘到寄存器的迁移

对于同时只有一个进程执行的计算环境，这种安排没有困难，因为对 A 的访问总是在层次结构的最高层进行。不过，对于多任务环境，CPU 会在进程之间来回切换，所以需要十分谨慎以确保当多个进程访问 A 时，每个进程都得到最近已更新的 A 值。

对于多处理器环境，这种情况变得更为复杂，因为每个 CPU 不但有自己的内部寄存器，还有本地高速缓存。对于这种环境，A 的副本会同时出现在多个高速缓存中。由于多个 CPU 可并发执行，必须确保在一个高速缓存中对 A 值的更新马上反映在所有其他 A 所在的高速缓存中。这称为**高速缓存一致性**（cache coherency），这通常是硬件问题（在操作系统级别之下处理）。

对于分布式环境，这种情况变得异常复杂。在这种情况下，同一文件的多个副本会出现在多个分布在不同场所的不同计算机上。由于各个副本可能会被并发访问和更新，所以必须确保当一处的副本被更新时，所有其他副本应尽可能快地加以更新。如第 17 章所述，有许多方法可达到这种条件。

1.8.4 I/O 系统

操作系统的目的之一在于对用户隐藏具体硬件设备的特性。例如，在 UNIX 系统中，I/O 子系统对操作系统本身隐藏了 I/O 设备的特性。I/O 子系统包括如下几个部分：

- 一个包括缓冲、高速缓存和假脱机的内存管理部分。
- 通用设备驱动器接口。
- 特定硬件设备的驱动程序。

只有设备驱动程序知道它被赋给的特定设备的特性。

1.2.3 小节讨论了中断处理器和设备驱动程序是如何应用到有效的 I/O 子系统中的。在第 13 章，将要讨论 I/O 子系统如何提供给其他系统部件接口、管理设备、传输数据，以及检测 I/O 完成。

1.9 保护和安全

如果计算机系统有多个用户，并允许多个进程并发执行，那么必须系统地管理对数据的访问。为此，系统采用了各种机制确保只有从操作系统中获得了恰当授权的进程才可以操作相应的文件、内存段、CPU 和其他的资源。例如，内存寻址硬件保证一个进程仅可以

在它自己的地址空间内执行，定时器确保没有进程能一直占有 CPU 控制权而不释放它，用户不能访问设备控制寄存器，因而保护了各种外部设备的完整性。

保护是一种控制进程或用户对计算机系统资源的访问的机制。这个机制必须为强加控制提供一种规格说明方法和一种强制执行方法。

通过检测组件子系统接口的潜在错误进行保护能够提高可靠性。早期检测接口错误通常能防止已经发生故障的子系统影响其他健康的子系统。一个未受保护的资源无法抵御未授权或不合格用户的访问（或误用）。面向保护的系统会提供辨别授权使用和未授权使用的方法，第 14 章将会涉及相关内容。

系统可以获得足够的保护，但也会出错和发生不合适的访问。考虑一个授权信息被偷窃的用户（向系统标识自己的方法），其数据可能被复制或删除，但文件和内存保护仍在运行。**安全（security）**的主要工作是防止系统不受外部或内部攻击。这些攻击范围很广，包括病毒和蠕虫、拒绝服务攻击（使用所有的系统资源以致合法的用户不能使用）、身份偷窃、服务偷窃（未授权的系统使用）。在有些系统中，阻止这些攻击需要考虑操作系统的功能，而另一些系统则是采用策略或者软件阻止方法。由于安全事件急剧增长，操作系统的安全问题成了快速增长的研究和实现的领域。第 15 章将介绍安全问题。

保护和**安全**需要系统能区分它的所有用户。绝大多数操作系统维护一个用户和相关用户标识（user ID, UID）的链表。在 Windows NT 中，这称为**安全 ID（Secure ID, SID）**。这些数值对每个用户来说是唯一的。当用户登录到系统，鉴别步骤会确定用户的合适 ID。该用户 ID 与所有该用户的进程和线程相关联。当用户（人）需要这些 ID 为可读时，它们可通过用户名称链表而转换成用户名。

有些环境中，需要区分用户集而不是单个用户。例如，UNIX 系统上一个文件的拥有者可能被允许对文件进行所有操作，而有些选定的用户只能读文件。为此，需要定义一个组名以及属于该组的用户集。组功能可用系统范围内的组名列表和**组标识（group identity）**来实现。一个用户可以属于一个或多个组，这取决于操作系统设计方法。用户的组 ID 也包含在每一个相关的进程和线程中。

在一般的系统使用中，用户 ID 和组 ID 就足够了。但用户有时需要**升级特权（escalate privilege）**来获取对一个活动的额外特权。例如，该用户可能需要访问受限的设备。操作系统提供了各种允许升级特权的方法。例如，在 UNIX 系统中，程序的 `setuid` 属性使得程序以文件所属用户的 ID 来运行，而不是当前的用户 ID。进程用此有效 **UID（effective UID）** 运行，直至它关掉特权或终止。考虑一个在 Solaris10 中的例子，通过 `/etc/passwd:pbg:x:101:14::/export/home/pbg:/usr/bin/bash`，赋予用户 `pbg` 的用户 ID 为 101，组 ID 为 14。

1.10 分布式系统

分布式系统是将一组物理上分开来的、各种可能异构的计算机系统通过网络连接在一起，为用户提供系统所维护的各种资源的计算机的集合。访问共享资源增加了计算速度、功能、数据可用性及可靠性。有些操作系统将网络访问简化为一种文件访问，网络细节包含在网络接口驱动程序中，而其他的系统采用用户调用网络函数的方式。通常，系统包括两种模式组合——如 FTP 和 NFS。生成分布式系统的协议通常会影响系统的效用和普及程度。

网络，简单地说，就是两个或多个系统之间的通信路径。分布式系统通过网络提供功能。网络随所使用的协议、节点距离、传输介质的变换而不同。TCP/IP 是最常用的网络协议，ATM 和其他协议也有所应用。同样，操作系统对协议的支持也不同。绝大多数操作系统（如 Windows 和 UNIX 操作系统）支持 TCP/IP。有的系统只支持专用协议以满足其需求。对于操作系统而言，一个网络协议只简单地需要一个接口设备，如网络适配器，加上管理它的驱动程序以及按网络协议处理数据的软件。这些概念将在本书中进行讨论。

网络可根据节点间的距离来划分。**局域网**（local-area network, LAN）位于一个房间、一楼层或一栋楼内。**广域网**（wide-area network, WAN）通常位于楼群之间、城市之间或国家之间。一个全球性的公司可以用 WAN 将其全世界范围内的办公室连起来。这些网络可能运行单个或多个协议。新技术的不断出现带来新型网络。例如，**城域网**（metropolitan-area network, MAN）可以将一个城市内的楼宇连接起来。蓝牙（BlueTooth）和 802.11 技术可以在数米内实现无线通信，创建了可能建在房间内的小域网（small-area network, SAN）。

承载网络的介质同样是不同的，包括铜线、光纤、卫星之间的无线传输、微波和无线电。当计算设备连接到手机时，就创建了一个网络。即使非常近距离的红外通信也可用来构建网络。基本上，无论计算机何时通信，它们都要使用或构建一个网络。这些网络的性能和可靠性各不相同。

有些操作系统采用了比只提供网络连接更进一步的网络和分布式系统的概念。**网络操作系统**（network operating system）就是这样一种操作系统，它提供跨网络的文件共享、包括允许在不同计算机上的进程进行消息交换的通信方法等功能。相对于网络上的其他计算机，运行网络操作系统的计算机是自治的。分布式操作系统提供较少的自治环境：不同的操作系统紧密地连接，好像是一个操作系统在控制网络一样。

第 16 章到第 18 章将介绍有关计算机网络和分布式系统的内容。

1.11 专用系统

迄今为止，所讨论的主要是大家都很熟悉的通用计算机系统。但也有些其他类型的计算机系统，它们的功能有限，其目的在于处理有限的计算领域。

1.11.1 实时嵌入式系统

嵌入式计算机是目前最为流行的一种计算机形式，从汽车引擎和制造机器人，到录像机和微波炉，到处都可以找到它们的身影。它们都具有特定的任务，所运行的系统都很简单，因此操作系统仅提供了有限的功能。通常，它们只具有很少甚至没有用户接口，而将它们的时间花费在监视和管理硬件设备上，如汽车引擎和机械手。

这些嵌入式系统种类变化相当大。有些是通用计算机，运行标准的操作系统，如 UNIX，具有专门的应用程序来实现其功能，而其他是具有专用嵌入式操作系统的硬件设备，仅提供所需要的功能。然而，还有其他的硬件设备，它们具有不采用操作系统就能完成任务的特殊集成电路（ASIC）。

嵌入式系统的应用还在延伸。这些设备的能力，无论是作为独立单元还是作为网络或 Web 的组成，都在增强。即便现在，一幢房屋也都可以电脑化，以便一个中心计算机（无论是通用计算机还是嵌入式计算机）可以控制光和热、警报系统，甚至是咖啡机。Web 访问使主人可以告诉他的房子在他回家之前加温。有一天，冰箱可能也可以在发现牛奶没有时通知食品杂货店送货。

嵌入式系统几乎都运行**实时操作系统**。当对处理器操作或数据流动有严格时间要求时，就需要使用实时系统。因此，它常用于控制特定应用的设备。传感器将数据送给计算机，计算机必须分析这些数据并可能调整控制以改变传感器的输入。对科学实验、医学成像系统、工业控制系统和部分显示系统进行控制的系统为实时系统。有些汽车喷油系统、家电控制器和武器系统也属于实时系统。

实时系统有明确而固定的时间约束。处理**必须在**固定时间约束内完成，否则系统将会失败。例如，如果机械手在打坏所造的汽车之后才能停止，那么这就不行了。一个实时系统只有在其时间约束内返回正确结果才是正确工作。可将这一要求与分时系统（只是需要（而不是必须）响应快）或批处理系统（没有任何时间约束）相比较。

第 19 章将更为详细地讲述嵌入式系统。第 5 章将讨论在操作系统中用来实现实时功能的调度组件。第 9 章将介绍实时计算的内存管理。最后，第 22 章将研究 Windows XP 操作系统的实时部件。

1.11.2 多媒体系统

大多数操作系统被设计用来处理传统的数据，如文本文件、程序、字处理文档和电子表格。但是，最近的技术发展趋势是将**多媒体数据**加入到计算机系统中。多媒体数据包括声音和图像数据，以及其他常用的文件。这些数据与常用数据不同之处在于多媒体数据，如图像帧，必须根据确定的时间限制（如每秒 30 帧）来传输（流）。

多媒体带来了广泛的应用，现在用得非常普遍，包括如 MP3、DVD 电影、视频会议、短的录像剪辑，或从 Internet 下载的新闻故事。多媒体应用还包括现场 Web 广播（WWW 上的广播）或运动事件，以及允许一个在曼哈顿的观察者观察在巴黎的咖啡馆的一个顾客的网络摄影。多媒体应用不需要声音或视频，但它通常包括它们的组合。例如，电影可能包括单独的声音和视频轨道。另外，不必仅为桌面个人计算机提供多媒体应用。多媒体越来越多地直接应用到更小的设备，包括 PDA 和手机。例如，一个股票交易人通过无线将股市行情实时地传输到他自己的 PDA 上。

第 20 章将研究多媒体应用的需求，多媒体数据与常用数据有何不同，这些特性如何影响操作系统的设计，以支持多媒体系统的需要。

1.11.3 手持系统

手持系统（handheld system）包括个人数字助理（personal digital assistant, PDA），如 Palm、Pocket-PC 和手机，其中许多都使用专门的嵌入式操作系统。手持系统和应用程序的开发人员面临着许多挑战，绝大多数是由于这些设备的有限尺寸。例如，PDA 通常长约 5 英寸而宽约 3 英寸，重不到半磅。由于尺寸有限，绝大多数手持设备内存小，处理速度慢，屏幕小。下面简要讨论一下这些限制。

手持设备的物理内存取决于设备本身，通常只有 512 KB~128 MB 的内存空间（而 PC 或工作站有数 GB 的内存）。因此，操作系统和应用程序必须有效地管理内存。这包括一旦已分配的内存不再使用就应返回给内存管理器。第 9 章将会研究虚拟内存，它允许开发人员编写程序时可以认为系统有比物理内存更多的可用内存。现在，许多手持设备都不使用虚拟内存，因此程序开发人员必须在有限物理内存的约束内工作。

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

手持设备程序设计人员所面临的最后一个问题是 I/O 问题。缺乏物理空间限制了小键盘、手写识别或者基于小屏幕键盘的输入方法。显示屏幕小限制了输出选择。虽然家用计

算机的显示器可达 30 英寸,但是手持设备的显示屏往往不到 3 平方英寸。常用的任务,如阅读电子邮件或浏览网页,必须在更小的显示器上进行。一种显示网页的方法是网页剪辑(Web clipping),即在手持设备上只传送和显示一小部分网页。

有些手持设备可使用无线技术,如蓝牙或 802.11,允许远程访问电子邮件和浏览网页,与 Internet 相连的手机就属于这一类型。然而,对于不支持无线访问的 PDA,为了下载数据到这些设备,通常需要先下载数据到 PC 或工作站,接着再下载到 PDA。有的 PDA 允许通过红外线在 PDA 之间实现数据复制。

不过,PDA 的这些功能限制被其方便性和便携性所抵消。随着网络功能的增加和其他选择(如照相机和 MP3 播放机)不断扩展其应用,它们的使用会不断地增加。

1.12 计 算 环 境

到目前为止,已经大致了解了计算机系统的组织以及主要的操作系统部件,下面简单地分析一下这些系统的计算环境。

1.12.1 传统计算

随着计算的不断发展,许多传统计算的分界已变得模糊。现在考虑一下“典型办公环境”。几年前,这种环境由一些连网的 PC 组成,服务器提供文件和打印服务,远程访问很不方便,移动功能是通过在办公室内携带笔记本计算机来完成的。许多公司都有与主机相连的终端,其远程访问能力和移动性就更差。

现代发展趋势是提供更多方法访问这些计算环境。Web 技术正在扩展传统计算机的边界。公司实现了入口(portal)以访问内部服务器。网络计算机(network computer)是可以实现 Web 计算的终端。手持计算机能与 PC 同步来实现对公司信息的可移动使用。手持 PDA 也可以通过无线网络来使用公司的 Web 入口(和其他 Web 资源)。

在家里,过去绝大多数用户都有一台计算机通过低速的调制解调器与公司或 Internet 相连。现在,曾经很昂贵的高速网络连接也已经很便宜,这允许用户访问更多的数据。这些快速数据连接允许家庭计算机提供 Web 服务,运行自己的网络(包括打印机、客户端 PC 和服务器的)。有的家庭还有防火墙(firewall)来保护家庭内部环境以避免被破坏。这些防火墙几年前价值数千美元,而 10 年前几乎不存在。

20 世纪下半叶,计算机资源非常贫乏(在此之前它们根本就不存在)。有一段时间,系统或是批处理的,或是交互式的。批处理系统以批量的方式处理作业,具有事先定义的输入(从文件或其他数据资源);而交互式系统等待来自用户的输入。为了优化计算资源的使用,采用多个用户来共享这些系统的时间。分时系统采用定时器和调度算法,通过 CPU 迅速地循环进程,给其中每一用户分配资源。

现在, 传统的分时系统不太常见了。相同的调度技术仍在工作站和服务器的使用, 但进程通常为同一用户所拥有 (或单个用户和操作系统), 用户进程及提供用户服务的系统进程一起管理, 都能获得一定时间的计算。例如, 在用户使用一台 PC 时, 会创建许多窗口, 它们同时执行不同的任务。

1.12.2 客户机-服务器计算

随着 PC 变得更快、更强大和更便宜, 设计者开始抛弃中心系统结构。与中心系统相连的终端开始被 PC 所取代。相应地, 过去为中心系统所处理的用户接口功能也被 PC 所取代。因此, 今天中心系统成为**服务器系统 (server system)** 以满足**客户机系统 (client system)** 的请求。这种被称为**客户机-服务器 (client-server)** 系统的专有分布式系统, 具有如图 1.11 所示的通用结构。

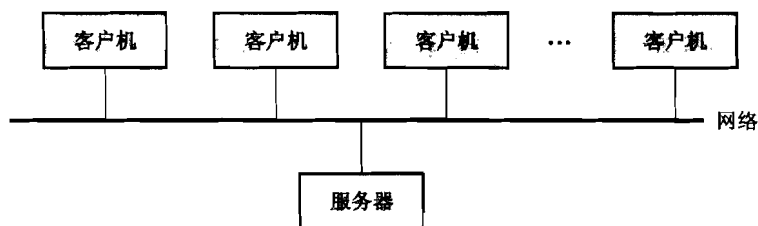


图 1.11 客户机-服务器系统的常用结构

服务器系统可大致分为计算服务器和文件服务器:

- **计算服务器系统**提供了一个接口, 以接收用户所发送的执行操作的请求 (如读数据), 执行操作, 并将操作结果返回给客户机。运行响应客户机数据请求的数据库的服务器就是一个这样的例子。
- **文件服务器系统**提供文件系统接口, 以便客户机能创建、更新、访问和删除文件。Web 服务器就是该系统的一个例子, 它将文件传送到正在运行 Web 浏览器的客户机。

1.12.3 对等计算

分布式系统的另一种结构是对等 (P2P) 系统模式。采用该模式, 客户机和服务器彼此并不区别, 而是系统中的所有节点都是对等的, 每一个都可作为客户机或服务器, 这取决于它是请求还是提供服务。对等系统相对于传统的客户机-服务器系统提供了更好的性能。在客户机-服务器系统中, 存在服务器瓶颈问题; 但在对等系统中, 可以由分布在网络中的多个节点来提供服务。

为了加入对等系统, 节点必须先加入对等网络。一旦节点加入对等网络, 它就可以开始向网络中的其他节点提供服务或请求服务。可用下面两种方法来决定当前有哪些服务

可用:

- 当一个节点加入网络时, 它用网络集中查询服务来注册它的服务。任何需要某种服务的节点首先与此集中查询服务联系, 以决定哪个节点能提供此服务。剩下的通信就在客户机和服务者之间进行。

- 作为客户机的对等行动必须首先通过向所有网络中的其他节点广播服务请求, 以发现哪个节点提供所需的服务。提供该服务的节点 (或多个节点) 响应发出此请求的对等节点。为了支持该方法, 必须提供一种发现协议 (discovery protocol), 以允许网络上的对等节点能发现服务。

对等网络在 20 世纪 90 年代后期在多文件共享服务上得到广泛的应用, 如 Napster 和 Gnutella, 它们能对等交换文件。Napster 系统采用类似上述的第一种方法: 一个中央服务器维护存储在 Napster 网络上对等节点的所有文件的索引, 实际的文件交换发生在对等节点之间。Gnutella 系统采用类似上述的第二种方法: 一个客户机向系统中的其他节点广播文件请求, 能够服务该请求的节点直接响应请求。文件交换的未来并不确定, 因为有很多文件是有产权保护的 (如音乐), 传播这些授权的材料存在着法律上的限制。但无论如何, 对等技术在将来许多服务中仍将扮演它的角色, 包括查询、文件交换和 E-mail。

1.12.4 基于 Web 的计算

Web 几乎无处不在, 能为多种设备所访问, 这是数年前所难以想象的。PC 仍然是最为普通的访问设备, 而工作站、手持 PDA 和手机等也能访问 Web。

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

Web 计算的实现也导致了新一类设备的出现, 如负载平衡器 (load balancer), 它能在一组相似的服务器之间实现负荷分配。操作系统过去只能作为 Web 客户机 (如 Windows 95), 现在也发展成为既可做 Web 服务器, 又可作为客户机 (如 Linux 和 Windows XP)。通常因为用户需要支持 Web 驱动, 所以增加了设备的复杂性。

1.13 小 结

操作系统是管理计算机硬件并提供应用程序运行环境的软件。也许操作系统最为直观之处在于它提供了人与计算机系统的接口。

为了让计算机执行程序, 程序必须位于内存中。内存是处理器能直接访问的唯一的大容量存储区域。内存为字节或字的数组, 容量为数百 KB 到数百 MB。每个字都有其地址。内存是易失性存储器, 当没有电源时会失去其内容。绝大多数计算机系统都提供了外存以扩充内存。二级存储器提供了一种非易失存储, 它可以长久地存储大量数据。最常用的二

级存储器是磁盘，它提供对数据和程序的存储。

根据速度和价格，可以将计算机系统的不同存储系统按层次来组织。最高层最为昂贵但也最快。随着向层次结构下面移动，每一个位的存储价格通常降低，而访问时间通常增加。

计算机系统的设计有多种不同的方法。单处理器系统只有一个处理器，而多处理器系统包含两个或更多的处理器来共享物理存储及外设。对称多处理技术（SMP）是最为普通的多处理器设计技术，其中所有的处理器被视为对等的，且彼此独立地运行。集群系统是一种特殊的多处理器系统，它由通过局域网连接的多个计算机系统组成。

为了最好地利用 CPU，现代操作系统采用允许多个作业同时位于内存中的多道程序设计，以保证 CPU 中总有一个作业在执行。分时系统是多道程序系统的扩展，它采用调度算法实现作业之间快速的切换，好像每个作业在同时进行一样。

操作系统必须确保计算机系统的正确操作。为了防止用户干预系统的正常操作，硬件有两种模式：用户模式和内核模式。许多指令（如 I/O 指令和停机指令）都是特权的，只能在内核模式下执行。操作系统所驻留的内存也必须加以保护以防止用户程序修改。定时器防止无穷循环。这些工具（如双模式、特权指令、内存保护、定时器中断）是操作系统所使用的基本单元，用以实现正确操作。

进程（或作业）是操作系统工作的基本单元。进程管理包括创建和删除进程、为进程提供与其他进程通信和同步的机制。操作系统通过跟踪内存的哪部分被使用及被谁使用来管理内存。操作系统还负责动态地分配和释放内存空间，同时还管理存储空间，包括为描述文件提供文件系统和目录，以及管理大存储设备设备的空间。

操作系统必须考虑到它与用户的保护和安全问题。保护是提供控制进程或用户访问计算机系统资源的机制。安全措施用来抵御计算机系统所受到的外部或内部的攻击。

分布式系统允许用户共享通过网络连接的、在地理位置上是分散的计算机的资源。可以通过客户机-服务器模式或对等模式来提供服务。在集群系统中，多个机器可以完成驻留在共享存储器上的数据的计算，即便某些集群的子集出错，计算仍可以继续。

局域网和广域网是两种基本的网络类型。局域网允许分布在较小地理区域内的处理器进行通信，而广域网允许分布在较大地理区域内的处理器进行通信。局域网通常比广域网快。

计算机系统具有一些特殊的服务目的，包括为嵌入式环境设计的实时操作系统，如消费设备、汽车和机器人。实时操作系统具有已定义的、固定的时间约束。进程必须在定义的约束内执行，否则系统将出错。多媒体系统涉及多媒体数据传送，常常有显示或使用音频、视频或者同步的音频和视频流的特别要求。

近来，由于 Internet 和 WWW 的影响，现代操作系统也集成了 WWW 浏览器、网络和通信软件。

习 题

- 1.1 在多道分时环境下,有几个用户同时使用一个系统,这种情况可能导致各种安全问题。
 - a. 列出两个此类问题。
 - b. 在一个分时系统中,能否像在特殊用途系统中一样确认同样的安全程度?并解释它。
- 1.2 资源利用问题在不同的操作系统中以不同的形式出现。请指出下面哪些资源必须被仔细地管理:
 - a. 主机系统或微型计算机
 - b. 通过服务器连接的工作站
 - c. 手持计算机
- 1.3 在何种环境下,用户使用分时系统优于 PC 或单用户工作站?
- 1.4 在所列的两种设置中,哪些功能需要操作系统提供支持:(a)手持设备和(b)实时系统。
 - a. 批编程
 - b. 虚拟内存
 - c. 分时
- 1.5 描述对称多处理和非对称多处理的区别。多处理系统有哪些优点和缺点?
- 1.6 集群系统与多处理器系统有何区别?属于一个集群的两个机器协作提供高可用性服务需要什么?
- 1.7 区别分布式系统的客户机-服务器模式和对等计算模式。
- 1.8 设想由两个运行数据库的节点构成的一个集群,说出集群软件管理磁盘数据访问的两种方法。论述每种方法的优点和缺点。
- 1.9 网络计算机与传统计算机有什么区别?请描述在哪些情况下使用网络计算机更为有利。
- 1.10 中断有何作用?陷阱和中断有何区别?用户程序能否有意地生成陷阱?如果是,有什么目的?
- 1.11 直接内存访问被用到高速 I/O 设备中,以避免日益增加的 CPU 执行负荷。
 - a. CPU 接口与设备如何协作调度?
 - b. CPU 如何知道内存操作何时结束?
 - c. 当 DMA 控制器在调度数据时,允许 CPU 执行其他程序。该进程与用户程序的执行会不会冲突?如是,说出将会导致何种冲突。
- 1.12 有些计算机系统不支持硬件操作特权模式。能否为这些计算机系统构建一种安全的操作系统?请给出能或不能的理由。
- 1.13 给出高速缓存有用的两个理由。它们解决什么问题?这些问题产生的原因是什么?如果一个高速缓存的容量可以做成和要缓存的设备一样大(如一个和磁盘一样大的缓存),为什么不直接用同样容量的缓存代替该设备呢?
- 1.14 举例说明在下列环境中,如何维护高速缓存数据的一致性:
 - a. 单处理器系统
 - b. 多处理器系统
 - c. 分布式系统
- 1.15 请描述一种加强内存保护以防止程序修改其他程序的内存的相关机制。
- 1.16 什么样的网络配置最适合下面的这些情况?
 - a. 宿舍的一层

- b. 一个大学校园
 - c. 一个州
 - d. 一个国家
- 1.17 列出下列类型操作系统的基本特点。
- a. 批处理
 - b. 交互式
 - c. 分时
 - d. 实时
 - e. 网络
 - f. 并行
 - g. 分布
 - h. 集群
 - i. 手持
- 1.18 手持计算机中固有的折中属性有哪些？

文献注记

Brookshear[2003]给出了计算机科学的概述。

Bovet 和 Cesati[2002]提供了关于 Linux 操作系统的综述。Solomon 和 Russinovich[2000]论述了 Windows 操作系统及重要的系统内部和部件的技术细节。Mauro 和 McDougall[2001]介绍了 Solaris 操作系统，而 <http://www.apple.com/macosx> 给出了有关 Mac OS X 的信息。

Parameswaran 等[2001]、Gong[2002]、Ripeaun 等[2002]、Agre[2003]、Balakrishnan 等[2003]和 Loo[2003]都涉及了对等系统的内容。Lee[2003]中讨论了对等文件共享系统。Buyya[1999]提供了一个关于集群计算的很好的综述，Ahmed[2000]对近来集群计算的进展进行了讨论。Tanenbaum 和 Van Renesse[1985]探讨了有关操作系统支持分布式系统的内容。

有许多关于操作系统的教科书，包括 Stallings[2000b]、Nutt[2004]和 Tanenbaum[2001]。

Hamacher 等[2002]介绍了计算机组织，Hennessy 和 Patterson[2002]讨论了一般 I/O 系统、总线和系统结构。

Smith[1982]描述和分析了高速缓存（包括关联存储器），还包括相关书目。

Freedman[1983]和 Harker 等[1981]论述了磁盘技术。Kenville[1982]、Fujitani[1984]，O'Leary 和 Kitts[1985]，Gait[1988]、Olsen 和 Kenly[1989]讨论了光盘。Pechura 和 Schoeffler[1983]和 Sarisky[1983]讨论了软盘。Chi[1982]和 Hoagland[1985]论述了大容量存储技术。

Kurose 和 Ross[2005]、Tanenbaum[2003]、Peterson 和 Davie[1996]，以及 Halsall[1992]提供了计算机网络的概观。Fortier[1989]给出了关于计算机网络硬件和软件的详细论述。

Wofl[2003]研究了开发嵌入式系统的最近发展，Myers 和 Beigl[2003]、Di Pietro 和 Mancini[2003]论述了有关手持设备的相关内容。

第2章 操作系统结构

操作系统为执行程序提供环境。从内部结构来说，操作系统变化很大，有很多组织方式。设计一个新的操作系统是个大型任务。在开始设计前，定义好系统目标非常重要。所要设计系统的类型决定了如何选择各种算法和策略。

可以从多个角度来研究操作系统。第一是通过考察所提供的服务。第二是通过考察为用户和程序员提供的接口。第三是研究系统的各个组成部分及其相互关系。在本章里，将从用户角度、程序员角度和操作系统设计人员角度来分别研究操作系统的三个方面。本章将研究操作系统提供什么服务、如何提供服务、设计操作系统的各种方法。最后，介绍如何生成操作系统，以及计算机如何启动它的操作系统。

本章目标

- 介绍操作系统为用户、进程和其他系统提供的服务。
- 讨论组织操作系统的不同方法。
- 解释如何安装、定制操作系统，以及如何启动。

2.1 操作系统服务

操作系统提供一个环境以执行程序。它向程序和这些程序的用户提供一定的服务。当然，所提供的具体服务随操作系统而不同，但还是有一些共同特点。这些操作系统服务方便了程序员，使得编程更加容易。

一组操作系统服务提供对用户很有用的函数：

- **用户界面**：所有的操作系统都有**用户界面**（user interface, UI）。用户界面可以有多种形式。一种是**命令行界面**（command-line interface, CLI），它采用文本命令，并用一定的方法输入（即一种允许输入并编辑的命令）。另一种是批界面，其中控制这些命令和命令的指令被输入文件中，通过执行文件来实现。最为常用的是**图形用户界面**（graphical user interface, GUI），此时界面是一个视窗系统，它具有定位设备来指挥 I/O、从菜单来选择、选中部分并用键盘输入文本。有些系统还提供了两种甚至所有这三种界面。

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

• **I/O 操作**：运行程序可能需要 I/O，这些 I/O 可能涉及文件或设备。对于特定设备，需要特定的功能（如刻录 CD 或 DVD 驱动器，或清屏）。为了提高效率和进行保护，用户通常不能直接控制 I/O 设备。因此，操作系统必须提供进行 I/O 操作的方法。

• **文件系统操作**：文件系统特别重要。很明显，程序需要读写文件和目录，也需要根据文件名来创建和删除文件、搜索一个给定的文件、列出文件信息。最后，有些程序还包括了基于文件所有权的允许或拒绝对文件或目录的访问管理。

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

• **错误检测**：操作系统需要知道可能出现的错误。错误可能发生在 CPU 或内存硬件（如内存错误或电源失败）、I/O 设备（磁带奇偶出错，网络连接出错，打印机缺纸）和用户程序中（如算术溢出，试图访问非法内存地址，使用 CPU 时间太长）。对于每种类型的错误，操作系统应该采取适当的动作以确保正确和一致的计算。调试工具可以在很大程度上加强用户和程序员有效使用系统的能力。

另外，还有一组操作系统函数，它们不是帮助用户而是确保系统本身高效运行。多用户系统通过共享计算机资源可以提高效率。

• **资源分配**：当同时有多个用户或多个作业运行时，系统必须为它们中的每一个分配资源。操作系统管理多种不同的资源。有的资源（如 CPU 周期、内存和文件存储）可能要有特别的分配代码，而其他的资源（如 I/O 设备）可能只需要通用的请求和释放代码。例如，为了最好地使用 CPU，操作系统需要采用 CPU 调度算法以考虑 CPU 的速度、必须执行的作业、可用的寄存器数和其他因素。还有一些其他程序可以分配打印机、Modem、USB 存储设备和其他外设。

• **统计**：需要记录哪些用户使用了多少和什么类型的资源。这种记录可用于记账（以便让用户交费），或用于统计数据。使用统计数据对研究人员很有用，可用于重新配置系统以提高计算服务能力。

• **保护和安全**：对于保存在多用户或网络连接的计算机系统上的信息，用户可能需要控制信息的使用。当多个进程并发执行时，一个进程不能干预另一个进程或操作系统本身。保护即确保所有对系统资源的访问是受控的。系统安全不受外界侵犯也很重要。这种安全从用户向系统证明自己（利用密码）开始，以获取对系统资源访问权限。安全也包括保护外部 I/O 设备，如 Modem 和网络适配器不受非法访问，并记录所有非法闯入的企图。如果一个系统需要保护和安全，那么系统中的所有部分都要预防。一条链子的强度与其最弱的链环相关。

2.2 操作系统的用户界面

用户与操作系统的界面有两种基本的方法。第一种方法是提供命令行界面或命令中断，允许用户直接输入通过操作系统完成的命令；第二种方法允许用户通过图形用户界面（GUI）与操作系统交互。

2.2.1 命令解释程序

有的操作系统在其内核部分包含命令解释程序。其他操作系统，如 Windows XP 和 UNIX，将命令解释程序作为一个特殊程序，当一个任务开始时或用户首次登录时（分时系统），该程序就会运行。在具有多个命令解释程序选择的系统中，解释程序被称为外壳（Shell）。例如，在 UNIX 和 Linux 系统中，有多种不同的 Shell 可供用户选择，包括：Bourne Shell、C Shell、Bourne-Again Shell、Korn Shell 等。许多 Shell 除细小的差别外，都提供类似的功能，许多用户对 Shell 的选择只是基于个人偏好。

命令解释程序的主要作用是获取并执行用户指定的下一条命令。这一层中提供的许多命令都是操作文件的：创建、删除、列出、打印、复制、执行等，MS-DOS 和 UNIX 的 Shell 就是这样工作的。执行这些命令有两种常用的方法。

一种方法是命令解释程序本身包含代码以执行这些命令。例如，删除文件的命令可能导致命令解释程序转到相应的代码段以设置参数和执行合适的系统调用。对于这种方法，所能提供的命令的数量决定了命令解释程序的大小，这是因为每个程序需要它自己实现代码。

另一种方法为许多操作系统如 UNIX 所使用，由系统程序实现绝大多数命令。这样，命令解释程序不必理解什么命令，它只要用命令来识别文件以装入内存并执行。因此，UNIX 删除文件的命令

`rm file.txt`

会搜索名为 `rm` 的文件，将该文件装入内存，并用参数 `file.txt` 来执行。与 `rm` 命令相关的功能是完全由文件 `rm` 的代码所决定。这样，程序员能通过创建合适名称的新文件以轻松地 toward 系统增加新命令。这种命令解释程序可能很小，在增加新命令时不必改变。

2.2.2 图形用户界面

与操作系统交互的第二种方法是采取友好的用户图形界面（GUI）。与用户通过命令行直接输入命令不同，GUI 允许提供基于鼠标的窗口和菜单系统作为接口。GUI 提供了桌面的概念，用户移动鼠标把指针定位到屏幕（桌面）的图像（图标，**icon**）上。图标代表程序、文件、目录和系统功能。根据鼠标指针的位置，按一下鼠标按钮可以调用程序、选择

文件和目录（被称为文件夹）或打开包含命令的菜单。

图形用户界面首次出现在 20 世纪 70 年代的 Xerox PARC 研究设备的研究工作中。1973 年产生了第一个 GUI 界面。但是，直到 20 世纪 80 年代随苹果公司 Macintosh 计算机的出现，图形界面才变得普及。这些年来，Macintosh 操作系统（Mac OS）的用户界面经过了许多变化，最显著的变化是 Mac OS X 采用的 Aqua 界面。微软公司的第一个 Windows 版本（版本 1.0）就是基于给 MS-DOS 操作系统提供的 GUI 界面。后续的 Windows 版本对 GUI 的外观做了一些表面改进，并加强了一些功能，包括 Windows Explorer。

传统上，UNIX 系统使用命令行界面。然而还有很多不同的 GUI 界面，包括通用桌面环境（CDE）和 X 窗口系统，已在诸如 Solaris 和 IBM 的 AIX 系统等 UNIX 商用系统中广为应用。当然，还是有大量的开源（open source）项目的 GUI 设计开发出现，如 K 桌面环境（KDE）和 GNU 项目的 GNOME 桌面。KDE 和 GNOME 桌面都可以运行在 Linux 和不同的 UNIX 系统中，可根据开放源代码许可获得（这意味着它们的源代码是公共的）。

选择命令行还是 GUI 界面取决于个人喜好。一般规律是，许多 UNIX 用户更喜欢命令行界面，因为它提供了更强大的 Shell 界面。另一方面，绝大多数 Windows 用户喜欢 Windows 的 GUI 环境，而几乎从不使用 MS-DOS 命令行 Shell 界面。Macintosh 操作系统所经历的变化提供了很好的与此相反的研究。在历史上，Mac OS 并没有提供命令行界面，而是要求它的使用者通过 GUI 与操作系统交互。但随着 Mac OS X（部分实现采用 UNIX 内核）的发行，操作系统提供了新的 Aqua 界面和命令行界面两种界面方式。

用户界面可随系统的不同甚至用户的不同而变化。它常被从实际系统结构中删除。因此，友好且有用的用户界面的设计不再是操作系统统管的功能。本书中，集中研究向用户程序提供足够功能的基本问题。从操作系统的角度而言，不用区分用户程序和系统程序。

2.3 系统调用

系统调用（system call）提供了操作系统提供的有效服务界面。这些调用通常用 C 或 C++ 编写，当然，对底层的任务（如必须直接访问的硬件），可能以汇编语言指令的形式提供。

在讨论操作系统如何使其系统调用可用之前，首先用一个例子来解释如何使用系统调用：编写一个从一个文件读取数据并复制到另一个文件的简单程序。程序首先所需要的输入是两个文件的名称：输入文件名和输出文件名。根据操作系统设计的不同，这些名称有许多不同的表示方法。一种方法是程序向用户提问然后得到两个文件名。对于交互系统，这种方法需要一系列的系统调用：先在屏幕上写出提示信息，再从键盘上读取定义两个文件名称的字符。对于基于鼠标和基于图标的系统，一个文件名的菜单通常显示在一个窗口中。用户通过鼠标选择源文件名，另一个类似窗口可以用来选择目的文件名。这个过程需

要许多 I/O 系统调用。

在得到两个文件名后，该程序打开输入文件并创建输出文件。每个操作都需要另一个系统调用。每个操作都有可能遇到错误情况。当程序设法打开输入文件时，它可能发现该文件不存在或者该文件受保护而不能访问。在这些情况下，程序应该在终端上打印出消息（另一系列系统调用），并且非正常地终止（另一个系统调用）。如果输入文件存在，那么必须创建输出文件。用户可能会发现具有同一名称的输出文件已存在。这种情况可能导致程序中止（一个系统调用），或者必须删除现有文件（另一个系统调用）并创建新的文件（另一个系统调用）。对于交互式系统，另一选择是问用户（一系列的系统调用以输出提示信息并从终端读入响应）是否需要替换现有文件或中止程序。

现在两个文件都已设置好，可以进入循环以从输入文件中读（一个系统调用）并向输出文件中写（另一个系统调用）。每个读和写都必须返回一些关于各种可能错误的状态信息。对于输入，程序可能发现已经到达文件的结束，或者在读过程中发生了一个硬件失败（如奇偶检验误差）。对于写操作，根据输出设备的不同可能出现各种错误（如没有磁盘空间、打印机没纸等）。

最后，在整个文件复制完成后，程序可以关闭两个文件（另一个系统调用），在终端或窗口上写一个消息（更多系统调用），最后正常结束（最后的系统调用）。可见，一个简单的程序也会大量使用操作系统。通常，系统每秒执行数千个系统调用。图 2.1 显示了这个系统调用序列。

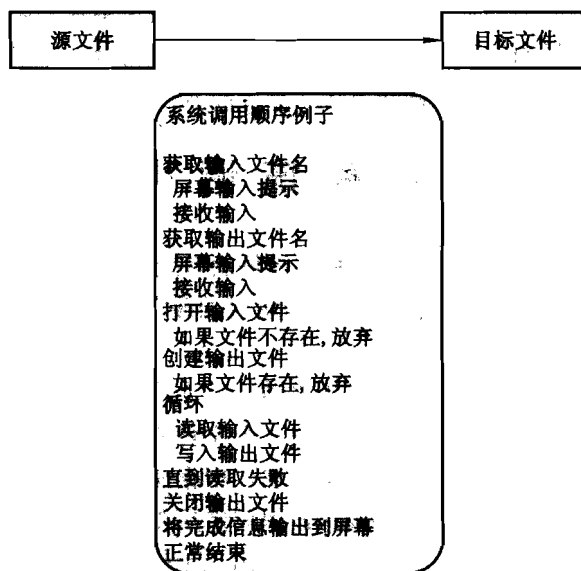


图 2.1 如何使用系统调用的例子

不过，绝大多数程序员不会看到这些细节。一般应用程序开发人员根据应用程序接口 (API) 设计程序。API 是一系列适用于应用程序员的函数，包括传递给每个函数的参数及其返回的程序员想得到的值。有三种应用程序员常用的 API：适用于 Windows 系统的 Win32 API，适用于 POSIX 系统的 POSIX API（包括几乎所有 UNIX、Linux 和 Mac OS X 版本），以及用于设计运行于 Java 虚拟机程序的 Java API。

注意，贯穿本书的系统调用名是常用的例子，每个操作系统都有自己的系统调用命名。

标准 API 的例子

作为标准 API 的一个例子，可考虑 Win32 API 中的 `ReadFile()` 方法，它从文件中读取。这个方法的 API 如图 2.2 所示。

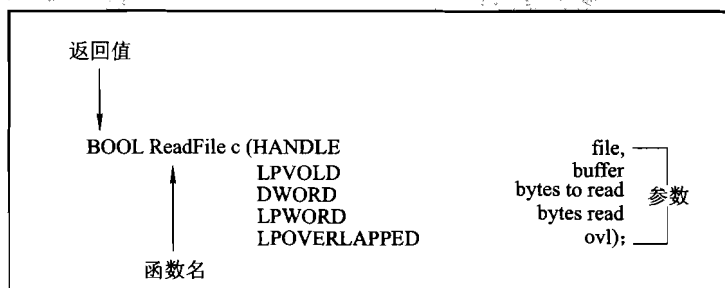


图 2.2 `ReadFile()` 函数的 API

`ReadFile()` 函数的参数描述如下：

- `HANDLE file`：所要读取的文件。
- `LPVOID buffer`：读进写出的数据缓冲。
- `DWORD bytesToRead`：将要读入缓冲区中的字节数。
- `LPDWORD bytesRead`：上次读操作读的字节数。
- `LPOVERLAPPED ovl`：指示是否使用重叠 I/O。

在后台，组成 API 的函数通常为应用程序员调用实际的系统调用。例如，Win32 函数 `CreateProcess()`（用于生成一个新的进程）实际上调用 Windows 内核中的 `NTCreateProcess()` 系统调用。为什么一个应用程序员宁可根据 API 来编程，而不是调用实际的系统调用呢？这有几个原因。根据 API 编程的好处之一在于程序的可移植性，一个采用 API 设计程序的应用程序员希望她的程序能在任何支持同样 API 的系统上编译并执行（尽管事实上，体系的不同常使其很困难）。此外，对一个应用程序员而言，实际的系统调用比 API 更为注重细节和困难。尽管如此，调用 API 中的函数和与其相关的内核系统调用之间还是常常存在

紧密的联系。事实上，许多 Win32 和 POSIX 的 API 与 UNIX、Linux 和 Windows 操作系统提供的自身的系统调用是相类似的。

绝大多数程序设计语言的运行时支持系统（与编译器一起的预先构造的函数库）提供了系统调用接口，作为应用程序与操作系统的系统调用的链接。系统调用接口截取 API 的函数调用，并调用操作系统中相应的系统调用。通常，每个系统调用一个与其相关的数字，系统调用接口根据这些数字维护一个列表索引。然后，系统调用接口来调用所需的操作系统内核中的系统调用，并返回系统调用状态及其他返回值。

调用者不需要知道如何执行系统调用或者执行过程中它做了什么，它只需遵循 API 并了解执行系统调用后，系统做了什么。因此，对于程序员，通过 API 操作系统接口的绝大多数细节被隐藏起来，并被执行支持库所管理。API、系统调用接口和操作系统之间的关系如图 2.3 所示，它表现了操作系统如何处理一个调用 `open()` 系统调用的用户应用。

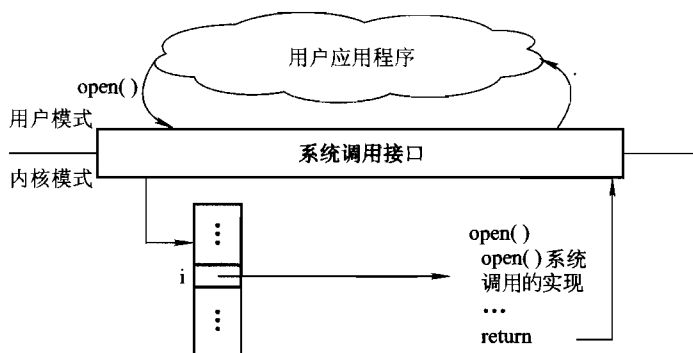


图 2.3 处理一个调用 `open()` 系统调用的用户应用程序

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

向操作系统传递参数有三种方法。最简单的是通过寄存器来传递参数。不过有时，参数数量会比寄存器多。这时，这些参数通常存在内存的块和表中，并将块的地址通过寄存器来传递（见图 2.4）。Linux 和 Solaris 就采用这种方法。参数也可通过程序放在或压入堆栈中，并通过操作系统弹出。有的系统采用块或堆栈方法，因为这些方法并不限制所传递参数的数量或长度。

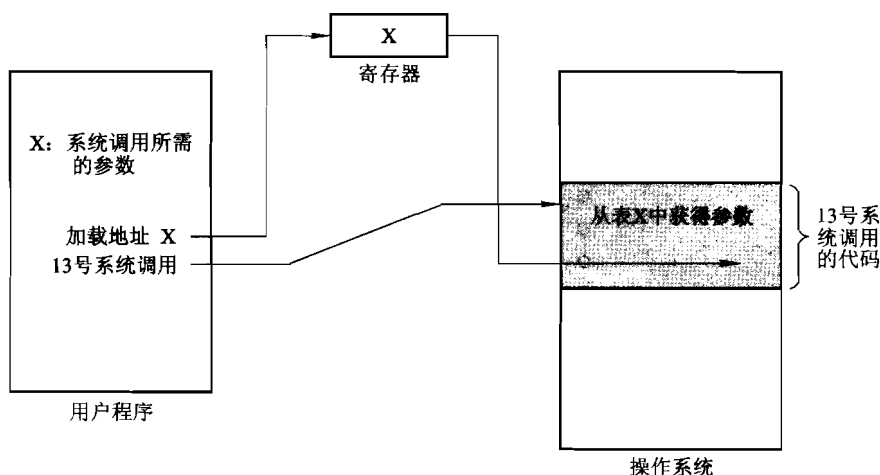


图 2.4 参数作为表传递

2.4 系统调用类型

系统调用大致可分成五大类：**进程控制**、**文件管理**、**设备管理**、**信息维护**和**通信**。在 2.4.1 小节到 2.4.5 小节，将简要描述操作系统可能提供的系统调用类型。这些系统调用大多支持后面几章所讨论的有关概念和功能，或被其所支持。图 2.5 总结了操作系统通常提供的系统调用类型。

2.4.1 进程控制

运行程序需要能正常或非正常地中断其执行（end 或 abort）。如果一个系统调用被用来非正常地中断执行程序，或者程序运行碰到问题而引起错误陷阱，那么可能会有内存信息转储并产生一个错误信息。内存信息转储通常写到磁盘上，并被**调试器**（帮助程序员发现和纠正错误的系统程序）检查和确定问题原因。不管是正常还是非正常中止，操作系统都必须将控制权转交给调用命令解释器。命令解释器接着读取下一个命令。对于交互系统，命令解释器只不过简单地读取下一个命令，因为假定用户会采取合适的命令以处理错误。对于 GUI 系统，一个弹出窗口可提醒用户出错并请求建议。对于批处理系统，命令解释器通常终止整个作业并继续下一个作业。当出现一个错误的时候，有的系统允许控制卡指出一个具体的恢复动作。**控制卡**是一个批处理系统概念，它是一个管理进程执行的命令。如果程序发现输入有错并想要非正常地终止，那么它可能也需要定义一个错误级别。更加严重的错误可以用更高级的错误参数来表示。如果将正常终止定义为级别为 0 的错误，那么

可能将正常和非正常终止混合起来。命令解释器和下一个程序能利用错误级别来自动决定下一个动作。

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

图 2.5 系统调用的类型

执行一个程序的进程或作业可能需要装入和执行另一个程序。这一点允许命令解释器来执行一个程序, 该命令可通过用户命令、鼠标单击或批处理命令来表示。当装入程序终止时, 一个有趣的问题是控制权返回到哪里。这个问题与现有程序是否丢失、保存或与新程序继续并发执行有关。

如果新程序终止时控制权返回到现有程序, 那么必须保存现有程序的内存映像。因此, 事实上创建了一个机制以便一个程序调用另一个程序。如果两个程序并发继续, 那么创建了一个新作业和进程以便多道执行。通常, 有的系统调用专门用于这一目的 (如 `create`

process 或 submit job)。

如果创建一个新作业或进程，或者一组作业或一组进程，那么应该能控制它的执行。这种控制要求能决定和重置进程或作业的属性，包括作业的优先级、最大允许执行时间等 (get process attributes 和 set process attributes)。如果发现所创建的进程不正确或不再需要，那么也要能终止它 (terminate process)。

标准 C 程序库的例子

标准 C 程序库提供了许多 UNIX 和 Linux 版本的部分系统调用接口。举个例子，假定 C 程序调用了 printf() 语句。C 程序库拦截这个调用来调用必要的操作系统系统调用 (在本例中是 write() 系统调用)。C 程序库把 write() 的返回值传递给用户程序，见图 2.6。

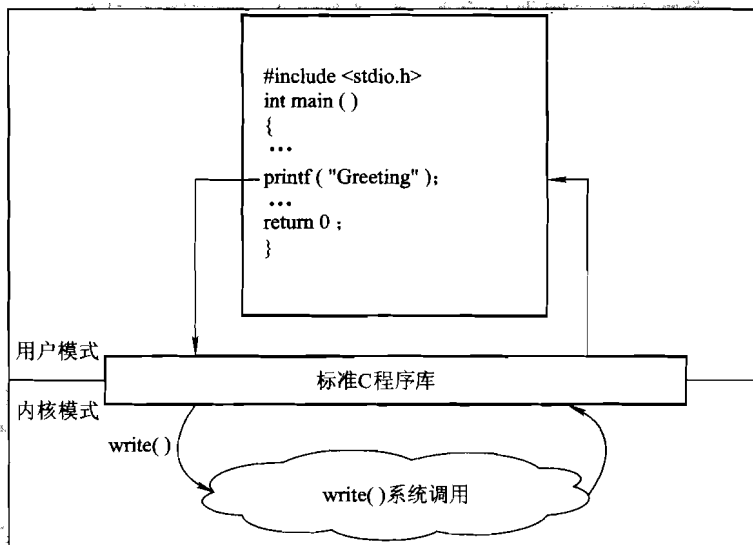


图 2.6 write() 的 C 库函数处理

创建了新作业和进程之后，可能需要等待其完成执行。这需要等待一定时间（等待时间），更有可能需要等待某个事件的出现（等待事件）。当事件出现时，作业或进程就会响应（响应事件）。第 6 章将深入讨论处理并发进程同步的系统调用。

另一组系统调用有助于调试程序。许多系统提供转储内存信息的系统调用。这有助于调试。程序 trace 在执行时能列出所用的每条执行的指令，但是只有少数几类系统提供。即使微处理器也提供一个称为单步的 CPU 模式，这种模式在每个指令运行后能执行一个陷阱。该陷阱通常为调试程序所用。

许多操作系统都提供程序的时间表，以表示一个程序在某个位置或某些位置执行所花的时间。时间表要求具有跟踪功能或定时时间中断。在每次出现定时中断时，会记录程序

计数器的值。如果有足够频繁的时间中断，就可得到程序各部分所用时间的统计数据。

进程或作业控制有许多方面和变化，下面将通过两个例子来解释这些概念——一个涉及单任务系统，而另一个涉及多任务系统。MS-DOS 操作系统是一个单任务系统的例子，它在计算机开始时就运行一个命令解释程序（见图 2.7(a)）。由于 MS-DOS 是单任务的，它采用了一个简单方法来执行程序且不创建新进程。它将程序装入内存，并改写它自己的绝大部分，以便为新程序提供尽可能多的空间（见图 2.7(b)）。然后它将指令指针设为程序的第一条指令。接着运行程序，或者一个错误会引起中断，或者程序执行一个系统调用以终止。不管怎样，错误代码会保存在系统内存中以便在后面使用。最后，命令解释程序中尚未改写的部分会重新开始执行。其首要任务是从磁盘中重新装入命令解释器的其他部分。当完成任务时，命令解释器会向用户或下一程序提供上一次的错误代码。

FreeBSD(源于 Berkeley UNIX)是多任务系统的一个例子。当用户登录到系统时，从用户所选择的 Shell 开始执行。这种 Shell 类似于 MS-DOS Shell，接受命令并执行用户所要求的程序。不过，由于 FreeBSD 是多任务系统，命令解释程序在另一个程序执行时可继续执行（图 2.8）。为了启动新进程，Shell 执行 `fork()` 系统调用。接着，所选择的程序通过 `exec()` 装入内存，程序开始执行。根据命令发布的方式，Shell 要么等进程完成，要么在后台执行进程。对于后一种情况，Shell 可马上接受下一个命令。当进程运行在后台时，它不能直接接受键盘输入，因为 Shell 在使用键盘。因此，I/O 通过文件或通过 GUI 接口完成。同时，用户可以让 Shell 执行其他程序，监视运行进程的状态，改变程序优先级，等等。当进程完成时，它执行 `exit()` 系统调用来终止，并将 0 或非 0 错误代码返回给调用进程。这一状态（或错误）代码可为 Shell 或其他程序所使用。第 3 章将通过一个使用 `fork()` 和 `exit()` 系统调用的程序例子来讨论进程问题。

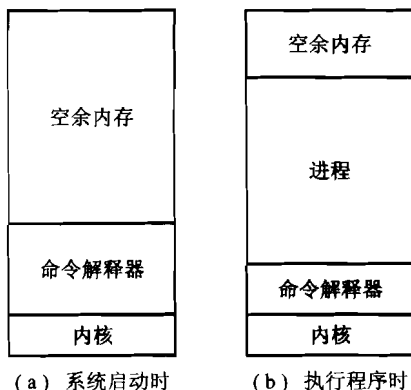


图 2.7 MS-DOS 执行状态

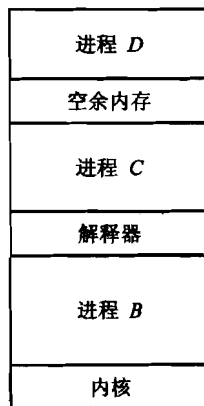


图 2.8 运行多个程序的 FreeBSD

Solaris 10 动态跟踪工具

使操作系统更容易理解、调试和调整是热门的研究方向,取得了一些成就。比如, Solaris 10 操作系统包含了 `dtrace` 动态跟踪工具,它可以动态探测运行的系统。可以用 D 语言来查询这些探测,从而确定数量惊人的内核、系统状态和进程活动信息。比如,图 2.9 所示为在一个应用程序执行系统调用 `ioctl` 的时候跟踪它,然后进一步显示内核在执行系统调用 `ioctl` 的时候进行的其他系统调用。以“U”结尾的行是在用户态执行的,以“K”结尾的行是在内核态执行的。

```
# ./all. d 'pgrep xclock' XEventQueued
Dtrace: script './all. d' matched 52377 probes
CPU FUNCTION
0 -> XEventQueued U
0 -> XEventQueued U
0 -> _XllTransBytesReadable U
0 <- _XllTransBytesReadable U
0 -> _XllTransSocketBytesReadable U
0 <- _XllTransSocketBytesReadable U
0 -> ioctl U
0 -> ioctl K
0 -> getf K
0 -> set_active_fd K
0 <- set_active_fd K
0 <- getf K
0 -> get_udatamodel K
0 <- get_udatamodel K
...
0 -> releasef K
0 -> clear_active_fd K
0 <- clear_active_fd K
0 -> cv_broadcast K
0 <- cv_broadcast K
0 <- releasef K
0 <- ioctl K
0 <- ioctl U
0 <- XEventQueued U
0 <- XEventQueued U
```

图 2.9 Solaris 10 内核中跟踪一个系统调用

其他操作系统开始包括各种性能和跟踪工具,这为许多机构的研究所支持(包括 Paradyne 项目)。

2.4.2 文件管理

在第 10 章和第 11 章,将深入讨论文件系统。不过,现在指出一些有关文件的常用系统调用。

首先需要能创建和删除文件。每个系统调用需要文件名,还可能也需要一些文件属性。一旦创建了文件后,就需要打开并使用它。也可能需要读、写或重定位(例如,倒回到或

跳到文件末尾)。最后, 需要关闭文件, 以表示不再使用它。

如果用目录结构来组织文件系统内的文件, 那么目录也需要同样的操作。另外, 不管是文件还是目录, 都需要能确定其属性, 或设置其属性。文件属性包括文件名、文件类型、保护模式、记账信息等。至少需要两个系统调用(读取文件属性和设置文件属性)完成这一功能。有的操作系统提供更多的调用, 如文件移动和复制。其余的, 一部分可能提供采用代码或系统调用完成这些操作的 API, 另一部分可能仅提供完成这些任务的系统程序。如果系统程序被其他程序所调用, 则其中每一个均可以被其他系统调用视为一个 API。

2.4.3 设备管理

程序在执行时需要用到一些资源才能继续运行, 例如内存、磁盘驱动、文件访问等。如果有可用资源, 那么系统允许请求, 控制应返回到用户程序; 否则, 程序必须等待可用的足够多的资源。

操作系统控制的不同的资源可当做设备看待, 这些设备有些是物理设备(如磁带), 而其他可当做抽象或虚拟的设备(如文件)。如果系统有多个用户, 那么用户必须请求设备以确保能独自使用它。在使用完设备之后, 用户需要释放它。这些函数类似于文件的打开和关闭(open 和 close)的系统调用。另一些操作系统允许不受管理的设备访问, 这带来的危害是潜在的设备争夺以及可能发生的死锁, 这将在第 7 章中讨论。

一旦请求了设备(并得到设备)之后, 就能如同对文件一样, 来对设备进行读、写、(可能)重定位。事实上, I/O 设备和文件非常相似, 以至于许多操作系统如 UNIX 将这两者合并为文件-设备结构。这时, 一套系统调用可用在文件和设备上。有时, I/O 设备可通过特殊文件名、目录位置或文件属性来表示。

即便内在的系统调用不同, UI 同样可以使文件和设备表现相似。这是为什么要深入研究操作系统和用户接口的另一个例子。

2.4.4 信息维护

许多系统调用只不过用于用户程序与操作系统之间传递信息。例如, 绝大多数操作系统都有一个系统调用以返回当前的时间和日期。其他系统调用可返回系统的其他信息, 如当前用户数、操作系统的版本、空闲内存或磁盘的多少等。

另外, 操作系统维护所有进程的信息, 有些系统调用可访问这些信息。一般来说, 也有系统调用用于设置进程信息(读取进程属性和设置进程属性)。在 3.1.3 小节, 将讨论通常需要维护什么信息。

2.4.5 通信

有两种通信模型: 消息传递模型和共享内存模型。对于消息传递模型(message-passing

model), 通信进程通过彼此之间交换消息来交换信息。直接或间接地通过一个共同的邮箱, 消息可以在进程之间得到交换。在通信前, 必须先打开连接。必须知道另一个通信实体的名称, 它可能是同一 CPU 的另一个进程, 也可能是通过与网络相连的另一计算机上的进程。网络上的每台计算机都有一个主机名, 这通常是已知的。同样, 主机也有一个网络标识, 如 IP 地址。类似地, 每个进程也有进程名, 它通常转换成标识符以便操作系统引用。系统调用 `get hostid` 和 `get processid` 用于这一转换。这些标识符再传递给文件系统提供的通用 `open` 和 `close` 系统调用, 或 `open connection` 和 `close connection` 系统调用, 这是由系统的通信模型决定的。接受方进程通常通过 `accept connection` 调用来允许通信。能接收连接的进程为特殊用户的后台程序, 这些程序是专用的系统程序。它们执行 `wait for connection` 调用, 当有连接时会被唤醒。通信源被称为客户机, 而接受方则被称为服务器, 通过 `read message` 和 `write message` 系统调用来交换消息。`close connection` 调用将终止通信。

对于共享内存模型 (shared-memory model), 进程使用 `shared memory create` 和 `shared memory attach` 系统调用来获得其他进程所拥有的内存区域的访问权。回想一下操作系统通常需要阻止一个进程访问另一个进程的内存。共享内存要求两个甚至多个进程都同意取消这一限制, 这样它们就可以通过读写公共区域来交换信息。数据的形式和位置是由这些进程来定的, 不受操作系统所控制。进程也负责确保它们不会同时向同一个地方写。这些机制将在第 6 章讨论。第 4 章将讨论进程模型的一种变形, 即线程, 一般来说它们共享内存。

上面讨论的两种模型在操作系统中都常用, 而且大多数系统两种都实现了。消息传递对交换少量数据很有用, 这是因为不必避免冲突。对于计算机间的通信, 它也比共享内存更容易实现。共享内存允许最大速度地通信并且十分方便, 这时因为当通信发生在计算机内, 它能以内存的速度进行。不过, 在保护和同步方面, 进程共享内存存在一些问题。

2.5 系统程序

现代系统的另一方面是一组系统程序。回顾一下图 1.1, 它描述了计算机逻辑层次。最底层是硬件, 上面是操作系统, 接着是系统程序, 最后是应用程序。系统程序提供了一个方便的环境, 以开发程序和执行程序。其中一小部分只是系统调用的简单接口, 其他的可能是相当复杂的。它们可分为如下几类:

- **文件管理:** 这些程序创建、删除、复制、重新命名、打印、转储、列出和操作文件和目录。
- **状态信息:** 一些程序从系统那里得到日期、时间、可用内存或磁盘空间的数量、用户数或类似状态信息。另一些更为复杂, 能提供详细的性能、登录和调试信息。通常, 这些信息经格式化后, 再打印到终端、输出设备或文件, 或在 GUI 的窗体上显示。有些系统还支持注册表, 它被用于存储和检索配置信息。

- **文件修改：**有多个编辑器可以创建和修改位于磁盘或其他存储设备上的文件内容。也可能有特殊的命令被用于查找文件内容或完成文本的转换。

- **程序语言支持：**常用程序设计语言（如 C、C++、Java、Visual Basic 和 Perl 等）的编译程序、汇编程序、调试程序和解释程序通常与操作系统一起提供给用户。

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

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

除系统程序外，绝大多数操作系统都提供程序以解决一般问题和执行一般操作。这些程序包括网页浏览器、字处理器和文本格式化器、电子制表软件、数据库系统、编译编译器、打印和统计分析包以及游戏。这些程序称为**系统工具或应用程序**。

绝大多数用户所看到的操作系统是由应用和系统程序而不是系统调用所决定的，例如 PC 的使用。当计算机运行 Mac OS X 操作系统时，用户可能看到 GUI、鼠标或窗口接口。另一种可能是，在 GUI 的某个窗体上，计算机可能有一个命令行 UNIX Shell。两者都使用同样集合的系统调用，但是系统调用看起来不同并且其动作也不同。

2.6 操作系统设计和实现

本节讨论设计和实现系统时所遇到的一些问题。虽然对这些问题没有完整的解决方案，但是有些方法还是很成功的。

2.6.1 设计目标

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

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

用户要求一些明显的系统特性：系统应该方便和容易使用、容易学习、可靠、安全和快速。当然，这些规格对于系统设计并不特别有用，因为人们并没有在如何达到这些目标上达成一致的意見。

设计、创建、维护和操作系统的有关人员可以定义另一组要求：操作系统应该容易设计、实现和维护，也应该灵活、可靠、高效且没有错误。显然，这些要求在系统设计时并不明确，并可能产生不同的理解。

总之，关于定义操作系统的要求，没有一个唯一的解决方案。现实中存在许多类型的系统，说明了不同要求能形成对不同环境的解决方案。例如，一种嵌入式系统的实时操作系统 VxWorks 的要求与用于 IBM 大型机的多用户、多访问操作系统 MVS 的要求差别很大。

操作系统的规格和设计属于高度创造性工作。虽然没有教科书会告诉你如何去做，但是一般的软件工程原理还是适用的。现在就来讨论这些规则。

2.6.2 机制与策略

一个重要原理是**策略**（policy）和**机制**（mechanism）的区分。机制决定如何做，策略决定做什么。例如，定时器结构（参见 1.5.2 小节）是一种 CPU 保护的机制，但是对于特定用户将定时器设置成多长时间是个策略问题。

策略和机制的区分对于灵活性来说很重要。策略可能会随时间或位置而有所改变。在最坏情况下，每次策略改变都可能需要底层机制的改变。系统更需要通用机制，这样策略的改变只需要重定义一些系统参数。例如，考虑一种赋予某种程序相对于其他程序的优先级的机制。如果该机制正确地与策略区分开来，则它可以用来支持 I/O 密集型程序应该比 CPU 密集型程序有更高的优先级的策略，亦或支持相反的策略。

通过实现一组基本且简单的构造块，微内核操作系统（参见 2.7.3 小节）把机制与策略的区分利用到了极致。这些块与策略无关，允许通过用户创建的内核模块或用户程序本身来增加更高级的机制和策略。例如，想一想 UNIX 的历史，刚开始时，它实行分时调试，而到最新的 Solaris 版本，调度由可加载的表控制。根据当前加载的表，系统可以是分时的、批处理的、实时的或公平分配的。制定调试机制的目的在于通过单个 `load-new-table` 命令得到巨大的策略改变。另一种极端如 Windows 系统，其中机制和策略在系统中被编码以形成统一的系统风格。所有应用程序都有类似的接口，因为接口本身已在内核和系统库中构造。Mac OS X 操作系统也有类似的功能。

策略决定对所有的资源分配都很重要。无论何时，只要决定是否分配资源，就必须做出策略决定。只要问题是“如何做”而不是“是什么”，这就必须要由机制决定。

2.6.3 实现

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

第一个不是用汇编语言编写的系统可能是用于 Burroughs 计算机的主控程序（MCP）。MCP 是采用 ALGOL 语言编写的，MIT 开发的 MULTICS 主要是用 PL/I 语言来编写的。Linux 和 Windows XP 操作系统主要是用 C 语言编写的，有少数主要用于设备驱动程序与保存和恢复寄存器状态的代码是用汇编语言来编写的。

使用高级语言或至少是系统实现语言来实现操作系统，可以得到与用高级语言来编写

应用程序同样的优点：代码编写更快，更为紧凑，更容易理解和调试。另外，编译技术的改进使得只要通过重新编译就可改善整个操作系统的生成代码。最后，如果用高级语言来编写，操作系统将更容易移植到不同的平台上。例如，MS-DOS 是用 Intel 8088 汇编语言编写的，因而只能用于 Intel 类型的 CPU。Linux 操作系统主要是用 C 语言来编写的，可用于许多不同的 CPU，如 Intel 80x86、Motorola 680x0、SPARC 和 MIPS RX000 等。

用高级语言来实现操作系统的缺点仅仅在于降低了速度和增加了存储要求，但这对当今的系统不再是主要问题。虽然汇编语言高手能编写更快、更小的子程序，但是现代编译器能对大程序进行复杂的分析并采用高级优化技术以生成优良代码。现代处理器都有很大的流水线和多个功能单元块，它们能处理复杂相关性，这些是人类的有限思维能力所难以处理的。

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

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

2.7 操作系统结构

对于像现代操作系统这样庞大而复杂的系统，为了能正常工作并能容易修改，必须认真设计。通常方法是这一任务分成小模块而不只是一个单块系统。每个这样的模块都应该是明确定义的系统部分，且具有明确定义的输入、输出和功能。在第 1 章中已简要讨论了操作系统的常用模块，本节讨论这些模块如何连接起来以组成内核。

2.7.1 简单结构

许多商业系统没有明确定义的结构。通常，这些操作系统最初是较小、简单且功能有限的系统，但后来渐渐超过了其原来的范围。MS-DOS 就是一个这样的操作系统。它最初是由几个人设计和实现的，当时并没有想到它会如此受欢迎。由于它主要是利用最小的空间提供最多的功能，因此它并没有被仔细地划分成模块。图 2.10 显示了其结构。

在 MS-DOS 系统中，并没有很好地区分接口和功能层次。例如，应用程序能够访问基本的 I/O 子程序，直接写到显示器和磁盘驱动程序中。这种任意性使 MS-DOS 易受错误（或

恶意) 程序的伤害, 从而导致用户程序出错时整个系统的崩溃。当然, MS-DOS 还受限于同时代的硬件, Intel 8088 未能提供双模式和硬件保护, MS-DOS 设计者除了允许基本的硬件的访问外, 没有其他选择。

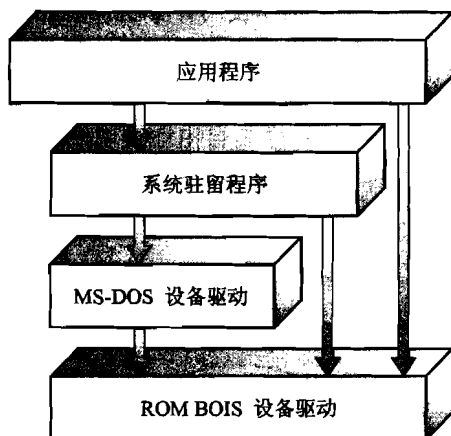


图 2.10 MS-DOS 层次结构

另一个受限结构的例子是原始的 UNIX 操作系统。UNIX 是另一个最初受到硬件功能限制的系统。它由内核和系统程序两个独立部分组成。内核进一步分成为一系列接口和驱动程序, 这些年随着 UNIX 的发展, 这些程序被不断地增加和扩展。可以将传统的 UNIX 操作系统分层来研究。如图 2.11 所示, 物理硬件之上和系统调用接口之下的所有部分作为内核。内核通过系统调用以提供文件系统、CPU 调度、内存管理和其他操作系统功能。总的来说, 这一层里面组合了大量的功能。这种单一式结构使得 UNIX 难以增强。

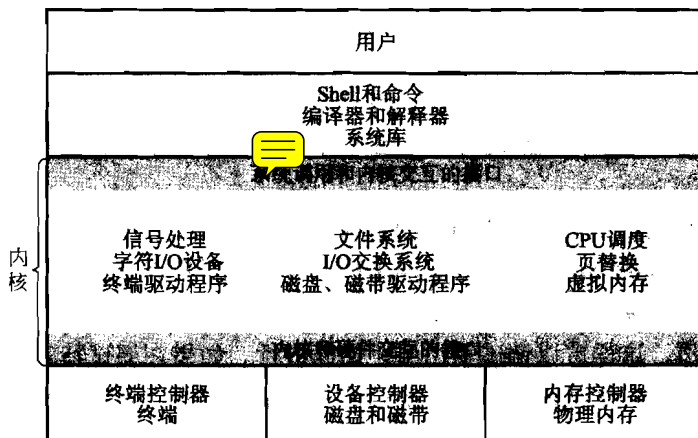


图 2.11 UNIX 系统结构

2.7.2 分层方法

采用适当硬件支持,操作系统可以分成比原来 MS-DOS 和 UNIX 所允许的更小和更合适的模块。这样操作系统能提供对计算机和使用计算机的应用程序更多的控制。实现人员能更加自由地改变系统的内部工作和创建模块操作系统。采用自顶向下方法,可先确定总的功能和特征,再划分成模块。隐藏信息同样很重要,因为它在保证子程序接口不变和子程序本身执行其功能的前提之下,允许程序员自由地实现低层函数。

系统模块化有许多方法。一种方法是**分层法**,即操作系统分成若干层(级)。最底层(层 0)为硬件,最高层(层 N)为用户接口。这种分层结构如图 2.12 所示。

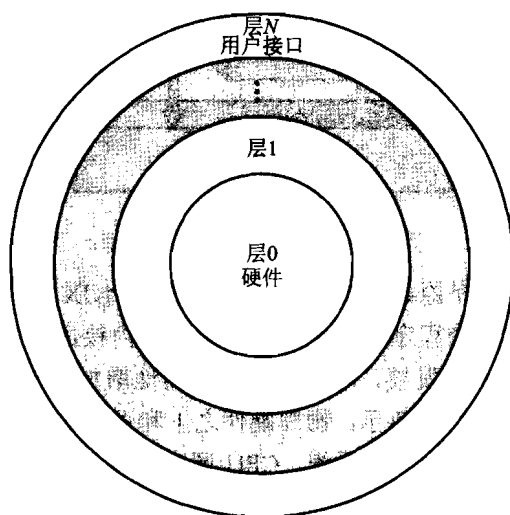


图 2.12 一种分层操作系统

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

分层法的主要优点在于构造和调试的简单化。每层只能利用较低层的功能和服务。这种方法简化了调试和系统验证。第一层能先调试而不需要考虑系统其他部分,因为根据定义它只使用了基本的硬件(通常认为是正确的)来实现其功能。在第一层调试之后,可以认为它已正确运作,这样可以调试第二层,如此进行。如果在调试特定层时发现了错误,那么错误必然在该层,这是因为其低层都已调试好了。因此,系统分层简化了系统的设计和实现。

每层都是利用较低层所提供的功能来实现的。该层不必知道如何实现这些操作,它只

需要知道这些操作能做什么。因此，每层为较高层隐藏了一定的数据结构、操作和硬件的存在。

分层法的主要困难涉及对层的详细定义，这是因为一层只能使用其下的较低层。例如，用于备份存储的设备驱动程序（虚拟内存算法所使用的磁盘空间）必须位于内存管理子程序之下，因为内存管理需要能使用磁盘空间。

其他则要求并不十分明显。备份存储驱动程序通常在 CPU 调用程序之上，因为该程序需要等待 I/O 完成，并在这段时间内可以重新调度 CPU。不过，对于大型系统，CPU 调度程序会有更多关于适合在内存中的活动进程的信息。因此，这些信息需要换入和换出内存，从而要求备份存储驱动程序位于 CPU 调度之下。

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

这些限制在近年来引起了分层法的略微倒退。现在使用数量更少而功能更多的分层设计，提供了绝大多数模块化代码的优点，同时避免了层定义和交互的问题。

2.7.3 微内核

随着 UNIX 操作系统的扩充，内核变得更大且更难管理。在 20 世纪 80 年代中期，卡内基-梅隆大学的研究人员开发了一个称为 Mach 的操作系统，该系统采用微内核（microkernel）技术来模块化内核。这种方法将所有非基本部分从内核中移走，并将它们实现为系统程序或用户程序。这样得到了更小的内核。关于哪些应保留在内核内，而哪些应在用户空间内实现，并没有定论。不过，微内核通常包括最小的进程和内存管理以及通信功能。

微内核的主要功能是使客户程序和运行在用户空间的各种服务之间进行通信。通信以消息传递形式提供，参见 2.4.5 小节。例如，如果客户程序希望访问一个文件，那么它必须与文件服务器进行交互。客户程序和服务器决不会直接交互，而是通过微内核的消息传递来通信。

微内核方法的好处之一在于便于扩充操作系统。所有新服务可以在用户空间增加，因而并不需要修改内核。当内核确实需要改变时，所做的改变也会很小，因为微内核本身很小。这样的操作系统很容易从一种硬件平台设计移植到另一种硬件平台设计。由于绝大多数服务是作为用户而不是作为内核进程来运行的，因此微内核也就提供了更好的安全性和可靠性。如果一个服务器出错，那么操作系统其他部分并不受影响。

许多现代操作系统使用了微内核方法。Tru64 UNIX（前身是 Digital UNIX）向用户提

供了 UNIX 接口，但是它是用 Mach 微内核来实现的。Mach 微内核将 UNIX 系统调用映射成对适当用户层服务的消息。

另一个例子是 QNX。QNX 也是一个基于微内核设计的实时操作系统。QNX 微内核提供了消息传递和进程调度服务。它也处理低层网络通信和硬件中断。所有 QNX 的其他服务是通过标准进程（运行在内核之外的用户模式）提供的。

遗憾的是，微内核必须忍受由于系统功能总开销的增加而导致系统性能的下降。回顾一下 Windows NT 的历史。它的第一个版本具有分层的微内核组织，不过，其性能要比 Windows 95 差。Windows NT 4.0 通过将有些层从用户空间移到内核空间，以及更紧密地集成这些层来提高性能。到设计 Windows XP 时，它更像是单一内核而不是微内核。

2.7.4 模块

也许最新的操作系统设计方法是用面向对象编程技术来生成模块化的内核。这里，内核有一组核心部件，以及在启动或运行时对附加服务的动态链接。这种方法使用动态加载模块，并在现代的 UNIX，如 Solaris、Linux 和 Mac OS X 中很常见。例如，如图 2.13 所示的 Solaris 操作系统结构被组织为 7 个可加载的内核模块围绕一个核心内核构成：

- ① 调度类。
- ② 文件系统。
- ③ 可加载的系统调用。
- ④ 可执行格式。
- ⑤ STREAMS 模块。
- ⑥ 杂项模块。
- ⑦ 设备和总线驱动。

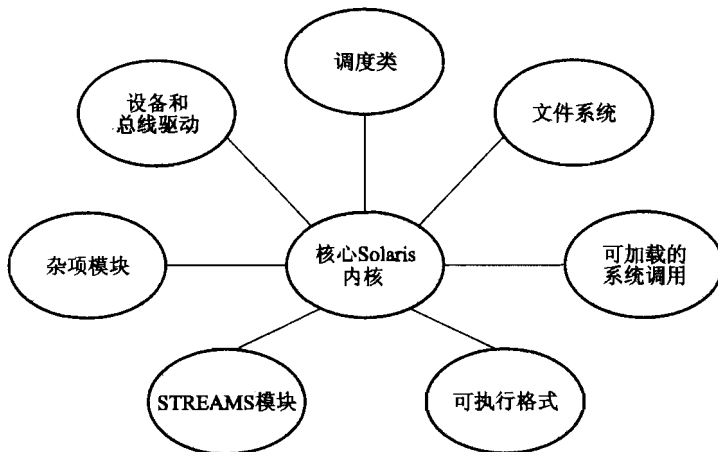


图 2.13 可加载的 Solaris 模块

这样的设计允许内核提供核心服务，也能动态地实现特定的功能。例如，特定硬件的设备和总线驱动程序可以加载给内核，而对各种文件系统的支持也可作为可加载的模块加入其中。所得到的结果就好像一个分层系统，它的每个内核部分都有被定义和保护接口。但它比分层系统更为灵活，它的任一模块都能调用任何其他模块。进一步讲，这种方法类似于微内核方法，核心模块只有核心功能以及其他模块加载和通信的相关信息，但这种方法更为高效，因为模块不需要调用消息传递来通信。

苹果 Mac OS X 操作系统采用一种混合结构。Mac OS X（也被称为 Darwin）采用分层技术构建操作系统，其中一层包括 Mach 微内核。Mac OS X 结构如图 2.14 所示。

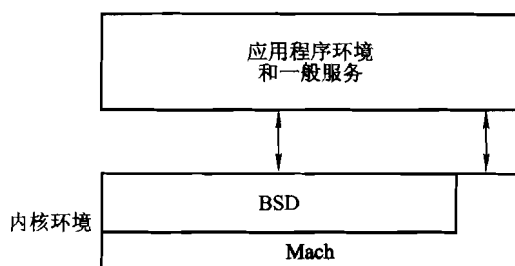


图 2.14 Mac OS X 结构

上面的层包括应用环境和一组向应用提供的图形接口的服务。它们的下面是内核环境，主要包括 Mach 微内核和 BSD 内核。Mach 提供内存管理，支持远程程序调用（RPC）和进程间通信（IPC）工具，包括消息传递和线程调度。而 BSD 提供了 BSD 命令行接口，支持网络和文件系统，以及 POSIX API 的实现，包括 Pthread。除 Mach 和 BSD 外，内核环境为设备驱动的开发和动态加载模块（Mac OS X 中指的是内核扩展）提供一个 I/O 工具。正如图 2.14 所示，应用和公共服务可以直接使用 Mach 或 BSD 方法。

2.8 虚拟机

2.7.2 小节介绍的分层方法逻辑可延伸为虚拟机概念。虚拟机的基本思想是单个计算机（CPU、内存、磁盘、网卡等）的硬件抽象为几个不同的执行部件，从而造成一种“幻觉”，仿佛每个独立的执行环境都在自己的计算机上运行一样。

通过利用 CPU 调度（参见第 5 章）和虚拟内存技术（参见第 9 章），操作系统能带来一种“幻觉”，即进程认为有自己的处理器和自己的（虚拟）内存。当然，进程通常还有其他特征，如系统调用和文件系统，这些是硬件计算机所不能提供的。而虚拟机方法除了提供了与基本硬件相同的接口之外，并不提供额外功能。每个进程都有一个与基本计算机一样的（虚拟）副本（见图 2.15）。

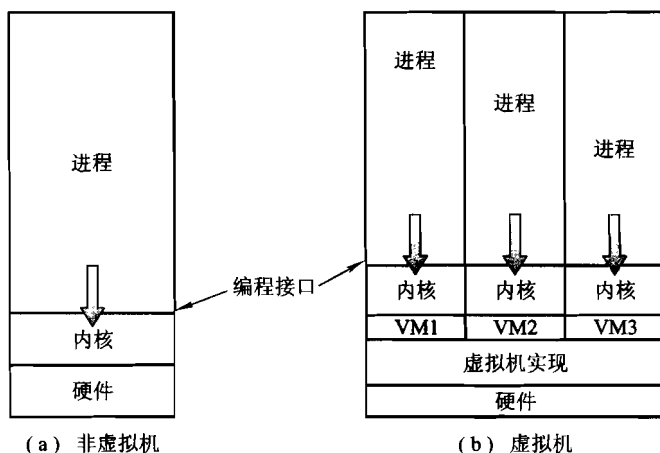


图 2.15 系统模型

创建虚拟机有几个原因，最根本的是，在并行运行几个不同的执行环境（即不同的操作系统）时能够共享相同的硬件。之后将在 2.8.2 小节更详细地学习虚拟机的优点。本小节将讨论 IBM 系统的 VM 操作系统，因为它提供了一个有用的例子，而且 IBM 首创了这方面的工作。

虚拟机方法的主要困难与磁盘系统有关。假设物理机器有 3 个磁盘驱动器但是要提供 7 个虚拟机。显然，它不能为每个虚拟机分配一个磁盘驱动器，因为虚拟机软件本身需要一定的磁盘空间以提供虚拟内存。解决方法是提供虚拟磁盘（在 IBM 的 VM 操作系统中被称为小型磁盘（minidisk），它们除了大小外，在其他各方面都相同。系统通过在物理磁盘上为小型磁盘分配所需要的磁道数以实现小型磁盘。显然，所有小型磁盘的大小的总和必须比可用的物理磁盘要小。

因此用户拥有自己的虚拟机后，他们能运行原来机器所具有的任何操作系统或软件包。对 IBM VM 系统来说，用户通常运行 CMS——一种单用户交互操作系统。虚拟机软件主要在一个物理机器上多道运行多个虚拟机，但并不需要考虑任何用户支持软件。这种安排将提供把多用户交互系统分为两个更小部分的一种有效的划分。

2.8.1 实现

虽然虚拟机概念有用，但是实现困难。提供与底层机器完全一样的副本需要做大量的工作。底层机器有两种模式：用户模式和内核模式。虚拟机软件可以运行在内核模式，因为它就是操作系统。虚拟机本身只能运行在用户模式。正如物理机器有两种模式一样，虚拟机也有两种模式。因此，必须有虚拟用户模式和虚拟内核模式，这两种模式都运行在物理用户模式。在真正机器上引起从用户模式到内核模式转换的动作（如系统调用或试图执

行特许指令)也必须在虚拟机上引起从虚拟用户模式到虚拟内核模式的转换。

这种转换可按下述方法实现。例如,当一个以虚拟用户模式而在虚拟机上运行的程序执行系统调用时,它会在真实机器上引起一个到虚拟机监控器的转换。当虚拟机监控器获得控制,它能改变虚拟机的寄存器内容和程序计数器以模拟系统调用的效果。接着它能重新启动虚拟机,注意它现在是在虚拟内核模式下执行。

当然,主要的差别是时间。虽然真正 I/O 可能需要 100 ms,但是虚拟 I/O 可能需要更少的时间(因为脱机操作)或更多时间(因为解释执行)。另外,CPU 在多个虚拟机间多道执行,也会使得虚拟机按不可预计的方式慢下来。在极端情况下,可能需要模拟所有指令以提供真正的虚拟机。VM 能在 IBM 机器上工作,这是因为虚拟机的通常指令能直接在硬件上执行。只有特许指令(主要用于 I/O)必须模拟,因而执行更慢。

2.8.2 优点

虚拟机的理念具有许多优点。注意,在这样的环境中,不同的系统资源具有完全的保护。每个虚拟机完全独立于其他虚拟机,因此没有安全问题,但同时也没有直接资源共享。提供共享有两种实现方法。第一,可以通过共享小型磁盘来共享文件,这种方案模拟了共享物理磁盘,但通过软件实现。第二,可以通过定义一个虚拟机的网络,每台虚拟机通过虚拟通信网络来传递消息。同样,该网络是按物理通信网络来模拟的,但是通过软件实现的。

这样的虚拟机系统是用于研究和开发操作系统的好工具。通常,修改操作系统是一项艰难的任务。因为操作系统程序庞大且复杂,所以改变一处可能会在另一处产生未知的错误。操作系统的威力使得这种情况尤为危险。由于操作系统工作在内核模式,一个指针的错误变化可能会引起足以破坏整个文件系统的错误。因此,有必要仔细检查所有操作系统的改变。

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

虚拟机系统可能基本取消这个问题。系统程序有自己的虚拟机,系统开发可在虚拟机而不是真实的物理机器上进行。正常系统操作无须进行中断来开发系统。

2.8.3 实例

尽管虚拟机有这么多优点,但它在第一次开发的很多年后都没引起注意。但是现在,虚拟机成为一种解决系统兼容性的流行方法。本节介绍两种同时流行的虚拟机:VMware 和 Java 虚拟机。正如将要看到的,这些虚拟机运行在前面所述的任何设计类型的操作系统之上。因此,操作系统设计方法(单层的、微内核的、模块化的、虚拟机的)并不相互

排斥。

1. VMware

VMware 是一个流行的商业应用程序，它将 Intel 80x86 硬件抽象为独立的虚拟机。VMware 作为一种应用程序运行在主操作系统如 Windows 或 Linux 之上，并允许主操作系统将几个不同的客户操作系统作为独立的虚拟机来并行地运行。

考虑下面的情形：一个开发人员设计了一个应用程序，并希望能在 Linux、FreeBSD、Windows NT 和 Windows XP 上测试。对她而言，一种选择是获取 4 个不同的计算机，每个都运行这些操作系统中的一个副本。另一种选择，她首先在计算机系统中装一个 Linux 系统并测试应用程序，然后装上 FreeBSD 并测试应用程序，如此继续。这个选择允许她使用同样的物理计算机，但是却更耗费时间，因为她必须为每一次测试装上新的操作系统。而这样的测试可以使用 VMware 在同样的物理计算机上并行完成。此时，程序员可以在主操作系统和三个客户操作系统上测试应用程序，每个客户操作系统都作为一个独立的虚拟机运行。

这种系统结构如图 2.16 所示。此时，Linux 作为主操作系统运行，FreeBSD、Windows NT 和 Windows XP 作为客户操作系统运行。虚拟层是 VMware 的核心，因为它将物理硬件抽象为独立的作为客户操作系统的虚拟机运行。每个虚拟机都有它自己的虚拟 CPU、内存、磁盘驱动、网络接口等。

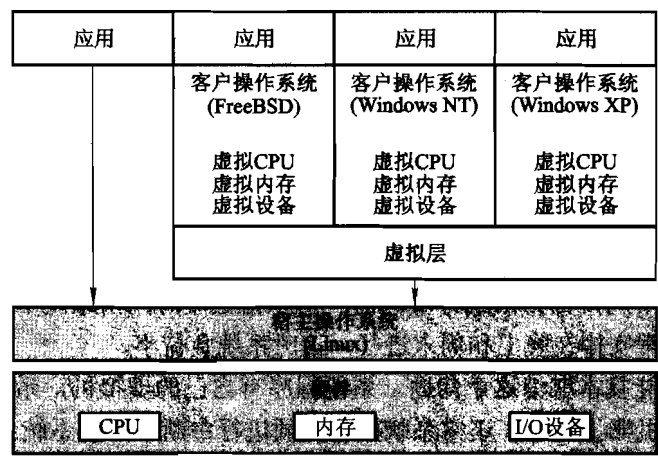


图 2.16 VMware 结构

2. Java 虚拟机

Java 是由 Sun Microsystems 公司在 1995 年后推出的一种深受欢迎的面向对象语言。除了其语言规范和大量 API 库，Java 还提供了 **Java 虚拟机 (JVM)**。

Java 对象用类结构来描述；Java 程序由一个或多个类组成。对于每个 Java 类，Java 编译器会生成与平台无关的字节码（bytecode）输出文件（.class），它可运行在任何 JVM 上。

JVM 是一个抽象计算机的规范。它包括类加载器和执行与平台无关的字节码的 Java 解释器，如图 2.17 所示。类加载器从 Java 程序和 Java API 中加载编译过的.class 文件，以便为 Java 解释器所执行。在装入类后，验证器会检查.class 文件是否为有效的 Java 字节代码，有无堆栈的溢出和下溢。它也确保字节代码不进行指针操作，因为这可能会提供非法内存访问。如果类通过验证，那么就可为 Java 解释器所执行。JVM 通过执行垃圾收集（garbage collection，回收不再使用的内存并返回给系统）来自动管理内存。为了提高虚拟机中 Java 程序的性能，许多研究集中在垃圾收集算法上。

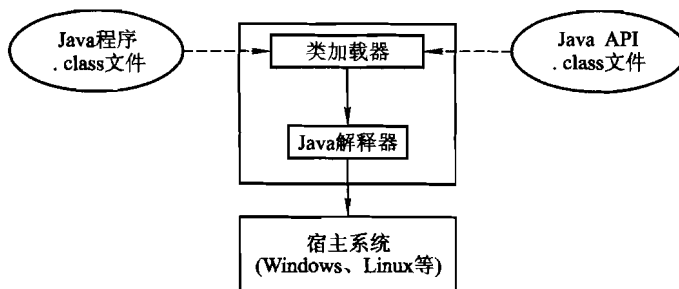


图 2.17 Java 虚拟机

JVM 可以在主操作系统如 Windows、Linux、Mac OS X 的上层软件中实现，或作为 Web 浏览器的一部分。另一个选择是 JVM 可以在特别为 Java 程序设计的芯片硬件上实现。如果在软件上实现 JVM，Java 解释程序一次只能执行一个字节代码。一种更快的软件技术是采用 JIT（just-in-time）编译器。第一次调用 Java 方法时，该方法的字节码被转换成主机的本地机器语言。然后，这些操作被隐藏起来，以使随后的调用通过采用本地机器指令和字节码操作来完成，而不再需要全部重新解释一次。有一种技术能使 JVM 在芯片硬件上运行得更快，它将 Java 字节码操作当做本地码来执行，从而不需要软件解释程序或 just-in-time 编译程序。

.NET 框架

.NET 框架是一套包含了类库集合、执行环境和软件开发平台的技术。这个平台允许基于 .NET 框架编程而不是针对任何特定平台。基于 .NET 框架编写的程序不需要担心平台特性或者底层操作系统的特性。因此，任何实现了 .NET 的体系结构都可以成功执行 .NET 程序。这是因为运行环境对这些细节进行了抽象，提供了一个介于底层体系结构和被运行的程序之间的虚拟机。

.NET 框架的核心是公共语言运行时间 (CLR)。CLR 是 .NET 虚拟机的实现。它提供了运行任何 .NET 语言编写程序的环境。用 C# 或者 VB.NET 编写的程序被编译为一种平台无关的中间语言 (叫做微软中间语言 MS-IL)。这些被编译好的文件叫做组合 (assembly)，它包含了 MS-IL 指令和元数据。它们的文件名后缀是 .dll 或者 .exe。当要运行这些程序的时候，CLR 把这些组合加载进应用程序域 (Application Domain)。当程序要求执行指令的时候，CLR 把 MS-IL 指令即时编译为特定的底层体系结构的本地代码。一旦指令被编译为本地代码，它们就被保存下来以便 CPU 执行。 .NET 框架的 CLR 结构见图 2.18。

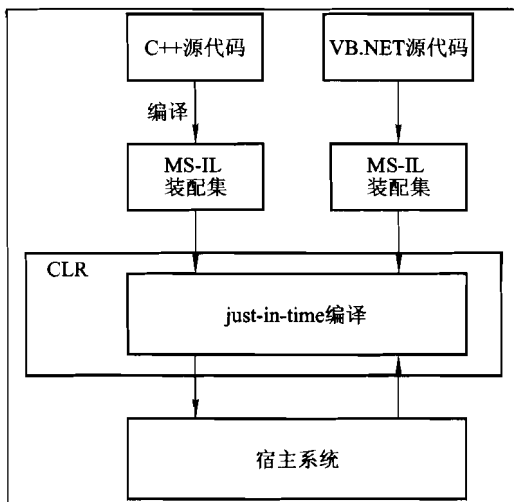


图 2.18 .NET 框架的 CLR 结构

2.9 系统生成

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

操作系统通常通过磁盘或 CD-ROM 来发布。为了生成系统，可以使用一个特殊程序。SYSGEN 程序从给定文件读取，或询问系统操作员有关硬件系统的特定配置，或直接检测硬件以决定有什么部件。下面几类信息必须要确定下来。

- 使用什么 CPU？安装什么选项（扩展指令集，浮点操作等）？对于多 CPU 系统，必须描述每个 CPU。

- 有多少可用内存？有的系统通过对内存位置一个个地访问直到出现非法地址的方法确定这一值。该过程定义了最后合法地址和可用内存的数量。

- 有什么可用设备？系统需要知道如何访问这些设备（设备号码）、设备中断号、设备类型和模型以及任何特别设备的特点。

- 需要什么操作系统选项或使用什么参数值？这些选项包括需要使用多少和多大的缓冲，需要什么类型的 CPU 调度算法，所支持进程的最大数量是多少，等等。

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

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

对于另外一种极端情况，可以构造完全由表驱动的系统。所有代码都是系统的组成部分，选择发生在执行时而不是在编译或连接时。系统生成只是创建适当的表以描述系统。绝大多数现代操作系统按这种方式来构造。

这些方法的主要差别是所生成系统的大小和通用性，以及因硬件配置变化所进行修改的方便性。考虑一下修改系统以支持新图形终端和另一个磁盘驱动器的代价。当然，与该代价相对的是这些改变的频率。

2.10 系统启动

在生成操作系统之后，它必须要为硬件所使用。但是硬件如何知道内核在哪里，或者如何装入内核？装入内核以启动计算机的过程称为引导系统。绝大多数计算机系统都有一小块代码，它称为引导程序或引导装载程序。这段代码能定位内核，将它装入内存，开始执行。有的计算机系统，如个人计算机，采用两步完成：一个简单的引导程序从磁盘上调入一个较复杂的引导程序，而后者再装入内核。

当 CPU 接收到一个重置事件时，例如它被加电或重新启动，具有预先定义内存位置的指令寄存器被重新装载，并在此开始执行。该位置就是初始引导程序的所在。该程序为只读存储器（ROM）形式，因为系统启动时 RAM 处于未知状态。由于不需要初始化和不受计算机病毒的影响，用 ROM 是很方便的。

引导程序可以完成一系列任务。通常，一个任务要运行诊断程序来确定机器的状态。如果诊断通过，程序可按启动步骤继续进行。系统的所有部分都可以被初始化，从 CPU 寄存器到设备控制器，以及内存的内容。最后，操作系统得以启动。

有些系统（如手机、PDA 和游戏控制台）在 ROM 中保存完整的操作系统。在 ROM 中存储完整的操作系统特别适合小型操作系统，它支持简单的硬件和操作。该方法存在的一个问题是，改变引导程序代码需要改变 ROM 芯片。有些系统通过使用可擦写只读存储器（EPROM）来解决这个问题，它是一个只读存储器，只有当明确给定一个命令时才会变为可写。所有形式的 ROM 都是固件，因为它们的特征介于硬件与软件之间。通常，固件存在的问题是在此执行代码比在 RAM 中慢。有些系统将操作系统存储在固件中，并将之复制在 RAM 中以获得更快的执行速度。固件的最后一个问题是它相对比较贵，所以通常用得很少。

对大型操作系统（包括大多数通用的操作系统，如 Windows、Mac OS X 和 UNIX）或经常改变的系統，引导程序被存储在固件中，而操作系统保存在磁盘上。此时，引导程序运行诊断程序，它具有能够从磁盘固定位置（0 区块）读取整块信息到内存的代码，并从引导块执行代码。存储在引导块的程序多半足够复杂，可以将一个完整的操作系统装载到内存并开始执行。更为典型的是，代码很简单（适合于单磁盘区块），仅仅知道在磁盘上的地址以及引导程序余下的长度信息。所有这些磁盘绑定的引导程序和操作系统本身可以通过向磁盘写入新的版本，从而很容易地进行改变。具有引导分区（详见 12.5.1 小节）的磁盘被称为引导磁盘或系统磁盘。

既然完全的引导程序已被装入，它可以扫描文件系统以找到操作系统内核，将之装入内存，启动并执行。只有到了这个时候才能说系统开始运行了。

2.11 小 结

操作系统提供若干服务。在最底层，系统调用允许运行程序直接向操作系统发出请求。在高层，命令解释程序或 Shell 提供了一个机制以使用户不必编写程序就能发出请求。命令可以来自文件（批处理模式），或者直接来自键盘输入（交互模式或分时模式）。系统程序用来满足一些常用用户操作。

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

在定义了系统服务之后，就可开发操作系统的结构。需要用各种表记录一些信息，这些信息定义了计算机的系统状态和系统的作业状态。

设计一个新操作系统是一项重大任务。在设计开始之前，定义好系统目标是很重要的。系统设计的类型是作为选择各种必要算法和策略的基础。

由于操作系统大，所以模块化很重要。按一系列层或采用微内核来设计系统是比较好

的技术。虚拟机概念采用了分层方法，并将操作系统内核和硬件都作为硬件来考虑。其他操作系统可以建立在这一虚拟机之上。

实现 JVM 的任何操作系统能运行所有 Java 程序，因为 JVM 为 Java 程序抽象化了底层系统，以提供平台无关接口。

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

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

为了运行计算机系统，必须初始化 CPU 和在固件系统中启动执行引导程序。如果操作系统也在固件系统中，引导程序可以直接启动操作系统。否则，它必须完成这样一道程序：逐步地从固件或磁盘装载更聪明的程序，直到操作系统本身被装入内存并执行。

习 题

2.1 操作系统提供的服务和功能可以主要分为两大类。简要描述这两大类并讨论它们的区别。

2.2 列出操作系统提供使用户更为方便地使用计算机系统的 5 个服务，并说明在哪些情况下用户级程序不能够提供这些服务。请解释为什么。

2.3 给出三种向操作系统传递参数的常用方法。

2.4 介绍一下如何获得一个程序在执行其不同部分的代码时所耗时间的统计简表。讨论获得该统计简表的重要性。

2.5 操作系统关于文件管理的 5 个主要功能是什么？

2.6 操作文件和设备时，采用同样的系统调用界面有什么优点和缺点？

2.7 命令解释器的用途是什么？为什么它经常是与内核分开的？是否可能采用操作系统提供的系统调用接口为用户开发一个新的命令解释器？

2.8 进程间通信的两个模式是什么？这两种方法有何长处和缺点？

2.9 为什么要将机制和策略区分开来？

2.10 为什么 Java 提供从 Java 程序调用以 C 或 C++编写的本地方法？举出一个本地方法的例子。

2.11 如果操作系统的两个部件相互依赖，有时实现分层方法会很困难。请区别两个功能紧密耦合的系统部件如何分层。

2.12 系统设计采用微内核设计的主要优点是什么？用户程序和系统服务在微内核结构内如何相互影响？采用微内核设计的缺点又是什么？

2.13 模块化内核方法和分层方法在哪些方面类似？哪些方面不同？

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

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

2.16 在 VMware 这样的系统中，客户操作系统与主操作系统有什么关系？选择主操作系统要考虑什么因素？

2.17 实验性的 Synthesis 操作系统在内核里有一个汇编器。为了优化系统调用的性能，内核通过在

内核空间内汇编程序来缩短系统调用必须经过的途径。这是一种与分层设计相对立的设计，经过内核的途径在这种设计中被延伸了，使操作系统的建立更加简单。分别从支持和反对的角度来讨论这种 Synthesis 设计方式对内核设计和性能优化的影响。

2.18 在 2.3 小节中，介绍了一个从一个文件向一个目标文件复制内容的程序。这个程序首先提示用户输入源文件和目标文件的名称。用 Win32 或 POSIX 的 API 写出这个 C 程序，并确信包括了所有必需的错误检测和文件存在的保证。一旦你正确地设计并测试了此程序，如果用一个系统来支持它，采用跟踪系统调用的工具来运行它。Linux 系统提供了 ptrace 工具，而 Solaris 系统则采用 truss 或 dtrace 命令。在 Mac OS X 中，dtrace 工具提供了类似的功能。

项目：向 Linux 内核增加一个系统调用

在此项目中，你将学习 Linux 操作系统提供的系统调用接口，以及一个用户程序如何通过该接口与操作系统内核实现通信。你的任务是将一个新的系统调用加入内核中，然后扩展该操作系统的功能。

1. 开始

用户模式过程调用通过堆栈或寄存器传递参数给被调用的过程来完成，保存当前的状态和程序计数器值，跳至与被调过程相对应的编码的开始部分。进程像以前一样继续拥有相同的特权。

对用户程序而言，系统调用就像过程调用一样，但在执行上下文和特权方面有所改变。在 Intel 386 结构的 Linux 中，系统调用通过将系统调用号存储在 EAX 寄存器中，将参数存储在另一个硬件寄存器中，并执行一个陷阱指令（即为 INT 0x80 汇编指令）来完成。陷阱执行后，系统调用号被用做一个代码指针表的索引，以获得执行系统调用的名柄号的开始地址。然后进程跳到该地址，进程的特权也从用户模式转为内核模式。得到扩展的特权的进程现在可以执行内核代码了，包括不能在用户模式下执行的特权指令。之后内核代码就可以完成与 I/O 设备交互等服务请求，以及完成进程管理和其他不能在用户模式下完成的活动。

Linux 内核最新版本的系统调用号列在 /usr/src/linux-2.x/include/asm-i386/unistd.h 下（如对应于系统调用 close() 的 _NR_close，它被用来调用关闭文件描述符，被定义为值 6）。系统调用句柄的指针列表一般存储在文件 /usr/src/linux-2.x/arch/i386/kernel/entry.S 的 ENTRY (sys_call_table) 下。请注意在表中 sys_close 被保存在 entry number 为 6 之处，以与在文件 unistd.h 中定义的系统调用号一致（关键词 long 表示 entry 将占据与 long 类型的数据值相同的字节数）。

2. 构建新的内核

在增加新的系统调用到内核前，必须使自己熟悉从内核源代码构造二进制码的任务，并用新构造的内核启动机器。该活动包括如下任务，其中一些任务取决于 Linux 操作系统

特定的安装。

- 获取 Linux 分发版的内核代码。如果已事先在机器上安装了代码包，`/usr/src/linux` 或 `/usr/src/linux-2.x`（此后缀相当于内核版本号）目录下的文件可以使用；如果没有事先安装此代码，可从 Linux 发行版提供商或 <http://www.kernel.org> 处下载。

- 学习如何配置、编译和安装 Linux 二进制文件。这在不同的 Linux 发行版间有所不同，但构建内核（进入保存内核代码的目录后）的一些典型的命令包括：

- `make xconfig`。
- `make dep`。
- `make bzImage`。

- 增加新的由系统支持的可启动的内核集的 `entry`。Linux 操作系统通常使用 `lilo` 或 `grub` 等工具来维护可启动内核列表，用户在机器启动期间能够从中加以选择。如果你使用的系统支持 `lilo`，可向 `lilo.conf` 增加一个 `entry`：

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

其中 `/boot/bzImage.mykernel` 是内核 `image`，`mykernel` 是新的内核相关的标签。完成这个步骤后，你可以选择启动新的内核，或在新构建的内核不能正常运作时启动没有修改过的内核。

3. 扩展内核源

现在可以试着增加新的文件到用来编译内核的源文件集中。通常，源代码保存在 `/usr/src/linux-2.x/kernel` 目录下，当然其位置可能在 Linux 发行版中有些不同。增加系统调用有两种方法。第一种选择是增加系统调用到一个该目录下已经存在的源文件中；第二种选择是在源文件目录下生成一个新的文件，并修改 `/usr/src/linux-2.x/kernel/Makefile` 以在编译过程中包括新生成的文件。第一种方法的优点在于通过修改已作为编译过程的一部分并已存在的文件，不再需要修改 `Makefile`。

4. 向内核增加新的系统调用

现在你已经熟悉与构建和启动 Linux 内核相关的各种背景任务，那么可以开始向 Linux 内核增加新的系统调用了。在这个项目中，系统调用具有有限的功能，它将简单地从用户模式转为内核模式，打印用内核消息记录的一则消息，并转为用户模式，在此称之为 `helloworld` 系统调用。尽管只有有限的功能，它还是说明了系统调用机制，并清楚地显示了用户程序和内核之间的交互。

- 生成新的名为 `helloworld.c` 的文件来定义系统调用，包括头文件 `linux/linkage.h` 和 `linux/kernel.h`，将下述代码增加到该文件中：

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
    printk(KERN_EMERG "hello world!");

    return 1;
}
```

这会生成名为 `sys_helloworld()` 的系统调用。如果将此系统调用增加到源代码目录下已存在的文件中，所需做的就是将 `sys_helloworld()` 函数增加到所选择的文件中。代码中的 `asmlinkage` 是从同时用 C 语言和 C++ 语言编写 Linux 的时代遗留下来，指示代码是用 C 语言写的。`printk()` 函数被用来打印给内核日志文件的消息，因此仅能从内核调用。在 `printk()` 的参数中指定的内核消息被记录到文件 `/var/log/kernel/warnings` 中，`printk()` 调用的函数原型在 `/usr/include/linux/kernel.h` 中定义。

- 在 `/usr/src/linux-2.x/include/asm-i386/unistd.h` 中为 `__NR_helloworld` 定义一个新的系统调用号。用户程序可以用此号来识别新增加的系统调用。同样还要保证增加 `__NR_syscalls` 的值，它被保存在相同的文件中，该常数跟踪在内核中定义的系统调用号。

- 增加一个条目 `.long sys_helloworld` 到 `/usr/src/linux-2.x/arch/i386/kernel/entry.S` 文件中的 `sys_call_table` 中，如前所述，系统调用号被用作该表的索引，以查找被调用的系统调用的句柄编码的位置。

- 将 `helloworld.c` 文件增加到 `Makefile`（如果为系统调用生成新的文件）。保存一个旧的内核二进制码镜像的备份（以防新生成的内核出现问题）。现在你可以构建新的内核了，将它重新命名以区别未修改的内核，并向装入程序配置文件增加一个 `entry`（如 `lilo.conf`）。完成这些步骤后，就既可以启动旧的内核，也可以启动包括你的系统调用的新内核了。

5. 从一个用户程序使用系统调用

当你用新的内核启动时，它将支持新定义的系统调用。你现在只需要从用户程序调用这个系统调用。通常，标准的 C 语言库支持为 Linux 操作系统定义的系统调用接口。当新的系统调用没有与 C 语言库连接时，调用你的系统调用将会需要人工干预。

正如前面提及的，通过保存适当的值到硬件寄存器并完成一个陷阱指令来调用系统调用。不幸的是，这些都是低级操作，不能用 C 语句来完成，而需要用到汇编语言。幸运的是，Linux 提供了宏指令。例如，如下 C 程序使用 `_syscall0()` 宏来调用新定义的系统调用：

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);
```

```
main()
{
    helloworld();
}
```

- `_syscall0()` 采用了两个参数。第一个参数定义系统调用返回值的类型，第二个参数是系统调用的名称。该名称被用来标识在执行陷阱指令前保存在硬件寄存器中的系统调用号。如果你的系统调用需要参数，可以采用一个不同的宏（如 `_syscall0()`，后缀表明参数数量）来代替完成系统调用所需的汇编代码。

- 用新构建的内核编译并执行程序。此时在内核日志文件 `/var/log/kernel/warnings` 中应有一个消息 “hello world”，以表明系统调用被执行。

下一步，需要考虑扩展系统调用的功能。如何将整数值或字符串传递给系统调用，并在其内核日志文件中打印出来？不是简单地使用硬件寄存器从用户程序传递一个整数值给内核，而是传递指向用户程序地址空间的指针。这意味着什么？

文献注记

Dijkstra[1968]提倡操作系统的层次设计。Brinch-Hansen[1970]是一个操作系统应作为一个内核，在其上能建立完整的系统的观点的早期支持者。

Tamches 和 Mille[1999]介绍了系统指令和动态跟踪。Cantrill 等[2004]中讨论了 Dtrace。Cheung 和 Loong[1995]从微内核到扩展的系统探讨了操作系统指令问题。

MS-DOS 3.1 版，见 Microsoft[1986]。Windows NT 和 Windows 2000 在 Solomon[1998]、Solomon 和 Russinovich[2000]中有所介绍。BSD UNIX 见 McKusick 等[1996]。Bovet 和 Cesati[2002]包括了详细的 Linux 内核的内容。一些 UNIX 系统（包括 Mach）详见 Vahalia[1996]。有关 Mac OS X 的信息可在 <http://www.apple.com/macosx> 中找到。Massalin 和 Pu[1989]论述了试验性的综合操作系统。Mauro 和 McDougall[2001]全面介绍了 Solaris。

第一个提供了一个虚拟机的操作系统是在 IBM 360/67 上的 CP/67。IBM VM/370 操作系统是源自 CP/67 的商业版本。关于基于微内核的操作系统 Mach 的详细信息，可在 Young 等[1987]中找到。Kaashoek 等[1997]提供了关于 exkernel 操作系统的详细信息，该系统的结构从保护中分离管理问题，从而给不信任的软件行使对硬件和软件资源控制的能力。

Gosling 等[1996]和 Lindholm 和 Yellin[1999]分别描述了 Java 语言和 Java 虚拟机的特点。Venners[1998]全面介绍了 Java 虚拟机的内部工作机制。Golm 等[2002]强调了 JX 操作系统，Back 等[2000]论述了 Java 操作系统的设计。更多的有关 Java 的信息见 <http://java.sun.com>。关于 VMware 的实现细节可在 Sugerman 等[2001]中找到相关信息。

第二部分 进 程 管 理

进程可看做是正在执行的程序。进程需要一定的资源(如 CPU 时间、内存、文件和 I/O 设备)来完成其任务。这些资源在创建进程或执行进程时被分配。

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

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

操作系统负责进程和线程管理，包括用户进程与系统进程的创建与删除，进程调度，提供进程同步机制、进程通信机制与进程死锁处理机制。

第3章 进 程

早期的计算机系统只允许一次执行一个程序。这种程序对系统有完全的控制，能访问所有的系统资源。现代计算机系统允许将多个程序调入内存并发执行。这一要求对各种程序提供更严格的控制和更好的划分。这些需求产生了进程的概念，即执行中的程序。进程是现代分时系统的工作单元。

操作系统越复杂，就越能为用户做更多的事。虽然操作系统的主要目标是执行用户程序，但是也需要顾及内核之外的各种系统任务。因此，系统由一组进程组成：操作系统进程执行系统代码而用户进程执行用户代码。通过 CPU 多路复用，所有这些进程可以并发执行。通过进程之间的切换，操作系统能使计算机更为高效。

本章目标

- 介绍进程的概念——执行中的程序，形成所有计算的基础。
- 介绍进程的不同特点，包括调度、创建和删除，以及通信。
- 介绍客户机-服务器系统间的通信。

3.1 进 程 概 念

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

本书中时常同时使用*作业*与*进程*这两个概念。虽然笔者自己偏爱*进程*，但是许多操作系统的理论和技术是在操作系统的主要活动被称为*作业处理*期间发展起来的。如果因为进程取代了作业，而简单地避免使用有关*作业*的常用短语（如*作业调度*），则会令人误解。

3.1.1 进程

正如前述，进程是执行中的程序，这是一种非正式的说法。进程不只是程序代码，程序代码有时称为**文本段**（或**代码段**）。进程还包括当前活动，通过**程序计数器**的值和**处理器**

寄存器的内容来表示。另外，进程通常还包括进程**堆栈段**（包括临时数据，如函数参数、返回地址和局部变量）和**数据段**（包括全局变量）。进程还可能包括**堆（heap）**，是在进程运行期间动态分配的内存。内存中的进程结构如图 3.1 所示。

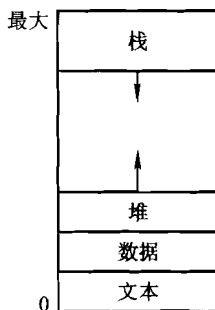


图 3.1 内存中的进程

这里强调：程序本身不是进程；程序只是**被动实体**，如存储在磁盘上包含一系列指令的文件内容（常被称为可执行文件），而进程是**活动实体**，它有一个程序计数器用来表示下一个要执行的命令和相关资源集合。当一个可执行文件被装入内存时，一个程序才能成为进程。装载可执行文件通常有两种方法，即双击一个代表此可执行文件的图标或在命令行中输入该文件的文件名（如 `prog.exe` 或 `a.out`）。

虽然两个进程可以是与同一程序相关，但是它们被当作两个独立的执行序列。例如，多个用户可运行不同的电子邮件副本，或者同一用户能调用多个 Web 浏览器程序的副本。这些都是独立的进程，虽然文本段相同，但是数据段、堆、堆栈段却不同。一个进程在执行时产生许多进程是很常见的。本书将在 3.4 节中讨论这些问题。

3.1.2 进程状态

进程在执行时会改变状态。进程状态在某种程度上是由当前活动所定义的。每个进程可能处于下列状态之一：

- **新的**：进程正在被创建。
- **运行**：指令正在被执行。
- **等待**：进程等待某个事件的发生（如 I/O 完成或收到信号）。
- **就绪**：进程等待分配处理器。
- **终止**：进程完成执行。

这些状态的名称较随意，且随操作系统不同而变化。不过，它们所表示的状态可以出现在所有系统上。有的系统更为详细地描述了进程状态。必须认识到一次只有一个进程可在一个处理器上运行，但是多个进程可处于**就绪**或**等待**状态。与这些状态相对的状态图见

图 3.2。

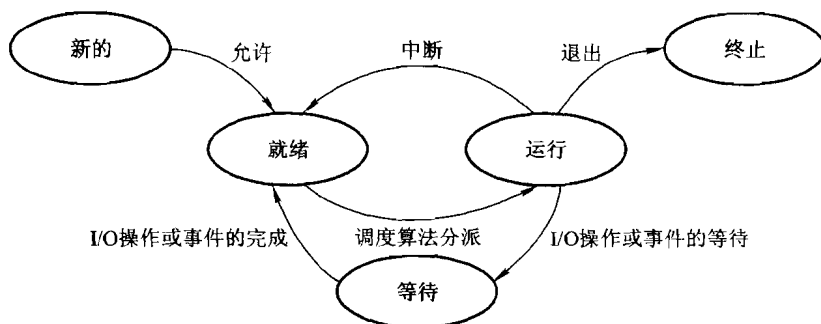


图 3.2 进程状态图

3.1.3 进程控制块

每个进程在操作系统内用**进程控制块**（process control block, PCB, 也称为任务控制块）来表示。图 3.3 给出了一个 PCB 的例子，它包含许多与一个特定进程相关的信息。

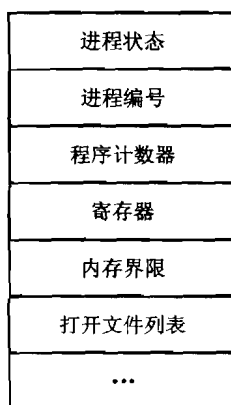


图 3.3 进程控制块（PCB）

- **进程状态**：状态可包括新的、就绪、运行、等待、停止等。
- **程序计数器**：计数器表示进程要执行的下个指令的地址。
- **CPU 寄存器**：根据计算机体系结构的不同，寄存器的数量和类型也不同。它们包括累加器、索引寄存器、堆栈指针、通用寄存器和其他条件码信息寄存器。与程序计数器一起，这些状态信息在出现中断时也需要保存，以便进程以后能正确地继续执行（见图 3.4）。
- **CPU 调度信息**：这类信息包括进程优先级、调度队列的指针和其他调度参数（第 5

章讨论进程调度)。

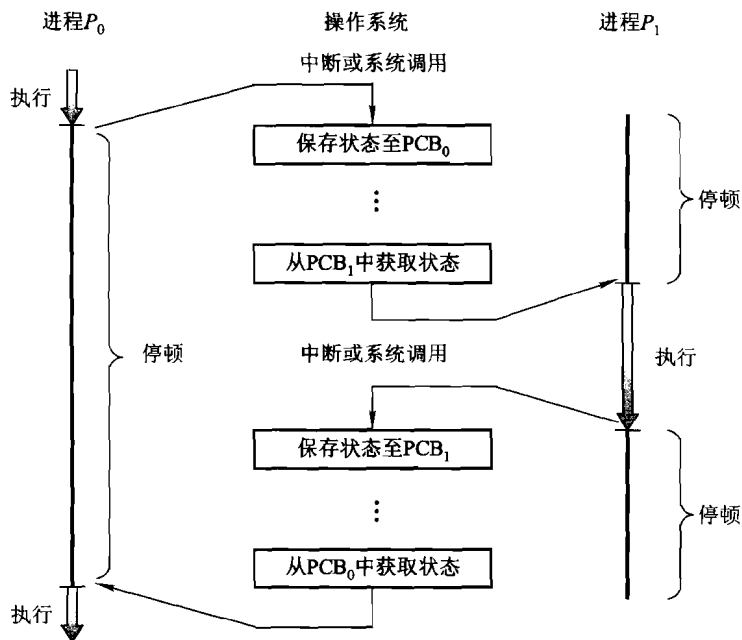


图 3.4 CPU 在进程间的切换

- **内存管理信息：**根据操作系统所使用的内存系统，这类信息包括基址和界限寄存器的值、页表或段表（见第 8 章）。

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

- **I/O 状态信息：**这类信息包括分配给进程的 I/O 设备列表、打开的文件列表等。

简而言之，PCB 简单地作为这些信息的仓库，这些信息在进程与进程之间是不同的。

3.1.4 线程

迄今为止所讨论的进程模型暗示：一个进程是一个只能进行单个执行线程的程序。例如，如果一个进程运行一个字处理器程序，那么只能执行单个线程指令。这种单一控制线程使得进程一次只能执行一个任务。例如，用户不能在同一进程内，同时输入字符和进行拼写检查。许多现代操作系统扩展了进程概念以支持一次能执行多个线程。第 4 章将讨论多线程进程。

3.2 进程调度

多程序设计的目的是一无论何时都有进程在运行，从而使 CPU 利用率达到最大化。分时系统的目的是在进程之间快速切换 CPU 以便用户在程序运行时能与其进行交互。为了达到此目的，**进程调度**选择一个可用的进程（可能从多个可用进程集合中选择）到 CPU 上执行。单处理器系统从不会有超过一个进程在运行。如果有多个进程，那么余下的则需要等待 CPU 空闲并重新调度。

3.2.1 调度队列

进程进入系统时，会被加到作业队列中，该队列包括系统中的所有进程。驻留在内存中就绪的、等待运行的进程保存在**就绪队列**中。该队列通常用链表来实现，其头节点指向链表的第一个和最后一个 PCB 块的指针。每个 PCB 包括一个指向就绪队列的下一个 PCB 的指针域。

Linux 中的进程表示

Linux 操作系统中的进程控制块是通过 C 结构 `task_struct` 来表示的。这个结构包含了表示一个进程所需要的所有信息，包括进程的状态、调度和内存管理信息、打开文件列表和指向父进程和所有子进程的指针（创建进程的进程是父进程，被进程创建的进程为子进程）。`task_struct` 的这些字段包括：

```
pid_t pid;                /* process identifier */
long state;               /* state of the process */
unsigned int time_slice   /* scheduling information */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;     /* address space of this process */
```

例如，进程的状态是通过这个结构中的 `long state` 字段来表示的。在 Linux 内核里，所有活动的进程是通过一个名为 `task_struct` 的双向链表来表示的，内核为当前正在运行的进程保存了一个指针(`current`)，如图 3.5 所示。

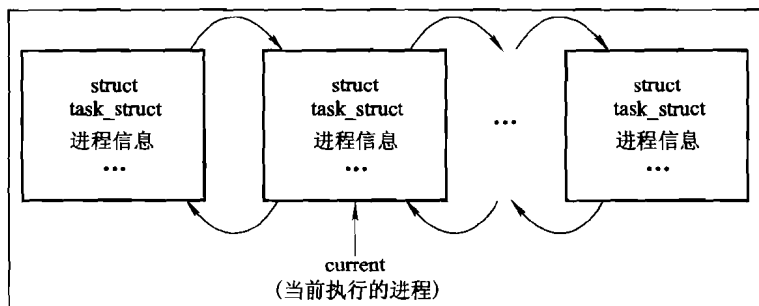


图 3.5 Linux 系统中的活动进程

解释一下内核如何操作一个指定进程的 `task_struct` 字段。假定操作系统想把当前运行进程的状态值修改成 `new_state`。如果 `current` 是指向当前进程的指针，那么要改变状态可以如下进行：

```
current->state = new_state;
```

操作系统也有其他队列。当给进程分配了 CPU 后，它开始执行并最终完成，或被中断，或等待特定事件发生（如完成 I/O 请求）。假设进程向一个共享设备（如磁盘）发送 I/O 请求，由于系统有许多进程，磁盘可能会忙于其他进程的 I/O 请求，因此该进程可能需要等待磁盘。等待特定 I/O 设备的进程列表称为设备队列。每个设备都有自己的设备队列（见图 3.6）。

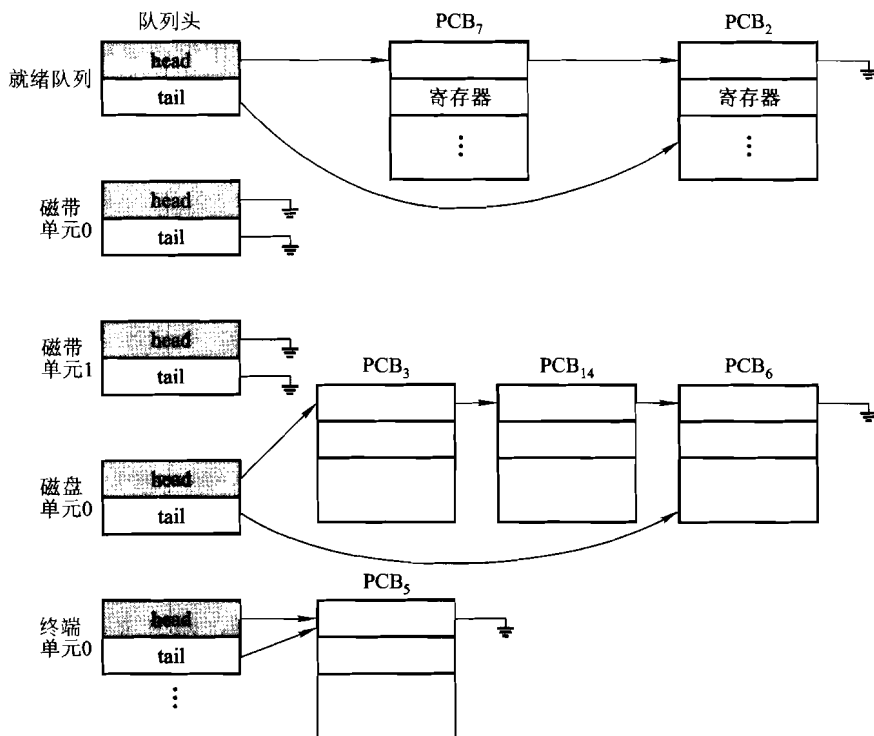


图 3.6 就绪队列和各种设备队列

讨论进程调度的常用表示方法是队列图，如图 3.7 所示。每个长方形表示一个队列。有两种队列：就绪队列和一组设备队列。圆形表示为队列服务的资源，箭头表示系统内进程的流向。

新进程开始处于就绪队列。它在就绪队列中等待直到被选中执行或被派遣。当进程分配到 CPU 并执行时，可能发生下面几种事件中的一种：

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

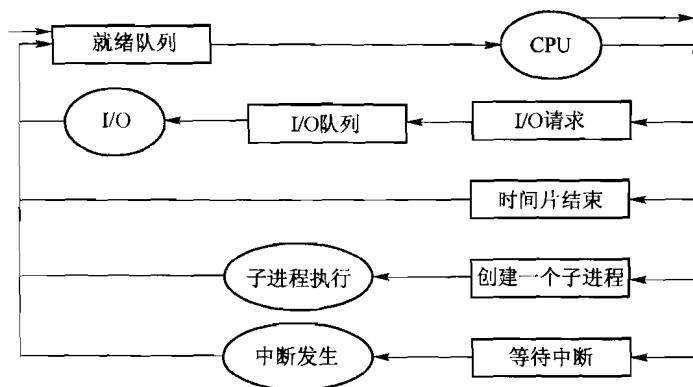


图 3.7 表示进程调度的队列图

对于前两种情况，进程最终从等待状态切换到就绪态，并放回到就绪队列中。进程继续这一循环直到终止，到时它将从所有队列中删除，其 PCB 和资源将得以释放。

3.2.2 调度程序

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

通常对于批处理系统，进程更多地是被提交，而不是马上执行。这些进程被放到大容量存储设备（通常为磁盘）的缓冲池中，保存在那里以便以后执行。长期调度程序（long-term scheduler）或作业调度程序（job scheduler）从该池中选择进程，并装入内存以准备执行。短期调度程序（short-term scheduler）或 CPU 调度程序从准备执行的进程中选择进程，并为之分配 CPU。

这两个调度程序的主要差别是它们执行的频率。短期调度程序必须频繁地为 CPU 选择新进程。进程可能执行数毫秒（ms）就会进行 I/O 请求，短期调度程序通常每 100 ms 至少执行一次。由于每次执行之间的时间较短，短期调度程序必须要快。如果需要 10 ms 来确定执行一个运行 100 ms 的进程，那么 $10/(100+10) \approx 9\%$ 的 CPU 时间会用于（或浪费在）调度工作上。

长期调度程序执行得并不频繁，在系统内新进程的创建之间可能有数分钟间隔。长期调度程序控制多道程序设计的程度（内存中的进程数量）。如果多道程序的程度稳定，那么创建进程的平均速度必须等于进程离开系统的平均速度。因此，只有当进程离开系统后，

才可能需要调度长期调度程序。由于每次执行之间时间间隔得较长，长期调度程序能使用更多时间来选择执行进程。

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

对于有些系统，可能没有或很少有长期调度程序。例如，UNIX 或微软 Windows 的分时系统通常没有长期调度程序，只是简单地将所有新进程放在内存中以供短期调度程序使用。这些系统的稳定性依赖于物理限制（如可用的终端数）或用户的自我调整。如果多用户系统性能下降到令人难以接受，那么将有用户退出。

有的操作系统如分时系统，可能引入另外的中期调度程序（medium-term scheduler），如图 3.8 所示。中期调度程序的核心思想是能将进程从内存（或从 CPU 竞争）中移出，从而降低多道程序设计的程度。之后，进程能被重新调入内存，并从中断处继续执行。这种方案称为交换（swapping）。通过中期调度程序，进程可换出，并在后来可被换入。为了改善进程组合，或者因内存要求的改变引起了可用内存的过度使用而需要释放内存，就有必要使用交换。交换将在第 8 章讨论。

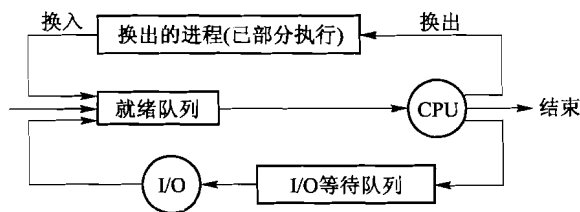


图 3.8 增加了中期调度的队列图

3.2.3 上下文切换

正如 1.2.1 小节所述，中断使 CPU 从当前任务改变为运行内核子程序，这样的操作在通用系统中发生得很频繁。当发生一个中断时，系统需要保存当前运行在 CPU 中进程的上下文，从而在其处理完后能恢复上下文，即先中断进程，之后再继续。进程上下文用进程的 PCB 表示，它包括 CPU 寄存器的值、进程状态（见图 3.2）和内存管理信息等。通常，

通过执行一个**状态保存** (state save) 来保存 CPU 当前状态 (不管它是内核模式还是用户模式), 之后执行一个**状态恢复** (state restore) 重新开始运行。

将 CPU 切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态, 这一任务称为**上下文切换** (context switch)。当发生上下文切换时, 内核会将旧进程的状态保存在其 PCB 中, 然后装入经调度要执行的并已保存的新进程的上下文。上下文切换时间是额外开销, 因为切换时系统并不能做什么有用的工作。上下文切换速度因机器而不同, 它依赖于内存速度、必须复制的寄存器的数量、是否有特殊指令 (如装入或保存所有寄存器的单个指令), 一般需几毫秒。

上下文切换时间与硬件支持密切相关。例如, 有的处理器 (如 Sun UltraSPARC) 提供了多组寄存器集合, 上下文切换只需要简单地改变当前寄存器组的指针。当然, 如果活动进程数超过了寄存器集合数量, 那么系统需要像以前一样在寄存器与内存之间进行数据复制。而且, 操作系统越复杂, 上下文切换所要做的工作就越多。如第 8 章将要谈到的, 高级内存管理技术在各个上下文切换中要求切换更多的数据。例如, 在使用下一个任务的空间之前, 当前进程的地址空间需要保存。进程空间如何保存和保存它需要做多少工作, 取决于操作系统的内存管理方法。

3.3 进程操作

绝大多数系统内的进程能并发执行, 它们可以动态创建和删除, 因此操作系统必须提供某种机制 (或工具) 以创建和终止进程。本节探讨进程创建和删除的机制, 并举例说明 UNIX 系统和 Windows 系统的进程创建。

3.3.1 进程创建

进程在其执行过程中, 能通过创建进程系统调用 (create-process system call) 创建多个新进程。创建进程称为父进程, 而新进程称为子进程。每个新进程可以再创建其他进程, 从而形成了进程树。

大多数操作系统 (包括 UNIX 和 Windows 系列操作系统) 根据一个唯一的进程标识符 (process identifier, pid) 来识别进程, pid 通常是一个整数值。图 3.9 所示是一个典型的 Solaris 系统中的进程树, 显示了每个进程的名字和 pid。在 Solaris 系统中, 树顶端的进程是标识符为 0 的 Sched 进程。Sched 进程生成几个子进程——包括 pageout 和 fsflush。这些进程负责管理内存和文件系统, Sched 进程还生成 init 进程, 它作为所有用户进程的根进程。在图 3.9 中, 有两个 init 的子进程——inetd 和 dtlogin。inetd 负责网络服务, 如 telnet 和 ftp; 而 dtlogin 表示用户登录界面。当一个用户登录时, dtlogin 生成一个 X-windows 会话, 它反过来又生成 sdt_shel 进程。在 sdt_shel 进程之下, 生成一个用户命令行 Shell——C Shell

或 Csh。这是一个用户调用不同子进程的命令行接口，如 ls 或 cat 命令。同样还可以看到进程标识符为 7778 的 Csh 进程，它表示一个登录到系统使用 telnet 的用户。该用户启动了 Netscape 浏览器（进程标识符为 7785）和 emacs 编辑器（进程标识符为 8105）。

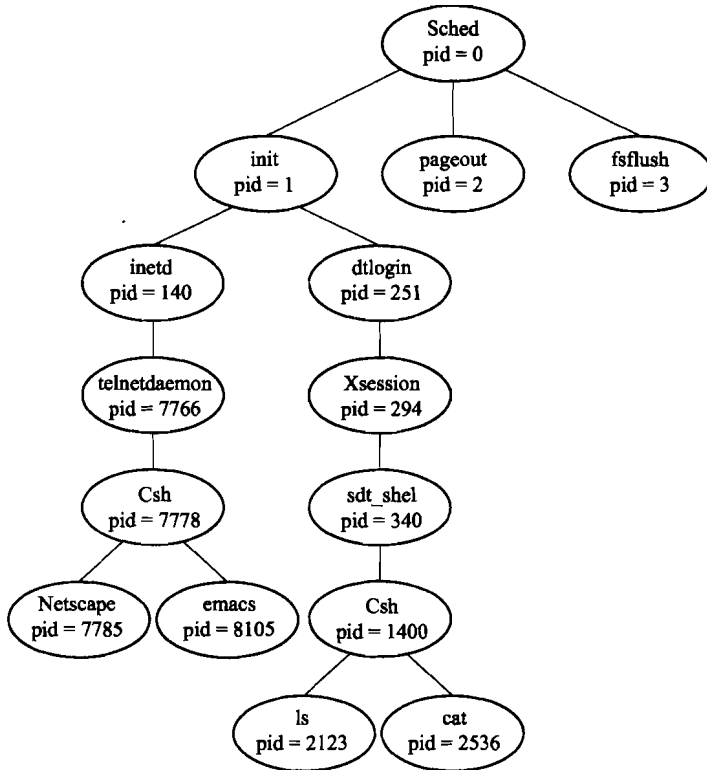


图 3.9 一个典型的 Solaris 系统中的进程树

在 UNIX 系统中，使用 ps 命令可以得到一个进程列表。例如，输入命令 `ps -el` 将会列出系统所有当前活动进程的完整信息。通过递归跟踪父进程至 init 进程，可以很方便地构造类似于图 3.9 的进程树。

通常，进程需要一定的资源（如 CPU 时间、内存、文件、I/O 设备）来完成其任务。在一个进程创建子进程时，子进程可能从操作系统那里直接获得资源，也可能只从其父进程那里获得资源。父进程可能必须在其子进程之间分配资源或共享资源（如内存或文件）。限制子进程只能使用父进程的资源能防止创建过多的进程带来的系统超载。

在进程创建时，除了得到各种物理和逻辑资源外，初始化数据（或输入）由父进程传递给子进程。例如，考虑一个进程，其功能是在终端屏幕上显示文件（如 img.jpg）的状态。在创建时，作为它的父进程的输入，它会得到文件 img.jpg 的名称，并能使用此名称打开

文件，以及写出内容。它也能得到输出设备的名称。有的操作系统将资源传递给子进程。在这类系统上，新进程可得到两个打开文件，即 `img.jpg` 和终端设备，新进程只需在两者之间传递数据。

当进程创建新进程时，有两种执行可能：

- ① 父进程与子进程并发执行。
- ② 父进程等待，直到某个或全部子进程执行完。

新进程的地址空间也有两种可能：

- ① 子进程是父进程的复制品（具有与父进程相同的程序和数据）。
- ② 子进程装入另一个新程序。

为了说明这些不同实现，现在来看一下 UNIX 操作系统。在 UNIX 中，每个进程都用一个唯一的整数形式的进程标识符来标识。通过 `fork()` 系统调用，可创建新进程。新进程通过复制原来进程的地址空间而成。这种机制允许父进程与子进程方便地进行通信。两个进程（父进程和子进程）都继续执行位于系统调用 `fork()` 之后的指令。但是，有一点不同：对于新（子）进程，系统调用 `fork()` 的返回值为 0；而对于父进程，返回值为子进程的进程标识符（非零）。

通常，在系统调用 `fork()` 之后，一个进程会使用系统调用 `exec()`，以用新程序来取代进程的内存空间。系统调用 `exec()` 将二进制文件装入内存（消除了原来包含系统调用 `exec()` 的程序的内存映射），并开始执行。采用这种方式，两个进程能相互通信，并能按各自的方法执行。父进程能创建更多的子进程，或者如果在子进程运行时没有什么可做，那么它采用系统调用 `wait()` 把自己移出就绪队列来等待子进程的终止。

如图 3.10 所示的 C 程序说明了上述 UNIX 系统调用。现在有两个不同的进程运行同一程序。子进程的 `pid` 值为 0，而父进程的 `pid` 值大于 0。子进程通过系统调用 `execlp()` (`execlp()` 是系统调用 `exec()` 的一种版本)，用 UNIX 命令 `/bin/ls`（用来列出目录清单）来覆盖其地址空间。父进程通过系统调用 `wait()` 来等待子进程的完成。当子进程完成时（通过显示或隐式调用 `exit()`），父进程会从 `wait()` 调用处开始继续，并调用系统调用 `exit()` 以表示结束。这可用图 3.11 表示。

再如，考虑在 Windows 中的进程生成。Win32 API 通过采用 `CreateProcess()` 函数（它与 `fork()` 中的父进程生成子进程类似）创建进程。然而，`fork()` 中子进程继承了父进程的地址空间，而 `CreateProcess()` 生成函数时，需要将一个特殊程序装入子进程的地址空间。进一步讲，与 `fork()` 不需要传递参数不同，`CreateProcess()` 至少需要传递 10 个参数。

图 3.12 所示的 C 程序是一个 `CreateProcess()` 函数，它生成一个装载 `mspaint.exe` 应用程序的子进程。选择 10 个参数中的默认值传递给 `CreateProcess()` 函数。需要了解 Win32 API 中进程生成和管理细节的读者可以查阅本章后面的推荐读物。

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

void main (int argc, char *argv[])
{
    pid_t pid;
    /* fork a child process */
    pid = fork();

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

图 3.10 创建另外一个进程的 C 程序

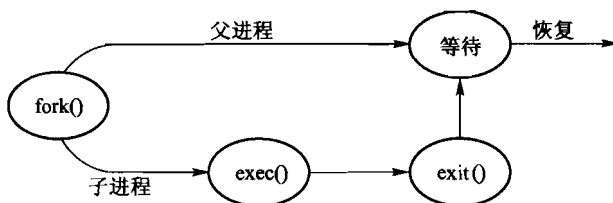


图 3.11 进程生成

传递给 `CreateProcess()` 的两个参数是 `STARTUPINFO` 和 `PROCESS_INFORMATION` 结构的实例。`STARTUPINFO` 指明新进程的许多特性，如窗口大小、标准输入及输出文件的句柄。`PROCESS_INFORMATION` 结构包含一个句柄以及新的生成进程和线程的标识。在调用 `CreateProcess()` 之前，调用 `ZeroMemory()` 函数来为其中每个结构清空内存。

```

#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    //allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    //create child process
    if (!CreateProcess(NULL, //use command line
        "C:\\WINDOWS\\system32\\mspaint.exe", //command line
        NULL, //don't inherit process handle
        NULL, //don't inherit thread handle
        FALSE, //disable handle inheritance
        0, //no creation flags
        NULL, //use parent's environment block
        NULL, //use parent's existing directory
        &si,
        &pi))
    {
        fprintf(stderr, "create Process Failed");
        return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    Printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

```

图 3.12 使用 Win32 API 生成一个单独的进程

首先传递给 `CreateProcess()` 函数的两个参数是应用名和命令行参数。如果应用名为 `NULL`（此时它就是 `NULL`），命令行参数指明了要装入的应用。在这个例子中，装入的是微软 Windows 中的 `mspaint.exe` 应用程序。除这两个初始参数之外，使用系统默认参数来继承进程和线程句柄和指定不创建标志，还使用父进程的已有环境块和启动目录。最后，

提供了两个指向程序刚开始生成的 STARTUPINFO 和 PROCESS_INFORMATION 结构的指针。在图 3.10 中, 父进程通过调用 wait() 系统调用等待子进程结束, 在 Win32 中有相同功能的是 WaitForSingleObject(), 它的参数指定一个子进程的句柄 (pi.hProcess) 即等待进程结束。一旦子进程结束, 控制将从 WaitForSingleObject() 函数返回到父进程。

3.3.2 进程终止

当进程完成执行最后的语句并使用系统调用 exit() 请求操作系统删除自身时, 进程终止。这时, 进程可以返回状态值 (通常为整数) 到父进程 (通过系统调用 wait())。所有进程资源 (包括物理和虚拟内存、打开文件和 I/O 缓冲) 会被操作系统释放。

在其他情况下也会出现终止。进程通过适当的系统调用 (如 Win32 中的 TerminateProcess()) 能终止另一个进程。通常, 只有被终止进程的父进程才能执行这一系统调用。否则, 用户可以任意地终止彼此的作业。记住, 父进程需要知道其子进程的标识符。因此, 当一个进程创建新进程时, 新创建进程的标识符要传递给父进程。

父进程终止其子进程的原因有很多, 如:

- 子进程使用了超过它所分配到的一些资源。(为判定是否发生这种情况, 要求父进程有一个检查其子进程状态的机制。)
- 分配给子进程的任务已不再需要。
- 父进程退出, 如果父进程终止, 那么操作系统不允许子进程继续。

有些系统, 包括 VMS, 不允许子进程在父进程已终止的情况下存在。对于这类系统, 如果一个进程终止 (正常或不正常), 那么它的所有子进程也将终止。这种现象, 称为级联终止 (cascading termination), 通常由操作系统进行。

为了说明进程执行和终止, 可考虑一下 UNIX: 可以通过系统调用 exit() 来终止进程, 父进程可以通过系统调用 wait() 以等待子进程的终止。系统调用 wait() 返回了终止子进程的进程标识符, 以使父进程能够知道哪个子进程终止了。如果父进程终止, 那么其所有子进程会以 init 进程作为父进程。因此, 子进程仍然有一个父进程来收集状态和执行统计。

3.4 进程间通信

操作系统内并发执行的进程可以是独立进程或协作进程。如果一个进程不能影响其他进程或被其他进程所影响, 那么该进程是独立的。显然, 不与任何其他进程共享数据的进程是独立的。另一方面, 如果系统中一个进程能影响其他进程或被其他进程所影响, 那么该进程是协作的。显然, 与其他进程共享数据的进程为协作进程。

可能需要提供环境以允许进程协作, 这有许多理由:

- 信息共享 (information sharing): 由于多个用户可能对同样的信息感兴趣 (例如共享

的文件)，所以必须提供环境以允许对这些信息进行并发访问。

- **提高运算速度 (computation speedup)**：如果希望一个特定任务快速运行，那么必须将它分成子任务，每个子任务可以与其他子任务并行执行。注意，如果要想实现这样的加速，需要计算机有多个处理单元（例如 CPU 或 I/O 通道）。

- **模块化 (modularity)**：可能需要按模块化方式构造系统，如第 2 章所讨论，可将系统功能分成独立进程或线程。

- **方便 (convenience)**：单个用户也可能同时执行许多任务。例如，一个用户可以并行进行编辑、打印和编译操作。

协作进程需要一种进程间通信机制 (interprocess communication, IPC) 来允许进程相互交换数据与信息。进程间通信有两种基本模式：(1) 共享内存，(2) 消息传递。在共享内存模式中，建立起一块供协作进程共享的内存区域，进程通过向此共享区域读或写入数据来交换信息。在消息传递模式中，通过在协作进程间交换消息来实现通信。图 3.13 给出了这两种模式的对比。

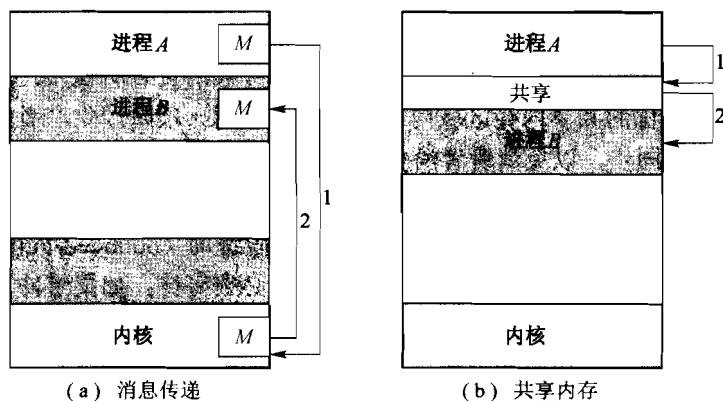


图 3.13 通信模型

在操作系统中，上述两种模式都很常用，而且许多系统也实现了这两种模式。消息传递对于交换较少数量的数据很有用，因为不需要避免冲突。对于计算机间的通信，消息传递也比共享内存更易于实现。共享内存允许以最快的速度进行方便的通信，在计算机中它可以达到内存的速度。共享内存比消息传递快，消息传递系统通常用系统调用来实现，因此需要更多的内核介入的时间消耗。与此相反，在共享内存系统中，仅在建立共享内存区域时需要系统调用，一旦建立了共享内存，所有的访问都被处理为常规的内存访问，不需要来自内核的帮助。本节后面的部分将更为详细地讨论每种 IPC 模式。

3.4.1 共享内存系统

采用共享内存的进程间通信需要通信进程建立共享内存区域。通常，一块共享内存区域驻留在生成共享内存段进程的地址空间。其他希望使用这个共享内存段进行通信的进程必须将此放到它们自己的地址空间上。回忆一下，通常操作系统试图阻止一个进程访问另一进程的内存。共享内存需要两个或更多的进程取消这个限制，它们通过在共享区域内读或写来交换信息。数据的形式或位置取决于这些进程而不是受控于操作系统。进程还负责保证它们不向同一区域同时写数据。

为了说明协作进程这一概念，可研究一下生产者-消费者问题，这是协作进程的通用范例。**生产者进程**产生信息以供**消费者进程**消费。例如，编译器产生的汇编代码供汇编程序使用，而汇编程序反过来产生目标代码供链接和装入程序使用。生产者-消费者问题同时还为客户机-服务器范例提供了有用的隐喻。通常将客户机当作一个生产者，而将服务器当作一个消费者。例如，一个 Web 服务器生产（提供）HTML 文件和图像，它们被请求资源的客户 Web 浏览器所消费（读取）。

采用共享内存是解决生产者-消费者问题方法中的一种。为了允许生产者进程和消费进程能并发执行，必须要有一个缓冲来被生产者填充并被消费者所使用。此缓冲驻留在生产者进程和消费者进程的共享内存区域内，当消费者使用一项时，生产者能产生另一项。生产者和消费者必须同步，以免消费者消费一个没有生产出来的项。

可以使用两种缓冲。**无限缓冲**（unbounded-buffer）对缓冲大小没有限制。消费者可能不得不在等待新的项，但生产者总是可以产生新项。**有限缓冲**（bounded-buffer）假设缓冲大小固定。对于这种情况，如果缓冲为空，那么消费者必须等待；如果缓冲为满，那么生产者必须等待。

更进一步了解进程共享内存如何使用有限缓冲。下面驻留在内存中的变量由生产者和消费者共享：

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

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

`BUFFER_SIZE == out` 时，缓冲为满。

生产者进程和消费者进程代码分别如图 3.14 和图 3.15 所示。生产者进程有一个局部变量 `nextProduced` 以存储所产生的新项。消费者进程有一个局部变量 `nextConsumed` 以存储所要使用的新项。

```

item nextProduced

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

```

图 3.14 生产者进程

```

item nextConsumed

while (true) {
    while ( in == out )
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    /* consume the item in nextConsumed */
}

```

图 3.15 消费者进程

这种方法最多允许缓冲的最大项数为 `BUFFER_SIZE-1`，允许最大项数为 `BUFFER_SIZE` 的问题留作练习。在 3.5.1 节，将讨论 POSIX API 中的共享内存。

刚才的例子没有解决生产者和消费者同时访问共享内存的问题。在第 6 章，将讨论在共享内存环境下协作进程如何有效实现同步。

3.4.2 消息传递系统

3.4.1 小节讨论协作进程如何能通过共享内存来通信。这种方法要求进程共享一个内存区域，并且需要应用程序员自己明确编写访问和操作共享内存的代码。实现同样效果的另一种方法是由操作系统提供机制，让协作进程能通过消息传递工具来进行通信。

消息传递提供一种机制以允许进程不必通过共享地址空间来实现通信和同步，这在分布式环境中（通信进程可能位于由网络连接起来的不同计算机上）特别有用。例如，用于 WWW 的 chat 程序就是通过消息交换来实现通信。

消息传递工具提供至少两种操作：发送（消息）和接收（消息）。由进程发送的消息可以是定长的或变长的。如果只能发送定长消息，那么系统级的实现十分简单。不过，这一限制却使得编程任务更加困难。相反地，变长消息要求更复杂的系统级实现，但是编程任务变得简单。这是贯穿整个操作系统设计的一种常见的折中问题。

如果进程 *P* 和 *Q* 需要通信，那么它们必须彼此相互发送消息和接收消息，它们之间必须要有通信线路（communication link）。该线路有多种实现方法。这里不关心线路的物理实现（如共享内存、硬件总线或网络，参见第 16 章），而只关心逻辑实现。如下是一些逻

辑实现线路和 send()/receive()操作的方法:

- 直接或间接通信。
- 同步或异步通信。
- 自动或显式缓冲。

下面研究这些相关问题。

1. 命名

需要通信的进程必须有一个方法以互相引用。它们可使用直接或间接通信。

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

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

这种方案的通信线路具有如下属性:

- 在需要通信的每对进程之间自动建立线路。进程仅需知道相互通信的标识符。
- 一个线路只与两个进程相关。
- 每对进程之间只有一个线路。

这种方案展示了对称寻址，即发送和接收进程必须命名对方以便通信。这种方案一个变形采用非对称寻址，即只要发送者命名接收者，而接收者不需要命名发送者。采用这种方案，原语 send()和 receive()定义如下:

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

对称和非对称寻址方案的缺点是限制了进程定义的模块化。改变进程的名称可能必须检查所有其他进程定义。所有旧名称的引用都必须找到，以便修改成为新名称。与下面介绍的间接调用方法相比，通常，这种标识符必需明确指出的**硬编码**技术用得更少。

在**间接通信**中，通过邮箱或端口来发送和接收消息。邮箱可以抽象成一个对象，进程可以向其中存放消息，也可从中删除消息，每个邮箱都有一个唯一的标识符。例如，POSIX 消息队列采用一个整数值来标识一个邮箱。对于这种方案，一个进程可能通过许多不同的邮箱与其他进程通信，但两个进程仅在其共享至少一个邮箱时可相互通信。原语 send()和 receive()定义如下:

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

对于这种方案，通信线路具有如下属性:

- 只有在两个进程共享一个邮箱时，才能建立通信线路。
- 一个线路可以与两个或更多的进程相关联。
- 两个通信进程之间可有多多个不同的线路，每个线路对应于一个邮箱。

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

- 允许一个线路最多只能与两个进程相关联。
- 一次最多允许一个进程执行 `receive()` 操作。
- 允许系统随意选择一个进程以接收消息(即进程 P_2 和 P_3 都可以接收消息，但两者不能同时接收消息)。系统同样可以定义一个算法来选择哪个进程是接收者（即 `round robin`，进程轮流接收消息的地方）。系统可以给发送者标识接收者。

进程或操作系统可以拥有邮箱。如果邮箱为进程所有（即邮箱是进程地址空间的一部分），那么需要区分拥有者（只能通过邮箱接收消息）和使用者的（只能向邮箱发送消息）。由于每个邮箱都有唯一的标识符，所以谁能接收发到邮箱的消息是没有什么疑问的。当拥有邮箱的进程终止，那么邮箱将消失。任何进程后来向该邮箱发送消息，都会得知邮箱不再存在。

与此相反，由操作系统所拥有的邮箱是独立存在的，并不属于某个特定的进程。因此，操作系统必须提供机制以允许进程进行如下操作：

- 创建新邮箱。
- 通过邮箱发送和接收消息。
- 删除邮箱。

创建新邮箱的进程默认为邮箱的拥有者。开始时，拥有者是唯一能通过该邮箱接收消息的进程。不过，通过系统调用，拥有权和接收特权可能传递给其他进程。当然，该规定会导致每个邮箱有多个接收者。

2. 同步

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

- **阻塞 send**：发送进程阻塞，直到消息被接收进程或邮箱所接收。
- **非阻塞 send**：发送进程发送消息并再继续操作。
- **阻塞 receive**：接收者阻塞，直到有消息可用。
- **非阻塞 receive**：接收者收到一个有效消息或空消息。

`send()` 和 `receive()` 可以进行多种组合。当 `send()` 和 `receive()` 都阻塞时，则在发送者和接收者之间就有一个**集合点 (rendezvous)**。当使用阻塞 `send()` 和 `receive()` 时，如何解决生产者-消费者问题就不再重要了。生产者仅需调用阻塞 `send()` 调用并等待，直到消息被送到接收者或邮箱。同样地，当消费者调用 `receive()` 时，发生阻塞直到有一个消息可用。

注意，同步和异步的概念常常出现在操作系统的 I/O 算法中，读者将在本书中多次见到。

3. 缓冲

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

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

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

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

零容量情况称为没有缓冲的消息系统，其他情况称为自动缓冲。

3.5 IPC 系统的实例

本节讨论三种不同的 IPC 系统。首先了解共享内存的 POSIX API；然后讨论 Mach 操作系统中的消息传递；最后讨论 Windows XP，它采用了共享内存作为提供特定类型消息传递的机制。

3.5.1 实例：POSIX 共享内存

有几种 IPC 机制适用于 POSIX 系统，包括共享内存和消息传递。在此讨论的是共享内存的 POSIX API。

进程必须首先用系统调用 `shmget()` 创建共享内存段（`shmget()` 由 Shared Memory GET 派生而来），下面的例子说明了 `shmget()` 的使用：

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

第一个参数指的是共享内存段关键字（标识符）。如果将其赋予 `IPC_PRIVATE`，则生成一个新的共享内存段。第二个参数指的是共享内存段的大小（按字节数）。最后第三个参数标识模式，它明确了如何使用共享内存段——即用来读、用来写或二者都包括。通过把模式设置为 `S_IRUSR | S_IWUSR`，指定了拥有者可以向内存段读出或写入。一个成功的 `shmget()` 调用返回一个共享内存段整数标识值。其他想使用共享内存区域的进程必须指明这个标识符。

想访问共享内存段的进程必须采用 `shmat()`（SHared Memory ATtach）系统调用来将其加入地址空间。对 `shmat()` 的调用需要三个参数。第一个是希望加入的共享内存段的整数标识值。第二个是内存中的一个指针位置，它表示将要加入到的共享内存所在。如果传递一

个值 `NULL`，操作系统则为用户选择位置。第三个参数表示一个标志，它指定加入到的共享内存区域是只读模式还是只写模式。通过传递一个参数 `0`，表示向共享内存区域进行读或写操作均可。

第三个参数指的是一个标识模式。如果设置了该模式，则允许将要加入到的共享内存区域为只读模式；如果设置为 `0`，则允许向共享内存进行读和写操作。下面用 `shmat()` 加入一个共享内存：

```
shared_memory = (char *) shmat(id, NULL, 0);
```

如果成功，`shmat()` 返回一个指向附属的共享内存区域的内存中初始位置的指针。

一旦共享内存区域被加入到进程的地址空间，进程就可以采用从 `shmat()` 返回的指针，作为一般的内存访问来访问共享内存。在这个例子中，`shmat()` 返回一个指向字符串的指针。因此，可以按照下面的方法写入共享内存区域：

```
sprintf(shared_memory, "Writing to shared memory");
```

其他共享这个内存段的进程将会看到这个更新。

通常，采用已有共享内存段的进程首先将共享内存段加入其地址空间，然后再访问（还可能更新）共享内存区域。当一个进程不再需要访问共享内存段时，它将从其地址空间中分离出这一段。为了分离出这一共享内存段，进程可以按照下面的方法将共享内存区域的指针传递给系统调用 `shmdt()`：

```
shmdt(shared_memory);
```

最后，可以采用系统调用 `shmctl()`（把标志 `IPC_RMID` 和共享内存段的标识符一起作为参数），从系统中删除共享内存段。

图 3.16 所示的程序演示了上述的 POSIX 共享内存 API。该程序生成了 4 096 B 的共享内存段。一旦共享内存区域被加入，进程向共享内存中写入消息“Hi, There!”。在向更新的内存输出这个内容后，它分离并删除共享内存区域。本章结尾的编程练习中提供了更进一步的使用 POSIX 共享内存 API 的练习。

3.5.2 实例：Mach

作为基于消息操作系统的例子，下面考虑一下由卡耐基-梅隆大学开发的 Mach 操作系统。作为 Mac OS X 的一部分，在第 2 章中曾介绍过 Mach 操作系统。Mach 内核支持多任务的创建和删除，这里的任务与进程相似，但能有多个控制线程。Mach 的绝大多数通信（包括绝大多数系统调用和所有任务间信息）是通过消息实现的。消息通过邮箱（Mach 称之为端口）来发送和接收。

即使系统调用也是通过消息进行的。每个任务在创建时，也创建了两个特别邮箱：内核邮箱和通报（notify）邮箱。内核使用内核邮箱与任务通信，使用通报邮箱发送事件发生的通知。消息传输只需要三个系统调用。调用 `msg_send()` 向邮箱发送消息。消息可通过 `msg_receive()` 接收。远程过程调用（RPC）通过 `msg_rpc()` 执行，它能发送消息并只等待来

自发送者的一个返回消息。这样，RPC 模拟了典型的子程序过程调用，但它还能在系统之间工作——这就解释了*远程*。

```
#include <stdio.h>
#include<sys/shm.h>
#include<sys/stat.h>

int main()
{
    /* the identifier for the shared memory segment */
    int segment_id;
    /* a pointer to the shared memory segment */
    char* shared_memory;
    /* the size (in bytes) of the shared memory segment */
    const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi There!");

    /* now print out the string from shared memory */
    printf("%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

图 3.16 演示 POSIX 共享内存 API 的 C 程序

系统调用 `port_allocate()` 创建新邮箱并为其消息队列分配空间。消息队列的最大长度默认为 8 个消息。创建邮箱是该邮箱拥有者的任务。拥有者也被允许接收来自邮箱的消息。一次只能有一个任务能拥有邮箱或从邮箱接收，但是如果需要，这些权利也能发送给其他

任务。

开始时，邮箱的消息队列为空。随着消息向邮箱发送，消息被复制到邮箱中。所有消息具有同样的优先级。**Mach** 确保来自同一发送者的多个消息按照 FIFO 顺序来排队，但并不确保绝对顺序。例如，来自两个发送者的消息可以按任意顺序排队。

消息本身由固定大小的头部和可变长的数据部分组成。头部包括消息长度和两个邮箱名。当发送消息时，一个邮箱名是消息发送的目的地。通常，发送线程也期待回应，所以发送的邮箱名传递到接收任务，接收任务可用它作为“返回地址”以发回消息。

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

发送和接收操作本身很灵活。例如，当向一个邮箱发送消息时，该邮箱可能已满。如果邮箱未满，消息可复制到邮箱，发送线程继续。如果邮箱已满，发送线程有 4 个选择：

- 无限等待，直到邮箱有空间为止。
- 最多等待 n 毫秒。
- 根本不等待，而是立即返回。
- 暂时缓存消息。即使所要发送到的邮箱已满，操作系统还是可以保存一条消息。当消息能被放进邮箱时，一条通报消息会送到发送者。对于给定发送线程，在任何时候，只能有一个给已满邮箱等待处理的消息。

最后一项用于服务器任务，如行式打印机的驱动程序。在处理完请求之后，这些任务可能需要给请求服务的任务发送一个一次性的应答，但即使在客户邮箱已满时也必须继续处理其他服务请求。

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

- 邮箱集合内的任何邮箱。
 - 特定的（已命名的）邮箱。
- 如果没有消息等待接收，那么接收线程可能等待（最多等 n 毫秒或不等待）。

Mach 系统专门为分布式系统而设计，关于这点将在第 16 章到第 18 章讨论，但是它也适用于单处理器系统，**Mach** 被 Mac OS X 系统包括进去证明了这一点。消息系统的主要问题是由于消息双重复制导致性能差，即消息首先被从发送方复制到邮箱，再从邮箱复制到接收方。通过使用虚拟内存管理技术（第 9 章），**Mach** 消息系统试图避免双重复制。其关键在于 **Mach** 将发送者的地址空间映射到接收者的地址空间，消息本身并不真正复制。

这种消息管理技术大大地提高了性能，但是只适用于系统内部的消息传递。Mach 操作系统在本书网站的附加章节中有相关讨论。

3.5.3 实例：Windows XP

Windows XP 操作系统是现代设计的典范，它利用模块化增加功能，并降低了用以实现新特性所需的时间。Windows XP 支持多个操作环境或子系统，应用程序可通过消息传递机制进行通信，并可作为 Windows XP 子系统服务器的客户。

Windows XP 的消息传递工具称为本地过程调用（LPC）工具。Windows XP 的 LPC 在位于同一机器的两进程之间通信。它类似于被广泛使用的 RPC，但是为 Windows XP 进行了优化。与 Mach 一样，Windows XP 使用了端口对象以建立和维护两进程之间的连接。调用子系统的每个客户需要一个通信频道，由端口对象提供且不能继承。Windows XP 使用两种类型的端口：连接端口和通信端口。它们事实上是相同的，但根据实际使用情况而具有不同名称。连接端口称为对象，为所有进程可见，它们允许应用程序建立通信频道（参见第 22 章）。这种通信工作如下：

- ① 客户机打开系统的连接端口对象的句柄。
- ② 客户机发送连接请求。
- ③ 服务器创建两个私有通信端口，并返回其中之一的句柄给客户机。
- ④ 客户机和服务器使用相应端口句柄以发送消息或回调，并等待回答。

Windows XP 使用两种端口消息传递技术，端口可在客户机建立频道时被指明。最为简单的类型，是用于小消息的，使用端口队列作为中间存储，并将消息从一个进程复制到另一个进程。采用这种方式，可发送最多 256 B 的消息。

如果客户机需要发送更大的消息，那么它可通过区段对象（构建共享内存）来传递消息。客户机在建立频道时，必须确定它是否需要发送大消息。如果客户机确定它确实需要发送大消息，那么它请求建立区段对象。同样，如果服务器确定回复将会是很大的消息，它创建一个区段对象。为了能使用区段对象，需要发送一个小消息，它包括关于区段的一个指针和大小信息。这种方法比第一种更为复杂，但是它避免了数据复制。对于这两种情况，当客户程序或服务程序不能马上响应请求时，可使用回调机制。回调机制允许它们执行异步消息传递。Windows XP 中的本地过程调用结构如图 3.17 所示。

注意，Windows XP 中的 LPC 工具并不是 Win32 API 的一部分，故也不能被应用程序员所见，这是很重要的。应用程序员使用 Win32 API 调用标准的远程过程调用。当 RPC 被同一系统中的同一进程所调用，RPC 通过本地过程调用被间接地处理。LPC 也用于其他的 Win32 API 函数中。

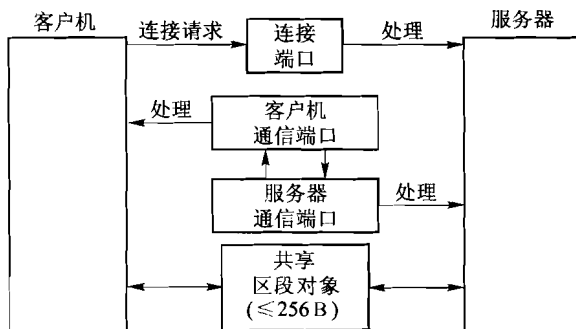


图 3.17 Windows XP 中的本地过程调用

3.6 客户机-服务器系统通信

3.4 小节介绍进程如何利用共享内存和消息传递进行通信。这些技术也可用于客户机-服务器系统的通信（参见 1.12.2 小节）。在本小节，将要探讨三种其他的客户机-服务器系统通信方法：Socket、远程过程调用（RPC）和 Java 的远程方法调用（RMI）。

3.6.1 Socket

Socket（套接字）可定义为通信的端点。一对通过网络通信的进程需要使用一对 Socket——即每个进程各有一个。Socket 由 IP 地址与一个端口号连接组成。通常，Socket 采用客户机-服务器结构。服务器通过监听指定端口来等待进来的客户请求。一旦收到请求，服务器就接受来自客户 Socket 的连接，从而完成连接。服务器实现的特定服务（如 telnet、ftp 和 http）是通过监听众所周知的端口来进行的（telnet 服务器监听端口 23，ftp 服务器监听端口 21，Web 或 http 服务器监听端口 80）。所有低于 1024 的服务器端口都被认为是众所周知的，可以用它们来实现标准服务。

当客户机进程发出连接请求时，它被主机赋予一个端口。该端口是大于 1024 的某个任意数。例如，如果 IP 地址为 146.85.5.20 的主机 X 的客户希望与地址为 161.25.19.8 的 Web 服务器（监听端口 80）建立连接，它可能被分配端口 1625。该连接由一对 Socket 组成：主机 X 上的（146.86.5.20：1625），Web 服务器上的（161.25.19.8：80），如图 3.18 所示。根据目的端口，在主机间传输的数据包可分送给合适的进程。

所有连接必须唯一。因此，如果主机 X 的另一个进程希望与同样的 Web 服务器建立另一个连接，那么它会被分配另一个大于 1024 但不等于 1625 的端口号。这确保了所有连接都有唯一的一对 Socket。

虽然本书的绝大多数程序例子使用 C 语言，但是这里使用 Java 语言来演示 Socket，这

是因为 Java 提供了一个 Socket 的简单接口，而且也提供了丰富的网络类库。如果对用 C 或 C++ 进行网络编程感兴趣，可以参考本章最后的推荐读物。

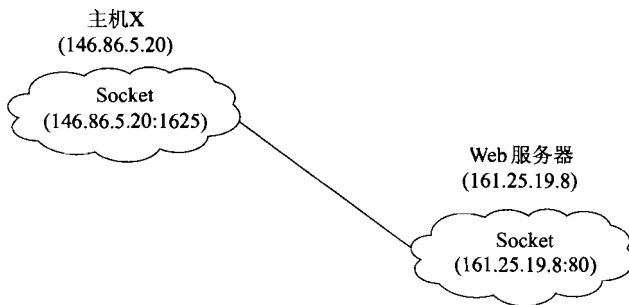


图 3.18 使用 Socket 通信

Java 提供了三种不同类型的 Socket。面向连接（TCP）Socket 是用 Socket 类实现的。无连接（UDP）Socket 使用了 DatagramSocket 类。最后一种类型是多点传送 Socket 类（MulticastSocket class），它是 DatagramSocket 类的子类。多点传送 Socket 允许数据发送给多个接收者。

下面通过例子介绍使用面向连接的 TCP Socket 的日期服务器。此操作允许客户机从服务器请求当前的日期和时间。服务器监听端口 6013，当然端口号可以是任何大于 1024 的数字。接收到连接时，服务器将日期和时间返回给客户机。

日期服务器程序如图 3.19 所示。服务器创建了 ServerSocket 以监听端口号 6013。接着它通过采用 accept() 方法开始监听端口。服务器阻塞在方法 accept() 上等待客户请求连接。当接收到连接请求时，accept() 会返回一个 Socket 以供服务器用来与客户进程通信。

有关服务器如何与 Socket 通信的细节如下。首先服务器建立 PrintWriter 对象，用来与客户进行通信。PrintWriter 对象允许服务器通过普通的输出方法 print() 和 println() 来向 Socket 进行写操作。服务器通过调用方法 println() 将日期时间发送给客户机。一旦将日期时间写到 Socket，服务器就关闭与客户相连的 Socket，并重新监听其他请求。

客户机通过创建 Socket 和服务器监听的端口相连来与服务器进行通信。图 3.20 所示的 Java 程序实现了客户机程序。客户机创建了 Socket，并请求与 IP 为 127.0.0.1、端口号为 6013 的服务器建立连接。一旦建立了连接，客户就通过普通流 I/O 语句来对 Socket 进行读。在得到服务器的日期时间后，客户机关闭端口并退出。IP 地址 127.0.0.1 为特殊 IP 地址，称为回送（loopback）。当计算机引用地址 127.0.0.1 时，它其实是在引用自己。这一机制允许同一主机上的客户机和服务器通过 TCP/IP 协议进行通信。IP 地址 127.0.0.1 可以被运行日期服务器的另一个主机的 IP 地址所替代。除 IP 地址外，也可使用如 www.westminster-

college.edu 这样的主机名。

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try{
            ServerSocket sock = new ServerSocket(6013);

            // now listen for connections
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

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

                //close the socket and resume
                //listening for connections
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

图 3.19 日期服务器程序

使用 Socket 进行通信，虽然常用和高效，但是它属于较为低级的分布式进程通信。原因之一在于 Socket 只允许在通信线程之间交换无结构的字节流。客户机或服务器程序需要负责加上数据结构。下面两小节将介绍两种更高级的通信方法：远程过程调用（RPC）和远程方法调用（RMI）。

```

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            // make connection to server socket
            sock = new Socket("127.0.0.1", 6013);

            inputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            //read the date from the socket
            String line;
            While ( (line = bin.readLine()) != null )
                System.out.println(line);

            //close the socket connection
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

图 3.20 日期客户端程序

3.6.2 远程过程调用

3.5.2 小节中简要介绍了一种最为普通的远程服务——RPC 方式。RPC 设计成抽象过程调用机制，用于通过网络连接系统。它在许多方面都类似于 3.4 节所述的 IPC 机制，并且通常建立在这种系统之上。因为在所处理的环境中，进程在不同系统上执行，所以必须提供基于消息的通信方案来提供远程服务。与 IPC 工具不同，用于 RPC 交换的消息有很好的结构，因此不再仅仅是数据包。每个消息传递给位于远程系统上监听端口号的 RPC 服务器，每个都包含要执行函数的名称和传递给函数的参数。该函数根据请求而执行，任何结果通过另一个消息送回给请求者。

端口只是一个数字，并包含在消息包的开始处。虽然一个系统通常只有一个网络地址，但是它在这一地址内有许多端口号以区分所支持的多种网络服务。如果一个远程进程需要服务，那么它就向适当端口发送消息。例如，如果一个系统允许其他系统能列出其当前用户，那么它可以有一个服务器支持这样的 RPC，并监听一个端口，例如 3027。任何远程系统只要向位于服务器的 3027 端口发送一个消息，就能得到所需要的信息（即列出当前用户），数据可通过回复消息收到。

RPC 语义允许客户机调用位于远程主机上的过程，就如同调用本地过程一样。通过在客户端提供存根（stub），RPC 系统隐藏了允许通信发生的必要细节。通常，对于每个独立的远程过程都有一个存根。当客户机调用远程过程时，RPC 系统调用合适的存根，并传递远程过程的参数。该存根位于服务器的端口，并编组（marshal）参数。参数编组涉及将参数打包成可通过网络传输的形式。接着存根使用消息传递向服务器发送一个消息。服务器的一个类似存根接收到这一消息，并调用服务器上的过程。如果有必要，返回值可通过同样技术传回给客户机。

有一个必须处理的事项是关于如何处理客户机和服务器系统的数据表示的差别。考虑一个 32 位整数的表示。有的系统使用高内存地址以存储高字节（称为大尾端，big-endian），而其他系统使用高内存地址以存储低字节（称为小尾端，little-endian）。为了处理这一问题，许多 RPC 系统都定义了数据的机器无关表示。一种这样的表示称为外部数据表示（XDR）。在客户机端，参数编组涉及将机器有关数据在被发送到服务器之前编组成 XDR。在服务器端，XDR 数据重新转换成服务器所用的机器有关表示。

另一个重要的事项就是调用的语义。虽然本地过程调用只有在极端情况下才可能失败，但是由于普通网络错误，RPC 可能会失败或重复多次执行。处理该问题的一种方法是操作系统确保一个消息刚好执行一次，而不是最多只执行一次。大多数本地过程调用具有“刚好一次”的属性，但是很难实现。

首先考虑“最多一次”。这可以通过为每个消息附加时间戳的方法来做。服务器对其所处理的消息，必须有一个完整的或足够长的时间戳历史，以便确保能检测到重复消息。进来的消息，如果其时间戳已在历史上，则被忽略。之后，客户机能够一次或多次发送消息，并确保仅执行一次（如何产生时间戳将在 18.1 小节中讨论）。

对“刚好一次”，需要消除服务器从未收到请求的风险。为了实现此目的，服务器必须执行前面介绍的“最多一次”协议，但必须通知客户端已经接收到 RPC 且已执行。网络中这些 ACK 消息很常用。客户机必须周期性重发每个 RPC 调用，直到它接收到对该调用的 ACK。

另一个重要事项是关于服务器与客户机间的通信问题。对于标准过程调用，在连接、装入或执行时（参见第 8 章）会出现一定形式的绑定，从而使过程名称被过程的内存地址所代替。RPC 方案要求有一个类似于客户机和服务器端口的绑定，但是客户机如何知道服务器上的端口呢？没有一个系统拥有其他系统完全的信息，因为它们并不共享内存。

对此有两种常用方法。第一种方法，绑定信息以固定端口地址形式预先固定。在编译时，RPC 调用有一个相应的固定端口。一旦程序编译后，服务器就不能改变请求服务的端口号。第二种方法，绑定通过集合点机制动态地进行。通常，操作系统在一个固定 RPC 端口上提供集合点服务程序（也称为 matchmaker）。客户机程序发送一个包括 RPC 的名称的消息给集合点服务程序，以请求它所需要执行的 RPC 端口地址。该端口号返回，RPC 调

用可发送到这一端口号直到进程终止(或服务器失败)。这种方式需要初始请求的额外开销,但是比第一种灵活。图 3.21 说明了一个简单的交互例子。

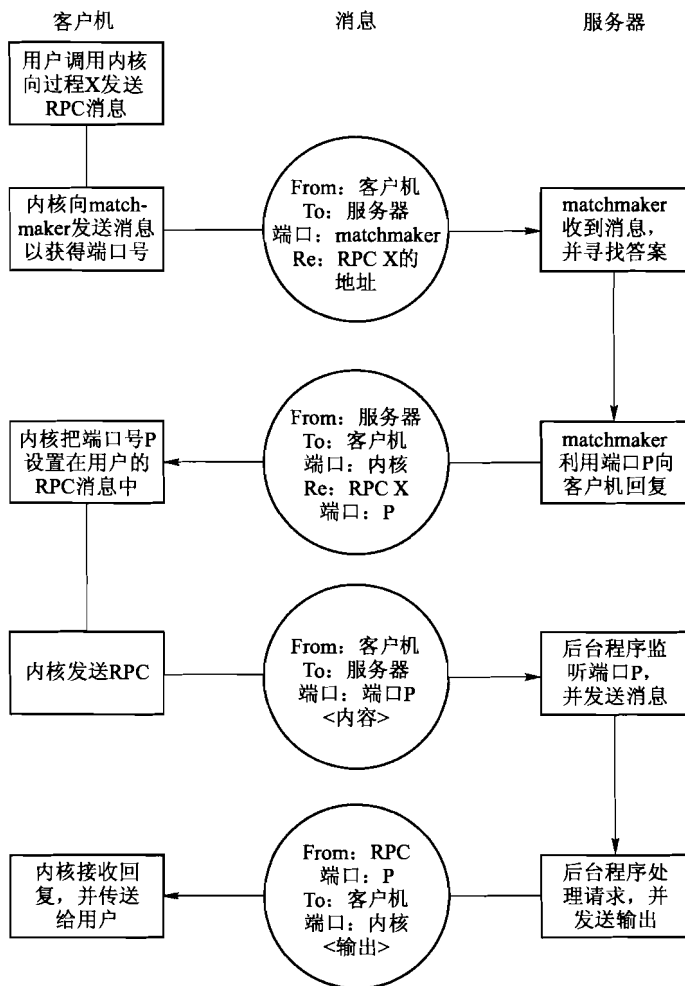


图 3.21 远程过程调用(RPC)的执行

RPC 方法对实现分布式文件系统(参见第 17 章)非常有用。这种系统可通过一组 RPC 服务程序和客户机来实现。消息发送到服务器的分布式文件系统端口以进行文件操作。消息包括要执行的磁盘操作。磁盘操作可能是 read、write、rename、delete 或 status, 对应通常的文件相关的系统调用。返回消息包括来自调用(分布式文件系统服务程序在客户机执行)的任何数据。例如, 一个消息可能包括一个传输整个文件到客户机上的请求, 或限制为简单块请求。对于后者, 如果需要传输整个文件, 可能需要多个这样的请求。

3.6.3 远程方法调用

远程方法调用 (remote method invocation, RMI) 是一个类似于 RPC 的 Java 特性。RMI 允许线程调用远程对象的方法。如果对象位于不同的 JVM 上, 那么就认为它是远程的。因此, 远程可能在同一计算机或通过网络连接的主机的不同 JVM 上。这种情况如图 3.22 所示。

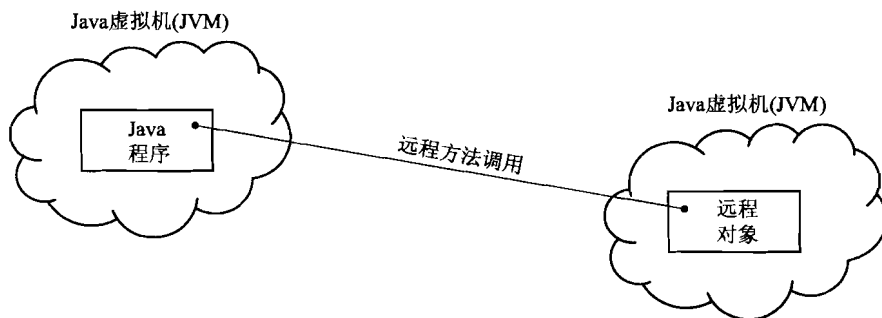


图 3.22 远程方法调用

RMI 和 RPC 在两方面有根本的不同。第一, RPC 支持子程序编程, 即只能调用远程的子程序或函数; 而 RMI 是基于对象的, 它支持调用远程对象的方法。第二, 在 RPC 中, 远程过程的参数是普通数据结构, 而 RMI 可以将对象作为参数传递给远程方法。RMI 通过允许 Java 程序调用远程对象的方法, 使得用户能够开发分布在网络上的 Java 应用程序。

为了使远程方法对客户机和服务器透明, RMI 采用存根 (stub) 和骨干 (skeleton) 实现远程对象。存根为远程对象的代理, 它驻留在客户机中。当客户机调用远程方法时, 远程对象的存根被调用。这种客户端存根负责创建一个包, 它具有服务器上要调用方法的名称和用于该方法的编排参数。存根将该包发送给服务器, 远程对象的骨干会接收它。骨干负责重新编排参数并调用服务器上所要执行的方法。骨干接着编排返回值 (或异常), 然后打包, 并将该包返回给客户机。存根重新编排返回值, 并传递给客户机。

下面更详细地说明这一过程是如何工作的。假设客户机希望调用远程对象 `server` 的一个方法, 该方法具有签名 `someMethod (Object, Object)` 并返回布尔值。客户机执行如下语句:

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

使用参数 `A` 和 `B` 的 `someMethod()` 调用了远程对象的存根。存根将参数 `A` 和 `B` 以及要在服务器上执行的方法名称一起打包, 接着将该包发送给服务器。服务器上的骨干会重新编排参数并调用方法 `someMethod()`。`someMethod()` 的真正实现驻留在服务器上。一旦方法完成, 骨干会编排从 `someMethod()` 返回的布尔值, 并将该值发回给客户机。存根重新编排该返回值, 并传递给客户机。这一过程如图 3.23 所示。

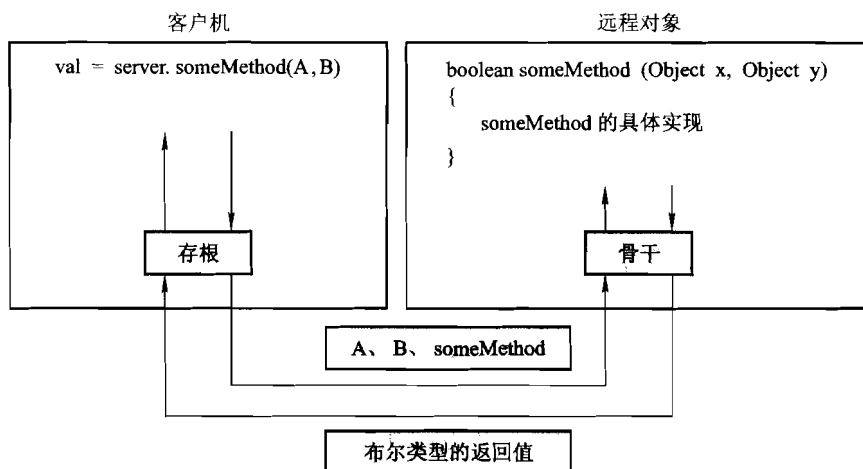


图 3.23 编排参数

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

- 如果编排参数是**本地**（非**远程**）对象，那么通过称为对象串行化的技术来复制传递。不过，如果参数也是远程对象，那么可通过引用传递。对于上述例子，如果 A 是本地对象而 B 是远程对象，那么 A 就串行化并复制传递，而 B 通过引用传递。这是可以允许服务器远程执行 B 的方法。

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

3.7 小 结

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

当前不在执行的进程会放在某个等待队列中。操作系统有两种主要队列：I/O 请求队列和就绪队列。就绪队列包括所有准备执行并等待 CPU 的进程。每个进程都有 PCB，PCB 链接起来就形成了就绪队列。长期（作业）调度通过选择进程来争用 CPU。通常，长期调度会受资源分配考虑，尤其是内存管理的影响。短期调度从就绪队列中选择进程。

操作系统必须为父进程创建子进程提供一种机制。父进程在继续之前可以等待它的子进程终止，也可以并发执行父进程和子进程。并发执行有许多优点，例如信息共享、提高

运算速度、模块化和便利性等。

操作系统的执行进程可以是独立进程或协作进程。协作进程需要进程间有互相通信的机制。主要有两种形式的通信：共享内存和消息系统。共享内存方法要求通信进程共享一些变量。进程通过使用这些共享变量来交换信息。对于共享内存系统，主要由应用程序员提供通信，操作系统只需要提供共享内存。消息系统方法允许进程交换信息。提供通信的主要责任在于操作系统本身。这两种方法并不互相排斥，能在同一操作系统内同时实现。

客户机-服务器系统中的通信可能使用：（1）Socket，（2）远程过程调用（RPC），（3）Java 的远程方法调用（RMI）。Socket 定义为通信的端点。一对应用程序间的连接由一对 Socket 组成，每端各有一个通信频道。RPC 是另一种形式的分布式通信。当一个进程（或线程）调用一个远程应用的方法时，就出现了 RPC。RMI 是 RPC 的 Java 版。RMI 允许线程如同调用本地对象一样来调用远程对象的方法。RPC 和 RMI 的主要区别是 RPC 传递给远程过程的数据是按普通数据结构形式的，而 RMI 允许把对象传递给远程方法。

习 题

3.1 论述长期、中期、短期调度之间的区别。

3.2 描述内核在两个进程间进行上下文切换的过程。

3.3 考虑 RPC 机制。描述因为没有强制或者“最多一次”或者“刚好一次”的语义带来的不必要的后果。讨论没有提供任何保证的机制的可能使用。

3.4 使用图 3.24 所示的程序，说明 LINE A 可能输出什么。

```
#include < sys/types.h >
#include < stdio.h >
#include < unistd.h >

int value = 5;
int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* child process */
        value += 15;
    }
    else if (pid > 0) { /* parent process */
        wait (NULL);
        printf("PARENT : value = %d", value); /*LINE A*/
        exit();
    }
}
```

图 3.24 C 程序

3.5 下面设计的优点和缺点分别是什么？系统层次和用户层次都要考虑。

- a. 同步和异步通信
- b. 自动和显式缓冲
- c. 复制传送和引用传送
- d. 固定大小和可变大小消息

3.6 Fibonacci 序列是一组数：0, 1, 1, 2, 3, 5, 8, ..., 通常它可以表示为：

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

使用系统调用 `fork()` 编写一个 C 程序，它在其子程序中生成 Fibonacci 序列，序列的号码将在命令行中提供。例如，如果提供的是 5，Fibonacci 序列中的前 5 个数将由子进程输出。由于父进程和子进程都有它们自己的数据副本，对子进程而言，输出序列是必要的。退出程序前，父进程调用 `wait()` 调用来等待子进程结束。执行必要的错误检查以保证不会接受命令行传递来的负数号码。

3.7 重复上述练习，这次使用 Win32 API 中的 `CreateProcess()`。在这个例子中，需要指定一个单独的进程以被从 `CreateProcess()` 中调用。此程序将会作为一个输出 Fibonacci 序列的子进程来运行。执行必要的错误检查以保证不会接受命令行传递来的负数号码。

3.8 修改图 3.19 所示的日期服务器，以使其发送随机的 fortune，而不是当前的日期。允许此 fortune 包含多行。图 3.20 所示的日期客户机可用来读取 fortune 服务器返回的多行 fortune。

3.9 一个 echo 服务器是一个无论从客户机接收到什么都原样返回（echoes back）的服务器。例如，客户机向服务器发送字符串“Hello there!”，服务器将会原样回答它从客户机收到的数据——即 Hello there!

使用 3.6.1 小节介绍的 Java 网络 API 编写一个 echo 服务器程序，该服务器将会使用 `accept()` 方法等待一个客户机连接。当接收到一个客户机连接时，服务器将循环完成如下步骤：

- ① 从 socket 读取数据到缓冲。
- ② 将缓冲的内容写回客户机。

只有当客户机关闭连接，服务器得以终止，才能使服务器摆脱循环。

图 3.19 的日期服务器使用了 `java.io.BufferedReader` 类。`BufferedReader` 扩展了 `java.io.Reader` 类，后者被用来读取字符流。然而，echo 服务器不能保证它将会从客户机读取字符，它还可能接收二进制数据。`java.io.InputStream` 类将数据处理为字节级而不是字符级。因此，echo 服务器必需使用对象来扩展 `java.io.InputStream`。当客户机关闭它末端的 socket 连接后，`java.io.InputStream` 类中的 `read()` 方法返回 -1。

3.10 在习题 3.6 中，由于父进程和子进程都有它们自己的数据副本，子进程必须输出 Fibonacci 序列。设计此程序的另一个方法是在父进程和子进程之间建立一个共享内存段。此方法允许子进程将 Fibonacci 序列的内容写入共享内存段，当子进程完成时，父进程输出此序列。由于内存是共享的，每个子进程的变化都会影响到共享内存，也会影响到父进程。

这个程序将采用 3.5.1 小节介绍的 POSIX 共享内存方法来构建。程序首先需要创建共享内存段的数据结构，这可以通过利用 `struct` 来完成。此数据结构包括两项：（1）长度为 `MAX_SEQUENCE` 的固定长度数组，它保存 Fibonacci 的值；（2）子进程生成的序列的大小——`sequence_size`，其中 `sequence_size` ≤ `MAX_SEQUENCE`。这些项可表示成如下形式：

```
#define MAX_SEQUENCE 10
```

```
typedef struct {
```

```

    long fib_sequence [MAX_SEQUENCE];
    int sequence_size;
} shared_data;

```

父进程将会按下列步骤进行:

- a. 接受命令行上传递的参数, 执行错误检查以保证参数不大于 MAX_SEQUENCE。
- b. 创建一个大小为 shared_data 的共享内存段。
- c. 将共享内存段附加到地址空间。
- d. 在命令行将命令行参数值赋予 shared_data。
- e. 创建子进程, 调用系统调用 wait() 等待子进程结束。
- f. 输出共享内存段中 Fibonacci 序列的值。
- g. 释放并删除共享内存段。

由于子进程是父进程的一个副本, 共享内存区域也将被附加到子进程的地址空间。然后, 子进程将会把 Fibonacci 序列写入共享内存并在最后释放此区域。

采用协作进程的一个问题涉及同步问题。在这个练习中, 父进程和子进程必须是同步的, 以使在子进程完成生成序列之前, 父进程不会输出 Fibonacci 序列。采用系统调用 wait(), 这两个进程将会同步。父进程将调用 wait(), 这将使其被挂起, 直到子进程退出。

3.11 大多数 UNIX 和 Linux 系统提供了 IPCS 命令。此命令列出各种 POSIX 进程间通信机制的状态, 包括共享内存段。许多关于此命令的信息来自于数据结构 struct shmid_ds, 它可在 /usr/include/sys/shm.h 文件中找到。此结构包括:

- int shm_segsz —— 共享内存段的大小。
 - short shm_nattch —— 附加到共享内存段的数目。
 - struct ipc_perm shm_perm —— 共享内存段的许可结构。
- struct ipc_perm 数据结构 (在 /usr/include/sys/ipc.h 文件中) 包括:
- unsigned short uid —— 共享内存段用户的标识。
 - unsigned short mode —— 许可模式。
 - key_t key (Linux 系统中为 __key) —— 用户指定的关键标识。

许可模式是根据如何利用系统调用 shmget() 建立共享内存段的设置, 按如下标识:

| 模 | 含 义 | 模 | 含 义 |
|------|-------|------|------|
| 0400 | 拥有者可读 | 0020 | 组可写 |
| 0200 | 拥有者可写 | 0004 | 全局可读 |
| 0040 | 组可读 | 0002 | 全局可写 |

可以通过使用位 AND 操作符 “&” 来得到许可。例如, 如果语句 mode & 0400 值为真, 则许可模式允许共享内存段的拥有者能进行读操作。

共享内存段可根据用户指定的方法或系统调用 shmget() 返回的整数值来标识, 后者表示创建共享内存段的整数标识。一个给定的整数段标识符的 shm_ds 结构可以通过如下的系统调用 shmctl() 获得:

```

/* identifier of the shared memory segment*/
int segment_id;
shm_ds shmbuffer;
shmctl(segment_id, IPC_STAT, &shmbuffer);

```

如果成功, `shmctl()` 返回 0; 否则, 返回-1。

编写一个 C 程序, 来为共享内存段传递标识。程序将调用 `shmctl()` 函数以得到它的 `shm_ds` 结构。然后它将输出给定共享内存段的下列值:

- 段的 ID。
- 关键字。
- 模式。
- 拥有者 UID。
- 大小。
- 附加的数目。

项目: UNIX Shell 和历史特点

此项目由修改一个 C 程序组成, 它作为接收用户命令并在单独的进程执行每个命令的 Shell 接口。Shell 接口在下一个命令进入之后为用户提供了提示符。下面的例子说明了提示符 `sh>` 和用户的下一个命令: `cat prog.c`, 此命令使用 UNIX `cat` 命令在终端上显示文件 `prog.c`。

```
sh> cat prog.c
```

实现 Shell 接口的一种技术是父进程首先读用户命令行的输入 (即 `cat prog.c`), 然后创建一个独立的子进程来完成这个命令。除非另做说明, 父进程在继续之前等待子进程退出。这在功能上有点类似于图 3.11。然而, UNIX Shell 一般也允许子进程在后台运行 (或并发地运行), 通过在命令的最后使用 “&” 符号。将上面的命令重写为:

```
sh> cat prog.c &
```

现在父进程和子进程可以并发执行了。

用系统调用 `fork()` 来创建独立的子进程, 通过使用 `exec()` 族中的一种系统调用 (3.3.1 小节所介绍) 来执行用户命令。

1. 简单 Shell

图 3.25 给出了一种基本的命令行 Shell 操作的 C 程序。此程序包括两个函数: `main()` 和 `setup()`。`setup()` 函数读取用户的下一条命令 (最多 80 个字符), 然后将之分析为独立的标记, 这些标记被用来填充命令的参数向量 (如果将要在后台运行命令, 它将以 “&” 结尾, `setup()` 将会更新参数 `background`, 以使 `main()` 函数相应地执行)。当用户按快捷键 `Ctrl+D` 后, `setup()` 调用 `exit()`, 此程序被终止。

`main()` 函数打印提示符 `COMMAND->`, 然后调用 `setup()`, 它等待用户输入命令。用户输入命令的内容被装入一个 `args` 数组。例如, 如果用户在 `COMMAND->` 提示符处输入 `ls -l`, `args[0]` 等同于字符串 `ls` 和 `args[1]` 被设置为字符串 `-l` (这里的字符串指的是以 0 结束的 C 字符串变量)。

```

#include <stdio.h>
#include <unistd.h>

#define MAX LINE 80

/** setup() reads in the next command line, separating it into
distinct tokens using whitespace as delimiters.
setup() modifies the args parameter so that it holds pointers
to the null-terminated strings that are the tokens in the most
recent user command line as well as a NULL pointer, indicating
the end of the argument list, which comes after the string
pointers that have been assigned to args. */

void setup(char inputBuffer[], char *args[],int *background)
{
    /** full source code available online */
}

int main(void)
{
    char inputBuffer[MAX LINE]; /* buffer to hold command entered */
    int background; /* equals 1 if a command is followed by '&' */
    char *args[MAX LINE/2 + 1]; /* command line arguments */

    while (1) {
        background = 0;
        printf("COMMAND->");
        /* setup() calls exit() when Control-D is entered */
        setup(inputBuffer, args, &background);

        /** the steps are:
        (1) fork a child process using fork()
        (2) the child process will invoke execvp()
        (3) if background == 1, the parent will wait,
        otherwise it will invoke the setup() function again. */

    }
}

```

图 3.25 简单 Shell 的框架

这个项目由两部分组成：（1）创建子进程，并在子进程中执行命令；（2）修改 Shell 以允许一个历史特性。

2. 创建子进程

项目的第一部分是修改图 3.25 中的 `main()` 函数，以使从 `setup()` 返回时，创建一个子进

程，并执行用户的命令。

如前面所指出的，`setup()`函数用用户指定命令装载 `args` 数组的内容，`args` 数组将被传递给 `execvp()`函数，该函数具有如下接口：

```
execvp(char *command, char *params[]);
```

其中 `command` 表示要执行的命令，`params` 保存命令的参数。对于该项目，`execvp()`函数应作为 `execvp(args[0], args)`来调用；需要保证检测 `background` 的值，以决定父进程是否需要等待子进程退出。

3. 创建历史特性

下面的任务就是修改图 3.25 中的程序，使其能提供一个历史特性来允许用户能访问 10 个最近输入的命令。这些命令将从 1 开始编号，并将继续增长甚而超过 10，例如，如果用户输入 35 个命令，那么 10 个最近输入的命令应为命令 26~35。这些历史特性将用一些不同的技术来实现。

首先，当用户按下 `SIGINT` 信号 `Ctrl+C` 键时，系统能列出这些命令。`UNIX` 使用信号（`signals`）来通知进程发生了一个特定事件。信号可以是同步的，也可以是异步的，这取决于资源及发出信号事件的原因。一旦因一个特定事件发生而生成了一个信号（如被零除，非法内存访问，或用户按快捷键 `Ctrl+C` 等），该信号被传送到必须处理该信号的进程。接收信号的进程可按如下方法处理：

- 忽略信号。
- 使用错误信号处理器。
- 提供一个单独的信号处理函数。

可以首先在 `C` 结构 `struct sigaction` 中设置特定的域，然后将此结构传递给 `sigaction()` 函数来处理信号。信号在 `include` 文件 `/usr/include/sys/signal.h` 中定义。例如，信号 `SIGINT` 表示用控制序列 `Ctrl+C` 终止程序的信号。`SIGINT` 默认的信号处理器为终止程序。

作为另一种选择，程序可以选择建立自己的信号处理函数，这可通过在 `struct sigaction` 中设置 `sa_handler` 域为处理信号的函数的名称，然后调用 `sigaction()` 函数，并将它传递给（1）正在为信号建立的处理器，（2）指向 `struct sigaction` 的指针来完成。

图 3.26 给出了一个使用函数 `handle_SIGINT()`来处理 `SIGINT` 信号的 `C` 程序。该函数用来打印消息“`Caught Control C`”，然后调用 `exit()`函数来终止程序（必须使用 `write()`函数来完成输出，而不是用前面的 `printf()`。因为 `write()`是信号安全的，表明它可以从一个信号处理函数内调用，而 `printf()`则完成不了）。该程序将在 `while(1)`循环运行，直至用户按快捷键 `Ctrl+C`。此时，信号处理函数 `handle_SIGINT()`将被调用。

信号处理函数应在 `main()`之前声明，并且由于控制可在任意点传递给该函数，没有参数可以传递给它。因此，在程序中，它所访问的任意数据必须定义为全局，即在源文件的顶部、函数声明之前。在从信号处理函数返回之前，它应重发指令提示。

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE];

/* 信号处理函数 */
void handle_SIGINT()
{
    write(STDOUT_FILENO, buffer, strlen(buffer));

    exit (0);
}

int main(int argc, char *argv[])
{
    /*创建信号处理器*/
    struct sigaction handler;
    handler.sa_handler = handle_SIGINT;
    sigaction(SIGINT, &handler, NULL);

    /*生成输出消息*/
    strcpy(buffer, "Caught Control C\n");

    /*循环运行，直至接收到<Ctrl+C>*/
    while(1)
        ;

    return 0;
}
```

图 3.26 信号处理程序

如果用户按快捷键 Ctrl+C，信号处理器将输出最近的 10 个命令列表。根据该列表，用户通过输入“r x”可以运行之前 10 个命令中的任何一个，其中“x”为该命令的第一个字母。如果有命令以“x”开头，则执行最近的一个。同样，用户可以通过仅输入“r”来再次运行最近的命令。可以假定只有一个空格来将“r”和第一个字母分开，并且该字母后面跟着‘\n’。而且，如果希望执行最近的命令，单独的“r”将紧跟\n。

用这种方式执行的任何命令应该在用户的屏幕上回显，而该命令被放入历史缓存中来

作为下一个命令（`r`、`x` 并不进入历史缓存列表，但它所指定的实际命令会进入）。

如果用户试图使用历史缓存工具来运行命令，而该命令被检测为 *错误*，则应将一则出错消息传给用户，并且该命令不进入历史列表，也不应调用 `execvp()` 函数（最好可以知道不正确组成的命令，它们被传递给 `execvp()`，表面看起来有效，但实际并非如此，并且在历史缓存中并不包括它们。但这超过了这个简单的 Shell 程序的能力）。同时应该修改 `setup()` 函数，以使其返回一个整数值，指示是否成功地创建了一个有效的变量列表，`main()` 也应做相应的修改。

文 献 注 记

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

Gray[1997] 详细论述了 UNIX 系统的进程间通信。Barrera[1991] 和 Vahalia[1996] 论述了 Mach 系统的进程间通信。Solomon、Rusinovich[2000] 和 Stevens[1999] 分别概述了 Windows 2000 和 UNIX 的进程间通信。

Brirrell 和 Nelson[1984] 论述了 RPC 的实现。Shrivastava 和 Panzieri[1982] 设计了可靠的 RPC 机制。Tay 和 Ananda[1990] 综述了 RPC。Stankovic[1982] 和 Staunstrup[1982] 比较了过程调用和消息传递通信。Crosso[2002] 非常详细地讨论了 RMI。Calvert 和 Donahoo[2001] 提供了 Java 中 Socket 编程的内容。

第4章 线程

第3章讨论的进程模型假设进程是一个具有单个控制线程的执行程序。现在，许多现代操作系统都提供使单个进程包括多个控制线程的功能。本章引入了与多线程计算机系统相关的许多概念，包括有关 Pthread API、Win32 API 和 Java 线程库的讨论。将研究与多线程编程相关的许多事项以及它是如何影响操作系统设计的。最后将研究 Windows XP 和 Linux 操作系统如何在内核级提供对线程的支持。

本章目标

- 引入线程的概念——一种 CPU 利用的基本单元，它是形成多线程计算机的基础。
- 讨论 Pthread API、Win32 API 和 Java 线程库。

4.1 概述

线程是 CPU 使用的基本单元，它由线程 ID、程序计数器、寄存器集合和栈组成。它与属于同一进程的其他线程共享代码段、数据段和其他操作系统资源，如打开文件和信号。一个传统重量级（heavyweight）的进程只有单个控制线程。如果进程有多个控制线程，那么它能同时做多个任务。图 4.1 说明了传统单线程进程和多线程进程的差别。

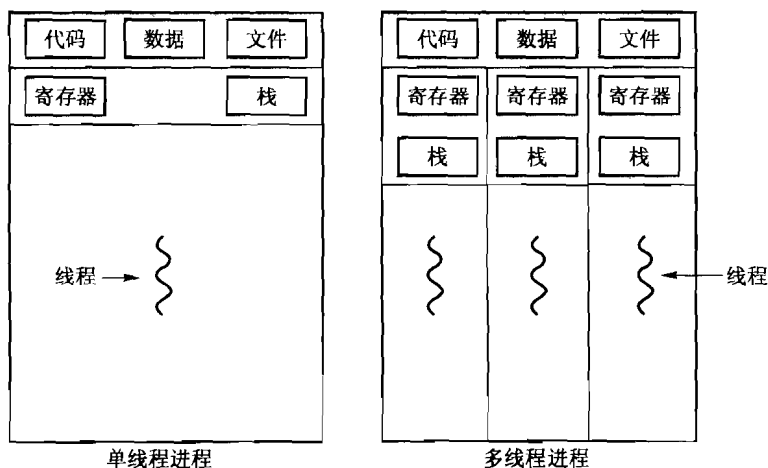


图 4.1 单线程进程和多线程进程

4.1.1 动机

运行在现代桌面 PC 上的许多软件包都是多线程的。一个应用程序通常是作为一个具有多个控制线程的独立进程实现的。例如，网页浏览器可能有一个线程用于显示图像和文本，另一个线程用于从网络接收数据；文档处理器可能有一个线程用于显示图形，另一个线程用于读入用户的键盘输入，还有一个线程用于在后台进行拼写和语法检查。

有的时候，一个应用程序可能需要执行多个相似任务。例如，网页服务器接收用户关于网页、图像、声音等的请求。一个忙碌的网页服务器可能有多个（或数千个）客户并发访问它。如果网页服务器作为传统单个线程的进程来执行，那么只能一次处理一个请求。这样，客户必须等待很长的处理请求的时间。

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

线程在远程过程调用（RPC）系统中也有很重要的作用。回顾第 3 章，RPC 通过提供一种类似于普通函数或子程序调用的通信机制，以允许进程通信。通常，RPC 服务器是多线程的。当一个服务器接收到消息，它使用独立线程处理消息。这允许服务器能处理多个并发请求。Java 的 RMI 系统也以类似方式工作。

最后，现代的许多操作系统都是多线程的，少数线程在内核中运行，每个线程完成一个指定的任务，如管理设备或中断处理。例如，Solaris 在内核中特别为中断处理创建线程集合，Linux 使用内核线程来管理系统中的空闲内存数量。

4.1.2 优点

多线程编程具有如下 4 个优点：

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

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

③ **经济**：进程创建所需要的内存和资源的分配比较昂贵。由于线程能共享它们所属进程的资源，所以创建和切换线程会更为经济。实际地测量进程创建和管理与线程创建和管理的差别较为困难，但是前者通常要比后者花费更多的时间。例如，对于 Solaris，进程

创建要比线程创建慢 30 倍，而进程切换要比线程切换慢 5 倍。

④ **多处理器体系结构的利用**：多线程的优点之一是能充分使用多处理器体系结构，以便每个进程能并行运行在不同的处理器上。不管有多少 CPU，单线程进程只能运行在一个 CPU 上。在多 CPU 上使用多线程加强了并发功能。

4.2 多线程模型

迄今为止只是泛泛地讨论了线程。不过有两种不同方法来提供线程支持：用户层的**用户线程**或内核层的**内核线程**。用户线程受内核支持，而无须内核管理；而内核线程由操作系统直接支持和管理。事实上所有当代操作系统，如 Windows XP、Linux、Mac OS X、Solaris、Tru64 UNIX (前身是 Digital UNIX)，都支持内核线程。

最后，在用户线程和内核线程之间必然存在一种关系。本节研究三种常用的建立此关系的方法。

4.2.1 多对一模型

多对一模型（见图 4.2）将许多用户级线程映射到一个内核线程。线程管理是由线程库在用户空间进行的，因而效率比较高。但是如果一个线程执行了阻塞系统调用，那么整个进程会阻塞。而且，因为任一时刻只有一个线程能访问内核，多个线程不能并行运行在多处理器上。**Green thread**（Solaris 所应用的线程库）就使用了这种模型，另外还有 **GNU 可移植线程**（GNU Portable Threads）。

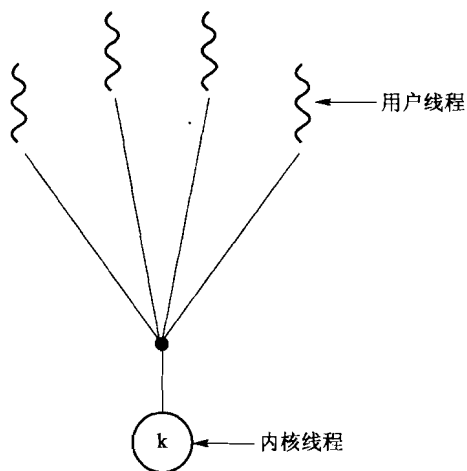


图 4.2 多对一模型

4.2.2 一对一模型

一对一模型（见图 4.3）将每个用户线程映射到一个内核线程。该模型在一个线程执行阻塞系统调用时，能允许另一个线程继续执行，所以它提供了比多对一模型更好的并发功能；它也允许多个线程能并行地运行在多处理器系统上。这种模型的唯一缺点是每创建一个用户线程就需要创建一个相应的内核线程。由于创建内核线程的开销会影响应用程序的性能，所以这种模型的绝大多数实现限制了系统所支持的线程数量。Linux 与 Windows 操作系统家族（包括 Windows 95、Windows 98、Windows NT、Windows 2000 和 Windows XP）实现了一对一模型。

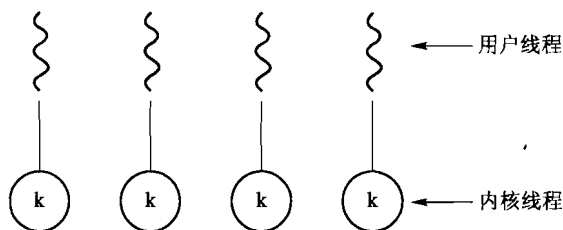


图 4.3 一对一模型

4.2.3 多对多模型

多对多模型（见图 4.4）多路复用了许多用户线程到同样数量或更小数量的内核线程上。内核线程的数量可能与特定应用程序或特定机器有关（位于多处理器上的应用程序可比单处理器上分配更多数量的内核线程）。虽然多对一模型允许开发人员创建任意多的用户线程，但是因为内核只能一次调度一个线程，所以并没有增加并发性。一对一模型提供了更大的并发性，但是开发人员必须小心，不要在应用程序内创建太多的线程（有时可能会限制创建线程的数量）。多对多模型没有这两者的缺点：开发人员可创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行。而且，当一个线程执行阻塞系统调用时，内核能调度另一个线程来执行。

一个流行的多对多模型的变种仍然多路复用了许多用户线程到同样数量或更小数量的内核线程上，但也允许将一个用户线程绑定到某个内核线程上。这个变种有时被称为二级模型（见图 4.5），被 IRIX、HP-UX、Tru64 UNIX 等操作系统所支持。Solaris 在其 Solaris 9 之前的版本中支持二级模型，但从 Solaris 9 开始使用一对一模型。

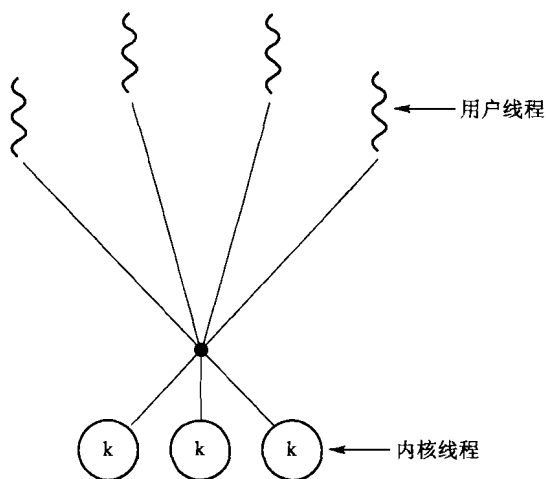


图 4.4 多对多模型

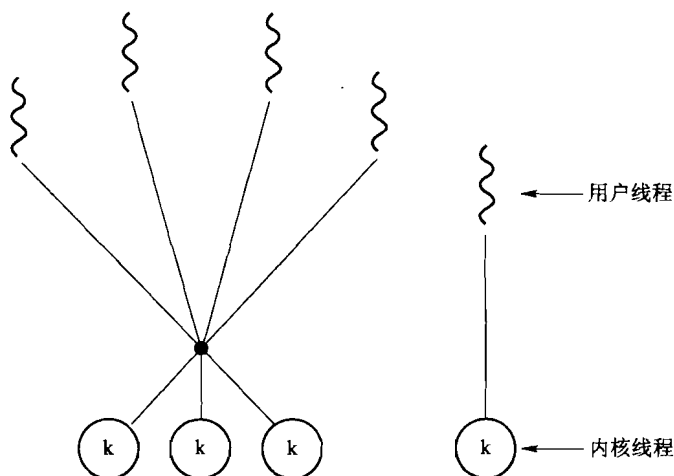


图 4.5 二级模型

4.3 线程库

线程库 (thread library) 为程序员提供创建和管理线程的 API。主要有两种方法来实现线程库。第一种方法是在用户空间中提供一个没有内核支持的库，此库的所有代码和数据结构都存在于用户空间中。调用库中的一个函数只是导致了用户空间中的一个本地函数调用，而不是系统调用。第二种方法是执行一个由操作系统直接支持的内核级的库。此时，库的代码和数据结构存在于内核空间中。调用库中的一个 API 函数通常会导致对内核的系

统调用。

目前使用的三种主要的线程库是：(1) POSIX Pthread、(2) Win32、(3) Java。Pthread 作为 POSIX 标准的扩展，可以提供用户级或内核级的库。Win32 线程库是适用于 Windows 操作系统的内核级线程库。Java 线程 API 允许线程在 Java 程序中直接创建和管理。然而，由于大多数 JVM 实例运行在宿主操作系统之上，Java 线程 API 通常采用宿主系统上的线程库来实现。这意味着在 Windows 系统上，Java 线程通常用 Win32 API 实现，而在 UNIX 和 Linux 系统中采用 Pthread。

接下来介绍采用这三种线程库来创建基本的线程。作为一个说明性的例子，设计一个多线程程序，在独立的线程中完成非负数整数的加法功能：

$$sum = \sum_{i=0}^N i$$

例如，如果 N 为 5，此函数表示从 0~5 的数相加起来，为 15。三个程序都要求在命令行输入加法的上限。例如，如果用户输入 8，将会输出从 0~8 的整数值和。

4.3.1 Pthread

Pthread 是由 POSIX 标准 (IEEE 1003.1c) 为线程创建和同步定义的 API。这是线程行为的规范，而不是实现。操作系统设计者可以根据意愿采取任何实现形式。许多操作系统实现了这个线程规范，包括 Solaris、Linux、Mac OS X 和 Tru64 UNIX，公开可获取的 Shareware 实现适用于各种 Windows 操作系统。

如图 4.6 所示的 C 程序显示部分构造一个多线程程序的基本 Pthread API，它通过一个独立线程计算非负整数的累加和。对于 Pthread 程序，独立线程是通过特定函数执行的。在图 4.6 中，这个特定函数是 runner() 函数。当程序开始时，单个控制线程在 main() 中开始。在初始化之后，main() 创建了第二个线程并在 runner() 中开始控制。两个线程共享全局数据 sum。

现在对该程序做一个更为详细的描述。所有 Pthread 程序都需要包括 pthread.h 头文件。语句 pthread_t tid 声明了所创建线程的标识符。每个线程都有一组属性，包括栈大小和调度信息。pthread_attr_t attr 表示线程的属性，通过函数调用 pthread_attr_init(&attr) 来设置这些属性。由于没有明确地设置任何属性，故使用提供的默认属性 (第 5 章讨论由 Pthread API 提供的一些调度属性)。通过函数调用 pthread_create() 创建一个独立线程。除了传递线程标识符和线程属性外，还要传递函数名称 (这里为 runner()) 以便新线程可以开始执行。最后传递由命令行参数 argv[1] 所提供的整数参数。

程序此时有两个线程：main() 的初始 (父) 线程和通过 runner() 函数执行累加和 (子) 线程。在创建了累加和线程之后，父线程通过调用 pthread_join() 函数，以等待 runner() 线程的完成。累加和线程在调用了函数 pthread_exit() 之后就完成了。一旦累加和线程返回，父线程将输出累加和的值。

```

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

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

int main (int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be <= 0\n", atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* now wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for(i=1; i<=upper; i++)
        sum += i;

    pthread_exit(0);
}

```

图 4.6 使用 Pthread API 的多线程 C 程序

4.3.2 Win32 线程

采用 Win32 线程库创建线程的技术在某些方面类似于 Pthread 技术。图 4.7 给出了 C 程序中的 Win32 线程。注意，在使用 Win32 API 时必须包括 windows.h 头文件。

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the threads*/

/* the thread runs in this separate function*/
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for(DWORD i = 0; i <= Upper; i++)
        Sum += i;

    return ();
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    /* perform some basic error checking*/
    if(argc != 2){
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if(Param < 0){
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }

    // create the thread
    ThreadHandle = CreateThread(
        NULL, // default security attributes
        0, // default stack size
        Summation, // thread function
        &Param, // parameter to thread function
        0, // default creation flags
        &ThreadId); // returns the thread identifier

    if(ThreadHandle != NULL){
        // now wait for the thread to finish
        WaitForSingleObject(ThreadHandle, INFINITE);

        // close the thread handle
        CloseHandle(ThreadHandle);

        printf("sum = %d\n", Sum);
    }
}
```

图 4.7 使用 Win32API 的多线程 C 程序

正如图 4.6 所示的 Pthread 例子，独立线程共享的数据（在此为 `sum`）被声明为全局变量（`DWORD` 数据类型是一个无符号的 32 位整数），还定义了一个在独立的线程中完成的 `Summation()` 函数，向该函数传递一个 `void` 指针，Win32 将其定义为 `LPVOID`。完成此函数的线程将全局数据 `sum` 赋值为从 0 到传递给 `Summation()` 的参数的和。

在 Win32 API 中，线程的创建还使用了 `CreateThread()` 函数（正如在 Pthread 中那样）将一组线程的属性传递给此函数。这些属性包括安全信息、栈大小、一个用以表明挂起状态的线程是否开始的标志。这个程序中采用了这些属性的默认值（没有将线程初始化为挂起状态，而是使其具有被 CPU 调度的资格）。一旦创建了累加和线程，在输出累加值之前，父线程必须等待累加和线程完成，因为该值是累加和赋予的。回顾 Pthread 程序（见图 4.6），其中采用 `pthread_join()` 语句实现父线程等待累加和线程。在 Win32 中采用同等功能的函数 `WaitForSingleObject()`，从而使创建者线程阻塞，直至累加和线程退出（第 6 章将详细介绍同步对象）。

4.3.3 Java 线程

线程是 Java 程序中程序执行的基本模型，Java 语言和它的 API 为创建和管理线程提供了丰富的特征集。所有 Java 程序至少由一个控制线程组成——即使一个只有 `main()` 函数的简单 Java 程序也是在 JVM 中作为一个线程运行的。

在 Java 程序中有两种创建线程的技术。一种方法是创建一个新的类，它从 `Thread` 类派生，并重载它的 `run()` 函数。另外一种更常使用的方法是定义一个实现 `Runnable` 接口的类。`Runnable` 接口定义如下：

```
public interface Runnable
{
    public abstract void run();
}
```

当一个类执行 `Runnable` 时，它必须定义 `run()` 函数。而实现 `run()` 函数的代码被作为一个独立的线程执行。

图 4.8 是计算非负整数累加和的多线程例子的 Java 版。`Summation` 类实现了 `Runnable` 接口。通过创建一个 `Thread` 类的对象实例和传递 `Runnable` 对象的结构来创建线程。

创建 `Thread` 对象并不会创建一个新的线程，实际上是用 `start()` 函数来创建新线程。为新的对象调用 `start()` 函数需要做两件事：

① 在 JVM 中分配内存并初始化新的线程。

② 调用 `run()` 函数，使线程适合在 JVM 中运行（注意，从不直接调用 `run()` 函数，而是调用 `start()` 函数，然后它再调用 `run()` 函数）。

```
class Sum
{
    private int sum;
    public int getSum(){
        return sum;
    }
    public void setSum(int value){
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue){
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run(){
        int sum = 0;
        for(int i = 0; i <= upper; i++){
            sum += i;
            sumValue.setValue(sum);
        }
    }
}

public class Driver
{
    public static void main(String[] args){
        if(args.length > 0){
            if(Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + "must be >= 0.");
            else{
                // create the object to be shared
                Sum sum = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sum));
                thrd.start();
                try{
                    thrd.join();
                    System.out.println
                        ("The sum of " + upper + " is " + sum.getSum());
                } catch (InterruptedException ie){}
            }
        }
        else
            System.err.println("Usage: Summation <integer value>");
    }
}
```

图 4.8 非负整数累加的 Java 程序

当累加和程序运行时，通过 JVM 创建两个线程。第一个是父线程，它在 `main()` 函数中开始执行。第二个线程在调用 `Thread` 对象的 `start()` 函数时创建，这个子线程在 `Summation` 类的 `run()` 函数中开始执行。在输出累加和的值后，当此线程从 `run()` 函数中退出后线程终止。

在 Win32 和 Pthread 中线程间共享数据很方便，因为共享数据被简单地声明为全局数据。作为一个纯面向对象语言，Java 没有这样的全局数据的概念。在 Java 程序中如果两个或更多的线程需要共享数据，通过向相应的线程传递对共享对象的引用来实现。在图 4.8 所示的 Java 程序中，`main` 线程和累加和线程共享 `Sum` 类的对象实例，通过 `getSum()` 和 `setSum()` 函数引用共享对象（读者可能好奇为什么不使用 `java.lang.Integer` 对象，而是设计一个新的 `Sum` 类。这是因为 `java.lang.Integer` 类是不可变的——即一旦被赋予值，就不可改变）。

回忆一下 Pthread 和 Win32 库中的父线程，它们在继续之前，分别使用 `pthread_join` 或 `WaitForSingleObject()` 等待累加和线程结束。Java 中的 `join()` 函数提供了类似的功能。注意，`join()` 可能扔掉中断异常，这里选择忽略。

JVM 和宿主操作系统

JVM 一般在操作系统之上实现（参见图 2.17）。这种方式允许 JVM 隐藏基本的操作系统实现细节，并提供一种一致的、抽象的环境以允许 Java 程序能在任何支持 JVM 的平台上运行。JVM 的规范并没有指明 Java 线程如何被映射到底层的操作系统，而是让特定的 JVM 实现来决定。例如，Windows XP 操作系统采用一对一模式，因此每一个运行在这样系统上的 JVM 的 Java 线程映射到内核线程。在使用多对多模式的操作系统上（如 Tru64 UNIX），根据多对多模型来映射 Java 线程。Solaris 系统刚开始时采用多对一模型（如前所述的绿色线程库）来实现 JVM，后来采用了多对多模型。从 Solaris 9 开始，采用多对多模型来映射 Java 线程。此外，在 Java 线程库和宿主操作系统线程库之间存在联系。例如，Windows 系列操作系统的 JVM 实现可以在创建 Java 线程时使用 Win32 API，Linux 和 Solaris 系统则可以采用 Pthread API。

4.4 多线程问题

本节将讨论与多线程程序有关的一些问题。

4.4.1 系统调用 `fork()` 和 `exec()`

第 3 章讨论了系统调用 `fork()` 如何用于创建独立的、复制的进程。在多线程程序中，系统调用 `fork()` 和 `exec()` 的语义有所改变。

如果程序中的一个线程调用 `fork()`，那么新进程会复制所有线程，还是新进程只有单

个线程？有的 UNIX 系统有两种形式的 `fork()`，一种复制所有线程，另一种只复制调用了系统调用 `fork()` 的线程。

系统调用 `exec()` 的工作方式与第 3 章所述的方式通常相同，即如果一个线程调用了系统调用 `exec()`，那么 `exec()` 参数所指定的程序会替换整个进程，包括所有线程。

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

4.4.2 取消

线程取消 (thread cancellation) 是在线程完成之前来终止线程的任务。例如，如果多个线程并发执行来搜索数据库，并且一个线程已经得到了结果，那么其他线程就可被取消。另一种可能发生的情况是用户单击网页浏览器上的按钮以停止进一步装入网页。通常由几个线程装入网页——每个图像都是在一个独立线程中被装入的。当用户单击浏览器的停止按钮时，所有装入网页的线程就被取消了。

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

- ① **异步取消 (asynchronous cancellation)**：一个线程立即终止目标线程。
- ② **延迟取消 (deferred cancellation)**：目标线程不断地检查它是否应终止，这允许目标线程有机会以有序方式来终止自己。

如果资源已分配给要取消的线程或要取消的线程正在更新与其他线程所共享的数据，那么取消就会有困难。对于异步取消尤其麻烦。操作系统回收取消线程的系统资源，但是通常并不回收所有资源。因此，异步取消线程并不会使所需的系统资源空闲。相反采用延迟取消时，一个线程指示目标线程要被取消，不过，只有当目标线程检查一个标志以确定它是否应该取消时才会发生取消。这允许一个线程检查它是否是在安全的点被取消，Pthread 称这些点为**取消点 (cancellation point)**。

4.4.3 信号处理

信号在 UNIX 中用来通知进程某个特定事件已发生了。根据需要通知信号的来源和事件的理由，信号可以同步或异步接收。不管信号是同步或异步的，所有信号具有同样模式：

- ① 信号是由特定事件的发生所产生的。
- ② 产生的信号要发送到进程。
- ③ 一旦发送，信号必须加以处理。

同步信号的例子包括非法访问内存或被 0 所除。在这种情况下，如果运行程序执行这些动作，那么就产生信号。同步信号发送到执行操作而产生信号的同一进程（这就是为什

么被认为是同步的原因)。

当一个信号由运行进程之外的事件产生,那么进程就异步接收这一信号。这种信号的例子包括使用特殊键(按如 **Ctrl+C** 键)或定时器到期。通常,异步信号被发送到另一个进程。

每个信号可能由两种可能的处理程序中的一种来处理:

- ① 默认信号处理程序。
- ② 用户定义的信号处理程序。

每个信号都有一个**默认信号处理程序**(default signal handler),当处理信号时是在内核中运行的。这种默认动作可以用**用户定义的信号处理程序**来改写。信号可按不同的方式处理。有的信号可以简单地忽略(如改变窗口大小),其他的(如非法内存访问)可能要通过终止程序来处理。

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

通常有如下选择:

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

发送信号的方法依赖于产生信号的类型。例如,同步信号需要发送到产生这一信号的线程,而不是进程的其他线程。不过,对于异步信号,情况就不是那么清楚了。有的异步信号如终止进程的信号(例如按 **Ctrl+C** 键)应该发送到所有线程。

大多数多线程版 UNIX 允许线程描述它会接收什么信号和拒绝什么信号。因此,有时一个异步信号只能发送给那些不拒绝它的线程。不过,因为信号只能处理一次,所以信号通常发送到不拒绝它的第一个线程。标准的发送信号的 UNIX 函数是 `kill(pid_t pid, int signal)`,在这里指定了信号的发送进程(pid)。不过 POSIX Pthread 还提供了 `pthread_kill(pthread_t tid, int signal)`函数,此函数允许信号被传送到一个指定的线程(tid)。

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

4.4.4 线程池

4.1 小节描述了对 Web 服务器进行多线程编程的情况。在这种情况下,每当服务器收

到请求，它就创建一个独立线程以处理请求。虽然创建一个独立线程显然要比创建一个独立进程要好，但是多线程服务器也有一些潜在问题。第一个是关于在处理请求之前用以创建线程的时间，以及线程在完成工作之后就要被丢弃这一事实。第二个问题更为麻烦：如果允许所有并发请求都通过新线程来处理，那么将没法限制在系统中并发执行的线程的数量。无限制的线程会耗尽系统资源，如 CPU 时间和内存。解决这个问题的一种方法是使用线程池（thread pool）。

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

线程池具有如下主要优点：

① 通常用现有线程处理请求要比等待创建新的线程要快。

② 线程池限制了在任何时候可用线程的数量。这对那些不能支持大量并发线程的系统非常重要。

线程池中的线程数量由系统 CPU 的数量、物理内存的大小和并发客户请求的期望值等因素决定。比较高级的线程池能动态调整线程的数量，以适应具体情况。这类结构的优点是在系统负荷低时减低内存消耗。

Win32 API 提供了几个与线程池相关的函数。使用线程池 API 类似于用 ThreadCreate() 函数创建新线程，如 4.3.2 小节所介绍的。在此，定义了一个作为独立线程运行的函数，该函数可能如下：

```
DWORD WINAPI PoolFunction(AVOID Param){
    /**
     * this function runs as a separate thread.
     */
}
```

一个指向 PoolFunction() 函数的指针被传递给线程池 API 中的一个函数，池中的一个线程执行这个函数。QueueUserWorkItem() 函数是线程池 API 的成员之一，它被传递了三个参数：

- LPTHREAD_START_ROUTINE Funtion: 指向作为独立线程运行的函数的指针。
- PVOID Param: 传递给 Funtion 的参数。
- ULONG Flags: 显示线程池如何创建和管理线程的执行的标志。

一个调用的例子如下：

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

这使线程池中的线程代表程序员来调用 PoolFunction() 函数。在这个例子中，没有向 PoolFunction() 传递参数。这是因为将 0 指定为一个标志，并提供了没有特别说明的线程池来创建线程。

Win32 线程池 API 的另一成员包括在一个周期性区间或一个异步 I/O 请求结束时调用函数的工具。Java 1.5 中的 `java.util.concurrent` 包提供了一个线程池工具。

4.4.5 线程特定数据

同属一个进程的线程共享进程数据。事实上，这种数据共享提供了多线程编程的一种优势。不过，在有些情况下每个线程可能需要一定数据的自己的副本。这种数据称为线程特定数据（thread-specific data）。例如，对于事务处理系统，可能需要通过独立线程以处理请求。而且，每个事务都有一个唯一标识符。为了让每个线程与其唯一标识符相关联，可能使用线程特定数据。绝大多数线程库，包括 Win32 和 Pthread，都提供了对线程特定数据的一定支持，Java 也提供这种支持。

4.4.6 调度程序激活

多线程编程的最后一个问题是内核与线程库之间的通信问题，这就需要用到 4.2.3 小节讨论的多对多模型和二级模型。这种协调允许动态调整内核线程的数量以保证其最好的性能。

许多实现多对多模型或二级模型的系统在用户和内核线程之间设置一种中间数据结构。这种数据结构（通常是轻量级进程（LWP）），如图 4.9 所示。对于用户线程库，LWP 表现为一种应用程序可以调度用户线程来运行的虚拟处理器。每个 LWP 与内核线程相连，该内核线程被操作系统调度到物理处理器上运行。如果内核线程阻塞（如在等待一个 I/O 操作结束），LWP 也阻塞。在这个关系链的顶端，与 LWP 相连的用户线程也阻塞。

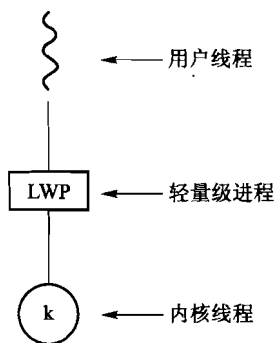


图 4.9 轻量级进程（LWP）

为了高效地运行，应用程序可能需要一定数量的 LWP。考虑一个 CPU 约束的运行在单处理器上的应用程序。此时，一次只能运行一个线程，所以只要一个 LWP 就够了，但一个 I/O 请求密集的应用程序可能需要多个 LWP 来执行。通常，每个并发阻塞系统调用需要一个 LWP。例如，设想一下有 5 个不同文件读请求可能同时发生的情况，此时就需要 5

个 LWP，因为每个都需要等待内核 I/O 的完成。如果进程只有 4 个 LWP，那么第 5 个请求必须等待其中一个 LWP 从内核返回。

一种解决用户线程库与内核间通信的方法被称为**调度器激活**（scheduler activation）。它按如下方式工作：内核提供一组虚拟处理器（LWP）给应用程序，应用程序可调度用户线程到一个可用的虚拟处理器上。进一步说，内核必须告知与应用程序有关的特定事件。这个过程被称为 **upcall**。**upcall** 由具有 **upcall 处理句柄** 的线程库处理，**upcall 处理句柄** 必须在虚拟处理器上运行。当一个应用线程将要阻塞时，事件引发一个 **upcall**。在这个例子中，内核向应用程序发出一个 **upcall**，通知它线程阻塞并标识特殊的线程。然后内核分配一个新的虚拟处理器给应用程序，应用程序在这个新的虚拟处理器上运行 **upcall 处理程序**，它保存阻塞线程状态和放弃阻塞线程运行的虚拟处理器。然后 **upcall** 调度另一个适合在新的虚拟处理器上运行的线程，当阻塞线程事件等待发生时，内核向线程库发出另一个 **upcall**，来通知它先前阻塞的线程现在可以运行了。此事件的 **upcall 处理程序** 也需要一个虚拟处理器，内核可能分配一个新的虚拟处理器或先占一个用户线程并在其虚拟处理器上运行 **upcall 处理程序**。在使非阻塞线程可以运行后，应用程序调度符合条件的线程来在一个适当的虚拟处理器上运行。

4.5 操作系统实例

本节研究在 Windows XP 和 Linux 中如何实现线程。

4.5.1 Windows XP 线程

Windows XP 实现了 Win32 API。Win32 API 是 Microsoft 操作系统家族的主要 API（如 Windows 95/98/NT/2000 和 Windows XP）。事实上，本节所讨论的也适用于这整个操作系统家族。

一个 Windows XP 应用程序以独立进程方式运行，每个进程可包括一个或多个线程。

4.3.2 小节讨论了创建线程的 Win32 API。Windows XP 使用了如 4.2.2 小节所述的一对一映射，其中每个用户线程映射到相关的内核线程。不过，Windows XP 也提供了对 **fiber** 库的支持，该库提供了多对多模型（参见 4.2.3 小节）的功能。通过使用线程库，同属一个进程的每个线程都能访问进程的地址空间。

一个线程通常包括如下部分：

- 一个线程 ID，以唯一标识线程。
- 一组寄存器集合，以表示处理器状态。
- 一个用户栈，以供线程在用户模式下运行；一个内核堆栈，以供线程在内核模式下运行。
- 一个私有存储区域，为各种运行时库和动态链接库（DLL）所用。

寄存器集合、栈和私有存储区域通常称为线程的上下文。线程的主要数据结构包括：

- ETHREAD：执行线程块。
- KTHREAD：内核线程块。
- TEB：线程执行环境块。

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

KTHREAD 包括线程的调度和同步信息。另外，KTHREAD 也包括内核栈（当线程在内核模式下运行时使用）和 TEB 的指针。

ETHREAD 和 KTHREAD 完全处于内核空间，这意味着只有内核可以访问它们。TEB 是用户空间的数据结构，可供线程在用户模式下运行时访问。TEB 除了包括许多其他域外，还包括用户模式栈和用于线程特定数据的数组（Windows 称之为线程本地存储）。Windows XP 线程的结构如图 4.10 所示。

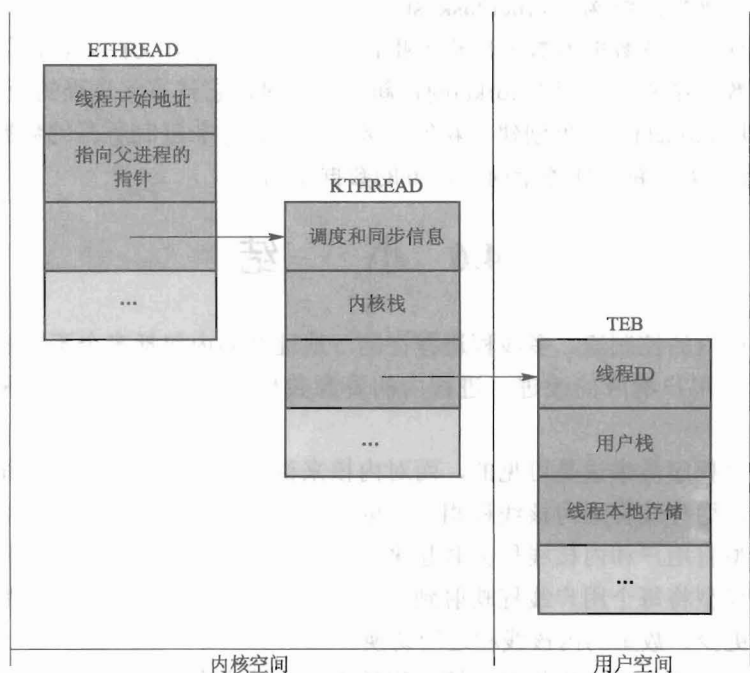


图 4.10 Windows XP 线程的数据结构

4.5.2 Linux 线程

正如第 3 章所讲，Linux 提供了具有传统进程复制功能的系统调用 `fork()`，还提供了使用系统调用 `clone()` 创建线程的功能，Linux 并不区分进程和线程。事实上，Linux 在讨论程

序控制流时，通常称之为任务而不是进程或线程。`clone()`被调用时，它被传递一组标志，以决定父任务与子任务之间发生多少共享。其中一些标志列举如下：

| 标 志 | 含 义 | 标 志 | 含 义 |
|----------|-----------|---------------|----------|
| CLONE_FS | 共享文件系统信息 | CLONE_SIGHAND | 共享信号处理程序 |
| CLONE_VM | 共享共同的内存空间 | CLONE_FILES | 共享打开文件集 |

例如，如果将 `CLONE_FS`、`CLONE_VM`、`CLONE_SIGHAND` 和 `CLONE_FILES` 标志传递给 `clone()`，父任务和子任务将共享相同的文件系统信息（如当前工作目录）、相同的内存空间、相同的信号处理程序和相同的打开文件集。使用这种 `clone()` 相当于本章介绍的创建线程，因为父任务和其子任务共享大多数资源。不过，如果当调用 `clone()` 时没有设置一个标志，则不会发生共享，导致类似于系统调用 `fork()` 提供的功能。

共享级别的变化是可能的，这源于 Linux 内核中任务表达的方式。系统中每个任务都有一个唯一的内核数据结构（`struct task_struct`），这个数据结构并不保存任务本身的数据，而是指向其他存储这些数据的数据结构的指针——如表示打开文件列表、信号处理信息和虚拟内存等的数据结构。当调用 `fork()` 创建新的任务时，它具有父进程的所有数据的副本。当调用系统调用 `clone()` 时，也创建了新的任务。不过，并非复制所有的数据结构，根据传递给 `fork()` 的标志集，新的任务指向父任务的数据结构。

4.6 小 结

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

用户线程对程序员来说是可见的，而对内核来说却是未知的。操作系统支持和管理内核线程。通常，用户线程跟内核线程相比，创建和管理要更快，因为它不需要内核干预。有三种不同模型将用户和内核线程关联起来：多对一模型将许多用户线程映射到一个内核线程；一对一模型将每个用户线程映射到一个相应的内核线程；多对多模型将多个用户线程在同样（或更少）数量的内核线程之间切换。

绝大多数现代操作系统提供对内核线程的支持，其中有 Windows 98、Windows NT、Windows 2000 和 Windows XP，还包括 Solaris 和 Linux。

线程库为应用程序员提供了创建和管理线程的 API，通常有三种主要的线程库：POSIX Pthread API、Windows 系统的 Win32 线程以及 Java 线程。

多线程程序为程序员带来了许多挑战，包括系统调用 `fork()` 和 `exec()` 的语义。其他事项包括线程取消、信号处理和特定线程数据。

习 题

- 4.1 举两个多线程程序设计的例子，其中多线程的性能比单线程的性能差。
- 4.2 描述线程库进行用户级线程上下文切换的过程所采取的措施。
- 4.3 在什么环境中，采用多内核线程的多线程方法比单处理器系统的单线程提供更好的性能？
- 4.4 在多线程进程中，下列哪些程序状态组成被共享？

- a. 寄存器值
- b. 堆内存
- c. 全局变量
- d. 栈内存

4.5 使用多用户线程的多线程解决方案，在多处理器系统中可以比在单处理器系统中获得更好的性能吗？

4.6 如 4.5.2 小节所介绍，Linux 并不区分进程和线程，而是将两者同样对待，将一个任务视为进程或线程，这取决于传递给 clone() 系统调用的标志集。然而，许多操作系统，如 Windows XP 和 Solaris，对待进程和线程是不一样的。通常，这类系统使用标记，其中进程的数据结构中包含指向属于进程的不同线程。试在内核中比较这两种对进程和线程建模的方法。

4.7 如图 4.11 所示的程序使用了 Pthread API，程序中 LINE C 和 LINE P 将会输出什么？

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

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    int pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork();
    if (pid == 0) { /* child process */
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = %d", value); /*LINE C */
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d", value); /*LINE P */
    }
}

void *runner(void *param){
    value = 5;
    pthread_exit(0);
}
```

图 4.11 习题 4.7 的 C 程序

4.8 考虑多处理器系统和采用多对多线程模式编写的多线程程序，使程序中用户级线程数比系统中处理器数多。讨论下列情形的性能影响：

- a. 分配给程序的内核线程数比处理器数少。
- b. 分配给程序的内核线程数与处理器数相等。
- c. 分配给程序的内核线程数比处理器数多，但少于用户线程数。

4.9 编写一个多线程的 Java、Pthread 或 Win32 程序来输出素数。程序应这样工作：用户运行程序时在命令行输入一个数字，然后创建一个独立线程来输出小于或等于用户输入数的所有素数。

4.10 修改第 3 章中基于 Socket 的日期服务器（图 3.19），使得服务器在不同的线程中服务每一个客户端请求。

4.11 Fibonacci 序列为 0,1,1,2,3,5,8,...，通常，这可表达为：

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

使用 Java、Pthread 或 Win32 线程库编写一个多线程程序来生成 Fibonacci 序列。程序应这样工作：用户运行程序时在命令行输入要产生 Fibonacci 序列的数，然后程序创建一个新的线程来产生 Fibonacci 数，把这个序列放到线程共享的数据中（数组可能是一种最方便的数据结构）。当线程执行完成后，父线程将输出由子线程产生的序列。由于在子线程结束前，父线程不能开始输出 Fibonacci 序列，因此父线程必须等待子线程的结束，这可采用 4.3 节所述的技术。

4.12 第 3 章的习题 3.9 使用 Java 线程 API 设计一个 echo 服务器，但这个服务器是单线程的，即服务器不能对当前并发的 echo 客户机进行响应，除非当前的客户机退出。修改习题 3.9 的解答，以使 echo 服务器在单独的线程中服务每个客户机。

项目：矩阵乘法

给定两个矩阵 A 和 B ，其中 A 是具有 M 行、 K 列的矩阵， B 为 K 行、 N 列的矩阵， A 和 B 的矩阵积为矩阵 C ， C 为 M 行、 N 列。矩阵 C 中第 i 行、第 j 列的元素 $C_{i,j}$ 就是矩阵 A 第 i 行每个元素和矩阵 B 第 j 列每个元素乘积的和，即

$$C_{i,j} = \sum_{n=1}^K A_{i,n} \times B_{n,j}$$

例如，如果 A 是 3×2 的矩阵， B 是 2×3 的矩阵，则元素 $C_{3,1}$ 将是 $A_{3,1} \times B_{1,1}$ 、 $A_{3,2} \times B_{2,1}$ 的和。

对于该项目，计算每个 $C_{i,j}$ 是一个独立的工作线程，因此它将会涉及生成 $M \times N$ 个工作线程。主线程（或称为父线程）将初始化矩阵 A 和 B ，并分配足够的内存给矩阵 C ，它将容纳矩阵 A 和 B 的积。这些矩阵将声明为全局数据，以使每个工作线程都能访问矩阵 A 、 B 和 C 。

矩阵 A 和 B 可以静态初始化为：

```
#define M 3
#define K 2
```

```
#define N 3

int A [M] [K] = { {1,4}, {2,5}, {3,6} };
int B [K] [N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

或者，它们也可以从一个文件中读入。

1. 向每个线程传递参数

父线程将生成 $M \times N$ 个工作线程，给每个线程传递行 i 和列 j 的值，工作线程利用行和列的值来计算矩阵积。这需要向每个线程传递两个参数。最简单的方法是利用 Pthread 和 Win32 中的 struct 生成一个数据结构，该数据结构的成员为 i 和 j ，即：

```
/*structure for passing data to threads */
struct v
{
    int i; /* row */
    int j; /* column */
};
```

Pthread 和 Win32 程序都将采用如下算法生成工作线程：

```
/* We have to creat M*N worker threads*/
for (i = 0; i < M, i++)
    for(j = 0; j < N, j++ ) {
        struct v *data = (struct v *) malloc(sizeof(struct v));
        data -> i = i;
        data -> j = j;
        /*Now create the thread passing it data as a parameter */
    }
```

数据指针将被传递给 pthread_create()(Pthreads)函数或 CreateThread()(Win32)函数，然后又将它作为参数传递给作为独立线程运行的函数。

Java 线程之间共享数据与 Pthread 或 Win32 是不同的。一种方法是主线程创建和初始化矩阵 A 、 B 和 C ，然后该主线程将创建工作线程，传递三个矩阵（与第 i 行和第 j 列一起）给每个工作线程的构造函数。工作线程大致如下：

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int [] [] A;
    private int [] [] B;
    private int [] [] C;
```



```

public WorkThread( int row, int col, int [] [] A, int [] [] B, int
[] [] C) {
    this.row = row;
    this.col = col;
    this.A = A;
    this.B = B;
    this.C = C;
}

public void run(){
    /*Calculate the matrix product in C[row][col]*/
}
}

```

2. 等待线程结束

一旦所有的工作线程结束，主线程将输出包含在矩阵 C 中的积。这需要主线程等待所有的工作线程完成其工作，然后才能输出矩阵积的值。有几种不同的方法用来使一个线程等待其他线程的结束。4.3 小节描述了在如何采用 Java、Pthread 或 Win32 线程库等待子线程结束。Win32 提供了 WaitForSingleObject() 函数，Pthread 和 Java 则分别采用 pthread_join() 和 join() 函数。然而，在这些程序例子中，父线程等待单个子线程的结束，完成这个练习需要等待多个线程。

4.3.2 小节介绍了 WaitForSingleObject() 函数，它被用来等待单个线程的结束。但 Win32 API 同时还提供了 WaitForMultipleObjects() 函数，它被用来等待多个线程的结束。WaitForMultipleObjects() 函数被传递了 4 个参数：

- 所要等待的对象数。
- 指向对象数组的指针。
- 表明是否所有对象都以 signal 通知的标志。
- 超时时间（或 INFINITE）。

例如，如果 THandles 是一个大小为 N 的线程 HANDLE 对象的数组，使用下列语句，父线程可以等待所有子线程结束：

```
WaitForMultipleObjects(N, Thandles, TRUE, INFINITE);
```

一个简单的策略是，使用 Pthread 的 pthread_join() 或 Java 的 join() 等待多个线程结束的算法被封装到一个简单的 for 循环的 join 操作中。例如，可以使用图 4.12 所示的 Pthread 代码加入 10 个线程，相应的 Java 线程代码如图 4.13 所示。

| | |
|--|---|
| <pre> #define NUM_THREADS 10 /*an array of threads to be joined upon*/ pthread_t workers[NUM_THREADS]; for(int i = 0; i < NUM_THREADS, i++) pthread_join(workers[i], NULL); </pre> | <pre> final static int NUM_THREADS = 10; /* an array of threads to be joined upon*/ Thread[] workers = new Thread[NUM_THREADS]; for(int i = 0; i < NUM_THREADS; i++){ try{ workers[i].join(); }catch(InterruptedException ie){} } </pre> |
|--|---|

图 4.12 等待 10 个线程的 Pthread 代码

图 4.13 等待 10 个线程的 Java 代码

文献注记

Anderson 等[1989]论述了线程的性能, 后来 Anderson 等[1991]对内核支持的用户级线程性能进行了评估。Bershad 等[1990]描述了结合使用线程与 RPC。Engleschall[2000]论述了支持用户级线程的一个技巧。Ling 等[2000]里有最佳线程池大小的分析。Anderson 等[1991] 和 Williams[2002]讨论了 NetBSD 系统中的调度器激活问题。Marsh 等[1991]、Govindan 和 Anderson[1991]、Draves 等[1991]、Black[1990]讨论了另一个用户级线程库和内核之间的协作机制。Zabatta 和 Young [1998]比较了在对称多处理器上的 Windows NT 和 Solaris 线程。Pinilla 和 Gill[2003]在 Linux、Windows 和 Solaris 系统中的 Java 线程的性能。

Vahalia[1996]论述了许多版本的 UNIX 线程问题。Mauro 和 McDougall[2001]描述了 Solaris 内核中的线程的新发展。Solomon 和 Russionvich[2000]描述了 Windows 2000 中的线程, Bovet、Cesati[2002] 和 Love[2004]解释了 Linux 如何处理线程。

Lewis 和 Berg[1998]、Butenhof[1997]中给出了 Pthread 的编程信息。Solaris 中的线程编程信息可以参看 Sun Microsystems[1995]。Oaks 和 Wong[1999]、Lewis 和 Berg[2000]和 Holub[2000]讨论了 Java 中的多线程编程问题。Beveridge 和 Wiener[1997]、Cohen 和 Woodring[1997]介绍了使用 Win32 进行多线程编程。

第5章 CPU 调度

CPU 调度是多道程序操作系统的基础。通过在进程之间切换 CPU，操作系统可以提高计算机的吞吐率。本章将介绍基本调度概念和多个不同的 CPU 调度算法，也将研究为特定系统选择算法的问题。

在第 4 章为进程模型引入了线程。对于支持它们的操作系统，是内核级的线程被操作系统调度，而不是进程。不过，术语线程调度或进程调度常常被交替使用。本章在讨论普通调度概念时使用进程调度，特别指定为线程概念时使用线程调度。

本章目标

- 介绍 CPU 调度，它是多道程序操作系统的基础。
- 描述各种 CPU 调度算法。
- 讨论为特定系统选择 CPU 调度算法的评估标准。

5.1 基本概念

对于单处理器系统，每次只允许一个进程运行；任何其他进程必须等待，直到 CPU 空闲能被调度为止。多道程序的目标是在任何时候都有某些进程在运行，以使 CPU 使用率最大化。多道程序的思想较为简单。进程执行直到它必须等待，通常等待某些 I/O 请求的完成。对于一个简单计算机系统，CPU 就会因此空闲，所有这些等待时间就浪费了，而没有完成任何有用的工作。采用多道程序设计，系统试图有效地使用这一时间，多个进程可同时处于内存中。当一个进程必须等待时，操作系统会从该进程拿走 CPU 的使用权，而将 CPU 交给其他进程，如此继续。在该进程必须等待的时间内，另一个进程就可以拿走 CPU 的使用权。

这种调度是操作系统的基本功能。几乎所有的计算机资源在使用前都要调度。当然，CPU 是最重要的计算机资源之一。因此，CPU 调度对于操作系统设计来说很重要。

5.1.1 CPU-I/O 区间周期

CPU 的成功调度依赖于进程的如下属性：进程执行由 CPU 执行和 I/O 等待周期组成。进程在这两个状态之间切换。进程执行从 CPU 区间（CPU burst）开始，在这之后是 I/O

区间 (I/O burst)，接着是另一个 CPU 区间，然后是另一个 I/O 区间，如此进行下去。最终，最后的 CPU 区间通过系统请求终止执行（见图 5.1）。

这些 CPU 区间的长度已被大量地测试过。虽然它们随着进程和计算机的不同变化很大，但是呈现出类似于图 5.2 所示的频率曲线。该曲线通常为指数或超指数形式，具有大量短 CPU 区间和少量长 CPU 区间。I/O 约束程序通常具有很多短 CPU 区间。CPU 约束程序可能有少量的长 CPU 区间。这种分布有助于选择合适的 CPU 调度算法。

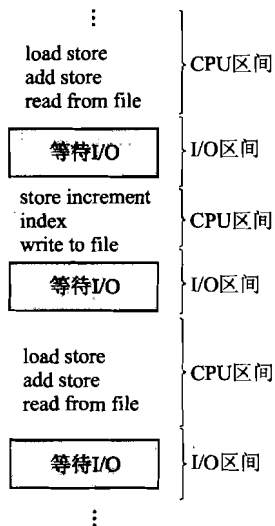


图 5.1 CPU 区间和 I/O 区间的交替序列

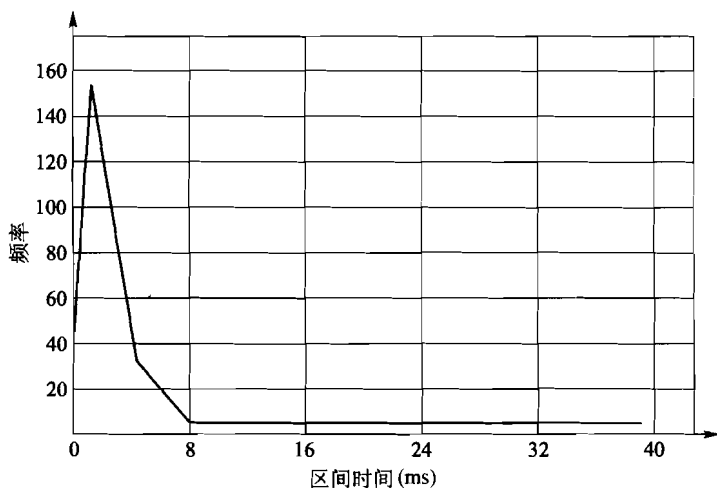


图 5.2 CPU 区间时间曲线图

5.1.2 CPU 调度程序

每当 CPU 空闲时，操作系统就必须从就绪队列中选择一个进程来执行。进程选择由短期调度程序 (short-term scheduler) 或 CPU 调度程序执行。调度程序从内存中选择一个能够执行的进程，并为之分配 CPU。

就绪队列不必是先进先出 (FIFO) 队列。正如研究各种调度算法时将看到的，就绪队列可实现为 FIFO 队列、优先队列、树或简单的无序链表。不过，从概念上来说，就绪队列内的所有进程都要排队以等待在 CPU 上运行。队列中的记录通常为进程控制块 (PCB)。

5.1.3 抢占调度

CPU 调度决策可在如下 4 种环境下发生：

- 当一个进程从运行状态切换到等待状态（例如，I/O 请求，或调用 wait 等待一个子

进程的终止)。

- 当一个进程从运行状态切换到就绪状态 (例如, 当出现中断时)。
- 当一个进程从等待状态切换到就绪状态 (例如, I/O 完成)。
- 当一个进程终止时。

对于第 1 和第 4 两种情况, 没有选择而只有调度。一个新进程 (如果就绪队列中已有一个进程存在) 必须被选择执行。不过, 对于第 2 和第 3 两种情况, 可以进行选择。

当调度只能发生在第 1 和第 4 两种情况下时, 称调度方案是**非抢占的** (nonpreemptive) 的或**协作的** (cooperative); 否则, 称调度方案是**抢占的** (preemptive)。采用非抢占调度, 一旦 CPU 分配给一个进程, 那么该进程会一直使用 CPU 直到进程终止或切换到等待状态。Windows 3.x 使用该种调度方法, Windows 95 引入了抢占调度, 所有之后的 Windows 操作系统版本都使用抢占调度。Macintosh 的 Mac OS X 操作系统使用抢占调度, 而之前的 Macintosh 操作系统依赖协作调度。协作调度在有的硬件平台上是唯一的方法, 因为它不要求抢占调度所需要的特别的硬件 (如定时器) 支持。

不幸的是, 抢占调度对访问共享数据是有代价的。考虑两个进程共享数据的情况。第一个进程正在更新数据时, 它被抢占以使第二个进程能够运行。第二个进程可能试图读取数据, 该数据现在处于不一致的状态。这种情况下需要一种新机制来协调对共享数据的访问。这一问题将在第 6 章中讨论。

抢占对于操作系统内核的设计也有影响。在处理系统调用时, 内核可能忙于进程活动。这些活动可能涉及要改变重要内核数据 (如 I/O 队列)。如果一个进程在进行这些修改时被抢占, 内核 (或设备驱动) 需要读取或修改同样的结构, 那么会有什么结果呢? 肯定会导致混乱。有的操作系统, 包括绝大多数 UNIX 系统, 通过在上下文切换之前等待系统调用完成或等待发生 I/O 阻塞来处理这一问题。不幸的是, 这种内核执行模式对实时计算和多进程的支持较差。这些问题及其解决方案将在 5.4 和 19.5 节中讨论。

因为根据定义中断能随时发生, 而且不能总是被内核所忽视, 所以受中断影响的代码段必须加以保护以避免同时访问。操作系统需要在任何时候都能接受中断, 否则输入会丢失或输出会被改写。为了这些代码段不被多个进程同时访问, 在进入时要禁止中断, 而在退出时要重新允许中断。注意到禁止中断代码段发生并不频繁, 而且常常只包括很少的指令, 这很重要。

5.1.4 分派程序

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

- 切换上下文。
- 切换到用户模式。

- 跳转到用户程序的合适位置，以重新启动程序。

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

5.2 调度准则

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

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

- **CPU 使用率**：需要使 CPU 尽可能忙。从概念上讲，CPU 使用率从 0%~100%。对于真实系统，它应从 40%（轻负荷系统）~90%（重负荷系统）。

- **吞吐量**：如果 CPU 忙于执行进程，那么就有工作在完成。一种测量工作量的方法称为**吞吐量**，它指一个时间单元内所完成进程的数量。对于长进程，吞吐量可能为每小时一个进程；对于短进程，吞吐量可能为每秒 10 个进程。

- **周转时间**：从一个特定进程的角度来看，一个重要准则是运行该进程需要多长时间。从进程提交到进程完成的时间段称为**周转时间**。周转时间为所有时间段之和，包括等待进入内存、在就绪队列中等待、在 CPU 上执行和 I/O 执行。

- **等待时间**：CPU 调度算法并不影响进程运行和执行 I/O 的时间；它只影响进程在就绪队列中等待所花的时间。**等待时间**为在就绪队列中等待所花费时间之和。

- **响应时间**：对于交互系统，周转时间并不是最佳准则。通常，进程能相当早就产生输出，并继续计算新结果同时输出以前的结果给用户。因此，另一时间是从提交请求到产生第一响应的的时间。这种时间称为**响应时间**，是开始响应所需要的时间，而不是输出响应所需要的时间。周转时间通常受输出设备速度的限制。

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

对于交互系统（如分时系统），有的分析人员建议最小化响应时间的方差要比最小化平均响应时间更为重要。具有合理的可预见的响应时间的系统，比对平均值来说更快但变化大的系统更为理想。不过，在 CPU 调度算法如何使方差最小化方面，得到的研究成果并不多。

下面在讨论各种 CPU 调度算法时，将会描述其操作。由于精确描述需要涉及许多进程，且每个进程有数百个 CPU 区间和 I/O 区间的序列。为了简化描述，在所举的例子中只考虑每个进程有一个 CPU 区间（以 ms 计）。所比较的量是平均等待时间。更为精确的评选机

制将在 5.7 节中讨论。

5.3 调度算法

CPU 调度处理是从就绪队列中选择进程并为之分配 CPU 的问题。有多种不同的 CPU 调度算法, 本节描述其中一些 CPU 调度算法。

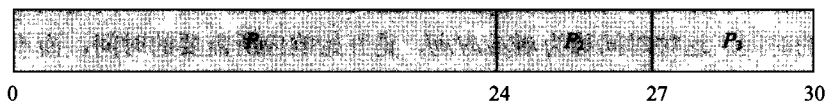
5.3.1 先到先服务调度

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

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

| 进程 | 区间时间 |
|-------|------|
| P_1 | 24 |
| P_2 | 3 |
| P_3 | 3 |

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



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



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

另外, 考虑 FCFS 调度在动态情况下的性能。假设有一个 CPU 约束进程和许多 I/O 约

束进程。随着进程在系统中运行，如下情况可能会发生：CPU 约束进程得到 CPU 并控制它。在这段时间内，所有其他进程会处理完它们的 I/O 并转移到就绪队列以等待 CPU。当这些进程在就绪队列里等待时，I/O 设备空闲。最终，CPU 约束进程完成其 CPU 区间并移动到 I/O 区间。所有 I/O 约束进程，由于只有很短的 CPU 区间，故很快执行完并移回到 I/O 队列。这时，CPU 空闲。之后，CPU 约束进程会移回到就绪队列并被分配到 CPU。再次，所有 I/O 进程会在就绪队列中等待 CPU 约束进程的完成。由于所有其他进程都等待一个大进程释放 CPU，这称为护航效果（convoy effect）。与让较短进程最先执行相比，这样会导致 CPU 和设备的使用率变得更低。

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

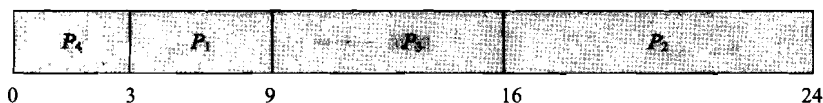
5.3.2 最短作业优先调度

另一种 CPU 调度方法是最短作业优先调度算法（shortest-job-first (SJF) scheduling algorithm）。这一算法将每个进程与其下一个 CPU 区间段相关联。当 CPU 为空闲时，它会赋给具有最短 CPU 区间的进程。如果两个进程具有同样长度，那么可以使用 FCFS 调度来处理。注意，一个更为适当的表示是最短下一个 CPU 区间的算法，这是因为调度检查进程的下一个 CPU 区间的长度，而不是其总长度。使用术语 SJF 是因为绝大多数教科书和人员称这种调度策略为 SJF。

作为一个例子，考虑如下一组进程，其 CPU 区间时间以 ms 计：

| 进程 | 区间时间 |
|-------|------|
| P_1 | 6 |
| P_2 | 8 |
| P_3 | 7 |
| P_4 | 3 |

采用 SJF 调度，就能根据如下 Gantt 图来调度这些进程：



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

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

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

虽然 SJF 算法最佳, 但是它不能在短期 CPU 调度层次上加以实现。因为没有办法知道下一个 CPU 区间的长度。一种方法是近似 SJF 调度。虽然不知道下一个 CPU 区间的长度, 但是可以预测它。认为下一个 CPU 区间的长度与以前的相似。因此, 通过计算下一个 CPU 区间长度的近似值, 能选择具有最短预测 CPU 区间的进程来运行。

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

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

公式定义了一个指数平均。 t_n 值包括最近信息, τ_n 存储了过去历史。参数 α 控制了最近和过去历史在预测中的相对加权。如果 $\alpha = 0$, 那么 $\tau_{n+1} = \tau_n$, 近来历史没有影响(当前情形为暂时的); 如果 $\alpha = 1$, 那么 $\tau_{n+1} = t_n$, 只有最近 CPU 区间才重要(历史的被认为是陈旧的、无关的)。更为常见的是, $\alpha = 1/2$, 这样最近历史和过去历史同样重要。初始值 τ_0 可作为常量或作为系统的总体平均值。图 5.3 说明了一个指数平均值, 其中 $\alpha = 1/2, \tau_0 = 10$ 。

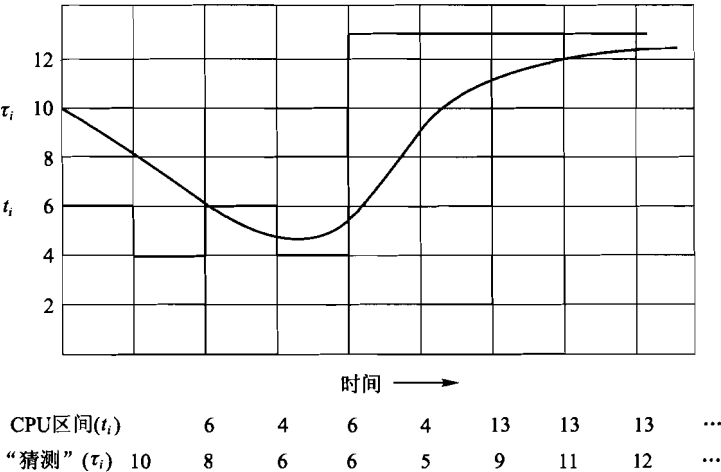


图 5.3 下一个 CPU 区间长度的预测

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

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

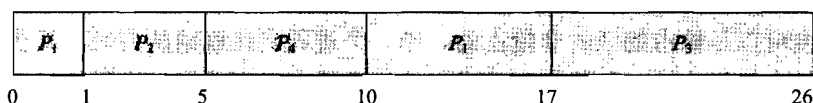
由于 α 和 $(1-\alpha)$ 小于或等于 1，所以后面项的权比前面项的权要小。

SJF 算法可能是抢占的或非抢占的。当一个新进程到达就绪队列而以前进程正在执行时，就需要选择。与当前运行的进程相比，新进程可能有一个更短的 CPU 区间。抢占 SJF 算法可抢占当前运行的进程，而非抢占 SJF 算法会允许当前运行的进程先完成其 CPU 区间。抢占 SJF 调度有时称为**最短剩余时间优先调度**（shortest-remaining-time-first scheduling）。

例如，考虑如下四个进程，其 CPU 区间时间以 ms 计：

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

如果进程按所给定的时间到达就绪队列，且需要所给定的区间时间，那么所产生的抢占 SJF 调度如下面的 Gantt 图所示。



进程 P_1 在时间 0 时开始，因为这时只有进程 P_1 。进程 P_2 在时间 1 时到达。进程 P_1 剩余时间（7 ms）大于进程 P_2 所需要的时间（4 ms），因此进程 P_1 被抢占，而进程 P_2 被调度。对于这个例子，平均等待时间为 $((10-1) + (1-1) + (17-2) + (5-3)) / 4 = 26/4 = 6.5$ ms。如果使用非抢占 SJF 调度，那么平均等待时间为 7.75 ms。

5.3.3 优先级调度

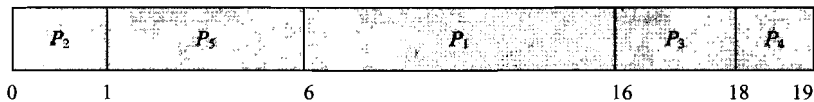
SJF 算法可作为通用**优先级调度算法**（priority scheduling algorithm）的一个特例。每个进程都有一个优先级与其关联，具有最高优先级的进程会分配到 CPU。具有相同优先级的进程按 FCFS 顺序调度。SJF 算法属于简单优先级算法，其优先级（ p ）为下一个（预测的）CPU 区间的倒数。CPU 区间越大，则优先级越小，反之亦然。

注意，按照高优先级和低优先级来讨论调度。优先级通常为固定区间的数字，如 0~7，或 0~4 095。不过，对于 0 是最高还是最低的优先级，并没有定论。有的系统用小数字表示低优先级，有的系统用小数字表示高优先级。这一差异可能导致混淆。在本书中，用小数字表示高优先级。

例如，考虑下面一组进程，它们在时间 0 时按顺序 P_1, P_2, \dots, P_5 到达，其 CPU 区间时间按 ms 计：

| 进程 | 区间时间 | 优先级 |
|-------|------|-----|
| P_1 | 10 | 3 |
| P_2 | 1 | 1 |
| P_3 | 2 | 4 |
| P_4 | 1 | 5 |
| P_5 | 5 | 2 |

采用优先级调度，会按照下面的 Gantt 图来调度这些进程。



平均等待时间为 8.2 ms。

优先级可通过内部或外部方式来定义。内部定义优先级使用一些测量数据以计算进程优先级。例如，时间极限、内存要求、打开文件的数量和平均 I/O 区间与平均 CPU 区间之比都可以用于计算优先级。外部优先级是通过操作系统之外的准则来定义的，如进程重要性、用于支付使用计算机的费用类型和数量、赞助工作的单位、其他（通常为政治）因素。

优先调度可以是抢占的或者非抢占的。当一个进程到达就绪队列时，其优先级与当前运行进程的优先级相比较。如果新到达进程的优先级高于当前运行进程的优先级，那么抢占优先级调度算法会抢占 CPU。而非抢占优先级调度算法只是将新进程加到就绪队列的头部。

优先级调度算法的一个主要问题是**无穷阻塞**（indefinite blocking）或**饥饿**（starvation）。可以运行但缺乏 CPU 的进程可认为是阻塞的，它在等待 CPU。优先级调度算法会使某个低优先级进程无穷等待 CPU。通常，会发生两种情况。要么进程最终能运行（在系统最后为轻负荷时，如星期日凌晨 2 点），要么系统最终崩溃并失去所有未完成的低优先级进程（据说，在 1973 年关闭 MIT 的 IBM 7094 时，发现有一个低优先级进程是于 1967 年提交但是一直还未运行）。

低优先级进程无穷等待问题的解决之一是**老化**（aging）。老化是一种技术，以逐渐增加在系统中等待很长时间的进程的优先级。例如，如果优先级为从 127（低）到 0（高），那么可以每 15 分钟递减等待进程的优先级的值。最终初始优先级值为 127 的进程会有最高优先级并能执行。事实上，不超过 32 小时，优先级为 127 的进程会老化为优先级为 0 的进程。

5.3.4 轮转法调度

轮转法（round-robin, RR）调度算法是专门为分时系统设计的。它类似于 FCFS 调度，

但是增加了抢占以切换进程。定义一个较小时间单元，称为时间片（time quantum, or time slice）。时间片通常为 10~100 ms。将就绪队列作为循环队列。CPU 调度程序循环就绪队列，为每个进程分配不超过一个时间片的 CPU。

为了实现 RR 调度，将就绪队列保存为进程的 FIFO 队列。新进程增加到就绪队列的尾部。CPU 调度程序从就绪队列中选择第一个进程，设置定时器在一个时间片之后中断，再分派该进程。

接下来将可能发生两种情况。进程可能只需要小于时间片的 CPU 区间。对于这种情况，进程本身会自动释放 CPU。调度程序接着处理就绪队列的下一个进程。否则，如果当前运行进程的 CPU 区间比时间片要长，定时器会中断并产生操作系统中断，然后进行上下文切换，将进程加入到就绪队列的尾部，接着 CPU 调度程序会选择就绪队列中的下一个进程。

不过，采用 RR 策略的平均等待时间通常较长。考虑如下一组进程，它们在时间 0 时到达，其 CPU 区间以 ms 计：

| 进程 | 区间时间 |
|-------|------|
| P_1 | 24 |
| P_2 | 3 |
| P_3 | 3 |

如果使用 4 ms 的时间片，那么 P_1 会执行最初的 4 ms。由于它还需要 20 ms，所以在第一时间片之后它会被抢占，而 CPU 就被交给队列中的下一个进程。由于 P_2 不需要 4 ms，所以在其时间片用完之前会退出。CPU 接着被交给下一个进程，即进程 P_3 。在每个进程都得到了一个时间片之后，CPU 又交给了进程 P_1 以继续执行。因此，RR 调度结果如下：



平均等待时间为 $17/3 = 5.66$ ms。

对于 RR 调度算法，队列中没有进程被分配超过一个时间片的 CPU 时间（除非它是唯一可运行的进程）。如果进程的 CPU 区间超过了一个时间片，那么该进程会被抢占，而被放回到就绪队列。RR 调度算法是可抢占的。

如果就绪队列中有 n 个进程且时间片为 q ，那么每个进程会得到 $1/n$ 的 CPU 时间，其长度不超过 q 时间单元。每个进程必须等待的 CPU 时间不会超过 $(n-1)q$ 个时间单元，直到它的下一个时间片为止。例如，如果有 5 个进程，且时间片为 20 ms，那么每个进程每 100 ms 会得到不超过 20 ms 的时间。

RR 算法的性能很大程度上依赖于时间片的大小。在极端情况下，如果时间片非常大，那么 RR 算法与 FCFS 算法一样。如果时间片很小（如 1 ms），那么 RR 算法称为处理器共

享,(从理论上来说) n 个进程对于用户都有它自己的处理器,速度为真正处理器速度的 $1/n$ 。这种方法用在 Control Data Corporation (CDC) 的硬件上,可以用一组硬件和 10 组寄存器实现 10 个外设处理器。硬件为一组寄存器执行一个指令,然后为下一组执行。这种循环不断进行,形成了 10 个慢处理器而不是 1 个快处理器。(实际上,由于处理器比内存快很多,而每个指令都要使用内存,所以这些处理器并不比 10 个真正处理器慢很多。)

如果使用软件,那么还必须考虑上下文切换对 RR 调度的影响。假设只有一个需要 10 个时间单元的进程。如果时间片为 12 个时间单元,那么进程在一个时间片不到就能完成,且没有额外开销。如果时间片为 6 个时间单元,那么进程需要 2 个时间片,并产生了一个上下文切换。如果时间片为 1 个时间单元,那么就会有 9 个上下文切换,相应地使进程执行减慢(见图 5.4)。

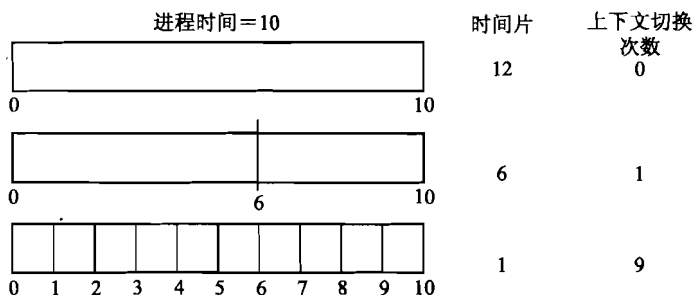


图 5.4 小的时间片如何增加上下文切换开销

因此,人们希望时间片要比上下文切换时间长。如果上下文切换时间约为时间片的 10%,那么约 10%的 CPU 时间会浪费在上下文切换上。事实上,绝大多数现代操作系统的时间分配为 10~100 ms,上下文切换的时间一般少于 10 μ s。因此,上下文切换的时间仅占时间片的一小部分。

周转时间也依赖于时间片的大小。正如从图 5.5 中所看到的,这组进程的平均周转时间并未随着时间片大小的增加而改善。通常,如果绝大多数进程能在一个时间片内完成,那么平均周转时间会改善。例如有 3 个进程,都需要 10 个时间单元,如果时间片为 1 个时间单元,那么平均周转时间为 29。如果时间片为 10,那么平均周转时间会降为 20。如果再加上上下文切换时间,那么平均周转时间对于较小时间片会增加,这是因为需要更多的上下文切换。

尽管时间片应该比上下文切换时间长,但也不能太大。如果时间片太大,那么 RR 调度就演变成了 FCFS 调度。根据经验,80%的 CPU 区间应该小于时间片。

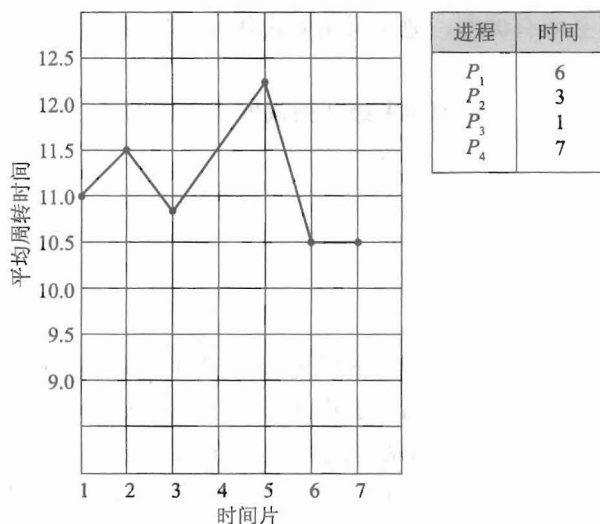


图 5.5 显示周转时间随着时间片的大小而改变

5.3.5 多级队列调度

在进程可容易地分成不同组的情况下，可以建立另一类调度算法。例如，一个常用的划分方法是前台（交互）进程和后台（批处理）进程。这两种不同类型的进程具有不同响应时间要求，也有不同调度需要。另外，与后台进程相比，前台进程要有更高（或外部定义）的优先级。

多级队列调度算法（multilevel queue scheduling algorithm）将就绪队列分成多个独立队列（见图 5.6）。根据进程的属性，如内存大小、进程优先级、进程类型，一个进程被永久地分配到一个队列。每个队列有自己的调度算法。例如，前台进程和后台进程可处于不同队列。前台队列可能采用 RR 算法调度，而后台队列可能采用 FCFS 算法调度。

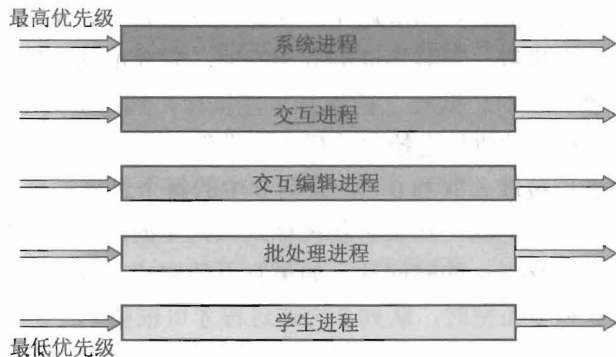


图 5.6 多级队列调度

另外，队列之间必须有调度，通常采用固定优先级抢占调度。例如，前台队列可以比后台队列具有绝对的优先级。

现在来研究一下具有 5 个队列的多级队列调度算法的例子，按优先级来排列：

- ① 系统进程。
- ② 交互进程。
- ③ 交互编辑进程。
- ④ 批处理进程。
- ⑤ 学生进程。

每个队列与更低层队列相比有绝对的优先级。例如，只有系统进程、交互进程和交互编辑进程队列都为空，批处理队列内的进程才可运行。如果在一个批处理进程运行时有一个交互进程进入就绪队列，那么该批处理进程会被抢占。

另一种可能是在队列之间划分时间片。每个队列都有一定的 CPU 时间，这可用于调度队列内的进程。例如，对于前台-后台队列的例子，前台队列可以有 80% 的 CPU 时间用于在进程之间进行 RR 调度，而后台队列可以有 20% 的 CPU 时间采用 FCFS 算法调度进程。

5.3.6 多级反馈队列调度

通常在使用多级队列调度算法时，进程进入系统时被永久地分配到一个队列。例如，如果前台进程和后台进程分别有独立队列，进程并不从一个队列转移到另一个队列，这是因为进程并不改变前台或后台性质。这种设置的优点是低调度开销，缺点是不够灵活。

与之相反，**多级反馈队列调度算法**（multilevel feedback queue scheduling algorithm）允许进程在队列之间移动。主要思想是根据不同 CPU 区间的特点以区分进程。如果进程使用过多 CPU 时间，那么它会被转移到更低优先级队列。这种方案将 I/O 约束和交互进程留在更高优先级队列。此外，在较低优先级队列中等待时间过长的进程会被转移到更高优先级队列。这种形式的老化阻止饥饿的发生。

例如，考虑一个多级反馈队列调度程序，它有三个队列，从 0~2（图 5.7）。调度程序首先执行队列 0 内的所有进程。只有当队列 0 为空时，它才能执行队列 1 内的进程。类似地，只有队列 0 和 1 都为空时，队列 2 的进程才能执行。到达队列 1 的进程会抢占队列 2 的进程。同样，到达队列 0 的进程会抢占队列 1 的进程。

进入就绪队列的进程被放入队列 0 内。队列 0 中的每个进程都有 8 ms 的时间片。如果一个进程不能在这一时间内完成，那么它就被移到队列 1 的尾部。如果队列 0 为空，队列 1 的头部进程会得到一个 16 ms 的时间片。如果它不能完成，那么将被抢占，并被放到队列 2 中。只有当队列 0 和 1 为空时，队列 2 内的进程才可根据 FCFS 来运行。

这种调度算法将给那些 CPU 区间不超过 8 ms 的进程最高优先级。这种进程可以很快地得到 CPU，完成其 CPU 区间，并处理下一个 I/O 区间。所需超过 8 ms 但不超过 24 ms

的进程也会很快被服务，但是它们的优先级比最短进程要低一点。长进程会自动沉入到队列 2，在队列 0 和队列 1 不用的 CPU 周期可按 FCFS 顺序来服务。

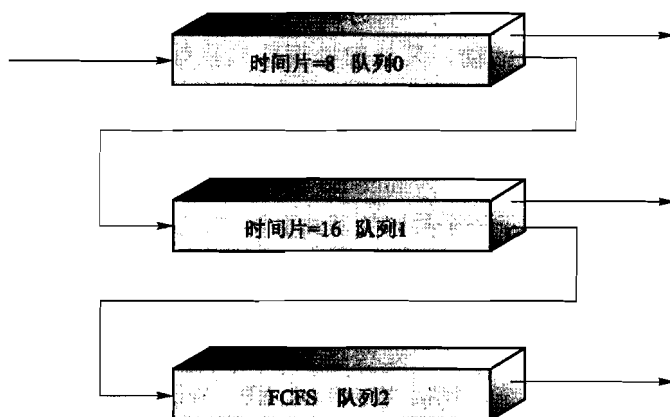


图 5.7 多级反馈队列

通常，多级反馈队列调度程序可由下列参数来定义：

- 队列数量。
- 每个队列的调度算法。
- 用以确定何时升级到更高优先级队列的方法。
- 用以确定何时降级到更低优先级队列的方法。
- 用以确定进程在需要服务时应进入哪个队列的方法。

多级反馈队列调度程序的定义使它成为最通用的 CPU 调度算法。它可被配置以适应特定系统设计。不幸的是，由于需要一些方法来选择参数以定义最佳的调度程序，它也是最复杂的算法。

5.4 多处理器调度

迄今为止，主要集中讨论了单处理器系统内的 CPU 调度问题。如果有多个 CPU，则负载分配（load sharing）成为可能，但调度问题也相应地变得更为复杂。已试验过许多可能的方法，与单处理器中的 CPU 调度算法一样，没有最好的解决方案。下面简要讨论多处理器调度的相关问题。其中主要讨论处理器功能相同（或同构）的系统，可以将任何处理器用于运行队列内的任何进程（但请注意，即使对同构多处理器，也有一些调度限制。考虑一个系统，有一个 I/O 设备与一个处理器通过私有总线相连，希望使用该设备的进程必须调度到该处理器上运行）。

5.4.1 多处理器调度的方法

在一个多处理器中，CPU 调度的一种方法是让一个处理器(主服务器)处理所有的调度决定、I/O 处理以及其他系统活动，其他的处理器只执行用户代码。这种**非对称多处理**(asymmetric multiprocessing)方法更为简单，因为只有一个处理器访问系统数据结构，减轻了数据共享的需要。

另一种方法是使用**对称多处理**(symmetric multiprocessing, SMP)方法，即每个处理器自我调度。所有进程可能处于一个共同的就绪队列中，或每个处理器都有它自己的私有就绪进程队列。无论如何，调度通过每个处理器检查共同就绪队列并选择一个进程来执行。正如将要在第 6 章中看到的，如果多个处理器试图访问和更新一个共同数据结构，那么每个处理器必须仔细编程：必须确保两个处理器不能选择同一进程，且进程不会从队列中丢失。事实上，许多现代操作系统，包括 Windows XP、Windows 2000、Solaris、Linux 和 Mac OS X，它们都支持 SMP。

下面的讨论主要是关于 SMP 系统的。

5.4.2 处理器亲和性

考虑一下，当一个进程在一个特定处理器上运行时，缓存中会发生些什么。进程最近访问的数据进入处理器缓存，结果是进程所进行的连续内存访问通常在缓存中得以满足。现在考虑一下，如果进程移到其他处理器上时，会发生什么：被迁移的第一个处理器的缓存中的内容必须为无效，而将要迁移到的第二个处理器的缓存需重新构建。由于使缓存无效或重新构建的代价高，绝大多数 SMP 系统试图避免将进程从一个处理器移至另一个处理器，而是努力使一个进程在同一个处理器上运行，这被称为**处理器亲和性**，即一个进程需有一种对其运行所在的处理器的亲和性。

处理器亲和性有几种形式。当一个操作系统具有设法让一个进程保持在同一个处理器上运行的策略，但不能做任何保证时，则会出现**软亲和性**(soft affinity)。此时，进程可能在处理器之间移动。有些系统，如 Linux，还提供一个支持**硬亲和性**(hard affinity)的系统调用，从而允许进程指定它不允许移至其他处理器上。

5.4.3 负载平衡

在 SMP 系统中，保持所有处理器的工作负载平衡，以完全利用多处理器的优点，这是很重要的。否则，将会产生一个或多个处理器空闲，而其他处理器处于高工作负载状态，并有一系列进程在等待 CPU。负载平衡(load balancing)设法将工作负载平均地分配到 SMP 系统中的所有处理器上。值得注意的是，负载平衡通常只是对那些拥有自己私有的可执行进程的处理器而言是必要的。在具有共同队列的系统中，通常不需要负载平衡，因为一旦

处理器空闲，它立刻从共同队列中取走一个可执行进程。但同样值得注意的是，在绝大多数支持 SMP 的当代操作系统中，每个处理器都具有一个可执行进程的私有队列。

负载均衡通常有两种方法：**push migration** 和 **pull migration**。对于 **push migration**，一个特定的任务周期性地检查每个处理器上的负载，如果发现不平衡，即通过将进程从超载处理器移到（或推送）空闲或不太忙的处理器，从而平均地分配负载。当空闲处理器从一个忙的处理器上推送（**pull**）一个等待任务时，发生 **pull migration**。**push migration** 和 **pull migration** 不能相互排斥，事实上，在负载均衡系统中它们常被并行地实现。例如，在 Linux 调度程序（参见 5.6.3 小节）及适用于 FreeBSD 系统的 ULE 调度程序中实现了这两种技术。Linux 每过 200 ms（**push migration**）或每当一个处理器的运行队列为空时（**pull migration**），运行其负载均衡算法。

有趣的是，负载均衡常会抵消 5.4.2 小节所介绍的处理器亲和性的优点。即保持一个进程在同一处理器上运行的优点在于，进程可以利用它在处理器缓存中的数据。无论是从一个处理器向另一处理器 **push** 或 **pull** 进程，都使此优点失效。事实上，在系统工程中，关于何种方式是最好的，没有绝对的规则。因此，在某些系统中，空闲的处理器常会从非空闲的处理器中 **pull** 进程，而在另一些系统中，只有当不平衡达到一定额度后才会移动进程。

5.4.4 对称多线程

通过提供多个物理处理器，SMP 系统允许同时运行几个线程。另一种方法是提供多个**逻辑**（而不是**物理的**）处理器来实现。这种方法被称为对称多线程（SMT），在 Intel 处理器中，它也被称为**超线程**（hyperthreading）技术。

SMT 的思想是在同一个物理处理器上生成多个逻辑处理器，向操作系统呈现一个多逻辑处理器的视图，即使系统仅有单处理器。每个逻辑处理器都有它自己的架构状态，包括通用目的和机器状态寄存器。进一步讲，每个逻辑处理器负责自己的中断处理，这意味着中断被送到并被逻辑处理器所处理，而不是物理处理器。否则，每个逻辑处理器共享其物理处理器的资源，如缓存或总线。图 5.8 所示是一个典型的具有两个物理处理器的 SMT 结构，每个物理处理器包括两个逻辑处理器。从操作系统的角度看来，系统有 4 个处理器可以工作。

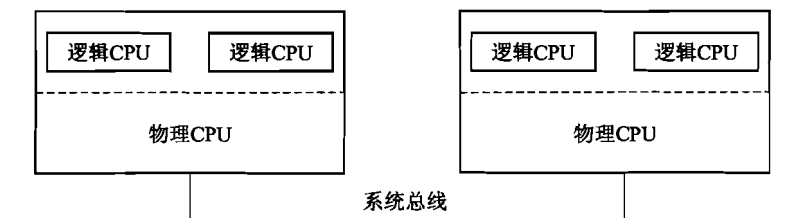


图 5.8 典型 SMT 架构

SMT 是硬件而不是软件提供的，认识到这一点很重要。硬件应该提供每个逻辑处理器的架构状态的表示以及中断处理方法。如果要在同一个 SMT 系统上运行，操作系统不必被特殊设计，但如果操作系统意识到它是在这样一个系统上运行，它还是可以得到性能的提升。例如，考虑一个具有两个物理处理器的系统，两个处理器均为空闲。调度程序首先设法把不同线程分别调度到每个物理处理器上，而不是调度到同一物理处理器中不同逻辑处理器上。否则，同一物理处理器上的两个逻辑处理器可能很忙，而另一个物理处理器则很空闲。

5.5 线程调度

第 4 章把线程引入到了进程模型，区别了*用户线程*和*内核线程*。对支持它们的操作系统而言，系统调度的是内核线程，而不是进程。用户线程由线程库管理，内核并不了解它们。为了能在 CPU 上运行，用户线程最终必须映射到相应的内核级线程，尽管这种映射可能是间接的，可能使用轻量级进程（LWP）。本节探讨有关用户线程和内核线程的调度问题，并介绍调度 Pthread 的例子。

5.5.1 竞争范围

用户线程与内核线程的区别之一在于它们是如何被调度的。在执行多对一模型（见 4.2.1 小节）和多对多（参见 4.2.3 小节）模型的系统上，线程库调度用户级线程到一个有效的 LWP 上运行，这被称为**进程竞争范围**（process-contention scope, PCS）方法，因为 CPU 竞争发生在属于相同进程的线程之间。当提及线程库调度用户线程到有效的 LWP 时，并不意味着线程实际上就在 CPU 上运行，这需要操作系统将内核线程调度到物理 CPU 上。为了决定调度哪个内核线程到 CPU，内核采用**系统竞争范围**（system-contention scope, SCS）方法来进行。采用 SCS 调度方法，竞争 CPU 发生在系统的所有线程中，采用一对一的模型（如 Windows XP、Solaris 9、Linux）的系统，调度仅使用 SCS 方法。

典型地，PCS 是根据优先级完成的——调度程序选择具有最高优先级的可运行的线程来运行。用户级线程优先级由程序员给定，并且不被线程库调节，尽管有些线程库允许程序员改变线程的优先级。值得注意的是，PCS 通常抢占当前具有较高优先级的正在运行的线程，但在具有相同优先级的线程间并没有时间分割（参见 5.3.4 小节）的保证。

5.5.2 Pthread 调度

4.3.1 小节给出了一个 POSIX Pthread 程序的例子，并介绍了 Pthread 的线程生成。现在，强调在线程生成过程中允许指定是 PCS 或 SCS 的 POSIX Pthread API。Pthread 识别下

面的竞争范围值:

- PTHREAD_SCOPE_PROCESS 调度线程采用 PCS 调度。
- PTHREAD_SCOPE_SYSTEM 调度线程采用 SCS 调度线程。

在实现多对多模型（参见 4.2.3 小节）的系统上，PTHREAD_SCOPE_PROCESS 方法调度用户线程到有效的 LWP 上。LWP 的数量由线程库维持，可能采用调度程序来激活（参见 4.4.6 小节）。在多对多模型的系统上，PTHREAD_SCOPE_SYSTEM 调度方法将为每个用户级线程生成并绑定一个 LWP，采用一对一方法（参见 4.2.2 小节）有效地映射线程。

Pthread IPC 为得到及设置竞争范围方法，提供下面两个函数：

- pthread_attr_setscope (pthread_attr_t *attr, int scope)。
- pthread_attr_getscope(pthread_attr_t *attr, int *scope)。

两个函数中的第一个参数包括一个到线程属性集的指针，pthread_attr_setscope()函数的第二个参数被传递 PTHREAD_SCOPE_SYSTEM 或 PTHREAD_SCOPE_PROCESS 的值，表示竞争范围如何被设置的。在 pthread_attr_getscope()函数中，第二个参数包含一个到赋予竞争范围当前值的整数值的指针。如果出错，每个函数返回一个非零值。

图 5.9 给出一个 Pthread 程序，它首先决定现有的竞争范围并将之赋予 PTHREAD_SCOPE_PROCESS。然后生成 5 个将使用 SCS 调度方法来运行的独立线程。注意，在某些系统中，仅允许特定的竞争范围值。例如，Linux 和 Mac OS X 就仅允许 PTHREAD_SCOPE_SYSTEM。

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main ( int argc, char *argv[] )
{
    int i, scope;
    pthread_t tid [ NUM_THREADS];
    pthread_attr_t attr;

    /*get the default attributes */
    pthread_attr_init(&attr);

    /* first inquire on the current scope */
    if ( pthread_attr_getscope(&attr, &scope) != 0 )
        fprintf ( stderr, " Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS ");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM ");
    }
}
```

```
else
    fprintf ( stderr, "Illegal scope value. \n" );
}

/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope ( &attr, PTHREAD_SCOPE_SYSTEM);

/* create the threads */
for ( i = 0; i < NUM_THREADS; i++ )
    pthread_create( &tid[i], &attr, runner, NULL);

/* now join on each thread */
for ( i = 0; i < NUM_THREADS; i++ )
    pthread_join( tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner( void *param)
{
    /* do some work...*/

    pthread_exit(0);
}
```

图 5.9 Pthread 调度 API

5.6 操作系统实例

接下来介绍 Solaris、Windows XP 和 Linux 操作系统的调度方法。注意，现在讨论的是 Solaris 和 Windows XP 的内核线程的调度，记得 Linux 并不区分进程和线程，在讨论 Linux 调度程序时使用术语*任务*（task）。

5.6.1 实例：Solaris 调度

Solaris 采用基于优先级的线程调度。根据优先级不同，它有 4 类调度，分别为：

- 实时。
- 系统。
- 分时。
- 交互。

每个类型内有不同的优先级和调度算法。Solaris 调度如图 5.10 所示。

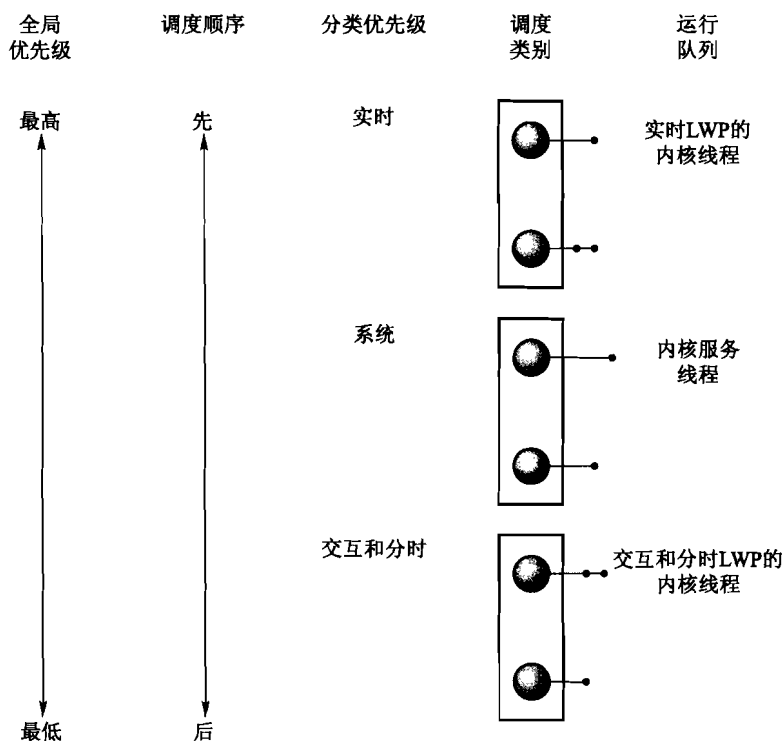


图 5.10 Solaris 调度

进程默认的调度类型是分时。分时调度方法采用多级反馈队列，动态地调整优先级和赋予不同长度的时间片。默认地，在优先级和时间片之间有反比关系：优先级越高，时间片越小；优先级越低，时间片越大。交互进程通常有更高的优先级，CPU 约束进程有更低的优先级。这种调度策略对交互进程有好的响应时间，对 CPU 约束进程有好的吞吐量。交互类型与分时类型使用同样的调度策略，但是它能给窗口应用程序更高的优先级以提高性能。

图 5.11 给出了调度交互和分时线程的分配表。这两种调度类型包括 60 个优先级，但为了简洁起见，此处仅给出其中的少数几个。图 5.11 中的分配表包括下面字段：

- **优先级：**分时和交互类型的依赖类型的优先级。数值越高，优先级越大。
- **时间片：**优先级相关的时间片。这表明优先级与时间片之间相反的关系：最低优先级（优先级为 0）具有最长的时间片（200 ms），最高优先级（优先级为 59）具有最短的时间片（20 ms）。
- **时间片到期：**用完了其全部时间片而未堵塞的线程的新的优先级。这种线程被认为是 CPU 密集的。如图 5.11 中所示，这些线程会被降低优先级。

| 优先级 | 时间片 | 时间片到期 | 从睡眠返回 |
|-----|-----|-------|-------|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

图 5.11 Solaris 中的交互和时间共享线程的分发表

• **从睡眠中返回**：从睡眠中返回的线程的优先级（如等待 I/O）。如表中所示，当一个等待线程有 I/O 可用的时候，它的优先级被提高到 50~59，以支持给交互进程提供好的响应时间的调度策略。

Solaris 9 引入了两种新的调度类型：**固定优先级** (fixed priority) 和 **公平共享** (fair share)。固定优先级类型的线程具有与分时类型相同的优先级范围，但其优先级不能动态调节。公平共享调度类型采用 **CPU 共享** 代替优先级来做出调度决定。CPU 共享表明了可用 CPU 资源的权利，并被分配进程集（被称为一个 **project**）。

Solaris 使用系统类来运行内核进程，如调度程序和换页服务。一旦创建，系统进程的优先级就不改变。系统类专为内核所使用（在内核模式下运行的用户进程并不属于系统类）。

实时类型的线程具有最高优先级。这种安排允许实时进程保证在给定时间内响应。实时进程能在其他类型的进程之前运行。通常，只有少数进程属于实时类型。

每种调度类型具有一定的优先级集合。然而，调度程序会将特定类的优先级转换为全局优先级，并从中选择最高全局优先级线程来执行。所选择的线程会在 CPU 上执行，直到它：（1）阻塞，（2）用完时间片，或（3）被更高优先级的线程抢占。如果多个线程具有同样优先级，那么调度程序采用循环队列。如前所讲，传统的 Solaris 使用多对多模型（见 4.2.3 小节），但 Solaris 9 却使用一对一模型（见 4.2.2 小节）。

5.6.2 实例：Windows XP 调度

Windows XP 采用基于优先级的、抢占调度算法来调度线程。Windows XP 调度程序确

保最高优先级的线程总是运行。Windows XP 内核中用于处理调度的部分称为**调度程序**。由调度程序选择运行的线程会一直运行，直到被更高优先级的进程所抢占，或终止，或其时间片已到，或调用了阻塞系统调用，如 I/O。如果在低优先级线程运行时更高优先级的实时线程就绪，那么低优先级线程被抢占。这种抢占使得实时线程在需要使用 CPU 时能优先得到使用。

调度程序使用 32 级优先级方案以确定线程执行的顺序。优先级分为两大类型：**可变类型**（variable class）包括优先级从 1~15 的线程，**实时类型**（real-time class）包括优先级从 16~31 的线程（还有一个线程运行在优先级 0，它用于内存管理）。调度程序为每个调度优先级使用一个队列，从高到低检查队列，直到它发现一个线程可以执行。如果没有找到就绪线程，那么调度程序会执行一个称为**空闲线程**（idle thread）的特别线程。

在 Windows XP 内核和 Win32 API 的优先级数值之间有一个关系。Win32 API 定义了一个进程可能属于的一些优先级类型。它们包括：

- REALTIME_PRIORITY_CLASS。
- HIGH_PRIORITY_CLASS。
- ABOVE_NORMAL_PRIORITY_CLASS。
- NORMAL_PRIORITY_CLASS。
- BELOW_NORMAL_PRIORITY_CLASS。
- IDLE_PRIORITY_CLASS。

除了 REALTIME_PRIORITY_CLASS 外，所有优先级类型都是可变类型优先级，这意味着属于这些类型的线程优先级能改变。

每个给定优先级类型的线程拥有相对优先级。相对优先级的值包括：

- TIME_CRITICAL。
- HIGHEST。
- ABOVE_NORMAL。
- NORMAL。
- BELOW_NORMAL。
- LOWEST。
- IDLE。

每个线程的优先级是基于它所属的优先级类型和它在其中的相对优先级。图 5.12 说明了这种关系。每个类型的值出现在顶行。左列包括不同的相对优先级的值。例如，如果一个线程属于 ABOVE_NORMAL_PRIORITY_CLASS 类型，且相对优先级为 NORMAL，那么该线程的优先级为 10。

另外，每个线程在其所属的类型中有一个基础优先级值。默认地，基础优先级为一个类型中的 NORMAL 相对优先级的值。每个优先级类型的基础优先级为：

- REALTIME_PRIORITY_CLASS——24。
- HIGH_PRIORITY_CLASS——13。
- ABOVE_NORMAL_PRIORITY_CLASS——10。
- NORMAL_PRIORITY_CLASS——8。
- BELOW_NORMAL_PRIORITY_CLASS——6。
- IDLE_PRIORITY_CLASS——4。

| | REALTIME_PRIORITY_CLASS | HIGH_PRIORITY_CLASS | ABOVE_NORMAL_PRIORITY_CLASS | NORMAL_PRIORITY_CLASS | BELOW_NORMAL_PRIORITY_CLASS | IDLE_PRIORITY_CLASS |
|---------------|-------------------------|---------------------|-----------------------------|-----------------------|-----------------------------|---------------------|
| TIME_CRITICAL | 31 | 15 | 15 | 15 | 15 | 15 |
| HIGHEST | 26 | 15 | 12 | 10 | 8 | 6 |
| ABOVE_NORMAL | 25 | 14 | 11 | 9 | 7 | 5 |
| NORMAL | 24 | 13 | 10 | 8 | 6 | 4 |
| BELOW_NORMAL | 23 | 12 | 9 | 7 | 5 | 3 |
| LOWEST | 22 | 11 | 8 | 6 | 4 | 2 |
| IDLE | 16 | 1 | 1 | 1 | 1 | 1 |

图 5.12 Windows XP 优先级

进程通常属于 NORMAL_PRIORITY_CLASS, 除非父进程为 IDLE_PRIORITY_CLASS 或在创建进程时指定了其他类型。线程的初值通常为线程所属进程的基础优先级。

在线程时间片用完时, 线程被中断。如果属于可变优先级类型, 那么优先级降低。不过, 优先级绝不会降低到基础优先级之下。降低线程优先级限制了 CPU 约束线程的 CPU 使用。当可变优先级线程从等待操作释放时, 调度程序提升其优先级。提升多少与线程等待什么有关, 例如, 等待键盘 I/O 的线程会得到较大提升, 而等待磁盘操作的线程得到一般提升。这种策略能给使用鼠标和窗口的交互线程更好的响应时间。它也能让 I/O 线程保持 I/O 设备忙, 同时允许计算约束线程在后台使用空闲的 CPU 周期。这种策略为包括多个分时操作系统的 UNIX 所采用。另外, 与用户交互的当前窗口也会得到优先级提升, 以缩短其响应时间。

当用户运行交互程序时, 系统需要为该进程提供特别好的性能。为此, Windows XP 对 NORMAL_PRIORITY_CLASS 进程有一个特别调度规则。Windows XP 区分前台进程(在屏幕上选择的)和后台进程(没有选择的)。当一个进程进入前台时, Windows XP 增加其调度时间片的倍数, 通常为 3。这一增加给了前台进程三倍多的运行时间。

5.6.3 实例: Linux 调度

在 2.5 版本之前, Linux 内核运行传统的 UNIX 调度算法。但传统的 UNIX 调度算法存

在两个问题，即它不提供对 SMP 系统足够的支持，以及当系统任务数量增加时不能按比例调整。在 2.5 版本中，调度程序被分解，内核提供了在固定时间内运行的调度算法，即 $O(1)$ ，而不管系统中的任务多少。新的调度程序还提供了对 SMP 的支持，包括处理器亲和性和负载平衡，以及提供了公平及对交互式任务的支持。

Linux 调度程序是抢占的、基于优先级的算法，具有两个独立的优先级范围：从 0~99 的 **real-time** 范围和从 100~140 的 **nice** 范围。这两个范围映射到全局优先级，其中数值越低表明优先级越高。

与包括 Solaris（参见 5.6.1 小节）和 Windows XP（参见 5.6.2 小节）在内的其他许多系统的调度程序不同，Linux 给较高的优先级分配较长的时间片，给较低的优先级分配较短的时间片。图 5.13 表明了优先级与时间片长度之间的关系。

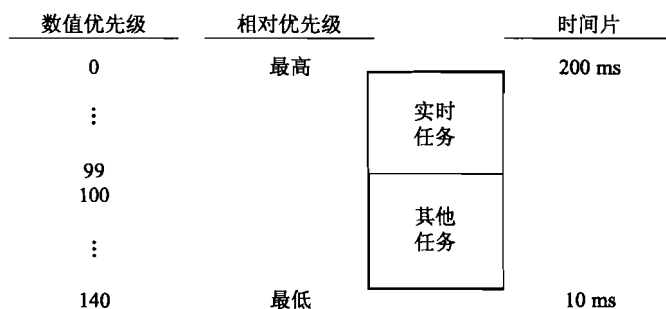


图 5.13 优先级和时间片长度的关系

一个可运行的任务被认为适合在 CPU 上执行，只要它在其时间片中具有剩余的时间。当任务耗尽其时间片后，它被认为是到期了，不适合再执行，直到所有其他任务都耗尽了它们的时间片。内核在运行队列数据结构中维护所有可运行任务的列表。由于对 SMP 的支持，每个处理器维护它自己的运行队列，并独立地调度它自己。每个运行队列包括两个优先级队列——活动的和到期的。活动队列包括所有在其时间片中尚有剩余时间的任务，而到期队列包括所有已到期的任务。每个优先级队列都有一个根据其优先级索引的任务列表（图 5.14），调度程序从活动队列中选择最高优先级的任务来在 CPU 上执行。对于多处理器系统，这意味着每个处理器从其自己的运行队列中调度最高优先级的任务。当所有任务都耗尽其时间片（即活动队列为空）时，两个优先级队列相互交换，到期队列变为活动队列，反之亦然。

Linux 根据在 5.5.2 小节中介绍过 POSIX.1b 来实现实时调度。实时任务被分配静态优先级，所有其他任务都具有动态优先级，并基于它们自己的 **nice** 值加上或减去 5。任务的交互性决定了从 **nice** 值中加上还是减去 5，而任务的交互性决定于它在等待 I/O 时沉睡了多长时间。交互性更强的任务通常具有更长的沉睡时间，因此更可能按 -5 来调整，因为调

度程序偏爱交互式任务。这种调度的结果将可能使这些任务的优先级更高。相反地，睡眠时间短的任务通常更受 CPU 制约，因此使其优先级更低。

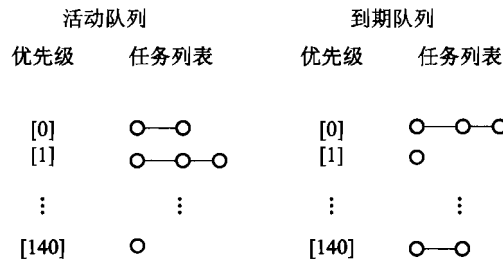


图 5.14 根据优先级编号的任务列表

当任务耗尽其时间片并移至到期队列中后，重新计算动态优先级。因此，当两个队列交换后，新的活动队列中的所有任务被分配以新的优先级及相应的时间片。

5.7 算法评估

如何选择 CPU 调度算法以用于特定系统？正如在 5.3 节所看到，调度算法有许多，且各有自己的参数。因此，选择算法可能会比较困难。

第一个问题是定义用于选择算法的准则。正如在 5.2 节所看到的，准则通常是通过 CPU 使用率、响应时间或吞吐量来定义的。为了选择算法，首先必须定义这些参数的相对重要性。准则可包括如下参数，如：

- 最大化 CPU 使用率，同时要求最大响应时间为 1s。
- 最大化吞吐量，例如，要求（平均）周转时间与总的执行时间成正比。

一旦定义了选择准则，需要评估所考虑的各种算法。接下来将讨论可以采用的不同的评估方法。

5.7.1 确定模型

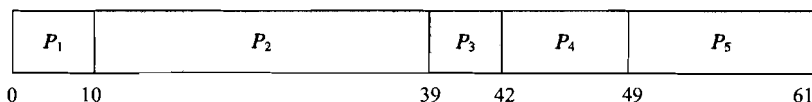
一种主要类型的评估方法称为分析评估法（analytic evaluation）。分析评估法使用给定算法和系统负荷，产生一个公式或数字，以评估对于该负荷算法的性能。

一种类型的分析评估是确定模型法（deterministic modeling）。这种方法采用特殊预先确定的负荷，计算在给定负荷下每个算法的性能。例如，假设下面的给定负荷。所有 5 个进程按所给顺序在时刻 0 时到达，CPU 区间时间的长度都以 ms 计：

| 进程 | 区间时间 |
|-------|------|
| P_1 | 10 |
| P_2 | 29 |
| P_3 | 3 |
| P_4 | 7 |
| P_5 | 12 |

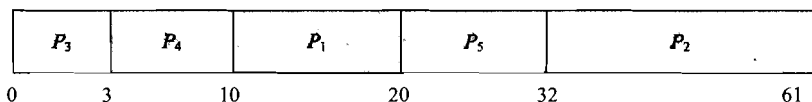
在这组进程中，主要研究 FCFS、SJF 和 RR 调度算法（时间片为 10 ms），并判断哪个算法的平均等待时间最小。

对于 FCFS 算法，按如下方式执行进程：



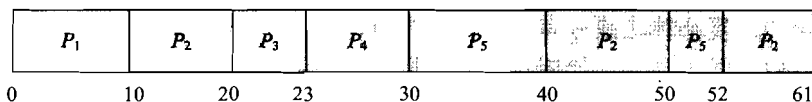
P_1 的等待时间是 0 ms， P_2 的等待时间是 10 ms， P_3 的等待时间是 39 ms， P_4 的等待时间是 42 ms， P_5 的等待时间是 49 ms。因此，平均等待时间 $(0 + 10 + 39 + 42 + 49) / 5 = 28$ ms。

对于非抢占 SJF 调度，按如下方式执行进程：



P_1 的等待时间是 10 ms， P_2 的等待时间是 32 ms， P_3 的等待时间是 0 ms， P_4 的等待时间是 3 ms， P_5 的等待时间是 20 ms。因此，平均等待时间 $(10 + 32 + 0 + 3 + 20) / 5 = 13$ ms。

对于 RR 算法，按如下方式执行进程：



P_1 的等待时间是 0 ms， P_2 的等待时间是 32 ms， P_3 的等待时间是 20 ms， P_4 的等待时间是 23 ms， P_5 的等待时间是 40 ms。因此，平均等待时间 $(0 + 32 + 20 + 23 + 40) / 5 = 23$ ms。

可以看到，在这种情况下，SJF 调度所产生的平均等待时间为 FCFS 调度的一半不到，而 RR 算法在两者之间。

确定模型不但简单而且快速。它给出了数字，以允许人们对算法进行比较。然而，它要求输入为精确数字，而且其答案只适用于这些情况。确定模型的主要用途在于描述调度算法和提供例子。在有的情况下，可以一次次地运行同样程序，并能精确测量程序的处理要求，以便使用确定模型来选择调度算法。而且，对于一组例子，确定算法可表示趋势以供分析和证明。例如，对于刚才所描述的环境（所有的进程和它们的时间都在时刻 0 已知），SJF 策略总能产生最小的等待时间。

5.7.2 排队模型

在许多系统上运行的进程每天都在变化，因此没有静态的进程（或时间）集合用于确定模型。然而，CPU 和 I/O 区间的分布是可以确定的。这些分布可以被测量，然后近似或简单估计。其结果是一个数学公式以表示特定 CPU 区间的分布。通常，这种分布是指数的，可用其平均值来表示。类似地，进程到达系统的时间分布（即到达时间分布）也必须给定。根据这两种分布，可以为绝大多数算法计算平均吞吐量、利用率和等待时间等。

计算机系统可描述为服务器网络。每个服务器都有一个等待进程队列。CPU 是具有就绪队列的服务器，而 I/O 系统是具有设备队列的服务器。知道了到达率和服务率，可计算使用率、平均队列长度、平均等待时间等。这种研究称为**排队网络分析**（queueing-network analysis）。

作为一个例子，设 n 为平均队列长度（不包括正在服务的进程）， W 为队列的平均等待时间， λ 为新进程到达队列的平均到达率（如每秒三个进程）。那么，在进程等待的 W 时间内，则有 $\lambda \times W$ 个新进程到达队列。如果系统处于稳定状态，那么离开队列的进程的数量必须等于到达进程的数量。因此，

$$n = \lambda \times W$$

这一公式称为 **Little 公式**。Little 公式特别有用，因为它适用于任何调度算法和到达分布。

大家可以利用 Little 公式，根据 3 个变量中的两个来计算另外一个。例如，已知平均每秒 7 个进程到达，通常有 14 个进程在队列里，那么可以计算平均等待时间是每个进程 2 秒。

排队分析可用于比较调度算法，但它也有限制。就目前而言，可以处理的算法和分布还是比较有限的。复杂算法或分布的数学分析可能难于处理。因此，到达和处理分布被定义成不现实的但数学上易处理的形式，而且也需要一些不精确的独立假设。这些困难产生的结果是，队列模型只是现实系统的近似，计算的结果也值得怀疑。

5.7.3 模拟

为了获得更为精确的调度算法评估，可使用**模拟**（simulation）。模拟涉及对计算机系统进行建模。软件数据结构表示系统的主要组成部分。模拟程序有一个变量以表示时钟；当该变量的值增加时，模拟程序会修改系统状态以反映设备、进程和调度程序的活动。随着模拟程序的执行，用以表示算法性能的统计数字可以被收集并打印出来。

驱动模拟的数据可由许多方法产生。最为普通的方法使用随机数生成器，以根据概率分布生成进程、CPU 区间时间、到达时间、离开时间等。分布可以数学地（统一的、指数的、泊松的）或经验地加以定义。如果经验定义分布，那么要对所研究的真实系统进行测

量。其结果可用来定义真实系统事件的真实分布，该分布再用来驱动模拟。

然而，由于真实系统前后事件之间有关系，分布驱动模拟可能不够精确。频率分布只表示每个事件发生了多少次，它并不能表示事件的发生顺序。可以采用跟踪磁带来解决这个问题。通过监视真实的系统，记录下事件发生的顺序来建立跟踪磁带（见图 5.15），然后使用这个顺序来驱动模拟。跟踪磁带提供了基于真实输入来比较两种算法的很好的方法。这种方法针对给定输入产生了精确的结果。

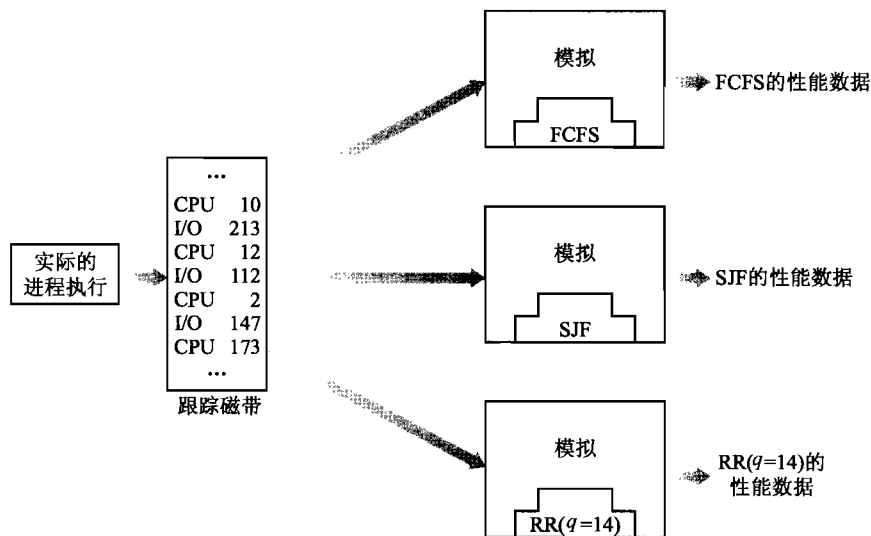


图 5.15 通过模拟来评估调度算法

然而模拟通常需要数小时的计算时间，所以是昂贵的。为了提供更精确的结果，需要更细致的模拟，但是也需要更多的计算时间。而且跟踪磁带需要大量的存储空间。最后，模拟程序的设计、编码、调试是其主要的工作。

5.7.4 实现

即使是模拟，精确度也是有限的。针对评估调度算法，唯一完全精确的方法是对它进行编程，将它放在操作系统内，并观测它如何工作。这一方法将真实算法放入操作系统，然后在真实操作系统内进行评估。

该方法的主要困难是其高昂的代价。所引起的代价不但包括对算法进行编程、修改操作系统以支持该算法（以及相关的数据结构），而且包括用户对不断改变操作系统的反应。绝大多数用户并不关心创建更好的操作系统，而只需要能执行程序并使用其结果。经常性地改变操作系统并不能帮助用户得到他们所想要的。

存在的另一个困难是算法所使用的环境会发生改变。环境变化不但包括新用户程序的编写和问题类型的变化,而且也包括调度程序性能所引起的结果。如果小进程获得优先级,那么用户会将大进程分成小进程。如果交互进程比非交互进程获得优先级,那么用户就切换到交互进程。

例如,研究人员设计一个系统,它通过观察终端 I/O 数量来自动将进程分成交互的和非交互的。如果一个进程在一分钟内没有对终端进行输入或输出,那么该进程就被划分为非交互的,并移到较低优先级队列。这种策略导致了如下情况:一位程序员修改了其程序,每隔一分钟不到的时间就输出一个任意字符到终端上。虽然该终端输出完全没有意义,但是系统却赋予该程序高的优先级。

最为灵活的调度算法可以为系统管理员和用户所改变,以使其能为特定的应用或应用集合所调节。例如,一个能完成高端图形应用的工作站与 Web 服务器或文件服务器有完全不同的调度需求。有些操作系统(特别是 UNIX 的几种版本)允许系统管理员为特定的应用配置调整调度参数。例如, Solaris 提供了 `dispadm` 命令,以允许系统管理员修改 5.6.1 小节所述的调度类的参数。

另一种方法是使用 API 来修改进程或线程的优先级。Java、POSIX 和 Win32 API 都提供了这类函数。这种方法的缺点在于调节一个系统或应用并不能在更通用的情况下改进性能。

5.8 小 结

CPU 调度的任务是从就绪队列中选择一个等待进程,并为其分配 CPU。CPU 由调度程序分配给所选中的进程。

先到先服务(FCFS)调度是最简单的调度算法,但是它会让短进程等待非常长的进程。最短作业优先(SJF)调度可证明是最佳的,它提供了最短平均等待时间。实现 SJF 调度比较困难,因为预测下一个 CPU 区间的长度有难度。SJF 算法是通用优先级调度算法(将 CPU 简单地分配给具有最高优先级的进程)的特例。优先级和 SJF 调度会产生饥饿。老化技术可阻止饥饿。

轮转法(RR)调度对于分时(交互)系统更为合适。RR 调度让就绪队列的第一个进程使用 CPU 的 q 个时间单元,这里 q 是时间片。在 q 时间单元之后,如果该进程还没有释放 CPU,那么它被抢占并放到就绪队列的尾部。该算法的主要问题是选择时间片。如果时间片太大,那么 RR 调度就成了 FCFS 调度;如果时间片太小,那么因上下文切换而引起的调度开销就过大。

FCFS 算法是非抢占的,而 RR 算法是抢占的。SJF 和优先级算法可以是抢占的,也可以是非抢占的。

多级队列调度算法允许多个不同算法用于各种类型的进程。最为常用的模型包括使用 RR 调度的前台交互队列，以及使用 FCFS 调度的后台批处理队列。多级反馈队列调度算法允许进程在队列之间迁移。

许多当前的计算机系统支持多处理器，并允许每个处理器独立地调度它自己。通常，每个处理器维护自己的私有进程（或线程）队列，它们都可以运行。与多处理器调度相关的问题包括处理器亲和性和负载平衡。

如果操作系统在内核级支持线程，那么必须调度线程而不是进程来执行。Solaris 和 Windows XP 就是这样的系统，它们采用抢占的、基于优先级的调度算法，并支持实时线程。Linux 进程调度也使用基于优先级算法，并提供实时支持。这三种操作系统通常偏爱交互进程而不是批处理进程或 CPU 约束进程。

因为有多种不同的调度算法可用，所以需要某种方法来选择它们。分析方法使用数学分析以确定算法性能。模拟方法通过对代表性的进程采用调度算法模拟并计算其性能来确定优劣。不过，模拟最多也只是提供对真实系统性能的近似，评估调度算法唯一可靠的技术是在真实系统上的实现算法并在真实环境中进行性能跟踪。

习 题

- 5.1 为什么对调度程序而言，区分 CPU 约束程序和 I/O 约束程序很重要？
- 5.2 讨论下列几对调度标准如何在一定设置中冲突：
- CPU 利用率和响应时间
 - 平均周转时间（turnaround time）和最大等待时间
 - I/O 设备利用率和 CPU 利用率
- 5.3 考虑用于预测下一个 CPU 区间长度的指数平均公式。将下面的值赋给算法中的参数的含义是什么？

- $\alpha = 0$ 且 $\tau_0 = 100$ ms
- $\alpha = 0.99$ 且 $\tau_0 = 10$ ms

- 5.4 考虑下面一组进程，进程占用的 CPU 区间长度以毫秒来计算：

| 进程 | 区间时间 | 优先级 |
|-------|------|-----|
| P_1 | 10 | 3 |
| P_2 | 1 | 1 |
| P_3 | 2 | 3 |
| P_4 | 1 | 4 |
| P_5 | 5 | 2 |

假设在 0 时刻进程以 P_1 、 P_2 、 P_3 、 P_4 、 P_5 的顺序到达。

- 画出 4 个 Gantt 图分别演示使用 FCFS、SJF、非抢占优先级（数字越小代表优先级越高）和 RR（时间片=1）算法调度时进程的执行过程。
- 每个进程在每种调度算法下的周转时间是多少？

- c. 每个进程在每种调度算法下的等待时间是多少？
- d. 哪一种调度算法的平均等待时间最小（对所有的进程）？

5.5 下面哪种调度算法能导致饥饿？

- a. 先到先服务
- b. 最短作业优先
- c. 轮转法
- d. 优先级

5.6 考虑 RR 调度算法的一个变种，在这个算法里，就绪队列里的项是指向 PCB 的指针。

- a. 在就绪队列中如果把两个指针指向同一个进程，会有什么效果？
- b. 这个方案的两个主要优点和两个主要缺点是什么？
- c. 如何修改基本的 RR 调度算法不用两个指针达到同样的效果？

5.7 考虑一个运行 10 个 I/O 约束任务和 1 个 CPU 约束任务的系统，假设 I/O 约束任务每进行 1 ms 的 CPU 计算发射一次 I/O 操作，以及每个 I/O 操作花费 10 ms 完成。同时假设上下文切换花费 0.1 ms，且所有的进程都是长运行任务。下列条件下 RR 调度程序的 CPU 利用率如何？

- a. 时间片为 1 ms
- b. 时间片为 10 ms

5.8 考虑一个实现多级队列调度的系统。计算机用户可以采用何种策略来最大化分配给用户进程的 CPU 时间？

5.9 考虑下面的动态改变优先级的抢占式优先级调度算法。大的优先级数代表高优先级。当一个进程在等待 CPU 时（在就绪队列中，但没执行），优先级以 α 速率改变；当它运行时，优先级以 β 速率改变。所有的进程在进入等待队列时指定优先级为 0。参数 α 和 β 可以进行设定得到许多不同的调度算法。

- a. $\beta > \alpha > 0$ 是什么算法？
- b. $\alpha < \beta < 0$ 是什么算法？

5.10 解释下面调度算法对短进程偏好程度上的区别。

- a. FCFS
- b. RR
- c. 多级反馈队列

5.11 采用 Windows XP 调度算法，下列情况的线程的数字优先级如何？

- a. 在 `REALTIME_PRIORITY_CLASS` 中的线程具有相对优先级 `HIGHEST`。
- b. 在 `NORMAL_PRIORITY_CLASS` 中的线程具有相对优先级 `NORMAL`。
- c. 在 `HIGH_PRIORITY_CLASS` 中的线程具有相对优先级 `ABOVE_NORMAL`。

5.12 考虑下面在 Solaris 操作系统的时间共享线程的调度算法中：

- a. 优先级为 10 的线程的时间片为多少（按 ms 计）？如果优先级为 55 呢？
- b. 假设优先级为 35 的线程使用了它的全部时间片而不产生阻塞，调度程序将分配给该线程的新优先级为多少？
- c. 假设优先级为 35 的线程在时间片到期之前阻塞在 IO 上。调度程序将分配给该线程的新优先级是多少？

5.13 传统的 UNIX 调度程序使优先级数字与优先级呈相反的关系。优先级数字越大，优先级越低。调度程序使用下列函数每计算进程优先级一次：

$$\text{Priority} = (\text{recent CPU usage} / 2) + \text{base}$$

其中, $\text{base} = 60$, 而 “recent CPU usage” 指的是从上次计算优先级以来进程使用 CPU 的频率。

假设进程 P_1 最近使用 CPU 为 40, 进程 P_2 为 18, 进程 P_3 为 10。当优先级被重新计算后, 这三个进程的新的优先级为多少? 基于此信息, 传统 UNIX 调度程序是提高还是降低了 CPU 约束进程的相对优先级?

文献注记

Corbato 等[1962]中描述了最初在 CTSS 系统中实现的反馈队列。Schrage[1967]分析了这种反馈队列调度系统。Kleinrock[1975]提供了关于抢占优先级调度算法的习题第 5.9 题。

Anderson 等[1989]、Lewis 和 Berg[1998]、Philbin 等[1996]讨论线程调度。Tucker 和 Gupta[1989]、Zahorjan 和 McCann[1990]、Feitelson 和 Rudolph[1990]、Leutenegger 和 Vernon[1990]、Blumofe 和 Leiserson[1994]、Polychronopoulos 和 Kuck[1987]以及 Lucco[1992]给出了关于多处理器的调度的论述。Fisher[1981]、Hall 等[1996]、Lowney 等[1993]考虑了关于在运行之间进程执行时间的问题。

Liu 和 Layland[1973]、Abbot[1984]、Jensen 等[1985]、Hong 等[1989]、Khanna 等[1992]给出了有关实时系统中的调度的论述。Zhao[1989]修改了一个特殊版的实时操作系统 *Operating System Review*。

Henry[1984]、Woodside[1986]、Kay 和 Lauder[1988]论述了公平共享调度。

Bach[1987]介绍了 UNIX V 操作系统中使用的调度策略, McKusick 等[1996]提出了 UNIX BSD 4.4 中所使用的策略, Black[1990]则介绍了 Mach 操作系统中所使用的策略。Bovet 和 Cesati[2002] 涉及了 Linux 调度。Mauro 和 McDougall[2001]给出了 Solaris 调度论述, Solomon [1998]、Solomon 和 Russinovich[2000]中分别包括 Windows NT 和 Windows 2000 的调度论述。Butenhof[1997]、Lewis 和 Berg[1998]介绍了 Pthread 系统中的调度问题。

第6章 进程同步

协作进程是可以与在系统内执行的其他进程互相影响的进程。互相协作的进程可以直接共享逻辑地址空间（即代码和数据），或者只通过文件或消息来共享数据。前者可通过轻量级进程或线程来实现，参见第4章。共享数据的并发访问可能会产生数据的不一致。本章将讨论各种机制，以用于确保共享同一逻辑地址空间的协作进程可有序地执行，从而能维护数据的一致性。

本章目标

- 引入临界区域问题，其解决方案可用于确保共享数据的一致性。
- 描述临界区域问题的多种软件解决方案。
- 描述临界区域问题的多种硬件解决方案。
- 引入原子事务的概念，并描述确保原子操作的有关机制。

6.1 背景

第3章讨论了一种系统模型，该模型包括若干协作顺序进程或线程（cooperating sequential processes or threads），这些进程或线程均异步执行且可能共享数据。下面用具有代表性的操作系统问题，即生产者-消费者问题，举例说明这个模型。在3.4.1小节中，还特地采用了有限缓冲方案来处理进程共享内存的问题。

现在回到有限缓冲问题的共享内存解决方案。正如所指出的，该解决方案允许同时在缓冲区内最多只有 `BUFFER_SIZE-1` 项。假如要修改这一算法以弥补这个缺陷。一种可能方案是增加一个整数变量 `counter`，并初始化为0。每当向缓冲区增加一项时，`counter` 就递增；每当从缓冲区移走一项时，`counter` 就递减。生产者进程代码可修改如下：

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

教育部高等教育司推荐
国外优秀信息科学与技术系列教学用书

一流的品质

优惠的价格

本套教学用书的特点

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

操作系统发展的又一关键时刻

非常小型的操作系统，如篇首的小恐龙所使用的驱动手持设备的操作系统，是 Silberschatz、Galvin 和 Gagne 第七版《操作系统概念》中的一种前沿应用。

通过保留最新的，保持有意义的，并改编为课程最需要的内容，这本引导市场潮流的教材继续指导着操作系统课程。第七版不仅提供最新且最有意义的系统，同时还从更深层次揭示了那些在当今操作系统发展过程中仍保持不变的基本概念。通过拥有这种坚实的概念基础，学生们能更容易理解与特定系统相关的细节问题。

作者 Abraham Silberschatz 是美国耶鲁大学计算机科学系教授，前任新泽西州 Murray Hill 的贝尔实验室信息科学研究中心副主任。Peter Baer Galvin 是技术合作公司的技术主管，曾任美国布朗大学计算机科学系的系统主管。作为顾问和培训教师，他在世界各地讲解和教授网络系统管理、安全和性能等。Greg Gagne 是美国盐湖城威斯特敏斯特学院计算机科学与数学系主任，除了讲授操作系统外，他还教授计算机网络、分布式系统、面向对象程序设计和数据结构等。



For more information,
please visit www.wiley.com.

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区及中国台湾地区）销售发行。

本书贴有 Wiley 防伪标签，无标签者不得销售



ISBN 978-7-04-028344-9



9 787040 283449 >

定价 60.00 元