



普通高等教育“十一五”国家级规划教材

操作系统教程

(第4版)

孙钟秀 主编

费翔林 骆斌 编著



高等教育出版社
Higher Education Press



面向 21 世纪课程教材
Textbook Series for 21st Century

操作系统教程（第4版）

本书特色：

- ◆ 保持和发扬前版特色。在前三版的基础上进行全面修订，既致力于传统操作系统基本概念、技术和方法的阐述，又融合现代操作系统最新技术发展和应用的讨论，着眼于操作系统学科知识体系的系统性、先进性和实用性。
- ◆ 突出重点教学内容。充分考虑大多数院校的实际教学需求，进一步精选内容、调整体系、合理组织，在教材的易读性、难点分散、循序渐进方面做出努力；重点突出操作系统的构造与整体设计、多道程序设计技术、进程与线程、并发程序设计、实时、文件系统等重点内容。
- ◆ 教材建设与软件产业的发展紧密结合。将成熟的基本原理与当代主流系统实例剖析相结合，有益于学生深入理解操作系统的整体概念，牢固掌握操作系统设计与实现的精髓。
- ◆ 配套教学资源丰富。提供与教材配套的电子教案、习题解答、实验课程教学大纲、实习报告样本等教学资源，并将推出配套的实验教程，使操作系统原理的讲授和实验环节紧密结合。

ISBN 978-7-04-023221-9

9 787040 232219 >

定价 38.00 元

TP316/194=2

2008

普通高等教育“十一五”国家级规划教材
面向 21 世纪课程教材

操作系统教程

(第 4 版)

孙钟秀 主编
费翔林 骆斌 编著

高等教育出版社

内容提要

操作系统是计算机系统的核心和灵魂,是计算机系统必不可少的组成部分,因而操作系统课程成为计算机相关专业的必修课,也是计算机应用从业人员必备的专业知识。本书在前三版的基础上进行全面修订,系统地介绍操作系统的经典内容和最新发展,选择当代具有代表性的主流操作系统 Linux 和 Windows 2003 作为实例贯穿全书。

本书共分八章,覆盖操作系统的基本概念、设计原理和实现技术,尽可能系统、全面地展示操作系统的概念、特性和精髓。与本书配套的《Linux 操作系统实验教程》同时出版,两门课程的教科书各有侧重,相辅相成完成操作系统教学任务。

本书既可作为高等学校计算机及相关专业的本科“操作系统”课程教材或参考书,也可供计算机技术和软件科技人员阅读和参考。

图书在版编目(CIP)数据

操作系统教程 / 孙钟秀主编; 费翔林, 骆斌编著. —4 版. —北京: 高等教育出版社, 2008. 4
ISBN 978 - 7 - 04 - 023221 - 9

I. 操… II. ①孙… ②费… ③骆… 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(2008)第 021356 号

策划编辑 倪文慧 责任编辑 康兆华 封面设计 于文燕 责任绘图 尹莉
版式设计 陆瑞红 责任校对 王超 责任印制 韩刚

出版发行 高等教育出版社
社址 北京市西城区德外大街 4 号
邮政编码 100011
总机 010 - 58581000

经 销 蓝色畅想图书发行有限公司
印 刷 北京中科印刷有限公司

开 本 787 × 1092 1/16
印 张 33
字 数 740 000

购书热线 010 - 58581118
免费咨询 800 - 810 - 0598
网 址 <http://www.hep.edu.cn>
<http://www.hep.com.cn>
网上订购 <http://www.landraco.com>
<http://www.landraco.com.cn>
畅想教育 <http://www.widedu.com>

版 次 1989 年 7 月第 1 版
2008 年 4 月第 4 版
印 次 2008 年 4 月第 1 次印刷
定 价 38.00 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 23221-00

第4版前言

《操作系统教程》(第3版)出版4年以来,已被国内几十所学校选为“操作系统”课程教材,同时,广大读者还提供大量宝贵的意见和建议,为改版做出了贡献,在此谨向所有读者表示衷心的感谢。本书被列入“普通高等教育‘十一五’国家级教材规划”,根据规划的制订原则之一“鼓励修订,锤炼精品”,第4版继承和发扬第3版的优点,既致力于传统操作系统的基本概念、基本原理、基本方法的阐述,又融合现代操作系统最新技术发展和应用的讨论,着眼于操作系统学科知识体系的科学性、系统性、先进性和实用性。修订版着重考虑以下几点。

(1) 在教材选材和内容上:参考国内外优秀教材和 ACM /IEEE2001 及 CCC2002 推荐的主题,符合教育部教学指导委员会“计算机科学与技术专业规范”中对操作系统课程大纲的要求,并且结合国内大多数学校计算机专业本科生的基本状况,全面审核操作系统课程知识领域中的知识单元和知识点,在内容新颖性、概念准确性、结构合理性、体系完整性、叙述清晰性等方面力求做得比第3版更好。

(2) 在教学计划和课时上:充分考虑多数院校的教学需求,在学时有限的情况下进一步精选内容、缩减篇幅,调整体系、合理组织。各院校在教学计划、教学要求、课程设置、学时安排、教学大纲,乃至学生水平等方面存在种种差异,采用本教材授课时,任课老师应酌情对内容进行取舍,课时充足时,可讲授全部内容,否则,应突出基本要点,舍弃某些实例,重点讲述原理。

(3) 在理论和实践的结合上:读者所提出的最多的建议是希望提供配套的实验教材,使课程的理论教学和实验环节形成统一整体。第4版既要提供操作系统原理的全面介绍,又要通过上机练习解决操作系统动手实践的问题。为此,把概念和原理性与应用和实践性内容区分开来,同时配套编写与出版《操作系统教程》(第4版)和《Linux 操作系统实验教程》。这两本教科书各有侧重,前者以讲授原理为主,后者为操作系统实践提供指导,两者相辅相成完成操作系统教学任务。

此外,第4版中所做的修订工作主要有:充实内容、突出重点;缩减篇幅、详略得当;调整结构、重写章节;简化实例、扩充习题。遵照多数读者的建议,全书中的算法均改用C语言风格描述。为了便于广大师生教学,本书配套提供电子教案,可与作者联系或登录高等教育出版社网站免费获取。

全书共分八章,内容包括:操作系统概论、处理器管理、同步、通信与死锁、存储管理、设备管理、文件管理、操作系统的安全与保护、网络和分布式操作系统,每章最后一节是小结,并配有丰富的思考题和应用题。

本教材由孙钟秀院士主编,费翔林和骆斌教授参编。衷心感谢南京大学谢立和谭耀铭教授在本教材前3版中所做的建设性工作;特别感谢上海交通大学尤晋元教授百忙中抽时间仔细审阅全书,并提出了许多极为宝贵的意见和建议;为了方便读者阅读和理解相关内容,由李贤、沈志

鹏按 C 语言风格重写部分示例程序；金志权教授及丛邵鹏、李敏、叶保留等对若干章节进行了讨论和校对，在此一并表示衷心的感谢。本教材中有些章节还引用参考文献中列出的国内外著作的一些内容，谨此向各位作者致以衷心的感谢和深深的敬意！

限于编者的水平，错误、不妥与不尽人意之处定然难免，恳请读者指正及赐教。作者的电子邮件地址为：feixl@nju.edu.cn 及 luobin@nju.edu.cn。

作 者

2007 年 10 月于南京

第3版前言

操作系统是计算机系统的重要组成部分,操作系统课程是计算机教育的必修课程,作为计算机专业的核心课,不但高等院校计算机相关专业学生必须学习它,而且从事计算机行业的从业人员也需要深入了解它。为了更好地学习和透彻地理解操作系统的基本原理和计算机系统的运作过程,一本适用的操作系统教材显得十分重要。本教材是多年来操作系统教学和科学的研究相结合的产物,是继《操作系统教程》第一版和第二版之后,更新教学内容后的新版本。本教材第一、二版多年来在南京大学和国内很多高校计算机专业的教学过程中得到了广泛的应用,曾在1992年第二届全国高等学校优秀教材评选中获国家级优秀教材奖。

进入20世纪90年代以后,计算机科学技术突飞猛进,而操作系统又是计算机领域最活跃的分支之一,操作系统的概念、新技术和新方法层出不穷,促使现代操作系统发生了巨大的变化。为了适应这种发展趋势,操作系统的教材必须尽快更新。除了反映经典内容外,当代操作系统的最新成果也应尽快、准确、全面地组织到教材中。国外非常重视操作系统教材的建设和更新工作,近年来又出版了若干有影响的操作系统教材。为此,我们在多年教学工作的基础上,结合国内外最新的资料和教材编写了本教材,以适应信息社会计算机科学技术飞速发展的形势和社会用人单位对计算机教学内容要求改革的迫切需求。

本教材的特点之一是:既致力于传统操作系统基本概念、基本技术、基本方法的阐述,又融合现代操作系统最新技术发展和应用的讨论,着眼于操作系统学科知识体系的系统性、先进性和实用性。本教材的特点之二是:把操作系统成熟的基本原理与当代具有代表性的具体实例;操作系统的概念与操作系统的实现技术;操作系统的理论知识与操作系统的实践实习紧密地结合起来。选择了具有代表性的Windows 2000 /XP 和 UNIX类(包括SVR4、Solaris、Linux)主流操作系统作为实例贯穿全书,这十分有益于学生深入理解操作系统的整体概念和牢固掌握操作系统设计与实现的精髓。本教材保持早期版本教材的编写特点,力求做到概念清晰、结构合理,内容丰富、取舍得当,由浅入深、循序渐进,既有利于学生的知识获取,又有利于学生的能力培养,希望能达到较好的教学效果。

本教材是一本关于操作系统的基本概念、基本方法、设计原理和实现技术的教材,目的是尽可能系统、清晰、全面、综合地展示当代操作系统的概念、特点、本质和精髓。全书共分八章,每章的最后一节是小结。全书涉及的内容如下:

第一章操作系统概论。介绍操作系统的基本概念、多道程序设计技术、操作系统的形成和发展,操作系统的分类;操作系统的服务、操作系统的功能、操作系统的接口;操作系统的结构,并以Windows 2000 /XP为例着重介绍了客户—服务器结构;对流行的一些主要操作系统也作了简单介绍。

第二章处理机管理。从处理器和中断技术开始,介绍了中断的概念、分类、处理、优先级和多

重中断。接着,引入进程和线程的概念,介绍进程管理的实现模型、线程不同级别的实现方法,介绍处理机调度的三个层次,着重讨论了各种单处理机调度算法,也涉及多处理机调度算法和实时调度算法。实例研究、讨论了 Windows 2000 /XP、Solaris 和 Linux 中断处理以及 UNIX SVR4、Windows 和 Linux 的处理机调度算法。

第三章并发进程。介绍进程的顺序性和并发性,进程的协作和竞争,以进程交互、进程控制、进程通信和进程死锁问题为重点,讨论并发程序设计的有关技术和各种进程互斥、同步、通信机制和工具。最后介绍 Windows 的同步和通信机制、Linux 的信号量机制。

第四章存储管理。讨论存储管理的基本功能、各种传统存储管理技术、虚拟存储管理技术和最新的存储管理技术,如多级页表、反置页表等。实例研究深入介绍 Intel x86 /Pentium 存储管理硬件设施、Windows 2000 /XP 虚拟存储管理和 Linux 虚拟存储管理。

第五章设备管理。讨论 I/O 硬件原理、I/O 控制方式、I/O 软件原理、I/O 缓冲技术,着重介绍磁盘驱动调度技术、RAID 技术以及设备分配/去配和虚拟设备技术,也介绍了具有通道的 I/O 系统管理。实例研究介绍了 Windows I/O 系统和 Linux 设备管理。

第六章文件管理。讨论文件概念、文件目录、文件逻辑结构、文件物理结构、文件的保护和保密、文件存储空间管理以及文件的操作和使用原理,也讨论了文件系统的新概念:主存映射文件和虚拟文件系统。实例研究介绍了 Windows 文件管理和 Linux 文件管理。

第七章操作系统安全与保护。讨论操作系统安全威胁和类型;操作系统保护的层次及保护的基本机制、策略和模型,其中着重讨论身份认证机制、授权机制、加密机制和审计机制;实例研究、介绍了 Windows 2000 /XP 安全机制。

第八章网络和分布式操作系统。简要介绍网络和分布式操作系统的基本概念和技术,包括网络和数据通信基础、网络体系结构、网络操作系统;分布式进程通信、分布式资源管理、分布式进程同步、分布式文件系统和进程迁移等。实例研究、介绍了 Windows 2000 /XP 网络体系结构和网络服务。

本教材的编写工作起始于 1999 年,通过南京大学计算机科学和技术系 98 级、99 级和 00 级三届本科学生的教学应用和其他教学系列的应用,在此基础上编写而成。为了便于教和学,我们还做了两项工作。一是与教材相配套提供了 ppt 讲稿。鉴于它的内容既不能做得太粗,这样无助于教和学;但也不能做得太细,显得太繁琐、累赘了。我们尽量做到繁简得当。各位老师在教学备课时,可以根据各校各相关专业的教学计划、教学需要和实际情况对配套提供的 ppt 讲稿进行增、删、改。二是订正和增加了大量思考题和应用题,便于老师布置作业,也便于学生课余选做。

本教材由孙钟秀院士主编,费翔林、骆斌、谢立参编。衷心感谢南京大学谭耀铭教授在本教材前两版中所做的许多建设性工作。特别感谢上海交通大学尤晋元教授百忙中抽空仔细审阅了全书,提出了许多极为宝贵的意见。本书的修订和出版还得到了南京大学 985 工程的经费支持,特此表示谢意;感谢陶先平、高阳和花蕾老师在教学过程中对教材提出的意见和建议;感谢田原、花蕾、张建莹、周德宇、王天青、蔡飞和裴永刚等同志在本书的 ppt 制作和校对过程中所提供的帮

助；感谢高等教育出版社的大力支持、合作和辛勤劳动。本教材中有些章节还引用了参考文献中列出的国内外著作的一些内容，谨此向各位作者致以衷心的感谢和深深的敬意！

限于编者的水平，错误与不妥之处定然难免，衷心希望读者指正及赐教。联系 E-mail 为：feixl@nju.edu.cn 及 luobin@nju.edu.cn。

作 者

2003 年 3 月

第2版前言

操作系统是计算机系统软件中的一个不可缺少的重要组成部分,它出现于20世纪50年代末,至今已有30余年。操作系统课程是有关计算机科学技术专业的一门专业基础课,因此,编写一本适用的操作系统教科书是十分需要的。20世纪70年代以来,许多操作系统教科书相继问世,其中《操作系统教程》(作者:孙钟秀、谭耀铭、费翔林、谢立、衣文国)于1989年由高等教育出版社出版。该书出版后得到广大读者的欢迎和支持,许多学校选择该书作为教材或主要参考书。在1992年第二届全国高等学校优秀教材评选中《操作系统教程》被评为国家级优秀教材。

随着计算机科学技术的迅速发展,计算机应用的日益广泛,操作系统的概念、新技术不断出现,为了适应这种发展的需要,必须对原教材进行更新。根据计算机科学技术的新发展和广大读者的反馈信息,我们对《操作系统教程》的内容作了适当修改、补充和调整,编写了这本《操作系统教程》(第二版)。

《操作系统教程》(第二版)保持了原教材的编写特点,力求做到:概念清晰,观点较高;深入浅出,便于自学;内容精选,取舍得当;难点分散,体系合理。全书着重讲解操作系统的基本原理和概念,以及设计方法和技巧,共分九章。第一章简述了操作系统的形成和发展历史;第二章至第六章叙述了操作系统的基本功能:处理器管理、存储管理、文件管理、设备管理和作业管理;第七章讨论了进程的互斥、同步、通信和死锁;第八章和第九章分别介绍了当前流行的UNIX操作系统和MS-DOS操作系统,希望能使读者有一个操作系统的整体印象。

参加本书修订工作的有:孙钟秀、谭耀铭、费翔林、谢立。

限于编著者的水平,错误与不妥之处定然难免,恳请读者批评和指正。

编著者

1994年12月于南京

第1版前言

操作系统是计算机系统的一个重要组成部分,它出现在20世纪50年代末,至今已有近30年。自20世纪70年代以来,许多操作系统教科书相继问世。但是,随着计算机科学技术的迅速发展和计算机应用的不断深入,操作系统的概念、新技术不断出现。为了适应这种发展的形势,迫切需要新的操作系统教材,《操作系统教程》就是为此目的而编写的。

本书是参照原教育部1983年颁布的计算机软件专业操作系统教学大纲,结合编者多年积累的教学经验和科研成果,吸收国内外近几年来的最新成就编写的。全书着重讲解操作系统的根本原理、概念、方法和技巧。力求做到:概念清晰,观点较高;深入浅出,便于自学;内容精选,取舍得当;难点分散,体系合理。

全书共分十章。第一章简述了操作系统的形成和发展的历史以及操作系统的类型;第二至第六章叙述操作系统的基本功能:处理器管理、存储管理、文件管理、设备管理和作业管理;第七章进程管理,讨论进程的互斥、同步、通信和死锁;第八章操作系统结构,介绍各种结构的设计方法;第九章和第十章介绍当前国内外较流行的几个操作系统,希望能给读者一个完整的操作系统的印象。其中,第九章介绍基于大、中型计算机的操作系统VM/SP,第十章介绍基于微型、小型计算机的操作系统UNIX、CP/M、MP/M和PC-DOS。

限于编者水平,错误与不妥之处定然难免,恳请读者批评指正。

本书承蒙吉林大学周长林副教授审阅,并提出了许多宝贵意见。在此表示衷心的感谢。

编著者

1987年2月

作者简介
孙冲秀



南京大学计算机系教授、博士生导师，中国科学院院士，计算机软件新技术国家重点实验室学术委员会主任。1935年生，毕业于南京大学数学系，1985年—1987年在英国进修，1979年—1981年在美国做访问学者。1986年被授予“国家级有突出贡献的中青年专家”称号。主要从事分布计算和操作系统方面的研究，主持研制我国国产系列计算机DJS200系列的DJS220和DJS210机操作系统。1982年，在国内率先研制成功ZCZ分布式微型计算机系统，以后又围绕分布式计算机系统和分布式系统软件开展研究工作，如提出和实现了基于CSP和Modula-2的分布式程序设计语言CSM以及面向对象程序设计语言CLUSTER86；设计和实现了异构型分布式操作系统ZGL；提出了一种分布式同步算法——令牌算法。他曾先后承担和主持国家“六五”、“七五”、“八五”、“九五”和“863”等科研项目20余项，获国家级科技进步二等奖一次、三等奖两次及部省级奖十多项，发表学术论文170余篇，著书5部。

目 录

第一章 操作系统概论	1	2.1 中央处理器	62
1.1 操作系统概观	1	2.1.1 处理器	62
1.1.1 操作系统的定义和目标	1	2.1.2 程序状态字	67
1.1.2 操作系统的资源管理技术	3	2.2 中断技术	68
1.1.3 操作系统的作用与功能	11	2.2.1 中断概念	68
1.1.4 操作系统的主要特性	14	2.2.2 中断源分类	69
1.2 操作系统的形成与发展	16	2.2.3 中断和异常的响应及服务	72
1.2.1 人工操作阶段	16	2.2.4 中断事件处理	74
1.2.2 管理程序阶段	17	2.2.5 中断优先级和多重中断	79
1.2.3 多道程序设计与操作系统的形成	18	2.2.6 Linux 中断处理	81
1.2.4 操作系统的发展与分类	21	2.2.7 Windows 2003 中断处理	86
1.3 操作系统的基本服务和用户接口	27	2.3 进程及其实现	91
1.3.1 基本服务和用户接口	27	2.3.1 进程的定义和属性	91
1.3.2 程序接口与系统调用	28	2.3.2 进程的状态和转换	93
1.3.3 作业接口与操作命令	33	2.3.3 进程的描述和组成	96
1.4 操作系统结构和运行模型	35	2.3.4 进程切换与模式切换	99
1.4.1 操作系统的构件和结构	36	2.3.5 进程的控制和管理	103
1.4.2 操作系统的运行模型	42	2.4 线程及其实现	106
1.4.3 Windows 2003 客户—服务器结构	44	2.4.1 引入多线程的动机	106
1.5 流行操作系统简介	47	2.4.2 多线程环境中的进程与线程	106
1.5.1 Windows 操作系统	47	2.4.3 线程的实现	108
1.5.2 UNIX 操作系统家族	48	2.5 Linux 进程与线程	110
1.5.3 自由软件和 Linux 操作系统	50	2.6 Windows 2003 进程与线程	113
1.5.4 IBM 系列操作系统	51	2.7 处理器调度	118
1.5.5 其他流行操作系统	54	2.7.1 处理器调度的层次	119
1.6 本章小结	55	2.7.2 选择调度算法的原则	121
习题一	57	2.7.3 作业和进程的关系	122
第二章 处理器管理	62	2.7.4 作业的管理与调度	123
		2.8 处理器调度算法	125
		2.8.1 低级调度的功能和类型	125
		2.8.2 作业调度和低级调度算法	126
		2.8.3 实时调度算法	132
		2.8.4 多处理机调度算法	134

II 目 录

2.9 Linux 调度算法	136	3.6.1 死锁产生	198
2.9.1 Linux 传统调度算法	136	3.6.2 死锁防止	199
2.9.2 Linux 2.6 调度算法	139	3.6.3 死锁避免	200
2.10 Windows 2003 调度算法	144	3.6.4 死锁检测和解除	206
2.11 本章小结	150	3.7 Linux 同步机制和通信机制	209
习题二	151	3.7.1 Linux 内核同步机制	209
第三章 同步、通信与死锁	163	3.7.2 System V IPC 机制	212
3.1 并发进程	163	3.8 Windows 2003 同步机制和通信 机制	214
3.1.1 顺序程序设计	163	3.9 本章小结	216
3.1.2 进程的并发性	164	习题三	217
3.1.3 进程的交互:协作和竞争	167		
3.2 临界区管理	168	第四章 存储管理	232
3.2.1 互斥和临界区	168	4.1 存储器	233
3.2.2 临界区管理的尝试	169	4.1.1 存储器的层次	233
3.2.3 实现临界区管理的软件算法	170	4.1.2 地址转换与存储保护	234
3.2.4 实现临界区管理的硬件设施	171	4.2 连续存储空间管理	237
3.3 信号量与 PV 操作	173	4.2.1 固定分区存储管理	237
3.3.1 同步和同步机制	173	4.2.2 可变分区存储管理	238
3.3.2 信号量与 PV 操作	174	4.2.3 伙伴系统	241
3.3.3 信号量实现互斥	177	4.2.4 主存不足的存储管理技术	244
3.3.4 信号量解决 5 位哲学家吃通心面 问题	177	4.3 分页存储管理	246
3.3.5 信号量解决生产者 - 消费者 问题	178	4.3.1 分页存储管理的基本原理	246
3.3.6 信号量解决读者 - 写者问题	180	4.3.2 快表	248
3.3.7 信号量解决理发师问题	181	4.3.3 分页存储空间的分配和去配	249
3.4 管程	182	4.3.4 分页存储空间的页面共享和 保护	250
3.4.1 管程和条件变量	182	4.3.5 多级页表	252
3.4.2 管程的实现	185	4.3.6 反量页表	253
3.4.3 使用管程解决进程同步问题	187	4.4 分段存储管理	254
3.5 进程通信	190	4.4.1 程序的分段结构	254
3.5.1 信号通信机制	190	4.4.2 分段存储管理的基本原理	255
3.5.2 管道通信机制	193	4.4.3 段的共享和保护	256
3.5.3 共享主存通信机制	194	4.4.4 分段和分页的比较	256
3.5.4 消息传递通信机制	195	4.5 虚拟存储管理	257
3.6 死锁	198	4.5.1 虚拟存储器的概念	257
		4.5.2 请求分页虚拟存储管理	258

4.5.3 请求分段虚拟存储管理 ······	275	5.4.4 缓冲区高速缓存 ······	324
4.5.4 请求段页式虚拟存储管理 ······	276	5.5 驱动调度技术 ······	324
4.6 Intel x86 分段和分页存储 结构 ······	277	5.5.1 存储设备的物理结构 ······	325
4.7 Linux 虚拟存储管理 ······	280	5.5.2 循环排序 ······	325
4.7.1 Linux 虚拟存储管理概述 ······	280	5.5.3 优化分布 ······	326
4.7.2 存储管理数据结构 ······	281	5.5.4 搜索定位 ······	327
4.7.3 主存页框调度 ······	286	5.5.5 独立磁盘冗余阵列 ······	330
4.7.4 进程虚存空间映射 ······	288	5.5.6 提高磁盘 I/O 速度的方法 ······	331
4.7.5 缺页异常处理 ······	289	5.6 设备分配 ······	332
4.8 Windows 2003 虚拟存储管理 ······	290	5.6.1 设备独立性 ······	332
4.8.1 主存管理的功能和地址空 间布局 ······	290	5.6.2 设备分配和设备分配数据 结构 ······	332
4.8.2 进程主存空间分配 ······	291	5.7 虚拟设备 ······	334
4.8.3 主存管理的实现 ······	294	5.7.1 问题的提出 ······	334
4.9 本章小结 ······	300	5.7.2 SPOOLing 的设计与实现 ······	334
习题四 ······	302	5.7.3 SPOOLing 应用 ······	336
第五章 设备管理 ······	310	5.8 Linux 设备管理 ······	337
5.1 I/O 硬件原理 ······	310	5.8.1 设备管理概述 ······	337
5.1.1 I/O 系统 ······	310	5.8.2 设备驱动程序 ······	337
5.1.2 I/O 控制方式 ······	311	5.8.3 设备 I/O 的处理 ······	338
5.1.3 设备控制器 ······	313	5.9 Windows 2003 I/O 系统 ······	340
5.2 I/O 软件原理 ······	314	5.9.1 I/O 系统结构和组件 ······	340
5.2.1 I/O 软件的设计目标和原则 ······	314	5.9.2 I/O 系统数据结构 ······	342
5.2.2 I/O 中断处理程序 ······	315	5.9.3 I/O 类型和处理 ······	344
5.2.3 I/O 设备驱动程序 ······	315	5.9.4 高速缓存管理 ······	349
5.2.4 独立于设备的 I/O 软件 ······	316	5.10 本章小结 ······	352
5.2.5 用户空间的 I/O 软件 ······	318	习题五 ······	353
5.3 具有通道的 I/O 系统 ······	319	第六章 文件管理 ······	358
5.3.1 通道命令和通道程序 ······	319	6.1 文件 ······	358
5.3.2 I/O 指令和主机 I/O 程序 ······	321	6.1.1 文件概念 ······	358
5.3.3 通道启动和 I/O 操作过程 ······	321	6.1.2 文件命名 ······	359
5.4 缓冲技术 ······	322	6.1.3 文件类型 ······	359
5.4.1 单缓冲 ······	322	6.1.4 文件属性 ······	360
5.4.2 双缓冲 ······	323	6.1.5 文件存取方法 ······	361
5.4.3 多缓冲 ······	323	6.2 文件目录 ······	362
		6.2.1 文件控制块、文件目录与目录	

IV 目 录

文件	362	7.3.1 安全模型概述	428
6.2.2 层次目录结构	363	7.3.2 安全模型示例	430
6.2.3 文件目录的检索	365	7.4 安全机制	432
6.3 文件组织与数据存储	366	7.4.1 硬件安全机制	432
6.3.1 文件的存储	366	7.4.2 认证机制	437
6.3.2 文件的逻辑结构	366	7.4.3 授权机制	439
6.3.3 文件的物理结构	370	7.4.4 加密机制	448
6.4 文件系统其他功能的实现	374	7.4.5 审计机制	451
6.4.1 文件系统调用的实现	374	7.5 安全操作系统设计和开发	453
6.4.2 文件共享	380	7.5.1 安全操作系统的结构和设计	
6.4.3 文件空间管理	384	原则	453
6.4.4 主存映射文件	386	7.5.2 安全操作系统的开发	455
6.4.5 虚拟文件系统	388	信息系统安全评价标准简介	460
6.5 Linux 文件系统	389	7.6 Linux 安全机制	462
6.5.1 Linux 虚拟文件系统	389	7.7 Windows 2003 安全机制	466
6.5.2 文件系统的注册与注销及安装与 卸载	400	7.7.1 安全性组件和安全登录	466
6.5.3 文件系统的缓存机制	401	7.7.2 访问控制	467
6.5.4 Ext2 文件系统	403	7.7.3 安全审计	471
6.6 Windows 2003 文件系统	404	7.7.4 加密文件系统	472
6.6.1 文件系统概述	404	7.8 本章小结	473
6.6.2 NTFS 在磁盘上的结构	405	习题七	474
6.6.3 文件系统模型和 FSD 体系 结构	408		
6.6.4 NTFS 可恢复性支持	411		
6.6.5 NTFS 安全性支持	413		
6.7 本章小结	414		
习题六	414		

第七章 操作系统的安全与 保护

7.1 安全性概述	420
7.2 安全策略	421
7.2.1 安全需求和安全策略	421
7.2.2 访问支持策略	423
7.2.3 访问控制策略	426
7.3 安全模型	428

第八章 网络和分布式操 作

系统	477
8.1 计算机网络概述	477
8.1.1 计算机网络的概念	477
8.1.2 网络体系结构	479
8.2 网络操作系统	481
8.2.1 网络操作系统概述	481
8.2.2 网络操作系统实例	482
8.3 分布式操作系统	484
8.3.1 分布式系统概述	484
8.3.2 分布式进程通信	485
8.3.3 分布式资源管理	491
8.3.4 分布式进程同步	492
8.3.5 分布式系统中的死锁	498
8.3.6 分布式文件系统	500

8.3.7 分布式进程迁移 ······	502	8.6 本章小结 ······	506
8.4 Linux 网络体系结构 ······	505	习题八 ······	507
8.5 Windows 2003 网络体系结构和 网络服务 ······	506	参考文献 ······	510

第一章



操作系统概论

1.1 操作系统概观

1.1.1 操作系统的定义和目标

操作系统(Operating System, OS)的出现、应用和发展是近半个世纪来计算机软件所取得的一个重大进展,它经历了从无到有、从小到大、从简单到复杂、从原始到先进的发展历程,随之产生许多相关的基本概念、重要理论和核心技术。尽管尚无严格定义,但一般认为操作系统是:管理系统资源、控制程序执行、改善人机界面、提供各种服务,并合理组织计算机工作流程和为用户方便而有效地使用计算机提供良好运行环境的最基本的系统软件。

计算机发展至今,无论个人计算机还是巨型计算机,都无一例外地配置一种或多种操作系统,它已经成为现代计算机系统不可分割的重要组成部分,为人们建立各种应用环境奠定了坚实的基础。设计与开发操作系统的目的是让用户更有效、更方便地使用计算机资源,其基本任务是:创建可供用户使用的抽象资源,管理这些资源的并发使用,为应用程序提供良好的运行环境。操作系统的主目标可归结为以下几点。

(1) 方便用户使用

操作系统通过提供用户与计算机之间的友好界面来方便用户使用。

(2) 扩充机器功能

操作系统通过扩充硬件功能为用户提供各种服务。

(3) 管理各类资源

操作系统有效地管理系统中的所有软硬件资源,使之得到充分的利用。

(4) 提高系统效率

操作系统合理地组织计算机的工作流程,改进系统性能,提高系统效率。

(5) 构筑开放环境

操作系统遵循国际标准来设计和构造一个开放环境,其含义是指:遵循相关的国际工业标准和开放系统标准(如 POSIX);支持体系结构的可伸缩性和可扩展性;支持应用程序在不同平台上的可移植性和互操作性。

计算机系统由硬件和软件两个部分组成,是硬件和软件相互交织形成的集合体,构成一个解决计算问题的工具。硬件是软件运行的物质基础,软件能够充分地发挥硬件的潜能并扩充硬件的功能,完成各种应用任务,两者互相促进,相辅相成,缺一不可。如图 1.1 所示是一个计算机系统的软硬件层次结构,其中,每层都具有一组功能并对外提供相应的接口,接口对层内隐蔽实现细节,对层外提供使用约定。

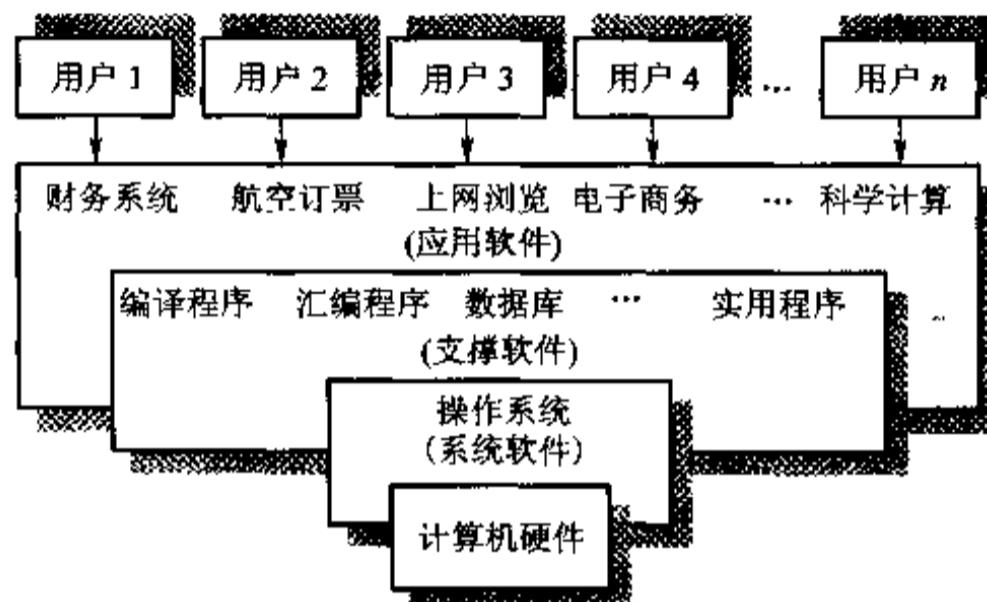


图 1.1 计算机系统的软硬件层次结构

硬件层提供基本的可计算性资源,包括处理器、寄存器、存储器及各种 I/O 设备,这些可计算性资源组成计算机系统的硬件,它们按照用户需求接收和存储信息、处理数据并输出运算结果,是操作系统发挥效能的基础。

操作系统层是最靠近硬件的软件层,负责管理和控制计算机硬件并对其作首次扩充和改造,主要做好资源的调度与分配、信息的存取与保护、并发活动的协调与控制等工作,把上层的支撑软件和应用软件与计算机硬件隔离开来,为其运行提供良好的基础和强有力的支持。

支撑软件层的工作基础建立在被操作系统扩充功能的机器上,利用系统所提供的扩展指令集,可以较容易地实现编译程序、汇编程序、语言处理程序、窗口系统、Internet 浏览器、数据库管理系统和其他实用程序,支持应用软件的开发和运行。支撑软件通常可归入系统软件一类,但它并不属于操作系统的组成部分。

应用软件层解决用户特定的或不同应用所需要的信息处理问题,任何计算机系统的价值都要通过应用软件的价值来评定和体现,用户所看到的是应用软件,其他软硬件只是应用软件运行的支撑部分。应用程序开发者借助于程序设计语言来表达应用问题,既快捷又方便,而最终的应用程序则在操作系统的控制下,在计算机系统中得以正确运行。

操作系统与支撑软件及应用软件之间的主要区别如下:虽然它们都是程序,但其意图不同,

操作系统有权分配资源,而其他程序只能使用资源,两者之间是控制与被控制的关系;操作系统直接作用于硬件之上,隔离其他上层软件,并为其提供接口和服务,所以,操作系统是软件系统的核心,是各种软件的基础运行平台,而支撑软件及应用软件在这方面的差别并不严格;通用操作系统对共性功能提供支持,与硬件相关但同应用领域无关,所以,它可以支持很多应用领域;支撑软件及应用软件只能通过操作系统来使用计算机系统的物理资源;操作系统实现资源管理机制,允许应用程序提供资源管理策略。

需要注意的是,在传统的计算机系统中,操作系统是指运行在核心态的、受硬件保护的软件,而在用户空间所运行的 shell、window 等模块虽然能够实现系统功能,但并不是操作系统的组成部分。随着客户-服务器结构操作系统的出现,传统上被认为是操作系统核心的组件,如文件系统、主存管理、设备管理等,都被移至用户空间运行,在这类系统中就很难划分明显的界限,在核心态所运行的代码都属于操作系统,而在用户态实现操作系统关键功能的软件也应该看做操作系统的一部分,至少同操作系统密切相关的。

1.1.2 操作系统的资源管理技术

1. 资源管理

现代计算机系统都包含各种各样的物理部件、设备和软件等资源,操作系统的主要任务之一就是对资源进行管理,在相互竞争的应用程序之间有序地控制软硬件资源的分配、使用和回收,使资源能够在多个程序之间共享。首先,由于物理资源有限,而竞争使用资源的应用程序众多,必须很好地解决资源数量不足和合理分配资源这两个问题。其次,由于物理资源在硬件实现上的复杂性,用户几乎不可能直接使用它们,要实现资源的易用性,只能借助系统所提供的功能、手段和设施来控制与使用。从更高的层次来看,操作系统将物理计算机的功能加以扩展,使之成为接口好、功能强、效率高、易使用的计算机系统,而这只是概念和逻辑上的,不是真实的、物理上的,称为虚拟机或虚机器。每个应用程序运行在自己的虚拟机上,在一台虚拟机上运行的程序称为“进程”。第二章将引入并详细讨论进程的概念,进程是贯穿于操作系统设计、实现和使用中的最重要的概念之一。操作系统通过共享硬件资源的方式来实现虚拟机抽象,所利用的资源管理技术如下。

(1) 资源复用

多道程序设计是现代操作系统所采用的基本技术,系统中相应地有多个进程竞争使用资源。由于计算机系统的物理资源是宝贵和稀有的,操作系统让众多进程共享物理资源,这种共享称为资源复用。通过适当的复用可以创建虚拟资源和虚拟机,以解决物理资源数量不足的问题。物理资源的复用有两种基本方法:空分复用共享和时分复用共享。

① 空分复用共享(space-multiplexed sharing)表明资源可以进一步分割成更多和更小的单位供进程使用。在计算机系统中,进程能够空分复用那些满足以下属性的物理资源,即能够将资源的不同单位同时分配给不同的进程。主存和辅助存储器(磁盘)资源是空分复用共享的例子。至于资源如何被空分复用,如果资源不够分配,或进程要求动态地申请更多的资源该如何应对,则都是操作系统所要完成的任务。

② 时分复用共享(time-multiplexed sharing)并非将资源进一步分割成更小的单位,这类资源也不能被分割成更小的单位,反之,进程可以在一个时间片内以独占方式使用整个物理资源,其他进程则在另外的时间片内使用此资源。至于资源如何实现时分复用,下一个进程如何被分配资源及其占用资源的时间,都是由操作系统控制的。计算机系统中,在某一段时间内,进程对分配所得计算机资源享有独占控制权,当时间片用完后,物理资源被释放出来并分配给其他进程使用。磁带机和处理器均采用时分复用共享技术管理。

进程能够空分复用主存资源,不同的进程映像装入不同的主存区域,拥有各自的地址空间,且通过硬件存储保护机制进行隔离。操作系统必须跟踪当前执行进程,确定其执行时间;而时分复用共享使得处理器可以执行已装入不同地址空间中的程序代码。这种共享硬件的技术十分重要,称为多道程序设计。例如,有两道程序运行在各自的虚拟机上,虚拟机空分复用主存,分别装入主存中不重叠的区域;虚拟机时分复用处理器,两道程序各自执行自己的任务,一个进程正在计算,另一个进程正在读盘。

(2) 资源虚化

虚化又称虚拟性,是指操作系统中的一类有效的资源管理技术,能进一步地提高操作系统为用户服务的能力和水平。虚化的本质是对资源进行转化、模拟或整合,把一个物理资源转变成逻辑上的多个对应物,创建无须共享的多个独占资源的假象,以达到多用户共享一套计算机物理资源的目的。空分复用与虚化两者相比较,空分复用所分割的是实际存在的物理资源,而虚化则实现假想的虚拟同类资源,采用虚化技术不但可以解决某类物理资源数量不足的问题,而且能够为应用程序提供更易于使用、高效的虚拟资源并创建更好的运行环境。例如,使用虚化技术创建易用且多于实际物理资源的最佳例子是基于物理主存的虚拟主存,其实现方法是:只要某个程序的运行空间超出可用物理主存空间的大小,操作系统便在主存和磁盘之间自动地传送与当前计算有关的程序段或数据段。虚化技术可用于外部设备,应用程序把组织成文件形式的输出信息写至虚拟打印机,而不是直接与物理打印机交互,当输出信息全部汇集后,才被传送到物理打印机上打印,这样就不会发生进程阻塞。这种虚化技术称为外部设备同时联机操作(Simultaneous Peripheral Operations On Line,SPOOLing),它将物理上的一台独占设备转化成逻辑上的多台虚拟独占设备,多个进程可以并行“打印”,因为每个进程都有自己的虚拟打印机。虚化技术也可用于文件系统,这就是虚拟文件系统(Virtual File System,VFS),使得在一个操作系统的控制下可以同时支持多种具体的文件系统。虚化技术的应用范围很广,如通过窗口技术可以把一个物理屏幕(终端)虚化成逻辑上的多个虚拟屏幕(终端);通过信道多路复用技术可以把一条物理信道改造成多条虚拟信道,由于虚拟资源可以有很多,于是便可分配给不同的进程使用。从本质上来看,SPOOLing技术、窗口技术、时分信道多路复用技术都建立在时分复用共享的基础上;虚拟存储器则是通过虚拟存储技术把物理上的多级存储器(主存和辅助存储器)映射为逻辑上的、单一的(虚拟)存储器,它与频分信道多路复用技术一样都是建立在空分复用共享基础之上的例子。

(3) 资源抽象

资源复用和资源虚化的主要目标是解决物理资源数量不足的问题,资源抽象则用于处理系

统的复杂性,重点解决资源的易用性。为了解决这一难题,操作系统对物理资源进行抽象。任何一种特定的硬件资源,如磁盘,都有一个接口和一组复杂的操作,其中定义程序员如何使用此接口和相应的操作来完成对资源的访问,抽象接口会比硬件接口简单得多。资源抽象是指通过创建软件来屏蔽硬件资源的物理特性和接口细节,简化对硬件资源的操作、控制和使用,即不考虑物理细节而对资源执行操作。例如,面向进程而不是处理器,面向文件而不是磁盘,而向窗口而不是屏幕,面向虚拟机而不是物理计算机。资源抽象软件对内封装实现细节,对外提供应用接口,这意味着用户不必了解更多的硬件知识,不必处理乏味的细节,只通过软件接口即可使用和操作物理资源,使得用户不必考虑低层细节和物理特性,让程序员集中精力编写高质量的代码以解决特定的应用问题。资源抽象要做得尽可能简单,良好的抽象不但会使用户十分容易理解和使用,又能够为使用低层硬件提供强有力的支持。

考察资源抽象的例子。为了从某个物理设备中输出一组字符,首先需要知道其硬件接口,包括控制寄存器、状态寄存器和数据寄存器,反复读取状态寄存器的信息,直到设备对于接收下一批字符准备就绪为止。接着,将字符值写入数据寄存器中,并发送“输出”命令至控制寄存器以控制输出。现在为某一类设备创建并实现一个设备驱动程序,它隐蔽物理设备处理 I/O 操作的细节,此设备驱动程序就是物理设备执行 I/O 操作的软件抽象;再定义一个标准化的软件接口(即系统调用),应用程序只要使用系统调用及指定输出的字符,由系统调用来调用设备驱动程序,便可从硬件设备上得到输出信息。这里的设备驱动程序是对此类设备的一种抽象。由于设备资源抽象能够隐蔽执行 I/O 操作的过程,应用程序就不必关心发送命令和加载数据至控制寄存器和数据寄存器的细节,仅通过调用设备驱动程序便可获得输出结果。

抽象技术也可用于定义和构造多层软件,每层软件都隐蔽下一层的实现细节,从而形成多级资源抽象。以磁盘设备为例,把信息块从主存写入磁盘需要执行下列机器指令:

load(block,length,device)	/* 把指定长度的信息块复制到磁盘缓冲存储器 */
seek(device,track)	/* 移动磁头至指定的磁道 */
out(device,sector)	/* 将数据写入指定的扇区 */

对于一个简单的抽象,可通过系统调用 write() 来打包上述指令:

```
void write(char * block,int length,int device,int track, int sector) {
    load(block,length,device);
    seek(device,track);
    out(device,sector);
}
```

一种更高层次的抽象是由操作系统提供文件系统,借助于 C 语言的 stdio 库,通过文件名来读写磁盘,于是写磁盘的操作如下:

```
int fprintf(fileID,"%s",datum); /* 从文件头隐含的偏移位置开始写数据至磁盘文件 */
...
write();
```

...

系统调用 write() 对机器指令进行抽象, 库函数 fprintf() 对 write() 做进一步的抽象, 抽象的层次越高, 使用起来越方便。

数据输入/输出也依赖于多层抽象。程序员使用文件类系统调用来使用文件, 而不必了解实现过程中所执行的成千上万条机器指令, 在实际执行 I/O 操作时, 文件类系统调用又要调用外部设备的抽象——设备驱动程序, 以完成信息的输入/输出, 这里也用到多层抽象。抽象的例子比比皆是, 例如, 用户通过输入操作命令要求运行指定的程序、查询系统的状态、申请所需的资源等, 都会涉及低层硬件操作, 但是, 由于已经实施一系列抽象, 从用户的角度来看, 操作系统就是一台以这些命令作为机器语言的虚拟计算机。

(4) 组合使用抽象和虚化技术

对于某一类资源, 操作系统往往同时实施抽象和虚化技术。例如, 为打印机既配置“打印函数”(设备驱动程序), 又应用虚拟设备, 通过打印函数的抽象来隐蔽打印细节, 实施 SPOOLing 虚化“扩充”物理打印机的数量, 每个用户都可以得到使用方便的虚拟打印机。又如, 窗口软件是对物理终端的抽象和虚化, 能够为用户提供虚拟终端和便捷的 I/O 服务, 应用程序可以用 scanf() 和 printf() 函数读写窗口, 好像此窗口是一个完整的终端设备, 完全不必涉及屏幕定位的细节, 窗口软件对虚拟终端的操作进行映射, 使其对应于屏幕的特定物理区域, 将应用软件对虚拟终端的操作转换成物理终端上的相应操作。通过多窗口技术一台物理终端可以支持多个虚拟终端。

至此, 已讨论三种基本的资源管理技术: 资源复用、资源虚化和资源抽象, 在操作系统的设计、实现和使用中自始至终贯穿这些技术的应用, 采用这些资源管理技术的目标之一是解决资源数量不足和易于使用问题。

2. 操作系统中的基础抽象——进程、虚存和文件

计算机系统的物理资源可被分成两大类: 计算类、存储及接口类。前者有处理器和主存; 后者有辅助存储器和外部设备等。为了方便对物理资源的管理和使用, 现代操作系统对资源进行三种最基础的抽象——进程抽象、虚存抽象和文件抽象。

(1) 进程抽象

进程是对于进入主存的当前运行程序在处理器上操作的状态集的一个抽象, 它是并发和并行操作的基础。从概念上讲, 每个进程都是一个自治执行单元, 执行时需要使用计算机资源, 至少需要处理器(包括程序计数器、通用寄存器、堆栈指针寄存器和其他寄存器)和主存。实际上, 若干进程透明地时分复用共享一个(或多个)处理器, 操作系统内核的主要任务之一是将处理器“虚化”, 制造一种每个运行进程都独自拥有一个处理器的假象。由于进程的执行依赖于主存和设备上的信息资源, 所以还需要以下几种资源抽象。

(2) 虚存抽象

操作系统对计算类资源的管理方式与对其他资源的管理方式不一样, 在创建进程时, 隐含着对处理器和主存资源的双重需求, 所以需要管理一个特殊的抽象资源——虚拟存储器。物理主

存被抽象成虚拟主存，给每个进程造成一种假象，认为它正在独占和使用整个主存。进程可以获得一个硕大的连续地址空间，其中存放着可执行程序和数据，可以使用虚拟地址来引用物理主存单元；而且进程的虚拟主存空间彼此隔离，具有很好的安全性。虚拟存储器是通过结合对主存和磁盘的管理来实现的，把一个进程的虚拟主存中的内容存储在磁盘上，然后用主存作为磁盘的高速缓存，以此为用户提供比物理主存大得多的虚拟主存空间。

结合(1)和(2)的讨论可见，仅有进程抽象和虚存抽象还是不够的，进程的执行依赖于存放在主存中的程序和数据，而它们又被存储在设备上，所以还要对设备进行抽象。

(3) 文件抽象

磁盘、磁带、光盘等存储设备都有极其复杂的物理接口，为了便于操作和使用，通常将其抽象使得所存放的信息可以表示为一个命名的逻辑字节流，称其为文件，这是资源抽象的一个特例。简单地说，文件是磁盘等设备的抽象，通过将文件中的字节映射到存储设备的物理块中来实现文件抽象。在概念上，“从文件中读取数据块”比操纵“查找柱面、搜索磁道、读出扇区”之类的细节简单得多，至于“检测状态、复位校正、启动电机”等底层动作则更不应该出现在提供给用户的抽象描述中。程序员通过创建、打开、读写和关闭操作来控制文件，而磁盘的所有I/O操作细节都对用户隐藏起来。这种抽象对于现代操作系统至关重要。文件抽象对于信息的存储、检索、更新、共享和保护会带来很多好处。

进一步进行分析，文件抽象也是操作系统对磁盘设备进行多层次抽象的结果。

① 第一层抽象：从磁盘到分区

一个物理磁盘可以被划分成多个分区，每个分区在逻辑上可以看做一个独立的磁盘，可以安装和驻留一个文件系统。

② 第二层抽象：从分区到扇区

磁盘由柱面号、磁道号和扇区号来定位，扇区是磁盘的基本存储单元。例如，每个扇区存储1 KB信息，可由外而内一个柱面接一个柱面、一个磁道接一个磁道地给每个扇区编号，一个对磁盘扇区进行编号的系统使得磁盘变为一系列扇区的集合。

③ 第三层抽象：从扇区到簇

不同磁盘的扇区大小可能不同，系统软件屏蔽这一事实并向高层软件提供统一尺寸的数据块，将若干扇区合并为一个逻辑块，称为簇。再对簇进行编号，这样高层软件只和大小相同的簇交互，而不必顾及物理扇区的尺寸。

④ 第四层抽象：从簇到文件系统分区

操作系统将簇序列分成超级块、inode区和数据块区等，再结合组织、控制和管理信息的软件便形成文件和文件系统。

事实上，操作系统为了管理方便，除了处理器和主存之外，将磁盘和其他外部设备资源都抽象为文件，如软盘文件、磁带文件、光盘文件、打印机文件等，这些设备均在文件的概念下统一管理，不但减少系统管理的开销，而且使得应用程序对数据和设备的操作有一致的接口，可以执行同一套系统调用。

在此对三种资源抽象加以综述,操作系统担负两项基本任务:防止硬件资源被失控的应用程序滥用;屏蔽复杂的硬件操作细节,为应用程序提供使用硬件资源的简单而一致的方法。这两项任务是通过如图 1.2 所示的抽象来完成的。存储器是十分重要的,它遍及计算机系统的各个层

而,进程映像、I/O 数据、共享对象都要存放在主存中,至于编译程序、链接程序以及程序的运行则更不能缺少主存的支持。所以,操作系统中最基础和最重要的三种抽象是:文件抽象、虚存抽象和进程抽象。三种抽象之间存在一种包含关系,文件是对设备的抽象;虚存是对主存和设备的抽象;进程则是对处理器、主存和设备的抽象。

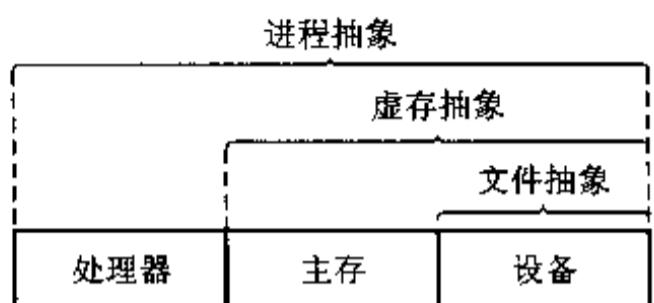


图 1.2 操作系统的基础抽象

核所提供的有限数目的原语或系统调用进行交互,操作系统在三种抽象的基础上能够很方便地控制程序的执行,调度并分配处理器资源。与进程抽象相关的所有工作称为进程管理。在此有必要指出,线程的概念是对进程抽象进一步深化的产物,线程是进程的组成部分,它只是将进程内部资源管理与控制执行相分离而得到的结果。

(4) 其他资源抽象

与磁盘抽象相似,操作系统还对其他低层硬件资源如中断、时钟、网络接口等进行抽象,向用户隐蔽其物理特性,在此不再一一剖析。值得注意的是,资源抽象也可用于没有特定基础硬件的软件资源,如消息、信号量和共享数据结构。对象和抽象数据类型是常用于创建抽象资源的软件机制。

3. 虚拟计算机

虚拟计算机是一台抽象计算机,它在硬件的基础上由软件来实现,并且与物理机器一样,具有指令集及可用的存储空间。如果某台机器上配有操作系统,对于用户来说,这就是一台以操作系统语言(系统调用)为机器语言的操作系统虚拟机;如果某台机器上配有操作系统,同时又安装了 C 语言,对于 C 语言的用户来说,这就是一台以 C 语言为机器语言的 C 语言虚拟机。随着因特网的快速发展和广泛应用,若干台计算机通过局域网或广域网连接起来,如果在结点计算机上配置网络操作系统,则它为用户提供可互相通信的一组虚拟机;如果在结点计算机上配置分布式操作系统,则它为用户提供一台具有分布计算能力的虚拟机。按此方法分类,在裸机的基础上,可以配置操作系统虚拟机、汇编语言虚拟机、高级语言虚拟机、数据库语言虚拟机、网络通信虚拟机和分布处理虚拟机等。下面来考察一个特殊的虚拟机——操作系统虚拟机。

(1) 虚拟计算机的虚拟处理器是由物理处理器实现的,利用处理器调度技术,虚拟处理器每次分得时间片后便可供进程执行时使用(时分复用)。

(2) 虚拟计算机的虚拟主存是主存利用虚拟存储技术而为其提供的,虚拟主存可供进程作为物理地址空间使用(空分复用)。

(3) 磁盘被抽象成命名文件,运行在虚拟机上的进程通过文件管理系统来使用和共享磁盘。

磁盘的部分空间可以作为主存空间的扩充,存放进程映像副本(空分复用)。

(4) 外部设备也被抽象成命名文件,运行在虚拟机上的进程通过设备管理来使用设备,根据外部设备自身物理特性的不同,有些类型的设备基于空分复用共享(如屏幕),有些类型的设备基于时分复用共享(如磁带、系统总线)。

综上所述,虚拟机是由操作系统通过共享硬件资源的方式来实现的,它定义进程运行的逻辑计算环境。从概念上来说,一个进程运行在一台虚拟机上,可以认为一个进程就是一台虚拟机,一台虚拟机就是一个进程。当然,实际上各进程之间是物理处理器在来回切换,由于每台虚拟机在同一时间段内只占用全部物理资源的一部分,因而可以创建许多台虚拟机,系统中也就允许有许多进程并发或并行执行。如图 1.3 所示是操作系统将物理计算机仿真成虚拟计算机的示意图。

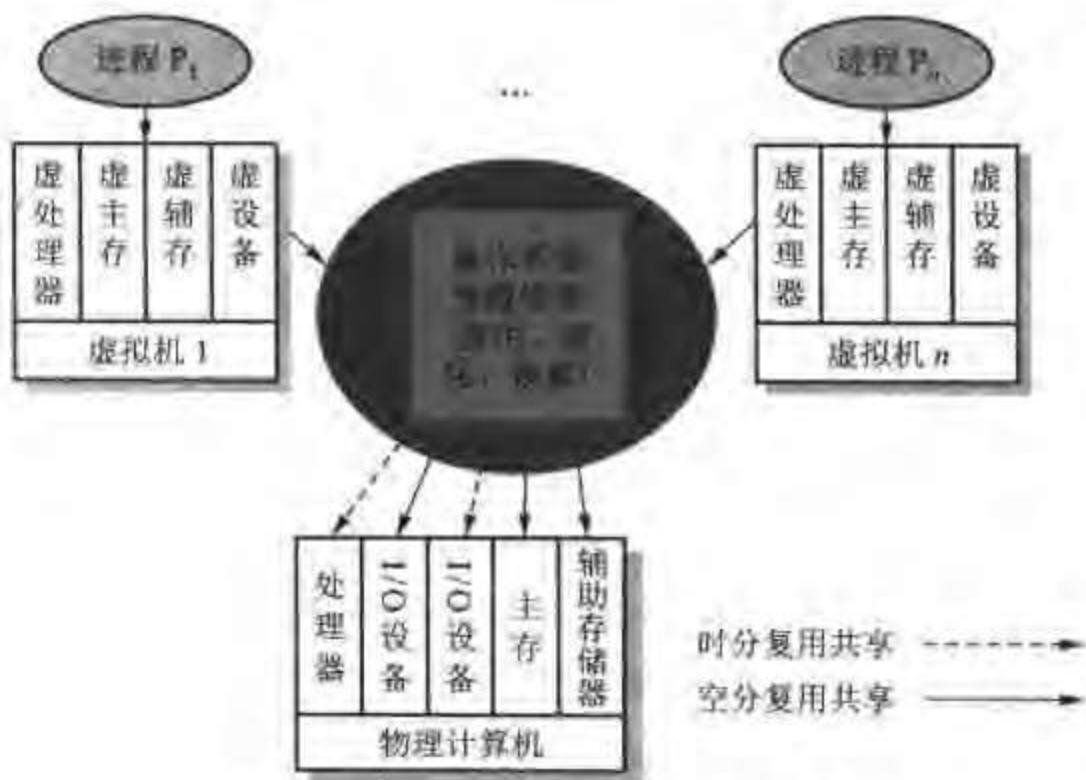


图 1.3 物理计算机仿真成虚拟计算机

在现代操作系统中,用户所使用的是虚拟计算机,它是由操作系统使用物理计算机仿真而来的计算机。与物理计算机类似,虚拟计算机有 4 个基本组成部分:虚处理器、虚主存(又称虚拟主存)、虚辅存(又称虚拟辅存)和虚设备(又称虚拟设备),每个虚拟资源都是物理资源通过复用或虚化而得到的产物。虚拟计算机与物理计算机在很多方面相像,主要差别在于:前者有很多台,后者只有一台;前者使用起来很方便,后者使用起来很困难。

(1) 虚处理器

对于用户而言,虚处理器与物理处理器的能力基本上是一样的,它们拥有同样的指令系统。虚处理器实质上是由系统直接利用物理处理器来实现的,并具有以下一些特点。

- ① 虚处理器没有中断,进程的设计者不需要有硬件中断的概念。

② 每个进程都有自己的虚处理器,用以实现多进程的并发执行。

③ 虚处理器为进程提供功能强大的指令系统,即由非特权指令和系统调用所组成的新指令系统集。

操作系统禁止某些特权指令在虚处理器上执行,其结果虽然会导致失去某些能力,但是可以带来虚拟机安全性高等好处。此外,操作系统还为虚处理器提供新功能,扩充虚处理器的指令集。进程通过使用系统调用可以做到:创建新进程(虚拟机),与其他进程(虚拟机)进行通信,申请资源,操作文件,执行 I/O 操作,等等。虚处理器是由操作系统通过时分复用和调度切换而实现的物理处理器的抽象,在某特定时刻仅有一个进程在一个物理处理器上真正运行,但在某时间段内多个进程都曾经运行过,虽然可能有许多进程分享同一个物理处理器,但是虚处理器造成假象,让这些进程感觉在独享物理处理器。

(2) 虚拟主存

虚拟计算机的主存和物理主存类似,都是从 0 开始的以连续数字命名的单元序列。操作系统分割物理主存,将其分配给虚拟机使用,同时各虚拟机所分得的主存空间相互隔离且互不干扰,虚拟主存让进程在获取和使用主存时感觉像拥有整个计算机的主存。虚拟存储器避免在主存和磁盘之间来回复制整个进程地址空间,相反地,当运行进程需要信息或信息被更新时,系统在主存和磁盘之间自动地传输当前计算所涉及的一小部分数据。对于程序员而言,这是完全透明的操作,且无须考虑物理主存的大小,编程十分方便。

(3) 虚拟辅存

辅助存储器(如磁盘)为信息提供持久性存储,通过空分复用的方式把辅助存储器空间分配给进程使用,有的空间用做主存的扩充,有的空间用于存放文件,信息以文件为单位存储在物理磁盘上,文件中的字节流映射到设备的物理块中,进程可以通过文件系统调用或映射文件输入/输出对文件信息进行存储、检索和处理。有的操作系统可以提供多个虚拟盘,按需分割物理磁盘的若干磁道,除了容量之外,其他各个方面与物理磁盘完全相同。

(4) 虚拟设备

SPOOLing 和文件系统为每台虚拟机提供虚拟读入机和虚拟打印机,分时用户的终端则提供虚拟机操作员控制台。虚拟机的 I/O 操作与物理计算机的 I/O 操作完全不同,物理设备的接口比较复杂,物理 I/O 操作需要了解设备的硬件特性并执行一系列低层操作。为此,操作系统为每类物理设备编写实现 I/O 操作的设备驱动程序以供应用程序调用,以此对物理设备进行抽象,屏蔽相关的实现细节,执行低层操作。进程执行 I/O 操作实质上是调用相应的设备驱动程序,既简单又方便。再次回顾虚拟打印机的工作过程,进程执行打印时把信息组织成文件形式输出至虚拟打印机的输出目录,输出信息全部汇集后,方由系统送到物理打印机进行打印,通过使用假脱机技术,多个进程都可以分配到虚拟打印机,“并行”执行打印,缩短进程等待时间,提高系统工作效率。

在 20 世纪 70 年代,IBM 公司在 VM/370 操作系统中把资源复用、虚化与抽象技术发展到前所未有的高度,它在裸机上应用这些技术,建立从完整的物理计算机精确地复制而来的多台虚

拟机。与其他虚拟机直接运行应用程序不同,每台 VM/370 虚拟机可以运行不同的操作系统,在不同的操作系统上运行各自的应用程序。

1.1.3 操作系统的作用与功能

操作系统在计算机系统中起到 3 个方面的作用。

(1) 操作系统作为用户接口和服务提供者

由于用户对计算机系统的需求与期望和现有硬件功能之间存在巨大的差距,因此需要操作系统来填补这种差距,操作系统处于用户和计算机硬件之间,用户通过操作系统来使用计算机系统。从内部来看,操作系统对计算机硬件进行改造和扩充,为应用程序提供强有力的支持。例如,提供原语和系统调用来扩展机器指令集,这些新的功能到目前为止还难于由硬件直接实现。从外部来看,操作系统提供友好的人机接口,使得用户能够方便、安全、高效地使用硬件和运行应用程序。此外,操作系统还能够合理地组织计算机的工作流程,协调各个机器部件有效地工作,为应用程序提供良好的运行环境。经过操作系统改造和扩充的计算机不但功能更强,使用起来也更为方便,用户通过“系统调用”来使用操作系统所提供的各种功能,而无须了解软硬件本身的细节。所以,操作系统是一个友善的用户接口和各种公共服务的提供者。

(2) 操作系统作为扩展机或虚拟机

人们在多年以前就认识到必须寻求某种方法把硬件的复杂性与用户隔离开来。经过不断地探索和研究,所采用的方法是在计算机裸机上安装软件来组成整个计算机系统,同时为用户提供易于理解和便于使用的接口。在操作系统中,隐蔽硬件细节并将其与用户隔离开来的情况随处可见,虚拟机就是一种例证,每当在计算机上添加一种软件,提供一种抽象,系统的功能便增加一点,使用起来就更方便一点,可用的运行环境就更加友好一点。所以,从这个角度来看,操作系统的作用是为用户提供比低层硬件的功能显著增强、使用更加方便、安全可靠性更好、效率明显提高的操作系统扩展机或虚拟机。

(3) 操作系统作为资源的管理者和控制者

在计算机系统中,分配给用户使用的各种软件和硬件设施统称为资源。资源主要包括两大类:硬件资源和信息资源。其中,硬件资源有处理器、存储器、外部设备等,外部设备又分为输入型设备、输出型设备和存储型设备;信息资源则可分为程序和数据等。为了使应用程序能够正常运转,操作系统必须对其分配足够的资源;为了提高系统效率,操作系统必须支持多道程序设计,合理调度和分配各种资源,充分发挥并行部件的性能,使各种部件和设备最大限度地执行操作和保持忙碌。

操作系统的重要任务之一是对资源进行抽象,找出各种资源的共性和个性,有序地管理计算机中的软硬件资源,记录资源的使用情况,确定资源分配策略,实施资源的分配和回收,满足用户对资源的需求。提供某种机制来协调应用程序对资源的使用冲突,研究资源利用的统一方法,为用户提供简单、有效的资源使用手段,在满足应用程序需求(如交互进程响应速度快、批进程周转时间短)的前提下,最大限度地实现各种资源的共享,提高资源利用率,从而提高计算机系统的

效率。

概括地说,操作系统既是“管理员”,又是“服务员”。

① 对内作为“管理员”,做好计算机系统软硬件资源的管理、控制与调度,提高系统效率和资源利用率。

② 对外作为“服务员”,是用户与硬件之间的接口和人机界面,为用户提供尽可能友善的运行环境和最佳服务,所以,资源管理和调度是操作系统的重要任务。

从资源管理的观点来看,操作系统具有 6 项主要功能。

(1) 处理器管理

处理器是计算机系统中最为稀有和宝贵的资源,应该最大限度地提高其利用率,可以采用多道程序设计技术,组织多个作业同时执行,解决处理器的调度、分配和回收等问题。随着多处理器系统的出现,处理器的管理就变得更加复杂。为了做好处理器的管理工作,描述多道程序的并发执行,操作系统引入进程的概念,处理器的分配、调度和执行都以进程作为基本单位。随着并行处理技术的发展,并发执行单位的粒度变细,并发执行的代价降低,在进程的基础上又引入线程的概念。对处理器的管理和调度最终归结为对进程和线程的管理和调度,包括:

- ① 进程控制和管理;
- ② 进程同步和互斥;
- ③ 进程通信;
- ④ 进程死锁;
- ⑤ 线程控制和管理;
- ⑥ 处理器调度,又可分为高级调度、中级调度和低级调度。

(2) 存储管理

存储管理的主要任务是管理主存资源,为多道程序运行提供有力的支撑,提高存储空间的利用率,其具体功能如下。

① 主存分配

根据应用程序的需要向其分配主存资源,这是多道程序并发执行的首要条件。在程序运行结束时,还需要回收主存资源。为此,操作系统必须记录主存的使用情况,处理用户所提出的存储资源申请,并实施存储资源的分配和回收。

② 地址转换与存储保护

负责把用逻辑地址编程的应用程序装入主存,并将逻辑地址转换成主存物理地址,同时保证各应用程序互不干扰,不允许它们访问操作系统存储区,保护系统和应用程序在主存中存放的信息不被破坏。

③ 主存共享

能够让主存(内存)中的多个应用程序实现存储共享,提高存储资源的利用率。

④ 存储扩充

由于受到处理器寻址能力的限制,一台计算机的物理主存容量总是有限的,难以满足大型应

用程序的需求,而辅助存储器容量大且价格便宜,故存储管理应能够从逻辑上扩充主存,把主存和辅助存储器混合起来使用,为用户提供比主存实际容量大得多的逻辑地址空间,方便用户编程,这就是第四章要讨论的虚拟存储器。操作系统的这方面功能与硬件存储器的组织结构和支持设施密切相关,操作系统设计者应根据硬件情况和用户需求,采取各种有效的存储资源分配策略和保护措施。

(3) 设备管理

设备管理的主要任务是:管理各种外部设备,完成用户所提出的 I/O 请求;加快数据传输速度,发挥设备的并行性,提高设备的利用率;提供设备驱动程序和中断处理程序,为用户隐藏硬件操作细节,提供简单的设备使用方法。为此,设备管理应该具备的功能如下:

- ① 提供设备中断处理;
- ② 提供缓冲区管理;
- ③ 提供设备独立性,实现逻辑设备到物理设备之间的映射;
- ④ 设备的分配和回收;
- ⑤ 实现共享型设备的驱动调度;
- ⑥ 实现虚拟设备。

(4) 文件管理

上述三种管理是针对计算机硬件资源的管理,文件管理则是针对信息资源的管理。在现代计算机系统中,通常把程序和数据以文件形式存储在辅助存储器(外存储器)上,以供用户使用。这样,辅助存储器上保存着大量文件,如果对这些文件不能采取合理的管理方式,就会导致混乱或使系统遭受破坏,造成严重的后果。为此,在操作系统中配置文件系统,主要任务是对用户文件和系统文件进行有效的管理,实现按名存取;实现文件的共享、保护和保密,保证文件的安全性;向用户提供一整套能够方便地使用文件的操作和命令。具体来说,文件管理的主要任务如下:

- ① 提供文件的逻辑组织方法;
- ② 提供文件的物理组织方法;
- ③ 提供文件的存取和使用方法;
- ④ 实现文件的目录管理;
- ⑤ 实现文件的共享和安全性控制;
- ⑥ 实现文件的存储空间管理。

(5) 网络与通信管理

计算机网络源于计算机技术与通信技术的结合,近些年来,从单机与终端之间的远程通信,到全世界或千上万台计算机联网工作,网络的应用范围已经十分广泛。操作系统至少应具有与网络有关的以下几项功能:

① 网络资源管理

实现网上资源共享,管理用户对资源的访问,保证信息资源的安全性和完整性。

② 数据通信管理

计算机联网后,结点之间可以互相传送数据,通过通信软件,按照通信协议的规定,完成网络上计算机之间的信息传送。

③ 网络管理

网络管理包括故障管理、安全管理、性能管理、日志管理和配置管理等。

(6) 用户接口

为了使用户能够灵活、方便地使用计算机硬件和系统所提供的服务,操作系统向用户提供一组使用其功能的手段,称为用户接口,包括两大类:程序接口和操作接口。用户通过这些接口能够方便地调用操作系统的功能,有效地组织作业及其处理流程,使得整个计算机系统高效地运行。

1.1.4 操作系统的主要特性

1. 并发性

并发性(concurrency)是指两个或两个以上的活动或事件在同一时间间隔内发生。操作系统是一个并发系统,有多道程序同时运行,这些程序称为并发程序,这样的系统就是并发系统。由于引入多道程序设计才导致程序的并发执行,因此,操作系统应该具有处理和调度多个程序同时执行的能力,多个程序都启动执行,但尚未完成执行。操作系统调度并发程序执行,使得计算机的多个硬部件同时工作,这是完全可能的。例如,CPU 和磁盘是不同的物理部件,可以让一个程序在 CPU 上执行,而另一个程序则向磁盘写数据;主存中同时有多个应用程序,或同时有操作系统程序和应用程序交替地启动、穿插地执行,这些都是并发性的例子。发挥并发性能够消除计算机系统中硬部件之间的相互等待,有效地改善资源利用率,提高系统效率。当一个程序正在等待执行 I/O 操作时,就要求它让出 CPU,调度另一个程序占有 CPU 运行。这样,CPU 资源便不会空闲,使得多个设备同时工作,也可使设备 I/O 操作和 CPU 计算同时进行,这是采用并发技术的结果。

并发性虽然能有效地改善系统资源的利用率,但却会引发一系列问题,使操作系统的设计和实现变得复杂化。例如,怎样实现不同程序之间的切换?以何种策略选择程序运行?怎样让多个运行程序互通消息及协作完成任务?怎样协调多个运行程序对资源的竞争?操作系统必须具有控制和管理程序并发执行的能力。为了更好地解决上述问题,系统引入“进程”的概念,并提供策略和机制进行协调,使各个进程顺利推进,并获得正确的运行结果。

采用并发技术的系统又称多任务处理系统(multitasking system)。在计算机系统中,采用并发技术实际上是物理 CPU 在若干道程序之间的多路复用,从而实现运行程序之间的并发,以及 CPU 与设备、设备与设备之间的并行。并发性的实质是对有限的物理资源强行复用,供多用户共享以提高效率。在单 CPU 系统中,多个程序并发执行是宏观上的概念,从微观上看,它们是顺序执行的;在多 CPU 系统中,并发性不仅体现在宏观上,而且体现在微观上,这称为并行。并行性(parallelism)指两个或两个以上的活动或事件在同一时刻发生。在多道程序环境下,并行性使

得多个程序同一时刻可以在不同的 CPU 上执行。在分布式系统中,多台计算机的并存使程序的并发性得到更充分的发挥,因为同一时刻每台计算机上都可以有程序在并发执行。可见并行活动一定是并发的,反之并发活动未必是并行的,并行性是并发性的特例,而并发性是并行性的扩展。并发技术的基本思想是:当一个程序发生事件(如等待执行 I/O 操作)时让出其占用的 CPU 而由另一个程序运行,由此不难看出,实现并发的关键技术之一是对系统内的多个运行程序(进程)进行切换的技术。

2. 共享性

共享性(sharing)是操作系统的另一个重要特性,指计算机系统中的资源可以被多个并发执行的程序共同使用,而不是被某个程序独占。出于经济上的考虑,向每个程序一次性提供所需的全部资源不但过于浪费,而且是不可能的,现实的方法是让系统程序和应用程序共用一套计算机系统资源。因而,并发性必然会产生资源共享的需要。资源共享的方式有以下两种。

(1) 透明资源共享

操作系统采用复用、虚化和抽象技术创建虚拟机,每个应用程序在各自的虚拟机上运行,虚拟机是物理计算机的仿真,系统通过共享硬件的方式来实现这种抽象。例如,操作系统为 3 台虚拟机分配不同的主存块,当应用程序装入虚拟机运行时,它们空分复用共享主存资源,时分复用共享处理器,其中,第一台虚拟机上的程序在计算,第二台虚拟机上的程序在读盘,第三台虚拟机上的程序在写数据,每个程序仅占用处理器的一个时间片,但却一直占用各自分得的主存区域直至运行完成。

磁盘资源已被抽象成命名文件,应用程序和系统程序都可以将各自的信息组织成文件,同时存储在磁盘上,在同一时间段内多个应用程序和系统程序都能访问磁盘,此处“同时”是宏观上的说法,从微观上看,多个程序访问文件仍然是先后进行的,只是这种先后访问的顺序对各自的访问结果并不会产生影响。当应用程序输入信息或打印输出时,感觉直接在控制硬件设备工作,但实际上所使用的是相应的虚拟设备,通过透明方式共享物理输入机和打印机。

透明资源共享机制必须妥善地处理资源隔离和授权访问。

① 资源隔离

这是操作系统的责任,无论时分复用还是空分复用,当资源被分配给一台虚拟机使用时,要防止其他虚拟机的未授权访问。存储器隔离机制允许两个应用程序同时加载到主存的不同区域,但是任何虚拟机都不能访问其他虚拟机的主存块;处理器隔离机制强制虚拟机串行地共享物理处理器。

② 授权访问

资源隔离是必要的,但是操作系统必须在出现资源请求时,通过授权方式,允许两个或多个应用程序合法地访问共享资源。例如,两个应用程序协作计算职工薪水,需要访问同一个职工信息文件,为了能做到协同工作,应允许一个程序共享另一个程序的计算结果,允许两个应用程序共享同一个主存块(其中有职工信息文件)中的内容。

(2) 显式资源共享

采用复用和虚化技术能够扩充某些资源的数量,应用程序不可能再因为竞争这些资源而造成阻塞,但系统中还有其他资源,如磁带机、扫描仪,由于其物理特性所决定,在同一时间段内只允许一个应用程序访问,即要求排它地使用这类资源。当一个程序在使用某种资源时,其他欲访问此资源的程序必须等待,仅当占有者访问完毕并释放资源后,才允许资源被再次分配。这种同一时间段内只允许一个程序访问的资源称为独占资源或临界资源,许多物理设备以及某些共享数据和表格都是独占资源,它们只能互斥地被访问和使用,否则会破坏数据完整性或造成数据不一致。为此,操作系统必须提供显式资源共享机制(申请/释放资源的系统调用,或锁机制),使得应用程序能够根据自己的需要及协调策略来请求和使用资源。

与共享性有关的问题是资源分配、信息保护、存取控制等,必须妥善解决这些问题。共享性和并发性是操作系统的两个基本特性,它们互为依存,一方面,资源共享是由程序的并发执行而引起的,若系统不允许程序并发执行,也就不存在资源共享的问题;另一方面,资源共享是支持并发性的基础,若系统不能对资源共享实施有效的管理,必然会影响程序的并发执行,甚至导致程序无法并发执行,整个系统就会丧失并发性,导致系统效率低下。

3. 异步性

操作系统的第三个特性是异步性(asynchronism),又称随机性。在多道程序环境中,允许多个程序并发执行,并发活动会导致随机事件的发生。由于资源有限而程序众多,每个程序的执行并非连贯的,而是“走走停停”。例如,程序A在CPU上运行一段时间后,由于等待所需资源的满足或某一事件发生,它被暂停执行,CPU被让给另一个程序B。系统中的程序何时执行成何时暂停?以何种速度向前推进?每个程序花费多少时间执行结束?这些都是不可预知的,或者说并发程序是以异步方式运行的。系统的功能及服务因响应事件而被激活,但是事件以不均等的频率发生,所导致的直接后果是程序执行结果可能不唯一。异步性会给系统带来潜在的危险,有可能导致并发程序的执行产生与时间有关的错误,但是操作系统必须保证:只要运行环境相同,多次运行同一程序,都会获得完全相同的计算结果。

操作系统中的随机性处处可见,例如,作业到达系统的类型和时间是随机的;操作员发出命令或单击按钮的时刻是随机的;程序运行过程中发生中断的时刻是随机的,等等。随机性并不意味着操作系统无法控制资源的使用和程序的执行,系统内部产生的事件序列有许多可能性,操作系统的一项重要任务是确保捕捉任何随机事件,正确地处理可能发生的随机事件及其序列,否则会导致严重的后果。

1.2 操作系统的形成与发展

1.2.1 人工操作阶段

从计算机诞生到20世纪50年代中期的机器属于第一代计算机,其运行速度慢,规模较小,设备较少,操作系统尚未出现。程序员采用手工方式直接控制和使用计算机,利用机器语言编

程,将事先准备好的程序和数据穿孔在卡片或纸带上,并从卡片或纸带输入机将程序和数据输入计算机。然后,启动程序运行,程序员通过控制台上的按钮、开关和氖灯来操纵和控制程序,程序运行完毕时取走计算结果,下一个用户方可上机。

随着时间的推移,产生汇编语言,采用汇编语言编程,原来的数字操作码被记忆码代替,程序按固定格式书写,系统程序员预先编制一个汇编程序,它负责把汇编语言书写的源程序解释成计算机可直接执行的机器语言形式的目标程序,在执行时需要把汇编程序、源程序和数据都穿孔在卡片或纸带上,然后装入并执行。其大致过程如下。

- (1) 把源程序和库函数用穿孔机人工穿在卡片或纸带上;
- (2) 将输入的汇编程序装入主存,并把控制权移交给它;
- (3) 汇编程序读入源程序和库函数;
- (4) 执行汇编程序,因为主存容量太小,所产生的目标程序被输出到卡片或纸带上;
- (5) 执行装入程序,把装载在输入机上的目标程序读入机器,并调用库函数连接装配成可执行程序;
- (6) 运行可执行程序,从输入机上读取数据卡片或纸带上的数据;
- (7) 产生计算结果,将其在打印机或卡片机上输出。

上述方式比直接使用机器语言有所进步,程序易于编制和阅读,汇编程序可以执行存取信息、存储分配等辅助性工作,还能够提供处理繁杂任务的函数库,从而在一定程度上减轻用户负担。但是,计算机的操作方式并未发生很大改变,仍然是由人工控制程序的装入和执行,其缺点是:用户独占全机资源串行计算,造成资源利用率不高,系统效率低下;人工干预环节较多,如装卡片或纸带、按开关等,手工操作不但浪费机器时间,而且极易发生差错;由于数据的输入、程序的执行、结果的输出均联机进行,因而每位用户从上机到下机之间的时间拉得很长。随着CPU速度的迅速提高,而设备运行速度却提高不多,导致CPU与设备之间的速度不匹配,这一矛盾越来越突出,需要妥善加以解决。

1.2.2 管理程序阶段

早期程序执行的每一步都需要人工干预和辅助,导致大量的计算机时间被消耗,使得极其昂贵的硬件设备只能低效使用,解决方法是利用一个控制程序对重复的操作过程“装入—汇编/编译—执行—输出”实现自动化,能够识别和装入所需的系统程序,如装入程序、汇编程序、编译程序、链接程序和函数库,能够处理作业之间的自动切换,这样用户就可以向系统提交多个作业同时处理,这个控制程序称为管理程序。

实现作业的自动切换和按步执行需要借助作业控制语言(Job Control Language,JCL),用户以脱机方式使用计算机,通过作业控制卡来描述对作业的加工和控制步骤,由作业控制卡、程序和数据所组成的作业被提交给计算机操作员,待收集一批作业后便输入机器。在管理程序控制下实现作业的计算及作业之间的自动转换,缩短作业的准备时间和建立时间,减少人工操作和干预,让计算机尽可能地连续运转。管理程序又称监督程序,FMS(FORTRAN Monitor System)和

IBSYS(IBM 7094 Monitor System)是这类系统的典型实例,其主存组织方式如图 1.4 所示,主要功能如下。

(1) 自动控制和处理作业流

作业流的自动控制和处理主要依靠 JCL 实现,它由描述作业控制过程的语句组成,每条语句附有一行作业或作业步骤信息编码,并以穿孔卡的形式提供。管理程序通过输入、解释和执行已嵌入用户作业的作业控制卡所规定的功能,能够自动地处理用户作业流,大幅度地减少作业的准备时间和建立时间。

(2) 提供一套操作命令

操作员通过打字机输入命令,管理程序识别并执行这些命令,这样做不仅速度快,还可以进行一些复杂的控制;输出信息可由打字机输出,代替早期的氖灯显示方式,易于理解。这种交互方式不仅提高系统效率,而且便于使用。

(3) 提供设备驱动程序和 I/O 控制功能

系统提供标准 I/O 程序,用户通过管理程序来控制和使用外部设备,管理程序还能处理某些设备故障,改进设备的可靠性和可用性。

(4) 提供库函数和程序链接功能

库函数包括格式编排、标准 I/O 程序等。

链接工作通常由管理程序完成。所有程序都按照相对地址编址,管理程序把相应的库函数和应用程序进行链接,并转换成绝对地址形式的目标程序,以便执行。

(5) 提供文件管理功能

用户的程序和数据需要反复使用,信息要求持久地保存,从而产生了文件系统。用户可以按文件名而非信息的物理地址进行存取,方便灵活,安全可靠。

1.2.3 多道程序设计与操作系统的形成

1. 多道程序设计

在早期的单道批处理系统中,主存中仅有单个作业在运行,CPU 和其他硬件设备串行工作,致使系统中仍有许多资源空闲,设备利用率很低。在 20 世纪 60 年代,有两项技术取得突破性进展:中断和通道,使得计算机体系结构由原先的以 CPU 为中心转变成以主存为中心。通道能够产生 I/O 中断,具有中断主机工作的能力,这两种技术结合起来,为实现 CPU 和其他硬件设备的并行工作提供了一定的基础,这时多道程序的概念才变成现实。

多道程序设计(multiprogramming)是指允许多个作业(程序)同时进入计算机系统的主存并启动交替计算的方法。也就是说,主存中多个相互独立的程序均处于开始和结束之间,从宏观上看是并行的,多道程序都处于运行过程中,但尚未运行结束;从微观上看是串行的,各道程序轮流占用 CPU 以交替地执行。引入多道程序设计技术,可以提高 CPU 的利用率,充分发挥计算机硬

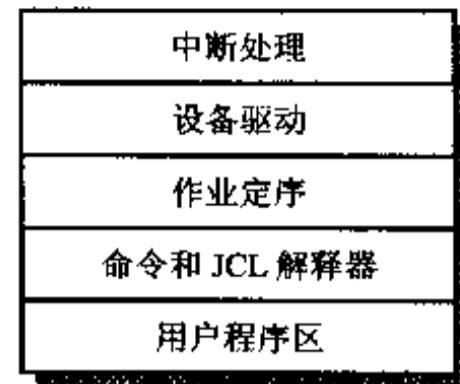


图 1.4 管理程序的主存组织

部件的并行性。现代计算机系统都采用多道程序设计技术。

首先分析程序运行时的特性。在现代计算机系统中,I/O 操作较慢而 CPU 运算速度快,故程序运行时花费在 I/O 操作上的时间最多;程序在执行 I/O 操作时并不需要 CPU;传统计算机系统配有许多硬件设备,但只有单一的 CPU。这样看来,让程序执行时使用计算机的不同部件来实现真正的并行操作是完全可能的。

接着讨论多道程序设计技术提高资源利用率的原理。从第二代计算机开始,系统具有 CPU 和设备并行工作的能力,使得计算机的运行效率有所提高。例如,求解某个数据问题,要求从输入机(运转速度为 6 400 个字符/s)输入 500 个字符,经处理(费时 52 ms)之后,将结果(假定为 2 000 个字符)存储到磁带上(磁带机的运转速度为 10⁵ 个字符/s),然后,再读取 500 个字符进行处理,直至所有数据处理完毕为止。如果 CPU 不具备同设备并行工作的能力,那么,CPU 的利用率为

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

可以看出系统效率之所以不高的原因,是因为当输入机输入 500 个字符之后,处理器只用 52 ms 进行处理,而第二批输入数据尚需等待 98 ms 才能输入完毕,在此期间 CPU 一直空闲。此例说明单道程序在工作时,计算机系统各部件的利用率并未得到充分的发挥。为了提高系统效率,让计算机同时接收两道计算题,当第一道程序等待设备进行数据传输时,让第二道程序运行,以缩短 CPU 空闲等待时间,那么,CPU 的利用率将得以提高。例如,计算机在接收上述例题时还接收另一道计算题:从另一台磁带机上输入 2 000 个字符,经 42 ms 的处理之后,从行式打印机(运转速度为 1 350 行/min)上输出两行。

当两道程序同时进入主存时,计算过程如图 1.5 所示。其中,P_甲 表示程序甲占用 CPU 对输入的 500 个字符进行处理,由于 52 ms 便处理结束,下次处理要等待 98 ms,因而这段时间内 CPU 是空闲的。系统调度程序乙工作,它从磁带机上输入 2 000 个字符,P_乙 表示程序乙对这批数据进行处理,相应的设备和 CPU 操作都是并行的。不难算出,此时 CPU 的利用率为

$$(52 + 42)/150 \approx 63\%$$

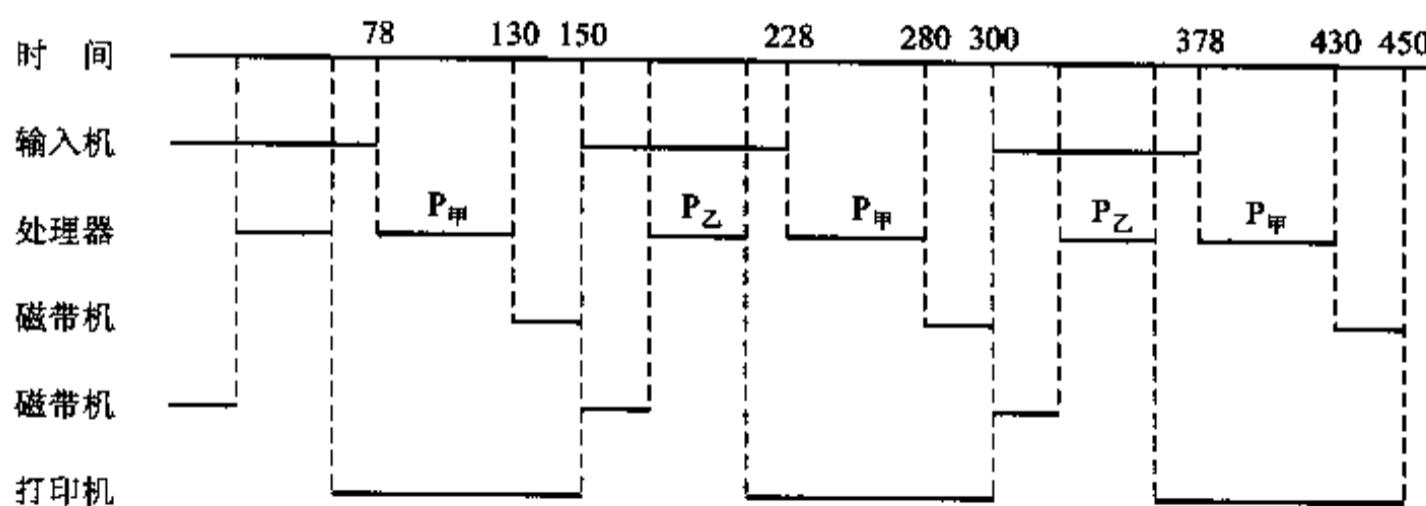


图 1.5 两道程序运行时处理器的利用率

所以,多个程序同时进入主存执行计算比逐个程序串行计算的 CPU 利用率要高,因为当某个程序因故不能继续运行时,系统会把 CPU 分配给另一个程序,使得 CPU 和设备尽量处于忙碌状态,这就是采用多道程序设计技术的主要原因。具有 CPU 和设备并行工作能力的计算机采用多道程序设计技术之后,可以提高 CPU 和设备的并行性,从而提高系统的吞吐率,即增加单位时间内完成运算题目的数量。

在多道程序设计中,值得注意的是道数的多少。从表面上看,似乎道数越多则效率越高,但是道数的具体数目往往受到系统资源的限制。如果上述甲、乙两个程序都要使用行式打印机,而系统只有一台打印机,即使同时接收两个程序进入计算机主存运行,也未必能够提高效率。可能程序甲运算一段时间后,要等待程序乙不再使用行式打印机即程序乙运行结束后,才能继续运行。此外,主存的容量和用户的响应时间等因素也会影响多道程序的道数。可以采用概率方法计算 CPU 的利用率,假设程序平均等待 I/O 操作的时间占其运行时间的比例为 p ,当主存中有 n 道程序时,所有程序都等待 I/O 操作的概率是 p^n ,即此时 CPU 是空闲的。那么

$$\text{CPU 利用率} = 1 - p^n$$

其中, n 称为多道程序的道数或度数(degree of multiprogramming)。可见 CPU 利用率是 n 的函数。如果进程平均花费 80% 的时间等待 I/O 操作,为了使 CPU 浪费的时间低于 10%,粗略计算至少要有 10 道程序在主存中运行。事实上,一个程序等待从终端输入信息或等待磁盘 I/O 操作的时间超过 80% 是常有的事,上述 CPU 利用率的计算模型虽然粗略但却十分有效。假设计算机主存容量为 1 MB,操作系统的运行占用 200 KB,其余主存空间允许 4 道程序共享,每道程序占用 200 KB,若 80% 的时间用于等待 I/O 操作,则在忽略操作系统的开销时,CPU 利用率 = $1 - (0.8)^4 = 59\%$ 。当增加 1 MB 主存空间后,多道程序可以从 4 道增加到 9 道,因而,CPU 利用率 = $1 - (0.8)^9 = 87\%$ 。也就是说,第二个 1 MB 主存空间可以增加 5 道程序,提高 47% 的 CPU 利用率。如果再增加第三个 1 MB 主存空间,只能将 CPU 的利用率从 87% 提高到 96%,CPU 利用率仅仅提高约 10%。

操作系统中引入多道程序设计的优点:一是提高 CPU、主存和设备的利用率;二是提高系统的吞吐率,使单位时间内完成的作业数增加;三是充分发挥系统的并行性,设备与设备之间、CPU 与设备之间均可并行工作。其主要缺点是延长作业的周转时间。

多道程序设计系统与多重处理系统(multiprocessing system)之间存在一定的差别,后者是指配置多个物理处理器,能真正同时执行多道程序的计算机系统,当然要有效地使用多重处理系统,必须采用多道程序设计技术;反过来,多道程序设计不一定要求多重处理系统的支持,虽然多重处理系统增加了硬件设备,但却换取提高系统吞吐量、可靠性、计算能力和并行处理能力的好处。

实现多道程序设计必须妥善地解决以下 3 个问题。

(1) 存储保护与程序浮动

在多道程序设计环境中,主存为多道程序所共享,因此,硬件必须提供相应的设施,使得主存中的各道程序只能访问自己的区域,以避免相互干扰。当某道程序发生错误时,不致影响其他程

序,更不会影响系统程序,这就是存储保护。同时,由于各道程序不是独占全机,程序员在编制程序时无法知道程序在主存中的确切地址,甚至在运行过程中,程序也可能随时改变位置,因此要求程序能够根据需要从一个主存区移动到另一个区,而不影响其正确执行,这称为程序浮动,或地址重定位。此外,多个程序共存于主存,会引起主存容量不足,因而主存扩充也成为操作系统必须要解决的问题。

(2) 处理器的管理与分配

在多道程序设计环境中,如果仅配置一个或几个物理处理器,那么多道程序必须轮流占有它们(们),这涉及 CPU 的调度和分配,要解决多道程序的切换和有效运行。同时,合理搭配具有不同特性的多道程序同时运行,也是 CPU 调度所要解决的问题。

(3) 资源的管理与调度

其他资源(如存储器、设备及文件)均需按照一定的策略来分配和调度,既要解决多道程序共享软硬件资源时的竞争与协作、共享与安全等问题,又要解决发挥各种资源的利用率的问题。有关调度算法及解决上述问题的机制,将在后续章节叙述。

2. 操作系统的形成

第三代计算机的性能有更大的提高,机器运行速度更快,主存容量增大,设备数量和种类增多,为软件发展提供了有力的支持。要更好地发挥硬件的功效,更好地满足各种应用的需要,都迫切要求扩充管理程序的功能。

中断和通道技术的出现使得硬部件具有并行工作的能力,从理论上说,实现多道程序系统已不存在问题,但是,从半自动的管理程序方式过渡到能够自动控制程序执行的操作系统方式,对辅助存储器性能的要求更高,操作系统的真正形成还期待大容量高速辅助存储器的出现,大约到 20 世纪 60 年代中期,随着磁盘的问世,相继出现多道批处理操作系统和分时操作系统、实时操作系统,此时标志操作系统的正式形成。计算机配置操作系统之后,资源管理水平和操作的自动化程度进一步得以提高,具体表现在:提供存储管理、文件管理、设备管理功能,支持分时操作,多道程序设计趋于完善。

1.2.4 操作系统的发展与分类

促使操作系统不断发展的因素有很多。首先,硬件技术的发展速度很快,器件更新换代迅速,从电子管到晶体管,从集成电路到超大规模集成电路,微电子技术作为推动计算机技术飞速发展的“引擎”,促使计算机系统快速更新,由 8 位机、16 位机发展到 32 位机,目前已经研制多种 64 位机,相应的 64 位操作系统也被研制和开发出来。计算机体系结构也在不断地发展,由硬件改进而导致操作系统发展的例子很多:主存管理支撑着硬件由分页或分段设施取代上下界寄存器后,出现虚拟存储器;图形界面终端代替字符显示终端后,窗口系统被广泛接受;随着因特网的迅猛发展和日趋普及,出现分布式操作系统和网络操作系统;随着信息家电的产业化,出现嵌入式操作系统。其次,提高计算机系统的资源利用率始终是操作系统发展的动力之一,多用户共享计算机系统的资源,必须想方设法地提高计算机系统的资源利用率,各种调度算法、分配策略相

继被研究和采纳。最后,应用需求不断地促使操作系统变革,从批处理操作系统到交互型分时操作系统,改善了用户上机、调试程序的环境;从命令行交互进化到图形用户界面,界面变得更加友善,使用计算机变得更加方便;当用户感觉现有功能不能满足需要时,操作系统往往要升级换代,开发新工具,加入新设施。

在操作系统发展的各个阶段,使用不同的策略为用户提供不同的服务。由于所提供的作业处理方式不同,虚拟计算机的一般特征也就不一样,从而在用户面前,呈现具有不同处理方式和不同运行特点的操作系统。按照功能、特点和使用方式,可以把操作系统分为三种基本类型。

1. 批处理操作系统

批处理操作系统服务于一系列称为批(batch)的作业,作业是把程序、数据连同作业说明书组织起来的任务单位,把批中的作业预先输入作业队列中,由操作系统按照作业说明书的要求来调度和控制作业的执行,大幅度减少人工干预,形成自动转接和连续处理的作业流,由操作员在计算结束之后把运算结果返回给用户。采用批处理方式工作的操作系统通常称为批处理操作系统(batch processing operating system),运行于 IBM 43xx 系列机上的 IBM DOS/VS 和 DOS/VSE 是典型的批处理操作系统。

批处理操作系统是最先采用多道程序设计技术的系统,它根据预先设定的调度策略选择若干作业并发地执行,系统的资源利用率高,作业吞吐量大。其缺点是:作业的周转时间延长,不具备交互式计算的能力,不利于程序的开发和调试。批处理操作系统的特征如下。

(1) 脱机工作

用户在提交作业直至获得计算结果之前,不再和计算机及其作业交互,因而 JCL 对批作业来说必不可少。

(2) 成批处理

操作员集中用户提交的一批作业,预先输入计算机中作为后备作业,由批处理操作系统按照调度策略逐批地选择并装入主存执行。

(3) 单/多道程序运行

早期批处理操作系统采用单道批处理,作业进入系统之后排定次序,逐道依次进入主存处理,并自动进行作业的转接;后来采用多道批处理,从后备作业中选取多个作业进入主存,并启动其运行,这是多道批处理系统。

早期批处理的作业控制说明使用 JCL 书写,以一组穿孔卡片的形式输入计算机中,对作业的执行进行控制。“批”的含义现今已经发展为表示非交互式计算,一个批作业是指它不需要与任何用户交互,这类作业的优先级比交互型作业要低,系统会推迟执行它,直到有足够的存储空间及空闲的处理器资源。在现代操作系统中,执行批作业的控制流采用文件形式表示,如 UNIX 中的 Shell 文件、Windows 中的 autoexec.bat 文件,用户在这类文件中定义一系列操作系统命令,利用批处理功能,可以自动、连续地执行控制文件,在无用户干预的情况下读入批处理控制流文件。当某个作业需求的资源可用时,系统便执行此作业,在批作业完成后,结果被打印输出并返回给用户。

2. 分时操作系统

批处理操作系统的焦点是性能,当时计算机硬件的成本较高,希望在单位时间内将处理作业的数量最大化,通过批处理方式产生足够的工作量,再利用多道程序设计技术,让 CPU 和设备并行执行,以达到提高作业吞吐能力的目的。但是,用户不能干预自己程序的运行,无法得知程序的进展情况,不利于程序调试和排错,为此,便产生分时操作系统。

允许多个联机用户同时使用一个计算机系统进行交互式计算的操作系统称为分时操作系统 (time-sharing operating system)。其实现思路如下:用户在各自的终端上进行会话,程序、数据和命令均在会话过程中提供,以问答方式控制程序的运行。系统把处理器的时间划分成时间片,轮流分配给各个联机终端,若时间片用完则产生时钟中断,控制权转至操作系统并重新进行调度。如果原程序尚未完成,挂起并等待再次分得时间片。由于调试程序的用户常常只发送简短的命令,这样其要求总能得到快速的响应,好像用户独自占用计算机系统一样。实质上,分时操作系统是多道程序的一个变种,CPU 被若干交互式用户多路复用,不同之处在于每个用户都拥有一台联机终端。

1961 年,美国麻省理工学院开发第一个分时系统 CTSS(Compatible Time-Sharing System,兼容分时系统),成功地运行在 IBM 709 和 IBM 7094 机上,支持 32 个交互式用户同时工作。1965 年,IBM 公司发布 IBM 360 机上的分时系统 TSS/360 (Time-Sharing System/360),这是一个失败的系统,由于过大过慢,没有用户愿意使用。

1965 年,在美国国防部的支持下,美国麻省理工学院、贝尔实验室和通用电气公司决定开发“公用计算服务系统”,以支持整个波士顿地区的所有分时用户,这就是著名的 MULTICS (MULTIple access Computer System,多路存取计算机系统),它运行在 GE - 635、GE - 645 计算机上,使用高级语言 PL/I 编程,引入现代操作系统的许多概念雏形,如分时处理、远程联机、段页式虚拟存储器、多级反馈调度、保护环安全机制、多种程序设计和开发环境等,对其后操作系统的设计产生极大的影响。

1970 年,曾经参与 MULTICS 的美国 AT&T 公司贝尔实验室的研究人员开发研制著名的 UNIX 分时操作系统,在 CTSS、MULTICS 等系统消失多年之后,它仍然是一个主流操作系统,表现出强大的生命力。当前,分时操作系统已经成为最流行的一种操作系统,几乎所有现代操作系统都具备分时处理的功能。

分时操作系统的特点如下:

(1) 同时性

若干终端用户联机使用计算机,分时是指多个用户分享同一台计算机的 CPU 时间。

(2) 独立性

终端用户彼此独立,互不干扰,每个终端用户感觉好像独占整台计算机

(3) 及时性

终端用户的立即型请求(没有大计算量的处理请求)能够在足够短的时间内得到响应(通常为 3 s)。

(4) 交互性

人机交互,联机工作,用户直接控制程序的运行,便于程序调试和排错。

分时操作系统和批处理操作系统虽然具有共性,都基于多道程序设计技术,但它们存在一些不同点。

(1) 追求目标不同

批处理操作系统以提高系统资源利用率和作业吞吐能力为目标;分时操作系统强调公平性,对于联机用户的立即型命令要快速响应。

(2) 适应作业不同

批处理操作系统适应已调试好的大型作业;而分时操作系统适应正在调试的小型作业。

(3) 资源利用率不同

批处理操作系统可以合理安排不同负载的作业,使资源利用率达到最佳;在分时操作系统中,多个终端的作业使用相同类型的编译系统、运行系统和公共子程序时,系统调度的开销较小,能够公平地调配 CPU 和主存资源。

(4) 作业控制方式不同

批处理操作系统由用户通过 JCL 书写作业控制流,预先提交,脱机工作;交互型作业由用户从键盘输入控制命令,以交互方式联机工作。

影响分时操作系统响应时间的因素有很多,如 CPU 的处理速度、联机终端的数目、所用时间片的长短、系统调度开销和对换信息量的多少,可以通过控制终端数目、调整时间片、采用重用代码以减少对换信息量等方法来改善响应时间。

3. 实时操作系统

虽然多道批处理操作系统和分时操作系统能够获得较佳的资源利用率和快速响应时间,使得计算机的应用范围日益扩大,但它们难以满足实时控制和实时信息处理的需要,于是便产生实时操作系统,有三种典型的实时系统:过程控制系统、信息查询系统和事务处理系统。计算机用于过程控制时,要求系统实时采集数据,并对其进行分析处理,进而自动发出控制信号以控制相应的执行机构;使某些参数(压力、温度、距离、湿度)能够按照预定的规律变化,保证产品质量,导弹制导系统、飞机自动驾驶系统都是实时过程控制系统。计算机还可用于实时信息处理,情报检索系统、仓库管理系统等都是信息查询系统,计算机同时接收各终端所发来的服务请求和提问,快速查询信息数据库,在极短的时间内做出响应。事务处理系统不仅对终端用户及时做出响应,还要对系统中的文件或数据库频繁加以更新,例如,银行业务处理系统和飞机订票系统在每次发生业务往来时均需查询和修改文件或数据库,这样的系统应该具有响应迅速、安全保密、可靠性高等特点。

实时操作系统(real-time operating system)是指当外部事件或数据产生时,能够对其予以接收并以足够快的速度进行处理,所得结果能够在规定的时间内控制生产过程或对控制对象做出快速响应,并控制所有实时任务协调运行的操作系统。因而,提供及时的响应和高可靠性是其主要特点。由实时操作系统所控制的过程控制系统较为复杂,通常由以下 4 个部分组成。

(1) 数据采集

收集、接收和输入系统工作必需的信息或进行信号检测。

(2) 加工处理

对进入系统的信息进行加工处理,获得控制系统工作所必需的参数或做出决定,然后,进行结果输出、记录或显示。

(3) 操作控制

根据处理结果采取适当的措施或动作,达到控制或适应环境的目的。

(4) 反馈处理

监督执行机构的执行结果,并将此结果反馈至信号检测或数据接收部件,以便系统根据反馈信息进一步采取措施,达到施加控制的预期目的。

在实时操作系统中要有实时时钟管理,以便对实时任务进行实时处理。系统中通常存在若干实时任务,往往通过“队列驱动”或“事件驱动”开始工作,当系统接收某个外部事件之后,驱动实时任务以完成相应的处理和控制。

上面介绍了操作系统的三种基本类型,如果某个操作系统兼具批处理、分时和实时处理的全部或两种功能,则此操作系统称为通用操作系统。

4. 微机操作系统

微型计算机的出现引发计算机产业革命,使其进入社会生活的各个领域,拥有巨大的使用量和最广泛的用户群。从 20 世纪 70 年代中期到 20 世纪 80 年代早期,微型计算机上运行单用户单任务操作系统,如 CP/M、CDOS(Cromemco 磁盘操作系统)、MDOS(Motorola 磁盘操作系统)和早期的 MS-DOS(Microsoft 磁盘操作系统)。从 20 世纪 80 年代中期到 20 世纪 90 年代初,微机操作系统开始支持单用户多任务和分时操作,其中以 MP/M、XENIX 和 Windows 9x 系列为典型代表。

近年来,微机操作系统得到进一步的发展,以 UNIX、Windows、OS/2 和 Linux 为代表的多用户多任务微机操作系统具有 GUI、虚拟存储管理、网络通信支持、数据库支持、多媒体支持、SMP 支持、API 支持等功能,及开放性、通用性、高性能等特点。

5. 网络操作系统

计算机网络通过通信设施将地理上分散且具有自治功能的多个计算机系统互连起来,网络操作系统能够控制计算机在网络中传送信息和共享资源,并为网络用户提供所需的各种服务,其主要功能有网络通信、资源管理、网络管理和网络服务等。目前,主要的三种网络操作系统是:UNIX、NetWare 和 Windows NT。UNIX 是唯一的跨平台操作系统;Windows NT 工作在微型计算机和工作站上;NetWare 则主要面向微型计算机。支持客户—服务器结构的网络操作系统主要有 NetWare、UNIX、Windows NT、LAN Manager 和 LAN Server 等。

6. 分布式操作系统

处理和控制功能高度集中在一台计算机上,所有任务均由它完成,这种系统称为集中式计算机系统,而分布式计算机系统是指由多台分散的计算机经网络连接而成的系统,每台计算机既高

度自治,又协同工作,能够在系统范围内实现资源管理和任务分配,能够并行运行分布式程序。用于管理分布式计算机系统的操作系统称为分布式操作系统,它与集中式操作系统之间的主要区别在于资源管理、进程通信和系统结构等方面,第八章将对分布式操作系统作进一步介绍。

著名的分布式操作系统有 Amoeba(荷兰自由大学)、Plan 9(AT&T 公司贝尔实验室)、Cm*(美国卡内基 - 梅隆大学)、X 树系统(美国加州大学伯克利分校)、Arachne(美国威斯康星大学)、CHORUS(法国国家信息与自动化研究所)、GUIDE(法国 Bull 研究中心)、Clouds(美国佐治亚理工学院)和 CMDS(英国剑桥大学)等。

7. 嵌入式操作系统

随着以计算机技术、通信技术为主导的信息技术的快速发展和因特网的广泛应用,3C(Computer, Communication, Consumer Electronics)合一趋势已初现端倪,计算机技术是贯穿信息化的核心技术,网络和通信设备是信息化赖以存在的基础设施,电子消费产品是人与社会信息化的主要接口。3C 合一的必然产物是信息电器,同时,计算机的微型化和专业化趋势已为人共识,这就为计算机技术渗透进各行各业、应用于各个领域、嵌入到各种设备、开发各种产品奠定了坚实的物质基础。这些领域有一个共同需求:计算机嵌入式应用。嵌入式(计算机)系统硬件不再以物理独立的装置或设备的形式出现,而是大部分甚至全部隐藏和嵌入各种应用系统中。由于嵌入式系统的应用环境与其他类型的计算机系统存在较大的区别,随之而来的是对嵌入式系统的软件,即嵌入式软件(embedded software)的需求,而嵌入式操作系统是嵌入式软件的基本支撑,从而形成现代操作系统的一个类别——嵌入式操作系统。随着信息电器和信息产业的飞速发展,面对巨大的需求量和生产量,嵌入式软件和嵌入式操作系统的应用前景将十分广阔。

嵌入式操作系统是指运行在嵌入式应用环境中,对整个系统及所有操作的各种部件、装置等资源进行统一协调、处理、指挥和控制的系统软件。由于嵌入式操作系统仍然是一个操作系统,因此它具有通用操作系统的功能,但由于其所处硬件平台的局限性、应用环境的多样性和开发手段的特殊性,它与一般的操作系统相比又有很大不同,具有微型化、可定制、实时性、可靠性、易移植性等特点。

嵌入式操作系统与具体的应用环境密切相关,按照应用范围划分,可分为通用型和专用型,前者适用于多种应用领域,著名的产品有 Windows CE、VxWorks 和嵌入式 Linux;而后者则面向特定的应用场合,如适用于掌上计算机的 Palm OS、适用于移动电话的 Symbian 等。至今已有几十种嵌入式操作系统而世,代表性的嵌入式操作系统有 CHORUS、Diba(Sun 公司)、Navio(Oracle 公司)、OS-9(Microware 公司)、pSOS(ISI 公司)和 QNX(QSSL 公司)等。

VxWorks 是美国 Wind River 公司开发的嵌入式实时操作系统,可靠性高,性能卓越,人机界面友好,广泛地应用于通信、军事、航空、航天等对精密性及实时性要求极高的领域,市场份额大,获得业界很高的声誉。美国 F-16 战斗机、FA-18 战斗机、B-2 隐形轰炸机、爱国者导弹和火星探测器(1997 年 4 月在火星表面登陆)均使用 VxWorks。它采用基于 Wind microkernel 的模块式结构,是一个可扩充、可配置的核心嵌入式操作系统,可选的扩充机制有消息通信、共享存储、网络连接、图形选件和 Java 支持等。Windows CE 是微软公司开发的用于通信、娱乐和移动

式计算设备的嵌入式操作系统,是微软公司“维纳斯”计划的核心,由内核、设备驱动、网络通信等服务组成,是32位多任务多线程嵌入式操作系统。

1.3 操作系统的基本服务和用户接口

1.3.1 基本服务和用户接口

1. 基本服务

操作系统要为应用程序的执行提供良好的运行环境和各种服务,虽然不同的操作系统所提供的服务不尽相同,但存在许多共同点,操作系统的共性服务将为程序员带来方便,使编程任务得以顺利完成。

(1) 创建程序和执行程序

提供程序编辑工具和调试工具,帮助程序员编程并生成高质量的源程序;将应用程序和数据装入主存,为其运行做好准备,并启动和执行程序;当程序编译或运行出现异常时,应能报告所发生的情况,终止程序执行或进行适当处理。

(2) 数据I/O和信息存取

程序在运行过程中需要设备中的数据时,通过I/O指令请求系统进行传输,而不是由用户直接控制设备I/O操作;文件系统使得用户按照文件名来建立、读写、修改及删除文件,使用方便,安全可靠;当涉及多用户访问或共享文件时,操作系统将提供信息保护机制。

(3) 通信服务

在许多情况下,进程之间需要交换信息,进程通信可借助于操作系统所提供的各种通信机制来实现。

(4) 差错检测和处理

捕获和处理各种硬件或软件所产生的错误或异常,并将其所造成的影响限制在最小范围之内,必要时及时向操作员或用户报告。

此外,操作系统还具有其他一些功能,以保证其自身高效率、高质量地工作,使得多个应用程序能够有效地共享系统资源,提高系统效率。这些功能列举如下。

(1) 资源分配

多道程序同时运行时,各程序必须获得系统资源,计算机系统中的各类资源均由操作系统管理,如CPU、主存、设备和文件等,配有专门的分配和管理程序、申请和释放程序。

(2) 统计

了解用户使用计算机资源的情况,如使用何种资源,占用多少时间或空间,以便用户付款或进行使用情况的统计,统计结果可作为改进系统服务、对系统进行重组的依据。

(3) 保护

在多用户多任务计算机系统中,保护意味着对系统资源的所有存取都要受到控制,应用程序

对各种资源的需求经常会发生冲突,为此,操作系统必须做出合理调度,提供多种服务。低层服务通过系统调用来实现,可被应用程序直接使用;高层服务通过实用程序来实现,用户借助命令管理或 shell 来请求执行。

2. 用户接口

操作系统通过程序接口和操作接口将其服务和功能提供给用户,如图 1.6 所示。程序接口是操作系统对外提供服务和功能的手段,它由一组系统调用组成,在应用程序中使用“系统调用”可获得操作系统的低层服务,访问或使用系统管理的各种软硬件资源;操作接口由一组控制命令和(或)作业控制语言组成,是操作系统为用户提供的组织和控制作业执行的手段。

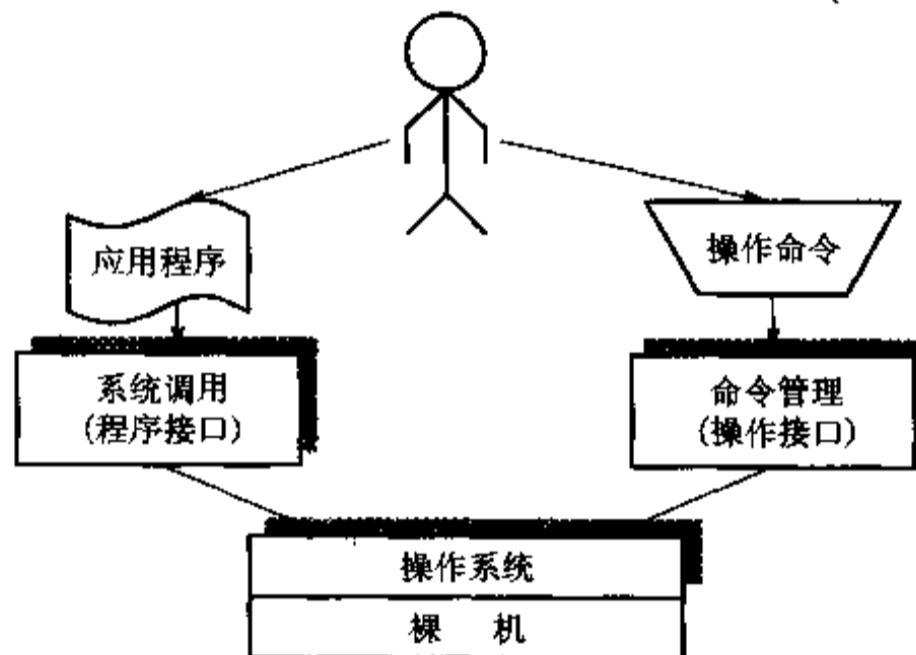


图 1.6 用户和操作系统之间的接口

1.3.2 程序接口与系统调用

1. 系统调用

操作系统的主要功能是为应用程序的运行创建良好的环境,为了达到这个目标,内核提供一系列具备预定功能的内核函数,通过一组称为系统调用(system call)的接口呈现给用户。系统调用把应用程序的请求传送至内核,调用相应的内核函数完成所需的处理,将处理结果返回给应用程序,如果没有系统调用和内核函数,用户将不可能编写出功能强大的应用程序。可以这样认为,内核的主体是系统调用的集合,可以将内核看成特殊的公共子程序。

操作系统服务之所以通过系统调用的方式供用户使用,其根本原因是为了对系统进行“保护”。程序的运行空间分为内核空间和用户空间,其程序各自按不同的特权运行,在逻辑上相互隔离。应用程序不能直接访问内核数据,也无法直接调用内核函数,它们只能在用户空间操纵用户数据,调用用户空间函数。但在很多情况下,应用程序需要获得系统服务,这时就必须利用系统提供给用户的特殊接口——系统调用。

系统调用是一种中介角色,把用户和硬件隔离开来,应用程序只有通过系统调用才能请求系

统服务并使用系统资源。系统调用的作用:一是内核可以基于权限和规则对资源访问进行裁决,保证系统的安全性;二是系统调用对资源进行抽象,提供一致性接口,避免用户在使用资源时发生错误,且使编程效率提高。

当 CPU 执行程序中预设的由访管指令实现的系统调用时,会产生异常信号,通过中断机制,处理器的状态由用户态转变为核心态,进入操作系统并执行相应的内核函数,以获得操作系统服务;当系统调用执行完毕时,控制返回至发出系统调用的程序,系统调用是应用程序获得操作系统服务的唯一途径。

2. API、库函数和系统调用

每个操作系统所提供的一组系统调用虽然功能大同小异,但是其实现细节不尽相同,与具体的机型相关。如果应用程序直接使用系统调用,则至少存在两个问题,一是接口复杂、使用困难,二是应用程序的跨平台可移植性受到很大的限制。为此,IEEE 开发 POSIX(Portable Operating System Interface for Computer Environments, 计算机环境可移植操作系统接口)标准,其中 POSIX 专门规定内核的系统调用接口标准,操作系统的实现若遵循此标准,那么应用程序在不同的操作系统之间就具有可移植性。UNIX/Linux 遵循 POSIX 标准,所以它们是 POSIX 兼容的操作系统。

为了能够在 C 语言程序中使用系统调用,UNIX/Linux 在标准 C 函数库中为每个系统调用构造一个同名的封装函数(wrapper function),屏蔽其下各层的复杂性,负责把操作系统所提供的服务接口——系统调用,封装成应用程序能够直接使用的 API(Application Program Interface, 应用程序接口),所以,一个库函数(封装函数)就是一种 API,它介于应用程序和操作系统之间。例如,GNU C(glibc)或标准 C(libc)函数库均属于此类,把操作系统的内部编程接口封装成 POSIX 标准接口,为每个系统调用设置一个封装函数,应用程序采用标准 C 调用序列来调用封装函数,封装函数按照系统所要求的形式和方法传递参数,执行访管指令,调用相应的内核函数,提供用户所需的服务。

API 是一个函数定义,说明如何获得给定的服务。一个 API 的实现可能会用到一个系统调用或多个系统调用,也可能若干 API 封装相同的系统调用,即使完全不使用系统调用,也不存在任何问题。例如,open()库函数与 open()系统调用一一对应,但 strcpy()库函数与系统调用无关。对于较复杂的库函数,通过系统调用转向内核函数只是其工作的一小部分,fprintf()就是这样,它要提供数据缓存和格式编排,最后才通过 write()系统调用把数据写到设备上。实际上,API 可以在不同的操作系统上实现,向应用程序提供完全相同的功能和接口,而其内部的实现却迥异。POSIX 标准仅定义一套过程,但并未规定过程的实现,是采用系统调用、库函数还是其他形式,如果无须陷入内核就可以实现一个函数,从性能方面考虑,它通常应在用户空间中完成。在 UNIX/Linux 系统中,根据 POSIX1003.1 定义的标准,库函数与系统调用之间大多存在直接对应关系。事实上,POSIX 标准主要仿照 UNIX 界面建立。另一方面,许多现代操作系统,如 Windows,尽管独立于 UNIX,但也提供与 POSIX 兼容的子系统。

Windows 操作系统不公开系统调用,仅提供以库函数形式定义的 API,称为 Win32 API。

UNIX/Linux 操作系统既支持库函数, 又公开系统调用, 所以, UNIX 应用程序既可通过宏 `_syscalln(n)`(n 为所传递的参数个数) 直接使用系统调用, 又可通过库函数间接使用系统调用, 以此获得操作系统的服务。如图 1.7 所示为应用程序执行 API `fprintf()` 时, 调用 C 函数库中的 `fprintf()` 库函数, 再调用 C 函数库中的 `write()` 库函数, 最终调用内核函数 `sys_write()` 的关系链。

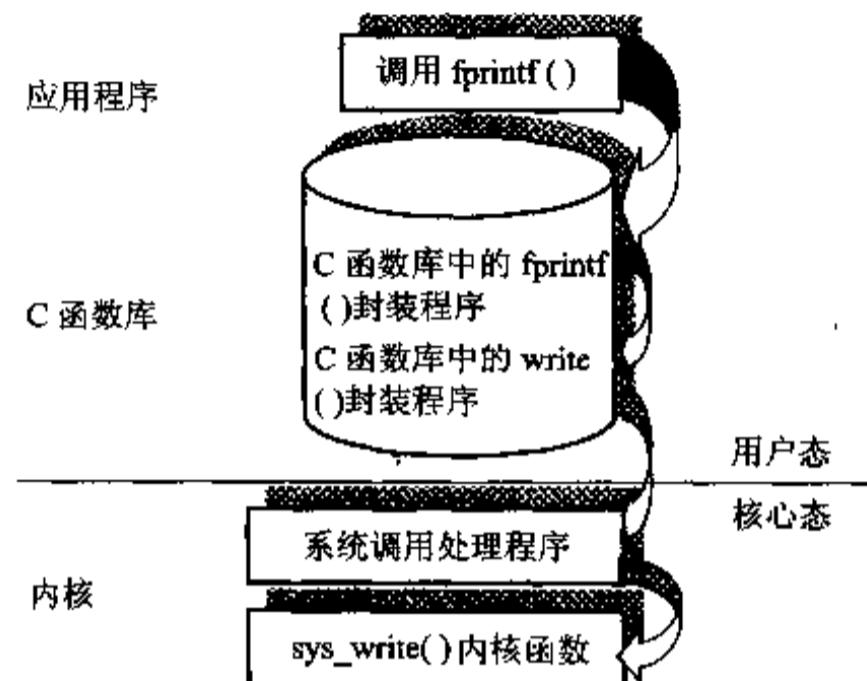


图 1.7 应用程序、库函数、系统调用的关系链

从应用程序的角度来看, 库函数与系统调用之间的差别并不重要, 但从实现的角度来看, 两者之间存在重大区别。使用库函数的好处是可以隐藏访管指令的细节, 使得系统调用更像函数调用, 对用户屏蔽系统调用, 这样在改动内核时不会影响应用程序的正确性。但是库函数属于应用程序, 在用户态运行, 系统调用属于系统程序, 在核心态运行, 如果需要的话, 用户可以替换库函数, 通常却不能替换系统调用。

3. 系统调用的分类

操作系统所提供的系统调用很多, 按功能可分成六类。

(1) 进程管理

创建和撤销进程; 终止或异常终止进程; 阻塞和唤醒进程; 挂起和激活进程; 获取和设置进程属性。

(2) 文件操作

建立文件; 删除文件; 打开文件; 关闭文件; 读写文件; 控制文件; 显示文件和目录的内容; 显示和设置文件属性。

(3) 设备管理

申请设备; 释放设备; 设备 I/O 操作和重定向; 获得和设置设备属性; 控制和检查设备状态。

(4) 主存管理

申请和释放主存; 增加或减少主存。

(5) 进程通信

建立和断开通信连接;发送和接收消息;传送状态信息;连接和断开远程设备。

(6) 信息维护

获取和设置日期及时间;获取和设置系统数据;生成诊断和统计数据。

Windows 操作系统的库函数称为 Win32 API,通过 3 个组件提供服务:Kernel、User 和 GDI (Graphical Device Interface, 图形设备接口)。Kernel 包含大多数操作系统函数,如主存管理、进程管理;User 集中窗口管理函数,如窗口的创建、撤销、移动、对话及各种相关函数;GDI 提供画图函数、打印函数。所有应用程序共享这 3 个模块的代码,每个 Windows 操作系统的 API 函数都通过函数名来访问,具体的做法是在应用程序中使用函数名,并与适当的函数库进行编译和链接,然后运行应用程序,实际上 Windows 将 3 个组件置于动态链接库中。Windows API 数量庞大,在 2000 年,Win32 API 全集函数超过 2 000 个,部分 Win32 API 由系统调用实现(需要调用内核函数),大部分 Win32 API 由库过程实现(用户空间中的库函数),而且某版本的系统调用会在另一个版本中由用户空间代码实现。从功能上来说,UNIX/Linux 和 Win32 API 的部分系统调用之间存在粗略的对应关系,表 1.1 列出其中一些。

表 1.1 UNIX/Linux 与 Win32 API 部分系统调用之对应关系

UNIX/Linux	Win32 API	说 明
fork	CreateProcess	创建进程
waitpid	WaitForSingleObject	等待进程终止
open/close	CreateFile/CloseHandle	创建或打开文件/关闭文件
read/write	ReadFile/WriteFile	读写文件
lseek	SetFilePointer	移动文件指针
mkdir/rmdir	CreateDirectory/Remove Directory	建立目录/删除目录
stat	GetFileAttributesEx	获取文件属性

4. 系统调用实现要点

操作系统实现系统调用功能的机制称为陷阱或异常处理机制。由于系统调用而引起处理器中断的机器指令称为访管指令(supervisor)、自陷指令(trap)或中断指令(interrupt),其中访管指令为非特权指令,在目态下执行时会将 CPU 转换到核心态。每个系统调用都事先规定编号,称其为功能号,发出访管、自陷或中断指令时,必须通过某种方式指明对应系统调用的功能号,在大多数情况下,还附带有传递给内核函数的参数。

系统调用的实现要点:一是编写系统调用服务函数;二是设计系统调用的人口地址表,每个人口地址都指向一个系统调用的内核函数,有些还包含系统调用自带参数的个数;三是陷阱处理机制,需要开辟现场保护区,以保存发生系统调用时的处理器现场。如图 1.8 所示是系统调用的

处理过程。

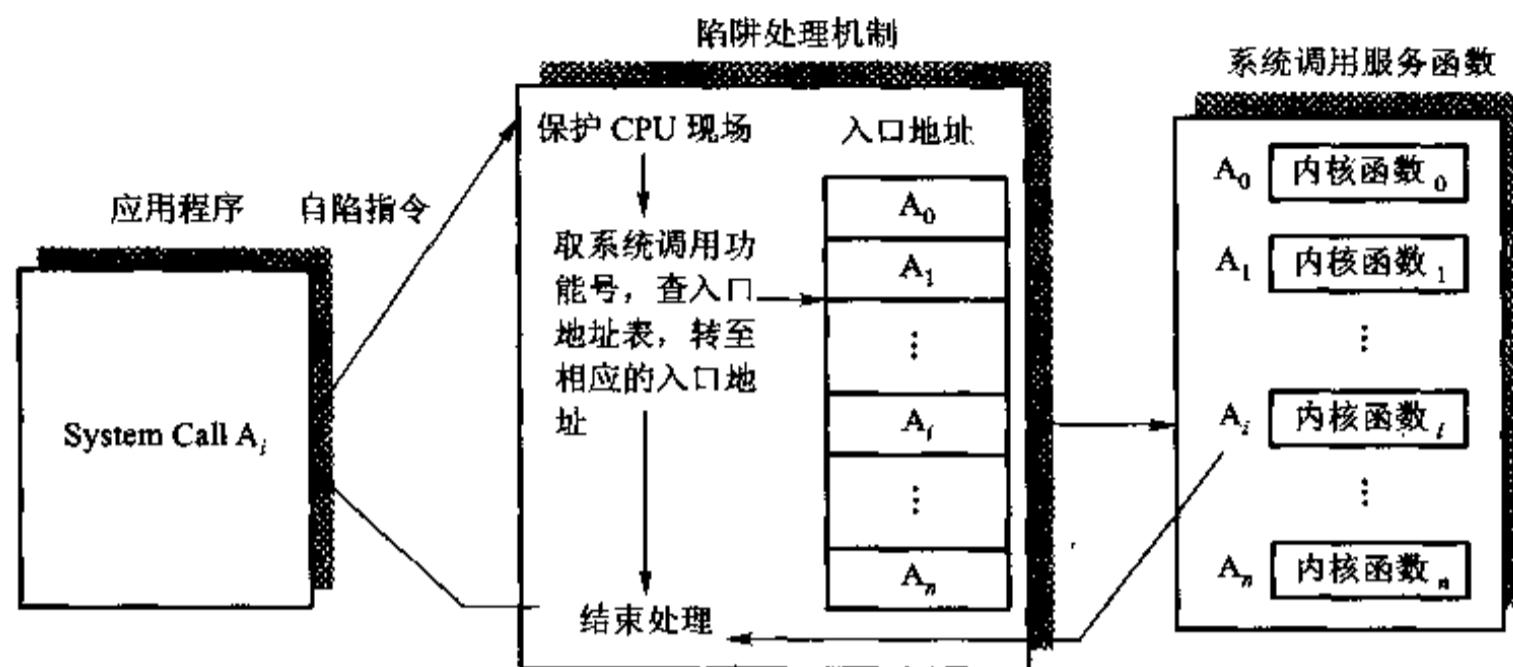


图 1.8 陷阱处理机制和系统调用的处理过程

参数传递是系统调用中需要处理的问题,不同的系统调用要向相应的内核服务函数传递不同的参数,反之,执行系统调用的结果也要以参数的形式返回给应用程序。实现应用程序和系统调用之间传递参数所采用的方法:一是访管指令或自陷指令自带参数,可以规定指令之后的若干单元存放参数,这叫做直接参数,或者在指令之后紧临的单元中存放参数的地址,这叫做间接参数,由间接地址指出参数的存放区;二是通过 CPU 的通用寄存器传递参数,这种方法不适用于传递大量参数,改进方法是:在主存的某个区或表中存放参数,将其首地址送入寄存器,实现参数传递;三是在主存中开辟专用的堆栈区传递参数。

5. 系统调用与函数调用之间的区别

在程序中执行系统调用或函数(过程)调用,虽然都是对某种功能或服务的需求,但两者从调用形式到具体实现都存在很大区别。

(1) 调用形式和实现方式不同

函数调用其所转向的地址是固定不变的,但系统调用中不包含内核服务函数入口,仅提供功能号,按功能号调用;函数调用是在用户态执行的,只能访问用户栈;系统调用要通过中断机制,从用户态转换到核心态,内核服务函数在核心态执行,并访问核心栈。

(2) 被调用代码的位置不同

函数调用是静态调用,调用程序和被调用代码处于同一程序内,经链接后可作为目标代码的一部分,这是用户级程序。当函数升级或修改时,必须重新编译和链接。系统调用是动态调用,系统调用的服务代码位于操作系统中,这是系统级程序。这样当系统调用处理代码的升级或修改时,与调用程序无关,而且调用程序的长度大为缩短,能减少其所占用的存储空间。

(3) 提供方式不同

函数通常由编程语言提供,不同语言所提供的函数的功能、类型和数量可以不同;系统调用

由操作系统提供,一旦操作系统设计好,系统调用的功能、类型和数量便固定不变。

1.3.3 作业接口与操作命令

1. 作业控制方式

用户通过操作控制命令和作业控制语言(命令)组织和控制作业的执行时,一般用到两个作业级接口——联机作业控制接口和脱机作业控制接口。

(1) 联机作业控制接口

此接口用于交互型作业处理,联机用户通过操作命令来调用系统功能,请求系统服务。操作系统负责提供一组命令及相应的命令解释程序。用户输入操作命令直接控制作业的执行,既可以一次输入一条命令,又可以预先编写批命令文件。每当系统接收一条命令,便立即转入命令解释程序,按此命令的要求控制作业执行。命令执行后,通过显示器向用户报告执行结果,用户根据此结果来决定下一步的操作。如此反复,通过人机交互方式控制作业的执行。

不同操作系统的命令接口有所不同,这不仅体现在命令的类型、数量及功能方面,也体现在命令的形式和用法等方面。不同的形式和用法组成不同的用户界面,用户界面可分成以下几种。

① 字符型用户界面

字符型用户界面通过命令语言来实现,可分为以下两种。

(a) 命令行方式

这种方式规定命令语言的词法、语法和语义,以命令为基本单位来完成预定的工作任务,完整的命令集构成命令语言,反映系统向用户提供的全部功能。命令以命令行的形式输入并提交给系统,命令行由命令动词和一组参数构成,指示操作系统完成规定的功能。命令的一般形式为

command arg1 arg2 … arg_n

其中,command 是命令名,又称命令动词,其后是此命令所带的执行参数。某些命令可以没有参数,如 Linux 常用命令可分成文件管理类、进程管理类、软件开发类及系统维护类等上百条。

(b) 批命令方式

在使用操作命令的过程中,有时需要连续使用多条命令,有时需要重复使用若干条命令,还有时需要有选择地使用不同命令,用户每次将命令由键盘逐条输入,既浪费时间,又容易出错。各种操作系统都支持称为批命令的特别命令,其实现思想是:规定批命令文件,这种文件有特殊的文件扩展名,例如,MS-DOS 约定其扩展名为 BAT。用户预先把一系列命令组织在 BAT 文件中,一次建立,多次执行,从而减少输入次数,方便用户操作,节省时间,降低出错几率。更进一步地,操作系统还支持命令文件使用一套控制子命令,可编写带形式参数的批命令文件,当批命令文件执行时,可以用不同的实际参数来替换形式参数。这类批命令文件可以执行不同的命令序列,大大增强命令集的处理能力。

② 图形用户界面

用户虽然可以通过命令行方式和批命令方式来获得操作系统的服务,控制自己作业的运行,但却需要牢记各种命令的动词和参数,严格地按照规定的格式输入命令,这样既费时又不方便。

于是,图形用户界面(Graphic User Interface,GUI)便应运而生,是近年来最为流行的联机作业控制接口形式。

GUI采用图形化操作界面,使用WIMP(Window,窗口;Icon,图标;Menu,菜单;Pointing device,鼠标)技术,引入图标将系统的各项功能、各个应用程序和文件直观、逼真地表示出来,用户通过选择窗口、菜单、对话框和滚动条来完成对作业和文件的控制和操作。用户不必死记硬背操作命令,能够轻松自如地完成各项工作,使计算机系统成为非常有效且生动有趣的工具。

20世纪90年代推出的主流操作系统都提供GUI。GUI的鼻祖首推Apple公司。施乐公司Palo Alto研究中心于1981年在Star 8010工作站操作系统中首先推出GUI，在此基础上，Apple公司于1983年研制成功商用GUI系统Apple Lisa和Macintosh。之后，微软公司的Windows、IBM公司的OS/2以及UNIX和Linux操作系统上纷纷推出GUI。为了促进图形用户界面的发展，制定了国际GUI标准，规定GUI的基本构件。最早由美国麻省理工学院开发的X-Windows已成为事实上的工业标准，许多系统软件如Windows NT、Visual C++、Visual Basic等，均可按照应用程序的要求自动生成GUI，大大地缩短应用程序的开发周期。

③ 新一代用户界面

随着个人计算机的广泛使用,缺乏计算机专业知识的用户数量随之增多,如何不断地更新技术,为用户提供形象直观、功能强大、使用简便、易于掌握的用户接口,便成为操作系统领域的一个热门研究课题。例如,具有沉浸式和临场感的虚拟现实应用环境目前已走向实际应用,把用户界面的发展推向新的阶段。多感知通道用户接口、自然化用户接口甚至智能化用户接口的研究也都取得一定的进展。

(2) 脱机作业控制接口

脱机作业控制接口用于批处理，用户使用 JCL 将其运行意图，即需要对作业进行的控制和干预，事先写在作业说明书上。然后，用户将作业连同作业说明书一起提交给操作系统。当调度这个批作业时，系统调用 JCL 处理程序对语句逐条解释执行。如果作业在执行过程中出现异常情况，系统会根据作业说明书上的指示进行干预。这样，作业一直在作业说明书的控制下运行，直到运行结束。

下面是 IBM 370 系统中使用 JCL 处理批作业的一个例子。用户 WILSON 的作业名为 HAROLD, 属于 B 类, 优先级为 6, 此作业需要编译和连接编辑, 源程序和数据都在穿孔卡片上, 编译和连接编辑的结果在行式打印机上输出, 且需要保存编译结果。那么, 这个批处理的作业可组织如下。

```
// HAROLD JOB WILSON, MSGLEVEL = (2,0), PRTY = 6, CLASS = B  
// COMP EXEC PGM = IEYFORT          /* 读出编译程序 IEYFORT 工作 */  
// SYSPRINT DD SYSOUT = A          /* 编译结果在行式打印机上列表打印 */  
// SYSLIN DD DSNAME = SYSL, DISP = OLD, VOLUME = SER = 123 /* 程序经编译后保存  
                                /* 到数据集 */
```

```

// SYSIN DD          /* 源程序卡片 */
<SOURCE PROGRAM CARDS>
/*
// GO EXEC PGM=FORTLINK    /* 执行连接编辑 */
// SYSPRINT DD SYSOUT=A    /* 连接编辑结果在行式打印机上列表打印 */
// FTOTF001 DD UNIT=SYSCP  /* 指出数据卡片在穿卡机上 */
...
// GO SYSIN DD          /* 数据卡片 */
<DATA CARDS>
/*
//                                         /* 分隔卡片 */
//                                         /* 作业结束 */

```

2. 命令解释程序

用户通过操作命令、会话语言或作业控制卡调用命令解释程序,其功能是接收用户所输入的命令,并解释执行命令。系统通常会保存一张命令动词表,其中记录着所有操作命令及其处理程序的人口地址信息,当一个新的交互型用户登录系统时,系统就自动地执行命令解释程序。

命令的实现有两种方式。一是命令解释程序包含命令的执行代码,一旦收到命令后,便转向相应的命令处理代码执行,在执行过程中常常使用“系统调用”帮助完成任务,由于用到终端进程的地址空间,故这类命令不宜过多;二是由专门的“实用程序”实现,在执行时把命令所对应的命令处理文件装入主存。很多操作系统把两者结合起来,像列目录、查询状态之类的简单命令由命令解释程序处理,像编译、编辑这样的复杂命令由独立的命令处理文件完成。

3. 实用程序

实用程序(utility)又称支撑程序。大多数用户只要求计算机解决自己的应用问题,对操作系统的特性、结构和实现过程并不感兴趣。实用程序虽非操作系统程序,但却是必不可少的软件,为应用程序的开发、调试、执行和维护解决共性问题或执行公共操作。于是,操作系统经常以操作命令的形式向用户提供实用程序,用户对操作系统的评价,不是看系统调用,而是看实用程序。所以,实用程序的功能和性能在很大程度上反映一个操作系统的功能和性能,作为在操作系统层实现的实用程序,从根本上来说要借助系统调用实现。实用程序具有文件管理、语言支持、状态修改、支持程序执行、通信等功能,为用户解决共性问题,如程序调试和排错、分类和合并、复制和转储、Web 浏览、电子邮件收发、远程登录、文字处理、电子表格管理、数据库管理、汇编、编译、画图等。

1.4 操作系统结构和运行模型

随着软件日趋大型化、复杂化,软件设计,特别是操作系统设计,呈现以下一些特征:一是复杂程度高,表现在功能繁多、程序规模庞大、接口复杂、并行度高等方面;二是生成周期长,从提出

系统制作要求、明确规范起，历经结构设计、模块设计、编码调试，直至整理文档，软件投入运行，需要若干年才能完成；三是正确性难以保证，大型操作系统有数十万、数百万甚至数千万行指令；参加研制的人员有数十、数百甚至数千之多，工作量之大、复杂程度之高是常人难以想象的。例如，美国麻省理工学院在 1963 年投入使用 CTSS 约有 32 000 行程序；其后一年出现的 IBM OS/360 含有超过百万条机器指令，共由 4 000 个模块组成，花费 5 000 人年；1975 年，由美国麻省理工学院和贝尔实验室开发的 MULTICS 增长到 2 000 万条机器指令；当今流行的自由软件 Linux 操作系统的内核较小，其源代码包含在约 4 500 个 C 语言和汇编语言文件中，存放于约 270 个子目录中，源代码约由 200 万行组成。Windows 2000 则由 2 500 名主要开发人员参与，系统含有超过 3 200 万行语句。Fred Brooks 描述他负责开发的 IBM OS/360 研制过程中的困难和混乱，谈及操作系统的开发就像一个泥潭，一群“史前动物”陷入其中而不能自拔，在每个修改版本中仍然隐藏着无数错误，这是 20 世纪 60 年代出现的软件危机的一个真实写照。

即使一个操作系统开发完成，它仍然是无生命的，必须继续开发系统中运行的大量应用程序；待应用程序问世之后，用户必须通过文件、培训及实践去学会操作和使用，这意味着用户所拥有并使用的是 10 多年前的操作系统技术，当一个操作系统投放市场后，硬件技术早已向前迈进，多处理机出现，主存容量更大，外设种类更多，使用多种文件系统，采用多种网络协议，配置多种数据库管理系统，于是操作系统的设计师们就急于扩展已有系统以利用新的硬件设施和软件技术。

因而，人们开始极其重视操作系统的软件结构和构造方法。软件危机推动软件工程学的研究，人们尝试采用软件工程方法，即系统的、规范的和可定量的技术和方法来开发、运行和维护操作系统，使得程序设计从手工作坊方式转向工程化生产。采用现代软件工程理论和方法构筑的操作系统还要达到正确性、高效性、可靠性、安全性、可扩展性、可移植性、分布计算和 POSIX 承诺等设计目标。

操作系统由于具有高度的动态性和随机性、逻辑上的并发性和物理上的并行性，其研究和开发必须从软件结构入手，力求设计出结构良好的系统程序。操作系统结构设计有三层含义：一是研究操作系统的整体结构，如功能如何分块，相互之间如何交互，并要考虑构造过程和方法；二是研究操作系统的局部结构，包括数据结构和控制结构；三是操作系统运行时的组织，如系统是组织成进程还是线程，在系统空间还是在用户空间运行，等等。

1.4.1 操作系统的构件和结构

通常把组成操作系统的基本单位称作构件。剖析现代操作系统，其基本单位除了内核之外，还有进程、线程、管程和类程。

1. 内核

(1) 内核的概念

现代操作系统都采用进程的概念，为了更好地处理系统的并发性、共享性和随机性，并使进程协调地工作，仅依靠计算机硬件所提供的功能是远远不够的。例如，进程的调度就不能用硬件

来实现,进程自我调度也是办不到的,必须有一组软件对处理器和硬件资源进行首次改造,以便为进程的执行提供良好的运行环境,这组软件就是操作系统内核。

内核(kernel)是一组程序模块,作为可信软件来提供支持进程并发执行的基本功能和基本操作,通常驻留在内核空间,运行于核心态,具有访问硬件设备和所有主存空间的权限,是仅有的能够执行特权指令的程序。有了内核的支撑,机器功能得到扩展,进程运行环境得到改善,安全性得到保证,系统效率得到提高。

(2) 内核的分类

① 单内核

从提高执行效率和满足性能需求等方面考虑,虽然有些内核的内部划分为层次或模块,但其运行时是一个大二进制映像,模块之间的交互通过直接调用其他模块中的函数来实现,这种内核称为单内核(monolithic kernel)。用户可以直接或通过系统库间接使用系统调用,此调用执行访管指令时被内核截获,此时处理器从用户态切换到核心态,系统执行相应的内核函数,为应用程序提供服务。

UNIX是单内核操作系统,Linux并未彻底地与UNIX决裂,它也是单内核操作系统,但引入加载模块和卸载模块机制,可以动态地装入和删除一个文件系统或设备驱动程序,能够克服单内核系统的某些缺点,在一定程度上解决了单内核的功能适应性、灵活性和可伸缩性问题。Linux的单内核结构如图1.9所示。Linus Torvalds曾经就结构问题做出解释:现代成功的操作系统基本上不具有微内核特性,因此,Linux也不必是微内核结构操作系统。

单内核操作系统有以下两种基本结构。

(a) 整体式结构

整体式结构的设计思想和步骤如下:把模块作为操作系统的基本单位,按照功能而非程序和数据的特性把整个系统分解为若干模块,还可再细分为子模块,每个子模块具有一定的独立功能,若干关联模块协作完成某项功能。从结构上来看,上层模块实现主控、调度功能;中层模块提供各种服务,执行相应的系统调用;下层模块是一组公用过程,用以支撑其上层模块的工作。明确各个模块之间的接口关系,模块间可不加控制地自由调用,数据大都作为全程量使用;模块之间需要传递参数或返回结果时,其个数和方式也可根据需要随意约定;然后,分别设计、编码、调试模块;最后,把所有模块连接成一个完整的系统。

整体式结构的缺点是:模块的独立性差,模块之间的牵连甚多,形成复杂的调用关系,甚至还存在循环调用,造成系统结构不清晰,正确性难以保证,可靠性降低,系

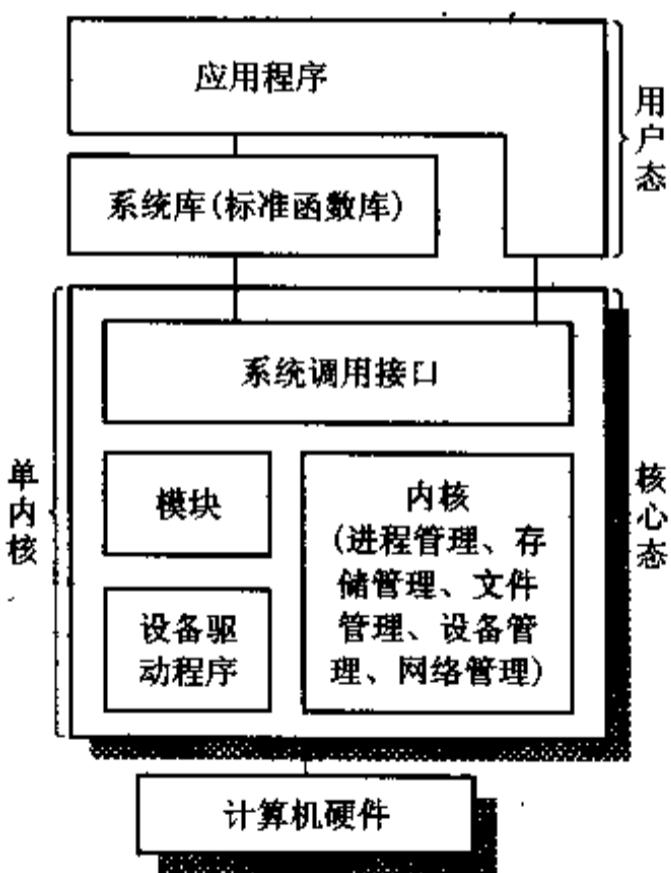


图1.9 Linux单内核结构

统功能的增加、删除、修改十分困难,这类操作系统的代表有 IBM 370 系列操作系统。整体式结构的优点是:结构紧密,组合方便,可通过组合不同的模块来满足不同环境和用户的不同需求,灵活性大;针对某个功能,可用最有效的算法并任意调用其他模块中的过程来实现,因此系统效率较高。此外,由于具有长期的发展历史,这种技术趋于成熟,可以吸取大量的经验和教训,导致 Linux 仍然采用整体式结构设计。

(b) 层次式结构

为了克服整体式结构的缺点而产生了层次式结构,其设计思路是:把操作系统划分为若干模块(或进程),这些模块(或进程)按照功能的调用次序排列成若干层次,各层之间必须是单向依赖或单向调用关系,即低层为高层服务,高层可以调用低层的功能,反之则不能。这样不但系统结构清晰,而且不会构成循环调用。

层次式结构的最大优点是把整体问题局部化,操作系统依照一定的原则分解成若干功能单一的模块(进程),这些模块(进程)组织成层次式结构,具有单向依赖性,使层次间的依赖和调用关系更为清晰和规范。上一(外)层功能是下一(内)层功能的扩充或延伸,下一(内)层功能为上一(外)层功能提供支撑和基础。因此,整个系统中的接口比其他结构方式的接口要少且简单,下一(内)层模块(进程)的设计是正确的,就为上一(外)层模块(进程)设计的正确性提供了基础,整个系统的正确性必可通过各层的正确性来保证,从而使系统的正确性大大提高。层次式结构的另一个优点是增加、修改或替换一个层次不会影响其他层次,有利于系统的维护和扩充。然而,层次式结构是分层单向依赖的,必须建立模块(进程)间的通信机制,系统花费在通信上的开销较大,就这一点而言,系统的效率会降低。

E.W.Dijkstra 于 1968 年发表的“THE 多道程序设计系统”中第一次提出操作系统的层次式结构设计方法,此系统是运行在荷兰 Electrologica X8 计算机上的共分 6 个层次的简单批处理系统。

② 微内核

操作系统仅将所有应用必需的核心功能放入内核,称为微内核(microkernel),其他功能都在内核之外,由在用户态运行的服务进程实现,通过微内核所提供的消息传递机制完成进程之间的通信,其结构如图 1.10 所示。

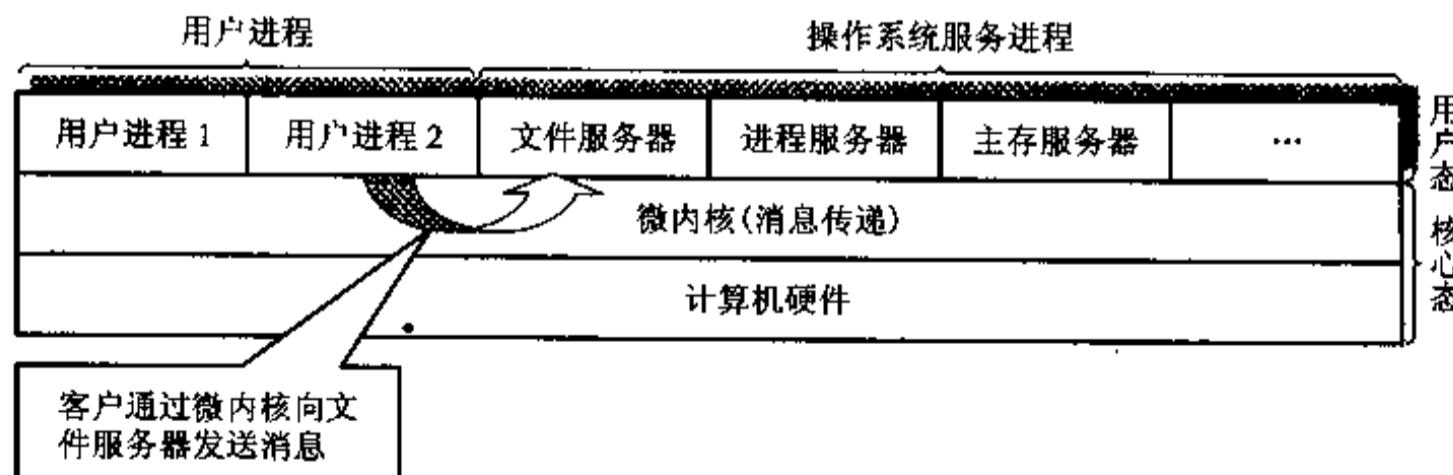


图 1.10 微内核结构

微内核结构的实现思想如下：将操作系统分成两部分，一是运行在核心态的内核，它提供系统的基本功能，如进程管理及调度、消息传递和设备驱动，内核构成操作系统的基本部分，只完成极少的核心态任务；二是运行在用户态并以客户—服务器方式运行的进程层。操作系统的其他部分由相对独立的若干进程来实现，每个进程完成一组服务，称为服务器进程，例如，提供文件管理服务、进程管理服务、存储管理服务、网络通信服务等，用户进程也在这一层运行，但不是操作系统的构件。由于进程具有不同的虚拟地址空间，客户和服务器进程之间采用消息传递机制进行通信，而内核被映射到所有进程的虚拟地址空间内，它可以控制所有进程。客户进程（应用进程）发出消息，内核将消息传送给相应的服务器进程，它实现客户所提出的服务请求，在满足要求后再通过内核发送消息把结果返回给客户。于是，客户进程与服务器进程形成客户—服务器关系。

微内核结构的优点：一是对进程的请求提供一致性接口，不必区分内核级服务和用户级服务，所有服务均借助消息传递机制提供；二是具有较好的可扩充性和易修改性，增加新服务或替换老功能只需增加或替换服务器；三是可移植性好，与特定 CPU 有关的代码均在微内核中，把系统移植到新平台所做的修改较小；四是为分布式系统提供有力的支撑，当消息从客户机发送给服务器进程时，不必知道它驻留在哪台机器上，客户的处理都是发送请求和接收应答。微内核结构的明显缺点是运行效率较低，这是因为进程间必须通过内核的通信机制才能进行通信。

操作系统的一个基本设计问题是内核的功能和结构设计，总体趋势是：内核应运行在具有特权的核心态，常驻主存，尽可能地小，仅确保实现操作系统正确、有效运转所必备的功能。由于内核很小，可被精心分析和设计成准确按意图执行的软件，称为可信软件（trusted software）。可信软件是最小化的操作系统功能集，提供支撑平台，操作系统的其他服务可由平台上运行的服务器进程实现。这种方法把内核同其外部的服务程序的开发分离，可根据特定的应用程序或运行环境要求定制服务器进程。虽然采用消息传递机制会影响系统执行速度，但却具有较好的可伸缩性和灵活性，适合构造分布式系统。

（3）内核的功能

一般而言，操作系统内核提供以下几种功能。

① 资源抽象

用软件抽象硬件资源，简化对其所执行的操作，屏蔽低层的物理细节，如提供设备驱动程序、创建虚拟设备等。抽象也可针对没有特殊硬件的资源进行。

② 资源分配

把所抽象的各种资源分配给多个应用程序使用，并负责回收资源。

③ 资源共享

根据资源的类型和特性，提供不同的机制以确保进程获得所需资源，允许进程共享资源并提供资源共享的同步和互斥机制。

Linux 操作系统内核由 6 个部分组成：进程调度与管理、主存管理和虚存管理、VFS 和文件管理、设备管理、两络接口和通信，用来实现资源抽象、资源分配和资源共享等功能。

(4) 内核的属性

内核的执行具有以下一些属性。

① 内核是由中断驱动的

只有在发生中断或异常事件时,才由硬件交换 PSW 引出操作系统内核工作,且在处理完中断或异常事件之后,内核自行退出。

② 内核是不可抢占的

传统操作系统的内核不可抢占,这意味着在内核中运行的进程即便其时间片已经用完,也不能被其他进程抢占,除非它自愿放弃 CPU。自愿放弃 CPU 通常是进程在等待资源或事件被阻塞时,或者是它已完成内核态操作而准备返回用户态时发生,自愿放弃 CPU 可确保内核处于一致性状态。所以,内核的不可抢占性能解决内核的大多数同步问题,特别是在单处理器系统中,当内核不存在抢占问题时,管理链表就无须加锁。但是由于内核的复杂性,还是需要原子操作、关中断和信号量等同步技术作为必备的手段。

为了改善实时性能,UNIX SVR4 对内核进行分段,定点提供可抢占性;Solaris 在 UNIX SVR4 的基础上设计完全可抢占式内核;Linux 2.6 也实现可抢占式内核,允许高优先级的任务以可抢占方式执行。

③ 内核部分程序在屏蔽中断状态下执行

虽然内核是不可抢占的,但却是可被中断的。所以,在处理某个中断时,为了避免中断嵌套可能引起错误,必须屏蔽这一级中断,甚至暂时屏蔽其他一些中断。Linux 引入中断处理程序的 bottom half(下半部分)概念,以尽可能地缩短中断处理屏蔽时间;类似地,在 Windows 中,设备中断服务程序 ISR 分两个阶段来处理,当设备中断发生且 ISR 被首次调用时,只在屏蔽中断状态下停留获得设备状态所必需的很短的一段时间,而把设备中断处理的主要工作留给排入 DPC 队列的处理程序完成,当退出此级别中断之后,在低级别中断状态下,执行 DPC 队列中的程序,完成对设备中断的其他处理工作。

④ 内核可使用特权指令

特权指令包括启动 I/O 操作、修改系统状态等,规定只允许这类指令在核心态下由内核使用,以防系统出现混乱,加强系统的安全性。

内核是操作系统对裸机的第一次改造,内核和操机组成了多台虚拟机,具有以下一些特性。

- ① 没有中断,进程设计者不再需要中断的概念,进程运行过程中无须处理中断。
- ② 为每个进程都提供一台虚拟机,进程好像在各自的私有处理机上顺序推进,实现多进程的并发执行。
- ③ 为进程提供功能强大的指令系统,即机器的非特权指令和系统调用所组成的新指令系统。

(5) 机制与策略分离

关于 UNIX/Linux 的设计有一句通用的格言“机制(mechanism)与策略(policy)分离”,意思是,操作系统的系统调用抽象并实现有特定目的和功能的函数(提供机制),至于如何使用这些函

数则与内核无关,而由上层软件(最上层是应用软件)决定(选择策略)。在机制与策略分离的操作系统中,应用问题的解决均可分成两部分:“提供并实现确定的功能(机制),将机制作为系统的可信软件来实现”和“如何使用这些功能(策略),可在不可信的环境中定义策略”。例如,一个主进程创建多个子进程并在其控制下运行,每个子进程实现不同的功能并处理不同的请求,主进程完全可以掌握哪个子进程最紧迫、最需要优先调度。那么,调度算法能否从应用进程接收有关的调度决策信息来做出最优的调度选择呢?解决此问题的方法是将调度机制从调度策略中分离出来,也就是将调度算法以某种形式进行参数化,而参数可由应用进程填写。具体的做法是:假设处理器采用优先级调度算法,提供可供应用进程改变或设置优先级的系统调用,这样,尽管主进程本身并不参与调度,但它却可以控制子进程被调度的细节。显然,调度程序是一种位于内核的进程调度机制,而调度策略(如何规定进程优先级)可由应用程序参与决定。另一个例子是允许将模块装载至内核,机制要解决的是模块如何被装入和链接,对其可发出怎样的系统调用;策略是确定允许由谁装载模块,也许只有超级用户来做,也许任何用户均可装载被某权威机构数字签名的模块。由程序中的独立部分来实现机制与策略,那么,这种软件易于开发,适应性好,还有助于内核保持短小和良好的结构。

2. 进程

进程能准确、动态地刻画计算机系统内部的并发性,解决系统资源的共享,在操作系统的發展史上,进程较早地被引入,在理论研究和设计实现上均发挥着重要的作用。进程使得操作系统的结构变得清晰,主要表现在:一个进程到另一个进程的控制转移由进程调度机制来统一管理,不能随意进行;进程之间的交互(如信号发送、消息传递和同步互斥等活动)由通信及同步机制完成,从而进程无法破坏其他进程的数据,每个进程相对独立,相互隔离,提高了系统的安全性和可靠性。因而,基于进程概念的操作系统的结构清晰,整齐划一,可维护性好。

3. 线程

早期,进程是操作系统中资源分配和系统调度的基本单位,拥有独立的地址空间和运行环境,进程之间进行通信和切换的系统开销相当大,限制了系统中并发运行的进程的数目。要更好地发挥硬件所提供的功能(如多 CPU),要实现各种复杂的并发应用,降低实现并发性所需付出的代价,单靠进程是无能为力的。于是,近年来流行多线程(结构)进程,亦称多线程。

在多线程环境中,进程是系统进行保护和资源分配的单位,线程则是进程中的一条执行路径,允许每个进程中多个线程,而线程是系统调度的独立单位。所以,可以把线程也看做一种构件,它是组成进程构件的更小的构件单位。在一个进程中包含多个可并发执行的控制流,而不是把多个控制流分散在多个进程中,这是并发多线程程序设计与并发多进程程序设计的主要不同之处。

4. 管程

管程是管理共享资源的一种同步机制,对管程的调用表示对共享资源的请求与释放,管程可被多个进程或管程嵌套调用,但是它们只能互斥地访问管程。管程应包含条件变量,当条件未被满足时,可以通过对条件变量做延迟操作使调用进程等待,直到另一个进程调用管程过程并执行

释放操作为止。管程的引入使得原来分散在进程中的临界区可集中起来统一控制和管理,这样可以方便地使用共享资源,使并发进程之间的交互作用更为清晰,便于用高级语言编写正确的并发程序。

5. 类程

类程用于管理私有资源,对类程的调用表示对私有资源执行操作,它只能被进程及起源于同一进程的其他类程或管程嵌套调用链所调用,其自身也可调用其他类程或管程。类程可以看做子程序概念的一种扩充,类程可包含多个过程;通常子程序只包含一个过程。

采用进程、管程、类程实现的操作系统中,进程在执行过程中若请求使用共享资源,可调用管程;若要控制私有资源的操作,可调用类程。1975年,Hansen 使用此方法成功地在 PDP 11/45 机上实现了单用户操作系统 Solo、作业流系统和过程控制实时调度系统等 3 个层次式结构的操作系统。

1.4.2 操作系统的运行模型

操作系统本身也是一组程序,从结构上来看,任何操作系统均可组织成函数的集合,这些函数屏蔽硬件细节,扩展底层功能,由对外可见的函数构成用户接口。那么,操作系统函数如何运行呢?这些函数是作为进程运行吗?人们称之为操作系统的运行模型,可分为独立运行的内核模型、在应用进程内执行的模型和作为独立进程运行的模型。

1. 独立运行的内核模型

这是早期操作系统的实现方式,系统的执行与应用进程不存在关联,如图 1.11 所示,它具有

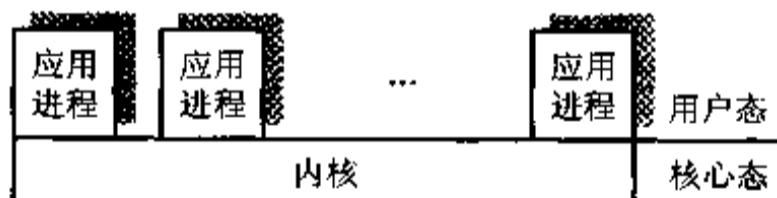


图 1.11 独立运行的内核模型

独立的存储空间,也可以访问应用进程空间。当发生中断、异常或系统调用时,当前运行进程的上下文将被保存到与进程相关的运行现场区域,内核接收控制权并开始执行。操作系统拥有自己的核心栈,用于控制函数的调用和返回,处理完成之后,根据具体情况,或是恢复被中断进程的现场并让其继续运行,或是转向进程调度指派另一个就绪进程运行。

在这种模型下,进程的概念仅仅是针对应用程序而言的,操作系统代码作为一个独立实体在内核模式下运行,因而,内核函数要并发执行是很困难的。

2. 在应用进程内执行的模型

为了提高内核函数的并发性,在创建应用进程时,同时为其分配一个核心栈,用于运行操作系统的内核函数,形成操作系统函数在应用进程内执行的方式,如图 1.12 所示。大部分的操作系统功能组织成一组内核函数供应用程序调用,操作系统的地址空间位于共享地址空间中,并不与应用程序的地址空间重叠,但被所有应用进程共享。进程在运行过程中产生异常或执行系统调用时,内核函数利用应用进程的核心栈完成相应的服务,只有由硬件中断调用的服务代码是一个例外,它在一个与所有进程无关的执行环境——操作系统内专门的中断上下文中执行。

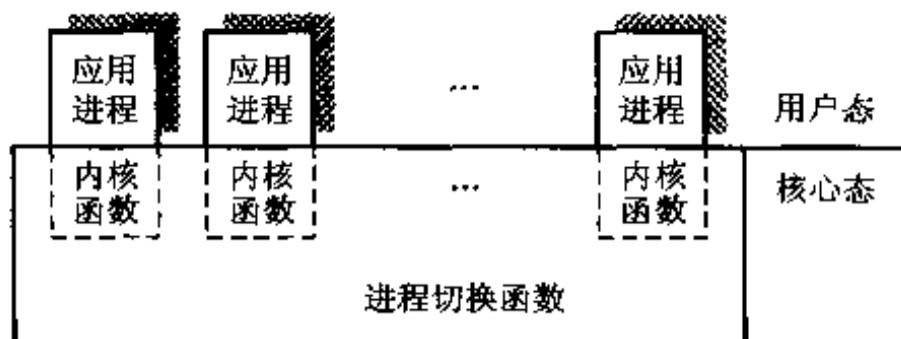


图 1.12 操作系统函数在应用进程内执行

当发生一次异常或系统调用时,处理器的状态将从用户态切换至核心态,控制权被传递给操作系统,应用进程的现场被保护,启用被中断进程的核心栈作为此后操作系统函数执行时的调用和返回栈区,此时发生模式切换,但进程上下文切换并未发生,还是认为程序在当前应用进程中执行。

当操作系统的内核函数完成工作之后,如果让当前进程继续运行,可执行一次模式切换来恢复原先被中断的用户进程继续执行,这种技术无须进程上下文切换就可让应用进程中途调用操作系统的函数(功能),运行完成后又可继续运行原来的应用程序。如果出现高优先级进程应发生进程切换的话,控制权就被传递给操作系统的进程切换函数,由其实现进程切换,让当前进程出让处理器,而指派另一个就绪进程占用处理器运行。

3. 作为独立进程运行的模型

操作系统函数作为独立进程执行的模型把操作系统组织成一组系统进程(也称服务器进程),操作系统的功能就是这些系统进程集合运行的结果,如图 1.13 所示,操作系统的大部分功能(进程切换和通信、底层存储管理、中断处理等)在核心态运行,大部分功能则被组织在一组分离的独立进程内实现,这组进程在用户态运行,而进程切换函数的执行仍然在进程之外。



图 1.13 操作系统函数作为独立进程执行

这种实现模型具备很多优点:首先,可采用模块化的操作系统实现方法,模块之间具有最少和最简洁的接口;其次,原来由内核实现的多数功能被组织成独立的进程在用户态运行,有利于操作系统的实现、配置和扩充,例如,性能监控程序用来记录各种资源的利用率和系统中用户进程的执行速度,因为监控程序并不向进程提供特别的服务,仅被操作系统调用,将其设计成一个服务器进程,便可赋予一定的优先级,插入其他进程中运行;最后,此结构在多处理器和多计算机环境下是非常有效的,操作系统的某些服务可指派到专门的处理器上执行,有利于系统性能的

改进。

1.4.3 Windows 2003 客户 - 服务器结构

Windows 2003 操作系统力图具备可扩充性、易移植性、兼容性、安全性和高性能；采用基于对象的技术来设计和实现，采用客户 - 服务器结构，应用进程和服务器进程之间通过执行体中的消息传递机制进行通信；硬件抽象层、内核和 I/O 设备管理器严格按分层设计；建立于内核基础上的执行体采用模块结构，根据需要可以相互调用。为了满足性能的要求，把文件服务、设备管理、图形引擎等功能放在内核中，并在核心态运行。如图 1.14 所示是 Windows 2003 系统结构图。

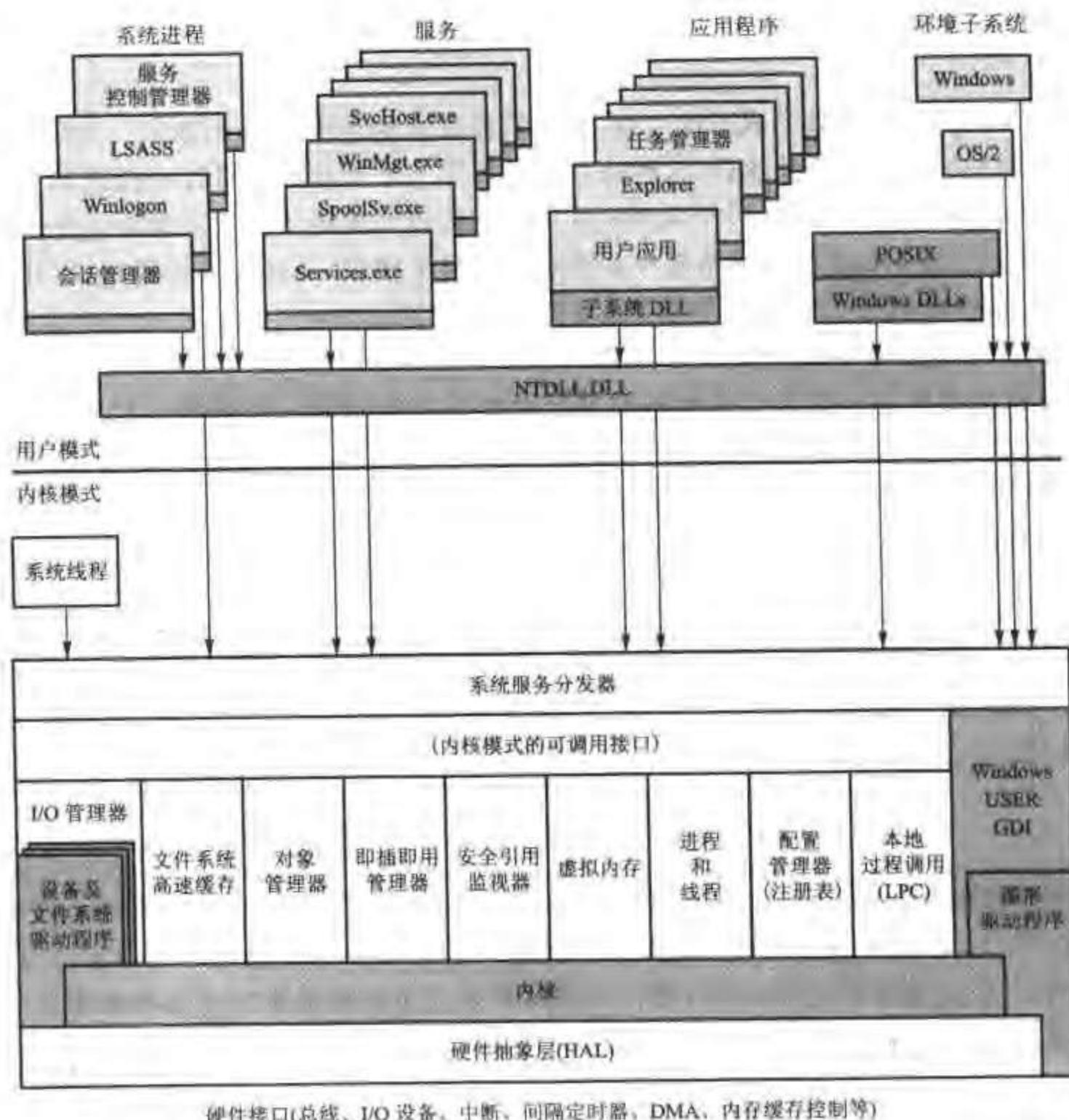


图 1.14 Windows 2003 系统结构图

1. 硬件抽象层

硬件抽象层(Hardware Abstract Level, HAL)直接操作硬件,是实现可移植性的关键组件,它是一个可加载的核心态模块,为运行在硬件平台的 Windows 提供低级接口。硬件抽象层隐蔽各种硬件细节,如系统总线、定时器、I/O 接口、DMA、中断控制器、多处理器通信机制等,还对 SMP 提供支持。硬件抽象层代码把操作系统内核、设备驱动程序及执行体与特殊硬件平台隔离开来。出于执行效率的考虑,I/O 及图形驱动程序可以直接访问硬件。

2. 内核

内核执行最基本的操作,主要功能有:线程的管理和调度;陷阱处理和异常调度;中断的处理和调度;多处理器同步;提供执行体所使用的基本内核对象。内核永远运行在核心态,为了保持高效率,代码短小且紧凑,不进行堆栈和传递参数检查,除了中断服务程序 ISR 之外,内核是不能被抢占的,也是 Windows 中唯一不能被分页的部分。

图形驱动程序被移入内核工作,旨在提高运行效率、加快画图速度。

3. 设备驱动程序

设备驱动程序是可加载的核心态模块,是 I/O 系统和相关硬件设备之间的接口,它不直接操作硬件,而是调用硬件抽象层来控制硬件接口,还把应用程序的 I/O 调用转换为特定硬件设备的 I/O 请求。设备驱动程序可分成硬件设备驱动程序、文件系统驱动程序、过滤器驱动程序、实现远程 I/O 请求的文件系统驱动程序。

4. 执行体

Windows 2003 操作系统分为两层:上层是执行体,下层是内核。执行体所包含的组件有:进程/线程管理器;虚拟主存管理器;安全访问监视器;I/O 管理器;高速缓存管理器。执行体还包括 4 组支持函数供执行体组件使用,它们是:对象管理器;本地过程调用(Local Procedure Call, LPC)机制;一组公用的“运行时”函数;执行体支持函数。

5. 系统支持库

NTDLL.DLL 是特殊的系统支持库,用于子系统的动态链接,包含两类函数:

(1) 作为执行体系统服务的接口供用户态调用的函数;

(2) 子系统、子系统动态链接库及其他本机映像使用的内部支持函数,如映像加载程序、堆管理程序和 Win32 子系统进程通信函数、通用运行时库函数、异步过程调用管理器和异常调度器。

6. 系统支持进程

系统支持进程均为用户态进程,有以下几种。

(1) Idle 进程

这是系统空闲进程,进程 ID 为 0,每个 CPU 都有一个 Idle 进程。Idle 进程包含一个相应的线程,用来统计空闲的 CPU 时间。

(2) System 进程

System 进程 ID 为 2,是一种特殊类型的 System 线程的宿主进程,它具有一般用户态进程的

属性和描述表,但只运行在核心态,执行加载至系统空间的代码,被用于发布、等待 I/O 和其他对象、轮询设备等场合。

(3) 会话管理器 SMSS.exe

这是由核心 System 线程在系统中创建的第一个用户态进程,用于执行关键的系统初始化步骤,包括创建 LPC 端口对象、设置系统环境变量、打开已知的动态链接库、启动 Win32 子系统进程的核心部分、加载 WinLogon 进程等。

(4) Win32 子系统 CSRSS 进程

这是用户态 Win32 子系统的部分,负责向应用程序提供调用接口,创建或删除进程、线程和 16 位的虚拟 MS-DOS 环境。

(5) 登录进程 WinLogon.exe

用于处理用户的登录和注销。

(6) 本地安全身份鉴别服务器进程 LSASS

接收来自于 WinLogon 进程的身份验证请求,调用适当的身份验证包来执行实际验证。当身份验证成功时,它将为用户进程生成安全访问令牌。

7. 服务控制器及服务进程

服务控制器是一个特殊的系统进程,负责服务的启动、停止和交互,并管理一系列用户态进程服务。类似于 UNIX 的守护进程,服务程序是合法的 Win32 映像,这些映像调用特殊的 Win32 函数以便与服务控制器交互,如注册、启动、响应状态请求、暂停或关闭服务。一些 Windows 组件是作为服务来实现的,如事件日志、假脱机、RPC 支持和各种网络组件。

8. 环境子系统

环境子系统的作用是将执行体系统服务的某些子集提供给应用程序,向应用程序展示本地操作系统服务,提供操作系统环境或个性。环境子系统有 3 个:Win32、POSIX 和 OS/2,每个环境子系统都有动态链接库,其中,Win32 比较特殊,它必须始终处于运行态,否则 Windows 就不能工作。Win32 组件有 Win32 环境子系统进程、核心态设备驱动程序、图形设备接口、子系统动态链接库及图形设备驱动程序等。

环境子系统又称虚拟机,是 Windows 操作系统实现兼容性的组件,其任务是接管操作系统的每个二进制代码请求,将它们转换为 Windows 能成功执行的相应指令。Windows 与其他软件的兼容性包括:与 DOS、OS/2、LAN Manager 和符合 POSIX 规范的系统的兼容性,及与多种文件系统和多种网络的兼容性。

9. 应用程序

应用程序可以是 Win32、Windows 3.1、MS-DOS、POSIX 或 OS/2 这 5 种类型之一,在 Windows 中,应用程序不能直接调用本地 Windows 服务,但能通过“子系统动态链接库”来调用相应的服务,子系统 DLL 的作用是将公开的调用接口转换为系统内部的 Windows 系统服务调用。

1.5 流行操作系统简介

1.5.1 Windows 操作系统

1. Windows 操作系统概况

微软公司成立于 1975 年,目前已经成为世界上最大的软件公司,其产品覆盖操作系统、编译系统、数据库管理系统、办公自动化软件和因特网支撑软件等各个领域。从 1983 年 11 月宣布 Windows 诞生到当前的 Windows Server 2003,已经走过 20 多个年头,成为风靡全球的微型计算机操作系统,微软公司几乎垄断了个人计算机软件行业。

1992 年 4 月 Windows 3.1 发布后,Windows 逐步取代 DOS 开始在全世界范围内流行。1995 年 8 月推出 Windows 95,随后相继推出 Windows 98 和 Windows Me(Microsoft Windows Millennium Edition)等家用操作系统版本。Windows 商用版本有 Windows NT,主要运行于小型机和服务器,Windows NT 3.1 于 1993 年 8 月推出,之后相继发布 NT 3.5、NT 3.51、NT 4.0 等版本。基于 Windows NT 5.0 内核,于 2000 年 2 月正式推出 Windows 2000,2001 年 3 月微软公司正式宣布把家用操作系统版本 Windows 98 和商用操作系统版本 Windows 2000 合二为一,新的 Windows 操作系统命名为 Windows XP(eXPerience)。

另外,Windows 还有嵌入式操作系统系列,包括 Windows CE(Consumer Electronics)、Windows CE.NET、Windows NT Embedded 4.0 和 Windows XP Embedded 等,其中,Windows CE 是 Windows 家族中最小的成员。

2. Windows NT 的技术特点

在 Windows 的发展过程中,硬件技术和系统性能在不断地进步,如基于 RISC 芯片和多处理器结构微型计算机的出现,客户—服务器模式的广泛采用,微型计算机存储容量的增大及配置多样化,对微机系统的安全性、可扩充性、可靠性、兼容性等提出更高的要求。1993 年推出 Windows NT(New Technology)便是为这些目标而设计的。Windows 产品除了上述功能之外,还具有以下技术特点:支持 SMP 和多线程,支持抢先可重入多任务处理、页式虚拟存储管理,支持多种应用程序接口,支持多种可装卸文件系统,具有容错功能、良好的可移植性和集成网络计算功能,等等。

3. Windows 2000/XP

Windows 2000 是在 Windows NT 5.0 的基础上修改和扩充而成的,能够充分发挥 32 位微型计算机的硬件能力,在处理速度、存储能力、多任务和网络计算支持诸方面与小型机竞争。Windows 2000 除了继承 Windows 98/NT 的特性之外,在与因特网连接、标准化安全技术、工业级可靠性和性能、支持移动用户等方面具有新的特征,它还支持即插即用和电源管理的功能,提供活动目录技术因特网应用软件服务。

基于 Windows NT,Windows XP 于 2001 年发布,它把家用操作系统和商用操作系统融为一体

体,结束 Windows 的双轨历史,具有一系列新特性:新的图形用户界面、整合防火墙、媒体播放器,更多的防止应用程序出现错误的手段,增强 Windows 安全性,简化系统的管理与部署,并革新远程用户工作方式。

4. Windows Server 2003 和 Windows XP 64 – Bit Edition

Windows Server 2003 是 Windows 多任务服务器操作系统,以集中或分布的方式实现各种服务器角色,其中包括:文件和打印服务器、Web 服务器和 Web 应用程序服务器、邮件服务器、终端服务器、远程访问/虚拟专用网络服务器、目录服务器、域名系统、流媒体服务器、主机配置协议服务器和 Windows 因特网命名服务器。Windows Server 2003 是一个可靠、安全、高效和具备联网功能的优质服务器操作系统,其新功能和改进有:服务器集群支持、可伸缩 SMP 支持、32/64 位处理器支持、公共语言运行库、提供 IIS 6.0 (Internet Information Services, 因特网信息服务)、XML Web 服务和 .NET 等。

客户机操作系统 Windows XP 64 – Bit Edition 运行在 Intel Itanium 处理器上,提供可伸缩的高性能平台,能够满足机械设计和分析、电影特效制作、3D 动画、工程和科学应用等需要超大主存和超强浮点运算性能的场合,主存是最大达 16 GB 的物理主存和 8 TB 虚拟主存。使用 Windows 程序设计模型,开发人员可用一种代码库创建 32 位和 64 位两种版本的应用程序。

1.5.2 UNIX 操作系统家族

1. UNIX 操作系统的历史及发展

UNIX 操作系统是一个通用、交互型的分时操作系统,最早由美国 AT&T 公司贝尔实验室的 Kenneth Lane Thompson 和 Dennis MacAlistair Ritchie 于 1969 年开发成功,1971 年 UNIX 操作系统被移植到 PDP – 11 上。1973 年, Ritchie 在 BCPL (Basic Combined Programming Language) 语言的基础上开发出 C 语言,并用 C 语言重写 UNIX 操作系统,这对于其后的发展产生重要的作用,为 UNIX 操作系统的迅速推广和普及迈出了决定性的一步。1974 年 7 月,“The UNIX Time-Sharing System”一文在美国权威杂志 Communications of ACM 上发表,引起公众的注意。外界最早可获得的是 UNIX 第 6 版,1978 年的 UNIX 第 7 版是当今 UNIX 的先驱,此版为 UNIX 的繁荣奠定了基础。20 世纪 70 年代中后期 UNIX 源代码的免费扩散引起很多大学、研究机构和公司的兴趣,大众的积极参与对 UNIX 操作系统的改进、完善、传播和普及起到了重要作用。

BSD UNIX 一直在学术界占据主导地位,而美国 AT&T 公司的 UNIX 则成为商业领域的领跑者。AT&T 公司所发布的 UNIX 版本包括 System III、System V、System V.2 和 System V.3。美国加州大学伯克利分校开发的 UNIX BSD (Berkeley Software Distribution) 的著名版本有 4.xBSD,最后版本是 1993 年发布的 4.4BSD。这些版本中蕴涵页式虚存管理、长文件名、快速文件系统、套接字、网络协议 TCP/IP 等大量先进技术,在 UNIX 的发展历程中起到重要的作用,成为教学、科研、商用的两大主流系统之一。著名的 Sun OS 及其 Solaris 就是基于 BSD 的。1989 年,4.3BSD 的代表 Sun OS 和 AT&T UNIX SVR 3.2 会合于 SVR 4.0 (System V Release 4) 版

本,成为事实上的标准,是最重要的 UNIX 变种,以完整的、有商业竞争力的方式被推出。

UNIX 取得成功的最重要原因是系统的开放性,公开源代码,可方便地向 UNIX 系统中添加新功能和工具,使系统越来越完善,成为有效的程序开发支撑平台,UNIX 是目前唯一可以安装和运行在从微型计算机、工作站直到大型机和巨型机上的操作系统。UNIX 操作系统的主要特点如下:

- (1) 多用户多任务操作系统,用 C 语言编写,具有较好的易读性、易修改性和可移植性;
- (2) 结构可分为核心部分和应用子系统,便于做成开放系统;
- (3) 具有分层可装卸卷的文件系统,提供文件保护功能;
- (4) 提供 I/O 缓冲技术,系统效率高;
- (5) 抢占式动态优先级 CPU 调度,有力地支持分时功能;
- (6) 命令语言丰富齐全,提供功能强大的 Shell 语言作为用户界面;
- (7) 具有强大的网络与通信功能;
- (8) 请求分页式虚拟存储管理,主存的利用率高。

实际上,UNIX 已成为操作系统的一种标准,而不是指特定的操作系统,许多公司和大学都推出自己的 UNIX 系统,到 20 世纪 90 年代,UNIX 版本已多达 100 余个,且互不兼容,局面非常混乱。为了使同一个应用程序能够在所有不同的 UNIX 版本上运行,IEEE 拟定一个 UNIX 标准,称作 POSIX,它定义相互兼容的 UNIX 操作系统所必须支持的最少系统调用接口和工具。此标准已被多数 UNIX 操作系统支持,同时,其他操作系统也都支持 POSIX 标准,这都进一步推动了 UNIX 的发展。

在计算机的发展历史上,没有哪个程序设计语言像 C 语言那样得到如此广泛的应用,也没有哪个操作系统像 UNIX 那样获得普遍的青睐和应用,它们对整个软件技术和软件产业都产生深远影响,为此,Ritchie 和 Thompson 共同获得 1983 年度的 ACM 图灵奖(ACM Turing Award)和软件系统奖(Software System Award)。

2. Solaris 操作系统

Sun 微系统公司开发的 Solaris 是 BSD UNIX 的变种,是一个可移植的操作系统,既可运行在 Sun 和 SPARC 平台上,也可运行在 Intel x86、AMD64 和 EMT64 平台上。

Solaris 主要追求 3 个目标:一是高性能,通过多用户、多任务、SMP 和多线程等能力,最大限度地发挥硬件的潜力;二是满足用户不断变化的需求,建立分布式客户—服务器解决方案,允许企业用户利用计算环境的能力和资源,更加高效地解决业务问题;三是支持异构计算环境,用户可从现有的信息技术投资中获得最大回报。

Sun 公司操作系统的早期版本称为 Sun OS。1982 年推出的 Sun OS 1.0 是 4.1 BSD 的移植版本,运行在 MOTOROLA 680x0 平台上,支持 1 个 MIPS 的处理器和约 1 MB 主存。1985 年推出 Sun OS 2.0 版,支持分布式、基于网络的计算和远程过程调用及网络文件系统(Network File System,NFS),提供在异种机型、异种操作系统的网络环境下共享文件的简单而有效的方法。1988 年推出 Sun OS 4.0,把运行平台从 MOTOROLA 680x0 迁移到 SPARC 平台,并开始支持

Intel 平台;1990 年推出 Sun OS 4.1,1992 年推出 Sun OS 4.1.3,在这些版本中出现对异步对称式多处理器 ASMP 的支持,其内核每次只在一个处理器上运行,同时在可用的任意多个处理器上调度应用程序。

1992 年 Sun 公司发布 Solaris,作为系列化的操作系统,Sun OS 4.x 版本称为 Solaris 1.x,接着 Sun 公司操作系统的开发转向 System V Release 4,并取新名称为 Solaris 2.0(内核版本为 Sun OS 5.0)。从 1992 年开始,Sun 公司相继推出 Solaris 2.1~Solaris 2.6 版本(内核版本为 Sun OS 5.6),具备越来越多的功能。在 1998 年后,Sun 公司推出 64 位操作系统 Solaris 7 和 8,在网络特性、可靠性、兼容性、互操作性、易于配置和管理等方面均有很大的改进。Solaris 9 于 2002 年发布,Solaris 10 于 2005 年发布,其内核版本是 Sun OS 5.10。

3. FreeBSD 操作系统

FreeBSD 是运行于 Intel 平台的优秀 UNIX 类操作系统自由软件,是著名的 BSD UNIX 的一个继承者,由 Bill Jolitz 和 Nate Williams 等人研发,1993 年 12 月发布 FreeBSD 1.0,1995 年 1 月发布 FreeBSD 2.0,其后相继推出 FreeBSD 2.1、2.2 和 3.0。FreeBSD 操作系统在因特网上的应用也越来越多,尤其对于要求高性能、高可靠性的网络服务器系统,提供了一个极具诱惑力的选择。例如,Walnet Creek 使用 FreeBSD 建成因特网上最大、最繁忙的匿名文件服务器——ftp.cdrom.com。

现存的系统有 FreeBSD 2.2.x-stable、FreeBSD3.x-stable、FreeBSD4.0-current 等版本。FreeBSD 不含 16 位代码,是真正的 32 位多用户多任务操作系统,其主要特性有:可调整的动态优先级抢占式多任务能力,全面支持 TCP/IP 协议,可用做 Internet/Intranet 服务器,提供 NFS、FTP、电子邮件、万维网和防火墙能力,提供工业标准 X-Window 及支持在 Intel x86 上运行的其他 UNIX 操作系统的二进制执行文件,等等。

1.5.3 自由软件和 Linux 操作系统

1. 自由软件

自由软件(free software,又称 freeware)是指遵循通用公共许可证(General Public License, GPL)规则,保证使用上的自由,获得源程序的自由,可以自行修改的自由,可以复制和推广的自由,也可以有收费的自由的一种软件。

自由软件的出现其意义深远,众所周知,科技是人类社会发展和进步的阶梯,而科技知识的探索和积累是组成这个阶梯的逐级台阶,人类社会的发展以知识的积累为依托,不断地在前人所获知识的基础上发展和创新才得以逐步地提高。软件产业也是如此,如果能把已有的开发成果加以利用,避免重复开发,将大大提高软件的生产率,借鉴他人的开发经验,互相利用,共同提高。带有源程序和设计思想的自由软件对于学习和进一步开发软件起到了极大的促进作用,自由软件的定义决定它是为了人类科技的共同发展和交流而出现的。

自由软件赋予人们极大的自由空间,这并不意味着它是完全无规则的,自由软件是“贡献型”,而不是“索取型”,只有人人贡献出自己的一份力量,自由软件才能得以健康发展。自由软件

之父 Richard Stallman 先生于 1984 年组织开发一个软件体系计划 GNU,GNU 的含义是 GNU is not UNIX, 同时拟定通用公共许可证协议 GPL, 并成立自由软件基金会 (Free Software Foundation, FSF)。

GNU 写出一套同 UNIX 兼容同时又是自由软件的 UNIX 系统, 完成大部分的外围工作, 包括外部命令 gcc/gcc++ 和 shell 等, 最终 Linux 内核为 GNU 工程划上一个完美的句号。目前人们熟知的软件如 gcc/gcc++ 编译器、Objective C、FreeBSD、Open BSD、FORTRAN77、C 库、BSD E-mail、BIND、Perl、Apache、TCP/IP、IP accounting、HTTPserver、Lynx Web 都是自由软件的经典之作和知名软件。

GPL 协议可看做一个伟大的协议, 是征求和发扬人类智慧和科技成果的宣言书, 是所有自由软件的支撑点, 没有 GPL 就没有当今的自由软件。

2. Linux 操作系统

Linux 是由芬兰籍科学家 Linus Torvalds 于 1991 年编写完成的一个操作系统内核, 当时他还是芬兰首都赫尔辛基大学计算机系的学生, 在学习“操作系统”课程的过程中, 动手编写一个内核原型, 从此, 一个新的操作系统诞生, Linus 本人把这个系统按自由软件版权在因特网上发布, 许多人对这个系统进行改进、扩充和完善, 其中不乏做出关键性贡献的人物, Linux 由最初一个人所写的原型变成在因特网上由无数志同道合的编程高手们参与的一场运动。

迄今为止, Linux 操作系统已得到广泛使用, 许多计算机公司如 IBM、Intel、Oracle、Sun 等都大力支持 Linux 操作系统, 各种成名软件纷纷移植到 Linux 平台上, 运行在 Linux 之上的应用软件越来越多, 如今 Linux 中文版已经开发出来, 同时, 也为发展我国的自主操作系统提供了良好的条件。从 Linux 的发展史可以看出, 是因特网孕育了 Linux, 没有因特网就不可能有 Linux 今天的成功, 从某种意义上来说, Linux 是 UNIX 和因特网相结合的产物。自由软件 Linux 是一个充满生机、已拥有巨大用户群和广泛应用领域的操作系统, 它是目前唯一能与 UNIX 和 Windows 较量与抗衡的一个操作系统。

Linux 操作系统的技术特点如下: 多用户多任务 32 位通用操作系统, 内置通信联网功能, 支持 TCP/IP 协议, 符合 POSIX1003.1 标准, 支持一系列 UNIX 开发工具, 提供强大的管理功能和远程管理功能, 支持 32 种文件系统, 提供多种图形用户界面及开放源代码, 有利于发展各种特色的操作系统。

1.5.4 IBM 系列操作系统

1. IBM 系列操作系统概述

国际商业机器公司 (International Business Machines Corporation, IBM) 成立于 1914 年, 是当今世界上最大的跨国 IT 公司之一, 在全球拥有数十万名雇员, 业务遍及 150 多个国家。计算机发展史上的许多重大革新是由 IBM 公司开创的, 第一个生产系列计算机、第一个研制出集成电路计算机、第一个制造出计算机用磁盘、开发出迄今为止的最小硬盘、开发出迄今为止的最快商用机。人类社会的许多重大事件都有 IBM 公司技术的参与, 1969 年阿波罗宇航器登月、1981 年

哥伦比亚航天飞机进入太空、1996 年 IBM 超级计算机“深蓝”战胜世界象棋冠军 Garry Kasparov, IBM 公司的 Blue Pacific 是迄今为止开发出来的最快的巨型机之一。

在过去的几十年里,IBM 公司研制开发和生产销售的 IT 产品包罗万象,品种繁多,从主机到外部设备、从系统软件到应用软件、从原始到先进、从简单到成熟,功能逐步完善,性能不断改进,IBM 公司的历史就是一部活生生的现代计算机发展史,回顾 IBM 操作系统的演变过程,可以看出现代操作系统的发展历程。随着硬件技术的不断发展,从计算机操作管理软件 FMS (FORTRAN Monitor System)、IBM SYS(IBM7094 Monitor)到 OS/360 操作系统和磁盘操作系统 DOS/360,及支持虚拟机的 VM/370 操作系统,微机操作系统 DOS 和 OS/2,以及今天的新型操作系统也得到很大的发展。下面仅介绍 IBM 公司目前在其各种机型中所开发和使用的部分操作系统。

2. AIX 操作系统

AIX(Advanced Interactive eXecutive, 高级交互执行)操作系统是超强设计的重负载高端 64 位 UNIX 操作系统,运行在 IBM RS/6000 系列服务器和 IBM 高端多处理器 RS/6000 SP 服务器集群产品上。AIX 操作系统于 1990 年推出,目前的最新版本是 AIX 5L,是一个具有可伸缩性、高安全性、高可靠性的软实时操作系统,支持广泛的工业标准和开放系统标准,支持多用户、多线程和动态装卸设备驱动程序,拥有特大的虚存空间,网络特性出色,管理工具多样,各种语言、商用 UNIX 软件大都可在其上运行。许多网站和研究中心都采用 RS/6000 及 AIX 系统,主要用于 FTP、电子邮件、Web 服务器、数据库服务器及计算机辅助设计、计算机辅助工程、可视化等各种科学和工程应用。

3. OS/390 操作系统

IBM S/390(System/390)是基于新一代 CMOS 电路的企业级服务器,运行 OS/390 操作系统、VM(Virtual Machine, 虚机器)操作系统和 DOS/VSE(Disk Operating System/Virtual Storage Extended, 扩充虚存环境磁盘操作系统)。随着 2000 年 12 月推出 IBM Z900 系列大型主机,2001 年 3 月又发布 OS/390 操作系统的更新版 ZOS。

IBM 系列计算机从 S/360、S/370 发展到 20 世纪 90 年代的 S/390,经过 40 余年的不断改进,IBM S/390 已成为具有高可靠性、可扩展性、安全性的现代大型计算机系统。据统计,目前全世界商用数据处理 70% 以上都运行在 S/390 企业级服务器上。中小型服务器所组合的公司网络因处理能力所限,导致维修成本过高,不能适应现代商业应用的需要。此外,电子商务的蓬勃发展大大刺激了对计算能力的需求,导致大型机市场再度升温,用一台或若干台大型机代替众多的中小型服务器已成为 IT 业务的大势所趋。这也反映计算机应用遵循“集中一分散一再集中”的曲折历程,S/390 大型机及其操作系统 OS/390 便在这样的大趋势下应运而生。最新一代 S/390 G6 是世界上第一个使用铜质互连芯片技术的企业级服务器,运行速度达 1600 MIPS。

OS/390 的前身是著名的 MVS(Multiple Virtual Storage, 多重虚拟存储器)操作系统,1996 年 IBM 公司发布 OS/390 1.1,1998 年 IBM 公司发布 OS/390 2.5,目前的最新版本是 OS/390 2.7。OS/390 是一个基于对象技术的现代开放式操作系统,支持 X/Open 标准,除了 UNIX 应用

程序可在 OS/390 上运行外,兼容 S/370 上开发的应用程序;它支持 TCP/IP 在内的多种通信协议,提供分布计算和 LAN 服务功能,集成防火墙提高网络的安全性;OS/390 采用面向对象程序设计技术、并行处理技术、分布式处理技术、客户—服务器技术,具有较强的可扩展性和可移植性。由于历史的原因,OS/390 有几种不同的运行方式:S/370 模式支持原 S/370 下运行的应用程序;MVS/ESA390(Enterprise Systems Architecture,企业系统体系结构)模式支持 10 个 240 MB 主存和 256 个通道的多处理器系统;ESA/390 LPAR(Logical Partitioning,逻辑分区)模式可以把计算机从逻辑上分成 10 个或更多的部分,每个部分都有自己的 CPU、主存和通道,且分别运行不同的操作系统,如 OS/370、MVS/ESA370 和 MVS/ESA390,也可运行 IBM 虚机操作系统 VM/ESA。

4. OS/400 操作系统

AS/400 服务器(Application System,应用系统)首次采用 64 位 RISC 技术,主要运行 OS/400 操作系统,也可运行 SSP(System Support Program,系统支持程序)操作系统。

AS/400 服务器是 IBM 公司开发的最新中型商用机,既可用在旅行、家庭、小型办公室中,也可用在大型商业应用中,特别适用于 C/S 计算。AS/400 上配置 OS/400 操作系统,在硬件层之上自底向上共设置 4 层软件:许可证内部代码、TIMI、OS/400、程序设计支持和应用支持。程序设计支持层提供 C、C++、Cobol、RPG、Java 等语言;应用支持层提供网络管理、工业应用、数据库和系统管理服务、多媒体应用和各种因特网应用。OS/400 主要提供以下功能:控制语言和菜单、系统操作员服务、程序员服务、工作管理、设备管理、数据管理、消息处理、通信和安全性保证。TIMI(Technology Independent Machine Interface)是逻辑上的(而非物理上的)把上层软件与低层软硬件完全隔离开来的一种系统接口。它提供一组应用程序接口,为 OS/400 和所有应用软件访问低层资源提供唯一的途径。

OS/400 是在 IBM AS/400 服务器上运行的专有多用户操作系统,其主要特色是内置 IBM DB2 数据库管理系统,此数据库对用户是透明的,使用起来方便、高效,最新版本是 DB2/400 的 4.4 版。在 AS/400 机器上,硬件、操作系统、数据库、联网、工具和应用软件紧密结合,从而给用户带来易用、可靠的优点。此外,OS/400 操作系统具有很高的安全性,已获得美国联邦机构的认证,目前全世界装机数达几千万台以上,适合于许多商用业务应用。

5. 微机操作系统

OS/2 是微软公司和 IBM 公司于 1987 年合作开发的配置在 PS/2 机上的图形用户界面操作系统,并于 1994 年推出高性能(新的文件系统和主存管理)、图形界面(Workplace Shell)、高速度、低配置的 32 位抢先多任务操作系统 OS/2 Warp,其功能和性能与 Windows 95 相当。1996 年,推出网络操作系统 OS/2 Warp Server 4.0 和 OS/2 Warp Server SMP,后者为对称式多处理器版,支持 TCP/IP 协议,支持 6 国语言的声音输入输出的 Navigator、Java 运行服务。

OS/2 的主要特点有:图形化用户界面、可在 16 位和 32 位两种 CPU 上工作,引入会话、进程、线程的概念来实现多任务控制,提供高性能文件系统,采用长文件名和扩展文件属性,提供应用程序编程接口(API)等。

1.5.5 其他流行操作系统

1. Mach 操作系统

Mach 的前身是由美国卡内基 - 梅隆大学的 Richard Rashid 开发的操作系统 Accent, 1984 年, 他在 Accent 的基础上开始实施称为 Mach 的第三代操作系统项目, 其目标是使 Mach 与 UNIX 兼容以便利用 UNIX 系统中的大量现有软件。这一研究计划得到美国国防部 ARPA 的支持, 从而使 Mach 系统得到进一步的发展, 相应的升级版本也确保与 BSD UNIX 系统完全兼容, 而这正是 ARPA 的一个重要战略目标。

Mach 的最早版本于 1986 年推出, 运行在具有 4 个 CPU 的 VAX 11/784 系统上, 接着完成向 IBM PC/RT 及 Sun3 的移植工作。在 1987 年, Mach 被移植到 Encore 及 Sequent 多处理器环境。此后不久, 在计算机供应商 IBM、DEC 和 HP 的联合领导下成立开放软件基金会 (Open Software Foundation, OSF) 联盟, 其目的是改变 UNIX 对用户的控制, 而 UNIX 的所属者 AT&T 公司当时正同 Sun 微系统公司紧密合作开发 System V。OSF 选择 Mach 2.5 作为它的第一个操作系统 OSF/1 的基础。Mach 2.5 的内核较庞大且非常单一, 这是因为系统内核中存在大量 BSD UNIX 代码的缘故。在 1988 年, 美国卡内基 - 梅隆大学将所有 Berkeley 代码都从内核移出, 放到用户空间中, 这样就使系统具有微内核以及纯粹由 Mach 所组成的特点。

Mach 的设计目标是: 为建造其他操作系统(如 UNIX 类操作系统)提供基础; 支持大型稀疏地址空间; 允许对网络资源的透明访问; 从系统与应用两个方面开发并行性; 使 Mach 可被移植到有更多数量的机器的系统中。这些目标中既包含研究也包含开发, 主要思想是在仿真现有系统(如 UNIX、MS-DOS 和 Macintosh 操作系统)的基础上探究多处理器和分布式系统。

Mach 所采用的主要技术有: 微内核结构, 使用面向对象程序设计方法, 引入进程、线程、主存、端口、消息等对象概念; 设计基本调度、传递调度和相关调度等高效的多处理器调度算法; 提供强大和灵活的基于分页的存储管理系统, 将机器相关部分和机器无关部分区分开来, 使得存储管理的可移植性非常好; 提供分布式共享存储, 是能够被所有进程共享的单一线性虚拟地址空间; 采用通信端口来支持异步消息传递、远程过程调用、位流以及其他不同风格的通信方式。

2. Mac OS 操作系统

Mac OS 操作系统是美国 Apple 公司推出的操作系统, 运行在 Macintosh 计算机上。Apple 公司的 Mac OS 是较早的图形化界面操作系统, 由于它拥有全新的窗口系统、强有力的多媒体开发工具和操作简便的网络结构而风光一时。其主要的技术特点是: 采用面向对象技术; 图形化用户界面; 虚拟存储管理技术; 应用程序间的相互通信; 强有力的多媒体功能; 简便的分布式网络支持; 丰富的应用软件。Macintosh 计算机的主要应用领域是: 桌面彩色印刷系统、科学和工程可视化计算、广告和市场经营、教育、财会和营销等。

3. NetWare 操作系统

Novell 公司是主要的网络产品制造商之一, 成立于 1983 年。其网络产品可以和 IBM、Mac、UNIX 及 DEC 系统并存和互操作, 组成开放的分布式集成计算环境, NetWare 是其开发的网络

操作系统。

NetWare 具有高性能文件系统,支持 DOS、OS/2、Mac 及 UNIX 文件格式;具有三级容错功能,可靠性高;具有良好的权限管理机制,安全性好;具有开放性,提供开放的开发环境。

4. 教学操作系统 Minix

UNIX 早期版本的源代码可免费获得并被广泛研究,许多大学的“操作系统”课程都讲解 UNIX 部分源码,在 AT&T 公司发布 UNIX 第 7 版时,它开始认识到 UNIX 的商业价值,并禁止大学在课程中研究其源码,以免商业利益受到侵害。许多学校遵守这一规定,就在课程中略去 UNIX 的源码分析而只讲操作系统原理,遗憾的是,只讲授理论使学生不能掌握操作系统的许多关键技术。为了改变这种局面,荷兰 Vrije 大学计算机系教授 A.S.Tanenbaum 决定开发一个与 UNIX 兼容,然而内核却是全新的操作系统 Minix。Minix 未借用 AT&T 公司一行代码,不受其许可证的限制,学生可通过它来剖析一个操作系统,研究其内部运作方式,其名称源于“小 UNIX”,因其非常简洁、短小,故称 Minix,最新版本为 Minix 3。

Minix 用 C 语言编写,为了确保可读性好,代码中加入数千行注释,可运行在多种平台上。Minix 一直恪守“Small is Beautiful”(小即美)的原则,早期 Minix 甚至不依靠硬盘就能运行。Minix 具有多任务处理能力,支持 3 个用户同时工作,支持 TCP/IP 协议,支持 4 GB 主存,提供 5 个编辑器及 200 个实用程序。

在 Minix 发布后不久,因特网上出现一个面向它的 USENET 新闻组,数以万计的用户订阅新闻,许多人都想向 Minix 中加入新功能或新特性,使之更强大、更实用,成百上千的人提供建议、思想,甚至源代码,而 Minix 作者数年来一直坚持不采纳这些建议,目的是使 Minix 保持足够的短小精悍,便于学生理解。人们最终意识到作者的意念不可动摇,于是芬兰学生 Linus Torvalds 决定编写一个类似于 Minix 的系统,但是它功能繁多,面向实用而非教学,这就是 Linux 操作系统。

1.6 本 章 小 结

操作系统是计算机系统中最重要的系统软件,可从三种观点来观察它:用户接口与服务用户的观点、扩展机与虚拟机的观点和资源管理与控制的观点。第一种观点指操作系统是用户和硬件之间的接口,它通过扩展机器功能、改造硬件设施来提供新的服务能力,使用户能方便、可靠、安全、高效地使用计算机;第二种观点指操作系统把硬件的复杂性与用户的使用隔离开来,通过在硬件上添加一层层软件来改造和增强计算机系统的功能,为用户提供比物理计算机效率更高、更易于使用的扩展机或虚拟机;第三种观点把操作系统看做资源管理程序,对资源进行抽象研究,找出各类资源的共性和个性,跟踪和监视各类资源的使用状况,协调对资源的使用冲突,提供使用资源的统一、简单的方法和手段,最大限度地实现各类资源的共享和提高资源的利用率。操作系统的资源管理功能主要包括:处理器管理、存储管理、设备管理、文件管理和网络与通信管理。

操作系统是资源管理程序,为了达到资源共享的目的,主要使用三种基本的资源管理技术——资源复用、资源虚化和资源抽象;三种最基础的抽象——进程抽象、虚存抽象和文件抽象,通过这些资源管理技术来创建虚拟机,在操作系统的设计、实现和使用中,自始至终贯穿着这些技术的应用。

操作系统是一个大型复杂的并发系统,并发性、共享性和随机性是其重要特征,并发机制支持多道程序设计,共享机制控制诸进程正确地使用软硬件资源。其中,并发性和共享性又是两个最基本的特征,并发和共享虽然能改善资源利用率和提高系统效率,但却引发了一系列问题,随机性使操作系统的实现更加复杂化,因而,设计操作系统时引进许多概念和设施来妥善解决这些问题。

多道程序设计技术是指将多个作业放入主存并使它们同时处于运行态,从宏观上看,作业均开始运行但并未运行结束;从微观上看,多个作业轮流占有 CPU 交替执行。采用多道程序设计技术能改善 CPU 的利用率,提高主存和设备的使用效率,充分发挥系统的并行性。早期的操作系统沿着 3 条主线发展:多道批处理系统、分时交互系统和实时处理系统。多道批处理系统着眼于让 CPU 和外部设备同时保持忙碌,提高作业的吞吐率和系统的效率。其关键机制是:在响应一个作业的处理结束信号时,CPU 将在主存中所驻留的不同作业之间切换。分时交互系统的设计目标是为用户提供方便的程序开发、调试环境并快速响应交互式用户的命令请求,但又要支持多用户同时工作,以降低系统的成本。其关键机制是:采用时间片轮转法,让 CPU 在多个交互式用户间多路复用。实时处理系统与分时系统相比往往局限于一个或几个应用,例如,数据库的查询和修改应用成生产过程控制实时应用,但同样有对于响应时间的要求,甚至某些实时应用有更加严格的时间限制。其关键机制是:事件或队列驱动机制,当系统接收来自外部的事件后,快速分析这些事件,驱动实时任务在规定的响应时间内完成相应的处理和控制。上述各类操作系统都要妥善解决各种资源的管理和调度问题,使得操作系统的功能变得愈加丰富和完整。

操作系统的基本类型有三种:批处理操作系统、分时操作系统和实时操作系统。凡具备全部或兼有两种功能的系统称为通用操作系统。随着硬件技术的发展和应用深入的需要,新形成的操作系统有:微机操作系统、网络操作系统、分布式操作系统和嵌入式操作系统。

操作系统要为应用程序的执行提供一个良好的运行环境,要为应用程序及其用户提供各种服务,这种服务是通过程序接口和操作接口来实现的。程序接口由一组系统调用组或,同时还分析 API、库函数和系统调用之间的关系;操作接口包括(键盘)操作控制命令和作业控制(语言)命令,这是用户向操作系统提交作业和说明运行意图的两个命令接口,分别用于向联机或终端交互式用户提供的联机作业控制方式和向批处理用户提供的脱机作业控制方式,所有计算机用户都通过这两种接口和两种操作方式与操作系统进行交互。

人们也可以从结构的观点观察操作系统,结构是影响软件正确性和性能的重要因素。操作系统是一个大型复杂的并发系统,为了开发操作系统,必须研究它的结构。从两方面来讨论操作系统的结构,一是剖析和研究操作系统的重要构件,包括内核、进程、线程和管程等,这些也是操作系统所涉及的重要的基本概念;二是考察构造操作系统的不同结构设计方法:单内核、微内核

及客户-服务器结构,其中单内核又分为模块组合法和层次分级法。

习 题

一、思考题

1. 简述现代计算机系统的组成及其层次结构。
2. 计算机系统的资源可分成哪几类? 试举例说明。
3. 什么是操作系统? 计算机系统配置操作系统的主要目标是什么?
4. 操作系统如何实现计算与操作过程的自动化?
5. 操作系统要为用户提供哪些基本的和共性的服务?
6. 试述操作系统所提供的各种用户接口。
7. 什么是系统调用? 可分为哪些类型?
8. 什么是实用程序? 可分为哪些类型?
9. 试述系统调用的实现原理。
10. 试述系统调用与过程调用之间的主要区别。
11. 试述 API、库函数与系统调用之间的关系。
12. 试解释脱机 I/O 与假脱机 I/O。
13. 为什么对作业进行批处理可以提高系统效率?
14. 举例说明计算机体系结构的不断改进是操作系统发展的主要动力之一。
15. 什么是多道程序设计? 多道程序设计技术有什么特点?
16. 简述实现多道程序设计所必须解决的基本问题。
17. 计算机系统采用通道部件后,已实现处理器与外部设备的并行工作,为什么还要引入多道程序设计技术?
18. 什么是实时操作系统? 试述实时操作系统的分类。
19. 在分时系统中,什么是响应时间? 它与哪些因素有关?
20. 试比较批处理操作系统和分时操作系统的不同点。
21. 试比较实时操作系统和分时操作系统的不同点。
22. 试比较单道和多道批处理系统。
23. 试述网络操作系统的功能。
24. 试述分布式操作系统的功能。
25. 试述嵌入式操作系统的发展背景及其特点。
26. 现代操作系统具有哪些基本功能? 请简单叙述之。
27. 试述现代操作系统的基本特性及其所要解决的主要问题。
28. 为什么操作系统会具有随机性特性?
29. 组成操作系统的构件有哪些? 请简单叙述之。
30. 什么是操作系统内核?
31. 列举内核的分类、属性和特点。

32. 解释单内核操作系统及其优、缺点。
33. 解释微内核及客户—服务器结构操作系统及其优、缺点。
34. 什么是层次式结构操作系统？说明其优、缺点。
35. 什么是模块式结构操作系统？说明其优、缺点。
36. 什么是虚机器操作系统？试对其作简单说明。
37. 从执行方式来看，试述操作系统的各种运行模型。
38. 分析下列操作系统使用了或具有哪些体系结构的特点：UNIX/Linux、Windows 2003、VM/370、Mach。
39. 试述 Windows 2003 操作系统的结构特点。
40. 试述 Windows 2003 操作系统的主要组件及其功能。
41. 试述 Windows 2003 的设备驱动程序类型，其各自的主要功能是什么？
42. 试分析 Windows 2003 达到了哪些设计目标？
43. 通用操作系统具有批处理和分时处理两种功能，试问这样做有何优点及特点？
44. 客户—服务器模型在分布式系统中很流行，它能够用于单机系统吗？
45. 解释操作系统资源管理的主要技术：资源复用、资源虚化和资源抽象。
46. 说明抽象资源与物理资源之间的区别，并列举两个例子。
47. 说明多级资源抽象，并列举两个例子。
48. 以驾驶汽车为例，说明如何应用抽象原理及抽象的重要性。
49. 什么是虚拟计算机？分析其组成。
50. 何谓 POSIX？试述 POSIX1003.1 的内容。
51. 试述 POSIX1003.1 与 Linux 操作系统之间的关系。
52. 试从资源管理的观点出发，分析操作系统在计算机系统中的角色和作用。
53. 试从服务用户的观点出发，分析操作系统在计算机系统中的角色和作用。
54. 试述操作系统是建立在计算机硬件平台上的虚拟计算机系统。

二、应用题

1. 有一台计算机，具有 1 MB 主存储器，操作系统占用 200 KB，各个用户进程分别占用 200 KB。如果用户进程等待 I/O 操作的时间为 80%，若增加 1 MB 主存空间，则 CPU 的利用率能够提高多少？
2. 在某个计算机系统中，有一台输入机和一台打印机，现有两道程序投入运行，且程序 A 先开始运行，程序 B 后开始运行。程序 A 的运行轨迹为：计算 50 ms、打印 100 ms、再计算 50 ms、打印 100 ms，结束。程序 B 的运行轨迹为：计算 50 ms、输入 80 ms、再计算 100 ms，结束。试说明：
 - (1) 两道程序运行时，CPU 是否空闲等待？若是，在哪段时间内等待？为什么等待？
 - (2) 程序 A、B 是否有等待 CPU 的情况？若有，指出发生等待的时刻。
3. 设有 3 道程序，按照 A、B、C 的优先次序运行，其内部计算和 I/O 操作时间如下所示。

A	B	C
$C_{11} = 30 \text{ ms}$	$C_{21} = 60 \text{ ms}$	$C_{31} = 20 \text{ ms}$
$I_{12} = 40 \text{ ms}$	$I_{22} = 30 \text{ ms}$	$I_{32} = 40 \text{ ms}$
$C_{13} = 10 \text{ ms}$	$C_{23} = 10 \text{ ms}$	$C_{33} = 20 \text{ ms}$

试画出多道运行的时间关系图(忽略调度执行时间)。完成 3 道程序共花费多少时间? 比单道运行节省多少时间? 若处理器调度程序每次进行程序转换费时 1 ms, 试画出各程序状态转换的时间关系图。

4. 在单 CPU 和两台 I/O(I_1, I_2)设备的多道程序设计环境下, 同时投入 3 个作业运行。其执行轨迹如下:

Job1: $I_2(30 \text{ ms}), \text{CPU}(10 \text{ ms}), I_1(30 \text{ ms}), \text{CPU}(10 \text{ ms}), I_2(20 \text{ ms})$

Job2: $I_1(20 \text{ ms}), \text{CPU}(20 \text{ ms}), I_2(40 \text{ ms})$

Job3: $\text{CPU}(30 \text{ ms}), I_1(20 \text{ ms}), \text{CPU}(10 \text{ ms}), I_1(10 \text{ ms})$

如果 CPU、 I_1 和 I_2 都能并行工作, 优先级从高到低依次为 Job1、Job2 和 Job3, 优先级高的作业可以抢占优先级低的作业的 CPU, 但不可抢占 I_1 和 I_2 。试求:

(1) 每个作业从投入 to 完成分别所需要的时间。

(2) 从作业的投入 to 完成, CPU 的利用率。

(3) I/O 设备利用率。

5. 在单 CPU 和两台 I/O(I_1, I_2)设备的多道程序设计环境下, 同时投入 3 个作业运行。其执行轨迹如下:

Job1: $I_2(30 \text{ ms}), \text{CPU}(10 \text{ ms}), I_1(30 \text{ ms}), \text{CPU}(10 \text{ ms})$

Job2: $I_1(20 \text{ ms}), \text{CPU}(20 \text{ ms}), I_2(40 \text{ ms})$

Job3: $\text{CPU}(30 \text{ ms}), I_1(20 \text{ ms})$

如果 CPU、 I_1 和 I_2 都能并行工作, 优先级从高到低依次为 Job1、Job2 和 Job3, 优先级高的作业可以抢占优先级低的作业的 CPU。试求:

(1) 每个作业从投入 to 完成分别所需要的时间。

(2) 从作业的投入 to 完成, CPU 的利用率。

(3) I/O 设备利用率。

6. 同第 5 题的条件, 每个作业的处理顺序和使用设备的时间如下:

Job1: $I_2(20 \text{ ms}), \text{CPU}(10 \text{ ms}), I_1(30 \text{ ms}), \text{CPU}(10 \text{ ms})$

Job2: $I_1(20 \text{ ms}), \text{CPU}(20 \text{ ms}), I_2(40 \text{ ms})$

Job3: $\text{CPU}(30 \text{ ms}), I_1(20 \text{ ms})$

试求:

(1) 每个作业从投入 to 完成分别所需要的时间。

(2) 每个作业从投入 to 完成 CPU 的利用率。

(3) I/O 设备利用率。

7. 若主存中有 3 道程序 A、B、C, 它们按照 A、B、C 的优先次序运行。各程序的计算轨迹为:

A: 计算(20 ms), I/O(30 ms), 计算(10 ms)

B: 计算(40 ms), I/O(20 ms), 计算(10 ms)

C: 计算(10 ms), I/O(30 ms), 计算(20 ms)

如果 3 道程序都使用相同的设备进行 I/O 操作(即程序以串行方式使用设备, 调度开销忽略不计), 试分别画出单道和多道运行的时间关系图。在两种情况下, CPU 的平均利用率各是多少?

8. 若主存中有 3 个程序 A、B、C, 其优先级从高到低依次为 A、B 和 C, 其单独运行时的 CPU 和 I/O 占用时间如表 1.2 所示。

表 1.2 程序单独运行时的 CPU 和 I/O 占用时间

程序	运行情况/ms						
	60 I/O ₂	20 CPU	30 I/O ₁	10 CPU	40 I/O ₁	20 CPU	20 I/O ₁
A							
B	30 I/O ₁	40 CPU	70 I/O ₂	30 CPU	30 I/O ₂		
C	40 CPU	60 I/O ₁	30 CPU	70 I/O ₂			

若 3 道程序并发执行, 调度开销忽略不计, 但优先级高的程序可以中断优先级低的程序, 优先级与 I/O 设备无关。试画出多道运行的时间关系图, 并指出最早与最迟结束的程序是哪个? 每道程序执行至结束分别花费多少时间? 计算 3 个程序全部运行结束时的 CPU 利用率。

9. 在单机系统中, 有同时到达的两个程序 A、B, 若每个程序单独运行, 则使用 CPU、DEV₁(设备 1)、DEV₂(设备 2) 的顺序和时间如表 1.3 所示。

表 1.3 程序单独运行时使用 CPU、DEV₁ 和 DEV₂ 的顺序和时间

程序	运行情况/ms						
	CPU	DEV ₁	CPU	DEV ₂	CPU	DEV ₁	CPU
程序 A	25	39	20	20	20	30	20
程序 B	CPU	DEV ₁	CPU	DEV ₂	CPU	DEV ₁	CPU
	20	50	20	20	10	20	45

给定下列条件:

- (1) DEV₁ 和 DEV₂ 是不同的 I/O 设备, 它们能够同时工作。
- (2) 程序 B 的优先级高于程序 A。但是, 当程序 A 占用 CPU 时, 即使程序 B 需要使用 CPU, 也不能打断程序 A 的执行而应等待。
- (3) 当使用 CPU 之后, 控制转向 I/O 设备, 或者使用 I/O 设备之后控制转向 CPU, 由控制程序执行中断处理, 但这段处理时间可以忽略不计。

试解答下列问题:

- (1) 哪个程序先结束?
- (2) 程序全部执行结束需要多长时间?
- (3) 程序全部执行完毕时, CPU 的利用率是多少?
- (4) 程序 A 等待 CPU 的累计时间是多少?
- (5) 程序 B 等待 CPU 的累计时间是多少?

10. 有两个程序, 程序 A 按顺序使用: (CPU)10 s、(设备甲)5 s、(CPU)5 s、(设备乙)10 s、(CPU)10 s。程序 B 按顺序使用: (设备甲)10 s、(CPU)10 s、(设备乙)5 s、(CPU)5 s、(设备乙)10 s。在顺序环境下先执行程序 A, 再执行程序 B, 求总的 CPU 利用率是多少?

11. 在某计算机系统中, 时钟中断处理程序每次执行的时间是 2 ms(包括进程切换的开销)。若时钟中断频

率是 60 Hz, 试问 CPU 用于时钟中断处理的时间比率是多少?

12. 在下列例子中, 区分“时分复用共享”与“空分复用共享”, 并对其做简单的解释。

- (1) 住宅区的土地
- (2) 个人计算机
- (3) 教室的黑板
- (4) 公共汽车上的座椅
- (5) UNIX 系统中的单用户文件
- (6) 分时系统中的打印机
- (7) C/C++ 运行时的系统堆栈



第二章

处理器管理

处理器管理是操作系统的重要组成部分,负责管理、调度和分配计算机系统的重要资源——处理器,并控制程序的执行。由于处理器管理是操作系统最核心的部分,无论是应用程序,还是系统程序,最终都要在处理器上执行以实现其功能,因此,处理器管理的优劣将直接影响系统的性能。程序以进程的形式来占用处理器和系统资源,处理器管理中最重要的是处理器调度,即进程调度,也就是控制、协调进程对处理器的竞争。

进程可被调度在一个处理器上交替地执行,或在多个处理器上并行执行。不同类型的操作系统可能采取不同的调度策略,交替执行和并行执行都是并发的类型。为了提高并发粒度和降低并发开销,现代操作系统引进了线程的概念,此时进程仍然是资源分配和管理的单位,线程则成为处理器调度的基本单位。本章在简要介绍处理器的硬件运行环境之后,首先将着重介绍计算机系统的中断机制,然后详细论述进程和线程的基本概念及其实现,最后全面讨论各个层次的处理器调度方法。

2.1 中央处理器

2.1.1 处理器

1. 单处理器系统和多处理器系统

处理器的任务是按照程序计数器的指向从主存读取指令,对指令进行译码,取出操作数,然后执行指令。如果计算机系统只包含一个运算处理器,称为单处理器系统;如果计算机系统包含多个运算处理器,则称为多处理器系统。

早期的计算机是基于单处理器的顺序处理机器,程序员编写串行代码,让指令在处理器上串行执行。为了提高计算机的处理速度,采用流水线技术,把指令分解成简单的、可独立执行的操作步骤,并将多条指令或多个操作步骤按照流水线方式部分重叠地执行,以此加快指令的执行速度。后来,流水线技术发展到更高级阶段,形成发射体系结构,其类型有超标量结构、超流水线结

构和超长指令字结构,设计的基本思路是:在一个机器周期内可以发射多条指令,同时取指令、译码并转储到保持缓冲区中,多个执行部件同时执行,只要存在空闲的执行部件,就会从非空保持缓冲区中读取指令并执行之,以提高指令执行的并行性。

随着硬件技术的不断进步,并行处理技术得到了迅猛的发展,多个功能部件和多个处理器可以同时工作以提高计算机系统的性能和可靠性,发展较为成熟的是多指令流多数据流结构的并行计算机,它分成两类:共享存储(紧密耦合)多处理器系统和分布存储(松散耦合)多处理器系统,前者共享所有处理器且平等地访问同一个物理主存;后者每个处理器均拥有自己的主存,处理器之间通过网络相连,在需要时通过网络交换数据。

根据处理器分配策略,共享存储多处理器系统分为主从式系统(Main/Slave Multi-Processor, MSP)和对称式系统(Symmetric Multi-Processor, SMP)。主从式系统的基本思路是:在特别的处理器上运行操作系统内核,在其他处理器上运行应用程序和系统程序,内核负责调度和分配处理器,并向其他程序提供各种服务。这种方式实现起来简单易行,但如果主处理器发生崩溃将导致系统崩溃,且极有可能形成性能瓶颈。

在对称式多处理器系统中,操作系统内核可以运行在任意处理机上,一个进程的多个线程可在不同的处理器上同时运行;服务器进程可使用多线程同时处理来自多个用户的请求;操作系统内核也可被设计成多进程或多线程以实现并行执行,系统中的任何处理器都可以访问任何存储单元及设备。SMP 机具有对称性、单一地址空间、低通信延迟和一致性高速缓存等特点,且可靠性及性价比高,可扩充性好。共享存储器只需保存一个操作系统和数据库副本,这样既有利于动态负载平衡,又有利于保证数据的完整性和一致性。很多计算机硬件设备制造商(如 HP、IBM、Sun、SGI)都设计和生产 SMP 机,这种机型大量用于联机分析处理、数据库和数据仓库等应用。

在分布存储多处理器系统中,每个处理机都有独立的主存和通道,各个处理单元之间通过预定的线路或网络进行通信,构成互联的计算机系统。集群(cluster)系统是分布存储多处理器系统的例子,是迄今为止所开发出来的另一种最为成功的并行机,它也是一种分布式系统。集群操作系统是分布式操作系统之一,运行时构成统一的计算资源;在集群系统中,每台计算机都是一个完整结点,脱离集群后可以独立地工作,从而具备很好的容错性和可伸缩性。当今,大多数通用操作系统既支持单处理器系统,又支持 SMP 系统和集群系统。

2. 寄存器

计算机系统的处理器包括一组寄存器,用于存放数据、变量和运算的中间结果。寄存器的数量根据处理器型号的不同而异,它们构成计算机一级存储,虽然其容量比主存小得多,但访问速度却很快,这组寄存器所存储的信息与程序的执行密切相关,构成处理器现场。不同类型的处理器由不同种类和数目的寄存器组成,在此以 Intel x86 为例,介绍各种寄存器。

(1) 通用寄存器

通用寄存器共 4 个,分别是 EAX、EBX、ECX 和 EDX,通常保存 32 位数据。为了进行 16 位操作并与 16 位机保持兼容性,通用寄存器的低位被当成 16 位寄存器,即 AX、BX、CX 和 DX;为了支持 8 位操作,寄存器的低 16 位可进一步划分为 8 位一组的高位字节和低位字节这两部分,

作为 8 个 8 位寄存器, 分别命名为 AH、BH、CH、DH 和 AL、BL、CL、DL。因此, 4 个通用寄存器既支持 1 位、8 位、16 位和 32 位数据运算, 又支持 16 位和 32 位存储器寻址。

(2) 指针及变址寄存器

指针及变址寄存器共 4 个, 分别是 ESP、EBP、ESI 和 EDI。使用其低 16 位时, 得到 SP、BP、SI 和 DI 寄存器, 与 16 位机保持兼容性。ESP 是堆栈指针寄存器, 保存栈顶元素的地址, 由于栈是向下增长的, 压入数据时首先要将栈指针 ESP 减 4, 当然在弹出数据时 ESP 要加 4。为函数调用而分配的那部分栈空间称为栈帧, ESP 是很重要的寄存器, 指向主存中当前堆栈的顶端, 栈内保存着已经调用但尚未退出的每个函数的栈帧, 其中包含返回地址、传递参数、局部变量等。EBP 是栈帧指针寄存器, 指向栈中当前帧的位置。ESI 是源变址寄存器, EDI 是目的变址寄存器, 也可当成通用寄存器使用, 具有自动增量和减量的功能。

(3) 段选择符寄存器

段选择符寄存器是 6 个 16 位的段寄存器, 它们分别是 CS、DS、SS、ES、FS、GS。在实模式下, 段寄存器存放段基址; 在保护模式下, 段寄存器所存放的是某个段的选择符。因为 16 位的寄存器无法存放 32 位段基址, 只好将其存放在描述符表中, 由描述符给出 32 位段基址。因此, 在 Intel x86 中, 段寄存器称为选择符寄存器, 主存中的指令和数据的地址可以通过段寄存器和指针寄存器共同给出。

(4) 指令指针寄存器和标志寄存器

指令指针寄存器 EIP 中存放下一条待执行指令的偏移量(相对于目前正在运行的代码段寄存器 CS 而言)偏移量加上当前代码段的基址形成下一条指令的地址。EIP 中的低 16 位可以被单独访问, 叫做指令指针 IP 寄存器, 用于 16 位寻址。

标志寄存器 EFLAGS 存放相关的处理器控制标志。

(5) 控制寄存器

共有 4 个 32 位控制寄存器, 分别是 CR0、CR1、CR2 和 CR3, 主要用于操作系统的分页机制。CR0 中包含标志位, 第 0 位是保护允许位, 用于启动保护模式; CR0 的第 31 位是分页允许位, 表示硬件的分页部件是否被允许工作, 其他位的功能还包括监控协处理器、任务转换等。CR1 暂未使用。CR2 是页故障线性地址寄存器, 保存最近一次出现缺页时的 32 位线性地址。CR3 是页目录基址寄存器, 保存页目录的物理地址, 页目录总是置于以 4 KB 为单位的存储器边界上, 因此, 其地址的低 12 位总是 0。

(6) 外部设备使用的寄存器

① 数据寄存器或缓冲区

存放当前要在设备和主机之间传送的 I/O 数据, 程序可以访问数据寄存器或缓冲区。

② 状态寄存器

保存外部设备或接口的状态信息, 以便 CPU 在必要时测试状态, 了解设备的工作情况, 例如, 命令是否完成、设备是否出错等。

③ 控制寄存器

CPU 传送给外部设备或接口的控制命令通过控制寄存器发送,例如,CPU 要启动磁盘工作,必须发送启动命令等。

每个设备控制器都有上述 3 种类型的寄存器用于和 CPU 之间的通信,每个接口所配备的寄存器数量是根据设备的具体需要而定的,例如,键盘只有一个 8 位数据寄存器,并把状态寄存器和控制寄存器合二为一;复杂的磁盘则需要多个数据寄存器、状态寄存器和控制寄存器。

CPU 和控制寄存器、数据寄存器及缓冲区之间的通信方式有以下三种。

① 为每个控制寄存器分配一个 I/O 端口号(8 位或 16 位),通过使用核心态 I/O 指令(IN/OUT),CPU 可以读写端口。在这种方式下,主存空间和 I/O 端口空间是各自独立的,早期的大多数计算机采用此方式工作。

② 把所有控制寄存器映射到主存空间,为每个寄存器分配唯一的主存地址,且与用户的可用主存地址不重叠。为控制寄存器所分配的地址位于地址空间的顶端,可以像主存单元一样被读写,不再需要专门的 I/O 指令。此方法由 PDP11 首先引入。

③ 混合方式。

既在主存空间开辟数据缓冲区,而控制寄存器又有其单独的 I/O 端口。Intel x86 采用这种结构:I/O 地址为 16 位,每个地址对应于一个字节宽的 I/O 端口,共表示 64 KB 的 I/O 地址空间;640 KB~1 MB 的主存地址保留,用做设备数据缓冲区。

上述方式的基本工作过程是:

- ① CPU 读取数据时,把所需数据地址(主存地址或 I/O 端口号)放在地址总线上;
- ② 在控制总线上插入读信号,同时另一条信号线表明数据来自 I/O 空间还是主存空间;
- ③ 由相应的对象(设备或主存)对请求做出响应。

3. 特权指令与非特权指令

计算机的基本功能是执行程序,最终被执行的是存储在主存中的机器指令代码,处理器根据程序计数器的指向从主存中取指令到指令寄存器,然后译码并执行,程序计数器将自动地增长或变为转移地址以指明下一条待执行指令的地址。

每台计算机的机器指令集合称为指令系统,反映计算机的功能和处理能力。一般来说,处理器的机器指令集是专用的,指令往往分为 5 类。

(1) 数据处理类

执行算术和逻辑运算。

(2) 转移类

如无条件转移、条件转移、计数转移等用于改变指令的执行序列。

(3) 数据传送类

在处理器的寄存器之间、寄存器和主存单元之间、主存单元之间交换数据。

(4) 移位与字符串类

移位分为算术、逻辑和循环移位;字符串处理包括字符串的传送、比较、查询和转换。

(5) I/O 类

启动和控制设备,使得主存和设备之间交换数据。

配置操作系统之后,操作系统内核程序拥有较高级别的特权,可以使用全部的机器指令。但是,应用程序的权限较低,只能使用指令系统的子集。这是因为应用程序在执行有关资源管理的机器指令时易于导致系统混乱,造成系统或用户信息被破坏,例如,置程序状态字指令将导致处理器占有程序的变更;如果错误地启动设备 I/O 指令,会有多个程序竞争使用设备而导致 I/O 混乱,这些指令只有操作系统才有权使用。因此,在多道程序设计环境中,从资源管理和控制程序执行的角度出发,必须把指令系统中的指令分成两类:特权指令 (privileged instruction) 和非特权指令。所谓特权指令是指仅供内核程序使用的指令,如启动设备、设置时钟、控制中断屏蔽位、清空主存、建立存储键、加载 PSW 等敏感性操作。内核能够执行全部指令,应用程序只能使用非特权指令。如果应用程序执行特权指令,会导致非法执行而产生保护中断,继而转向操作系统的“用户非法执行特权指令”的异常处理程序进行处理。

4. 处理器状态

(1) 核心态和用户态

处理器如何得知当前是操作系统还是应用程序在其上运行呢?这将取决于处理器状态标志,在执行不同的程序时,根据执行程序对资源和机器指令的使用权限将处理器设置成不同的状态,处理器状态又称处理器模式,可划分为核心态 (kernel mode, 又称管态 (supervisor mode)) 和用户态 (user mode, 又称目态)。

当处理器处于核心态时,CPU 运行可信软件,硬件允许执行全部机器指令,可以访问所有主存单元和系统资源,并具有改变处理器状态的能力;当处理器处于用户态时,CPU 运行非可信软件,程序无法执行特权指令,且访问仅限于当前 CPU 上进程的地址空间,这样就能防止内核受到应用程序的侵害。处理器模式位扩展了操作系统的保护权限,意味着在核心态执行的程序比用户态的拥有更多权限去访问主存和执行特权指令,所以模式位被用来区分可信软件和非可信软件。

Intel x86 处理器的状态有 4 种,分别支持 4 个特权级别,0 级权限最高,3 级权限最低。典型的应用是特权级别依次设定为:0 级为内核级,处理 I/O 操作、存储管理并执行其他关键操作;1 级为系统调用级,可以执行系统调用,获得特定的和受保护的程序的服务;2 级为共享库级,可被多个运行进程共享,允许调用库函数,读取但不修改相关的数据;3 级为应用程序级,所受到的保护最少。当然,各个操作系统在实现过程中可以根据具体策略有选择地使用硬件所提供的保护级别,如运行在 Intel x86 上的 Windows 操作系统只使用 0 级和 3 级特权。

(2) 处理器模式转换

有两类情况会导致处理器从用户态向核心态转换。一是程序请求操作系统服务,执行系统调用;二是在程序运行时,产生中断或异常事件,运行程序被中断,转向中断处理程序或异常处理程序工作。这两类情况都通过中断机制发生,可以说中断和异常是用户态到核心态转换的仅有途径。当系统中产生中断或异常(如设备中断或非法操作码异常),处理器将做出响应并交换程序状态字,此时会导致处理器从用户态转向核心态,中断事件或异常处理程序的程序状态字中的处理器模式位一定是“核心态”。从用户模式转向内核模式,不存在类似指令或其他方法,否则任

任何进程都可以轻易转入特权状态,从而使保护机制失效。那么,如何实现核心态到用户态的转换呢?计算机通常提供一条称作加载程序状态字的特权指令(IBM 370 为 LOAD PSW 指令,Intel x86 为 IRET 指令),用来实现从系统(核心态)返回用户态,将控制权转交给应用进程。

2.1.2 程序状态字

计算机如何知道当前处理器处于何种状态?此时能否执行特权指令?操作系统通常引入程序状态字(Program Status Word,PSW)来区分不同的处理器状态。如图 2.1 所示是 IBM 370 系统计算机程序状态字的基本格式。

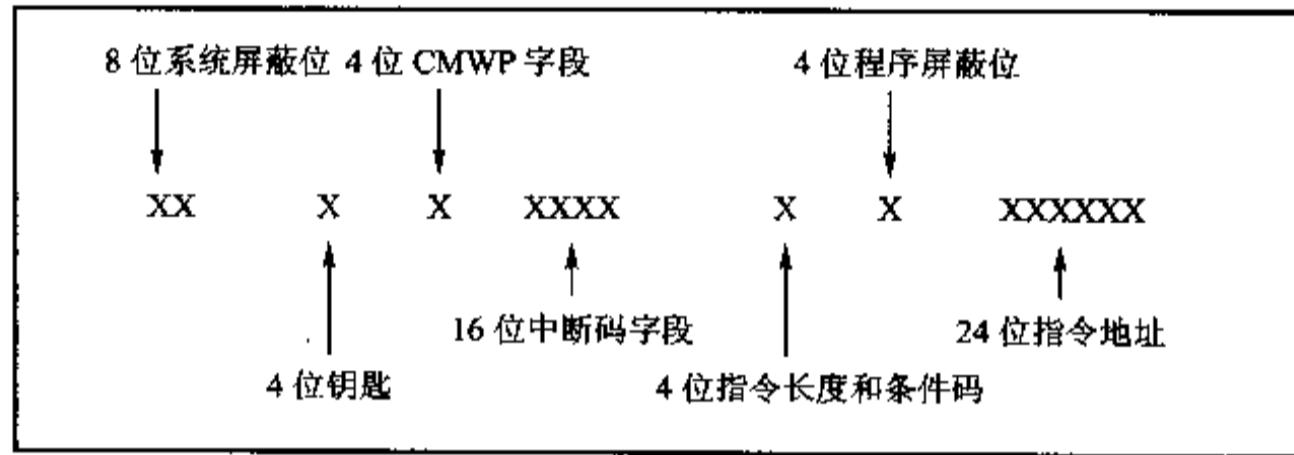


图 2.1 IBM 370 系统程序状态字基本格式

(1) 系统屏蔽位(0~7 位):表示允许或禁止某个中断事件发生,相应位为 0 或 1 时分别表示屏蔽或响应中断。0~7 位依次是 0~6 号通道和外中断屏蔽位。

(2) 钥匙(8~11 位):当未设置存储器保护时,这 4 位是 0;当设置存储器保护时,PSW 中的这 4 位钥匙与欲访问的主存区域的存储锁必须匹配,否则将不能访问。

(3) CMWP(12~15 位):依次是 PSW 基本/扩充控制方式位、开/关中断位、运行/等待位、用户态/核心态位。

(4) 中断码字段(16~31 位):中断码字段与中断事件相对应,记录当前的中断源。

(5) 指令长度(32~33 位):01/10/11 分别表示半字长指令、整字长指令和一字半长指令。

(6) 条件码(34~35 位):表示指令执行的结果状态。

(7) 程序屏蔽位(36~39 位):表示允许(为 1)或禁止(为 0)程序性中断,自左向右各位所对应的程序性事件是:定点溢出、十进制数溢出、阶下溢,第 39 位备用。

(8) 程序计数器(40~63 位):指明下一条执行指令的绝对地址。

不难看出,PSW 用来指示处理器状态,控制指令的执行顺序,并且保留和指示与运行程序有关的各种信息,主要作用是实现程序状态的保护和恢复。每个正在执行的程序都有一个与其当前状态相关的 PSW,而每个处理器都设置一个硬件 PSW 寄存器,一个程序占用处理器执行时,其 PSW 将占用硬件 PSW 寄存器。

不同处理器的控制寄存器的组织方式不同,多数计算机的处理器现场可能找不到一个具体的 PSW 寄存器,但总会有一组控制寄存器和状态寄存器实际上起到这一作用。为了方便操作

系统原理的讨论,本书使用 PSW 和 PSW 寄存器这两个术语。

在 Intel x86 中,PSW 由标志寄存器 EFLAGS 和指令指针寄存器 EIP 组成,均为 32 位。EFLAGS 的低 16 位称为 FLAGS,可以当做一个单元来处理。标志划分为三组:状态标志、控制标志、系统标志。

(1) 状态标志

使得一条指令的执行结果影响其后指令的执行。算术运算指令使用 OF(溢出标志)、SF(符号标志)、ZF(结果为 0 标志)、AF(辅助进位标志)、CF(进位标志)、PF(奇偶校验标志)、AC(地址对齐检查);串扫描、串比较、循环指令使用 ZF 通知其操作结束。

(2) 控制标志

控制标志包括 DF、VM、TF 和 IF 位。DF(方向标志)控制串指令操作,DF 为 1 时,使得串指令自动减量,即从高地址向低地址处理串操作;DF 为 0 时,串指令自动增量。VM(虚拟 8086 方式标志)为 1 时,从保护模式进入虚拟 8086 模式。TF(步进标志)为 1 时,使处理器执行单步操作。IF(中断允许标志)为 1 时,允许响应中断,否则关中断。

(3) 系统标志

系统标志与进程管理有关,共 3 个:IOPL(I/O 特权级别标志)、NT(嵌套任务标志)和 RF(恢复标志),用于保护模式。指令指针寄存器 EIP 的低 16 位称为 IP(保护模式使用 32 位),存放顺序执行的下一条指令相对于当前代码段起始地址的一个偏移量,IP 可以当做一个单元使用。

■ ■ ■ 中 断 技 术

2.2.1 中断概念

中断机制是现代计算机系统的重要组成部分之一,每当应用程序执行系统调用要求获得操作系统服务、I/O 通道及设备报告传输情况或者产生形形色色的内部和外部事件时,都要通过中断机制产生中断信号并启动内核工作,可以说操作系统是由“中断驱动”的。最初,中断技术仅作为设备向 CPU 报告 I/O 操作情况的一种手段,以免 CPU 因不断轮询设备而耗费时间,中断的出现帮助解决主机和设备的并行性问题。现在中断技术的应用范围越来越广,在计算机运行过程中,许多事件有可能随机发生,如硬件故障、网络通信、人机交互和程序出错等,必须对这些事件及时加以处理,所以,为了请求系统服务,实现并行工作,处理突发事件,满足实时要求,需要打断处理器的正常工作流,为此提出中断的概念。中断(interrupt)是指在程序执行过程中,遇到急需处理的事件时,暂时中止现行程序在 CPU 上的运行,转而执行相应的事件处理程序,待处理完成后返回断点或调度其他程序执行。例如,从磁带上读入一组信息,当发现读入操作有错时,将产生读数据错中断,操作系统暂停当前的工作,并组织磁带退回重读这一组信息,这样就有可能克服错误。

在不同的计算机系统中,通常有不同的中断源和中断装置,它们有一个共性,即在中断事件

发生后，中断装置能改变处理器内操作的执行顺序。可见中断是现代操作系统实现并行性的基础之一。引入中断机制，操作系统在让应用程序放弃控制权或从应用程序获得控制权时将具有更大的灵活性。

2.2.2 中断源分类

1. 按中断事件的性质和激活方式划分

按中断事件的性质和激活方式可将中断分成两类：强迫性中断和自愿性中断。强迫性中断事件肯定不是正在运行的程序所期待的，是由随机事件或外部请求信息引起的。强迫性中断事件有以下几种。

(1) 机器故障中断

机器执行指令过程中硬件可能出现种种事件，例如，电源故障、通路校验错误、主存出错等。

(2) 程序性中断

程序执行过程中可能发生种种例外情况，例如，非法操作码、定点溢出、除数为 0、地址越界等。

(3) 外部中断

由计算机系统外部发送中断信号，反映外界对本机的种种要求，例如，时钟中断、控制台中断、它机中断等。

(4) 输入输出中断

来自通道、控制器、设备的中断能够反映 I/O 操作情况，例如，设备出错、传输结束、启动失败等。

自愿性中断事件是正在运行的程序所期待的，是由于执行“访管指令”而引起的，它表示运行程序对操作系统有某种需求，一旦机器执行访管指令，就会使 CPU 状态从用户态转向核心态，停止现行程序的执行而转入内核的相应系统调用例程进行处理。例如，要求操作系统协助启动外部设备工作。

这两类中断事件的响应过程略有不同，如图 2.2 所示。

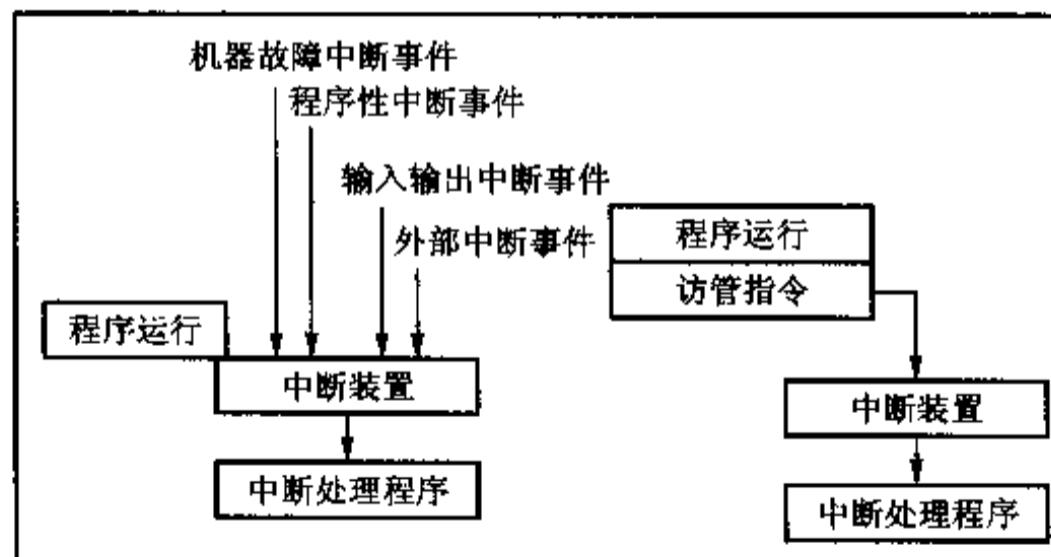


图 2.2 按中断事件的性质和激活方式分类

2. 按中断事件的来源和实现手段划分

按中断事件的来源和实现手段可将中断划分为硬中断和软中断两类,如图 2.3 所示。

(1) 硬中断:外中断和内中断

硬中断可划分为外中断和内中断两种。

① 外中断,又称中断或异步中断,是指来自处理器之外的中断信号,包括时钟中断、键盘中断、它机中断和设备中断等。外中断又分为可屏蔽中断和不可屏蔽中断,各个中断具有不同的中断优先级,表示事件的紧急程度,在处理高一级中断时,往往会部分或全部屏蔽低级中断。

② 内中断,又称异常(exception)或同步中断,是指来自处理器内部的中断信号,通常是由于在程序执行过程中,发现与当前指令关联的、不正常的或错误的事件。内中断可被细分为三种:访管中断,由执行系统调用而引起;硬件故障中断,如电源失效、协处理器错误、奇偶校验错误、总线超时等;程序性异常,如非法操作、地址越界、页面故障、调试指令、除数为 0 和浮点溢出等。所有这些事件均由异常处理程序来处理,响应处理器中状态的变化且通常依赖于执行程序的当前现场。内中断不能被屏蔽,一旦出现应立即予以响应并进行处理。

中断和异常之间的区别如下。

① 中断是由与当前程序无关的中断信号触发的,系统不能确定中断事件的发生时间,所以,中断与 CPU 是异步的,CPU 对中断的响应完全是被动的。中断的发生与 CPU 模式无关,既可发生在用户态,又可发生在核心态,通常在两条机器指令之间才能响应中断。一般来说,中断处理程序所提供的服务不是当前进程所需要的,如时钟中断、硬盘中断等,中断处理程序在系统的中断上下文中执行。

② 异常是由 CPU 控制单元产生的,源于现行程序执行指令过程中检测到例外。异常与 CPU 是同步的,允许指令在执行期间响应异常,而且允许多次响应异常,大部分异常发生在用户态。异常处理程序所提供的服务通常是当前进程所需要的,如处理程序出错或页面故障,异常处理程序在当前进程的上下文中执行。以 Linux 为例,异常按照错误报告方式分为 4 种:故障(fault)、陷阱(trap)、终止(abort)和编程异常(programmed exception)。此外,中断的嵌套发生是允许的,但异常多为一重,异常处理过程中可能会产生中断,但反之则不会发生。下面是 Intel x86 体系结构中发布的且在 Linux 中使用的部分异常:除法溢出错(0 号,故障)、调试异常(1 号,故障或陷阱)、断点中断(3 号,陷阱)、算术溢出(4 号,陷阱)、边界异常(5 号,故障)、无效操作(6 号,故障)、设备不可用(7 号,故障)、双异常(8 号,故障)、协处理器段溢出(9 号,故障)、任务状态段异常(10 号,故障)、段不存在(11 号,故障)、堆栈溢出(12 号,故障)、一般性保护(13 号,故障)、页故障(14 号,故障)、浮点数错(16 号,故障)和边界异常(17 号,故障)等。

编程异常用于实现系统调用,进程自愿进入核心态以请求系统服务,所以属于主动行为;故障是指程序运行中系统捕获的潜在可恢复的错误,经处理后可返回当前指令再次执行,页面故障是最典型的例子;终止是指致命的不可恢复的错误,如主存芯片发生奇偶校验错误,通常不会返

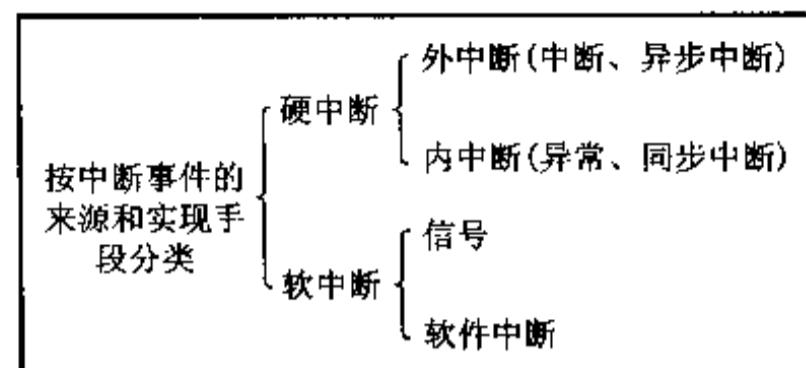


图 2.3 按中断事件的来源和实现手段分类

回原来的程序而转向内核 abort 例程,有时甚至需要重新启动计算机系统;陷阱是在执行特定的调试指令时触发的,被调试的进程遇到所设置的断点时会暂停等待。

发生故障时所保存的返回指令地址指向触发异常的那条指令,故障处理完毕后会重新执行这条指令。编程异常和陷阱是由于执行访管指令而引起的同步操作,异常将返回至触发异常的下一条指令。

IBM 中大型机上的中断源使用上述第一种分类方法,Intel x86 机上的中断源则采用上述第二种分类方法。

(2) 软中断:信号和软件中断

外中断与内中断(中断和异常)要通过硬件设施来产生中断请求,它们都是硬中断。与其相对应,不必由硬件产生中断源而引发的中断称为软中断。软中断利用硬中断的概念,采用软件方法对中断机制进行模拟,实现宏观上的异步执行。

软中断可分为两种。“信号”是一种软中断机制,信号的发送者相当于中断源,而信号的接收者必然是一个进程(相当于 CPU)。“软件中断”是另一种软中断机制,其第一个典型应用例子是 Linux 中的 bottom half,它的升级就是 Linux 中复杂、庞大的软中断子系统 softirq 机制;第二个典型应用例子是 Windows 中由内核发出的 Dispatch/DPC 和 APC 等中断,用于启动线程调度,延迟过程调用和异步过程调用的执行。

(3) 硬中断与软中断的类比

上述几种中断的通常用法如下。

①“中断”(硬中断)用于外部设备对 CPU 的中断(中断正在运行的任何程序),转向中断处理程序执行。

②“异常”(硬中断)因指令执行不正常而中断 CPU(中断正在执行这条指令的程序),转向异常处理程序执行。

③“软件中断”(软中断)用于硬中断服务程序对内核的中断,在上半部分中发出软件中断(即标记下半部分),使得中断下半部分在适当时刻获得处理。

④“信号”(软中断)用于内核或进程对某个进程的中断,向进程通知某个特定事件发生或迫使进程执行信号处理程序。

中断机制与信号机制可以进行类比:

①两者在概念上是一致的,进程接收到信号与 CPU 接收到中断是相似的。进程可以向其他进程发送信号,而 CPU 也可以向其他 CPU 发出中断请求。

②两者都是“异步”的,CPU 和进程在工作过程中,并不知道中断或信号何时产生,更不需要停顿下来等待中断或信号的发生。硬中断在一条指令执行结束时安排查询,信号则在异常处理末尾或时钟中断处理结束之前进行查询。

③两者在实现上均采用“向量表”,中断机制设置“中断向量表”,以向量号为索引查找中断向量,然后转入中断处理程序或异常处理程序。信号机制采用“信号向量表”,通过信号编号查表,从而找到并转入相应的信号处理程序。

④ 两者均有“屏蔽”设施。中断机制可直接对某些中断源加以屏蔽；信号机制中也存在相似的手段，可以设置信号字段位屏蔽，使进程对于发来的信号不予理睬。

中断机制与信号机制之间的区别是：前者由硬件和软件相结合来实现，而后者则完全由软件实现；中断向量表和中断处理程序（全部由系统提供）均位于系统空间，而信号向量表虽然处于系统空间，但信号处理程序往往由应用程序提供，并在用户空间执行。

也可以把硬中断和软中断 BH 进行类比：

① 数组 bh_base[] 相当于硬件中断机制中的数组 irq_desc[], 不过 irq_desc[] 中的每个元素代表一个中断服务程序队列，而 bh_base[] 的每个元素只代表一个 BH 函数；

② bh_active 在概念上相当于硬件的“中断请求寄存器”，而 bh_mask 则相当于硬件中的“中断屏蔽寄存器”；

③ 需要执行一个 BH 函数时，就通过 mark_bh() 将 bh_active 中的某一位设置成 1，相当于中断源发出（软件）中断请求，而所设置的具体标志位则类似于“中断向量”；

④ 如果相当于“中断屏蔽寄存器”的 bh_mask 中的相应位也是 1，即系统允许执行这个 BH 函数，那么就会在每次执行完 do_IRQ() 中的中断服务程序后，及每次系统调用结束后，在函数 do_bottom_half() 中执行相应的 BH 函数，而 do_bottom_half() 则类似于 do_IRQ()。

除了操作系统中的一部分低层硬件异常由内核异常处理程序来处理之外，大部分异常均转化为信号（软中断）再处理。由于异常与当前运行进程紧密相关，每当执行指令产生异常事件时，可通过信号处理程序向当前运行进程发送一个信号。

关于硬中断或软中断处理的延迟问题，一般来说，CPU 在接收和响应硬中断之后会立即调用中断处理程序或异常处理程序；对于所接收的信号或软件中断，由于此时进程未必占有处理器运行或内核正在执行敏感性操作，通常会有一定的时间延迟，内核或相关进程在适当的时刻才能加以处理。信号和软件中断虽然都由软件产生，并由软件处理，但是它们的中断来源、适用场合、实现手段并不相同。

2.2.3 中断和异常的响应及服务

无论是外部产生的中断，还是内部出现的异常，或程序执行系统调用自愿访管；无论中断源是被动的，还是主动的，CPU 的响应过程基本上是一致的，这就是：在执行完当前指令后，或在执行当前指令的中途，根据中断源所提供的“中断向量”，在主存中找到相应服务程序的入口地址并调用此服务程序。中断向量由硬件或操作系统内核预先分配和设置，系统调用所对应的向量则在访管指令中给出，各种异常的向量在 CPU 的硬件结构中预先规定，这样，不同情况就因中断向量的不同而区分开来。因此，中断和异常可以作为统一模式加以实现，国内许多教材一律将其归入“中断机制”就是这个道理。

那么，处理器如何响应中断和异常，操作系统如何转到中断处理程序和异常处理程序执行呢？先看中断，它主要是由外部设备、时钟部件或其他机器发出的，发现中断源并产生中断的硬件称为中断装置，这些硬件包括中断逻辑线路和中断寄存器，当前指令执行结束后，CPU 会检查

中断寄存器是否有中断事件发生,若无中断信号或中断信号被屏蔽,继续执行程序的后续指令,否则将暂停执行当前程序,转向内核的中断处理程序。再看异常,它是在执行指令时,由于指令本身的原因发生的,指令的实现和执行逻辑一旦发现异常情况,便转向内核的异常处理程序,这个由硬件对中断和异常事件做出反应的过程称为中断响应。

迄今为止,所有计算机系统都采用硬件和软件,即硬件中断装置和软件中断处理程序/异常处理程序相结合的方法实现中断/异常处理。一般来说,中断/异常的响应需要顺序做 4 件事。

(1) 发现中断源

在中断未被屏蔽的前提下,硬件发现中断/异常事件,并由 CPU 响应中断/异常请求。当发现多个中断源时,将根据预定的中断优先级先后响应中断请求。

(2) 保护现场

暂停当前程序的运行,硬件将中断点的现场信息(PSW)保存至核心栈,使得中断处理程序/异常处理程序在运行时,不会破坏被中断程序中的有用信息,以便在中断处理结束后返回原程序继续运行。

(3) 转向中断/异常事件的处理程序

这时处理器状态已从用户态切换至核心态,中断处理程序/异常处理程序开始工作。

(4) 恢复现场

当中断处理结束后,恢复 PSW,重新返回中断点以便执行后续指令。当异常处理结束后,返回点会因异常类型而异。大部分应用程序指令执行出错时,异常处理会结束进程,不可能回到原程序;如果是执行访管指令,则异常处理完成后返回这条访管指令的下一条指令;对于页面故障,异常处理结束后会返回发生异常的那条指令重新执行。

如图 2.4 所示是 IBM 中大型机的中断响应过程。如果把被中断程序的 PSW 称为旧 PSW,而把中断处理程序的 PSW 称为新 PSW,如何实现新旧 PSW 之间的交换呢?系统通常为每种中断都开辟主存的固定单元以存放新的和旧的 PSW,主存中开辟专用的双字单元(用十六进制数标出),用于存放各类中断的旧的和新的 PSW(分别为旧的和新的外中断、访管中断、程序中断、机器故障中断和 I/O 中断),CPU 中还有硬件 PSW 寄存器保存运行程序的现行 PSW。当响应中断时,由硬件执行①把中断码装配至现行 PSW 中,然后执行②把现行 PSW 保存到中断类型对应的旧 PSW 单元,再执行③把相应的新 PSW 加载至现行 PSW,这样就会引出相应的中断处理程序。中断事件处理结束后,如果执行④便可从断点返回,继续执行被中断的程序。

Linux 中断机制在保护模式下的实现采用中断描述符表(Interrupt Descriptor Table, IDT),此表包含 256 个中断描述符,与中断或异常一一对应。描述符的作用是把程序控制权转给中断/异常服务程序,每个描述符均占用 8B,通过它就能找到服务程序的起始地址、属性以及程序特权级别等信息。IDT 的位置由硬件中断描述符表寄存器 IDTR 指定,它是一个 48 位的寄存器,高 32 位是 IDT 的基址,低 16 位限定 IDT 的长度。

每个中断/异常都有一个向量号,其取值范围在 0~255 之间,是中断在 IDT 中的索引。每个中断/异常均有其相应的处理程序,在使用中断之前,必须在 IDT 中注册以保证发生中断时能

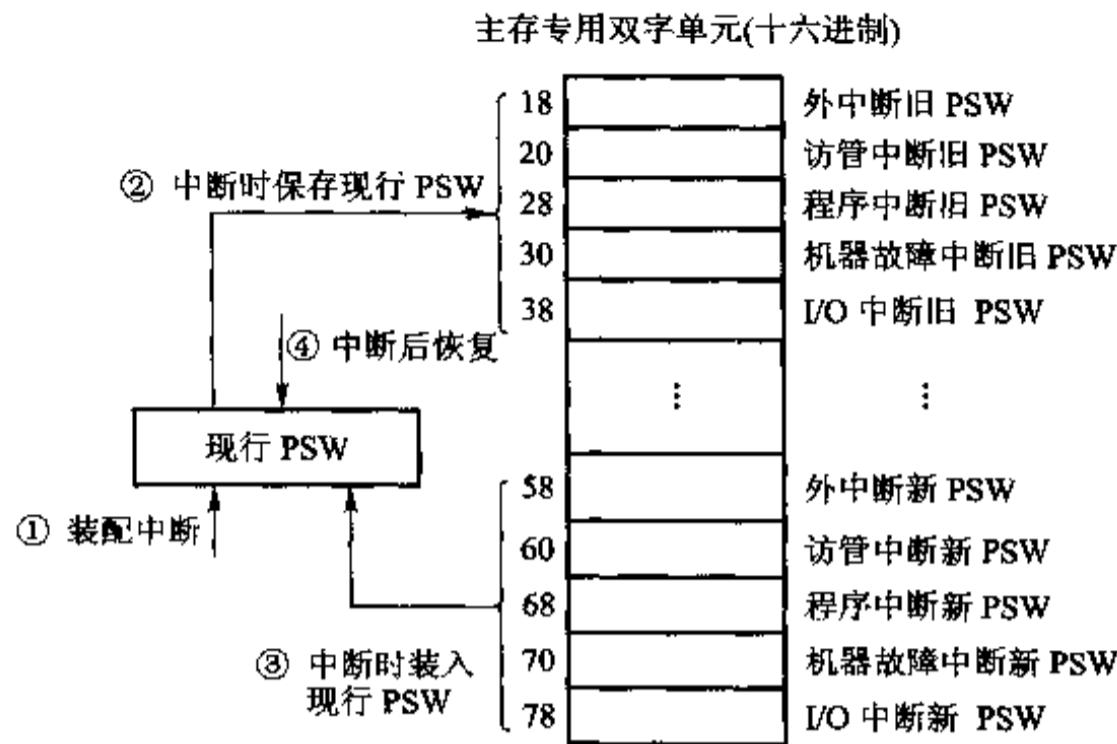


图 2.4 IBM 中大型机的中断响应过程

找到相应的处理程序。在系统初始化时创建 IDT，向量号的使用情况为：0~31 号对应异常或硬件非屏蔽中断，32~47 号分配给硬件可屏蔽中断，48~255 号分配给软件中断，其中，128 号（0x80）中断向量用来实现系统调用。

2.2.4 中断事件处理

1. 中断和异常的一般处理过程

中断事件的处理比异常事件的处理复杂得多，这是因为：对于当前进程而言，中断是异步事件，中断处理程序工作在核心态的中断上下文中，不允许被阻塞；中断处理程序应能够为共享同一根中断请求线的多台设备服务；中断处理程序的执行时间要尽可能的短，以缩短关中断时间，为此，中断处理被分成上半部分和下半部分，这也增加了中断处理的复杂性。内核处理所有中断的过程是相同的，其执行流程大致如下。

(1) 当设备发出中断请求时，中断信号由设备发送到中断控制器，中断控制器将 IRQ 号转换成中断向量号并传送给 CPU；CPU 响应中断之后，自动保护现场，把 PSW(EFLAGS、CS 和 EIP)、中断向量号、用户栈段寄存器 SS 和栈指针 ESP 压入核心栈，并根据中断向量号查找 IDT，生成指向中断描述符表的指针，找到中断服务程序 IRQn_interrupt 的地址，开始执行中断处理程序；

(2) 进入中断公共代码段 common_interrupt 执行 SAVE_ALL，将硬件未保护的所有寄存器内容保护到核心栈；

(3) 调用 do_IRQ() 函数，对中断控制器进行确认，设置中断源状态等；

(4) 根据 IRQ 为发出中断请求的设备提供服务，调用服务程序执行相关的中断处理任务，标记中断下半部分；如果有多个设备共享此中断 IRQ，需要执行此中断线上所有设备的中断服

务程序；

(5) 检查 softirq_active 和 softirq_mask 是否有标记的下半部分,若有则调用 do_softirq() 执行之;

(6) 跳转至 ret_from_intr 退出,恢复发生中断之前的现场。

异常处理可分为三步:一是当前进程执行指令产生异常;二是进入异常处理程序并执行之;三是从异常处理程序返回。产生异常之后,将触发 0~31 号中断向量所对应的异常,并自动转向异常处理程序的公共入口以执行下列操作。

(1) 将硬件错误代码和异常向量号存入当前进程的 PCB(Process Control Block, 进程控制块)中;

(2) 判别异常是产生于核心态还是用户态,对于前者,将直接转向内核预定义服务程序进行处理,未被处理的核心态异常是操作系统的致命错误;对于后者,终止当前进程的运行,调用 force_sig() 函数向当前进程发信号;

(3) 从 ret_from_exception 返回用户空间时,将检查进程是否有信号等待处理,若有则根据信号类型调用相应的函数进行处理。

当然,也有些异常处理是很特殊的,典型的例子是页面故障异常,它是实现分页式虚存管理的硬件支撑,由系统来处理这种异常,异常处理程序最终转向 do_page_fault() 执行页面调度。

中断和异常的处理过程大致相同,但在产生异常时,硬件并不清除中断标志位,此时还允许外部硬件中断;而在产生中断时,硬件将立即清除中断标志位,以禁止其他硬件中断。

2. 硬件故障中断

一般来说,这种中断事件是由硬件故障导致的,如电源故障、主存故障、设备故障,排除这种故障需要人工干预,如复位、设置或替换。中断处理程序所能做的事情一般是保护现场,停止设备工作,停止处理器运行,将故障信息向操作员报告,并对故障所造成的破坏进行估计和恢复。

3. 程序性中断

应用程序的错误有以下几类:一是语法错误,这可由编译程序发现并报错;二是逻辑错误,可由测试程序发现并报错;三是程序运行过程中所产生的异常,例如,定点溢出、阶码下溢、除数为 0 等。不同的用户往往有不同的处理要求,借助于信号机制,操作系统可将所捕获的这类中断事件原封不动地转交给应用程序自行处理。

4. I/O 中断

I/O 中断的处理原则如下。

(1) I/O 操作正常结束

把等待传输的进程均设置为就绪态,查看是否有其他进程等待设备或通道,若有则释放之。

(2) I/O 操作发生故障

先向设备发命令索取状态字,然后分析产生故障的确切原因,再采用复执方式或请求人工

干预。

(3) I/O 操作发生异常

分析情况,采取相应的措施,向操作员报告,如通知操作员换卷、装纸等。

(4) 设备报到或设备结束

表示有设备接入可供使用或设备断开暂停使用,操作系统应修改系统数据中相应设备的状态。

5. 访管中断

这类中断是由程序执行访管指令而引起的,表示当前运行程序对操作系统功能的调用,可看做机器指令的一种扩充。访管指令包括操作码和访管参数两部分,前者表示此指令是访管指令,后者则表示具体的访管要求。IBM 出品的机器在执行访管指令时,将访管参数作为中断字并入 PSW,同时送入主存中的指定单元,操作系统的访管中断处理程序分析访管参数、进行合法性检查后,按照访管参数的具体要求进行相应的处理。在 Intel x86 机器上,当 CPU 接收一个中断请求时,由设备接口发出唯一的“中断向量”,Linux 将其作为系统调用号装入 EAX 寄存器,其他参数装入 EBX、ECX、EDX、ESI 和 EDI 寄存器中。访管参数超过 6 个的情况并不多见,寄存器存放参数在用户空间的地址,以此为指针来传递参数,然后执行访管指令 int 0x80 陷入操作系统。返回值则放入 EAX 寄存器中。

不同的访管参数对应不同的服务要求,就像机器指令的不同操作码对应不同的动作要求一样,系统调用机制在本质上通过特殊的硬指令和中断机制来实现。不同机器其系统调用命令的格式和功能号的含义不尽相同,但是任何机器的系统调用都有如下共性处理流程。

- (1) 程序执行访管指令,并通过适当的方式指明系统调用号;
- (2) 通过中断机制进入访管中断处理程序,现场保护到核心栈,按功能号实现跳转;
- (3) 通过系统调用人口表找到相应功能服务例程的入口地址;
- (4) 执行相应的服务例程,正常情况下在结束后返回系统调用的下一条指令继续执行。

6. 时钟中断

时钟是操作系统进行调度工作的重要工具,如维护系统的绝对日期和时间、让分时进程按时间片轮转、让实时进程定时发送或接收控制信号、系统定时唤醒或阻塞进程、对用户进程记账、测量系统性能等,利用定时器能够确保操作系统在必要时获得控制权,陷入死循环的进程最终会因时间片耗尽而被迫让出处理器。

时钟可分为绝对时钟和间隔时钟。通常使用一个硬件时钟,它按照固定周期发出中断请求,Linux 早期版本每 10 ms(频率 100 Hz)时钟中断 1 次,从 Linux 2.5 版开始,时钟中断频率提高至 1 000 Hz,即每 1 ms 产生 1 次时钟中断。系统设置一个绝对时钟寄存器,定时地把此寄存器的内容加 1。如果这个寄存器的初始内容为 0,那么,只要操作员告诉系统开机时的年、月、日、时、分、秒,就可以据此推算当前的年、月、日、时、分、秒。

间隔时钟在每个时间切换点将间隔时钟寄存器的内容减 1,通过程序设置此寄存器的初值,当间隔时钟寄存器的内容为 0 时,就产生间隔时钟中断,等同于闹钟的作用,意味着预定时间已

到。操作系统经常利用间隔时钟执行调度控制。

时钟硬件的作用是按已知时间间隔产生中断,与时间有关的其他各项任务必须由软件完成,要建立操作系统和应用程序所使用的高层时钟和定时器服务。

(1) 绝对时钟服务

绝对时钟服务提供以下函数: update_clock() 更新当前时间,每个时钟滴答时刻都会调用此函数,标识从某个已知起始时刻起的时钟滴答的数目; get_time() 返回当前时钟值,既可以是整数值,也可以经过换算返回“年、月、日、时、分、秒”; set_clock() 把当前时间设置为新的时钟值。

(2) 间隔定时器

进程可以被延迟或阻塞,直至被间隔定时器的中断信号唤醒。间隔定时器提供以下函数。

① delay(tdel) 把调用进程阻塞由参数 tdel 所指定的时间长度,进程保持阻塞直至本地时间到达发生进程阻塞时的当前时间 + tdel 的时刻。

② set_timer(tdel) 硬件间隔定时器被设置为初始递减值 tdel,当此值变为 0 时,将产生间隔时钟中断,调用 timeout() 函数进行处理。

利用间隔定时器,函数 delay(tdel) 可实现如下:

```
delay(tdel) {
    set_timer(tdel); //硬件间隔定时器设置为 tdel
    P(delsem);      //进程执行 P 操作而发生阻塞,等待中断
}
//delsem 为二元信号量,其初值为 0
timeout() {
    V(delsem);      //中断产生,执行 V 操作以释放进程
};
```

(3) 逻辑定时器

硬件间隔定时器可供所有进程共享,当存在多个进程在一个时间段内都需要延迟执行时,操作系统将按照进程阻塞时间的升序排成延时队列,队首指针可放在实时时钟的描述数据结构中,相当于为每个进程创建逻辑定时器。为此系统需要提供以下函数:

- ① tn = create_ltime(): 创建逻辑定时器,在 tn 中存放返回标识符。
- ② destroy_ltime(tn): 撤销 tn 所标识的逻辑定时器。
- ③ set_ltime(tn, tv): 将 tv 值装入逻辑定时器 tn 中,当其值为 0 时,产生时间到中断。

使用硬件间隔定时器实现进程的逻辑定时器的常用方法有以下两种。

① 使用带有绝对唤醒定时器的优先级队列

此方法使用绝对时钟和间隔定时器,被阻塞进程的唤醒时间保存在定时器的优先级延时队列中,每个队列元素都是三元组(p, tn, wakeup),其中,p 是进程标识,tn 是进程的逻辑定时器标识,wakeup 是未来时间值(即进程阻塞结束的时刻)。优先级延时队列对元素进行排序,wakeup 值最小的进程其优先级最高。图 2.5 上半部分是定时器优先级延时队列的例子,为了表达清晰

起见,元素中省略了逻辑时钟标识,进程 P1 希望在标准时刻 115 被唤醒,进程 P2、P3、P4 以此类推。图 2.5 中本地时钟的当前时间为 103,间隔时钟的当前值为 12,下一个中断将在 12 个时间单位后产生,即在时刻 115 产生。

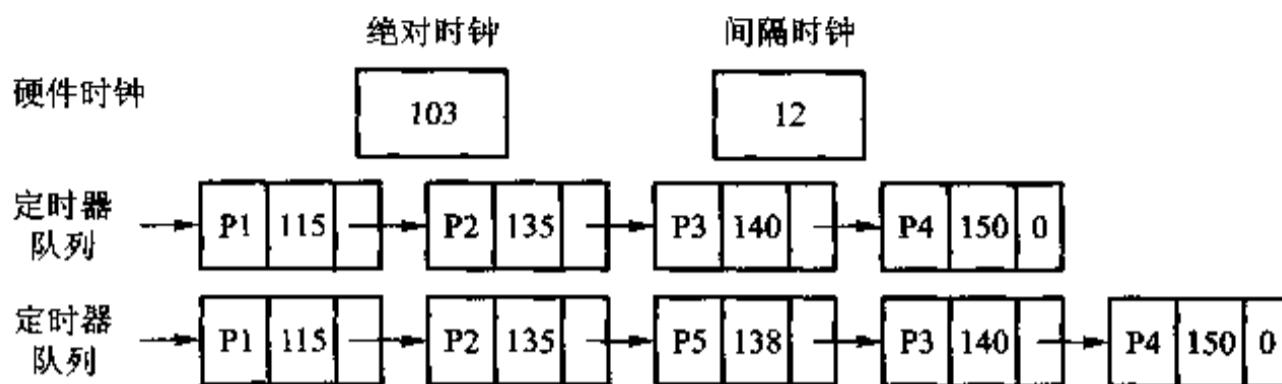


图 2.5 绝对唤醒定时器队列

需要阻塞 tdel 延时值的进程进入定时器优先级延时队列的算法如下。

步骤 1: 使用 `get_time()` 函数, 利用计算机本地时钟来获得当前时间;

步骤 2: `wakeup = 当前时间 + tdel`, 用以确定新元素移入队列的时刻;

步骤 3: 如果新元素的唤醒值小于队列第一个元素的 `wakeup` 值, 则新元素移入队列头部, 同时使用 `set_timer(tdel)` 将硬件间隔定时器设置为新的延时值 tdel;

步骤 4: 否则, 按照新元素的唤醒值将其移入队列中的适当位置。

上述算法必须在临界区执行, 设置间隔时钟的过程中禁止产生定时器中断。考察图 2.5 下半部分的例子, 假设新进程 P5 发出函数调用 `set_ltimer(tn, 35)`, 由于它所希望的唤醒时刻为 $103 + 35 = 138$, 则在新元素移入 P3 元素前面之前, 间隔时钟的值保持不变。但是如果调用函数 `set_ltimer(tn, 5)`, 所创建的新元素将成为此队列的当前头部, 且间隔时钟被设置为 5。

至此, `delay(tdel)` 函数可利用逻辑定时器来实现, 这与硬件间隔时钟的效果相同, 它使用函数 `set_ltimer(tn, tdel)` 为逻辑定时器 tn 装载 tdel 值, 再将调用进程阻塞在 `P(delsem)` 操作上。当硬件间隔时钟到达值 0 时, 时钟中断处理程序为定时器优先级延时队列的第一个元素服务, 移出此元素, 并用 `V(delsem)` 唤醒相应的进程, 而且也为硬件间隔定时器设置新值, 此值由队列中下一个元素的 `wakeup` 值减去当前时间(由 `get_time()` 函数获取)而得到。

② 使用带有时间差值的优先级队列

如果不使用绝对时钟, 仅利用间隔时钟也可以实现定时器优先级延时队列, 其基本思路是: 记录相邻元素的唤醒时间之差值, 而不记录唤醒的绝对时间。图 2.6 上半部分给出实现这种定时器延时队列的例子, 进程 P1 在队首, 12 个时间单位后, 间隔时钟值减为 0, 进程 P1 被唤醒。进程 P2 会在 P1 之后 20 个时间单位被唤醒, 依此类推。这样, 时钟中断处理程序的任务很简单, 把定时器队列的队首移走, 硬件间隔时钟被设置为下一个元素中所存储的延时值。

`set_ltimer(tn, tdel)` 会变得复杂, 其任务是查找左、右两个元素, 在它们之间插入新元素, 这个工作是通过累加元素中的差值来完成的。从间隔时钟的当前值开始, 直至到达 tdel 值; 然后, 把右元素中当前存储的差值分为两部分: 一部分被存储在新元素中, 并且代表左元素与新元素之

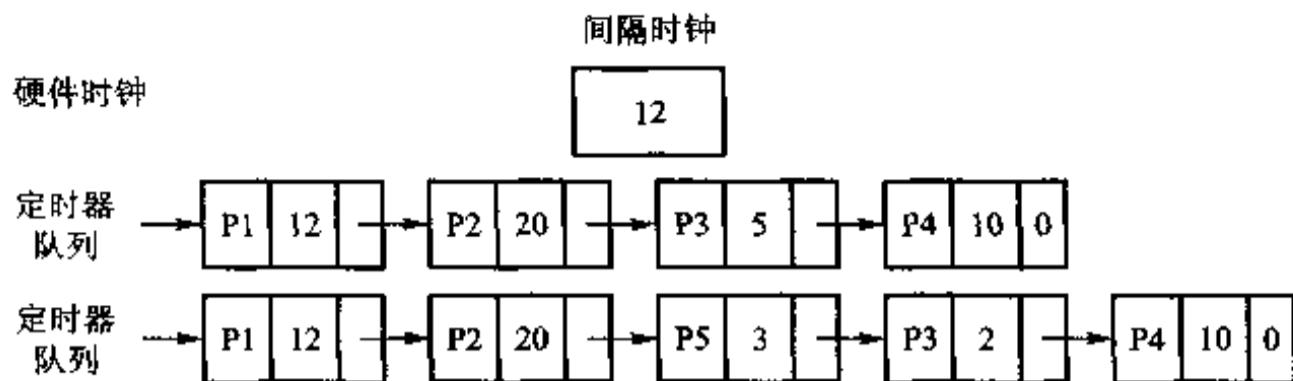


图 2.6 带有时间差值的定时器队列

间的差值，另一部分被存储在右元素中，代表新元素与右元素之间的差值。例如，如果此时进程 P5 执行函数 `set_ltimer(tn, 35)`，P5 的元素被插入进程 P2 和 P3 的元素之间，因为 P5 的唤醒时间位于这两个进程的累加元素中的差值 32 和 37 之间。进程 P3 中最初的差值(5)被分成两部分 3 和 2，其中，3 被存储在 P5 对应的新元素中，2 被存储在右邻进程 P3 的元素中。

2.2.5 中断优先级和多重中断

1. 中断优先级

中断是随机发生的，在计算机执行的每一瞬间，都可能有几个中断事件同时出现。例如，发生机器故障中断的同时可能产生设备中断请求，这时，中断装置如何响应这些同时发生的中断呢？一般来说，以不发生中断丢失为前提，把紧迫程度相当的中断源归为同一级别，紧迫程度差别大的中断源归于不同级别，级别高的中断有优先获得响应的权利。中断装置所预设的响应顺序称为中断优先级。优先级是按照中断请求的轻重缓急程度，若得不到及时响应将造成计算机出错的严重程度来界定的。其重要性在于，如果系统正在执行某优先级的中断服务程序，那么只有更高优先级的中断请求才能中断此服务程序。例如，机器故障中断表明已经产生硬件故障，对计算机和当前任务造成的影响最大，因而其优先级最高；至于设备中断事件，其优先级可降低，这样做只会推迟一次 I/O 启动或推迟处理一个 I/O 中断事件，对系统正确工作的影响不大；同样是设备，可设定高速设备的优先级高，慢速设备的优先级低，以提高高速设备的利用率。

当某一时刻有多个中断源或中断级提出中断请求时，中断系统如何按照预先规定的优先顺序予以响应呢？可以使用硬件和软件的方法，前者根据排定的优先级顺序做一个硬件链式排队器，当产生高一级的中断事件时，应该屏蔽比它优先级低的所有中断源；后者编写一个查询程序，依据优先级顺序从高到低进行查询，一旦发现有中断请求，便转入相应的中断事件处理程序入口。

在 IBM 系列机中，硬件排定的中断优先级响应顺序从高到低是：机器校验中断、自愿性中断、程序性中断、外部中断、I/O 中断；在 Intel x86 体系结构中，最多允许有 256 个中断或异常信号，这些信号分为 5 类，硬件排定的中断和异常优先级响应顺序从高到低是：复位、异常、软件中断、非屏蔽中断和可屏蔽中断。

需要注意的是,中断优先级只是表示中断装置响应中断的次序,在系统设计时已经确定并且不能被改变;操作系统在处理所响应的中断时也存在先后次序的问题,中断处理顺序和响应顺序未必一致,也就是说,先响应的中断可滞后处理。

2. 中断屏蔽

计算机均配置可编程中断控制器,CPU 可以通过指令设置可编程中断控制器的屏蔽码。中断屏蔽是指产生并提出中断请求后,CPU 允许响应或禁止响应的状态位,大部分中断源均有对应的中断屏蔽位,当被中断装置接收并产生一个中断请求后,如果相应的中断屏蔽位复位,暂时不能被 CPU 响应,需要等待直至中断屏蔽位置位,被屏蔽的中断才能被 CPU 响应并获得处理。某些中断是不能被禁止的,例如,电源断电中断、自愿访管中断就不能被禁止。

中断屏蔽位的作用是暂时禁止对某些中断做出响应,例如,处理某优先级中断时,防止同级或低级中断的干扰,或在执行不可分割操作及临界区时,需要防止被中断事件打断。此外,中断屏蔽位还能协调中断响应与中断处理之间的关系,保证高级中断处理能够打断低级中断处理,反之不然。

在当前 PSW 中通常已设置某些中断屏蔽位,当中断屏蔽位置位(1)时,允许响应相关的中断;当中断屏蔽位复位(0)时,禁止响应相关的中断。有了中断屏蔽功能,能够增加中断排队的灵活性,采用编程方法在某段时间内屏蔽一些中断请求,可以改变中断的响应顺序。

中断屏蔽方式在计算机设计时即决定,有的采用整级屏蔽方式,有的采用单个屏蔽方式。在 IBM 370 系统中,使用 PSW 的 0~7 位作为系统屏蔽位整级屏蔽通道及外中断,但其 36~39 程序屏蔽位可单个屏蔽程序性中断;在运行 Windows 的 Intel x86 系统中,先把所有单个中断事件排定一个中断请求优先级,称为处理器 IRQL,当内核线程运行时,可以提高或降低处理器 IRQL,如果中断源的 IRQL 等于或低于当前中断优先级,则此中断被屏蔽,直至内核降低 IRQL。

3. 多重中断事件的处理

在计算机系统的运行过程中,可能同时出现多个中断,或者前一个中断尚未处理完,接着又发生新的中断,于是 CPU 暂停正在运行的中断处理程序,转而执行新的中断处理程序,这就叫做多重中断或中断嵌套。一般来说,优先级别高的中断具有打断优先级别低的中断的中断处理程序的权利,反之却不允许优先级别低的中断干扰优先级别高的中断的中断处理程序的运行。系统如何处理多重中断呢?这是操作系统的中断处理程序所必须解决的问题。

对于多重中断,可能是同一优先级的不同中断,也可能是不同优先级的中断。对于前者,通常由同一个中断处理程序按自左至右的顺序逐个处理并清除之;对于后者,可区分不同的情况做如下处理。

(1) 串行处理

在运行一个中断处理程序时,禁止再次发生中断,这可以通过屏蔽中断来实现。例如,在执行 I/O 中断处理程序时,可以屏蔽外中断或其他 I/O 中断,甚至所有中断。这种方法简单易行,所有中断都严格地按顺序串行处理,但并未考虑相对优先级和时间限制方面的要求。

(2) 嵌套处理

对于有些必须处理且优先级更高的中断事件,采用屏蔽方法似有不妥,因此,往往允许在运行某些中断处理程序时,仍然能够响应中断,这时,系统应负责保护被中断的中断处理程序的现场,然后转向处理新中断事件的中断处理程序,以便在处理结束时可以返回原来的中断处理程序继续运行。操作系统必须预先规定,赋予每类中断以一定的优先级,允许高优先级中断打断低优先级中断的中断处理程序;还要规定嵌套的最大重数,嵌套的重数视系统规模而定,一般以不超过3重为宜,过多的“嵌套”会增加不必要的系统开销。

(3) 即时处理

在运行中断处理程序时,如果出现任何程序性中断事件,在一般情况下,表明此时中断处理程序有例外,应对其立即响应并进行处理。

2.2.6 Linux 中断处理

1. Linux 内核处理流程

如图 2.7 所示是 Linux 内核处理流程,在用户态执行的进程,因执行系统调用,或者因产生中断或异常,促使 CPU 转换为核心态,内核开始工作。假定进程在用户态执行应用程序,并且进行系统调用,那么,进程会按照调用号自陷至内核代码中的服务函数人口点,此函数由内核任务执行,执行结束后,返回至内核 `ret_from_sys_call` 代码来完成一组标准任务,分派累积的系统工作,例如,下半部分处理、信号处理以及调度其他任务。系统调用结束时,进程处于 `TASK_RUNNING` 态(就绪态),一旦 CPU 可用时进程就投入运行。如果内核无法立即完成进程所请求的工作,此进程将处于 `TASK_INTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE` 状态,如果进程被阻塞,或时间片耗尽,那么,调度程序就会选择新进程。

当中断事件发生时,正在运行的进程执行完当前指令之后,CPU 响应中断,转入相应的中断服务例程(Interrupt Service Routine, ISR)处理,区分两类中断事件:快中断和慢中断,两者的主要区别如下。

(1) 处理慢中断之前需保存所有寄存器的内容,而处理快中断仅保存那些被常规 C 函数修改的寄存器。

(2) 在处理慢中断时,通常不屏蔽其他中断信号,而处理快中断时会屏蔽其他中断。

(3) 慢中断处理完毕后,通常不返回被中断的进程,而是转向调度程序重新进行调度,调度结果未必是被中断的进程继续运行(属于抢占式调度)。而快中断处理完毕后,通常会恢复现场,返回被中断的进程继续执行(属于非抢占式调度)。

慢中断要做很多工作,如果全部在屏蔽中断状态下完成,则不能及时响应中断处理期间到达的其他中断信号,于是引入中断下半部分的概念。

2. 下半部分处理概述

中断处理程序的特点是:它以异步方式运行,有可能打断关键代码的执行,甚至打断其他中断处理程序的执行;它在屏蔽中断状态下运行,最坏的情形会禁止所有中断;它要对硬件进行操作,对

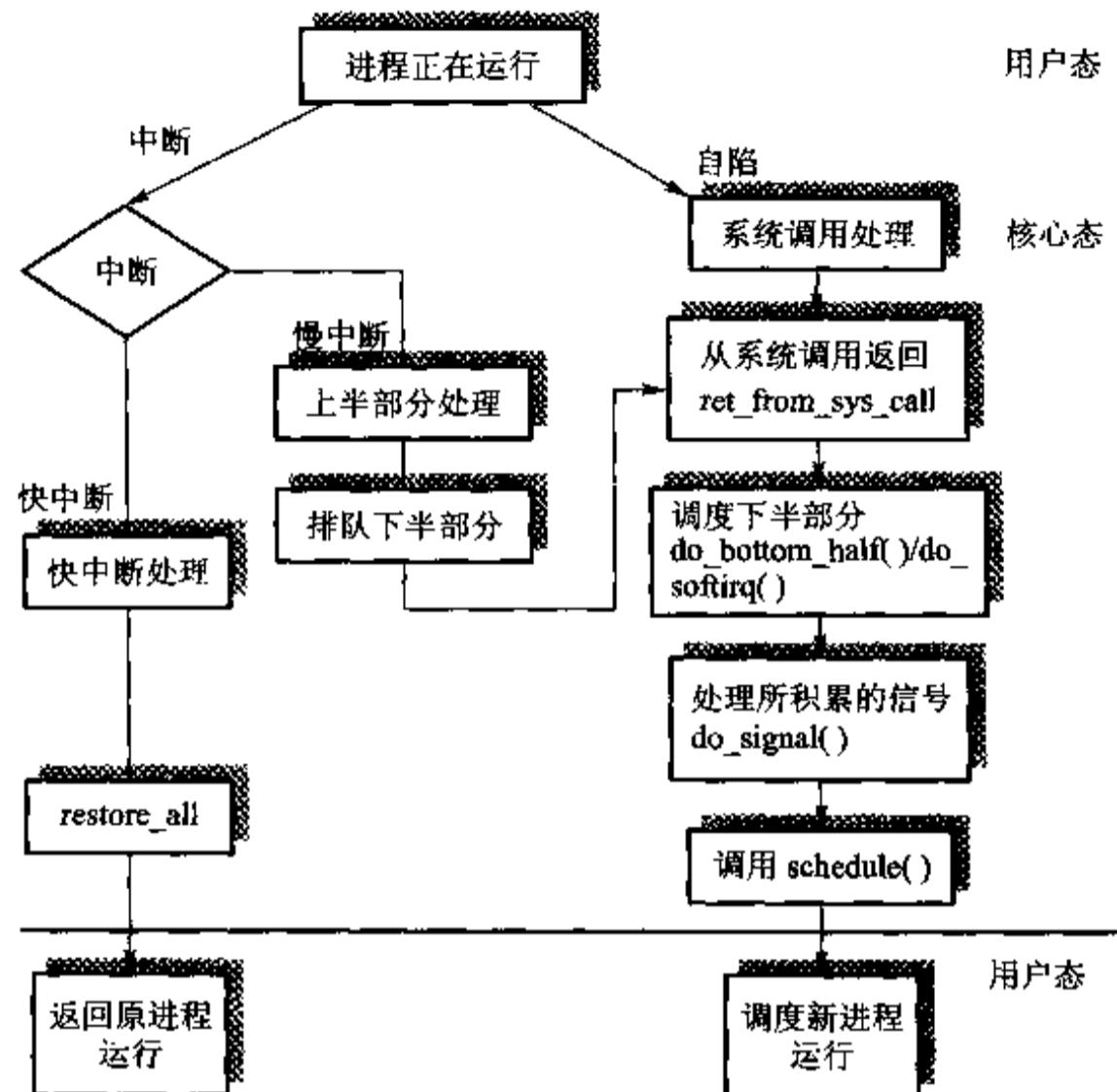


图 2.7 Linux 内核处理流程

于时限有很高的要求；它在中断上下文中运行，故不能被阻塞。凡此种种都要求中断处理程序的执行越快越好，因为缩短屏蔽中断的时间对于系统的响应能力和性能都至关重要。于是，Linux 中断服务程序被分成上半部分 (top half) 和下半部分 (bottom half) 两个处理函数。每当上半部分函数接收到中断，便立即开始工作，在关中断状态下只执行限时严格且与硬件相关的任务，如获取来自设备的数据，将其保存到预定缓冲区，然后“标记中断”，通知下半部分做剩余的工作，其他的工作由可中断代码来处理，称为中断下半部分处理，可见这是一种任务延迟处理机制。

上半部分是中断向量表中登记的中断服务程序的入口部分，在执行过程中它要决定其相关的下半部分函数是否执行。当处理快中断时，如果有其他中断到达，无论是快中断还是慢中断，它们都必须等待，否则，为了更快地处理所到来的其他中断，内核要尽可能地将耗时的处理工作放到下半部分执行，由于它是可中断的，如果在运行期间其他设备产生新的中断，则下半部分暂时被中断，等到那个设备的上半部分运行完毕，再回来运行它。

将上半部分与下半部分加以区分的另一个原因是下半部分函数含有某些中断处理并不一定非要执行的操作，对于这些操作，内核可以在一系列中断之后对其集中处理一次，在这种情况下，每次都单独执行下半部分显然是一种浪费。

与上半部分只能通过中断处理程序实现不同，下半部分可采用多种机制来实现，不同的机制由不同的子系统和接口组成，它们分别是：bottom half、task queue、tasklet、work queue 和 softirq。

3. 下半部分

Linux 最早提供的任务延迟执行机制称为 BH, 用于实现下半部分(bottom half), 提供静态创建的下半部分处理的数据结构, 建立一个函数指针数组, 采用数组索引的方式访问, 最多有 32 个不同的下半部分处理函数。bh_base 是下半部分处理函数的入口指针, bh_mask 和 bb_active 共同控制下半部分处理函数能否运行, 分别指明是否有下半部分处理函数被安装和上半部分处理函数是否标出哪个 BH 要执行。若 bh_mask 的位 n 被置位, 则 bh_base 的第 n 个元素包含一个下半部分处理函数的入口地址; 若 bh_active 的位 n 被置位, 则第 n 个下半部分处理函数能够被调度程序调度执行。只要对 bh_mask 和 bh_active 进行“按位与”运算就能表明应该运行哪个下半部分处理函数。索引在人口表中被静态地设置, 计时器下半部分处理函数的入口位于索引 0, 具有最高的优先级; 控制台下半部分处理函数的入口位于索引 1, 等等。

BH 机制存在两方面的局限性:

- (1) 下半部分处理函数的数量限制为 32 个, 且每个 BH 上只能挂接一个函数, 随着系统设备的增多, BH 的应用范围越来越广, 这个数目还不够用。
- (2) 每个 BH 在全局范围内同步, 即使属于不同的处理器, 也不允许任何两个 BH 同时执行, 这种机制使用方便但不够灵活, 安全简单但尚存在性能瓶颈。所以, 在开发 V2.5 内核版本时, BH 接口最终被抛弃。

4. 任务队列

进一步的改进方法是引入任务队列(task queue)机制, 实现对各种任务的延迟执行。为此, 内核定义一组队列, 每个队列包含一个由等待调用的函数所组成的链表, 不同队列中的函数在某个时刻会被触发执行。显然, 任务队列不一定非要与中断处理有关, 但是可以用它来替代 BH, 当驱动程序或内核相关部分要将任务排队进行延迟处理时, 可将任务添加到相应的任务队列中, 然后, 采用适当的方式通知内核执行任务队列函数。

典型的下半部分处理均有相关联的任务队列, 任务队列在 Linux 中的应用范围相当广泛, 还在其他场合被使用。任务队列与下半部分相比较, 任务队列可以动态地定义和管理, 而下半部分却由内核静态定义, 且其处理函数不能超过 32 种。由于任务队列的灵活性较差, 无法代替整个 BH 接口, 也不能胜任像网络等对性能要求较高的子系统, 此接口已经从 V2.5 版本中除去。

任务队列由 tq_struct 单向链表结构组成, 每个 tq_struct 数据结构是任务队列的结点, 包含处理函数 routine() 的地址、避免重复入队标志 sync、函数参数的指针 data 和下一个任务结点的指针 next, 当任务队列中的结点被处理时, 将调用相应的处理函数并传递参数指针。Linux 核心中的任何部分, 如设备驱动程序, 都可以建立并使用任务队列机制。内核已经预定义以下 4 个任务队列。

(1) 定时器队列(TQ_TIMER)

每次时钟滴答时刻到来时, 都要检查定时器队列是否为空, 如果不空则把定时器的下半部分标记为活动状态, 以便激活定时器任务队列机制。

(2) 即时队列(TQ_IMMEDIATE)

下半部分即时处理被一些设备驱动程序用来对任务进行排队, 此队列的优先级较低, 与定时

器队列的下半部分处理函数相比,这个队列中的任务处理要稍微迟缓一些。

(3) 进程调度队列(TQ_SCHEDULE)

当内核运行 schedule() 函数时,此任务队列会被处理,调度程序在被调度进程的上下文中运行,所以此队列中的任务很少会受到限制。

(4) 磁盘队列(TQ_DISK)

内核的主存管理使用磁盘 I/O 请求队列。

run_task_queue() 函数用于启动在某队列中排队的任务,通过将其挂接在 bottom half 向量表中来自动启动任务的执行。

下面通过定时器中断(0 号中断 IRQ0)的例子来说明中断服务程序的上半部分与下半部分之间的联系。定时器中断服务程序的名称是 timer_interrupt,其上半部分处理函数是 do_timer(),下半部分处理函数是 timer_bh()。当产生定时器中断时,timer_interrupt()调用上半部分处理函数 do_timer(),并做如下工作:更新全局变量 jiffies(加 1);校正未被下半部分处理的时钟滴答数;执行 mark_bh(timer_bh)标记下半部分为活动状态;如果定时器队列中有任务在等待,执行 mark_bh(tq_timer)标记下半部分。此外,如果进程运行在核心态,则需递增所失去的时钟滴答。整个定时器中断的处理十分简单,这是因为很多工作已被延迟到下半部分进行处理。timer_bh()需要调用 update_times()计算与时间有关的统计数据:系统平均负载、当前时间全局变量、更新进程的 CPU 时间并修改 PCB 的有关成员。此外,还要执行定时器操作,调用新老定时器 run_old_timer()和 run_times_list(),检查并执行定时服务。

5. 小任务

在 Linux 2.3 版之后,有两种机制用来实现任务的延迟执行:tasklet 和 softirq。tasklet(小任务)也是一种下半部分机制,能够更好地支持对称式多处理器,它基于软中断来实现,但比软中断的接口要简单,锁保护的要求低;softirq 保留在对执行频率及时间要求特高的下半部分使用(如网络和 SCSI)。在大多数场合下可以使用 tasklet。

使用 tasklet 的步骤如下:

- (1) 声明 tasklet。既可以静态地创建并直接引用,也可以动态地创建并通过指针间接引用。
- (2) 编写 tasklet 处理程序。注意在运行时可以响应中断但不能发生阻塞,若使用共享数据结构需提前获取锁。
- (3) 调度 tasklet。通过调用 tasklet 调度函数,并向它传递相应的结构体参数指针,启动 tasklet 的执行。

因为 BH 是全局串行处理,不适应对称式多处理器系统环境,引入 tasklet 之后,不同的 tasklet 可同时运行于不同的 CPU 之上。当然,由系统保证相同的 tasklet 不会同时在不同的 CPU 上运行。在这种情形下,tasklet 就无须是可重入的。在新版 Linux 中,tasklet 是系统推荐的异步任务延迟执行机制。

6. 工作队列

Linux 2.5 内核引入另一种任务延迟执行机制——工作队列(work queue),与其他机制的工

作原理都不同,它把一个任务延迟,并将其交给内核线程去完成,且此任务总是在进程上下文中执行。这样,通过工作队列执行的代码能够占尽进程上下文的优势,最重要的是,工作队列允许重新调度和阻塞。如果延迟执行的任务需要阻塞、需要获取信号量或需要获得大量主存空间,那么,可以选择工作队列,否则就使用 tasklet 或 softirq。

工作队列子系统是一个用于创建内核线程的接口,由它所创建的线程负责执行由内核的其他部分排到队列中的任务,这些内核线程称为工作者线程。工作队列可以让驱动程序创建一个专门的工作者线程来处理需要延迟的任务,但是工作队列子系统已经提供一个默认的工作者线程来处理延迟任务,这样一来,工作队列最基本的表现形式就是把需要推迟执行的任务交给特定的通用线程这样一种接口。默认的工作者线程是 events/n, n 是处理器编号,每个处理器对应一个线程。许多内核驱动程序都将其下半部分交给默认的工作者线程去完成,除非一个驱动程序或子系统必须建立属于自己的内核线程,否则最好使用默认线程。

7. 软中断

迄今,Linux 沿用最早的 BH 思想,但在此机制上实现了庞大和复杂的软中断子系统——softirq,它既是一种软中断机制,又是一个框架,包括 tasklet 及为网络操作专门设计的软中断。tasklet 允许动态注册,但 softirq 在编译时静态定义。每个软中断的结构表示如下:

```
struct softirq_action {
    void (*action)(struct softirq_action *); //待执行的函数
    void *data; //传递给函数的参数
};
```

最多可以注册 32 个软中断,目前 Linux 版本预定义 6 个元素,枚举表示如下:

```
enum {
    H1_SOFTIRQ, //高优先级 tasklet
    TIMER_SOFTIRQ, //定时器下半部分
    NET_TX_SOFTIRQ, //发送网络数据包
    NET_RX_SOFTIRQ, //接收网络数据包
    SCSI_SOFTIRQ, //SCSI 下半部分
    TASKLET_SOFTIRQ, //公共 tasklet
};
```

softirq 保留给对时间要求严格的下半部分使用,其中,网络和 SCSI 属于这种情况,它们直接使用软中断。软中断的使用步骤如下:

- (1) 声明一个 softirq。在编译期间静态声明并分配索引。
- (2) 注册 softirq 处理函数。在软中断子系统进行初始化时,调用 open_softirq() 注册中断处理函数,编程时注意不能发生阻塞,但允许响应中断。
- (3) 触发软中断 softirq。raise_softirq() 函数可置标志位来激活 softirq,在中断上半部分调用。

(4) 处理 softirq。当调用 do_softirq() 函数时,将查找软中断有标志位的 softirq, 调用相应的软中断处理函数执行下半部分。

do_softirq() 共有 4 个执行时机, 分别是: 从系统调用中返回(ret_from_sys_call)时、从异常中返回(ret_from_exception)时、在调度程序中(schedule)以及处理完硬件中断之后(do_irq)。在 V2.6 内核中, do_softirq() 的执行时机有些变化, 分别是: 在处理完硬件中断之后; 在 ksoftirqd 内核线程中; 在显式检查和执行待处理的软中断的代码中, 如网络子系统中。处理软中断时将遍历所有 softirq_vec[] 元素, 依次启动其中的 action() 函数, 在中断服务程序(上半部分)中触发 softirq 是常见的形式, 它执行硬件设备相关操作, 再触发 softirq, 内核在执行完中断服务程序之后, 调用 do_softirq() 函数, 于是 softirq 开始执行下半部分任务。需要注意的是, 不允许 softirq 服务函数递归执行, 但允许同一个 softirq 服务函数的两个实例同时在两个 CPU 上并行执行, 当然, 这时 softirq 必须是可重入的。

每个处理器都有一组称为 ksoftirqd/n 的辅助处理软中断的内核线程, 例如, 在进行大流量网络通信期间, 软中断被触发的频率相当高, 内核便在特殊时刻唤醒 ksoftirqd/n, 它们就会调用 do_softirq() 进行处理。在一个双处理器机器上就有两个线程 ksoftirqd/0 和 ksoftirqd/1, 它们在最低优先级(nice=19)上运行, 这样就避免与重要任务抢夺资源, 也能够保证所累积的软中断终究会获得处理。

2.2.7 Windows 2003 中断处理

1. 陷阱调度

在 Windows 2003 中, 中断是异步事件, 随时都有可能发生。与处理器正在执行的指令无关, 中断主要由设备、时钟或定时器产生, 可以启用或禁用。异常是同步事件, 是执行某条特定指令的结果, 如主存访问错误、调试指令及被 0 除, 内核将系统调用也视为异常。硬件和软件都可以产生中断和异常, 如总线出错异常由硬件造成, 而被 0 除异常则由软件引起。同样的, 设备可以产生硬件中断, 而内核可以发出启动线程调度之类的软件中断。

陷阱调度设施(trap dispatching)是用来处理意外事件的硬件机制, 当中断或异常发生时, 硬件或软件可以检测并捕获它们, 暂停当前线程的执行, 让处理器从用户态切换到核心态, 并将控制权交给内核陷阱调度程序。陷阱调度程序将检测中断和异常的类型并进行调度, 将控制权交给相应的处理代码。如图 2.8 所示是 Windows 陷阱调度程序框架。

当陷阱调度程序被触发时, 在记录机器状态期间将禁用中断, 并创建陷阱帧(trap frame)来保存被中断线程的运行现场, 以便适时恢复线程的执行。陷阱调度程序本身可以处理一些事件, 如虚地址异常, 但大多数情况下它只判断和确定所发生的情况, 并把控制权转交给内核的其他部分或执行体。例如, 如果是设备中断事件, 将把控制权转交给设备驱动程序的中断服务例程 ISR; 如果是调用系统服务事件, 将把控制权转交给执行体中的系统服务代码; 其他异常事件则由内核自身的异常调度器发送并由相应的处理程序进行处理。

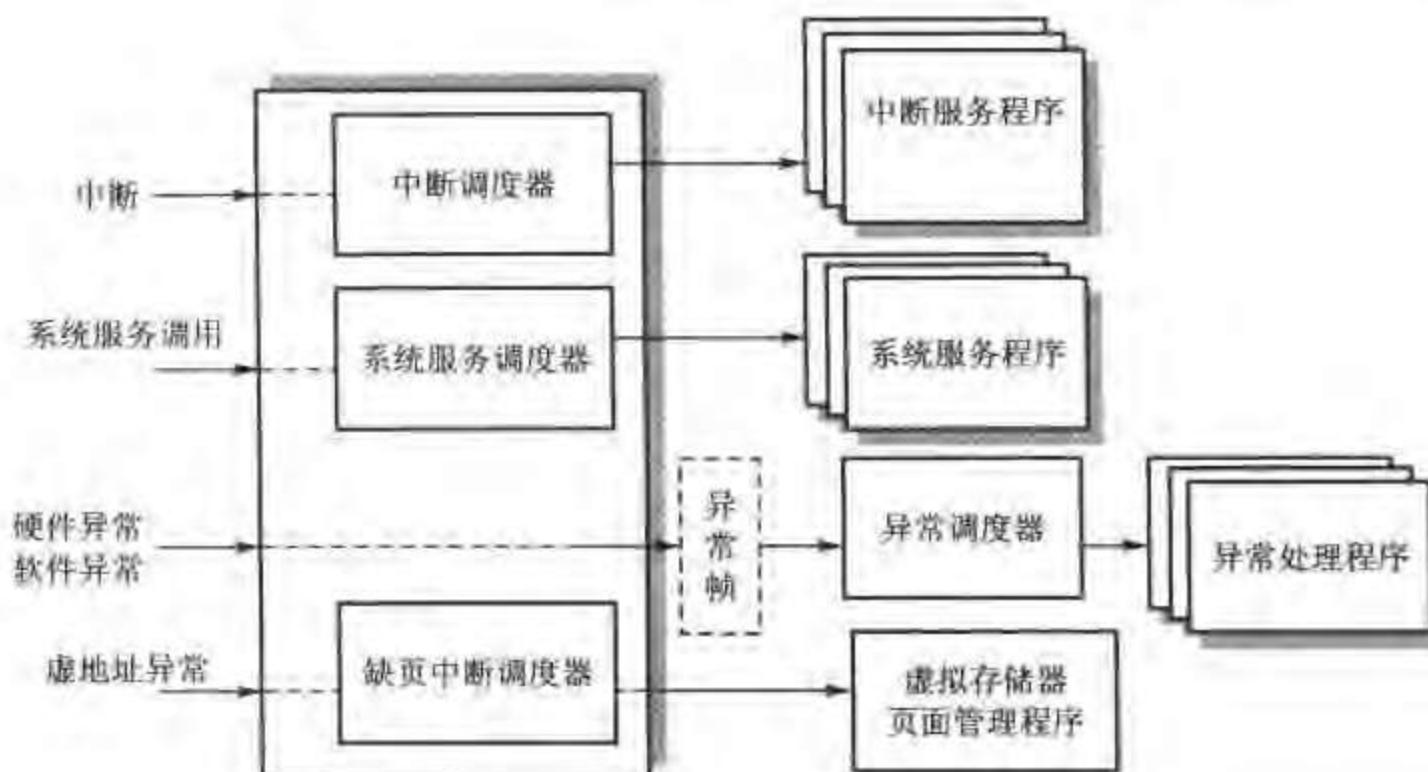


图 2.8 Windows 陷阱调度程序框架

2. 中断类型和优先级

中断由中断调度器响应，它确定中断源并将控制权转交给设备的中断服务例程，或转交给响应中断的内核程序。设备驱动程序提供设备的中断服务程序，内核提供其他类型的中断处理程序。不同处理器的中断机制不尽相同，中断调度器将硬件级中断映射到由操作系统识别的中断请求级别(Interrupt Request Level, IRQL)或称中断优先级的标准集上。

中断优先级 IRQL 与线程调度优先级的含义不同，线程调度优先级是线程的属性，而 IRQL 是中断源的属性。每个处理器都有一个 IRQL 设置，其值随着内核代码的执行而改变，运行于核态的线程可以提高或降低正在运行处理器的 IRQL，以屏蔽低级中断。

如图 2.9 所示是中断请求级别，从高优先级到设备级都为硬件中断保留；Dispatch/DPC (Deferred Procedure Call, 延迟过程调用) 和 APC (Asynchronous Procedure Call, 异步过程调用) 中断是内核和设备驱动程序所产生的软件中断，其 IRQL 分别为 2 和 1，它们启动线程调度、延迟过程调用和异步过程调用的执行；普通线程运行于 0 级，允许发生所有级别的中断。

当产生中断时，陷阱调度程序将提高处理器的 IRQL 至中断源所具有的 IRQL 上，保证服务于此中断的处理器不被同级别或低级的中断抢先，被屏蔽的中断将被另一个处理器响应，或被阻挡直至 IRQL 降低到相应的 IRQL 以下时才能被处理。由于改变处理器的 IRQL 对操作系统有着十分重要的影响，故处理器的



图 2.9 中断请求级别

IRQL 只能在核心态改变。如图 2.10 所示,如果中断源的 IRQL 高于当前的 IRQL 设置,则处理器响应此中断;处理器 A 正在处理时钟中断,时钟中断以下的所有中断将被屏蔽;同理,处理器 B 上将屏蔽 0,1 和 2 级中断,直到 IRQL 降低至适当级别。

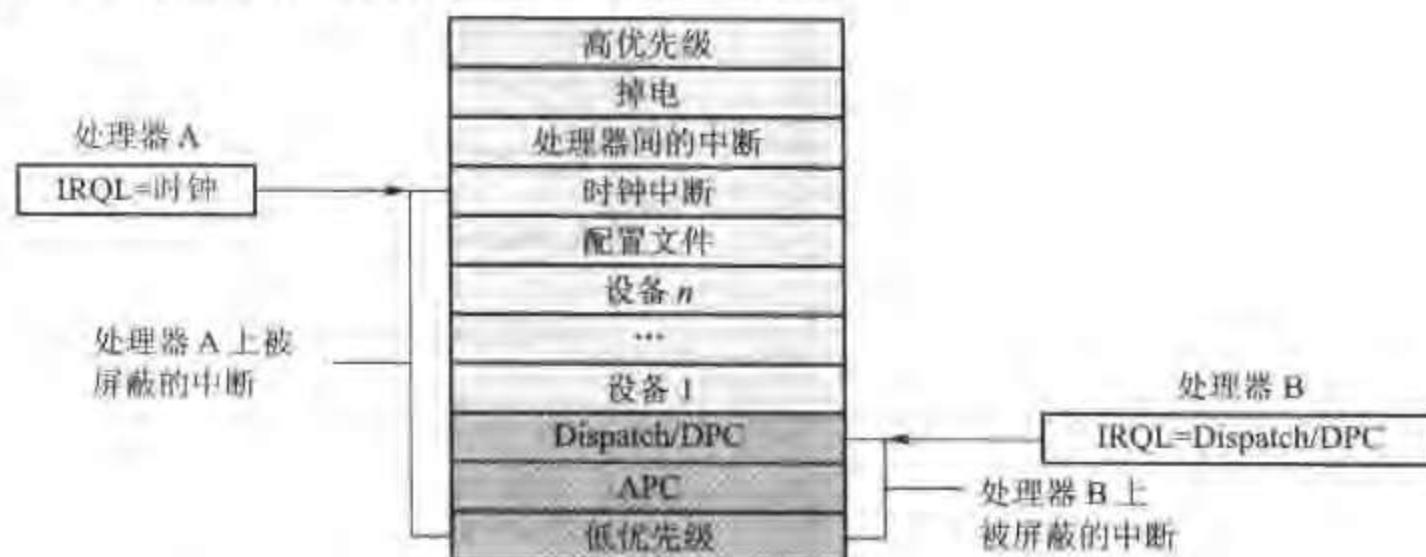


图 2.10 Windows 的中断屏蔽

3. 硬件中断处理

当中断产生时,陷阱调度程序将处理器状态保存在陷阱帧中,然后禁止中断并调用中断调度器,立即提高处理器的 IRQL 至当前中断源级别,再重新启用中断,以便屏蔽等于或低于当前中断源级别的其他中断。

Windows 使用中断分派表(Interrupt Dispatch Table, IDT)来查找处理特定中断的中断处理程序,以中断源的 IRQL 作为表的索引,找到中断服务程序的人口地址,将控制权转交给它。在中断服务程序执行结束后,中断调度器将处理器的 IRQL 降低至此中断发生之前的级别,再加载已保存的机器状态,恢复被中断线程的执行。在内核降低 IRQL 之后,被屏蔽的低优先级中断就可能出现,这样,内核将重复上述过程来处理新的中断事件。

大多数中断处理程序都在内核中,例如,内核更新时钟时间值、在产生电源级中断时关闭系统。然而,键盘和磁盘驱动器等设备也会产生中断,其种类繁多,变化很大,设备驱动程序需要有一种方法来告诉内核当设备中断发生时应调用哪个中断处理程序。为此,内核提供称为中断对象的内核控制对象,以允许设备驱动程序注册其设备的 ISR,中断对象包含内核所需要的将设备 ISR 和中断特定级别相联系的所有信息:ISR 地址、设备的 IRQL 及与 ISR 相联系的内核中的 IDT 入口地址。

将 ISR 与一个特殊的中断级别相关联,称为连接中断对象,而从 IDT 入口点切断 ISR,叫做断开中断对象。当安装设备驱动程序时,打开 ISR,也就是将 ISR 的人口地址存储在中断对象中;当卸载设备驱动程序时,关闭 ISR,也就是将 ISR 的人口地址从中断对象中清除。

4. 软件中断处理

虽然大多数中断是由硬件产生的,但是 Windows 内核也为多种任务产生软件中断,包括:启动线程调度、延迟过程调用、处理定时器到时、在特定的线程描述表中异步执行过程及支持异步

I/O 操作等。

(1) 延迟过程调用

当一个线程运行结束或进入等待状态时,内核直接调用线程调度程序进行描述表的切换。然而,有时内核在进行系统嵌套调用时,可能检测到应进行重调度,为了保证调度的正确性,内核使用延迟过程调用 DPC 软件中断来延迟调度请求的产生,直至内核完成当前的活动为止。

需要同步访问共享的内核数据结构时、内核总是将处理器的 IRQL 提高至 Dispatch/DPC 级或高于此级,禁止其他软件中断和线程调度。当内核检测到线程调度应该发生时,它将请求一个 Dispatch/DPC 级的软件中断,但是由于 IRQL 等于或高于 Dispatch/DPC 级,处理器将在检查期间保存此中断;当内核完成当前活动后,它将 IRQL 降至低于 Dispatch/DPC 级,便可出现此中断。可见,通过软件中断来激活线程调度程序是延缓调度直到条件合适为止的一种有效方法。

除了线程调度之外,内核在其他 IRQL 上也处理延迟过程调用。有一种 DPC 是执行系统任务的函数,此任务逊于当前任务,这些函数叫做“延迟函数”,因为它们可以不立即执行。DPC 为操作系统提供在核心态产生软件中断并执行系统函数的能力,内核使用 DPC 处理定时器到时、在线程时间片结束后重新调度处理器;设备驱动程序使用 DPC 完成 I/O 操作任务的处理。

延迟过程调用由 DPC 对象表示,它是用户态程序不可见的内核控制对象,但对于设备驱动程序和其他系统代码是可见的。DPC 对象所包含的最重要信息是当内核处理 DPC 中断时将调用的系统函数的地址,待执行的 DPC 程序被保存在内核管理的 DPC 队列中。为了请求一个 DPC,系统代码将调用内核来初始化 DPC 对象,然后将其放入 DPC 队列中,如图 2.11 所示。

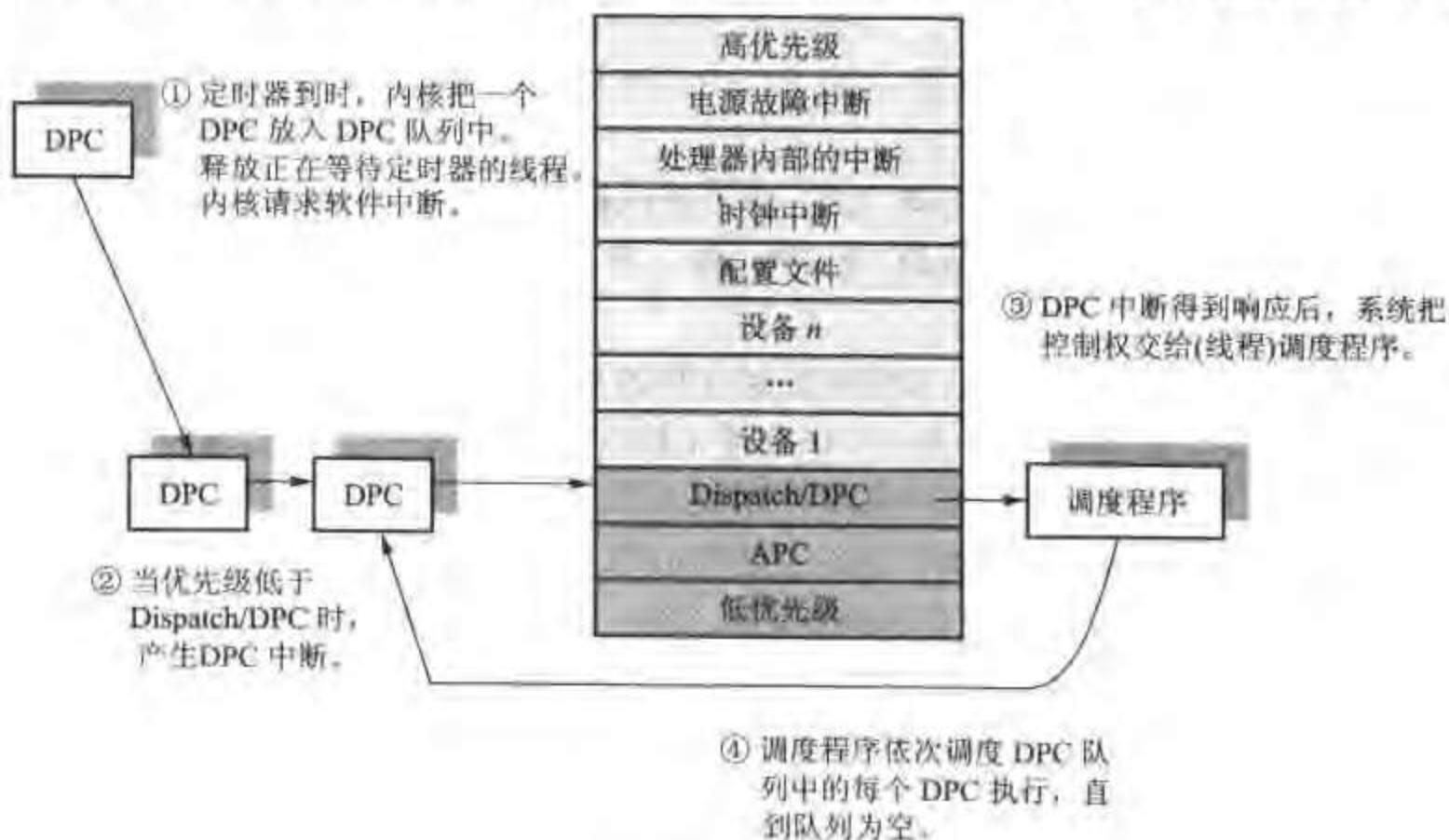


图 2.11 DPC 的提交和执行

将 DPC 放入 DPC 队列会促使内核请求一个 Dispatch/DPC 级的软件中断，通常由运行在较高 IRQL 级的软件对 DPC 进行排队，所以，被请求的中断直到内核降低 IRQL 至 APC 级或低于 APC 级时才出现。用户态线程以低 IRQL 级执行，故 DPC 能够中断用户线程的执行，且不必考虑哪个线程正在运行。DPC 程序可以调用内核函数，但不能调用系统服务、不能产生页面故障、不能创建或等待对象。

DPC 主要是为设备驱动程序提供的，把可推迟的设备中断服务程序的部分工作分离出来，放入相应的 DPC 中延迟处理；内核使用 DPC 处理时间片到时，系统时钟的每个滴答在时钟 IRQL 级都会产生一个中断，运行在时钟中断级的中断处理程序更新系统时间，并减小用来记录当前线程运行时间的计数器值，当计数器值为 0 时，线程时间片到时，内核可能需要重新调度处理器，这是一个应在 Dispatch/DPC 级完成的低优先级任务，完成之后将降低处理器的 IRQL。因为 DPC 中断的优先级低于设备中断的优先级，所以挂起的任何设备中断将在 DPC 中断产生之前得到处理。

(2) 异步过程调用

APC 用于中断一个特定线程的执行，为应用程序和系统代码提供一种在特殊线程描述表（特殊地址空间）中执行代码的方法。APC 在特殊线程描述表中执行，使用专用的 APC 队列，以低于 2 级的 IRQL 运行，所受的限制与 DPC 存在很大的不同。APC 程序可以获得对象（资源）、等待对象句柄及调用系统服务。

APC 也是内核控制对象，称为 APC 对象，等待执行的 APC 在内核管理的 APC 队列中。APC 队列与 DPC 队列的不同之处在于：DPC 队列是系统范围之内的；APC 队列是特定于线程的，每个线程都有自己的 APC 队列。当要求内核对 APC 排队时，内核将 APC 插入将要执行 APC 的线程的 APC 队列中，内核依次请求 APC 级的软件中断，并在线程开始运行时执行 APC。

APC 可分为用户态 APC 和核心态 APC，后者在线程描述表中运行且无须得到目标线程的“允许”，而前者则需要得到目标线程的“允许”。核心态 APC 可以中断线程及执行过程，而不需要线程干预或同意。

执行体使用核心态 APC 来执行必须在特定线程的地址空间（描述表）中完成的操作系统工作。例如，可使用核心态 APC 来命令一个线程停止执行可中断的系统服务，或将异步 I/O 操作的结果记录在线程地址空间中；环境子系统使用核心态 APC 将线程挂起或终止线程的运行，或设置它的用户态执行描述表；POSIX 子系统使用核心态 APC 来模仿 POSIX 信号到 POSIX 进程的发送。设备驱动程序也使用核心态 APC，例如，启动一个 I/O 操作且线程进入等待状态，则另一个进程中的线程就被调度运行。当设备完成数据传输时，I/O 系统必须以某种方式恢复进入启动 I/O 系统线程的描述表中，以便将 I/O 操作的结果复制到此线程地址空间的缓冲区中。一些 Win32 API，如 ReadFileEX、WriteFileEX 和 QueueUserAPC，使用用户态 APC，ReadFileEX 和 WriteFileEX 允许调用者指定 I/O 操作完成时所要调用的完成程序，此完成程序是通过把 APC 排队到发出 I/O 操作的线程来实现的。

5. 异常调度

异常是由运行程序的执行而直接产生的例外，除了简单的可由陷阱调度程序解决的异常之外，所有异常均由内核的“异常调度器”提供服务，其任务是找到能够处理此异常的异常处理程序。内核所定义的异常有：主存访问越界、被 0 除、整数溢出、浮点异常和调试程序断点。Win32 引入异常处理工具，允许应用程序在异常发生时能够得到控制，应用程序可将这个状态固定并返回至异常发生的地方展开堆栈，也可以向系统声明不能识别异常，并继续搜寻能处理异常的异常处理程序。

内核俘获和处理对应用程序透明的某些异常，如程序调试断点异常，此时内核将调用调试程序来处理这个异常。对于主存访问越界或算术溢出等异常，内核将原封不动地返回用户态程序来处理。环境子系统能够建立“基于帧的异常处理器”来处理这些异常，基于帧是指与特殊过程的激活相关的异常处理程序，当过程被调用时，代表此过程激活的堆栈帧会被推入堆栈。堆栈帧可以有一个或多个与其相关的异常处理程序，每个处理程序都保存在源程序的一个特定代码块内。当异常发生时，内核将查找与当前堆栈帧相关的异常处理程序，如果不存在，内核将继续查找与前一个堆栈帧相关的异常处理程序，如此往复，如果还未能找到，内核将调用系统默认的异常处理程序。

6. 系统服务调度

陷阱调度程序可以调度系统服务，执行 INT 2E 指令引起一个系统陷阱，从用户态切换到核心态，进入系统服务调度器(system service dispatcher)，被传递的参数指明被请求的系统服务号，内核将根据参数在系统服务调度表中查找相应的系统服务程序。

系统服务调度器将校验参数，并且将调用者的参数从线程的用户堆栈复制到核心态堆栈中，然后执行系统服务。如果传递给系统服务的参数指向用户空间中的缓冲区，则在核心态代码访问用户缓冲区之前，必须查明这些缓冲区的可访问性。

每个线程都有一个指向系统服务表的指针，Windows 设有两个内置的系统服务表，一个默认表在 NTOSKRNL.EXE 核心执行系统服务，另一个表在 Win32 子系统 Win32K.SYS 的核心态实现 Win32 USER 及 GDI 服务。用于执行服务的系统服务调度命令存放于系统动态链接库 NTDLL.DLL 中，子系统的动态链接库通过调用 NTDLL 中的函数来实现其系统服务。出于执行效率的考虑，Win32 USER 及 GDI 的系统服务调度命令没有经过 NTDLL.DLL，而是在 USER32.DLL 和 GDI32.DLL 中直接实现。

进程及其实现

2.3.1 进程的定义和属性

“进程”是操作系统中最基本、最重要的概念，是在多道程序系统出现后，为了刻画系统内部的动态状况、描述运行程序的活动规律而引进的新概念，所有多道程序设计操作系统都建立在进

程的基础上。从理论角度看,进程的概念是对当前运行程序的活动规律的抽象;从实现角度看,进程是一种数据结构,用来准确地刻画系统动态变化的内在规律,有效地管理和调度在计算机系统主存运行的程序。基于进程的概念及其衍生思想,应用程序和操作系统都可以分解为逻辑上并发和物理上并行的程序。首先讨论操作系统中引入进程的目的。

一是刻画系统的动态性,发挥系统的并发性。在多道程序环境下,处理器在各程序之间来回切换,程序是并发执行的,程序的任意两条指令之间都可能发生事件而引发程序切换,因而程序的执行可能不是连续的而是走走停停的。此外,程序的并发执行又会引发资源共享和竞争问题,造成并发执行的各个程序之间可能存在制约关系,执行的程序不再处于封闭式环境中,出现许多新特征,而“程序”自身只是计算任务的指令和数据的描述,这种静态的概念无法刻画程序的并发性,系统要寻找能够描述程序动态执行过程的概念,这就是进程。进程是并发程序设计的一种有力工具,操作系统中的进程概念能较好地刻画系统内部的“动态性”,发挥系统的“并发性”,从而提高资源利用率。

二是解决共享性,正确地描述程序的执行状态。首先,引进“可再入”程序的概念,所谓“可再入”程序是指能够被多个程序同时调用的程序;另一种称为“可再用”的程序在被调用过程中可以自身修改,在调用它的程序退出之前是不允许其他程序来调用的。“可再入”程序是纯代码,在执行过程中不被修改;调用它的各应用程序提供工作区。因此,“可再入”程序可同时被几个应用程序调用。

在多道程序设计系统中,编译程序常常是“可再入”程序,可同时编译若干源程序。假定编译程序 P 正在编译源程序_甲,P 从起始点 A 开始工作,当执行到 B 点时需要将信息记录到磁盘上,那么 P 应在 B 点等待磁盘上的数据传输完成。此时处理器空闲,为了提高系统效率,利用编译程序的“可再入”特性,让 P 再为源程序_乙进行编译,仍然从起始点 A 开始工作,现在应如何描述编译程序 P 的状态呢?称它在 B 点等待磁盘数据传输完成,还是称它处于从 A 点开始执行的运行状态? P 只有一个,但其加工对象有源程序_甲和源程序_乙两个,所以,以程序作为占用处理器的单位显然不适宜。为此,可以把 P 与服务对象联系起来,P 为源程序_甲服务就构成进程_甲,P 为源程序_乙服务则构成进程_乙,两个进程虽然共享同一个编译程序,但是这两个进程可同时执行且彼此按各自的速度独立推进。现在可以说进程_甲(在 B 点)处于“等待磁盘数据传输完成状态”,而进程_乙从 A 点开始执行并处于“运行状态”。可见,程序与程序的执行(计算)不再一一对应,延用程序的概念不能描述这种共享性,必须引入新概念,这就是进程。

把上述两点结合起来,进程是既能描述程序的并发执行状态、又能共享资源的一个基本单位,当然操作系统也要为引入进程而付出(进程占用的)空间和(调度进程的)时间代价。

进程的概念最早是 1960 年在美国麻省理工学院 MULTICS 和 IBM 公司 CTSS/360 系统中提出和实现的,直到目前为止,进程的定义和名称尚未统一,美国麻省理工学院称进程(process)、IBM 公司称任务(task)、Univac 公司称活动(action)。下面给出进程的定义:进程是可并发执行的程序在某个数据集合上的一次计算活动,也是操作系统进行资源分配和保护的基本单位。进程具有以下一些属性。

(1) 结构性

进程包含数据集合和运行于其上的程序,它至少由程序块、数据块和进程控制块等要素组成。

(2) 共享性

同一程序同时运行于不同的数据集合上时,将构成不同的进程,即多个进程可以执行相同的程序,所以,进程和程序不是一一对应的。共享性还表现在进程之间可以共享某些公用变量,通过引用公用变量就能够交换信号,从而,进程的运行环境不再是封闭的。

(3) 动态性

进程是程序在数据集合上的一次执行过程,是动态的概念,同时,进程有生命周期,由创建而产生、由调度而执行、由事件而等待、由撤销而消亡;而程序是一组有序指令所组成的序列,是静态的概念,所以,程序作为一种系统资源是永久存在的。

(4) 独立性

进程是系统中资源分配、保护和调度的基本单位,说明它具有独立性,凡是未建立进程的程序,都不能作为独立单位参与调度和运行。此外,每个进程都可以有各自独立的、不可预知的速度在处理器上推进,即按照异步方式执行,这也表现出进程的独立性。

(5) 制约性

并发进程之间存在着制约关系,造成进程执行速度的不可预测性,必须对进程的并发执行次序、相对执行速度加以协调。

(6) 并发性

进程的执行可以在时间上有所重叠,在单处理器系统中可并发执行,在多处理器系统中可并行执行。对于单处理器系统而言, m 个进程 P_1, P_2, \dots, P_m 轮流占用处理器并发地执行。例如,进程 P_1 执行 N_1 条指令后让出处理器给进程 P_2 , P_2 执行 N_2 条指令后让出处理器给进程 P_3, \dots , P_m 执行 N_m 条指令后让出处理器给进程 P_1, \dots 。因此,进程的执行是可以被打断的,或者说,进程在执行完一条指令且下一条指令未执行前,可能需要被迫让出处理器,由其他若干进程执行若干条指令之后才能再次获得处理器继续执行。

2.3.2 进程的状态和转换

1. 三态模型

进程从因创建而产生直至撤销而消亡的整个生命周期中,有时占用处理器执行,有时虽然可以运行但分不到处理器,有时虽然处理器空闲但因等待某个事件发生而无法执行,这一切都说明进程和程序不同,进程是活动的且有状态变化,状态及状态之间的转换体现进程的动态性。为了便于系统管理,一般来说,按照进程在执行过程中的不同情况至少要定义三种进程状态。

(1) 运行态(running):进程占用处理器运行的状态。

(2) 就绪态(ready):进程具备运行条件,等待系统分配处理器以便其运行的状态。

(3) 等待态(wait):又称阻塞态(blocked)或睡眠态(sleep),是指进程不具备运行条件,正在等待某个事件完成的状态。

处于运行态的进程个数不能大于处理器的个数,处于就绪态和等待态的进程可以有多个。进程通常在创建之后处于就绪态,进程在执行过程中的任一时刻必然处于上述三种状态之一,进程在执行过程中其状态将发生改变。如图 2.12 所示是进程的状态转换。

处于运行态的进程会因出现等待事件而进入等待态,当等待事件完成之后,等待态进程转入就绪态,处理器调度将引发运行态和就绪态进程之间的切换。引起进程状态转换的具体原因有以下几点。

(1) 运行态 - 等待态:运行进程等待使用某种资源或某事件发生,如等待设备传输数据或人工干预。

(2) 等待态 - 就绪态:所需资源得到满足或某事件已经完成,如设备传输数据结束或人工干预完成。

(3) 运行态 - 就绪态:运行时间片到时或出现更高优先级的进程时,当前进程被迫让出处理器。

(4) 就绪态 - 运行态:当 CPU 空闲时,调度程序选中一个就绪进程执行。

2. 五态模型

在很多系统中,增加两个进程状态:新建态(new)和终止态(exit)。

新状态的引入对于进程管理非常有用,新建态对应于进程被创建时的状态,进程尚未进入就绪队列,创建进程要通过两个步骤:首先,为新进程分配所需资源,建立必要的管理信息;然后,设置此进程为就绪态,等待被调度执行。必须指出的是,有时将根据系统性能的要求或主存容量的限制推迟新建态进程的提交。

终止态是指进程完成任务,到达正常结束点,或因出现无法克服的错误而异常终止,或被操作系统及有终止权的进程所终止时所处的状态。处于终止态的进程不再被调度执行,下一步将被系统撤销,最终从系统中消失。类似地,进程终止也要通过两个步骤实现:首先,等待操作系统或相关进程进行善后处理(如抽取信息),然后,回收被占用的资源并由系统删除进程。

在一个实际操作系统中,为了方便管理和调度,往往设置多种进程状态,如 UNIX 进程状态分为 9 种,Linux 进程状态分为 6 种。

3. 具有挂起功能的进程状态及其转换

到目前为止,总是假设所有进程都在主存中。事实上,可能出现这样一些情况,由于不断地创建进程,系统资源特别是主存资源已经不能满足进程运行的要求,此时必须把某些进程挂起(suspend),置于磁盘对换区中,释放其所占用的某些资源,暂时不启用低级调度,起到平滑系统负载的目的;也可能系统出现某种故障,需要暂时挂起一些进程,以便在故障消除之后,解除挂起并恢复进程的运行;用户在调试程序的过程中,可以请求挂起其进程,以便进行某种检查或修改。总之,引起进程挂起的原因多种多样。

如图 2.13 所示是具有挂起进程功能的系统中的进程状态,两个新状态是挂起就绪态(ready

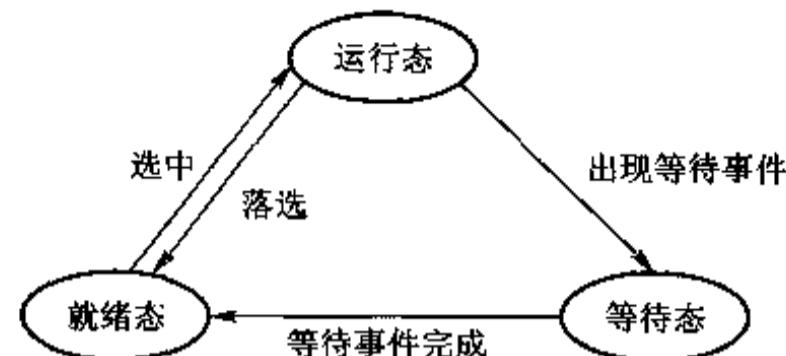


图 2.12 进程三态模型及其状态转换

suspend)和挂起等待态(blocked suspend)。挂起就绪态表明进程具备运行条件,但目前在辅助存储器中,只有当进程被对换到主存时才能调度执行;挂起等待态则表明进程正在等待某一事件发生且进程在辅助存储器中。

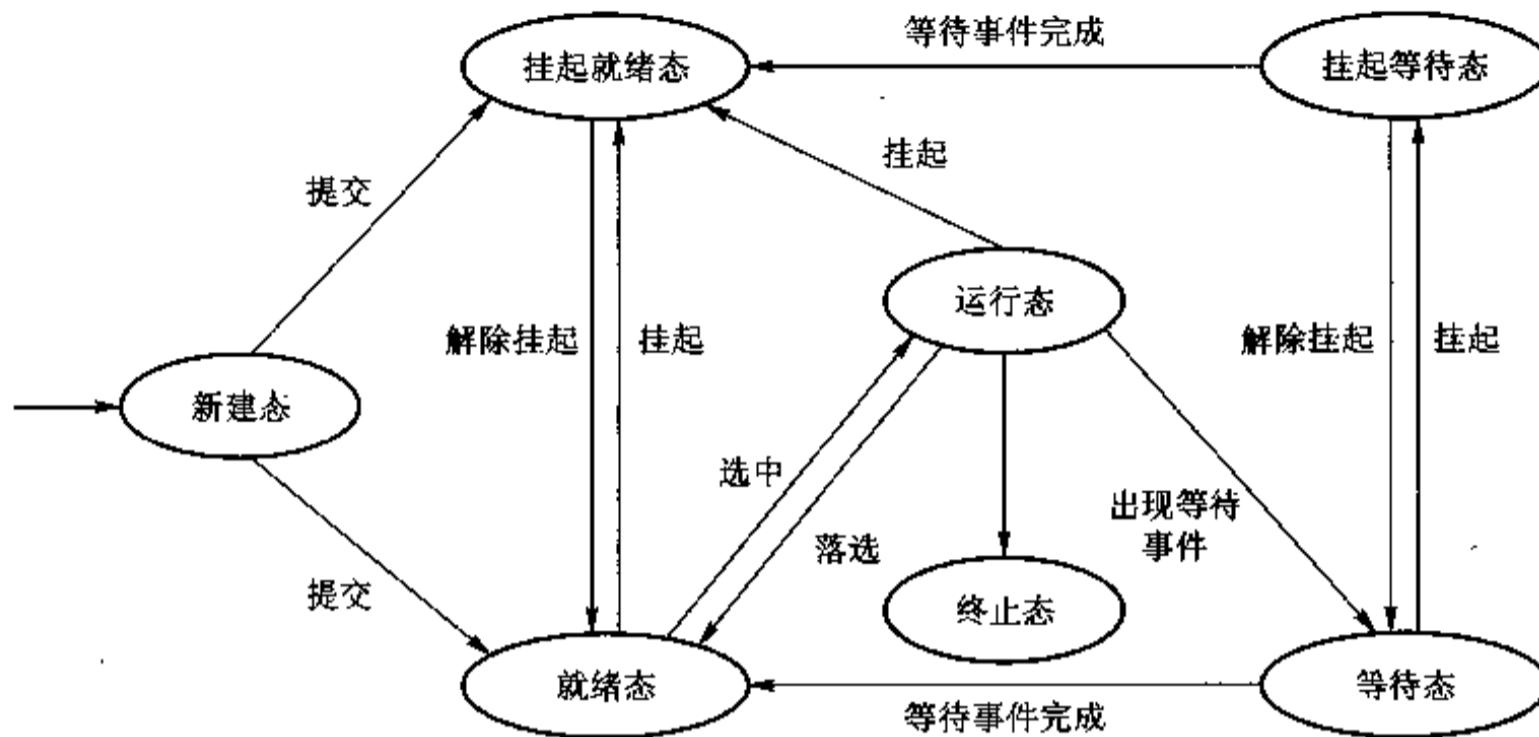


图 2.13 具有挂起进程功能的系统的进程状态及其状态转换

引起进程状态转换的具体原因如下。

- (1) 等待态→挂起等待态:如果当前不存在就绪进程,系统根据资源分配状况和性能要求,选择等待态进程对换出去,使之处于挂起等待态。
- (2) 挂起等待态→挂起就绪态:导致进程等待的事件完成后,相应的处于挂起等待态的进程将转换为挂起就绪态。
- (3) 挂起就绪态→就绪态:当主存中不存在就绪态进程,或者挂起就绪态进程具有比就绪态进程更高的优先级,系统将把挂起就绪态进程换回主存并转换成就绪态。
- (4) 就绪态→挂起就绪态:系统根据当前资源分配状况和性能要求,决定把就绪态进程对换出去,使之处于挂起就绪态。
- (5) 挂起等待态→等待态:进程等待事件发生时,原则上无须将其调入主存,但当某些进程撤销后,主存拥有足够的自由空间,而某个挂起等待态进程具有较高的优先级,且系统得知导致它阻塞的事件即将结束,便可能发生这一类状态变化。
- (6) 运行态→挂起就绪态:当一个具有较高优先级的挂起等待态进程所等待的事件完成后,它需要抢占 CPU,而此时主存空间不够,可能会导致正在运行的进程转换为挂起就绪态。另外,运行态进程也可自我挂起。
- (7) 新建态→挂起就绪态:考虑系统当前资源分配状况和性能要求,决定将新建进程对换出去,使之处于挂起就绪态。

不难看出,挂起进程等同于不在主存的进程,因此,挂起进程不会参与低级调度直到它们被对换进主存。挂起进程具有以下特征:此进程不能立即执行;此进程可能会等待某事件发生,所

等待的事件独立于挂起条件,事件结束并不能导致进程具备可执行条件;此进程进入挂起状态是由于操作系统、父进程或进程自身阻止其运行;进程挂起状态的结束命令只能通过操作系统或父进程发出。

2.3.3 进程的描述和组成

1. 进程映像

进程的活动包括占用处理器执行程序以及对相关的数据进行操作,因而,程序和数据是进程必需的组成部分,两者刻画进程的静态特征。还需要数据结构来刻画进程的动态特征,描述进程状态、占用资源状况、调度信息等,通常使用一种称为进程控制块的数据结构。由于进程的状态在不断发生变化,某时刻进程的内容及其状态集合称为进程映像(process image),包括以下一些要素。

(1) 进程控制块

每个进程捆绑一个控制块,用来存储进程的标志信息、现场信息和控制信息。进程创建时建立进程控制块,进程撤销时回收进程控制块,它与进程一一对应。

(2) 进程程序块

进程程序块是被执行的程序,规定进程的一次运行所应完成的功能。

(3) 进程核心栈

每个进程捆绑一个核心栈,进程在核心态工作时使用,用来保存中断/异常现场,保存函数调用的参数和返回地址,等等。

(4) 进程数据块

进程数据块是进程的私有地址空间,存放各种私有数据,用户栈也要在数据块中开辟,用于在函数调用时存放栈帧、局部变量等参数。

可见,每个进程由 4 个要素组成:控制块、程序块、核心栈和数据块。如果只具备前 3 个要素,而共享用户地址空间,就称“用户线程”;如果完全没有用户空间,就称“内核线程”。

为了运行应用程序,必须要创建进程,进程运行过程中产生中断或执行系统调用时又要运行操作系统内核程序,这时系统将保存应用程序的所有现场信息及其用户栈,使之不被内核程序破坏;而内核程序在运行时,系统使用为进程所分配的核心栈,以便内核程序在函数调用或中断时存放现场信息。

实质上,进程在系统中存在及活动除了本身的实体外,还需要环境的支撑,如硬件寄存器、程序状态字寄存器、支持动态地址转换的页表和相关的核心数据结构。在操作系统中,进程物理实体和支持进程运行的环境合称进程上下文(process context),进程在其当前上下文中运行,当系统调度新进程占有处理器时,新老进程随之发生上下文切换。进程上下文由以下三部分组成。

(1) 用户级上下文

用户级上下文(user level context)由正文(程序)、数据、共享存储区、用户栈所组成,它们占用进程的虚拟地址空间。对换至磁盘的分段或页面仍然是用户级上下文的组成部分。

(2) 寄存器上下文

寄存器上下文(register context)由程序状态字寄存器、指令计数器、栈指针(机器状态决定它指向用户栈或核心栈)、控制寄存器、通用寄存器等组成。

(3) 系统级上下文

系统级上下文(system level context)由进程控制块、主存管理信息(页表或段表)、核心栈等组成。

在 UNIX 中,系统级上下文分为静态部分和动态部分,前者由 proc 结构、user 区和主存管理信息表等组成;后者由核心栈及其“上下文层”所组成。当发生中断、系统调用或进程上下文切换时,内核将对系统级上下文的动态部分(包含前一层寄存器上下文的内容)进行压入和弹出操作。机器硬件所支持的中断级数目将决定“上下文层”的数目,假设支持 5 级中断,加上系统调用和用户级,一个进程至多有 7 个“上下文层”。例如,应用进程发出一个系统调用,在执行系统调用服务程序时机器收到磁盘中断,执行磁盘中断服务程序时机器又收到时钟中断并执行时钟中断服务程序。每个中断事件产生时,内核将创建一个新的“上下文层”,并向核心栈压入当前的“上下文层”,于是核心栈中将依次压入应用程序寄存器上下文、系统调用服务程序寄存器上下文、磁盘中断服务程序寄存器上下文,而当前时钟中断服务程序正在运行。

2. 进程控制块

每个进程有且仅有一个进程控制块(Process Control Block, PCB),或称进程描述符(process descriptor),它是进程存在的唯一标识,是操作系统用来记录和刻画进程状态及有关信息的数据结构,是进程动态特征的一种汇集,也是操作系统掌握进程的唯一资料结构和管理进程的主要依据。进程控制块包括进程执行时的情况以及进程让出处理器之后所处的状态、断点等信息,一般来说,应包含以下三类信息。

(1) 标识信息

标识信息用于唯一地标识一个进程,分为用户使用的外部标识符和系统使用的内部标识号。系统中的所有进程都被赋予唯一的、内部使用的数值型进程号(通常是 0~32 768 的正整数),操作系统内核函数可通过进程号来引用 PCB。常用的标识信息包括进程标识 ID、进程组标识 ID、用户进程名、用户组名等。

(2) 现场信息

现场信息用于保留进程在运行时存放在处理器现场中的各种信息。进程在让出处理器时,必须将此时的现场信息保存到它的 PCB 中,而当此进程恢复运行时也应恢复处理器现场。现场信息包括:通用寄存器内容、控制寄存器内容、栈指针等。

(3) 控制信息

控制信息用于管理和调度进程,包括:进程调度的相关信息,如进程状态、等待事件和等待原因、进程优先级、队列指针等;进程组成信息,如正文段指针、数据段指针、进程之间的族系信息,如指向父/子/兄弟进程的指针;进程间通信信息,如消息队列指针、所使用的信号量和锁;进程段/页表指针、进程映像在辅助存储器中的地址;CPU 的占用和使用信息,如时间片剩余量、已占用 CPU 时间、进程已执行时间总和、定时器信息、记账信息;进程特权信息,如主存访问权限和处理器特权;资源清单,如进程所需的全部资源、已经分得的资源,比如主存、设备、打开文件表等。

PCB 是操作系统中最为重要的数据结构, 它包含管理进程所需要的全部信息, PCB 的集合实际上定义一个操作系统的当前状态, 其使用权和修改权均属于操作系统, 包括调度程序、资源分配程序、中断处理程序、性能监视和分析程序等。系统在创建进程时就为它建立 PCB, 当进程运行结束被撤销时, 将回收其所占用的 PCB。操作系统根据 PCB 对并发执行的进程进行控制和管理, 进程借助于 PCB 才能被调度执行。

3. 进程队列及其管理

并发系统中往往同时存在许多进程, 有的进程处于就绪态, 有的进程处于等待态, 等待的原因各不相同。进程的特征主要由 PCB 来刻画, 为了便于对进程进行管理和调度, 常常将进程的 PCB 通过某种方法组织起来, 一般来说, 把处于同一状态的所有进程的 PCB 链接在一起的数据结构称为进程队列(process queue), 简称队列。例如, 运行队列、就绪队列和等待队列。有三种通用的队列组织方式: 线性方式、链接方式和索引方式。

(1) 线性方式

操作系统根据系统内进程的最大数目, 静态地分配主存中的某块空间, 所有进程的 PCB 都组织在一个线性表中。线性方式的优点是简单易行; 缺点是它限定系统中的进程最大数, 经常要扫描整个线性表, 调度效率较低。

(2) 链接方式

对于同一状态进程的 PCB, 通过 PCB 中的链接指针将其链接成队列, 可以采用单向链接和双向链接。单向链接是在每个 PCB 内设置一个链接指针, 指出本队列中跟随着其后的下一个 PCB 在 PCB 表中的编号或地址, 编号为 0 表示队尾。双向链接是在每个 PCB 内设置两个链接指针, 其中前向链接指针指出队列中的上一个 PCB 在 PCB 表中的编号或地址, 编号为 0 表示排在队首; 后向链接指针指出队列中跟随着其后的下一个 PCB 在 PCB 表中的编号或地址, 编号为 0 表示排在队尾。

为了标志和识别队列, 系统为每个队列均设置队列标志, 存放于内核空间中。单向链接时, 队列标志指针指向队列中的第一个 PCB 在 PCB 表中的编号或地址。双向链接时, 队列标志的后向指针指向队列中的第一个 PCB 在 PCB 表中的编号或地址; 队列标志的前向指针指向队列中的最后一个 PCB 在 PCB 表中的编号或地址。如图 2.14 所示是链接方式的原理。

不同状态的进程可以排成不同的队列, 如运行队列、就绪队列和等待队列。运行队列中通常只有一个进程; 就绪队列可按照优先级或 FIFO 的原则排队, 也可按照进程优先级的高低分成多个就绪队列; 等待队列通常有多个, 对应不同的等待状态, 如等待 I/O 操作完成、等待信号量等。此外, 还可以将空闲 PCB 结构链接成自由队列以便使用。

当发生某个事件使进程的状态发生转换时, 此进程就要退出所在队列而排入另一个队列中去。一个进程从所在队列中退出的事件称为出队; 相反地, 一个进程排入指定队列中的事件称为入队。处理器调度中负责进程入队和出队工作的功能模块称为队列管理模块, 其任务是对进程的 PCB 重新排队, 并修改成相应的链接结构。

(3) 索引方式

索引方式是线性方式的一种改进,利用索引表记录不同状态进程的 PCB 地址,系统建立若干索引表,如就绪索引表、等待索引表、空闲索引表等。状态相同进程的 PCB 组织在同一张索引表中,每个索引表的表目中存放 PCB 在 PCB 表中的编号或地址,各索引表在主存中的起始地址放在内核的专用指针单元中。如图 2.15 所示是索引方式的原理。

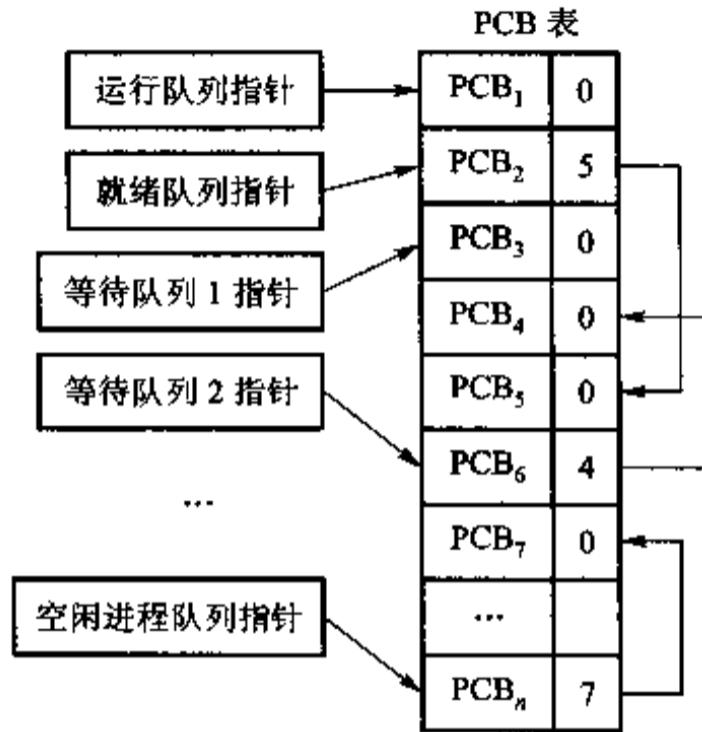


图 2.14 链接方式

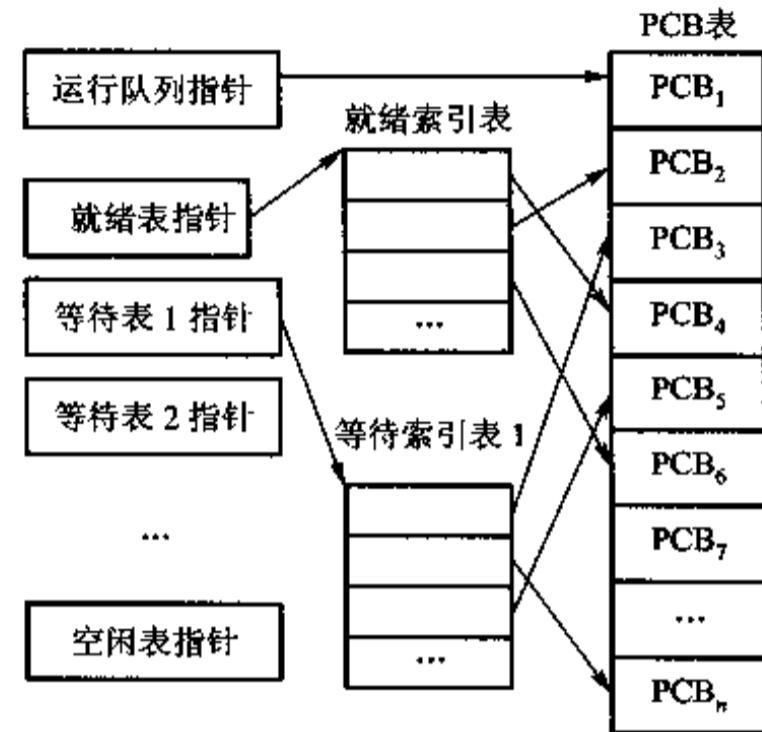


图 2.15 索引方式

下面介绍 Linux 进程链表。

Linux 中有双向循环链表、进程可运行队列链表、散列表和等待队列链表等多种组织进程控制块 task_struct 结构的方式,它们都属于上述三种方式的一种。

双向循环链表反映进程的创建顺序及进程之间的亲属关系,每个 task_struct 中的 prev_task 和 next_task 域用来实现这个结构,链表的表头是 init_task 的进程控制块,任务数组 *task 的第一个元素指向它。

当内核执行进程调度,需要查找和选择就绪进程到处理器中运行时,引入可运行状态(TASK_RUNNING)进程的双向循环链表,解决了浏览链表效率低的问题,它通过 task_struct 结构中的 prev_run 和 next_run 成员链接,而就绪进程的总数存放在 nr_running 中。

pidhash 散列表是通过 32 位 pid 来快速查找所对应的进程控制块 task_struct 结构而引入的,采用索引方式,同一表项所有进程 task_struct 的 pidhash_next 和 pidhash_prev 组成双向链表。

把 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 状态进程排入不同特定事件的等待队列的进程链表中,以备在事件发生时,由内核唤醒进程。等待队列与进程调度相结合,用于实现内核的异步事件通知机制。

2.3.4 进程切换与模式切换

1. 进程上下文切换

中断和异常是激活操作系统的仅有方法,它暂停当前运行进程的执行,把处理器切换至核心态,内核获得处理器的控制权之后,如果需要就可以实现进程切换。所以,进程切换必定在核心态而非用户态发生。内核在处理中断事件或系统调用的过程中可能会导致被阻塞的高优先级进程变为就绪态,或在处理时钟中断事件期间发现运行进程的时间片耗尽,或当前运行进程执行阻塞型 I/O 指令等,均有可能引发内核实施进程上下文切换。由于进程在让出处理器时,寄存器上下文将被保存到系统级上下文的相应的现场信息位置,这时内核就把这些信息压入核心栈的一个上下文层,当内核处理完中断返回时,或进程完成其系统调用返回用户态,内核进行上下文切换并从核心栈中弹出一个上下文层。因此,上下文切换总会引起系统级上下文层的进栈和退栈。内核在下列情况会发生上下文切换。

- (1) 当进程因各种原因进入等待态时;
- (2) 当进程完成其系统调用返回用户态,但尚无资格获得 CPU 时;
- (3) 当内核完成中断处理,进程返回用户态但尚无资格获得 CPU 时;
- (4) 当进程执行的时间片到时或进程结束时。

在执行进程上下文切换时,保存老进程的上下文且装入被保护的新进程的上下文,以便新进程运行。进程切换的实现步骤如下。

- (1) 保存被中断进程的处理器现场信息;
- (2) 修改被中断进程 PCB 的有关信息,如进程状态等;
- (3) 把被中断进程的 PCB 加入相关队列;
- (4) 选择占用处理器运行的另一个进程;
- (5) 修改被选中进程 PCB 的有关信息,如改为就绪态;
- (6) 设置被选中进程的地址空间,恢复存储管理信息;
- (7) 根据被选中进程的上下文信息来恢复处理器现场。

2. 进程上下文切换的时机

进程在运行过程中执行系统调用、产生中断或异常时,操作系统从当前运行进程那里获得控制权,此后,进程切换可以在任何时刻发生。系统在发现当前进程时间片耗尽,或执行一个阻塞型系统调用时,都会导致当前进程让出处理器,请求重新调度,内核转向低级调度执行,当低级调度程序选中新的就绪进程后,就会进行进程上下文切换。按道理说,请求重新调度事件之后,执行低级调度和进程上下文切换的工作应该连续进行,但实际上往往出于某种原因不能一气呵成。例如,在运行内核中断处理程序期间,发生一个优先级更高的 I/O 中断,当高优先级中断的处理结束之后,因为等待此 I/O 中断的进程转换为就绪态,故此时会请求低级调度。如果立即执行低级调度并做进程上下文切换,原先被高优先级中断所打断的那个低级中断处理程序尚未完成,不但会影响中断的响应时间,原先保存在 I/O 硬件中的现场信息也可能丢失;再如,若内核正在系统的临界区中执行,如果此时产生时钟中断,将导致请求低级调度的事件发生,若立即进行进程上下文切换则会带来临界区所占资源不能尽快释放的后果。此外,在各种原语操作、现场保护和恢复等过程中,即使发生请求低级调度的事件,也不能立刻执行低级调度和进行进程上下

文切换。

如果在上述过程中出现引起调度的条件，并不能立即进行调度和切换，系统将采用置请求调度标志，延迟至敏感性操作完成之后才进行低级调度和进程上下文切换。为此，Linux 在进程 task_struct 中设计重调度标志 need_resched，当需要进程重调度时先行置位，在调度时机来临时，判别标志位是否为 1，以决定是否进行调度；UNIX System V 中引入重调度标志 runrun，其作用与 need_resched 相同；Windows 操作系统为了实现这类延迟调度，专门设计延迟过程调用 Dispatch/DPC 软件中断，将 IRQL 设置为 2，以推迟线程调度的执行。

3. 处理器模式切换

与进程上下文切换有关的是 CPU 模式切换，从用户态到核心态或者从核心态到用户态的转换是 CPU 模式切换，此时仍然在同一个进程中运行。当发生中断或系统调用时，暂停正在运行的进程，把处理器状态从用户态切换到核心态，执行操作系统服务程序，这就是一次模式切换，此时进程仍在自己的上下文中执行，仅模式发生变化，内核在被中断进程的上下文中进行处理。模式切换的步骤如下：

- (1) 保存被中断进程的处理器现场信息；
- (2) 处理器从用户态切换到核心态，以便执行系统服务程序或中断处理程序；
- (3) 如果处理中断，可根据所规定的中断级别设置中断屏蔽位；
- (4) 根据系统调用号或中断号，从系统调用表或中断入口地址表中找到系统服务程序或中断处理程序的地址。

模式切换不同于进程切换，它不一定会引起进程状态的转换，在大多数情况下，也不一定引起进程切换，在完成系统调用服务或中断处理之后，可通过逆向模式切换来恢复被中断进程的运行。CPU 上所执行进程在任何时刻必定处于三个活动范围之内：

- (1) 用户空间中，处于进程上下文，用户进程在运行，使用用户栈。
- (2) 内核空间中，处于进程上下文，内核代表某进程在运行，使用核心栈。
- (3) 内核空间中，处于中断上下文，与进程无关，中断服务程序正在处理特定的中断，Intel x86 未提供中断栈，借用核心栈。

第一种情况通常对应于应用进程正常运行的状态，当请求执行系统调用时，发生第二种情况，内核接受控制，代表此进程在运行。也可以说，应用程序通过系统调用在内核空间运行，而内核运行于进程的上下文中，这种复杂的关系，即应用程序通过系统调用陷入内核，是应用程序在请求操作系统服务，完成其工作任务。在应用进程正常运行的过程中，若发生中断事件，系统将予以响应并转入相应的中断服务程序进行处理，第三种情况发生，许多操作系统的中断服务程序不在进程上下文中执行，而是在一个与所有进程毫无关联的执行环境——专门的中断上下文中执行，其目的是保证中断事件在第一时间获得响应和处理，然后尽快退出。既然不存在进程背景，中断上下文就不可能被阻塞，不能从中断上下文中调用某些引起阻塞的函数。另外，中断处理程序没有自己的栈，工作时共享被中断进程的核心栈，如果没有正在运行的进程，便会使用 idle 进程的核心栈，所以中断处理程序非常节省存储空间。需要注意的是，用户空间和内核空间

的数据不能互访,在应用程序执行系统调用的过程中,如果有需要,可通过内核函数 `copy_from_user()` 和 `copy_to_user()` 将数据在用户空间和内核空间中进行复制。

4. UNIX/Linux 中的进程切换与模式切换

在 UNIX/Linux 中,进程通常运行在用户态和核心态。如果当前运行的是应用程序或内核之外的系统程序(如 shell),对应的进程就在用户态运行;如果应用程序执行系调用,或发生中断/异常事件,就要运行系统程序,且处理器处于核心态。Linux 把内核空间中运行的程序称为任务,而把用户空间中运行的程序称为进程,且经常交替地使用这两个术语。那么,在系统中存在两种进程(任务):系统进程(任务)和用户进程(任务),实质上是指一个进程(任务)的两个侧面,系统任务是在核心态执行操作系统代码的进程,用户进程是在用户态执行应用程序的进程,应用程序通过系统调用陷入内核时,系统任务就代表用户进程开始执行。但是,这两个进程所执行的程序不同,映射到不同的物理地址空间,使用不同的堆栈。

在如图 2.16 所示的进程生命周期中,列出进程切换和模式切换的示意。事实上,进程状态在动态地转换,图 2.16 中有箭头所指的状态转换是合法的,通常进程每次完成且只完成一个合法的状态转换。在任何时刻,处理器仅执行一个进程,所以,至多只有一个进程处于状态(1)或状态(2),这两个状态相当于处理器处于用户模式和内核模式。当发生系统调用或中断时,进程通过一次模式切换从用户态运行转换为核心态运行,在核心态运行的进程是不能被抢占的。当进程在内核模式运行时,可以继续响应中断,处理中断或系统调用之后,通过一个模式切换返回用户态继续运行。也可以根据具体的情况使进程等待某个事件发生(状态(4)),只有此时才允许内核发生进程切换,使原先运行的进程让出处理器。当被等待事件完成后,进程会从等待状态被唤醒,进入就绪状态(状态(3)),若被调度程序选中,此进程会重新占用处理器运行。在多道程序设计系统中,有许多进程并发执行,如果两个以上的进程同时执行系统调用,并要求在内核模式执行,就有可能破坏核心数据结构中的信息。通过禁止任意的上下文切换和控制对于中断的响应,就能够保证数据的一致性和完整性。

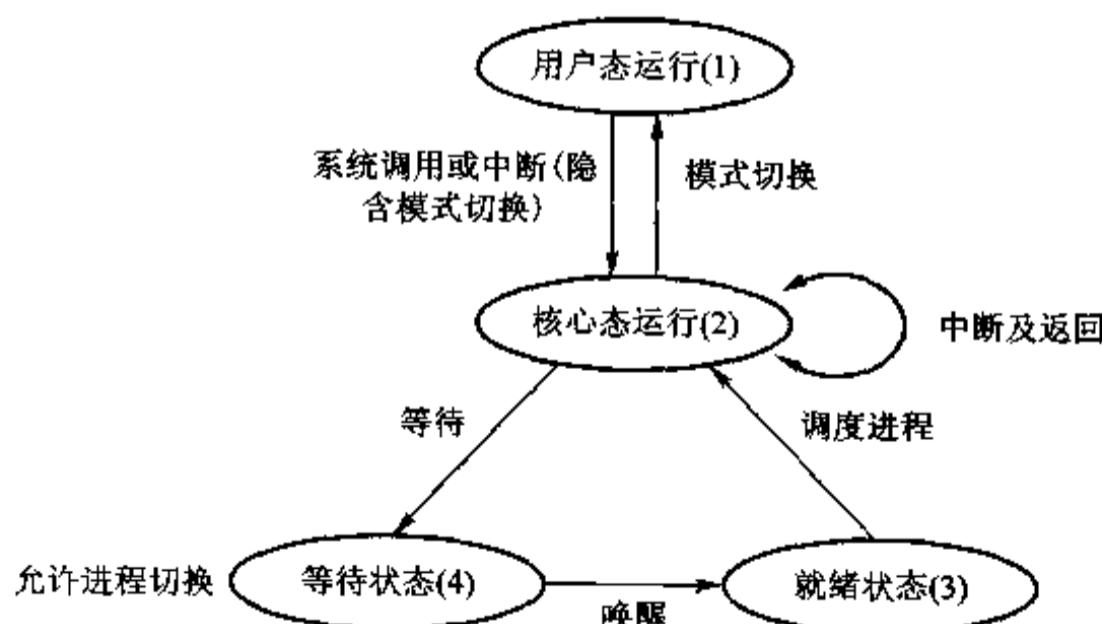


图 2.16 进程上下文切换和模式切换

2.3.5 进程的控制和管理

系统中的进程不断地产生和消亡,进程生命周期的动态变化过程由进程管理程序来控制,对于进程的控制和管理包括:创建进程、阻塞进程、唤醒进程、挂起进程、激活进程、终止进程和撤销进程等,这些功能均由系统中的原语来实现。原语(primitive)在核心态执行,是完成系统特定功能的不可分割的过程,它具有原子操作性,其程序段不允许被中断,或者说原语不能并发执行。系统对进程的控制如果不使用原语,就会造成状态的不确定性,不能达到进程控制的目的。原语的实现方法之一是以系统调用的方式提供原语接口,采用屏蔽中断的方式来实现,以保证原语操作不被中断的特性。原语和系统调用都使用访管指令(机器指令)实现,具有相同的调用形式,但原语由内核来实现,而系统调用由系统进程或系统服务程序实现(例如,文件系统调用);原语不可被中断,而系统调用执行时允许被中断,甚至某些操作系统中的系统进程或系统服务程序就在用户态运行。原语通常供系统进程或系统服务程序使用,反之决不会形成调用关系,系统进程或系统服务程序向实用程序提供系统调用,而实用程序向应用程序提供高层功能。下面介绍部分进程控制原语。

1. 进程创建

可以动态地创建新进程,当通过内核为一个程序构造 PCB 并分配地址空间后,就创建了一个进程。进程的创建来源于以下事件:

- (1) 提交批处理作业;
- (2) 有交互式作业登录终端;
- (3) 操作系统创建服务进程;
- (4) 已存在的进程创建新进程。

当用户作业被选中进入主存时,需要创建用户进程来完成这个作业。一个用户进程在请求某种服务时,也需要创建一个或多个子进程或系统进程来为其服务。例如,当用户进程读取卡片上的一段数据时,可能要求创建卡片输入机管理进程。有的操作系统把“创建”用父子进程的关系来表示,当一个进程创建另一个进程时,生成进程称为父进程,被生成进程称为子进程,父进程可以创建多个子进程,从而形成树状族系关系。一般来说,进程的创建过程如下:

- (1) 在进程列表中增加一项,从 PCB 池中申请一个空闲 PCB,为新进程分配唯一的进程标识符;
- (2) 为新进程的进程映像分配地址空间,以便容纳进程实体。由进程管理程序确定加载至进程地址空间中的程序;
- (3) 为新进程分配除主存空间以外的其他各种资源;
- (4) 初始化 PCB,如进程标识符、处理器初始状态、进程优先级等;
- (5) 把新进程的状态设置为就绪态,并将其移入就绪进程队列;
- (6) 通知操作系统的某些模块,如记账程序、性能监控程序。

不同操作系统创建进程的方式不尽相同,传统 UNIX 系统中是这样处理的:父进程使用 fork

`()` 创建子进程，此系统调用不带任何参数，子进程的行为完全由父进程和默认行为所决定，把自己的地址空间制作一个副本，其中包括 `user` 结构（包含所有打开文件的文件描述符）、正文段、数据段、用户栈和核心栈，即子进程继承父进程的全部资源；同时，系统将子进程的 `pid` 返回给父进程，向子进程返回值 0，这种实现方式使得父子进程易于通信。当然，如果子进程需要，可以使用系统调用 `execve()` 让新程序替换老程序，此后，父子进程可以自行其道。

Linux 保留传统的 `fork()` 创建子进程的方法，在用 `fork()` 创建子进程之后，父子进程存在以下关系：调用一次，返回两次，分别返回给父子进程；父子进程是独立的进程，可以并发执行；父子进程具有独立的地址空间，如果父进程改变某个变量的值，子进程将不会看到这个变化，反之亦然，因为父子进程在各自的地址空间中并发执行；父子进程可以共享文件，父进程通过复制数据结构全盘传给子进程，子进程继承父进程的所有已打开文件。

此外，Linux 还增加了两个新的系统调用。一个系统调用是 `clone()`，它带有多个参数，允许定义父子进程所共享的内容，这与线程的实现思路类似。但是如果不限定内容共享，则 `clone()` 和 `fork()` 是相同的。因而，Linux 认为线程就是共享上下文的进程。实现 `clone()` 的主要工作如下：对于 `task_struct` 结构中的 `fs` 指针、`files` 指针、`sig` 指针和 `mm` 指针并不进行复制操作，仅将这些数据结构成员中的计数器 `count` 的值加 1，这样，只有当父进程中相关数据结构成员的 `count` 值为 0 时，才会释放这些数据结构成员所占用的主存空间。此外，父子进程共享主存地址空间，采用 `copy_on_write` 策略，即仅当父进程或子进程对虚存进行写操作时，才为子进程的指针所指数据结构分配主存，并将父进程的 `mm` 指针所指数据结构的内容复制到子进程的 `mm` 指针上。虽然 `clone` 进程是共享同一主存空间的进程组的组成部分，但它们不能共享同一个用户栈，所以系统为 `clone` 进程创建独立的栈空间。

另一个系统调用是 `vfork()`，它允许子进程借用父进程的地址空间，父进程将被阻塞直至子进程执行 `execve()` 或 `exit()`，向父进程归还地址空间并唤醒它。

在 Windows 中，进程使用 Win32 API 的 `CreateProcess()` 函数创建进程，新老进程之间不存在族系关系或层次关系，两者是平等的，操作系统为新进程创建新的地址空间，并创建一个基线程。Windows 采用更为一般的做法，对于分配给进程的每个抽象资源，都有一个句柄与之关联，这意味着在创建进程时，可以把句柄的子集赋予新进程，如给予所有打开文件的句柄，若未赋予其他资源则不允许使用。新进程中的线程通常运行与父进程不一样的新程序，这意味着新进程可以有许多选择，所以，`CreateProcess()` 函数有 10 个参数，如线程可执行代码模块名指针、进程安全属性指针、线程安全属性指针、新环境块指针、当前目录名指针、进程视窗格式、启动信息指针等，在简单的情况下，大多数参数可使用默认值。在成功创建新进程之后，将返回一个子进程的句柄和基线程的句柄以便通信。

许多操作系统都有最多进程数量的限制，早期的 UNIX 允许在系统内最多创建几十个进程。在 Solaris 中，可在启动时根据检测的主存容量自动调整参数。在 Linux 2.4 中，最多进程数也是运行时可调参数，按默认值设置为：(物理主存的字节数 / 内核堆栈大小) / 2。假设一台机器有 512 MB 主存，则默认的可用进程数的上限为 $(512 \times 1\,024 \times 1\,024 / 8\,192) / 2 = 32\,768$ ，看似很

多,但对于一个 Linux 数据库服务器来说,这是一个合理的数目。

2. 进程撤销

进程完成特定的工作或出现严重错误之后,操作系统将收回其所占有的地址空间和 PCB,此时就是撤销一个进程。进程撤销可分为正常撤销和非正常撤销,前者如分时系统中的注销和批处理系统中的撤离作业步,后者如进程运行过程中出现错误或异常。进程撤销的主要原因有:进程运行结束;进程执行非法指令;进程在用户态执行特权指令;进程的运行时间超过所分配的最大时间配额;进程的等待时间超过所设定的最长等待时间;进程所申请的主存空间超过系统所能提供的最大容量;越界错误;对共享主存区的非法使用;算术错误,如除数为 0 和操作数溢出;I/O 操作故障;操作员或操作系统干预;父进程撤销其子进程;父进程撤销,其所有子进程被撤销;操作系统终止,等等。

一旦发生上述事件,系统或进程将调用撤销原语来终止进程或子进程。具体的步骤如下:

步骤 1:根据撤销进程的标识号,从相应的队列中查找并移除它;

步骤 2:将此进程所拥有的资源归还给父进程或操作系统;

步骤 3:若此进程拥有子进程,先撤销其所有子进程,以防止它们脱离控制;

步骤 4:回收 PCB,并将其归还至 PCB 池。

3. 进程阻塞和唤醒

进程阻塞是指使进程让出处理器,转而等待一个事件,如等待资源、等待 I/O 操作完成、等待某事件发生等。进程通常自调用阻塞原语来阻塞自己,所以,阻塞是进程的自主行为,是一个同步事件。当等待事件完成时,会产生一个中断,激活操作系统,在系统的控制之下将被阻塞的进程唤醒。等待事件完成是指 I/O 操作结束、某个资源可用或所期待的事件发生。进程的阻塞和唤醒显然是由进程切换来完成。进程阻塞的步骤如下:

步骤 1:停止进程执行,将现场信息保存到 PCB;

步骤 2:修改进程 PCB 的有关内容,如进程状态由运行态改为等待态等,并把状态已修改的进程移入相应事件的等待队列中;

步骤 3:转入进程调度程序,调度其他进程运行。

进程唤醒的步骤如下:

步骤 1:从相应的等待队列中移出进程;

步骤 2:修改进程 PCB 的有关内容,如进程状态改为就绪态,并将进程移入就绪队列;

步骤 3:若被唤醒的进程比当前运行进程的优先级高,重新设置调度标志。

阻塞原语和唤醒原语的作用正好相反。如果进程因某种事件调用阻塞原语来阻塞自己,则在等待事件发生后,必须由与其相关的另一进程调用唤醒原语来唤醒被阻塞进程,否则,被阻塞进程会因未被唤醒而处于永远阻塞状态。

在 UNIX/Linux 中,与进程阻塞和唤醒相关的原语主要有 sleep(暂停)、pause(暂停并等待信号)和 wait(等待子进程暂停或终止)。

4. 进程挂起和激活

当出现引起挂起的事件时，系统或进程会利用挂起原语把指定进程或处于等待态的进程挂起，其执行过程大致是：检查要被挂起进程的状态，若处于活动就绪态，就修改为挂起就绪态；若处于等待态，就修改为挂起等待态，被挂起进程 PCB 的非常驻部分要交换到磁盘对换区。

当系统资源尤其是主存资源充裕或请求激活指定进程时，系统或相关进程会调用激活原语把指定进程激活，此原语所做的工作是：把被挂起进程 PCB 的非常驻部分调入主存，然后修改其状态，挂起等待态改为等待态，挂起就绪态改为就绪态，并将进程移入相应的队列中。需要注意的是，挂起原语既可由进程自己也可由其他进程调用，但是激活原语只能由其他进程调用。

在 Windows 中，处理器的调度对象是线程，用户通过系统调用 SuspendThread 和 ResumeThread 来挂起和激活线程。一个线程可被多次挂起和多次激活，在线程控制块中有一个挂起计数器，挂起操作将使此计数器加 1，激活操作将使此计数器减 1。当挂起计数从 0 变为 1 时，线程进入等待态；当挂起计数从 1 变为 0 时，线程被唤醒并恢复运行。

2.4 线程及其实现

2.4.1 引入多线程的动机

如果说操作系统中引入进程的目的是为了使多个程序并发执行，以便改善资源利用率和提高系统效率，那么，在进程之后再引入线程的概念，则是为了减少程序并发执行时所付出的时空开销，使得并发粒度更细、并发性更好。解决问题的基本思路是：把进程的两项功能“独立分配资源”和“被调度分派执行”分离开来，前一项任务仍然由进程完成，作为系统资源分配和保护的独立单位，无须频繁地切换；后一项任务交给称作线程的实体来完成，线程作为系统调度和分派的基本单位，会被频繁地调度和切换。在这种思想的指导下，产生了多线程的概念，即多线程（结构）进程。

传统的操作系统一般只支持单线程（结构）进程，如 MS-DOS 支持单用户进程，进程是单线程的；传统 UNIX 支持多用户进程，进程也是单线程的。目前，很多操作系统都支持多线程（结构）进程，如 Solaris、Mach、SVR4、OS/390、OS/2、Windows NT、CHORUS 等；Java 的运行引擎则是单进程多线程的例子。许多计算机公司都推出自己的线程接口规范，如 Solaris 线程接口规范、OS/2 线程接口规范、Windows NT 线程接口规范等，IEEE 也推出 UNIX 类操作系统的多线程程序设计标准 POSIX 1003.4a。

2.4.2 多线程环境中的进程与线程

1. 多线程

多线程结构进程如图 2.17 所示。在此给出多线程环境中进程的定义：进程是操作系统中进

行除处理器以外的资源分配和保护的基本单位,它有一个独立的虚拟地址空间,用来容纳进程映像(如与进程关联的程序和数据),并以进程为单位对各种资源实施保护,如受保护地访问处理器、文件、外部设备和其他进程(进程间通信)。

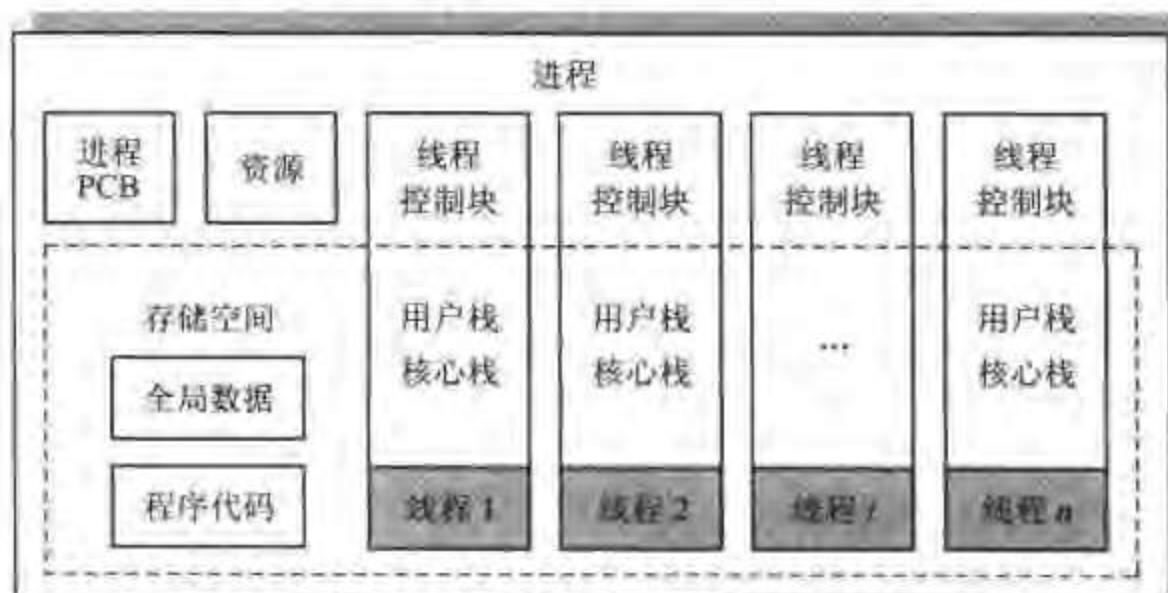


图 2.17 多线程结构进程

线程是进程中能够并发执行的实体,是进程的组成部分,也是处理器调度和分派的基本单位。允许进程包含多个可并发执行的线程,这些线程共享进程所获得的主存空间和资源,可以为完成某一项任务而协同工作。线程的组成部分有:

- (1) 线程的唯一标识符及线程状态信息;
- (2) 未运行时所保存的线程上下文;可以把线程看成进程中一个独立的程序计数器;
- (3) 核心栈,在核心态工作时保存参数,在函数调用时的返回地址,等等;
- (4) 用于存放线程局部变量和用户栈的私有存储区。

进程可以分为两个部分:资源集合和线程集合。进程要支撑线程的运行,为线程提供地址空间和各种资源,进程封装管理信息,包括对指令代码、全局数据、打开的文件和信号量等共享部分的管理;线程封装执行信息,包括对 CPU、寄存器、执行栈(用户栈与核心栈)和局部变量、过程调用参数、返回值等私有部分的管理。由于线程具有传统进程的许多特征,所以,也把线程称为轻量进程(Light Weight Process, LWP)。

2. 线程状态

与进程类似,线程也有生命周期,因而也存在各种状态。线程的状态有运行、就绪和等待,线程的状态转换也与进程类似。由于线程不是资源的拥有单位,挂起状态对于线程是没有意义的。如果进程在挂起后被对换出主存,它的所有线程因共享地址空间,也必须全部对换出去。可见由挂起操作所引起的状态是进程级状态,不是线程级状态。类似地,进程的终止将导致进程中所有线程的终止。

进程中可能有多个线程,当处于运行态的线程在执行过程中要求系统服务,如执行 I/O 请求而转换为等待态时,那么,多线程进程中是否要阻塞整个进程,对于某些线程实现机制,所在进

程也转换为等待态；对于另外一些线程实现机制，如果存在另一个处于就绪态的线程，则调度此线程运行，否则进程才会转换为等待态。显然前一种做法欠妥，丧失多线程机制的优越性，降低系统的灵活性。对于多线程进程的进程状态，由于进程不是调度单位，不必将其划分成过细的状态，如 Windows 操作系统中仅把进程分成可运行态和不可运行态，挂起状态属于不可运行态。

3. 线程管理和线程库

多线程技术利用线程库提供一整套有关线程的过程调用或系统调用来支持多线程运行，有的操作系统直接支持多线程，有的语言则提供线程库。因而，线程库可分为用户空间线程库和内核空间线程库，线程库实际上是多线程应用程序的开发和运行环境。一般来说，线程库至少提供以下功能的调用：孵化、封锁、活化、结束、通信、同步、互斥、切换（保护和恢复线程上下文），还要提供线程调度的代码，同时提供相关的 API 以支持应用程序的创建、调度、撤销并管理线程的运行。

4. 多线程程序设计的优点和线程的组织

多线程概念的引入使得单个进程可以利用 CPU 和设备之间的并行性，及多处理器之间的并行性。多线程程序设计的优点是提高系统性能，具体表现在：快速线程切换；节省主存空间；减少管理开销；通信易于实现；并发程度提高。由于队列管理和 CPU 调度是以线程为单位的，因此，涉及执行的大多数状态信息被维护在线程数据结构中。然而，存在一些影响进程中所有线程的活动，如挂起、终止等操作必须在进程级进行管理。

一个进程可以包含若干线程，这些线程可以有多种组织方式：第一种是调度员－工作者模式，进程中的一个线程担任调度员，接收和处理工作请求，其他线程是工作者线程，由调度员线程分配任务并唤醒工作者线程；第二种是组模式，进程中的各个线程都可以取得并处理此请求，不存在调度员线程。有时每个线程被设计成专门执行特定的任务，同时，建立相应的任务队列；第三种是流水线模式，线程排成某个次序，第一个线程所产生的数据传送给下一个线程进行处理，依次类推，数据按照排定的次序由线程依次传递以完成被请求的任务。

多线程技术在现代计算机软件中得到广泛的应用，多线程技术的主要应用包括：前台和后台工作、客户－服务器应用模式、任务异步处理、用户界面设计等。

在此列举多线程在字处理程序中发挥作用的例子。如果字处理程序是单线程的，那么输入、编辑和存盘三项任务串行完成。例如，在进行磁盘备份时，来自键盘和鼠标的命令和信息不会被处理，直至备份结束，这样用户会感觉系统性能很差。假设字处理程序被编写成含有 3 个线程，第一个线程在前台与用户交互，监控键盘和鼠标，一旦发现有变化便通知第二个线程；第二个线程在后台工作，得到通知时重新进行编辑处理；第三个线程在后台周期性地将文件写盘，以免因发生故障而丢失已处理的内容。由于 3 个线程处于同一个地址空间中，且共享同一个文件，因而能协调、高效地工作。

2.4.3 线程的实现

多线程的实现分为三类：用户级线程（User Level Thread, ULT），如 POSIX1003.4a 的

Pthreads、Java 线程库；内核级线程(Kernel Level Thread, KLT)，如 Windows 2003、OS/2 和 Mach C-thread；某些系统(如 Solaris UI-threads)提供混合方式，同时支持 ULT 和 KLT 两种线程。

1. 内核级线程

内核级线程是指线程的管理工作由内核完成，由内核所提供的线程 API 来使用线程，当任务提交给操作系统执行时，内核为其创建进程和一个基线程，线程在执行过程中可通过内核的创建线程原语来创建其他线程，应用程序的所有线程均在一个进程中获得支持。内核需要为进程及进程中的单个线程维护现场信息，所以，应在内核空间中建立和维护进程控制块 PCB 及线程控制块 TCB，内核的调度在线程的基础上进行。

内核级线程的主要优点是：首先，在多处理器上，内核能够同时调度同一进程中的多个线程并行执行；其次，若进程中的一个线程被阻塞，内核能够调度同一进程的其他线程占有处理器运行，也可以运行其他进程中的线程；最后，由于内核级线程只有很小的数据结构和堆栈，其切换速度比较快，内核自身也可用多线程技术实现，从而提高系统的执行效率。内核级线程的主要缺点是：线程在用户态运行，而线程的调度和管理在内核实现，在同一进程中，控制权从一个线程传送到另一个线程时需要用户态—核心态—用户态的模式切换，系统开销较大。

2. 用户级线程

用户级线程是指线程的管理由应用程序完成，在用户空间中实现，内核无须感知线程的存在。用户级多线程由用户空间中的线程库来完成，应用程序通过线程库进行设计，再与线程库连接、运行以实现多线程。线程库是由用户级线程管理的例行程序包，在这种情况下，线程库是线程运行的支撑环境。

用户级线程有许多优点：线程切换无须使用内核特权方式，所有线程管理的数据结构均在进程的用户空间中，管理线程切换的线程库也在用户空间中运行，这样可以节省模式切换的开销和内核的宝贵资源；允许进程按照应用的特定需要选择调度算法，且线程库的线程调度算法与操作系统的低级调度算法无关；能够运行在任何操作系统上，内核无须做任何改变。用户级线程的明显缺点是：由于大多数系统调用是阻塞型的，因此，一个用户级线程的阻塞将引起整个进程的阻塞；用户级线程不能利用多重处理的优点，进程由内核分配到 CPU 上，仅有一个用户级线程可以执行。因此，不可能得益于多线程的并发执行。

3. 混合式线程

某些操作系统既支持用户级线程，又支持内核级线程，Solaris 便是一个例子。线程的实现分为两个层次：用户层和核心层。用户层线程在用户线程库中实现；核心层线程在操作系统内核中实现，处于两个层次的线程分别称为用户级线程和内核级线程。在混合式线程系统中，内核必须支持内核级多线程的建立、调度和管理，同时也允许应用程序建立、调度和管理用户级线程。

在混合式线程中，一个应用程序中的多个用户级线程能分配和对应于一个或多个内核级线程，内核级线程可同时在多处理器上并行执行，且在阻塞一个用户级线程时，内核可以调度另一个线程执行，使得宏观上和微观上都具有很好的并行性。例如，窗口系统是典型的逻辑并行性程

度较高的应用,用一组用户级线程来表达多个窗口,用一个内核级线程来支持这一组用户级线程,这样,屏幕上同时出现多个窗口(对应于多个用户级线程),窗口之间的切换很频繁,但某一时刻只有一个窗口(对应于内核级线程)处于活跃状态,系统开销小,窗口系统执行效率高。又如,大规模并行计算是对物理并行性要求较高的应用,可以把数组按行或列划分给不同的用户级线程处理,同时,让每个用户级线程与一个内核级线程绑定,每个内核级线程占用一个CPU并行执行,减少了线程切换的次数,通过并行计算提高系统效率。

2.5 Linux 进程与线程

Linux 是多用户多任务操作系统,支持多道程序设计、分时处理和软实时处理,从 V2.2.x 内核开始,支持对称式多处理器和多线程。Linux 认为线程就是共享地址空间及其他资源的进程,因而并未单独为线程定义数据结构,有一套在用户模式下运行的线程库 pthread,但每个线程都拥有属于自己的唯一 task_struct。Linux 还支持内核线程,这类线程周期性地被内核唤醒和调用,没有虚拟地址空间,运行在核心态,工作在内核空间中,可以被调度和抢占,主要用于实现系统后台操作,如页面对换、刷新磁盘缓存、网络连接等系统工作。

在 Linux 系统启动时,最先产生 idle 进程(其 pid 为 0),此进程会创建一个内核线程来执行初始化工作,结果使处理器的运行模式由核心态切换至用户态,内核线程演变成用户进程 init(其 pid 为 1),一切用户进程都是它的后代进程。在创建每个进程时,要为其建立新的进程描述符结构体:

```
struct task_struct {
    volatile long state;           //进程状态
    unsigned long flags;          //进程标志
    pid_t pid;                    //进程 ID
    struct thread_info *thread_info; //指向进程的 thread_info 指针
    int load_weight;              //平衡负载所用的权重
    int prio, static_prio, normal_prio; //动态优先级、静态优先级和正常优先级
    struct list_head run_list;     //连接 prio_array 中的元素
    struct prio_array *array;      //当前 CPU 的就绪队列
    unsigned long policy, sleep_avg, rt_priority; //调度策略、平均等待时间、实时进程优先级
    unsigned int time_slice, first_time_slice; //时间片余额;是否是第一次分得时间片的标志
    struct list_head tasks;        //任务队列
    struct mm_struct *mm, *active_mm; //进程虚存信息;内核线程借用的地址空间
    struct task_struct *parent;     //父进程
```

```

    struct list_head children, sibling;           //子进程链表、兄弟进程链表
    struct pid_link pids[PIDTYPE_MAX];           //pid 的哈希表链
    struct list_head thread_group;                //线程队列链
    struct timer_list real_timer;                 //时间信息
    unsigned long it_real_value, it_prof_value, it_virt_value, utime, stime;
    uid_t uid, euid, suid, fsuid;                //各种用户的标识符
    gid_t gid, egid, sgid, fsgid;                //各种组的标识符
    int link_count, total_link_count;            //各种文件链接数
    struct fs_struct *fs;                        //文件系统信息
    struct files_struct *files;                  //打开文件表指针
    struct signal_struct *signal;                //信号处理信息
    struct sighand_struct *sighand;              //信号处理函数指针
    sigset_t blocked, real_blocked;
    struct sigpending pending;
    ...
};


```

进程描述符中包含:进程标识、状态、标志、链接、调度、文件、虚存、信号等信息。在 V2.2 之前的版本中,进程重调度标志 need_resched 是全局变量;在 V2.2 到 V2.4 版中它被移入进程 task_struct 中;而在 V2.6 版中,它被移至 thread_info 结构体中,用特别的标志变量中的一位 TIF_NEED_RESCHED 来表示。

Linux 内核中保留一个大小为 NR_TASKS 的全局数组 *task[NR_TASKS],默认长度为 512 bit,数组元素是指向 task_struct 的指针,在创建新进程时,从系统空间中分配一个 task_struct 结构,并将其首地址填入 *task 数组。进程核心栈和 task_struct 结构间存在紧密的联系,V2.4 中,内核为进程建立 task_struct 时,一次分配两个连续页框(8 KB),其底部约 1 KB 空间用做 task_struct 结构,上端约 7 KB 为进程栈心栈。在 V2.6 中,task_struct 结构变为由 slab 分配器动态分配,如图 2.18(a)所示,进程的运行环境信息 thread_info 结构在栈心栈尾端分配,此结构中的 task 成员是指向此进程的 task_struct 的指针。current 指针通过堆栈指针 esp 来计算,去掉 esp 末尾 13 位得到 thread_info 的地址,再通过 thread_info->task_struct 得到 task_struct 的地址。图 2.18(h)为 Linux 进程的虚存映像。

Linux 进程的运行环境信息 thread_info 是 V2.6 新创建的一个结构,除进程描述符 task_struct 指针外,还包含当前 CPU 号、底层标志、运行域、线程同步标志、内核抢占计数器等。task_struct 约占 1.7 KB,基本内容与先前版本相同,为了支持新的 O(1) 调度算法,增加用于调度的新成员有:动态优先级 prio、静态优先级 static_prio、正常优先级 normal_prio、优先级数组 prio_array、进程时间片剩余值 time_slice、进程平均等待时间 sleep_avg、负载平衡权重 load_weight 等,延用的有实时优先级 rt_priority、调度策略 policy 等。

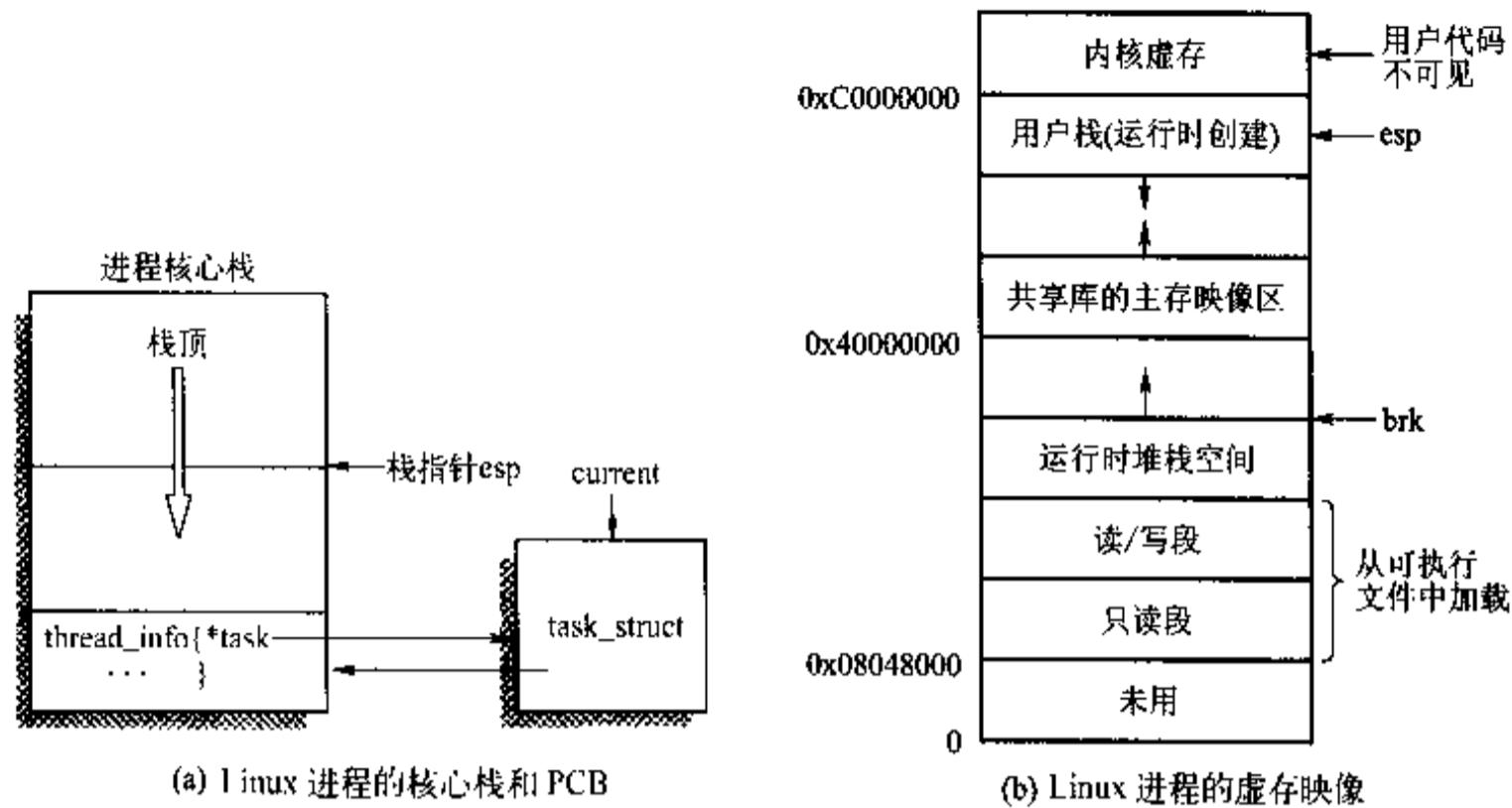


图 2.18 Linux 进程

处于相同状态的进程通过前后向队列指针连接成双向循环链表,链表的第一个元素是 `init_task` 描述符,又称 0 号进程,它是所有进程的祖先。内核还会建立运行队列链表及多个等待队列链表,宏 `current` 是当前正在运行的进程的指针,在对称式多处理器系统中则指向 CPU 组中正在被调度的 CPU 上的当前进程。

Linux 进程的状态 `state` 共有以下 6 种。

(1) TASK_RUNNING

可运行状态:进程正在运行(已获得 CPU)或准备运行(等待获得 CPU),由 `run_list()` 把所有运行态进程链接起来。V2.6 内核改为每个 CPU 单独维护一个进程运行队列。

(2) TASK_INTERRUPTIBLE

可中断等待状态:进程排入等待队列中,一旦资源可用时进程就被唤醒,也可由其他进程通过信号或中断来唤醒。

(3) TASK_UNINTERRUPTIBLE

不可中断等待状态:进程排入等待队列中,一旦资源可用时进程就被唤醒,但是不可由信号或中断来唤醒。此状态通常意味着进程在等待某些硬件条件,如进程打开设备文件时会处于此状态。

(4) TASK_ZOMBIE

僵死状态:进程运行完成,已释放所占用的大部分资源,尚未释放 `task_struct` 结构体。

(5) TASK_STOPPED

暂停状态:此状态用于调试,因收到信号而暂停执行,也可能受到其他进程的跟踪和调用而暂时让出处理器。

(6) TASK_SWAPPING

页面被对换出主存的进程所处的状态。

2.6 Windows 2003 进程与线程

1. 进程与线程

Windows 2003 进程是资源的容器,容纳所分配到的各种资源如主存、已打开文件等;线程是可以被内核调度的执行实体,它可以被中断,使 CPU 转向另一线程执行。

进程和线程均通过对象来实现,进程包含一个或多个线程。Win32 进程由三个部分组成:虚拟地址空间描述符(Virtual Address Descriptor, VAD),描述进程地址空间各部分的属性,用于虚存管理;线程块列表,包含进程中所有线程的相关信息,供调度器控制 CPU 的分配和回收;对象句柄列表,当进程创建或打开对象时,将得到一个代表此对象的句柄,用于对象的访问,此列表维护进程正在访问的所有对象。进程及其所控制和使用的资源之间的关系如图 2.19 所示。

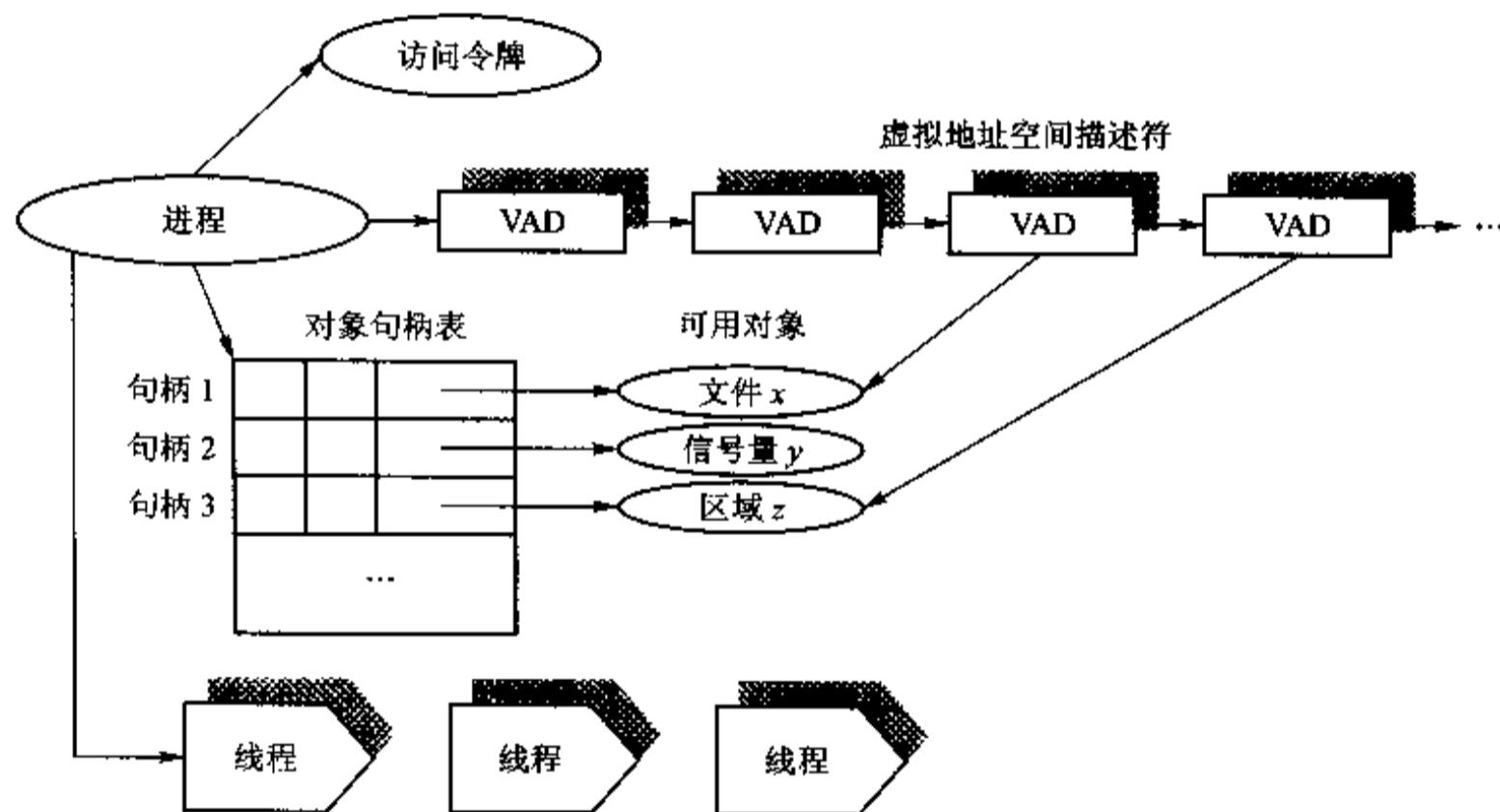


图 2.19 进程及其所控制和使用的资源

当用户首次注册时,Windows 将为用户创建一个包括用户安全 ID 的访问令牌,每个用户所创建的进程或代表用户运行的进程均持有此访问令牌的副本,内核用它来验证用户是否可以访问受保护的对象,或在受保护的对象上执行限定的功能。

为了支持 Win32、OS/2、POSIX 等多种子系统,Windows 子系统的进程之间不存在任何关系,各环境子系统分别建立、维护和表达自己的进程关系。Win32 子系统是 Windows 的主子系统,实现基本的进程管理功能;POSIX 和 OS/2 利用 Win32 子系统的功能来实现自身的功能。因此,在 Windows 中,与一个环境子系统中的应用进程相关的进程控制块信息会分布在本运行环

境子系统、Win32 子系统和系统内核中。

2. 对象及对象管理器

Windows 是一个基于对象(object-based)的操作系统,用对象来表示所有系统资源,可以认为对象类就是资源类。定义两大类对象如下:

(1) 执行体对象

由执行体组件所实现的对象,用来实现各种外部功能。用户态程序可以访问的执行体对象有进程、线程、区域、文件、事件、队列、文件映射、文件端口、互斥量、信号量、计时器、对象目录、符号连接和访问令牌等。

(2) 内核对象

由内核所实现的原始对象,内核对象包括内核过程对象、异步过程调用对象、延迟过程调用对象、中断对象、电源通知对象、电源状态对象、调度程序对象等。内核对象对用户态代码是不可见的,仅在执行体内部创建和使用,许多执行体对象包含一个或多个内核对象,而内核对象提供仅能由内核来完成的基本功能。

Windows 对象由对象头和对象体所组成。对象头由对象管理器控制,各执行体组件控制自己创建的对象类型的对象体。每个对象头都指向打开此对象的进程列表,同时还有一个称为类型对象的特殊对象,其所包含的信息对每个对象实例是公用的。标准的对象头属性有对象名、对象目录、安全描述符、配额账、打开句柄计数器、打开句柄数据库、永久/暂时状态、内核/用户模式和对象类型指针等。

对象体的格式和内容随对象类型的不同而不同,对象类型的属性有对象类型名、存取类型、同步能力、分页/不分页、一个或多个方法。对象管理器所提供的通用服务程序有关闭句柄、复制句柄、对象查询、对象安全性查询、对象安全性设置、等待单个对象或多个对象等。

Windows 通过对象管理器为执行体中的各种内部服务提供一致的、安全的访问手段,它是一个用于创建、删除、保护和跟踪对象的执行体组件,提供使用系统资源的一致性机制。对象管理器在接收创建对象的系统服务之后,将完成以下工作:为对象分配主存空间;为对象设置安全描述体,以确定使用对象者,及允许访问对象者执行的操作;创建和维护对象目录表;创建对象句柄并将其返回给创建者。

当应用程序使用 CreateProcess() 和 CreateThread() 创建其他进程和线程时,它将名称传递给对象管理器,返回一个句柄和系统范围的标识,一旦得到一个句柄,就可以在其他请求中将句柄作为参数传递给操作系统。句柄是一个 32 位二进制数,用户线程以此来引用执行体对象,如图 2.20 所示,可通过将表的索引值赋予句柄来完成,然后,再将它作为内核进程句柄表索引,使得应用程序可以引用执行体对象而不必知道它的地址。

对象头包含句柄引用计数,用来记录对象被引用的次数,用户空间和内核空间的打开操作都会使句柄引用次数有所增加,关闭操作使得句柄引用次数减 1。如果句柄引用次数为 0,则此对象未发生任何引用,系统会释放对象。指向对象比使用对象名要快,因为对象管理器可通过句柄

直接找到对象，所有用户态线程只有获得对象句柄才能使用对象。句柄作为系统资源的间接指针使用，对象管理器具有创建和定位句柄以引用对象的专用权限。

当进程创建对象或打开已存在对象的句柄时，必须指定所期望的一组访问权限，这时对象管理器将调用安全引用监视程序，并把进程的一组访问权限发送给它。安全引用监视程序将检查对象的安全描述体是否允许进程所请求的访问类型，如果允许则返回一组授权的访问权限，对象管理器会把它们存储在对象句柄中。此后，当进程的线程使用句柄时，对象管理器可以快速检查句柄中所存储的授权访问权限集是否符合由线程调用的对象服务所隐含的用法。

3. 进程对象

执行体的进程管理器所实现的功能有：创建和撤销进程及线程；监视资源的分配状况；提供同步原语；控制进程和线程的状态变化，等等。进程是作为对象来管理的，具有对象的通用结构。每个进程对象由属性和所封装的若干可执行服务来定义，当接收到消息时，进程就执行一个服务，只能通过向提供服务的进程对象传递消息来调用服务。进程对象的属性包括进程标识、访问令牌、进程基本优先级和默认的亲合处理器集合等。进程是对应于存储器、打开文件表等资源的应用程序实体；线程是执行工作时的调度单位，并且可以被中断，这样处理器可被其他线程所占用。

在创建进程时，进程描述表分两部分实现，如图 2.21 所示。内核中的部分称为 KPROCESS，涉及对象管理、中断处理和线程调度；执行体中的部分称为 EPROCESS，它处理进程的其他方面，如管理地址空间的域、协调线程执行的域、跟踪进程分配资源的域及保护和共享任务资源的域。执行体进程块 EPROCESS 描述进程的基本属性。

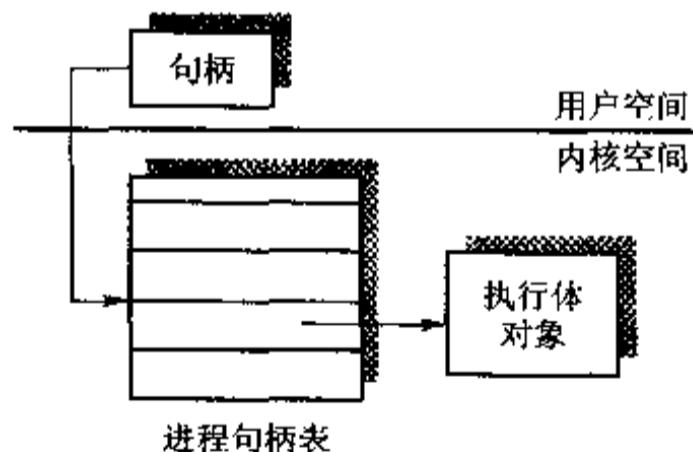


图 2.20 句柄和进程句柄表

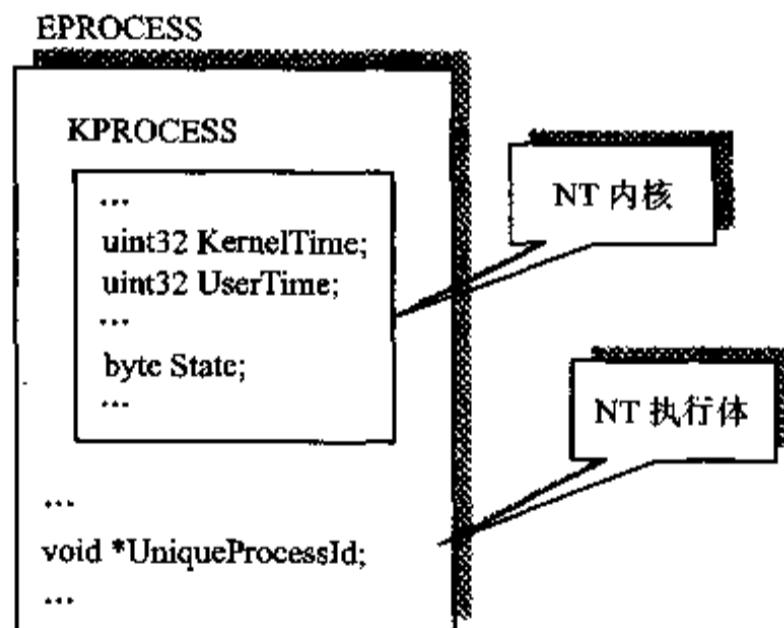


图 2.21 进程描述表的结构

(1) 内核进程块 KPROCESS：是公共的调度程序对象头，包含线程调度的基本优先级、默认时间片、进程状态和自旋锁、进程所在的处理器簇、进程总的核心态运行时间和用户态运行时间。

(2) 进程标识符:进程的唯一标识符、父进程标识符、进程所在的窗口位置。

(3) 配额限制:限制非页交换区、页交换区和页面文件的使用,限制进程所能使用的 CPU 时间。

(4) 主存管理信息:一系列虚拟地址空间描述符数据结构,描述进程虚拟地址空间的状态。

(5) 工作集信息:是指向工作集列表的指针,包括当前的、峰值的、最小的和最大的工作集尺寸,上次裁剪时间,页错误计数,主存优先级,对换标志,页错误历史记录,等等。

(6) 虚拟主存信息:包括当前值和峰值,页文件的使用,用于进程页面目录的硬件页表人口地址,等等。

(7) 异常/调试端口:当进程的一个线程发生异常时,这是进程管理程序发送消息的内部通信通道。

(8) 访问令牌:负责协调所有必需的安全信息,安全子系统通过访问令牌来确定用户的访问权限。

(9) 对象句柄表:进程对象句柄表地址,当进程创建或打开对象时,就会得到代表此对象的句柄,通过句柄可以引用对象。

(10) 进程环境块(Process Environment Block,PEB):存放于进程的用户态地址空间中,它包含映像信息:操作系统版本号、映像版本号、模块列表和映像进程亲和性掩码等。还包括线程所用堆栈的数量和大小、进程堆栈指针以及下一个进程块的链接指针。

Win32 子系统所提供的进程控制类系统调用主要有 CreateProcess、OpenProcess、ExitProcess、TerminateProcess、SetPriorityClass 和 GetPriorityClass。

当应用程序调用 CreateProcess() 函数时,将创建一个 Win32 进程,创建过程在操作系统的 3 个部分中分阶段地完成,它们是:Win32 客户端的 KERNEL32.DLL、Windows 执行体和 Win32 子系统进程 CSRSS。具体执行步骤如下:打开将在进程中执行的 .exe 映像文件;创建 Windows 执行体进程对象,申请并初始化 EPROCESS,创建并初始化 KPROCESS 和进程环境块,向进程分配地址空间;创建主线程(堆栈、描述表、执行体线程对象);向 Win32 子系统通知创建进程和线程的句柄,以便初始化新的进程和线程;在新进程和线程描述表中;完成地址空间的初始化,加载所需的动态链接库;启动初始线程执行。

4. 线程对象

Windows 线程是内核级线程,是 CPU 调度的独立单位。每个线程都由一个线程描述表来表示,在进程描述表 EPROCESS 的 KPROCESS 中包含 struct LIST_ENTRY ThreadListHead 域,它指向一组线程描述表,称为 ETHREAD 结构。ETHREAD 把 KTHREAD 结构作为它的一个域,类似于进程描述表,由执行体管理 ETHREAD 结构的内容,包括创建时间、相关进程标识和人口地址等;由内核管理 KTHREAD 结构的内容,包括用户态时间、核心态时间、栈信息和调度信息等。如图 2.22 所示是 Windows 2003 线程描述表的结构。

执行体线程块 ETHREAD 描述线程的基本属性:

(1) 内核线程块 KTHREAD:是公共的调度程序对象头,包含核心栈的栈指针及其大小、指

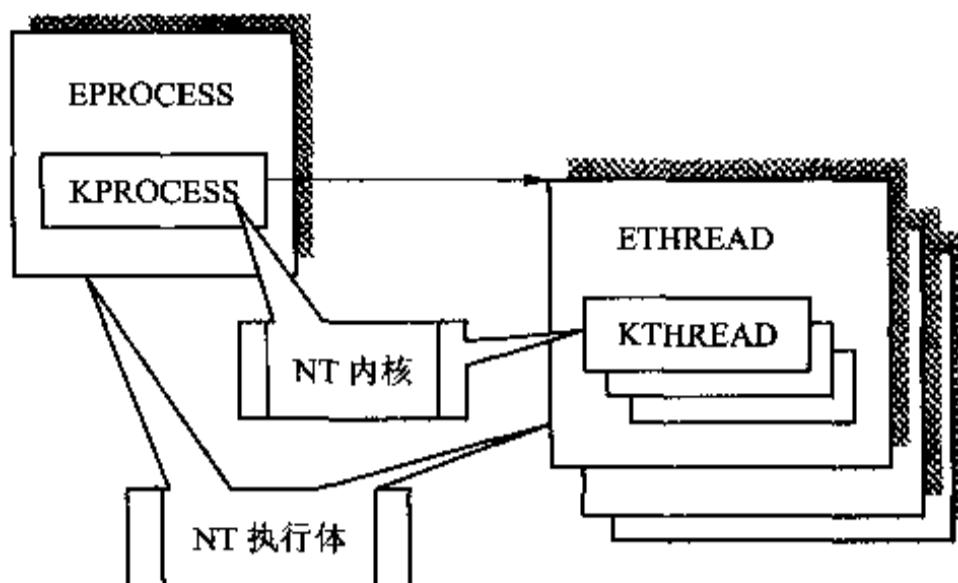


图 2.22 Windows 2003 线程描述表的结构

向系统调用服务表 USER 和 GDI 的指针、与调度有关的信息(基本的和动态的优先级、时间片、亲和性掩码、首选处理器、当前状态、挂起计数和线程的用户态和核心态时间总和、等待信息等)、与本线程有关的 APC 列表、线程环境块指针(存储用于映像加载程序和 Win32 的动态链接库描述信息,如线程 ID 和线程启动程序的地址)。

- (2) 进程标识和指向线程所属进程 EPROCESS 的指针。
- (3) 访问令牌和线程类别(客户线程或服务器线程)。
- (4) LPC 消息信息:线程正在等待的消息 ID 和消息地址。
- (5) I/O 请求信息:指向挂起的 I/O 请求包列表的指针。

Win32 子系统所提供的线程控制类系统调用主要有 CreateThread、OpenThread、ExitThread、TerminateThread、SuspendThread、ResumeThread 和 SetThreadPriority。

应用程序调用 CreateThread() 函数创建一个 Win32 线程的具体步骤如下: 在进程的地址空间中为线程创建用户态堆栈; 初始化线程描述表; 调用 NtCreateThread() 创建执行体线程对象, 内容包括: 增加进程中的线程计数; 创建并初始化执行体线程块; 生成新线程 ID; 从非页交换区分配线程的内核堆栈; 设置线程环境块 (Thread Environment Block, TEB); 设置线程的起始地址和用户指定的 Win32 起始地址; 设置 KTHREAD 块; 设置指向进程访问令牌的指针和创建时间; 通知 Win32 子系统已经创建新线程, 以便 Win32 子系统设置新的进程和线程; 设置新线程为准备态, 将其句柄和 ID 返回到调用进程; 调用 ResumeThread, 激活线程并调度执行。

5. 进程和线程的状态

在 Windows 操作系统中, 进程被简单地划分为可运行态和不可运行态。线程是处理器调度的基本单位, 如图 2.23 所示, 它可以处于 7 种状态之一。

- (1) 就绪态(ready): 线程已经获得除 CPU 以外的所有资源, 处于等待调度执行的状态。内核的调度程序维护所有就绪线程队列, 并按照优先级次序进行调度。
- (2) 准备态(stanby): 已选中在特定处理器上运行的下一个线程, 此线程处于准备态并等待

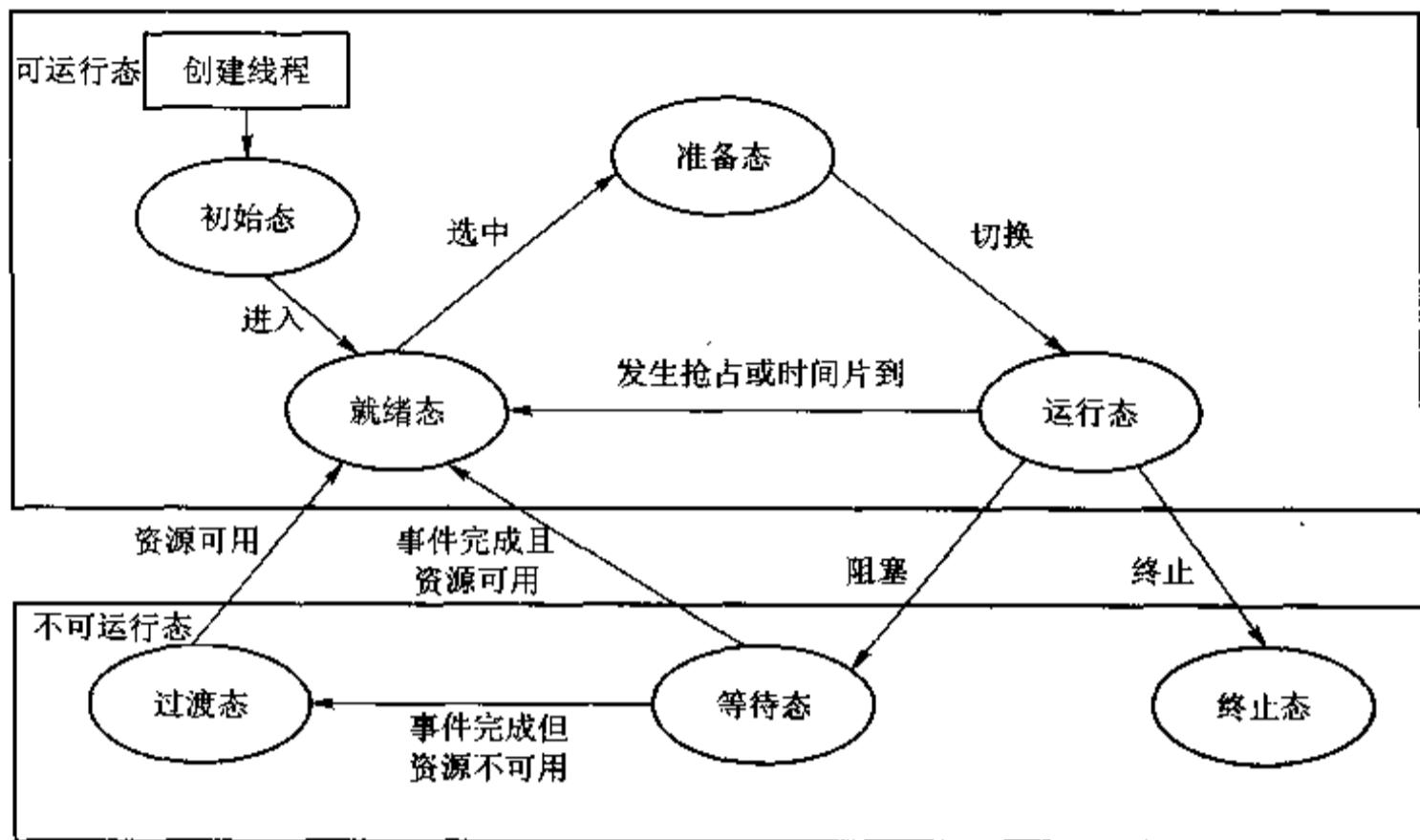


图 2.23 Windows 线程的状态

线程描述表切换。如果准备态线程的优先级足够高,可以从运行线程那里抢占处理器,否则将等待直至运行线程进入等待态或时间片用完。系统中的每个处理器上只能有一个处于准备态的线程。

(3) 运行态(running):内核执行线程切换,准备态线程进入运行态并开始执行,直至它被抢占、或用完时间片、或阻塞、或终止时为止。在前两种情况下,线程转换为就绪态。

(4) 等待态(waiting):线程进入等待态的原因有:出现阻塞事件;等待同步信号;环境子系统要求线程挂起。当等待事件条件满足且所有资源可用时,线程就转换为就绪态。

(5) 过渡态(transition):线程完成等待后准备运行,但此时资源不可用,线程就转换为过渡态。例如,线程的内核堆栈被调出主存,当资源可用时(内核堆栈被调入主存),过渡态线程进入就绪态。

(6) 终止态(terminated):线程可以被自己或其他线程终止,此时线程进入终止态。一旦结束工作完成后,线程就可以从系统中移去。如果执行体有一个指向线程对象的指针,可将处于终止态的线程对象重新初始化并再次使用。

(7) 初始态(initial):线程在创建过程中处于初始态,创建完成后,此线程被放入就绪队列。

2.7 处理器调度

在计算机系统中,可能有很多批处理作业同时存放在磁盘的后备作业队列中,或者有很多终端与主机相连,交互型作业不断地进入系统,这样主存和处理器等资源便供不应求。按照何种原

则挑选批处理作业进入主存运行、能否继续接纳分时用户、如何在进程之间分配处理器资源，无疑是操作系统进行资源管理所要面对的重要问题，由处理器调度完成涉及处理器调度和资源分配的工作。

2.7.1 处理器调度的层次

处理器调度按照层次可分为三级：高级调度、中级调度和低级调度。用户作业从进入系统成为后备作业开始，直到运行结束退出系统为止，均需经历不同级别的调度。

1. 高级调度

高级调度（high level scheduling）又称作业调度、长程调度，在多道批处理操作系统中，从输入系统的一批作业中按照预定的调度策略挑选若干作业进入主存，为其分配所需资源，并创建作业的相应用户进程后便完成启动阶段的高级调度任务，已经为进程做好运行前的准备工作，等待进程调度挑选进程运行，在作业完成后还要做结束阶段的善后工作。

高级调度将控制多道程序的道数，被选择进入主存的作业越多，每个作业所获得的 CPU 时间就越少，为了向用户提供满意的服务，有时需要限制多道程序的道数。每当有作业执行完毕并撤离时，作业调度会选择一个或多个作业补充进入主存。此外，如果 CPU 的空闲时间超过一定的阈值，系统也会引出作业调度选择后备作业。

2. 中级调度

中级调度（medium level scheduling）又称平衡调度、中程调度，根据主存资源决定主存中所能容纳的进程数目，并根据进程的当前状态来决定辅助存储器和主存中的进程的对换。当主存资源紧缺时，会把暂时不能运行的进程换出主存，此时这个进程处于“挂起”状态，不参与低级调度；当进程具备运行条件且主存资源有空闲时，再将进程重新调回主存工作，起到短期均衡系统负载的作用，充分提高主存的利用率和系统吞吐率。

3. 低级调度

低级调度（low level scheduling）又称进程调度/线程调度、短程调度，其主要功能是：根据某种原则决定就绪队列中的哪个进程/内核级线程获得处理器，并将处理器出让给它使用。低级调度是操作系统最为核心的部分，执行十分频繁，其调度策略的优劣将直接影响整个系统的性能，因而，这部分代码要求精心设计，并常驻主存。

在上述三级调度中，低级调度是各类操作系统必备的功能，在纯粹的分时操作系统或实时操作系统中，通常不需要高级调度；一般的操作系统都配置高级调度和低级调度；而功能完善的操作系统为了提高主存的利用率和作业吞吐量，引进中级调度。所以，从处理器调度的层次来说，可分成三级调度模型、两级调度模型和一级调度模型。

如图 2.24 所示是处理器三级调度模型，由于引入中级调度，相应地为进程增加“挂起”状态，某些进程需要挂起时，将其对换到磁盘镜像区中，释放其所占有的某些资源，暂时不参与低级调度。这些进程对换出去后，主存就绪态进程转换为辅存就绪态进程，主存等待态进程转换为辅存等待态进程。反之，在适当时刻，由中级调度把进程对换回主存，使其辅存就绪态转换为主存就

绪态,辅存等待态转换为主存等待态。

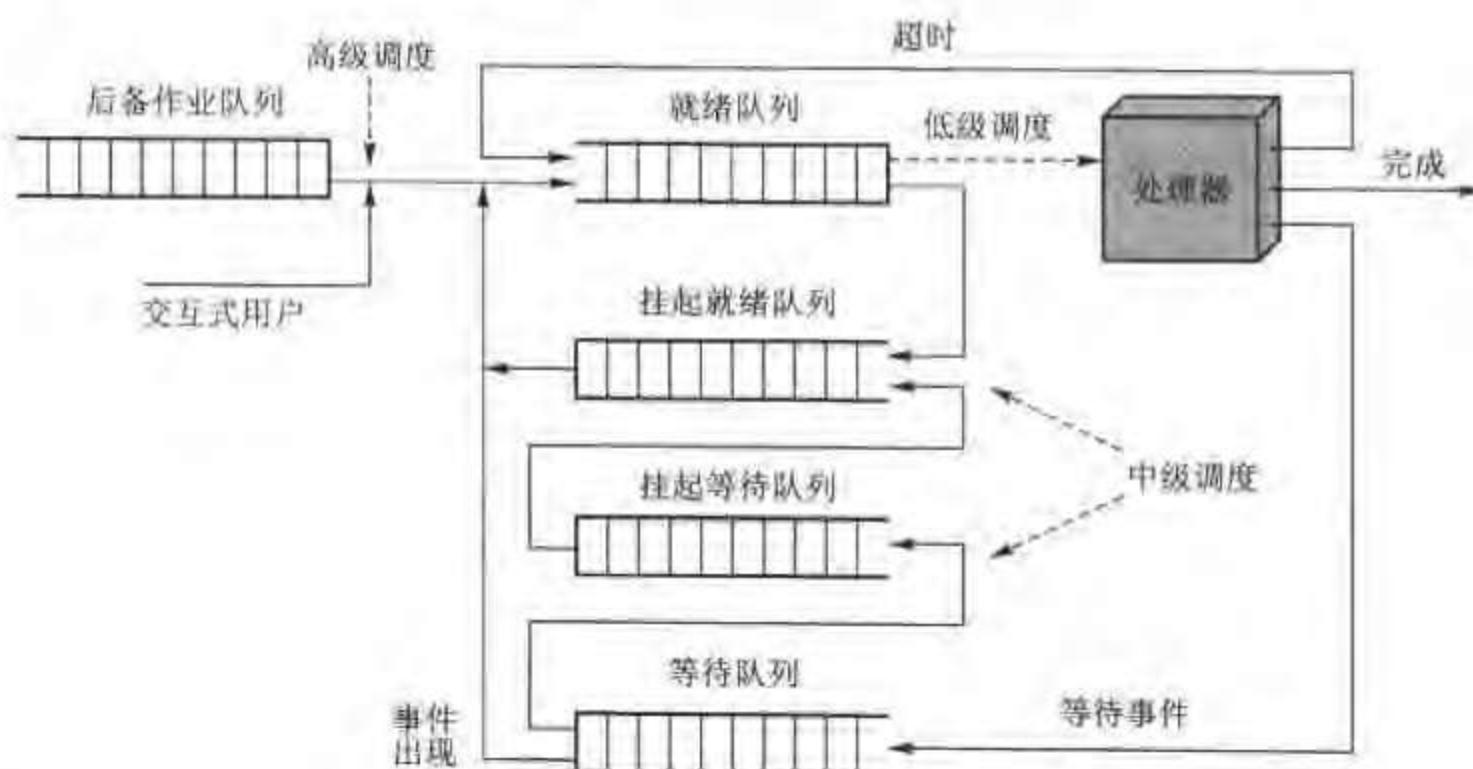


图 2.24 处理器三级调度模型

如图 2.25 所示是具有高级调度和低级调度的处理器两级调度模型。进入计算机系统的作业经过两级调度之后才能占用处理器,第一级是高级调度,选择所需资源已被满足的部分后备作业进入主存,同时生成对应于作业的进程,使其获得参与竞争处理器的资格;第二级是低级调度,按照规定算法动态地调度进程占用处理器运行,图 2.25 中仅画出一个等待队列,也可以设置多个等待队列,让等待不同事件的进程排入不同的等待队列。为了充分利用处理器,可以把多个作业同时装入主存,这样就有多个用户进程,这些进程都要竞争处理器,高级调度和低级调度相配合便能实现多道程序的同时执行。

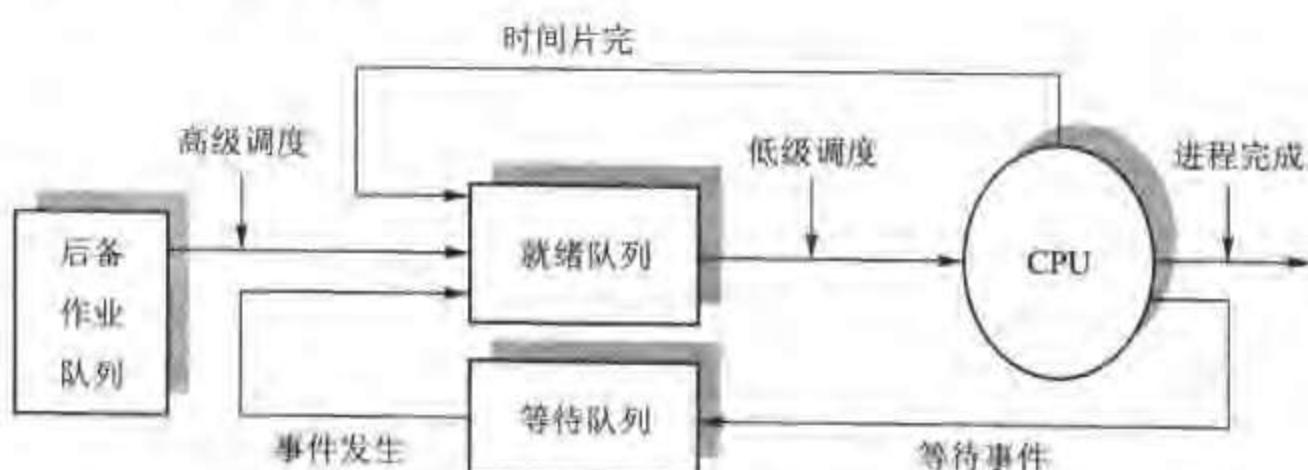


图 2.25 处理器两级调度模型

纯粹的分时操作系统可以仅设置低级调度,即一级调度模型,但是系统应提供以下功能:决定是否接受终端用户的连接,决定交互型作业能否被接纳并创建作业进程,系统通常会接纳所有授权用户,直到饱和为止;一个新建态的进程应能立即加入就绪队列参与调度。系统设置就绪队

列和一个或多个等待队列,常用的调度策略是时间片轮转法,进程运行时可能产生以下情况:此进程在时间片内运行完成,释放处理器;此进程在时间片内未运行完成,内核将其排入就绪队列尾,等待下一个时间片;此进程在执行期间等待某事件发生,内核将进程移入相应的等待队列。

2.7.2 选择调度算法的原则

操作系统的调度算法选择将受到很多因素的影响,评价调度算法的优劣和性能也是十分复杂的事情,不同类型操作系统的调度算法往往不一样,由于应用进程的特性、对响应时间和系统资源的要求不尽相同,使得调度算法的设计比较复杂。选择调度算法的基本原则是计算机系统的性能要高,下面列出的前三条是面向系统的性能指标,后两条是面向用户的性能指标。

(1) 资源利用率

让 CPU 和各种资源尽可能并行工作,使得资源的利用率尽可能要高。

$$\text{CPU 利用率} = \text{CPU 有效工作时间} / \text{CPU 总的运行时间}$$

$$\text{CPU 总的运行时间} = \text{CPU 有效工作时间} + \text{CPU 空闲等待时间}$$

在一定的 I/O 操作等待时间的比率下,运行程序的道数越多,CPU 空闲时间所占的百分比越低。

(2) 吞吐率

单位时间内 CPU 处理作业的个数。显然,所处理的长作业多则吞吐率低,所处理的短作业多则吞吐率高。这是批处理系统衡量调度性能的重要指标之一。

(3) 公平性

确保每个进程都能获得合理的 CPU 份额和其他资源份额,不会出现饥饿现象。

(4) 响应时间

从交互式进程提交一个请求(命令)至得到响应之间的时间间隔称为响应时间,细分起来包括:所输入的请求命令传送到 CPU 的时间、CPU 处理这一请求命令的时间和处理所形成的响应回送到终端显示器的时间。使交互式用户的响应时间尽可能短,或尽快处理实时任务,这是分时系统和实时系统衡量调度性能的重要指标之一。

(5) 周转时间

批处理用户从向系统提交作业开始,到作业完成为止的时间间隔称为作业周转时间,细分起来包括:作业在后备队列中等待的时间、相应的进程进入主存之后在就绪队列中等待的时间、进程在 CPU 上执行的时间和等待事件发生(如等待数据传输)的时间。应使得作业的周转时间或平均作业周转时间尽可能短,这是批处理系统衡量调度性能的一项重要指标。

当然,这些目标本身就存在着矛盾之处,在设计操作系统时必须根据其类型的不同进行权衡,以达到较好的效果。

下面着重讨论批处理系统的调度性能指标。批处理系统的调度性能用作业周转时间和带权作业周转时间来衡量,此时间越短,则系统效率越高,作业的吞吐率越高。如果作业 i 提交给系

统的时刻是 t_s , 完成时刻是 t_f , 那么, 作业 i 的周转时间 t_i 为

$$t_i = t_f - t_s$$

实际上, 这是作业在系统中的等待时间和运行时间之和。从操作系统的角度来说, 为了提高系统性能, 要让若干用户的平均作业周转时间和平均带权作业周转时间最小。

$$\text{平均作业周转时间 } T = \left(\sum_{i=1}^n t_i \right) / n$$

如果作业 i 的周转时间为 t_i , 所需运行时间为 t_k , 则称 $w_i = t_i / t_k$ 为此作业的带权周转时间。因为 t_i 是作业等待时间和运行时间之和, 故带权周转时间总是大于 1 的。

$$\text{平均带权作业周转时间 } W = \left(\sum_{i=1}^n w_i \right) / n$$

通常, 平均作业周转时间用来衡量不同调度算法对同一作业流的调度性能, 平均带权作业周转时间用来衡量同一调度算法对不同作业流的调度性能, 这两个数值都是越小越好。

2.7.3 作业和进程的关系

作业管理是为了合理地组织工作流程和方便用户解决应用问题而在操作系统中提供的管理模块, 用于对作业进行组织、控制和管理, 要完成三项任务: 一是作业组织; 二是作业调度; 三是运行控制。

作业(job)是用户提交给操作系统计算的一个独立任务。每个作业必须经过若干相对独立且相互关联的顺序加工步骤才能得到结果, 其中, 每个加工步骤称为作业步(job step)。例如, 一个作业可分成“编译”、“链接”、“装入”和“运行”这 4 个作业步, 上一个作业步的输出往往是下一个作业步的输入。作业由用户组织, 作业步由用户指定, 作业从提交给系统直到运行结束获得结果, 需要经过不同的处理阶段, 当一个作业被作业调度选中进入主存并投入运行时, 操作系统将为此用户作业生成相应的用户进程完成其计算任务。若干批处理作业进入系统并依次存放在磁盘上, 在系统的控制下逐个取出执行便形成作业流; 交互型作业则通过命令管理提交并进入系统。进程是已提交完毕并选中运行的作业(程序)的执行实体, 也是为完成作业任务向系统申请和分配资源的基本单位, 它处于“运行”、“就绪”、“等待”等多个状态的变化之中, 在 CPU 上推进, 最终完成应用程序的任务。应用进程在执行的过程中可生成作业步子进程, 当然, 子进程还可以生成子进程, 这些进程并发执行, 相互协作高效地完成用户的作业任务。在多道程序设计环境中, 由于多个作业可以同时投入运行, 因此, 在并发系统中, 某一时刻并发执行的进程相当多, 操作系统要负责众多进程的协调和管理。在一个进程中还可以创建多个线程, 由线程并发执行来完成作业任务。

综上所述, 可以看出作业和进程之间的主要关系: 作业是任务实体, 进程是完成任务的执行实体; 没有作业任务, 进程无事可做; 没有进程, 作业任务无法完成。作业的概念更多地用于批处理操作系统中, 而进程则用于各种多道程序设计系统。

2.7.4 作业的管理与调度

根据作业处理方式的不同,作业可分为批处理作业和交互型作业。

1. 批处理作业的组织和管理

(1) 批处理作业的输入

批处理作业采用脱机控制方式,作业由程序、数据和作业说明书组成。程序和数据用于解决用户的应用问题或完成业务处理;作业说明书体现用户对作业的控制意图,汇集作业的基本描述、控制描述和资源要求描述,作业说明书与程序和数据一起提交给系统管理员,SPOOLing 系统成批接收并控制作业的输入,将其存放在输入井,然后,在系统的管理和控制下被调度和执行。

(2) 批处理作业的建立

为了有效地管理作业,必须像进程管理那样为进入系统的每个作业建立一个作业控制块(Job Control Block,JCB),所有 JCB 组成作业表。JCB 是在批处理作业进入系统时由 SPOOLing 和作业管理模块建立的,它是批处理作业存在于系统的标志,作业撤离时其 JCB 也被撤销。作业控制块的内容主要从用户的作业说明书中获得,包括:作业情况(用户名、作业名、语言名等),资源需求(CPU 运行估计时间、截止时间、主存容量、设备类型及台数、文件数和输出量、函数库/实用程序等),资源使用情况(进入系统时间、开始运行时间、已运行时间、主存地址、外设编号等),作业控制(优先级、联机/脱机、操作顺序、出错处理等),作业类型(CPU 繁忙型、I/O 繁忙型、均衡型等)信息。作业控制块是一个共享数据结构,SPOOLing 预输入程序、进程管理程序、缓输出程序及作业调度程序、作业控制程序等都需要访问它,把已输入并建好作业控制块的作业排入后备作业队列,以便等待作业调度。

(3) 批处理作业的调度

批处理作业调度是指按照某种算法从后备作业队列中选择部分作业进入主存运行,当作业运行结束时做好善后工作。作业调度程序需要完成以下几项任务。

① 选择作业

由系统规定作业调度算法,按此算法从后备作业队列中挑选作业。作业控制块中列出作业所需要的资源,由于资源要求被满足的作业可能有多个,需要根据资源状况和多道程序的道数决定选中哪些作业。各种作业调度算法将在下一节介绍。

② 分配资源

作业调度程序与主存管理和设备管理程序进行通信,为作业分配所需要的资源。

③ 创建进程

每当中选作业且将其装入主存时,系统就为此作业创建应用进程,生成 PCB 及各种进程实体,这些进程将在低级调度的控制下占用处理器运行。可以把多个作业同时装入主存并启动运行,这些进程要竞争处理器,也就实现了多道程序设计。所以,进入计算机系统的批处理作业至少要经过两级调度之后才能占用处理器。第一级是作业调度,作业通过竞争进入主存,同时生成相应的进程;第二级是低级调度,进程竞争处理器资源。作业调度与低级调度相配合能够实现多

道作业同时执行。作业调度与进程调度的关系及作业和进程的状态转换如图 2.26 所示。

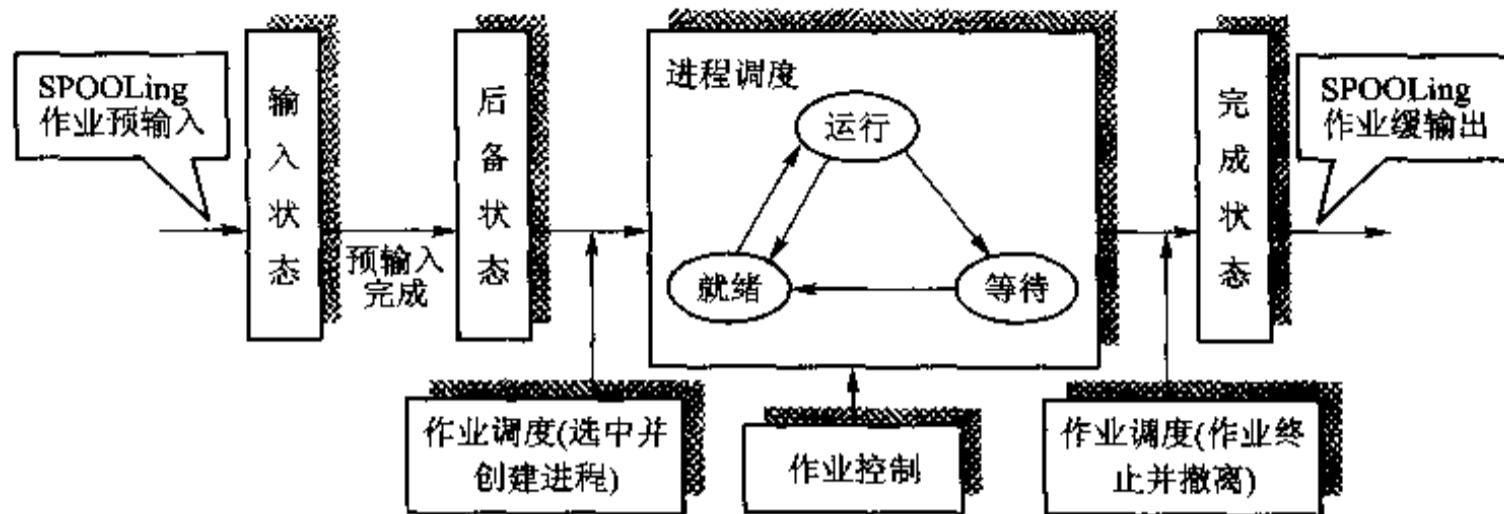


图 2.26 作业调度与进程调度的关系及作业和进程的状态转换

④ 作业控制

作业按照 JCL 书写的作业说明书运行, 作业启动、作业步转接、程序调入、数据 I/O、异常处理均由作业控制块和 SPOOLing 管理程序完成。

⑤ 后续处理

作业正常结束或出错终止时, 作业调度程序要做好作业撤离和善后工作, 如打印输出信息、回收各种资源、撤销作业控制块等。同时, 启动作业调度程序选择新的作业进入主存运行。

2. 交互型作业的组织和管理

操作系统在启动时, 为连接至系统的每个终端设备创建一个终端进程, 此进程执行命令解释程序, 其处理过程如下: 输出命令提示符, 从终端设备读入命令, 等待键盘中断的到来, 每当用户输入一条命令(暂存于命令缓冲区)并回车换行时, 申请键盘中断; CPU 响应后, 将控制权交给命令解释程序, 读入命令缓冲区的内容, 分析命令并接收参数; 若为简单命令则立即转向命令处理代码执行; 否则创建子进程, 传递参数, 加载和执行命令处理文件; 命令处理结束后, 再次输出命令提示符, 等待下一条命令。若当前的终端命令是一条后台命令, 则处理子进程可以和下一条终端命令的处理子进程并行执行, 各子进程在运行过程中可以根据需要创建子进程, 当终端命令所对应的进程结束之后, 命令功能也处理完毕。用户本次上机全部结束后, 通过输入退出命令来结束本次上机过程。

在分时系统中, 交互型作业采用联机作业控制方式, 作业的组织、提交和控制均与批处理作业存在差别。分时系统的作业就是用户的一次上机交互过程, 可以认为终端进程的创建是一个交互型作业的开始, 退出命令的运行结束代表交互型作业的中止。交互型作业的情况和资源需求通过具体命令告知系统, 分时系统用户逐条输入命令, 即提交作业(步)和控制作业运行, 系统则逐条命令执行并给出应答。每输入一条或一组操作命令, 便在系统内部创建一个进程或若干进程来完成相应的命令。

命令解释程序的作用与批处理系统中的 JCL 解释程序相类似, 只不过命令解释程序是从终端接收和执行命令, 而 JCL 解释程序是从作业说明书中读取和执行语句。通常, 操作系统提供

丰富的联机控制命令供交互型用户使用,具体的键盘命令主要有:

- (1) 作业控制类:编辑、编译、链接、运行、排错等;
- (2) 资源申请类:申请/归还主存资源、申请/归还独占型设备等;
- (3) 文件操作类;
- (4) 目录操作类;
- (5) 设备控制类。

现代操作系统很少使用真正的批处理策略,为了减轻联机用户重复输入命令的负担,允许将一组系统命令组织成一个批处理文件,然后对这个文件中的命令进行批处理,使联机用户可以享受到自动批处理操作方式的好处。

2.8 处理器调度算法

2.8.1 低级调度的功能和类型

低级调度的最初对象是进程,随着现代操作系统引入多线程机制,进程演变成资源分配和保护的单位,从而进程只作为中级调度的对象,内核级线程替代进程成为低级调度对象。适用于进程调度的算法一般都适用于内核级线程调度,因此在下面的讨论中,并不严格地区分进程和内核级线程(有时用“进程”,有时用“进程/线程”)。需要指出的是,用户级线程的调度是应用程序自己的事,虽然程序设计语言的函数库会提供默认的调度算法,但事实上它们与操作系统无关。在纯用户级多线程系统中,低级调度的对象依然是进程;而在内核级多线程或混合式策略中,低级调度的对象是内核级线程。

1. 低级调度的主要功能

操作系统的调度程序(scheduler)担负两项任务:调度和分派。前者实现调度策略(scheduling policy),确定就绪态进程/线程竞争使用处理器的次序的裁决原则,即进程/线程何时应放弃CPU和选择哪个进程/线程来执行;后者实现调度机制(scheduling mechanism),确定如何时分复用CPU,处理上下文交换细节,完成进程/线程同CPU的绑定及放弃的实际工作。

当进程/线程正常运行时,为什么会出现低级调度的需求呢?正在执行的进程/线程运行结束或由于某事件而无法继续运行,也可能出现更高优先级的进程/线程,需要剥夺当前使用者所占用的处理器,这些原因都可能引起低级调度需求。具体地说,下述事件发生后需要执行低级调度:创建新进程/线程,进程/线程终止,进程/线程阻塞自己,进程/线程运行过程中发生中断或异常,进程/线程执行系统调用,进程/线程所请求的I/O操作完成,进程/线程耗尽时间片,进程/线程改变优先级,等等。

从概念上来看,调度机制由3个逻辑功能程序模块组成:

(1) 队列管理程序

当一个进程/线程转换为就绪态时,其PCB/TCB会被更新以反映这种变化,队列管理程序

(queuer)将 PCB/TCB 指针放入等待 CPU 资源的进程/线程列表中,每当把进程/线程移入就绪队列时,可计算为此进程/线程分配 CPU 的优先级备用。

(2) 上下文切换程序

当调度程序把 CPU 从正在运行的进程/线程那里切换至另一个进程/线程时,上下文切换程序(context switcher)将当前运行进程/线程的上下文信息保存至其 PCB/TCB 中,恢复选中进程/线程的上下文信息,从而使其占用处理器运行。

(3) 分派程序

当一个进程/线程让出 CPU 资源后,分派程序(dispatcher)被激活。当然,为了运行分派程序,需要将其上下文装入 CPU,分派程序从就绪队列中选择进程/线程,而后完成从其自身到所选择的进程/线程间的又一次上下文切换,把 CPU 让给被选中的进程/线程。

在一些计算机系统中,使用两组或多组硬件上下文寄存器来缩短上下文切换时间,CPU 在核心态使用一组寄存器,CPU 在用户态使用另一组寄存器,那么,在上下文切换的过程中,当操作系统与应用程序代码来回切换时,只需花费改变指向当前寄存器组的指针的时间。

2. 低级调度的基本类型

低级调度的基本类型是指调度策略选择下一个运行进程的时间点和仲裁方式,在其他时间不能改变已获得处理器的进程的分派情况。据此,低级调度有两类基本的调度方式。

第一类称剥夺式(preemptive):又称抢先式。当进程/线程正在处理器上运行时,系统可根据所规定的原则剥夺分配给此进程/线程的处理器,并将其移入就绪队列,选择其他进程/线程运行。有两种常用的处理器剥夺原则,一是高优先级进程/线程可剥夺低优先级进程/线程;二是当运行进程/线程的时间片用完后被剥夺,在动态改变进程/线程优先级的系统中,经常会出现这种情况。

第二类称非剥夺式(nonpreemptive):又称非抢先式。一旦某个进程/线程开始运行后便不再让出处理器,除非此进程/线程运行结束,或主动放弃处理器,或因发生某个事件而不能继续执行。

剥夺式策略的开销比非剥夺式策略的开销要大,主要是调度程序自身的开销,及主存和磁盘对换程序、数据的开销,但是由于可以避免进程/线程长时间地独占处理器,对于实时系统和分时系统有利,能为用户提供高性能的服务。很多操作系统使用两种调度策略的组合,内核关键程序是非剥夺式的,用户进程是剥夺式的。

2.8.2 作业调度和低级调度算法

在操作系统中,存在多种调度算法,有的算法仅适用于作业调度,有的算法仅适用于进程/线程调度,但大多数调度算法对两者都适用。

1. 先来先服务算法

先来先服务算法(First Come First Served, FCFS)按照作业进入系统后备作业队列的先后次序来挑选作业,先进入系统的作业将优先被挑选进入主存,创建用户进程,分配所需资源,然后,

移入就绪队列。这是一种非剥夺式调度算法,易于实现,但效率不高。只顾及作业的等候时间,未考虑作业要求服务时间的长短,不利于短作业而优待长作业,不利于I/O繁忙型作业而有利于CPU繁忙型作业。有时为了等待长作业执行结束,短作业的周转时间和带权周转时间将变得很大,从而使若干作业的平均周转时间和平均带权周转时间也变得很大。例如,有以下3个作业同时到达系统并立即进入调度:

假设系统中没有其他作业,采用FCFS算法进行作业调度,3个作业的周转时间分别为28 ms,37 ms和40 ms,因此有,

$$\text{平均作业周转时间 } T = (28 + 37 + 40) / 3 = 35 \text{ ms}$$

若3个作业的调度顺序改为作业2、1、3,平均作业周转时间缩短为约29 ms;若3个作业的调度顺序改为作业3、2、1,则平均作业周转时间缩短为约18 ms,可见,FCFS算法的平均作业周转时间与作业的提交和调度顺序有关。

FCFS算法同样适用于进程/线程调度,每次从就绪队列中选择最先进入此队列的进程/线程,它一直运行,直至完成或出现等待事件被阻塞而让出处理器为止。这里要说明的是,在进行处理器调度时,允许作业调度与进程/线程调度分别采用不同的调度算法,而且往往需要采用不同的调度算法处理这两级调度。

2. 最短作业优先算法

最短作业优先算法(Shortest Job First,SJF)以进入系统的作业所要求的CPU运行时间的长短为标准,总是选取预计计算时间最短的作业投入运行。这是一种非剥夺式调度算法,能够克服FCFS算法偏爱长作业的缺点,易于实现,但执行效率也不高。SJF算法的主要弱点:一是要预先知道作业所需要的CPU运行时间,很难精确估算,如果估值过低,系统可能提前终止此作业;二是忽视作业的等待时间,由于系统不断地接受新作业,作业调度程序总是选择计算时间短的作业投入运行,因此,进入系统时间早但计算时间长的长作业的等待时间会过长,出现饥饿现象;三是尽管能克服对长作业的偏爱,但由于缺少剥夺机制,对于分时、实时处理仍然不理想。例如,若有4个作业同时到达系统并立即进入调度:

假设系统中没有其他作业,采用SJF算法进行作业调度,作业的调度顺序应为作业2、4、1、3,则

$$\text{平均作业周转时间 } T = (4 + 12 + 21 + 31) / 4 = 17 \text{ ms}$$

$$\text{平均带权作业周转时间 } W = (4/4 + 12/8 + 21/9 + 31/10) / 4 \approx 1.98 \text{ ms}$$

如果对它们施行FCFS调度算法,则

$$\text{平均作业周转时间 } T = (9 + 13 + 23 + 31) / 4 = 19 \text{ ms}$$

$$\text{平均带权作业周转时间 } W = (9/9 + 13/4 + 23/10 + 31/8) / 4 \approx 2.61 \text{ ms}$$

作业名	所需CPU时间/ms
作业1	28
作业2	9
作业3	3

作业名	所需CPU时间/ms
作业1	9
作业2	4
作业3	10
作业4	8

可见, SJF 算法的平均作业周转时间比 FCFS 算法要短, 能提高系统处理作业的吞吐率, 并能给出最小的平均等待时间, 故其调度性能比 FCFS 算法好。但是, 实现 SJF 调度算法需要预先知道作业的运行时间, 否则调度就没有依据。

SJF 调度算法也可用于低级调度, 借用作业的估计运行时间作为进程/线程的估计运行时间, 调度时从就绪队列中选择一个估计运行时间最短者投入运行。如果有两个以上的进程/线程具有相同的估计运行时间, 就对其按照 FCFS 算法处理。更确切地说, SJF 算法应称作“最短下一个 CPU 用时优先(shortest next CPU burst)”法, 在低级调度时需要计算进程/线程的下一个 CPU 周期长度, 而不是进程/线程整体用时的长短, 虽然无法求出下一个 CPU 周期的准确长度, 但是可以根据当前及过去所用的 CPU 周期数据来进行估算。令 τ_n 是估算的第 n 个 CPU 周期长度, τ_{n+1} 是估算的进程下一个 CPU 周期长度, 那么, 估算公式为

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

其中, t_n 的值是进程/线程最近一个 CPU 周期长度, 这是最近的信息; τ_n 是估算的第 n 个 CPU 周期长度, 它保存历史信息; 参数 α 控制在估算值的计算中最近信息与历史信息的权重, 若 $\alpha = 0$, 则 $\tau_{n+1} = \tau_n$, 仅最近 CPU 周期长度起作用。通常, α 取值 0.5, 意思是兼顾最近信息与历史信息。对上式展开可得

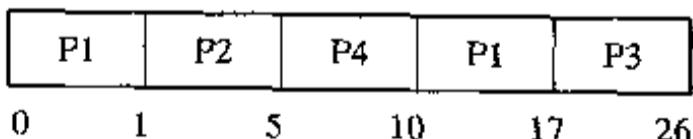
$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_1$$

由于 α 和 $(1 - \alpha)$ 均小于或等于 1, 所以, 每个后继项都比前驱项的权重小, 即历史越久远, 对估算值的影响越小。在发生低级调度时, 首先按此公式计算每个就绪进程/线程的 τ_k (已经计算过的应予以记录), 然后, 调度 τ 值最小的进程运行。

3. 最短剩余时间优先算法

最短作业优先算法是非剥夺式的, 可将其改造成剥夺式调度算法。假设当前某进程/线程正在运行, 如果有新进程/线程移入就绪队列, 若它所需要的 CPU 运行时间比当前运行进程/线程所需要的剩余 CPU 时间还短, 抢占式最短作业优先算法强行剥夺当前执行者的控制权, 调度新进程/线程执行, 这叫做最短剩余时间优先算法 (Shortest Remaining Time First, SRTF), 它确保一旦新的短作业进入系统或新进程/线程就绪就能够很快得到服务。讨论下面的例子, 假设有 4 个就绪进程先后移入就绪队列, 所需要的 CPU 时间如下:

进程	到达系统时间	所需 CPU 时间/ms
P1	0	8
P2	1	4
P3	2	9
P4	3	5



P1 从 0 开始执行, 此时就绪队列中只有一个进程; P2 在时间 1 到达, 而进程 P1 的剩余时间 (7 ms) 大于 P2 所需 CPU 时间 (4 ms), 所以, P1 的控制权被剥夺, P2 被调度执行。此例采用 SRTF 的平均等待时间是

$$((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)) / 4 = 26 / 4 = 6.5 \text{ ms}$$

相应的平均周转时间是

可见, SJF 算法的平均作业周转时间比 FCFS 算法要短, 能提高系统处理作业的吞吐率, 并能给出最小的平均等待时间, 故其调度性能比 FCFS 算法好。但是, 实现 SJF 调度算法需要预先知道作业的运行时间, 否则调度就没有依据。

SJF 调度算法也可用于低级调度, 借用作业的估计运行时间作为进程/线程的估计运行时间, 调度时从就绪队列中选择一个估计运行时间最短者投入运行。如果有两个以上的进程/线程具有相同的估计运行时间, 就对其按照 FCFS 算法处理。更确切地说, SJF 算法应称作“最短下一个 CPU 用时优先(shortest next CPU burst)”法, 在低级调度时需要计算进程/线程的下一个 CPU 周期长度, 而不是进程/线程整体用时的长短, 虽然无法求出下一个 CPU 周期的准确长度, 但是可以根据当前及过去所用的 CPU 周期数据来进行估算。令 τ_n 是估算的第 n 个 CPU 周期长度, τ_{n+1} 是估算的进程下一个 CPU 周期长度, 那么, 估算公式为

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

其中, t_n 的值是进程/线程最近一个 CPU 周期长度, 这是最近的信息; τ_n 是估算的第 n 个 CPU 周期长度, 它保存历史信息; 参数 α 控制在估算值的计算中最近信息与历史信息的权重, 若 $\alpha = 0$, 则 $\tau_{n+1} = \tau_n$, 仅最近 CPU 周期长度起作用。通常, α 取值 0.5, 意思是兼顾最近信息与历史信息。对上式展开可得

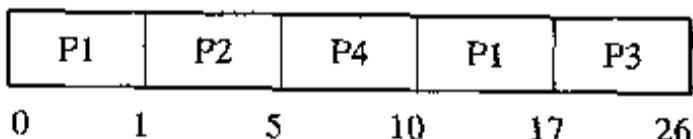
$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_1$$

由于 α 和 $(1 - \alpha)$ 均小于或等于 1, 所以, 每个后继项都比前驱项的权重小, 即历史越久远, 对估算值的影响越小。在发生低级调度时, 首先按此公式计算每个就绪进程/线程的 τ_k (已经计算过的应予以记录), 然后, 调度 τ 值最小的进程运行。

3. 最短剩余时间优先算法

最短作业优先算法是非剥夺式的, 可将其改造成剥夺式调度算法。假设当前某进程/线程正在运行, 如果有新进程/线程移入就绪队列, 若它所需要的 CPU 运行时间比当前运行进程/线程所需要的剩余 CPU 时间还短, 抢占式最短作业优先算法强行剥夺当前执行者的控制权, 调度新进程/线程执行, 这叫做最短剩余时间优先算法 (Shortest Remaining Time First, SRTF), 它确保一旦新的短作业进入系统或新进程/线程就绪就能够很快得到服务。讨论下面的例子, 假设有 4 个就绪进程先后移入就绪队列, 所需要的 CPU 时间如下:

进程	到达系统时间	所需 CPU 时间/ms
P1	0	8
P2	1	4
P3	2	9
P4	3	5



P1 从 0 开始执行, 此时就绪队列中只有一个进程; P2 在时间 1 到达, 而进程 P1 的剩余时间 (7 ms) 大于 P2 所需 CPU 时间 (4 ms), 所以, P1 的控制权被剥夺, P2 被调度执行。此例采用 SRTF 的平均等待时间是

$$((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)) / 4 = 26 / 4 = 6.5 \text{ ms}$$

相应的平均周转时间是

平均作业周转时间 $T = (20 + (25 - 10) + (40 - 5) + (50 - 15)) / 4 = 26.25 \text{ ms}$

平均带权作业周转时间 $W = (20/20 + 15/5 + 35/15 + 35/10) / 4 \approx 2.46 \text{ ms}$

可见 HRRF 算法的性能界于 SJF 算法和 FCFS 算法之间。HRRF 算法同样可用于进程调度, 此时, 响应比可定义为:

$$1 + \text{进程等待处理器时间} / \text{进程估计计算时间}$$

5. 优先级调度算法

优先级调度算法根据确定的优先级来选取进程/线程, 总是选择就绪队列中的优先级最高者投入运行。在进程/线程的运行过程中, 如果就绪队列中出现优先级更高的进程/线程, 系统可预先规定策略: 非剥夺式或剥夺式。前者让当前进程/线程继续运行, 直到它结束或出现等待事件而主动让出处理器, 再调度另一个优先级高的进程/线程运行; 后者则立即重新调度, 剥夺当前运行进程/线程所占有的处理器, 分配给更高优先级的进程/线程使用。

规定用户进程/线程优先级的方法多种多样, 一种方法是用户提出优先级, 称作外部指定法, 有的用户为了尽快获得计算结果, 就设法提高其进程/线程的优先级, 系统可规定优先级越高所需付出的费用就越多, 以施加限制; 另一种方法是由系统综合考虑有关因素来确定进程/线程的优先级, 称作内部指定法。例如, 根据进程/线程类型、空间需求、运行时间、打开文件数、I/O 操作多少、资源申请情况等来确定, 确定优先级时各因素所占的比例应根据系统设计目标来分析这些因素在系统中的权重而定。

优先级通常用 0~4 095 的整数表示, 这个整数叫做优先数, 优先级与优先数之间的关系因系统而异, 如在 UNIX/Linux 系统中, 优先数越小, 优先级越高, 而有些系统做出相反的规定。

进程/线程优先级的确定可采用静态和动态两种方式。静态优先级在进程/线程创建时即确定, 且生命周期中不再改变, 可按照外部指定和内部指定方法计算静态优先级。静态优先级算法的实现简单, 但会产生饥饿现象, 使某些低优先级进程/线程无限期地被推迟执行。动态优先级使各进程/线程的优先级随时间而改变, 基本原则是: 正在运行的进程/线程随着占有 CPU 时间的增加, 逐渐降低其优先级; 就绪队列中等待 CPU 的进程/线程随着等待时间的增加, 逐渐提高其优先级, 等待时间足够长的进程/线程会因其优先级不断提高而被调度运行, 克服了静态优先级的饥饿问题。

6. 轮转调度算法

轮转法调度 (Round-Robin, RR) 也称时间片调度, 具体做法是: 调度程序每次把 CPU 分配给就绪队列首进程/线程使用规定的时间间隔, 称为时间片, 通常为 10 ms~200 ms, 就绪队列中的每个进程/线程轮流地运行一个时间片, 当时间片耗尽时, 就强迫当前运行进程/线程让出处理器, 转而排列到就绪队列尾部, 等候下一轮调度, 所以, 一个耗时型进程/线程需要经过多次轮转才能完成。实现这种调度需要使用间隔时钟, 进程/线程开始运行时, 就将时间片的值置入间隔时钟内, 发生间隔时钟中断时, 表明连续运行且用光时间片, 此时, 时钟中断处理程序通知处理器调度进行进程/线程切换。RR 调度策略可以防止那些很少使用设备的进程/线程长时间地占用处理器, 导致要使用设备的那些进程/线程没有机会去启动设备。

轮转法调度是一种剥夺式调度,系统耗费在进程/线程切换上的开销比较大,这个开销与时间片的大小有关。如果时间片取值太小,将导致大多数进程/线程都不可能在一个时间片内运行完毕,就会频繁切换,开销显著增大,所以,从系统效率的角度来看,时间片取得大一点好;另一方面,时间片的长度较大,那么,随着就绪队列中进程/线程数目的增加,轮转一次所耗费的总时间加长,即对每个进程/线程的响应速度均放慢,甚至时间片大到让进程/线程足以完成其所有任务,RR 调度算法便退化成 FCFS 算法。为了满足用户对响应时间的要求,要么限制就绪队列中的进程/线程数量,要么采用变化的时间片长度,根据当前负载状况,及时调整时间片大小。所以,确定时间片长度要从进程数目、切换开销、系统效率和响应时间等多方面加以考虑。

Windows 2003 时间片由注册表的一个记录项控制,短时间片持续 2 个时钟滴答,用于客户端;长时间片持续 12 个时钟滴答,能够减少线程调度上下文切换的次数,用于服务器端,使得响应性能和吞吐能力更好。可以设置所有线程为定长时间片,也可以把后台和前台线程的时间片设置为不同的值,后者的时间片可取 2~12 个时钟滴答(以 2 为增量)。

7. 多级反馈队列调度算法

多级反馈队列调度(Multi-Level Feedback Queue, MLFQ)又称反馈循环队列,其主要思想是:由系统建立多个就绪队列,每个队列对应于一个优先级,第一个队列的优先级最高,第二个队列的优先级次之,其后队列的优先级逐个降低。较高优先级队列的进程/线程分配给较短的时间片,较低优先级队列的进程/线程分配给较长的时间片,最后一个队列进程/线程按 FCFS 算法进行调度。处理器调度每次先从第一个队列中选取执行者,同一队列中的进程/线程按 FCFS 原则排队,只有在未选到时,才从较低一级的就绪队列中选取,仅当前面所有队列为空时,才会运行最后一个就绪队列中的进程/线程。

进程/线程的优先级可事先规定。例如,使用设备频繁者优先级高,计算型进程/线程的优先级低;终端用户定为高优先级,非终端用户定为低优先级。优先级也可事先不规定,一个新进程/线程被创建后,首先移入第一个就绪队列末尾等待调度,如果能够在给定的时间片内完成,便可从系统中撤离;凡是耗尽时间片仍未运行结束的,就移入下一个就绪队列的末尾,直至进入最低优先级就绪队列。

MLFQ 调度算法具有较好的性能,能满足各类应用的需要。对于分时交互型短作业,系统通常可在第一队列(最高优先级队列)规定的时间片内完成工作,使终端型用户感到满意;对于短的批处理作业,通常只需在第一队列和第二队列中各执行一个时间片就能完成工作,周转时间仍然很短;对于长的批处理作业,它将依次在第一队列、第二队列、第三队列等各个队列中获得时间片运行。例如,在 XDS940 操作系统中,把进程划分为 4 个优先级类,分别是终端、I/O 操作、短时间片和长时间片;当一个等待终端输入的进程被唤醒时,它排入最高优先级类(终端类);当一个等待磁盘数据传输的进程就绪时,它将排入第二类优先级队列;当进程在时间片用完时仍处于就绪态时,它将排入第三类优先级队列;如果一个进程多次用完时间片面从未因终端或其他 I/O 操作的原因发生阻塞,它将被排入第四类优先级队列,即最低优先级类。

MLFQ 调度算法会导致“饥饿”问题。假如有一个长作业进入系统,它最终必将移入优先级

最低的就绪队列中,如果其后有源源不断的短作业进入系统,且形成稳定的作业流,则长作业一直等待,陷入“饥饿”状态。解决此问题的一种有效方法是对于低优先级队列中等待时间足够长的进程提升其优先级,从而让它获得运行机会。如图 2.27 所示是三级反馈队列调度算法示例。

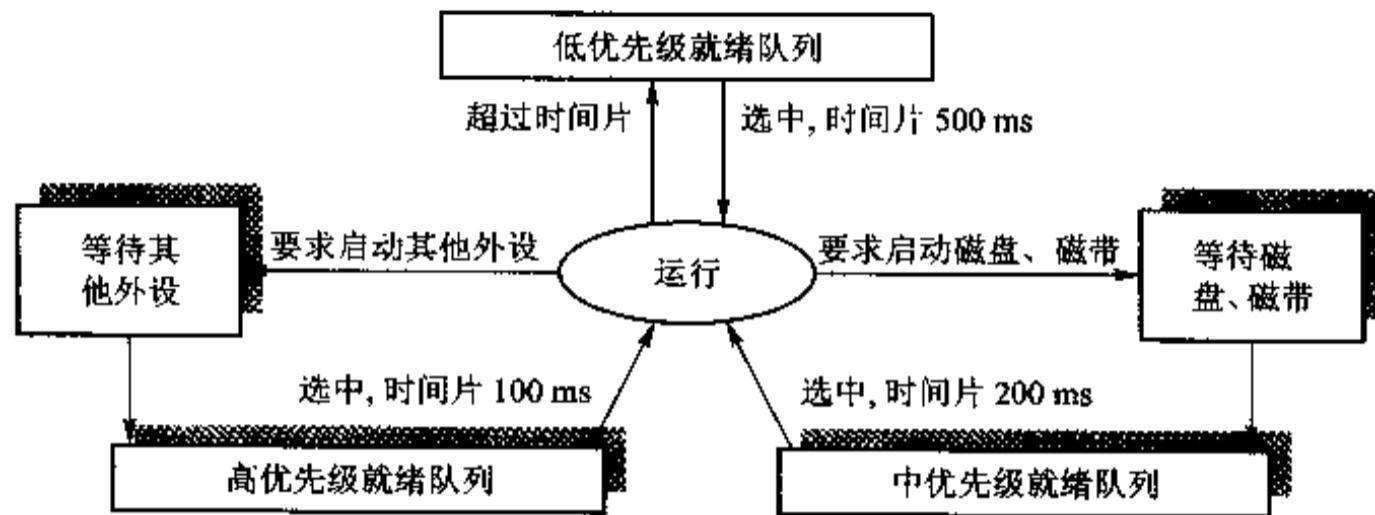


图 2.27 三级反馈队列调度算法示例

8. 彩票调度算法

一种可给出预见的结果、实现起来也较为简单的调度算法称为彩票调度算法 (lottery scheduling), 其基本思想是: 为进程/线程发放针对各种资源(如 CPU 时间)的彩票, 当调度程序需要做出决策时, 随机选择一张彩票, 彩票的持有者将获得相应的系统资源。对于 CPU 调度, 系统可能每秒钟抽取彩票 50 次, 中奖者每次可获得 20 ms 的运行时间。

在这种情况下,所有进程/线程都是平等的,它们享有相同的运行机会。如果某些进程/线程需要更多的机会,就可给予其额外的彩票,以增加中奖机会。如果发出 100 张彩票,某进程拥有 20 张彩票,它就有 20% 的中奖概率,也将获得大约 20% 的 CPU 时间。

彩票调度与优先数调度不同,后者很难说明优先数为 40 终将意味着什么,而前者则很清楚,进程拥有多少彩票份额,就能获得多少资源。彩票调度算法具有有趣的特性:其反应非常迅速,例如,新进程被创建并得到一些彩票,那么在下次抽奖时,这个进程的中奖机会就立即与其所持有的彩票成正比。此外,协作进程之间可交换彩票,客户进程向服务器进程发送一条消息并阻塞,客户进程可以把所持有的彩票全部交给服务器进程,以增加后者下一次被选中运行的机会;服务器进程在完成响应服务之后,又将彩票返还给客户进程使其能够再次运行。实际上,在没有客户时,服务器进程/线程根本不需要彩票。

彩票调度算法还可用来解决其他算法难以解决的问题。例如,一个视频服务器有若干将视频信息传送给相应客户的进程,假设它们分别需要 10 帧/秒、20 帧/秒和 25 帧/秒的传输速度,则分别给这些进程分配 10、20 和 25 张彩票,它们将自动按照正确的比例获得 CPU 资源。

2.8.3 实时调度算法

1. 实时操作系统的特性

实时系统是那些时间因素起关键作用的系统,如监控系统、自动驾驶系统、安全控制系统等,

在这些系统中,迟到的响应即使正确,也同没有响应一样糟糕。实时系统中存在若干实时进程或任务,用来响应外部事件,由于存在时间上的紧迫性,对任务调度提出特殊的要求。实时任务分为硬实时(hard real time)任务和软实时(soft real time)任务,前者意味着必须满足任务的开始截止期限或完成截止期限;后者意味着任务与预期截止时间关联,但偶尔超出截止时间的限制还是可以容忍的。实时进程的行为预先可知,处理周期都很短,当检测到外部事件时,调度程序会按照满足其截止期限的方式调度这些任务。按照任务的执行是否呈周期性来划分,实时任务可分为周期性任务和非周期性任务,前者指以固定的时间间隔出现的事件;后者指事件的出现无法预计,但事先规定开始截止期限或完成截止期限。

一个系统可能有多个周期性事件的事件流,都要求系统做出及时的响应,系统能否及时处理所有事件取决于每个任务的处理时间。例如,有 m 个周期性任务,任务 i 的出现周期为 P_i ,处理所需要的 CPU 时间为 C_i ,则给出条件:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

满足这一条件的实时系统称为任务可调度的。举例来说,一个实时系统要处理 3 个事件,其出现周期分别为 100 ms、200 ms 和 500 ms,如果事件的处理时间分别为 50 ms、30 ms 和 100 ms,则这个系统是任务可调度的,因为

$$0.5 + 0.15 + 0.2 \leq 1$$

如果加入周期为 1 s 的第 4 个事件,只要其处理时间不超过 150 ms,此实时系统仍将是任务可调度的。当然,隐含条件是进程切换的时间足够短,可以忽略不计。

2. 实时调度算法

实时调度算法分为动态实时调度和静态实时调度两类,前者在运行时做出调度决定;后者在提供截止期限等信息的前提下,在系统开始运行之前完成调度决策。下面介绍几种经典的实时调度算法。

(1) 单比率调度算法

在实时系统中,事件是周期性发生的,单比率调度是视周期长度而定的抢占式策略:周期越短,优先级越高。在具体实现时,可事先为每个实时进程/线程分配与事件发生频率成正比的优先数,运行频率越高的进程/线程其优先数越高。例如,周期为 20 ms 的进程/线程的优先数为 50,周期为 100 ms 的进程/线程的优先数为 10,运行时调度程序总是调度优先数最高的就绪进程/线程,并采取抢占式分配策略。

(2) 限期调度算法

当一个事件发生时,对应的实时进程/线程就被加入就绪队列,此队列按照截止期限排序,对于周期性事件,截止期限即为事件下一次发生的时间。限期调度算法首先运行队首的截止期限最近的那个进程/线程;新进程/线程就绪后,系统检测其截止期限是否比当前运行者早,以决定是否剥夺当前运行进程/线程的资源。

(3) 最少裕度法

首先计算各个进程/线程的富裕时间,即裕度(laxity),然后选择裕度最少者执行。计算公式为:

$$\text{裕度} = \text{截止时间} - (\text{就绪时间} + \text{计算时间})$$

裕度越小说明进程/线程越紧迫,就绪后令其尽快运行。

2.8.4 多处理机调度算法

单处理机调度和多处理机调度存在一定的区别,现代操作系统往往采用进程调度与线程调度相结合的方式来完成多处理机调度。

1. 多处理机调度的设计要点

多处理机调度的设计要点有3个:为进程分配处理机、在单个处理机上是否使用多道程序设计技术和实际指派进程的方法。所需考虑的这些问题与可用处理机数目及并行处理的应用程序之间的同步频率密切相关。

(1) 为进程分配处理机

假定多处理器系统中的处理机都是同构的,可将所有处理机放入处理机池,采用静态或动态方式分配。静态分配方式把进程分配到一个处理机上,每个处理机对应于一个就绪队列,其优点是调度开销小,缺点是易造成处理机忙闲不均;动态分配方式让所有处理机共用一个就绪队列,当某处理机空闲时,就选择一个进程占有此处理机运行,这样,进程可以在不同时间运行于不同的处理机上。

无论采取哪种分配策略,系统必须提供相应的分配和调度机制。在主从式结构中,只使用一个调度程序,它运行在主控机上,考虑所有处理机和所有符合条件的进程。这种方式简单易行,调度效率高,但系统的坚定性与主控机上所运行的操作系统之间的关系过大,极易成为系统性能瓶颈。在对等式结构中,不同的处理机有不同特性,各处理机自行实施调度,且可以使用不同的调度程序,从就绪队列中选出进程运行,但必须保证两个处理机不能选中同一个进程,操作系统的调度过程比较复杂。

(2) 在单个处理机上是否使用多道程序设计技术

当有多个处理机可用时,使单个处理机处于繁忙状态不是那么重要,系统追求的是为应用程序提供最好的平均性能。如果一个应用程序由多线程组成,应该让所有线程同时在多处理机上运行,而不是在单个处理机上多道运行,这时性能最佳。

(3) 实际指派进程的方法

大量实验数据证明,随着处理机数目的增多,采用复杂的低级调度算法来调度会降低有效性,这时往往为进程设置一个就绪队列或按照优先数排列的多个就绪队列,采用最简单的FCFS调度算法或优先数算法。这样做不但实现起来简单,而且系统开销很低。

2. 多处理机调度算法

下面讨论几种经典的多处理机进程/线程调度算法。

(1) 负载共享调度算法

负载共享调度算法(load sharing)的基本思想是:进程并不被指派到特定的处理机上,系统维护全局性进程就绪队列,当处理机空闲时,就选择进程的一个线程去运行。

此算法的优点是:把负载均分到所有的可用处理机上,保证没有处理机是空闲的;无须集中调度,调度程序可以运行在不同的处理机上,可采用优先级等方法组织全局队列。此算法的缺点是:就绪队列必须被互斥访问,当系统拥有很多处理机时,将成为性能瓶颈;被剥夺线程很难在原处理机上运行,恢复高速缓存信息会带来系统性能的下降;所有线程都被放在一个公共的池中,一个进程的所有线程未必能同时获得处理机,而如果其线程间的同步频繁,那么进程的切换次数增多将导致系统性能下降。

具体的负载共享调度算法有:先来先服务、最少线程数优先和剥夺式最少线程数优先。著名的 Mach 操作系统包括一个全局共享的就绪线程队列,而且每个处理机还对应于一个局部的就绪线程队列,其中含有临时绑定到此处理机上的就绪线程,处理机调度时首先在局部就绪线程队列中选择绑定线程,如果没有,才到全局就绪线程队列中选择未绑定线程。

(2) 群调度算法

群调度算法(gang scheduling)的基本思想是:基于一对一的原则,一群相关线程被同时调度到一组处理机上运行。紧密相关线程的并行执行能够减少同步阻塞,从而减少进程切换,降低调度代价,系统性能得到提高。

群调度引发对处理机分配的要求。如果有 N 个处理机和 M 个应用进程,每个应用进程至多有 N 个线程,那么,每个应用进程将分得 N 个处理机的可用时间的 $1/M$,此分配策略的效率可能并不高。考虑一个例子,有 4 个处理机和两个应用进程,应用进程 1 有 4 个线程,应用进程 2 有 1 个线程,使用均分时间分配的方法,每个应用进程可获得 50% 的 CPU 时间,由于应用进程 2 只有 1 个线程运行,将有 3 个处理机空闲,所浪费的 CPU 资源为 37.5%。可以采用线程数加权调度法,具体来说,为应用进程 1 分配 80% 的 CPU 时间,为应用进程 2 分配 20% 的 CPU 时间,则处理机时间的浪费可降至 15%。

(3) 专用处理机调度算法

专用处理机调度算法(dedicated processor assignment)的基本思想是:将同属一个进程的一组线程同时分派到一组处理机上运行,每个线程均获得一个处理机,专用于处理这个线程,直到进程运行结束。这是群调度的一种极端形式。

采用专用处理机调度算法,处理机将不适用于多道程序设计,当一个线程因等待事件而阻塞后,其所专用的处理机处于空闲状态。此算法所追求的是通过高度并行来达到最快的执行速度,它在应用进程的运行过程中避免低级调度和切换。对于拥有几十或数百个处理机的高度并行系统而言,单个处理机只占系统总代价的一小部分,处理机的使用效率不再是衡量有效性或系统性能的重要因素。

(4) 动态调度算法

在实际应用中,一些应用程序通过语言或工具能够动态地改变进程中的线程数目,这就要求操作系统能够调整负载以提高处理机利用率。动态调度算法(dynamic scheduling)的基本思想

是：由操作系统和应用进程共同做出调度决策，操作系统负责在应用进程之间分配处理机；应用进程在所分配的处理机上执行可运行线程的子集，哪些线程应该执行？哪些线程应该挂起？完全是应用进程的任务，可借助于运行时库函数完成。

在动态调度算法中，当一个进程请求一个或多个处理机时，操作系统的调度程序按照下面的原则进行处理。

- ① 如果有空闲处理机，则满足请求；
- ② 否则，对于新创建的进程，从当前分得多个处理机的任一进程中收回一个处理机，把它分配给新进程；
- ③ 如果这个请求的任何分配都不能被满足，则此进程保持未完成状态，直到有处理机可用或此进程取消请求；
- ④ 当释放一个或多个处理机时，对尚未处理过的请求处理机的进程队列进行扫描，且为队列中尚未分配处理机的每个进程分配一个处理机，然后，再次扫描进程队列，按照先来先服务原则分配其余处理机。

Linux 调度算法

2.9.1 Linux 传统调度算法

Linux 2.4 内核调度函数 `schedule()` 决定采用何种调度策略选择就绪进程运行，确保进程所获得的 CPU 访问时间和其指定的优先级及进程类型相匹配，消除进程对 CPU 资源的饥饿状态。调度机制由内核实现，包括调度时机的选择、调度参数的计算等工作。

1. 进程调度策略

在进程 `task_struct` 结构中，存放着与进程调度有关的重要成员供进程调度使用。

(1) `policy`: 标识进程的调度策略，有以下三种类型。

① `SCHED_OTHER` 普通类任务：普通进程采用基于优先级的时间片轮转法调度，只要有实时进程就绪，这类进程便不能运行。

② `SCHED_FIFO` 先进先出实时类任务：进程一直运行，除非出现等待事件或有另一个具有更高 `rt_priority` 的实时进程出现，才让出处理器。

③ `SCHED_RR` 轮转法实时类任务：除了时间片是定量之外，与 `SCHED_FIFO` 类似，在时间片耗尽之后，使用相同的 `rt_priority` 排到原队列的队尾。

(2) `priority` 进程静态优先级：内核 V2.4 及以上版本已取消此变量。

(3) `nice` 进程可控优先级因子：其值是 -20~19 的整数，可用于改变进程的静态优先级。其默认值为 0，增加 `nice` 值会降低进程的优先级。

(4) `rt_priority` 实时进程静态优先级：仅被实时进程所使用，是 0~99 的一个整数，用于区分实时进程的等级，权值较高的进程总是优先于权值较低的进程。`rt_priority + 1000` 给出实时

进程的优先级,因此,实时进程的优先级总是高于普通进程。

(5) counter:进程目前时间片配额(单位为时钟滴答),也称进程动态优先级。可运行队列中的普通进程都有 counter 值,每次时钟中断时其值由 update_process_times() 函数减 1。counter 值等于 0 时,表示进程的时间片耗尽。注意 counter 的用法,对于采用时间片轮转法调度的普通进程,counter 才起作用;对于采用 SCHED_RR 的实时进程,counter 仅起到时间片的作用,并不计算动态优先级;对于采用 SCHED_FIFO 的实时进程,counter 值将被忽略,此时的时间片可看做无穷大。

2. 动态优先级的产生和变化

于进程的静态优先级和调度策略 policy 是在其创建时从父进程那里继承而来的,且在进程的生命周期内保持不变,除非使用相关的系统调用来设置它。在执行 do_fork() 函数创建子进程时,其动态优先级 counter 也从父进程处继承,其值为(父进程的 counter 值 + 1)/2。此后,每当发生时钟中断时,运行进程的 counter 值减 1,直到发生下列情况时停止。

(1) 当 counter 值递减至 0 时,运行进程被迫让出处理器;当可运行队列中所有进程的 counter 值变为 0 后,表明一轮调度已经结束。

(2) 处于等待态进程的动态优先级通常会逐渐增加,当所有可运行进程的 counter 值都为 0 时,系统将重新计算进程的 counter 值,这既包括就绪态进程,也包括等待态进程。计算公式为

$$p->counter = (p->counter >> 1) + \text{NICE_TO_TICKS}(p->nice)$$

即进程的 counter 值右移一位,加上此进程的 nice 值,得到新的 counter 值。对于就绪态进程而言,因其 counter 值都为 0,计算结果就是由 nice 值转换而来的时钟滴答数。等待态进程则未然,其 counter 值都不为 0,计算结束后,等待态进程的动态优先级会大于 nice 值。因此,等待态进程会有较高的优先级,处于等待态越久的进程,其动态优先级越高。计算量较大的进程因其占用 CPU 过多会使优先级越来越低。

3. Linux 进程调度机制

(1) 进程调度的依据和时机

进程调度的依据有以下几种。

① 进程被动放弃 CPU,当前进程的时间片用完,或一个进程被唤醒且其优先级高于当前进程的优先级时,task_struct 中的 need_resched 置 1,通知内核需要重新调度。

② 进程主动放弃 CPU,是由于进程执行系统调用,状态发生变化,直接调用 schedule() 进入调度。这类系统调用有 yield()、pause()、sleep()、wait() 和 exit()。

③ 进程的执行等待系统调用,如 read 或 write 等,此时进程进入等待队列,系统调用 schedule() 进入调度,此函数的执行结果往往是当前进程放弃处理器。

在进程被动放弃 CPU 时,内核并不在 need_resched 置 1 后立即执行 schedule() 函数,而是在适当的时刻才检测重调度标志。内核将调度时机安排在:系统调用的处理结束即将返回时和中断服务程序的处理结束即将返回时,在这两种情形下,系统均需调用 ret_from_sys_call,只有在这个程序段中才检测 need_resched,若需要的话则执行调度函数 schedule()。

(2) 进程调度任务

每当 schedule() 函数工作时, 完成以下任务。

① 处理软中断服务请求

如果有软中断服务请求, 则先响应这些请求。

② 处理当前进程

(a) 如果当前进程的调度策略为 SCHED_RR 且 counter 值为 0, 则保持运行状态(TASK_RUNNING), 并将其移入可运行队列尾部, counter 在适当时刻被重新赋值。

(b) 如果当前进程是可中断(TASK_INTERRUPTIBLE)的, 且信号已经到达, 则将其状态修改为 TASK_RUNNING, 移入可运行队列尾部。

(c) 把当前既非运行态、又非可中断状态的进程从可运行队列中移出, 这些进程暂无资格被调度, 把当前进程描述符的 need_resched 标志清 0。

③ 选择进程运行

(a) 当 CPU 空闲时, 调度函数扫描可运行队列中的所有就绪进程, 从中选择合适的进程运行, 这时需要执行 goodness() 函数, 最终最大的权值 weight 保存在变量 c 中, 表示进程值得运行的程度; 变量 next 指向调度后所要运行的进程; 计算普通进程的权值 weight 时, 内核先计算静态优先级 20 - nice(由 -20 ~ 19 转换成 40 至 1 的值), 再将其加到 counter 上算出进程的权值:

```
weight = counter + 20 - nice;
```

```
c = weight;
```

(b) 如果 c 值为 0, 表明所有可运行进程的时间片已经耗尽, 应重新计算进程的时间片, 对 counter 重新赋值, 再次转而执行 goodness() 函数。

(c) 如果 next 就是当前进程, 则结束调度工作, 让当前进程返回运行; 否则, 进行进程切换, 改由 next 进程占有 CPU。

goodness() 函数的参数是待查进程的描述符, 返回值 c 反映待查进程值得运行的程度。c 的取值范围如下:

$c = -1000$: 永远不必选择待查进程, 当可运行队列中仅有一个进程时, c 值为 -1000。

$c = 0$: 待查进程的时间片用完, 在其他进程的时间片耗尽之前不会选择它。

$0 < c < 1000$: 待查进程的时间片尚未用完, 剩余时间片可看做进程动态优先级。

$c > 1000$: 待查进程是实时进程, 应优先运行。

如果进程是实时进程, 其权值为 $1000 + rt_priority$, 这是普通进程无法达到的权值, 能够保证实时进程总是比普通进程优先运行。对于普通进程, 其权值为 $counter + 20 - nice$, 如果它又是内核线程, 由于无须切换至用户空间, 则将权值加 1 作为奖励。

(d) 当 CPU 空闲且不存在可运行态进程时, 进程调度程序选择空闲进程 task0 投入运行, 空闲进程不能被终止, 也不会进入阻塞状态, 即总是处于运行态, 与其他进程不同的是其 task_struct 数据结构由系统静态地分配。

(3) 进程切换

`switch_to()` 函数可进行进程切换, 改由 `next` 进程占用处理器, 切换代码与处理器密切相关, 用汇编代码来实现, 按照 `task_struct` 结构中的相关信息进行进程上下文切换; 如果当前进程或新进程使用虚拟主存, 那么相关页表必须同步更新。

4. SMP 进程调度

Linux 2.0 版开始支持 SMP, 多处理器系统中的每个 CPU 上都可以运行一个进程, 因此, 在实施进程调度时, 可供选择的 CPU 不止一个。在每个进程的 `task_struct` 结构中都含有当前所用的 CPU 编号及刚用过的 CPU 编号, 虽然系统并不限制进程每次必须在同一个 CPU 上运行, 但可以利用 `processor_mask` 掩码限制进程只能在某些 CPU 上运行。如果掩码的第 n 位设置为 1, 则此进程可在第 n 个 CPU 上运行, 即进程可与某个 CPU 绑定, 称为亲缘关系。在实施进程调度时, 总是优先考虑进程在其刚用过的 CPU 上运行, 因为此 CPU 的高速缓存中最有可能包含此进程的一些特定数据, 若改变 CPU 则会增加对存储器的访问时间, 在一定程度上影响系统效率。然而, 亲缘关系的使用必须特别仔细, 在某个进程限制于特定 CPU 而无其他进程可在此 CPU 上执行时, CPU 亲缘关系很有意义。但是这仅针对系统有特别多的 CPU 而言, 在一个双 CPU 计算机系统中使用亲缘关系肯定太过奢侈。

2.9.2 Linux 2.6 调度算法

从 Linux 2.5 内核版本开始使用新的称为 $O(1)$ 的调度程序, 其特点是能够保证无论系统负载(进程数目或处理器数目)怎样增加, 选择合适的进程且为其分配处理器的时间是恒定的。此外, 还具有新的特性: 支持 SMP, 每个处理器都拥有可运行队列; 强化 SMP 的亲和力, 尽量将相关任务分配到一个处理器上连续运行; 确保响应时间, 及时地调度交互式进程; 保证公平性, 无进程处于饥饿状态。进程描述符 `task_struct` 中含有与调度有关的成员是: `policy`、`static_prio`、`prio`、`rt_priority`、`sleep_avg`、`time_slice` 和 `load_weight` 等。

1. 调度算法的数据结构

调度程序中最主要的数据结构是可运行队列(`runqueue`), 它是给定处理器上的就绪进程链表, 每个处理器都有可运行队列, 每个就绪进程都归属于一个可运行队列。此外, 可运行队列还包含每个处理器的调度信息, 所以也是处理器的重要数据结构。

```
struct runqueue {
    spinlock_t lock;                      // 可运行队列自旋锁
    unsigned long nr_running;               // 总的就绪进程数
    unsigned long nr_switches;              // 上下文切换数
    unsigned long expired_timestamp;        // 队列最后被对换时间
    unsigned long nr_uninterruptible;       // 不可中断阻塞态的任务数
    struct task_struct * curr;              // 处理器当前运行进程
    struct task_struct * idle;              // 处理器的空闲进程
    struct mm_struct * prev_mm;            // 最后运行任务的 mm_struct
```

```

...
| else
idx = sched_find_first_bit(array->bitmap);           //从 active 队列选择进程
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
...
                                         //任务切换

```

4. 负载平衡

系统中的每个处理器都有自己的运行队列,只对属于自己的进程进行调度。如果运行队列出现负载不均的情况,则由负载平衡程序 `load_balance` 来解决这个问题。有两种调用方法,在执行 `schedule()` 函数时,只要当前可运行队列为空就会被调用,找到一些就绪进程并插入这个队列中;负载平衡程序还会被定时器调用,每隔适当的时间调用一次,这时需要解决所有运行队列间的失衡问题。

`load_balance()` 函数工作时需要锁住当前处理器的运行队列并屏蔽中断,避免出现竞争,负载平衡任务完成后,解除对当前运行队列的锁定并开放中断。具体的操作步骤如下。

(1) 找到最繁忙的运行队列,此队列中的就绪进程数目最多,如果没有其他运行队列中的就绪进程数目比当前运行队列中的就绪进程数目多 25% 或更多,就结束平衡负载处理。

(2) 找到最繁忙的运行队列,从中选择一个优先级数组以便抽取就绪进程,最好是过期数组,因为那里的就绪进程已经有相对较长的时间未曾运行,很可能不在处理器的高速缓存中。如果过期数组为空,就只能选择活跃数组。

(3) 找到含有进程且优先级最高的链表,把优先级高的就绪进程平均分散开来。

(4) 分析所找到的所有优先级相同的就绪进程,选择一个未运行、不会因为处理器相关性而不可移动且不在高速缓存中的进程。如果有满足这些条件的进程,便将其从最繁忙的队列中抽取至当前可运行队列。

(5) 只要可运行队列之间的负载仍然不均衡,就重复上述步骤,继续从繁忙队列中抽取进程到当前可运行队列,最终消除不平衡局面。

5. 用户抢占和内核抢占

内核即将返回用户空间时,如果 `TIF_NEED_RESCHED` 标志被设置,将导致 `schedule()` 被调用,此时发生用户抢占。这时内核知道自己是安全的,既可继续执行当前进程,也可选择新进程运行。用户抢占通常发生在:从系统调用返回用户空间时;从中断处理程序返回用户空间时。

Linux 2.6 完整地支持内核抢占,只要重新调度是安全的,内核就可以在任何时间抢占正在运行的任务。那么,重新调度何时是安全的呢?只要没有持有锁,内核就可以进行抢占,锁是非抢占区域的标志。由于内核支持 SMP,所以,如果没有持有锁,正在运行的代码是可重入的,也就是可抢占的。

为了支持内核抢占而做的一处修改是为进程引入 `preempt_count` 计数器,其初值为 0,每当使用锁时此计数器的值加 1,释放锁时其值减 1。当计数器的值为 0 时,内核就可以执行抢占。

指定后就不能改变。调度程序所用到的动态优先级存放在 prio 域中,它通过一个关于静态优先级和进程交互性的函数关系计算而来。

`effective_prio()` 函数可以返回一个进程的动态优先级,它以 nice 值为基数,再加上 -5~+5 之间的进程交互性奖励或罚分。例如,对于交互性很强的进程,即使其 nice 值为 10,它的动态优先级最终也有可能达到 5;相反地,对于温和的处理器吞噬者,虽然 nice 值原本是 10,它最后的动态优先级却可能是 12;交互性不强不弱的进程不会得到优先级奖励,同样也不会被罚分,其动态优先级与 nice 值相当。

如果进程大部分时间都在睡眠,它属于 I/O 消耗型;如果进程的执行时间比睡眠时间长,它属于处理器消耗型。Linux 记录进程用于睡眠和执行的时间,此值存放在 `task_struct` 的 `sleep_avg` 域中,范围从 0 到 `MAX_SLEEP_AVG`,默认值为 10 ms。当一个进程从睡眠状态恢复到运行态时,`sleep_avg` 会根据睡眠时间的长短而增加,直至达到 `MAX_SLEEP_AVG` 为止;相反地,进程每运行一个时钟滴答,`sleep_avg` 就相应地递减,直至 0 为止。

上述计算不仅基于睡眠时间,而且运行时间也要被计算进去,所以,尽管某进程的睡眠时间很长,但是如果它把自己的时间片用得一干二净,就不会得到优先级奖励。这种推断机制不仅会奖励交互性强的进程,还会惩罚处理器资源耗费量大的进程,如果某个进程发生变化,开始大量地占用处理器时间,它很快就会失去曾经得到的优先级提升。

由于动态优先级已经以 nice 值和交互性为基础来取值,重新计算时间片就相对比较简单,在进程创建时,子进程和父进程平均分配父进程的剩余时间片,这样的分配方法很公平,可以防止用户通过不断创建新进程来不停地攫取时间片。然而,当一个进程的时间片用完后,就要根据进程的动态优先级重新计算时间片。`task_timeslice()` 函数为给定任务返回一个新的时间片,时间片的计算只需将优先级按比例进行缩放,使其符合时间片的数值范围要求,进程的优先级越高,它每次执行所得到的时间片就越长。优先级最高的进程所能获得的最大时间片长度(`MAX_TIMESLICE`)是 800 ms;优先级最低的进程所获得的最短时间片(`MIN_TIMESLICE`)是 5 ms;默认优先级(`nice=0`,没有交互性奖励或罚分)的进程所得时间片长度为 100 ms。

调度程序还提供另外一种机制以支持交互式进程。如果进程的交互性非常强,当其时间片用完之后,会再次被放置到活跃数组而不是过期数组中。重新计算时间片是通过活跃数组与过期数组之间的切换来进行的,进程在用尽时间片后,通常会被移至过期数组,当活跃数组中没有剩余进程时,这两个数组就会被交换。活跃数组变成过期数组,过期数组替代活跃数组,对换操作提供时间复杂度为 $O(1)$ 的时间片重新计算。在发生对换之前,交互性很强的某进程有可能处于过期数组中,当它需要交互时却无法执行,因为必须等待数组发生交换,而将交互式进程重新插入活跃数组可以避免这类问题。

3. $O(1)$ 调度算法

调度算法由 `schedule()` 函数实现,既简单又有效。对于给定的处理器,调度程序选择优先级活跃数组中第一个被设置的位,对应于优先级最高的就绪进程队列,然后,选择此优先级链表中的表头进程,它就是要被调度执行的进程。如图 2.28 所示是 Linux 2.6 处理器的调度数据结构

示意图,其中仅画出活跃队列的优先级数组及相应的优先级队列;过期队列的优先级数组及相应的优先级队列完全类似。新调度程序减少对循环的依赖,活跃数组内的可执行队列上的进程都有剩余时间片;过期数组内的可执行队列上的进程都耗尽时间片。当某进程的时间片用完时,重新计算它的时间片,并移至过期数组。因为数组是通过指针进行访问的,活跃和过期数组之间来回切换所用的时间就是交换指针所需要的时间。

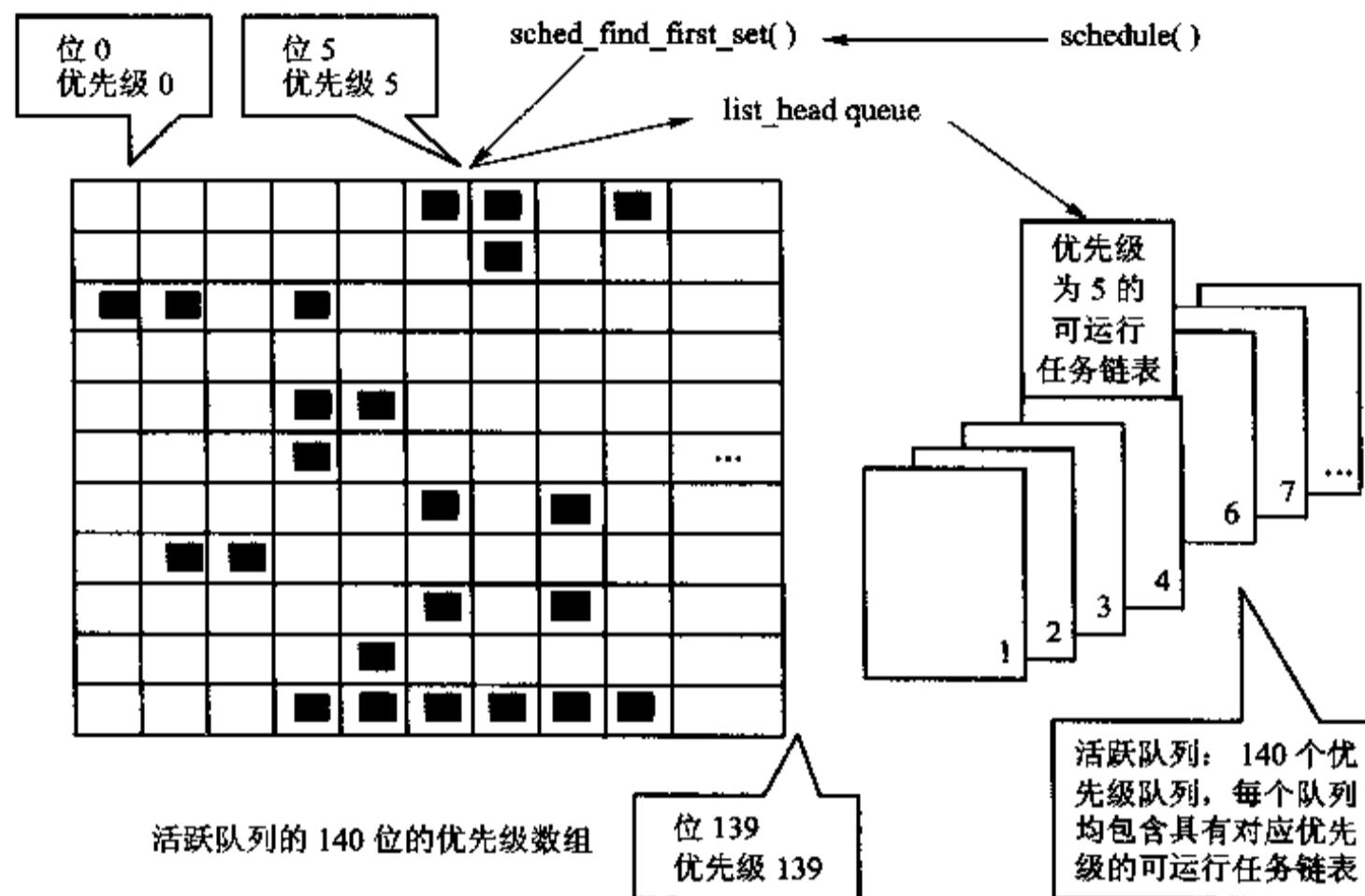


图 2.28 Linux 2.6 处理器调度数据结构图

schedule()函数进入调度,寻找和调度最高优先级进程的执行过程则由以下代码段实现。

```

task_t * prev, * next;
runqueue * rq;
prio_array_t * array;
int idx;

prev = current; //保存当前进程
rq = this_rq(); //当前运行队列
array = rq->active;
if (unlikely(!array->nr_active)) { //运行队列的 active 变成空
    rq->active = rq->expired; //活跃、过期数组切换
    rq->expired = array;
    array = rq->active;
}

```

```

...
| else
idx = sched_find_first_bit(array->bitmap);           //从 active 队列选择进程
queue = array->queue + idx;
next = list_entry(queue->next, task_t, run_list);
...
                                         //任务切换

```

4. 负载平衡

系统中的每个处理器都有自己的运行队列,只对属于自己的进程进行调度。如果运行队列出现负载不均的情况,则由负载平衡程序 load_balance 来解决这个问题。有两种调用方法,在执行 schedule() 函数时,只要当前可运行队列为空就会被调用,找到一些就绪进程并插入这个队列中;负载平衡程序还会被定时器调用,每隔适当的时间调用一次,这时需要解决所有运行队列间的失衡问题。

load_balance() 函数工作时需要锁住当前处理器的运行队列并屏蔽中断,避免出现竞争,负载平衡任务完成后,解除对当前运行队列的锁定并开放中断。具体的操作步骤如下。

(1) 找到最繁忙的运行队列,此队列中的就绪进程数目最多,如果没有其他运行队列中的就绪进程数目比当前运行队列中的就绪进程数目多 25% 或更多,就结束平衡负载处理。

(2) 找到最繁忙的运行队列,从中选择一个优先级数组以便抽取就绪进程,最好是过期数组,因为那里的就绪进程已经有相对较长的时间未曾运行,很可能不在处理器的高速缓存中。如果过期数组为空,就只能选择活跃数组。

(3) 找到含有进程且优先级最高的链表,把优先级高的就绪进程平均分散开来。

(4) 分析所找到的所有优先级相同的就绪进程,选择一个未运行、不会因为处理器相关性而不可移动且不在高速缓存中的进程。如果有满足这些条件的进程,便将其从最繁忙的队列中抽取至当前可运行队列。

(5) 只要可运行队列之间的负载仍然不均衡,就重复上述步骤,继续从繁忙队列中抽取进程到当前可运行队列,最终消除不平衡局面。

5. 用户抢占和内核抢占

内核即将返回用户空间时,如果 TIF_NEED_RESCHED 标志被设置,将导致 schedule() 被调用,此时发生用户抢占。这时内核知道自己是安全的,既可继续执行当前进程,也可选择新进程运行。用户抢占通常发生在:从系统调用返回用户空间时;从中断处理程序返回用户空间时。

Linux 2.6 完整地支持内核抢占,只要重新调度是安全的,内核就可以在任何时间抢占正在运行的任务。那么,重新调度何时是安全的呢?只要没有持有锁,内核就可以进行抢占,锁是非抢占区域的标志。由于内核支持 SMP,所以,如果没有持有锁,正在运行的代码是可重入的,也就是可抢占的。

为了支持内核抢占而做的一处修改是为进程引入 preempt_count 计数器,其初值为 0,每当使用锁时此计数器的值加 1,释放锁时其值减 1。当计数器的值为 0 时,内核就可以执行抢占。

从中断返回内核空间的时候,内核会检查 TIF_NEED_RESCHED 和 preempt_count 的值,如果 TIF_NEED_RESCHED 已被设置且 preempt_count 的值为 0,说明有更为重要的任务需要执行并且可安全地抢占,此时就会调用调度程序。如果 preempt_count 的值不为 0,说明当前任务持有锁,抢占是不安全的,这时就会如常直接从中断返回当前运行进程。如果当前进程所持有的所有锁已经被释放,那么 preempt_count 的值会重新为 0,此时,释放锁的代码会检查 TIF_NEED_RESCHED 是否被设置,如果已被设置,就会调用调度程序。内核抢占发生时刻如下:

- (1) 当从中断处理程序返回内核空间的时候;
- (2) 当内核代码再一次具有可抢占性的时候;
- (3) 如果内核中的任务显式调用 schedule(); 如果内核中的任务阻塞,也会导致调用 schedule()。

有些内核代码需要允许或禁止内核抢占,这可以通过 preempt_disable() 和 preempt_enable() 实现。由于内核是可抢占的,内核中的进程在任何时刻都有可能停止以便另一个具有更高优先级的进程运行,这意味着一个任务与被抢占的任务可能会在同一个临界区内运行。为了避免发生这类情况,内核抢占代码使用自旋锁作为非抢占区域的标记,如果一个自旋锁被持有,内核便不能进行抢占。因为内核抢占和 SMP 面对相同的并发性问题,而且内核已经是 SMP 安全的,因此,这种简单的变化使得内核也是抢占安全的。



Windows 2003 调度算法

1. 线程调度及其特征

Windows 支持内核级线程,是一个基于优先级抢占式的多处理器调度系统。系统总是运行优先级最高的就绪线程,线程可以在任何可用的处理器上运行,但可限制某线程只能在某处理器上运行,亲合处理器集合允许用户线程通过 Win32 调度函数选择其所偏好的处理器。

Windows 在内核中实现线程调度代码,这些代码分布于内核中与调度相关的事件的出现位置,并不存在单独的线程调度模块。内核中完成线程调度功能的函数统称为内核调度器,线程调度出现在 DPC/dispatch 中断优先级,触发事件如下。

- (1) 一个线程进入就绪态,如新线程刚被创建或线程刚刚结束等待态。
- (2) 一个线程由于时间配额用完而从运行态转入退出态或等待态。
- (3) 一个线程由于调用系统服务而改变优先级或被系统本身改变其优先级。
- (4) 一个正在运行的线程改变其亲合处理器集合。

在这些事件出现时,系统选择下一个将要运行的线程。当选中一个新线程进入运行态时,将执行线程上下文切换,保存当前线程的运行环境,加载另一个线程的运行环境,并开始执行新线程。

处理器调度是严格地针对线程进行的,并不考虑被调度线程属于哪个进程。例如,进程 A 有 10 个可运行的线程,进程 B 有两个可运行的线程,若这些线程的优先级相同,则每个线程将得到

1/12的处理器时间。

2. 进程优先级

在调用 CreateProcess 时,根据被创建进程的类型向它指派优先级,通常用默认值 NORMAL_PRIORITY_CLASS 表示,进程被分为 4 种类型:空闲进程(优先级为 4),普通进程(优先级为 7 或 9),高优先级进程(优先级为 13),实时进程(优先级为 24)。这也是进程的相应线程开始运行时的优先级。

当进程在前台运行时,其优先级为 9,而进程在后台运行时,其优先级为 7。从一个进程切换到另一个进程时,新激活进程转换成前台进程,原运行进程则变为后台进程。如果新激活进程具有普通优先级,将其优先级由 7 升为 9,而新的后台进程的优先级由 9 降为 7,使前台进程的响应时间更短。仅在需要时使用高优先级,任务管理器 TASKMAN.exe 以高优先级 13 运行,此系统进程的线程平时被挂起,一旦用户单击“开始”菜单,它就立即被系统唤醒,并抢占处理器。会话管理器、服务控制器和本地认证服务器的优先级比默认值 8 要高,以保证这些进程的线程有较高的初始默认值。

实时进程的优先级通常为 24,用于核心态系统进程,完成存储管理、高速缓存管理、本地和网络文件系统及设备驱动程序的功能。

3. 线程优先级

Windows 的调度是基于内核级线程的抢占式调度,包括多个优先级层次。在某些层次中,线程的优级是固定的,而在另一些层次中,线程的优先级将根据执行情况作动态调整,采用动态优先级多级反馈队列调度策略,每个优先级都对应于一个就绪队列,每个线程队列中的线程按照时间片方式轮转调度。

Windows 的内部使用 32 个线程优先级,范围为 0~31,它们被分成以下 3 个部分。

(1) 线程实时优先级(优先数为 16~31)

线程实时优先级用于通信任务和实时任务。一个线程被赋予实时优先数之后,在执行过程中此优先数不可变,一旦一个就绪线程的实时优先级比运行线程高,它将抢占处理器运行。

(2) 线程可变优先级(优先数为 1~15)

线程可变优先级用于用户提交的交互式任务。具有这一层次优先数的线程可根据执行过程中的具体情况动态地调整优先数,但优先数不能突破 15。

(3) 系统线程优先级(0)

系统线程优先级仅用于对系统中的空闲物理页面进行清 0 的零页线程。

线程在被创建时,其优先级继承自所属进程的优先级,所有线程的初始优先级均为普通,用户可通过 Win32 指定线程优先级,内核也可以根据线程在前台或后台运行来动态地升高或降低线程优先级。通过 Win32 所指定的优先级由进程优先级类型和线程相对优先级共同控制。线程的基本优先级在进程优先级的基础上分为 5 级:高优先级线程(比所属进程的优先级高 2 级)、中上优先级线程(比所属进程的优先级高 1 级)、中优先级线程(等于所属进程的优先级)、中下优先级线程(比所属进程的优先级低 1 级)和低优先级线程(比所属进程的优先级低 2 级)。

在进程内所创建的各线程其相对优先级可设定为相对实时、相对高级、相对中上级、中级、相对中下级、相对低级和相对空闲级。线程在启动时继承进程的基本优先级(中级),而后根据需要改变其优先级。

表 2.1 给出进程优先级与线程相对优先级之间的映射关系,进程只有基本优先级(对应于中级所在行),而线程则有当前优先级和基本优先级两个优先级的值,线程的当前优先级值的范围在 1~15 内动态变化,通常会高于基本优先级。系统从不调整实时范围在 16~31 内的线程优先级,故这些线程的基本优先级和当前优先级相同。

表 2.1 进程优先级与线程优先级的映射

Win32 线程优先级	进程优先级类为中级			
	实时级	高级	普通级	空闲级
相对实时	31	15	15	15
相对高级	26	15	10	6
相对中上级	25	14	9	5
中级	24	13	8	4
相对中下级	23	12	7	3
相对低级	22	11	6	2
相对空闲级	16	1	1	1

4. 线程时间配额

当线程被调度进入运行态时,它可以运行一个时间配额(quantum),时间配额是允许线程连续运行的最大时间总和,系统此后会中断线程的运行,判断是否需要降低此线程的优先级,并查找是否有其他高优先级或相同优先级的线程等待运行,不同版本 Windows 的时间配额不同,同一系统中各线程的时间配额可被修改。由于系统具有抢占式调度的特征,因此,一个线程的一次调度执行可能未用完其时间配额,如果一个高优先级的线程进入就绪态,当前运行的线程可能在用完其时间配额之前就被抢占。事实上,一个线程甚至可能在被调度进入运行态之后且开始运行之前就被抢占。

时间配额不是时间的长度值,而是配额单位(quantum unit)的一个整数,在应用性能部分有一个滑动块,控制应用程序的时间配额:默认值、居中值和最大值,可调整的时间长度为:Windows Professional 版的默认值为 6、居中值为 12、最大值为 18;Windows Server 版对应的 3 个值均为 36。Windows Server 版中取较长默认时间配额的原因是,要保证客户请求所唤醒的服务器有足够的在其时间配额用完之前完成客户请求并返回到等待态。每次发生时钟中断时,时钟中断服务程序便从线程的时间配额中减少固定值 3,如果没有剩余时间配额,系统将选择其他线程运行。

允许用户指定线程时间配额的相对长度(长或短)和前台线程(指拥有当前窗口的线程所在的进程)的时间配额是否加长。那么,为什么要增加前台线程的时间配额,而不是提高前台线程的优先级呢?这是因为当前台、后台应用程序都在计算时,一味地提高前台线程的优先级将导致后台线程几乎得不到处理器时间,但增加前台线程的时间配额则不会停止后台线程的运行,而只

是给前台线程的时间多些。

5. 线程调度程序数据结构

如图 2.29 所示,为了进行线程调度,内核维护一组“调度程序数据结构”,负责记录各线程的状态,如哪些线程处于等待态、处理器正在执行哪个线程等。调度程序数据结构中最主要的是调度器的就绪(KiDispatcherReadyListHead)队列,此队列由一组子队列组成,每个优先级都有一个队列,共 32 个。

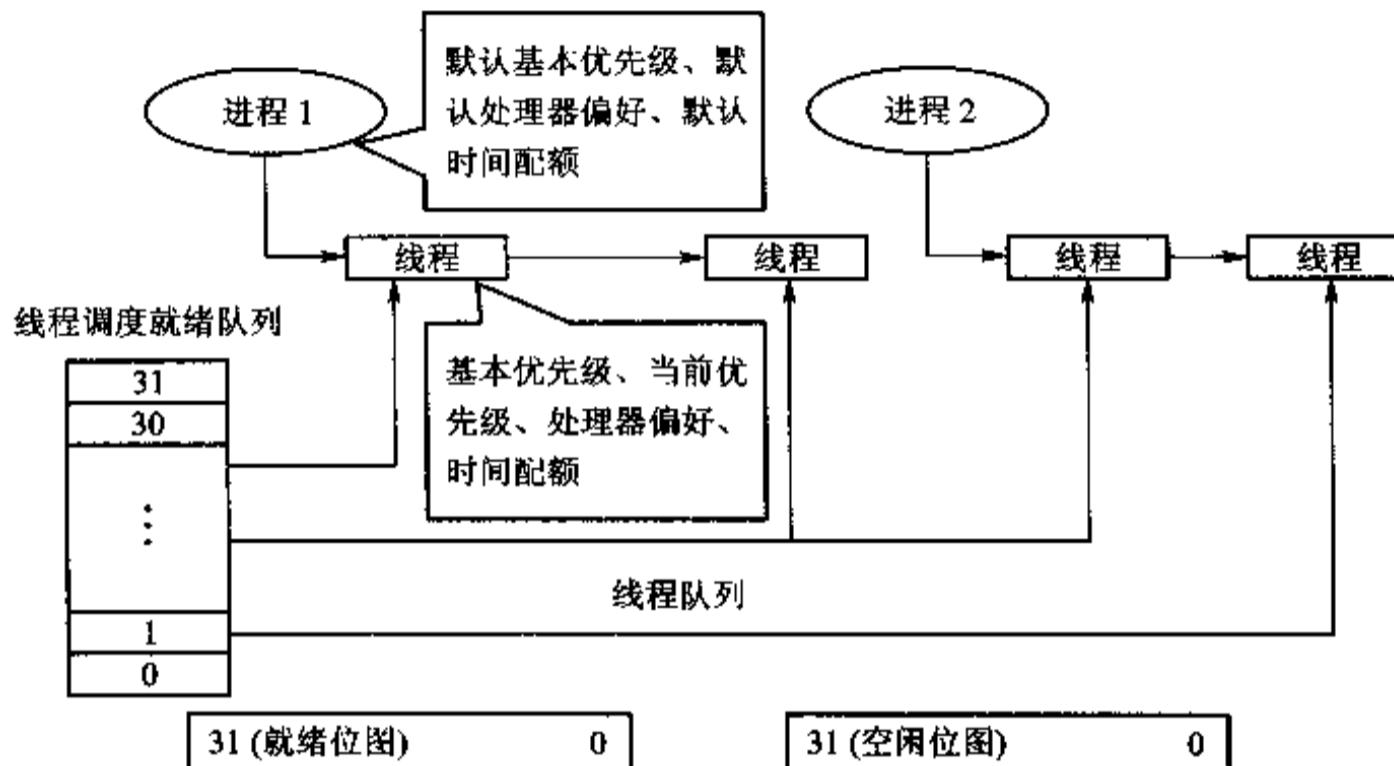


图 2.29 线程调度程序数据结构

为了加快调度速度,Windows 维护一个称为就绪位图(KiReadySummary)的 32 位掩码,就绪位图中的每一位指示一个调度优先级的就绪队列中是否有线程等待运行,位 0 与调度优先级 0 相对应,位 1 与调度优先级 1 相对应,等等;Windows 还维护一个称为空闲位图(KiIdleSummary)的 32 位掩码,空闲位图中的每一位指示一个处理器是否处于空闲状态。为了防止调度器代码与线程在访问调度器数据结构时发生冲突,线程调度仅出现在 DPC/Dispatch 中断优先级,在多处理器系统中,修改调度器数据结构需要额外的步骤来得到内核调度程序自旋锁,以协调各处理器对调度程序数据结构的访问。

6. 线程调度策略

严格地基于线程的优先级采用抢占式策略来确定哪个线程将占用处理器并进入运行态,Windows 在单处理器系统和多处理器系统中的线程调度是不同的。在此介绍单处理器系统中的线程调度。

(1) 主动切换

线程因进入等待态而主动放弃处理器,许多 Win32 等待函数的调用都使得线程等待某个对象,包括事件、互斥信号量、资源信号量、I/O 操作、进程、窗口消息等。当线程主动放弃所占用的处理器时,将调度就绪队列中的第一个线程运行,主动放弃处理器的线程会被降低优先级,但这

并不是必需的,可以仅被放入等待对象的等待队列中。

(2) 抢占

当一个高优先级线程进入就绪态时,正处于运行态的低优先级线程将被抢占,这可能在以下情况中发生:高优先级线程所等待的对象出现信号,即线程等待事件完成;或线程的优先级被提高,系统都要确定是否让当前线程继续运行,或被高优先级线程所抢占。需要注意的是,在用户态运行的线程可抢占在内核态运行的线程,在判断线程是否被抢占时,并不考虑线程是处于用户态还是内核态,调度程序只是依据线程的优先级进行判断。当线程被抢占时,它被放至相应优先级就绪队列的队首,而非队尾。

(3) 时间配额耗尽

当处于运行态的线程用完其时间配额时,Windows 必须确定是否需要降低此线程的优先级,再确定是否需要调度另一个线程进入运行态。如果刚用完时间配额的线程的优先级被降低了,系统将寻找更适合的线程进入运行态,即优先级高于刚用完时间配额的线程的新设置值的就绪线程。如果刚用完时间配额的线程的优先级未被降低,且存在优先级相同的其他就绪线程,系统将选择相同优先级的就绪队列中的下一个线程进入运行态,刚用完时间配额的线程被排列至就绪队列的队尾,即分配一个新的时间配额并把线程状态从运行态改为就绪态。

(4) 结束

当线程完成运行时,其状态从运行态转换到终止态,线程完成运行的原因可能是通过调用 ExitThread 而从主函数中返回,或被其他线程通过调用 TerminateThread 来终止。如果处于终止态的线程对象上没有未关闭的句柄,则此线程将被从进程的线程列表中删除,相关的数据结构将被释放。

7. 线程优先级提升

线程优先级提升的目的是改进系统吞吐量、响应时间等整体性能,解决线程调度策略中潜在的不公正性。与其他调度算法一样,线程优先级提升也不是完美的,它并不会使所有应用都受益,但系统永远不会提升实时优先级范围内的线程优先级,因此,在实时优先级范围内的线程调度总是可预测的。在下列 5 种情况下,Windows 会提升线程的当前优先级。

(1) I/O 操作完成后的优先级提升

在完成 I/O 操作之后,系统将临时提升等待此操作的线程的优先级,以保证等待 I/O 操作的线程有更多机会立即处理所得结果。线程优先级的实际提升值是由设备驱动程序决定的,在完成 I/O 请求时,通过内核函数 IoCompleteRequest 来指定优先级的提升幅度。线程优先级的提升幅度与 I/O 请求的响应时间要求通常是一致的,对响应时间的要求越高,优先级的提升幅度越大,如磁盘、光盘驱动器、并行接口和视频为 1;网络、串行接口和命名管道为 2;键盘和鼠标为 6;音频为 8。

线程优先级的提升是以线程的基本优先级为基点,而不是以线程的当前优先级为基点,线程优先级提升后将运行一个时间配额,在时间配额用完后,线程会降低一个优先级,并运行另一个时间配额。这个降低的过程会一直进行下去,直到线程的优先级降为原来的基本优先级,其他优

先级较高的线程仍可抢占因 I/O 操作而提升优先级的进程。

(2) 信号量或等待事件结束后的优先级提升

当一个等待事件对象或信号量对象的线程完成等待后,将被提升一个优先级,且在等待结束时,线程的时间配额减 1,在提升后的优先级上将执行剩余的时间配额。随后降低一个优先级,并运行另一个时间配额,这个降低的过程会一直进行下去,直到线程的优先级降为原来的基本优先级。

(3) 前台线程完成等待操作后的优先级提升

对于前台线程,一个内核对象上的等待操作完成时,为了改进交互型应用的响应时间,内核会提升线程的当前优先级,提升幅度为变量 PsPrioritySeparation 的值。窗口子系统负责确定哪个线程是前台线程,在前台应用完成其等待操作时,小幅度提升其优先级,以便立即进入运行态。

(4) 图形用户接口线程被唤醒后的优先级提升

为了改进交互型应用的响应时间,拥有窗口的线程在被窗口消息唤醒时,将得到一个幅度为 2 的额外的优先级提升,在调用函数 KeSetEvent 时实施这种优先级提升。

(5) 对处理器饥饿线程的优先级提升

如果一个优先级为 11 的线程正在等待被优先级为 5 的另一线程所占用的资源,若优先级为 5 的线程得不到处理器执行,将导致优先级为 11 的线程被无限期推迟执行,这称为优先级逆转问题,下面是 Windows 的解决办法。平衡集管理器每秒检查一次就绪队列,查找是否存在一直在就绪队列中排队超过 300 个时钟中断间隔的线程(约 3 s~4 s)。如果找到这样的线程,将其优先级提升至 15,并分配给它一个长度为正常值 2 倍的时间配额。当被提升线程用完其时间配额后,此线程的优先级立即衰减至其原来的基本优先级。如果在此线程结束之前出现其他高优先级的就绪线程,此线程会被放回就绪队列,并在就绪队列中等待超过另外 300 个时钟中断间隔后再次被提升的优先级。为了减少系统开销,平衡集管理器每次只扫描 16 个就绪线程,如果就绪队列中还有更多的线程,它将记住临时位置,以便下次继续扫描。与此同时,每次平衡集管理器扫描最多提升 10 个线程的优先级,如果一次扫描已提升 10 个线程的优先级,表明系统当前十分繁忙,平衡集管理器会停止扫描,并在下次开始时从当前位置继续扫描。这种算法很有效,处于 CPU 饥饿中的线程通常都能得到足够的处理器时间来完成其操作。

8. 对称多处理器系统上的线程调度

如果完全基于线程优先级进行线程调度,在多处理器系统中将出现什么情况?当 Windows 调度优先级最高的可执行线程时,会有若干因素影响到处理器的选择。在此介绍几个概念。

(1) 亲合关系

每个线程都有一个亲合掩码(affinity mask),描述此线程可在哪些处理器上运行,这个亲合掩码是从进程的亲合掩码继承而来的。在默认状态,所有进程的亲合掩码是系统上所有可用处理器的集合,即所有线程均可在所有处理器上运行。应用程序通过 Win32 API 修改亲合掩码。

(2) 线程的首选处理器和最近使用的处理器

每个线程在对应的内核线程控制块中都保存有两个处理器标识:

- ① 首选处理器:线程运行时的偏好处理器。
- ② 最近使用的处理器:线程最近刚用过的处理器。

线程的首选处理器是基于进程控制块的索引值在线程创建时随机选择的,索引值在每个线程创建时递增,这样进程的每个新线程所得到的首选处理器会在系统中的可用处理器范围内循环。线程创建之后,应用程序可通过 Win32 API 来修改线程的首选处理器。

(3) 就绪线程的运行处理器选择

当线程进入运行态时,Windows 首先试图调度此线程到一个空闲处理器上运行。如果有多个处理器空闲,线程调度器的调度顺序如下。

- ① 线程的首选处理器。
- ② 如果线程指定亲合处理器集合,并且此集合中有处理器空闲,则将可选处理器集合缩小至此亲合处理器集合与空闲处理器集合的交集。
- ③ 如果此线程最近使用的处理器在此空闲集合中,则选择这个处理器。
- ④ 最后选择当前执行处理器(即正在执行调度器代码的处理器)。
- ⑤ 如果这些处理器都不空闲,系统将依据处理器标识从高到低扫描系统中的空闲处理器的状态,选择所找到的第一个空闲处理器。

如果线程进入就绪态时所有处理器都处于繁忙状态,系统将检查是否可抢先一个处于运行态或备用态的线程,若可以,将首先选择线程的首选处理器,其次是线程最近使用的处理器。如果这两个处理器都不在线程的亲合掩码中,Windows 将依据活动处理器掩码选择此线程可运行的编号最大的处理器作为线程运行的处理器。

如果被选中的处理器已有一个线程处于备用态(即在此处理器上运行的下一个线程),并且此线程的优先级低于正在检查的线程,则正在检查的线程将取代原处于备用态的线程,成为此处理器的下一个运行线程。如果有线程在被选中的处理器上运行,系统将检查当前运行线程的优先级是否低于正在检查的线程。如果正在检查的线程的优先级高,则标记当前运行线程为被抢占,系统会发出一个处理器中断,以抢占正在运行的线程,让新线程在此处理器上运行。如果在被选中的处理器上不存在线程可被抢占,则将新线程放入相应优先级的就绪队列中等待调度。

9. 空闲线程调度

在多处理器系统中,系统为每个处理器设有一个空闲线程,其优先级为 0,当处理器上无线程可运行时,系统会调度相应处理器上的空闲线程。空闲线程的功能就是在一循环中检测是否有工作要进行:处理所有待处理的中断请求;检查是否有待处理的 DPC 请求,如果有,则清除相应的软件中断并执行 DPC;检查是否有就绪线程可进入运行态,如果有,则调度线程运行,否则调用硬件抽象层的处理器空闲例程,执行相应的电源管理功能。

本 章 小 结

中断和异常是激活操作系统的两种手段,中断装置识别并响应中断事件,通过交换 PSW 让

中断处理程序占有处理器，在处理完此中断事件后，通常会改变一些进程的状态，引起相应进程队列的调整，然后，返回被中断程序或转向低级调度执行，由于中断能够改变处理器内操作的执行顺序，它成为操作系统实现并发性的硬件基础之一；异常事件由硬件指令逻辑发现，其响应、处理和通回的过程与中断类似。中断按照事件来源和实现手段可分为硬中断和软中断，硬中断又分为外中断（中断、异步中断）和内中断（异常、同步中断）。异常可分为错误、陷阱、终止和编程异常。软中断又分为信号和软件中断。

进程是操作系统中最重要和最基本的概念之一，引入进程是系统资源的有限性和系统内操作的并发性所决定的。进程具有生命周期，由创建而产生，由调度而执行，由撤销而消亡，操作系统的基本功能是进程的创建、管理和撤销。为了实现对进程的管理，操作系统维护每个进程的资料结构——进程控制块，这是进程的唯一标志，创建进程必须为其创建进程控制块，撤销进程时系统便回收进程控制块。

进程是并发程序的执行，是动态的概念，但是为了在处理器上执行仍然需要静态描述。从结构上来看，进程由控制块、程序块、数据块和核心栈组成，进程的静态描述及其处理器执行环境合称为进程上下文，包括用户级上下文、系统级上下文、寄存器上下文。一个进程在运行过程中或执行系统调用，或产生中断事件，处理器都会进行模式切换，操作系统接收控制权，内核完成必需的操作之后，或恢复被中断进程或切换至新进程。当系统调度新进程占有处理器时，新老进程随之发生上下文切换，操作系统在完成必要的操作之后，可以恢复被中断的进程或切换至其他进程。

如果说操作系统中引入进程的目的是使多个程序并发执行，以便改善资源利用率和提高系统效率，那么，在操作系统中再引入线程，则是为了减少程序并发执行时所付出的时空代价，使得并发粒度更细、并发性更好。在多线程环境中，进程封装管理信息，线程封装执行信息；线程的实现有用户级线程、内核级线程和混合式三种方法。

处理器调度分为三级：高级调度、中级调度和低级调度。高级调度用于决定哪些满足资源需求的后备作业被选中进入主存去多道运行，FCFS、SJF、SRTF、HRRF、优先级调度算法等是常用的作业调度算法；低级调度用于决定选择哪个进程或线程占有处理器运行，FCFS、RR、优先数、MLFQ 等是常用的进程/线程调度算法；实时系统调度算法有：单比率调度、限期调度和最少裕度调度；多处理器调度算法有：负载共享调度、群调度、处理器专派调度和动态调度。衡量调度算法优劣的因素包括响应时间、周转时间、资源利用率、作业吞吐率和公平性等。

习题二

一、思考题

1. 什么是 PSW？其主要作用是什么？
2. 当从具备运行条件的程序中选取一道程序运行后，怎样才能让它占有处理器工作？
3. 为什么现代计算机要设置两种或多种 CPU 状态？

4. 试述多种 CPU 状态与两种 CPU 状态相比较的优点。
5. 为什么要把机器指令分成特权指令和非特权指令？
6. 硬件如何发现中断事件？发现中断事件后应做什么工作？
7. 从中断事件的性质来说，可以把它分成哪些类型？
8. 从中断事件的来源来说，可以把它分成哪些类型？
9. 从中断事件的实现来说，可以把它分成哪些类型？
10. 试述中断处理程序所应完成的任务。
11. 概述程序性中断的处理方式。
12. 简述时钟中断的处理过程。
13. 何谓中断的优先级？为什么要对中断事件进行分级？
14. 为什么中断事件的处理可以嵌套，但不能递归？
15. 试述系统调用的执行过程。
16. 试述中断在操作系统中的重要性及其主要作用。
17. 试述时钟中断在操作系统中的重要性及其主要作用。
18. 试述中断屏蔽的作用，哪些中断事件可被屏蔽？
19. 操作系统如何处理多重中断事件？
20. 试解释中断号和中断向量。
21. 试讨论硬中断与软中断。
22. 解释 Windows 2003 的中断、异常和陷阱。
23. Windows 2003 如何动态地实现中断屏蔽功能？
24. 何谓 DPC？它何时产生及如何调度？
25. 何谓 APC？它在何种环境下执行？
26. 简述 Linux 的时钟机制。
27. 简述 Linux 的快中断与慢中断。
28. 简述 Linux 中断源的分类及处理原则。
29. 简述 Linux 中断下半部分处理的原理、数据结构和处理过程。
30. 讨论中断下半部分处理与队列机制之间的关系。
31. 讨论 Linux 的 bottom half、task queue、tasklet、work queue 和 softirq 机制。
32. 什么是进程？计算机操作系统中为什么要引入进程？
33. 进程有哪些主要属性？试解释之。
34. 进程最基本的状态有哪些？哪些事件可能引起不同状态之间的转换？
35. 五态模型的进程中，新建态和终止态的主要作用是什么？
36. 试说明引发创建一个进程的主要事件。
37. 多数时间片轮转调度使用固定大小的时间片，请给出：
 - (1) 选择小时间片的理由。
 - (2) 选择大时间片的理由。
38. 什么是进程的挂起状态？列出挂起进程的主要特征。
39. 什么情况下会产生挂起等待态和挂起就绪态？试举例说明。

40. 试述组成进程的基本要素，并说明其作用。
41. 何谓进程控制块(PCB)？它包含哪些基本信息？
42. 何谓进程队列、入队和出队？
43. 请列举组织进程队列的各种方法。
44. 试述创建进程系统所做的主要工作。
45. 什么是进程的上下文？简述其主要内容。
46. 什么是进程切换？试述进程切换的主要步骤。
47. 什么是模式切换？它与进程切换之间有何差别？
48. 进程切换主要应保存哪些处理器状态？
49. 试述引起撤销一个进程的主要事件。
50. 试述系统撤销进程时所做的主要工作。
51. 简单解释操作系统功能的实现模型。
52. 在 UNIX 系统中，为什么把 PCB 的 proc 和 user 结构分离？
53. 列举进程被阻塞和唤醒的主要事件。
54. 在操作系统中引入进程概念后，为什么还要引入线程的概念？
55. 列举支持线程概念的商业性操作系统。
56. 试述多线程环境中，进程和线程的定义。
57. 什么是线程控制块(TCB)？它有哪些主要内容？
58. 试从调度、并发性、拥有资源和系统开销等 4 个方面对传统进程和线程进行比较。
59. 什么是内核级线程、用户级线程和混合式线程？试对其进行比较。
60. 试对下列系统任务进行比较：
 - (1) 创建一个进程和创建一个线程。
 - (2) 两个进程间通信与同一进程中的两个线程间通信。
 - (3) 同一进程中两个线程的上下文切换与不同进程中两个线程的上下文切换。
61. 列举与线程状态变化有关的线程操作。
62. 挂起状态与线程之间有何关系？为什么？
63. 试述并发多线程程序设计的主要优点及其应用。
64. 列举线程的组织方式和应用场合。
65. 试述 Linux 的进程和线程。
66. 试分析 Linux 的 fork()、vfork() 和 clone() 系统调用。
67. 试述 Windows 2003 中的进程和线程的概念。
68. 试述 Windows 2003 中的进程对象和线程对象。
69. 试述 Windows 2003 中的线程状态及其相互转换的原因。
70. 试述 Windows 2003 的内核对象和执行体对象。
71. 试说明访管指令与特权指令之间的区别。
72. 试说明访管指令与系统调用之间的联系和区别。
73. 处理器调度分为哪几种类型？简述各类调度的主要任务。
74. 试述衡量一个处理器调度算法优劣的主要标准。

75. 试述作业调度和低级调度之间的关系。
76. 试述中级调度的主要作用。
77. 解释：
 - (1) 作业周转时间；
 - (2) 作业带权周转时间；
 - (3) 响应时间；
 - (4) 吞吐率。
78. 简述批处理作业的组织、输入和调度过程。
79. 什么是 JCB？列举其主要内容和作用。
80. 试述作业、进程、线程和程序之间的关系。
81. 试述作业、作业步和作业流的概念。
82. 何谓响应比最高优先算法？它有何主要特点？
83. 在时间片轮转低级调度算法中，根据哪些因素确定时间片的长短？
84. 优先权调度是否会导致进程进入饥饿状态？为什么？
85. 当系统内的所有进程都因种种原因进入睡眠之后，系统还有可能复活吗？
86. 为什么多级反馈队列算法能较好地满足各种用户的需求？
87. 分析静态优先数和动态优先数低级调度算法各自的优、缺点。
88. 试述剥夺式和非剥夺式低级调度策略。
89. 试述高级调度和低级调度各自所使用的调度“参数”。
90. 试述典型的实时调度算法。
91. 试述典型的多 CPU 调度算法。
92. 列出并解释传统 UNIX 的动态优先数计算公式。
93. 试述 Linux 2.4 和 Linux 2.6 处理器调度算法。
94. 试述 Windows 2003 的处理器调度算法。
95. 证明：在非抢占式调度算法中，最短作业优先算法具有最小的平均等待时间。
96. 试述处理器的三级调度模型和两级调度模型。
97. 在多级反馈队列中，对不同的队列分配大小不同的时间片值，其意义何在？
98. 试述先来先服务调度算法和时间片轮转调度算法之间的本质区别。
99. 在操作系统设计中强调将机制与策略分离，请提出一种调度机制（scheduling mechanism），允许父进程控制其子进程的调度策略（scheduling policy）。
100. 现有一组进程被分别排入三级优先级队列（高、中、低），若各级之间采用优先级调度法，而同级进程之间采用时间片轮转调度法，试述这批进程被调度执行的过程。
101. 有一个单向连接的进程队列，其队首进程由系统队列标志指出，其队尾进程的队列指针为 null。分别写出一个进程从队首、队中和队尾入队/出队的工作流程。
102. 有一个双向连接的进程队列，其队首进程由系统队列标志指出，其队尾进程的队列指针为 null。分别写出一个进程从队首、队中和队尾入队/出队的工作流程。
103. 总结 Linux 中的各种进程队列。
104. 为什么说操作系统是由中断驱动的？

二、应用题

1. 下列指令中,哪些只能在核心态运行?

- (1) 读时钟日期;
- (2) 访管指令;
- (3) 设时钟日期;
- (4) 加载 PSW;
- (5) 置特殊寄存器;
- (6) 改变存储器映像图;
- (7) 启动 I/O 指令。

2. 假设有一种低级调度算法是让“最近使用处理器较少的进程”运行,试解释这种算法对“I/O 繁重”型作业有利,但并不是永远不受理“处理器繁重”型作业。

3. 并发进程之间有何种制约关系?下列日常生活中的活动属于哪种制约关系?

- (1) 踢足球;
- (2) 吃自助餐;
- (3) 图书馆借书;
- (4) 电视机生产流水线工序。

4. 在按照动态优先数调度进程的系统中,每个进程的优先数需定时重新计算。在处理器不断在进程之间交替的情况下,重新计算进程优先数的时间从何而来?

5. 若后备作业队列中同时等待运行的有 3 个作业 Job1、Job2、Job3,已知其各自的运行时间为 a、b、c,且满足 $a < b < c$,试证明采用短作业优先调度算法能获得最小的平均作业周转时间。

6. 若有一组作业 J_1, J_2, \dots, J_n ,其执行时间依次为 S_1, S_2, \dots, S_n 。这些作业同时到达系统,并在单处理器上按照单道方式执行。试找出一种作业调度算法,使得平均作业周转时间最短。

7. 假定执行作业 Job1 ~ Job5,作业号即为其到达顺序,依次在时刻 0 按照序号 1、2、3、4、5 进入单处理器系统。

(1) 分别采用先来先服务调度算法、时间片轮转算法、短作业优先算法及非抢占优先权调度算法计算出各作业的执行次序(注意优先权越高其数值越小);

(2) 计算每种情况下作业的平均周转时间和平均带权周转时间。

8. 在道数不受限制的多道程序系统中,作业进入系统的后备队列时立即进行作业调度。现有 4 个作业进入系统,有关信息列举如下,作业调度和进程调度均采用高优先级算法(规定数值越大则优先级越高)。

作业号	执行时间/ms	优先权
Job1	10	3
Job2	1	1
Job3	2	3
Job4	1	4
Job5	5	2

作业名	进入后备队列的时间	执行时间/min	优先数
Job1	8:00	60	1
Job2	8:30	50	2
Job3	8:40	30	4
Job4	8:50	10	3

试填充下表。

作业名	进入后备队列的时间	执行时间/min	开始执行时间	结束执行时间	周转时间/min	带权周转时间/min

平均周转时间 $T =$

带权平均周转时间 $W =$

9. 对某系统进行监测后表明,每个进程在 I/O 阻塞之前的平均运行时间为 T ,一次进程切换的系统开销时间为 S 。若采用时间片长度为 Q 的时间片轮转法,对下列各种情况计算 CPU 利用率。

- (1) $Q = \infty$ (2) $Q > T$ (3) $S < Q < T$ (4) $Q = S$ (5) Q 接近于 0。

10. 有 5 个待运行的作业,预计其运行时间分别是 9、6、3、5 和 x ,采用哪种运行次序可以使得平均响应时间最短?

11. 有 5 个批处理作业 A~E 均已到达计算中心,其运行时间分别为 2 min、4 min、6 min、8 min 和 10 min;各自的优先级分别规定为 1、2、3、4 和 5,其中 5 是最高级。对于时间片轮转算法、优先数法、短作业优先算法、先来先服务调度算法(按照作业到达次序 C、D、B、E、A),在忽略进程切换时间的前提下,计算出平均作业周转时间。(对于时间片轮转算法,每个作业获得相同的时间片;对于其他算法采用单道运行方式,直到结束。)

12. 有 5 个批处理作业 A~E 均已到达计算中心,其运行时间分别为 10、6、2、4 和 8 min;各自的优先级分别规定为 3、5、2、1 和 4,其中 5 是最高级。若不考虑系统切换的开销,计算出平均作业周转时间。

- (1) FCFS(按照作业 A、B、C、D、E 的顺序);
 (2) 优先级调度算法;
 (3) 时间片轮转法。

13. (1) 假定一个处理器正在执行两道作业,其中一道作业以计算为主,另一道作业以 I/O 操作为主,将怎样赋予其占有处理器的优先级?为什么?

(2) 假定一个处理器正在执行 3 道作业,第一道作业以计算为主,第二道作业以 I/O 操作为主,第三道作业为计算与 I/O 操作均匀。应该如何赋予其占有处理器的优先级,使得系统效率较高?

14. 请设计一种先进的计算机体系结构,它使用硬件而非中断来完成进程切换,则 CPU 需要哪些信息?请描述用硬件完成进程切换的工作过程。

15. 在单道批处理系统中,下列 3 个作业采用先来先服务调度算法和最高响应比优先算法进行调度,哪一种算法的性能较好?请完成下表。

作业	提交时间	运行时间	开始时间	完成时间	周转时间/min	带权周转时间/min
1	10:00	2:00				
2	10:10	1:00				

续表

作业	提交时间	运行时间	开始时间	完成时间	周转时间/min	带权周转时间/min
3	10:25	0:25				

平均周转时间 $T =$ 带权平均周转时间 $W =$

16. 若有如下表所示的 4 个作业进入系统, 分别计算在 FCFS、SJF 和 HRRF 算法下的平均周转时间和带权平均周转时间。

17. 如果在限制为两道的多道程序系统中, 有 4 个作业进入系统, 其进入系统时间、估计运行时间列于下表中。系统采用 SJF 作业调度算法, 采用 SRTF 进程调度算法, 请填充下表。

作业	提交时间	估计运行时间/min
1	8:00	120
2	8:50	50
3	9:00	10
4	9:50	20

作业	进入系统时间	估计运行时间/min	开始运行时间	结束运行时间	周转时间/min
Job1	10:00	30			
Job2	10:05	20			
Job3	10:10	5			
Job4	10:20	10			

平均周转时间 $T =$ 带权平均周转时间 $W =$

18. Kleinrock 提出一种动态优先权算法: 进程在就绪队列中等待时, 其优先权以速率 α 变化; 当进程在处理器上运行时, 其优先权以速率 β 变化。为参数 α 、 β 赋予不同的值可得到不同算法。

(1) 若 $\alpha > \beta > 0$, 这是什么算法?

(2) 若 $\alpha < \beta < 0$, 这是什么算法?

19. 在单处理器多道分时系统中, 有 3 道作业依次提交, 其提交时间、运行时间分别如下表所示。

作业	作业提交时间	运行时间/hr	其中	
			I/O 时间/hr	CPU 时间/hr
Job1	8.0	0.36	0.18	0.18
Job2	8.2	0.32	0.16	0.16
Job3	8.4	0.36	0.18	0.18

如果已知下列情况:

(1) 每道作业的 I/O 等待时间占各自总运行时间的一半;

(2) 分时运行两道作业, CPU 将有 20% 的时间空闲;

(3) 除了 CPU, 系统有充足的资源供作业使用。

试计算各作业运行完成时间。

20. 有一个 4 道作业的操作系统, 若在一段时间内先后到达 6 个作业, 其提交时间和估计运行时间由下表给出:

系统采用剩余 SJF 调度算法, 作业被调度进入系统后中途不会退出, 但作业运行时可被剩余时间更短的作业所抢占。

- (1) 分别给出 6 个作业的执行时间序列, 即开始执行时间、作业完成时间、作业周转时间。

(2) 计算平均作业周转时间。

21. 有一个具有 3 道作业的多道批处理系统, 作业调度采用短作业优先调度算法, 进程调度采用以优先数为基础的抢占式调度算法。在下表所示的作业序列中, 作业优先数即为进程优先数, 优先数越小则优先级越高。

作业名	到达时间	估计运行时间/min	优先数
A	10:00	40	5
B	10:20	30	3
C	10:30	60	4
D	10:50	20	6
E	11:00	20	4
F	11:10	10	4

试填充下表。

作业	进入主存时间	运行结束时间	作业周转时间/min
A			
B			
C			
D			
E			
F			

平均作业周转时间 =

22. 设有 4 个进程 P_1, P_2, P_3, P_4 , 它们到达就绪队列的时间、运行时间及优先级如下表所示。

进程	到达就绪队列的时间	运行时间/ms	优先级
P_1	0	9	1
P_2	1	4	3
P_3	2	8	2
P_4	3	10	4

(1) 若采用可剥夺的优先级调度算法,给出各个进程的调度次序以及进程的平均周转时间和平均等待时间。

(2) 若采用时间片轮换调度算法,且时间片取 2 ms,给出各个进程的调度次序以及平均周转时间和平均等待时间。

23. 有 5 个作业依次进入系统,其提交时间、运行时间、作业长度分别列于下表。设主存容量为 100 KB,采用可变分区存储管理,且作业在主存储器中不能移动。作业调度采用先来先服务算法,作业所对应的进程调度采用主存中的就绪进程平分 CPU 时间的方式,不计作业对换及其他系统开销。试求各作业(进程)的开始执行时间、完成时间、周转时间。

作业	提交时间	运行时间	作业长度/KB	开始执行时间	完成时间	作业周转时间/min
Job1	10:00	25	15			
Job2	10:20	30	60			
Job3	10:20	25	40			
Job4	10:30	15	20			
Job5	10:35	10	30			

24. 有一个具有两道作业的批处理系统,作业调度采用最高响应比调度算法,进程调度采用短进程优先的抢占式调度算法。在下表所示的作业序列中,作业优先数即为进程优先数,优先数越小则优先级越高。试计算作业的平均周转时间。

作业名	到达时间	估计运行时间/min	优先数
A	10:00	40	5
B	10:20	30	3
C	10:30	50	4
D	10:40	20	6

25. 有一个具有两道作业的批处理系统,作业调度采用短作业优先调度算法,进程调度采用以优先数为基础的抢占式调度算法。在下表所示的作业序列中,作业优先数即为进程优先数,优先数越小则优先级越高。

作业名	到达时间	估计运行时间/min	优先数
A	10:00	40	5
B	10:20	30	3
C	10:30	50	4
D	10:50	20	6

(1) 列出所有作业进入主存的时间及结束时间。

(2) 计算作业的平均周转时间。

26. 有一个多道批处理系统,作业调度采用短作业优先调度算法,进程调度采用优先数抢占式调度算法,且优先数越小则优先级越高。如系统拥有一台打印机,采用静态方法进行分配,忽略系统调度开销。现有以下作业序列到达系统:

作业名	到达时间	估计运行时间/min	打印机需求/台	进程优先数
Job1	14:00	40	1	4
Job2	14:20	30	0	2
Job3	14:30	50	1	3
Job4	14:50	20	0	5
Job5	15:00	10	1	1

试回答：

(1) 按照作业运行结束的次序排序, 哪个作业第一个、第二个、…、最后一个运行结束?

(2) 平均作业周转时间和平均作业带权周转时间是多少?

27. 某多道程序系统供用户使用的主存空间为 100 KB, 磁带机 2 台, 打印机 1 台。采用可变分区主存管理, 采用静态方式分配外部设备, 忽略用户作业 I/O 操作时间。现有作业序列如下:

作业号	进入输入井时间	运行时间/min	主存需求量/KB	磁带机需求/台	打印机需求/台
1	8:00	25	15	1	1
2	8:20	10	30	0	1
3	8:20	20	60	1	0
4	8:30	20	20	1	0
5	8:35	15	10	1	1

作业调度采用 FCFS 策略, 优先分配主存低地址区且不准移动已在主存中的作业, 主存中的各作业平分 CPU 时间。现求:

(1) 作业调度的先后次序;

(2) 全部作业运行结束的时间;

(3) 作业平均周转时间;

(4) 最大作业周转时间。

28. 某多道程序系统采用可变分区主存管理, 供用户使用的主存空间为 200 KB, 磁带机 5 台。采用静态方式分配外部设备, 且不能移动主存中的作业, 进程调度采用 FCFS 算法, 忽略用户作业 I/O 操作时间。现有作业序列如下:

作业号	进入输入井时间	运行时间/min	主存需求量/KB	磁带机需求/台
A	8:30	40	30	3
B	8:50	25	120	1
C	9:00	35	100	2
D	9:05	20	20	3
E	9:10	10	60	1

现求:

(1) FIFO 算法选中作业执行的次序及作业平均周转时间;

(2) SJF 算法选中作业执行的次序及作业平均周转时间。

29. 在第 28 题中,若允许移动已在主存中的作业,其他条件保持不变,现求:

- (1) FIFO 算法选中作业执行的次序及作业平均周转时间;
- (2) SJF 算法选中作业执行的次序及作业平均周转时间。

30. 多道批处理系统中配有一台处理器和两台外部设备(I1 和 I2),用户存储空间为 100 MB。已知系统的作业调度及进程调度采用可抢占的高优先数调度算法,主存采用不允许移动的可变分区分配策略,设备分配遵从动态分配原则。现有 4 个作业同时提交给系统,如下表所示。试求作业平均周转时间。

作业名	优先数	运行时间及顺序/min	主存需求/MB
A	7	CPU:1,I1:2,I2:2	50
B	3	CPU:3,I1:1	10
C	9	CPU:2,I1:3,CPU:2	60
D	4	CPU:4,I1:1	20

31. 设计一个进程定时唤醒队列和定时唤醒处理程序。

- (1) 说明一个等待唤醒进程入队的过程。
- (2) 说明在发生时钟中断时,定时唤醒处理程序的过程。
- (3) 现有进程 P₁ 要求 20 s 后运行,经过 40 s 后再次运行;P₂ 要求 25 s 后运行;P₃ 要求 35 s 后运行,经过 35 s 后再次运行;P₄ 要求 60 s 后运行。试建立相应的进程定时唤醒队列。

32. 一个实时系统有 4 个周期性事件,周期分别为 50 ms、100 ms、300 ms 和 250 ms。若假设其处现分别需要 35 ms、20 ms、10 ms 和 x ms,则此系统的调度可允许的 x 值最大是多少?

33. 在 UNIX 系统中运行以下程序,最多可以产生多少进程? 绘制进程家属树。

```
main() {
    fork(); /* <- PC(程序计数器), 进程 A */
    fork();
    fork();
}
```

34. 假设一个单处理器系统以单道方式处理作业流,作业流中有两道作业,其占用 CPU 计算时间、输入卡片数、打印输出行数如下表所示。

作业号	占用 CPU 计算时间/min	输入卡片数/张	打印输出行数
1	3	100	2 000
2	2	200	600

其中,卡片输入机的运转速度是 1 000 张/min(平均值),打印机的打印速度是 1 000 行/min(平均值),忽略读写盘时间。试计算:

(1) 不采用 SPOOLing 技术,计算这两道作业的总运行时间(从第一个作业输入开始,到最后一个作业输出完毕);

(2) 如果采用 SPOOLing 技术,计算这两道作业的总运行时间。

35. 假如当前绝对时钟为 502。(1) 进程 A、B、C 和 D 请求唤醒时间为 508、514、520 和 538;(2) 进程 E

请求唤醒时间为 525。试设计使用绝对唤醒定时器的优先级队列。

36. 假如当前绝对时钟为 502。(1) 进程 P_1, P_2, P_3 和 P_4 请求唤醒时间分别为 8、14、25 和 10；(2) 插入进程 P_5 ，其请求唤醒时间为 50。试设计使用时间差值的优先级队列。

37. 修改 `set_ltime()` 函数，使得它可以阻塞所调用的进程，直至一个绝对标准时间 `tabs`，即函数的形式为 `set_ltime(tn, tabs)`。

38. 修改 `delay(tdel)` 函数，成为形如 `delay(tn, tdel)` 的函数，它使用一个逻辑递减定时器 `tn`，而不是物理定时器。

39. 写出设置逻辑时钟 `set_ltime(tn, tdel)` 函数的算法流程。

40. UNIX System V 的进程优先数计算公式如下：

$$P - pri = P - cpu/2 + PUSER + P - nice + NZERO$$

其中， $P - pri$ 为优先数（优先数越大，优先级越低）， $PUSER$ 和 $NZERO$ 是基本用户优先级阈值（UNIX 版本所确定的一对常量）， $P - nice$ 是用户可调节的进程优先数偏置值（接近常量）， $P - cpu$ 是进程最近时间段内使用处理器的时间。试述此调度策略实现了“循环多级反馈动态优先级”进程调度算法。

第三章

同步、通信与死锁



并发进程

3.1.1 顺序程序设计

早期,人们引入程序的概念,编写程序并在计算机上运行程序以完成所需要的功能。程序是实现算法的操作(指令)序列,程序执行的顺序性是指其在处理器上的执行是严格有序的,即只有在前一个操作结束后,才能开始后继操作,这称为程序内部的顺序性;如果完成一个任务需要若干不同的程序,则这些程序也按照调用次序严格有序执行,这称为程序外部的顺序性。传统的程序设计方法是顺序程序设计,把程序设计成顺序执行的指令序列,不同程序也按顺序执行。顺序程序设计具有以下一些特性。

(1) 执行的顺序性

一个程序在处理器上是严格按序执行的,每个操作必须在下一个操作开始之前结束。

(2) 环境的封闭性

运行程序独占全机资源,资源的状态只能由此程序本身决定和改变,不受外界因素的影响。

(3) 结果的确定性

程序在执行过程中允许出现中断,但这种中断不会对程序的最终结果产生影响,也就是说程序的执行结果与它的执行速度无关。

(4) 过程的可再现性

程序针对同一个数据集合的执行过程在下一次执行时会重现,即重复执行程序会获得相同的执行过程和计算结果。

顺序程序设计的顺序性、封闭性、确定性和可再现性表明程序及其执行(计算)是一一对应的,为程序的编制和调试带来很大的方便,其缺点是计算机系统效率不高。

3.1.2 进程的并发性

1. 并发程序设计

操作系统中引入并发程序设计技术之后,程序的执行不再是顺序的,一个程序未执行完而另一个程序便已开始执行,程序外部的顺序特性消失,程序与计算不再一一对应。所以,引入进程的概念来描述这种变化。进程的并发性是指一组进程的执行在时间上是重叠的,所谓时间重叠是指一个进程执行第一条指令是在另一个进程执行完最后一条指令之前开始的。例如,两个进程 A 和 B 分别执行操作 a_1, a_2, a_3 和 b_1, b_2, b_3 。在单处理器上,顺序执行操作序列分别为 a_1, a_2, a_3 和 b_1, b_2, b_3 ,然而,若允许进程交叉执行,如执行操作序列为 $a_1, b_1, a_2, b_2, a_3, b_3$ 或 $a_1, b_1, a_2, b_2, b_3, a_3$ 等,则称进程 A 和 B 的执行是并发的。从宏观上来看,并发性反映一个时间段内有几个进程都处于运行态但运行尚未结束的状态;从微观上来看,任一时刻仅有一个进程的一个操作在处理器上执行。反过来,并发的实质是处理器在几个进程之间的多路复用,并发是对有限物理资源强制行使多用户共享,消除计算机部件之间的互等现象,提高系统资源的利用率。程序的并发执行产生资源共享的需求,从而使程序失去封闭性、顺序性、确定性和可再现性。

现代计算机硬部件能同时进行工作,但怎样才能充分发挥不同硬部件的并行工作能力呢?很重要的一点是取决于程序编制。在图 3.1(a)中,由于程序是按照 `while (TRUE) {input, process, output}` 串行地输入-处理-输出来编制的,所以此程序只能顺序执行,这时系统的效率相当低。如果把求解这个问题的程序分成 3 个部分:

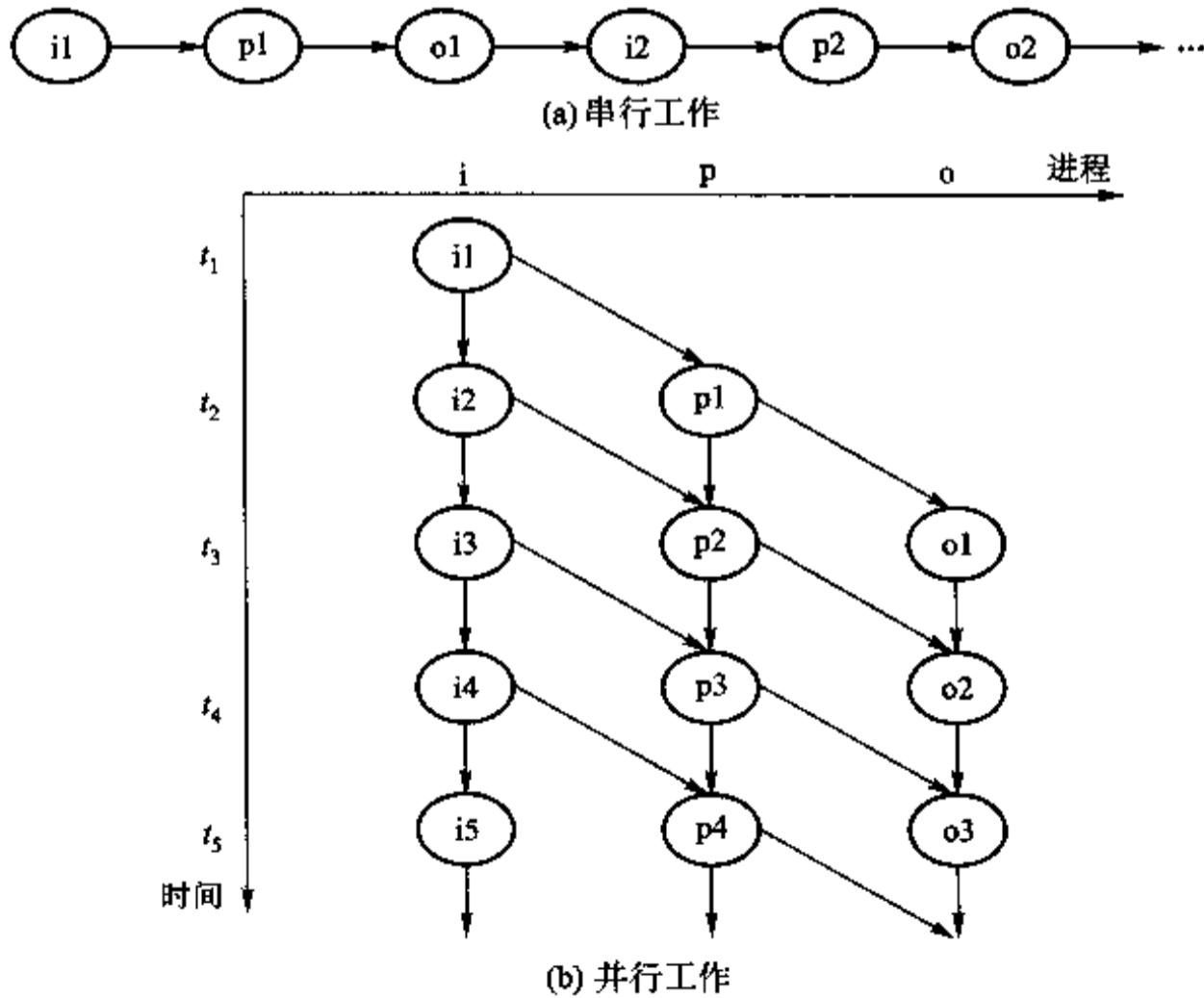


图 3.1 串行工作和并行工作

```

while(TRUE) {input,send}
  while(TRUE) {receive,process,send}
    while(TRUE) {receive,output}

```

每一部分是一个小程序,它们可以并发执行,并会产生制约关系,其中 send 和 receive 操作用于小程序之间通过通信机制解决制约关系,以便协调一致地工作。3 个小程序的功能分别是:

- 小程序 1: 循环执行, 读入字符, 将读入字符送缓冲区 1;
- 小程序 2: 循环执行, 处理缓冲区 1 中的字符, 把计算结果送缓冲区 2;
- 小程序 3: 循环执行, 取出缓冲区 2 中的计算结果并写到磁带上。

从图 3.1(b)可以看出,3 个小程序能并发执行,使得在 t_3 时刻输入 i3、处理 p2 与输出 o1 可以并行工作,在 t_4 、 t_5 等时刻同样可以让 CPU、输入设备和输出设备并行工作。显然,这种程序设计方法能发挥 CPU 和设备的并行工作能力,从而提高计算机系统的效率。一个程序被分成若干可同时执行的小程序的程序设计方法称为并发程序设计 (concurrent programming), 每个小程序及其执行时所处理的数据就组成一个进程。

2. 并发进程的特性

并发进程可能是无关的,也可能是交互的。无关的并发进程是指它们分别在不同的变量集合上操作,一个进程的执行与其他并发进程的进展无关,即一个进程不会改变另一个与其并发执行的进程的变量。然而,交互的并发进程共享某些变量,一个进程的执行可能会影响其他进程的执行结果,交互的并发进程之间具有制约关系。因此,进程的交互必须是有控制的,否则会出现不正确的计算结果。

并发进程的无关性是进程的执行与时间无关的一个充分条件,这一条件在 1966 年首先由 Bernstein 提出,假设

$R(P_i) = \{a_1, a_2, \dots, a_n\}$, 程序 P_i 在执行期间所引用的变量集

$W(P_i) = \{b_1, b_2, \dots, b_m\}$, 程序 P_i 在执行期间所改变的变量集

若两个进程的程序 P_1 和 P_2 能满足 Bernstein 条件,即引用变量集与改变变量集的交集之并为空集:

$$R(P_1) \cap W(P_2) \cup R(P_2) \cap W(P_1) \cup W(P_1) \cap W(P_2) = \emptyset$$

则并发进程的执行与时间无关。上述条件中 $R(P_1) \cap W(P_2) \cup R(P_2) \cap W(P_1) \cup W(P_1) \cap W(P_2) = \emptyset$ 表明一个程序在两次读操作之间,存储单元的数据不会被改变; $W(P_1) \cap W(P_2) = \emptyset$ 表明程序的写操作结果不会丢失。因此只要满足 Bernstein 条件,并发执行的程序就可以保持封闭性和可再现性。例如,有如下 4 条语句:

$$S1: a = x + y; \quad S2: b = z + 1; \quad S3: c = a - b; \quad S4: w = c + 1;$$

于是有: $R(S1) = \{x, y\}$, $R(S2) = \{z\}$, $R(S3) = \{a, b\}$, $R(S4) = \{c\}$; $W(S1) = \{a\}$, $W(S2) = \{b\}$, $W(S3) = \{c\}$, $W(S4) = \{w\}$ 。可见 $S1$ 和 $S2$ 可以并发执行,因为满足 Bernstein 条件,其他语句之间因变量交集之并非空,并发执行可能会产生与时间有关的错误。

采用并发程序设计的好处是:

(1) 若为单处理器系统,可以有效地利用资源,让处理器和设备、设备和设备同时工作,充分发挥硬部件的并行工作能力;

(2) 若为多处理器系统,可让进程在不同的处理器上物理地并行工作,加快计算速度;

(3) 简化程序设计任务,一般来说,编制并发执行的小程序进度快,容易保证正确性。

并发程序设计是在多道程序设计的基础上发展起来的,采用并发程序设计的目的是:充分发挥硬件的并行性,消除处理器和设备的互等现象,提高系统效率。计算机硬部件能并行工作仅具备提高效率的可能性,而并行工作的实现还需要软件技术来发挥,这种软件技术就是并发程序设计。

3. 与时间有关的错误

两个交互的并发进程,其中一个进程对另一个进程的影响往往不可预期,甚至无法再现,这是因为它们执行时的相对速度不可预测,交互进程的速率不仅受到操作系统处理中断的方式及处理器调度策略的影响,而且还受到与其交互的并发进程的影响,甚至受到与其无关的其他进程的影响。由于一个进程的执行速度通常无法为另一个进程所知,对于共享公共变量(资源)的并发进程来说,计算结果往往取决于这一组并发进程执行的相对速度。可能出现各种与时间有关的错误,与时间有关的错误有两种表现形式:一种形式是结果不唯一;另一种形式是永远等待。为了说明与时间有关的错误,现在观察下面的例子。

例 1 (结果不唯一) 飞机票售票问题。假设飞机票预订系统有两个终端,售票进程分别运行函数 T1() 和 T2()。此系统的公共数据区中的一些单元 A_j ($j = 1, 2, 3, \dots$) 分别存放某月某日某次航班的余票数,而 X1 和 X2 表示进程 T1 和 T2 执行时所用的工作单元。

//飞机票售票问题

```
void T1() {
    {按旅客订票要求找到  $A_j$ ;}
    int X1 =  $A_j$ ;
    if(X1 >= 1) {
        X1--;
         $A_j$  = X1;
        {输出一张票};
    }
    else
        {输出信息"票已售完"};
}

void T2() {
    {按旅客订票要求找到  $A_j$ ;}
    int X2 =  $A_j$ ;
    if(X2 >= 1) {
        X2--;
         $A_j$  = X2;
        {输出一张票};
    }
    else
        {输出信息"票已售完"};
}
```

由于售票进程可并发执行,它们运行在同一个计算机系统中,共享同一批票源数据,因此,可能出现如下交叉情况:

T1:X1 = A_j ; // $X1 = m$ T2:X2 = A_j ; // $X2 = m$

T2:X2--; A_j = X2; {输出一张票}; // $A_j = m - 1$ T1:X1--; A_j = X1; {输出一张票}; // $A_j = m - 1$

显然,此时出现把同一张票卖给两位旅客的情况,两位旅客可能各自都买到一张同天同次航班的

机票,可是, A_j 的值实际上只减去1,造成余票数不正确。特别是,当某次航班只有一张余票时,可能把一张票同时出售给两位旅客。

例2 (永远等待)主存管理问题。假定有两个可并发执行的程序 borrow 和 return 分别负责申请和归还主存资源,进程在申请和归还主存资源时调用它们。在算法描述中,X 表示现有空闲主存总量,B 表示申请或归还的主存量。

```
//申请和归还主存资源问题
int X = memory;           //memory 为初始主存容量
void borrow(int B) {
    while(B > X)
        {进程进入等待主存资源队列};
    X = X - B;
    {修改主存分配表,进程获得主存资源};
}
void return(int B) {
    X = X + B;
    {修改主存分配表};
    {释放等待主存资源的进程};
}
```

由于 borrow 和 return 共享代表主存物理资源的临界变量 X,对并发执行不加限制将会导致错误。例如,一个进程调用 borrow 申请主存资源,在执行比较 B 和 X 的大小的指令后,发现 $B > X$,但在执行{进程进入等待主存资源队列}之前,另一个进程调用 return 抢先执行,归还所借全部主存资源;这时,由于前一个进程尚未成为等待者,return 中的{释放等待主存资源的进程}相当于空操作,以后当调用 borrow 的应用进程被置成{等待主存资源}时,可能已经没有其他进程再来归还主存,从而,申请资源的进程处于永远等待状态。

3.1.3 进程的交互:协作和竞争

在多道程序设计系统中,同一时刻可能存在许多进程,它们可以并发或并行地执行。这些进程之间存在两种基本关系:竞争和协作。在引入线程的系统中,多线程之间的制约关系是类似的。

1. 竞争关系

批处理系统中建立多个批处理进程,分时系统中建立多个交互式进程,它们共享一套计算机系统资源,使得原本不存在逻辑关系的诸进程因共享资源而产生交互和制约关系,这是间接制约关系,又称互斥关系,操作系统必须协调进程对共享资源的争用。

由于竞争资源的独立进程之间并不交换信息,一个进程的执行可能影响到竞争资源的其他进程,如果它们要访问同一独占型资源,那么,一个进程通过操作系统分配获得这一资源,另一个进程将不得不等待。在极端的情况下,被阻塞进程永远得不到访问权,从而不能成功地终止。资源竞争会引发两个控制问题:一是死锁(deadlock),一组进程如果都获得部分资源,还想要得到其他进程所占有的资源,最终所有进程将陷入永远等待的状态;二是饥饿(starvation),一个可运行进程由于其他进程总是优先于它,而被调度程序无限期地拖延而不能被执行。在并发系统中,任何时刻都会产生申请资源的事件,需要规定一些策略来决定在什么时候、哪个进程应该获得资源,有些策略看似合理,却可能使某些进程总是得不到服务,但死锁却并未发生。例如,为了打印

```

cobegin
process P0() {
    inside[0] = true;
    turn = 1;
    while(inside[1] && turn == 1);
    {临界区};
    inside[0] = false;
}
process P1() {
    inside[1] = true;
    turn = 0;
    while(inside[0] && turn == 0);
    {临界区};
    inside[1] = false;
}
coend

```

在上面的程序中,利用对 turn 的赋值和 while 语句来限制每次最多只有一个进程进入临界区,当有进程在临界区执行时,不会有另一个进程闯入;进程执行完临界区的程序之后,修改 inside[i] 的状态而使等待进入临界区的进程可以在有限时间内进入。所以,Peterson 算法满足对临界区管理的三个原则。由于 while 语句中的判别条件是“inside[i] && turn = 0(或 1)”,因此,任意进程进入临界区的条件是对方不在临界区或对方不想进入临界区,于是,任何进程均可多次进入临界区。

3.2.4 实现临界区管理的硬件设施

在单处理器计算机系统中,并发进程不会同时执行,只会交替地执行。为了保证互斥性,仅需保证进程不被中断。下面是一些硬件设施,可用来实现对临界区的管理。

1. 关中断

实现互斥的最简单方法是在进程进入临界区时关中断,在进程退出临界区时开中断。中断被关掉后,时钟中断也被屏蔽,因为进程上下文切换都是由中断事件引起的,这样进程的执行再也不会被打断,因而,采用关中断、开中断的方法就可确保并发进程互斥地进入临界区。关中断的方法简单、有效,对操作系统自身很有用,可在更新共享变量或列表的几条指令期间禁止中断,但不适合作为通用的互斥机制,关中断的时间过长会影响性能和系统效率;不适用于多处理器计算机系统,因为一个处理器关中断,并不能防止进程在其他处理器上执行相同的临界段代码;若将这项权力赋予用户也存在危险,如果用户未开中断,则系统可能因此而终止。

2. 测试并建立指令

另一种方法是使用硬件所提供的“测试并建立”机器指令 TS(Test and Set),可把这条指令看做函数,它有布尔型参数 x 和返回条件码,当 TS(&x) 测到 x 值为 true 时则置 x 为 false,且根据所测试到的 x 值形成条件码。下面给出 TS 指令的处理过程。

```

bool TS(bool &x) {
    if(x) {
        x = false;
        return true;
    }
}

```

A_j 。并发进程中与共享变量有关的程序段称为“临界区”(critical section)。共享变量所代表的资源称为“临界资源”(critical resource),如独占型硬件是临界资源,被共享的数据结构和文件也是临界资源。当多个并发进程访问临界资源时,结果依赖于它们执行的相对速度,便称出现了竞争条件(race condition)。显而易见,临界区必须以一种相对于其他进程而言互相排斥的方式执行。例 1 中的临界区表述如下:

```
//进程 T1 的临界区           //进程 T2 的临界区
X1 = Aj;
if(X1 >= 1) {
    X1--;
    Aj = X1;
}
|
X2 = Aj;
if(X2 >= 1) {
    X2--;
    Aj = X2;
}
```

与同一共享变量有关的临界区分散在各个并发进程的程序段中,而进程的执行速度不可预知,如果能够保证一个进程在临界区执行时,不让另一个进程进入相同的临界区,即各进程对共享变量的访问是互斥的,那么,就不会引发与时间有关的错误。

临界区的概念由 Dijkstra 于 1965 年首先提出。为了正确而有效地使用临界资源,共享变量的并发进程应遵守临界区调度的三个原则:

- (1) 一次至多有一个进程进入临界区内执行;
- (2) 如果已有进程在临界区中,试图进入此临界区的其他进程应等待;
- (3) 进入临界区内的进程应在有限时间内退出,以便让等待队列中的一个进程进入。

可把临界区的调度原则总结成 3 句话:互斥使用,有空让进;忙则等待,有限等待;择一而入,算法可行。其中,“算法可行”是指:不能因为所选的调度策略造成进程饥饿甚至死锁。

3.2.2 临界区管理的尝试

考虑并发进程在单处理器或共享主存的多处理器计算机系统上执行,先讨论采用软件方法实现临界区管理,通常假设具有存储器访问级的基本互斥性,对主存中的同一个单元同时访问时,必定由存储器进行仲裁使其串行化,此外,硬件、操作系统或语言未提供任何支持。临界区的管理可采用标志方式,即用标志来表示哪个进程可进入临界区。然而,如何使用标志,仍是值得探讨的问题。下列程序是第一种尝试,对进程 P1 和 P2 分别用标志 inside1 和 inside2 与其相对应,当进程在它的临界区内时其值为“真”(true),进程不在临界区内时其值为“假”(false);P1(P2)进入它的临界区前先测试 inside2(inside1),确保当前无进程在临界区中,然后把 inside1 (inside2)置成 true,封锁 P2(P1)进入临界区,直至 P1(P2)执行完毕退出临界区,再将相应的标志置成 false。

```
//临界区管理尝试 1
bool inside1 = false;          //P1 不在其临界区内
bool inside2 = false;          //P2 不在其临界区内
```

```

cobegin                                //cobegin 和 coend 表示括号中的进程是一组并发进程
process P1() {
    while(inside2);      //等待
    inside1 = true;
    {临界区};
    inside1 = false;
}
process P2() {
    while(inside1);      //等待
    inside2 = true;
    {临界区};
    inside2 = false;
}
coend

```

但是,这种管理是不正确的,原因是在进程 P1(P2)测试 inside2 (inside1) 及随后置 inside1 (inside2) 为 true 之间,P2(P1)可能发现 inside1 (inside2) 的值为 false,于是它将置 inside2 (inside1) 为 true,并且与进程 P1(P2)同时进入临界区。

第二种尝试对第一种方案作一定的修正:延迟进程 P1(P2)对 inside2 (inside1) 的测试,先置 inside1 (inside2) 为 true,用以封锁 P2(P1),修正后的程序如下所示。遗憾的是,它也是无效的,有可能每个进程都把自己的标志置成 true,从而出现死循环,这时没有进程能在有限的时间内进入临界区,造成永远等待的现象发生。

```

//临界区管理尝试 2
bool inside1 = false;          //P1 不在其临界区内
bool inside2 = false;          //P2 不在其临界区内
cobegin
process P1() {
    inside1 = true;
    while(inside2);      //等待
    {临界区};
    inside1 = false;
}
process P2() {
    inside2 = true;
    while(inside1);      //等待
    {临界区};
    inside2 = false;
}
coend

```

3.2.3 实现临界区管理的软件算法

1981 年,G.L.Perterson 提出一个简单且巧妙的算法来解决进程互斥进入临界区的问题,此方法为每个进程设置标志,当标志值为 false 时表示此进程要求进入临界区,另外再设置一个指示器 turn 以指示可以由哪个进程进入临界区,当 turn = i 时则可由进程 Pi 进入临界区。

```

//Peterson 算法
bool inside[2];
inside[0] = false;inside[1] = false;
enum {0,1} turn;

```

```

cobegin
process P0() {
    inside[0] = true;
    turn = 1;
    while(inside[1] & & turn == 1);
    {临界区};
    inside[0] = false;
}
process P1() {
    inside[1] = true;
    turn = 0;
    while(inside[0] & & turn == 0);
    {临界区};
    inside[1] = false;
}
coend

```

在上面的程序中,利用对 turn 的赋值和 while 语句来限制每次最多只有一个进程进入临界区,当有进程在临界区执行时,不会有另一个进程闯入;进程执行完临界区的程序之后,修改 inside[i] 的状态而使等待进入临界区的进程可以在有限时间内进入。所以,Peterson 算法满足对临界区管理的三个原则。由于 while 语句中的判别条件是“inside[i] & & turn = 0(或 1)”,因此,任意进程进入临界区的条件是对方不在临界区或对方不想进入临界区,于是,任何进程均可多次进入临界区。

3.2.4 实现临界区管理的硬件设施

在单处理器计算机系统中,并发进程不会同时执行,只会交替地执行。为了保证互斥性,仅需保证进程不被中断。下面是一些硬件设施,可用来实现对临界区的管理。

1. 关中断

实现互斥的最简单方法是在进程进入临界区时关中断,在进程退出临界区时开中断。中断被关掉后,时钟中断也被屏蔽,因为进程上下文切换都是由中断事件引起的,这样进程的执行再也不会被打断,因而,采用关中断、开中断的方法就可确保并发进程互斥地进入临界区。关中断的方法简单、有效,对操作系统自身很有用,可在更新共享变量或列表的几条指令期间禁止中断,但不适合作为通用的互斥机制,关中断的时间过长会影响性能和系统效率;不适用于多处理器计算机系统,因为一个处理器关中断,并不能防止进程在其他处理器上执行相同的临界段代码;若将这项权力赋予用户也存在危险,如果用户未开中断,则系统可能因此而终止。

2. 测试并建立指令

另一种方法是使用硬件所提供的“测试并建立”机器指令 TS(Test and Set),可把这条指令看做函数,它有布尔型参数 x 和返回条件码,当 TS(&x) 测到 x 值为 true 时则置 x 为 false,且根据所测试到的 x 值形成条件码。下面给出 TS 指令的处理过程。

```

bool TS(bool &x) {
    if(x) {
        x = false;
        return true;
    }
}

```

```

    }
    else
        return false;
}

```

用 TS 指令管理临界区时, 可把一个临界区与一个布尔型变量 s 相关联, 由于变量 s 代表临界资源的状态, 把它看成一把锁, s 的初值置为 true, 表示没有进程在临界区内, 资源可用, 系统利用 TS 指令实现临界区的上锁和开锁原语操作。在进入临界区之前, 首先用 TS 指令测试 s, 如果没有进程在临界区内, 则可以进入, 否则必须循环测试直至 s 值为 true; 当进程退出临界区时, 把 s 值置为 true。由于 TS 指令是不可分割指令, 在测试和形成条件码之间不可能有其他进程测试变量值, 从而保证临界区管理的正确性。

```

//TS 指令实现进程互斥
bool s = true;
cobegin
process Pi() { //i=1,2,...,n
    while(! TS(s)); //上锁
    {临界区};
    s = true; //开锁
}
coend

```

3. 对换指令

对换指令的功能是交换两个字的内容, 其处理过程描述如下:

```

void SWAP(bool &a, bool &b) {
    bool temp = a;
    a = b;
    b = temp;
}

```

在 Intel x86 中, 对换指令称为 XCHG, 使用它可以简单而有效地实现互斥, 方法是为每个临界区设置布尔型锁变量, 如 lock, 当其值为 false 时表示无进程在临界区内。

```

//对换指令实现进程互斥
bool lock = false;
cobegin
process Pi() { //i=1,2,...,n
    bool keyi = true;
    do {
        SWAP(keyi, lock);

```

```

    | while(keyi);           //上锁
    {临界区};
    SWAP(keyi, lock);      //开锁
}
coend

```

信号量与 PV 操作

3.3.1 同步和同步机制

著名的生产者 - 消费者问题(producer-consumer problem)是计算机操作系统中并发进程内在关系的一种抽象,是典型的进程同步问题。在操作系统中,生产者进程可以是计算进程、发送进程;而消费者进程可以是打印进程、接收进程等,解决好生产者 - 消费者问题就解决了一类并发进程的同步问题。

生产者-消费者问题表述如下：有 n 个生产者和 m 个消费者，连接在具有 k 个单位缓冲区的有界环状缓冲上，故又称有界缓冲问题。其中， p_i 和 c_j 都是并发进程，只要缓冲区未满，生产者进程 p_i 所生产的产品就可投入缓冲区；类似地，只要缓冲区非空，消费者进程 c_j 就可从缓冲区取走并消耗产品。

//生产者-消费者问题的程序描述

```

int k;
typedef anyitem item; //item 类型
item buffer[k];
int in = 0, out = 0, counter = 0;
process producer(void) {
    while(true) { //无限循环
        {produce an item in nextp}; //生产一个产品
        if(counter == k) //缓冲区满时,生产者睡眠
            sleep(producer);
        buffer[in] = nextp; //将一个产品放入缓冲区
        in = (in + 1) % k; //指针推进
        counter++; //缓冲内产品数加 1
        if(counter == 1) //缓冲区空了,加进一件产品
            wakeup(consumer); //并唤醒消费者
    }
}

```

```

process consumer(void) {
    while(true) {                                //无限循环
        if(counter == 0)                         //缓冲区为空,消费者睡眠
            sleep(consumer);
        nextc = buffer[out];                     //取一个产品到 nextc
        out = (out + 1) % k;                      //指针推进
        counter--;                               //取走一个产品,计数器减 1
        if(counter == k - 1)                      //缓冲区满,取走一件产品
            wakeup(producer);                   //并唤醒生产者
        {consume the item in nextc};           //消耗产品
    }
}

```

一般的高级语言都有 sleep() 和 wakeup() 函数,从上面的程序代码可以看出,算法是正确的,两组进程顺序执行的结果也正确。但若进程并发执行,就会出现错误的计算结果,出错的根源在于进程之间共享变量 counter,对 counter 的访问未加限制。

生产者进程和消费者进程对 counter 的交替操作会使其结果不唯一,例如,counter 的当前值为 8,如果生产者生产一件产品并投入缓冲区,拟执行 counter 加 1 操作;同时消费者获取一个产品消费,拟执行 counter 减 1 操作;假如两者交替执行加或减 1 操作,取决于其执行速度,counter 的值可能是 9,也可能是 7,但其正确值应为 8。

更为严重的是,生产者进程和消费者进程的交替执行会导致进程永远等待,造成系统死锁。假定消费者读取 counter 值时发现它为 0,此时调度程序暂停消费者进程让生产者进程运行,生产者加入一个产品,将 counter 值加 1,它想当然地推想由于 counter 值刚才为 0,所以,此时消费者一定是在睡眠,于是生产者调用 wakeup 函数来唤醒消费者;遗憾的是,消费者尚未睡眠时,唤醒信号被丢失,当消费者下次运行时,因已测到 counter 值为 0,于是去睡眠,这样生产者迟早会填满缓冲区,然后去睡眠,造成所有进程都永远处于睡眠状态。

出现不正确的结果并不是因为并发进程共享缓冲区,而是因为它们访问缓冲区的速度不匹配,或者说 p_i, c_j 的相对速度不协调,需要调整并发进程的推进速度,并发进程间的这种制约关系称为进程同步,交互的并发进程之间通过交换信号或消息来达到调整相互速度、保证进程协调运行的目的。

操作系统实现进程同步的机制称为同步机制,它通常由同步原语组成。不同的同步机制采用不同的同步方法,迄今已设计出多种同步机制,本书将介绍最常用的同步机制:信号量及 PV 操作,管程和消息传递。

3.3.2 信号量与 PV 操作

前面介绍各种方法解决进程互斥,软件算法太过复杂,效率低下;硬件方法虽然简单、有效,

但采用忙式等待,白白浪费 CPU 时间;将测试能否进入临界区的责任推卸给各个竞争的进程,会削弱系统的可靠性,加重用户的编程负担,而且这些方案只能解决进程竞争却不能解决进程协作问题。

1965 年,荷兰计算机科学家 Edsger W. Dijkstra 提出新的同步工具—信号量和 P、V 操作,他将交通管制中多种颜色的信号灯管理方法引入操作系统,让两个或多个进程通过特殊变量展开交互。一个进程在某一关键点上被迫停止执行直至接收到对应的特殊变量值,通过这一措施,任何复杂的进程交互要求均可得到满足,这种特殊变量就是信号量(semaphore)。为了通过信号量传送信号,进程可通过 P、V 两个特殊操作来发送和接收信号,如果协作进程的相应信号仍未送到,则进程被挂起直至信号到达为止。

在操作系统中,用信号量表示物理资源的实体,是一个与队列有关的整型变量。具体实现时,信号量是一种变量类型,用一个结构型数据结构表示,有两个分量:一个是信号量的值,另一个是信号量队列的指针。信号量在操作系统中主要用于封锁临界区、进程同步及维护资源计数。

除了赋初值之外,信号量仅能由同步原语 P 和 V 对其进行操作,不存在其他方法可以检查和操作信号量,PV 操作的不可分割性确保执行时的原子性及信号量值的完整性。Dijkstra 发明信号量操作原语:P 和 V 操作(荷兰语中“检测”(Proberen)和“增量”(Verhogen)的首字母),常用的符号还有 up 和 down 等。本书采用 Dijkstra 最早论文中使用的符号 P 和 V。利用信号量和 P、V 操作既可解决并发进程的竞争问题,又可解决并发进程的协作问题。

信号量按其用途可分为两种:(1)公用信号量,联系一组并发进程,相关的进程均可在此信号量上执行 P、V 操作,初值往往为 1,用于实现进程互斥;(2)私有信号量,联系一组并发进程,仅允许此信号量所拥有的进程执行 P 操作,而其他相关进程可在其上施行 V 操作,初值往往为 0 或正整数,多用于并发进程同步。信号量按其取值可分为两种:(1)二元信号量,仅允许取值为 0 或 1,主要用于解决进程互斥问题;(2)一般信号量,又称计数信号量,允许取大于 1 的整型值,主要用于解决进程同步问题。

1. 一般信号量

设 s 为一个结构型数据结构,其中 value 为整型变量,系统初始化时为其赋值;list 是等待使用此类资源的进程队列的头指针,初始状态为空队列,P 和 V 原语操作的描述如下:

(1) P(s):将信号量 value 值减 1,若结果小于 0,则执行 P 操作的进程被阻塞,排入与 s 信号量有关的 list 所指队列中;若结果大于等于 0,则执行 P 操作的进程继续执行。

(2) V(s):将信号量 value 值加 1,若结果不大于 0,则执行 V 操作的进程从信号量 s 有关的 list 所指队列中释放一个进程,使其转换为就绪态,自己则继续执行;若结果大于 0,则执行 V 操作的进程继续执行。

结构型信号量和 P、V 操作定义为如下的数据结构和不可中断过程。

```
typedef struct semaphore {
    int value;                      //信号量值
    struct pcb * list;              //信号量队列指针
```

```

    ;
void P(semaphore &s) {
    s.value--;                                //信号量值减 1
    if(s.value<0)                            //若信号量值小于 0,执行 P 操作的进程
        W(s.list);                          //调用 W(s.list)阻塞自己,被置成等待信号量 s 状态
    }                                            //并移入 s 信号量队列,然后转向进程调度
void V(semaphore &s) {
    s.value++;                                //信号量值加 1
    if(s.value<=0)
        R(s.list);                          //从信号量 s 队列中释放一个等待信号量 s 的进程
    }                                            //并转换成就绪态,自己则继续执行
}

```

其中 W(s.list)和 R(s.list)是操作系统的基本系统调用,执行前者表示把调用过程的进程置成等待信号量 s 的状态,并移入 s 信号量队列,同时释放 CPU,转向进程调度;执行后者表示释放一个等待信号量 s 的进程,从 s 信号量队列中移出一个进程,转换成就绪态并移入就绪队列,执行 V 操作的进程继续执行,或转向进程调度。需要注意的是,当有多个进程等待同一个信号量时,它们被排成等待队列,由信号量指针指向队首进程的 PCB,而队列中的其他进程则通过其 PCB 中的指针项进行链接,队尾进程 PCB 的链接指针为 0。

进程从队列中移出的次序应遵循的公平策略是 FCFS 算法,被阻塞时间最长久的进程最先从队列中被释放,此策略确保进程不会饥饿,信号量 s 的初值可定义为 0、1 或其他整型数,在系统初始化时确定。从信号量和 P、V 操作的定义可获得如下推论。

推论 1 若信号量 s.value 为正值,此值等于在封锁进程之前对信号量 s 可施行的 P 操作数,亦即 s 所代表的实际可用的物理资源数。

推论 2 若信号量 s.value 为负值,其绝对值等于登记排列在 s 信号量队列之中等待的进程个数,即恰好等于对信号量 s 实施 P 操作而被封锁并进入信号量 s 等待队列的进程数。

推论 3 P 操作通常意味着请求一个资源,V 操作意味着释放一个资源,在一定的条件下,P 操作代表挂起进程的操作,而 V 操作代表唤醒被挂起进程的操作。

2. 二元信号量

设 s 为一个结构型数据结构,其中分量 value 仅能取值 0 或 1,分量 list 为等待信号量进程队列的头指针,若把二元信号量的 P、V 操作记为 BP 和 BV,其定义如下。

```

typedef struct binary_semaphore {
    int value;                                //value 取值 0 或 1
    struct pcb * list;
};

void BP(binary_semaphore &s) {
    if(s.value==1)

```

```

    s.value = 0;
else
    W(s.list);
}

void BV(binary_semaphore &s) {
    if(s.list is empty())
        s.value = 1;
    else
        R(s.list);
}

```

虽然二元信号量仅能取值 0 或 1,但可以证明它有着与其他结构型信号量一样的表达能力。

3.3.3 信号量实现互斥

信号量和 P、V 操作可用来解决进程互斥问题,与 TS 指令相比较,P、V 操作也用测试信号量的方法来决定是否进入临界区,但不同的是,P、V 操作只对信号量测试一次,而 TS 指令则必须反复测试。用信号量和 P、V 操作管理并发进程互斥进入临界区的一般形式为:

```

semaphore mutex;
mutex = 1;
cobegin
process Pi() //i=1,2,...,n
P(mutex);
{临界区};
V(mutex);
}
coend

```

当有进程在临界区中时,mutex 的值为 0 或负值;否则 mutex 值为 1,因为只有一个进程可用 P 操作把 mutex 减至 0,故可保证互斥操作,这时试图进入临界区的其他进程会因执行 P(mutex)而被迫等待。mutex 的取值范围是 $1 \sim -(n - 1)$,表明有一个进程在临界区内执行,最多有 $n - 1$ 个进程在信号量队列中等待。

3.3.4 信号量解决 5 位哲学家吃通心面问题

下面讨论使用信号量和 P、V 操作解决操作系统中经典的 5 位哲学家吃通心面问题(Dijkstra,1968 年)。如图 3.2 所示,有 5 位哲学家围坐在一张圆桌旁,桌子中央放有一盘通心面,每人面前有一只空盘子,每两人之间放一把叉子;每位哲学家思考、饥饿,然后吃通心面;为了吃面,哲学家必须获得两把叉子,且每人只能直接从紧邻自己的左边或右边去取叉子。

在这道题目中，每把叉子都必须互斥使用，因此，应为每把叉子设置互斥信号量 $\text{fork}[i]$ ($i = 0, 1, 2, 3, 4$)，其初值均为 1，当一位哲学家吃通心面之前必须执行两个 P 操作，获得自己左边和右边的两把叉子；在吃完通心面后必须执行两个 V 操作，放下两把叉子。

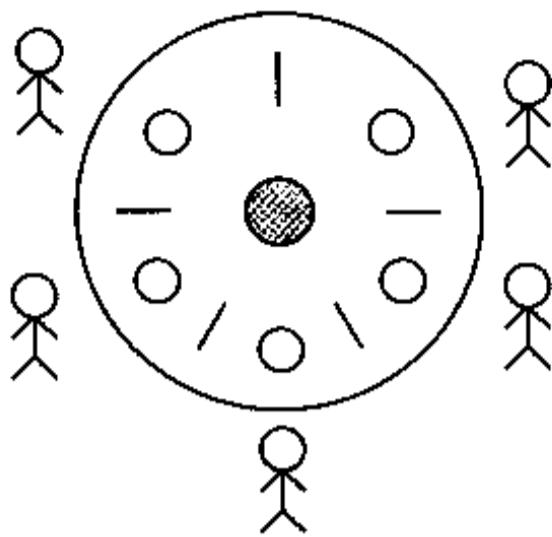


图 3.2 5 位哲学家吃通心面问题

```
//信号量解决哲学家吃通心面问题
semaphore fork[5];
for(int i=0;i<5;i++)
    fork[i]=1;
cobegin
process philosopher_i() { //i= 0,1,2,3,4
    while(true) {
        think();
        P(fork[i]);
        P(fork[(i+1)%5]);
        eat();
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
coend
```

在上述解法中，如果 5 位哲学家同时拿起右边（或左边）的叉子，当每位哲学家企图再拿起其左边（或右边）的叉子时，将出现死锁。有若干种方法可以避免这类死锁。

- (1) 至多允许 4 位哲学家同时吃面；
- (2) 奇数号哲学家先取左边的叉子，然后再取右边的叉子；偶数号哲学家先取右边的叉子，然后再取左边的叉子；
- (3) 每位哲学家拿到手边的两把叉子才开始吃面，否则，连一把叉子也不取。

3.3.5 信号量解决生产者 - 消费者问题

信号量和 P、V 操作不仅可以解决进程互斥问题，而且是实现进程同步的有力工具。在协作进程之间，一个进程的执行依赖于协作进程的信号或消息，在尚未得到来自协作进程的信号或消息时则等待，直至信号或消息到达时才被唤醒。

生产者 - 消费者问题是典型的进程同步问题，这些进程必须按照一定的生产率和消费率来访问共享缓冲区，用 P、V 操作解决生产者和消费者共享单缓冲区的问题，可设置两个信号量 empty 和 full，其初值分别为 1 和 0，empty 指示能否向缓冲区放入产品，full 指示能否从缓冲区取出产品。

//单缓冲区生产者 - 消费者问题

```

int B;
semaphore empty;           //可用的空缓冲区数
semaphore full;            //缓冲区内可用的产品数
empty=1;                   //缓冲区内允许放入一件产品
full=0;                    //缓冲区内没有产品
cobegin
process producer(){
    while(true) {
        produce();
        P(empty);
        append() to B;
        V(full);
    }
}
process consumer(){
    while(true) {
        P(full);
        take() from B;
        V(empty);
        consume();
    }
}
coend

```

需要注意的是,如果 P、V 操作使用不当,仍会出现与时间有关的错误。例如,有 m 个生产者和 n 个消费者,它们共享可存放 k 件产品的缓冲区,为了使其协调工作,必须使用一个信号量 mutex(初值为 1),以限制生产者和消费者互斥地对缓冲区进行存取,另用两个信号量 empty(初值为 k)和 full(初值为 0),以保证生产者不向已满的缓冲区中放入产品,消费者不从空缓冲区中取产品。

// m 个生产者和 n 个消费者共享 k 件产品缓冲区的问题

```

item B[k];
semaphore empty; empty=k;           //可用的空缓冲区数
semaphore full; full=0;              //缓冲区内可用的产品数
semaphore mutex; mutex=1;            //互斥信号量
int in=0;                            //放入缓冲区指针
int out=0;                           //取出缓冲区指针
cobegin
process producer_i(){
    while(true) {
        produce();
        P(empty);
        P(mutex);
        append to B[in];
    }
}
process consumer_j(){
    while(true) {
        P(full);
        P(mutex);
        take() from B[out];
        out=(out+1)%k;
    }
}
coend

```

```

    in = (in + 1) % k;           V(mutex);
    V(mutex);                  V(empty);
    V(full);                  consume();
}

}

coend

```

程序中的 P(mutex) 和 V(mutex) 必须成对出现, 夹在两者之间的代码段是进程的临界区; 施加于信号量 empty 和 full 上的 P、V 操作也必须成对出现, 但分别位于不同的程序中。在这个问题中, P 操作的次序是很重要的, 如果把生产者进程中的两个 P 操作交换次序, 那么, 当缓冲区中存满 k 件产品 (empty = 0, mutex = 1, full = k) 时, 生产者又生产一件产品, 在它欲向缓冲区存放时, 将在 P(empty) 上等待 (注意, 现在 empty = -1), 由于此时 mutex = 0, 它已经占有缓冲区, 这时消费者欲取产品时将停留在 P(mutex) 上而得不到使用缓冲区的权力。这就导致生产者等待消费者取走产品, 而消费者却在等待生产者释放缓冲区的占有权, 这种相互之间的等待永远不可能结束。所以, 在使用信号量和 P、V 操作实现进程同步时, 特别要当心 P 操作的次序, 而 V 操作的次序无关紧要。一般来说, 用于互斥的信号量上的 P 操作总是在后面执行。

3.3.6 信号量解决读者 - 写者问题

读者 - 写者问题 (reader writer problem) (Courtois, Heymans, Parnas, 1971 年) 也是一个经典的并发程序设计问题。有两组并发进程: 读者和写者, 共享文件 F, 要求:

- (1) 允许多个读者同时对文件执行读操作;
- (2) 只允许一个写者对文件执行写操作;
- (3) 任何写者在完成写操作前不允许其他读者或写者工作;
- (4) 写者在执行写操作前, 应让已有的写者和读者全部退出。

单纯使用信号量不能解决此问题, 必须引入计数器 readcount 对读进程计数, mutex 是用于对计数器 readcount 操作的互斥信号量, writeblock 表示是否允许写的信号量。

//信号量解决读者 - 写者问题

```

int readcount = 0;           //读进程计数
semaphore writeblock, mutex;
writeblock = 1; mutex = 1;

cobegin
process reader_i() {
    P(mutex);
    readcount++;
    if(readcount == 1)
        P(writeblock);
}
process writer_j() {
    P(writeblock);
    {写文件};
    V(writeblock);
}

```

```

V(mutex);
{读文件};
P(mutex);
readcount--;
if(readcount==0)
    V(writeblock);
V(mutex);
|
coend

```

此解法中,读者是优先的,当存在读者时,写者将被延迟,且只要有一个读者活跃,随后而来的读者都将被允许访问文件,从而导致写者长时间等待,并有可能出现写者饥饿的现象。增加信号量并修改此程序可得到写者具有优先权的解决方案,确保当一个写进程声明想写时,不允许新的读者访问共享文件。

为了有效地解决读者-写者问题,有的操作系统专门引进读者-写者锁,允许多名读者同时以只读方式存取有锁保护的对象;或一位写者以写方式存取有锁保护的对象。当一名或多名读者已经上锁后,此时形成读锁,写者将不能访问有读锁保护的对象;当锁被请求者用于写操作时,形成写锁,其他进程的读写操作必须等待。

3.3.7 信号量解决理发师问题

另一个经典的进程同步问题是 1968 年由 Dijkstra 提出的睡眠理发师问题(sleepy barber problem)。理发店里有一位理发师、 n 把理发椅和 n 把供等候理发的顾客休憩的椅子;如果没有顾客,理发师便在理发椅上睡觉,当有顾客到来时,他唤醒理发师;如果理发师正在理发时又有新的顾客来到,那么如果还有空椅子,顾客就坐下来等待,否则就离开理发店。

给出的解法中引入 3 个信号量和一个控制变量:控制变量 waiting 记录等候理发的顾客坐的椅子数,初值为 0;信号量 customers 记录等候理发的顾客数,并用于阻塞理发师进程,其初值为 0;信号量 barbers 记录正在等候顾客的理发师数,并用于阻塞顾客进程,其初值为 0;信号量 mutex 用于互斥,其初值为 1。

```

//信号量解决理发师问题
int waiting = 0;           //等候理发的顾客坐的椅子数
int CHAIRS = N;             //为顾客准备的椅子数
semaphore customers, barbers, mutex;
customers = 0; barbers = 0; mutex = 1;
cobegin
process barber() +
    while(true) {

```

```

P(customers);           //判断是否有顾客,若无顾客,理发师睡眠
P(mutex);               //若有顾客,进入临界区
waiting--;              //等候顾客数减1
V(barbers);             //理发师准备为顾客理发
V(mutex);               //退出临界区
cut_hair();              //理发师正在理发(非临界区)
}

process customer_i() {
    P(mutex);           //进入临界区
    if(waiting<CHAIRS) {
        waiting++;       //等候顾客数加1
        V(customers);     //唤醒理发师
        V(mutex);         //退出临界区
        P(barbers);       //理发师忙,顾客坐着等待
        get_haircut();     //否则,顾客可以理发
    }
    else
        V(mutex);         //入满了,顾客离开
}
coend

```

管 程

3.4.1 管程和条件变量

使用信号量和 P、V 操作实现进程同步时,对共享资源的管理分散于各个进程中,进程能够直接对共享变量进行处理,不利于系统对临界资源的管理,难以防止进程有意或无意的违法同步操作,容易造成程序设计错误。在进程共享主存的前提下,如果能集中和封装针对一个共享资源的所有访问,包括所需的同步操作,即把相关的共享变量及其操作集中在一起统一控制和管理,就可以方便地管理和使用共享资源,使并发进程之间的相互作用更为清晰,更易于编写正确的并发程序。

在 1974 年和 1977 年,Hoare 和 Brinch Hansen 根据抽象数据类型的原理提出新的同步机制——管程(monitor)。其基本思路是:把分散在各进程中的临界区集中起来管理,并把共享资源用数据结构抽象地表示出来,由于临界区是访问共享资源的代码段,建立一个“秘书”程序管理

到来的访问。“秘书”每次只让一个进程来访,这样既便于对共享资源进行管理,又能实现互斥访问。在后来的实现中,“秘书”程序更名为管程。采用这种方法,对共享资源的管理可借助数据结构及其上所实施操作的若干进程来进行;对共享资源的申请和释放通过进程在数据结构上的操作来实现。代表共享资源的数据结构及并发进程在其上执行的一组过程就构成管程,管程被请求和释放资源的进程所调用,管程实质上是把临界区集中到抽象数据类型模板中,可作为程序设计语言的一种结构成分。对于同步问题的解决,管程和信号量具有同等的表达能力。

管程有以下属性,因而,调用管程的过程时要有一定的限制。

(1) 共享性

管程中的移出过程可被所有要调用管程的过程的进程所共享。

(2) 安全性

管程的局部变量只能由此管程的过程访问,不允许进程或其他管程来直接访问,一个管程的过程也不应访问任何非局部于它的变量。

(3) 互斥性

在任一时刻,共享资源的进程可以访问管程中的管理此资源的过程,但最多只有一个调用者能够真正地进入管程,其他调用者必须等待直至管程可用。

可见,管程是由局部于自己的若干公共变量及其说明和所有访问这些公共变量的过程所组成的软件模块,它提供一种互斥机制,进程可互斥地调用这些过程。管程把分散在各个进程中互斥地访问公共变量的那些临界区集中在一起,提供对它们的保护。由于共享变量每次只能被一个进程所访问,把代表共享资源状态的共享变量放置在管程中,那么,管程就可以控制共享资源的使用。管程可作为程序设计语言的一个成分,采用管程作为同步机制便于用高级语言来编写程序,也便于程序正确性验证。管程的概念和机制已在 Pascal_plus、Modula-2、Modula-3 和 Java 等语言中实现。

每个管程都有一个名字以供标识,一般的形式表示为:

```
type 管程名 = monitor {
    局部变量说明;
    条件变量说明;
    初始化语句;
    define 管程内定义的、管程外可调用的过程或函数名列表;
    use 管程外定义的、管程内将调用的过程或函数名列表;
    过程名/函数名(形式参数表) {
        <过程/函数体>;
    }
    ...
    过程名/函数名(形式参数表) {
        <过程/函数体>;
    }
}
```

管程的结构如图 3.3 所示。

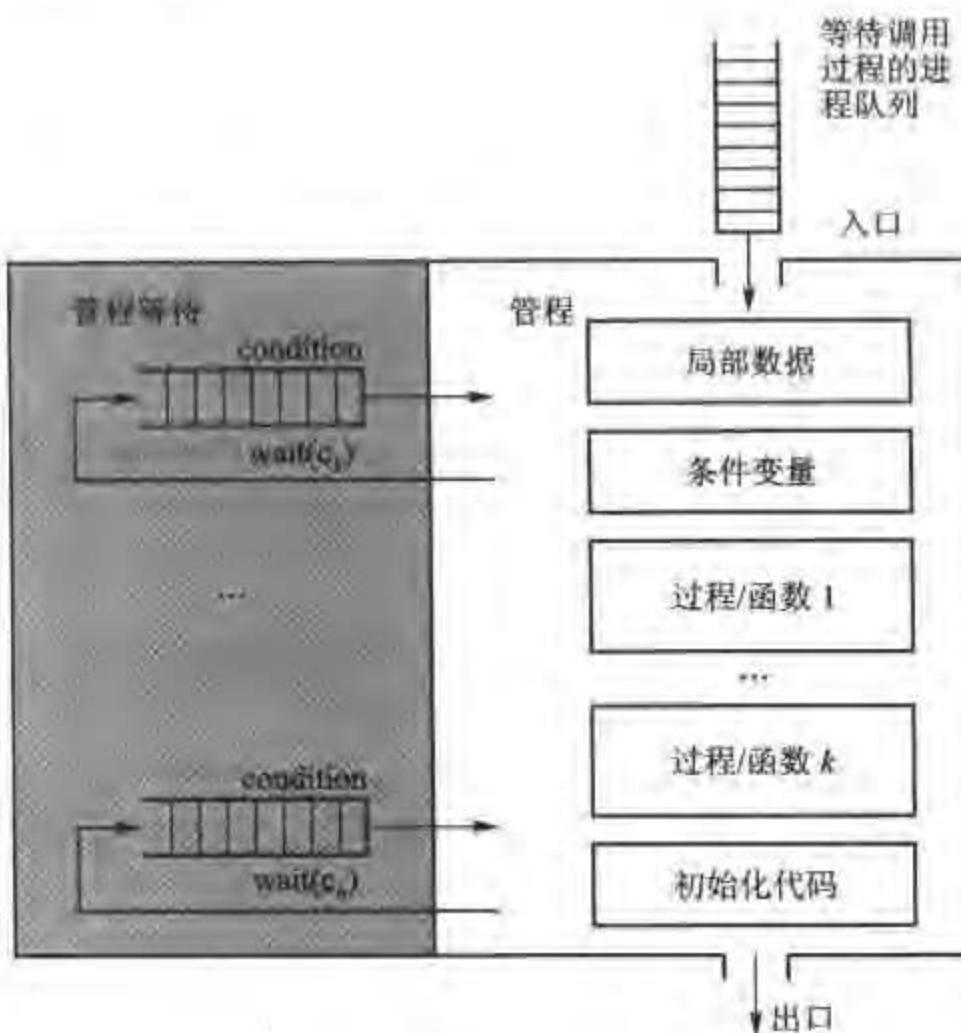


图 3.3 管程结构示意图

管程通过防止对一个资源的并发访问,达到实现临界区的效果,提供一种实现互斥的简单途径,但是并未提供进程与其他进程通信或同步的手段。为了进行并发处理,需要引入同步工具使得进程在资源不能满足而无法继续运行时被阻塞,同时还需开放管程。解决的方法是,使用称作条件变量(conditional variable)的同步机制,让等待的进程临时放弃管程控制权,然后,在适当的时刻,再尝试检测管程内状态的变化。

(1) 条件变量

条件变量是出现在管程内的一种数据结构,且只有在管程中才能被访问,它对管程内的所有过程是全局的,只能通过两个原语操作来控制它。

(2) wait()

挂起调用进程并释放管程,直至另一个进程在条件变量上执行 signal()。

(3) signal()

如果有其他进程因对条件变量执行 wait() 而被挂起,便释放之;如果没有进程在等待,那么,信号不被保存。

当进程调用管程的过程发现无法继续执行,例如,发现没有可用资源,它在某些条件变量 c 上执行 c.wait(),此操作将引起调用进程阻塞并移入与条件变量 c 相关的队列中,这时应开放管程,允许被阻塞在管程之外的一个进程进入管程;另一个进程可通过对其伙伴在等待的同一个条件变量 c 上执行 c.signal() 来唤醒等待进程并将其移出条件变量 c 队列,然后进程会在 wait 操作之后继续执行。值得注意的是,虽然条件变量也是一种信号量,但它并不是 P、V 操作中纯粹的计数信号量,没有与条件变量关联的值,不能像信号量那样积累供以后使用,仅仅起到维护等待进程队列的作用,当在一个条件变量上不存在等待的进程时,signal 操作所发出的信号被丢弃,等于空操作。wait 操作一般应在 signal 操作之前发出,这一规则能大大简化管程的实现。

管程有时要对一个不能得到共享资源的进程进行延迟,当其他进程释放它所需要的资源时,再恢复其执行,wait 原语可延迟进程的执行,signal 原语使得被延迟进程中的某一个进程恢复执行。现在的问题是,当某进程执行 signal 操作之后,另一个被延迟的进程可恢复执行,于是可能有两个进程同时调用一个管程中的两个过程,造成管程中有两个进程同时执行,根据管程的互斥性,这是绝对不允许的。采用两种方法来防止这种现象的出现。

假定进程 P 执行 signal 操作,条件变量队列中有一个进程 Q 在等待,当进程 P 执行 signal 操作后,进程 Q 被释放,对它们可作如下处理:

- (1) 进程 P 等待直至进程 Q 退出管程,或者进程 Q 等待另一个条件。
- (2) 进程 Q 等待直至进程 P 退出管程,或者进程 P 等待另一个条件。

Hoare 采用第一种方法,而 Hansen 选择两者的折中,他规定管程中的过程所执行的 signal 操作是过程体的最后一个操作,于是,进程 P 执行 signal 操作后立即退出管程,因而,进程 Q 马上恢复执行。可见 Hansen 所实现的管程功能较弱,本章介绍 Hoare 管程实现方法,Hansen 管程实现方法作为练习由读者自己完成。

在此进一步把管程和进程作比较,它们存在以下区别。

- (1) 管程所定义的是公用数据结构,而进程所定义的是私有数据结构。
- (2) 管程把共享变量上的同步操作集中起来统一管理,而临界区却分散在每个进程中。
- (3) 管程是为解决进程共享资源的互斥而建立的,而进程是为占有系统资源和实现系统并发性而引入的。
- (4) 管程被欲使用共享资源的所有进程所调用,管程和调用它的进程不能并行工作;而进程之间能够并行工作,并发性是其固有特性。
- (5) 管程可作为语言或操作系统成分,不必创建或撤销;而进程有生命周期,由创建而产生至撤销便消亡。

3.4.2 管程的实现

Hoare 方法使用 PV 操作原语来实现对管程中过程的互斥调用功能,并实现对共享资源互斥使用的管理。每当有进程等待资源时,此方法将让执行 signal 操作的进程挂起,直到被它释放的进程退出管程或产生其他的等待条件为止,而且 wait 和 signal 操作可被设计成两个可中断的过程。

1. 互斥信号量 mutex

对于每个管程, 使用一个用于管程中过程互斥调用的信号量 mutex(初值为 1)。任何进程调用管程中的任何过程时, 应执行 P(mutex); 进程退出管程时应执行 V(mutex)开放管程, 以便让其他调用者进入。为了使进程在等待资源期间, 其他进程能够进入管程, 故在 wait 操作中也必须执行 V(mutex), 否则, 会妨碍其他进程进入管程, 导致无法释放资源。

2. 挂起发出 signal 操作的进程的信号量 next 和计数器 next_count

对于每个管程, 还必须引入信号量 next(初值为 0), 凡发出 signal 操作的进程用 P(next)挂起自己, 直到被释放进程退出管程或产生其他等待条件。每个进程在退出管程的过程之前, 必须检查是否有其他进程在信号量 next 上等待, 若有, 则用 V(next)唤醒它。因此, 在引入信号量 next 的同时, 提供 next_count(初值为 0), 记录在信号量 next 上等待的进程个数。

3. 挂起等待资源的进程的信号量 x_sem 和计数器 x_count

为了使申请资源者在资源被占用时能将其封锁起来, 引入信号量 x_sem(初值为 0), 当申请资源得不到满足时, 执行 P(x_sem)挂起自己。由于在释放资源时, 需要知道是否有其他进程正在等待资源, 因而, 要用一个计数器 x_count(初值为 0)记录等待资源的进程数。在执行 signal 操作时, 应让等待资源的诸进程中的某个进程立即恢复运行, 而不让其他进程抢先进入管程, 这可以用 V(x_sem)来实现。

//Hoare 方法数据结构和 enter、wait、signal 及 leave 操作

```

typedef struct InterfaceModule {
    semaphore mutex;           //InterfaceModule 是结构体名
    semaphore next;            //进程调用管程的过程之前所使用的互斥信号量
    int next_count;            //发出 signal 操作的进程挂起自己的信号量
                                //在 next 上等待的进程数
};

mutex = 1; next = 0; next_count = 0; //初始化语句

void enter(InterfaceModule &IM) {
    P(IM.mutex);             //互斥进入管程
}

void leave(InterfaceModule &IM) {
    if (IM.next_count > 0)      //判断有否发出 signal 操作的进程
        V(IM.next);            //若有就释放一个发出 signal 的进程
    else
        V(IM.mutex);           //否则开放管程
}

void wait(semaphore &x_sem, int &x_count, InterfaceModule &IM) {
    x_count++;                //等待资源的进程数加 1, x_count 初始化为 0
    if (IM.next_count > 0)      //判断有否发出 signal 操作的进程

```

```

    V(IM.next);           //若有就释放一个
else
    V(IM.mutex);         //否则开放管程
    P(x_sem);            //等待资源的进程阻塞自己,x_sem 初始化为 0
    x_count--;           //等待资源的进程数减 1
}

void signal(semaphore &x_sem,int &x_count,InterfaceModule &IM) {
    if(x_count>0) {      //判断是否有等待资源的进程
        IM.next_count++;   //发出 signal 操作的进程数加 1
        V(x_sem);           //释放一个等待资源的进程
        P(IM.next);          //发出 signal 操作的进程阻塞自己
        IM.next_count--;      //发出 signal 操作的进程数减 1
    }
}

```

3.4.3 使用管程解决进程同步问题

1. 管程解决 5 位哲学家吃通心面问题

首先引入枚举类型 enum {thinking,hungry,eating} state[5]表示哲学家的状态,哲学家 i 建立状态 state[i] = eating 仅当他的两个邻座不在吃面的时候,即 state[(i - 1) % 5] != eating 及 state[(i + 1) % 5] != eating;还要引入条件变量(信号量)semaphore self[5],当哲学家 i 饥饿但又不能获得两把叉子时,进入其信号量等待队列。

```

//Hoare 方法解决哲学家吃通心面问题
type dining_philosophers=monitor
    enum {thinking,hungry,eating} state[5];
    semaphore self[5];
    int self_count[5];
    InterfaceModule IM;
    for(int i=0;i<5;i++)           //初始化,i 为进程号
        state[i] = thinking;
    define pickup,putdown;
    use enter,leave,wait,signal;
void pickup(int i)           //i=0,1,⋯⋯,4
    enter(IM);
    state[i] = hungry;
    test(i);

```

```

if(state[i] != eating)
    wait(self[i], self_count[i], IM);
leave(IM);
}

void putdown(int i) { //i=0,1,...,4
    enter(IM);
    state[i] = thinking;
    test((i-1)%5);
    test((i+1)%5);
    leave(IM);
}

void test(int k) { //k=0,1,...,4
    if((state[(k-1)%5] != eating) && (state[k] == hungry)
        && (state[(k+1)%5] != eating)) {
        state[k] = eating;
        signal(self[k], self_count[k], IM);
    }
}

```

任何一位哲学家想吃通心面时调用过程 pickup, 吃完通心面之后调用过程 putdown。

```

cobegin
process philosopher_i() { //i=0,1,...,4
    while(true) {
        thinking();
        dining_philosophers.pickup(i);
        eating();
        dining_philosophers.putdown(i);
    }
}

```

coend

2. 管程解决生产者 - 消费者问题

```

//管程解决生产者 - 消费者问题
type producer_consumer = monitor
    item B[k]; //缓冲区个数
    int in,out; //存取指针
    int count; //缓冲区中产品数

```

```

semaphore notfull,notempty;           //条件变量
int notfull_count,notempty_count;
InterfaceModule IM;
define append,take;
use enter,leave,wait,signal;
void append(item & x) {
    enter(IM);
    if(count==k)                  //缓冲区已满
        wait(notfull,notfull_count,IM);
    B[in]=x;
    in=(in+1)%k;
    count++;                     //增加一个产品
    signal(notempty,notempty_count,IM); //唤醒等待的消费者
    leave(IM);
}
void take(item &x) {
    enter(IM);
    if(count==0)
        wait(notempty,notempty_count,IM); //缓冲区已空
    x=B[out];
    out=(out+1)%k;
    count--;                     //减少一个产品
    signal(notfull,notfull_count,IM); //唤醒等待的生产者
    leave(IM);
}
cobegin
process producer_i() {               //i=1,2,...,n
    item x;
    produce(x);
    producer_consumer.append(x);
}
process consumer_j() {                //j=1,2,...,m
    item x;
    producer_consumer.take(x);
    consume(x);
}

```

+

coend

3.5 进程通信

并发进程之间的交互必须满足两个基本要求：同步和通信。进程同步本质上是一种仅传送信号的进程通信，通过修改信号量，进程之间可以建立联系，相互协调运行和协同工作，但它缺乏传递数据的能力。在多任务环境中，可由多个进程分工协作完成同一任务，于是，它们需要共享一些数据和相互交换信息，某些情况下所交换的信息量很少，但在很多场合需要交换大批数据，可以通过通信机制来完成，进程之间互相交换信息的工作称为进程通信（InterProcess Communication, IPC）。通信方式有很多，包括：

- (1) 信号(signal)通信机制
- (2) 管道(pipeline)通信机制
- (3) 消息传递(message passing)通信机制
- (4) 信号量(semaphore)通信机制
- (5) 共享主存(shared memory)通信机制

其中，前两种是 UNIX 最早的版本就提供的进程通信机制，信号通信机制只能发送单个信号而不能传送数据；管道通信机制虽然能传送数据，却只能在进程家族内使用，应用范围有限。此后，UNIX System V 版研制和开发后三种进程通信机制，这就是著名的 System V IPC，其共性是，在同一机器上的任何进程都可以使用这些机制通信，且相互通信的进程并不需要有家族关系。而 BSD UNIX 则实现了套接字(socket)网络进程通信机制，第八章将对此作简单介绍。

3.5.1 信号通信机制

信号(signal)是一种软中断，是传递短消息的简单通信机制，通过发送指定信号来通知进程某个异步事件发生，以迫使进程执行信号处理程序。信号处理完毕后，被中断进程将恢复执行。同中断和异常相比，多数信号对用户态进程是可见的，可以被应用进程捕获。信号可通过异常(硬中断)直接产生，如进程执行非法指令或段违例等，内核会发送给它 SIGILL 或 SIGSEGV；进程也可发出信号，以实现进程的同步或终止，如 SIGKILL 将强迫另一进程终止。一般地，分或操作系统标准信号和应用进程定义信号，这种机制模拟硬中断，但不分优先级，简单且有效，但不能传送数据，故能力较弱。

有关软中断与硬中断的相似点和差别已在中断机制中介绍过，举例来说，当用户正在运行一个耗时的应用程序时，如果已发现有错误，且断定此程序要失败，为了节省时间，可杀死当前运行进程。系统中所发生的相应事件链如下：

步骤 1：用户按中断组合键 Ctrl+C。

步骤 2：终端驱动程序收到输入字符，并调用信号系统；

步骤 3:信号系统发送 SIGINT 信号给 shell,shell 再把它发送给进程;

步骤 4:进程收到 SIGINT 信号;

步骤 5:进程撤销。

用户、内核和进程都能生成信号请求:

(1) 用户

用户能通过按键 Ctrl+C,或终端驱动程序分配给信号控制字符的其他键来请求内核产生信号。

(2) 内核

当进程执行出错时,内核检测到事件并向进程发送信号,例如,非法段存取、浮点数溢出或非法操作码,内核也利用信号通知进程有特定事件发生。

(3) 进程

进程可通过系统调用 kill 给另一个进程发送信号,一个进程可通过信号与另一个进程通信。

由进程执行指令而产生的信号称为同步信号,如被 0 除;像击键之类的进程以外的事件所引起的信号称为异步信号,接收进程对信号所做出的响应为:执行默认操作(如 SIGINT 的默认处理是进程撤销)、执行预置的信号处理程序或忽略此信号。标准版 UNIX 提供 19 个信号,Linux 中定义 64 个信号,采用与标准版兼容且增加新信号的方法保证信号通信的一致性和兼容性。信号可分成以下几类。

(1) 与终止进程相关的信号:SIGCHLD(子进程暂停或终止)、SIGHUP(进程挂起)、SIGKILL(强行杀死进程)、SIGABRT(进程异常终止)、SIGSTOP(被调试器阻塞,进程暂停)、SIGTSTP(来自终端的暂停信号)等;

(2) 与例外事件相关的信号:SIGBUS(总线超时)、SIGSEGV(段违例)、SIGPWR(电源故障)、SIGFPE(浮点溢出)、SIGSTKFLT(协处理器栈出错)、SIGURG(套接字紧急情况)、SIGXCPU(CPU 限时超出)、SIGXFSZ(文件限制超出)、SIGWINCH(窗口大小变化)等;

(3) 与执行系统调用相关的信号:SIGPIPE(管道有写者无读者)、SIGILL(非法指令)、SIGIO(与 I/O 操作有关)等;

(4) 与终端交互相关的信号:SIGINT(键盘中断)、SIGQUIT(输入退出命令)、SIGTTIN(后台进程读终端)SIGTTOU(后台进程写终端)等;

(5) 与用户进程相关的信号:SIGTERM(软件终止)、SIGALRM(定时报警)、SIGUSR1(用户定义信号 1)、SIGUSR2(用户定义信号 2)等;

(6) 与跟踪进程执行相关的信号:SIGTRAP(跟踪陷阱)等。

信号有一个产生、传送、捕获和释放的过程,Linux 信号实现机制包括:数据结构、信号的产生和发送、信号的检测和处理。

(1) 每个进程 task_struct 结构中的 signal 域专门保存接收到的信号,内核根据所发生的事件产生相应的信号并发送给接收进程。进程接收到信号时,对应位设置为 1,相当于“中断请求寄存器”的某位置位,由于所发送的信号被置于接收进程的进程控制块中,所以,无论进程是否在主存,总可以立即被记录。一个进程可能收到多个不同的信号,系统仅允许进程每次处理编号最小

的--一个,其余信号只有在此进程下次调度运行时才被处理。

(2) 进程 task_struct 结构中的 blocked 是信号屏蔽标记,相当于“中断屏蔽寄存器”。进程需要忽略某信号时就把对应的位设置为 1。sigpending 标志指示在 sigqueue 队列中是否有挂起的信号,如果信号发送之后目标进程在睡眠就唤醒它,让它接收和处理此信号,任何时候一种类型的信号至多只有一个信号挂起。

(3) 信号处理程序的人口存放在进程 task_struct 的 sigaction[] 数组中,数组共有 64 个元素,每个元素占 32 bit,信号的编号对应于数组下标,数组元素的值是信号处理程序的人口地址。有很多方法可设置信号处理方式,除了 signal 之外,如执行函数 fork、exec、exit 等都会改变数组的内容。

(4) 函数 sigaction(signo,act,oldact) 为指定信号预置处理程序,signo 指出接收信号的类型,act 是收到信号后希望调用的信号处理函数地址,oldact 存放最近一次为信号 signo 定义的函数地址。为保持兼容性,仍保留 signal 函数,但功能上已被 sigaction 替代。

(5) 函数 kill(pid,sig) 用来向指定的进程发送指定信号。当信号被发送时,内核的工作是更新进程控制块的信号域,并对与信号相关的数据结构进行操作,如置位、屏蔽和清除等。pid 确定信号 sig 所发往的进程,sig 是信号类型,由于这种信号要知道所发往进程的标识号,所以,信号发送通常在关系密切的进程之间进行。

(6) 信号的检测和响应总发生在系统空间,在中断或异常处理程序末尾,在进程从核心态返回用户态之前,或在处理运行进程所遇到的时钟中断结束之前,或进程以 interruptible 状态进入等待队列之前(若它已收到信号,就不睡眠),系统会调用内核函数 do_signal() 检查此进程是否已收到信号,若是则执行 handle_signal() 让它返回用户态并转入信号处理程序执行。当信号处理结束后,通过执行系统调用 sigreturn() 陷入内核,内核做好善后工作后返回用户态,回到应用程序的断点执行。如图 3.4 所示是运行进程收到一个信号后的检测和处理流程。

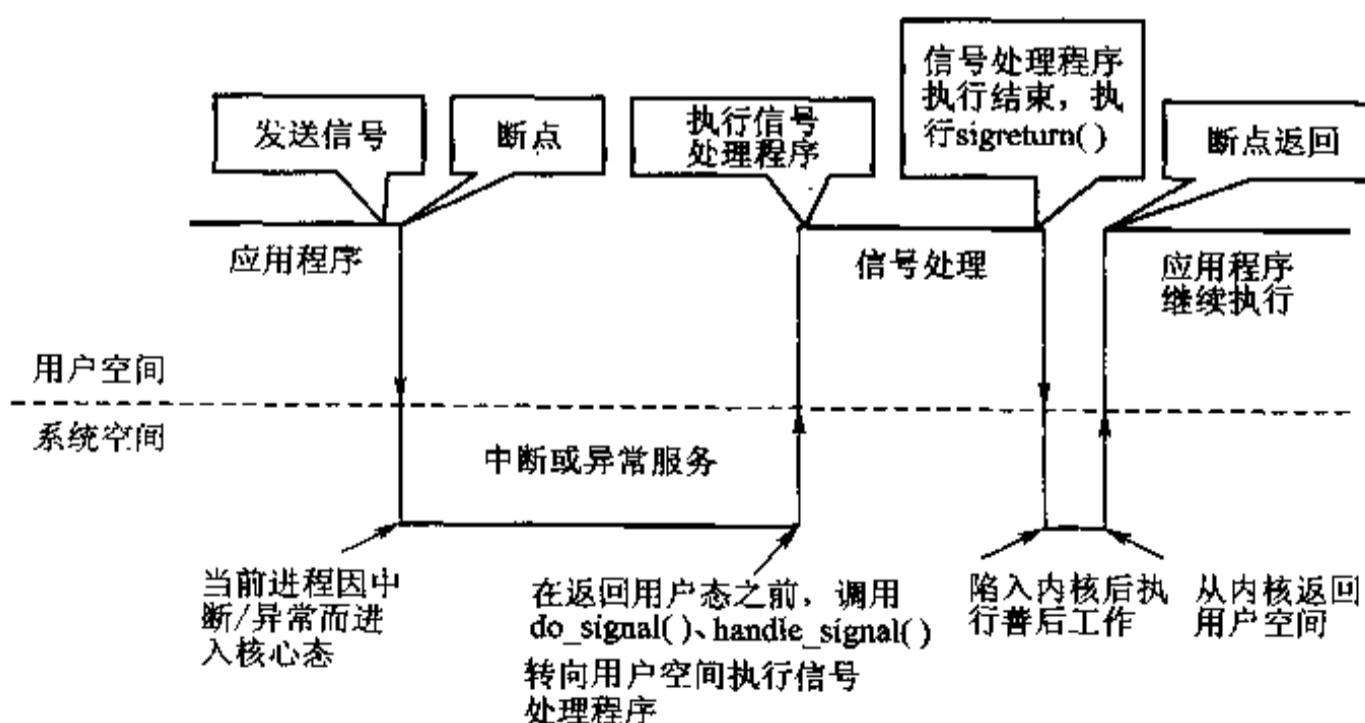


图 3.4 信号的检测和处理流程

Windows 操作系统的信号通信机制统一在分派器对象(dispatcher object)的公共框架中,每个分派器对象可处于两种状态之一:已发信号(signaled)和未发信号(unsignaled)。进程可以使用函数 WaitForSingleObject 把自己阻塞,以等待一个未发信号的对象,当某个进程把此对象的状态改为已发信号时,此进程就会恢复运行。进程也可使用函数 WaitForMultipleObjects 来在多个对象上阻塞自己,当所有对象均被发送信号后,它才会恢复执行。对象的类型包括进程、线程、文件、事件、信号量、定时器、互斥量和队列。例如,当线程终止时,就会向线程对象发送信号,所有等待线程都被唤醒;当线程释放互斥锁之后,就会向互斥量对象发送信号,一个等待线程被唤醒;当定时器到时的时候,就会向定时器对象发信号,可以唤醒所有等待线程或仅唤醒一个等待线程;当某文件的 I/O 操作终止时,就会向此文件对象发送信号,所有等待线程将被唤醒。

3.5.2 管道通信机制

管道(pipeline)是 UNIX 的传统进程通信方式,也是 UNIX 发展最有意义的贡献之一。它是连接读写进程的一个特殊文件,允许按照 FCFS 方式传送数据,也能使进程同步执行。管道是单向的,发送进程视管道文件为输出文件,以字符流的形式把大量数据送入管道;接收进程视管道文件为输入文件,从管道中接收数据,所以也称为管道通信。由于管道通信机制方便有效,能够在进程间进行大信息量的通信,目前已被引入许多操作系统中。

管道的实质是一个共享文件,即利用辅助存储器来进行数据通信。因此,管道通信基本上可借助于文件系统的机制来实现,包括管道文件的创建、打开、关闭和读写。但是,写进程和读进程之间的相互协调单靠文件系统机制是解决不了的,读写进程相互同步,必须做到以下几点。

(1) 进程正在使用管道写入或读取数据时,另一个进程必须等待,这一点是进程在读写管道之前,通过测试文件 inode 节点的特征位——读写互斥标志来保证的;若已锁住,进程便等待,否则,把 inode 上锁,然后,进行读写操作。操作结束后再行解锁并唤醒因节点上锁而等待的进程。

(2) 发送者和接收者双方必须知道对方是否存在,如果对方已经不存在,就没有必要再发送或接收信息,这时系统会发出 SIGPIPE 信号通知进程。

(3) 发送信息和接收信息之间要实现正确的同步关系,这是由于管道文件只使用 inode 节点中的直接地址项,长度限于 10 个盘块,管道的长度限制对于进程的 write 和 read 操作会产生影响。如果进程执行一次写操作,且管道有足够的空间,那么,write 操作把数据写入管道后唤醒因此管道空而等待的进程;如果此次操作会引起管道溢出,则本次 write 操作必须暂停,直到其他进程从管道中读取数据,使管道有空余空间为止,这叫做 write 阻塞。解决这个问题的方法是:把数据进行切分,每次均小于管道限制的字节数,写完后此进程睡眠,直到读进程把管道中的数据取走,并判别有进程等待时唤醒对方,以便继续写下一批数据。反之,当读进程读空管道时,要出现 read 阻塞,读进程应睡眠,直到写进程唤醒它。

应用程序用 pipe() 创建管道,pipe 分配文件磁盘 inode 和活动 inode 节点,然后,为读写进程各分配一个文件描述符,同时,系统打开文件表中应建立两个 file 结构,分别控制管道的写操作和读操作,即定义 pipe 文件的写入端和读出端。两个 file 结构分别与文件描述符相连,并指向同

一个活动 inode 节点, 活动 inode 节点的 `i_count` 值为 2, 因为有两个 file 项指向它。内核在执行完 `pipe()` 系统调用创建无名管道后, 返回文件句柄 `files[0]` 和 `files[1]`, 接收者进程通过 `files[0]` 从管道中用 `read()` 读数据, 发送者进程则通过 `files[1]` 用 `write()` 向管道写入数据, 结束之后用 `close()` 关闭管道。`pipe` 的数据结构如图 3.5 所示。

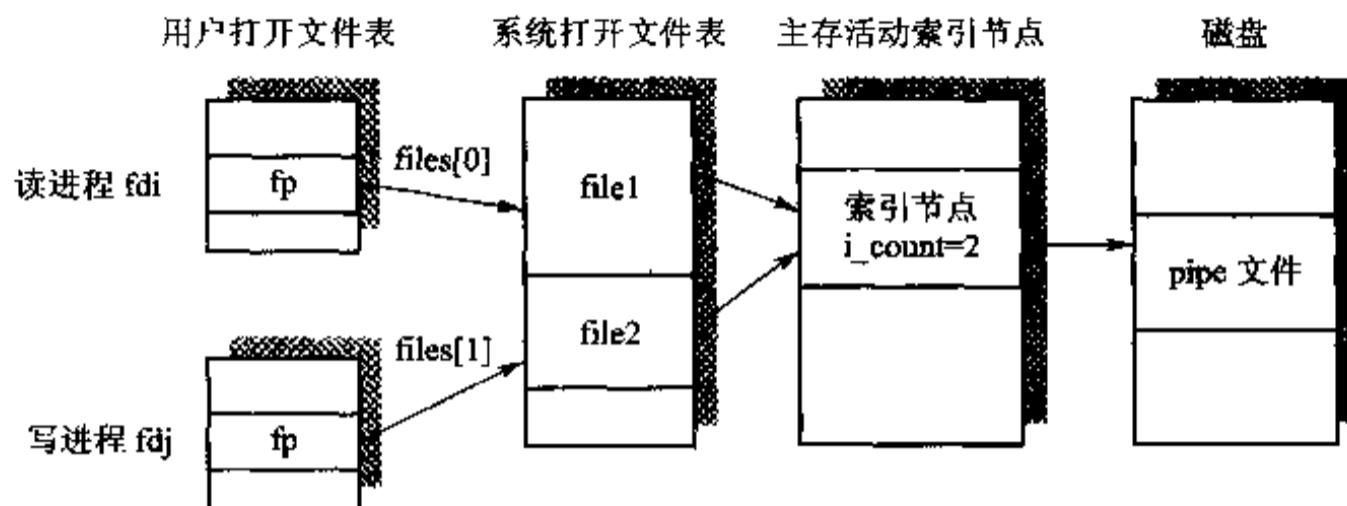


图 3.5 `pipe` 的数据结构

管道是一种功能很强的通信机制, 但仅用于连接具有共同祖先的进程, 使用时需要临时建立, 难以提供全局服务。为了克服这些缺点, UNIX 推出管道的一个变种, 称为有名管道或 FIFO 通信机制, 用来在不同的地址空间之间进行通信, 特别为服务器通过网络与多个客户进行交互而设计。这是一种永久性通信机制, 具有文件名、目录项、访问权限, 能像一般文件一样被操作, 但在读或写时, 其性能与管道相同。可以通过系统调用 `mknod` 而非 `pipe` 来创建命名管道, 然后, 接收者进程像使用其他文件一样通过系统调用 `open` 来打开管道以便取走信息; 发送者进程则通过系统调用 `open` 来打开管道以写入数据。对管道的读写操作通过系统调用 `read` 和 `write` 来实现。

3.5.3 共享主存通信机制

共享主存是指在主存中开辟一个公用存储区, 要通信的进程把自己的虚地址空间映射到共享主存区, 如图 3.6 所示。当发送进程将信息写入共享主存区的某个位置时, 接收进程可从此位置读取信息, 由于共享主存区同时出现在进程 1 和进程 2 的虚存空间中, 从而能实现进程之间的通信。这也是进程通信中最快捷和最有效的方法, 此机制最早是在 UNIX System V 中作为进程通信的一部分而设计的。通过共享主存 API, 允许进程动态地定义共享主存区, 由于进程的虚拟地址空间相当大, 所定义的共享主存区应对应于一段未使用的虚地址区, 以免与进程映像区发生冲突。共享主存的页面在每个共享进程的页表中都有页表项引用, 但无须在所有进程的虚存段都有相同的地址。因为不

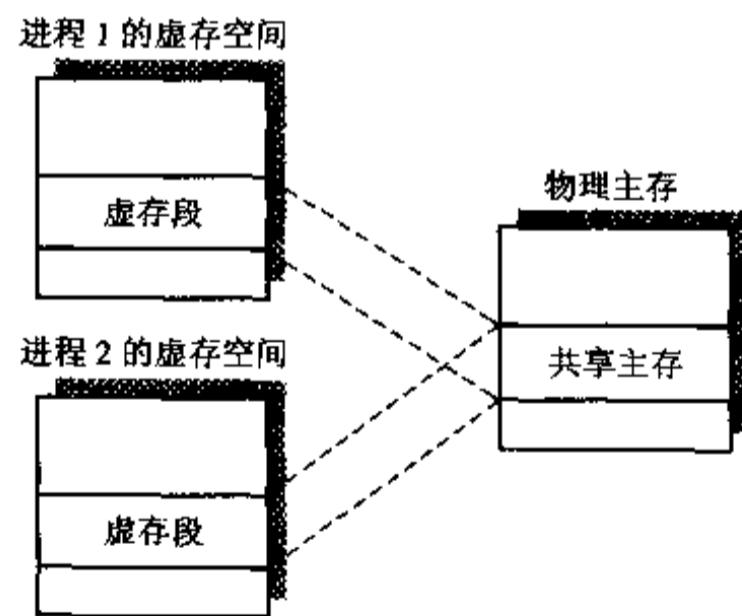


图 3.6 共享主存通信机制

止一个进程可将共享主存映射到自己的虚地址空间中去,读写共享主存区的代码段通常被认为是临界区。

3.5.4 消息传递通信机制

1. 消息传递的概念

交互式并发进程通过信号量及 PV 操作可实现进程的互斥和同步。生产者 - 消费者问题是一组相互协作的进程,它们通过信号量来协同工作,另外引入有界缓冲区来存取产品,这种低级通信方式既不方便且局限性也大。当进程之间需要交换更多的信息时,例如,要求把数据从一个进程传送给另一个进程,甚至不同机器上的进程之间要传送数据,需要引入高级通信方式——消息传递(message passing)机制(UNIX 和 Linux 中称其为消息队列)来实现。由于操作系统所提供的这类机制隐蔽实现细节,通过消息传递机制完成通信,就能简化程序编制的复杂性,方便易用。

消息传递的复杂性在于:地址空间的隔离,发送进程无法将消息直接复制到接收进程的地址空间中,这项工作仅能由操作系统来完成。为此,消息传递机制至少需要提供两条原语 send 和 receive,前者向一个给定的目标进程发送一条消息,后者则从一个给定的源进程接收一条消息。如果没有消息可用,则接收者可能阻塞直至一条消息到达,或者也可立即返回,并带回一个错误码。采用消息传递机制后,进程之间用消息来交换信息,一个运行进程可在任何时刻向另一个运行进程发送一条消息;一个运行进程也可在任何时刻向另一个运行进程请求一条消息,如果进程在某一时刻的执行依赖于另一进程的消息或等待其他进程对所发消息的应答,那么,消息传递机制将紧密地与进程的阻塞和释放相联系,使得进程通信机制不但具有进程通信的能力,还提供进程同步的能力。

为了实现异步通信,必须采用间接的通信方式,进程之间发送或接收消息通过一个共享的数据结构——信箱进行,消息可被理解成信件,每个信箱都有唯一标识符。当两个以上的进程拥有共享信箱时,它们就能进行通信。间接通信解除了发送进程和接收进程之间的直接联系,在消息的使用上加大了灵活性。一个进程可以分别与多个进程共享信箱,于是,一个进程可同时和多个进程进行通信,一对一关系允许在两个进程间建立不受干扰的专用通信链接;多对一关系对客户 - 服务器间的交互非常有用;一个进程为其他进程提供服务,这时的信箱又称作端口(port),端口通常划归接收进程所有并由接收进程创建,服务进程被撤销时,其端口也随之毁灭;一对多关系适用于一个发送者和多个接收者,可在一组进程间发送广播消息;多对多关系允许建立公用信箱,多个进程既可向信箱发送信件,也可从中取出所属信件。间接通信方式中的“发送”和“接收”原语的形式如下:

- (1) send(A, 消息):把一条消息(信件)传送到信箱 A。
- (2) receive(A, 消息):从信箱 A 接收一条消息(信件)。

信箱是存放信件的存储区域,每个信箱可分成信箱头和信箱体,信箱头指出信箱容量、信件格式、存放信件的位置的指针等;信箱体用来存放信件,信箱体分成若干区,每个区可容纳一条消

息。“发送”和“接收”两条原语的功能为：

(1) 发送信件

如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待此信箱中信件的等待接收者；否则，发送信件者被置成等待信箱状态。

(2) 接收信件

如果指定的信箱中有信，则取出一封信件，并释放等待信箱的等待发送者，否则，接收信件者被置成等待信箱信件的状态。

2. 信箱的设置

信箱可以在用户空间或系统空间开辟。若信箱设置在用户空间，那么，此信箱属于进程私有，需要区分信箱的所有者及其使用者，创建信箱的进程就是信箱的所有者，它有权从中收信，其他进程都可以向此进程的信箱发信，当拥有信箱的进程撤销时，其信箱也随之消失，这时必须把这一情况及时通知信箱的所有使用者。

一种替代方法是在系统空间设置每个进程的私有信箱，并推迟消息的复制操作，直到接收进程发出接收系统调用。由于进程不能直接访问消息，能够避免信箱和消息被破坏的危险，但需要系统为所有进程的信箱分配存储空间，分配多大空间较难决定，给定时间内等待传送的消息数目也会受到限制。一种改进的方法是：在系统空间统一设置公用信箱，供系统中的所有进程共享，B.Hansen 的消息缓冲机制就是一个著名的例子。

3. 通信进程的同步

两个进程间的消息通信就隐含着某种程度的同步，当发送进程执行 `send()` 发出消息后，本身的执行可分为两种情况，一种是同步的（阻塞型），等待接收进程回答消息后才继续进行；另一种是异步的（非阻塞型），将消息传送到接收进程的信箱中，允许继续运行，直到某个时刻需要接收进程送来回答消息时，才查询和处理。对于接收进程而言，执行 `receive` 后也可以是阻塞型（同步的）和非阻塞型（异步的），前者是指直到消息交付完成它都处于等待消息的状态；后者则不要求接收进程等待，当它需要消息时，再接收并处理消息。

无论同步或异步 `send` 操作，都会有失败的情形。如果发送进程试图向一个并不存在的进程发送消息，操作系统将无法识别用哪个信箱来缓存消息，那么，下一步如何处理呢？同步 `send` 时，会返回一个错误码至发送方，发送方依赖错误码来工作；异步 `send` 时，发送方可继续发送消息，而不期望有任何返回码，系统可以使用像 UNIX 中的信号机制来告诉发送方消息的发送失败。

`receive` 操作可以是阻塞型或非阻塞型的。阻塞型接收操作的行为是：如果信箱中没有消息，接收进程会被挂起，直到有消息投入信箱；如果信箱中有消息，则立即获得一条消息并返回。因而，阻塞型 `receive` 操作能够同步发送进程和接收进程的通信；非阻塞型 `receive` 操作在查询信箱之后，立即向调用进程返还控制权，如果信箱中有消息，就返回消息，否则返回标志码，表明无消息可用。这种方法允许接收进程轮询信箱，如果信箱中没有待处理的消息，它可以去做其他工作。

采用非阻塞型 send 和阻塞型 receive 是用得较广的同步方式,发送进程不阻塞,故可把一条或多条消息发给目标进程;而接收进程通常是服务器进程,平时处于阻塞状态,直到发送进程发来消息才被唤醒。发送进程和接收进程均不阻塞的同步方式也经常被使用,例如,发送进程和接收进程共同联系一个能容纳 n 条消息的消息队列,发送进程可连续发送消息至消息队列,而接收进程可连续地从消息队列获取消息,仅当消息队列中的消息数达到 n 条时,发送进程才会被阻塞。类似地,仅当消息队列中没有消息时,接收进程才会被阻塞。

4. 消息传通机制解决进程的互斥和同步问题

(1) 解决进程互斥问题

```

create_mailbox(box);
send(box,null);

void Pi() {           //i=1,2,...,n
    message msg;
    while(true) {
        receive(box,msg);
        {临界区};
        send(box,msg);
    }
}

cobegin
    Pi();
coend

```

一组并发进程共享一个信箱 box, 使用阻塞型 receive 和非阻塞型 send, 信箱内容被初始化为一条无内容的消息。当消息队列为空时,所有进程被阻塞;当系统构造一条消息发送到 box 时,只有一个进程能获得消息,它进入临界区执行,完成退出时把消息发送回信箱,因此,此消息可看做进程之间传递进入临界区权力的令牌。

(2) 生产者 - 消费者问题的一种解法

```

int capacity;           //缓冲区大小
create-mailbox(producer); //创建信箱
create-mailbox(consumer);
for(int i=0;i<capacity;i++)
    send(producer,null); //发送空消息
void producer_i() {
    message pmsg;
    while(true) {
        produce(); //生产消息
    }
}

```

```

        receive(producer,null);           //等待空消息
        build();                         //构造一条消息
        send(consumer,pmsg);            //发送消息
    }

}

void consumer_j() {                //j=1,2,...,m
    message cmsg;
    while(true) {
        receive (consumer,cmsg);     //接收消息
        extract();                   //读取消息
        send(producer,null);         //回送空消息
        consume(cmsg);              //消耗消息
    }
}

cobegin
    producer_i();
    consumer_j();
coend

```

此程序中假设消息大小相同,共用 capacity 条消息,类似于共享主存缓冲的 capacity 个槽。初始化时,由主程序负责发送 capacity 条空消息给生产者。当生产者生产出一个产品后,它接收一条空消息并回送一条填入产品的消息给消费者,通过这种方式,系统中总的消息数保持不变。如果生产者的速度比消费者快,则所有空消息取尽,于是生产者发生阻塞以等待消费者返回一条空消息;如果消费者的速度比生产者快,则正好相反,所有的消息均为空,等待生产者填充数据发送消息来释放它。

在这个解法中,生产者和消费者均创建足够容纳 capacity 条消息的信箱,消费者向生产者信箱发送空消息,生产者向消费者信箱发送含有产品的消息。当使用信箱时,通信机制会知道:目标信箱容纳那些已被发送但尚未被目标进程接收的消息。

死 锁

3.6.1 死锁产生

计算机系统中有一些资源,在任一时刻都只能被一个进程所使用,如磁带机、绘图仪等独占型外部设备,或共享数据结构、等待链表等软件资源。两个进程同时向一台打印机输出数据将导致一片混乱,两个进程同时进入临界区将导致数据错误乃至系统崩溃,因此,所有操作系统都具

有向一个进程授权独立访问某个资源的能力,进程使用独占型资源必须按照以下的次序进行:申请资源、使用资源、归还资源。若资源不可用,则申请进程进入等待态。对于不同类型的独占型资源,进程的等待方式是有差异的,如申请打印机资源、临界区资源时,申请失败将意味着阻塞申请进程;而申请打开文件资源时,申请失败将返回一个错误代码,由申请进程等待一段时间后再次尝试。

在许多应用中,进程需要以独占方式访问多个资源,而当操作系统允许多个进程并发执行时,可能出现进程永远被阻塞的现象。例如,两个进程分别等待对方所占有的一个资源,于是两者都不能执行而处于永远等待状态,此现象称为“死锁”。产生死锁的原因有很多,例如,进程推进顺序不当、P或V操作使用不妥、同类资源分配不均或对某些资源的使用未加限制,可见,产生死锁的因素不仅与系统拥有的资源数量有关,而且与资源分配策略、进程对资源的使用要求以及并发进程的推进顺序有关。

死锁通常被定义为:如果一个进程集合中的每个进程都在等待只能由此集合中的其他进程才能引发的事件,而无限期陷入僵持的局面称为死锁。

本章基于进程来讨论死锁,这些概念也同样适用于线程之间产生死锁的情况。死锁会让系统造成很大的损失,必须付出额外的代价来解决死锁问题,主要的解决方法是:死锁防止(deadlock prevention)、死锁避免(deadlock avoidance)、死锁检测和恢复(deadlock detection and recovery)。

3.6.2 死锁防止

1. 死锁产生的条件

1971年,Coffman总结系统产生死锁必定同时保持4个必要条件如下。

- (1) 互斥条件(mutual exclusion):系统中存在临界资源,进程应互斥地使用这些资源。
- (2) 占有和等待条件(hold and wait):进程在请求资源得不到满足而等待时,不释放已占有的资源。
- (3) 不剥夺条件(no preemption):已被占用的资源只能由属主释放,不允许被其他进程剥夺。
- (4) 循环等待条件(circular wait):存在循环等待链,其中,每个进程都在链中等待下一个进程所持有的资源,造成这组进程处于永远等待状态。

前3个条件是死锁存在的必要条件,但不是充分条件,第4个条件是前3个条件同时存在时所产生的结果,故条件并不完全独立。但是,单独考虑每个条件是有用的,只要能破坏4个必要条件之一,就可以防止死锁。

2. 死锁防止策略

(1) 破坏条件1(互斥条件)

使资源可同时访问而非互斥使用,也就没有进程会阻塞在资源上,从而不发生死锁。可重入程序、只读数据文件、时钟、磁盘等软硬件资源均可采用这种办法管理,但有许多资源如可写文

件、磁带机等由于其特殊性质决定只能互斥地占有,而不能被同时访问,所以这种做法在许多场合行不通。

(2) 破坏条件 2(占有和等待条件)

静态分配是指进程必须在执行之前就申请所需要的全部资源,且直至所要的资源都得到满足后才开始执行,无疑所有并发执行的进程所要求的资源总和不超过系统拥有的资源数。采用静态分配策略后,进程在执行过程中不再申请资源,就不会出现占有某些资源再等待另一些资源的情况。这种策略实现简单,被许多操作系统所采用,但会严重地降低资源利用率,因为在每个进程所占有的资源中,有些资源在运行后期使用,甚至有些资源在例外情况下才被使用,可能会造成进程占有一些几乎用不到的资源,而使其他想使用这些资源的进程等待。

(3) 破坏条件 3(不剥夺条件)

剥夺调度能够防止死锁,但只适用于主存和处理器资源。方法一是占有资源的进程若要申请新的资源,必须主动释放已占用资源(剥夺式),若仍需要占用此资源,应该向系统重新提出申请,从而破坏了不剥夺条件,但会造成进程重复地申请和释放资源;方法二是资源管理程序为进程分配新资源时,若有则分配之,否则将剥夺此进程所占有的全部资源,并让进程进入等待资源的状态,资源充足后再唤醒它重新申请所有所需资源。

(4) 破坏条件 4(循环等待条件)

采用层次分配策略,将系统中的所有资源排列到不同的层次中,一个进程得到某层的一个资源后,只能再申请较高一层的资源;当进程释放某层的一个资源时,必须先释放所占用的较高层资源;当进程获得某层的一个资源后,如果想申请同层的另一个资源,必须先释放此层中的已占用资源。

层次分配策略的一个变种是按序分配策略。把所有的资源按顺序编号,规定进程请求所需资源的顺序必须按照资源的编号依次进行。例如,若系统中共有 n 个进程,共有 m 个资源,用 r_i 表示第 i 个资源,于是 m 个资源是 r_1, r_2, \dots, r_m ,规定进程不得在占用资源 r_i ($1 \leq i \leq m$) 后再申请 r_j ($j < i$),就可防止系统发生死锁,因为采用有序使用资源的方法能够有效地破坏循环等待条件。

3.6.3 死锁避免

1. 银行家算法

各种死锁防止方法能够防止发生死锁,但必然会降低系统的并发性并导致低效的资源利用率。死锁避免却与此相反,允许系统中同时存在前 3 个必要条件,通过合适的资源分配算法确保不会出现进程循环等待链,从而避免死锁。死锁避免方法能支持更多的进程并发执行,它不是对进程随意强加规则,而是动态地确定是否分配资源给提出请求的进程。如果一个进程当前请求的资源会导致死锁,系统将拒绝启动此进程;如果一个资源的分配会导致下一步死锁,系统便拒绝本次分配。

Dijkstra 于 1965 年提出能够避免死锁的调度方法,称为银行家算法。此算法描述如下,假定

小城镇银行家拥有资金,数量为 Σ ,被 N 个客户共享,银行家对客户提出下列约束条件:

- (1) 每个客户必须预先说明所要的最大资金量;
- (2) 每个客户每次提出部分资金量的申请并获得分配;
- (3) 如果银行满足客户对资金的最大需求量,那么客户在资金运作后,应在有限的时间内全部归还银行。

只要客户遵守上述约束条件,银行家将保证做到:若一个客户所要的最大资金量不超过 Σ ,银行一定会接纳此客户,并满足其资金需求;银行在收到一个客户的资金申请时,可能会因资金不足而让客户等待,但保证在有限的时间内让客户获得资金。在银行家算法中,客户可看做进程,资金可看做资源,银行家可看做操作系统。银行家算法虽然能避免死锁,但实现时受到种种限制,要求所涉及的进程不相交,即不能有同步要求,事先要知道进程的总数和每个进程请求的最大资源数,这些都很难办到。

2. 银行家算法数据结构

考虑一个系统有 n 个进程和 m 种不同类型的资源,定义以下向量和矩阵数据结构。

- (1) 系统中每类资源总数向量

$$\text{Resource} = (R_1, R_2, \dots, R_m)$$

- (2) 系统中当前每类资源可用数向量

$$\text{Available} = (V_1, V_2, \dots, V_m)$$

- (3) 进程对各类资源的最大需求矩阵(C_{ij} 表示进程 P_i 需要 R_j 类资源的最大数目)

$$\text{Claim} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \cdots & & & \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$$

- (4) 系统中当前资源的已分配情况矩阵(A_{ij} 表示进程 P_i 已分到 R_j 类资源的数目)

$$\text{Allocation} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdots & & & \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$$

于是下列关系式成立:

$$R_i = V_i + \sum_{k=1}^n A_{ki}, \quad \text{对 } i = 1, 2, \dots, m \text{ 表示所有资源要么已被分配,要么尚可分配}$$

$C_{ki} \leq R_i$, 对 $i = 1, 2, \dots, m; k = 1, 2, \dots, n$ 表示进程申请资源数不能超过系统拥有这类资源的总数

$A_{ki} \leq C_{ki}$, 对 $i = 1, 2, \dots, m, k = 1, 2, \dots, n$ 表示进程申请任何类型资源数不能超过声明的最大资源需求数

系统若要启动一个新进程工作,其对资源 R_i 的需求应满足不等式:

$$R_i \geq C_{(n+1)i} + \sum_{k=1}^n C_{ki}, \quad i = 1, 2, \dots, m; k = 1, 2, \dots, n$$

也就是说,应满足当前系统中所有进程对资源 R_i 的最大需求数,加上启动的新进程的资源最大需求数,不超过系统所拥有的最大资源数时,才能启动此进程,这种死锁避免方法也称作“进程启动拒绝法”。这个算法考虑最坏的情况,即所有进程同时要使用它们声明的资源最大需求数,因而很难是最优的。

现在给出系统安全性的定义:在时刻 T_0 系统是安全的,仅当存在一个进程序列 P_1, P_2, \dots, P_n , 对进程 $P_k (k = 1, 2, \dots, n)$ 满足公式:

$$C_{ki} - A_{ki} \leq V_i + \sum_{j=1}^{k-1} A_{ji}, \quad k = 1, 2, \dots, n; i = 1, 2, \dots, m$$

此序列称为安全序列。其中,公式左边表示进程 P_k 尚缺少的各类资源, V_i 是 T_0 时刻系统尚可用于分配且为 P_k 所想要的那类资源数, $\sum_{j=1}^{k-1} A_{ji}$ 表示排在进程 P_k 之前的所有进程所占用的 P_k 所需要的资源总数。显然,进程 P_k 所需资源若不能立即被满足,那么,在所有 $P_j (j = 1, 2, \dots, k-1)$ 运行完成后可以满足,然后, P_k 也能获得资源以完成任务;当 P_k 释放全部资源后, P_{k+1} 也能获得资源以完成任务;如此下去,直到最后一个进程完成任务,从 T_0 时刻起按照这个进程序列运行的系统是安全的,绝对不会产生死锁。

3. 银行家算法描述

银行家算法又称“资源分配拒绝”法,其基本思想是:系统中的所有进程放入进程集合,在安全状态下系统收到进程的资源请求后,先把资源试探性地分配给它。现在,系统将剩下的可用资源和进程集合中其他进程还需要的资源数作比较,找出剩余资源能满足最大需求量的进程,从而保证进程运行完毕并归还全部资源。这时,把这个进程从进程集合中删除,归还其所占用的所有资源,系统的剩余资源则更多,反复执行上述步骤。最后,检查进程集合,若为空则表明本次申请可行,系统处于安全状态,可以真正实施本次分配;否则,只要进程集合非空,系统便处于不安全状态,本次资源分配暂不实施,让申请资源的进程等待。

//避免死锁的银行家算法

```
typedef struct state{                                //全局数据结构
    int resource[m];
    int available[m];
    int claim[n][m];
    int allocation[n][m];
};

void resource_allocation() {                         //资源分配算法
    if(allocation[i, *] + request[*] > claim[i, *])
        //处理资源分配逻辑
    else
        //处理资源释放逻辑
}
```

```

    {error} ;                                //申请量超过最大需求值
else {
    if(request[ * ] > available[ * ])
        {suspend process};
    else{                               //尝试分配,定义 newstate:
        allocation[i, * ] = allocation[i, * ] + request[ * ];
        available[ * ] = available[ * ] - request[ * ];
    }
}
if(safe(newstate))
    {carry out allocation};
else {
    {restore original state};
    {suspend process};
}
}

bool safe(state s) {                      //安全性测试算法
    int currentavail[m];
    set <process> rest;
    currentavail[ * ] = available[ * ];
    rest = {all process};
    possible = true;
    while(possible) {                    //rest 集合中找一个 Pk, 满足以下条件
        claim[k, * ] - allocation[k, * ] <= currentavail[ * ];
        if(found){
            currentavail[ * ] = currentavail[ * ] + allocation[k, * ];
            rest = rest - {Pk};
        }
        else
            possible = false;
    }
    return(rest = null);
}

```

对上述算法作简短说明如下。

(1) 申请量超过最大需求量时出错,否则转步骤(2);

- (2) 当申请量超过当前系统所拥有的可分配量时,挂起进程,处于等待态,否则转步骤(3);
 (3) 系统对 P_i 进程请求资源进行试探性分配,执行

$$\text{allocation}[i, *] = \text{allocation}[i, *] + \text{request}[*]$$

$$\text{available}[*] = \text{available}[*] - \text{request}[*]$$

- (4) 执行安全性测试算法(5),如果状态安全则承认试分配,否则抛弃试分配,进程 P_i 等待;
 (5) 安全性测试算法

- ① 定义工作向量 currentavail 、布尔型标志 possible 和进程集合 rest ;
 ② 执行初始化操作:将 rest 设置为全部进程, $\text{currentavail}[*] = \text{available}[*]$, $\text{possible} = \text{true}$;
 ③ 保持 $\text{possible} = \text{true}$,从进程集合 rest 中找出满足下列条件的进程 P_k :

$$\text{claim}[k, *] - \text{allocation}[k, *] \leq \text{currentavail}[*]$$

- ④ 如找到满足上述条件的进程,释放进程 P_k 所占用的资源,并执行以下操作:

$$\text{currentavail}[*] = \text{currentavail}[*] + \text{allocation}[k, *]$$

把 P_k 从进程集合中去掉,即 $\text{rest} = \text{rest} - \{P_k\}$;

- ⑤ 否则 $\text{possible} = \text{false}$,停止执行本算法;
 ⑥ 最后,查看进程集合 rest ,若其为空集则返回安全标记;否则返回不安全标记。

下面通过例子来说明银行家算法用于检查系统所处状态是否安全。假设系统中共有 5 个进程 $\{P_0, P_1, P_2, P_3, P_4\}$ 和 A、B、C 三类资源;A 类资源共有 10 个,B 类资源共有 5 个,C 类资源共有 7 个。在时刻 T_0 ,系统资源分配情况如下:

进 程	allocation			claim			available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

每个进程目前所需要的资源为 $C_{ki} - A_{ki}$,如下所示:

进 程	$C_{ki} - A_{ki}$		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

(1) T_0 时刻的安全序列

可以断言, 系统目前处于安全状态, 因为 T_0 时刻的序列 $\{P_1, P_3, P_4, P_2, P_0\}$ 能满足安全性条件。 T_0 时刻的安全序列如下所示:

资源 进程	currentavail			$C_{ki} - A_{ki}$			allocation			currentavail + allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	3	3	2	1	2	2	2	0	0	5	3	2	true
P_3	5	3	2	0	1	1	2	1	1	7	4	3	true
P_4	7	4	3	4	3	1	0	0	2	7	4	5	true
P_2	7	4	5	6	0	0	3	0	2	10	4	7	true
P_0	10	4	7	7	4	3	0	1	0	10	5	7	true

(2) 进程 P_1 请求资源

假定进程 P_1 又要申请 1 个 A 类资源和 2 个 C 类资源, 为了判别此申请能否立即获得准许, 按照银行家算法进行检查:

$$\text{request1}(1, 0, 2) \leq C_{k1} - A_{k1}(1, 2, 2)$$

$$\text{request1}(1, 0, 2) \leq \text{available}(3, 3, 2)$$

系统尝试先为进程 P_1 分配资源, 修改 available、allocation 和 $C_{ki} - A_{ki}$, 得到如下所示的新状态:

进 程	allocation			$C_{ki} - A_{ki}$			available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

再用安全性测试算法检查系统此时的状态是否安全, 得到如下所示的进程 P_1 申请资源时的安全性分析表。

资源 进程	currentavail			$C_{ki} - A_{ki}$			allocation			currentavail + allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	2	3	0	0	2	0	3	0	2	5	3	2	true
P_3	5	3	2	0	1	1	2	1	1	7	4	3	true
P_4	7	4	3	4	3	1	0	0	2	7	4	5	true
P_0	7	4	5	7	4	3	0	1	0	7	5	5	true
P_2	7	5	5	6	0	0	3	0	2	10	5	7	true

可见能够找到一个安全序列 $\{P_1, P_3, P_4, P_0, P_2\}$, 因此, 系统此时处于安全状态, 可正式把资源分配给进程 P_1 。

(3) 进程 P_4 请求资源

如果进程 P_4 发出资源请求, 系统按照银行家算法进行检查:

$$\text{request}_4(3, 3, 0) \leq C_{k4} - A_{k4}(4, 3, 1)$$

$$\text{request}_4(3, 3, 0) > \text{available}(2, 3, 0)$$

由于这时可用系统资源不足, 申请被系统拒绝, 令进程 P_4 等待。

(4) 进程 P_0 请求资源

如果进程 P_0 申请资源, 系统按照银行家算法进行检查:

$$\text{request}_0(0, 2, 0) \leq C_{k0} - A_{k0}(7, 3, 1)$$

$$\text{request}_0(0, 2, 0) \leq \text{available}(2, 3, 0)$$

系统先为进程 P_0 进行尝试性分配, 修改相应的数据, 得到中间结果如下:

资源 进程	allocation			$C_{ki} - A_{ki}$			available		
	A	B	C	A	B	C	A	B	C
P_0	0	3	0	7	2	3	2	1	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

此时, 利用安全性测试算法检查发现: 系统能满足进程 P_0 的资源请求 $(0, 2, 0)$, 但可以看出剩余资源已不能满足任何进程的新需求, 故系统已处于不安全状态, 不能为进程 P_0 分配资源。

3.6.4 死锁检测和解除

1. 资源分配图和死锁定理

对资源的分配加以适当限制可防止和避免死锁的发生, 但不利于进程对系统资源的充分共享。解决死锁的另一条途径是死锁检测和解除, 这种方法对资源的分配不施加任何限制, 也不采取死锁避免措施, 系统定时地运行“死锁检测”程序, 判断系统内是否已经出现死锁, 如果检测到系统已死锁, 再采取措施解除它。这种方法的难点在于: 要确定何时运行死锁检测算法, 如果这一算法执行得很频繁, 将会浪费处理器时间; 如果这一算法执行得太稀疏, 则死锁进程和系统资源会一直被锁定。

设某个计算机系统中有多种资源和多个进程, 每个资源类用一个方框表示, 方框中的黑圆点表示此资源类中的各个资源, 每个进程用一个圆圈来表示, 用有向边表示进程申请资源和资源被

分配的情况。约定 $P_i \rightarrow R_j$ 为请求边, 表示进程 P_i 申请资源类 R_j 中的一个资源得不到满足而当前处于等待 R_j 类资源的状态, 这条有向边从进程开始指向资源类方框的边缘。反之 $R_j \rightarrow P_i$ 为分配边, 表示 R_j 类中的一个资源已被进程 P_i 占用, 故此有向边从方框内的某个黑圆点出发指向进程。如图 3.7 所示是进程 - 资源分配图的例子, 其中共有 3 个资源类, 每个进程的资源占有和申请情况已在图中表示出来。在此例中, 由于存在占有和等待资源的环路, 导致一组进程永远处于等待资源的状态, 因而发生了死锁。

在进程 - 资源分配图中存在环路并不一定必发生死锁, 因为循环等待资源仅是死锁发生的必要条件, 而不是充分条件, 如图 3.8 所示便是一个有环路而无死锁的例子。虽然进程 P_1 和进程 P_3 分别占有资源 R_1 和 R_2 , 并且等待另一个资源 R_2 和另一个资源 R_1 而形成环路, 但进程 P_2 和进程 P_4 分别占有资源 R_1 和 R_2 各一个, 它们所申请的资源已得到全部满足, 因而, 能在有限时间内归还已占有的资源。于是进程 P_1 和进程 P_3 分别获得另一个所需资源, 这时进程 - 资源分配图中已减少两条请求边, 环路不再存在, 系统中就不存在死锁。

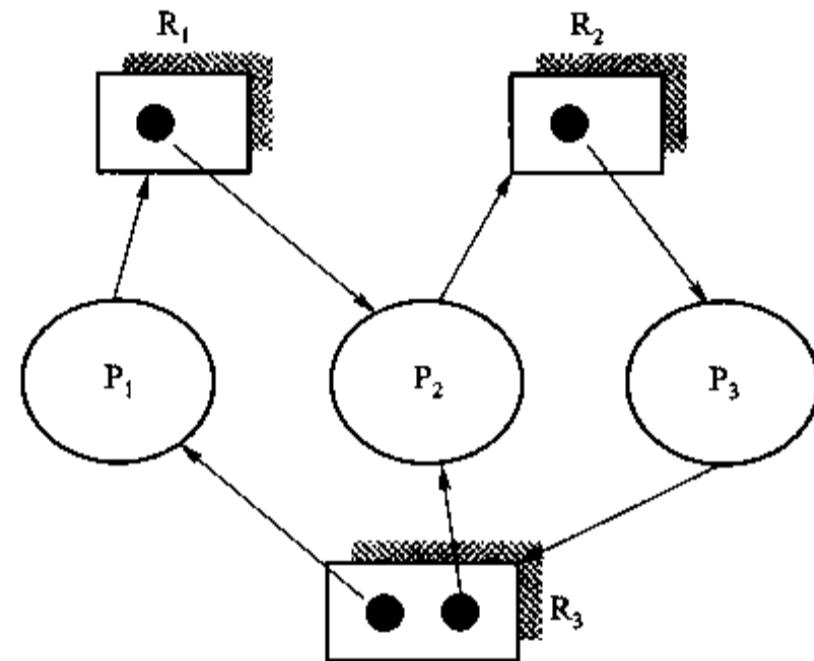


图 3.7 进程 - 资源分配图的例子

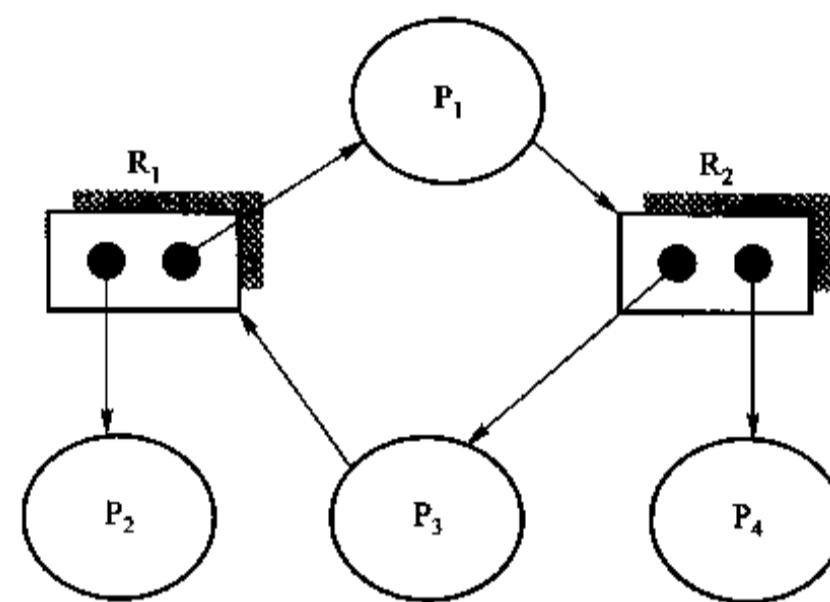


图 3.8 有环路而无死锁的例子

可利用下述步骤运行一个“死锁检测”程序, 对进程 - 资源分配图进行分析和简化, 以此方法来检测系统是否处于死锁状态。

- (1) 如果进程 - 资源分配图中无环路, 此时系统没有发生死锁;
- (2) 如果进程 - 资源分配图中有环路, 且每个资源类中仅有一个资源, 则系统中发生死锁, 此时, 环路是系统发生死锁的充分和必要条件, 环路中的进程就是死锁进程;
- (3) 如果进程 - 资源分配图中有环路, 且所涉及的资源类中有多个资源, 则环路的存在只是产生死锁的必要条件而不是充分条件, 系统未必会发生死锁。如果能在进程 - 资源分配图中找出一个既不阻塞又与其他进程因请求资源相关联的进程, 它在有限的时间内有可能获得所需资源类中的资源而继续执行, 直到运行结束, 再释放其所占有的全部资源。相当于消去进程 - 资源分配图中此进程的所有请求边和分配边, 使之成为孤立点。接着可使进程 - 资源分配图中另一

一个进程获得前面进程释放的资源而继续执行,直到完成后释放其所占用的所有资源,相当于又消除图中的若干请求边和分配边。如此往复,经过一系列简化后,若能消去图中的所有边,使所有进程成为孤立点,则此进程-资源分配图是可完全简化的;否则称此图是不可完全简化的。系统处于死锁状态的充分条件是:当且仅当此状态的进程-资源分配图是不可完全简化的,这一充分条件称为死锁定理。

2. 死锁的检测和解除方法

下面介绍一种具体的死锁检测方法,所定义的数据结构与银行家安全性测试算法类似。

- (1) available 是长度为 m 的向量,说明每类资源中可供分配的资源数目。
- (2) allocation 是 $n \times m$ 矩阵,说明已分配给每个进程的每类资源数目。
- (3) request 是 $n \times m$ 矩阵,表明当前每个进程对每类资源的申请数目。
- (4) currentavail 是长度为 m 的工作向量。
- (5) finish 是长度为 n 的布尔型工作向量。

令 $k = 1, 2, \dots, n$, 死锁检测算法的步骤如下:

- (1) currentavail = available;
- (2) 如果 $\text{allocation}[k, *] \neq 0$, 令 $\text{finish}[k] = \text{false}$; 否则 $\text{finish}[k] = \text{true}$;
- (3) 寻找一个 k , 应满足条件:

$(\text{finish}[k] == \text{false}) \& \& (\text{request}[k, *] \leq \text{currentavail}[*])$

若找不到这样的 k , 则转向步骤(5);

- (4) 修改 $\text{currentavail}[*] = \text{currentavail}[*] + \text{allocation}[k, *]$; $\text{finish}[k] = \text{true}$; 然后转向步骤(3);
- (5) 如果存在 $k (1 \leq k \leq n)$, $\text{finish}[k] = \text{false}$, 则系统处于死锁状态, 并且 $\text{finish}[k] = \text{false}$ 的 P_k 是处于死锁的进程。

死锁的检测和解除往往配套使用, 当死锁被检测到之后, 采用各种方法解除系统的死锁, 常用的方法有资源剥夺法、进程回退法、进程撤销法和系统重启法。

(1) 结束所有进程的执行, 并重新启动操作系统。这种方法很简单, 但先前的工作全部作废, 损失很大。

(2) 撤销陷于死锁的所有进程, 解除死锁, 继续运行。

(3) 逐个撤销陷于死锁的进程, 回收其资源并重新分派, 直至死锁解除。但是究竟先撤销哪个死锁进程呢? 可选择符合下面条件之一的进程先撤销: CPU 消耗时间最少者、产生的输出最少者、预计剩余执行时间最长者、分得的资源数量最少者或优先级最低者。

(4) 剥夺陷于死锁的进程所占用的资源, 但并不撤销此进程, 直至死锁解除。可仿照撤销陷于死锁的进程那样来选择剥夺资源的进程。

(5) 根据系统保存的检查点, 让所有进程回退, 直到足以解除死锁, 这种措施要求系统建立保存检查点、回退及重启机制。

(6) 当检测到死锁时, 如果存在某些尚未卷入死锁的进程, 随着这些进程执行至结束, 有可

能释放出足够的资源来解除死锁进程的死锁。

尽管检测死锁是否出现和发现死锁后实现恢复的代价大于防止和避免死锁所花费的代价,但由于死锁不是经常出现的,因而这样做还是值得的。检测策略的代价依赖于死锁出现的频率,而恢复的代价是指处理器时间的损失。

3.7 Linux 同步机制和通信机制

3.7.1 Linux 内核同步机制

操作系统内核在执行过程中,由于以下原因会造成并发执行。(1) 中断及异步信号:随时可能打断正在执行的内核代码;(2) 可抢占:如果内核具有抢占性,运行的内核任务会被另一个任务所抢占;(3) 对称式多处理器系统:两个或多个 CPU 同时执行内核代码,访问同一共享数据结构。这就要求有同步机制来保证不出现竞争条件,Linux 提供多种内核同步机制。

1. 原子操作

原子操作在执行过程中保证不被打断,以防止简单的竞争条件的发生,确保操作结果的正确性,复杂的锁机制能够在原子操作的基础上构建。Linux 内核定义以下两类原子操作。

(1) 原子整数操作

针对整型变量,定义特殊数据类型 atomic_t 和专门的原子整型操作函数,典型的用途是实现计数器。以下是部分原子整型操作:atomic_int(int i)(初始化原子变量为 i)、atomic_read(v)(读整数值 v)、atomic_set(v,i)(把 v 设置成 i)、atomic_add(i,v)(把 v 增加量值 i)、atomic_sub(i,v)(把 v 减去量值 i)、atomic_sub_and_test(i,v)(从 v 中减去 i,当结果为 0 时,则返回 1,否则返回 0)、atomic_add_negative(i,v)(将 v 中增加 i,当结果为 0 时,则返回 1,否则返回 0)、atomic_inc(v)(将 v 加 1)和 atomic_dec(v)(将 v 减 1)等。

(2) 原子位图操作

针对指针变量所指定的任意一块主存区域的位序列进行操作。以下是部分原子位图操作:set_bit(int nr,void *addr)(设置位图地址 addr 的 nr 位)、clear_bit(int nr,void *addr)(清除位图地址 addr 的 nr 位)、change_bit(int nr,void *addr)(反转位图地址 addr 的 nr 位)和 test_bit(int nr,void *addr)(返回位图地址 addr 的 nr 位的值)等。

2. 内核信号量

在 Linux 系统中,也使用等待队列来实现内核使用的信号量机制。内核信号量 semaphore 定义如下:

```
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
```

```
};
```

(1) 资源计数器 count 表示可用的某种资源数,若为正整数则表示这些资源尚可用;若为 0 或负整数则表示资源已用完,且因申请资源而等待的进程数是 count 的绝对值。sema_init 宏用于初始化 count 为任意值,可以是二元信号量,也可以是一般信号量。在 DOWN 操作中, count 减 1 后的值若小于 0, 进程便进入等待队列; 在 UP 操作中, count 加 1 后的值若大于 0, 立即唤醒等待队列中的所有进程。

(2) sleepers 对等待当前临界资源的进程个数进行辅助计数。

(3) wait 用于存放等待队列链表的地址,此链表包含目前正在等待信号量(资源)而被阻塞的所有进程。

内核信号量上所定义的函数有 DOWN、DOWN_interruptible、DOWN_trylock 和 UP, 分别对应信号量的 P、V 操作。此外,还提供读者 - 写者信号量,相应的操作有 down_read 读者 down 操作、up_read 读者 up 操作、down_write 写者 down 操作及 up_write 写者 up 操作。

3. 等待队列

Linux 内核信号量采用非忙式等待来实现,当进程执行 DOWN 操作而等待时,将被放入等待信号量队列;事实上,并发进程同步时,只要等待条件不满足,就必须挂起,放入相应的等待队列。所以,等待队列是支持进程同步的重要数据结构,定义如下:

```
struct wait_queue {
    unsigned int compiler_warning;
    struct task_struct *task;           //指向等待进程的 PCB
    struct list_head task_list;         //等待队列链表
};
```

等待队列也是临界资源,对其进行修改时应加锁,以避免竞争条件。内核为每个事件都设立一个等待队列,当进程等待一个指定事件发生时,必须调用 add_wait_queue() 把自己加入此事件的等待队列,并通过 schedule() 交出 CPU。被等待的事件发生后,内核调用 remove_wait_queue() 将一个进程从等待队列中移出。

4. 关中断

一个正在对内核数据结构进行操作的进程可以被 I/O 设备中断,如果设备中断处理程序恰好也要访问这一数据结构,就会引起系统的不一致状态。解决这类问题的有效方法是关中断。关中断是把内核态执行的程序段作为一个临界区来保护的一种手段,主要保护中断处理程序也要访问的数据结构,如磁盘中断就需要被屏蔽。此外,关中断还能禁止内核抢占。为了防止死锁,内核在关中断期间不能执行阻塞操作。在对称式多处理器环境中,关中断只能防止来自本机其他中断处理程序的并发访问,需要引入自旋锁在禁止本地中断的同时,防止来自它机的并发访问。

5. 自旋锁

原子操作能够保证对变量操作的原子性,比如,原子加操作把读取和增加变量的操作包含在一个单步中执行,从而防止发生竞争。可是临界区却不像增加变量这样简单,有时它可以跨越多

个函数。例如,先从一个数据结构中读取数据,对其进行格式转换和解析,再把它写到另一个数据结构中。整个执行过程必须是原子的,在数据更新完毕前,不能有其他代码读取这些数据。显然,简单的原子操作对此无能为力,这就需要使用更为复杂的同步方法——锁来提供保护。

Linux 内核中最常见的锁是自旋锁,自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个已被锁住的自旋锁,那么此线程就会一直进行忙式等待(旋转),等待锁重新可用,期间这个 CPU 不能处理其他工作,同时等待其他 CPU 上运行的进程执行解锁操作。如果锁未被征用,请求锁的执行线程便可立刻锁住它,继续执行。在任意时刻,自旋锁都可以防止多个一个的执行线程同时进入临界区。

自旋锁是最简单的一种锁原语,锁的取值为 0 表示资源可用,锁的取值为 1 表示资源加锁。自旋锁很像二元信号量,但在具体实现上还是有区别的。若一个资源得到自旋锁的保护,另一个试图取得此资源的内核例程将保持忙式等待,直至资源被解锁。而在实现二元信号量时,不必循环等待信号量,而是进入等待队列,暂时放弃对资源的请求。

在 Linux 中,自旋锁变量 lock 被定义成 spinlock_t 类型,lock 成员只有最低位被使用,如果锁可用,lock 的最低位为 0;如果上锁,lock 的最低位为 1。lock 成员在初始化时被设为 0,即未上锁。自旋锁定义为:

```
typedef struct {
    volatile unsigned int lock;
} spinlock_t;

void spin_lock(spinlock_t * plock) {
    int flag;
    do {
        flag = plock->lock & 1;           //取出 lock 的第 0 位,若为 1,表明已上锁
        plock->lock = 1;                //上锁,将 lock 的第 0 位设置为 1
    } while(flag != 0);              //若已上锁,则循环测试,直到成功
}

void spin_unlock(spinlock_t * plock) {
    plock->lock &= ~1;             //开锁,将 lock 的第 0 位设置为 0
}
```

在对称式多处理器系统中,自旋锁最重要的特点是内核例程在等待锁被释放时一直占据着某个 CPU,用来保护那些只需简短访问数据结构的操作,如从双向队列链表中增加或删除一个元素。因此,在内核需要进程同步的位置,自旋锁使用得比较频繁。

自旋锁的上锁或解锁通过以下宏来实现:spin_lock(lock)/spin_unlock(lock)(获得/释放自旋锁)、spin_lock_irq(lock)/spin_unlock_irq(lock)(获得自旋锁并关闭中断/释放自旋锁并开放中断)、spin_lock_bh(lock)/spin_unlock_bh(lock)(获得自旋锁并关闭 bh 的执行/释放自旋锁并开放 bh 的执行)和 spin_lock_init(lock)(初始化自旋锁)等。

读者-写者自旋锁机制允许在内核中实现比基本自旋锁更大的并发度。每个读者-写者自旋锁包含一个 24 位的读者计数器和一个解锁标记,有三种状态:当计数器 = 0、标记 = 1 时,自旋锁释放,可以使用;当计数器 = 0、标记 = 0 时,自旋锁被一个写线程所获得;当计数器 > 0、标记 = 0 时,自旋锁已被若干读线程所获得。

因为自旋锁在同一时刻至多被一个执行线程持有,所以一个时刻只能有一个线程位于临界区内,这就为多处理器系统提供防止并发访问所需的保护机制。在单处理器系统中,编译操作系统时并不会加入自旋锁,它仅仅被当做设置内核抢占机制是否被启用的开关。

3.7.2 System V IPC 机制

1. Linux 的 System V IPC 系统调用

在 UNIX System V 系统中,把信号量、消息队列和共享主存资源统称为 IPC(InterProcess Communication, 进程间通信)资源,用于进程之间的通信。Linux 提供给用户的 IPC 资源是通过系统调用实现的,它们为应用进程提供三种服务:

- (1) 用信号量对进程所要访问的临界资源进行保护;
- (2) 用消息队列在进程之间以异步方式发送和接收消息;
- (3) 预留共享主存段以供进程之间交换和共享数据。

IPC 和管道之间的区别是,它允许同一机器上的许多进程互相通信,而不是仅限于两个进程,且管道有一个限制,两个通信进程必须是相关的,它们都有共同的祖先进程。System V IPC 没有这方面的限制,只需要一个经过协商的协议。

每个 IPC 资源都有一个 32 位的 IPC 关键字(IPC key)和一个 32 位的 IPC 标识符(IPC identifier),IPC 标识符由内核分配给 IPC 资源,在系统内部是唯一的,而 IPC 关键字可由程序员自由选择。当进程间需要通过 IPC 资源进行通信时,就要引用它的 IPC 标识符。

IPC 数据结构是在进程请求 IPC 资源(指信号量、消息队列或共享主存)时动态地创建的,任何进程都可以使用 IPC 资源,即使是祖先进程创建 IPC 资源,但某些进程并不由这个祖先进程派生而来,这样的进程也可以使用 IPC 资源。每个 IPC 资源都有一个 ipc_perm 数据结构记录 IPC 关键字、创建者和所有者的 UID 和 GID,及每个创建者、创建者群组和其他用户的读写许可权限,还有 IPC 的关键字。

IPC 资源中信号量、消息队列和共享主存的创建分别通过系统调用 semget()、msgget()、shmget() 函数来完成,这 3 个函数都以 IPC 关键字作为第一个参数,创建成功后返回 IPC 标识符,以后进程就可以引用这个 IPC 标识符对资源进行访问。用完 IPC 资源后必须使用 ipcrm 命令显式地释放 IPC 资源,否则 IPC 资源将永远驻留在主存中。

在创建 IPC 资源之后,可通过 xxxctl() 函数对 IPC 资源进行控制,这些函数为用户提供命令来获得和设置资源的状态信息。当一个 IPC 资源被创建后,除了可对其进行控制和处理之外,还可通过专用函数对其进行操作,如获得和释放一个 IPC 信号量、发送和接收一个 IPC 消息以及将共享主存段附加到进程的地址空间或把共享主存段从进程的地址空间剥离。

2. Linux 的 System V IPC 实现概要

(1) IPC 信号量

内核所提供的信号量资源是一个指针数组, semid_ds * semary[SEMMNI]数组中的每一个元素都是 struct semid_ds 结构类型的指针, 数组中的每一个元素描述一个 IPC 信号量资源, 系统最多提供 128 个信号量资源。内核在 IPC 信号量机制中, sys_semop() 函数的作用是增加或减少一个信号量集合中的信号量的值。当用户调用 semop() 函数时, 相应的内核函数将执行以下操作:

① 在访问临界资源之前, 等待信号量集合的信号量可用。这个操作将导致信号量集合中相应的信号量的计数值减 1。

② 在访问临界资源之后, 通知信号量集合释放相应的信号量。这个操作将导致信号量集合中相应的信号量的计数值加 1。

当进程进入临界区后, 如果因为某种原因而无法退出, 此时挂起在信号量等待队列上的等待进程将永远得不到机会运行, 为此, Linux 维护一个信号量数组 struct sem_undo * undo 结构的调整链表来解决这个问题。

(2) IPC 消息队列

应用程序可用 msgget() 函数创建一个消息队列, 然后用 msgsnd() 将消息插入消息队列的末尾, 用 msgrcv() 将消息移出消息队列, 用 msgctl() 来释放或改变消息队列的许可权。IPC 资源的一条消息由两部分组成: 一部分是消息的头部, 由 struct msg 结构描述; 另一部分是消息的正文, 正文区的长度由相应的结构成员给出。内核中代表 IPC 消息的数据结构是 struct msgid_ds。

因为消息的发送是异步的, 因此允许把一条消息写入消息队列旋即离开。但是, 消息队列有最大容量的限制, 当消息队列达到规定容量时, 如果还有向此队列发送消息的进程则必须等待, 直至队列中有消息被读取而腾出空间, 或者在这种情况下发送消息的尝试被立刻拒绝。当消息队列中没有要读取的消息时, 要么读消息的尝试被立即拒绝, 要么读进程进入相应的等待队列。

(3) IPC 共享主存

多个通信进程访问共享主存段时, 都要将其映射到自己地址空间中的一个虚拟地址范围内, 对于每个进程而言, 这个地址段可以不同; 在映射之后, 进程访问这个地址段就和访问本地空间一样, 无须使用读写系统调用。

① IPC 共享主存的数据结构

采用 IPC 共享主存进行进程间通信时, 要发出系统调用 shmget(), 由其调用内核函数获得一个共享主存段的 IPC 标识符 shmid。如果此共享主存段尚未存在, 则要创建它。用来描述 IPC 共享主存段的数据结构是 struct shmid_kernel。

当进程第一次访问所创建的这块共享主存段之前, 必须通过系统调用 shmat() 申请连接, 它要申请一个 vm_area_struct 结构专门用于映射共享主存段, 同时重新填写结构成员 vm_ops 的操

作函数,使之分别对应于共享主存段的打开、关闭以及缺页中断时的换入和换出操作,然后,把这个 vma 添加到属于这个 IPC 共享主存资源的共享主存段链表。最后,生成一个新的 vm_area_struct 结构,并返回这个共享主存段的地址。

② IPC 共享主存请求调页的过程

shmat() 函数虽然把共享主存段连接到进程上,但是它并未修改这个进程的页表,当进程对此共享主存段的一个单元进行访问时,就会发生缺页错误。由 Linux 存储器管理及实现可知,这时由内核请求调页函数 handle_pte_fault() 确定引起缺页的地址是在进程地址空间的内部,而且所对应的页表项为 NULL,所以它会调用函数 do_no_page() 执行,再调用 shm_swap_in() 函数来实现。此函数根据 struct shmid_kernel 成员 shm_pages 所引用的数组获得这个缺页地址所在的页对应的页表项,为其分配或找回页框,最后由 do_no_page() 函数将返回的物理地址放在进程页表中的相应位置。因此,在缺页处理结束时,进程所要访问的共享主存单元的内容已调入主存供进程间通信使用。

③ 剥离和释放 IPC 共享主存段

当进程间通信结束从而不再需要共享主存时,通过系统调用 shmdt() 将共享主存段从进程的地址空间剥离,以便释放用于管理这段主存的系统资源。虽然 shmdt() 执行后将共享主存从进程的地址空间分离,但被分离之后的这个共享主存仍然存在于系统之中,还要使用命令显式地把这个共享主存段从系统中删除。



Windows 2003 同步机制和通信机制

1. 同步机制

在 Windows 系统中,提供互斥对象、信号量对象和事件对象及相应的系统调用,用于进程和线程的互斥和同步。这些同步对象都有用户所指定的对象名称,不同进程用同样的对象名称来创建或打开对象,从而获得此对象在本进程的句柄。从本质上讲,这些同步对象的能力是相同的,其区别在于适用场合和效率有所不同。

互斥对象用于控制共享资源的互斥访问,在一个时刻,互斥对象只能被一个线程使用,其相关 API 包括:

- (1) CreateMutex: 创建一个互斥对象,返回对象句柄。
- (2) OpenMutex: 打开并返回一个已存在的互斥对象句柄。
- (3) ReleaseMutex: 释放对互斥对象的占用,使之可用。

信号量对象就是资源信号量,其初始值范围在 0 到指定的最大值之间,用于限制并发访问的线程数,其相关 API 包括:

- (1) CreateSemaphore: 创建一个信号量对象,在输入参数中指定初始值和量大值,返回对象句柄。
- (2) OpenSemaphore: 打开并返回一个已存在的信号量对象句柄。

(3) ReleaseSemaphore: 释放对信号量对象的占用,使之可用。

事件对象相当于触发器,可用于通知一个或多个线程某事件已出现,其相关 API 包括:

- (1) CreateEvent: 创建一个事件对象,返回对象句柄。
- (2) OpenEvent: 打开并返回一个已存在的事件对象句柄。
- (3) SetEvent 和 PulseEvent: 设置所指定的事件对象为可用状态。
- (4) ResetEvent: 设置所指定的事件对象为不可用状态。

对于这三种同步对象,系统提供两个统一的等待操作 WaitForSingleObject 和 WaitForMultipleObjects,前者可在指定的时间内等待指定对象转换为可用状态;后者可在指定的时间内等待多个对象转换为可用状态。

Windows 系统还提供一些与进程同步有关的机制,如临界区对象和互锁变量访问 API 等。临界区对象只能用于在同一进程中所使用的临界区,同一进程中各线程对它的访问是互斥进行的,变量被说明为 CRITICAL_SECTION 类型。相关的 API 有 InitializeCriticalSection(对临界区对象进行初始化)、EnterCriticalSection(等待临界区的使用权,得到使用权则返回)、TryEnterCriticalSection(非等待方式申请临界区的使用权,申请失败时返回 0)、LeaveCriticalSection(释放临界区的使用权)和 DeleteCriticalSection(释放与临界区对象相关的所有系统资源)。

互锁变量访问 API 相当于硬件指令,用于整型变量操作,可避免线程切换对操作的连续性产生影响。这组 API 包括 InterlockedExchange(32 位数据的先读后写原子操作)、InterlockedCompareExchange(根据比较结果进行赋值的原子操作)、InterlockedExchangeAdd(先加后存结果的原子操作)、InterlockedDecrement(先减 1 后存结果的原子操作)和 InterlockedIncrement(先加 1 后存结果的原子操作)。

2. 通信机制

Windows 系统的信号(signal)是一种低级通信方式,进程可以发送信号,每个进程都有指定的信号处理程序,信号通信是单向和异步的。存在两组与信号相关的 API,分别处理不同的信号。

(1) SetConsoleCtrlHandler 和 GenerateConsoleCtrlEvent

前者定义或取消本进程的信号处理程序中的用户定义程序,例如,默认每个进程在收到 CTRL_C_EVENT 时都有一个信号 Ctrl + C 的处理程序来处理,可利用 SetConsoleCtrlHandler 调用来忽略或恢复对 Ctrl + C 的处理;后者可发送信号到与本进程共享同一控制台的控制台进程组。

(2) signal 和 raise

前者用于设置中断信号处理程序,后者用于发送信号,这组系统调用所处理的信号与 UNIX 系统相同,有非正常终止、浮点错、非法指令、非法存储访问、终止请求等。

Windows 系统的共享存储区(shared memory)可用于进程间的大数据量通信,通信进程可任意读写共享存储区,也可在共享存储区上使用任意数据结构。进程使用共享存储区时,需要互斥

和同步机制来确保数据的一致性。Windows 采用文件映射(file mapping)机制来实现共享存储区,进程可以把整个文件映射为进程虚拟地址空间的一部分来访问。与共享存储区相关的 API 有 CreateFileMapping(为指定文件创建一个文件映射对象)、OpenFileMapping(打开一个命名的文件映射对象)、MapViewOfFile(把文件映射到本进程的地址空间)、FlushViewOfFile(把映射地址空间的内容写入物理文件)、UnmapViewOfFile(拆除文件与本进程地址空间的映射关系)和 CloseHandle(关闭文件映射对象)。

Windows 系统的管道(pipe)是一条在进程间以字节流方式传送的通信通道,它利用系统核心缓冲区来实现单向通信,常用于命令行所指定的 I/O 重定向和管道命令。与 UNIX 系统类似,提供无名管道和命名管道,但安全机制更为完善。利用 CreatePipe 创建无名管道,并得到两个读写句柄,然后用 ReadFile 和 WriteFile 进行无名管道的读写。命名管道是服务器进程与客户进程之间的通信通道,可实现不同机器上的进程通信,采用客户 - 服务器模式连接本机或网络中的两个进程。利用 CreateNamePipe 在服务器端创建命名管道,ConnectNamePipe 在服务器端等待客户进程的请求,CallNamePipe 从客户进程建立与服务器的管道连接,ReadFile、WriteFile(阻塞方式)、ReadFileEx 和 WriteFileEx(非阻塞方式)用于命名管道读写。

Windows 系统的邮件槽(mailslot)是一种不定长和不可靠的单向消息通信机制,消息的发送无须接收方做准备,随时可以发送。邮件槽采用客户 - 服务器模式,只能从客户进程发往服务器进程。服务器进程负责创建邮件槽,它可从邮件槽中读取消息,而客户进程可利用邮件槽的名字向它发送消息。相关的 API 有 CreateMailslot(服务器创建邮件槽)、GetMailslotInfo(服务器查询邮件槽信息)、SetMailslotInfo(服务器设置读操作的等待时限)、ReadFile(服务器读邮件槽)、CreateFile(客户端打开邮件槽)和 WriteFile(客户端发送消息)。

Windows 系统的套接字(socket)是一种网络通信机制,通过两络在不同计算机的进程之间双向通信。套接字所采用的数据格式可以是可靠的字节流或不可靠的报文,通信模式可以是客户 - 服务器模式或对等模式。为了实现不同操作系统上的进程通信,需要约定两络通信时不同层次的通信过程和信息格式。TCP/IP 是广泛使用的网络通信协议,Windows 系统中的套接字规范称为 Winsock,它除了支持标准的 BSD 套接字之外,还实现独立于协议的 API,支持多种网络通信协议。

本章小结

把操作系统看做用户接口、资源管理者、虚拟机等,分三种观点来观察操作系统,这是静态的观点,这种观点没有把进程/线程在系统中执行的本质过程、内在联系和状态变化揭示出来。实际上,在操作系统所提供的运行环境中,多个进程/线程共享同一套计算机系统资源,它们不能独立运行,相互之间必然会发生交互和制约关系,系统控制进程/线程的执行是一个动态的过程。本章指出:操作系统可看做由多个独立运行的进程及一个对诸进程进行控制和协调的进程所组成,故可根据进程/线程交互的动态观点来观察操作系统。

在多道程序设计系统中,同一时刻可能有多个进程/线程,它们之间存在两种基本关系:竞争关系和协作关系。并发进程/线程可能需要竞争资源,互斥是协调进程/线程间竞争关系的一种手段。为了避免出现竞争条件,引入临界区的概念以解决进程互斥问题。为了完成同一任务,某些进程/线程需要分工协作,同步是协调进程/线程间协作关系的一种手段。进程同步的主要目的是使协作的并发进程之间能够有效地共享资源和协同工作,从而使进程的执行过程具有可再现性和执行结果的唯一性。

进程的低级通信机制主要有:原子操作、锁机制、信号量和P、V操作及管程;进程的高级通信机制有:消息传递、共享主存和管道机制等。本章用低级递信机制解决这些问题,如生产者-消费者问题、读者-写者问题、5位哲学家吃通心面问题、睡觉的理发师问题,这些问题是操作系统中并发进程/线程相互制约和内在关联的一种抽象,了解它们就能更好地理解操作系统的动态、并发、复杂的本质。每当研制一种新的同步机制,往往要用经典问题作为试金石,看是否能很好地解决这些问题,从理论上说,各种同步机制都是等价的,每一种机制都可以用另一种机制来实现,但是在实际的系统中,信号量与P、V操作、消息传递、共享主存等方式用得最多。

死锁是系统中一组并发进程因等待其他进程所占有的资源而永远不能向前推进的僵化状态,对操作系统十分有害。系统产生死锁有4个必要条件:互斥条件、占有并等待条件、不剥夺条件和循环等待条件。解决死锁问题有三种策略和方法:死锁防止、死锁避免、死锁检测和解除。死锁的防止是指系统预先确定资源分配策略,进程按此规定来申请和使用资源,保证死锁的一个必要条件不会被满足,使得系统不发生死锁,其缺点是资源利用率低,或对资源使用的限制过严;死锁的避免涉及动态地分析和检测新的资源请求和资源的分配情况,以确保系统始终处于安全状态,放宽资源的使用条件,银行家算法是著名的死锁避免算法,但缺乏实用价值;死锁的检测和解除表明操作系统总是同意对资源的申请,对资源的分配不施加任何限制,允许系统发生死锁,但必须建立检测机制选择安全检测算法,周期性地检测是否发生死锁,若有则将其检测出来再采取措施解除死锁。虽然死锁检测对于资源的使用不施加任何限制,但其实现开销较大。

习题三

一、思考题

1. 试述顺序程序设计的特点以及采用顺序程序设计的优、缺点。
2. 试述并发程序设计的特点以及采用并发程序设计的优、缺点。
3. 程序并发执行为什么会失去封闭性和结果可再现性?
4. 解释并发性与并行性。
5. 解释可再入程序与可再用程序。
6. 解释并发进程的无关性和交互性。
7. 并发进程的执行可能产生与时间有关的错误,试各举一例来说明与时间有关错误的两种表现形式。
8. 解释进程的竞争关系和协作关系。

9. 试述进程的互斥和同步两个概念之间的异同。
10. 什么是临界区和临界资源？临界区管理的基本原则是什么？
11. 试述 Dekker 算法实现临界区互斥的原理。
12. 试述 Peterson 算法实现临界区互斥的原理。
13. 哪些硬件设施可以实现临界区管理？简述其用法。
14. 什么是信号量？如何对其进行分类？
15. 为什么 P、V 操作均为不可分割的原语操作？
16. 从信号量和 P、V 操作的定义可以获得哪些推论？
17. 何谓管程？它有哪些属性？
18. 试述管程中的条件变量的含义和作用。
19. 试比较管程与进程的不同点。
20. 试比较 Hanson 和 Hoare 两种管程实现方法。
21. 已经有信号量和 P、V 操作可用做同步工具，为什么还要有消息传递机制？
22. 试述信件、信箱和间接通信原语。
23. 简述消息缓冲通信机制的实现思想。
24. 什么是管道(pipeline)？如何通过管道机制实现进程间通信？
25. 什么是消息队列机制？试述其工作原理。
26. 试述信号通信机制及其实现。
27. 试述进程的低级通信工具和高级通信工具。
28. 什么是死锁？什么是饥饿？试举日常生活中的例子加以说明。
29. 试述产生死锁的必要条件。
30. 列举死锁的各种防止策略。
31. 何谓银行家算法？试述其基本思想。
32. 解释：进程—资源分配图、死锁判定法则、死锁定理。
33. 系统有输入机和打印机各一台，现有两个进程都要使用它们，采用 P、V 操作实现请求使用和归还资源后，还会产生死锁吗？请说明理由。若是，则给出防止死锁的方法。
34. 假设 3 个进程共享 4 个资源，每个进程一次只能申请/释放一个资源，每个进程最多需要两个资源，证明此系统不会产生死锁。
35. 有 20 个进程，竞争使用 65 个同类资源，申请方式是逐个进行的，一旦某进程获得所需要的全部数量的资源，立即归还所有资源。若每个进程最多使用 3 个资源，问系统会产生死锁吗？为什么？
36. 在 5 个哲学家就餐问题中，如果至少有一个左撇子或右撇子，则他们的任意就座安排是否可以避免死锁，为什么？
37. 在 5 个哲学家就餐问题中，如果至少有一个左撇子或右撇子，则他们的任意就座安排是否可以防止饥饿，为什么？
38. 一台计算机有 8 台磁带机，被 N 个进程竞争使用，每个进程可能需要 3 台磁带机。请问 N 值为多少时，系统没有死锁的危险，并说明原因。
39. 有一个进程尚未获得任何资源，现在它开始申请资源，试问此进程是否会进入死锁？通过例子对此加以说明。

40. 使用上锁法实现进程的互斥时,有时会导致饥饿状态,试说明之。
41. 一个系统会处于既不死锁但也不安全的状态吗?试分析之。
42. 某系统有 m 个同类资源供 n 个进程共享,若每个进程最多申请 x 个资源($1 \leq x \leq m$),推导出系统不发生死锁(n 、 m 、和 x)的关系式。
43. 什么是竞争条件?试解释之。
44. 什么是忙式等待?试解释之。
45. 小河中铺了一串垫脚石用于过河,试说明什么是过河问题中的死锁?给出过河问题解决死锁的方法。
46. 在信号量机制中,若 P、V 操作不是原语,将会产生什么问题?
47. 试举出系统资源分配图有环锁和环而不锁的示例。
48. 针对死锁发生的必要条件,找出防止死锁的方法并填入下表。

死锁发生的必要条件	防止死锁的方法
互斥	
占有并等待	
不可剥夺	
循环等待	

二、应用题

1. 有 3 个并发进程:R 负责从输入设备读入信息块,M 负责对信息块进行加工处理;P 负责打印输出信息块。现提供:(1) 一个缓冲区,可放置 K 个信息块;(2) 两个缓冲区,每个缓冲区可放置 K 个信息块;试用信号量和 P、V 操作写出 3 个进程正确工作的流程。

2. 设有 n 个进程共享一个互斥段,如果:(1) 每次只允许一个进程进入互斥段;(2) 每次最多允许 m 个进程($m \leq n$)同时进入互斥段。试问:所采用的信号量的初值是否相同?信号量值的变化范围如何?

3. 有两个优先级相同的进程 P1 和 P2,其各自执行的操作如下,信号量 S1 和 S2 的初值均为 0。试问 P1、P2 并发执行后,x、y、z 的值各为多少?

P1()	P2()
$y = 1;$	$x = 1;$
$y = y + 3;$	$x = x + 5;$
V(S1);	P(S1);
$z = y + 1;$	$x = x + y;$
P(S2);	V(S2);
$y = z + y;$	$z = z + x;$

4. 现有 5 条语句,S1: $a = 5 - x$; S2: $b = a * x$; S3: $c = 4 * x$; S4: $d = b + c$; S5: $e = d + 3$;试用 Bernstein 条件证明语句 S2 和 S3 可以并发执行,而语句 S3 和 S4 不可并发执行。

5. 有一个阅览室,读者进入时必须先在一张登记表上登记,此表为每个座位列出一个表目,包括座位号、姓名,读者离开时要注销登记信息;假如阅览室共有 100 个座位。试用:(1)信号量和 P、V 操作;(2)管程,来实现用

户进程的同步算法。

6. 在一个盒子里,混装了数量相等的黑白围棋子。现在利用自动分拣系统把黑子、白子分开,设分拣系统有两个进程 P1 和 P2,其中进程 P1 拣白子;进程 P2 拣黑子。规定每个进程每次拣一子;当一个进程在拣时,不允许另一个进程去拣;当一个进程拣了一子时,必须让另一个进程去拣。试写出进程 P1 和 P2 能够正确并发执行的程序。

7. 管程的同步机制使用条件变量和 Wait 及 Signal,尝试为管程设计一种仅使用一条原语操作的同步机制。

8. 设在公共汽车上,司机和售票员的活动分别如下。

(1) 司机的活动:启动车辆;正常行车;到站停车。

(2) 售票员的活动:关车门;售票;开车门。

在汽车不断地到站、停车、行驶的过程中,这两个活动之间有什么同步关系?用信号量和 P、V 操作实现其同步。

9. 一个快餐厅有 4 类职员:(1) 领班:接受顾客点菜;(2) 厨师:准备顾客所需的饭菜;(3) 打包工:将烹制好的饭菜打包;(4) 出纳员:收款并提交食物。每位职员可被看做一个进程,试用一种同步机制写出能让 4 类职员正确并发工作的程序。

10. 在信号量 S 上执行 P、V 操作时,S 的值发生变化,当 $S > 0$ 、 $S = 0$ 、 $S < 0$ 时,其物理含义是什么?

11. (1) 两个并发进程并发执行,其中,A、B、C、D、E 是原语,试给出可能的并发执行路径。

```
process P() {
    A;
    B;
    C;
}
process Q() {
    C;
    D;
}
```

(2) 两个并发进程 P1 和 P2 并发执行,其程序代码分别如下。

```
P1() {
    while(true) {
        k = k * 2;
        k = k + 1;
    }
}
P2() {
    while(true) {
        print k;
        k = 0;
    }
}
```

若令 k 的初值为 5,让进程 P1 先执行两个循环,然后,进程 P1 和 P2 又并发执行一个循环,写出可能的打印值,指出与时间有关的错误。

12. 证明信号量与管程的功能是等价的。

(1) 用信号量实现管程;

(2) 用管程实现信号量。

13. 证明消息传递与管程的功能是等价的。

(1) 用消息传递实现管程;

(2) 用管程实现消息传递。

14. 证明信号量与消息传递是等价的。

(1) 用信号量实现消息传递;

(2) 用消息传递实现信号量。

15. 使用(1) 消息传递; (2) 管程, 实现生产者 - 消费者问题。
16. 试利用结构型信号量和 P、V 操作写出一个不会出现死锁的 5 位哲学家进餐问题的算法。
17. Dijkstra 临界区软件算法的描述如下。

```
enum {idle, wantin, incs} flag[n];
int turn;
enum{0,n-1}turn;
process Pi() { //i=0,1,...,n-1
    int j;
    do {
        flag[i] = wantin;
        while(turn != i)
            if(flag[turn] == idle) turn = i;
        flag[i] = incs;
        j = 0;
        while(j < n && (j == i || flag[j] != incs))
            j++;
    } while(j < n);
    {critical section};
    flag[i] = idle;
}
```

试说明此算法满足临界区管理原则。

18. 一个经典的同步问题: 吸烟者问题(Patil, 1971 年)。3 名吸烟者在同一个房间内, 还有一位香烟供应者。为了制造并抽掉香烟, 每位吸烟者需要三样东西: 烟草、纸和火柴, 供应者有丰富的货物提供。在 3 名吸烟者中, 第一个人有自己的烟草, 第二个人有自己的纸, 第三个人有自己的火柴。供应者随机地将两样东西放在桌子上, 允许一位吸烟者进行对健康不利的吸烟。当吸烟者完成吸烟后唤醒供应者, 供应者再把两样东西放在桌子上, 唤醒另一位吸烟者。试采用:(1) 信号量和 P、V 操作; (2) 管程, 编写他们同步工作的程序。

19. 有 4 个进程 $P_i (i=0,1,\dots,3)$ 和 4 个信箱 $M_j (j=0,1,\dots,3)$, 进程间借助相邻的信箱传递消息, 即 P_i 每次从 M_i 中取出一条消息, 经加工后送入 $M_{(i+1) \bmod 4}$, 其中 M_0, M_1, M_2, M_3 分别可存放 3, 3, 2, 2 条消息。在初始状态下, M_0 装了 3 条消息, 其余为空。试以 P、V 操作为工具, 写出 $P_i (i=0,1,\dots,3)$ 的同步工作算法。

20. 设有 3 组进程 P_i, Q_j, R_k , 其中 P_i, Q_j 构成一对生产者和消费者, 共享由 M_1 个缓冲区构成的循环缓冲池 $buf1$ 。 Q_j, R_k 构成另一对生产者和消费者, 共享由 M_2 个缓冲区构成的循环缓冲池 $buf2$ 。如果 P_i 每次生产一个产品投入 $buf1$, Q_j 每次从中取两个产品组装成一个并投入 $buf2$, R_k 每次从中取 3 个产品包装出厂。试用信号量和 P、V 操作写出其同步工作的程序。

21. 在一个实时系统中, 有两个进程 P 和 Q 循环工作。 P 每隔 1 s 由脉冲寄存器获得输入, 并将其累计到整型变量 W 上, 同时清除脉冲寄存器。 Q 每隔 1 小时输出这个整型变量的内容并将其复位。系统提供标准程序 INPUT 和 OUTPUT 供 I/O 操作时使用, 提供延时系统调用 Delay(seconds)。试写出两个并发进程循环工作的算法。

22. 系统有同类资源 m 个, 被 n 个进程共享, 问: 当 $m > n$ 和 $m \leq n$ 时, 每个进程最多可以请求多少个这类资源, 使系统一定不会发生死锁?

23. n 个进程共享 m 个资源, 每个进程一次只能申请/释放一个资源, 每个进程最多需要 m 个资源, 所有进程的资源需求总数少于 $m + n$ 个, 证明此系统此时不会产生死锁。

24. 试证明层次分配策略的变种按序分配策略能够防止系统发生死锁。

25. 一条公路两次横跨运河, 两个运河桥相距 100 m, 均带有闸门, 供船只通过运河桥, 运河和公路的交通均是单向的。运河上的运输由驳船担负, 在一艘驳船接近吊桥 A 时就拉汽笛警告, 若桥上无车辆, 吊桥就吊起, 直到驳船尾 P 通过此桥为止。对吊桥 B 也按照同样的次序进行处理。典型的驳船长度为 200 m, 当它在河上航行时是否会产生死锁? 若会, 请说明理由, 提出一种防止死锁的方法, 并用信号量来实现驳船的同步。

26. Jurassic 公园有一个恐龙博物馆和一个花园, 有 m 名旅客和 n 辆车, 每辆车仅能允许承载一名旅客。旅客在博物馆参观了一阵, 然后, 排队乘坐旅行车。当一辆车可用时, 它载入一名旅客, 再绕花园行驶任意长的时间。若 n 辆车均已被旅客乘坐游玩, 则想坐车的旅客还需要等待; 如果一辆车已经空闲, 但没有要游玩的旅客了, 那么, 车辆要等待。试用信号量和 P、V 操作同步 m 名旅客和 n 辆车。

27. 现有 k 个进程, 其标号依次为 $1, 2, \dots, k$, 如果允许它们同时读文件 file, 但必须满足条件: 参加同时读文件的进程的标号之和要小于正整数 M ($k < M$), 请使用: (1) 信号量与 P、V 操作; (2) 管程, 编写出协调多进程读文件的程序。

28. 设当前的系统状态如下, 此时 $\text{Available} = (1, 1, 2)$ 。

进 程	Claim			Allocation		
	R ₁	R ₂	R ₃	R ₁	R ₂	R ₃
P ₁	3	2	2	1	0	0
P ₂	6	1	3	5	1	1
P ₃	3	1	4	2	1	1
P ₄	4	2	2	0	0	2

(1) 计算各个进程还需要的资源数 $C_{ki} - A_{ki}$ 。

(2) 系统是否处于安全状态, 为什么?

(3) 进程 P_2 发出请求向量 $\text{request}2(1, 0, 1)$, 系统能把资源分配给它吗?

(4) 若在进程 P_2 申请资源后, P_1 发出请求向量 $\text{request}1(1, 0, 1)$, 系统能把资源分配给它吗?

(5) 若在进程 P_1 申请资源后, P_3 发出请求向量 $\text{request}3(0, 0, 1)$, 系统能把资源分配给它吗?

29. 系统有 A、B、C、D 共 4 种资源, 在某时刻进程 P_0, P_1, P_2, P_3 和 P_4 对资源的占有和需求情况如下表所示, 试解答下列问题。

进 程	Allocation				Claim				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P ₀	0	0	3	2	0	0	4	4	1	6	2	2
P ₁	1	0	0	0	2	7	5	0				
P ₂	1	3	5	4	3	6	10	10				
P ₃	0	3	3	2	0	9	8	4				
P ₄	0	0	1	4	0	6	6	10				

(1) 系统此时处于安全状态吗?

(2) 若此时进程 P_1 发出 $\text{request1}(1, 2, 2, 2)$, 系统能分配资源给它吗? 为什么?

30. 把死锁检测算法用于下面的数据:

$$\text{Need} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}, \quad \text{Allocation} = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(1) 系统此时处于安全状态吗?

(2) 若第二个进程提出资源请求 $\text{request2}(0, 0, 1, 0)$, 系统能分配资源给它吗?

(3) 若第五个进程提出资源请求 $\text{request5}(0, 0, 1, 0)$, 系统能分配资源给它吗?

31. 考虑一个共有 150 个存储单元的系统, 按如下方式分配给 3 个进程, P_1 最大需求 70 个, 已占有 25 个; P_2 最大需求 60 个, 已占有 40 个; P_3 最大需求 60 个, 已占有 45 个。使用银行家算法, 以确定下面的任何一个请求是否安全。

(1) 进程 P_4 到达, P_4 最大需求 60 个, 最初请求 25 个。

(2) 进程 P_4 到达, P_4 最大需求 60 个, 最初请求 35 个。

如果安全, 请找出安全序列; 如果不安全, 请给出结果的分配情况。

32. 有一个仓库, 可存放 X、Y 两种产品, 仓库的存储空间足够大, 但要求: (1) 每次只能存入一种产品 X 或 Y; (2) 满足 $N < X$ 产品数量 - Y 产品数量 $< M$ 。其中, N 和 M 是正整数, 试用信号量与 P、V 操作实现产品 X 和 Y 的入库过程。

33. 有一个仓库可以存放 A、B 两种零件, 最大库容量各为 m 个。生产车间不断地取 A 和 B 进行装配, 每次各取一个。为了避免零件锈蚀, 按照先入库者先出库的原则。有两组供应商分别不断地供应 A 和 B, 每次一个。为了保证配套和库存合理, 当某种零件比另一种零件超过 n ($n < m$) 个时, 暂停对数量大的零件的进货, 集中补充数量少的零件。试用信号量和 P、V 操作正确地实现它们之间的同步关系。

34. 进程 A_1, A_2, \dots, A_{n_1} 通过 m 个缓冲区向进程 B_1, B_2, \dots, B_{n_2} 不断地发送消息。发送和接收工作符合以下几条规则。

(1) 每个发送进程每次发送一条消息, 写入一个缓冲区, 缓冲区的大小与消息长度相等;

(2) 对于每条消息, B_1, B_2, \dots, B_{n_2} 都需要接收一次, 并读入各自的数据区内;

(3) 当 m 个缓冲区已满时, 则发送进程等待; 当没有消息可读时, 接收进程等待。试用信号量和 P、V 操作编制正确控制消息的发送和接收的程序。

35. 某系统有 R_1 设备 3 台, R_2 设备 4 台, 它们被 P_1, P_2, P_3 和 P_4 进程共享, 且已知这 4 个进程均按以下顺序使用设备:

→申请 R_1 →申请 R_2 →申请 R_1 →释放 R_1 →释放 R_2 →释放 R_1

(1) 系统运行过程中可能产生死锁吗? 为什么?

(2) 若有可能, 请列举一种情况, 并画出表示此死锁状态的进程-资源图。

36. 独木桥问题 1: 东西向汽车驶过独木桥, 为了保证交通安全, 只要桥上无车, 则允许一方的汽车过桥, 待其全部过完后, 才允许另一方的汽车过桥。请用信号量和 P、V 操作写出汽车过独木桥问题的同步算法。

37. 独木桥问题 2: 在独木桥问题 1 中, 限制桥面上最多可以有 k 辆汽车通过。试用信号量和 P、V 操作写

出汽车过独木桥问题的同步算法。

38. 独木桥问题 3: 在独木桥问题 1 中, 要求保证东西方向交替通过一辆汽车。试用信号量和 P、V 操作写出汽车过独木桥问题的同步算法。

39. 独木桥问题 4: 在独木桥问题 1 中, 要求各方向的汽车串行过桥, 但当另一方提出过桥请求时, 应能阻止对方尚未上桥的后继车辆, 待桥面上的汽车过完桥后, 另一方的汽车开始过桥。试用信号量和 P、V 操作写出汽车过独木桥问题的同步算法。

40. 假设一个录像厅有 0、1 和 2 三种不同的录像片由观众选择放映。录像厅的放映规则为:

(1) 任一时刻最多只能放映一种录像片, 正在放映的录像片是自动循环放映的, 最后一名观众主动离开时结束当前录像片的放映。

(2) 选择当前放映录像片的观众可以立即进入, 允许同时有多名观众选择同一录像片观看, 同时观看的人数不受限制。

(3) 等待观看其他录像片的观众按到达顺序排队, 当一种新的录像片开始放映时, 所有等待观看此录像片的观众可依次进入录像厅同时观看。用一个进程代表一个观众, 实现观众进程观看录像函数 `Videoshow(int Vcd_id)`, 以遵守放映规则。Vcd_id 表示观众所选择的录像编号。要求用信号量和 P、V 操作写出同步活动的程序。

41. 修改读者 - 写者的同步算法, 使它对写者优先, 即一旦有写者到达, 后续的读者必须等待。

(1) 用信号量和 P、V 操作实现;

(2) 用管程实现。

42. 假定某计算机系统有 R_1 和 R_2 两类可再用资源(其中 R_1 有两个单位, R_2 有一个单位), 它们被进程 P_1 、 P_2 所共享, 且已知两个进程均以下列顺序使用两类资源。

→ 申请 R_1 → 申请 R_2 → 申请 R_1 → 释放 R_1 → 释放 R_2 → 释放 R_1 →

试求出系统运行过程中可能到达的死锁点, 并画出死锁点的进程 - 资源图。

43. 某工厂有两个生产车间和一个装配车间, 两个生产车间分别生产 A、B 两种零件, 装配车间的任务是把 A、B 两种零件组装成产品。两个生产车间每生产一个零件后都要分别把它们送到装配车间的货架 F_1 、 F_2 上, F_1 存放零件 A, F_2 存放零件 B, F_1 和 F_2 的容量均为可以存放 10 个零件。装配工人每次从货架上取一个 A 零件和一个 B 零件, 然后将其组装成产品。请用:

(1) 信号量和 P、V 操作进行正确管理;

(2) 管程进行正确管理。

44. 桌上有一只盘子, 每次只能放入一只水果。爸爸专向盘子中放苹果(apple), 妈妈专向盘子中放橘子(orange), 一个儿子专等吃盘子里的橘子, 一个女儿专等吃盘子里的苹果。写出能使爸爸、妈妈、儿子和女儿正确同步工作的管程。

45. 桌上有一只盘子, 最多可以容纳两个水果, 每次仅能放入或取出一个水果。爸爸向盘子中放苹果(apple), 妈妈向盘子中放橘子(orange), 两个儿子专等吃盘子里的橘子, 两个女儿专等吃盘子里的苹果。试用:(1)信号量和 P、V 操作;(2)管程, 来实现爸爸、妈妈、儿子、女儿间的同步与互斥关系。

46. 一组生产者进程和一组消费者进程共享 9 个缓冲区, 每个缓冲区可以存放一个整数。生产者进程每次一次性地向 3 个缓冲区中写入整数, 消费者进程每次从缓冲区取出一个整数。请用:(1)信号量和 P、V 操作;(2)管程, 写出能够正确执行的程序。

47. 设有 3 个进程 P、Q、R 共享一个缓冲区, P 进程负责循环地从磁带机读入一批数据并放入缓冲区, Q 进程负责循环地从缓冲区取出 P 进程所放入的数据进行加工处理并把结果放入缓冲区, R 进程负责循环地从缓冲

区读出 Q 进程所放入的数据并在打印机上输出。请用:(1)信号量和 P、V 操作;(2)管程,写出能够正确运行的程序。

48. 设计一条机器指令和一种与信号量机制不同的算法,使得并发进程对共享变量的使用不会出现与时间有关的错误。

49. 下述流程是解决两进程互斥访问临界区问题的一种方法。试从“互斥”(mutual exclusion)、“空闲让进”(progress)、“有限等待”(bounded waiting)等三方面讨论其正确性。如果它是正确的,则证明之;如果它不正确,请说明理由。

```

int c1,c2;
void p1() {
    remain section 1;
    do {
        c1 = 1 - c2;
    } while(c2 == 0);
    critical section; //临界区
    c1 = 1;
}
void p2() {
    remain section 2;
    do {
        c2 = 1 - c1;
    } while(c1 == 0);
    critical section; //临界区
    c2 = 1;
}
main() { //主程序
    c1 = 1;c2 = 1;
    cobegin
        p1();p2(); //进程 p1 和 p2 并发执行
    coend
}

```

50. 分析下列算法是否正确,为什么?

```

while(true) {
    key = true;
    do {
        swap(lock, key);
    } while(key == true);
    critical section; //临界区
    lock = false;
    other code;
}

```

51. 如下所示的并发执行的程序仅当数据装入寄存器后才能加 1。

```

const n = 50;
int tally;
void total() {
    for(int count = 1;count <= n;count++) tally++;
}

```

```

main() {
    tally = 0;
    cobegin
        total(); total();
    coend;
    writeln(tally);
}

```

- (1) 给出这个并发程序所输出的 tally 值的上限和下限。
(2) 若并发执行的程序可以是任意多个,这对于 tally 值的范围有何影响?

52. 举例说明下列算法不能解决互斥问题。

```

bool blocked[2];
enum {0,1} turn;
blocked[0] = blocked[1] = false;
turn = 0;
void P(int id) {
    blocked[id] = true;
    while(turn != id) {
        while(blocked[1 - id]);
        turn = id;
    }
    {CriticalSection};
    blocked[id] = false;
    {remainder}
}

```

cobegin

P[0];P[1];

coend;

53. 现有 3 个生产者 P_1, P_2, P_3 , 他们都要生产橘子汁, 每个生产者都已分别购得两种不同的原料, 待购齐第三种原料后就可配制成橘子汁装瓶出售。有一供应商能源源不断地供应糖、水、橘子精, 但每次只拿出一种原料放入容器中供应给生产者。当容器中有原料时, 需要这种原料的生产者可以取走, 当容器空时供应商又可放入一种原料。假定:

生产者 P_1 已购得糖和水;

生产者 P_2 已购得水和橘子精;

生产者 P_3 已购得糖和橘子精;

试用:(1) 管程;(2) 信号量与 P、V 操作,写出供应商和 3 个生产者之间能正确同步的程序。

54. 有一位材料保管员负责保管纸和笔若干。有 A、B 两组学生, A 组学生每人都备有纸, B 组学生每人都备有笔。任一名学生只要能得到其他一种材料就可以开始写信。有一个可以放置一张纸或一支笔的小盒, 当小盒中无物品时, 保管员就可任意放一张纸或一支笔供学生取用, 每次允许一名学生从中取出自己所需要的材料,

当学生从盒中取走材料后,允许保管员再存放一件材料,请用:(1)信号量与 P、V 操作;(2)管程,写出他们并发执行时能正确工作的程序。

55. 进程 A 向缓冲区 buffer 发送消息,当发出一条消息后要等待进程 B、C、D 都接收这条消息,进程 A 才能发送新消息。(1) 用信号量和 P、V 操作;(2) 管程,写出其同步工作的程序。

56. 试设计一个管程来实现磁盘调度的电梯调度算法。

57. 有 3 个进程 P_1, P_2, P_3 共享一个表格 F, P_1 对 F 只读不写, P_2 对 F 只写不读, P_3 对 F 先读后写。进程可同时读表格 F,但当有进程写时,其他进程则不能读和写。用信号量和 P、V 操作实现正确的读写。

58. 有 n 个进程将字符逐个读入一个容量为 80 个字符的缓冲区中($n > 1$),当缓冲区已满后,由输出进程 Q 负责一次性地取走这 80 个字符。这种过程循环往复,请用信号量和 P、V 操作写出 n 个读入进程(P_1, P_2, \dots, P_n)和输出进程 Q 能正确工作的动作序列。

59. 设儿童小汽车生产线上有一只大的储存柜,其中有 N 个槽(N 为 5 的整数倍数且其值不小于 5),每个槽可以存放一个车架或一个车轮。设有 3 组生产工人,其活动如下:

组 1 工人的活动

L1: 加工一个车架;

车架放入储存柜的槽中;

goto L1;;

组 2 工人的活动

L2: 加工一个车轮;

车轮放入储存柜的槽中;

goto L2;;

组 3 工人的活动

L3: 在槽中取一个车架;

在槽中取 4 个车轮;组装为一台小汽车

goto L3;;

试用(1)信号量及 P、V 操作;(2)管程,正确实现这三组工人的生产合作。

60. 某大型银行办理人民币储蓄业务,由 n 名储蓄员负责。每位顾客进入银行后先至取号机领取一个号,并且在等待区找到空沙发坐下等待叫号。取号机给出的号码依次递增,并假定有足够的空沙发容纳顾客。当一位储蓄员空闲下来,就呼叫下一个号。请用信号量和 P、V 操作正确编写储蓄员进程和顾客进程的程序。

61. 现有 100 名毕业生去甲、乙两家公司求职,两家公司合用一间接待室,其中甲公司招收 10 个人,乙公司招收 10 个人,招满为止。两家公司各有一位人事主管接待毕业生,每位人事主管每次只可接待一个人,其他毕业生在接待室外排成一列队伍等待。试用信号量和 P、V 操作实现人员招聘的过程。

62. 有一个电子转账系统共管理 10 000 个账户,为了向客户提供快速转账业务,有许多并发执行的资金转账进程,每个进程读取一行输入,其中,含有:贷方账号、借方账号、借贷的款项数。然后,把款项从贷方账号划转到借方账号上,这样便完成了一笔转账交易。写出进程调用 Monitor 以及 Monitor 控制电子资金转账系统的程序。

63. 某高校开设网络课程并安排上机实习,如果机房共有 $2m$ 台机器,有 $2n$ 名学生选课,规定:

(1) 每两名学生分成一组,并占用一台机器,协同完成上机实习;

(2) 仅当一组两名学生到齐,并且机房机器有空闲时,这组学生才能进入机房;

(3) 上机实习由一名教师检查,检查完毕,一组学生同时离开机房。

试用信号量和 P、V 操作模拟上机实习过程。

64. 某寺庙有小和尚和老和尚若干,水缸一只,由小和尚提水入缸供老和尚饮用。水缸可容水 10 桶,水取自同一口水井中。水井径窄,每次仅能容纳一只水桶取水,水桶总数为 3 个。每次放入、取出的水量仅为 1 桶,而且不能同时进行。试用一种同步工具写出小和尚和老和尚人水、取水的活动过程。

65. 假设缓冲区 buf1 和 buf2 都无限大,进程 P_1 向 buf1 写数据,进程 P_2 向 buf2 写数据,现要求 buf1 中的数据个数与 buf2 中的数据个数之差保持在[20,80]之间,请用信号量和 P、V 操作描述这一同步关系。

66. 在一个分页存储管理系统中,用 free[index] 数组记录每个页框状态,共有 n 个页框(index = 0, 1, ..., n -

}

coend.

试解释此算法的执行过程，并说明它能够保证进程互斥地进入临界区。

85. Eisenberg 和 McGuire 的 N 个并发进程的临界区互斥算法如下：

INITIALIZATION:

```
shared enum states {IDLE, WAITING, ACTIVE} flags[n - 1];
shared int turn;
turn = 0;
for (int index = 0; index < n; index++) {
    flags[index] = IDLE;
}

```

ENTRY PROTOCOL(for Process i):

```
do {
    flags[i] = WAITING;
    index = turn;
    while(index != i) {
        if(flag[index] != IDLE)      index = turn;
        else index = (index + 1) % n;
    }
    flags[i] = ACTIVE;
    index = 0;
    while((index < n) && ((index == i) || (flags[index] != ACTIVE))) {
        index = index + 1;
    }
} while(! ((index >= n) && ((turn == i) || (flags[turn] == IDLE))));
turn = i;
```

EXIT PROTOCOL (for Process i):

```
index = (turn + 1) % n;
while (flags[index] == IDLE) {
    index = (index + 1) % n;
}
turn = index;
flag[i] = IDLE;
```

(1) 试说明 n 个进程能正确实现临界区互斥算法的设计思想；

(2) 对每条语句或程序段进行注释。

其中, t_i 是这类临界资源的阈值, d_i 是这类临界资源的本次请求数。试回答通用型信号量机制的主要特点, 它适用于什么场合?

72. 下面是通用型信号量的一些特殊情况:(1) $SP(s,d,d)$, (2) $SP(s,1,1)$, (3) $SP(s,1,0)$, 试解释其物理含义及所起的作用。

73. 试利用通用型信号量机制解决读者-写者问题。

74. 试利用二元信号量机制解决生产者-消费者问题。

75. 试利用二元信号量机制来实现一般信号量。

76. 试利用关中断实现单处理器上的信号量机制。

77. 试利用 test & set 指令实现单处理器上的信号量机制。

78. Bakery 算法 (Lamport, 1974 年) 是解决 n 个进程访问临界区问题的一种方法。

bool choosing[n];

```
int number[n];
```

```
while(true) {
```

choosing[i] = true;

`number[i] = 1 + max(number[0], number[1], ..., number[n-1])`

choosing $\alpha[1] = \text{false}$.

```
for(int i=0;i< n;i++) {
```

```
while( obusizing[1] ) {
```

```
while((number[i])!=0) & & (number[i].i)<(number[i].j))
```

1

<critical section> :

number[i] = 0;

<remainder>:

七

(1) $(-1, b) \leq (-1)$ 定义为 $b \leq 0$ 。

(3) `checkbox[i]`和`checkbox[i].checked`可以被直接读写，但是不能被直接修改。

(3) 数组 arr_1 和 arr_2 都初始化为 $\{1\}$ 。而数组 arr_3 则初始化为 $\{1, 2\}$ 。

请用自然语言描述如何一算法，能说明

(1) B-1.....筛选器如何解决反向访问收尾段的问题的?

(2) Belady 算法是如何解决死锁问题的?

79. 面包店烹制面包和蛋糕,由 n 名销售员出售。当有顾客进店购买面包或蛋糕时,应先在取号机上取号,然后等待叫号,若有销售员空闲时,便呼叫下一个号。试用信号量和 P、V 操作写出 Bakery 算法的同步程序。

80. 试述 Hansen 方法实现管程所需要的 4 条原语 check()、wait()、signal() 及 release() 的功能。

81. 试写出 Hansen 方法实现管程所需数据结构和 4 条原语 check()、wait()、signal() 和 release() 的实现过程。

82. Hansen 管程方法实现读者 - 写者问题。

83. Hansen 管程方法实现生产者 - 消费者问题。

84. 荷兰数学家 T. Dekker 所提出的 Dekker 算法如下:

```

bool inside[2];
turn = 0 or 1;
inside[0] = false;
inside[1] = false;
cobegin
process P1 {
    inside[0] = true;
    while(inside[1]) {
        if(turn == 1) {
            inside[0] = false;
            while(turn == 1);
            inside[0] = true;
        }
    }
    临界区;
    turn = 1;
    inside[0] = false;
}
process P2 {
    inside[1] = true;
    while(inside[0]) {
        if(turn == 0) {
            inside[1] = false;
            while(turn == 0);
            inside[1] = true;
        }
    }
    临界区;
    turn = 0;
    inside[1] = false;
}

```

}

coend.

试解释此算法的执行过程，并说明它能够保证进程互斥地进入临界区。

85. Eisenberg 和 McGuire 的 N 个并发进程的临界区互斥算法如下：

INITIALIZATION:

```
shared enum states {IDLE, WAITING, ACTIVE} flags[n - 1];
shared int turn;
turn = 0;
for (int index = 0; index < n; index++) {
    flags[index] = IDLE;
}

```

ENTRY PROTOCOL(for Process i):

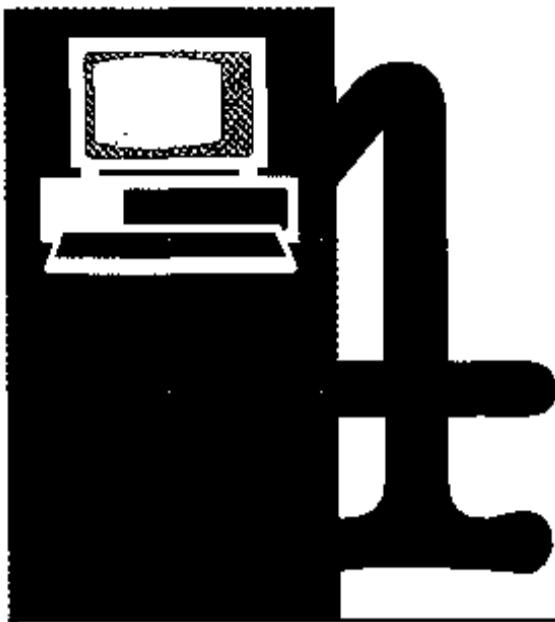
```
do {
    flags[i] = WAITING;
    index = turn;
    while(index != i) {
        if(flag[index] != IDLE)      index = turn;
        else index = (index + 1) % n;
    }
    flags[i] = ACTIVE;
    index = 0;
    while((index < n) && ((index == i) || (flags[index] != ACTIVE))) {
        index = index + 1;
    }
} while(! ((index >= n) && ((turn == i) || (flags[turn] == IDLE))));
turn = i;
```

EXIT PROTOCOL (for Process i):

```
index = (turn + 1) % n;
while (flags[index] == IDLE) {
    index = (index + 1) % n;
}
turn = index;
flag[i] = IDLE;
```

(1) 试说明 n 个进程能正确实现临界区互斥算法的设计思想；

(2) 对每条语句或程序段进行注释。



第四章

存储管理

存储管理是操作系统的重要组成部分,负责管理计算机系统的重要资源——主存储器。由于任何程序和数据必须占用主存空间才能得以执行和处理,因此,存储管理的优劣直接影响系统性能。主存储器对数据的存取比处理器处理数据的速度慢得多,硬件技术的不断发展还在进一步拉大这种距离,通过高速缓存可以部分缩小差距,但高效的主存管理仍然是操作系统设计中的重要课题。

主存空间一般分为两部分:一部分是系统区,用于存放操作系统内核程序和数据结构等;另一部分是用户区,用于存放应用程序和数据。存储管理对核心区和用户区都提供相应的支持和进行管理,当然也包括对辅助存储器(磁盘)空间的管理。尽管现代计算机的主存容量不断增大,但仍然不能保证有足够的空间支持大型应用和系统程序及数据的使用。因此,操作系统的任务之一是尽可能方便用户使用和提高主存的利用率。此外,有效的存储管理也是多道程序设计系统的关键支撑。具体地说,存储管理包含以下一些功能。

(1) 分配和去配

进程可请求对主存区的独占式使用,主存区的请求和释放即主存空间的分配和去配操作由存储管理完成。

(2) 抽象和映射

主存储器被抽象,使得进程认为分配给它的地址空间是一个大且连续地址所组成的数组,或者把主存储器抽象成二维地址空间,以支持模块化程序设计;同时建立抽象机制支持进程用逻辑地址来映射到物理主存单元,实现地址的转换。

(3) 隔离和共享

系统负责隔离已分配给进程的主存区,互不干扰免遭破坏,确保进程对存储单元的独占式使用,以实现存储保护功能;系统允许多个进程共享主存区,在这种情况下,超越隔离机制并授权进程允许共享访问,达到既能提高主存利用率又能共享主存某区内信息的目的。

(4) 存储扩充

物理主存容量不应限制应用程序的大小,主存和辅助存储器被抽象为虚拟主存,允许用户的

虚拟地址空间大于主存物理地址空间,存储管理自动在不同的存储层次中移动信息。

本章在介绍计算机存储器的层次之后,先后分析连续存储管理方法、页式和段式存储管理方法,再讨论虚拟存储管理系统,最后介绍 Linux 虚拟存储管理及 Windows 2003 虚拟存储管理。

■■■ 存 储 器

4.1.1 存储器的层次

目前,计算机系统均采用层次结构的存储子系统,以便在容量大小、速度快慢、价格高低等诸多因素中取得平衡点,获得较好的性能/价格比。计算机系统的存储器层次结构分为寄存器、高速缓存、主存储器、磁盘、磁带等 5 层。如图 4.1 所示,存储介质的访问速度由下而上越来越快,价格也越来越高。其中,寄存器、高速缓存和主存储器均属于操作系统存储管理的管辖范畴,掉电后它们所存储的信息不复存在;磁盘和磁带属于设备管理的管辖对象,它们所存储的信息将被持久性保存。

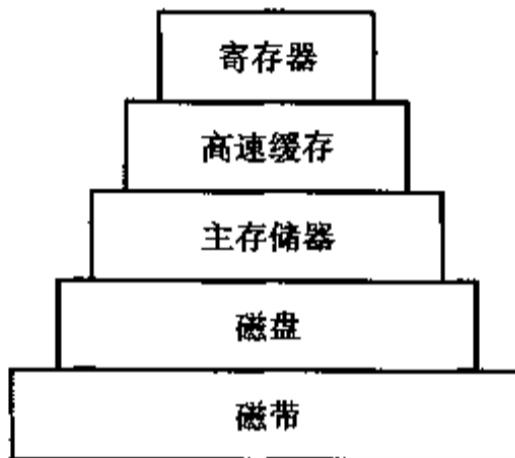


图 4.1 计算机系统存储器层次结构

可执行程序必须被保存在主存储器中,与设备相交换的信息也依托于主存地址空间。由于处理器在执行指令时的主存访问时间远大于其处理时间,所以,寄存器和高速缓存被引入来加快指令的执行。

寄存器是访问速度最快但价格最昂贵的存储器,其容量较小,一般以字为单位,一个计算机系统可能包括几十个寄存器,用于加速存储访问速度,如用寄存器存放操作数,或用做地址寄存器,或用做变址寄存器,以加快地址的转换速度。

高速缓存的容量较寄存器稍大,其访问速度快于主存。利用高速缓存来存放主存中经常访问的一些信息,以提高程序执行速度。目前,计算机的典型配置为:CPU 寄存器 1 KB,存取周期 1 ns;高速缓存 1 MB,存取周期 2 ns;主存储器 512 MB,存取周期 10 ns;磁盘 80 GB,存取周期 10 ms;磁带 100 GB,存取周期 100 s,这样的机器足可用于中小型科研项目的开发。多层次的存储体系十分有效和可靠,能达到很高的性能/价格比。

由于程序在执行和处理数据时往往存在顺序性和局部性,执行时并不需要将其全部调入主

存,仅调入当前使用的一部分,其他部分待需要时再逐步调入。这样,计算机系统为了容纳更多的作业,或为了处理更大批量的数据,可在磁盘上建立磁盘高速缓存以扩充主存储器的存储空间,计算程序和所处理的数据可装入磁盘高速缓存,操作系统自动实现主存储器和磁盘高速缓存之间的程序和数据的调进调出,从而向用户提供比实际主存容量大得多的存储空间。基于这个原理,就可以设计出多级层次式体系结构的存储子系统。

4.1.2 地址转换与存储保护

大多数应用程序、操作系统和实用程序都用高级程序设计语言或汇编语言编写,所编写的程序称为源程序,源程序中的符号名集合所限定的空间称为程序名字空间。源程序是不能被计算机直接运行的,需要通过如图 4.2 所示的 3 个阶段处理后才能装入主存运行。

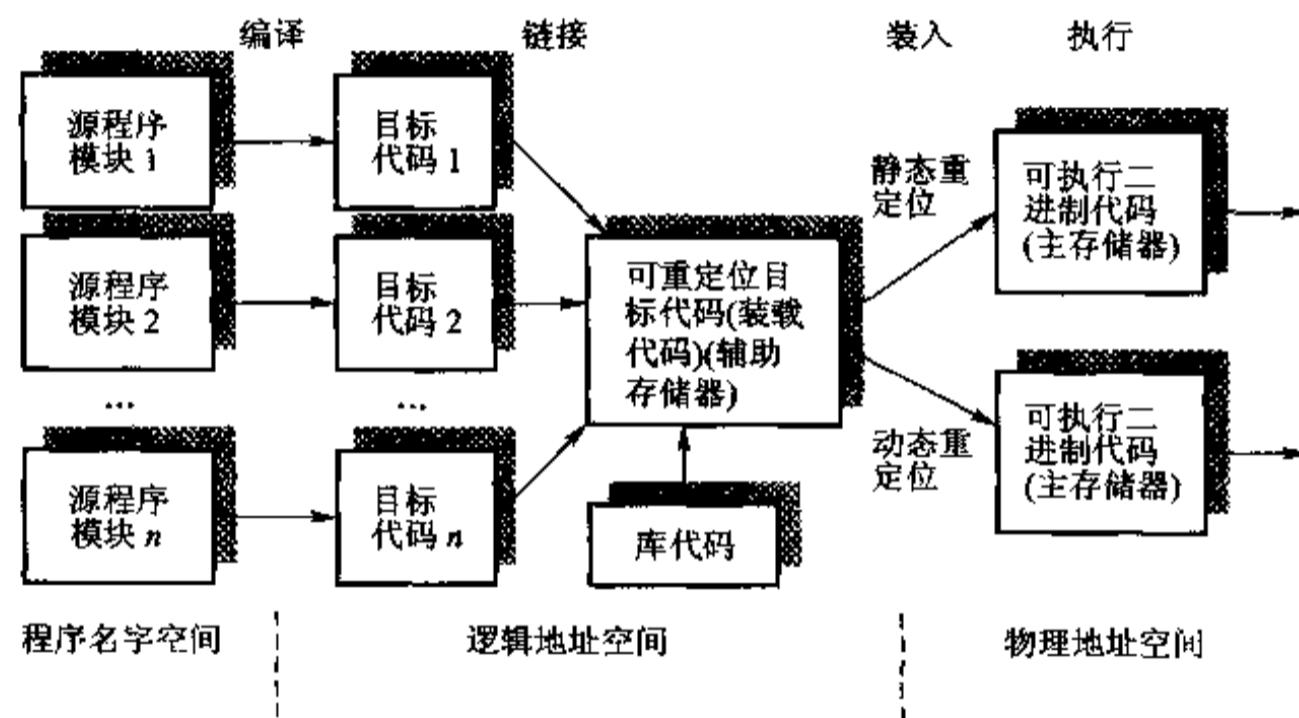


图 4.2 程序的编译、链接、装入和执行

1. 编译

源程序经过编译程序(compiler)或汇编程序(assembler)的处理来获得目标代码(也称目标模块)。一个程序可由独立编写且具有不同功能的多个源程序模块组成,在 C 程序设计模型中,至少分为 3 个程序模块:文本段、数据段和堆栈段。由于模块包含外部引用,即指向其他模块中的数据或指令的操作数地址,或包含对库函数的引用,编译程序负责记录引用的发生位置,编译或汇编的结果将产生相应的多个目标模块,每个目标模块都附有供引用使用的内部符号表和外部符号表。符号表中依次给出各个符号名及在本目标模块中的名字地址,在模块被链接时进行转换。例如,编写一个名为 simplecomputing 的源程序,其主程序 main 中有函数和子程序调用指令:求平方根 SQRT 和转子程序 SUB1, SQRT 是函数库中已被编译成可链接的目标模块的标准子程序,SUB1 是另一个模块中定义的已被编译成可链接的子程序,这时所调用的入口地址均是未知的;编译程序或汇编程序将在外部符号表中记录外部符号名 SQRT 和 SUB1,同时两条调用指令指向函数和子程序的位置。

2. 链接

链接程序(linker)的作用是把多个目标模块链接成一个完整的可重定位程序(其中包括应用程序要调用的标准库函数、所引用的其他模块中的子程序),需要解析内部和外部符号表,把对符号名的引用转换为数值引用,要将涉及名字地址的程序入口点和数据引用点转换为数值地址。仍采用上例,linker首先将主程序调入工作区,然后,扫描外部符号表,获得外部符号名 SQRT,用此名字从标准函数库中找出函数的 sqrt.o 并装入工作区,拼接在主程序的下面;SQRT 函数的主存位置就是调用 SQRT 指令的人口地址,将此指令代真;调用 SUB1 的链接过程与此相似,只是从另一个模块中找到 sub1.o 的位置并进行指令代真;经过链接处理后,主程序 main 与 SQRT 函数和 SUB1 子程序链接成完整的可重定位目标程序 simplecomputing.o。

可重定位目标程序又称装载代码模块,它存放于磁盘中,由于程序在主存中的位置不可预知,链接时程序地址空间中的地址总是相对某个基准(通常为 0)开始编号的顺序地址,称为逻辑地址或相对地址。

3. 装入

在加载一个装载代码模块之前,存储管理程序总会分配一块实际主存区给进程,装入程序(loader)根据指定的主存区首地址,再次修改和调整被装载模块中的逻辑地址,将逻辑地址绑定到物理地址,使之成为可执行二进制代码。这样,就可用逻辑地址来引用分配到的主存物理块内相应的物理地址,将再次修改和调整的装载代码模块复制到指定主存区中,以便进程在物理地址空间中执行。

磁盘中的装载代码模块所使用的是逻辑地址,其逻辑地址集合称为进程的逻辑地址空间。逻辑地址空间可以是一维的,这时逻辑地址限制在从 0 开始顺序排列的地址空间内;逻辑地址空间也可以是二维的,这时整个程序被分为若干段,每段都有不同的段号,段内地址从 0 开始顺序编址。进程运行时,其装载代码模块将被装入物理地址空间中,此时程序和数据的实际地址通常不可能同原来的逻辑地址一致。物理主存储器从统一的地址开始顺序编址的存储单元称为物理地址或绝对地址,物理地址的总体构成物理地址空间。需要注意的是,物理地址空间是由存储器地址总线扫描出来的空间,其大小取决于实际安装的主存容量。把逻辑地址转换(绑定)为物理地址的过程称为地址重定位、地址映射或地址转换,有以下两种方式。

1. 静态地址重定位

由装入程序实现装载代码模块的加载和地址转换,把它装入分配给进程的主存指定区域,其中的所有逻辑地址修改成主存物理地址,称静态重定位(static relocating address)。地址转换工作在进程执行前一次完成,无须硬件支持,易于实现,但不允许程序在执行过程中移动位置。这种技术只在早期单用户单任务系统中使用过。

2. 动态地址重定位

由装入程序实现装载代码模块的加载,把它装入分配给进程的主存指定区域,但对链接程序处理过的应用程序的逻辑地址则不做任何修改,程序主存起始地址被置入硬件专用寄存器——重定位寄存器,如图 4.3 所示。程序执行过程中,每当 CPU 引用主存地址(访问程序和数

据)时,由硬件截取此逻辑地址,并在它被发送到主存储器之前加上重定位寄存器的值,以便实现地址转换,称动态重定位(dynamic relocating address),地址转换推迟到最后的可能时刻,即进程执行时才完成。与静态地址重定位相比,动态地址重定位具有允许程序在主存中移动、便于程序共享和主存利用率高等优点。

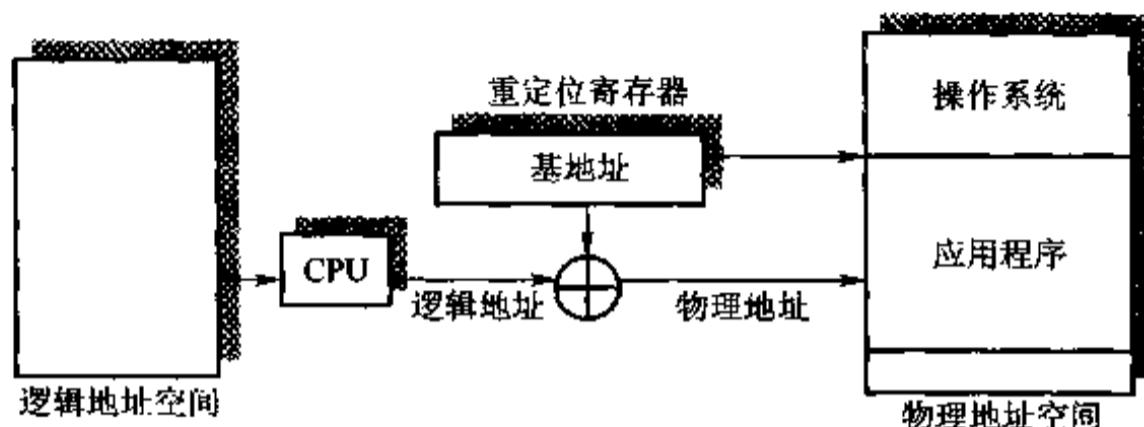


图 4.3 动态地址重定位

为了显式地支持 C 语言模型,处理器中至少要有 3 个重定位寄存器,将文本段、数据段和堆栈段分别作为 3 个可重定位代码模块进行管理。Intel x86 计算机系统有 6 个重定位寄存器,由操作系统负责控制和管理这些寄存器。

非常有趣的一点是,虚拟存储器使得动态加载可执行文件或共享代码变得十分容易。以 Linux 系统为例,装入程序只要为进程分配一个连续的虚拟页面区(从虚地址 0x08048000H 开始),同时将对应页表的页表项标记为“不在主存”,通过进程外页表找到目标文件中的适当位置,装入程序无须真正地从磁盘复制应用程序到主存储器中,当页面首次被引用时,虚存管理将自动地从磁盘把程序或数据调入主存储器。

在多道程序系统中,可用的主存空间常常被许多进程共享,程序员编程时不可能事先知道程序执行时的物理驻留位置,必须允许程序因对换或空闲区收集而被移动,这些现象都需要程序的动态地址重定位,即允许正在执行的程序在不同时刻处于主存储器的不同位置。从系统效率出发,动态地址重定位要借助于硬件地址转换机制来实现,重定位寄存器的内容通常保护在进程控制块中,每当执行进程上下文切换时,当前运行进程的重定位寄存器中的内容与其他相关信息一起被保护起来,新进程的重定位寄存器的内容会被恢复,这样进程就在上次中断的位置恢复运行,所使用的是与上次在此位置的同样的主存基地址。

存储保护涉及防止地址越界和控制正确存取。计算机系统中可能同时存在操作系统和多个应用程序,系统程序和多个应用程序在主存储器中各有自己的存储区域,各道程序只能访问自己的主存区而不能互相干扰,因此,操作系统必须对主存储器中的程序和数据进行保护,以免受到其他程序有意或无意的破坏。无论采用何种地址重定位方式,通常进程运行时所产生的所有主存访问地址都应进行检查,确保进程仅访问自己的主存区,这就是地址越界保护。地址越界保护依赖于硬件设施,常用的有界地址和存储键。如何保证存取的正确性呢? 进程在访问分配给自己的主存区时,要对访问权限进行检查,如允许读、写、执行等,从而确保数据的安全性和完整性,

防止有意或无意的误操作而破坏主存信息,这就是信息存取保护。

连续存储空间管理

4.2.1 固定分区存储管理

固定分区存储管理的基本思想是:主存空间被划分成数目固定不变的分区,各分区的大小不等,每个分区只装入一个作业,若多个分区中都装有作业,则它们可以并发执行,这是支持多道程序设计的最简单的存储管理技术。固定分区(fixed partition)存储管理又称为定长分区或静态分区模式,早期 IBM 操作系统 OS/MFT(Multiprogramming with a Fixed Number of Tasks)就采用这种存储管理技术。

为了说明各分区的分配和使用情况,需要设置一张“主存分配表”,记录主存储器中划分的分区及其使用情况。主存分配表指出各分区的起始地址和长度,“占用标志”用来指示此分区是否被使用,当其值为“0”时,表明此分区尚未被占用。主存分配时总是选择那些“占用标志”为“0”的分区,当某分区被分配给一个长度小于或等于分区长度的作业后,则在“占用标志”中填入占用此分区的作业名。在图 4.4 中,第 2、5 分区分别被作业 Job1 和 Job2 占用,其余分区空闲,当分区中的程序执行结束归还主存区时,相应分区的“占用标志”置“0”,其占用的分区又变成空闲,可被重新分配使用。由于固定分区是预先将主存分割成若干连续区域,分割时各分区在主存分配表中可按地址顺序排列,那么,其主存分配算法就十分简单。

分区号	起始地址	长度	占用标志
1	8 KB	8 KB	0
2	16 KB	16 KB	Job1
3	32 KB	16 KB	0
4	48 KB	16 KB	0
5	64 KB	32 KB	Job2
6	96 KB	32 KB	0

图 4.4 固定分区存储管理的主存分配表

固定分区的一项任务是何时及如何把主存空间划分成分区。这项工作通常由系统管理员和操作系统初始化模块协同完成。系统初次启动时,系统操作员根据当天的作业情况把主存储器划分成大小不等但数目固定的分区。

作业进入分区有两种排队策略:一是每个分区有单独的作业等待队列,调度程序选中作业后,创建用户进程并将其排入一个能够装入它的最小分区的进程等待队列尾部,当此分区空闲

时,就装入队首进程执行。这样做好处是可使装入分区的未用空间最小,但如果等待处理的作业的大小很不均匀,将导致分区有的空闲而有的忙碌;二是所有等待处理的作业排成一个等待队列,每当有分区空闲时,就从队首起依次搜索分区长度能容纳的作业以便装入执行,为了防止小作业占用大分区,也可以搜索分区长度所能容纳的最大作业装入执行。

固定分区能够解决单道程序运行在并发环境下不能与 CPU 速度匹配的问题,同时也解决了单道程序运行时主存空间利用率低的问题。其缺点是:首先,由于预先已规定分区的大小,使得大作业无法装入,用户不得不采用覆盖等技术加以补救,这样不但加重用户的负担,而且极不方便;其次,主存空间利用率不高,作业很少会恰好填满分区。例如,图 4.4 中若 Job1 和 Job2 两个作业实际只需 10 KB 和 18 KB 的主存空间,但它们却占用 16 KB 和 32 KB 的区域,共有 20 KB 的主存区域占而不用被白白浪费,出现分区内的“碎片”;再者,如果一个作业在运行过程中要求动态扩充主存空间,采用固定分区是相当困难的;最后,因为分区的数目是在系统初启时确定的,这就会限制多道运行的程序的个数,特别不适应分时系统交互型用户及主存需求变化很大的情形。然而,固定分区方法实现简单,因此,对于程序大小和出现频率已知的情形,还是比较合适的。

4.2.2 可变分区存储管理

1. 可变分区主存空间的分配和去配

可变分区(variable partition)存储管理又称动态分区模式,按照作业的大小来划分分区,但划分的时间、大小、位置都是动态的。系统把作业装入主存时,根据其所需要的主存容量查看是否有足够的空间,若有,则按需分割一个分区分配给此作业;若无,则令此作业等待主存资源。由于分区的大小是按照作业的实际需求量而定的,且分区的数目也是可变的,所以,可变分区能够克服固定分区中的主存资源的浪费,有利于多道程序设计,提高主存资源的利用率。使用可变分区存储管理的一个例子是 IBM 操作系统 OS/MVT(Multiprogramming with a Variable Number of Tasks)。

在可变分区模式下,在系统初启且用户作业尚未装入主存储器之前,整个用户区是一个大空闲分区,随着作业的装入和撤离,主存空间被分成许多分区,有的分区被占用,而有的分区是空闲的。主存中分区的数目和大小随着作业的执行而不断改变,为了方便主存空间的分配和去配,用于管理的数据结构可由两张表组成:“已分配区表”和“未分配区表”。当装入新作业时,从未分配区表中找出一个足够容纳它的空闲区,将此区分成两部分,一部分用来装入作业,成为已分配区;另一部分仍是空闲区(若有)。这时,应从已分配区表中找出一个空栏目登记新作业的起始地址、占用长度,同时修改未分配区表中空闲区的长度和起始地址。当作业撤离时,已分配区表中的相应状态变为“空”,而将收回的分区登记到未分配区表中,若有相邻空闲区再将其连接后登记。可变分区的回收算法较为复杂,当一个作业 X 撤离时,可分成 4 种情况:其邻近都有作业(A 和 B),其一边有作业(A 或 B),其两边均为空闲区(黑色区域)。可变分区回收情况如图 4.5 所示,同时应修改主存分配表。

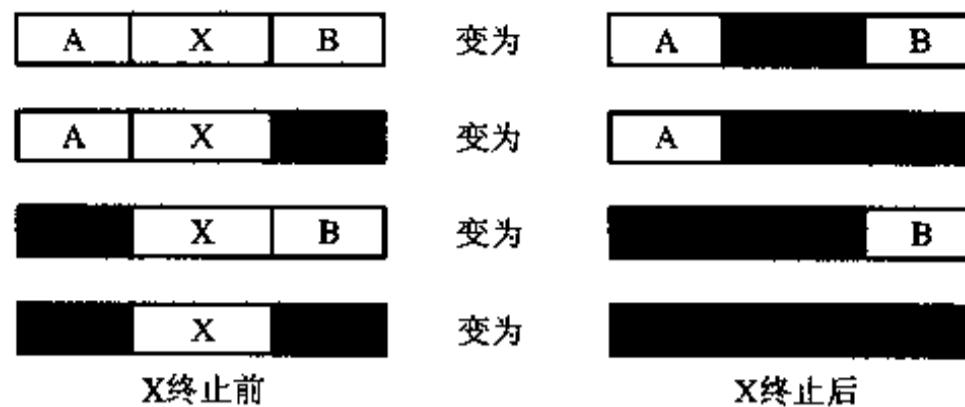


图 4.5 可变分区回收情况

由于分区的数目不定,采用链表是另一种较好的空闲区管理方法,用链指针把所有空闲分区链接起来,每个主存空闲区的开头单元存放本空闲区长度及下一个空闲区起始地址指针,系统设置指向空闲区链的头指针。在使用时,沿链查找并取一个长度能满足要求的空闲区给进程,再修改链表;归还时,把此空闲区链入空闲区链表的相应位置即可。空闲区链表管理比空闲区表格管理要复杂,但其优点是链表自身并不占用存储单元。

无论空闲区表格管理还是空闲区链表管理,表格和链表中的空闲区都可按一定规则排列。例如,按空闲区大小,从大到小或从小到大排列;或按空闲区地址,从大到小或从小到大排列,以方便空闲区的查找和回收。常用的可变分区分配算法有以下 5 种:

(1) 最先适应分配算法

最先适应(first fit)分配算法顺序查找未分配区表或链表,直至找到第一个能满足长度要求的空闲区为止,分割此分区,一部分分配给作业,另一部分仍为空闲区(若有)。采用这一分配算法时,未分配区表或链表中的空闲区通常按地址从小到大排列。这样,为进程分配主存空间时从低地址部分的空闲区开始查找,可使高地址部分尽可能少用,以保持一个大空闲区,有利于大作业的装入;但这样做会使主存储器低地址和高地址两端的分区利用不均衡,也将给回收分区带来麻烦,需要搜索未分配区表或链表来确定它在表格或链表中的位置且要移动相应的登记项。

(2) 下次适应分配算法

下次适应(next fit)分配算法总是从未分配区的上次扫描结束处顺序查找未分配区表或链表,直至找到第一个能满足长度要求的空闲区为止,分割这个未分配区,一部分分配给作业,另一部分仍为空闲区(若有)。这一算法是最先适应分配算法的变种,能够缩短平均查找时间,且存储空间利用率更加均衡,不会导致小空闲区集中于主存储器一端。

(3) 最优适应分配算法

最优适应(best fit)分配算法扫描整个未分配区表或链表,从空闲区中挑选一个能满足用户进程要求的最小分区进行分配。此算法保证不会分割一个更大的区域,使得装入大作业的要求容易得到满足,同时,通常把空闲区按长度递增顺序排列,查找时总是从最小的一个空闲区开始,直至找到满足要求的分区为止,这时,最优适应分配算法等同于最先适应分配算法。此算法的主存利用率好,所找出的分区如果正好满足要求则是最合适的。如果比所要求的分区略大则分割后会使剩下的空闲区很小,难以利用,其查找时间也是最长的。

(4) 最坏适应分配算法

最坏适应(worst fit)分配算法扫描整个未分配区表或链表,总是挑选一个最大的空闲区分割给作业使用,其优点是使剩下的空闲区不致过小,对中小型作业有利。采用此分配算法可把空闲区按长度递减顺序排列,查找时只需看第一个分区能否满足进程要求,这样使最坏适应分配算法的查找效率很高,此时,最坏适应分配算法等同于最先适应分配算法。

(5) 快速适应分配算法

快速适应(quick fit)分配算法为那些经常用到的长度的空闲区设立单独的空闲区链表。例如,有一个 n 项的表,此表第一项是指向长度为 2 KB 的空闲区链表表头的指针,第二项是指向长度为 4 KB 的空闲区链表表头的指针,第三项是指向长度为 8 KB 的空闲区链表表头的指针,依此类推。像 9 KB 这样的空闲区既可放在 8 KB 的链表中也可放在一个特殊的空闲区链表中。此算法查找十分快速,只要按用户进程长度直接搜索能容纳它的最小空闲区链表并取第一块分配,但归还主存空间时与相邻空闲区的合并既复杂又费时。

由于最先适应分配算法简单、快速,在实际操作系统中用得较多,其次是下次适应分配算法和最优适应分配算法。

2. 地址转换与存储保护

对固定分区采用静态地址重定位,进程运行时使用绝对地址,可由加载程序进行地址越界检查。对可变分区则采用动态地址重定位,进程的程序和数据的地址转换由硬件完成,硬件设置两个专用控制寄存器:基址寄存器和限长寄存器,基址寄存器存放分配给进程使用的分区的起始地址,限长寄存器存放进程所占用的连续存储空间的长度。当进程占有 CPU 运行后,操作系统可把分区的起始地址和长度送入基址寄存器和限长寄存器,在执行指令或访问数据时,由硬件根据基址寄存器进行地址转换得到绝对地址。

当逻辑地址小于限长值时,逻辑地址加基址寄存器的值就可获得绝对地址;当逻辑地址大于限长值时,表示进程所访问的地址超出所分得的区域,此时不允许访问,达到存储保护的目的。

在多道程序系统中,硬件只需设置一对基址/限长寄存器,一个进程在执行过程中出现等待事件时,操作系统把基址/限长寄存器的内容随同此进程的其他信息,如 PSW、通用寄存器等一起保存起来,另一个进程被选中执行时,则将其基址/限长值再送入基址/限长寄存器。世界上最早的巨型机 CDC6600 便采用这种方案。

C 语言程序会被编译成至少 3 个段:代码段、数据段、堆栈段,UNIX 进程模型是在这种模块化基础上形成的;相应地,Intel x86 平台提供专用的存放段基址的寄存器,代码段寄存器 CS 在指令执行期间重定位指令地址,堆栈段寄存器 SS 为栈指令的执行重定位地址,数据段寄存器 DS 在指令执行周期内重定位其他地址。在有 N 个重定位寄存器的机器中,允许每个进程获得 N 个不同的主存段,并在运行时进行动态地址重定位。

如果每个进程只能占用一个分区,那么,就不允许各个进程之间有公共区域,这样,当多个进程共享例行程序时就只好在各自的主存区存放一套,从而主存利用率低。提供两对或多对基址/限长寄存器的机器中,允许一个进程占用两个或多个分区。可规定某对基址/限长寄存器的区域

是共享的,用来存放共享的程序和数据,当然,共享区域中的信息只能读出不能写入,于是多个用户进程共享的例行程序就可放在限定的公用区域中,如图 4.6 所示,让进程的共享部分取相同的基址/限长值。

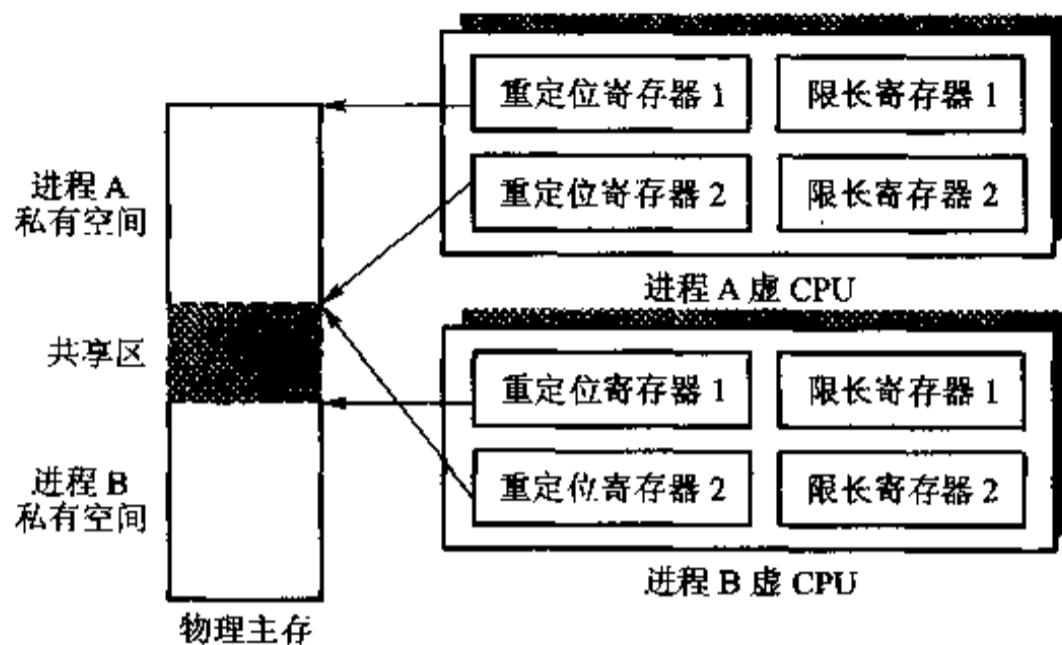


图 4.6 多对重定位寄存器支持主存共享

4.2.3 伙伴系统

1. 伙伴系统原理

伙伴系统(Knuth, 1973 年)又称 buddy 算法,是一种固定分区和可变分区折中的主存管理算法,其基本原理是:任何尺寸为 2^i 的空闲块都可被分为两个尺寸为 2^{i-1} 的空闲块,这两个空闲块称作伙伴,它们可以被合并成尺寸为 2^i 的原先的空闲块。

伙伴系统通常用在以固定尺寸为单位分配的系统中,单位为字长或固定字长序列。假设主存储器包含 2^m 个分配单位,则最大空闲块为 2^m 个单位,小一些的为 2^{m-1} 个单位,依此类推,最小空闲块尺寸为 2^0 ,即一个单位。为了实现这一算法,需要位图和空闲链表作为辅助工具,位图用来跟踪主存块的使用情况,空闲链表用做维护生存中尚未使用的主存块。

伙伴系统维护数组 free 来记录空闲块,它包含 $m+1$ 个列表头,每种空闲块的尺寸有一个,即 free[i] 把所有尺寸为 2^i 的空闲块用双向指针链接在一起。系统处理 n 个存储单位请求的步骤如下:查找满足 $n \leq 2^i$ 的最小空闲块尺寸 ($i = 0, 1, \dots$),若此尺寸空闲块列表非空,则从列表中移走此空闲块,将其分配给请求进程,处理结束;否则,再检查 2^{i+1} 的空闲块尺寸,循环直至找到一个非空列表。假设找到 free[$i+1$],把此列表的第一个空闲块取出,分成大小减半的两个空闲块,分别称为 L 和 R。空闲块 R 被移入列表 free[i] 中,如果 L 的尺寸是请求的可能的最小尺寸,就把空闲块 L 分配给请求进程;如果 L 仍然太大,重复执行等分操作,即把 L 再拆分为大小减半的两个空闲块,其中一个空闲块移入尺寸更小一级的空闲块列表中,再考虑分配另一半。

当已分配尺寸为 2^i 的主存块被释放时,要进行相反的操作,系统检查被释放块的伙伴是否被占用,如果被占用,则新空闲块被移入尺寸为 2^i 的空闲块列表;否则,从 2^i 尺寸的空闲块列表

中移出伙伴,两个空闲块合并成一个尺寸为 2^{i+1} 的空闲块;重复这种操作,直到生成可能的最大空闲块。为了提高操作速度,空闲块合并不是通过搜索位图快速找到被释放块的对应位,并检查伙伴块的对应位是否为0来实现的,回收过程是递归的,一直进行到没有空闲伙伴为止。

如图4.7所示的例子用来说明伙伴算法的申请和归还原理。初始时1 MB的链表仅有一个表项包含1 MB的空闲块,其他链表均为空。当有作业被调入主存储器时,假如所需主存空间如下:A为80 KB、B为50 KB、C为100 KB和D为60 KB,则伙伴算法的申请和归还情况列于图4.7中。伙伴系统分配和合并操作的速度快,其缺点是:当所申请的主存空间大小非2的整数次幂时,内部碎片较大。

128 KB	256 KB	384 KB	512 KB	640 KB	768 KB	896 KB	1 MB
A	128 KB		256 KB		512 KB		
A	B	64 KB	256 KB		512 KB		
A	B	64 KB	C	128 KB	512 KB		
128 KB	B	64 KB	C	128 KB	512 KB		
128 KB	B	D	C	128 KB	512 KB		
128 KB	64 KB	D	C	128 KB	512 KB		
256 KB			C	128 KB	512 KB		
1 024 KB							

图4.7 伙伴算法的申请和归还原理

2. Linux伙伴系统

Linux采用请求分页存储管理,分配时并不需要连续分布的页框,然而在I/O操作及特殊操作时,会绕过分页机制,要求获得连续分布的页框,直接实现磁盘与主存间的数据传送。为此,引进伙伴系统,使用buddy算法及下面的数据结构来实现。

- (1) 以page结构为数组元素的mem_map[]数组。
- (2) 以free_area_struct结构为数组元素的free_area数组:

```
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int *map;
};

static free_area_struct free_area[NR_MEM_LISTS];
```

此数组记录空闲主存页框,共11个元素(NR_MEM_LISTS默认值),维护11个不同空闲页框数的链表,大小从 $2^0=1$ 到 $2^{10}=1\,024$,第*i*个元素代表mem_map数组中第*i*组空闲块链表头。

(3) 位图数组(bitmap)共 11 个, 每个空闲页框块数的链表对应一张, 用二进制数表示主存页框的使用情况, 第 0 组的每一位表示单个页框的使用情况, 为 1 表示此页框正在使用, 为 0 表示空闲; 第 1 组的每一位表示相邻两个页框的使用情况, 如果其中的位置 1, 表示所对应的两个页框正在使用, 依此类推; 第 i 组中的每一位表示相邻 2^i 个页框被使用的情况, 例如, 第 6 组中的某位置 1, 说明对应的 64 个相邻页框正在被使用, 仅当 64 个页框全部回收后, 此位才能清 0。直接向伙伴系统申请空间和释放空间的函数是 alloc_pages() 和 free_pages_ok()。

3. Linux 基于伙伴系统的 slab 分配器

伙伴系统以页框为基本分配单位, 在很多情况下, 内核所需要的主存量远远小于页框大小, 如 inode、vma、task_struct 等。为了更经济地使用内核主存资源, 引入 solaris 操作系统中首创的基于伙伴系统的 slab 分配器, 其基本思想是: 为经常使用的小对象建立缓冲存储, 小对象的申请和释放都通过 slab 分配器来管理, 仅当缓冲存储不够用时才向伙伴系统申请更多的空间。这样做的好处是: 充分利用主存储器, 减少内部碎片, 对象管理局部化, 尽可能少地与伙伴系统打交道, 从而提高效率。

slab 分配器的结构如图 4.8 所示, 主存中建立多个 cache, 每个 cache 有一个 slab 链, 每个 slab 由一个或最多 32 个物理连续的页框组成, 用于存放对象。例如, 一个 slab 中存放 task_struct 对象, 另一个 slab 中存放 inode 对象, 等等。

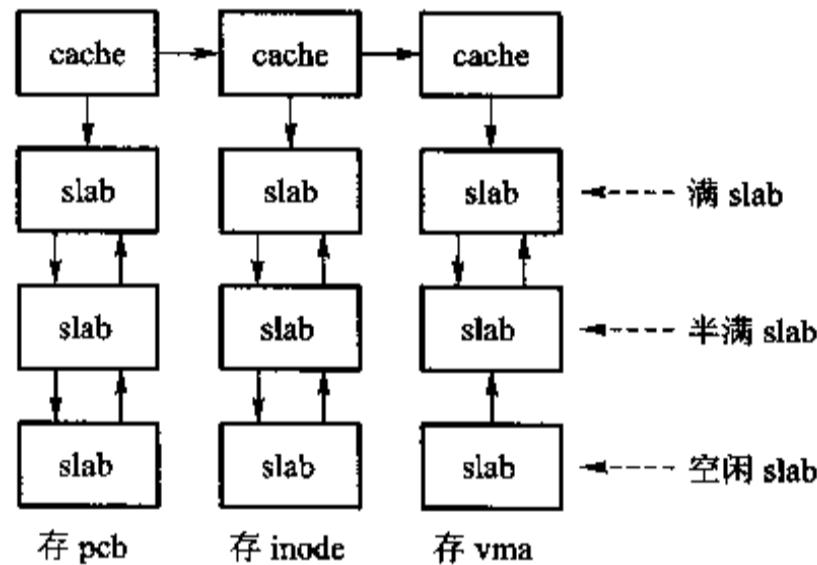


图 4.8 slab 分配器的结构

从图 4.8 中可见, 每个 slab 均处于三种状态之一: 满的(所有对象都已被分配, 排在上面)、半满的(尚有空闲对象, 排在中间)和空闲的(所有对象均空闲, 排在下面)。当内核分配一个对象时, 先从半满的 slab 中查找, 然后从空闲 slab 中查找, 如果没有空闲对象就应创建一个 slab 以供分配, 这种策略能减少主存碎片。下面来看 task_struct 的例子, 内核用一个全局变量存放指向 task_struct slab 的指针: kmem_struct_t * task_struct_cachep; 当内核初始化时, 在 fork_init() 中创建高速缓存, 其中可以存放类型为 task_struct 的对象。每当进程调用 fork() 时, 调用内核函数 do_fork(), 由它使用 kmem_cache_alloc() 函数在对应的 slab 中建立一个 task_struct 对象, 进程运行结束后, task_struct 对象被释放, 返还给 task_struct_cachep slab。

除了这些特定对象的缓冲存储器之外,Linux 系统还提供 13 种通用缓存,其存储对象的大小分别为 32 B、64 B、128 B、256 B、512 B、1 KB、2 KB、4 KB、8 KB、16 KB、32 KB、64 KB 和 128 KB,用来满足特定对象之外的普通主存空间需求,单位大小呈级数增长,保证内部碎片率不超过 50%。

slab 分配器的主要操作如下。

- (1) kmem_cache_create() 函数: 创建特定对象的 slab 结构, 并加入 cache 所管理的队列。
- (2) kmem_cache_alloc() 与 kmem_cache_free() 函数: 分别用于分配和取消一个拥有专用 slab 队列的对象。
- (3) kmem_cache_grow() 与 kmem_cache_reap() 函数: kmem_cache_create() 函数只是建立所需的专用缓冲区队列的基础设施, 所形成的 slab 是一个空队列。具体 slab 的创建则要等到需要分配缓冲区却发现并无空闲的缓冲区可供分配时, 通过 kmem_cache_grow() 来进行, 它向伙伴系统申请空间; kmem_cache_reap() 用于减少 slab。
- (4) kmalloc() 与 kfree() 函数: 分别用来从通用缓冲区队列中申请和释放空间。
- (5) kmem_getpages() 与 kmem_freepages() 函数: slab 与页框级分配器的接口。

4.2.4 主存不足的存储管理技术

1. 移动技术

可变分区法中, 必须把进程装入一个连续的主存区域, 由于进程不断地装入和撤销, 导致主存中常常出现分散的小空闲区, 称之为“碎片”。有时“碎片”会小到竟然连小进程都容纳不下, 这样, 不但浪费主存资源, 还会限制进入主存的进程数目。

当在未分配区表中找不到足够大的空闲区来装入新进程时, 可采用移动技术把已在主存中的进程分区连接到一起, 使分散的空闲区汇集成片, 这就是移动技术, 也叫做主存紧凑。第一种方法是把所有当前占用的分区移动到主存的一端; 第二种方法是把占用分区移动到主存的一端, 但当产生足够大小的空闲区时就停止移动。

移动操作需要把主存中的进程“搬家”, 即读出每个字并写回主存, 凡涉及地址的信息均应修改, 如基址寄存器、地址指针等, 移动分配的示例如图 4.9 所示。移动虽然可以汇集主存空闲区, 但其开销很大, 现代操作系统都不再采用。“搬家”不是任何时候都能进行的, 由于块设备在与主存储器交换信息时, 通道或 DMA 总是按确定的主存绝对地址完成信息传输, 所以, 当一道程序正在与设备交换数据时往往不能移动, 系统应设法减少移动, 比如, 在装入时总是先挑选不经移动即可装入的进程, 在不得不移动时应力求所移动的道数最少。那么, 何时进行移动呢? 一是进程撤销之后释放分区时, 如果它不与空闲区邻接, 立即实施移动, 于是, 系统始终保持只有一个空闲区; 二是进程装入分区时, 若空闲区的总和够用, 但没有一个空闲区能容纳此进程时, 实施移动。

假设进程 A 请求分配 x KB 主存区, 采用移动技术分配主存空间的算法如下。

步骤 1: 查主存分配表, 若有大于 x KB 的空闲区, 则转步骤 4;

步骤 2: 若空闲区总和小于 x KB, 则令进程 A 等待主存资源;

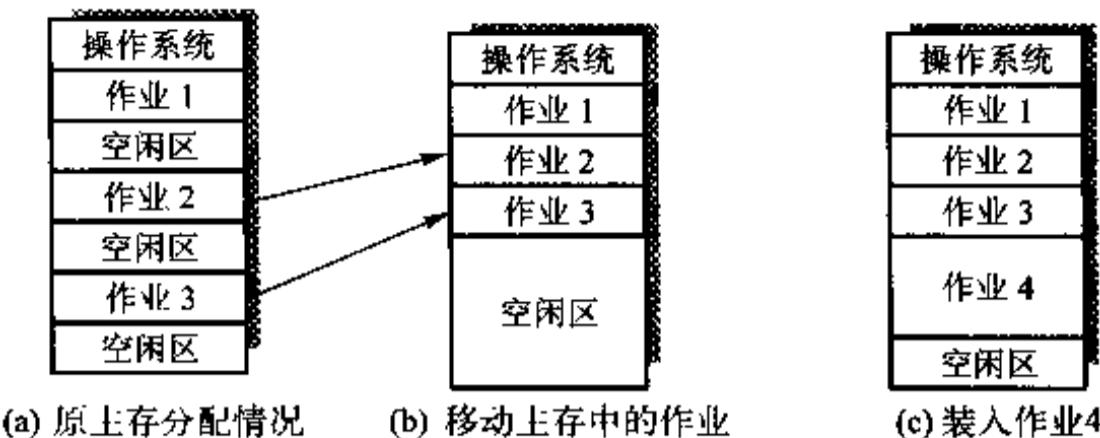


图 4.9 移动分配示例

步骤 3: 移动主存的相关分区信息; 修改主存分配表的有关项; 修改被移动者的基址寄存器等信息;

步骤 4: 分配 x KB 主存; 修改主存分配表的有关项; 设置进程 A 的基址寄存器; 有申请者等待时即予以释放, 算法结束。

移动操作也为进程运行过程中动态扩充主存空间提供了方便。当进程在执行过程中要求增加主存分配区时, 只需适当移动邻近的占用分区就可增加其所占有的连续区的长度, 移动后的基址值和经扩大的限长值都应相应修改。

2. 对换技术

对换技术(swapping)广泛应用于分时系统的调度中, 以解决主存容量不足的问题, 使分时用户获得快速响应时间; 也可用于批处理系统, 以平衡系统负载。如果当前一个或多个驻留进程都处于阻塞态, 此时选择其中的一个进程, 将其暂时移出主存, 腾出空间给其他进程使用, 同时把磁盘中的某个进程换入主存, 让其投入运行, 这种互换称为对换。例如, 当一个进程执行某系统调用时变成阻塞态, 这时存储管理程序会收到进程管理的通知, 以决定是否将此进程的主存映像对换到磁盘; 反之, 当进程管理程序把已对换出去的进程转换为就绪态时, 会通知存储管理程序, 一旦主存可用, 立即把此进程对换回主存。由于有硬件地址重定位寄存器的支持, 对换进来的进程映像被复制到新分配的主存区域并重置重定位寄存器的值。

为了有效地实施对换, 首先, 要解决选择哪个进程换出。如果选择不当, 将造成系统效率欠佳。通常系统把时间片耗尽或将优先级较低的进程换出, 因为短时间内它们不会投入运行。其次, 决定把进程的哪些信息移出去。开始时, 进程从可执行文件被装入主存, 其未修改部分(如代码)在主存与磁盘中始终保持一致, 这些信息不必保存, 当进程换回主存时, 只需简单地从最初的可执行文件再加载一次。数据区和堆栈是进程运行时所创建和修改的, 操作系统可通过文件系统把这些可变信息作为特殊文件保存。有些系统从降低开销的角度考虑, 开辟一块特殊的磁盘区域作为对换空间, 它包含连续的柱面和磁道, 可通过低层磁盘读写实现高效访问。最后, 需要确定对换时机, 在批处理系统中, 当进程要求动态扩充主存空间且得不到满足时可触发对换; 在分时系统中, 对换可与调度结合, 每个时间片结束或执行 I/O 操作时实施, 调度程序启动一个挪出的进程换入, 这样, 轮到它执行时立即可以启动, 对换进主存的进程其主存位置未必还在换出

之前的位置上,所以,需要解决对换过程中进程的地址重定位问题。

假设一个被对换的进程映像占用 k 个磁盘块,那么,一次进程对换的所有开销是 $2k$ 个磁盘块输入/输出的时间,再加上进程重新请求主存资源所造成的时间延迟。对换比移动技术更有效,移动不能保证得到一个满足请求的空闲区,而利用对换技术总可按需挪出若干驻留的阻塞进程,且对换仅涉及少量进程,只需更少的主存访问。与移动不同的是,对换要访问磁盘,这是一个 I/O 集中型操作,会影响对用户的响应时间,但系统可让对换与计算型进程并行工作,不会造成系统性能的显著下降。

UNIX 早期版本通过称作对换器的专门进程实施对换,每当创建新进程、进程动态扩充主存空间时便挪出一个或多个驻留进程,每隔 4 s,对换器进行检查以保证把挪出已久的进程换进。换出的候选者当首选被阻塞的进程,否则就挑选就绪进程。需要考虑进程属性,如已消耗 CPU 时间、在主存已逗留的时间等。Windows 对换空闲线程负责把进程从主存挪出,此线程每隔 4 s 被唤醒,查找已空闲一定时间(秒级)的线程,将其核心栈换出,当一个进程的所有线程都被换出时,余下部分(包括线程共享的代码和数据)也被换出,那么,这意味着整个进程被对换出去。

3. 覆盖技术

移动和对换技术解决因其他程序存在而导致主存区不足的问题,这种主存短缺只是暂时的;如果程序的长度超出物理主存总和,或超出固定分区的大小,则出现主存永久性短缺,大程序无法运行,前述两种方法无能为力,解决方法之一是采用覆盖(overlaying)技术。覆盖是指程序执行过程中程序的不同模块在主存中相互替代,以达到小主存执行大程序的目的,基本的实现技术是:把用户空间分成固定区和一个或多个覆盖区,把控制或不可覆盖部分放在固定区,其余按调用结构及先后关系分段并存放在磁盘上,运行时依次调入覆盖区。系统必须提供覆盖控制程序及相应的系统调用,当进程装入运行时,由系统根据用户给出的覆盖结构进行覆盖处理,程序员必须指明同时驻留在主存的是哪些程序段,哪些是被覆盖的程序段,这种声明可从程序调用结构中获得。覆盖技术的不足是把主存管理工作转给程序员,他们必须根据可用物理主存空间来设计和编写程序。此外,同时运行的代码量超出主存容量时仍不能运行,所以现代操作系统极少采用覆盖技术。

4.3 分页存储管理

4.3.1 分页存储管理的基本原理

用分区方式管理存储器,每道程序要求占用主存的一个或多个连续存储区域,导致主存中产生“碎片”。有时为了接纳新作业,往往需要移动已在主存的信息,这样不仅不方便,而且处理器的开销太大。采用分页存储管理允许程序存放到若干不相邻的空闲块中,既可免除移动信息工作,又可充分利用主存空间,消除动态分区法中的“碎片”问题,从而提高主存空间的利用率。分页存储管理涉及的基本概念如下。

1. 页面

进程逻辑地址空间分成大小相等的区,每个区称为页面或页,页号从 0 开始依次编号。

2. 页框

页框又称页帧,把主存物理地址空间分成大小相等的区,其大小与页面大小相等,每个区是一个物理块或页框,块号从 0 开始依次编号。

3. 逻辑地址

与此对应,分页存储器的逻辑地址由两部分组成:页号和页内位移,格式如下:

页 号	页内位移
-----	------

前面部分表示地址所在页面的编号,后面部分表示页内位移。计算机地址总线通常是 32 位,页面尺寸若规定为 12 位(页长 4 KB),那么,页号共 20 位,表示地址空间最多包含 2^{20} 个页面。

采用分页存储管理时,逻辑地址是连续的,用户在编制程序时仍使用相对地址,不必考虑如何分页,由硬件地址转换机构和操作系统的管理需要来决定页面尺寸,从而确定主存分块大小。进程在主存中的每个页框内的地址是连续的,但页框之间的地址可以不连续,进程主存地址由连续到离散的变化为虚拟存储器的实现奠定了基础。

4. 页表和地址转换

在进行存储分配时,以页框为单位,进程的信息有多少页,那么,把它装入主存时就分配多少块,虽然进程的逻辑地址分成页面后是连续的,但被装入后的相应物理块(页框)未必紧邻,即进程的信息按页面分散存放在主存不相邻的页框中,那么,当进程的程序和数据被分散存放在主存中后,其页面与被分配的页框如何建立联系呢?逻辑地址(页面)如何转换成物理地址(页框)呢?进程被装入后的物理地址空间由连续变成分散后,如何保证程序正确执行呢?仍然采用动态地址重定位技术,让程序在执行时动态地进行地址变换,由于程序以页面为单位存储,所以为每个页面设立一个重定位寄存器,这些重定位寄存器的集合称为页表(page table)。页表是操作系统为进程建立的,是程序页面和主存对应页框的对照表,页表中的每一栏指明程序中的一个页面和分得页框之间的对应关系。使用页表的目的是把页面映射为页框,从数学的角度而言,页表是一个函数,其变量是页面号,函数值为页框号,通过页表可以把逻辑地址中的逻辑页面域替换成物理页框域。为了减少系统开销,不用硬件而是在主存中开辟存储区以存放进程页表,系统另设置专用的硬件——页表基址寄存器,存放当前运行进程的页表起始地址,以加快地址转换速度。系统应为主存中的进程进行存储分配,并建立页表,指出逻辑地址页号与主存页框号之间的对应关系,页表的长度随进程大小而定。

进程运行前由系统把它的页表基址送入页表基址寄存器,运行时借助于硬件的地址转换机构,按页面动态地址重定位,当 CPU 获得逻辑地址后,由硬件自动按设定的页面尺寸分成两部分:页号 p 和页内位移 d ,先从页表基址寄存器找到页表基址,再用页号 p 作为索引查页表,得到对应的页框号,根据关系式:

$$\text{物理地址} = \text{页框号} \times \text{块长} + \text{页内位移}$$

计算出欲访问的主存单元。因此,虽然进程存放在若干不连续的页框中,但在执行过程中总能按正确的物理地址进行存取。

如图 4.10 所示是分页存储管理的地址转换,在实际进行地址转换时,只要把逻辑地址中的页内位移 d 作为绝对地址中的低地址,根据页号 p 从页表中查得页框号 b 作为绝对地址中的高地址,就组成访问主存储器的绝对地址。

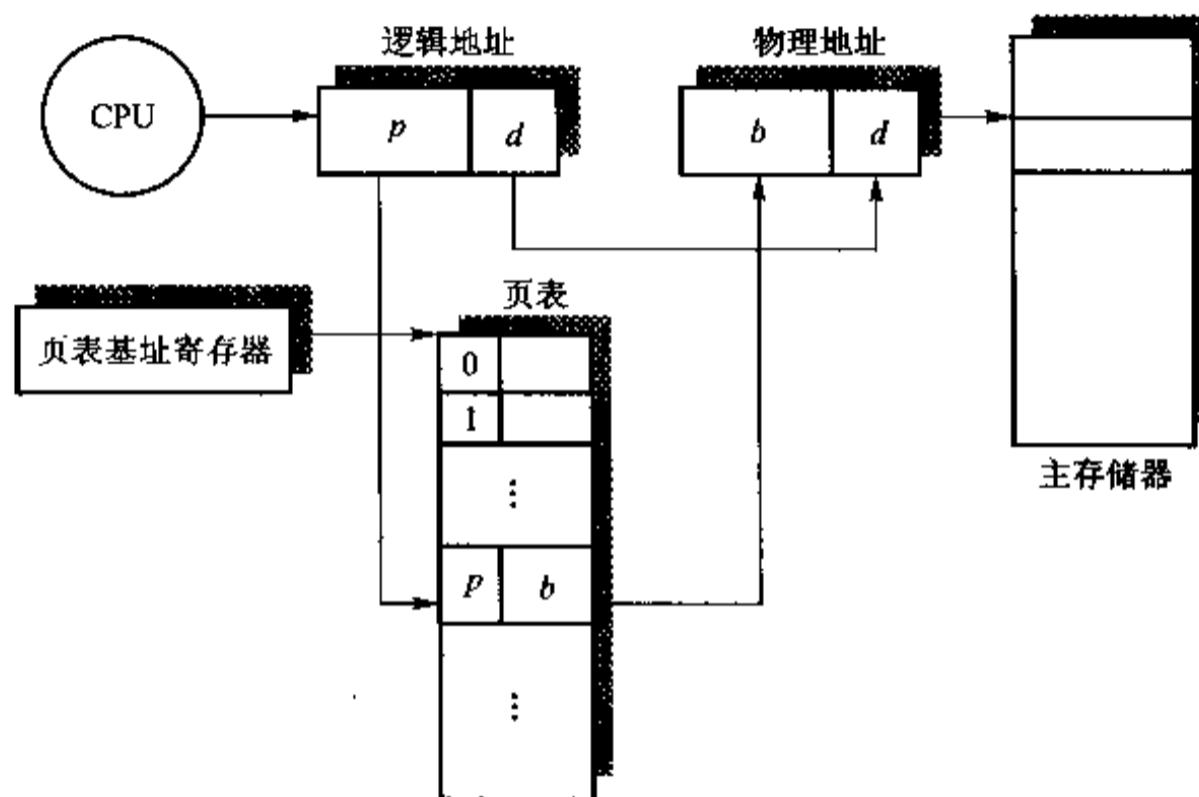


图 4.10 分页存储管理的地址转换

整个系统只有一个页表基址寄存器,只有占用 CPU 的进程才占有它。在多道程序中,当某道程序让出处理器时,应同时让出此寄存器供其他进程使用。

4.3.2 快表

页表可存放在一组寄存器中,地址转换时只要从相应寄存器中取值就可得到页框号,这样做虽然能加快地址转换,但硬件代价太高;页表也可存放在主存中,这样做可以降低系统开销,但是按照给定逻辑地址进行读写操作时,至少访问主存两次:一次访问页表,另一次根据物理地址访问指令或存取数据,这将降低运算速度,比通常执行指令时的速度慢一半。

为了提高运算速度,在硬件中设置相联存储器,用来存放进程最近访问的部分页表项,也称转换后援缓冲(Translation Lookaside Buffer, TLB)或快表,它是分页存储管理的重要组成部分。快表的存取时间远小于主存,速度快但造价高,故容量较小,只能存放几十个页表项。快表项包含页号及对应的页框号,当把页号交给快表后,它通过并行匹配同时对所有快表项进行比较,如果找到,则立即输出页框号,并形成物理地址;如果找不到,再查主存中的页表以形成物理地址,同时将页号及页框号登记到快表中;当快表已满且要登记新页时,系统需要淘汰旧的快表项,最

简单的策略是“先进先出”，总是淘汰最先登记的页。

通过快表实现主存访问的比率称为命中率，命中率越高，性能越好，接近 100% 的命中率表明绝大部分的主存访问都通过快表实现，几乎不用页表；反之，当进程访问遍布主存页面的跳跃性地址时，命中率近乎为 0，这意味着每次访问主存都要使用页表。采用快表后，地址转换时间大大下降，假定访问主存的时间为 100 ns，访问快表的时间为 20 ns，快表为 32 个单元时的查找命中率若为 90%，主存数据要先存入快表，然后再由处理器存取。于是，按逻辑地址进行存取的平均时间为：

$$(100 + 20) \times 90\% + (100 + 100 + 20) \times (1 - 90\%) = 130 \text{ ns}$$

比两次访问主存的时间 200 ns 缩短 35%。

需要注意的是，快表和高速缓存是不同的，两者在一个系统中可同时使用，前者记录最近转换的页号及页框号，而后者保存最近实际访问的指令或数据的副本。

4.3.3 分页存储空间的分配和去配

分页存储管理中，系统要建立一张主存物理块表，用来记录页框的状态，管理主存物理块的分配，所包含的信息有主存总块数、哪些为空闲块、哪些块已分配及分给哪个进程等，最简单的方法可用位示图来记录分配情况，每位与一个页框相对应，用 0/1 表示对应块为空闲/已占用，用另一专门字记录当前空闲块数，如图 4.11 所示。

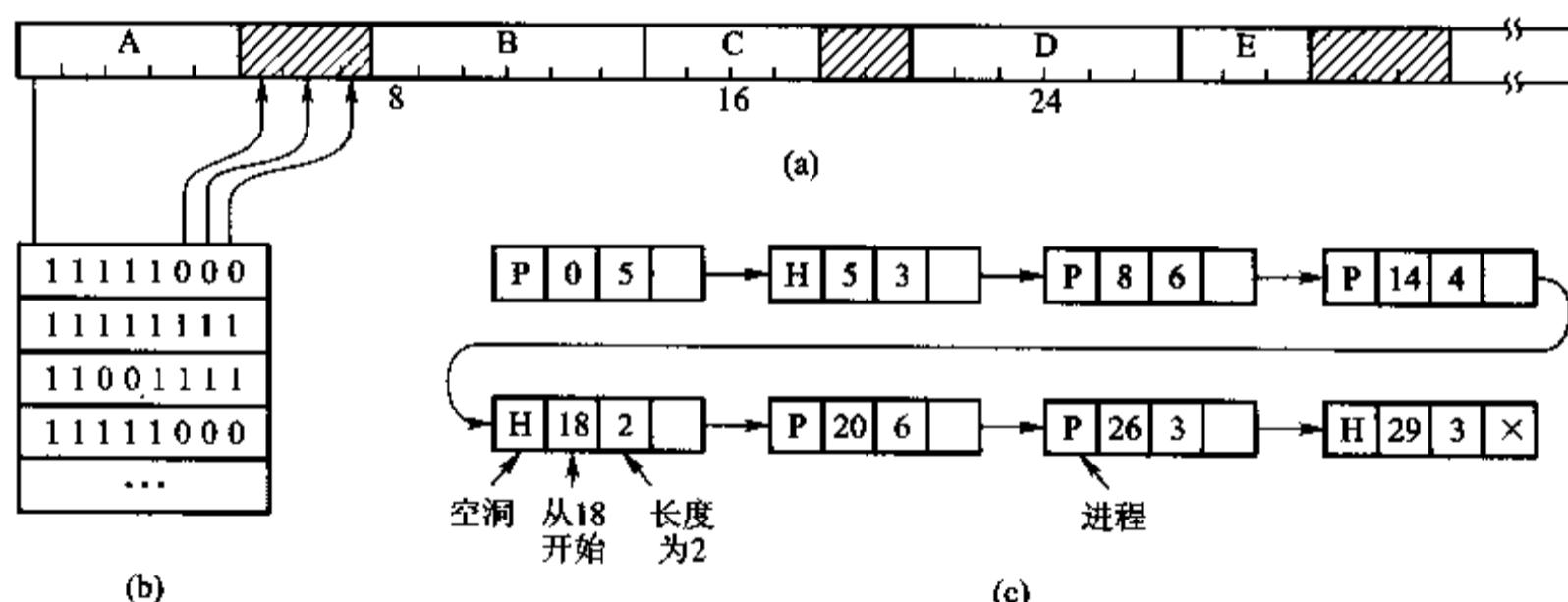


图 4.11 主存分配的位示图和链表方法

分页存储管理页框分配算法如下：进行主存分配时，先查空闲块数能否满足用户进程的要求，若不能，令进程等待；若能，则查位示图，找出为“0”的那些位，置占用标志，从空闲块数中减去本次占用块数，按所找到的位的位置计算对应页框号，填入此进程的页表。进程执行结束归还主存时，根据归还的页框号，计算出对应位在位示图中的位置，将占有标志清“0”，并将归还块数加入空闲块数中。

图 4.11(c)是主存分配的链表方法，表中各项都包含以下内容：是进程占用区(P)还是空闲

区(H)、起始地址、长度和指向下一表项的指针。在本例中，链表是按照地址从小到大排序的，其优点是链表的更新和修改比较方便，运行结束的进程通常有两个邻居，既可能是进程也可能是空闲区，只需修改邻近的链表项。

4.3.4 分页存储空间的页面共享和保护

1. 页面共享和保护

在多道程序系统中，编译程序、编辑程序、解释程序、公共子程序、公用数据等都是可共享的，这些共享信息在主存中保留副本，分页存储管理能实现多个进程共享程序和数据，共享页面信息可大大提高主存空间的利用率。

实现页面共享，必须区分数据共享和程序共享。实现数据共享时，允许不同进程对共享的数据页用不同的页号，只要让各自页表中的有关表项指向共享的数据页框；实现程序共享时，由于指令包含指向其他指令或数据的地址，进程依赖于这些地址才能执行，所以，不同进程正确执行共享代码页面，必须为它们在所有逻辑地址空间中指定同样的页号。假定有一个被共享的编辑程序 EDIT，此程序一定是可再入的，假设其中含有指向其自身的转移指令 branch $n, 64$ ，其中， n 是共享代码的页号，64 是页内位移。问题是 n 要依赖于执行 EDIT 的进程，当进程 P_1 执行时，目标地址必须为 $(n_1, 64)$ ，当进程 P_2 执行时，目标地址必须为 $(n_2, 64)$ 。因为一个存储器单元在任何时刻只能有一个值，所以，页号 n_1 和 n_2 必须相同，即共享页面在进程 P_1 和 P_2 的页表中必须被分配同样的记录，因此，对共享的程序必须规定统一的页号，这样在需要时才能转换成相同的物理地址。

实现信息共享必须解决共享信息的保护问题。通常的做法是在页表中增加标志位，指出此页的信息只读/读写/只可执行/不可访问等，进程访问此页时核对访问模式。例如，欲向只读块写入信息则指令停止执行，产生违例异常信号。另外，也可采取存储保护键作为保护机制，本书第七章将介绍 IBM System/370 系列操作系统的存储保护键保护机制。

2. 动态链接

当进程需要使用各种标准库函数时，需要采用静态方式全部链接到应用程序中，每个可执行代码中都有库函数的副本，这样就增加了对主存容量的要求。如果应用程序仅使用其中一小部分，采用静态链接不但麻烦，而且开销大，影响系统的效率。为此，可把函数定位和链接推迟到运行时刻，只在实际调用发生时才进行。

动态链接需使用共享库(shared library)，它包含共享函数的目标代码模块，在运行时可加载到任意的主存区域，并在主存中和一个程序链接起来，这个过程称为动态链接(dynamic linking)，这是通过动态链接器(dynamic linker)来执行的。在 UNIX/Linux 系统中，共享库的共享代码通常用后缀 .so 来表示，共享库在任何给定的文件系统中，对于一个库只有一个 .so 文件，所有引用此库的可执行目标代码共享此 .so 文件中的代码和数据，而不是像静态链接那样被复制和嵌入引用它们的可执行应用程序中。

假如应用程序 main1.c 需要使用库函数，头文件中包含函数原型 stdio.h 等定义，下面列出

编译和动态链接共享库的过程。

- (1) 编译时,给出 main1.c,并包含 #include <stdio.h> 等头文件。
- (2) 链接器对编译输出信息 main1.o 和标准共享库 libc.so 的重定位和符号表信息进行静态链接。获得部分链接的可执行目标代码命名为 Exmain1。
- (3) 当装入器(execl())加载和运行 Exmain1 时,发现包含动态链接器的路径名,动态链接器本身是一个共享目标代码(如 Linux 系统上的 LD-LINUX.so),装入器不像通常那样将控制传递给应用程序,取而代之的是加载和运行这个动态链接器。
- (4) 动态链接器通过执行下面的重定位完成链接任务。
 - ① 重定位 libc.so 的文本和数据到某个主存段。在 Linux 系统中,标准共享库被加载到从地址 0x40000000H 开始的区域中。
 - ② 重定位 Exmain1 中所有对由 libc.so 定义的符号的引用。
 - ③ 动态链接器将控制传递给应用程序,从此开始,共享库的位置便固定,并在程序执行过程中不会改变。

Windows 中大量利用共享库,称为动态链接库(Dynamic Link Library,DLL),实现操作系统的所有 API。共有数百个 DLL 文件,均包含上百个函数,这些 DLL 文件共同提供系统的全部功能,如底层 I/O 操作、文件操作,还有用户层的窗口及菜单系统等。

Windows 中的动态链接思想如图 4.12 所示,需要引入库(import library)来协助动态链接器使用 DLL,引入库中包含 DLL 文件中每个函数的定位信息,编译时遇到一个 DLL 函数调用时,将在 obj 文件中嵌入函数定位信息,供动态链接器在链接时分析和处理。链接过程中检查发现有关函数调用信息时,自动生成 lib 文件供应用程序动态链接使用。

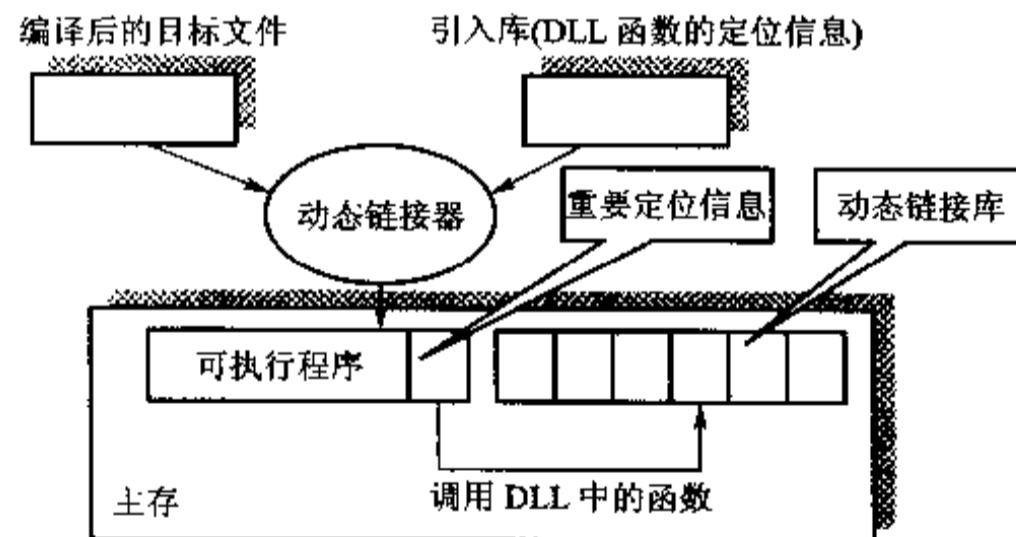


图 4.12 Windows 动态链接

- (1) LoadLibrary 用于加载 DLL 文件到进程地址空间;FreeLibrary 用于从进程地址空间中解除 DLL 文件的映射。
 - (2) GetProcAddress 用于获得函数地址,以便调用函数。
- 无论有多少进程引用 DLL 文件,只有一个副本被装入主存,调用 LoadLibrary 时,此进程的

DLL 引用数加 1, 而调用 FreeLibrary 时, 此进程的 DLL 引用数减 1。如果 DLL 引用数为 0, 则 DLL 文件将从进程地址空间中解除映射。DLL 为变量所分配的主存在调用进程的地址空间中, 若两个进程调用同一个 DLL, 则在每个进程的地址空间中都要分配一次, 这样就有各自的 DLL 数据副本。

4.3.5 多级页表

现代计算机普遍支持 $2^{32} \sim 2^{64}$ B 容量的逻辑地址空间, 采用分页存储管理时, 页表相当大。以 Windows 为例, 其运行的 Intel x86 平台具有 32 位地址, 规定页面 4 KB(2^{12}), 那么, 4 GB(2^{32}) 的逻辑地址空间由 $1 M(2^{20})$ 个页组成, 若每个页表项占用 4 B, 则需要占用 $4 MB(2^{22})$ 连续主存空间来存放页表, 这还是一个进程的地址空间。对于地址空间为 64 位的系统而言, 问题将变得更加复杂。为此, 页表和页面一样也要进行分页, 这就形成多级页表的概念, 具体做法是: 把整个页表分割成许多小页表, 每个小页表称为页表页, 其大小与页框长度相同, 于是, 每个页表页含有若干页表项。页表页从 0 开始顺序编号, 允许被分散存放在不连续的页框中, 为了找到页表页, 应建立地址索引, 称为页目录表, 其表项指出页表页的起始地址。系统为每个进程建立一张页目录表, 其表项指出一个页表页, 而页表页的每个表项给出页面和页框之间的对应关系。页目录表是一级页表, 页表页是二级页表, 共同构成二级页表机制。于是, 逻辑地址结构由 3 个部分组成: 页目录位移、页表页位移和页内位移。

如图 4.13 所示是二级页表实现逻辑地址到物理地址转换的过程, 具体步骤如下: 由硬件页目录表基址寄存器指出当前运行进程的页目录表的主存起始地址, 加上“页目录位移”作为索引, 可找到页表页在主存的起始地址, 再以“页表页位移”作为索引, 找到页表页的表项, 此表项中包含一个面对应的页框号, 由页框号和“页内位移”便可生成物理地址。

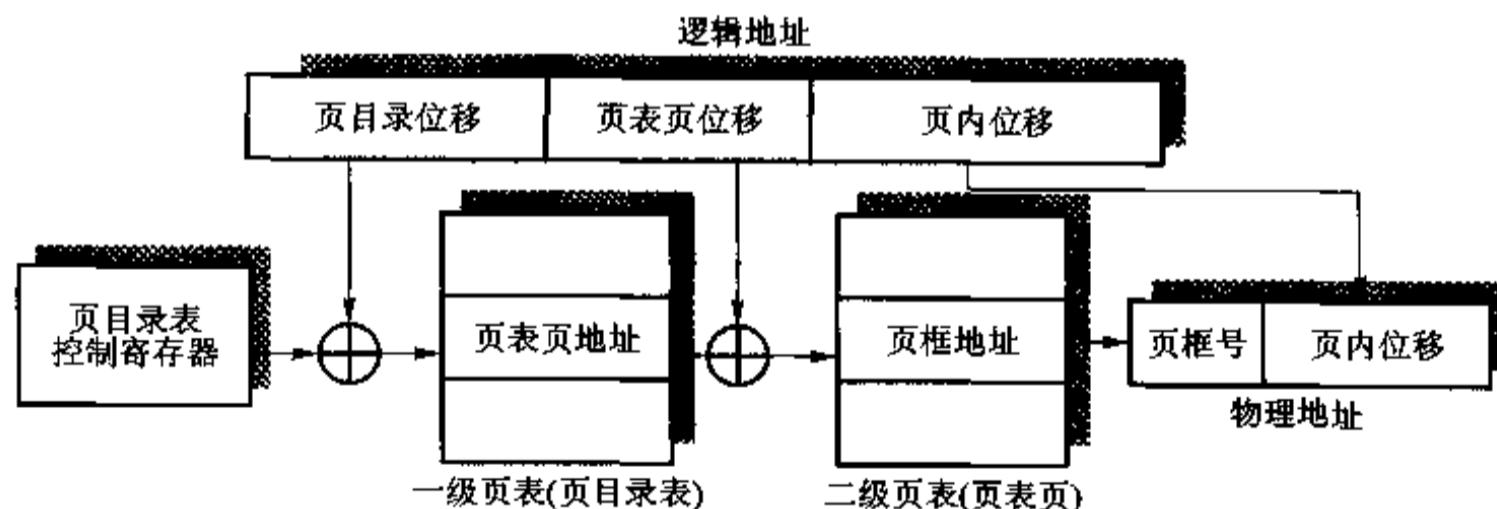


图 4.13 二级页表地址转换过程

上述方法能解决分散存放页表页的问题, 并未解决页表页如何占用主存空间的问题, 解决方法如下: 进程运行涉及页面的页表页应存放在主存, 而其他页表页使用时动态调入, 为此, 需要在页目录表中增加标志位, 指示对应的页表页是否已调入主存, 地址转换机制根据逻辑地址中的“页目录位移”来查页目录表对应表项的标志位, 如未调入, 应产生“缺页表页”中断信号, 请求操

作系统将页表页调入主存。

二级页表地址转换需 3 次访问主存,一次访问页目录、一次访问页表页、一次访问指令或数据,随着 64 位地址的出现,二级页表仍不够用,所以,三级、四级页表也已被引入系统。

SUN 微系统公司计算机使用 SPARC 芯片,采用如图 4.14 所示的三级分页结构。为了避免进程切换时重新装入页目录表指针,硬件支持多达 4 096 个上下文,每个进程一个上下文,当新进程装入主存时,操作系统分配一个上下文号,进程保持这个上下文号直到终止。当 CPU 访问主存时,上下文号和逻辑地址一并送入地址转换机构,使用上下文号作为上下文表的索引,找到顶级页目录,然后,使用逻辑地址中的索引值查找下一级页目录项;直至找到访问页面,形成物理地址。在分页系统中,为了加快地址转换的过程,都会使用快表。

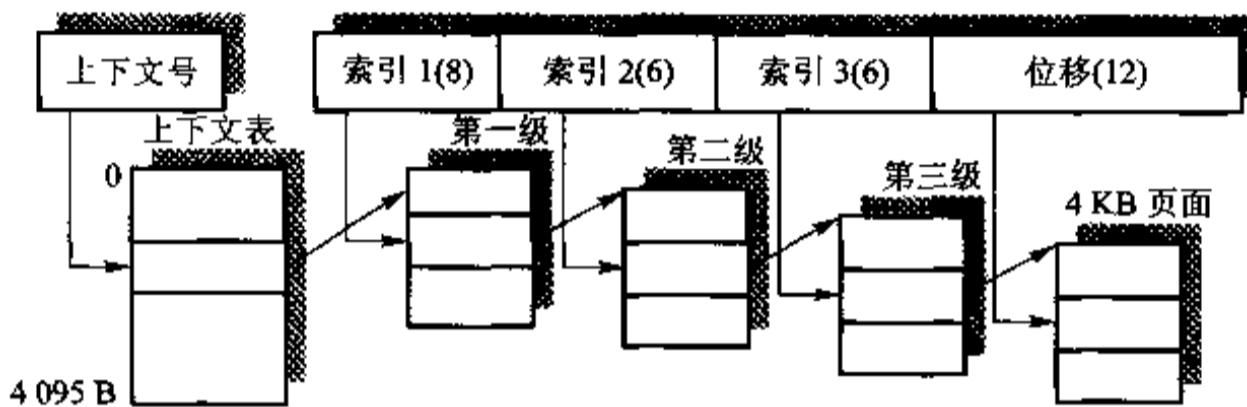


图 4.14 SPARC 三级分页结构

多级页表结构的本质是多级不连续,导致多级索引。以二级页表结构为例,应用程序的页面不连续存放,需要有页面地址索引,此索引就是进程页表。由于进程页表是不连续存放的多个页表页,故这些页表页也需要页表页地址索引,此索引就是页目录,这就形成二级索引,页目录项是页表页的索引,而页表页项是程序页面的索引。Motorola 68030 系列计算机为操作系统设计者提供选择,页表级数和每级的位数可以编程控制。

4.3.6 反置页表

计算机逻辑地址空间越来越大,页表所占用的主存空间也越来越多,页表尺寸与虚地址空间呈正比增长。为了减少主存空间开销,不得不使用多级页表,但也有些操作系统,如 IBM AS/400、PowerPC、UltraSPARC 和 IA-64 体系结构中均采用反置页表(Inverted Page Table, IPT),它的主要优点是只需为所有进程维护一张表。此表为主存中的每个物理块建立一个 IPT 表项并按照块号进行排序,其表项包含:在此页框中页面的页号、页面所属进程的标识符和哈希链指针,用来完成逻辑地址到物理地址的转换,与此相适应,逻辑地址由进程标识符、页号和页内位移 3 个部分组成。

如图 4.15 所示是反置页表及其地址转换的过程:需要访问主存地址时,地址转换机制用进程标识符与页号作为输入,由哈希函数先映射到哈希表,哈希表项存放的是指向 IPT 表项的指针,此指针要么就是指向匹配的 IPT 表项,否则,遍历哈希链直至找到进程标识符与页号均匹配

的 IPT 表项，而此表项的序号就是页框号，通过拼接页内位移便可生成物理地址。若在反置页表中未能找到匹配的 IPT 页表项，说明此页不在主存，触发缺页中断，请求操作系统通过页表调入。

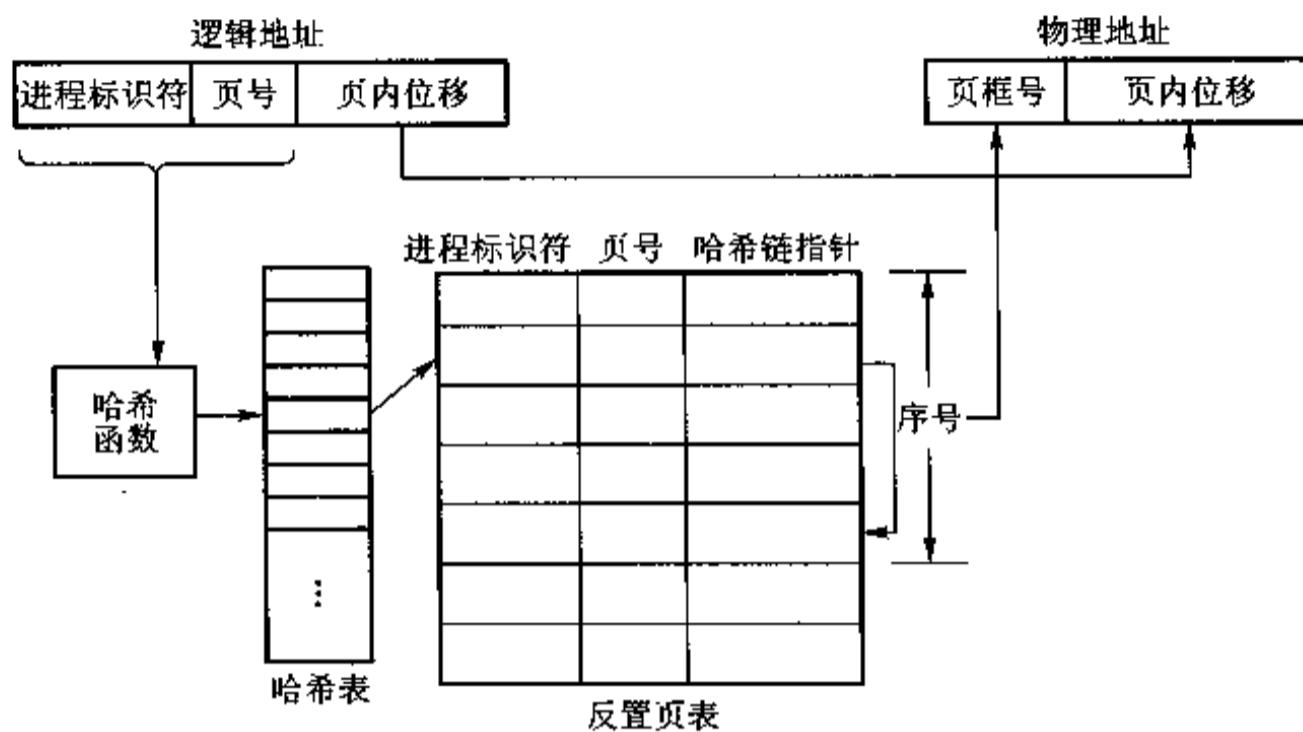


图 4.15 反置页表及其地址转换

IPT 表项中增加哈希链指针，是由于有多个页号经哈希函数转换后可能获得相同的哈希值，所以，利用哈希链来处理这种冲突，在进行地址映射时，用哈希表定位后还可能要遍历哈希链逐一找出所需的页面。

为了使进程能共享主存中的同一页面，必须扩展 IPT 表的内容，使得每个表项可以记录多个进程。这样做虽然能解决问题，但却增加了复杂性。IPT 能减少页表对主存的占用，然而 IPT 仅包含调入主存的页面，不包含未调入的页面，仍需要为进程建立传统页表，不过此页表不再放在主存中，而是存放在磁盘上。当发生缺页中断时，把所需页面调入主存要多访问一次磁盘，速度会比较慢。

4.4 分段存储管理

4.4.1 程序的分段结构

促使存储管理方式从固定分区到动态分区，从分区方式向分页方式发展的主要原因是要提高主存空间利用率。那么，分段存储管理的引入主要是满足用户（程序员）编程和使用上的要求，其他存储管理技术难以满足这些要求。在分页存储管理中，经链接编辑处理得到一维地址结构的可装配目标模块，这是从 0 开始编址的单一连续逻辑地址空间，虽然可以把程序划分成页面，但页面与源程序并不存在逻辑关系，也就难以对源程序以模块为单位进行分配、共享和保护。事实上，程序更多是采用分段结构，高级语言往往采用模块化程序设计方法。如图 4.16 所示，应用

程序由若干程序段(模块)组成,例如,由主程序段、子程序段、数据段和工作区段组成,每段都从0开始编址,有各自的名字和长度,且实现不同的功能。

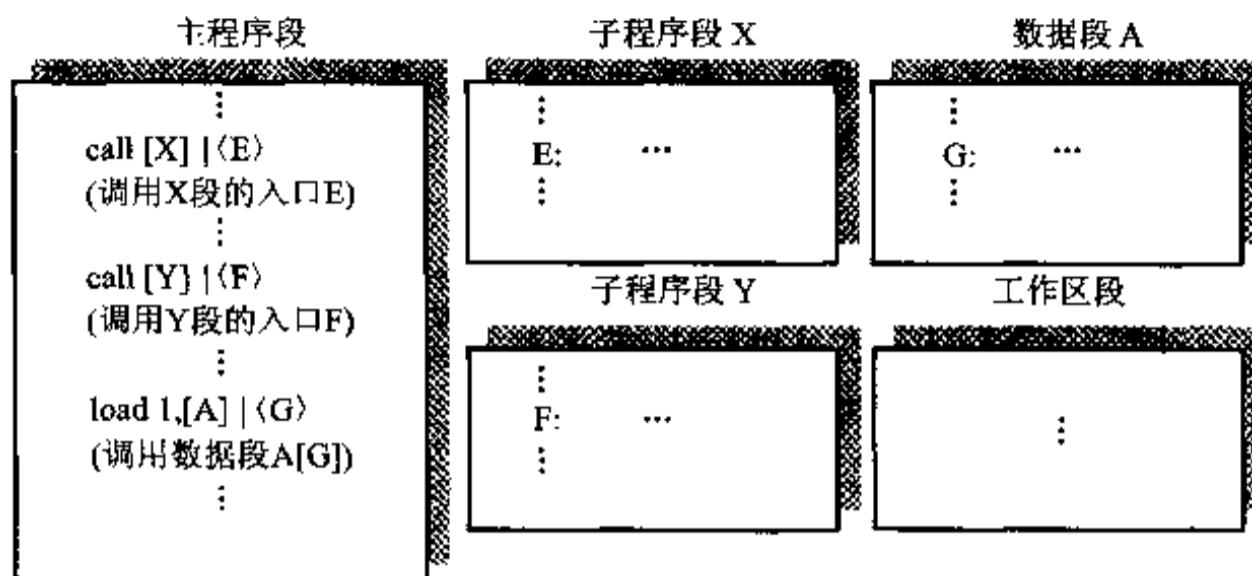
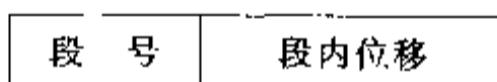


图 4.16 程序的分段结构

在源程序中,可用符号形式(指出段名和入口)调用某段的功能,源程序经编译或汇编后,仍按照自身逻辑关系分为若干段,每段有一个段号,段之间的地址不一定连续,而段内地址是连续的。可见这是二维地址结构,模块化的程序被装入物理地址空间后,仍保持二维地址结构,这种地址结构需要编译程序的支持,但对程序员而言是透明的。

4.4.2 分段存储管理的基本原理

分段存储管理把进程的逻辑地址空间分成多段,提供如下形式的二维逻辑地址:



在分页存储管理中,页的划分,即逻辑地址划分为页号和页内位移,是用户不可见的,连续的地址空间将根据页面的大小自动分页;而在分段存储管理中,地址结构是用户可见的,用户知道逻辑地址如何划分为段和段内位移,在设计程序时,段的最大长度由地址结构规定,程序中所允许的最多段数会受到限制。例如,PDP-11/45 的段地址结构为:段号占 3 位,段内位移占 13 位,一个作业最多分为 8 段,各段长度可达 8 KB。

分段存储管理的实现基于可变分区存储管理的原理。可变分区以整个作业为单位来划分和连续存放,也就是说,作业在分区内是连续存放的,但独立作业之间不一定连续存放。而分段方法是以段为单位来划分和连续存放,为作业的各段分配一个连续的主存空间,而各段之间不一定连续。在进行存储分配时,应为进入主存的作业建立段表,各段在主存中的情况可由段表来记录,它指出主存中各分段的段号、段起始地址和段长度。在撤销进程时,回收所占用的主存空间,并清除此进程的段表。

段表表项实际上起到基址/限长寄存器的作用,进程运行时通过段表可将逻辑地址转换成物

理地址,由于每个用户作业都有自己的段表,地址转换应按各自的段表进行。类似于分页存储管理,也设置一个硬件——段表基址寄存器,用来存放当前占用处理器的作业段表的起始地址和长度。分段存储管理的地址转换和存储保护流程如图 4.17 所示,将段控制寄存器中的段表长度与逻辑地址中的段号进行比较,若段号超过段表长度则触发越界中断,再利用段表项中的段长与逻辑地址中的段内位移进行比较,检查是否产生越界中断。

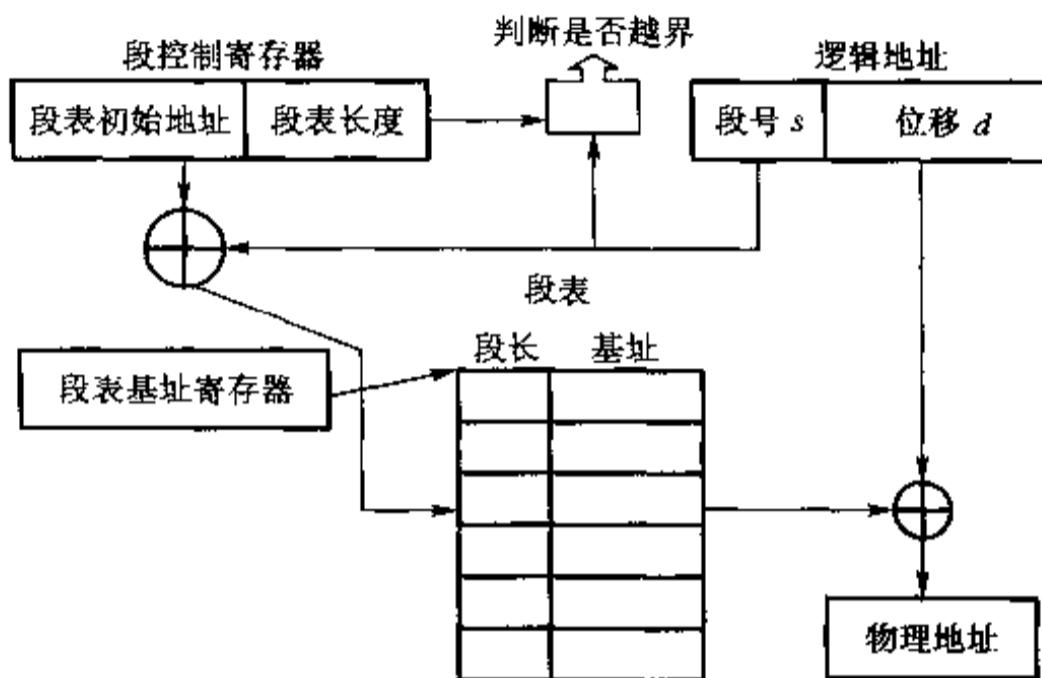


图 4.17 分段存储管理的地址转换和存储保护

4.4.3 段的共享和保护

与分页共享类似,只包含数据的段的共享不成问题,共享的数据段在作业进程的段表中可指定不同的段号。对于代码段,面临着与分页一样的困难,也涉及指令和数据的地址问题,要求所有共享函数段在所有作业的逻辑地址空间中拥有同样的段号。Intel x86 支持有限的段共享,每个用户进程的段表分成两个部分,局部描述符表 LDT 和全局描述符表 GDT,前者对应于进程私有段,后者包含操作系统及其他共享代码,GDT 表项所对应的分段可被所有进程共享。当然,必须对共享段的信息进行存取控制和保护,如规定只能读出不能写入,欲向此段写入信息时将遭到拒绝并触发保护中断。

4.4.4 分段和分页的比较

分段是信息的逻辑单位由源程序的逻辑结构及含义所决定,是用户可见的,段长由用户根据需要来确定,段起始地址可从任何主存地址开始。在分段方式中,源程序(段号、段内位移)经链接装配后仍保持二维(地址)结构,引入目的是满足用户模块化程序设计的需要。

分页是信息的物理单位与源程序的逻辑结构无关,是用户不可见的,页长由系统(硬件)确定,页面只能从页大小的整数倍地址开始。在分页方式中,源程序(页号、页内位移)经链接装配后变成一维(地址)结构,引入目的是实现离散分配并提高主存利用率。

4.5 虚拟存储管理

4.5.1 虚拟存储器的概念

前面所介绍的存储管理称为实存管理,必须为进程分配足够的主存空间,装入其全部信息,否则进程无法运行。把进程的全部信息装入主存后,实际上并非同时使用,有些部分运行一遍,有些部分甚至从不使用,进程在运行时不用的,或暂时不用的,或某种条件下才用的程序和数据,全部驻留下主存是对宝贵的存储资源的一种浪费,会降低主存利用率,这种做法很不合理。于是,提出新的想法:不必装入进程的全部信息,仅将当前使用部分先装入主存,其余部分存放在磁盘中,待使用时由系统自动将其装进来,这就是虚拟存储管理技术的基本思路。当进程所访问的程序和数据在主存中时,可顺利执行;如果处理器所访问的程序或数据不在主存中,为了继续执行,由系统自动将这部分信息从磁盘装入,这叫做“部分装入”;如若此刻没有足够的空闲物理空间,便把主存中暂时不用的信息移至磁盘,这叫做“部分替换”。如果“部分装入、部分替换”能够实现,那么,当主存空间小于进程的需要量时,进程也能运行;更进一步地,当多个进程的总长超出主存总容量时,也可将进程全部装入主存,实现多道程序运行。这样,不仅能充分地利用主存空间,而且用户编程时不必考虑物理空间的实际容量,允许用户的逻辑地址空间大于主存物理地址空间,对于用户而言,好像计算机系统具有一个容量硕大的主存储器,称其为“虚拟存储器”(virtual memory)(Fotheringham, 1961 年)。

虚拟存储器可定义如下:在具有层次结构存储器的计算机系统中,自动实现部分装入和部分替换功能,能从逻辑上为用户提供一个比物理主存容量大得多的、可寻址的“主存储器”。实际上,虚拟存储器对用户隐蔽可用物理存储器的容量和操作细节,虚拟存储器的容量与物理主存大小无关,而受限于计算机的地址结构和可用的磁盘容量,如 Intel x86 的地址线是 32 位,则程序可寻址范围是 4 GB, Windows 和 Linux 都为应用进程提供一个 4 GB 的逻辑主存。

如图 4.18 所示是虚拟存储器的概念图,其中,逻辑地址是从进程角度所看到的逻辑主存单元,而物理地址是从处理器角度所看到的物理主存单元,虚拟地址可以说是将逻辑地址映射到物理地址的一种手段。逻辑地址空间是程序员的编程空间,物理地址空间是程序的执行空间,虚拟地址空间等同于实际物理主存加部分硬盘区域所组成的存储空间。

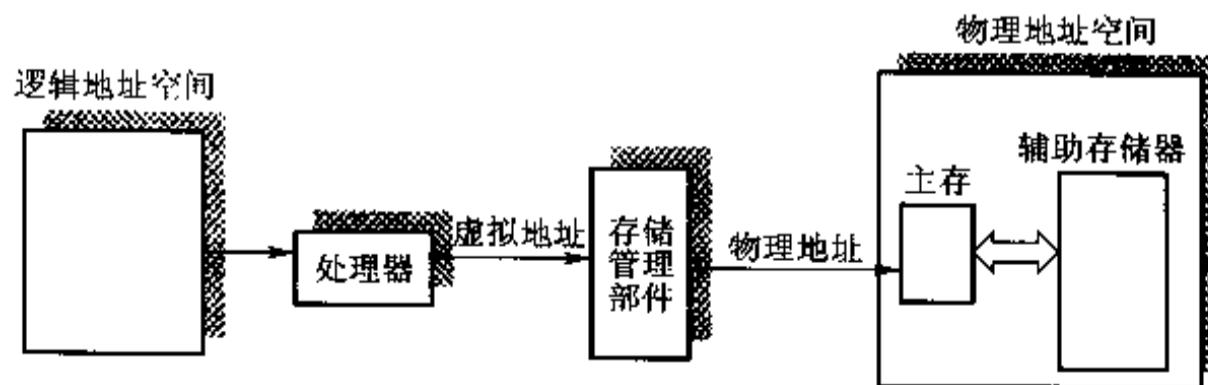


图 4.18 虚拟存储器概念图

下面讨论进程信息不全部装入主存时能否正确运行。早在 1968 年, P. Denning 研究程序执行时的局部性原理,对此进行研究的人士还有 Knuth(分析一组学生的 FORTRAN 程序)、Tanenbaum(分析操作系统的部分)、Huck(分析通用科学计算程序),发现程序和数据的访问都有聚集成群的倾向。某存储单元被使用之后,其相邻的存储单元也很快会被使用(称为空间局部性, spatial locality),或者最近访问过的程序代码和数据很快又被访问(称为时间局部性, temporal locality)。对程序的执行进行分析可以发现:第一,程序中只有少量分支和过程调用,大都是顺序执行的指令;第二,程序往往含有若干循环结构,由少量代码组成,而被多次执行;第三,过程调用的深度限制在小范围内,因而,指令引用通常被局限在少量的过程中;第四,许多计算涉及数组、记录之类的数据结构,对它们的连续引用是对位置相邻的数据项的操作;第五,程序中有些部分彼此互斥,不是每次运行时都用到,例如,出错处理部分在正常情况下用不到。种种情况说明,程序具有局部性,进程运行时没有必要把全部信息调入主存,只装入一部分进程信息的假设是合理的,此时只要调度得当,不仅可正确运行进程,而且能在主存中放置更多的进程,充分利用处理器和存储空间。虚拟存储器是基于程序局部性原理的一种假想的而非物理存在的存储器,其主要任务是:基于程序的局部性特点,当进程使用某部分地址空间时,保证将相应部分加载至主存中。这种将物理空间和逻辑空间分开编址、互相隔离,但又统一管理和使用的技术为用户编程提供了极大的方便。

虚拟存储管理与对换技术虽说都是在主存储器和磁盘之间交换信息,但却存在很大区别。对换技术以进程为单位,当其所需主存空间大于当前系统的拥有量时,进程无法被对换进主存工作;而虚拟存储管理以页或段为单位,即使进程所需主存空间大于当前系统拥有的主存总量,仍然能正常运行,因为系统可将其他进程的一部分页或段换出至磁盘。

虚拟存储器的思想早在 20 世纪 60 年代初就在英国 Atlas 计算机上出现,到 20 世纪 60 年代中期,较完整的虚拟存储器在分时系统 MULTICS 和 IBM 系列操作系统中得到实现,20 世纪 70 年代开始推广应用,逐步为广大计算机研制者和用户所接受。这项技术不仅用于大型计算机,也逐步被用到微型计算机系统中。为了实现虚拟存储器,必须解决好以下问题:主存与辅助存储器(磁盘)统一管理问题、逻辑地址到物理地址的转换问题、部分装入和部分替换问题。实现虚拟存储器要付出一定的开销,其中包括:管理地址转换的各种数据结构所用的存储开销、执行地址转换指令所花费的时间开销和主存与辅助存储器交换页或段的 I/O 开销等。目前,虚拟存储管理主要采用以下几种技术实现:请求分页、请求分段和请求段页虚拟存储管理。

4.5.2 请求分页虚拟存储管理

1. 请求分页虚拟存储管理的硬件支撑

操作系统的存储管理依靠低层硬件的支撑来完成任务,此硬件称为主存管理部件(Memory Management Unit, MMU),它提供地址转换和存储保护的功能,并支持虚拟存储管理和多任务管理。MMU 由一组集成电路芯片组成,逻辑地址作为输入,物理地址作为输出,直接送达总线,对主存单元进行寻址,其位置和功能如图 4.19(a)所示,其内部执行过程如图 4.19(b)所示。主要

功能列举如下。

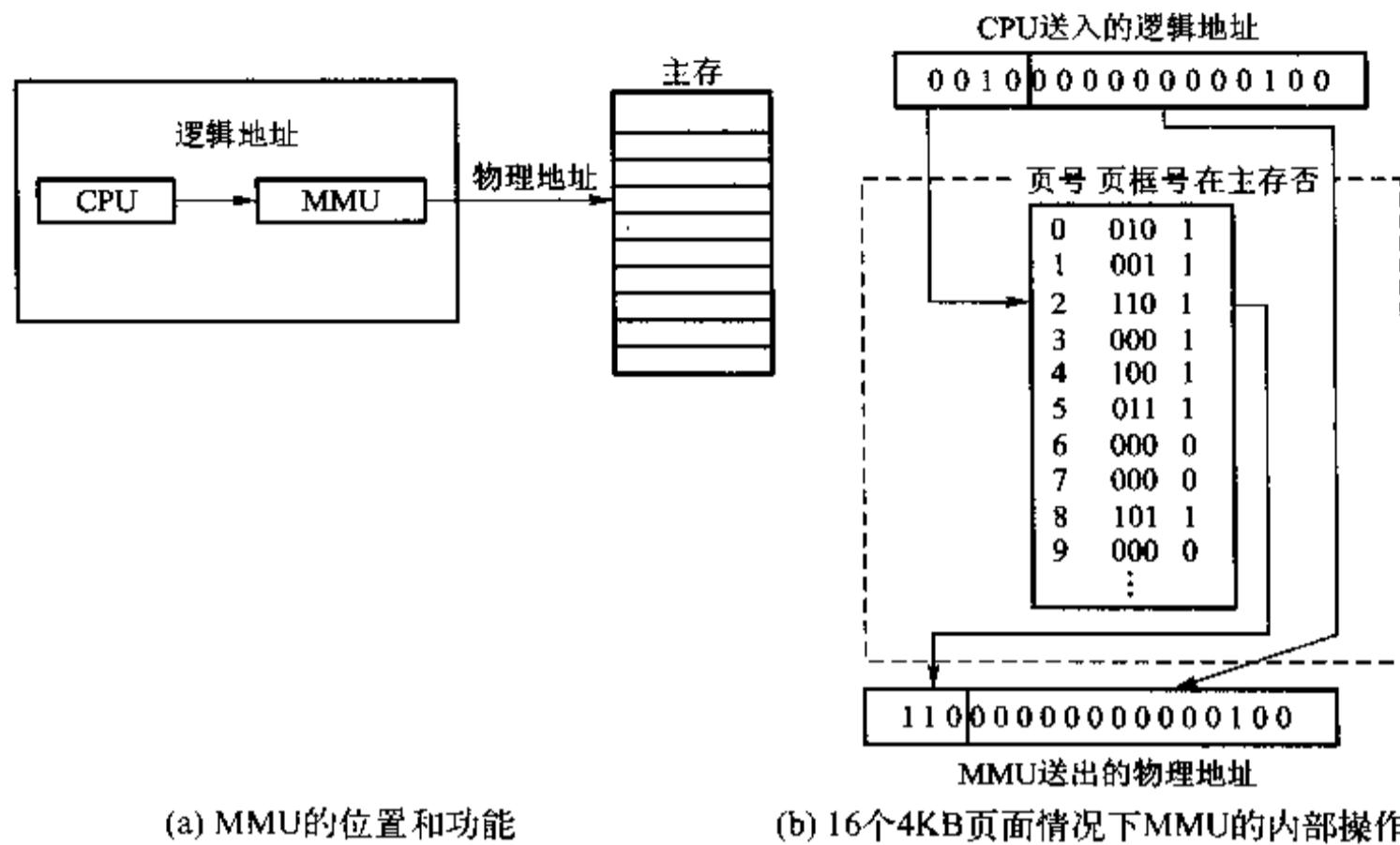


图 4.19 主存管理部件

(1) 管理硬件页表基址寄存器

每当发生进程上下文切换时,系统负责把将要运行的进程的页表基址装入硬件页表基址寄存器,此页表便成为活动页表,MMU 只对由硬件页表基址寄存器所指出的活动页表进行操作。

(2) 分解逻辑地址

把逻辑地址分解为页面号和页内位移,以便进行地址转换。

(3) 管理快表

对 TLB 进行管理,一是直接查找快表,找到相应的页框后去拼接物理地址;二是执行 TLB 的基本操作:装入表目和清除表目,每次发生快表查找不到命中的情况后,待缺页中断处理结束,把相应的页面和页框号装入。此外,每次写硬件页表基址寄存器时,负责清除快表项,将 TLB 清空。

(4) 访问页表

当 TLB 不命中时,根据页表基址寄存器直接访问进程页表,若所需页面已装入,则可访问主存完成指令,同时,把此页面信息装入 TLB。

(5) 发出相应中断

当查出页表中有页失效位或页面访问越界位时,发出缺页中断或越界中断,并将控制权交给内核存储管理。

(6) 管理特征位

负责设置和检查页表中的引用位、修改位、有效位和保护权限位等各个特征位。

下面考察 MMU 的工作过程。在图 4.19(b)中,给出一个逻辑地址 8196(二进制表示为 001000000000100),MMU 进行映射,输入的 16 位逻辑地址被解释为 4 位页号和 12 位页内位移。用页号作为索引,找出虚页所对应的页框号,如果“在主存否”位为 1,表明此页在主存,把页框号复制到输出寄存器的高 3 位,再加上逻辑地址中的 12 位页内位移,生成 15 位的物理地址(二进制表示为 110000000000100),并把它送到主存总线;如果“在主存否”位为 0,则将触发缺页中断,陷入操作系统进行调页处理。

2. 请求分页虚拟存储管理的基本原理

请求分页虚拟存储管理是将进程信息的副本存放在辅助存储器中,当它被调度投入运行时,并不把程序和数据全部装入主存,仅装入当前使用的页面,进程执行过程中访问到不在主存的页面时,再把所需信息动态地装入。使用频率较高的分页虚拟存储管理是请求分页(demand paging),当需要执行某条指令或使用某个数据而发现它们不在主存时,产生缺页(page fault)异常,系统从磁盘中把此指令或数据所在的页面装入,这样做能够保证用不到的页面不会被装入主存。

请求分页虚拟存储管理与分页实存管理不同,仅让进程当前使用部分装入,必然会发生某些页面不在主存中的情况。那么,如何发现页面不在主存中呢?所采用的办法是:扩充页表项的内容,增加驻留标志位,又称页失效异常位,用来指出页面是否装入主存。当访问一个页面时,如果某页的驻留标志位为 1,表示此页已经在主存中,可被正常访问;如果某页的驻留标志位为 0,不能立即访问,产生缺页异常,操作系统根据磁盘地址将这个页面调入主存,然后重新启动相应的指令。那么,如何查找页面对应的磁盘地址呢?磁盘的物理地址由磁盘机号、柱面号、磁头号和扇区号所组成,通常规定扇区的长度等于页面的长度,页面与磁盘物理地址的对应表称外页表,由操作系统管理。进程启动运行前系统为其建立外页表,并把进程程序页面装入辅助存储器,外页表也按进程页号的顺序排列。为了节省主存空间,外页表可存放在磁盘中,当发生缺页中断需要查用时才被调入。

缺页异常是由于发现当前访问页面不在主存时由硬件所产生的一种特殊中断信号,CPU 通常在一条指令执行完成后才检查是否有中断到达,也就是说只能在两条指令之间响应中断,但缺页中断却是在指令执行期间产生并获得系统处理的。而且,一条指令可能涉及多个页面,例如,指令本身跨页、指令所处理的数据跨页,完全有可能在执行一条指令的过程中发生多次缺页异常。

为了对页面实施保护和淘汰等各种控制,可在页表中增加标志位,其他标志位包括修改位、引用位和访问权限位等,用来跟踪页面的使用情况。当页面被修改后,硬件自动设置修改位,一旦修改位被设置,当此页被调出主存时必须先被写回磁盘;引用位则在页面被引用即无论是读或写时设置,其值帮助系统进行页面淘汰;访问权限位则限定页面允许的访问权限(如读、写、执行等)。

可见,在请求分页虚拟存储管理中,页表中存放的是把逻辑地址转换成物理地址时硬件所需要的信息,其作用主要有:

- (1) 获得页框号以实现虚实地址转换；
- (2) 设置各种访问控制位，对页面信息进行保护；
- (3) 设置各种标志位来实现相应的控制功能（如缺页标志、脏页标志、访问标志、锁定标志和淘汰标志等）。

下面讨论使用快表但是页表存放在主存的情况下，请求分页虚实地址转换的过程：当进程被调度到 CPU 上运行时，操作系统自动把此进程 PCB 中的页表起始地址装入硬件页表基址寄存器中，此后，进程开始运行并要访问某个虚地址，MMU 工作，它将完成图 4.20 虚线框内的任务，可看到地址转换过程如下。

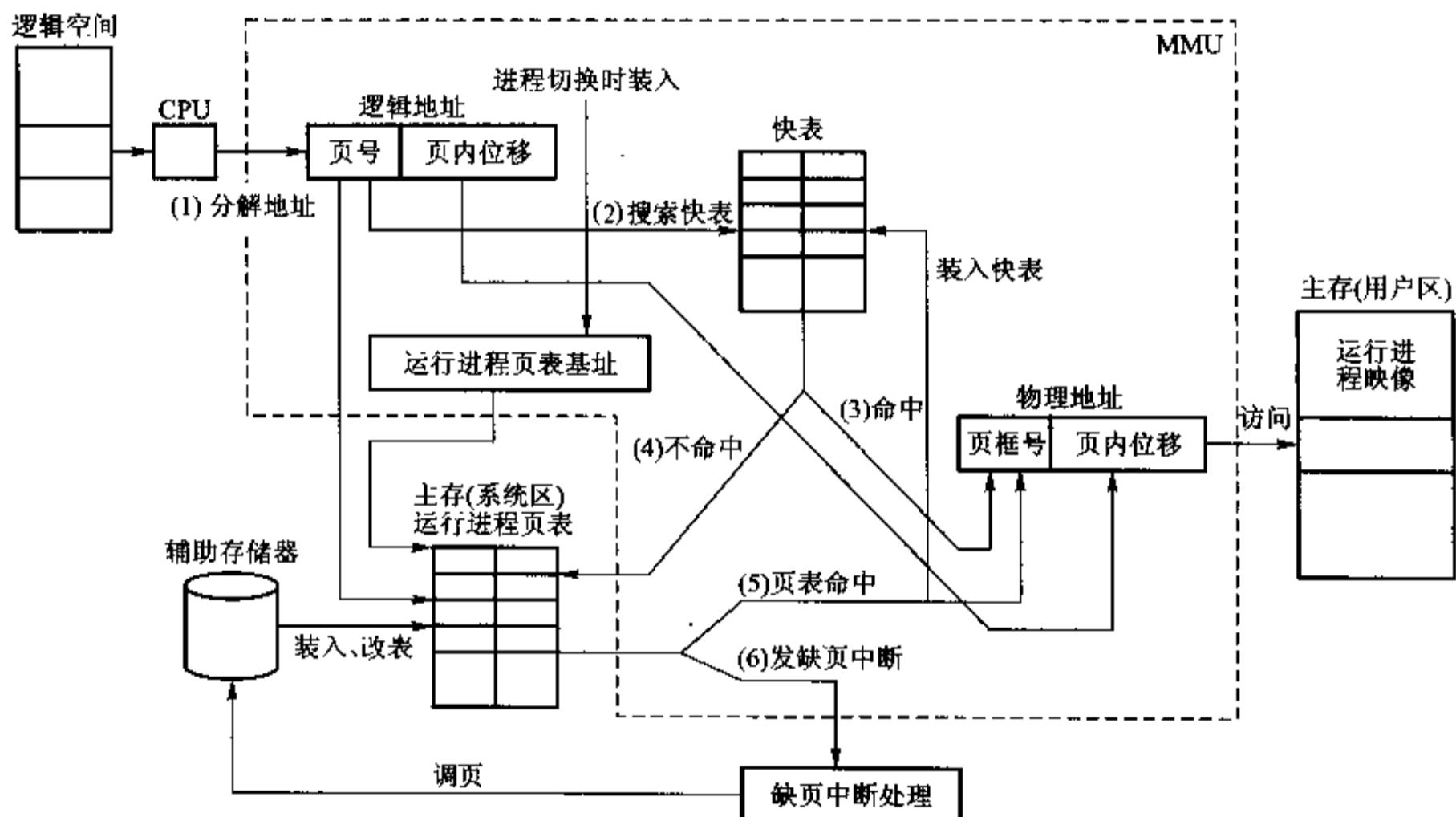


图 4.20 请求分页虚实地址转换

- (1) MMU 接收 CPU 传送过来的逻辑地址并自动按页面大小把它从某位起分解成两部分：页号和页内位移；
- (2) 以页号为索引搜索快表 TLB；
- (3) 如果命中，立即送出页框号，并与页内位移拼接成物理地址，然后进行访问权限检查，如获通过，进程就可以访问物理地址；
- (4) 如果不命中，由硬件以页号为索引搜索进程页表，页表基址由硬件页表基址寄存器指出；
- (5) 如果在页表中找到此页面，说明所访问页面已在主存中，可送出页框号，并与页内位移拼接成物理地址，然后进行访问权限检查，如获通过，进程就可以访问物理地址，同时要把这个页

面的信息装入快表 TLB,以备再次访问;

(6) 如果发现页表中的对应页面失效,MMU 发出缺页中断,请求操作系统进行处理,MMU 工作到此结束。

MMU 发现缺页并发出缺页中断,存储管理接收控制,进行缺页中断处理的过程如下。

步骤 1:挂起请求缺页的进程;

步骤 2:根据页号搜索外页表,找到存放此页的磁盘物理地址;

步骤 3:查看主存是否有空闲页框,如有则找出,修改主存管理表和相应页表项的内容,转步骤 6;

步骤 4:如果主存中无空闲页框,按照替换算法选择淘汰页面,检查其是否被写过或修改过,若否则转步骤 6;若是则转步骤 5;

步骤 5:淘汰页面被写过或修改过,将其内容写回磁盘的原先位置;

步骤 6:进行调页,把页面装入主存所分配的页框中,同时修改进程页表项;

步骤 7:返回进程断点,重新启动被中断的指令。

在分页虚拟存储系统中,由于页面在需要时是根据进程的请求装入主存的,因此称为请求分页虚拟存储管理,IBM/370 系统的 VS/1、VS/2 和 VM/370, Honeywell 的 MULTICS 以及 UNIVAC 系列 70/64 的 VMS 等都采用请求分页虚拟存储管理。这种技术的优点是:进程的程序和数据可按页分散存放在主存中,既有利于提高主存利用率,又有利于多道程序运行。这种技术的缺点是:要有一定的硬件支持,要进行缺页中断处理,机器成本增加,系统开销加大,此外,页内会出现碎片,如果页面较大,则主存的损失仍然很大。

3. 页面装入策略和清除策略

页面装入策略决定何时把页面装入主存,有两种策略可供选择:请页式(demand paging)和预调式(prepaging)。

请页式是仅当需要访问程序和数据时,通过缺页中断并由缺页中断处理程序分配页框,把所需页面装入主存。进程运行时缺页中断很频繁,随着越来越多的页面被装入主存,根据局部性原理,大多数将要访问的程序和数据页面都在最近被装入主存,于是,缺页中断率就会下降。这一策略的优点是确保只有被访问的页面才会调入主存,节省主存空间;其缺点是处理缺页中断的次数多,调页的系统开销大;由于每次仅调用一页,磁盘 I/O 操作次数猛增。

预调式装入主存的页面并非缺页中断所请求的页面,是由操作系统依据某种算法,动态预测进程最可能要访问的那些页面。在使用页面前预先调入主存,尽量做到进程要访问的页面已经调入主存,且每次调入若干页面,而不是仅调入一页。由于进程的页面大多连续存放在磁盘中,一次调入多个连续存放的页面能够减少磁盘 I/O 启动次数,节省寻道和搜索的时间。但是,如果所调入的页面大多未被使用,则效率就很低,可见,预调页要建立在可靠预测的基础之上。Windows 系统使用簇式分页的预调度策略,根据物理主存储器的大小,代码页面会装入 3~8 页,数据页面会装入 2~4 页,这种期望是基于程序的局部性,访问到预取的页面而不会产生额外的缺页。

页面清除策略与装入策略相对应,要考虑何时把修改过的页面写回辅助存储器,常用的方法

是：请页式和预约式。请页式清除是仅当一页被选中进行替换且其被修改过，才把它写回磁盘。预约式清除是对于所有更改过的页面，在需要替换之前把它们都写回磁盘，可成批进行。对于预约式清除，写出的页仍然在主存中，直到页替换算法选中此页从主存中移出。但如若刚刚写回很多页面，在它们被替换之前，其中大部分又被修改过，那么预约式清除就毫无意义。对于请页式清除，写出一页是在读进新页之前进行的，它要成对操作，虽然仅需写回一页，但进程不得不等待两次 I/O 操作完成，可能会降低系统性能。

4. 页面分配策略

请求分页虚拟存储管理排除主存实际容量的约束，使更多的进程能同时多道运行，从而提高系统效率，但是，缺页处理要付出很大的代价，由于页面的调入和调出要增加 I/O 操作负担，因此应尽可能地减少缺页次数。那么，究竟如何为进程分配页框呢？当出现缺页时，页面替换算法的作用范围应局限于此进程的页面，还是主存中所有进程的页面？这两个问题均涉及进程驻留页面的管理。

如果在进程的生命周期中，保持页框数固定不变，称其为面定分配，在创建进程时，根据进程的类型和系统的规则决定页框数，只要有一个缺页中断产生，进程就会有一页被替换；如果在进程的生命周期中，所分得的页框数可变，称为可变分配，当进程运行的某一阶段缺页率较高，说明目前局部性较差，系统可多分页框降低缺页率，反之说明局部性较好，可减少所分配的页框数。

页面替换可采用两种策略：局部替换和全局替换。如果页面替换算法的作用范围是整个系统，不考虑进程属主，称为全局页面替换算法；如果页面替换算法的作用范围局限于进程自身，称为局部页面替换算法。要为每个进程维护一组页面，称其为工作集，其大小随进程的执行而变化，应自动地排除不再在工作集中的页面。在工作集大小会在进程运行期间发生较大变化时，全局算法比局部算法好，但是系统必须不断地确定应给每个进程分配的页框数，这是比较困难的任务。

面定分配往往和局部替换策略配合使用，进程运行期间分得的页框数不再改变，如果发生缺页中断，只能从进程在主存的页面中选出一页替换，以保证进程的页框总数不变。这种策略的难点在于：应给进程分配多少页框才合适。分配少了，缺页中断率高；分配多了，会使主存中能同时运行的进程数减少，进而造成处理器和其他设备空闲。常用的固定分配算法有平均分配、比例分配、优先权分配等。

可变分配往往和全局替换策略配合使用，为系统中的进程分配一定数目的页框，操作系统保留若干空闲页框。进程发生缺页时，从系统空闲页框中对其分配，把所缺页面调入此页框，这样产生缺页的进程的主存空间会逐渐增大，有助于减少系统的缺页中断总次数。当系统所拥有的空闲页框几近耗尽时，要从主存中选择页面淘汰，可以是主存中任一进程的页面，这样又会使某进程的页框数减少，缺页中断率上升。这种方法的难点在于选择哪个页面作替换，应用某一种淘汰策略选页时，并未确定哪个进程会失去页面。如果选择某个进程，此进程工作集的缩小会严重影响其运行，那么，这个选择就不是最佳的。

可变分配配合局部替换可克服可变分配配合全局替换时存在的缺点，实现要点如下：

(1) 新进程装入主存时,根据应用类型、程序要求,分配一定数目的页框,可用请页式或预调式完成这个分配;

(2) 当产生缺页中断时,从进程的页面中选择一个淘汰;

(3) 不时重新评价进程的缺页率,增加或减少分配给进程的页框数,以改善系统的总性能。

5. 缺页中断率

虚拟存储器能够给用户提供一个容量很大的存储空间,但当主存空间已满而又要装入新页时,必须按照预定算法把已在主存中的页面写回,这项工作称为页面替换。用来确定被淘汰页的算法称为淘汰算法。如果选用不合适的算法,会出现这样的现象:刚被淘汰的页面立即又要调用,而调入不久随即被淘汰,淘汰不久再被调入,如此反复,使得整个系统的页面调度非常频繁以致大部时间都花费在来回调度页面上,而不是执行计算任务,这种现象叫做“抖动”(thrashing),一个好的调度算法应减少和避免抖动现象。

首先定义缺页中断率,假定进程 P 共计 n 页,此进程分得的主存块为 m 块(m, n 均为正整数,且 $1 \leq m \leq n$),即主存中最多只能容纳进程的 m 页。如果进程 P 在运行中成功访问的次数为 S ,不成功访问的次数为 F ,则总的访问次数 $A = S + F$,定义

$$f = F/A$$

称 f 为缺页中断率。影响缺页中断率 f 的因素有:

(1) 主存页框数:进程所分得的块数多,缺页中断率低,反之缺页中断率就高;

(2) 页面大小:页面大,缺页中断率低,否则缺页中断率就高;

(3) 页面替换算法:算法的优劣影响缺页中断次数;

(4) 程序特性:程序局部性要好,它对缺页中断率有很大影响。例如,一个程序将 128×128 的数组置初值“0”,假定它仅分得一个主存块,页面尺寸为 128 个字,数组中的元素各行分别存放在一页中,开始时第一页在主存中。若程序按如下左边编写:

```
int A[128][128];
for(int j=0;j<128;j++)
    for(int i=0;i<128;i++)
        A[i][j]=0;
```

```
int A[128][128];
for(int i=0;i<128;i++)
    for(int j=0;j<128;j++)
        A[i][j]=0;
```

则每执行一次 $A[i][j]=0$ 将产生一次缺页,共产生 $(128 \times 128 - 1)$ 次缺页中断;若按如上右边重新编写这个程序,共只产生 $(128 - 1)$ 次缺页中断。显然,虚拟存储器的效率与程序局部性程度密切相关,局部性程度因程序而异。

6. 全局页面替换策略

在多道程序的正常运行过程中,属于不同进程的页面被分散存放在主存页框中,当发生缺页中断时,如果已无空闲页框,系统要选择一个驻留页而进行淘汰。在此讨论的是所有驻留页面都可作为置换对象的情况,而不管页面所属进程的全局页而置换算法。

(1) 最佳页面替换算法

1966 年,Belady 提出最佳页面替换算法(OPTimal replacement, OPT)。当要调入一页而必

须淘汰旧页时,应该淘汰以后不再访问的页,或距现在最长时间后要访问的页面。它所产生的缺页数最少,然而,却需要预测程序的页面引用串,这是无法预知的,不可能对程序的运行过程做出精确的断言,不过此理论算法可用做衡量各种具体算法的标准。

(2) 先进先出页面替换算法

基于程序总是按线性顺序来访问物理空间这一假设,总是淘汰最先调入主存的页面,即淘汰在主存中驻留时间最长的页面,认为驻留时间最长的页不再使用的可能性较大。先进先出页面替换算法(First-In First-Out replacement, FIFO)的一种实现方法是系统中设置一张具有 m 个元素的页号表:

$$P[0], P[1], \dots, P[m-1]$$

其中,每个 $P[i] (i = 0, 1, \dots, m-1)$ 存储一个装入主存中的页面的页号。假设用索引 k 指示当前调入新页时应淘汰页在页号表中的位置,则淘汰页的页号应是 $P[k]$ 。每当调入新页后,执行

$$P[k] = \text{新页的页号}; k = (k + 1) \% m;$$

假定主存中已经装入 m 页, k 的初值为 0,那么,第一次淘汰页的页号应为 $P[0]$,而调入新页后, $P[0]$ 的值为新页的页号, k 取值为 1;……;第 m 次淘汰页的页号为 $P[m-1]$,调入新页后, $P[m-1]$ 的值为新页的页号, k 取值为 0;第 $m+1$ 次页面淘汰时,淘汰页的页号为 $P[0]$ 所指向的页面,因为它是在主存中驻留时间最长的页面。

这种算法较易实现,对具有线性顺序特性的程序比较适用,面对具有其他特性的程序则效率不高,因为在主存中驻留时间最长的页面未必是最长时间后才使用的页面,很可能是最近要被访问的页。也就是说,如果某个页面经常被使用,采用 FIFO 算法,在一定时间后此页面变成驻留主存时间最长的页,这时若淘汰它,可能立即又要用到,必须重新调入。据估计,采用 FIFO 调度算法,缺页中断率为最佳算法的 2~3 倍。FIFO 算法还伴有一种奇怪的现象,称 Belady 异常,增加可用主存块的数量会导致更多的缺页。

页面缓冲算法是对 FIFO 替换算法的一种改进,其策略如下:系统维护两个 FIFO 队列,修改页面队列和非修改(空闲)页面队列,前者是由修改页面的页框所构成的链表;后者是由可直接用于装入页面的页框所构成的链表,只不过有些未修改的淘汰页暂时还留在其中,当进程再次访问这些页面时,可不经 I/O 操作而快速找回。当发生缺页中断时,按照 FIFO 算法选出淘汰页,并不立即抛弃它,而是根据其内容是否被修改过进入两个队列之一的末尾,需要装入的页面被读进非修改队列的队首所指向的页框中,不必等待淘汰页写回,使得进程能快速恢复运行。当选中的淘汰页被写回磁盘时,只需把此页占用的页框链接到非修改队列的末尾即可。每当修改页面队列中的页面达到一定数量时,将成批地写回磁盘,并把空闲页框加入非修改页面队列尾部。

(3) 最近最少使用页面替换算法

最近最少使用页面替换算法(Least Recently Used, LRU)淘汰的页面是在最近一段时间内最久未被访问的那一页,它是基于程序局部性原理来考虑的,认为那些刚被使用过的页面可能还要立即被使用,而那些在较长时间内未被使用的页面可能不会立即使用。

为了能准确地淘汰最近最少使用的页面,必须维护一个特殊的队列——页面淘汰队列,此队

列存放当前在主存中的页号,每访问一页时就调整一次,使队列尾总是指向最近访问的页,队列头就是最近最少使用的页,显然,发生缺页中断时总淘汰队列头所指的页;而执行页面访问后,需要从队列中把此页调整到队列尾。例如,给某进程分配 3 个主存块,依次访问的页号为:4,3,0,4,1,1,2,3,2。于是当访问这些页时,淘汰页面序列的变化情况如下所示:

访问页号	淘汰页面序列	被淘汰页面
4	4	
3	4,3	
0	4,3,0	
4	3,0,4	
1	0,4,,1	3
1	0,4,1	
2	4,1,2	0
3	1,2,3	4
2	1,3,2	

LRU 算法的实现需要硬件支持,关键是确定页面最后访问以来所经历的时间,可采用多种模拟方法。

模拟方法一是引用位法,又称最近未使用页面替换算法(Not Recently Used, NRU)。此方法为每页设置引用位 R,每次访问某页时,由硬件将此页的 R 位置 1,间隔时间 t,周期性地将所有页的 R 位清 0。页面置换时,从引用位 R 为 0 的那些页中挑选页面进行淘汰,在选中要淘汰的页后,也将其他页的引用位 R 清 0。这种实现方法开销小,但 t 的大小不易确定且精确性差。t 大了,缺页中断时所有页的 R 值均为 1;t 小了,缺页中断时可能所有页的 R 值均为 0,这样就很难挑选出应淘汰的页面,通常把 t 定为一个或数个时钟中断周期。

模拟方法二是计数器法,每当页面被引用时,硬件计数器自动计数,更换访问页面时,把硬件计数器的值记录到页表的计数值字段,经过时间 t 后,将所有计数器全部清除。页面置换时,系统检查所有页表项,计数值最小的页面就是最不经常使用的页,故称最不经常使用页面替换算法(Not Frequently Used, NFU)。

模拟方法三是记时法,为每页增设一个记时单元,每当页面被引用时,把系统的绝对时间置入记时单元。经过时间 t 后,将所有记时单元全部清除。页面置换时,系统对各页面的记时值进行比较,值最小的页面就是最久未使用的页面从而淘汰之。

(4) 第二次机会页面替换算法

FIFO 算法会把经常使用的页面淘汰掉,为了避免这一点,可对算法进行改造,把 FIFO 算法与页表中的“引用位”结合起来使用,实现思想如下:首先检查 FIFO 页面队列中的队首,这是最早进入主存的页面,如果其“引用位”是 0,那么这个页面既时间长又没有用,选择此页面淘汰;如果其“引用位”是 1,说明虽然它进入主存的时间较早,但最近仍在使用,于是将其“引用位”清 0,并把这个页面移至队尾,把它看做一个新调入的页,再给一次机会。这一算法称为第二次机会页

面替换算法(Second Chance Replacement, SCR),其含义是最先进入主存的页面如果最近还在被使用(其“引用位”总保持为1),仍然有机会像新调入页面一样留在主存中。如果主存中的页面都被访问过,即它们的“引用位”均为1,那么,第一遍检查把所有页面的“引用位”清0,第二遍又找出队首,并把此页面淘汰,此时,SCR算法便退化为FIFO算法。

(5) 时钟页面替换算法

如果利用标准队列机制构造FIFO队列,SCR算法可能产生频繁的出队和入队,实现代价较高,可采用循环队列机制构造页面队列,形成类似于钟表面的环形表,队列指针相当于钟表面上的表针,指向可能要淘汰的页面,这就是时钟页面替换算法(Clock policy replacement,Clock)的得名。此算法与SCR算法在本质上没有区别,仅仅是实现方法有所改进,仍然要使用页表中的“引用位”,把进程已调入主存的页面链接成循环队列,用指针指向循环队列中下一个将被替换的页面。算法实现要点如下:

- ①一个页面首次装入主存时,其“引用位”置0;
- ②主存中的任何一个页面被访问时,其“引用位”置1;
- ③淘汰页面时,存储管理从指针当前指向的页面开始扫描循环队列,把所遇到的“引用位”是1的页面的“引用位”清0,并跳过这个页面;把所遇到的“引用位”是0的页面淘汰,指针推进一步;
- ④扫描循环队列时,如果遇到所有页面的“引用位”均为1,指针就会环绕整个循环队列一圈,把碰到的所有页面的“引用位”清0;指针停在起始位置,并淘汰这一页,然后指针推进一步。

图4.21给出Clock算法的一个例子。当发生缺页中断时,将装入主存的页面是Page727,指针所指向的是Page45(在页框2中),Clock算法执行过程如下:Page45的“引用位”是1,它不能被淘汰,仅将其“引用位”清0,指针推进;同样道理,Page191(在页框3中)也不能被淘汰,将其“引用位”清0,指针继续推进;下一页Page556(在页框4中)的“引用位”是0,于是Page556被Page727替换,并把Page727的“引用位”置1,指针前进到下一页Page13(在页框5中),算法执行到此结束。

淘汰页面时,如果此页面已被修改过,必须将它先重新写回磁盘;但如果所淘汰的是未被修改过的页面,就不需要写磁盘操作,这样看来淘汰修改过的页面比淘汰未被修改过的页面的开销要大。如果把页表项的“引用位”和“修改位”结合起来使用,可以改进Clock算法,页面共组合成4种情况:

- ①最近未被引用,未被修改($r=0, m=0$);
- ②最近被引用,未被修改($r=1, m=0$);
- ③最近未被引用,但被修改过($r=0, m=1$);
- ④最近被引用,也被修改过($r=1, m=1$)。

于是,改进的Clock页面替换算法可如下执行:

步骤1:选择最佳淘汰页面。从指针当前位置开始扫描循环队列,扫描过程中不改变“引用位”,把遇到的第一个 $r=0, m=0$ 的页面作为淘汰页。

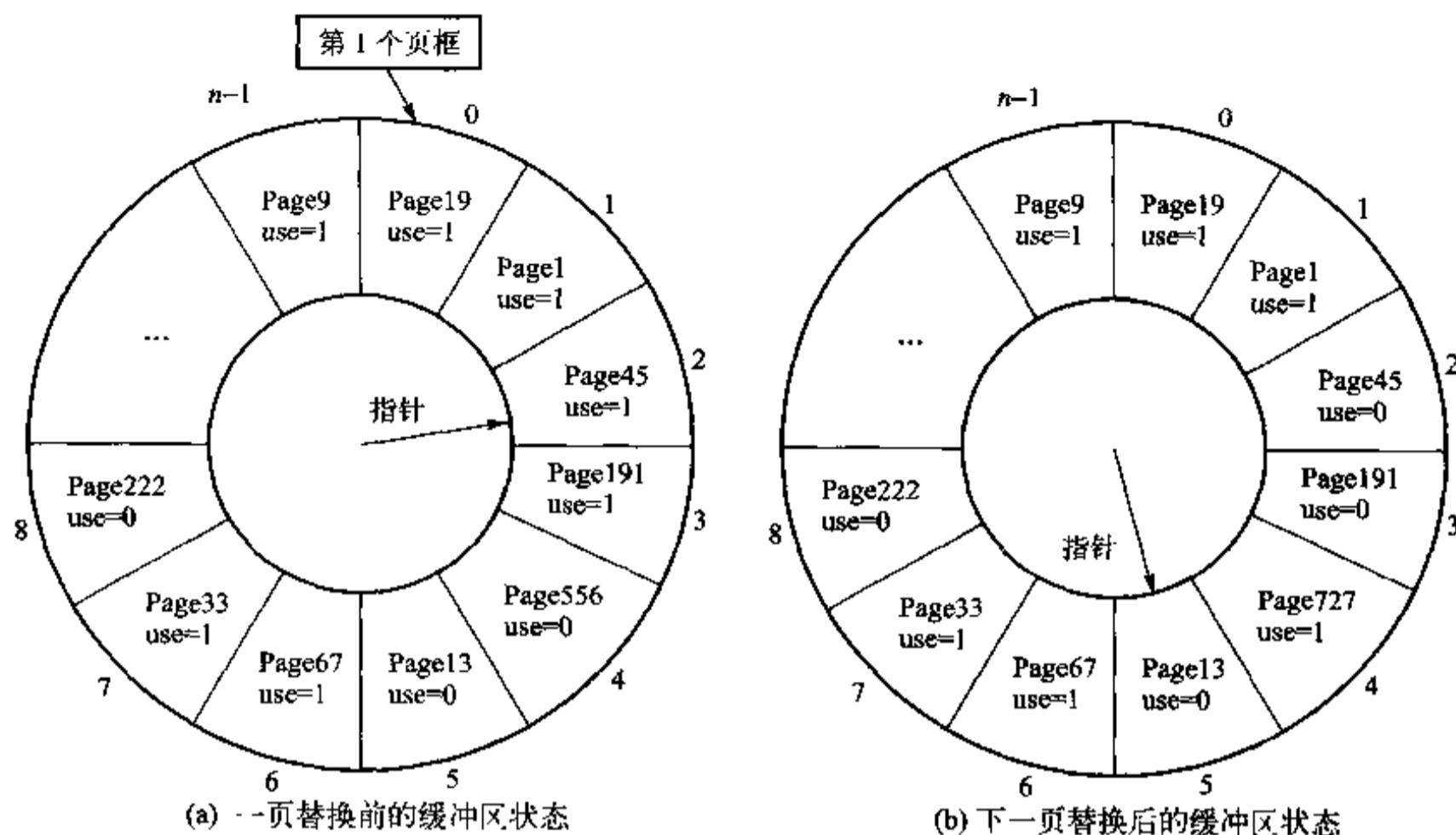


图 4.21 时钟页面替换算法

步骤 2: 如果步骤 1 失败, 再次从原位置开始, 查找 $r = 0, m = 1$ 的页面, 把遇到的第一个这样的页面作为淘汰页, 而在扫描过程中把指针所经过的页面的“引用位” r 置 0。

步骤 3: 如果步骤 2 失败, 指针再次回到起始位置, 由于此时所有页面的“引用位” r 均为 0, 再转向步骤 1 或步骤 2 操作, 这次一定能挑出一个可淘汰的页面。

改进的 Clock 页面替换算法就是扫描循环队列中的所有页面, 寻找既未被修改且最近又未被引用的页面作为首选页面淘汰, 因为未曾被修改过, 淘汰时不用把它写回磁盘; 如果步骤 1 失败, 算法再次扫描循环队列, 欲寻找一个被修改过但最近未被引用的页面, 虽然这种淘汰页面需要写回磁盘, 但依据程序局部性原理, 这类页面不会立刻被再次使用; 如果步骤 2 也失败, 则所有页面已被标记为最近未被引用, 可进入第三次扫描, 也称为“第三次机会时钟替换算法”, 因为一个被修改过的页面直到指针已经完成对队列的两次完全扫描之前, 将不会被移出, 与未被修改的页面相比, 它在被选中替换之前还有额外一次机会被引用。

UNIX SVR4 使用改进的 Clock 页面替换算法, 称双指针 Clock 算法, 实现思想如下: 主存中的每个页面都有“引用位”相连, 当一个页面被装入主存时, “引用位”置 0; 当一个页面被引用时, “引用位”置 1。系统设置两个时钟指针, 称为前指针和后指针, 页面守护进程被周期性地唤醒工作, 两个指针都扫描一遍: 前指针先依次扫过页面并把“引用位”置 0; 再延迟一定的时间之后, 后指针依次扫描页面, 若此页面的“引用位”为 1, 表明这个期间页面被引用过, 则跳过此页面继续扫描, 若此页面的“引用位”为 0, 表明页面未被引用过, 那么, 此页面可作为候选的淘汰者。双指针 Clock 算法的关键: 一是两个指针扫描页表的速度; 二是两个指针之间时间间隔的选择, 在系

统初始化时,可根据物理空间的大小为这两个参数设置默认值,速度参数可变以满足条件的变化。当空闲的存储空间较少时,两个指针移动速度加快以释放更多的页面。

下面给出一个例子,分别用 OPT、FIFO、LRU 和 Clock 页面替换算法来计算缺页次数和被淘汰的页面,并对性能作简单的比较。进程分得 3 个页框,执行过程中按下列次序引用 5 个独立的页面:2,3,2,1,5,2,4,5,3,2,5,2。如图 4.22 所示是 4 种算法的计算过程和结果。访问前 3 个页面 2,3,1 必产生缺页,于是,OPT 算法共产生 6 次缺页中断,第 4 次淘汰 Page1,因为它以后不再使用;第 5 次淘汰 Page2,它要在最远的将来才被再次使用;第 6 次淘汰 Page4,它以后不再被使用。LRU 算法共产生 7 次缺页中断,根据局部性原理,淘汰的页面依次为 Page3、Page1、Page2、Page4,最顶端一行是最近被访问的页面,最低部一行是最近最少被访问的页面。FIFO 算法共产生 9 次缺页中断,它认为进入主存时间最长的页面最不可能被引用,应该被淘汰,所淘汰的页面依次为 Page2、Page3、Page1、Page5、Page2、Page4,最顶端一行是最先进入主存的页面,最低部一行是最近进入主存的页面。Clock 算法共产生 8 次缺页中断,图中 * 表示相应页面的“引

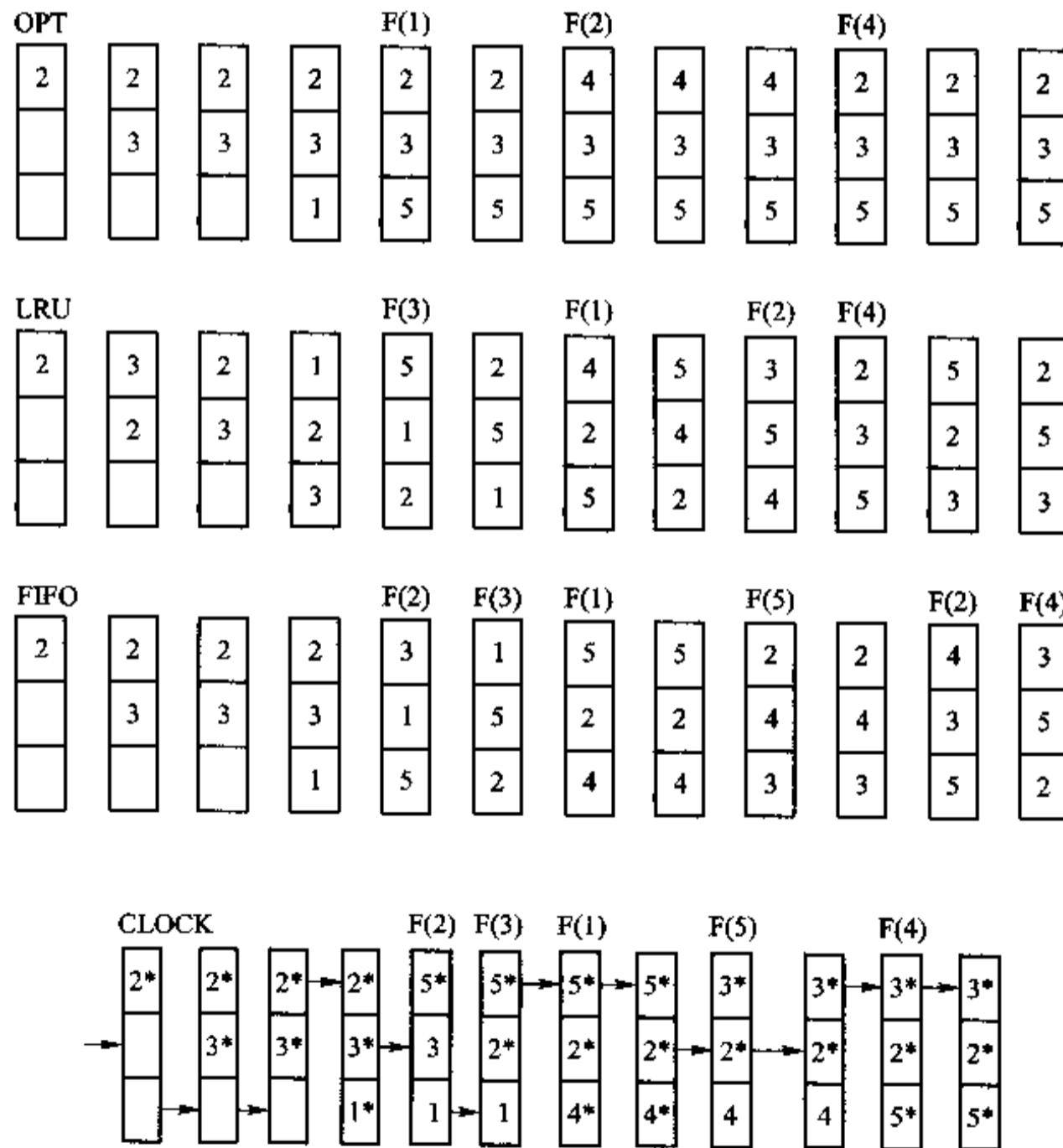


图 4.22 4 种算法的计算过程和结果

用位”为 1,箭头“ \rightarrow ”表示指针的当前位置,当第一次引用 Page5 时,由于此时循环队列中所有页面的“引用位”均为 1,所以指针绕过一圈并指向 Page2,故 Page5 替换 Page2,同时 Page3 和 Page1 的“引用位”置 0;接着引用 Page2 时,很容易看出应淘汰 Page3,所以 Page2 替换 Page3;同样,当引用 Page4 时,Page4 替换 Page1;第二次引用 Page3 时,循环队列中所有页面的“引用位”再次为 1,因此,指针绕过一圈后 Page3 替换 Page5;当再次引用 Page2 时,循环队列中 3 个页面:Page3 的“引用位”为 1、Page2 的“引用位”为 1 和 Page4 的“引用位”为 0,且指针指向 Page2,所以当第三次引用 Page5 时,显然应替换 Page4。可以看出 FIFO 算法的性能最差,OPT 算法的性能最好,而 Clock 算法与 LRU 算法的性能十分接近。

7. 局部页面替换算法

请求分页虚拟存储管理的目标是:找出满足当前进程访问的局部性所需要的页面,然后,将这些页面加载到主存中,随着程序执行阶段的变化,从一个局部集转到另一个局部集,原来局部集的页面将从主存中卸载,包含新的局部集的页面会被加载到空出来的页框中。类似的情况会出现在程序访问数据的不同部分,当某个给定的进程引起缺页时,不允许通过缩小其他进程的驻留页面集来解决问题。下面所讨论的替换算法能用来解决这个难点。

(1) 局部最佳页面替换算法

1976 年,Prieve 提出局部最佳页面替换算法(local Minimum replacement,MIN),与全局最佳替换算法类似,需要预知程序的页面引用串,再根据进程行为改变驻留页面数量,实现思想如下:进程在时刻 t 访问某页面,如果此页面不在主存中,将导致一次缺页,把此页面装入一个空闲页框。无论发生缺页与否,算法在每一步都要考虑引用串,如果此页面在时间间隔 $(t, t + \tau)$ 内未被再次引用,那么就移出;否则,此页面被保留在进程的驻留集中,直到再次被引用。 τ 是一个系统常量,间隔 $(t, t + \tau)$ 称作滑动窗口,因为在任意给定时刻,驻留集包含这个窗口中可见的那些页面(当前引用的页面、未来的 τ 次主存访问引用的页面),因此,窗口的实际大小为 $\tau + 1$ 。

通过例子说明此算法,假如进程页面引用串为 P3,P3,P4,P2,P3,P5,P3,P5,P1,P4,滑动窗口 $\tau = 3$,初始时页面 P4 已被装入,若采用局部替换,通过图 4.23 来了解驻留集的变化情况。在时刻 $t = 0$,P4 被引用,因为它在时刻 $t = 3$ 再次被引用,即在时间间隔 $(0, 0 + 3)$ 之内,故 P4 留在驻留集;在时刻 $t = 1$,P3 被引用,它被装入空闲页框中,这时驻留集中包含 P3 与 P4,在时刻 $t = 2$ 和 $t = 3$,显然,页面 P3 与 P4 被保留;页面 P4 在时刻 $t = 4$ 被移出驻留集,因为在时间间隔 $(4, 4 + 3)$ 之内不再被引用;同时,P2 被装入空闲页框,但 P2 在时刻 $t = 5$ 就脱离滑动窗口并移出驻留集,而 P3 依然驻留,直到时刻 $t = 7$ 再次被引用;发生在时刻 $t = 6$ 的下次缺页把 P5 装入页框,它被保持驻留,直到时刻 $t = 8$ 再次被引用;最后两次引用装入页面 P1 和 P4。本例中,缺页总数为 5 次,驻留集大小在 1~2 之间变化,任何时刻至多有两个页框被占用,通过增加 τ 值,可减少缺页数目,但其代价是花费更多的页框。

(2) 工作集模型和工作集置换算法

P.J.Denning 提出工作集(Working Set,WS)模型,用来对局部最佳页面替换算法进行模拟实现,也使用滑动窗口的概念,但并不向前查看页面引用串,而是基于程序局部性原理向后看,这

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	-	-	-	-	-	-	-	-	-	✓	-
P2	-	-	-	-	✓	-	-	-	-	-	-
P3	-	✓	✓	✓	✓	✓	✓	✓	-	-	-
P4	✓	✓	✓	✓	-	-	-	-	-	-	✓
P5	-	-	-	-	-	✓	✓	✓	-	-	-
In _t		P3			P2		P5			P1	P4
Out _t					P4	P2			P3	P5	P1

图 4.23 局部最佳页面替换算法示例

意味着在任何给定的时刻,一个进程不久的将来所需主存页框数可通过考查其最近时间内的主存需求做出估计。

进程工作集指“在某一段时间间隔内进程运行所需访问的页面集合”,用 $W(t, \Delta)$ 表示在时刻 $t - \Delta$ 到时刻 t 之间所访问的页面集合,它就是进程在时刻 t 的工作集。变量 Δ 称为“工作集窗口尺寸”,可通过窗口来观察进程的行为。工作集中所包含的页面数目称为“工作集尺寸”。

在图 4.24 的例子中,页面引用串与上例相同,工作集窗口尺寸 $\Delta = 3$ 。如果系统有空闲页框供分配,并且在时刻 $t = 0$ 时,初始工作集为 $(P1, P4, P5)$,其中, $P1$ 在时刻 $t = 0$ 被引用, $P4$ 在时刻 $t = -1$ 被引用,而 $P5$ 在时刻 $t = -2$ 被引用。第一次缺页中断发生在时刻 $t = 1$,页面 $P3$ 被装入一个空闲页框,另外 3 个当前驻留页面 $P1$ 、 $P4$ 和 $P5$ 在窗口 $(1 - 3, 1)$ 中仍然可见,并被保留;在时刻 $t = 2$,页面 $P5$ 离开当前窗口 $(2 - 3, 2)$,它被移出工作集;在时刻 $t = 4$,缺页中断会把 $P2$ 装入,它占用移出的页面 $P1$ 的位置,因为 $P1$ 已离开当前窗口 $(4 - 3, 4)$;在时刻 $t = 6$,发生缺页中断并装入 $P5$,并且当前驻留页面 $P2$ 、 $P3$ 和 $P4$ 作为由当前窗口 $(6 - 3, 6)$ 定义的当前工作集的一部分被保留;在下面两次引用中,工作集会缩小到仅两个页面 $P3$ 和 $P5$,并因为在时刻 $t = 9$ 和 $t = 10$ 发生两次缺页中断,使工作集再次增长到 4 个页面。此算法总的缺页数为 5 次,工作集尺寸在 2~4 个页框间波动。

工作集是程序局部性的近似表示,可通过它来确定驻留集的大小。

- ① 监视每个进程的工作集,只有属于工作集的页面才能驻留在主存;
- ② 定期地从进程驻留集中删去那些不在工作集中的页面;
- ③ 仅当一个进程的工作集在主存时,进程才能执行。

Windows 的页面替换机制结合工作集模型和 Clock 页面替换算法的优点,采用局部替换算法,进程缺页时,不会逐出其他进程的页面。系统为每个进程维护一个工作集,系统指定工作集的最小尺寸(20~50 个页框)和最大尺寸(45~345 个页框);发生缺页中断时,把引用到的页面添

加至进程工作集中,直至达到最大值,此时,若还发生缺页中断,必须从工作集中移出一个页面;淘汰页面的选择使用模拟 LRU 和 Clock 策略的变种,每个页框有一个访问位 u 及一个计数器 $count$ 。此页被引用时, u 位被硬件置 1;当在工作集中寻找淘汰页面时,工作集管理程序扫描工作集中页面的访问位,并执行操作。如果 $u=1$,那么,把 u 和 $count$ 清 0;否则, $count$ 加 1,扫描结束时,将 $count$ 值最大的页面移出。从工作集中移出的页面被放入两个主存队列之一:一是保存暂时移出并已被修改过的页面;二是保存暂时移出并为“只读”的页面。如果其中的页面被再次引用,可迅速地从队列中找回,而不会产生缺页中断,仅当实际的空闲页框队列为空时,它们才被用来满足缺页需求。

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串	P1	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	✓	✓	✓	✓	-	-	-	-	✓	✓	
P2	-	-	-	-	✓	✓	✓	✓	-	-	-
P3	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P4	✓	✓	✓	✓	✓	✓	✓	-	-	-	✓
P5	✓	✓	-	-	-	-	✓	✓	✓	✓	✓
In _t		P3		P2		P5		P1	P4		
Out _t			P5	P1			P4	P2			

图 4.24 工作集替换示例

(3) 模拟工作集替换算法

工作集策略在概念上很有吸引力,但实现中监督驻留页面变化的开销却很大,估算合适的窗口 Δ 的大小也是一个难题,为此,已经设计出各种模拟工作集替换算法,下面介绍两种。

进程在运行前要把它的工作集预先装入主存,为每个页设置引用位 r 及年龄寄存器,寄存器初始值为 0,每隔时间 t ,系统扫描主存中的所有页面,先将寄存器右移一位,再把引用位 r 的值加到对应寄存器的最左边,这样,未引用页面其年龄寄存器的值逐步减小,当达到下限或值 0 时,由于页面已经落在窗口之外,就可以把它从工作集中移出去。

例如,年龄寄存器共有 4 位,时间间隔 t 定为 1 000 次存储器引用(即 1 000 个指令周期),页面 P 在时刻 $t+0$ 时年龄寄存器的值为“1 000”,在时刻 $t+1\ 000$ 时年龄寄存器的值为“0100”,在时刻 $t+2\ 000$ 时年龄寄存器的值为“0010”,在时刻 $t+3\ 000$ 时年龄寄存器的值为“0001”,在时刻 $t+4\ 000$ 时年龄寄存器的值为“0000”,此时,页面 P 被移出工作集,这就有效地模拟窗口大小为 $1\ 000 \times 4$ 的一个工作集。

修改后的算法称为“老化(aging)算法”,年龄寄存器各位的累加值反映页面最近使用的情况,访问次数越多,累加值越大,而较早被访问的页面随着寄存器各位的右移,由于老化使得其作

用越来越小。

另一种方法是为每个页面设置引用位及关联的时间戳，通过超时中断，至少每隔若干条指令就周期性地检查引用位及时间戳，当发现引用位为 1 时，就将其置 0 并把这次改变的时间作为时间戳记录下来。每当发现引用位为 0 时，通过系统当前时间减去时间戳的时间，计算出从上次使用以来未被再次访问的时间量，记作 t_{off} 。 t_{off} 值会随着每次超时中断的处理而不断增加，除非页面在此期间被再次引用，导致其引用位为 1。把 t_{off} 与系统时间参数 t_{max} 相比，若 $t_{\text{off}} > t_{\text{max}}$ ，就把页面从工作集中移出，释放相应的页框。

(4) 缺页频率替换算法

在工作集算法中，保证最少缺页次数是通过调整工作集的大小来间接实现的，一种直接改善系统性能的方法是使用缺页频率替换算法 (Page Fault Frequency, PFF)。这种算法根据连续的缺页之间的时间间隔来对缺页频率进行测量，每次缺页时，利用测量时间调整进程工作集尺寸。其规则是：如果本次缺页与前次缺页之间的时间间隔超过临界值 τ ，那么，在这个时间间隔内未引用的所有页而都被移出工作集。这就能保证进程工作集不会不必要地扩大，与工作集模型相比，实现效率高，只在发生缺页中断时才调整页面，而不是每次引用时都需要调整。

如图 4.25 所示的例子再次使用上述引用串，并设临界值 $\tau = 2$ ，在时刻 $t = 0$ ，驻留集合中包含页面 P1、P4 和 P5。在时刻 $t = 1$ 发生第一次缺页，假设前一次缺页刚刚发生，故本次无页而被移出；下次缺页发生在时刻 $t = 4$ ，因为本次缺页时刻(4) - 上次缺页时刻(1) $> \tau$ 成立，所以，在时间间隔(1,4)内未被引用的页面 P1 和 P5 应当移出；继而缺页发生在时刻 $t = 6$ ，但由于本次缺页时刻(6) - 上次缺页时刻(4) $\leq \tau$ ，所以，这次无页而被移出；在时刻 $t = 9$ 发生下次缺页时，因移出条件为“真”，故页面 P2 和 P4 被移出；在时刻 $t = 10$ 发生最后一次缺页时，无页面需要移出。缺页频率替换算法对此引用串共产生 5 次缺页，工作集页面数在 3~4 之间波动。

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串		P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	✓	✓	✓	✓	-	-	-	-	-	✓	✓
P2	-	-	-	-	✓	✓	✓	✓	-	-	-
P3	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P4	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓
P5	✓	✓	✓	✓	-	-	✓	✓	✓	✓	✓
In _t		P3			P2		P5			P1	P4
Out _t					P1, P5					P2, P4	

图 4.25 缺页频率替换示例

8. 请求分页虚拟存储管理的几个设计问题

(1) 页面大小

在虚拟空间大小一定的前提下,页面大小的变化会对页表大小产生影响,如果页面较小,虚存空间的页面数就会增加,页表也随之扩大。由于每个进程都必须有自己的页表,因此,为了控制页表所占的主存空间,页面的尺寸还是大一些为好。

从主存利用率来考虑,主存以块为分配单位,一般而言,进程的最后一页会产生内部碎片,平均将造成半个主存块的浪费。为了减少内部碎片,页面的尺寸还是小一些为好。

从读写页面所需的时间来考虑,从磁盘读入一个页面所需的时间包括等待时间(移臂时间+旋转时间)和传输时间,通常等待时间远远大于传输时间,显然,加大页面尺寸,有利于提高I/O操作的效率。

综合上述几点因素,现代操作系统中的页面尺寸取512B~8KB不等,可以从减少内部碎片和页表耗费的存储空间推导出来。一些著名机型所选择的页面尺寸:Atlas为512B,IBM 370系列机为2048B或4096B,VAX为512B,IBM AS/400为512B,Macintosh为4096B,Intel x86为4096B,MIPS R4000提供4096B~16MB共7种页面长度。页面长度是由CPU中的MMU规定的,操作系统通过特定寄存器的指示位来指定当前选用的页面长度。

(2) 页面交换区

替换算法经常要挑选淘汰页面,被淘汰的页面可能很快又要被使用,需要重新装入主存。页面从何而来?放至何处?操作系统通常把被淘汰的页面保存在磁盘的特殊区域,例如,UNIX/Linux系统使用交换区(可以是磁盘分区称为交换设备,也可以是文件称为交换文件)临时保存淘汰页面。系统初始化时,保留一定的磁盘空间作为页面交换区,初始化内容为空,不能被文件系统使用,而是作为物理主存的扩充,它和主存储器一起组成虚存空间。当一个进程被创建时,就预留出与此进程一样大小的交换空间;当进程撤销时,释放其所占用的交换空间。

当某个被保存在交换区的页面再次被使用时,操作系统就从交换区中读出这个页面,所以,需要建立和维护交换区映射表,记录所有被换出的页面在交换区中的位置。如果页面要被换出主存,仅当其内容与保存在交换区的副本不同时才对其进行复制。交换区最常用的数据结构是进程页面号与盘块号的对照表,称为磁盘外页表。当发生缺页中断时,虚拟地址中的页号被映射到外页表,从而获得对应的盘块号,由盘块号读出页面。

(3) 写时复制

写时复制(copy-on-write)是存储管理用来节省物理主存(页框)、降低进程创建开销的一种页面级优化技术,已被UNIX/Linux和Windows等许多操作系统所采用,能减少主存页面内容的复制操作,减少相同内容页面在主存中的副本数目。

在创建进程时系统并不复制父进程的完整空间,而仅复制父进程的页表,使父子进程共享物理空间,并把这个共享空间的访问权限设置为“只读”。当其中某个(父或子)进程要修改页面内容以执行写操作时,会产生“写时复制”中断,操作系统处理这个中断信号,为此进程创建一个新页,设置其为可读可写,并将其映射到进程的地址空间,此进程就可以修改复制的页,所有未修改

页仍可与其他进程共享。可见，采用写时复制技术之后，就可推迟到修改时才对共享页面做出副本，避免对“只读”页的复制，从而减少页面复制操作并节省副本所占用的空间。

4.5.3 请求分段虚拟存储管理

请求分段虚拟存储管理也为用户提供比主存实际容量大得多的虚拟主存空间。请求分段虚拟存储系统把作业的所有分段的副本都存放在辅助存储器中，当作业被调度投入运行时，首先把当前需要的段装入主存，在执行过程中访问到不在主存的段时再将其动态装入。因此，在段表中必须增设供管理使用的若干表项，如特征位、存取权限、扩充位、标志位、主存起始地址和段长等。其中，特征位指示段是否在主存中：0（不在主存），1（在主存）；存取权限给出段的可访问模式：可执行、可读、可写等；扩充位指出段可否动态扩充：0（固定长度），1（可扩充）；标志位又分为修改位、访问位、可否移动位等。

在作业执行过程中访问某段时，由硬件的地址转换机制查段表，若所需的段在主存中，则按分段存储管理方法进行地址转换以得到绝对地址；若所需的段不在主存中，触发缺段中断，操作系统处理此中断时，查找主存分配表，找出一个足够大的连续区域容纳此分段；如果找不到足够大的连续区域则检查空闲区的总和，若空闲区总和能满足分段要求，那么，进行适当移动后，将此分段装入主存；若空闲区总和不能满足要求，可调出一个或多个分段到磁盘上，再将此分段装入主存。

在执行过程中，有些表格或数据段随着输入数据的多少而变化。例如，某个分段在执行期间因表格空间用完而要求扩大分段，这只需在此分段后添置新信息即可，添加后的长度应不超过硬件所允许的每段的最大长度。对于这种变化的数据段，当向其中添加新数据时，由于欲访问的地址超出原有的段长，硬件将产生一个越界中断，系统处理这个中断时，先判别此段的“扩充位”标志，如果可以扩充，则增加段的长度，必要时还要移动或调出分段以腾出主存空间；如果此段不允许扩充，那么，这个越界中断就表示程序出错。缺段中断和段扩充处理流程如图 4.26 所示。

请求分段虚拟存储管理便于实现分段的共享和保护。为了实现分段的共享，除了原有的进程段表外，还要在系统中建立一张段共享表，每个共享分段占一个表项，每个表项包含两部分内容：第一部分含有共享段名、段长、主存起始地址、状态位（如是否在主存位）、磁盘地址、共享进程个数计数器；第二部分含有共享此段的所有进程名、状态、段号、存取控制位（通常为“只读”）。分段共享涉及段的动态链接，技术实现比较复杂，可参考 MULTICS 动态链接和共享机制。

分段存储器系统中，每个分段在逻辑上是独立的，实现存储保护时也很方便。一是越界检查，在段表寄存器中存放段长信息，在进程段表中存放每个段的段长，每当存储访问时，首先把指令逻辑地址中的段号与段表长度进行比较，如果段号等于或大于段表长度，将发出地址越界中断；其次，还需检查段内位移是否等于或大于段长，若是，则产生地址越界中断，从而确保每个进程只在自己的地址空间内运行。二是存取控制检查，在段表的每个表项中，均设有存取控制字段，用于规定此段的访问方式，通常设置的访问方式有“只读”、“可读写”、“只执行”等。

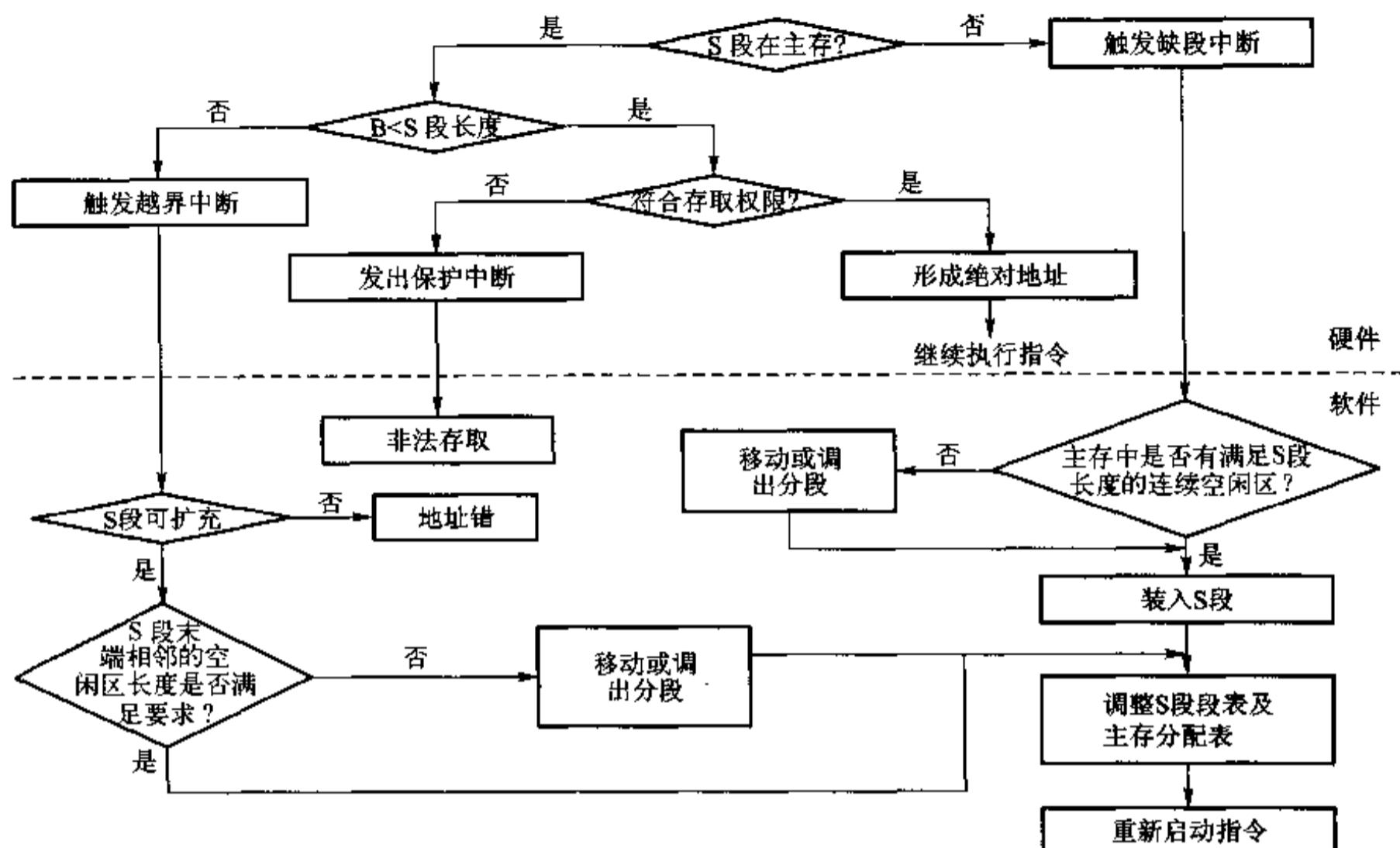


图 4.26 请求分段虚拟存储管理的地址转换和存储保护

4.5.4 请求段页式虚拟存储管理

段式存储是基于应用程序结构的存储管理技术,有利于模块化程序设计,便于段的扩充、动态链接、共享和保护,但往往会产生段之间的碎片,浪费存储空间;页式存储是基于系统存储器结构的存储管理技术,存储利用率高,便于系统管理,但不易实现存储共享、保护和动态扩充。如果把两者优点结合起来,在分段存储管理的基础上实现分页存储管理就是段页式存储管理。下面介绍请求段页式虚拟存储管理的基本原理。

- (1) 虚地址以程序的逻辑结构划分成段,这是段页式存储管理的段式特征。
- (2) 实地址划分成位置固定、大小相等的页框(块),这是段页式存储管理的页式特征。
- (3) 将每一段的线性地址空间划分成页框大小相仿的页面,于是形成段页式存储管理的特征。
- (4) 逻辑地址分为3个部分:段号s、段内页号p、页内位移d。对于用户而言,段式虚拟地址应该由段号s和段内位移d'组成,操作系统内部自动把d'解释成两部分:段内页号p和页内位移d,也就是说,d'=p×块长+d。
- (5) 请求段页式虚拟存储管理的数据结构较为复杂,包括作业表、段表和页表三级结构。作业表中登记进入系统的所有作业及此作业段表的起始地址;段表中至少包含此段是否在主存

的信息，及此段页表的起始地址；页表中包含页是否在主存的信息（中断位）、对应的主存块号。

请求段页式虚拟存储管理的动态地址转换机构由段表、页表和快表构成，当前运行作业的段表起始地址已被操作系统置入段表控制寄存器，其动态地址转换过程如下：从逻辑地址出发，先以段号 s 和页号 p 作为索引去查找快表，如果找到，立即获得页 p 的页框号 p' ，并与位移 d 一起拼装得到访问主存的实地址，从而完成动态地址转换；若查找快表失败，就要通过段表和页表进行地址转换，用段号 s 作为索引，找到相应的表目，得到 s 段页表的起始地址 s' ，再以页号 p 作为索引得到 s 段 p 页所对应的表目，由此得到页框号 p' ；这时一方面把 s 段 p 页和页框号 p' 置换进快表，另一方面用 p' 和 d 生成主存物理地址，完成地址转换。

上述过程假设所需信息都在主存中，事实上，许多情况都会发生，如查段表时，发现 s 段不在主存，于是产生“缺段中断”，引起操作系统查找 s 段在磁盘上的位置，并将段调入主存；如查页表时，发现 s 段 p 页不在主存，于是产生“缺页中断”，引起操作系统查找 s 段 p 页在磁盘上的位置，并将此页调入主存。当主存已无空闲页框时，就会导致淘汰页面。如图 4.27 所示是段页式动态地址转换和存储保护示意图。

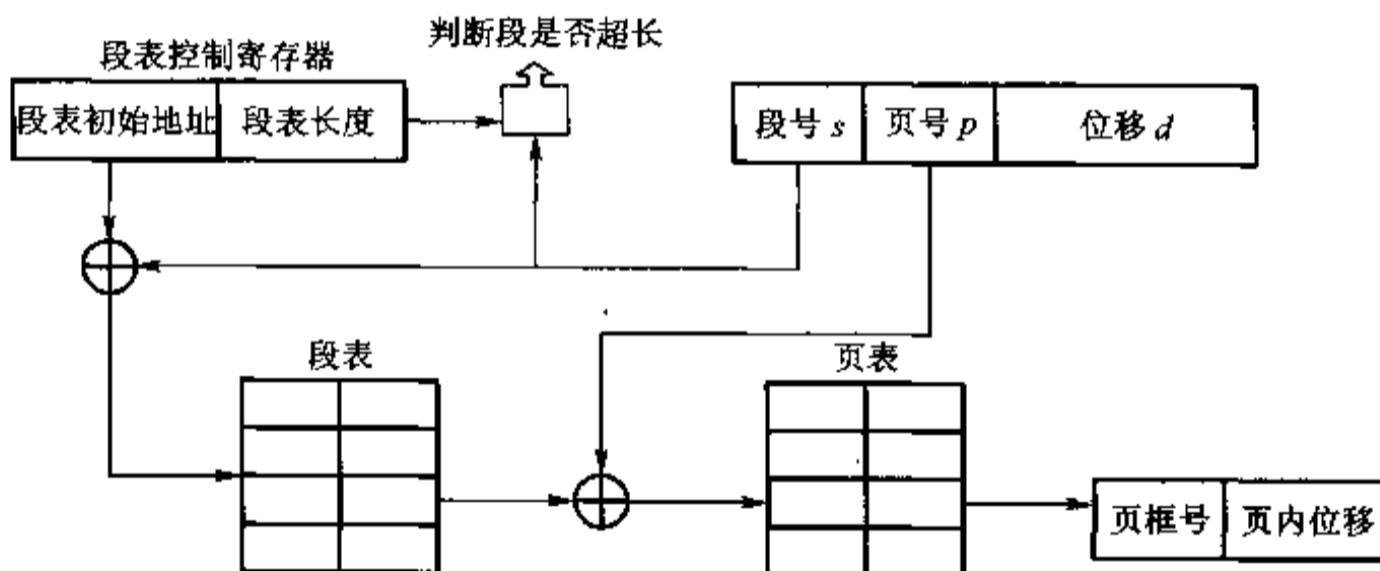


图 4.27 段页式动态地址转换和存储保护示意图

第 6 章 Intel x86 分段和分页存储结构

Intel x86 系列 CPU 提供 3 种工作模式：实地址模式、保护模式和虚拟 8086 模式。实地址模式采用段式寻址或单一连续分区，无特权级，不启用分页机制，寻址范围为 1 MB；保护模式采用分段机制并可启用分页机制，共有段式、页式、段页式这 3 种虚拟存储管理模式；虚拟 8086 模式是在保护模式下对实地址模式的仿真，允许多个 8086 应用程序同时在 386 以上 CPU 上运行。DOS 在实地址模式下工作，而 Windows 和 Linux 等操作系统在保护模式下运行。

Intel x86 实现虚拟存储管理的核心是维护主存中的两张表:局部描述符表(Local Descriptor Table,LDT)和全局描述符表(Global Descriptor Table,GDT),分别用 GDTR 和 LDTR 两个寄存器指向主存中的描述符表。每个进程都有其私有的 LDT,描述局部于它的分段,包括代码段、数据段、堆栈段、扩充段等的基地址、段大小和有关控制信息等;系统的所有进程共享一个 GDT,描述系统段,包括操作系统的基地址、段大小、相关控制信息和所有软件共享的系统资源。

为了访问一个段,必须把段选择符装入机器的 6 个段寄存器之一,如 CS 寄存器保存代码段选择符、DS 寄存器保存数据段选择符和 SS 寄存器保存堆栈段选择符等。硬件还配有 6 个 8 B 的高速缓存寄存器,存放取自 LDT/GDT 的描述字,每次访问主存时不必都到主存中读取描述字,大大地加快存取速度。控制寄存器 CR0 给出分页标志 PG、保护允许标志 PE 等;CR2 保存缺页中断时所缺页的线性地址;CR3 存放页目录基址。物理地址空间最大为 $2^{32} = 4$ GB,但虚存地址空间最大可为 64 TB,为此,定义如图 4.28 所示的虚拟地址。

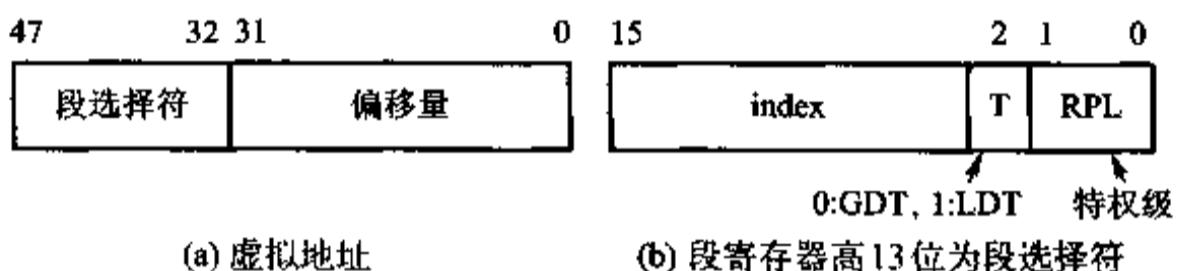


图 4.28 Intel x86 虚拟地址和段选择符

在 16 位的段寄存器中,第 0、1 位 RPL 是描述符请求的特权级,它不参与段选择;第 2 位 T 为 0 或 1 分别指明选择 GDT 或 LDT;高 13 位 index 用做访问段描述符表中某个描述符的下标。由此可知虚拟地址空间共包含 $2^{14} = 16$ K 个存储器分段,其中 GDT 映射一半(8 192 个)全局虚拟地址空间,LDT 映射另一半(8 192 个)局部虚拟地址空间。当发生进程切换时,LDT 更新为待运行进程的 LDT,而 GDT 保持不变。由于段内偏移量 32 位即 $2^{32} = 4$ GB,整个虚存地址空间为 $2^{14+32} = 64$ TB。

描述符表中的描述符是存储管理硬件 MMU 管理虚存空间分段的一种依据,一个描述符直接对应于虚存空间中的一个主存分段,定义段的基址、大小和属性。GDT 和 LDT 拥有相同格式的描述符,一个段描述符由 8 B 组成,如图 4.29 所示。

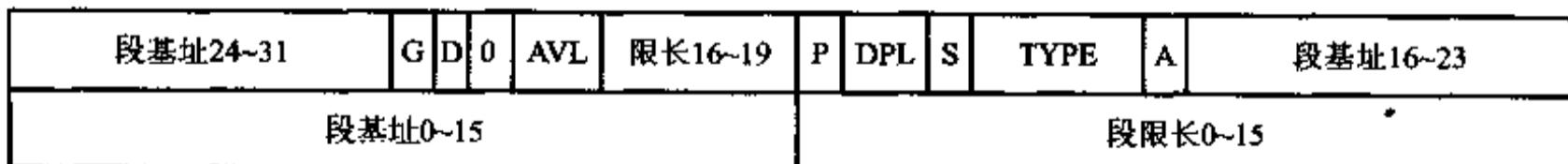


图 4.29 描述符

其中的主要内容有:段基址,共 32 位,加上 32 位偏移量形成线性地址;段限长,共 20 位,限定段描述符寻址的主存段的长度。各种特征位如下:

(1) G 位: 描述段长的单位, $G=0$ 表示以字节为单位; $G=1$ 表示以页面为单位, 页长固定为 4 KB, 于是段的长度分别为 2^{20} B 或 $2^{20} \times 4$ KB = 2^{32} B = 4 GB。

(2) D 位: 当 $D=1$ 时, 为 32 位代码段; 当 $D=0$ 时, 为 16 位代码段。

(3) P 位: 若为 1, 表示段包含有效基址和界限; 若为 0, 无定义。

(4) DPL 位: 指明描述符特权级(0~3)。

(5) S 位: 是段内容标志, S 为 1 时, 表示代码和数据段描述符; S 为 0 时, 表示系统段描述符。

(6) TYPE 字段: 表示段类型和保护方式, 如可执行代码段、只读数据段等。

(7) A 位: 是访问位, 0 表示未访问; 1 表示访问过, 为淘汰页面做准备。

(8) AVL 位: 用户编程可用位。

从虚拟地址(16 位选择符 + 32 位偏移量)到物理地址的转换要分两步走, MMU 使用分段机制把 48 位虚拟地址先转换成 32 位线性地址, 转换过程是通过描述符表中的描述符来实现的。在保护模式下, 段选择符被装入段选择符寄存器时, 从选择符的 T 位就知道是选 LDT 还是 GDT, 再根据 index 由硬件自动从 LDT 或 GDT 中取出描述符装入段描述符高速缓存寄存器中, 以实现 16 位选择符到 32 位段基址的转换, 再把描述符中的 32 位段基址与 32 位偏移量相加以形成 32 位线性地址。

如果控制寄存器 CR0 的 PE 位为 1, 处理器工作在保护模式下, 而 CR0 的控制寄存器的 PG 位为 0 时, 表明此时禁止分页, 那么, 线性地址就是访问存储器的物理地址, 这是纯分段结构。如果把 6 个段寄存器设置为同一个段选择符来实现不分段(实际仅有一段), 且段描述符基址设置为 0, 段大小设置为最大值 4 GB, 此时是单段且也有 CR0 的 PE 位为 1, 同时 PG 位为 1, 系统运行于分页模式, 32 位地址空间中单段分页运行, 实际上就是纯分页结构。

当 PG 为 1 时, 启用分页机制, 此时线性地址并不是最终访问的物理地址, 需要通过分页机制进行第二次地址转换。由分段得到的线性地址又分成 3 个域: 10 位页目录 dir、10 位页 page 和 12 位偏移量 offset。在进行页转换时, 根据控制寄存器 CR3 给出的页目录表的起始地址, 用 dir 作为索引在页目录表中找到指向页表的起始地址, 再用 page 作为索引在页表中找到页框起始地址, 最后, 把偏移量加到页框起始地址上, 得到访问单元的物理地址。页目录表和页表的结构是类似的, 均为 32 位, 其中左边 20 位是页表或页框的基地址, 保证页表或页框对齐 4 KB 边界。页表表项的右 12 位包括由硬件设置的供操作系统使用的中断位、引用位、修改位、保护位和其他一些有用的位。当从页表中读出页表项时, 就被硬件同时装入快表, 以便继续引用时无须再次查找页表。Intel x86 存储管理机制的设计非常巧妙, 实现了避免互相冲突的目标: 段页式存储管理、纯页式存储管理、纯段式存储管理, 并能够同实地址模式兼容。图 4.30 给出最复杂的保护模式下段页式虚拟地址到物理地址转换的过程。

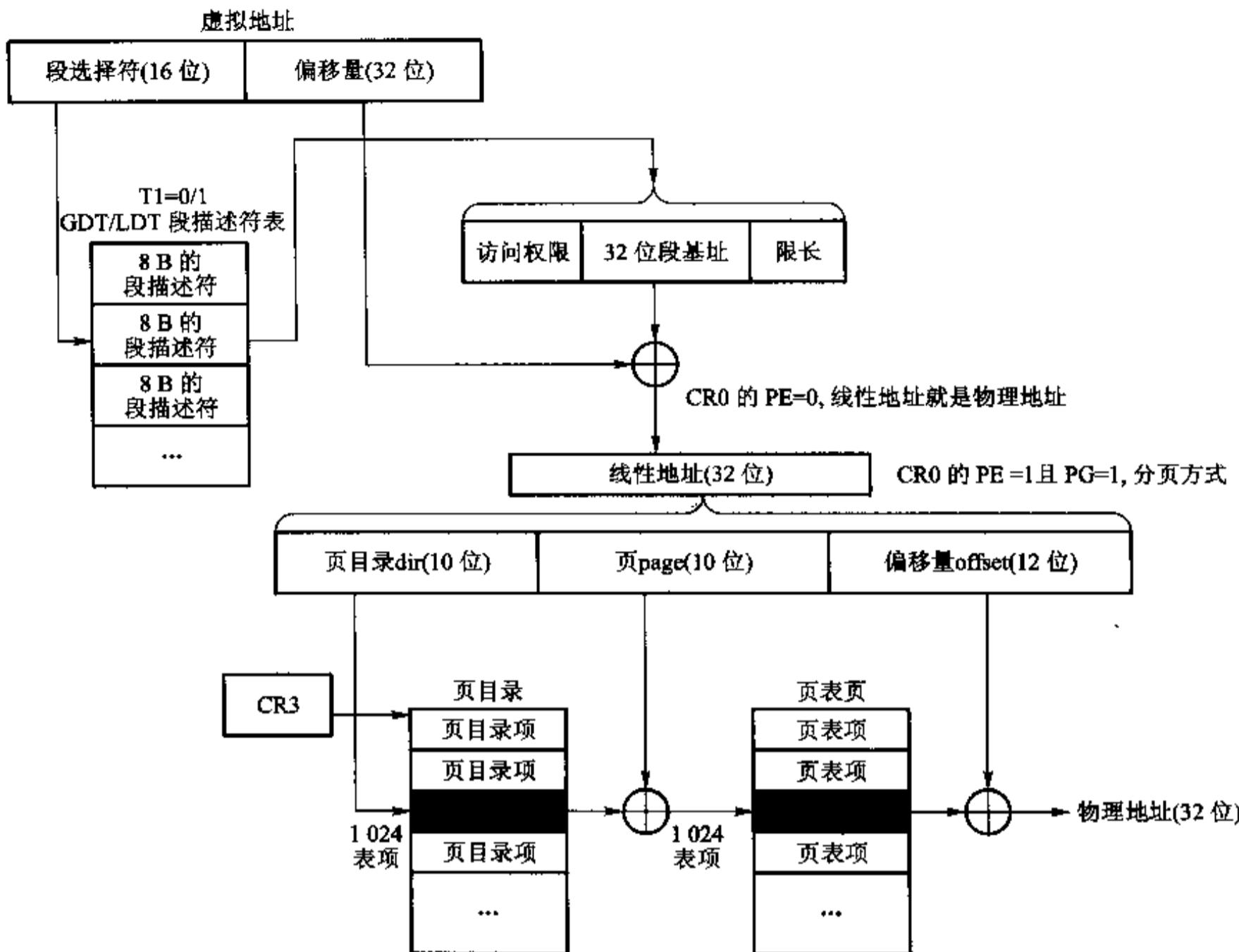


图 4.30 段页式地址转换过程

4.7 Linux 虚拟存储管理

4.7.1 Linux 虚拟存储管理概述

在 Linux 系统中, 进程可以访问 4 GB 虚拟地址空间, 其中, 0~3 GB 被用户进程独占且可直接访问; 3 GB~4 GB 是内核空间, 由所有核心态进程共享, 存放系统代码和数据。每个进程都有一个页目录, 大小为一页, 页目录的起始地址存放于进程 `mm_struct` 结构中, 工作时被装入寄存器 `CR3`。页目录项为 4 B, 共有 1 024 项, 用来保存页表的起始地址。每张页表用一个页面存储, 每项为 4 B, 共有 1 024 项, 用来保存页框基址。在 Intel x86 上, 页面大小为 4 KB; 在 Alpha AXP 上, 页面大小为 8 KB。页表项的格式如下:


```

    unsigned long map_nr;           /* 页框在 mem_map 表中的下标 */
    struct page ** pprev_hash;     /* page cache 的哈希表中的前驱指针 */
    struct buffer_head * buffers;  /* 若此页框用做缓冲区, 指示缓冲区地址 */
    struct inode * inode;          /* 页框主存放代码或数据所属文件的 inode */
    unsigned long offset;          /* 页框主存放代码或数据所属文件的位移 */
    struct zone_struct zone;      /* 页框所在管理区 */

} mem_map_t;

```

其中, flags 共 32 位, 描述页框状态, 如页不能换出、调页时发生 I/O 操作错误、完成读操作、从高速缓存或磁盘换入页、DMA 传输、内核专用等。

(2) 管理区管理

主存通常被划分成 3 个区: ZONE_DMA 区, 专供 DMA 使用; ZONE_NORMAL 区, 被常规使用; ZONE_HIGHMEM 区, 内核不能直接映射区。设置 ZONE_DMA 是保证磁盘 I/O 操作所需的连续物理页框, ZONE_NORMAL 中的页框用做通常的主存空间分配。

每个管理区都有数据结构 zone_struct, 其中含有一组空闲区队列: 空闲区的连续长度为 1 的页框, 连续长度为 2 的页框, …, 直至连续长度为 $2^{\text{MAX_ORDER}}$ 的页框。常量 MAX_ORDER 的值定义为 10, 即最大的连续物理块可达 1 024 个页框(4 MB), 数据结构描述如下。

```

typedef struct free_area_struct {           /* 空闲区队列头部结构 */
    struct list_head free_list;               /* 指向空闲区队列 */
    unsigned int * map;                      /* 指向 bitmap 表 */
} free_area_t;

```

除了 mem_map 表之外, 系统还设立 bitmap 表, 以位示图来记录物理主存的使用状况。管理区的数据结构由 zone_struct 描述:

```

typedef struct zone_struct {
    spinlock_t lock;                         /* 自旋锁, 保证对 zone 的互斥访问 */
    unsigned long offset;                     /* offset 表示分区在 mem_map 中的起始页框号, 一旦
                                                建立管理区, 每个页框就永久属于某个管理区 */
    unsigned long free_pages;                /* 此区空闲页框数 */
    unsigned long pages_min, pages_low, pages_high; /* 此区最少、次少和最多页框数描述 */
    free_area_t free_area[MAX_ORDER];        /* 伙伴系统中的空闲页框链表数组 */
    struct pglist_data * zone_pgdat;         /* 此区所在存储节点 pglist_data */
    struct page * zone_mem_map;              /* 此区主存映射表 */
    unsigned long zone_start_paddr;          /* 此区起始物理地址 */
    unsigned long zone_start_mapnr;          /* 在 mem_map 中的下标 */
    unsigned long size;                      /* 管理区物理主存的大小 */
    char * name;                           /* 管理区的名字 */
} zone_t;

```

```

| zone_t;
(3) 存储节点管理
typedef struct pglist_data| /* 存储节点的结构 */
{
    zone_t node_zones[MAX_NR_ZONES]; /* 存储节点的管理区数组 */
    zonelist_t node_zonelists[NR_GFPINDEX];
    struct page * node_mem_map; /* 存储节点的主存映射表 */
    int nr_zones; /* 存储节点的管理区数目 */
    unsigned long * valid_addr_bitmap; /* 位示图表示的有效地址 */
    struct bootmem_data * bdata; /* 存放位图的数据结构 */
    unsigned long node_start_paddr; /* 存储节点起始物理地址 */
    unsigned long node_start_mapnr; /* 在 mem_map 中的下标 */
    unsigned long node_size; /* 存储节点物理主存的大小 */
    int node_id; /* 存储节点标识符 */
    struct pglist_data * node_next; /* 下一存储节点指针 */
} pg_data_t;
typedef struct zonelist_struct {
    zone_t * zones[MAX_NR_ZONES+1];
    int gfp_mask;
} zonelist_t;
#define NR_GFPINDEX 0x100 /* 规定管理区分配策略数 */

```

这里的 zones[] 是指针型数组,各元素按特定次序指向具体的页框管理区,表示分配页框时先尝试 zones[0] 指向的管理区,如不能满足要求就尝试 zones[1] 指向的管理区,等等。这些管理区可以属于不同的存储节点。由于每个 zonelist_t 规定一种分配策略,而每个存储节点可以有多种分配策略,所以在 pglist_data 结构中提供一个 zonelist_t 结构数组,数组大小为 NR_GFPINDEX,表示最多可规定 0x100 种不同的策略。

2. 虚拟主存空间管理

虚拟主存空间管理以进程为基础,进程都拥有私有虚存空间,进程的内核空间被所有进程共享,虚拟主存空间主要由 mm_struct 结构和 vm_area_struct 结构描述。

(1) 虚存区 vm_area_struct

用户进程最多可以使用 3 GB 地址空间,事实上几乎没有进程真正用到如此大规模的空间,因此,有必要显式地表示真正被进程用到的那部分虚存区(也称为段)。另外,进程所用虚存中的各区域未必连续但绝不重叠,由于采用面向对象方法管理,使得虚存区结构体可以代表多种类型的主存区,如代码段、数据段、堆栈段、共享段和主存映射文件等,形成若干离散的虚存区域,因此,对虚存区的抽象是一个重要的问题。内核将进程的每个虚存区作为一个单独的主存对象管理,每个虚存区都拥有一致的属性,比如访问权限等。采用虚存区 vma(virtual memory area) 来描

述进程虚拟主存的一个区域,而 vma 链表用来表示此进程实际用到的虚拟地址空间。vm_area_struct 数据结构描述如下。

```
struct vm_area_struct {
    struct mm_struct * vm_mm;           /* 虚存区间的结构 */
    unsigned long vm_start;             /* 虚存区间所在的 mm_struct 指针 */
    unsigned long vm_end;               /* vma 的起始地址 */
    pgprot_t vm_page_prot;             /* vma 的结束地址 */
    unsigned short vm_flags;            /* 此 vma 存取权限 */
    struct vm_area_struct * vm_next;    /* 此 vma 操作标志 */
    struct vm_area_struct * vm_next;    /* 按地址降序排列的链接下一个 vma 的指针 */
    unsigned long vm_avl_height;        /* vma 的 AVL 树的高度 */
    struct vm_area_struct * vm_avl_left;  /* vma 的 AVL 树的左节点 */
    struct vm_area_struct * vm_avl_right; /* vma 的 AVL 树的右节点 */
    struct vm_operations_struct * vm_ops; /* vma 上所定义的操作集 */
    struct vm_area_struct * vm_next_share; /* 共享同一文件的下一个 vma */
    struct vm_area_struct ** vm_pprev_share; /* 共享同一文件的上一个 vma */
    unsigned long vm_pte;               /* pte 表 */
    unsigned long vm_pgoff;             /* vm_file 中的偏移量 */
    struct file * vm_file;              /* 此 vma 被映射的文件(若有) */
    unsigned long vm_raend;             /* 存放预读信息 */
    void * vm_private_data;            /* 共享主存 */
    ...
} vma;
```

vm_flags 给出进程虚拟地址空间中此 vma 页面的访问权限和页面的相关信息,页面访问权限主要有:“只读”页、“可写”页、“可执行”页、“可共享”页、vma 页面被锁住、vma 可作为共享主存、vma 可向下增长、vma 可向上增长、vma 映射不可写文件、vma 映射可执行文件、vma 不能换出、vma 是 I/O 设备的地址空间、页被连续访问、页被随机访问等。vm_next 指针把所有 vma 连成一个线性队列。vma 也可以记录区域中是否有指定的后援存储及其位置,正文段使用可执行二进制文件作为后援存储,主存映射文件使用磁盘文件作为后援存储。当 vma 数量较少时,用单链表管理,当 vma 节点超过一定的数量时,为其建立 AVL(Adelson_Velskii and Landis)平衡树,加快 vma 检索效率;vm_avl_height、vm_avl_left 和 vm_avl_right 用于 AVL 树,表示本 vma 在 AVL 树中的位置。对 vma 的操作由以下结构体定义。

```
struct vm_operations_struct {
    void( * open)(struct vm_area_struct * area);      /* 打开映射的 vma */
    void( * close)(struct vm_area_struct * area);     /* 关闭映射的 vma */
    void( * unmap)(struct vm_area_struct * area,      /* 解除映射并释放 vma */
                   unsigned long start,
                   unsigned long end);
}
```

```

        unsigned long, size_t);
void (*protect) (struct vm_area_struct *area, /* 设置和检查 vma 的保护权限 */
                 unsigned long, size_t, unsigned int newprot);
struct page * (*nopage)(struct vm_area_struct *area, /* 处理缺页异常 */
                       unsigned long address, int write_access);
int (*swapout) (struct vm_area_struct *, struct page *); /* 换出 vma */
pte_t (*swapin) (struct vm_area_struct *, unsigned long, /* 换入 vma */
                  unsigned long);
};


```

此结构中全是函数指针,这些函数分别用于 vma 的操作,如打开、关闭、调页、建立映射和解除映射。

(2) 主存描述符 mm_struct

vm_area_struct 中的指针 vm_mm 指向主存描述符 mm_struct 结构,主存描述符 mm_struct 结构中包含用户进程虚拟地址空间的全部信息,其定义如下。

```

struct mm_struct {
    struct vm_area_struct *mmap;           /* 指向 vma 的链表 */
    struct vm_area_struct *mmap_avl;        /* 指向 vma 的 AVL 树 */
    struct vm_area_struct *mmap_cache;      /* 最后使用的 vma, 放入 cache 中 */
    pgd_t *pgd;                          /* 进程页目录表基址 */
    int map_count;                      /* vma 个数, 最多 65536 个 */
    struct semaphore mmap_sem;           /* 对 mmap 操作的互斥信号量 */
    spinlock_t page_table_lock;          /* 页表访问的自旋锁 */
    struct list_head mmlist;             /* 全部 mm_struct 链表 */
    atomic_t mm_users;                  /* 用户空间中的用户数 */
    atomic_t mm_count;                  /* 对 mm_struct 的引用计数 */
    unsigned long start_code, end_code;   /* 进程代码段的起始地址和结束地址 */
    unsigned long start_data, end_data;   /* 进程数据段的起始地址和结束地址 */
    unsigned long start_brk, brk, start_stack; /* 进程堆的首尾地址及进程栈的起始地址 */
    unsigned long arg_start, arg_end, env_start, env_end; /* 命令行参数和环境有关的信息 */
    unsigned long rss, total_vm, locked_vm; /* 进程驻留主存的页面总数, 进程所需
                                             页面总数, 锁在主存的页面数 */
    unsigned long def_flags, cpu_vm_mask, swap_address; /* 默认访问标志、lazy 快表
                                                       交换掩码、对换地址 */
    unsigned long swap_cnt;              /* 下一次交换页面数 */
    mm_context_t context;               /* 体系结构特殊数据 */
};


```

```

    struct kioctx * iocctx_list;           /* I/O 链表 */
    struct kioctx * default_kioctx;        /* 默认 I/O 上下文 */
};


```

每个进程都有一个 mm_struct 结构,在进程的 task_struct 结构中有一个指针 mm 指向此进程的 mm_struct 结构,mm_struct 结构是进程整个虚拟地址空间的抽象。mm_struct 结构中的前 3 个虚存区指针:mmap 用来建立一个虚存区间结构的链接队列;mmap_avl 用来建立一个虚存区结构的 AVL 树;mmap_cache 用来指向最近一次用到的那个虚存区结构,因为程序具有局部性,很可能这就是下次要用到的区间,以便提高执行效率。指针 pgd 指向此进程的页目录表,当进程被调度时,此指针被转换成物理地址,写入控制寄存器 CR3。

进程只有一个 mm_struct,而一个 mm_struct 却可能被多个进程所共享,当运行进程创建子进程时,子进程就可能与父进程共享同一个 mm_struct 结构,所以在 mm_struct 中设置计数器 mm_users 和 mm_count。

下面讨论主存描述符的分配和撤销,进程描述符的 mm 域存放进程所使用的主存描述符,故 current->mm 便指向当前进程的主存描述符。fork() 函数利用 copy_mm() 函数复制父进程的主存描述符,即把 current->mm 域交给其子进程,而子进程中的 mm_struct 结构体则是通过内核的 allocate_mm() 来建立,每个进程都有唯一的 mm_struct 结构体,即唯一的进程地址空间。如果父进程希望和子进程共享地址空间,可在调用 clone() 时设置 CLONE_VM 标志,这样的子进程称作线程。当 CLONE_VM 被指定后,内核就无须调用 allocate_mm() 函数,而仅需要在调用 copy_mm() 函数时将 mm 域指向其父进程的主存描述符。

进程退出时,内核会调用 exit_mm() 函数执行常规的撤销工作,同时更新一些统计量,如 exit_mm() 函数会调用 mmput() 函数减少主存描述符中的 mm_users 用户计数,如果用户计数降至 0,继续调用 mmdrop() 函数,减少 mm_count 计数;如果 mm_count 计数也等于 0,说明此主存描述符不再有使用者,调用 free_mm() 宏通过 kmem_cache_free() 函数回收 mm_struct 结构体。

如图 4.31 所示是进程虚存管理数据结构。从进程控制块的内嵌 mm_struct 可找到主存管理数据结构,从主存管理数据结构指向 vma 区的链接指针 mmap 可找到按照升序用 vm_next 链接起来的进程的所有 vma。此外,每个进程有一个页目录 pgd,存储此进程所使用的主存页面情况,根据缺页调度原则,只分配所用到的主存页而,从而避免页表占用过多的物理主存空间。

4.7.3 主存页框调度

主存页框调度有两项工作:一是页框的分配、使用和回收;二是盘交换区管理,交换的目的也是页的回收,并非所有的主存页均可交换出去,只有映射到用户空间的页才会被换出。

页框分配时,为了提高向主存读入和从主存写出页的效率,采用伙伴系统把连续的页映射到连续的页框中。为了方便主存页框的管理和分配,内核为每个主存页框都建立一个 page 数据结构;与此类似,交换设备(磁盘)的每个物理页也要在主存中有相应的数据结构,用以表示此页是否已被分配,及有多少用户在共享此页面。内核中定义 swap_info_struct 数据结构来描述和

管理用于页面交换的设备。

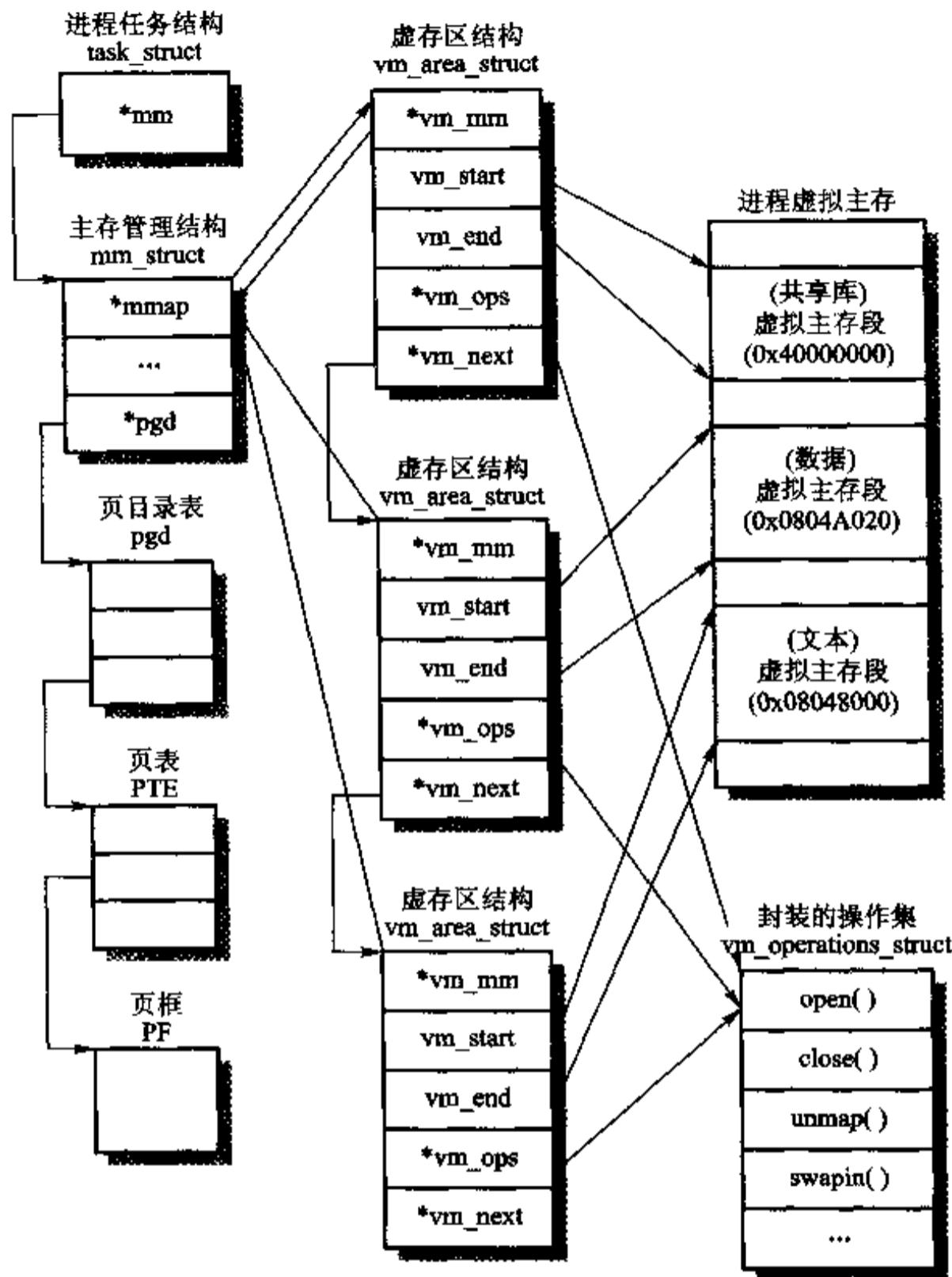


图 4.31 进程虚拟管理数据结构

```
struct swap_info_struct {
    unsigned int flags;           /* 交换区标志 */
    kdev_t swap_device;          /* 交换设备的主、次设备号 */
    spinlock_t sdev_lock;         /* 互斥自旋锁 */
    struct dentry * swap_file;   /* 指向文件或目录设备的目录项 */
    struct vfsmount * swap_vfsmnt;
    unsigned short swap_map;     /* 指向计数数组的指针, 交换区每个页面对应一个元素 */
}
```

```

    unsigned int lowest_bit; /* 交换设备中第一个可用页面位置 */
    unsigned int highest_bit; /* 交换设备中最后一个可用页面位置 */
    unsigned int cluster_next; /* 搜索空闲页面时要扫描的下一个页面 */
    unsigned int cluster_nr; /* 重新开始时分配空闲页面的个数 */
    int prio; /* 交换区优先级,按优先级排序 */
    int pages; /* 设备上的可用页面数 */
    unsigned long max; /* 交换区的大小,以页为单位 */
    int next; /* 指向下一个交换区的指针 */
};

swap_map 指向一个数组,此数组中的每个无符号短整数代表磁盘上的一个物理页面,而数组下标则决定此页在磁盘上的位置,数组的大小取决于 pages 值,它表示此页交换设备的大小。设备上的第一页 swap_map[0]不用于页交换,它包含设备自身的信息,及表明哪些页可供使用的位示图,这些信息最初是在把此设备格式化成页交换区时设置的。根据不同的页交换区格式,还有其他一些页也不供页交换使用,这些页都集中在开头和结尾处,所以 swap_info_struct 结构中的 lowest_bit 和 highest_bit 就说明设备中从何处起止是供页交换使用的。max 表示此设备中的最大页号。

```

由于存储介质是磁盘,将地址连续的页存储在连续的磁盘扇区中未必是最有效的方法,所以在分配磁盘上的页空间时尽可能地按簇进行,字段 cluster_next 和 cluster_nr 就是为此目的而设置的。内核允许使用多个页面交换设备,在内核中建立一个 swap_info_struct 的数组 swap_info。

4.7.4 进程虚存空间映射

1. mmap()

进程的虚拟地址空间内包括一组主存对象,实际上这些对象代表虚拟存储(如磁盘交换区、文件系统中的一个文件)与进程地址空间的映射,把对文件的访问转化为对虚存区的访问,这样的一个映射称为一个 vma 区,用数据结构 vm_area_struct 来表示。换句话说,在虚拟存储系统中,一个 vma 区就是一个主存对象(映像对象),它是对应于一个文件、共享主存、交换设备或其他特殊对象而建立的一块连续虚拟地址区域。在 Linux 内核中,创建并初始化一个 vma 是由称为主存映射的系统调用 mmap()来完成的。

mmap()为当前进程在主存生成一个 vm_area_struct 结构体的虚存区并连接到相应的链表上,如果此 vma 区所填充的是指令代码并标记为可执行,系统就跳转到代码段的首地址处开始执行,因为只有代码的小部分被调入主存,在 MMU 进行虚实地址转换时会产生缺页异常,系统通过异常调用执行 do_page_fault() 函数来调入可执行文件的剩余页面。mmap()通过内核函数 do_mmap() 实现其功能,它为进程创建一个新的 vma。传递给 do_mmap() 的参数有 6 个,参数 file 是指向虚拟映射的文件;参数 addr 指定从何处开始查找一个空闲区域;参数 len 给出 vma 区的地址空间长度;参数 offset 是 vma 区相对于文件 file 的起始地址的偏移量;参数 prot 是 VAM

段所包含页的访问权限(只读、可写、可执行或禁止访问);参数 flags 描述 vma 区的属性,如段中的页是私有的或可共享的。映射时,如果 file 和 offset 参数非空,称为文件映射;否则称为匿名映射。

2. munmap()

系统调用 munmap() 从当前进程的地址空间中删除一个虚存区,此函数的参数是区域起始地址 addr 和区域的长度 len,要释放的区间并非总对应于一个线性区,它既可以是虚存区的一部分,也可以删除两个或多个虚存区。munmap() 函数的执行可分为两步,首先,从进程所拥有的虚存区中删除与指定段地址区相重叠的所有区;其次,更新进程的页表,并重新调整虚存区链表。

4.7.5 缺页异常处理

Linux 系统的缺页异常处理程序 do_page_fault() 能够区分两种情况:由编程错误而引起的对保护页面非法访问的异常,这时应根据 vm_area_struct 结构体中的 vm_flags 域来区别情况,分门别类地进行处理;由进程引用尚未分配的页框而引起的异常,CPU 将产生缺页中断,并把缺页的线性地址(保存在控制寄存器 CR2 中)和缺页时访问虚存的模式一并传给缺页中断处理程序,其步骤如下:

- (1) 读取引起缺页的线性地址。
- (2) 检查异常发生时 CPU 是否正在处理中断,或者在执行内核线程,如是则进行出错处理。
- (3) 调用 find_vma 找到发生页面错误的虚拟地址所在的 vm_area_struct 结构,以确定此错误的线性地址是否包含在进程地址空间中,或在堆栈的合理扩展区中。
- (4) 若异常是由读或执行访问所引起的,则函数检查此页是否已经在 RAM 中,若不在 RAM 中且线性地址区的访问权限与引起异常的访问类型相匹配,则执行“请求调页”处理。
- (5) 检查进程页表项中的位 P,区分缺页所对应的页面是在交换空间($P=0$ 且页表项非空)还是磁盘中某执行文件的映像中。最后,进行页面调入操作。

Linux 系统的页面替换基于 Clock 算法,使用位被一个 8 位的 age 变量所取代,每当一页被访问时,age 变量值增 1;在后台,内核周期性地扫描全局页池,且当它在主存中的所有页间循环时,对每一页的 age 变量值减 1;age 值为 0 的页是一个“旧”页,有一段时间未被访问过,因而是可用于替换的最佳候选页;age 值越大,此页最近被使用过的频率就越高,也就越不适合于替换。因此,Linux 页面替换算法是一种最少使用频率策略。

当系统中的物理主存空间减少时,主存管理子系统必须释放物理页面,这个任务由内核线程 kswapd 来完成,它是一个特殊的守护线程,没有虚拟主存,在内核空间中以核心态模式运行,其目标是保证系统中有足够的空闲页框来满足应用的需求和系统的运行效率。kswapd 由内核的 init 进程在系统启动时创建,被内核的对换定时器周期性地调用,它将检查系统中的空闲页框数是否太少,用两个变量 free_pages_high 和 free_pages_low 来判断是否释放一些页面;如果系统中的空闲页框数大于上限,kswapd 就不做任何工作,并睡眠到下一次定时器到时为止;在检查过程中,kswapd 将当前被写入交换文件中的页面数也计算在内,它使用 nr_async_pages 来记录这个数

值,当有页面准备写入交换文件队列中时, `nr_async_pages` 将递增一次,同时在写入操作完成后递减一次。如果系统中的空闲页框数在上限甚至下限以下时,将通过 3 条途径来减少系统中所使用的物理页框的数量。

- (1) 换出页面缓存和数据缓存的页面。
- (2) 换出 System V IPC 的共享主存页面。
- (3) 换出或者丢弃进程所占有的页面。

如果系统中的空闲页面数低于下限, `kswapd` 将在下次运行之前释放 6 个页面,否则它只释放 3 个页面。将依次使用上述途径直到系统释放足够的空闲页框,然后对换线程再次睡眠到下次定时器到时为止。如果导致释放页面的原因是系统中的空闲页框数小于下限,则它只睡眠平时的一半时间,一旦空闲页面数大于 `free_pages_low`,那么它的睡眠时间又会延长。

Windows 2003 虚拟存储管理

4.8.1 主存管理的功能和地址空间布局

1. 主存管理的功能

主存管理器是 Windows 执行体中的一个组件,是基本的存储管理系统,实现虚拟主存,为每个进程提供一个受保护的、大且专用的地址空间,它由以下几个部分组成:

(1) 存储管理系统服务程序:用于虚拟主存的分配、回收和管理,大多以 Win32 API 和核心态设备驱动程序接口的形式提供。

(2) 转换无效和访问错误的陷阱处理程序:用于解决硬件所检测到的与主存管理有关的异常,并将虚页面装入主存。

(3) 一组系统线程:在核心态上下文中执行。

① 工作集管理器:优先级 16,每秒钟运行一次,被系统线程平衡集管理器调用。当空闲主存区低于某界限时,便启动主存管理策略,如工作集修整和已修改页面的回写等。

② 进程/堆栈交换程序:优先级 23,完成进程和内核线程堆栈的对换操作,当需要换入/换出时,平衡集管理器和内核线程调度代码将唤醒此线程。

③ 已修改页面写入器:优先级 17,将修改链表上的“脏”页写回页文件。当要减小修改链表的大小时,此线程被唤醒。

④ 映射页面写入器:优先级 17,将映射文件的“脏”页写回磁盘。当要减小修改链表的大小,或映射文件中的某些页面在修改链表中超过 5 分钟时,此线程被唤醒。

⑤ 废弃段线程:优先级 18,当没有任何一个区域对象、映射窗口指向某个段,且此段也没有原型页表项处于过渡状态时,废弃段线程注销这个段。

⑥ 零页线程:优先级 0,将空闲链表中的页面清 0,以便满足零页需求。

2. 地址空间布局

在 32 位地址空间中, 系统允许用户进程占有 2 GB 私有地址空间, 操作系统占有剩下的 2 GB 空间。Windows 系统提供引导选项, 允许用户拥有 3 GB 虚地址空间, 而仅留给系统 1 GB 空间, 以改善大型应用程序的运行性能。

如图 4.32 所示是进程虚拟地址空间布局。在用户空间中, 0~0xFFFF 的约 64 KB 存放查错帮助信息, 拒绝进程访问; 0x10000~0x7FFEFFFF 是进程的独立地址空间; 0x7FFDE000~0x7FFDEFFF 存放第一个线程的环境块; 0x7FFDF000~0x7FFDFFFF 存放进程环境块; 0x7FFE0000~0x7FFE0FFF 存放共享用户数据页, 包括系统时间、时钟计数和版本号等信息, 以便用户直接从用户态读取相关数据; 0x7FFE1000~0x7FFFFFFF-1 是拒绝访问区, 防止线程跨过用户/系统边界。在系统空间中, 80000000~A0000000 存放系统代码, 包括 Ntoskrnl.exe、HAL、引导程序、初始的未分页缓冲池等; A0000000~A4000000-1 存放系统映射视图(如 Win32.sys、GDI 驱动程序)、用户会话空间; A4000000~C0000000-1 是附加的系统页表区; C0000000~C0400000-1, 存放进程页表和页目录; C0400000~C1000000-1 映射进程工作集超空间、进程工作集列表、系统工作集列表; C1000000~E1000000-1 是系统高速缓存区; E1000000~EB000000-1 是分页缓冲池; EB000000~FFBE0000-1 是系统页表项、非分页缓冲池(系统主存堆), 系统 PTE 缓冲池用来映射系统页, 如 I/O 空间、核心栈和 VAD; FFBE0000~FFC00000-1 存放故障转储信息; FFC00000~FFFFFFFFFF 是 HAL 所保留的系统主存。

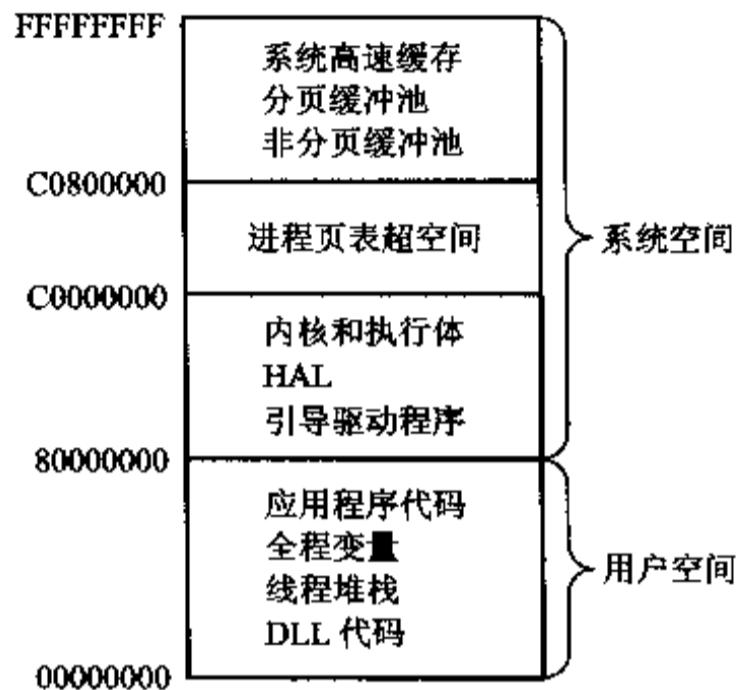


图 4.32 虚拟地址空间布局

4.8.2 进程主存空间分配

Windows 管理应用程序的主存空间, 使用两个数据结构: 虚拟地址描述符和区域对象; 使用三种应用程序主存管理方法: 虚页主存分配——管理大型对象数据或动态结构数组; 主存堆分配——管理大量的小型主存申请; 主存映射文件——管理大型数据流文件及多个进程之间的数据共享。

1. 虚拟地址描述符

主存管理采用请求分页调度算法, 直到线程访问地址并引起缺页中断时, 才为进程构筑页表项, 并调入相应的页面, 进程虚拟地址空间可达 4 GB, 这意味着进程的虚拟地址不是连续的, 系统维护数据结构“虚拟地址描述符”(Virtual Address Descriptor, VAD)来描述已经在进程中被保留的虚拟地址。

当进程保留地址空间或映射一个主存区域时, 主存管理器创建一个 VAD 来保存分配请求

所提供的信息,例如,保留地址范围(起始地址和结束地址)、此范围是共享的还是私有的、子进程能否继承此地址范围的内容以及此地址范围内应用于页面的保护限制等。对于每个进程,主存管理器都维护一组 VAD,用来描述进程虚拟地址空间的状态,为了加快虚拟地址查找速度,VAD 被构造成记录虚拟地址分布范围的一棵平衡二叉树。

当线程首次访问一个地址时,主存管理器必须为包含此地址的页面创建页表项,为此,先找到包含被访问地址的 VAD,并利用所得信息填充页表项。如果此地址落在 VAD 覆盖的地址范围之外,或所在的地址范围仅被保留而未提交,主存管理器就会知道这个线程在试图使用主存之前并未分配的主存区,因此,将产生一次访问违规。

2. 区域对象

区域对象(section object)在 Win32 子系统中被称为文件映射对象,表示可被两个或多个进程所共享的主存块。区域对象也可被映射至页文件或另一个磁盘文件。其主要作用如下:

(1) 系统利用区域对象将可执行映像和动态链接库装入主存;

对象类型	区域
对象属性	最大尺寸 页保护限制 页文件/映射文件 基准的/非基准的
对象服务	创建区域 打开区域 扩展区域 映射/取消映射视窗 查询区域

图 4.33 区域对象

(2) 高速缓存管理器利用区域对象访问高速缓存文件中的数据;

(3) 使用区域对象将文件映射至进程虚拟地址空间,然后,可以像访问主存中的数组一样访问此文件。

如图 4.33 所示是区域对象的结构,其中,最大尺寸是指区域对象可增长至的最大字节数,若映射文件,就是文件的大小,最大可达 2^{64} B;页保护限制是指分配给此区域的所有页框的保护方式;页文件/映射文件是指区域是否被创建为空(基于页文件),或是用于加载一个文件(基于映射文件);基准的:要求共享此区域的所有进程在相同的虚拟地址空间出现(如共享代码或实用程序);非基准的:可出现在不同进程的不同虚拟地址空间。

页文件是指作为主存补充的那部分磁盘空间,若主存容量为 128 MB,如果磁盘上有 128 MB 的页文件,则应用程序就认为计算机拥有 256 MB 的虚拟主存。Windows 最多支持 16 个页文件,系统启动时打开页文件,它在系统运行期间不能被删除。

3. 应用程序主存管理方法

(1) 虚页主存分配

进程私有地址空间的页面可能是空闲的(未被使用过)、被保留的(已预留虚存,尚未分配物理主存)或被提交的(已分配物理主存或交换区)。应用程序使用主存分 3 个阶段进行:保留主存(reserved memory)、提交主存(committed memory)和释放主存(release memory)。

① 保留主存

用户使用 Win32 的 virtualAlloc 可申请保留一段连续的虚拟地址空间,且用虚拟地址描述符 VAD 加以记录,使本进程的其他线程不能使用这段虚拟地址,其目的是让用户表明将要使用多

大的主存区,以便系统节省主存空间和磁盘空间。试图访问已保留的主存会造成页面无效错误,因为这时主存页面尚未映射到一个可满足这次访问的物理页框上。保留主存操作不占用和消耗物理页框和进程页文件配额,在申请保留主存时,应用进程可指定起始虚拟地址(lpAddress),或系统随便保留一段地址空间,分配单位通常为 64 KB;此时,还需提供保留地址空间的大小(cbSize),保留或提交(fdwAllocationType),保护限制信息(fdwProtect),指定页面不可访问、只读或可读写等。

② 提交主存

提交主存是在已保留的虚拟地址区域中请求分配物理主存,同时系统在它的分页文件中为这类页留出适当的位置,当它们被移出主存时,就可写入磁盘分页文件。所提交的页面在访问时会映射为物理主存中的有效页框。提交主存仍然使用 Win32 函数 VirtualAlloc,与保留主存的区别在于参数设置不同。

区分保留空间和提交空间是很有用的,能够减少为特定进程留出的磁盘空间总量,同时允许进程在需要时迅速获得物理主存空间。

③ 释放主存

当进程不再需要被提交的主存或保留的地址空间时,使用释放函数 Win32 VirtualFree 来回收盘交换区空间或从进程的虚拟地址空间中释放虚拟地址,所需要的参数有:释放起始地址(lpAddress),释放长度(cbSize),是否仅释放物理存储(fdwFreeType)。VirtualFree 函数对于减少和控制盘交换区和进程虚拟地址空间的占用很有用。

(2) 主存堆分配

堆(heap)是保留地址空间中一个或多个页所组成的区域,由堆管理器按照更小块划分和分配主存。堆管理器用于分配和回收可变主存空间,不必像虚页分配一样按页对齐。Win32 API 可调用 Ntdll 中的函数,执行组件和设备驱动程序可调用 Ntoskrnl 中的函数进行堆管理。

进程在启动时带有默认的进程堆,通常有 1 MB 大小,如果需要会自动扩大。Win32 应用程序将使用此进程堆,线程调用 GetProcessHeap 函数得到一个指向堆的句柄,然后,再调用 HeapAlloc 和 HeapFree 函数从堆中分配和回收主存块。另外,也可使用 HeapCreate 函数创建另外的私有堆,使用完之后通过 Heap Destroy 释放堆空间,也只有另外创建的私有堆才能在进程的生命周期中被释放。

(3) 主存映射文件

主存映射文件主要用于以下三种场合:

- ① 把可执行文件 .exe 和动态链接库 .dll 文件装入主存,节省应用程序启动所需要的时间。
- ② 存取磁盘数据文件,减少文件 I/O 操作,且不必对文件进行缓存。
- ③ 实现多个进程共享数据和代码。

此外,高速缓存管理程序使用主存映射文件来读写高速缓存的页。

Win32 子系统向其应用进程提供文件映射对象(file mapping object)服务,实际上,此服务就是 Win32 子系统将区域对象服务通过 Win32 API 向应用进程所提供的。因此,区域对象是实现

主存映射文件功能的关键所在,用来映射虚拟地址,无论区域对象是在主存、页文件或是其他文件中,当应用程序访问它时,就像它在主存中一样。

区域对象能映射比进程地址空间大得多的文件,进程访问非常大的区域对象时,只能在自己的虚拟地址空间保留一个区域来映射区域对象的某一部分,被进程映射的那部分称为此区域的一个“视窗”。视窗机制通过映射区域对象的不同部分,允许进程访问超过其地址空间的区域。主存映射文件通过下面的函数和步骤来实现。

步骤 1 打开文件:使用 CreateFile 函数来建立和打开文件,此函数指定要建立或打开的文件名、访问方式等,执行此函数将返回一个文件句柄。

步骤 2 建立文件映射:使用 CreateFileMapping 函数创建文件映射对象,其实质是为文件创建一个区域对象,参数包括文件句柄、安全属性指针、指定保护属性和映射文件(区域)的大小等。执行此函数将返回一个句柄给文件映射对象。

步骤 3 读写文件视窗:通过 MapViewOfFile 函数所返回的指针来对视窗进行读写操作,此函数的参数包括文件映射对象句柄、访问方式、视窗起始地址在映射文件中的位移、视窗映射的虚拟地址空间字节数。

步骤 4 打开文件映射对象:使用 OpenFileMapping 函数打开已存在的文件映射对象,以便共享信息或进行通信,参数包括访问权限、子进程能否继承句柄和对象名。

步骤 5 解除映射:访问结束,使用 UnmapViewOfFile 函数解除映射,释放其地址空间中的视窗,参数指定释放区域的基地址。

4.8.3 主存管理的实现

1. 进程页表与地址映射

Windows 系统在 Intel x86 平台上采用二级页表来实现进程的逻辑地址到物理地址的转换。32 位逻辑地址被解释为 3 个分量:页目录索引(10 位)、页表页索引(10 位)和位移索引(12 位),页面大小为 4 KB。

进程都拥有单独的页目录,这是内核管理器所创建的特殊页,用于映射进程所有页表页的位置,它被保存在核心进程块(KPROCESS)中。页目录通常是进程私有的,但需要时也可在进程之间共享。页表是根据需要而创建的,大多数进程页目录仅指向页表的一小部分,所以进程页表中的虚地址往往不连续。

控制寄存器 CR3 指向页目录的存放地址,进程切换时,系统负责把运行进程的页目录的物理地址放入此寄存器中。页目录是由页目录项(Page Directory Entry,PDE)所组成的,每个 PDE 有 4 B 长,描述进程所有页表的状态和位置。每个进程需要一张页目录表(有 1 024 个 PDE)指出 1 024 张页表页,每张页表页描述 1 024 个页面,合计描述 4 GB 的虚拟地址空间。其中,用户进程最多占用 512 张页表页,系统进程占用另外 512 张页表页,并被所有用户进程共享。硬件页表项(Page Table Entry,PTE)包含有效/无效位、读写/只读位、用户页/系统页位、禁用高速缓存位、访问位、修改位等。为了加快页表的访问速度,系统也设置快表 TLB,关于逻辑地址到物理

地址的转换过程已经在前面介绍过,在此不再重复。

2. 页框数据库

在 Windows 系统中,所有主存页框组成页框数据库(page frame database),每个页框占一项,每一项称为一个 PFN(Page Frame Number,页框号)结构,用来跟踪物理主存的使用情况。页框数据库用一个数组表示,索引号从 0 起直至最大页框号。每一项记录页框状态是:空闲的、被占用的及被谁占用。PFN 在任一时刻可能处于下面 8 种状态之一。

- (1) 有效(valid):又称活动的,此页框在进程工作集中,有一个有效页表项指向此页。
- (2) 过渡(transition):此页框不在进程工作集中,但页的内容未被破坏,或正在进行 I/O 操作。
- (3) 后备(stand by):此页框曾被进程使用,内容未被修改过,但已从进程工作集中删除。页表项仍然指向这个页框,但被标记为“无效”和处于“过渡”状态。

(4) 修改(modified):此页和“后备”状态时相同,但页面被修改且尚未写入磁盘。页表项仍然指向此页框,但被标记为“无效”或处于“过渡”状态。所以,此页框被重新使用之前应先写回磁盘。

(5) 修改不写入(modified no write):此页和“修改”状态相同,但已被标记不写回磁盘。在文件系统驱动程序发出请求时,高速缓存管理器标记页面为“修改不写入”状态。

- (6) 空闲(free):此页框是空闲的,且不属于任何进程,但未被初始化清 0。
- (7) 零初始化(zeroed):此页框是空闲的,并且已被初始化为全零,随时可用。
- (8) 坏(bad):此页框已发生奇偶校验错或其他物理故障,不能被使用。

对于零初始化、空闲、后备、修改、坏、修改不写入共 6 种状态的页框都分别组成链表链接在一起,以便主存管理器能够很快定位特定状态的一个页框。“有效”页框和“过渡”页框不在链表中,“有效”页框由进程的页表来管理。

页框数据库项是定长的,根据页框的使用情况,页框数据结构 PFN 可能处于四种状态之一:有效(在工作集中)、后备或修改、清 0 或空闲、正在 I/O 页面,如图 4.34 所示。

对其中的相关项简单说明如下。

- (1) 页表项地址:指向此页框的页表项的虚拟地址。
- (2) 页面访问计数:当页框被首次加入工作集或当页面由于 I/O 操作在主存中被锁定时,页面访问计数就会增加;从主存解锁时,页面访问计数将减少。当页面访问计数为 0 时,此页不再属于工作集。可根据页面访问计数来更新 PFN,以便将此页框加入空闲、备用或修改链表中。
- (3) 类型:指 PFN 可能取的八种类型。
- (4) 标志:区别此页是否被修改过、是否为原型页表项(共享页)、是否包含奇偶校验错、此页正在被调入、此页正在执行写操作、非分页缓冲池开始/终止和页框调入错误。
- (5) 页表项的页框号:包含指向这个页表项 PTE 的页框号。
- (6) 原型页表项的内容:表示页框号项包含指向原始页框的页表项的内容。此页是共享页。
- (7) 共享计数:指向此页的页表项的个数。对于页表页,这个域是页表中有效页表项的个数。

程引发缺页中断的次数,减少调页 I/O 操作的数量。默认页面读取簇的数量取决于物理主存的大小,当主存大于 19 MB 时,通常代码页簇为 8 页、数据页簇为 4 页、其他页簇为 8 页。

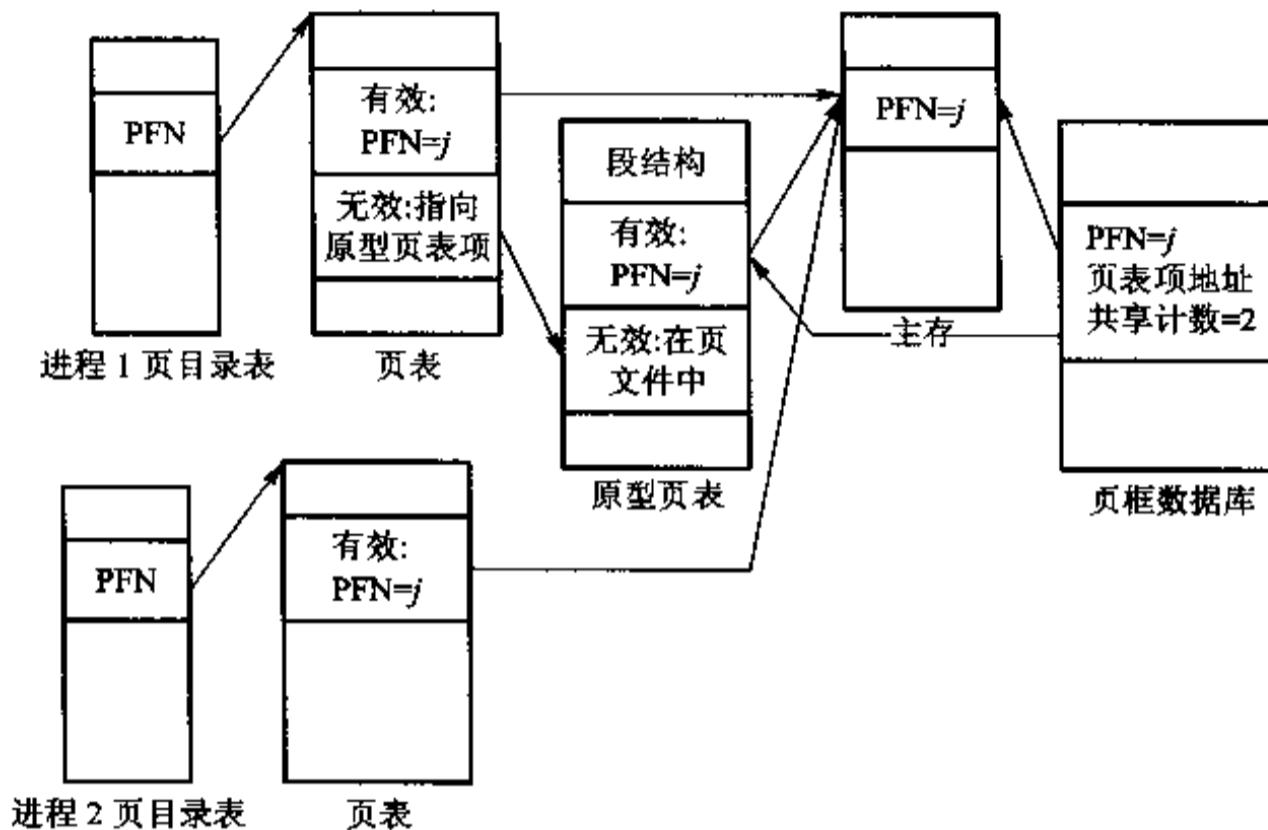


图 4.37 原型页表项

当线程产生缺页中断时,主存管理器还必须确定将所调入的虚页放在物理主存的何处,称为“置换策略”。如果缺页中断发生时物理主存已满,“置换策略”被用于确定哪个虚页必须从主存存储器中移出。在多处理器系统中,Windows 系统采用局部先进先出置换策略,而在单处理器系统中,其实现更接近于 LRU 策略。系统为每个进程分配一定数量的页框,称为“进程工作集”(或者为可分页的系统代码和数据分配一定数量的页框,称为“系统工作集”)。

(1) 工作集管理

① 进程工作集

系统初始化时基于物理主存的大小计算工作集,当物理主存大于 32 MB(Server 版大于 64 MB)时,进程的默认最小工作集为 50 页,最大工作集为 345 页。当缺页中断产生时,将检测进程的工作集限制和系统中的空闲主存。如果系统有足够的空闲页框,则允许进程把工作集规模增至最大值,甚至可以超过最大值,否则只能替换工作集中的页。

当物理主存中所剩的空闲页框数量较少时,主存管理器使用“平衡工作集管理器”自动修剪各个工作集,以增加系统中可用的空闲页框数量。平衡工作集管理器作为一个系统线程,检测每个进程的工作集,设法减少各个进程的当前工作集,直到主存空闲页框的数量足够多,且每个进程达到最小工作集为止。系统定时从进程中淘汰一个有效页,观察进程是否对此页发生缺页中断,以此测试和调整进程当前工作集的合适尺寸。如果进程继续执行,并未对被淘汰页发生缺页中断,则此进程的工作集减 1,此页框被链接至空闲链表中。

型 PTE 的值为无效。为了做到这一点,页框号数据库项中包含指向原始进程页表项或原型 PTE(共享页)的指针。如图 4.35 所示是页框的状态转换。

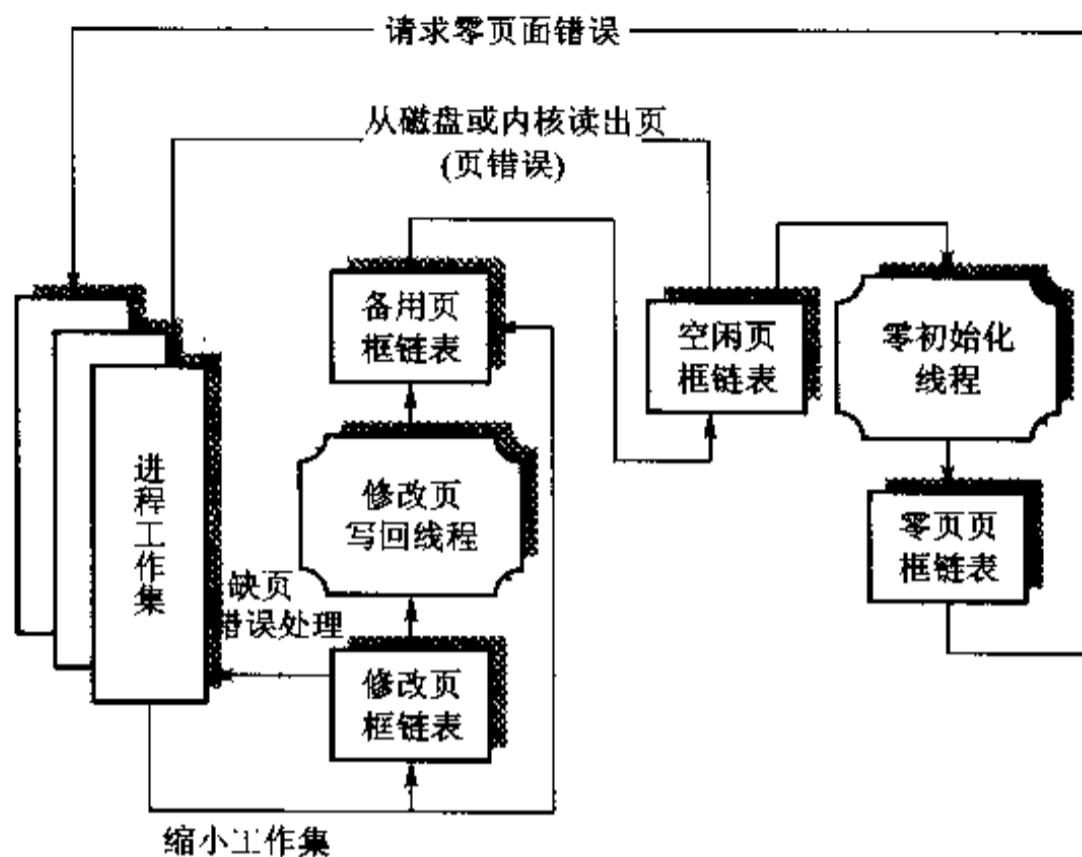


图 4.35 页框的状态转换

3. 缺页处理

(1) 无效页处理

当被访问的页无效时,将产生“无效页错误”。内核中断处理程序将这类错误分派给主存管理故障处理程序。此例程运行在引起错误的线程描述表上,系统根据情况分别进行处理。

① 访问一个未知页,其页表项为 0,或者页表不存在,说明线程首次访问一个地址。此时,系统必须为包含这个地址的页创建页表项,系统从此进程的虚拟地址描述符树中查找包含这一地址的 VAD,并利用它填充页表项。如果这个地址没有落在此 VAD 所覆盖的地址域中,或所在的地址域仅为保留而未被提交,主存管理程序将产生访问违约错误。

② 所访问的页没有驻留在主存中,而是在磁盘的某个页文件或映像文件中,系统分配一个物理页框,将所需的页从磁盘读出并放入工作集中。

- ③ 所访问的页在后备链表或修改链表中,将此页移至进程或系统工作集。
- ④ 访问一个请求零页,为进程工作集添加一个由零初始化的页。
- ⑤ 对一个“只读”页执行写操作,访问违约。
- ⑥ 从用户态访问一个只能在核心态访问的页,访问违约。
- ⑦ 对一个写保护页执行写操作,写违约。
- ⑧ 对一个写时复制的页执行写操作,为进程进行页复制。
- ⑨ 在多处理器系统中,对有效但尚未执行写操作的页执行写操作,将页表项的修改位置“1”。

(2) 原型页表项

当两个进程共享一个物理页面——页框时,主存管理器在进程页表中插入一个称作“原型页表项”(prototype Page Table Entry, prototype PTE)的数据结构来间接映射所共享的页面,它是进程间共享主存的内部实现机制。由于区域对象所指定的区是被多个进程所共享的主存,当区域对象第一次被创建时,对应于它的一个原型页表项同时被创建,于是多个进程可通过原型页表项来共享页框。

进程首次访问一个映射到区域对象视窗的页框时,主存管理器利用原型页表项中的信息填写进程的页表。当共享页框有效时,进程页表项和原型页表项都指向此物理页框。而当共享页框无效时,进程页表中的页表项由一个特殊的指针来填充,此指针指向描述此页框的原型页表项。此后,当页面被访问时,主存管理器就可以利用这个页表项中的指针来定位原型页表项,而原型页表项确切地描述被访问的页框的情况。

为了跟踪有多少进程正在访问共享页框,“页框号数据库项”内增加一个访问计数器。这样,当其访问计数值为 0 时,就将这个页框标记为无效,并且移至过渡链表或将修改页写回磁盘。使一个共享页框无效时,进程页表中的页表项由一个特殊的页表项来填充,这个特殊的页表项指向描述此页框的原型页表项,如图 4.36 所示。这样,当页面以后被访问时,主存管理器可以利用这个页表项中的编码信息来定位原型页表项,而原型页表项描述被访问的页面。

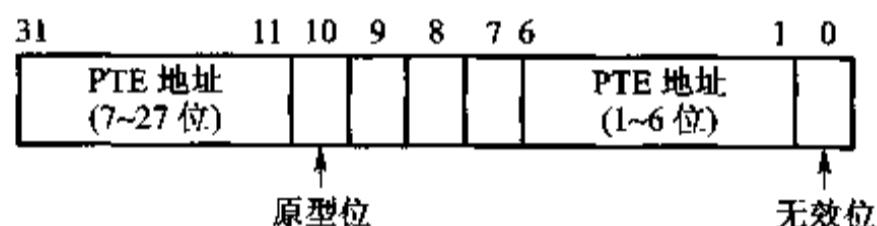


图 4.36 指向原型页表项的无效页表项的结构

引入原型页表项是为了尽可能地减少对各进程的页表项的影响。当一个共享页被换至磁盘时,只需修改原型页表项,各进程的页表项仍然保持不变。当共享页被换入主存时,系统将进程的页表项和原型页表项都指向对应的页框。例如,一段共享代码或一个数据页框在某时被调至磁盘,当将此页重新从磁盘调入时,只需更新原型页表项,使之指向此页新的物理位置,而共享此页的诸进程的页表项始终保持不变;此后,当进程访问此页时,实际的页表项才会得到更新。原型页表项位于可交换的系统空间内,它与页表一样可在必要时换出主存。

如图 4.37 所示是映射视窗中进程所涉及的各个数据结构之间的关系。在图 4.37 中,一个进程有两个虚页,第一页是有效的,并且进程页表项和原型页表项均指向它;第二页在页文件中,由原型页表项保存其确切位置,这个进程及映射到此页的其他进程的页表项均指向这个原型页表项。

4. 页面淘汰算法与工作集管理

Windows 系统采用请页式和页簇化调页技术,当线程发生缺页中断时,主存管理器将引发中断的页面及其后续的少量页面一起装入主存储器。根据局部性原理,这种页簇化策略能减少线

程引发缺页中断的次数,减少调页 I/O 操作的数量。默认页面读取簇的数量取决于物理主存的大小,当主存大于 19 MB 时,通常代码页簇为 8 页、数据页簇为 4 页、其他页簇为 8 页。

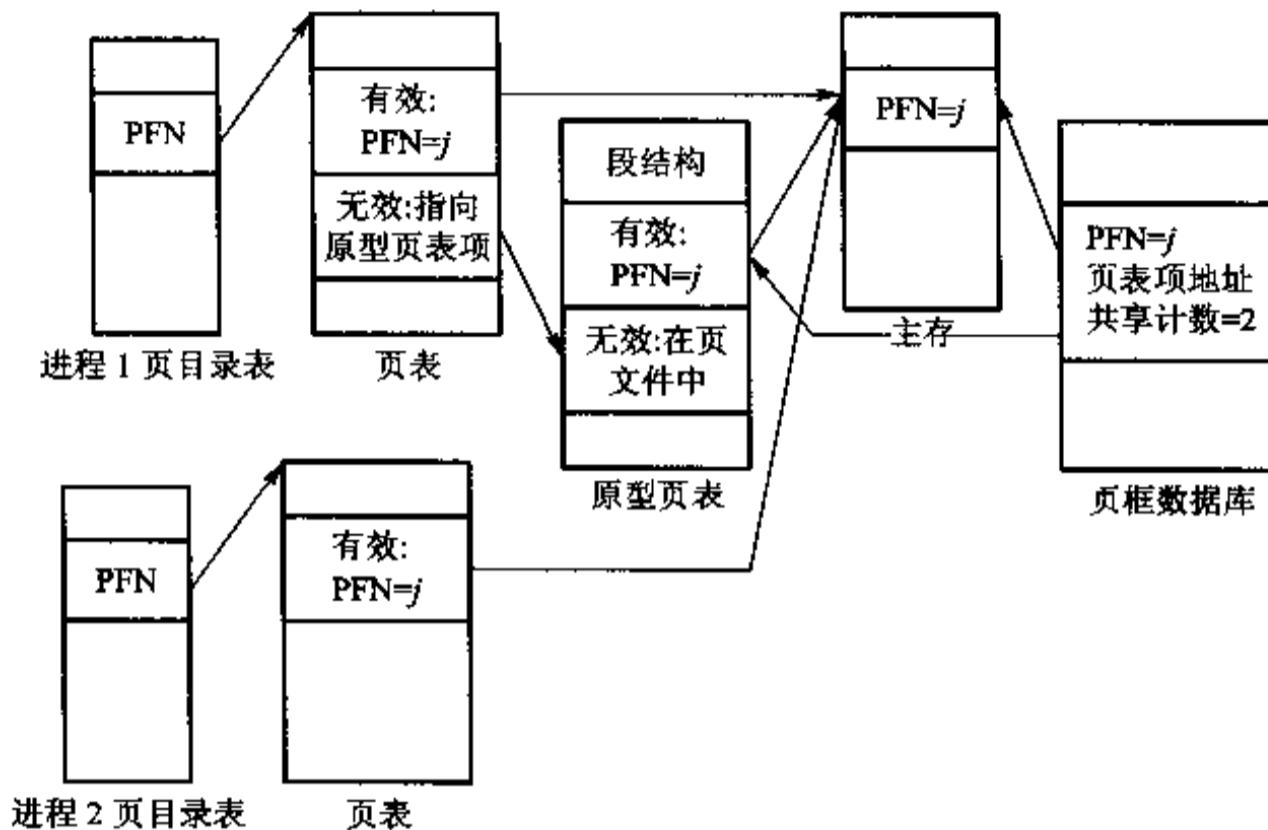


图 4.37 原型页表项

当线程产生缺页中断时,主存管理器还必须确定将所调入的虚页放在物理主存的何处,称为“置换策略”。如果缺页中断发生时物理主存已满,“置换策略”被用于确定哪个虚页必须从主存储器中移出。在多处理器系统中,Windows 系统采用局部先进先出置换策略,而在单处理器系统中,其实现更接近于 LRU 策略。系统为每个进程分配一定数量的页框,称为“进程工作集”(或者为可分页的系统代码和数据分配一定数量的页框,称为“系统工作集”)。

(1) 工作集管理

① 进程工作集

系统初始化时基于物理主存的大小计算工作集,当物理主存大于 32 MB(Server 版大于 64 MB)时,进程的默认最小工作集为 50 页,最大工作集为 345 页。当缺页中断产生时,将检测进程的工作集限制和系统中的空闲主存。如果系统有足够的空闲页框,则允许进程把工作集规模增至最大值,甚至可以超过最大值,否则只能替换工作集中的页。

当物理主存中所剩的空闲页框数量较少时,主存管理器使用“平衡工作集管理器”自动修剪各个工作集,以增加系统中可用的空闲页框数量。平衡工作集管理器作为一个系统线程,检测每个进程的工作集,设法减少各个进程的当前工作集,直到主存空闲页框的数量足够多,且每个进程达到最小工作集为止。系统定时从进程中淘汰一个有效页,观察进程是否对此页发生缺页中断,以此测试和调整进程当前工作集的合适尺寸。如果进程继续执行,并未对被淘汰页发生缺页中断,则此进程的工作集减 1,此页框被链接至空闲链表中。

② 系统工作集

系统工作集用来存储操作系统的可分页的代码和数据，其中可以驻留 5 种不同的页框：系统高速缓存页框、分页缓冲池、核心 Ntoskrnl.exe 中的可分页代码和数据、设备驱动程序中的可分页代码和数据、系统映像视窗。系统工作集的最大集和最小集与物理主存大小、专业版或服务器版有关，中等规模的主存的系统工作集的大小在 688~1 150 个页框之间。

(2) 平衡工作集管理器

平衡工作集管理器(KeBalanceSetManager)是在系统初始化时被创建的一个系统线程，主要对进程和系统工作集进行扩展和缩减。在其生命周期中等待两个不同的事件对象：在每秒激发一次的定时器到时后所产生的一个事件；由主存管理器确定工作集需要调整时所发出的另一个内部工作集管理器事件。

当系统缺页率很高或者空闲链表中的页框太少时，主存管理器就会唤醒平衡工作集管理器，它将调入工作集管理器开始修剪工作集。如果主存空间充足，当进程的工作集未达到最大值时，通过把所缺页调入主存的方式，使进程达到允许的最大工作集。

当平衡工作集管理器由于自身的 1 秒定时器到期而被唤醒时，执行以下一些操作。

- ① 它每被唤醒 4 次就产生一个事件，此事件唤醒另一个负责交换的系统线程。
- ② 为了改善访问时间，它检查备用链表，设法增加其页框数。
- ③ 它寻找并且提高处于 CPU“饥饿状态”的线程的优先级。
- ④ 它调节执行工作集修剪的时间和速度。

如果需要运行的线程核心栈被换出主存，或者此线程的进程已经被换出主存，交换程序也可由内核的调度代码唤醒。之后，交换程序寻找在一段时间内(小主存系统 3 s，中主存或大主存系统 7 s)一直处于等待态的线程。如果找到这样的一个线程，则将标记并回收线程的核心栈所占有的物理页框。所遵循的原则是：如果一个线程已经等待相当长的时间，那么它将等待更长的时间。当进程最后一个线程的核心栈也从主存中移出时，这个进程将标记为完全换出。这也正是已经等待很长时间的进程(如 Winlogon)可以有零工作集的原因。

综上所述，Windows 的虚拟存储管理系统总是为每个进程提供尽可能高的性能，而无须用户或系统管理员的干预。尽管这样，系统还提供 Win32 函数 SetProcessWorkingSet，让用户或系统管理员可以改变进程工作集的尺寸，不过工作集的最大规模不能超过系统初始化时所计算并保存的最大值。

本章小结

操作系统存储管理的基本功能有：存储分配、地址转换和存储保护、存储共享和存储扩充。存储分配是指为选中的多道运行作业分配主存空间；地址转换是把逻辑地址空间中的应用程序通过静态或动态地址重定位转换和映射到分配的物理地址空间中，以便应用程序执行；存储保护指各道程序只能访问自己的存储区域，而不能互相干扰，以免其他程序受到有意或无意的破坏；

存储共享指主存中的某些程序和数据可供不同的用户进程同时使用。

存储管理的另一个重要任务是解决好存储扩充的问题,使得主存利用率得以提高,使得主存中存放尽可能多的进程,使得应用程序不受可用物理主存大小的限制。操作系统支持多个用户进程在主存中同时运行,能满足多道程序设计需要的最简单的存储管理技术是分区方式,又分为固定分区和可变分区,可变分区的分配算法包括:最先适应、下一次适应、最佳适应、最坏适应和快速适应等分配算法。主存空间不足时主要采用的技术有:移动、对换和覆盖技术。

现代计算机系统都有某种虚拟存储硬设备支持,其中简单也是常用的是请求分页式虚拟存储管理,允许把进程的页面存放到若干不相邻接的主存页框中。虚拟存储器的思路是基于程序的局部性原理,不把一个用户进程的全部信息装入主存储器,而是仅将其中的当前使用部分先装入主存储器,它的任务是:当进程使用某部分地址空间时,保证将相应部分的程序加载到主存中。它采用自动的部分装入、部分对换的技术、主存/辅助存储器独立编址但统一使用的技术,使得进程的虚拟地址空间可以远远大于系统的物理地址空间,为用户编程提供了极大的方便。请求分页虚拟存储管理的基本原理包括:页面、页框、逻辑地址、页表、地址转换等,相关的概念还有:加快地址映射的快表、减少主存空间占用的多级页表和反置页表。在一个实际的操作系统中,还有一些重要的问题。

- (1) 页面装入策略:决定页面何时被装入主存储器,有请页式和预调式两种装入策略。
- (2) 页面清除策略:决定修改过的页面何时被回写到磁盘上,有请页式和预约式两种清除策略。
- (3) 页面分配策略:根据进程生命周期中分配的页框数是否改变,分为固定分配和可变分配。
- (4) 页面替换策略:根据页面替换算法的作用范围是整个系统还是局部于进程,分为全局页面替换算法和局部页面替换算法。

已经设计出许多页面替换算法:OPT、LRU、SCR、Clock、工作集、模拟工作集、缺页频率算法等。可通过工作集模型来指导确定进程常驻集的大小,使得进程的缺页中断率低,主存空间又能得到充分的利用。OPT 算法虽然好但并不可行,常用的算法是 LRU 和 Clock 算法及其变种。

请求分页虚拟存储管理不能支持模块化程序设计,请求分段虚拟存储管理能满足现代高级语言模块化程序设计所需的二维地址要求以及方便用户(程序员)编程和使用。段划分的基本原则是:按用户应用中在逻辑上有完整意义的内容进行分段,需要建立每个作业的段表来记录用户逻辑段与主存物理段之间的对应关系,主存的分配和去配管理与可变分区方式十分类似,可通过直接分配、移动分配或调出分配来完成。如果把上述两者有效地结合起来,在分页式存储管理的基础上实现分段式存储管理的就是段页式虚拟存储管理。

操作系统的存储管理功能与硬件存储器的组织结构和支撑设施密切相关,操作系统设计者应根据硬件情况和用户需要,采用各种有效的存储资源分配策略和保护措施。

习题四

一、思考题

1. 试述存储管理的基本功能。
2. 试述计算机系统中的存储器层次。为什么要配置层次式存储器？
3. 什么是逻辑地址(空间)和物理地址(空间)？
4. 何谓地址转换(重定位)？哪些方法可以实现地址转换？
5. 分区存储管理中常采用哪些分配策略？比较其优、缺点。
6. 什么是移动技术？在什么情况下采用这种技术？
7. 若采用表格方式管理可变分区，试绘制分配和释放一个存储区的算法流程图。
8. 什么是存储保护？在分区存储管理中如何实现分区的保护？
9. 什么是虚拟存储器？列举采用虚拟存储技术的必要性和可能性。
10. 试述请求分页虚拟存储管理的实现原理。
11. 试述请求分段虚拟存储管理的实现原理。
12. 分页虚拟存储管理中有哪几种常见的页面淘汰算法？
13. 试比较分页式存储管理和分段式存储管理。
14. 试述几种存储保护方法，其各适用于何种场合？
15. 试述存储管理中的碎片。各种存储管理中可能产生何种碎片？
16. 采用可变分区方式进行存储管理，假如允许用户运行时动态申请/归还主存资源，这时，系统可能因竞争主存资源而产生死锁吗？如果否，试说明之；如果是，试设计一种解决死锁的方案。
17. 试述分页式存储管理中决定页面大小的主要因素。
18. 试述实现虚拟存储器的基本原理。
19. 采用页式存储管理的存储器是否就是虚拟存储器，为什么？实现虚拟存储器必须要有哪些软硬件支撑？
20. 如果主存中的某页正在与外部设备交换信息，那么，当发生缺页中断时，可以将这一页淘汰吗？为什么？出现这种情况时，你能提出怎样的处理方法？
21. 为什么在页式存储器中实现程序共享时，必须对共享程序给出相同的页号？
22. 在段式存储器中实现程序共享时，共享段的段号是否一定要相同？为什么？
23. 试述段页式存储器的主要优、缺点。
24. 试述虚存管理与实存管理之间的主要区别。
25. 试述虚拟存储器与下列技术的关系：(1) 多道程序设计；(2) 程序地址重定位。
26. 什么是“抖动”？试给出抖动的例子。
27. 在可变分区存储管理中，回收一个分区时有多种不同的邻接情况，试讨论各种情况的处理方法。
28. 在请求分页存储管理中，若把进程的页框数增加一倍，则缺页中断次数会减少至一半吗？为什么？
29. 试讨论虚拟存储器容量与地址总线宽度、主存容量及辅存容量之间的关系。
30. 分页式存储管理中，试分析大页面与小页各自的优点。

31. 试述缺页中断与一般中断之间的区别。
32. 试述虚拟存储器与辅助存储器之间的关系。
33. 在请求分页虚拟存储的替换算法中,对于任何给定的驻留集尺寸,在什么引用串情况下,FIFO 与 LRU 替换算法一样(即所替换的页面与缺页中断率一样)?举例说明。
34. 解决大作业和小主存之间的矛盾有哪些途径?简述其实现思想。
35. 在请求分页虚拟存储系统中,若已测得时间利用率为:CPU 20%、分页磁盘 97.7%、其他外部设备 50%。试问哪些措施可以改善 CPU 利用率?
36. 在请求分页虚拟存储系统中,分析下列程序设计风格对系统性能所产生的影响:(1) 迭代法;(2) 递归法;(3) 常用 goto 语句;(4) 转子程序;(5) 动态数组。

二、应用题

1. 在一个请求分页虚拟存储管理系统中,一个程序运行的页面走向是:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

分别用 FIFO、OPT 和 LRU 算法,对于分配给程序 3 个页框、4 个页框、5 个页框和 6 个页框的情况,求出缺页中断次数和缺页中断率。

2. 在一个请求分页虚拟存储管理系统中,一个作业共有 5 页,执行时其访问页面的次序为:

- (1) 1,4,3,1,2,5,1,4,2,1,4,5;
- (2) 3,2,1,4,4,5,5,3,4,3,2,1,5。

若分配给此作业 3 个页框,分别采用 FIFO 和 LRU 页面替换算法,求出各自的缺页中断次数和缺页中断率。

3. 一个页式存储管理系统使用 FIFO、OPT 和 LRU 页面替换算法,如果一个作业的页面走向为:

- (1) 2,3,2,1,5,2,4,5,3,2,5,2;
- (2) 4,3,2,1,4,3,5,4,3,2,1,5;
- (3) 1,2,3,4,1,2,5,1,2,3,4,5。

当分配给此作业的物理块数分别为 3 和 4 时,试计算访问过程中所发生的缺页中断次数和缺页中断率。

4. 在可变分区存储管理下,按地址排列的主存空闲区为:10 KB、4 KB、20 KB、18 KB、7 KB、9 KB、12 KB 和 15 KB。对于下列连续存储区的请求:(1) 12 KB、10 KB、9 KB,(2) 12 KB、10 KB、15 KB、18 KB,试问:使用首次适应算法、最佳适应算法、最差适应算法和下次适应算法,哪个空闲区将被使用?

5. 给定主存空闲区,按照地址从小到大排列为:100 KB、500 KB、200 KB、300 KB 和 600 KB。现有用户进程依次为 212 KB、417 KB、112 KB 和 426 KB。

- (1) 分别用 first-fit、best-fit 和 worst-fit 算法将它们装入主存的哪个分区?
- (2) 哪个算法能最有效地利用主存?

6. 一个 32 位计算机系统使用二级页表,虚地址被分为 9 位顶级页表、11 位二级页表和页内位移。试问:页面长度是多少?虚地址空间共有多少个页面?

7. 一个进程以下列次序访问 5 个页:A、B、C、D、A、B、E、A、B、C、D、E;假定使用 FIFO 替换算法,在主存储器中有 3 个和 4 个空闲页框的情况下,分别给出页面替换次数。

8. 某计算机有缓存、主存储器、辅助存储器来实现虚拟存储器。如果数据在缓存中,访问数据需要 A ns;如果数据在主存但不在缓存,需要 B ns 将其装入缓存,然后才能访问;如果数据不在主存而在辅存,需要 C ns 将其读入主存。最后,用 B ns 再读入缓存,然后才能访问。假设缓存命中率为 $(n - 1)/n$,主存命中率为 $(m - 1)/m$

m , 则数据平均访问时间是多少?

9. 某计算机有 cache、主存、辅存来实现虚拟存储器。如果数据在 cache 中, 访问它需要 20 ns; 如果数据在主存但不在 cache, 需要 60 ns 将其装入缓存, 然后才能访问; 如果数据不在主存而在辅存, 需要 12 μ s 将其读入主存, 然后, 用 60 ns 再读入 cache, 然后才能访问。假设 cache 命中率为 0.9, 主存命中率为 0.6, 则数据平均访问时间是多少(单位:ns)?

10. 有一个分页系统, 其页表存放在主存中。

(1) 如果对主存的一次存取需要 1.2 μ s, 试问实现一次页面访问的存取需花费多少时间?

(2) 若系统配置了相联存储器, 命中率为 80%, 假定页表表目在相联存储器的查找时间忽略不计, 试问实现一次页面访问的存取时间是多少?

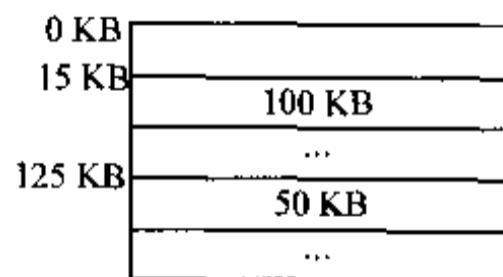
11. 给定段表如下:

段号	段首址	段长
0	219	600
1	2 300	14
2	90	100
3	1 327	580
4	1 952	96

给定地址为段号和位数:(1) $<0,430>$; (2) $<3,400>$; (3) $<1,1>$; (4) $<2,500>$; (5) $<4,42>$, 试求出对应的主存物理地址。

12. 某计算机系统提供 24 位虚存空间, 主存空间为 2^{18} B, 采用分页式虚拟存储管理, 页面尺寸为 1 KB。假定应用程序产生虚拟地址 11123456(八进制), 而此页而分得的块号为 100(八进制), 说明此系统如何产生相应的物理地址并写出物理地址。

13. 主存中有两个空闲区如下图所示, 现有作业序列依次为: Job1 要求 30 KB; Job2 要求 70 KB; Job3 要求 50 KB; 使用首次适应、最坏适应和最佳适应算法处理这个作业序列, 试问哪种算法可以满足分配要求? 为什么?



14. 设有一页式存储管理系统, 向用户所提供的逻辑地址空间最大为 16 页, 每页 2 048 B, 主存共有 8 个存储块。试问逻辑地址至少应是多少位? 主存空间有多大?

15. 在一分页存储管理系统中, 逻辑地址长度为 16 位, 页面大小为 4 096 B, 现有逻辑地址 2F6AH, 且第 0、1、2 页依次存放在第 10、12、14 号物理块中, 试问相应的物理地址是多少?

16. 有数组 int A[100][100]; 元素按行存储。在虚存系统中, 采用 LRU 淘汰算法, 一个进程有 3 页主存空间, 每页可以存放 200 个整数。其中第 1 页存放程序, 且假定程序已在内存中。

程序 A:

```
for(i=0;i<100;i++)
    for(j=0;j<100;j++)
        A[i,j]=0;
```

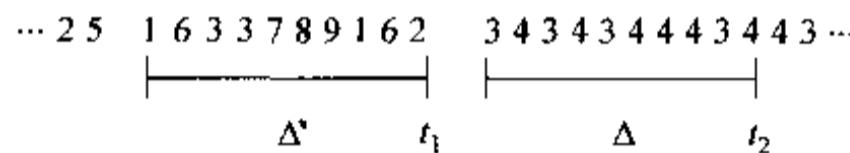
程序 B:

```
for(j=0;j<100;j++)
    for(i=0;i<100;i++)
        A[i,j]=0;
```

分别就程序 A 和 B 的执行进程计算缺页次数。

17. 一台机器有 48 位虚地址和 32 位物理地址, 若页长为 8 KB, 问页表共有多少个页表项? 如果设计一个反置页表, 则有多少个页表项?

18. 在页式虚拟存储管理中, 为了解决抖动问题, 可采用工作集模型以决定分给进程的物理块数。有如下的页面访问序列: 窗口尺寸 $\Delta=9$, 试求 t_1 、 t_2 时刻的工作集。



19. 有一个分页虚拟存储系统, 测得 CPU 和磁盘的利用率如下。试指出每种情况下所存在的问题和可采取的措施。

- (1) CPU 利用率为 13%, 磁盘利用率为 97%;
- (2) CPU 利用率为 87%, 磁盘利用率为 3%;
- (3) CPU 利用率为 13%, 磁盘利用率为 3%。

20. 在一个分页虚拟存储系统中, 用户编程空间为 32 个页, 页长 1 KB, 主存空间为 16 KB。如果应用程序有 10 页长, 若已知虚页 0、1、2、3, 已分得页框 4、7、8、10, 试把虚地址 0AC5H 和 1AC5H 转换成对应的物理地址。

21. 某计算机有 4 个页框, 每一页的装入时间、最后访问时间、访问位 R、修改位 D 如下表所示(时间用时钟点数表示)。

Page	loaded	last reference	R	D
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

使用 FIFO、LRU、二次机会算法分别会淘汰哪一页?

22. 一个进程已分配到 4 个页框, 每一页的装入时间、最后访问时间、访问位 R、修改位 D 如下表所示(所有数字为十进制数, 且从 0 开始), 当进程访问第 4 页时, 产生缺页中断。请分别用 FIFO、LRU 和 NRU 算法决定缺页中断服务程序所选择换出的页面。

Page	Page frame	loaded	last reference	R	D
2	0	60	161	0	1
1	1	130	160	0	0
0	2	26	162	1	0
3	3	20	163	1	1

23. 考虑下面的程序：

```
for (int i=0; i<20; i++)
    for(int j=0; j<10; j++)
        a[i] = a[i] * j;
```

试举例说明此程序的空间局部性和时间局部性。

24. 在某页式虚拟存储系统中,假定访问主存的时间是 2 ms,平均缺页中断处理时间为 25 ms,平均缺页中断率为 5 %,试计算在此虚拟存储系统中,平均有效访问时间是多少?

25. 一个有快表的页式虚拟存储系统,设主存访问周期为 1 μ s,内、外存传送一个页面的平均时间为 5 ms。如果快表的命中率为 75 %,缺页中断率为 10 %,忽略快表访问时间,试求主存的有效存取时间。

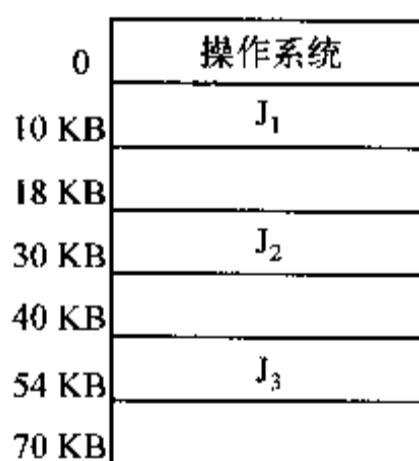
26. 假设某虚存的用户空间为 1 024 KB,页面大小为 4 KB,主存空间为 512 KB。已知用户的 10、11、12、13 号虚页分得主存页框号为 62、78、25、36,求出虚地址 0BEBC(H)(十六进制)的实地址(十六进制)。

27. 某请求分页存储系统使用一级页表,假设页表全部放在主存储器内。

- 若一次访问主存花费 120 ns,那么,访问一个数据的时间是多少?
- 若增加一个快表,在命中或失误时均需有 20 ns 开销,如果快表的命中率为 80 %,则访问一个数据的时间是多少?

28. 设某系统中作业 J₁、J₂、J₃ 占用主存储器的情况如下图所示。现有一个长度为 20 KB 的作业 J₄ 要装入主存,当采用可变分区分配方式时,请回答:

- J₄ 装入前的主存已分配表和未分配表的内容。
- 写出装入 J₄ 的工作流程,并说明采用何种分配算法。



29. 考虑下列段表:

段号	起始地址	段长
0	200	500
1	890	30
2	120	100
3	1250	600
4	1800	88

对于下面的逻辑地址,求物理地址,如发生越界请指明。(1) $<0,480>$; (2) $<1,25>$; (3) $<1,14>$; (4) $<2,200>$; (5) $<3,500>$; (6) $<4,100>$ 。

30. 请页式存储管理中,进程访问地址的序列为: 10, 11, 104, 170, 73, 305, 180, 240, 244, 445, 467, 366。试问:

- (1) 如果页面大小为 100 B,给出页面访问序列。
- (2) 若进程分得 3 个页框,采用 FIFO 和 LRU 替换算法,求缺页中断率。

31. 设程序大小为 460 个字,考虑如下访问序列:

55, 20, 108, 180, 79, 310, 170, 255, 246, 433, 488, 369

- (1) 设页面大小为 100 个字,试给出访问序列页面走向。
- (2) 假设程序可用主存为 200 个字,采用 FIFO、LRU 和 OPT 淘汰算法,试求出缺页中断率。

32. 假设计算机主存为 2 MB,其中,操作系统占用 512 KB,每个应用程序使用 512 KB 主存空间。如果所有程序都有 70% 的 I/O 等待时间,那么,再增加 1 MB 主存,吞吐率增加多少?

33. 一个计算机系统有足够的主存空间存放 4 道程序,这些程序有一半时间在空闲等待 I/O 操作。问多大比例的 CPU 时间被浪费了?

34. 如果执行一条指令平均需要 1 ms,处理一个缺页中断另外需要 n ms,给出当缺页中断每 k 条指令发生一次时,指令的实际执行时间。

35. 一台计算机的主存空间为 1 024 个页面,页表置于主存中,从页表中读取一个字的开销是 500 ns。为了减少开销,使用有 32 个字的快表,查找速度为 100 ns。要把平均开销降至 200 ns,所需的快表命中率是多少?

36. 假设一条指令平均需要花费 $1 \mu s$,但若发生缺页中断则需 $2 001 \mu s$ 。如果一个程序运行 60 s,其间发生 15 000 次缺页中断,若可用主存是原来的两倍,这个程序运行需要花费多少时间?

37. 在分页式虚拟存储管理中,若采用 FIFO 替换算法,会发生:分给作业的页面越多,进程执行时缺页中断率越高的奇怪现象。试举例说明这个现象。

38. 假设一个任务被划分成 4 个大小相等的段,每段有 8 项页描述符表,页面大小为 2 KB。试问段页式存储系统中:(1) 每段的最大尺寸是多少? (2) 此任务的逻辑地址空间最大是多少? (3) 若此任务访问逻辑地址空间 5ABCH 中的一个数据,试给出逻辑地址的格式。

39. 进程在某时刻的页表如下,设页面大小为 1 KB,表中的所有数字是十进制数。

页号	有效位	访问位	修改位	页框号
0	1	1	0	4
1	1	1	1	7
2	0	0	0	
3	1	0	0	2
4	0	0	0	
5	1	0	1	0

下列虚地址转换为物理地址的值是多少?

- (1) 1052;
- (2) 2221;
- (3) 5499。

40. 已知某系统页面为 4 KB, 页表项 4 B, 采用多级页表映射 64 位虚地址空间。若限定最高层页表占 1 页, 问它可以采用几级页表?

41. 采用 LRU 置换算法的虚拟分页存储管理系统, 其页面尺寸为 4 KB, 主存访问速度为 100 ns, 快表访问速度为 20 ns, 缺页中断处理耗时 25 ms。现有一个长度为 30 KB 的进程 P 进入系统, 分配给 P 的页框有 3 块, 进程的所有页面都在运行时动态装入。若 P 访问快表的命中率为 20%, 对于下述页面号访问序列:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

试计算平均有效访问时间是多少(单位:ns)?

42. 在请求分页虚拟存储管理系统中, 若驻留集为 m 个页框, 页框初始时为空, 在长为 p 的引用率中具有 n 个不同的页面 ($n > m$), 对于 FIFO、LRU 两种页面替换算法, 试给出缺页中断的上限和下限, 并举例说明之。

43. 在请求分页虚拟存储管理系统中, 页表保存在寄存器中。若替换一个未修改过页面的缺页中断处理需要 2 ms, 若替换一个已修改过页面的缺页中断处理需要另加写盘时间 8 ms, 主存存取周期为 1 μ s。假定 70% 被替换的页面被修改过, 为了保证有效存取时间不超过 2 μ s, 所允许的最大缺页中断率是多少?

44. 若主存中按照地址递增次序有 3 个不邻接的空间区 F_1 、 F_2 、 F_3 , 其大小分别是 50 KB、120 KB 和 25 KB。请给出后备作业序列, 使得实施分配时:

- (1) 采用最佳适应算法效果好, 但采用首次适应与最坏适应算法效果不好。
- (2) 采用最坏适应算法效果好, 但采用首次适应与最佳适应算法效果不好。
- (3) 采用最佳适应算法效果不好, 但采用首次适应与最坏适应算法效果好。

45. 有两台计算机 P_1 和 P_2 , 它们各有一个硬件高速缓存 C_1 和 C_2 , 且各有一个主存储器 M_1 和 M_2 。其性能为:

	C_1	C_2	M_1	M_2
存储容量	4 KB	4 KB	2 MB	2 MB
存取周期	60 ns	80 ns	1 μ s	0.9 μ s

若两台机器的指令系统相同, 其指令执行时间与存储器的平均存取周期成正比。如果在执行某个程序时, 所需指令或数据在高速缓存中有取到的概率 P 是 0.7, 试问: 这两台计算机哪个速度更快? 当 $P=0.9$ 时, 哪个处理

器速度更快?

46. Linux 系统中采用多种 cache 来改善系统运行性能,试解释所采用的缓存 cache、页面 cache、交换 cache 和硬件 cache 的作用。

47. 假设一个物理存储器有 4 个页框,对于下面每种策略,给出引用串:

P1,P2,P3,P1,P4,P5,P1,P2,P1,P4,P5,P3,P4,P5

的缺页数目(所有页框最初都是空的)。试用下列算法求出缺页中断次数。

- (1) OPT;
- (2) FIFO;
- (3) SCR;
- (4) 改进的 CLOCK;
- (5) LRU;
- (6) MIN
- (7) WS。

48. 考虑下面的引用串:

P1,P2,P3,P4,P1,P2,P5,P1,P2,P3,P4,P5

对于范围 1~6 的页框,使用 FIFO 页面置换方案,确定其所产生的缺页数目。画图表示缺页次数和页框数之间的关系,以说明 Belady 异常。

49. 考虑 $\Delta=3$ 的工作集模型,给出进程 P 的引用串如下:

y x x x x x y y u x x x y z y z w w w z x x w w

- (1) 进程 P 能够拥有的最大工作集是什么?
- (2) 进程 P 能够拥有的最小工作集是什么?

50. 考虑引用串 abcdebcdbddbdd。假设使用工作集置换策略,确定最小的窗口大小,以保证其上的引用串至多产生 5 次缺页。说明每次引用时哪些页面驻留在主存中。用一个星号标记缺页。

51. 设进程分得 3 个页框,其执行访问序列为:

0,1,2,3,0,1,2,3,0,1,2,3,4,5,6,7

试采用:(1) Belady;(2) LRU;(3) LFU;(4) FIFO 算法来分别计算缺页中断次数,并给出缺页时加进主存的页号。



第五章

设备管理

现代计算机系统配置大量外部设备,外部设备分为两大类:一类是存储型设备(如磁带机、磁盘机等),以存储大量信息和快速检索为目标,在系统中存储持久性信息,它作为主存储器的扩充,所以又称辅助存储器;另一类是 I/O 型设备(如显示器、卡片机、打印机、通信设备等),它们把外界信息输入计算机,把计算结果从计算机中输出,完成计算机之间的通信成机交互。

设备管理是操作系统中最为庞杂和琐碎的部分,通常使用 I/O 中断、缓冲区管理、通道、设备驱动调度等多种技术,这些措施较好地克服了由于设备和 CPU 速度的不匹配所引起的问题,使主机和设备能够并行工作,提高设备使用效率。另一方面,操作系统将所有设备(如磁带机、打印机、显示器和终端等)都定义为文件,将其统一在文件系统之下,赋予文件属性,对设备的操作就类似于对文件的操作,其优点是:尽可能统一文件和设备的 I/O 处理;尽可能把设备文件和普通文件纳入同一保护机制下。为了方便用户或高层进程使用,设备管理还对各种设备进行抽象,配置驱动程序,提供统一界面,屏蔽设备的物理细节和操作过程。为此,设备管理应具有以下功能:设备中断处理;缓冲区管理;设备分配和去配;设备驱动调度;虚拟设备及其实现。

I/O 硬件原理

5.1.1 I/O 系统

通常把 I/O 设备及其接口线路、控制部件、通道和管理软件称为 I/O 系统,把计算机的主存储器和设备介质之间的信息传送操作称为 I/O 操作。随着计算机技术的飞速进步和应用领域的不断扩大,计算机的 I/O 信息量急剧增加,I/O 设备的种类和数量越来越多,它们与主机的联络和信息交换方式各不相同,I/O 操作不仅影响计算机的通用性和可扩充性,而且成为计算机系统综合处理能力及性能价格比的重要因素。

可按照不同方式对设备进行分类:按 I/O 操作特性,分为输入型设备、输出型设备和存储型设备;按 I/O 信息交换单位,分为字符设备和块设备。输入型设备和输出型设备通常是字符设备,它与主存储器进行信息交换的单位是字节,即一次交换 1 个或多个字节;存储型设备通常是

块设备。存储型设备又可分为顺序存取存储设备和直接存取存储设备。前者严格依赖信息的物理位置进行定位和读写,如磁带。后者的重要特性是存取任何一个物理块所需的时间几乎不依赖于此信息所处的位置,如磁盘。所谓块是存储介质上连续信息所组成的一个区域,块设备每次与主存储器交换一个或多个块信息。

不同设备的物理特性存在很大的差异,主要表现在:数据传输速率差别大,数据表示方式和传输单位不尽相同,错误的性质、报错方法及应对措施都不一样。这些差异使得无论从操作系统还是用户的角度都难以获得规范、一致的 I/O 解决方案。

5.1.2 I/O 控制方式

I/O 控制在计算机处理中占据重要的地位,为了有效地实现物理 I/O 操作,必须通过软硬件技术,对 CPU 和设备的职能进行合理分工,以平衡系统性能和硬件成本之间的矛盾。按照 I/O 控制器功能的强弱以及它和 CPU 之间联系方式的不同,可以把设备的控制方式分为 4 类,它们之间的差别在于:CPU 和设备并行工作的方式和程度不同。CPU 和设备并行工作具有重要的意义,能大幅度提高计算机系统的效率和资源利用率。

1. 轮询方式

轮询方式又称程序直接控制方式,使用查询指令测试设备控制器的忙闲状态位,确定主存储器和设备是否能交换数据。假如 CPU 上所运行的程序需要从设备读入一批数据,则 CPU 程序设置交换字节数和数据读入主存储器的起始地址,然后,向设备发出查询指令,设备控制器便把状态返回给 CPU。如果 I/O 操作忙或未就绪,则重复测试过程,继续进行查询;否则,开始数据传送,CPU 从 I/O 接口(控制器数据寄存器)读取一个字,再用存储指令保存到主存储器。如果传送尚未结束,再次向设备发出查询指令,直到全部数据传输完成。轮询方式使用 3 条指令:

- (1) 查询指令:查询设备是否就绪;
- (2) 读写指令:当设备就绪时,执行数据交换;
- (3) 转移指令:当设备未就绪时,执行转移指令转向查询指令继续查询。

有时系统中同时有多个设备要求执行 I/O 操作,那么对每个设备都编写一段 I/O 数据处理程序,然后,轮流查询这些设备的状态位,当某一个设备准备就绪时,就调用这个设备的 I/O 处理程序进行数据传输,否则依次轮询下一个设备是否准备好。

一方面,CPU 轮询设备当前的状态,会终止原程序的执行,浪费宝贵的时间;另一方面,I/O 准备就绪后,需要 CPU 参与数据传输工作。可见 CPU 和设备只能串行工作,使主机不能充分发挥功效,设备也不能得到合理利用,整个系统效率很低。

2. 中断方式

中断方式要求 CPU 与设备控制器及设备之间存在中断请求线,设备控制器的状态寄存器有相应的中断允许位,CPU 与设备之间传输数据的过程如下。

- (1) 进程发出启动 I/O 指令,这时 CPU 会加载控制信息到设备控制器的寄存器,然后,进程继续执行不涉及本次 I/O 数据的任务,或放弃 CPU 等待设备 I/O 操作完成;

(2) 设备控制器检查状态寄存器的内容,按照 I/O 指令的要求,执行相应的 I/O 操作(如发现读请求时,命令设备把数据传送到 I/O 缓冲寄存器),一旦传输完成,设备控制器通过中断请求线发出 I/O 中断信号;

(3) CPU 收到并响应 I/O 中断后,转向设备的 I/O 中断处理程序执行;

(4) 中断处理程序(有的操作系统把控制权再交给设备驱动程序,并退出中断)执行数据读取操作,将 I/O 缓冲寄存器的内容写入主存储器,操作结束后退出中断处理程序,返回发生中断前的执行状态;

(5) 进程调度程序在适当时刻对于得到数据的进程恢复执行。

在 I/O 中断方式中,如果设备控制器的数据缓冲区较小,当缓冲器装满后便会发生中断,那么在数据传输的过程中,发生中断的次数会较多,这样会耗用大量 CPU 时间。若系统配置多种设备,这些设备都通过中断处理方式实现并行工作,会使中断次数急剧增加,造成 CPU 来不及响应或丢失数据的现象。但是,程序中断方式 I/O 由于不必忙式轮询 I/O 的准备情况,CPU 和设备可实现部分并行操作,与程序查询 I/O 的串行工作方式相比,使 CPU 资源得到更充分的利用。

3. DMA 方式

虽然中断方式消除程序轮询中的忙式测试,提高了 CPU 资源的利用率,但在响应中断请求之后,必须停止现行程序,转入中断处理程序并参与数据传输操作。如果设备能直接与主存储器交换数据而不占用 CPU,那么,CPU 资源的利用率还可以提高,这就出现了直接存储器存取(Direct Memory Access,DMA)方式。在 DMA 方式中,主存储器和设备之间有一条数据通路,成块地传送数据,无须 CPU 干预,实际数据传输操作由 DMA 直接完成。为了实现直接存储器存取操作,至少需要以下一些逻辑部件。

(1) 主存地址寄存器:存放主存中需要交换数据的地址,DMA 传送之前,由程序送入首地址;DMA 传送过程中,每次交换数据都把地址寄存器的内容加 1;

(2) 字计数器:记录传送数据的总字数,每次传送一个字就把字计数器减 1;

(3) 数据缓冲寄存器或数据缓冲区:暂存每次传送的数据;

(4) 设备地址寄存器:存放 I/O 信息的地址,如磁盘的柱面号、磁道号、块号;

(5) 中断机制和控制逻辑:用于向 CPU 提出 I/O 中断请求,及保存 CPU 发来的 I/O 命令,管理 DMA 的传送过程。

DMA 与主存储器之间采用字传送,DMA 与设备之间可能是字位或字节传送,所以,DMA 中要设置数据移位寄存器、字节计数器等硬件逻辑。可能会出现这样的问题:为什么控制器从设备读取数据后不立即将其送入主存储器,而需要内部缓冲区呢?其原因是,一旦磁盘开始读数据,从磁盘读出比特流的速率是恒定的,无论控制器是否做好接收这些比特的准备,若此时控制器要将数据直接复制到主存储器中,必须在每个字传送完毕后获得对系统总线的控制权。如果其他设备也在争用总线,则有可能暂时等待,当上一个字还未送入主存储器而下一个字已经到达时,控制器只能另寻暂存之地,如果总线非常忙,则控制器可能需要对大量的信息暂存。可见,采用内部缓冲区,在 DMA 操作启动前不需要使用总线,这样控制器的设计就比较简单,因为从

DMA 到主存储器的传输对时间的要求并不十分严格。

DMA 不仅设有中断机制,还增加 DMA 传输控制机制,若出现 DMA 与 CPU 同时经总线访问主存储器的情况,CPU 总是把总线占有权让给 DMA,DMA 的这种占有称为“周期窃用”。窃取时间通常为一个存取周期,让设备和主存储器之间交换数据,而不再需要 CPU 干预,这样可以减轻 CPU 的负担。每次传输数据时,不必进入中断系统,进一步提高 CPU 资源的利用率。

由于 DMA 方式的实现线路简单,价格低廉,小型计算机、微型计算机中的快速设备均采用这种方案。但是,DMA 传输需要窃用时钟周期,会降低 CPU 的处理效率;DMA 的功能不够强,不能满足复杂的 I/O 操作要求,因而在大中型计算机中一般使用通道技术。

4. 通道方式

DMA 方式与程序中断方式相比,进一步减少了 CPU 对 I/O 操作的干预,但每发出一次 I/O 指令,只能读写一个数据块,用户希望一次能够读写多个离散的数据块,并把它们传送到不同的主存区域或相反,则需要由 CPU 发出多条启动 I/O 指令及进行多次 I/O 中断处理才能完成。通道方式是 DMA 方式的发展,能够再次减少 CPU 对 I/O 操作的干预。同时,为了充分发挥 CPU 和设备之间的并行工作能力,也让种类繁多且物理特性各异的设备能够以标准的接口连接到系统中,计算机系统引入自成体系的通道结构,通道方式的出现是现代计算机系统功能不断完善、性能不断提高的结果,是计算机技术的重要进步。

通道又称 I/O 处理器,能完成主存储器和设备之间的信息传送,与 CPU 并行地执行操作。采用通道技术主要解决 I/O 操作的独立性和硬部件工作的并行性,由通道来管理和控制 I/O 操作,大大减少设备和 CPU 之间的逻辑联系,把 CPU 从琐碎的 I/O 操作中解放出来,实现设备和 CPU 并行操作,通道之间并行操作,设备之间并行操作,达到提高整个系统效率的目的。

具有通道装置的计算机系统的主机、通道、控制器和设备之间采用四级连接,实施三级控制。一个 CPU 通常可以连接若干通道,一个通道可以连接若干控制器,一个控制器可以连接若干台设备。CPU 通过执行 I/O 指令对通道实施控制,通道通过执行通道命令对控制器实施控制,控制器发出动作序列对设备实施控制,设备执行相应的 I/O 操作。

采用 I/O 通道设计后,I/O 操作的过程如下:CPU 在执行主程序时遇到 I/O 请求,启动在指定通道上选址的设备,一旦启动成功,通道开始控制设备进行操作,这时 CPU 就可以执行其他任务并与通道并行工作,直到 I/O 操作完成;当通道发出 I/O 操作结束中断时,处理器才响应并停止当前的工作,转而处理 I/O 操作结束事件。

5.1.3 设备控制器

I/O 设备通常由机械部件和电子部件所组成,一般是将其分开处理,以达到模块化和通用性的设计目标。电子部件称为设备控制器或适配器,在微型计算机中,它是可插入主板扩充槽的印刷电路板;机械部件是指设备本身。

控制器卡上一般都设有接线器,与设备相连的电缆线相接,许多控制器可以控制两个或更多的同类设备,它和设备之间的接口采用国际标准,这样各计算机厂商都可制造与标准接口相匹配

的控制器和设备,例如,IDE(Integrated Device Electronics,集成设备电子器件)设备、SCSI(Small Computer System Interface,小型计算机系统接口)设备。

操作系统是与控制器交互,而非与设备本身交互,多数微型计算机和小型计算机采用单总线模型,实现CPU和控制器之间的数据传输,而大中型计算机则采用多总线结构和多通道方式,以提高CPU与设备的并行操作程度。

控制器和设备之间的接口是一种低级接口,例如,磁盘被格式化为每个扇区512B,实际上从磁盘读出来的是字位串,以头标开始,随后是扇区的4 096比特(512×8),最后是纠错码。其中,头标在磁盘格式化时写入,记录扇区地址,包括柱面号、磁头号、扇区号及同步信息。控制器的任务是把串行的位流装配成字节,存入控制器内部的缓冲区以形成字节块,在校验和确认无错后,这块数据将被复制到主存储器。

CRT控制器也是串行位流设备,它从主存储器中读取欲显示字符的字节,产生用来调制CRT射线的信号,将结果显示在屏幕上。控制器还产生当水平方向扫描结束后的折返信号,及当整个屏幕扫描结束后的垂直方向的折返信号。CRT正常工作之前,操作系统对它进行初始化,设置每屏行数、每行字符数。

如果没有控制器,这些复杂的操作就必须由操作系统设计人员通过编写程序来解决,而引入控制器后,只需通过传递简单的参数就可执行I/O操作,大大地简化系统的设计,有利于计算机系统对各类控制器和设备的兼容性。

综述设备控制器的功能和结构,设备控制器是CPU和设备之间的接口,它接收从CPU发来的命令,控制设备的操作,实现主存储器和设备之间的数据传输,从而使CPU从繁杂的设备操作中解放出来。设备控制器是可编址设备,当它连接多台设备时,应具有多个设备地址。设备控制器的主要功能是:

- (1) 接收和识别CPU或通道所发来的命令,例如,磁盘控制器能接收读、写、查找等各种命令;
- (2) 实现数据交换,包括设备和控制器之间的数据传输,且通过数据总线或通道,控制器和主存储器之间传输数据;
- (3) 发现和记录设备及自身的状态信息,供CPU处理使用;
- (4) 设备地址识别。

为了实现上面列举的各项功能,设备控制器必须有以下组成部分:控制寄存器及译码器,数据缓冲寄存器,状态寄存器,地址译码器,及用于对设备操作进行控制的I/O逻辑。

I/O 软件原理

5.2.1 I/O 软件的设计目标和原则

I/O软件的总体设计目标是:高效率和通用性。在改善设备效率的过程中,最应关注的是磁

盘 I/O 操作的效率。通用性意味着用统一标准的方法来管理所有设备。为了达到这些目标,把软件组织成层次结构,低层软件用来屏蔽硬件细节,高层软件向用户提供简洁、友善的界面。I/O 软件设计需要考虑的问题有:

- (1) 设备无关性:程序员编写访问文件数据的程序时,与具体的物理设备无关;
- (2) 出错处理:数据传输过程中产生的错误应该在尽可能靠近硬件的地方处理,低层软件能够解决的错误不让高层软件感知;
- (3) 同步/异步传输:CPU 在启动 I/O 操作后既可继续执行其他工作,直至中断到达,称此为异步传输;又可采用阻塞方式,让启动 I/O 操作的进程挂起等待,直至数据传输完成,称此为同步传输,I/O 软件应支持这两种工作方式;
- (4) 缓冲技术:建立数据缓冲区,让数据的到达率与离去率相匹配,以提高系统吞吐率。

为了合理、高效地解决以上问题,操作系统把 I/O 软件依次组织成 4 个层次:I/O 中断处理程序;I/O 设备驱动程序;独立于设备的 I/O 软件和用户空间的 I/O 软件。

5.2.2 I/O 中断处理程序

通常中断处理程序是设备驱动的组成部分之一,位于操作系统底层,是与硬件设备密切相关的软件,它与系统的其余部分尽可能少地发生联系。当进程请求 I/O 操作时,通常将被挂起,直到数据传输结束并产生 I/O 中断时,操作系统接管 CPU 后转向中断处理程序执行。

当设备向 CPU 提出中断请求时,CPU 响应请求并转入中断处理程序执行,通常需要做到:检查设备状态寄存器的内容,判断产生中断的原因,根据 I/O 操作的完成情况进行相应的处理;若数据传输有错,应向上层软件报告设备出错信息,实施重执行;若正常结束,应唤醒等待传输的进程,使其转换为就绪态;若有等待传输的 I/O 命令,应通知相关软件启动下一个 I/O 请求。

5.2.3 I/O 设备驱动程序

设备驱动程序包括与设备密切相关的所有代码,其任务是:把用户提交的逻辑 I/O 请求转化为物理 I/O 操作的启动和执行,如设备名转化为端口地址、逻辑记录转化为物理记录、逻辑操作转化为物理操作等。同时,监督设备是否正确执行,管理数据缓冲区,进行必要的纠错处理。

笼统地说,设备驱动程序的功能是从独立于设备的软件中接收并执行 I/O 请求。一条典型的 I/O 请求是读第 n 块,如果请求到来时驱动程序空闲,它立即执行这个请求;但如果它正在处理请求,会将新到来的请求放入 I/O 请求队列中,待当前请求完成后,再依次从 I/O 请求队列中取出 I/O 请求进行处理。

执行 I/O 请求的第一步是将其转换为更具体的形式。对于磁盘驱动程序而言,它包含:计算所请求块的物理地址、检查驱动器是否运转、检测磁头臂是否定位在正确的柱面位置等,简而言之,它必须确定需要哪些控制器命令及其执行次序,然后向控制器的设备寄存器写入这些命令和相应的参数。某些控制器一次只能接收一条命令(如 DMA),另一些控制器可接收一串命令并自动进行处理(如通道方式)。

设备驱动程序发出控制命令之后,系统有两种处理方式。在大多数情况下,执行设备驱动程序的进程需要等待命令完成,在命令开始执行后,进程阻塞自己,直到中断处理时才解除阻塞。另一种情况是执行驱动程序的进程无须阻塞,例如,有些终端的滚动屏幕只需向控制器的设备寄存器中写入几个字节,整个操作可在几微秒内完成。无论是哪种情况,在操作完成后都要检查数据传输是否有错,如果有错,返回错误状态提示信息给调用者;如果正常,驱动程序将数据传送给与设备无关的软件层;如果 I/O 请求队列中有请求在排队,则选中其中一个请求执行,否则等待下一个 I/O 请求的到来。

设备驱动程序主要包含三部分功能:

1. 设备初始化

在系统初次启动或设备传输数据时,预置设备和控制器以及通道的状态。

2. 执行设备驱动例程

负责启动设备,进行数据传输,对于具有通道的 I/O 系统,此例程还负责形成通道指令,启动通道工作。

3. 执行中断处理例程

负责处理设备和控制器及通道所发出的各种中断。

5.2.4 独立于设备的 I/O 软件

虽然设备驱动程序是设备专用的,但大部分 I/O 软件却与设备无关,独立于设备的 I/O 软件和设备驱动程序之间的界限取决于具体的系统,其基本功能是执行适用于所有设备的常用 I/O 功能,并向用户层软件提供一致性接口。

1. 设备命名和设备保护

对于操作系统而言,设备都被看做文件,它与磁盘文件一样,通过路径名来寻址,每个设备具有文件名、inode、文件所有者、权限位等属性。例如,在 UNIX/Linux 中,设备名 /dev/tty00 唯一地确定为一个特别文件设置的 inode 节点,其中包含主设备号,通过它来分配正确的终端设备驱动程序;次设备号作为参数,确定驱动程序要读写的是哪台终端设备。设备不仅具有文件名,且支持与文件相关的所有系统调用,如 open、close、read、write、stat 及 lseek 等。

设备保护需要检查用户是否有权访问所申请的设备,在多数大中型计算机系统中,用户进程对 I/O 设备的直接访问是绝对禁止的,I/O 指令定义为特权指令,通过系统调用的方式间接地供用户使用。此外,在 UNIX/Linux 中还使用一种灵活的方法,对应于 I/O 设备的特别文件采用 rwx 保护机制,以判断文件的所有者和组成员是否具有向设备发送和读取数据的权限。Windows 中的设备作为命名对象出现在文件系统中,对文件的保护规则也适用于 I/O 设备。

设备文件依赖于 inode 来实现,文件目录并不能区分文件名是代表一个磁盘文件还是设备文件,但 inode 的内容是不同的,磁盘文件的 inode 包含指向数据块的指针,而设备文件的 inode 则包含指向内核设备驱动程序的指针,用来控制设备的 I/O 操作。

2. 提供与设备无关的块尺寸

当创建文件并向其输入数据时,此文件必须被分配新的磁盘块。为了完成这种分配工作,操作系统需要为每个磁盘都配置一张记录空闲块的表或位示图,分配空闲块的算法是独立于设备的,可在高于设备驱动程序的层次处理。

不同磁盘的扇区大小可能不同,独立于设备的 I/O 软件屏蔽这一事实并向高层软件提供统一的数据块尺寸,它可将若干扇区合并为一个逻辑块,称为簇。这样,高层软件就只能和大小相同的簇交互,而不必关心物理扇区的尺寸。类似地,字符设备每次对一个字符进行操作,有些字符设备操作更大单位的数据(如 80 个字符),这类差别也必须在本层进行屏蔽。

3. 缓冲技术

数据离开设备之后,通常不能直接送达目的地,如网络上传送的数据包需要经过分析、检查才能知道置于何处。某些设备有严格的时间约束,如数字音频设备,所以,必须通过缓冲区来消除填满速率和清空速率的影响。块设备和字符设备都需要缓冲技术,可通过在主存储器建立缓冲区的方法来解决。对于块设备,硬件每次读写均以块为单位,而应用程序可按照任意大小单元处理数据,如果应用程序读写的长度和位置不恰好是完整的扇区时,也必须通过缓冲区来实现,先将数据读或写入缓冲区,再从缓冲区将数据传送给用户,或当缓冲区满时再写至磁盘。对于字符设备,当用户把数据写入系统的速度快于系统输出数据的速度,或字符设备提供数据的速度快或慢于应用程序消耗数据的速度时,也需要建立缓冲区。

通常在内核空间开辟缓冲区,数据在缓冲区中缓冲后,再在用户缓冲区和设备之间传输。缓冲涉及大量复制操作,故对 I/O 件能会产生很大影响。

4. 设备分配和状态跟踪

根据设备的物理特性,操作系统制订分配策略,可采用静态分配、动态分配和虚拟分配。一些独占型设备,如 CD-ROM,同一时刻只能被一个进程使用。当用户发出设备使用的请求时,要求操作系统检查相应设备的使用状态,并根据其忙闲状况来决定接受或拒绝这一请求。一种简单的处理方法是,直接用 OPEN 打开相应设备的特别文件来进行申请,若设备不可用,则 OPEN 失败;当进程用完独占型设备时,应该关闭并释放之。

5. 错误处理和报告

I/O 设备出错是经常发生的,错误应尽可能在接近硬件的地方加以处理,如果控制器发现错误,应该设法纠正和加以解决。如果未能处理错误,再交给设备驱动程序。多数错误是与设备紧密相关的,驱动程序知道应如何对其处理(如重试、忽略或报警),一种典型的错误是磁盘块受损导致不能读写,驱动程序在尝试若干次读写操作不成功后才会放弃。如果发现错误处理不成功,或不能处理错误时,可向独立于设备的 I/O 软件报错,此后这个错误的处理就与设备驱动程序无关。低层软件所不能处理的情况交给高层软件处理。在许多情况下,错误恢复可由低层软件透明地得到解决,而高层软件甚至不知道错误的存在。如果发现 I/O 编程出错,如无效操作码、无效设备或无效缓冲区地址,则返回一个错误码并向调用者报错;如果是在读关键系统数据结构(如磁盘位示图)时出错,则只能向操作员报告,打印出错提示信息并终止运行。诸如此类与设备

无关的错误处理应该由独立于设备的 I/O 软件完成。

5.2.5 用户空间的 I/O 软件

1. 库函数

尽管大部分 I/O 软件都包含在操作系统中,但有一小部分是与应用程序链接在一起的库函数,甚至完全由运行于用户态的程序组成,系统调用通常由库函数封装后供用户使用。如下面的 C 语言语句

```
count = write(fd, buffer, nbytes);
```

有对应的系统调用,所调用的库函数 write 将与应用程序链接在一起,形成可执行代码装入主存储器,这些库函数显然也是 I/O 系统的组成部分。

库函数所做的工作只是将系统调用时所用的参数放在合适的位置,然后执行访管指令来陷入内核,再由内核函数实现真正的 I/O 操作。I/O 数据的格式处理常常由库函数实现,以 printf 为例,它以格式串和一些可能的变量作为输入,构造一个 ASCII 字符串,然后通过 write 系统调用完成输出这个字符串的实际操作。标准函数库包含许多涉及 I/O 操作的函数,它们都是作为应用程序的一部分运行的。

2. SPOOLing 软件

并非所有的用户层 I/O 软件都由库函数构成,SPOOLing 就是在内核外运行的系统 I/O 软件,它采用预输入、缓输出和井管理技术,是多道程序设计系统中处理独占型设备的一种方法,创建守护进程和特殊目录解决独占型设备的空占问题。

如图 5.1 所示是 I/O 软件的层次及其主要功能,图中的箭头表示 I/O 控制流。各层所实现的功能已在前面叙述,现以读操作为例,总结实现一个 I/O 操作所应执行的步骤。

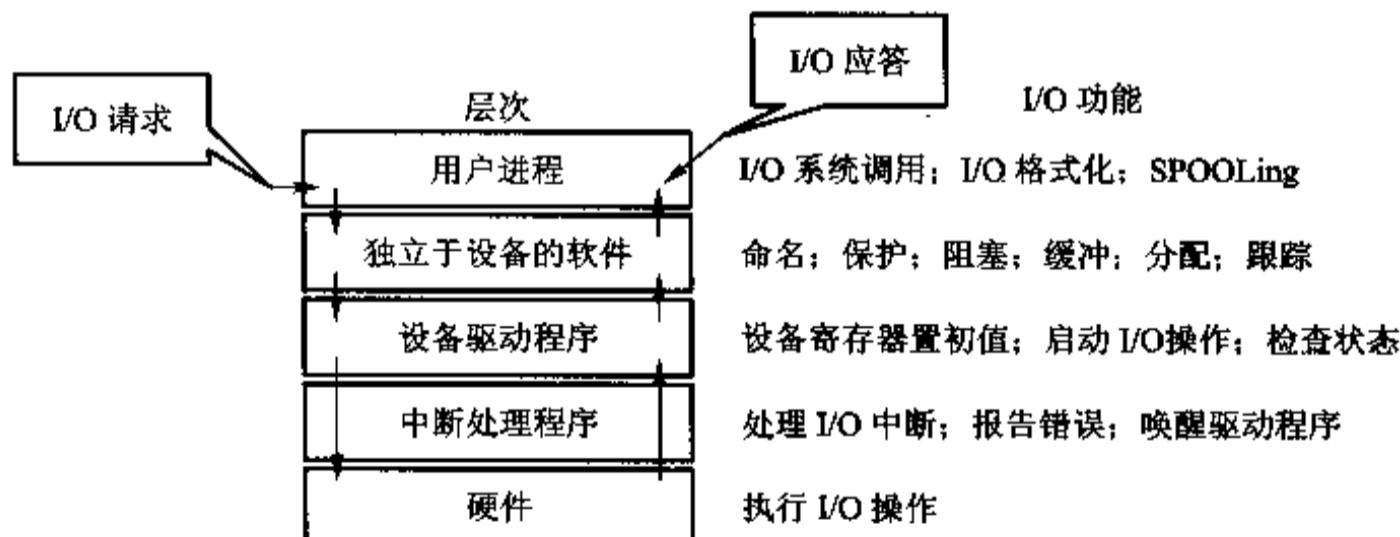


图 5.1 I/O 软件的层次及其主要功能

- (1) 应用进程对已打开文件的文件描述符执行读库函数(系统调用);
- (2) 独立设备的 I/O 软件检查参数是否正确,若正确,检查高速缓存中有无要读取的信息块;若有,从缓冲区直接读至用户区,完成 I/O 请求;

- (3) 若数据不在缓冲区,执行物理 I/O 操作,独立于设备的 I/O 软件将设备的逻辑名转换成物理名,检查对设备操作的许可权,将 I/O 请求排队,阻塞应用进程且等待 I/O 操作完成;
- (4) 内核启动设备驱动程序,分配存放读出块的缓冲区,准备接收数据,并向设备控制寄存器发送启动命令,或建立 DMA 传输,启动 I/O;
- (5) 设备控制器操作设备,执行数据传输;
- (6) 当采用 DMA 控制器控制数据传输时,一旦传输完成,硬件产生 I/O 结束中断;
- (7) CPU 响应中断,转向磁盘中断处理程序。它检查中断产生原因和设备的执行状态,若设备有错,向设备驱动程序发信号,检查是否能重复执行,如果允许,重发启动设备的命令,再次传输;否则,向上层软件报告错误。若设备 I/O 正确完成,将数据传输给指定的用户进程空间,唤醒阻塞进程并将其放入就绪队列;然后,系统检查有无 I/O 请求在排队,若有,再启动设备,继续传输。至此,中断处理完成且返回,将成功或失败的信号逐层向上报告;
- (8) 当应用进程被再次调度执行时,从 I/O 系统调用的断点处恢复执行。

具有通道的 I/O 系统

具有通道的计算机系统其 I/O 操作涉及 CPU 执行 I/O 指令、通道执行通道命令以及 CPU 和通道之间的通信。

5.3.1 通道命令和通道程序

1. 通道命令

通道又称 I/O 处理机,有自己的指令系统,常常把 I/O 处理机的指令称为通道命令。通道命令字(Channel Command Word, CCW)是通道从主存储器取出并控制 I/O 设备执行 I/O 操作的命令字,用通道命令编写的程序称为通道程序。一条通道命令能够实现一种功能,由于通道程序由多条通道命令组成,每次启动就可以完成复杂的 I/O 控制。IBM370 及后继产品 IBM S/390 均采用通道技术,IBM370 系统的通道命令为双字长,其格式如图 5.2 所示。



图 5.2 IBM370 通道命令字

通道命令字为双字长,各字段的含义介绍如下。

(1) 命令码

命令码规定设备所执行的操作。通道命令码分为三类:数据传输类(读、反读、写、取状态),通道转移类(转移),设备控制类(随设备类的不同执行不同的控制)。

(2) 数据主存地址

对于数据传输类命令,规定本条通道命令所访问的主存数据区起始或末尾地址;对于通道转

移类命令,规定转移地址。

(3) 标志码

标志码用来定义通道程序的链接方式或标志通道命令的特点,其 32 位~36 位依次为:数据链、命令链、禁发长度错、封锁读入主存、程序进程中断。32 位和 33 位均为 0 时,称为无链,表示本条通道命令是通道程序的最后一条命令;这两位为 01 时,称为命令链,表示本命令的操作已是最后一条命令,后面执行其他通道命令;32 位为 1 时,称为数据链,表示下一条通道命令将沿用本条命令码,但却未指明新的主存地址。34 位为 1 时,这条通道命令执行过程中,禁发长度错。35 位为 1 时,能使读型操作实现假读功能。36 位为 1 时,执行到这条通道命令将发出程序进程中断,将通道程序操作沿链推进的程度用中断方式通知操作系统。

(4) 传送字节个数

对于数据传输类命令,规定本次交换的字节个数;对于通道转移类命令,规定填一个非 0 数。

2. 通道程序

启动设备按指定要求工作,要编写实现指定功能的通道程序,如下所示是用汇编格式编写的通道程序例子。

```

CCW X '02',inarea,      X '40',80
CCW X '02', *,          X '50',80
CCW X '02',inarea + 80, X '40',80
CCW X '02', *,          X '50',80
CCW X '02',inarea + 160, X '40',80
inarea DS CL240

```

此通道程序把磁带上 3 个不连续的信息块读入主存储器的连续区域,其中,“*”表示不用主存地址,X '50' 表示使用“封锁读入主存”标志位。

3. 通道地址字和通道状态字

通道方式执行 I/O 操作时,要使用主存储器的两个固定存储单元:通道地址字(Channel Address Word,CAW)和通道状态字(Channel Status Word,CSW)。编写好的通道程序存放在主存储器中,为了使通道能获取通道命令去执行,在主存储器的一个固定单元中存放当前启动并要求设备执行的通道程序的首地址,以后的命令地址可由前一个地址加 8 获得,这个用来存放通道程序的首地址的单元称为通道地址字。

通道状态字是通道向操作系统报告情况的一种汇集,通道利用通道状态字可以提供通道和设备执行 I/O 操作的情况,IBM 系统中的通道状态字也采用双字表示。其中各字段的含义介绍如下。

(1) 通道命令地址

通道命令地址指向最后一条执行的通道命令地址加 8。

(2) 设备状态

设备状态是由控制器或设备所产生、记录和供给的信息,包括注意状态、状态修正位、控制器

结束、忙、通道结束、设备结束、设备出错和设备特殊等。

(3) 通道状态

通道状态是由通道所发现、记录和供给的信息，包括程序进程中断、长度错误、程序出错、存储保护错、通道数据错、通道控制错、接口操作错和链溢出等。

(4) 剩余字节个数

剩余字节个数是指最后一条通道命令执行之后还剩余多少字节未交换。

5.3.2 I/O 指令和主机 I/O 程序

IBM 系统主机提供一组 I/O 指令，以便完成 I/O 操作。I/O 指令有：启动 I/O、查询 I/O、查询通道、停止 I/O 和停止设备，它们都是特权指令，以防用户擅自使用而引起 I/O 操作错误。例如 SIO X '00E'

将启动 0 号通道、0E 号设备工作，根据系统约定可以把通道程序的首地址存放在主存储器的 CAW 中。当 CPU 执行 I/O 操作时，只是简单地将 I/O 指令发给通道即可，SIO 指令发出后，如果条件码为 0，表示设备已成功启动，通道从 CAW 获取通道程序首地址以便开始工作，CPU 可以返回执行计算任务；如果条件码为 1，表示设备启动不成功，通道有情况要报告，对此要进一步检查通道状态字 CSW；如果条件码为 2，表示通道或设备忙，启动不成功，本指令执行结束；如果条件码为 3，表示指定通道或设备已断开，启动不成功，本指令执行结束。

每次执行 I/O 操作都要为通道编制通道程序，要为主机编制主机 I/O 程序，CPU 执行 I/O 指令时，将首地址放在 CAW 中的通道程序交给通道，通道将根据 CPU 发来的 I/O 指令和通道程序对设备进行控制。正确执行 I/O 操作的步骤可以归纳如下：确定 I/O 任务，了解使用何种设备、属于哪个通道、操作方法如何等；确定算法，决定对于例外情况的处理方法；编写通道程序，完成相应的 I/O 操作；编写主机 I/O 程序，对不同的条件码进行不同的处理。

5.3.3 通道启动和 I/O 操作过程

CPU 是主设备，通道是从设备，CPU 和通道之间存在主从关系，需要相互配合才能完成 I/O 操作。那么，CPU 如何通知通道应该做什么？CPU 又如何了解通道的工作情况呢？通道方式的 I/O 操作过程可分成以下 3 个阶段。

1. I/O 启动阶段

用户在 I/O 主程序中调用文件操作请求传输信息，文件系统根据用户给予的参数可以确定设备、传输信息的位置、传送字节个数和信息主存区地址。然后，把存取要求通知设备管理，设备管理按照规定组织通道程序并将首地址放入 CAW。CPU 执行 I/O 主程序，向通道发出 SIO 以命令通道工作，通道根据自身的状态形成条件码作为回答，若通道可用，则 I/O 操作开始。这一通信过程发生在操作开始时期，CPU 根据条件码便可决定转移的方向。

2. I/O 操作阶段

I/O 启动成功后，通道访问 CAW 并获取第一条通道命令，通道开始执行通道程序，同时将

I/O 地址传送给控制器,向其发出读、写或控制命令,控制设备进行数据传输。控制器接收通道发来的命令后,检查设备的状态,若设备不忙,则告知通道释放 CPU,并开始 I/O 操作,向设备发出动作序列,设备执行相应的动作;之后,通道独立执行通道程序中的各条通道命令字,直到通道程序执行结束。本次 I/O 操作结束之后,通道向 CPU 发出 I/O 操作结束中断,再次请求 CPU 的干预。

3. I/O 结束阶段

通道发现 I/O 系统中出现通道结束、控制器结束、设备结束或其他能产生中断的信号时,就向 CPU 申请 I/O 中断。同时,把产生中断的通道号和设备号以及通道状态字存入主存固定单元,CPU 响应 I/O 中断后,暂停现行程序的执行,转向 I/O 中断处理程序。

缓冲技术

为了解决 CPU 与设备之间速度不匹配的矛盾,及协调逻辑记录大小与物理记录大小不一致的问题,提高 CPU 和设备的并行性,减少 I/O 操作对 CPU 的中断次数,放宽对 CPU 中断响应时间的要求,操作系统普遍采用缓冲技术。缓冲用于平滑两种不同速度的硬部件或设备之间的信息传输,在主存储器中开辟一个存储区,称为缓冲区,专门用于临时存放 I/O 操作的数据。

实现缓冲技术的基本思想如下:当进程执行写操作输出数据时,先向系统申请一个输出缓冲区,然后,将数据送至缓冲区,若是顺序写请求,则不断地把数据填入缓冲区,直到装满为止,此后,进程可以继续计算,同时,系统将缓冲区的内容写到设备上。当进程执行读操作输入数据时,先向系统申请一个输入缓冲区,系统将设备上的一条物理记录读至缓冲区,然后根据要求,把当前所需要的逻辑记录从缓冲区中选出并传送给进程。

用于上述目的的专用主存区称为 I/O 缓冲区。在输出数据时,只有在系统来不及腾空缓冲区而进程又要写数据时,它才需要等待;在输入数据时,仅当缓冲区为空而进程又要从中读取数据时,它才被迫等待。其他时间可进一步提高 CPU 和设备的并行性,设备和设备之间的并行性,从而提高系统效率。在操作系统的管理下,常辟出缓冲区以服务于各种设备,常用的缓冲技术有单缓冲、双缓冲和多缓冲。

5.4.1 单缓冲

单缓冲是最简单的缓冲技术,每当应用进程发出 I/O 请求时,操作系统在主存储器的系统区中开设一个缓冲区。

对于块设备输入,单缓冲机制的工作过程是:先从磁盘把一块数据读至缓冲区,假设花费时间 T ;接着,系统把缓冲区中的数据送到用户区,设所消耗的时间为 M ,由于此时缓冲区已空,系统可预读紧接着的下一块,大多数应用将使用邻接块,然后应用进程对这批数据进行计算,共耗时 C 。若不采用缓冲技术,数据直接从磁盘传送到用户区,每批数据处理时间约为 $T + C$,而采用单缓冲,每批数据处理时间为 $\max(C, T) + M$,通常 M 远远小于 C 或 T ,故速度会快很多。

对于块设备输出,单缓冲机制的工作方式类似,先把数据从用户区复制到系统缓冲区,应用进程可继续请求输出,直到缓冲区填满,由系统写到磁盘上。对于字符设备输入,缓冲区用于暂存用户输入的一行数据,在输入时,应用进程挂起,等待一行数据输入完毕;在输出时,应用进程将第一行数据送入缓冲区后继续执行,如果在第一个输出操作未腾空缓冲区之前,又有第二行数据要输出,应用进程等待。如果希望实现 I/O 的并行工作,如把输入设备中的数据输入并加工,再从输出设备上输出,必须引入双缓冲技术。

5.4.2 双缓冲

为了加快 I/O 操作的执行速度,实现 I/O 的并行工作和提高设备利用率,需要引入双缓冲。在输入数据时,首先从设备读出数据填充缓冲区 1,系统从缓冲区 1 把数据传送到用户区,应用进程便可对数据进行加工和计算;与此同时,从设备读出数据填充缓冲区 2。当缓冲区 1 为空时,再次从设备读出数据到缓冲区 1,系统又可以把缓冲区 2 的数据传送到用户区,应用进程开始加工缓冲区 2 中的数据。两个缓冲区交替使用,使 CPU 和设备、设备和设备之间的并行性进一步提高,仅当两个缓冲区都为空且进程还要提取数据时,它才被迫等待。粗略地估计传输和处理一块的时间,如果 $C < T$,输入操作比计算操作速度慢,这时由于 M 远远小于 T ,故将磁盘上的一块数据传送到一个缓冲区期间(所花费时间为 T),计算机已完成将另一个缓冲区中的数据传送到用户区并对这块数据进行计算的工作,所以,一块数据的传输和处理时间为 T ,即 $\max(C, T)$,显然,这种情况可保证块设备连续工作;如果 $C > T$,计算操作比输入操作速度慢,每当上一块数据计算完毕后,需要把一个缓冲区中的数据传送到用户区,所花费时间为 M ,再对这块数据进行计算,所花费时间为 C ,所以,一块数据的传输和处理时间为 $C + M$,即 $\max(C, T) + M$,显然,这种情况使得进程不必等待 I/O 操作。双缓冲技术使系统效率提高,但复杂性也随之增加了。

采用双缓冲读卡及打印可这样进行:第一张卡片读入缓冲区 1,在打印缓冲区 1 中数据的同时,又把第二张卡片读入缓冲区 2。缓冲区 1 打印完毕时,缓冲区 2 也刚好输入完毕,让读卡机和打印机交换缓冲。这样,I/O 设备就能够处于并行工作的状态。

5.4.3 多缓冲

采用双缓冲技术虽然能提高设备的并行工作程度,但在设备和处理进程速度不匹配的情况下仍不十分理想。举例来说,若输入设备的速度快于进程消耗数据的速度,则输入设备很快就会把两个缓冲区填满;反之,若进程处理数据的速度快于数据输入速度,很快会把两个缓冲区抽空,造成进程经常处于等待态。为了改善这种情况,获得较高的并行度,常常采用多缓冲所组成的循环缓冲技术。

操作系统从主存区域中分配一组缓冲区,每个缓冲区都有一个链接指针指向下一个缓冲区,最后一个缓冲区指针指向第一个缓冲区,组成循环缓冲,缓冲区的大小等于物理记录的大小,多缓冲的缓冲区是系统的公共资源,可供进程共享,并由系统统一分配和管理。缓冲区用途可分为输入缓冲区、处理缓冲区和输出缓冲区。为了管理各类缓冲区,执行各种操作,必须设计专门的

软件,这就是缓冲区自动管理系统。

5.4.4 缓冲区高速缓存

终端、打印机等字符型设备的缓冲区采用先进先出方式工作得很好,也就是说数据以其产生的先后顺序依次被使用。对于磁盘类的直接存取型设备,通常需要按照应用程序的要求随机地访问数据块,同一数据块可能会被多次访问,为了减少访问磁盘的次数,避免数据项的重复产生,内核建立一个数据缓冲区高速缓存(data buffer cache),专门用于保存最近使用过的磁盘数据块。数据缓冲区高速缓存位于文件系统和磁盘设备驱动程序之间,不要把它与硬件联想存储器的快表相混淆。Linux 2.4 内核版本起已取消缓冲区高速缓存,只剩唯一的磁盘缓存——页高速缓存。每当应用进程打开、关闭或撤销文件时,激活高速缓存管理程序,下面讨论数据缓冲区高速缓存的实现思想。

(1) 当请求从指定文件读写数据时,给定设备号和盘块号(能唯一标明是哪个文件系统的哪个数据块),必须快速查询所需数据块是否在高速缓存中,如果在高速缓存中,是在哪个缓冲区并获得其内容。为此,可设计散列表来访问缓冲区,散列数组的每个元素指向一个缓冲区链表,把具有相同散列值的盘块链接在一起。高速缓存中的每个缓冲区链表都有一个缓冲控制块,包含如下信息:逻辑设备号、盘块号、高速缓存虚拟地址、数据所属文件的文件描述符、数据在文件内的位移、缓冲区链接指针、空闲表缓冲区指针、活动计数(有多少读写操作在访问此块)、状态字等。如果数据块不在高速缓存中,则需要从磁盘读取数据,并将其缓冲起来,系统采用合适的算法把尽可能多的数据保存在缓冲区高速缓存。

类似地,向磁盘写入的数据也暂存于数据缓冲区高速缓存中,以供回写磁盘之前再次使用。内核还会判定数据是否需要回写,或数据是否很快就要被回写,尽量采用延迟写来减少 I/O 操作的次数。

(2) 当文件关闭或撤销时,需要解决释放的缓冲区的重用,可采用一定的策略(如 LRU)把单独的缓冲区链接在一起,于是,最不可能被再次访问的缓冲区将被最先释放重用。

(3) 提供一组高速缓存操作,为文件驱动程序实现读写文件数据。这些操作通常有:写缓存、延迟写缓存、读缓存、预读缓存等。

驱动调度技术

作为操作系统的辅助存储器,用来存放文件的磁盘是一类高速大容量旋转型存储设备,在繁重的 I/O 负载下,同时会有若干传输请求来到并等待处理,系统必须采用一种调度策略,能够按最佳次序执行要求访问的诸多请求,这叫做驱动调度,所使用的算法叫做驱动调度算法。驱动调度能减少为若干 I/O 请求服务所需消耗的总时间,从而提高系统效率。除了 I/O 请求的优化排序外,信息在磁盘上的排列方式、存储空间的分配方法都能影响存取访问速度。本节首先介绍直接存取存储设备的结构,然后针对磁盘来讨论与驱动调度有关的技术。

5.5.1 存储设备的物理结构

磁盘是一种直接存取存储设备,又称随机存取存储设备,它的每条物理记录都有确定的位置和唯一的地址。磁盘结构如图 5.3 所示,包括多个盘面用于存储数据,每个盘面有一个读写磁头,所有读写磁头都固定在唯一的移动臂上同时移动。一个盘面上的读写磁头的轨迹称为磁道,读写磁头下的所有磁道所组成的圆柱体称为柱面,一个磁道又可划分成多个扇区(物理块)。

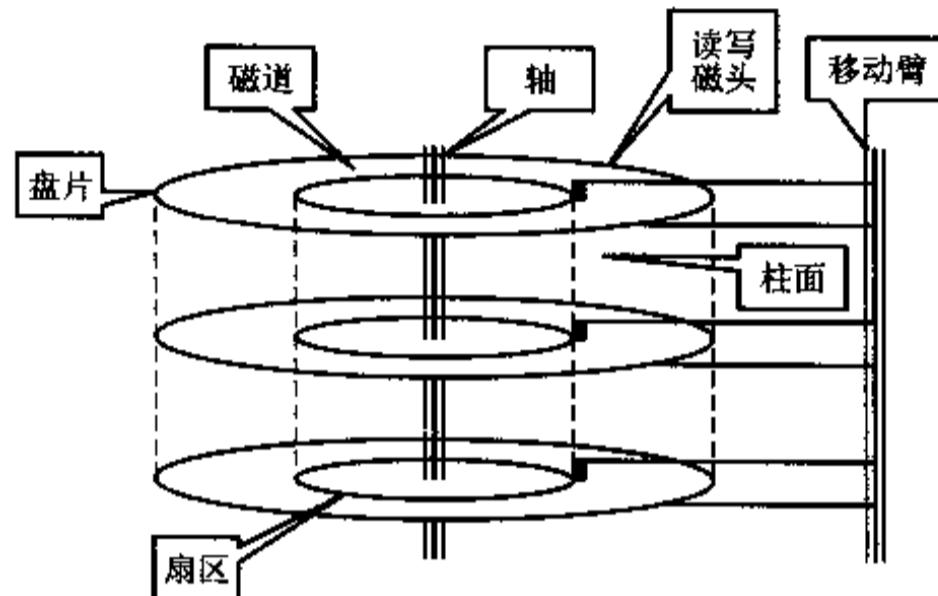


图 5.3 磁盘结构

文件信息通常并不记录在同一盘面的各个磁道上,而是记录在同一柱面的不同磁道上,这样可减少移动臂的移动次数,缩短存取信息的时间。为了访问磁盘上的一条物理记录,必须给出 3 个参数:柱面号、磁头号、扇区号。磁盘机根据柱面号控制移动臂作横向机械移动,带动读写磁头到达指定柱面,这个动作的执行速度较慢,称为“查找时间”;下一步选择磁头号,然后等待被访问的扇区旋转到读写磁头下时,按扇区号存取信息,称为“搜索延迟”,实现磁盘机操作的命令有查找、搜索、转移和读写等。

5.5.2 循环排序

旋转型存储设备上不同物理块的存取时间存在明显的差别,为了缩短延迟时间,I/O 请求的排序方法有实际意义。考虑磁道上保存 4 个物理块的旋转型设备,旋转一周耗时 20 ms,假定收到以下 4 个 I/O 请求。

请求次序	物理块
1	读块 4
2	读块 3
3	读块 2
4	读块 1

对这些 I/O 请求可以有多种排序方法。

方法 1: 按照 I/O 请求的次序读块 4、3、2、1, 假定平均花费 $1/2$ 周定位, 再加上 $1/4$ 周读出, 由于当读出块 4 后需要转过 $3/4$ 周才能读块 3, 总处理时间为 $1/2 + 1/4 + 3 \times 3/4 = 3$ 周, 即 60 ms 。

方法 2: 按照 I/O 请求的次序读块 1、2、3、4, 那么, 总处理时间为 $1/2 + 1/4 + 3 \times 1/4 = 1.5$ 周, 即 30 ms 。

方法 3: 如果已知当前读位置是记录 3, 处理次序为读块 4、1、2、3 会更好, 总处理时间为 1 周, 即 20 ms 。

为了实现方法 3, 驱动调度算法必须知道旋转型设备的当前位置, 称作旋转位置测定。如果未设置这种硬件, 那么因无法测定磁头的当前位置会平均多花费半周时间。

5.5.3 优化分布

信息在存储空间中的排列方式会影响存取等待时间。考虑 10 条逻辑记录 A, B, …, J 被存放于旋转型设备上, 每道存放 10 个物理块, 安排如图 5.4(a)所示。

物理块	逻辑记录	物理块	逻辑记录
1	A	1	A
2	B	2	H
3	C	3	E
4	D	4	B
5	E	5	I
6	F	6	F
7	G	7	C
8	H	8	J
9	I	9	G
10	J	10	D

(a)

(b)

图 5.4 优化分布示例

(a) 优化前 (b) 优化后

假定需要经常顺序处理这些记录, 旋转速度为一周 20 ms , 处理程序读出每块后花 4 ms 进行处理, 由于读出并处理记录 A 之后将旋转到记录 D 的开始, 为了读出记录 B, 必须再转一周。于是, 处理 10 条记录的总时间为 10 ms (旋转到记录 A 的平均时间) + 2 ms (读记录 A) + 4 ms (处理记录 A) + $9 \times [16\text{ ms}(访问下一条记录) + 2\text{ ms}(读记录) + 4\text{ ms}(处理记录)] = 214\text{ ms}$ 。

按照图 5.4(b)所示方式对信息优化分布: 当读出记录 A 并处理结束后, 恰巧旋转至记录 B 的位置, 立即就可读出记录 B 并处理。按照这一方案, 处理 10 条记录的总时间为 10 ms (旋转到记录 A 的平均时间) + $10 \times [2\text{ ms}(读记录) + 4\text{ ms}(处理记录)] = 70\text{ ms}$, 所花费的时间是原方案

的 1/3。如果有众多记录需要处理,所节省的时间更加可观。

5.5.4 搜索定位

对于磁盘,除了旋转延迟之外,还有搜索寻道的延迟。I/O 请求需要 3 个参数:柱面号、磁道号和物理块号。例如,对磁盘依次有 5 个访问请求如下所示。

柱面号	磁道号	物理块号
7	4	1
7	4	8
7	4	5
40	6	4
2	7	7

如果采用 FCFS 算法,假设移动臂当前处于 0 号柱面,按照上述次序访问磁盘,移动臂将从 0 号柱面移至 7 号柱面,再移至 40 号柱面,然后,回到 2 号柱面,显然这样移动移动臂很不合理。如果将访问请求按照柱面号 2、7、7、7、40 的次序进行处理,将节省很多移动时间。再考查 7 号柱面上的 3 个访问,按照上述次序,必须使磁盘旋转近两圈才能访问完毕。若再次将访问请求排序,按照 7-4-1、7-4-5 和 7-4-8 执行,显然,对 7 号柱面的 3 次访问大约只需旋转一圈或更少就能完毕。可见,对于磁盘设备,在启动之前按驱动调度策略对访问的请求优化其排序十分必要。除了使旋转圈数达到最少的调度策略外,还应考虑使移动臂的移动时间最短的调度策略。

移动臂调度有若干算法,下面介绍其中的几种。

1. “先来先服务”算法

在“先来先服务”(First-Come-First-Served, FCFS)算法中,磁盘臂是随机移动的,不考虑各 I/O 请求之间的相对次序和移动臂当前所处的位置,进程等待 I/O 请求的时间会很长,寻道性能较差。

2. “电梯调度”算法

“电梯调度”算法(elevator algorithm)如图 5.5 所示,磁盘柱面号通常由外向里递增,磁头越向外,所处的柱面号越小,反之越大。每次总是选择沿移动臂的移动方向最近的那个柱面,若同一柱面上有多个请求,还需进行旋转优化。如果当前移动方向上没有访问请求时,就改变移动臂的移动方向,然后,处理所遇到的最近的 I/O 请求,这十分类似于电梯的调度规则。每当要求访问磁盘时,操作系统查看磁盘机是否空闲,如果空闲就立即移臂,将当前移动方向和本次停留位置都记录下来;如果非空,就让请求者等待并把记录的访问位置按照既定的调度算法对全体等待者进行寻查定序优化。

3. “最短查找时间优先”算法

“最短查找时间优先”算法(shortest seek time first algorithm)考虑 I/O 请求之间的区别,总是先执行查找时间最短的请求,与 FCFS 算法相比有较好的寻道性能。上例中,采用 FCFS 算法磁

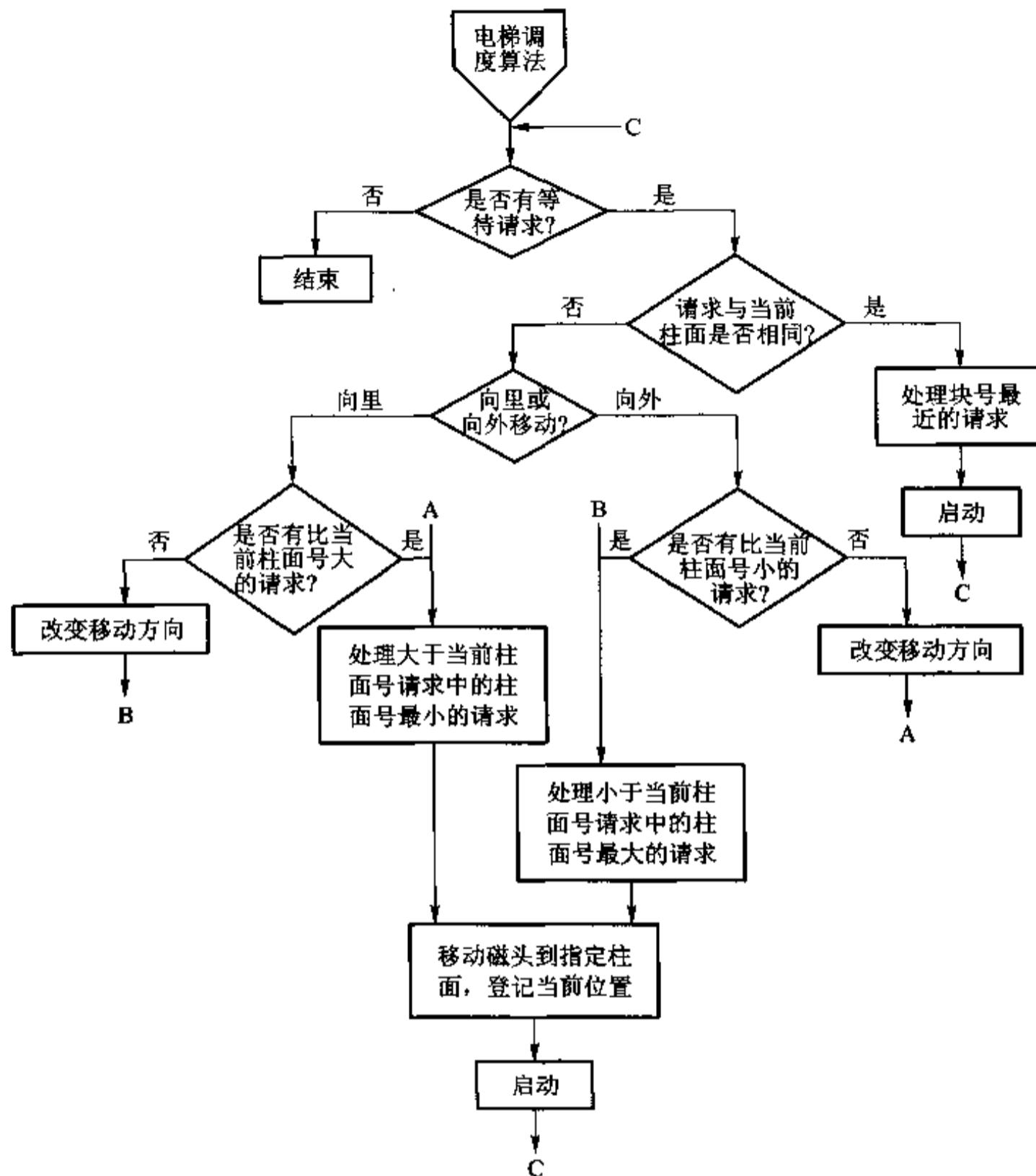


图 5.5 电梯调度算法

头共移动 78 个柱面,采用“最短查找时间优先”算法磁头共移动 40 个柱面,节省将近一半的移动臂时间。此算法存在“饥饿”现象,随着靠近当前磁头位置读写请求的不断到来,使得到来时间早但距离当前磁头位置较远的读写请求服务被无限期地推迟。

4. “扫描”算法

在“扫描”算法(scan algorithm)中,移动臂每次沿一个方向移动,扫过所有柱面,遇到最近的 I/O 请求便进行处理,直至到达最后一个柱面后,再向相反的方向移动回来。如果一个请求到达时其所访问的柱面恰巧在磁头移动的前方,这个请求立即获得服务;反之如果恰好在磁头移动的后面,则要等到磁头调转方向后才加以处理。“扫描”算法与“电梯调度”算法的不同之处在于:即

使当前移动方向暂时没有 I/O 请求,移动臂也需要扫描到头。“最短查找时间优先”算法虽然具有较好的寻道性能,但可能会造成进程“饥饿”,“扫描”算法能克服这一缺点。“扫描”算法偏爱那些最接近里面或靠近外面的请求,对最近扫描所跨越区域的 I/O 请求的响应速度会较慢。

5. “分步扫描”算法

在“分步扫描”算法(N-steps scan algorithm)中,考虑进程重复请求同一磁道会垄断整个设备,造成“磁头臂的粘性”,采用分步扫描可以避免这类问题。将 I/O 请求分组,每组不超过 N 个请求,每次选择一个组进行扫描,处理完一组后再选择下一组,这种调度算法能保证每个存取请求的等待时间不致太长。当 N 值很大时,接近于“扫描”算法的性能;当 N=1 时,接近于 FCFS 算法的性能。

6. “循环扫描”算法

在磁盘请求对柱面的分布是均匀的情况下,当移动臂移动到头并转向时,靠近磁头一端的请求特别少,许多请求集中分布在远离磁头的一端,它们的等待时间就会较长,“循环扫描”算法(circular scan algorithm)能克服这个缺点,这是为适应不断有大量柱面均匀分布的存取请求进入系统的情况而设计的扫描方式。移动臂总是从 0 号柱面至最大号柱面顺序扫描,然后,直接返回 0 号柱面重复进行,归途中不再提供服务,构成一个循环,缩短处理新来请求的最大延迟。在一个柱面上,移动臂停留至磁盘旋转一定的圈数,再移向下一个柱面。这样能够缩短刚离开的柱面上又到达的大量 I/O 请求的等待时间。为了在磁盘转动的每一圈时间内执行更多的存取,必须考虑旋转优化问题。

上面讨论的驱动调度算法能减少 I/O 请求的总时间,但都是以增加处理器时间为代价的,排队技术并不适用于所有场合,这些算法的价值依赖于处理器的速度和 I/O 请求的数量。如果 I/O 请求较少,采用多道程序设计就能够达到较高的吞吐量;如果处理器速度很慢,处理器的开销可能折损这些调度算法所带来的好处。

7. Linux 磁盘调度算法

在 Linux 2.4 中,默认磁盘调度采用“电梯调度”算法,当磁盘的某些区域有繁重的操作负载时,会导致远离此区域的请求不能及时得到处理,甚至出现“饥饿”现象。为此,Linux 2.6 增加两种新的磁盘调度算法:最终期限调度算法和预期调度算法,尽力确保处理期限已到达的请求获得响应。

“电梯调度”算法的基本原理已经介绍过,Linux“电梯调度”算法把磁盘读写请求保持在一个队列中,且对请求按照块号执行排序和合并功能,当有新的请求添加到队列中时,会依次考虑下列操作。

- (1) 如果新请求与队列中等待请求的数据处于同一磁盘扇区或者直接相邻的扇区,则现有请求和新请求合并成一个请求。
- (2) 如果队列中的请求已经存在了很长的时间,则新请求将被插入队列尾部。
- (3) 如果存在合适的位置,则新请求将按顺序插入队列中;如果没有合适的位置,则新请求将被插入队列尾部。

“电梯调度”算法有两个问题,一是出于队列动态更新的原因,一个相距较远的请求可能会延

迟相当长的时间；二是由于写请求通常是异步的，而读请求大部分是同步操作，这样一来，在写一个大文件时，很可能将一个读请求堵塞很长时间，从而堵塞进程。

为了克服这些问题，引入时限调度算法，它使用 3 个队列：读 FIFO 队列、写 FIFO 队列和电梯排序队列。每个新请求被放置到电梯排序队列中，此队列与前面所述一致，此外，同样的请求还被放置在 FIFO 读队列（如果是读请求）或 FIFO 写队列（如果是写请求）中，这样，读和写队列维护一个按请求发生时间排序的请求列表。每个请求都有一个到期时间，对于读请求其默认值为 0.5 s，对于写请求其默认值为 5 s。通常，调度器从排序队列中分派服务，当一个请求得到满足时，将其从电梯排序队列头部移走，同时也从对应的 FIFO 队列移走。然而，当 FIFO 队列头部的请求超过其到期时间时，调度程序将从此 FIFO 队列中派遣任务，取出到期请求，再加上接下来的几个队列中的请求。当然，任何请求被服务时，也将从电梯排序队列中移出它。所以，时限调度算法能够克服“饥饿”和读写不一致的问题。

当存在很多同步读请求时，上述策略可能达不到预期的效果。典型地，应用程序会在一个读请求得到满足且数据可用后，才会发出下一个读请求，在接收上一次读请求的数据和发出下一次读请求之间有很短的延迟，利用这个延迟，调度程序可转去服务其他等待的请求。由于局部性原理，相同进程的连续读请求会发生在相邻的磁盘块上，如果调度程序在满足一个读请求后能延迟一小段时间，检查是否有新的邻近的读请求发生，则可提高整个系统的性能，这就是预期调度算法的原理。

预期调度是对时限调度的一种补充，当一个读请求被分派时，预期调度程序会将调度程序的执行延迟若干毫秒（取决于配置文件），在延迟时段中，发出前一条读请求的应用程序有机会发出后继的读请求，且发生在相同的磁盘区域。如果确实是这样，新请求会立刻获得服务；如果没有新请求发生，则调度程序继续使用时限调度算法。

5.5.5 独立磁盘冗余阵列

独立磁盘冗余阵列（Redundant Array of Independent Disks, RAID）的概念在 1987 年由美国加利福尼亚大学提出，已得到业界的认可，被广泛应用于大中型计算机和计算机网络系统中，其目的是：增加容错性，获得高性能。通过在不同的磁盘上维护冗余数据来增加容错性；通过把数据分布到多个磁盘，采用并行存取以加快数据传输速率来获得高性能。其策略是：用一组容量较小的、独立的、可并行工作的磁盘组成阵列来代替单一的大容量磁盘，再加入冗余技术，数据能以多种方式组织和分布存储，于是，独立的 I/O 请求能够被并行处理，分布式的单个 I/O 请求也能并行地从多个磁盘同时得到处理，从而增加容错性，改进 I/O 性能。

在基本的组织方式中，有多种类型的 RAID，其主要差别在于冗余信息数量（增加磁盘的数量）和容错性级别（可纠正错位的数目），及冗余信息是集中在一个磁盘上还是分散在多个磁盘上。具体类型列举如下。

（1）RAID0：连续的数据条带（每个条带可规定为 1 个或多个扇区）以轮转方式写到全部的磁盘上，然后，采用并行交叉方式存取，缩短 I/O 请求的排队时间，适用于大数据量的 I/O 请求，

但并无冗余校验功能,可靠性较差。

(2) RAID1:采用镜像盘备份所有数据来提高容错性,读请求拥有最短寻道时间,写请求可并行完成。其缺点是容量下降一半,故成本很高。

(3) RAID2:采用数据字或字节方式交叉存放,并行存取以获得高性能,使用海明校验码,适合大量顺序数据的访问。由于使用多个冗余盘,其成本较高。

(4) RAID3:是 RAID2 的简化版本,其差别是它仅使用一只冗余盘,采用奇偶校验技术。

(5) RAID4:采用独立存取磁盘阵列,数据条带交叉存放,访问请求可并行地获得满足,适合要求较高 I/O 请求速度的应用场合,使用一只冗余盘存放奇偶校验码。

(6) RAID5:与 RAID4 的组织方式类似,但奇偶校验码循环分布在每个盘上,使得容错性更好。

(7) RAID6:采用双重冗余技术,即使有两个数据磁盘发生错误,也可以重新生成数据。

5.5.6 提高磁盘 I/O 速度的方法

通常为磁盘设置数据缓冲区高速缓存,这样能够显著缩短等待磁盘 I/O 的时间。下面所介绍的一些方法也能有效地提高磁盘 I/O 速度。

1. 提前读

对于采用顺序方式所访问的文件数据,在读当前盘块时已知下次要读出的盘块地址,因此,可在读当前盘块的同时,提前把下一盘块的数据也读入磁盘缓冲区。这样一来,当下次要读盘块中的那些数据时,由于已经提前把它们读入缓冲区,便可直接使用,从而缩短读数据的时间,相当于提高磁盘 I/O 速度。“提前读”功能已被许多操作系统如 UNIX、OS/2、Windows 所采用。

2. 延迟写

在执行写操作时,磁盘缓冲区中的数据本应立即写回磁盘,但考虑到此缓冲区中的数据不久之后会再次被进程访问,因此,并不立即将缓冲区中的数据写盘,而是把它挂在空闲缓冲区队列的末尾。随着空闲缓冲区的使用,存有输出数据的缓冲区不停地向队列头移动,直至移动到空闲缓冲区队列之首,当再有进程申请缓冲区且分配到此缓冲区时,才把其中的数据写到磁盘上,于是这个缓冲区可作为空闲缓冲区分配。只要存有输出数据的缓冲区还在队列中,任何对此数据的访问均可直接从中找回,不必再去访问磁盘,这样做可以减少磁盘 I/O 的次数。同样,在 UNIX、OS/2 和 Windows 系统中也都采用这一技术。

UNIX/Linux 提供两种读盘方式和三种写盘方式。“正常读”是指把磁盘块信息读入主存缓冲区;“提前读”是指在读磁盘当前块时,把下一磁盘块也读入主存缓冲区;“正常写”是指把主存缓冲区中的信息写至磁盘块,且写进程应等待写操作完成;“异步写”是指写进程无须等待写盘结束就可返回工作;“延迟写”是指仅在缓冲区头部设置“延迟写”标志,然后,释放此缓冲区并将其链入空闲缓冲区链表的尾部,当其他进程申请到此缓冲区时,才真正把缓冲区信息写回磁盘块。

3. 虚拟盘

虚拟盘是指用主存空间去仿真磁盘,又叫 RAM 盘。虚拟盘的设备驱动程序可接收所有标

准的磁盘操作,但这些操作的执行不是在磁盘上而是在主存中,操作过程对用户而言是透明的,并不会发现这与真正的磁盘操作有何不同,而仅仅是更快一些。虚拟盘是易失性存储器,一旦系统或电源发生故障,保存在虚拟盘中的数据会全部丢失,因此,虚拟盘常用于存放临时文件。虚拟盘与数据缓冲区高速缓存之间的主要区别在于:前者的内容完全由用户控制,而后者的内容是由操作系统控制的。

设备分配

5.6.1 设备独立性

计算机系统常常配置许多类型的外部设备,同类设备又可能有多台,作业在执行之前,应对静态分配的设备提出申请要求。如果申请时指定某台具体的物理设备,那么,分配工作就变得很简单,但当所指定的某台设备存在故障时,就不能满足申请要求,此作业也就不能投入运行。例如,系统拥有 A、B 两台卡片机,现有作业 J2 申请一台卡片机,如果它指定使用卡片机 A,作业 J1 已经占用 A 或者 A 坏了,虽然系统还有同类设备 B 是好的且未被占用,但也不能接受作业 J2,这样做显然很不合理。为了解决这一问题,用户通常不指定物理设备,而是指定逻辑设备,使得用户作业和物理设备分离开来,再通过其他途径建立逻辑设备和物理设备之间的映射,设备的这种特性称为“设备独立性”。在具有设备独立性的系统中,用户编写的程序可访问任何设备而无须事先指定物理设备号,即程序中所指定的设备与物理设备无关,逻辑设备名是用户命名(号)的,是可更改的,物理设备名(号)是系统规定的,是不可更改的。设备管理的功能之一就是把逻辑设备名(号)转换成物理设备名(号),为此,系统需要提供逻辑设备名和物理设备名(设备地址)的对照表以供转换使用。

设备独立性所带来的好处是:应用程序与具体的物理设备无关,系统增减或变更设备时对源程序不必加以任何修改;易于应对 I/O 设备故障,例如,某台打印机发生故障时,可用另一台打印机替换,甚至可用磁带机或磁盘机等不同类型的设备代替,从而提高系统的可靠性,增加设备分配的灵活性,能更有效地利用设备资源,实现多道程序设计。

操作系统提供设备独立性后,用户可以利用逻辑设备进行 I/O 操作,逻辑设备与物理设备之间的转换通常由系统命令或语言来实现。由于操作系统的大小和功能不同,逻辑设备到物理设备的转换过程就存在差别,通常使用的方法有:利用作业控制语言实理批处理作业的设备转换;利用操作命令实现交互型作业的设备转换;利用高级语言实现设备转换。

5.6.2 设备分配和设备分配数据结构

1. 设备分配方式

计算机系统可同时承担若干用户的多个计算任务,设备管理的功能之一就是为计算机系统所接纳的每个计算任务分配所需的外部设备。从设备的物理特性来看,可分为独占型设备、共享

设备和虚拟设备这三类,相应的管理和分配设备的技术就分为静态分配、动态分配和虚拟分配。

有些外部设备,如卡片机、打印机、磁带机等,往往只能由一个作业以独占方式使用,这是由这类设备的物理特性决定的,例如,用户在一台分配给他的卡片机上装一叠卡片,卡片上存放这位用户的作业要处理的数据,作业执行过程中将随机地读入卡片上的数据,对其进行加工处理,因此,不可能在此作业暂时不使用卡片机时,人为地换上另一作业的一叠卡片。只有当某作业归还卡片机之后,才能让另一作业去占用它。

对于独占型设备,往往采用静态分配,即在作业执行之前,将所要使用的设备全部分配给它,当作业在执行过程中不再需要使用这类设备或作业执行结束将要撤离时,再收回设备。静态分配实现起来简单,能够防止系统发生死锁,但会降低设备利用率。例如,对打印机采用静态分配,在作业执行前分配,但是直到作业产生结果时才使用打印机,这样,尽管这台打印机在大部分时间处于空闲状态,但是,其他作业却不能使用它。

如果对打印机采用动态分配,在作业执行过程中要求输出一批信息时,系统才把打印机分配给作业,当一个文件输出完毕要关闭时,系统就收回分配给此作业的打印机。采用动态分配后,在打印机上可能依次输出若干作业的信息,由于输出信息以文件为单位,每个文件的头和尾均设有标志,如用户名、作业名、文件名等,操作员很容易辨认输出信息属于哪个用户,所以,对于某些以独占方式使用的设备,采用动态分配方式,不仅是可行的而且也能提高设备利用率。

另一类设备,如磁盘,往往可让多个作业同时使用,这是因为这类设备的存储容量大、存取速度快且可直接访问。例如,每个作业的信息组织成文件存放在磁盘上,使用信息时可按名称查找文件,并从磁盘上读出,当用户提出存取信息的要求时,总是先由文件管理进行处理,确定信息的存放位置,再向设备管理提出 I/O 请求。所以,对于这类设备,设备管理的主要工作是驱动调度和实施驱动,对于磁盘等共享设备,一般不必进行分配,但有些系统也采用静态分配方式把一些柱面或磁道分配给不同的作业使用,这样做会使存储空间的利用率降低。

操作系统中常用的设备分配算法有先请求先服务、优先级高者先服务等。此外,在多个进程请求分配设备时,应预先进行检查,防止因循环等待对方所占用的设备而产生死锁。

2. 设备分配数据结构

实现设备分配,系统中应建立设备分配数据结构:设备类表和设备表。系统拥有一张设备类表,每类设备对应于设备类表中的一栏,其中包括设备类、总台数、空闲台数和设备表起始地址等。每类设备都有各自的设备表,用来登记这类设备中的每一台物理设备,所包含的内容有:物理设备名(号)、逻辑设备名(号)、占有设备的进程号、已分配/未分配、好/坏标志等。按照上述数据结构不难设计出 I/O 设备的分配/去配流程。

在采用通道结构的系统中,设备分配的数据结构要复杂得多,需要对通道、控制器和每台物理设备进行管理和控制,必须设置:系统设备表、通道控制表、控制器控制表和设备控制表。整个系统建立一张系统设备表,记录系统配置的所有物理设备的情况,每台物理设备占有一栏,包括设备类型、台数、设备号、设备控制表指针等。对于通道控制表、控制器控制表和设备控制表,每个通道、控制器、设备各设置一张,分别记录各自的地址(标识符)、状态(忙/闲、已分配/未分配)、

设备和虚拟设备这三类,相应的管理和分配设备的技术就分为静态分配、动态分配和虚拟分配。

有些外部设备,如卡片机、打印机、磁带机等,往往只能由一个作业以独占方式使用,这是由这类设备的物理特性决定的,例如,用户在一台分配给他的卡片机上装一叠卡片,卡片上存放这位用户的作业要处理的数据,作业执行过程中将随机地读入卡片上的数据,对其进行加工处理,因此,不可能在此作业暂时不使用卡片机时,人为地换上另一作业的一叠卡片。只有当某作业归还卡片机之后,才能让另一作业去占用它。

对于独占型设备,往往采用静态分配,即在作业执行之前,将所要使用的设备全部分配给它,当作业在执行过程中不再需要使用这类设备或作业执行结束将要撤离时,再收回设备。静态分配实现起来简单,能够防止系统发生死锁,但会降低设备利用率。例如,对打印机采用静态分配,在作业执行前分配,但是直到作业产生结果时才使用打印机,这样,尽管这台打印机在大部分时间处于空闲状态,但是,其他作业却不能使用它。

如果对打印机采用动态分配,在作业执行过程中要求输出一批信息时,系统才把打印机分配给作业,当一个文件输出完毕要关闭时,系统就收回分配给此作业的打印机。采用动态分配后,在打印机上可能依次输出若干作业的信息,由于输出信息以文件为单位,每个文件的头和尾均设有标志,如用户名、作业名、文件名等,操作员很容易辨认输出信息属于哪个用户,所以,对于某些以独占方式使用的设备,采用动态分配方式,不仅是可行的而且也能提高设备利用率。

另一类设备,如磁盘,往往可让多个作业同时使用,这是因为这类设备的存储容量大、存取速度快且可直接访问。例如,每个作业的信息组织成文件存放在磁盘上,使用信息时可按名称查找文件,并从磁盘上读出,当用户提出存取信息的要求时,总是先由文件管理进行处理,确定信息的存放位置,再向设备管理提出 I/O 请求。所以,对于这类设备,设备管理的主要工作是驱动调度和实施驱动,对于磁盘等共享设备,一般不必进行分配,但有些系统也采用静态分配方式把一些柱面或磁道分配给不同的作业使用,这样做会使存储空间的利用率降低。

操作系统中常用的设备分配算法有先请求先服务、优先级高者先服务等。此外,在多个进程请求分配设备时,应预先进行检查,防止因循环等待对方所占用的设备而产生死锁。

2. 设备分配数据结构

实现设备分配,系统中应建立设备分配数据结构:设备类表和设备表。系统拥有一张设备类表,每类设备对应于设备类表中的一栏,其中包括设备类、总台数、空闲台数和设备表起始地址等。每类设备都有各自的设备表,用来登记这类设备中的每一台物理设备,所包含的内容有:物理设备名(号)、逻辑设备名(号)、占有设备的进程号、已分配/未分配、好/坏标志等。按照上述数据结构不难设计出 I/O 设备的分配/去配流程。

在采用通道结构的系统中,设备分配的数据结构要复杂得多,需要对通道、控制器和每台物理设备进行管理和控制,必须设置:系统设备表、通道控制表、控制器控制表和设备控制表。整个系统建立一张系统设备表,记录系统配置的所有物理设备的情况,每台物理设备占有一栏,包括设备类型、台数、设备号、设备控制表指针等。对于通道控制表、控制器控制表和设备控制表,每个通道、控制器、设备各设置一张,分别记录各自的地址(标识符)、状态(忙/闲、已分配/未分配)、

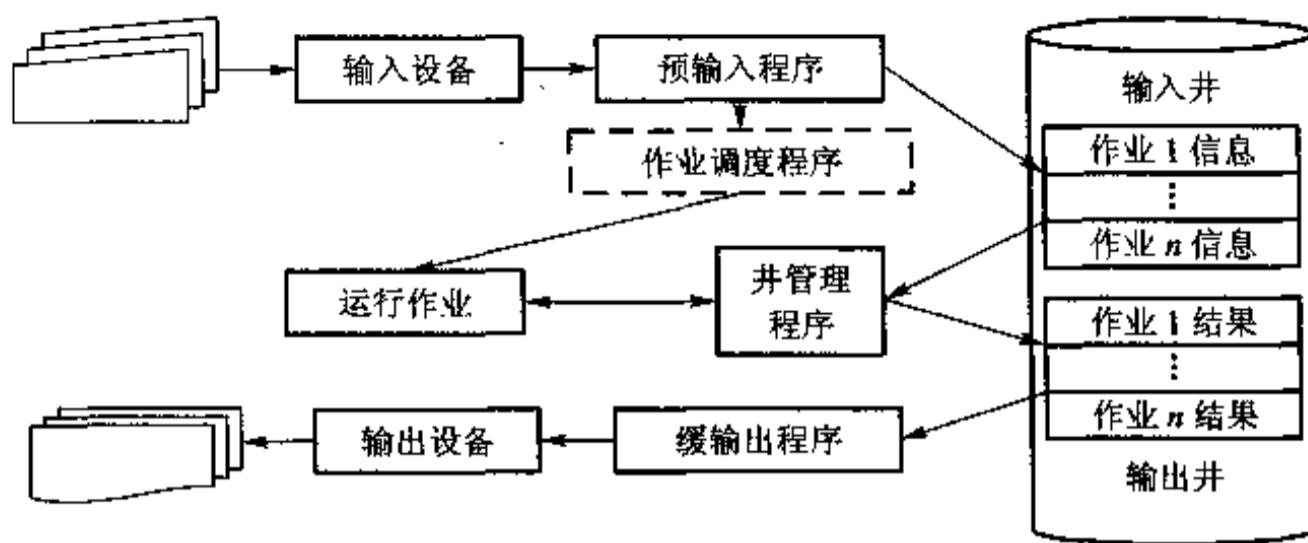


图 5.6 SPOOLing 系统的组成和结构

来登记作业的各个输入文件的情况,包括设备类、文件名、信息长度及存放位置等;缓输出表用来登记作业的各个输出文件的情况,包括设备类、文件名、信息长度及存放位置等。输入井中的作业有以下 4 种状态:

- (1) 输入状态:作业信息正在从输入设备上预输入;
- (2) 收容态:作业预输入结束,但未被选中执行;
- (3) 执行状态:作业已被选中运行,它在运行过程中可以从输入井中读取数据,也可向输出井写入数据;
- (4) 完成状态:作业已经撤离,作业的输出结果等待缓输出。

作业表的状态项指示哪些作业正在预输入,哪些作业已经预输入完成,哪些作业正在执行,等等,作业调度程序根据预定的调度算法选择处于收容状态的作业执行。

通常由操作员输入命令启动预输入程序工作,系统响应预输入命令后,调出预输入程序,查看作业表及输入井能否容纳新作业。如果允许,便通知操作员在输入设备上安装输入介质,然后,预输入程序在工作过程中读出作业控制说明书,将所获得的作业信息以及预输入表的位置登入作业表;此后,依次读入作业的其他信息,在输入井中寻找空闲块,把读入的信息以文件形式登记到预输入表中,直到预输入结束。

存放在井中的文件称为井文件,井文件空间的管理比较简单,它被划分成等长的物理块,每块用于存放一条或多条逻辑记录。可以采用两种方式存放作业的数据信息。第一种方式是连接方式,输入信息被组织成连接文件,文件的第一块的位置登记在预输入表中,其后各块用指针连接起来,读出 n 块后,由连接指针就可找到第 $n + 1$ 块数据的位置。这种方式的优点是:数据信息可以不连续存放,文件空间的利用率高。第二种方式是计算方式,假定从读卡机读入信息并将其存放至磁盘的井文件空间,每张卡片为 80 B,每个磁道可存放 100 个 80 B 的记录,若每个柱面有 20 个磁道,则一个柱面可存放 2 000 张卡片的信息。如果把 2 000 张卡片作为一叠存放在一个柱面上,输入数据在磁盘上的位置为:第 1 张卡片信息是 0 号磁道的第 1 条记录,第 2 张卡片信息是 0 号磁道的第 2 条记录, …, 第 101 张卡片信息是 1 号磁道的第 1 条记录,那么,第 n 张卡

片信息被存放在：

$$\text{磁道号} = [\text{卡片号 } n / 100], \text{记录号} = (\text{卡片号 } n) \% 100$$

卡片号 n 除以 100 的商和余数部分分别为卡片信息存放的磁道号和记录号。

2. 井管理程序

作业执行过程中要求启动某台设备进行 I/O 操作时,作业控制程序截获这个要求并调用井管理程序控制从相应的输入井读取信息,或将信息送至输出井。例如,某作业执行时要求从指定设备上读入某文件的信息,并输入管理程序根据文件名查看其预输入表,获得文件的起始盘地址,计算每次读请求所需信息的存放位置;采用连接方式时,每次保留连接指针,就可将后继块的信息读入;采用计算方式时,只需作简单的计算,获得后继块的地址;当输入井中的信息被使用之后,应归还相应的井区。通过预输入管理程序从输入井读入信息同通过设备管理从设备上输入信息,对于用户而言是一样的。

3. 缓输出程序

当处理器空闲时,操作系统调用缓输出程序执行缓输出,它查看缓输出表,是否有输出打印的文件,文件打印前还可能需要组织作业或文件标题,也可能对从输出井中读出的信息进行格式加工。在具体的实现中,一些操作系统的 SPOOLing 由后台进程管理,而在另一些操作系统中,则由一个核心态线程管理。无论出现哪种情况,操作系统均向用户和系统管理员提供显示 spool 队列、删除不准备打印的文件和挂起暂停打印服务等功能。

5.7.3 SPOOLing 应用

SPOOLing 是多道程序系统中采用虚化技术处理独占型设备的一种方法,这种技术已被广泛应用于许多设备和场合:早期的读卡机、穿孔卡片机;现在的打印机、网络上的电子邮件和 FTP 等。实现相应功能的守护进程(线程)都在用户空间运行,但所完成的是操作系统任务,即把本应由内核实现的功能外移。下面是两个例子。

1. 打印机 SPOOLing 守护进程

对于打印机,尽管可以让用户进程采用打开其设备文件的方式来进行申请和使用,但往往一个进程打开打印机的设备文件后可能长达几个小时不用,其他进程在此期间又都无法打印。为了解决这个问题,可创建一个特殊的进程,叫做打印机守护进程,及一个特殊的目录,称为 spool 打印目录。在打印一个文件之前,应用程序首先产生完整的待打印文件并将其存放在打印目录下。规定系统中此守护进程是唯一有特权使用打印机设备文件的进程,当打印机空闲时,便启动守护进程,打印待输出的文件。通过禁止用户直接使用打印机的设备文件便能解决上述打印机空占的问题。

2. 网络通信 SPOOLing 守护进程

SPOOLing 技术不仅可用于打印机,还可用于其他场合。例如,在网络上传输文件常使用网络守护进程,发送文件之前先将其放在特定的网络 spool 目录下,而后由网络通信守护进程将其取出并发送出去。这种文件传送方式的用途之一是因特网电子邮件系统,此网络将全世界数以

千万台计的计算机联结在一起,通过拨号方式或其他通信网络进行通信。当向某人发送电子邮件时,用户使用一个类似于 send 的程序,它接收所要发送的信件并将其送入固定的电子邮件目录下等待以后发送。整个电子邮件系统在操作系统之外作为一种应用程序运行。

Linux 设备管理

5.8.1 设备管理概述

在 Linux 系统中,所有设备被当做文件来处理,可以使用标准文件系统调用来控制设备的 I/O 操作。对于字符设备和块设备,其设备文件用 mknod 命令创建,用主设备号和次设备号标识,同一个设备驱动程序所控制的所有设备均具有相同的主设备号,并用不同的次设备号加以区别;网络设备也被当做设备文件来处理,所不同的是这类设备由 Linux 系统创建,并由网络控制器初始化。代表设备文件的索引节点通常需要通过 inode 才能寻访,这些 inode 实质上相当于普通文件,所以在打开设备文件的过程中即隐含着对普通文件的操作。Linux 2.3.46 版内核正式引入设备文件系统 devfs,并被纳入文件系统的管辖范围,当设备被挂接至特定的目录下后,就可以将对设备文件的所有操作对应到对设备的操作。

设备文件与普通文件及目录文件有着根本的不同,当进程访问普通文件时,会通过文件系统访问磁盘分区中的数据块;当进程访问设备文件时,它只要驱动物理设备就行,虚拟文件系统负责为应用程序隐蔽设备文件与普通文件之间的差异,可把对设备文件的任一系统调用转换成对设备驱动程序的函数调用。

5.8.2 设备驱动程序

文件系统通常使用标准 I/O 函数来控制设备,这些函数的实现由设备驱动程序全权负责。设备驱动程序与外界的接口分为三部分:驱动程序与内核的接口,通过数据结构 file_operations 来完成;驱动程序与系统引导的接口,利用驱动程序对设备进行初始化;驱动程序与设备的接口,描述驱动程序如何与设备进行交互。设备驱动程序由以下功能模块组成:驱动程序的注册与注销,设备的打开与释放,设备的读写操作,设备的控制操作,设备的中断和轮询处理。

1. 驱动程序的注册与注销

系统引导时调用 sys_setup(),它又调用 device_setup()进行设备的初始化。字符设备的初始化由 chr_dev_init()完成,包括对终端、打印机、鼠标及声卡等字符设备的初始化。块设备的初始化由 blk_dev_init()完成,包括对 IDE 硬盘、软盘、光盘驱动器等块设备的初始化。对字符设备和块设备的初始化都要通过相应的函数向内核注册;在关闭字符设备或块设备时,要通过相应的函数从内核中注销。

2. 设备的打开与释放

打开设备由 open 完成,如用 lp_open()打开打印机,用 hd_open()打开硬盘等。在打开设备时,首先检查设备是否就绪;若首次打开设备,还需进行初始化,并增加设备的引用计数。设备的释放由 release()完成,如用 lp_release()释放打印机,用 tty_release()释放终端设备等。在释放设备时,需要检查并减少设备的引用计数,设备的最后一个释放者需要关闭设备。

3. 设备的读写操作

字符设备通过各自的 read() 和 write() 读写设备数据;块设备通过 block_read() 和 block_write() 读写数据,所带参数与 UNIX 系统的参数完全相同。

4. 设备的控制操作和控制方式

系统通过设备驱动程序中的 ioctl() 来控制设备,如对光盘驱动器的控制使用(cdrom_ioctl()),与读写设备数据不同,ioctl() 与具体的设备密切相关。在 Linux 系统中,设备与主存储器之间的数据传输控制方式有程序查询、中断及 DMA 方式等。

5.8.3 设备 I/O 的处理

1. 数据传输和设备驱动

对字符设备的处理相当容易,既不需要对数据进行缓冲,也不涉及对磁盘的高速缓存。当然,字符设备因各自的需求不同而有所不同:有些字符设备必须实现复杂的通信协议来驱动硬件设备,而有些字符设备只需从硬件设备的 I/O 端口中读取几个值即可,例如,多端口的串口卡设备的驱动程序要比总线鼠标的驱动程序复杂得多。

诸如磁盘之类的典型设备都有很高的平均访问时间,每个操作都需要几毫秒才能完成,这是因为磁盘控制器必须将磁头移动到记录数据的正确的磁道和扇区上。当磁头到达正确位置时,数据传输就可稳定在每秒几千万字节的速率。Linux 内核对于块设备的支持具有以下特点:通过虚拟文件系统提供统一接口,对磁盘数据进行预读,为数据提供磁盘高速缓存。

Linux 内核基本上把 I/O 数据传输划分为两类。

(1) 缓冲区 I/O 操作

所传输的数据保存在缓冲区中,缓冲区是磁盘数据在内核中的普通主存容器,每个缓冲区都和一个特定的块相关联,而这个块由设备号和块号来标识。缓冲区 I/O 操作经常用在进程直接读取块设备文件时,或者当内核读取文件系统中的特定类型的数据块时。

(2) 页 I/O 操作

所传输的数据保存在主存页中,每个页所包含的数据都属于普通文件。因为没有必要把这种数据存放在相邻的磁盘块中,所以使用文件的索引节点及文件内的偏移量来标识这种数据。页 I/O 操作主要用于读取普通文件、文件主存映射和交换。

Linux 的块设备驱动程序被划分为高级驱动程序和低级驱动程序两部分,前者处理虚拟文件系统层,后者处理物理设备。假设进程对一个设备文件发出 read 或 write 系统调用,虚拟文件系统执行相应文件对象的 read 或 write 方法,由此调用高级块设备处理程序中的一个函数,此函数执行的所有操作都与这个硬件设备的具体读写请求有关,然后,激活操纵设备控制器的低级驱

动程序,以执行对块设备所请求的操作。

2. 块设备请求

块设备驱动程序一次可传送一个单独的数据块,但内核并不会为磁盘上被访问的每个数据块都单独执行一次 I/O 操作,确定磁盘块的物理位置是相当费时的,只要有可能,内核就试图把多个块合并在一起,作为一个整体来处理,这样就可以缩短磁头的平均移动时间。

当进程通过虚拟文件系统层或其他的内核部分读写磁盘块时,就真正引发一个块设备请求。这个请求描述所请求的块,及对它执行的操作类型(读写)。然而,并不是请求一经发出,内核就能够满足它,要调度 I/O 操作,之后才会被执行。当请求传送新的数据块时,内核检查能否通过稍微扩大前一个一直处于等待态的请求而满足这个新请求,即能否不进行进一步搜索的操作就能满足新请求。由于磁盘访问大都是顺序的,因此这种简单的机制非常有效。

每个块设备请求都用请求描述符表示,它存放于 request 数据结构中,请求队列只是简单的链表,其元素就是请求描述。请求队列链表的排序状况通常是:首先根据设备号,其次根据最初的扇区号。

```
struct request {
    int rq_status;                      /* 请求状态 */
    kdev_t rq_dev;                      /* 设备号 */
    int cmd;                            /* 所请求的操作 */
    int errors;                         /* 成功或出错代码 */
    unsigned long sector;               /* 第一个扇区号 */
    unsigned long nr_sector;            /* 请求的扇区数 */
    unsigned long current_nr_sector;    /* 当前块的扇区个数 */
    char buffer;                        /* I/O 传送所用的主存区 */
    struct semaphore sem;               /* 请求信号量 */
    struct buffer_head * bh;             /* 第一个缓冲区描述符 */
    struct buffer_head * bhtail;         /* 最后一个缓冲区描述符 */
    struct request * next;              /* 请求队列链表 */
};
```

每个物理块设备都应该有一个请求队列,由块设备驱动程序来维护请求队列,以优化方式对请求进行排序,策略程序顺序扫描这种队列,以最少次数的移动磁头为所有请求提供服务。Linux 系统的块设备驱动程序描述符是一个 blk_dev_struct 类型的数据结构,所有块设备描述符都存放在 blk_dev 表中,此表的索引就是块设备的主设备号。

```
struct blk_dev_struct {
    void * ( * )( void )request_fn;      /* 与具体设备相关的策略程序 */
    void * data;                          /* 驱动程序的私有数据通用队列 */
    struct request plug;                 /* 空插入请求 */
```

```

    struct request current_request;           /* 通用队列中的当前请求 */
    struct request **( * )(kde_t)queue;      /* 从队列中获得一个请求的方法 */
    struct tq struct plug_tq;                /* 插入任务队列元素 */
};

request_fn 字段包含驱动程序的策略程序的地址,策略程序是低级块设备驱动程序的关键函数,为了传送队列中的一个请求所指定的数据,它与物理块设备(磁盘控制器)真正交互。

```

3. 低级请求处理

块设备处理系统的最底层是策略程序,它与物理块设备之间相互作用以满足将队列中的请求聚集在一起的要求,在把新请求插入请求队列之后,策略程序通常才被启动。只要低级块设备驱动程序被激活,就应该对队列中的所有请求都进行处理,直到队列为空时才结束。策略程序的执行过程是:对于队列中的每个元素,与块设备控制器相互作用,共同为请求服务,等待数据传输完成;然后把已处理的请求从队列中删除,继续处理下一个请求。这样处理的效率并不高,即使用 DMA 传送数据,策略程序在等待 I/O 操作完成的过程中必须自行挂起,因此不相关的应用进程将受到严重的影响。因此,很多低级设备驱动程序都采用如下模式。

(1) 策略程序处理队列中的当前请求并设置块设备控制器,以便在数据传输完成时可以产生一个中断,然后策略程序终止。

(2) 当块设备控制器产生中断时,中断处理程序就激活下半部分,下半部分处理程序把这个请求从队列中删除,重新执行策略程序来处理队列中的下一个请求。

低级块设备驱动程序通常进一步划分成两类:为请求的每个块单独提供服务的驱动程序;为请求的多个块一起服务的驱动程序。后一类驱动程序的设计和实现比前一类驱动程序更加复杂,实际上,虽然这些扇区在物理块设备上都是相邻的,但是在 RAM 中的缓冲区并不一定连续。因此,这样的驱动程序都可为 DMA 数据传送分配一个临时区域,然后在临时区域和请求队列的每个缓冲区之间执行一种主存到主存的数据复制。



5.9 Windows 2003 I/O 系统

5.9.1 I/O 系统结构和组件

1. 系统结构和组件

Windows I/O 系统是执行体的组件,它向用户提供统一的高层接口,方便用户执行 I/O 操作,其设计目标是:高效、快速地进行 I/O 处理;使用标准的安全机制保护共享资源;支持多种可安装的文件系统;支持即插即用,能在系统中动态地添加或删除设备;允许系统或单个设备进入和离开低功耗状态以节约能源;能满足 Win32、OS/2 和 POSIX 子系统所指定的 I/O 服务的需要。

I/O 系统在用户态 I/O 库函数和物理 I/O 硬件之间存在多层系统组件,包括文件系统驱动

程序、高速缓存管理器、过滤器驱动程序、低层网络和网络驱动程序。

下面介绍 Windows I/O 系统结构和组件,再描述设备驱动程序和 I/O 请求的关键数据结构,及在整个系统中完成 I/O 请求的必要步骤。I/O 系统由执行体组件和设备驱动程序组成,如图 5.7 所示,各组件的功能如下。

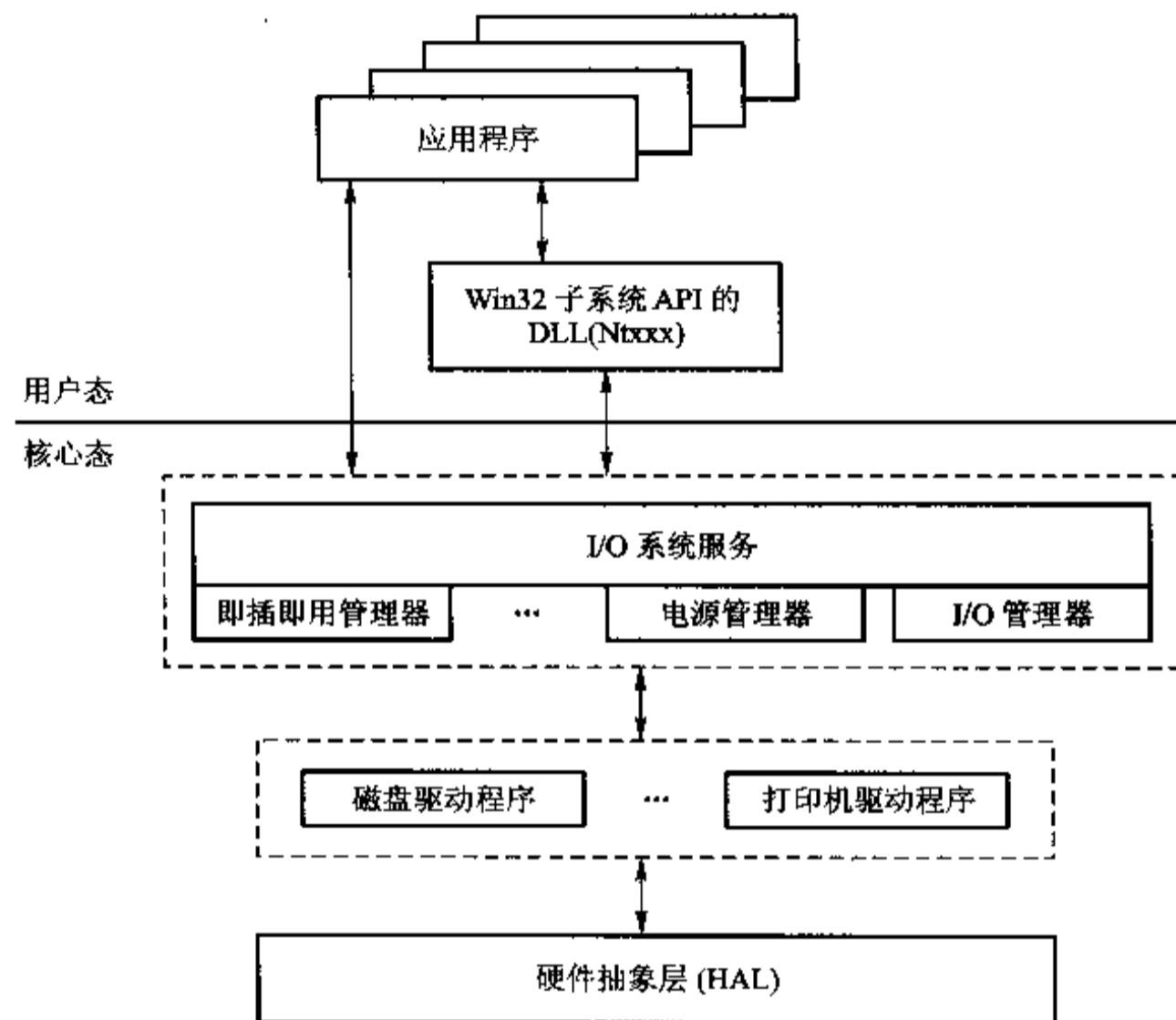


图 5.7 I/O 系统组件

- (1) Win32 子系统 I/O 服务以 API 形式提供,通过子系统 DLL 调用核心态的执行体系统服务,完成应用程序的 I/O 请求。
- (2) I/O 系统服务是核心态执行的系统调用,通过 I/O 管理器驱动 I/O 请求,完成下层 I/O 处理。
- (3) I/O 管理器负责协调各驱动程序之间的工作。
- (4) 设备驱动程序为某种类型的设备提供 I/O 接口,它从 I/O 管理器接收处理要求,把 I/O 请求转化为特定类型设备的 I/O 控制命令,完成指定的 I/O 请求,并把处理结果通知 I/O 管理器。
- (5) 即插即用管理器通过与 I/O 管理器和总线驱动程序的协同工作来检测硬件资源的分配情况,随时检测相应的硬件设备的添加和删除情况。
- (6) 电源管理器通过与 I/O 管理器的协同工作来检测整个系统和单个硬件设备的当前工作状况,完成不同情况下电源状态的转换。

此外,I/O 系统还使用注册表,作为一个数据库,注册表存储基本硬件设备的描述信息以及驱动程序的初始化和配置信息,用户态即插即用组件则用于控制和配置设备的用户态 API。

2. I/O 管理的特点

(1) I/O 包驱动

在 Windows 中,I/O 系统是由“包”驱动的,大多数 I/O 请求用“I/O 请求包”(I/O Request Packet,IRP)表示,它从一个 I/O 系统组件移动到另一个组件。IRP 是各个阶段控制如何处理 I/O 操作的数据结构。

I/O 管理器把用户态 I/O 请求转化为“I/O 请求包”(IRP),将其传递给正确的驱动程序,驱动程序接收 IRP 后,执行 IRP 所指定的操作,且在完成操作之后把 IRP 送回 I/O 管理器,或为下一步处理而通过 I/O 管理器将其送到另一个驱动程序。

(2) 通过虚拟文件实现 I/O 操作

在 Windows 中,所有 I/O 操作都通过虚拟文件执行,用以隐藏 I/O 操作的实现细节,为应用程序提供统一的使用设备的接口。虚拟文件是指用于 I/O 的所有源或目标都被当做文件处理,所有被读取或写入的数据都看做直接读写到这些虚拟文件的流。用户态应用程序(Win32、POSIX 或 OS/2)调用本机的文件对象服务进行文件读写并完成其他文件操作,I/O 管理器动态地把这些虚拟文件请求指向适当的设备驱动程序,以控制真正的文件、文件目录、物理设备、管道和网络。

5.9.2 I/O 系统数据结构

有 4 种主要的数据结构代表 I/O 请求:文件对象、驱动程序对象、设备对象和 I/O 请求包。

1. 文件对象

文件作为对象来处理,它是两个或多个用户态线程可共享的系统资源,文件对象提供基于主存储器的共享物理资源的表示。当调用者打开文件或设备时,I/O 管理器将为文件对象返回句柄,调用者使用文件句柄对文件进行操作。像其他的执行体对象一样,文件对象由包含访问控制表(Access Control List,ACL)的安全描述符保护,安全子系统决定文件的 ACL 是否允许线程访问文件。

当使用共享资源时,线程必须保证它对共享文件、文件目录或设备访问的同步。例如,如果一个线程正在写入一个文件,当它打开文件句柄以防止其他线程在同一时间对此文件执行写入操作时,它应该指定独占式的写访问,还可使用 Win32 LockFile 函数,在写的同时锁定文件的某些部分。

2. 驱动程序对象和设备对象

当线程为文件对象打开句柄时,I/O 管理器必须根据文件对象的名称来决定调用哪个或哪些驱动程序来处理请求,通过驱动程序对象和设备对象来满足这些要求,而且 I/O 管理器必须在线程下一次使用同一个文件句柄时能够定位这个信息。

(1) 驱动程序对象代表系统中一个独立的驱动程序,I/O 管理器从这些驱动程序对象中获

得和记录每个驱动程序的调度例程的入口地址。

(2) 设备对象在系统中代表物理的、逻辑的或虚拟的设备并描述其特征,例如,所需缓冲区的对齐方式及其保存即将到来的 I/O 请求包的设备队列的位置。

当驱动程序被加载至系统中时,I/O 管理器将创建一个驱动程序对象,然后,调用驱动程序的初始化例程,把驱动程序的入口点填入此驱动程序对象中。初始化例程还为每个设备创建设备对象,使得设备对象脱离驱动程序对象。

如图 5.8 所示是驱动程序对象和设备对象之间的关系。一个驱动程序对象通常有多个与其相关的设备对象,设备对象列表代表驱动程序可控制的物理设备、逻辑设备和虚拟设备。例如,磁盘的每个分区都有一个独立的包含具体分区信息的设备对象。然而,相同的磁盘驱动程序被用于访问所有的逻辑分区。当一个驱动程序从系统中卸载时,I/O 管理器会使用设备对象队列来确定哪个设备由于驱动程序的卸载而受到影响。

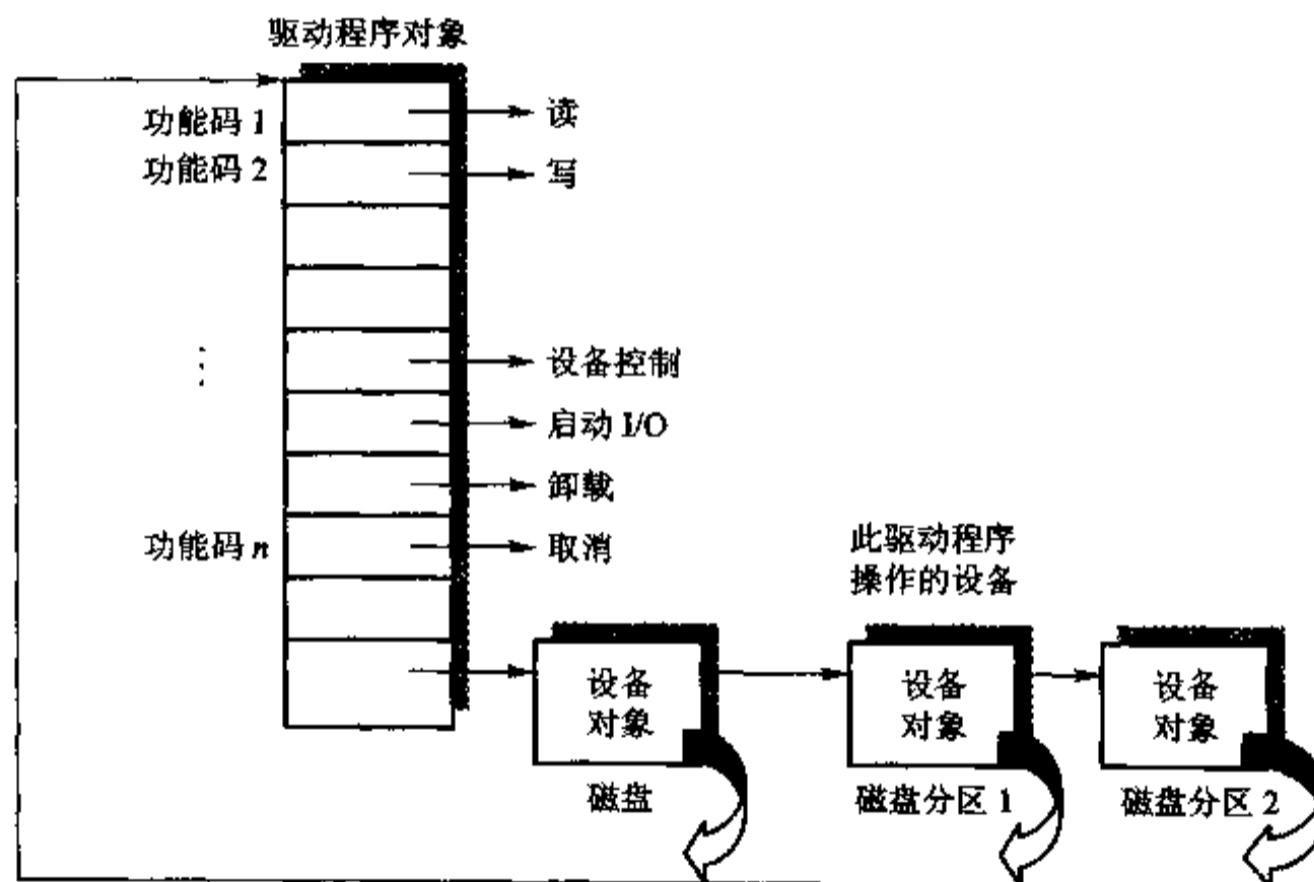


图 5.8 驱动程序对象和设备对象

设备对象反过来指向其自己的驱动程序对象,这样 I/O 管理器就知道在接收一个 I/O 请求时应调用哪个驱动程序,它使用设备对象找到代表服务于此设备驱动程序的驱动程序对象,然后,利用初始请求所提供的功能码来索引驱动程序对象,以便找到对应的驱动程序。

3. I/O 请求包

I/O 请求包(IRP)是 I/O 系统用来存储处理 I/O 请求所需信息的数据结构,当线程调用 I/O 服务时,I/O 管理器就构造 IRP 来表示整个系统 I/O 进展中所要执行的操作。IRP 由两部分组成:固定部分(标题)和一个或多个堆栈单元。固定部分包括:请求的类型和所请求的数据个数,是同步请求还是异步请求,用于缓冲 I/O 的指向缓冲区的指针,及随着请求的进展而变化的状

态信息,等等。IRP 堆栈单元包括功能码、完成特定功能的参数和一个指向调用者文件对象的指针。

在处于活动状态时,每个已构造好的 IRP 都存储在与请求 I/O 的线程相关的 IRP 队列中,如果线程终止或者被终止时还拥有未完成的 I/O 请求,这种安排就允许 I/O 系统找到并释放未完成的 IRP。

4. 驱动程序的分类和结构

(1) 驱动程序分类

① 核心模式驱动程序:包括文件系统驱动程序、即插即用管理驱动程序、电源管理驱动程序、图形驱动程序和操作系统驱动程序模型(Windows Driver Model, WDM)(又分为总线驱动程序、功能驱动程序和过滤驱动程序);

② 用户模式驱动程序:包括虚拟设备驱动程序和 Win32 打印驱动程序;

③ 硬件支持驱动程序:类驱动程序、端口驱动程序和小端口驱动程序。

(2) 驱动程序组成

驱动程序包括一组处理 I/O 请求的不同阶段的例程,主要有初始化例程、功能例程、启动 I/O 例程、中断服务例程、延迟过程调用例程、完成例程、取消例程、卸载例程、系统关闭通知例程和错误记录例程。

5. 多处理器的 I/O 同步问题

在多处理器系统中,Windows 能同时在多个 CPU 上运行,驱动程序必须同步执行对全局驱动程序数据的访问。这有两个主要原因:驱动程序的执行可被高优先级的线程抢先,或时间片到时而被中断,或被其他中断所中断;在多处理器系统中,可能同时运行同一个驱动程序。若不能同步执行,就会导致相应的错误。例如,若设备驱动程序运行在低优先级 IRQL 上,其间可能被设备中断请求所中断,导致在设备驱动程序正在运行时,其中断服务例程又去执行。如果中断服务例程也要修改设备驱动程序正在修改的数据,例如,设备寄存器、堆存储器或静态数据,则在中断服务例程执行时,数据可能被破坏。

要避免这种情况的发生,所编写的设备驱动程序就必须和其中断服务例程同步对共享数据的访问,在尝试更新共享数据之前,设备驱动程序必须锁定其他的线程或 CPU,以防止它们修改同一个数据结构。当设备驱动程序访问其中断服务例程也要访问的数据时,由内核提供设备驱动程序必须调用的特殊的同步例程,当共享数据被访问时,这些内核同步例程将禁止中断服务例程的执行。在单处理器系统中更新一个数据结构之前,这些例程将 IRQL 提高到指定的级别,然而在多处理器系统中,因为一个驱动程序能同时在多个处理器上执行,这种技术就不足以阻止其他的访问。因此,通过“自旋锁”来防止其他 CPU 对共享数据结构的访问。

5.9.3 I/O 类型和处理

1. I/O 类型

应用程序在发出 I/O 请求时可设置不同的选项,例如,设置同步 I/O 或异步 I/O,设置获取

I/O 数据的方式,等等。

(1) 同步 I/O 和异步 I/O

通常 I/O 操作都以“同步”方式完成,即当调用者启动 I/O 成功之后,在设备执行数据传输期间,必须等待直到 I/O 操作完成并返回状态码为止。

“异步 I/O”却允许应用程序发出 I/O 请求,在设备传输数据的同时,应用程序可继续执行,调用者不能访问任何来自 I/O 操作的数据,直到设备完成数据传输。线程可通过等待同步对象的句柄使其执行与 I/O 请求的完成同步。当 I/O 操作完成时,这些同步对象将变成有信号状态。

(2) 快速 I/O

快速 I/O 是一种特殊的传输机制,允许 I/O 系统不产生 IRP,而直接转向文件系统驱动程序或高速缓存管理器去执行 I/O 请求。

(3) 映射文件 I/O 和文件高速缓存

通过使用 Win32 的 CreateFileMapping 和 MapViewOfFile 函数,映射文件 I/O 可被应用程序使用。在操作系统中,映射文件 I/O 用于文件高速缓存和映像加载活动,从而为 I/O 绑定程序提供更快的响应速度。

(4) “分散/集中”I/O

支持特殊类型的高性能 I/O,称为“分散/集中”I/O (scatter/gather),可通过 Win32 的 ReadFileScatter 和 WriteFileScatter 函数来实现,这些函数允许应用程序执行读取或写入操作,从虚拟主存的多个缓冲区读取数据并写入磁盘文件的一个连续区域,反之亦然。

2. I/O 处理步骤

一个 I/O 请求会经过若干处理阶段,根据请求所指向的是由单层驱动程序操作的设备,还是经过多层驱动程序才能到达的设备,它经历的阶段也有所不同,处理步骤的不同还依赖于 I/O 的类型。Windows 对核心态设备驱动程序的 I/O 请求的处理包含以下步骤。

- (1) I/O 库函数经过语言的运行时库转换成对子系统 DLL 的调用;
- (2) 子系统 DLL 调用 I/O 系统服务;
- (3) I/O 系统服务调用对象管理程序,检查给定的文件名,再搜索名空间,把控制权转交给 I/O 管理器寻找文件对象;
- (4) 驱动程序询问安全子系统,确定线程的存取权限。如果不允许,就出错返回;否则,由对象管理器把所允许的存取权限和返回的文件句柄连在一起,返回用户态线程,之后线程用文件句柄对文件实施操作;
- (5) I/O 管理器以 IRP 的形式将 I/O 请求传送给设备驱动程序,驱动程序启动 I/O 操作;
- (6) 设备完成指定的操作,请求 I/O 中断,设备驱动程序的中断服务例程服务于中断;
- (7) I/O 管理器调用 I/O 完成例程,将完成状态返回给调用线程。

上述是同步 I/O 的执行步骤,对于异步 I/O,还要增加一步,I/O 管理器将控制权返回调用线程,使得调用线程与 I/O 操作并行执行。

3. 对单层驱动程序的 I/O 请求处理

(1) 单层驱动程序 I/O 请求的步骤

通过例子说明单层驱动程序的 I/O 处理,假设应用线程同步地向打印机的写缓冲区中写若干字符,且打印机连接在计算机的并行端口,由单层打印驱动程序控制。在 Windows 中,应用程序通过系统调用

```
WriteFile(hFile, Buf, nbytes, NULL);
```

实现同步地向打印机输出信息。其中,hFile 是写文件对象的句柄,子系统事先用 CreateFile() 系统调用打开并行口(名为 \ device \ parallel() 的虚拟文件)句柄且同步 I/O(NULL 参数)。单层驱动程序处理同步 I/O 请求的过程如图 5.9 所示。

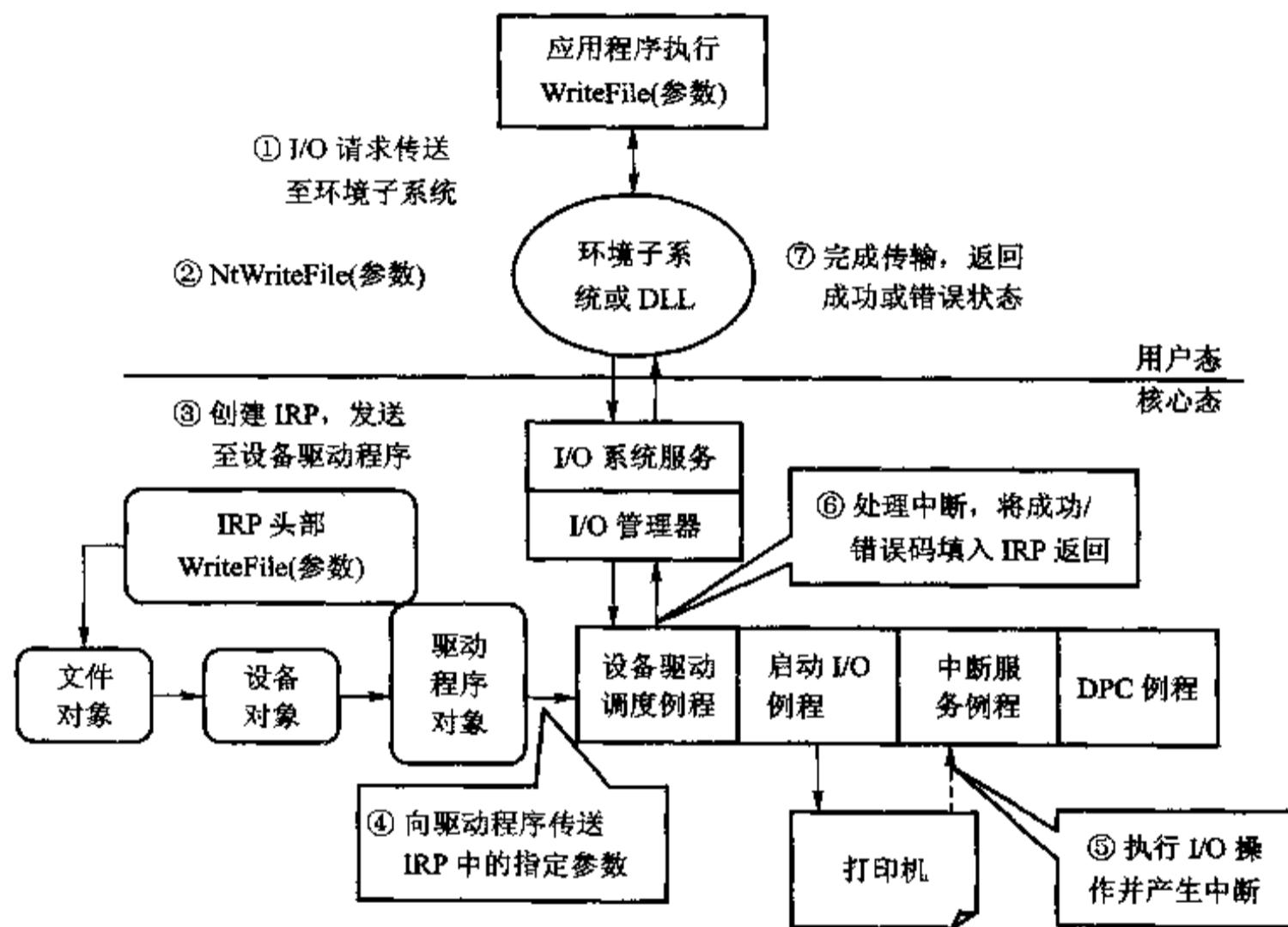


图 5.9 单层驱动程序处理同步 I/O 请求的过程

环境子系统接到写命令,检查命令和参数的合法性之后,转换成对核心系统服务 `NtWriteFile` 的调用,之后 I/O 管理器接管此任务,完成 6 个方面的工作。

① I/O 管理器根据调用命令生成一个 I/O 请求包 (IRP),其中包括:指向文件对象的指针,所执行的操作和参数,初始化堆栈单元;

② I/O 管理器根据 IRP 中的文件对象指针,定位打印机设备对象,找到打印机的驱动程序,且以 IRP 中的参数调用此驱动程序;

③ 驱动程序启动设备,执行写操作;

④ 设备完成写操作后,请求中断系统响应中断,转入中断处理程序,产生和排队 DPC。DPC 执行,且将完成或出错状态码写入 IRP 中,对 I/O 完成过程进行初始化,控制权返回 I/O 管理器;

⑤ I/O 管理器调用驱动程序的 I/O 完成例程,将核心态 APC 排到启动 I/O 的线程中,并且清除 IRP;

⑥ I/O 管理器最终将控制权返回环境子系统或 DLL,从而完成一次 I/O 操作。

(2) 设备 I/O 的中断处理

在完成数据传输之后,设备产生中断,处理器响应中断,最终将控制权转交给设备的中断服务例程(ISR)。Windows 上的 ISR 典型地用以下两个步骤来处理设备中断。

① 当 ISR 首次被调用时,它在设备 IRQL 上获得设备状态,然后产生一个软件中断 DPC,且将中断的后半段任务排入 DPC 队列,退出服务例程,清除中断;

② 当中断请求级降低到 DPC 以下时,软件中断 DPC 出现,使 DPC 例程被调用,最终完成对设备的中断处理。DPC 所做的工作是检查设备是否正常完成,若是,则启动下一个正在设备队列中等待的 I/O 请求,将本次 I/O 操作完成状态码记录在 IRP 中。DPC 完成这些工作后,通过调用 I/O 管理器来完成本次 I/O 操作并清除 IRP。

使用 DPC 执行大部分设备服务的优点是:任何优先级位于此设备 IRQL 和 Dispatch/DPC IRQL 之间被阻塞的中断,允许在低中断优先级的 DPC 处理之前发生。因此,中间优先级的中断可更快地得到响应和服务。

(3) I/O 请求的完成过程处理

当设备驱动程序的 DPC 例程执行完后,在结束本次 I/O 请求之前还要做一些工作,这就是中断处理的第三步,称作“I/O 完成过程”处理,它因 I/O 操作的不同而不同。例如,全部的 I/O 服务都把操作的结果记录在由调用者提供的数据结构“I/O 状态块”中;与此相似,一些执行缓冲 I/O 的服务要求 I/O 系统返回数据给调用线程。

在上述两种情况中,I/O 系统必须把存储在系统主存储器中的数据复制到调用者的虚拟地址空间中。若要获得调用者的虚拟地址,I/O 管理器必须在调用者线程的上下文中进行数据传输。这样,I/O 完成过程处理也需要两步完成。

① 由 I/O 管理器把一个核心态 APC 排队到调用线程的描述表中,实现这个任务。APC 在特定线程的描述表上执行,而 DPC 在任意线程的描述表上执行,这就意味着 DPC 例程不能涉及用户态进程的地址空间;

② 当线程开始在较低的 IRQL 上执行时,出现核心态 APC 中断,此 APC 可以在指定线程的描述表上执行,将被缓存的数据复制到此线程、置文件句柄为有信号状态、排队任何用户态 APC 以备执行和清除 I/O 请求包(IRP),完成 I/O 的第二步。

系统响应 APC 中断,内核把控制权转交给 I/O 管理器的 APC 例程,它将把数据(若有)和返回状态复制到调用者的地址空间,释放代表 I/O 操作的 IRP,且将调用者的文件句柄设置为有信号状态,至此完成一次 I/O 请求。在文件句柄上等待的最初调用者或其他线程都将从它们的等

待态中被唤醒且恢复执行。

4. 对多层驱动程序的 I/O 请求处理

大部分 I/O 请求都必须通过调用多层驱动程序来实现,下面给出基于文件系统实现读写文件 I/O 请求的两层驱动程序的处理过程。

当应用程序同步请求从磁盘文件读写数据时,首先调用打开文件命令,为指定文件创建文件对象,并返回文件对象的句柄,然后,使用文件句柄对文件进行读写。

(1) 打开一个文件

应用程序使用 C 语言所提供的标准库函数 fopen 打开一个文件时,其调用格式为:

```
fp = fopen(d:\myfile.dat, "r");
```

其执行过程如下。

① C 运行时库将 fopen 转换成对 Win32 DLL 的 CreateFile 函数的调用:

```
fp = CreateFile(d:\myfile.dat, ...);
```

② Win32 DLL 调用 NT DLL 再将它转换为对核心态系统服务的调用:

```
fp = NtCreateFile(d:\myfile.dat, ...);
```

③ 系统服务的 I/O 管理器调用对象管理器,当它根据文件名在文件对象名字空间中查找时,发现驱动器 d 是一个符号连接(" \ device \ harddisk01"),找到一个代替它的字符串" \ device \ harddisk01",以字符串 harddisk01 为参数调用文件系统并将文件名" \ myfile.dat"传递给它,请求文件系统定位一个磁盘分区,打开此磁盘分区上的这个文件。

④ 文件系统在检查存取权限的合法性后,由 I/O 管理器创建一个文件对象和设备对象,并在文件对象中存储 harddisk01 设备对象的一个指针,然后向调用者返回一个文件句柄,之后用户使用此文件句柄就可以对文件进行读写。

(2) 读文件

调用者使用文件句柄发出读请求时,调用 C 标准库函数:

```
ReadFile(hFile, Buf, nbytes, NULL);
```

其执行过程如下。

① 通过环境子系统将此命令转换成 NtReadFile()。系统使用文件对象通过存取控制表检查操作方式的合法性。若合法,转②,否则出错返回。

② I/O 管理器创建代表此操作的 I/O 请求包(IRP),传递 IRP 给文件系统驱动程序。

③ 文件系统根据 IRP 的读请求类型和文件的读指针,计算读文件的相对块和块内偏移地址。检查高速缓存中是否已有所指定的数据,若有,立即将数据送至用户区,返回;若无,进而找到要读磁盘的相对块,并填入 IRP 中或新生成一个 IRP。将修改后的 IRP 返回给 I/O 管理器。

④ I/O 管理器用这些 IRP 调用磁盘驱动程序,将磁盘相对块转换为磁盘的物理地址(柱面号、磁头号和扇区号),填入 IRP,并排队 IRP。

⑤ 若磁盘空闲,启动磁盘完成读一个盘块到缓冲区的操作,并把执行状态返回 I/O 管理器。

⑥ 磁盘完成传输,产生中断。中断处理程序进行简单的处理,生成一个 DPC 并排队 DPC,

之后结束中断上半部处理并返回。

⑦ 当 IRQL 降到低于 DPC 的 IRQL 时,磁盘 DPC 软中断被系统接收,并具体处理设备的中断,主要检查设备是否正常完成,若是,查看磁盘队列是否还有 IRP 排队,若有,再启动下一个 I/O 请求,之后返回 I/O 管理器。

⑧ I/O 管理器调用 I/O 完成例程,最终将完成状态返回给用户,将应用程序所需数据送至用户区,完成本次读请求。

(3) 写文件

写文件操作与读操作类似,调用看使用文件句柄进行写请求时,调用 C 标准库函数:

```
WriteFile(hFile, Buf, nbytes, NULL);
```

其执行过程如下:

① 环境子系统将此命令转换成 NtWriteFile()。系统使用文件对象通过存取控制表检查操作方式的合法性。若合法,转②,否则出错返回。

② I/O 管理器创建代表此操作的 I/O 请求包(IRP),传递 IRP 给文件系统驱动程序。

③ 文件系统根据 IRP 的写请求类型和文件的写指针,计算写文件的相对块和块内偏移地址。检查是否为其分配高速缓存,若已经分配,直接写高速缓存;若无,则申请一个,将数据写入高速缓存。再检查是否已经分配磁盘块,若未分配,为其分配,之后返回。

系统在后台完成文件的写操作。以下几步完成写磁盘:

④ 将磁盘的相对块转换为磁盘的物理地址(柱面号、磁头号和扇区号),填入 IRP,并排队 IRP。

⑤ 若磁盘空闲,启动磁盘完成将高速缓存中的数据写入磁盘的操作,并把执行状态送回 I/O 管理器。

⑥ 磁盘完成传输,产生中断,中断处理程序进行简单的处理,生成一个 DPC 并排队 DPC,之后中断返回。

⑦ 磁盘的 DPC 软中断被系统接收,并具体处理设备的中断。

⑧ I/O 管理器调用 I/O 完成例程,最终将完成状态返回给用户,完成本次写请求。

5.9.4 高速缓存管理

1. 高速缓存管理器的主要特征

高速缓存管理器是一组核心态函数和系统线程,它们与主存管理器一起为所有 Windows 文件系统驱动程序提供本地与网络数据的高速缓存。高速缓存管理器提供一种高速、智能的机制,用以减少磁盘 I/O 操作、增加系统的吞吐量,基于虚拟块的高速缓存使高速缓存管理器能够进行智能预读。依靠全局主存管理器的映射文件机制访问文件数据,高速缓存管理器提供特殊的快速 I/O 机制缩短用于读写操作的时间,而且将与物理主存有关的管理工作交给主存管理器,这样能减少代码冗余,提高系统效率。高速缓存管理器的主要特征和实现要点如下。

(1) 单一集中式系统高速缓存

对于本地硬盘、软盘、网络文件服务器或 CD-ROM 上的数据,无论是用户文件数据或是文件系统元数据(目录和文件头),都能对其进行缓存。

(2) 与主存管理器结合

采用将文件 I/O 映射到系统虚拟空间的方法来访问数据,其中使用标准区域对象。访问位于映射视图中的地址时,主存管理器为不在物理主存的逻辑块分配页面,以后需要主存储器时,再将高速缓存中的数据页面换出,写回映射文件。通过映射文件实现基于虚拟地址空间的高速缓存,高速缓存管理器在访问缓存中文件的数据时避免产生读写 I/O 请求包,取而代之,仅在主存储器和被缓存的文件部分所映射的虚拟地址之间复制数据,并依靠主存管理器处理换页。这种设计使得打开缓存文件就像将文件映射到用户地址空间一样。

(3) 高速缓存的一致性

高速缓存管理器的一个重要功能是保证访问高速缓存数据的任何进程都能得到这些数据的最新版本。当进程打开一个被缓存的文件,而另一个进程直接将文件映射到其地址空间时,可能产生数据的不一致性。这种潜在的问题不会在 Windows 中出现,因为高速缓存管理器和应用程序使用相同的主存管理文件映射服务将文件映射到其地址空间,而主存管理器保证每一个被映射文件只有唯一的版本,它将文件的所有视图映射到物理主存页面的单独集合。例如,如果进程 1 有一个文件视图被映射到其用户地址空间,进程 2 通过系统缓存访问同一视图,那么,进程 1 所做的任何改变都可由进程 2 看到,而不用等到这些改变被回写。

(4) 虚拟块缓存

多数操作系统高速缓存管理器基于磁盘逻辑块(logical block)缓存数据,采用这种方式,高速缓存管理器知道磁盘分区中的哪些块在高速缓存中。与之相比,Windows 高速缓存管理器用虚拟块缓存(virtual block caching)方式对缓存中文件的某些部分进行追踪。通过主存管理器的特殊系统高速缓存例程将 256 KB 大小的文件视图映射到系统虚拟地址空间,高速缓存管理器能够管理文件的这些部分。这种方式的好处有:

- ① 使智能化的文件预读成为可能,高速缓存能够追踪哪些文件的哪些部分在缓存中,因而能够预测调用者下一步将访问哪里;
- ② 允许 I/O 系统绕开文件系统访问已经在缓存中的数据(快速 I/O),高速缓存管理器知道哪些文件的哪些部分在缓存中,它能返回被缓存数据的地址从而满足 I/O 的需要,而无须调用文件系统。

(5) 基于流的缓存

高速缓存管理器与文件缓存相对应,设计了字节流的缓存。一个流是指在文件内的字节序列,NTFS 允许文件包括多个流对象;高速缓存管理器通过独立地缓存每个字节流来适应这些文件系统。NTFS 能够拥有这种特点,得益于把主文件表放入字节流中并缓存这些字节流。事实上,虽然 Windows 高速缓存管理器被认为是高速缓存文件,但它所缓存的是字节流,这些字节流

通过文件名标识,如果在文件中有多个字节流存在,还要标明字节流名。

(6) 可恢复文件系统支持

可恢复文件系统(recoverable file system),如NTFS,在系统失败后可以修复磁盘卷结构。这就是说,当系统失败时,正在进行的I/O操作必须全部完成,或在系统重启时从磁盘中全部恢复。未完成的I/O操作可能破坏磁盘卷,甚至导致整个磁盘卷不可访问。为了避免出现这个问题,在改变卷之前,可恢复文件系统将维护一个日志文件(log file)。在每次涉及文件系统结构(文件系统的元数据)的修改写入卷之前,此日志文件进行记录。如果因系统失败而中断了正在进行的卷修改,可恢复文件系统可以根据日志文件中的信息重新执行卷修改操作。

为了保证成功地恢复一个卷,在卷修改操作开始之前,记录卷修改操作的日志记录必须被完全写入磁盘。由于写磁盘操作可以被高速缓存,因此高速缓存管理器和文件系统必须协同工作以确保下列操作按顺序进行。

- ① 文件系统写一个日志文件记录,记录将要进行的卷修改操作。
- ② 文件系统调用高速缓存管理器将日志文件记录刷新到磁盘上。
- ③ 文件系统把卷修改内容写入高速缓存,即修改文件系统在高速缓存中的元数据。
- ④ 高速缓存管理器将被修改的元数据刷新到磁盘上,更新卷结构。

2. 高速缓存的结构

高速缓存管理器基于虚拟地址空间缓存数据,所以,它管理一块系统虚拟地址空间区域,而不是一块物理主存区域。高速缓存管理器把每个地址空间区域分或256 KB的槽(slot),称为视图(view)。高速缓存管理器只映射活跃的视图,不映射那些未被激活的文件视图。利用数据结构虚拟地址控制块(Virtual Address Control Block,VACB)跟踪和描述被缓存的文件系统高速缓存的槽。

3. 高速缓存的操作

高速缓存的操作主要包括:回写和延迟写缓存,计算脏页阈值,屏蔽对文件的延迟写,强制写缓存到磁盘,刷新被映射的文件,智能预读,虚拟地址预读和带历史信息的异步预读。

4. 快速I/O机制

通过快速I/O机制,I/O管理器可以调用文件系统驱动程序的快速I/O例程来查看是否能够直接从高速缓存管理器得到所需数据,而不必产生IRP。当线程执行读写操作时,如果文件被缓存而且I/O同步,I/O请求就会被传送到文件系统驱动程序的快速I/O人口点。快速I/O例程一旦断定可以使用快速I/O,就调用高速缓存管理器的读写例程去直接访问缓存中的数据。如果快速I/O不可能进行,文件系统驱动程序返回I/O系统,之后为I/O产生一个IRP,最终调用文件系统的常规读例程。如果文件未被缓存,文件系统驱动程序设置此文件用于高速缓存,这样下一次就可以使用快速I/O来满足读写请求。

5. 高速缓存支持例程

文件设置高速缓存访问后,文件系统驱动程序就可以调用一个函数来访问文件中的数据。

有三种基本的访问缓存数据的方法,每种方法都适合一种特定的情况。

(1) “复制读取”方法:在系统空间中的高速缓存数据缓冲区和用户空间中的进程数据缓冲区之间复制用户数据。

(2) “映射暂留”方法:使用虚拟地址直接读写高速缓存的数据缓冲区。

(3) “物理主存访问”方法:使用物理地址直接读写高速缓存的数据缓冲区。

本 章 小 结

由于现代计算机系统的外部设备种类繁多、特性各异,使得设备管理成为操作系统中最为庞杂和琐碎的部分,其主要任务是控制设备和 CPU 之间的 I/O 操作。设备管理模块在控制各类设备和 CPU 进行 I/O 操作的同时,还要尽量提高设备与设备之间、设备与 CPU 之间的并行性,使得系统效率得以提高。同时,要为用户使用 I/O 设备屏蔽硬件实现细节,提供方便易用的接口。

设备和 CPU 之间的数据传输控制方式主要有 4 种:程序轮询方式、中断控制方式、DMA 方式和通道方式。程序轮询方式和中断控制方式仅适用于配置少量设备の場合,前者采用忙式测试标志,浪费 CPU 时间,设备与 CPU 只能串行工作;后者虽然改进了上述缺点,但中断次数太多,CPU 累计花费在处理中断上的时间很可观,且并行操作的设备数量也受到中断处理速度的限制。DMA 方式和通道方式较好地解决了这些缺陷,它们均采用设备与主存储器直接交换数据的方法,仅当一块数据传输结束,这两种方式才发出中断信号请求 CPU 的干预,从而把 CPU 从繁杂的 I/O 事务中解放出来。这两种方式的区别是:DMA 方式要求 CPU 执行设备驱动程序来启动设备,并做好传输数据的相关准备工作;通道方式则完全是一个相对独立的 I/O 控制系统,当 CPU 发出 I/O 启动命令后,它便接收控制权,完成全部的 I/O 操作。

I/O 软件分为 4 层:设备中断处理程序、设备驱动程序、独立于设备的 I/O 软件、用户层 I/O 软件。设备中断处理程序通常是设备驱动程序的组成部分,置于底层,其主要工作有:分析中断类型以做出相应的处理,检查和修改进程的状态等,它的任务要尽量少,以便提高性能;设备驱动程序中包括与设备相关的所有代码,其工作是:把用户所提交的逻辑 I/O 请求转化为物理 I/O 操作的启动和执行,如设备名转化为端口地址、逻辑记录转化为物理记录、逻辑操作转化为物理操作等,它对于其上层软件屏蔽所有的硬件细节;独立于设备的 I/O 软件的基本功能是实现适用于所有设备的常用 I/O 操作,并向用户层软件提供一致性接口,如设备命名、设备保护、缓冲管理、存储块分配等;用户层 I/O 软件包括在用户空间运行的 I/O 库函数和 SPOOLing 程序。

通道又称为 I/O 处理机,采用通道技术主要解决 I/O 操作的独立性和设备与 CPU 工作的并行性,从而大幅度提高系统的整体性能。对于具有通道的计算机系统,输入输出程序设计涉及 CPU 执行 I/O 指令,通道执行通道命令,及 CPU 和通道之间的通信。与通道有关的若干概念包括:CPU 的 I/O 指令、通道的通道命令、通道程序、CAW 和 CSW、I/O 主程序和通道程序的编写、通道的启动和 I/O 操作的执行等。

缓冲技术主要用于匹配设备和 CPU 的处理速度,I/O 的一个重要特点是使用缓冲区,常用的缓冲技术有:单缓冲、双缓冲、多缓冲。缓冲区是由 I/O 实用程序处理的,而不是由应用进程控制的。

对系统性能产生重要影响的是磁盘 I/O,为了提高磁盘 I/O 的性能,广为使用的有两种方法:磁盘驱动调度和磁盘缓冲区。磁盘驱动算法有:电梯调度、最短查找时间优先、扫描、分步扫描、单向扫描等调度算法。磁盘缓冲区通常在主存区域开辟,作为磁盘块在磁盘与其余主存区之间的高速缓冲。由于程序局部性原理,磁盘缓冲的使用可以大幅度减少磁盘与主存储器之间的 I/O 传送的次数。

独立磁盘冗余阵列 RAID 采用一组容量较小的、独立的、可并行工作的磁盘驱动器组成阵列来代替单一的大容量磁盘,再加入冗余技术,数据能够用多种方式进行组织和分布存储,于是,独立的 I/O 请求能够被并行处理,数据分布的单个 I/O 请求也能并行地从多个磁盘驱动器同时存取数据,从而改进 I/O 性能和系统的可靠性。

SPOOLing 系统是把一个物理设备虚拟化成多个虚拟(逻辑)设备的技术,能够用共享设备来模拟独享设备。在中断机制和通道硬件的支撑下,操作系统采用多道程序设计技术,合理地分配和调度各种资源,实现外部设备的联机操作。SPOOLing 系统主要由预输入、并管理和缓输出所组成,已用于打印控制和电子邮件收发等许多场合。

习 题 五

一、思考题

1. 试述设备管理的基本功能。
2. 试述各种 I/O 控制方式及其主要优、缺点。
3. 试述直接存储器存取(DMA)传输信息的工作原理。
4. 大型机常常采用通道实现信息传输,试问什么是通道?为什么要引入通道?
5. I/O 软件主要涉及哪些问题?简单说明之。
6. 试述 I/O 中断的类型及其功能。
7. 试述 I/O 系统的层次及其功能。
8. 外部设备与 CPU 并行工作的基础是什么?
9. 什么是通道命令字(CCW)和通道程序?
10. 什么是通道地址字(CAW)和通道状态字(CSW)?
11. 试述采用通道技术时 I/O 操作的全过程。
12. 为什么要引入缓冲技术?其基本思想是什么?
13. 试述常用的缓冲技术。
14. 什么是驱动调度?有哪些常用的驱动调度技术?
15. 外部设备分为哪些类型?各类设备的物理特点是什么?

16. 解释:设备类、设备相对号、设备绝对号。
17. 解释:设备的静态分配、设备的动态分配。
18. 假定一种分配算法既能按指定的“设备类”,又能按指定的“设备号”进行分配,试给出这种设备分配的流程图。
19. 什么是“井”?什么是输入井和输出井?
20. 井管理程序有什么功能?它是如何工作的?
21. 什么是虚拟设备?实现虚拟设备的主要条件是什么?
22. 什么原因使得旋转型设备比顺序型设备更适宜于共享?
23. 试述 SPOOLing 系统和作业调度之间的关系。
24. 操作系统提供 SPOOLing 功能后,系统在单位时间内所处理的作业数是否增加?为什么?每个作业的周转时间是延长了还是缩短了?为什么?
25. 为什么 SPOOLing 又称为假脱机技术?
26. SPOOLing 是如何把独占型设备改造成共享设备的?
27. 试述 Windows 2003 系统的结构和模型。
28. Windows 2003 支持哪些类型的设备驱动程序?
29. Windows 2003 核心态设备驱动程序由哪些例程所组成?
30. 设单缓冲情况下,磁盘把一块数据输入缓冲区所花费的时间为 T ;系统从缓冲区将数据传送到用户区所花费的时间为 M ;处理器处理这块数据所花费的时间为 C ,试证明系统对一块数据的处理时间为 $\max(C, T) + M$ 。
31. 为什么要引入设备独立性?如何实现设备独立性?
32. 以 IBM 370 系列机为例,说明大型机系统的 I/O 执行过程。
33. 目前常用的磁盘驱动调度算法有哪几种?分别适用于何种数据应用场合?
34. 假如对磁盘空间进行连续分配,试讨论其优、缺点。
35. 定时紧缩磁盘空间会带来哪些好处?
36. 块设备文件和字符设备文件的本质区别是什么?
37. 设备分配中可能出现死锁吗?为什么?

二、应用题

1. 旋转型设备上信息的优化分布能够减少为若干 I/O 服务的总时间。设磁鼓分为 20 个区,每区存放一条记录,磁鼓旋转一周用时 20 ms,读取每条记录平均用时 1 ms,之后经 2 ms 处理,再继续处理下一条记录。在当前磁鼓位置未知的情况下:
 - (1) 顺序存放记录 1、记录 2、…、记录 20 时,试计算读出并处理 20 条记录的总时间;
 - (2) 给出优化分布 20 条记录的一种方案,使得总处理时间缩短,计算出这个方案所花费的总时间。
2. 现有如下请求队列:8,18,27,129,110,186,78,147,41,10,64,12;试用查找时间最短优先算法计算处理所有请求所移动的总柱面数,假设磁头的当前位置在磁道 100。
3. 在第 2 题中,分别按照升序和降序移动,讨论电梯调度算法计算处理所有存取请求移动的总柱面数。
4. 某文件是连接文件,由 5 条逻辑记录组成,每条逻辑记录的大小与磁盘块大小相等,均为 512 B,并依次存放在 50、121、75、80、63 号磁盘块上。现要读出文件的 1 569 B,问访问哪一个磁盘块?

5. 对磁盘存在如下 5 个请求：

请求	柱面号	磁头号	扇区号
1	7	2	8
2	7	2	5
3	7	1	2
4	30	5	3
5	3	6	6

假如当前磁头位于 1 号柱面。试分析对这 5 个请求如何调度，可使得磁盘的旋转圈数最少？

6. 现有含 40 个磁道的盘面，编号为 0~39，当磁头位于第 11 磁道时，顺序到来如下磁道请求：(磁道号)1, 36, 16, 34, 9, 12；试用：(1)先来先服务算法 FCFS；(2)最短查找时间优先算法 SSTF；(3)扫描算法 SCAN 等 3 种磁盘驱动调度算法，计算其各自要来回穿越多少磁道？

7. 假定磁盘有 200 个柱面，编号 0~199，当前存取臂的位置在 143 号柱面上，并刚刚完成 125 号柱面的服务请求。如果请求队列的先后顺序是：86, 147, 91, 177, 94, 150, 102, 175, 130；试问：为了完成上述请求，下列算法存取臂所移动的总量是多少？并计算存取臂移动的顺序。

- (1) 先来先服务算法 FCFS；
- (2) 最短查找时间优先算法 SSTF；
- (3) 扫描算法 SCAN；
- (4) 电梯调度算法。

8. 除了 FCFS 之外，所有磁盘调度算法都不公平，如造成某些请求饥饿。试分析：

- (1) 为什么不公平？
- (2) 提出一种公平性调度算法。
- (3) 为什么公平性在分时系统中是一个很重要的指标？

9. 若磁头的当前位置是第 100 号柱面，磁头正在向磁道号减小的方向移动。现有磁盘读写请求队列，柱面号依次为：

190, 10, 160, 80, 90, 125, 30, 20, 29, 140, 25

若采用最短寻道时间优先算法和电梯调度算法，试计算出各种算法的移臂所经过的柱面数？

10. 若磁头的当前位置是第 100 号柱面，磁头正在向磁道号增加的方向移动。现有磁盘读写请求队列，柱面号依次为：

23, 376, 205, 132, 19, 61, 190, 398, 29, 4, 18, 40

若采用先来先服务算法、最短寻道时间优先算法和扫描算法，试计算出各种算法中的移臂所经过的柱面数？

- 11. 设有长度为 L 字节的文件存放在磁带上，若规定磁带物理块长为 B 字节，试问：
 - (1) 存放此文件需要多少块？
 - (2) 若一次启动磁带机交换 K 块，则存取这个文件需要执行多少次 I/O 操作？
- 12. 某磁盘共有 200 个柱面，每个柱面有 20 个磁道，每个磁道有 8 个扇区，每个扇区为 1 024 B。如果驱动程序所接到的访问请求是读出 606 块，计算此信息块的物理位置。
- 13. 假定磁带记录密度为每英寸 800 个字符，每条逻辑记录是 160 个字符，块间隙为 0.6 英寸。现有 1 500

条逻辑记录需要存储,尝试:

- (1) 计算磁带利用率。
- (2) 1 500 条逻辑记录占用多少磁带空间?
- (3) 若要使磁带空间利用率不少于 50%,至少应以多少条逻辑记录为一组?

14. 假定磁带记录密度为每英寸 800 个字符,每条逻辑记录为主 200 个字符,块间隔为 0.6 英寸。现有 3 200 条逻辑记录需要存储,如果不考虑存储记录,则不成组处理和以 8 条逻辑记录为一组的成组处理时磁带的利用率各是多少?在两种情况下,3 200 条逻辑记录需要占用多少磁带空间?

15. 一个软磁盘有 40 个柱面,查找移过每个柱面花费 6 ms。若文件信息块凌乱地存放,则相邻逻辑块平均间隔 13 个柱面。但若优化存放,相邻逻辑块平均间隔 2 个柱面。如果搜索延迟为 100 ms,传输速度为每块 25 ms,现问在两种情况下传输长度为 100 块的文件各需多长时间?

16. 磁盘请求以 10、22、20、2、40、6、38 柱面的次序到达磁盘驱动器,如果磁头当前位于柱面 20,若查找移过每个柱面要花费 6 ms,用以下算法计算查找时间:(1)FCFS;(2)最短查找优先;(3)电梯调度算法(正向柱面大的方向移动)。

17. 假定在某移动臂磁盘上,刚刚处理了访问第 75 号柱面的请求,目前正在第 80 号柱面读取信息,并且有下述请求序列等待访问磁盘。

请求次序	1	2	3	4	5	6	7	8
欲访问的柱面号	160	40	190	188	90	58	32	102

试用:(1) 电梯调度算法;(2) 最短查找时间优先算法;分别列出实际处理上述请求的次序。

18. 在计算机系统中,屏幕的显示分辨率为 640×480 像素,若要存储一屏 256 彩色的图像,需要多少字节的存储空间?

19. 某操作系统中,CPU 用 1 ms 来处理中断请求,其他 CPU 时间用于计算,若时钟中断频率为 100 Hz,试计算 CPU 的利用率。

20. 某操作系统采用单缓冲技术传送磁盘数据。设从磁盘将数据传送到缓冲区所用的时间为 T_1 ,将缓冲区中数据传送到用户区所用的时间为 T_2 ,CPU 处理数据所用的时间为 T_3 ,试问系统处理此数据所用的总时间是多少?

21. 某操作系统采用双缓冲技术传送磁盘数据。设从磁盘将数据传送到缓冲区所用的时间为 T_1 ,将缓冲区中数据传送到用户区所用的时间为 T_2 (假设 $T_2 \ll T_1$),CPU 处理数据所用的时间为 T_3 ,试计算处理此数据,系统所用的总时间。

22. 有一个计算机系统,其磁盘容量为 520 MB,盘块大小为 1 KB,其中前 4 MB 用于存储 inode 等,后 10 MB 用做对换区。采用成组链接法管理外存储器空间,每组 100 个盘块。试绘制辅助存储器尚未使用的成组链接图。

23. 在流水线通信机制中,试用 P、V 操作描述读写进程访问管道文件的过程。假设管道文件的大小为 10 KB。

24. 若某机房有两台打印机,其中一台打印机尽量满足系统的打印要求,只有在系统不需要时才可以被一般用户所共享;另一台打印机直接作为网络共享打印机,供一般用户使用。

- (1) 请给出利用 SPOOLing 技术所实现的系统组成;
- (2) 使用记录型信号量机制实现对这两台打印机的使用过程的管理,要求写出所需设计的数据结构和

算法。

25. 磁盘组共有 n 个柱面, 编号顺序为 $0, 1, 2, \dots, n - 1$; 共有 m 个磁头, 编号顺序为 $0, 1, 2, \dots, m - 1$; 每个磁道内的 k 个信息块从 1 开始编号, 依次为 $1, 2, \dots, k$ 。现用 x 表示逻辑磁盘块号, 用 a, b, c 分别表示任意逻辑磁盘块的柱面号、磁头号、磁道内的块号, 则 x 与 a, b, c 可通过如下公式进行转换:

$$\begin{aligned}x &= k \times m \times a + k \times b + c \\a &= (x - 1) \text{DIV}(k \times m) \\b &= ((x - 1) \% (k \times m)) \text{DIV } k \\c &= ((x - 1) \% (k \times m)) \% (k + 1)\end{aligned}$$

若某磁盘组为 $n = 200, m = 20, k = 10$; 试问:

- (1) 柱面号为 185、磁头号为 12、磁道内的块号为 5 的磁盘块的逻辑磁盘块号是多少?
- (2) 逻辑磁盘块号为 1200, 其所对应的柱面号、磁头号及磁道内的块号是多少?
- (3) 若每一磁道内的信息块从 0 开始编号, 编号依次为 $0, 1, \dots, k - 1$, 其余条件同前, 试写出 x 与 a, b, c 之间的转换公式。



第六章

文件管理

文件系统是操作系统中负责存取和管理信息的模块,它采用统一方法管理用户信息和系统信息的存储、检索、更新、共享和保护,并为用户提供一整套行之有效的文件使用及操作方法。“文件”这一术语不但反映用户概念中的逻辑结构,而且同存放它的辅助存储器的存储结构紧密相关,所以,必须从逻辑文件和物理文件两个侧面来观察文件。对于用户而言,可按照需要并遵循文件系统的规则来定义文件信息的逻辑结构,由文件系统提供“按名存取”方式来实现对文件信息的存储和检索;对于系统而言,必须采用特定的数据结构和有效算法,实现文件的逻辑结构到存储结构的映射,实现对文件存储空间和文件信息的管理,提供多种存取方法。例如,用户希望与具体的存储硬件无关,使用路径名、文件名、文件内位移就可执行数据的读、写、修改、删除操作;而作为实现这些功能的文件系统来说,其工作与存储硬件紧密相关,是将用户的文件操作请求转化为对磁盘上的信息按照所在的物理位置进行寻址、读写和控制。所以,文件系统的功能就是在逻辑文件与物理文件、逻辑地址与物理地址、逻辑结构与物理结构之间实现转换,使得存取速度快、存储空间利用率高、数据可共享、安全可靠性好。

文件系统的主要功能有:文件的按名存取,实现从逻辑文件到物理文件的转换;文件目录的建立和维护;文件的查找和定位;文件存储空间的分配和管理;提供文件的存取方法和文件存储结构;实现文件的共享、保护和保密;提供一组易用的文件操作和命令;提供与设备管理交互的统一接口。

6.1 文 件

6.1.1 文件概念

早期计算机系统中,用户自行管理辅助存储器,按照物理地址安排信息,记录信息的分布情况,组织数据输入/输出,繁琐复杂,极易出错,可靠性差。大容量直接存取存储器的问世为建立文件系统提供良好的物质基础,多道程序和分时系统的出现,使得多个用户及操作系统都要共享大容量辅助存储器。因而,现代操作系统中都配备文件系统,以适应系统管理和用户使用信息资源的需要。对计算机系统中信息资源的管理形成了操作系统的文件系统。

文件是由文件名所标识的一组信息的集合,文件名是字母或数字组成的字母数字串,其格式和长度因系统而异。

组成文件的信息可以是各式各样的:源程序、数据、编译程序可各自组成一个文件,操作系统提供文件系统后,首先,便于用户使用,无须记住信息存放在辅助存储器中的物理位置,无须考虑如何将信息存放到介质上,只要知道文件名,给出有关的操作要求便可访问,实现了“按名存取”。特别地,当文件存放位置发生改变,甚至更换文件的存储设备,对使用者也不会产生丝毫影响;其次,文件安全可靠,由于用户通过文件系统才能实现对文件的访问,而系统能提供各种安全、保密和保护措施,可防止对文件信息的有意或无意的破坏或窃用。此外,系统能有效地利用存储空间,优化安排不同属主文件的位置;如果在文件使用过程中出现设备故障,系统可组织重执或恢复,对于因硬件失效而可能造成的信息破坏,可组织转储以加强可靠性;最后,文件系统还能提供文件共享功能,不同的用户可使用同名或异名的同一个文件,这样,合理利用文件存储空间,缩短传输信息的交换时间,提高文件空间的利用率。把数据组织成文件形式加以管理和控制是计算机数据管理的重大发展。

6.1.2 文件命名

文件是存储设备的一种抽象机制,这一机制中最重要的是文件命名。系统按名管理和控制文件信息,进程创建文件时必须给出文件名,以后此文件将独立于进程存在直到它被显式地删除。当其他进程要使用文件时,必须显式地指出相应的文件名。

各个操作系统的文件命名规则略有不同,文件名的格式和长度因系统而异。一般来说,文件名由文件名称和扩展名两部分组成,前者用于识别文件,后者用于区分文件类型,中间用“.”分隔开来。它们都是字母或数字所组成的字母数字串,操作系统还提供通配符“?”和“*”,便于对一组文件进行分类或操作。

早期操作系统中,文件名称的长度限于1~8个字符,扩展名长度限于0~3个字符,现在文件名最长可达255个字符。Windows的文件名不区分字母大小写。相反地,UNIX/Linux却区分字母大小写。文件名的可用字符包括字母、数字及特殊符号,每个操作系统均对可用字符作一定的限制,像Windows的文件名称和扩展名不能使用\、/、<、>、|和”等字符。扩展名用于定义和区分文件类型,说明文件的内容和内部格式。系统有一些约定的扩展名,例如,.txt指明纯文本文件,.exe表示可执行浮动二进制代码文件,.bat表示批命令文件,.obj表示编译或汇编生成的目标文件等。定义文件的扩展名是一种习惯,并不源于何种系统,已被大多数用户所默认。例如,汇编语言源程序的扩展名取“.asm”,Fortran77源程序的扩展名取“.f77”,C语言编译器要求被编译的源程序的扩展名取“.c”,一个能被Web浏览器识别的超文本置标语言信息文件的扩展名应取“.html”。

6.1.3 文件类型

按照各种方法对文件进行分类,如按用途分成:系统文件、库文件和用户文件;按保护级别分

成：只读文件、读写文件和不保护文件；按信息流向分成：输入文件、输出文件和输入输出文件；按存放时限分成：临时文件、永久文件、档案文件；按数据类型分成：源程序文件、目标文件和可执行文件；按设备类型分成：磁盘文件、磁带文件、软盘文件。此外，还可按文件的逻辑结构或物理结构进行分类。在现代操作系统中，不但信息组织成文件，对设备的访问也都基于文件进行。例如，打印一批数据就是向打印机设备文件写数据，从键盘接收一批数据就是从键盘设备文件读数据。

UNIX/Linux 操作系统支持以下不同类型的文件。

(1) 普通文件：源程序文件、数据文件、目标代码文件及操作系统文件、库文件、实用程序文件都是普通文件，它们通常存储在磁盘上。

(2) 目录文件：是由文件目录所构成的用来维护文件系统结构的系统文件。普通文件的查找依赖于目录文件，由于它也是由字符信息所组成的文件，故可进行与普通文件类似的读写等各种目录操作。

(3) 特别文件：指各种外部设备文件，又可分为：块设备文件，如存放在磁盘或光盘等块设备上的文件；字符设备文件，如终端、打印机等设备文件；FIFO 命名管道文件，socket 套接字文件。

一般来说，普通文件包括 ASCII 文件或者二进制文件。ASCII 文件由多行正文组成，在 Windows 等系统中，每行均以回车换行结束，整个文件以 Ctrl+Z 结束；在 UNIX 等系统中，每行以换行结束，文件以 Ctrl+D 结束。ASCII 文件的最大优点是可以原样显示和打印，也可用通常的文本编辑器进行编辑。另一种正规文件是二进制文件，它具有一定的内部结构，组织成字节流，如可执行文件是指令和数据流，记录式文件是逻辑记录流。块设备文件和字符设备文件合称特殊文件，它们与普通文件类似，也需要查找目录、验证访问权限、进行读写操作等。把所有 I/O 设备统一在文件系统下，有利于系统管理，方便用户使用。

6.1.4 文件属性

大多数操作系统设置专门的文件属性用于文件的管理控制和安全保护，它们虽非文件的信息内容，但对于系统的管理和控制是十分重要的。这组属性包括：

- (1) 文件基本属性：文件名称和扩展名、文件属主 ID、文件所属组 ID 等。
- (2) 文件类型属性：如普通文件、目录文件、系统文件、隐式文件、设备文件、pipe 文件、socket 文件等。也可按文件信息分为：ASCII 码文件、二进制码文件等。
- (3) 文件保护属性：规定谁能够访问文件，以何种方式访问。常用的文件访问方式有可读、可写、可执行、可更新、可删除等；上锁标志和解锁标志；有的系统还为文件设置口令，用做保护。
- (4) 文件管理属性：如文件创建时间、最后访问时间、最后修改时间等。
- (5) 文件控制属性：逻辑记录长、文件当前长、文件最大长，关键字位置、关键字长度、信息位置、文件打开次数等。

文件保护属性用于防止文件被破坏，包括两个方面：一是防止因系统发生崩溃所造成的文件破坏；二是防止文件主和其他用户有意或无意的非法操作所造成的文件不安全性。为了防止系

统崩溃造成文件破坏,定时转储是经常采用的方法,系统管理员每隔一日、或一周、或一月,把需要保护的文件保存到另一种介质上,以备数据破坏后恢复使用。由于需要备份的数据文件非常多,增量备份是必需的,为此专门为文件设置档案属性,用以指明文件是否被备份过。操作系统往往也提供备份和转储工具以便用户转储,如 UNIX/Linux 系统的 compress 命令、tar 命令、tar 命令等;第三方公司也提供备份工具,比较著名的有 arj、winzip 等。定时转储的成本虽然低,但不能做到完全的数据恢复,这对于实时应用和商业应用而言是不够的,为此引入多副本技术(如磁盘镜像),把重要的数据存放于不同的物理磁盘,乃至不同的联网计算机上,当系统崩溃造成文件破坏后,可从文件副本中读取数据。

访问控制则用于防止文件主和其他用户有意或无意的非法操作所造成的文件不安全性,这往往需要通过操作系统的安全策略和安全机制来控制,第七章将对此作进一步介绍。

6.1.5 文件存取方法

存取方法是指读写文件存储器上的物理记录的方法。由于文件类型不同,用户的使用要求不同,因而需要操作系统提供多种存取方法来满足用户要求。常用的存取方法如下。

1. 顺序存取

无论是无结构字节流文件,还是有结构记录式文件,存取操作都在上次的基础上进行。系统设置读写两个位置指针,指向要读出或写入的字节位置或记录位置。读操作总是读出位置指针所指向的若干字节或下一条记录,写操作将若干字节或一条记录写到写指针所指的位置。根据读出或写入的字节个数或记录号,系统自动修改相应指针的值。

允许对固定长记录的顺序文件采用随机访问,当要读出第 i 条记录时,其逻辑地址可由记录长 $\times i$ 得到。顺序存取主要用于磁带文件,但也适用于磁盘上的顺序文件。

对于可变长记录的顺序文件,每条记录的长度信息存放于记录的第一个字段中,其存取操作分两步进行。读出时,根据读指针值先读出一条物理记录到缓冲区,得到记录长后,再取出当前逻辑记录;写入时,先在缓冲区装配逻辑记录和记录长,再写入写指针所指向的物理记录。由于顺序文件是顺序存取的,可采用成组和分解操作来加快文件的 I/O 操作。

2. 直接存取

很多应用场合要求快速地以任意次序直接读写某条记录。例如,在航空订票系统中,把特定航班的所有信息用航班号作为标识,存放在物理块中,当用户预订某航班时,需要直接将此次航班的信息取出,直接存取方法便适合于这类应用,它通常用于磁盘文件。

为了实现直接存取,一个文件可看做由顺序编号的物理块组成,这些块常常划分成等长,作为定位和存取的最小单位,于是,用户可请求读块 22,然后,写块 48,再读块 9,等等。直接存取文件对于读或写块的次序没有任何限制。用户向操作系统提供的是相对块号,它是相对于文件开始位置的位移量,而绝对块号则由系统经换算得到。

对于记录式文件,要为每个文件记录指定关键字,可通过关键字映射来直接检索和存取文件信息,这也是各类应用中用得最多的一种直接存取方式。

3. 索引存取

这是基于索引文件的存取方法,由于文件中的记录不按位置而是按其记录名或记录键来编址,所以,用户提供记录名或记录键之后,先按名搜索,再查找所需要的记录。采用记录键时应按某种顺序存放,例如,按字母先后次序来排序,对于这种文件,除了可采用按键存取外,也可采用顺序存取或直接存取的方法,信息块的地址都可以通过查找记录键而换算出来。在实际的系统中,大都采用多级索引,以加速记录的查找过程。

文件目录

文件系统通常采用分层结构实现,大致分为三层:文件管理、目录管理和磁盘主存映射管理。文件管理层实现文件的逻辑结构,为用户提供各种文件系统调用,及文件访问权限的设置等工作;目录管理负责查找文件描述符,进而找到需要访问的文件,并进行访问权限检查等工作;磁盘主存映射管理将文件的逻辑地址转换成磁盘的物理地址,即由逻辑块号找到柱面号、磁道号和扇区号,具体的数据传输操作由设备管理实现。

6.2.1 文件控制块、文件目录与目录文件

文件系统给每个文件建立唯一的管理数据结构,称为文件控制块(File Control Block,FCB),一个文件由两部分组成:FCB和文件体(文件信息)。FCB一般应包括以下文件属性信息:

- (1) 文件标识和控制信息:文件名、用户名、文件主存取权限、授权者存取权限、文件口令、文件类型等。
- (2) 文件逻辑结构信息:文件的逻辑结构,如记录类型、记录个数、记录长度、成组因子数等。
- (3) 文件物理结构信息:文件所在设备名、文件物理结构类型、记录存放在辅助存储器的盘块号或文件信息首块盘块号,也可指出文件索引所在的位置等。
- (4) 文件使用信息:共享文件的进程数、文件修改情况、文件最大长度和当前大小等。
- (5) 文件管理信息:文件建立日期、最近修改日期、最近访问日期、文件保留期限、记账信息等。

有了 FCB 就可以方便地实现文件的“按名存取”。每当创建一个文件时,系统就要为其建立一个 FCB,用来记录文件的属性信息,存取此文件时,先找到其 FCB,再找到文件信息盘块号或首块物理位置就能存取文件信息。

为了加快文件的查找速度,通常把 FCB 集中起来进行管理,组成文件目录。文件目录包含许多目录项,目录项有两种,分别用于描述子目录和文件的 FCB。目录项的格式按统一标准定义,全部由目录项所构成的文件称为目录文件。与普通文件不同的是,目录文件永远不会空,它至少包含两个目录项:当前目录项“.”和父目录项“..”。文件目录的基本功能是将文件名转换成此文件信息在磁盘上的物理位置。实际上,文件系统就是文件和目录的层次结构和集合,是用来管理文件系统结构的系统文件,其基本功能之一就是负责文件目录的建立、维护和检索,要求所

编排的目录便于查找、防止冲突，目录的检索方便、迅速。

文件系统的信息空间可认为是由一系列逻辑块所组成的，每个逻辑块的长度为 512 B 的整数倍数。不同文件系统的逻辑块大小允许不同，在系统生成时指定。至于逻辑块号到物理块号的转换则由独立于设备的操作系统软件负责，这些逻辑块被用于存放文件数据、文件目录或文件管理信息。UNIX/Linux 系统采用巧妙的文件目录建立方法，能够减少检索文件所需访问的磁盘物理块数。FCB 中的文件名和其他管理信息分开，其他信息单独组成一个数据结构，称为索引节点 inode，此索引节点的位置由 inode 号标识。于是，目录项中仅剩下 14 B 的文件名和 2 B 的 inode 号，称为基本目录项。因此，一个磁盘块可以存放几十个基本目录项，系统对于由文件目录项组成的目录文件和普通文件同等对待，均存放在磁盘中，文件系统中的每个文件都有一个磁盘 inode 与之对应，这些 inode 被集中存放于磁盘上的 inode 区。FCB 对于文件的作用，犹如 PCB 对于进程的作用，集中这个文件的所有相关信息，找到 inode，就能获得此文件的必要信息。每个磁盘 inode 结构使用 32 B 或 64 B，包含如下信息：文件长度及文件在存储设备上的物理位置、文件主标识、文件类型（普通/目录/特别/管道文件）、存取权限、文件链接数（共享数）、文件大小（字节数）、文件创建、访问和修改时间，及 inode 节点是否空闲，其中的 addr[40] 数组用于存放文件数据或基本目录项所在的磁盘块号的索引表，每 3 个字节记录一个磁盘块号，因此索引表共 13 项。

由于磁盘 inode 记录文件的属性和相关信息，文件访问过程中会频繁地用到它，不断来回于主辅存之间引用它，当然是极不经济的。为此，UNIX/Linux 在系统所占用的主存区内开辟一张主存索引节点表，又称活动 inode 表，含有 100 个表项，每个表项称为一个活动 inode。磁盘 inode 反映文件的静态特性，活动 inode 还需反映文件的动态特性，为此增加部分信息，包括文件活动标志 i_flag（占用/修改/安装点/上锁位等）、共享索引节点数 i_count、索引节点所在设备名 i_dev、索引节点号 i_number、文件链接数 i_nlink、文件属性 i_mode 及空闲链和占用链哈希表指针。当访问某文件时，若在活动 inode 表中找不到其 inode，就申请一个空闲活动 inode，把磁盘 inode 内容复制给它，随之就可用来控制文件的读写。当用户关闭此文件后，活动 inode 的内容被回写到对应的磁盘 inode 中，然后释放活动 inode 以供它用。把 FCB 的主要内容与索引节点号分开，不仅能够加快目录检索速度，而且，便于实现文件共享。

UNIX System V 对文件目录项的内容进行扩充，共有 4 个部分：inode 号（4 B）、本记录长度、文件名长度和文件名（255 B）。

6.2.2 层次目录结构

最简单的文件目录是一级目录结构，所有 FCB 排列在一张线性表中，其缺点是文件重名和文件共享问题难以解决。实际上，所有文件系统都支持多级层次结构，根目录是唯一的，每一级目录可以是下一级目录的说明，也可以是文件的说明，从而形成树状目录结构。图 6.1 给出 Linux 和 Windows 目录层次结构，它是一棵倒置的有根树，树根是根目录；从根向下，每个树枝是子目录；而树叶是文件。树状多级目录结构有许多优点，较好地反映现实世界中具有层次关系的数据集合，确切地反映系统内部文件的分支结构；不同的文件可以重名，只要它们不位于同一末

端子目录中,易于规定不同层次或子目录中文件的不同存取权限,便于文件的保护、保密和共享等,有利于系统的维护和查找。

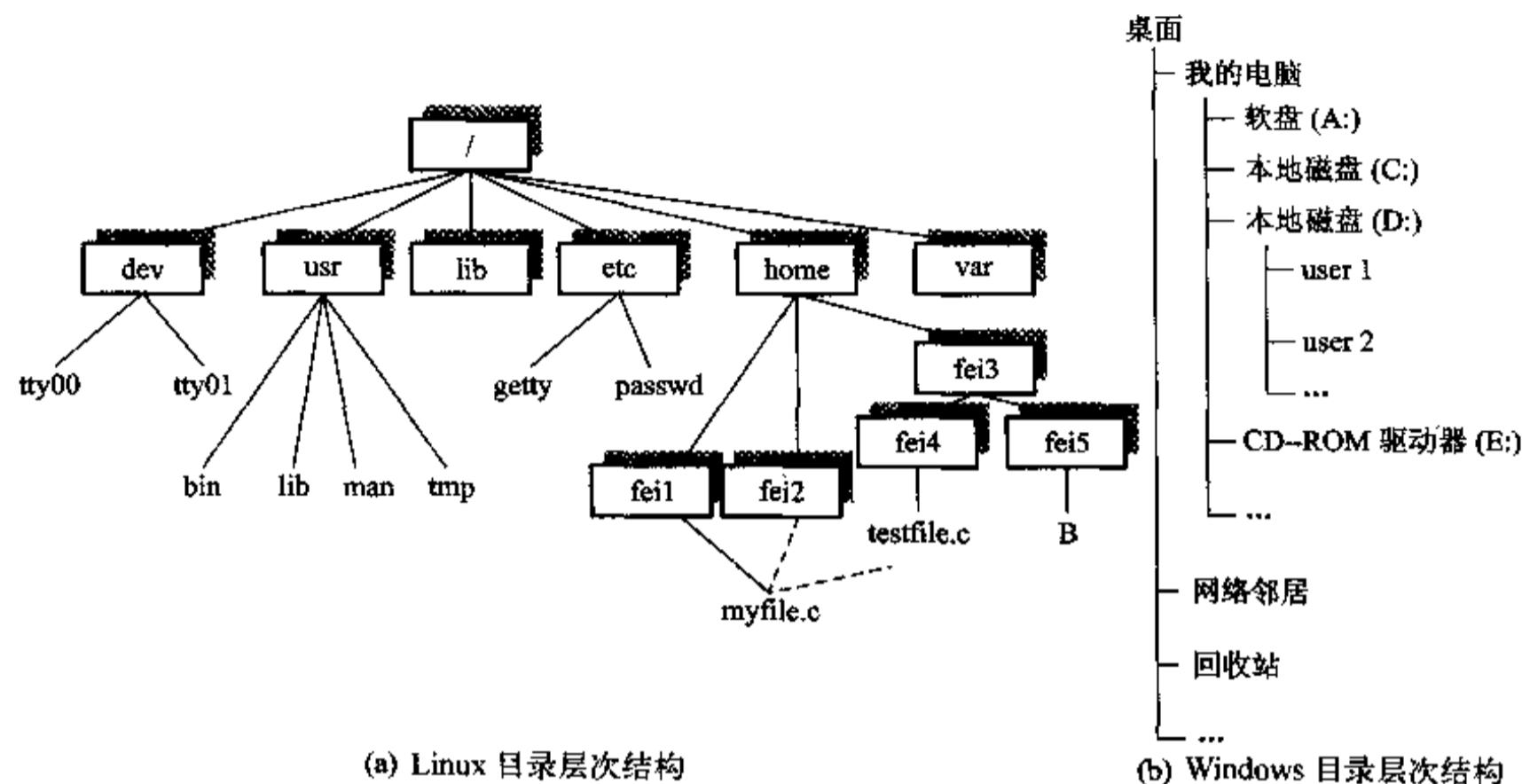


图 6.1 目录结构

在树状目录结构中,一个文件的全名包括从根目录起至文件为止,在通路上所遇到的所有子目录路径,各子目录名之间用斜线隔开,子目录名所组成的部分又称为路径名。文件可在目录中被聚合成组,文件目录自身也被作为文件存储,在很多方面可以类似子文件一样处理。

Linux 的文件系统也采用层次多级目录结构,在根目录下包含:dev 设备子目录;usr 不需修改的命令程序文件和库程序子目录;var 变化的文件子目录;etc 基本数据和维护实用程序子目录;home 用户文件主目录,等等。如图 6.1(a)所示是 Linux 树状目录结构,以方框代表目录文件,未加框者代表一般文件或特殊文件。Windows 文件系统的根目录称为“桌面”,对应于用户屏幕,其上显示子目录、程序和文件的图标。其中,最重要的目录之一是“我的电脑”,它包括不同存储设备的子目录,如软磁盘驱动器(A:)、硬盘驱动器(C:)和(D:)、CD - ROM 驱动器(E:),目录 C 包含许多操作系统程序,目录 D 包含用户目录 user1 和 user2 及各类用户信息。桌面上所出现的其他目录有“网络邻居”和“回收站”等。如图 6.1(b)所示是 Windows 目录层次结构。

如果规定每个文件都只有一个父目录,称为纯树型目录结构。其缺点是,文件的共享不是对称的,父目录有效地拥有此文件,其他被授权的用户必须经过属主目录才能对文件进行访问。有向无环图目录尽管允许文件有多个父目录而破坏树状特性,但不同的用户可以对称方式实现文件共享,即可能属于不同用户的多个目录,使用不同的文件名访问和共享同一个文件。有向无环图目录结构的维护比纯树型目录结构要复杂,由于一个文件可能有多个父目录,需要为每个文件维护一个引用计数,以此记录文件的父目录个数,仅当引用计数值为 1 时,删除操作才移去文件,

否则仅仅把相关记录从父目录中删除。

Linux 系统支持多父目录,但其中一个是主父目录,它是文件拥有者,且文件被物理地存储在此目录下,其他次父目录通过 link 方式来连接和引用文件,允许任一父目录删除共享文件。图 6.1(a)中便示例这种文件共享的情形,文件 /home/fei1 是 myfile.c 的主父目录(图中以实线表示),/home/fei2 和 /home/fei3 /fei4 均为文件 myfile.c 的次父目录(图中以虚线表示)。

Windows 实现被称作“快捷方式”的多父目录连接，“快捷方式”是一些指向不同文件夹(子目录)和菜单之间任意复制和移动的文件及文件夹的指针，删除“快捷方式”就是删除其所对应的指针。

6.2.3 文件目录的检索

每个目录在创建时都自动含有两个特殊目录项，“.”项指出目录自身的 inode 入口，“..”项指出其父目录的 inode 入口，如图 6.2 所示。图 6.2(a)是用户概念中的目录结构，在系统中是按图 6.2(b)来实现目录链接的。父目录包含子目录意味着包含一个指向子目录的 inode 链接“.”，反之，子目录具有父目录也通过指向 inode 来实现，它有一项“..”，这是父目录的保留名字。如何知道是文件系统的根目录呢？通常采用“.”和“..”都指向同一个 inode 的方法来表明。

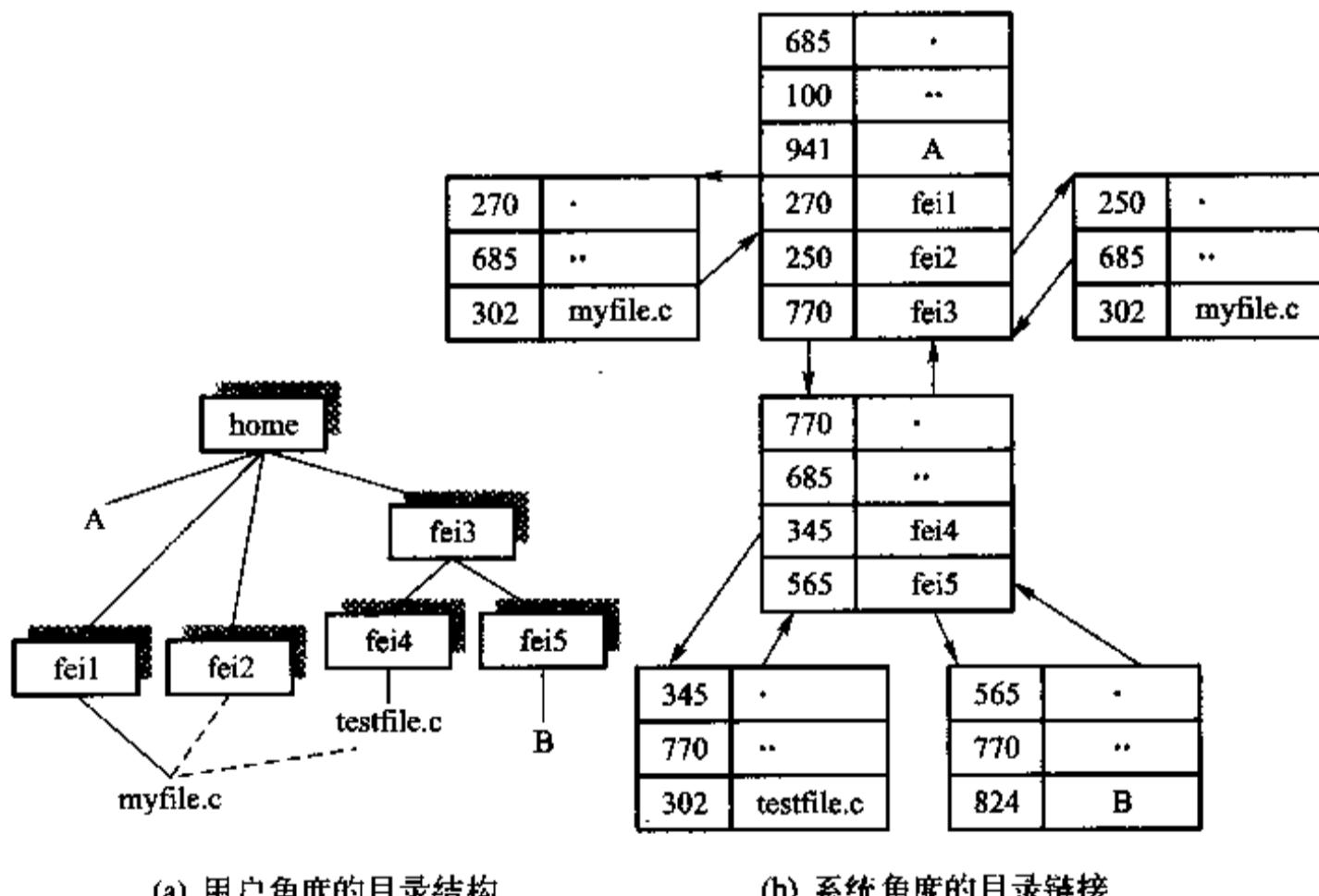


图 6.2 不同角度的目录结构

执行打开文件操作时,根据用户所提供的文件路径名,从根目录或当前工作目录开始,逐级查找路径名中的各子目录名,用其作为索引,逐层搜索各级目录文件,最后找到相匹配的文件目录项。下面是目录查找的例子,假设应用进程要打开文件/home/fei1/myfile.c,文件系统开始搜

类似于文件操作，文件系统也提供一系列处理文件目录的系统调用和操作命令，有的系统甚至允许应用程序像对待文件一样打开、读写和操作目录。与文件目录有关的操作有：创建目录、删除目录、打开目录、关闭目录、读目录、列目录、改变目录、移动目录、重命名目录、改变保护、链接和删除链接目录等。

6.3 文件组织与数据存储

6.3.1 文件的存储

目前广泛使用的文件存储介质是磁盘、光盘和磁带，一盘磁带、一张光盘、一个硬盘分区都称为一卷。卷是存储介质的物理单位，块是存储介质上连续信息所组成的一个区域，也称物理记录。块是主存储器和辅助存储器进行信息交换的物理单位，每次总是交换一块或整数块信息。决定块的大小要考虑到用户使用方式、数据传输效率和存储设备类型等多种因素，不同类型的存储介质，其块的长短各不相同；同一类型的存储介质，其块的长短也可以不同。有些外部设备出于启停机械动作的要求或识别不同块的特殊需要，相邻块之间必须留有间隙，间隙是块之间不记录用户代码信息的区域。

6.3.2 文件的逻辑结构

1. 流式文件和记录式文件

文件组织是指文件中信息的配置和构造方式,应该从文件的逻辑结构和组织以及文件的物理结构和组织两方面加以考虑。文件的逻辑结构和组织是从用户的观点出发,研究用户概念中抽象的信息组织方式,这是用户所能观察到的、可加以处理的数据集合。由于数据可独立于物理环境构造,故称为逻辑结构,相关数据的集合构成逻辑文件。系统提供若干操作以便使用者构造文件,这样,用户不必顾及文件信息的物理结构,利用文件名和有关操作就能存储、检索和处理文

件信息。当然,为了提高操作效率,对于各类设备的物理特性及其适宜的文件类型仍应有所了解,换句话说,存储设备的物理特性会影响到数据的逻辑组织和所采用的存取方法。

文件的逻辑结构分为两种基本形式:流式文件和记录式文件。

(1) 流式文件

文件内的数据不再组成记录,只是一串顺序的信息集合,称为字节流文件。流式文件中的每个字节都有一个索引,第一个字节的索引为0,第二个字节的索引为1,…,以此类推,打开文件的进程使用文件读写位置来访问文件中的特定字节。当文件打开时,文件读写位置指向首字节,每 k 个字节的读或写操作完成,则将文件读写位置加 k 。提供使用流式文件的系统调用有:`read()`/`write()`,参数包括文件名、读入/写出缓冲区地址、读入/写出字节个数;`seek()`,参数包括文件名和改变文件读写位置的值。事实上,有许多应用不要求文件内再区分记录,如用户作业的源程序就是一个顺序字符流,强制分割源程序文件为若干记录只会带来操作复杂、开销加大等缺点,因而,为了简化系统,大多数现代操作系统如Windows和UNIX/Linux只提供流式文件。

(2) 记录式文件

这是一种有结构的文件,它包含若干逻辑记录,逻辑记录是文件中按信息在逻辑上的独立含义所划分的信息单位,记录在文件中的排列按其出现次序编号,记录0,记录1,…,等等。逻辑记录的概念被应用于许多场合,如某单位财务处的工资文件中,每个职工的工资信息是一条逻辑记录,整个单位职工的工资信息,即全部逻辑记录,便组成这个单位的工资文件。IBM MVS及Macintosh操作系统既提供流式文件也提供记录式文件,为面向商业和企业应用提供有力的数据结构化支持。

从操作系统管理的角度来看,逻辑记录是文件内独立的最小信息单位,每次总是为使用者存储、检索或更新一条逻辑记录,同流式文件一样,通过文件读写位置来指定对文件信息的访问,但是在记录式文件中,文件的记录位置取代字节位置。记录式文件中有两种常用的记录组织和使用方法。

① 记录式顺序文件:文件的记录顺序生成并被顺序访问,早期计算机使用卡片机,一个文件由一叠卡片所组成,每张卡片对应于一条逻辑记录,这类文件中的逻辑记录可依次编号,并被顺序访问。提供使用记录式顺序文件的系统调用有:`getrecord()`/`putrecord()`,参数包括文件名称和读/写记录号;`seek()`,参数包括文件名称和改变文件读写位置的记录号。

② 记录式索引顺序文件:有些应用不使用顺序访问方法,例如,自动语音查询系统,每次查询请求只是针对特定的记录,而不是所有记录,应用程序不依赖于文件中记录的位置来访问指定记录。记录式索引顺序文件提供这种能力,同时它还保持着顺序访问记录的功能。这种文件使用索引表,表项包含记录键和索引指针,记录键由应用程序确定,而索引指针便指向相应的记录。提供使用记录式索引顺序文件的系统调用有:`getrecord()`,参数包括文件名称和记录键,返回一条指定记录;`putrecord()`,参数包括文件名称和记录键,文件系统在所选择的文件位置处写入指定的记录,并建立索引表项。

2. 成组和分解

逻辑记录是按信息在逻辑上的独立含义由用户所划分的单位,而块是系统划分的存储介质上连续信息所组成的区域。因此,一条逻辑记录被存放到文件存储器的存储介质上时,可能占用一块或多块,或者一个物理块包含多条逻辑记录。若干逻辑记录合并成一组,写入一块叫做记录成组,这时每块中的逻辑记录的个数称为块因子。成组操作先在系统输出缓冲区内进行,凑满一块后才将缓冲区内的信息写到存储介质上。反之,当存储介质上的一个物理块读进系统输入缓冲区后,把逻辑记录从块中分离出来的操作叫做记录的分解。例如,对于穿孔卡片,通常逻辑记录长 80 个字符,如果把存储介质上的数据块也划分成 80 B 长为一块,一张卡片的 80 B 组成一条逻辑记录,占用一个物理块,此例中,逻辑记录和物理块是等长的。假定把卡片上的数据写到磁带上,可规定磁带上的物理块长为 800 B,每块内就可存放 10 张卡片数据,这时块因子数等于 10。如果卡片上的数据存放到磁盘上,可规定磁盘存储介质的物理块长为 1 600 B,这样每块内就可容纳 20 张卡片数据,这时块因子数等于 20,后两者的逻辑记录长小于物理块长,是成组处理的例子。

记录成组和分解处理不仅节省存储空间,还能减少 I/O 操作的次数,提高系统效率。记录成组和分解的处理过程如图 6.3 所示,应用程序的第一个读请求导致文件管理将包含逻辑记录的整个物理块读入系统输入缓冲区,再把第一条逻辑记录传送到用户工作区;随后的读请求可直接从系统输入缓冲区取得相继的逻辑记录,直到块中的逻辑记录全部处理完毕,紧接着的读请求便重复上述过程。应用程序写请求的操作过程与此相反,开始的若干条命令仅将所处理的逻辑记录依次从用户工作区传送到输出缓冲区装配,当某一个写请求所传送的逻辑记录恰好填满系统缓冲区时,文件管理才发出一次 I/O 请求,将填满的系统缓冲区的内容写到存储介质的相应块中。采用成组和分解处理记录的缺点是:需要软件进行成组和分解的额外操作;需要能容纳最大块长的系统 I/O 缓冲区。

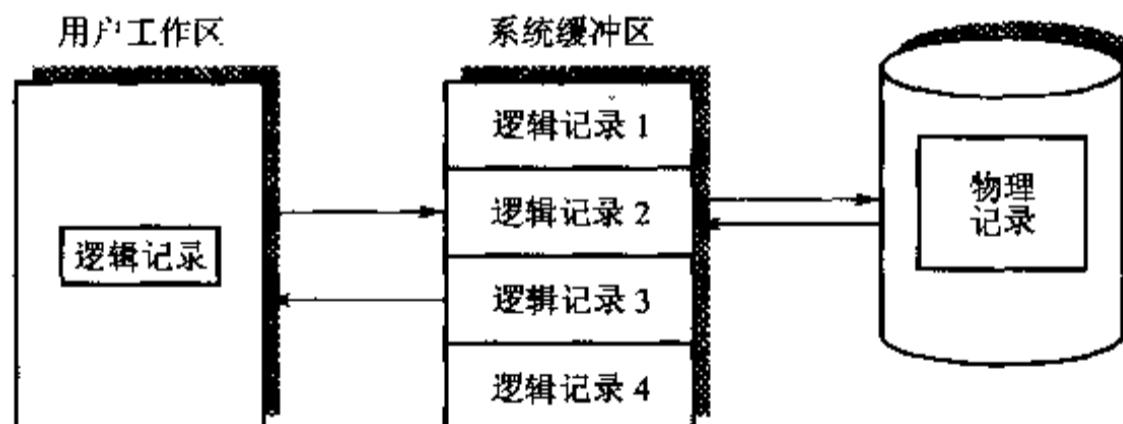


图 6.3 记录成组和分解的处理过程

3. 记录格式

记录式文件中的记录可以有不同的记录格式。提供多种记录格式是考虑数据处理和各种应用、I/O 传输效率、存储空间利用率和存储设备硬件特点等多种因素,为了进行存储、检索、处理和加工,必须按照可接受的类型和格式把信息提交给操作系统。一条逻辑记录中所有数据项长

度的总和称为此逻辑记录的长度。

记录格式就是记录内数据的排列方式。在记录式文件中,记录的长度取决于应用程序的需要,记录的格式有:定长记录、变长记录和跨块记录。为了存储和处理记录,系统必须获得记录的有关信息,在有些情况下,信息来自应用程序,其他场合则由系统自动提供。

定长记录是指记录式文件中所有的逻辑记录具有相同的长度,同时所有数据项的相对位置也是固定的。定长记录处理方便,易于控制,在传统的数据处理中被普遍采用,它可成组或不成组处理,成组时除了最末一块外,每块中的逻辑记录数是一个常数,在搜索到文件末端且最后一块的逻辑记录数小于块因子数时,操作系统就能发现并加以处理。

变长记录是指记录式文件中的逻辑记录长度不相等,但每条逻辑记录的长度处理之前能预先确定。有两种情况会造成变长记录:包含一个或多个可变长度的数据项、包含可变数目的定长数据项。虽然变长记录处理复杂,却能节省存储空间,定长记录的长度应设置为诸逻辑记录中可能出现的最大长度,这样会浪费存储空间。为了对变长逻辑记录进行存取,逻辑记录的第一字段指明单个变长逻辑记录的字节个数(也包括第一字段本身的长度),第二字段存放记录信息。

存储介质上的块通常划分成固定长度的物理块,当所处理的变长记录大于块长时,会发生逻辑记录跨越物理块的情形,这就是跨块记录。在这种情况下,逻辑记录被分割成段写到块中,读出时再作装配,段的分割和装配工作由文件系统自动实现。文件在不同物理特性的设备类型之间传送时,跨块记录特别有用。例如,可用于具有不同块长的接收类型设备和发送类型设备间的信息传送;再如在图像处理中,可存放非常长的彩色位图图像记录。跨块记录是变长记录处理的延伸,主要差别是:变长记录处理时,程序员必须知道 I/O 缓冲区的大小,而跨块记录处理时却不需要,文件系统将自动分割和装配跨块逻辑记录的信息段,而这些块不会超过 I/O 缓冲区的容量。

4. 记录键

为了方便记录式文件的组织和管理,提高文件记录的查找效率,通常对逻辑文件的每条逻辑记录至少指定一个与其对应的基本数据项,利用它可与同一文件中的其他逻辑记录区别开来。这个用于标识某条逻辑记录的数据项称为记录键,也叫做关键字,简称键。在同一个文件中,能唯一地标识某条逻辑记录的记录键称为主键。例如,工资信息记录中的职员编号、住房信息记录中的家庭住址、银行存款信息记录中的存折号都可用做相应记录的主键。在记录式文件中,主键是最重要的,但未必是唯一的,如果一个部门没有同名同姓的职员,那么,工资信息记录既允许使用职员编号,也允许使用职员姓名作为主键。

采用键来标识逻辑记录的存储格式如下:固定长逻辑记录存储时分为两个字段,第一字段是记录键,第二字段是记录信息;变长逻辑记录存储时分为 3 个字段,第一字段指明单个变长逻辑记录的记录键和记录信息的字节数(也包括第一字段本身的长度),第二字段存放记录键,第三字段是记录信息。采用键来标识的变长逻辑记录格式鲜有操作系统提供,其原因是各种数据库管理系统中都已经实现这种复杂的文件组织方式。

6.3.3 文件的物理结构

文件系统往往根据存储设备类型、存取要求、记录使用频度和存储空间容量等因素提供若干种文件存储结构,把逻辑文件以不同方式保存到物理存储设备的介质上去。所以,文件的物理结构和组织是指逻辑文件在物理存储空间中的存放方法和组织关系,这时的文件看做物理文件,即相关物理块的集合。文件的存储结构涉及块的划分、记录的排列、索引的组织、信息的搜索等许多问题,其优劣直接影响文件系统的性能。

两类方法可用来构造文件物理结构。第一类方法称计算法,设计映射算法,通常用线性计算法、杂凑法等,通过对记录键进行计算转换成对应的物理地址,从而找到所需要的记录。直接寻址文件、计算寻址文件、顺序文件均属此类。计算法的存取效率高,不必增加存储空间存放附加的控制信息,能够把分布范围较广的键均匀地映射到一个存储区域中。第二类方法称指针法,这类方法设置专门的指针,指明相应记录的物理地址或表达各记录之间的关联。索引文件、索引顺序文件、连接文件等均属此类。使用指针的优点是可将文件信息的逻辑次序与在存储介质上的物理块排列次序完全分开,便于随机存取,便于更新,能够加快存取速度。但是,使用指针要占用较多的存储空间,大型文件的索引查找要耗费较多的处理机时间,所以,究竟采用哪种文件存储结构,必须根据应用需要、响应时间和存储空间等多种因素进行权衡和折中。下面介绍几种常用的文件物理结构和组织方法。

1. 顺序文件

将文件中逻辑上连续的信息存放到存储介质的相邻物理块上形成顺序结构,叫做顺序文件,又称连续文件。这是一种逻辑记录顺序和物理块顺序完全一致的文件。记录按照出现的次序通常被顺序读出或修改,FCB中所要保存的磁盘定位信息很简单,它由第一个物理块地址和文件信息块的总块数组成。

存于磁带上的所有文件都只能是顺序文件,此外,卡片机、打印机、纸带机文件也属于这一类,是最简单的文件组织形式,在数据处理历史上最早使用。存储在磁盘上的文件也可组织成顺序文件。为了改善顺序文件的处理效率,可对顺序文件中的记录按某个或多个数据项的值从小(大)到大(小)重新排列,经排列处理后,记录有某种确定的顺序,成为有序的顺序文件,能较好地适应批处理等顺序应用。

顺序文件的优点是:顺序存取记录时速度较快,批处理文件、系统文件用得很多。采用磁带存放顺序文件时,总可以保持快速存取的优点;若以磁盘作为存储介质,顺序文件的逻辑记录也被存于邻接的物理块中,因而,顺序的磁盘文件也能像磁带文件一样进行严格的顺序处理,但当多个进程访问时,在同一时间段内另外的进程可能驱动磁头移向其他文件,这会影响访问后继物理记录的速度。顺序文件的缺点是:建立文件之前需要预先确定文件长度,以便分配存储空间;修改、插入和添加文件记录有一定的难度;对于变长记录的处理很困难;对磁盘作连续分配,会造成空闲块的浪费。

2. 连接文件

连接结构的特点是使用连接字,又称指针,来表示文件中各条记录之间的关系。如图 6.4 所示,文件信息存放在磁盘的若干物理块中,第一块文件信息的物理地址由 FCB 给出,而每块的连接字指出文件的下一个物理块位置。通常,当连接字的内容为 0 时,表示文件至本块结束。这种文件称为连接文件,又称串联文件,像输入井、输出井等都使用此类文件。

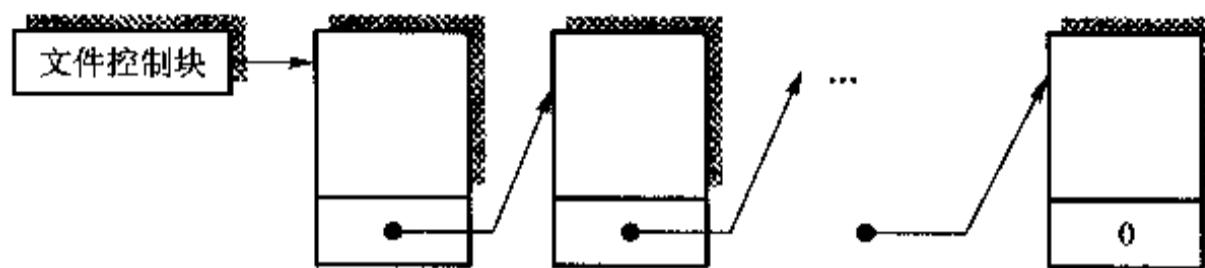


图 6.4 连接文件结构示意图

引进指向其他数据的连接表示是计算机程序设计的一种重要手段,是表示复杂数据关系的一种重要方法。使用指针可将文件的逻辑记录顺序与它所在存储空间的物理块顺序独立开来,即存放信息的物理块不必连续,借助于指针表达记录之间的逻辑关系;连接结构必须将连接字与数据信息混合存放,这样会破坏数据块的完整性;存取信息必须通过缓冲区,待获得连接字后,才能找到下一个物理块的地址,因而,仅适宜于顺序存取。连接结构恰好克服顺序结构不适宜于增、删、改的缺点,对某些操作会带来好处,但在其他方面又会失去一些性能。

下而是连接文件的一个变种,已被 OS/2 等操作系统所采用,它既能克服连接字与数据信息混合存放的缺点,又能高效地执行顺序存取。其方法是,把连接指针从数据块中分离出来,单独建立一个指针数组 PTRS[n], n 是组成磁盘连接文件物理块的总块数,每个 PTRS[i] 对应于一个磁盘块 i ,如果块 j 在文件中跟随在块 i 之后,那么,元素 PTRS[i] = j (即元素 PTRS[j] 的值为 j)。

指针数组保存在磁盘的一个专门区域中,如 0 磁道的前 k 个块,k 的大小依赖于磁盘块总数 n 和用于表示每个指针数组项 PTRS[i] 的大小。假定用 4 B 的整数记录磁盘块号,若块的大小是 1 KB,那么,每个指针块中可放 $1024/4 = 256$ 个磁盘块号。当 $k = 100$ 时,PTRS[n] 可记录 $256 \times 100 = 25600$ 个数据块的磁盘块号,此时, n 至少为 $25600 + 100$ 。例如,有一个连接文件共占用 4 个磁盘块,其 FCB 指出首块地址,从首块 6 开始,依次为块 18、块 11 和块 13,则有 FCB 指出 6、PTRS[6] = 18、PTRS[18] = 11、PTRS[11] = 13、PTRS[13] = NULL。在包含指针数组 PTRS[n] 的少量连续盘块中集中存放所有必须顺序访问的指针。为了进一步缩短定位文件信息块所需要的时间,可以把这些指针连续盘块装进主存高速缓存。

3. 直接文件

使用连接文件很容易把数据记录组织起来,但是查找某条记录需遍历链接结构,效率很低,另一种技术称为散列法或杂凑(hash)法可以解决效率问题。在直接存取存储设备上,利用哈希法将记录的关键字与其地址之间建立某种对应关系,以便实现快速存取的文件叫做直接文件或散列文件。采用散列技术需要建立哈希表,这是一个指针数组,数组通过索引访问,索引是与数据记录有

关的关键字,而记录在介质上的位置是通过对记录键施加变换来获得相应的地址。这种存储结构用在不能采用顺序组织方法、次序较乱、又需在极短时间内进行存取的场合,对于实时处理文件、目录文件、存储管理的页表查找、编译程序变量名表等十分有效。例如,在 Linux 系统中,常用散列法实现 cache,cache 用来存放重要的数据结构,内核需要高频率地快速访问这些信息。

计算寻址结构中的难点是“冲突”问题。一般说来,地址的总数和可选择的关键字之间并不存在一一对应关系,不同的关键字可能变换出相同的地址,这叫做“冲突”。某种散列算法是否成功的重要标志是将不同键映射成相同地址的几率有多大,几率越小则冲突就越少,此散列算法的性能也就越好。解决冲突的方法叫做溢出处理技术,这是设计散列文件所需考虑的主要内容。常用的溢出处理技术有:顺序探查法、两次散列法、拉链法、独立溢出区法等。

假定有一个文件系统,采用散列法来管理 FCB,以便加快文件目录的查找过程,下面来讨论 hash 文件的设计过程。首先,构造一个散列函数,它把文件名称转换为其 FCB 所在的盘块地址的索引,根据索引找到相应的物理块;然后,把这个物理块读入主存缓冲区;最后,采用线性比较法逐项查找即可。

步骤 1 构造散列函数:假定有效文件名为 8 个 ASCII 字符,不足时用空格补足,超长时截取前 8 位,构造一个简单的散列函数模 2 加“ \oplus ”,求已知文件名的各个 ASCII 字符值的模 2 加值作为此文件名的 FCB 所在物理块在目录文件中的索引 A,若 a_i 是各文件名字符的 ASCII 字符值,那么

$$A = (a_1 \oplus a_2 \oplus \cdots \oplus a_8)$$

步骤 2 建立目录文件:目录文件如图 6.5 所示采用索引结构,建立文件时由步骤 1 求出文件名的散列值 A,凡 A 值相同的文件的 FCB 都存放在同一个物理块中,磁盘的物理块号存放在索引表中的相对位置应等于 A 值。

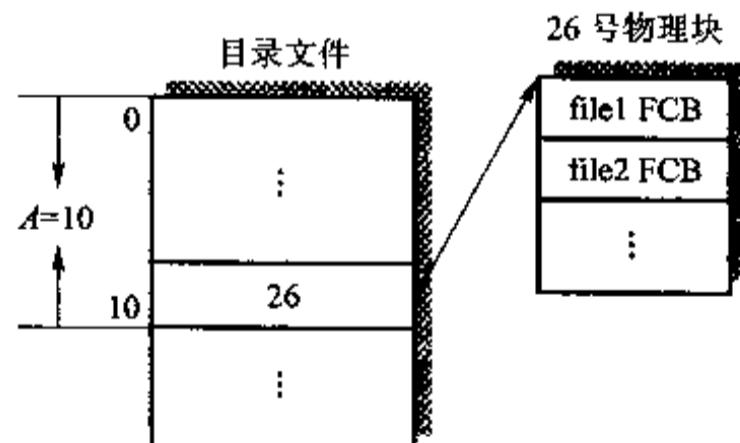


图 6.5 目录文件结构

步骤 3 查找文件:根据给定的文件名,由步骤 1 计算出此文件名的 FCB 所在物理块号在索引表中的相对位置 A,根据 A 值就可找到此 FCB 所在的物理块号,把此物理块读入主存缓冲区,再用文件名逐个比较,找出所要求的 FCB。

步骤 4 溢出处理:一个物理块中所存放的 FCB 是有限的。在建立目录文件时,如果 A 值相同的文件数目超过一个物理块所能容纳的 FCB,产生溢出,这时,系统再申请一个盘区,其物理块号将放在 $A + k$ 的索引表目中, k 是一个位移常数,往往选择质数作为位移常数;如果第二块盘区也溢出,则申请第三块,块号放在 $A + 2 \times k$ 表目中, …, 以此类推;在查找目录时,如果第一个物理块找不到,可找 $A + k$ 表目中的物理块号,读出后继续进行比较,以此类推。

A + k 的索引表目中, k 是一个位移常数,往往选择质数作为位移常数;如果第二块盘区也溢出,则申请第三块,块号放在 $A + 2 \times k$ 表目中, …, 以此类推;在查找目录时,如果第一个物理块找不到,可找 $A + k$ 表目中的物理块号,读出后继续进行比较,以此类推。

4. 索引文件

索引结构是实现非连续存储的另一种方法,适用于数据记录保存在磁盘上的文件。如图 6.6 所示,系统为每个文件建立索引表(index table),可以有不同的索引形式,一种形式是它记录组成指定文件的磁盘块号,这种索引表只是盘块号的序列,适用于流式文件;另一种形式是其索引表项包含记录键及其磁盘块号,适用于记录式文件。利用索引表来搜索记录的文件称为索引文件,索引表可存放在 FCB 中,打开文件时就可使用索引表访问文件信息,大文件的索引表很大。有些文件系统让索引表置于单独的物理块中且可驻留在磁盘的任意位置,FCB 中仅包含索引表的地址。

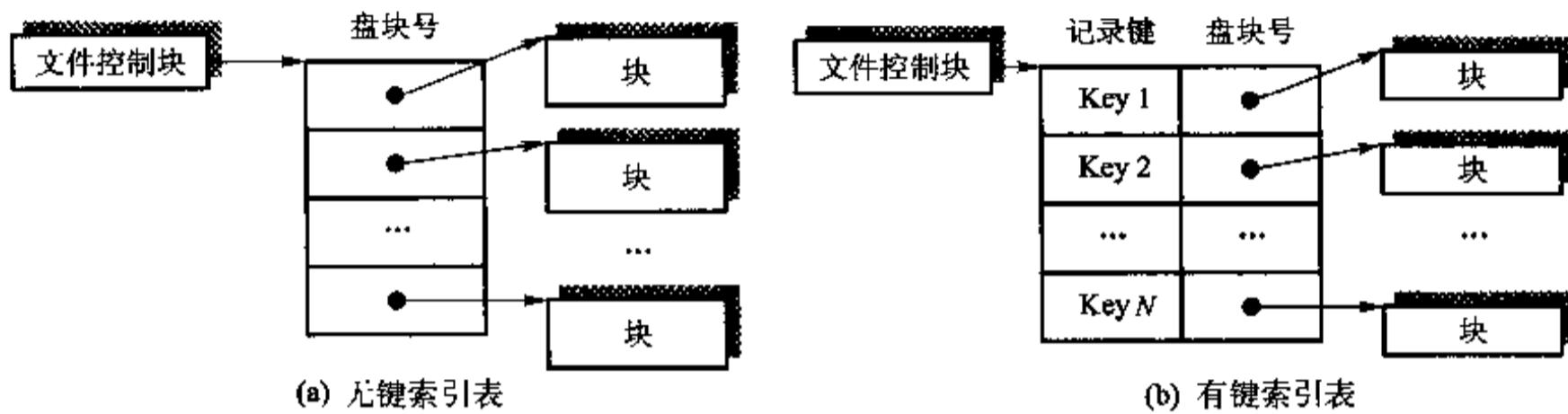


图 6.6 两种索引文件的结构示意图

索引结构是连接结构的一种扩展,除了具备连接文件的优点之外,记录可以散列存储,具有直接读写任意记录的能力,便于信息的增、删、改。其缺点是:索引表的空间开销和查找时间开销大,大型文件的索引表的信息量甚至可能远远超过文件记录本身的信息量。

索引顺序文件是顺序文件的一种扩展,各条记录在介质上顺序排列。例如,按字母序排列,由于有随机访问的关键字,具有直接处理和修改记录的能力。索引顺序文件能够进行快速顺序处理,既允许按物理存放顺序(记录出现的次序)又允许按逻辑顺序(记录键次序)进行处理。

有时记录数目很多,索引表要占用许多物理块,在查找某记录键所对应的索引项时,可能需要依次交换很多块。若索引表占用 n 块,则平均要交换 $(n + 1)/2$ 次,才能找到所需记录的物理地址。当 n 值很大时,这是费时的操作。提高查找速度的方法是:做一个索引的索引,称其为二级索引,二级索引表的表项列出一级索引表每一块最后一个索引项的记录键及此索引表的盘块号,也就是说,若干条记录的索引本身也是一种记录,查找时先查看二级索引表,找到某键所在的索引表盘块号,再搜索一级索引表,按键找出数据记录。当记录数十分庞大时,索引的索引也可能占用许许多多块,那样,可做索引的索引的索引,称其为三级索引。有些计算机系统还建立更多层次的索引,当然这些工作都由文件系统自动完成。

UNIX/Linux 操作系统的多重索引结构稍有不同,如图 6.7 所示,每个文件的 FCB 中规定 13 个索引项,每项占用 3 B,登记一个存放文件信息的物理块号。由于系统仅提供流式文件,无记录的概念,因而,登记项中没有记录键与之对应。前面 10 项存放文件信息的盘块号,称为直接

寻址；如果文件大于 10 块，利用第 11 个索引项指向一个物理块，此块中最多可放 128 个存放文件信息的磁盘块的块号，叫做一次间接寻址。大型文件还可以利用第 12、第 13 索引项作二次和三次间接寻址。因为如果每个物理块存放 512 B，则每个文件的最大长度约为 11 亿字节。多重索引结构的优点与一般索引文件相同，其缺点是多次间接寻址会降低查找速度。对于分时使用环境进行统计表明，长度不超过 10 个磁盘块的文件占总数的 80%，通过直接寻址便能找到文件信息，对仅占总数 20% 的超过 10 个磁盘块的文件才施行间接寻址。

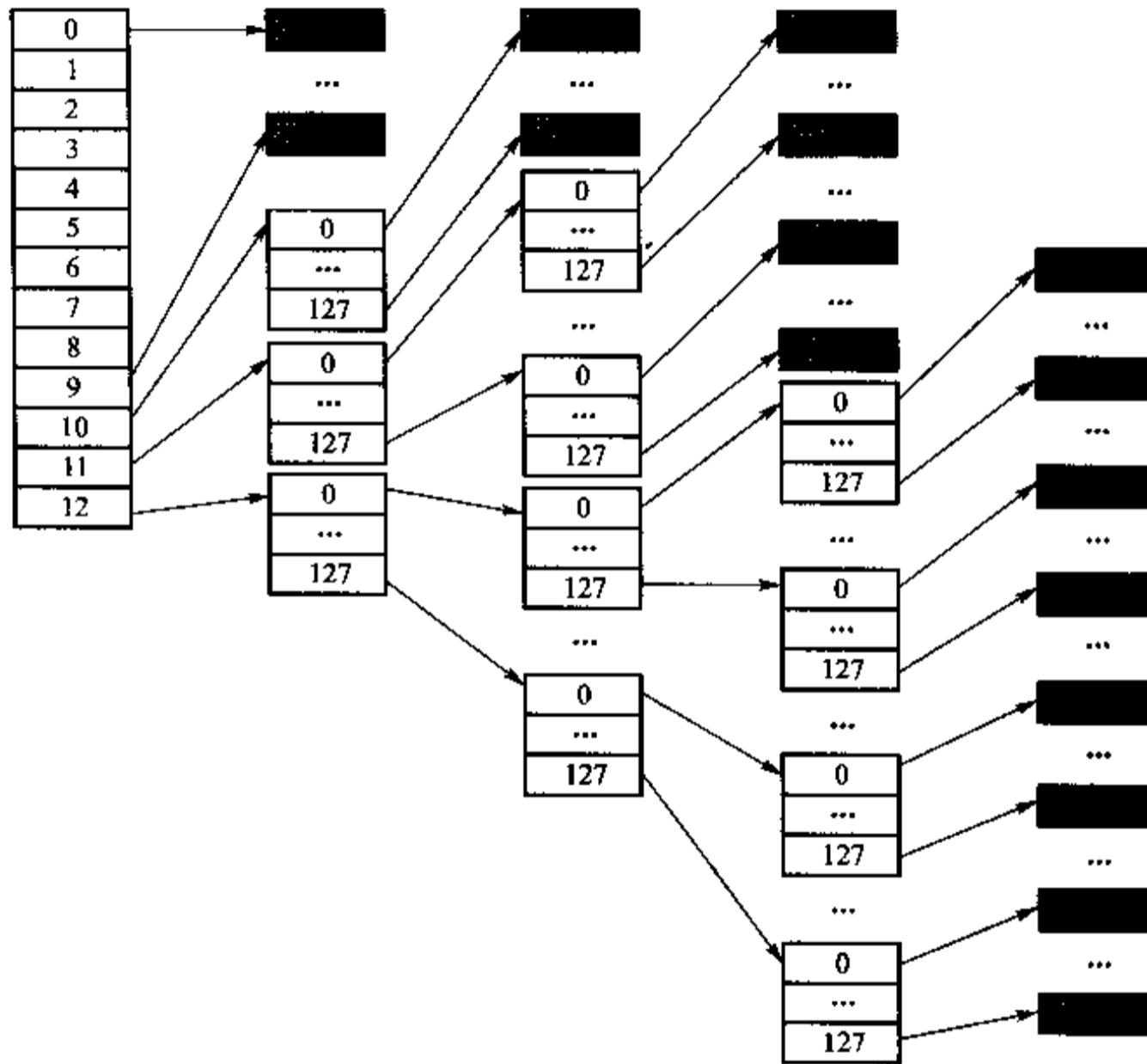


图 6.7 UNIX/Linux 多重索引结构

6.4

文件系统其他功能的实现

6.4.1 文件系统调用的实现

文件系统向应用程序所提供的一组系统调用包括：建立、打开、关闭、撤销、读、写和控制，通过这些系统调用，用户能够获得文件系统的各种服务。在为应用程序服务时，文件系统需要沿路径查找目录以获得文件的各种信息，这往往需要多次访问文件存储器，使访问速度减慢。若把所

有文件目录都复制到主存储器,访问速度可以加快,但却又会增加主存开销。一种行之有效的方法是,把常用和正在使用的那些文件目录复制进主存储器,这样,既不增加太多的主存开销,又可以明显地缩短查找时间,系统为每个用户进程建立一张打开文件表,用户使用文件之前先通过“打开”操作,把此文件的文件目录复制到指定的主存区域,当不再使用这个文件时,使用“关闭”操作切断和此文件目录的联系,这样,文件被打开后,可被用户多次使用,直至文件被关闭或撤销,大大地减少磁盘访问次数,提高文件系统的效率。

下面以 Linux 文件系统调用为例,介绍其功能和实现。Linux 文件系统调用的种类和功能与 UNIX 大致相同,但在具体实现上有一定的差别。内核将磁盘扇区编号,扇区序列分成 3 个部分。

(1) 超级块:占用 1# 块,存放文件系统的结构和管理信息,如记录 inode 表所占的盘块数、文件数据所占的盘块数、主存中登记的空闲盘块数、主存中登记的空闲块的物理块号、主存中登记的空闲 inode 数、主存中登记的空闲 inode 编号以及其他文件管理控制信息。可见,超级块既有盘位示图的功能,又记录整个文件卷的控制数据。每当一个块设备作为文件卷被安装时,此设备的超级块就要复制到主存系统区中备用,而在拆卸文件卷时,修改过的超级块需要复制回磁盘的超级块中。

(2) 索引节点区: $2\# \sim (k+1)\#$ 块,存放 inode 表,每个文件都有各种属性,将其记录在称为索引节点 inode 的结构中。所有 inode 都有相同的大小,且 inode 表是 inode 结构的列表,文件系统中的每个文件在表中都有一个 inode。又分为磁盘 inode 表和主存活动 inode 表,后者解决频繁访问磁盘 inode 表的效率问题。

(3) 数据区: $(k+2)\# \sim n\#$ 为数据区,文件内容保存在这个区域中,磁盘上所有物理块的大小是一样的。如果文件包含超过一块的数据,则文件内容会存放在多个盘块中。

这里还要介绍两个重要的数据结构。

(1) 用户打开文件表:进程的 PCB 结构中保留一个 files_struct,称为用户打开文件表或文件描述符表。表项的序号是文件描述符 fd,此登记项内登记系统打开文件表的一个入口指针 fp,通过此系统打开文件表项连接到打开文件的活动 inode。

(2) 系统打开文件表:这是为解决多用户进程共享文件、父子进程共享文件而设置的系统数据结构 file_struct。主存专门开辟最多可登记 256 项的系统打开文件表区,当打开一个文件时,通过此表项把用户打开文件表的表项与文件活动 inode 连接起来,以实现数据的访问和信息的共享。如图 6.8 所示是文件系统内部结构示意图。图 6.9 给出目录项、inode 和数据块之间的关系,有两个目录项指向同一个 inode,inode 的连接计数用于记录所连接的目录项数。

1. 创建和删除文件

当文件尚未存在时,需要对其创建,或者文件原来已经存在,有时需要重新创建。注意这同文件的打开是完全不同的概念,文件打开是指当文件已经存在,需要使用时先执行打开操作,以便建立应用进程与文件之间的联系。相应地,不再需要文件时,可以删除之,以便节省存储空间。

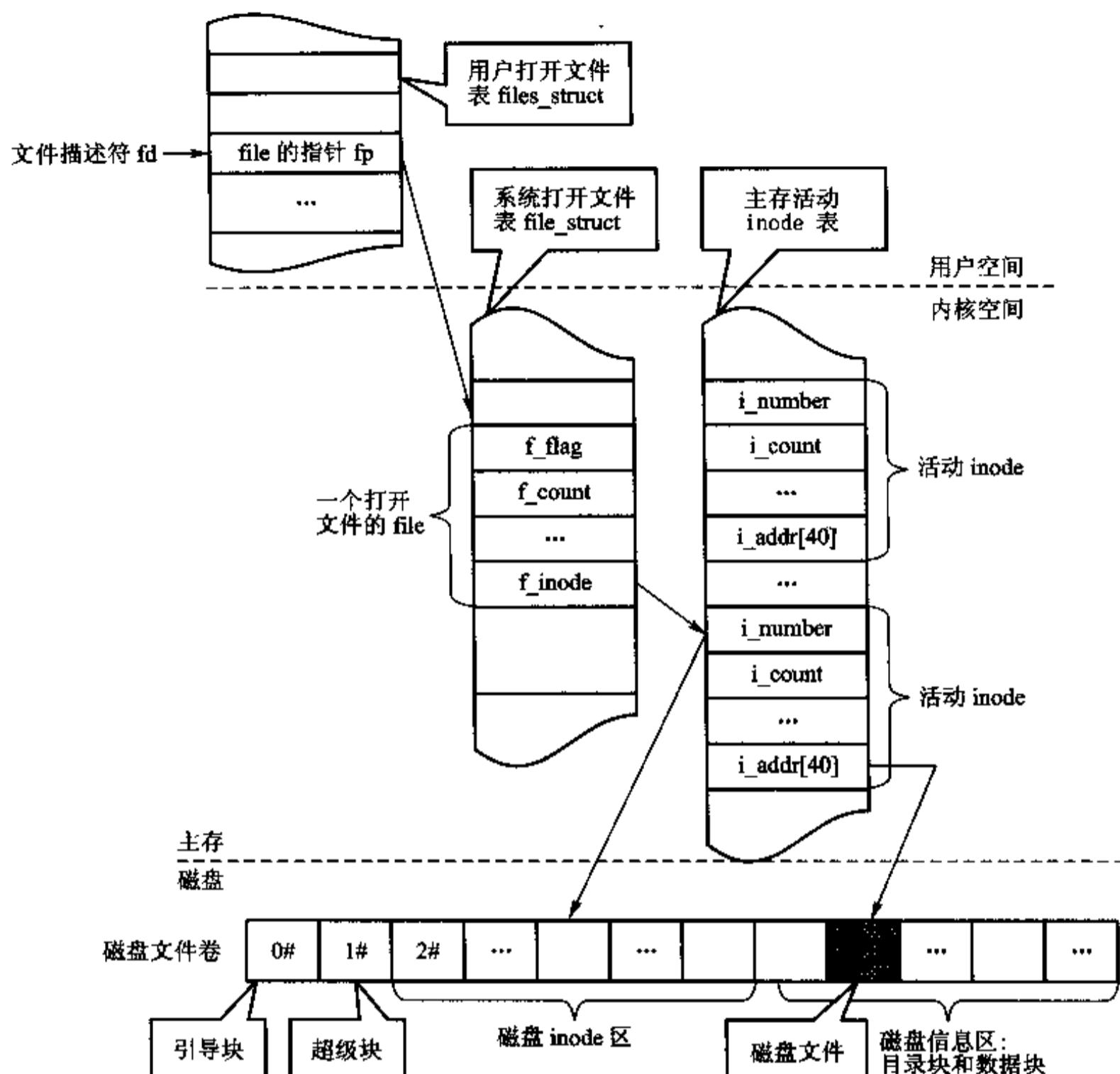


图 6.8 文件系统内部结构

(1) 创建文件

系统调用的 C 语言格式为：

```
fd = create(filenamep, mode);
```

其中，参数 `filenamep` 是指向所要创建的文件路径名的字符串指针；参数 `mode` 是文件所具有的存取权限，在文件成功创建之后，权限被记录在相应索引节点的 `i_mode` 中。变量 `fd` 是创建成功后系统所返回的文件描述符，即用户打开文件表中相应文件表项的序号。由此可见，`create` 兼具文件的“打开”功能，随后执行文件写系统调用，就可使用这个 `fd` 进行写操作。例如，要创建文件的路径名是 `/home/home1/newfile`，可使用如下系统调用：

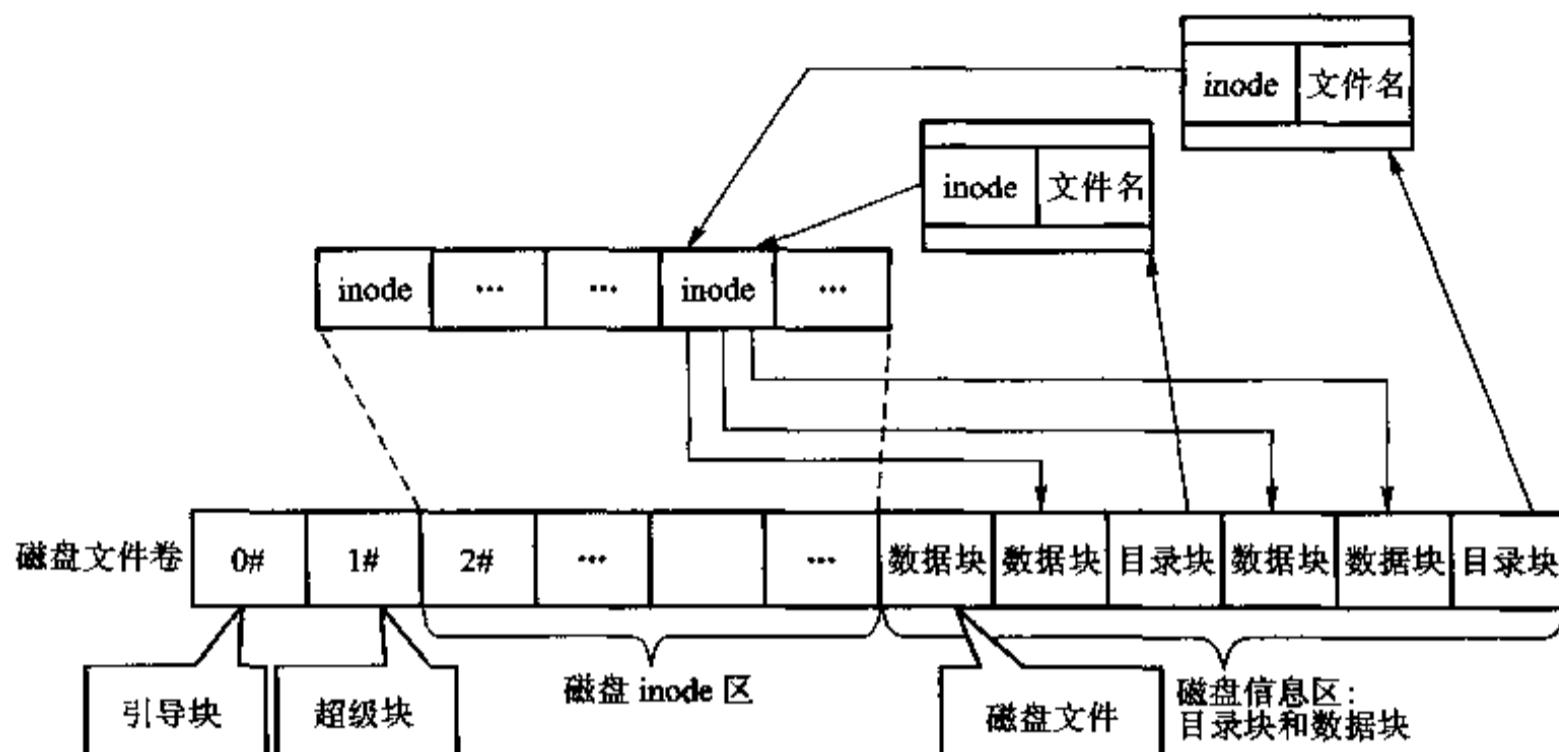


图 6.9 目录项、inode 和数据块之间的关系

```
fd = create("/home/home1/newfile", 0775);
```

下面简述其实现过程,假定文件是首次创建,即在执行之前,文件尚未存在。

① 为新文件 newfile 分配磁盘 inode 和活动 inode,并把 inode 号与文件分量名 newfile 组成新的目录项,记录到目录路径 /home/home1 的目录文件中。在这一过程中,需要执行目录检索程序。

② 在新文件所对应的活动 inode 中置初值,包括把存取权限 i_mode 置为 0775,连接计数 i_link 置为“1”,等等。

③ 为新文件分配用户打开文件表项和系统打开文件表项,置系统打开文件表项的初值,包括在 f_flag 中置“写”标志,读写位移 f_offset 清 0,等等;把用户打开文件表项、系统打开文件表项及 newfile 所对应的活动 inode 用指针连接起来,最后,把文件描述符 fd 返回给调用者。

由于上述步骤中兼有文件“打开”功能,因此在以后的操作中,无须执行“打开”操作即可进行读写。

(2) 删除文件

删除的任务是把指定文件从其所在的目录文件中去除。如果没有连接的用户,即如果在执行删除之前 i_link 为“1”,还要把这个文件所占用的存储空间释放。文件删除系统调用的形式是 unlink(filenamep),其中参数与 create 中的含义相同,在执行删除操作时,必须要求用户对文件具有“写”操作权。

2. 打开和关闭文件

文件在使用之前必须先“打开”,以建立进程与文件之间的联系,而文件描述符唯一地标识这样一种连接,其任务是把文件的磁盘 inode 复制到主存的活动 inode 表中,同时建立一个独立的读写文件数据结构,即系统打开文件表的一个表项。另一方面,活动 inode 表的大小受到主存容

量的限制,这就要求用户一旦不再对文件进行操作时,应立即释放相应的活动 inode,以便让其他进程使用,这就是“关闭”文件的主要功能。

(1) 打开文件的调用形式为:

```
fd = open(filenamep, mode);
```

其中,mode 是打开方式,表示打开后的操作要求,如读(0)、写(1)或读写(2),其余参数的意义与 create 中的相同。打开操作的实现过程如下。

① 检索目录,要求打开的文件应该是已经创建的文件,它应登记在文件目录中,否则就会出错。在检索到指定的文件之后,就将其磁盘 inode 复制到活动 inode 表中。

② 把参数 mode 所给出的打开方式与活动 inode 中在创建文件时所记录的文件访问权限相比较,如果非法,则此次打开操作失败。

③ 当“打开”合法时,为文件分配用户打开文件表项和系统打开文件表项,并为系统打开文件表项设置初值,通过指针建立表项与活动 inode 之间的联系,然后把文件描述符 fd 返回给调用者。

需要指出的是,如果在执行这一调用之前,其他用户已打开同一文件,则活动 inode 表中已有此文件的 inode,于是,不用执行第①步中复制 inode 的工作,而仅把活动 inode 中的计数器 i_count 加 1 即可。在此 i_count 反映通过不同的系统打开文件表项来共享同一活动 inode 的进程数目,它是以后执行文件关闭操作时,活动 inode 能否被释放的依据。

(2) 文件的关闭

文件使用完毕,执行关闭系统调用,切断应用进程与文件之间的联系。其调用形式为 close(fd)。显然,要关闭的文件先前已打开,故文件描述符 fd 一定存在,其执行过程如下。

① 根据 fd 找到用户打开文件表项,再找到系统打开文件表项;释放用户打开文件表项。

② 把对应的系统打开文件表项中的 f_count 减 1,如果其值不为 0,说明进程族中还有进程正在共享它,不用释放此表项直接返回;否则释放此表项,并找到与之连接的活动 inode。

③ 把活动 inode 中的 i_count 减 1,若其值不为 0,表明其他进程正在使用此文件,直接返回;否则,把此活动 inode 的内容复制回文件卷上的相应磁盘 inode 中,释放此活动 inode。

可见,f_count 和 i_count 分别反映进程动态地共享一个文件的两种方式,前者反映不同进程通过同一个系统打开文件表项共享一个文件的情况;而后者反映不同进程通过不同系统打开文件表项共享一个文件的情况。通过这两种方式,进程之间既可用相同的位移指针 f_offset,也可用不同的位移指针 f_offset 共享同一个文件。

3. 文件的读和写

文件的读和写是文件最基本的操作,“读”是指文件的内容读入用户数据区中,“写”是指把用户数据区中的信息写入文件中。从文件的哪个逻辑位置读入数据,或者把数据写入文件的哪个逻辑位置,均由系统打开文件表中的 f_offset 决定。

(1) 读文件

```
nr = read(fd, buf, count);
```

这里,fd 是系统调用执行后返回给应用进程的文件描述符;buf 是读出信息所应送入的用户数据区首地址;count 是要求传送的字节数,nr 是此系统调用执行后所返回的实际读入字节数。nr 所指出的字节数可能小于 count 所要求的字节数,例如,一旦读到文件末尾,系统调用就返回,无论是否已读够用户所要求的字节数。如果文件的位移指针已经指向文件末尾,又使用 read 系统调用,则返回值 0。

假定已通过打开系统调用打开文件 /home/home1/newfile,与它有关的用户打开文件表项、系统打开文件表项和活动 inode 之间的关系如图 6.10 所示。

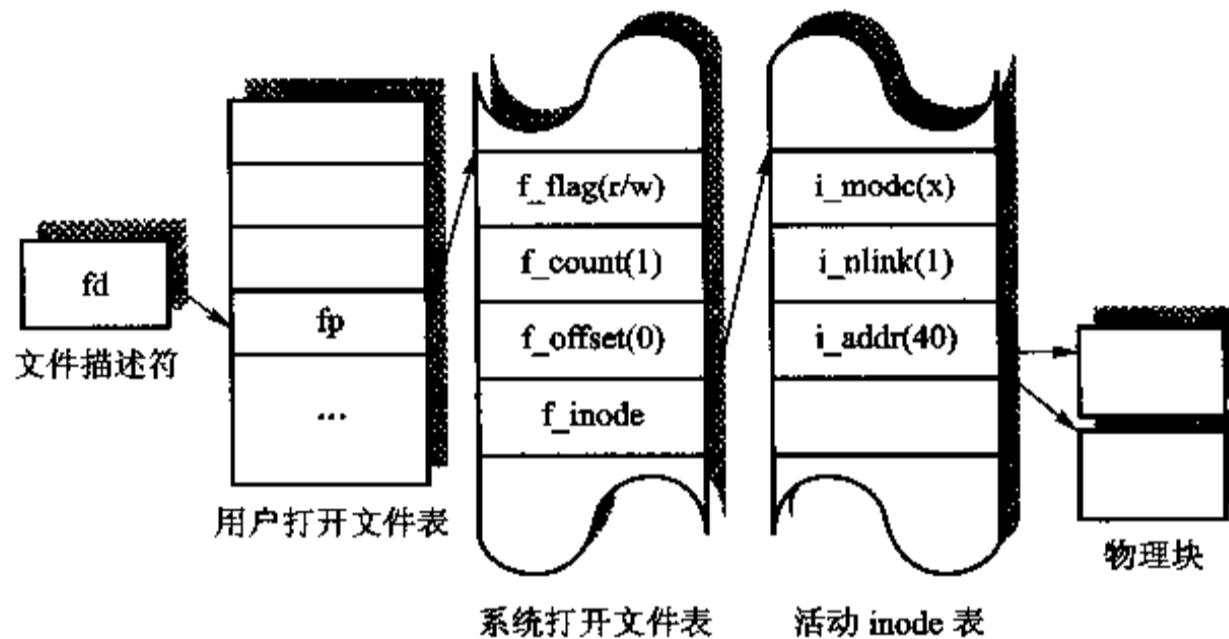


图 6.10 读操作时文件数据结构的关系

现要求读文件 newfile 的 1 500 个字符到指针 bufp 所指向的用户数据区中,n 用来存放实际传送的字节数,则可按如下方式调用:

```
n = read(fd, bufp, 1500);
```

在执行 read 系统调用的过程中,首先根据 f_flag 中所记录的信息,检查读操作的合法性。如果读操作合法,按活动 inode 中 i_addr 指出的文件物理块存放地址,从文件当前的位移量 f_offset 处开始,读出所要求的字节数到块设备缓冲区中。然后,再送到 bufp 所指向的用户数据区中。由此可见,在执行读操作的过程中,一定要用到块设备管理中的读程序。

(2) 写文件

此系统调用的形式为:

```
 nw = write(fd, buf, count);
```

其中,fd、count 和 nw 的含义类似于 read 中的含义,buf 是信息传送的源地址,即把 buf 所指向的用户数据区中的信息写入文件中。只要情况一切正常,nw 与 count 相等。

4. 文件的随机存取

在文件初次“打开”时,文件的位移量 f_offset 清空为 0,以后的文件读写操作总是根据 offset 的当前值来顺序地读写文件。为了支持文件的随机访问,提供系统调用 lseek,它允许用户在读写文件之前,事先改变 f_offset 的指向。系统调用的形式为:

```
lseek(fd, offset, whence);
```

其中,文件描述符 fd 必须指向一个以读或写方式打开的文件,当 whence 值为 0 时,则 f_offset 被置为 offset;当 whence 值为 1 时,则 f_offset 被置为文件当前位置值加上 offset。

6.4.2 文件共享

文件共享是指不同的进程共同使用同一个文件,文件共享不仅为不同的进程完成共同任务所必需,而且还节省大量的辅存空间,减少因文件复制而增加的 I/O 操作次数。文件共享有多种形式:静态共享、动态共享和符号链接共享。在 UNIX/Linux 系统中,允许多个用户静态共享,或动态共享同一个文件,当一个文件被多个进程动态地共享使用时,每个进程可以使用各自的读写指针,但也可以共用读写指针。

1. 文件静态共享

操作系统允许一个文件同时属于多个目录,但实际上文件仅有一处物理存储。这种在物理上一处存储、从多个目录可到达此文件的“多对一关系”称为文件链接。如果让各进程采用物理副本,把同一文件复制到自己的目录下,意味着会因冗余而浪费磁盘空间;而且还可能造成数据的不一致性,即当一个用户修改共享文件的一个副本时,其他用户不知道所发生的修改,最终变成多个内容不同的文件。在 UNIX/Linux 系统中,两个或多个进程可通过文件链接达到共享同一个文件的目的,无论进程是否运行,其文件的链接关系都是存在的,因此,称为静态共享。用文件链接来代替文件的复制可以提高文件资源利用率,节省文件物理存储空间。

链接的文件和目录多处出现时,可能会由不同的进程成单一进程使用;可能在不同的父目录下,这时可能以不同或相同的文件(目录)名出现;也可能在同一父目录下,在这种情况下,应以不同的文件(目录)名出现。在图 6.1 中,在 fei1 和 fei2 目录下以相同文件名 myfile.c 出现,但在 fei4 目录下虽为同一个文件却以 testfile.c 的文件名出现。

静态共享是通过文件所对应的 inode 节点来实现链接的,并且只允许链接到文件而非目录。文件链接的系统调用形式为:

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

其中,oldnamep 和 newnamep 分别是指向已存在文件名的字符串和文件别名的字符串的指针。其执行步骤如下。

- (1) 检索目录,找到 oldnamep 所指文件的 inode;
- (2) 再次检索目录,找到 newnamep 所指文件的父目录文件,并把已存在文件的 inode 编号与别名构成目录项,记录到目录中去;
- (3) 把已存在文件 inode 的连接计数 i_nlink 值加 1;所以,所谓链接,实际上是共享已存在文件的 inode。完成文件链接的系统调用为:

```
link("/home/fei1/myfile.c", "/home/fei2/myfile.c");
```

```
link("/home/fei1/myfile.c", "/home/fei3/fei4/testfile.c");
```

执行后,以下 3 个路径名所指的是同一个文件:/home/fei1/myfile.c、/home/fei2/myfile.c, 和 /

home/fei3/fci4/testfile.c。图 6.2(b)中相对应的目录链接中有 3 个指针指向文件 myfile.c 的 inode(inode 号为 302)。

文件解除链接的调用形式为 `unlink(namep)`。实际上,UNIX/Linux 中解除链接与文件删除所执行的是同一段系统调用代码。删除文件是从文件属主角度来讲的,而解除文件链接是从共享文件的其他用户角度来讲的。无论删除还是解除链接,都要删去对应的目录项,把 `i_nlink` 值减 1。

在文件的静态链接中,为了反映共享同一文件的用户数,在每个索引节点中设立一个变量 `i_nlink`。当文件第一次创建时,`i_nlink` 值为 1,以后,每次链接时就把 `i_nlink` 值加 1。当用户删除文件(对于文件主)或解除链接(对于其他用户)时,就把此计数值减 1,直到发现结果为 0 时,才释放文件的物理存储空间,从而真正删除这个文件。

2. 文件动态共享

文件动态共享是指系统中不同的应用进程或同一用户的不同进程并发地访问同一文件,这种共享关系只有当进程存在时才可能出现,一旦进程消亡,其共享关系也就随之消失。在 UNIX/Linux 系统中,文件打开后,它的 inode 就在活动 inode 表中。活动 inode 表是整个系统公用的,为了使用户掌握当前使用文件的情况,系统在每个进程的 PCB 中设立用户打开文件表,并通过它与各自打开文件的活动 inode 联系。

由于 UNIX/Linux 系统中的文件采用无结构字符流序列,因此,文件的每次读写都要由一个位移指针指出所要读写的位置。现在的问题是:若一个文件可被多个进程所共享,那么,应让多个进程共用同一个位移指针,还是应让各个进程具有各自的读写位移指针呢?下面分两种情况进行讨论。

在 UNIX/Linux 系统中,可以动态地创建进程,被创建的进程继承父进程的一切资源,包括 PCB 中用户打开文件表中所指定的打开文件。同一用户的父、子进程往往要协同完成同一任务,若使用同一读/写位移指针,那么,当一个进程改变它时,另一个进程就能够感觉到它的变化。这样,使父、子进程很容易同步地对文件进行操作。因此,这个位移指针似乎适合放在相应文件的活动 inode 中。此时,当执行 `fork` 函数创建子进程时,父进程的 PCB 被复制到子进程的 PCB 之中,使得两个进程的打开文件表指向同一活动 inode,达到共享同一位移指针的目的。如图 6.11 所示是共享位移指针的文件共享。

另一方面,若一个文件为两个以上的用户所共享,每个用户必然都希望能独立地读写此文件,彼此互不干扰。这时不能只设置一个读写位移指针,而必须为每个应用进程分别设置读写位移指针。据此,位移指针不应放在相应文件的活动 inode 中,而需独立设置。这样,当一个进程读写文件并修改位移指针时,另一个进程的位移指针不会随之改变,从而,两个进程能够独立地访问同一个文件。如图 6.12 所示是不共享指针的文件共享。

在上述两种动态共享方式中,对读写位移指针的位置设置和数目是不同的。为了解决这个矛盾,系统建立一张系统打开文件表,在此表的每个表项中,除了含有读写位移指针 `f_offset` 之外,还有文件的访问计数 `f_count`、读写标志 `f_flag` 和指向对应文件的活动索引节点指针 `f_inode`,其中 `f_count` 指出使用同一系统打开文件表目的进程数目,它是系统打开文件表目资源能否被释

放的标志。用户打开文件表、系统打开文件表和活动 inode 表之间的关系已在图 6.11 和图 6.12 中表明。

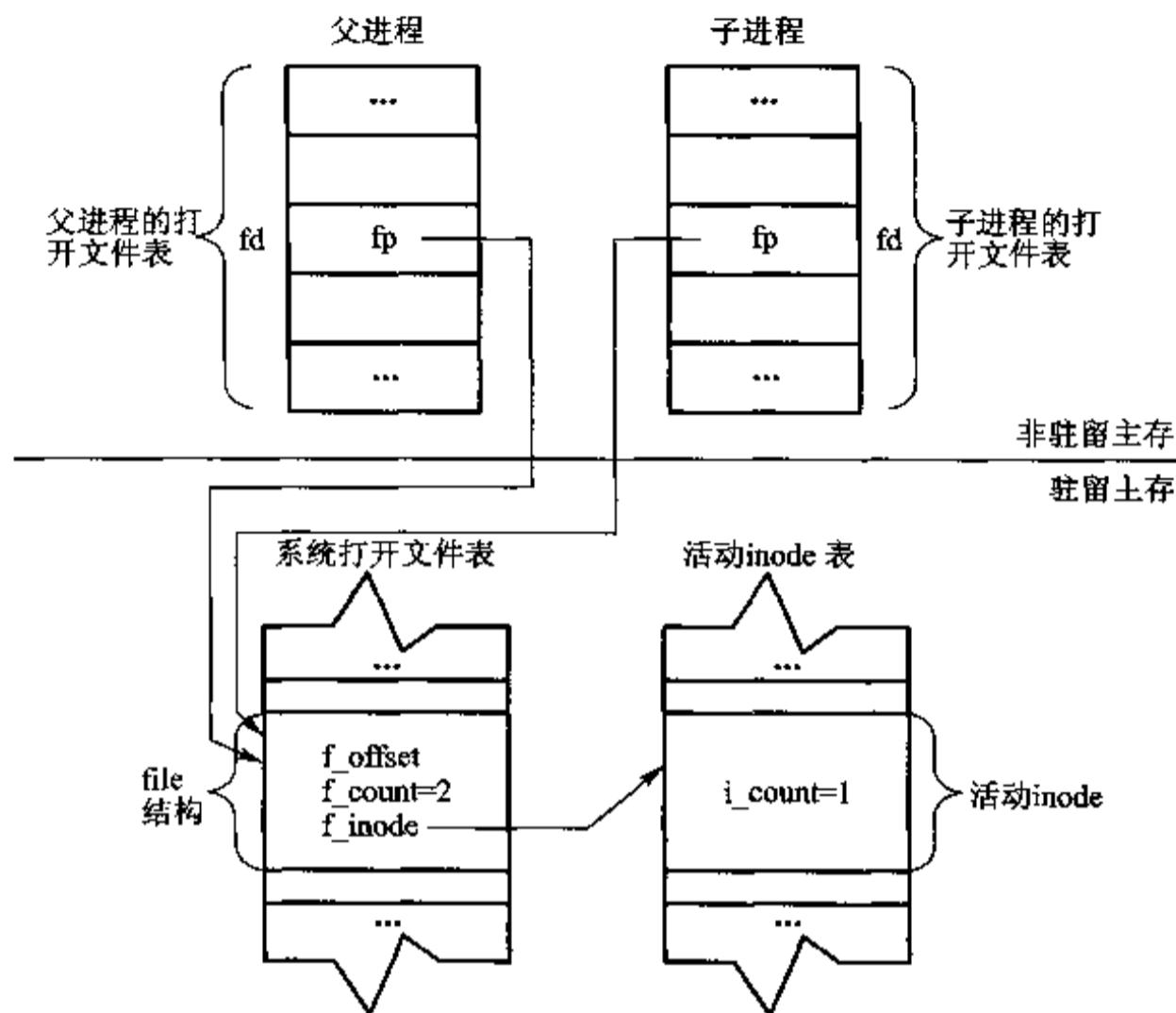


图 6.11 共享位移指针的文件共享

在图 6.12 中, 进程 A 及其子进程通过同一个系统打开文件表项共享文件 newfile, 其 *f_count* 值为 2。而进程 B 独立使用一个系统打开文件表项, 其 *f_count* 值为 1, 但它却通过同一个活动 inode 与进程 A 及其子进程共享文件 newfile。

下面来看系统打开文件表项的申请和释放过程。当进程要求打开文件时, 首先, 系统为它申请一个系统打开文件表项, 并建立此表项与相应文件活动 inode 之间的联系; 然后, 把用户打开文件表项中的一个空闲项指向它。如果在此之后, 应用进程通过系统调用 fork 创建一个子进程, 系统自动把用户打开文件表项所指的系统打开文件表项中的 *f_count* 值加 1。反之, 当进程关闭一个文件时, 系统不能简单地释放系统打开文件表项, 必须首先判断 *f_count* 的值是否大于 1。如果其值大于 1, 说明还有进程共享相应的系统打开文件表项, 此时只需把 *f_count* 值减 1 即可。由此可知, 进程之间到底采用哪一种方式动态地共享同一个文件, 若是先执行 fork, 再用“打开”系统调用来打开同一名称的文件, 则系统分别为两个进程分配不同的系统打开文件表项, 并指向同一活动 inode, 于是两个进程独立地使用各自的读写位移量指针。但是如果进程在打开一个文件之后, 再用 fork 系统调用建立子进程, 由于子进程的用户打开文件表项是从父进程那里复制得到的, 它自然会指向父进程所使用的系统打开文件表项, 因此, 父子进程就共用同一个位

移量指针来共享文件。

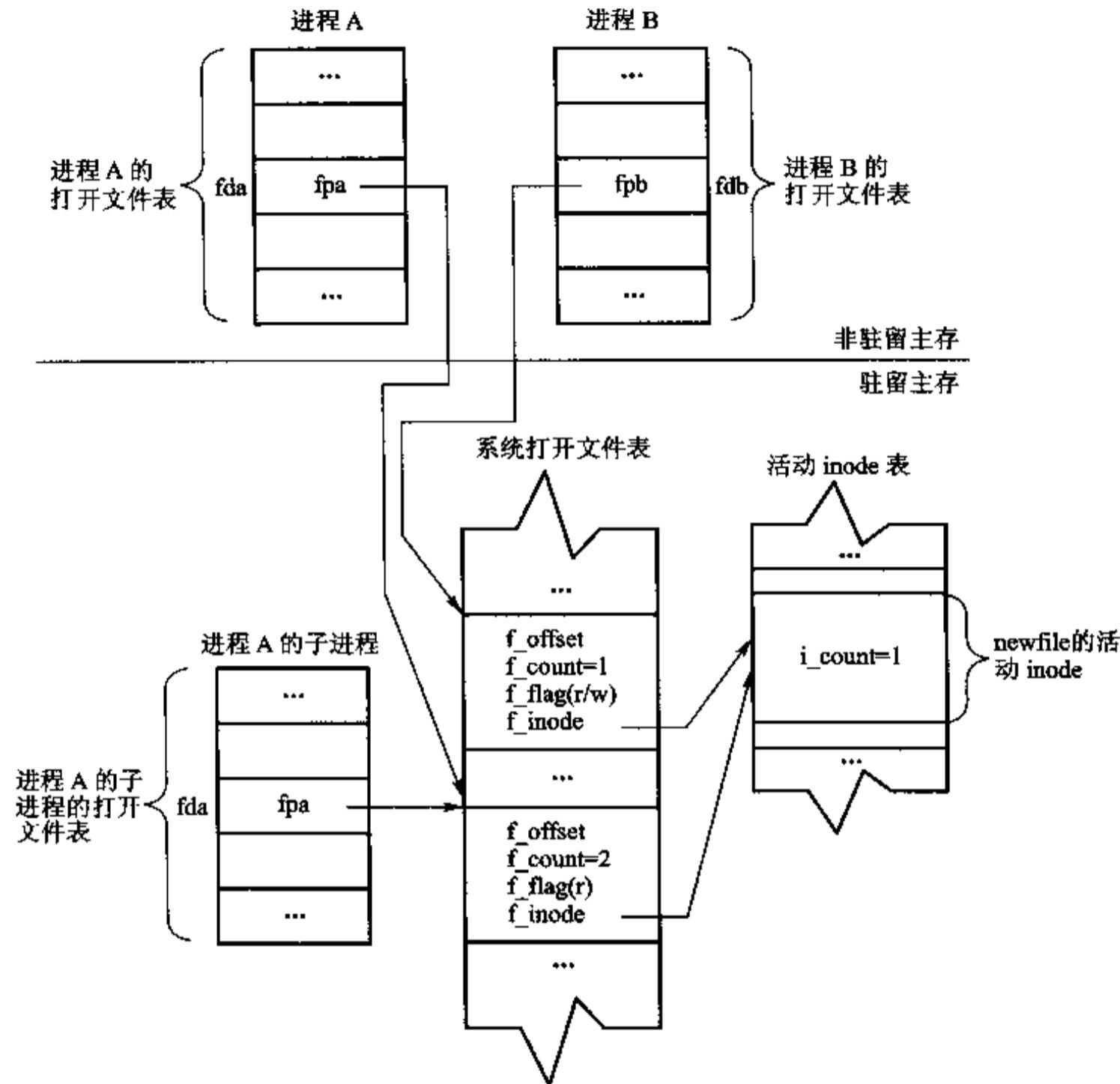


图 6.12 不共享位移指针的文件共享

3. 文件符号链接共享

操作系统可以支持多个物理磁盘或逻辑磁盘(分区),那么,文件系统是建立一棵目录树还是多棵目录树呢?有两种方法,将盘符或卷标分配给磁盘或分区,并将其名字作为文件路径名的一部分。Windows 操作系统采用这种方法。

UNIX/Linux 系统使用另外的方法,每个分区都有自己的文件目录树,当有多个文件系统时,可通过安装的方法整合成一棵更大的文件目录树。现在出现一个小问题,系统中的每个文件对应于一个 inode,其编号是唯一的,但是两个不同的磁盘或分区可能都含有相同 inode 号所对应的文件,也就是说,在整合的目录树中,inode 号并不唯一地标识一个文件,因而,无法做到从不同的文件系统生成指向同一个文件的链接。对于 link 之类的系统调用,只能拒绝创建跨越文件系

统的链接。

将文件名和自身的 inode 链接起来,称为硬链接(hard link)。硬链接只能用于单个文件系统,却不能跨越文件系统,可用于文件共享但不能用于目录共享,其优点是实现简单,访问速度快。另一种是软链接,又称符号链接(symbolic link),可以克服上述缺点。符号链接是只有文件名、不指向 inode 的链接,通过名称来引用文件。例如,用户 A 通过文件名 afile 来共享用户 B 的文件 bfile,可由系统生成 bfile 的一个符号链接,把所创建的新链接称为 afile,把此“符号链接”写入用户 A 的用户目录中,形式为 afile→bfile,以实现 A 的目录与 B 的文件的链接。符号链接中只包含被链接文件 bfile 的路径名而非其 inode 号,而文件的拥有者才具有指向 inode 的指针。当用户 A 要访问被符号链接的用户 B 的文件 bfile,且要读“符号链接”类文件时,被操作系统截获,依据符号链接中的路径名去读文件,于是就能实现用户 A 使用文件名 afile 对用户 B 的文件 bfile 的共享。符号链接的优点是能用于链接计算机系统中不同文件系统中的文件,也可用于链接目录,进一步可链接计算机网络中不同机器上的文件,此时,仅需提供文件所在机器地址和其中文件的路径名。这种方法的缺点是:搜索文件路径的开销大,需要额外的空间查找存储路径。

6.4.3 文件空间管理

在磁盘等大容量辅助存储器空间中,用户作业运行期间经常要建立、扩充和删除文件,文件系统应能自动管理和控制辅存文件空间。在创建和扩充文件时,决定分配哪些磁盘块是很重要的,这将会影响今后的磁盘访问次数,删除文件或缩短其长度时,需回收磁盘块并把盘块移入空闲队列。

辅存文件空间的有效分配和释放是文件系统所要解决的一个重要问题。最初,整个存储空间可连续分配给文件使用。随着用户文件的不断建立和撤销,存储空间中会出现“碎片”。系统应定时或根据命令要求来集中碎片,在收集过程中往往要对文件重新组织,让其存放到连续存储区中。辅存文件空间分配常采用以下两种方法。

1. 连续分配

文件被存放在辅存空间的连续存储区(连续的物理块号)中,在建立文件时,用户必须给出文件大小,然后,查找能满足要求的连续存储块供使用;否则不能建立文件,进程必须等待。连续分配的优点是顺序访问时通常无须移动磁头,查找速度快,管理简单,但为了获得足够大的连续存储块,需要定时进行“碎片”收集。因而,不适用于文件频繁进行动态增长和收缩的情况,用户事先不知道文件长度时也无法进行分配。

2. 非连续分配

非连续分配的一种方法是以块(扇区)为单位,按文件的动态要求向其分配若干扇区,这些扇区不一定连续,属于同一文件的扇区按文件记录的逻辑次序用链指针连接或用位示图指示。另一种方法是以簇为单位,簇是若干连续扇区所组成的分配单位,实质上是连续分配和非连续分配的结合。各个簇可用链指针、索引表、位示图来管理。非连续分配的优点是:辅存空间管理效率高,便于文件动态增长和收缩,访问文件的执行速度快,特别是以簇为单位的分配方法已被广泛

使用。下面介绍常用的几种文件辅存空间管理方法。

1. 位示图

磁盘空间通常使用固定大小的块,可方便地用位示图管理。用若干字节构成一张位示图,其中每一字位对应于一个物理块,字位的次序与块的相对次序一致,字位为“1”表示相应块已被占用,字位为“0”表示相应块空闲。微机操作系统 VM/SP、Windows 和 Macintosh 等均使用这种技术来管理文件存储空间。其主要优点是:每个盘块仅需 1 个附加位,如盘块长 1 KB,位示图开销仅占 0.012%;可把位示图全部或大部分保存在主存储区,再配合现代机器都具有的位操作指令,实现高速物理块分配和去配。

2. 空闲区表

这种分配方法常常用于连续文件,将空闲存储块的位置及其连续空闲的块数构成一张表。分配时,系统依次扫描空闲区表,寻找合适的空闲块并修改登记项;删除文件并释放空闲区时,把空闲区位置及连续空闲区长度填入空闲区表,出现邻接的空闲区时,还需执行合并操作并修改登记项。空闲区表的搜索算法包括优先适应、最佳适应和最坏适应算法等,这在前面已经介绍过。

3. 空闲块链

把所有空闲块连接在一起,系统保持指针指向第一个空闲块,每一空闲块中包含指向下一空闲块的指针。申请一个空闲块时,从链头取一块并修改系统指针;删除时释放占用块,使其成为空闲块并挂到空闲链上。这种方法的执行效率低,每申请一块都要读出空闲块并取得指针,申请多块时要多次读盘,但便于文件动态增长和收缩。

4. 成组空闲块链

图 6.13 给出 UNIX/Linux 所采用的空闲块成组连接法。存储空间分成 512 B 一块。为了方便讨论,假定文件卷启用时共有可用空闲块 338 块,编号从 12 至 349,每 100 块划分成一组,每组第一块登记下一组空闲盘块号和空闲总数,其中,50# - 12# 一组中,50# 物理块中登记下一组 100 个空闲盘块号 150# - 51#, 同样下一组的第一块 150# 中登记再下一组 100 个空闲盘块号 250# - 151#。需要注意的是,在最后一组中,即 250# 块中的第 1 项是 0,作为结束标志,表明系统空闲盘块链已经结束。操作系统启动时,将磁盘专用块复制到主存系统工作区中,访问主存大多数情况可完成申请或释放空闲盘块的操作。

分配空闲盘块时,总是先把专用块中的空闲块计数减 1,以它为指针找到专用块的相应表项,其内容就是要分配的空闲盘块号。当空闲块计数减 1 后等于 0 时,专用块中仅剩 1 个盘块号,此时要取出表项中的盘块号 i ,再把此盘块中所保存的下一组空闲盘块链接信息经缓冲区复制到主存专用块中,然后,把当前盘块 i 分配出去。

释放存储块时,将块号记录在由专用块所指示的表项中,然后空闲块计数加 1。若发现此表已满(达 100 项),则应把整个表的内容经缓冲区复制到下面要释放的盘块中,然后将释放块块号写入专用块中的第一个位置,置空闲块计数为 1。

搜索到全 0 块时,系统应向操作员发出警告,表明空闲块已经用完。需要注意的是,开始时空闲块是按序排列的,但只要符合分组及组间连接的原则,空闲块可按任意次序排列。事实上,

经过若干次分配和释放操作后,空闲块物理块号必定不再按序排列。

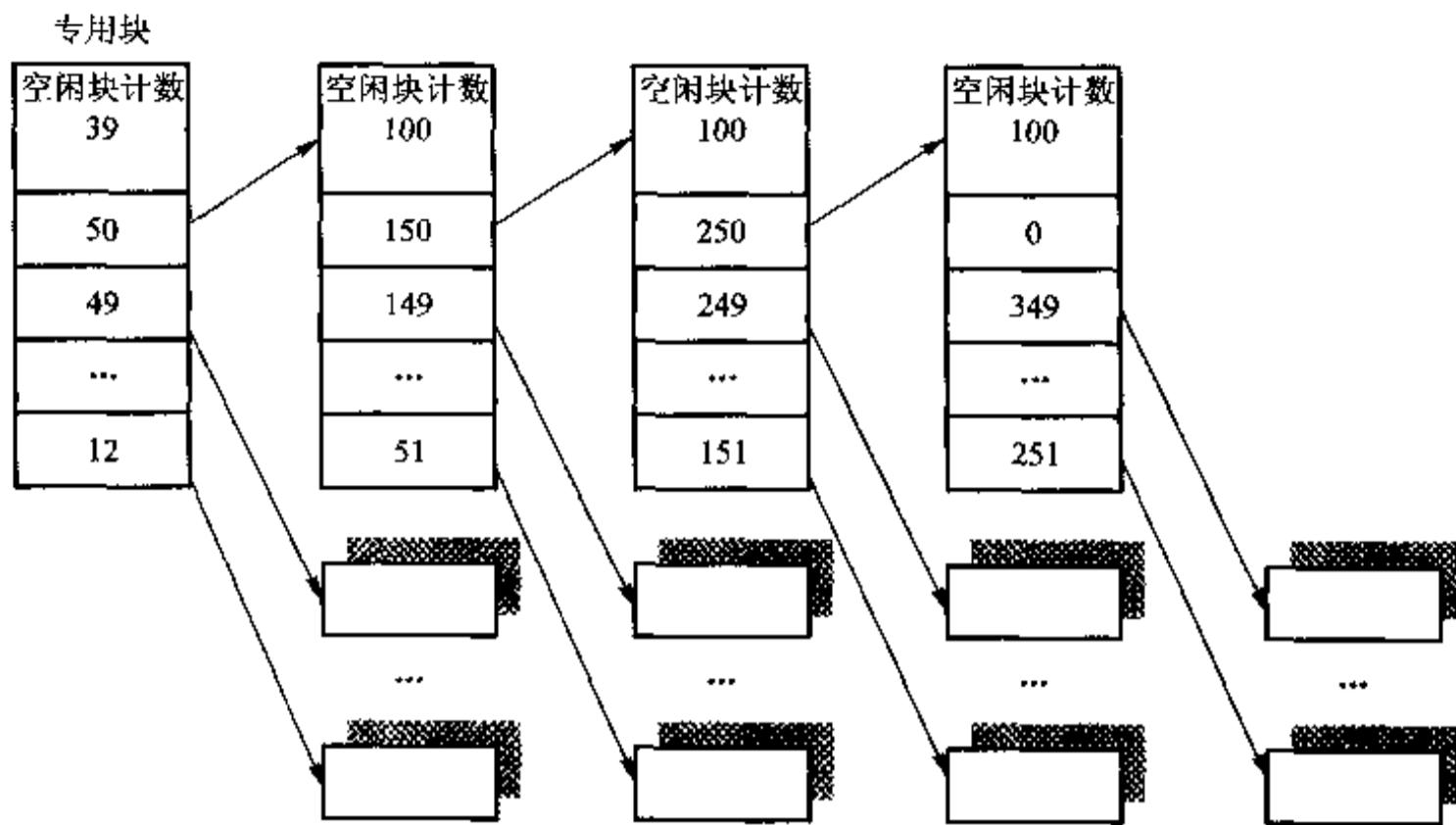


图 6.13 UNIX/Linux 系统的空闲块成组连接示意图

6.4.4 主存映射文件

首先,用于读写文件的操作在功能和格式上与读写主存的操作存在着很大的不同,如果能消除这种差异,就能简化编程。其次,文件中的数据分批在进程空间与磁盘空间之间传送,文件操作的实现不但复杂而且开销较大,能否找出一种方法,既降低开销又能通过直接读写主存来使用文件信息呢?针对这一点,由 MULTICS 首创通过结合虚拟存储管理和文件管理技术来提供一种文件使用方法,称为主存映射文件,UNIX/Linux 及 Windows 等现代操作系统都已实现这一功能。

采用虚拟存储管理的系统中,由于进程的地址空间非常大,可以把整个文件映射到一段进程虚拟地址空间中,然后,进程便可直接读写这段虚地址进行文件访问。主存映射文件(memory mapped file)具有把文件内容直接映射到进程虚拟地址空间的功能,即为进程分配一段虚地址空间,再把某个磁盘文件直接映射到此虚地址空间中。也就是说,把磁盘中的文件视为进程的虚拟地址的一部分,所以也称“映射文件 I/O”。因为主存映射文件是按照文件名来访问的,多个进程可同时把同一个文件映射到各自的虚拟地址空间中,且虚拟地址不必相同。随着进程的运行,被引用的映射文件部分由存储管理程序装入主存,多个进程读写虚拟地址的映射文件区,可以共享文件信息。其优点是:方便易用,节省空间,便于共享,灵活高效。

在分段系统中,映射文件成为一个新段。在分页系统中,根据文件大小,分配若干页表表项,这些页面与进程的其他页面一样进行处理,即如果页面不在主存储区,访问会触发缺页中断,系统便从磁盘文件(而不是页文件)中加载被引用的页面。反之,当页面从主存储区回收时,要把其

中的信息写回相应的磁盘文件。所以,每个页表表项指向一个页框,同时也指向一个磁盘块。对于主存映射文件而言,磁盘块属于被映射的文件,可使用和分页相同的底层机制来访问主存映射文件。

在具体实现上,系统提供新的系统调用,其中一个称映射文件 mmap,需要两个参数:文件名和虚拟地址,把一个文件映射到进程虚拟地址空间;另一个称移去映射文件 munmap,让文件与进程虚拟地址空间断开。例如,一个文件有 64 KB,被映射到 512 KB 的虚地址处,这样,在 512 KB 虚地址处读字节内容的任何机器指令会得到文件的 0 字节,而向 512 KB + 100 KB 虚地址处写入则修改文件的第 100 字节。当进程开始执行访问文件的操作时,例如,从 512 KB 开始读取信息,因为此时文件信息尚未调入主存,会触发缺页中断,系统装入文件的页面 0,接着就能够顺利地进行读操作;类似地,对 512 KB + 100 KB 的写操作会触发缺页中断,系统便装入文件中含有这一地址的页面,对主存的写操作可以开始。假如进程映射文件地址空间中的一个页而被替换算法逐出,它会被写回到磁盘文件的原来位置,当进程运行结束后,被映射且被修改的全部页面写回磁盘文件中。实际上,只需让进程所映射的进程虚拟地址空间的页表项中的辅存地址指向文件所在的盘块,就能很容易地实现映射文件和移去映射文件系统调用。

当多进程共享文件时,实现技术是把共享映射文件的进程的虚页面指向相同的页框,而页框中则保存磁盘文件的页面副本。图 6.14 是主存映射文件示意图,其优点为:进程读写虚存内容相当于执行文件读写操作,在建立文件映射之后,不再需要使用文件系统调用来读写数据,能够大大降低开销;在主存储区中仅需一个页面副本,既节省空间,又无须缓冲到主存的复制操作。

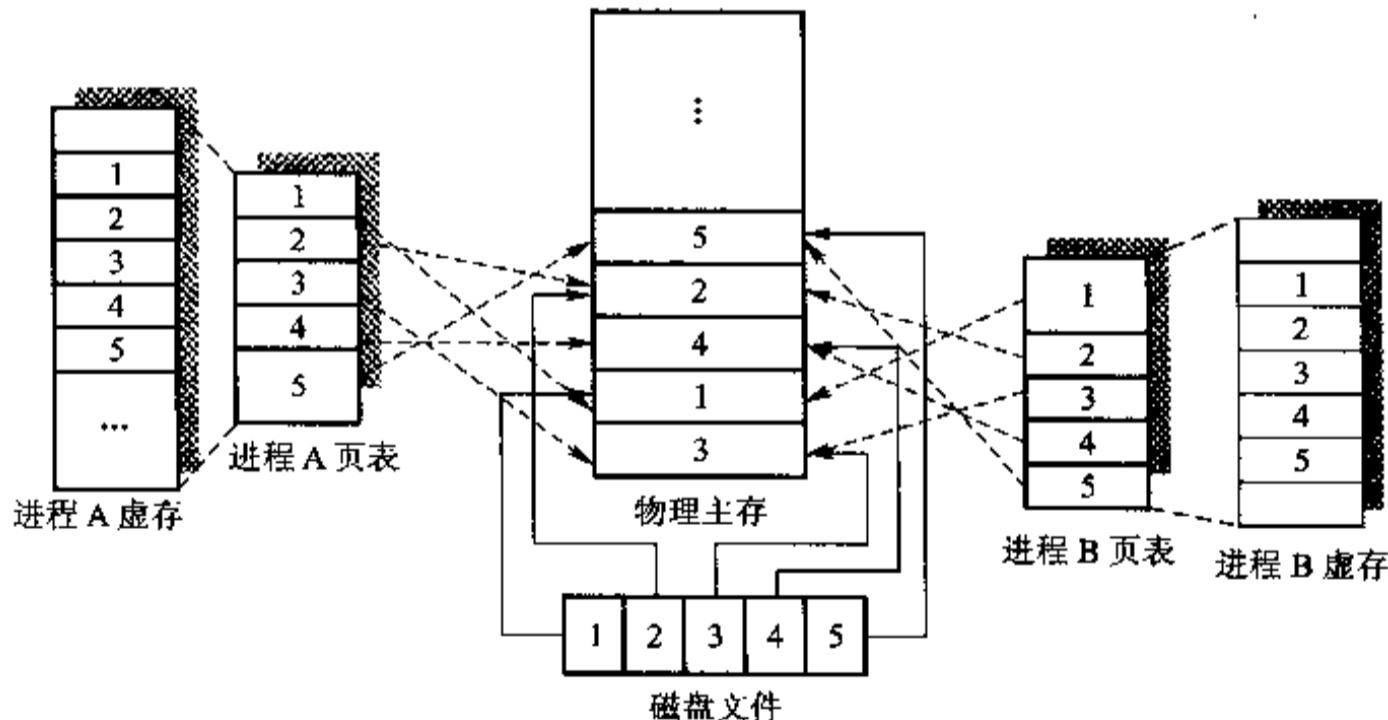


图 6.14 主存映射文件示意图

使用主存映射文件也会导致一些问题。首先,系统很难知道输出文件的确切长度,假设程序只引用页面 0,系统并不知道其中写入多少个字节,它所做的只是创建一个同页面长度一致的文件;其次,当一个文件被某个进程映射,而又被另一个进程以通常的读方式打开,如果第一个进程

修改一个页面,直到这个页面被换出,其修改内容才会在磁盘上反映出来,系统必须解决数据一致性问题;最后,文件可能比整个进程虚拟地址空间还要大,唯一的方法是安排系统调用映射文件的某一部分,而不是整个文件。尽管这种方法也能够工作,但较之映射整个文件而言并不令人满意。此外,共享主存映射文件并未为临界区的自动管理提供支持,如果它被一组进程共享,需要使用同步机制来确保临界区的正常使用。

6.4.5 虚拟文件系统

传统操作系统仅支持一种类型的文件系统,随着信息技术的发展和应用需求的增长,对文件系统的使用提出新的要求。例如,要求在 UNIX 系统中支持非 UNIX 类文件系统;Linux 系统在设计时便考虑能同时支持几十种文件系统。随着网络技术的发展,迫切要求计算机之间能够共享网络文件系统,甚至一些用户希望能定制专用的文件系统。

为了能同时支持多种文件系统,不同的操作系统采用不同的技术方案来提供虚拟文件系统,其目标是:把多种文件系统纳入统一框架,不同的磁盘分区可包含不同的文件系统,对它们的使用和传统的单一文件系统并无区别;用户可通过同一组系统调用对不同的文件系统及文件进行操作,系统调用可以跨物理介质和跨文件系统执行,如从一个文件系统复制或移动数据到另一个文件系统中,即提供对不同文件系统透明的相互访问;对网络共享文件提供完全的支持,访问远程节点上的文件应与访问本地节点的文件一致;提供对特殊文件系统(如 proc 及设备文件系统)的支持。

第一个虚拟文件系统在 1986 年由 Sun 公司开发成功,并在 SunOS 中使用。虚拟文件系统也称虚拟文件系统开关(Virtual File system Switch, VFS),它是内核的一个子系统,提供一个通用文件系统模型,概括所能见到的文件系统的常用功能和行为,处理一切与底层设备管理相关的细节,为应用程序提供标准接口(文件系统 API)。具体的文件系统不但依赖于 VFS 共存,而且也依靠 VFS 协同工作。其设计思想涉及以下三点。

1. 应用层

VFS 模型源于 UNIX 文件系统,使得用户可以直接使用标准 UNIX 文件系统调用来操作文件,无须考虑具体文件系统的特性和物理存储介质,通过 VFS 访问文件系统,才使得不同文件系统之间的协作性和通用性成为可能。

2. 虚拟层

在对具体文件系统的共同特性进行抽象的基础之上,形成与具体文件系统的实现无关的虚拟层,并在其上定义与用户的一致性接口。

3. 实现层

使用类似于开关表的技术进行具体文件系统的转接,实现各种文件系统的细节。实现层是自包含的,包含文件系统的各种实现设施,如超级块、节点区、数据区以及各种数据结构和文件类的操作函数。

VFS 实质上是一种存在于主存中的、支持多种类型文件系统的运行环境,其功能有:

- (1) 记录所安装的文件系统类型；
- (2) 建立设备与文件系统之间的联系；
- (3) 实现面向文件的通用操作；
- (4) 涉及特定文件系统的操作时，映射到具体的文件系统中去。

VFS 抽象层之所以能衔接各种不同的文件系统，是因为它定义了所有文件系统都支持的基本抽象接口和数据结构，同时文件系统也将自己的诸如“文件如何打开”、“目录如何定义”等概念在形式上与 VFS 的定义保持一致。对于像 FAT 和 NTFS 这类非 UNIX 风格的文件系统，必须经过封装，提供符合 VFS 概念的接口，比如，某文件系统不支持 inode 概念，它也必须在主存中装配 inode 结构体，如同其本身包含 inode 一样。这些装配和转换需要在使用现场引入特别处理，使得非 UNIX 文件系统能够兼容 UNIX 文件系统的使用规则，满足 VFS 的接口需求，这样一来，非 UNIX 文件系统便可与 VFS 共同工作，只是在性能上会产生少许影响。

6.5 Linux 文件系统

6.5.1 Linux 虚拟文件系统

Linux 最早采用 Minix 文件系统，20世纪90年代初推出专为自己设计的优秀、高效的文件系统 Ext(Extended File System)，目前常用的是 Ext2，最新升级版本是 Ext3。为了实现开放性，Ext 的设计有重大改进，引入虚拟文件系统(VFS)，为用户提供统一、抽象的文件系统界面，以支持多种文件系统。迄今所能支持的文件系统有 Ext、Ext2、Ext3、FAT、Minix、UMSDOS(DOS 和 Linux 共存文件系统)、proc(Linux 进程信息文件系统)、ISO9660(CD-ROM 文件系统)、HPFS(OS/2 文件系统)、ROMFS(只读主存文件系统)、Xenix(Xenix 文件系统)、NFS(Sun 网络文件系统)、UFS(BSD 文件系统)、SYSV(UNIX System V)等。更进一步地，还支持多种文件系统的互操作。图 6.15 给出 VFS 和具体文件系统之间的关系。

Linux 虚拟文件系统采用面向对象设计思想，虚拟文件系统相当于面向对象系统中的抽象基类，它可派生不同的子类，以支持多种文件系统。但从效率考虑，内核纯粹使用 C 语言编程，故并未直接利用面向对象的语义。下面的讨论中使用术语“对象”，实际上是一个结构体(struct)，它所代表的确实是一个对象(object)，虚拟文件系统由下列 4 个对象类组成。

(1) 超级块(super_block)对象：代表一个文件系统，存放已安装的文件系统信息。如果是基于磁盘的文件系统，此对象便对应于存放在磁盘上的文件系统控制块，每个文件系统都对应于一个超级块对象。

(2) 索引节点(inode)对象：代表一个文件，存放通用的文件信息。如果是基于磁盘的文件系统，此对象对应于存放在磁盘上的文件的 FCB，即每个文件的 inode 对象，而每个 inode 都有一个 inode 索引节点号，唯一标识某个文件系统中的指定文件。

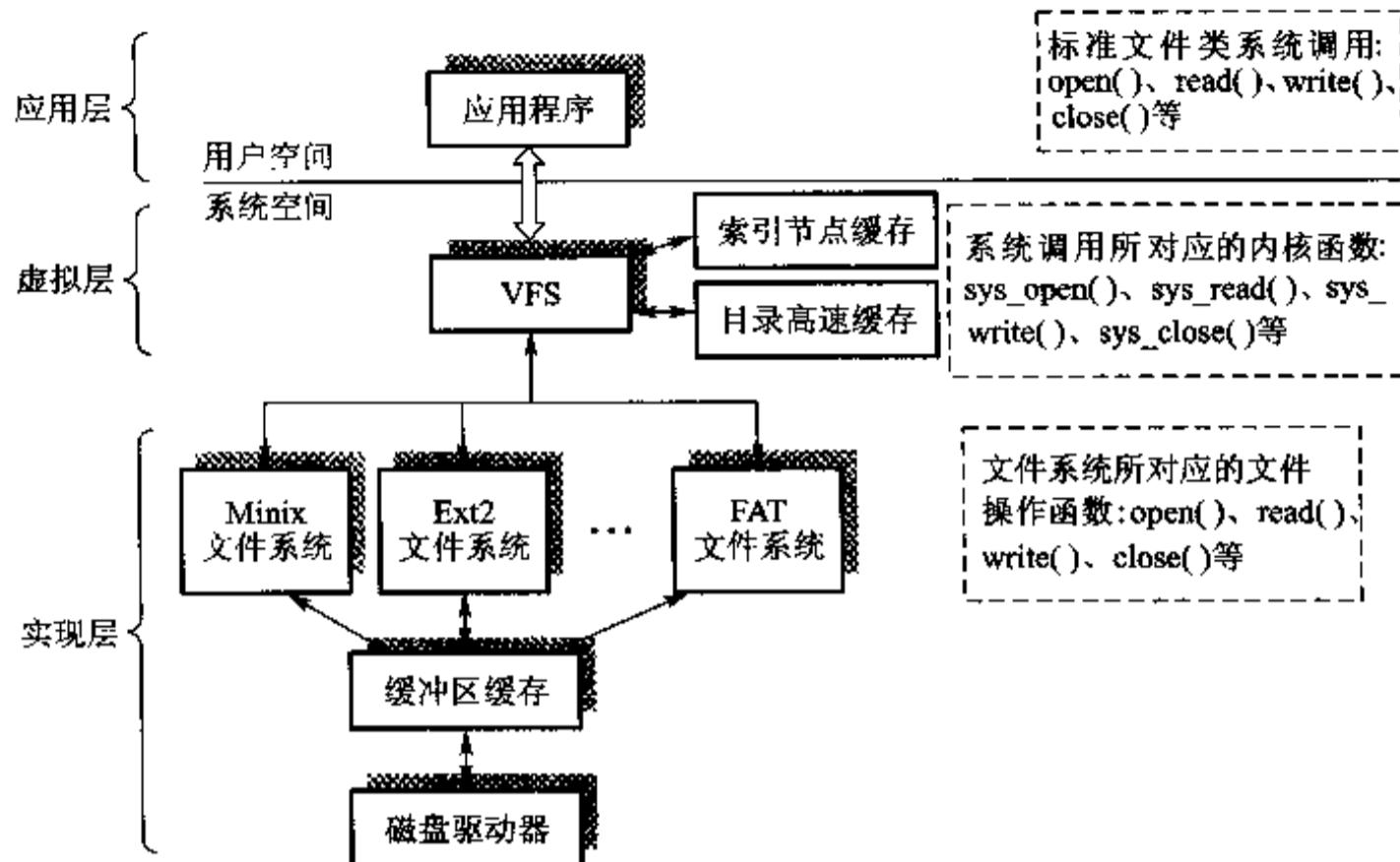


图 6.15 VFS 和具体文件系统的关系

(3) 目录项(dentry)对象:代表路径中的一个组成部分,存放目录项与对应文件进行链接的所有信息。虚拟文件系统把最近使用的 dentry 对象放在目录项高速缓存中,加快文件路径名的搜索过程,以提高系统性能。

(4) 文件(file)对象:代表已由进程打开的一个文件。存放已打开文件与进程的交互信息,这些信息仅当进程访问文件期间才存放于主存中。执行系统调用 `open()` 时创建文件对象,执行系统调用 `close()` 对其撤销。

每个主要的对象都包含一个操作对象,描述内核针对主要对象的可用方法。这些操作对象是:`:super_operation` 对象,其中包括内核针对特定文件所调用的方法;`inode_operation` 对象,其中包括内核针对特定文件所调用的方法;`dentry_operation` 对象,其中包括内核针对特定目录所调用的方法;`file_operation` 对象,其中包括进程针对已打开文件所调用的方法。

操作对象作为指针结构体被实现,此结构体包含指向操作其父对象的函数指针,其中的许多方法可继承使用虚拟文件系统的通用函数。若通用函数不能满足需要,就必须使用文件系统的独有方法来填充函数指针,使其指向文件系统实例。

如图 6.16 所示是 VFS 的各种对象——`files` 对象、`file` 对象、`dentry` 对象、`inode` 对象和 `super_block` 对象之间的关系。虚拟文件系统使用大量的结构体对象,除上述的 4 个主要对象之外,还有注册时所使用的 `file_system_type` 对象、安装时所使用的 `vfsmount` 对象等。

1. 超级块对象

超级块描述一个文件系统的信息,对于具体的文件系统而言,都有其超级块(如 Ext2 超级块),并被存放在磁盘的特定扇区上。当内核对一个具体文件系统进行初始化和注册时,调用

`alloc_super()` 函数为其分配一个 VFS 超级块，并从磁盘读取具体文件系统超级块中的信息填充进来，即 VFS 超级块在具体文件系统安装时才建立，并在卸载时自动删除，可见 VFS 超级块仅存于主存中。超级块的数据结构如下：

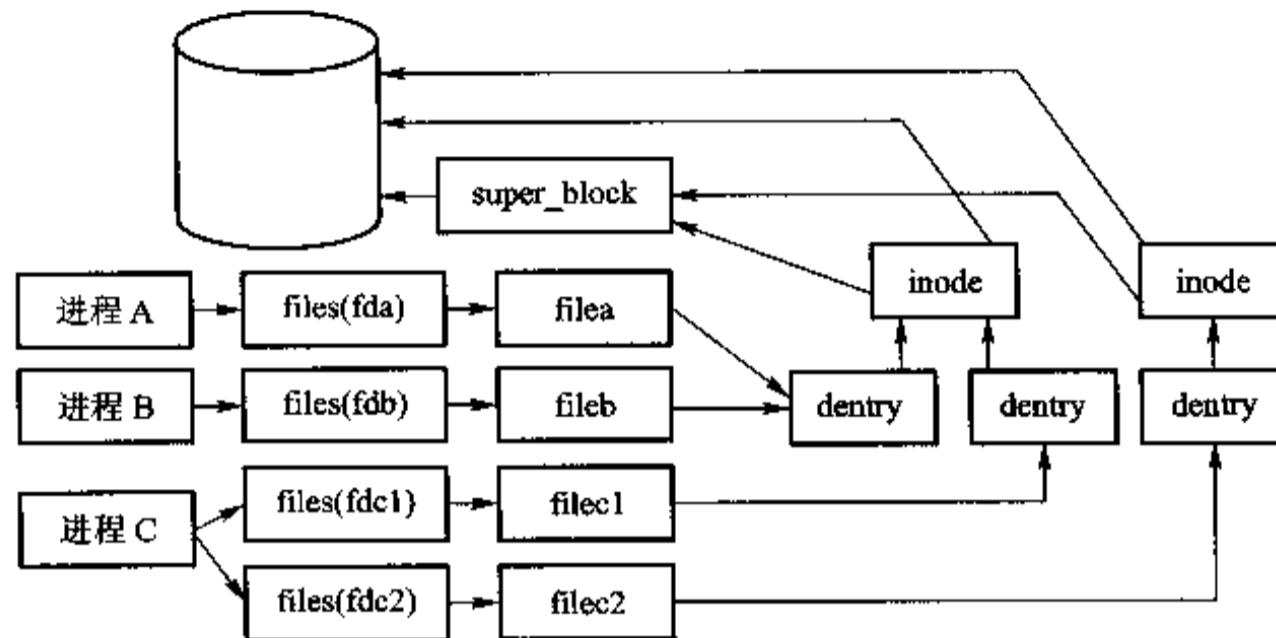


图 6.16 VFS 对象之间的关系

```
struct super_block {
    struct list_head s_list;          // 把所有超级块双向链接
    kdev_t s_dev;                   // 文件系统所在设备标识符
    unsigned long s_blocksize;        // 以字节为单位的盘块大小
    unsigned char s_blocksize_bits;   // 以 2 的幂表示的盘块大小,如盘块为 4KB,值为 12
    unsigned long s_maxbytes;         // 文件大小的上限
    unsigned char s_dirt;            // 修改(脏)标志
    ...
    struct file_system_type * s_type; // 指向注册表 file_system_type 结构的指针
    struct super_operations * s_op;   // 指向超级块操作函数集的指针
    struct dentry * s_root;          // 安装目录的目录项对象
    struct rw_semaphore s_umount;    // 卸载信号量
    struct semaphore s_lock;         // 超级块信号量
    int s_count;                    // 超级块引用计数
    struct list_head s_dirty;        // 脏节点 inode 链表
    struct list_head s_io;           // 回写链表
    char s_id[32];                  // 文本名
    ...
} union{ //联合体,其成员是各个具体文件系统的 fsname_sb_info 数据结构
    struct minix_sb_info minix_sb;
}
```

```

    struct ext2_sb_info ext2_sb;
    struct msdos_sb_info msdos_sb;
    ...
    | u;
};


```

被安装的具体文件系统都有一个 super_block 结构,以环状双向链表把它们链接在一起,指向此链表第一个元素和最后一个元素的指针存放在超级块的成员 s_list 域中。

联合体中的成员 super_block.u 是实现支持多种具体文件系统的关键,它指向各种具体文件系统的超级块,如当安装文件系统 MS-DOS 时,磁盘上的 msdos 超级块被复制到主存的 msdos_sb_info 结构体中,由 super_block.u.msdos_sb 指向此结构体,此后允许文件系统直接对主存超级块的 u 联合体进行操作,无须读盘。

与超级块关联的方法就是超级块操作对象,这些操作由 super_operation 结构来描述。

```

struct super_operation {
    void (* write_super)(struct super_block *);      // 把超级块写回磁盘
    void (* put_super)(struct super_block *);        // 释放超级块对象
    void (* read_inode)(struct inode *);             // 读取指定的 inode
    void (* write_inode)(struct inode *, int);        // 将给定 inode 写回磁盘
    void (* put_inode)(struct inode *);               // 逻辑上释放 inode
    void (* delete_inode)(struct inode *);            // 物理上释放 inode
    ...
};


```

结构体中的每一项是指向超级块操作函数的指针,而超级块操作函数执行文件系统和 inode 的低层操作。

2. 索引节点对象

inode 对象内包含内核在操作文件或目录时所需要的全部信息,文件名可更改,但 inode 对文件是唯一的,且伴随文件的存在而存在。对于 UNIX 类文件系统而言,这些信息从磁盘 inode 直接读入 VFS 的 inode 对象中。如果某文件系统没有 inode,那么,无论相关信息在磁盘上如何存放,都必须将其提取出来,并构造它的 inode。可以把具体文件系统存放在磁盘上的 inode 称为静态节点,其内容被读入主存 VFS 的 inode 后才能工作,后者也称为动态索引节点,其数据结构的主要域定义如下。

```

struct inode {
    struct list_head i_hash;           // 散列值相同的 inode 链表
    struct list_head i_list;          // 指向 inode 链表的指针
    struct list_head i_dentry;         // 同属一个 inode 的 dentry 链表
    unsigned long i_ino;              // inode 号
};


```

```

kdevt idev;           // 所在设备的设备号
umode_t i_mode;      // 文件类型及存取权限
nlink_t i_nlink;     // 连接到此 inode 的硬链接数
uid_t i_uid;          // 文件拥有者的用户 ID
gid_t i_gid;          // 用户所在组的 ID
loff_t i_size;        // 以字节为单位的文件大小
...
struct semaphore;    // inode 信号量
struct inode_operation * i_op; // 指向 inode 操作函数的指针
struct super_block * i_sb; // 指向文件系统超级块的指针
atomic_t i_count;      // 当前使用此 inode 的引用计数, 0 表示空闲
atomic_t i_writecount; // 写者计数
struct file_operation * i_fop; // 指向文件操作函数的指针
time_t i_atime; i_mtime; i_ctime; // 最近访问时间/修改时间/创建时间
struct pipe_inode_info * i_pipe; // 管道信息
struct page * i_pages; // 指向页结构的指针
unsigned char i_sock; // 套接字标志
unsigned long i_state; // inode 状态标志
unsigned int i_flags; // 文件系统标志
...
union{                // 联合体成员指向具体文件系统的 inode 结构
    struct minix_inode_info minix_i;
    struct ext2_inode_info ext2_i;
    struct msdos_inode_info msdos_i;
    ...
} u;
};

```

inode 代表文件系统中的文件, 也可以是设备、套接字或管道等特殊文件, 故 inode 中会包含特殊项。与 inode 关联的方法就是 inode 操作对象, 这些操作由 inode_operation 结构来描述。

```

struct inode_operation {
    int (*create)(struct inode *, struct dentry *, int);
    struct dentry * (*lookup)(struct inode *, struct dentry *);
    int (*link)(struct dentry *, struct dentry *);
    int (*symlink)(struct inode *, struct dentry *, const char *);
    int (*mkdir)(struct inode *, struct dentry *, int);
}

```

```

int (* rmdir)(struct inode *, struct dentry *);
...

```

|;

其中,主要函数的功能是:create()创建新的 inode,lookup()查找 inode 所在的目录,link()和 unlink()创建和删除一个硬链接,symlink()为符号链接创建 inode,mkdir()和 rmdir()为目录项创建和删除 inode。

3. 目录项对象

VFS 把每个目录看做一个文件,如在路径 /bin/vi 中,bin 和 vi 都是文件,bin 是目录文件,而 vi 是普通文件,路径中的每个组成部分都由一个 inode 对象表示。为了方便查找,VFS 引入目录项的概念,每个 dentry 代表路径中的一个部分,如 /、bin 和 vi 都是目录项对象,前两个是目录,后一个是普通文件。于是在路径查找中,VFS 为根目录 /、bin 和 vi 分别创建 3 个目录项对象,每个文件除了有一个 inode 数据结构外,还有一个 dentry 结构与之关联,dentry 结构中的 d_inode 指针指向相应的 inode 结构,引入 dentry 的主要目的是对目录进行缓存,加快文件定位,改进文件系统效率。dentry 结构代表逻辑意义上的文件,描述文件的逻辑属性,它在磁盘上并没有对应的映像;而 inode 结构代表物理意义上的文件,记录文件的物理属性,它在磁盘上有对应的映像。dentry 数据结构的定义如下。

struct dentry {	
atomic_t d_count;	//目录项引用计数
unsigned long d_vfs_flags;	//目录项缓存标志
unsigned int d_flags;	//目录项状态标志
struct inode * d_inode;	//dentry 所属的 inode
struct dentry * d_parent;	//父目录的目录项对象
struct list_head d_hash;	//目录项形成的哈希表
struct list_head d_lru;	//未用的 LRU 双向链表
struct list_head d_child;	//父目录的子目录项形成的双向链表
struct list_head d_subdirs;	//此目录项的子目录的双向链表
struct list_head d_alias;	//inode 别名的链表
int d_mounted;	//判断是否是安装点目录项
struct qstr d_name;	//目录项名,用于快速查找
unsigned long d_time;	//重新生效时间
struct dentry_operations * d_op;	//操作目录项的函数
struct super_block * d_sb;	//指向文件的超级块
struct hlist_node d_hash;	//哈希表
struct hlist_head * d_hucket;	//哈希表头
void * d_fsdata;	//文件系统特殊数据

```

unsigned char d_iname[DNAME_INLINE_LEN]; //文件名的前 15 个字符
...
|;

```

一个有效的 dentry 结构必定对应于一个 inode 结构,这是因为目录项要么代表一个目录,要么代表一个文件,目录实际上也是文件,只要 dentry 结构有效,则其指针 d_inode 必定指向一个 inode 结构。反之不然,一个 inode 可能对应多个 dentry 结构,也就是说,一个文件可以有多个文件名或路径名,这是因为已经建立的文件可被链接至其他文件名。所以,在 inode 结构中有一个队列 i_dentry,凡代表同一个文件的所有目录项都通过其 dentry 结构中的 d_alias 域链入相应 inode 结构中的 i_dentry 队列。

在内核中有一个散列表 dentry_hashtable,这是一个 list_head 的指针数组,一旦在主存中建立一个目录节点的 dentry 结构,就通过其 d_hash 域链入散列表中的某个队列中。内核中还有一个队列 dentry_unused,凡是已经没有用户使用的 dentry 结构就通过其 d_lru 域链入空闲队列。dentry 结构中除了 d_alias、d_hash 和 d_lru 这 3 个队列外,还有 d_vfsmntd_child 及 d_subdirs 队列,d_vfsmntd_child 仅在 dentry 为安装点时使用;当此目录节点有父目录时,则其 dentry 结构就通过 d_child 链入其父节点的 d_subdirs 队列中,同时,又通过 d_parent 指向其父目录的 dentry 结构,而其自身各个子目录的 dentry 结构则挂在其 d_subdirs 域所指向的队列中。

可见一个文件系统中的所有目录项结构,或组织成一个散列表,或组织成一棵树,或组织成一个链表,这为文件访问和文件路径搜索奠定了良好的基础。与目录项相关联的方法就是目录项操作对象,操作由 dentry_operation 结构来描述。

```

struct dentry_operation {
    int (*d_revalidate)(struct dentry *, int);
    int (*d_hash)(struct dentry *, struct qstr *);
    int (*d_compare)(struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete)(struct dentry *);
    int (*d_release)(struct dentry *);
    int (*d_iput)(struct dentry *, struct inode *);
    ...
};

```

其中,主要函数的功能是:d_revalidate()判定目录项是否有效,d_hash()生成散列值,d_compare()比较两个文件名,d_delete()删除 d_count 为 0 的目录项对象,d_release()释放目录项对象,d_iput()丢弃目录项所对应的 inode。

4. 与进程相关的文件系统数据结构

(1) 系统打开文件表的 file 结构

每个文件都用一个 32 位数字来表示下一个读写的字节位置,通常称其为文件位置或偏移量 (offset)。每当打开一个文件时,偏移量置 0,读写操作便从这里开始,允许通过系统调用 lseek 对

文件位置进行随机定位。Linux 建立文件对象(file)来保存打开文件,file 结构除了保存文件的当前位置之外,还把指向此文件的 inode 指针也放在其中,并形成一个双向链表,称为系统打开文件表。每当打开文件时,就要创建一个 file 结构,其定义如下。

```
struct file {
    struct list_head f_list;           //所有打开文件形成的链表
    struct dentry * f_dentry;          //指向相关目录项的指针
    struct vfsmount * f_vfsmnt;        //指向 VFS 安装点的指针
    struct file_operations * f_op;      //指向文件操作函数的指针
    unsigned long f_reada;             //预读标志
    unsigned long f_ramax;             //预读的最多页数
    unsigned long f_raend;             //上次预读后的指针
    unsigned long f_ralen;             //预读的字节数
    unsigned long f_rawin;             //预读的页数
    mode_t f_mode;                   //文件访问模式
    loff_t f_pos;                    //文件的当前偏移量
    unsigned short f_count;            //使用此文件的进程数
    unsigned int f_uid;                //使用者的用户标识
    unsigned int f_gid;                //使用者的用户组标识
    ...
};
```

每个文件对象总是包含在如下的一个双向环状链表中。

①“未使用”文件对象链表:此链表可用做文件对象的主存缓冲区,通常设置为可放 10 个对象,且其中必须包含 NR_RESERVED_FILES 对象。

②“正在使用”文件对象链表:此链表中的每个元素至少由一个进程使用,故各个元素的 f_count 域必定非空。如果 VFS 需要分配一个新的文件对象,就调用函数 get_empty_file(),检测“未使用”文件对象链表的元素是否多于 NR_RESERVED_FILES,如果是,则选择一个元素使用,否则退回到正常的主存分配。

与文件相关联的方法是文件操作对象,由 file_operations 结构描述。

```
struct file_operations {
    loff_t( * llseek)(struct file *,loff_t,int);
    ssize_t( * read)(struct file *, char *,size_t,loff_t *);
    ssize_t( * aio_read)(struct iocb *, char *,size_t,loff_t *);
    ssize_t( * write)(struct file *, const char *,size_t,loff_t *);
    ssize_t( * aio_write)(struct kiocb *, const char *,size_t,loff_t *);
    int ( * mmap)(struct file *,struct vm_area_struct *);
};
```

```

int (* open)(struct inode *, struct file *);
int (* ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
int (* flush)(struct file *);
int (* release)(struct dentry *);
int (* fsync)(struct file *, struct dentry *, int datasync);
...
};


```

其中,主要函数的功能是:llseek()修改文件指针,read()从文件的偏移处读出若干字节,write()向文件的指定偏移处写入若干字节,aio_read()以异步方式从文件的偏移处读取若干字节,aio_write()以异步方式向文件的指定偏移处写入若干字节,mmap()文件到主存的映射,open()打开一个文件,ioctl 向硬设备发送命令,flush()关闭文件时减少 f_count 计数,release()释放文件对象,fsync()将文件在缓冲区中的数据写回磁盘。

(2) 用户打开文件表的 files_struct 结构

文件描述符 fd 用来描述打开的文件,每个进程用一个 files_struct 结构来记录文件描述符的使用情况,这个结构称为用户打开文件表。指向此结构的指针被保存在进程的 task_struct 结构的成员 files 中。此结构定义如下:

```

struct files_struct{
    atomic_t count;           //共享此表的进程数
    rwlock_t file_lock;       //保护此结构体的锁
    int max_fds;              //进程当前所具有的最大文件数
    int max_fdset;             //当前文件描述符的最大数
    int next_fd;               //已分配的最大文件描述符加 1
    struct file ** fd;         //指向文件对象(系统打开文件表项)的指针数组
    fd_set * close_on_exec;    //指向执行 exec() 时所需关闭的文件描述符
    fd_set * open_fds;          //指向打开文件的描述符的指针
    fd_set close_on_exec_init; //执行 exec() 时需关闭的文件描述符初值集
    fd_set open_fds_init;      //文件描述符初值集
    struct file * fd_array[32]; //指向文件对象的初始化指针数组
};


```

fd 是指向文件对象的指针数组中的指针,数组的长度存放于 max_fds 域中,fd 域通常指向 files_struct 结构的 fd_array 域,此域包含 32 个文件对象指针。如果进程打开的文件数目大于 32,内核分配一个新的文件指针数组,并将其地址存放在 fd 域中,同时更新 max_fds 域的值。

对于在 fd 数组中有人口地址的文件而言,数组下标就是文件描述符,数组的第一个元素(索引为 0)、第二个元素(索引为 1)和第三个元素(索引为 2)分别表示标准输入文件、标准输出文件和标准错误文件,且这 3 个文件通常从父进程处继承而来。通过适当的系统调用,两个文件描述

符可指向同一个打开的文件,亦即数组的两个元素可指向同一个文件对象。

注意区分系统打开文件表和用户打开文件表,前者是由 file 对象所组成的链表,它处于核心态,由内核控制;而后者是文件描述符表,在用户进程空间,是进程的私有数据,因而,应用程序只能用文件描述符作为参数的系统调用才能访问系统打开文件表。

(3) 进程工作的文件系统信息(根目录和当前工作目录)fs_struct 结构

当程序通过文件名访问文件时,内核首先通过遍历文件系统树找到对应的目录,然后在此目录中找到文件名和存放它的 inode 号。有了 inode 号,就能确定文件在磁盘上的物理位置,为此,每个进程都有一个当前工作目录和当前工作目录所在文件系统的根目录,这是进程状态的一部分,以便用户既可使用相对路径名也可使用绝对路径名来访问所需要的文件。后面图 6.17 中的 fs_struct 结构体就含有指向当前进程的工作目录的 inode 指针 pwd,以及指向当前工作目录所在文件系统的根目录 inode 指针 root。

对一个文件进行访问时,搜索的起点要视路径名是相对路径名还是绝对路径名而定。如果是相对路径名,则从当前目录开始搜索;如果是绝对路径名,则从根目录开始搜索,因为相同的文件名允许出现在不同的目录中,说明只有文件名还不能唯一地确定一个文件,因此绝对路径名才可以唯一地指定一个文件。要求绝对路径名给出目录树中从根目录访问文件的路径上的所有节点。

在很多情况下可以不给出全路径名,在系统正常运行后的任意时刻,每个用户都有一个正在使用的当前工作目录,如果路径名不以“/”开头,则路径被认为是以当前工作目录开头,这就是相对路径名,用户登录时与 shell 相关联的当前工作目录成为用户的主目录。fs_struct 结构的定义如下。

```
struct fs_struct {
    atomic_t count;           //共享 fs_struct 结构的进程数
    rwlock_t lock;            //保护此结构体的锁
    int umask;                //默认的文件访问权限掩码
    struct dentry *root;       //根目录的目录项对象
    struct dentry *pwd;        //当前工作目录的目录项对象
    struct dentry *altroot;    //可替换的根目录的目录项对象
    struct vfsmount *rootmnt; //根目录的安装点对象
    struct vfsmount *pwdmnt;  //当前工作目录的安装点对象
    struct vfsmount *altrootmnt; //可替换的根目录的安装点对象
};
```

fs_struct 中的 dentry 结构是对一个目录项的描述,root、pwd 和 altroot 这 3 个指针都指向这个结构,在进程实际运行时,这 3 个目录不一定都在同一个文件系统中。例如,进程的根目录通常安装于“/”节点上的 Ext2 文件系统,而当前工作目录可能安装于/msdos 目录下的 DOS 文件系统。因而,fs_struct 结构中的 rootmnt、pwdmnt 及 altrootmnt 就是对上述 3 个目录的安装点的

描述,安装点的数据结构类型为 `vfsmount`。

5. Linux 文件系统的逻辑结构

Linux 中的每个进程都有两个数据结构来描述进程与文件的相关信息,一个进程所处的位置由 `fs_struct` 结构来描述,它包含两个指向 VFS `inode` 的指针,分别指向根目录节点(`root`)和当前目录节点(`pwd`);而进程打开的文件由 `files_struct` 结构来描述,最多能同时打开 256 个文件,由 `fd[0]~fd[255]` 所表示的指针指向对应的 `file` 结构。每当打开一个文件时,就从 `files_struct` 结构中寻找一个空闲的文件描述符,使其指向打开文件的描述结构 `file`,对文件的操作通过 `file` 结构中定义的文件操作函数和 VFS `inode` 的信息来完成,整个系统打开的文件则由 `file` 结构来描述。图 6.17 给出这些数据结构之间的关系。

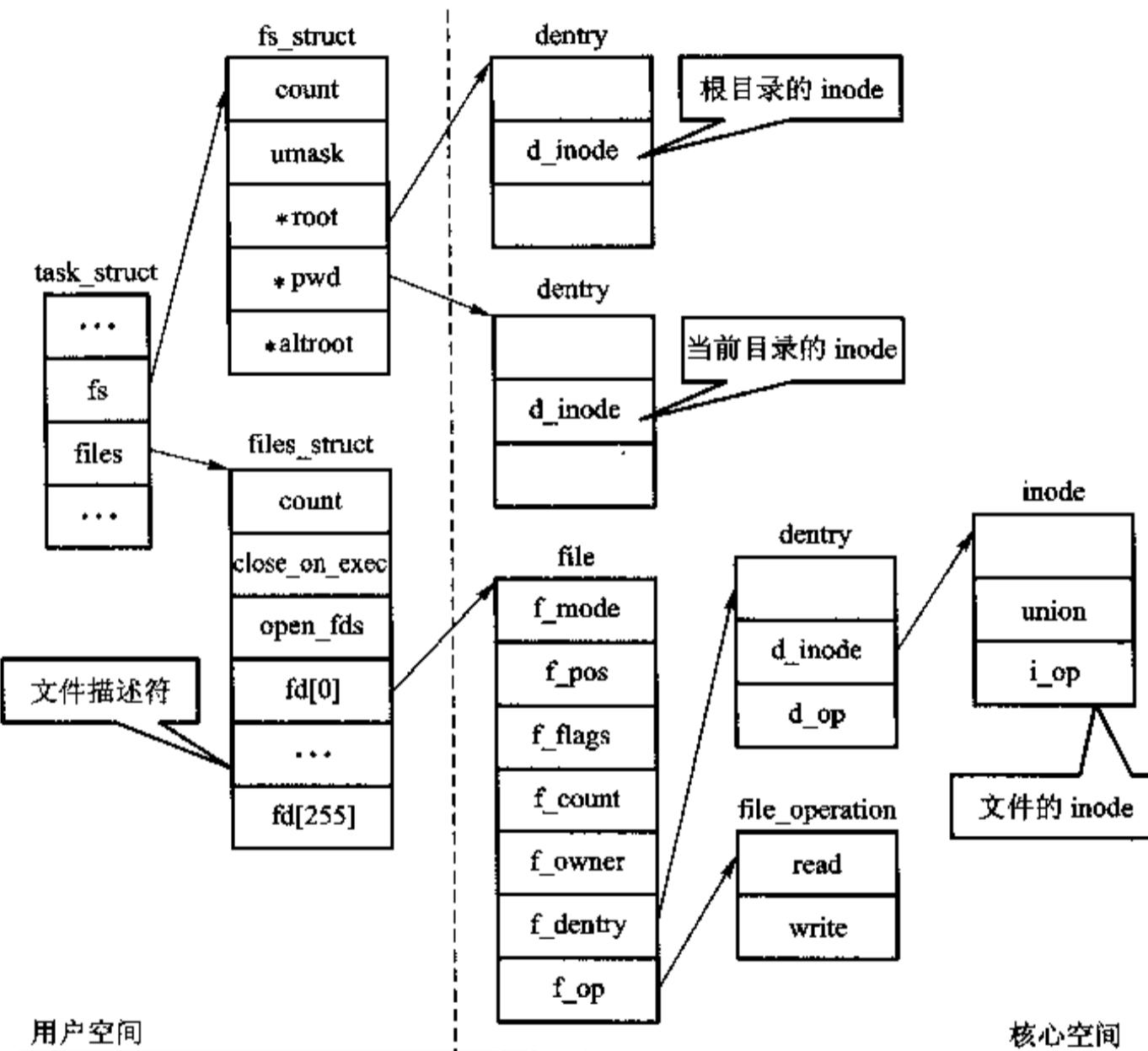


图 6.17 Linux 文件系统逻辑结构

有了 `fs_struct`、`file` 和 `files_struct` 结构,可通过两种途径共享文件,一是多个进程共享同一个 `file` 结构,`file` 唯一对应于一个文件;二是多个进程的 `file` 结构共享一个 `inode`,`inode` 唯一对应于一个文件。前者在父子进程间发生,当调用 `fork` 生成一个子进程时,子进程复制父进程的文件描述符,两者有相同的用户打开文件表,都有项指向同一个 `file` 结构。后者通过文件的 `link` 机制实

现,两个独立进程打开同一文件也属于这种共享,系统中专门设计文件锁,解决两个进程同时读写文件时出现的互斥问题。

6.5.2 文件系统的注册与注销及安装与卸载

1. 文件系统的注册与注销

同其他操作系统一样,Linux 支持多个物理磁盘,每个磁盘可划分为一个或多个磁盘分区,每个分区上可建立一个文件系统。一个安装好的 Linux 操作系统究竟支持多少种不同类型的文件系统,是通过文件系统类型注册链表来描述的,VFS 以链表形式管理已注册的具体文件系统。向系统注册文件系统类型有两种途径,一是在编译操作系统内核时确定可支持哪些文件系统,在文件系统被引导时,在 VFS 中进行注册;二是文件系统当做可装载模块,通过 insmod/rmmmod 命令在装入文件系统模块时向 VFS 注册/注销。

每个文件系统都有其初始化函数,用于向 VFS 注册,即填写由 file_systems 所指向的文件系统注册表数据结构 file_system_type,每个文件系统类型在注册表中有一个登记项,记录文件系统的类型、特性、指向对应的 VFS 超级块读取函数的地址以及已注册项的链指针等。

```
struct file_system_type {
    const char * name;           //文件类型名称
    struct super_block * (*read_super)(struct super_block *, void *, int);
    struct file_system_type * next;
    ...
};
```

函数 register_filesystem()用于注册文件系统类型,函数 unregister_filesystem()用于从注册表中注销一个文件系统类型。read_super()函数完成以下功能:从磁盘文件系统中读取给定文件系统数据;文件管理器把这些数据翻译成独立于设备的可用信息;把信息存入 super_block 结构体。

2. 文件系统的安装与卸载

Linux 系统不通过设备标识和访问某个文件系统,而是通过命令将其安装到文件系统树状目录结构的某个目录节点 vfsmount,于是文件系统的所有文件和子目录就是此目录节点的文件和子目录,直到用命令显式地卸载这个文件系统。安装 Linux 时,磁盘中的一个分区已经安装 Ext2,它是作为根文件系统在启动时自动安装的。如果超级用户欲安装另外的具体文件系统,需要指定文件系统类型名、所在的物理设备名、安装点等信息,再用 mount 命令进行安装。

安装文件系统时,内核首先检查参数的合法性,VFS 通过查找由 file_systems 所向的注册表,寻找匹配的 file_system_type。当找到匹配项时,就可获得读取文件系统超级块函数的地址;接着查找作为新文件系统安装点的 VFS inode,且同一目录下不能安装多个文件系统;VFS 安装程序必须分配一个 VFS 超级块,利用 read_super() 函数读入安装文件系统的辅存超级块,并进行填充;再申请一个 vfsmount 数据结构(其中包含文件系统所在的块设备的标识、安装点、指向 VFS

超级块的指针等),使其指针指向所分配的 VFS 超级块。当文件系统安装后,它的根 inode 便常驻在 inode 高速缓存中。

使用 umount 卸装某个文件系统时,首先检查文件系统是否是可卸载的。当文件系统正在被使用时,此文件系统不能卸载;如果文件系统中的文件或目录正在使用,则 VFS inode 缓存中可能包含对应的 VFS inode。内核根据文件系统所在设备的标识,检查在缓存中是否有来自此文件系统的 inode,如果有且使用计数大于 0,则此文件系统不能卸载;否则查看对应的 VFS 超级块的标志,如果其值为“脏”,必须把 VFS 超级块写回磁盘。上述过程结束后,释放对应的 VFS 超级块,安装点数据结构从 vfsmntlist 中断开并释放,从而卸载文件系统。

6.5.3 文件系统的缓存机制

Linux 支持多种类型的文件系统,且能保持很高的性能,探究其原因,除了 VFS 之外,多种复杂的高速缓存也起到了关键作用。

1. VFS inode 缓存

VFS 提供缓存,把当前使用的 inode 保存起来,同时为了从中快速找到所需要的 inode,还采用散列技术,其基本思想是:根据文件系统所在的逻辑设备号和 inode 号,计算出每个已分配的 inode 的散列值,凡具有相同散列值的 inode 均被链入同一个队列中。系统建立一张散列表,其中每项包含一个指向 VFS inode 散列队列的头指针。

当 VFS 按需计算一个散列值时,就将其作为访问散列表的索引,从散列表得到指向相应的 inode 队列的指针。如果所指队列中包含想要查找的 inode,只需将此 inode 访问计数加 1,表明又有一个进程使用此 inode;否则,必须找出一个空闲的 VFS inode,且从具体文件系统中读取此 inode,把新的 VFS inode 加入对应的散列队列中。

2. VFS 目录高速缓存

通过路径名查找目录文件的 inode 是一个使用频率极高的操作,为了提高此类操作的执行效率,系统维护表达路径与 inode 对应关系的 VFS 目录缓存,其中存放被访问过的目录,这样当同一目录被再次访问时,就可快速获得。目录缓存也采用散列表的方法管理,表的每项是一个指针,指向具有相同散列值的目录缓存队列,而散列值利用文件系统所在的逻辑设备号和目录名计算出来。由于高速缓存的容量有限,VFS 便采用 LRU 算法替换缓存中的目录项,具体方法是:VFS 维护两级 LRU 链表,当第一次查找一个目录项时,此目录项就被放入目录缓存中,同时进入第二级 LRU 链表的末尾;如果此时缓存已满,将替换 LRU 链表中的队首目录项,并将其放入目录缓存中,以后此目录项再次被存取时,它将被提升并链入第一级 LRU 链表的末尾;类似地,如果此时缓存已满,则替换此 LRU 链表的队首目录项,这样越是最近使用的目录项越靠近链尾,只有最近不常用的目录项才逐步移至链表头部,最终被替换。

3. 页高速缓冲区

Linux 维护一组页缓冲区,它独立于任何类型的文件系统,被所有的物理设备所共享,采用页缓冲区有两个优点:数据一经使用,就在页缓冲区中留下备份,由于局部性原理,在从页缓冲区

清除以前,再次使用时可直接找回,避免不必要的磁盘 I/O 操作;“脏”页写回磁盘时,可适当排序,实现磁盘驱动调度优化。

页缓冲区由主存物理页框组成，缓冲区中的一页对应于磁盘上的多个块。例如，块大小为 1 KB，则一页对应于 4 个块，所以，页缓冲区实际所缓存的是以页为单位的文件块。页缓冲区中的页来自读写普通文件、块设备文件和主存映射文件，在读写 I/O 操作之前，内核会检查数据是否已驻留在页缓冲区，以决定是否访问磁盘。Linux 页面缓冲由 address space 结构体描述。

```
struct address_space {
    struct inode * host;                                // 属主的 inode
    struct list_head clean_pages;                      // 干净页面链表
    struct list_head dirty_pages;                     // 肮脏页面链表
    struct list_head locked_pages;                   // 锁定页面链表
    struct list_head io_pages;                         // I/O 使用页面链表
    struct address_space_operations * a_ops;           // 操作函数表
    struct list_head i_mmap;                           // 私有映射链表
    struct list_head i_mmap_shared;                  // 共享映射链表
    struct semaphore i_shared_sem;                  // 保护链表信号量
    unsigned long nrpages;                            // 页面总数
    ...
    struct list_head private_list;                  // 私有 address_space 链表
    struct address_space * assoc_mapping;            // 缓冲区
};
```

`address_space` 结构与某些对象相关联,这些对象就是页面内数据的属主,可能是普通文件、目录、块设备文件,也可能是交换区,通常是一个文件(一个 `inode`),这时 `host` 域指向此 `inode`,除此以外的情况,`host` 域设置为 `NULL`。`a_ops` 指向地址空间对象中的操作函数表,它由 `address_space_operations` 结构表示。

```
struct address_space_operations {  
    writepage(); //把页写入磁盘  
    readpage(); //从磁盘读入页  
    sync_page(); //启动页中所安排的 I/O 操作,传输数据  
    prepare_write(); //准备写操作  
    commit_write(); //完成写操作  
    bmap(); //从文件块索引获得逻辑块号  
    flushpage(); //删除来自磁盘的页  
    releasepage(); //日志文件系统准备释放页  
    direct_io(); //数据页的直接 I/O 传输
```

；

6.5.4 Ext2 文件系统

Ext2(second extended file system)文件系统支持标准 UNIX 文件类型,包括普通文件、目录文件、特别文件和符号链接文件。Ext2 文件系统可管理特大磁盘分区,文件系统最大可达 4 TB。此外,还支持长文件名(最长 1 012 个字符)、可选的逻辑数据块大小,提供数据更新时同步写入磁盘和快速符号链接功能。

Ext2 文件系统的信息都保存在数据块中,数据块的长度相同,但不同 Ext2 文件系统中的数据块大小可以不同,其物理结构如图 6.18 所示。Ext2 所占用的磁盘除了引导块外,逻辑分区划分为块组,每个块组依次包括超级块、块组描述符表、块位示图、inode 位示图、inode 表以及数据块区,重复保存有关文件系统的关键信息及所存储的文件和目录信息。文件系统保存逻辑块号,由块设备驱动程序将逻辑块号转换成块设备的物理存储位置。引导块是磁盘上的第一个数据块,只有根文件系统才有引导程序放在这里,Linux 以 Ext2 作为其根文件系统。

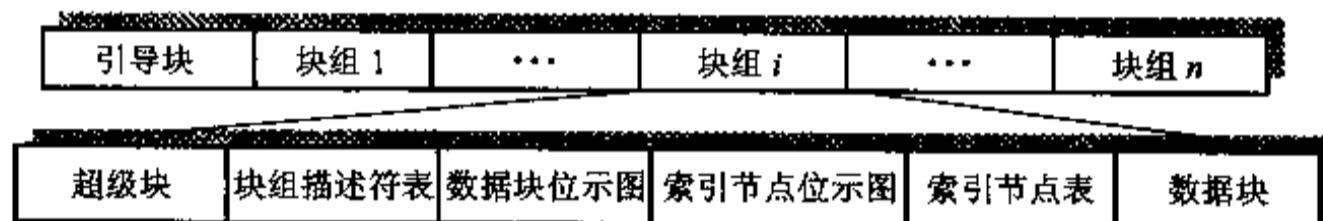


图 6.18 Ext2 文件系统结构

采用块组划分的目的:一是提高文件系统的可靠性,每个块组中都有管理信息的副本,当文件系统崩溃时可以容易地恢复;二是提高文件系统的性能,由于块组内的数据块靠近其 inode,文件 inode 靠近其目录 inode,从而将磁头定位时间缩减至最短,加快磁盘访问速度。

1. 超级块

Ext2 超级块用来描述目录和文件在磁盘上的静态分布情况,包括尺寸和结构,每个块组都有一个超级块,只有块组 1 的超级块才被读入主存工作,直至卸载,其他块组的超级块仅作为恢复备份。超级块主要包括:块组编号、块数量、块长度(1 KB~4 KB)、空闲块数量、inode 数量、空闲 inode 数量、第一个 inode 号、第一个数据块位置、每个块组中的块数、每个块组的 inode 数,及安装时间、最后一次写时间、安装信息、文件系统状态信息等内容。

2. 块组描述符

每个块组都有一个块组描述符,记录此块组的以下信息。

(1) 数据块位示图

表示数据块位示图占用的块号,此位示图反映块组中数据块的分配情况,在分配或释放数据块时需使用数据块位示图。

(2) inode 位示图

表示 inode 位示图占用的块号,此位示图反映块组中 inode 的分配情况,在创建或删除文件

时需使用 inode 位示图。

(3) inode 表

块组中 inode 所占用的数据块数,系统中的每个文件对应于一个 inode,每个 inode 都由一个数据结构来描述。

(4) 空闲块数、空闲 inode 数和已用数目。

一个文件系统中的所有块组描述符结构组成一个块组描述结构表,每个块组在其超级块之后都包含一个块组描述结构表的副本。实际上,Ext2 文件系统仅使用块组 1 中的块组描述结构表。

3. 索引节点

在 Ext2 中,每个文件都由一个 inode 来唯一描述,每个块组的 inode 集中存放在一个 inode 表中,每个 inode 有唯一的 inode 号。inode 位示图记录 inode 的分配情况,inode 起着文件控制块的作用,可用来对文件进行控制和管理。进一步可区分为磁盘 inode 和主存活动 inode。创建文件时就分给一个磁盘 inode,当文件被打开时,其对应的磁盘 inode 被复制到活动 inode 中;当文件关闭时,其活动 inode 的内容写回磁盘 inode,并释放此活动 inode。

4. 数据块分配策略

文件空间的碎片是每个文件系统都需要解决的问题,是指系统经过一段时间的读写后,导致文件的数据块散布在磁盘空间的各处,访问这类文件时,致使磁头移动次数急剧增多,访问速度大幅下降。操作系统提供“碎片合并”实用程序,定时运行可把碎片集中起来,Linux 的碎片合并程序叫做 defrag(defragmentation program)。操作系统通过分配策略避免碎片的发生则更加重要,Ext2 采用两种策略来减少文件碎片。

(1) 原地先查找策略

为文件新数据分配数据块时,尽量先在文件原有数据块的附近查找。首先试探紧跟文件末尾的那个数据块,再试探位于同一个块组的相邻的 64 个数据块,接着就在同一个块组中寻找其他空闲数据块;实在不得已才搜索其他块组,且首先考虑 8 个一簇的连续的块。

(2) 预分配策略

如果 Ext2 引入预分配机制,就从预分配的数据块中取一块来用,紧跟此块后的若干数据块如果空闲,也被保留下来。当文件关闭时,仍保留的数据块予以释放,这样保证尽可能多的数据块集中成一簇。Ext2 的 inode 数据结构的 ext2_inode_info 域中包含两个属性 prealloc_block 和 prealloc_count,前者指向可预分配数据块链表中第一块的位置,后者表示可预分配数据块的总数。

6.6 Windows 2003 文件系统

6.6.1 文件系统概述

Windows 支持传统的 FAT 文件系统,包括 FAT12、FAT16 和 FAT32,还支持光盘文件系统

CDFS、通用磁盘格式 UDF、高性能文件系统 HPFS 等文件系统。从 Windows NT 开始提供一种新的文件系统 NTFS(New Technology File System,新技术文件系统),它除了克服 FAT 系统容量的不足之外,出发点是设计服务器端适用的文件系统,保持向下兼容性,有较好的容错性和安全性。NTFS 具有一系列新的特性:可恢复性、高安全性、文件加密/解密、大磁盘和大文件、基于 Unicode 的文件名等。此外,还有动态添加卷磁盘空间、动态坏簇重映射、文件数据和目录压缩技术、分布式链接跟踪和 POSIX 标准支持。

6.6.2 NTFS 在磁盘上的结构

1. MFT 的结构

物理磁盘可组织成一个或多个卷,卷与磁盘逻辑分区有关。NTFS 以簇为单位管理磁盘空间,每个簇包含 2 的整数次幂个扇区,扇区是磁盘的最小物理存储单位。卷上簇的大小在执行格式化命令时确定,扇区通常为 512 B,每个簇中的扇区数可以是 1 个、2 个直至 128 个,故每簇最大可达 64 KB。

NTFS 使用逻辑簇号(Logical Cluster Number,LCN)和虚拟簇号(Virtual Cluster Number,VCN)来定位簇。LCN 是对整个卷中的所有簇从头到尾进行编号,VCN 则是对特定文件的簇从头到尾进行编号。NTFS 支持文件的物理结构是索引文件,它通过 LCN 引用文件在磁盘上的物理位置,通过 VCN 引用文件中的数据,而 VCN 和 LCN 之间的映射通过索引表来实现。

主控文件表(Master File Table,MFT)是 NTFS 卷的管理控制中心,它包含系统引导程序,用于定位和恢复卷中所有文件的数据结构,记录整个卷分配状态的位示图等信息,这些信息称为“元数据”。MFT 由若干条记录构成,记录的大小固定为 1 KB,每条记录描述一个文件或目录。前 16 条记录保留用于存储“元数据”文件,并且有以符号“\$”开头的文件名,但此符号是隐藏的。“元数据”文件之后是一般文件和目录记录,MFT 的结构如图 6.19 所示。

通常情况下,每个 MFT 记录与不同的文件相对应。然而,如果一个文件有很多属性或分散成很多碎片,就可能占用多个 MFT 文件记录。在此情况下,用于存放同一文件属性的第一条记录就称为文件基记录,其他记录称为文件扩展记录。

2. MFT 的记录结构

MFT 的文件记录由记录头和紧跟其后的一系列(属性,属性值)对组成。记录头包含一个用于有效性检查的魔数、文件生成时的顺序号、文件的引用计数、记录中实际使用的字节数。对于扩展记录,还有文件基记录的索引、顺序号的标识符以及其他字段。记录头之后依次是第一个属性及其属性值,第二个属性及其属性值,等等。其中,(属性,属性值)对是指属性的名字和属性的具体内容。NTFS 通过在大写字母前加符号“\$”来指定属性,如 \$FILE_NAME 和 \$DATA 分别是文件名属性和文件内容属性,其所对应的属性值就是具体的文件名和文件内容。NTFS 不是简单地将文件视为一系列字节的集合,而是将其看成由许多(属性,属性值)集合来进行存储和处理的。文件属性分为有名属性和无名属性,有名属性包括文件名、文件拥有者、时间标记和安

全描述符等,而文件内容则是文件的无名属性。文件的每个属性通过单独的字节流进行存取。严格地说,NTFS 并不对文件进行操作,而只是对属性流进行读写,NTFS 提供对属性流的各种操作,包括:创建、删除、读取及写入。这样,NTFS 只负责读写有名字的属性流,应用程序才读写无名属性,即实际文件的数据。

记录号	文件名	含义
0	\$Mft	记录卷中所有文件的所有属性
1	\$MftMirr	MFT 表前 9 行的副本
2	\$LogFile	日志文件,记录影响卷结构的操作,用于系统恢复
3	\$Volume	卷文件,包含卷名、卷的 NTFS 版本等信息
4	\$AttrDef	属性定义表,定义卷所支持的属性类型,如可恢复
5	\$/	根目录,存放根目录下的文件和目录信息
6	\$Bitmap	盘空间位示图,记录簇的使用情况,每位对应于一簇
7	\$Boot	Win 引导程序
8	\$BadClus	坏簇文件,用于记录磁盘坏道
9	\$Secure	安全文件,存储卷的安全性描述数据库
10	\$UpCase	大写文件,包含大小写字符转换表
11	\$Ext. metadata Directory	扩展元数据目录,包含多种信息,如磁盘配额等
12~15		备用
>15		普通文件和目录

图 6.19 MFT 中 NTFS 元数据文件记录

MFT 中的属性定义文件,定义 NTFS 卷上文件的常用属性,如:文件基本属性(文件拥有者及连接数)、可变长度文件名、安全描述符、文件内容、索引根及索引分配、位示图、属性列表、文件标识、符号连接、EFS 属性、卷名及卷的版本号等。

3. 文件和目录的 MFT 记录

根据文件的大小,文件属性分为常驻属性和非常驻属性。当一个文件很小时,其所有属性和属性值可存放在 MFT 文件的一条记录中,此属性称为常驻属性,文件的每个属性以标准头开始,且标准头总是常驻的,标准头中包含相应的属性及其属性值是常驻还是非常驻等信息。若常驻,则给出从头到属性值的偏移、属性值的长度;若非常驻,它的头包含查找属性值所需要的信息。文件的某些属性总是常驻的,如标准信息属性和根索引,NTFS 通过这些信息确定文件的其他非常驻属性,文件的数据属性可能有些是非常驻的。如果属性值直接存放在 MFT 中,那么 NTFS 只需访问一次磁盘即可获得数据。

小文件或小目录的所有属性常驻在 MFT 中, 小文件的无名属性可以包括文件的所有数据, 小目录的索引根属性可以包括其中所有文件和子目录的索引。图 6.20 给出小文件和小目录的 MFT 记录。

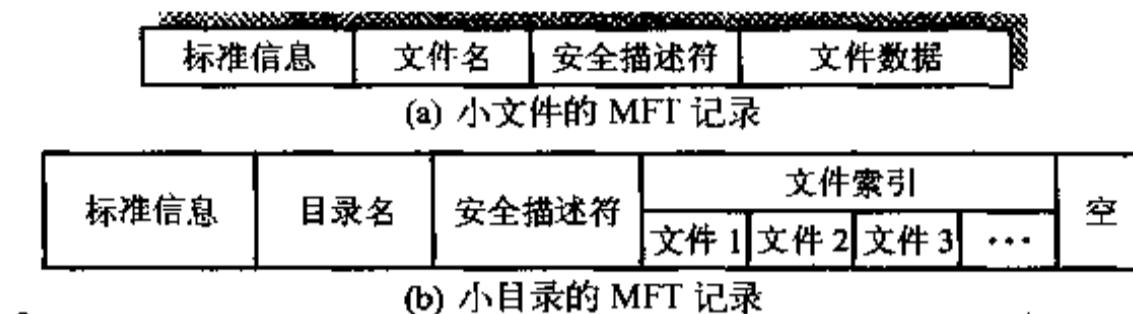


图 6.20 小文件和小目录的 MFT 记录

大文件或大目录的所有属性不可能全部常驻在 MFT 中, 如果一个属性(如文件内容属性)太大而不能存放在只有 1 KB 的 MFT 文件记录中, 那么将为其分配一个与 MFT 分开的区域, 此区域称为一个扩展(extent), 用来存储属性值, 如文件数据。值存储在扩展中而非 MFT 文件记录中的属性称为非常驻属性, 如果以后此属性值增加, 那么会再分配一个扩展。对于非常驻属性, 其中“标准信息”和“文件名”属性总是常驻的, 且其他属性的头和至少部分属性索引根的属性值是常驻的。

对于大文件的数据属性, 其头包含 NTFS 在磁盘上查找属性值所必需的信息, 通过建立虚拟簇号和逻辑簇号之间的映射, 记录扩展盘区的占用情况, 以定位文件。如图 6.21 所示是一个非常驻数据属性存储在两个扩展中的 VCN 与 LCN 之间的映射关系。

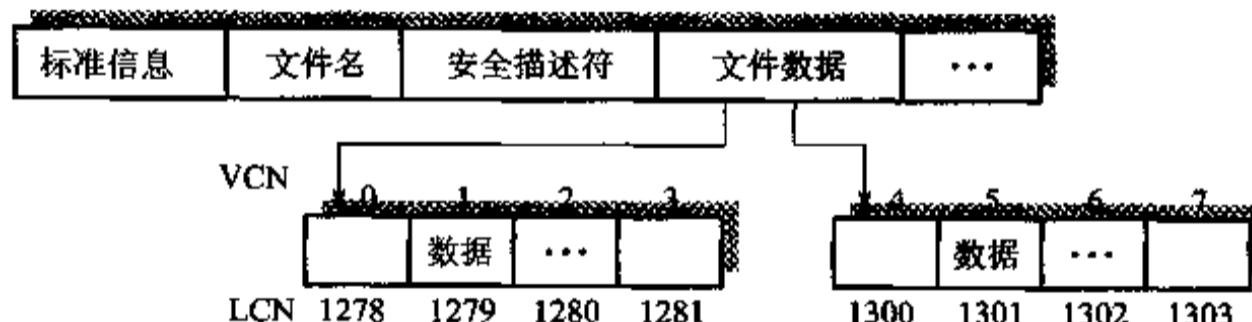


图 6.21 非常驻数据属性的 VCN 与 LCN 映射

当一个文件具有超过两个扩展时, 第三个扩展从 VCN 8 开始, 数据属性头部含有前两个扩展的 VCN 映射, 为了便于 NTFS 快速查找, 在具有多个扩展的文件的常驻数据属性头中包含 VCN 与 LCN 的映射关系。

如果一个文件有太多的属性而不能存放在 MFT 记录中, 那么第二个 MFT 文件记录就可用来容纳这些额外的属性(或非常驻属性的头), 此时引入一个称为“属性列表”的属性。属性列表包括文件属性的名称和类型代码, 及属性所在 MFT 的文件引用计数。属性列表通常用于太大或太零散的文件, 这种文件因 VCN 与 LCN 映射关系太多而需要多个 MFT 文件记录。

4. NTFS 文件的索引

在 NTFS 系统中,文件的物理结构是索引式的,文件目录则是文件名的一个索引。当创建一个目录时,NTFS 必须对目录中的文件名和子目录名属性进行索引,并保存在索引根属性中,NTFS 根目录文件在 MFT 记录中的文件名索引如图 6.22 所示。

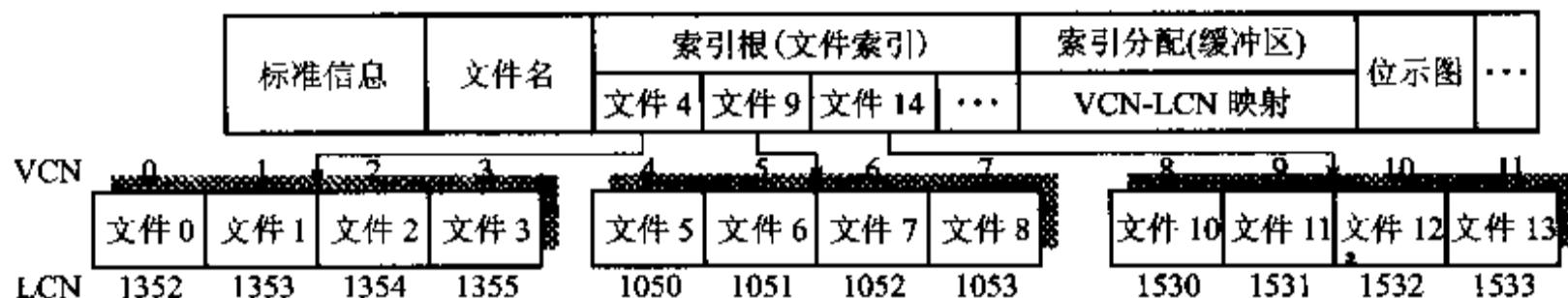


图 6.22 NTFS 根目录的文件名索引

对于一个大目录,文件名实际存储在固定 4 KB 的索引缓冲区(索引分配)中。索引缓冲区是采用 $B +$ 树数据结构实现的, $B +$ 树是平衡树的一种,它使得查找一个特定文件的访问磁盘次数降至最少,根索引属性包含 $B +$ 树的第一级(根目录)并指向包含下一级(子目录或文件)的索引缓冲区。

5. NTFS 文件的引用

NTFS 卷上的每个文件都有唯一的文件引用号,系统通过文件引用号引用文件。一个文件引用号由 64 位组成,分为文件号和文件顺序号两部分,文件号占低 48 位,文件顺序号占高 16 位,文件号指出文件在 MFT 中的索引位置,当一个文件在 MFT 中占有两条记录时,顺序号指出被引用文件在 MFT 中是第几条记录。

6.6.3 文件系统模型和 FSD 体系结构

在 Windows 中,I/O 管理器负责处理所有设备的 I/O 操作,文件系统的组成和结构模型如图 6.23 所示。

- (1) 设备驱动程序:位于 I/O 管理器的最底层,直接控制设备的 I/O 操作。
- (2) 中间驱动程序:与低层设备驱动程序一起提供增强容错功能,如发现 I/O 操作失败时,设备驱动程序只会简单地返回出错提示信息,而中间驱动程序却可以在收到此信息后,向设备驱动程序下达重执请求。
- (3) 文件系统驱动程序(File System Driver, FSD):扩展低层设备驱动程序的功能,以实现特定的文件系统(如 NTFS)。
- (4) 过滤驱动程序:可位于设备驱动程序与中间驱动程序之间,也可位于中间驱动程序与文件系统驱动程序之间,还可位于文件系统驱动程序与文件系统 API 之间。例如,一个网络重定向过滤驱动程序,用于截取对远程文件系统的各种操作,并重定向到远程文件服务器上。

与文件管理联系最密切的是 FSD,分为本地 FSD 和远程 FSD。前者允许用户访问本地计算机上的文件,后者则允许用户通过网络访问远程计算机上的文件。

1. 本地 FSD

本地 FSD 支持 NTFS 文件系统、FAT 文件系统、光盘文件系统和只读光盘文件系统等。本地 FSD 工作在核心态，系统启动时向 I/O 管理器注册，当开始访问某个卷时，I/O 管理器将调用 FSD 来进行卷识别。文件系统每个卷的第一个扇区都作为启动扇区预留，其上保存足够多的信息以供确定卷上文件系统的类型和定位元数据的位置等信息。另外，卷识别常常需要对文件系统作一致性检查。

本地 FSD 负责对本机上的文件系统进行管理，当系统初始化时，I/O 管理器通过卷参数块（Volume Parameter Block, VPB）在存储管理器中创建卷设备对象，与本地 FSD 所创建的设备对象之间建立连接，即 I/O 管理器提供 VPB 的连接，将有关卷的 I/O 请求转交给本地 FSD 的设备对象，本地 FSD 通过高速缓存管理器来缓存文件系统的数据以提高性能，它与主存管理器一起实现主存文件映射。

2. 远程 FSD

远程 FSD 由两部分组成：客户端 FSD 和服务器端 FSD。客户端 FSD 接收来自本地应用程序的 I/O 请求，通过过滤驱动程序转换为网络文件系统协议命令，再通过网络发送给服务器端 FSD。服务器端 FSD 监听网络命令，接收网络文件系统协议命令，并转交给其本地 FSD 执行。

3. FSD 的功能

Windows 文件系统的有关操作都是通过 FSD 来完成的。FSD 主要实现以下一些功能。

(1) 处理文件系统操作命令

应用程序通过 Win32 I/O 函数，如 CreateFile、ReadFile 和 WriteFile 等访问文件。当用户使用 fopen(文件名, 操作方式)打开一个文件时，这个请求传送给 Win32 客户端 Kernel32.dll，它进行参数合法性的检查之后，以函数 CreateFile()取代，继续执行并转换成系统调用 NtCreateFile()，开始在对象管理器中检查文件名字符串，对象管理器搜索对象名空间，把控制权提交 I/O 管理器的 FSD。FSD 询问安全子系统，以确定此文件存取控制表是否允许用户的访问方式。若允许，将把经核准的存取权和文件句柄一起返回用户，之后，用户使用文件句柄对文件进行存取。

用户通过 fread()或 fwrite()读写文件时，同样在进行合法性检查后，用函数 ReadFile()或 WriteFile()通过动态链接库转换成对 NtReadFile()或 NtWriteFile()的系统调用。NtReadFile()

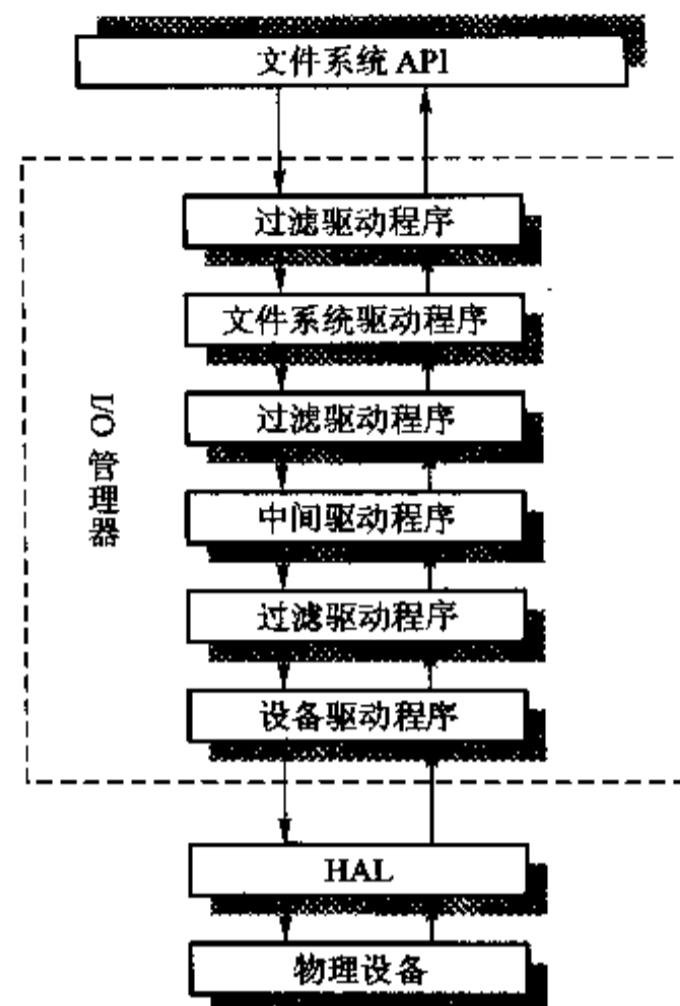


图 6.23 Windows 文件系统的组成和结构模型
图 6.23 展示了 Windows 文件系统的组成和结构模型。该模型从用户 API 层级（文件系统 API）开始，通过一系列驱动程序层（过滤驱动程序、文件系统驱动程序、中间驱动程序、过滤驱动程序、设备驱动程序）到达 HAL 层，最后连接到物理设备。I/O 管理器位于中间驱动程序和设备驱动程序之间，负责管理卷设备对象与本地 FSD 设备对象之间的连接。

将已打开文件的句柄转换成文件对象指针,检查访问权限,创建 I/O 请求包 IRP,处理读请求,且把 IRP 转交给合适的 FSD。之后检查文件是否放在高速缓存中,如果不在,则申请一个高速缓存映射结构,将指定文件块读入其中。最后,NtReadFile()从高速缓存中读取数据,送至用户指定区域,完成本次 I/O 操作。

(2) 高速缓存延迟写

高速缓存管理器的延迟写线程定期地异步调用主存管理器,把高速缓存中已被修改过的页面移交给 FSD,以便将数据写入磁盘。

(3) 高速缓存提前读

高速缓存管理器的提前读线程通过分析已执行的读操作,来决定提前读多少,再通过缺页中断将数据读至高速缓存。

(4) 主存脏页写

主存脏页写线程定期清理高速缓冲区,将不再使用的页面写入页文件或映射文件,使得主存管理器有空闲页框可用。此线程通过异步写命令来创建 I/O 请求包 IRP,由于 IRP 被标识为不能通过高速缓存,因此,被 FSD 直接送交磁盘驱动程序。

(5) 主存缺页处理

应用程序访问不在主存中的页面时,触发缺页中断,且向文件系统发送 I/O 请求包 IRP,完成缺页处理。

4. NTFS 的 FSD

对 NTFS 的访问通过 I/O 管理器来完成,I/O 管理器将 I/O 请求送交 NTFS 的 FSD 去执行。这一过程与高速缓存管理器(用于为 NTFS 提供高速缓存服务)、主存管理器(用于缺页时向文件系统发送请求调页)、日志文件服务器(用于记录影响 NTFS 卷结构的操作,系统失败时恢复 NTFS 格式化卷)、卷管理器、磁盘驱动程序等一起协同完成 I/O 操作。应用程序通过 NTFS 的 FSD 创建和存取文件的过程涉及以下步骤:

- (1) Windows 进行有关使用权限的检查,只有合法用户的请求才会被执行;
- (2) I/O 管理器将文件句柄转换为文件对象指针;
- (3) NTFS 通过文件对象指针获得磁盘上的文件。

NTFS 如何通过文件对象指针获得磁盘上的文件呢? 用户打开文件表和系统打开文件表在 Windows 中表现为每个进程都有进程对象表及其所指向的具体文件对象,当 I/O 系统调用 NTFS 时,此句柄已经被转换成指向文件对象的指针。NTFS 通过文件对象指针获得文件属性的流控制块(Stream Control Block,SCB),每个 SCB 表示文件的一个属性,包含如何获得属性的信息。同一个文件的所有 SCB 都指向一个共同的数据结构——文件控制块(File Control Block,FCB),FCB 包含指向主控文件表 MFT 中此文件记录的指针,实际上是一个文件引用,NTFS 通过这个指针访问文件,如图 6.24 所示。

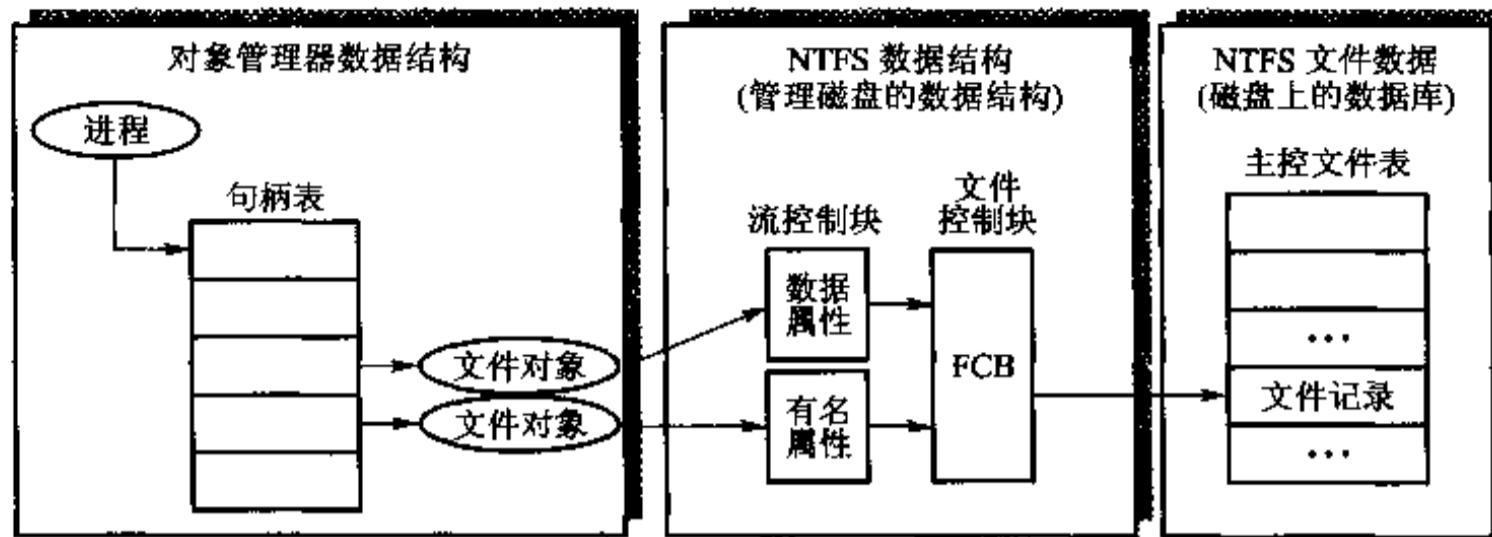


图 6.24 NTFS 数据结构和文件查找

6.6.4 NTFS 可恢复性支持

NTFS 的可恢复性支持确保系统在掉电或发生软件故障时,使磁盘卷保持完整的结构,确保文件系统的一致性。它采用基于事务的日志文件方法来实现可恢复性,但 NTFS 只能恢复文件系统的元数据,无法恢复用户数据。

NTFS 要求把改变卷结构的每个事务(如写或删除操作)的各个子操作在被写入磁盘之前,首先被记录在日志文件中,一旦出现故障,根据记录在日志中的文件操作信息,对那些部分完成的事务进行重做或撤销,保证磁盘文件的一致性,这种技术称为预写日志记录(write-ahead logging)。NTFS 用延迟写文件的优化技术提高系统速度,为了保证用户数据尽可能不被丢失,允许应用程序采用写直通和高速缓存的快速刷新能力,确保对文件的修改以适当时间间隔记录到磁盘上。NTFS 的可恢复性既保证卷结构不遭受破坏,又保证用户数据尽可能地少受损失。下面讨论日志文件服务的实现要点。

1. 日志文件服务

日志文件服务(Log File Service,LFS):是一组 NTFS 驱动程序内的核心态程序,NTFS 通过 LFS 例程来访问日志文件。每当系统启动时,NTFS 打开一个日志文件,并将此文件对象的指针传递给 LFS,以便 LFS 记录将要发生的事务,LFS 对日志文件进行初始化。其工作过程如下:

- (1) NTFS 执行所有修改卷结构的事务时,调用 LFS,LFS 调用高速缓存管理器,要求将此次写操作记录在高速缓存的日志文件中;
- (2) NTFS 在高速缓存中修改卷结构;
- (3) 高速缓存管理器调用 LFS,提示它将高速缓存中的日志文件刷新到磁盘。LFS 通过回调高速缓存管理器,要求其将应刷新的主存日志文件刷新到磁盘上;
- (4) 高速缓存管理器将修改卷结构的缓存内容写入磁盘。

LFS 分为两个区域:重启区和日志记录区。重启区保存用于失败后恢复系统所需要的信息,在 NTFS 进行重启恢复时,给出应当读取的日志记录区的起始地址。为了保证系统的可

恢复性,LFS 还紧随其后保存重启动区的一个副本。重启动区之后是日志记录区,用于记录 NTFS 写入的日志记录。

LFS 利用逻辑序号 LSN 标识写入日志文件中的记录,LFS 循环使用日志记录区,从而可保存无限多个日志记录。NTFS 不直接存取日志文件,而是通过 LFS 进行,LFS 提供打开、写入、向前、向后和更新等操作来处理日志文件。

2. 日志记录类型

LFS 允许 NTFS 向日志文件中写入任何类型的记录,其中,更新记录和检查点记录是 NTFS 所支持的两种日志记录,它们在系统恢复过程中起到重要的作用。

(1) 更新记录

所记录的是文件系统的更新信息,是 NTFS 写入日志文件中最普通的记录,每当发生下列事件:创建文件、删除文件、扩展文件、截断文件、设置文件信息、重命名文件、更改文件安全信息,NTFS 都会写入更新记录。它一般包含两种信息:

① 重做信息

当系统崩溃时,在事务从高速缓存刷新到磁盘之前,如果事务已经提交,即日志文件已经记录到磁盘上,则可通过日志记录重新执行事务,以恢复对卷的修改。

② 撤销信息

当系统崩溃时,若对卷修改的事务尚未提交,即日志文件未记录到磁盘上或记录不完整,则撤销这个对卷修改未提交的事务的所有子操作。

当一个事务的最后一个子操作被记录后,NTFS 就对高速缓存中的卷自身执行子操作。在完成高速缓存的卷更新以后,NTFS 就向日志文件写入事务的最终记录,即“提交一个事务”的子操作。在将整个事务完整地记录到磁盘后,完成此事务的提交过程。

当系统失败需要进行恢复时,NTFS 通过读取日志文件,重做每个所提交的事务。由于 NTFS 并不清楚已经提交的事务是否已从高速缓存中及时刷新到磁盘上,所以还要重做一次已提交事务的各个子操作。在文件系统恢复过程完成“重做”操作之后,NTFS 根据系统崩溃时未被提交事务的日志文件中的撤销信息来回退已经记录的每个子操作。

(2) 检查点记录

除了更新记录之外,NTFS 还周期性地向日志文件中写入检查点记录。在写入检查点记录后,再在重启动区存储此检查点记录的逻辑顺序号 LSN。当系统由于失败进入恢复过程时,NTFS 通过存储在检查点记录中的信息,定位日志文件中最近写入的检查点记录。

3. 系统的可恢复过程

NTFS 可恢复过程依赖于主存中所维护的两张表。

(1) 事务表:跟踪已经启动但尚未提交的事务,以便恢复过程中从磁盘删除这些活动事务的子操作;

(2) 脏页表:记录尚未写入磁盘的高速缓存中时,用于改变 NTFS 卷结构操作的页面,在恢复过程中,这些改动必须刷新到磁盘上。

NTFS 每隔 5 秒向日志文件写入一个检查点记录,在写入之前,调用 LFS 在日志文件中存储事务表和脏页表的当前副本。然后,NTFS 在检查点记录中记录包含已复制表的日志记录的 LSN。当进行系统恢复时,NTFS 调用 LFS 定位日志文件记录中最近的检查点记录及最近的事务表和脏页表的副本,将这些表复制到主存中。

在最近的检查点记录之后,日志文件通常包含更多的更新记录,这些更新记录显示最近一次检查点记录写入后,卷所发生的更改。为此,NTFS 必须更新事务表和脏页表,通过更新这些表和日志文件中的内容来更新卷本身。

为了实现卷的恢复,NTFS 要对日志文件进行以下 3 次扫描。

(1) 分析扫描

NTFS 从日志文件中最近一个检查点操作的起点开始,进行分析扫描。因为检查点操作起点之后的每条更新记录都代表对事务表或脏页表的修改,如一个“事务提交”的更新记录所代表的事务必须从事务表中删除,一个“页面更新”记录则表示修改了文件系统的一个数据结构,相应的脏页表也必须更新。这两个表被复制到主存后,NTFS 将搜索这两个表。事务表包含未提交事务的 LSN,脏页表包含高速缓存中尚未刷新到磁盘的记录的 LSN,NTFS 根据这两个表的信息,确定最早的更新记录的 LSN,决定重做扫描的开始点。

(2) 重做扫描

在重做扫描过程中,NTFS 将从分析扫描所得到的事务表和脏页表中最早记录的 LSN 开始,在日志文件中向前扫描,查找“页面更新”记录(这条记录包含系统失败前已经写入的卷更新,但是可能尚未刷新到磁盘上)。之后,NTFS 将边查找边在高速缓存中重做这些更新。当 NTFS 到达日志文件的末尾时,它已经利用必要的卷更改更新了高速缓存,之后,高速缓存管理器在后台向磁盘写入高速缓存中的内容,恢复文件系统。

(3) 撤销扫描

在 NTFS 完成重做扫描之后,它将开始撤销扫描,以撤销系统失败时未提交的事务。

6.6.5 NTFS 安全性支持

NTFS 为了保证文件系统的安全及可靠性,对卷上的每个文件和目录都设置权限,以确保文件被安全地使用。在创建文件时,由文件创建者指定自己及其他用户对文件的访问权限,记录在文件目录中,文件拥有者也可通过系统所提供的命令,随时修改文件的访问权限。可能的权限有:读、读和添加、更改权限、控制、列表、执行和拒绝访问。NTFS 制定以下权限设置规则:

- (1) 用户或其所在组必须按照指定权限对文件和目录进行访问;
- (2) 权限是累积的,如果组 A 用户对文件拥有“写”权限,组 B 用户对此文件只有“读”权限,

而用户 C 同属两个组，则 C 将获得“写”权限；

(3) “拒绝访问”权限的优先级高于其他权限，如果组 A 用户对文件拥有“写”权限，组 B 用户对此文件“拒绝访问”，那么同属两个组的用户 C 被拒绝访问；

(4) 文件权限始终优先于目录权限；

(5) 当用户在相应权限的目录中创建新文件或子目录时，所创建的文件或子目录继承此目录的权限。

本 章 小 结

计算机系统使用高速大容量磁盘存储器作为系统的辅助存储器，一种简单方便、高度抽象的磁盘使用方法是：认为它是一种容纳一组命名文件的设备，文件系统把磁盘的硬件特性和用户隔离开来，为用户提供“按名存取”的功能。简单地说，文件系统是操作系统中负责存取和管理信息的模块，采用统一的方式管理用户和系统信息的存储、检索、更新、共享和保护，并为用户提供一整套方便而有效的文件使用和操作方法。

文件是由文件名标识的一组信息的集合。一个文件必须从逻辑结构和物理结构两个方面来观察。文件的逻辑结构分为：流式文件和记录式文件。文件的物理结构分为：顺序文件、连接文件、索引文件、直接文件。

文件目录是实现按名存取的主要数据结构，文件系统的基本功能之一就是负责文件目录的建立、维护和检索，要求所编排的目录便于查找，防止冲突，目录的检索方便、迅速。实际的操作系统常使用树状目录结构，目录可按不同方法组织，有的把文件名、文件属性、磁盘地址存放在一起组成 FCB；有的仅存放文件名和 inode 号，其他信息则放到文件 inode 中。

文件共享是指不同用户（进程）共同使用同一个文件。文件共享有时不仅为不同用户完成共同任务所必需，而且还可以节省大量的辅存空间，减少由于文件复制而增加的访问辅存的次数。文件共享分为静态共享、动态共享和符号链接共享。

对于用户而言，文件的保护和保密是至关重要的问题。有关保护和保密技术将在第七章讨论。常用的文件操作有：建立文件、打开文件、读写文件、控制文件、关闭文件和删除文件。

有两种特殊类型的文件：主存映射文件和虚拟文件系统。前者是进程不使用文件类系统调用而通过直接读写主存来使用文件信息的一种技术；后者是在一个操作系统中同时支持多种具体的文件系统而采用的一种通用文件系统模型。

习 题 六

一、思考题

1. 试述下列术语的定义并说明它们之间的关系：卷、块、记录、文件。

2. 什么是记录的成组和分解操作？采用这种技术有什么优点？
3. 列举文件系统面向用户的主要功能。
4. 什么是文件的逻辑结构？它有哪几种组织方式？
5. 什么是文件的物理结构？它有哪几种组织方式？
6. 试述文件的各种物理组织方式的主要优、缺点。
7. 什么是记录键？有何用途？
8. 连接文件的连接字可如下定义：
 - (1) 连接字的内容为(上一块的地址)+(下一块的地址)。
 - (2) 首块连接字的内容为(下一块的地址)。
 - (3) 末块连接字的内容为(上一块的地址)。其中，+是模2按位加。试述这种连接字的主要特点。
9. 文件系统所提供的主要文件操作有哪些？试述各自的主要功能。
10. 试述Linux虚拟文件系统的设计思想和实现要点。
11. 试述Windows 2003 NTFS文件系统的主要特点和实现要点。
12. 试述Windows 2003 NTFS文件系统的可恢复性支持。
13. 试述Windows 2003 NTFS文件系统的安全性支持。
14. 解释：FCB、文件目录、文件目录项、目录文件。
15. 解释：用户打开文件表、系统打开文件表。
16. 解释：根目录、父目录、子目录、当前目录。
17. 解释：路径名、绝对路径名、相对路径名。
18. 什么是设备文件？如何实现设备文件？
19. 什么是文件的共享？介绍文件共享的分类和实现思想。
20. 什么是文件的安全控制？有哪些方法可实现文件的安全控制？
21. 为了快速访问，又易于更新，当数据为以下形式时，选用何种文件组织方式？
 - (1) 不经常更新，经常随机访问；
 - (2) 经常更新，按照一定的顺序访问；
 - (3) 经常更新，经常随机访问。
22. 一些系统允许用户同时访问文件的副本实现共享，另一些系统为每个用户提供一个共享文件副本，试讨论两种方式各自的优、缺点。
23. 目前采用广泛的是哪种文件目录结构？它有什么优点？
24. 试述hash文件的优点和缺点。
25. 试述hash文件解决冲突的方法。
26. 设计一种hash算法，用于快速检索文件控制块。
27. 试说明树状目录结构中线性检索法的检索过程。
28. 试说明采用二分法检索文件目录的检索过程。
29. 什么是“按名存取”？文件系统如何实现文件的按名存取？
30. 何时建立文件目录？它在文件管理中起到什么作用？

31. UNIX/Linux 把文件描述信息从文件目录项中分离出来,为什么?
32. UNIX/Linux 的 inode 是文件内容的一部分,这种说法对吗?请说明理由。
33. UNIX/Linux 进程 0 的主要任务是什么?
34. UNIX/Linux 采用空闲块成组连接的方式管理文件空间,试给出申请和归还块的工作流程。
35. 使用文件系统时,通常要显式地进行 OPEN、CLOSE 操作。
 - (1) 这样做的目的是什么?
 - (2) 系统提供显式的 OPEN、CLOSE 操作有什么优点?
 - (3) 系统不提供显式的 OPEN、CLOSE 操作,那么系统如何实现对文件信息的存取?
36. 在支持顺序文件的系统中常常有文件返绕操作,支持随机存取文件的系统是否也需要这个操作,为什么?
37. 文件系统提供系统调用或命令 rename 实现文件更名,同样也可以通过把文件复制到新文件并删去原文件来实现文件更名,试述两种方法的区别。
38. 对文件系统根目录的长度是否要施加限制,为什么?
39. 在 UNIX/Linux 系统中,用户 2 对用户 1 的一个文件建立了一个链接,当用户 1 删除此文件后,用户 2 访问这个文件,其结果如何?

二、应用题

1. 磁带卷上记录了若干文件,假定当前磁头停留在第 j 个文件的文件头标前,现要按名读取文件 i ,试给出其实现步骤。
2. 令 B =物理块长, R =逻辑记录长, F =块因子,对于定长记录(一个块中有整数个逻辑记录),给出计算 F 的公式。
3. 某操作系统的磁盘文件空间共有 500 块,若用字长为 32 位的位示图管理磁盘空间,试问:
 - (1) 位示图需要多少个字?
 - (2) 第 i 字第 j 位所对应的块号是多少?
 - (3) 给出申请/归还一块的工作流程。
4. 若两个用户共享一个文件系统,用户甲使用文件 A、B、C、D、E;用户乙要使用文件 A、D、E、F。已知用户甲的文件 A 与用户乙的文件 A 实际上不是同一个文件;用户甲、乙的文件 D 和 E 是同一个文件。试设计一种文件系统组织方案,使得用户甲、乙能共享此文件系统又不致造成混乱。
5. 在 UNIX 系统中,如果一个盘块的大小为 1 KB,每个盘块号占用 4 B,即每块可存放 256 个地址。请将下列文件的字节偏移量转换为物理地址:
 - (1) 9 999; (2) 18 000; (3) 420 000。
6. 在 UNIX/Linux 系统中,如果当前目录是 /usr/wang,那么,相对路径为 /ast/xxx 的文件的绝对路径名是什么?
7. 一个 UNIX 文件 F 的存取权限为: rwxr-x---,此文件的文件主 uid=12, gid=1, 另一个用户 uid=6, gid=1, 是否允许此用户执行文件 F?
8. 设某文件为连接文件,由 5 个逻辑记录组成,每个逻辑记录的大小与磁盘块大小相等,均为 512 B,并依次存放在 50、121、75、80、63 号磁盘块上。若要存取文件的第 1569 逻辑字节处的信息,问要访问哪一个磁盘块?

9. 设有一个 UNIX/Linux 文件,如果一个盘块的大小为 1 KB,每个盘块号占用 4 B,那么,若进程欲访问偏移量 263 168 B 处的数据,需要经过几次间接寻址?

10. 设某个文件系统的文件目录中,指示文件数据块的索引表长度为 13,其中 0~9 项为直接寻址方式,后 3 项为间接寻址方式。试描述文件数据块的索引方式,给出对文件第 n 个字节(设块长为 512 B)的寻址算法。

11. 设文件 ABCD 为定长记录的连续文件,共有 18 个逻辑记录。如果记录长为 512 B,物理块长为 1 024 B,采用成组方式存放,起始块号为 12,叙述第 15 号逻辑记录读入主存缓冲区的过程。

12. 若某操作系统仅支持单级目录,但允许此目录有任意多个文件,且文件名可为任意长,试问能否模拟一个层次式文件系统。如果能的话,如何模拟?

13. 文件系统的性能取决于高速缓存的命中率,从高速缓存读取数据需要 1 ms,从磁盘读取数据需要 40 ms。若命中率为 h ,给出读取数据所需平均时间的计算公式,并画出 h 从 0 到 1 变化时的函数曲线。

14. 某个磁盘组共有 10 个盘面,每个盘面有 100 个磁道,每个磁道有 16 个扇区。若以扇区为分配单位,试问:

(1) 用位示图管理磁盘空间,则位示图占用多少空间?

(2) 若空白文件目录的每个目录项占用 5 B,则空白文件目录何时大于位示图?

15. 某磁盘共有 100 个柱面,每个柱面有 8 个磁头,每个盘面分为 4 个扇区。若逻辑记录与扇区等长,柱面、磁道、扇区均从 0 开始编号。现用 16 位的 200 个字(0~199)组成位示图来管理盘空间。试问:

(1) 位示图第 15 个字的第 7 位为 0 而准备分配某一记录,此块的柱面号、磁道号、扇区号是多少?

(2) 现回收第 56 柱面第 6 磁道第 3 扇区,这时位示图的第几个字的第几位应清 0?

16. 如果一个索引节点为 128 B,磁盘块指针长 4 B,状态信息占用 68 B,而每块大小为 8 KB。试问索引节点中有多大空间留给磁盘块指针?使用直接、一次间接、二次间接和三次间接指针分别可表示多大的文件?

17. 在一个操作系统中,inode 节点中分别含有 10 个直接地址的索引和一、二、三级间接索引。若设每个盘块有 512 B 大小,每个盘块中可存放 128 个盘块地址,则一个 1 MB 的文件占用多少间接盘块?一个 25 MB 的文件占用多少间接盘块?

18. 设一个文件由 100 个物理块所组成,对于连续文件、连接文件和索引文件,分别计算执行下列操作时的启动磁盘 I/O 操作的次数(假如头指针和索引表均在主存中)。

(1) 把一块加在文件的开头;

(2) 把一块加在文件的中间(第 51 块);

(3) 把一块加在文件的末尾;

(4) 从文件的开头删去一块;

(5) 从文件的中间(第 51 块)删去一块;

(6) 从文件的末尾删去一块。

19. 一个文件系统基于索引节点的组织方式,假设物理块长为 512 B。

(1) 文件名目录的每个表项占用 16 B,文件目录从物理块 111 号开始存放;

(2) 索引节点占用 64 B,索引节点区从物理块 2 号开始存放;

(3) 假设索引节点编号是从 1 到某个最大值,现有文件 file 为顺序文件,file 位于文件名目录的第 34 个目录项中,它所对应的索引节点号为 64。

为打开文件 file 需要启动几次磁盘,每次所读的物理块号是什么?请说明原因。

20. 某文件系统采用索引文件结构,设文件索引表的每个表目占用 3 B,存放盘块的块号,磁盘块的大小为 512 B。此文件系统采用直接、二级和三级索引所能管理的最大磁盘空间是多少?

21. 一个 UNIX inode 中有 10 个地址用于数据块的访问,还有单间接、双间接、三间接的访问地址各一个。若每个盘块的大小为 1 KB,可以存放 256 个磁盘地址,那么一个文件最大是多少?

22. 一个树状结构的文件系统如下图所示(其中的框表示目录,圆圈表示文件)。

(1) 可否进行下列操作:

① 在目录 D 中建立一个文件,取名为 A。

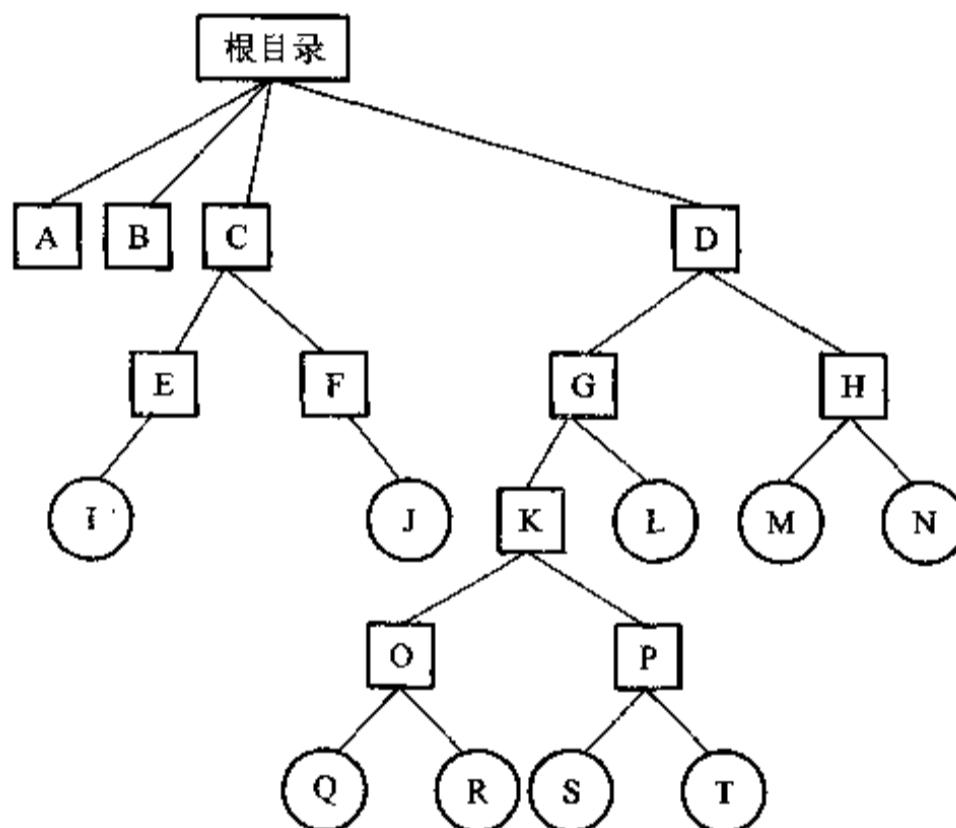
② 将目录 C 更名为 A

(2) 若 E 和 G 分别是两个用户的目录:

① E 的用户欲共享文件 Q,应有什么条件,如何操作?

② 在一段时间内,G 的用户主要使用文件 S 和 T。为了简化操作和提高速度,应如何处理?

③ E 的用户欲对文件 I 施加保护,不允许别人使用,能否实现? 如何实现?



23. 在 UNIX System V 中,如果一个盘块的大小为 1 KB,每个盘块号占用 4 B,那么,一个进程要访问偏移量 263 168 B 处的数据时,需要经过几次间接寻址?

24. 设某文件系统采用索引文件结构,假定文件目录项中有 10 个表目用于描述文件的物理结构(每个表目占用 2 B),磁盘块的大小与文件逻辑块大小相等,都是 512 B。经统计发现,此系统处理的文件具有如下特点:60%文件其大小 \leqslant 10 个逻辑块,30%文件其大小 \leqslant 2 000 个逻辑块,10%文件其大小 \leqslant 6 000 个逻辑块。设计此系统的索引结构,使得系统能够处理各类文件,并使读盘的次数尽可能少。

25. 假设在一个文件系统中,物理块的大小为 512 B;文件控制块(FCB)占用 48 B。如果把 FCB 分解成两个部分:符号目录项占 8 B(其中文件名占用 6 B,inode 号占用 2 B);基本目录项 inode 占用 $48 - 8 = 40$ B。试计算不分解 FCB 和分解 FCB 时,查找一个文件的平均访问磁盘次数。

26. 在文件系统中,为了加快文件目录的检索速度,可采用“FCB 分解法”。假设目录文件存放在磁盘上,每个盘块的大小为 512 B。FCB 占用 64 B,其中文件名占 8 B。通常将 FCB 分成两个部分,第 1 部分占 10 B(包括文

件名和文件内部号),第 2 部分占 56 B(包含文件内部号和文件的其他描述信息)。

(1) 若某个目录文件共有 254 个 FCB, 试分别给出采用分解法的前后, 查找此目录文件的 FCB 的平均访问磁盘次数。

(2) 若目录文件分解之前占用 n 个盘块, 分解后改用 m 个盘块存放文件名和文件内部号, 请给出使访问磁盘次数减少的条件。



第七章

操作系统的安全与保护

7.1 安全性概述

计算机安全所涉及的范围很广,至今尚无统一定义,其基本内容是对计算机系统的硬件、软件、数据加以保护,不会因偶然或恶意的原因而造成信息的破坏、更改和泄密,使计算机系统得以连续正常地运行。计算机安全既包括物理方面的,如对计算机环境、设施、设备、载体和人员,需要采取行政管理上的安全对策和措施,防止突发性或人为蓄意破坏;又包括逻辑方面的,针对计算机系统,特别是计算机软件及应用系统的安全和保护,严防信息被窃取和破坏。影响计算机系统安全性的因素有很多。首先,操作系统是一个并发系统,支持多用户共享一套计算机系统的资源,有资源共享就需要有资源保护,涉及各种安全性问题;其次,随着计算机网络的迅速发展,除了信息的存储和处理之外,还存在大量的数据传输操作,客户机要访问服务器,一台计算机要传输数据给另一台计算机,传输过程中对信息的安全性的威胁极大,于是就需要有网络安全,使得系统中信息的存取、处理和传输满足安全策略;另外,在信息系统中,越来越多的个人资料、商业机构和政府机关的机密信息被存放在数据库中,可被拥有它及依赖它的用户所使用,但不能被未授权者访问,这就提出了信息系统的基础——数据库安全性问题;最后,计算机安全中的一个特殊问题是计算机病毒,它会对系统资源和信息造成很大的危害和破坏,需要采取措施预防、发现和解除它。

计算机系统的安全性和可靠性是两个不同的概念。可靠性是指系统正常持续运行的程度,其目标是反故障;安全性是指不因人为疏漏或蓄谋作案而导致信息资源的泄漏、篡改和破坏,其目标是反泄密。可靠性是基础,安全性则更为复杂。鉴于计算机系统自身的脆弱性和安全模式的先天不足以及计算机犯罪现象的普遍存在,构造安全计算机信息系统绝非易事。一般来说,信息系统的安全模型涉及管理和实体安全性、网络通信安全性、软件系统安全性和数据库系统安全性。软件系统中最重要的是操作系统,由于它所处的特殊地位,计算机安全问题大都由操作系统来保证,所以,操作系统安全性是计算机系统安全性的基础。本章重点讨论操作系统安全性,主要内容包括:

(1) 安全策略:描述一组用于授权使用其计算机及信息资源的规则。

(2) 安全模型:精确描述系统的安全策略,它是对系统的安全需求及如何设计和实现安全控制的一个清晰而全面的理解和描述。

(3) 安全机制:实现安全策略所描述的安全问题,关注如何实现系统安全性,包括认证机制(authentication)、授权机制(authorization)、加密机制(encryption)、审计机制(audit)、最小特权机制(least privilege)等。

计算机系统资源分为硬件、软件、数据及网络和通信线路等,操作系统所面临的安全威胁也来自这些方面。

(1) 硬件

对硬件设备的威胁主要表现在可用性方面,最易受到攻击,也最不易得到自动控制,威胁包括对设备有意或无意的破坏及偷窃,需要物理上和行政管理上的安全措施来解除这些威胁。

(2) 软件

由于操作系统、编译程序及数据库管理系统等系统软件的功能多样、接口复杂、规模庞大,又缺乏安全理论的指导,其设计和实现过程中的疏漏在所难免,问题颇多。现代计算机系统都把信息存储在共享设备中,用户所拥有的信息有时允许共享,有时希望私有,导致操作系统必须提供保护和安全性功能;另一个更难处理的问题是对软件的非法修改导致程序仍能运行但其行为却发生变化,诸如特洛伊木马的各种病毒的攻击均可能对系统安全构成致命的威胁。

(3) 数据

在信息系统中,各种数据是共享资源,数据的安全性涉及可用性、机密性和完整性,主要涉及对数据有意和无意的窃取和破坏、对数据或数据库的未授权访问和非法修改等。

(4) 网络和通信线路

通信系统用来传送数据,数据从源端到目的端的流动可能受到切断、截取、篡改和伪造等威胁。服务的切断是对可用性构成的威胁;信息截取是对保密性构成的威胁;信息的篡改和伪造是对完整性构成的威胁。

开放TCP/IP协议族在规划之初未能对安全性给予足够的重视,网络的脆弱性、网络的配置及操作错误再加上主机系统的漏洞给无孔不入的攻击者以更多的可乘之机,造成网络病毒猖獗,黑客事件层出不穷。严峻的现实让人们认识到,计算机、网络和操作系统的发展离不开信息安全技术的有力保障,要不遗余力地提高计算机信息系统的安全性。

安全策略

7.2.1 安全需求和安全策略

1. 安全需求和策略

操作系统的安全需求是指设计一个安全操作系统时所期望得到的安全保障,通常要求系统

无错误配置、无漏洞、无后门、无特洛伊木马病毒等,能防止非法用户对计算机资源的非法存取。总结起来,用户对计算机操作系统(信息系统)的安全需求主要有以下4项。

(1) 机密性(confidentiality)需求:为秘密数据提供保护方法及保护等级,系统要防止信息被泄露给未授权用户,授权用户也仅能进行规定范围内的访问。

(2) 完整性(integrity)需求:系统中的数据和原始数据不发生变化,未遭到偶然或恶意的修改或破坏;系统要能防止未授权用户对信息进行修改、删除和破坏,通过对计算机信息系统的存储、传输和处理过程采取有效的措施,维护系统资源,使其保持有效的、预期的和一致的状态,尤其能防止在涉及审计的事件中发生“舞弊”。

(3) 可记账性(accountability)需求:又称审计,指要求能够证实用户身份,可对有关的安全活动进行完整记录、检查和审核,防止用户对访问过某信息或执行过某操作的否认。在实际系统中,用户的任何操作都能反映其主观意图,有必要记录用户执行过何种操作和进行过何种访问。这些信息都是分析系统受损害程度、恢复丢失或破坏的信息、评估系统安全性等的重要依据。

(4) 可用性(availability)需求:防止以非法方式独占资源,每当合法用户需要时保证其能够访问所需信息,为其提供所需服务;对于授权用户的任何正确输入,系统都会有对应的正确输出。

不同的行业或应用领域对4个安全需求的强调程度不一样。例如,军事安全策略注重信息的机密性需求,商业安全策略注重信息的完整性及可记账性需求,电信部门又侧重于信息的可用性需求。当然,某个应用领域并非强调某种需求而不需要其他需求。

安全策略是指用于授权使用其计算机及信息资源的规则,即有关管理、保护、分配和发布系统资源及敏感信息的规定和实施细则,一个系统可以有一个或多个安全策略,其目的是使安全需求得到保障。从安全的角度来看,说一个计算机系统是安全系统,是指此系统达到设计时所制定的安全策略要求,安全的计算机系统从设计开始就要考虑安全问题,安全策略是构建可信系统的坚实基础,而安全策略的制定取决于用户的安全需求。制定安全策略时着重考虑与安全需求相关的问题:对于机密性,应找出将信息泄露给未经授权者的所有漏洞,如权限泄露、信息的非法访问和传输,还要处理授权的动态改变问题;对于完整性,应当控制更改信息的授权途径,标识出能够更改信息的被授权的实体;对于可用性,必须描述提供何种服务,并保证服务达到预先设计的质量;对于可记账性,应对有关的安全活动进行完整记录、检查和审核。

基于计算机信息系统的应用场合,可将安全策略分成两类。

(1) 军事安全策略

其主要目的是提供机密性,同时还涉及完整性、可记账性和可用性。军事安全策略用于涉及国家、军事和社会安全部门等机密性要求很高的单位,一旦泄密将会带来灾难性的危害。

(2) 商业安全策略

其主要目的是提供完整性,也涉及机密性、可记账性和可用性。商业安全策略要保证商业公司的数据不被随意篡改。例如,银行的计算机系统如果受到完整性侵害,客户账目金额被改动,引起金融数据混乱的后果将十分严重。

对于计算机信息系统而言,制定安全策略时涉及以下问题:允许谁进入系统? 进入系统的访

问者以何种方式访问哪些信息？依据何种特征制定访问决策？系统是允许最大化共享还是实现最小特权？对安全属性的操作是集中管理还是分散管理？根据这些问题可以把安全策略归入两类：访问支持策略和访问控制策略，前者用来支持和保障访问控制策略的正确实施，反映可记账性和可用性要求；后者确定相应的授权和访问规则来控制对系统资源的访问，反映机密性和完整性需求。

2. 可信计算基

操作系统的安全依赖于具体实施安全策略的一切基础，为了讨论起来方便，先介绍一个基本概念。计算机系统内安全保护装置的总体包括硬件、固件、可信软件和负责执行安全策略的管理员在内的组合体，称为可信计算基（Trusted Computing Base, TCB），它建立一个基本的保护环境并提供可信计算机系统所要求的附加用户服务。具体来说，TCB 的组成部分有：操作系统的安全内核，具有特权的程序和命令，处理敏感信息的程序，实施安全策略的文件，相关的固件、硬件和设备，机器诊断程序，安全管理员，等等。

TCB 能完成的任务有：内核的安全运行、标识系统中的用户、保持用户到 TCB 登录的可信路径、实施主体对客体的访问控制、维护 TCB 功能的正确性和监视及记录系统中所发生的事件。TCB 的软件部分是安全操作系统的核心，其中最重要的组成部分是“引用监视程序”，它全权负责针对涉及安全性的操作（如执行写系统调用）的安全性检查，确定此操作能否执行。如图 7.1 所示是引用监视程序的工作原理。

通用操作系统的 TCB 可包括多个安全功能模块（TCB Security Function, TSF），每个 TSF 实现一个安全功能策略（TCB Security Policy, TSP），TSP 的集合构成安全域，以防止不可信主体的干扰和篡改。实现 TSF 有两种方法，一是设置前端过滤器，防止入侵者非法进入系统；二是设置访问监督器，它能防止越权访问，两者都是在硬件的基础上通过软件实现的安全策略，能够提供所要求的附加服务。在单机环境下，根据安全操作系统设计方法的不同，TCB 可以是一个安全内核或是一个前端过滤器，当然也可以是操作系统的关键部分，甚至包括全部操作系统；在网络环境下，一个 TSF 可能跨网络实现，各节点上的 TSF 协同工作，形成物理上分散但逻辑上统一的分布式安全系统。

7.2.2 访问支持策略

这类安全策略把系统中的用户与访问控制策略中的“主体”联系起来，用户欲进入系统必须要经过“身份认证”才能访问被授权的资源。访问支持策略为有效地实施访问控制策略筑起首道屏障，提供保障支持。这类策略有很多种，对其分别介绍如下。

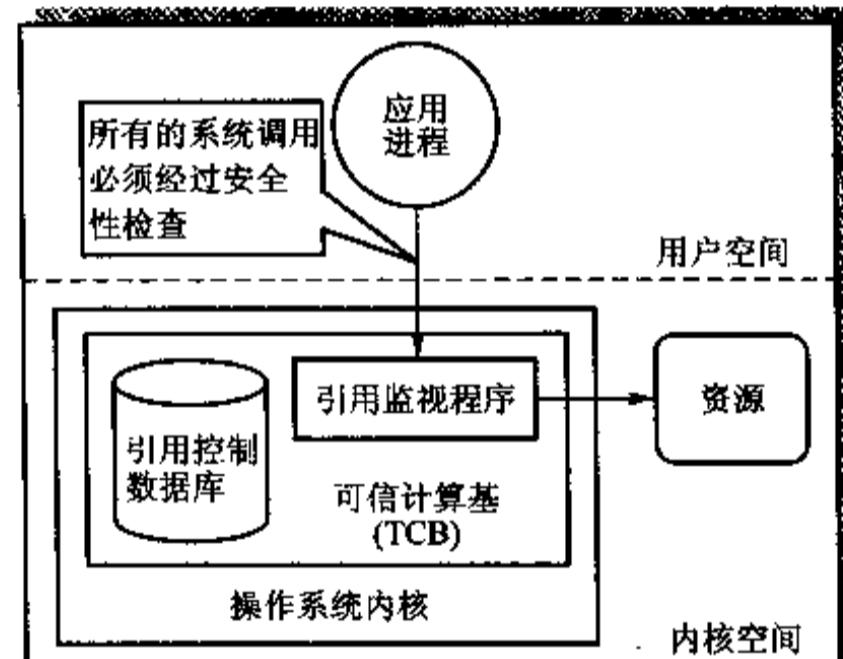


图 7.1 引用监视程序的工作原理

1. 标识与鉴别

(1) 用户标识(identification):这是操作系统用来标明用户身份的内部名称,确保用户的唯一性和可辨认性,一般选择用户名和用户标识符(用户 ID)来标明用户,用户名和标识符均是公开的明码信息。用户标识是有效实施其他安全策略(如用户数据保护和安全审计)的基础,通过为用户提供标识,TCB 能使用户对自己做出的行为负责。

(2) 用户鉴别(authentication):将用户标识与用户相联系的过程称为鉴别。鉴别的目的是对用户的真实身份进行确认,鉴别过程中通常要求用户具有能够证明其合法身份的特殊信息,这些信息是非公开的和难以仿造的,如口令、指纹和虹膜。用户鉴别是有效实施其他安全策略的基础。

标识和鉴别用于保证只有合法用户才能进入系统和访问相关资源。当用户登录计算机系统时,要向机器证明自己的身份,要求以特定的身份信息来标识自己,并且此身份必须经过系统的认证和鉴别,这一过程通常在用户登录时完成,系统会在整个生命周期为用户保留认证和鉴别信息,并以此来确认用户身份的唯一性和真实性。三类信息可用于用户标识和鉴别:用户所知道的信息、用户所拥有的物品和用户的生物特征。利用其中任何一类都可进行用户身份认证,但若能利用多类信息,或同时利用三类不同信息,会增强认证机制的有效性和稳健性。

基于第一类信息的常见实现形式是口令机制,用户提供用户名和口令以实现登录。对于口令的选择、分发和管理,有许多工作要做:管理员负责口令的分发和变更;用户经常改变并记住口令。这一身份认证机制的有效性建立在用户口令的安全性基础上,因此,在选择口令时,一定要排除弱口令,如用户名、出生年月、电话号码、车牌号码、身份证号码、短字符串等。基于第二类信息的身份认证要使用用户所拥有的物品作为身份认证的依据,如用户的财产——智能卡或钥匙,住户进入小区住宅楼需要刷卡或开锁,持有磁卡或钥匙的人可以进入。基于第三类信息的身份认证依赖于用户的生物特征,如签名(人的书写速度、运笔趋势和给予纸的压力不同)、指纹(人的指纹都存在差异)、语音(人的声音频率和振幅都存在差异)和虹膜(人的虹膜都存在差异)等,通过专门设备把生物特征转换为认证信息,这类认证技术的实现很复杂,且价格昂贵。安全系统必须保护用户标识和鉴别信息,防止未授权用户非法访问,以加密方式存放这些数据就是一种办法。

2. 可记账性

要求任何影响系统安全性的行为都被跟踪和记录,安全系统具有把用户标识与其被跟踪和记录的行为联系起来的能力。必须有选择性地保留和保护审计信息,所有与安全相关的事件均记录在审计日志文件中,审计数据必须防止受到未授权用户的访问、修改和破坏,以作为日后对备案事件开展调查的依据。

审计系统所记录的事件通常有:同标识和鉴别相关的事件,将客体导入用户地址空间的操作,客体的删除,系统管理员所执行的操作及其他与安全相关的事件。具体来说,审计记录项至少应包括事件发生的日期和时间、用户标识、事件类型、事件结果(成功/失败)。对于高级别的安全系统,要求审计系统记录隐蔽信息通道、用户认证时记录请求源(终端标识)、删除操作时记录客体名称及安全级别等附加信息。

3. 确切保证和连续保护

确切保证是指系统事先制定的安全策略能得到正确执行，并且安全系统能正确、可靠地实施安全策略的意图，为此，把握安全系统设计、开发、安装和维护的各个环节，基于硬件、固件、软件来保证系统信息的安全，防止可能造成的保护机制失效或来自旁路的未授权修改。此外，要创建系统在操作过程中安全策略不被歪曲的功能特征和系统架构。确切保证还包括系统功能的确切实现，如抗篡改、可验证、抗欺骗、抗旁路，这些功能通常要通过理论分析、系统测试、最终验证等多种环节方能达到。

连续保护策略要求安全系统必须连续不断地保护系统免遭篡改和非授权修改，如果用来实现安全策略的基础硬件、固件、软件自身易受篡改和破坏，那么没有任何一种计算机系统可称得上是安全的，也就不可能实现连续保护。

4. 客体重用

高安全级别的系统中引入客体重用的概念，它是指在对曾经包含一个或多个客体的存储介质，如页框、磁带、磁盘块、软盘、可擦光盘等，进行重新分配和重用时，为了安全实施重分配或重用，存储介质不得包含重分配之前的残留数据。为了达到安全再分配的目的，在 TCB 安全控制范围内的存储介质作为系统资源被动态地分配给新主体时，必须确保其中不能包含任何客体残留信息，包括经加密的信息，以防止可能造成的泄密。

5. 隐蔽信道分析

隐蔽信道是指可被进程用来以违反系统安全策略的方式进行信息非法传输的通信通道，有两类隐蔽信道：存储隐蔽信道和时间隐蔽信道。前者包括所有允许一个进程直接或间接地写一个存储客体，而允许另一个进程对此客体进行直接或间接的读访问的传输手段；后者包括所有进程通过调节其对系统资源的使用而向另一个进程发送信号，此进程通过观察响应时间的改变而进行信息传输的手段。从安全的角度来看，低带宽的隐蔽信道比高带宽的隐蔽信道有更小的威胁，带宽越大，潜在的安全威胁也就越大，但如果带宽降到一定的程度，系统性能也会随之降低，高安全系统中不应有高带宽隐蔽信道存在。

隐蔽信道分析适用于采用信息流控制安全策略设计的系统，系统开发者应根据要求搜索和标识隐蔽信道，并根据实际测量或工程估算确定每一个被标识信道的带宽。可采用三级的隐蔽信道分析：一般性隐蔽信道分析、系统化隐蔽信道分析和彻底隐蔽信道分析。所需要做的分析工作大致有：标识隐蔽信道及估算容量；确定隐蔽信道存在的程序；描述在最坏情况下对隐蔽信道容量进行估计的方法；描述每个可标识的隐蔽信道的最坏利用情形。

6. 可信路径和可信恢复

可信路径是一种实现用户与 TCB 之间的直接交互作用的机制，当连接用户（如登录、更改主体安全级）时，TCB 应提供其与用户之间的可信通信路径，此路径上的通信只能由用户和 TCB 激活，不能由其他软件（恶意软件）模仿，且在逻辑上与其他路径上的通信相隔离，并能正确地加以区分。此外，可信路径机制应具有可扩展能力，支持系统安全策略管理员根据需要添加可信应用程序。更为一般化的概念是保护路径，它是为了保证关键系统功能不被假冒而提出的，是确保

两个对象之间的相互认证渠道的一种机制。这种机制也可以在应用空间中建立,而无须系统认证支持,但是操作系统最好提供自己的保护路径机制,这样不但能够得到安全保证,而且可能更加有效。

可信恢复是指 TCB 由于某种原因发生中断时,能够在不降低安全性的前提下恢复运行。

7.2.3 访问控制策略

1. 访问控制属性

在计算机信息系统中,与访问控制策略相关的因素有三类:主体、客体和主客体属性。

(1) 主体

主体是主动实体,是系统行为的发起者。主体通常是用户和代表用户的进程,如运行进程正在访问文件,此时进程是存取文件的主体,而文件是客体。一般而言,系统中的所有事件请求几乎都是由主体激发的,系统合法用户的分类如下。

① 普通用户(进程)

普通用户是指获得授权可访问系统资源的客户,相应的授权包括对信息的读、写、删除、追加和执行等。

② 信息属主(进程)

信息属主对此信息拥有完全的处理权限,包括对信息的读、写、修改、删除和执行,撤销另一用户对信息的访问权限,向其他用户授予对信息所拥有的某些权限,除非此信息被系统额外追加控制。

③ 系统管理员(进程)

系统管理员对系统进行控制和管理,维护系统正常运转,如 UNIX/Linux 系统中,root 用户是系统管理员,其他的管理员主体还有安全员、审计员和操作员等。

(2) 客体

客体是被动实体,是系统内所有主体行为的直接承担者,客体常被分成:一般客体,指系统内以具体形式存在的信息实体,如文件、目录、数据和程序等;设备客体,指系统内的硬件设备,如磁盘、磁带、显示器、打印机、网络节点等;特殊客体,有时某些进程是另外一些进程的行为的承担者,那么,这类进程也是客体的一部分。

(3) 主客体属性

主客体属性又称敏感标记(sensitivity label)或标记,是 TCB 维护与可被外部主体直接或间接访问到的计算机信息系统资源相关的安全标记,是实施自主或强制访问的基础。通过安全标记把主体、客体的安全属性联系起来,并在系统运行全过程中发挥作用,信息系统的安全决策就是通过比较系统内主、客体的相关属性来制定的。为了输入未加设安全标记的数据,TCB 授权用户要求并接收这些数据的安全级别,并由 TCB 进行审计。

下面通过几类属性对访问控制策略的基础进行说明。

① 主体属性

主体属性是用户特征,是系统用来决定访问控制的常用因素,用户的任何属性都可作为访问

控制决策点。系统访问控制策略中常用的主体属性有：

- (a) 用户 ID/用户组 ID: 把系统中的用户与唯一 ID 对应起来, 进行访问控制时, 基于用户 ID 来判断他是否有权访问此信息, 自主访问控制策略便基于此属性;
- (b) 用户访问许可级别: 根据用户许可级别和系统内信息的安全级别来保护敏感信息, 强制访问控制策略便基于此属性;
- (c) 用户需知属性: 表明用户需要知道哪一类特定信息, 例如, 某单位的员工工资只有财务部人员和领导有“需知”属性的必要;
- (d) 角色: 在信息系统中, 可设置若干“角色”来表达主体在系统中的地位, 如 root 充当系统管理员的角色, 其他用户充当普通用户的角色。用户(组)和角色之间的不同之处在于: 前者是用户集合, 不涉及授权许可, 后者既是用户集合, 又是授权许可集合;
- (e) 权能列表: 这是与主体相联系的访问权力表, 用来表明主体能够对系统内的哪些客体进行何种访问。

② 客体属性

除了主体属性之外, 与系统内的客体相关联的属性也可作为访问控制策略的一部分。客体安全属性有:

- (a) 敏感性标记: 信息按照“安全等级”进行分类; 还可将系统内的信息按非等级方式分类, 进行模拟人力资源系统的划分, 它们是实施强制访问控制的依据, TCB 应该支持两种或多种成分所组成的安全属性;
- (b) 访问控制列表: 与客体相关联的有“访问控制列表”, 用来指定系统中的哪些用户(组)可用何种模式来访问此客体。客体属主可对此列表进行管理, 按意愿指定谁以何种模式访问此客体, 基于此访问控制列表的访问控制策略称为自主访问控制策略。

③ 外部状态

某些策略是基于主客体属性之外的外部状态来制定的。

- (a) 地点: 某些访问策略基于地点来制定, 如只有校长办公室的工作人员才准许看某文件;
- (b) 时间: 对系统内信息的访问可能会随着时间的流逝而变化, 如上午 10:00 宣布聘任为教授的人员的名单, 10:00 之前是敏感信息, 而其后是公开信息;
- (c) 状态: 有时状态也可用做对信息的访问策略, 如向某负责人打探信息, 他心情好时你就知道真实情况, 否则得不到所需消息。

④ 数据内容和上下文环境

有些访问控制策略可能基于数据值, 如不允许公司财务人员看到月薪超过 1 万元的技术员的财务记录。更为复杂的访问控制策略可能基于信息的上下文, 这种动态访问控制策略常用于数据库中。

(4) 用户与主体绑定

进程是操作系统中最为活跃的实体, 系统内的所有事件请求都要通过进程的运行来处理。应用进程是固定为某特定用户服务的, 在运行过程中代表此用户对客体资源进行访问, 其权限应

与所代表的用户相同,这一点可通过用户与主体的绑定来实现。

系统进程是动态地为所有用户提供服务的,当应用进程执行正常的用户态应用程序时,它所拥有的权限是其代表的用户的权限;然而,当应用进程执行系统调用时,开始执行内核函数,这时系统进程代表此用户在执行,运行于核心态,拥有操作系统权限。

2. 自主访问控制策略

自主访问控制策略根据系统中信息属主的指定方式或默认方式,即按照用户的意愿,来确定自己所拥有的资源允许系统中的哪些用户以何种权限共享,在这一点上对信息属主来说是“自主的”,它能提供精细的访问控制策略,将访问控制粒度细化到单个用户(进程)。按照系统访问控制策略实现的访问控制机制能够为每个命名客体指定命名用户(组),并规定其对客体的访问权限。未拥有访问权限的用户只允许由授权用户指定其对客体的访问权。

常用的自主访问控制工具都是访问控制矩阵的某种变形,如权能表和访问控制表。例如,基于系统内用户 ID 加上访问授权(权能表)或客体访问属性(访问控制表)来决定此用户是否有相应的权限访问这个客体。当然,也可基于信息内容或用户角色来实现对客体的访问控制。大多数操作系统提供基于掩码的保护方式,如 owner、group 和 other,但其保护粒度粗,不如上述几种方法精细。实际上,自主访问控制策略可用三元组(S,O,A)表示,其中,S 是主体,O 是客体,A 表示访问模式,自主访问控制中模式 A 的设定是非常灵活的。

自主访问控制策略的缺点是:安全性差,不能防范“特洛伊木马”病毒或“恶意程序”。所以,在系统进行自主访问控制的同时,利用强制访问控制策略来加强系统的安全性是很有必要的。

3. 强制访问控制策略

在强制访问控制机制下,系统内的每个主体被赋予许可标记或访问标记,以表示其对敏感性客体的访问许可级别。同样地,系统内的每个客体被赋予敏感性标记,以反映此客体的安全级别。安全系统通过比较主、客体的相应标记来决定是否授予某个主体对客体的访问权限。有些操作系统中的主体只有一个访问标记,但在另外一些操作系统中,一个主体可能有多个访问标记。

强制访问控制策略既可防止对信息的非授权篡改(保证完整性),又可防止未授权的信息泄露(保证机密性),安全标记可用不同的形式来实现信息的完整性和机密性。在系统中,无论何时何地,主、客体的标记不会被改变,表现出显著的“全局性”和“永久性”特征。如果一个安全系统中存在两套独立的强制访问控制策略,主、客体必然有两套不同的访问标记,则这两套标记是无关的。

7.3 安全模型

7.3.1 安全模型概述

安全模型是对安全策略所表达的安全需求的一种精确、无歧义的抽象描述,在安全策略与安全机制的关联之间提供一种框架。安全模型本身描述安全策略需要用哪种机制满足以及如何将特定的机制应用于系统中,从而实现某种安全策略所需要的安全与保护。

安全模型可分为形式化和非形式化两种。非形式化安全模型仅模拟系统的安全功能,其开发过程为:从安全需求出发,推出功能规范,再实现安全系统,其间主要采用论证和测试技术;形式化安全模型使用数学模型精确地描述安全性及其在系统中的使用情况,其开发途径为:建立抽象模型,推导形式化规范,通过证明方法来实现安全系统。

在形式化开发途径中,开发安全系统必须要建立安全模型,通过安全模型来模拟安全系统,从而可正确地综合系统的各类因素,如使用方式、应用环境类型、授权定义、共享客体(资源)、共享类型等,所有这些因素构成安全系统的形式化抽象描述,使得系统可被证明是完整的、反映真实环境的、逻辑上能实现程序受控制的执行。迄今为止,实际开发形式化安全模型仍然十分困难,指导实现安全系统的一般步骤为:确定外部要求,明确系统的安全需求,提出可信对象条件,描述安全需求机制,构造外部模型;确定内部要求,对系统的控制对象进行限制,以形成模型的安全性定义,把安全需求与系统抽象相结合,提出合理的模型变量,构造内部模型;为策略的执行设计操作规则,即把安全策略规则化,确保系统在有效完成任务的同时,始终处于安全状态;选择形式化规范语言,开发形式化验证工具,优化设计开发过程;一致性和正确性的验证,如安全需求表达是否准确、全面、合理,安全操作规则是否与安全需求协调一致,模型形式化与模型对应性论证等;模型实现阶段,分层次论述关联性,包括:实现模式、实现架构、模型在架构内的解释、所实现的对应性论证。

在当前技术条件下,安全模型都采用状态机模型,此模型将系统描述成抽象的数学状态机,状态变量表示机器的状态,转移函数或操作规则描述状态变量的变化过程,通过对影响系统安全的各种变量和规则进行描述和限制来达到确保系统安全状态不变量的维持的目的。由于描述操作系统的所有可能状态是不现实的,甚至是不可能的,但安全模型仅涉及与安全有关的状态变量,所以,状态机模型在安全模型中得到较为广泛的应用,可以较好地模拟和处理与安全相关的各种变量与函数。开发状态机模型通常需要下述步骤。

(1) 定义与安全有关的状态变量。典型的状态变量是系统的主体和客体、其安全属性以及主体对客体的访问权限。

(2) 定义安全状态所需满足的条件。这是一个不变式,它表达在状态转移期间,状态变量值必须保持何种关系。

(3) 定义状态转移函数。状态转移函数也称为操作规则,它描述状态变量可能发生的变化,其目的是限制系统可能产生的状态类型,而不是列举所有可能的状态变化,系统不能以函数所不允许的方式修改状态变量。

(4) 证明转移函数能够维持安全状态。为了确定模型与安全状态之间的一致性关系,必须证明:对于每个转移函数,如果系统在执行某种操作之前处于安全状态,那么,操作完成后,系统仍然处于安全状态。

(5) 定义初始状态。为每个状态变量选择初始值,这些值模拟系统在最初的安全状态中如何启动。

(6) 依据安全状态的定义,证明初始状态是安全的。

7.3.2 安全模型示例

对状态机模型进行改进,一类是把系统状态中与安全相关的因素涵盖在访问矩阵中;另一类引入“格”的概念,它是一个有限偏序集,是有最小上界和最大下界操作符的数学结构,利用格的性质来约束安全系统中的变量,以实现多级安全策略。因此,可以把安全模型分成基于访问控制矩阵的安全模型和基于格的安全模型。

1. 基于访问控制矩阵的安全模型

(1) Lampson 模型

在这种模型中,客体被认为是存储器,访问控制检查并非基于存储的内容而是基于系统的状态,系统状态中与安全相关的因素涵盖在访问矩阵中。在此模型中,系统状态由三元组(S, O, M)决定,其中 S 表示主体集合; O 表示客体集合,在某些情况下,主体 S 也可变成客体; M 是二维访问矩阵,用行来表示主体,用列来表示客体,主、客体的交叉点就表示主体对客体所拥有的访问权限,访问权限集包含读、写、追加、修改和执行等。系统状态的改变取决于访问矩阵 M 的改变,因而访问矩阵也称为系统的“保护状态”。系统中所有主体对客体的访问均由“引用监视器”控制,其任务是确保只有那些在访问矩阵中获得授权的操作才被允许执行。

(2) Graham-Denning 模型

此模型的保护性能更具一般性,对主体集合 S 、客体集合 O 、权力集合 R 和访问控制矩阵 A 进行操作。在矩阵中,每个主体拥有一行,对每个主体及所有客体都对应一列,一个主体对另一个主体或对一个客体的权力用矩阵元素表示。对于每个客体,标明为“拥有者”的主体拥有特殊权力;对于每个主体,对标明为“控制者”的另一主体有特殊权力。本模型设计多个基本保护权限,构造保护系统的访问控制机制模型所必需的一些性质,这些保护权限被表示成主体能够发出的命令,作用于其他主体或客体,具体包括:创建客体、删除客体;创建主体、删除主体;读访问权;授予访问权;删除访问权;转移访问权。

上述规则给出构造一个保护系统的访问控制机制模型所必需的性质,例如,这个机制可表示一种引用监视器。

(3) Harrison-Ruzzo-Ullman 模型

这是Graham-Denning模型的一个变种,在此模型中,基于命令(command)来描述主、客体的访问控制机制,每条命令均含有条件(condition)和原语操作(primitive operations),原语有:输入权限、删除权限、创建主体、创建客体、取消主体和取消客体。系统的保护基于三元组(S, O, P),其中, S 表示当前系统主体集, O 表示当前系统客体集, P 表示访问控制矩阵。在矩阵 P 中,行表示主体,列表示客体, $P[S, O]$ 是权限集的子集,表示主体 S 对客体 O 所拥有的权限。

2. 基于格的安全模型

(1) Bell-LaPadula 模型

BLP模型是最早和最常用的军事安全策略操作系统多级安全模型,其目标是详细说明计算机多级安全操作规则,此模型将主体定义为能发动行为的实体,如进程;将客体定义为主体行为

的被动承担者,如文件、目录、数据;将主体对客体的访问分为只读、可读写、只写、执行、控制等访问模式,控制是指主体授予或撤销另一主体对某客体的访问权限的能力。BLP 模型的安全策略包括:自主安全策略和强制安全策略。前者用一个访问控制矩阵表示,其中,第 i 行第 j 列的元素 M_{ij} 表示主体 S_i 对客体 O_j 的所有允许的访问模式,主体只能按照访问控制矩阵中被授予对客体的访问权限对客体进行访问;后者包括简单安全特性和 * 特性,系统对所有主体和客体都分配一个访问类属性,包括主体和客体的保密级别和范畴,系统通过比较主体和客体的访问类属性来控制主体对客体的访问。BLP 模型有以下两条基本规则。

① 简单安全规则

主体对客体进行读访问的必要条件是主体的安全级别支配客体的安全级别,即主体的安全级别不小于客体的保密级别,主体的范畴集包含客体的全部范畴,或者说主体只能向下读,不能向上读。例如,将军可阅读中校的文件,但反之则不允许。

② * 特性规则

主体对客体进行写访问的必要条件是客体的安全级别支配主体的安全级别,即客体的安全级别不小于主体的保密级别,客体的范畴集包含主体的全部范畴,或者说主体只能向上写,不能向下写。例如,中校可发消息给将军的信箱以告知情况,但反之则不允许。

如图 7.2 所示,从客体到主体的(实线)箭头代表此主体正在执行读访问,信息从客体流向主体;同样,从主体到客体的(虚线)箭头代表此主体正在执行写访问,信息从主体流向客体。这样所有的信息流都沿着箭头的方向流动,例如主体 B 可从客体 1 读取信息却不能从客体 3 读取信息。实际上,大多数现实的多级安全系统只允许主体向与其安全级别相等的客体写入信息。当然,为了使主体既能读客体,又能写客体,两者的安全级别必须相等。

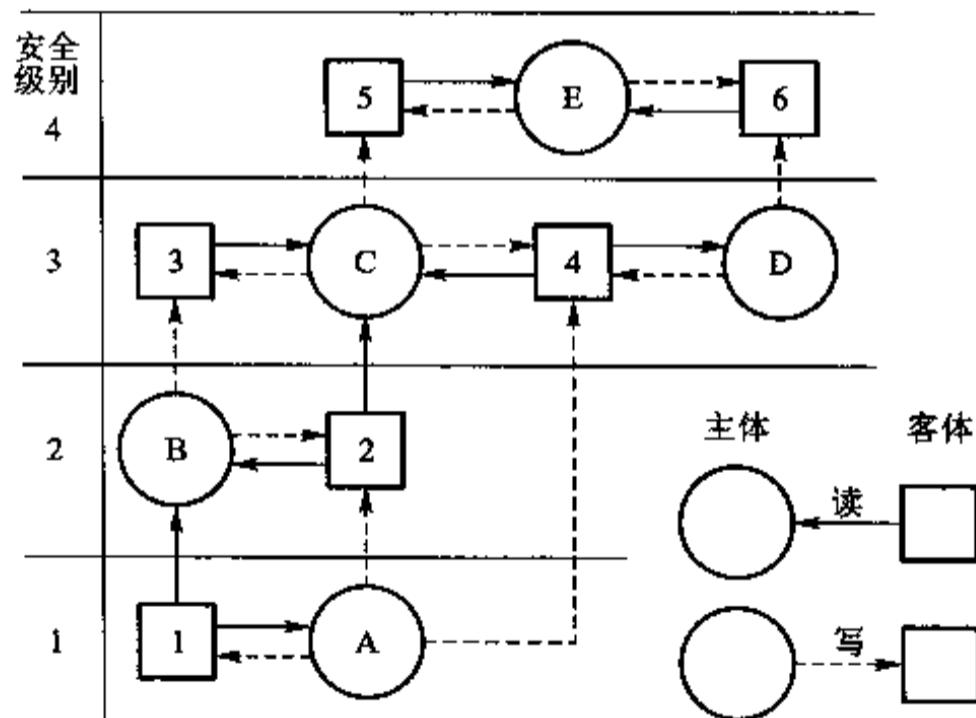


图 7.2 Bell-LaPadula 多级安全模型

上述两条规则表明,实线和虚线要么横向运动,要么纵向向上运动。既然信息要么水平,要么向上,那么从任何一层开始的信息都不可能出现在更低的级别,也就是说,没有路径可以让信

息向下流动,这就能保证模型的安全性。

BLP 模型采用状态机模型来形式化地定义系统,制定一组安全公理,给出一组系统状态和状态转换规则,定义安全的概念,制定一组安全特性,以此对系统状态和状态转换规则进行限制和约束,使得对于一个系统,如果初始状态是安全的,且所经过一系列的规则保持安全,那么,可以证明此系统是安全的。BLP 模型的影响渗透到所有操作系统的安全建模策略中,是第一个以规则形式反映现实系统属性的数学模型,也是构成安全性相关标准的依据,推动了对计算机安全理论的进一步研究。

(2) D. Denning 信息流模型

某些信息泄露问题(如隐蔽信道)不是因为访问控制机制不完善,而是由于缺乏对信息流的必要保护而引起的。在系统中,需要确定一个主体 S 能否获得资源 R 所包含的信息。在这种情况下,主体 S 不必具有对资源 R 的实际访问权限,信息可经由其他主体到达 S,或者信息只是简单地被复制到主体 S 可以访问的资源中。

例如,遵守 BLP 模型的系统应施行“下读上写”规则,即低安全级进程不能读高安全级文件,高安全级进程不能写低安全级文件。然而,在实际系统中,许多客体,如缓冲池、额定变量、全程计数器等,尽管不一定能直接被主体所访问,但还是可以被所有不同安全级别的主体所修改和读取,这样入侵者就可能利用客体间接地传递信息。一般说来,访问控制机制无法检测此类信息传输,要建立高安全级别的操作系统,必须在建立完善的访问控制机制的同时,依据适当的信息流模型实现对信息流的控制。

信息流模型是存取控制模型的变种,它不检查主体对客体的存取,而是试图控制从一个客体到另一个客体的信息传输过程,根据两个客体的安全属性来决定是否允许当前操作执行。隐蔽信道的核心是低安全级主体对高安全级主体所产生的信息的间接存取,信息流分析能保证操作系统在对敏感信息进行存取时,不会把数据泄露给调用者。

其他的安全模型还有 Biba 完整性模型、Clark-Wilson 完整性模型、Chinese-Wall 机密性完整性模型和基于角色的存取控制 RBAC 模型。

7.4 安全机制

操作系统是计算机硬件和用户之间的接口,不但要让用户通过此接口方便、高效地使用系统资源,而且还要保证计算机系统安全、可靠地运行,应提供机制和工具来实现安全策略所描述的安全问题。安全机制的主要目标是:根据安全策略对用户的操作进行控制,防止用户对系统资源的非法存取;标识系统中的用户并对其进行身份鉴别;监督系统运行的安全性;保证系统自身的安全性和完整性。为了实现这些目标,需要建立有效的安全机制。

7.4.1 硬件安全机制

优秀的硬件保护设施是实现高效、安全、可靠的操作系统的基础,计算机硬件安全的目标是

保证其自身的可靠性，并为操作系统提供基本安全设施，常用的方法有：主存保护、运行保护和 I/O 保护等。

1. 主存保护

对于安全操作系统而言，存储保护是最基本的要求，存储保护机制保证指令只能访问程序中被授权访问的存储单元，保护单位可以是存储段、页面、字块或字，保护单位越小，保护精度越高，但其开销也越大。在多道程序系统中，由于同时存在系统程序和应用程序，在运行时，操作系统必须把它们相互隔离以便互不干扰。如果进程能够不经意地写入系统或其他进程的存储空间，将导致严重的后果。既要控制进程对自己的存储区域的访问，又要控制对其他进程主存区的访问，需要主存保护功能，即必须保护进程不受其他进程的干扰。需要解决两个问题：一是进程必须被禁闭在操作系统分配给它的主存区内，此类禁闭的实施依赖于系统所提供的硬件保护机制及采用的存储管理方案；二是考虑进程对不同主存区的访问类型，在理想的情况下，对于给定的主存区，每个进程都应有自己的权限集合。

(1) 不支持虚拟主存的系统

为了保证进程禁闭在自己的区域内，需要对所访问的每个物理地址进行有效性检查，以验证它仅对所分得的主存区进行引用。使用以下三种保护机制实施主存保护。

① 下界和上界寄存器法

对寄存器指定一个连续主存区的下界和上界地址，进程的代码和数据便存放在此区域中。对于给定的逻辑地址，先与下界寄存器的内容相加，计算出对应的物理地址，计算结果与上界寄存器的内容作比较，如果超出边界值，则发出越界中断，并终止进程的执行。下界寄存器又称重定位寄存器，用来实现动态地址绑定，如果进程的代码和数据移动到新的主存区，只需相应地修改上下界寄存器的值。

② 基址和限长寄存器法

基址和限长寄存器指定一个连续主存区的起始地址和大小，其保护原理与下界和上界寄存器法相同，只是用限长寄存器代替上界寄存器而已。

以上两种保护机制仅提供对给定主存区“全有全无”的存储段粗粒度主存保护，且不能区分不同的访问类型。

③ 主存块的锁与进程的钥匙的配对法

如图 7.3 所示，把主存分成等长的物理块，访问判决是基于块来识别，每块附加一个 k 位二进制位组称为锁，每个进程的 PSW 中有 k 位二进制位组称为钥匙，当一个主存块被分配给进程时，就设定一个和进程钥匙等值的主存锁，只有内核程序才能设定进程钥匙和主存锁。每当进程运行引用主存时，硬件检查现行 PSW 中的钥匙与被访问主存块的锁的匹配情况，如果两者相符合，此次访问被许可；否则，就引起非法访问中断。

IBM System/370 操作系统采用钥匙和主存锁的方法作为存储保护和授权机制，固定分区的主存被分割成 2 KB 大小的主存块，每块都由硬件另外设置 7 个二进制位的存储控制键。其中，有 2 位指示此物理块的内容是否已被引用或修改过，以供页面替换算法使用；还有 4 位用做“主

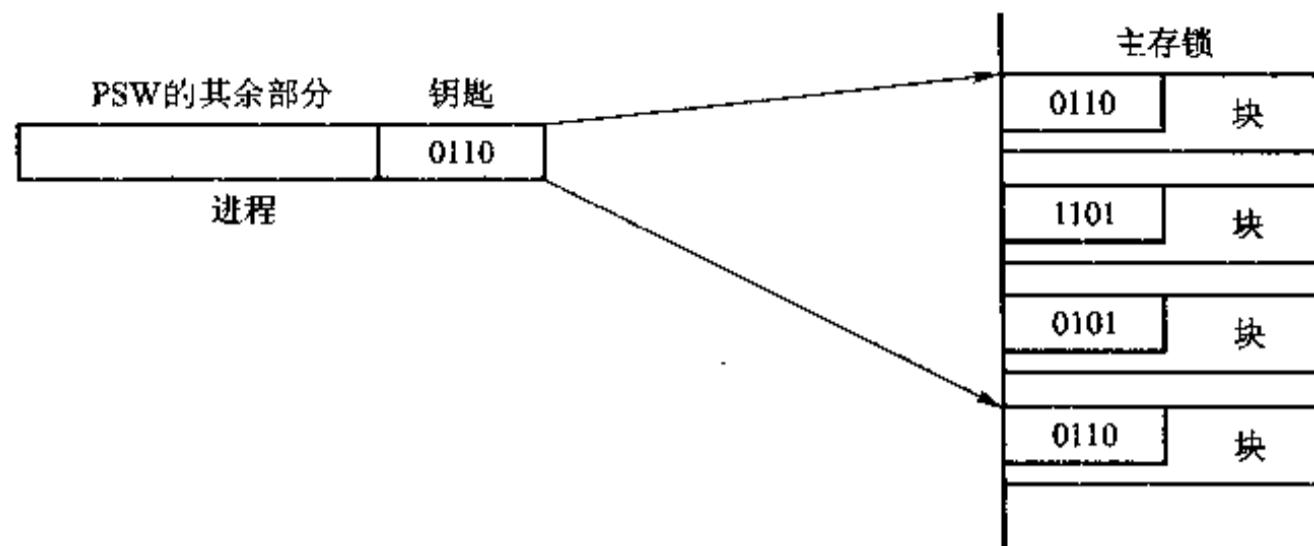


图 7.3 钥匙和主存锁

存锁”,其余1位称为“取保护位”。每个进程分得与其他进程不同的钥匙值,此进程的所有主存块均被设置与钥匙值相同的主存锁。当进程运行时,操作系统把分给它的钥匙放入PSW的存储控制键字段,每当处理器访问主存或DMA执行一个页面I/O操作时,都把钥匙和锁进行匹配,以决定是否允许访问。若运行进程试图访问钥匙和锁不符的主存块时,便会产生保护中断。应用进程使用值1~15;操作系统则使用值0,被允许访问整个主存储器,且有权修改和设置钥匙和主存锁。“取保护位”指示“主存锁”是适用于“写”还是“读写”,钥匙与锁匹配时允许写操作。若“取保护位”为1,则执行读操作时也要检查钥匙和锁是否匹配;若“取保护位”为0,即使钥匙和锁不匹配,也允许读访问,从而实现信息共享。

(2) 支持虚拟存储器的系统

进程存储空间的隔离可通过虚拟存储器的方法来实现,虚拟地址空间分为内核空间和用户空间。在用户态只能访问用户空间,在核心态可以访问内核空间和用户空间。内核空间在每个进程的虚拟地址空间中都是固定的,系统中仅有一个内核在运行,所有进程均映射到单一的核心地址空间。

进程的地址空间通过地址转换机制映射到不同的物理页面上,分段、分页或段页式虚拟存储管理系统是通过段表、页表和段页表间接地访问虚拟主存的一段或一页。由于管理表对于进程是私有的,因此,通过在相关表项中设置保护信息,就可表示进程对所访问(私有或共享)的段或页面具有不同的访问权限,以便在每次硬件保护机制进行地址转换时执行必需的权限检查,如果不允许,则产生一个异常。

具体实现技术如下:在分段或段页式虚拟存储管理中,可以在段表中增加访问类型字段,扩展逻辑地址到物理地址的映射计算步骤,增加检查申请访问类型是否是对此段的有效访问,若否,则触发保护中断,以保证每个虚拟地址总会禁闭在分配给进程的地址空间之内。这种方式的灵活性很大,段内信息的保护与逻辑命名空间相关,而与物理主存无关,不同的段可规定不相同的访问类型。

分页式虚存管理中,只需在页表中增加访问类型字段,信息保护的原理完全一样。Windows系统采用纯分页方式,通过不同的页面模式、页面类型和页面状态来完成页而安全性检查,具体

来说：

① 区分内核模式页面和用户模式页面：内核模式页面仅在核心态才可访问，且仅用于系统的数据结构，用户只能通过适当的内核函数对其访问；

② 区分页面类型：Win32 API 区分页面访问模式为不可访问、只读、读写、只能执行、执行和可读、执行和读写，其他模式对页面的访问都会造成违法；

③ 进程地址空间页面状态：有空闲、保留和提交三种，被记录在对应的页表项中。空闲页面是指尚未分配给进程的无效页面，任何访问企图都会产生无效页号错误；访问保留或提交页面可能会造成不同类型的缺页中断处理。

(3) 沙盒技术

在大多数操作系统中，进程所调用的函数会自动继承调用进程的所有访问权限，特别是可访问进程的整个虚拟主存。若函数是不可信的，如从因特网下载的程序（可能带有“特洛伊木马”病毒），这种不受限制的访问是不允许的，因为会给系统造成严重的威胁。为了限制不可信程序造成潜在损害的范围，系统可限制访问权限为调用进程所具有的授权的一小部分特权，通常称这种缩小后的访问环境为“沙盒”。最重要的一点是，限制程序在小范围的主存区（主存沙盒）中运行，访问沙盒以外的数据及转向沙盒以外的代码的企图都会产生保护中断，并终止程序的执行。主存沙盒的一种变体是为每个程序提供两个单独的沙盒，一个沙盒用于存放代码，另一个沙盒用于存放数据。只允许程序在数据沙盒中读写，且只能在代码沙盒中执行，这样进一步缩小程序对主存的访问权限，从而实现更严格的主存保护。

Java 提供一个可定制的沙盒安全模型，保护用户免受恶意程序代码的攻击，Java 程序可在沙盒范围内执行任何操作，但不能超越范围。对于非可信 Java Applet，沙盒会限制其活动，如读写本地磁盘、连接网络主机、建立新的进程、下载相关代码等都被施以一定的限制，使其没有实施有害动作的机会。Java 沙盒的基本组件有：类下载器、类文件验证器和安全管理器等。

2. 运行保护

“运行保护”机制为进程的运行设置不同的保护域，安全操作系统很重要的一点是进行分层设计，而运行域正是一种基于保护环的层次等级式结构。运行域是进程的运行区域，最内层拥有最小环号的环具有最高特权，最外层拥有最大环号的环具有最低特权，一般的系统不少于 3~4 个环。

最简单的方式是设置两层保护域：核心域和用户域，它隔离操作系统程序和应用程序。保护系统程序或应用程序不受其他程序的破坏，核心域运行的程序处于系统模式（核心态），用户域运行的程序处于用户模式（用户态）。如图 7.4 所示，运行在核心域下的程序比运行在用户域下的程序有更多的访问权限，以达到保护系统程序和其他应用程序的目的，这些权限包括可访问全部主存地址（包括 I/O 端口）和执行特权指令。

多层域结构中，最内层域存放操作系统程序；最靠近操作系统域的外层是存放受限制的系统程序（实用程序）域，如编辑程序、编译程序、汇编程序和数据库管理系统；最外层域存放应用程序。运行不同的程序时，最重要的安全措施是：

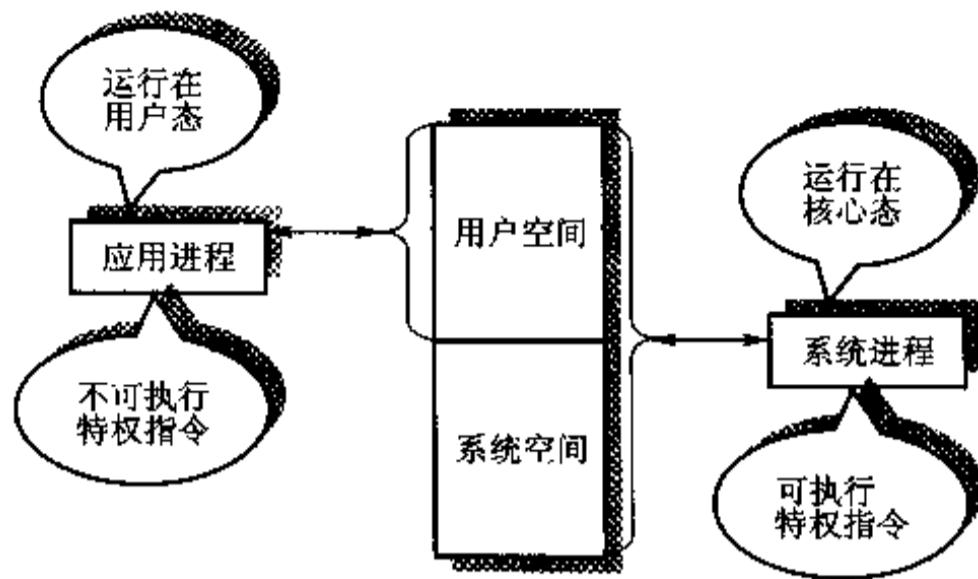


图 7.4 处理器模式扩展操作系统的访问权限

- (1) 程序可以访问驻留在同域或外域中的数据,但不会调用外域中的程序;
- (2) 程序可以调用驻留在同域或内域中的服务(函数),但不能访问内域中的数据。

这样可以保证等级域机制中某域不被外层域侵入。进程隔离机制与等级域机制不同,进程可在任何时刻在任何域内运行,在运行期间还可从一个域转移到另一个域。当进程在某域内运行时,进程隔离机制将保护此进程免遭在相同域内运行的其他进程的破坏,即系统将隔离同一域内同时运行的多个进程。

VAX/VMS 操作系统利用处理器的 4 种模式构成保护域来加强对系统资源的保护和共享。处理器模式决定:指令执行特权,即处理器当前可执行的指令系统子集;随当前模式增减的存储访问权限,即当前指令可存取的虚拟主存的位置。

- (1) 内核(kernel)态:执行操作系统内核,包括主存管理、中断处理、I/O 操作等。
- (2) 执行(executive)态:执行操作系统的系统调用,如文件操作等。
- (3) 监管(supervisor)态:执行操作系统其余的系统调用,如应答用户请求。
- (4) 用户(user)态:执行应用程序,如编辑、编译、链接等实用程序和各种应用程序。

在较低特权状态下执行的进程常常需要调用在更高特权状态下执行的例程。例如,应用程序需要操作系统的某种服务,使用访管指令可获得这种调用,此指令会引起中断,从而将控制权转交给处于高特权状态下的例程。执行返回指令可通过正常或异常中断返回至断点。采用保护环时具有单向调用关系,如果在域 D_i 中欲调用 D_j 中的例程,则必然有 $i < j$ 。

3. I/O 保护

CPU 与计算机系统所连接的各种设备进行通信时,采用两条途径:文件映射 I/O 和 I/O 指令。在前一条途径中,对设备的访问控制与主存保护机制相似,仅有授权的系统进程,如设备驱动程序和存储管理程序才能实现文件到主存区的文件映射操作。

采用后一条途径时,必须让这些 I/O 指令具有特权,以保证 I/O 设备不会被应用进程直接访问。相反地,想使用设备的应用进程发出一个 I/O 系统调用,切换至核心态并把控制权交给操作系统,系统的设备驱动程序代表发出调用的应用进程执行 I/O 操作。这些系统程序经过

专门的设计和调试,是一类可信软件,能以最有效和最安全的方式同设备交互,并可使用特权 I/O 机器指令。操作结束后,控制权返回给用户态调用进程,因而从用户的角度来看,一条 I/O 系统调用只是一个简单的高层 I/O 指令。

7.4.2 认证机制

1. 用户身份的标识与鉴别

认证机制主要包括标识和鉴别。标识是指操作系统识别用户身份,并将其转换为系统内部识别码——用户标识符,它是唯一的且不能伪造,以防止某用户冒充另一位用户。将用户标识符与用户关联的过程称为鉴别,此过程主要用于识别用户的真实身份,此操作需要用户具有能够证明其身份的特殊信息,且这些信息是秘密的和独一无二的。在很多操作系统中,标识和鉴别发生在用户登录时,系统提示用户输入用户名和密码,然后,检查用户所输入的密码是否与系统中所保存的此用户的口令相一致。密码鉴别机制虽然简便易行,但比较脆弱和不安全。

在安全操作系统中,TCB 要求先进行用户标识和鉴别,然后,才启动需要 TCB 调节的其他活动。所以,应建立一个登录进程来与用户交互,得到用于标识和鉴别的必要信息,TCB 必须能证实此用户的确对应于所提供的用户标识符,这就要求认证机制做好以下几项工作。

- (1) TCB 维护认证数据,包括用户身份信息、决定用户安全策略属性的信息,并确保那些代表用户行为的、位于 TCB 之外的主体的属性对安全策略予以满足;
- (2) TCB 保护认证数据,防止非法用户使用,当连续或不连续登录的失败次数超过规定次数时,不但将此事件记录到审计档案中,而且应暂时禁止此用户身份的使用,TCB 发送警告信息给系统控制台或管理员;
- (3) TCB 维护、显示、保护所有活动用户、用户账户信息;
- (4) 一旦把口令用做保护机制,TCB 应做好:单向加密和存储口令、自动隐藏口令明文、默认时禁止使用空口令、允许用户更换口令、口令失效管理、口令复杂性管理、维护口令文件或数据库、做好口令的使用和更改的审计等。

对于多级安全操作系统,认证过程不但要完成普通的用户管理和登录,如检查登录的用户名和密码、赋予用户的唯一标识用户 ID 和组 ID,还要核对用户所申请的安全级、计算特权集及审计屏蔽码。

2. UNIX/Linux 系统的标识和鉴别

系统的/etc/passwd 文件含有系统所掌握的用户登录信息,passwd 中的记录是用冒号隔开的 7 个字段,例如:

```
smith:xyz246ght:6759:4302:Jame Q smith:/home/smith/:/bin/sh
```

各项依次包含:用户登录名、加密口令、用户标识号 uid、用户组标识号 gid、注释字段(用户的全名及情况简介)、用户主目录和用户使用 shell 程序的路径。其中,uid 和 gid 用于系统唯一地标识用户和同组用户及用户的访问权限,每个用户至少属于一个用户组,每个用户组都有组标识。新版本将加密后的口令移至/etc/shadow 文件中。

用户登录系统时,需要输入用户名和密码供系统鉴别和标识其合法身份。用户名是标识,告诉计算机此用户是谁,而密码是确认证据,证明是正确的用户。当用户要求登录时,由守护进程 getty 接收用户所输入的用户名、密码并激活 login,它根据 /etc/passwd 文件检查匹配,如果是合法用户,就为他启动 shell 进程。

3. Kerberos 网络身份认证

Kerberos 网络认证机制是由美国麻省理工学院在 20 世纪 80 年代的研究计划 Athena 中所开发出来的著名密钥分发和鉴别系统,适用于客户—服务器模式的网络系统,用户可以在一个不安全网络中的多台计算机上使用此协议来对另一台计算机进行认证访问。

在 Kerberos 中,假设客户机 A 上的一个进程欲通过网络通信来调用应用服务器 B 上的一个进程的服务。Kerberos 提供身份认证服务(Authentication Service, AS)及票证颁发服务(Ticket Granting Service, TGS),允许客户和应用服务器在一个特定的会话中传送验证消息到合作进程。首先,客户必须向 KDC(Key Distribution Center, 密钥分发中心)的 AS 服务器证明自己的身份,然后,才能接收分发给自己的票证,可通过口令完成这项工作。此后,客户会收到在整个会话期均有效的用于访问专用 TGS 服务器的票证,此专用服务器的任务是分发其他应用服务器的票证。图 7.5 给出 Kerberos V.5 中使用应用服务器 B 的客户机 A 必须遵守的协议。

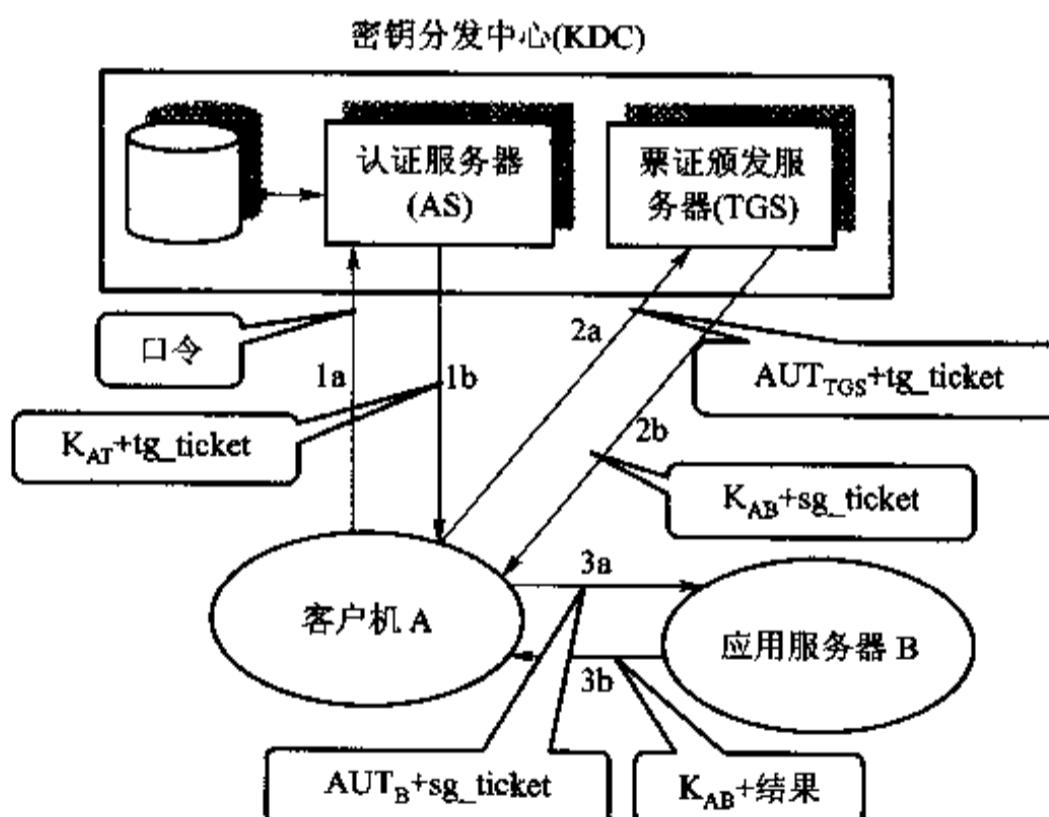


图 7.5 Kerberos 体系结构

步骤 1: 客户机 A 从 AS 处得到 TGS 所需要的许可票证。

1a. 登录时,客户机 A 向 AS 提供口令,以证明其合法身份,AS 在数据库中验证 A 的访问权限,并生成相应的票证。

1b. AS 向客户机 A 返回一条消息 $E(\{K_{AT}, tg_ticket\}, K_A)$, 其中, K_{AT} 是 A 和 TGS 通信的会话密钥, tg_ticket 是一个通往 TGS 的许可票证。此消息使用客户机 A 的密钥 K_A 进行加密,

因此,只有 A 可以获得 K_{AT} 和 tg_ticket 。

步骤 2:客户机 A 从 TGS 处得到应用服务器 B 所需要的许可票证。

2a. 客户机 A 向 TGS 申请针对所需应用服务器 B 的票证,在申请消息中,A 提供 tg_ticket (步骤 1 中已获得)、应用服务器标识 B 和一个认证符 AUT_{TGS} 。许可票证的形式为 $tg_ticket = E(\{A, TGS, K_{AT}\}, K_{TGS})$,此消息使用 TGS 的密钥 K_{TGS} 进行加密,只有 TGS 可以对其解密,其含义是:允许客户机 A 从 TGS 处取得应用服务器 B 的许可票证,并且必须使用 K_{AT} 进行认证。认证符 AUT_{TGS} 的形式为 $AUT_{TGS} = E(A, K_{AT})$,用于向 TGS 证明客户机 A 是此申请请求的创建者。

2b. TGS 向客户机 A 返回一条消息 $E(\{K_{AB}, sg_ticket\}, K_{AT})$,其中, K_{AB} 是用于 A 和 B 通信的会话密钥, sg_ticket 是通往应用服务器 B 的许可票证。此消息使用会话密钥 K_{AT} 进行加密,因此,只有客户机 A 可以获得 K_{AB} 和 sg_ticket 。

步骤 3:客户机 A 从应用服务器 B 处得到所需要的服务。

3a. 客户机 A 向应用服务器 B 请求所需要的服务。在请求中,客户机 A 提供 sg_ticket (步骤 2 中已获得)和一个认证符 AUT_B 。服务许可票证的形式为 $sg_ticket = E(\{A, B, K_{AB}\}, K_B)$,此消息使用应用服务器 B 的密钥进行加密,仅能被 B 所解密,具有以下含义:允许客户机 A 从应用服务器 B 处获得服务,并且客户机 A 必须使用密钥 K_{AB} 进行认证。认证符 AUT_B 的形式为 $AUT_B = E(A, K_{AB})$,用于向应用服务器 B 证明客户机 A 是此申请请求的创建者。

3b. 应用服务器 B 执行所请求的服务,然后,将结果返回客户机 A。如有需要,可用 K_{AB} 对结果进行加密。

在此协议中,密钥分发中心的服务器必须是可信的,因为它拥有客户身份的副本,知道如何加密只有客户才能解密的信息,并能生成一个独特的会话密钥去加密客户机和应用服务器之间的逻辑会话。同时,由于密钥分发中心的服务器能为客户机和应用服务器加密信息,它可向客户提供一个“容器”——许可票证,其中包含客户所不能读取的信息,但可以传送到应用服务器。这类似于一张信用卡,账号被加密放在其背面的磁条中,客户不能实际读出磁条中的内容,但当把此信用卡插入一台自动柜员机(应用服务器)时,柜员机能通过磁条读取账号信息,信用卡就是一张带有加密账号的许可票证(虽然它没有会话密钥)。

从上述协议步骤可以看出,客户进程和应用服务器进程都有一份会话密钥的副本,是从可信的密钥分发中心服务器中作为加密信息收到的。如果不知道如何解密,网络上的入侵者就不能读取这些加密信息,也不可能改变信息而生成有某种意义的欺骗票证。另外,应用服务器拥有客户身份的一份可信副本,因而当客户发送消息给应用服务器时,就能够认证客户身份和会话密钥。

7.4.3 授权机制

1. 授权机制的功能和安全系统模型

授权(authorization)是指在一个主体被认证后,确定它是否有权访问客体,只有在安全策略许可时才能访问允许其访问的客体。授权机制依赖于安全认证机制的存在,如图 7.6 所示是计

计算机系统两种机制的关键点,当用户试图访问计算机系统时,认证机制首先标识和鉴别用户身份,

筑起第一道保护墙;用户进入系统后,再由授权机制检查其是否拥有使用本机资源的权限及访问权限的大小,筑起第二道保护墙。

授权机制的主要功能是授权和访问控制,其任务如下。

- (1) 授权:确定给予哪些主体访问哪些客体的权力;
- (2) 确定访问权限,通常有:读、写、执行、删除、追加等方式;
- (3) 实施存取权限。

一个安全系统由符合安全策略的主体集、客体集、授权机制和访问监控器构成。“授权机制”在安全策略的指导下,负责确定通过访问权限定义的主体对客体的可访问性,系统确保每次对客体的访问都由“访问监控器”验证访问权限。在图 7.7 中,主体 S 访问客体 O 时,需要通过检查。只有当安全策略改变时才涉及授权机制的改变,从而提供新的访问权限集。

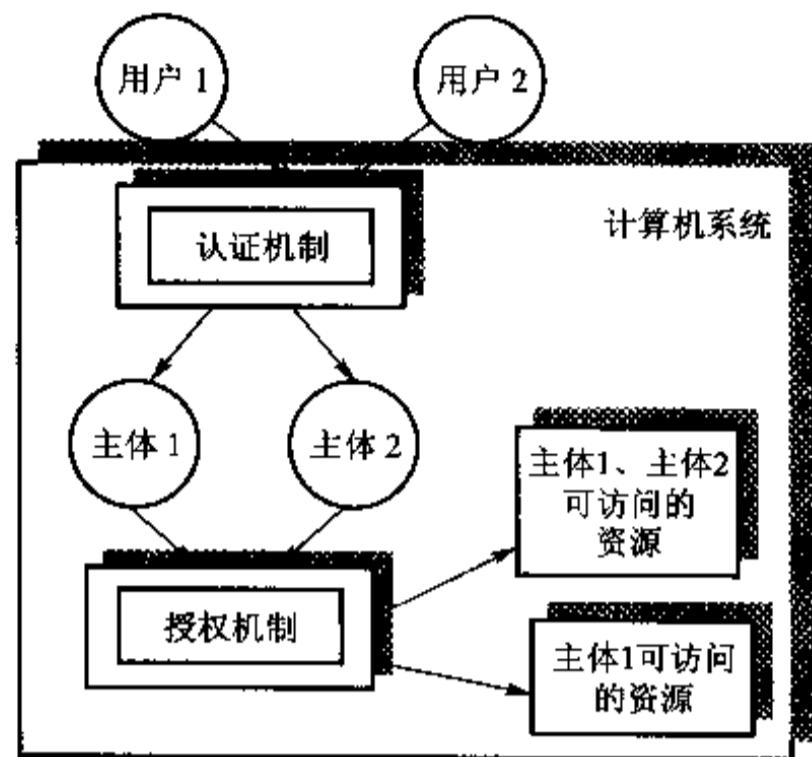


图 7.6 认证和授权

义的主体对客体的可访问性,系统确保每次对客体的访问都由“访问监控器”验证访问权限。在图 7.7 中,主体 S 访问客体 O 时,需要通过检查。只有当安全策略改变时才涉及授权机制的改变,从而提供新的访问权限集。

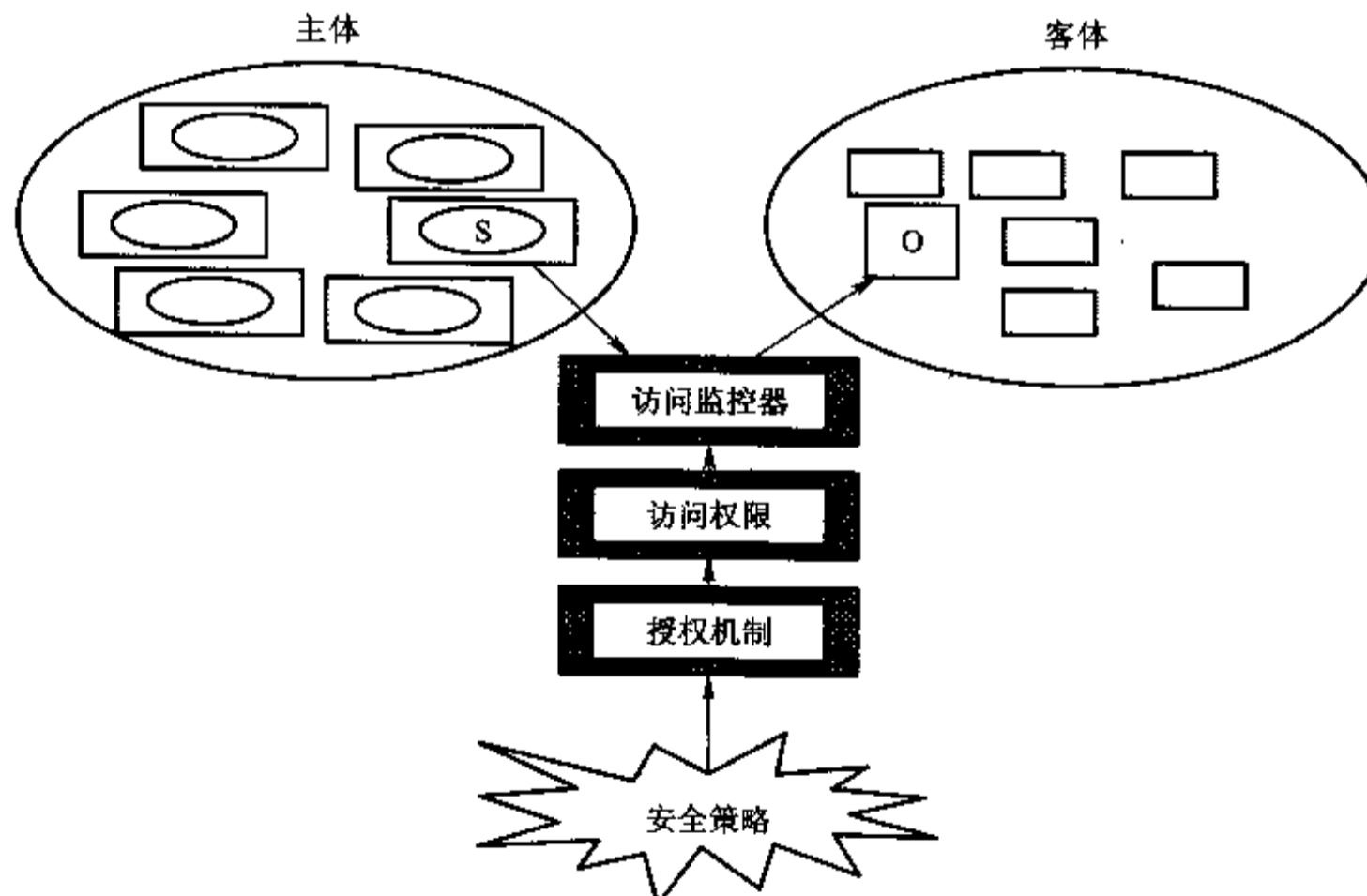


图 7.7 安全系统模型

在实际的安全操作系统中,还要对授权机制进行细分,它与安全策略相对应可分成:自主访问控制(Discretionary Access Control, DAC)机制和强制访问控制(Mandatory Access Control,

MAC)机制。

2. 自主访问控制机制

资源属主可以按照自己的意愿指定系统中的其他用户对其资源的访问权,故称自主访问控制,用户可自由说明所拥有的资源允许系统中的哪些用户以何种权限共享。另外,自主也指对其他具有某种访问权限的用户来说能自主地把访问权限或部分访问权限转授予另外的用户,几乎所有系统都把文件、目录、信号量及设备等纳入自主访问控制的范围。

实现完备的自主访问控制机制,必须建立访问控制矩阵模型,它试图回答有关安全性的基本问题:在给定的情况下,某个主体能够访问某个客体吗?图 7.8 给出访问控制矩阵非形式化描述的示例,由 3 个部分组成:客体(资源),以受控方式被访问;主体代表访问客体的主动实体;访问权限指示可作用于客体上的操作。如果把系统中的所有主体和所有客体组成 $n \times m$ 的二维存储矩阵,将是非常浪费的。在典型情况下,一个主体仅访问很少的客体,形成稀疏矩阵,可采用两种方式进行简化。

主体	客体		
	file 1	file 2	file 3
P ₁	read/write/execute	read	write
P ₂	read	read/write/execute	read
P ₃	read	write	read/write/execute

图 7.8 访问控制矩阵示例

(1) 基于行的自主访问控制机制

此方法相等于将访问矩阵以行的方式来存储,基于主体来实现访问控制矩阵,系统中的每个主体与一个信息表相联系,用来指明系统中的某个主体对哪些客体拥有访问权限。根据在主体上所附加的信息表的不同可分成以下三种。

① 权能表

权能表(Capability List, CL)决定用户是否可对客体进行访问及进行何种模式的访问,拥有相

应能力的主体可按照给定的模式访问客体。如图 7.9 所示是权能表的例子,列出每个进程对各个文件的访问权限。要采用加密技术对权能表进行保护,防止其被非法修改,系统需要为每个用户记录和维护权能表,此表包含许多条目,即使一个简单的问题“谁能存取这个文件?”也要花费系统的大量时间从每个用户的权能表中寻找。因此,由权能表所实现的自主访问控制机制的应用

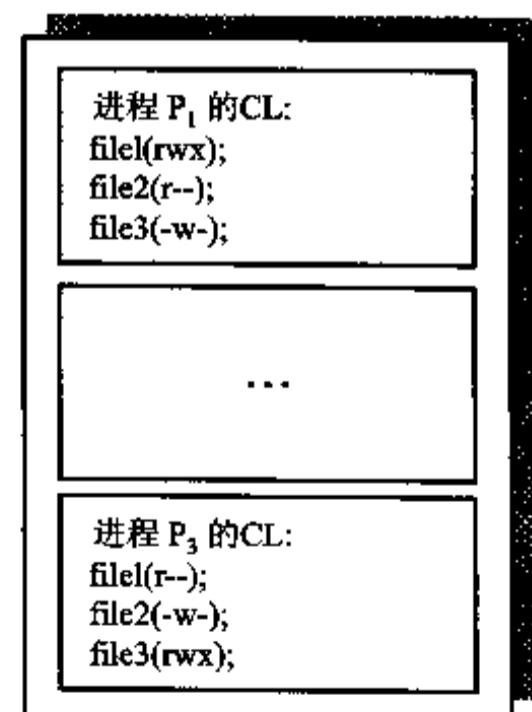


图 7.9 权能表

范围并不广。

② 前缀表

对每个主体赋予前缀(profile)表，它包含受保护的客体名及主体对它的访问权限。每当主体访问某个客体时，自主访问控制机制将检查主体的前缀是否具有其所请求的访问权。

前缀表实现起来方便，效率高。但是，在频繁更迭对客体的访问权限的环境下，这种方法就不太适宜。此外，还要求所有受保护的客体名必须唯一，不允许出现重名，这在资源众多的系统中应用起来十分困难。

③ 口令表

在基于口令表(password list)的自主访问控制机制中，每个客体都有一个口令，主体在对客体进行访问之前，必须向安全系统提供此客体的口令，如果正确方能允许访问。

若对系统中的每个客体，每个主体都拥有独有的口令，则其类似于权能表系统。不同之处在于，口令不像权能那样是动态的，系统允许对每个客体或对每个客体的每种访问模式都分配一个口令，一般来说，一个客体至少需要两个口令，分别用于读和写。口令的设置对于确认用户身份是行之有效的方法，但用于客体控制并不十分合适，因为只有通过改变客体的口令才能撤销某用户对一个客体的访问权限，这同时意味着废除所有其他可访问此客体的访问权限。口令依靠手工分发，系统无法知道哪个用户访问过此客体，当主体在运行期间要访问某客体时，此客体的口令必须嵌入程序中，这就会增加口令意外泄密的危险。

在基于权能表的系统中，很容易查出某一主体在此系统中所拥有的权限，只需检查此主体的权能表即可；相反地，如果要判定在所有主体中，有哪些主体能够访问某一特定客体，就不会像基于访问控制表的方法那样方便，需要遍历系统中所有主体的权能表才能做到。

相比较而言，一个权能可看成是一个票证，持有者可以执行某种特权；而一个访问控制表类似于一张登记表，仅有在表中登记名字的登记者才能行使某种特权。由于基于权能表所控制的安全访问都不甚成功，现代操作系统基本上都采用基于访问控制表的方法。

(2) 基于列的自主访问控制机制

此方法相当于将访问控制矩阵以列的方式来存储，在访问控制表(Access Control List, ACL)中，让每个客体与一个 ACL 相对应，此表指明系统中的每个主体获得对此客体访问的相应授权。ACL 是十分有效的自主访问控制机制，被许多系统所采用，其结构如图 7.10(a)所示。

借助于“*”可简化 ACL，“*”代表任何组名或主体标识符，优化的访问控制表如图 7.10(b)所示，GROUP5 中的用户 P_1 对文件 file1 拥有 rwx 权限，GROUP5 中的其他用户具有 x 权限； P_3 不在 GROUP5 中，只拥有 r 权限，其他用户不拥有权限。

通过查询客体的 ACL，系统“引用监视器”很容易判断试图访问此客体的主体是否有相应的访问权限，即在此客体的 ACL 中，是否有这个主体的相关项以及项中的访问权限是否包含所申请的访问权限；同时，也很容易撤销所有主体对一个客体的所有访问权限，只需将此客体的 ACL 设置为空即可。但另一方面，在基于 ACL 的系统中，要判定一个主体所拥有的所有访问权限是一件非常困难的事，必须要遍历系统中所有客体的 ACL，然后，组成此主体所拥有的访问权

(a) 访问控制表

		P ₁ (rwx)	P ₂ (r--)	P ₃ (r--)	...
file1					

(b) 优化的访问控制表

file 1		
P ₁	GROUPS	rwx
*	GROUP5	-x
P ₃	*	r-
*	*	---

图 7.10 ACL 和优化 ACL

限集。

(3) 自主访问控制机制实现举例

① “拥有者/同组用户/其他用户”模式

UNIX/Linux 的文件保护机制是一种简单的授权机制, 系统内的活动都可看做主体对客体的操作, 客体是信息实体, 或是从其他主体或客体接收信息的实体, 如文件、消息、网络包和设备; 主体通常是用户或代表用户的进程, 它引起信息在客体之间流动。访问控制机制的功能是控制系统中的主体对客体的访问, 如读、写、执行, 具体表现为通过一组访问控制规则来确定主体能否存取指定的客体。

每个主体都有唯一的用户 ID(uid), 且属于某个用户组, 每个用户组有唯一的用户组 ID(gid), 代表用户进程继承用户的 uid 和 gid, 实际上是为用户启动的 shell 进程, 此 shell 进程及其所有子孙进程都继承这个 uid 和 gid。

系统中能对客体进行访问的主体被分为客体属主、同组用户和其他用户, 对每个客体的访问模式分为: 读(r)、写(w)和执行(x), 这些信息构成一个简化的访问控制矩阵, 允许客体的属主和特权用户为客体设定访问控制信息。当主体访问客体时, 根据进程的 uid、gid 和客体的访问控制信息验证访问的合法性。

UNIX/Linux 的自由访问控制机制通过文件系统来实现, 每个文件附有 10 个二进制位所组成的存取权限位, 分别为: 文件类型、文件属主权限、同组用户权限和其他用户权限。授权模式同样适用于目录, 列目录要有“读”许可, 在目录中增删文件要有“写”许可, 进入或改变目录要有执行许可, 因此, 要使用文件, 必须有文件所在路径上的所有子目录的权限许可。

这种模式的缺点是: 客体的拥有者不能精确授予某个用户对此客体的访问权限, 例如, 不能指定同组用户或其他用户组中的 A 和 B 两个用户拥有不同的访问权限。

② “访问控制表”和“拥有者/同组用户/其他用户”结合模式

在安全操作系统 UNIX SVR 4.1 中,采用“访问控制表”和“拥有者/同组用户/其他用户”结合的实现方法,ACL 只对于“拥有者/同组同户/其他用户”无法分组的用户使用。两种自主的控制模式共存于系统,不但与以前的版本保持兼容性,而且将访问控制粒度细化到单个用户,系统能够赋予或取消某用户对某客体的访问权限,克服原系统仅把访问权限分配到用户组的粗粒度所带来的局限性。

在 UNIX SVR 4.1 中,每个文件对应于一个 ACL;在通信机制 IPC 中,每个消息队列/信号量集合/共享存储区对应于一个 ACL。一个 ACL 是对应于某个客体的三元组 $\langle \text{type}, \text{id}, \text{perm} \rangle$ 集合,每个三元组称为 ACL 的一个项,每项表示允许某个或某些用户对此客体的访问权限。其中,第一项 type 表示 id 是用户 ID 还是用户组 ID,perm 表示允许 id 所代表的用户对此客体的访问权限。用户可以对一个客体所对应的 ACL 进行以下三种操作。

- (a) “授权”用于将一个指定用户的标识符和相应权限加入 ACL 中;
- (b) “取消”用于从 ACL 中取消指定标识符项的某些访问权限;
- (c) “查阅”用于读取一个指定客体所对应的 ACL 内容。

自主访问控制机制的安全检验策略如下:若进程以 x 权限访问客体,则 x 必须在客体的相应 ACL 项中;若进程搜索路径,则它必须具有此路径中每个子目录的搜索权。例如,当进程访问文件时,调用自主访问控制机制,将进程 uid、gid 等用户标识信息和请求访问方式 mode 同 ACL 中的 ACL 项进行比较,检验是否允许进程以 mode 方式访问此文件。

自主访问控制机制是保护计算机系统资源不被非法访问的一种有效手段,同时也为用户提供了很大的灵活性。但是,“自主性”控制是其明显缺点,缺乏高安全性,如果需要更高的安全性,系统需要采用更强的存取控制手段和控制机制。

3. 强制访问控制机制

(1) 强制访问控制的概念

强制访问控制用于将系统中的信息分密级和范畴进行管理,保证用户只能访问那些被标明能够由其访问的信息的一种访问约束机制。系统中的每个主体(进程)、每个客体(文件、消息队列、信号量集、共享存储区等)都被赋予相应的安全属性,这些安全属性不能改变,它由安全系统自动地按照严格的规则来设置或由安全管理员管理,而不是像 ACL 那样由用户直接或间接地修改。当主体访问一个客体时,调用强制访问控制机制,根据主体的安全属性和访问方式,比较其与客体的安全属性,确定是否允许主体对客体进行访问。代表用户的进程不能改变自身或任何客体的安全属性,进程也不能简单地通过授权方式与其他用户共享客体,如采系统判定拥有某一安全属性的主体不能访问某个客体,那么,没有任何办法可让此主体实现访问,这一条是“强制的”。

一个操作系统往往把两种机制结合起来使用,仅当主体同时通过自主访问控制和强制访问控制检验时,才能访问一个客体。用户使用自主访问控制防止其他用户非法入侵自己的文件,强制访问控制则用来作为更强的安全手段,使用户不能通过意外事件和有意的操作来逃避安全检查。

强制访问控制通常与多级安全体系相提并论,多级安全的思想最早由美国国防部研究开发保护计算机机密信息的项目提出,称为军事安全策略。多级安全是军事安全策略的数学描述,是计算机能够实现的一种安全形式化定义,多级安全访问控制机制的实现大多基于 BLP 模型,在实现之前,首先必须对系统的主体和客体分别赋予与其身份相对应的安全属性的外在表示——安全标签,它由两部分组成:{安全类别:范畴}。

① 安全类别——有等级的分类

- (a) 安全级别:也称为密级,系统用来保护信息(客体)的安全程度。
- (b) 敏感性标签:这是客体安全级别的外在表示,系统利用敏感性标签来判定进程是否拥有对此客体的访问权限。
- (c) 许可级别:进程(主体)的安全级别,用来判定此进程对信息的访问程度。
- (d) 许可标签:这是进程安全级别的外在表示,系统利用进程的安全级别来判定它是否拥有对要访问信息的相应权限。

② 范畴——无等级概念

范畴是指相应安全级别信息所涉及的部门,通过安全标签机制,可以实现多级安全策略,建立多个不同安全标签的分类信息,控制用户只能访问那些允许其访问的信息(客体),其中,安全标签的类别部分用来告诉系统某用户(主体)访问信息的可信级别以及可信程度,而范畴部分是系统用来判定此用户是否属于信息所对应的部门。例如,在公司内,可以建立信息安全的类别:Confidential Restricted(技术信息)、Restricted(内部信息)和 Unrestricted(公开信息);在军事部门,典型的信息安全类别:Top Secret(绝密)、Secret(秘密)、Confidential(机密)和 Unclassified(公开)。安全级别是按等级分类的,例如,安全级别以降序排列,因此,Confidential Restricted 大于 Restricted,Top Secret 大于 Confidential。在一个信息分类中,可同时建立不同的范畴,例如,公司内的范畴可以是:Accounting(财务部)、Marketing(市场部)、Advertising(广告部)、Engineering(工程部)和 Research & Development(研发部),在公司内,财务部经理与市场部经理的级别虽然相同,但由于两人分属不同的部门,从而分属两个不同的范畴(Accounting 与 Marketing),故市场部经理不能访问财务部经理的信息。

(2) 强制访问控制的实现

① 安全标签的实现

基于多级安全策略,系统的访问控制机制的实现要基于以下内容:

- (a) 对每个客体赋予一个敏感性标签,此标签标明系统用来保护信息的安全程度(级别);对每个主体(进程)赋予一个许可标签,此标签用来指定此进程的可信程度;
 - (b) 系统通过基于主体(进程)的许可标签和客体(信息)的敏感性标签来判定此进程是否拥有对这一信息的相应访问权限;
 - (c) 系统保证对所有 I/O 信息都能正确地标记反映其安全级别的敏感性标签。
- 基于多级安全策略的安全标签分为:安全类别与范畴,其实现要点如下。
- (a) 类别部分:由于类别部分所反映的是一种等级关系,故又称安全级别或密级。在安全类

别中,密级按线性顺序排列,例如,公开<机密<秘密<绝密,在实现时以数字从小到大依次递增表示其安全等级。

(b) 范畴部分:范畴部分由无等级概念的元素组成,表示一个清晰的信息领域。范畴集之间不存在等级,但具有“包含”或“被包含”的关系,也可以是无关的。在实现时范畴用数字代表,范畴集是数字的集合表示。

安全标签包括一个密级和一个范畴集,可以有任意多个范畴,如{机密:研发部、财务部}。实际上,范畴集经常是空的,且很少会有多个范畴。

两个安全标签 A 和 B 之间存在以下 4 种关系:

(a) A 支配 B:当且仅当 A 的安全等级大于或等于 B 的安全等级,A 的范畴集包含 B 的范畴集,即 B 的范畴集是 A 的范畴集的子集。

(b) B 支配 A:当且仅当 B 的安全等级大于或等于 A 的安全等级,B 的范畴集包含 A 的范畴集,即 A 的范畴集是 B 的范畴集的子集。

(c) A 等于 B:当且仅当 A 的安全等级等于 B 的安全等级,A 的范畴集中的任何一项也是 B 的范畴集中的一项,反之亦然,即 A 支配 B,B 支配 A。

(d) A 与 B 无关:A 安全等级的范畴集不包含 B 安全等级的范畴集,同时 B 安全等级的范畴集也不包含 A 安全等级的范畴集。

支配表示为“<”是一种偏序关系,类似于“大于或等于”的意思。例如,一个文件的安全级别是{机密:参谋部,后勤部},如果用户安全级别是{绝密:作战部,参谋部,后勤部},则可以阅读此文件,因为用户安全级别高且涵盖文件的范畴集,反之,具有安全级别{绝密:作战部,参谋部}的用户不能读此文件,其原因是缺少后勤部范畴。

② 基于安全标签的强制访问控制

安全系统控制主体对客体的访问基于它们的安全标签,故应设计标签比较算法,用来判断安全级别标签的支配关系,以决定主体对客体的访问权限。一个进程(主体)访问资源(客体),其许可标签必须具备:

(a) 若想要对客体具有“写”访问权限,主体的安全级别{安全等级 + 范畴}必须被客体的安全级别支配,也就是说,主体安全等级 < 客体安全等级,主体范畴集 < 客体范畴集。

(b) 若想要对客体具有“读”访问权限,主体的安全级别{安全等级 + 范畴}必须支配客体的安全级别,也就是说,主体安全等级 > 客体安全等级,主体范畴集 > 客体范畴集。

如果进程(主体)不能满足上述任何一条要求,则不能访问客体,基于主体、客体安全标签的强制访问控制机制的示例如图 7.11 所示。

(3) 强制访问控制的应用

“特洛伊木马”病毒是一段计算机程序,其表面在执行合法任务,实际上却具有用户不曾预料到的非法功能。由于它通常继承应用程序的用户 ID、存取权限、优先级甚至特权级,故能够在不破坏系统安全规则的情况下进行非法操作。防止“特洛伊木马”病毒极其困难,如不依赖强制手段,想避免其破坏是不可能的。在强制访问控制的场合下,对于违反强制访问控制的“特洛伊木

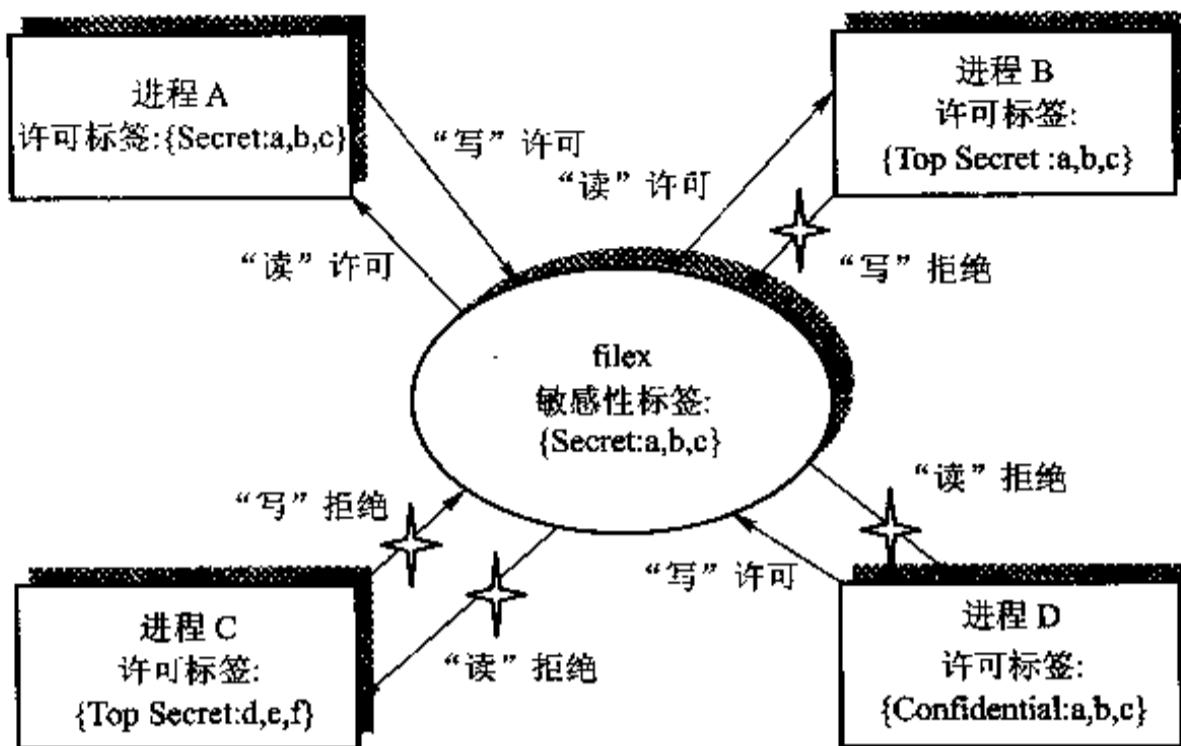


图 7.11 基于安全标签的强制访问控制机制

马”病毒,可防止它取走信息。例如,在多级安全系统中,*特性能阻止正在机密级运行的进程中的“特洛伊木马”把机密信息写入一个公开的文件内,因为用机密进程写入的文件的安全级别必须至少是机密级;再如,一个公司对系统中自己拥有的信息指定强制访问范畴,只有公司雇员才可能进入这个范畴,如果一名雇员使用“特洛伊木马”,他不可能将此公司的信息传递到这个范畴以外的地点,但在这个范畴内,信息可以在用户之间自由传递。

为了防止“特洛伊木马”口令截获攻击,Linux 提供“安全注意键”以促使用户确认自己的用户名和密码不被他人窃取,其工作过程如下:当用户输入“安全注意键”时,系统通过中断系统陷入核心态,内核接收中断并处理它,杀死当前终端的所有应用进程(包括“特洛伊木马”病毒),并重新激活登录界面而为用户开通可信登录路径,然后,用户就可以放心地输入合法用户名及密码。

4. 最小特权原理

(1) 最小特权原理

为了使系统能正常运行,系统中的某些进程,如安全管理员、网络管理员和系统操作员,需要具有一些可违反安全策略的操作能力,定义一个特权就是定义一个可违反系统安全策略的操作能力,它与存取控制结合使用,提高了系统的安全性。在现有的多用户操作系统中,超级用户通常拥有所有特权,而普通用户却没有任何特权,这种特权方式便于系统的配置和维护,但不利于系统的安全性,一旦超级用户的密码丢失或超级用户被他人冒充,会对系统造成极大损失。超级用户的误操作也是系统的潜在隐患。因此,必须实行一种特殊授权机制——最小特权原理(Least Privilege Principle)。

最小特权原理是指:系统中的每个主体只能拥有与其操作相符的必需的最小特权集,特别是不应给超级用户、用户、进程和程序超过执行任务所需特权以外的任何特权。POSIX 中指出,要想在系统安全性方面达到合理的保障程度,必须严格地实施最小特权原理。

① 在硬件特权方面,当处理器不是以特权模式或特权域方式运行时,必须限制特权机器指令的使用,限制对某些存储区域的访问;

② 在软件特权方面,由操作系统赋予某些程序恰好够用的特权,这些特权允许超越在应用程序上实施的常规访问控制,或者调用所选择的函数,具有多种类型软件特权的系统允许在最小特权方面实施细粒度控制;

③ 进程在系统中代表用户和系统管理员的行为,不应获得多于完成其工作所需的特权。

(2) 最小特权管理的实现

最小特权管理的实现可分为以下两个方面。

① 用户特权

将超级用户的特权划分为一组细粒度的特权,分别授予不同的系统操作员和管理员,使他们各自只拥有完成其任务所需要的最少特权。例如,可将系统的管理分为 n 个权限,然后,将 n 个权限中用来实现同一功能的若干权限组成一个子权限集。从 UNIX 系统的实际出发,至少需要以下子特权集:系统审计特权集、系统安全管理特权集、系统日常管理特权集、网络管理特权集等,然后,将这些特权集分别映射到不同的管理用户身上。

(a) 系统安全管理员:用来维护系统中与安全性相关的信息,包括对系统用户账户的管理,系统中的主、客体安全级别的赋予,限制隐蔽通道活动及设置和维护系统的安全数据库;

(b) 系统审计员:用来设置审计开关和审计阈值,启动和关闭审计机制,修改和删除审计信息,及管理审计日志;

(c) 系统操作员:用来完成系统中的日常操作和维护,包括开启和关闭系统、磁盘的一致性检查、格式化新介质及设置终端参数等;

(d) 安全操作员:除了完成操作员的责任之外,具体实施安全性操作,如安全性定义、档案的备份与恢复、文件系统的安装与卸载;

(e) 网络管理员:用来完成与网络相关的所有管理与操作,如网络软件管理、网络通信管理、设置与连接服务器、启动和停止网络文件系统。

② 进程特权和文件特权

对可执行文件赋予相应的特权集,这样可以让非特权用户完成部分特权但又不具有其他特权,对每个进程可根据其所执行的程序和所代表的用户,赋予相应的特权集。当进程请求一个特权操作(如安装文件系统)时,将调用特权管理机制,判定此进程的特权集中是否具有所需的操作特权。这样,特权将不再和用户标识相关,即不是基于用户 ID,它直接与进程及可执行文件相关联。一个新进程所继承的特权既有进程的特权,也有所执行文件的特权,这种机制一般称为“基于文件的特权机制”,其最大优点是特权的细化,其继承性提供了执行进程可增加特权的能力。

7.4.4 加密机制

1. 数据加密模型

加密(encryption)是用某种方式伪装信息以隐藏其内容的过程。加密的关键是要能高效地

建立从根本上不可能被未授权用户解密的加密算法,以提高信息系统及数据的安全性和保密性,防止信息被窃取和泄密。数据加密技术分为两类:一类是数据传输加密技术,其目的是对网络传输中的数据流加密,又分成链加密和端加密;另一类是数据存储加密技术,其目的是防止系统中所存储的数据泄密,又分成文件级(对单个文件)加密和驱动器级(对逻辑驱动器上的所有文件)加密。

加密技术可用于将明文转化为密文来保护暴露在未受保护的介质上的原文,定义加密函数(算法)和解密函数(算法),使用加密函数和加密密钥把明文变成密文的过程称为加密;使用解密函数和解密密钥将密文转化为原始明文的过程称为解密。明文(plain text)是指被加密的文本;密文(cipher text)是指加密后的文本;密钥是加解密算法中所使用的关键参数。

一般而言,密码系统依其应用范围可对信息提供下列功能。

- (1) 秘密性:解决信息泄漏问题,防止非法的信息接收者窃取信息;
- (2) 鉴别性:解决发送者的真实性问题,信息接收者能够确认信息来源,即此信息确实是发送方传送而非他人伪造的;
- (3) 完整性:解决信息被修改或损坏的问题,信息接收者能验证信息未被修改,也不可能被假消息所替代;
- (4) 不可抵赖:发送者在事后不可能虚假地否认其所发送的信息。

2. 基于密钥的算法分类

(1) 对称算法

对称算法又称传统密码算法,就是加密密钥能够从解密密钥中推算出来,反之亦然。在大多数对称算法中,加密/解密密钥是相同的,这种算法的安全性依赖于密钥,泄漏密钥就意味着任何人都能对信息进行加密和解密。对称算法分为两种:一是只对明文中的字位(或字节)运算的算法,称序列算法或序列密码;二是对明文中的一组字位运算,这些位组称为分组,相应的算法称为分组算法或分组密码。现代计算机密码的典型分组长度为 64 位,这一长度大到足以防止分析破译,但又小到足以方便使用。

对称技术的优点是具有很高的保密强度,可经受国家级破译力量的分析和攻击,但其密钥必须通过安全、可靠的途径来传递,密钥管理成为影响系统安全的关键性因素,这使其难以满足系统的开放性要求。

(2) 公开密钥算法

公开密钥算法又称非对称密码算法,其设计机理是:用做加密的密钥不同于用做解密的密钥,而且解密密钥不能根据加密密钥计算出来。之所以称其为公开密钥算法,是因为加密密钥可以公开,其他人能够用加密密钥来加密信息,但只有用相应的解密密钥才能解密信息,因此,加密密钥称为“公开密钥”,解密密钥称为“私有密钥”。

采用非对称密码体制的每位用户都有一对选定的密钥,一个可公开,一个由用户秘密保存。公开密钥算法的出现是现代密码学研究的一项重大突破,其主要优点是:适应开放性的使用环境,密钥管理简单,可以方便、安全地实现数字签名和验证,但其保密强度目前还远未达到对称密

码体制的水平,至今所发明的非对称密码体制绝大多数已被破译,剩下的几种也不能证明其完全不存在缺陷。

3. 计算机密码算法

(1) 数据加密标准

数据加密标准(Data Encryption Standard,DES)是通用的计算机加密算法,是美国标准,用于政府和商业应用。这是一种对称算法,加密/解密密钥是相同的,20世纪70年代中期由美国IBM公司开发。尽管DES历经几十年头,但在已知的公开文献中,仍然无法彻底地把DES破解,这种加密方法至今仍被公认是安全的。

由于DES的密钥只有56位,仅有256个不同的密钥,随着计算机运行速度的不断提高,容易采用尝试法破解DES密文。于是,3次DES加密算法被提出来,它依次使用3个56位密钥进行DES处理,使密钥长度增加至168位,使用目前世界上运算速度最快的计算机穷尽尝试,将需要万亿年之久。

(2) 公开密钥算法 RSA

RSA(Rivest,Shamir,Adleman)是最流行的公开密钥算法,已成为事实上的国际标准,能被用做加密和数字签名。DES属于传统的加密算法,要求加解密的密钥是对称的,加密者必须用非常安全的方法把密钥送给解密者,如果通过计算机网络来传送密钥,则密钥就有泄密的可能。彻底解决这个问题的方法是不分配密钥,这就是公开密钥法。要求密钥是对称的,并不是一个必要条件,可以设计出算法,加密时用一个密钥,而解密时使用有联系的另一个密钥。另外,还可能设计出一个算法,即使知道加密算法和加密密钥也无法确定解密密钥。其基本技术如下:

- ① 网络中每个节点都产生一对密钥,用来对其所接收的消息进行加密和解密;
- ② 每个系统都把加密密钥放在公共文件中,这是公开密钥,另一个设置为私有的,称为私有密钥;
- ③ 若A要向B发送消息,它就用B的公开密钥加密消息;
- ④ 当B收到消息时,就用私钥进行解密。由于只有B知道其私钥,于是没有其他接收者可破解消息。公开密钥算法的主要缺点是算法复杂,开销大,效率低。

4. 数字签名

在金融和机要系统中,许多业务都要求签名,以备检查和核实,任何签名都不得伪造,也不可抵赖。公开密钥算法可用于数字签名,以代替传统的手工签名。

(1) 简单数字签名

- ① 发送者A使用私有解密密钥对明文进行加密,所形成的密文传送给接收者B;
- ② B利用A的公开加密密钥对所得密文进行解密,便得到明文。除了A之外,没有人具有解密密钥,因此,也只有A才能送出用他的解密密钥加密过的密文;
- ③ 如果A要对自己的行为抵赖,只需出示其解密密钥加密过的密文,使其无法抵赖。

(2) 保密数字签名

上述方法能解决数字签名的问题,但不能达到保密的目的,因为任何人都能接收密文,并可

用 A 的公开加密密钥对所得密文进行解密。为了使 A 所传送的密文只让 B 接收, 可按下面的步骤进行。

- ① 发送者 A 使用私有解密密钥对明文进行加密, 得到密文 1;
- ② A 再用 B 的公开加密密钥对密文 1 进行加密, 得到密文 2, 再将其传送给 B;
- ③ B 收到密文 2 之后, 先用自己的私有密钥对密文 2 进行解密, 得到密文 1;
- ④ B 再利用 A 的公开加密密钥对所得密文 1 进行解密, 于是得到明文。

5. 网络加密

防止网络资源被窃取、泄露和篡改的最好方法是网络加密。那么, 要决定加密什么, 在网络中的何处加密。通常有两种网络加密技术: 链 - 链加密和端 - 端加密。

(1) 链 - 链加密

对相邻节点之间通信链路上所传输的数据报文进行加密, 叫做链 - 链加密。通常物理层接口是标准的, 在此处最方便配置硬件加密装置, 对通过它们的所有数据进行加密, 包括数据、路由信息、协议信息等, 从而通信链路上所流动的信息都是安全的, 在发送端和接收端之间的任何智能交换或存储节点都必须在处理数据之前对其进行解密。其优点是能实现流量保密, 不易泄密; 其缺点是每次包交换都要加密和解密, 因为交换时必须从报头部读出地址来获得路由。另外, 网络中的每个物理链路都必须加密, 开销会非常大。链路加密常采用序列加密算法, 它能有效地防止搭线窃听所造成的威胁。那么, 应在何时对数据进行加/解密呢? 对于不同的数据链路控制规程是不完全相同的, 如面向字符的同步传输规程可利用报头标识符作为加密数据的开始信号, 而面向比特的传输规程可利用帧标识符作为加/解密的开始和结束信号。

(2) 端 - 端加密

把加密设备配置在网络层和传输层之间, 只对传输层的数据加密, 加密数据与未加密的路由信息重新结合, 送往下一层传输。这样, 通信线路上所传输的是密文数据, 到目的地再被解密, 可保证在中间节点不会出现明文。但是, 在端 - 端加密方式中, 路由信息并未进行加密, 密码分析者能据此进行流量分析, 判新谁和谁在通信及其发生时间, 频繁程度如何, 而并无须知道通信内容; 其次, 密钥管理困难, 每位用户必须确保与其他人有相同的密钥; 另外, 通信系统都有各自的协议, 使得端 - 端加密设备制造困难。

(3) 链 - 链加密和端 - 端加密的组合

把上述两种方法结合起来使用, 尽管这样做价格昂贵, 但却是有效的网络安全传输方法。加密各条物理链路使得报头以密文形式在链路中传输, 从而对路由分析不再可能; 而端 - 端加密使得用户数据以密文形式穿越各个中间节点, 减少网络节点中未加密数据处理所带来的威胁。对两种方法的密钥管理可以完全分开: 网络管理员关心物理层密钥, 用户负责相应的端 - 端加密。

7.4.5 审计机制

审计(auditing)就是对系统中有关安全的活动进行完整记录、检查及审核, 作为一种事后追踪手段来保证系统的安全性, 是对系统安全性所实施的一种技术措施, 也是对付计算机犯罪者的

利器。其目的是检测和阻止非法用户侵入系统,显示合法用户的误操作,进行事故发生的预测和报警,提供事故发生后进行分析和处理的依据,如违反系统安全规则的事件的发生地点、时间、类型、过程、结果和所涉及的主体、客体及其安全级别。

如果将审计与报警功能结合起来,就能做到每当有违反系统安全的事件发生或有涉及系统安全的重要操作进行时,就及时向安全操作员终端发送报警信息,安全效果会更好。审计过程是一个独立的过程,应将其与系统的其他功能隔离开来,同时要求操作系统能够生成、维护、保护审计过程,使其免遭非法访问和破坏。特别要保护审计数据,严格禁止未经授权的用户访问。

1. 审计事件

审计事件是系统审计安全性活动的基本单位,系统把所有要求审计或可以审计的用户操作都归纳成可识别和可标识的用户行为和可记录的审计单位,即审计事件。例如,创建文件的操作是由 `create()` 系统调用实现的,为了反映用户的这个操作,当执行系统调用时,系统可设置事件 `create`,包括创建文件的文件名、访问模式等参数,将此事件作为审计事件由审计机制记录下来。

审计机制把主、客体都定义为可审计对象,安全操作系统通常将所要审计的事件分成:注册事件(用户的标识、鉴别和退出)、使用系统或访问资源的事件(特权用户所执行的操作、创建/删除文件、执行程序、修改密码/安全级别)、管理员及操作员实施的操作事件和利用隐蔽信道的事件(隐蔽存储通道发生的活动)等。这些审计事件有些属于系统外部事件,即进入系统的用户所产生的事件,另外一些属于系统内部事件,即已进入系统的用户发生的事件。由于审计会增大系统开销,所以,系统应该确认和选择重要的事件加以审计,系统审计员可以通过设置审计事件标准来确定对系统中的哪些用户或哪些事件进行审计。

2. 审计记录和审计日志

审计记录一般包括以下信息:事件的日期和时间、事件主体的唯一标识、事件类型、事件结果(成功/失败)等。对于认证事件,应记录事件发生的地点(终端标识符);对于客体的创建或删除事件,应记录客体名及其安全级。

审计日志是存放审计记录的二进制码文件,每当启动审计进程之后,都会按照设定的路径和命名规则产生新的审计日志文件,系统审计员有权查询和打印审计日志文件中的审计结果,有权选择所需要的内容,如与某用户、某事件或某个时间有关的记录信息。

3. 审计机制的实现

实现审计机制,首先要求系统中所有安全相关的事件都能被审计到,操作系统的用户接口主要是系统调用,当用户请求系统服务时,必定要使用系统调用。因此,把系统调用的总入口位置称作审计点,在此增加审计控制,就可以成功地审计系统调用,也就能全面地审计系统中所有使用内核服务的事件。

调用系统中的特权命令应属于可审计事件。一条特权命令需要使用多个系统调用,如果逐一审计所用到的系统调用,会使审计数据变得复杂和难于理解。所以,对特权命令的审计还需要增加措施,可在被审计的特权命令的每个出口处安排一个新的系统调用,专用于此命令的审计。

当发生可审计事件时,要在审计点调用审计函数并向审计进程发送消息,由审计进程完成审

计信息的缓冲、写盘和归档工作。另外,审计机制应提供灵活的选择手段,使审计员可以开/闭审计机制、增/删系统或用户审计事件类型、修改审计控制参数等。

可审计事件是否被写入审计日志需要由审计机制进行判定,可在同审计事件相关的操作的程序入口处、出口处设置审计点。在程序入口处的审计点进行审计条件的判断,若需要审计,则设置审计状态并分配主存空间;在程序出口处的审计点收集审计内容,包括操作的类型、参数和结果。系统在记录审计信息时会增加时间开销,因此,不必每有一条审计记录便立即将其写入日志文件,可开辟审计缓冲区,系统在大多数情况下只要把审计记录放入审计缓冲区,仅当缓冲区被存满时才写盘一次。

4. 审计缓冲区的设计

审计数据来自于系统的各处,在审计点所搜集的各种审计事件信息汇集到一起,待信息量达到一定的程度,就要将其写入物理介质(磁盘、磁带),以供分析使用。由于 I/O 操作十分耗时,为了提高系统整体效率,必然要引入缓冲区。在缓冲区的设计中,可用多个相同大小的缓冲区构成缓冲池以便提高并发度,当某进程写某个缓冲区时,审计驻留进程可读取其他就绪缓冲区,至于缓冲区的大小和数目,可由审计管理员根据实际系统的运行状况和审计范围灵活地进行调整,审计缓冲机制如图 7.12 所示。对于缓冲区的管理,可通过专门设计的缓冲区控制结构来实现,缓冲区控制结构的数据项包括:缓冲区基址、缓冲区大小、缓冲区个数、读指针、写指针、缓冲区临界水位线等。

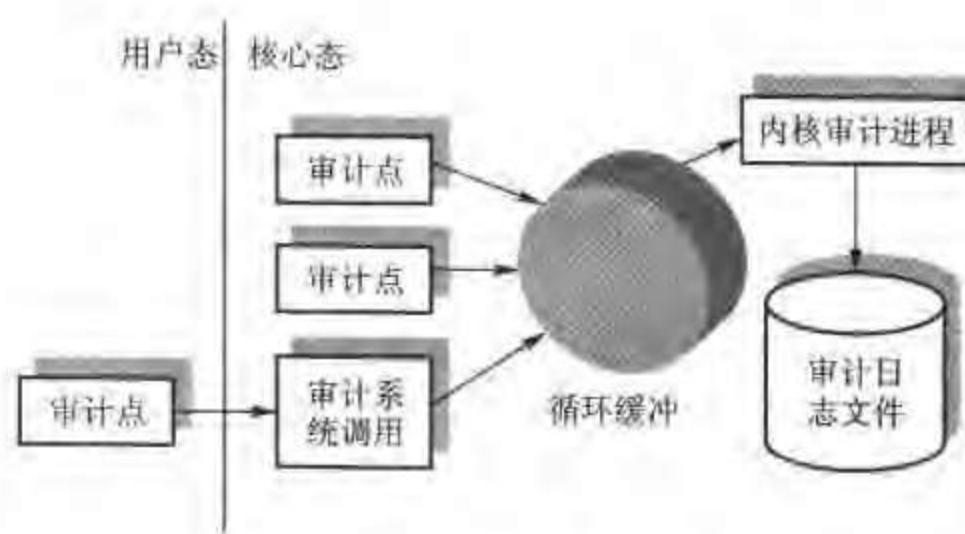


图 7.12 审计缓冲机制

7.5 安全操作系统设计和开发

7.5.1 安全操作系统的结构和设计原则

在通用操作系统中,除了提供常规的资源管理功能之外,还要设计和实现与安全性有关的机制,如用户身份认证、存储空间隔离、文件访问控制、设备访问控制、资源共享、互斥、同步和通信

机制等。但是,高安全操作系统要求在任何环境下都能安全、可靠地运行,对其安全性的要求十分严格。由于安全功能渗透于操作系统的整体设计和结构中,这就意味着在设计安全操作系统时,必须在系统设计的各个方面都考虑安全性。安全性必须是操作系统初始设计的一部分,每当设计一部分后,应及时检查其所提供的安全程度,在设计上不够安全的系统仅靠修修补补达到高安全性是很困难的。Saltzer J. H. 和 Schroeder M. D. 给出设计安全系统和保护系统的原则。

1. 公开系统设计方案

保护系统的设计方案必须公开,“假设入侵者不知道系统的工作原理”无疑是自欺欺人,保护机制必须不依赖于潜在攻击者的无知。保护机制的公开设计还有利于接受广泛的公开审查,安全性不能依赖于保密而达到。认为用户没有软件手册和源程序清单就不能进入系统,无疑是一种很危险的观点。所以,为了安全起见,最保险的假设就是认为入侵者已经了解系统内部的一切,设计保密也不是许多安全系统的需求,将必要的机制引入系统后,应使得即便是系统的开发者也不能入侵这个系统。

2. 机制的经济性

所设计的保护系统小型化、简单,应该能承受穷举测试、严格的测试或验证,因而是可以信赖的。

3. 最小特权

此原则蕴涵着小粒度的保护方案,为了使无意或恶意的攻击所造成的损失降至最低限度,每个用户和进程必须按照“需知”的规则,尽可能地赋予最小特权进行操作。例如,编辑器只有权存取其所编辑的文件,这时即使带有“特洛伊木马”病毒,也不会造成太大的损失。

4. 严密的访问控制机制

对于每个访问操作必须检查其权限,以确定此访问的合法性。

5. 基于许可的模式

许可是基于否认的背景的,其默认条件应该是拒绝访问。应当标识什么资源是可存取的,而非标识什么资源是不可存取的。

6. 特权分离

对实体的存取应该依赖于多个安全条件,如用户身份鉴别及密钥,这样侵入保护系统者将不会拥有对全部资源的访问权。

7. 避免信息流的潜在通道

可共享实体提供了信息流的潜在通道。系统为了防止这种潜在通道所带来的威胁,采取物理或逻辑分离方法。

8. 便于使用

所设计的安全机制应使用方便,接口友善。

安全操作系统(secure operating system)是指能对所管理的数据和资源提供适当的保护级以便有效地控制软硬件功能的操作系统。在此基础上,能够进一步实现多级安全策略的安全操作系统称为多级安全操作系统。

计算机系统中的软件分为三类。

(1) 可信软件:是指由设计人员根据严格标准开发出来并通过软件工程技术仔细编写和调试过,甚至被证明是正确的软件。它保证能够安全运行,但是系统安全仍然依赖于对软件的无错操作。

(2) 良性软件:并不确保安全运行,但由于使用特权或对敏感信息的访问权,必须确信它不会有意违反规则。

(3) 恶意软件:软件来源不明,从安全的角度出发,必须将其视为恶意软件,即认为它将对系统进行破坏。

在大多数情况下,认为操作系统内核是可信软件,而其他系统软件和应用程序是不可信软件。不可信软件即使进行正确的操作,也不能保证结果是正确的。在系统设计中,不允许不可信软件破坏和修改可信软件。

安全操作系统的一般结构如图 7.13 所示,安全内核用来控制整个计算机系统的安全操作;可信应用软件分为两部分,系统管理员和操作员运行安全管理所需的应用软件及具有特权的、保障系统正常工作所需的应用软件;用户软件由可信软件以外的应用程序所组成。操作系统的可信应用软件和安全内核组成系统的可信软件,它们是可信计算基(TCB)的一部分,系统必须保护可信软件不被修改和破坏。

高安全级别操作系统对其内核进行分解以产生安全内核,安全内核是指内核中分离出来的、与系统安全控制相关的部分软件。如 Ford 太空通信公司的 KSOS 和美国加利福尼亚大学洛杉矶分校 UCLA Secure UNIX 均具有这种结构,它们的安全内核足够小,故能对其进行严格的安全性验证。低开发代价的安全操作系统不再对其内核进行分解,安全内核就是内核。这种结构的例子有 LINUS IV、Secure Xenix、Tmach 和 Secure TUNIS 等。

7.5.2 安全操作系统的开发

1. 安全操作系统的研究和发展

1965 年,由美国 AT&T 公司贝尔实验室、麻省理工学院等联合开发的 MULTICS 是开发安全操作系统的最早尝试,BLP 机密性安全模型首次成功地应用于 MULTICS。从 20 世纪 60 年代末到 20 世纪 70 年代,许多安全思想和机制都被提出来,如 Lampson 的主、客体与访问控制矩阵、隐蔽通道等概念;Anderson 的参照监视器、引用验证机制、授权机制、安全内核和安全建模;接着,很多安全操作系统被开发出来,如,美国国防部发起和资助、Ford 太空通信公司承担和研发的安全内核操作系统 KSOS(Kernelized Secure Operating System)和美国加利福尼亚大学洛杉矶分校承担和研发的 UCLA Secure UNIX;1983 年,美国国防部出版历史上第一个计算机系统安全评价标准《可信计算机系统评价标准(TCSEC)》(又称橘皮书)。20 世纪 80 年代所开发的安全操作系统有:LINUS IV,此系统基于 BSD 4.1 UNIX,实现身份鉴别、自主控制访问、强制控制访

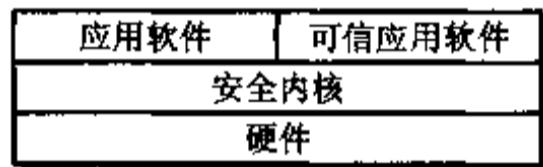


图 7.13 安全操作系统的一般结构

问、安全审计、特权用户权限分离等安全功能;Secure Xenix,此系统基于 SCO Xenix,引入安全形式化验证,实现可信通路和安全功能测试,将试验改进/增强法加入安全机制。此外,还有基于 Mach 的 Tmach、基于 UNIX 的 System V /MLS 和 TUNIS 等安全操作系统。20世纪 90 年代所开发的安全操作系统有 ASOS(Army Secure Operating System)、Flask、OSF/1、UNIX SVR 4.1 ES、DTOS、SE-Linux、STOP、VMM 等;1991 年,欧洲共同体推出《信息技术安全评价标准 (ITSEC)》;1996 年,美国、加拿大和欧洲共同体联合发布《通用安全评价标准(CC)》,其改进版已被确立为国际标准,即 ISO/IEC15408。

2. 安全操作系统的一般开发方法

设计安全内核与设计操作系统类似,要使用常规操作系统设计方法和技术,只是在设计安全内核时,必须优先考虑完整性、隔离性、可验证性原则,而非通常对操作系统而言更为重要的高性能、低成本、易用性和灵活性。安全操作系统的开发有两种做法,一是自设计开始,就建立完整的安全操作系统开发,包括软硬件在内的设计,这种做法除了少数国家的少数系统(如美国 MULTICS)有过,在其他系统中并不常见;二是在现有的非安全操作系统上增强和引入安全机制,从而形成安全操作系统,如美国数据通用公司的 DG UX/B1 和 DG UX/B2,其前身就是 DG UX 操作系统。然而,用此方法所开发的操作系统的安全级别不会很高,要进行离级别安全操作系统的开发还需要从头做起。

基于第二种方法开发安全操作系统,一般采用以下 3 种方法。

(1) 虚拟机法

在原有操作系统和硬件之间增加一个分层,作为安全内核,操作系统几乎不变地作为虚拟机运行,安全内核的接口同原有的硬件接口等价,其自身并未意识到已被安全内核所控制,仍然像在裸机上一样执行自己的系统功能,因此,它可不变地支持现有的应用程序,且能很好地兼容原操作系统的未来版本。

虚拟机法在 IBM 系列机中运用得相当成功,其原因是 IBM370 硬件与原操作系统 VM/370 的结构都支持虚拟机,采用此法开发操作系统的安全性时,硬件特性对虚拟机的实现起着至关重要的作用,因而,虚拟机法开发安全操作系统的局限性比较大。

(2) 改进/增强法

在原有操作系统的基础上,对其内核和应用程序进行面向安全策略的分析,引入安全机制,经改造后的安全系统基本上保持原通用操作系统的用户接口。采用这种方法的主要问题是拘于体系结构和现有应用程序的限制,很难达到较高的安全级别,但它不破坏原系统的体系结构,开发代价小,且基本上保持原来的系统效率。

(3) 仿真法

对现有操作系统的内核做面向安全策略的修改,在安全内核和原操作系统用户界面之间再编写一层仿真程序,这样在建立安全内核时,可不受现有应用程序的限制,且能够自由地定义仿真程序和安全内核之间的接口。不过采用这种方法要同时设计仿真程序和安全内核,还会受到上层通用操作系统接口的限制。另外,根据安全策略,有些通用操作系统的接口功能不安全,因

而不能仿真;有些接口的功能虽然安全,但仿真实现起来特别困难。

安全操作系统的开发过程包括以下一些步骤。

(1) 系统需求分析:描述各种安全需求;

(2) 系统功能描述:准确地定义应实现的安全功能,包括描述验证,即证明描述与需求分析相符;

(3) 系统实现:设计并建立系统,包括实现验证,即论证实现与功能描述相符。

安全需求分析涉及对系统的安全部分的描述和分析,这一步比起对系统的功能需求分析来说要简单得多,可采用非形式化途径,通过论证和测试等步骤,分别证明安全功能的描述和系统的实现满足安全需求分析。但是以自然语言作为描述工具,极易造成歧义和遗漏,且无法进行数学证明。所以,当系统要求高安全保证时,可采用形式化途径,将自然语言所写的安全需求分析表示为数学形式的抽象模型,它与安全需求分析严格地出自同一个安全策略,自然语言所表示的安全功能描述可写成能够由计算机处理的形式化描述语言,论证和测试一致性步骤代之以数学证明。当然,形式化途径只是非形式化途径的一种补充,二者有效地结合起来,才能较好地完成安全需求分析任务,在此基础上进一步抽象、归纳出系统安全策略。

下一步是建立安全模型,它是安全策略与安全机制之间的桥梁,描述计算机系统和用户的安全特性,帮助设计者尽可能精确地描述系统安全功能,以堵住安全漏洞。此外,还要对模型进行对应性分析,考虑如何将模型应用于系统开发之中,并说明所建模型与安全策略的一致性。

建立安全模型之后,再结合系统的特点,选择实现这一模型的方法,逐步建立安全模型所要求的安全访问机制。同时,在设计一部分安全功能之后,便检查其所提供的安全性尺度。在安全操作系统设计完成后,就要进行反复测试和安全性分析,将结果提交给相应的评测部门,对其安全可信度进行认证。

3. 安全功能和安全保证

安全操作系统的开发应从安全功能(security function)和安全保证(security assurance)两方面加以实施,安全功能说明一个操作系统所实现的安全策略和安全机制符合评价准则中哪一级功能的要求,而安全保证(保障)是通过一定的方法保证操作系统所提供的安全功能确实达到指定功能要求,能够确保系统的安全性。这就要求在设计一个安全操作系统时,首先要按照安全需求分析确定总体安全应达到的安全保护等级,然后,进一步明确此安全保护等级所规定的安全功能和安全保证的要求。

安全功能包括 10 个安全元素:标识与鉴别、自主访问控制、标记、强制访问控制、客体重用、审计、数据完整性、可信路径、隐蔽信道分析和可信恢复。在通用操作系统中,可信计算基(TCB)包含多个安全功能 TSF(Trusted Security Function)模块,每一个 TSF 实现一种安全策略 TSP(Trusted Security Policy),这些 TSP 共同构成安全域,以防止不可信主体的干扰和篡改。安全保证则涵盖 3 个方面:TCB 自身安全保护,包括 TSF 模块、资源利用、TCB 访问等;TCB 的设计和实现,包括配置管理、分发和操作、开发、指导性文档、生命周期支持、测试、脆弱性评定等;TCB 安全管理。

4. 安全操作系统的设计技术

在此对安全操作系统设计中所使用的一些技术加以综述。

(1) 隔离技术

将系统中的用户(进程)与其他用户(进程)隔离开来是安全性的基本要求,可通过4种方法实现:物理分离、时间分离、密码分离和逻辑分离。

① 物理分离:各进程使用不同的硬件设施,例如,敏感计算通过保留的计算机系统完成,非敏感任务则在公共计算机系统上运行;为打印机设置不同的安全级别,不同安全级别的进程使用不同的打印机。

② 时间分离:让各种进程在不同的时间运行,如某些军用系统从上午8:00到中午12:00运行非敏感任务,而敏感计算只在中午12:00到下午5:00之间运行。

③ 密码分离:进程以其他进程不可知的方式隐藏敏感数据和程序,使得未经授权的用户不能访问。

④ 逻辑分离:也称隔离,限制进程的活动空间,使得其不能访问允许范围以外的客体,让用户感觉似乎没有其他进程存在而可独自占用机器。

安全操作系统使用所有这些分离方法。早期应用隔离技术比较成功的系统有IBM多虚存操作系统MVS和虚拟机操作系统VM。

(2) 安全内核

核(kernel)又称内核(nucleus)或核心(core),在传统的操作系统中,它实现底层功能,如进程通信、同步机制、中断处理及基本的主存管理。安全内核(security kernel)是通过控制对系统资源的访问来实现基本安全规程的操作系统内核中相对独立的一部分程序,它在硬件和操作系统功能模块之间提供安全接口,凡是与安全有关的功能和机制都必须被隔离在安全内核之中。

安全内核所实现的与安全性有关的机制有:认证机制、访问控制机制、授权机制及审计机制等。安全内核方法是常用的建立安全操作系统的方法,其设计和开发以一系列严格的原则作为基础,能够极大地提高用户对系统安全控制的可信度。安全内核的理论依据是:在大型操作系统中,只有其中的一小部分软件用于安全,所以,在建立安全操作系统的过程中,集中与安全相关的软件来构成操作系统的可信内核,即安全内核。安全内核必须受到保护,不能被篡改,也绝不能有任何绕过安全内核访问控制检查的存取行为发生。安全内核的设计和实现要符合以下3条基本原则。

① 完整性:要求主体访问客体时必须通过安全内核,所有信息的访问都不能绕过安全内核。其他的完整性要求是:硬件必须确保程序不能绕过内核的存取控制、未通过内核控制的进程之间不能通信、对主存和设备等的引用必须经过访问控制机制的合法性检查。

② 隔离性:通过把安全功能和机制与操作系统功能模块和用户空间分离开来,易保护安全机制不被上层程序所侵入和破坏;要求安全内核具有防篡改的能力,保护自己防止偶然破坏。实施隔离需要硬件和软件相结合,硬件的特性是使安全内核能防止上层程序访问它的代码和数据,这与内核防止进程访问其他进程区域是同一种主存管理机制。同时,还必须防止应用程序执行

安全内核用于控制主存管理机制的特权指令,这需要域控制或环保护机制。在拥有这些硬件特性的系统中,应用程序几乎不能通过写安全内核的存储区域、执行特权指令或修改内核代码等方法使安全内核受到直接攻击。

③ 可验证性:通过采用以下设计方法和技术达到可验证性:新的软件工程技术(结构化设计、模块化设计、信息隐蔽、层次分级、抽象说明及使用合适的高级语言等);内核接口精简、内核小型化、代码测试与检查、安全性测试、形式化数学描述与验证。

安全内核的设计和可用性在某种程度上取决于设计方法,对其介绍如下。

① 单独设计安全内核功能

安全内核由介于硬件和操作系统之间的软件层所组成,硬件和安全内核是可信的,处于安全周界(security perimeter)内,但操作系统和应用软件处于安全周界之外。安全内核向操作系统提供服务,操作系统向应用程序提供服务,正如操作系统要对应用程序施加限制一样,安全内核也要对操作系统施加限制。当安全策略完全由安全内核而非操作系统实现时,仍需操作系统维持系统的正常运行,防止由于应用程序的致命错误而引发的拒绝服务,但是操作系统与应用程序的任何错误均不能破坏安全内核的安全策略。

安全内核必须维护每个层次的保密性和完整性,要监控4种基本的交互活动。

(a) 进程激活:在多道程序设计环境下,进程的创建和撤销会频繁地发生,进程上下文切换要求改变控制寄存器、重定位映像、文件访问表、进程状态及其他相关信息,其中许多都是对安全性敏感的信息。

(b) 区域切换:在一个区域运行的进程频繁地调用或访问其他区域中的程序和数据,以获取更多的敏感数据和服务。

(c) 存储保护:因为每个存储区域中都包含代码和数据,安全内核必须监控对主存储器的引用以确保每个存储区域的保密性和完整性。

(d) I/O 操作:由于所有 I/O 操作不是向设备写数据就是从设备接收数据,所以,进行 I/O 操作的进程必须受到设备读写两种访问控制机制的监督。

Honeywell 公司在设计 SCOMP 安全操作系统时采用安全内核技术,安全内核包含 20 多个模块,约 10 000 行源码,用来实现安全功能。

② 将安全功能融合到操作系统中

从理论上讲,安全内核也可以实现操作系统的所有功能,如果设计者在安全内核中所加入的操作系统功能和特点越多,安全内核就变得越大,接近一个通常的操作系统,此时其安全信任度越差。

(3) 分层设计

层次分级适用于将安全内核分离的安全操作系统的设计,自内层至外层可想象为同心圆结构,越敏感的软件越是处于较内层,反之则处于较外层;越是内层的信息越可信,反之则越不可信。所构成的分层安全操作系统的层次依次为:硬件—安全机制—同步和通信机制—存储管理、调度、共享—设备管理—文件管理—实用功能程序—系统程序(编辑、编译、汇编、DBMS)—应用

进程的子进程—应用进程。

在这种设计中,某些安全功能是在安全内核之外完成的。例如,用户认证包含一张口令表,用以查验用户密码的正确性,在安全内核之内全部完成这类任务的缺点是其中的一些操作(如设置用户终端的交互形式、在已知用户表中搜索用户)并不需要高度安全性。在实现过程中,安全内核通常与可信模块交互,如认证功能可以在安全内核之外的模块中完成,这些模块是可信的,因为它们已经被验证或审查过,可以高度信赖其可靠性,认证模块适当地完成一些机械性认证工作。

如图 7.14 所示,在多个模块中实现单一功能是分层设计所能做到的,可信赖的访问权限是分层的基础,一个功能可由不同层次中的一组模块来完成,而每一层的模块均实现一部分敏感操作。

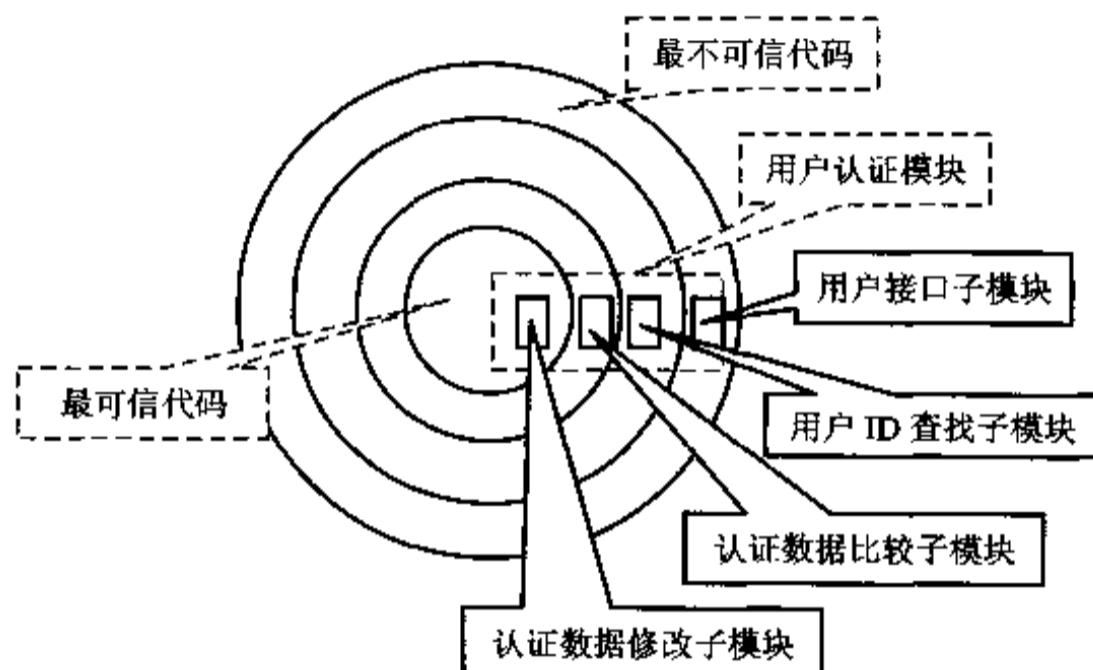


图 7.14 不同层中实现的认证模块

分层方法是一种很好的安全操作系统设计方法,每一层把距中心更近的那些内层用做对自己所提供的服务,每一层又向外部层次提供服务功能。采用这种方法,使得“剥去”个别层次后的操作系统仍然是一个逻辑上完全的系统,仅仅是功能减少一点而已。

MULTICS 操作系统用环结构(ring structure)实现安全保护,这是分层设计的更进一步发展。为了指定各个进程所具有的访问权限,共设计 8 个环,其中 4 个环用于操作系统,另外 4 个环用于应用程序。在 MULTICS 中,一个环就是一个进程在其中所执行的一个运行域,各环从 0 号起向上编号,内核为 0 号环,环的实现可想象为围绕计算机系统所使用的同心环带。每个执行的进程运行在某个特定的环级上,越可信赖的进程操作在越低的环级上,环是交叠的,在 i 号环上运行的进程包含 j 号环的所有特权,其中 $i < j$; 环号越低,进程所拥有的访问权限越多,它能访问的主存区域越大。

7.5.3 信息系统安全评价标准简介

1. 操作系统安全漏洞扫描

为了确保安全操作系统的安全性,人们首先考虑采用专用工具来扫描操作系统的安全漏洞,

以达到发现漏洞和采取补救措施的目的。操作系统安全漏洞扫描的目的是：自动评估由于操作系统的固有缺陷或配置不当所造成的安全漏洞。操作系统安全漏洞扫描的主要内容有：

(1) 设置错误

从安全的角度来看，操作系统软件的设置是很困难的，一个微小的设置错误可能导致一系列安全漏洞，扫描工具软件可以检查系统设置，搜索安全漏洞，判断是否符合安全策略。

(2) 黑客踪迹

黑客所留下的踪迹往往可以检测到，例如，扫描工具软件能检查是否有黑客侵入系统、盗取口令，也可检查某些关键目录下是否有黑客所放置的可疑文件。

(3) 特洛伊木马

黑客常常在系统内嵌入这类程序，对安全的威胁很大，扫描工具软件试图找出这种程序的存在。

(4) 系统文件完整性威胁

扫描工具软件能够检查系统文件的非授权修改和不合适的版本，既检测漏洞，又有利于版本控制。

2. 操作系统安全评测方法

一个操作系统是安全的，是指它满足给定的安全策略，其安全性与设计密切相关。只有有效保证从设计者到使用者都相信设计准确地表达模型，并且代码准确地表达设计时，这样的操作系统才可以说是安全的，这也是安全操作系统评测的主要内容。常用的安全操作系统安全测评方法有三种。

(1) 形式化验证

这是最精确的一种方法，安全操作系统被简化为一个要证明的“定理”，定理断言此安全操作系统是正确的，但完成证明所需的工作量巨大，尤其对于大型实用系统，试图描述和验证均十分困难。

(2) 非形式化确认

非形式化确认包括验证，也包括一些不太严格的测试程序正确性的方法，确认方法有：安全需求检查、设计及代码检查和模块及系统测试。

(3) 入侵测试

入侵者应当掌握操作系统典型的安全漏洞，并试图发现和利用系统中的安全缺陷。安全操作系统在一次入侵测试中失效，说明其内部有错；然而，安全操作系统在某次入侵测试中未失效，却并不能保证系统中不存在错误，入侵测试在确定错误存在方面是很有用的。

一般来说，评价计算机系统安全性的高低，应从两方面进行：一是安全功能，系统提供哪些安全功能；二是可信性，安全功能在系统中得以实现的可信任程度，通常通过文档规范、系统测试、形式化验证等安全保证来说明。

3. 操作系统安全测评准则

美国国防部于 1983 年推出历史上第一个计算机系统安全评测准则 TCSEC (Trusted

Computer System Evaluation Criteria), 又称橘皮书, 从而带动国际计算机系统安全评测的研究, 德国、英国、加拿大等国纷纷制定各自的计算机系统安全评价标准。近年来, 我国制定了 GB17859-1999 和 GB/T18336-2001 等国家标准。

下面介绍 TCSEC 的内容, 对可信任的计算机信息系统有 6 项基本需求, 其中, 4 项涉及存取控制, 两项涉及安全保障: 需求 1“安全策略”; 需求 2“标记”; 需求 3“标识”; 需求 4“审计”; 需求 5“保证”; 需求 6“连续保护”。

根据上述 6 项基本需求, TCSEC 在用户登录、授权管理、访问控制、审计跟踪、隐蔽信道分析、可信通路建立、安全检测、生命周期保证、文档写作等各方面均提出规范性要求, 并根据系统所采用的安全策略和所具备的安全功能把系统分为 4 类共 7 个安全等级。

- (1) D 类:D 级(仅一个级别), 其安全性最低级, 整个系统不可信任。
- (2) C 类:自主保护类
 - ① C1 级:自主安全保护。
 - ② C2 级:受控制的存取控制系统, 引入 DAC 和审计机制。
- (3) B 类:强制保护类
 - ① B1 级:标记安全保护级, 引入 MAC、标记和标记管理。
 - ② B2 级:结构保护级, 具有形式化安全模型、完善的 MAC、可信通路、系统结构化设计、最小特权管理、隐蔽信道分析。
 - ③ B3 级:安全域级, 访问监控机制、TCB 最小复杂性设计、审计实时报告和对硬件的要求。
- (4) A 类:A1 级(仅一个级别), 验证保护类, 严格的设计、控制和验证过程。

7.4 Linux 安全机制

1. Linux 基本安全机制

(1) 标识与鉴别

为用户创建账户时, 系统为其分配唯一的用户号(UID)和用户组号(GID), 并为用户建立主目录, 每个用户可属于一个或多个用户组。系统使用 DES 或 MD5 算法对用户口令(密码)进行加密后, 将其存储在 /etc/shadow 中, 用户登录时, 需要输入口令, 并与口令密文进行比较, 鉴别用户的真实身份。当用户设置或修改口令时, 如果所输入的口令安全性不够, 系统会发出警告。系统管理员还可以设置口令的最小长度、一次性口令及口令的有效期限。

(2) 存取控制

Linux 系统实现粗粒度的自主访问控制, 用户可为自己的文件设置和修改访问权限; 其类型有三种: 读、写和执行; 授权对象有三类: 文件属主、用户组和其他用户。粗粒度的自主访问控制不能满足许多应用系统的安全要求, 为此, 开发 Linux 访问控制表, 以提供更完善的文件授权设置, 可将存取控制细化到单个用户。

系统中的进程都有真实 UID、真实 GID、有效 UID 及有效 GID。进程的真实 UID 和 GID 是

创建此进程的用户的 UID 和 GID, 表示进程隶属于谁; 有效 UID 和 GID 标识进程的主体身份, 当进程试图访问文件时, 核心将进程的有效 UID、GID 与文件的存取权限域中的相应值进行比较, 决定是否向其赋予访问权限。在一般情况下, 进程的真实 UID、GID 就是进程的有效 UID、GID。

有时需要让未被授权的用户完成某些要求授权的任务, 例如, 允许普通用户使用 password 程序来改变自己的口令, 但不能拥有“写”/etc/password 文件的权限, 以防止它修改其他用户的口令。为此, Linux 系统允许对可执行文件设置 SUID(或 SGID)标志, 当进程执行带有 SUID(或 SGID)标志的文件时, 进程的有效 UID(或 GID)被改为执行文件属主的 UID(或 GID), 于是, 进程就拥有执行文件属主所拥有的存取权限。

(3) 审计

Linux 所实现的审计功能包含丰富的审计内容, 遍及系统层、应用层和网络协议层, 能全面地监控系统中所发生的事件, 并及时地对系统异常报警提示。大部分日志文件存放在 /var/log 目录下, 审计服务程序 syslogd 专门负责审计信息的存储, 系统和内核将需要记录的信息发送给 syslogd, 它根据配置(/etc/syslog.conf), 按照信息的来源和重要性, 将其记录到不同的日志文件、输出到指定的设备或发送到其他主机中。

(4) 特权管理

超级用户 (UID=0) 拥有所有特权, 负责系统的配置和管理, 控制用户账户、文件、目录和网络等资源, 因此, 超级账户及其口令的管理至关重要。

从 Linux 2.1 版开始, 引入权能的概念, 实现基于权能的特权管理机制, 已实现 7 个 POSIX1003.1e 规定的权能, 还有 22 个特有权限。

(5) 网络安全

为了支持网络计算环境, 提供多种网络操作命令, 如远程登录命令 (telnet、rlogin)、远程文件复制 (ftp、rcp)、远程执行命令 (rsh、rcmd), 这些命令的执行结果都是对远程计算机的访问, 或是使用远程主机的资源, 或请求远程服务器的服务。要防止利用这些服务进行入侵的行为, 所采用的措施是对远程用户的访问进行有效控制, 仅对特定的用户开放必要的网络服务, 减少本机系统受攻击的可能性。

使用超级守护进程 xinetd 实现网络服务的访问控制, 负责监听在 inetd.conf 配置文件中登记的网络端口, 启动被请求的服务程序; 使用网络过滤工具对 inetd 所启动的网络服务进行访问控制并记录访问请求, 还安装 OpenSSH 和 OpenSSL 程序包实现网络安全通信。

(6) 其他安全机制

① 加密: 提供点对点的加密方法, 保护传输中的数据。利用加密方法加强 shell 功能, 保证远程登录的安全, 也可对本地文件进行加密以防止文件被非法访问, 同时保证文件的一致性, 防止对文件的非法篡改, 并在一定程度上防止病毒等恶意程序的攻击。

② 备份和恢复: 使用系统备份加强系统的安全性和可靠性, 这是一种加强措施, 当系统发生崩溃后将系统恢复到一个稳定的状态。备份类型有三种: 实时备份、整体备份、增量备份, 备份文

档经过处理(压缩、加密等)后保存,系统中有专门的备份程序,如 dump/restore、backup;网络备份程序有 rdump/restore、rcp、ftp、rdist 等。

2. 安全操作系统 SELinux

(1) 安全体系结构

2001 年,美国国家安全局(National Security Agency, NSA)发布安全增强 Linux—SELinux,系统的安全体系基于 Flask 结构,Flask 也是由 NSA 开发的安全体系结构。目前,已经在 Linux 内核的主要子系统中实现包括进程、文件和套接字等操作的强制控制机制。

SELinux 安全体系结构如图 7.15 所示,由两部分组成:策略(policy)和实施(enforcement),策略封装在内核于系统安全服务器中,实施由内核对象管理器具体执行,主要特点是通过安全判定与安全实施的分离实现安全策略的独立性。

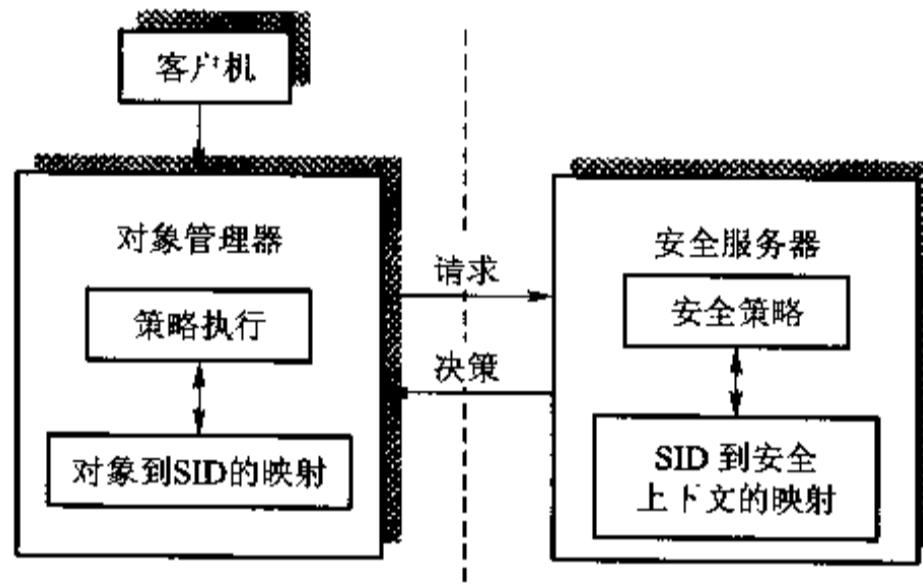


图 7.15 SELinux 安全体系结构

当需要对安全性进行判断时,向安全服务器提出请求,对象管理器只关心 SID。请求到达安全服务器后,实现与安全上下文的映射并进行计算。然后,将决策返回给对象管理器。系统中关于安全的请求和决策有以下三种情况。

① 标签确定(labeling decision):确定一个新的主体和客体采用何种安全标签(如创建主体或客体)。

② 存取决策(access decision):确定主体是否能访问客体的某种服务(如文件读写)。

③ 多实例决策(polyinstantiation decision):描述一个特定的请求应该访问多实例化资源中的哪一个成员。例如,TCP 和 UDP 的端口空间资源是固定的,多实例支持用来在端口与 socket 绑定时,进入包的目的端口号存在于多个成员端口空间时,决定采用哪个端口空间。

SELinux 安全服务器实现一种混合型安全策略,包括类型实施(Type Enforcement, TE)、基于角色的访问控制(Role-Based Access Control, RBAC)和可选的多级别安全性(optional MultiLevel Security, MLS)。

Flask 结构还提供一个访问向量缓存(Access Vector Cache, AVC)模块,允许对象管理器缓存访问向量,减少整体性能损失,并实现对动态安全策略的支持。每次做安全性检查时,系统首先

查找放在 AVC 中的访问向量,若找不到才向安全服务器提出查询请求。Flask 有两个用于安全性标签但与安全策略无关的数据类型:安全性上下文(security context)和安全性标识(SID),前者是表示安全性标签的变长字符串,由用户、角色、类型和可选的 MLS 等组成;后者是由安全服务器映射到安全上下文的一个整数,SID 作为实际上下文的句柄服务于系统。一般来说,对象管理器根据主体和客体的 SID 以及对象类来查询安全服务器,其目的是获得访问决策——访问向量。对象类用以标识对象的类型,如表示常规文件、目录、进程、套接字等的整数。

(2) 安全策略配置

SELinux 中每个主体都有一个域(domain),每个客体都有一个类型(type),策略的配置决定对类型的存取是否被允许,及一个域能否转到另一个域。类型的概念用于应用程序时,可以决定类型是否可由域执行。角色在配置中进行定义,每个进程都有一个与其相关的角色,如系统进程是一种角色,应用进程是另一种角色。

安全策略配置的目标包括:控制对数据的原始访问、保护内核和系统软件的完整性、防止特权进程执行危险代码、限制由特权进程缺陷所导致的伤害、防止未通过身份鉴别就进入管理员角色或域、防止普通进程干扰系统进程或管理员进程等。策略可根据策略文件灵活生成,客体的类型有:设备、文件、网络文件、网络等;主体域的策略定义有:管理、系统、用户等。

3. Linux 安全模块

作为对 NSA 发布 SELinux 的反应,Linux 创始人 Linus Torvalds 描述其所考虑的包含主流 Linux 内核的安全框架。此安全框架必须是:

- (1) 真正通用,使用不同的安全模型仅加载不同的核心模块;
- (2) 概念简单,最小的扩散,有效;
- (3) 能够作为一个可选的安全模块,支持现有的 POSIX.1e 权能机制。

这个通用安全框架将提供一组安全“钩子”(hooks),控制对核心客体的操作,提供核心数据结构中的一组不透明的安全域来维护安全属性。此外,这个框架也可用做可加载核心模块,通过这种方式在系统中实现任何所需要的安全模型。

Linux 安全模块(LSM)项目就是要开发这样一个框架,LSM 为主流 Linux 核心开发一个轻量级的、通用目的的存取控制框架,使得很多不同的存取控制模型可以作为可加载模块来实现。

LSM 通过在内核源代码中放置钩子,来仲裁对内核内部对象所进行的访问。这些对象包括任务、inode、打开的文件等。应用进程执行系统调用时首先遍历 Linux 内核的原有逻辑,找到并分配资源,进行错误检查,并经过经典的 9 bit 自主访问控制,恰好在 Linux 内核试图对内部对象进行访问之前,一个 LSM 的钩子对安全模块提供函数进行调用,以决定是否允许执行访问。LSM 根据安全策略做出回答,如果被拒绝,则返回一个错误码。

目前 LSM 是以 Linux 内核补丁的形式实现的,主要对 Linux 内核进行以下修改:在特定的内核数据结构中加入安全域,在内核代码中的管理域和实现存取控制的关键点处插入对钩子函数的调用,加入通用的安全系统调用,提供函数以允许内核模块注册为安全模块和注销安全模块,将权能机制移植为可选的安全模块。

7.7 Windows 2003 安全机制

Windows 提供一组可配置的安全性服务和灵活的访问控制能力,能够满足分布式系统的安全性需求,主要的服务有:安全登录、访问控制、安全审计、主存保护、活动目录、Kerberos 5 身份验证协议、基于 Secure Sockets Layer 3.0 的安全通道、EFS 加密文件系统。

7.7.1 安全性组件和安全登录

1. 安全性组件

实现 Windows 安全性机制的组件和数据库如下。

(1) 安全引用监视器(SRM):是执行体的一个组件,针对对象负责执行安全访问检查、管理对象的访问权限和产生安全审计消息。

(2) 本地安全权限(LSA)服务器:是运行 LSASS.exe 的用户态进程,负责本地系统的安全性规则(例如,允许用户登录的规则、密码规则、授予用户和组的权限列表以及系统安全性审计设置)、用户身份验证以及向“事件日志”发送安全性审计消息。

(3) LSA 策略数据库:是一个包含系统安全性规则的数据库,保存在注册表的 HKEY - LOCAL - MACHINE/security 下,所包含的信息有:哪些域被信任用于认证登录企图;哪些用户可以访问系统以及如何访问(交互、网络和服务登录方式);对何人赋予哪些权限;所执行的安全性审计种类。

(4) 安全账号管理服务器:是一组负责管理数据库的子例程,此数据库包含定义于本地机或用于域(如果系统是域控制器)的用户名和组,SAM 在 LSASS 进程的描述表中运行。

(5) SAM 数据库:是一个包含用户和组定义及其密码、属性的数据库,此数据库被保存在 HKEY - LOCAL - MACHINE/SAM 下的注册表中。

(6) 默认身份认证包:是一个称为 MSV1_0 的动态链接库,在进行身份验证的 LSASS 进程的描述表中运行,这个 DLL 负责检查给定的用户名和密码是否和 SAM 数据库中所指定的内容相匹配。如果匹配,则返回用户信息。

(7) 登录进程:是一个运行 WinLogon.exe 的用户态进程,负责搜寻用户名和密码,并发送给 LSA 用以验证它们,然后在用户会话中创建初始化进程。

(8) 网络登录服务:是一个响应网络登录请求的 services.exe 进程内部的用户态服务,身份验证同本地登录一样,通过发送到 LSASS 进程来验证。

2. 安全登录

登录是通过登录进程 WinLogon、LSA、身份认证包和 SAM 相互作用完成的,在系统初始化过程中,在激活任何应用程序之前,WinLogon 将执行特定的步骤以确保一旦系统为用户做好准备,就能够控制工作站。当用户按下热键时,就开始登录,在获得用户名和口令之后,由 LSA 本地安全认证服务器、msv1_0(网络中用 Kerberos)身份验证软件包、SAM 数据库协同工作,进行

标识和鉴别,msv1_0 查看所请求的登录与 SAM 数据库中允许的访问是否有匹配项。如果是合法用户,LSA 会汇集各种用户信息,如用户 SID、组 SID、主目录等配置文件信息,这时,为用户创建应用进程。为了实现对它的访问控制,本地安全认证子系统同时创建两个重要的管理实体:与进程相关联的“访问令牌”和与对象相关联的“安全描述符”。

7.7.2 访问控制

1. 保护对象

保护对象是谨慎访问控制和审计的基本要素,被保护的对象包括文件、设备、邮件槽、管道、进程、线程、事件、互斥体、信号量、定时器、访问令牌、窗口、桌面、网络共享、服务、注册表和打印机。

被导出至用户态的系统资源是作为对象来实现的,因此,对象管理器就成为执行安全访问检查的关键。要控制访问对象者,安全系统就必须知道每个用户的标识,因为在访问任何资源之前都要进行身份验证。当一个线程打开某对象的句柄时,对象管理器和安全机制就会使用调用者的安全标识来决定是否将所申请的句柄授予调用者。

2. 访问控制方案

访问令牌是一个包含进程或线程安全标识的数据结构,包括:安全 ID(SID);用户所属组的列表及启用/禁止的特权列表,它是基于安全的目的,在系统中所使用的此用户的唯一标识符;当应用进程创建新进程时,新进程对象继承同一个访问令牌。访问令牌有以下两种用途。

(1) 负责协调所有安全信息,加速访问确认,当与一个用户相关联的任何进程试图访问时,安全子系统使用与此进程相关联的访问令牌来确定用户的访问权限。

(2) 允许进程以受限方式修改自己的安全特性,而不会影响代表用户运行的其他进程。

访问令牌指明一个用户所拥有的特权,标记通常被初始化成每种特权都处于禁止状态。随后,如果应用进程中的某个进程需要执行一个授权操作,则此进程可以被赋予适当的特权并试图访问。之所以不希望在系统范围内保留一个用户的所有安全信息,因为这样做会导致只要允许某个进程的一项特权,就等于允许所有进程拥有这项特权。

与每个对象相关联,且使得进程间的访问成为可能的是安全描述符,其主要组件是访问控制表,此表为一个对象确定各个用户和用户组的访问权限。当一个进程试图访问一个对象时,将此进程的 SID 与这个对象的访问控制表内容进行比较,确定本次访问是否被允许。当应用程序打开对一个可访问对象的引用时,系统验证此对象的安全描述符是否同意这个应用程序的用户访问。如果检测成功,系统将缓存所允许的访问权限。

Windows 安全机制的一个重要特征是代理的概念,它可以在客户-服务器环境中简化安全机制的使用。如果客户和服务器通过 RPC 连接进行对话,服务器就可以临时代理此客户的身份,以此客户的权限去评估一个访问请求。在访问结束之后,服务器恢复自己的身份。

3. 访问令牌

访问令牌是一个包含进程或线程安全标识的数据结构,图 7.16(a)给出访问令牌的结构,它

包括以下参数。

- (1) 安全 ID: 在网络中的所有机器上唯一地确定一个用户, 通常对应于用户登录名。
- (2) 组 SID: 关于表示用户属于哪些组的列表, 组是指一组用户 ID 基于访问控制的目的被标识为一个组, 每一组都有唯一的组 SID。对于对象的访问可基于组 SID、个人 SID 或其组合来定义。
- (3) 特权: 是指用户可调用的一组对安全性敏感的系统服务。一个例子是创建令牌, 另一个例子是设置备份特权, 具有这个特权的用户允许使用备份工具对其通常不能读的文件进行备份, 大多数用户没有特权。
- (4) 默认所有者: 如果进程创建另一个对象, 则“默认所有者”域确定新对象的所有者。通常, 新进程的所有者与派生它的进程的所有者相同。但是, 用户可指定由此进程派生的任何进程的默认所有者是此用户所属的组 SID。
- (5) 默认 ACL: 这是用于保护用户所创建的所有对象的初始表, 用户可为其拥有的或所在的组所拥有的任何对象修改 ACL。

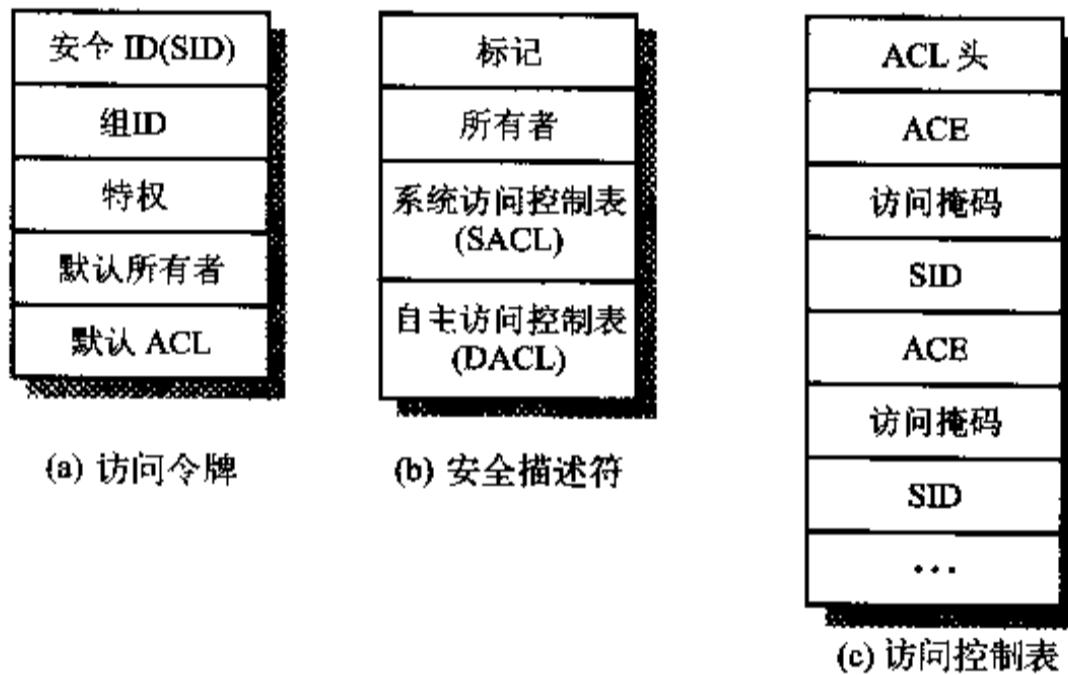


图 7.16 Windows 的安全结构

4. 安全描述符

系统中的所有对象在被创建时都应指定安全描述符, 图 7.16(b)给出安全描述符的一般结构, 主要参数有:

- (1) 标记: 定义安全描述符的类型和内容。此标记指明是否存在 SACL 和 DACL, 它们是否通过默认机制被放置在对象中, 及描述符中的指针所使用的是绝对寻址还是相对寻址。在网络上所传送的对象, 如在 RPC 中传送的信息, 也需要相关的描述符。
- (2) 所有者: 对象的所有者可以在安全描述符上执行任何动作。所有者可以是单一 SID, 也可以是组 SID, 所有者具有改变 DACL 内容的特权。
- (3) 系统访问控制表(SACL): 确定哪些用户的哪些操作应登录到安全审核日志中, 应用程序必须在其访问令牌中具有相应的特权才可以读或写对象的 SACL, 这是为了防止未授权的应

用程序读 SACL(从而知道为了规避审核信息而不做什么)或者写 SACL(产生大量的审核信息,从而导致忽略违法操作)。

(4) 自主访问控制表(DACL):确定有哪些用户和组以何种权限访问对象。DACL 由一组访问控制项(ACE)组成。

当创建一个对象时,创建进程可以把此进程的所有者指定为自己的 SID 或者其访问令牌中的任何组 SID,创建进程不能指定一个不在当前访问令牌中的 SID 作为进程的所有者。随后,任何被授权改变一个对象的所有者的进程都可以这样做,但是也有同样的限制。使用这种限制的原因是防止用户在试图进行某些未授权的操作后隐蔽自己的踪迹。

访问控制表是 Windows 访问控制机制的核心,如图 7.16(c)所示。每个表由 ACL 头和许多 ACE(访问控制项)组成,每项均定义个人 SID 或组 SID,访问掩码定义此 SID 被授予的权限。访问控制表可包括 0 个或多个访问控制项 ACE 结构。具有 0 个 ACE 的 ACL 被称为空 ACL 表,表示没有用户可以访问此对象。DACL 中可能存在两种 ACE:“访问允许”和“访问拒绝”,“访问允许”ACE 向用户授予访问权,而“访问拒绝”ACE 拒绝访问掩码中所指定的访问权力。由各个 ACE 所授予的访问权限的集合构成由 ACE 授予的一组访问权限。如果安全描述符中没有 DACL,则每个用户就拥有对象的完全访问权;另一方面,如果 DACL 为空,即 0 个 ACE,就没有用户可访问的对象。系统 ACL 仅包含一种 ACE,称为审计 ACE,用来指明特定用户或组在对象上所进行的应得到审计的操作,成功或不成功的操作均可被审计。如果系统 ACL 为空,则对象不会被审计。

当进程试图访问对象时,对象管理器从访问令牌中读取 SID 和组 SID,然后,扫描此对象的 DACL。如果发现匹配项,即找到一个 ACE,其 SID 与访问令牌中的 SID 相匹配,那么,此进程具有这一 ACE 的访问掩码所确定的访问权限。

5. ACL 的分配

要确定分配给新对象的 ACL,系统使用三种互斥规则之一,具体的步骤如下。

步骤 1:如果调用者在创建对象时,明确地提供一个安全描述符,则系统将把此描述符应用到对象上。

步骤 2:如果调用者未提供安全描述符,而对象拥有名称,则系统将在存储新对象名称的目录中查看安全描述符。一些对象目录的 ACE 可以被指定为可继承的,表示其可应用于在对象目录中所创建的新对象上。如果存在可继承的 ACE,系统就将其编入 ACL,并与新对象连接。

步骤 3:如果上述情况均未出现,系统会从调用者访问令牌中检索默认 ACL,并将其应用于新对象。系统的一些子系统,如服务、LSA 和 SAM 对象等,均有其创建对象时所分配的硬性编码 DACL。

6. 访问控制算法

有两种类似的访问控制算法,第一种算法确定允许访问对象的最大权限,第二种算法确定是否允许一个针对对象的有效访问。这两种方法是类似的,下面介绍其中一种。

依据调用者的访问令牌来确定是否授予所申请的访问权限,其步骤为:

- (1) 如果对象没有 DACL, 对象就得不到保护, 系统将授予所希望的访问权限;
- (2) 如果调用者具有所有权特权, 系统将在检查 DACL 之前授予其写入访问权限;
- (3) 如果调用者是对象的所有者, 则被授予读取控制和写入 DACL 的访问权限;
- (4) 检查 DACL 中的每个 ACE, 如果 ACE 中的 SID 与调用者访问令牌中的“启用”SID 相匹配, 则处理 ACE。如果是一个访问拒绝 ACE, 所申请的任何访问权限都在拒绝访问的范围内, 则对对象的访问被拒绝;
- (5) 如果 DACL 检查完毕, 而一些被请求的访问权限未被授予, 则访问被拒绝。

算法的访问有效性依赖于访问拒绝的 ACE 被置于访问允许的 ACE 之前。在 Windows 中, 由于引入对象所指定的 ACE 且自动继承, 所以, ACE 的顺序变得更加复杂, 非继承的 ACE 要置于继承的 ACE 之前。

7. 访问掩码

图 7.17 给出访问掩码的内容, 具体的访问位共有 16 位, 确定适用于某特定类型的对象的访问权限。例如, 文件对象的第 0 位是 file _ read _ data 访问, 事件对象的第 0 位是 event _ query _ status 访问。

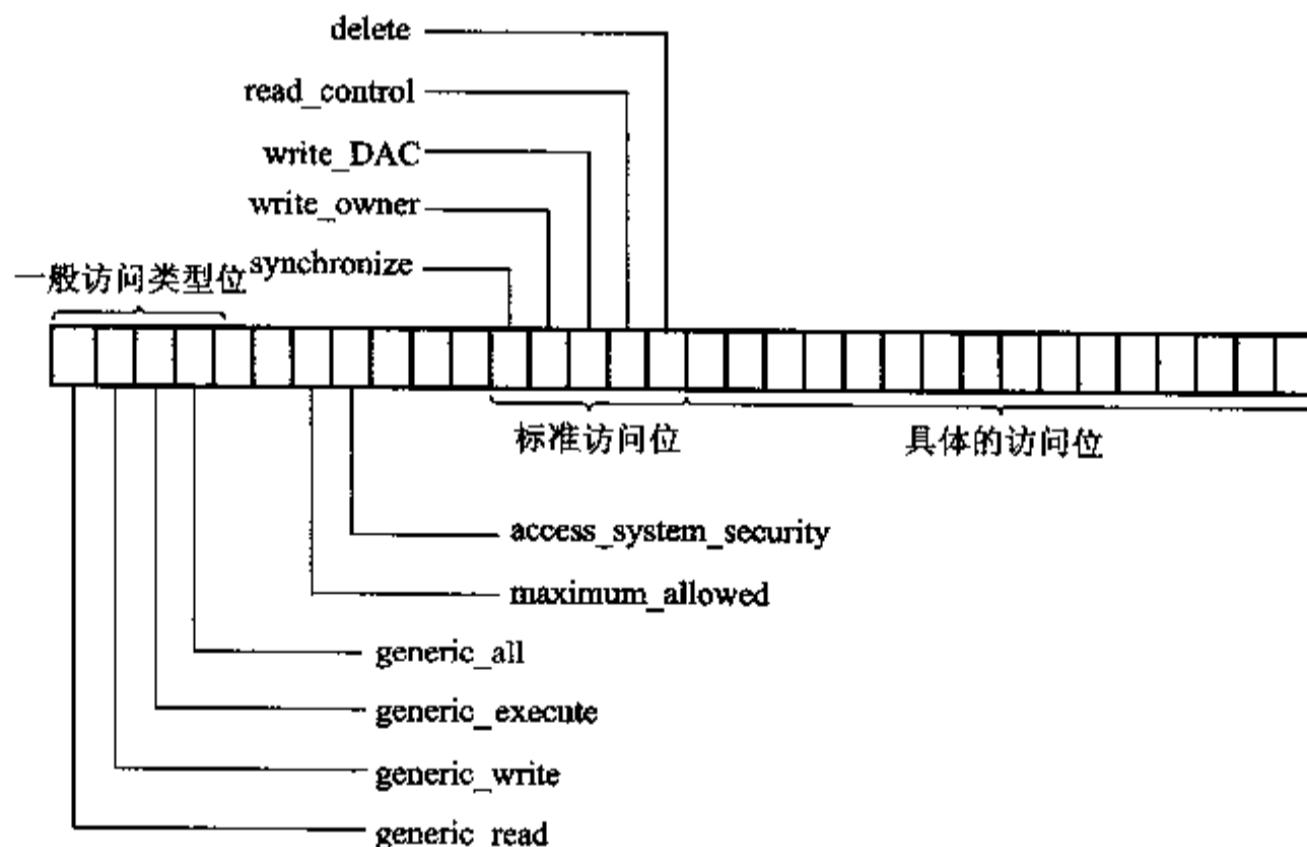


图 7.17 访问掩码

掩码中最重要的 16 位包含适用于所有类型的对象的位, 其中 5 位称为标准访问位。

- (1) synchronize: 允许与对象相关联的某些事件同步执行。特别地, 此对象可用于等待函数。
- (2) write _ owner: 允许程序修改对象的所有者。这一点非常有用, 因为对象的所有者经常会改变对此对象的保护(所有者不会被拒绝写 DAC 的访问)。
- (3) write _ DAC: 允许应用程序修改 DACL, 因而可以修改在对象上的保护。
- (4) read _ control: 允许应用程序查询所有者和对象安全描述符的 DACL 域。

(5) delete: 允许应用程序删除这个对象。

访问掩码的高端包含 4 个一般访问类型位。这些位为在不同的对象类型中设置具体的访问类型提供一种方便的途径。例如,假如一个应用程序希望创建几种类型的对象,并且确保用户可对这些对象进行读访问。为了保护不具备一般访问类型位的每种类型的每个对象,应用程序必须为各类对象构造一个不同的 ACE,并且在创建对象时,小心地传递正确的 ACE。而创建表示读许可的 ACE,并且把这个 ACE 应用于所创建的每个对象,比前面的方法要方便得多,这就是设置一般访问类型位的目的。访问类型位包括:

- (1) generic_all: 允许所有访问。
- (2) generic_execute: 如果是可执行的,则允许执行。
- (3) generic_write: 允许只写访问。
- (4) generic_read: 允许只读访问。

一般访问类型位还反映标准访问类型。例如,对于一个文件对象,generic_read 可以映射到标准位 read_control 和 synchronize,且可以映射到具体的对象位 file_read_data、file_read_attribute 和 file_read_EA。因此,把一个 ACE 放置在一个给某些 SID 赋予 generic_read 权限的文件对象上,就等同于给这些 SID 赋予上述 5 种访问权限,并且好像它们是在访问掩码中所单独定义的一样。

访问掩码中的剩余两位也有具体的含义。access_system_security 允许为对象修改审计和警告控制,不仅一个 SID 的 ACE 中的这一位必须设置,并且此 SID 的进程的访问令牌也必须允许相应的特权。最后,maximum_allowed 允许对象的所有者定义赋予某个给定用户的一组最大访问权限。

7.7.3 安全审计

在 Windows 系统中,对象管理器可将访问检查的结果生成审计事件,获得特权后,用户使用 Win32 函数也可直接生成审计事件。本地安全规则所调用的审计规则是 LSA 所维护的安全规则的一部分,LSA 负责接收审计记录(小记录通过发送消息的方式或大记录借助于共享主存传递),对它们进行编辑并把记录发送至“事件记录器”,事件记录器把审计记录写入“安全日志”,LSA 也能够产生直接发送至“事件记录器”的审计记录。

Windows 系统产生三类日志:系统日志、应用程序日志和安全日志。审计事件分为 7 类:系统类、登录类、对象存取类、特权应用类、账号管理类、安全策略类和详细审计类。对于每类事件可以选择成功、失败或两者同时审计。对于对象存取类的审计,还可以在资源管理器中进一步指定文件和目录的具体审计标准,如读、写、修改、删除、执行等操作,这些操作均分为成功或失败两种审计;对于注册表项及打印机设备的审计与此类似。审计数据以二进制结构文件存于磁盘,每条记录包括:事件发生时间、事件源、事件号及事件所属类别、机器名、用户名和事件的详细叙述。

可以使用事件查看器浏览和依据条件过滤显示,任何人都可以查看系统日志和应用程序日志;安全日志对应于审计数据,只能由审计管理员查看和管理,前提是必须存放于 NTFS 文件系

统中,使得访问控制 SACL 生效。

7.7.4 加密文件系统

加密文件系统(Encrypted File System, EFS)把 NTFS 文件中的数据进行加密,并存储在磁盘上。EFS 的加密技术是基于公共密钥的,它用随机产生的文件密钥(File Encryption Key, FEK)通过加强型数据加密标准 DES 算法对文件进行加密。EFS 的加密技术的特点是集成服务,易于管理,不易遭受攻击。

EFS 通过基于 RSA 的公共密钥加密算法利用 FEK 进行加密,并将其和文件存储在一起,形成文件的 EFS 属性字段:数据解密字段(Data Decryption Field, DDF)。在解密时,用户使用私钥解密存储于文件 DDF 中的 FEK,再用所得到的 FEK 对文件进程解密,最后得到文件原文。只有文件拥有者和管理员掌握解密用的私钥,任何人都可以得到加密的公钥,但由于没有私钥,无法破解加密文件。EFS 的体系结构如图 7.18 所示。

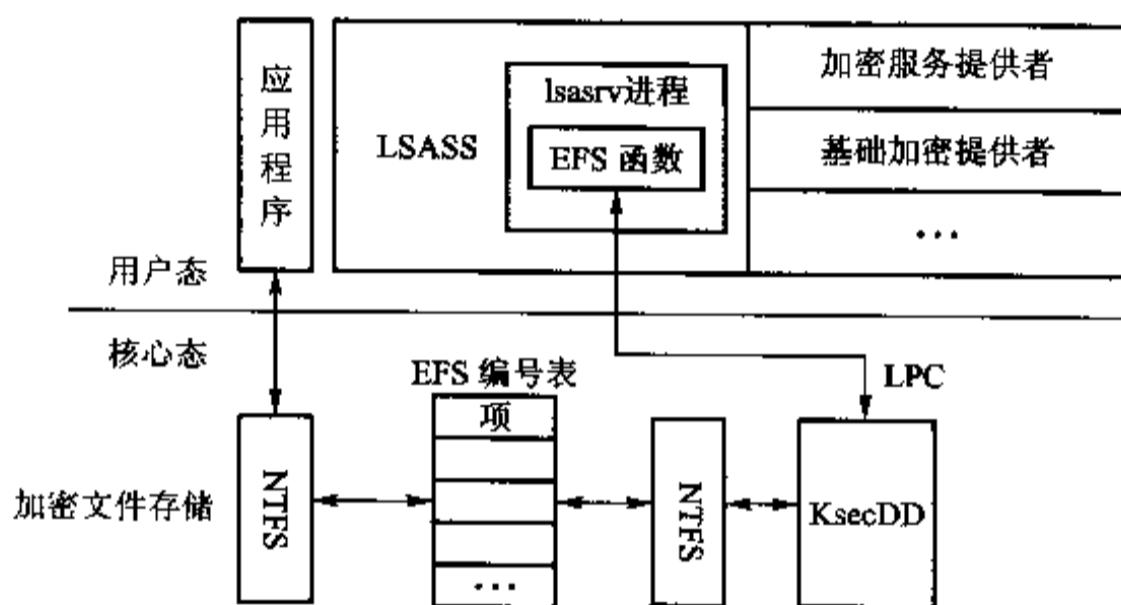


图 7.18 EFS 体系结构

EFS 的实现类似于在核心态运行的设备驱动程序,与 NTFS 密切相关。当用户态应用程序要访问加密文件时,它向 NTFS 发出访问请求,NTFS 收到请求之后执行 EFS 驱动程序,EFS 通过 KsecDD 设备驱动程序转发 LPC(Local Procedure Call,本地过程调用)给 LSASS(Local Security Authority Subsystem Service,本地安全授权子系统服务)。LSASS 不仅处理登录事务,还管理 EFS 密钥,其组成部分 lsasrv 进程则侦听这一请求,并执行所包含的 EFS 函数,在处于用户态的加密服务 API(CryptoAPI)的帮助下,进行文件的加密和解密。

Windows 通过命令 Cipher.exe 或目录安全选项来加密文件,当 lsasrv 进程收到加密文件 LPC 信息后,采用特殊技术压缩用户文件,在文件所在卷的系统卷信息目录下创建相关的日志文件记录,调用“微软基础加密提供者”为文件产生一个基于 RSA 的 FEK,再构建 EFS 信息并把它作为文件属性同加密文件存储在一起。最后的步骤是加密和保存数据文件,工作流程如图 7.19 所示。

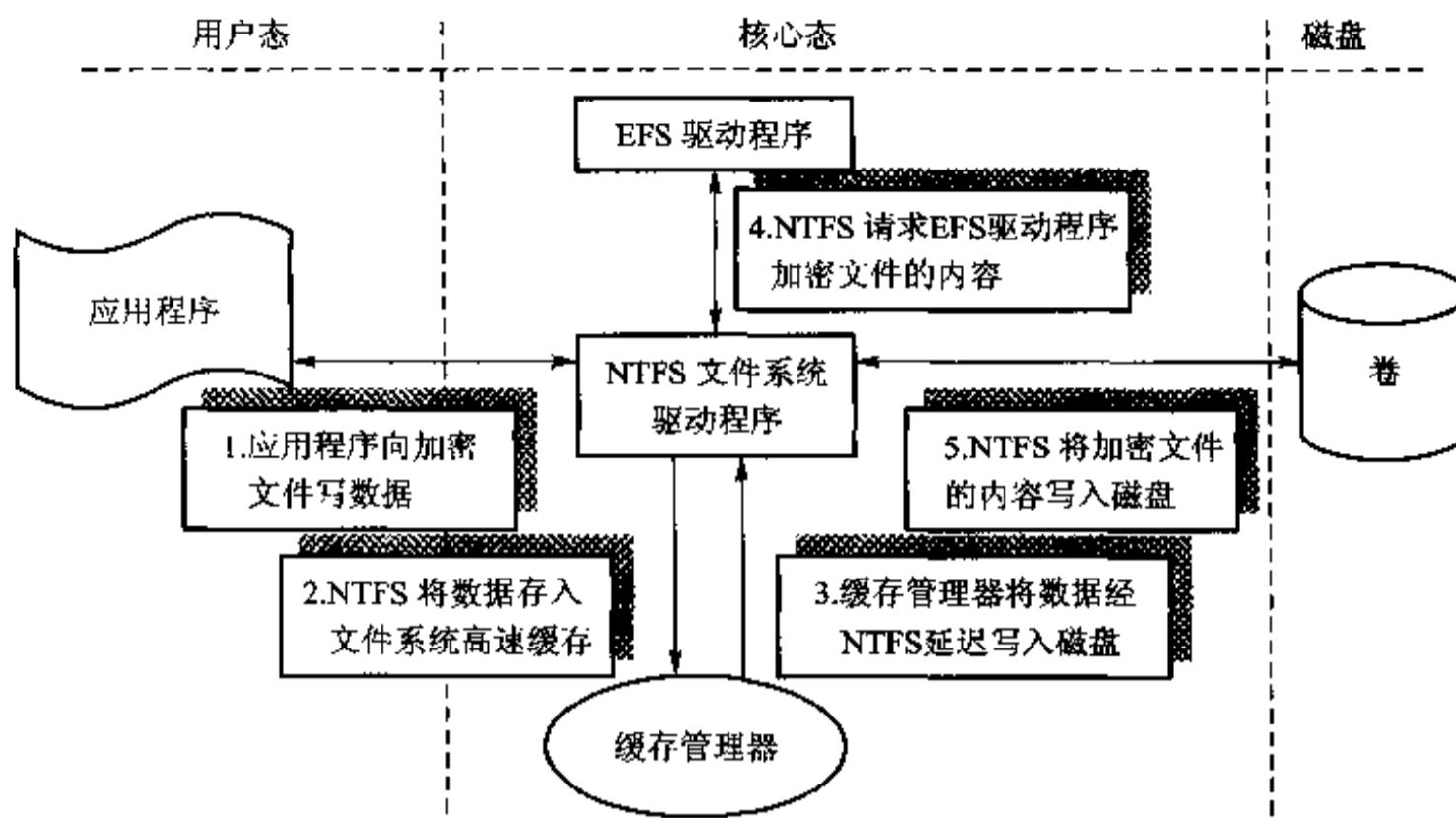


图 7.19 EFS 工作流程

解密过程比较简单, lsasrv 进程取得 EFS 的 FEK 之后, 通过 LPC 返回 EFS 驱动程序。然后, EFS 利用 FEK 通过 DES 算法进行文件的解密运算, 并通过 NTFS 把结果返回给用户。

7.9 本章小结

计算机信息系统的安全性涉及:管理和实体的安全性、网络通信的安全性、软件系统的安全性和数据库的安全性。软件系统中最重要的是操作系统, 其安全性是系统安全性的基础, 操作系统安全性内容包括:安全策略、安全模型、安全机制。

安全策略分为访问支持策略和访问控制策略, 前者用于验证用户的合法身份, 将非法人员拒于系统之外;后者确保只有授权用户才能访问特定的资源, 以满足系统机密性、完整性、可记账性和可用性等安全需求。

安全模型是对安全策略所表达的安全需求的一种精确、无歧义的抽象描述, 为安全机制的实现提供框架。目前, 大都采用状态机安全模型, 分为:基于访问控制矩阵模型(Lampson 模型、Graham - Denning 模型、Harrison - Ruzzo - Ullman 模型)和基于格的安全模型(Bell - LaPadula 模型、D. Denning 信息流模型)。多级安全系统中应用最广泛的是 BLP 机密性安全模型。

优秀的硬件保护设施是实现高效、安全、可靠操作系统的基础, 硬件安全机制有:主存保护和运行保护, 软件安全机制有:认证机制、授权机制、加密机制、审计机制。认证机制是确定主体是否是其所声称身份的过程, 以确保登录者就是使用这个账户的合法用户;授权机制提供访问控制技术, 确认主体只有在安全策略许可时才能访问客体, 两类访问控制技术是:自主访问控制机制(权限表、前缀表、口令表、ACL)和强制访问控制机制(信息分密级和范畴管理);加密机制采用某

种方式伪装信息来隐藏其内容,以提高信息系统及数据的安全性和保密性,防止信息被窃取和泄密;审计机制用于对系统中有关安全的活动进行完整记录、检查和审核,是对系统安全性所实施的一种技术措施。由于针对计算机的大多数访问都通过网络进行,因而,访问控制机制必须设计成能够在分布式网络环境中有效工作的模式。本章还介绍了特殊的授权机制——最小特权原理及其实现。

开发安全操作系统有两条途径:初始设计和现有系统的加固,本章从一般性的角度出发,讨论安全操作系统的设计原则和结构、开发方法和设计技术,对信息系统安全评价标准作了简要介绍。

习题七

一、思考题

1. 试述计算机系统的可靠性和安全性之间的联系和区别。
2. 试述操作系统安全性的主要内容。
3. 试述计算机安全需求的主要内容。计算机安全需求与安全策略有何关系?
4. 何谓安全策略? 试按照应用和功能对其进行分类。
5. 计算机系统或网络系统在安全性上所受到的攻击类型有哪些? 试述其主要内容。
6. 计算机系统的资源分为硬件、软件、数据以及通信线路和网络等几种,试述每种资源类型所面临的威胁。
7. 为什么说操作系统安全是整个计算机系统安全的基础?
8. 什么是访问支持策略? 试述其主要内容。
9. 什么是用户的标识和鉴别? 两者有何不同。
10. 什么是客体重用? 如何确保重用客体的安全性。
11. 什么是隐蔽信道? 如何做隐蔽信道分析?
12. 试述可信路径和可信恢复。
13. 讨论访问控制中的主体、客体及其属性。
14. 讨论自主访问控制策略和强制访问控制策略。
15. 何谓安全模型? 试述其分类。
16. 什么是状态机模型? 开发状态机模型通常需要哪些步骤?
17. 基于访问控制矩阵的安全模型有哪些? 介绍其主要内容。
18. 试述 Bell-LaPadula 安全模型及其特点。
19. 试述计算机硬件安全机制的类型和作用。
20. 何谓 TCB? 试述其组成。
21. 试述单机用户身份认证和 Kerberos 网络身份认证的工作原理。
22. 当口令用做鉴别机制时,TCB 要做哪些工作?
23. 试述授权机制的功能和作用。
24. 试述自主访问控制的实现机制和原理。

25. 什么是访问控制矩阵、访问控制表和访问权能表?
26. 什么是许可标签、敏感性标签、安全级别和许可级别?
27. 什么是范畴? 举例说明之。
28. 试述强制访问控制机制的实现原理。
29. 试述多级保护系统中“不向上读”规则和“不向下写”规则的重要性。
30. 试述最小特权原理。
31. 试述 IBM 存储保护键的基本工作原理。存储保护键中的取保护位有何作用?
32. 试述操作系统中所使用的两态模式和环模式保护机制。
33. 什么是密文和明文?
34. 什么是加密、解密和密钥?
35. 密码系统为应用系统提供了哪些功能?
36. 试述对称算法和公开密钥算法。
37. 举例说明基本的加密与解密算法。
38. 计算机密码算法有哪些? 试详细讨论之。
39. 试述 DES 加解密的处理过程。
40. 试述 RSA 的工作原理。
41. 试述简单数字签名和保密数字签名。
42. 试述网络加密的分类和原理。
43. 试述审计、审计事件和审计日志。
44. 试述审计缓冲区的设计和管理。
45. 试述安全操作系统的结构和设计原则。
46. 试述可信软件、良性软件和恶意软件。
47. 试述安全操作系统的一般开发步骤。
48. 讨论安全操作系统开发的 3 种方法:虚拟机法、改进/增强法和仿真法。
49. 试述安全功能和安全保证。
50. 何谓隔离技术? 具体有哪些隔离技术?
51. 试比较多虚存空间与虚机隔离之间的异同。
52. 何谓安全内核? 设计安全内核的基本原则和基本方法有哪些?
53. 试述 MULTICS 的环结构安全机制。
54. 试述设计访问控制表所涉及的主要问题。
55. 何谓安全标签? 叙述其组成。
56. 如何基于安全标签来实现强制访问?
57. 试述 UNIX SVR 4.1 强制访问控制机制的实现要点。
58. 如何用强制访问控制防止“特洛伊木马”病毒?
59. 如何用 Linux“安全注意键”防止“特洛伊木马”病毒?
60. 试述最小特权机制的实现。
61. 试述审计缓冲区的工作原理,怎样协调对缓冲共享资源的使用?
62. 何谓审计点? 有哪些关键位置可作为审计点?

63. 何谓操作系统安全漏洞扫描？试述其主要内容。
64. 试述 TCSEC 的主要内容。
65. 试述 SELinux 的安全体系结构。
66. 试述 SELinux 的安全请求和决策的类型。
67. 试述 SELinux 的安全决策配置。
68. 试述 Linux 的安全模块。
69. Windows 中提供了哪些安全服务，其基本特征是什么？
70. Windows 的安全组件有哪些？试述其主要功能。
71. 试述 Windows 的安全控制方案。
72. 试述 Windows 访问令牌的结构及其功能。
73. 试述 Windows 安全描述符的结构及其功能。
74. 试述 Windows 访问掩码。
75. 试述 Windows 安全审计。
76. 试述 Windows 加密文件系统。

二、应用题

1. 有三种不同的保护机制：访问控制表、权能表和 UNIX/Linux 的 rwx 位。下面的问题分别适用于哪些机制？

- (1) Rick 希望除了 Jennifer 以外，任何人都可以读取他的文件；
- (2) Helen 和 Anna 希望共享某些秘密文件；
- (3) Cathy 希望公开她的一些文件。

对于 UNIX/Linux 系统，假设用户被分为教职工、学生、秘书等。

2. 考查下面的保护机制：为每个对象和进程赋予一个号码，规定仅当对象号大于进程号时，进程才可以存取对象。这种保护机制有什么特点？

3. 如果从英文字母表中选取 4 个字母形成一条口令，一个攻击者以每秒钟一条口令的速度试探口令，直到试探结束再将结果反馈给攻击者。那么，找到正确口令的时间是多长？

4. 考虑一个有 5 000 名用户的系统，假如只允许其中的 4 990 名用户存取某个文件。试问：如何实现？请给出更为有效的另一种保护方案。

5. 用户甲有 A1、A2 和 A3 三个私有文件，用户乙有 B1 和 B2 两个私有文件，而且这两名用户都需要使用共享文件 S。若文件系统对所有用户提供按名存取的功能，试画出能保证存取正确性的文件系统目录结构。

6. 在一个带有 4 个终端的计算机系统中，有 4 名学生上机实习，各自从终端输入程序和数据，并都保存在磁盘上，恰巧他们各自的文件均取名为 WJ，请问：系统应建立怎样的目录结构才能区分 4 名学生的程序？简述系统如何为这 4 名学生存取各自的程序。

信息资源,不必考虑用户或资源所在的地理位置。

(3) 支持分布式信息处理:在计算机网络的支撑下,许多大型信息处理问题可借助于分散在网络中的多台计算机协同完成,解决单台计算机无法或极难完成的信息处理任务,于是促进了分布式数据库管理系统的发展,使得分散存储于网络中不同计算机节点上的数据在使用时如同集中存储和处理一样。

由于网络中的计算机可互为备份,让同类资源分布在不同的计算机上,一旦某台机器出现故障,网络可通过不同的路由来访问这些资源,或故障机器的任务可由其他计算机承担,从而提高计算机系统的可靠性。当网络中的某些机器负荷过重时,也可以把一些任务分配给网络中的空闲计算机去完成,提高每台计算机的可用性和使用效率。

2. 计算机网络的产生和发展

计算机网络是计算机与通信技术相结合的产物,始于 20 世纪 60 年代,近 40 年来得到迅猛发展。最早的计算机网络是一台主机通过电话线连接若干远程终端,它是以单个主机为中心的星状网,这类结构称为第一代计算机网络。由美国麻省理工学院林肯实验室为空军所设计的 SAGE 半自动化地面防空系统被认为是计算机和通信技术相结合的先驱。

现代意义上的计算机网络是从 1969 年美国国防部高级研究计划署(DARPA)建成的 ARPANET 试验网和欧洲支持商业应用的 X.25 网开始的,也被称为第二代计算机网络。它以通信子网为中心,由许多主机和终端设备在通信子网的外围构成资源子网,通信子网不再仅仅依赖于类似电话通信的电路交换方式,更多地采用适合数据通信的分组交换方式,网络控制方式也由集中趋于分散,引入专门的通信控制处理器,大大降低了计算机网络的通信成本,这些都是现代计算机网络的典型特征。

计算机网络是一个非常复杂的系统,相互通信的计算机系统必须高度协调地工作。经过多年的研究,人们对网络技术、方式和理论的研究日趋成熟,为了促进网络产品的开发,许多计算机厂商制定自己的网络技术标准。如 IBM 公司的 SNA(System Network Architecture, 系统网络体系结构);DEC 公司的 DNA(Digital Network Architecture, 数字网络体系结构);Univac 公司的 DCA(Data Communication Architecture, 数据通信体系结构);Burroughs 公司的 BNA(Burroughs Network Architecture, Burroughs 网络体系结构)等,这些网络技术标准仅适用于一个公司,不同公司的产品很难互通,终于促成网络国际标准的制定。1977 年,国际标准化组织(ISO)成立“计算机与信息处理标准化委员会(TC97)”下属的“开放系统互连分技术委员会(SC16)”,在现有各种网络标准的基础上,制定“开放系统互连参考模型”(OSI-RM)作为网络的国际标准,推动网络技术进一步发展,迈向第三代计算机网络的时代。

20 世纪 80 年代中期以来,计算机网络领域最引人注目的事件是因特网的飞速发展,它仍然属于第三代计算机网络,有自己的网络体系结构,未完全使用 OSI-RM。进入 20 世纪 90 年代后,计算机网络的发展更加迅速,目前正在向宽带综合业务数字网(BISDN)的方向演变,这就是通常所说的第四代计算机网络。1993 年 9 月,美国政府提出国家信息基础设施计划(National Information Infrastructure, NII),俗称“信息高速公路”,其重要内容之一就是建设一个覆盖全美

国的宽带综合业务数字网。

8.1.2 网络体系结构

1. 网络体系结构

计算机网络是一个复杂的大型系统，其实现过程中要解决很多技术问题，如支持不同通信介质、支持不同机器平台、支持多种通信规范、支持多种应用业务等，实现计算机网络时通常采用层次结构，将其分成若干层，每一层完成特定的功能，各层协调起来构成整个网络系统。网络体系是指完成计算机之间的通信协作，把互联的功能划分成有明确定义的层次，把网络的同层进程通信协议及邻层接口称为网络体系结构（network architecture）。具体地说，网络体系结构是关于计算机网络应设置几层、每层应提供哪些协议的精确定义，可见，它从层次结构和功能上来描述计算机网络结构，并不涉及每层硬件和软件的组成和实现问题。对于同样的网络体系结构，可以采用不同的方法设计出完全不同的硬件和软件，相应的层次提供完全相同的功能和接口。

2. 通信协议

在计算机网络中，为了保证数据通信双方能够正确、自动地进行通信，必须在信息传输顺序、信息格式和信息内容等方面制定一套规则和约定，这种规则和约定的集合就是网络协议（network protocol）。

由于网络体系结构是有层次的，因而，通信协议也被分为多个层次，每一层内还允许分成若干子层次，协议的各层次有高低之分。另外，通信协议应是可靠、有效的，否则会造成通信的混乱和中断。网络通信协议含有以下 3 个要素：

(1) 语义：是指对构成协议的元素含义的解释。例如，在基本型数据链路控制协议中规定，数据报文中的第一个协议元素的语义表示所传输报文的报头开始，接着是报头；而第二个协议元素的语义表示正文的开始，接着是正文，等等。

(2) 语法：规定将若干协议元素和数据结合起来表示一个完整内容时所应遵循的格式。例如，在传送数据报文时，协议元素和数据组合的次序为：报头开始符、报头；正文开始符、正文、正文结束符；最后是奇偶校验码。

(3) 规则：规定事件的执行顺序和匹配速率，如通信双方进行发送和应答的次序。

综上所述，网络协议实质上是网络中互相通信的对等实体间交换信息所使用的一种语言。

3. 开放系统互连参考模型

为了实现计算机系统的互连，开放系统互连参考模型（Open Systems Interconnection Reference Model, OSI-RM）把网络通信功能划分成顺序的七层模型，即物理层、数据链路层、网络层、传输层、会话层、表示层、应用层，每层实现各自的功能，通过各层接口和功能的组合与其相邻层连接。各层的主要功能介绍如下。

(1) 物理层：为通信提供物理链路，实现数据链路实体之间比特(bit)流的透明传输，处理与物理传输介质有关的机械、电气和信号的特性和接口，维持或拆除数据链路实体之间的物理连

接，并实现数据的发送和接收。

(2) 数据链路层：分为介质访问控制和数据链路控制子层，前者解决网络多用户竞争信道使用权的问题；后者负责建立、维持和释放数据链路的连接、数据传输时的流量控制、链路控制和差错控制，提供数据帧及无差错地传送以帧为单位的数据。

(3) 网络层：负责把数据从物理连接的一端送到另一端，其主要任务有：路由选择和中继，网络连接和多路复用，数据分段和组块，拥塞和流量控制。

(4) 传输层：通过向会话层提供可靠的端到端的服务，使其上层与下三层隔离开来，提供进程间的通信机制，保证数据传输的可靠性，其具体功能有：地址映射、多路复用、数据分段与重装、组块与分块。

(5) 会话层：在两个相互通信的应用进程之间建立、组织和协调其交互，提供会话控制和传输同步控制。

(6) 表示层：处理数据的语法表示问题，实现信息转换，包括：格式表示、数据压缩、加密和解密等功能。

(7) 应用层：提供通用应用程序，包含各种应用协议，例如，文件传送协议 FTP、存取和管理协议 FTAP、虚拟终端协议 VTP、报文处理系统协议 MHS、作业传递及操作协议 JTM 等。

采用 OSI 七层模型的网络中，不同计算机上的应用进程在进行数据递信时，其信息流动过程如下：源计算机系统的应用进程把欲发送的数据传送至最高层，由其在用户数据前面加上这一层的控制信息，形成最高层数据单元后送至次高层；次高层又在数据单元前面加上这一层的控制信息，形成次高层的数据单元后，把它传送至其下一层。以下各层以此类推，信息按此方式逐层向下传送直至最低层，由于最低层实现比特流传送，故不必再添加控制信息。当比特流经过传输介质到达目标系统时，依次按由低向高的次序逐层向上传送，且在每一层都依照相应的控制信息完成指定操作，再把本层的控制信息去掉，剩余下来的数据单元向上一层传输，依次类推，当数据最后到达目标系统应用层时，再由应用层将用户数据交给目标计算机系统的接收进程，这样便结束一次递信过程。

4. TCP/IP 网络体系结构

除了 OSI 之外，还流行其他网络体系结构，其中最广为人知的是 TCP/IP 协议，虽然此协议并非国际标准，但由于其简洁、实用，在计算机网络中占有很重要的地位，成为事实上的工业标准。

传输控制协议 (Transmission Control Protocol, TCP) 和网际协议 (Internet Protocol, IP) 是一个协议集，表示因特网所使用的一种网络体系结构，其主要特点是：适用于多种异构网络的互连，遵从可靠的端—端协议，与操作系统紧密结合，效率高且有较好的网络管理功能。

TCP/IP 技术是为容纳物理网络技术的多样化而设计的，是一个完整的传输协议，位于网络层 IP 之上，为进程提供高可靠的通信能力。TCP 提供基于连接的数据流管道传输，所采用的可靠性技术有：确认和超时重传、流量控制和拥塞控制等。IP 解决兼容性问题，由于各种网络技术的帧格式、地址格式等上层协议之可见因素差别很大，通过 IP 数据报和 IP 地址将它们统一起来，达到屏蔽低层细节，向上提供统一服务的目的。TCP/IP 协议的层次如图 8.1 所示。

TCP/IP 与 OSI 之间有不少差别,共有三层,顶层是应用层协议,相当于 OSI 的高三层;与 OSI 传输层相当的是中层传输控制协议 TCP 或 UDP;与 OSI 网络层相当的是底层网际协议 IP。TCP/IP 没有对更低层次做出规定,这是因为在设计时考虑到要与具体的物理传输介质无关。TCP/IP 协议集的主要内容和组成有:

(1) 应用层:对应于 OSI 的应用层、表示层和会话层。常用的协议有:简单邮件传送协议 SMTP,多用途因特网邮件扩充协议 MIME,域名系统 DNS,远程登录协议 Telnet,文件传送协议 FTP,网络基本 I/O 系统 NetBIOS,超文本传送协议 HTTP,等等。

(2) 传输层:对应于 OSI 的传输层。主要协议有:传输控制协议 TCP,提供可靠的数据传输服务;用户数据报协议 UDP,提供无连接、高效的数据报传送服务。

(3) 网络层:对应于 OSI 的网络层,提供主机间的数据传输能力。主要协议有:网际协议 IP,为传输层提供网际传输服务;网际控制报文协议 ICMP,允许主机或路由器报告有关 IP 服务状况;地址转换协议 ARP,把 IP 地址转换成网络物理地址;反向地址转换协议 RARP,将网络物理地址转换成 IP 地址。

(4) 数据链路层:提供 TCP/IP 与各种物理网络的接口,包括广域网和局域网、Ethernet、X.25、Token Ring 等。

5. IP 和 IPv6

TCP/IP 协议体系结构的关键是 IP 协议,1995 年,开发互联网协议标准的因特网工程任务组 (Internet Engineering Task Force, IETF) 发布下一代互联网的规范,将其命名为 IPng (IP Next Generation),此规范提出在新一代网络中用 IPv6 协议取代现有的 IPv4 协议。它增强了 IP 网络的许多功能,以适应更高网络速度及图像和视频应用所形成的混合数据流。推动新的 IPv6 协议发展的主要原因是对更大的网络地址空间的需求,现有的 IP 协议使用 32 位网络地址来表示目标地址和源主机地址,随着互联网呈爆炸式增长,大量网络连接到互联网,现有地址长度已经不能满足所有系统需要,IPv6 提供 128 位长的源和目的地址域。采用 IPv6 的网络将比现有网络具有更好的可扩展性、安全性和服务质量,最终,使用 TCP/IP 的所有系统都会从目前的 IP 体系结构过渡到 IPv6 体系结构。

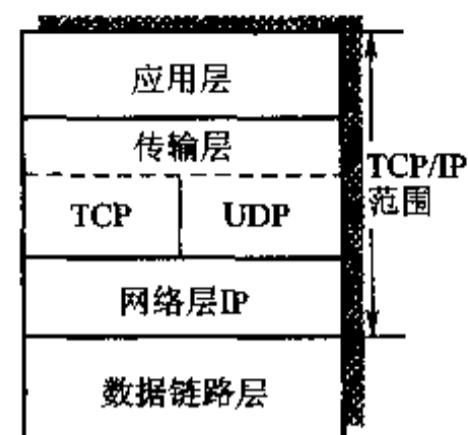


图 8.1 TCP/IP 协议的层次

8.2.1 网络操作系统概述

网络操作系统是网络用户和计算机网络之间的接口,除了具备通常操作系统的基本功能之外,还应具有联网功能,支持网络体系结构和各种网络通信协议,提供网络互连能力,支持可靠、

安全的数据传输,方便地实现网络资源共享,为网络用户提供各种服务。

典型的网络操作系统的特征有:硬件独立性,网络操作系统可以运行在不同的网络硬件上,可通过网桥或路由器与其他网络连接;多用户支持,能同时支持多名用户对网络的访问,应对信息资源提供安全和保护功能;支持网络实用程序及其管理功能,如系统备份、安全管理、容错和性能控制;多种客户端支持,如 Windows NT 网络操作系统可支持 OS/2、Windows 98、Windows for Workgroup、UNIX 等多种客户端,方便网络用户的使用;提供目录服务,以单一逻辑的方式让用户访问位于全世界范围内的所有网络服务和资源的技术;支持多种增值服务,如文件服务、打印服务、通信服务、数据库服务、万维网服务等;可操作性,这是网络工业的发展趋势,允许多种操作系统共享相同的网络电缆系统,且彼此可连通访问。网络操作系统分成以下三种类型:

1. 集中模式

集中式网络操作系统是由分时操作系统加上网络功能演变而成的,系统由一台主机和若干台与主机相连的终端构成,信息的处理和控制都是集中的。早期的 UNIX 分时系统是这类系统的典型例子。

2. 客户-服务器模式

网络中连接多台计算机,其中,一部分称为服务器,提供文件、打印、通信、数据库访问等功能,提供集中的资源管理和安全控制。另外一部分称为客户机,它向服务器请求服务,如文件下载和信息打印等。服务器的配置通常很高,运算能力强,有时还需要专职网络管理员来维护;客户机与集中式网络中的终端不同,它有独立的处理和计算能力,仅在需要某种服务时才向服务器发出请求。这一模式的特点是信息的处理和控制都是分布的,因而,又称分布式处理系统,NetWare 和 Windows NT 是这类操作系统的代表。

客户-服务器模式在逻辑上归入星状结构,以服务器为中心,与各客户之间采用点到点通信方式。当今两种主要的服务器应为:文件服务器和数据库服务器。无论是哪一种,客户机在请求服务器的服务时,双方需要多次交互:客户机发送请求包、服务器接收请求包、服务器回送响应包、客户机接收响应包。这种结构的优点是:数据分布存储、数据分布处理、应用编程较为方便。

3. 对等模式

让网络中的每台计算机同时具有客户机和服务器两种功能,既可向其他机器提供服务,又可向其他机器请求服务,网络中不存在中央控制手段。对等模式适用于工作组内几台计算机之间仅需提供简单的通信和资源共享的场合,也适用于把处理和控制分布到每台计算机的分布计算模式,NetWare Lite 和 Windows for Workgroup 是这类网络操作系统的代表。对等模式的优点是:平等性、可靠性和可扩展性较好。

8.2.2 网络操作系统实例

本节以 Windows NT 为例,介绍与操作系统的网络管理和控制有关的功能。其适用网络基于对等结构,可与其他类型的网络服务器进行通信,支持 NetWare 网络、基于 TCP/IP 的 UNIX 环境。其特点为:网络功能融合在操作系统中,网络驱动程序是执行体中 I/O 子系统的组成部

分；内置远程打印、电子邮件、文件传输等功能；支持多种网络协议和网络，如 Novell、Banyan、Sun NFS 和 VINES；采用开放式结构，如重定向程序、传输驱动程序和网络服务器均可动态地装入和卸载。下面介绍 Windows NT 的网络管理及服务功能。

1. 域管理模型

域(domain)是改进的工作组，或称超级工作组，具有集中的安全控制，对域资源(如打印机、目录等)的所有访问都由此域中的某台计算机授权监控，这台计算机是主域控制器。使用域的优点：一是域对用户来说有单独的密码，此密码可以打开域的所有被授权使用的资源；二是密码可由用户自行设定。每位用户都有一个账号及其用户信息的数据库记录，权限是指允许对网络资源进行访问的权力(只读、读/写等)，组可用于为相同的用户集合分配权限。

在 Windows NT 服务器中，域用户管理是管理网络用户的账号、组及计算机和域安全规则的最基本的管理工具，域是 NT 服务器网络中最基本的维护安全和进行管理的单元，域是共享同一个安全数据库和浏览列表的所有计算机的集合，同一域中的服务器共享安全规则和所有的账号信息。

域用户管理器是 NT 服务器中提供网络安全的程序之一，负责分配系统级的权力或定义网络所采用的监察规则。允许网络管理员完成以下工作：创建、修改、删除域中的用户账号，控制用户账号的密码特性；定义用户环境和网络信息，决定用户或组的系统权限；为用户账号分配登记信息；管理域中的组和组中各账号间的成员关系；管理网络中不同域之间的信任关系和域的安全规则，审核和定义记录的安全事件类型。

2. 文件管理及网络资源管理

在 NT 服务器中，仍然由资源管理器进行磁盘控制、目录和文件权限管理，完成通常文件管理所实现的所有任务，但在网络环境下有些特殊情况需要处理。此外，资源管理器还要为网络管理员提供目录共享功能、共享特性的修改、共享目录和文件的访问管理等工作。NT 服务器使用 NTFS 来提供可靠性和对共享文件的安全支持，提供多种存取权限，用来加强文件系统的安全管理，本地用户也可设置文件访问权限来限制用户的访问。

3. 文件和目录复制功能

文件和目录复制是 Windows NT 所提供的服务之一，用于在一个或多域的若干计算机上保存登录脚本、系统规则文件及其他常用文件备份。目录复制用于在多个服务器或工作站上设置相同的目录，主目录保存在指定的 NT 服务器上，其上的文件更新会被复制到其他指定的计算机上。文件和目录复制使得多个服务器都有相同的文件可被使用，因此，允许用户在任何可用的服务器上访问这些文件。此外，目录复制还有利于均衡服务器间的负荷，以避免过载。

4. 网络服务

Windows NT 提供强大的网络功能和广泛的 Internet/Intranet 支持，所能实现的功能有：DNS 域名服务；Internet 信息服务；多协议路由服务；动态主机配置协议中继代理服务；点对点通道协议及远程访问服务；拨号登录及网络环境管理，等等。此外，分布式组件对象模型 DCOM 能将组件对象 COM 的能力从本地应用扩展到包括组网与因特网的应用程序。

分布式操作系统

8.3.1 分布式系统概述

分布式系统(distributed system)是由一组独立的计算机系统,经互连网络连接而形成的“单计算机系统”(single computer system)。通常满足以下条件:

- (1) 系统中的任意两台计算机可通过系统的安全通信机制来交换信息;
- (2) 系统中的资源为所有用户共享,且无须考虑资源的物理位置,即为用户提供对资源的透明访问;
- (3) 系统中的若干台机器可互相协作来完成同一项任务,即一个程序可分布于多台计算机上并行运行,所以,分布式系统是一种特殊的计算机网络系统;
- (4) 系统中的一个节点出错不影响其他节点运行,具有较好的容错性和稳健性。

分布式系统要让用户使用起来像“单计算机系统”,实现分布式系统以达到这一目标的技术称为透明性。透明的概念适用于分布式系统的各个方面:位置透明性,是指用户不知道包括硬件、软件及数据库等在内的系统资源所处的位置,资源名中也不应包含位置信息;迁移透明性,是指资源无须更名就可从一个节点自由地迁移到另一个节点;复制透明性,是指系统可以任意复制文件或资源的多个副本,而用户无须对此有所了解;并发透明性,是指用户不必也不会知道系统中同时还存在其他用户与其竞争使用资源;并行透明性,是指在分布式系统中解决大型应用时,可由系统(编译、操作系统)自动找出潜在的并行模块去分布执行,而不为用户所察觉,这是分布式系统追求的最高目标。

用于管理分布式系统的操作系统称为分布式操作系统,它具备以下4项基本功能。

- (1) 进程通信:提供有力的通信手段,让运行在不同计算机上的进程之间可以交换数据;
- (2) 资源共享:提供访问其他机器资源的功能,使得用户可以访问或使用位于其他机器上的资源;
- (3) 并行计算:提供某种程序设计语言,使用户可以编写分布式程序,此程序可在系统中多个节点上并行运行;
- (4) 网络管理:高效地控制和管理网络资源,对用户具有透明性,即使用分布式系统与传统单机系统相似。分布式计算机系统的优点是:坚定性强、易扩充、可靠性好、便于维护和效率较高。

为了实现分布式系统的透明性,至少需要以下机制:一是有单一全局性的进程通信机制,在任何一台机器上,进程都采用同一种方法与其他进程通信;二是有单一全局性的进程管理和安全保护机制,进程的创建、运行和撤销以及保护方式不会因机器不同而异;三是有单一全局性的资源管理机制,隐蔽资源的分布,控制资源的分配,例如,提供统一的文件系统,用户存取文件的方式和单机一致。

分布式系统与网络系统的基础都是网络技术，在硬件连接、拓扑结构和通信控制等方面基本一样，都具有数据通信和资源共享功能，其主要区别在于：在网络系统中，用户在通信或资源共享时必须知道网络的构成、计算机及资源的位置，通常通过远程登录或让计算机直接相连来传输信息或进行资源共享；而在分布式系统中，用户在通信或资源共享时并不知道有多台计算机存在，其数据通信和资源共享如同在单计算机系统上一样。此外，互联的各计算机可互相协调工作、共同完成一项任务，可以把一个大型程序分布在多台计算机上并行运行。虽然分布式和网络操作系统都要管理通过网络联结的多台计算机，就像传统单机操作系统一样，对管辖的所有资源进行抽象和虚化，达到隐蔽硬件细节、提供接口、共享系统资源的目的，但它们的本质区别是呈现在用户面前的透明程度不同，分布式操作系统为用户提供的是一个具有分布处理功能的虚拟机；而网络操作系统为用户提供一组可互相通信的虚拟机。

8.3.2 分布式进程通信

分布式系统的主要特性之一是分布性，它源于具体应用的需求，应用程序可分布于分散的若干台计算机上运行；而通信则来源于分布性，因为机器的分散要求必须通过通信来实现进程的交互、合作和资源共享，可见分布式系统中的通信机制十分重要。

分布式系统中的底层进程通信有三种：消息传递机制、远程过程调用（Remote Procedure Call, RPC）和套接字（socket），它们都依赖于网络的数据传输功能。此外，为了解决互操作性，上层应用程序和下层通信软件之间增加一层标准的编程接口及协议——中间件（middleware），用来为分布式程序设计提供更好的支持和服务，目前已得到广泛的应用。

1. 消息传递机制

最简单的分布式消息传递模型称为客户—服务器模型，客户进程请求服务，如读入数据、打印文件，向服务器进程发送一条包含请求的消息，消息按照通信协议的规定来传递；服务器进程接收到消息后完成请求内容并返回结果。在设计分布式消息传递机制时，需要妥善解决目标进程寻址和通信原语的设计问题。

（1）目标进程寻址

客户向服务器发送消息，首先必须知道服务器的地址，这样消息才能到达此机器；其次，如果目标机器上有多个进程运行，那么消息到达机器后应该传递给哪个进程来处理呢？这就要解决目标进程的寻址问题。

第一种方法采用机器号和进程号寻址法：机器号用于使消息能正确地发送到目标机器上，进程号用来确定消息应传递给哪个进程。每台机器上的进程编号都从 0 开始，无须互相协调，因为进程 i 在机器 1 与机器 2 上不会发生混淆。这种方法的缺点是位置的不透明性，用户必须要知道服务器的地址。

第二种方法采用广播寻址法：让分布式系统中的进程在硕大且专用的地址空间中选择自己的标识号，如 64 位二进制整数空间内，两个进程选择同一数值的可能性极小。那么，发送消息时，发送到哪一台机器上呢？在支持广播通信的网络中，发送者广播一个特殊的定位包，包含目

标进程的地址。由于网络中的所有机器都能接收,由内核检查此进程是否在本机上。如果回答消息“我在这里”并给出机器号,发送消息的内核使用并记住这个地址,避免下次再广播。这种方法虽然具有位置透明性,但其缺点是广播通信将增加系统开销。

第三种方法采用名字服务器寻址法:系统中增加一台机器,由其提供机器名和机器地址的映射,称为名字服务器。客户机中存放 ASCII 码形式的服务器名,每次客户机运行并发送消息给一台服务器时,先发出请求消息给名字服务器,询问被请求服务器所在的机器号,有了这个地址,就可以直接发送消息。这种方法的缺点是要增加服务器及所发送的消息量。

(2) 同步和异步通信原语

分布式进程通信原语分为同步和异步两种,其基本形式为:

·Send(P, Message) ·Receive(Q, Buffer)

其中,P 表示接收进程,Q 表示发送进程,Message 表示要发送的消息,Buffer 则为接收者进程的信箱或缓冲区。在分布式系统中,进程标识实际上要由计算机地址和进程标识号组成。

同步发送时,发送进程要求接收进程做好接收消息的准备,在发送完消息后阻塞,并等待接收进程的回答。如果在预定时间内未收到回答,则认为消息已经丢失,应重新发送。同步发送消息的缺点是系统的并行性差,且无法利用广播消息功能,而广播功能在很多场合是很有用的。

异步发送时,发送进程把消息发送出去后,并不阻塞自己,而是继续运行。可见发送进程和消息传送可以并行工作,使系统的并行性能提高,异步通信的缺点是在消息被发送出去前,发送进程不能修改发送的消息缓冲区,它不知道传输何时进行,无法知道何时可以重新使用缓冲区。解决这个问题有两个方法:一是由发送原语把消息复制到系统缓冲区,允许发送进程继续工作;二是当消息被发送出去后,向发送进程发出中断并通知它缓冲区可用。

与发送消息相类似,接收消息也分为阻塞型和非阻塞型,在此不再细述。此外,一个进程可能希望不加选择地向一组欲接收消息的进程中的任意一个进程传递消息,一种受限的广播形式叫做“多播”发送,可向一组命名进程中的每个成员传递消息。类似地,进程可以按照消息的到达顺序来接收多个发送进程中的任意一个进程所发送的消息,一种使用隐含命名的接收消息操作可用于这个目的。阻塞型的接收消息和发送消息是解决各种进程协作问题的有力机制;广播发送往往使用非阻塞形式;阻塞型的接收比非阻塞形式更为有用,如打印或文件服务器进程接收客户端的服务请求消息,接收消息使用隐含命名的阻塞形式,因为事先并不知道发送者是谁,应该接收来自一组客户端的任何一个客户端发来的消息。

(3) 缓冲和非缓冲原语

非缓冲原语不使用缓冲区,有一个指向特定进程的地址,一个 receive(addr, &m) 调用告诉内核,此调用过程正在地址 addr 上监听,并准备接收一条发送到此地址上的消息,m 所指向的消息接收区将保存消息。当消息到达时,接收方内核把它复制到消息接收区,然后释放被阻塞的接收进程。

第一种方法是服务器方先调用 receive,客户端后调用 send,这种策略会工作得很好,但当 send 在 receive 之前被调用时,对于新来的消息,服务器内核如何知道哪个进程正在使用此地址?

消息又被复制到哪里呢？答案是“不知道”。一种简单的策略是：忽略这条消息，等到客户端超时，这样服务器就可以在客户重传消息之前调用 receive。这种方法虽然容易实现，但可能导致客户端内核重传多次。如果多次重传失败，客户端内核会放弃发送。

第二种方法是让接收方内核保留所到达的每条消息一段时间，直到相应的 receive 被调用。每到达一条消息就启动一个计时器，若在相应的 receive 被调用之前计时器超时，则此消息被丢弃。从理论上说，保留消息一段时间可定义新的数据结构“信箱”，准备接收消息的进程请求内核创建信箱，并指明一个地址以便查找网络信息包，以此地址标识的消息都被放入信箱中。对 receive 的调用只是从信箱中读取消息，若无消息可读则阻塞进程，采用这一技术的原语称为缓冲原语。信箱可连接多个发送者和多个接收者，很遗憾，在分布式环境中，接收消息的代价太高，因为指向同一个信箱的接收者可能驻留在不同的节点上。通常使用信箱的变种“端口”，端口仅与一个接收者相关联，从不同进程发来的指向同一个端口的消息都被送往一个与接收者相关联的位置。

(4) 可靠和非可靠通信原语

通常设想发送进程发送消息，服务器进程会接收到消息。在实际的通信过程中，消息可能会丢失，这样会影响消息传递模型的语义。解决这个问题的方法：一是尽可能保证传送，使用可靠的传输协议，附带完成检错、确认、重传、重排序，使消息正确到达目的地；二是发送进程所在机器的内核与接收进程所在机器的内核进行应答，仅当发送方收到接收方发来的确认消息后，发送方内核才释放客户进程。

2. 远程过程调用

远程过程调用是分布式系统中广泛采用的进程通信方法，它把单机环境下的过程调用拓展到分布式环境中，允许不同计算机上的进程使用简单的过程调用和返回结果方式进行交互，但这个过程调用是用来访问远程计算机所提供的服务，用户却感觉像在执行本地过程调用一样。在实现上，普通过程与远程过程不一样，后者在不同计算机上的地址空间中运行，它不能成为调用过程的一部分，相反地，必须有一个独立进程来执行被调用的过程，这个进程可在每次调用时动态地创建，也可作为专用服务进程静态地创建。

为了能以相同方式完成本地过程调用和远程过程调用，在客户-服务器上增设客户代理和服务器代理，当客户机上的进程需要调用服务器上的一个过程时，发出一条带参数的 RPC 命令给客户代理，它收到 RPC 命令后，便执行本次远程过程调用，把参数打成消息包，执行 send 原语请求内核把消息发送到服务器，通常客户代理调用 receive 原语阻塞自己直至服务器发来应答。消息到达服务器之后，远地内核把消息传送给服务器代理，接着，它拆包从消息中取出参数，然后，以一般方式创建或调用一个服务器进程，此进程执行相应的过程调用并以一般方式返回结果。当过程调用完毕后，服务器代理获得控制权，把结果打成消息包，再调用 send 原语请求内核把消息发回调用者。整个过程调用结束后，服务器代理回到 receive 状态，等待下一条消息。消息送回客户机后，内核找到消息并把它送给客户代理，客户代理检查并拆开消息包，取出结果返回给调用进程，调用进程获得控制权并得到本次过程调用的结果。图 8.2 说明远程过程调用的

流程,可把 RPC 的执行步骤总结如下。

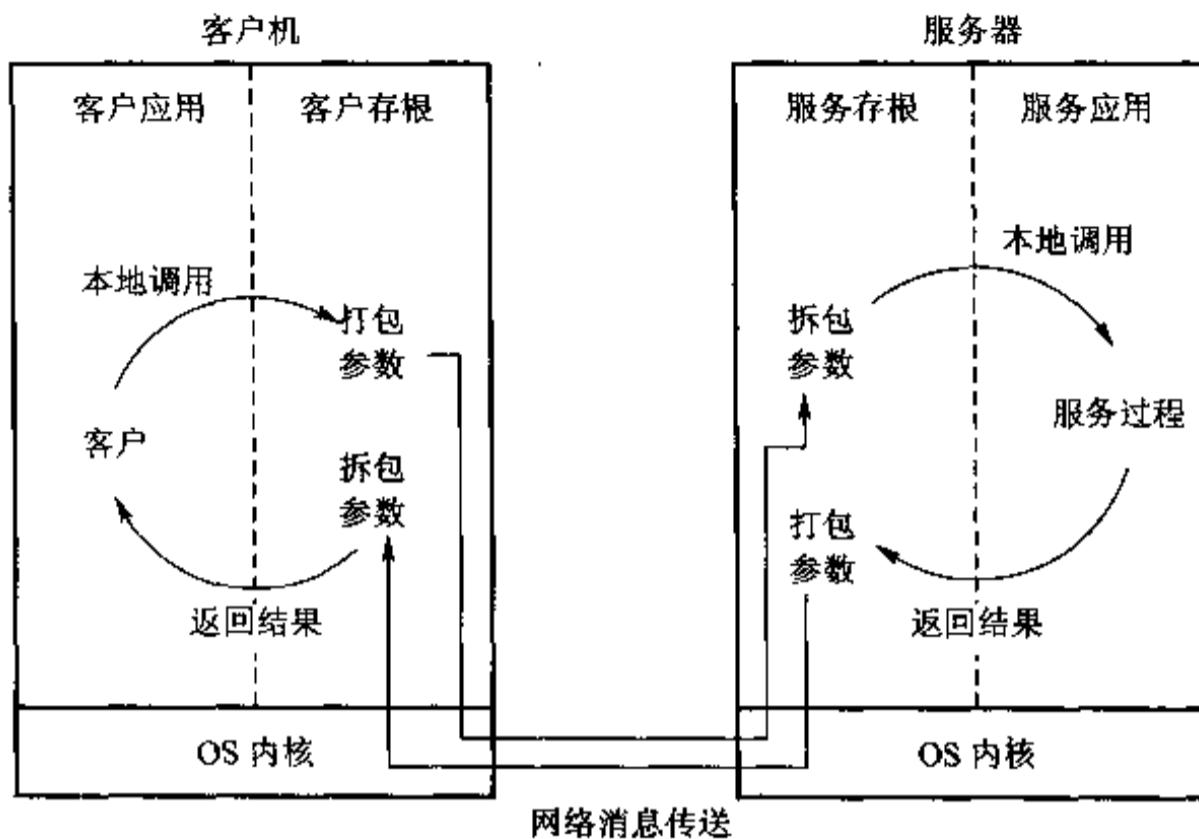


图 8.2 远程过程调用机制

- (1) 客户进程以普通方式调用客户代理；
- (2) 客户代理组织 RPC 消息并执行 send, 激活内核程序；
- (3) 本地内核通过网络把消息发送到远地内核；
- (4) 远地内核把消息送给服务器代理；
- (5) 服务器代理取出消息中的参数后调用服务器进程；
- (6) 服务器进程执行完后把结果返回至服务器代理；
- (7) 服务器代理将结果打包并激活内核；
- (8) 服务器内核通过网络把消息发送至客户内核；
- (9) 客户内核把消息交给客户代理；
- (10) 客户代理从消息中取出结果返回给客户进程；
- (11) 客户进程获得控制权并获得过程调用的结果。

上述 RPC 仅适用于同构型分布式系统,运行相同的操作系统,采用同种程序设计语言,有一致的参数表示方法;反之,对于异构型分布式系统来说,由于不同系统对数据的表示方式不同,导致客户机和服务器无法交互,故应该在 RPC 设施中增加参数表示的转换机制。例如,对于整数、浮点数、字符和字符串提供一个标准格式,这样,任何机器上的本地数据和标准表示均按规定作相应的转换。

3. 套接字

套接字是 UNIX BSD 首创的网络通信机制,提供进程通信端口,socket 是面向客户 - 服务器模式设计的低层通信机制。每个 TCP 及 UDP 的协议头都含有源端口和目的端口号域,端口号指

明两个相互独立的 TCP 单元使用者(应用程序),同时,IP 包含源地址和目的地址域,IP 地址指明两个相互独立的主机系统,端口号和 IP 地址结合起来唯一地表示一个应用在因特网上的地址。

服务器端套接字通常保持 TCP 或 UDP 端口处于打开状态,等待未知时间的连接呼叫,客户端通过查找域名系统(DNS)数据库,确定所期望的服务器的套接字标识号,一旦连接建立,服务器端就将会话切换至其他的端口号,以释放主端口号并等待其他连接请求。

套接字定义应用程序编程接口 API,它是编写程序使用 TCP 或 UDP 进行通信的通用编程接口,套接字 API 使用协议、IP 地址、端口号来共同定义唯一的套接字。有两种类型的套接字 API:流套接字和数据报套接字。流套接字使用 TCP 协议,提供面向连接的可靠数据传输,在两个套接字之间所传送的所有数据包能够保证以发送时的顺序递交和到达目的端;数据报套接字使用 UDP 协议,它不像 TCP 那样提供面向连接的特性,传输可靠性得不到保证,也不能确保数据的顺序递交。Socket API 主要有:创建 socket(),指定本地地址 bind(),建立连接 connect(),愿意接收连接 listen(),接收连接 accept(),发送数据 send()、sendto()、sendmsg()、write()、writev(),接收数据 recv()、read()、readv()、recvfrom()、recvmsg()。

套接字是 TCP/IP 网络通信的基本构件之一,由于基于 TCP/IP 协议的应用一般采用客户-服务器模式,因此,在实际应用中,必须有客户端和服务器两个进程,且首先启动服务器进程,通过无连接协议或有连接协议服务的调用时序进行工作。服务器的工作方式还可分为两种类型:重复服务器和并发服务器,前者面向短时间内能处理完的客户请求,由服务器自己处理到来的请求;后者面向需要长时间才能处理完的客户请求,每接到一个客户请求,创建一个子进程响应它,自己退回等待态,监听新的客户请求,形成一个主服务器和多个从服务器并发执行的局面。

4. 客户-服务器计算模型和中间件

(1) 客户-服务器计算模型

客户-服务器模型是一种基于局域网或广域网的分布式系统,例如,万维网基本上是一个客户-服务器系统,用户是客户机,Web 站点是服务器。客户-服务器模型的思想是:把操作系统看做一组协作进程,为用户提供各种服务,用户称为客户,协作进程称为服务器,客户和服务器通常运行相同的操作系统,且都作为用户进程运行,一台机器可以运行一个进程,也可以运行多个客户进程、多个服务器进程或两者的混合。为了避免面向连接协议所造成的大量开销,客户-服务器模型通常基于简单的、无连接的请求/应答协议,客户向服务器发送一条请求消息,要求提供某些服务(如查询信息),服务器工作并返回所需要的数据或出错代码,请求和应答消息都要经过内核进行。

客户-服务器模型的优点是简洁性和高效率,此模型所用协议集比 OSI 的要小,因而效率高。如果所有机器都相同,那么只需要三层协议:物理层、数据链路层和请求应答层(相当于 OSI 的会话层),前两层协议处理客户与服务器之间来往的数据分组,这是由硬件处理的,不需要路由选择,也不需要建立连接。在请求/应答层有相应的协议,它定义一组合法的请求和应答,由于结构简单,操作系统的通信机制只需提供两个系统调用:发送消息和接收消息,这两个系统调用可

通过函数库加以封装。

传统的客户-服务器体系结构包括两层：客户层和服务器层。近年来，三层结构模型变得日益普遍，在这种结构中，应用软件分布在三种类型的机器上：用户层机器（客户）、中间层机器（应用服务）及后端服务器（数据资源）。用户层机器一般是瘦型客户机；中间层机器基本上是位于客户和很多后端数据库之间的网关，它能够转换协议，将对一种类型的数据库查询映像为另一种类型的数据库查询，另外，中间层机器能够融合来自不同数据源的结果，它还因介于两个层次之间而可充当桌面应用程序和后端应用程序之间的网关；中间层机器和后端服务器之间的交互也遵从客户-服务器模型，因此，中间层同时充当着客户机和服务器的角色。

此外，基于多层客户-服务器模型结构的应用框架使用得越来越广泛，Sun 公司的 J2EE 体系结构可用于构建复杂的 Web 应用，J2EE 采用四层结构：Web 浏览器、Web 服务器、应用服务器和数据资源服务器。用户通过 Web 页而提交请求和接收结果；Web 服务器负责接收来自用户的请求并转交给应用服务器，另一方面负责接收来自应用服务器的计算结果并组织成 Web 页面返回给客户端；应用服务器负责具体的逻辑运算，在需要时访问数据资源服务器以获得计算所需的数据资源；数据资源服务器只负责应用数据的保存和维护，且对外提供访问数据的接口。

（2）中间件

① 中间件的概念

客户-服务器计算模型的开发和使用对分布计算的标准化提出很高的要求，缺乏标准化使得实现集成的、多厂商的、企业范围的客户-服务器配置变得十分困难，因为客户-服务器方式的优势是与其模块化及将平台和应用程序混合、协调来提供商业解决方案的能力紧密相联的，这种“互操作性”问题必须得到很好的解决。

实现这一要求的一种方法是：在上层应用程序和下层通信软件及操作系统之间使用标准编程接口和协议，建立支持在异构网络、异构计算机和不同操作系统中访问系统资源的工具，这种标准化的接口、协议和工具被称为“中间件”。

目前已经有很多中间件软件包，有些简单而有些非常复杂，其共同特点是能隐蔽不同网络协议和操作系统的复杂性和不一致性。客户机和服务器厂商一般都提供很多流行的中间件软件包以供选择，这样，用户可以采取一种特定的中间件策略，然后从厂商那里集成设备来支持这种策略。

② 中间件体系结构

图 8.3 给出在客户-服务器体系结构中的中间件的作用，中间件的确切作用将取决于所使用的客户-服务器计算的类型，即应用程序的功能划分。中间件具有客户端组件和服务器端组件两个部分，其基本作用是使位于客户端的应用程序或用户能够访问服务器上的各种服务，同时无须考虑服务器之间的区别。例如，对于特定的应用领域，结构化查询语言 SQL 提供本地或远程用户访问关系数据库的标准访问方式，然而，很多数据库厂商都对 SQL 进行特定的扩展，这就使得众多产品存在差别，产生潜在的不兼容性问题。

中间件提供一个软件层，其目的是隐蔽下层网络操作系统和平台的异构性，支持不兼容系统的统一访问，提供一致的分布式服务接口，把底层服务综合起来，以一种更透明、更高级的形式为

上层应用提供服务。从这种意义上来说,中间件好像是分布式操作系统。从逻辑的角度而非实现的角度来看,中间件所扮演的角色如图 8.4 所示,分布式系统可看做一组应用程序和用户可用资源的集合,用户无须关心数据或应用程序的实际位置,所有应用程序操作都建立在统一的 API 之上。中间件使分布式客户 - 服务器计算模式的互操作成为可能,它贯穿所有客户和服务器平台,负责将客户请求定位至合适的服务器上,中间件软件是针对互操作性的,其实现通常基于前面介绍过的底层通信机制:消息传递和远程过程调用。

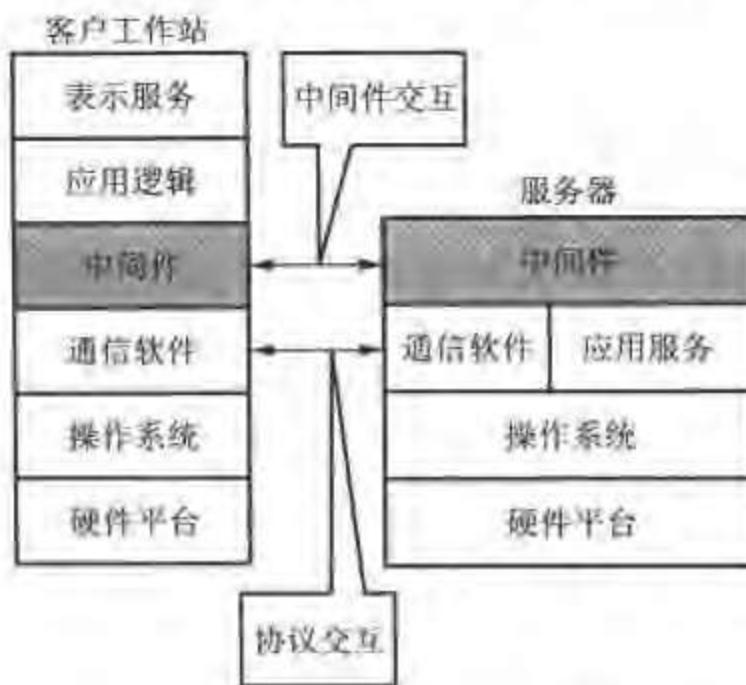


图 8.3 中间件在客户 - 服务器结构中的作用

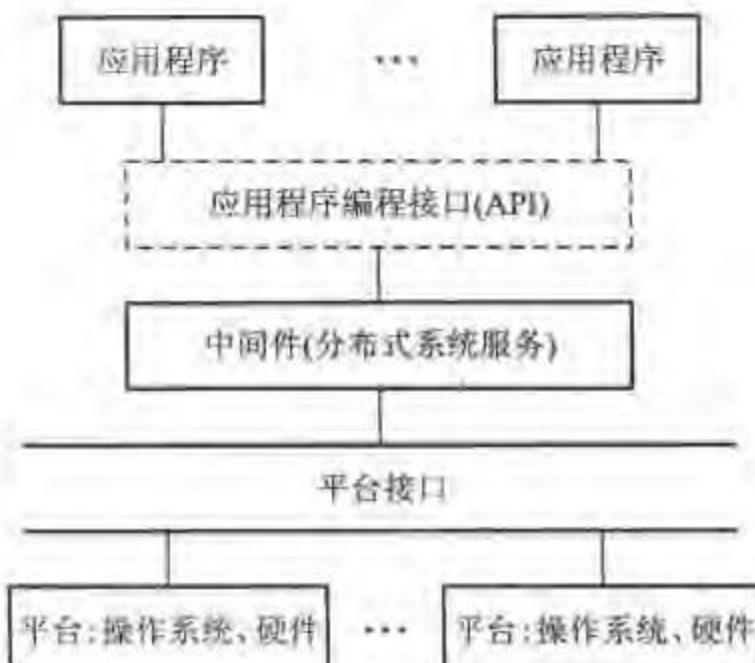


图 8.4 中间件的逻辑视图

大多数客户 - 服务器模型都采用中间件技术来实现,无论客户如何向服务器发送请求,无论客户访问哪个数据库,都通过中间件来帮助客户沟通从用户界面到数据访问的通路;同时,中间件还起着增强透明性的作用,隐蔽通信协议、隐蔽数据库查询语言、转换异构硬件的数据格式等。迄今为止,虽然没有一种标准的分布式计算环境,但已经出现的各种中间件,如微软公司的 COM (Component Object Model, 构件对象模型)、OSF 的 DCE(Distributed Computing Environment, 分布式计算环境)和 OMG 的 CORBA(Common Object Request Broker Architecture, 公共对象请求代理体系结构)等一系列实现方法推动着不同的分布式计算环境的迅速发展。

8.3.3 分布式资源管理

1. 分布式资源管理的概念

资源管理和调度是操作系统的一项主要任务,单机操作系统采用一类资源由一个资源管理者来管理的集中式管理方式,例如,所有主存空间都由存储管理负责分配与去配;所有打印机都由打印机管理负责打印事务等。在分布式系统中,资源分布在各台计算机上,一类资源归一个管理者来管理的办法会使系统性能下降,假如系统中各台计算机的存储资源由位于某台计算机上的资源管理者来管理,那么无论何方发出申请,即使所申请的是本机资源,都必须发送消息给存储管理者,这就大幅度地增加了系统开销;如果存储管理所在的那台计算机发生损坏,系统便会

瘫痪。可见,分布式操作系统如采用集中方式来管理资源,不仅系统开销大,而且稳定性差。

分布式操作系统通常采用一类资源由多个管理者来管理的方式,可分为两种:集中分布管理和完全分布管理。两者的区别在于:前者对所管资源拥有完全控制权,一类资源中的每一个资源仅受控于一个资源管理者;而后者对所管资源仅持有部分控制权,不仅一类资源存在多个管理者,而且此类资源中的每个资源都由多个管理者共同管理。

集中分布管理中,一类资源有多个管理者,但每个具体的资源仅有一个管理者对其负责。比如,系统有多个文件管理者,但每个文件仅受管于一个文件管理者,使用某个文件必须也仅须通过其相应的文件管理者进行。而完全分布管理却不是这样,假如一个文件有若干副本,这些副本分别受管于不同的文件管理者,为了保证文件副本的一致性,当一份副本正在被修改时,应禁止使用其他副本,因此,当一个文件管理者接收到使用文件的申请时,它只有在和管理此文件其他副本的所有管理者协商后,才能决定是否让申请者使用文件,在这种情形下,一个具有多副本的文件资源是由多个文件管理者共同管理的。

2. 集中分布管理算法

采用集中分布管理方式时,虽然每类资源由多个管理者管理,但此类资源中的一个资源却由唯一的管理者来管理,当一个资源管理者不能满足某申请者的请求时,它应当帮助用户向其他资源管理者申请资源。这样,用户申请资源的过程同单机操作系统一样,只要向本机的资源管理者提出申请,无须知道系统中有多少资源管理者,也无须知道资源的分布情况。因而,集中分布管理方式应具有向其他资源管理者提交资源申请和接受其他资源管理者所转来的申请的功能。由于资源管理者分布在不同的计算机上,系统必须制定资源搜索算法,使得能够按此算法帮助用户找到所需要的资源。设计分布式资源搜索算法应尽量满足以下条件:效率高,系统开销小,避免饥饿,资源使用均衡,具有坚定性。常用的分布式资源搜索算法有以下三种:

(1) 投标算法

资源管理者欲向它机资源管理者申请资源时,首先广播招标消息,向网络中位于其他节点的每个资源管理者发送招标消息,各节点的资源管理者接收到招标消息后,根据一定的算法计算出“标数”,然后向申请者发送一条投标消息,当申请者收到所有回答消息后,根据一定的策略选择一位投标者,并向其发送申请资源消息。

(2) 由近及远算法

让资源申请者按节点由近及远地搜索,直到某节点上具有所申请的资源时为止。

(3) 回声算法

资源申请者向其各个邻接节点发送探查消息,在消息中附有对资源的需求。类似地,每个邻接节点需要向每个邻接节点转发资源申请请求,申请者根据所有节点的回声消息,按照一定的策略选中一个资源提供者,然后,向其发送申请资源的消息。

8.3.4 分布式进程同步

在分布式系统中,各台计算机相互分散,未共享主存储器,因而在单处理器系统中所采用的

种种进程同步方式不再适用。如果两个进程运行在一台机器上,它们就能共享信号量,并通过执行系统调用来访问并发生交互作用;如果进程运行在不同的机器上,这种方法就不起作用。采用完全分布管理方式时,每个资源由位于不同节点上的资源管理者共同管理,某个资源管理者在决定分配其所管理的资源之前,必须和其他资源管理者协商,对于计算机系统的共享资源都要采用这种管理模式。因此,必须设计一个算法,各资源管理者按此算法共同协商资源的分配,此算法应满足:资源分配的互斥性,不产生饥饿现象,且各资源管理者处于平等地位而无主控者。通常称这种算法为分布式同步算法,由同步算法所构成的机制称为分布式同步机制。

实现分布式进程同步比实现集中式进程同步要复杂得多,由于进程分散在不同的计算机上,只能根据本地可用信息做出决策,并通过网络通信联系。由于消息通过网络传递时会有一定的延迟,当资源管理者接收到不同机器上的进程同时发来的资源申请时,先接到的申请的提出时间,很可能晚于后接到的申请的提出时间,所以,必须要解决对不同计算机中发生的事件进行排序的问题。然后,再设计性能优越的分布式同步算法。

1. 事件排序

进程同步的实质是对多个进程在执行顺序上施加规定,为此,应对系统中所发生的事件进行排序。由于在分布式系统中,各台计算机无公共时钟,也无共享存储器,很难确定系统中所发生的两个事件的先后次序。1978年,Lamport提出不使用物理时钟而对分布式系统中所发生的事件进行排序的方法。如果两个进程是无关的,那么,它们的时钟根本无须同步,对于相交进程而言,通常并无必要让进程在绝对时间上完全一致,只要限定它们执行的先后次序就行。

定义关系“先发生”,将其表达为“ $a \rightarrow b$ ”,读作“ a 在 b 之前发生”,意思是指系统中的所有进程认为事件 a 先于事件 b 发生。有三种情况产生“先发生”关系:

情况(1):如果 a 和 b 是同一进程中的两个事件,且 a 发生在 b 之前,则 $a \rightarrow b$ 为“真”;

情况(2):如果 a 是一个进程的发送消息事件, b 为另一个进程接收此消息事件,则 $a \rightarrow b$ 为“真”;

情况(3):存在某个事件 c ,若有 $a \rightarrow c$ 且 $c \rightarrow b$,则 $a \rightarrow b$ 为“真”。

情况(1)说明同一进程中所发生的事件之间,可按发生时间的先后次序来确定“先发生”关系;情况(2)指出消息决不能在发送之前就已接收,因为传送消息的过程会有时间延迟,发送消息事件总是先于接收消息事件发生;情况(3)说明“先发生”关系具有传递性。

按照上述方法定义事件的“先发生”关系之后,同一进程中两个事件的先后关系被明确定;不同进程中所发生事件间的先后关系,一部分可被确定,另一部分则不能确定。例如,有 3 个进程 P_1 、 P_2 和 P_3 ,它们分别发生以下事件:

事件 a: P_1 发送消息给 P_2 ;

事件 b: P_2 接收来自 P_1 的消息;

事件 c: P_2 接收到 P_1 的消息后,发送消息给 P_3 ;

事件 d: P_3 接收来自 P_2 的消息。

显然,有 $a \rightarrow b \rightarrow c \rightarrow d$ 。然而,假设 P_2 在事件 b 之前发生过某事件 f,如打印输出,尽管可以确定:

$$f \rightarrow b, f \rightarrow c, f \rightarrow d$$

这些关系,但是 a 和 f 之间的先后关系却无法确定。如果两个事件 x 和 y 发生在不同的进程中,且这两个进程不交换信息,那么 $x \rightarrow y$ 和 $y \rightarrow x$ 都不成立,这两个事件称为并发事件。简单地说,无法确定这两个事件孰先孰后。

2. 逻辑时钟

Lamport 定义的逻辑时钟又称时间戳(timestamp),是指能为系统中的所有活动赋予一个编号的机制,可利用整数计数器来实现。定义逻辑时钟的实质是把一个系统中的事件映射到正整数集合上的一个函数 C ,并满足:若事件 a 先发生于事件 b ,则 $C(a) < C(b)$,此处 $C(a)$ 和 $C(b)$ 分别是事件 a 和事件 b 所对应的逻辑时钟函数值,系统中的每个进程都拥有自己的逻辑时钟。

构造逻辑时钟函数的方法很多,任何满足上述映射关系的正整数函数都可作为逻辑时钟。下面是逻辑时钟函数的一种构造方法,定义在某系统集合上的逻辑时钟函数 C 如下。

(1) 对任一进程 P 中的非接收消息事件 e_j ,若 e_j 是 P 的第一个事件,则

$$C(e_j) = 1$$

若 e_j 是 P 的第 j 个事件,而第 $j-1$ 个事件是 e_{j-1} ,则

$$C(e_j) = C(e_{j-1}) + 1$$

(2) 对于任一进程 P 中的接收消息事件 e_r ,若 e_r 是 P 的第一个事件,则

$$C(e_r) = 1 + C(e_s)$$

此处, e_s 是进程 P' 发送这条消息的事件;若 e_r 是 P 的第 r 个事件,而 P 的第 $r-1$ 个事件是 e_{r-1} ,则

$$C(e_r) = 1 + \max[C(e_{r-1}), C(e_s)]$$

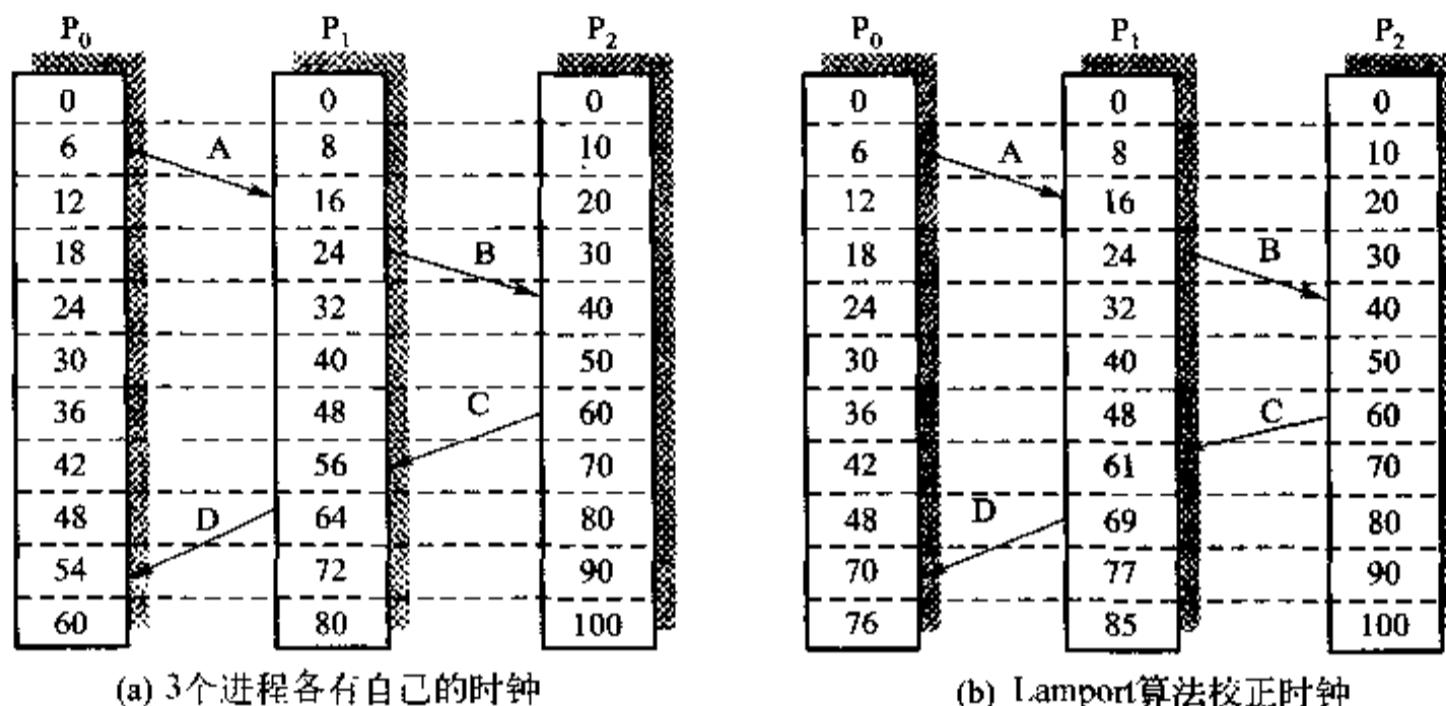
定义逻辑时钟后,可对一个系统中的所有事件人为地排出前后关系。对于由 n 个进程所组成的网络系统,逻辑时钟可用于对由消息传输所组成的事件进行排序,网络中的每个节点 i 都维护一个本地计数器 C_i ,其功能相当于本地时钟。每次系统发送消息时,首先把时钟加1,再发送一条消息,形式为 (m, T_i, i) (其中, m 为消息内容, T_i 为此消息的逻辑时钟, i 为节点编号)。分布式系统中的进程可以拥有自己的逻辑时钟(各个节点上的本地时钟),而这些时钟并非同步运行,可能出现这种情况:一个进程发送消息中所含的逻辑时钟大于接收进程收到此消息时它所具有的逻辑时钟,由于发送消息事件必定出现在接收消息事件之前,故需要调整接收进程的逻辑时钟。所以,接收进程 j 按照上述规则(2)应将其时钟设置为当前值和到达的逻辑时钟值这两者取最大值再加1。由于逻辑时钟只能向前走,即为单调递增,所以校正逻辑时钟值是加至少为1的正数。在每个节点上,事件的排序由下列规则确定:对于来自站点 i 的消息 x 和来自站点 j 的消息 y ,若下列条件之一成立,则说事件 x 先发生于事件 y 。

- (1) $T_i < T_j$;
- (2) $T_i = T_j \wedge i < j$

与每条消息有关的时间是附加在消息上的时间戳,时间的顺序是通过上述两条规则确定的。

现在来讨论事件排序规则的原理,看看 Lamport 提出的时钟同步算法如何校正系统中发生事件的逻辑时间。如图 8.5 所示,考虑有 3 个进程并发工作,它们运行在不同的机器上,每台机

器都带有自己的时钟,且按照自己的速度计时。在进程 P_0 中,时钟滴答为 6;而此时在进程 P_1 中,时钟滴答为 8;而在进程 P_2 中,时钟滴答为 10;每个时钟都按照恒定的速率计时,但不同机器上的晶体振荡存在差别,造成各自的时钟速率不一样。



(a) 3个进程各有自己的时钟

(b) Lamport算法校正时钟

图 8.5 应用 Lamport 算法的示例

在时钟滴答 6, 进程 P_0 发送消息 A 给进程 P_1 , 这条消息花多长时间到达目的地, 取决于基于哪一个时钟来计算。如果进程 P_1 接收到这条消息是在时钟滴答 16, 而同时消息 A 所携带的时钟滴答为 6, 那么, 进程 P_1 可认为此消息在路上花费 10 个时钟滴答, 这是完全可能的时间值。同样道理, 消息 B 从进程 P_1 传送到进程 P_2 花费 16 个时钟滴答, 这也是完全可能的时间值。

现在来看消息 C, 它从进程 P_2 传送到进程 P_1 , 开始发送时的逻辑时钟为 60, 而消息到达时的时钟滴答为 56;类似地, 消息 D 从进程 P_1 传送到进程 P_0 , 开始传送时的时钟滴答为 64, 而消息到达时的时钟滴答为 54。显然这些时间值是不可能的,要对其加以校正。

Lamport 的时钟校正方法是从“先发生”关系直接得出来的。因为消息 C 被发送时的时钟滴答为 60, 当它到达时,其时钟滴答值应为 61 或大于 61。因此,让每条消息都携带其发送者的时钟所确定的发送时间,当消息到达目的地时,若接收者的时钟当时的指示值先于消息的发送时间,接收者的时钟值就应校正为发送时间加 1 之后的值。从图 8.5(b)可以看出,消息 C 到达的时间现在是 61;同样,消息 D 到达的时间是 70。

增加一个附加条件,Lamport 算法就能满足系统中全局性时间的需要,为所有事件确定全局顺序关系。所增加的条件是:在两个事件之间,时钟至少需要滴答一次。如果一个进程在极短的时间内快速发送或接收两条消息,那么,必须调整时钟,使时钟在这两个事件之间至少要前进一个滴答。

在某些情况下,上面的算法需要满足条件:任何两个事件都不会恰巧在完全相同的时刻发生。为了满足这个关系,对具有相同时间戳的两条消息则通过其所在的站点编号来排序。这种规定能够避免各速信进程的不同时钟之间的重合问题。

在如图 8.6 所示的时间戳算法的操作示例中，有 3 个节点都通过一个控制时间戳算法的进程来表达。进程 P_1 的开始时钟值为 0，为了传送消息 a ，把时钟值加 1 并发送 $(a, 1, 1)$ ，这条消息被节点 2 和 3 的进程收到，由于这两种情况中本地时钟值是 0，则时钟值应被设置成 $2 = 1 + \max[0, 1]$ 。接着， P_2 首先将其时钟增加为 3，再发出下一条消息。当接收到消息后， P_1 和 P_3 必须把它们的时钟增加到 4。然后，在大致相同的时间，以相同的时间戳， P_1 发出消息 b ，而 P_3 发出消息 j 。前面所介绍的排序原则不会产生混淆，在所有这些事件发生之后，消息的顺序在所有节点上是相同的，依次为 a, x, b 和 j 。

如果不考虑在两个节点间传送消息时间上的差别，讨论如图 8.7 所示的例子。这里 P_1 和 P_4 以相同的时间戳发出消息，来自 P_1 的消息在节点 2 上比来自 P_4 的消息到得早；但在节点 3 上，来自 P_1 的消息比来自 P_4 的消息到得晚。不过，当所有消息在所有节点上都接收完毕后，消息的顺序在所有节点上是相同的，依次为 a, q 。为什么在节点 3 上的消息次序也是 a, q 呢？这是因为 P_1 和 P_4 以相同的时间戳发出消息 ($T_1 = 1$ ，同时 $T_4 = 1$)，但由于节点号 $i < j$ ，所以， P_1 发消息的事件先发生， P_4 发消息的事件后发生，故在节点 3 上，消息次序应为 a 先而 q 后。

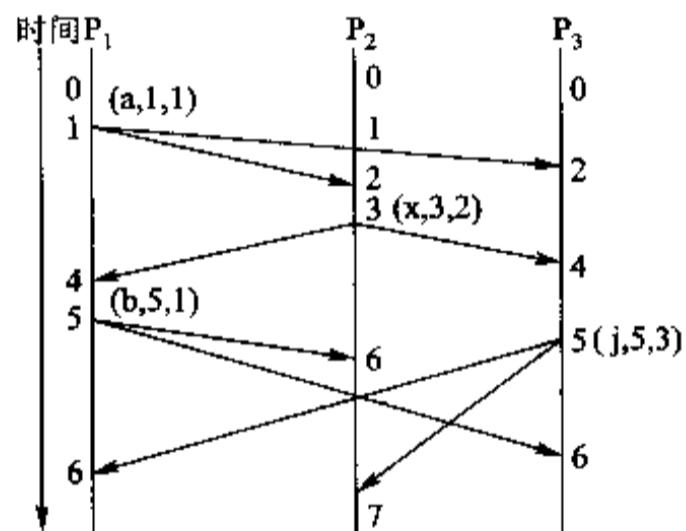


图 8.6 时间戳算法的操作示例

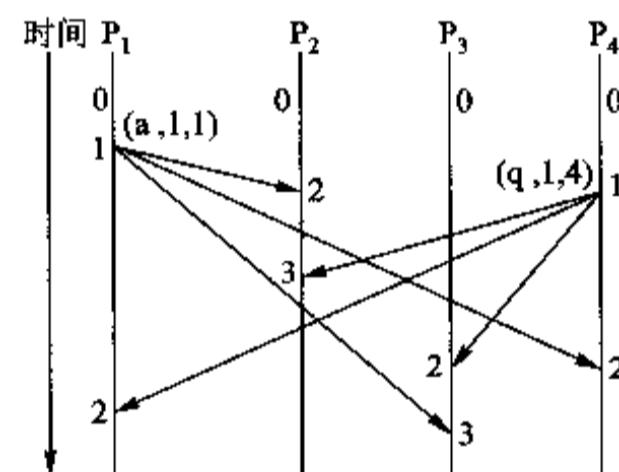


图 8.7 时间戳算法的另一个操作示例

3. 分布式同步算法

(1) Lamport 算法

最早提出的分布式同步算法是 Lamport 算法，它利用事件排序方法，对要求访问临界资源的全部事件进行排序，按照先来先服务的原则，对事件进行处理。Lamport 算法基于以下假设：分布式系统由 N 个节点所组成，每个节点建立一个数据结构，称其为请求队列，用来记录此节点最近收到的消息和此节点自己产生的消息，队列中的消息按照逻辑时钟排序，不妨假定每个节点只有一个进程且仅负责控制一种临界资源，并处理那些同时到达的请求。Lamport 算法用到三种类型的消息，每条消息的内容包括：消息类型、时间戳、节点号：

$(\text{request}, T_i, i)$ ：进程 P_i 发出的申请资源的请求消息；

(reply, T_j, j) ：进程 P_j 同意请求进程访问其所控制的资源的回答消息；

$(\text{release}, T_k, k)$ ：占有资源的进程 P_k 释放资源时发给各进程的释放消息。

Lamport 算法描述如下：

① 进程 P_i 要求访问临界资源时, 把申请资源消息($\text{request}, T_i, i$)插入自己的请求队列, 并发送给集合中的每个进程。

② 进程 P_j 接收到申请资源消息($\text{request}, T_i, i$)时, 按照逻辑时间“先发生”关系插入自己的请求队列; 形成一个带有时间戳的回答消息(reply, T_j, j), 广播发送给集合中的每个进程。

③ 若满足以下条件, 则允许进程 P_i 进入临界区访问资源。

(a) P_i 申请资源的请求消息已处于自己的请求队列的最前面, 因为消息在所有节点上一致排序, 故在任意时刻只有一个进程可以访问资源。

(b) 进程 P_i 已收到其他进程的回答消息, 当这些消息上的时间戳晚于(T_i, i)时, 此条件表明其他进程要么不想访问相同的资源, 要么要求访问资源但其时间戳较晚。

④ 为了释放所请求的资源, 进程 P_i 从自己的请求队列中删除申请资源消息($\text{request}, T_i, i$), 再发送一条带有时间戳的释放资源消息($\text{release}, T_i, i$)给所有其他进程。

⑤ 进程 P_j 收到进程 P_i 的释放资源消息后, 从自己的请求队列中删除进程 P_i 的申请资源消息($\text{request}, T_i, i$)。

为了确保互斥性, 此算法共需要传送 $3(N - 1)$ 条消息, 包括 $(N - 1)$ 条 request 消息、 $(N - 1)$ 条 reply 消息和 $(N - 1)$ 条 release 消息。

(2) Ricart 算法

Ricart 等人对上述算法进行改进, 试图通过消除 release 消息, 使得进程在访问资源时, 减少所发送的消息量。Ricart 算法描述如下:

① 当进程 P_i 要求访问临界资源时, 发送一个带有时间戳的广播消息($\text{request}, T_i, i$)给所有节点的进程。

② 当其他节点进程 P_j 接收到消息($\text{request}, T_i, i$)时, 执行以下几个操作。

(a) 如果进程 P_j 既不是资源申请者, 又不是资源占有者, 则立即返回一条回答消息(reply, T_j, j)给进程 P_i 。

(b) 如果进程 P_j 是资源申请者, 并满足条件 $T_i < T_j$, or $T_i = T_j \&\& i < j$, 则返回一条回答消息(reply, T_j, j)给进程 P_i ; 否则推迟发送 reply 响应。

③ 当进程 P_i 已收到其他进程的回答消息 reply 时, 便可访问此临界资源;

④ 占有资源的进程 P_i 在释放资源时, 对那些曾经接到过其申请消息但未予回答的进程补送回答消息 reply。

当一个进程欲进入临界区时, 它就给所有其他进程发送一条带有时间戳的请求消息, 当它从所有其他进程接收到回答后, 就可进入临界区。当一个进程收到另一个进程的请求时, 必须发送一个对应的回答, 如果此进程不想进入临界区, 它就马上发送回答消息; 若它想要进入临界区, 就把自己的请求的时间戳与所收到的请求的时间戳进行比较, 如果后者较迟, 它就延迟发送回答; 否则马上发送回答。此算法需要使用 $2(N - 1)$ 条消息, 其中 $(N - 1)$ 条请求消息表示 P_i 要进入临界区, $(N - 1)$ 条回答消息表示允许其请求资源的消息。

(3) 令牌环算法

Suzuki 在 1982 年提出的用于分布式系统的另一种同步算法是令牌环算法,为了实现进程互斥,所有进程构成一个逻辑环(logical ring),环中的每个进程都有前驱和后继,这样的逻辑环可由软件实现,并不意味着任何物理拓扑结构。系统中设置一个象征存取权力的令牌(token),它是一种特定格式的报文,不断地在进程所组成的逻辑环中循环。

利用令牌实现进程互斥的过程如下:令牌在初始化后,被逻辑环中的任意一个进程获得,这样令牌开始绕环移动,它从进程 K 传递给它的下一个进程 $K+1$,可按照点到点方式进行传递。当进程从其前驱那里得到令牌时,它就检查所欲进入的临界区。如果临界区是开放的,则此进程进入临界区,访问共享资源;当此进程退出临界区时,便把令牌传送给其后继,不允许使用同一张令牌进入第二个临界区。当进程得到前驱传递来的令牌,又不想进入临界区,就把令牌往后传,这样一来,如果没有进程想进入临界区,令牌就会在逻辑环中高速循环运行。这种算法能实现进程互斥是显而易见的,由于只有一张令牌,任何时刻只有一个进程拥有令牌,所以只有一个进程可以进入临界区。由于令牌以固定的顺序移动,存在着前驱优先于后继的关系,也不会出现饥饿现象,一旦某个进程想进入临界区,最坏的情况是等待所有其他进程进入后再退出临界区所耗费时间的总和。

8.3.5 分布式系统中的死锁

1. 死锁类型

在网络和分布式系统中,死锁的防止、避免和解除等方法与单处理器系统相似,但其难度和复杂度要大得多。在分布式环境下,由于进程和资源的分布性,竞争资源的诸进程来自不同的节点,拥有共享资源的每个节点,通常只知道本节点中的资源使用情况,因而,检测来自不同节点中的进程在竞争共享资源时是否会产生死锁,显然是很困难的。分布式系统中的死锁分为两类:资源死锁和通信死锁。资源死锁是因为一组进程竞争系统中可重复使用的资源,如打印机、磁带机以及存储器等而引起的,进程的推进顺序如果不当,很容易引发系统死锁。通信死锁是指:在不同节点中的进程,为发送和接收报文而竞争缓冲区,出现既不能发送,又不能接收的僵持状态。

2. 分布式死锁检测与预防

客观地存在两种检测方法。

(1) 集中式死锁检测

分布式系统中的每台计算机都有一张进程资源图,用以描述进程及其占有资源的状况。一台中心计算机上拥有一张整个系统的进程资源图,当检测进程检测到环路时,就中止一个进程以解决死锁。检测进程必须适时地获得从各个节点发送来的更新信息,可采用以下方法解决更新问题:一是每当进程资源图中添加或删除一条弧时,就应将相应的变动信息发送给检测进程;二是每个进程周期性地把自己从上次更新之后新添加或删除的弧的信息发送给检测进程;三是检测进程在需要的时候主动请求更新信息。

上述方法可能会产生假死锁问题。在网络和分布式系统环境下,如果检测出进程资源图中的环形链,系统是否真的已发生死锁呢?答案是不确定的。其原因是,进程所发出的请求与释放资源命令的时序同执行这两条命令的时序未必一致。下面通过例子来说明假死锁的情况。考虑进程 A 和 B 运行在节点 1 上,进程 C 运行在节点 2 上;共有 3 种资源 R、S 和 T;开始状态如图 8.8(a)、(b)所示:A 拥有 S 且请求 R,但 R 被 B 所占用;B 使用 R;C 使用 T 且请求 S。

检测进程所检测到的状态如图 8.8(c)所示,这时系统是安全的,一旦进程 B 运行结束,进程 A 就可以得到资源 R,然后,运行就可以结束,并释放进程 C 所等待的资源 S。但不久之后,进程 B 释放资源 R 并同时请求资源 T,这是一个合法操作;节点 2 向检测进程发送消息以声明进程 B 正在等待其资源 T;假如节点 2 的消息比节点 1 发送的消息先到达,就将导致图 8.8(d)所示的进程资源图,检测进程错误地得出存在死锁的结论,并中止其中某进程。由于消息的不完整和延迟使得分布式死锁算法产生假死锁问题。

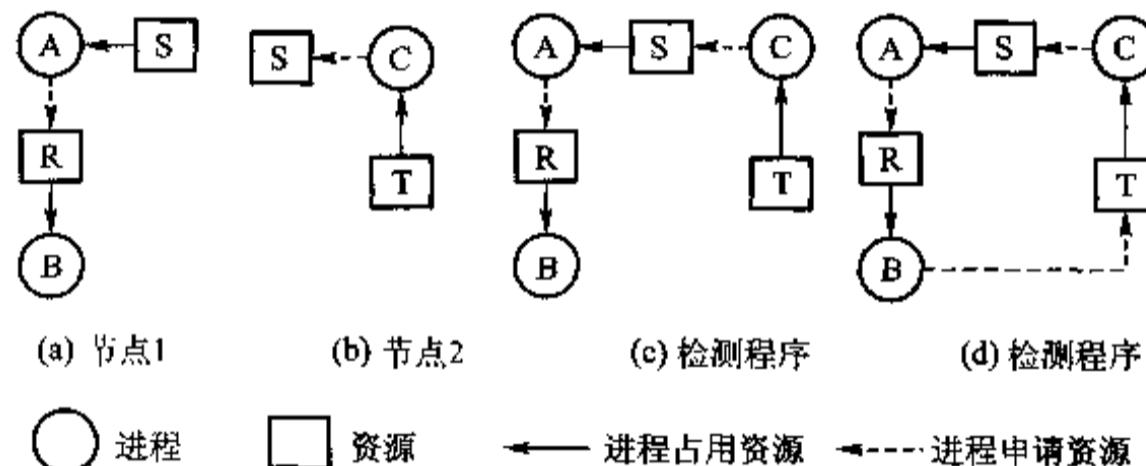


图 8.8 集中式死锁检测

可用 Lamport 算法所提供的全局时间来解决假死锁问题。从节点 2 到检测进程的消息是由于节点 1 的请求而发出的,那么,从节点 2 到检测进程的消息的逻辑时钟就应该晚于从节点 1 到检测进程的消息的逻辑时钟,当检测进程收到从节点 2 发来的含有导致死锁倾向的消息后,它将给系统中的每台机器发送消息:“收到从节点 2 发来的会导致死锁的带有逻辑时钟 T 的消息,如果有任何小于此逻辑时钟的消息要发送给我,请立即发送。”当每台机器都给出肯定或否定的响应消息后,检测进程会发现从资源 R 到进程 B 的弧已经消失,因而,系统仍然是安全的。这一方法的缺点是需要全局时间,系统开销较大。

(2) 分布式死锁检测

分布式检测算法无须在网络中设置掌握全局资源使用情况的检测进程,而是通过网络中竞争资源的进程相互协作来实现对死锁的检测。具体的实现方法如下:

- ① 在每个节点中都设置一个死锁检测进程;
- ② 必须对请求和释放资源的消息进行排队,在每条消息上附加逻辑时钟;
- ③ 当一个进程欲存取某资源时,它应先向所有其他进程发送请求信息,在获得这些进程的响应信息后,才把请求资源的消息发送给管理这一资源的进程;

④ 每个进程应将资源的分配情况通知所有进程。

由此可见,为了实现分布式系统环境下的死锁检测,通信开销相当大,而且还有可能出现假死锁,因而,在实际应用中,主要还是采用死锁预防方法。

为了防止在网络中出现死锁,可采取破坏产生死锁的 4 个必要条件之一的方法来实现。为了防止发生资源死锁,第一种方法可以采用静态分配方法,让所有进程在运行之前,一次性地申请其所需要的全部网络资源,这样,进程在运行过程中不会再提出资源申请,破坏“占用和等待”条件。如果网络系统无法满足进程的所有资源请求,索性连一个资源也不分配给此进程,这样也能预防死锁。第二种方法是按序分配,把网络中可供共享的网络资源进行排序,同时要求所有进程对网络资源的请求严格按照资源号从小到大的顺序提出,这样可以防止在资源分配图中出现循环等待事件。第三种方法主要解决报文组装、存储和转发造成缓冲区溢出而产生的死锁问题,为了避免发生组装型死锁,源节点的发送进程在发送报文之前,应先向目标节点申请一份报文所需要的全部缓冲区,如果目标节点无缓冲区,索性一个也不分配,让发送进程等待。为了避免存储转发型死锁,可为每条链路上的进程配置一定数量的缓冲区,且不允许其他链路上的进程使用;或者当节点使用公共缓冲池时,系统限制每个进程只能使用一定数量的缓冲区,而留出足够大的后备缓冲空间。

8.3.6 分布式文件系统

1. 分布式文件系统概述

分布式文件系统是分布式系统的重要组成部分,是允许通过网络来实现互连的、使不同机器上的用户共享文件的一种文件系统,其任务也是存储和读取信息,许多功能都与传统的文件系统相同。分布式文件系统不是一个分布式操作系统,而是一个相对独立的软件系统,它被集成到分布式操作系统中,并为其提供远程访问服务。分布式文件系统具有以下特点。

(1) 网络透明性

客户访问远程文件服务器上的文件的操作如同访问本机文件的操作一样。

(2) 位置透明性

客户通过文件名访问文件,但并不知道此文件在网络中的具体位置;同理,文件的物理位置若改变,但只要文件名不变,客户仍然可以对其进行访问。

2. 分布式文件系统的组成

分布式文件系统由两部分组成:运行在服务器上的分布式文件系统软件和运行在客户机上的分布式文件系统软件,两部分程序代码在运行过程中都要与本机操作系统的文件系统紧密配合,共同起作用。由于现代操作系统都支持多种类型的文件系统,因此,本机上的文件系统均是虚拟文件系统,它支持多个具体的文件系统,分布式文件系统将通过虚拟文件系统与本机文件系统交互作用。例如,Sun 公司的网络文件系统 NFS 由以下几部分组成:网络文件系统协议、远程过程调用协议、扩展数据表达、网络文件系统服务器代码、网络文件系统客户机代码、安装协议、服务器监听进程、服务器安装进程、客户机 I/O 进程和网络锁定管理器及状态监视器。

3. 分布式文件系统的体系结构

分布式文件系统的体系结构大多采用客户—服务器模式,客户是要访问文件的计算机,服务器是存储文件且允许用户访问文件的计算机。分布式文件系统需要解决命名透明性的问题,有两种方法:一是通过“机器名+路径名”来访问文件;二是将远程文件系统安装到本机文件目录上。分布式文件系统中需要解决的另一个问题是远程文件的访问方法,在客户—服务器模式中,客户使用远程服务方法访问文件,服务器则响应客户的请求。但是某些系统中的服务器能够提供更多的服务,它不仅响应客户的请求,还对客户机高速缓存的一致性做出预测,一旦客户数据无效时便通知客户。下面介绍实际的分布式文件系统——网络文件系统(Network File System,NFS),它已成为因特网上进行分布式访问的一种事实上的标准。

(1) NFS 结构

NFS 的基本思想是让任意组合的客户机和服务器通过局域网或广域网共享一个公共的文件系统,每个 NFS 服务器都输出一个或多个目录供远程客户机访问。一个目录可用,总是意味着其所有子目录也都是可用的,即所输出的总是一棵目录树,服务器所输出的所有目录都列在文件/etc(exports 中,以便当服务器引导时能自动地予以输出。

客户机在访问服务器的目录之前必须先行安装,客户机安装远程目录后,此目录就成为客户机目录层次的一部分,对于运行在客户机上的程序而言,文件是位于本地机还是远程机器上是没有什么相异的。因此,NFS 的基本结构特征就是服务器输出目录,而客户机安装目录,如果两个或多个客户机同时安装某个目录,它们就可以通过共享其中的文件来实现通信。

(2) NFS 协议

NFS 的主要目标是支持异构型系统,客户—服务器可能运行在不同的硬件平台和操作系统环境下,为此,需要对客户机和服务器的接口进行定义。NFS 定义两个客户—服务器协议来实现这一目标。

① NFS 处理安装协议:客户机向服务器发送一个路径名,请求将远程目录安装在其目录层次的某个位置,实际安装位置不包含在消息中,因为服务器并不关心它。如果路径名是合法的,且此目录已被服务器输出,服务器就返回一个文件句柄给客户机,文件句柄中的域指出相应目录的文件系统类型、所在硬盘、目录的 i 节点号以及安全信息,此后对已安装目录内的文件的访问就通过这个句柄进行。

② NFS 文件和目录访问协议:客户机可向服务器发送消息,以便操作目录和读写文件,也可以访问文件属性,如文件访问模式、大小、最近修改时间等。NFS 支持大部分 UNIX 系统调用。

NFS 使用 UNIX 对文件访问方式的保护机制,现在也可用公开密钥系统来为服务器和客户机间的每个请求和应答建立安全通信。

(3) NFS 实现

NFS 的实现分为三层:顶层是系统调用层,处理诸如 open、read、close 等函数,通过对系统调用的语法进行分析,并核对参数后,它就调用第二层 VFS 层。

VFS 层的作用是建立并维持一个打开文件表,它的每一项对应于一个打开的文件,很像

UNIX 中打开文件的 inode 表, 常规 inode 是通过(设备, inode 号)唯一指定的, 而 VFS 层则为每个打开文件设立一个称为 v 节点(虚拟 inode)的项, 用于指出文件是本地的还是远程的。

下面考查 mount、open、read 等系统调用, 说明 v 节点的使用方式, 系统管理员在安装远程文件系统时, 通过 mount 程序指定远程目录、本地安装位置及其他相关信息。

① 安装(mount): 安装程序分析远程目录, 找出远程机器名, 然后与远程机器取得联系, 要求得到远程目录的文件句柄。如果远程目录存在, 且允许远程安装, 服务器就返回此目录的句柄。最后, mount 程序执行 mount 系统调用, 将这个句柄交给核心。

核心接收到句柄后, 为远程目录构造 v 节点, 要求 NFS 客户机代码在其内部表中创建 r 节点(远程 inode)来安装这个文件句柄, 并将 v 节点指向 r 节点。VFS 层中的每个 v 节点最终都有一个指针指向 NFS 客户机代码中的一个 r 节点, 或本地操作系统中的 inode。因此, 从 v 节点中可以看出文件或目录是本地的还是远程的, 对后者还可以找到句柄。

② 打开(open): 打开远程文件时, 通过分析路径名, 内核会遇到安装远程文件系统的那个目录, 当发现此目录不在本地机后, 通过 v 节点就可以找到指向 r 节点的指针, 于是内核就要求 NFS 客户机程序去打开文件。客户机程序在远程服务器的相关目录中查找剩余的那部分路径名, 如果存在, 就取回一个文件句柄。它在自己的表中为远程文件建立一个 r 节点, 并报告 VFS 层, 后者就在自己的表中加入一个 v 节点, 并让它指向 r 节点。

open 调用者将得到一个远程文件描述符, 实际上它对应于 VFS 层中的某个 v 节点, 注意在服务器侧没有建立任何表格, 尽管它一直准备着为每个请求提供文件句柄, 它并不记录哪些文件的句柄已经给出, 哪些还没有, 当它收到一个文件句柄和相应的访问请求后, 只要核查结果是合法的, 就可以使用。

③ 读/写(read/write): 当文件描述符在后续的系统调用(如 read)中使用时, 由 VFS 层确定相应的 v 节点, 弄清是本地的还是远程的, 并得到对应的 inode 或 r 节点。

出于系统效率的考虑, 客户机和服务器之间的数据传送以较大的块(通常是 8 192 B)进行, 尽管实际所需要的数据量有时较小。客户机在得到其所需要的 8 KB 后, 立即自动请求得到下一块, 这样当真正需要下一块时, 很快就能得到, 这种称为预读的机制对提高性能是很有帮助的。写操作所采用的策略是类似的, 如果 write 系统调用所提供的数据少于 8 KB, 数据将只在本地累积, 只有当整个块已满后才送往服务器。当然, 当文件被关闭时, 所有的数据都将被送往服务器。

8.3.7 分布式进程迁移

在计算机网络中, 允许程序或数据从一个节点迁移到另一个节点。在分布式系统中, 更是允许将一个进程从一个系统迁移到另一个系统中。

1. 迁移的类型

(1) 数据迁移

假如系统 A 中的用户欲访问系统 B 中文件的数据, 可以采用两种方法来实现数据迁移(data migration)。第一种方法, 是将系统 B 中的整个文件送至系统 A, 这样, 凡是系统 A 中的用户要访

问此文件时,都成为本地访问,当用户不再需要此文件时,若文件副本已被修改,则必须把已修改过的副本送回系统 B;若未被修改,便不必将文件回送。如果文件比较大,系统 A 中用户所用到的文件数据又比较少,采用这种来回传送整个文件的方法,系统效率较低。

第二种方法,是仅把文件中用户当前要使用的部分从系统 B 传送到系统 A,若以后用户又要用到此文件中的另一部分,可继续将另一部分从系统 B 传送到系统 A。当用户不再需要使用此文件时,则只需把修改过的部分送回系统 B。

(2) 计算迁移

在某些情况下,传送计算要比传递数据的效率高。例如,一个应用程序需要访问多个驻留在不同系统中的大型文件,以获得有关数据,此时,若采用数据迁移方式,便必须将驻留在不同系统上的所需文件传送到应用程序驻留的系统中,这样,要传送的数据量相当大,可采用计算迁移 (computation migration) 来解决这个问题。

计算迁移可以有多种不同的执行方式,可通过 RPC 调用不同系统上的例行程序来处理文件,并把处理后的结果传回给自己;也可发送多条消息给各个驻留文件的系统,这些机器上的操作系统将创建进程来处理相应的文件,进程处理完毕后再把结果传递回请求进程。需要注意的是,后一种方式中请求进程和执行请求的进程是在不同的机器上并发执行的。两种方法经过网络传输的数据都相当少,如果传输数据的时间长于这段命令的执行时间,则计算迁移方式更可取;反之,数据迁移方式更有效。

(3) 进程迁移

进程迁移 (process migration) 是计算迁移的一种延伸,当一个新进程被启动执行后,并不一定始终都在同一处理器上运行,也可被迁移到另一台机器上继续运行。出于下列原因需要引入进程迁移:

(1) 负载均衡

在分布式系统中,各个节点的负荷往往不均匀,可以通过进程迁移的方法来均衡各个系统的负荷,把重负荷系统中的进程迁移到轻负荷系统中去,以便改善系统性能。

(2) 通信性能

对于分布在不同系统中而彼此交互性又很强的一些进程,应将其迁移到同一系统中,以减少由于频繁交互而加大通信开销。类似地,某进程在执行数据分析时,如果它们所需要的文件远远大于进程,则此时应该把进程迁移到文件所驻留的系统中去,以便进一步降低通信开销。

(3) 加速计算

对于一个大型应用,如果始终在一个处理器上执行,可能要花费较多时间,使作业周转时间延长,但如果能为作业建立多个进程,并把这些进程迁移到多台处理器上工作,会大大加快作业的执行。

(4) 特殊功能和资源的使用

通过进程迁移来利用特殊节点上的硬件、软件功能或资源。此外,在分布式系统中,如果某个系统发生故障,而此系统中的进程又希望继续运行下去,则分布式操作系统可以把这些进程迁

移到其他系统中运行,提高系统的可用性。

2. 迁移的实现

为了实现进程迁移,在分布式系统中必须建立相应的进程迁移机制,主要负责解决:由谁来发动进程迁移;如何进行进程迁移;如何处理未完成的信号和消息等问题。

进程迁移的发动取决于进程迁移机制的目标,如果目标是平衡负载,则由系统中的监视模块负责在适当时刻进行进程迁移。在分布式系统中配置系统负荷监视模块,设定其中一个节点上的是主模块,主模块定时地与各个系统的监视模块交互有关系统负荷情况的信息。一旦发现有些系统忙碌,而有些系统空闲时,主模块便启动进程迁移,向负荷沉重的系统发出命令,让其将若干进程迁移到负荷轻的系统中去。当然,对于用户而言是透明的,进程迁移工作均由系统完成。类似地,如果进程迁移是为了其他目标,则分布式系统中的其他相应的部分成为进程迁移的发动者。

在进程进行迁移时,应撤销系统中的已迁移进程,在目标系统中建立一个相同的新进程,因为这是进程的迁移而不是进程的复制。进程迁移时,所迁移的是进程映像,包括进程控制块、程序、数据和栈,此外,被迁移进程与其他进程之间的关联应作相应的修改。

进程迁移的过程并不复杂,但是需要一定的通信开销,困难在于进程地址空间和已经打开的文件。由于现代操作系统均采用虚拟存储技术,对于进程地址空间可使用以下两种方法:一是传送整个地址空间,把进程的所有映像全部从源系统传递到目标系统,这种方法简单,但当地址空间很大且进程只需用到一部分程序和数据时,会造成资源浪费。二是仅传送主存中已修改的那部分地址空间,若程序运行时还需要附加的虚存空间部分信息,则可通过请求方式予以传送。这样,所传送的数据量是最少的,但源系统中仍然需要保存被迁移进程的数据及相关信息,源系统并未从对此进程的管理中解脱出来。三是预先复制,进程继续在原节点上运行,而地址空间被复制到目标节点上,由于原节点上的某些地址空间内容又被修改过,所以,需要有两次迁移,这种方法能够减少进程被冻结的时间。如果被迁移的进程已打开系统中的某些文件,可采用两种方法来处理,一是将已打开的文件随进程一起迁移,这里存在的问题是:进程有可能仅作临时迁移,返回时才需要访问文件;二是暂时不迁移文件,仅当迁移后的进程又提出对文件的访问要求时,再进行迁移。如果文件被多个分布式进程所共享,则需要维护对文件的分布式访问,而不必迁移。

在一个进程由源系统向目标系统迁移期间,可能会有其他进程继续向源系统中已迁移的进程发来消息或信号,这时应如何处理?一种可行的方法是在源系统中提供一种机制,用于暂时保存这类信息,还需要保存被迁移进程所在目标系统的新地址,当被迁移进程已在目标系统中被建成新进程后,源系统便可将接收到的相关信息转发至目标系统。

3. 进程迁移举例

IBM 公司的 AIX 是一种分布式 UNIX 操作系统,它提供一种实用的进程迁移机制。进程迁移的步骤如下:

- (1) 当进程决定迁移自身时,先选择一个目标机,发送一条远程执行任务的消息,此消息传送进程映像及打开文件的部分信息;

- (2) 在接收端,内核服务进程生成一个子进程,将这些信息交给它;
- (3) 这个新进程收集完成其操作所需要的环境、数据、变量和栈信息,如果它是“脏”的就复制程序文件;如果是“干净”的,则请求从全局文件系统中调页。迁移完成之后,发送消息通知源进程,源进程就发送一条最后完成消息给新进程,然后撤销自己。

8.4 Linux 网络体系结构

Linux 网络体系结构如图 8.9 所示,采用四层的层次结构来实现 TCP/IP 协议族。

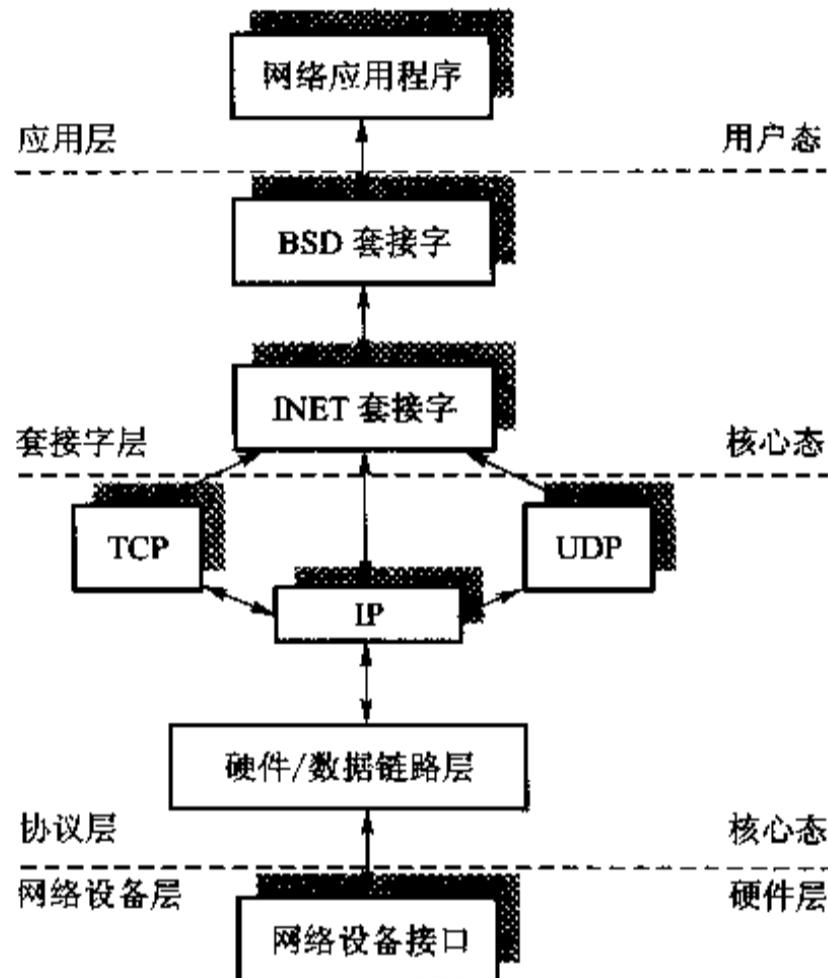


图 8.9 Linux 网络体系结构

BSD 套接字包含 BSD socket 编程接口的内容,有了这一层的支持,使用 UNIX 的 BSD socket 通用接口所编写的程序均可在 Linux 系统中运行。Linux 所支持的 BSD 套接字类型有:流(stream)、数据报(datagram)、原始型(raw)、可靠发送的消息(reliable delivered message)、顺序数据包(sequenced packet)和数据包(packet)。INET 套接字实现比 IP 高一级的管理,实现 IP 分组排序、网络效率控制等功能。TCP 协议为应用级过程提供面向连接的数据传输服务,数据包是编号的,传输两端都要确认数据包的正确性;UDP 协议为应用级过程提供面向无连接的数据传输服务;IP 协议是网络互连所实现的内容,要为上一层数据准备 IP 数据头,且决定如何把接收到的 IP 数据包传送到 TCP 协议层或 UDP 协议层。硬件/数据链路层包含设备驱动程序和硬件信息传输控制。网络设备是一个接收或发送数据包的实体,可以是以太网设备或点到点(PPP)设备。



5.5 Windows 2003 网络体系结构和网络服务

1. 网络构架组件

Windows 网络构架为网络 API、网络协议驱动程序和网络适配驱动程序提供一种灵活的基础设施。网络构架的各类组件包含：

(1) 网络 API：为应用程序提供独立于协议方式的网络通信，支持传统的应用，且兼容工业标准。网络 API 有：命名管道和邮件槽、套接字、远程过程调用和公共互连网络文件系统等。

(2) 传输驱动程序接口（Transmission Driver Interface, TDI）：TDI 客户（核心态的设备驱动程序）从发送至协议驱动程序的 I/O 请求分组（IRP）中获得自己的名称，这些 IRP 的格式符合传输驱动程序接口标准，为核心态设备驱动程序定义公共编程接口。

(3) TDI 传送器：这是一种核心态的协议驱动程序，接收从 TDI 客户端传来的 IRP，处理 IRP 中的请求，通过透明的消息操作，如分段与重组、序列化、确认和重传，TDI 传送器简化应用程序的网络通信。

(4) NDIS 库：为适配驱动程序提供封装，隐蔽核心态环境中的具体细节，为适配驱动程序提供支持函数，而且也为 TDI 传送器的使用提供函数接口。

(5) NDIS 小端口驱动程序：这是一种核心态驱动程序，负责将 TDI 传送器接入特定的网络适配器。

Windows 网络构架的组件与 OSI-RM 参考模型存在着对应关系，但并不精确，还有一些跨层的组件，例如，TDI 传送器便对应于传输（第四）层和网络（第三）层。

2. 层次化网络服务

Windows 中建立在 API 及组件之上的网络服务有：远程访问、活动目录、网络负载均衡、文件复制服务以及分布式文件系统等。另外，Windows 支持多种基于 TCP/IP 协议的扩展特性的服务，包括网络地址转换（NAT）、网际协议安全性以及服务质量控制。



本 章 小 结

计算机网络是计算机与通信技术相结合的产物，随着微型计算机性能的不断提高和网络技术的飞速发展，很容易通过高速网络把许多计算机连接起来构成一个大型计算机系统。仅管理单台计算机的操作系统称为集中式操作系统，而不仅管理自身所在机器还具有联网通信及相关功能的操作系统称为网络或分布式操作系统。

网络操作系统是基于松散耦合的计算机上的松散耦合软件，配置网络操作系统是为了管理网络中的共享资源，实现网络用户的通信和方便用户对网络的使用，可把它看做网络用户和网络系统之间的接口。计算机网络系统中除了硬件，最重要的是要配置网络操作系统，它除了具备通常集中式操作系统所应有的基本功能外，还应增加联网功能，支持网络体系结构和各种网络通信

协议,提供网络互连能力,支持有效、可靠、安全地传输数据。网络管理的主要功能有:域管理、网络文件管理、网络资源管理、网络服务等。

分布式操作系统是基于松散耦合的计算机上的紧密耦合软件,它应该具备4项基本功能:进程通信、资源共享、并行计算和网络管理,以此为用户提供一台具有分布处理功能的虚拟机。分布式操作系统的实现主要涉及:分布式进程通信、分布式资源管理、分布式进程同步、分布式文件系统、分布式进程迁移和分布式死锁。

分布式进程通信源于分布式系统的分布性,分布性源于应用的需求,而通信则来源于分布性,因为机器的分散而必须通过通信来实现进程之间的交互和合作。进程通信是分布式系统的关键机制,它可分成三种:一是消息传递机制;二是远程过程调用;三是套接字。消息传递机制类似于单机系统中的发送消息和接收消息操作,涉及目标进程的寻址、同步和异步通信、可靠和非可靠通信原语、缓冲和非缓冲通信原语等;远程过程调用是在分布式系统中广泛采用的进程通信方法,它把单处理器环境下的过程调用拓展到分布式系统环境中,允许不同计算机上的进程使用简单的过程调用和返回结果的方式进行交互,但这个过程调用是用来访问远程计算机所提供的服务的,调用者和被调用者就好像运行在同一台机器上一样;套接字提供进程通信的端点,其实现原理与电话通信十分相似,利用客户-服务器模式巧妙地解决进程之间的通信问题。为了解决客户-服务器模式中的互操作性,中间件已得到广泛的应用。

分布式操作系统采用一类资源由多个管理者来管理的方式,可分为两种:集中分布管理和完全分布管理。两者的主要区别在于:前者对所管理的资源拥有完全控制权,一类资源中的每一个资源仅受控于一个资源管理者;而后者对所管理的资源仅有部分控制权,不仅一类资源存在多个管理者,而且此类资源中的每个资源都由多个管理者共同控制。集中分布管理有3种资源搜索算法:投标算法、回声算法和由近及远算法。完全分布管理较为复杂,在分布式系统中,各台计算机相互分散,未共享主存储器,因而,需要设计出适用的分布式同步算法。Lamport提出不使用物理时钟而对分布式系统中所发生的事件进行排序的方法,定义逻辑时钟,基于逻辑时钟实现了Lamport、Ricart 和令牌分布式进程同步算法。分布式进程同步和分布式死锁处理均比集中式操作系统来得复杂。

分布式操作系统支持分布式进程迁移,需要把进程从一台机器转移到另一台机器上,其目的是使进程能够在目标机器上运行。进程迁移用于均衡系统负荷、降低通信开销、加快应用计算。为了实现进程迁移,必须建立相应的进程迁移机制,主要负责解决:由谁来发动进程迁移;如何进行进程迁移;如何处理未完成的信号和消息等问题。

习 题 八

一、思考题

1. 什么是计算机网络? 它有哪些组成部分?

2. 试述计算机网络系统的主要组成。
3. 试述计算机网络的主要功能。
4. 计算机网络发展至今已有四代, 试述每代的主要标志。
5. 如何对计算机网络进行分类? 对每种类型作简单描述。
6. 什么是通信协议? 说明其所包含的 3 个要素。
7. 说明开放系统互连参考模型 OSI/RM。
8. 说明在层次式结构的网络中, 数据通信时信息的流动过程。
9. 说明 TCP/IP 网络体系结构。
10. 试述网络操作系统的主要特征和分类。
11. 试述网络操作系统与网络管理和控制有关的功能。
12. 试述分布式计算机系统所应满足的基本条件。
13. 分布式系统与网络系统之间的主要区别是什么?
14. 分布式操作系统应具有哪些基本功能?
15. 什么是分布式文件系统? 试述其组成。
16. 试述分布式操作系统的特性及优点。
17. 什么是 RPC? 说明其工作原理。
18. 什么是 socket? 说明其工作原理。
19. 什么是中间件? 说明其在分布式系统中的作用。
20. 试列举目前广为使用的一些中间件。
21. 试述分布式系统中的集中分布资源管理方法。
22. 试述分布式系统中的完全分布资源管理方法。
23. 试述集中分布资源管理算法: 投标算法、由近及远算法和回声算法。
24. 实现分布式进程同步的主要困难表现在哪些方面?
25. 什么是逻辑时钟? 说明其用途。
26. 试述 Lamport 分布式同步算法。
27. 试述 Ricart 分布式同步算法。
28. 试述令牌分布式同步算法。
29. 对于令牌算法, 证明能够保证互斥和避免死锁。
30. 什么是数据迁移和计算迁移?
31. 为什么要进行进程迁移? 哪些情况要进行进程迁移?
32. 如何实现进程迁移? 对于已打开的文件应怎样处理?
33. 哪些情况会导致分布式系统死锁? 试述防止和检测方法。
34. 比较同步通信和异步通信的优劣。
35. 简述 Linux 网络体系结构的层次及其功能。
36. 简述 Windows 网络构架和各类组件。

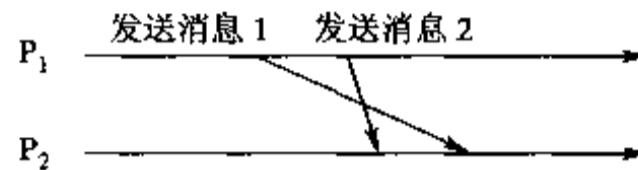
二、应用题

1. 在一个分布式系统中, 如果进程 P_1 的逻辑时钟在其向 P_2 发送消息时为 10, 进程 P_2 接收到来自 P_1 的消

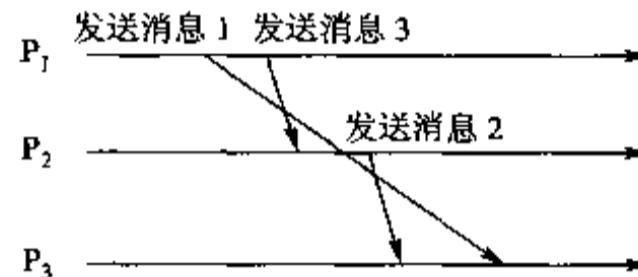
息时,逻辑时钟为 8,下一个局部事件 P_2 的逻辑时钟是多少? 如果接收消息时 P_2 的逻辑时钟为 16,下一个局部事件 P_2 的逻辑时钟是多少?

2. 在图 8.5(b)中,加入一条与消息 A 并发的新消息,而且它不发生在消息 A 之前,也不发生在消息 A 之后。

3. 由于在网络中可以通过不同的路由发送消息,因而,可能出现后发送的消息先到达的乱序情况(如下图所示)。试设计一种逻辑时钟算法对消息进行排序。



4. 试设计一种逻辑时钟算法,对于来自不同进程的乱序消息(如下图所示)进行排序。



参 考 文 献

- [1] STALLINGS W. Operating systems: internals and design principles[M]. 5th ed. Prentice – Hall International Inc, 2005.
- [2] NUTT G. Operating systems[M]. 3rd ed. Addison – Wesley, 2004.
- [3] SILBERSCHATZ A, GALVIN P B, GAGNE G. Operating systems concepts[M]. 6th ed. John Wiley & Sons Inc, 2002.
- [4] TANENBAUM A S. Modern operating systems[M]. 2nd ed. Prentice Hall, 2001.
- [5] 罗宇, 等. 操作系统[M]. 2 版. 北京: 电子工业出版社, 2007.
- [6] 陈向群, 杨美清. 操作系统教程[M]. 2 版. 北京: 北京大学出版社, 2006.
- [7] BIC L F, SHAW A C. 操作系统原理[M]. 梁洪亮, 等译. 北京: 清华大学出版社, 2005.
- [8] LOVE R. Linux 内核设计与实现[M]. 陈莉君, 康华, 张波, 译. 北京: 机械工业出版社, 2005.
- [9] 张效祥, 等. 计算机科学技术百科全书[M]. 2 版. 北京: 清华大学出版社, 2005.
- [10] 孟庆昌. 操作系统[M]. 北京: 电子工业出版社, 2004.
- [11] 蒋静, 徐志伟. 操作系统原理·技术与编程[M]. 北京: 机械工业出版社, 2004.
- [12] 卿斯汉, 刘文清, 温红子. 操作系统安全[M]. 北京: 清华大学出版社, 2004.
- [13] 刘克龙, 冯登国, 石文昌. 安全操作系统原理与技术[M]. 北京: 科学出版社, 2004.
- [14] 尤晋元, 史美林, 等. Windows 操作系统原理[M]. 北京: 机械工业出版社, 2001.
- [15] 毛德操, 等. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社, 2002.
- [16] 汤子瀛, 哲凤屏, 汤小丹. 计算机操作系统[M]. 2 版. 西安: 西安电子科技大学出版社, 2001.
- [17] 尤晋元. UNIX 操作系统教程[M]. 西安: 西安电子科技大学出版社, 2000.
- [18] 屠祁, 屠立德, 等. 操作系统基础[M]. 3 版. 北京: 清华大学出版社, 2001.

参 考 网 站

- [1] <http://www.kernel.org>.
- [2] <http://linux-mm.org>.

[General Information]

书名 = 普通高等教育“十一五”国家级规划教材 操作系统教程 (第四版)

作者 =

页数 = 510

S S 号 = 11982518

出版日期 =

[封面](#)
[书名](#)
[版权](#)
[前言](#)
[目录](#)
[正文](#)