

面向 21 世 纪 课 程 教 材
Textbook Series for 21st Century

操作系 统 教 程

孙 钟 秀 主 编

费 翔 林 骆 斌 谢 立 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。本书在前两版的基础上进行了全面修订,系统地介绍了操作系统的经典内容和最新发展,选择当代具有代表性的 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。

本书共分八章,覆盖了操作系统的基本概念、基本方法、设计原理和实现技术,尽可能系统、清晰、全面、综合地展示操作系统的概念、特性和精髓。力求做到:概念清晰、结构合理;取材得当、启发思考。为便于教学,与教材相配套提供了 ppt 讲稿(可以从高教出版社网站 www.hep.edu.cn 上下载获得),同时各章配有丰富的思考题和应用题。

本书既可作为高等院校计算机科学与技术专业本科教材或参考书,也可供计算机等级考试、水平考试的考生以及计算机技术和软件开发人员阅读参考。

第三版前言

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。为了适应这种发展的趋势,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。全书涉及的内容有:

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,也涉及到多处理机调度算法和实时调度算法。实例研究讨论了 Window2000/XP、Solaris 和 Linux 中断处理,UNIX SVR4、Windows2000/XP 和 Linux 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,讨论并发程序设计有关技术和各种进程互斥、同步、通信机制和工具。最后介绍了 Windows2000/XP 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。实例研究深入介绍了 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 Windows2000/XP 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:内存映射文件和虚拟文件系统。实例研究介绍了 Windows2000/XP 文件管理和 Linux 文件管理。

第七章操作系统安全性。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,其中着重讨论了身份认证机制、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, 衷心希望读者指正赐教, 联系 email 为: feixl@nju.edu.cn 及 luobin@nju.edu.cn。

作者

2003 年 3 月

第二版前言

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 50 年代末,至今已有三十余年。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。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,操作系统的新的概念、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根本原理和概念,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、第九两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,恳请读者批评指正。

编著者

1994 年 12 月于南京

第一版前言

操作系统是计算机系统的一个重要组成部分,它出现在 50 年代末,至今已有三十余年。自 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根本功能: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、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。

编著者
1987 年 2 月

目 录

| | |
|----------------------------------------------|---------|
| 第一章 操作系统概论 | (1) |
| 1.1 操作系统概观 | (1) |
| 1.1.1 操作系统的定义和目标 | (1) |
| 1.1.2 操作系统的作用与功能 | (2) |
| 1.1.3 操作系统的主要特性 | (5) |
| 1.2 操作系统的形成和发展 | (8) |
| 1.2.1 人工操作阶段 | (8) |
| 1.2.2 管理程序阶段 | (9) |
| 1.2.3 多道程序设计与操作系统 的形成 | (10) |
| 1.2.4 操作系统的发展与分类 | (15) |
| 1.3 操作系统提供的服务和用户接口 | (25) |
| 1.3.1 操作系统提供的基本服务 | (25) |
| 1.3.2 操作系统提供的用户接口 | (26) |
| 1.3.3 程序接口与系统调用 | (26) |
| 1.3.4 操作接口与系统程序 | (30) |
| 1.4 操作系统的结构设计 | (35) |
| 1.4.1 操作系统的构件 | (36) |
| 1.4.2 整体式结构的操作系统 | (41) |
| 1.4.3 层次式结构的操作系统 | (42) |
| 1.4.4 虚拟机结构的操作系统 | (44) |
| 1.4.5 客户/服务器与微内核结构的 操作系统 | (45) |
| 1.4.6 操作系统的运行模型 | (49) |
| 1.4.7 实例研究:Windows 2000/XP 客户/服务器结构 | (51) |
| 1.5 流行操作系统简介 | (57) |
| 1.5.1 DOS 操作系统 | (57) |
| 1.5.2 Windows 操作系统 | (58) |
| 1.5.3 UNIX 操作系统家族 | (60) |
| 1.5.4 自由软件和 Linux 操作系统 | (63) |
| 1.5.5 IBM 系列操作系统 | (65) |
| 1.5.6 其他流行操作系统 | (68) |
| 1.6 本章小结 | (70) |
| 习题一 | (72) |
| 第二章 处理器管理 | (76) |
| 2.1 中央处理器 | (76) |
| 2.1.1 单处理器系统和多处理器 系统 | (76) |
| 2.1.2 寄存器 | (78) |
| 2.1.3 特权指令与非特权指令 | (78) |
| 2.1.4 处理器状态 | (79) |
| 2.1.5 程序状态字寄存器 | (80) |
| 2.2 中断技术 | (82) |
| 2.2.1 中断的概念 | (82) |
| 2.2.2 中断源分类 | (83) |
| 2.2.3 中断装置 | (84) |
| 2.2.4 中断处理程序 | (86) |
| 2.2.5 中断事件的具体处理方法 | (87) |
| 2.2.6 中断的优先级和多重中断 | (93) |
| 2.2.7 实例研究:Windows 2000/XP 中断处理 | (95) |
| 2.2.8 实例研究:Solaris 中断处理 | (103) |
| 2.2.9 实例研究:Linux 中断处理 | (105) |
| 2.3 进程及其实现 | (113) |
| 2.3.1 进程的定义和属性 | (113) |
| 2.3.2 进程的状态和转换 | (115) |
| 2.3.3 进程的描述 | (118) |
| 2.3.4 进程切换与模式切换 | (123) |
| 2.3.5 进程的控制 | (125) |

| | | | |
|----------------------------------------------|-------|---------------------------------------------------------|-------|
| 2.3.6 实例研究:UNIX SVR4 进程管理 | (129) | 习题二 | (201) |
| 2.3.7 实例研究:Linux 进程管理 | (133) | 第三章 并发进程 | (208) |
| 2.4 线程及其实现 | (137) | 3.1 并发进程 | (208) |
| 2.4.1 引入多线程技术的动机 | (137) | 3.1.1 顺序程序设计 | (208) |
| 2.4.2 多线程环境中的进程与线程 | (138) | 3.1.2 进程的并发性 | (209) |
| 2.4.3 线程的实现 | (144) | 3.1.3 与时间有关的错误 | (212) |
| 2.4.4 实例研究:Solaris 的进程 与线程 | (148) | 3.1.4 进程的交互(Interaction Among Processes):协作和竞争 | (213) |
| 2.4.5 实例研究:Windows 2000/XP 的进程与线程 | (153) | 3.2 临界区管理 | (215) |
| 2.5 处理器调度 | (162) | 3.2.1 互斥和临界区 | (215) |
| 2.5.1 处理器调度的层次 | (162) | 3.2.2 临界区管理的尝试 | (216) |
| 2.5.2 高级调度(High Level Scheduling) | (164) | 3.2.3 实现临界区管理的软件方法 | (218) |
| 2.5.3 中级调度(Medium Level Scheduling) | (165) | 3.2.4 实现临界区管理的硬件设施 | (221) |
| 2.5.4 低级调度(Low Level Scheduling) | (165) | 3.3 信号量与 PV 操作 | (222) |
| 2.5.5 选择调度算法的原则 | (165) | 3.3.1 同步和同步机制 | (222) |
| 2.6 批处理作业的管理与调度 | (167) | 3.3.2 记录型信号量与 PV 操作 | (224) |
| 2.6.1 作业和进程的关系 | (167) | 3.3.3 用记录型信号量实现互斥 | (227) |
| 2.6.2 批处理作业的管理 | (167) | 3.3.4 记录型信号量解决生产者— 消费者问题 | (230) |
| 2.6.3 批处理作业的调度 | (169) | 3.3.5 记录型信号量解决读者— 写者问题 | (233) |
| 2.6.4 作业调度算法 | (169) | 3.3.6 记录型信号量解决理发师 问题 | (235) |
| 2.7 低级调度 | (173) | 3.4 管程 | (236) |
| 2.7.1 低级调度的功能 | (173) | 3.4.1 管程和条件变量 | (236) |
| 2.7.2 低级调度算法 | (173) | 3.4.2 Hoare 方法实现管程 | (241) |
| 2.7.3 实时调度 | (178) | 3.4.3 Hanson 方法实现管程 | (244) |
| 2.7.4 多处理器调度 | (179) | 3.5 进程通信 | (252) |
| 2.7.5 实例研究:UNIX SVR4 调度 算法 | (183) | 3.5.1 信号通信机制 | (252) |
| 2.7.6 实例研究:Windows 2000/XP 调度算法 | (185) | 3.5.2 共享文件通信机制 | (254) |
| 2.7.7 实例研究:Linux 调度算法 | (195) | 3.5.3 共享存储区通信机制 | (257) |
| 2.8 本章小结 | (199) | 3.5.4 消息传递通信机制 | (258) |
| | | 3.5.5 有关消息传递实现的若干 问题 | (261) |
| | | 3.6 死锁 | (266) |
| | | 3.6.1 死锁的产生 | (266) |

| | | | |
|---------------------------------------------|-------|--------------------------------------------------|-------|
| 3.6.2 死锁的定义 | (268) | 4.5 虚拟存储管理 | (328) |
| 3.6.3 死锁的防止 | (268) | 4.5.1 虚拟存储器的概念 | (328) |
| 3.6.4 死锁的避免 | (270) | 4.5.2 请求分页虚拟存储管理 | (330) |
| 3.6.5 死锁的检测和解除 | (280) | 4.5.3 请求分段虚拟存储管理 | (350) |
| 3.7 实例研究:Windows 2000/XP 的同步 和通信机制 | (283) | 4.5.4 请求段页式虚拟存储管理 | (352) |
| 3.7.1 Windows 2000/XP 的同步和 互斥机制 | (283) | 4.6 实例研究:Intel X86/Pentium 存储管理硬件设施 | (354) |
| 3.7.2 Windows 2000/XP 进程通信 机制 | (284) | 4.6.1 Intel x86/Pentium 段机制 ——段选择符和段描述符 | (354) |
| 3.8 实例研究:Linux 信号量机制 | (285) | 4.6.2 Intel x86/Pentium 运行模式 选择 | (356) |
| 3.9 本章小结 | (289) | 4.6.3 Intel x86/Pentium 地址转换 | (356) |
| 习题三 | (291) | 4.6.4 Intel x86/Pentium 页式或段页式 地址转换 | (358) |
| 第四章 存储管理 | (304) | 4.7 实例研究:Windows 2000/XP 虚拟 存储管理 | (360) |
| 4.1 存储器 | (304) | 4.7.1 进程地址空间布局 | (360) |
| 4.1.1 存储器的层次 | (304) | 4.7.2 用户空间内存分配 | (361) |
| 4.1.2 快速缓存 | (305) | 4.7.3 内存管理的实现 | (366) |
| 4.1.3 地址转换与存储保护 | (306) | 4.8 实例研究:Linux 虚拟存储管理 | (373) |
| 4.2 连续存储空间管理 | (307) | 4.8.1 Linux 虚拟存储管理概述 | (373) |
| 4.2.1 单用户连续存储管理 | (307) | 4.8.2 Linux 进程的虚拟地址空间 | (373) |
| 4.2.2 固定分区存储管理 | (309) | 4.8.3 Linux 物理内存空间的管理 | (376) |
| 4.2.3 可变分区存储管理 | (311) | 4.8.4 用户态内存的申请与释放 | (378) |
| 4.3 分页式存储管理 | (317) | 4.8.5 内存的共享和保护 | (379) |
| 4.3.1 分页式存储管理的基本原理 | (317) | 4.8.6 交换空间、页面换出和调入 | (379) |
| 4.3.2 相联存储器和快表 | (319) | 4.8.7 缓冲机制 | (381) |
| 4.3.3 分页式存储空间的分配 和去配 | (320) | 4.9 本章小结 | (382) |
| 4.3.4 分页存储空间的页面共享 和保护 | (321) | 习题四 | (384) |
| 4.3.5 多级页表 | (322) | 第五章 设备管理 | (391) |
| 4.3.6 反置页表 | (324) | 5.1 I/O 硬件原理 | (391) |
| 4.4 分段式存储管理 | (325) | 5.1.1 I/O 系统 | (392) |
| 4.4.1 程序的分段结构 | (325) | 5.1.2 I/O 控制方式 | (392) |
| 4.4.2 分段式存储管理的基本原理 | (325) | 5.1.3 设备控制器 | (397) |
| 4.4.3 段的共享 | (327) | 5.2 I/O 软件原理 | (399) |
| 4.4.4 分段和分页的比较 | (327) | 5.2.1 I/O 软件的设计目标和原则 | (399) |

| | | | |
|---------------------------------------------|-------|----------------------------------------|-------|
| 5.2.2 I/O 中断处理程序 | (400) | 程序 | (440) |
| 5.2.3 设备驱动程序 | (402) | 5.8.4 Windows 2000/XP I/O 处理 | (444) |
| 5.2.4 与硬件无关的操作系统 I/O 软件 | (403) | 5.8.5 Windows 2000/XP 高速缓存 管理 | (447) |
| 5.2.5 用户空间的 I/O 软件 | (404) | 5.9 实例研究:Linux 设备管理 | (462) |
| 5.3 具有通道的 I/O 系统管理 | (405) | 5.9.1 Linux 设备管理概述 | (462) |
| 5.3.1 通道命令和通道程序 | (405) | 5.9.2 Linux 硬盘管理 | (464) |
| 5.3.2 I/O 指令和主机 I/O 程序 | (407) | 5.9.3 Linux 网络设备 | (465) |
| 5.3.3 通道启动和 I/O 操作过程 | (409) | 5.9.4 Linux 设备驱动程序 | (465) |
| 5.4 缓冲技术 | (410) | 5.10 本章小结 | (466) |
| 5.4.1 单缓冲 | (411) | 习题五 | (468) |
| 5.4.2 双缓冲 | (411) | 第六章 文件管理 | (472) |
| 5.4.3 多缓冲 | (412) | 6.1 文件 | (473) |
| 5.5 驱动调度技术 | (413) | 6.1.1 文件的概念 | (473) |
| 5.5.1 存储设备的物理结构 | (413) | 6.1.2 文件的命名 | (473) |
| 5.5.2 循环排序 | (415) | 6.1.3 文件的类型 | (474) |
| 5.5.3 优化分布 | (416) | 6.1.4 文件的属性 | (475) |
| 5.5.4 交替地址 | (417) | 6.1.5 文件的存取方法 | (477) |
| 5.5.5 搜索定位 | (417) | 6.1.6 文件的使用 | (478) |
| 5.5.6 独立磁盘冗余阵列 | (420) | 6.2 文件目录 | (479) |
| 5.5.7 提高磁盘 I/O 速度的一些 方法 | (423) | 6.2.1 文件目录与文件目录项 | (479) |
| 5.6 设备分配 | (424) | 6.2.2 一级目录结构 | (480) |
| 5.6.1 设备独立性 | (424) | 6.2.3 二级目录结构 | (481) |
| 5.6.2 设备分配 | (425) | 6.2.4 树形目录结构 | (482) |
| 5.7 虚拟设备 | (427) | 6.3 文件组织与数据存储 | (484) |
| 5.7.1 问题的提出 | (427) | 6.3.1 文件的存储 | (484) |
| 5.7.2 Spooling 的设计和实现 | (428) | 6.3.2 文件的逻辑结构 | (484) |
| 5.7.3 Spooling 应用例子 | (430) | 6.3.3 文件的物理结构 | (490) |
| 5.8 实例研究:Windows 2000/XP 的 I/O 系统 | (431) | 6.4 文件系统其他功能的实现 | (496) |
| 5.8.1 Windows 2000/XP I/O 系统结构 和组件 | (431) | 6.4.1 文件系统调用的实现 | (496) |
| 5.8.2 Windows 2000/XP I/O 系统的 数据结构 | (437) | 6.4.2 UNIX 文件系统调用 | (498) |
| 5.8.3 Windows 2000/XP 设备驱动 | | 6.4.3 文件卷的安装和使用 | (504) |
| | | 6.4.4 文件共享 | (507) |
| | | 6.4.5 层次式文件系统模型 | (512) |
| | | 6.4.6 辅存空间管理 | (513) |
| | | 6.4.7 内存映射文件 | (515) |

| | | | |
|-------------------------------------------------|-------|------------------------------------------|-------|
| 6.4.8 虚拟文件系统 | (517) | 7.5.2 病毒的特性 | (572) |
| 6.5 实例研究:Linux 文件管理 | (519) | 7.5.3 病毒的类型 | (572) |
| 6.5.1 Linux 文件管理概述 | (519) | 7.5.4 反病毒的方法 | (573) |
| 6.5.2 Linux 文件系统安装 | (520) | 7.5.5 电子邮件病毒 | (575) |
| 6.5.3 虚拟文件系统 VFS | (522) | 7.6 保护的基本机制、策略与模型 | (575) |
| 6.5.4 文件系统管理的缓冲机制 | (527) | 7.6.1 机制、策略与模型 | (575) |
| 6.5.5 系统打开文件表和主要文件 操作 | (529) | 7.6.2 身份认证机制 | (592) |
| 6.5.6 EXT2 文件系统 | (532) | 7.6.3 授权机制 | (597) |
| 6.6 实例研究:WINDOWS 2000/XP 文件系统 | (534) | 7.6.4 加密机制 | (610) |
| 6.6.1 Windows 2000/XP 文件系统 概述 | (534) | 7.6.5 审计机制 | (615) |
| 6.6.2 Windows 2000/XP 文件系统模型和 FSD 体系结构 | (536) | 7.7 实例研究:Windows 2000/XP 的 安全机制 | (616) |
| 6.6.3 NTFS 文件系统驱动程序 | (540) | 7.7.1 Windows 2000/XP 安全性概述 | (616) |
| 6.6.4 NTFS 在磁盘上的结构 | (542) | 7.7.2 Windows 2000/XP 安全性系统 组件 | (617) |
| 6.6.5 NTFS 的实现机制 | (545) | 7.7.3 Windows 2000/XP 保护对象 | (617) |
| 6.6.6 NTFS 可恢复性支持 | (549) | 7.7.4 访问控制策略 | (618) |
| 6.6.7 NTFS 安全性支持 | (550) | 7.7.5 访问令牌 | (619) |
| 6.7 本章小结 | (551) | 7.7.6 安全描述符 | (619) |
| 习题六 | (552) | 7.8 本章小结 | (622) |
| 第七章 操作系统的安全与保护 | (556) | 习题七 | (623) |
| 7.1 安全性概述 | (556) | 第八章 网络和分布式操作系统 | (626) |
| 7.2 安全威胁及其类型 | (557) | 8.1 计算机网络概述 | (626) |
| 7.3 保护 | (561) | 8.1.1 计算机网络的概念 | (626) |
| 7.3.1 操作系统保护层次 | (561) | 8.1.2 数据通信基本概念 | (629) |
| 7.3.2 内存储器的保护 | (561) | 8.1.3 网络体系结构 | (631) |
| 7.3.3 面向用户的访问控制 | (562) | 8.2 网络操作系统 | (638) |
| 7.3.4 面向数据的访问控制 | (562) | 8.2.1 网络操作系统概述 | (638) |
| 7.4 入侵者 | (563) | 8.2.2 几个流行的网络操作系统 | (639) |
| 7.4.1 入侵技术 | (563) | 8.2.3 网络操作系统实例 | (640) |
| 7.4.2 口令保护 | (565) | 8.3 分布式操作系统 | (643) |
| 7.4.3 入侵检测 | (568) | 8.3.1 分布式系统概述 | (643) |
| 7.5 病毒(恶意软件) | (570) | 8.3.2 分布式进程通信 | (644) |
| 7.5.1 病毒及其威胁 | (570) | 8.3.3 分布式资源管理 | (650) |
| | | 8.3.4 分布式进程同步 | (654) |
| | | 8.3.5 分布式系统中的死锁 | (665) |

| | | | |
|---------------------------------------------|-------|-----------------------------|-------|
| 8.3.6 分布式文件系统 | (667) | 网络服务 | (695) |
| 8.3.7 分布式进程迁移 | (673) | 8.5 实例研究:Linux 网络体系结构 | (700) |
| 8.4 实例研究:Windows 2000 网络体系 结构和网络服务 | (676) | 8.6 本章小结 | (701) |
| 8.4.1 Windows 2000 网络体系结构 ... | (676) | 习题八 | (703) |
| 8.4.2 Windows 2000 的层次化 | | 参考文献 | (706) |

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, 简称 OS)的出现、使用和发展是近四十余年来计算机软件的一个重大进展。尽管操作系统尚未有一个严格的定义,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,合理组织计算机工作流程和为用户有效使用计算机提供良好运行环境的一种系统软件。

计算机发展到今天,从个人机到巨型机,无一例外都配置一种或多种操作系统,操作系统已经成为现代计算机系统不可分割的重要组成部分,它为人们建立各种各样的应用环境奠定了重要基础。配置操作系统的主要目标可归结为:

- 方便用户使用。OS 通过提供用户与计算机之间的友善接口来方便用户使用。
- 扩大机器功能。OS 通过扩充改造硬件设施和提供新的服务来扩大机器功能。
- 管理系统资源。OS 有效管理好系统中所有硬件软件资源,使之得到充分利用。
- 提高系统效率。OS 合理组织好计算机的工作流程,以改进系统性能和提高系统效率。
- 构筑开放环境。OS 遵循有关国际标准来设计和构造,以构筑出一个开放环境。其含义主要是指:遵循有关国际标准(如开放的通信标准、开放的用户接口标准、开放的线程库标准等);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可互操作性。

计算机系统包括硬件和软件两个组成部分。硬件是所有软件运行的物质基础,软件能充分发挥硬件潜能和扩充硬件功能,完成各种系统及应用任务,两者互相促进、相辅相成、缺一不可。图 1-1 给出了一个计算机系统的软硬件层次结构。其中,每一层具有一组功能并提供相应的接口,接口对层内掩盖了实现细节,对层外提供了使用约定。

硬件层提供了基本的可计算性资源,包括处理器、寄存器、存储器,以及各种 I/O 设施和设备,这些设施和设备组成了计算机系统的硬件,它可以按照用户的需要接收与存储信息、进行数据处理和输出运算结果,是操作系统和软件赖以工作的基础。操作系统层通常是最

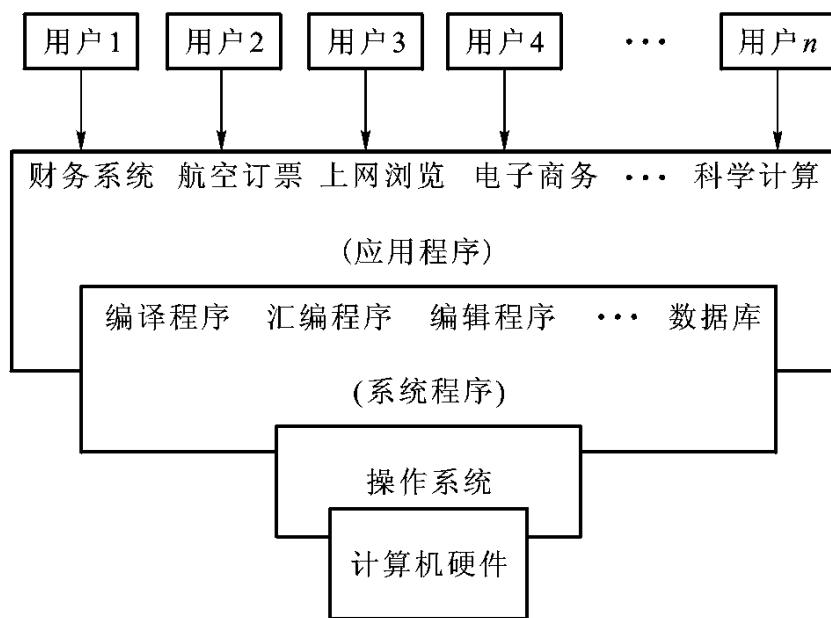


图 1-1 计算机系统的层次结构

靠近硬件的软件层,对计算机硬件作首次扩充和改造,主要完成资源的调度和分配,信息的存取和保护,并发活动的协调和控制等许多工作。操作系统是上层其他软件运行的基础,为编译程序和数据库管理系统等系统程序的设计者提供了有力支撑。系统程序层的工作基础建立在被操作系统改造和扩充过的机器上,利用操作系统提供的扩展指令集,可以较为容易地实现各种各样的语言处理程序、数据库管理系统和其他系统程序。此外,还提供种类繁多的实用程序,如连接装配程序、库管理程序、诊断排错程序、分类/合并程序等供用户使用。应用程序层解决用户特定的或不同应用需要的问题,应用程序开发者借助于程序设计语言来表达应用问题,开发各种应用程序,既快捷又方便。而最终用户则通过应用程序与计算机系统交互来解决他的应用问题。

1.1.2 操作系统的作用与功能

操作系统在计算机系统中的作用可以从三个方面来理解:

操作系统是用户与计算机硬件之间的接口。可以认为操作系统是对计算机硬件系统的第一次扩充,用户通过操作系统来使用计算机系统。换句话说,操作系统紧靠着计算机硬件并在其基础上提供了许多新的设施和能力,从而,使得用户能够方便、可靠、安全、高效地操纵计算机硬件和运行自己的程序。例如,改造各种硬件设施,使之更容易使用;提供原语和系统调用,扩展机器的指令系统,而这些功能到目前为止还难于由硬件直接实现。操作系统还能合理组织计算机的工作流程,协调各个部件有效工作,为用户提供一个良好的运行环境。经过操作系统改造和扩充过的计算机不但功能更强,使用也更为方便,用户可以直接调

用操作系统提供的各种功能,而无需了解许多软硬件本身的细节,对于用户来讲操作系统便成为他与计算机硬件之间的一个接口。

操作系统为用户提供了虚拟机(Virtual Machine)。许多年以前,人们就认识到必须找到某种方法把硬件的复杂性与用户隔离开来,经过不断的探索和研究,目前采用的方法是在计算机裸机上加上一层又一层的软件来组成整个计算机系统,同时,为用户提供一个容易理解和便于程序设计的接口。在操作系统中,类似地把硬件细节隐藏并把它与用户隔离开来的情况处处可见,例如,I/O管理软件、文件管理软件和窗口软件向用户提供了一个越来越方便的使用I/O设备的方法。由此可见,每当在计算机上覆盖了一层软件,提供了一种抽象,系统的功能便增加一点,使用就更加方便一点,用户可用的运行环境就更加好一点。所以,当计算机上覆盖了操作系统后,可以扩展基本功能,为用户提供了一台功能显著增强,使用更加方便,安全可靠性好,效率明显提高的机器,称为虚拟计算机,或操作系统虚拟机(Virtual Machine)。

操作系统是计算机系统的资源管理者。在计算机系统中,能分配给用户使用的各种硬件和软件设施总称为资源。资源包括两大类:硬件资源和信息资源。其中,硬件资源分为处理器、存储器、I/O设备等;I/O设备又分为输入型设备、输出型设备和存储型设备;信息资源则分为程序和数据等。操作系统的重要任务之一是对资源进行抽象研究,找出各种资源的共性和个性,有序地管理计算机中的硬件、软件资源,跟踪资源使用情况,监视资源的状态,满足用户对资源的需求,协调各程序对资源的使用冲突,研究使用资源的统一方法,为用户提供简单、有效的资源使用手段,最大限度地实现各类资源的共享,提高资源利用率,从而,使得计算机系统的效率有很大提高。

资源管理是操作系统的一项主要任务,而控制程序执行、扩充机器功能、提供各种服务、方便用户使用、组织工作流程、改善人机界面等等都可以从资源管理的角度去理解。下面就从资源管理的观点来看操作系统具有的几个主要功能。

1. 处理器管理

处理器管理的第一项工作是处理中断事件。硬件只能发现中断事件,捕捉它并产生中断信号,但不能进行处理,配置了操作系统,就能对中断事件进行处理。

处理器管理的第二项工作是处理器调度。处理器是计算机系统中一种稀有和宝贵的资源,应该最大限度地提高处理器的利用率。在单用户单任务的情况下,处理器仅为一个用户的一个任务所独占,处理器管理的工作十分简单。为了提高处理器的利用率,操作系统采用了多道程序设计技术。在多道程序或多用户的情况下,组织多个作业或任务执行时,就要解决处理器的调度、分配和回收等问题。近年来设计出各种各样的多处理器系统,处理器管理就更加复杂。为了实现处理器管理的功能,描述多道程序的并发执行,操作系统引入了进程(process)的概念,处理器的分配和执行都是以进程为基本单位;随着并行处理技术的发展,

为了进一步提高系统并行性,使并发执行单位的粒度变细,并发执行的代价降低,操作系统又引入了线程(thread)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,又分高级调度,中级调度和低级调度。

正是由于操作系统对处理器的管理策略不同,其提供的作业处理方式也就不同,例如,批处理方式、分时处理方式、实时处理方式等等。从而,呈现在用户面前,成为具有不同处理方式和不同特点的操作系统。

2. 存储管理

存储管理的主要任务是管理存储器资源,为多道程序运行提供有力的支撑,便于用户使用存储资源,提高存储空间的利用率。存储管理的主要功能包括:(1)存储分配。存储管理将根据用户程序的需要分配给它存储器资源,这是多道程序能并发执行的首要条件,当然程序运行结束时,还需回收存储资源。(2)存储共享。存储管理能让内存储器(又叫主存储器)中的多个用户程序实现存储资源的共享,以提高存储器的利用率。(3)地址转换与存储保护。存储管理负责把用户的逻辑地址转换成物理地址,同时要保证各个用户程序相互隔离起来互不干扰,更不允许用户程序访问操作系统的程序和数据,从而,保护系统和用户程序存放在存储器中的信息不被破坏。(4)存储扩充。由于受到处理器寻址能力的限制,一台计算机的物理内存容量总是有限的,难以满足用户大型程序的需求,而外存储器容量大且价格便宜。存储管理还应该能从逻辑上来扩充内存储器,把内存和外存混合起来使用,为用户提供一个比内存实际容量大得多的逻辑编程空间,方便用户的编程和使用。

操作系统的这一部分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使用需要,采用各种相应的有效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类外围设备,完成用户提出的I/O请求,加快I/O信息的传送速度,发挥I/O设备的并行性,提高I/O设备的利用率,以及提供每种设备的设备驱动程序和中断处理程序,为用户隐蔽硬件细节,提供方便简单的设备使用方法。为实现这些任务,设备管理应该具有以下功能:(1)提供外围设备的控制与处理;(2)提供缓冲区的管理;(3)提供设备独立性;(4)外围设备的分配和去配;(5)实现共享型外围设备的驱动调度;(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。文件管理则是针对系统中的信息资源的管理。在现代计算机中,通常把程序和数据以文件形式存储在外存储器(又叫辅存储器)上,供用户使用,这样,外存储器上保存了大量文件,对这些文件如不能采取良好的管理方式,就

会导致混乱或破坏,造成严重后果。为此,在操作系统中配置了文件管理,它的主要任务是对用户文件和系统文件进行有效管理,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;并提供给用户一套能方便使用文件的操作和命令。具体来说,文件管理要完成以下任务:(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)提供文件的存取方法;(4)提供文件的使用方法;(5)实现文件的目录管理;(6)实现文件的共享和存取控制;(7)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与通信技术的结合,近二十年来,从单机与终端之间的远程通信,到今天全世界成千上万台计算机联网工作,计算机网络的应用已十分广泛。联网操作系统至少应具有以下管理功能:(1)网上资源管理功能。计算机网络的主要目的之一是共享资源,网络操作系统应实现网上资源的共享,管理用户应用程序对资源的访问,保证信息资源的安全性和完整性。(2)数据通信管理功能。计算机联网后,结点之间可以互相传送数据,进行通信,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)网络管理功能。包括故障管理、安全管理、性能管理、记帐管理和配置管理等。

6. 用户接口

为了使用户能灵活、方便地使用计算机和系统功能,操作系统还提供了一组友好的使用其功能的手段称用户接口,它包括两大类:程序接口和操作接口。用户通过这些接口能方便地调用操作系统功能,有效地组织作业及其工作和处理流程,并使整个系统能高效地运行。

1.1.3 操作系统的主要特性

1. 并发性(**concurrence**)

并发性是指两个或两个以上的事件或活动在同一时间间隔内发生。操作系统是一个并发系统,并发性是它的重要特征,操作系统的并发性是指计算机系统中同时存在若干个运行着的程序,因此,它应该具有处理和调度多个程序同时执行的能力。内存中同时有多个用户程序,或内存中同时有操作系统程序和用户程序被启动交替、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部件和部件之间的相互等待,有效地改善系统资源的利用率,改进系统的吞吐率,提高系统效率。例如,一个程序等待 I/O 时,就出让 CPU,操作系统调度另一个程序占有 CPU 运行。这样一来,在程序等待 I/O 时, CPU 便不会空闲,使得多个 I/O 设备可同时在输入输出,也可使得设备 I/O 和 CPU 计算同时进行,这就是并发技术。

并发性虽然能有效改善系统资源的利用率,但却会引发一系列的问题,使操作系统的设
计和实现变得复杂化。如,怎样从一个运行程序切换到另一个运行程序?以什么样的策略

来选择下一个运行的程序？怎样将各个运行程序隔离开来，使之互不干扰，免遭对方破坏？怎样让多个运行程序互通消息和协作完成任务？怎样协调多个运行程序对资源的竞争？多个运行程序共享文件数据时，如何保证数据的一致性？操作系统必须具有控制和管理程序并发执行的能力，为了更好的解决上述问题，操作系统必须提供机制和策略来进行协调，以使各个并发进程能顺利推进，并获得正确的运行结果。另外，操作系统还要合理组织计算机的工作流程，协调各类硬软件设施工作，充分提高资源的利用率，充分发挥系统的并行性。

采用了并发技术的系统又称为多任务系统（multitasking system），计算机系统中，并发实际上是一个物理 CPU 在若干道程序之间多路复用，这样就可以实现运行程序之间的并发，以及 CPU 与 I/O 设备、I/O 设备与 I/O 设备之间的并行，并发性的实质是对有限物理资源强制行使多用户共享以提高效率。在多处理器系统中，程序的并发性不仅体现在宏观上，而且体现在微观上，这称为并行的。并行性（parallelism）是指两个或两个以上事件或活动在同一时刻发生。在多道程序环境下，平行性使多个程序同一时刻可在不同 CPU 上同时执行。而在分布式系统中，多台计算机的并存使程序的并发性得到了更充分的发挥，因为，同一时刻每台计算机上都可以有程序在执行。可见并行的事件或活动一定是并发的，但反之并发的事件或活动未必是并行的，并行性是并发性的特例，而并发性是并行性的扩展。由于并发技术的本质思想是：当一个程序发生事件（如等待 I/O）时出让其占用的 CPU 而由另一个程序运行，据此不难看出，实现并发技术的关键之一是如何对系统内的多个运行程序（进程）进行切换的技术。

2. 共享性（sharing）

共享性是操作系统的另一个重要特性。共享指计算机系统中的资源（包括硬件资源和信息资源）可被多个并发执行的用户程序和系统程序共同使用，而不是被其中某一个程序所独占。出于经济上的考虑，一次性向每个用户程序分别提供它所需的全部资源不但是浪费的，有时也是不可能的。现实的方法是让操作系统程序和多个用户程序共用一套计算机系统的所有资源，因而，必然会产生共享资源的需要。资源共享的方式可以分成两种：

第一种是互斥访问。系统中的某些资源如打印机、磁带机、卡片机，虽然它们可提供给多个程序使用，但在同一时间段内却只允许一个程序访问这些资源，即要求互相排斥地使用这些资源。当一个程序还在使用该资源时，其他欲访问该资源的程序必须等待，仅当占有者访问完毕并释放资源后，才允许另一个程序对该资源进行访问。这种同一时间内只允许一个程序访问的资源称临界资源，许多物理设备，以及某些数据和表格都是临界资源，它们只能互斥地被访问和共享。

第二种是同时访问。系统中还有许多资源，允许同一时间内多个程序对它们进行访问，这里“同时”是宏观上的说法，从微观上看多个程序访问资源仍然是交错的，只是这种交错访问的顺序对访问的结果没有影响罢了。典型的可供多个程序同时访问的资源是磁盘，各种

可重入程序也可被同时访问。

与共享性有关的问题是资源分配、信息保护、存取控制等,必须要妥善解决好这些问题。

共享性和并发性是操作系统两个最基本的特性,它们互为依存。一方面,资源的共享是因为程序的并发执行而引起的,若系统不允许程序并发执行,自然也就不存在资源共享问题。另一方面,若系统不能对资源共享实施有效管理,必然会影响到程序的并发执行,甚至程序无法并发执行,操作系统也就失去了并发性,导致整个系统效率低下。

3. 异步性(asynchronism)

操作系统的第三个特性是异步性,或称随机性。在多道程序环境中,允许多个进程并发执行,由于资源有限而进程众多,多数情况下,进程的执行不是一貫到底,而是“走走停停”。例如,一个进程在 CPU 上运行一段时间后,由于等待资源满足或事件发生,它被暂停执行,CPU 转让给另一个进程执行。系统中的进程何时执行?何时暂停?以什么样的速度向前推进?进程总共要化多少时间执行才能完成?这些都是不可预知的,或者说该进程是以异步方式运行的,其导致的直接后果是程序执行结果可能不惟一。异步性给系统带来了潜在的危险,有可能导致进程产生与时间有关的错误,可以说操作系统运行在随机的环境下。但只要运行环境相同,操作系统必须保证多次运行进程,都会获得完全相同的结果。

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;操作员发出命令或按按钮的时刻是随机的;程序运行发生错误或异常的时刻是随机的;各种各样硬件和软件中断事件发生的时刻是随机的等等,随机性并不意味着操作系统就无法控制资源的使用和程序的执行,操作系统内部产生的事件序列有许许多多可能,而操作系统的一个重要任务是必须确保捕捉任何一种随机事件,正确处理可能发生的随机事件,正确处理任何一种产生的随机事件序列,否则将会导致严重后果。

4. 虚拟性(virtual)

虚拟性是指操作系统中的一种管理技术,它是把物理上的一个实体变成逻辑上的多个对应物,或把物理上的多个实体变成逻辑上的一个对应物的技术。显然,前者是实际存在的而后者是虚构假想的,采用虚拟技术的目的是为用户提供易于使用、方便高效的操作环境。例如,在多道程序系统中,物理 CPU 可以只有一个,每次也仅能执行一道程序,但通过多道程序和分时使用 CPU 技术,宏观上有多个程序在执行,就好像有多个 CPU 在为各道程序工作一样,物理上的一个 CPU 变成了逻辑上的多个 CPU。Spooling 技术可把物理上的一台独占设备变成逻辑上的多台虚拟设备;窗口技术可把一个物理屏幕变成逻辑上的多个虚拟屏幕;通过时分或频分多路复用技术可以把一个物理信道变成多个逻辑信道;IBM 的 VM 技术把物理上的一台计算机变成逻辑上的多台计算机。虚拟存储器则是把物理上的多个存储器(主存和辅存)变成逻辑上的一个(虚存)的例子。

1.2 操作系统的形成和发展

1.2.1 人工操作阶段

从计算机诞生到 50 年代中期的计算机属于第一代计算机, 机器速度慢、规模小、外设少, 操作系统尚未出现。由程序员采用手工方式直接控制和使用计算机硬件, 程序员使用机器语言编程, 并将事先准备好的程序和数据穿孔在纸带或卡片上, 从纸带或卡片输入机将程序和数据输入计算机。然后, 启动计算机运行程序, 程序员可以通过控制台上的按钮、开关和氖灯来操纵和控制程序, 运行完毕, 取走计算输出的结果, 才轮到下一个用户上机。

随着时间的推移, 汇编语言产生了, 在汇编系统中, 数字操作码被记忆码代替, 程序按固定格式的汇编语言书写。系统程序员预先编制一个汇编程序, 它把用汇编语言书写的“源程序”解释成计算机能直接执行的机器语言格式的目标程序。随后, 一些高级程序设计语言相继出现, FORTRAN、ALGOL、和 COBOL 语言分别于 1956、1958 和 1959 年设计完成并投入使用, 进一步方便了编程。

执行时需要把汇编程序或编译系统以及源程序和数据, 都穿在卡片或纸带上, 然后, 再装入和执行。其大致过程为:

- (1) 人工把源程序用穿孔机穿在卡片或纸带上;
- (2) 将准备好的汇编程序或编译系统装入计算机;
- (3) 汇编程序或编译系统读入人工装在输入机上的穿孔卡片或穿孔带上的源程序;
- (4) 执行汇编过程或编译过程, 产生目标程序, 并输出到目标卡片迭或纸带;
- (5) 通过引导程序把装在输入机上的目标程序读入计算机;
- (6) 执行目标程序, 从输入机上读入人工装好的数据卡片或数据带上的数据;
- (7) 产生计算结果, 把执行结果从打印机上或卡片机上输出。

上述方式比直接用机器语言前进了一步, 程序易于编制和读取, 汇编程序或编译系统可执行存储、分配等辅助工作, 从而, 在一定程度上减轻了用户的负担。但是计算机的操作方式并没有多大改变, 仍然是在人工控制下进行程序的装入和执行。人工操作方式存在严重缺点:

- 用户独占资源。用户一个个、一道道的串行算题, 上机时独占了全机资源, 造成计算机资源利用率不高, 计算机系统效率低下。
- 人工干预较多。要求程序员装纸带或卡片、按开关、看氖灯等等。手工操作多了, 不但浪费处理机时间, 而且也极易发生差错。

- 计算时间拉长。由于数据的输入,程序的执行、结果的输出均是联机进行的,因而,每个用户从上机到下机的时间拉得非常长。

这种人工操作方式在慢速的计算机上还能容忍,但是随着计算机速度的提高,其缺点就更加暴露出来了。譬如,一个作业在每秒 1 万次的计算机上,需运行 1 个小时,作业的建立和人工干预化了 3 分钟,那么,手工操作时间占总运行时间的 5%;当计算机速度提高到每秒 10 万次时,作业运行时间仅需 6 分钟,而手工操作不会有太大变化,仍为 3 分钟,这时手工操作时间占了总运行时间的 50%。由此看出缩短手工操作和人工干预时间是十分必要的。此外,随着 CPU 速度迅速提高而 I/O 设备速度却提高不多,导致 CPU 与 I/O 设备之间的速度不匹配,矛盾越来越突出,需要妥善解决这些问题。

1.2.2 管理程序阶段

早期批处理系统借助于作业控制语言变革了计算机的手工操作方式。用户不再通过开关和按钮来控制计算机的执行,而是通过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,并把作业控制卡连同程序、数据一起提交给计算机的操作员,操作员收集到一批作业后一起把它们放到卡片机上输入计算机。计算机上则运行一个驻留在内存中的执行程序,以对作业进行自动控制和成批处理,自动进行作业转换以减少系统空闲和手工操作时间。其工作流程如下:执行程序将一批作业从纸带或卡片机输入到磁带上,每当一批作业输入完成后,执行程序自动把磁带上的第一个作业装入内存,并把控制权交给作业。当该作业执行完成后,执行程序收回控制权并再调入磁带上的第二个作业到内存执行。计算机在执行程序的控制下就这样连续地一个作业一个作业地执行,直至磁带上的作业全部做完。这种系统能实现作业到作业的自动转换,缩短作业的准备和建立时间,减少人工操作和干预,让计算机尽可能地连续运转。

早期的批处理系统中,一开始作业的输入和输出均是联机的,联机 I/O 的缺点是速度慢,I/O 设备和 CPU 仍然串行工作,CPU 时间浪费相当大,为此,在批处理中引进了脱机 I/O 技术。除主机外,另设一台辅机,该机仅与 I/O 设备打交道,不与主机连接。输入设备上的作业通过辅机输到磁带上,这叫脱机输入;主机负责从磁带上把作业读入内存执行,作业完成后,主机负责把结果输出到磁带上,这叫脱机输出;然后,由辅机把磁带上的结果信息在打印机上打印输出。这样一来,I/O 工作脱离了主机,辅机和主机可以并行工作,大大加快了程序的处理和数据的输入及输出,这称作脱机 I/O 技术,这比早期联机处理系统具有了处理能力。

为了发挥批处理系统的性能,缩短作业的准备和建立时间,驻留在内存工作的执行程序的功能得到了很大的扩充,进化到管理程序(resident monitor)。FMS(FORTRAN Monitor System)和IBSYS(IBM 7094 Monitor System)是这类系统的典型实例。管理程序的内存组织如图 1-2 所示,它的主要功能小结如下:

- 自动控制和处理作业流。管理程序把控制权传送给一个作业,当作业运行结束时,它又收回控制权,继续调度下一个作业执行,自动控制和处理作业流,减少了作业的准备和建立时间。作业流的自动控制和处理依靠作业控制语言(Job Control Language),因而,促进了作业控制语言的发展。作业控制语言是由一些描述作业控制过程的语句组成的,每个语句附有一行作业或作业步信息编码,并以穿孔卡的形式提供。例如,\$JOB 卡表示启动一个新作业;\$FTN 卡表示调用 FORTRAN 编译系统;\$ASM 卡表示调用汇编程序;\$LOAD 卡表示调用装配程序;\$DATA 卡指定数据;\$RUN 卡执行用户程序;\$END 卡表示一个作业结束。管理程序通过输入、解释并执行嵌入用户作业的作业控制卡规定的功能,就能自动地处理用户作业流。每个作业完成后,管理程序又自动地从输入机上读取下一个作业运行,直到整批作业处理结束。

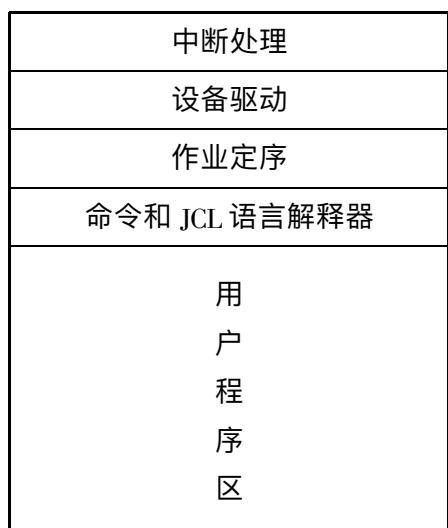


图 1-2 管理程序的内存组织

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,内存中仅有单个作业在运行,致使系统中仍有许多资源空闲,设备利用率低,系统性能较差。如图 1-3 所示,当 CPU 工作时,外部设备不能工作;而

外部设备工作时, CPU 必须等待。

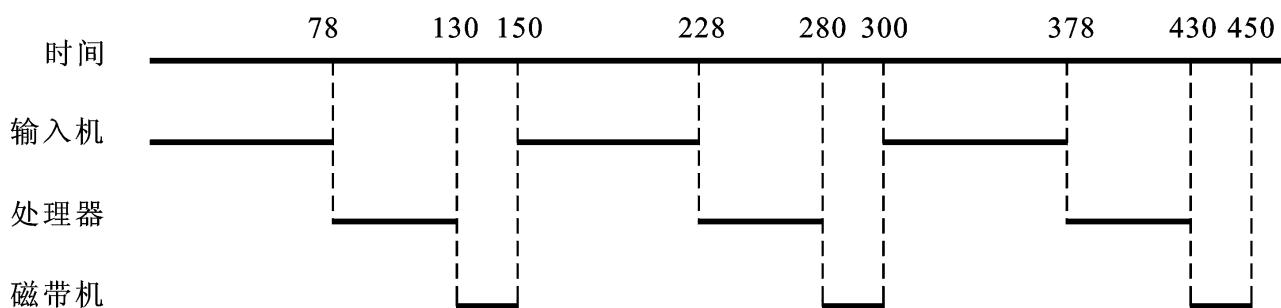


图 1-3 单道算题运行时处理器的使用效率

20 世纪 60 年代初, 有两项技术取得了突破: 中断和通道, 这两种技术结合起来为实现 CPU 和 I/O 设备的并行工作提供了基础, 这时, 多道程序的概念才变成了现实。

多道程序设计 (multiprogramming) 是指允许多个程序 (作业) 同时进入一个计算机系统的内存储器并启动进行交替计算的方法。也就是说, 计算机内存中同时存放了多道 (二个以上相互独立的) 程序, 它们均处于开始和结束点之间。从宏观上看是并行的, 多道程序都处于运行过程中, 但都未运行结束; 从微观上看是串行的, 各道程序轮流占用 CPU, 交替地执行。引入多道程序设计技术的根本目的是提高 CPU 的利用率, 充分发挥计算机系统部件的并行性, 现代计算机系统都采用了多道程序设计技术。

下面来分析多道程序设计技术提高资源利用率和系统吞吐率的原理。从第二代计算机开始, 计算机系统具有处理器和外围设备并行工作的能力, 这使得计算机的效率有所提高。但是, 仅仅这样做, 计算机的效率仍不会很高。例如, 计算某个数据处理问题, 要求从输入机 (速度为 6 400 字符/秒) 输入 500 个字符, 经处理 (费时 52 ms) 后, 将结果 (假定为 2 000 个字符) 存到磁带上 (磁带机速度为 10 万字符/秒), 然后, 再读 500 个字符处理, 直至所有的输入数据全部处理完毕。如果处理器不具有和外围设备并行工作的能力, 那么, 上述计算过程如图 1-3 所示, 不难看出在这个计算过程中, 处理器的利用率为:

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

分析上面的例子, 可以看出效率不高的原因, 当输入机输入 500 个字符后, 处理器只花了 52 ms 就处理完了, 而这时第二批输入数据还要再等 98 ms 时间才能输入完毕, 在此期间 CPU 一直空闲着。

这个例子说明单道程序工作时, 计算机系统的各部件的利用率没有得到充分发挥。为了提高效率, 考虑让计算机同时接受两道算题, 当第一道的程序在等待外围设备的时候, 让第二道的程序运行, 以降低 CPU 空闲等待时间, 那么, 处理器的利用率显然可以有所提高。例如, 计算机在接受上述算题时还接受了另一道算题: 从另一台磁带机上输入 2 000 个字符, 经 42 ms 的处理后, 从行式打印机 (速度为 1 350 行/分) 上输出两行。

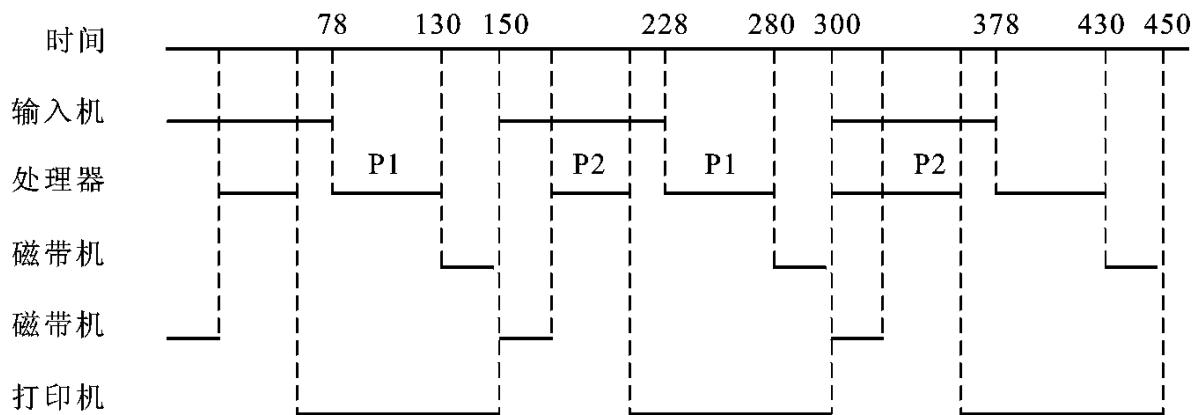


图 1-4 两道算题运行时处理器的使用效率

当这两道算题同时进入内存计算时,计算过程示意在图 1-4 中。其中,P1 表示程序甲占用 CPU 对输入机输入的 500 个字符进行处理,由于 52 ms 处理便结束,下次处理要等待 98 ms 之后,故这个时间段内 CPU 是空闲的。系统调度程序乙工作,它从磁带机上输入 2 000 个字符后,P2 表示对这批数据进行处理。相应的 I/O 设备和 CPU 的操作都是并行的。不难算出,此时处理器的利用率为:

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

由此可以看出,让几个程序同时进入内存计算比一个个串行地进行计算时的 CPU 效率要高。因为,当某个程序因故不能继续运行下去时,管理程序便把 CPU 分给另外一个程序执行,这样可使 CPU 和 I/O 设备尽量都处于忙碌状态,这就是要采用多道程序设计方法的主要原因。具有处理器和外围设备并行工作能力的计算机采用多道程序设计技术后,可以提高处理器和 I/O 设备的并行性,从而,也就能提高整个系统的效率,即增加单位时间内算题的数量。例如,有甲、乙两道程序,如果让一个程序独占计算机单道运行时每道程序要花去一个小时,若此时处理器的利用率为 30%,粗略地说,甲(或乙)程序执行时所需要的处理器时间为:

$$1 \text{ 小时} \times 30\% = 18 \text{ 分钟}$$

假定甲、乙两个程序按多道程序设计方法同时运行,处理器的利用率达 50%,那么,要提供 36 分钟的处理器时间,大约要运行 72 分钟。所以,粗略地估计,采用多道程序设计技术只要大约 72 分钟就可以将两个程序计算完毕。然而,由于操作系统调度要花费处理器时间,所以,实际花费的时间可能还要长些,例如,共要花 80 分钟。而单道运行时,甲、乙依次执行完需 120 分钟。因而;采用多道程序设计方法后可以提高效率:

$$(120 - 80) / 120 \approx 33\%$$

但是从甲、乙两个程序来看,如果单道运行,它花 60 分钟就可以得到结果,而多道运行时,却要花 80 分钟才有结果,延长了 20 分钟,即延长了 33% 的时间。所以,采用多道程序设

计方法后,提高了系统效率,即增长了单位时间的算题量,但是,对于每一个程序来说,却延长了计算时间。所以,多道程序设计技术提高资源利用率和系统吞吐率是以牺牲用户的作业周转时间为代价的。对于一些实时响应的计算问题,延长 10% 的计算时间可能都是难以接受的。因此,这个问题在多道程序设计中必须引起注意。

在多道程序设计中,还有一个值得注意的问题是道数的多少。从表面上看,似乎道数越多越能提高效率,但是道数的多少绝不是任意的,它往往由系统的资源以及用户的要求而定。例如,如果上述甲、乙两个程序都要用行式打印机,而系统只有一台行式打印机,就算它们被同时接受进入计算机内存运行,未必能提高效率。因为,可能程序甲计算了一段时间后要等程序乙不再使用行式打印机时,即程序乙结束后,才能继续运行。此外,内存储器的容量和用户的响应时间等因素也影响多道程序道数的多寡。可以采用概率方法来算出 CPU 的利用率,假如一个程序等待 I/O 操作的时间占其运行时间的比例为 p ,当内存中有 n 个程序时,则所有 n 个程序都等待 I/O 的概率是 p^n ,亦即这时 CPU 是空闲的。那么,

$$\text{CPU 的利用率} = 1 - p^n$$

其中, n 称多道程序的道数或度数 (degree of multiprogramming), 可见 CPU 的利用率是 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 时间的浪费低于 10%,粗略计算至少要有 10 个程序在内存中多道运行。事实上,一个用户程序等待从终端输入信息或等待磁盘 I/O,等待时间超过 80% 也是常有的事。上述 CPU 利用率计算模型很粗略,但却是有效的。假设计算机有 1 MB 内存,操作系统占用 200 KB,其余空间允许 4 个用户程序共享,每个占用 200 KB。若它们 80% 时间用于 I/O 等待,则 CPU 的利用率(忽略操作系统开销时) $= 1 - (0.8)^4 = 59\%$ 。当增加 1 MB 内存后,多道程序可从 4 个增加到 9 道,因而,CPU 的利用率 $= 1 - (0.8)^9 = 87\%$,也就是说第二个 1 MB 内存提高了 47% 的系统吞吐量。增加第三个 1 MB 内存只将 CPU 的利用率从 87% 提高到 96%,吞吐量仅提高了 10%。

下面小结一下操作系统中引入多道程序设计的好处:一是提高了 CPU 的利用率,二是提高了内存和 I/O 设备的利用率,三是改进了系统的吞吐率,四是充分发挥了系统的并行性。其主要缺点是延长了作业周转时间。

注意,多道程序设计系统与多重处理系统 (multiprocessing system) 有差别,后者是指配置了多个物理 CPU,从而,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有多重处理系统支持。多重处理系统的硬件结构可以多种多样,如共享内存的多 CPU 结构、网络连接的独立计算机结构。虽然多重处理系统增加了硬件,但却换来了提高系统吞吐量、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三个问题:存储保护与程序浮动;处理器的管理和分配;系统资源的管理和调度。

在多道程序设计的环境中,内存储器为几道程序所共享,因此,硬件必须提供必要的手段,使得在内存储器中的各道程序只能访问它自己的区域,以避免相互干扰。特别是当一道程序发生错误时,不致影响其他的程序,更不能影响系统程序,这就是存储保护。同时,由于每道程序不是独占全机,这样,不能事先规定它运行时将放在哪个区域,所以,程序员在编制程序时无法知道程序在内存储器中的确切地址。甚至,在运行过程中,一个程序也可能改变其运行区域,所有这些,都要求一个程序或程序某一部分能随机地从某个内存储器区域移动到另一个区域,而不影响其执行,这就是程序浮动,或称地址重定位。此外,多个程序共存于内存,会引起内存容量不足,因此,内存扩充也成为操作系统必须要解决好的问题。

在多道程序设计系统里,如果系统仅配置一个物理处理器,那么,多个程序必须轮流占有处理器,这涉及到处理器调度问题。为了说明一个程序是否占有或可以占有处理器,可以把程序在执行中的状态分成三种。当一个程序正占有处理器运行时,就说它是处于运行状态(运行态);当一个程序在等待某个事件发生时,就说它处于等待状态(等待态);当一个程序等待的条件已满足可以运行而未占用处理器时,则说它处于就绪状态(就绪态),所以,一个程序在执行中总是处于运行、就绪、等待三种状态之一。一个程序在执行过程中,它的程序状态是变化的,从运行态到等待态的转换是在发生了某种事件时产生的。这些事件可能是由于启动外围设备输入输出而使程序要等待输入输出结束后才能继续下去;也可能是在运行中发生了某种故障使程序不能继续运行下去等等。从等待态转换成就绪态是在等待的某个事件完成时产生的。例如,程序甲处于等待外围设备传输完毕的等待状态,当传输结束时,程序甲就从等待态转为就绪态。从运行态也能转变为就绪态。例如,当程序乙运行时发生了设备传输结束事件,而当设备传输结束后,使得程序甲从等待态转变为就绪态;假定程序甲的优先级高于程序乙,因此,让程序甲占有处理器运行,这样,程序乙就从运行态转为就绪态。

在多道程序设计系统里,系统的资源为几个程序所共享,上面谈到的处理器就是一例。此外,如内存储器、外围设备以及一些信息资源等也需要按一定策略去分配和调度,要解决好多道程序共享系统硬软件资源的竞争与协调。有关调度算法与实现及解决程序之间资源竞争与协调的机制将在以后各章叙述。

2. 操作系统的形成

第三代计算机的性能有了更大提高,机器速度更快,内外存容量增大,I/O设备数量和种类增多,为软件的发展提供了有力支持。如何更好地发挥硬件功效,如何更好地满足各种应用的需要,这些都迫切要求扩充管理程序的功能。

中断技术和通道技术的出现使得硬部件具有了较强的并行工作能力,从理论上来说,实现多道程序系统已无问题。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增高。这个阶段虽然有个别的磁带操作系统出

现,但操作系统的真正形成还期待着大容量高速辅助存储器的出现。大约到 60 年代中期以后,随着磁盘的问世,相继出现了多道批处理操作系统和分时操作系统、实时操作系统,到这个时候标志着操作系统正式形成。

计算机配置操作系统后,其资源管理水平和操作自动化程度有了进一步提高,具体表现在:

- 操作系统实现了计算机操作过程的自动化。批处理方式更为完善和方便,作业控制语言有了进一步发展,为优化调度和管理控制提供了新手段。
- 资源管理水平有了提高,实现了外围设备的联机同时操作(即 SPOOLING),进一步提高了计算机资源的利用率。
- 提供虚存管理功能,由于多个用户作业同时在内存中运行,在硬件设施的支持下,操作系统为多个用户作业提供了存储分配、共享、保护和扩充的功能,导致操作系统步入实用化。
- 支持分时操作,多个用户通过终端可以同时联机地与一个计算机系统交互。
- 文件管理功能有改进,数据库系统开始出现。
- 多道程序设计趋于完善,采用复杂的调度算法,充分利用各类资源,最大限度地提高计算机系统效率。

1.2.4 操作系统的发展与分类

促使操作系统不断发展的主要动力有以下五个方面:

1. 器件快速更新换代。微电子技术是推动计算机技术飞速发展的“引擎”,每隔 18 个月其性能要翻一番。推动微机快速更新换代,它由 8 位机、16 位机发展到 32 位,当前已经研制出了 64 位机,相应的微机操作系统也就由 8 位微机操作系统发展到 16 位、32 位微机系统,而 64 位微机操作系统也已研制出来。

2. 计算机体体系结构不断发展。硬件的改进导致操作系统发展的例子很多,内存管理支撑硬件由分页或分段设施代替了界寄存器以后,操作系统中便增加了分页或分段存储管理功能。图形终端代替逐行显示终端后,操作系统中增加了窗口管理功能,允许用户通过多个窗口在同一时间提出多个操作请求。引进了中断和通道等设施后,操作系统中引入了多道程序设计功能。计算机体体系结构的不断发展有力地推动着操作系统的发展。例如,计算机由单处理机改进为多处理机系统,操作系统也由单处理机操作系统发展到多处理机操作系统和并行操作系统。随着计算机网络的出现和发展,出现了分布式操作系统和网络操作系统。随着信息家电的发展,又出现了嵌入式操作系统。

3. 提高计算机系统资源利用率的需要。多用户共享一套计算机系统的资源,必须千方百计地提高计算机系统中各种资源的利用率,各种调度算法和分配策略相继被研究和采用,这也成为操作系统发展的一个动力。

4. 让用户使用计算机越来越方便的需要。从批处理到交互型分时操作系统的出现,大

大改变了用户上机、调试程序的环境;从命令行交互进化到 GUI 用户界面,操作系统的界面变得更加友善。

5. 满足用户的新要求,提供给用户新服务。当用户发现现有的工具和功能不能满足用户需要时,操作系统往往要进行升级换代,开发新工具,加入新功能。

从操作系统形成以来,按照功能、特点和使用方式的不同,可把操作系统区分为三种基本类型:

1. 批处理操作系统

过去,在计算中心的计算机上一般所配置的操作系统都采用以下方式工作:用户把要计算的应用问题编成程序,连同数据和作业说明书一起交给操作员,操作员集中一批作业,并输入到计算机中。然后,由操作系统来调度和控制用户作业的执行,形成一个自动转接的连续处理的作业流。最后,由操作员把运算结果返回给用户。通常,采用这种批量处理作业方式的操作系统称为批处理操作系统(Batch Operating System)。

批处理操作系统根据一定的调度策略把要求计算的算题按一定的组合和次序去执行,从而,系统资源利用率高,作业的吞吐量大。缺点是:作业周转时间长,不能提供交互计算能力,不利于程序的开发和调试。批处理系统的主要特征是:

- 用户脱机工作。用户提交作业之后直至获得结果之前不再和计算机及他的作业交互。因而,作业控制语言对脱机工作的作业来说是必不可少的。由于发现程序错误不能及时修正,这种工作方式对调试和修改程序极不方便。
- 成批处理作业。操作员集中一批用户提交的作业,输入计算机成为后备作业。后备作业由批处理操作系统一批批地选择并调入内存执行。
- 单/多个程序运行。早期采用单道批处理,作业进入系统后排定次序,依次一道道进入内存处理,形成了一个自动转接的作业流处理,这是单道批处理系统。后来采用多道批处理,由作业调度程序按预先规定的调度算法,从后备作业中选取多个作业进入主存,并启动它们运行,这是多道批处理系统。

2. 分时操作系统

在批处理系统中,用户不能干预自己程序的运行,无法得知程序运行情况,对程序的调试和排错不利。为了克服这一缺点,便产生了分时操作系统。

允许多个联机用户同时使用一台计算机系统进行计算的操作系统称分时操作系统(Time Sharing Operating System)。其实现思想如下:每个用户在各自的终端上以问答方式控制程序运行,系统把中央处理器的时间划分成时间片,轮流分配给各个联机终端用户,每个用户只能在极短的时间内执行,若时间片用完,而程序还未做完,则挂起等待下次分得时间片。由于调试程序的用户常常只发出简短的命令,这样一来,每个用户的每次要求都能得到

快速响应,每个用户获得这样的印象,好像他独占了这台计算机一样。实质上,分时系统是多道程序的一个变种,CPU被若干个交互式用户多路分用,不同之处在于每个用户都有一台联机终端。

分时的思想于 1959 年由 MIT 正式提出,并在 1962 年开发出了第一个分时系统 CTSS (Compatible Time Sharing System),成功地运行在 IBM 7094 机上,能支持 32 个交互式用户同时工作。1965 年 8 月 IBM 公司公布了 360 机上的分时系 TSS/360 (Time Sharing System/360),这是一个失败的系统,由于它太大太慢,没有一家用户愿意使用。

1965 年在美国国防部的支持下,MIT、BELL 和 GE 公司决定开发一个“公用计算服务系统”,以支持整个波士顿地区所有分时用户,这个系统就是 MULTICS (MULTIplexed Information and Computing Service)。MULTICS 运行在 GE635、GE645 计算机上使用高级语言 PL/1 编程。特别值得一提的是:MULTICS 引入了许多现代操作系统的概念雏形,如分时处理、远程联机、段页式虚拟存储器、文件系统、多级反馈调度、保护环安全机制、多 CPU 管理,多种程序设计环境等,对后来操作系统的设计有着极大的影响。今天,分时操作系统已成为最流行的一种操作系统,几乎所有的现代通用操作系统都具备分时系统的功能。

分时操作系统具有以下特性:

- 同时性。若干个终端用户同时联机使用计算机,分时就是指多个用户分享使用同一台计算机的 CPU 时间。
- 独立性。终端用户彼此独立,互不干扰,每个终端用户感觉上好像他独占了这台计算机。
- 及时性。终端用户的立即型请求(即不要求大量 CPU 时间处理的请求)能在足够快的时间之内得到响应(通常应该为 2~3 秒钟)。这一特性与计算机 CPU 的处理速度、分时系统中联机终端用户数目和时间片的长短密切相关。
- 交互性。人机交互,联机工作,用户直接控制其程序的运行,便于程序的调试和排错。

分时操作系统和批处理操作系统虽然有共性,它们都基于多道程序设计技术,但存在下列不同点:

- 追求的目标不同。批处理系统以提高系统资源利用率和作业吞吐率为目;分时系统则要满足多个联机用户立即型命令的快速响应。
- 适应的作业不同。批处理适应已经调试好的大型作业;而分时系统适应正在调试的小作业。
- 资源的利用率不同。批处理操作系统可合理安排不同负载的作业,使各种资源利用率较佳;分时操作系统中,多个终端作业使用相同类型编译系统、运行系统和公共子程序时,系统调用它们的开销较小。
- 作业控制的方式不同。批处理由用户通过 JCL 的语句书写作业控制流,预先提交,

脱机工作;交互型作业,由用户从键盘输入操作命令控制,交互方式,联机工作。

时间片轮转法调度中,时间片长度的选取是一个重要问题。系统进行进程切换,需要分别保存和恢复相应进程现场,修改进程状态,调整和更新表格和队列,假如每次进程切换花费 2.5 ms,而时间片设为 25 ms 时,CPU 时间的 10% 将被耗费在管理上。为了提高效率,可设时间片为 500 ms,这时仅有 0.5% 的 CPU 时间被耗在管理上。但在一个分时系统中,假如有十个交互型用户正排在就绪队列中等待轮转,当 CPU 空闲时,立即启动第一个就绪进程,第二个就绪进程大约在 1/2 秒之后被启动,依次类推。假如每个进程用足了分到的时间片,最后一个进程不得不等待 5 秒钟之后才获得运行机会,多数用户无法忍受一条简短的命令要 5 秒钟才能响应。为此得出结论:分时系统中,时间片设得太短会导致过多的进程切换,降低了 CPU 的效率;但时间片设得太长又可能对短的交互型请求的响应时间变长。应当根据机器的速度、用户的多少、响应的要求、系统的开销折中考虑,选择合理的时间片长度。

实现分时系统有不同的方法。简单分时操作系统中,主存中仅存放一个现行作业,其余均存在辅存上,为了使每个作业能得到及时响应,规定作业运行一个时间片后便暂停并调出至辅存,再从辅存上选一个作业装入主存运行,这样轮转一段时间后使每个作业都运行一个时间片,就能让用户通过终端与自己的作业交互,以保证及时响应用户的操作请求。为了改进系统性能,可引入前台/后台的分时系统,前台交互型作业不断在主存和辅存间调进/调出并按时间片轮转运行作业;当前台无作业可运行时,调度后台作业执行。进一步提高效率可以在多道程序设计技术基础上实现分时系统,主存中同时装入许多道(小)作业,这些作业排成多个优先级不同的队列,高优先级队列中的交互型作业依次获得一个时间片运行,保证了终端用户的操作请求能获得及时响应,仅当高优先级队列空或无作业可运行时,才调度低优先级队列的作业。

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获得了较佳的资源利用率和快速的响应时间,从而使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领域的需要。于是,便产生了实时操作系统,目前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。计算机用于生产过程控制时,要求系统能现场实时采集数据,并对采集的数据进行及时处理,进而能自动地发出控制信号控制相应执行机构,使某些参数(压力、温度、距离、湿度)能按预定规律变化,以保证产品质量。导弹制导系统,飞机自动驾驶系统,火炮自动控制系统都是实时过程控制系统。计算机还可用于控制和进行实时信息处理,情报检索系统是典型的实时信息处理系统,计算机同时接收成千上百从各处终端发来的服务请求和提问,系统应在极快的时间内做出回答和响应。事务处理系统不仅对终端用户及时作出响应,而且要对系统中的文件或数据库频繁更新。例如,银行业务处理系统,每次银行客户发生业务往来,自己均需修改文件或数据库。这样的系统应该具有响应快捷、安全保密,

可靠性高的特点。

实时操作系统(Real Time Operating System)是指当外界事件或数据产生时,能够接收并以足够快的速度予以处理,其处理的结果又能在规定的时间之内来控制生产过程或对处理系统作出快速响应,并控制所有实时任务协调一致运行的操作系统。因而,提供及时响应和高可靠性是其主要特点。由实时操作系统控制的过程控制系统较为复杂,通常由四部分组成:(1)数据采集。它用来收集、接收和录入系统工作必须的信息或进行信号检测。(2)加工处理。它对进入系统的信息进行加工处理,获得控制系统工作所必须的参数或作出决定,然后,进行输出,记录或显示。(3)操作控制。它根据加工处理的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,它监督执行机构的执行结果,并将该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采取进一步措施,达到控制的预期目的。

在实时系统中要有实时时钟管理,以便对实时任务进行实时处理。通常系统中存在若干个实时任务,它们常常通过“队列驱动”或“事件驱动”开始工作,当系统接收到某些外部事件后,分析这些消息,驱动实时任务完成相应处理和控制。可以从不同角度对实时任务加以分类。按任务执行是否呈现周期性可分成周期性实时任务和非周期性实时任务;按实时任务截止时间可分成硬实时任务(极严格的时间要求内完成任务)和软实时任务(不太严格的时间要求,可在一定的时间范围内完成任务)。

也可以对分时操作系统和实时操作系统作一个简单比较,它们的主要区别是:设计目标不同,前者为了给多用户提供一个通用的交互型开发运行环境,后者通常为特殊用途提供专用系统;交互性强弱不同,前者交互性强,后者交互性弱;响应时间要求不同,前者以用户能接受的响应时间为标准,后者则与受控对象及应用场合有关,变化范围很大。

上述三种是操作系统的基本类型,如果一个操作系统兼有批处理、分时和实时处理的全部或两种功能,则该操作系统称为通用操作系统。

4. 微机操作系统

微型计算机的出现犹如一颗重磅炸弹,导致了计算机产业革命。今天微型计算机已经进入社会的各个领域,拥有巨大的使用量和最广泛的用户。从 20 世纪 70 年代中期到 80 年代早期,微型计算机上运行的一般是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)。80 年代中期到 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。以 MP/M、XENIX 和后期 MS-DOS 为代表。

近年来,微机操作系统得到了进一步发展,以 Windows、OS2、MACOS 和 Linux 为代表的新一代微机操作系统具有 GUI、多用户和多任务、虚拟存储管理、网络通信支持、数据库支持、多媒体支持、应用编程支持 API 等功能。并且还具有以下特点:(1)开放性。支持不同系统互连、支持分布式处理和支持多 CPU 系统。(2)通用性。支持应用程序的独立性和在不同

平台上的可移植性。(3)高性能。随着硬件性能提高、64位机逐步普及、CPU速度进一步提高,微机操作系统中引进了许多以前在中、大型机上才能实现的技术,支持虚拟存储器,支持多线程,支持对称处理器SMP,导致计算机系统性能大大提高。(4)采用微内核结构。提供基本支撑功能的内核极小,大部分操作系统功能由内核之外运行的服务程序(也称服务器)来实现。

5. 并行操作系统

计算机的应用经历了从数据处理到信息处理,从信息处理到知识处理,每前进一步都要求增加计算机的处理能力。为了达到极高的性能,除提高元器件的速度外,计算机系统结构也必须不断的改进,而这一点主要采用增加同一时间间隔内的操作数量,通过并行处理(parallel processing)技术,研究并行计算机来达到的,已经开发出的并行计算机有:阵列处理器、流水线处理机、多处理机。并行处理技术已成为近年来计算机的热门研究课题,它在气象预报、石油勘探、空气动力学、基因研究、核技术及航空航天飞行器设计等领域均有广泛应用。

为了发挥并行计算机的性能,需要有并行算法、并行语言等许多软件的配合,而并行操作系统则是并行计算机发挥高性能的基础和保证。所以,人们越来越重视并行操作系统的研究和开发。目前已经研究出来的并行操作系统有:美国Stanford大学的V-Kernel、美国Bell实验室的Meglos、美国卡耐基梅隆大学的MACH等。

6. 网络操作系统

计算机网络是通过通信设施将地理上分散的并具有自治功能的多个计算机系统互连起来的系统。网络操作系统(Network Operating System)能够控制计算机在网络中方便地传送信息和共享资源,并能为网络用户提供各种所需服务的操作系统。网络操作系统主要有两种工作模式:第一种是客户机/服务器(Client/Server)模式,这类网络中分成两类站点,一类作为网络控制中心或数据中心的服务器,提供文件打印、通信传输、数据库等各种服务;另一类是本地处理和访问服务器的客户机。这是目前较为流行的工作模式。另一种是对等(Peer-to-Peer)模式,这种网络中的站点都是对等的,每一个站点既可作为服务器,而又可作为客户机。

网络操作系统应该具有以下几项功能:(1)网络通信。其任务是在源计算机和目标计算机之间,实现无差错的数据传输。具体来说完成建立/拆除通信链路、传输控制、差错控制、流量控制、路由选择等功能。(2)资源管理。对网络中的所有硬、软件资源实施有效管理,协调诸用户对共享资源的使用,保证数据的一致性、完整性。典型的网络资源有:硬盘、打印机、文件和数据。(3)网络管理。包括安全控制、性能监视、维护功能等。(4)网络服务。如电子邮件、文件传输、共享设备服务、远程作业录入服务等。

目前,计算机网络操作系统有三大主流:UNIX、Netware 和 Windows NT。UNIX 是惟一能

跨多种平台的操作系统;Windows NT 工作在微机和工作站上;Netware 则主要面向微机。支持 C/S 结构的微机网络操作系统则主要有:Netware、UNIXware、Windows NT、LAN Manager 和 LAN Server 等。

下一代网络操作系统应能提供以下功能支撑:

- 位置透明性。支持客户机、服务器和系统资源不停地在网络中装入卸出,且不固定确切位置的工作方式。
- 名空间透明性。网络中的任何实体都必须从属于同一个名字空间。
- 管理维护透明性。如果一个目录在多台机器上有映象,应负责对其同步维护;应能将用户和网络故障相隔离;同步多台地域上分散的机器的时钟。
- 安全权限透明性。用户仅需使用一个用户名及口令,就可在任何地点对任何服务器的资源进行存取,请求的合法性由操作系统验证,数据的安全性由操作系统保证。
- 通信透明性。提供对多种通信协议支持,缩短通信的延时。

7. 分布式操作系统

以往的计算机系统中,其处理和控制功能都高度地集中在一台计算机上,所有的任务都由它完成,这种系统称集中式计算机系统。而分布式计算机系统是指由多台分散的计算机,经互连网络连接而成的系统。每台计算机高度自治,又相互协同,能在系统范围内实现资源管理,任务分配,能并行地运行分布式程序。

用于管理分布式计算机系统的操作系统称分布式操作系统 (Distributed Operating System)。它与单机集中式操作系统的主要区别在于资源管理,进程通信和系统结构三个方面。和计算机网络类似,分布式操作系统中必须有通信规程,计算机之间的发信收信都将按规程进行。分布式系统的通信机构、通信规程和路径算法都是十分主要的研究课题。集中式操作系统的资源管理比较简单,一类资源由一个资源管理程序来管理。这种管理方式不适合于分布式系统,例如,一台机器上的文件系统来管理其他计算机上的文件是有困难的。所以,分布式系统中,对于一类资源往往有多个资源管理程序,这些管理者必须协调一致的工作,才能管好资源。这种管理比单个资源管理程序的方式复杂得多,人们已开展了许多研究工作,提出了许多分布式同步算法和同步机制。分布式操作系统的结构也和集中式操作系统不一样,它往往有若干相对独立部分,各部分分布于各台计算机上,每一部分在另外的计算机上往往有一个副本,当一台机器发生故障时,由于操作系统的每个部分在其他机上有副本,因而,仍可维持原有功能。

Plan9 是由 AT&T 公司的 BELL 实验室于 1987 年由 Ken Thompson (UNIX 设计者之一) 参与开发的一个具有全新概念的分布式操作系统。开发 Plan9 的主要动机是为解决 UNIX 系统的日趋庞大,以及在可移植性、可维护性和对硬件环境的适应性方面所遇到的种种困难。小而精的思想贯彻到系统的设计中,整个系统仅有约 15000 行 C 源代码。Plan9 是运行在由

不同网络联接的 CPU 服务器、文件服务器及终端机的分布式硬件上的一个分布式操作系统。Plan9 实现中引入和使用了以下概念和技术:命名空间 (Naming Space)、进程文件系统 (Process File System)、窗口系统 (Windows System)、CPU 命令 (CPU Command) 等。

分布式操作系统 Amoeba 由荷兰自由大学和数学信息科学中心联合研制。设计 Amoeba 的一个基本思想是:用户就象使用传统的计算机那样来使用 Amoeba、即用户不知道他正在用哪些机器?用了几台?是哪几台?用户也不知道文件存在哪台机器上?共有几个备份?其主要目标是:进行分布式系统的研究,建立一个良好的试验平台,以便在上面进行算法、语言和应用的试验。Amoeba 将网络中的机器分成若干个组,包括 CPU 池组、工作站组、专用服务器组,通过一专用的 Gateway 将局域网联到全局网中构成 Amoeba 的硬件体系结构。Amoeba 的微内核具有四项基本功能:管理进程和线程;支持底层内存管理;支持线程间的透明通信;实现 I/O 处理。Amoeba 的服务功能由下列服务程序实现:快速文件服务程序,目录服务程序,监控服务程序等。

其他著名的分布式操作系统还有:Cm^{*} (美国卡耐基梅隆大学), 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 (美国乔治亚理工学院), CMDS (英国剑桥大学)。

分布式系统研究和开发的主要方向有:

- 分布式系统结构。研究非共享通路结构和共享通路结构。
- 分布式操作系统。研究资源管理方法、同步控制机制、死锁的检测与解除、进程通信模型及手段等。
- 分布式程序设计。扩充顺序程序设计语言使其具有分布式程序设计能力;开发新的分布式程序设计语言。
- 分布式数据库。设计开发新的分布式数据库。
- 分布式应用。研究各种分布式并行算法,研究在办公自动化、自动控制、管理信息系统等各个领域的应用。

8. 嵌入式操作系统

随着以计算机技术、通信技术为主的信息技术的快速发展和 Internet 网的广泛应用,3C (Computer, Communication, Consumer Electronics) 合一的趋势已初露端倪,计算机是贯穿社会信息化的核心技术,网络和通信是社会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成事实,这就为把计算机技术渗透到各行各业,应用到各个领域、嵌入到各种设备,开发出各种新型产品,奠定了坚实的物质基础。在这些领域内部产生了一个共同需求:计算机嵌入式应用。嵌入式(计算机)系统硬件不再以物理上独立的装置或设备形式出现,而是大部分甚至全部都隐藏和嵌入到各种应用系统中。由于嵌入式(计算机)系统的应用环境与其他类

型的计算机系统有着巨大的区别,随之带来了对嵌入式(计算机)系统的软件、即嵌入式软件(embedded software)的要求,而嵌入式操作系统是嵌入式软件的基本支撑。从而,形成了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迅速发展,面对巨大的生产量和用户量,嵌入式软件和嵌入式操作系统的应用前景十分广阔。

嵌入式操作系统指运行在嵌入式(计算机)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。由于它仍旧是一个操作系统,因此,具有通常操作系统的功能,包括与硬件相关的底层软件、操作系统核心功能(文件系统、存储管理、设备管理、进程管理、处理器管理和中断处理),功能强大的还提供图形界面、通信协议、小型浏览器等设施。但由于嵌入式操作系统的硬件平台的局限性、应用环境的多样性和开发手段的特殊性,它与一般操作系统相比又有很大不同,嵌入式操作系统和其他嵌入式软件都有下列特点:

- 微型化。硬件平台的局限性表现在可用内存少(1兆以内)、往往不配置外存、微处理器字长短且运算速度有限(8位、16位字长居多)、能提供的能源较少(用微型电池)、外部设备和被控设备千变万化。因而,不论从性能还是从成本角度考虑,都不允许它占用很多资源,系统代码量少,应在保证应用功能的前提下,以微型化作为出发点来设计嵌入式操作系统的结构与功能。
- 可定制。嵌入式操作系统运行的平台多种多样,应用更是五花八门,因而,表现出专业化的特点。从减少成本和缩短研发周期考虑,要求它能运行在不同微处理器平台上,能针对硬件变化进行结构与功能上的配置,以满足不同应用需要。例如,提供可配置和可剪裁的内核处理器模块和对不同存储器(ROM、RAM与Flash Memory)的存储管理模块供应用选择;同样,如文件系统、图形接口、网络通信与TCP/IP、多媒体处理等各种功能可做成模块作为选件,供用户选择。
- 实时性。嵌入式操作系统广泛应用于过程控制、数据采集、传输通信、多媒体信息(语音、视频影像处理)及关键要害领域等要求迅速响应的场合,实时响应要求严格,因而,实时性是其主要特点之一。针对应用的响应时间要求,可设计成硬实时、软实时和非实时不同级别的系统。对于应用在军事武器、航空航天、交通运输等特殊领域有硬实时要求的系统,其中断响应、处理器调度等机制必须严格符合时间要求。
- 可靠性。系统构件、模块和体系结构必须达到应有的可靠性,对关键要害应用还要提供容错和防故障措施,进一步改进可靠性。
- 易移植性。为了提高系统的易移植性,通常采用硬件抽象层HAL(Hardware Abstraction level)和板级支撑包BSP(Board Support Package)的底层设计技术。HAL提供了与设备无关的特性,屏蔽硬件平台的细节和差异,向操作系统上层提供统一接口,保证了系统的可移植性。而一般由硬件厂家提供的,按给定的编程规范完成BSP,保证了嵌入式操作系

统可在新推出的微处理器硬件平台上运行。目前国际上主要的嵌入式操作系统可以支持的微处理器已经超过几十种。

- 开发环境。嵌入式操作系统与其定制或配置工具联系密切,构成了嵌入式操作系统集成开发环境,其中,通常提供了代码编辑器、编译器和链接器、程序调试器、系统配置器和系统仿真器。

嵌入式操作系统与应用环境密切相关。按应用范围划分,可把它分成通用型嵌入式操作系统和专用型嵌入式操作系统。前者可适用于多种应用领域,比较有名的有 Windows CE、VxWorks(被美国火星探险计划使用)和嵌入式 Linux;而后者则面向特定的应用场合,如适用于掌上电脑的 plam 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面世。

目前国际上嵌入式应用软件丰富多彩,具有代表性的嵌入式操作系统有:chorus(Chorus 公司),Diba(SUN 公司),Navio(Oracle 公司),OS - 9(Microsoft 公司),Psos(ISI 公司),QNX(QSSL 公司),VxWork(WindRiver 公司)和 Windows CE(Microsoft 公司)。我国中科院北京软件工程研究中心已研制出具有自主版权的嵌入式操作系统 HOPEN。

Windows CE 是微软开发的,用于通信、娱乐和移动式计算设备的嵌入式操作系统,它是微软“维纳斯”计划的核心。Windows CE 是具有开放性的,32 位多任务、多线程嵌入式操作系统。Personal Java 是 SUN 公司开发的一种用于给家庭、办公室和移动信息电器创建联网应用而专门开发的 Java 应用环境。它继承了 Java 产品家属跨硬件平台的优秀特性,非常适宜更新换代快的信息电器的应用开发。同时,SUN 公司又开发出专门用于信息电器应用开发的实时操作系统 Java OS for Consumers 和适用于存储空间有限的专用实时嵌入式操作系统 Embedded Java。HOPEN 是由中科院凯思软件集团开发的嵌入式操作系统(又称“女娲”操作系统),HOPEN 是一个微内核结构的多任务可抢占实时操作系统,核心程序约占 10kB,用 C 语言编写。主要特点有:单用户多任务、支持多进程多线程、提供多种设备驱动程序、图形用户界面、Win32API、支持 Gb2312 – 80 字符集(汉字)。Hopen 支持面向信息电器产品的 Personal Java 应用环境,可以开发在机顶盒、媒体电话、汽车导航器、嵌入式工控设备、联网服务等许多方面的应用。

1.3 操作系统提供的服务和用户接口

1.3.1 操作系统提供的基本服务

操作系统要为用户程序的执行提供一个良好的运行环境,它要为程序及其用户提供各种服务,当然不同的操作系统提供的服务不完全相同,但有许多共同点。提供操作系统共性

服务为程序员带来了方便,使编程任务变得更加容易,操作系统提供给程序和用户的共性服务大致有:

(1) 创建程序。提供各种工具和服务,程序的编辑工具和调试工具,帮助用户编程并生成高质量的源程序等服务。

(2) 执行程序。将用户程序和数据装入主存,为其运行做好一切准备工作并启动和执行程序。当程序编译或运行出现异常时,应能报告发生的情况,终止程序执行或进行适当处理。

(3) 数据 I/O。程序运行过程中需要 I/O 设备上的数据时,可以通过 I/O 命令或 I/O 指令,请求操作系统的服务。操作系统不允许用户直接控制 I/O 设备,而能让用户以简单方式实现 I/O 控制和读写数据。

(4) 信息存取。文件系统让用户按文件名来建立、读写、修改以及删除文件,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。

(5) 通信服务。在许多情况下,一个进程要与另外的进程交换信息,这种通信一般分为两种情况,一是在同一台计算机上执行的进程之间通信;二是在被网络连接在一起的不同计算机上执行的进程之间通信。进程通信可以借助共享内存(shared memory)方法实现,也可以使用消息传送(message passing)技术实现。采用前一种方法,操作系统要让两个进程连结到共享存储区;采用后一种方法,操作系统实现消息在进程之间的移动。

(6) 错误检测和处理。操作系统能捕捉和处理各种硬件或软件造成的差错或异常,并让这些差错或异常造成的影响缩小在最小范围内,必要时及时报告给操作员或用户。

此外,操作系统除上述提供给用户的服务外,还具有另外一些功能,以保证它自身高效率、高质量地工作,从而,使得多个用户程序能够有效地共享系统资源,提高系统效率,这些功能有:

(1) 资源分配。多道作业同时运行时,每一个必须获得系统资源。系统中的各类资源均由操作系统管理,如 CPU 时间、内存资源、文件存储空间等,都配有专门的分配程序,而其他资源(如 I/O 设备)配有通用的申请与释放程序。例如,为了能更好地利用 CPU,操作系统配有 CPU 调度例行程序,专门关注 CPU 的使用,掌握欲使用 CPU 的进程的状态。再如,可以设置一个例行程序探查未被使用的磁带驱动器,并且标记在内部表格,以便把它分给新用户,还可配置例行程序用来分配绘图仪、调制/解调器和其他外围设备。

(2) 统计。当希望知道用户使用计算机资源的情况,如用了多少?什么类型?以便用户付款或简单地进行使用情况统计,统计结果可以作为进一步改进系统服务,对系统进行重组的有价值的数据。

(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确保受到控制。用户程序对各种资源的需求经常发生冲突,为此,操作系统必须做出合理的调度。

操作系统提供了许多服务,底层的服务通过系统调用来实现,可被用户程序直接使用。

高层的服务通过系统程序来实现, 用户不必自己编写程序而是借助命令管理或 shell 来请求执行完成各种功能的系统程序。

1.3.2 操作系统提供的用户接口

操作系统可以通过程序接口和操作接口两种方式把它的服务和功能提供给用户, 反过来也可以这样说, 用户可以通过两个接口来调用操作系统提供的服务和功能, 如图 1-5。

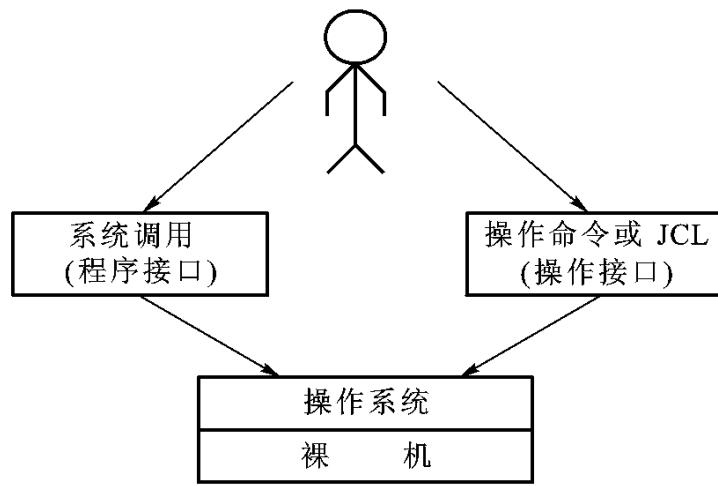


图 1-5 用户和操作系统间的两种接口

程序接口又称应用编程接口 API (Application Programming Interface), 程序中使用这个接口可以调用操作系统的服务和功能。许多操作系统的程序接口由一组系统调用 (system call) 组成, 因此, 用户在编写的程序中使用“系统调用”就可以获得操作系统的底层服务, 使用或访问系统管理的各种软硬件资源。

操作接口又称作业(或功能)级接口, 是操作系统为用户操作控制计算机工作和提供服务的手段的集合, 通常可借助操作控制命令、图形操作界面(命令), 以及作业控制语言(命令)等等来实现。

1.3.3 程序接口与系统调用

1. 系统调用

系统调用是为了扩充机器功能、增强系统能力、方便用户使用而在内核中建立的过程(函数)。用户程序或其他系统程序通过系统调用就可以访问系统资源, 调用操作系统功能, 而不必了解操作系统内部结构和硬件细节, 它是用户程序或其他系统程序获得操作系统服务的惟一途径。有些计算机系统中, 把系统调用称为广义指令。广义指令与机器指令是不相同的, 机器指令由硬件实现, 而广义指令(系统调用)是由操作系统在机器指令(访管指令)

基础上实现的、能完成特定功能的过程或子程序。操作系统为用户提供系统调用也出于安全和效率考虑,使得用户态程序不能自由地访问内核关键数据结构或直接访问硬件资源。

“基于 UNIX 的可移植操作系统接口”是国际标准化组织给出的有关系统调用的国际标准 POSIX1003.1 (Portable Operating System Interface based on UNIX),任何操作系统只有符合这一标准,才有可能运行 UNIX 程序。这个标准指定了系统调用的功能,但并未明确规定系统调用以什么形式实现,是库函数,还是其他形式。目前许多操作系统都有完成类似功能的系统调用,至于在细节上的差异就比较大了。早期操作系统的系统调用使用汇编语言编写,这时的系统调用可看成是扩展的机器指令,因而,能在汇编语言编程中直接使用。在一些程序设计语言(如 C 语言)中,往往提供一些与各系统调用对应的库函数(有些库函数与系调调用无关),因而,在高级语言中,应用程序可通过对应的库函数来使用系统调用,库函数的目的是隐藏访管指令的细节,使系统调用更象过程调用,但一般地说,库函数属于用户程序而非系统程序。最新推出的一些操作系统,如 UNIX 新版本、Linux、Windows 和 OS 2 等,其系统调用干脆用 C 语言编写,并以库函数形式提供,故在用 C 语言编制的程序中,可直接使用系统调用。图 1-6 列出了 UNIX/Linux 的系统程序、库函数、系统调用的层次关系。

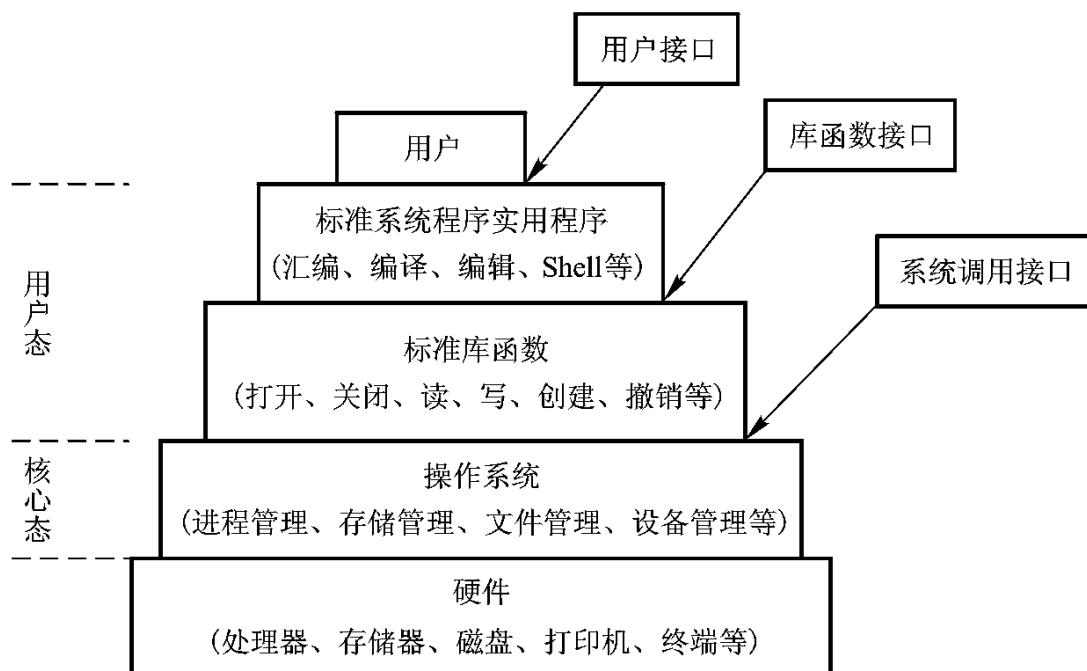


图 1-6 Unix/Linux 系统程序、库函数、系统调用的分层关系

操作系统提供的系统调用很多,从功能上大致可分成五类:(1)进程和作业管理。终止或异常终止进程、装入和执行进程、创建和撤销进程、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、获得和设置文件属性。(3)设备管理。申请设备、释放设备、设备 I/O 和重定向、获得和设置设备属性、逻辑上连接和释放设备。(4)内存管理。申请内存和释放内存。(5)信息维护。获取和设置日期及时间、获得和设置

系统数据。(6)通信。建立和断开通信连接、发送和接收消息、传送状态信息、联接和断开远程设备。

Windows 通过三个组件来支持 API: Kernel、User 和 GDI。Kernel 包含了大多数操作系统函数,如内存管理、进程管理;User 集中了窗口管理函数,如窗口创建、撤销、移动、对话及各种相关函数;GDI 提供画图函数、打印函数。所有应用程序都共享这三个模块的代码,每个 Windows 的 API 函数都可通过名字来访问,具体做法是在应用程序中使用函数名,并用适当的函数库进行编译和链接,然后,应用程序便可运行。实际上 Windows 将三个组件置于动态链接库 DLL(Dynamic Link Library)中。Windows API 的数量庞大,如 Win32 API 超过了一千个。从功能上来说,Win32 的 API 和 UNIX/Linux 的部分系统调用有粗略的对应关系,下表列出其中一些。

| UNIX/Linux | Win32 | 说 明 |
|-------------|------------------------|--------------|
| fork | CreatProcess | 创建进程 |
| waitpid | WaitForSingleObject | 等待进程终止 |
| open/close | CreatFile/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 | Creat/Remove Directory | 建立/删除目录 |
| stat | GetFileAttributesEx | 获得文件属性 |

2. 系统调用的实现要点

每个操作系统都提供几十到几百条系统调用。在操作系统中,实现系统调用功能的机制称陷入或异常处理机制,由于系统调用而引起处理器中断的机器指令称访管指令(supervisor),陷入指令(trap)或异常中断指令(interrupt)。在操作系统中,每个系统调用都事先规定了编号,称功能号,在访管或陷入指令中必须指明对应系统调用的功能号,在大多数情况下,还附带有传递给内部处理程序的参数。

系统调用的实现有以下几点:一是编写系统调用处理程序;二是设计一张系统调用入口地址表,每个入口地址都指向一个系统调用的处理程序,有的系统还包含系统调用自带参数的个数;三是陷入处理机制,需开辟现场保护区,以保存发生系统调用时的处理器现场。图 1-7 是系统调用的处理过程。

参数传递是系统调用中应处理好的问题,不同的系统调用需传递给系统调用处理程序不同的参数,反之,系统调用执行的结果也要以参数形式返回给用户程序。实现用户程序和系统调用之间的参数传递可采用以下方法:一是访管指令或陷入指令自带参数,可以规定指令之后的若干单元存放的是参数,这叫直接参数;或者在指令之后紧靠的单元中存放参数的

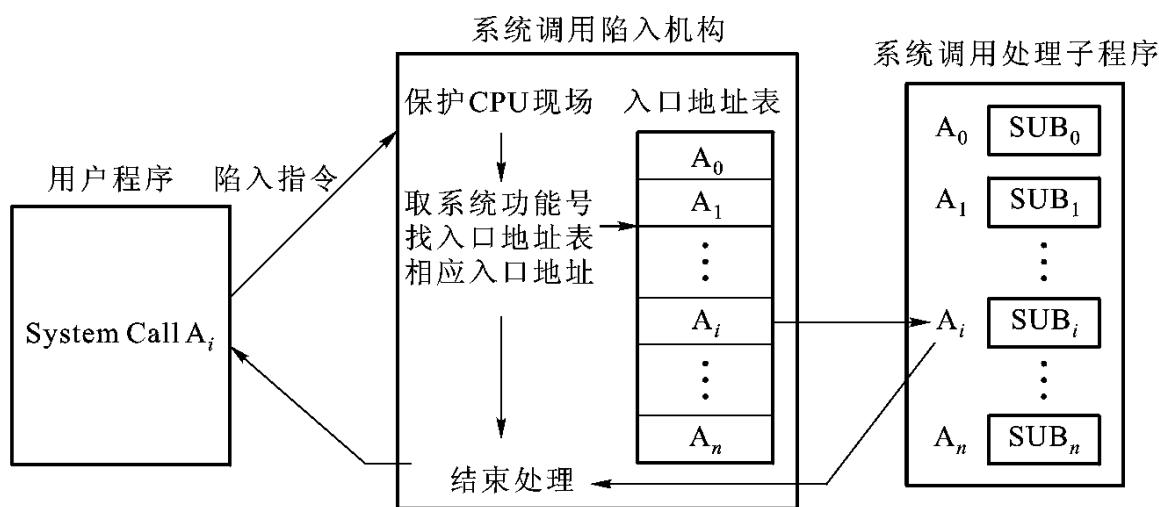


图 1-7 陷入机构和系统调用处理过程

地址,这叫间接参数,由间接地址再指出参数的存放区。二是通过 CPU 的通用寄存器传递参数,这种方法不宜传递大量参数。改进的方法是:在内存的一个区或表中存放参数,其首地址送入寄存器,实现参数传递。三是在内存中开辟专用堆栈区域传递参数。

3. 系统调用与过程(函数)调用的区别

程序中执行系统调用或过程(函数)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有很大区别。

(1) 调用形式不同。过程(函数)使用一般调用指令,其转向地址是固定不变的,包含在跳转语句中,但系统调用中不包含处理程序入口,而仅仅提供功能号,按功能号调用。

(2) 被调用代码的位置不同。过程(函数)调用是一种静态调用,调用程序和被调用代码在同一程序内,经过连接编辑后作为目标代码的一部份。当过程(函数)升级或修改时,必须重新编译连接。而系统调用是一种动态调用,系统调用的处理代码在调用程序之外(在操作系统中),这样一来,系统调用处理代码升级或修改时,与调用程序无关。而且,调用程序的长度也大大缩短,减少了调用程序占用的存储空间。

(3) 提供方式不同。过程(函数)往往由编译系统提供,不同编译系统提供的过程(函数)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、种类与数量便固定不变了。

(4) 调用的实现不同。程序使用一般机器指令(跳转指令)来调用过程(函数),是在用户态运行的;程序执行系统调用,是通过中断机构来实现,需要从用户态转变到核心态,在管理状态执行。因此程序执行系统调用安全性好。

4. Linux 的系统调用

Linux 采用类似 UNIX 技术实现系统调用,用户不能任意拦截或修改,保证了内核的安全

性。Linux 最多可以有 190 个系统调用,应用程序和 Shell 通过系统调用机制访问 Linux 内核(功能)。每个系统调用由两部分组成:(1)核心函数:是实现系统调用功能的(内核)代码,作为操作系统的内核驻留在内存中,是一种共享代码,用 C 语言书写。它运行在核心态,数据也存放在内核空间,通常它不能再使用系统调用,也不能使用应用程序可用的库函数。(2)接口函数:是提供给应用程序的 API,以库函数形式存在 Linux 的 lib.a 中,该库中存放了所有系统调用的接口函数的目标代码,用汇编语言书写。其主要功能是把系统调用号、入口参数地址传送给相应的核心函数,并使用户态下运行的应用程序陷入核心态。

Linux 中有一个用汇编语言编写的系统调用入口程序 entry(sys_call_table),它包含了系统调用入口地址表和所有系统调用核心函数的名字,而每个系统调用核心函数的编号由 include/asm/unistd.h 定义:

```
ENTRY(sys-call-table)
    long SYMBOL_NAME(sys_ni_syscall)      0
    long SYMBOL_NAME(sys_exit)            1
    long SYMBOL_NAME(sys_fork)           2
    long SYMBOL_NAME(sys_read)          3
    long SYMBOL_NAME(sys_write)         4
    long SYMBOL_NAME(sys_open)          5
    long SYMBOL_NAME(sys_close)         6
    ...
    long SYMBOL_NAME(sys_ni_syscall)      190
    long SYMBOL_NAME(sys_ni_syscall )
    long SYMBOL_NAME(sys_vfork )
```

Linux 的系统调用号就是系统调用入口表中位置的序号,所有系统调用通过接口函数将系统调用号传给内核,内核转入系统调用控制程序,再通过调用号的位置来定位核心函数,Linux 内核的陷入由 0x80(int80h)中断实现。系统调用控制程序的工作流程为:(1)取系统调用号,检验合法性;(2)建立调用堆栈,保护现场信息;(3)根据系统调用号定位核心函数地址;(4)根据通用寄存器内容,从用户栈中取入口参数;(5)核心函数执行,把结果返回应用程序;(6)执行退栈操作,判别调度程序 scheduler 是否要被执行。

1.3.4 操作接口与系统程序

1. 作业控制方式

下面提出这样一个问题,用户如何来向操作系统提交作业和说明运行意图呢?操作系统一般都提供了联机作业控制方式和脱机作业控制方式两个作业级的接口,这两个接口使

用的手段为：操作控制命令和作业控制语言（命令）。

（1）联机用户接口——操作控制命令

这是为联机用户提供的调用操作系统功能，请求操作系统为其服务的手段，它由一组命令及命令解释程序组成，所以也称为命令接口。当用户在键盘上每键入一条命令后，系统便立即转入命令解释程序，对该命令进行处理和执行。用户可先后键入不同命令，来实现对他的作业的控制，直至作业完成。

不同操作系统的命令接口有所不同，这不仅体现在命令的种类、数量及功能方面，也可能体现在命令的形式、用法等方面。不同的用法和形式组成了不同的用户界面，常用的用户界面可分成以下几种：

1) 字符显示式用户界面

主要通过命令语言来实现的，又可分成两种方式：

① 命令行方式

命令语言具有规定的词法、语法和语义，它以命令为基本单位来完成预定的工作任务，完整的命令集构成了命令语言，反映了系统提供给用户可使用的全部功能。每个命令以命令行的形式输入并提交给系统，一个命令行由命令动词和一组参数构成，它指示操作系统完成规定的功能。对新手用户来说，命令行方式十分繁琐，难以记忆；但对有经验的用户而言，命令行方式用起来快捷便利、十分灵活，所以，至今许多操作员仍欢迎并使用这种命令形式。

简单命令的一般形式为：Command arg1 arg2 ... argn

其中 Command 是命令名，又称命令动词，其余为该命令所带的执行参数，有些命令可以没有参数。

Linux 常用的命令可以分成五大类：1) 文件管理类：cd、chmod、chown、chgrp、comm、cp、crypt、diff、file、find、ln、ls、mkdir、mv、od、pr、pwd、rm、rmdir。2) 进程管理类：at、kill、mail、nice、nohup、ps、time、write、mesg。3) 文本加工类：cat、crypt、grep、norff、uniq、wc、sort、spell、tail、troff。4) 软件开发类：cc、f77、login、logout、size、yacc、vi、emacs、dbs、lex、make、lint、ld。5) 系统维护类：date、man、passwd、stty、tty、who。

② 批命令方式

在使用操作命令过程中，有时需要连续使用多条命令，有时需要多次重复使用若干条命令；还有时需要有选择地使用不同命令，用户每次都将这一条条命令由键盘输入，既浪费时间，又容易出错。现代操作系统都支持一种特别的命令称为批命令，其实现思想如下：规定一种特别的文件称批命令文件，通常该文件有特殊的文件扩展名，例如，MS – DOS 约定为 BAT。用户可预先把一系列命令组织在该 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，减少出错。更进一步，操作系统还支持命令文件使用一套控制子命令，从而，可以写出带形式参数的批命令文件。当带形式参数的批命令文件执行

时,可用不同的实际参数去替换,从而,一个这样的批命令文件可以执行不同的命令序列,大大增强了命令接口的处理能力。

UNIX 和 Linux 的 Shell 不但是一种交互型命令解释程序,也是一种命令级程序设计语言解释系统,它允许用户使用 Shell 简单命令、位置参数和控制流语句编制带形式参数的批命令文件,称作 Shell 文件或 Shell 过程,Shell 可以自动解释和执行该文件或过程中的命令。

2) 图形化用户界面

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,并控制自己的作业运行,但却要牢记各种命令的动词和参数,必须严格按规定的格式输入命令,这样既不方便又浪费时间。于是,图形化用户接口 GUI(Graphics User Interface)便应运而生,是近年来最为流行的联机用户接口形式。

GUI 采用了图形化的操作界面,使用 WIMP 技术(即窗口 Window、图标 Icon、菜单 Menu 和鼠标 Pointing device),引入形象的各种图标将系统的各项功能、各种应用程序和文件,直观、逼真地表示出来。用户可以通过选择窗口、菜单、对话框和滚动条完成对他们的作业和文件的各种控制和操作。此时,用户不必死记硬背操作命令,而能轻松自如地完成各项工作,使计算机系统成为一种非常有效且生动有趣的工具。

90 年代推出的主流操作系统都提供了 GUI, GUI 的鼻祖首推 Xerox 公司的 Palo Alto Research Center 于 1981 年在 Star8010 工作站操作系统中的图形用户接口;1983 年 Apple 公司又在 Apple Lisa 机和 Macintosh 机上的操作系统中成功使用 GUI;之后,还有 Microsoft 公司的 Windows, IBM 公司的 OS/2, UNIX 和 Linux 使用的 X – Window。为了促进 GUI 的发展,已制订了国际 GUI 标准,该标准规定了 GUI 由以下部件构成:窗口、菜单、列表框、消息框、对话框、按钮、滚动条等,最早由 MIT 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,均可应用户程序要求自动生成应用程序的 GUI,大大缩短了应用程序的开发周期。

图形化操作界面又称多窗口系统,采用事件驱动的控制方式,用户通过动作来产生事件以驱动程序开始工作,事件实质上是发送给应用程序的一个消息。用户按键或点击鼠标等动作都会产生一个事件,通过中断系统激发事件驱动控制程序开始工作,它的任务是:接收事件、分析和处理事件,最后,还要清除处理过的事件。系统和用户都可以把各个命令定义为一个菜单、一个按钮或一个图标,当用户用键盘或鼠标进行选择之后,系统会自动执行该命令。

3) 新一代用户界面

随着个人计算机的广泛流行,缺乏计算机专业知识的用户随之增多,如何不断更新技术,为用户提供形象直观、功能强大、使用简便、掌握容易的用户接口,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已走向实用,把用户界面的发展推向新的阶段。目前多感知通道用户接口,自然化用户接口,甚至智能化用户接

口的研究都取得了一定的进展。

(2) 脱机用户接口——作业控制语言(命令)

这种接口是专为批处理作业的用户提供的,所以,也称批处理用户接口。操作系统提供了一个作业控制语言 JCL(Job Control Language),它由一组作业控制卡,或作业控制语句,或作业控制操作命令组成。

脱机用户接口源于早期批处理系统,其主要特征是用户事先使用作业控制语言描述好对作业的控制步骤,由计算机上运行的内存驻留程序(执行程序、管理程序、作业控制程序、命令解释程序)根据用户的预设要求自动控制作业的执行。

下面是 IBM 370 系统中使用作业控制处理批作业的一个例子。用户 WILSON 有一个作业叫 HAROLD,该作业为 B 类,优先级为 6。需要编译和连接编辑的此作业,它的源程序和数据都在穿孔卡片上,编译和连接编辑的结果需在行式打印机上输出,且编译的结果尚需保存,那么,这个批处理的作业可如下组织:

```
// HAROLD JOB, WILSON, MSGLEVEL = (2, 0), PRTY = 6, CLASS = B
// COMP EXEC PGM = IEYFORT
// SYSPRINT DD SYSOUT = A
// SYSIN DD *

< SOURCE PROGRAM CARDS >

/*
// GO EXEC PGM = FORTLINK
// SYSPRINT DD SYSOUT = A
// FTOTF001 DD UNIT = SYSCP
// GO SYSIN DD *

< DATA CARDS >

/*
//
```

批处理命令的一些应用方式有时也被认为是联机控制方式下对脱机用户接口的一种模拟。因此,UNIX/Linux 中的 Shell 也可以认为是一种 JCL 语言。由于批处理作业的用户不能直接与他们的作业交互,只能委托操作系统来对作业进行控制和干预,作业控制语言便是提供给用户,为实现所需作业控制功能委托系统代为控制的一种语言。用户使用 JCL 语句,把他的运行意图,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,将作业连同

作业说明书一起提交给系统。当调度到该批处理作业时,系统调用 JCL 语句处理程序或命令解释程序,对作业说明书上的语句或命令逐条地解释执行。如果作业在执行过程中出现异常情况,系统会根据用户在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,直到作业运行结束。可见 JCL 为用户的批作业提供了一种作业一级的接口。

2. 命令解释程序

操作系统提供的最重要的系统程序是命令解释程序(command interpreter),用户通常通过操作命令、会话语言或作业控制卡来调用系统程序。命令解释程序的主要功能是接受和执行一条由用户提供的对作业的加工处理的要求,它通常保存一张命令名字(动词)表,其中记录着所有操作命令及其处理程序的入口地址或有关信息。当一个新的批作业被启动,或新的交互用户登录进系统时,系统就自动地执行命令解释程序,它负责读入控制卡或命令行,并作出相应的解释和执行。

命令解释程序的实现有两种常见方式。一种是它自身包含了命令的执行代码,于是收到命令后,便转向该命令处理代码区执行,在执行过程中常常会使用“系统调用”帮助完成相应功能。在这种情况下,所提供的命令的数目就决定了命令解释程序功能的大小。另一种是全部命令都由专门的“系统程序”实现,它自身不含命令处理代码,也不进行处理,而仅仅把这条命令对应的命令处理文件装入内存执行。例如,键入命令,

```
delete G
```

命令解释程序将寻找名字叫 delete 的命令处理文件,把它装入内存并将参数 G 传给它,由这个文件中的代码执行相应操作。因此,与 delete 命令相关的功能全部由 delete 命令文件代码决定,而与命令解释程序无关。这样一来可把命令解释程序做得很小,添加命令也很方便,只要创建一个实现新命令功能的命令处理文件就行了。这种方法也有缺点,由于命令处理程序是独立的系统程序,因此,参数传递会增加难度。所以,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,而像编译、编辑这样的复杂命令由独立的命令处理文件完成。

下面讨论命令解释程序的处理过程。操作系统做完初始化工作后便启动命令解释程序,它输出命令提示符,等待键盘中断到来。每当用户打入一条命令(暂存在命令缓冲区)并按回车换行时,申请键盘中断。CPU 响应后,将控制权交给命令解释程序,接着读入命令缓冲区的内容,分析命令、接受参数。若为简单命令立即转向命令处理代码执行。否则查找命令处理文件,装入内存并传递参数,将控制权交给其执行。命令处理结束后,再次输出命令提示符,等待下一条命令。

UNIX/Linux shell 负责解释用户输入的操作命令,提供用户与内核程序对话的功能。大多数命令都用一个可执行二进制文件进行处理,shell 执行一个命令的大致步骤如下:打印提示符、获取命令行、解析命令、搜索相应命令文件、传递参数和执行命令。

3. 系统程序

系统程序又称标准程序或实用程序(utilities),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、结构和实现不感兴趣。实用程序虽非操作系统的核心,但却是必不可少的,它们为用户程序的开发、调试、执行和维护解决带有共性的问题或执行公共操作,于是操作系统常以外部操作命令形式向用户提供了许多系统程序。常用的有:汇编程序、编辑程序、编译系统、调试和排错程序、分类和合并程序等。用户看待操作系统,不是看系统调用怎么样,而是看系统程序怎么样,所以,系统程序功能和性能很大程度上反映了一个操作系统的功能和性能。作为操作系统的高层功能,系统程序功能的实现从根本上来说要借助系统调用的实现。系统程序大致可分成以下几类:

(1) 文件管理。这些系统程序用来对文件和目录进行建立、删除、复制、改名、打印、列表、转存和各种管理工作。

(2) 状态信息。这些系统程序提供给用户向操作系统提问,以获得日期、时间、可用内存和磁盘空间数量、用户数或其他状态信息。然后,把这些信息格式化并打印到终端或其他输出设备或保存到文件中。

(3) 程序设计语言支持。操作系统以系统程序方式提供给用户通用程序设计语言的编译程序或汇编程序或解释程序,如 Fortran、Cobol、Pascal、Basic、C 和 Lisp 等。这反映了操作系统对程序设计环境的支持能力,这些系统程序随硬件和操作系统一起出售,有一些则独立提供和另外收费。

(4) 程序的装入和执行支持。每当一个程序被编译或汇编后,必须被装入内存才能运行,于是,又提供了如绝对装入工具、重定位装入工具、连接编辑程序等系统工具。对高级语言或机器语言来说,调试工具(Debugging)也是必不可少的。

(5) 通信。操作系统提供一类系统程序,它们为建立多个进程、多个用户及不同系统之间的逻辑连接提供了机制。这些机制允许用户发送消息到其他用户的屏幕上;或以电子邮件发出一批信息;或从一台机器传送文件到另一台机器;或甚至可以通过通信进行远程登录,使用远地的计算机。

(6) 其他软件工具。操作系统提供这类系统软件为用户解决共性问题,如 Web 浏览器、字处理工具、正文格式化工具、电子表格、数据库系统、编译程序的编译程序(YACC)、画图软件包、统计分析包及游戏程序。

1.4 操作系统的结构设计

随着软件大型化、复杂化,软件设计,特别是操作系统设计呈现出以下特征:一是复杂程

度高,表现在程序庞大、接口复杂、并行度高;二是生成周期长,从提出要求明确规范起,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,需要许多年才能完成;三是正确性难保证,一个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,工作量之大,复杂程度之高可想而知。例如,MIT在1963年投入使用的CTSS约有32 000行程序;一年后出现的IBM OS/360已有超过百万条的机器指令,共有4 000个模块组成,花费了5 000人年;1975年由MIT和Bell开发的Multics增长到了千万条机器指令;而当今流行的Windows 2000则由2 500个主要开发人员参与,系统超过了3 200万行语句。Brooks描述了他负责开发的OS/360研制过程中的困难和混乱,程序设计就像是一个泥潭,一批批程序员在泥潭中挣扎,挣扎得越猛,泥浆就沾得越多,在每个OS/360修改版本中仍隐藏着无数的错误。这也是20世纪60年代出现的软件危机的一个真实写照。

一个操作系统即使开发完成,仍然是无生命的,必须要开发该系统下运行的大量的应用程序;待应用程序开发问世后,用户还必须通过文件、培训及实践去学会操作和使用。这意味着用户拥有并使用的是10年或20年以前的操作系统技术。当一个操作系统投放市场后,硬件技术却又在迈进,多CPU的计算机出现,计算机的处理器更快、内存更大、外设种类更多,使用多种文件系统,采用多种网络协议,可以和多种数据库管理系统相连接,于是操作系统设计和开发者们就要急急忙忙扩展已有系统以利用新的硬件性能和软件技术。

因而,人们开始极大地重视操作系统的软件结构和构造方法,软件危机推动了软件工程学的研究,尝试采用软件工程的方法,即运用工程的概念与原理,以及系统的、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。另外,采用现代软件工程理论构筑的操作系统还要达到正确性、高效性、可靠性、可扩充性、可移植性、可伸缩性、分布计算、认证安全性和POSIX承诺等设计目标。

操作系统是一种大型、复杂的并发系统,为了研制操作系统,首先必须研究它的结构,力求设计出结构良好的程序。操作系统的结构设计有两层含义,一是研究操作系统的整体结构,由程序的构成成分组成操作系统程序的构造过程和方法;二是研究操作系统程序的局部结构,包括数据结构和控制结构。采用不同的构件和构造方法可组成不同结构的操作系统。本节将在讨论操作系统构件之后,全面介绍各种操作系统的构造方法。

1.4.1 操作系统的构件

通常把组成操作系统程序的基本单位称作操作系统的构件。剖析现代操作系统,构成操作系统的基本单位除内核之外,主要还有进程、线程、类程和管程。

1. 内核

现代操作系统中大都采用了进程的概念,为了解决系统的并发性、共享性和随机性,并

使进程能协调地工作,单靠计算机硬件提供的功能是十分不够的。例如,进程调度工作目前就不能用硬件来实现;而进程自己调度自己也是很困难的。所以,系统必须有一组软件能对硬件处理器及有关资源进行首次改造,以便给进程的执行提供良好的运行环境,这个部分就是操作系统的内核。内核(kernel)不是进程,而是提供支持系统运行的基本功能和基本操作的一组程序模块,有了内核的支撑,进程运行环境得到改善,安全性得到保证,系统效率就能提高。

由于操作系统设计的目标和环境不同,内核的大小和功能有很大差别。有些设计希望把内核做得尽量小仅具有极少的必须功能,称为微内核(microkernel),其他功能都在核外实现,通过微内核提供的消息传递机制(这是微内核结构最主要的特点)完成其余功能模块间的联系;有些则希望内核具有较多的功能以提高系统的效率,虽然其内部也可划分成层次或模块,但运行时是一个大二进制映象,模块间的联系可通过函数或过程调用实现,称为单内核(monolithic kernel)。操作系统的一个基本设计问题是内核的功能设计。微内核结构是现代操作系统的特征之一,这种方法把内核和核外服务程序的开发分离,可根据特定应用程序或运行环境要求定制服务程序,虽说采用消息传递机制会影响执行速度,但却具有较好的可伸缩性,简化了实现,提供了灵活性,很适合分布式系统的构造。

Linux是单内核操作系统,操作系统程序(包括核心数据结构和系统服务例程)都包含在单内核中。Linux模块化的内核结构中将微内核的许多优点引入到其设计中,特别是提出了一种称为模组(module)的机制,来克服单内核的缺点。模组不是进程而是系统的一些核心部件(一种目标程序),设备驱动程序、伪设备驱动程序,如网络驱动或文件系统都组织成模组。每当系统启动后,模组可以根据需要使用命令方式(装入核心模组insmod/卸出核心模组rmmod)或通过核心守护进程(kerneld)动态地装入和卸出模组,有利于减小内核的尺寸,增强了核心的适应性,一定程度上解决了核心功能的灵活性和可伸缩性问题。Linus Torvalds本人曾经就结构问题做出过解释:现代成功的操作系统基本上都不具有微内核特性,因此,Linux也不必是微内核结构操作系统。

一般而言,内核必须提供以下三方面功能:

(1) 中断处理。中断处理是内核中最基本的功能,也是操作系统赖以活动的基础,为了缩短屏蔽中断的时间,增加系统内的并发性,通常它仅仅进行有限的、简短的处理,其余任务交给在内核之外的特殊用户态进程完成。当中断事件产生时,先由内核截获并转向中断处理例行程序进行原则处理,它分析中断事件的类型和性质,进行必要的状态修改,然后交给内核之外的进程去处理了。例如,产生外围设备结束中断事件时,内核首先分析是否正常结束,如果是正常结束,那么,就应释放等待该外围传输的进程,否则启动相应设备管理进程进行出错或异常处理。又如当操作员请求从控制台输入命令时,内核将把这一任务转交给命令管理进程去处理,以接收和执行命令。

(2) 短程调度。主要职能是分配处理器。当系统中发生了一个事件之后,可能一个进程要让出处理器,而另一个进程又要获得处理器。短程调度按照一定策略管理处理器的转让,以及完成保护和恢复现场的工作。由于它是协调进程竞争处理器资源的程序,所以它不是进程而是内核中的一个程序。

(3) 原语管理。原语是内核中实现某一功能的不可中断过程。为了协调进程完成通信、并发执行和共享资源,各种原语是必不可少的。通信原语为进程相互传递消息,同步原语能协调并发进程之间的种种制约关系。此外,还有其他原语,如启动外围设备工作的启动原语,若启动不成功则请求启动者应等待。显然,这个启动过程应该是完整的,否则在成为等待状态时,可能外围设备已经空闲。由于设备的操作与硬件密切相关,通常设备驱动程序等功能都放在内核中完成。

内核的执行有以下属性:

(1) 内核是由中断驱动的。只有当发生中断事件后由硬件交换程序状态字才引出操作系统的内核进行中断处理,且在处理完中断事件后内核自行退出。

(2) 内核的执行是连续的。在内核运行期间不能插入内核以外的程序执行,因而,能保证在一个连续的时间间隔内完成任务。为了缩短中断屏蔽时间,满足实时处理要求,Solaris操作系统在内核程序中设置了许多安全点,内核程序的这些位置允许被中断,而其他程序段都必须连续工作。

(3) 内核在屏蔽中断状态下执行。在处理某个中断时,为避免中断的嵌套可能引起的错误,必须屏蔽该级中断。有时为处理简单,把其他一些中断也暂时屏蔽了。Linux 引入了快中断和慢中断的概念,引入了中断处理程序的 top half 和 bottom half 的概念,以尽可能地缩短中断屏蔽时间。与 Linux 十分类似,Windows 2000/XP 中,设备中断服务例程 ISR 分两个阶段来处理。当设备中断发生 ISR 被首次调用时,它通常只在屏蔽中断下停留获得设备状态所必需的很短一段时间,而把设备中断处理的主要工作留给排入 DPC 队列的子例程完成,并退出服务例程,清除中断。过一段时间后,在 DPC 例程被调用时,再完成对设备中断的处理。使用 DPC 来执行大多数设备中断服务的优点是,任何优先级位于设备 IRQL 和 Dispatch/DPC IRQL 之间被阻塞的中断允许在低优先级的 DPC 处理发生之前发生。因而,中间优先级的中断就可以更快地得到服务。

(4) 内核可以使用特权指令。现代计算机都提供常态和特态等多种机器工作状态,有一类指令称特权指令,只允许在特态下使用,例如,输入输出、状态修改等。规定这类指令只允许内核使用,可防止系统出现混乱。

内核是操作系统对裸机的第一次改造,内核和裸机组成了台虚拟机,进程就在这台虚拟机上运行,它比裸机的功能更强大,具有以下特性:

(1) 虚拟机没有中断,因而,进程的设计者不再需要有硬件中断的概念,用户进程执行

中无需处理中断。

(2) 虚拟机为每个进程提供了一台虚拟处理器, 每个进程就好象在各自的私有处理器上顺序地推进, 实现了多个进程的并发执行。

(3) 虚拟机为进程提供了功能较强的指令系统, 即它们能够使用机器的非特权指令, 系统调用和原语所组成的新的指令系统。

图 1-8 给出了内核被触发开始工作, 以及内核处理的流程。用户进程在目态运行, 产

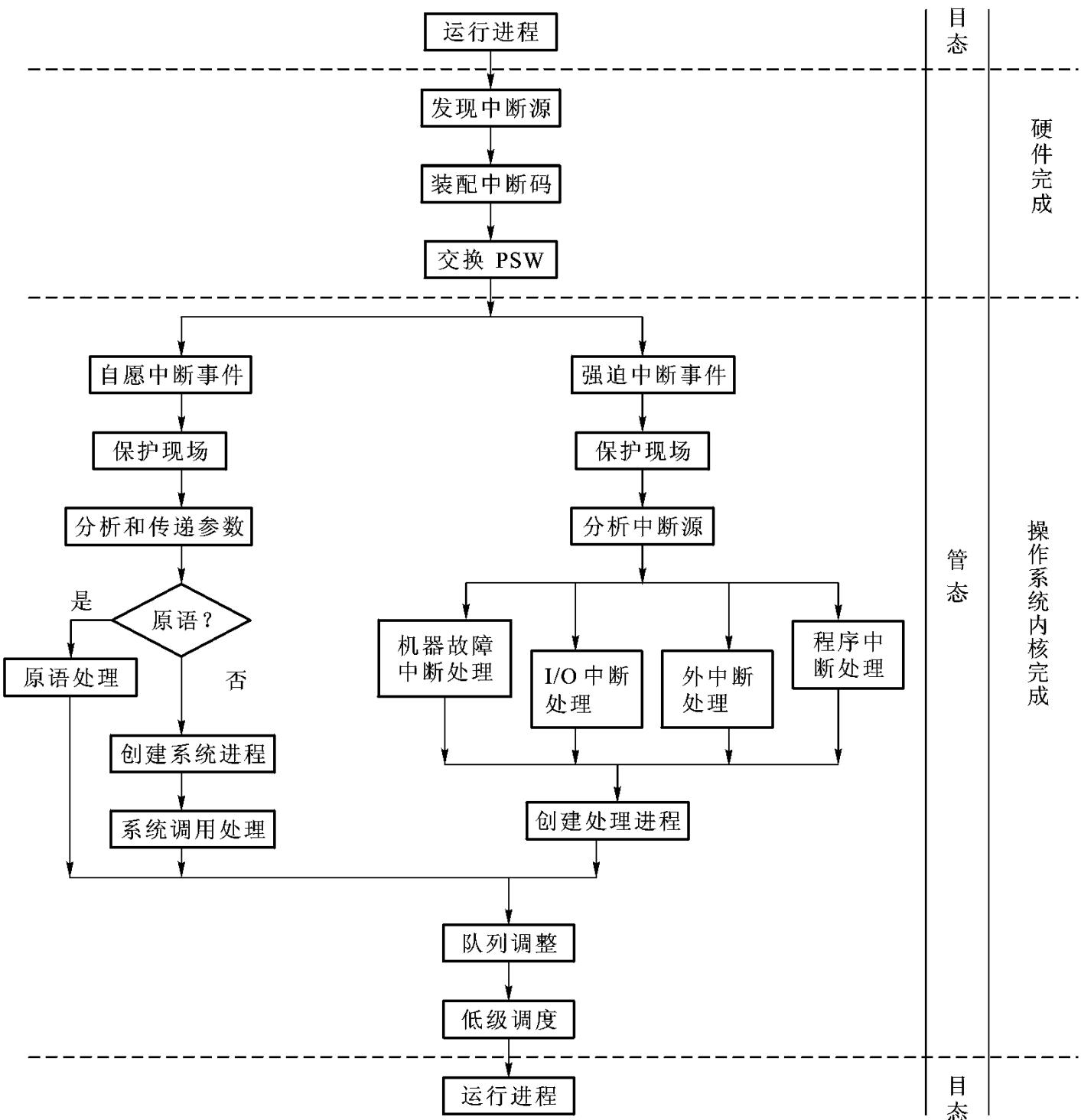


图 1-8 内核被触发和内核处理流程

生中断并被中断机制发现、响应之后由硬件完成执行现场保护,交换程序状态字 PSW。根据发生的中断事件的类型,进入对应的中断处理程序进行处理。对于由访管中断发生的自愿性中断事件,通常区分两类,一类是由内核直接完成的各种原语;另一类需要创建系统进程来完成用户进程请求的系统调用。类似地,强迫性中断事件按不同类型进行机器故障中断、程序性中断、外中断和 I/O 中断处理。中断处理结束后,相关进程的状态也发生了变化,需要修改进程状态,调整进程队列,然后,转向低级调度选择下一个执行进程。可能是从断点恢复被中断的进程运行,也可能是选出新的进程占有处理器执行。这里有两点要加以说明:第一,对于不同的中断机制其中断的响应和处理流程未必一样,图中是以 IBM 中大型机的中断机制为背景来说明的,在采用中断向量结构的 IBM PC 机中,当硬件发现并响应中断时,便保护执行现场并按中断向量号,直接转入该中断事件处理程序。第二,不同结构操作系统中,中断处理进程的实现方式不尽相同,图中实现系统调用的系统进程是在管态下工作的(UNIX 便是这样做的),但采用 C/S 结构的操作系统却把系统调用交给在用户态的服务器进程来实现。类似地,实现各种强迫性中断事件,少量的由内核直接处理,但更多的是由都创建系统进程来完成中断处理。同学们在学习完第二章后,请再回来阅读这一段内容,对中断机制和内核的处理流程一定会有更深切的体会。

为了保证系统的有效性和灵活性,设计内核应遵循少而精的原则。如果内核功能过强,则一方面在修改系统时,可能牵动内核,另一方面它占用的内存量和执行时间都会增大,且屏蔽中断的时间过长也会影响系统效率。因而,设计内核时应注意:中断处理要简单;调度算法要有效;原语应灵活有力,数量适当。这样就可以做到修改系统时,尽量少改动内核,执行时中断屏蔽时间短。

2. 进程

进程是并发程序设计的一个工具,并发程序设计支撑了多道程序设计,由于进程能确切、动态地刻画计算机系统内部的并发性,更好地解决系统资源的共享性,所以,在操作系统的发展史上,进程概念被较早地引入。它在操作系统理论研究和设计实现上均发挥了重要作用。采用进程概念使得操作系统结构变得清晰,主要表现在:(1)一个进程到另一个进程的控制转移由进程调度机构统一管理,不能杂乱无章,随意进行。(2)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,进程无法有意或无意破坏其他进程的数据。因此,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。(3)进程结构较好地刻画了系统的并发性,动态地描述出系统的执行过程,因而,具有进程结构的操作系统结构清晰、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以及系统调度的基本单位。由于每个进程拥有自己

独立的存储空间和运行环境,进程和进程之间并发性进程通信和切换的系统开销相当大,限制了系统中并发执行的进程数目。要更好地发挥硬件提供的能力(如多CPU);要实现复杂的各种并发应用,降低发挥并发性的代价,单靠进程是无能为力的。于是,近年来开始流行多线程(结构)进程(multithreaded process),亦叫多线程。

在一个多线程环境中,进程是系统进行保护和资源分配的单位,而线程则是进程中的一条执行路径,每个进程中允许有多个线程,而线程才是系统进行调度的独立单位。所以,可以把线程也看作是一种构件,它是组成进程构件的更小的构件单位。在一个进程中包含有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程序(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。管程可以被多个进程或管程嵌套调用,但它们只能互斥地访问管程。管程应包含条件变量,当条件不满足时,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一个释放操作为止。由于管程的引入,使得原来分散在进程中的临界区集中了起来统一控制和管理,就可方便对共享资源的使用,使并发进程之间的相互作用更为清晰,更容易编写出正确的并发程序。在第二章中将深入地介绍管程的概念、实现和应用。

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的操作。它仅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。其本身也可以调用其他类程或管程。类程可以看作子程序概念的扩充,类程可以包含多个过程,而子程序只可以包括一个过程。

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,则可以调用管程;若要控制私有资源操作,可以调用类程,这样便于使用高级程序设计语言来书写操作系统。1975年,汉森使用这一方法就成功地在PDP 11/45机上实现了单用户操作系统Solo、处理小作业的作业流系统和过程控制实时调度系统等三个层次管程结构的操作系统。

上面简单讨论了操作系统的构件,采用不同构件和构造方法可组成不同结构的操作系统。从操作系统的体系结构来看,可以把操作系统分成:整体式结构、层次式结构、虚拟机结构和客户服务器及微内核结构。

1.4.2 整体式结构的操作系统

操作系统的整体式结构又叫模块组合法,是基于结构化程序设计的一种软件结构设计方法。早期操作系统(如IBM S/360操作系统)采用这种结构设计方法,主要设计思想和步骤如下:把模块作为操作系统的基本单位,按照功能需要而不是根据程序和数据的特性把整

个系统分解为若干模块(还可以再分成子模块),每个模块具有一定独立功能,若干个关连模块协作完成某个功能。明确各个模块之间的接口关系,各个模块间可以不加控制,自由调用(所以,又叫无序调用法),数据多数作为全程量使用。模块之间需要传递参数或返回结果时,其个数和方式也可以根据需要随意约定;然后,分别设计、编码、调试各个模块。最后,把所有模块连结成一个完整的系统。

这种结构设计方法的主要优点是:结构紧密、组合方便,对不同环境和用户的不同需求,可以组合不同模块来满足,灵活性大;针对某个功能可用最有效的算法和任意调用其他模块中的过程来实现,因此,系统效率较高;由于划分成模块和子模块,设计及编码可齐头并进,能加快操作系统研制过程。它的主要缺点是:模块独立性差,模块之间牵连甚多,形成了复杂的调用关系,甚至有很多循环调用,造成系统结构不清晰,正确性难保证,可靠性降低,系统功能的增、删、改十分困难。随着系统规模的扩大,采用这种结构的系统复杂性迅速增长,这就促使人们去研究操作系统新的结构概念及设计方法。

1.4.3 层次式结构的操作系统

为了能让操作系统的结构更加清晰,使其具有较高的可靠性,较强的适应性,易于扩充和移植,在模块接口结构的基础上产生了层次式结构的操作系统。所谓层次式结构,是把操作系统划分为内核和若干模块(或进程),这些模块(或进程)按功能的调用次序排列成若干层次,各层之间只能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,而且不构成循环调用。

层次结构可以有全序和半序之分。如果各层之间是单向依赖的,并且每层中的诸模块(或进程)之间也保持独立,没有联系,则这种层次结构被称为是全序的。如果各层之间是单向依赖的,但在某些层内允许有相互调用或通信的关系,则这种层次结构称为半序的。

层次结构常常采用由底向上的方法来构造,从裸机 A_0 开始,在它上面添加一层软件,使机器的功能得以扩充,形成了一台功能比原来机器要强的虚拟机 A_1 。又从 A_1 出发,在它上面添加一层新的软件,把 A_1 改造成功能更强的虚拟机 A_2 。就这样“添加——扩充——再添加”,由底向上地增设软件层,每一层都在原来虚拟机的基础上扩充了原有的功能,于是最后实现一台具有所需操作系统各项功能的虚拟机。另一种是自顶向下法,其设计方法与由底向上方法正好相反,从目标系统出发,通过若干层软件过渡到宿主机器,其实质是对目标系统的逐步求精。

在用层次结构构造操作系统时,目前还没有一个明确固定的分层方法,只能给出若干原则,供划分层次中的模块(或进程)时参考。

(1) 应该把与机器硬件有关的程序模块放在最底层,以便起到把其他层与硬件隔离开的作用。在操作系统中,中断处理、设备启动、时钟等反映了机器的特征,因此,与这些特征

有关的程序都应该放在离硬件尽可能近的层次中。这样安排既增强了系统的适应性也有利于系统的可移植性,因为,只需把这层的内容按新机器硬件的特征加以改变后,其他层内容都可以基本不动。

(2) 为进程(和线程)的正常运行创造环境和提供条件的内核程序如 CPU 调度、进程(和线程)控制和通信机构等,应该尽可能放在底层,以支撑系统其他功能部件的执行。

(3) 对于用户来讲,可能需要不同的操作方式,譬如可以选取批处理方式,联机控制方式,或实时控制方式。为了能使一个操作系统从一种操作方式改变或扩充到另一种操作方式,在分层时就应把反映系统外特性的软件放在最外层,这样改变或扩充时,只涉及到对外层的修改,内层共同使用的部分保持不变。

(4) 应该尽量按照实现操作系统命令时模块间的调用次序或按进程间单向发送信息的顺序来分层。这样,最上层接受来自用户的操作系统命令,随之根据功能需要逐层往下调用(或传递消息),自然而有序。譬如,文件管理要调用设备管理,因此,文件管理诸模块(或进程)应该放在设备管理诸模块(或进程)的外层;作业调度程序控制用户程序执行时,要调用文件管理的功能,因此,作业调度模块(或进程)应该放在文件管理模块(或进程)的外层等等。一个操作系统按照层次结构的原则,从底向上可以被安排为:裸机、CPU 调度及其他内核功能、内存管理、设备管理、文件管理、作业管理、命令管理、用户。

Edsger. W. Dijkstra 于 1968 年发表的 THE 多道程序设计系统中第一次提出了操作系统的层次结构设计方法。THE 系统是运行在荷兰的 Electrologica X8 计算机上的一个简单批处理系统,共分 6 个层次。第 0 层完成中断处理、定时器管理和处理器调度,提供多道程序功能。第 1 层是内存和磁鼓管理,为进程分配内存空间,并自动实现内存和磁鼓对换区的数据交换。第 2 层处理进程与操作员间的通信,为每个进程生成一个虚操作员控制台。第 3 层是 I/O 管理,它屏蔽了 I/O 设备的细节,管理信息缓冲区,使进程能方便简单地使用设备。第 4 层是用户(进程)层,他不必考虑进程、内存、控制台、I/O 设备的细节,方便地在计算机上解决问题。第 5 层系统操作员(进程)层。

层次结构的最大优点是把整体问题局部化,由于把复杂的操作系统依照一定的原则分解成若干单一功能的模块(进程),这些模块(进程)组织成层次结构,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)层功能提供了支撑和基础。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)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。这种结构的另一个优点是增加、修改或替换一个层次不影响其他层次,所以,也有利于系统的维护和扩充。然而,层次结构是分层单向依赖的,必须要建立模块(进程)间的通信机制,系统花费在通信上的开销较大,就这一点来说,系统的效率也就会降低。

1.4.4 虚拟机结构的操作系统

虚拟机系统的最早尝试是 IBM 公司的 CP/CMS, 后来改名为 VM/370 (Seawright and MacKinnon, 1979)。这一系统的后继产品今天仍然在 IBM S/390 等大型主机上广泛使用。它基于如下思想, 一个分时系统应该提供以下特性: (1) 多道程序, (2) 一个具有比裸机更方便、界面扩展的计算机。VM/370 的主旨在于将此二者彻底地隔离开来。图 1-9(a) 给出了虚拟机的概念结构。

物理计算机资源通过多重化和共享技术可改变成多个虚拟机。这种技术的基本做法是: 通过用一类物理设备来模拟另一类物理设备, 或通过分时地使用一类物理设备, 把一个物理实体改变成若干个逻辑上的对应物。物理实体是实际存在的, 而逻辑上的对应物是虚幻的、感觉上的。虚拟机监控程序 VM/370 向上层提供了若干台虚拟计算机, 与传统的操作系统不同的是: 这些虚拟计算机不是具有文件管理、设备管理和作业控制之类的虚拟机, 而仅仅是实际物理计算机(裸机)的逻辑复制品。其多重化和共享硬件的做法如下: CPU 调度程序使各个进程共享物理 CPU, 或者说多重化出许多虚 CPU, 每个进程可分得一个; 虚存管理使每台虚 CPU 都有自己的虚存空间; SPOOLING 技术和文件系统提供了虚拟读卡机、穿卡机和行式打印机; 各个用户的终端通过分时使用处理器时间, 提供了虚拟机操作员控制台; 每台虚拟机的磁盘是通过划分物理磁盘若干磁道而形成的, 称作“小盘”。这样一来, 每台复制出来的虚拟计算机包含有: 核心/用户态、中断、CPU、I/O 设备、内存、辅存等, 以及物理计

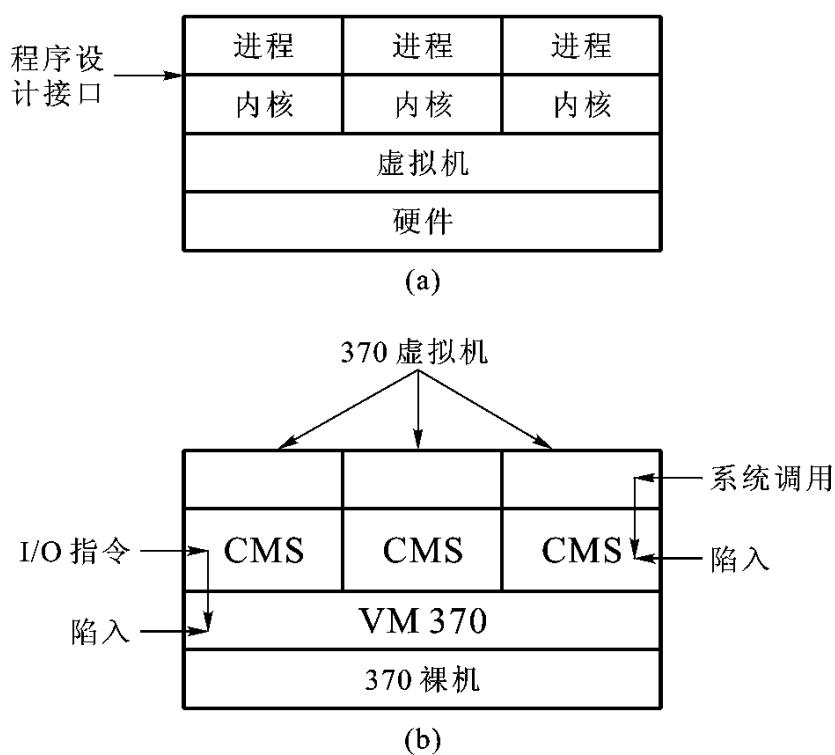


图 1-9

算机具有的全部部件。

因为每台虚拟机都与裸机完全一样,所以,每台虚拟机可以运行裸机能够运行的任何操作系统。不同的虚拟机可以运行不同的操作系统而且往往如此。例如,某些虚拟机运行 OS/360 的后续版本作批处理或事务处理,同时,另一些运行一个单用户交互系统供分时用户使用,该系统称作 CMS(Conversational Monitor System,会话监控系统)。

如图 1-9(b)所示,当 CMS 上的应用程序执行一条系统调用时,该系统调用陷入其自己的虚拟机操作系统 CMS,而不是 VM/370,这就像在真正的计算机上一样。CMS 然后发出正常的硬件 I/O 指令来执行该系统调用。这些 I/O 指令被 VM/370 捕获,随后 VM/370 执行这些指令,作为对真实硬件模拟的一部分。通过将多道程序功能和提供虚拟机分开实现,它们各自都更简单、更灵活和易于维护。

1.4.5 客户/服务器与微内核结构的操作系统

1. 客户/服务器与微内核结构

操作系统结构的改进与计算机硬件软件技术的发展紧密相连,随着网络技术的实用化和数据库联网应用的新趋势,为用户提供一个符合企业信息处理应用要求的分布式环境十分必要,实际上在一个企业或部门中,大部分数据的获取、存储、加工、管理和使用都是在各个就近结点进行的,所以,分布式数据处理最适合客观实际和应用的需要。在这样的运行平台和应用环境下,操作系统的体系结构也在发展和变化。而采用客户/服务器(Client/Server)结构的操作系统,非常适用于网络环境及分布式计算环境。卡耐基 - 梅隆大学研制的 Mach 操作系统较早的成功采用了客户/服务器和微内核结构。客户可以是一个用户程序,也可以是另一个服务器进程,它通过发送一个消息给服务器来请求一项服务。

客户/服务器结构的思想如下:将操作系统分成两大部分,一是运行在用户态并以客户/服务器方式活动的进程;二是运行在核心态的内核。除内核部分外,操作系统的其他部分被分成若干相对独立的进程,每一个进程实现一类服务,称服务器进程。例如,提供文件管理服务、进程管理服务、存储管理服务、网络通信服务等等,用户进程也在该层并以客户/服务器方式活动。由于每个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进行通信,而内核被映射到所有进程的虚拟地址空间内,它就可以控制所有进程。客户进程发出消息,内核将消息传送给服务器进程,服务器进程执行客户提出的服务请求,在满足客户的要求后再通过内核发送消息把结果返回给用户。于是,客户进程与服务器进程形成了客户/服务器关系。由于操作系统的绝大多数功能由用户态进程来实现,内核只完成极少的核心态任务,主要起信息验证、交换的作用,因而称微内核(microkernel),这种结构也就称为客户/服务器与微内核结构。

微内核用水平型代替传统的垂直型结构操作系统, 内核中仅存放那些最基本的核心操作系统功能, 其他服务和应用则建立在微内核之外, 作为独立的服务器进程在用户模式下运行。尽管那些功能应该放在内核内实现, 那些服务应该放在内核外实现, 不同的操作系统设计未必一样, 但事实上过去在操作系统内核中的许多服务, 现在已经成为了与内核交互或相互之间交互的外部子系统, 这些服务主要包括: 设备驱动程序、文件系统、虚存管理器、窗口系统和安全服务。

如图 1-10 所示, 分层结构操作系统的内核很大, 互相之间调用关系复杂。微内核结构则把大量的操作系统功能放到内核外实现, 这些外部的操作系统构件是作为进程来实现的, 它们之间的信息交互均借助微内核提供的消息传递机制实现。这样, 微内核起消息交换功能, 它验证消息, 在构件之间传送消息, 并授权存取硬件。微内核还执行保护功能, 除非允许交换, 否则会阻止信息传递。例如, 当一个应用程序要打开一个文件, 它就传送一个消息给文件服务器; 当它希望建立一个进程或线程, 就发送一个消息给进程服务器。每个服务器进程都可以传送消息给另外的服务器进程。通常微内核只提供: 进程通信、少量内存管理、低层进程管理和调度及低层 I/O 操作在内的最小的服务。微内核结构的优点主要有:

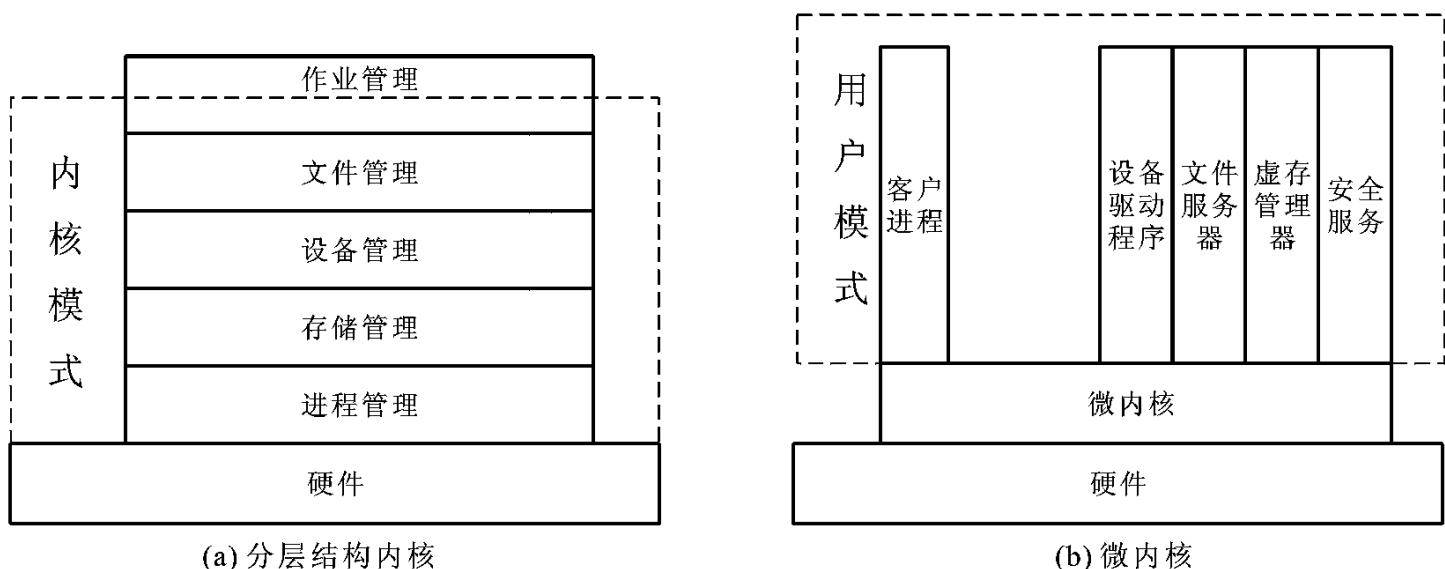


图 1-10 分层结构内核和微内核结构

(1) 一致性接口。微内核结构对进程的请求提供了一致性接口, 进程不必区别内核级服务或用户级服务, 因为, 所有这些服务均借助消息传递机制提供。

(2) 可扩充性。任何操作系统都要增加目前设计中没有的特性, 微内核结构具有较好的可扩充性, 它允许增加新服务, 以及在相同功能范围内提供多种可选服务。例如, 对磁盘上的多种文件组织方法, 每一种可以作为一个服务器进程来实现, 而并不是在内核中实现多种文件服务。每次修改时, 新的或修改过的服务的影响被限制在系统的子集中, 并不需要建

立一个新的内核。

(3) 可移植性。随着各种各样的硬件平台的出现,可移植性成为操作系统的一个有吸引力的特性。在微内核结构中,所有与特定 CPU 有关的代码均在内核中,因而,把系统移植到一个新 CPU 上所作修改较小。

(4) 可靠性。对大型软件产品,较困难的是确保它的可靠性,虽然模块化设计对可靠性有益,但从微内核结构中可以得到更多的好处。较少的微内核代码容易进行测试,较少的 API 接口提高了给内核之外的操作系统服务生成高质量代码的机会。

(5) 支持分布式系统。微内核提供了对分布式系统的支撑,包括通过分布式操作提供的集群控制。当消息从一个客户机发送给服务器进程时,消息必须包含一个请求服务的标识,如果配置了一个分布式系统,所有进程和服务均有惟一标识,并且在微内核级存在一个单一的系统映象。进程可以传送一个消息,而不必知道目标服务进程驻留在哪台机器上。

(6) 支持面向对象的操作系统。微内核结构能在一个面向对象的环境中工作得很好,面向对象方法能为设计微内核以及基于模块化扩充的操作系统提供指导,许多微内核设计时采用了面向对象技术。另外一些系统,如 Windows 2000/XP 操作系统,并不完全依赖于面向对象技术,但在内核设计时结合面向对象原理。

客户/服务器和微内核结构一个潜在缺点是性能问题,发送消息和接收消息需要花费一定的时间代价,所有进程只能通过微内核相互通信,所以它就成为系统的瓶颈。在一个通信频繁的系统中,微内核往往不能提供高效率。例如 GUI 的系统中经常有大量数据及图形信息在不同用户进程间来回复制,那么,把图形引擎作为一个用户态下的服务器进程运行对有着高性能图形需求的系统来说是很不合适的。Windows 2000/XP 的图形引擎便作为核心的一部分,而不作为服务器进程运行。系统的性能与微内核的大小和功能直接有关。一种可行的方法是扩充微内核的功能,把一些关键的服务程序的驱动程序重新放回微内核,减少用户/内核模式切换的次数和进程地址空间切换的次数,这样设计的例子有 Mach 和 Chorus,但这直接提高了微内核的设计代价,损失了微内核在小型接口定义和适应性方面的优点。另一解决方法是把微内核做得更小,通过合理的设计,一个非常小的微内核能提高性能、灵活性及可靠性。典型的第一代微内核结构大小为 300 KB 代码和 140 个系统调用接口。一个小的二代微内核的例子是 L4,它仅有 12 KB 代码和 7 个系统调用。对这些系统的试验表明它们的工作性能要优于采用传统分层结构的操作系统。

2. 微内核的设计

目前存在着多种不同微内核,它们之间规模和功能差别很大,并且不存在一个必须遵守的规则来规定微内核应提供什么功能或基于什么结构实现。因此,在本节中只讨论一个最小化的微内核所应提供的功能与服务。

微内核必须包括那些直接依赖于硬件的功能,以及支撑操作系统用户模式的应用程序

和服务所需的功能,这些功能可概括为:存储管理、进程通信(IPC)、I/O 和中断管理。

(1) 基本存储管理

为了实现进程级的保护,微内核必须控制地址空间的硬件设施。内核负有把每个虚页面映射到物理页框的责任,而大量的存储管理功能,包括进程地址空间之间的相互保护、页面淘汰算法、其他页面控制功能,都能在内核之外来实现。例如,由一个内核之外的虚存模块来决定页的调入和替换,内核负责映射这些页到内存的具体物理地址空间。

页面调度和虚存管理可以在内核外执行,图 1-11 给出了建立在内核外的页面管理程序。例如,当一个应用进程中的一条线程引用了一个不在内存中的页面时,缺页中断发生,执行陷入到内核;内核传送一个消息给页面管理程序,指明要引用的页面;页面管理程序决定装入页面,并预先分配一个页框;页面管理程序和内核必须交互以映射分页管理的逻辑页面操作到物理存储空间;当页面调入后,页面管理程序发送一个恢复消息给应用程序。

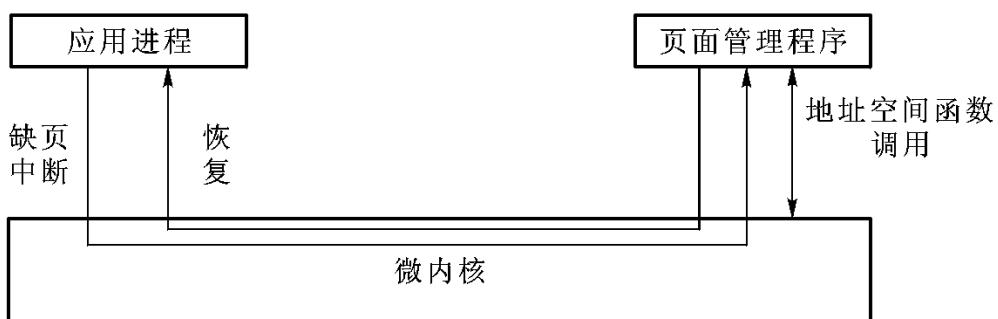


图 1-11 建立在内核外的页面管理程序做缺页处理

这种技术不必调用内核(功能),而让一个非内核进程来映射文件和数据库到用户地址空间,并且与应用相关的内存共享策略也能在内核之外来实现。这里介绍三个基本的微内核操作,以支持内核外部的页面管理和虚存管理:

- 转让(grant)。一个地址空间(进程)的拥有者能够转让它的一些页面给其他进程使用。执行了这个操作之后,内核将从转让进程的地址空间中移去这些页面,并分配给被转让的进程。
- 映射(map)。一个进程可以映射它的任何一个页面到另一进程的地址空间中。执行了这个操作之后,在两个进程间建立了共享存储区,两个进程均可以存取这些页面。
- 刷新(flush)。一个进程能再次回收已经被转让或映射给其他进程的任何页面。

一开始,内核定义所有的物理内存作为由基本系统进程控制的单一地址空间,当建立新进程时,原有的总地址空间中的页面能被转让或映射给新进程,这种模式能支撑同时执行的多重虚拟存储管理。

(2) 进程间通信

在微内核操作系统中,进程通信和线程通信的基本形式是消息。一个消息包括了消息

头标(header)和消息体,头标指明了发送和接收消息的进程,消息体直接包含数据,即一个数据块指针和进程的有关控制信息。进程间通信基于进程之间相关联的端口(Ports),一个端口是一个特定进程的消息队列,与端口相关的是一张能力表,记录了可以与这个进程通信的进程,端口的标识和能力表由内核维护。一个进程通过发送给内核一个指明新端口功能的消息,进程可以允许对自身的新的访问。

值得注意,地址空间不重叠的进程间的消息传递涉及到从存储器的不同区域内的信息的复制,这主要会受到存储器速度的限制,而与CPU速度没有比例关系。因此,当前对操作的研究热衷于基于线程的共享存储区通信和多进程共享页面之类的共享存储器方案。

(3) I/O 和中断管理

对于微内核结构,硬件中断可如同消息一样来处理,地址空间中可能包含I/O端口。微内核能够发现中断,但不能处理它,微内核将生成一个消息传送给用户层中相关的处理中断的进程。因而,当中断出现时,一个特别用户级进程被指派到中断(信号)上,内核将维护这一映射。转换中断为消息的工作必须由微内核来做,但内核中并不包含特定设备专有的中断处理代码。

可以把硬件看作为一组具有惟一线程标识的线程,并且可以发消息给地址空间中的有关软件线程,接收线程决定消息是否来自于一个中断并且确定中断的类型,这类用户级代码的通用结构如下:

```
driver thread;
do
    wait for (mhg, sender);
    if sender = my _ hardware _ interrupt
    {
        read/writer I/O ports;
        reset hardware interrupt
    }
    else ...
while (true);
```

1.4.6 操作系统的运行模型

操作系统本身是一组程序,像其他程序一样也在处理器上执行,那么,操作系统程序是否被组织成进程?它是如何控制和怎样运行的呢?它在什么模式下运行呢?下面来讨论这个问题。从操作系统的运行方式来看,可以把它分成:非进程内核模型、OS功能(函数)在用户进程内执行的模型和OS功能(函数)作为进程执行的模型。

1. 非进程内核模型

许多老式操作系统的采用非进程内核模型,亦即操作系统的功能都不组织成进程来实现。如图 1-12 所示,该模型包括一个较大的操作系统内核程序,进程的执行在内核之外。当中断发生时,当前运行进程的上下文现场信息将被保存,并把控制权传递给操作系统内核。操作系统具有自己的内存区和系统堆栈区,用于控制过程调用和返回。它将在核心态执行相应的操作,并根据中断的类型和具体的情况,或者是恢复被中断进程的现场并让它继续执行,或是转向进程调度指派另一个就绪进程运行。

在这种情况下,进程的概念仅仅是针对用户程序而言的,操作系统代码作为一个独立实体在内核模式下运行。

2. OS 功能(函数)在用户进程内执行的模型

小型机和工作站操作系统(如 UNIX 等)往往采用 OS 的所有功能(函数)在用户进程内执行的模型。如图 1-13 所示,在这种实现模型中,大部分操作系统功能组织成一组例行程序供用户程序调用,认为操作系统例程与用户进程是上下文相关的,操作系统的地址空间被包含在用户进程的地址空间中,因而,操作系统例行程序也在用户进程的上下文环境中执行。图 1-14 给出了 OS 功能(函数)在用户进程内执行的模型中的进程映像,它既包含进程控制块、用户堆栈、容纳用户程序和数据的地址空间等,还包括操作系统内核的程序、数据和系统堆栈区。

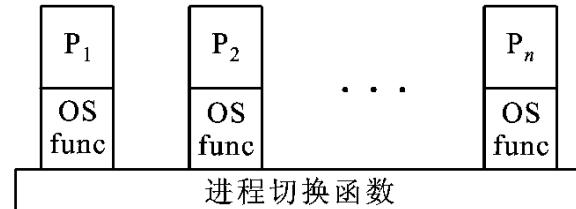


图 1-13 OS 的功能(函数)在用户进程内执行的模型



图 1-14 OS 的所有功能(函数)在用户进程内执行模型的进程映象

当发生一次中断或系统调用后,处理器状态将被置成内核模式,控制从用户进程中被剥夺并传递给操作系统例行程序。此时,发生了模式切换,模式上下文(现场)信息被保存,但是进程上下文切换并没有发生,操作系统仍在该用户进程中执行,提供单独的内核堆栈用于管理进程在核心态下执行时的调用和返回,操作系统例行程序和数据放在共享地址空间,且被所有用户进程共享。

当操作系统例程完成了工作之后,如果应该让当前进程继续运行的话,就可以做一次模式切换来恢复执行原先被中断的用户进程。这种技术提供了不必要通过进程上下文切换就可以中断用户进程来调用操作系统例行程序的手段。如果应该发生进程切换的话,控制就被传递给操作系统的进程切换例行程序,由它来实现进程切换操作,把当前进程的状态置为非运行状态,而指派另一个就绪进程来占有处理器运行。值得指出的是,一些系统中进程切换例行程序是在当前进程中执行的,而另一些系统则不是。

3. OS 功能(函数)作为独立进程执行的模型

OS 功能(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,即操作系统功能是这些系统进程集合运行的结果,这些系统进程也称服务器或服务器进程,于是与用户进程或其他服务器进程之间构成了 Client/Server 关系,Window 2000/XP 采用了这种结构。如图 1-15 所示,除了极少部分功能在内核模式下运行,大部分操作系统功能被组织在一组分离的独立进程内实现,这组进程在用户模式下运行,而进程切换例行程序的执行仍然在进程之外。

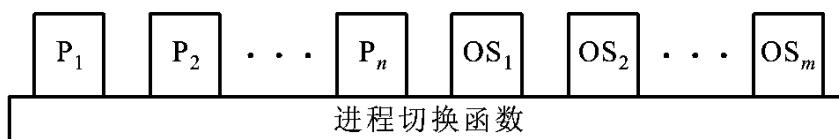


图 1-15 OS 功能(函数)作为独立进程执行的模型

这一实现模型有很多优点。首先,它采用了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,大多数操作系统功能被组织成独立的进程,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,这些程序并不提供给进程特别的服务,仅仅被操作系统调用,把它设计成一个服务器进程,便可赋予一定的优先级,夹在其他进程中运行。最后,这一结构在多处理器和多计算机的环境下非常有效,一些操作系统服务可指派到专门处理器上执行,有利于系统性能的改进。

1.4.7 实例研究:Windows 2000/XP 客户/服务器结构

在一个实际操作系统的设计过程中,要综合考虑来自用户、系统、兼容性等方方面面的因素。Windows 2000/XP 系统结构力图达到如下设计目标:(1)可扩充性。当市场需求变化时,代码必须易于扩充和改动。(2)可移植性。能够在多种体系结构中运行,并能简单地移入新体系结构。(3)可靠性与坚固性。能够防止内部故障和外部侵扰造成的损害。(4)兼容性。与 DOS、Windows 的旧版本兼容,并和一些其他的操作系统如 UNIX、OS2 和 Netware 互操

作。(5)性能。能够达到较高的效率。

为此,Windows 2000/XP 的设计者们认为:(1)采用整体式或层次式操作系统体系结构是不恰当的。它们在可扩充性和可移植性方面效果不好。(2)采用类似于 Mach 的微内核结构也是不恰当的。纯的微内核设计只涉及最小内核,其他服务都运行在用户态,它的运算成本太高,在商业上不适用。Windows 2000/XP 把许多系统服务代码放在核心态运行,包括:文件服务、设备管理、图形引擎等。

因此,Windows 2000/XP 采用基于对象的技术来设计系统,提出了一种客户/服务器系统结构,该结构在纯微内核结构的基础上做了一些扩展,它融合了层次式结构和纯微内核结构的特点。对操作系统性能影响很大的组件放在内核下运行,而其他一些功能则在内核外实现。这种结构的主要优点是模块化程度高、灵活性大、便于维护、系统性能好。图 1-16 简单说明了 Windows 2000/XP 的系统结构。

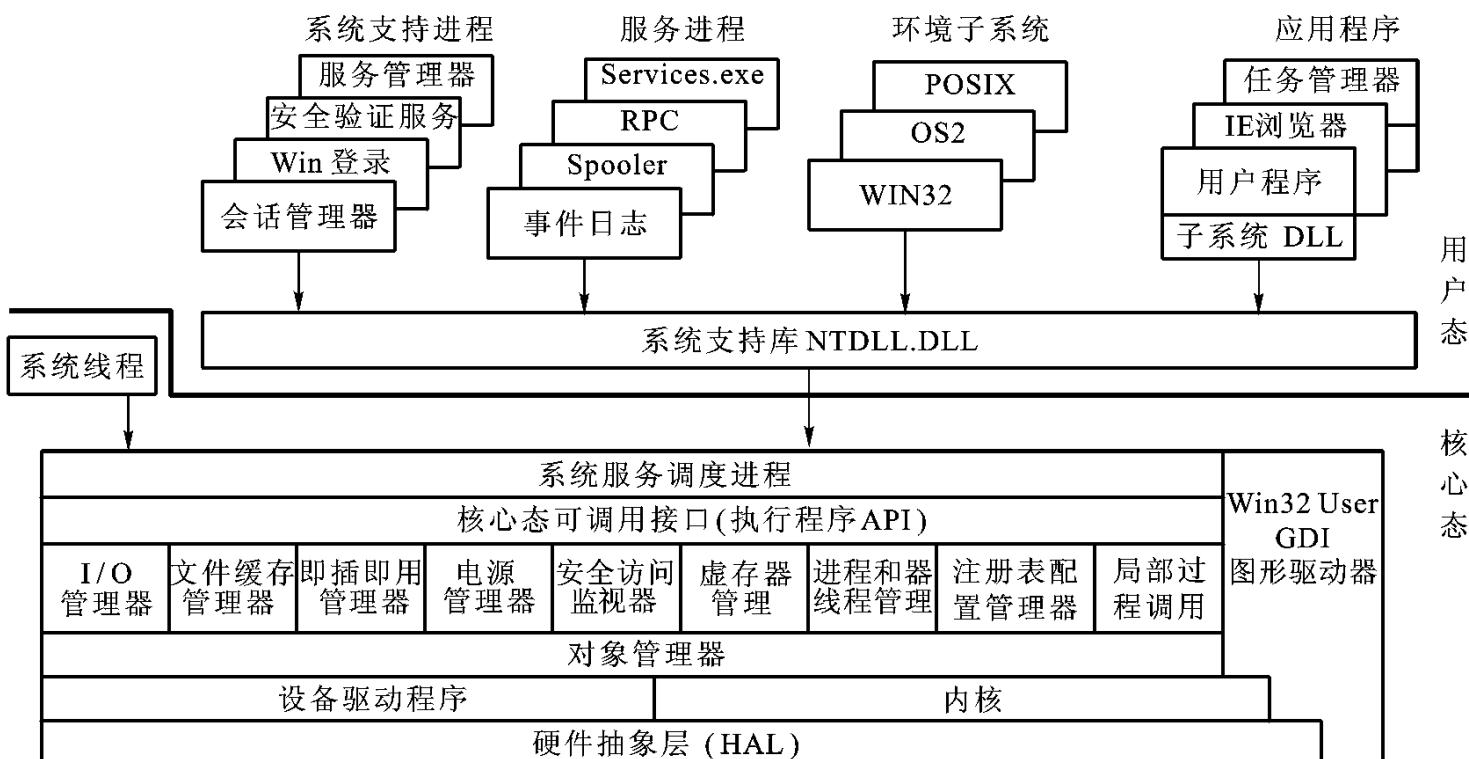


图 1-16 Windows 2000/XP 的系统结构

在核心状态下运行的组件实现最低级的操作系统功能,包括:线程调度、中断和异常调度、多处理器同步等;同时也提供了执行体来实现高级结构的一组例程和基本对象。执行体包含基本的操作系统服务,如内存管理、进程和线程管理、安全控制、I/O 管理及进程通信。在核心态情况下,组件可以和硬件交互,也可以相互交互,不会引起描述表切换和模式转变。所有这些内核组件都受到保护,用户态程序不能直接访问操作系统特权部分的代码数据,这样就不会被错误的应用程序侵扰。除应用程序外,在用户状态下运行的还有系统进程、服务

和环境子系统,他们提供一定的操作系统服务。以下介绍 Windows 2000/XP 的关键系统组件。

(1) 硬件抽象层 HAL

Windows 2000/XP 的硬件抽象层 HAL(Hardware Abstract Level)是实现可移植性的关键部分,位于硬件的最上面和 Windows 2000/XP 的最低层,它在通用的硬件命令和响应与某一特定专用平台的硬件命令和响应之间进行映射,把操作系统的内核、设备驱动程序及执行体从与平台相关的硬件差异中分隔开来。HAL 隐藏各种与硬件有关的细节,例如,系统总线、计时器、I/O 接口、DMA、中断控制器、多处理器通信机制等,对内核来说看上去都是相同的。它是一个可加载的核心态模块 HAL.DLL,是运行在计算机硬件平台上的低级接口。

(2) 设备驱动程序

设备驱动程序是可加载的核心态模块,它们是 I/O 系统和相关硬件之间的接口,把用户的 I/O 函数调用转换成特定硬件设备的 I/O 要求。Windows 2000/XP 的设备驱动程序不直接操作硬件,而是调用 HAL 的某些部分来控制硬件的接口。设备驱动程序包括以下几类:
①硬件设备驱动程序。将用户的 I/O 函数调用转换为对特定硬件设备的 I/O 请求,再通过 HAL 读写物理设备或网络。
②文件系统驱动程序。接受面向文件的 I/O 请求,并把它们转化为对特定设备的 I/O 请求。
③过滤器驱动程序。截取 I/O 并在传递 I/O 到下一层之前执行某些增值处理,如磁盘镜像、加密。
④网络重定向程序和服务器,一类文件系统驱动程序,传输远程 I/O 请求。

Windows 2000/XP 增加了对即插即用和高级电源选项的管理,并使 WDM(Windows Driver Modle)作为标准的驱动程序模型。从 WDM 的角度看,共有三种驱动程序:
①总线驱动程序——用于控制各种总线控制器、适配器、桥、或可连接子设备的设备。
②功能驱动程序——用于驱动主要设备,提供设备的操作接口。
③过滤器驱动程序——用于为一个设备或一个已存在的驱动程序增加功能,或改变来自其他驱动程序的 I/O 请求和响应行为,这类驱动程序是可选的,且可以有任意的数目,它存在于功能驱动程序的上层或下层、总线驱动程序的上层。

在 WDM 的驱动程序环境中,没有一个单独的设备驱动控制某个设备,总线设备驱动程序负责向即插即用管理器报告它上面有的设备,而功能驱动程序则负责操纵这些设备。

(3) 内核

内核执行操作系统最基本的操作,决定操作系统如何使用处理器并确保慎重使用它们。内核和执行体都存在于 NTOSKRNL.EXE,其中,内核在最低层,提供如下一些函数:
①线程管理和调度;
②陷阱处理和异常调度;
③中断处理和调度;
④多处理器同步;
⑤提供由执行体使用的基本内核对象。

内核与执行体在某些方面有所不同,它永远运行在核心态,不以线程方式运行,也不能被其他正在运行的线程中断,是 Windows 中惟一不能被剥夺和调页的部分。为保持效率,代

码短小紧凑,不进行堆栈和传递参数检查。

内核通过执行操作系统机制和避免制定策略而使其自身与执行体的其他部分分开。内核除了执行线程安排和调度外,几乎将所有的策略制定留给了执行体。在内核以外,执行体代表了作为对象的线程和其他可共享的资源。这些对象需要一些策略开销,例如,处理它们的对象句柄、保护它们的安全检查以及在它们被创建时扣除的资源配额。在内核中则要除去这种开销,因为,内核通过一组称作“内核对象”的简单对象来帮助内核控制中央处理器并支持执行体对象的创建。大多数执行体级的对象都封装了一个或多个内核对象及其内核定义属性。

一个称作“控制对象”的内核对象集为控制各种操作系统功能建立了语义。这个对象集包括:内核进程对象、异步过程调用 APC(Asynchronous Procedure Call)对象、延迟过程调用 DPC(Deferred Procedure Call)对象和几个由 I/O 系统使用的对象,例如中断对象。

另一个称作“调度程序对象”的内核对象集负责同步性能并改变或影响线程调度。调度程序对象包括:内核线程、互斥体、事件、内核事件对、信号量、定时器和可等待定时器。执行体使用内核函数创建内核对象的实例,使用它们并构造为用户态提供的更复杂的对象。

内核的另外一个重要功能就是把执行体和设备驱动程序从 Windows 2000/XP 支持的在硬件体系结构之间的变更中提取或隔离开来。这个工作包括处理功能之间的差异,例如,中断处理、异常情况调度和多处理器同步。即使对于这些与硬件有关的函数,内核的设计也是尽可能使公用代码的数量达到最大。内核支持一组在整个体系结构上可移植和在整个体系结构上语义完全相同的接口。大多数实现这种可移植接口的代码在整个体系结构上完全相同。

然而,一些接口的实现因体系结构而异,或者说某些接口的一部分是由体系结构特定的代码实现的。可以在任何机器上调用那些独立于体系结构的接口,不管代码是否随体系结构而异,这些接口的语义总是保持不变。一些内核接口(例如转锁例程)实际上是在 HAL 中实现的。因为,其实现在同一体系结构族内可能因系统而异。

(4) 执行体

Windows 2000/XP 执行体是 NTOSKRNL.EXE 的上层(内核是其下层),执行体包括五种类型的函数:①从用户态被导出并且可以调用的函数。这些函数的接口在 NTDLL.DLL 中,通过 WIN32 API 或一些其他的环境子系统可以对它们进行访问。②从用户态被导出并且可以调用的函数,但当前通过任何文档化的子系统函数都不能使用。这种例子包括本地调用 LPC(Local Procedure Call)和各种查询函数,例如 NtQueryInformationxxx,以及专用函数 NtCreatePagingFile 等。③只能从在 Windows 2000/XP DDK(Driver Development Kit)中已经导出并且文档化的核心态调用的函数。④在核心态组件之间调用的但没有文档化的函数。例如,在执行体内部使用的内部支持例程。⑤组件内部的函数。这里所指的文档化函数是公开的调用接口,而非文档化函数是系统内部调用的例程。

执行体包含下列重要的组件:①进程和线程管理器。创建、跟踪、中止及删除进程和线程。对进程和线程的基本支持在 Windows 2000/XP 内核中实现,执行体给这些低级对象添加附加语义和功能。②虚拟内存管理器。实现“虚拟内存”,把进程地址空间中的虚地址映射成内存页框。从而,为每个进程提供了一个大的专用地址空间,同时保护每个进程的地址空间不被其他进程占用。内存管理器也为高速缓存管理器提供基本的支持。③安全访问监视器。在本地计算机上执行安全策略,它保护了操作系统资源(保护对象包括:文件、进程、地址空间和 I/O 设备),执行运行时对象的保护和监视。④I/O 管理器。提供了应用程序访问 I/O 设备的一个框架,执行独立于设备的输入/输出,并为进一步处理分配适当的设备驱动程序。它还实现了所有的 I/O API,并实施安全性、设备命名。⑤高速缓存管理器。通过将最近引用的磁盘数据驻留在内存中来提高文件 I/O 的性能,并且通过在把更新数据发送到磁盘之前将它们在内存中保持一个短的时间来延缓磁盘的写操作,这样就可以实现快速访问,以提高基于文件的 I/O 性能。正如您将看到的那样,它是通过使用内存管理器对映射文件的支持来做到这一点的。

另外,执行体还包括四组主要的支持函数,它们由上面列出的执行体组件使用,设备驱动程序也使用这些支持函数,这四类支持函数包括:①对象管理器。创建、管理以及删除 Windows 2000/XP 执行体对象和用于代表操作系统资源(如进程、线程、同步对象)的抽象数据类型,例如,进程、线程和各种同步对象。②本地过程调用 LPC(Local Procedure Call)机制。在同一台计算机上的客户进程和服务器进程之间传递信息,强制实施客户器/服务器关系。LPC 是一个灵活的、经过优化的远程过程调用 RPC 版本。③一组公用的“运行时库”函数。例如,字符串处理、算术运算、数据类型转换和安全结构处理。④执行体支持例程。例如,系统内存分配(页交换区和非页交换区)、互锁内存访问和两种特殊类型的同步对象,资源和快速互斥体。此外,还提供窗口管理器:创建面向窗口的屏幕接口,管理图形设备。

(5) 系统支持库 NTDLL.DLL

NTDLL.DLL 是一个特殊的系统支持库,主要用于子系统动态链接。NTDLL.DLL 包含两种类型的函数:①作为 Windows 2000/XP 执行体系统服务的系统服务调度占位程序。②子系统动态链接库、及其他本机映像使用的内部支持函数。

第一组函数提供了可以从用户态调用的作为 Windows 执行体系统服务的接口。这里有 200 多种这样的函数,例如 NtCreateFile、NtSetEvent 等。正如前面提到的那样,这些函数的大部分功能都可以通过 WIN32 API 访问。

对于这些函数中的每个函数,NTDLL 都包含一个有相同名称的入口点,在函数内的代码含有体系结构专用的指令,它能够产生一个进入核心态的转换以调用系统服务调度程序。在进行一些验证后,系统服务调度程序将调用包含在 NTOSKRNL.EXE 内的核心态系统服务。NTDLL 也包含许多支持函数,例如,映像加载程序、堆管理器和 WIN32 子系统进程通信函

数以及运行时的通用库例程。它还包含用户态异步过程调用(APC)调度器和异常调度器。

(6) 系统支持进程

Windows 2000/XP 包括一系列系统支持进程,下面作简单介绍。

- Idle 进程。系统空闲进程,进程 ID 为 0。对于每个 CPU,Idle 进程都包括一个相应的线程,用来统计空闲的 CPU 时间。它不运行在真正的用户态。因此,由不同系统显示实用程序显示名称随实用程序的不同而不同。如任务管理器(Task Manager)中为 System Idle 进程;进程状态(PSTAT.EXE)和进程查看器(PVIEWER.EXE)中为 Idle 进程;进程分析器(PVIEW.EXE)、任务列表(TLIST.EXE)、快速切片(QLICE.EXE)中为 System 进程。

- System 进程和 System 线程。System 进程 ID 为 2,是一种特殊类型的 System 线程的宿主进程。它具有一般用户态线程的属性和描述表,但只运行在核心态,执行加载于系统空间中的代码,而不管它们是在 NTOSKRNL.EXE 中还是在任何其他已经加载的设备驱动程序中。System 线程本身没有用户进程地址空间,必须从系统内存堆中动态分配存储区。

- 会话管理器 SMSS.EXE。是第一个由核心 System 线程在系统中创建的用户态进程,用于执行一些关键的系统初始化步骤,包括:创建 LPC 端口对象和两个线程、设置系统环境变量、加载部分系统程序、启动 WIN32 子系统进程(必要时还有 POSIX 和 OS2 子系统进程)和 WinLogon 进程等。在执行完初始化步骤后,SMSS 中的主线程将等待 CSRSS 和 WinLogon 进程句柄。另外,SMSS 还可作为应用程序和调试器之间的开关和监视器。

- WIN32 子系统 CSRSS.EXE。WIN32 子系统的核心部分。
- 登录进程 WinLogon.EXE。用于处理用户的登录和注销。
- 本地安全身份鉴别服务器进程 LSASS.EXE。接收来自于 WinLogon 进程的身份验证请求,并调用一个适当的身份验证包执行实际验证。

(7) 服务控制器及服务进程

服务控制器是一个运行映象为 SERVICES.EXE 的特殊系统进程,它负责启动、停止和与服务控制器交互,并管理一系列用户态进程服务。服务类似于 UNIX 的守护进程,可以配置成在系统引导时自动启动而不需交互式登录。服务程序是合法的 Win32 映象,这些映象调用特殊的 Win32 函数以与服务控制器相互使用,例如,注册、启动、响应状态请求、暂停或关闭服务。一些 Windows 2000/XP 组件是作为服务来实现的,例如,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基本的执行体系统服务的某些子集提供给应用程序,向用户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。Windows 2000/XP 带有三个环境子系统:WIN32、POSIX(实现了 POSIX.1 标准)和 OS/2 1.2。每个环境子系统包括动态链接库(DLL)。其中,Win32 比较特殊,它必须始终处于运行状态,否则 Windows 2000/XP 就不

能工作, Win32 由下面重要组件构成:①Win32 环境子系统进程 CSRSS, 用来支持控制台窗口、创建及删除进程与线程等;②核心态设备驱动程序, 包括:控制窗口显示、管理屏幕输出、收集有关输入信息、把用户信息传送给应用程序;③图形设备接口(GDI), 用于图形输出设备的函数库, 包括线条、文本、绘图和图形操作函数;④子系统动态链接库;⑤图形设备驱动程序, 包括图形显示驱动程序、打印机驱动程序和视频小端口驱动程序;⑥其他混杂支持函数。

环境子系统又称为虚拟机, 是 Windows 操作系统实现兼容性的重要组成部分, 它的主要任务是接管 CPU 或 OS 的每个二进制代码请求, 将它们转换为 Windows 2000/XP 能成功执行的相应指令。Windows 2000/XP 与其他软件的兼容性主要包括:与应用系统 DOS、Windows、OS/2、LAN Manager 和符合 POSIX 规范的系统的兼容性, 以及与多种文件系统和多种网络的兼容性。

(9) 用户应用程序

用户应用程序可以是 Win32、Windows 3.1、MS – DOS、POSIX 或 OS/2 五种类型之一。在图 1 – 13 中, 请注意“子系统动态链接库”框在“用户应用程序”框之下。在 Windows 2000/XP 中, 用户应用程序(服务进程也是这样)不能直接调用本地 Windows 2000/XP 操作系统服务, 但它们能通过一个或多个“子系统动态链接库”调用。子系统动态链接库的作用是将文档化函数(公开的调用接口)转换为适当的非文档化(系统内部形式)的 Windows 2000/XP 系统服务调用。该转换可能会给正在为用户应用程序提供服务的环境子系统进程发送消息, 也可能不会。

1.5 流行操作系统简介

1.5.1 DOS 操作系统

DOS 的全称是磁盘操作系统(Disk Operating System), 是一种单用户、普及型微机操作系统。主要用于以 Intel 公司的 86 系列芯片为 CPU 的微机及其兼容机, 曾经风靡了整个 20 世纪 80 年代。

DOS 由 IBM 公司和 Microsoft 公司开发, 包括 PC – DOS 和 MS – DOS 两个系列。20 世纪 80 年代初, IBM 公司决定涉足 PC 机市场, 并推出 IBM – PC 个人计算机。1980 年 11 月, IBM 公司和 Microsoft 公司正式签约委托 Microsoft 为其即将推出的 IBM – PC 机开发一个操作系统, 这就是 PC – DOS, 又称 IBM – DOS。1981 年, Microsoft 推出了 MS – DOS 1.0 版, 两者的基本功能一致, 统称 DOS。IBM – PC 的开放式结构在计算机技术和市场两个方面都带来了革命性的变革, 随着 IBM – PC 在 PC 机市场上的份额不断减少, MS – DOS 逐渐成为 DOS 的同

义词,而 PC - DOS 则逐渐成为 DOS 的一个支流。DOS1.0 版于 1981 年随 IBM PC 微型机一起推出,此后的十多年里,随着微机的发展,DOS 也不断更新改进,直到 1994 年推出了最后的版本 DOS6.22。

DOS 的主要功能有:命令处理、文件管理和设备管理。DOS4.0 版以后,引入了多任务概念,强化了对 CPU 的调度和对内存的管理,但 DOS 的资源管理功能比起其他操作系统却简单得多。

DOS 采用汇编语言书写,系统开销小,运行效率高。另外,DOS 针对 PC 机环境来设计,实用性好,较好地满足了低档微机工作的需要。但是,随着 PC 机性能的突飞猛进,DOS 的缺点不断显露出来,已经无法发挥硬件的能力,又缺乏对数据库、网络通信、多媒体等技术的支持,没有通用的应用程序接口,加上用户界面不友善,操作使用不方便,从而,逐步让位于 Windows 等其他操作系统。但是由于用户在 DOS 下开发了大量的应用程序,因而,直到今天新型操作系统都提供对 DOS 的兼容性。

1.5.2 Windows 操作系统

1. Windows 操作系统概况

Microsoft 公司成立于 1975 年,到现在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 Microsoft 公司宣布 Windows 诞生到今天的 WindowsXP,Windows 已经走过了 20 个年头,并且成为风靡全球的微机操作系统。目前个人计算机上采用 Windows 操作系统的占 90%,微软公司几乎垄断了 PC 机软件行业。

图形化用户界面操作环境的思想并不是 Microsoft 公司率先提出的,Xerox 公司的商用 GUI 系统(1981 年)、Apple 公司的 Lisa(1983 年)和 Macintosh(1984 年)是图形化用户界面操作环境的鼻祖。Microsoft 公司于 1983 年 11 月推出 Windows 计划,11 月正式发布其 Windows 产品。但是一开始这一产品很不成功,直到 1985 年 11 月产品化的 Windows 1.01 版才开始投放市场。1987 年又推出 Windows 2.0,这两个版本基本上没有多少用户。1990 年发布的 Windows 3.0 版对原来系统做了彻底改造,在功能上有了很大扩充,从而赢得了用户;到 1992 年 4 月 Windows 3.1 发布后,Windows 逐步取代 DOS 开始在全世界流行。

从 Windows 1.x 到 Windows 3.x,系统都必须依靠 DOS 提供的基本硬件管理功能才能工作,因此,从严格意义上来说它还不能算作是一个真正的操作系统,只能称为图形化用户界面操作环境。1995 年 8 月 Microsoft 公司推出了 Windows 95 并放弃开发新的 DOS 版本,Windows 95 能够独立在硬件上运行,是真正的新型操作系统。以后 Microsoft 公司又相继推出了 Windows 98、Windows 98 SE 和 Windows Me(Microsoft Windows Millennium Edition)等后继版

本。Windows 3.x 和 Windows 9x 都属于家用操作系统范畴, 主要运行于个人计算机系列。

除了家用操作系统版本外, Windows 还有其商用操作系统版本: Windows NT 和早期的 Windows 2000, 它们也是独立的操作系统, 主要运行于小型机、服务器, 也可以在 PC 机上运行。Windows NT 3.1 于 1993 年 8 月推出, 以后又相继发布了 NT 3.5、NT 3.51、NT 4.0、NT 5.0 Beta1 和 Beta2 等版本。基于 NT 内核, Microsoft 公司于 2000 年 2 月正式推出了 Windows 2000。2001 年 1 月 Microsoft 公司又正式宣布停止 Windows 9x 内核的改进, 把家用操作系统版本 Windows Me 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 操作系统还有嵌入式操作系统系列, 包括嵌入式操作系统 Windows CE (Consumer Electronics)、Windows NT Embedded4.0 和带有 Server Appliance Kit 的 Windows 2000 等。

2. Windows 早期版本的技术特点

Windows 3.x 以前的版本在 DOS 操作系统的基础上运行, 那时 Windows 的主要技术特点是:(1)友好、直观、高效的面向对象的图形化用户界面, 易学易用; (2)丰富的与设备无关的图形操作; (3)多任务的操作环境; (4)新的内存管理, 突破了 DOS 系统 1 MB 的限制, 实现了虚存管理; (5)提供各种系统管理工具, 如程序管理器、文件管理器、打印管理器及各种实用程序; (6)Windows 环境下允许装入和运行 DOS 下开发的程序; (7)提供数据库接口、网络通信接口; (8)提供丰富的软件开发工具; (9)采用面向对象的程序设计思想。

3. 家用操作系统 Windows 9x 版本的技术特点

Windows 9x 版本包括 Windows 95、Windows 98、Windows 98 SE 和 Windows Me (Millennium edition) 等版本。与 Windows 早期版本相比, 它能够独立在硬件上运行, 是真正意义上的新型操作系统。

Windows 9x 版本不仅具有更直观的工作方式和优良的性能, 而且还具有支持新一代软硬件需要的强大能力, 其主要技术特点是:(1)独立的 32 位操作系统, 同时也能运行 16 位的程序; (2)真正的多用户多任务操作系统, 在 32 位下具有抢先多任务能力; (3)提供“即插即用”功能, 系统增加新设备时, 只需把硬件插入系统, 由 Windows 解决设备驱动程序的选择和中断设置等问题; (4)支持新的硬件配置, 如 USB (Universal Serial BUS)、AGP (Accelerated Graphics Port)、ACPI (Advanced Configuration and Power Interface) 和 DVD; (5)多媒体支持, 包括: MPEG 音频、WAV 音频、MPEG 视频、AVI 视频和 Apple Quiet Time 视频。 (6)有内置网络功能, 直接支持联网和网络通信, 同时也提供环球邮箱和对 Internet 的访问工具; (7)新的图形化界面, 具有较强的多媒体支持功能; (8)支持 FAT32 文件系统。

4. 商用机操作系统 Windows NT 的技术特点

在 Windows 发展过程中, 硬件技术和系统性能在不断进步, 如基于 RISC 芯片和多 CPU

结构的微机的出现;客户机/服务器模式的广泛采用;微机存储容量增大及配置多样化;同时,对微机系统的安全性、可扩充性、可靠性、兼容性等也提出了更高的要求。1993年推出的Windows NT(New Technology)便是为这些目标而设计的。除了Windows产品的上述功能外,它还有以下的技术特点:(1)支持对称多处理SMP和多线程,即多个任务可基于多个线程对称地分布到各个CPU上工作,具有良好的可伸缩性,从而大大提高了系统性能;(2)支持抢先的可重入多任务处理;(3)32位页式授权虚拟存储管理;(4)支持多种API(常用及标准Win32、OS/2、DOS和POSIX API等)和源码级兼容性;(5)支持多种可装卸文件系统,包括DOS的FAT文件系统、OS2的HPFS、CD-ROM文件系统CDFS和NT文件系统NTFS;(6)具有各种容错功能,C2安全级;(7)可移植性好,可在Intel x86、PowerPC、Digital Alpha AXP、以及MIPS RISC等平台上运行,既可作为网络客户,又可提供网络服务;(8)集成网络计算,支持LAN Manager,为其他网络产品提供接口;(9)能与其Microsoft SQL Server结合,提供C/S数据库应用系统的最好组合。

5. Windows 2000 和 Windows XP

Windows 2000是在Window NT基础上修改和扩充而成的,能充分发挥当今32位微处理器的硬件能力,在处理速度、存储能力、多任务和网络计算支持诸方面与大型机和小型机进行竞争。它不是单个操作系统,而包括了四个系统用来支持不同对象的应用。“专业版Windows 2000 Professional”为个人用户设计,可支持2个CPU,最大内存可配4GB;“服务器版Windows 2000 Server”为中小企业设计,可支持4个CPU,最大内存4GB;“高级服务器版Windows 2000 Advanced Server”为大型企业设计,支持8个CPU和最大8GB内存;“数据中心服务器版Windows 2000 Datacenter Server”专为大型数据中心开发,支持最多32个CPU和最大64GB内存。Windows 2000除继承Windows 98和NT的特性外,在与Internet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,它还支持新的即插即用和电源管理功能,提供活动目录技术,支持两路到四路对称式多处理器系统,全面的Internet应用软件服务。

Windows XP是一个把家用操作系统和商用操作系统融合为一的操作系统,它将结束Windows两条腿走路的历史,包括家庭版、专业版和一系列服务器版。它具有一系列运行新特性,具备更多的防止应用程序错误的手段,进一步增强了Windows安全性,简化了系统的管理与部署,并革新了远程用户工作方式。

1.5.3 UNIX 操作系统家族

1. UNIX 操作系统的历史与发展

UNIX操作系统是一个通用、交互型分时操作系统。它最早由美国电报电话公司贝尔实

验室的 Kenneth Lane Thompson 和 Dennis MacAlistair Ritchie 于 1969 年在 DEC 公司的小型系列机 PDP - 7 上开发成功, 1971 年被移植到 PDP - 11 上。1973 年 Ritchie 在 BCPL (Basic Combined Programming Language, M. Richard 于 1969 年开发) 语言基础上开发出 C 语言, 这对 UNIX 的发展产生了重要作用, 用 C 语言改写后的第 3 版 UNIX 具有高度易读性、可移植性, 为迅速推广和普及走出了决定性的一步。1974 年 7 月,《The UNIX Time - Sharing System》一文在美国权威杂志《Communication of ACM》上发表, 引起了广泛注意。最早外界可获得的 UNIX 是 1975 年的 UNIX 第 6 版; 1978 年的 UNIX 第 7 版, 可以看作当今 UNIX 的先驱, 该版为今天 UNIX 的繁荣奠定了基础。70 年代中后期 UNIX 源代码的免费扩散引起了很多大学、研究所和公司的兴趣, 大众的参与为 UNIX 的改进、完善、传播和普及起到了重要的作用。

BSD UNIX 一直在学术界中占据主导地位, 而 AT&T 的 UNIX 则成为商业领域的领头羊。AT&T 公司发布的 UNIX 版本包括 1981 年的 System III, 1983 年的 System V (基于变种产品程序员工作平台 PWB), 1984 年的 System V .2, 和 1987 年的 System V .3。Microsoft 公司和 SCO 公司在 1978 年开始研制的 PC 机上的 UNIX 商业版变种 XENIX 也曾经是 UNIX 的一个重要流派, 它与 AT&T UNIX 在使用标准上会合于 SVR3.2 版本。由大学开发的非 AT&T 系统的 UNIX 是美国加州大学 Berkeley 分校开发的 UNIX BSD (Berkeley Software Distribution, BSD), 最初在 PDP 机上运行, 后来移植到 VAX 机上, 其中, 比较著名的版本有 1978 年的 1BSD 和 2BSD、1979 年 3BSD 和 1980 年之后的 4.0BSD、4.1BSD、4.2BSD 和 4.3BSD, 由于资金的原因, UNIX BSD 的最后版本是 1993 年发布的 4.4BSD。这些版本中加入了页式虚存存储管理、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 的发展中起了重要作用, 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 4BSD。在标准上, 4.3BSD 的代表 Sun OS 和 AT&T UNIX SVR3.2 会合于 SVR4.0 (System V Release 4) 版本 (1989 年)。它是现有最重要的 UNIX 变种, 合并了 BSD 和以往 UNIX 的许多功能, 并以一种完整的、有商业生存力的方式实现这些功能。

UNIX 取得成功的最重要原因是系统的开放性, 公开源代码, 用户可以方便地向 UNIX 系统中逐步添加新功能和工具, 这样可使 UNIX 越来越完善, 能提供更多服务, 成为有效的程序开发的支撑平台。它是目前惟一可以安装和运行在从微型机、工作站直到大型机和巨型机上的操作系统。

UNIX 的主要特点:(1)多用户多任务操作系统, 用 C 语言编写, 具有较好的易读、易修改和可移植性;(2)结构分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)剥夺式动态优先级 CPU 调度, 有力地支持分时功能;(6)命令语言丰富齐全, 提供了功能强大的 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, 内存利用率高。

实际上, UNIX 已成为操作系统的一种标准, 而不是指一个具体操作系统。许多公司和

大学都推出了自己的 UNIX 系统,如 IBM 公司的 AIX 操作系统,SUN 公司的 Solaris 操作系统,Berkeley 大学的 UNIX BSD 操作系统,DEC 公司的 Digital UNIX 操作系统(并入 Compaq 称 Tru64 UNIX),HP 公司的 HP - UX 操作系统,SGI 公司的 Irix 操作系统,SCO 公司的 SCO UNIXWare 和 Open Server 操作系统以及 AT&T 公司自己的 SVR 操作系统等。到了 20 世纪 90 年代,UNIX 版本多达 100 余个,各个版本互不兼容,非常混乱。为了使同一个程序能在所有不同 UNIX 版本上运行,IEEE 拟定了一个 UNIX 标准,称作 POSIX。它定义了相互兼容的 UNIX 系统必须支持的最少系统调用接口和工具。该标准已被多数 UNIX 支持,同时,其他一些操作系统也在支持 POSIX 标准。此外,还有一些 UNIX 标准和规范,如 SVID (System V Interface Definition)、XPG (X - open Portability Guideline),这些均进一步推动了 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 那样得到如此广泛的流行,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了深远的影响,为此,Ritchie 和 Thompson 共同获得了 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

SunMicrosystem 公司开发的 Solaris 是 UNIX SVR4 的变种,是一个支持完全对称多处理器、多任务和混合式多线程策略的 32 位分布式计算环境。目前 Solaris2.x 是一个可移植操作系统,既可以运行在 SUN 和 OEM SPARC 平台上,也可以运行在标准的 Intel 80x86 平台上。

Solaris 的设计追求三个目标:一是高性能,通过多用户、多任务、SMP 和多线程能力,最大限度地发挥硬件的潜力;二是满足用户变化的需要,建立分布式 C/S 解决方案,允许企业用户利用企业计算环境的能力和资源,更加高效率地解决业务问题;三是支持异构计算环境,用户可以从现有的信息技术投资中得到最大回报。

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 BSD 4.1 的一个移植版本,运行在 MOTOROLA 680X0 平台上,支持 1 个 MIPS 的处理器和大约 1M 内存。

20 世纪 80 年代中期,网络化的 UNIX 正在流行,网络化成为 SUN 计算战略的重要部分。SUN 在网络开发技术方面开展了有重大意义的工作,并于 1985 年推出了 SUN OS 2.0 版本,它能够支持分布式、基于网络的计算。这些工作包括构建分布式应用程序接口(远程过程调用 RPC),通过网络共享数据的操作系统程序(网络信息系统 NIS),以及网络文件系统 NFS。SUN NFS(Network File System)是 1984 年推出的一个网络文件系统产品,如今已成为一种事实上的标准。NFS 提供了在异种机、异种操作系统的网络环境下共享文件的简单有效的方法。NFS 基于 C/S 模式,使用 UDP 协议和 RPC(Remote Procedure Call)机制,主要特点有:(1)提供透明的文件访问和文件传送,用户使用本地或远程的文件没有区别,真正实现了分布式数据处理;(2)易于扩充,扩充新的资源或软件,不需改变现有工作环境;(3)性能高、可靠性好、具有可伸缩性。

SUN 于 1988 年推出了 SUN OS 4.0, 把运行平台从 MOTOROLA 680X0 平台迁移到 SPARC 平台, 并开始支持 Intel 平台; 1990 年又推出了 SUN OS 4.1, 1992 年推出了 SUN OS 4.1.3, 这些版本中出现了对异步多处理器 ASMP 的支持, 它的内核每次只在一个处理器上运行, 而同时可以在可用的任意多个处理器上调度用户程序。

随着多处理器系统市场按照预期的不断增长, SUN 将开发的重点放在解决多处理器扩展性的新操作系统内核上。新内核允许执行多线程, 并提供了在进程级的线程化功能。基本的操作系统变化伴随着一个新的命名的出现, 这就是 1992 年 SUN 发布的 Solaris 2.0。新的 Solaris 2.0 操作环境还符合 SVR4 标准, 并采用了模块化的实现方法。

作为一个系列化的操作环境, SUN OS 4.x 版本被称为 Solaris 1.x, 而 Solaris 2.x 版本又被称为 SUN OS 5.x。从 1992 年开始, Sun 公司相继推出了 Solaris 2.1 到 Solaris 2.6 版本, 具备了越来越多的功能。1998 年后, Sun 公司推出 64 位操作系统 Solaris 2.7 和 2.8, 在网络特性、可靠性、兼容性、互操作性、易于配置和管理方面均有很好改进。

1.5.4 自由软件和 Linux 操作系统

1. 自由软件

自由软件 (Free Software or Freeware) 是指遵循通用公共许可证 GPL (General public License) 规则, 保证您有使用上的自由, 获得源程序的自由, 可以自己修改的自由, 可以复制和推广的自由, 也可以有收费的自由的一种软件。

自由软件的出现意义深远。众所周知, 科技是人类社会发展的阶梯, 而科技知识的探索和积累是组成这个阶梯的一个个台阶。人类社会的发展是以知识积累为依托的, 不断地在前人获得知识的基础上发展和创新才得以一步步地提高。软件产业也是如此, 如果能把已有的成果加以利用, 避免每次都重复开发, 将大大提高目前软件的生产率, 借鉴别人的开发经验, 互相利用, 共同提高。带有源程序和设计思想的自由软件对学习和进一步开发软件, 起到极大促进作用。自由软件的定义就确定了它是为了人类科技的共同发展和交流而出现的。free 指的是自由, 但并不是免费。自由软件之父 Richard Stallman 先生将自由软件划分为若干等级, 其中, 自由之 0 级是指对软件的自由使用; 自由之 1 级是指对软件的自由修改; 自由之 2 级指对软件的自由获利。

自由软件赋予人们极大的自由空间, 但这并不意味自由软件是完全无规则的, 例如 GPL 就是自由软件必须遵循的规则, 由于自由软件是“贡献型”, 而不是“索取型”的, 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。

GNU 的含义是 GNU is not UNIX 的意思, 由自由软件的积极倡导者 Richard Stallman 指导并启动的一个组织, 同时拟定了通用公用许可证协议 GPL。20 世纪 70 年代后期起很多软件

不再提供源码,使用户无法修改软件中的错误。为了摆脱商业软件公司对软件特别是对操作系统软件的控制,自 1984 年起,MIT 开始支持 Stallman 启动的 GNU 计划,并成立了自由软件基金会 FSF(Free Software Foundation)。他通过 GNU 写出一套和 UNIX 兼容,但同时又是自由软件的 UNIX 系统,GNU 完成了大部分外围工作,包括外围命令 gcc/gcc ++ 和 shell 等,最终 Linux 内核为 GNU 工程划上了一个完美句号,现在所有工作继续在向前发展。目前人们熟悉的一些软件如 gcc/gcc ++ 编译器、Objective C、Free BSD、Open BSD、FORTRAN77、C 库、BSD email、BIND、Perl、Apache、TCP/IP、IP accounting、HTTPserver、Lynx Web 都是自由软件的经典之作和著名软件。

GPL 协议可以看成为一个伟大的协议,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 就没有今天的自由软件。

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中,自己动手编写了一个操作系统原型,从此,一个新的操作系统诞生了。Linus 把这个系统放在 Internet 上,允许自由下载,许多人对这个系统进行改进、扩充、完善,许多人做出了关键性贡献。Linux 由最初一个人写的原型变化成在 Internet 上由无数志同道合的程序高手们参与的一场运动。

Linux 属于自由软件,而操作系统内核是所有其他软件最基础的支撑环境,再加上 Linux 的出现时间正好是 GNU 工程已完成了大部分操作系统外围软件,水到渠成,可以说 Linux 为 GNU 工程画上了一个圆满句号。短短几年, Linux 操作系统已得到广泛使用。1998 年, Linux 已在构建 Internet 服务器上超越 Windows NT。计算机的许多大公司如 IBM、Intel、Oracle、Sun、Compaq 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 下的应用软件越来越多。如今, Linux 的中文版已开发出来, Linux 已经开始在中国流行。同时,也为发展我国的自主操作系统提供了良好条件。

Linux 是一个开放源代码, UNIX 类的操作系统。它在继承了历史悠久和技术成熟 的 UNIX 操作系统的特点和优点外,还作了许多改进,成为一个真正的多用户、多任务的通用操作系 统。1993 年,第一个产品版 Linux1.0 问世的时候,全部按自由扩散版权进行扩散,即公开源码,不准获利。不久发现这种纯粹理想化的自由软件会阻碍 Linux 的扩散和发展,特别限制了商业公司参与并提供技术支持的积极性。于是 Linux 转向 GPL 版权,除允许享有自由软件的各项许可权外,还允许用户出售自由软件拷贝程序。这一版权上的转变后来证明对 Linux 的进一步发展十分重要。

从 Linux 的发展史可以看出,是 Internet 孕育了 Linux,没有 Internet 就不可能有 Linux 今天的成功。从某种意义上来说, Linux 是 UNIX 和 Internet 国际互联网结合的一个产物。自由软件 Linux 是一个充满生机,已有巨大用户群和广泛应用领域的操作系统,看来它是惟一能

与 UNIX 和 Windows 较量和抗衡的一个操作系统了。

Linux 技术特点如下:(1)继承了 UNIX 的优点,又有了许多更好的改进,开放、协作的开发模式,是集体智慧的结晶,能紧跟技术发展潮流,具有极强的生命力;(2)真正的多用户、多任务通用操作系统,可作为 Internet 上的服务器,可用做网关路由器,可用做数据库、文件和打印服务器,也可供个人使用;(3)全面支持 TCP/IP,内置通信联网功能,并方便地与 LAN Manager、Windows for Workgroups、Novell Netware 网络集成,让异种机方便地联网;(4)符合 POSIX1003.1 标准,各种 UNIX 应用可方便地移植到 Linux 下,反之也是一样。支持 DOS 和 Windows 上应用,对 UNIX BSD 和 UNIX System V 应用程序提供代码级兼容功能。也支持绝大部分 GNU 软件;(5)是完整的 UNIX 开发平台,支持一系列 UNIX 开发工具,几乎所有主流语言如 C、C++、Fortran、Ada、PASCAL、Modual2 和 3、SmallTalk 等都已移植到 Linux 下;(6)提供庞大的管理功能和远程管理功能,支持大量外部设备;(7)支持 32 种文件系统,如 EXT2、EXT、XI AFS、ISO FS、HPFS、MS DOS、UMS DOS、PROC、NFS、SYSV、Minix、SMB、UFS、NCP、VFAT、AFFS 等;(8)提供 GUI,有图形接口 X – Window,有多种窗口管理器;(9)支持并行处理和实时处理,能充分发挥硬件性能;(10)开放源代码,可自由获得,在 Linux 平台上开发软件成本低,有利于发展各种特色的操作系统。

1.5.5 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 IBM (International Business Machines Corporation) 成立于 1914 年,是当今世界上最大的 IT 跨国公司之一,在全球有数十万雇员,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 开创,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、开发出迄今为止最小的硬盘、开发出迄今为止最快的商用机。人类许多重大事件都有 IBM 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 Deep Blue 战胜世界象棋冠军 Garry Kasparov、IBM 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 研制开发和生产销售的 IT 产品包罗万象、品种繁多。从主机到外设、由系统软件到应用软件、从原始到先进、由简单到成熟,功能逐步完善,性能不断改进。IBM 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。随着硬件技术的发展,从计算机操作管理软件 FMS (Fortran Mornitor System)、IBM SYS(IBM7094 Mornitor),到 OS/360 操作系统和磁盘操作系统 DOS/360,支持 Virtual Machine 的 VM/370 操作系统,微型机操作系统 DOS 和 OS2,以及今天的新型操作系统也得到了很大的发展。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) 操作系统是一种超强设计的重负载高端 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。战胜世界象棋冠军 Kasparov 的超级计算机 Deep Blue、迄今为止最快的巨型机之一 Blue Pacific 上运行的都是 AIX 操作系统。

AIX 操作系统于 1990 年推出, 目前最新版本是 AIX 5L。它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, 配合相关硬软件后, 可以全年不停机工作。它提供了一个安全的图形化界面的多用户环境, 支持多线程、支持动态装卸设备驱动程序、虚存空间可达 252 字节, 网络特性出色、管理工具多样, 各种语言支持、商用 UNIX 软件大都可在其上运行, 许多自由软件可以移植过来。世界上许多 Internet 网站和研究中心都采用 RS/6000 及 AIX 系统, 主要用在 FTP、email、Web 服务器、数据库服务器, 及 CAD、CAE、可视化等各种科学和工程应用。

3. OS/390、VM 和 DOS/VSE 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) 操作系统和 DOS/VSE (Disk Operating System/Virtual Storage Extended) 操作系统。随着 2000 年 12 月推出 IBM Z900 系列大型主机, 2001 年 3 月又发布了 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 年的不断改进, IBM S/390 已成为有高度可靠性、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。由于中小型服务器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, 用一台或几台大型机代替众多的中小服务器已成为 IT 业务的大势所趋。这也反映了计算机应用遵循“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 是当今世界上第一个使用铜质互联芯片技术的企业级服务器, 运行速度达 1600MIPS。

OS/390 的前身是著名的 MVS (Multiple Virtual Storages) 操作系统。1996 年 IBM 宣布 OS/390 1.1 版, 1998 年 IBM 宣布 OS/390 2.5 版, 目前最新版本是 OS/390 2.7 版。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 上开发的所有应用程序。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, 集成了防火墙提高网络安全性。OS/390 采用了面向对象程序设计

计技术、并行处理技术、分布式处理技术、Client/Server 技术,具有较强的交互操作性、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 下运行的程序;MVS/ESA390(Enterprise System Architecture, ESA)模式可支持 10 个 240 MB 内存和 256 个通道的多处理器系统;ESA/390LPAR 模式可把计算机从逻辑上分成(Logical Partitioning)最多 10 个部分(有些多 CPU 型号甚至可分成 20 个 LPAR)每个部分有自己的 CPU、内存和通道,且分别运行不同操作系统,如 OS/370、MVS/ESA370 和 MVS/ESA390,也可以运行 IBM 虚拟机操作系统 VM 和虚存扩充操作系统 DOS/VSE。

4. OS/400 操作系统

AS/400 服务器(Advanced System/400)首次采用 64 位的 RISC 技术,主要运行 OS/400 操作系统,也可以运行 SSP(IBM System Support Program)操作系统。

AS/400 服务器是 IBM 开发的最新中型商用机器,既可以用在旅行、家庭、小型办公室中,也可以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了四层软件: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 Internet 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)是一种逻辑上的(不是物理上的),把上层软件与底层软硬件完全隔离开来的一种系统接口。它提供了一个 API 集,为 OS/400 和所有应用软件访问底层资源提供了惟一的途径。许可证内部代码(Licensed Internal Code, LIC)由 IBM 提供并在提交系统之前预先安装在 AS/400 上的一组用户不可见指令,用户程序需经硬件自动转换成 LIC 才能被 CPU 执行。

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,该数据库对用户是透明的,使用起来方便高效,目前最新版本是 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,它有很高的安全性(C2 level security),已获得了美国联邦机构的认证。目前世界上装机达数十万台以上。OS/400 支持系统逻辑划分,每个逻辑划分有独立的处理器、内存和辅存,适合许多商用业务应用。

5. PC 微型机的操作系统

PC 微型机运行 DOS、OS2、Windows 等操作系统。

OS/2 是 Microsoft 公司和 IBM 公司合作于 1987 年开发的配置在 PS/2 微机上的图形化用户界面操作系统。目前 IBM 公司继续在研究和开发 OS/2 2.0,并于 1994 年推出了高性能(新的文件系统和内存管理)、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, 它的功能和性能与 Windows 95 相当。1996 年, 推出网络操作系统 OS/2 最新版本 OS/2 Warp Server4.0 和 OS/2 Warp Server SMP, 后者为对称多处理器版, 支持 TCP/IP 协议, 支持 6 国语言的声音 I/O 的 Navigator、Java 运行服务。

OS/2 的主要特点有: (1) 采用图形化用户接口, 操作直观方便; (2) 可以在 16 位和 32 位两种 CPU 上工作; (3) 使用虚存可扩充到 4 GB; (4) 引入会话、进程、线程概念, 实现多任务控制; (5) 提供高性能文件系统, 采用长文件名和扩展文件属性; (6) 提供应用程序设计接口 (API), 可以支持多任务、多线程和动态连接; (7) 具有对 MS – DOS 的向上兼容性, MS – DOS 的文件可在 OS/2 下读写。

1.5.6 其他流行操作系统

1. Mach 操作系统

Mach 的前身是由 Carnegie-Mellon 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。这一研究计划得到了美国国防部 ARPA 的支持, 从而, Mach 系统得到了进一步的发展, 推出的相应升级版本也确保了与 Berkeley UNIX 系统的完全兼容, 而这正是 ARPA 的一个重要战略目标。

Mach 最早的版本于 1986 年推出的运行在具有四个 CPU 的 VAX 11/784 系统上, 接着完成了向 IBM PC/RT 及 Sun3 的移植工作。在 1987 年, Mach 也被移植到 Encore 及 Sequent 多处理机环境。之后不久, 在计算机供应商 IBM、DEC 和 HP 的联合领导下成立了开放软件基金会 (Open Software Foundation) 联盟, 其目的是改变 UNIX 对他们用户的控制, 而 UNIX 的属有者 AT&T 公司当时正同 Sun Microsystem 紧密合作开发 System V。OSF 选择了 Mach2.5 作为它的第一个操作系统 OSF/1 的基础。Mach2.5 内核比较庞大且非常单一, 这是因为系统内核中存在大量 Berkeley UNIX 代码的缘故, 在 1988 年, CMU 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 组成的特点。

Mach 的设计目标为: (1) 为建造其他操作系统而提供一个基础 (如 UNIX 类操作系统); (2) 支持大型稀疏地址空间; (3) 允许对网络资源的透明访问; (4) 从系统与应用两个方面开发并行性; (5) 使 Mach 可以被移植到有更多数量机器的系统中。这些目标中既包含了研究也包含了开发, 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机和分布式系统。

Mach 采用的主要技术有: (1) 微内核结构, 使用面向对象的设计方法, 引入了进程、线程、内存、端口、消息等对象概念; (2) 设计了基本调度、传递调度和相关调度等高效的多处理器调度算法; (3) 提供了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;(4)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,但却在没有物理共享内存的机器上运行;(5)采用通信端口来支持异步消息传递、RPC、位流以及其他的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的操作系统,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,Apple 公司也就成为当时惟一能与 IBM 公司抗衡的 PC 机生产公司。Mac OS 的主要技术特点有:(1)采用面向对象技术;(2)图形化用户界面;(3)虚拟存储管理技术;(4)应用程序间的相互通信;(5)强有力的多媒体功能;(6)简便的分布式网络支持;(7)丰富的应用软件。

Macintosh 计算机的主要应用领域为: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 PC 网络产品制造商之一,成立于 1983 年。它的网络产品可以和 IBM、Apple、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 NOS(Networking Operating System)。

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,可靠性高;具有良好的权限管理,安全保密性好;具有开放性,提供开放的开发环境。Netware 有多种版本,最新版本为 Netware5。

4. 教学操作系统 Minix

UNIX 早期版本的源代码可以免费获得并被加以广泛的研究,世界上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,并禁止在课程中研究其源代码,以免商业利益受到损害。许多学校为了遵守该规定,就在课程中略去 UNIX 的源码分析而只讲操作系统原理。不幸的是,只讲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 没有借用 AT&T 一行代码,所以,不受其许可证的限制,学生可以通过它来剖析一个操作系统,研究其内部如何运作,其名称源于“小 UNIX”,因为它非常简洁、短小,故称 Minix。

Minix 用 C 语言编写,着眼于可读性好,代码中加入了数千行注释。可运行在多种平台上。Minix 一直恪守“Small is Beautiful”的原则,早期 Minix 甚至没有硬盘就能运行。目前最常用的是 Minix2.0,它具有多任务处理能力,可支持三个用户同时工作,支持 TCP/IP,支持 4 GB 内存。提供 5 个编辑器、200 个实用程序。

在 Minix 发布后不久, Internet 上出现了一个面向它的 USENET 新闻组, 数以万计的用户订阅新闻。其中的许多人都想向 Minix 中加入新功能或新特性, 使之更大更实用, 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。人们终于意识到作者的立场不可动摇, 于是芬兰的一个学生 Linus Torvalds 决定编写一个类似 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

在本章中, 首先介绍了操作系统的定义和目标, 它是计算机系统中最重要的系统软件, 讨论操作系统是从三种观点开始的: 服务用户的观点、资源管理的观点和虚拟计算机的观点。从服务用户的观点来看, 操作系统是用户与计算机硬件之间的接口, 它通过扩大机器功能、改造硬件设施来提供新的能力。从而, 用户能方便、可靠、安全、高效地使用计算机。从资源管理的观点来看, 操作系统的任务是高效地管理整个计算机系统的硬软件资源, 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, 协调各程序对资源的使用冲突, 提出使用资源的统一方法和提供简单有效的使用手段, 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。从虚拟机的观点来看, 操作系统把硬件的复杂性与用户使用设施隔离开来, 通过在硬件上加上一层层软件来改造和增强计算机硬件的功能。因此, 在硬件上配置操作系统后, 为用户提供了一台比物理计算机效率更高、容易使用的虚拟计算机。

操作系统是一个大型复杂的并发系统, 并发性、共享性、随机性和虚拟性是它的重要特征。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, 使操作系统的实现复杂化, 因而, 设计操作系统时引进了许多概念和设施(如进程和线程)来妥善解决好这些问题。

其次, 本章讨论了操作系统的发展简史, 计算机的管理和监控从手工操作阶段、管理程序阶段, 直到操作系统的形成和发展, 大约经过了 20 多年的时间。其发展是由硬件基础、软件技术和应用需求推动的, 从这个发展过程, 人们可以了解操作系统的历史和进展, 以及当前使用的操作系统已经进展到了的水平。

在 60 年代, 计算机硬件取得了两方面的突破: 通道技术的引进和中断技术的发展。这就导致了操作系统由单道作业处理流进入了多道程序设计系统阶段。多道程序设计技术是将多个作业放入主存并使它们同时处于执行状态, 从宏观上看作业均开始运行但未运行结

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进 CPU 的利用率,提高内存和设备的使用效率和充分发挥系统的并行性。这一阶段,操作系统沿着三条主线发展:多道批处理系统、分时交互系统和实时事务系统。多道批处理系统着眼于让处理器和外围设备同时保持忙碌,提高作业的吞吐率和整个系统的效率。其关键机制是:在响应一个作业的处理结束信号时,处理器将在主存中驻留的不同作业间切换。分时交互系统的主要设计目标是为用户提供方便的程序开发和调试环境和快速响应交互式用户的命令请求,但又要支持许多用户同时工作,以降低系统的成本。由于用户键盘操作速度较慢,快速响应用户请求和同时支持多用户这两个目标是可以共存的。其关键机制是:采用时间片轮转法,让处理器在多个交互式用户间多路复用。实时事务系统与分时系统相比常局限于一个或几个应用,例如,数据库的查询和修改应用或生产过程控制实时应用,但同样有响应时间的要求,甚至某些实时应用有更加严格的时间限制。其关键机制是:事件驱动机制,当系统接受来自外部的事件后,快速分析这些事件,驱动实时任务在规定的响应时间完成相应处理和控制。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,使得这时的操作系统功能变得丰富和完整。

操作系统的基本类型有三种:批处理系统、分时系统和实时系统。凡具备全部或兼有两者功能的系统称通用操作系统。随着硬件技术的发展和应用深入的需要,新发展和形成的操作系统有: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。

本章还讨论了操作系统提供的基本服务和用户接口,操作系统要为用户程序的执行提供一个良好的运行环境,要为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成,同时介绍了一般操作系统系统调用的分类和实现要点。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联系。系统程序是为用户程序的开发、调试、执行和维护解决带有共性问题或执行公共操作而由操作系统通过外部命令形式提供给用户的,它成为操作系统必须提供的功能。一般来说,一个操作系统提供的系统调用和系统程序越多,它的功能就越强,但系统也就越复杂。

人们也可以从结构的观点观察和讨论操作系统。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,首先必须研究它的结构,从两方面来讨论操作系统的结构,一是剖析和研究它的重要构件,包括:内核、进程、线程和管程等,这些也是操作系统涉及到的重要基本概念,将在本书后面章节作详细讨论。二是考察了构造操作系统的不同结构的设计方法:整体式结构、层次式结构、虚拟机构架,客户/服务器及微内核结构。客户/服务器及微内核结构、多线程机制和对称多处理器(SMP)系统是当

代操作系统的三大特点。一个微内核操作系统包含一个非常小的在内核模式下运行的内核,该内核仅含最基本最关键的操作系统功能,其他系统功能都被实现在用户模式下作为服务器运行,且使用最基本的内核服务,客户/服务器及微内核结构设计产生了灵活性高、模块化好的操作系统实现方法,但这种结构的性能问题还有待改进。此外,还对操作系统的执行模型进行了论述,非进程内核模型、在用户进程内执行的模型和作为独立进程执行的模型。以 Windows 2000/XP 客户/服务器结构为例结束了对操作系统结构设计的讨论。

最后,对当代流行的诸多操作系统作了粗略介绍,包括:DOS、Windows、UNIX、Solaris、Linux、IBM 系列操作系统。其他流行操作系统,介绍了:Mach、Mac OS、Netware、Minix 等。

本章对操作系统作了概要性介绍,读者在学习时,特别要注意它在计算机系统中的特殊地位及作用,掌握它与硬件和其他各类软件之间的关系,进而了解操作系统控制和管理整个计算机系统工作的全过程。学习单一的概念和具体的算法是比较容易的,但掌握具有操作系统的整体概念和掌握其运作过程是颇为困难的。

习 题 一

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? 计算机系统中配置操作系统的主要目标是什么?
4. 操作系统怎样实现计算操作过程的自动化?
5. 操作系统要为用户提供哪些基本和共性的服务?
6. 试叙述操作系统为提供的各种用户接口。
7. 什么是系统调用? 可分哪些类型?
8. 什么是系统程序? 可分哪些类型?
9. 试叙述系统调用的实现原理。
10. 试叙述系统调用与过程调用的主要区别。
11. 试叙述系统调用与库函数的关系。
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? 采用多道程序设计技术有什么特点?
16. 简述实现多道程序设计必须解决的基本问题。
17. 计算机系统采用通道部件后,已能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? 叙述实时操作系统的分类。
19. 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. 试叙述网络操作系统的主要功能。
24. 试叙述分布式操作系统的主要功能。
25. 试叙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? 简单叙述之。
27. 试叙述现代操作系统的基本特性及其要解决的主要问题。
28. 为什么操作系统会具有随机性特性。
29. 什么是虚拟性? 操作系统用什么方法实现虚拟性?
30. 组成操作系统的构件有哪些? 简单叙述之。
31. 什么是操作系统的内核? 列举内核的功能、属性和特点。
32. 解释微内核操作系统及其优缺点。
33. 解释微单核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. 什么是虚拟机结构操作系统? 说明其优缺点。
36. 什么是客户机/服务器及微内核结构操作系统? 说明其优缺点。
37. 试从执行方式来看, 叙述操作系统的各种实现模型。
38. 分析下列操作系统使用了或具有哪些体系结构特点: MS – DOS、UNIX、Windows 2000/XP、VM/370、Mach。
39. 叙述 Windows 2000/XP 操作系统的结构特点。
40. 叙述 Windows 2000/XP 操作系统的主要组件及其功能。
41. Windows 2000/XP 的设备驱动程序有哪些类型, 各自的主要功能是什么?
42. 一个优秀的操作系统应达到哪些主要设计目标? 试分析 Windows 2000/XP 达到了哪些设计目标?
43. 一个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. 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。
46. 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。
47. 试论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 内存, 操作系统占用 200 KB, 每个用户进程各占 200 KB。如果用户进程等待 I/O 的时间为 80%, 若增加 1 MB 内存, 则 CPU 的利用率提高多少?
2. 一个计算机系统, 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 先开始做, 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 有无空闲等待?若有, 在哪段时间内等待? 为什么会等待? (2) 程序 A、B 有无等待 CPU 的情况? 若有, 指出发生等待的时刻。

3. 设有三道程序, 按 A、B、C 优先次序运行, 其内部计算和 I/O 操作时间由图给出。

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

试画出按多道运行的时间关系图(忽略调度执行时间)。完成三道程序共花多少时间? 比单道运行节省了多少时间? 若处理器调度程序每次进行程序转换花时 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(I1, I2)设备的多道程序设计环境下, 同时投入三个作业运行。它们的执行轨迹如下:

Job1: I2(30 ms)、CPU(10 ms)、I1(30 ms)、CPU(10 ms)、I2(20 ms)

Job2: I1(20 ms)、CPU(20 ms)、I2(40 ms)

Job3: CPU(30 ms)、I1(20 ms)、CPU(10 ms)、I1(10 ms)

如果 CPU、I1 和 I2 都能并行工作, 优先级从高到低为 Job1、Job2 和 Job3, 优先级高的作业可以抢占优先级低的作业的 CPU, 但不抢占 I1 和 I2。试求:(1) 每个作业从投入到完成分别所需的时间。(2) 从投入到完成 CPU 的利用率。(3) I/O 设备利用率。

5. 在单 CPU 和两台 I/O(I1, I2)设备的多道程序设计环境下, 同时投入三个作业运行。它们的执行轨迹如下:

Job1: I2(30 ms)、CPU(10 ms)、I1(30 ms)、CPU(10 ms)

Job2: I1(20 ms)、CPU(20 ms)、I2(40 ms)

Job3: CPU(30 ms)、I1(20 ms)

如果 CPU、I1 和 I2 都能并行工作, 优先级从高到低为 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) 每个作业从投入到完成分别所需的时间。(2) 每个作业投入到完成 CPU 的利用率。(3) I/O 设备利用率。

6. 若内存中有 3 道程序 A、B、C, 它们按 A、B、C 优先次序运行。各程序的计算轨迹为:

A: 计算(20)、I/O(30)、计算(10)

B: 计算(40)、I/O(20)、计算(10)

C: 计算(10)、I/O(30)、计算(20)

如果三道程序都使用相同设备进行 I/O(即程序用串行方式使用设备, 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。两种情况下, CPU 的平均利用率各为多少?

7. 若内存中有 3 道程序 A、B、C, 优先级从高到低为 A、B 和 C, 它们单独运行时的 CPU 和 I/O 占用时间

为：

| | | | | | | | | |
|-------|------|------|------|------|------|------|------|------|
| 程序 A: | 60 | 20 | 30 | 10 | 40 | 20 | 20 | (ms) |
| | I/O2 | CPU | I/O1 | CPU | I/O1 | CPU | I/O1 | |
| 程序 B: | 30 | 40 | 70 | 30 | 30 | (ms) | | |
| | I/O1 | CPU | I/O2 | CPU | I/O2 | | | |
| 程序 C: | 40 | 60 | 30 | 70 | (ms) | | | |
| | CPU | I/O1 | CPU | I/O2 | | | | |

如果三道程序同时并发执行，调度开销忽略不计，但优先级高的程序可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，并问最早与最迟结束的程序是哪个？每道程序执行到结束分别用了多少时间？计算三个程序全部运算结束时的 CPU 利用率？

8. 有两个程序，A 程序按顺序使用：(CPU) 10 s、(设备甲) 5 s、(CPU) 5 s、(设备乙) 10 s、(CPU) 10 s。B 程序按顺序使用：(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 A，再执行 B，求出总的 CPU 利用率为多少？

9. 在某计算机系统中，时钟中断处理程序每次执行的时间为 2 ms（包括进程切换开销）。若时钟中断频率为 60 Hz，试问 CPU 用于时钟中断处理的时间比率为多少？

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,它负责管理、调度和分派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中最核心的组成部分,任何程序的执行都必须真正占有处理器,因此,处理器管理直接影响系统的性能。

操作系统的基本任务是对“进程”实施管理,操作系统必须有效控制进程的执行、给进程分配资源、允许进程之间共享和交换信息、保护每个进程在运行期间免受其他进程干扰、控制进程的互斥、同步和通信。为达到这些要求,操作系统的处理器管理必须为每一个进程维护一个数据结构,用以描述该进程的状态和分配到的资源,并允许操作系统行使对进程的控制权。

进程可以被调度在一个处理器上交替执行,或在多个处理器上同时执行。不同类型的操作系统可能采用不同的调度策略。交替执行和同时执行都是并发的类型,为了提高并发的力度和降低并发的开销,现代操作系统又进一步引进了线程的概念。

本章在简要介绍处理器的硬件运行环境之后,着重讨论了计算机系统的中断管理;然后详细介绍进程和线程的基本概念及其实现;最后又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 单处理器系统和多处理器系统

无论是在操作系统控制下执行的应用程序,还是操作系统自己,都最终要在处理器上执行,以便实现其功能。计算机系统的核心是中央处理器。如果一个计算机系统只包括一个运算处理器,称之为单处理器系统。如果有多个运算处理器,则称之为多处理器系统。

早期的计算机系统是基于单个处理器的顺序处理机器,程序员编写串行执行的代码,让其在处理器上串行执行,每一条指令的执行也是串行的(取指令、取操作数、执行操作、存储结果)。

为提高计算机处理的速度,首先发展起来的是联想存储器系统和流水线系统,前者提出了数据驱动的思想,后者解决了指令并行执行的问题,这两者都是最初计算机并行化发展的例子。随着硬件技术的进步,并行处理技术得到了迅猛的发展,计算机系统不再局限于单处

理器和单数据流,各种各样的并行结构得到了应用。目前计算机系统可以分作以下四类:

- 单指令流单数据流(SISD)。一个处理器在一个存储器中的数据上执行单条指令流。
- 单指令流多数据流(SIMD)。单条指令流控制多个处理单元同时执行,每个处理单元包括处理器和相关的数据存储,一条指令事实上控制了不同的处理器对不同的数据进行了操作。向量机和阵列机是这类计算机系统的代表。
- 多指令流单数据流(MISD)。一个数据流被传送给一组处理器,通过这一组处理器上的不同指令操作最终得到处理结果。该类计算机系统的研究尚在实验室阶段。
- 多指令流多数据流(MIMD)。多个处理器对各自不同的数据集同时执行不同的指令流。可以把 MIMD 系统划分为共享内存的紧密耦合 MIMD 系统和内存分布的松散耦合 MIMD 系统两大类。

根据处理器分配策略,紧密耦合 MIMD 系统可以分为主从式系统 MSP (Main/Slave Multiprocessor) 和对称式系统 SMP(Symmetric Multi-Processor) 两类。

主从式系统的基本思想是:在一个特别的处理器上运行操作系统内核,其他处理器上则运行用户程序和操作系统例行程序,内核负责分配和调度各个处理器,并向其他程序提供各种服务(如输入输出)。这种方式实现简单,但是主处理器的崩溃会导致整个系统的崩溃,并且极可能在主处理器形成性能瓶颈。

在对称式多处理器系统中有两个或两个以上的处理器,操作系统内核可以运行在任意一个处理器上。每个处理器都可以自我调度运行的进程和线程,单个进程的多个线程可在不同处理器上同时运行,服务器进程可以使用多个线程去处理同时来自多个客户的请求,并且操作系统内核也被设计成多进程或多线程,内核的各个部分可以并行执行。对称多处理器是迄今为止开发出的最为成功的两种并行机之一,有一种 SMP 机最多可支持 64 个处理器,多个处理器之间采用共享主存储器。SMP 机有对称性、单一地址空间、低通信延迟和一致的高速缓存等特点,具有高可靠性、可扩充性、易伸缩性。这一系统中任何处理器都可以访问任何存储单元及 I/O 设备;处理器之间通信代价很低,而并行度较高。由于共享存储器中只要保存一个操作系统和数据库副本,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。几乎所有的大型计算机硬件制造商,如 Dec、HP、IBM、SUN、SGI 都生产 SMP 机,主要用于联机分析处理、数据库和数据仓库等应用。

在松散耦合 MIMD 系统中,每个处理单元都有一个独立的内存储器,各个处理单元之间通过设定的线路或网络通信,多计算机系统和集群(Cluster)系统都是松散耦合 MIMD 系统的例子。

集群系统是迄今为止开发出的另一种最为成功的并行机,它是一组互联的计算机系统。因此也是分布式系统的一种,集群操作系统也是分布式操作系统的一个品种。集群系统运行时构成统一的计算资源,给人以一台机器的感觉。集群系统的配置一般有两种方法,一是

各个节点计算机自带磁盘,二是多个节点计算机共享 RAID 磁盘。在集群系统中,每一台计算机都是一个完整的节点,离开集群后自己可以独立地工作,所以一个节点的失效并不意味着服务的失败,从而使集群系统具备很好的容错性。集群系统还具有很好的可伸缩性,可以用低成本的微机和以太网设备等产品构成。用户开始可以在一个适度大小的系统上工作,随着需求的增加,集群系统可以被方便的扩展,直到具有数十或数百个节点。集群系统的性能价格比也很好,使用计算机集群,能够用比较低的价格,获得与一台大型计算机相同或更大的计算能力。今天,大多数通用操作系统既支持简单的单处理器计算机系统,也支持 SMP 系统和集群系统。

2.1.2 寄存器

计算机系统的处理器包括一组寄存器,其个数根据机型(处理器型号)的不同而不同,它们构成了一级存储,虽然比主存储器容量要小的多,但是访问速度要快的多。这组寄存器所存储的信息与程序的执行有很大的关系,构成了处理器现场。

不同类型的处理器具有不同的寄存器组成。一般来说,这些寄存器可以分为以下几类:

- 通用寄存器。可由程序设计者指定许多功能,如存放操作数或用作寻址寄存器。
- 数据寄存器。用以存放操作数。它们作为内存数据的高速缓存,可以被系统程序和用户程序直接使用并进行计算。
- 地址寄存器。用于指明内存地址,如索引寄存器、段寄存器(基址/限长)、堆栈指针寄存器等。
- I/O 地址寄存器。用于指定 I/O 设备。
- I/O 缓冲寄存器。用于处理器和 I/O 设备交换数据。
- 控制寄存器。用于存放处理器的控制和状态信息,它至少应该包括程序计数器 PC (Program Counter) 和指令寄存器 IR (Instruction Register), 中断寄存器以及用于存储器和 I/O 模块控制的寄存器。此外还有存放将被访问的存储单元地址的存储器地址寄存器,以及存放从存储器读出或欲写入的数据的存储器数据寄存器。

2.1.3 特权指令与非特权指令

计算机的基本功能是执行程序,而最终被执行的程序是存储在内存中的机器指令。处理器根据程序计数器(PC)从内存中取一条指令到指令寄存器(IR)并执行它,PC 将自动地增长或改变为转移地址以指明下一条要执行的指令的入口地址。

每台计算机的机器指令的集合称指令系统,它反映了一台机器的功能和处理能力,可以分为以下五类:(1)数据处理类指令:用于执行算术和逻辑运算。(2)转移类指令:如无条件

转移、条件转移、计数转移等用于改变指令执行序列。(3)数据传送类指令:用于在处理器的寄存器和寄存器、寄存器和存储器单元、存储器单元和存储器之间交换数据。(4)移位与字符串指令,移位分算术、逻辑和循环移位。字符串处理有字符串的传送、比较、查询和转换。(5)I/O 类指令:用于启动外围设备,让主存和外围设备之间交换数据。

引入操作系统后,操作系统核心程序可以使用全部机器指令,但用户程序只能使用机器指令系统的一个子集。这是因为,用户程序执行一些有关资源管理的机器指令时很容易导致系统混乱,造成系统或用户信息的破坏。如,置程序状态字指令将导致处理器占有程序的变更,它只能被操作系统使用;同样启动外围设备进行输入/输出的指令也只能在操作系统程序中执行,否则会出现多个用户程序竞争使用外围设备而导致 I/O 混乱。

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作两类:特权指令(Privileged Instructions)和非特权指令。所谓特权指令是指那些只能提供给操作系统的核心程序使用的指令,如启动输入/输出设备、设置时钟、控制中断屏蔽位、清内存、建立存储键,加载 PSW 等。只有操作系统才能执行全部指令(特权指令和非特权指令),如果一般用户执行特权指令,会导致非法执行而产生保护中断,转交给操作系统的“用户非法执行特权指令”的特殊处理程序处理。

2.1.4 处理器状态

那么,中央处理器怎么知道当前是操作系统还是一般用户在其上运行呢?这将依赖于处理器状态的标志。在执行不同程序时,根据执行程序对资源和机器指令的使用权限把处理器设置成不同状态。

处理器状态又称为处理器的运行模式,有些系统把处理器状态划分为核心状态、管理状态和用户状态,而大多数系统把处理器状态简单划分为管理状态(又称特权状态、系统模式、特态或管态)和用户状态(又称目标状态、用户模式、常态或目态)。

当处理器处于管理状态时,程序可以执行全部指令,访问所有资源,并具有改变处理器状态的能力;当处理器处于用户状态时,程序只能执行非特权指令。

Intel Pentium 的处理器状态有四种,支持 4 个特权级别,0 级权限最高,3 级权限最低。一种典型的应用是把 4 个特权级别依次设定为:

- 0 级为操作系统内核级。处理 I/O、存储管理和其他关键操作。
- 1 级为系统调用处理程序级。用户程序可以通过调用这里的进程执行系统调用,但是只有一些特定的和受保护的过程可以被调用。
- 2 级为共享库过程级。它可以被很多正在运行的程序共享,用户程序可以调用这些过程,读取它们的数据,但是不能修改它们。
- 3 级为用户程序级。它受到的保护最少。

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提供的保护级别,如运行在 Pentium 上的 Windows 操作系统只使用了 0 级和 3 级。

下面两类情况会导致从用户状态向管理状态转换,一是程序请求操作系统服务,执行一条系统调用;二是程序运行时,产生了一个中断事件,运行程序被中断,让中断处理程序工作。这两类情况都是通过中断机构才发生的,可以说中断是目态到管态转换的惟一途径。当系统中断响应交换程序状态字时,这个处理中断事件的处理程序的程序状态字的处理器状态位标志一定为“管态”。那么,怎样实现管态到目态的转换呢?每台计算机通常会提供一条特权指令称作加载程序状态字 LPSW (Load PSW),用来实现操作系统向用户程序的转换。

2.1.5 程序状态字寄存器

计算机如何知道当前处于何种工作状态?这时能否执行特权指令?通常操作系统都引入程序状态字 PSW (Program Status Word) 来区别不同的处理器工作状态。图 2-1 给出了 IBM360/370 系列计算机程序状态字的基本格式,包括:

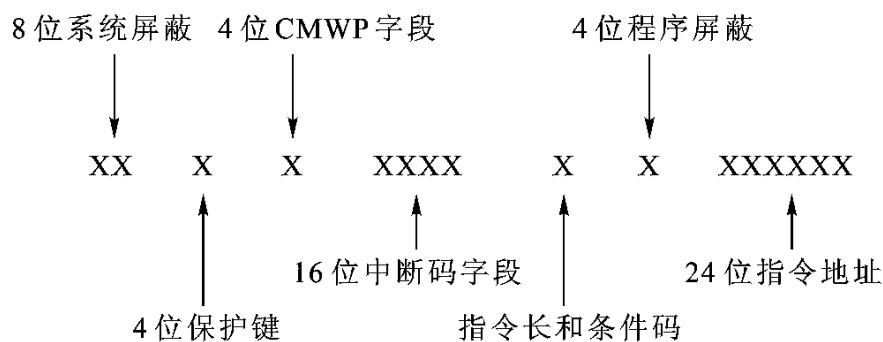


图 2-1 IBM 370 系统程序状态字的基本控制格式

- 系统屏蔽位 8 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 依次为通道 0~6 和外中断屏蔽位。
- 保护键 4 位(8~11 位) 当没有设置存储器保护时,该 4 位为 0;当设置存储器保护时,PSW 中的这四位保护键与欲访问的存储区的存储键相匹配,否则指令不能执行。
- CMWP 位(12~15 位) 依次为 PSW 基本/扩充控制方式位、开/关中断位、运行/等待位、目态/特态位。
- 中断码 16 位 中断码字段与中断事件对应,记录当前产生的中断源。
- 指令长度字段 2 位(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。
- 条件码 2 位(34,35 位)
- 程序屏蔽 4 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

位体对应的程序性事件是:定点溢出、十进溢出、阶下溢、39 位备用。

- 指令地址 24 位(40~63 位)

不难看出,程序状态字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,其主要作用是方便地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执行相关的 PSW,而每个处理器都设置一个程序状态字寄存器。一个程序占有处理器执行,它的 PSW 将占有程序状态字寄存器。一般来说,程序状态字寄存器包括以下几类内容:

- 程序基本状态。包括:(1)程序计数器:指明下一条执行的指令地址;(2)条件码:表示指令执行的结果状态;(3)处理器状态位:指明当前的处理器状态,如目态或管态、运行或等待。

- 中断码。保存程序执行时当前发生的中断事件。

- 中断屏蔽位。指明程序执行中发生中断事件时,是否响应出现的中断事件。

由于不同处理器中的控制寄存器组织方式不同,所以在大多数计算机的处理器现场中可能找不到一个称为程序状态字寄存器的具体寄存器,但总是有一组控制与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,在本书中继续使用程序状态字和程序状态字寄存器这两个概念。

在 Intel Pentium 中,程序状态字由标志寄存器 EFLAGS 和指令指针寄存器 EIP 组成,均为 32 位。EFLAGS 的低 16 位称 FLAGS,可当作一个单元来处理。标志可划分为三组:状态标志、控制标志、系统标志。

- 状态标志:它使得一条指令的执行结果影响后面的指令。算术运算指令使用 OF(溢出标志)、SF(符号标志)、ZF(结果为零标志)、AF(辅助进位标志)、CF(进位标志)、PF(奇偶校验标志);串扫描、串比较、循环指令使用 ZF 通知其操作结束。

- 控制标志:有以下几位,DF(方向标志)控制串指令操作,设定 DF 为 1,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 86 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(陷阱标志)为 1 时,允许响应中断,否则关中断。

- 系统标志:与进程管理有关,共三个:IOPL(I/O 特权级标志)、NT(嵌套任务标志)和 RF(恢复标志),被用于保护模式。指令指针寄存器 EIP 的低 16 位称为 IP(保护模式使用 32 位),存放下一条顺序执行的指令相对于当前代码段开始地址的一个偏移地址,IP 可当作一个单元使用,这在某些情况下是很有用的。

2.2 中断技术

2.2.1 中断的概念

现代计算机中都配置了硬件中断装置,中断机制是操作系统的重要组成部分之一。每当用户程序执行系统调用以求获得系统的服务和帮助、或操作系统管理 I/O 设备和处理形形色色的内部和外部事件时,都需要通过中断机制进行处理。所以,也有人说操作系统是由“中断驱动”的。最初,中断技术作为用来向 CPU 报告某设备已完成操作的一种手段,现在中断技术的应用越来越广泛。例如,在计算机运行过程中,有许多事件会随机发生,如硬件故障、电源断电、人机联系和程序出错等,这些事件必须及时加以处理。在生产自动控制这类实时系统中,通过发中断信号以便即时将传感器传来的温度、距离、压力、湿度等变化信息送给计算机,计算机则暂停当前工作,转去处理和解决异常情况。所以,为了请求操作系统服务,实现并行工作,处理突发事件,满足实时要求,都需要打断处理器正常的工作,为此,中断概念被提出来了。中断(interrupt)是指程序执行过程中,当发生某个事件时,中止 CPU 上现行程序的运行,引出处理该事件的服务程序执行的过程。现代计算机系统一般都具有处理突发事件的能力。例如,从磁带上读入一组信息,当发现读入的信息有错误时,会产生一个读数据错中断,操作系统暂停当前的工作,并组织让磁带退回重读该组信息就可能克服错误,而得到正确的信息。在提供中断装置的计算机系统中,在每两条指令或某些特殊指令执行期间都检查是否有中断事件发生,若无则立即执行下一条或继续执行,否则响应该事件并转去处理中断事件。

这种处理突发事件的能力是由硬件和软件协作完成的。首先,由硬件的中断装置发现产生的中断事件,然后,中断装置中止现行程序的执行,引出处理该事件的程序来处理。计算机系统不仅可以处理由于硬件或软件错误而产生的事件,而且可以处理某种预见要发生的事件。例如,外围设备工作结束时,也发出中断请求,向系统报告它已完成任务,系统根据具体情况做出相应处理。引起中断的事件称为中断源。发现中断源并产生中断的硬件称中断装置。在不同的硬件结构中,通常有不同的中断源和不同的中断装置,但它们有一个共性,即当中断事件发生后,中断装置能改变处理器内操作执行的顺序,可见中断是现代操作系统实现并发性的基础之一。

2.2.2 中断源分类

引起中断的事件称为中断源。不同硬件结构的中断源各不相同,从中断事件的性质和激活的手段来说,可以分成强迫性中断事件和自愿性中断事件两大类。

强迫性中断事件不是正在运行的程序所期待的,而是由于随机发生的某种事故或外部请求信息所引起的。这类中断事件大致有以下几种:

- 机器故障中断事件。例如电源故障、主存储器出错等。

• 程序性中断事件。例如定点溢出、除数为 0、地址越界等。由于这类中断反映程序执行中发现的例外情况,所以又称异常。

- 外部中断事件。例如时钟的定时中断、控制台发控制信息等。

- 输入输出中断事件。例如设备出错、传输结束等。

自愿性中断事件是正在运行的程序所期待的事件。这种事件是由于执行了一条访管指令而引起的,它表示正在运行的程序对操作系统有某种需求,一旦机器执行到一条访管指令时,便自愿停止现行程序的执行而转入访管中断处理程序处理。例如,要求操作系统协助启动外围设备工作。

两类中断事件的响应过程略有不同,详见图 2-2。

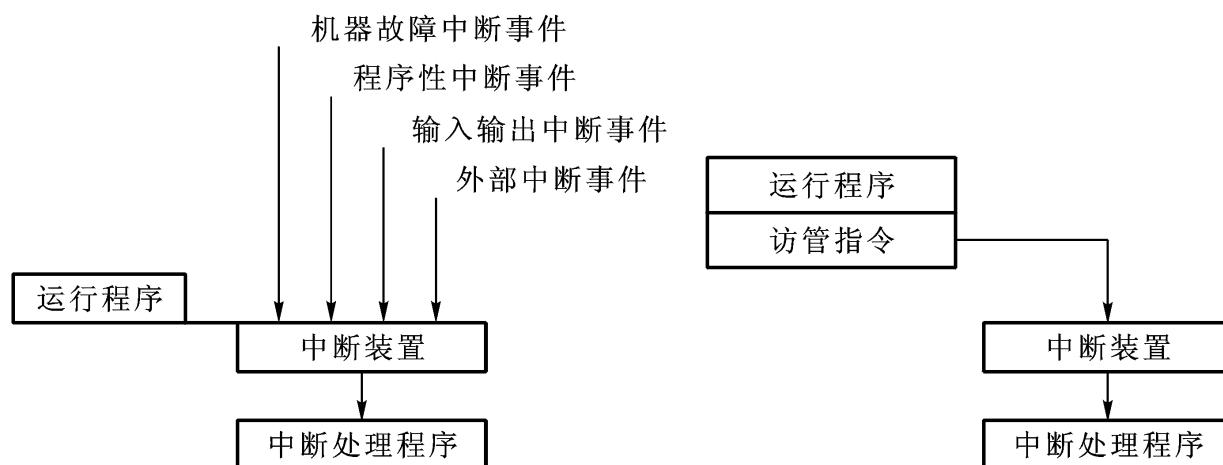


图 2-2 两类中断事件

还可以按照中断信号的来源,把中断分为外中断和内中断两类:

• 外中断。一般又称中断,是指来自处理器和主存储器之外的中断,包括电源故障中断、时钟中断、控制台中断、它机中断和 I/O 中断等。每个不同的中断具有不同的中断优先级,在处理高一级中断时,往往屏蔽部分或全部低级中断。

• 内中断。是指来自处理器和主存内部的中断,一般又称异常(exception),包括通路校验错、主存奇偶错、非法操作码、地址越界、页面失效、调试指令、访管中断、算术操作溢出等。

各种程序性中断。其中访管中断是由机器指令提供的特殊指令,该指令执行时将会引起内中断。异常是不能被屏蔽的,一旦出现应立即响应并加以处理。

中断和异常的区别如下:中断是由与现行指令无关的中断信号触发的,所以它是异步的,而且中断的发生与 CPU 处在用户模式或内核模式无关,通常在两条机器指令之间才可以响应中断,一般来说,中断处理程序提供的服务不是为当前进程所需要的,如时钟中断、硬盘读写服务请求中断;而异常则是由处理器正在执行现行指令而引起的。因而一条指令执行期间允许响应陷入。通常,异常处理程序提供的服务是为当前进程所用的。异常包括很多方面,有出错(fault),也有陷入(trap)。出错和陷入的主要一点区别是:它们发生时保存的返回指令地址不同,出错保存指向触发异常的那条指令,而陷入保存指向触发异常的那条指令的下一条指令。因此,当从异常返回时,出错会重新执行那条指令,而陷入就不会重新执行那条指令。如缺页异常是一种出错,而陷入主要应用在调试中。此外,中断的嵌套发生是允许的,异常多数情况为一重处理,异常处理过程中可能产生中断,但反之则不会发生。

IBM 中大型机操作系统使用了上述第一种分类方法,Windows 2000/XP 则采用了上述第二种分类方法。

上述的内中断与外中断(中断和异常)要通过硬件设施来产生中断请求,可以看作硬中断。与其相对应的不必由硬件发信号而能引发的一种中断称为软中断,软中断是利用硬件中断的概念,用软件方式进行模拟,实现宏观上的异步执行效果。它通常是由内核或进程对某个进程发出的中断信号,可以看作内核与进程或进程与进程之间用来模拟硬中断的一种信号通信方式。两者的共同点是:当中断源产生中断请求或发出软中断信号后,CPU 或者接收进程在适当的时机自动进行中断处理或完成软中断信号所对应的功能。这里所说适当时机表示接收的硬中断会及时获得中断处理程序的处理,但接收软中断信号的进程不一定正好在接到此信号时占有处理器,而相应的软中断信号处理必需等到该接收进程获得处理器后才能进行,通常会有一定时间的延迟,将在第三章中介绍软中断通信。

2.2.3 中断装置

发现中断源并产生中断的硬件称中断装置,这些硬件包括中断逻辑线路和中断寄存器。迄今为止,所有的计算机系统都采用硬件和软件(硬件中断装置和软件中断处理程序)结合的方法实现中断处理。一般来说,硬件中断装置主要做以下三件事:

- 发现中断源,响应中断请求。当发现多个中断源时,它将根据规定的优先级,先后发出中断请求。
- 保护现场。将运行程序中断点在处理器中某些寄存器内的现场信息(又称运行程序的执行上下文)存放于内存储器。使得中断处理程序运行时,不会破坏被中断程序的有用信息,以便在中断处理结束后能够返回被中断程序继续运行。

- 启动处理中断事件的中断处理程序,处理器状态已从目态被切换到管态。

中断来源于正在执行的程序以及计算机系统的各个部件,甚至计算机的外部环境。当一个具体的中断事件发生时,计算机的硬件中断装置必须把它记录下来。中断寄存器是用来记录中断事件的寄存器,中断寄存器的内容称中断字,中断字的每一位对应一个中断事件。每当一条机器指令执行结束的时刻,中断控制部件扫描中断字,查看是否有中断事件发生,若是则处理器便响应这个中断请求。

当中断发生后,中断字的相应位会被置位。由于同一时刻可能有多个中断事件发生,中断装置将根据中断屏蔽要求和中断优先级选取一个,然后把中断寄存器的内容送入程序状态字寄存器的中断码字段,且把中断寄存器的相应位清“0”。当处理中断事件的程序执行时就可以读出中断信息进行分析,从而知道发生了什么中断事件。

紧接着中断装置进行必要的保护现场工作。此时并不一定要将处理器中所有寄存器中的信息全部存于(写回)存储器中,但是,对程序状态字寄存器中的那些信息一定要保护起来。最后,将中断处理程序的程序状态字送入现行程序状态字寄存器,这就引出了相应的中断事件处理程序。

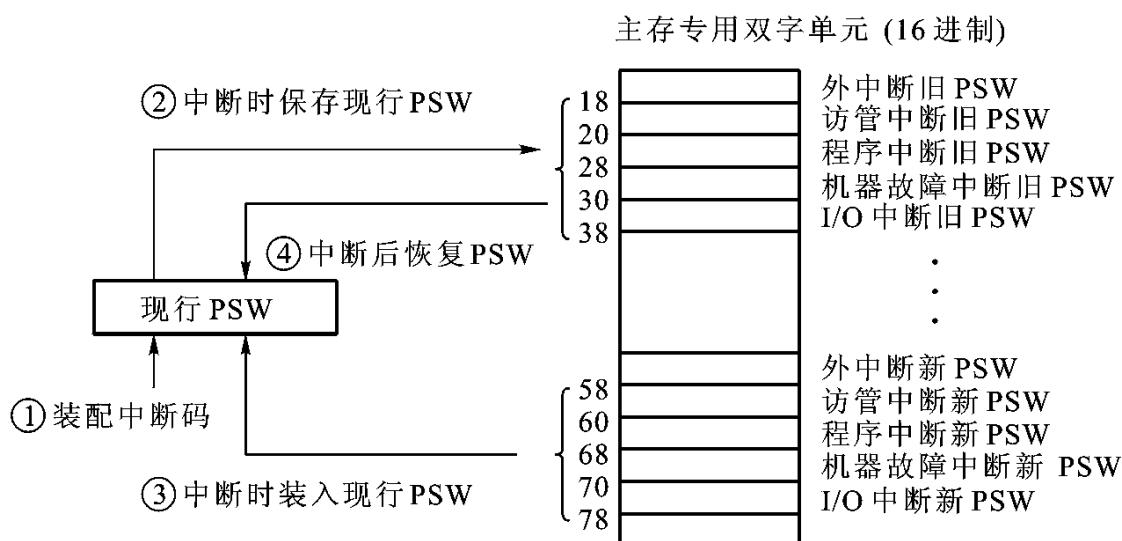


图 2-3 IBM 中大型机中断响应过程

如果把被中断的程序的程序状态字称为旧程序状态字,而把中断处理程序的程序状态字称为新程序状态字的话,如何来实现新旧程序状态字的交换呢?通常,系统为每一种中断都开辟了主存的固定单元存放新的和旧的程序状态字。图 2-4 是 IBM 中大型机中断响应过程,主存中开辟了专用的双字单元(用 16 进制标出),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 中还有硬件程序状态字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把中断码装配到现行 PSW 中,然后,执行②把现行 PSW 保存到中断类相应的旧 PSW 单元;同时,执

行③把中断类相应的新 PSW 加载到现行 PSW, 这就引出了相应中断类的中断处理程序。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。

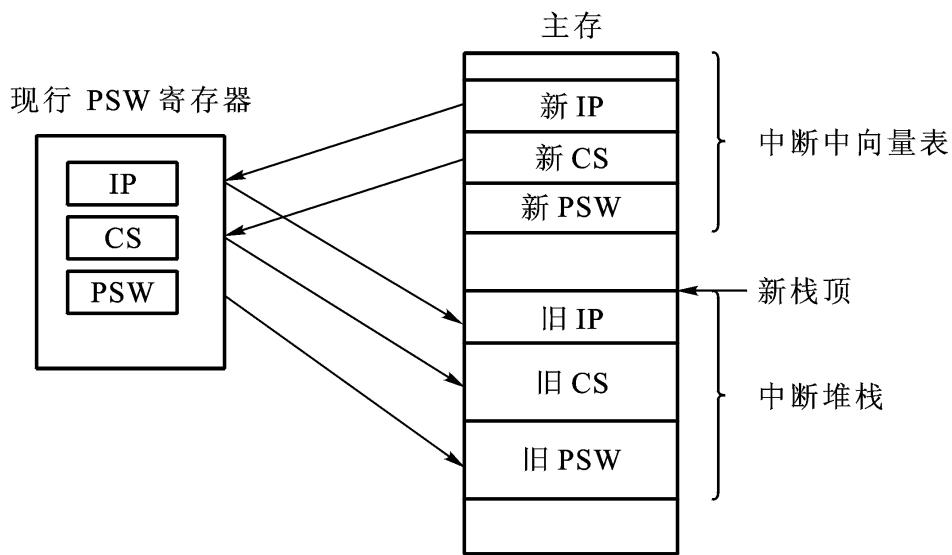


图 2-4 IBM PC 机中断的响应过程

在 IBM PC 机上,为了方便地找到中断处理程序,通常在计算机内存的低地址处开辟了一个称为中断向量表的区域。表中每一项称为一个中断向量,其中存放了一个中断处理程序的入口地址及相关信息,不同中断源需要用不同的中断处理程序处理,也就对应了不同的中断向量。另外,采用堆栈方式保存被中断程序状态信息,如图 2-4 所示,当发现中断源并响应中断时,中断装置将把现行 PSW 内容压进堆栈,接着再把指令指针 IP 和代码段基址内容也压进堆栈,这就保存了原运行程序的状态。处理器根据硬件中断装置提供的中断向量号,获得被接受的中断请求的中断向量地址,再按照中断向量地址把中断处理程序的 PSW 送入现行程序状态字寄存器,加载新的程序状态字。从而,就引出了处理特定中断事件的中断处理程序。返回原程序时,只要把栈顶内容弹出送入现行 IP、CS 和 PSW 中。

2.2.4 中断处理程序

处理中断事件的程序称为中断处理程序。它的主要任务是处理中断事件和恢复正常操作。由于不同中断源对应不同中断处理程序,故快速找到中断处理程序的入口地址是一个关键问题。寻找入口地址可用如下办法:在主存储器(常在低地址区)设置一张向量地址表,存储单元的地址对应向量地址,存储单元的内容为入口地址。CPU 响应中断后,根据预先规定的次序找到相应向量地址,便可获得该中断事件处理程序的入口地址。

一个操作系统设计者将根据中断的不同类型和不同的应用环境,来确定不同的处理原则。具体地讲,一个中断处理程序主要做以下四项工作:

- 保护未被硬件保护的一些必需的处理状态。例如,将通用寄存器的内容保存到主存储器,从而使中断处理程序在运行中可以使用通用寄存器。
- 识别各个中断源,分析产生中断的原因。
- 处理发生的中断事件。中断处理程序将根据不同的中断源,进行各种处理操作。有简单的操作,如置一个特征标志;也有相当复杂的操作,如重新启动磁带机倒带并执行重读操作。
- 恢复正常操作。恢复正常操作一般有几种情况:恢复中断前的程序按断点执行;重新启动一个新的程序或者甚至重新启动操作系统。

2.2.5 中断事件的具体处理方法

1. 机器故障中断事件的处理

一般来说,这种事件是由硬件的故障产生的,排除这种故障必须进行人工干预。中断处理能做的工作一般是保护现场,防止故障蔓延,报告给操作员并提供故障信息以便维修和校正,以及对程序中所造成的破坏进行估价和恢复。下面列举一些硬件失效中断事件的处理办法。

1) 电源故障的处理

当电源发生故障,例如断电时,硬设备能保证继续正常工作一段时间。操作系统利用这段时间可以做以下三项工作:

- 将处理器中有关寄存器内的信息经主存储器送到磁盘保存起来,以便在故障排除后恢复现场,继续工作。
- 停止外围设备工作。有些外围设备(例如磁带机)不能立即停止,中断处理程序将把这些正在交换信息又不能立即停止的设备记录下来。
- 停止处理器工作。一般可以让主机处于停机状态,此时,整个系统既不执行指令又不响应中断。

当故障排除后,操作员可以从一个约定点启动操作系统以恢复工作。恢复程序做的主要工作是:

- 恢复中断前的有关现场。
- 启动被停止的外围设备继续工作。
- 如果发生故障时有不能立即停止的外围设备正在工作,那么,涉及这些外围设备的程序将被停止执行而等待操作员的干预命令。

完成上述各项工作后,系统将选择可以运行的程序继续运行。

2) 主存储器故障的处理

主存储器的奇偶校验或海明校验装置发现主存储器读写错误时,就产生这种中断事件。中断处理程序首先停止与出现的中断事件有关的程序的运行。然后向操作员报告出错单元的地址和错误的性质。

2. 程序性中断事件的处理

处理程序性中断事件大体上有两种办法。对于那些纯属程序错误而又难以克服的事件,例如非法使用特权指令,企图访问一个不允许其使用的主存储器单元等,操作系统只能将出错程序的名字、出错地点和错误性质报告给操作员并请求干预。对于其他一些程序性中断,例如定点溢出、阶码下溢等,不同的用户往往有不同的处理要求。所以,操作系统可以将这种程序性中断事件转交给用户程序自行处理。如果用户程序对发生的中断事件没有提出处理办法,那么操作系统将进行标准处理。

用户怎样来编制处理中断事件的程序呢?有些语言提供了称之为 on 语句的调试语句,它的形式如下:

```
on <条件> <中断续元入口>
```

它表示当指定条件的中断发生时,由中断续元来进行处理。例如:

```
on fixed overflow go to LA;
```

每当发生定点溢出时,转向以 LA 为标号的语句。对于发生在不同地方的同一种程序性中断事件允许用户采用不同的处理方法。例如,在执行了上述调试语句后又执行调试语句:

```
on fixed overflow go to LB;
```

就表示今后再发生溢出时将转向 LB 而不是转向 LA 去处理了。

有了调试语句后,用户用程序设计语言编制程序时,也就可以编写处理程序性中断事件的程序了。编译程序为每个用户设置一张中断续元入口表,且在编译源程序产生目标程序时,把调试语句翻译成一段程序。其功能是:将中断续元入口地址送入中断续元入口表中对应该语句的中断条件的那一栏。中断续元入口表的形式如图 2-5:

| | | | |
|----------|--------|---|---|
| 中断事件 0 | 中断事件 1 | 0 | 0 |
| 中断续元入口 0 | | | |
| 中断续元入口 1 | | | |
| ... | | | |
| 中断续元入口 N | | | |

图 2-5 中断续元入口

对应每一个用户处理的中断事件,表格中有一栏用以填写处理该中断事件的中断续元

入口地址。如果用户没有给出处理其中断事件的中断续元时, 相应栏的内容为 0。当程序运行执行到调试语句时, 就将中断续元的入口地址送入相应栏内。显然, 对于同一中断事件, 当执行第二次对应该事件的调试语句时, 就将第二次规定的中断续元入口地址填入表内相应栏中而冲去了第一次填写的内容。这就是上面所说的, 利用对同一条件多次使用调试语句时, 可以做到对发生于不同地点的同一种中断事件采用不同的处理方法。

当发生程序中断事件后, 操作系统是怎样转交给用户程序去处理的呢? 操作系统只要根据中断事件查看表中对应栏, 如果对应栏为“0”它表示用户未定义该类中断续元, 此时系统将按规定的标准办法进行处理。例如, 将程序停下来, 向操作员报告出错位置和性质, 或者置之不顾, 就好像什么事也没有发生一样。如果对应栏不为“0”, 则强迫用户程序转向中断续元去处理。但是, 如果在中断续元的执行中又发生中断事件时, 就不能这样简单地处理了。首先, 中断续元的嵌套一般应规定重数, 在上面的表格中规定嵌套重数为 2。表格第一栏的第 0 字节记录了第一次进入中断续元的事件号; 第 1 个字节记录了第二次(嵌套)进入中断续元的事件号。其次, 中断续元的嵌套不能递归, 例如, 处理定点溢出的中断续元, 在执行时不允许又发生定点溢出程序性中断事件。

下面按步骤小结一下中断续元的处理过程和原则:

- 编译程序编译到 on 语句时, 生成填写相应中断续元入口表的目标代码段;
- 程序运行执行到 on 语句时, 根据中断条件号, 将中断续元入口填入相应栏, 这是通过执行上述代码段来实现的;
 - 执行同一中断条件号的 on 语句时, 中断续元入口被填入同一栏, 从而, 用户可在他的程序的不同部分对同一中断条件采用不同的处理方法;
 - 每当一个中断条件发生时, 检查中断续元入口表相应栏, 或转入中断续元处理, 或进行操作系统标准处理;
 - 程序性中断处理允许嵌套, 应预先规定嵌套重数, 但不允许递归。

3. 外部中断事件的处理

时钟定时中断以及来自控制台的信息都属外部中断事件, 它们的处理原则如下:

1) 时钟中断事件的处理

时钟是操作系统进行调度工作的重要工具, 如让分时进程作时间片轮转、让实时进程定时发出或接收控制信号、系统定时唤醒或阻塞一个进程、对用户进程进行记账。时钟可以分成绝对时钟和间隔时钟(即闹钟)两种。利用计时器能确保操作系统必要时获得控制权, 例如, 陷入死循环的进程最终因时间片耗尽会被迫出让处理器。

系统设置一个绝对时钟寄存器, 计算机的绝对时钟定时地(例如每 10 ms)把该寄存器的内容加 1。如果开始时这个寄存器的内容为 0, 那么, 只要操作员告诉系统开机时的年、月、日、时、分、秒, 以后就可推算出当前的年、月、日、时、分、秒了。当绝对时钟寄存器记满溢出

时,就产生一次绝对时钟中断,操作系统处理这个中断时,只要在主存的固定单元上加 1 就行了。这个固定单元记录了绝对时钟中断的次数,这样就可保证有足够的计时量。计算当前时间时,只要按绝对时钟中断的次数和绝对时钟寄存器的内容推算就可得到。

间隔时钟是定时将一个间隔时钟寄存器的内容减 1,当间隔时钟寄存器的内容为 0 时,就产生一个间隔时钟中断。所以,只要在间隔时钟寄存器中放一个预定的值,那么就可起到闹钟的作用,每当产生一个间隔时钟中断,就意味着预定的时间到了。操作系统经常利用间隔时钟作控制调度。

时钟硬件做的工作仅仅是按已知时间间隔产生中断,其余与时间有关的任务必须由软件来做,不同的操作系统有关时钟的任务不同,但一般包括以下内容:

- 维护绝对日期和时间;
- 防止进程的运行时间超出其允许值,发现陷入死循环的进程;
- 对使用 CPU 的用户进程记账;
- 处理进程的间隔时钟(闹钟);
- 对系统的功能或部件提供监视定时器。

在 Intel x86/pentium 微机中,Linux 利用 CMOS 中记录的时间作为系统启动时的基准时间,在系统运行时,利用时钟滴答(clock tick)来维护系统的时间。Linux 使用一个全局变量称 jiffies(瞬时)作为所有系统时间的测量基准,系统启动时,CMOS 中记录的时间转化为从 1970 年 1 月 1 日 0 时 0 分 0 秒(UNIX 纪元)算起的 jiffies 值(累积秒数)。操作系统中需要有定时服务的机制,以实现准时调度任务或处理与时间相关的工作,这些都是通过定时器机制来实现的。Linux 中存在两种类型的系统定时器,这两种定时器都具有对应的处理例程,必须在到达给定的系统时间时被进程调用,但实现方法有些不同。如图 2-6 所示。第一类是老的定时器机制,有一个 32 个指针的数组定义的定时器。每个指针可指向一个 timer-struct 结构,而 timer-active 是活动定时器掩码,数组元素是静态定义的,在系统初始化时入口被加到该数组中。第二类是新的定时器机制,突破了 32 个定时器的限制,使用一个 timer-list 数据结构的链表,按定时器到期时间的升序排列。两类定时器中 expires 给出该定时器被激活的时间,而 * fn() 指出定时器激活后的处理函数。

两类定时器都使用 jiffies 值作为到期比较时间。例如,某个定时器要在 2 s 之后到期,则必须将 2 s 转换成对应的 jiffies 值,加上当前的系统时间(也是以 jiffies 为单位)后,得到的便是该定时器到期的系统时间 expires。每次系统时钟滴答到来时,定时器 bottom half 处理过程被标记为活动状态,这样当调度程序下次运行时,定时器队列能获得处理。定时器 bottom half 处理过程要处理两种类型的系统定时器。对于老的系统定时器,检查 timer-active 中被置位的位掩码,以便确定活动的定时器。如果一个活动的定时器到期,便调用对应的定时器例程, timer-active 对应位被清除。对于新的系统定时器,检查链表中的 timer-list 数据结构。

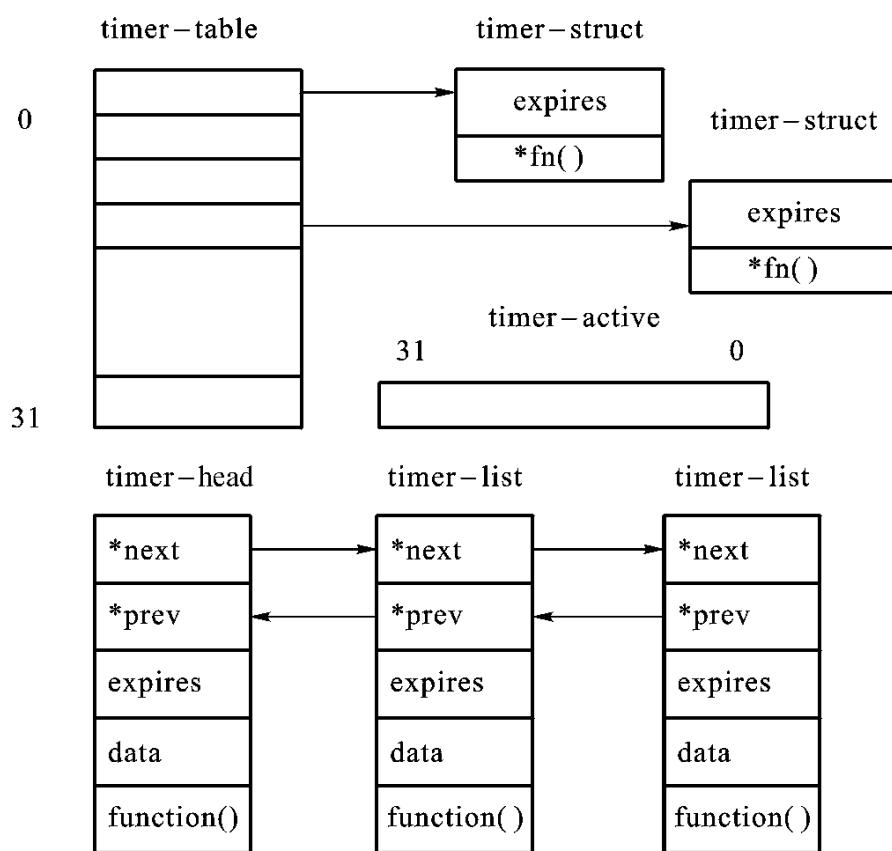


图 2-6 Linux 定时器机制

每个到期的定时器将被从链表中移出, 对应的定时器例程被调用。新的定时器机制的优点是能传递参数 `data` 到定时器例程中。

有了上述定时器, Linux 就可以统计用户的记账信息, 它记录进程的创建时间及进程在生命周期占用的 CPU 时间。每个时钟滴答到来时, 核心都修改当前进程在内核态和用户态占用的时间, 这些时间称为记账信息 (accounting time)。对于不同的时间, Linux 运行了不同的间隔定时器, 这些间隔定时器的类型有三种:

- **real** 这种间隔定时器按实际时间计时, 不管进程处在何种模式下运行 (包括进程被挂起时), 计时总在进行, 当定时到达时发送给进程一个 SIGALRM 信号。
- **virtual** 这种间隔定时器仅当进程在用户态下执行时才计时, 当定时到达时发送给进程一个 SIGVTALRM 信号。
- **profile** 这种间隔定时器是当进程执行在用户态或核心态时都计时, 当定时到达时发送给进程一个 SIGROF 信号。

Linux 允许进程同时启动多个定时器, 通过在一个进程中设定上述三个定时器, 就可以了解一个进程在用户态、内核态和总的执行时间。把定时器工作所需的时间值及有关信息保存在进程的 task-struct 中, 可以使用系统调用设置、启动、停止定时器或读出定时器的当前值。Virtual 和 profile 定时器的工作原理相同, 当前进程的定时值随时钟 tick 而递减, 直到为

0时,表示定时时刻到达,定时器就会发出相应定时信号。Real定时器的工作原理稍有不同,当使用这种定时器时,进程使用系统的timer-list数据结构。每次时钟滴答时它的定时值也会递减,但是当定时时刻到达时,是由时钟bottom half处理过程把它从定时器队列中删除,并调用间隔定时器处理程序,这一处理过程中会产生一个SIGALRM信号。

最后,给出Linux时钟系统调用。在Lniux中,通过时钟系统调用完成系统时钟的读取、设置和校准功能。这些系统调用为计时服务提供支持,也为用户查询当前系统时间提供了接口。

- sys-time:读取系统时间,精度秒级。
- sys-stime:设置系统时间,精度秒级。
- sys-gettimeofday:读取系统时间和时区,精度微秒级。
- sys-settimeofday:设置系统时间和时区,精度微秒级。
- sys-adjtimex:用于在网络环境下,当本地的系统时间和网络服务器时间不符时调整系统时钟。

2) 控制台中断事件的处理

操作员可以利用控制台开关请求操作系统工作,当使用控制台开关后,就产生一个控制台中断事件通知操作系统。操作系统处理这种中断就如同接受一条操作命令一样,转向处理操作命令的程序执行。

4. I/O 中断的处理

输入输出中断种类比较多,处理方法各异。

(1) I/O 操作正常结束后的处理

首先,把正在等待输入输出操作完成的进程设置为可执行的状态,然后要查看是否有等待该设备或通道的其他进程,若有则释放。

(2) I/O 操作发生故障后的处理

对于设备本身的故障,可以先向相应设备发命令索取状态字节。然后进行分析就可以知道故障的确切原因。如果该外围设备的控制器有复执功能,就组织复执。如果该外围设备的控制器没有复执功能,那么,对于某些故障系统可组织软复执。对于不能复执的故障或复执多次仍不能克服的故障,系统将向操作员报告,请求人工干预。

对于启动命令的错误,例如,启动外围设备的命令要求从输入机上读入1 000个字符,然而读了500个字符就遇到“停码”,输入机停止。操作系统在处理这类错误时,可以把其转交给用户,转向用户程序的中断续元,由用户自己处理。

(3) I/O 操作发生异常后的处理

如果设备在操作中发生了某些特殊事件,那么在设备操作结束发生中断时,也要将这个情况向系统报告。操作系统从设备状态字节中的设备特殊位为1,可以判知设备在操作中

发生了某个特殊事件。对于磁带机,这意味着在写入一块信息遇到了带末点或读出信息时遇到了带标。在写操作的情况,系统知道磁带即将用完,如果文件还未写完,应立即组织并写入卷尾标,然后通知操作员换卷以便将文件的剩余部分写在后继卷上。在读操作的情况,系统判知这个文件已经读完或这个文件在此卷上的部分已经读完,进行文件结束的处理;若只读了一部分,则带标后面是卷尾标,系统将通知操作员换卷,以便继续读入文件。对于行式打印机,这意味着纸用完,因此系统可暂停输出,通知操作员装纸,然后继续输出。

(4) 设备报到或设备结束的处理

如果是外围设备上来的“设备报到”或“设备结束”等异步信号,表示有外围设备接入可供使用或断开暂停使用,操作系统应修改系统表格中相应设备的状态。

5. 自愿中断事件的处理

这类中断是由于系统程序或用户程序执行访管指令(例如,UNIX中用的trap指令,MS-DOS中用的int指令,IBM370中用的supervisor指令等)而引起的,表示运行程序对操作系统功能的调用,所以也称系统调用,可以看作是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,前者表示这条指令是访管指令,后者表示具体的访管要求。硬件在执行访管指令时,把访管参数作为中断字并入程序状态字,同时将它送入主存指定单元,然后转向操作系统处理。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,按照访管参数的要求进行相应的处理。不同的访管参数对应不同的要求,就像机器指令的不同操作码对应不同的要求一样。

操作系统的基本服务是通过系统调用来请求的,是操作系统为用户程序调用其功能提供的接口和手段。系统调用机制本质上通过特殊硬指令和中断系统来实现。不同机器系统调用命令的格式和功能号的解释不尽相同,但任何机器上的系统调用都有共性处理流程。这一共性处理流程如下:

- 用户程序执行 n 号系统调用。
- 通过中断系统进入访管中断处理,保护现场,按功能号跳转。
- 通过系统调用入口表找到相应功能入口地址。
- 执行相应例行程序,结束后正常情况返回系统调用的下一条指令执行。

2.2.6 中断的优先级和多重中断

1. 中断的优先级

在计算机执行的每一瞬间,可能有几个中断事件同时发生。例如,由非法操作码引起程序性中断的同时可能发生外部中断要求。这时,中断装置如何来响应这些同时发生的中断呢?一般说,中断装置按照预定的顺序来响应。这个按中断请求的轻重缓急的程度预定的

顺序称为中断的优先级, 中断装置首先响应优先级高的中断事件。

在一个计算机系统中, 各中断源的优先顺序是根据某个中断源或中断级若得不到及时响应, 造成计算机出错的严重性程度来定的。例如, 机器校验中断表明产生了一个硬件故障, 对计算机及当前执行任务的影响最大, 因而优先级排在最高; 至于 I/O 中断事件其优先级可放低, 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, 对系统工作的正确性影响不大。当某一时刻有多个中断源或中断级提出中断请求时, 中断系统如何按预先规定的优先顺序响应呢? 可以使用硬件和软件两种办法。前者根据排定的优先次序做一个硬件链式排队器, 当有高一级的中断事件产生时, 应该封住比它优先级低的所有中断源; 后者编写一个查询程序, 依据优先级次序自高到低进行查询, 一旦发现有一个中断请求, 便转入该中断事件处理程序入口。

一种可能的中断优先级由高到低的顺序是: 机器校验中断、自愿性中断、程序性中断、外部中断、输入输出中断、重启动中断。读者注意, 中断的优先级只是表示中断装置响应中断的次序, 而并不表示处理它的先后顺序。

现代大部分计算机配置可编程中断控制器, CPU 可执行指令设置可编程中断控制器的屏蔽码, 当发现屏蔽码置位的中断位有了中断也不向 CPU 申请中断。这时硬件自动保存这次中断, 以便屏蔽解除再行申请中断并进行处理。

2. 中断的屏蔽

主机可以允许或禁止某类中断的响应, 如主机可以允许或禁止所有的输入输出中断、外部中断、机器校验中断以及某些程序性中断。对于被禁止的中断, 有些以后可继续响应, 有些将被丢弃。例如, 对于被禁止的输入输出中断的条件将被保留以便以后响应和处理, 对于被禁止的程序中断条件, 除了少数置特征码以外, 都将丢弃不管。有些中断是不能被禁止的, 例如计算机中的电源断电中断、自愿性访管中断就不能被禁止。

主机是否允许某类中断, 由当前程序状态字中的某些中断屏蔽位来决定。一般, 当屏蔽位为 1 时, 主机允许相应的中断, 当屏蔽位为 0 时, 相应中断被禁止。按照屏蔽位的标志, 可能禁止某一类内的全部中断, 也可能有选择地禁止某一类内的部分中断。有了中断屏蔽功能, 就增加了中断排队的灵活性, 采用程序的方法在某段时间中屏蔽一些中断请求, 以改变中断响应的顺序。

3. 多重中断事件的处理

在一个计算机系统运行过程中, 由于中断可能同时出现, 或者虽不同时出现但却被硬件同时发现, 或者出现在其他中断正在进行处理期间, 这时 CPU 又响应了这个新的中断事件, 于是暂时停止正在运行的中断处理程序, 转去执行新的中断处理程序, 这就叫多重中断(又称中断嵌套)。一般来说, 优先级别高的中断有打断优先级别低的中断处理程序的权利, 但

反之则不允许优先级别低的中断干扰优先级别高的中断处理程序的运行。系统如何来处理出现的多个中断呢？这是操作系统的中断处理程序所必须解决的问题。

对于多个中断，可能是同一中断类型的不同中断源，也可能是不同类型的中断。对于前者，一般由同一个中断处理程序按预定的次序分别处理之；对于后者，可以区别不同情况做如下处理：

- 禁止再发生中断。在运行一个中断处理程序时，对任何新产生的中断不予理睬，这可以通过屏蔽某些中断来实现。例如，在运行处理 I/O 中断的例行程序时，可以屏蔽外部中断或其他 I/O 中断，甚至所有中断。这种方法简单易行，所有中断都严格按顺序处理，但没有考虑相对优先级和时间限制的要求。

- 定义中断优先级。对于有些必须处理且优先级更高的中断源，采用屏蔽方法有时可能是不妥的，因此在中断系统中往往允许在运行某些中断例行程序时，仍然可以响应中断。这时，系统应负责保护被中断的中断处理例行程序的现场（有的计算机中断系统对断点的保存是在中断周期内，由中断隐指令实现，对用户是透明的），然后再转向处理新中断的例行程序，以便处理结束时可以返回原来的中断处理例行程序继续运行。操作系统必须预先做出规定，哪些中断类型允许嵌套？嵌套的最大级数？嵌套的级数视系统规模而定，一般不超过三重为宜，因为过多重的“嵌套”将会增加不必要的系统开销。

- 响应并进行中断处理。在运行中断处理例行程序时，如果出现任何程序性中断源，一般情况下，表明这时中断处理程序有错误，应立即响应并进行处理。

每个中断处理程序的程序状态字中，究竟应该屏蔽哪些中断源，将由系统设计而定，需要考虑的情况有：硬件的中断优先级，应用的需要，软件处理所希望的优先级，可能丢失的中断源及其对系统的影响等。

2.2.7 实例研究：Windows 2000/XP 中断处理

1. Windows 2000/XP 中断处理概述

在 Pentium 芯片上 Windows 中断处理的实现中，使用了陷阱、中断和异常等术语。

中断和异常是把处理器转向正常控制流之外的代码的操作系统情况，硬件和软件都可以产生异常和中断，这两个术语相当于本节前面所讨论的“中断概念”。内核按照以下的方法来区分中断和异常。

中断是异步事件，可能随时发生，与处理器正在执行的内容无关。中断主要由 I/O 设备、处理器时钟或定时器产生、可以启用或禁用。异常是同步事件，它是某一个特定指令执行的结果。异常的一些例子是内存访问错误、调试指令及被零除。内核也将系统服务调用视作异常，在技术上也可以把它作为系统陷阱。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,而被零除异常是由软件引起的。同样,I/O设备可产生中断,而内核自身也可以发出中断。

陷阱是指处理意外事件的一种硬件机制,当中断或异常发生时,它能发现并俘获正在执行的线程,把它从用户态切换到核心态,并将控制权交给内核的陷阱处理程序。不难看出,陷阱机制相当于本节前面所讨论的中断响应和中断处理机构。

图2-7说明了激活陷阱处理程序的情况和由陷阱处理程序调用来为它们服务的例程。陷阱处理程序被调用时,它在保存机器状态(机器状态将在另一个中断或异常发生时抹去)期间,暂时禁用中断。同时,创建一个“陷阱帧 Trap Frame”来保存被中断线程的执行状态,当内核处理完中断或异常后恢复线程执行前,通过陷阱帧中保存的信息恢复处理器现场。陷阱帧通常是完整的线程描述表的子集。陷阱处理程序本身可以解决一些问题,如一些虚拟地址异常,但在大多数情况下陷阱处理程序只是确定发生的情况,并把控制转交给其他的内核或执行体模块。例如,如果情况是设备中断产生的,陷阱处理程序把控制转交给设备驱动程序提供给该设备的中断服务例程ISR(Interrupt Service Routine);如果情况是调用系统服务产生的,陷阱处理程序把控制转交给执行体中的系统服务代码;其他异常由内核自身的异常调度器响应。

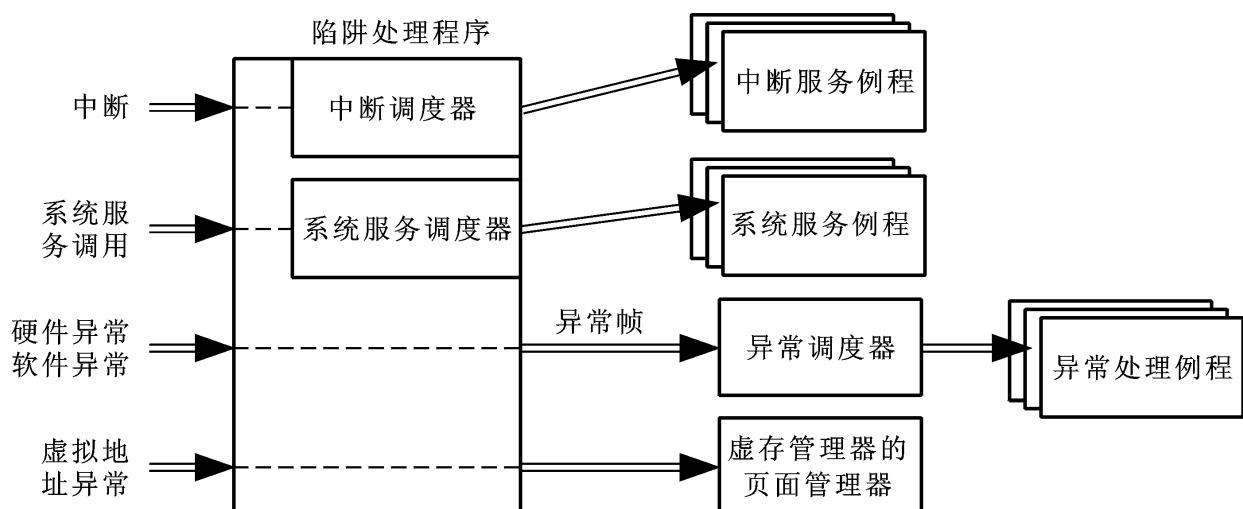


图2-7 Windows 2000/XP的陷阱调度

2. Windows 2000/XP 中断类型和优先级

由于不同处理器能识别不同的中断号和中断类型,所以先由Windows 2000/XP的中断调度器将中断级映射到由操作系统识别的中断请求级IRQL(Interrupt Request Level)的标准集上。这一组内核维护的IRQL是可以移植的,如果处理器具有特殊的与中断相关的特性(如第二时钟),则可以增加可移植的IRQL。IRQL将按照优先级排列中断,并按照优先级顺序服务中断,较高优先级中断可以抢占较低优先级中断服务。

图 2-8 显示了 x86 体系结构上可移植 IRQL 的映射。IRQL 从高往下直到设备级都是为硬件中断保留的；Dispatch/DPC（Deferred Procedure Call）和 APC（Asynchronous Procedure Call）中断是内核和设备驱动程序产生的软件中断。低优先级又叫被动级，并不是真正的中断级，在该级上运行普通线程，并允许发生所有中断。

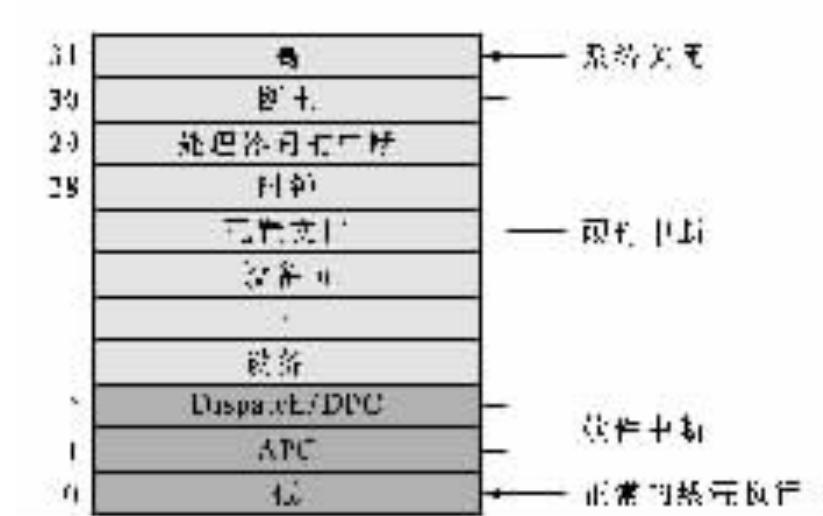


图 2-8 x86 体系结构 Windows 的中断请求级

每一个处理器都有一个 IRQL 设置，决定了该处理器可以接收哪些中断，其值随着操作系统代码的执行而改变。IRQL 也被用于同步访问核心数据结构，当核心态线程运行时，它可以提高或降低处理器的 IRQL。如图 2-9 所示，如果中断源高于当前的 IRQL 设置，则响应中断；否则该中断将被屏蔽，处理器不会响应该中断，直到一个正在执行的线程降低了 IRQL。

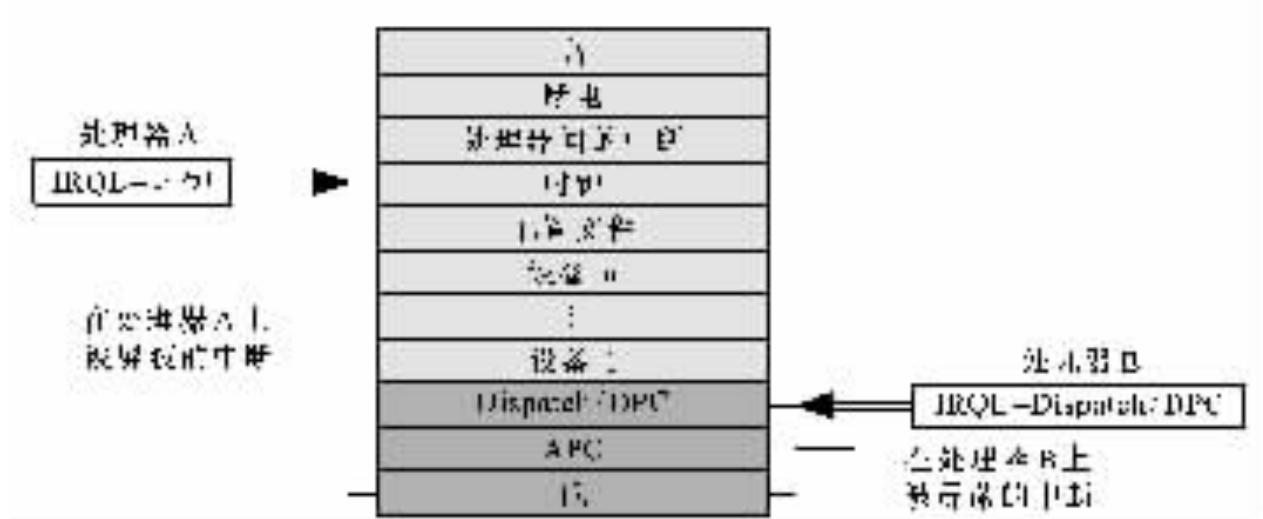


图 2-9 Windows 的中断屏蔽

核心态根据它要做的事情来提高或降低它所使用的处理器的 IRQL，例如，当中断产生

时, 陷阱处理程序提高处理器的 IRQL 直到与中断事件所指定的 IRQL 相同。这种技术确保了正在服务于该中断的处理器不会被同级或较低级的中断抢先。被屏蔽的中断将被另一个处理器处理或阻挡, 直到 IRQL 降低。由于改变处理器的 IRQL 对操作系统具有十分重要的影响, 所以, 它只能在核心态下改变, 用户态线程将始终无权改变 IRQL。

3. Windows 2000/XP 中断处理

(1) 硬件中断处理

当中断产生时, 陷阱处理程序将保存计算机运行程序的状态, 然后, 禁用中断并调用中断调度程序。中断调度程序立刻提高处理器的 IRQL 到当前中断源的级别, 以便在中断服务过程中屏蔽等于和低于当前中断源级别的其他中断。然后, 重新启用中断, 以使高优先级的中断仍然能够得到服务。

Windows 2000/XP 使用中断分配表 IDT (Interrupt Dispatch Table) 来查找处理特定中断的例程。中断源的 IRQL 作为表的索引, 表的入口指向中断处理例程, 如图 2-10 所示。

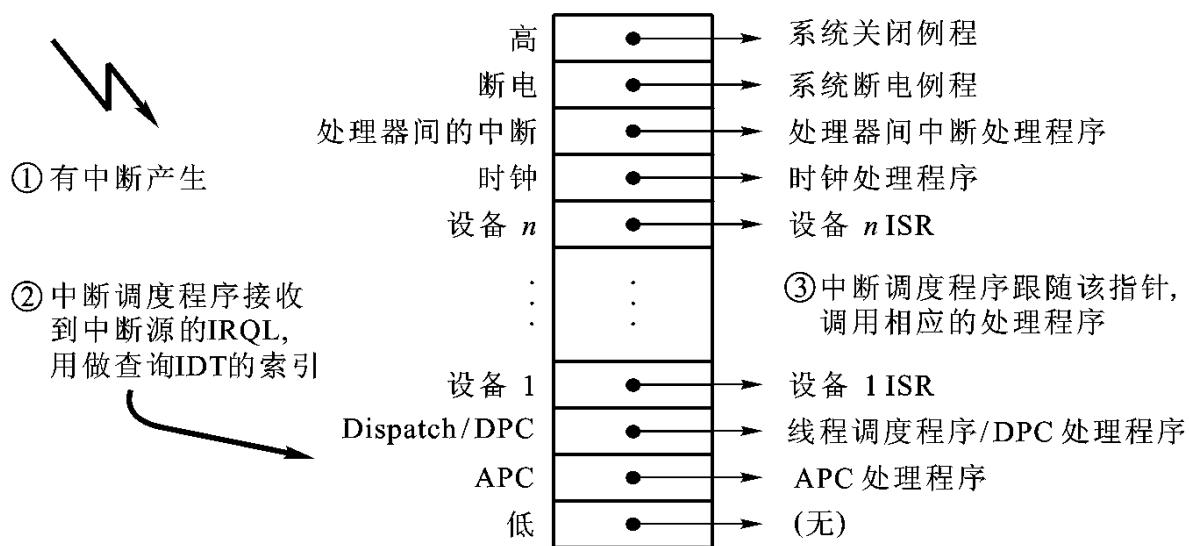


图 2-10 Windows 中断服务

在 x86 系统中, IDT 是处理器控制区 PCR (Processing Control Region) 指向的硬件结构, 有的系统用软件实现。PCR 和它的扩展——处理器控制块 PRCB 包括了系统中各种处理器状态信息。内核和硬件抽象层 HAL 使用该信息来执行体系结构特定的操作和机器特定的操作。这些信息包括: 当前运行线程、选定的下一个运行线程、处理器的中断级别等等。

在 x86 的体系结构中, 中断控制器可以支持 256 个中断行, 但是特定机器可以支持的中断行数量仍旧依赖于具体的中断控制器的设计, 大多数 x86 PC 机的中断控制器使用 16 个中断行。中断实际上进入了中断控制器的某一行。中断控制器依次在单个行上中断处理器。一旦处理器被中断, 它将询问控制器以获得中断向量。处理器利用此中断向量索引进

入 IDT 并将控制交给适当的中断服务例程。

在中断服务例程执行之后, 中断调度程序将降低处理器的 IRQL 到该中断发生前的级别, 然后加载保存的机器状态。被中断的线程将从它停止的位置继续执行。在内核降低了 IRQL 后, 被封锁的低优先级中断就可能出现。在这种情况下, 内核将重复以上过程来处理新的中断。

每个处理器都有单独的 IDT。这样, 不同的处理器就可以运行不同的中断服务例程 ISR。在多处理器系统中, 每个处理器都可以收到时钟中断, 但只有一个处理器在响应该中断时更新系统时钟。然而, 所有处理器都使用该中断来测量线程的时间片并在时间片结束后启动线程调度。同样的, 某些系统配置可能要求特殊的处理器处理某一设备中断。

大多数处理中断的例程都在内核中, 例如, 内核更新时钟时间, 在电源级中断产生时关闭系统。然而, 键盘、I/O 设备和磁盘驱动器等外部设备也会产生许多中断。这些设备的种类很多、变化很大, 设备驱动程序需要一种方法来告诉内核当设备中断发生时应调用哪个例程。为此, 内核提供了一个可移植的机制——中断对象, 它是一种内核控制对象, 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的将设备 ISR 和中断特定级相联系的所有信息, 其中有 ISR 地址、设备中断的 IRQL 以及与 ISR 相联系的内核入口。当中断对象被初始化后, 称为“调度代码”的一些汇编语言代码指令就会被存储在对象中。当中断发生时执行此代码, 这个中断对象常驻代码调用真正的中断调度程序, 给它传递一个指向中断对象的指针。中断对象包括了第二个调度程序例程所需要的信息, 以便定位和正确使用设备驱动程序提供的 ISR。

把 ISR 与特殊中断级相关联称为连接一个中断对象, 而从 IDT 入口分离 ISR 叫做断开一个中断对象。这些操作允许在设备驱动程序加载到系统时打开 ISR, 在卸载设备驱动程序时关闭 ISR。如果多个设备驱动程序创建多个中断对象并将它们连接到同一个 IDT 入口, 那么, 当中断在指定中断级上发生时, 中断调度程序会调用每一个例程。

使用中断对象来注册 ISR, 可以防止设备驱动程序直接随意中断硬件, 并使设备驱动程序无需了解 IDT 的任何细节, 从而有助于创建可移植的设备驱动程序。另外, 通过使用中断对象, 内核可以使 ISR 与可能同 ISR 共享数据的设备驱动程序的其他部分同步执行。进而, 中断对象使内核更容易调用多个任何中断级的 ISR。

(2) 软件中断

虽然硬件产生了大多数的中断, 但 Windows 2000/XP 内核也为多种任务产生软件中断, 它们包括: 启动线程调度、处理计时器到时、在特定线程的描述表中异步执行一个过程及支持异步 I/O 等。

4. 延迟过程调用和异步过程调用

当一个线程不能继续执行时, 可能由于它已经结束或进入了等待状态, 内核直接调用调

度程序立即实现描述表切换。然而,有时内核在深入多层代码内时检测到应该进行重调度。在这种情况下,理想的解决方法是请求调度,延迟它的产生直到内核完成当前活动为止。使用延迟过程调用 DPC 软件中断是实现这种延迟的简便方法。

当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 提高到 Dispatch/DPC 级或高于 Dispatch/DPC 级,这样就禁止了其他软件中断和线程调度。当内核检测到调度应该发生时,它将请求一个 Dispatch/DPC 级的中断。但由于 IRQL 等于或高于 Dispatch/DPC 级,处理器将在检查期间保存该中断。当内核完成当前活动后,它将 IRQL 降至低于 Dispatch/DPC 级,于是调度中断便可出现。

除了线程调度外,内核在其他 IRQL 上也处理延迟过程调用 DPC。有一种 DPC 是执行系统任务的函数,该任务比当前任务次要。这些函数叫做“延迟函数”,因为它们可能不立即执行。DPC 为操作系统提供了在内核态下产生中断并执行系统函数的能力。内核使用 DPC 处理定时器到时和在线程时间片结束后重调度处理器。设备驱动程序使用 DPC 完成 I/O 请求。

DPC 由 DPC 对象表示,它是一个内核控制对象。内核控制对象对于用户态的程序是不可见的,但对于设备驱动程序和其他系统代码是可见的。DPC 对象包含的最重要的信息是当内核处理 DPC 中断时将调用的系统函数的地址。等待执行的 DPC 例程被保存在称作“DPC 队列”的内核管理队列中。为了请求一个 DPC,系统代码将调用内核来初始化 DPC 对象,然后放入 DPC 队列中。

将一个 DPC 放入 DPC 队列会促使内核请求一个在 Dispatch/DPC 级的软件中断,因为通常 DPC 是由运行在较高 IRQL 级的软件对它进行排队的,所以,被请求中断直到内核降低 IRQL 到 APC 级或低于 APC 时才出现。

用户态线程是以低 IRQL 执行的,这是 DPC 中断普通用户线程执行的良好时机。DPC 例程不考虑什么线程正在运行,因而它不能假定当前映射的进程地址空间是什么。DPC 例程可以调用内核函数,但不能调用系统服务、产生页面故障及创建或等待对象。

DPC 主要是为设备驱动程序提供的,但内核也使用它们,最经常的应用是处理时间片到时。系统时钟的每个滴答在时钟 IRQL 都产生一个中断,内程序更新系统时间,并减小用来记录当前线程运行时间的计数器值。当计数值为 0 时,线程的时间片到,内核就可能需要重调度处理器,这是一个应在 Dispatch/DPC IRQL 完成的低优先级任务。时钟中断处理程序对 DPC 排队以启动线程调度,然后完成它的工作并降低处理器的 IRQL。因为 DPC 中断的优先级低于设备中断的优先级,所以任何挂起的设备中断将在 DPC 中断产生之前得到处理。

异步过程调用 APC 为用户程序和系统代码提供了一种在特殊用户线程的描述表(一个特殊的地址空间)中执行代码的方法。APC 在特殊线程描述表中执行,使用专用队列,它们以低于 2 的 IRQL 运行,APC 例程可以获得资源(对象)、等待对象句柄、导致页错错误及调用系

统服务。

APC 也是内核控制对象,称 APC 对象,等待执行的 APC 在内核管理的 APC 队列中。APC 队列与 DPC 队列的不同在于:DPC 队列是系统范围的,APC 队列是特定于线程的,每个线程都有自己的 APC 队列。当内核被要求对 APC 排队时,内核将 APC 插入到将要执行 APC 例程的线程的 APC 队列中。内核依次请求 APC 级的软件中断,并当线程最终开始运行时执行 APC。

有两种 APC:用户态 APC 和核心态 APC。后者在线程描述表中运行并不需要得到目标线程的“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,而不需要线程的干预和同意。

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)中完成操作系统的工作。例如,可以使用核心态 APC 命令一个线程停止执行可中断的系统服务,或记录在线程地址空间中的异步 I/O 操作的结果。环境子系统使用核心态 APC 将线程挂起或终止自身的运行,或得到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,如果启动了一个 I/O 操作并且线程进入等待状态,则另一个进程中的另一个线程就可以被调度去运行。当设备完成传输数据时,I/O 系统必须以某种方式重新进入到启动 I/O 系统线程的描述表中,以便它能够来执行这个动作。

一些 Win32 API,如 ReadiEX,WriteFileEX 和 QueueUserAPC,使用用户态 APC。例如,ReadiEX 和 WriteFileEX 允许调用者指定 I/O 操作完成时将被调用的完成例程。该完成例程是通过把 APC 排队到发出 I/O 操作的线程来实现的。

5. Windows 2000/XP 的异常调度

与随时可能发生的中断相比,异常是直接由运行程序的执行产生的情况。WIN32 引入了结构化异常处理(Structure Exception Handling)工具,它允许应用程序在异常发生时可以得到控制。然后,应用程序可以固定这个状态并返回到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。

除了那些简单的可以由陷阱处理程序解决的异常外,所有异常均由异常调度程序提供服务,异常调度程序的任务是找到能够处理该异常的异常处理程序。

如果异常产生于核心态,异常调度程序将简单地调用一个例程来定位处理该异常的基于框架的异常处理程序。由于没有被处理的核心态异常是一种致命的操作系统错误,所以,异常调度程序必须能找到异常处理程序。

内核俘获和处理某些对用户程序透明的异常,如调试断点异常,此时内核将调用调试程序来处理这个异常。少数异常可以被允许原封不动地过滤回用户态,如操作系统不处理的内存越界或算术异常。环境子系统能够建立“基于框架的异常处理程序”来处理这些异常。

“基于框架的异常处理程序”是指与特殊过程激活相关的异常处理程序,当调用过程时,代表该过程激活的堆栈框架就会被推入堆栈,堆栈框架可以有一个或多个与它相关的异常处理程序,每个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框架相关的异常处理程序,如果没有,内核将查找与前一个堆栈框架相关的异常处理程序。如此往复,如果还没有找到,内核将调用系统默认的异常处理程序。

当异常发生时都将在内存产生一个事件链。硬件把控制交给陷阱处理器,陷阱处理器将创建一个陷阱框架。如果完成了异常处理,陷阱框架将允许系统从中断处继续运行。陷阱处理器同时还要创建一个包含异常原因和其他有关信息的异常纪录。

6. Windows 2000/XP 系统服务调度

在 x86 处理器上执行 INT 2E 指令将引起一个系统陷阱,进入系统服务调度 (System Service Dispatcher) 程序,被传递的参数指明了被请求的系统服务号。如图 2-11 所示,内核将根据入口参数在系统服务调度表 (System Service Dispatch Table) 中查找系统服务信息。

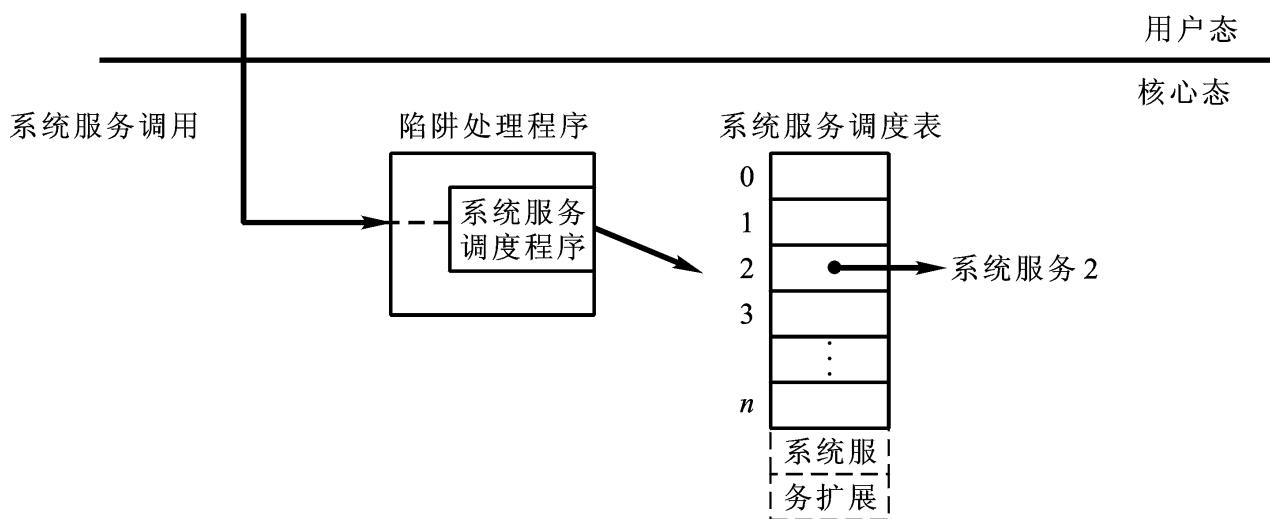


图 2-11 Windows 2000/XP 的系统服务

系统服务调度程序将校验参数,并且将调用者的参数从线程的用户堆栈复制到它的核心态堆栈中,然后执行系统服务。如果传递给系统服务的参数指向了在用户空间中的缓冲区,则在核心态代码访问用户缓冲区前,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 2000/XP 有两个内置的系统服务表,第一个默认表定义了在 NTOSKRNL.EXE 中实现的核心执行体系统服务;另一个包含了在 WIN32 子系统 WIN32K.SYS 的核心态部分中实现的 WIN32 USER 及 GDI 服务。当 WIN32 线程第一次调用 WIN32 USER 及 GDI 服务时,线程系统服务表的地址将指向包含 WIN32 USER 及 GDI 的服务表。

用于 Windows 2000/XP 执行体服务的系统服务调度指令存在于系统库 NTDLL.DLL 中。子

系统的 DLL 通过调用 NTDLL 中的函数来实现它们的文档化函数。出于效率考虑,例外的是,WIN32 USER 及 GDI 函数的系统服务调度指令是在 USER32.DLL 和 GDI32.DLL 中直接实现。

如图 2-12 所示, KERNEL32.DLL 中的 WIN32 WriteFile 函数调用 NTDLL.DLL 中的 NtWriteFile 函数,它依次执行适当的指令以引发系统陷阱,传递代表 NtWriteFile 的系统服务号码。然后系统服务调度程序(NTOSKRNL.EXE 中的 KiSystemService 函数)调用真正的 NtWriteFile 来处理 I/O 请求。对于 WIN32 USER 及 GDI 函数,系统服务调度调用在 WIN32 子系统可加载核心态部分 WIN32K.SYS 中的函数。

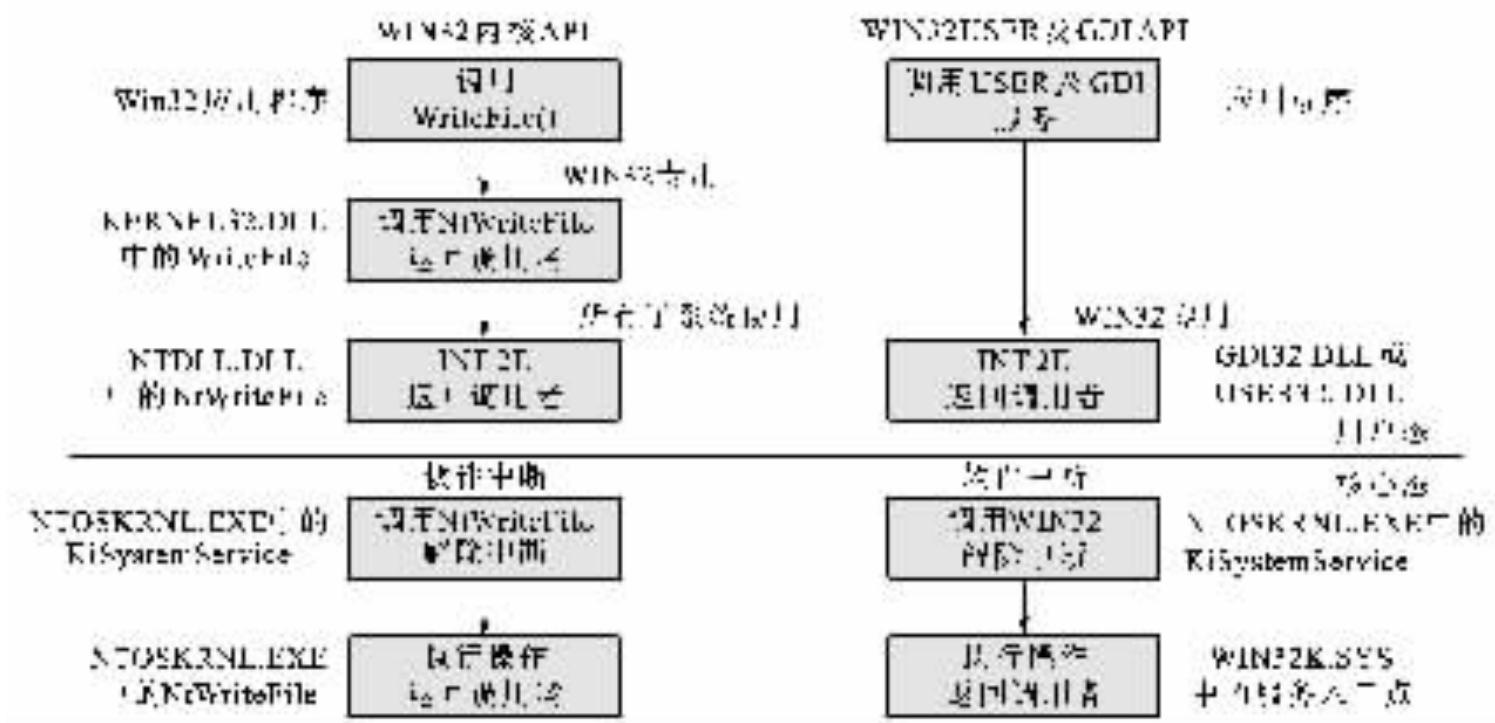


图 2-12 Windows 2000/XP 的系统服务调度

2.2.8 实例研究: Solaris 中断处理

1. Solaris 中断处理概述

在 SPARC 和 ULTRA SPARC 芯片上的 Solaris 中断处理的实现中, 使用了系统调用、中断和陷阱等术语。控制从用户程序转换到内核可以通过一个系统调用、或一个中断、或一个处理器陷阱引起。

系统调用(system call)是用户进程请求内核服务的机制。一般来说, 系统调用是从用户模式通过一条陷阱指令或一个软件中断引起的, 并且依赖于微处理器和平台。

中断(interrupt)是一个控制进入内核的向量转移。中断一般是由硬件设备引起的, 与正在执行的线程异步产生。另外, 中断也可以由软件产生。

陷阱(trap)也是一个控制进入内核的向量转移,它是由处理器引起的。陷阱和中断之间最大的区别是:陷阱由正在执行的线程导致,而中断则是异步事件。

2. ULTRA SPARC 的陷阱

SPARC 和 ULTRA SPARC 处理器体系使用陷阱作为一个统一的机制来处理系统调用、处理器异常和陷阱。SPARC 陷阱是由微处理器引起的过程调用,用于处理同步的处理器异常、异步的处理器异常、软件引起的陷阱指令、或设备中断引起的中断处理。

SPARC 和 ULTRA SPARC 处理器在接到一个陷阱时进入特权模式,并将控制转移给预先存放在陷阱表中的起始指令。然后,根据接收到的陷阱类型执行该陷阱的处理程序,处理结束后将控制权返回给被中断的线程。陷阱使硬件进行以下动作:(1)保存某些处理器状态信息(程序计数器、状态寄存器、陷阱类型等);(2)进入特权执行模式;(3)开始执行陷阱表中的相应代码。

陷阱表驻留在内存中,它含有每种类型陷阱的前 8 条指令。陷阱表的内存位置存放在陷阱基址寄存器 TBA 中,该寄存器在系统引导时被初始化。Solaris 把陷阱表的指令放在内核底部的一个 4M 的锁定页面中,这样在执行陷阱表的指令时还可以处理如页面处理等其他与内存无关的陷阱。

SPARC 和 ULTRA SPARC 为每一类陷阱在陷阱表中设置一个入口,提供一个特定的处理程序。SPARC 和 ULTRA SPARC 的陷阱可以分为以下几种:(1)处理器重置(加电重置、机器重启、软件引起的重置);(2)存储管理异常(页面错误、破坏页保护、存储错误、偏移访问);(3)指令异常(非法指令,常态下运行特权指令);(4)浮点异常;(5)寄存器异常;(6)中断陷阱,用于系统调用入口的软件引起陷阱。

SPARC 和 ULTRA SPARC 陷阱有一个相关的优先级别,加电重置最高,以下是各类处理器异常,软件陷阱和中断陷阱。当某时刻有多个陷阱发生时,陷阱硬件使用优先级来确定哪个陷阱应该优先执行。另外,中断陷阱还要和处理器中断级 PIL(Processor Interrupt Level)进行比较,只有那些比 PIL 级别高的中断陷阱才会被处理。

ULTRA SPARC 支持嵌套陷阱,即支持陷阱的嵌套处理。嵌套陷阱共有 5 级,0 为正常执行,4 为错误处理状态,事实上支持 3 层嵌套处理。

3. Solaris 中断和中断处理

Solaris 中,中断是一种设备使用的机制,设备向内核发送信号,以通知内核该设备需要得到注意并要求立即处理。Solaris 对中断的服务是通过切换出正在处理器上运行的线程的上下文,并为该中断设备执行一个中断处理程序进行的。

中断也有优先级,从 1 级到 15 级,15 级是最高的优先级别。内核可以通过设置处理器的中断级别来屏蔽设定优先级以下的中断。比如,处理器正在执行一个 9 级中断处理程序,

它就不会处理 9 级或者 9 级以下的中断, 它仅处理更高优先级的中断。等于或低于处理器中断级别的中断暂时被忽略, 直到处理器中断级别低于正在等待的中断的优先级。

为了降低系统开销, Solaris 的中断被转换成一个内核线程进行处理, 称为中断处理内核线程。和任何内核线程一样, 中断处理内核线程有自己的标识号、优先级、上下文环境和堆栈。由内核来控制访问共享的数据结构, 并使用互斥原语在中断处理内核线程之间进行同步, 也就是说, 通常用于线程的同步技术也可用于中断处理。为了让中断得到优先处理, 通常中断处理内核线程被赋予更高的优先级, 高于所有其他类型的内核线程。当一个中断发生时, 它被传送给某个特定的处理器, 正在该处理器上执行的线程被钉住。被钉住的线程不能移动到另一个处理器上, 并且它的上下文被保存起来, 该线程仅仅被“挂起延迟”, 直到处理完中断。然后, 处理器开始执行一个中断处理内核线程, 有一个不活跃的中断线程池可供使用, 因而不需要创建一个新线程。中断处理内核线程开始执行, 即开始处理中断事件, 如果这时需要访问一个数据结构, 而该数据结构当前正以某种方式被另一个正在执行的线程锁定, 中断处理内核线程必须等待访问。一个中断处理内核线程仅能被另一个具有更高优先级的中断处理内核线程剥夺。Solaris 把中断转换为内核线程来处理的经验表明, 这种方法可以比传统的中断处理策略提供更好的性能。

4. Solaris 系统调用

Solaris 通过内核 sysent 表来引用系统调用, 该表含有系统支持的所有系统调用的入口。sysent 表是一个结构数组, 使用系统调用号进行索引, 每个分量记录一个系统调用的入口。

系统调用的处理过程是:(1)通过系统调用陷阱陷入内核;(2)调用系统调用陷阱处理程序;(3)保护未被硬件保护的处理器现场;(4)分析参数;(5)查 sysent 表, 找到具体的系统调用处理程序;(6)执行系统调用处理程序;(7)设置返回值;(8)返回。

系统调用陷阱通过陷阱指令 tcc 引起。该指令包含一个 6 位的系统调用陷阱号, 可以用于表示系统调用号。

系统调用陷阱有 3 个:一个用于本地系统调用, 一个用于在 64 位内核上调用 32 位系统的系统调用, 一个用于调用 SUN OS 的系统调用。另外, 硬件还提供了几个额外的软件陷阱, 以便操作系统实现快速系统调用。

2.2.9 实例研究:Linux 中断处理

1. Linux 中断处理过程

Linux 中断处理子系统的任务之一是当中断发生时调用正确的中断处理例程。为了实现这一点, 使用了两个数据结构:irq-action 和 irqaction, 其中, irqaction 中含有处理一种中断所需的各种信息:

```

struct irqaction
{
    void (* handler)(int, void *, struct pt-reg * );
    unsigned long flags;
    unsigned long mask;
    const char * name;
    void * dev-id;
    struct irqaction * next;
};

```

其中,包括函数指针 `handler`,指向中断例程地址;特征位 `flags`,如是否允许中断嵌套;SPARC64 平台需使用的 `mask`;生成中断的硬件设备名字 `name`;硬件厂商定义的硬件类型标识符 `dev-id` 和共享 IRQ 时,队列中下一个 `irqaction` 结构的指针 `next` 等。`irq-action` 是一个向量,其中的每一个元素是一个指向 `irqaction` 的指针。

外设组件互连 PCI(Peripheral Component Interconnect)是 PC 机上的一个外设组件互连标准,在高端服务器上,为了使用更多 PCI 设备,需要采用 PCI 桥技术。这样一来,PCI 中断源的数目可能超过中断控制器的中断引脚数,可以让多个 PCI 设备共用一个中断引脚,这就是设备的中断共享。在 Linux 中,实现中断共享时让某个中断号对应多个 PCI 设备,由一个中断源的第一次中断请求声明是否允许共享中断。使用一个 `irq-action` 指向多个链接起来的 `irqaction`,而且链表上的中断处理例程应为相同类型。

当中断发生时,Linux 首先根据中断控制器的状态寄存器确定中断源,再根据中断源的编号计算出该中断在 `irq-action` 中的偏移量。如果该中断源设备相应的 `irqaction` 不存在,系统会在日志中记录一个错误,否则根据 `irqaction` 中的地址调用中断处理例程。当共享中断发生时,系统调用该中断所对应的所有中断处理例程(`irqaction` 中的 `handler`),即使某个设备未发生中断,其中断处理例程也会被调用。

当设备驱动程序中的中断处理子例程被核心调用时,中断处理例程必须找出发生中断的原因并作出处理。为了找出发生中断的原因,设备驱动程序必须读出中断设备的状态寄存器,发生中断的原因可能是设备完成操作或产生了某种错误。一旦确定了中断发生的原因,设备驱动程序就可进行相应的处理。

Linux 核心含有多种设备驱动程序,并且有如下一些特性:(1)核心代码 设备驱动程序是系统核心的重要组成部分,代码都是核心代码。如果出现错误,会造成系统严重破坏,如破坏文件系统或丢失数据。(2)核心接口 设备驱动程序提供标准的核心接口,如终端设备驱动程序提供文件 I/O 接口,SCSI 设备驱动程序提供文件 I/O 接口及缓冲区缓存接口等。(3)核心机制和服务 设备驱动程序使用标准的系统服务,如内存分配、等待队列等来完成操

作,还可以利用核心系统服务:请求中断、允许中断、屏蔽中断或向系统注册设备专用的中断处理例程。此外,设备驱动程序还具有可装载性:它可以核心模组形式动态地装入到核心和从中卸出;可配置性:在编译核心时指定并配置到核心等特性。

2. 快中断与慢中断

在 Linux 中,可以区分快中断和慢中断两类中断事件。前者用于时间短、简单的中断处理任务;而后者处理常见的中断,需要时间较长且处理复杂。两者的主要区别为:(1)处理慢中断前需保存所有寄存器的内容,而快中断处理最初保存现场时,仅要保存那些被常规 C 函数修改的寄存器。(2)在慢中断处理时,通常不屏蔽其他中断信号,而快中断处理时会屏蔽所有其他中断。(3)慢中断处理完毕后,通常不立即返回被中断的进程,而是进入调度程序重新进行调度,调度结果未必是被中断的进程运行(是抢先式调度)。而快中断处理完毕后,通常恢复现场返回被中断的进程继续执行(是非抢先式调度)。为了尽快缩短快中断处理时间,以便及时响应处理期间到达的其他中断信号,便引入了底半处理的概念。

3. 底半处理

发生中断时,处理器暂停当前执行的指令,系统负责把中断发送到相应的设备驱动程序去处理。由于设备驱动程序都是核心态代码,通常在中断处理过程中,系统要关闭中断,不再能进行其他任何工作,这段时间里系统资源的利用率是非常低的。为了缩短屏蔽中断的时间,希望设备驱动程序以最快的速度在核心态下完成与中断事件有关的处理,而把中断事件的其他大部分耗时的工作留在中断处理例程之外,由系统自行安排运行时机(不在中断服务上下文中)执行。这种一部分工作由核心代码在关中断状态下处理,另外一部分工作由非核心态代码来处理的方式称为底半处理 (bottom half handling)。Linux 用底半处理机制来实现中断事件的快速处理,它可以让设备驱动程序和核心其他部分将这些工作进行排序以延迟执行,可见 bottom half handling 是一种任务延迟处理机制。于是, Linux 的中断服务例程可以分成上半部分 (top half of an interrupt handler) 中断处理过程和底半部分 (bottom half of an interrupt handler) 中断处理过程。其中前者就是在中断向量表中登记的中断服务例程的入口部分,每当中断事件发生时,在关中断状态下 top half 处理过程立即执行,但是 bottom half 处理过程却被推迟并可打开中断执行,这是通过把 top half 处理过程和 bottom half 处理过程分立为独立的函数并对其区别对待实现的。top half 处理过程要决定其相关的 bottom half 处理过程是否需要执行,不能推迟的部分显然不会属于 bottom half 部分。但可以推迟的部分总让它属于底半处理部分,再去排队等待工作,这一任务是通过设置 bh-active 中的一个位来标志的。因此,当处理快中断时,如果有其他中断到达——不论是快中断还是慢中断,它们都必须等待。为了尽可能快地处理来到的其他中断,内核就需要尽可能地将耗时的处理延迟到底半处理过程执行。

区分 top half 处理过程和底半处理过程的另一个原因, 是中断服务例程的底半处理过程包含有一些中断所不一定非要执行的操作, 只要内核可以在一系列中断之后从某些地方得到。在这种情况下, 执行中断的底半处理过程是一种浪费, 它可以稍稍延迟并在以后仅执行一次。例如, top half 处理过程已标记 bottom half 处理过程必须执行, 这种标记可能重复多次。如果在内核有机会运行 bottom half 处理过程之前给定的设备就已经发生了 100 次中断, 那么, 中断服务程序的 top half 处理过程就运行 100 次, 底半处理过程只要运行 1 次。bottom half 处理在 Linux 内核中又被称为是“软中断处理”。很多情况下, 软中断又是和硬中断相对应的; “硬中断”是外部设备对 CPU 的中断, top half 是硬中断; 同时, “软中断”通常是硬中断服务程序对内核的中断, bottom half 是软中断; 而“信号”也是一种软中断, “信号”是由内核或进程对其他进程的中断。

4. 任务队列

在 Linux 的内核中设立了任务队列, 这是核心对任务进行延迟处理的一种方法, 提供了对任务队列中任务排队及处理的通用机制。遇到一个任务时, 可以立即被处理, 也可以把它插入某个队列留在稍后再行处理。那么, 简单地说, 底半处理要利用到队列机制。底半处理是中断的下半部分, 在下半部分的处理中可以挂接多个任务(其实是挂接函数, 比如一些处理例程 handler()), 这时就要用到任务队列。队列在 Linux 中应用相当广泛, 不止在底半处理中要用到, 还在其他许多场合被使用。但是任务队列可以由用户动态定义和管理, 而底半处理过程却是由 Linux 静态定义的, 且不能超过 32 种。

任务队列由 tq-struct 单向链表结构组成, 如图 2-13 所示。每个 tq-struct 数据结构是任务队列的节点, 它包含一个处理例程地址、指向一些数据的指针和下一个任务节点指针。当任务队列中的节点被处理时, 将调用例程并传递参数指针。

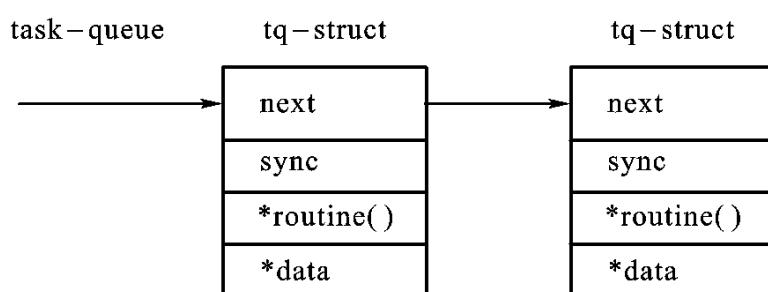


图 2-13 任务队列

Linux 核心中的任何部分, 如设备驱动程序, 都可以建立并使用队列机制, 核心建立和维护三个一般性任务队列:

(1) 定时器队列(TQ-TIMER) 该队列用来对在系统时钟信号到来之后需要尽快处理的任务进行排队。每次时钟滴答到来时, 都要检查定时器队列是否为空, 如果不空则要把定

时器的底半处理过程标记为活动状态,以便激活此任务。在进程调度下一次执行时,定时器的底半处理过程被调用,定时器队列中排队的任务也就被处理了。

(2) 即时队列(TQ - IMMEDIATE) 调度程序对的底半处理过程进行处理时,即时队列中的任务立即得到处理。即时底半处理过程的优先级较低,因此,与定时器的底半处理过程相比,这个队列中的任务处理要稍迟一些。

(3) 进程调度队列(TQ - SCHEDULER) 该任务队列由进程调度程序直接处理,主要用来支持系统中的其他任务队列。

5. 底半处理数据结构

图 2-14 描述了内核的底半处理的数据结构,最多能有 32 个不同的底半处理过程。bh-base 是底半处理过程的入口指针,bh-mask 和 bh-active 共同控制的底半处理过程能否运行,分别指明是否有底半处理过程被安装和是否是有效的位设置,若 bh-mask 的位 n 被置位则 bh-base 的第 n 个元素包含一个底半处理过程的入口地址。若 bh-active 的位 n 被置位,则第 n 个处理程序能被调度程序任意调用,只要对 bh-mask 和 bh-active 进行位“AND”运算就能够表明应该运行哪一个底半处理过程,如果“位与”运算的结果为 0,就没有底半处理过程需要运行。这些索引在入口表中被静态设置,计时器的底半处理过程入口在索引 0,具有最高优先级;控制台的底半处理过程入口在索引 1 等等。典型的底半处理过程有一个任务队列相关联,例如,用于多个设备驱动器排队工作的通用处理程序是 immediate,它的底半处理过程通过包含需要被立即实现的任务的任务队列(tq-immediate)工作,当调度程序调用该底半处理过程时,它就会处理与之相关的任务队列中的各个任务。

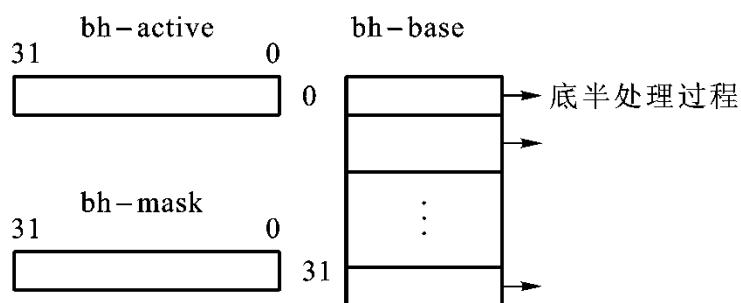


图 2-14 底半处理数据结构

6. 底半处理的执行过程

核心中的某些底半处理过程是和特定设备相关的,而其他底半处理过程则可以灵活地定义。多数低半处理过程与设备驱动有关,由核心定义的部分通用底半处理过程有:

(1) 定时器(TIMER - BH)。当系统周期性定时信号(时钟中断)到来时,定时器底半处理过程被标记为活动状态,并被系统用来驱动核心的定时器队列机制。

(2) 控制台(CONSOLE - BH)。处理进程控制台消息的底半处理过程。

(3) 消息队列(TQUEUE - BH)。处理进程消息的底半处理过程。

(4) 串行口(SERIAL - BH)。串行口底半处理过程。

(5) 网络(NET - BH)。用于一般网络处理的底半处理过程。

(6) SCSI 设备(SCSI - BH)。用于 SCSI 设备底半处理过程。

(7) 即时(IMMEDIATE - BH)。一种一般性处理过程,许多设备驱动程序利用该过程对要在随后处理的任务进行排队的通用底半处理过程。

(8) 键盘(KEYBOARD - BH)。键盘消息的底半处理过程。

只要一个设备驱动器,或核心的其他一些部分需要对某些任务进行排队处理时,系统把任务添加到适当的任务队列(如核心的定时器队列)中,然后通知核心进行底半处理。具体做法是需要设置 bh-active 的适当位来完成。例如,如果设备驱动器在 immediate 队列上将某个任务排队,并希望运行 immediate 的底半处理过程来处理排队的任务。由于 bh-active 的位掩码的相应位已置为 1,在下面三种情况下:

- 当调度程序欲选择下一个运行进程之前(Schedule())。
- 当从系统调用返回之前(ret_from_syscall)。
- 当每个中断处理和异常处理返回前(ret_from_intr 和 ret_from_exception)。

系统都要检查 bh-active 中的每个位,如果有任何位被置位,则会调用 do-bottom-half() 执行有效的底半处理过程。首先检查位 0,然后位 1,直到 31 位。bh-active 的位在每一个底半处理过程被调用时清除。有关的数据结构和函数请参阅 include/linux/interrupt.h 和 kernel/softirq.c 文件。

下面通过定时器中断(零号中断 IRQ0)的例子来说明中断服务例程的 top half 与底半处理过程之间的联系。定时器中断服务例程(函数)叫 timer-interrupt,它的 top half 函数为 do-timer 而它的 bottom half 函数叫 timer-bh。当产生定时器中断时,timer-interrupt 从 CPU 计时器中读取数据后,调用 top half。do-timer() 做如下工作:更新全局变量 jiffies(加 1),该值记录了机器启动以来系统时钟滴答的次数;递增丢失(没有被底半处理过程处理、也就是两次 bottom half 处理之间的滴答次数)的定时器滴答数;执行 mark-bh 标记底半处理过程为活动状态;如果定时器队列有任务在等待,标记底半处理过程准备好运行。此外,还调用 update-process-times() 来更新进程的时间片及修改进程的动态优先级。整个定时器中断处理看起来十分简单,这是因为有很大一部分工作都被延迟到底半中处理了。timer-bh 需要调用 update-times() 和 run-times-list() 做更新进程和内核有关时间的统计数字。涉及有关的时间的统计数据有:系统当前的平均负载、当前时间的全局变量、当前运行进程在核心态和用户态使用的以 jiffies 为单位的时间值等。此外,它的一个主要任务是执行定时器的操作,检查和执行定时服务。

7. Linux 软中断机制

在 Linux 内核中, bottom half 最初用于在特权级较低的上下文中完成中断服务的非关键耗时动作, 现在也用于一切可在低优先级的上下文中执行的异步动作。最早的 bottom half 实现是借用了中断向量表, 在目前的 v2.4.x 内核中仍然可以看到:

```
static void (*bh_base[32])(void); /* kernel/softirq.c */
```

系统定义了一个函数指针数组, 共有 32 个函数指针, 采用数组索引来访问, 与此相对应的是一套函数: void init_bh(int nr, void (*routine)(void)); void remove_bh(int nr) 和 void mark_bh(int nr) 等配合工作。但在 Linux 新版本中 bh 的语义和使用方式已经很不一样了, 这三个函数仅仅是在接口上保持了向下兼容, 在实现上一直都在随着内核的软中断机制在变化。现在, 在 v2.4.x 内核里, 它用的是特殊的软中断——tasklet 机制。

介绍 tasklet 之前, 有必要先看一下 task queue 机制。显而易见, 原始的 bottom half 机制有很大的局限性, 最重要就是软中断个数限制在 32 个以内, 随着系统硬件越来越多, 软中断的应用范围越来越大, 这个数目显然是不够用的, 而且, 每个 bottom half 上只能挂接一个函数, 也是不够用的。因此, 在 v2.0.x 内核里, 已经用任务队列 task queue 的办法对其进行了扩充, task queue 是在系统队列数据结构的基础上建成的, 以下即为 task queue 的数据结构, 这里使用的是 v2.4.2 中的实现:

```
struct tq_struct
{
    struct list_head list; /* 链表结构 */
    unsigned long sync; /* 初始为 0, 入队时置 1, 以避免重复入队 */
    void (*routine)(void *); /* 激活时调用的函数 */
    void *data; /* routine(data) */
};
```

run_task_queue(task_queue *list) 函数可用于启动 list 中挂接的所有 task, 可以挂接在 bottom half 向量表中来启动, 从而, 可以用于扩充 bottom half 的个数。此时, 只需调用 mark_bh(IMMEDIATE_BH), 让 bottom half 机制在合适的时候调度它。

可见, task queue 以 bottom half 为基础; 而 bottom half 在 v2.4.x 中则以新引入的 tasklet 为实现基础。之所以引入 tasklet, 是因为 bottom half 是串行处理的, 不能适应 SMP 环境, 提高 SMP 多个 CPU 的利用率: 不同的 tasklet 可以同时运行于不同的 CPU 上, 相同的 tasklet 不会同时在不同的 CPU 上运行。

```
struct tasklet_struct
```

```
{
    struct tasklet_struct *next; /* 队列指针 */
    unsigned long state;          /* tasklet 的状态,按位操作,目前定义了两个位的
                                    含义:TASKLET_STATE_SCHED(第 0 位)或 TASKLET_STATE_RUN(第 1 位) */
    atomic_t count;               /* 引用计数,通常用 1 表示 disabled */
    void (*func)(unsigned long); /* 函数指针 */
    unsigned long data;           /* func(data) */
};

};
```

把 tasklet 结构与 tq_struct 比较,可以看出,tasklet 扩充了一点功能,主要是 state 属性,用于 CPU 间的同步。tasklet 的使用比 task queue 更简单,而且,tasklet 还能更好的支持 SMP 结构,因此,在新的 v2.4.x 内核中,tasklet 是建议的异步任务执行机制。

从前面的讨论可以看出,task queue 基于 bottom half, bottom half 基于 tasklet,而 tasklet 则基于 softirq。可以说,softirq 沿用的是最早的 bottom half 思想,但在这个“bottom half”机制之上,已经实现了一个更加庞大和复杂的软中断子系统。和 bottom half 类似,系统也预定义了 4 个 softirq_vec[] 结构,通过以下枚举表示:

```
enum {
    HI_SOFTIRQ = 0,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    TASKLET_SOFTIRQ
};
```

HI_SOFTIRQ 被用于实现 bottom half, TASKLET_SOFTIRQ 用于公共的 tasklet 使用,NET_TX_SOFTIRQ 和 NET_RX_SOFTIRQ 用于网络子系统的报文收发。在软中断子系统初始化(softirq_init())时,调用了 open_softirq() 对 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 做了初始化。do_softirq() 用于处理软中断,当发现那个软中断标志位置位,就调用相应的软中断处理函数。Do-softirq() 有 4 个执行时机,分别是:从系统调用中返回(ret_from_sys_call)、从异常中返回(ret_from_exception)、调度程序中(schedule),以及处理完硬件中断之后(do_IRQ)。它将遍历所有的 softirq_vec,依次启动其中的 action()。需要注意的是,软中断服务程序,不允许在硬中断服务程序中执行,也不允许在软中断服务程序中嵌套执行,但允许多个软中断服务程序同时在多个 CPU 上并发执行。

2.3 进程及其实现

2.3.1 进程的定义和属性

进程的概念是操作系统中最基本、最重要的概念。它是多道程序系统出现后,为了刻画系统内部出现的动态情况,描述系统内部各道程序的活动规律而引进的一个新概念,所有多道程序设计的操作系统都建立在进程的基础上。操作系统专门引入进程的概念,从理论角度看,是对正在运行的程序活动规律的抽象;从实现角度看,则是一种数据结构,目的在于清晰地刻画动态系统的内在规律,有效管理和调度进入计算机系统主存储器运行的程序。下面进一步来讨论,操作系统中为什么要引入进程呢?

一是刻画系统的动态性,发挥系统的并发性,提高资源利用率。在多道程序环境下,程序可以并发执行,一个程序的任意两条指令之间都可能发生随机事件而引发程序切换。因而,每个程序的执行都可能不是连续的而是走走停停的。此外,程序的并发执行又引起了资源共享和竞争的问题,造成了各并发执行的程序间可能存在制约关系。并发执行的程序不再处在一个封闭的环境中,出现了许多新的特征,系统需要一个能描述程序动态执行过程的单位,这个基本单位就是进程。同静态的程序相比较,进程依赖于处理器和主存储器资源,具有动态性和暂时性。进程随着一个程序模块进入主存储器并获得一个数据块和一个进程控制块而创建,因等待某个事件发生或资源得不到满足而暂停执行,随着运行的结束退出主存储器而消亡,从创建到消亡期间,进程处于不断的动态变化之中。此外,由于程序的并发执行,使得处理器和 I/O 设备、I/O 设备和 I/O 设备能有效地并行工作,提高了资源的利用率和系统的效率。由此可见,进程是并发程序设计的一种有力工具,操作系统中引入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”和提高资源的利用率。

二是解决共享性,正确描述程序的执行状态。也可以从解决“共享性”来看操作系统中引入进程概念的必要性。首先,引入“可再入”程序的概念,所谓“可再入”程序是指能被多个程序同时调用的程序。另一种称“可再用”程序由于它被调用过程中具有自身修改,在调用它的程序退出以前是不允许其他程序来调用它的。“可再入”程序具有以下性质:它是纯代码,即它在执行中自身不被改变;调用它的各程序应提供工作区,因此,可再入程序可以同时被几个程序调用。

在多道程序设计系统里,编译程序常常是“可再入”程序,因此它可以同时编译若干个源程序。假定编译程序 P 现在正在编译源程序甲,编译程序从起始点 A 开始工作,当执行到 B 点时需要将信息记到磁盘上,且程序 P 在 B 点等待磁盘传输数据完成。这时处理器空闲,

为了提高系统效率,利用编译程序 P 的“可再入”性,可让编译程序 P 再为源程序乙进行编译,仍从 A 点开始工作。现在应怎样来描述编译程序 P 的状态呢?称它为在 B 点等待磁盘传输状态,还是称它为正在从 A 点开始执行的状态?编译程序 P 只有一个,但加工对象有甲、乙两个源程序,所以再以程序作为占用处理器的单位显然是不适合的了。为此,可把编译程序 P 与服务对象联系起来,P 为源程序甲服务就说构成进程 $P_甲$,P 为源程序乙服务则构成进程 $P_乙$ 。这两个进程虽共享编译程序 P,但它们可同时执行且彼此按各自的速度独立进行。现在可以说进程 $P_甲$ 在 B 点处于等待磁盘传输的状态,而进程 $P_乙$ 正处在(从 A 点开始)执行的状态。可见程序与计算(程序的执行)不再一一对应,延用程序概念不能描述这种共享性,因而引入了新的概念——进程,它是既能描述程序并发执行过程又能用来共享资源的一个基本单位。但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(process)这个名词最早是 1960 年在 MIT 的 MULTICS 和 IBM 公司的 TSS/360 系统中提出的,直到目前对进程的定义和名称均不统一,不同的系统中采用不同的术语名称,例如,MIT 称进程(process),IBM 公司称任务(task)和 Univac 公司称活动(active)。可以说进程的定义多种多样,国内学术界较为一致的看法是:进程是一个可并发执行的具有独立功能的程序关于某个数据集合的一次执行过程,也是操作系统进行资源分配和保护的基本单位(1978 年全国操作系统学术会议)。它具有如下属性:

- 结构性:进程包含了数据集合和运行于其上的程序。为了描述和记录进程的动态变化过程使其能正确运行,还需配置一个进程控制块,所以每个进程至少有三要素组成:程序块、数据块和进程控制块。
- 共享性:同一程序同时运行于不同数据集合上时,构成不同的进程。或者说,多个不同的进程可以共享相同的程序,所以进程和程序不是一一对应的。
- 动态性:进程是程序在数据集合上的一次执行过程,是动态概念。同时,它还有生命周期,由创建而产生,由调度而执行,由撤销而消亡。而程序是一组有序指令序列,是静态概念,所以,程序作为一种系统资源是永久存在的。
- 独立性:进程既是系统中资源分配和保护的基本单位,也是系统调度的独立单位(单线程进程)。凡是未建立进程的程序,都不能作为独立单位参与运行。通常,每个进程都可以各自独立的速度在 CPU 上推进。
- 制约性:并发进程之间存在着制约关系,进程在进行的关键点上需要相互等待或互通消息,以保证程序执行的可再现性和计算结果的唯一性。
- 并发性:进程可以并发地执行,进程的并发性能改进资源利用率和提高系统效率。对于一个单处理器的系统来说, m 个进程 P_1, P_2, \dots, P_m 是轮流占用处理器并发地执行。例如,可能是这样进行的:进程 P_1 执行了 n_1 条指令后让出处理器给 P_2 , P_2 执行了 n_2 条指令

后让出处理器给 P_3, \dots, P_m 执行了 nm 条指令后让出处理器给 P_1, \dots 。因此, 进程的执行是可以被打断的, 或者说, 进程执行完一条指令后在执行下一条指令前, 可能被迫让出处理器, 由其他若干个进程执行若干条指令后才能再次获得处理器而执行。

2.3.2 进程的状态和转换

1. 三态模型

一个进程从创建而产生至撤销而消亡的整个生命期间, 有时占有处理器执行, 有时虽可运行但分不到处理器, 有时虽有空闲处理器但因等待某个事件的发生而无法执行。这一切都说明进程和程序不相同, 它是活动的且有状态变化的, 这可以用一组状态加以刻画。为了便于管理进程, 一般来说, 按进程在执行过程中的不同情况至少要定义三种不同的进程状态:

- 运行(running)态: 进程占有处理器正在运行。
- 就绪(ready)态: 进程具备运行条件, 等待系统分配处理器以便运行。
- 等待(wait)态: 又称为阻塞(blocked)态或睡眠(sleep)态, 指进程不具备运行条件, 正在等待某个事件的完成。

通常, 一个进程在创建后将处于就绪状态。每个进程在执行过程中, 任一时刻当且仅当处于上述三种状态之一。同时, 在一个进程执行过程中, 它的状态将会发生改变。图 2-15 表示进程的状态转换。



图 2-15 进程三态模型及其状态转换

运行状态的进程将由于出现等待事件而进入等待状态, 当等待事件结束之后等待状态的进程将进入就绪状态, 而处理器的调度策略又会引起运行状态和就绪状态之间的切换。引起进程状态转换的具体原因如下:

- 运行态 → 等待态: 等待使用资源或某事件发生, 如等待外设传输、等待人工干预。
- 等待态 → 就绪态: 资源得到满足或某事件已经发生, 如外设传输结束; 人工干预完成。
- 运行态 → 就绪态: 运行时间片到, 或出现有更高优先权进程。
- 就绪态 → 运行态: CPU 空闲时被调度选中一个就绪进程执行。

2. 五态模型

在一个实际的系统里, 进程的状态及其转换比上节叙述的复杂一些, 例如, 引入专门的新建态(new)和终止态(exit)。图2-16给出了进程五态模型及其转换。

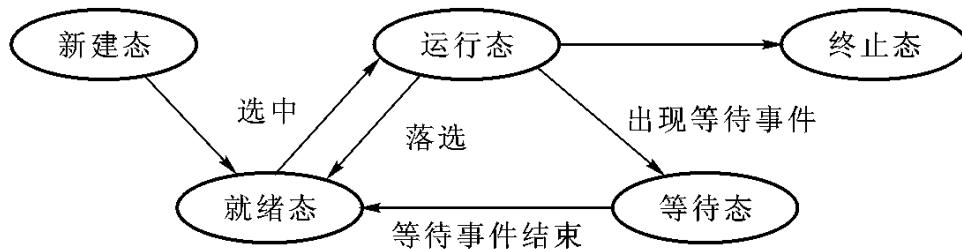


图2-16 进程五态模型及转换

引入新建态和终止态对于进程管理来说是非常有用的。新建态对应于进程刚刚被创建的状态。创建一个进程要通过两个步骤:首先是为一个新进程创建必要的管理信息,然后让该进程进入就绪态。此时进程将处于新建态,它并没有被提交执行,而是在等待操作系统完成创建进程的必要操作。必须指出的是,操作系统有时将根据系统性能或主存容量的限制推迟新建态进程的提交。

类似地,进程的终止也要通过两个步骤:首先是等待操作系统进行善后,然后退出主存。当一个进程到达了自然结束点,或是出现了无法克服的错误,或是被操作系统所终结,或是被其他有终止权的进程所终结,它将进入终止态。进入终止态的进程以后不再执行,但依然保留在操作系统中等待善后。一旦其他进程完成了对终止态进程的信息抽取之后,操作系统将删除该进程。引起进程状态转换的具体原因如下:

- NULL → 新建态。执行一个程序,创建一个子进程。
- 新建态 → 就绪态。当操作系统完成了进程创建的必要操作,并且当前系统的性能和内存的容量均允许。
- 运行态 → 终止态。当一个进程到达了自然结束点,或是出现了无法克服的错误,或是被操作系统所终结,或是被其他有终止权的进程所终结。
- 终止态 → NULL。完成善后操作。
- 就绪态 → 终止态。未在状态转换图中显示,但某些操作系统允许父进程终结子进程。

3. 具有挂起功能系统的进程状态及其转换

到目前为止,或多或少总是假设所有的进程都在内存中。事实上,可能出现这样一些情况,由于进程的不断创建,系统的资源特别如内存资源已经不能满足进程运行的要求。这个时候就必须把某些进程挂起(suspend),对换到磁盘镜像区中,释放它所占有的某些资源,暂

时不参与低级调度,起到平滑系统操作负荷的目的。也可能系统出现故障,需要暂时挂起一些进程,以便故障消除后,再解除挂起恢复这些进程运行。用户调试程序过程中,也可能请求挂起他的进程,以便进行某种检查和修改。总之,引起进程挂起的原因是多样的,主要有:

- 系统中的进程均处于等待状态,处理器空闲,此时需要把一些等待进程对换出去,以腾出足够的内存装入就绪进程运行。
- 进程竞争资源,导致系统资源不足,负荷过重,此时需要挂起部分进程以调整系统负荷,保证系统的实时性或让系统正常运行。
- 把一些定期执行的进程(如审计程序、监控程序、记账程序)对换出去,以减轻系统负荷。
- 用户要求挂起自己的进程,以便根据中间执行情况和中间结果进行某些调试、检查和改正。
- 父进程要求挂起自己的后代进程,以进行某些检查和改正。
- 操作系统需要挂起某些进程,检查运行中资源使用情况,以改善系统性能;或当系统出现故障或某些功能受到破坏时,需要挂起某些进程以排除故障。图 2-17 给出了具有挂

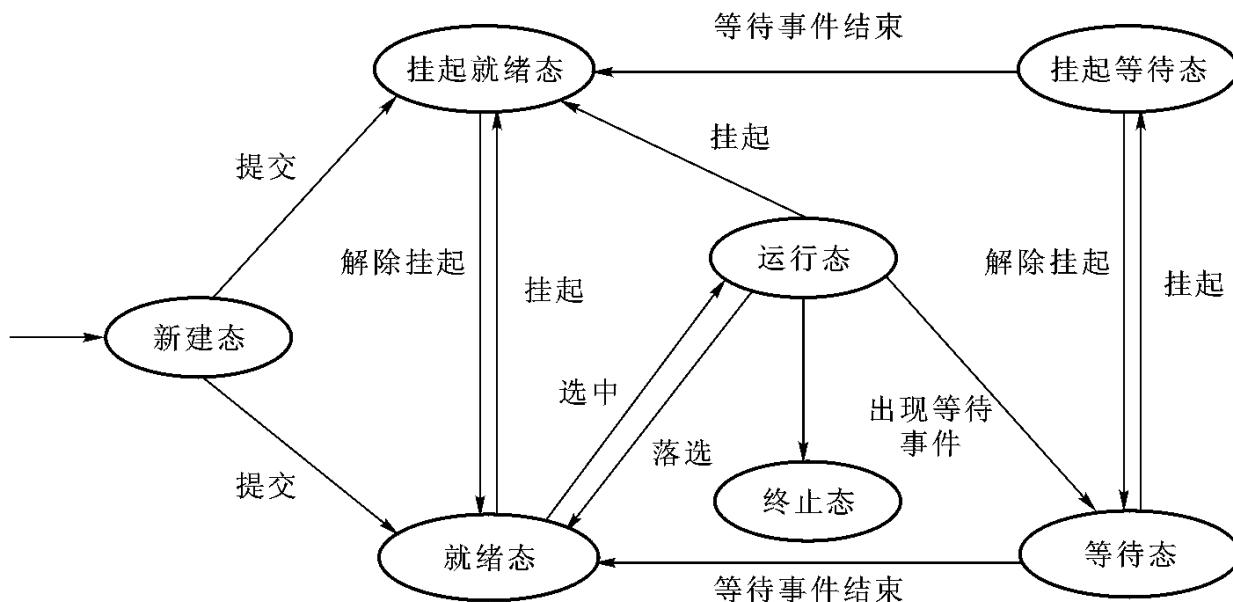


图 2-17 具有挂起功能系统的进程状态

起进程功能的系统中的进程状态。在此类系统中,进程增加了两个新状态:挂起就绪态(ready suspend)和挂起等待态(blocked suspend)。挂起就绪态表明了进程具备运行条件但目前在辅存储器中,只有当它被对换到主存才能被调度执行。挂起等待态则表明了进程正在等待某一个事件且在辅存储器中。

引起进程状态转换的具体原因如下:

- 等待态→挂起等待态:如果当前不存在就绪进程,那么,至少有一个等待态进程将

被对换出去成为挂起等待态；操作系统根据当前资源状况和性能要求，可以决定把等待态进程对换出去成为挂起等待态。

- 挂起等待态→挂起就绪态：引起进程等待的事件发生之后，相应的挂起等待态进程将转换为挂起就绪态。

- 挂起就绪态→就绪态：当内存中没有就绪态进程，或者挂起就绪态进程具有比就绪态进程更高的优先级的，系统将把挂起就绪态进程调回主存并转换成就绪态。

- 就绪态→挂起就绪态：操作系统根据当前资源状况和性能要求，也可以决定把就绪态进程对换出去成为挂起就绪态。

- 挂起等待态→等待态：当一个进程等待一个事件时，原则上不需要把它调入内存。但是在下面一种情况下，这一状态变化是可能的。当一个进程退出后，主存已经有了足够的自由空间，而某个挂起等待态进程具有较高的优先级并且操作系统已经得知导致它阻塞的事件即将结束，便可能发生这一状态变化。

- 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的等待事件结束后，它需要抢占 CPU，而此时主存空间不够，从而可能导致正在运行的进程转化为挂起就绪态。另外，处于运行态的进程也可以自己挂起自己。

- 新建态→挂起就绪态：考虑到系统当前资源状况和性能要求，可以决定新建的进程将被对换出去成为挂起就绪态。

不难看出，可以把一个挂起进程等同于不在主存的进程，因此，挂起的进程将不参与低级调度直到它们被对换进主存。一个挂起进程具有如下特征：

- 该进程不能立即被执行。
- 挂起进程可能会等待一个事件，但所等待的事件是独立于挂起条件的，事件结束并不能导致进程具备执行条件。
- 进程进入挂起状态是由于操作系统、父进程或进程本身阻止它的运行。
- 结束进程挂起状态的命令只能通过操作系统或父进程发出。

2.3.3 进程的描述

1. 操作系统的控制结构

操作系统作为资源管理和分配程序，其本质任务是自动控制程序的执行，并满足进程执行过程中提出的资源使用要求。因此，操作系统的核心控制结构是进程结构，资源管理的数据结构将围绕进程结构展开。

本节在研究进程的控制结构之前，首先介绍一下操作系统的控制结构。为了有效地管理进程和资源，操作系统必须掌握每一个进程和资源的当前状态。从效率出发，操作系统的

控制结构及其管理方式必须是简明有效的,通常是通过构造一组表来管理和维护进程和每一类资源的信息。操作系统的控制表分为四类:进程控制表,存储控制表,I/O 控制表和文件控制表。

- 进程控制表用来管理进程及其相关信息。
- 存储控制表用来管理一级(主)存储器和二级(辅)存储器,主要内容包括:主存储器的分配信息,二级存储器的分配信息,存储保护和分区共享信息,虚拟存储器管理信息。
- I/O 控制表用来管理计算机系统的 I/O 设备和通道,主要内容包括:I/O 设备和通道是否可用,I/O 设备和通道的分配信息,I/O 操作的状态和进展,I/O 操作传输数据所在的主存区。
- 文件控制表用来管理文件,主要内容包括:被打开文件的信息,文件在主存储器和二级存储器中的位置信息,被打开文件的状态和其他属性信息。

图 2-18 给出了操作系统控制表的通用结构,虽然具体操作系统的实现各有特色,但其控制表的基本结构是类似的。值得指出的是,图中仅仅是操作系统控制结构的示意图,在实际的操作系统中,其具体形式要复杂的多,并且这四类控制表也是交叉引用的。

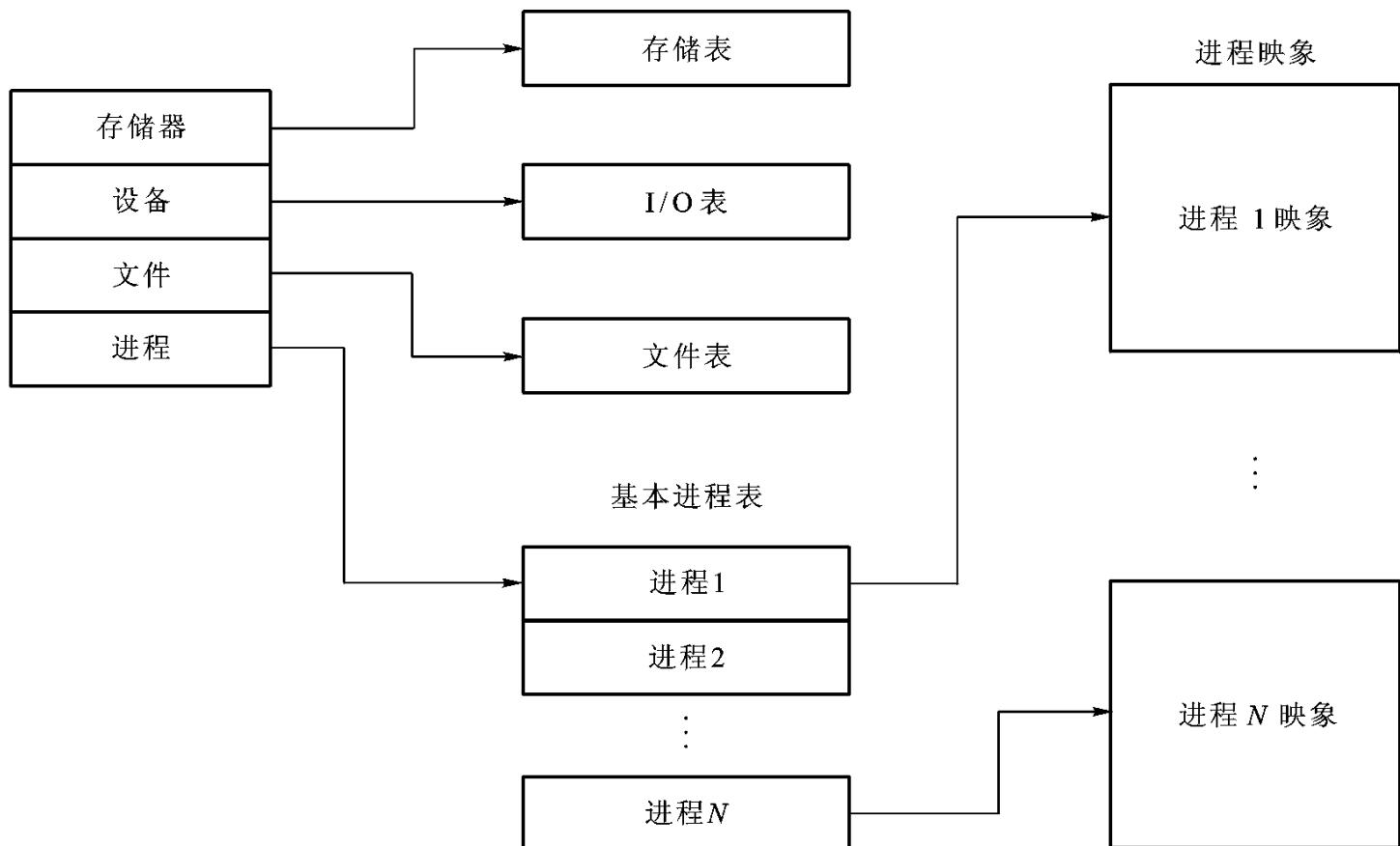


图 2-18 操作系统控制表的通用结构

2. 进程的内存映像

当一个程序进入计算机的主存储器进行计算就构成了进程, 主存储器中的进程到底是如何组成的? 操作系统中把进程物理实体和支持进程运行的环境合称为进程上下文 (process context)。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, 因此, 进程的运行被认为是在进程的上下文中执行的。在操作系统中, 进程上下文包括三个组成部分:

- 用户级上下文(user-level context)。由用户进程的程序块、用户数据块(含共享数据块)和用户堆栈组成的进程地址空间。

- 系统级上下文(system-level context)。包括进程控制块、内存管理信息、进程环境块, 以及系统堆栈等组成的进程地址空间。

- 寄存器上下文(register context)。由程序状态字寄存器、各类控制寄存器、地址寄存器、通用寄存器、用户栈指针等组成。进程的内存映像可以很好地说明进程的组成。简单的说, 一个进程映像(Process Image)包括:

- 进程程序块, 即被执行的程序, 规定了进程一次运行应完成的功能。通常它是纯代码, 作为一种系统资源可被多个进程共享。

- 进程数据块, 即程序运行时加工处理对象, 包括全局变量、局部变量和常量等的存放区以及开辟的工作区, 常常为一个进程专用。• 系统/用户堆栈, 每一个进程都将捆绑一个系统/用户堆栈, 用来解决过程调用或系统调用时的信息存储和参数传递。

- 进程控制块, 每一个进程都将捆绑一个进程控制块, 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。

可见, 每个进程有四个要素组成: 控制块、程序块、数据块和堆栈。例如, 用户进程在虚拟内存中的组织如图 2-19 所示。

3. 进程控制块

每一个进程都有一个也只有一个进程控制块 PCB (Process Control Block), 进程控制块是操作系统用于记录和刻画进程状态及有关信息的数据结构, 也是操作系统掌握进程的惟一资料结构, 是操作系统控制和管理进程的主要依据。它包括了进程执行时的情况, 以及进程让出处理器后所处的状态、断点等信息。一般说, 进程控制块包含三类信息:

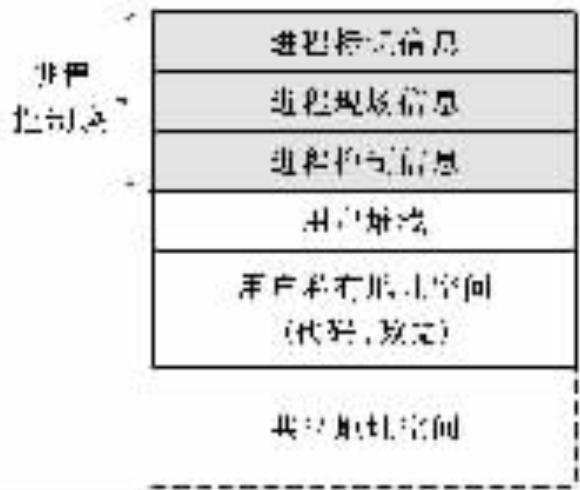


图 2.19 用户进程在虚拟内存中的组织

- 标识信息。用于惟一地标识一个进程,常常分为由用户使用的外部标识符和被系统使用的内部标识号。几乎所有操作系统中进程都被赋予一个惟一的、内部使用的数值型的进程号,操作系统的其他控制表可以通过进程号来交叉引用进程控制表。常用的标识信息包括进程标识符、父进程的标识符、用户进程名、用户组名等。
- 现场信息。用于保留一个进程在运行时存放在处理器现场中的各种信息,任何一个进程在让出处理器时必须把此时的处理器现场信息保存到进程控制块中,而当该进程重新恢复运行时也应恢复处理器现场。常用的现场信息包括:通用寄存器的内容、控制寄存器(如 PSW 寄存器)的内容、用户堆栈指针、系统堆栈指针等。
- 控制信息。用于管理和调度一个进程。常用的控制信息包括:(1)进程调度相关信息,如进程状态、等待事件和等待原因、进程优先级、队列指引元等;(2)进程组成信息,如正文段指针、数据段指针;(3)进程间通信信息,如消息队列指针、信号量等互斥和同步机制;(4)进程在辅存储器内的地址;(5)CPU 资源的占用和使用信息,如时间片余量、进程已占用 CPU 的时间、进程已执行的时间总和、记账信息;(6)进程特权信息,如在内存访问权限和处理器状态方面的特权;(7)资源清单,包括进程所需全部资源、已经分得的资源,如主存资源、I/O 设备、打开文件表等。

进程控制块是操作系统中最为重要的数据结构,每个进程控制块包含了操作系统管理所需的所有进程信息,进程控制块的集合事实上定义了一个操作系统的当前状态。进程控制块的使用权或修改权均属于操作系统程序,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。当系统创建一个进程时,就为它建立一个 PCB,当进程执行结束被撤销时,便回收它占用的 PCB。操作系统是根据 PCB 来对并发执行的进程进行控制和管理的,借助于进程控制块 PCB,进程才能被调度执行。

4. 进程队列及其管理

并发系统中同时存在许多进程,有的处于就绪态,有的处于等待态,等待原因各不相同。进程的主要特征是由 PCB 来刻画的。为了便于管理和调度,常常把各个进程的 PCB 用某种方法组织起来,用得较多的是用队列来组织 PCB。下面先介绍这种方法。(一般说来,把处于同一状态(例如就绪态)的所有进程控制块链接在一起的数据结构称为进程队列(process queues),简称队列。)同一状态进程的 PCB 既可按先来先到的原则排成队列,也可以按优先数或其他原则排成队列。对于等待态的进程队列可以进一步细分,每一个进程按等待的原因进入相应的等待队列。例如,如果一个进程要求使用某个设备,而该设备已经被占用时,此进程就链接到与该设备相关的等待态队列中去。

在一个队列中,链接进程控制块的方法可以是多样的,常用的是单向链接和双向链接。单向链接方法是在每个进程控制块内设置一个队列指引元,它指出在队列中跟随着它的下一个进程的进程控制块内队列指引元的位置。双向链接方法是在每个进程控制块内设置两

个指引元,其中一个指出队列中该进程的上一个进程的进程控制块内队列指引元的位置,另一个指出队列中该进程的下一个进程的进程控制块的队列指引元的位置。为了标志和识别一个队列,系统为每一个队列设置一个队列标志,单向链接时,队列标志指引元指向队列中第一个进程的队列指引元的位置。双向链接时,队列标志的后向指引元指向队列中第一个进程的后向队列指引元的位置;队列标志的前向指引元指向队列中最后一个进程的前向队列指引元的位置。这两种链接方式如图 2-20(a)、(b) 所示。

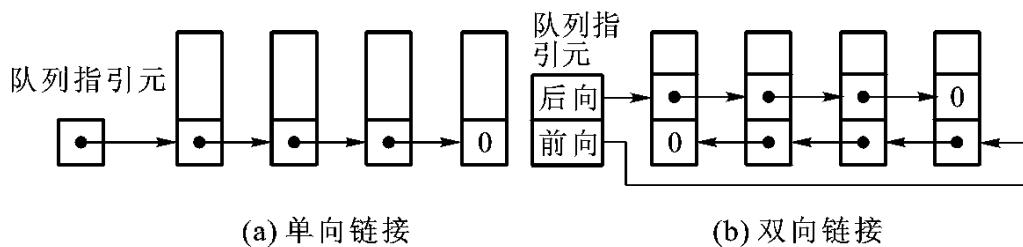


图 2-20 进程控制块的链接

当发生的某个事件使一个进程的状态发生变化时,这个进程就要退出所在的某个队列而排入到另一个队列中去。一个进程从一个所在的队列中退出的事件称为出队,相反,一个进程排入到一个指定的队列中的事件称为入队。处理器调度中负责入队和出队工作的功能模块称为队列管理模块,简称队列管理。图 2-21 给出了操作系统的队列管理和状态转换示意图。

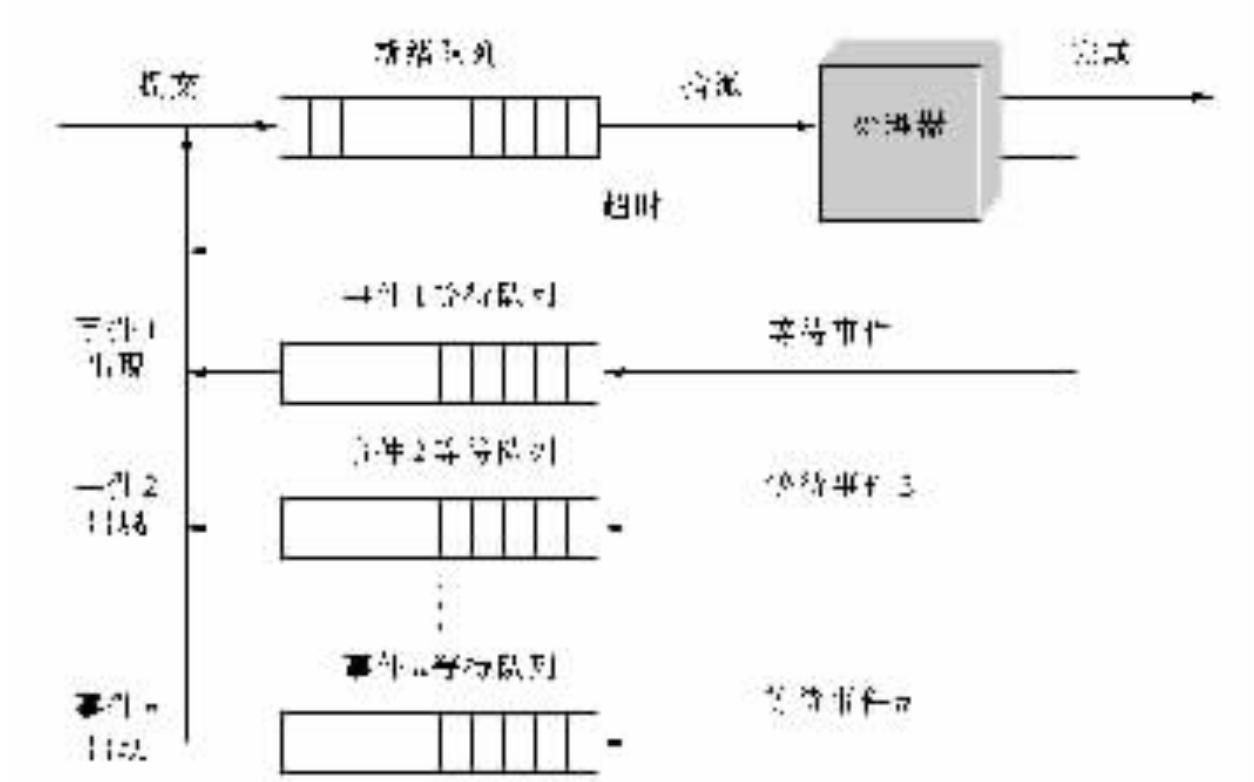


图 2-21 操作系统的队列管理和状态转换示意图

下面来考虑队列管理是如何将一个进程从某个队列中移出而加入到另一个队列中去。假设采用双向链接,如图 2-20(b)所示,每个进程有两个队列指引元,用来指示该进程在队列中的位置,其中第一个队列指引元为后向指引元,第二个为前向指引元。根据一个进程排在一个队列中的情况,前(后)向指引元的内容规定如下:

- 情况 1。它是队列之首。此时,它的前向指引元为 0,而后向指引元指出它的下一个进程的后向指引元位置。
- 情况 2。它是队列之尾。此时,它的后向指引元为 0,而它的前向指引元指出它的上一个进程的前向指引元位置。
- 情况 3。它的前后均有进程。此时,后(前)向指引元指出它的下(上)一个进程的后(前)向指引元位置。

现在来考虑一个进程的出队。假设进程 Q 在某个队列中,它的前面是进程 P,后面是进程 R。进程 Q 出队过程为:把 Q 的前向指引元的内容送到 R 的前向指引元中,把 Q 的后向指引元的内容送到 P 的后向指引元中。于是 P 的后向指引元指向 R,而 R 的前向指引元指向 P,Q 就从队列中退出,通常还把 Q 的后、前向队列指引元置为自身的地址。类似的可实现队首进程,队尾进程的出队,或者让一个进程加入到队列中去。

此外,用来组织 PCB 的方法为表格法。把所有进程的 PCB 都组织在一个线性表中,进程调度时需要查找整个 PCB 表;也可以把相同状态进程的 PCB 组织在一个线性表中,系统有多个线性表,这样可缩短查表时间。

2.3.4 进程切换与模式切换

中断是激活操作系统的惟一方法,它暂时中止当前运行进程的执行,把处理器切换到操作系统的控制之下。而当操作系统获得了处理器的控制权之后,它就可以实现进程切换,所以,进程切换必定在核心态而不是在用户态下发生。操作系统在处理发生的中断事件或系统调用过程中可能会导致被阻塞了的高优先级进程变为就绪,或处理时钟中断期的发现正在运行的进程时间片耗尽,或当前运行进程执行了一条阻塞型 I/O 指令等等,均有可能引发内核进行进程上下文切换。由于一个进程让出处理器时,其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系统栈的一个上下文层。当内核处理中断返回,或一个进程完成其系统调用返回用户态,或内核进行上下文切换时,内核就从系统栈弹出一个上下文层(context layer)。因此,上下文的切换总会引起上下文的压入和弹出堆栈。内核在四种情况下允许发生上下文切换:

- (1) 当进程进入等待态。
- (2) 当进程完成其系统调用返回用户态,但不是最有资格获得 CPU 时。
- (3) 当内核完成中断处理,进程返回用户态但不是最有资格获得 CPU 时。

(4) 当进程执行结束时。

做一次进程上下文切换时,即保存老进程的状态而装入被保护了的新进程的状态,以便新进程运行。进程切换的步骤如下:

- 保存被中断进程的处理器现场信息。
- 修改被中断进程的进程控制块的有关信息,如进程状态等。
- 把被中断进程的进程控制块加入有关队列。
- 选择下一个占有处理器运行的进程。
- 修改被选中进程的进程控制块的有关信息。
- 根据被选中进程设置操作系统用到的地址转换和存储保护信息。
- 根据被选中进程的信息来恢复处理器现场。

从上面介绍的切换工作可以看出,当进行上下文切换时,内核需要保存足够的信息,以便将来适当时机能够切换回原进程,并恢复它继续执行。类似地,当从用户态转到核心态时,内核保留足够信息以便后来能返回到用户态,并让进程从它的断点继续执行。用户态到核心态或者核心态到用户态的转变是 CPU 模式的改变,而不是进程上下文切换。为了进一步说明进程的上下文切换,下面来讨论模式切换。当中断发生的时候,暂时中断正在执行的用户进程,把进程从用户状态切换到内核状态,去执行操作系统例行程序以获得服务,这就是一次模式切换。注意,此时该进程仍在自己的上下文中执行,仅仅模式变了。内核在被中断了的进程的上下文中对这个中断事件作处理,即使该中断事件可能不是此进程引起的。另一点要注意的是,被中断的进程可以是正在用户态下执行的,也可以是正在核心态下执行的,内核都要保留足够信息以便在后来能恢复被中断了的进程执行。内核在核心态下对中断事件进行处理时,决不会再产生或调度一个特殊进程来处理中断事件。模式切换的步骤如下:

- 保存被中断进程的处理器现场信息。
- 根据中断号设置程序计数器。
- 把用户状态切换到内核状态,以便执行中断处理程序。

注意模式切换不同于进程切换,它并不一定引起进程状态的变化,在大多数操作系统中,它也不一定引起进程的切换,在完成了中断调用之后,完全可以再通过一次逆向的模式切换来继续执行被中断了的用户进程。

显然,有效合理地使用模式切换和进程切换有利于操作系统效率和安全性的提高。为此,大多数现代操作系统存在两种进程:系统进程和用户进程。它们并不是指两个具体的进程实体,而是指一个进程的两个侧面,系统进程是在核心态下执行操作系统代码的进程,用户进程在用户态下执行用户程序的进程。用户进程因中断或系统调用进入内核态,系统进程就开始执行。这两个进程(用户进程和系统进程)使用同一个 PCB,所以实质上是一个进程实体。但是,这两个进程所执行的程序不同,映射到不同物理地址空间、使用不同堆栈。

一个系统进程的地址空间中包含所有的系统核心程序和各进程的进程数据区,所以,各进程的系统进程除数据区不同外,其余部分全相同,但各进程的用户进程部分则各不相同。

图 2-22 清楚地给出了一个进程(未对换出主存)的生命周期中,可能出现的进程切换和模式切换的示意。其中:

- 用户态运行表示进程在用户模式下执行;
- 核心态运行表示进程在内核模式执行;
- 就绪状态表示进程处于就绪态,但未正在执行;
- 等待状态表示进程正处于等待态,由于发生某个事件(如等 I/O 完成)而进入此状态。

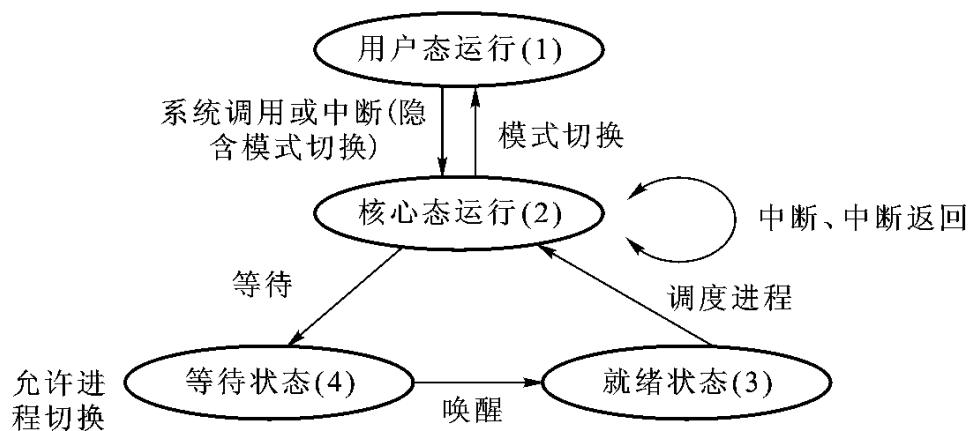


图 2-22 进程上下文切换和模式切换

上面的状态给出了描述进程的静态观点,事实上进程状态是在连续动态地转换的,状态图中若有箭头指向边,这种状态转换是合法的,通常进程完成且只完成一个合法状态转换。任何时刻一个处理器上仅能执行一个进程,所以,至多只有一个进程可以处在状态(1)或状态(2),这两个状态相当于用户模式和内核模式。当系统调用或中断发生时,进程通过一个模式切换从用户态运行转化为核心态运行,在核心态下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,当处理完中断或系统调用后,可以通过一个模式切换转回用户模式继续运行,也可以根据具体情况使进程等待一个事件(状态(4)),只有在此时内核才会允许发生进程切换,使原先运行的进程让出处理器。当等待事件完成后,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,进程会重新占有处理器运行。在多道程序设计系统中,有许多进程在并发执行,如果两个以上进程同时执行系统调用,并要求在内核模式下执行,则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中断的响应,就能保证数据的一致性和完整性。

2.3.5 进程的控制

处理器管理的一个主要工作是对进程的控制,对进程的控制包括: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。这些控制和管理功能是由操作系统中的原语来实现的。原语(Primitive)是在管态下执行、完成系统特定功能的过程。原语和机器指令类似,其特点是执行过程中不允许被中断,是一个不可分割的基本单位,原语的执行是顺序的而不可能是并发的。系统对进程的控制如不使用原语,就会造成其状态的不确定性,从而达不到进程控制的目的。一种原语的实现方法是以系统调用方式提供原语接口,且采用屏蔽中断的方式来实现原语功能,以保证原语操作不被打断的特性。原语和系统调用都使用访管指令实现,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器实现;原语不可中断,而系统调用执行时允许被中断,甚至有些操作系统中系统进程或系统服务器干脆在用户态运行。通常情况下,原语提供给系统进程或系统服务器使用(反之决不会形成调用关系),系统进程或系统服务器提供系统调用给系统程序(和用户)使用,而系统程序提供高层功能给用户使用,例如,语言编译程序提供语句供用户解决问题。下面介绍部分进程控制原语。

1. 进程的创建

每一个进程都有生命期,即从创建到消亡的时间周期。当操作系统为一个程序构造一个进程控制块并分配地址空间之后,就创建了一个进程。进程的创建来源于以下四个事件:

- 提交一个批处理作业。
- 在终端上一个交互式作业登录。
- 操作系统创建一个服务进程。
- 存在的进程创建新的进程。

下面来讨论最后一种情况,当一个用户作业被接受进入系统时,系统需要创建一个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,也可能要创建一个或多个子进程或系统进程来为之服务。例如,当一个用户进程要求读卡片上的一段数据时,可能创建一个卡片输入机管理进程。有的系统把“创建”用父子进程关系来表示,当一个进程创建另一个进程时,生成进程称为父进程(parent process),被生成进程称为子进程(child process)。一个父进程可以创建多个子进程,从而形成树形结构,如 UNIX/Linux 就是这样。当父进程创建了子进程后,子进程就继承了父进程的全部资源,父子进程常常要相互通信和协作,子进程结束时,又必须要求父进程对其作某些善后处理。

进程的创建过程描述如下:

- 在主进程表中增加一项,并从 PCB 池中申请一个空白 PCB(在 UNIX/Linux 中,分配一个 task_struct)。
 - 为新进程的进程映像分配地址空间。对于进程创建操作还需要传递环境变量,构造共享地址空间。
 - 为新进程分配资源,除内存空间外,还有其他各种资源。

- 查找辅助存储器,找到进程正文段并装入到进程地址空间的正文区。
- 初始化进程控制块(如状态、PSW、栈等),为新进程分配一个惟一的进程标识符,
- 把进程加入某一就绪进程队列(这时子进程就绪),或直接将进程投入运行(这时父进程就绪)。
- 通知操作系统的某些模块,如记账程序、性能监控程序。

Linux 保留了传统的 fork() 创建子进程的方法,同时又增加了一种新的称之为克隆(clone)的创建子进程的方法。使用 fork() 创建子进程后,子进程虽是其祖先进程的拷贝,它们所有的变量都有相同的值,打开的文件也相同,但是它们并不共享任何内容,如果父进程改变了一个变量的值,子进程将看不到这个变化,反之亦然。克隆允许定义父进程和子进程所应该共享的内容。如果不定义内容共享,则 clone 相当于 fork,反之则子进程就可以和父进程共享定义的内容,这和线程的实现思路类似,因而,Linux 中认为线程就是共享上下文的进程。Linux 中实现 fork() 和 clone() 的主要工作如下:在生成克隆进程时,对于 task_struct 结构中的 fs 指针、files 指针、sig 指针和 mm 指针实际上并不进行进程拷贝操作,而仅将当前这些数据结构成员中的计数器 count 值加 1。这样,只有当父进程中相关数据结构成员的 count 为 0 时,才会释放这些数据结构成员占用的内存空间。此外,父子进程共享存储区时,由于采用了 copy on write 的策略,即仅当父进程或子进程对虚存进行写操作时,才给子进程的指针所指向的数据结构分配内存,并将父进程的 mm 指针所指向的数据结构内容拷贝到子进程的 mm 指针上。

在许多操作系统中,都有一个最多进程数的限制,有的还设置了参数,可加以调整。早期的 UNIX 允许系统内最多创建几十个进程。商业化操作系统 Solaris 中,可在启动时根据发现的内存容量自动调整参数。在 Linux2.4 中,最多进程数也是个运行时可调参数,按缺省值设置为: size-of-memory-in-the-system/kernel-stack-size/2。假如你的机器有 512 MB 内存,则缺省的可用进程的上限为 $512 \times 1\,024 \times 1\,024 / 8\,192 / 2 = 32\,768$ 。看起来好像很多,但对于一个 Linux 的数据库服务器来说,这是一个合理的数目。

2. 进程的阻塞和唤醒

进程的阻塞是指使一个进程让出处理器,去等待一个事件,如等待资源、等待 I/O 完成、等待一个事件发生等。通常,进程自己调用阻塞原语阻塞自己,所以,阻塞是进程自主行为,是一个同步事件。当一个等待事件结束时会产生一个中断,从而激活操作系统,在系统的控制之下将被阻塞的进程唤醒,如 I/O 操作结束、某个资源可用或期待事件出现。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:

- 停止进程执行,保存现场信息到 PCB。
- 修改进程控制块的有关内容,如进程状态由运行改为等待等。

- 把修改状态后的进程控制块加入相应等待进程队列。
- 接着便应转入进程调度程序去调度其他进程运行。

进程唤醒的步骤如下：

- 从相应的等待进程队列中取出进程控制块。
- 修改进程控制块的有关信息,如进程状态改为就绪。
- 把修改后的进程控制块加入有关就绪进程队列。

阻塞原语和唤醒原语的作用正好相反,如果一个进程因某种事件调用阻塞原语阻塞自己,则当事件发生后,必须在与其相关的另一进程调用唤醒原语来唤醒阻塞进程,否则,被阻塞进程将因不能被唤醒而处于永远阻塞状态。

在 UNIX/Linux 中,与进程的阻塞和唤醒相关的原语主要有:sleep(暂停)、pause(暂停并等信号)、wait(等待子进程暂停或终止)和 kill(发信号)。在 Windows 2000/XP 中,处理器的调度对象为线程,用户可以通过系统调用 SuspendThread 和 ResumeThread 来挂起或激活线程。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(suspend count),挂起操作使该计数器加 1,激活操作使该计数器减 1。当挂起计数从 0 变为 1 时,线程进入阻塞状态;当挂起计数从 1 变为 0 时,线程被唤醒并恢复运行。

3. 进程的撤销

一个进程完成了特定的工作或出现了严重的异常后,操作系统则收回它占有的地址空间和进程控制块,此时就说撤销了一个进程。进程撤销可以分正常和非正常撤销,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与异常。下面罗列出进程撤销的主要原因:

- 进程正常运行结束。
- 进程执行了非法指令。
- 进程在常态下执行了特权指令。
- 进程运行时间超越了分配给它的最大时间配额。
- 进程等待时间超越了所设定的最大等待时间。
- 进程申请的内存超过了系统所能提供的最大量。
- 越界错误。
- 对共享内存区的非法使用。
- 算术错误,如除零和操作数溢出。
- 严重的输入输出故障。
- 操作员或操作系统干预。
- 父进程撤销其子进程。
- 父进程撤销,因而其所有子进程被撤销。

- 操作系统终止。

一旦发生了上述事件, 系统或有关进程将调用撤销原语终止进程或子进程。具体步骤如下:

- 根据撤销进程标识号, 从相应队列中找到它的 PCB;
- 将该进程拥有的资源归还给父进程或操作系统;
- 若该进程拥有子进程, 应先撤销它的所有子孙进程, 以防它们脱离控制;
- 撤销进程出队, 将它的 PCB 归还到 PCB 池。

4. 进程的挂起和激活

前一小节已经讨论了进程挂起的概念, 当出现了引起挂起的事件时系统或进程利用挂起原语把指定进程或处于阻塞状态的进程挂起。其执行过程大致如下: 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, 若处于阻塞态, 则修改为挂起阻塞。被挂起进程 PCB 的非常驻部分要交换到磁盘对换区。

当系统资源尤其是内存资源充裕或进程请求激活指定进程时, 系统或有关进程会调用激活原语把指定进程激活。该原语所做的主要工作是: 把进程 PCB 非常驻部分调进内存, 然后修改它的状态, 挂起等待态改为等待态, 挂起就绪态改为就绪态, 并分别排入相应队列中。

注意, 挂起原语既可由进程自己也可由其他进程调用, 但激活原语却只能由其他进程调用。

2.3.6 实例研究: UNIX SVR4 进程管理

UNIX SVR4 进程管理的实现采用基于用户进程的实现模型, 大多数操作系统功能在用户进程的环境中执行, 因此, 它需要在用户模式和内核模式间的切换。UNIX SVR4 允许两类进程: 用户进程和系统进程。系统进程在内核模式下执行, 完成操作系统的一些重要功能, 如内存分配和进程对换。而用户进程在用户模式下执行用户程序, 在内核模式下执行操作系统的代码, 系统调用, 中断和异常将引起模式切换。

1. UNIX SVR4 的进程状态

图 2-23 给出了 UNIX SVR4 进程状态及其转换, 其中包括两种运行状态, 分别为核心运行态 (kernel running) 和用户运行态 (user running); 两种等待状态, 分别对应了在内存 (asleep in memory) 或被换出内存 (sleeping, swapped); 三种就绪状态, (preempted)、(ready to run, in memory) 和 (ready to run, swapped), 后者被换出内存, 再换入内存前不能被调度执行。preempted 和 ready to run, in memory 本质上是同一种状态, 也被组织在同一个就绪进程队列中, 因此在图中用虚线连接在一起。但是 preempted 态和 ready to run, in memory 态也是有区别的, 当一个 kernel running 态进程完成了相应的操作准备切换到 user running 态时, 出现了更高优先级的就绪进程, 那么, 原来那个进程将转化为 preempted 态。

具体的进程状态包括：

- user running：在用户模式下运行。
- kernel running：在内核模式下运行。
- preempted：当一个进程从内核模式返回用户模式，发生了进程切换后处于的就绪状态。
- ready to run, in memory：就绪状态，在内存。
- asleep in memory：等待状态，在内存。
- ready to run, swapped：就绪状态，被对换出内存，不能被调度执行。
- sleeping, swapped：睡眠状态，被对换出内存。
- created：新建状态。
- zombie：终止状态：进程已不存在，留下状态码和有关信息使父进程收集。

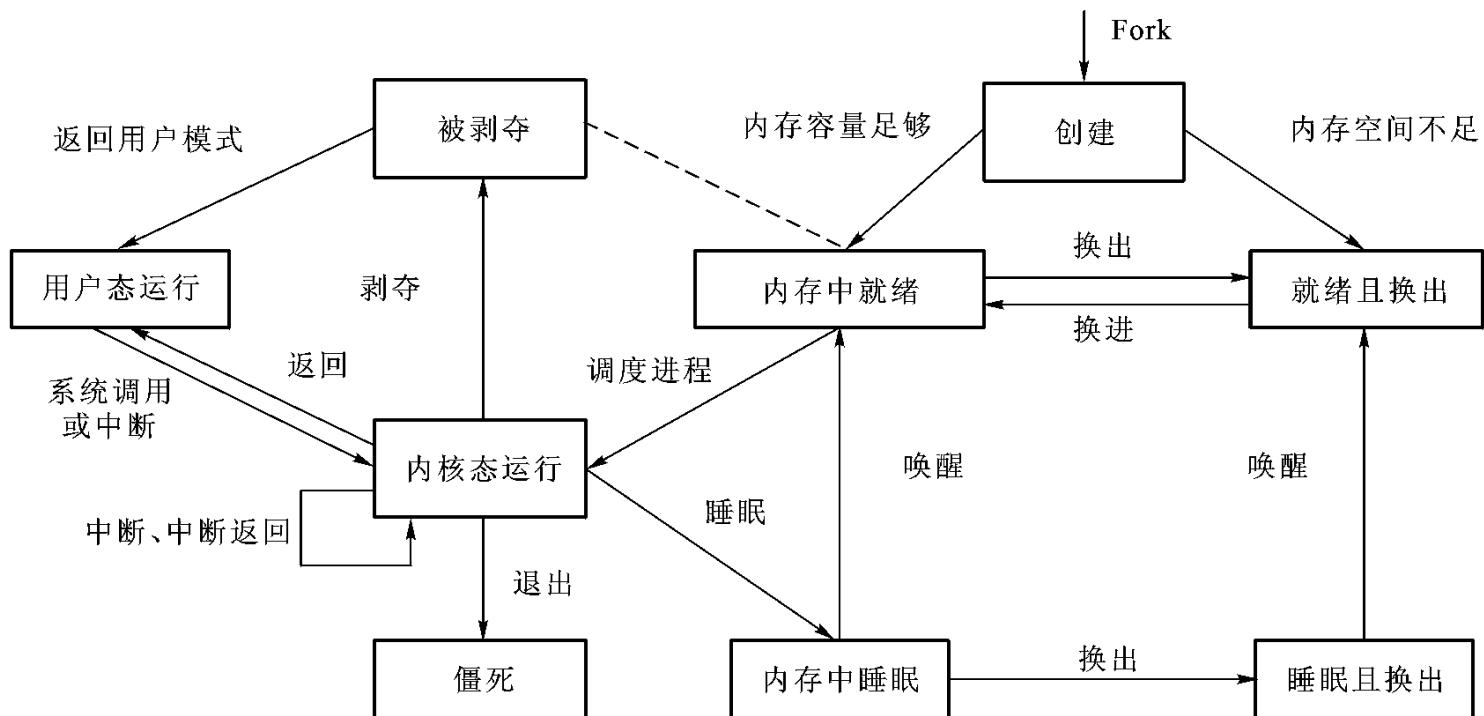


图 2-23 Unix SVR4 进程状态及其转换

UNIX 操作系统中有两个固定的进程：0 号进程 (proc[0]) 是 swap 进程，在系统自举时被创建，它的功能是执行调度，把外存的准备就绪的进程调入内存，该进程永不消亡，要么工作要么处于睡眠状态；1 号进程 (proc[1]) 是 init 进程，由 0 号进程孵化而创建。系统中的其他进程都是 1 号进程的子进程，因而，它是除了 0 号进程外所有进程的祖先。当一个交互式用户登录到系统中时，1 号进程为这个用户创建一个用户进程，用户进程在执行具体应用时进一步创建子进程，从而就构成了一棵进程树。

2. UNIX SVR4 的进程描述

UNIX SVR4 的进程结构和组成如图 2-24 所示。它由三部分组成：proc 结构、数据段和

正文段,它们合称为进程映像。UNIX 中把进程定义为映像的执行。其中,PCB 由基本控制块 proc 结构和扩充控制块 user 结构两部分组成。在 proc 结构里存放着关于一个进程的最基本、最必需的信息,因此它常驻内存。在 user 结构里存放着只有进程运行时才用到的数据和状态信息,为了节省内存空间,当进程暂时不在处理器上运行时,就把它放在磁盘上的对换区中,进程的 user 结构总和进程的数据段一起,在主存和磁盘对换区之间换进/换出。

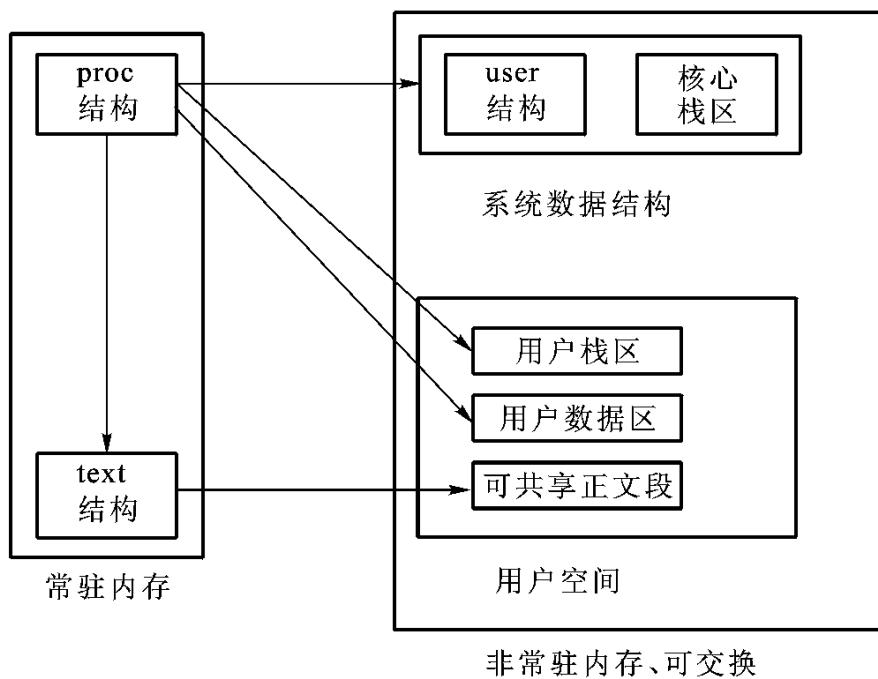


图 2-24 UNIX SVR4 进程结构

系统中维持一张名叫 proc 的结构数组,每个表目为一个 proc 结构,供一个进程使用。创建进程时,在 proc 表中找一个空表目,以建立起相应于该进程的 proc 结构。所有进程的 proc 结构集中形成 proc 结构数组,存放在操作系统的核心数据区中。

proc 结构:存放的信息包括进程标识符、父进程标识符、进程用户标识符、进程状态、等待的事件、指向 user 结构和进程存储区 (text/data/stack) 的指针、软中断信息、有关进程执行时间/核心资源使用/用户设置示警信号的计时器、进程的大小、调度优先数、就绪队列指针等。进程即使不运行,核心也需要访问有关信息,所以 proc 结构常驻内存。

user 结构:存放的信息包括本进程 proc 结构表项的指针、保护现场、本进程正文段/数据区/栈区长度、用户标识、用户组标识、用户打开文件表、当前目录和当前根、计时信息、文件 I/O 参数、限制字段、错误码字段、返回值字段和信号处理数组等。为了节省内存空间,当进程暂时不在处理器上运行时,就把它放在磁盘上的对换区中,进程的 user 结构没有集中存放,而是分散存放在进程的数据段内,在主存和磁盘对换区之间换进/换出。进程的 user 结构和系统栈组成了系统数据区,虽然放在每个进程的数据块内,以便于与其他数据一起换进

换出内存,但却与其他数据不属于同一地址空间。系统数据区是 PCB 的一部分,属核心态空间,用户程序不能访问它。

系统数据结构:是进程系统数据区,它位于数据段的前面,进程 proc 结构中有指针指向这个区域的首址。该区通常有 1 024 个字节,由两块内容组成:最前面的若干字节为进程的扩充控制块 user 结构,剩下的字节为核心栈,当进程运行在核心态时,这里是它的工作区,用来保存过程调用和中断访问时用到的地址和参数。

用户数据区:用于存放程序运行时用到的数据,如果进程运行的程序是非共享的,那么这个程序也放于此地。

用户栈区:当进程运行在用户态时,这里是它的工作区。

text 结构:正文段在磁盘上和主存中的位置和大小、访问正文段的进程数、在主存中访问正文段的进程数、标志信息、地址转换信息。由于共享正文段在进程映像中的特殊性,为了便于对它们的管理,UNIX 系统在内存中设置了一张正文段表。每一个表目都是一个 text 结构,用来记录一个共享正文段的属性(磁盘和主存中的位置、尺寸、共享的进程数、正文段文件节点指针等),有时也把这种结构称为正文段控制(信息)块。这是可以被多个进程共享的可重入程序和常数。如果一个进程的程序是不被共享的,那么,它的映像中就不出现这一部分。若一个进程有共享正文段,那么,当把该进程的非常驻内存部分调入内存时,应该关注共享正文段是否也在内存,如果发现不在内存,则要将它调入。当把该进程的非常驻内存部分调出内存时,同样要关注它的共享正文段目前被共享的情况,只要还有一个别的共享该正文段的进程映像全部在内存,那么,这个共享正文段就不得调出去。如果一个进程有共享正文段,该共享正文段在正文段表里一定有一个 text 结构与之相对应,而在该进程的基本控制块 proc 里,有专门指针指向这一个 text 结构。综上所述,在 UNIX 进程映像的三个组成部分中,proc、user 和 text 这三个数据结构是最为重要的角色。

系统区表和进程区表:用于实现地址转换。系统区表用于记录进程虚拟地址空间的连续区域,包括正文区、数据区、栈区等。这些区是可被共享和保护的独立实体,并允许多个进程共享一个区。为了对区进行管理,系统区表记录了区的类型、大小、状态、位置、引用计数以及相应的文件索引节点指针。系统为每个进程建立一张进程区表 PPRT (Per Process Region Table),由存储管理系统使用。它定义了物理地址与虚拟地址之间的对应关系,还定义了进程对存储区域的访问权限。其中含有正文段、数据段和堆栈的区域表的指针和各区域的逻辑起始地址;区域表中含有该区域属性(正文/数据,可以共享)的信息和页表的指针;而每个页表中含有相应区域的页面在内存的起始地址。

3. UNIX SVR4 的进程创建

UNIX SVR4 通过系统调用 fork() 来创建一个子进程,当父进程调用 pid = fork() 后,操作系统执行下面的操作:

- 为子进程分配一个进程表。
- 为子进程分配一个进程标识符。
- 复制父进程的进程映像,但不复制共享内存区。
- 增加父进程所打开文件的计数,表示新进程也在使用这些文件。
- 把子进程置为就绪状态。
- 返回子进程的标识符给父进程,把 0 值返回给子进程。

以上的操作都在父进程的内核模式下完成。然后,内核的分派程序还应执行以下的操作之一,以完成进程的指派:

- 继续呆在父进程中。此时把进程控制切换到父进程的用户模式,在 fork()点继续向下运行,而子进程进入 Ready to Run 状态。
- 把进程控制传递到子进程,子进程在 fork()点继续向下运行,而父进程进入 Ready to Run 状态。
- 把进程控制传递到其他进程,父进程和子进程进入 ready to run 状态。

用 fork()创建子进程之后,父子进程均执行相同的代码段,即子进程的程序是父进程程序的复制品,父子进程的指令执行点均在 fork()之后的那个语句。那么,如何来区别父子进程呢?事实上,执行 fork()调用后,子进程 pid 的返回值是 0,而父进程 pid 的返回值为非 0 正整数(即子进程惟一内部标识号),这样就可以通过程序控制流判别 pid 的值,在父子进程中各自执行需要的操作。

2.3.7 实例研究:Linux 进程管理

Linux 是一个多用户操作系统,支持多道程序设计、分时处理和软实时处理,为实现上述目标,必须要引入进程的概念。

1. 进程和进程状态

Linux 的进程概念与传统操作系统中的进程概念完全一致,它目前不支持线程概念(Linux 2.2.x 开始,内核支持 SMP 线程化),进程是操作系统调度的最小单位。如图 2-25 所示,Linux 的进程状态共有 6 种:

- TASK_RUNNING:正在运行(已获得 CPU)或准备运行(等待获得 CPU)的进程,由 current 指针指向当前运行的进程,由 run-list 把所有运行态进程链接起来。
- TASK_INTERRUPTIBLE:可中断等待状态。进程处于等待队列中,一旦资源可用时被唤醒,也可以由其他进程通过信号(SIGNAL)或中断唤醒。
- TASK_UNINTERRUPTIBLE:不可中断等待状态。进程处于等待队列中,一旦资源可用时被唤醒,但不可以由其他进程通过信号(SIGNAL)或中断唤醒。

- TASK_ZOMBIE: 进程僵死状态。进程停止运行但是尚未释放 PCB。
- TASK_STOPPED: 进程停止态。可能被特定信号终止, 也可能是受其他进程的跟踪调用而暂时将 CPU 出让给跟踪它的进程。
- TASK_SWAPPING: 页面被交换出内存的进程。

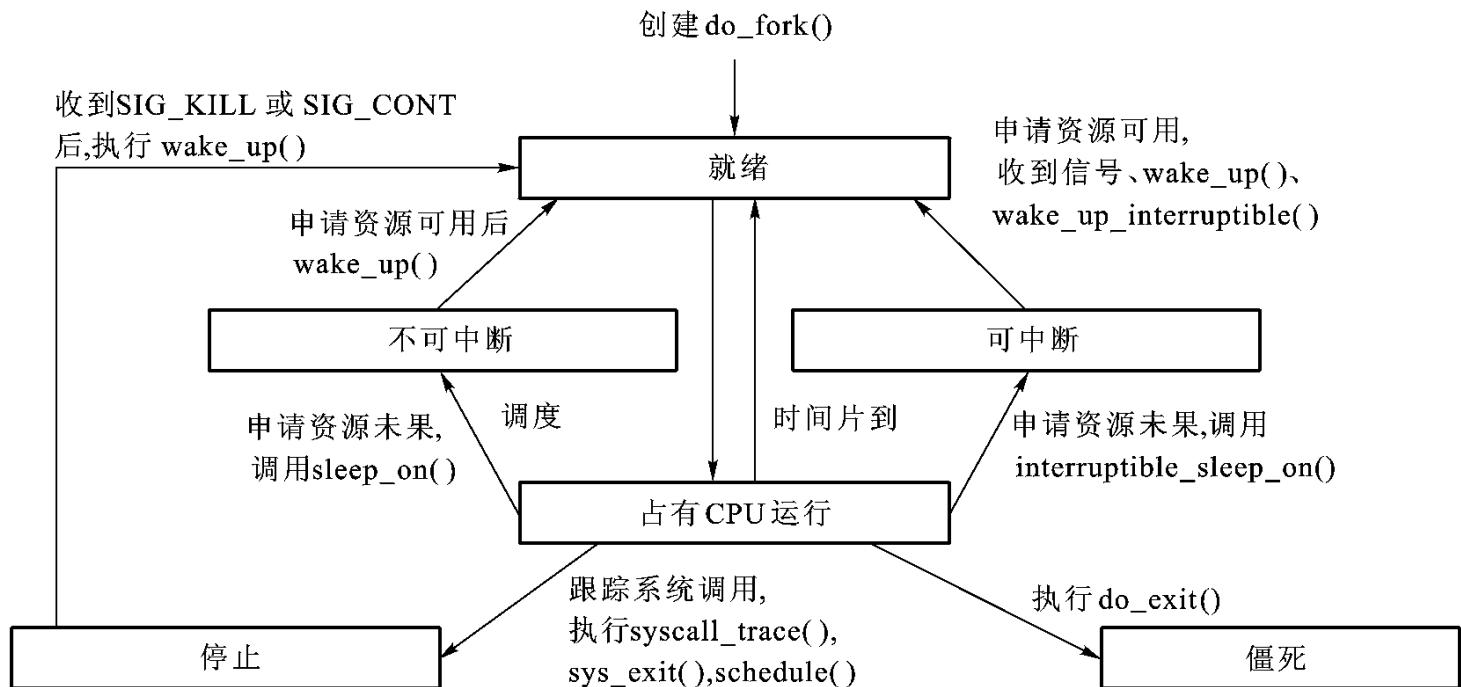


图 2-25 Linux 的进程状态转换

创建进程的系统调用 `sys_fork()` 和 `sys_clone()` 都通过调用 `do_fork()` 函数来完成进程的创建。在 `do_fork()` 函数中, 首先分配进程控制块 `task_struct` 的内存和进程所需的堆栈, 并检测系统是否可以增加新的进程; 然后, 拷贝当前进程的内容, 并对一些数据成员进行初始化; 再为进程的运行做准备; 最后, 返回生成的新进程的进程标识号(`pid`)。如果进程是根据 `sys_clone()` 产生的, 那么, 它的进程标识号就是当前进程的进程标识号, 并且对于进程控制块中的一些成员指针并不进行复制, 而仅仅把这些成员指针的计数 `count` 增加 1。这样, 父子进程可以有效地共享资源。

进程终止的系统调用 `sys_exit()` 通过调用 `do_exit()` 函数实现。函数 `do_exit()` 首先释放进程占用的大部分资源, 然后进入 `TASK_ZOMBIE` 状态, 调用 `exit_notify()` 通知父子进程, 调用 `schedule()` 重新调度。

2. Linux 的进程表

Linux 中进程和任务是一个概念, 核心态中称任务而用户态中就叫进程。进程表(process table)是由 `NR_TASKS` 个任务结构 `task_struct` 组成的静态数组, 包括在 `/include/linux/sched.h` 中, 在 Linux 2.2.10 中其大小为 512, 对于 x86 微机, `NR_TASKS` 最大为 4 096。

Linux 为每个进程分配一个 task-struct 的数据结构, 相当于前面所说的 PCB。task-struct 的主要内容见下表。

表 2-1 task-struct 的构成

| | | |
|---------|---------------------------------------------------------|---------------------------------------------------|
| 调度用数据成员 | state | 进程状态(见上述 6 种状态) |
| | flags | 进程标记(共有 10 多种标志) |
| | priority | 进程静态优先数 |
| | rt_priority | 进程实时优先数 |
| | counter | 进程动态优先数(时间片) |
| | policy | 调度策略(0 基于优先权的时间片轮转、1 基于先进先出的实时调度, 2 基于优先权轮转的实时调度) |
| 信号处理 | signal | 记录进程接收到的信号, 共 32 位, 每位对应一种信号 |
| | blocked | 进程屏蔽信号的屏蔽位, 置位为屏蔽, 复位为不屏蔽 |
| | * sig | 信号对应的自定义或缺省处理函数 |
| 进程队列指针 | * next_task * prev_task | 进程 PCB 双向链接指针、即前向和后向链接指针 |
| | * next_run * prev_run | 运行或就绪队列进程 PCB 双向链接指针 |
| | * p_opptr, * p_pptr * p_cptr * p_ysptr, * p_osptr | 指向原始父进程、父进程, 子进程、新老兄弟进程的队列指针 |
| | uid, gid | 运行进程的用户标识和用户组标识 |
| | Groups[NGROUPS] | 允许进程同时拥有一组用户组号 |
| 进程标识 | euid, egid | 有效的 uid 和 gid, 用于系统安全考虑 |
| | fsuid, fsgid | 文件系统的 uid 和 gid, Linux 特有, 用于合法性检查 |
| | suid, sgid | 系统调用改变 uid/gid 时, 用于存放真正的 uid/gid |
| | pid, pgrp, session | 进程标识号、组标识号、session 标识号 |
| | leader | 是否是 session 的主管, 布尔量 |
| | timeout | 进程间隔多久被重新唤醒, 用于软实时, tick 为单位 |
| | it_real_value it_real_incr | 用于间隔计时器软件定时, 时间到发 SIGALRM |
| 时间数据成员 | real_timer | 一种定时器结构(新定时器) |
| | it_virt_value it_virt_incr | 进程用户态执行间隔计时器软件定时, 时间到发 SIGVTALRM |
| | it_prof_value it_prof_incr | 进程执行间隔计时器软件定时(包括用户和核心态), 时间到发 SIGPROF |
| | utime, stime, cutime ctime, start_time | 进程在用户态、内核态的运行时间, 所有层次子进程在用户态、内核态运行时间总和, 创建进程的时间 |

续表

| | | |
|--------|------------------------|-------------------------------------------------|
| 信号量 | * semundo | 进程每次操作信号量对应的 undo 操作 |
| | * semsleeping | 信号量集合对应的等待队列的指针 |
| 上下文 | * ldt | 进程关于段式存储管理的局部描述符指针 |
| | tss | 保存任务状态信息, 如通用寄存器等 |
| | saved_kernel_stack | 为 MSDOS 仿真程序保存的堆栈指针 |
| | saved_kernel_page | 内核态运行时, 进程的内核堆栈基地址 |
| 文件系统 | * fs | 保存进程与 VFS 的关系信息 |
| | * files | 系统打开文件表, 包含进程打开的所有文件 |
| | link_count | 文件链的数目 |
| 内存数据成员 | * mm | 指向存储管理的 mm_struct 结构 |
| | swappable | 指示进程占用页面是否可以换出, 1 为可换出 |
| | swap_address | 进程下次可换出的页面地址从此开始 |
| | min_flt, maj_flt | 该进程累计的 minor 和 major 缺页次数 |
| | nswap | 该进程累计换出的页面数 |
| | cmin_flt, cmaj_flt | 该进程及其所有子进程累计的缺页次数 |
| | cnswap | 该进程及其所有子进程累计换入和换出的页面计数 |
| | swap_cnt | 下一次循环最多可以换出的页数 |
| SMP 支持 | processor | SMP 系统中, 进程正在使用的 CPU |
| | last_processor | 进程最后一次使用的 CPU |
| | lock_depth | 上下文切换时系统内核锁的深度 |
| 其他 | used_math | 是否使用浮点运算器 FPU |
| | Comm[16] | 进程正在运行的可执行文件的文件名 |
| | rlim | 系统使用资源的限制, 资源当前最大数和资源可有的最大数 |
| | errno | 最后一次系统调用的错误号, 0 表示无错误 |
| | Debugreg[8] | 保存调试寄存器值 |
| | * exec_domain | 与运行 iBCS2 标准程序有关 |
| | personality | |
| | * binfmt | 指向全局执行文件格式结构, 包括 a.out, script, elf, java 等 4 种 |
| | exit_code, exit_signal | 引起进程退出的返回代码, 引起出错的信号名 |
| | dumpable | 出错时是否能够进行 memory dump, 布尔量 |
| | did_exec | 用于区分新老程序代码, POSIX 要求的布尔量 |
| | tty_old_pgrp | 进程显示终端所在的组标识 |
| | * tty | 指向进程所在的显示终端的信息 |
| | * wait_chldexit | 在进程结束需要等待子进程时处于的等待队列, |

此外, Linux 为进程队列的调度定义了以下重要的全局变量:

(1) current 当前正在运行的进程的指针,在 SMP 中则指向 CPU 组中正被调度的 CPU 的当前进程。

(2) init-task 即 0 号进程的 PCB,是进程树的根。

(3) * task[NR - TASKS] 进程 PCB 数组,规定系统可同时运行的最大进程数,每个进程占一个数组元素(元素下标不一定就是进程 pid)。task[0] 必须指向 0 号进程 init-task,可以通过 tasks[] 数组遍历所有进程的 PCB。另外,还提供了宏 for-each-task(),它通过 next-task 遍历所有进程的 PCB。

(4) jiffies 是 Linux 的基准时间,系统初始化时清 0,每隔 10 ms 由时钟中断处理程序 do-timer() 增 1。

(5) need_resched 重新调度标志位,当进程需要系统调度时置位,在系统调用返回前或其他情况下,判别标志位是否为 1,以决定是否调用 schedule() 进行 CPU 调度。

(6) intr_count 记录中断服务程序的嵌套重数。

2.4 线程及其实现

2.4.1 引入多线程技术的动机

在传统的操作系统中,进程是系统进行资源分配的基本单位,按进程为单位分给存放其映像所需要的虚地址空间、执行所需要的主存空间、完成任务需要的其他各类外围设备资源和文件。同时,进程也是处理器调度的基本单位,进程在任一时刻只有一个执行控制流,通常将这种结构的进程称单线程(结构)进程(single threaded process)。

首先来考察一个文件服务器的例子。当它接受一个文件服务请求后,由于等待磁盘传输而经常被阻塞。假如不阻塞可继续接受新的文件服务请求并进行处理,则文件服务器的性能和效率便可以提高,由于处理这些请求时要共享一个磁盘缓冲区,程序和数据要在同一个地址空间中操作。这一类应用非常多,例如,航空售票系统需要处理多个购票和查询请求,这些信息都与同一个数据库相关;而操作系统在同时处理许多用户进程的查询请求时,都要去访问数据库所在的同一个磁盘。对于上述这类基于同数据区的同时多请求应用,用单线程结构的进程难以达到这一目标,即使能解决问题代价也非常高,需要寻求新概念、提出新机制。随着并行技术、网络技术和软件设计技术的发展,给并发程序设计效率带来了一系列新的问题,主要表现在:

- 进程时空的开销大,频繁的进程调度将耗费大量处理器时间,要为每个进程分配存储空间限制了操作系统中进程的总数。

- 进程通信的代价大,每次通信均要涉及通信进程之间或通信进程与操作系统之间的信息传递。
- 进程之间的并发性粒度较粗,并发度不高,过多的进程切换和通信延迟使得细粒度的并发得不偿失。
- 不适合并行计算和分布并行计算的要求,对于多处理器和分布式的计算环境来说,进程之间大量频繁的通信和切换,会大大降低并行度。
- 不适合客户/服务器计算的要求。对于 C/S 结构来说,那些需要频繁输入输出并同时大量计算的服务器进程(如数据库服务器、事务监督程序)很难体现效率。

这就迫切要求操作系统改进进程结构,提供新的机制。使得应用能够按照需求在同一进程中设计出多条控制流;多控制流之间可以并行执行;多控制流切换不需通过进程调度;多控制流之间还可以通过内存区直接通信,降低通信开销。这就是近年来流行的多线程(结构)进程(multiple threaded process)。如果说操作系统中引入进程的目的是为了使多个程序能并发执行,以改善资源使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开销,使得并发粒度更细、并发性更好。这里解决问题的基本思路是:把进程的两项功能——“独立分配资源”与“被调度分派执行”分离开来,前一项任务仍由进程完成,它作为系统资源分配和保护的独立单位,不需要频繁地切换;后一项任务交给称作线程的实体来完成,它作为系统调度和分派的基本单位,能轻装运行,会被频繁地调度和切换。在这种指导思想下,产生了线程的概念。

传统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;传统的 UNIX 支持多用户进程,每个进程也是单线程的。目前,很多著名的操作系统都支持多线程(结构)进程,如 Solaris、Mach、SVR4. OS/390、OS/2. WindowNT、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了自己的线程接口规范,如 Solaris thread 接口规范、OS/2 thread 接口规范、Windows NT thread 接口规范等,IEEE 也推出了 UNIX 类操作系统的多线程程序设计标准 POSIX 1003.4a。事实上,线程概念不仅局限于操作系统中,在程序设计语言、数据库管理系统和其他一些应用软件中,也通过引入线程来改善系统和应用程序的性能,可以相信多线程技术在程序设计中将会被越来越广泛地采用。

2.4.2 多线程环境中的进程与线程

1. 多线程环境中的进程概念

在传统操作系统的单线程进程中,进程和线程概念可以不加区别。图 2 – 26 给出了单线程进程的内存布局和结构,它由进程控制块和用户地址空间,以及管理进程执行的调用/

返回行为的系统堆栈或用户堆栈构成。一个进程的结构可以划分成两个部分：对资源的管理和实际的指令执行序列。显然，采用并发多进程程序设计时，并发进程之间的切换和通信均要借助于操作系统的进程管理和进程通信机制，因而实现代价较大，而较大的进程切换和进程通信代价，又进一步影响了并发的粒度。

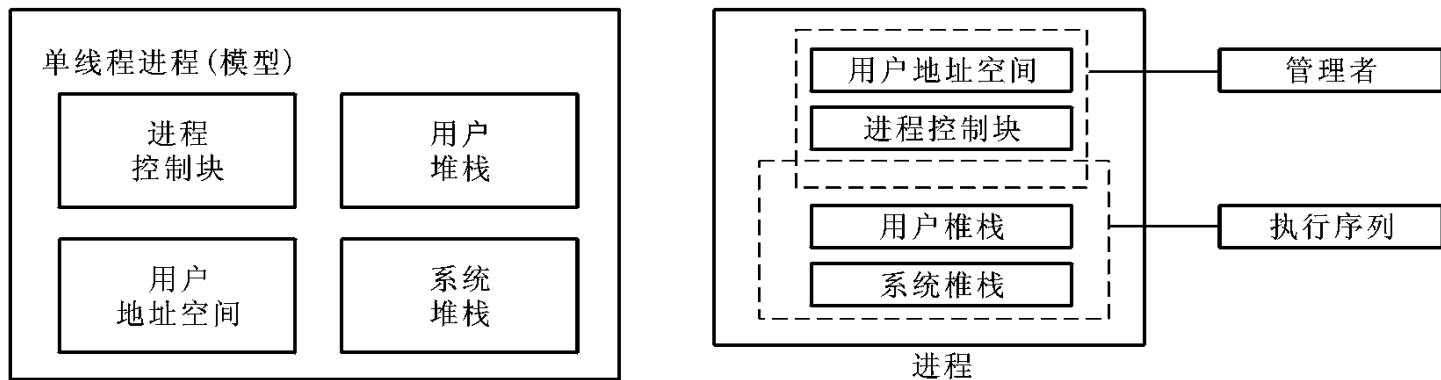


图 2-26 单线程进程的内存布局

设想是否可以把进程的管理和执行任务相分离，如图 2-27 所示，让进程是操作系统中进行保护和资源分配的单位，允许一个进程中包含多个可并发执行的控制流。这些控制流切换时不必通过进程调度，通信时可以直接借助于共享内存区。每个控制流称为一个线程，这就是并发多线程程序设计。

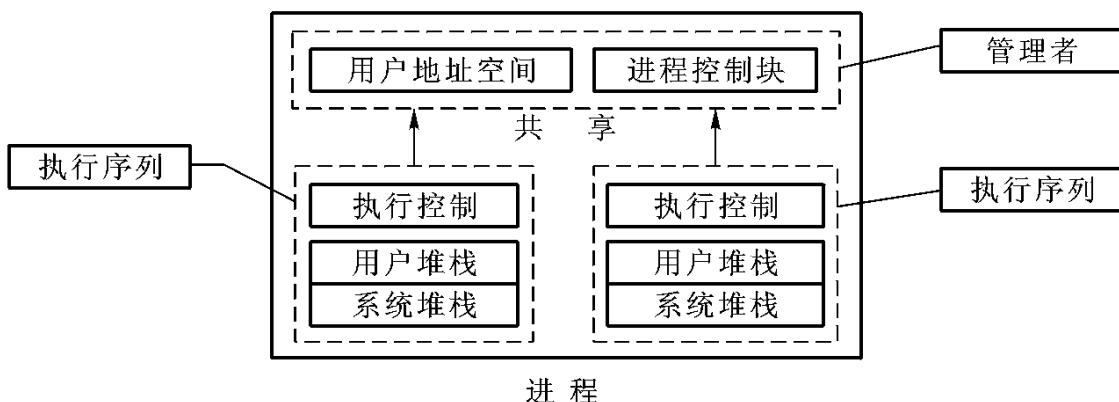


图 2-27 管理和执行相分离的进程

多线程进程的内存布局如图 2-28 所示。在多线程环境中，仍然有与进程相关的内容是 PCB 和用户地址空间，而每个线程除了有独立堆栈，以及包含现场信息和其他状态信息外，也要设置线程控制块 TCB (Thread Control Block)。线程间的关系较为密切，一个进程中的所有线程共享其所属进程拥有的资源，它们驻留在相同的地址空间，可以存取相同的数据。例如，当一个线程改变了主存中一个数据项时，如果这时其他线程也存取这个数据项，它便能看到相同的结果。

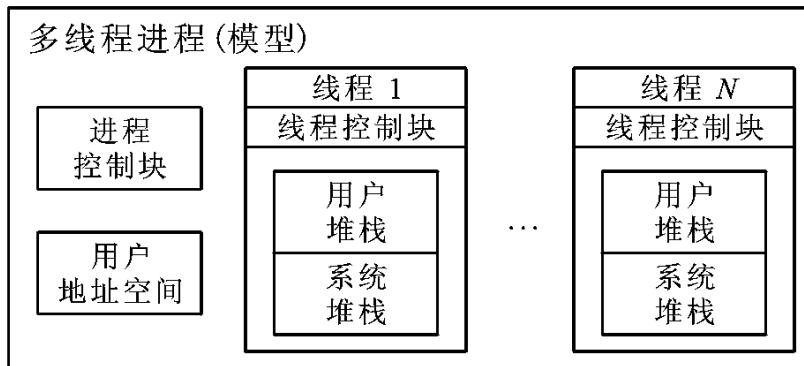


图 2-28 多线程进程的内部布局

最后,给出多线程环境中进程的定义:进程是操作系统中进行保护和资源分配的基本单位。它具有:

- 一个虚拟地址空间,用来容纳进程的映像;
- 对处理器、其他(通信的)进程、文件和 I/O 资源等的有控制有保护的访问。

而传统进程原先所承担的控制流执行任务交给称作线程的部分完成。

2. 多线程环境中的线程概念

线程是操作系统进程中能够独立执行的实体(控制流),是处理器调度和分派的基本单位。线程是进程的组成部分,每个进程中允许包含多个并发执行的实体(控制流),这就是多线程。同一个进程中的所有线程共享进程获得的主存空间和资源,但不拥有资源。线程具有:

- 线程执行状态(运行、就绪、等待……)。
- 当线程不运行时,有一个受保护的线程上下文,用于存储现场信息。所以,观察线程的一种方式是运行在进程内一个独立的程序计数器。
- 一个执行堆栈。
- 一个容纳局部变量的主存存储区。

线程的主要特性是:

(1) 并发性:同一进程的多个线程可在多个处理器上并发或并行地执行,而进程之间的并发执行演变为不同进程的线程之间的并发执行。

(2) 共享性:同一个进程中的所有线程共享但不拥有进程的状态和资源,且驻留在进程的同一个主存地址空间中,可以访问相同的数据。所以,需要有线程之间的通信和同步机制。通信和同步的实现十分方便。

(3) 动态性:线程是程序在相应数据集上的一次执行过程,由创建而产生,至撤销而消亡,有其生命周期,经历各种状态的变化。每个进程被创建时,至少同时为其创建一个线程,需要时线程可以再创建其他线程。

(4) 结构性:线程是操作系统中的基本调度和分派单位,因此,它具有惟一的标识符和线程控制块,其中应包含调度所需的一切私有信息。

如图 2-29 所示,进程可以划分为两个部分:资源集合和线程集合。进程要支撑线程运行,为线程提供地址空间和各种资源,它封装了管理信息,包括对指令代码、全局数据和 I/O 状态数据等共享部分的管理。线程封装了执行信息,包括对 CPU 寄存器、执行栈(用户栈、内核栈)和局部变量、过程调用参数、返回值等线程私有部分的管理。由于线程具有许多传统进程所具有的特征,所以也把线程称为轻量进程 LWP(Light-Weight Process)。

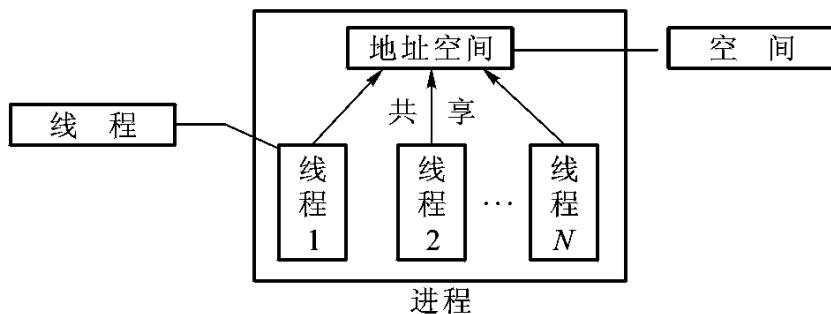


图 2-29 线程的内存布局

下面用一个日常生活中的例子来说明多线程结构进程可以进一步提高系统的并发性。某大厦的装璜工程可作为一个“进程”运行,下有许多工程队,如瓦工队、木工队、水电工队、油漆工队等,每个工程队作为一个“线程”运行。“进程”负责采购资源(原料)和工程管理,有原料时这些工程队可以按进度齐头并进同时工作(多线程并行执行),以加快装璜进度。缺少原料时,如缺少水泥、木材、水管、油漆之一时,相应工程队等待(线程被阻塞),而可以调度其他工程队(线程)工作。从而,提高了整个系统(装璜工程)的并发性,加快了工程进度。

3. 线程的状态

和进程类似,线程也有生命周期,因而也存在各种状态。从调度需要来说,线程的关键状态有:运行、就绪和等待。另外,线程的状态转换也类似于进程。由于线程不是资源的拥有单位,挂起状态对线程是没有意义的,如果一个进程挂起后被对换出主存,则它的所有线程因共享了进程的地址空间,也必须全部对换出去。可见由挂起操作引起的状态是进程级状态,不作为线程级状态。类似地,进程的终止会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执行中要求系统服务,如执行一个 I/O 请求而成为等待态时,那么,多线程进程中是不是要阻塞整个进程,即使这时还有其他就绪的线程,对于某些线程实现机制,所在进程也转换为阻塞态;对于另一些线程实现机制,如果存在另外一个处于就绪态的线程,则调度该线程处于运行状态,否则进程才转换为等待态。显然前一种做法欠妥,丢失了多线程机制的优越性,降低了系统的灵活性。

多线程进程的进程状态是怎样定义的？由于进程不是调度单位，不必划分成过细的状态，如 Windows 操作系统中仅把进程分成可运行和不可运行态，挂起状态属于不可运行态。

4. 线程管理和线程包(库)

多线程技术是利用线程包(库)来提供一整套有关线程的原语集来支持多线程运行的，有的操作系统直接支持多线程，而有的操作系统不支持多线程。因而，线程包(库)可以分成两种：用户空间中运行的线程包(库)和内核中运行的线程包(库)。一般地说，线程包(库)至少应提供以下功能的原语调用：孵化、封锁、活化、结束、通信、同步、互斥等，以及切换(保护和恢复线程上下文)的代码，调度(对线程的调度算法及实施处理器调度)的代码。同时应提供一组与线程有关的应用程序编程接口 API，支持应用程序创建、调度、撤销和管理线程的运行。

基本的线程控制原语有：

- 孵化(spawn)，又称创建线程。当一个新进程被生成后，该进程的一个线程也就被创建。此后，该进程中的一个线程可以孵化同一进程中的其他线程，并为新线程提供指令计数器和参数。一个新线程还将被分配寄存器上下文和堆栈空间，并将其加入就绪队列。
- 封锁(block)，又称线程阻塞或等待。当一个线程等待一个事件时，将变成阻塞态，保护它的用户寄存器、程序计数器和堆栈指针等现场。处理器现在就可以转向执行其他就绪线程。
- 活化(unblock)，又称恢复线程。当被阻塞线程等待的事件发生时，线程变成就绪态或相应状态。
- 结束(finish)，又称撤销线程。当一个线程正常完成时，便回收它占有的寄存器和堆栈等资源，撤销线程 TCB。当一个线程运行出现异常时，允许强行撤销一个线程。

对于在用户空间运行的线程包(库)，由于它完全在用户空间中运行，操作系统内核对线程包(库)不可见，而仅仅知道管理的是一般的单线程进程。这种情况下，线程包(库)起到一个微内核的作用，实质上是多线程应用程序的开发和运行支撑环境。优点是：节省了内核的宝贵资源，减少内核态和用户态之间的切换。因而，线程(包)库的运行开销小效率高，容易按应用特定需要选择进程调度算法。也就是说，线程库的线程调度算法与操作系统的低级调度算法是无关的，能运行在任何操作系统上。缺点是：当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程会被阻塞，这种多线程应用就不能充分利用多处理器的优点。

在内核中运行的线程包(库)，则是通过内核来管理线程包(库)的。内核中不但要保存进程的数据结构，也要建立和维护线程的数据结构及保存每个线程的入口，线程管理的所有工作由操作系统内核来实现。由内核专门提供一组应用程序编程接口(API)，供开发者开发多线程应用程序。优点是：能够调度同一进程中多个线程同时在处理器上并行执行，充分发

挥多处理器的能力,若进程中的一个线程被阻塞了,内核能调度同一进程的其他线程占有处理器运行,也可以调度其他进程中的线程运行。缺点是: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—内核态—用户态的模式切换,系统开销较大。

基于上述两种线程包(库)可把线程的实现分成三类:用户级线程ULT(User Level Thread,如POSIX1003.4a的P-threads、Java的线程库)和内核级线程KLT(Kernel Level Thread,如Windows 2000/XP、OS/2和Mach C-thread)。也有一些系统(如Solaris)提供了混合式,同时支持ULT和KLT两种线程的实现。下面将会进一步讨论三种线程包(库)的实现方法。

近年来,已出现了具有支持多线程运行的微处理器体系结构,称为超线程技术。如Intel公司发布的新一代奔腾4处理器,是具有超线程技术的微处理器。它的微处理器包含两个逻辑上独立的微处理器,能够同时执行两个独立的线程代码流。每个均可被单独启停、中断和被调度执行特定的线程,而不会影响芯片上另一个逻辑上独立的处理器共享微处理器内核的执行资源,包括引擎、高速cache、总线接口和固件等。

5. 并发多线程程序设计的优点

在一个进程中包含多个并行执行的控制流,而不是把多个可并发执行的控制流——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不同之处。并发多线程程序设计的主要优点是使系统性能获得很大提高,具体表现在:

- 快速线程切换。进程具有独立的虚地址空间,以进程为单位进行任务调度,系统则必须交换地址空间,切换时间长;而在同一进程中的多线程共享同一地址空间,因而能使线程快速切换。
- 减少(系统)管理开销。对多个进程的管理(创建、调度、终止等)系统开销大,如响应客户请求建立一个新的服务进程的服务器应用中,创建的开销比较显著。面对创建、终止线程,虽也有系统开销,但要比进程小得多。Mach开发者的研究表明,与没有使用线程的UNIX相比,进程创建的速度提高了10倍。
- (线程)通信易于实现。为了实现协作,进程或线程间需要交换数据。对于自动共享同一地址空间的各线程来说,所有全局数据均可自由访问,不需什么特殊设施就能实现数据共享。而进程通信则相当复杂,必须借助诸如通信机制、消息缓冲、管道机制等设施,而且还要调用内核功能才能实现,同时线程通信的效率也很高。
- 并发程度提高。许多多任务操作系统限制用户能拥有的最多进程数目,如早期UNIX一般为50个,这对许多并发应用来说是不够的。而对多线程技术来说,一般可达几千个,基本上不存在线程数目的限制。
- 节省内存空间。多线程合用进程地址空间,而不同进程独占地址空间,使用不经济。

由于队列管理和处理器调度是以线程为单位的,因此,多数涉及执行的状态信息被维护在线程数据结构中。然而,存在一些影响到一个进程中所有线程的活动,操作系统必须在进

程级进行管理。挂起意味着将主存中的地址空间对换到磁盘上,因为在同一个进程中的所有线程共享同一地址空间,此时所有线程也必须进入挂起状态。相似地,终止一个进程时所有线程应被终止。

6. 线程组织和多线程技术的应用

一个进程中可包括若干线程,每个线程拥有自己的程序计数器和堆栈,用来跟踪程序运行的轨迹,这些线程分享了进程的处理器时间配额。在单处理器系统中,一个线程先执行,接着同一进程或其他进程中的另一个线程执行,这很像分时系统的做法。在多处理器系统中,同一进程或不同进程中的多个线程真正并行执行。一个进程中的所有线程都在同一个地址空间中,共享全局变量和各种资源,如打开文件、定时器、信号量、内存区。

进程中的线程可有多种组织方式: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的角色接受和处理工作请求,其他线程为工作者线程。由调度员线程分配任务和唤醒工作者线程工作。第二种是组模式,进程中的各个线程都可以取得并处理该请求,不存在调度者线程。有时每个线程被设计成专门处理特定的任务,同时建立相应的任务队列。第三种是流水线模式,线程排成一个次序,第一个线程产生的数据传送给下一个线程处理,依次类推,数据沿排定的卜次序由线程依次传递以完成请求的任务。

多线程技术在现代计算机软件中得到了广泛的应用,取得了较好的效果。下面举例说明多线程技术的一些主要应用:

- 前台和后台工作。如在一个电子表格软件中,一个线程执行显示菜单和读入用户输入,同时,另一个线程执行用户命令和修改电子表格。
- C/S 应用模式。局域网上文件(网络)服务器处理多个用户文件(任务)请求时,创建多个线程,若该服务器是多 CPU 的,则同一进程中的多线程可以同时运行在不同 CPU 上。
- 加快执行速度。一个多线程进程在计算一批数据的同时,读入设备(网络、终端、打印机、硬盘)上的下一批数据,而这分别由两个线程实现。
- 异步处理。程序中的异步成分可用线程实现。例如,为避免掉电带来损失,可把文字编辑器设计成周期性把内存缓冲内容写入到磁盘中。可以创建一个线程完成周期性写盘任务,该线程由操作系统调度,并不需要应用进程提供代码来做检查或输入输出。
- 设计用户接口。每当用户要求执行一个动作时,就建立一个独立线程来完成这项动作。当用户要求有多个动作时,就由多个线程来实现,窗口系统应有一个线程专门处理鼠标的动作。例如,GUI 中,后台进行屏幕输出或真正计算,同时要求对用户输入(鼠标)做出反映。有了多线程,可同时运行 GUI 输入线程和后台计算线程,便能实现这一功能。

2.4.3 线程的实现

从实现的角度看,线程可以分成用户级线程 ULT(如 POSIX 的 P-threads、Java 的线程库)

和内核级线程 KLT(如 Windows 2000/XP、OS/2 和 Mach 的 C-thread), 分别在用户空间和核心空间实现。也有一些系统(如 Solaris)提供了混合式线程, 同时支持两种线程实现。图 2-30 给出了各种线程实现方法。

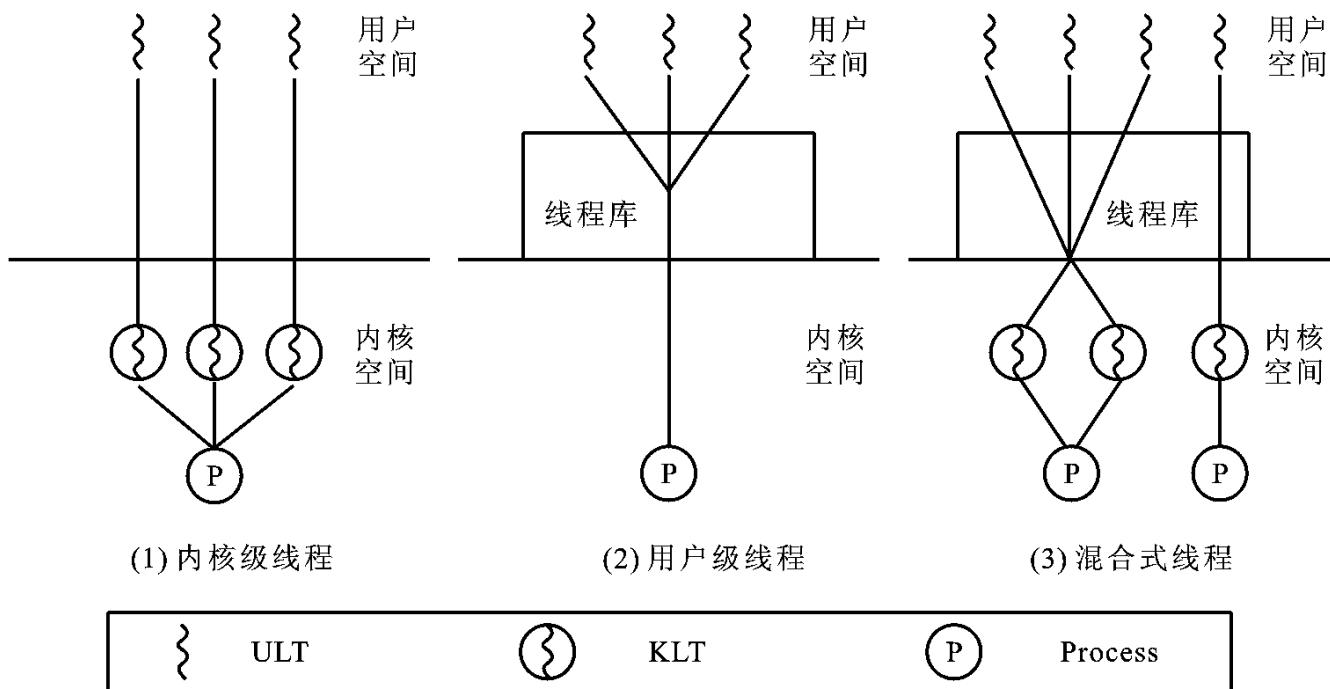


图 2-30 线程实现方法

1. 内核级线程

在纯内核级 KLT(Kernel Level Threads)线程设施中, 线程管理的所有工作由操作系统内核来做。内核专门提供了一个 KLT 应用程序设计接口(API), 供开发者使用, 应用程序区不需要有线程管理的代码。Windows 2000/XP 和 OS/2 都是采用这种方法的例子。

任何应用都可以被程序设计成多个线程, 当提交给操作系统执行时, 内核为它创建一个进程和一个线程, 线程在执行中可以通过内核创建线程原语来创建其他线程, 这个应用的所有线程均在一个进程中获得支持。内核要为整个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, 内核的调度是在线程的基础上进行的。

这一方法有两个主要优点:首先, 在多处理器上, 内核能够同时调度同一进程中多个线程并行执行;其次, 若进程中的一个线程被阻塞了, 内核能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, 由于内核线程仅有很小的数据结构和堆栈, KLT 的切换比较快, 内核自身也可以用多线程技术实现, 从而, 能提高系统的执行速度和效率。

KLT 的主要缺点是:应用程序线程在用户态运行, 而线程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—内核态—用户态的模式切换,系统开销较大。

2. 用户级线程

纯用户级线程 ULT(User Level Threads) 设施中,线程管理的全部工作都由应用程序来做,在用户空间内实现,内核是不知道线程的存在的。用户级多线程由用户空间运行的线程库来实现,任何应用程序均需通过线程库进行程序设计,再与线程库连接后运行来实现多线程。线程库是一个 ULT 管理的例行程序包,在这种情况下,线程库是线程的运行支撑环境。

当一个应用程序提交给系统后,系统为它建立一个由内核管理的进程,该进程在线程库环境下开始运行时,只有一个由线程库为进程建立的线程。首先,运行这个线程,当应用进程处于运行状态时,线程通过调用线程库中的“孵化”过程,可以孵化出运行在同一进程中的新线程。步骤如下:通过过程调用把控制权传送给“孵化”过程,由线程库为新线程创建一个 TCB 数据结构,并置为就绪态。然后,按一定的调度算法把控制权传递给该进程中处于就绪态的一个线程。当控制权传送到线程库时,当前线程的现场信息应被保存,而当线程库调度一个线程执行时,便要恢复它的现场信息。现场信息主要包括:用户寄存器内容、程序指令计数器和堆栈指针。当线程运行,执行系统调用,导致它挂起并进入等待态时,线程库代码会寻找一个就绪态线程来执行,这时将进行线程上下文切换,而新线程被激活。

上述活动均发生在用户空间,且在单个进程中,内核并不知道这些活动。内核按进程为单位调度,并赋予一个进程状态(就绪、运行、阻塞……)。下面的例子清楚地表明了线程调度和进程调度之间的关系。假设进程 B 正在执行它的线程 2,则可能出现下列情况:

- 正在执行的进程 B 的线程 2 发出了一个封锁进程 B 的系统调用,例如做了一个 I/O 操作。这导致控制转移到内核,内核启动 I/O 操作,把进程 B 被置为阻塞状态,并切换到另一个进程。按照由线程库所维护的数据结构,进程 B 的线程 2 仍然处在运行态。十分重要的是线程 2 的运行并不是真正意义上的被处理器执行,而是可理解为在线程库的运行态中。这时,进程 B 中的其他线程尽管有的处于可运行的就绪态,但因进程 B 被阻塞而也都被阻塞了。

- 一个时钟中断传送控制给内核,内核中止当前时间片用完的进程 B,并把它放入就绪队列,切换到另一个就绪进程。此时,由于进程 B 中线程 2 正在运行,按由线程库维护的数据结构,进程 B 的线程 2 仍处于运行态,进程 B 却已处于就绪态。

- 线程 2 执行到某处,它需要进程 B 的线程 1 的某些操作。于是让线程 2 变成阻塞态,而线程 1 从就绪态转为运行态,注意进程始终处在运行态。

上述头两种情况中,当内核切换控制权返回到进程 B 时,便恢复原来断点现场,线程 2 继续执行。注意到当正在执行线程库中的代码时,一个进程也有可能由于时间片用完或被更高优先级的进程剥夺而被中断。当中断发生时,一个进程可能正处在从一个线程切换到

另一个线程的过程中间；当一个进程恢复时，继续在线程库中执行，完成线程切换，并传送控制权给进程中的一个新线程。使用 ULT 代替 KLT 有许多优点：

- 线程切换不需要内核特权方式，因为，所有线程管理数据结构均在单个进程的用户空间中，管理线程切换的线程库也在用户地址空间运行，因而进程不要切换到内核方式来做线程管理。这就节省了模式切换的开销，也节省了内核的宝贵资源。

- 按应用特定需要允许进程选择调度算法，一种应用可能从简单轮转调度算法得益，同时，另一种应用可能从优先级调度算法获得好处。在不干扰操作系统调度的情况下，根据应用需要可以裁剪调度算法，也就是说，线程库的线程调度算法与操作系统的低级调度算法是无关的。

ULT 能运行在任何操作系统上，内核在支持 ULT 方面不需要做任何改变。线程库是可以被所有应用共享的应用级实用程序，许多当代操作系统和语言均提供了线程库，传统的 UNIX 并不支持多线程，但已有了多个基于 UNIX 的用户线程库。

和 KLT 比较，ULT 有两个明显的缺点：

- 在传统的基于进程操作系统中，大多数系统调用将阻塞进程。因此，当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程会被阻塞。而在 KLT 中，这时可以选择另一个线程运行。

- 在纯 ULT 中，多线程应用不能利用多重处理的优点。内核在一段时间里，分配一个进程仅占用一个 CPU，因而进程中仅有一个线程能执行。因此，尽管多道程序设计能够明显地加快应用处理速度，也具备了在一个进程中进行多线程设计的能力，但通常不可能得益于多线程并发地执行。

克服上述问题的方法有两种。一是用多进程并发程序设计来代替多线程并发程序设计。这种方法事实上放弃了多线程带来的所有优点，每次切换变成了进程而非线程切换，导致开销过大。第二种方法是采用称为护套技术 (jacketing) 来解决阻塞线程的问题。主要思想是：把阻塞式的系统调用改造成非阻塞式的系统调用，当线程调用系统调用前，首先调用 jacketing 实用例程，来检查 I/O 设备使用情况。如果忙碌，该线程进入就绪态并把控制权传递给另一个线程，当这个线程后来又重新得到控制权时，再重新调用 jaketing 例程检查 I/O 设备。

3. 混合式线程

有些操作系统提供了混合式 ULT/KLT 线程，Solaris 便是一个例子。在混合式线程系统中，内核支持 KLT 多线程的建立、调度和管理。同时也提供线程库，允许用户应用程序建立、调度和管理 ULT。一个应用程序的多个 ULT 映射成一些 KLT，程序员可按应用需要和机器配置调整 KLT 数目，以达到较好效果。

混合式线程中，一个应用中的多个线程能同时在多处理器上并行执行，且阻塞一个线程

时并不需要封锁整个进程。如果设计得当的话,则混合式多线程机制能够结合两者优点,并舍去它们的缺点。

2.4.4 实例研究:Solaris 的进程与线程

1. Solaris 中的进程与线程概念

Solaris 采用了 4 个与进程和线程有关的概念,用来支持完全可抢占、SMP 和内核级多线程结构:

- 进程 (Process): 通常的 UNIX 进程,它包含用户的地址空间、堆栈和 PCB。
- 用户级线程 (User-Level Threads): 通过线程库在用户地址空间中实现,对操作系统来讲是不可见的,用户级线程 (ULT) 是应用程序并行机制的接口。
- 轻量进程 (Light Weight Process): 一个 LWP 可看作是 ULT 和 KLT 之间的映射,每个 LWP 支持一个或多个 ULT,并映射到一个 KLT 上。LWP 由内核独立调度,可以在多个处理器上并行执行。
- 内核级线程 (Kernel-Level Threads): KLT 是能被调度和指派到处理器上去运行的基本实体。

Solaris 的线程实现分为两个层次:用户层和核心层。用户层在用户线程库中实现;核心层在操作系统内核中实现,处于两个层次的线程分别叫用户级线程和内核级线程。核心实现 LWP 和 KLT,核心向用户程序提供的(界面)是 LWP,不提供 KLT,它只知道 KLT 和 LWP,而不知道 ULT。用户程序只看到 LWP 和 ULT,而不知道 KLT,ULT 的运行是由 LWP 实现的。

ULT 是一个代表对应线程的数据结构,它是纯用户级的概念,占用用户空间资源,对核心是透明的。一个进程中可以设计一个或多个 LWP(为节省系统开销,不可能设置太多的 LWP),而一个 LWP 上又可以设计一个或多个 ULT。为使每个 ULT 都能利用 LWP 与内核通信,可让多个 ULT 多路复用一个 LWP,但只有当前连到 LWP 上的线程,才能与内核通信,其余线程或者阻塞或者等待 LWP。而每一个 LWP 都要连接到一个 KLT 上,这样,通过 LWP 可把 ULT 和 KLT 连接起来,ULT 通过 LWP 来访问内核。

有了 LWP,就可以在用户级实现 ULT,每个进程可以创建许多 ULT,而又不占用核心资源。当 LWP 在一个进程的不同 ULT 间切换时,仅是数据结构的切换,其时间开销远低于两个 KLT 间的切换时间。同样线程间的同步也独立于核心的同步体系,在用户空间中独立实现,而不陷入内核。核心能看见的是 LWP,而看不到 ULT。由 LWP 把 ULT 和 KLT 隔离开来。

当 ULT 不需要与内核通信时,并不需要 LWP;而要通信时,便须借助于 LWP 的帮助,而且每个要通信的 ULT 都需要一个 LWP。例如,一个任务中,如果同时有 5 个 ULT 发出了对文件的读写请求,这时就需要有 5 个 LWP 来予以帮助。即由 LWP 将对文件的读写请求,发

送给对应的 KLT, 再由这个 KLT 执行具体的读写操作。若这个任务中仅有 4 个 LWP, 则只能有 4 个 ULT 的文件读写请求传送给 KLT, 余下的一个 ULT 必须等待。

在 KLT 执行操作时, 如果发生阻塞, 那么, 如果进程中只包含了一个 LWP, 此时进程也应阻塞, 这种情况与传统操作系统一样, 在进程执行系统调用时实际上该进程是被阻塞的。但如果一个进程中多个 LWP, 则当一个 LWP 阻塞时, 进程中的另一个 LWP 可继续运行, 而即使进程中的所有 LWP 全部阻塞, 进程中的 ULT 也仍能继续运行, 只是不能再去访问内核。

多线程的程序员接口包括两个层次:一层为线程接口, 提供给用户编写多线程应用程序;第二层为 LWP 接口, 提供给线程库, 以管理 LWP。多线程的两个程序员接口是类似的, 线程接口的切换代价较小, 特别适用于解决逻辑并行性问题;而 LWP 接口则适用于那些需要在多个处理器进行并行计算的问题。如果程序设计得当的话, 混合应用两种程序员接口可以带来更大的灵活性和优越性。

图 2-31 显示了 4 个实体:进程、ULT、LWP 和 KLT 间的关系。LWP 和 KLT 是一一对应的, 在一个应用进程中 LWP 是可见的, LWP 的数据结构存在于它们各自的进程地址空间中。同时, 每个 LWP 与一个可调度的内核线程 KLT 绑定, 而内核线程 KLT 的数据结构则维护在内核地址空间中。

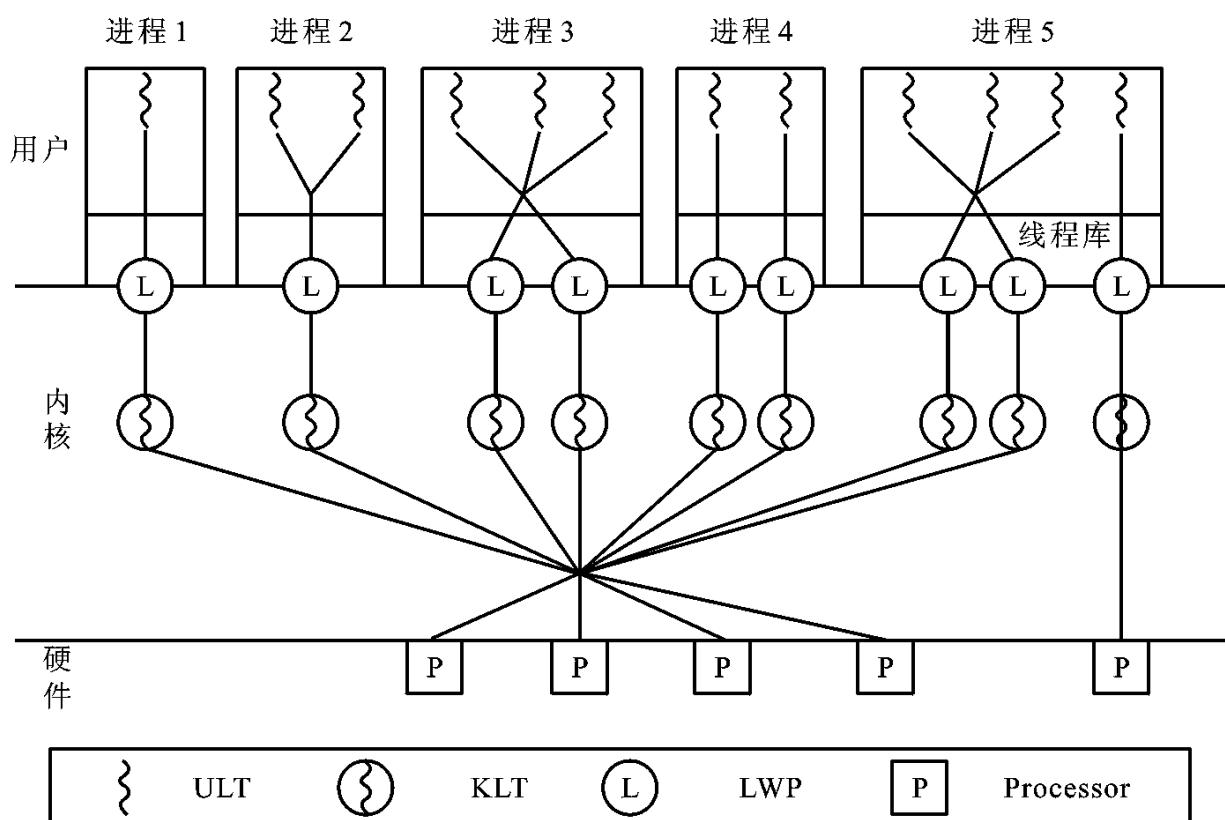


图 2-31 Solaris 线程的使用

图中,进程 1 是传统的单线程进程,当应用不需要并发运行时,可以采用这种结构。进程 2 是一个纯的 ULT 应用,所有的 ULT 由单个内核线程支撑,因而某一时刻仅有一个线程可运行。对于那些在程序设计时要表示并发而不真正需要多线程并行执行的应用来说,这种方式是有用的。进程 3 是多线程与多个 LWP 的对应,Solaris 允许应用程序开发多个 ULT 工作在小于或等于 ULT 个数的 LWP 上。这样应用可以定义进程在内核级上的某种程度的并行。进程 4 的线程与 LWP 是一对一地捆扎起来的,这种结构使得内核并行机制完全对应可见,这在线程常常因阻塞而被挂起时是有用的。进程 5 包括多 ULT 映射到多 LWP 上,以及 ULT 与 LWP 的一一捆绑,并且还有一个 LWP 捆在单个处理器上。

2. Solaris 的进程结构

在 Solaris 中已经有了进程、KLT 和 ULT,为什么还要引入 LWP? 先来讨论这个问题。最主要的原因是各种应用的需求,有些应用逻辑并行性程度高,有些应用物理并行性要求高。像窗口系统是典型的逻辑并行性程度高的应用,同时在屏幕上开出了多个窗口,窗口切换很频繁,但一个时刻仅有少数窗口处于活跃状态。这可以用一组 ULT 来表达窗口系统,用一个或很少几个 LWP 来支持这一组 ULT,可根据活跃窗口数目调整 LWP 的个数。由于 ULT 是由用户线程库实现的,对它们的管理不涉及内核;内核仅要管理与 ULT 对应的一个或几个 LWP,系统开销小,窗口系统的效率高。

大规模并行计算是物理并行性要求高的应用,可以把数组按列划分给不同的 ULT 线程处理,如果每个 CPU 对应一个 LWP,而一个 LWP 对应多个 ULT,那么 CPU 有很多时间化在线程切换上。这种情况下,最好把列分给少量 ULT,而一个 ULT 和一个 LWP 绑定,以减少线程切换次数,提高并行计算的效率。

对某些应用程序来说,混合使用永久绑定到 LWP 上的线程和没有绑定的线程(多个线程共享多个 LWP)是最合适的。例如,一个实时应用程序可能希望某些线程具有系统范围的优先级且实时调度,而其他线程执行后台功能且可以共享一个或少量几个 LWP。

Solaris 中,每个进程包括不同数目的轻量进程,进程结构如图 2-32 所示。

进程数据结构(每个进程一个)包括:进程标识符、用户标识符、内核使用的信号表(当一个信号发送给一个进程时,内核要做什么)、进程打开的文件描述符、进程地址空间指针、有关处理器状态结构信息(该进程的内核堆栈)。Solaris 中由于进程不再是调度单位,每个进程对应有不同数量的 LWP。所以,原 UNIX 进程结构中的处理器信息取消,而被加进包含有 LWP 的数据结构的表格。

LWP 数据结构(每个 LWP 一个)包括:轻量进程标识符、优先数、定义内核能够接受信号的信号掩码、存放现场信息(用户级寄存器)的值、含有系统调用的参数/返回值/出错码的内核堆栈、使用的资源和数据、指向对应内核线程的指针、指向进程结构的指针等有关信息。

每个 LWP 相应的 KLT 的数据结构包括:优先级和调度信息、运行或等待队列指针、相应

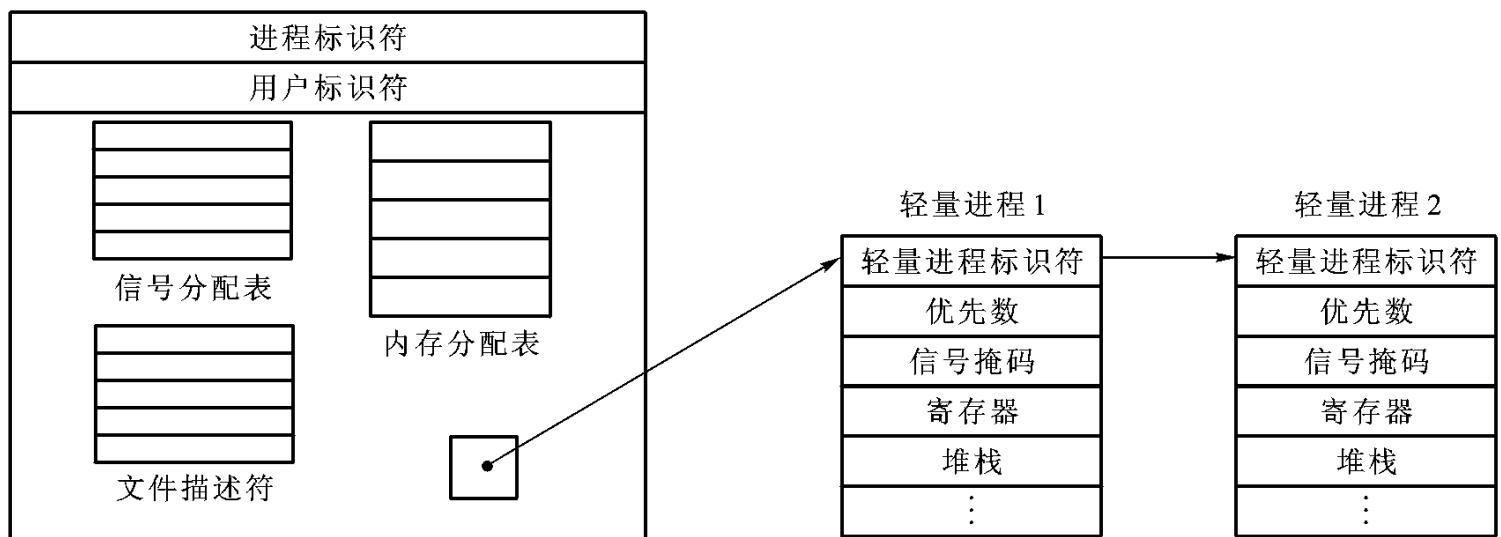


图 2-32 Solaris 的进程结构

进程所有线程形成的链表指针、有关 LWP 非交换数据、指向相关 LWP 的指针等有关信息和一个内核堆栈。

3. Solaris 的线程状态

图 2-33 简单说明了 ULT 和 LWP 的状态。用户级线程的执行是由线程库管理的, 这里首先考虑非捆绑的线程, 即多个 ULT 可共享 LWP。一个非捆绑的线程可能处于四个状态之一: 可运行、活跃、睡眠和停止。一个处在活跃状态的 ULT 当前指定给一个 LWP, 并且在对应的内核线程 KLT 执行时, 它被执行。许多事件会导致 ULT 离开活跃态, 考虑一个活跃的 ULT 正在执行, 下面的事件可能发生:

- 同步: 有一个 ULT 调用了一条同步原语与其他线程协调活动。于是它就进入睡眠态, 直到同步条件满足后, 这个 ULT 才被放入可运行队列。
- 挂起: 任何线程可挂起其他线程; 一个线程也可挂起自己并进入停止状态, 直到别的线程发出一个继续运行的请求, 再把该线程移入可运行队列。
- 剥夺: 一个活跃线程在执行时, 另一个更高优先级的线程变为可运行状态, 如果处于活跃状态的运行线程优先级较低, 它就要被剥夺并放入可运行状态, 而更高优先级的线程被分配到可用的 LWP 上执行。
- 让位: 若运行线程执行了 `thread_yield()` 库命令, 库中的线程调度程序将查看是否有另一个可运行线程。若它与当前执行的线程具有相同的优先级, 则把执行线程放入可运行队列, 另一个可运行线程分配到可用 LWP 上; 否则原线程继续运行。

在所有上述情况中, 当该线程让出活跃态时, 线程库选择另一个非捆绑的线程进入可运行队列, 并在新的可用的 LWP 上运行它。

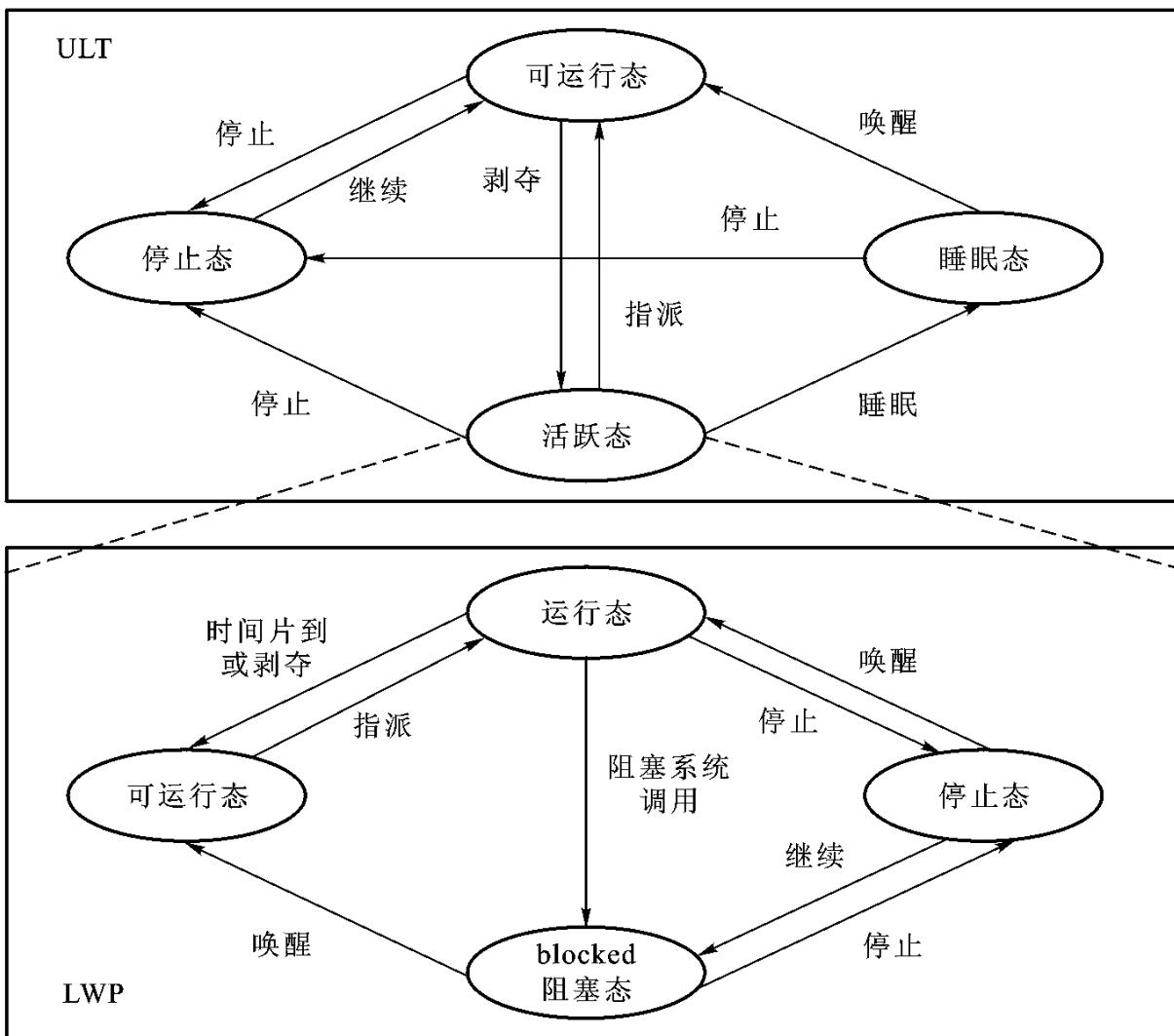


图 2-33 ULT 和 LWP 的状态转换

图 2-32 还显示了 LWP 的状态转换图, 可以把它看作是 ULT 活跃状态的详尽描述, 因为, 当一个 ULT 处于活跃状态时只能捆绑到一个 LWP 上。只有当 LWP 处于运行状态时, 一个处于活跃状态的 ULT 才真正的处于活跃状态并被执行。当一个处于活跃状态的 ULT 调用一个阻塞的系统调用时, 则它对应的 LWP 进入阻塞状态, 这个 ULT 将继续处于活跃状态并继续与相应的 LWP 绑定, 直到线程库显式地做出变动。

下面再讨论 ULT 和 LWP 一一绑定的情况, ULT 和 LWP 的关系只有轻微的不同。例如, 当一个 ULT 等待一个同步事件而进入睡眠状态之后, 相应的 LWP 也必须停止运行, 这种状态转换是通过让 LWP 阻塞在核心层同步变量上实现的。

表 2-2 Solaris 提供了以下线程操作供用户线程程序设计。

| | |
|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| thread_create() | 创建一个新的线程 |
| thread_setconcurrency() | 置并发程度(即 LWP 的数目) |
| thread_exit() | 终止当前进程,收回线程库分配给它的资源 |
| thread_wait() | 阻塞当前线程,直到有关线程退出 |
| thread_get_id() | 获得线程标识符 |
| thread_sigsetmask() thread_sigprocmask() | 置线程信号掩码 |
| Thread_kill() | 生成一个送给指定线程的信号 |
| Thread_stop() | 停止一个线程的执行 |
| Thread_priority() | 置一个线程的优先数 |
| mutex_enter() mutex_exit() mutex_tryenter() | 线程获得互斥锁 线程释放互斥锁 线程忙式等待获得互斥锁 |
| Sem_p() sema_v() sema_tryp() | 计数信号量 P 操作 计数信号量 V 操作 忙式等待 P 操作 |
| rw_enter() rw_exit() rw_tryenter() rw_downgrade() rw_tryupgrade() | 读线程/写线程获得读/写锁 读线程/写线程释放读/写锁 忙式等待获得读/写锁 把 write lock 转换成 read lock 把 read lock 转换成 write lock |
| cv_wait() cv_signal() cv_broadcast() | 线程等待直到条件变量为真 唤醒等待在 cv_wait() 上的一个线程 唤醒等待在 cv_wait() 上的所有线程 |

2.4.5 实例研究:Windows 2000/XP 的进程与线程

1. Windows 2000/XP 中的进程与线程概念

Windows 2000/XP 包括三个层次的执行对象:进程、线程和作业。其中作业是 Windows 2000 开始新引进的,在 Windows NT4 中不存在,它是共享一组配额限制和安全性限制的进程的集合。配额限制如最多进程数、总的和每个进程可用的 CPU 时间、总的和每个进程最大的可用内存;进程是资源的容器,它容纳的资源如主存、打开的文件。线程是可被内核调度的执行实体,它可以被中断,使 CPU 能转向另一线程执行。

Windows 2000/XP 进程设计的目标是提供对不同操作系统环境的支持,具有多任务(多进程)、多线程、支持 SMP、采用了 C/S 模型、能在任何可用 CPU 上运行操作系统等特点。由

内核提供的进程结构和服务相对来说简单、适用,其重要的特性如下:

- 作业、进程和线程是用对象来实现的。
- 一个可执行的进程可以包含一个或多个线程。
- 进程或线程两者均有内在的同步设施。

进程以及它控制和使用的资源的关系如图 2-34 所示。当一个用户首次注册时,Windows 2000/XP 为用户建立一个访问令牌 access token, 它包括安全标识和进程凭证。这个用户建立的每一个进程均有这个访问令牌的拷贝。内核使用访问令牌来验证用户是否具有存取安全对象或在系统上和安全对象上执行受限功能的能力。访问令牌还控制了进程能否改变自己的属性,在这种情况下,安全系统首先决定这是否允许,进而决定进程能否改变自己的属性。

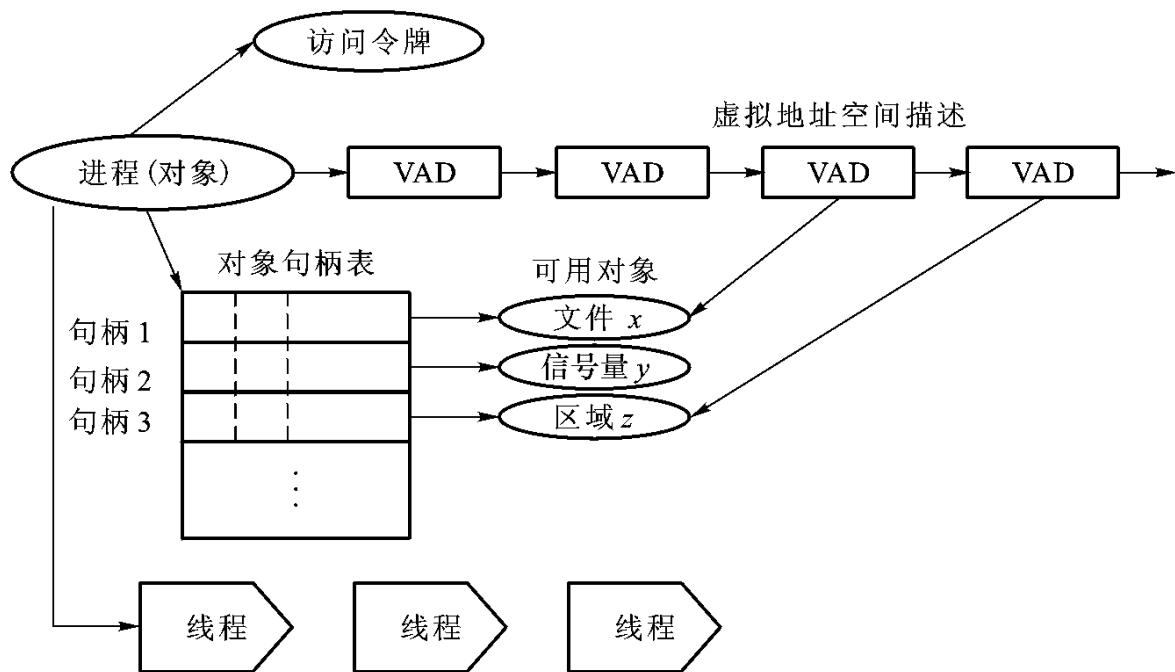


图 2-34 进程以及控制和使用的资源

与进程有关的还有一组当前分配给这个进程的虚拟地址空间块,进程不能直接改变它,必须依靠为进程提供内存分配服务的虚存管理例程。

进程包括一个对象表,其中包括了这个进程与其使用资源之间的联系,通过对象句柄便可对某资源进行引用。图 2-33 包含了线程对象,线程可以访问一个信号量对象和一个共享内存区对象,有一系列分给进程的虚拟地址空间块。

2. 对象及对象管理器

面向对象(object oriented)概念在计算机领域变得十分流行,由于它能为软件带来可重用、可组装、易扩展和易维护的优点,已被许多系统软件设计者所青睐,最新流行的一些操作

系统也都采用了面向对象技术来设计和开发。

对象是一种抽象数据类型(结构),它把内部的数据变量(又称属性)和相关的操作(又称方法)封装起来。体现了信息隐蔽的特性,保护对象的变量不被外界破坏,使得扩充对象的功能和修改对象的实现时不会影响外界和受外界影响,这就大大加强了软件的易维护性。对象中的属性的值表示已知的关于对象所代表的事物的信息,方法则包括那些执行起来会影响对象中属性值的过程。对象是现实世界中实体的抽象,具有共同属性的对象便归为一个对象类(object class),也就是说,它是定义了包含特定类型的对象中的方法与变量的模板。所以,对象类是具有共同属性的对象的抽象,而对象则是对象类的一个实例(instance)。一个实例对象包含了类定义中的属性,对象类中定义的变量在实例对象中都有具体的值。

对象是只通过消息交互的,当一个对象接收到外部来的一个消息时,便触发执行其封装的对应方法。于是该方法对对象封装的变量进行修改,从而,改变了对象的状态。一条消息包括发送消息的对象名、接收消息的对象名、触发的接收对象的方法名及执行所需的参数。对于本地对象,消息触发等同于过程调用,当对象是分布式的,传递消息确实名符其实。由于对象仅能通过消息与其他对象或外部世界交互,这一性质便称为封装性(encapsulation)。封装性能保护对象的变量不被其他对象破坏,同时将对象内部的结构与实现隐藏起来,从而与对象的交互相对简单并可标准化。

对象类具有一个有用的机制叫继承。继承使得可以根据已有的类来定义新的类,新的类称子类(subclass),原有类称超类(superclass)。子类中自动包含了超类中定义的方法和变量,但它可以定义超类中没有的方法和变量,也可以用新的定义和同样的名字来重载在超类中定义的方法和变量。继承机制是循环的,这样可以产生一个对象的类层次结构,一个软件的功能通常可用一个对象类的层次结构来表达。这些对象类可以实例化出许多对象,一个对象是一个单独的软件单元,它们相互交互,共同协作便能实现一个软件所提供的功能。

Windows 2000/XP 是一个基于对象(object-based)的操作系统,在系统中,用对象来表示所有的系统资源。那么,Windows 2000/XP 中定义了多少种对象类呢?由于系统资源都是用对象来表示的,所以,可以认为对象类就是资源类。在 Windows 2000/XP 中,主要定义了两类对象:

(1) 执行体对象 由执行体的各种组件实现的对象,主要用来实现各种外部功能,用户态程序(如各个服务器对象)可以访问执行体对象。具体来说,下列对象类都是执行体的对象:进程、线程、区域、文件、事件、事件对、文件映射、互斥、信号量、计时器、对象目录、符号连接、关键字、端口、存取令牌和终端等。如果把这些对象作进一步分类,那么,执行体创建以下对象:事件对象、互斥对象、信号量对象、文件对象、文件映射对象、进程对象、线程对象和管道对象等。

(2) 内核对象 是由内核实现的一种更原始的对象集合,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。它们对用户态代码是不可见的,仅在执行体内创建和使用,许多执行体对象包含一个或多个内核对象,而内核对象能提供仅能由内核来完成的基本功能。

Windows 2000/XP 每个对象由一个对象头和一个对象体组成。对象头由对象管理器控制,各执行体组件控制它们自己创建的对象类型的对象体。每个对象头都指向打开该对象的进程的列表,同时还有一个叫类型对象的特殊对象,它包含的信息对每一个对象实例是公用的。标准的对象头属性有:

- 对象名 对象为共享进程可见
- 对象目录 表示对象名的层次结构
- 安全描述体 指定对象的安全级
- 配额账 指定打开该对象的进程使用系统资源的限额
- 打开句柄计数器 对象句柄打开的次数
- 打开句柄数据库 列出打开该对象的进程
- 永久/暂时状态 对象不用时,对象名及存储空间可否释放
- 内核/用户模式 对象是否在用户模式下使用的标志
- 对象类型指针 一个指向该对象的类型对象的指针。

对象体的格式和内容随对象类不同而不同。对象体中列出的各对象类的属性有:对象类名、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、等待单个对象和等待多个对象。

Window 2000/XP 通过对象管理器为执行体中的各种内部服务提供一致的和安全的访问手段,它是一个用于创建、删除、保护和跟踪对象的执行体组件,提供了使用系统资源的公共和一致的机制。对象管理器接收到创建对象的系统服务后,要完成以下工作:为对象分配主存;为对象设置安全描述体,以确定谁可使用对象,及访问对象者被允许执行的操作;创建和维护对象目录表;创建一个对象句柄并返回给创建者。

当进程通过名称来创建或打开一个对象时,它会收到一个代表进程访问对象的句柄,通过句柄指向对象比使用名称要快,因为,对象管理器可通过句柄直接找到对象。所有用户态进程只有获得了对象句柄才可以使用对象,句柄作为系统资源的间接指针使用,对象管理器有创建和定位句柄引用对象的专用权限。对象句柄是一个由执行体进程 EPROCESS 所指向的进入进程句柄表(见图 2-34)的索引,其中包含了进程已打开句柄的所有对象的指针。

当进程创建对象或打开一个已有对象的句柄时,必须指定一组期望的访问权限,这时对象管理器将调用安全引用监视程序,并把进程的一组期望的访问权限发送给它。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正在请求的访问类型,如果允许,将返回一

组授予的访问权限,对象管理器会把它们存储在创建的对象句柄中。此后,任何时候,当进程的线程使用句柄时,对象管理器都可以快速检查在句柄中存储的已授权访问权限集是否符合由线程调用的对象服务隐含的用法。

3. 进程对象

进程是作为对象来管理的,由一个对象的通用结构来表示。每个进程对象由属性和封装了的若干可以执行的动作和服务所定义。当接受到适当消息时,进程就执行一个服务,只能通过传递消息给提供服务的进程对象来调用这一服务。用户使用进程对象类(或类型)来建立一个新进程,对进程来说,这个对象类被定义作为一个能够生成新的对象实例的模板,并且在建立对象实例时,属性将被赋值。进程对象的属性包括:进程标识、资源访问令牌、进程基本优先权和默认亲合处理器集合等。进程是对应一个拥有存储器、打开文件表等资源的用户作业或应用程序实体;线程是顺序执行的工作的一个可调度单位,并且它可以被中断,于是处理器可被另一个线程占用。

每一个进程都由一个执行体进程(EPROCESS)块表示。下面给出了 EPROCESS 的结构,它不仅包括进程的许多属性,还包括并指向许多其他相关的属性,如每个进程都有一个或多个执行体线程(ETHREAD)块表示的线程。除了进程环境块 PEB(Process Environment Block,每个进程有一个)存在于进程地址空间中以外,EPROCESS 块及其相关的其他数据结构存在于系统空间中。另外,WIN32 子系统进程 CSRSS 为执行 WIN32 程序的进程保持一个平行的结构,该结构存在于 WIN32 子系统的核心态部分 WIN32K.SYS,在线程第一次调用在核心态实现的 WIN32 USER 或 GDI 函数时被创建。

EPROCESS 块中的有关项目的内容如下:

- 内核进程块 KRPROCESS:公共调度程序对象头、指向进程页面目录的指针、线程调度默认的基本优先级、时间片、相似性掩码、用于进程中线程的总内核和用户时间。
- 进程标识符等:操作系统中惟一的进程标识符、父进程标识符、运行映像的名称、进程正在运行的窗口位置。
- 配额限制:限制非页交换区、页交换区和页面文件的使用,进程能使用的 CPU 时间。
- 虚拟地址空间描述符 VAD:一系列的数据结构,描述了进程虚拟地址空间的状态。
- 工作集信息:指向工作集列表的指针,当前的、峰值的、最小的和最大的工作集大小,上次裁剪时间,页错误计数,内存优先级,交换出标志,页错误历史纪录。
- 虚拟内存信息:当前和峰值虚拟值,页面文件的使用,用于进程页面目录的硬件页表入口。
- 异常/调试端口:当进程的一个线程发生异常或引起调试事件时,进程管理程序发送消息的进程间通信通道。
- 访问令牌:又称安全描述符,指明谁建立的对象,谁能存取对象,谁被拒绝存取该对象。

- 对象句柄表:整个进程的句柄表地址。当进程创建或打开一个对象时,就会得到一个代表该对象的句柄,通过句柄就可以引用进程对象。
- 进程环境块 PEB:映像基址、模块列表、线程本地存储数据、代码页数据、临界区域超时、堆栈的数量、大小、进程堆栈指针、GDI 共享的句柄表、操作系统版本号信息、映像版本号信息、映像进程相似性掩码。

- WIN32 子系统进程块:WIN32 子系统的核心组件需要的进程细节。

操作系统还提供一组用于进程的 WIN32 函数,这些函数就是进程对象提供的服务:

- CreateProcess:使用调用程序的安全标识,创建新的进程及其主线程。
- CreateProcessAsUser:使用交替的安全标识,创建新的进程及其主线程,然后执行指定的 EXE 映像文件。

- OpenProcess:返回指定进程对象的句柄。
- ExitProcess:正常情况下,终止一个进程及其所有线程。
- TerminateProcess:异常情况下,终止一个进程及其所有线程。
- FlushInstructionCache:清空另一个进程的指令高速缓存。
- GetProcessTimes:得到另一个进程的时间信息,描述进程在用户态和核心态所用的时间。

- GetExitCodeProcess:返回另一个进程的退出代码,指出关闭这个进程的方法和原因。
- GetCommandLine:返回传递给进程的命令行字符串。
- GetCurrentProcessID:返回当前进程的 ID。
- GetProcessVersion:返回指定进程希望运行的 Windows 的主要和次要版本信息。
- GetStartupInfo:返回在 CreateProcess 时指定的 STARTUPINFO 结构的内容。
- GetEnvironmentStrings:返回环境块的地址。
- GetEnvironmentVariable:返回一个指定的环境变量。
- GetProcessShutdownParameters:取当前进程的关闭优先级和重试次数。
- SetProcessShutdownParameters:置当前进程的关闭优先级和重试次数。

当应用程序调用 CreateProcess 函数时,将创建一个 WIN32 进程。创建 WIN32 进程的过程在操作系统的 3 个部分中分阶段完成,这三个部分是:WIN32 客户方的 KERNEL32.DLL、Windows 2000/XP 执行体和 WIN32 子系统进程 CSRSS。步骤如下:

- 打开将在进程中被执行的映像文件(.EXE)。
- 创建 Windows 2000/XP 执行体进程对象。
- 创建初始线程(堆栈、描述表、执行体线程对象)。
- 通知 WIN32 子系统已经创建了一个新的进程,以便它可以设置新的进程和线程。
- 启动初始线程的执行(除非指定了 CREATE_SUSPENDED 标志)。

- 在新进程和线程的描述表中,完成地址空间的初始化,加载所需的 DLL,并开始程序的执行。

4. 线程对象

Windows 2000/XP 线程是内核级线程,它是 CPU 调度的独立单位。每一个线程都由一个执行体线程(ETHREAD)块表示。图 2-35 给出了 ETHREAD 的结构。除了线程环境块(TEB)存在于进程地址空间中以外,ETHREAD 块及其相关的其他数据结构存在于系统空间中。另外,WIN32 子系统进程 CSRSS 为执行 WIN32 程序的线程保持一个平行的结构,该结构存在于 WIN32 子系统的核心态部分 WIN32K.SYS。

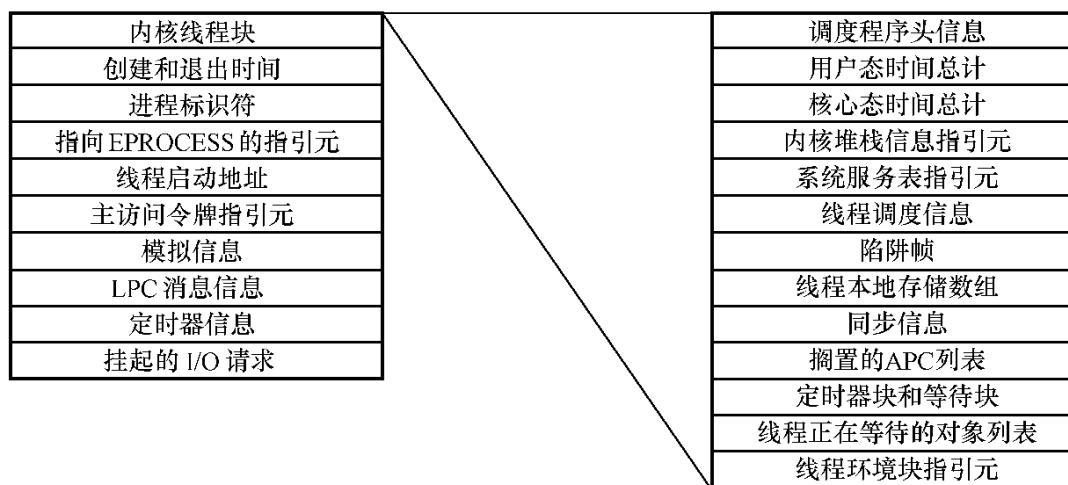


图 2-35 THRED 块的结构

ETHRED 块中的有关项目的内容如下:

- 创建和退出时间:线程的创建和退出时间。
- 进程识别信息:进程标识符和指向 EPROCESS 的指引元。
- 线程启动地址:线程启动例程的地址。
- LPC 消息信息:线程正在等待的消息 ID 和消息地址。
- 挂起的 I/O 请求:挂起的 I/O 请求数据包列表。
- 调度程序头信息:指向标准的内核调度程序对象。
- 执行时间:在用户态运行的时间总计和在核心态运行的时间总计。
- 内核堆栈信息指引元:内核堆栈的栈底和栈顶信息。
- 系统服务表指引元:指向系统服务表的指针。
- 调度信息:基本的和当前的优先级、时间片、相似性掩码、首选处理器、调度状态、冻结计数、挂起计数。

操作系统提供了一组用于线程的 WIN32 函数,这些函数就是线程对象提供的服务:

- CreateThread: 创建新线程。
- OpenThread: 打开线程
- CreateRemoteThread: 在另一个进程创建线程。
- ExitThread: 退出当前线程。
- TerminateThread: 终止线程。
- GetExitCodeThread: 返回另一个线程的退出代码。
- GetThreadTimes: 返回另一个线程的定时信息。
- GetThreadSelectorEntry: 返回另一个线程的描述符表入口。
- GetThreadContext: 返回线程的 CPU 寄存器。
- SetThreadContext: 更改线程的 CPU 寄存器。

当应用程序调用 CreateThread 函数创建一个 WIN32 线程的具体步骤如下：

- 在进程地址空间为线程创建用户态堆栈。
- 初始化线程的硬件描述表。

• 调用 NtCreateThread 创建处于挂起状态的执行体线程对象。包括：增加进程对象中的线程计数，创建并初始化执行体线程块，为新线程生成线程 ID，从非页交换区分配线程的内核堆栈，设置 TEB，设置线程起始地址和用户指定的 WIN32 起始地址，调用 KeInitializeThread 设置 KTHREAD 块，调用任何在系统范围内注册的线程创建注册例程，把线程访问令牌设置为进程访问令牌并检查调用程序是否有权创建线程。

- 通知 WIN32 子系统已经创建了一个新线程，以便它可以设置新的进程和线程。
- 线程句柄和 ID 被返回到调用程序。
- 除非调用程序用 CREATE_SUSPEND 标识设置创建线程，否则线程将被恢复以便调度执行。

线程是 Windows 2000/XP 操作系统的最终调度实体，如图 2-36 示，它可能处于以下 7 个状态之一：

- 就绪态——线程已获得除 CPU 外的所有资源，等待被调度去执行的状态，内核的调度程序维护所有就绪线程队列，并按优先级次序调度。
- 准备态——已选中下一个在一个特定处理器上运行的线程，此线程处于该状态等待描述表切换。如果准备态线程的优先级足够高，则可以从正在运行的线程手中抢占处理器，否则将等待直到运行线程等待或时间片用完。系统中每个处理器上只能有一个处于准备态的线程。

- 运行态——内核执行线程切换，准备态线程进入运行态，并开始执行。直到它被剥夺、或用完时间片、或阻塞、或终止为止。在前两种情况下，线程进入到就绪态。
- 等待态——线程进入等待态是由于以下原因：1) 出现了一个阻塞事件（如，I/O）；2)

等待同步信号;3)环境子系统要求线程挂起自己。当等待条件满足时,且所有资源可用,线程就进入就绪态。

- 过渡态——一个线程完成等待后准备运行,但这时资源不可用,就进入过渡态,例如,线程的内核堆栈被调出内存。当资源可用时(内核堆栈被调入内存),过渡态线程进入就绪态。

- 终止态——线程可被自己、其他线程、或父进程终止,这时进入终止态。一旦结束工作完成后,线程就可从系统中移去。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。

- 初始态——线程创建过程中所处状态,创建完成后,该线程被放入就绪队列。

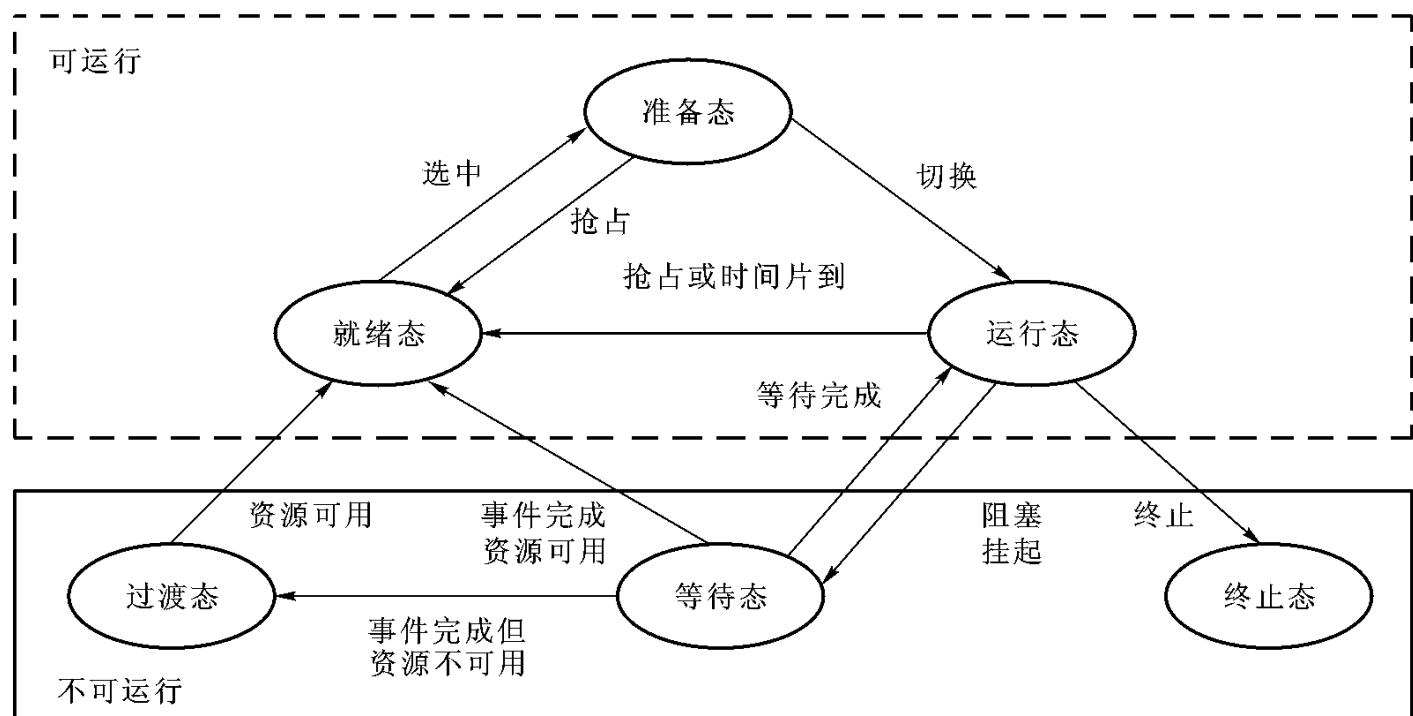


图 2-36 Windows 2000/XP 的线程状态

5. 作业对象

Windows 2000/XP 包含一个被称为“作业”的进程模式扩展。作业对象是一个可命名、保护、共享的对象,它能够控制与作业有关的进程属性。作业对象的基本功能是允许系统将进程组看作是一个单元,对其进行管理和操作。有些时候,作业对象可以补偿 Windows NT4 在结构化进程树方面的缺陷。作业对象也为所有与作业有关的进程和所有与作业有关但已被终止的进程记录基本的账号信息。

作业对象包含一些对每一个与该作业有关的进程的强制限制,这些限制包括:

- 默认工作集的最小值和最大值。
- 作业范围用户态 CPU 时限。

- 每个进程用户态 CPU 时限。
- 活动进程的最大数目。
- 作业处理器相似性。
- 作业进程优先级类。

用户也能够在作业中的进程上设置安全性限制。用户可以设置一个作业,使得每一个进程运行在同样的作业范围的访问令牌下。然后,用户就能够创建一个作业,限制进程模仿或创建其访问令牌中包括本地管理员组的进程。另外,用户还可以应用安全筛选,当进程中的线程包含在作业模仿客户机线程中时,将从模仿令牌中删除特定的特权和安全 ID(SID)。

最后,用户也能够在作业中的进程上设置用户接口限制。其中包括:限制进程打开作业以外的线程所拥有的窗口句柄,对剪贴板的读取或写入,通过 WIN32 SystemParameterInfo 函数更改某些用户接口系统参数等。

一个进程只能属于一个作业,一旦进程建立,它与作业的联系便不能中断;所有由进程创建的进程和它们的后代也和同样的作业相联系。在作业对象上的操作会影响与作业对象相联系的所有进程。

有关作业对象的 WIN32 函数包括:

- CreateJobObject: 创建作业对象。
- Open JobObject: 通过名称打开现有的作业对象。
- AssignProcessToJobObject: 添加一个进程到作业。
- TerminateJobObject: 终止作业中的所有进程。
- SetInformationToJobObject: 设置限制。
- QueryInformationToJobObject: 获取有关作业的信息,如:CPU 时间、页错误技术、进程的数目、进程 ID 列表、配额或限制、安全限制等。

2.5 处理器调度

在计算机系统中,可能同时有数百个批处理作业存放在磁盘的作业队列中,或者有数百个终端与主机相连接,这样一来内存和处理器等资源便供不应求。如何从这些作业中挑选作业进入主存运行、如何在进程之间分配处理器时间,无疑是操作系统资源管理中的一个重要问题。处理器调度用来完成涉及处理器分配的工作。

2.5.1 处理器调度的层次

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,可能会经历如图

2-37 示的调度过程。处理器调度可以分为三个级别：高级调度、中级调度和低级调度。

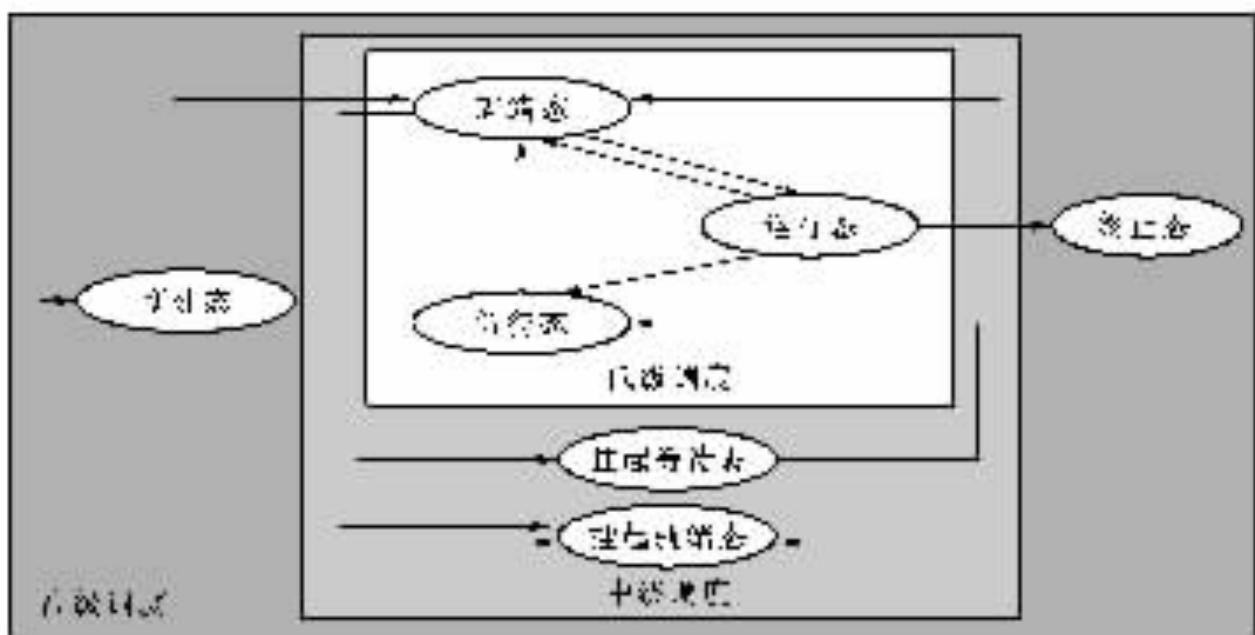


图 2-37 调度的层次

上述三级调度中，低级调度是各类操作系统必须具有的功能；在纯粹的分时或实时操作系统中，通常不需要配备高级调度；在分时系统或具有虚拟存储器的操作系统中，为了提高内存利用率和作业吞吐量，专门引进了中级调度。图 2-38 给出了三级调度功能与进程状态转换的关系。高级调度发生在新进程的创建中，它决定一个进程能否被创建，或者是创建后能否被置成就绪状态，以参与竞争处理器资源从而获得运行；中级调度反映到进程状态上就是挂起和解除挂起，它根据系统的当前负荷情况决定停留在主存中进程数；低级调度则是决定哪一个就绪进程或线程占用 CPU。

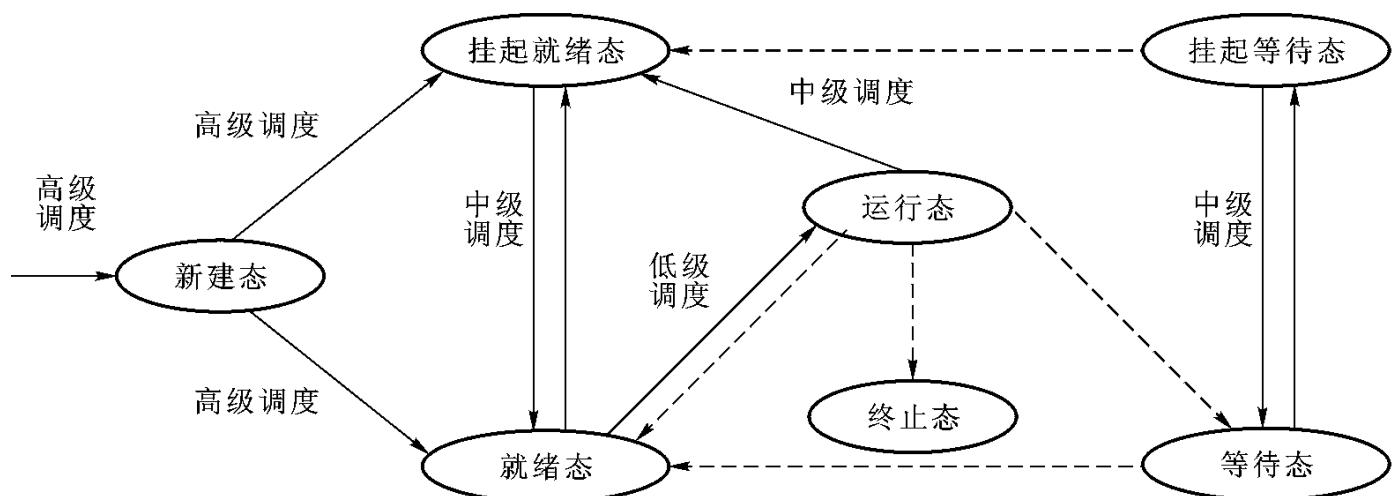


图 2-38 处理器调度与进程状态转换

2.5.2 高级调度 (High Level Scheduling)

高级调度:又称作业调度、长程调度 (Long-term Scheduling) 在多道批处理操作系统中, 作业是用户要求计算机系统完成的一项相对独立的工作, 新提交的作业被输入到磁盘, 并保存在一个批处理后备作业队列中。高级调度将按照系统预定的调度策略决定把后备队列作业中的部分满足其资源要求的作业调入主存, 为它们创建进程, 分配所需资源。为作业做好运行前的准备工作并启动它们运行, 当作业完成后还为它做好善后工作。在批处理操作系统中, 作业首先进入系统在辅存上的后备作业队列等候调度, 因此, 作业调度是必须的, 它执行的频率较低, 并和到达系统的作业的数量与速率有关。

高级调度程序控制多道程序的道数, 调度选择进入主存的作业越多, 每个作业获得的 CPU 时间就越少, 为了给进入主存的作业提供满意的服务, 有时需要限制多道程序的道数。每当一个作业执行完成撤离时, 高级调度会决定增加一个或多个作业到主存, 此外, 如 CPU 空闲时间超过一定阈值, 系统也会引出高级调度调度后备作业。图 2-39 展示了处理器的调度模型。

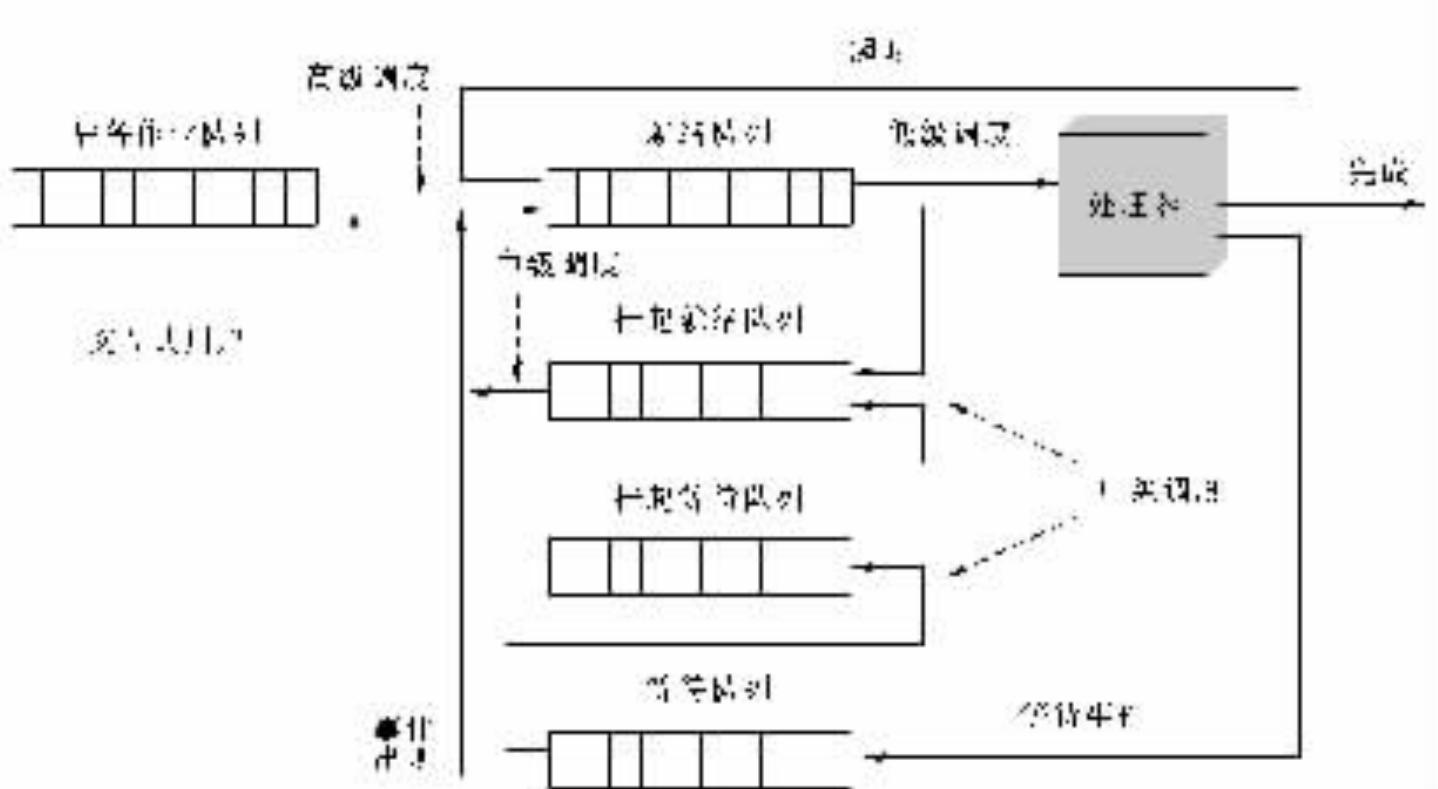


图 2-39 处理器的调度模型

对于分时操作系统来说, 高级调度决定:1) 是否接受一个终端用户的连接;2) 一个交互作业能否被计算机系统接纳并构成进程, 通常系统将接纳所有授权用户, 直到系统饱和为

止;3)一个新建态的进程是否能够立即加入就绪进程队列。有的分时操作系统虽没有配置高级调度程序,但上述的调度功能是必须提供的。

2.5.3 中级调度 (Medium Level Scheduling)

中级调度又称平衡负载调度,中程调度 (Medium-term Scheduling)。它决定主存储器中所能容纳的进程数,这些进程将允许参与竞争处理器和有关资源,而有些暂时不能运行的进程被调出主存,这时这个进程处于挂起状态,当进程具备了运行条件,且主存又有空闲区域时,再由中级调度决定把一部分这样的进程重新调回主存工作。中级调度根据存储资源量和进程的当前状态来决定辅存和主存中的进程的对换,它所使用的方法是通过把一些进程换出主存,从而,使之进入“挂起”状态,不参与低级调度,起到短期平滑和调整系统负荷的作用。

2.5.4 低级调度 (Low Level Scheduling)

低级调度又称进程调度(或线程调度)、短程调度 (Short _ term Scheduling)。它的主要功能是按照某种原则决定就绪队列中的哪个进程或内核级线程能获得处理器,并将处理器出让给它进行工作。低级调度中执行分配 CPU 的程序称分派程序 (dispatcher),它是操作系统最为核心的部分,执行十分频繁。低级调度策略优劣直接影响到整个系统的性能,因而,这部分代码要求精心设计,并常驻内存工作。有两类低级调度方式:

第一类称剥夺方式:当一个进程正在处理器上执行时,系统可以根据规定的原则剥夺分配给它的处理器,而把处理器分配给其他进程使用。常用的有两种剥夺原则。前者高优先级进程或线程可以剥夺低优先级进程或线程运行。后者当运行进程时间运用完后被剥夺处理器。

第二类称非剥夺方式:一旦某个进程或线程开始执行后便不再出让处理器,除非该进程或线程运行结束或发生了某个事件不能继续执行。

剥夺式策略的开销比非剥夺式策略来得大,但由于可以避免一个进程或线程长时间独占处理器,因而,能给进程或线程提供较好的服务。对不同调度类型,相应的调度算法也不同,低级调度的核心是采用何种算法把处理器分配给进程或线程。

2.5.5 选择调度算法的原则

无论是哪一个层次的处理器调度,都由操作系统的调度程序 (scheduler) 实施,而调度程序所使用的算法称为调度算法 (scheduling algorithm),不同类型的操作系统,其调度算法通常不同。

在讨论具体的调度算法之前,首先,讨论调度所要达到的总的目标。设计调度程序首先

要考虑的是确定策略,然后,才是提供机制。一个好的调度算法应该考虑很多因素,其中可能有:

- 资源利用率——使得 CPU 或其他资源的使用率尽可能高且能够并行工作,CPU 的利用率 = CPU 有效工作时间/CPU 总的运行时间,而 CPU 总的运行时间为 CPU 有效工作时间 + CPU 空闲等待时间。
- 响应时间——交互式进程从提交一个请求(命令)到接收到响应之间的时间间隔称响应时间。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一个重要指标。
- 周转时间——批处理用户从作业提交给系统开始,到作业完成为止的时间间隔称作作业周转时间,应使作业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个重要指标。
- 吞吐率——使得单位时间内处理的作业数尽可能多。
- 公平性——确保每个用户每个进程获得合理的 CPU 份额或其他资源份额,不会出现饿死情况。

当然,这些目标本身就存在着矛盾之处,操作系统在设计时必须根据其类型的不同进行权衡,以达到较好的效果。下面着重看一下批处理系统的调度性能指标。

批处理系统的调度性能主要用作业周转时间和作业带权周转时间来衡量,此时间越短,则系统效率越高,作业吞吐能率越强。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,作业的周转时间 t_i 为:

$$t_i = t_f - t_s$$

实际上,它是作业在系统里的等待时间与运行时间之和。从操作系统来说,为了提高系统的性能,要让若干个用户的平均作业周转时间和平均带权作业周转时间最小。

$$\text{平均作业周转时间 } T = (\sum t_i) / n$$

如果作业 i 的周转时间为 t_i ,所需运行时间为 t_k ,则称 $w_i = t_i / t_k$ 为该作业的带权周转时间。因为, t_i 是等待时间与运行时间之和,故带权周转时间总大于 1。

$$\text{平均作业带权周转时间 } W = (\sum w_i) / n$$

通常,用平均作业周转时间来衡量对同一作业流施行不同作业调度算法时它们呈现的调度性能;用平均作业带权周转时间来衡量对不同作业流施行同一作业调度算法时它们呈现的调度性能。这两个数值均越小越好。

2.6 批处理作业的管理与调度

2.6.1 作业和进程的关系

作业(JOB)是用户提交给操作系統计算的一个独立任务。一般每个作业必须经过若干个相对独立又相互关联的顺序加工步骤才能得到结果,其中,每一个加工步骤称一个作业步(Job Step),例如,一个作业可分成“编译”、“连结装配”和“运行”三个作业步,往往上一个作业步的输出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,一个作业从提交给系统,直到运行结束获得结果,要经过提交、收容、执行和完成四个阶段。当收容态作业被作业调度程序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(根)进程完成其计算任务。若干个作业进入系统并依次存放在后备存储器上形成了输入的作业流,在系统控制下逐个取出并运行便形成了处理的作业流。进程是对系统中已提交完毕并选中运行的任务(程序)的执行过程,也是为完成作业任务向系统申请和分配资源的基本单位。它在“运行”、“就绪”、“等待”等多个状态的交替之中,在CPU上推进,最终完成一个程序的任务。用户进程在执行过程中可生成作业步子进程,当然子进程还可以生成其他的子进程,这些进程并发执行,高效地协作完成用户作业的任务。在多道程序设计环境中,由于多个作业可同时被投入运行,因此,并发系统中,某一时刻并发执行的进程是相当多的(如UNIX中有几十个),操作系统要负责众多进程的协调和管理。一个进程中还可以孵化出多个线程,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,进程无事可干,没有进程,作业任务没法完成。作业概念更多地用在批处理操作系统,而进程则可以用在各种多道程序设计系统。

2.6.2 批处理作业的管理

多道批处理操作系统采用脱机控制方式,它提供一个作业控制语言,用户使用作业控制语言书写作业说明书,它是按规定格式书写的一个文件,把用户对系统的各种请求和对作业的控制要求集中描述,并与程序和数据一起提交给系统(管理员)。计算机系统成批接受用户作业输入,把它们放到输入井,然后,在操作系统的管理和控制下执行。

多道批处理操作系统具有独立的作业管理模块,为了有效管理作业,必须像进程管理一样为每一个作业建立作业控制块JCB(Job Control Block)。JCB通常是在批作业进入系统时,

由 Spooling 系统建立的,它是作业存在于系统的标志,作业撤离时,JCB 也被撤销。JCB 的主要内容是从用户作业说明书中获得,包括:作业情况(用户名、作业名、语言名等),资源需求(估计 CPU 运行时间、最迟截止期、主存量、设备类型/台数、文件数和数据量、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、已运行时间等),作业控制(优先数、联机/脱机控制、操作顺序、出错处理等),作业类型(CPU 繁忙型、I/O 繁忙型、批量型、终端型)等信息。

当一个作业被操作系统接受,就必须创建一个作业控制块,并且这个作业在它的整个生命周期中将顺序地处于以下 4 个状态:

- 输入状态:此时作业的信息正在从输入设备上预输入。
- 后备状态:此时作业预输入结束但尚未被选中执行。
- 执行状态:作业已经被选中并构成进程去竞争处理器资源以获得运行。
- 完成状态:作业已经运行结束,甚至已经撤离,但正在等待缓输出运算结果。

多道批处理操作系统的处理器调度至少应该包括作业调度和低级调度两个层次。作业调度属于高级调度层次,处于后备状态的作业在系统资源满足的前提下可以被选中,从而,进入主存计算。只有处于执行状态的作业才真正构成进程获得运行和计算的机会。

作业调度选中了一个作业且把它装入主存储器时就为该作业创建一个用户进程。这些进程将在低级调度的控制下占有处理器运行。为充分利用处理器,往往可以把多个作业同时装入主存储器,这样就会同时有多个用户进程,这些进程都要竞争处理器。

所以,进入计算机系统的作业只有经过两级调度后才能占用处理器。第一级是作业调度,使作业进入主存储器,同时生成相应于作业的进程;第二级是低级调度,使进程占用处理器。作业调度与低级调度的配合能实现多道作业的同时执行,作业调度与低级调度关系及作业和进程的状态转换如图 2-40。

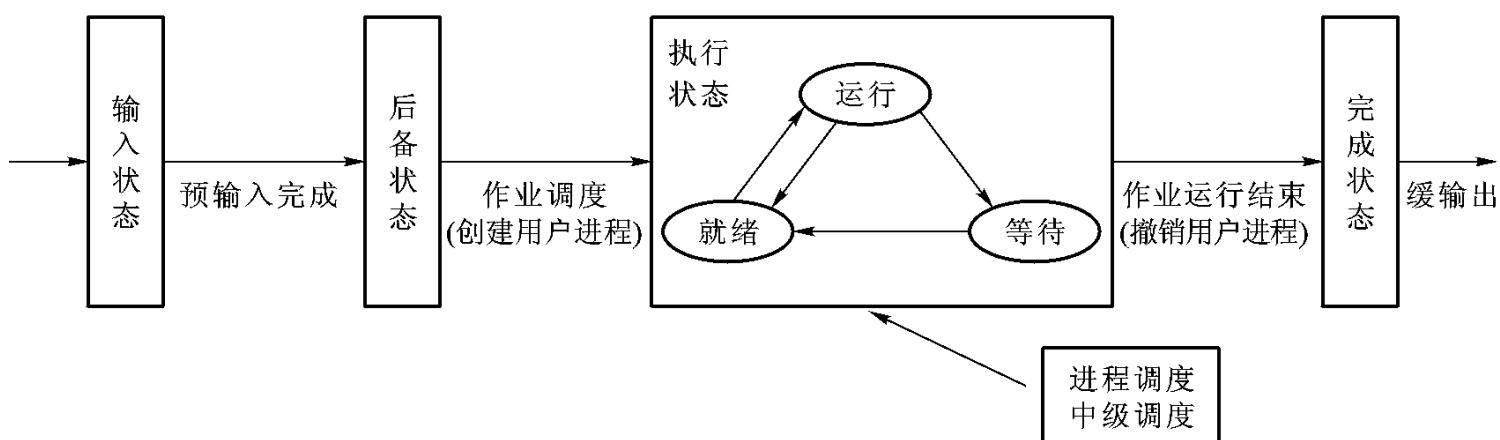


图 2-40 作业调度与低级调度关系及作业和进程状态

2.6.3 批处理作业的调度

对成批进入系统的用户作业,根据作业控制块信息,按一定的策略选取若干个作业使它们可以去获得处理器运行,这项工作称为作业调度。上面已经说过,对于每个用户来说总希望自己的作业的周转时间 T_i 尽可能的小,最理想的情况是进入系统后能立即投入运行,即希望 T_i 等于作业执行的时间。对于系统来说,则希望进入系统的作业的平均周转时间: $T = (T_1 + T_2 + \dots + T_n)/n$ 尽可能的小,使得 CPU 的利用率尽量高。于是,每个计算机系统都必须选择适当的作业调度算法,既考虑用户的要求又要有利于系统效率的提高。当选中一个作业后,首先要建立此作业的用户进程,同时为其分配所需资源,接着就可以投入运行了。当一个作业执行结束进入完成状态时,负责回收资源,撤销它的作业控制块。

2.6.4 作业调度算法

1. 先来先服务算法 FCFS(First Come, First Served)

先来先服务算法是按照作业进入系统的作业后备队列的先后次序来挑选作业,先进入系统的作业优先被挑选。这是一种非剥夺式算法,容易实现,但效率不高。只顾及到作业等候时间,而没考虑作业要求服务时间的长短。显然这不利于短作业而优待了长作业,或者说有利于 CPU 繁忙型作业而不利于 I/O 繁忙型作业。有时为了等待长作业的执行,而使短作业的周转时间变得很大。从而,平均周转时间也变大。

例如,下面三个作业同时到达系统并立即进入调度:

| 作业名 | 所需 CPU 时间 |
|------|-----------|
| 作业 1 | 28 |
| 作业 2 | 9 |
| 作业 3 | 3 |

假设系统中没有其他作业,现采用 FCFS 算法进行调度,那么,三个作业的周转时间分别为: 28. 37 和 40,因此,

$$\text{平均作业周转时间 } T = (28 + 37 + 40)/3 = 35$$

若这三个作业调度顺序改为作业 2.1.3, 平均作业周转时间缩短为约 29。如果三个作业调度顺序改为作业 3. 2. 1, 则平均作业周转时间缩短为约 18。由此可以看出, FCFS 调度算法的平均作业周转时间与作业提交及调度的顺序有关。

2. 最短作业优先算法 SJF(Shortest Job First)

最短作业优先算法是以进入系统的作业所要求的 CPU 时间长短为标准,总是选取估计

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, 它克服了 FCFS 偏爱长作业的缺点, 易于实现, 但效率也不高。它的主要弱点: 一是需要预先知道作业所需的 CPU 时间, 这个估计值很难精确, 如果程序员估计过低, 系统就可能提前终止该作业; 二是忽视了作业等待时间, 由于系统不断地接受新作业, 而作业调度又总是选择计算时间短的作业投入运行, 因此, 使进入系统时间早但计算时间长的作业等待时间过长, 会出现饥饿现象; 三是尽管减少了对长作业的偏爱, 但由于缺少剥夺机制, 对分时、实时处理仍然很不理想。

例如, 若有以下四个作业同时到达系统并立即进入调度:

| 作业名 | 所需 CPU 时间 |
|------|-----------|
| 作业 1 | 9 |
| 作业 2 | 4 |
| 作业 3 | 10 |
| 作业 4 | 8 |

假设系统中没有其他作业, 现对它们实施 SJF 调度算法, 这时的作业调度顺序为作业 2、4、1、3, 则:

$$\text{平均作业周转时间 } T = (4 + 12 + 21 + 31) / 4 = 17$$

$$\text{平均带权作业周转时间 } W = (4/4 + 12/8 + 21/9 + 31/10) / 4 = 1.98$$

如果对它们施行 FCFS 调度算法, 则:

$$\text{平均作业周转时间 } T = (9 + 13 + 23 + 31) / 4 = 19$$

$$\text{平均带权作业周转时间 } W = (9/9 + 13/4 + 23/10 + 31/8) / 4 = 2.51$$

由此可见, SJF 的平均作业周转时间比 FCFS 要小, 故它的调度性能比 FCFS 好。但实现 SJF 调度算法需要知道作业所需运行时间, 否则调度就没有依据, 作业运行时间只知道估计值, 要精确知道一个作业的运行时间是办不到的。

SJF 算法是非抢占式的, 可以改进成抢占式的调度算法。当一个作业正在执行时, 一个新作业进入就绪状态, 如果新作业需要的 CPU 时间比当前正在执行的作业剩余下来还需要的 CPU 时间短, 抢占式短作业优先调度算法强行赶走当前正在执行的作业, 这种方式叫最短剩余时间优先 SRTF(Shortest Remaining Time First) 算法。此算法不但适用于作业调度, 同样也适用于进程调度。下面来看一个例子, 假如现有四个就绪作业其到达系统和所需 CPU 时间如下:

| 作业名 | 到达系统时间 | 所需 CPU 时间(毫秒) |
|------|--------|---------------|
| 作业 1 | 0 | 8 |
| 作业 2 | 1 | 4 |
| 作业 3 | 2 | 9 |
| 作业 4 | 3 | 5 |

| | | | | |
|----------------|----------------|----------------|----------------|----------------|
| J ₁ | J ₂ | J ₄ | J ₁ | J ₃ |
| 0 | 1 | 5 | 10 | 17 |

26

Job1 从 0 开始执行, 这时系统就绪队列仅有一个作业。Job2 在时间 1 到达, 而 Job1 的剩余时间(7 ms)大于 JOB2 所需时间(4 ms), 所以, Job1 被剥夺, Job2 被调度执行。这个例子采用 SRTF 的平均等待时间是 $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)) / 4 = 26 / 4 = 6.5$ ms。如果采用非抢占式 SJF 调度, 那么, 平均等待时间是 7.75 ms。

3. 响应比最高者优先(HRRF)算法

先来先服务算法与最短作业优先算法都是比较片面的调度算法。先来先服务算法只考虑作业的等候时间而忽视了作业的计算时间, 而最短作业优先算法恰好与之相反, 它只考虑用户估计的作业计算时间而忽视了作业的等待时间。响应比最高者优先算法(Highest Response Ratio First)是介乎这两种算法之间的一种折衷的策略, 既考虑作业等待时间, 又考虑作业的运行时间, 这样既照顾了短作业又不使长作业的等待时间过长, 改进了调度性能。缺点是每次计算各道作业的响应比会有一定的时间开销, 需要估计期待的服务时间, 性能要比 SJF 略差。这里把作业进入系统后的等待时间与估计计算时间之和称为作业的响应时间, 作业的响应时间除以作业估计计算时间称作响应比, 现定义:

$$\text{响应比} = \text{作业响应时间} / \text{作业估计计算时间} = 1 + \text{作业等待时间} / \text{作业估计计算时间}$$

每当调度一个作业运行时, 都要计算后备作业队列中每个作业的响应比, 选择响应比最高者投入运行。显然, 计算时间短的作业容易得到较高的响应比, 因为, 这时分母较小, 使得 HRRF 较高, 因此, 本算法是优待短作业的。但是, 如果一个长作业在系统中等待的时间足够长后, 由于它的年龄较大使得分子足够大, 则 HRRF 较大, 那么, 它也将获得足够高的响应比, 从而可以被选中执行, 不至于长时间地等待下去, 饥饿的现象便不会发生。

例如, 若有以下四个作业先后到达系统进入调度:

| 作业名 | 到达系统时间 | 所需 CPU 时间(毫秒) |
|------|--------|---------------|
| 作业 1 | 0 | 20 |
| 作业 2 | 5 | 15 |
| 作业 3 | 10 | 5 |
| 作业 4 | 15 | 10 |

假设系统中没有其他作业, 现对它们实施 SJF 调度算法, 这时的作业调度顺序为作业 1、3、4、2,

平均作业周转时间 $T = (20 + 25 + 35 + 50) / 4 = 32.5$

平均带权作业周转时间 $W = (20/20 + 25/5 + 35/10 + 50/15) / 4 = 3.2$

如果对它们施行 FCFS 调度算法, 这时的作业调度顺序为作业 1、2、3、4,

平均作业周转时间 $T = (20 + 35 + 40 + 50) / 4 = 36.25$

平均带权作业周转时间 $W = (20/20 + 35/15 + 40/5 + 50/10) / 4 = 4.1$

如果对这个作业流执行 HRRF 调度算法:

- 开始时只有作业 1, 作业 1 被选中, 执行时间 20;

- 作业 1 执行完毕后, 响应比依次为 $1 + 15/15$ 、 $1 + 10/5$ 、 $1 + 5/10$, 作业 3 被选中, 执行时间 5;

- 作业 3 执行完毕后, 响应比依次为 $1 + 20/15$ 、 $1 + 10/10$, 作业 2 被选中, 执行时间 15;
- 作业 2 执行完毕后, 作业 4 被选中, 执行时间 10;

平均作业周转时间 $T = (20 + 25 + 40 + 50) / 4 = 33.75$

平均带权作业周转时间 $W = (20/20 + 25/5 + 40/15 + 50/10) / 4 = 3.4$

可见 HRRF 的性能界于 SJF 和 FCFS 之间。

4. 优先数法

这种算法是根据确定的优先数来选取作业, 每次总是选择优先数高的作业。规定用户作业优先数的方法是多种多样的。一种是由用户自己提出作业的优先数, 称作外部优先数法。有的用户为了自己的作业尽快的被系统选中就设法提高自己作业的优先数, 这时系统可以规定优先数越高则需付出的计算机使用费就越多, 以作限制。另一种是由系统综合考虑有关因素来确定用户作业的优先数, 称作内部优先数法。例如, 根据作业的缓急程度; 作业的类型; 作业计算时间的长短、I/O 量的多少、资源申请情况等来确定优先数。确定优先数时各因素的比例应根据系统设计目标来分析这些因素在系统中的地位而决定。

5. 分类调度算法

分类调度算法预先按一定的原则把作业划分成若干类, 以达到均衡使用操作系统资源和兼顾大小作业的目的。分类原则包括: 作业计算时间、对内存的需求、对外围设备的需求等。作业调度时还可以为每类作业设置优先级, 从而, 照顾到同类作业中的轻重缓急。

6. 用磁带与不用磁带的作业搭配

这种算法将需要使用磁带机的作业分开来。在作业调度时, 把使用磁带机的作业和不使用磁带机的作业搭配挑选。在不使用磁带机的作业执行时, 可预先通知操作员将下一批作业要用的磁带预先装上, 这样可使要用磁带机的作业在执行时省去等待装磁带的时间。显然, 这对缩短系统的平均周转时间是有益的。

2.7 低 级 调 度

2.7.1 低级调度的功能

低级调度负责动态地把处理器分配给进程或内核级线程。操作系统中实现低级调度的程序称为低级调度程序,或分派程序(dispatcher)。那么,当进程或线程正常运行时,为什么会出现低级调度的需求呢?这是因为一方面正在执行的进程或线程运行结束或由于某种事件而无法继续运行下去,另一方面可能出现更高优先级进程或线程,需要剥夺当前占有者占用的处理器,这些原因都可能引起再次低级调度。低级调度的主要功能是:

- 记录进程(内核级线程)的状态。这个信息一般记录在一个进程(内核级线程)的进程控制块(线程控制块)内。
- 决定某个进程(内核级线程)什么时候获得处理器,以及占用多长时间。
- 把处理器分配给进程(内核级线程)。即进行进程(内核级线程)上下文切换,把选中进程(内核级线程)的进程控制块(线程控制块)内有关现场的信息如程序状态字、通用寄存器等内容送入处理器相应的寄存器中,从而,让它占用处理器运行。
- 收回处理器。将处理器有关寄存器内容送入该进程(内核级线程)的进程控制块(线程控制块)内的相应单元,从而,使该进程让出处理器。

低级调度的最初对象是传统操作系统中的进程,随着现代操作系统引入了多线程技术,进程演变成资源分配和管理的单位。从而,进程只作为中级调度的对象,内核级线程则替代进程成为低级调度的对象。适用于进程调度的算法一般都适用于内核级线程的调度,第二小节首先介绍这些通用的调度算法。当然内核级线程的引进从某种程度上是为了提供并行性,因此,第四小节将介绍为适应并行性而进一步扩充的多处理器调度算法,它们的主要涉及对象是并行执行的内核级线程,当然同样适用于并行执行的进程。

因此,在下面的讨论中,并不区分进程和内核级线程,在第二小节中统称为进程,而第四小节中统称为线程。值得指出的是,用户级线程的调度是应用程序自己的事,虽然操作系统或计算机语言的程序库会提供一些缺省的算法,但事实上它们与操作系统无关。在纯用户级多线程策略中,低级调度的对象依然是进程;而在混合策略中,低级调度的对象是内核级线程。

2.7.2 低级调度算法

低级调度策略很多,现介绍如下几种:

1. 先来先服务算法

先来先服务算法是按照进程进入就绪队列的先后次序来分配处理器。先进入就绪队列的进程优先被挑选,运行进程一旦占有处理器将一直运行下去直到运行结束或被阻塞,这是一种非剥夺式调度。这种算法容易实现,但效率不高,显然不利于 I/O 频繁的进程。

2. 时间片轮转调度

轮转法调度也称之为时间片调度。具体做法是调度程序每次把 CPU 分配给就绪队列首进程使用所定的时间(例如 100 ms),这样的一段时间称为一个时间片,就绪队列中的每个进程轮流地运行一个这样的时间片。当这个时间片结束时,就强迫当前进程让出处理器,让它排列到就绪队列的尾部,等候下一轮调度。实现这种调度要使用一个间隔时钟。当一个进程开始运行时,就将时间片的值置入间隔时钟内,当发生间隔时钟中断时,就表明该进程连续运行的时间已超过一个规定的时间片。此时,中断处理程序就通知处理器调度进行处理器的切换工作。这种调度策略可以防止那些很少使用外围设备的进程过长的占用处理器,导致要使用外围设备的那些进程没有机会去启动外围设备。

最常用的轮转法是基本轮转法。它要求每个进程轮流地运行相同的一个时间片。在分时系统中,这是一种较简单又有效的调度策略。一个分时系统有许多终端设备,终端用户在各自的终端设备上同时使用计算机,如果某个终端用户的程序长时间的占用处理器,势必使其他终端用户的要求不能得到及时得到响应。一般来说分时系统的终端用户提出要求后到计算机响应给出回答的时间只能是几秒钟,这样才能使终端用户感到满意。采用基本轮转的调度策略可以使系统及时响应。例如,一个分时系统有 10 个终端,如果每个终端用户进程的时间片为 100 ms,那么,粗略地说,每个终端用户在每秒钟内可以得到大约 100 ms 的处理器时间。如果对于终端用户的每个要求,处理器花费 300 ms 的时间就可以给出回答时。那么,终端响应的时间大致就在 3 s 左右,这样可算得上及时响应了。

基本轮转法的策略可以略加修改。例如,对于不同的进程给以不同的时间片;时间片的长短可以动态地修改等等,这些做法主要是为了进一步提高效率。

轮转法调度是一种剥夺式调度,系统耗费在进程切换上的开销比较大,这个开销与时间片的大小很有关系。如果时间片取值太小,以致于大多数进程都不可能在一个时间片内运行完毕,切换就会频繁,系统开销显著增大,所以,从系统效率来看,时间片取大一点好。另一方面,时间片长度较大,那么,随着就绪队列里进程数目的增加,轮转一次的总时间增大,亦即对每个进程的响应速度放慢了,甚至时间片大到让每个进程足以完成其所有任务,这一算法便退化成先来先服务算法。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数量,要么采用动态时间片法。根据当前负载状况,及时调整时间片的大小。所以,时间片大小的确定要从进程个数、切换开销、系统效率和响应时间等多方面考虑。

3. 优先数调度

给每一个进程确定一个优先数, 处理器调度每次选择就绪进程中优先数最大者, 让它占用处理器运行称优先数调度。怎样确定优先数呢? 可以有以下几种考虑, 使用外围设备频繁者优先数大, 这样有利于提高效率; 重要算题进程的优先数大, 这样有利于用户更早得到计算结果; 进入计算机时间长的进程优先数大, 这样有利于缩短作业的周转时间; 交互式用户的进程优先数大, 这样有利于终端用户的响应时间等等, 以上采用了静态优先数法。效率高性能好的低级调度可采用动态优先数法, 在创建一个进程时, 根据进程类型和资源使用情况确定一个优先数, 而当进程耗尽时间片或重新被调度时, 再次计算并调整所有进程的优先数。基本原则是: ①根据进程占有 CPU 时间多少来决定, 当一个进程占有 CPU 时间愈长, 那么在它被阻塞之后再次获得调度的优先数就越低, 反之, 进程获得调度的可能性越大; ②根据进程等待 CPU 时间多少来决定, 一个进程在队列中等待 CPU 的时间愈长, 那么在它再次获得调度时的优先数就越高, 反之, 进程获得调度的可能性越小。基于优先数的低级调度算法可以按调度方式不同分为剥夺式和非剥夺式优先数调度算法。

早期 UNIX 版本的动态优先数计算公式为:

$$p - pri = \min \{ 127, (p - cpu / 16 + PUSER + p - nice) \}$$

其中, $p - pri$ 为进程优先数, 其值越小, 则该进程的优先权越高。优先数的取值范围为 $-100 \sim +127$; $p - nice$ 是一个用户可以按执行任务的紧急程度通过系统调用命令来设置不同的 $p - nice$ 值; $PUSER$ 为常数 100; $p - cpu$ 反映了进程使用处理机的程度; 进程优先数 $p - pri$ 在数值 127 和 $(p - cpu / 16 + PUSER + p - nice)$ 两者中取其小的值。

最关键的参数是 $p - cpu$, 它按照如下办法来改变: 系统时钟中断处理程序每 20 ms 工作一次, 每工作一次就将当前运行进程的 $p - cpu$ 加 1。每到 1 秒时它就依次检查系统中所有进程的 $p - cpu$, 如果被查进程的 $p - cpu < 10$, 表明该进程在这一秒内所使用处理机的时间不超过 200 ms, 于是把这个 $p - cpu$ 置为 0; 如果被查进程的 $p - cpu > 10$, 表明该进程在这一秒内所使用处理机的时间超过 200 ms, 于是把 $p - cpu$ 减 10。这种改变办法的效果是:

(1) 如果一个进程连续占用处理机较长时间, 那么, 它的 $p - cpu$ 值就增大, 从而, 计算出的 $p - pri$ 也增大, 于是优先数下降。当实行进程切换调度时, 这种进程被调度到的可能性就减少。

(2) 如果一个进程长时间未被调用到或者虽频繁调度到, 但每次占用的时间都小于 200 ms, 那么, $p - cpu$ 就较小甚至为 0, 从而, 计算出的 $p - pri$ 也较小, 于是优先数上升。当实行进程切换调度时, 这种进程被调度到的可能性就增加。

UNIX 系统计算优先数的时机有二: 一是对所有优先数大于 100 的进程, 系统每秒钟重新计算一次它的优先数; 二是每次系统调用命令处理完毕之时, 就重新对现进程的优先数进行计算。

在 UNIX 的后来版本中,按下述公式来计算进程的优先数:

$$p - pri = (p - cpu/2) + \text{用户基本优先数}$$

4. 多级反馈队列调度

多级反馈队列调度这种方法又称反馈循环队列或多队列策略。其主要思想是将就绪进程或线程分为两级或多级,系统相应建立两个或多个就绪队列,较高优先级的队列一般分配给较短的时间片。处理器调度每次先从高一级的就绪进程队列中选取可占有处理器的进程,同一队列中按先来先服务原则排队,只有在选不到时,才从较低一级的就绪进程队列中选取。

进程的分级可以事先规定,例如,使用外围设备频繁者属于高级。在分时系统中可以将终端用户进程定为高级,而非终端用户进程为低级。进程分级也可以事先不规定,一个新进程进入内存后,首先进入高优先级队列等待调度执行,如能在该时间片内完成,便可以撤离系统;凡是运行超越时间片后,就进入低优先级就绪队列,以后给较长的时间片;凡是运行中启动磁盘或磁带而成为等待的进程,在结束等待后就进入中优先级就绪队列,这种调度策略如图 2-41 示。多级反馈队列调度算法具有较好的性能,能满足各类用户的需要。对分时交互型短作业,系统通常可在第一队列(高优先级队列)规定的时间片内让其完成工作,使终端型用户都感到满意;对短的批处理作业,通常,只需在第一或第一、第二队列(中优先级队列)中各执行一个时间片就能完成工作,周转时间仍然很短;对长的批处理作业,它将依次在第一、第二、...,各个队列中获得时间片并运行,决不会出现得不到处理的情况。例如,XDS940 操作系统中,把进程划分为四个优先级类,分别是终端、I/O、短时间片和长时间片。当一个等待终端输入的进程被唤醒时,它排入最高优先级类(终端类)。当一个等待磁盘传输数据的进程就绪时,它将排入第二类优先级队列。当进程在时间片用完时仍为就绪时,它将排入第三类优先级队列。如果一个进程多次用完了时间片而从未因终端或其他 I/O 阻塞,它将被排入第四类优先级队列、即最低优先级类。

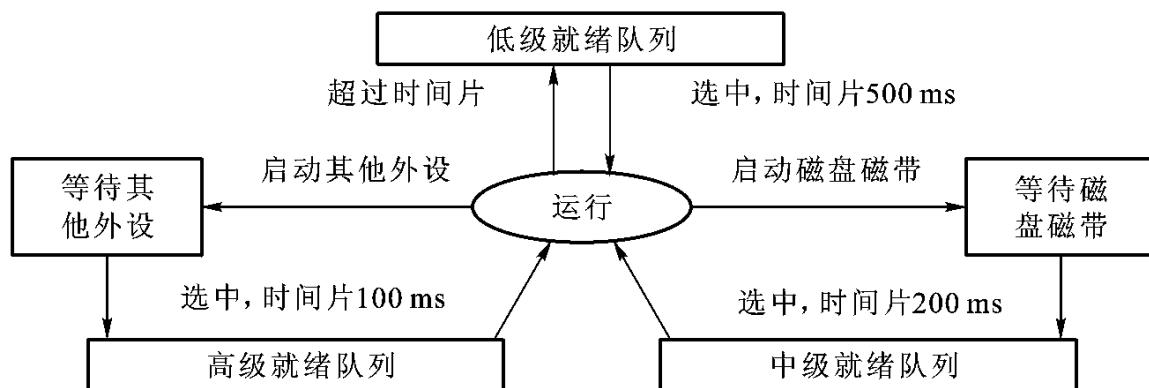


图 2-41 多级反馈队列调度算法

5. 保证调度算法

一种完全不同的调度算法是向用户做出明确的性能保证,然后去实现它,称保证调度算法。一种很实际并很容易实现的保证是:当你工作时已有 n 个用户登录在系统,则你将获得 CPU 处理能力的 $1/n$ 。类似地,如果在一个有 n 个进程运行的用户系统中,每个进程将获得 CPU 处理能力的 $1/n$ 。

为了实现所作的保证,系统必须跟踪各个进程自创建以来已经使用了多少 CPU 时间。然后,计算各个进程应获得的 CPU 时间,即自创建以来的时间除以 n 。由于各个进程实际获得的 CPU 时间已知,所以很容易计算出实际获得的 CPU 时间和应获得的 CPU 时间之比,于是调度将转向比率最低的进程。

6. 彩票调度算法

尽管向用户做出承诺并履行它是一个好主意,但实现却很困难。不过有另一种可以给出类似的可预见结果,而且实现起来简单许多,这种算法称为彩票调度算法。

其基本思想是:为进程发放针对系统各种资源(如 CPU 时间)的彩票。当调度程序需要做出决策时,随机选择一张彩票,持有该彩票的进程将获得系统资源。对于 CPU 调度,系统可能每秒钟抽 50 次彩票,每次中奖者可以获得 20 ms 的运行时间。

在此种情况下,所有的进程都是平等的,它们有相同的运行机会。如果某些进程需要更多的机会,就可以被给予更多的额外彩票,以增加其中奖机会。如果发出 100 张彩票,某一个进程拥有 20 张,它就有 20% 的中奖概率,它也将获得大约 20% 的 CPU 时间。

彩票调度与优先数调度完全不同,后者很难说明优先数为 40 到底意味着什么,而前者则很清楚,进程拥有多少彩票份额,它将获得多少资源。

彩票调度法有几点有趣的特性。彩票调度的反映非常迅速,例如,如果一个新进程创建并得到了一些彩票,则在下次抽奖时,它中奖的机会就立即与其持有的彩票成正比。

如果愿意的话,合作的进程可以交换彩票。一个客户进程向服务器进程发送一条消息并阻塞,它可以把所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;当服务器进程完成响应服务后,它又将彩票交还给客户进程使其能够再次运行;实际上,在没有客户时,服务器进程根本不需要彩票。

彩票调度还可以用来解决其他算法难以解决的问题。例如,一个视频服务器,其中有若干个在不同的视频下将视频信息传送给各自的客户的进程,假设它分别需要 10、20 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率分配 CPU 资源。

2.7.3 实时调度

1. 实时操作系统的特性

实时系统是那些时间因素非常关键的系统。例如,计算机的一个或多个外设发出信号,计算机必须在一段固定时间内做出适当的反应。一个实例是,计算机用 CD - ROM 放 VCD 时,从驱动器中获得的二进制数据必须在很短时间内转化成视频和音频信号,如果转换的时间太长,图像显示和声音都会失真。其他的实时系统还包括监控系统、自动驾驶系统、安全控制系统等等,在这些系统中,迟到的响应即使正确,也和没有响应一样糟糕。

实时系统通常分为硬实时(hard real time)系统和软实时(soft real time)系统。前者意味着存在必须满足的时间限制;后者意味着偶尔超过时间限制是可以容忍的。这两种系统中,实时性的获得是通过将程序分成很多进程,而每个进程的行为都预先可知,这些进程处理周期通常都很短,当检测到一个外部事件时,调度程序按满足它们最后期限的方式调度这些进程。

实时系统要响应的事件可以进一步划分为周期性(每隔一段固定的时间发生)事件和非周期性(在不可预测的时间发生)事件。一个系统可能必须响应多个周期的事件流,根据每个事件需要的处理时间,系统可能根本来不及处理所有事件。例如,有 m 个周期性事件,事件 i 的周期为 P_i ,其中每个事件需要 C_i 秒的 CPU 时间来处理,则只有满足以下条件

$$C_1/P_1 + C_2/P_2 + \cdots + C_m/P_m \leq 1$$

时,才可能处理所有的负载。满足该条件的实时系统称作任务可调度的(schedulable)。

举例来说,一个软实时系统处理三个事件流,其周期分别为 100 ms, 200 ms 和 500 ms,如果事件处理时间分别为 50 ms, 30 ms 和 100 ms,则这个系统是可调度的,因为:

$$0.5 + 0.15 + 0.2 \leq 1$$

如果加入周期为 1 秒的第 4 个事件,则只要其处理时间不超过 150 ms,该系统仍将是可调度的。当然,这个运算的隐含条件是进程切换的时间足够小,可以忽略。

尽管在理论上采取了下面将要讨论的实时调度算法后就可以把一个通用操作系统改造成实时操作系统,但实际上,通用操作系统的进程切换的开销太大,以至于只能满足那些时间限制较松的应用的实时性能要求。这就导致多数实时系统使用专用的实时操作系统。这些系统具有一些很重要的特征,典型的包括:规模小、进程切换很快、中断被屏蔽和处理的时间很短,以及能够管理毫秒或微秒级的多个定时器。

2. 实时调度算法

实时调度算法可以分为动态实时调度算法和静态实时调度算法两类。动态实时调度算法在运行时做出调度决定,静态实时调度算法在系统启动之前完成所有的调度决策。下面来介绍几种经典实时调度算法。

1) 单比率调度算法

单比率调度事先为每个进程分配一个与事件发生频率成正比的优先数, 运行频率越高的进程其优先就数越高。例如, 周期为 20 ms 的进程优先数为 50, 周期为 100 ms 的进程优先数为 10, 运行时调度程序总是调度优先数最高的就绪进程, 并采取抢占式分配策略。可以证明该算法是最优的。

2) 限期调度算法

限期调度的基本思想是: 当一个事件发生时, 对应的进程就被加入就绪进程队列。该就绪队列按照截止期限排序, 对于一个周期性事件, 其截止期限即为事件下一次发生的时间。该调度算法首先运行队首进程, 即截止时间最近的那个进程。

3) 最少裕度法

最少裕度法的基本思想是: 首先计算各个进程的富裕时间, 即裕度 (laxity), 然后选择裕度最少的进程执行。计算公式为: 裕度 = 截止时间 - (就绪时间 + 计算时间), 裕度小说明很紧迫了, 就绪后让它尽快运行。

2.7.4 多处理器调度

一些计算机系统包括多个处理器, 目前应用较多的、较为流行的多处理器系统有:

- 松散耦合多处理器系统: 如 cluster, 它包括一组相对独立的处理器, 每个处理器拥有自己的主存和 I/O 通道。

- 紧密耦合多处理器系统: 它由共享主存和外设的一组处理器组成。

因此, 操作系统的调度程序必须考虑多处理器的调度。显然, 单个处理器的调度和多处理器的调度有一定的区别, 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器调度。

1. 同步的粒度

同步的粒度, 就是系统中多个进程之间同步的频率, 它是刻画多处理系统特征和描述进程并发度的一个重要指标。一般来说, 可以根据进程或线程之间同步的周期 (即每间隔多少条指令发生一次同步事件), 把同步的粒度划分成以下 5 个层次:

- 细粒度 (fine-grained): 同步周期小于 20 条指令。这是一类非常复杂的并行操作的使用, 类似于多指令并行执行。它属于超高并发度的应用, 目前有很多不同的解决方案, 本书将不涉及这些解决方案, 有兴趣的可以参见有关资料。

- 中粒度 (medium-grained): 同步周期为 20 ~ 200 条指令。此类应用适合用多线程技术实现, 即一个进程包括多个线程, 多线程并发或并行执行, 以降低操作系统在切换和通信上的代价。

- 粗粒度(coarse-grained) : 同步周期为 200 ~ 2 000 条指令。此类应用可以用多进程并发程序设计来实现。
- 超粗粒度(very coarse-grained) : 同步周期为 2 000 条指令以上。由于进程之间的交互非常不频繁,因此,这一类应用可以在分布式环境中通过网络实现并发执行。
- 独立(independent) : 进程或线程之间不存在同步,如独立的作业或应用程序。

对于那些具有独立并行性的进程来说,多处理器环境将得到比多道程序系统更快的响应。考虑到信息和文件的共享问题,在多数情况下,共享主存的多处理器系统将比那些需要通过分布式处理实现共享的多处理器系统的效率更高。

对于那些具有粗粒度和超粗粒度并行性的进程来说,并发进程可以得益于多处理器环境。如果进程之间交互不频繁的话,分布式系统就可以提供很好的支持,而对于进程之间交互频繁的情况,多处理器系统的效率更高。

无论是有独立并行性的进程,还是具有粗粒度和超粗粒度并行性的进程,在多处理器环境中的调度原则和多道程序系统并没有太大的区别。但在多处理器环境中,一个应用的多个线程之间交互非常频繁,针对一个线程的调度策略可能影响到整个应用的性能。因此,在多处理器环境中,主要关注的是线程的调度。

2. 多处理器调度的设计要点

多处理器调度的设计要点有三个:为进程分配处理器、在单个处理器上支持多道程序设计和如何指派进程。

多处理器调度的设计要点之一是如何把进程分配到处理器上。假定在多处理器系统中所有的处理器都是相同的,即对主存和 I/O 设备的访问方式相同,那么,所有的处理器可以被放入一个处理器池(pool)。如果采取静态分配策略,把一个进程永久的分配给一个处理器,分配在进程创建时执行,每个处理器对应一个低级调度队列。这种策略调度代价较低,但容易造成在一些处理器忙碌时另一些处理器空闲。也可以采取动态分配策略,所有处理器共用一个就绪进程队列,当某一个处理器空闲时,就选择一个就绪进程占有该处理器运行,这样,一个进程就可以在任意时间在任意处理器上运行。对于紧密耦合的共享内存的多处理器系统来说,由于所有处理器的现场相同,因此,采用此策略时低级调度实现较为方便,效率也较好。

无论采取哪一种分配策略,操作系统都必须提供一些机制来执行分配和调度,那么,操作系统程序在多处理器系统中又是怎样分布呢?方法之一是采用主从式(master/slave)管理结构,操作系统的部分核心运行在一个特殊的处理器上,其他处理器运行用户程序。当用户程序需要请求操作系统服务时,请求将被传递到主处理器上的操作系统。显然这种方式实现上较为简单,并且比多道程序系统的调度效率高,但也有两个缺点:(1)整个系统的坚定性与在主处理器上运行的操作系统程序关系过大;(2)主处理器极易成为系统性能的瓶颈。因

此,还可以采用分布式(peer-to-peer)管理结构,在此种管理结构下,操作系统可以在所有处理器上执行,每一个处理器也可以自我调度。这种方式虽然比较灵活,但实现比较复杂,操作系统本身也需要同步。作为前面两种方法的折衷,可以把操作系统内核组成几部分,允许分别放在不同的处理器上执行。

多处理器调度的设计要点之二是是否要在单个处理器上支持多道程序设计。对于独立、超粗粒度和粗粒度并行性的进程来说,回答是肯定的。但是对于中粒度并行性的进程来说,答案是不明朗的。当很多的处理器可用时,尽可能的使单个处理器繁忙不是那么重要,系统要追求的可能是给应用提供最好的性能,并非是让每个处理器都十分忙碌。

多处理器调度的设计要点之三是如何指派进程。在单处理器的低级调度中讨论了很多复杂的调度算法,但是在多处理器环境中这些复杂的算法可能是不必要的、甚至难以达到预期目的。调度策略的目标是简单有效且实现代价低,线程的调度尤其是这样。

3. 多处理器调度算法

大量的实验数据证明,随着处理器数目的增多,复杂低级调度算法的有效性却逐步下降。因此,在大多数采取动态分配策略的多处理器系统中,低级调度算法往往采用最简单的先来先服务算法或优先数算法,就绪进程组成一个队列或多个按照优先数排列的队列。

多处理器调度的主要研究对象是线程调度算法。线程概念的引进把执行流从进程中分离出来,同一进程的多个线程能够并发执行并且共享进程地址空间。尽管线程也给单处理器系统带来很大益处,但在多处理器环境中线程的作用才真正得到充分发挥。多个线程能够在多个处理器上并行执行,在共享用户地址空间进行通信时,线程切换的代价也远低于进程切换。多处理器环境中的线程调度是一个研究热点,下面讨论几种经典的调度算法。

1) 负载共享调度算法(load sharing)

负载共享调度算法的基本思想是:进程并不分配给一个特定处理器,系统维护一个全局性就绪线程队列,当一个处理器空闲时,就选择一个就绪线程占有处理器运行。这一算法有如下优点:

- 把负载均分到所有的可用处理器上,保证了处理器效率的提高。
- 不需要一个集中的调度程序,一旦一个处理器空闲,操作系统的调度程序就可以运行在该处理器上以选择下一个运行的线程。
- 运行线程的选择可以采用各种可行的策略(雷同于前面介绍的各种低级调度算法)。

Leutenegger, S. 和 Vernon, M. 分析了三种不同线程的负载分配算法:(1)先来先服务。用户进程到达时,它的所有线程被连续地排到就绪队列尾,依先后次序被调度执行;(2)最少线程数优先。共享就绪队列组织成一个优先级队列,如果一个用户进程包含的未被调度的线程数最少,则给它指定最高优先数,被优先调度执行;(3)有剥夺的最少线程数优先。刚到达的用户进程的线程数少于执行的进程的线程数时,前者有权剥夺后者。

这一算法也有不足：

- 就绪线程队列必须被互斥访问,当系统包括很多处理器,并且同时有多个处理器同时挑选运行线程时,它将成为性能的瓶颈。
- 被抢占的线程很难在同一个处理器上恢复运行,因此,当处理器带有高速缓存时,恢复高速缓存的信息会带来性能的下降。
- 如果所有的线程都被放在一个公共的线程池中的话,所有的线程获得处理器的机会是相同的。如果一个程序的线程希望获得较高的优先级,进程切换将导致性能的折衷。

尽管有这样一些缺点,负载共享调度算法依然是多处理器系统最常用的线程调度算法。如著名的 Mach 操作系统,它包括一个全局共享的就绪线程队列,并且每一个处理器还对应于一个局部的就绪线程队列,其中包括了一些临时绑定到该处理器上的就绪线程,处理器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(gang scheduling)

群调度算法的基本思想是:把一组进程在同一时间一次性调度到一组处理器上运行。它具有以下的优点:

- 当紧密相关的进程同时执行时,同步造成的等待将减少,进程切换也相应减少,系统性能自然得到提高。
- 由于一次性同时调度一组处理器,调度的代价也将减少。

群调度引发了对处理器分配的要求,如果有 N 个处理器和 M 个应用程序,每个应用程序有最多 N 个线程。那么,使用时间片,每个应用程序将被给予 N 个处理器中可用时间的 $1/M$,这个分配策略可能效率不高。考虑下面图 2-42 两个应用程序,一个有 4 个线程,另一个有 1 个线程。若使用统一的时间分配,每个应用程序可获得 50% 的 CPU 时间,由于后一个线程运行时,有三个处理器是空闲的,于是浪费的 CPU 资源为 37.5%。可供选另一种统一时间分配称线程数加权调度法,具体来说,给第一个应用程序分 $4/5$ CPU 时间,给第二个应用程序分 $1/5$ 的时间,则处理器时间浪费可降到 15%。

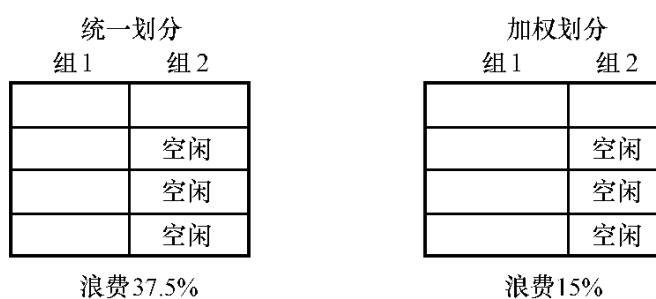


图 2-42 群调度的例子

从上面两个优点来看,群调度算法针对多线程并行执行的单个应用来说具有较好的效率,因此,它被广泛应用在支持细粒度和中粒度并行的多处理器系统中。

3) 处理器专派调度算法(dedicated processor assignment)

处理器专派调度算法的基本思想是:给一个应用专门指派一组处理器,一旦一个应用被调度,它的每一个线程被分配一个处理器并一直占有这个处理器运行直到整个应用运行结束。采用这一算法之后,这些处理器将不适用多道程序设计,即该应用的一个线程阻塞后,该线程对应的处理器不会被调度给其他线程,而将处于空闲状态。

显然,这一调度算法追求的是通过高度并行来达到最快的执行速度,它在应用进程的整个生命周期避免低级调度和切换,且毫不考虑处理器的使用效率。对于高度并行的计算机系统来说,可能包括几十或数百个处理器,它们完全可以不考虑单个处理器的使用效率,而集中关注于提高总的计算效率。处理器专派调度算法适用于此类系统的调度。

最后值得指出的是,无论从理论上还是从实践中都可以证明,任何一个应用任务,并不是划分的越细,使用的处理器越多,它的求解速度就越快。在多处理器并行计算环境中,任何一种算法的加速比的提高都是有上限的。

4) 动态调度算法

在实际应用中,一些应用提供了语言或工具以允许动态地改变进程中的线程数,这就要求操作系统能够调整负载以提高可用性。

动态调度(dynamic scheduling)算法的基本思想是由操作系统和应用进程共同完成调度。操作系统负责在应用进程之间划分处理器。应用进程在分配给它的处理器上执行可运行线程的子集,哪一些线程应该执行,哪一些线程应该挂起完全是应用进程自己的事(当然系统可能提供一组缺省的运行库例程)。相当多的应用将得益于操作系统的这一特征,但一些单线程进程则不适用这一算法。

在动态调度算法中,当一个进程到达系统或要求新的处理器时,操作系统的调度程序主要限制处理器的分配,并且按照下面的步骤处理:

- 如果有空闲的处理器,则满足要求。否则,对于新到达进程,从当前分配了一个以上处理器的进程中收回一个,并把它分配给新到达的进程。
- 如果一部分要求不能被满足,则保留申请直到出现可用的处理器或要求取消。
- 当释放了一个或多个处理器后,扫描申请处理器的进程队列,按照先来先服务的原则把处理器逐一分配给每个申请进程直到没有可用处理器。

2.7.5 实例研究:UNIX SVR4 调度算法

UNIX SVR4 的调度算法同传统 UNIX 相比有了较大变动,其设计目的是优先考虑实时进程,次优先考虑内核模式进程,最后考虑用户模式进程(又称分时进程)。UNIX SVR4 对调度

算法的主要修改包括：提供了基于静态优先数的抢占式调度，包括 3 类优先级层次，160 个优先数，引入了抢占点。由于 UNIX 的基本内核不是抢占式的，它将被划分成一些处理步骤，如果不发生中断的话，这些处理步骤将一直运行直到结束。在这些处理步骤之间，存在着抢占点，称为 safe place，此时内核可以安全地中断处理过程并调度新进程。每个 safe place 被定义成临界区，从而，保证内核数据结构通过信号量上锁并被一致性地修改。

在 UNIX SVR4 中，每一个进程必须被分配一个优先数，并属于一类优先级层次。优先级和优先数的划分如下：

- 实时优先级层次（优先数为 159 ~ 100）：这一优先级层次的进程先于内核优先级层次和分时优先级层次的进程运行，实时进程能利用抢占点抢占内核进程和用户进程。
- 内核优先级层次（优先数为 99 ~ 60）：这一优先级层次的进程先于分时优先级层次进程执行，但迟于实时优先级层次进程运行。
- 分时优先级层次（优先数为 59 ~ 0）：最低的优先级层次，一般用于非实时的用户应用程序。

UNIX SVR4 的低级调度参见图 2-43，它事实上还是一个多级反馈队列，每一个优先数都对应于一个就绪进程队列并由 dispq 指示，每一个进程队列中的进程按照时间片轮转方式调度。位向量 dqactmap 用来标志每一个优先数就绪进程队列是否为空（置 1，对应队列非空）。当一个运行进程由于阻塞、时间片用完或剥夺等原因让出处理器时，调度程序首先查找 dqactmap，发现一个较高优先级的非空队列，指派一个进程占有处理器运行。另外，当执行到一个定义的抢占点时（内核允许产生处理器转让的位置），内核将检查一个叫做 kprunrun 的标志位，如果发现它被置位，则表明至少有一个实时进程处于就绪状态，如果当前进程的优先数低于优先数最高的实时就绪进程，则内核剥夺当前进程。

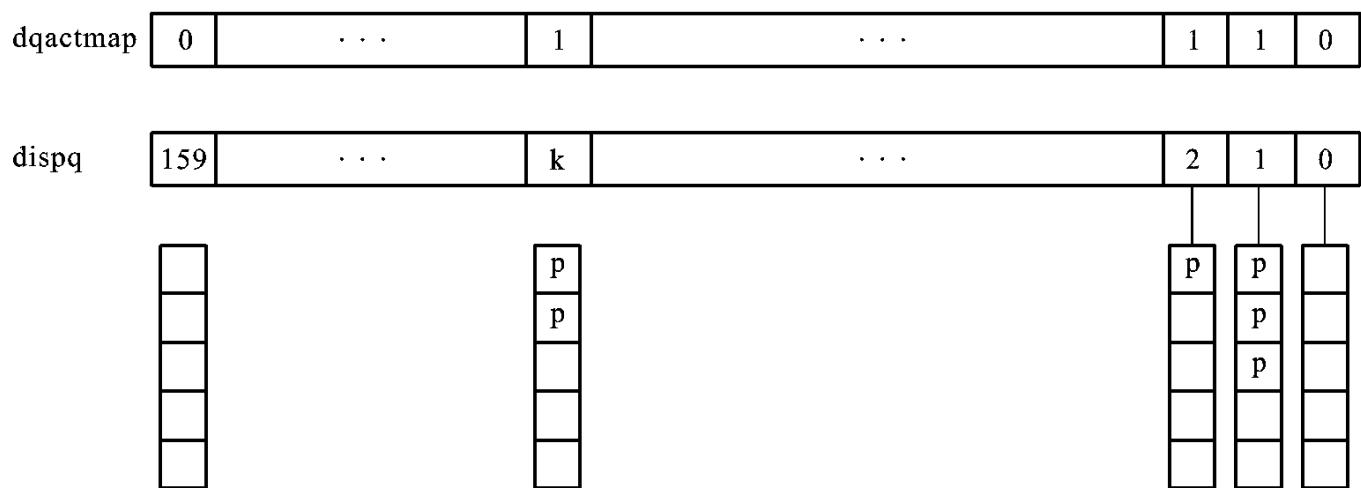


图 2-43 Unix SVR4 的就绪进程队列

对于分时优先级层次,进程的优先数是可变的,当运行进程用完了时间片,调度程序将降低它的优先数,而当运行进程阻塞后,调度程序则将提高它的优先数。分配给分时进程的时间片取决于它的优先数,其范围从优先数 0 分配的 100 ms 到给优先数 59 分配的 50 ms。每个实时进程的优先数和时间片长都是固定的。

2.7.6 实例研究:Windows 2000/XP 调度算法

1. Windows 2000/XP 线程调度及其特征

Windows 2000/XP 处理器调度的调度对象是线程,也称为线程调度。Windows 2000/XP 的设计目标有两个,一是向单个用户提供交互式的计算环境,二是支持各种服务器(server)应用,而且作为一个商用的操作系统性能要求很高。它的线程调度并不是单纯使用某一种调度算法,而是多种算法的结合体,根据系统的实际需要进行针对性的优化和改进。首先,简要描述 Windows 2000/XP 的线程调度及其特征,接着,从内核的角度出发讨论线程的优先级,最后,说明 Windows 2000/XP 线程调度的数据结构和调度算法。

Windows 2000/XP 实现了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。通常线程可在任何可用处理器上运行,但可限制某线程只能在某处理器上运行,亲合处理器集合(由应用进程预先通过系统调用请求的那些运行处理器集)允许用户线程通过 Win32 调度函数选择它偏好的处理器。

当一个线程被调度进入运行状态时,它可运行一个被称为时间配额(quantum)的时间单位。时间配额是允许一个线程连续运行的最大时间总和,随后系统会中断线程的运行,判断是否需要降低该线程的优先级,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。Windows 2000 不同版本的时间配额是不同的,同一系统中各线程的时间配额是可修改的。由于系统的抢占式调度特征,因此,一个线程的一次调度执行可能并没有用完它的时间配额。如果一个高优先级的线程进入就绪状态,当前运行的线程可能在用完它的时间配额前就被抢占。事实上,一个线程甚至可能在被调度进入运行状态之后开始运行之前就被抢走。

Windows 2000/XP 在内核中实现它的线程调度代码,这些代码分布在内核中与调度相关事件出现的位置,并不存在一个单独的线程调度模块。内核中完成线程调度功能的这些函数统称为内核调度器(kernel's dispatcher)。线程调度出现在 DPC/线程调度中断优先级,线程调度的触发事件有以下四种:

- (1) 一个线程进入就绪状态,如一个刚创建的新线程或一个刚刚结束等待状态的线程。
- (2) 一个线程由于时间配额用完而从运行状态转入退出状态或等待状态。
- (3) 一个线程由于调用系统服务而改变优先级或被 Windows 2000/XP 系统本身改变其优先级。

(4) 一个正在运行的线程改变了它的亲合处理器集合。

这些触发事件出现时, 系统必须选择下一个要运行的线程。当 Windows 2000/XP 选择一个新线程进入运行状态时, 将执行一个线程上下文切换以使新线程进入运行状态。线程上下文是指保存正在运行线程的相关运行环境, 加载另一个线程的相关运行环境, 并开始新线程执行的过程。

由于 Windows 2000/XP 的处理器调度对象是线程, 这时的进程仅作为提供资源对象和线程的运行环境, 而不是处理器调度的对象。处理器调度是严格针对线程进行的, 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 2 个可运行的线, 这 12 个线程的优先级都相同, 则每个线程将得到 1/12 的处理器时间。

2. Windows 2000/XP 中与线程调度相关的应用程序编程接口

下表给出了 Win32 API 中与线程调度相关的函数列表。更详细的信息可参见 Win32 API 参考文档。

表 2-3 Win32 API 中与线程调度相关的函数

| 与线程调度相关的 API 函数名 | 函数功能 |
|-----------------------------|----------------------------------------------------|
| Suspend/ResumeThread | 挂起一个正在运行的线程或激活一个暂停运行的线程 |
| Get/SetPriorityClass | 读取或设置一个进程的基本优先级类型 |
| Get/SetThreadPriority | 读取或设置一个线程相对优先级(相对进程优先级类型) |
| Get/SetProcessAffinityMask | 读取或设置一个进程的亲合处理器集合 |
| SetThreadAffinityMask | 设置线程的亲合处理器集合(必须是进程亲合处理器集合的子集), 只允许该线程在指定的处理器集合运行 |
| Get/SetThreadPriorityBoost | 读取或设置暂时提升线程优先级状态; 只能在可调范围内提升 |
| SetThreadIdealProcessor | 设置一个特定线程的首选处理器; 不限制该线程只能在该处理器上运行 |
| Get/SetProcessPriorityBoost | 读取或设置当前进程的缺省优先级提升控制。该功能用于在创建线程时控制线程优先级的暂时提升状态 |
| SwitchToThread | 当前线程放弃一个或多个时间配额的运行 |
| Sleep | 使当前线程等待指定的一段时间(时间单位为毫秒)。0 表示放弃该线程的剩余时间配额 |
| SleepEx | 使当前线程进入等待状态, 直到 I/O 处理完成、有一个与该线程相关的 APC 或经过一段指定的时间 |

3. 线程优先级

为了理解线程调度算法, 首先要说明 Windows 2000/XP 所使用的优先级。Windows

2000/XP 的调度是基于内核级线程的抢占式调度, 包括多个优先数层次。在某些层次线程的优先数是固定的, 在另一些层次线程的优先数将根据执行的情况动态调整。它的调度策略是一个动态优先数多级反馈队列, 每一个优先数都对应于一个就绪队列, 而每一个进程队列中的进程按照时间片方式轮转调度。

如图 2-44, Windows 2000/XP 内部使用 32 个线程优先级, 范围从 0 到 31, 它们被分成以下三个部分:

- 实时优先级(优先数为 31 – 16): 用于通信任务和实时任务。当一个线程被赋予一个实时优先数, 在执行过程中这一优先数是不可变的, 一旦一个就绪线程的实时优先数比运行线程高, 它将抢占处理器运行。
- 可变优先级(优先数为 15 – 1): 用于用户提交的交互式任务。具有这一层次优先数的线程, 可以根据执行过程中的具体情况动态地调整优先数, 但是 15 这个优先数是不能被突破的。
- 系统线程优先级(0): 仅用于对系统中空闲物理页面进行清零的零页线程。

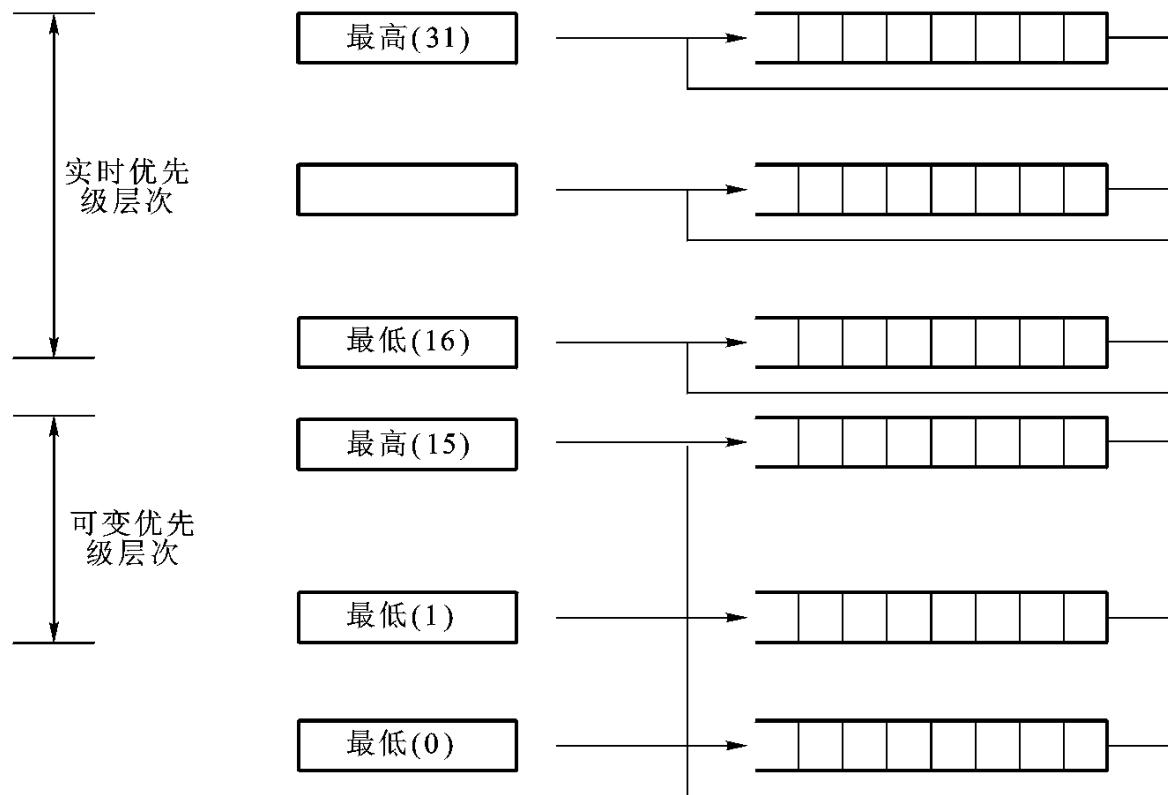


图 2-44 Windows 2000/XP 线程优先级

线程优先级的指定可从两个不同的角度进行: 用户可通过 Win32 应用程序编程接口来指定线程的优先级, Windows 2000/XP 内核也可控制线程的优先级。Win32 应用程序编程接口可在进程创建时指定其优先级类型为实时、高级、中上、中级、中下和空闲, 并进一步在进

程内各线程创建时指定线程的相对优先级为相对实时、相对高级、相对中上、相对中级、相对中下、相对低级和相对空闲。通过 Win32 应用程序编程接口指定的线程优先级是由进程优先级类型和线程相对优先级共同控制的。此外，通过任务管理器（Task Manager）或 Win32 应用程序编程接口的 SetPriorityClass 函数可指定进程的基本优先级，线程从继承的进程基本优先级开始运行。

进程基本优先级和线程开始时的优先级通常缺省地设置为各进程优先级类型的中间值（24、10、8、6 或 4）。Windows 2000/XP 的一些系统进程（如会话管理器、服务控制器和本地安全认证服务器等）的基本优先级比缺省的中级（8）要高一些。这样可保证这些进程中的线程在开始时就具有高于缺省值 8 的优先级。系统进程可使用 Windows 2000/XP 的内部函数来设置比 Win32 基本优先级更高的进程基本优先级。

一个进程仅有单个优先级取值（基本优先级），而一个线程有当前优先级和基本优先级这两个优先级取值。线程的当前优先级可在一定范围（1 至 15）内动态变化，通常会比基本优先级高。系统从不调整在实时范围（16 至 31）内的线程优先级，因而，这些线程的基本优先级和当前优先级总是一样的。

在应用程序中，用户可在一定范围内升高或降低线程优先级。要把线程的优先级提升到实时优先级，用户必须有升高线程优先级的权限。Windows 2000/XP 有许多重要内核系统线程是运行在实时优先级的，如果用户进程在实时优先级运行时间过多，它将可能阻塞关键系统线程（如存储管理器、缓存管理器、本地和网络文件系统、甚至设备驱动程序等）的执行。但由于硬件中断的优先级比任何线程要高，因此，它不会阻塞硬件中断处理。

在被其他线程抢先时，实时优先级线程的行为与可变优先级线程的行为是不同的，前者在被抢先时，它的时间配额将被重置成进入运行状态时的初值。

下面再来讨论一下 Windows 2000/XP 中的中断优先级与线程优先级的关系。回忆图 2-8，可以看出所有线程都运行在中断优先级 0 和 1，用户态线程运行在中断优先级 0，内核态异步过程调用 APC 运行在中断优先级 1，它们会中断线程的运行。虽然高优先级的实时线程可阻塞重要的系统线程执行，但不管用户态线程的优先级是多少，它都不会阻塞硬件中断。线程调度代码是运行在 DPC/dispatch 中断优先级的，因此，当内核正在选择下一个运行线程时，系统中不会有线程正在运行。在多处理器系统中，访问线程调度数据结构是通过请求线程调度器自旋锁来实现同步的。

4. 线程时间配额

时间配额是一个线程从进入运行状态到 Windows 2000/XP 检查是否有其他优先级相同的线程需要开始运行之间的时间总和。一个线程用完了自己的时间配额时，如果没有其他相同优先级的线程，系统将重新给该线程分配一个新的时间配额，并继续运行。每个线程都有一个代表本次运行最大时间长度的时间配额。时间配额不是一个时间长度值，而是一个

称为配额单位(quantum unit)的整数。

1) 时间配额的计算

缺省时, Windows 2000 专业版中线程开始时的时间配额为 6;而在 Windows 2000/XP 服务器版中线程开始时的时间配额为 36。在服务器版中取较长缺省时间配额的原因是要保证客户请求所唤醒的服务器应用有足够的时间在它的时间配额用完前完成客户的请求并回到等待状态。

每次时钟中断,时钟中断服务便从线程的时间配额中减少一个固定值(3)。如果没有剩余的时间配额,系统将触发时间配额用完处理,选择另外一个线程进入运行状态。在 Windows 2000 专业版中,由于每个时钟中断减少的时间配额为 3,因此一个线程的缺省运行时间为 2 个时钟中断间隔;在 Windows 2000/XP 服务器版中,一个线程的缺省运行时间为 12 个时钟中断间隔。不同硬件平台的时钟中断间隔是不同的,时钟中断的频率是由硬件抽象层确定的,而不是由内核确定的。例如,大多数 x86 单处理器系统的时钟中断间隔为 10 ms,大多数 x86 多处理器系统的时钟中断间隔为 15 ms。

2) 时间配额的控制

允许用户指定线程时间配额的相对长度(长或短)和前台进程(指拥有屏幕当前窗口的线程所在进程)的时间配额是否加长。那么,为什么在 Window2000/XP 中要增加前台进程的时间配额,而不是提高前台线程的优先级呢?这是因为当前台、后台应用都在计算时,一味提高前台线程的优先级会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执行,而仅仅给前台线程的时间多些。

为了优化应用的性能,时间配额的设置可为短时间配额和改变前台进程的时间配额,这是 Windows 2000 专业版的缺省设置。为了优化后台服务的性能,可设置为长时间配额和前后台进程的时间配额相同,这是 Windows 2000 服务器版的缺省设置。如果在 Windows 2000 高级服务器或 Windows 2000 数据中心服务器上安装远程终端服务 (terminal services),并且配置该服务器为应用服务器时,时间配额设置为优化应用的性能。

5. 线程调度数据结构

如图 2-45 所示,为了进行线程调度,内核维护了一组“调度器数据结构”。调度器数据结构负责记录各线程的状态,如哪些线程处于等待状态、处理器正在执行哪个线程等。调度器数据结构中的最主要内容是调度器的就绪(KiDispatcherReadyListHead)队列,该队列由一组子队列组成,每个调度优先级有一个队列,其中包括该优先级的等待调度执行的就绪线程。

为了提高调度速度,Windows 2000/XP 维护了一个称为就绪位图(KiReadySummary)的 32 位量。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。还维护一个称为空闲位图(KiIdleSummary)的 32 位量,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

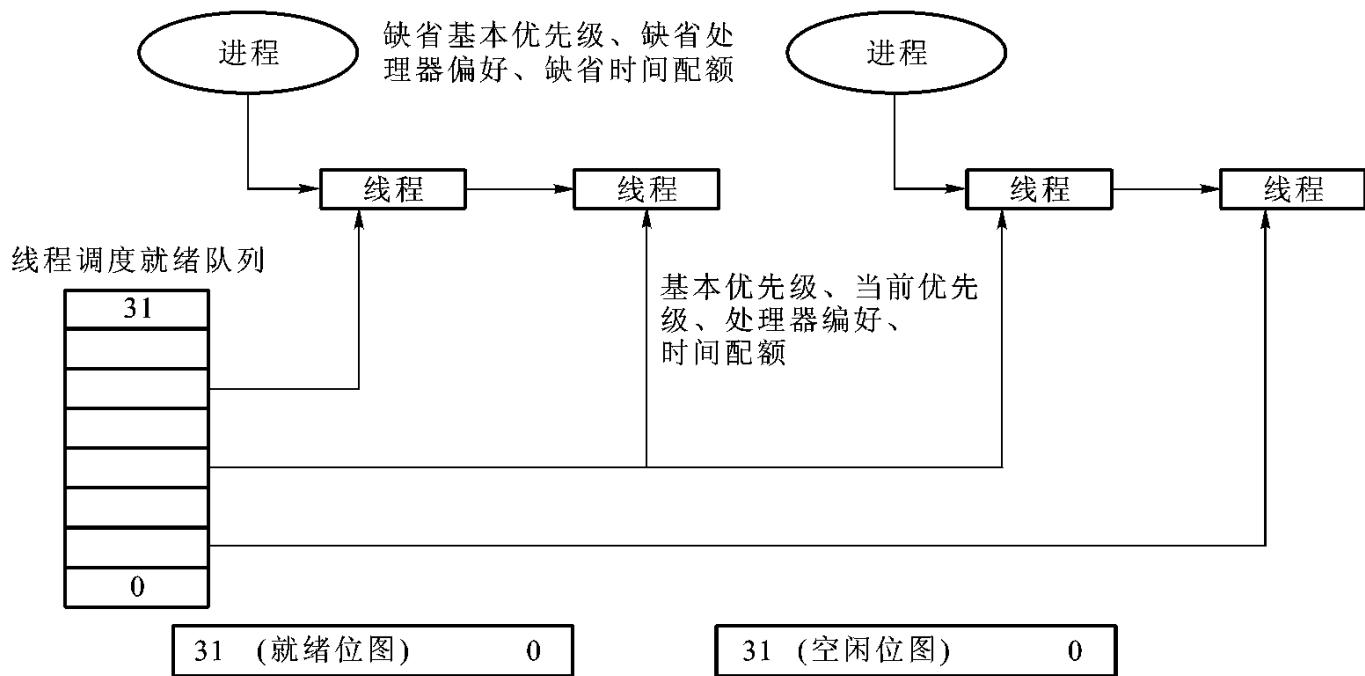


图 2-45 线程调度器数据结构

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/线程调度中断的优先级。但在多处理器系统中,修改调度器数据结构需要额外的步骤来得到内核调度器自旋锁(KiDispatcherLock),以协调各处理器对调度器数据结构的访问。

6. 线程调度策略

Windows 2000/XP 严格基于线程的优先级采用抢占式策略来确定哪一个线程将占用处理器并进入运行状态,Windows 2000/XP 在单处理器系统和多处理器系统中的线程调度是不同的。首先介绍单处理器系统中的线程调度,再介绍多处理器系统中的线程调度。

1) 主动切换

一个线程可能因为进入等待状态而主动放弃处理器的使用,许多 Win32 等待函数调用(如 WaitForSingleObject 或 WaitForMultipleObjects 等)都使线程等待某个对象,等待的对象可能有事件、互斥信号量、资源信号量、I/O 操作、进程、窗口消息等。当线程主动放弃占用的处理器时,调度就绪队列中的第一个线程进入运行状态。

主动放弃处理器的线程会被降低优先级,但这并不是必须的,可以仅仅是被放入等待对象的等待队列中。通常进入等待状态线程的时间配额不会被重置,而是等待事件出现时,线程的时间配额被减 1,但如果线程的优先级大于或等于 14,在等待事件出现时,线程的优先级被重置。

2) 抢占

在这种情况下,当一个高优先级线程进入就绪状态时,正在处于运行状态的低优先级线

程被抢占,可能在以下两种情况下出现抢占:

- 高优先级线程的等待完成,即一个线程等待的事件出现。
- 一个线程的优先级被增加或减少。

在这两种情况下,系统都要确定是否让当前线程继续运行或是否当前线程要被一个高优先级线程抢占。注意,用户态下运行的线程可以抢占内核态下运行的线程,在判断一个线程是否被抢占时,并不考虑线程处于用户态还是内核态,调度器只是依据线程优先级进行判断。

当线程被抢占时,它被放回相应优先级的就绪队列的队首,而不是队尾。处于实时优先级的线程在被抢占时,时间配额被重置为一个完整的时间片;而处于动态优先级的线程在被抢占时,时间配额不变。当抢占线程完成运行后,被抢占的线程可继续运行直到剩余的时间配额用完。

3) 时间配额耗尽

当一个处于运行状态的线程用完它的时间配额时,Windows 2000/XP 首先必须确定是否需要降低该线程的优先级,然后确定是否需要调度另一个线程进入运行状态。如果刚用完时间配额的线程的优先级被降低了,系统将寻找一个更适合的线程进入运行状态、即是指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没有降低,并且有其他优先级相同的就绪线程,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状态,刚用完时间配额的线程被排列到就绪队列的队尾(即分配一个新的时间配额并把线程状态从运行状态改为就绪状态)。如果没有优先级相同的就绪线程可运行,刚用完时间配额的线程将得到一个新的时间配额并继续运行。

4) 结束

当线程完成运行时,它的状态从运行状态转到终止状态。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。如果处于终止状态的线程对象上没有未关闭的句柄,则该线程将被从进程的线程列表中删除,相关数据结构将被释放。

7. 线程优先级提升

在下列 5 种情况下,Windows 2000/XP 会提升线程的当前优先级:

- (1) I/O 操作完成。
- (2) 信号量或事件等待结束。
- (3) 前台进程中的线程完成一个等待操作。
- (4) 由于窗口活动而唤醒图形用户接口线程。
- (5) 线程处于就绪状态超过一定时间,但没能进入运行状态(处理器饥饿)。

其中,前两条是针对所有线程进行的优先级提升,而后三条是针对某些特殊的线程在正

常的优先级提升基础上进行额外的优先级提升。线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征,解决线程调度策略中潜在的不公正性。与任何调度算法一样,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(16至31)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以预测的。

I/O 完成后的线程优先级提升。在完成 I/O 操作后,系统将临时提升等待该操作线程的优先级,以保证等待 I/O 操作的线程能有更多机会立即开始处理得到的结果。线程优先级的实际提升值是由设备驱动程序决定的,设备驱动程序在完成 I/O 请求时,通过内核函数 IoCompleteRequest 来指定优先级的提升幅度,通常线程优先级的提升幅度与 I/O 请求的响应时间要求是一致的,响应时间要求越高,优先级提升幅度越大,如磁盘、光驱、并口和视频为 1;网络、串口和命名管道为 2;键盘和鼠标为 6;音频为 8。

等待事件和信号量后的线程优先级提升。当一个等待执行的事件的对象或信号量对象的线程完成等待后,它的线程将提升一个优先级。线程在等待事件和信号量后提升优先级的原因与等待 I/O 操作的优先级提升的原因类似,阻塞于等待事件和信号量的线程得到的 CPU 时间比处理器繁忙型线程要少,这种提升可减少不平衡带来的影响。

前台线程在等待结束后的优先级提升。对于前台进程中的线程,一个内核对象上的等待操作完成时,内核函数 KiUnwaitThread 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一个进程是前台进程,前台线程优先级的提升是为了改进交互型应用的响应特征。

图形用户接口线程被唤醒后的优先级提升。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 的额外优先级提升,在调用函数 KeSetEvent 时实施这种优先级提升。同样这种优先级的提升是为了改进交互型应用的响应时间。

对处理器饥饿线程的优先级提升。Windows 2000/XP 的平衡集管理器 (balance set manager)是内存管理的一个系统线程,它会每秒钟检查一次就绪队列,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(300 个时钟中断间隔约 3 至 4 秒钟)。如果找到这样的线程,将把它的优先级提升到 15,并分配给它一个长度为正常值 2 倍的时间配额;当被提升线程用完它的时间配额后,该线程的优先级立即衰减到它原来的基本优先级。如果在该线程结束前出现其他高优先级的就绪线程,该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。每次平衡集管理器并不扫描所有就绪线程,为减少系统开销,只扫描 16 个就绪线程。如果就绪队列中有更多的线程,它将记住暂时位置,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了 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 饥饿的线程通常都能得到足够处理器时间来完成它的处理操作并重新进入等待状态。

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，在多处理器系统中会出现什么情况？当 Windows 2000/XP 试图调度优先级最高的可执行线程时，有几个因素会影响到处理器的选择。系统只保证一个优先级最高的线程处于运行状态。在描述算法前，首先定义几个术语。

1) 亲合关系

每个线程都有一个亲合掩码，描述该线程可在哪些处理器上运行。线程的亲合掩码是从进程的亲合掩码继承得到的。缺省时，所有进程（即所有线程）的亲合掩码为系统上所有要用处理器的集合。也就是说，所有线程可在所有处理器上运行。应用程序通过调用 SetProcessAffinityMask 或 SetThreadAffinityMask 函数来修改缺省的亲合掩码。

2) 线程的首选处理器和第二处理器

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：

- 首选处理器：线程运行时的偏好处理器。
- 第二处理器：线程第二个选择的运行处理器。

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，这样进程中每个新线程得到的首选处理器会在系统中的可用处理器中循环。线程创建后，系统不会修改线程的首选处理器设置；但应用程序可通过 SetThreadIdealProcessor 函数来修改线程的首选处理器。

3) 就绪线程的运行处理器选择

当线程进入运行状态时，Windows 2000/XP 首先试图调度该线程到一个空闲处理器上运行。如果有多个空闲处理器，线程调度器的调度顺序为：首先是线程的首选处理器，其次是线程的第二处理器，第三是当前执行处理器（即正在执行调度器代码的处理器）。如果这些处理器都不是空闲的，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状态，选择找到的第一个空闲处理器。

如果线程进入就绪状态时所有处理器都处于繁忙状态，系统将检查它是否可抢先一个处于运行状态或备用状态的线程。检查的顺序如下：首先是线程的首选处理器，其次是线程的第二处理器。如果这两个处理器都不在线程的亲合掩码中，Windows 2000/XP 将依据活动处理器掩码选择该线程可运行的编号最大的处理器。注意，线程的亲合掩码与首选处理器、第二处理器的设置是相互独立的，首选和第二处理器由系统在创建线程时指定，而亲合掩码由用户选择。

如果被选中的处理器已有一个线程处于备用状态（即下一个在该处理器上运行的线程），并且该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状态的线程，成为该处理器的下一个运行线程。如果已有一个线程正在被选中的处理器上运行，Windows 2000/XP 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

线程优先级高，则标记当前运行线程为被抢先，系统会发出一个处理器中断，以抢先正在运行的线程，让新线程在该处理器上运行。

4) 为特定的处理器调度线程

在有些情况下（如线程降低它的优先级、修改它的亲合处理器推迟或放弃执行等），Windows 2000/XP 必须选择一个新线程在刚让出的处理器上运行。在单处理器系统中，简单地从就绪队列中选择最高优先级的第一个线程。在多处理器系统，不能简单地从就绪队列中取第一个线程，它要寻找一个满足下列四个条件之一的线程：

- 线程的上一次运行是在该处理器上。
- 线程的首选处理器是该处理器。
- 处于就绪状态的时间超过 2 个时间配额单位。
- 优先级大于等于 24。

显然，由线程亲合掩码限制不能在指定处理器上运行的线程将在检查中跳过。如果系统不能找到满足要求的线程，它将从就绪队列的队首取第一个线程进入运行状态。为什么在为处理器选择运行线程时要考虑线程上一次运行时使用的处理器？主要的原因是速度问题。如果线程的连续两次运行都在同一处理器上，将增加线程数据仍保留在处理器第二级缓存的可能性。

5) 最高优先级就绪线程可能不处于运行状态

在多处理器系统中，Windows 2000/XP 并不总是选择优先级最高的线程在一个指定的处理器上运行，从而有可能出现这种情况，一个比当前正在运行的线程优先级更高的线程处于就绪状态，但不能立即抢占当前线程，进入运行状态。

一种高优先级线程可能不抢占当前线程的情况是，线程的亲合掩码限制线程只能在一部分可用处理器上运行。在这种情况下，某线程可运行的处理器上运行着高优先级线程，而其他处理器可能空闲或运行着该线程可抢占的低优先级线程。虽然把一个处于运行状态的线程从一个处理器移到另一个处理器的做法可允许另一个由于亲合掩码限制而不能运行的线程进入运行状态，但 Windows 2000/XP 不会因此把一个正在运行的线程从一个处理器移到另一个处理器上。

例如，若 0 号处理器上正运行着一个可在任何处理器上运行的优先级为 8 的线程，1 号处理器上正运行着一个可在任何处理器上运行的优先级为 4 的线程；这时一个只能在 0 号处理器上运行的优先级为 6 的线程进入就绪状态。在这种情况下，优先级为 6 的线程只能等待 0 号处理器上优先级为 8 的线程结束。因为 Windows 2000/XP 不会为了让优先级为 6 的线程在 0 号处理器上运行，而把优先级为 8 的线程从 0 号处理器移到 1 号处理器。即 0 号处理器上的优先级为 8 的线程不会抢占 1 号处理器上优先级为 4 的线程。

9. 空闲线程

如果在一个处理器上没有可运行的线程, 系统会调度相应处理器上对应的空闲线程。因为在多处理器系统中可能两个处理器同时运行空闲线程, 所以, 系统中的每个处理器都有一个对应的空闲线程。Windows 2000/XP 给空闲线程指定的线程优先级为 0, 但实际上该空闲线程只在没有其他线程要运行时才运行。空闲线程的功能就在一个循环中检测是否有要进行的工作。虽然不同处理器结构下空闲线程的流程有一些区别, 但基本的控制流程都如下所述:

- 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, 则清除相应软中断并执行 DPC。
- 检查是否有就绪线程可进入运行状态。如果有, 调度相应线程进入运行状态。
- 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。

2.7.7 实例研究: Linux 调度算法

Linux 进程调度子程序 `schedule()` 负责协调进程的运行, 管理进程对 CPU 的使用, 让每一个就绪进程都能公平地共享 CPU 时间, 并尽可能最大限度地利用 CPU。调度程序必须确保一个进程所获得的 CPU 访问时间和其指定的优先级及进程类型匹配, 确保消除任何进程对 CPU 资源处于饥饿状态。

在每一个进程的 `task_struct` 结构中, 都存放有与进程调度有关的重要参数, 供 `schedule()` 调度程序使用, 主要参数有:

(1) `policy`: 进程调度策略, 进程可以通过宏定义来区分三类进程, 即 `policy` 有三种值:

- `# define SCHED_OTHER 0`: 为普通类任务。非实时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。

- `# define SCHED_FIFO 1`: 先进先出实时类任务。符合 POSIX.1b 的 FIFO 规定。它会一直运行(相当于时间片硕大或没有时间片的概念), 除非它自己出现等待事件或有另一个具有更高 `rt_priority` 的实时进程出现时, 才出让 CPU。

- `# define SCHED_RR 2`: 轮转法实时类任务。符合 POSIX.1b 的 RR 规定。除了时间片是一个定量外, 和 `SCHED_FIFO` 类似。当时间片耗尽后, 就使用相同的 `rt_priority` 排到原队列的队尾。

(2) `priority`: 静态优先级, 非实时类进程使用。其值是从 -20 至 19 的整数(内核转化为 1 至 40 的优先级), 不会随时间而改变, 但能由用户进行修改。它指明在被迫和其他进程竞争 CPU 之前, 该进程被允许运行的时间片最大值(滴答次数)。当然, 可能由于一些原因, 在该时间片耗尽前进程就被迫出让 CPU。

(3) counter: 动态优先级, 主要被非实时类进程使用, 它指出了进程时间片的剩余时间量。只要进程占有了 CPU, 它就随着时间不断减小, 当 counter 小于等于 0 时, 标记进程时间片耗尽应重新调度(这时设置 need-resched 标志)。Update-process-times() 函数在每个时钟滴答将 counter 值减 1。注意, counter 不是在所有场合都有效的, 对不同类型进程, 它的作用不一样。只有对非实时类进程, 因为采用时间片轮转法调度, counter 才起作用, 并不断地计算动态优先级; 对于采用时间片轮转法调度策略的实时类进程, counter 仅起到时间片作用, 并不计算动态优先级; 对于采用先进先出调度策略的实时进程, counter 被忽略, 因为这时的时间片看作无穷大。

(4) rt_priority: 实时优先级, 仅被实时进程使用。这是从 0 至 99 的一个整数, 用来区分实时进程的等级, 较高权值的进程总优先于较低权值的进程。rt_priority + 1000 给出实时进程的优先级, 因此, 实时进程的优先级总高于普通进程(普通进程优先级必小于 999, 实际上仅使用 0 到 56)。

从上面的讨论可以看出, 实时进程有很高的优先级, 这样同时存在普通和实时进程时, 如果一个实时进程准备运行, 调度子程序调用 goodness() 函数保证实时进程优先于普通进程被调度。实时进程有两种可选的调度算法, 先进先出法与轮转法。用轮转法调度时, 每一个能运行的实时进程将依次运行; 用先进先出调度, 每一个能运行的实时进程将依排在运行队列中的次序运行, 该次序不能被改变。因此, Linux 进程共有三种进程调度策略, 用户进程可以使用系统调用 sched_setscheduler() 和 sched_setparam() 改变自己的调度策略和实时优先级, 通过系统调用 nice() 和 setpriority() 改变进程的静态优先级, 一旦进程变为实时进程, 其新产生的子进程也是实时进程。

上面已经提到了动态优先级的概念, 由于 Linux 进程的优先级会随时间动态变化, 只要进程占用 CPU, 它就随着时间的流失而不断减小, 当它小于 0 时, 标志着进程要被重新调度, 动态优先级指明了在这个时间片中所剩余的时间量。Linux 的动态优先级与 task_struct 中的 counter 有关, 它被规定为: 在抢占中断调度之前, 该进程在这个时间片中还能连续运行的时间(除非进程自己放弃 CPU), 而且其值较大的会有较高的调度优先级。所以, 可以把 counter 看作动态优先级。由于时钟中断发生的周期为 10 ms(一个滴答), 因而, 一个进程的动态优先级为 60 的话, 那么, 该进程这次还能连续运行 600 ms。

接着来讨论动态优先级 counter 是如何产生和变化的, 这与静态优先级 priority 有关。一个进程的静态优先级 priority 和调度策略 policy 是在其创建时从父进程处继承来的, 且在进程整个生命周期保持不变, 除非使用相关系统调用设置它。在进程被创建时, 它的动态优先级 conuter 的初值为 priority 的值, 此后, 每当发生时钟中断时, 正在运行的进程的 counter 值减 1, 直到发生以下情况后停止:

- 当 counter 递减到 0 时, 运行进程被迫从运行态进入就绪态。但系统中处于就绪态的

进程,其动态优先级不一定全为 0,一个从等待态进入就绪态的进程,其动态优先级通常不为 0。从运行态进入就绪态的那些进程,会在就绪态中一直保持动态优先级为 0,直到系统中所有就绪态进程的动态优先级全为 0 时(因为不为 0 的那些进程会很快获得处理器执行,直到动态优先级减为 0 或进入等待态),调度程序重置各进程的 counter 为静态优先级的值,然后,按动态优先级的值大小依次进入运行态运行。

- 处于等待态的进程的动态优先级通常会逐渐增加,也可能不变,但不会减小。因为当所有就绪进程的动态优先级都为 0 时,系统将重新计算进程的动态优先级,这既包括了就绪态进程,也包括了等待态进程。计算方法为:将每个进程的动态优先级的值右移一位,再加上该进程的静态优先级值,就得到了该进程的动态优先级的新值。对于就绪态进程来说,由于其动态优先级都为 0,所以,计算结果仍为赋值(priority 的值)。但对于等待态进程就不一样了,此时它们的动态优先级都不为 0,所以,计算结果等待态进程的动态优先级(priority + counter/2)会大于静态优先级的值(priority),但决不会比静态优先级值的两倍更大。

按照上述算法,等待态进程会有较高的优先级,处于等待态越久的进程,其进程的动态优先级会越高,这就意味着,I/O 较多的进程由于其等待时间久,会有较高的优先级;计算较多的进程因其占用 CPU 多使优先级会较低,这不仅因为它不会像等待态进程那样增加优先级,还因为它会在就绪态中停留较长时间直到系统中所有就绪态进程的动态优先级为 0。

现在来讨论 Linux 的调度工作,Linux 进程调度的时机有多种:

- 当前运行进程进入等待队列,CPU 空闲时。
- 当进程执行一个系统调用的结尾时。
- 当系统定时器将运行进程的时间片 counter 减至 0 时。

每当 Schedule() 工作时,主要完成以下任务:

(1) 处理当前进程

Schedule() 首先判断是否有被标记过的底半处理程序,如果有的话,调用 do-bottom-half() 去执行有效的 bottom half 底半中断处理过程,然后,再调用 run-task-queue() 函数去处理核心的任务队列 tq-scheduler。当新的进程投入运行之前,必须对正在执行的进程作相应处理,以确保这个进程在以后能正确再次运行。

- 如果当前进程的调度策略为轮转法且 counter = 0,则保持执行状态(TASK_RUNNING),并把它追加到运行队列的尾部,并分给它一个新的时间片。
- 如果某进程是可中断(TASK_INTERRUPTIBLE)的,且收到了中断信号,那么,把其状态修改为 TASK_RUNNING。
 - 如果当前进程超时(这时 counter = 0),则其状态修改为 TASK_RUNNING。
 - 如果当前进程的状态为 TASK_RUNNING,则保持执行状态。
 - 把既非执行状态又非可中断(TASK_UNINTERRUPTIBLE)的进程从运行队列中移出,

这些进程暂无资格被调度, 把进程 `read - resched` 清为 0。

(2) 选择进程运行

当 CPU 空闲时, 调度子程序扫描运行队列所有活动进程, 从运行队列中选择一个合适进程运行, 这时要执行 `godness()` 函数, 同时用变量 `c` 记录运行队列中所有进程的最好 `godness` 值, 用 `host` 指向对应进程的 `task - struc`:

- 如果循环结束后, 运行队列所有进程的 `godness` 为 0, 那么, `c` 就是 0。则重新计算进程的时间片, 相当于把 `priority` 赋给 `counter`, 再次转向执行 `godness()` 函数。
- 如果执行队列中有实时进程, 则选择一个优先级最高(`goodness` 最大)的进程。普通进程的权值为 `counter` 的值;而实时进程的权值为 `counter + 1 000`, 这就保证了实时进程总比普通进程优先执行。
- 如果运行队列中的进程有相同优先级, 则耗时多(`counter` 值较大)的进程比耗时少的进程优先调度。
- 如果优先级相同且 `counter` 值也相等, 则位于队列前面的进程被优先调度。

经过上面步骤的选择, 如果 `next` 就是当前进程, 则结束时 `Schednle()` 并当前进程返回运行。否则, 由 `switch - to()` 进行进程切换, 处理器改由 `next` 进程占用。

当 CPU 空闲且没有就绪状态的进程时, 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, 也不进入阻塞状态, 即总处于执行状态, 与其他进程不同的是它的 `task - struct` 数据结构是由系统静态分配的。

(3) 切换进程

如果当前进程被剥夺, 调度程序从运行队列中选中一个新的进程来运行, 就按照 `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了虚拟内存, 那么相关页表必须同步更新。

(4) SMP 进程调度

Linux2.0.0 版本开始支持对称多处理器 SMP, 多处理器系统中每个 CPU 都可以运行一个进程, 因此, 实施进程调度时, 可供选择的 CPU 不止一个。在 SMP 中, 每一个进程的 `task - struct` 结构中都含有当前运行所用的 CPU 编号及最后一次使用的 CPU(`last _ processor`)编号, 虽然系统不限制进程每次必须在不同的 CPU 上运行, 但可以利用 `processor _ mask` 掩码字限制进程只能在某些 CPU 上运行。如果掩码字的第 `N` 位设置为 1, 则该进程可以在第 `n` 个 CPU 上运行, 也就是说, 进程可以与某个 CPU 绑定(也称亲缘关系)。进行进程调度时, 调度程序总是优先考虑进程在它最后一次使用的 CPU 上运行, 因为, 该 CPU 的高速 cache 中最有可能包含此进程的一些特定数据, 改变 CPU 会增加起它对存储器的访问, 在一定程度上影响系统的效率。然而, 使用亲缘关系必须特别仔细, 在某个进程限制于特定 CPU 而没有其他进程能够在该 CPU 上执行时, CPU 亲缘关系很有意义, 但这是针对系统有特别多的 CPU

而言(如几十个),在一个双CPU计算机系统中使用亲缘关系肯定太奢侈了。

2.8 本 章 小 结

本章在处理器硬件的基础上着重讨论中断技术、进程及其实现、线程及其实现和各种处理器调度算法。操作系统是由中断激活的,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工作,在处理完该中断事件后,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,转向低级调度执行调度工作。由于中断能改变处理器内操作执行的顺序,它成为操作系统实现并发性的硬件基础之一,用户程序在执行过程中不仅可用系统调用方式,还可以中断方式获得操作系统的服务,中断系统在实现进程的并发执行、满足用户对系统的功能需求、处理硬件和软件故障、满足实时处理要求等方面均起着重要作用。但由此也增加了操作系统设计的复杂性。这里介绍了中断、中断装置、中断源及分类、中断处理、中断优先级、中断屏蔽、多重中断等概念。作为实例研究,讨论了Window2000/XP、Solaris和Linux中断处理。

为了屏蔽中断的影响,操作系统较早地引入了一个由并发执行的顺序程序组成的概念模型—进程,通过中断方式进程可以在实处理器上交替执行,每个进程可以认为都运行在自己的一个虚拟处理器上。进程是操作系统中最重要和最基本的概念之一,引入进程是因为系统资源的有限性和系统内操作的并行性所决定的,进程较好地刻画了操作系统的并发性、共享性和随机性,在操作系统的理论研究和设计实现上均发挥了重要作用,采用进程结构的操作系统具有结构清晰、整齐划一,可维护性好的优点。进程具有生命周期,由创建而产生,到撤销而消亡。操作系统的基本功能是进程的创建、管理、撤销。为了实现对进程的管理,操作系统维护着每个进程的资料结构—进程控制块PCB,这是进程的惟一标志,创建一个进程必须为它创建一个PCB,反之撤销一个进程系统便回收它的PCB。进程在推进过程中,其状态会不断变化,最终完成其承担的系统或应用任务。围绕这些内容,本章对进程的定义、进程的属性、进程的状态及转换、进程的描述和组成、进程的控制都作了详细讨论。由于进程是并发程序的执行,是一个动态的概念,但为了能在处理机上执行仍然需要静态描述,一个进程的静态描述是处理器的一个执行环境,称之为进程上下文(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,或产生了一个中断事件,处理器都进行一次模式切换,操作系统接收控制权,有关系统例程完成必须的操作后,或恢复被中断进程或切换到新进程。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。因此,进程的运行被认为是在进程的上下文中执行,这时的控制权在操作系统手中,它在完成必要的操作后,可以恢复被中断的进程或切换到别的进程。本章中

讨论的进程实例是 UNIX SVR4 和 Linux 进程。

随着并行技术、网络技术和软件设计技术的发展,那些需要频繁输入输出并同时大量计算的服务器进程(如数据库服务器、事务监督程序)很难体现效率。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,以改善资源使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开销,使得并发粒度更细、并发性更好。引入多线程的进程称为多线程结构进程(反之称单线程结构进程),其中进程涉及到资源所有权及其保护,线程是 CPU 调度的独立单位并涉及到程序的执行,这样做能发挥并发性、提高性能和方便编程。本章中讨论了多线程环境中,进程与线程的定义、特性、组成,以及不同的线程实现方法(ULT、KLT、混合式)。用户级线程与操作系统无关,而由一个在进程的用户空间中运行的线程库创建和管理,由于线程切换不需要模式切换,所以十分有效。但一个进程中一次仅一个用户线程可以执行,如果一个线程发生阻塞,整个线程会被阻塞。内核级线程是指进程中的线程由内核维护,由于内核管理它们,因而一个进程中的多个线程可在多处理器上同时执行,且一个线程阻塞不阻塞整个进程,但是一个线程到另一个线程的切换需要模式转换。在一个进程中包含有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,同一进程的多个线程可在多个处理器上并发或并行地执行,而进程之间的并发执行演变为不同进程的线程之间的并发执行。本章中实例研究讨论了 Solaris 、Linux 和 Windows 2000/XP 操作系统的进程和线程。

无论是在操作系统控制下执行的应用程序,还是操作系统程序自己,都最终要在处理器上执行,以便实现其功能。在计算机系统中,可能同时有许多批处理作业存放在后备作业队列中,或者有许多终端与主机相连接。如何从这些作业中挑选作业进入主存运行、如何在进程之间分配处理器时间,无疑是操作系统资源管理中的一个重要问题,这些均涉及到处理器调度问题,调度算法的优劣直接影响多道程序系统的效率。本章中在三级调度中,主要讨论高级调度和低级调度。高级调度的工作是决定满足资源需求的哪些后备作业选中进入主存去多道运行,本章中讨论了 FCFS、SJF、SRTF、HRRF、优先数和分类调度等作业调度算法。低级调度的工作是决定下面该哪一个进程或线程占有处理器运行,讨论了 FCFS、时间片轮转、优先数、多级反馈等调度算法。对于实时系统调度算法(单比率调度、限期调度、最少裕度调度)和多处理器调度算法(负载共享调度、群调度、处理器专派调度、动态调度)也作了扼要讨论。衡量算法优劣的因素有: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个 CPU 状态。
4. 试讨论多种 CPU 状态比两种 CPU 状态的优点。
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. 从中断事件性质来说, 可把它分成哪些类型?
8. 从中断事件来源来说, 可把它分成哪些类型?
9. 在处理程序性中断时, 什么情况下可转用户中断续元处理?
10. 叙述中断处理程序应完成的任务。
11. 何谓中断的优先级? 为什么要对中断事件分级?
12. 简述程序性中断的处理过程。
13. 为什么中断续元处理可以嵌套, 但不能递归?
14. 叙述系统调用的一般执行过程。
15. 试述中断在操作系统中的重要性及其主要作用。
16. 试述时钟中断在操作系统中的重要性及其主要作用。
17. 叙述中断屏蔽的作用, 哪些中断事件可加以屏蔽?
18. 操作系统如何处理多重中断事件。
19. 试解释中断号和中断向量。
20. 试讨论硬中断与软中断。
21. 解释 Windows 2000/XP 的中断、异常和陷阱。
22. 解释 Windows 2000/XP 中, 如何动态地实现中断屏蔽功能?
23. 简述 Linux 的时钟机制。
24. 简述 Linux 中断源的分类及处理原则。
25. 简述 Linux 的中断底半处理: 原理、数据结构和处理过程。
26. 什么是进程? 计算机操作系统中为什么引入进程?
27. 进程有哪些主要属性? 试解释之。
28. 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?
29. 五态模型的进程中, 新建态和终止态的主要作用是什么?
30. 试说明引起创建一个进程的主要事件?

31. 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。
32. 什么是进程的挂起状态? 列出挂起进程的主要特征。
33. 什么情况下会产生挂起等待态和挂起就绪态,试举例说明。
34. 叙述组成进程的基本要素,并说明它们的作用。
35. 何谓进程控制块(PCB)? 它包含哪些基本信息?
36. 何谓进程队列、入队和出队?
37. 叙述创建进程系统所要做的主要工作。
38. 什么是进程的上下文? 简述其主要内容。
39. 什么叫进程切换? 叙述进程切换的主要步骤。
40. 什么叫模式切换? 它与进程切换有何主要差别。
41. 进程切换主要应该保存哪些处理器状态?
42. 试说明引起撤销一个进程的主要事件。
43. 叙述系统撤销进程时应做的主要工作。
44. 简单解释操作系统功能的三种实现模型。
45. 小结 UNIX SVR4 进程管理的特点。
46. UNIX 中,为什么把 PCB 的 proc 和 user 结构分离?
47. 列举进程被阻塞和唤醒的主要事件。
48. 操作系统中引入进程概念后,为什么又引入线程概念?
49. 列举支持线程概念的商业性操作系统。
50. 试叙述多线程环境中,进程和线程的定义。
51. 什么是线程控制块(TCB)? 它有哪些主要内容?
52. 试从调度、并发性、拥有资源和系统开销四个方面对传统进程和线程进行比较。
53. 试叙述 ULT 和 KLT 的区别。
54. 试对下列系统任务作出比较:
 - a) 创建一个进程与创建一个线程
 - b) 两个进程间通信与同一进程中两个线程间通信
 - c) 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
55. 列举与线程状态变化有关的主要线程操作。
56. 挂起状态与线程有何关系? 为什么?
57. 叙述并发多线程程序设计的主要优点及其应用。
58. 什么是内核级线程、用户级线程和混合式线程? 对它们进行比较。
59. 叙述 Solaris 中的进程与线程概念。
60. 叙述 Solaris 中的线程种类? 轻量级进程的作用是什么?
61. 叙述 Solaris 进程的数据结构。
62. 叙述 Solaris 用户级线程的数据结构。

63. 叙述 Solaris 轻量级进程的数据结构。
64. 叙述 Solaris 用户级线程的状态及其转换原因。
65. 叙述 Windows 2000/XP 中的进程和线程概念。
66. 试说明 Windows 2000/XP 中的进程对象。
67. 试说明 Windows 2000/XP 中的线程对象。
68. 叙述 Windows 2000/XP 中的线程状态及其转换原因。
69. 试说明 Windows 2000/XP 的作业对象。
70. 试说明访管指令与特权指令的区别。
71. 试说明访管指令与系统调用的联系和区别。
72. 处理器调度分哪几种类型？简述各类调度的主要任务。
73. 叙述衡量一个处理器调度算法好坏的主要标准。
74. 叙述作业调度和低级调度的关系。
75. 叙述中级调度的主要作用。
76. 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。
77. 简述批处理作业的组织、输入和调度过程。
78. 什么是 JCB，列举它的主要内容和作用。
79. 试叙述作业、进程、线程和程序三者的关系。
80. 试叙述：作业、作业步和作业流。
81. 何谓响应比最高优先算法？它有何主要特点？
82. 时间片轮转低级调度算法中，根据哪些因素确定时间片长短？
83. 优先权调度会否导致进程饥饿状态，为什么？
84. 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？
85. 为什么多级反馈队列算法能较好地满足各种用户的需求？
86. 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。
87. 试述剥夺式和非剥夺式低级调度策略。
88. 试述高级调度和低级调度各自所使用的“调度参数”。
89. 叙述典型的实时调度算法。
90. 试述典型的多 CPU 调度算法。
91. 列出并解释传统 UNIX 的动态优先数计算公式。
92. 叙述 UNIX SVR4 的处理器调度算法的主要特点。
93. 叙述 Windows 2000/XP 的处理器调度算法的主要特点。
94. 证明在非抢占式调度算法中，最短作业优先算法具有最小平均等待时间。
95. 通常进程产生了等待事件便进入该事件队列等待，如果想让一个进程同时等待不止一个事件，可以做到吗？若可以请设计出多事件队列。
96. 多级反馈队列中，给不同队列以大小不同的时间片值，意义何在？
97. 试叙述先来先服务调度算法和时间片轮转调度算法的本质区别？

98. 操作系统设计中强调把机制与策略分离,请建议一种调度机制(scheduling mechanism),允许父进程能控制其子进程的调度策略(scheduling policy)。

99. 现有一组进程被分别排入有三级的优先级队列(高、中、低),若各级之间采用优先级调度,而同级进程之间采用时间片轮转调度。试简述这批进程被调度执行的过程。

100. 有一个单向连接的进程队列,它的队首进程由系统队列标志指出,队尾进程的队列指引元为0。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。

101. 采用双向连接的进程队列。假定队首和队尾进程的后(前)向指引元由队列标志指出;且队首(尾)的前(后)向指引元为0”,写出任一进程入队和出队的工作流程。

102. 上题中进程若按优先数从高到低进行排队,写出进程入队和出队的工作流程。

103. 为什么说操作系统是由中断驱动的?

104. 采用时间片轮转调度时,每个进程在就绪队列中出现一次。如果一个进程在就绪队列中出现二次以上,情况会怎样?什么原因会出现这种情况?

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) 改变存储器映象图;(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. 并发进程之间有什么样的相互制约关系?下列日常生活中的活动是属哪种制约关系:(1) 踢足球,(2) 吃自助餐,(3) 图书馆借书,(4) 电视机生产流水工序。

4. 在按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。在处理器不断地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. 若后备作业队列中等待运行的同时有三个作业 J₁. J₂. J₃,已知它们各自的运行时间为 a、b、c,且满足 a < b < c,试证明采用短作业优先算法调度能获得最小平均作业周转时间。

6. 若有一组作业 J₁, …, J_n,其执行时间依次为 S₁, …, S_n。如果这些作业同时到达系统,并在一台单 CPU 处理器上按单道方式执行。试找出一种作业调度算法,使得平均作业周转时间最短。

7. 假定执行表中所列作业,作业号即为到达顺序,依次在时刻 0 按次序 1、2、3、4、5 进入单处理器系统。

1) 分别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强占优先权调度算法算出各作业的执行先后次序(注意优先权高的数值小);

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。

| 作业号 | 执行时间 | 优先权 |
|-----|------|-----|
| 1 | 10 | 3 |
| 2 | 1 | 1 |
| 3 | 2 | 3 |
| 4 | 1 | 4 |
| 5 | 5 | 2 |

8. 对某系统进行监测后表明平均每个进程在 I/O 阻塞之前的运行时间为 T 。一次进程切换的系统开销时间为 S 。若采用时间片长度为 Q 的时间片轮转法, 对下列各种情况算出 CPU 利用率。

- 1) $Q = \infty$ 2) $Q > T$ 3) $S < Q < T$ 4) $Q = S$ 5) Q 接近于 0

9. 有 5 个待运行的作业, 各自预计运行时间分别是: 9、6、3、5 和 x , 采用哪种运行次序使得平均响应时间最短?

10. 有 5 个批处理作业 A 到 E 均已到达计算中心, 其运行时间分别 2、4、6、8 和 10 分钟; 各自的优先级分别被规定为 1、2、3、4 和 5, 这里 5 为最高级。对于 1) 时间片轮转算法、2) 优先数法、3) 短作业优先算法、4) 先来先服务调度算法(按到达次序 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 1) 每个作业获得相同的时间片; 对 2) 到 4) 采用单道运行, 直到结束。)

11. 有 5 个批处理作业 A 到 E 均已到达计算中心, 其运行时间分别 10、6、2、4 和 8 分钟; 各自的优先级分别被规定为 3、5、2、1 和 4, 这里 5 为最高级。若不考虑系统切换开销, 计算出平均作业周转时间。(1) FCFS(按 A、B、C、D、E); (2) 优先级调度算法, (3) 时间片轮转法。

12. (1) 假定一个处理器正在执行两道作业, 一道以计算为主, 另一道以输入输出为主, 你将怎样赋予它们占有处理器的优先级? 为什么?

(2) 假定一个处理器正在执行三道作业, 一道以计算为主, 第二道以输入输出为主, 第三道为计算与输入输出均匀。应该如何赋予它们占有处理器的优先级使得系统效率较高?

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

14. 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。

15. 单道批处理系统中, 下列三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 | 带权周转时间 |
|------------------|-------|------|------|------|------|--------|
| 1 | 10:00 | 2:00 | | | | |
| 2 | 10:10 | 1:00 | | | | |
| 3 | 10:25 | 0:25 | | | | |
| 平均作业周转时间 = | | | | | | |
| 平均作业带权周转时间 $W =$ | | | | | | |

16. 若有如表所示四个作业进入系统, 分别计算在 FCFS、SJF 和 HRRF 算法下的平均周转时间与带权平均周转时间。(时间以十进制表示)

| 作业 | 提交时间 | 估计运行时间(小时) | 开始执行时间(小时) |
|----|------|------------|------------|
| 1 | 8.00 | 2.00 | 8.00 |
| 2 | 8.50 | 0.50 | 10.30 |
| 3 | 9.00 | 0.10 | 10.00 |
| 4 | 9.50 | 0.20 | 10.10 |

17. Kleinrock 提出一种动态优先权算法:进程在就绪队列等待时,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。给参数 α 、 β 赋以不同值可得到不同算法。(1)若 $\alpha > \beta > 0$ 是什么算法? (2) 若 $\alpha < \beta < 0$ 是什么算法?

18. 有一个四道作业的操作系统,若在一段时间内先后到达 6 个作业,它们的提交和估计运行时间由下表给出:

| 作业 | 提交时间 | 估计运行时间(分钟) |
|----|------|------------|
| 1 | 8:00 | 60 |
| 2 | 8:20 | 35 |
| 3 | 8:25 | 20 |
| 4 | 8:30 | 25 |
| 5 | 8:35 | 5 |
| 6 | 8:40 | 10 |

系统采用 SJF 调度算法,作业被调度进入系统后中途不会退出,但作业运行时可被更短作业抢占。(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。

19. 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的调度算法,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,作业优先数即为进程优先数,优先数越小优先级越高。

| 作业名 | 到达时间 | 估计运行时间 | 优先数 |
|-----|-------|--------|-----|
| A | 10:00 | 40 分 | 5 |
| B | 10:20 | 30 分 | 3 |
| C | 10:30 | 50 分 | 4 |
| D | 10:50 | 20 分 | 6 |

(1) 列出所有作业进入内存时间及结束时间。

(2) 计算平均周转时间。

20. 某多道程序设计系统供用户使用的主存为 100K, 磁带机 2 台, 打印机 1 台。采用可变分区内存管理, 采用静态方式分配外围设备, 忽略用户作业 I/O 时间。现有作业序列如下:

| 作业号 | 进入输入井时间 | 运行时间 | 主存需求量 | 磁带需求 | 打印机需求 |
|-----|---------|-------|-------|------|-------|
| 1 | 8:00 | 25 分钟 | 15K | 1 | 1 |
| 2 | 8:20 | 10 分钟 | 30K | 0 | 1 |
| 3 | 8:20 | 20 分钟 | 60K | 1 | 0 |
| 4 | 8:30 | 20 分钟 | 20K | 1 | 0 |
| 5 | 8:35 | 15 分钟 | 10K | 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的作业, 在主存中的各作业平分 CPU 时间。现求:(1)作业被调度的先后次序? (2)全部作业运行结束的时间? (3)作业平均周转时间为多少? (4)最大作业周转时间为多少?

21. 某多道程序设计系统采用可变分区内存管理, 供用户使用的主存为 200K, 磁带机 5 台。采用静态方式分配外围设备, 且不能移动在主存中的作业, 忽略用户作业 I/O 时间。现有作业序列如下:

| 作业号 | 进入输入井时间 | 运行时间 | 主存需求量 | 磁带需求 |
|-----|---------|-------|-------|------|
| A | 8:30 | 40 分钟 | 30K | 3 |
| B | 8:50 | 25 分钟 | 120K | 1 |
| C | 9:00 | 35 分钟 | 100K | 2 |
| D | 9:05 | 20 分钟 | 20K | 3 |
| E | 9:10 | 10 分钟 | 60K | 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? (4) SJF 算法选中作业执行的次序及作业平均周转时间?

22. 上题中, 若允许移动已在主存中的作业, 其他条件不变, 现求:(1) FIFO 算法选中作业执行的次序及作业平均周转时间? (4) SJF 算法选中作业执行的次序及作业平均周转时间?

23. 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)说明时钟中断时, 定时唤醒处理程序的处理过程。(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。

24. 一个实时系统有 4 个周期性事件, 周期分别为 50、100、300 和 250 ms。若假设其处理分别需要 35、20、10 和 x ms, 则该系统可调度允许的 x 值最大为多少?

第三章 并发进程

3.1 并发进程

3.1.1 顺序程序设计

早期,人们引入程序的概念,他们编写一个程序,在计算机上运行这个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顺序处理器上的执行是严格按序的,即只有当一个操作结束后,才能开始后继操作,这可以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,则这些不同程序在时间上也按调用次序严格有序执行,这可以称为程序外部的顺序性。上述计算过程就是程序的顺序执行过程,传统的程序设计方法是顺序程序设计(Sequential Programming),即把一个程序设计成一个顺序执行的程序模块,不同程序也是按序执行的。顺序程序设计具有如下的特性:

- 执行的顺序性。一个程序在顺序处理器上的执行是严格按序的,即每个操作必须在下一个操作开始之前结束。
- 环境的封闭性。在顺序处理情况下,运行程序独占系统全部资源,除初始状态之外,其所处的环境都是由程序本身决定的,只有程序本身的动作才能改变其环境,不会受到任何其他程序和外界因素的干扰。
- 执行结果的确定性。程序执行过程中允许出现中断,但这种中断对程序的最终结果没有影响,也就是说程序的执行结果与它的执行速率无关。
- 计算过程的可再现性。一个程序针对同一个数据集合一次执行的结果,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结果。这样当程序中出现了错误时,往往可以重现错误,以便进行分析。

顺序程序设计的顺序性、封闭性、确定性和再现性表明了程序与计算(程序的执行)是一对应的,给程序的编制、调试带来很大方便,其缺点是计算机系统效率不高。

3.1.2 进程的并发性

操作系统中引入并发程序设计技术后, 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, 程序外部的顺序性特性消失, 程序与计算不再一一对应, 所以, 操作系统中引进进程概念来描述这种变化。进程的并发性(Concurrency)是指一组进程的执行在时间上是重叠的。所谓执行在时间上是重叠的, 是指一个进程执行的第一条指令是在另一个进程执行的最后一条指令完成之前开始的。例如: 有两个进程 A 和 B, 它们分别执行操作 a_1, a_2, a_3 和 b_1, b_2, b_3 。在一个单处理器上, 就 A 和 B 两个进程而言, 它们的执行顺序分别为 a_1, a_2, a_3 和 b_1, b_2, b_3 , 这是进程执行的操作的顺序性。然而, 这两个进程在单处理器上它们的操作可能是交叉执行的, 如执行序列为 $a_1, b_1, a_2, b_2, a_3, b_3$ 或 $a_1, b_1, a_2, b_2, b_3, a_3$ 等, 则说 A 和 B 两个进程的执行是并发的。从宏观上来看, 并发性反映出一个时间段中有几个进程都处于运行还未运行结束的状态, 且这些进程都在同一处理器上运行, 但从微观上来看任一时刻仅有一个进程在处理器上运行。反过来看, 并发的实质是一个处理器在几个进程之间的多路复用, 并发是对有限的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, 以提高系统资源利用率。

在采用多道程序设计的系统中, 利用了外围设备与处理器, 外围设备与外围设备的并行工作能力, 从而, 提高了计算机的工作效率。怎样才能充分利用外围设备与处理器, 外围设备与外围设备的并行工作能力呢? 很重要的一个方面是取决于程序的编制。在图 3-1 的例子中, 由于程序是按照 `while(TRUE) {input, process, output}` 串行地输入—处理—输出的来编制的, 所以, 这个程序只能顺序地执行。这时系统的效率是相当低的。

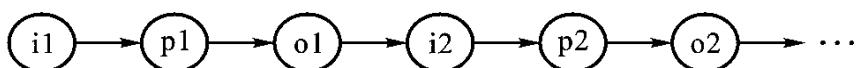


图 3-1 串行工作

如果把这个求解问题的程序分成三部分:

```

while(TRUE) {input, send}
while(TRUE) {receive, process, send}
while(TRUE) {receive, output}
  
```

每一部分称为一个程序(子)模块, 其中执行 `send` 和 `receive` 操作表明程序模块之间需要通信以便协调一致地工作, 于是这三个程序模块的功能是:

模块 1: 循环执行, 读入字符, 将读入字符送缓冲区 1。

模块 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。

模块 3: 循环执行, 取出缓冲区 2 中的计算结果并写到磁带上。

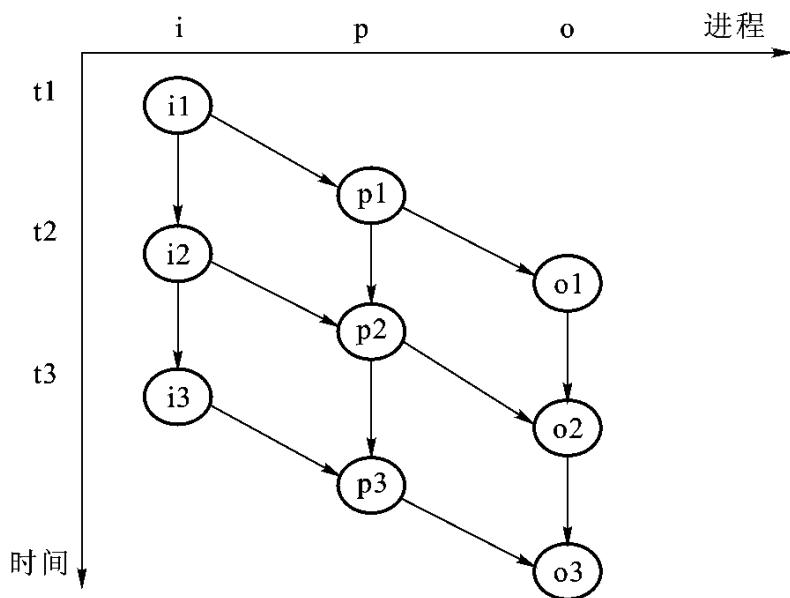


图 3-2 并行工作

从图 3-2 可以看出,这三个程序模块能同时执行,在 t_3 时刻输入 i_3 、处理 p_2 与输出 o_1 可以并行工作,在 t_4 、 t_5 等时刻同样可以并行工作。于是就得到了图 3-3 所示的情形,假定循环的次数为 n ,由于输入、处理和输出的重叠执行,处理器的使用效率是: $(52 \times n) / (78 \times n + 52 + 20)$ 。

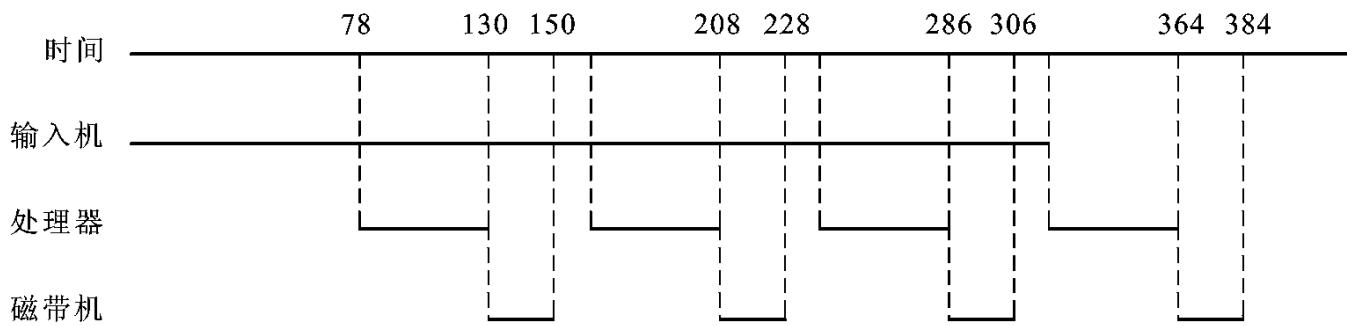


图 3-3 采用并发程序设计时处理器的使用的效率

当 n 趋向于无穷大时,处理器的使用效率是 67%。显然,这种程序设计方法发挥了处理器与外围设备的并行能力,从而,提高了计算机系统的效率。这种使一个程序分成若干个可同时执行的程序模块的方法称并发程序设计 (concurrent programming), 每个程序模块和它执行时所处理的数据就组成一个进程。

并发的进程可能是无关的,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。所以,一个进程的执行与其他并发进程的进展无关,即一个并发进程不会改变另一个并发进程的变量值。然而,交互的并发进程,它们共享某些变量,一个进程的执行可能影响其他进程的执行结果,可以认为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。该条件在 1966 年首先由 Bernstein 提出,又称之为 Bernstein 条件。下面作简单介绍:假设,

$R(p_i) = \{a_1, a_2, \dots, a_n\}$, 表示程序 p_i 在执行期间引用的变量集;

$W(p_i) = \{b_1, b_2, \dots, b_m\}$, 表示程序 p_i 在执行期间改变的变量集。

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交集之和为空集:

$$R(p_1) \& W(p_2) \cup R(p_2) \& W(p_1) \cup W(p_1) \& W(p_2) = \{\}$$

则并发进程的执行与时间无关。例如,有如下四条语句:

$S_1 : a := x + y$

$S_2 : b := z + 1$

$S_3 : c := a - b$

$S_4 : w := c + 1$

于是有: $R(S_1) = \{x, y\}$, $R(S_2) = \{z\}$, $R(S_3) = \{a, b\}$, $R(S_4) = \{c\}$; $W(S_1) = \{a\}$, $W(S_2) = \{b\}$, $W(S_3) = \{c\}$, $W(S_4) = \{w\}$ 。可见 S_1 和 S_2 可并发执行,因为满足 Bernstein 条件。其他语句间因变量交集之和非空,并发执行可能会产生与时间有关的错误。

采用并发程序设计技术构造的一组程序模块在执行时,具有如下的特征:

- 并发性,进程的执行在时间上可以重叠,在单处理器系统中可以并发执行;在多处理器环境中可以并行执行。
- 共享性,它们可以共享某些变量,通过引用这些共享变量就能互相交换信号,从而,程序的运行环境不再是封闭的。
- 制约性,进程并发执行或协作完成同一任务时,会产生相互制约关系,必须对它们并发执行的次序(即相对执行速率)加以协调。
- 交互性,由于进程共享某些变量,所以,一个程序的执行可能影响其他程序的执行结果。因此,这种交互必须是有控制的,否则会出现不正确的结果。即使程序自身能正确运行,由于程序的运行环境不再是封闭的,程序结果仍可能是不确定的,计算过程具有不可再现性。

采用并发程序设计的好处是:(1)若为单处理器系统,可有效利用资源,让处理器和 I/O 设备,I/O 设备和 I/O 设备同时工作,充分发挥机器部件的并行能力;(2)若为多处理器系统,可让进程在不同处理器上物理地并行工作,从而,加快计算速度;(3)简化了程序设计任务,一般地说,编制并发执行的小程序(模块)进度快,容易保证正确性。

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 I/O 设备的互等现象,提高系统效率。机器部件能并行工作仅仅有了提高效率的可能性,而机器部件并行工作

的实现还需要软件技术去利用和发挥,这种软件技术就是并发程序设计。并发程序设计是多道程序设计的基础,多道程序的实质就是把并发程序设计引入到单处理器的系统中。

3.1.3 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常常是不可预期的,甚至无法再现。这是因为两个并发进程执行的相对速度不可预测,交互进程的速率不仅受到处理器调度的影响,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。所以,一个进程的速度通常无法为另一个进程所知。因此,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的相对速度,各种与时间有关的错误就可能出现,与时间有关的错误有两种表现形式,一种是结果不惟一;另一种是永远等待。为了说明与时间有关的错误,现观察下面的例子。

例 1 (结果不惟一) 购买飞机票问题。

假设一个飞机订票系统有两个终端,分别运行进程 T1 和 T2。该系统的公共数据区中的一些单元 $A_j (j = 1, 2, \dots)$ 分别存放某月某日某次航班的余票数,而 x_1 和 x_2 表示进程 T1 和 T2 执行时所用的工作单元。飞机票售票程序如下:

```
process Ti (i = 1, 2)
var Xi: integer;
begin
    按旅客订票要求找到 Aj} ;
    Xi := Aj;
    if Xi > = 1 then begin Xi := Xi - 1; Aj := Xi; 输出一张票} ; end
        else 输出信息“票已售完”} ;
end;
```

由于 T1 和 T2 是两个可同时执行的并发进程,它们在同一个计算机系统中运行,共享同一批票源数据,因此,可能出现如下所示的运行情况。

| | |
|------------------------------------|----------------|
| T1: X1 := Aj; | X1 = m (m > 0) |
| T2: X2 := Aj; | X2 = m |
| T2: X2 := X2 - 1; Aj := X2; 输出一张票; | Aj = m - 1 |
| T1: X1 := X1 - 1; Aj := X1; 输出一张票; | Aj = m - 1 |

显然,此时出现了把同一张票卖给了两个旅客的情况,两个旅客可能各自都买到一张同天同次航班的机票,可是, A_j 的值实际上只减去了 1,造成余票数的不正确。特别是,当某次航班只有一张余票时,就可能把这一张票同时售给了两位旅客,这是不能允许的。

例 2 (永远等待)内存管理问题。

假定有两个并发进程 borrow 和 return 分别负责申请和归还主存资源, 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。并发进程算法及执行描述如下:

```

procedure borrow (var B:integer)
begin
    if B > x then {申请进程进入等待队列等主存资源};
    x := x - B;
    {修改主存分配表, 申请进程获得主存资源};
end;

procedure return (var B:integer)
begin
    x := x + B;
    {修改主存分配表};
    {释放等主存资源的进程};
end;

cobegin
    var x:integer;
    repeat borrow (var B:integer);
    repeat return(var B:integer);
coend

```

由于 borrow 和 return 共享了表示主存物理资源的临界变量 x , 对并发执行不加限制会导致错误。例如, 一个进程调用 borrow 申请主存, 在执行了比较 B 和 x 的指令后, 发现 $B > x$, 但在执行 {申请进程进入等待队列等主存资源} 前, 另一个进程调用 return 抢先执行, 归还了所借全部主存资源。这时, 由于前一个进程还未成为等待状态, return 中的 {释放等主存资源的进程} 相当于空操作。以后当调用 borrow 的进程被置成等主存资源时, 可能已经没有其他进程来归还主存资源了, 从而, 申请资源的进程处于永远等待状态。

3.1.4 进程的交互(Interaction Among Processes): 协作和竞争

在多道程序设计系统中, 同一时刻可能有许多进程, 这些进程之间存在两种基本关系: 竞争关系和协作关系。

第一种是竞争关系 系统中的多个进程之间彼此无关, 它们并不知道其他进程的存在, 并且也不受其他进程执行的影响。例如, 批处理系统中建立的多个用户进程, 分时系统中建立的多个终端进程。由于这些进程共用了一套计算机系统资源, 因而, 必然要出现多个进程

竞争资源的问题。当多个进程竞争共享硬设备、存储器、处理器和文件等资源时,操作系统必须协调好进程对资源的争用。

由于相互竞争资源的进程间并不交换信息,但是一个进程的执行可能影响到同其竞争资源的其他进程。如果两个进程要访问同一资源,那么,一个进程通过操作系统分配得到该资源,另一个将不得不等待。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。所以,资源竞争出现了两个控制问题:一个是死锁(deadlock)问题,一组进程如果都获得了部分资源,还想要得到其他进程所占有的资源,最终所有的进程将陷入死锁;另一个是饥饿(starvation)问题,这是指这样一种情况,一个进程由于其他进程总是优先于它而被无限期拖延。在并发运行的系统中,任何时刻都会产生申请资源,这就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,有些策略表面上看很有道理,却可能使某些进程永远得不到服务,但死锁并没有发生。例如,为了让尽量多的文件打印出来,系统规定了一种打印机分配策略:总是把打印机分配给打印文件最短的进程(假设每个打印文件的长度已知)。在一个繁忙的系统中,如果有一个进程生成了一个很大的文件需要打印,每当打印机空闲时,系统会检查所有打印文件的长度,并把打印机分配给打印文件最短的进程。如果不间断有新进程加入系统,并且它们都只打印小文件,那么,要打印大文件的进程总是分不到打印机。尽管这里没有产生死锁,但却出现了饥饿,即该进程被无限期推迟执行。解决饥饿问题的最简单策略是FCFS资源分配策略,在这种机制下,等待最久的进程会是下一个被调度的进程,随着时间的流失,每个进程会变成最“老”的进程,因而,能获得资源并完成任务。

由于操作系统负责资源分配,资源竞争的控制应由系统来解决,因而,操作系统应该提供各种支持。例如,提供锁机制给并发进程在使用共享资源之前来表达互斥的要求。操作系统需要保证诸进程能互斥地访问临界资源,既要解决饥饿问题,又要解决死锁问题。

进程的互斥(mutual exclusion)是解决进程间竞争关系(间接制约关系)的手段。进程互斥是指若干个进程要使用同一共享资源时,任何时刻最多允许一个进程去使用,其他要使用该资源的进程必须等待,直到占有资源的进程释放该资源。

临界区管理可以解决进程互斥问题,本章第二节将详细介绍临界区的解决方案。

第二种是协作关系 某些进程为完成同一任务需要分工协作,由于合作的每一个进程都是独立地以不可预知的速度推进,这就需要相互协作的进程在某些协调点上协调各自的工作。当合作进程中的一个到达协调点后,在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己,直到其他合作进程发来协调信号或消息后方被唤醒并继续执行。这种协作进程之间相互等待对方消息或信号的协调关系称为进程同步。前面曾经给出了一个例子, input、process、和 output 三个进程分工协作完成读入数据、加工处理和打印输出任务。这是一种典型的协作关系,各自都知道对方的存在。这时操作系统要确保诸进程在执行次序上协调一致,没有输入完一块数据之前不能加工处理,没有加工处理完一块数据之前不能打印输

出等等,每个进程都要接收到其他协作进程完成一次处理的消息后,才能进行下一步工作。进程间的协作可以是双方不知道对方名字的间接协作。例如,通过共享访问一个缓冲区进行松散式协作;也可以是双方知道对方名字,直接通过通信机制进行紧密协作。允许进程协同工作有利于共享信息、加快计算速度、实现模块化程序设计。

进程的同步(Synchronization)是解决进程间协作关系(直接制约关系)的手段。进程同步指两个以上进程基于某个条件来协调它们的活动。一个进程的执行依赖于另一个协作进程的消息或信号,当一个进程没有得到来自于另一个进程的消息或信号时则需等待,直到消息或信号到达才被唤醒。

不难看出,进程互斥关系是一种特殊的进程同步关系,即逐次使用互斥共享资源,也是对进程使用资源次序上的一种协调。

3.2 临界区管理

3.2.1 互斥和临界区

例1中的飞机票售票管理系统之所以会产生错误,原因在于多个售票进程交叉访问了共享变量 A_j 。这里把并发进程中与共享变量有关的程序段称为“临界区”(critical section),共享变量代表的资源叫“临界资源”(critical resource)。在售票管理系统中,进程 T1 的临界区为:

```
X1 := Aj;  
if X1 > = 1 then begin X1 := X1 - 1; Aj := X1;
```

进程 T2 的临界区为:

```
X2 := Aj;  
if X2 > = 1 then begin X2 := Y2 - 1; Aj := X2;
```

与同一变量有关的临界区分散在各有关进程的程序段中,而各进程的执行速度不可预知。如果能保证一个进程在临界区执行时,不让另一个进程进入相关的临界区,即各进程对共享变量的访问是互斥的,那么,就不会造成与时间有关的错误。

关于临界区的概念是由 Dijkstra 在 1965 年首先提出的。可以用与一个共享变量相关的临界区的语句结构来书写交互的并发进程,这里用 shared 说明共享变量。

shared variable

```
region variable do statement
```

一个进程执行到一个临界区的语句时,不管该进程目前是否正在运行,都说它是在临界区内。根据它们的临界区重写的售票管理进程如下。

```
shared Aj
process Ti ( i = 1, 2 )
var Xi:integer;
begin
按旅客定票要求找到 Aj;
region Aj do begin
Xi := Aj;
if Xi > = 1 then begin Xi:= Xi - 1; Aj:= Xi; {输出一张票};end
else {输出票已售完};
end;
end;
```

对若干个进程共享一个变量的相关临界区,有三个调度原则:

- 一次至多一个进程能够在它的临界区内;
- 不能让一个进程无限地留在它的临界区内;
- 不能强迫一个进程无限地等待进入它的临界区。特别,进入临界区的任一进程不能妨碍正等待进入的其他进程的进展;
- 可把临界区的调度原则总结成四句话:无空等待、有空让进、择一而入、算法可行。算法可行指不能因为所选的调度策略造成进程饥饿甚至死锁。
- 临界区是允许嵌套的,例如

```
region x do begin ... region y do ... end;
```

但是粗心的嵌套可能导致进程无限地留在它的临界区内,例如,如果又有一个进程执行:

```
region y do begin ... region x do ... end;
```

这样,当两个进程在大约差不多的时间进入了外层的临界区后,将发现它们每个都被排斥在内层临界区之外,造成无限地等待进入临界区。

3.2.2 临界区管理的尝试

这里先讨论用软件方法实现互斥,软件方法是为在具有一个处理器或共享主存的多处理器机器上执行的并发进程实现的,这些方法通常假设在存储器访问级的基本互斥,即对主存中同一个单元的同时访问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。前述要求实现临界区的管理,可采用一种标志的方式,即用标志来表示哪个进程可以进入临界区。然而,如何使用标志,仍是值得讨论的问题。下面的程序是第一种尝试,对进程 P1 和 P2 分别用标志 insidel 和 inside2 与其相连,当该进程在它的临界区内时其值为真(true),不在临界区时其值为假(false)。P1(P2)要进入它的临界区前先测试 inside2(insidel)以确保当前无进程在临界区,然后,把 insidel(inside2)置成 true,以封锁进程 P2(P1)进入临界区,直至 P1(P2)退出临界区,再将相应标志置成 false。

```

insidel,inside2:Boolean;
inside1 := false;      {P1 不在其临界区内}
inside2 := false;      {P2 不在其临界区内}
cobegin
process P1
begin
    while inside2 do begin end;
    insidel := true;
    临界区;
    insidel := false;
end;
process P2
begin
    while insidel do begin end;
    inside2 = true;
    临界区;
    inside2 := false;
end;
coend.

```

但是,这种管理是不正确的,原因是在 P1(P2) 测试 inside2(insidel) 与随后置 insidel(inside2) 之间,并发进程 P2(P1) 可能发现 insidel(inside2) 有值 false,于是它将置 inside2(insidel) 为 true,并且与 P1(P2) 同时进入临界区。

第二种尝试是对第一种作如下修正:延迟 P1(P2) 对 inside2(insidel) 的测试,而先置 insidel(inside2) 为 true 以封锁 P2(P1)。修正后的程序如下。不幸,它也是无效的。因为,有可能每个进程都把它的标志置成 true,从而,出现死循环。这时,没有一个进程能在有限的时间内进入临界区,造成了永远等待。

```

insidel,inside2:boolean;
inside1 := false;      {P1 不在其临界区内}

```

```

inside2 := false;      {P2 不在其临界区内}

cobegin
process P1
begin
    inside1 := true;
    while inside2 do begin end;
    临界区;
    inside1 := false;
end;
process P2
begin
    inside2 := true;
    while inside1 do begin end;
    临界区;
    inside2 := false;
end;
coend.

```

3.2.3 实现临界区管理的软件方法

1. Dekker 算法

荷兰数学家 T. Dekker 算法能保证进程互斥地进入临界区, 这是最早提出的一个软件互斥方法, 此方法用一个指示器 turn 来指示应该哪一个进程进入临界区。若 $turn = 1$ 则进程 P1 可以进入临界区; 若 $turn = 2$ 则进程 P2 可以进入临界区。Dekker 算法如下:

```

var inside : array[1..2] of Boolean;
    Turn : integer;
    turn := 1 or 2;
    inside[1]:= false;
    inside[2]:= false;

cobegin
process P1
begin
    inside[1]:= true;
    while inside[2] do
        if turn = 2 then begin

```

```

        inside[1] := false;
        while turn = 2 do begin end;
        inside[1] := true;
    end

临界区;

turn = 2;
inside[1] := false;
end;
process P2
begin
    inside[2] := true;
    while inside[1] do
        if turn = 1 then begin
            inside[2] := false;
            while turn = 1 do begin end;
            inside[2] := true;
        end
   临界区;
    turn = 1;
    inside[2] := false;
end;
coend.

```

Dekker 算法的执行过程描述如下：当进程 P1 (或 P2) 想进入自己的临界区时，它把自己的标志位 inside1 (或 inside2) 置为 true，然后，继续执行并检查对方的标志位。如果对方的标志位为 false，表明对方不在也不想进入临界区，进程 P1 (或 P2) 可以立即进入自己的临界区；否则，咨询指示器 turn，若 turn 为 1 (或为 2)，那么 P1 (或 P2) 知道应该自己进入，从而反复地去测试 P2 (或 P1) 的标志值 inside2 (或 inside1)；进程 P2 (或 P1) 注意到应该礼让对方，故把其自己的标志位置为 false，允许进程 P1 (或 P2) 进入临界区；在进程 P1 (或 P2) 结束其临界区工作后，把自己的标志置为 false，且把 turn 置为 2 (或 1)，从而，把进入临界区的权力交给进程 P2 (或 P1)。

这种方法显然能保证互斥进入临界区的要求，这是因为仅当 $turn = i$ ($i = 1, 2$) 时进程 P_i ($i = 1, 2$) 才能有权力进入其临界区。因此，一次只有一个进程能进入临界区，且在一个进程退出临界区之前，turn 的值是不会改变的，保证不会有另一个进程进入相关临界区。同时，turn 的值不是 1 就是 2，故不可能同时出现两个进程均在 while 语句中等待而进不了临界区。Dekker 算法虽能解决互斥问题，但算法复杂难于理解，与之相比 peterson 解法较为简单。

2. Peterson 算法

在 1981 年, G.L.Perterson 提出了一个简单得多的软件互斥算法来解决互斥进入临界区的问题。此方法为每个进程设置一个标志, 当标志为 false 时表示该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 算法的程序如下:

```

var inside:array[1..2] of boolean;
turn:integer;
turn := 1 or 2;
inside[1] := false;           {P1 不在其临界区内}
inside[2] := false;           {P2 不在其临界区内}

cobegin
process P1
begin
    inside[1]:= true;
    turn := 2;
    while (inside[2] and turn = 2)
        do begin end;
    临界区;
    inside[1]:= false;
end;
process P2
begin
    inside[2]:= true;
    turn := 1;
    while (inside[1] and turn = 1)
        do begin end;
    临界区;
    inside[2]:= false;
end;
coend.

```

在上面的程序中, 用对 turn 的置值和 while 语句来限制每次最多只有一个进程可以进入临界区, 当有进程在临界区执行时不会有另一个进程闯入临界区; 进程执行完临界区程序后, 修改 inside[i] 的状态而使等待进入临界区的进程可在有限的时间内进入临界区。所以, Peterson 算法满足了对临界区管理的三个条件。由于在 while 语句中的判别条件是‘ inside[i]

和 $turn = 1$ (或 2)”，因此，任意一个进程进入临界区的条件是对方不在临界区或对方不请求进入临界区。于是，任何一个进程均可以多次进入临界区。本算法也很容易推广到 n 个进程的情况。

3.2.4 实现临界区管理的硬件设施

在单处理器计算机中，并发进程不会同时，只会交替执行。为了保证互斥，仅需要保证一个进程不被中断就可以了。分析临界区管理尝试中的两种算法，问题出在管理临界区的标志时要用到两条指令。而这两条指令在执行过程中有可能被中断，从而，导致了执行的不正确。能否把标志看作为一个锁，开始时锁是打开的，在一个进程进入临界区时便把锁锁上以封锁其他进程进入临界区，直至它离开其临界区，再把锁打开以允许其他进程进入临界区。如果希望进入其临界区的一个进程发现锁未开，它将等待，直到锁被打开。可见，要进入临界区的每个进程必须首先测试锁是否打开，如果是打开的则应立即把它锁上，以排斥其他进程进入临界区。显然，测试和上锁这两个动作不能分开，以防两个或多个进程同时测试到允许进入临界区的状态。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之一是关中断。当进入锁测试之前关闭中断，直到完成锁测试并上锁之后再开中断。进程在临界区执行期间，计算机系统不响应中断。因此，不会转向调度，也就不会引起进程或线程切换，保证了锁测试和上锁操作的连续性和完整性，也就保证了互斥，有效地实现了临界区管理。关中断方法有许多缺点：滥用关中断权力可能导致严重后果；关中断时间过长会影响系统效率，限制了处理器交叉执行程序的能力；关中断方法也不适用于多 CPU 系统，因为，在一个处理器上关中断，并不能防止进程在其他处理器上执行相同临界段代码。

2. 测试并建立指令

实现这种管理的另一种办法是使用硬件提供的“测试并建立”指令 TS (Test and Set)。可把这条指令看作为函数过程，它有一个布尔参数 x 和一个返回条件码，当 $TS(x)$ 测到 x 为 true 时则置 x 为 false，且根据测试到的 x 值形成条件码。下面给出了 TS 指令的处理过程。

$TS(x)$ ：若 $x = \text{true}$ ，则 $x := \text{false}$ ；return true；否则 return false；

用 TS 指令管理临界区时，可把一个临界区与一个布尔变量 s 相连，由于变量 s 代表了临界资源的状态，可把它看成一把锁。 s 初值置为 true，表示没有进程在临界区内，资源可用。系统可以提供在锁上的利用 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到 $TS(s)$ 为 true，此时 s 的值一定为 false；当进程退出临界区时，把 s 置为 true。由于 TS 指

令是一个不可分指令,在测试和形成条件码之间不可能有另一进程去测试 x 值,从而,保证了临界区管理的正确性。

```

s : boolean;
s := true;
process Pi                                {i = 1,2,...,n}
    pi : boolean;
begin
    repeat pi := TS(s) until pi;          {上锁}
    临界区;
    s := true;                           {开锁}
end;

```

3. 对换指令

对换(Swap)指令的功能是交换两个字的内容,处理过程描述如下:

```
Swap (a, b): temp := a; a := b; b := temp;
```

在 Intel 80x86 中,对换指令称为 XCHG 指令。用对换指令可以简单有效地实现互斥,方法是为每个临界区设置一个布尔锁变量。例如,称为 lock,当其值为 false 时表示无进程在临界区,实现进程互斥的程序如下:

```

lock : boolean;
lock := false;
process Pi                                {i = 1,2,...,n}
    pi : boolean;
begin
    pi := true;
    repeat Swap(lock, pi) until pi = false; {上锁}
    临界区;
    lock := false;                         {开锁}
end;

```

3.3 信号量与 PV 操作

3.3.1 同步和同步机制

下面通过例子来进一步阐明进程同步的概念。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、接收进程等等。解决好了生产者——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, 连接在一个有 k 个单位缓冲区的有界环形缓冲上, 故又叫有界缓冲问题。其中, pi 和 cj 都是并发进程, 只要缓冲区未满, 生产者 pi 生产的产品就可投入缓冲区; 类似地, 只要缓冲区不空, 消费者进程 cj 就可从缓冲区取走并消耗产品。可以用程序把生产者——消费者问题的算法描述如下:

```

var k:integer;
type item:any;
buffer:array[0..k-1] of item;
in,out:integer:=0;
counter:integer:=0;

process producer
  while (TRUE)
    produce an item in nextp;           {无限循环}
    if (counter = = k) sleep();         {生产一个产品}
    buffer[in]:=nextp;                 {缓冲满时, 生产者睡眠}
    in:=(in+1) mod k;                  {将一个产品放入缓冲区}
    counter:=counter+1;                {指针推进}
    if (counter = = 1) wakeup( consumer); {缓冲内产品数加 1}
                                         {缓冲为空了, 加进一件产品
                                         并唤醒消费者}

process consumer
  while (TRUE)
    if (counter = = 0) sleep();         {无限循环}
    nextc:=buffer[out];               {缓冲区空, 消费者睡眠}
    out:=(out+1) mod k;                {取一个产品到 nextc}
    counter:=counter-1;                {指针推进}
    if (counter = = k-1) wakeup( producer); {取走一个产品, 计数减 1}
                                         {缓冲满了, 取走一件产品
                                         并唤醒生产者}

    consume thr item in nextc;        {消耗产品}

```

其中, 假如一般的高级语言都有 `sleep()` 和 `wakeup()` 这样的系统调用。从上面的程序可以看出, 算法是正确的, 两组进程顺序执行结果也正确。但若并发执行, 就会出现错误结果, 出错的根源在于进程之间共享了变量 `counter`, 对 `counter` 的访问未加限制。

生产者和消费者进程对 `counter` 的交替执行会使其结果不惟一。例如, `counter` 当前值为

8,如果生产者生产了一件产品,投入缓冲区,拟做 counter 加 1 操作。同时消费者获取一个产品消费,拟做 counter 减 1 操作。假如两者交替执行加或减 1 操作,取决于它们的进行速度,counter 的值可能是 9,也可能是 7,正确值应为 8。

更为严重的是生产者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 发现它为 0。此时调度程序暂停消费者让生产者运行,生产者加入一个产品,将 counter 加 1,现在 counter 等于 1 了。它想当然地推想由于 counter 刚刚为 0,所以,此时消费者一定在睡眠,于是生产者调用 wakeup 来唤醒消费者。不幸的是,消费者还未去睡觉,唤醒信号被丢失掉。当消费者下次运行时,因已测到 counter 为 0,于是去睡眠。这样生产者迟早会填满缓冲区,然后去睡觉,形成了进程都永远处于睡眠状态。

出现不正确结果不是因为并发进程共享了缓冲区,而是因为它们访问缓冲区的速率不匹配。或者说 p_i, c_j 的相对速度不协调,需要调整并发进程的进行速度。并发进程间的这种制约关系称进程同步,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率,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,迄今已设计出许多种同步机制,本书中将介绍几种最常用的同步机制:信号量及 PV 操作,管程和消息传递。

3.3.2 记录型信号量与 PV 操作

前一节介绍的种种方法包括硬件设施方法和软件方法,硬件方法简单且行之有效,能保证互斥,可正确解决临界区调度问题,但有明显缺点;软件算法太复杂,效率低下;硬件设施采用忙式等待(busy waiting)测试,白白浪费 CPU 时间。将测试能否进入临界区的责任推给各个竞争的进程,会削弱系统的可靠性,加重了用户编程负担。

1965 年荷兰的计算机科学家 E.W.Dijkstra 提出了新的同步工具——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交通的方法引入操作系统,让两个或多个进程通过特殊变量展开交互。一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值,通过特殊变量这一设施,任何复杂的进程交互要求可得到满足,这种特殊变量就是信号量(semaphore)。为了通过信号量传送信号,进程可以通过 P、V 两个特殊的操作来发送和接收信号,如果进程相应的信号仍然没有送到,进程被挂起直到信号到达为止。

在操作系统中,信号量用以表示物理资源的实体,它是一个与队列有关的整型变量。实现时,信号量是一种变量类型,常常用一个记录型数据结构表示,它有两个分量:一个是信号量的值,另一个是信号量队列的队列指针。

除赋初值外,信号量仅能由同步原语对其进行操作,没有任何其他方法可以检查和操作信号量。原语是操作系统内核中执行时不可中断的过程、即原子操作(atomic operation)。

Dijkstra 发明了两个信号量操作原语:P 操作和 V 操作(荷兰语中“测试(Proberen)”和“增量(Verhogen)”的头字母),此外,还常用的符号有:wait 和 signal;up 和 down;sleep 和 wakeup 等。本书中采用 Dijkstra 最早论文中使用的符号 P 和 V。利用信号量和 P、V 操作既可以解决并发进程的竞争问题,又可以解决并发进程的协作问题。

信号量按其用途可分为两种:

- 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 和 V 操作。初值常常为 1,用于实现进程互斥。
- 私有信号量,联系一组并发进程,仅允许此信号量拥有的进程执行 P 操作,而其他相关进程可在其上施行 V 操作。初值常常为 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:

- 二元信号量,仅允许取值为 0 和 1,主要用于解决进程互斥问题。
- 一般信号量,允许取值为非负整数,主要用于解决进程同步问题。

下面讨论整形信号量、记录型信号量和二元信号量。

1. 整型信号量

设 s 为一个正整形量,除初始化外,仅能通过 P、V 操作来访问它,这时 P 操作原语和 V 操作原语定义如下:。

- P(s):当信号量 s 大于 0 时,把信号量 s 减去 1,否则调用 P(s)的进程等待直到信号量 s 大于 0 时。
- V(s):把信号量 s 加 1。

P(s)和V(s)可以写成:

```

P(s) : while s <= 0 do null operation
        s := s - 1;
V(s) : s := s + 1;
    
```

整型信号量机制中的 P 操作,只要信号量 $s \leq 0$,就会不断测试,进程处于“忙式等待”。后来对整型信号量进行了扩充,增加了一个等待 s 信号量所代表资源的等待进程的队列,以实现让权等待,这就是下面要介绍的记录型信号量机制。

2. 记录型信号量

设 s 为一个记录型数据结构,其中一个分量为整形量 value,另一个分量为信号量队列 queue,value 通常是一个具有非负初值的整型变量,queue 是一个初始状态为空的进程队列。这时 P 操作原语和 V 操作原语的定义修改如下:

- P(s):将信号量 s 减去 1,若结果小于 0,则调用 P(s)的进程被置成等待信号量 s 的状态。
- V(s):将信号量 s 加 1,若结果不大于 0,则释放一个等待信号量 s 的进程。

记录型信号量和 P 操作、V 操作可表示成如下的数据结构和不可中断过程：

```

type semaphore = record
    value: integer;
    queue: list of process;
end
procedure P(var s:semaphore);
begin
    s.value := s.value - 1;           {把信号量减去 1 }
    if s.value < 0 then W(s.queue);   {若信号量小于 0, 则执行 P(s) 的进程调
                                         用 W(s.queue) 进行自我封锁, 被置成等待
                                         信号量 s 的状态, 进入信号量队列 queue}
end;
procedure V(var s:semaphore);
begin
    s.value := s.value + 1;           {把信号量加 1 }
    if s.value ≤ 0 then R(s.queue);  {若信号量小于等于 0, 则调用 R(s.queue)
                                         从信号量 s 队列 queue 中释放一个等待
                                         信号量 s 的进程并置成就绪态}
end;

```

其中 $W(s.queue)$ 表示把调用过程的进程置成等待信号量 s 的状态，并链入 s 信号量队列，同时释放 CPU； $R(s.queue)$ 表示释放一个等待信号量 s 的进程，从信号量 s 队列中移出一个进程，置成就绪态并投入就绪队列。进程按照什么次序从队列中移出？公平的策略是先进先出法，被阻塞时间最久的进程最先从队列释放，该策略能保证进程不会被饿死。信号量 s 的初值可定义为 0, 1 或其他整数，在系统初始化时确定。从信号量和 P、V 操作的定义可以获得如下推论：

推论 1：若信号量 s 为正值，则该值等于在封锁进程之前对信号量 s 可施行的 P 操作数，亦即等于 s 所代表的实际还可以使用的物理资源数。

推论 2：若信号量 s 为负值，则其绝对值等于登记排列在该信号量 s 队列之中等待的进程个数，亦即恰好等于对信号量 s 实施 P 操作而被封锁起来并进入信号量 s 等待队列的进程数。

推论 3：通常，P 操作意味着请求一个资源，V 操作意味着释放一个资源。在一定条件下，P 操作代表挂起进程操作，而 V 操作代表唤醒被挂起进程的操作。

3. 二元信号量

设 s 为一个记录型数据结构，其中一个分量为 $value$ ，它仅能取值 0 和 1，另一个分量为信号

量队列 queue, 这时可以把二元(binary)信号量上的 P、V 操作记为 BP 和 BV, 其定义如下:

```

type binary semaphore = record
    value(0,1);
    queue: list of process
end;
procedure BP (var s:semaphore);
    if s.value = 1;
        then
            s.value = 0;
    else begin
        w(s.queue);
    end;
procedure BV (var s:semaphore);
    if s.queue is empty;
        then
            s.value := 1;
    else begin
        R(s.queue);
    end;

```

虽然二元信号量仅能取 0 和 1 值, 但可以证明它与记录型信号量一样, 有着同等的表达能力。

3.3.3 用记录型信号量实现互斥

记录型信号量和 P、V 操作可以用来解决进程互斥问题。与 TS 指令相比较, P、V 操作也是用测试信号量的办法来决定是否能进入临界区, 但不同的是 P、V 操作只对信号量测试一次, 而用 TS 指令则必须反复测试。用信号量和 P、V 操作管理几个进程互斥进入临界区的一般形式如下:

```

Var mutex: semaphore;
    mutex := 1;
cobegin
    .....
process Pi
    begin
        .....

```

```

P(mutex);
临界区;
V(mutex) .....
end;
.....
coend;

```

下面的程序用记录型信号量和 P、V 操作解决了飞机票售票问题。

```

Var A : ARRAY[1..m] of integer;
mutex : semaphore;
mutex:= 1;
cobegin
process Pi
var Xi:integer;
begin
L1:
按旅客定票要求找到 A[j];
P(mutex);
Xi := A[j];
if Xi > = 1
then begin
Xi:= Xi - 1;A[j]:= Xi;
V(mutex); 输出一张票};
end;
else begin
V(mutex); 输出“票已售完”};
end;
goto L1;
end;
coend.

```

```

Var A : ARRAY[1..m] of integer;
s : ARRAY[1..m] of semaphore;
s[j] := 1;
cobegin
process Pi
var Xi:integer;
begin
L1:
按旅客定票要求找到 A[j];
P(s[j]);
Xi := A[j];
if Xi > = 1
then begin
Xi:= Xi - 1;A[j]:= Xi;
V(s[j]); 输出一张票};
end;
else begin
V(s[j]); 输出“票已售完”};
end;
goto L1;
end;
coend.

```

左面的程序引入一个信号量 mutex，用于管理票源数据，其初值为 1。假设进程 T1 首先调用 P 操作，则 P 操作将把信号量 mutex 减 1，T1 进入临界区；此时，若进程 T2 也想进入临界区而调用 P 操作，那么，P 操作便会阻塞 T2 并使它等待 mutex。当 T1 离开临界区时，它调用 V 操作，V 操作将唤醒等待 mutex 的进程 T2；于是 T2 就可进入临界区执行。事实上，当进程 T1 和 T2 只有同时买一个航班的机票时才会发生与时间有关的错误，因此，临界区应该是与

$A[j]$ 有关的。所以, 可以对左面的程序进行改进, 引入一组信号量 $s[j]$, 从而, 得到了右面的程序。不难看出, 它提高了进程的并发程度。

要提醒注意的是: 任何粗心地使用 P、V 操作会违反临界区的管理要求。如忽略了 else 部分的 V 操作, 将致使进程在临界区中判到条件不成立时无法退出临界区, 而违反了对临界区的管理要求。

若有两个进程在等待进入临界区的队列中排队, 当允许一个进程进入临界区时, 应先唤醒哪一个进程进入临界区? 一个使用 P、V 操作的程序, 如果它是正确的话, 那么, 这种唤醒应该是无选择的。在证明使用 P、V 操作的程序的正确性时, 必须证明进程按任意次序进入临界区都不影响程序的正确执行。

下面再来讨论使用信号量和 P、V 操作解决操作系统经典的五个哲学家吃通心面问题 (Dijkstra, 1965)。有五个哲学家围坐在一圆桌旁, 桌子中央有一盘通心面, 每人面前有一只空盘子, 每两人之间放一把叉子。每个哲学家思考、饥饿、然后, 欲吃通心面。为了吃面, 每个哲学家必须获得两把叉子, 且每人只能直接从自己左边或右边去取叉子。

在这道经典题目中, 每一把叉子都是必须互斥使用的, 因此, 应为每把叉子设置一个互斥信号量 S_i ($i = 0, 1, 2, 3, 4$), 初值均为 1。当一个哲学家吃通心面之前必须获得自己左边和右边的两把叉子, 即执行两个 P 操作; 吃完通心面后必须放下叉子, 即执行两个 V 操作。程序如下:

```

var fork[i] :array[0..4] of semaphore;
forki := 1;
cobegin
process Pi                                // i = 0, 1, 2, 3, 4
begin
L1:
    思考;
    P(fork[i]);
    P(fork[i + 1] mod 5);
    吃通心面;
    V(fork[i]);
    V(fork[i + 1] mod 5);
    goto L1;
end;

```

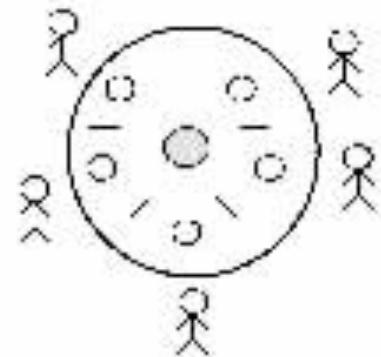


图 3-4 五个哲学家吃通心面问题

coend.

上述解法中,如果第五个哲学家先执行 $P(fork[4])$,再执行 $P(fork[0])$ 的话,就有可能出现每个哲学家举起右边一把叉子,却又在永远等待相邻哲学家手中的叉子的情况。有若干种办法可避免这类死锁:

- 至多允许四个哲学家同时吃。
- 奇数号先取左手边的叉子,然后再取右手边的叉子;偶数号先取右手边的叉子,然后,再取左手边的叉子。
- 每个哲学家取到手边的两把叉子才吃,否则,一把叉子也不取。

3.3.4 记录型信号量解决生产者—消费者问题

记录型信号量和 P、V 操作不仅可以解决进程的互斥,而且更是实现进程同步的有力工具。进程的同步是指一个进程的执行依赖于另一个进程的信号或消息,当一个进程没有得到来自于另一个进程的信号或消息时则等待,直到信号或消息到达才被唤醒。

生产者和消费者问题就是一个典型的进程同步问题,出现不正确结果的原因在于它们访问缓冲器的相对速率。为了能使它们正确工作,生产者和消费者必须按一定的生产率和消费率来访问共享的缓冲器。用 P、V 操作来解决生产者和消费者共享一个缓冲器的问题,可以使用两个信号量 empty 和 full,它们的初值分别为 1 和 0,empty 指示能否向缓冲器内存放产品,full 指示是否能从缓冲器内取出产品。于是生产者和消费者问题的程序如下所示。

```

var B : integer;
    empty:semaphore;           {可以使用的空缓冲区数}
    full:semaphore;            {缓冲区内可以使用的产品数}
    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
begin
L1:
    Produce a product;
    P(empty);
    B := product;
    V(full);
    Goto L1;
end;

```

```

process consumer
begin
    L2:
    P(full);
    Product := B;
    V(empty);
    Consume a product;
    Goto L2;
end;
coend.

```

要提醒注意的是 P、V 操作使用不当的话,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 件产品的缓冲器。为了使它们能协调的工作,必须使用一个信号量 mutex(初值为 1),以限制它们互斥地对缓冲器进行存取,另用两个信号量 empty(初值为 k) 和 full(初值为 0),以保证生产者不往满的缓冲器中存产品,消费者不从空的缓冲器中取产品。程序如下:

```

var B : array[0..k-1] of item;
empty:semaphore:=k;                      {可以使用的空缓冲区数}
full:semaphore:=0;                        {缓冲区内可以使用的产品数}
mutex:semaphore:=1;                       {互斥信号量}
in:integer:=0;                            {放入缓冲区指针}
out:integer:=0;                           {取出缓冲区指针}

cobegin
process producer_i
begin
    L1:produce a product;
    P(empty);
    P(mutex);
    B[putptr]:=product;
    in:=(in+1) mod k;
    V(mutex);
    V(full);
    goto L1;
end;
process consumer_j
begin

```

```

L2:P(full);
P(mutex);
Product := B[out];
out := (out + 1) mod k;
V(mutex);
V(empty);
consume a product;
goto L2;
end;
coend.

```

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,当缓冲器中存满了 k 件产品(此时, $\text{empty} = 0$, $\text{mutex} = l$, $\text{full} = k$)时,生产者又生产了一件产品,它欲向缓冲器存放时将在 $P(\text{sput})$ 上等待(注意,现在 $\text{sput} = 0$),但它已经占有了使用缓冲器的权力。这时消费者欲取产品时将停留在 $P(\text{mutex})$ 上得不到使用缓冲器的权力。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使用缓冲器的权力,这种相互等待永远结束不了。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 操作的次序倒是无关紧要的。一般来说,用于互斥的信号量上的 P 操作,总是在后面执行。

下面再来研究一个较为复杂的生产者—消费者问题。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),妈妈专向盘子中放桔子(orange),一个儿子专等吃盘子中的桔子,一个女儿专等吃盘子中的苹果。

这个问题实际上是两个生产者和两个消费者被连结到仅能放一个产品的缓冲器上。生产者各自生产不同的产品,但就其本质而言,他们是同一类生产者。而消费者则各自取需要的产品消费,他们的消费方式不同。实现爸爸、妈妈、儿子和女儿正确同步工作的程序如下:

```

var
plate : integer;
sp:semaphore; { 盘子里可以放几个水果 }
sg1:semaphore; { 盘子里有桔子 }
sg2:semaphore; { 盘子里有苹果 }
sp := 1; { 盘子里允许放入一个水果}
sg1:= 0; { 盘子里没有桔子 }
sg2:= 0; { 盘子里没有苹果}

cobegin
process father

```

```
begin
    L1: 削一个苹果;
    P(sp);
    把苹果放入 plate;
    V(sg2);
    goto L1;
end;

process mother
begin
    L2: 剥一个桔子;
    P(sp);
    把桔子放入 plate;
    V(sg1);
    goto L2;
end;

process son
begin
    L3: P(sg1);
    从 plate 中取桔子;
    V(sp);
    吃桔子;
    goto L3;
end;

process daughter
begin
    L4: P(sg2);
    从 plate 中取苹果;
    V(sp);
    吃苹果;
    goto L4;
end;

coend.
```

3.3.5 记录型信号量解决读者—写者问题

读者与写者问题 (reader – writer problem) (Courtois, 1971) 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，共享一个文件 F，要求：(1) 允许多个读者可同时对文件执行读操作；(2) 只允许一个写者往文件中写信息；(3) 任一写者在完成写操作之前不允许其他读者或写者工作；(4) 写者执行写操作前，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读者与写者问题，必须引入计数器 rc 对读进程计数，mutex 是用于对计数器 rc 操作的互斥信号量，W 表示是否允许写的信号量，于是管理该文件的程序可如下设计：

```

var rc: integer;
W, mutex: semaphore;
rc := 0;                                {读进程计数}
W := 1;
mutex := 1;

procedure read;
begin
P(mutex);
rc := rc + 1;
if rc = 1 then P(W);
V(mutex);
读文件;
P(mutex);
rc := rc - 1;
if rc = 0 then V(W);
V(mutex);
end;
procedure write;
begin
P(W);
写文件;
V(W);
end;
cobegin
process readeri;
process writerj;
coend.
process readeri;
begin

```

```

read;
end.

process writerj
begin
    write;
end.

```

在上面的解法中,读者是优先的。当存在读者时,写操作将被延迟,并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,导致了写进程长时间等待,并有可能出现写进程被饿死。增加信号量并修改上述程序可以得到写进程具有优先权的解决方案,能保证当一个写进程声明想写时,不允许新的读进程再访问共享文件。

为了有效解决读者——写者问题,有的操作系统专门引进了读者——写者锁。读者——写者锁允许多个读者同时以只读方式存取有锁保护的对象;或一个写者以写方式存取有锁保护的对象。当一个或多个读者已上锁后,此时形成了读锁,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,形成了写状态,所有其他进程的读写操作必须等待。Solaris 提供的 reader-writer 锁有:rw-enter() (上锁)、rw-exit() (开锁)、rw-tryenter() (忙式测试上锁)、rw-downgrade() (把 write lock 转换成 read lock) 和 rw-trygrade() (把 read lock 转换成 write lock)。

3.3.6 记录型信号量解决理发师问题

另一个经典的进程同步问题是理发师问题。理发店里有一位理发师、一把理发椅和 n 把供等候理发的顾客坐的椅子。如果没有顾客,理发师便在理发椅上睡觉。当一个顾客到来时,它必须叫醒理发师。如果理发师正在理发时又有顾客来到,那么,如果有空椅子可坐,顾客就坐下来等待,否则就离开理发店。

给出的解法引入 3 个信号量和一个控制变量:控制变量 waiting 用来记录等候理发的顾客数,初值均为 0;信号量 customers 用来记录等候理发的顾客数,并用作阻塞理发师进程,初值为 0;信号量 barbers 用来记录正在等候顾客的理发师数,并用作阻塞顾客进程,初值为 0;信号量 mutex 用于互斥,初值为 1。用信号量解决理发师问题的程序如下:

```

var waiting : integer;           等候理发的顾客数}
                                为顾客准备的椅子数}
CHAIRS:integer;
customers, barbers, mutex : semaphore;
customers := 0; barbers := 0;
waiting := 0; mutex := 1;
procedure barber;

```

```

begin
while(TRUE);
P(customers);          {理完一人, 还有顾客吗?}
P(mutex);              {若无顾客, 理发师睡眠}
waiting := waiting - 1; {进程互斥}
V(customers);          {等候顾客数少一个}
V(mutex);              {理发师去为一个顾客理发}
cut-hair();            {开放临界区}
end;                   {正在理发}

procedure customer
begin
P(mutex);              {进程互斥}
if waiting < CHAIRS   {看看有没有空椅子}
begin
waiting := waiting + 1; {等候顾客数加 1}
V(customers);           {必要的话唤醒理发师}
V(mutex);              {开放临界区}
P(barbers);            {找理发师, 顾客坐着养}
get-haircut();          {一个顾客坐下等理发}
end;
else V(mutex);          {人满了, 走吧!}
end.

```

3.4 管程

3.4.1 管程和条件变量

使用信号量和 P、V 操作实现同步时, 对共享资源的管理分散在各个进程之中, 进程能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, 而且容易造成程序设计错误。如果能把有关共享变量的操作集中在一起统一控制和管理, 就可方便对共享资源的使用, 使并发进程之间的相互作用更为清晰, 更容易编写出正确的并发程序。

在 1974 年和 1975 年, 汉森 (Brinch Hansen) 和霍尔 (Hoare) 提出了新的同步机制——管程

(monitor)。其基本的思路是：把分散在各进程中的临界区集中起来进行管理，并把系统中的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”程序管理来到的访问。“秘书”每次仅让一个进程来访，这样既便于对共享资源的管理，又实现了互斥访问。在后来的实现中，“秘书”程序改名为管程。采用这种方法，对共享资源的管理就可借助数据结构及在其上实施操作的若干过程来进行；对共享资源的申请和释放通过过程在数据结构上的操作来实现。而代表共享资源的数据结构及在其上操作的一组过程就构成了管程，管程被请求和释放资源的进程所调用。管程是一种程序设计语言结构成分，它和信号量有同等的表达能力。

管程有以下属性，因而，调用管程的过程时要有一定限制：

- 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。
- 安全性：管程的局部变量只能由该管程的过程存取，不允许进程或其他管程来直接存取，一个管程的过程也不应该存取任何非局部于它的变量。
- 互斥性：在任一时刻，共享资源的进程可访问管程的管理该资源的过程，但最多只有一个调用者能真正地进入管程，而任何其他调用者必须等待，直到访问者退出。

由上面的讨论可以看出：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；管程提供了一种互斥机制，进程可以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了起来，提供对它们的保护。由于管程中的共享变量每次只能被一个进程访问，当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，并为访问这些共享资源提供一种互斥机制。管程可以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书写程序，也便于程序正确性验证。管程的概念和机制已在 Pascal、Modula-2 和 Java 等语言中被实现。

每一个管程都要有一个名字以供标识，如果用语言来写一个管程，它的形式如下：

```
TYPE <管程名> = MONITOR
  <管程变量说明>;
  define <(能被其他模块引用的)过程名列表>;
  use <(要引用的模块外定义的)过程名列表>;
  procedure <过程名>(<形式参数表>);
    begin
      <过程体>;
    end;
    .....
  procedure <过程名>(<形式参数表>);
```

```

begin
<过程体>;
end;
begin
<管程的局部数据初始化语句>;
end.

```

注意，在正常情况下，管程的过程体可以有局部数据。管程中的过程可以有两种，由 define 定义的过程可以被进程或其他管程模块引用，而未定义的则仅在管程内部使用。管程要引用模块外定义的过程，必须用 use 说明。

管程的结构可以如图 3-5 所示，下面先举一个例子来说明管程的结构。系统中有一个资源可以为若干个进程所共享，但每次只能由一个进程使用，用管程作同步机制，对该资源的管理可如下实现：

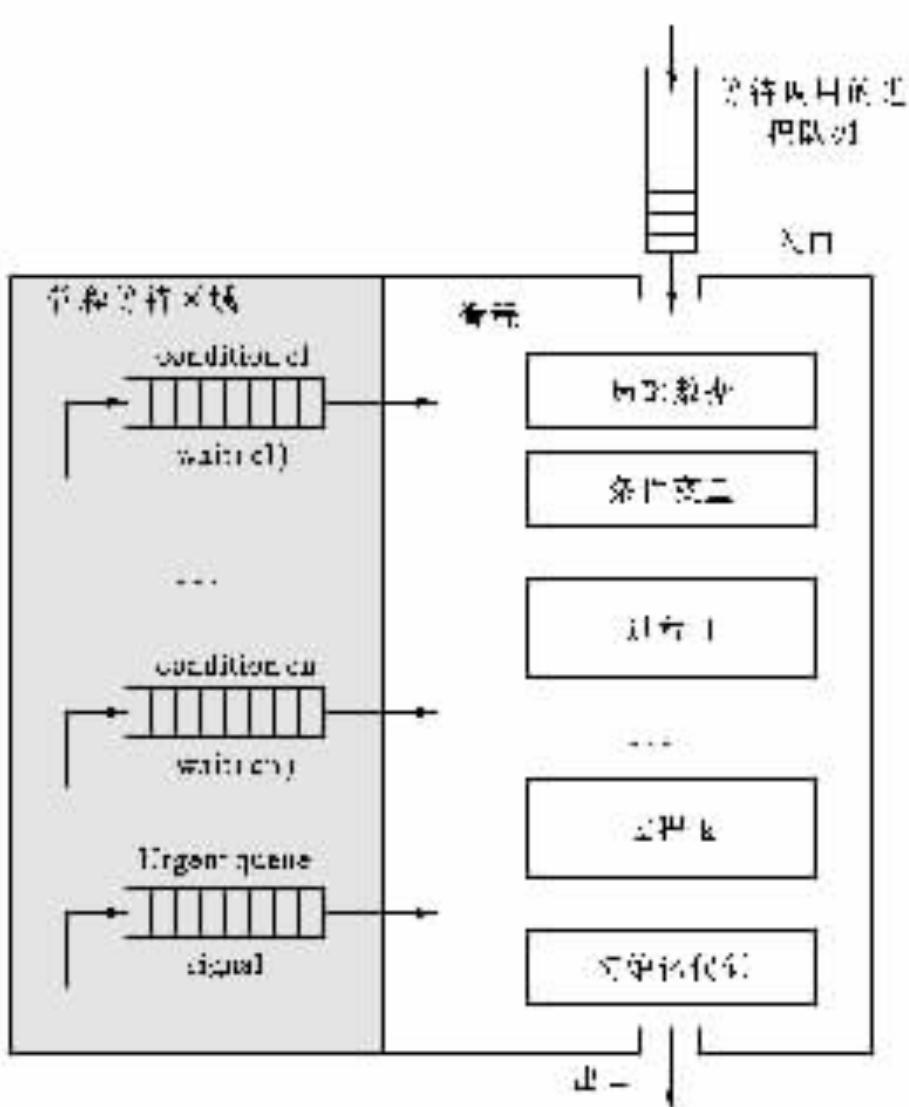


图 3-5 管程的结构示意图

```

TYPE SSU = MONITOR
var busy : boolean;
nobusy : condition;
define require, return;
use wait, signal;
procedure require;
begin
  if busy then wait(nobusy);           调用进程加入等待队列}
  busy := ture;
end;
procedure return;
begin
  busy := false;                      从等待队列中释放进程}
  signal(nobusy);
end;
begin                                     管程变量初始化}
  busy := false;
end;

```

这是一个名叫 SSU 的管程, 它有管理申请和释放资源的两个过程, 这两个过程可以在模块外引用, 在管程中用 define 子句声明; 这两个过程又要用到模块外定义的操作 wait 和 signal, 在管程中用 use 子句声明。

尽管如前所述, 对管程过程的调用是互斥的, 从而, 提供了一种实现互斥的简单途径, 但是这还不够, 需要一种办法使得进程在资源不能满足而无法继续运行时被阻塞。解决的方法在于引进另外一种称作条件变量 (condition variables) 的同步机制, 以及在其上操作的仅有的两个同步原语 wait 和 signal。当一个管程过程发现无法继续执行下去时, 例如, 发现没有可用资源时, 它在某些条件变量 condition 上执行 wait, 这个操作引起调用进程阻塞, 当然, 这时允许先前被挡在管程之外的一个进程进入管程。另一个进程可以通过对其伙伴在等待的同一个条件变量 condition 上执行 signal 操作来唤醒等待进程。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 操作中所论述纯粹的计数信号量, 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条件变量的进程时, signal 操作发出的信号丢失, 等于做了一次空操作。wait 操作一般应在 signal 操作之前发出, 这一规则大大简化了实现。

管程 SSU 自身定义了一个局部变量 busy, 它是一个标志, 用来表示共享资源的忙闲状态, 再定义一个条件变量 nobusy。当有进程要使用资源时调用 require 过程, require 过程判

busy 的状态,若有进程在使用资源时,则 busy 为 true,这时调用者执行条件变量 nobusy 上的 wait 操作并进入等待队列,同时开放管程,让其他进程进入工作;若无进程在使用资源,则 busy 为 false,这时调用者可以去使用资源,同时将 busy 置成 true。当进程归还资源时调用 return 过程,该过程把 busy 恢复成 false。此时,若有进程等待使用资源,则它将被条件变量 nobusy 上的一个 signal 操作所释放且立即可得到资源,同时 busy 再次变成 true;若没有进程在等待使用资源,那么,busy 保持 false 状态。

wait 和 signal 是两条原语,在执行时不允许被中断。它们分别表示把某个进程排入等待使用资源的条件变量的等待队列和从等待资源的条件变量上释放出来。当执行 wait 之后,相应的进程被置成等待状态,同时开放管程,允许其他进程调用管程中过程。当执行 signal 之后,指定条件变量上的一个进程被释放。

从上面的例子中可看到,管程有时要延迟一个不能得到共享资源的进程的执行,当别的进程释放了它所需要的资源时,再恢复该进程的执行。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恢复执行。现在的问题是当某进程执行了 signal 操作后,另一个被延迟的进程可恢复执行,于是就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在执行,根据管程的互斥性,这是绝对不允许的。可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,而条件变量队列中有一个进程 Q 在等待,当进程 P 执行了 signal 操作后,进程 Q 被释放,对它们可作如下处理:

- 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。
- 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。

霍尔采用了第一种办法,而汉森选择了两者的折衷,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 马上被恢复执行。

这里进一步把管程与进程作一个比较,它们有以下的区别:

- 管程定义的是公用数据结构,而进程定义的是私有数据结构;
- 管程把共享变量上的同步操作集中起来,而临界区却分散在每个进程中;
- 管程是为管理共享资源而建立的,进程主要是为占有系统资源和实现系统并发性而引入的;
 - 管程是被欲使用共享资源的进程所调用的,管程和调用它的进程不能并行工作,而进程之间能并行工作,并发性是其固有特性;
 - 管程是语言或操作系统的成分,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。

下面将分别介绍霍尔和汉森的管程实现方法。

3.4.2 Hoare 方法实现管程

霍尔方法是一种更为一般的管程实现方法, 它使用 P 操作原语和 V 操作原语来实现对管程中过程的互斥调用功能, 以及实现对共享资源互斥使用的管理。每当有进程等待资源时, 霍尔方法将让执行 signal 操作的进程挂起自己, 直到被它释放的进程退出管程或产生了其他的等待条件为止。与汉森的实现相比这种方法不要求 signal 操作是过程体的最后一个操作, 而且 wait 和 signal 操作可被设计成两个可以中断的过程。

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(其初值为 1)。任何一个进程调用管程中的任何一个过程时, 应执行 P(mutex); 一个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。为了使一个进程在等待资源期间, 其他进程能进入管程, 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。

对于每个管程, 还必须引入另一个信号量 next(其初值为 0), 凡发出 signal 操作的进程应该用 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, 都必须检查是否有别的进程在信号量 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, 提供一个 next - count(初值为 0), 用来记录在 next 上等待的进程个数。

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(其初值为 0), 申请资源得不到满足时, 执行 P(x - sem) 挂起自己。由于释放资源时, 需要知道是否有别的进程正在等待资源, 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。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。

于是每个管程都应该定义一个如下的数据结构:

```
TYPE interf = RECORD
    mutex: semaphore;           {进程调用管程过程前使用的互斥信号量}
    next: semaphore;           {发出 signal 的进程挂起自己的信号量}
    next_count: integer;        {在 next 上等待的进程数}
END;
```

现在来写 wait 操作和 signal 操作的两个过程:

```
procedure wait(var x_sem: semaphore, x_count: integer, IM: interf);
begin
    x_count := x_count + 1;
```

```

if IM.next_count > 0 then V(IM.next); else V(IM.mutex);
P(x_sem);
X_count := X_count - 1;
end;
procedure signal(var x_sem:semaphore,x_count:integer, IM:interf);
begin
if x_count > 0 then begin
IM.next_count := IM.next_count + 1;
V(x_sem);
P(IM.next);
IM.next_count := IM.next_count - 1;
end;
end;

```

任何一个调用管程中过程的外部过程都应该组织成下列形式,以确保互斥地进入管程。

```

P(IM.mutex);
<过程体>;
if IM.next_count > 0 then V(IM.next);
else V(IM.mutex);

```

下面的例子将说明如何用霍尔方法实现进程的同步。

例子:五个哲学家吃通心面问题。有五个哲学家围坐在一圆桌旁,桌子中央有一盘通心面,每人面前有一只空盘子,每两人之间放一把叉子。每个哲学家思考、饥饿、然后,欲吃通心面。为了吃通心面,每个哲学家必须获得两把叉子,且每人只能直接从自己左边或右边去取叉子。

首先,引入枚举类型来表示哲学家状态的变量:

```
var state:array[0..4] of (thinking, hungry, eating);
```

哲学家*i*能建立状态 $state[i] = eating$,仅当他的两个邻座不在吃的时候,即 $state[(i-1) \bmod 5] \neq eating$,以及 $state[(i+1) \bmod 5] \neq eating$ 。另外还要引入信号量:

```
var self:array[0..4] of semaphore;
```

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。特别要注意:信号量 $self[]$ 实质上是条件变量,但由于 Hoare 方法是采用信号量及 P、V 实现管程的,所以,这时的 condition 就用 semaphore 代替。于是有:

```

TYPE dining - philosophers = MONITOR
var state : array[0..4] of (thinking, hungry, eating);

```

```

    self : array[0..4] of semaphore;
    s - count : array[0..4] of integer;
define pickup, putdown;
use wait, signal;
procedure test(k : 0..4);
begin
  if state[(k - 1) mod 5] < > eating and state[k] = hungry
    and state[(k + 1) mod 5] < > eating then begin
      state[k] := eating;
      signal(self[k], s - count[k], IM);
    end;
end;
procedure pickup(i:0..4);
begin
  state[i] := hungry;
  test(i);
  if state[i] < > eating then wait(self[i], s - count[i], IM);
end;
procedure putdown(i:0..4);
begin
  state[i] := thinking;
  test((i - 1) mod 5);
  test((i + 1) mod 5);
end;
begin
  for i := 0 to 4 do state[i] := thinking;
end;

```

任一个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。即：

```

cobegin
  process philosopher - i
begin
  .....
  P(IM.mutex);
  call dining - philosopher.pickup(i);
  if IM.next - count > 0 then V(IM.next);

```

```

        else V(IM.mutex);

吃通心面;

.....
P(IM.mutex);
call dining-philosopher.putdown(i);
if IM.next-count > 0 then V(IM.next);
else V(IM.mutex);

.....
end;
coend.

```

3.4.3 Hanson 方法实现管程

汉森方法是一种折中的管程实现方法, 规定管程中的过程所执行的 signal 操作一定是过程体的最后一个操作。于是, 一个进程当所调用的过程执行了 signal 操作后, 便立即退出了管程。汉森方法要求使用四条原语: wait, signal, check, release 来实现对管程的控制。

- 等待原语 wait: 执行这条原语后相应进程被置成等待状态, 同时开放管程, 允许其他进程调用管程中的过程。
- 释放原语 signal: 执行这条原语后指定等待队列中的一个进程被释放, 如果指定等待队列中没有进程, 则它相当于空操作。

只有这两条原语是不够的, 为了实现管程的互斥调用功能, 还应有另外两条原语:

- 调用查看原语 check: 如果管程是开放的, 则执行这条原语后关闭管程, 相应进程继续执行下去; 如果管程是关闭的, 则执行这条原语后相应进程被置成等待调用状态。
- 开放原语 release: 如果除了发出这条原语的进程外, 不再有调用了管程中的过程但又不处于等待状态的进程, 那么就释放一个等待调用者(有等待调用者时), 或开放管程(无等待调用者时)。执行这条原语后就认为相应的进程结束了一次过程调用。

不难看出, 在管程的每个过程的头尾分别增加 check 和 release 原语后, 互斥调用功能就可实现。因为, 进程调用管程中的过程时, 过程执行的第一个语句是 check, 它的执行保证了互斥调用; 当管程中的过程执行结束时, 最后一个语句是 release, 它的执行保证了管程的开放。

假定对每个管程都定义了一个如下类型的数据结构:

| | |
|-------------------------------------------------------------------------------|-------------------------------|
| <pre> TYPE interf = RECORD intsem: condition; count1: integer; </pre> | {开放和关闭管程的条件变量} {等待调用的进程个数} |
|-------------------------------------------------------------------------------|-------------------------------|

```

count2 : integer;           {调用了管程中的过程且不
END;                      处于等待状态的进程个数 }

```

其中, intsem 是开放管程的条件变量, count1 是等待调用的进程个数, count2 是调用了管程中过程且不处于等待状态的进程个数。那么, 上述四条原语的功能可用如下四个过程描述:

```

procedure wait(var s:condition; var IM interf);
begin
  s := s + 1;
  IM.count2 := IM.count2 - 1;
  if IM.count1 > 0 then
    begin
      IM.count1 := IM.count1 - 1;
      IM.count2 := IM.count2 + 1;
      R(IM.intsem);
    end;
  W(s);
end;

procedure signal(var s:condition; var IM interf);
begin
  if s > 0 then
    begin
      s := s - 1;
      IM.count2 := IM.count2 + 1;
      R(s);
    end;
end;

procedure check(var IM interf);
begin
  if IM.count2 = 0
  then IM.count2 := IM.count2 + 1;
  else
    begin
      IM.count1 := IM.count1 + 1;
      W(IM.intsem);
    end;
end;

```

```

end;

procedure release(var IM interf);
begin
  IM.count2 := IM.count2 - 1;
  if IM.count2 = 0 and IM.count1 > 0 then
    begin
      IM.count1 := IM.count1 - 1;
      IM.count2 := IM.count2 + 1;
      R(IM.intsem);
    end;
end;

```

signal 所能释放的一定是同一管程中的某个 wait 原语的执行者。参数 s 表示等待管程中某个条件变量(资源)的进程个数,它的初值为零。

$W(s)$ 表示将调用过程的进程置成等待条件变量 s 的状态; $R(s)$ 表示释放一个等待条件变量 s 的进程。同样, $W(IM.intsem)$ 和 $R(IM.intsem)$ 分别表示把调用者置成等待调用管程的状态和释放一个等待调用管程的进程。

用管程实现进程同步时,每个进程应按下列次序工作:

- 请求资源。
- 使用资源。
- 释放资源。

其中,请求资源是调用管程中的一个管理资源分配的过程;释放资源是调用管程中的一个管理回收资源的过程。

现在用例子来说明如何用 Hanson 方法实现进程同步。

例 1:读者与写者问题。

这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,共享一个文件 F,要求:(1)允许多个读者同时执行读操作,(2)任一写者在完成写操作之前不允许其他读者或写者工作;(3)写者欲工作,但在他之前已有读者在执行读操作,那么,待现有读者完成读操作后才能执行写操作,新的读者和写者均被拒绝。

用两个计数器 rc 和 wc 分别对读进程和写进程计数,用 R 和 W 分别表示允许读和允许写的条件变量,于是管理该文件的管程可如下设计:

```

type read-writer = MONITOR
var rc, wc : integer;
R, W : condition;

```

```
define start - read, end - read, start - writer, end - writer;
use wait, signal, check, release;
procedure start - read;
begin
  check(IM);
  if wc > 0 then wait(R, IM);
  rc := rc + 1;
  signal(R, IM);
  release(IM);
end;
procedure end - read;
begin
  check(IM);
  rc := rc - 1;
  if rc = 0 then signal(W, IM);
  release(IM);
end;
procedure start - write;
begin
  check(IM);
  wc := wc + 1;
  if rc > 0 or wc > 1 then wait(W, IM);
  release(IM);
end;
procedure end - write;
begin
  check(IM);
  wc := wc - 1;
  if wc > 0 then signal(W, IM);
  else signal(R, IM);
  release(IM);
end;
begin
  rc := 0; wc := 0; R := 0; W := 0;
end.
```

任何一个进程读(写)文件前,首先调用 start - read (start - write),执行完读(写)操作后,

调用 end-read(end-write)。即：

```
cobegin
    process reader
begin
    .....
    call read - writer.start - read;
    .....
    read;
    .....
    call read - writer.end - read;
    .....
end;

process writer
begin
    .....
    call read - writer.start - write;
    .....
    write;
    .....
    call read - writer.end - write;
    .....
end;
coend.
```

上述程序能保证在各种并发执行的情况下，读写进程都能正确工作，请读者自行验证。

例 2：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)，妈妈专向盘子中放桔子(orange)，一个儿子专等吃盘子中的桔子，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。

这个问题实际上是两个生产者和两个消费者被连接到仅能放一个产品的缓冲器上，生产者各自生产不同的产品，消费者各自取需要的产品消费。用一个包括两个过程：put 和 get 的管程来实现同步：

```
type FMSD = MONITOR
var plate : (apple, orange);
    full : boolean;
    SP, SS, SD : condition;
```

```

define put, get;
use wait, signal, check, release;
procedure put(var fruit:(apple, orange));
begin
  check(IM);
  if full then wait(SP, IM);
  full := true;
  plate := fruit;
  if fruit = orange
    then signal(SS, IM);
  else signal(SD, IM);
  release(IM);
end;
procedure get(var fruit:(apple, orange), x:plate);
begin
  check(IM);
  if not full or plate < > fruit
  then begin
    if fruit = orange
      then wait(SS, IM);
    else wait(SD, IM);
  end;
  x := plate;
  full := false;
  signal(SP, IM);
  release(IM);
end;
begin
  full := false; SP := 0; SS := 0; SD := 0;
end;

```

爸爸妈妈向盘中放水果时调用过程 put, 儿子女儿取水果时调用过程 get。即：

```

cobegin
  process father
begin
  .....

```

```

准备好苹果;
call FMSD.put(apple);
.....
end;
process mother
begin
.....
准备好桔子;
call FMSD.put(orange);
.....
end;
process son
begin
.....
call FMSD.get(orange,x);
吃取到的桔子;
.....
end;
process daughter
begin
.....
call FMSD.get(apple,x);
吃取到的苹果;
.....
end;
coend;

```

例 3:用 monitor 解决生产者和消费者问题。

```

type producer-consumer = MONITOR
var B:array[0..k-1] of item;           缓冲区个数}
in,out:integer;                      存取指针}
count:integer;                        缓冲中产品数}
notfull,notempty:condition;          条件变量}

define append,take;

use wait, signal, check, release;

procedure append(x:item);

```

```

begin
  check(IM);
  if count = k then wait(notfull, IM);      {缓冲已满}
  B[in] := x;
  in := (in + 1) mod k;
  count := count + 1;                      {增加一个产品}
  signal(notempty, IM);                   {唤醒等待者}
  release(IM);
end;

procedure take(x:item);
begin
  check(IM);
  if count = 0 then wait(notempty, IM);      {缓冲已空}
  x := B[out];
  out := (out + 1) mod k;
  count := count - 1;                      {减少一个产品}
  signal(notfull, IM);                   {唤醒等待者}
  release(IM);
end;

begin                                         {初始化}
  in := 0; out := 0; count := 0;
end;

cobegin                                     {主程序}
process producer;
  var x:item;
begin
  produce(x);
  append(x);
end;

process consumer;
  var x:item;
begin
  take(x);
  consume(x);
end;
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。进程竞争资源时要实施互斥，互斥是一种特殊的同步，实质上需要解决好进程同步问题，进程同步是一种进程通信，通过修改信号量，进程之间可建立起联系，相互协调运行和协同工作。但是信号量与 PV 操作只能传递信号，没有传递数据的能力。虽然有些情况下进程之间交换的信息量很少（例如，仅仅交换某个状态信息），但很多情况下进程之间需要交换大批数据（例如，传送一批信息或整个文件），这可以通过一种新的通信机制来完成，进程之间互相交换信息的工作称之为通信 IPC（InterProcess Communication）。进程间通信的方式很多，包括：

- 信号（signal）通信机制；
- 信号量及其原语操作（PV、读写锁、管程）控制的共享存储区（shared memory）通信机制。
- 管道（pipeline）提供的共享文件（shared file）通信机制；
- 信箱和发信/收信原语的消息传递（message passing）通信机制。

其中前两种通信方式由于交换的信息量少且效率低下，故称为低级通信机制，相应地可以把发信号/收信号及 PV 之类操作称为低级通信原语，仅适用于集中式操作系统。消息传递机制属于高级通信机制，共享文件通信机制是消息传递机制的变种，这两种通信机制，既适用于集中式操作系统，又适用于分布式操作系统。

3.5.1 信号通信机制

信号（signal）机制又称软中断，是一种进程之间进行通信的简单通信机制，通过发送一个指定信号来通知进程某个异常事件发生，并进行适当处理。进程运行时不时地检查有无软中断信号到达，如果有，则中断原来正在执行的程序，转向该信号的处理程序对该事件进行处理，处理结束后便可返回原程序的断点执行。一般地可以分成 OS 标准信号和用户进程自定义信号，这种机制类似硬件中断，不分优先级，简单有效，但不能传送数据。软中断与硬中断的差别在于：软中断运行在用户态，往往延时较长，而硬中断运行在核心态，由于是硬件实现，故都能及时响应。举例来说，当系统正在运行一个耗时的前台程序，如若已发现有错误，并断定该程序要失败，为了节省时间，用户可以按软中断键（一般为 del + ctrl + c）停止程序的执行，这一过程中就用到了信号（signal）。系统具体的操作为：响应键盘输入的中断处理程序向发来中断信号的终端进程发一个信号，进程收到信号后，完成相关处理，然后执行终止。

信号不但能从内核发给一个进程，也能由一个进程发给同组的另一个进程或多个进程，

一个信号的发送是指把它送到指定进程的 PCB 中的软中断域的某一位,由于每个信号都被看作一个单位,给定类型的信号不能排队,只是在进程被唤醒继续运行时,或者在进程准备从系统调用请求返回时,才处理信号。进程可以执行默认操作、执行一个处理函数或忽略该信号,来对信号做出响应。像 UNIX 系统信号多达几十种(不超过 32 种),主要分成以下几类:

- 与进程终止相关的信号 SIGCLD、SIGHUP、SIGKILL、SIGCHLD、SIGSTOP 等,如进程结束、父进程杀死子进程;
- 与进程例外事件相关的信号 SIGBUS、SIGSEGV、SIGPWR、SIGFPE 等,如进程执行特权指令、写只读区、地址越界、总线超时、硬件故障;
- 与进程执行系统调用相关的信号 SIGPIPE、SIGSYS、SIGILL 等,如进程执行非法系统调用、管道存取错;
- 与进程终端交互相相关的信号 SIGINT、SIGQUIT 等,如进程挂断终端、用户按 delete 键或 break 键。
- 用户进程发信号 SIGTERM、SIGALRM、SIGUSR1、SIGUSR2 等,如进程向另一进程发一个信号、要求报警;
- 跟踪进程执行的信号 SIGTRAP 等。

关于软中断机制需要具有以下功能:软中断信号的发送,软中断信号的处理和软中断信号的预置。下面以 UNIX 为例简单回答这些问题:UNIX 的软中断信号通信机制涉及的数据结构在 proc 和 user 结构中,①p - clktim 是报警时钟计数器,由系统调用 alarm 放置,时间到就发出 SIGALARM 报警信号。P - sigign 是信号忽略标记,共 32 位,当进程要忽略信号时就把对应位置 1。P - sig 进程接收信号位,共 32 位,进程收到信号时,对应位置 1。U - signal [NSIG] 是 32 个元素的数组,每个元素存放一个软中断处理程序的入口地址,让用户进程提供特殊处理。②信号发送工作由系统调用 kill(pid, sig) 完成, pid 确定了信号 sig 发往的进程,sig 是软中断信号类型,由于这种信号要知道发往进程的标识号,所以,信号发送通常在关系密切进程之间进行。信号发送需有一定权力,超级用户进程可把信号发送给任何进程,但普通用户进程向非同组用户发送信号,那么, kill 调用会失败。③信号的响应方式使用系统调用 signal(sig, func), sig 指出软中断信号类型;function 描述与信号关联的操作,随其值的不同,可以分成三种处理方式,分别为: function = 1 时,忽略 sig(不予理采); function = 0 时,进行标准处理(终止进程);当 function 为其他值时,转向用户指定的软中断信号处理程序(function 的值为入口指针)处理。如果 signal 调用成功,则将转入关联函数执行,否则,返回错误值供程序处理。④对信号的处理过程如下:当一个进程处于核心态时,即使收到软中断信号也不理睬,仅当它要进入等待态或退出一个低优先级等待队列时,或一个进程从核心态返回用户态时,系统都会检查该进程是否已收到软中断,是则让它返回用户态并处理收到的软中断。

3.5.2 共享文件通信机制

管道(pipeline)是连接读写进程的一个特殊文件,允许进程按先进先出方式传送数据,也能使进程同步执行操作。如图 3-6 所示,发送进程视管道文件为输出文件,以字符流形式把大量数据送入管道;接收进程将管道文件视为输入文件,从管道中接收数据,所以,也叫管道通信。由于方便有效,能在进程间作大量信息的通信,目前已被引入到许多操作系统中。管道和消息队列的主要区别在于:管道中的消息是无界的,它存于外存;消息队列是位于内存的。

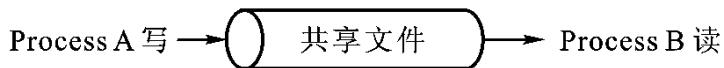


图 3-6 pipe 通信机制

管道(pipe)是 UNIX 和 C 语言的传统通信方式,也是 UNIX 发展最有意义的贡献之一。管道通信方式是发送者进程和接收者进程之间通过一个管道交流信息,管道是单向的,发送者进程只能写入(字节流)信息,接收者进程也只能接收(字节流)信息,先写入的必定先读出。管道的实质是一个共享文件,亦即利用辅存来进行数据通信,因此,管道通信基本上可以借助于文件系统原有的机制实现,包括(管道)文件的创建、打开、关闭和读写。但是,写入进程和读出进程之间的相互协调单靠文件系统机制是解决不了的。读写进程相互协调,必须做到以下几点:

- 一个进程正在使用某个管道写入或读出数据时,另一个进程就必须等待。这一点是进程在读写管道之前,通过测试文件 i-node 节点的特征位来保证的,该特征位实质是一个读写互斥标志;若已锁住,进程便等待,否则,把 i-node 上锁,然后进行读或写操作,操作结束后再行解锁并唤醒因节点上锁而等待的进程。
- 发送者和接收者双方必须能够知道对方是否存在,如果对方已经不存在,就没有必要再发送或接收信息。这时会发出 SIGPIPE 信号通知进程。
- 发送信息和接收信息之间一定要实现正确的同步关系,这是由于管道文件只使用 i-node 节点中的直接地址项,故长度限于几 K 字节,管道的长度限制对进程 write 和 read 操作会有影响。如果进程执行一次写操作,且管道有足够的空间,那么,write 把数据写入管道后唤醒因该管道空而等待的进程;如果这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出数据,使管道有空间为止,这叫 write 阻塞。解决此问题的办法是:把数据进行切分,每次最多小于管道限制的字节数,写完后该进程睡眠,直到读进程把管道中的数据取走,并判断有进程等待时应唤醒对方,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。

- 进程在关闭管道的读出或写入端时,应唤醒等待写或读此管道的进程。

在 UNIX 中,管道的定义如下:

```
int pipe(files);
int files[2];
```

建立管道的主要工作是在系统打开文件表中建立该 pipe 的两个表目,分别控制管道的写操作和读操作,也就是定义了该 pipe 的写入端和读出端。两个表目指向同一个 i-node,此节点对应了外存上的 pipe 文件。核心在执行完 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] 从管道中取走信息,发送者进程则通过 files[1] 向管道写入数据,pipe 的数据结构可用图 3-7 表示。下面是父子进程通过管道传送信息的一个例子:

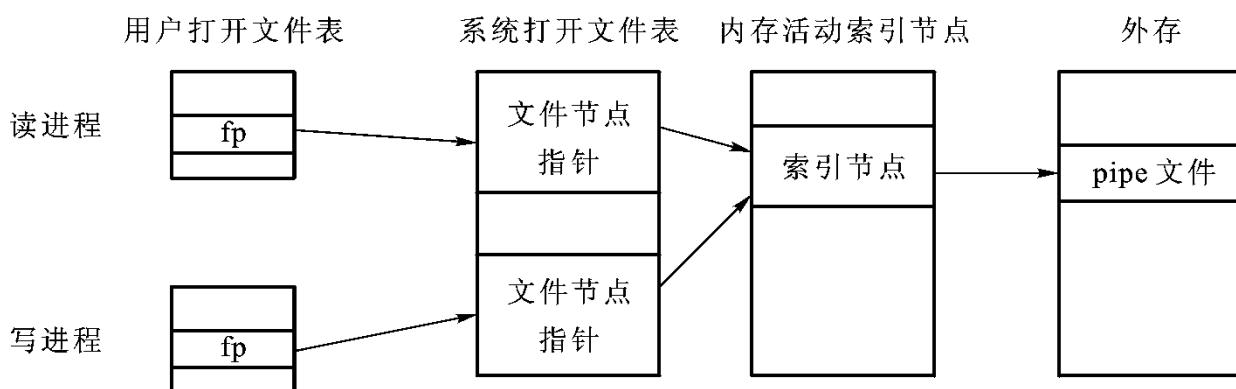


图 3-7 pipe 的数据结构

```
# include < stdio.h >
#define MSGSIZE 16
char * msg1 = "hello,world # 1";
char * msg2 = "hello,world # 2";
char * msg3 = "hello,world # 3";
main()
{
    char inbuf[MSGSIZE];
    int p[2], j, pid;
    /* open pipe */
    if(pipe(<p>) < 0)
    {
        perror("pipe call");
        exit(1);
    }
}
```

```

    }

    if((pid=fork()) < 0)
    {
        perror("fork call");
        exit(2);
    }

{if parent, then close read file descriptor and write down pipe}
    if(pid>0
    {
        close(p[0]);
        write(p[1],msg1,MSGSIZE);
        write(p[1],msg2,MSGSIZE);
        write(p[1],msg3,MSGSIZE);
        wait((int *)0);
    }

{if child ,then close write file descriptor and read from pipe}
    if (pid==0) {
        close(p[1]);
        for (j = 0;j < 3;j++)
            read(p[0],inbuf,MSGSIZE);
        printf("%s\n",inbuf);
    }

}
exit(0);
}

```

管道是一种功能很强的通信机制,但它仅能用于连接具有共同祖先的进程,管道也不是常设的,需临时建立,难于提供全局服务。为了克服这些缺点,UNIX 中又推出了管道的一个变种,称为有名管道或 FIFO 通信机制。这是一种永久性通信机制,具有 UNIX 文件名、访问权限,能像一般文件一样被打开、关闭、删除,但 write 和 read 时,其性能与管道相同。可以通过系统调用 mknod(pipename,S_FIFO + rw,0) 而不是用 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(pipename,O_RDONLY)

来打开管道以取走信息;发送者进程则通过系统调用:

open(pipename,O_WRONLY)

来打开管道以写入数据;对管道的读写通过系统调用 read 和 write 来实现。

另外,在操作控制命令层也可以使用管道,例如:

```
who | sort | more
```

管道是基于信箱的消息传递方式的一种变种,它们与传统的信箱方式等价,区别在于没有预先设定消息的边界。换言之,如果一个进程发送 10 条 100 字节的消息,而另一个进程接收 1000 个字节,那么,接收者将一次获得 10 条消息。

随着应用的扩展,管道机制不仅仅适用于集中式系统中的进程一对通信,在 AT&T UNIX 的新版本中管道机制也适用于网络上多进程之间的通信。

3.5.3 共享存储区通信机制

内存中开辟一个共享存储区,如图 3-8 所示,诸进程通过该存储区实现通信,这是进程通信中最快捷和有效的方法。进程通信之前,向共享存储区申请一个分区段,并指定关键字;若系统已为其他进程分配了这个分区,则返回关键字给申请者,于是该分区段就可连到进程的虚地址空间,以后,进程便像通常存储器一样共享存储区段,通过对该区段的读、写来直接进行通信。

UNIX/Linux 与共享存储有关的系统调用有四个:

- `shmget(key, size, permflags)`: 用于建立共享存储区,或返回一个已存在的共享存储区,相应信息登入共享存储区表中。`size` 给出共享存储区的最小字节数;`key` 是标识这个段的描述字;`permflags` 给出该存储区的权限。

- `shmat(shmid, daddr, shmflg)`: 用于把建立的共享存储区连入进程的逻辑地址空间。`shmid` 标识存储区,其值从 `shmget` 调用中得到;`daddr` 用户的逻辑地址;`permflags` 表示共享存储区可读可写或其他性质。

- `Shmdt(memptr)`: 用于把建立的共享存储区从进程的逻辑地址空间中分离出来。`memptr` 为被分离的存储区指针。

- `Shmctl(shmid, command, &shmstat)`: 实现共享存储区的控制操作。`shmid` 为共享存储区描述字;`command` 为规定操作;`&shmstat` 为用户数据结构的地址。

当执行 `shmget` 时,内核查找共享存储区中具有给定 `key` 的段,若已发现这样的段且许可权可接受,便返回共享存储区的 `key`;否则,在合法性检查后,分配一个存储区,在共享存储区表中填入各项参数,并设标志指示尚未存储空间与该区相连。执行 `shmat` 时,首先,查证进程对该共享段的存取权,然后,把进程合适的虚空间与共享存储区相连。执行 `shmdt` 时,其过程与 `shmat` 类似,但把共享存储区从进程的虚空间断开。

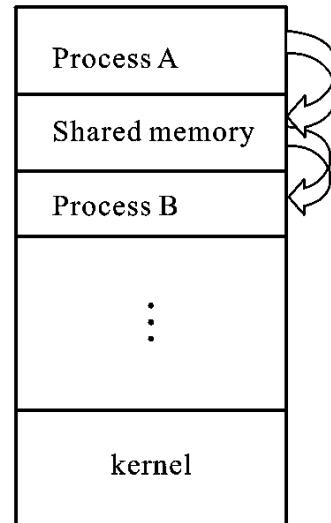


图 3-8 共享存储区通信机制

3.5.4 消息传递通信机制

1. 消息传递的概念

在前面几节讨论中已经看到, 系统中的交互进程通过信号量及有关操作可以实现进程的互斥和同步。生产者和消费者问题是一组相互协作的进程, 它们通过交换信号量和使用缓冲器达到产品递交的目的, 在用信号量解决生产者消费者问题时, 不是单靠信号量而是要另外引入有界缓冲来存放产品, 既不方便局限性也大, 这可以看作是一种低级的通信方式。有时进程间可能需要交换更多的信息, 例如, 一个输入输出操作请求, 要求把数据从一个进程传送给另一个进程, 这种大量的信息传递可使用一种高级通信方式——消息传递(message passing)来实现。由于操作系统提供的这类机制隐蔽了许多实现细节, 通过消息传递机制通信, 就能简化程序编制的复杂性, 方便易用, 得到了广泛应用。

消息是一组信息, 由消息头和消息体组成。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发送一个消息, 后者则从一个给定的源接受一条消息。如果没有消息可用, 则接收者可能阻塞直到一条消息到达, 或者也可以立即返回, 并带回一个错误码。

采用了消息传递机制后, 进程间用消息来交换信息。一个正在执行的进程可以在任何时刻向另一个正在执行的进程发送一个消息; 一个正在执行的进程也可以在任何时刻向正在执行的另一个进程请求一个消息。如果一个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发出消息的回答, 那么, 消息传递机制将紧密地与进程的阻塞和释放相联系。这样, 消息传递就进一步扩充了并发进程间对数据的共享, 提供了进程同步的能力。

2. 消息传递的方式

消息传递方式的变种很多, 常用的有直接通信(消息缓冲区)方式和间接通信(信箱)方式, UNIX 的 pipeline 和 socket 机制属于一种信箱方式的变种。图 3-9 是消息传递通信机制模型。

1) 直接通信方式

在直接通信方式下, 企图发送或接收消息的每个进程必须指出信件发给谁或从谁那里接收消息, 可用 send 原语和 receive 原语来实现进程之间的通信, 这两个原语定义如下:

- send(P, 消息): 把一个消息发送给进程 P。
- receive(Q, 消息): 从进程 Q 接收一个消息。

这样, 进程 P 和 Q 通过执行这两个操作而自动建立了一种联结, 并且这一种联结仅仅发生在这一对进程之间。消

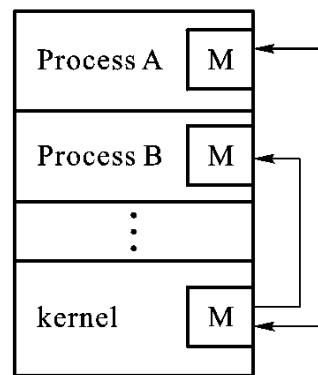


图 3-9 是消息传递通信机制

息可以有固定长度或可变长度两种。固定长度便于物理实现,但使程序设计增加困难;而消息长度可变使程序设计变得简单,但使消息传递机制的实现复杂化。

2) 间接通信方式

采用间接通信方式时,进程间发送或接收消息通过一个共享的数据结构——信箱来进行,消息可以被理解成信件,每个信箱有一个唯一的标识符。当两个以上的进程有一个共享的信箱时,它们就能进行通信。间接通信方式解除了发送进程和接收进程之间的直接联系,在消息的使用上灵活性较大。一个进程也可以分别与多个进程共享信箱,于是,一个进程可以同时和多个进程进行通信。一对一的关系允许在两个进程间建立不受干扰的专用通信链接;多对一的关系对客户机/服务器间的交互非常有用;一个进程给许多别的进程提供服务,这时的信箱又称作一个端口(port),端口通常归接收进程所有并由接收进程创建,当一个进程被撤销时,它的端口也随之被毁灭;一对多的关系适用于一个发送者和多个接收者,它对于在一组进程间广播一条消息的应用程序十分有用。间接通信方式中的“发送”和“接收”原语的形式如下:

- send(A, 信件):把一封信件(消息)传送到信箱 A。
- receive(A, 信件):从信箱 A 接收一封信件(消息)。

信箱是存放信件的存储区域,每个信箱可以分成信箱头和信箱体两部分。信箱头指出信箱容量、信件格式、存放信件位置的指针等;信箱体用来存放信件,信箱体分成若干个区,每个区可容纳一封信。

“发送”和“接收”两条原语的功能为:

- 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,并释放等待该信箱中信件的等待者;否则,发送信件者被置成等待信箱状态。
- 接收信件。如果指定信箱中有信,则取出一封信件,并释放等待信箱的等待者,否则,接收信件者被置成等待信箱中信件的状态。

两个原语的算法描述如下,其中,R()和W()是让进程入队和出队的两个过程。

```

type box = record
    size:integer;                      {信箱大小}
    count:integer;                     {现有信件数}
    letter:array[1..n] of message;     {信箱}
    S1,S2:semaphore;                  {等信箱和等信件信号量}
end

procedure send(varB:box,M:message)
var i:integer;
begin

```

```

if B.count = B.size then W(B.s1);
i := B.count + 1;
B.letter[i] := M;
B.count := i;
R(B.s2)
end;

procedure receive(varB:box, x:message)
var i:integer;
begin
  if B.count = 0 then W(B.s2);
  B.count := B.count - 1;
  x := B.letter[1];
  if B.count ≠ 0 then for i = 1 to B.count do B.letter[i] := B.letter[i + 1];
  R(B.s1);
end;

```

下面的程序是用消息传递机制解决生产者—消费者问题的一种解法。

```

var capacity:integer;           {缓冲大小}
i:integer;

procedure producer;
  var pmsg:message;
begin
  while true do
    begin
      pmsg := produce;          {生产消息}
      receive (mapproduce, pmsg); {等待空消息}
      build message;            {构造一条消息}
      send(mayconsume, pmsg);   {发送消息}
    end;
end;

procedure consumer;
  var cmsg:message;
begin

```

```

while true do
begin
    receive (mayconsume, cmsg);      接收消息}
    extract message;                取消息}
    send(mayproduce, null);         回送空消息}
    consume(csmg);                消耗消息}

end;
end;

begin                                主程序}
creat - mailbox(mayprocuce);        创建信箱}
creat - mailbox(mayconsume);
for i = 1 to capacity do send (mayproduce, null);   发送空消息}

cobegin
    producer;
    consumer;
coend

end.

```

这里对该程序作简单说明, 假设消息大小相同, 程序中共用了 capacity 条消息, 类似于共享内存缓冲的 capacity 槽。初始化时, 由主控程序负责发 capacity 条空消息给生产者(也可由消费者发出)。当生产者生产出一个数据后, 它接收一条空消息并回送一条填入数据的消息给消费者。通过这种方式, 系统中总的消息数保持不变。如果生产者速度比消费者快, 则所有空消息取尽, 于是生产者阻塞以等待消费者返回一条空消息。如果消费者速度快, 则正好相反, 所有的消息均为空, 等待生产者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了足够容纳 capacity 条消息的信箱, 消费者向生产者信箱发空消息, 生产者向消费者信箱发含数据的消息。当使用信箱时, 通信机制知道目标信箱中容纳那些已被发送到的但尚未被目标进程接收的消息。

3.5.5 有关消息传递实现的若干问题

这一节讨论消息传递实现中的几个问题。

首先, 是信箱容量问题。一个极限的情况是信箱容量为 0, 那么, 当 send 在 receive 之前执行的话, 则发送进程被阻塞, 直到 receive 做完。执行 receive 时信件可从发送者直接拷贝

到接收者,不用任何中间缓冲。类似的,如果 receive 先被执行,接受者将被阻塞直到 send 发生。上述策略称为回合(reendezvous)原则。这种方案实现较为容易,但却降低了灵活性,发送者和接收者一定要以步步紧接的方式运行。通常情况采用带有信件缓冲的方案,即信箱可放有限封信,这时一个进程可以连续做发送信件操作而无需等待直到信箱满,一个进程也可以连续做接收信件操作而无需等待直到信箱空。在这种方式下,系统具有迫使一个进程等信箱(当信箱满且还要发信)和等信件(当信箱空且还要接收信件)的功能。

其次,是关于多进程与信箱相连的信件接收问题。采用间接通信时,有时会出现如下问题,假设进程 P1, P2 和 P3 都共享信箱 A, P1 把一封信件送到了信箱 A, 而 P2 和 P3 都企图从信箱 A 取这个信件,那么,究竟应由谁来取 P1 发送的信件呢? 解决的办法有以下三种:

- 预先规定能取 P1 所发送的信件的接收者。
- 预先规定在一个时间至多一个进程执行一个接收操作。
- 由系统选择谁是接收者。

第三,关于信箱的所有权问题。一个信箱可以由一个进程所有,也可以由操作系统所有。如果一个信箱为一个进程所有,那么必须区分信箱的所有者和它的用户,区分信箱的所有者和它的用户的一个方法是允许进程说明信箱类型 mailbox,说明这个 mailbox 的进程就是信箱的所有者,其他任何知道这个 mailbox 名字的进程都可成为它的用户。当拥有信箱的进程执行结束时,它的信箱也就消失,这时必须把这一情况及时通知这个信箱的用户。信箱为操作系统所有是指由操作系统统一设置信箱,归系统所有,供相互通信的进程共享。消息缓冲机制就是一个著名的例子。

消息缓冲是在 1973 年由 P.B.Hansan 提出的一种进程间高级通信设施,并在 RC4000 系统中实现。消息缓冲通信的基本思想是:由操作系统统一管理一组用于通信的消息缓冲存储区,每一个消息缓冲存储区可存放一个消息(信件)。当一个进程要发送消息时,先在自己的消息发送区里生成待发送的消息,包括:接收进程名、消息长度、消息正文等。然后,向系统申请一个消息缓冲区,把消息从发送区复制到消息缓冲区中,注意在复制过程中系统会将接收进程名换成发送进程名,以便接收者识别。随后该消息缓冲区被挂到接收消息的进程的消息队列上,供接收者在需要时从消息队列中摘下并复制到消息接收区去使用,同时释放消息缓冲区。如图 3-10 所示:

消息缓冲通信涉及的数据结构有:

- sender:发送消息的进程名或标识符
- size:发送的消息长度
- text:发送的消息正文
- next - ptr:指向下一个消息缓冲区的指针

在进程的 PCB 中涉及通信的数据结构:

- mptr: 消息队列队首指针
 - mutex: 消息队列互斥信号量, 初值为 1
 - sm: 表示接收进程消息队列上消息的个数, 初值为 0, 是控制收发进程同步的信号量
- 发送原语和接收原语的实现如下:
- 发送原语 send: 申请一个消息缓冲区, 把发送区内容复制到这个缓冲区中; 找到接收进程的 PCB, 执行互斥操作 P(mutex); 把缓冲区挂到接收进程消息队列的尾部, 执行 V(sm)、即消息数加 1; 执行 V(mutex)。
 - 接收原语 receive: 执行 P(sm) 查看有否信件; 执行互斥操作 P(mutex), 从消息队列中摘下第一个消息, 执行 V(mutex); 把消息缓冲区内容复制到接收区, 释放消息缓冲区。

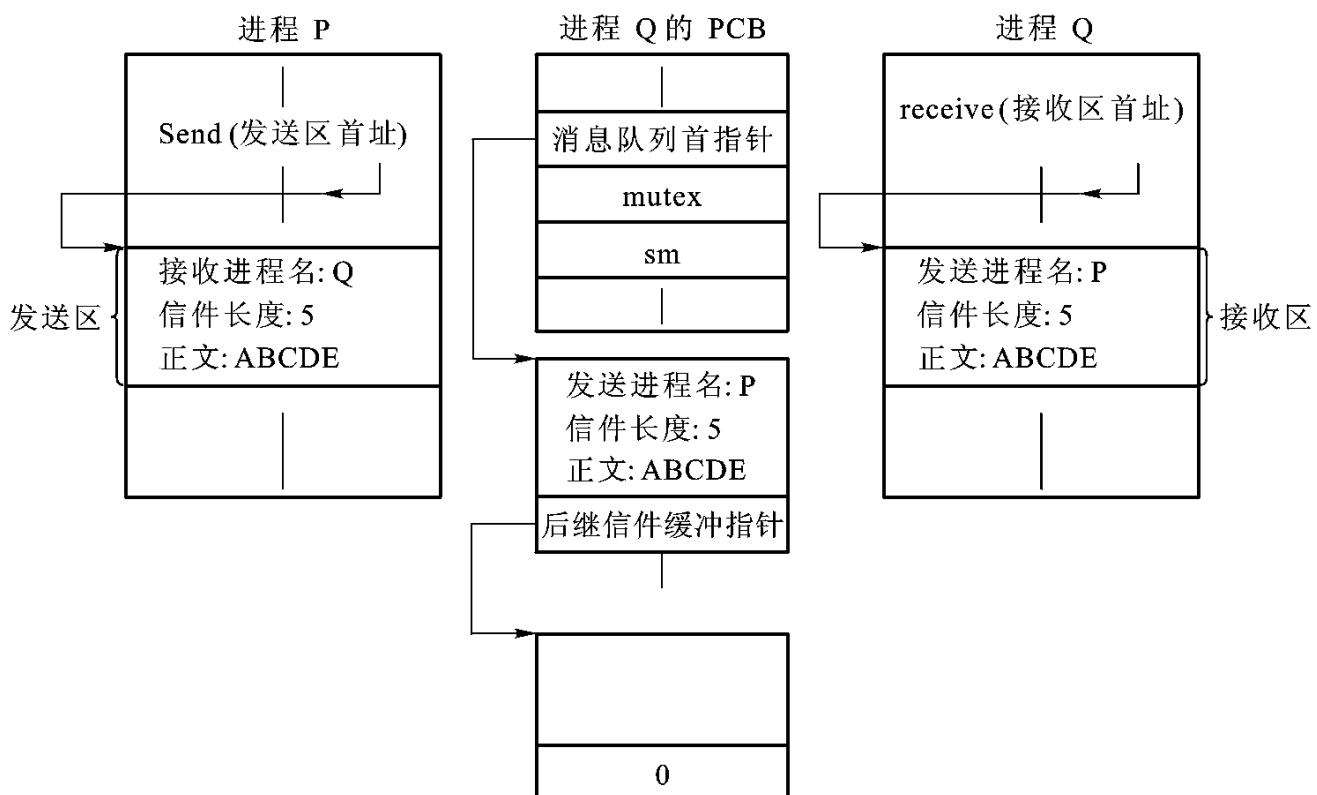


图 3-10 消息缓冲通信

消息队列本身是操作系统核心为通信双方进程建立的数据结构, 两个用户进程间通过发送和接收系统调用来借助消息队列传递和交换消息, 这样通信进程间不再需要共享变量。如图 3-11 所示, UNIX/Linux 的消息传递机制与信箱通信很类似, 进程间的通信通过消息队列进行。消息队列可以是单消息队列, 也可以是多消息队列(按消息类型);既可以单向, 也可以双向通信;既可以仅和两个进程有关, 也可以被多个进程使用。消息队列所用数据结构有:

- 消息缓冲池和消息缓冲区, 前者包含消息缓冲池大小和首地址; 后者除存放消息正文外, 还有消息类型字段。

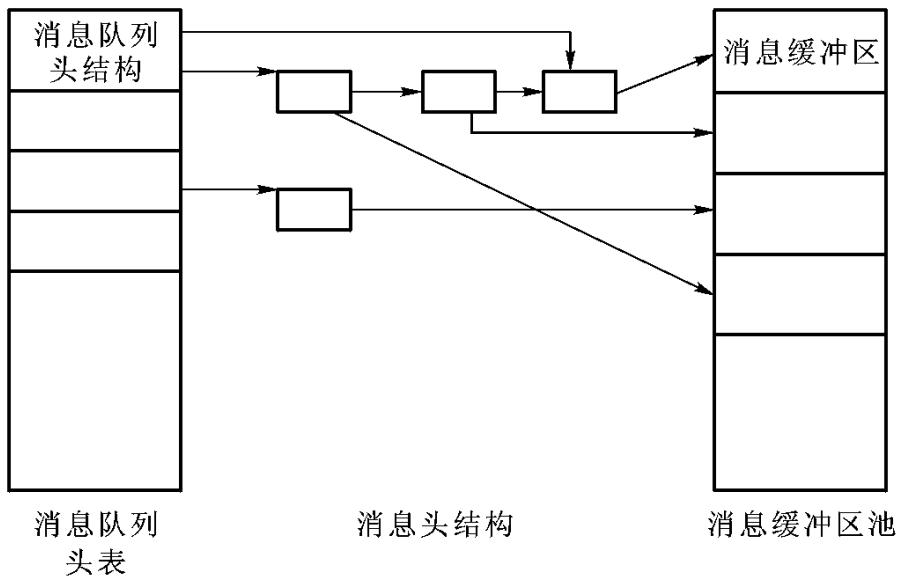


图 3-11 UNIX/Linux 消息队列数据结构

- 消息头结构和消息头表, 消息头表是由消息头结构组成的数组, 个数为 100。消息头结构包含消息类型、消息正文长度、消息缓冲区指针和消息队列中下一个消息头结构的链指针。
- 消息队列头结构和消息队列头表, 由于可有多个消息队列, 于是对应每个消息队列都有一个消息队列头结构, 消息队列头表是由消息队列头结构组成的数组。消息队列头结构包括: 指向队列中第一个消息的头指针、指向队列中最后一个消息的尾指针、队列中消息个数、队列中消息数据的总字节数、队列允许的消息数据最大字节数、最近一次发送/接收消息进程标识和时间。

UNIX/Linux 消息传递机制的系统调用有四个:

- 建立一个消息队列 msgget
- 向消息队列发送消息 msgsnd
- 从消息队列接收消息 msgrecv
- 取或送消息队列控制信息 msgctl

当用户使用 msgget 系统调用来建立一个消息队列时, 内核查遍消息队列头表以确定是否已有一个用户指定的关键字的消息队列存在, 如果没有, 内核创建一个新的消息队列, 并返回给用户一个队列消息描述符; 否则, 内核检查许可权后返回。进程使用 msgsnd 发送一个消息, 内核检查发送进程是否对该消息描述符有写许可权, 消息长度不超过规定的限制等; 接着分配给一个消息头结构, 链入该消息头结构链的尾部; 在消息头结构中填入相应信息, 把用户空间的消息复制到消息缓冲池的一个缓冲区, 让消息头结构的指针指向消息缓冲区, 修改数据结构; 然后, 内核便唤醒等待该消息队列消息的所有进程。进程使用 msgrecv 接收一个消息, 内核检查接收进程是否对该消息描述符有读许可权, 根据消息类型(大于、小

于、等于 0)找出所需消息(等于 0 时取队列中的第一个消息。大于 0 时取队列中给定类型的第一个消息。小于 0 时取队列中小于或等于所请求类型的绝对值的所有消息中最低类型的第一个消息。),从内核消息缓冲区复制内容到用户空间,消息队列中删去该消息,修改数据结构,如果有发送进程因消息满而等待,内核便唤醒等待该消息队列的所有进程。用户在建立了消息队列后,可使用 msgctl 系统调用来读取状态信息并进行修改,如查询消息队列描述符、修改消息队列的许可权等。

第四,关于信件的格式问题和其他有关问题。单机系统中信件的格式可以分直接信件(又叫定长格式)和间接信件(又叫变长格式)。前者将消息放在信件中直接交给收信者,但信息量较小;后者信件中仅传送消息的地址,一般说信息量没有限制。计算机网络环境下的信件格式较为复杂,通常分成消息头和消息体,前者包括了发送者、接收者、消息长度、消息类型、发送时间等各种控制信息;后者则包含了消息内容。此外,还要考虑其他特征:如通信链路、收发双方同步方式等。对进程通信有影响的通信链路特征有:链路是点对点方式还是广播方式、链路是否带缓冲区、链路是单向还是双向的。数据格式分成字节流和报文两类,报文方式还可细分为定长/不定长报文。

第五,关于通信进程的同步问题。两个进程间的消息通信就隐含有某种程度的同步,当发送进程执行 send 发出一封信件后,它本身的执行可以分两种情况,一种是阻塞型,等待收到接收进程回答消息后才继续进行下去;另一种是非阻塞型,发出信件后不等回信立即执行下去,直到某个时刻需要接收进程送来的消息时,才对回答信件进行处理。对于接收进程来说,执行 receive 后也可以是阻塞型和非阻塞型的,前者指直到信件交付完成它都处于等待信件状态;后者则不要求进程等待,当它需要信件时,再去查找并接收信件,需要时再发送回答信件。

发送进程阻塞和接收进程也阻塞的情况主要用于进程不设缓冲区,采用紧密同步方式,两个进程平时均处于阻塞状态,直到有消息传递才被唤醒工作。采用非阻塞型 send 和阻塞型 receive 是用得很广的一种同步方式,发送进程不阻塞,故可把一个或多个消息发给目标进程;而接收进程平时处于阻塞状态,直到发送进程发来消息才被唤醒。例如,在服务器上提供的服务进程平时处于阻塞状态,一旦请求服务的消息到达,它被唤醒来完成用户要求的服务。发送进程和接收进程均不阻塞的同步方式也常被使用,平时两个进程均忙于自己的工作,仅当发生事件而无法继续时,才把自己阻塞起来。例如,发送和接收进程共同联系一个能容纳 n 个消息的消息队列,发送进程可连续发送消息进消息队列,而接收进程可连续地从消息队列获取消息。仅当消息队列中消息数达到 n 个时,发送进程才会被阻塞。类似地,仅当消息队列中消息数为 0 时,接收进程才会被阻塞。而采用阻塞型 receive 是最自然的,因为请求消息的进程大都需要接收并处理消息后才能继续执行下去。但是在分布式系统中很可能丢失发送消息,导致接收进程无限期被阻塞,这个问题要妥然解决。

3.6 死 锁

3.6.1 死锁的产生

计算机系统中有许多独占资源,它们在任一时刻都只能被一个进程使用,如磁带机、绘图仪等独占型外围设备,或进程表、临界区等软件资源。两个进程同时向一台打印机输出将导致一片混乱,两个进程同时进入临界区将导致数据错误乃至程序崩溃。正因为这些原因,所有操作系统都具有授权一个进程独立访问某一资源的能力。一个进程需要使用独占型资源必须通过以下的次序:

- 申请资源
- 使用资源
- 归还资源

若申请时资源不可用,则申请进程进入等待状态。对于不同的独占资源,进程等待的方式是有差异的,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码,由申请进程等待一段时间之后重试。值得指出的是,不同的操作系统对于同一种资源采取的等待方式也是有差异的。

在许多应用中,一个进程需要独占访问多个资源,而操作系统允许多个进程并发执行共享系统资源时,此时可能会出现进程永远被阻塞的现象。例如,两个进程分别等待对方占有的一个资源,于是两者都不能执行而处于永远等待。这种现象称为“死锁”。为了清楚地说明死锁情况,下面列举若干死锁的例子。

例 1 进程推进顺序不当产生死锁。

设系统有打印机、读卡机各一台,它们被进程 P 和 Q 共享。两个进程并发执行,它们按下列次序请求和释放资源:

| 进程 P | 进程 Q |
|-------|-------|
| 请求读卡机 | 请求打印机 |
| 请求打印机 | 请求读卡机 |
| 释放读卡机 | 请求读卡机 |
| 释放打印机 | 释放打印机 |

由于进程 P 和 Q 执行时,相对速度无法预知,当出现进程 P 占用了读卡机,进程 Q 占用了打印机后,进程 P 又请求打印机,但因打印机被进程 Q 占用,故进程 P 处于等待资源状

态;这时,进程 Q 执行,它又请求读卡机,但因读卡机被进程 P 占用而也只好处于等待资源状态。它们分别等待对方占用的资源,致使无法结束这种等待,产生了死锁。但是如果它们速度有快有慢,可以避免上述僵局是而不产生死锁的。

例 2 PV 操作使用不当产生死锁。

设进程 Q1 和 Q2 共享两个资源 r1 和 r2, s1 和 s2 是分别代表资源 r1 和 r2 能否被使用的信号量时,由于资源是共享的,必须互斥使用,因而, s1 和 s2 的初值均为 1。假定两个进程都要求使用两个资源,它们的程序编制如下:

| 进程 Q1 | 进程 Q2 |
|-------------|-------------|
| | |
| P(s1); | P(s2); |
| P(s2); | P(s1); |
| | |
| 使用 r1 和 r2; | 使用 r1 和 r2; |
| | |
| V(s1); | V(s2); |
| V(s2); | V(s1); |
| | |

由于 Q1 和 Q2 并发执行,于是可能产生这样的情况:进程 Q1 执行了 P(s1) 后,在执行 P(s2) 之前,进程 Q2 执行了 P(s2),当进程 Q1 再执行 P(s2) 时将等待,此时 Q2 再继续执行 P(s1),也处于等待。这种等待都必须由对方来释放,显然就产生了死锁。注意这里发生死锁未涉及到资源,而是 P 操作安排不当,所以死锁也可能在不包括资源的情况下产生。

例 3 同类资源分配不当引起死锁

若系统中有 m 个资源被 n 个进程共享,当每个进程都要求 K 个资源,而 $m < nrK$ 时,即资源数小于进程所要求的总数时,如果分配不得当就可能引起死锁。例如, $m = 5$, $n = 5$, $k = 2$, 采用的分配策略是为每个进程轮流分配。首先,为每个进程轮流分配一个资源,这时系统中的资源都已分配完了;于是第二轮分配时,各进程都处于等待状态,导致了死锁产生。

例 4 对临时性资源使用不加限制引起死锁

在进程通信时使用的信件可以看作是一种临时性资源,如果对信件的发送和接收不加限制的话,则可能引起死锁。例如进程 P1 等待进程 P3 的信件 S3 来到后再向进程 P2 发送信件 S1;P2 又要等待 P1 的信件 S1 来到后再向 P3 发送信件 S2;而 P3 也要等待 P2 的信件 S2 来到后才能发出信件 S3。在这种情况下就形成了循环等待,永远结束不了,产生死锁。

综合上面的例子可见,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。

出现死锁会造成很大的损失,因此,必须花费额外的代价来预防死锁的出现。可从三个方面来解决死锁问题,它们是死锁防止(deadlock prevention)、死锁避免(deadlock avoidance)、死锁检测和恢复(deadlock detection and recovery)。

3.6.2 死锁的定义

死锁可能是由于竞争资源而产生,也可能是由于程序设计的错误所造成,因此,在讨论死锁的问题时,为了避免和硬件故障以及其他程序性错误纠缠在一起,特作如下假定:

假定 1:任意一个进程要求资源的最大数量不超过系统能提供的最大量。

假定 2:如果一个进程在执行中所提出的资源要求能够得到满足,那么,它一定能在有限的时间内结束。

假定 3:一个资源在任何时间最多只为一个进程所占有。

假定 4:一个进程一次申请一个资源,且只在申请资源得不到满足时才处于等待状态。换言之,其他一些等待状态,例如:人工干预、等待外围设备.传输结束等,在没有故障的条件下,可以在有限长的时间内结束,不会产生死锁。因此,这里不考虑这种等待。

假定 5:一个进程结束时释放它占有的全部资源。

假定 6:系统具有有限个进程和有限个资源。

现在来给出死锁的定义:

一组进程处于死锁状态是指:如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件,则称一组进程或系统此时发生了死锁。例如, n 个进程 $P_1, P_2, \dots, P_n, P_i$ ($i = 1, \dots, n$) 因为申请不到资源 R_j ($j = 1, \dots, m$) 而处于等待状态,而 R_j 又被 P_{i+1} ($i = 1, \dots, n-1$) 占有, P_n 欲申请的资源被 P_1 占有,显然,此时这 n 个进程的等待状态永远不能结束,则说这 n 个进程处于死锁状态。

3.6.3 死锁的防止

1. 死锁产生的条件

1971 年 Coffman 总结出了对于可再使用资源,系统产生死锁必定同时保持四个必要条件:

- 互斥条件(mutual exclusion):进程应互斥使用资源,任一时刻一个资源仅为一个进程独占,若另一个进程请求一个已被占用的资源时,它被置成等待状态,直到占用者释放资源。

- 占有和等待条件(hold and wait):一个进程请求资源得不到满足而等待时,不释放已占有的资源。

- 不剥夺条件(no preemption):任一进程不能从另一进程那里抢夺资源,即已被占用的资源,只能由占用进程自己来释放。

- 循环等待条件(circular wait)：存在一个循环等待链，其中，每一个进程分别等待它前一个进程所持有的资源，造成永远等待。

以上前三个条件是死锁存在的必要条件，但不是充分条件。第四个条件是前三个条件同时存在时产生的结果，所以，这些条件并不完全独立。但单独考虑每个条件是有用的，只要能破坏这四个必要条件之一，死锁就可防止。

破坏第一个条件(互斥条件)，使资源可同时访问而不是互斥使用，是个简单的办法，磁盘可用这种办法管理，但有许多资源往往是不能同时访问的，所以这种做法许多场合行不通。

采用剥夺式调度方法可以破坏第三个条件(不剥夺条件)，但剥夺调度方法目前只适用于对主存资源和处理器资源的分配，当进程在申请资源未获准许的情况下，如能主动释放资源(一种剥夺式)，然后才去等待，以后再一起向系统提出申请，也能防止死锁，但这些办法不适用于所有资源。由于种种死锁防止办法施加于资源的限制条件太严格，会造成资源利用率下降。下面介绍两种比较实用的死锁防止方法，它们能破坏第二个条件或第四个条件。

2. 静态分配策略

所谓静态分配是指一个进程必须在执行前就申请它所要的全部资源，并且直到它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要求的资源总和不超过系统拥有的资源数。采用静态分配后，进程在执行中不再申请资源，因而不会出现占有了某些资源再等待另一些资源的情况，即破坏了第二个条件(占有和等待条件)的出现。静态分配策略实现简单，被许多操作系统采用。但这种策略严重地降低了资源利用率，因为在每个进程所占有的资源中，有些资源在进程较后的执行时间里才使用，甚至有些资源在例外的情况下被使用。这样就可能造成一个进程占有了一些几乎不用的资源而使其他想用这些资源的进程产生等待。

3. 层次分配策略

层次分配策略将阻止第四个条件(循环等待条件)的出现。在层次分配策略下，资源被分成多个层次，一个进程得到某一层的一个资源后，它只能再申请在较高一层的资源；当一个进程要释放某层的一个资源时，必须先释放所占用的较高层的资源，当另一个进程获得了某一层的一个资源后，它想再申请该层中的另一个资源，必须先释放该层中的已占资源。

这种策略的一个变种是按序分配策略。把系统的所有资源排列成一个顺序，例如，系统若共有 n 个进程，共有 m 个资源，用 r_i 表示第 i 个资源，于是这 m 个资源是：

$$r_1, r_2, \dots, r_m$$

规定如果进程不得在占用资源 r_i ($1 \leq i \leq m$) 后再申请 r_j ($j < i$)。不难证明，按这种策略分配资源时系统不会发生死锁。可以用反证法来证明按序分配不会产生死锁，事实上，若在时刻 t_1 ，进程 P1 处于等资源 r_{k1} 的状态，则 r_{k1} 必为另一进程假定是 P2 所占用，若 P2 在

有限时间里可以运行结束, P₁ 就不会处于永远等待状态; 所以, 一定在某个时刻 t_2 , 进程 P₂ 占有了资源 r_{k1} 而处于永远等待资源 r_{k2} 状态。如此推下去, 按假定系统只有有限个进程, 即必有某个 n , 在时刻 t_n 时, 进程 P_n 永远等待资源 r_{kn} 的状态, 而 r_{kn} 必为前面的某一个进程 P_i 占用 ($1 \leq i < n$)。于是, 按照上述的按序分配策略, 当 P₂ 占用了 r_{k1} 后再申请 r_{k2} 必有:

$$k_1 < k_2$$

依此类推, 可得:

$$k_2 < k_3 < \cdots < k_i < \cdots < k_n$$

但是, 由于进程 P_i 是占有了 r_{kn} 却要申请资源 r_{ki} , 那么, 必定有:

$$k_n < k_i$$

这就产生了矛盾。所以, 按序分配策略可以防止死锁。

层次分配比静态分配在实现上要多花一点代价, 但它提高了资源使用率。然而, 如果一个进程使用资源的次序和系统内的规定各层资源的次序不同时, 这种提高可能不明显。假如系统中的资源从高到低按序排列为: 卡片输入机、行式打印机、卡片输出机、绘图仪和磁带机。若一个进程在执行中, 较早地使用绘图仪, 而仅到快结束时才用磁带机。但是, 系统规定, 磁带机所在层次低于绘图仪所在层次。这样, 进程使用绘图仪前就必须先申请到磁带机, 这台磁带机就在一长段时间里空闲着直到进程执行到结束前才使用, 这无疑是低效率的。

3.6.4 死锁的避免

破坏死锁的四个条件之一能防止系统发生死锁, 但这会导致低效的进程运行和资源使用率。死锁的避免则相反, 它允许系统中同时存在四个必要条件, 如果能掌握并发进程中与每个进程有关的资源动态申请情况, 做出明智和合理的选择, 仍然可以避免死锁的发生。每当在为申请者分配资源前先测试系统状态, 若把资源分配给申请者会产生死锁的话, 则拒绝分配, 否则接受申请, 为它分配资源。

死锁避免不是通过对进程随意强加一些规则, 而是通过对每一次资源申请进行认真的分析来判断它是否能安全地分配。问题是: 是否存在一种算法总能做出正确的选择从而避免死锁? 答案是肯定的, 但条件是必须事先获得与进程有关的一些特定信息。本节将讨论使用银行家算法(banker's algorithm)来避免死锁。

1. 资源轨迹图

避免死锁的主要方法是让系统处于安全状态, 在图 3-12 中, 给出了一个处理两个进程和两种资源(打印机和绘图仪)的模型。横轴表示进程 A 的指令执行过程, 纵轴表示进程 B 的指令执行过程。进程 A 在 I_1 处请求一台打印机, 在 I_3 处释放, 在 I_2 处申请一台绘图仪, 在 I_4 处释放。进程 B 在 I_5 到 I_7 之间需要绘图仪, 在 I_6 到 I_8 之间需要打印机。

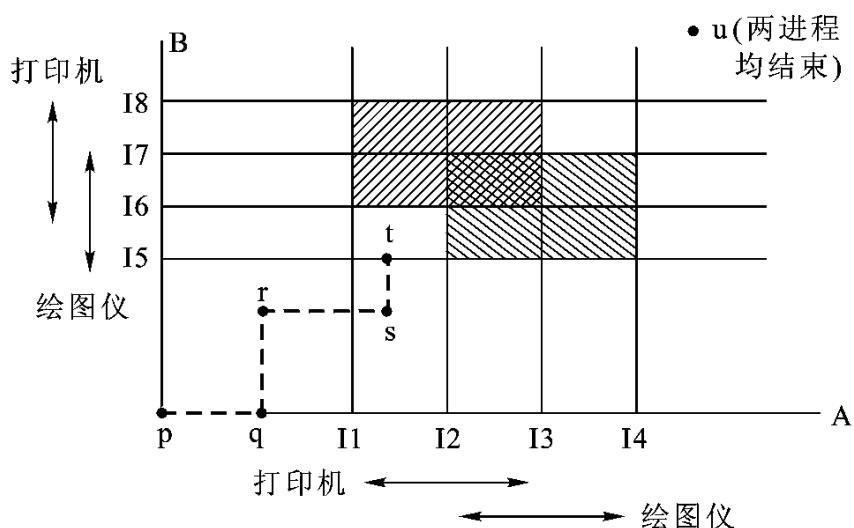


图 3-12 两个进程的资源轨迹图

图中的每一点都示出了两个进程的状态。初始点为 P, 若 A 先运行, 则在 A 执行一段指令后到达 q, 在 q 点若 B 开始运行, 则轨迹向垂直方向移动。在单处理机情况下, 所有路径都只能是水平或垂直方向的。同时运动方向一定是向右或向上, 而不会是向左或向下, 因为, 进程的执行不可能后退。

当进程 A 由 r 向 s 移动, 穿过 I_1 线时, 它请求打印机并获得成功。当进程 B 到达 t 时, 它申请绘图仪。图中的阴影部分很重要, 打着从左下到右上斜线的部分表示在该区域中两个进程都拥有打印机, 而互斥使用的规则决定了不可能进入该区域。同样的, 另一种斜线的区域表示两个进程都拥有绘图仪, 且同样不可进入。

如果系统一旦进入由 I_1 、 I_2 和 I_5 、 I_6 组成的矩形区域, 那么, 最后一定会到达 I_2 和 I_6 交叉点, 此时进程处在不安全区域, 就会发生死锁。在该点处, A 申请绘图仪, B 申请打印机, 而且这两种资源均已被分配。这整个矩形区域都是不安全的, 因此, 绝不能进入这个区域。在 t 处惟一的办法是运行进程 A 直到 I_4 , 过了 I_4 后则可以按任何路线前进, 直到终点 u。

2. 单种资源的银行家算法

Dijkstra(1965)提出了一种能够避免死锁的调度方法, 称为银行家算法。它的模型基于一个小城镇的银行家, 现将该算法描述如下: 假定一个银行家拥有资金, 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:

- 每个客户必须预先说明自己所要求的最大资金量;
- 每个客户每次提出部分资金量申请和获得分配;
- 如果银行满足了客户对资金的最大需求量, 那么, 客户在资金运作后, 应在有限时间内全部归还银行。

只要每个客户遵守上述约束, 银行家将保证做到: 若一个客户所要求的最大资金量不超

过 Σ , 则银行一定接纳该客户, 并可处理他的资金需求; 银行在收到一个客户的资金申请时, 可能因资金不足而让客户等待, 但保证在有限时间内让客户获得资金。在银行家算法中, 客户可看作进程, 资金可看作资源, 银行家可看作操作系统, 这里叙述的是单资源银行家算法,

还可以扩展到多资源银行家算法。在图 3-13(a) 中列出了 4 个客户, 每个客户都有一个贷款额度, 银行家知道不可能所有客户同时都需要最大贷款额, 从而, 他只保留 10 个单位的资金来为客户服务, 而不是 22 个单位。图 3-13(b) 所示的状态是客户们各做自己的生意, 在某些时刻需要贷款。客户已获得的贷款(已分配的资源)和可用的最大数额贷款称为与资源分配相关的系统状态。

| | 已使用 | 最大 | | |
|---------|-----|----|--------|--|
| 名字 | ↓ | ↓ | | |
| Andy | 0 | 6 | | |
| Barbara | 0 | 5 | | |
| Marvin | 0 | 4 | | |
| Suzanne | 0 | 7 | | |
| | | | 可用: 10 | |

| | 已使用 | 最大 | | |
|---------|-----|----|-------|--|
| 名字 | ↓ | ↓ | | |
| Andy | 1 | 6 | | |
| Barbara | 1 | 5 | | |
| Marvin | 2 | 4 | | |
| Suzanne | 4 | 7 | | |
| | | | 可用: 2 | |

| | 已使用 | 最大 | | |
|---------|-----|----|-------|--|
| 名字 | ↓ | ↓ | | |
| Andy | 1 | 6 | | |
| Barbara | 1 | 5 | | |
| Marvin | 2 | 4 | | |
| Suzanne | 4 | 7 | | |
| | | | 可用: 1 | |

图 3-13 三种资源分配状态 (a) 安全 (b) 安全 (c) 不安全

一个状态被称为是安全的, 当存在一个状态序列能够使所有的客户均得到其所有的贷款(即所有的进程得到所需的全部资源并终止)。图 3-13(b) 所示的状态是安全的, 能使 Marvin 运行结束, 然后释放所有的 4 个单位资金。如此这样下去便可以满足 Suzanne 或者 Barbara 的请求, 等等。

考虑假如给 Barbara 另一个她申请的资源, 如图 3-13(c), 则得到如图 3-13(c) 所示的状态, 该状态是不安全的。如果忽然所有的客户都申请, 希望得到最大贷款额, 而银行家无法满足其中任何一个的要求, 则发生死锁。不安全状态并不一定导致死锁, 而仅仅是有可能产生死锁, 因为, 客户未必需要其最大贷款额度, 但银行家不敢抱这种侥幸心理。

银行家算法就是对每一个请求进行检查, 检查这次资源申请是否会导致不安全状态。若是, 则不满足该请求, 否则便满足。检查状态是否安全的方法是看他是否有足够的资源满足一个距最大需求最近的客户。如果可以, 则这笔投资认为是能够收回的, 接着检查下一个距最大需求最近的客户, 如此反复下去。如果所有投资最终都被收回, 则该状态是安全的, 最初的请求可以批准。

4. 多种资源的银行家算法

资源轨迹图的方法很难被扩充到系统中有任意数目的进程、任意种类的资源, 并且每种

资源有多个实例的情况。但银行家算法可以被推广用来处理这个问题。图 3-14 示出了其工作原理。从中看到了两个矩阵。左边的显示出对 5 个进程分别已分配的各种资源数, 右边的则显示了使各进程运行完所需的各种资源数。与单种资源的情况一样, 各进程在执行前给出其每种资源所需的全部资源量, 系统的每一步都可以计算出右边的矩阵。

图 3-14 最右边的三个向量分别表示总的资源 E、已分配资源 P, 和剩余资源 A。由 E 可知系统中共有 6 台磁带机, 3 台绘图仪, 4 台打印机和 2 台 CD-ROM。由 P 可知当前已分配了 5 台磁盘机, 3 台绘图仪, 2 台打印机和 2 台 CD-ROM。该向量可通过将左边矩阵的各列相加得到, 剩余资源向量可通过从资源总数中减去已分配资源数得到。

| | 绘图仪 | | | | |
|-----------|-----|---|---|---|---|
| 磁带机 进程 | A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 | |
| C | 1 | 1 | 1 | 0 | |
| D | 1 | 1 | 0 | 1 | |
| E | 0 | 0 | 0 | 0 | |

已分配的资源

| | 绘图仪 | | | | |
|-----------|-----|---|---|---|---|
| 磁带机 进程 | A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 | |
| C | 3 | 1 | 0 | 0 | |
| D | 0 | 0 | 1 | 0 | |
| E | 2 | 1 | 1 | 0 | |

仍需要的资源

$$E = (6342)$$

$$P = (5322)$$

$$A = (1020)$$

图 3-14 多种资源的银行家算法

检查一个状态是否安全的步骤如下:

- (1) 查找右边矩阵是否有一行, 其未被满足的设备数均小于或等于向量 A。如果找不到, 则系统将死锁, 因为任何进程都无法运行结束。
- (2) 若找到这样一行, 则可以假设它获得所需的资源并运行结束, 将该进程标记为结束, 并将资源加到向量 A 上。
- (3) 重复以上两步, 直到所有的进程都标记为结束。若达到所有进程结束, 则状态是安全的, 否则将发生死锁。

如果在第 1 步中同时存在若干进程均符合条件, 则不管挑选哪一个运行都没有关系, 因为, 可用资源或者将增多, 或者在最坏情况下保持不变。

图中所示的状态是安全的, 进程 B 现在再申请一台打印机, 可以满足它的请求, 而且保持系统状态仍然是安全的(进程 D 可以结束, 然后是 A 或 E, 剩下的进程最后结束)。假设进程 B 获得一台打印机后, E 试图获得最后的一台打印机, 若分配给 E, 可用资源向量将减到 (1000), 这时将导致死锁。显然 E 的请求不能立即满足, 必须延迟一段时间。

该算法最早由 Dijkstra 于 1965 年发表。从那之后几乎每本操作系统的专著都详细地描

述它,许多论文的内容也围绕该算法讨论,其主要优点是不需要死锁预防中加上的种种限制,如资源剥夺或重新运行进程。但很少有作者指出该算法缺乏实用价值。因为,进程很难在运行前就知道其所需资源的最大量;而且系统中的进程必须是无关的,相互间没有同步要求;进程的个数和分配的资源数目应该是固定的。这些要求往往事先难以满足。

总之,死锁预防的方案过于严格,死锁避免的算法又需要获得难以得到的信息。对于特殊的应用有许多很好的算法。例如,在许多数据库系统中,常常需要将若干记录上锁然后进行更新。当有多个进程同时运行时,有可能发生死锁。常用的一种解法是两阶段上锁法。第一阶段,进程试图将其所需的全部记录加锁,一次锁一个记录。若成功,则数据进行更新并解锁。否则若有些记录已被上锁,还有一些没有锁住,那么,它将已上锁的记录解锁并重新开始执行,该解法有点类似提前申请全部资源的方法。

但这种方法不通用,在实时系统和过程控制系统中不能够因为资源不可用而将进程中途终止并重新执行。同样,若一个进程已进行过网络消息的读写、更新文件、或其他不宜重复的操作,则将进程重新从头执行是不可接受的。该算法仅适用于那些在第一阶段可以随时停止并重新执行的程序,遗憾的是并非所有的应用都可以按这种方式组织。

5. 银行家算法的数据结构和安全性测试算法

考虑一个系统有 n 个进程和 m 种不同类型的资源,现定义包含以下向量和矩阵的数据结构:

- 系统每类资源总数——该 m 个元素的向量为系统中每类资源的数量:

$$\text{Resource} = (R_1, R_2, \dots, R_m)$$

- 每类资源未分配数量——该 m 个元素的向量为系统中每类资源尚可

$$\text{Available} = (V_1, V_2, \dots, V_m)$$

- 最大需求矩阵——每个进程对每类资源的最大需求量, C_{ij} 表示进程 P_i 需 R_j 类资源最大数,为实现死锁避免,这个信息必须事先获得:

$$\text{Claim} = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$$

- 分配矩阵——表示进程当前已分得的资源数, A_{ij} 表示进程 P_i 已分到 R_j 类资源的个数:

$$\text{Allocation} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$$

于是下列关系式确保成立:

- $R_i = V_i + \sum A_{ki}$ 对 $i = 1, \dots, m, k = 1, \dots, n$; 表示所有的资源要么已被分配, 要么尚可分配。
- $C_{ki} \leq R_i$ 对 $i = 1, \dots, m, k = 1, \dots, n$; 表示进程申请资源数不能超过系统拥有的这类资源总数。
- $A_{ki} \leq C_{ki}$ 对 $i = 1, \dots, m, k = 1, \dots, n$; 表示进程申请任何类资源数不能超过声明的最大资源需求数。

现在可以给出一种死锁避免策略: 系统中若要启动一个新进程工作, 其对资源 R_i 的需求仅当满足下列不等式:

$$R_i \geq C_{(n+1)i} + \sum C_{ki} \quad i = 1, \dots, m, k = 1, \dots, n;$$

也就是说, 应满足当前系统中所有进程对资源 R_i 的最大资源需求数, 加上启动的新进程的最大资源需求数, 不超过系统拥有的最大数时, 才能启动该进程, 所以, 这种死锁避免方法也称作“进程启动拒绝法”。该算法考虑了最坏的情况、即所有进程同时要使用它们声明的最大资源需求数, 因而, 很难是最优的。

根据上面的讨论, 可以给出系统安全性定义: 在时刻 T_0 系统是安全的, 仅当存在一个进程序列 P_1, \dots, P_n , 对进程 P_k ($k = 1, \dots, n$) 满足公式:

$$C_{ki} - A_{ki} \leq Available_i + \sum A_{ji} \quad j = 1, \dots, k-1; k = 1, \dots, n; i = 1, \dots, m;$$

该序列称安全序列。其中公式左边表示进程 P_k 尚缺少的各类资源; $Available_i$ 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; $\sum A_{ji}$ 表示排在进程 P_k 之前的所有进程占用的 P_k 所需要的资源的总数。

显然, 一个进程 P_k 所需资源若不能立即被满足, 那么, 在所有 P_j ($j = 1, \dots, k-1$) 运行完成后可以满足, 然后 P_k 也能获得资源完成任务; 当 P_k 释放全部资源后, P_{k+1} 也能获得资源完成任务; 如此下去, 直到最后一个进程完成任务, 从 T_0 时刻起按这个进程序列运行系统是安全的, 绝不会产生死锁。

银行家算法又称作“资源分配拒绝”法, 其基本思想是: 系统中的所有进程进入进程集合, 在安全状态下系统收到一个进程的资源请求后, 先把资源试探性分配给它。现在, 系统用剩下的可用资源和每个进程集合中其他进程还要的资源数作比较, 在进程集合中找到剩余资源能满足最大需求量的进程, 从而, 保证这个进程运行完毕并归还全部资源。这时, 把这个进程从集合中去掉, 系统的剩余资源更多了, 再反复执行上述步骤。最后, 检查进程集合, 若为空表明本次申请可行, 系统处于安全状态, 可真正实施本次分配; 否则, 只要有进程执行不完, 系统便处于不安全状态, 本次资源分配暂不实施, 让申请进程等待。下面是银行家算法(banker's algorithm)的程序及简短说明。

```
type state = record
```

```
全局数据结构}
```

```

        resource, available: array[0...m-1] of integer;
        claim, allocation: array[0...n-1, 0...m-1] of integer;
        end

                资源分配算法}

if alloc[i, *] + request[*] > claim[i, *] then < error >
                申请量超过最大需求量}

else
    if request[*] > available[*] then < suspend process. >
    else
                    模拟分配}

        < define newstate by:
            allocation[i, *]:= allocation[i, *] + request[*]
            available[*]:= available[*] - request[*] >
    end;

if safe(newstate) then
    < carry out allocation >
else
    < restore original state >
    < suspend process >
end
end

                安全性测试算法}

function safe(state:s):boolean;
var currentavail: array[0...m-1] of integer;
    rest: set of process;
begin
    currentavail:= available;
    rest:= {all process};
    possible:= true;
while possible do
    find a Pk in rest such that
        claim[k, *] - allocation[k, *] ≤ currentavail;
    if found then
        currentavail:= currentavail + allocation[k, *];
        rest:= rest - [Pk];
    else

```

```

possible := false;
end
end;
safe := (rest = null);
end.

```

再对上述算法作简短说明：

(1) 申请量超过最大需求量时出错, 否则转(2);

(2) 申请量超过目前系统拥有的可分配量时, 挂起进程等待, 否则转(3);

(3) 系统对 P_i 进程请求资源作试探性分配、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), 如果安全状态则承认识试分配, 否则抛弃试分配, 进程 P_i 等待。

(5) 安全性测试算法

- 定义工作向量 currentavail 和布尔型标志 possible ; 初始化

$$\text{currentavail} := \text{available}, \text{possible} := \text{true};$$

- 保持 $\text{possible} := \text{true}$, 从进程集合 rest 中找出 $\text{claim}[k, *] - \text{allocation}[k, *] \leq \text{currentavail}$ 的进程来, 如找到, 则释放这个进程 P_k 的全部资源、执行以下操作 $\text{currentavail} := \text{currentavail} + \text{allocation}[k, *]$, 把 P_k 从进程集合中去掉 $\text{rest} := \text{rest} - [P_k]$; 否则 $\text{possible} := \text{false}$, 停止执行算法;

- 最后查看进程集 rest , 若为空集返回安全标记; 否则返回不安全标记。

下面用一个实例来说明银行家算法用于检查系统所处的状态是安全或不安全的。如果系统中共有五个进程 $\{P_0, P_1, P_2, P_3, P_4\}$ 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 , 系统目前资源分配情况如下:

| Process | 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 | | | |

每个进程目前还需资源为 $C_{ki} - A_{ki}$,

| Process | $C_{ki} - A_{ki}$ | | |
|----------------|-------------------|---|---|
| | 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

可以断言目前系统处于安全状态, 因为, 在 T_0 时刻序列 $\{P_1, P_3, P_4, P_2, P_0\}$ 能满足安全性条件。下表列出了 T_0 时刻的安全序列:

| 资源 进程 | currentavil | | | $C_{ki} - A_{ki}$ | | | allocation | | | currentavil + 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) P_1 请求资源

现在假定进程 P_1 又要申请 1 个 A 类资源和 2 个 C 类资源, 为了判别这个申请能否立即准许, 按银行家算法进行检查:

- $request1(1, 0, 2) \leq C_{k1} - A_{k1}(1, 2, 2)$
- $request1(1, 0, 2) \leq Available(3, 3, 2)$

系统先尝试为 P_1 分配资源, 修改 Available、Allocation 和 $C_{ki} - A_{ki}$ 得到了下面的新状态:

| Process | allocation | | | claim | | | 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 | | | |

再用安全性算法检查系统此时的状态是否安全? 得到下面的 P_1 申请资源时的安全性

分析表：

| 资源 进程 | currentavil | | | $C_{ki} - A_{ki}$ | | | allocation | | | currentavil + 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 |

可见能找到一个安全序列 {P₁, P₃, P₄, P₀, P₂}，因此，系统此时处于安全状态，可正式把资源分配给进程 P₁。

(3) 进程 P₄ 请求资源

如果进程 P₄ 发出资源请求，系统按银行家算法进行检查：

- $\text{request}_4(3, 3, 0) \leq C_{k4} - A_{k4}(4, 3, 1)$
- $\text{request}_4(3, 3, 0) > \text{Available}(2, 3, 0)$

由于这时可用资源不足，申请被系统拒绝，应令进程 P₄ 等待。

(4) 进程 P₀ 申请资源

进程 P₀ 发出资源申请，系统按银行家算法进行检查：

- $\text{request}_0(0, 2, 0) \leq C_{k0} - A_{k0}(7, 3, 1)$
- $\text{request}_4(0, 2, 0) \leq \text{Available}(2, 3, 0)$

系统先为 P₀ 进行尝试性分配，修改对应数据，得到以下中间结果：

| 资源 进程 | currentavil | | | $C_{ki} - A_{ki}$ | | | allocation | | | |
|----------------|-------------|---|---|-------------------|---|---|------------|---|---|--|
| | 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，利用安全性算法检查发现：系统能满足进程 P₀ 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需求，故系统已处于不安全状态，故不能为进程 P₀ 分配资源。

3.6.5 死锁的检测和解除

1. 资源分配图和死锁定理

对资源的分配加以限制可以防止和避免死锁的发生,但这不利于各进程对系统资源的充分共享。解决死锁问题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加任何限制,也不采取死锁避免措施,但系统定时地运行一个“死锁检测”程序,判断系统内是否已出现死锁,如果检测到系统已发生了死锁,再采取措施解除它。

操作系统中的每一时刻的系统状态都可以用进程—资源分配图 PRAG (Process Resource Allocation Graph) 来表示,进程 – 资源分配图是描述进程和资源间申请及分配关系的一种有向图,可用以检测系统是否处于死锁状态。设一个计算机系统中有许多类资源和许多个进程。每一个资源类用一个方框表示,方框中的黑圆点表示该资源类中的各个资源,每个进程用一个圆圈来表示,用有向边来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处于等待 R_j 类资源的状态,该有向边从进程开始指到方框的边缘,表示进程 P_i 申请 R_j 类中的一个资源。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,由于已把一个具体的资源分给了进程 P_i ,故该有向边从方框内的某个黑圆点出发指向进程。图 3 – 15 是进程 – 资源分配图的一个例子,其中共有三个资源类,每个进程的资源占有和申请情况已明白地表示在图中。这个例子中,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状态,发生了死锁。

在进程 – 资源分配图中存在环路并不一定发生死锁,因为循环等待资源仅是死锁发生的必要条件,而不是充份条件,图 3 – 16 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 分别占有了一个资源 R_1 和一个资源 R_2 ,并且等待另一个资源 R_2 和另一个资源 R_1 形成了环路,但进程 P_2 和进程 P_4 分别占有了资源 R_1 和资源 R_2 各一个,它们申请的资源已得到了全部满足,因而,能在有限时间内归还占有的资源,于是进程 P_1 和进程 P_3 分别能获得另一个所需资源,这时进程 – 资源分配图中减少了两条请求边,环路不再存在,系统中也就不存在死锁了。

可以利用下列步骤运行一个“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,则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,则系统中发生了死锁,此时,环路是系统发生死锁的充要条件,环路中的进程便为死锁进程;
- (3) 如果进程 – 资源分配图中有环路,且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,未必系统一定就会发生死锁。如果能在进程 – 资

源分配图中找出一个既不阻塞又非独立的进程, 它在有限的时间内有可能获得所需资源类中的资源继续执行, 直到运行结束, 再释放其占有的全部资源。相当于消去了图中此进程的所有请求边和分配边, 使之成为孤立结点。接着可使进程 – 资源分配图中另一个进程获得前面进程释放的资源继续执行, 直到完成又释放出它所占用的所有资源, 相当于又消去了图中若干请求边和分配边。如此下去, 经过一系列简化后, 若能消去图中所有边, 使所有进程成为孤立结点, 则该图是可完全简化的; 否则则称该图是不可完全简化的。系统为死锁状态的充分条件是: 当且仅当该状态的进程 – 资源分配图是不可完全简化的。该充分条件称为死锁定理。

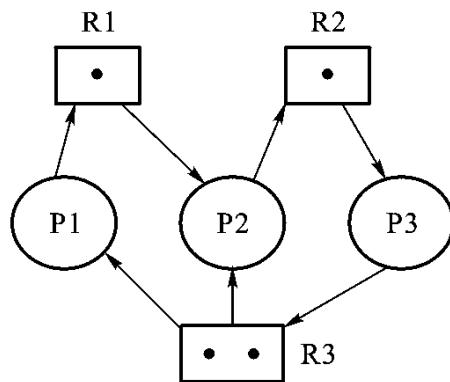


图 3-15 进程资源分配图一个例子

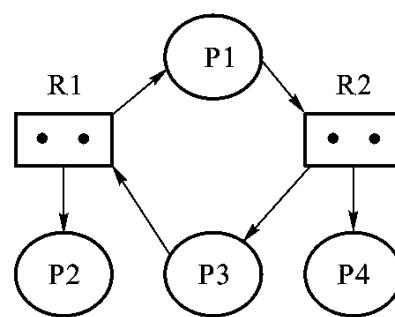


图 3-16 有环路而无死锁的一个例子

2. 死锁的检测和解除方法

下面介绍两种具体的死锁检测方法。

死锁的检测可借助于死锁的安全性测试算法来实现。今定义布尔型向量 $\text{possible}[k]$, $k = 1, \dots, n$ 。检测死锁算法如下:

(1) $\text{currentavail} := \text{available}$;

(2) 在 rest 中查每一个进程 P_k , 如果 $\text{claim}[k, *] - \text{allocation}[k, *] = 0$, 则 $\text{possible}[k] := \text{true}$; 否则 $\text{possible}[k] := \text{false}$; 这里 $k = 1, \dots, n$ 。

(3) 在 rest 中找一个进程 P_k , $\text{request}[*]$ 表示还需要的所有资源, 需满足条件:

$\text{possible}[k] = \text{false} \& (\text{request}[*] \leq \text{currentavail})$

找到这样的 P_k 便转(4); 否则转(5);

(4) $\text{currentavail} := \text{currentavail} + \text{allocation}; \text{possible}[k] := \text{true}$; 然后转(3);

(5) 如果对 $k = 1, \dots, n$ 若 $\text{possible}[k] = \text{true}$ 不成立, 那么, 系统出现了死锁, 并且 $\text{possible}[k] = \text{false}$ 的 P_k 为死锁进程。

下面介绍另一种实现死锁检测的方法, 把进程使用和等待资源的情况用一个状态矩阵 A 来表示:

$$A[b_{ij}] = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{pmatrix}$$

其中 b_{ij} 为 1: 当 P_i 等待被 P_j 占用的资源时, b_{ij} 为 0: 当 P_i 和 P_j 不存在资源等待占用关系时。

于是“死锁检测”程序可用 warshall 的传递闭包算法来测试系统是否有死锁发生。warshall 的传递闭包算法对状态矩阵 A 构成传递闭包 $A^*[b_{ij}]$ 中的每一个 b_{ij} 是对 $A[b_{ij}]$ 执行如下算法得到的:

```

for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
             $b_{ij} := b_{ij} \vee (b_{ik} \wedge b_{kj})$ 

```

其中, b_{ij} 表示进程 P_i 与进程 P_j 有直接等待关系。 $b_{ik} \wedge b_{kj}$ 表示当前进程 P_i 等待进程 P_k 所占的资源且进程 P_k 等待进程 P_j 所占的资源时取值为 1。也就是说, 进程 P_i 与进程 P_j 之间有间接等待关系。warshall 传递闭包算法循环检测了矩阵 A 中的各个元素, 把直接等待资源关系和间接等待资源关系都在传递闭包 A^* 中表示出来。显然, 当 A^* 中有一个 $b_{ii} = 1$ ($i = 1, 2, \dots, n$) 时, 就说明存在着一组进程, 它们循环等待资源, 也即系统出现了死锁。

下面举一个例子进行说明, 假如死锁检测程序已检查并构造出三个进程 P_1, P_2, P_3 的资源等待状态矩阵如下:

$$A[b_{ij}] = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

状态矩阵中有 b_{12}, b_{23}, b_{31} 为 1。当进入死锁检测程序工作后, 对 $k = 1$ 时的各个 i, j 进行检测, 由于 $b_{31} = 1$ 且 $b_{12} = 1$ 而使得 b_{32} 被置成 1。继续对 $k = 2$ 时的各个 i, j 进行检测, 通过 $(b_{12} \wedge b_{23})$ 会使 b_{31} 成为 1, 通过 $(b_{32} \wedge b_{23})$ 会使 b_{23} 置 1。继续对 $k = 3$ 时的各个 i, j 进行检测, 可以发现在检测程序结束时有 b_{11} 和 b_{22} 都为 1, 所以, 肯定系统中发生了死锁, 而且 P_1, P_2, P_3 为死锁进程。

死锁的检测和解除往往配套使用, 当死锁被检测到后, 用各种办法解除系统的死锁, 常用的办法有资源剥夺法、进程回退法、进程撤销法和系统重启法:

- 立即结束所有进程的执行, 并重新启动操作系统。这种方法简单, 但以前工作全部作废, 损失可能很大。
- 撤销陷于死锁的所有进程, 解除死锁继续运行。

- 逐个撤销陷于死锁的进程,回收其资源,直至死锁解除。但是先撤销哪个死锁进程呢?可选择符合下面一种条件的进程先撤销:消耗的CPU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。
- 剥夺陷于死锁的进程占用的资源,但并不撤销它,直至死锁解除。可仿照撤销陷于死锁进程的条件一样来选择剥夺资源的进程。
- 根据系统保存的checkpoint,让所有进程回退,直到足以解除死锁。
- 当检测到死锁时,如果存在某些未卷入死锁的进程,而这些进程随着建立一些新的抑制进程能执行到结束,则它们可能释放足够的资源来解除这个死锁。

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频率,而恢复的代价是时间的损失。

3.7 实例研究:Windows 2000/XP 的同步和通信机制

3.7.1 Windows 2000/XP 的同步和互斥机制

在 Windows 2000/XP 中,提供了互斥对象、信号量对象和事件对象三种同步对象及相应的系统调用,用于进程和线程的同步。这些同步对象都有一个用户指定的对象名称,不同进程中用同样的对象名称来创建或打开对象,从而,获得该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。

互斥对象就是互斥信号量,在一个时刻只能被一个线程使用。它的相关 API 包括:

- CreateMutex 创建一个互斥对象,返回对象句柄。
- OpenMutex 打开并返回一个已存在的互斥对象句柄,用于后续访问。
- ReleaseMutex 释放对互斥对象的占用,使之成为可用。

信号量对象就是资源信号量,初始值所取范围在 0 到指定最大值之间,用于限制并发访问的线程数。它的相关 API 包括:

- CreateSemaphore 创建一个信号量对象,在输入参数中指定初值和最大值,返回对象句柄。
- OpenSemaphore 打开并返回一个已存在的信号量对象句柄,用于后续访问。
- ReleaseSemaphore 释放对信号量对象的占用,使之成为可用。

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的出现,它的相关 API 包括:

- CreateEvent 创建一个事件对象,返回对象句柄。
- OpenEvent 打开并返回一个已存在的事件对象句柄,用于后续访问。

- SetEvent 和 PluseEvent 设置指定事件对象为可用状态。
- ResetEvent 设置指定事件对象为不可用状态。

对于这三种同步对象, 系统提供了两个统一的等待操作 WaitForSingleObjecs 和 WaitForMultipleObjects。前者可在指定的时间内等待指定对象为可用状态; 后者可在指定的时间内等待多个对象为可用状态。

除了上述三种同步对象外, Windows 2000/XP 还提供了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。把变量说明为 CRITICAL_SECTION 类型, 就可作为临界区使用。相关 API 有: InitializeCriticalSection (对临界区对象进行初始化)、EnterCriticalSection (等待占用临界区的使用权, 得到使用权时返回)、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, 用于对整型变量的操作, 可避免线程间切换对操作连续性的影响。这组 API 包括: InterlockedExchange (32 位数据的先读后写原子操作)、InterlockedCompareExchange (依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)。

3.7.2 Windows 2000/XP 进程通信机制

Windows 2000/XP 的信号 (signal) 是进程与外界的一种低级通信方式, 相当于进程的软中断。进程可发送信号, 每个进程都有指定的信号处理例程, 信号通信是单向和异步的。存在两组与信号量相关的系统调用, 分别处理不同信号。

(1) SetConsoleCtrlHandler 和 GenerateConsoleCtrlEvent。前者定义或取消本进程的信号处理例程中的用户定义例程。例如, 缺省时, 每个进程当收到 CTRL_C_EVENT 时都有一个信号 CTRL + C 的处理例程来处理, 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例程, 后者用于发送信号。这一组系统调用处理的信号与 UNIX 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 2000/XP 的共享存储区 (shared memory) 可用于进程间的大数据量通信。进行通信的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 2000/XP 采用文件映射 (file mapping) 机制来实现共享存储区, 用户进程可把整个文件映射为进程虚拟地址空间的一部分来加以访问。与共享存储区相关的系统调用有: 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到物理文件中)、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 2000/XP 的管道(pipe)是一条在进程间以字节流方式传送的通信通道,它利用了系统核心的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。和 UNIX 类似提供了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 进行无名管道读写。命名管道是服务器进程与一个客户进程间的一条通信通道,可实现不同机器上的进程通信,采用 C/S 方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 用在服务器端等待客户进程的请求,CallNamePipe 从管道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 2000/XP 中提供的邮件槽(mailslot)是一种不定长和不可靠的单向消息通信机制。消息的发送不需接收方准备好,随时可发送。邮件槽也采用 C/S 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,它可从邮件槽中读消息,而客户进程可利用邮件槽的名字向它发消息。有关的系统调用有:CreateMailslot(服务器创建邮件槽)、GetMailSlotInfo(服务器查询邮件槽信息)、SetMailslotInfo(服务器设置读操作等待时限)、ReadFile(服务器读邮件槽)、CreateFile(客户方打开邮件槽)和 WriteFile(客户方发送消息)。

Windows 2000/XP 的套接字(socket)是一种网络通信机制,它通过网络在不同计算机上的进程间作双向通信。套接字采用的数据格式可以是可靠的字节流或不可靠的报文,通信模式可为 C/S 或对等模式。为实现不同操作系统上的进程通信,需约定网络通信时不同层次的通信过程和信息格式,TCP/IP 是目前广泛使用的网络通信协议。Windows 2000/XP 中的套接字规范称“Winsock”,它除了支持标准的 BSD 套接字外,还实现了一个与协议独立的 API,可支持多种网络通信协议。

3.8 实例研究:Linux 信号量机制

Linux 支持三种在 UNIX System V 中首创的进程通信机制,它们是:消息队列、共享内存和信号量。前面两种已经在进程通信中介绍过,这里来讨论信号量机制。在 Linux 中,有两类信号量,一类主要被内核使用的信号量称内核信号量,另一类用户和内核都可使用的信号量称信号量集。

内核信号量的定义如下:

```

struct semaphore
{
    atomic_t count;
    int waking;
    struct wait-queue *wait;
};

```

其中,仅有三个分量,资源计数器 count 表示可用的某种资源数,若为正整数则尚有这些资源可用;若为 0 或负整数则资源已用完,且因申请资源而等待的进程有 count 绝对值个。sema-init 宏用于初始化 count 为任何值,可以是 1(二元信号量),可以是任意正整数(一般信号量)。唤醒计数器 waking 记录等待该资源的进程个数,也是当该资源被占用时等待唤醒的进程个数,up 工作期间使用。wait-queue 用于因等待这个信号量代表的资源再次可用而被挂起的进程所组成的等待队列。内核信号量上定义的函数有:down(本书中称作 P 操作)、up(本书中称作 V 操作),以及 down-interruptible、wake-up、wake-up-interruptible 等。Down 和 up 的含义已经在前面讨论过,其余几个函数的含义如下:down-interruptible 函数被进程用于获得信号量,但也愿意在等待它时可被信号中断,当函数的返回的值为 0 时获得了信号量,当函数的返回的值为负时被一个信号中断;wake-up 唤醒所有等待进程;wake-up-interruptible 仅唤醒处 TASK-INTERRUPTIBLE 状态的进程。

下面讨论信号量集。一次可以对一组信号量、即信号量集进行操作,不但能对信号量做加 1 和减 1 操作,也可增减任意整数。Linux 维护着一个信号量集的数组 semary,数组的元素类型是指向 semid-ds 结构的指针:

```
static struct semid_ds * semary[SEMMNI]
```

信号量集定义了以下主要数据结构:

```

struct sem
{
    int semval;
    int sempid;
};

```

每个信号量占一个 struct sem 数据结构,它有两个成员:semval 若为 0 或正值,表示可用资源数,若为负值则绝对值为等待访问信号量的进程数。缺省为二元信号量,可使用 sys-semctl 变为计数型的。Sempid 存储最后一个操作该信号量的进程的 pid。

```

struct semid_ds
{
    struct ipc_perm sem_perm;

```

```

- kernel - time - t sem - otime;      /* 最后一次操作信号量集合的时间 */
- kernel - time - t sem - ctime;      /* 最后一次更改信号量集合的时间 */
struct sem * sem - base;
struct sem - queue * sem - pending;
struct sem - queue * sem - pending - last;
struct sem - undo * undo;
unsigned short sem - nsems;           /* 信号量集合中的信号量个数 */
* /;

```

semid - ds 数据结构跟踪所有关于单独一个信号量及在它上面所执行的一系列操作的信息。sem - base 指向一个信号量数组,其中包含了多个信号量,该数组信号量总和称“信号量集合”,其容量大小是固定的,由 SEMMSL 决定。sem - pending 是信号量操作的等待队列,仅当操作让进程等待时,这个队列才会增加节点。sem - pending - last 是跟踪挂起的信号量操作队列的队尾,它并不直接指向最后一个节点而是指向最后一个节点的指针。Sem - undo 当各个进程退出时所应该执行的操作的一个队列,下面作进一步介绍。

```

struct sem - queue {                      /* 每个信号量集合占一个 sem - queue 结构
* /
struct sem - queue * next;                /* 队列下一个节点指针 */
struct sem - queue * prev;                /* 队列前一个节点指针 */
struct wait - queue * sleeper;            /* 进程等待信号量队列 */
struct sem - undo * undo;                 /* 由 sops 暗示的撤销操作的操作数组 */
int pid;                                  /* 当前操作的进程 */
int status;                               /* 记录一个阻塞进程被唤醒的过程 */
struct semid - ds * sma;                  /* 一个指向 struct sem - ds 的指针 */
struct sembuf * sops;                     /* 信号量操作队列 */
int nsops;                                /* 信号量操作队列的操作个数 */
int alter;                                /* 说明操作会影响集合中的信号量否 */
};

struct sem - undo { /* 每个 undo 操作占一个 sem - undo 结构 */
struct sem - undo * proc - next;          /* 指向进程的下一个 undo 操作 */
struct sem - undo * id - next;            /* 指向信号量集的下一个 undo 操作 */
int semid;                                /* 信号量集的标识号,代表一个 semid - ds */
short * semadj;                            /* 需调节值的信号量数组,每项表示一个信号量 */
};

```

图 3-17 给出了信号量集的结构。在信号量集上定义的系统调用主要有:semget、semop

和 semctl。Semget 用于创建和打开一个信号量集；semop 用于对信号量用操作值(加或减的数值)进行操作,决定是否阻塞或唤醒；semctl 用于获取、设置或删除某信号量集的信息。每次操作给出三个参数:信号量索引值(信号量数组的索引)、操作值和一组标志。Linux 检查操作是否成功,如果操作值与信号量当前值相加大于 0,或操作值与信号量均为 0,操作均会成功。如果操作失败,仅仅把那些操作标志没有要求系统调用为非阻塞类型的进程挂起,这时必须保存信号量操作的执行状态并将当前进程放入等待队列 sleeper 中,系统还在堆栈上建立 sem - queue 结构并填充各个域,这个 sem - queue 结构利用 sem - pending 和 sem - pending - last 指针放到此信号量等待队列的尾部,然后启动调度程序重新选择其他进程执行。执行释放操作时,Linux 依次检查挂起队列 sem - pending 中的每个成员,看看信号量操作能否继续进行。如果可以继续进行,则将其 sem - queue 结构从挂起链表中删除并对信号量集发出信号量操作,同时还将唤醒处于阻塞状态的进程并使之成为下一个可执行进程。如果挂起队列中有的进程的信号量操作不能完成,系统将在下一次处理信号量时重复这个过程,直到所有信号量操作完成且没有进程需要继续阻塞。

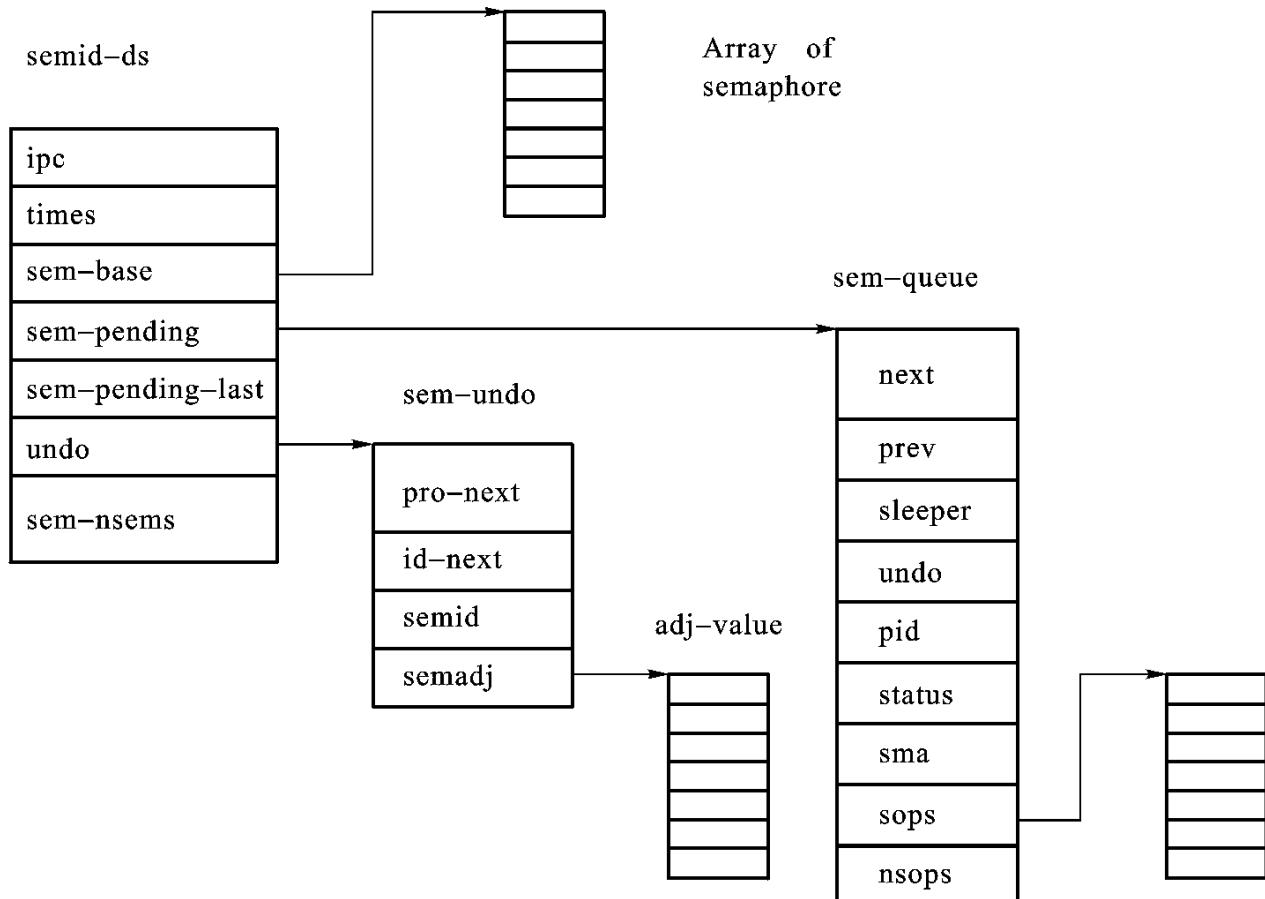


图 3-17 信号量集的结构

为了防止进程错误使用信号量而可能产生的死锁,Linux 使用了 sem - undo 结构,这发

生在当一个进程进入一个临界区而改变了信号量的值,但因进程崩溃或被撤销却没有离开临界区。Linux 通过维护一组描述信号量集变化的链表来进行保护,具体做法是:将信号量设置为进程对其进行操作之前的状态,这些状态值同时保存在使用该信号量数组进程的 semid - ds 和 task - struct 结构的 sem - undo 结构中。每一个单独的信号量操作可以请求被调节,系统为每个进程所对应的每一个信号量集维护一个 sem - undo 结构,如果请求进行信号量操作的进程没有此结构,系统会为其创建一个。当对信号量进行操作时,信号量变化值的负值被加入进程 sem - undo 结构的调节数组的信号量入口。例如,如果操作值为 2,信号量的调节入口加上 -2。进程被删除时,Linux 遍历该进程的 sem - undo 结构并对信号量集使用调节值。如果信号量集已被删除,但 sem - undo 结构还在进程的 task - struct 结构中,则此信号量集标识符视为无效,信号量集清除代码只需丢弃 sem - undo 结构即可。

3.9 本章小结

在第二章中,提出了把操作系统看作接口、资源管理者、虚拟计算机等三种观点来观察操作系统,这些都是静态的观点;这种观点没有把一个进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,在操作系统提供的运行环境中,多个进程共享了同一套计算机系统的资源,进程不可能独立运行,进程之间必然会发生交互和制约关系,操作系统控制进程执行是一个动态过程,在本章中我们提出了根据进程交互的观点来观察操作系统的方法。

讨论从进程的并发性开始,并发性是指系统中存在两个或多个活动,这些活动都已经开始,但又都没有结束,进程的并发执行能有效改善资源利用率和提高系统效率但却使系统更加复杂,因为相交并发进程的执行可能产生与时间有关的错误。在多道程序设计系统中,同一时刻可能有许多进程,这些进程之间存在两种基本关系:竞争关系和协作关系。并发进程可能需要竞争使用资源,进程的互斥是解决进程间竞争关系的手段,为了避免竞争条件,引入了临界区概念以解决进程互斥问题,本章详细介绍了临界区解决方案。为完成同一任务,某些进程需要分工协作,进程的同步是解决进程间协作关系的手段。进程互斥关系是一种特殊的进程同步关系,进程同步的主要任务是使并发协作的进程之间能有效地共享资源和相互协作,从而,使进程的执行过程具有可再现性和执行结果的惟一性。

在尝试使用软件和硬件设施解决进程同步问题后,发现存在许多缺点,软件算法(如著名的 Dekker 算法)过于复杂,硬件指令采用忙式等待而浪费 CPU 时间等等。于是许多新的低级同步机制被开发出来,主要有:Dijkstra 开发的信号量和 P、V 操作,Hansen 和 Hoare 开发的管程。本章介绍了这些同步机制的功能和实现,并且,给出了使用这些工具解决许多经典

同步问题的实例,如生产者与消费者问题、读者与写者问题、五个哲学家吃通心面问题、睡觉的理发师问题,这些问题是在计算机操作系统中并发进程相互制约和内在关系的一种抽象,了解它们就能更好理解操作系统的动态、并发、复杂的本质。每当研究出一种新的同步原语,往往要用经典问题作试金石,看看是否能解决好这些问题。

本章中的另一个重要概念是进程的高级同步机制——进程通信。信号量与P、V操作只能传递信号,没有传递数据的能力,常称之为低级通信机制。有些情况下进程之间交换的信息量虽很少,例如,仅仅交换某个状态信息,但很多情况下进程之间需要交换大批数据,例如,传送一批信息或整个文件,这可以通过一种高级通信机制来完成,进程之间互相交换信息的工作称之为进程通信。进程间通信的方式很多,本章中介绍了多种进程间通信的方式,包括:通过软中断提供的信号通信机制;使用信号量及其原语操作(PV、读写锁、管程)控制的共享存储区通信机制;通过管道提供的共享文件通信机制;以及使用信箱和发信/收信原语的消息传递通信机制。从理论上说,它们都是等价的,每一种原语都可以用另一种来实现,但在实际的系统中,信号量与P、V操作、消息传递及其变种用得最多。

本章最后讨论了死锁问题,无论是相互通信的进程或是共享资源的进程,都可能因推进顺序不当或资源分配不妥而造成系统死锁。死锁是系统中一组并发进程因等待其他进程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。这里讨论了死锁的定义,发生死锁的各类例子,系统产生死锁的四个必要条件:互斥条件、占有并等待条件、不剥夺条件和循环等待条件,以及解决死锁问题的三种策略和方法:死锁的预防、死锁的避免、死锁的检测和解除。死锁的预防是系统预先确定一些资源分配策略,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满足,使得系统不发生死锁,缺点是资源利用率低,或对资源使用的限制过严。死锁的避免涉及到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,对资源的使用放宽了条件。银行家行算法是著名的死锁避免算法,但该算法要预先获得有关信息,很少有进程能够在运行前就知道其所需的最大资源量,而且系统中进程数不是固定的,往往在不断地变化,所以,银行家算法缺乏实用价值。死锁的检测和解除表明操作系统总是同意资源申请,对资源的分配不加任何限制,也允许系统发生死锁,但必须建立一个检测机制选择检测算法(如进程资源分配图、传递闭包等),周期性地检测是否发生了死锁,如果发生了,把它检测出来再采取措施去解除死锁。虽然死锁检测对资源的使用不加任何限制,但造成的开销比较大。

习 题 三

一、思考题

1. 叙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. 叙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. 试说明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?对临界区管理的基本原则是什么?
11. 叙述 Dekker 算法实现临界区互斥的原理。
12. 叙述 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?简述它们的用法。
14. 什么是信号量?如何对它进行分类。
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. 叙述管程中的条件变量有什么含义和作用。
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 操作可用作同步工具,为什么还要有消息传递机制?
22. 叙述信件,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?叙述其工作原理。
26. 试叙述信号通信机制及其实现。
27. 试叙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说明之。
29. 叙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？叙述其基本思想？
32. 解释：进程 – 资源分配图、死锁判定法则、死锁定理。
33. 系统有输入机和行印机各一台，今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？说明理由，若是，则给出一种防止死锁的方法。
34. 假设三个进程共享四个资源，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，证明该系统不会产生死锁。
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，一旦某进程获得了所需的全部数量的资源，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. 五个哲学家就餐问题中，如果至少有一个左撇子或右撇子，则他们的任何就座安排是否可避免死锁，为什么？
37. 五个哲学家就餐问题中，如果至少有一个左撇子或右撇子，则他们的任何就座安排是否可防止饥饿，为什么？
38. 一台计算机有 8 磁带机，它们被 N 个进程竞争使用，每个进程可能需要 3 台。请问 N 为多少时，系统没有死锁危险，并说明原因。
39. 有一个进程还没有获得任何资源，现在它开始申请资源，试问该进程会否进入死锁？通过例子加以说明。
40. 使用上锁法实现进程互斥时，有时会发生饥饿状态，试说明之。
41. 一个系统会处于既不死锁但也不安全状态吗？试分析之。
42. 某系统有同类资源 m 个供 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. 什么叫竞争条件？试解释之。
44. 什么叫忙式等待？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办法。
46. 在信号量机制中，若 P 和 V 操作不是原语，会产生什么问题？
47. 试画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 方 法 |
|------------------|---------------|
| 互 斥 | |
| 占 有 并 等 待 | |
| 不 可 剥 夺 | |
| 循 环 等 待 | $y_4 \hat{u}$ |

二、应用题

1. 有三个并发进程: R 负责从输入设备读入信息块, M 负责对信息块加工处理; P 负责打印输出信息块。今提供:

- 1) 一个缓冲区, 可放置 K 个信息块;
- 2) 二个缓冲区, 每个可放置 K 个信息块;

试用信号量和 P、V 操作写出三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, 各自执行的操作如下, 信号量 S1 和 S2 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: |
|---------------|---------------|
| begin | begin |
| $y := 1;$ | $x := 1;$ |
| $y := y + 3;$ | $x := x + 5;$ |
| $V(S1);$ | $P(S1);$ |
| $z := y + 1;$ | $x := x + y;$ |
| $P(S2);$ | $V(S2);$ |
| $y := z + y$ | $z := z + x;$ |
| end. | End. |

4. 有一阅览室, 读者进入时必须先在一张登记表上登记, 该表为每一座位列出一个表目, 包括座号、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。

5. 在一个盒子里, 混装了数量相等的黑白围棋子。现在用自动分拣系统把黑子、白子分开, 设分拣系统有二个进程 P1 和 P2, 其中 P1 拣白子; 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。试写出两进程 P1 和 P2 能并发正确执行的程序。

6. 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仅使用一个原语操作的同步机制。

7. 设公共汽车上, 司机和售票员的活动分别如下:

司机的活动: 启动车辆; 正常行车; 到站停车。

售票员的活动: 关车门; 售票; 开车门。

在汽车不断地到站、停车、行驶过程中, 这两个活动有什么同步关系? 用信号量和 P、V 操作实现它们的同步。

8. 一个快餐厅有 4 类职员: (1) 领班: 接受顾客点菜; (2) 厨师: 准备顾客的饭菜; (3) 打包工: 将做好的

饭菜打包; (4) 出纳员: 收款并提交食品。每个职员可被看作一个进程, 试用一种同步机制写出能让四类职员正确并发运行的程序。

9. 在信号量 S 上作 P、V 操作时, S 的值发生变化, 当 $S > 0$ 、 $S = 0$ 、 $S < 0$ 时, 它们的物理意义是什么?

10. (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 | Process Q |
|-----------|-----------|
| begin | begin |
| A; | C; |
| B; | D; |
| C; | end; |
| end; | |

(2) 两个并发进程 P1 和 P2 并发执行, 它们的程序分别如下:

| P1 | P2 |
|--------------------|--------------|
| repeat | repeat |
| $k := k \times 2;$ | print $k;$ |
| $k := k + 1;$ | $Sk := 0;$ |
| until false; | until false; |

若令 k 的初值为 5, 让 P1 先执行两个循环, 然后,

P1 和 P2 又并发执行了一个循环, 写出可能的打印值, 指出与时间有关的错误。

11. 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。

12. 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。

13. 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。

14. 使用(1)消息传递, (2)管程, 实现生产者和消费者问题。

15. 试利用记录型信号量和 P、V 操作写出一个不会出现死锁的五个哲学家进餐问题的算法。

16. Dijkstra 临界区软件算法描述如下:

```

var flag:array[0···n] of (idle,want - in,in - cs);
turn:integer;
process Pi(i = 0,1,···,n - 1)
var j:integer;
begin

```

```

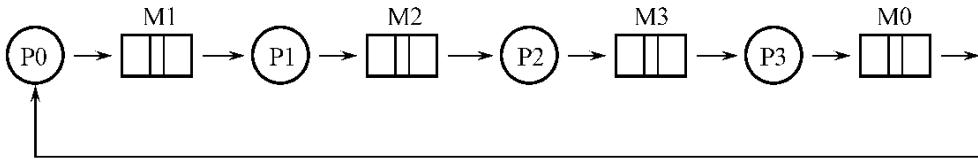
repeat
    flag[i] := want_in;
    while turn ≠ i do
        if flag[turn] == idle then turn := i;
        flag[i] := in_cs;
        j := 0;
        while (j < n) & (j == i or flag[j] ≠ in_cs)
            do j := j + 1;
        until j = n;
        critical section;
        flag[i] := idle;
        ...
    until false;
end

```

试说明该算法满足临界区原则。

17. 另一个经典同步问题:吸烟者问题(patil, 1971)。三个吸烟者在一个房间内,还有一个香烟供应者。为了制造并抽掉香烟,每个吸烟者需要三样东西:烟草、纸和火柴,供应者有丰富货物提供。三个吸烟者中,第一个有自己的烟草,第二个有自己的纸和第三个有自己的火柴。供应者随机地将两样东西放在桌子上,允许一个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,唤醒另一个吸烟者。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。

18. 如图所示,四个进程 P_i ($i = 0 \cdots 3$) 和四个信箱 M_j ($j = 0 \cdots 3$), 进程间借助相邻信箱传递消息,即 P_i 每次从 M_i 中取一条消息,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 个消息。初始状态下, M_0 装了三条消息,其余为空。试以 PV 操作为工具,写出 P_i ($i = 0 \cdots 3$) 的同步工作算法。



19. 设有三组进程 P_i, Q_j, R_k , 其中 P_i, Q_j 构成一对生产者和消费者, 共享一个由 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, 共享一个由 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后成一个并投入 $buf2$, R_k 每次从中取三个产品包装出厂。试用信号量和 P、V 操作写出它们同步工作的程序。

20. 在一个实时系统中,有两个进程 P 和 Q,它们循环工作。P 每隔 1 秒由脉冲寄存器获得输入,并把它累计到整型变量 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它复位。系统提供了标准例程 INPUT 和 OUTPUT 供 I/O, 提供了延时系统调用 Delay(seconds)。试写出两个并发进程循环工作的算法。

21. 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

这类资源时,使系统一定不会发生死锁?

22. 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,所有进程总共的资源需求少于 $M + N$ 个,证明该系统此时不会产生死锁。

23. 一条公路两次横跨运河,两个运河桥相距 100 米,均带有闸门,以供船只通过运河桥。运河和公路的交通均是单方向的。运河上的运输由驳船担负。在一驳船接近吊桥 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 也按同样次序处理。一般典型的驳船长度为 200 米,当它在河上航行时是否会产生死锁?若会,说明理由,请提出一个防止死锁的办法,并用信号量来实现驳船的同步。

24. Jurassic 公园有一个恐龙博物馆和一个花园,有 m 个旅客和 n 辆车,每辆车仅能乘一个旅客。旅客在博物馆逛了一会,然后,排队乘坐旅行车,当一辆车可用时,它载入一个旅客,再绕花园行驶任意长的时间。若 n 辆车都已被旅客乘坐游玩,则想坐车的旅客需要等待。如果一辆车已经空闲,但没有游玩的旅客了,那么,车辆要等待。试用信号量和 P、V 操作同步 m 个旅客和 n 辆车子。

25. 今有 k 个进程,它们的标号依次为 $1, 2, \dots, k$,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需小于 k ,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。

26. 设当前的系统状态如下,系统此时 $\text{Available} = (1, 1, 2)$:

| 进程, | Claim | | | Allocation | | |
|-----|-------|----|----|------------|----|----|
| | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 3 | 2 | 2 | 1 | 0 | 0 |
| P2 | 6 | 1 | 3 | 5 | 1 | 1 |
| P3 | 3 | 1 | 4 | 2 | 1 | 1 |
| P4 | 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) P_1 发出请求向量 $\text{request}_1(1, 0, 1)$,系统能把资源分给它吗?

(4) 若在 P_1 申请资源后,若 P_0 发出请求向量 $\text{request}_0(1, 0, 1)$,系统能把资源分给它吗?

(5) 若在 P_0 申请资源后,若 P_2 发出请求向量 $\text{request}_0(0, 0, 1)$,系统能把资源分给它吗?

27. 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表,试解答下列问题:

| Process | 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 P_1 发出 $\text{request}_1(1, 2, 2, 2)$, 系统能分配资源给它吗? 为什么?

28. 把死锁检测算法用于下面的数据, 并请问:

$$\begin{aligned} \text{Available} &= (1, 0, 2, 0) \\ \text{Need} &= \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \quad \text{Allocation} = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

(1) 此时系统此时处于安全状态吗?

(2) 若第二个进程提出资源请求 $\text{request}_2(0, 0, 1, 0)$, 系统能分配资源给它吗?

(3) 若第五个进程提出资源请求 $\text{request}_5(0, 0, 1, 0)$, 系统能分配资源给它吗?

29. 考虑一个共有 150 个存储单元的系统, 如下分配给三个进程, P_1 最大需求 70, 已占有 25; P_2 最大需求 60, 已占有 40; P_3 最大需求 60, 已占有 45。使用银行家算法, 以确定下面的任何一个请求是否安全。(1) P_4 进程到达, P_4 最大需求 60, 最初请求 25 个。(2) P_4 进程到达, P_4 最大需求 60, 最初请求 35。如果安全, 找出安全序列;如果不安全, 给出结果分配情况。

30. 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 与 Y 的入库过程。

31. 有一个仓库可存放 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。为避免零件锈蚀, 按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。为保证配套和合理库存, 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。试用信号量与 P、V 操作正确地实现它们之间的同步关系。

32. 进程 A_1, A_2, \dots, A_{n1} 通过 m 个缓冲区向进程 B_1, B_2, \dots, B_{n2} 不断地发送消息。发送和接收工作符合以下规则:

- (1) 每个发送进程每次发送一个消息, 写进一个缓冲区, 缓冲区大小与消息长度相等;
- (2) 对每个消息, B_1, B_2, \dots, B_{n2} 都需接收一次, 并读入各自的数据区内;
- (3) 当 M 个缓冲区都满时, 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 PV 操作编制正确控制消息的发送和接收的程序。

33. 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:

\rightarrow 申请 $R_1 \rightarrow$ 申请 $R_2 \rightarrow$ 申请 $R_1 \rightarrow$ 释放 $R_1 \rightarrow$ 释放 $R_2 \rightarrow$ 释放 R_1

(1) 系统运行中可能产生死锁吗? 为什么?

(2) 若可能的话, 请举出一种情况, 并画出表示该死锁状态的进程 - 资源图。

34. 如图所示, 左右两队杂技演员过独木桥, 为了保证安全, 请用 PV 操作和信号量来解决过独木桥问题。只要桥上无人, 则允许一方的人过桥, 待一方的人全部过完后, 另一方的人才允许过桥。



35. 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,而无论是否有读者在读文件。(1)用信号量和 P、V 操作实现;(2)用管程实现。

36. 假定某计算机系统有 R_1 和 R_2 两类可再使用资源(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,并画出死锁点的资源分配图(或称进程 - 资源图)。

37. 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。

38. 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),妈妈向盘子中放桔子(orange),两个儿子专等吃盘子中的桔子,两个女儿专等吃盘子中的苹果。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。

39. 一组生产者进程和一组消费者进程共享九个缓冲区,每个缓冲区可以存放一个整数。生产者进程每次一次性向 3 个缓冲区写入整数,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。

40. 设有三个进程 P、Q、R 共享一个缓冲区, A 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 A 进程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 进程放入的数据并在打印机上打出。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。

41. 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)等三方面讨论它的正确性。如果它是正确的,则证明之;如果它不正确,请说明理由。

```

program attemp;
  var c1,c2:integer;
procedure    p1;                                对第一个进程 p1}
begin
repeat
  Remain Section 1;
  repeat
    c1:= 1 - c2
  until c2 < > 0;

```

```

CriticalSection;           //临界区}
c1 := 1
until false
end;
procedure p2;           //对另一个进程 p2}
begin
repeat
    Remain Section 2;
    repeat
        c2 := 1 - c1
        until c1 < > 0;
    Critical Section;           //临界区}
    c2 := 1
    until false
end;
begin                   //往程序}
    c1 := 1;
    c2 := 1;
    cobegin
        p1;p2           //俩进程 p1, p2 开始执行}
    coend
end.

```

42. 分析下列算法是否正确,为什么?

```

repeat
    key := true;
repeat
    swap(lock, key);
until key = false;
CriticalSection;           //临界区}
lock := false;
other code;
until false;

```

43. 以下并发执行的程序,仅当数据装入寄存器后才能加 1。

```

const n = 50;
var tally :integer;
procedure total()

```

```

var count:integer;
begin
  for count:=1 to n do tally:=tally+1
end;
begin                                     {main program}
tally:=0;
cobegin
  total();total()
coend;
writeln(tally);
end.

```

- (1) 给出该并发程序输出的 tally 值的上限和下限。
(2) 若并发执行的程序可为任意个,这对 tally 值的范围有何影响。

44. 举例说明下列算法不能解决互斥问题。

```

var  balocked:array[0..1] of boolean;
      turn:0..1;
procedure P(id:integer);
begin
  repeat
    blocked[id]:=true;
    while turn≠id do
      begin
        while blocked[1-id] do skip;
        turn:=id;
      end;
    {Critical Section}
    blocked[id]:=false;
    {remainder}
  until false
end;
begin
  blocked[0]:=blocked[1]:=false;
  turn:=0;
  cobegin
    P[0];P[1]
  coend;

```

End.

45. 现有三个生产者 P_1, P_2, P_3 , 他们都要生产桔子水, 每个生产者都已分别购得两种不同原料, 待购得第三种原料后就可配制成桔子水, 装瓶出售。有一供应商能源源不断地供应糖、水、桔子精, 但每次只拿出一种原料放入容器中供给生产者。当容器中有原料时需要该原料的生产者可取走, 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 已购得水和桔子精;

生产者 P_3 已购得糖和桔子精;

试用: 1) 管程, 2) 信号量与 P、V 操作, 写出供应商和三个生产者之间能正确同步的程序。

46. 有一材料保管员, 他保管纸和笔若干。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。任一学生只要能得到其他一种材料就可以写信。有一个可以放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, 每次允许一个学生从中取出自己所需的材料, 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。

47. 进程 A 向缓冲区 buffer 发消息要当发出一消息后要等待进程 B、C、D 都接收这条消息后, 进程 A 才能发新消息。试写出: (1) 用信号量和 P、V 操作, (2) moniter, 写出它们同步工作的程序。

48. 试设计一个管程来实现磁盘调度的电梯调度算法。

49. 有 P_1, P_2, P_3 三个进程共享一个表格 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 F, 但有进程写时, 其他进程不能读和写。用(1)信号量和 P、V 操作, (2)管程编写三进程能正确工作的程序。

50. 现有 100 名毕业生去甲、乙两公司求职, 两公司合用一间接待室, 其中甲公司招收 10 人, 乙公司准备招收 10 人, 招完为止。两公司各有一位人事主管在接待毕业生, 每位人事主管每次只可接待一人, 其他毕业生在接待室外排成一个队伍等待。试用信号量和 P、V 操作实现人员招聘过程。

51. 有一个电子转帐系统共管理 10 000 个帐户, 为了向客户提供快速转帐业务, 有许多并发执行的资金转帐进程, 每个进程读取一行输入, 其中, 含有: 贷方帐号、借方帐号、借贷的款项数。然后, 把款项从贷方帐号划转到借方帐号上, 这样便完成了一笔转帐交易。写出进程调用 Monitor, 以及 Monitor 控制电子资金转帐系统的程序。

52. 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 $2n$ 个学生选课, 规定: (1) 每两个学生分成一组, 并占用一台机器, 协同完成上机实习; (2) 仅当一组两个学生到齐, 并且机房机器有空闲时, 该组学生才能进机房; (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。

53. 某寺庙有小和尚和老和尚各若干人, 水缸一只, 由小和尚提水入缸给老和尚饮用。水缸可容水 10 桶, 水取自同一口水井中。水井径窄, 每次仅能容一只水桶取水, 水桶总数为 3 个。若每次入、取水仅为 1 桶, 而且不可同时进行。试用一种同步工具写出小和尚和老和尚入水、取水的活动过程。

54. 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 ($index = 0, \dots, n - 1$)。当 $free[index] = true$ 时, 表示第 $index$ 个页框空闲, $free[index] = false$ 时, 表示第 $index$ 个页框。试设计

一个管程, 它有两个过程 acquire 和 release 分别负责分配和回收一个页框。

55. AND 型信号量机制是记录型信号量的扩充, 在 P 操作中增加了与条件“AND”, 故称“同时”P 操作和 V 操作, 记为 SP 和 SV(Simultaneous P 和 V)于是 $SP(s_1, s_2, \dots, s_n)$ 和 $SV(s_1, s_2, \dots, s_n)$ 其定义为如下的原语操作:

```

procedure SP(var s1, ..., sn: semaphore)
begin
  if s1 > = 1 & ... & sn > = 1 then begin
    for i := 1 to n do
      si := si - 1;
  end
  else begin
    进程进入第一个遇到的满足 si < 1 条件的 si 信号量队列等待,
    同时将该进程的程序计数器地址回退, 置为 SP 操作处。};
  end
procedure VP(var s1, ..., sn: semaphore)
begin
  for i := 1 to n do begin
    si := si + 1;
  从所有 si 信号量等待队列中移出进程并置入就绪队列。};
end

```

试回答 AND 信号量机制的主要特点, 适用于什么场合?

56. 试用 AND 型信号量和 SP、SV 操作解决生产者 – 消费者问题。

57. 试用 AND 型信号量和 SP、SV 操作解决五个哲学家吃通心面问题。

58. 如果 AND 型信号量 SP 中, 并不把等待进程的程序计数器地址回退, 亦即保持不变, 则应该对 AND 型信号量 SV 操作做何种修改?

59. 一般型信号量机制(参见汤子瀛等编著的计算机操作系统, 西安电子科技大学出版社)

对 AND 型信号量机制作扩充, 便形成了一般型信号量机制, $SP(s_1, t_1, d_1, \dots, s_n, t_n, d_n)$ 和 $SV(s_1, d_1, \dots, s_n, d_n)$ 的定义如下:

```

procedure SP(s1, t1, d1; ...; sn, tn, dn)
var s1, ..., sn: semaphore;
     t1, ..., tn: integer;
     d1, ..., dn: integer;
begin
  if s1 > = t1 & ... & sn > = tn then begin
    for i := 1 to n do
      si := si - di;
  end

```

```

end
else
进程进入第一个遇到的满足  $s_i < t_i$  条件的  $s_i$  信号量队列等待,
同时将该进程的程序计数器地址回退, 置为 SP 操作处。};

end
end
procedure SV( $s_1, d_1; \dots s_n, d_n$ )
var  $s_1, \dots s_n$ : semaphore;
 $d_1, \dots d_n$ : integer;

begin
for i := 1 to n do begin
 $s_i := s_i + d_i$ ;
从所有  $s_i$  信号量等待队列中移出进程并置入就绪队列。};

end
end

```

其中, t_i 为这类临界资源的阀值, d_i 为这类临界资源的本次请求数。

试回答一般型信号量机制的主要特点, 适用于什么场合?

60. 下面是一般信号量的一些特殊情况:

- $SP(s, d, d)$
- $SP(s, 1, 1)$
- $SP(s, 1, 0)$

试解释它们的物理含义或所起的作用。

61. 试利用一般信号量机制解决读者一写者问题。

第四章 存 储 管 理

存储管理是操作系统的重要组成部分,它负责管理计算机系统的重要资源——主存储器。由于任何程序及数据必须占用主存空间后才能执行,因此,存储管理的优劣直接影响系统的性能。主存储空间一般分为两部分:一部分是系统区,存放操作系统核心程序以及标准子程序,例行程序等;另一部分是用户区,存放用户的程序和数据等,供当前正在执行的应用程序使用。存储管理主要是对主存储器中的用户区域进行管理,当然,也包括对辅存储器的管理。尽管现代计算机中主存的容量不断增大,已达到 GB 级的范围,但仍然不能保证有足够的空间来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要尽可能地方便用户使用和提高主存储器的利用效率,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,存储管理有下面几个方面的功能:

- 主存储空间的分配和去配。
- 地址转换和存储保护。
- 主存储空间的共享。
- 主存储空间的扩充。

本章在简介计算机存储器的层次之后,先后分析了多种连续存储管理方法以及页式和段式存储管理方法,并进一步讨论了虚拟存储管理系统,最后介绍了 Intel 的 Pentium 芯片存储管理支持,Windows 2000/XP 虚拟存储管理及 Linux 的虚拟存储管理实例。

4.1 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可以分为寄存器、高速缓存、主存储器、磁盘缓存、固定磁盘、可移动存储介质等 7 层来组成层次结构。如图 4-1 所示,越往上,存储介质的访问速度越快,价格也越高。其中,寄存器、高速缓存、主存储器和磁盘缓存均属于操作系统存储管理的管辖范畴,掉电后它们存储的信息不再存在。固定磁盘和可移动存储介质属于设备管理的管辖范畴,它们存储的信息将被长期保存。而磁盘

缓存本身并不是一种实际存在的存储介质, 它依托于固定磁盘, 提供对主存储器存储空间的扩充。

可执行的程序必须被保存在计算机的主存储器中, 与外围设备交换的信息一般也依托于主存储器地址空间。由于处理器在执行指令时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最昂贵的存储器, 它的容量小, 一般以字(word)为单位。一个计算机系统可能包括几十个甚至上百个寄存器, 用于加速存储访问速度, 如用寄存器存放操作数, 或用作地址寄存器加快地址转换速度。

高速缓存的容量稍大, 其访问速度快于主存储器, 利用它存放主存中一些经常访问的信息可以大幅度提高程序执行速度。例如, 主存访问速度为 $1 \mu\text{s}$, 高速缓存为 $0.1 \mu\text{s}$, 假使访问信息在高速缓存中的几率为 50%, 那么, 存储器访问速度可以提高到 $0.55 \mu\text{s}$ 。假设, 某台计算机的存储器层次及其配置如下: CPU 中的寄存器 100 个字; 高速缓存 512 KB, 存取周期 15 ns; 主存储器 128 MB, 存取周期 60ns; 磁盘容量 40 GB, 存取周期毫秒级; 后援存储容量 1TB, 存取周期秒级。则这台机器足可用于中小型科研项目的开发, 多层次组成的存储体系十分有效与可靠, 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着顺序性、局部性、循环性和排他性, 因此, 在程序执行时有时并不需要把程序和数据全部调入内存, 而只需先调入一部分, 待需要时再逐步调入。这样, 计算机系统为了容纳更多的算题数, 或是为了处理更大批量的数据, 就可以在磁盘上建立磁盘缓存以扩充主存储器的存储空间。算题的程序和处理的数据可以装入磁盘缓存, 操作系统自动实现主存储器和磁盘缓存之间的数据的调进调出, 从而, 向用户提供了比实际主存存储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器子系统。

4.1.2 快速缓存

快速缓存(caching)是现代计算机结构中的一个重要部件。通常, 运算的信息存放在主存中, 每当使用它时, 被临时复制到一个速度较快的 cache 中。当 CPU 访问一组特定信息时, 首先, 检查它是否在 cache 中, 如果已存在, 可直接从中取出使用; 否则, 要从主存中读出信息。通常认为这批信息被再次用到的概率很高, 所以, 同时还把主存中读出的信息复制到 cache 中。在 CPU 的内部寄存器和主存之间建立了一个 cache, 而由程序员或编译系统实现寄存器的分配或替换算法, 以决定信息是保存在寄存器中还是在主存中。有一些 cache 是硬件实现的, 如大多数计算机有指令 cache, 用来暂存下一条欲执行的指令。如果没有指令

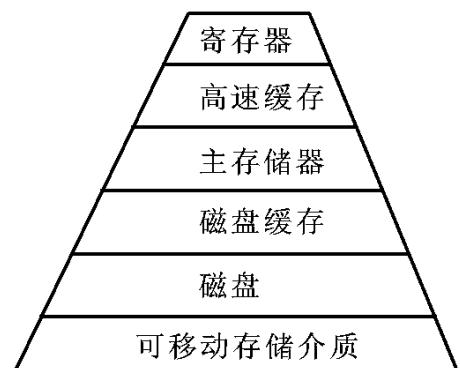


图 4-1 计算机系统存储器层次

cache, CPU 将会空等若干个周期, 直到下一条指令从主存中取出。同样道理, 多数计算机系统的存储层次中设置了一个或多个高速数据 cache。可见 cache 是解决主存速度与 CPU 速度不相匹配的一种部件。

由于高速 cache 的容量有限(速度小于 25 ns, 容量有 128 KB 和 256 KB 等), 所以, cache 的管理是一个重要的设计问题。仔细确定其大小和替换策略, 能够做到 80% – 99% 的所需信息在 cache 中找到, 因而, 系统性能极高。主存也可以看作是辅存的 cache, 因为, 辅存中的数据必须复制到主存方能使用; 反之, 数据也必须先存在主存中, 才能输出到辅存。一个文件的数据可能出现在存储器层次的不同级别中, 例如, 操作系统可在主存中维护一个文件数据的 cache。大容量的辅存常常使用磁盘, 磁盘数据经常备份到磁带或可移动磁盘组上, 以防止硬盘故障时丢失数据。有些系统自动地把老文件数据从辅存转存到海量存储器中, 如磁带上, 这样做还能降低存储价格。

下面来讨论采用 cache 后数据的一致性问题。在层次式存储结构中, 相同的数据可能出现在不同的层次上。例如, 考虑对文件 B 中的一个整数 A 做加 1 操作, 假如文件 B 储存在磁盘上。首先, 发出 I/O 指令, 它把 A 所在的盘块读到主存中, 再做加法操作。跟着这个操作, 可能把 A 的一个副本复制到 cache, 而 A 的另一个副本复制到 CPU 寄存器。因此, A 的副本出现在层次的若干个地方, 当加 1 操作在 CPU 寄存器执行之后, 在存储器各层次中, A 的值是不同的, 仅当 A 的新值被写回磁盘之后, A 的值才会变成相同。

在每次仅有一个进程执行的环境中, 上述安排没有什么问题, 因为, 对整数 A 的存取总是得到存储层次中最高层的值。然而, 在一个多任务环境中, CPU 将在许多进程之间来回切换, 如果多个进程需要存取 A, 那么, 每个进程应获得 A 的最新被修改过的值。

多 CPU 环境中, 情况变得较为复杂, 除了维护 CPU 寄存器外, CPU 还包含一个局部数据 cache, 于是, A 的副本可能同时出现在若干个 cache 中。所有的 CPU 都并发地执行, 对 A 值在某个 cache 中的修改, 立即会影响到 A 所有的所有其他 cache, 这个问题称作缓冲一致性(coherency)问题, 这种一致性通常由硬件来保证。

在分布式环境中, 情况变得尤其复杂, 同一个文件的不同副本可能存储在地理上分散的不同计算机中。因此, 某个副本在一个地方被修改, 所有其他副本立刻就成为过时的了。有许多途径能确保文件一致性, 这些将在分布式系统中讨论。

4.1.3 地址转换与存储保护

用户作业的程序通常用高级语言编写, 称为源程序。但源程序是不能被计算机直接执行的, 需要通过编译程序或汇编程序编译或汇编来获得目标程序。目标程序的地址不是内存的实际地址, 把用户目标程序使用的地址单元称为逻辑地址(相对地址), 一个用户作业的目标程序的逻辑地址集合称为该作业的逻辑地址空间。作业的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;也可以是二维的,这时整个用户作业被分为若干段,每段有不同的段号,段内地址从 0 开始。当程序运行时,它将被装入主存储器地址空间的某些部分,此时程序和数据的实际地址一般不可能同原来的逻辑地址一致。把主存中的实际存储单元称为物理地址(绝对地址),物理地址的总体相应构成了用户程序实际运行的物理地址空间。注意,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。

为了保证程序的正确运行,必须把程序和数据的逻辑地址转换为物理地址,这一工作称为地址转换或重定位。地址转换有两种方式,一种方式是在作业装入时由作业装入程序(装配程序)实现地址转换,称为静态重定位;这种方式要求目标程序使用相对地址,地址变换在作业执行前一次完成;另一种方式是在程序执行过程中,CPU 访问程序和数据之前实现地址转换,称为动态重定位。在多道程序系统中,可用的主存空间常常被许多进程共享,程序员编程时事先不可能知道程序执行时的驻留位置,而且必须允许程序因对换或空区收集而被移动,这些现象都需要程序的动态重定位,动态重定位必须借助于硬件的地址转换机构实现。

在计算机系统中可能同时存在操作系统程序和多道用户程序,操作系统程序和各个用户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工作区而不能互相干扰,因此,操作系统必须对主存中的程序和数据进行保护,以免其他程序有意或无意的破坏,这一工作称为存储保护。计算机中使用的存储保护硬件主要有:界地址和存储键方式等,将在本章详细介绍。

无论是地址转换还是存储保护,都必须借助于前面提到的地址寄存器以及一些硬件线路。用软件来模拟实现地址转换或存储保护都是不可行的。因为,每一条命令都可能牵涉到地址转换和存储保护,模拟的结果将使得每一条指令的执行代价升级为一段程序的执行代价。

4.2 连续存储空间管理

4.2.1 单用户连续存储管理

单用户连续存储管理又称单分区模式,适用于单用户的情况,个人计算机和专用计算机系统可采用这种存储管理方式。采用单连续存储管理时主存分配十分简单,主存空间分为系统区和用户区,系统区存放操作系统驻留代码和数据,用户区全部归一个用户作业所占用。在这种管理方式下,任一时刻主存储器中最多只有一道程序,各个作业的程序只能按次

序一个个地装入主存储器运行。

单用户连续存储管理的地址转换多采用静态重定位。程序执行之前由装入程序完成逻辑地址到物理地址的转换工作。如图 4-2 所示。具体来说, 可设置一个栅栏寄存器(fence register)用来指出主存中的系统区和用户区的地址界限, 通过装入程序把程序装入到从界限地址开始的区域。由于用户是按逻辑地址(相对地址)来编排程序的, 所以, 当程序被装入主存时, 装入程序必须对它的指令和数据进行重定位。存储保护也是很容易实现的, 由装入程序检查其绝对地址是否超过栅栏地址。若是, 则可以装入; 否则, 产生地址错误而不能装入, 于是一个被装入的程序执行时, 总是在它自己的区域内进行, 而不会破坏系统区的信息。采用静态重定位的主要优点是实现简单, 无需硬件地址变换机构支持。缺点是作业只能分配到一个连续存储区域中, 程序执行期间不能在主存中移动, 无法实现程序共享。

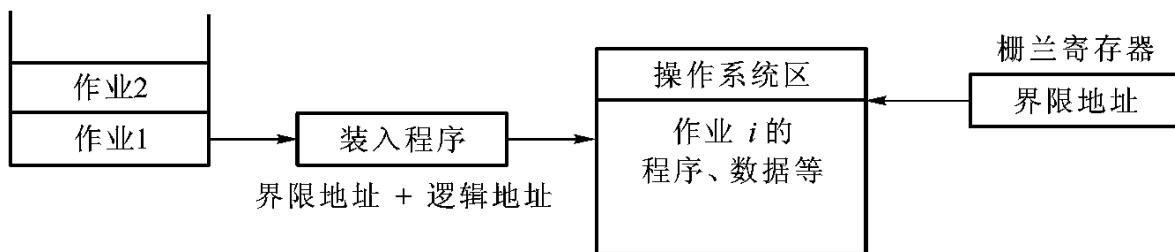


图 4-2 采用静态重定位的单连续存储管理

单用户连续存储管理的地址转换也可以采用动态重定位。程序执行过程中, 在 CPU 访问主存前, 把要访问的程序和数据的逻辑地址转换成物理地址。如图 4-3 所示。具体来说, 可设置一个定位寄存器, 它既用来指出主存中的系统区和用户区的地址界限, 又作为用户的基址。然后通过装入程序把程序装入到从界限地址开始的区域, 但此时并不进行地址转换, 程序执行过程中动态地将逻辑地址与定位寄存器中的值相加就可得到绝对地址。存储保护的实现很容易, 程序执行中由硬件的地址转换机构根据逻辑地址和定位寄存器的值产生绝对地址, 且检查该绝对地址是否存在所分配的存储区域内, 若超出所分配的区域, 则产生地址错误, 不允许访问该单元中的信息。

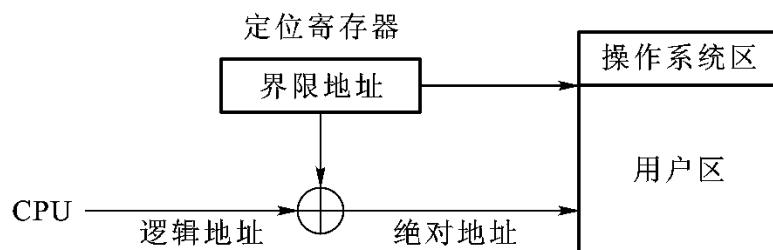


图 4-3 采用动态重定位的单连续存储管理

动态重定位的优点是使内存的使用灵活有效, 用户作业不限制分配在一个连续存储区域中, 容易实现内存的动态扩充和共享。缺点是需要附加硬件支持, 管理软件也较为复杂。

由于单用户连续存储管理只适合单道程序系统, 采用这种存储管理方法有几个主要缺点:

- 正在执行的程序因等待某个事件, (比如, 等待从外部设备输入数据时) 处理器便处于空闲状态。

- 不管用户作业的程序和数据量的多少, 都是一个作业独占主存储空间, 这就可能降低存储空间的利用率。

- 计算机的外围设备利用率不高。

在 70 年代由于小型计算机和微型计算机的主存容量不大, 所以, 单用户连续存储管理曾得到了广泛的应用。IBM7094 的 FORTRAN 监督系统, IBM1130 磁盘监督系统, MIT 兼容分时系统 CISS 以及微型计算机 cromemco 的 CDOS 系统, Digital Research 和 Dyhabyte 的 CP/M 系统, DJS0520 的 0520FDOS 等等均采用单用户连续存储管理。

4.2.2 固定分区存储管理

分区存储管理的基本思想是给进入主存的用户作业划分一块连续存储区域, 把作业装入该连续存储区域, 若有多个作业装入主存, 则它们可并发执行, 这是能满足多道程序设计需要的最简单的存储管理技术。

固定分区(fixed partition)存储管理又称定长分区或静态分区模式, 是静态地把可分配的主存储器空间分割成若干个连续区域。每个区域的位置固定, 但大小可以相同也可以不同, 每个分区在任何时刻只装入一道程序执行。早期的 IBM 操作系统 OS/MFT(Multiprogramming with a Fixed Number of Tasks) 使用了固定分区存储管理。为了说明各分区的分配和使用情况, 存储管理需设置一张“主存分配表”, 用以记录主存中划分的分区和分区的使用情况, 该表具有如图 4-4 所示的模式。

| 分区号 | 起始地址 | 长度 | 占用标志 |
|-----|------|-----|------|
| 1 | 8K | 8K | 0 |
| 2 | 16K | 16K | Job1 |
| 3 | 32K | 16K | 0 |
| 4 | 48K | 64K | 0 |
| 5 | 64K | 32K | Job2 |
| 6 | 96K | 32K | 0 |

图 4-4 固定分区存储管理的主存分配表

主存分配表指出各分区的起始地址和长度, 表中的占用标志位用来指示该分区是否被

占用了,当占用的标志位为“0”时,表示该分区尚未被占用。进行主存分配时总是选择那些标志为“0”的分区,当某一分区分配给一个长度要求小于等于分区长的作业后,则在占用标志栏填上占用该分区的作业名。在图 4-4 中,第 2、5 分区分别被作业 Job1 和 Job2 占用,而其余分区为空闲。当分区中的一个程序执行结束退出系统时,相应的分区占用标志位被置“0”,其占用的分区又变成空闲,可被其他程序使用。

由于固定分区存储管理是预先将主存分割成若干个连续区域,分割时各区在主存分配表中可按地址顺序排列,如图 4-5,那么,固定分区存储管理的主存分配算法十分简单,有兴趣的同学可以把它作为课后练习。



图 4-5 固定分区存储管理

固定分区存储管理的地址转换可以采用静态定位方式,装入程序在进行地址转换时检查其绝对地址是否落在指定的分区中,若是,则可把程序装入,否则不能装入,且应归还所分得的存储区域。固定分区方式的主存去配很简单,只需将主存分配表中相应分区的占用标志位置成“0”即可。

固定分区存储管理的地址转换也可以采用动态定位方式。如图 4-6 所示,系统专门设置一对地址寄存器——上限/下限寄存器;当一个进程占有 CPU 执行时,操作系统就从主存分配表中取出其相应的地址和长度,换算后置入上限/下限寄存器;硬件的地址转换机构根据下限寄存器中保存的基地址 B 与逻辑地址得到绝对地址;硬件的地址转换机构同时把绝对地址和上限/下限寄存器中保存的相应地址进行比较,而实现存储保护。

固定分区的一个任务是何时和如何把主存空间划分成分区?这个任务通常由系统操作员和操作系统初始化模块协同完成,系统开机初启时,系统操作员根据当天作业情况把主存划分成大小可以不等但位置固定的分区。

作业进入分区时有两种作业排队策略:一是每个作业被调度程序选中时就排到一个能够装入它的最小分区的等待处理队列中。如果等待处理的作业大小很不均匀,会导致有的分区空闲而有的分区忙碌;二是所有等待处理的作业排成一个队列,当调度其中一个进入分区运行时,选择可容纳它的最小可用分区,以充分利用主存。

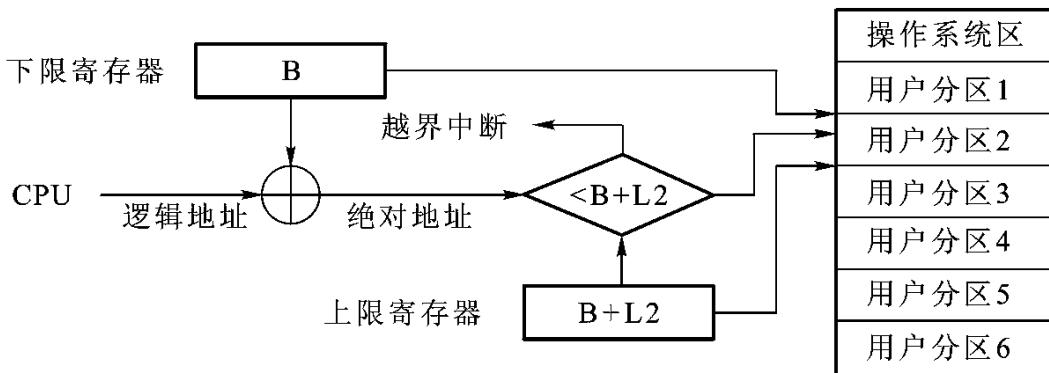


图 4-6 固定分区存储管理的地址转

固定分区的提出能解决单道程序运行在并发环境下不能与 CPU 速度很好匹配的问题，同时也解决了单道程序运行主存空间利用率低的问题。采用固定分区存储管理有许多缺点，首先，由于预先规定了分区大小，使得大程序无法装入，用户不得不采用覆盖等技术补救，不但加重用户负担，而且极不方便。其次，主存空间的利用率不高，往往一个作业不可能恰好填满分区。例如，图 4-5 中若 Job1 和 Job2 两个作业实际只是 10K 和 18K 的主存，但它们却占用了 16K 和 32K 的区域，共有 20K 的主存区域占而不用被白白浪费了。再者，如果一个作业运行中要求动态扩充主存，采用固定分区是困难的；各分区作业要共享程序和数据也难实现。最后，因为分区的数目是在系统初启时确定的，这就限制了多道运行的程序数。然而，这种方法实现简单，因此，对于程序大小和出现频率已知的情形，还是较合适的。

4.2.3 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)存储管理又称变长分区模式，是按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。系统在作业装入主存执行之前并不建立分区，当要装入一个作业时，再根据作业需要的主存量查看主存中是否有足够的空间。若有，则按需要量分割一个分区分配给该作业；若无，则令该作业等待主存空间。由于分区的大小是按作业的实际需要量来定的，且分区的个数也是可变的，所以，可变分区方式可以克服固定分区方式中的主存空间的浪费，有利于多道程序设计，实现了多个作业对主存的共享，进一步提高了主存资源利用率。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，系统初启后但用户作业尚未装入主存之前，整个用户区是一个大空闲分区。随着作业的装入、撤离，主存空间被分成许多个分区，有的分区被作业占用，而有的分区是空闲的。当一个新的作业要求装入时，必须找一个足够大的空闲区，如果找到的空闲

区大于作业需要量，则作业装入前把这个空闲区分成两部分，一部分分配给作业使用，另一部分又分成为一个较小的空闲区。当一个作业运行结束撤离时，它归还的区域如果与其他空闲区相邻，则应该合成一个较大的空闲区，方便大作业的装入。采用可变分区方式的主存分配示例如图 4-7。

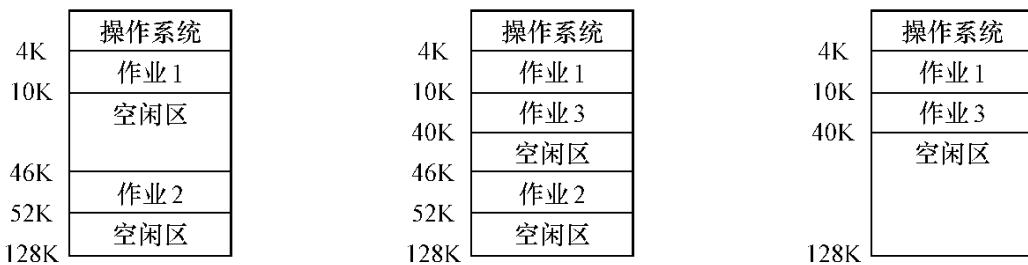


图 4-7 可变分区存储管理的主存分配

从图中可以看出，主存中分区的数目和大小随作业的执行而不断改变。为了方便主存的分配和去配，主存分配表可由两张表格组成，一张称“已分配区表”，另一张是“未分配区表”，如图 4-8。

| 分区号 | 起始地址 | 长度 | 标志 |
|-----|------|-----|------|
| 1 | 4 K | 6 K | Job1 |
| 2 | 46 K | 6 K | Job2 |
| | | | |

(a) 已分配区表

| 分区号 | 起始地址 | 长度 | 标志 |
|-----|------|------|-----|
| 1 | 10 K | 36 K | 未分配 |
| 2 | 52 K | 76 K | 未分配 |

(b) 未分配区表

图 4-8 可分区主存分配表

上面两张表的内容是按图 4-7 最左边的情况填写的，当要装入长度为 30 K 的作业时，从未分配表中可找一个足够容纳它的长度 36 K 的空闲区，将该区分成两部分，一部分为 30 K，用来装入作业 3，成为已分配区；另一部分为 6 K，仍是空闲区。这时，应从已分配区表中找一个空栏目登记作业 3 占用的起址、长度，同时修改未分配区表中空闲区的长度和起址。当作业撤离时则已分配区表中的相应状态改成“空”，而将收回的分区登记到未分配表中，若有相邻空闲区则将其连成一片后登记。可变分区的回收算法较为复杂，当一个进程 X 撤离时，可分成以下四种情况：其邻近都有进程（A 和 B），一边有进程（A 或 B），两边均为空

闲区(黑色区域)。回收后如下图 4-9 右边所示,同时应修改主存分配表或链表。

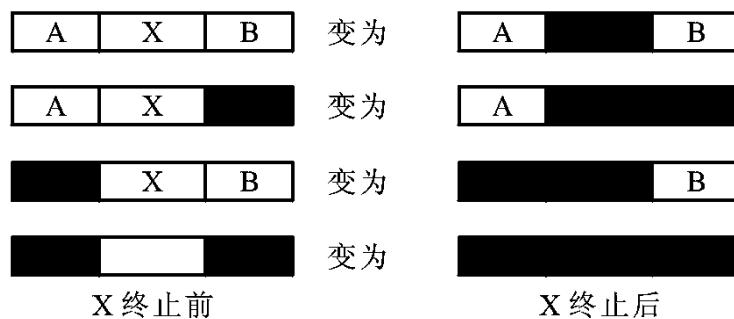


图 4-9 可变分区回收情况

由于分区的个数不定,采用链表是较好的另一种空闲区管理方法。每个内存空闲区的开头单元存放本空闲区长度及下一个空闲区的起始地址指针,于是所有的空闲区都被链接起来,系统设置一个第一块空闲区地址指针,让它指向第一块空闲区的地址。使用时,沿链查找并摘下一个长度能满足申请要求的空闲区交给进程,再修改链表;归还时,把空闲区链入空闲区链表即可。空闲区链表管理比空闲区表格管理较为复杂,但其优点是链表自身不占用存储单元。

不论是空闲区链表管理还是空闲区表格管理,链和表中的空闲区都可按一定规则排列,例如,按空闲区从大到小排或从小到大排,以方便空闲区的查找和回收。

常用的可变分区管理的分配算法有五种:

(1) 最先适应(first fit)分配算法。每次分配时,总是从未分配区表头顺序查找未分配表或链表,找到第一个能满足长度要求的空闲区为止。分割这个找到的未分配区,一部分分配给作业,另一部分仍为空闲区。

这种分配算法优先利用主存低地址空闲分区,从而,保留了高地址的大的空闲区。但由于低地址空闲分区不断被分割,既可能将大的空间分割掉,也造成低地址部分有较多难以使用的“碎片”。作为改进,可把空闲区按地址从小到大排列在未分配表或链表中。因为,为作业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,以保持一个大的空闲区,有利于大作业的装入。但是,这给回收分区带来一些麻烦,每次收回一个分区后,必须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(next fit)分配算法。每次分配时,总是从未分配区上次扫描结束处顺序查找未分配表或链表,找到第一个能满足长度要求的空闲区为止。分割这个找到的未分配区,一部分分配给作业,另一部分仍为空闲区。这一算法是最先适应分配算法的一个变种,能使得存储空间的利用率更加均衡,不会导致小的空闲区集中在存储器的一端。

(3) 最优适应(best fit)分配算法。另一种分配算法要扫描整个未分配区表或链表,从空闲区中挑选一个能满足作业要求的最小分区进行分配。这种算法可保证不去分割一个更大

的区域,使装入大作业时比较容易得到满足。采用这种分配算法时可把空闲区按长度以递增顺利排列,查找时总是从最小的一个区开始,直到找到一个满足要求的分区为止。按这种方法,在回收一个分区时也必须对分配表或链表重新排列。最优适应分配算法找出的分区如果正好满足要求则是最合适的了,如果比所要求的略大则分割后使剩下的空闲区就很小,以致无法使用。

(4) 最坏适应(worst fit)分配算法。最坏适应分配算法要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,其优点是可使剩下的空闲区不至于太小,对中、小作业有利。采用这种分配算法时可把空闲区按长度以递减顺序排列,查找时只要看第一个分区能否满足作业要求,这样使最坏适应分配算法查找效率很高。

(5) 快速适应(quick fit)算法。它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,该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 这样的空闲区既可以放在 8 KB 的链表中也可以放在一个特殊的空闲区链表中。该算法查找十分快速,只要按用户程序长度找到能容纳它的最小空闲区链表并取下第一块分配。其缺点与其他分配算法相似,归还主存时与相邻空闲区的合并既复杂又费时。

下面简单地比较一下五种查找和分配算法。从搜索空闲区速度及主存利用率来看,最先适应分配算法、下次适应分配算法和最佳适应算法比最坏适应算法性能好。如果空闲区按从小到大排列,则最先适应分配算法等于最优适应分配算法。反之,如果空闲区按从大到小排列,则最先适应分配算法等于最坏适应分配算法。空闲区按从小到大排列时,最先适应分配算法能尽可能使用低地址区,从而,在高地址空间有较多较大的空闲区来容纳大的作业。下次适应分配算法会使存储器空间得到均衡使用。最优适应分配算法的主存利用率最好,因为,它把刚好或最接近申请要求的空闲区分给作业但是它可能会导致空闲区分割下来的部分很小。在处理某种作业序列时,最坏适应分配算法可能性能最佳,因为,它选择最大空闲区,使得分配后剩余下来的空闲区不会太小,仍能用于再分配。由于最先适应算法简单、快速,在实际的操作系统中用得较多;其次是最佳适应算法和下次适应算法。

2. 地址转换与存储保护

对可变分区方式采用动态重定位装入作业,作业程序和数据的地址转换是硬件完成的。硬件设置两个专门控制寄存器:基址寄存器和限长寄存器。基址寄存器存放分配给作业使用的分区的起始地址,限长寄存器存放作业占用的连续存储空间的长度。

当作业占有 CPU 运行后,操作系统可把该区的始址和长度送入基址寄存器和限长寄存器,启动作业执行时由硬件根据基址寄存器进行地址转换得到绝对地址,地址转换如图 4-10 所示。

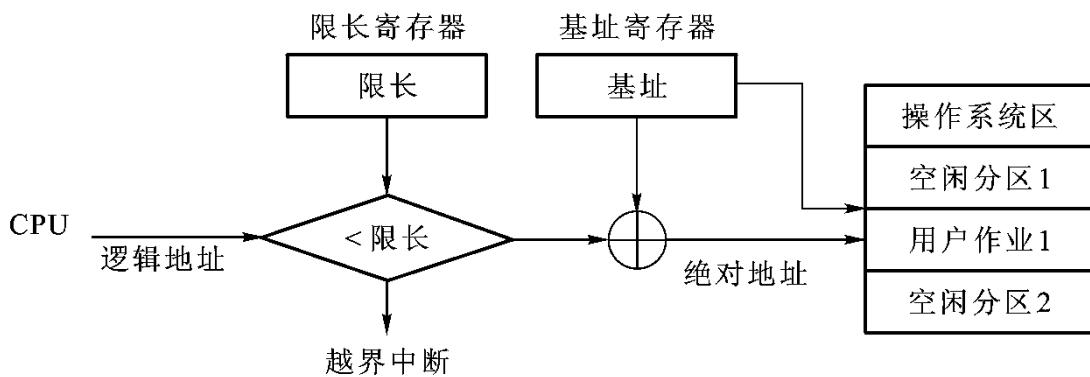


图 4-10 可变分区存储管理的地址转换和存储保护

当逻辑地址小于限长值时，则逻辑地址加基址寄存器值就可获得绝对值地址；当逻辑地址大于限长值时，表示作业欲访问的地址超出了所分得的区域，这时，不允许访问，达到了保护的目的。

在多道程序系统中，硬件只需设置一对基址/限长寄存器，作业相应的进程在执行过程中出现等待时，操作系统把基址/限长寄存器的内容随同该进程的其他信息，如 PSW，通用寄存器等一起保存起来。当作业相应的进程被选中执行时，则把选中作业的基址/限长值再送入基址/限长寄存器。世界上最早的巨型机 CDC6600 便采用这一方案。

如果每个作业只能占用一个分区，那么，就不允许各个作业之间有公共的区域，这样，当几个作业共享一个例行程序时就只好在各自的主存区域各放一套，从而，主存利用率差。所以，有些计算机硬件提供多对基址/限长寄存器，允许一个作业占用两个或多个分区。可以规定某对基址/限长寄存器的区域是共享的，用来存放共享的程序和常数，当然对共享区域的信息只能读出不能写入。于是几个作业共享的例行程序就可放在限定的公用区域中，而只要让作业的共享部分具有相同的基址/限长值就行了。

3. 移动技术

当在未分配表中找不到一个足够大的空闲区来装入作业时，可采用移动技术把在主存中的作业改变存放区域，同时修改它们的基址/限长值，从而，使分散的空闲区汇集成一片而有利于作业的装入。移动分配的示例如图 4-11。

移动虽可汇集主存的空闲区，但也增加了系统的开销，而且不是任何时候都能对一道程序进行移动的。由于外围设备（块设备）与主存储器交换信息时，通道总是按已经确定的主存绝对地址完成信息传输，所以，当一道程序正在与外围设备交换数据时往往不能移动，故应尽量设法减少移动。比如，当要装入一道作业时总是先挑选不经移动就可装入的作业；在不得不移动时力求移动的道数最少。采用移动技术分配主存的算法如下：

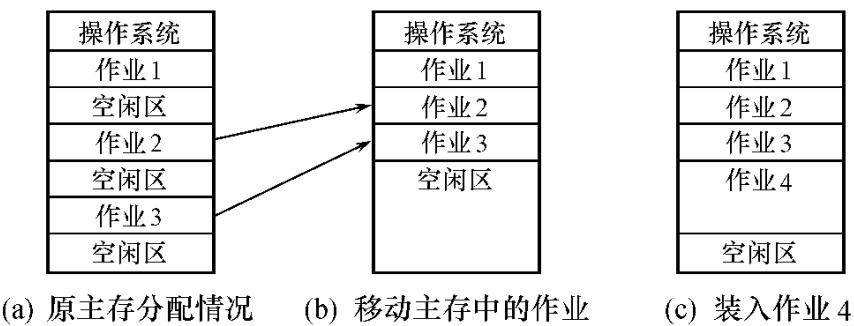


图 4-11 移动分配示例

作业 i 请求分配 x K 主存

- 步骤 1 查主存分配表；
- 步骤 2 若有大于 x K 主存的空闲区，则转步骤 5；
- 步骤 3 若空闲区总和小于 x K，则作业 i 等待主存资源；
- 步骤 4 移动主存中有关信息；
修改主存分配表中有关项；
修改被移动者的基址/限长；
- 步骤 5 分配 x K 主存；
修改主存分配表中有关项；
置作业 i 的基址/限长；
- 步骤 6 算法结束。

移动也为作业执行过程中动态扩充主存提供了方便。一道作业在执行过程中要求增加主存量时, 只需适当移动邻近的作业就可增加它所占的连续区的长度, 移动后的基址值和扩大后的限长值都应作相应的修改。

在多道程序系统中, 如果允许作业在执行过程中动态扩充主存, 那么, 有时会遇到死锁问题。例如, 主存已被 A、B 两道作业全部占用了, 当 A 申请主存时, 由于空闲区不够不能继续执行下去而处于等待主存资源状态; 以后, B 占有处理器执行, 执行中也要申请主存, 同样, 它也变成等待主存资源状态。显然, A、B 的等待永远不能结束, 这就是死锁。为了避免死锁, 可以考虑发生 A、B 竞争主存资源而都得不到满足的情形, 操作系统将采用剥夺资源的办法, 把 A 或 B 送出主存, 存到辅助存储器中; 然后, 让留在主存的那道作业获得主存资源并继续执行下去, 直到它归还主存后, 再将送出去的作业调回来。

为了接纳送出去的作业, 辅助存储器中要留下一个暂存区域称映像区。为使这个区域小一些, 可以考虑每次总是将最小的一道作业从主存送出去。假定主存中最多容纳 n 道作

业,主存能提供给一道作业的最大容量为 V (一般说来, V 为主存用户区域的容量),那么,送出去的作业的辅助存储器区域大小为:

$$(1/2 + 1/3 + \dots + 1/n) \cdot V$$

就足够了。因为,当主存中有 n 道作业时,送出去的那道所占主存量一定不超过 $1/n \times V$ 。而最坏的情况是主存中开始有 n 道,发生竞争主存后送出一道,剩下 $n - 1$ 道;在作业执行中又发生竞争主存,再送出一道;这样直到最后两道时发生竞争主存还要送出一道,从而得出上面的公式。

4.3 分页式存储管理

4.3.1 分页式存储管理的基本原理

用分区方式管理的存储器,每道程序总是要求占用主存的一个或几个连续存储区域,主存中会产生许多碎片。因此,有时为了接纳一个新的作业而往往要移动已在主存的信息,这不仅不方便,而且开销不小。采用分页式存储器允许把一个作业存放到若干不相邻接的分区中,既可免去移动信息的工作,又可充分利用主存空间,尽量减少主存内的碎片。分页式存储管理的基本原理如下:

1. 页框,物理地址分成大小相等的许多区,每个区称为一块(又称页框 page frame);
2. 页面,逻辑地址分成大小相等的区,区的大小与块的大小相等,每个区称一个页面(page)。
3. 逻辑地址形式,与此对应,分页存储器的逻辑地址由两部分组成:页号和单元号。逻辑地址格式如下:

| | |
|-----|-------|
| 页 号 | 单 元 号 |
|-----|-------|

采用分页式存储管理时,逻辑地址是连续的。所以,用户在编制程序时仍只须使用顺序的地址,而不必考虑如何去分页。由地址转换机构和操作系统管理的需要来决定页面的大小,从而,也就确定了主存分块的大小。用户进程在主存空间中的每个页框内的地址是连续的,但页框和页框之间的地址可以不连续。存储地址由连续到离散的变化,为以后实现程序的“部分装入、部分对换”奠定了基础。

4. 页表和地址转换:在进行存储分配时,总是以块(页框)为单位的,一个作业的信息有多少页,那么,在把它装入主存时就给它分配多少块。但是,分配给作业的主存块可以不连续,即作业的信息可按页分散存放在主存的空闲块中。这就避免了为得到连续存储空间而进行的移动。那么,当作业的程序和数据被分散存放后,作业的页面与分给的块(页框)如何

建立联系呢？逻辑地址（页面）如何变换成物理地址（页框）呢？作业的物理地址空间由连续变成分散后，如何保证程序的正确执行呢？采用的办法仍然是动态重定位技术，让程序的指令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，给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便称页表（page table）。页表是操作系统为每个用户作业建立的，用来记录程序页面和主存对应块（页框）的对照表，页表中的每一栏指明了程序中的一个页面和分得的块（页框）的对应关系。所以，页表的目的是把页面映射为页框，从数学的角度来说，页表是一个函数，它的变量是页面号，函数值为页框号，通过这个函数可把逻辑地址中的逻辑页面域替换成物理页框域。通常为了减少开销，不是用硬件，而是在主存中开辟存储区存放页表，系统中另设一个页表主存起址和长度控制寄存器（page table control register），存放当前运行作业的页表起址和页表长，以加快地址转换速度。每当选中作业运行时，应进行存储分配，为进入主存的每个用户作业建立一张页表，指出逻辑地址中页号与主存中块号的对应关系，页表的长度随作业的大小而定。同时分页式存储管理系统还建立一张作业表，将这些作业的页表地址进行登记，每个作业在作业表中有一个登记项。作业表和页表的一般格式如图 4-12。然后，借助于硬件的地址转换机构，在作业执行过程中按页面动态重定位。调度程序在选择作业后，从作业表的登记项中得到被选中作业的页表始址和长度，将其送入硬件设置的页表控制寄存器。地址转换时，只要从页表控制寄存器就可以找到相应的页表，再按照逻辑地址中的页号作索引查页表，得到对应的块号，根据关系式：

$$\text{绝对地址} = \text{块号} \times \text{块长} + \text{单元号}$$

计算出欲访问的主存单元的地址。因此，虽然作业存放在若干个不连续的块中，但在作业执行中总是能按正确的地址进行存取。

| 页号 | 块号 | 作业名 | 页表始址 | 页表长度 |
|-------|------|-----|------|------|
| 第 0 页 | 块号 1 | A | XXX | XX |
| 第 1 页 | 块号 2 | B | XXX | XX |
| ... | ... | ... | ... | ... |

页表

作业表

图 4-12 页表和作业表

图 4-13 给出了页式存储管理的地址转换和存储保护，根据地址转换公式：块号 \times 块长 + 单元号，在实际进行地址转换时，只要把逻辑地址中的单元号作为绝对地址中的低地址部分，而根据页号从表中查得的块号作为绝对地址中的高地址部分，就组成了访问主存储器的绝对地址。

整个系统只有一个页表控制寄存器，只有占用 CPU 的作业才占有页表控制寄存器。在多道程序中，当某道程序让出处理器时，应同时让出页表控制寄存器。

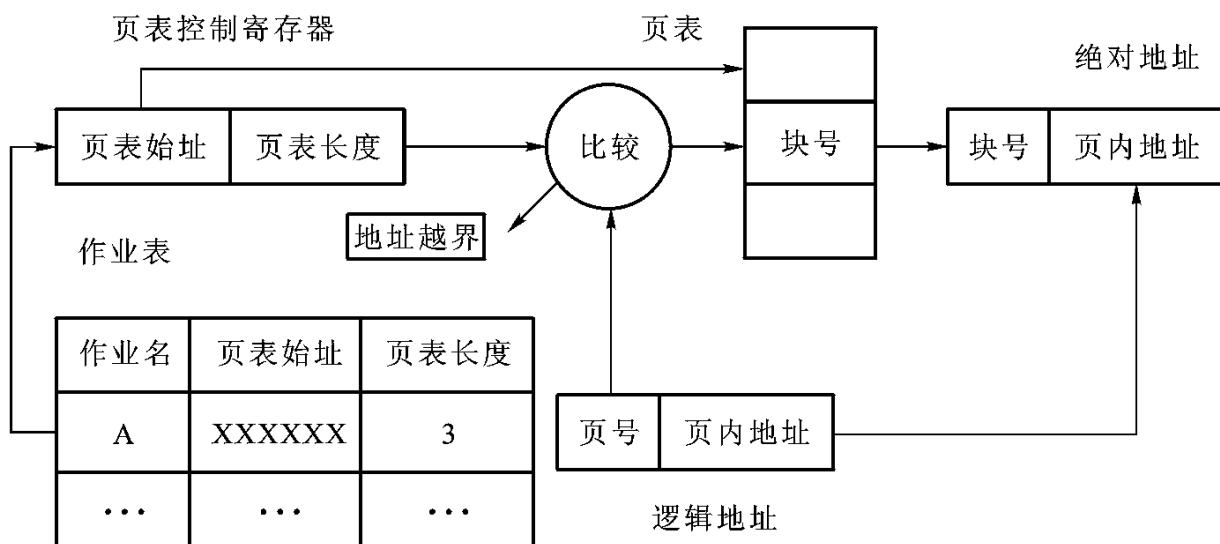


图 4-13 页式存储管理的地址转换和存储保护

4.3.2 相联存储器和快表

页表可以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块号。这虽然方便了地址转换, 但硬件花费代价太高, 如果把页表放在主存中就可降低计算的开销。但是, 当要按给定的逻辑地址进行读/写时, 必须访问两次主存。第一次按页号读出页表中相应栏内容的块号, 第二次根据计算出来的绝对地址进行读/写, 降低了运算速度, 这比通常执行指令的速度慢了一倍。

为了提高运算速度, 通常都在 MMU(将在后面介绍)中设置一个专用的高速缓冲存储器, 用来存放最近问的部分页表。这种高速存储器称为相联存储器(associative memory), 也称 TLB(Translation Lookaside Buffer), 它成为分页式存储管理的一个重要组成部分。存放在相联存储器中的页表称快表。相联存储器的存取时间是远小于主存的, 速度快但造价高, 故一般都是小容量的, 例如 Intel 80486 的快表为 32 个单元。根据程序执行局部性的特点, 在一段时间内总是经常访问某些页面, 若把这些页面登记在快表中, 无疑地将大大加快指令的执行速度。

由于只是进程页表的部分内容在快表中, 它仅指出页框号是不行的, 还要给出逻辑页号, 以及该页面的其他特征位, 如保护位、修改位。有了快表后, 绝对地址形成的过程是: 按处理器给出的逻辑地址中的页号, 由地址转换机构 MMU 自动查快表, 若该页已登记在快表中, 并且符合访问权限, 则由块号和单元号形成绝对地址; 若快表中查不到对应页号, 则再查主存中的页表而形成绝对地址, 同时将该页登记到快表中。为了加快地址转换过程, 实际上上述两者的查找是同时进行的, 一旦快表中发现了要查找的页号, 则立即停止主存中的页表查找。当快表填满后, 又要在快表中登记新页时, 则需在快表中按一定策略淘汰一个旧的登

记项,最简单的策略是“先进先出”,总是淘汰最先登记的那一页。

通过快表实现内存访问的比率称命中率(hit ratio),命中率越高,性能越好。接近100%的命中率表明绝大部分访问主存都通过快表实现,而几乎不用页表。反之,当一个进程访问遍布主存的页的跳跃连接的地址时,命中率接近为0,这意味着每次访问主存都要使用页表。采用相联存储器的方法后,地址转换时间大大下降。假定访问主存的时间为100毫微秒,访问相联存储器的时间为20毫微秒,相联存储器为32个单元时查快表的命中率可达90%,于是按逻辑地址进行存取的平均时间为:

$$(100 + 20) \times 90\% + (100 + 100) \times (1 - 90\%) = 128 \text{ 毫微秒}$$

比两次访问主存的时间200毫微秒下降了近四成。

同样,整个系统也只有一个相联存储器,只有占用CPU者才占有相联存储器。在多道程序中,当某道程序让出处理器时,应同时让出相联存储器。由于快表是动态变化的,所以,让出相联存储器时应把快表保护好以便再执行时使用。当一道程序占用处理器时,除置页表控制寄存器外,还应将它的快表内容送入相联存储器。

4.3.3 分页式存储空间的分配和去配

分页式存储管理把主存的可分配区按页面大小分成若干块,主存分配以块为单位。最简单的办法可用一张位示图来记录主存分配情况,指出已分配的块和尚未分配的块以及当前有多少空闲块。该表可存放在主存的专门区域中,表格的每一位与一个物理块对应,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4-14。

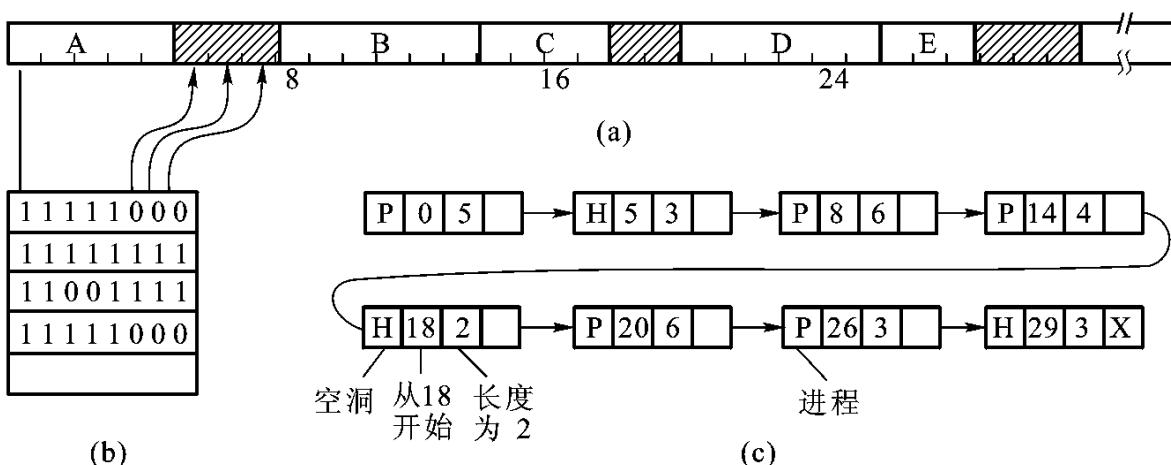


图4-14 内存分配的位示图和链表方法

分页式存储管理的页框分配算法如下:进行主存分配时,先查空闲块数能否满足作业要求,若不能满足,令该作业等待;若能满足,则查位示图,找出为“0”的那些位,置上占用标志,从空闲块数中减去本次占用块数,按找到的位计算出对应的块号,填入该作业的页表。当一

个作业执行结束, 归还主存时, 则根据归还的块号, 计算出在位示图中的位置, 将占有标志清成“0”, 归还块数加入到空闲块数中。

图 4-14(c) 是内存分配的链表方法, 表中每一项都含以下内容: 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, 链表是按照地址排序的, 其优点是链表的更新和修改较方便。一个要结束的进程一般讲有两个邻居, 可能是进程也可能是空闲区, 只要修改紧靠的几个链表项。采用链表方法的页框分配和去配算法作为练习留给同学们完成。

4.3.4 分页存储空间的页面共享和保护

分页存储管理能实现多个作业共享程序和数据。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, 这些共享的信息在主存中只要保留一个副本。页的共享可大大提高主存空间的利用率。例如, 一个编辑程序共 30 K, 现有 10 个作业, 他们所处理的数据平均为 5 K。如果不采用共享技术的话, 那么, 10 个作业共需 350 K 主存, 而共享编辑程序的话, 则只需 80 K 主存。

分页存储管理在实现共享时, 必须区分数据共享和程序共享。实现数据共享时, 可允许不同的作业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目指向共享的数据信息块就行了。实现程序共享时, 情况就不同了, 由于页式存储结构要求逻辑地址空间是连续的, 所以, 程序运行前它们的页号是确定的。现假定有一个被共享的编辑程序 EDIT, 其中含有转移指令, 转移指令中的转移地址必须指出页号和单元号, 如果是转向本页, 则页号应与本页的页号相同。现在若有两个作业共享这个 EDIT 程序, 假定一个作业定义它的页号为 3, 另一作业定义它的页号为 5, 然而, 在主存中只有一个 EDIT 程序, 它要为两个作业以同样的方式服务, 则这个程序一定是可再入的。于是转移地址中的页号不能按作业的要求随机的改成 3 或 5, 因此, 对共享程序必须规定一个统一的页号。当共享程序的作业数增多时, 要规定一个统一的页号是较困难的。

实现信息共享必须解决共享信息的保护问题。通常的办法是在页表中增加一些标志位, 用来指出该页的信息可读/写 \ 只读 \ 只可执行 \ 不可访问等, 指令执行时进行核对。例如, 要想向只读块写入信息则指令停止执行, 产生中断信号。

另外, 也可采取存储保护键作为保护机制, 系统为每道作业分配一个存储保护键, 为某作业分配主存时根据它分得的保护键, 由系统为它写到分配的页框保护键中。程序执行时将程序状态字 PSW 中的控制键和访问页的“存储保护键”进行核对, 相符时才可访问该块, 用这种办法来实现存储保护。本书 CH7 中将介绍 IBM System/370 系列操作系统的存储保护键保护机制。

4.3.5 多级页表

现代计算机已普遍使用 32 或 64 位虚拟地址,可以支持 $2^{32} \sim 2^{64}$ 容量的逻辑地址空间,采用页式存储管理时,页表会相当的大。以 Windows 2000/XP 为例,其运行的 Intel x86 CPU 具有 32 位地址,使用 2^{32} 逻辑地址空间的分页系统,规定页面 4 KB (2^{12}) 时,那么,4 GB (2^{32}) 的虚地址空间有 1 兆 (2^{20}) 个页组成,若以每个页表项占用 4 个字节计算,则需要占用 4 MB (2^{22}) 连续内存空间存放页表,这还是一个进程的地址空间,系统中有许多进程,这样做存储开销太大。为此,页表和页面一样也要进行分页,提出了多级页表的概念,采取以下措施:对进程页表使用的空间也进行动态分配,当创建进程时并不分配所需的全部页表空间,而是缺页中断调页或分配页表页时,才在这个页框中装入页表或页表页。显然,这时页表中的逻辑页号不再连续顺序,因而,页表中每项要同时存放页号和页框号。由于页表占用多个页,通常页表本身的空间分配也以页面为单位,称为页表页,动态分配决定了进程页表的页表页占用的内存空间不一定连续存放。此外,鉴于页表占用的内存空间很大,大部分已经在内存的页面多数时间不在使用,进程页表占用页面也允许页淘汰。具体做法是:把整个页表进行分页,分成一张张页表页,每个页表页的大小与页面长度相同,例如,每个页表页形成的页面可以有 1 K (2^{10}) 个页表项。可对页表页顺序编号,允许页表页分散存放在不连续的页框中,为了进行索引查找,应该为这些页表页建立一张地址索引,称为页目录表 (page directory table),其表项指出页表页所在页框号及相关信息。于是系统要为每个进程建一张页目录表,它的每个表项指出一个页表页,页表页的长度被限制成等于一页,而页表页的每个表项给出了页面和页框的对应关系,页目录表是一级页表,页表页是二级页表,共同构成了二级页表机制,现在已有一些 CPU 对二级页表机制提供硬件支持。于是逻辑地址结构有三部分组成:页目录位移、页表页位移和页内位移。图 4-15 是二级页表实现逻辑地址到物理地址的转换过程,具体步骤如下:

由硬件页目录表控制寄存器指出当前运行进程的页目录表内存所在地址,由页目录表起址加上“页目录位移”作索引,可找到某个页表页在内存的页框(其中存放着一个页表页),再以“页表页位移”作索引,找到页表页的页表项,而该表项中包含了一个页面对应的页框号,页框号和“页内位移”便可生成物理地址。

上面的方法虽解决了可分散存放页表页的问题,但并未解决页表页占用内存空间的问题。解决后一个问题的方法如下:对于进程运行涉及页面的页表页应放在内存,而其他页表页使用时再行调入。为了实现这一点,需要在页目录表中增加一个特征位,用来指示对应的页表页是否已调入内存,地址转换机构根据逻辑地址中的“页目录位移”去查页目录表对应表项,如未调入内存,应产生一个“缺页表页”中断信号,请求操作系统将这张页表页调入内存。

二级页表地址变换需三次访问主存,一次访问页目录、一次访问页表页、一次访问指令

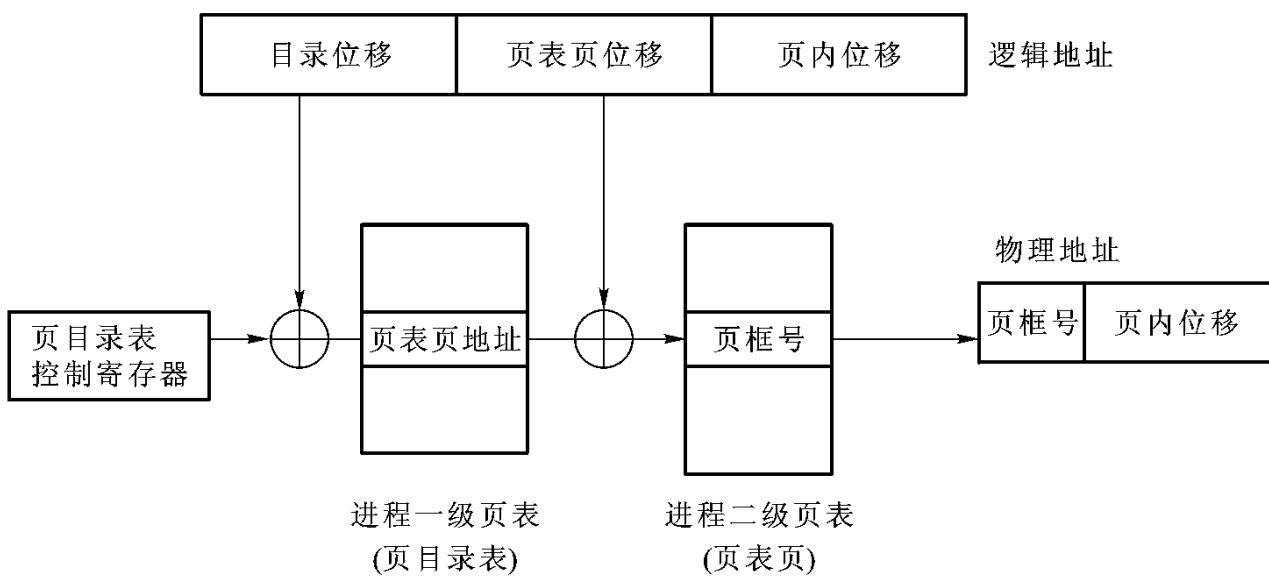


图 4-15 二级页表地址转换过程

或数据，访问时间加了两倍。随着 64 位地址出现，三级、四级页表也已被引入系统。

SUN 公司的计算机使用 SPARC 芯片，采用如图 4-16 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，硬件可以支持多达 4096 个上下文，每个进程一个。当一个新进程装入主存时，操作系统分给一个上下文号，进程保持这个上下文号直到终止。当 CPU 访问内存时，上下文号和逻辑地址一起送入称作 MMU 的地址转换机构（本章后面介绍），它使用上下文号作上下文表的索引，以找到进程的顶级页目录，然后，使用逻辑地址中的索引值找下一级页目录表项，直至找到访问页面，形成物理地址。在分页系统中，为了加快地址转换过程，都会使用相联存储器。

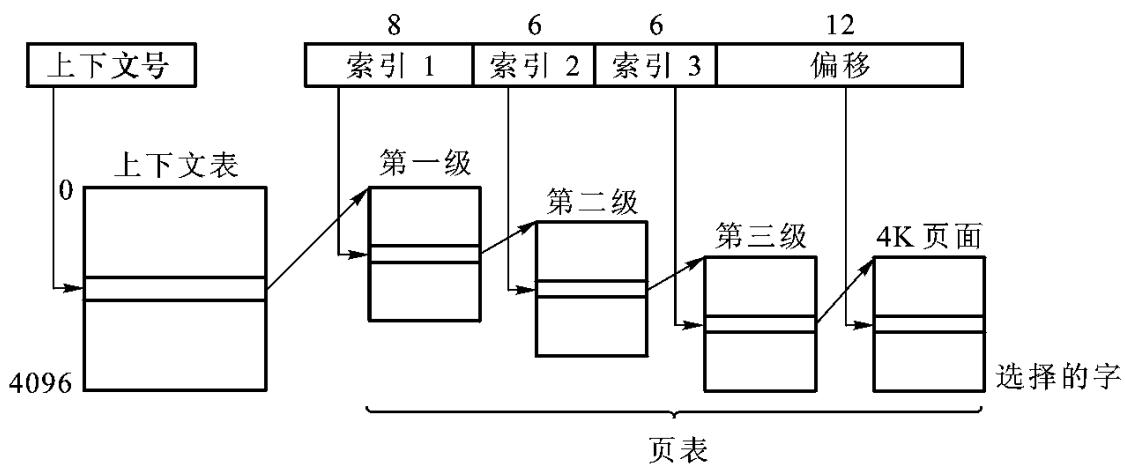


图 4-16 SPARC 三级分页结构

多级页表结构的本质是多级不连续，这导致了多级索引。以二级页表结构为例，用户的页面不连续存放，需要有页面地址索引，该索引就是进程页表；而由于进程页表又是不

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,该索引就是页目录。这就形成了二级索引,页目录项是页表页的索引,而页表页项是进程程序的页面索引。

4.3.6 反置页表

由于计算机逻辑空间越来越大,页表占用的内存空间也越来越多,页表尺寸与虚地址空间成正比增长。为了减少内存空间的开销,不得不使用多级页表,但也有许多机器和操作系统如 IBM AS/400、Mac OS 等,采用了称为反置页表 IPT(Inverted Page Table)的方法, IPT 维护了一个页表的反置页表,它为内存中的每一个物理块建立一个页表并按照块号排序,该表的每个表项包含正在访问该页框的进程标识、页号及特征位,和哈希链指针等,用来完成内存页框到访问进程的页号,即物理地址到逻辑地址的转换。图 4-17 是反置页表及地址转换,其地址转换过程如下:处理器给出逻辑地址,由于 MMU 的任务是把逻辑地址转换成物理地址,IPT 不能完成这样的转换。因此,系统采用哈希表技术完成虚地址转换,MMU 通过哈希表把进程标识和虚页号转换成一个哈希值,该值指向 IPT 的一个表目,然后,遍历哈希链找到所需进程的虚页号,而该项的索引就是页框号,通过拼接位移便可生成物理地址。若整个反置页表中未能找到匹配的页表项,说明该页不在内存,产生请页中断,请求操作系统调入。

表目中增加链指针,是由于有多于一个的虚页号经哈希函数转换后可能得到同样的哈希值。所以,利用哈希链来处理冲突,在进行地址映射时,用哈希表定位后要遍历哈希链再逐个找出所需的虚页面。

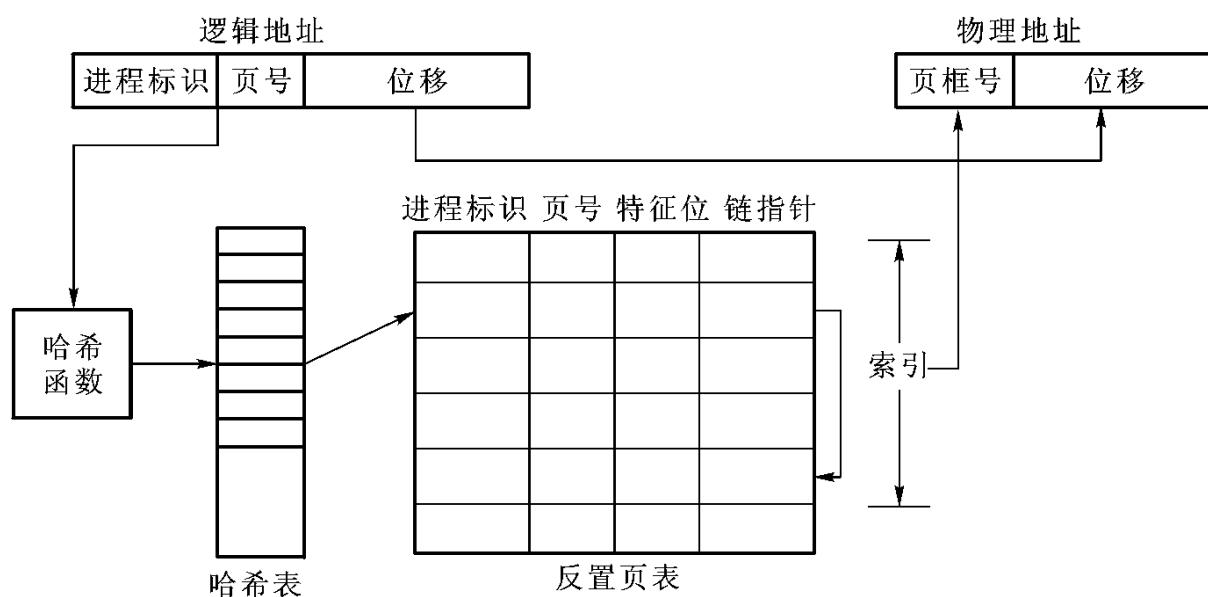


图 4-17 反置页表及其地址转换

虽然 IPT 能减少页表占用内存,如一个 128 MB 的内存空间,若页面大小为 1 KB,则 IPT 只需 128 KB。然而, IPT 仅包含了调入内存的页面,不包含未调入内存的页面,所以,仍需要

为进程建立传统的页表,不过这种页表不再放在内存中,而存在磁盘上。当发生缺页中断时,把所需页面调入内存要多访问一次磁盘,这时的速度是较慢的。

4.4 分段式存储管理

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,从分区方式到分页方式发展的主要原因是提高了主存空间利用率。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,这些要求其他各种存储管理技术难以满足。在分页存储管理中,经连接编辑处理得到了一维地址结构的可装配模块,这是从0开始编址的一个单一连续的逻辑地址空间,虽然,操作系统可把程序划分成页面,但页面与源程序无逻辑关系,也就难以实现对源程序以模块为单位进行分配、共享和保护。事实上,程序还可以有一种分段结构,现代高级语言常常采用模块化程序设计。如图4-18所示,一个程序由若干程序段(模块)组成。例如,由一个主程序段、若干子程序段、数组段和工作区段所组成,每个段都从“0”开始编址,每个段都有模块名,且具有完整的逻辑意义。段与段之间的地址不一定连续,而段内地址是连续的。用户程序中可用符号形式(指出段名和入口)调用某段的功能,程序在编译或汇编时给每个段再定义一个段号。可见这是一个二维地址结构,分段方式的程序被装入物理地址空间后,仍应保持二维地址结构,这样才能满足用户模块化程序设计的需要。这种二维地址结构需要编译程序的支持,但对程序员来说通常是透明的。

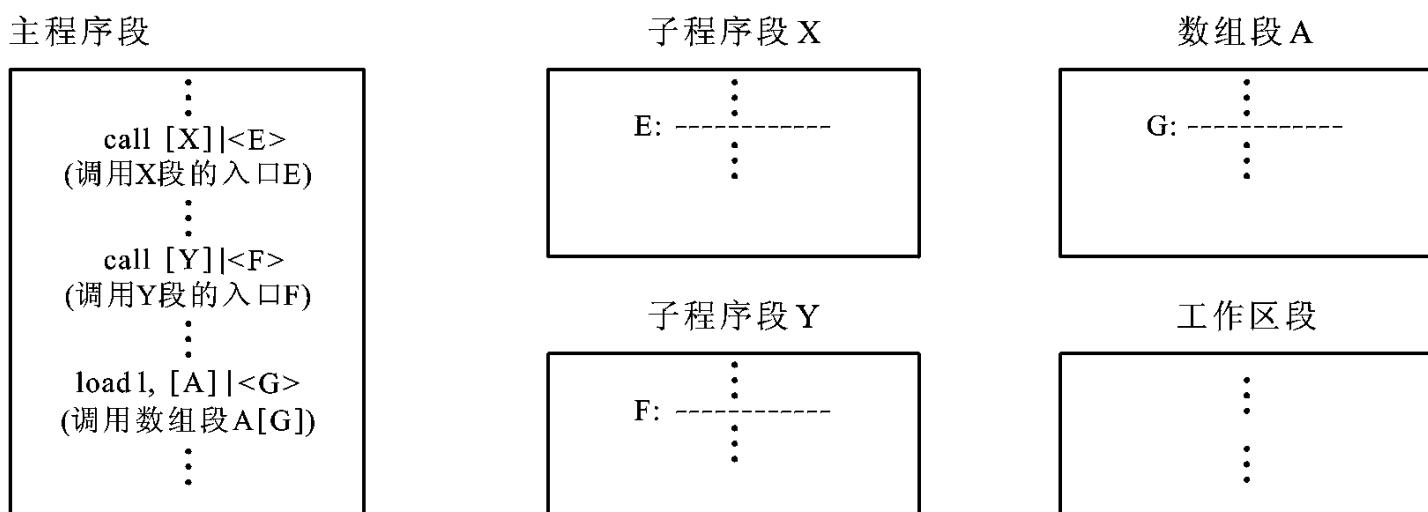


图4-18 程序的分段结构

4.4.2 分段式存储管理的基本原理

分段式存储管理是以段为单位进行存储分配,为此提供如下形式的两维逻辑地址:

| | |
|----|------|
| 段号 | 段内地址 |
|----|------|

在分页式存储管理中,页的划分——即逻辑地址划分为页号和单元号是用户不可见的,连续的用户地址空间将根据页框(块)的大小自动分页;而在分段式存储管理中,地址结构是用户可见的,即用户知道逻辑地址如何划分为段号和单元号,用户在程序设计时,每个段的最大长度受到地址结构的限制,进一步,每一个程序中允许的最多段数也可能受到限制。例如,PDP-11/45 的段地址结构为:段号占 3 位,单元号占 13 位,也就是一个作业最多可分 8 段,每段的长度可达 8K 字节。

分段式存储管理的实现可以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划分和连续存放,也就是说,在一个分区内作业是连续存放的,但独立的作业之间不一定连续存放。而分段方法是以段为单位划分和连续存放,为作业的每一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,应为进入主存的每个用户作业建立一张段表,各段在主存的情况可用一张段表来记录,它指出主存储器中每个分段的起始地址和长度。同时段式存储管理系统包括一张作业表,将这些作业的段表进行登记,每个作业在作业表中有一个登记项。作业表和段表的一般格式如图 4-19:

| 段号 | 始址 | 长度 | 作业名 | 段表始址 | 段表长度 |
|-------|-----|-----|-----|------|------|
| 第 0 段 | XXX | XXX | A | XXX | XX |
| 第 1 段 | XXX | XXX | B | XXX | XX |
| ... | ... | ... | ... | ... | ... |

段表

作业表

图 4-19 段表和作业表的一般格式

段表实际上起到了基址/限长寄存器的作用。作业执行时通过段表可将逻辑地址转换成绝对地址。由于每个作业都有自己的段表,地址转换应按各自的段表进行。类似于分页存储器那样,分段存储器也设置一个段表控制寄存器,用来存放当前占用处理器的作业的段表的起始地址和长度。段式存储管理的地址转换和存储保护流程如图 4-20。利用段表寄存器中的段表长度与逻辑地址中的段号比较,若段号超过段表长则产生越界中断,再利用段表项中的段长与逻辑地址中的段内位移比较,检查是否产生越界中断。

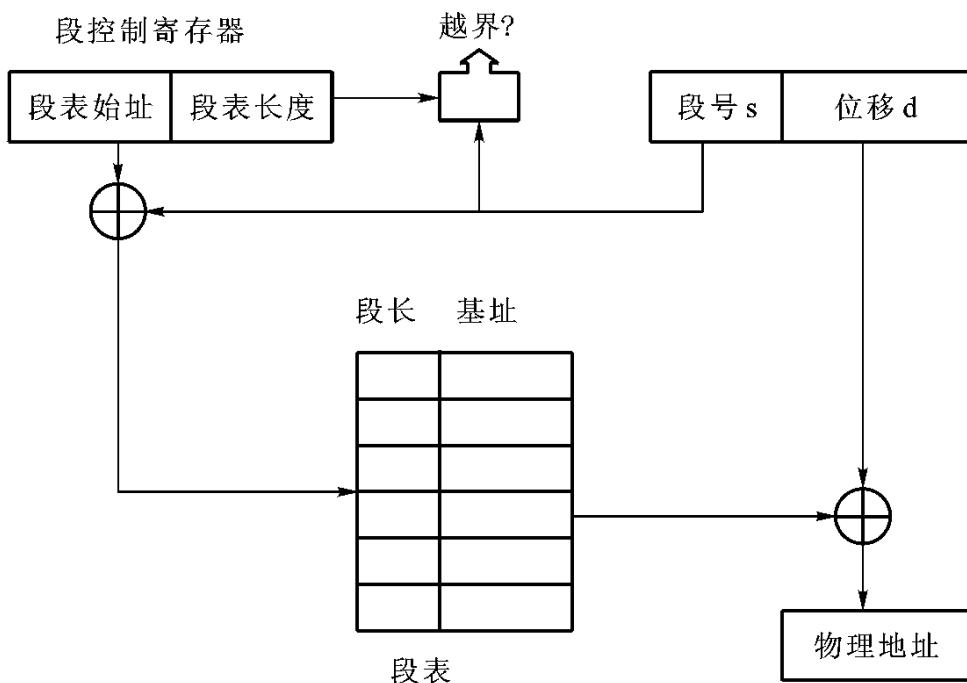


图 4-20 分段式存储管理的地址转换和存储保护

4.4.3 段的共享

在可变分区存储管理中,每个作业只能占用一个分区,那么,就不允许各道作业有公共的区域。这样,当几道作业都要用某个例行程序时就只好在各自的区域内各放一套。显然,降低了主存的使用效率。

在分段式存储管理中,由于每个作业可以有几个段组成,所以,可以实现段的共享,以存放共享的程序和数据。所谓段的共享,是通过不同作业段表中的项指向同一个段基址来实现的。于是,几道作业共享的例行程序就可放在一个段中,只要让各道作业的共享部分有相同的基址/限长值就行了。在请求分段虚拟存储管理中还会对段的共享作进一步介绍。

当然对共享段的信息必须进行保护,如规定只能读出不能写入,欲想往该区域写入信息时将遭到拒绝并产生中断。

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所决定,用户可见,段长可根据用户需要来规定,段起始地址可以从任何主存地址开始。在分段方式中,源程序(段号,段内位移)经连接装配后仍保持二维(地址)结构。

分页是信息的物理单位,与源程序的逻辑结构无关,用户不可见,页长由系统确定,页面只能以页大小的整倍数地址开始。在分页方式中,源程序(页号,页内位移)经连接装配后变

成了一维(地址)结构。

4.5 虚拟存储管理

4.5.1 虚拟存储器的概念

在前面介绍的各种存储管理方式中,必须为作业分配足够的存储空间,以装入有关作业的全部信息,当然作业的大小不能超出主存的可用空间,否则,这个作业是无法运行的。但当把有关作业的全部信息都装入主存储器后,作业执行时实际上不是同时使用全部信息的,有些部分运行一遍便再也不用,甚至有些部分在作业执行的整个过程中都不会被使用(如错误处理部分)。作业在运行时不用的,或暂时不用的,或某种条件下才用的程序和数据,全部驻留于主存中是对宝贵的主存资源的一种浪费,大大降低了主存利用率。于是,提出了这样的问题:作业提交时,先全部进入辅助存储器,作业投入运行时,能否不把作业的全部信息同时装入主存储器,而是将其中当前使用部分先装入主存储器,其余暂时不用的部分先存放在作为主存扩充的辅助存储器中,待用到这些信息时,再由系统自动把它们装入到主存储器中,这就是虚拟存储器的基本思路。当一个进程访问的程序和数据在主存中,执行就可以顺利进行。如果处理器访问了不在主存的程序或数据,为了继续执行下去,需要由系统自动将这部分信息装入主存储器,这叫部分装入;如若主存中没有足够空闲空间,便需要把主存中暂时不用的信息从主存移到辅存上去,这叫部分对换。如果“部分装入、部分对换”这个问题能解决的话,那么,当主存空间小于作业需要量时,这个作业也能执行;更进一步,多个作业存储总量超出主存总容量时,也可以把它们全部装入主存,实现多道程序运行。这样,不仅使主存空间能充分地被利用,而且用户编制程序时可以不必考虑主存储器的实际容量的大小,允许用户的逻辑地址空间大于主存储器的绝对地址空间。对于用户来说,好像计算机系统具有一个容量硕大的主存储器,把它称作为“虚拟存储器”(virtual memory)(Fotheringham, 1961)。

现在给出虚拟存储器的定义如下:在具有层次结构存储器的计算机系统中,采用自动实现部分装入和部分对换功能,为用户提供一个比物理主存容量大得多的,可寻址的一种“主存储器”。实际上虚拟存储器是为扩大主存而采用的一种设计技巧,虚拟存储器的容量与主存大小无直接关系,而受限于计算机的地址结构及可用的辅助存储器的容量,如果地址线是20位,那么,程序可寻址范围是1 MB, Intel pentium 的地址线是32位,则程序可寻址范围是4 GB, Windows 2000/XP 便为应用程序提供了一个4 GB的逻辑主存。图4-21给出了虚拟存储器的概念图。

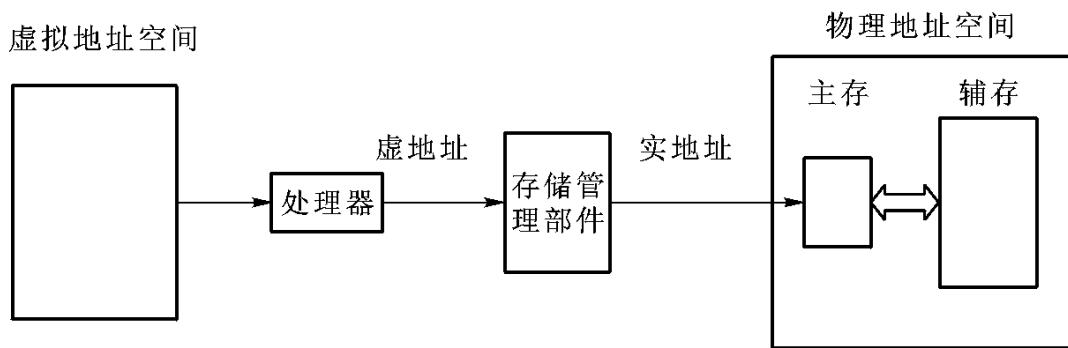


图 4-21 虚拟存储器概念

现在,进而讨论在作业信息不全部装入主存的情况下能否保证作业的正确运行?回答是肯定的,早在 1968 年 P. Denning 就研究了程序执行时的局部性(principle of locality)原理,对程序局部性原理进行研究还有 Knuth(分析一组学生的 Fortran 程序)、Tanenbaum(分析操作系统的进程)、Huck(分析通用科学计算的程序),发现程序和数据的访问都有聚集成群的倾向,在一个时间段内,仅使用其中一小部分(称空间局部性),或者最近访问过的程序代码和数据,很快又被访问的可能性很大(称时间局部性)。这只要对程序的执行进行分析就可以发现以下一些情况:

第一,程序中只有少量分支和过程调用,大都是顺序执行的,即要读取的下一条指令紧跟在当前执行指令之后。第二,程序往往包含若干个循环,这些是由相对较少的几个指令重复若干次组成的,在循环过程中,计算被限制在程序中一个很小的相邻部分中(如计数循环)。第三,很少会出现连续不断的过程调用序列,相反,程序中过程调用的深度限制在一个小的范围内,因而,一段时间内,指令引用被局限在很少几个过程中。第四,对于连续访问数组之类的数据结构,往往是对存储区域中的数据或相邻位置的数据(如动态数组)的操作。第五,程序中有些部分是彼此互斥的,不是每次运行时都用到的,例如,出错处理程序,仅当在数据和计算中出现错误时才会用到,正常情况下,出错处理程序不放在主存,不影响整个程序的运行。上述种种情况充分说明,作业执行时没有必要把全部信息同时存放在主存储器中,而仅仅只需装入一部分的假设是合理的。在装入部分信息的情况下,只要调度得好,不仅可以正确运行,而且可以在主存中放置更多进程,有利于充分利用处理器和提高存储空间的利用率。

虚拟存储器是基于程序局部性原理上的一种假想的而不是物理存在的存储器,允许用户程序以逻辑地址来寻址,而不必考虑物理上可获得的主存大小,这种将物理空间和逻辑空间分开编址但又统一管理和使用的技术为用户编程提供了极大方便。虚存管理与对换技术虽说都是在主存和辅存之间交换信息块,但却有很大区别。对换是以进程为单位的,当进程所需主存大于当前系统拥有量时,则该进程是无法被对换进主存工作的;而虚存管理以页或

段为单位,即使一个进程所需主存大于当前系统拥有量时,该进程仍然能在系统中正常运行,因为它或其他进程的一部分页或段可部分换出到辅存上。

虚拟存储器的思想早在 60 年代初期就已在英国的 Atlas 计算机上出现,到 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。70 年代初期开始推广应用,逐步为广大计算机研制者和用户接受,虚拟存储技术不仅用于大型机上,而且随着微型机的迅速发展,也研制出了微型机虚拟存储系统。为了要实现虚拟存储器,必须解决好以下有关问题:主存辅存统一管理问题、逻辑地址到物理地址的转换问题、部分装入和部分对换问题。实现虚拟存储器系统要付出一定的开销,其中包括:管理地址转换各种数据结构所用的存储开销、执行地址转换的指令花费的时间开销和主存与外存交换页或段的 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:请求分页式、请求分段式和请求段页式虚拟存储管理。

4.5.2 请求分页虚拟存储管理

1. 分页式虚拟存储系统的硬件支撑

操作系统的存储管理依靠低层硬件的支撑来完成任务,该硬件称为主存管理单元 MMU (Memory Management Unit)。现代计算机中的 MMU 完成逻辑地址到物理地址的转换功能,通常它是由一个或一组芯片组成,接受虚拟地址作为输入,物理地址作为输出,直接送到总线上,对内存单元进行寻址。其位置和功能如图 4-22(a) 所示,其内部执行过程如图 4-22(b) 所示。主要功能包括:

- 管理硬件页表基址寄存器。每当发生进程上下文切换时,由系统负责把将要运行进程的页表的基地址装入该寄存器,此页表便成为活动页表,MMU 只对由硬件页表基址寄存器指出的活动页表进行操作。
- 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。
- 管理快表 TLB。MMU 对 TLB 的管理有两项,一是直接查找快表,找到相应页框后去拼接物理地址;二是执行 TLB 的两个基本操作:装入表目和清除表目,每次发生快表查找不到的情况,待缺页中断处理结束后负责把相应页面和页框装入快表。此外,在每次写硬件页表基址寄存器时,负责清除快表项,将 TLB 清空。
- 访问页表。当 TLB 不命中时,对进程页表的访问有软件处理和硬件处理两种方式。软件处理方式下,当 TLB 不命中时,由 MMU 产生中断,转让给操作系统访问进程页表并修改 TLB。硬件处理方式下,由 MMU 根据硬件页表基址寄存器直接访问进程页表。
- 当出现页表中有页失效位,或页面访问越界位时,MMU 发出缺页中断或越界中断,并将控制权交给内核存储管理处理。

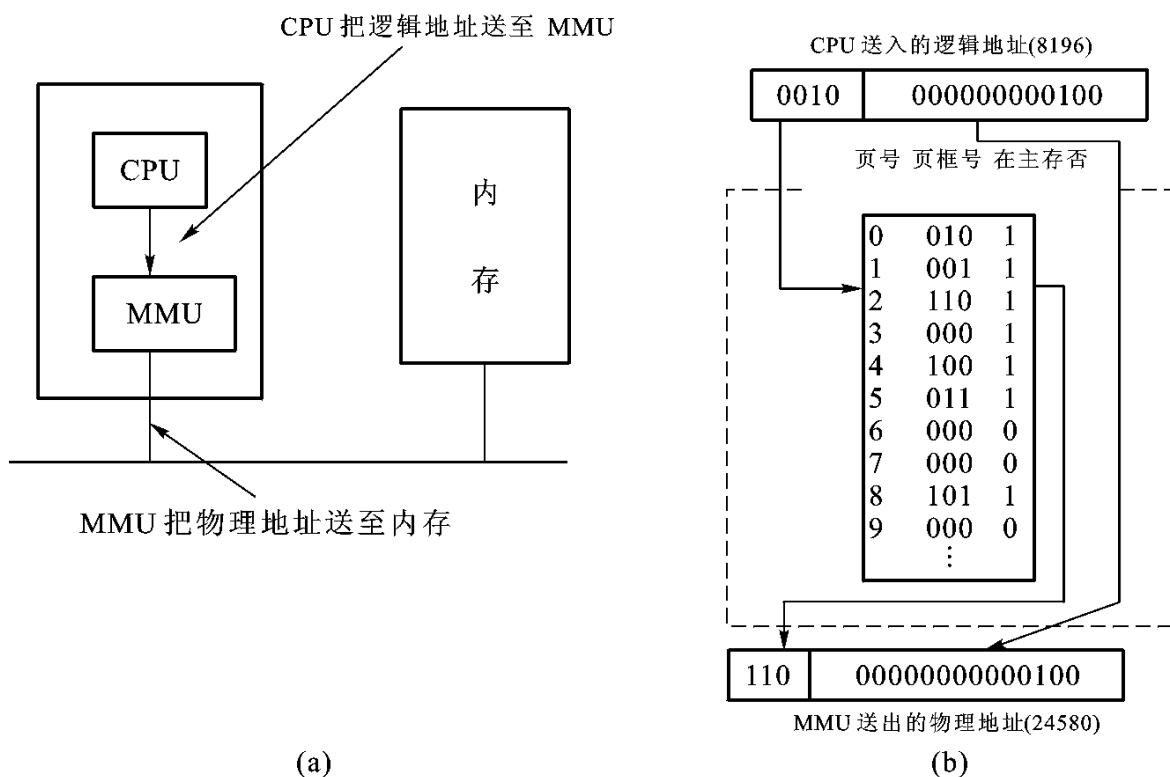


图 4-22 (a) MMU 的位置和功能 (b) 16 个 4 KB 页面情况下 MMU 的内部操作

- MMU 负责设置和检查页表中的引用位、修改位、有效位和保护权限等各个特征位。
- 下面来考察 MMU 的工作过程。在图 4-22(b) 中, 给出了一个虚地址 8196(二进制为 0010000000000100), MMU 进行映射, 输入的 16 位虚地址被解释为 4 位页号和 12 位的偏移量。用页号作索引, 以查找到该虚页对应的页框号。如果“在主存位”为 1, 表明该页在主存, 把页框号拷贝到输出寄存器的高三位中, 再加上虚地址中的 12 位偏移, 就生成了 15 位的物理地址(二进制为 1100000000000100), 并把它作为物理地址送到主存总线。如果“在主存位”为 0, 则将引起一个缺页中断, 陷入操作系统进行调页处理。

2. 请求分页虚拟存储系统的基本原理

请求分页虚拟存储系统是将作业信息的副本存放在磁盘这一类辅助存储器中, 当作业被调度投入运行时, 并不把作业的程序和数据全部装入主存, 而仅仅装入立即使用的那些页面, 至少要将作业的第一页信息装入主存, 在执行过程中访问到不在主存的页面时, 再把它们动态地装入。用得较多的分页式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, 而发现它们并不在主存时, 产生一个缺页中断, 系统从辅存中把该指令或数据所在的页面调入内存。

由于请求分页虚存管理与分页式实存管理不同, 仅让作业或进程当前使用部分放在主存中, 所以, 执行过程中必然会发生某些页面不在主存中的情况, 那么, 怎样才能发现页面不

在内存中呢？怎样处理这种情况呢？采用的办法是：扩充页表的内容，增加驻留标志位和页面辅存的地址等信息，扩充后的页表，如图 4-23 所示：

| 页号 | 驻留标志 | 页框号 | 辅存地址 | 其它标志 |
|----|------|-----|------|------|
| | | | | |
| | | | | |
| | | | | |

图 4-23 请求分页式虚拟存储管理表页表

驻留标志位(又称页失效中断位)用来指出对应页是否已经装入主存，访问一个页面时，如果某页所对应栏的驻留标志位为 1，则表示该页已经在主存；若驻留标志位为 0，此时产生一个缺页中断信号，可以根据辅存地址知道该页在辅助存储器中的位置，将这个页面调入主存。缺页中断是由于发现当前访问页面不在主存时由硬件产生的一种特殊中断信号，通常，CPU 都是在一条指令执行完成后再去检查是否有(一般)中断到达，也就是说只能在两条指令之间响应(一般)中断，但一个缺页中断却是在指令执行期间产生和获得系统的处理。而且，一条指令可能涉及到多个页面，例如，指令本身跨页、指令处理的数据跨页，完全有可能执行一条指令过程中发生多次缺页中断。为此，系统需要提供寄存器和特殊的返回指令等硬件设施，确保在出现缺页中断时，保存未完成指令的状态和恢复原指令的执行。

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(Modified)、引用位(Renferenced)和访问权限位等，用来跟踪页的使用情况。当一个页被修改后，硬件自动设置修改位，一旦修改位被设置，当该页被调出主存时必须先重新被写回辅存。引用位则在该页被引用时(无论是读或写)设置，其值被用来帮助操作系统进行页面淘汰。访问权限位则限定了该页允许什么样的访问权限(如是否可写)。还有的操作系统中，增加了年龄标志，表示页面在主存中已有多久未被访问过，用于页面淘汰。

可以看出，在请求分页虚拟存储系统中，页表的作用主要有三点：获得页框号以实现虚实地址转换，设置各种访问控制位对页面信息进行保护，设置各种标志位来实现相应控制功能(如缺页标志、脏页标志、访问标志、锁定标志和淘汰使用的标志等)。例如，Linux 的页表项包含的主要内容有：– PAGE – PRESENT 位，置为 1 这个页面在内存中是有效的；– PAGE – RWFOE 位 置为 0 表示页面只读，置为 1 表示页面可读可写，没有只写的页面；– PAGE – USER 位 置 1 表示这是用户空间页面，置 0 表示这是内核空间页面；– PAGE – WT 位 置 1 表示页面高速缓存策略是透写(writethrough)，否则为回写(writeback)。前者会立刻把写入高速缓存的数据复制到主存，后者仅当其必须为其他数据腾出空间，需要移出时才(由硬件而不是 Linux)复制到内存；– PAGE – PCD 位 关闭页面高速缓存，该标志对于主存与 I/O 设备的传输操作有用；– PAGE – ACCESSED 位 置 1 表示该页面最近被访问过，可由硬件完成标志

清除工作; - PAGE - DIRTY 位置位表示该页面自从上次该位被清除后已发生修改; - PAGE - 4M 位 表示 pentium 平台上采用 4 MB 的页长; PFN 物理页框或页在交换文件中的信息。

下面来讨论使用快表但页表放在主存的情况下(目前多数系统采用这个方案), 请求分页虚存地址转换过程:当进程被调度到 CPU 上运行时, 操作系统自动把该进程 PCB 中的页表起始地址装入到硬件页表基址寄存器中, 此后, 进程开始执行并要访问某个虚拟地址, MMU 硬件开始工作, 它将完成图 4-24 虚线框内的任务, 对照该图可以看到地址转换过程为:①MMU 接受 CPU 传送过来的虚地址并自动按页面大小把虚地址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 TLB, 立即送出页框号, 并与页内位移拼接成物理地址, 然后, 进行权限检查, 如获通过进程就可以访问物理地址了;④如果不命中 TLB, 以页号为索引搜索进程页表(硬件直接处理方案), 页表的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, 说明访问页面已在主存, 那么, 可送出页框号, 并与页内位移拼接成物理地址, 然后, 进行权限检查, 如获通过进程就可以访问物理地址了;⑥同时要把这个页面的信息装入快表 TLB, 以备再次访问;⑦如果发现页表中对应页面页失效, MMU 就发出一个缺页中断, 请求操作系统进行处理, MMU 工作已经结束;⑧这时由存储管理软件按替换策略进行调页;⑨接着把该页面装入主存, 修改页表, 返回用户进程重新执行被中断的指令。

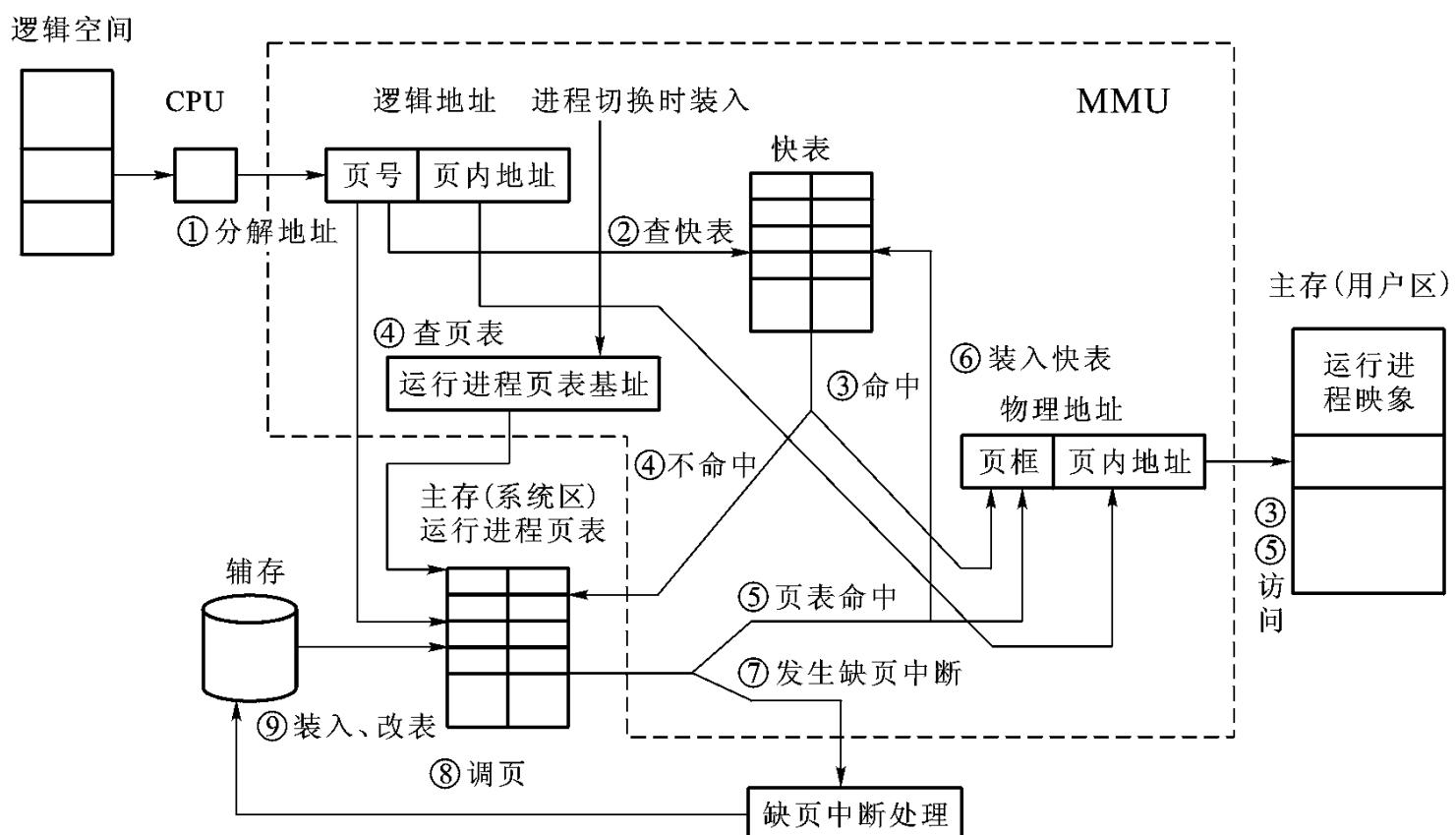


图 4-24 请求分页虚存地址转换过程

MMU 发现缺页并发出缺页中断后, 操作系统存储管理接收控制, 进行缺页中断处理的过程如下:

步 1 查看主存是否有空闲页框, 如有则可以找出一个空闲页框, 修改主存管理表和相应页表项内容, 转步 4;

步 2 如主存中无空闲页框, 按替换算法选择一个淘汰页面, 淘汰页面被写过或修改过吗? 若未则转步 4;若是则转步 3;

步 3 若淘汰页面被写过或修改过, 那么, 把淘汰页的内容写回辅存, 转步 4;

步 4 按该页在辅存储器中的地址把此页面调入, 同时修改进程页表相应项;返回用户进程, 重新执行被中断的指令。

在分页虚拟存储系统中, 由于作业的诸页是根据进程的请求把相应页面装入主存的, 因此, 这种存储系统也称为请求分页虚存管理。IBM/370 系统的 VS/1, VS/2 和 VM/370, Honeywell 6180 的 MULTICS 以及 UNIVAC 系列 70/64 的 VMS 等都采用请求分页虚拟存储系统。请求分页虚拟存储系统有以下优点:作业的程序和数据可以按页分散存放在主存中, 减少了移动的开销, 有效地解决了碎片问题;由于采用请求分页虚存管理, 用户可用的主存空间大大扩展, 既有利于改进主存利用率, 又有利于多道程序运行。其主要缺点是:要有一定硬件支持, 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, 则主存的损失仍然较大。

3. 页面装入策略和清除策略

页面装入策略决定何时把一个页面装入主存, 有两种策略可供选择:请页式(demand paging)调入和预调式(prepaging)调入。

请页式调度是仅当需要访问程序和数据时, 通过发生缺页中断并由缺页中断处理程序分配页框再把所在页面装入主存。那么, 当某个进程第一次执行时, 开始会有许多缺页中断, 随着越来越多的页面装入主存, 根据局部性原理, 大多数未来访问的程序和数据都在最近被装入主存的页面中, 一段时间之后, 缺页中断就会下降到很低的水平, 程序进入相对平稳阶段。这种策略的主要优点是确保只有被访问的页才调入主存, 节省主存空间;缺点是处理缺页中断次数过多和调页的系统开销较大, 由于每次仅调一页, 增加了磁盘 I/O 次数。

预调式调度取进主存的页并不是缺页中断请求的页面, 而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, 在使用之前预先调入主存, 尽量做到进程在访问页面之前已经预先调入该页, 而且, 每次调入若干个页面, 而不是像请页式那样仅调一个页面。由于进程的页面大多数连续存放在辅存储器中, 一次调入多个相连续的页面能减少磁盘 I/O 启动次数, 节省了寻道和搜索时间。但是如果调入的一批页面中多数未被使用, 则效率就很低了, 可见预调页要建立在预测的基础上, 目前所用预调页的成功率在 50% 左右。

许多系统把预调式调度应用在一个进程刚开始执行时(装入进程初始工作集)或每次缺

页中断发生时(同时调入马上要用到的一些页面),对于前一种情况往往要程序员提供所需程序部分的信息。预调式调度和进程的对换不是一回事,当一个进程挂起被换出主存时,它的所有驻留页面将被全部移出;当该进程继续执行时,先前移出的页面都会重新调入主存,但不是由预调页调度来完成的,而是由中级调度的对换程序来处理的。

清除策略是与装入策略相对的,它要考虑何时把一个修改过的页面写回辅存储器,常用的两种方法是:请页式清除和预约式清除。请页式清除是仅当一页被选中进行替换,且之前它又被修改过,才把这个页面写回辅助存储器。预约式清除是对所有更改过的页面,在需要之前就把它们都写回辅助存储器,因此,可以成批进行。两个方法都有缺点,对于预约式清除,写出的页仍然在内存中,直到页替换算法选中该页从内存中移出,它允许成批地把页面写出,但如若刚刚写出了很多页面,在它们被替换前,其中大部分又被更改,那么,预约式清除就毫无意义。对于请页式清除,写出一页是在读进一个新页之前进行的,它要成双操作,虽然仅需要写出一页,但引起进程不得不等待两次 I/O 操作完成,可能会降低 CPU 使用效率。

较好的方法是采用页缓冲技术,其策略如下:仅清除淘汰的页面,并使清除操作和替换操作不必成双进行。在页缓冲中,淘汰了的页面进入两个队列中,修改页面队列和非修改页面队列。在修改页面队列中的页不时地成批写出并加入到非修改页面队列;非修改页面队列中的页面,当它被再次引用时回收,或者淘汰掉以作替换。

4. 页面分配策略

分页式虚拟存储系统排除了主存储器实际容量的约束,能使更多的作业同时多道运行,从而,提高了系统的效率。但缺页中断的处理要付出相当大的代价,由于页面的调入、调出要增加 I/O 的负担而且影响系统效率,因此,应尽可能的减少缺页中断的次数。到目前为止,一直没有讨论如何在相互竞争的多个可运行进程之间分配内存资源,究竟如何为进程分配页框?当出现一次缺页中断时,页面替换算法的作用范围究竟应该局限于本进程的页面还是整个系统的页面?这两个问题涉及到进程驻留页面的管理。

在分页式虚拟存储管理中,不可能也不必要把一个进程的所有页面调入主存,那么,操作系统决定为某进程分配多大的主存空间,需要考虑以下因素:(1)分配给一个进程的空间越小,同一时间处于主存的进程就越多,于是至少有一个进程处于就绪态的可能性就越大,从而,可减少进程对换的时间。(2)如果进程只有一小部分在主存里,即使它的局部性很好,缺页中断率还会相当高。(3)因为程序的局部性原理,分配给一个进程的主存超过一定限度后,再增加主存空间,也不会明显降低进程的缺页中断率。基于这些因素,在请页式系统中,可采用两种策略分配页框给进程:固定分配和可变分配。

如果进程生命周期中,保持页框数固定不变,称页面分配为固定分配。在进程创建时,根据进程类型和程序员的要求决定页框数,只要有一个缺页中断产生,进程就会有一页被替

换。如果进程生命周期中,分得的页框数可变,称页面分配为可变分配。当进程执行的某一段阶段缺页率较高,说明进程程序目前的局部性较差,系统可多分些页框以降低缺页率,反之说明进程程序目前的局部性较好,可以减少分给进程的页框数。固定分配策略缺少灵活性,相比之下可变分配的性能会更好些,被许多操作系统所采用。采用可变分配策略的困难在于操作系统要经常监视活动进程的行为和进程缺页中断率的情况,这会增加操作系统的开销。

在进行页面替换时,也可采用两种策略: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,称为全局页面替换算法,它可以在可运行进程之间动态地分配页框。如果页面替换算法的作用范围局限于本进程,称为局部页面替换算法,它实际上需要为每个进程分配固定的页框。在通常情况下,尤其是驻留页面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。如果使用局部算法,那么,即使有大量的空闲页框存在,工作集的增长仍然会导致颠簸;如果工作集收缩了,局部算法又会浪费内存。但是使用全局算法时,系统必须不断地确定应该给每个进程分配多少主存,这是比较困难的。

固定分配往往和局部替换策略配合使用,每个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。这种策略的主要难点在于:应给每个进程分配多少个页框?给少了,缺页中断率高;给多了,会使主存中能同时执行的进程数减少,进而造成处理器空闲和其他设备空闲,并把许多时间花费在对换上。采用固定分配算法时,系统把可供分配的页框分配给进程,可采用下列方式:(1)平均分配,不论进程大小,每个进程得到的页框数相等,这样做会造成大进程缺页中断率高。(2)按比例分配,大进程获得页框多,小进程获得页框少。(3)优先权分配,可以把页框分成两类,一类按比例分配,另一类根据进程优先权适当增加份额。

可变分配往往和全局替换策略配合使用,这是采用得较多的一种分配和替换算法。先为系统中的每个进程分配一定数目的页框,操作系统自己保留若干空闲页框。当一个进程发生缺页中断时,从系统空闲页框中选一个给进程,添加到它的驻留集中,于是可把缺页调入这个页框中,这样产生缺页中断的进程的主存空间会逐渐增大。凡是产生缺页中断的进程都能获得新的页框,有助于减少系统的缺页中断总次数,直到系统拥有的空闲页框耗尽,才会从主存中选择一页淘汰,该页可以是主存中任一进程的页面,这样又会使那个进程的页框数减少,缺页中断率上升。这个方法的难点在于选择哪个页面进行替换,选择范围是主存中除锁定外的全部页框,应用一种淘汰策略选页时,并没有规则可以确定哪一个进程会失去一页,因而,如果选择了一个进程,这个进程工作集的减少会严重影响它的执行,那么,这个选择就不是最佳的。解决可变分配全局替换性能问题的一种方法是使用页缓冲,按这种方法选择替换哪一页变得不太重要,因为,如果在下次重写这些页面之前访问到的话,相应页面是可以回收的。

可变分配配合局部替换可以克服可变分配全局替换的缺点,其实现要点如下:(1)新进程装入主存时,根据应用类型、程序要求,分配给一定数目页框,可用请页式或预调式完成这个分配。(2)当产生缺页中断时,从产生缺页中断的进程的驻留集中选一个页面替换。(3)不时重新评价进程的分配,增加或减少分配给进程的页框以改善系统总的性能。

5. 页面替换策略

实现虚拟存储器能给用户提供一个容量很大的存储空间,但当主存空间已装满而又要装入新页时,必须按一定的算法把已在主存的一些页面调出去,这个工作称页面替换。所以,页面替换就是用来确定应该淘汰哪页的算法,也称淘汰算法。算法的选择是很重要的,选用了一个不适合的算法,就会出现这样的现象:刚被淘汰的页面又立即要用,因而,又要把它调入,而调入不久再被淘汰,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至于大部时间都花在来回调度页面上。这种处理器花费大量时间用于对换页面而不是执行计算任务的现象叫做“抖动”(Thrashing),又称“颠簸”,一个好的调度算法应减少和避免抖动现象。

替换策略要处理的是:当把一个新的页面调入主存时,选择已在主存的哪个页面作替换。由于下列因素,使得替换策略变得比较困难:(1)分给每个活跃进程多少页框?(2)页面替换时,仅限于缺页中断进程还是包括主存中所有页面?(3)在被考虑的页面集合中,选出哪个页面进行替换?为了衡量替换算法的优劣,考虑在固定空间的前提下讨论各种页面替换算法。这一类算法是假定每道作业都给固定数的主存分配,即每道作业占用的主存块数不允许页面替换算法加以改变。在这样的假定下,怎样来衡量一个算法的好坏呢?先来叙述一个理论算法。假定作业 p 共计 n 页,而系统分配给它的主存块只有 m 块(m, n 均为正整数,且 $1 \leq m \leq n$),即最多主存中只能容纳该作业的 m 页。如果作业 p 在运行中成功的访问次数为 S (即所访问的页在主存中),不成功的访问次数为 F (即缺页中断次数),则总的访问次数 A 为:

$$A = S + F$$

又定义:

$$f = F / A$$

则称 f 为缺页中断率。影响缺页中断率 f 的因素有:

- 主存页框数。作业分得的主存块数多,则缺页中断率就低,反之,缺页中断率就高。
- 页面大小。如果划分的页面大,则缺页中断率就低,否则,缺页中断率就高。
- 页面替换算法。替换算法的优劣影响缺页中断次数。
- 程序特性。程序编制的方法不同,对缺页中断的次数有很大影响,程序的局部性要好。

例如:有一个程序要将 128×128 的数组置初值为“0”。现假定分给这个程序的主存块数只有一块,页面的尺寸为每页 128 个字,数组中的元素每一行存放在一页中,开始时第一

页在主存。若程序如下编制：

```
Var A: array[1..128] of array [1..128] of integer;
  For j := 1 to 128
    do for i := 1 to 128
      do A[i][j] := 0
```

则每执行一次 $A[i][j] := 0$ 就要产生一次缺页中断，于是总共要产生 $(128 \times 128 - 1)$ 次缺页中断。如果重新编制这个程序如下：

```
Var A: array[1..128] of array [1..128] of integer;
  for i := 1 to 128
    do for j := 1 to 128
      do A[i][j] := 0
```

那么，总共只产生 $(128 - 1)$ 次缺页中断。

显然，虚拟存储器的效率与程序的局部性程度密切相关，局部性的程度因程序而异，一般说，总希望编出的程序具有较好的局部性。这样，程序执行时可经常集中在几个页面上进行访问，减少缺页中断率。

同样存储容量与缺页中断次数的关系很大。从原理上说，提供虚拟存储器以后，每个作业只要能分到一块主存储空间就可以执行，从表面上看，这增加了可同时运行的作业个数，但实际上却是低效率的。试验表明当主存容量增大到一定程度，缺页中断次数的减少就不明显了。大多数程序都有一个特定点，在这个特定点以后再增加主存容量收效就不大，这个特定点是随程序而变的，试验分析表明，对每个程序来说，要使其有效的工作，它在主存中的页面数应不低于它的总页面数的一半。所以，如果一个作业总共有 n 页，那么，只有当主存至少有 $n/2$ 块页框时才让它进入主存执行，这样可以使系统获得高效率。

下面介绍页面替换算法。一个理想的替换算法是：当要调入一页而必须淘汰一个旧页时，所淘汰的页应该是以后不再访问的页或距现在最长时间后再访问的页。这样的调度算法使缺页中断率为最低。然而，这样的算法是无法实现的，因为在程序运行过程中无法对以后要使用的页面做出精确的断言。不过，这个理论上的算法可以用来作为衡量各种具体算法的标准。这个算法是由 Belady 提出来的，所以，叫做 Belady 算法，又叫做最佳替换算法（Optimal）。

Belady 算法是一种理想化的页面调度算法，下面分别介绍几个比较典型又实用的页面调度算法。

1) 随机页面替换算法

要淘汰的页面是由一个随机数产生程序所产生的随机数来确定。选择一个不常使用的

页面会使系统性能较好,但这种调度算法做不到这一点,虽很简单但效率却低,一般不采用。

2) 先进先出页面替换算法(FIFO)

先进先出调度算法是一种低开销的页面替换算法,基于程序总是按线性顺序来访问物理空间这一假设。这种算法总是淘汰最先调入主存的那一页,或者说在主存中驻留时间最长的那一页(锁定的页面除外),认为驻留时间最长的页不再被使用的可能性较大。这种算法可以采用不同的技术来实现。一种实现方法是系统中设置一张具有 m 个元素的页号表:

$$P[0], P[1], \dots, P[m-1]$$

所组成的一个数组,其中,每个 $P[i]$ ($i = 0, 1, \dots, m-1$)存储一个在主存中的页面的页号。假设用指针 k 指示当前调入新页时应淘汰的那一页在页号表中的位置,则淘汰的页号应是 $P[k]$ 。每当调入一个新页后,执行

$$P[k] := \text{新页的页号};$$

$$k := (k + 1) \bmod m;$$

假定主存中已经装了 m 页, k 的初值为0,那么,第一次淘汰的页号应为 $P[0]$,而调入新页后 $P[0]$ 的值为新页的页号, k 取值为1; \dots ;第 m 次淘汰的页号为 $P[m-1]$,调入新页后, $P[m-1]$ 的值为新页的页号, k 取值为0;显然,第 $m+1$ 次页面淘汰时,应淘汰页号为 $P[0]$ 的页面,因为,它是主存中驻留时间最长的那一页。

这种算法较易实现,对具有线性顺序特性的程序比较适用,而对其他特性的程序效率不高,因为在主存中驻留时间最长的页面未必是最长时间以后才使用的页面,很可能有最近要被访问的页。也就是说,如果某一个页面要不断地和经常地被使用,采用FIFO算法,在一定的时间以后就会变成驻留时间最长的页,这时若把它淘汰了,可能立即又要用,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至3倍。

另一个简单的实现算法是引入指针链成队列,只要把进入主存的页面按时间的先后次序链接,新进入的页面从队尾入队,淘汰总是从队列头进行。

3) 最近最少用页面替换算法(LRU, Least Recently Used)

最近最少用页面替换算法是一种通用的有效算法,被操作系统、数据库管理系统和专用文件系统广泛采用。该算法淘汰的页面是在最近一段时间里较久未被访问的那一页。它是根据程序执行时所具有的局部性来考虑的,即那些刚被使用过的页面,可能马上还要被使用,而那些在较长时间里未被使用的页面,一般说可能不会马上使用到。

为了能比较准确地淘汰最近最少使用的页,从理论上来说,必须维护一个特殊的队列——页面淘汰队列。该队列中存放当前在主存中的页号,每当访问一页时就调整一次,使队列尾总指向最近访问的页,队列头就是最近最少用的页。显然,发生缺页中断时总淘汰队列头所指示的页;而执行一次页面访问后,需要从队列中把该页调整到队列尾。

例如,给某作业分配了三块主存,该作业依次访问的页号为:4,3,0,4,1,1,2,3,2。于是

当访问这些页时, 页面淘汰序列的变化情况如下:

| 访问页号 | 页面淘汰序列 | 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 | |

从实现角度来看, LRU 算法的操作复杂, 代价高。因此, 在实现时往往采用模拟的方法。

第一种模拟方法通过设置标志位来实现, 又叫最近没有使用页面替换算法 NUR (Not Recently Used)。此方法给每一页设置一个引用标志位 R, 每次访问某一页时, 由硬件将该页的标志 R 置 1, 隔一定的时间 t 将所有页的标志位 R 均清 0。在发生缺页中断时, 从标志位 R 为 0 的那些页中挑选一页淘汰。在挑选到要淘汰的页后, 也将所有其他页的标志 R 清 0。这种实现方法开销小, 但 t 的大小不易确定而且精确性差。 t 大了, 缺页中断时所有页的标志 R 值均为 1; t 小了, 缺页中断时, 可能所有页的 R 值均为 0”, 同样很难挑选出应该淘汰的页面。

第二种模拟方法是为每个页设置一个多位寄存器 r 。当页面被访问时, 对应的寄存器的最左边位置 1; 每隔时间 t , 将 r 寄存器右移一位; 在发生缺页中断时, 找最小数值的 r 寄存器对应的页面淘汰。

例如, r 寄存器共有四位, 页面 P0、P1、P2 在 T1、T2、T3 时刻的 r 寄存器内容如下:

| 页面 | 时刻 | | |
|----|------|------|------|
| | T1 | T2 | T3 |
| P0 | 1000 | 0100 | 1010 |
| P1 | 1000 | 1100 | 0110 |
| P2 | 0000 | 1000 | 0100 |

在时刻 T3 时, 应该淘汰的页面是 P2。这是因为同 P0 比较, 它不是最近被访问的页面; 同 P1 比较, 虽然它们在时刻 T3 都没有被访问, 且在时刻 T2 都被访问过, 但在时刻 T1 时 P2 没有被访问。

显然, 第二种模拟方法优于第一种模拟方法, 它又被称为“老化算法”。老化算法可以比

较好地模拟运行进程的当前工作集,使得系统达到比较好的性能。有关工作集模型的讨论参见下节。

第三种模拟方法是为每个页面设置一个多位计数器,又叫最不常用页面替换算法 LFU (Least Frequently used)。每当访问一页时,就使它对应的计数器加 1。当发生缺页中断时,可选择计数值最小的对应页面淘汰,并将所有计数器全部清 0。显然,它是在最近一段时间里最不常用的页面。这种算法实现不难,但代价太高,而且选择多大的 t 最适宜也是个难题。

第四种模拟方法是为每个页面设置一个多位计时器,每当页面被访问时,系统的绝对时间记入计时器。比较各页面的计时器的值,选最小值的未使用的页面淘汰,因为,它是最“老”的未使用的页面。

4) 第二次机会页面替换算法

FIFO 算法可能会把经常使用的页面淘汰掉,可以对 FIFO 算法进行改进,把 FIFO 算法与页表中的“引用位”结合起来使用,算法可实现如下:首先检查 FIFO 中的队首页面(这是最早进入主存的页面),如果它的“引用位”是 0,那么,这个页面既老又没有用,选择该页面淘汰;如果它的“引用位”是 1,说明虽然它进入主存较早,但最近仍在使用。于是把它的“引用位”清成 0,并把这个页面移到队尾,把它看作是一个新调入的页,再给它一个机会。这一算法称为第二次机会(second chance)算法,其含义是最先进入主存的页面,如果最近还在被使用的话,仍然有机会作为像一个新调入页面一样留在主存中。

5) 时钟页面替换算法(Clock Policy)

如果利用标准队列机制构造 FIFO 队列,第二次机会页面调度算法将可能产生频繁的出队入队,实现代价较大。因此,往往采用循环队列机制构造页面队列,这样就形成了一个类似于钟表面的环形表,队列指针则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的得名。时钟页面替换算法与第二次机会算法本质上没有区别,仅仅是实现方法不同并作了改进,仍然要使用页表中的“引用位”,把作业已调入主存的页面链接成循环队列,用一个指针指向循环队列中下一个将被替换的页面,算法的实现要点如下:

- 一个页面首次装入主存时,其“引用位”置 1。
- 在主存中的任何一个页面被访问时,其“引用位”置 1。
- 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”清成 0,并跳过这个页面;把所遇到的“引用位”是 0 的页面淘汰掉,指针推进一步。
- 扫描循环队列时,如果遇到的所有页面的“引用位”为 1,指针就会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,并淘汰掉这一页,然后,指针推进一步。

图 4-25 给出了时钟页面替换算法的一个例子。当发生缺页中断时,将要进入主存的

页面是 page727, 指针指向的是 page45(在页框 2 中)。时钟页面替换算法执行过程如下: page45 的“引用位”是 1, 故它不能被淘汰掉, 仅把其“引用位”清 0, 指针推进。同样道理, page191(在页框 3 中)也不能被替换, 把其“引用位”清 0, 指针继续推进。在下一页即 page556(在页框 4 中), 它的“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。

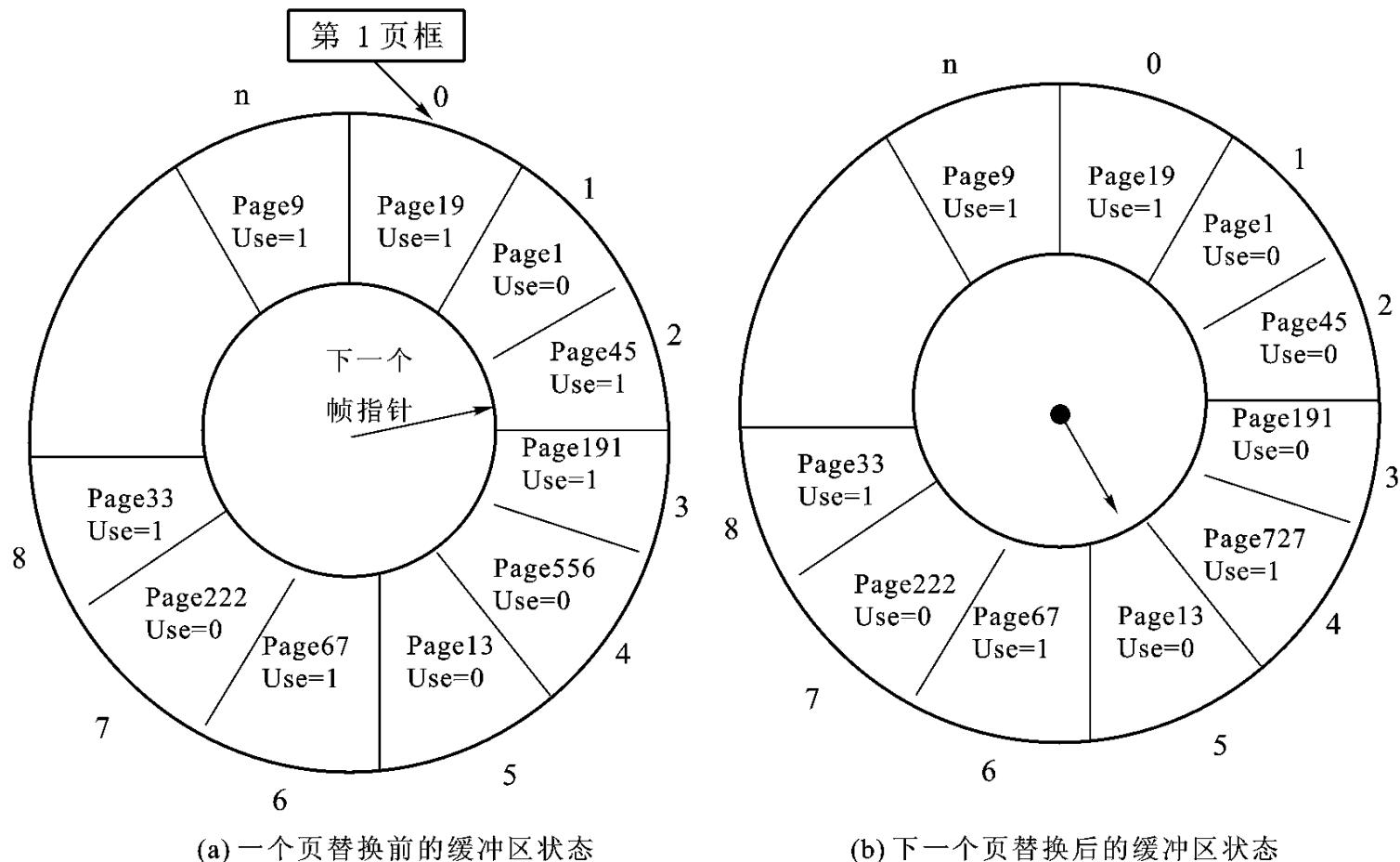


图 4-25 时钟页面替换算法

淘汰一个页面时, 如果该页面已被修改过, 必须将它重新写回磁盘; 但如果淘汰的是未被修改过的页面, 就不需要写盘操作了。这样看来淘汰修改过的页面比淘汰未被修改过的页面开销要大。如果把页表中的“引用位”和“修改位”结合起来使用可以改进时钟页面替换算法, 它们一共组合成四种情况:

- (1) 最近没有被引用, 没有被修改 ($r = 0, m = 0$)
- (2) 最近被引用, 没有被修改 ($r = 1, m = 0$)
- (3) 最近没有被引用, 但被修改 ($r = 0, m = 1$)
- (4) 最近被引用过, 也被修改过 ($r = 1, m = 1$)

于是, 改进的时钟算法可如下执行:

步 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 $r = 0, m = 0$ 的页面作为淘汰页面。

步 2: 如果步 1 失败, 再次从原位置开始, 查找 $r = 0$ 且 $m = 1$ 的页面, 把遇到的第一个这样的页面作为淘汰页面, 而在扫描过程中把指针所扫过的页面的“引用位” r 置 0。

步 3: 如果步 2 失败, 指针再次回到了起始位置。由于此时所有页面的“引用位” r 均已为 0, 再转向步 1 操作, 必要时再做步 2 操作, 这次一定可以挑出一个可淘汰的页面。

改进的时钟页面替换算法就是扫描循环形队列中的所有页面, 寻找一个既没有被修改且最近又没有被引用过的页面, 把这样的页面挑出来作为首选页面淘汰是因为没有被修改过, 淘汰时不用把它写回磁盘。如果第一步没找到这样的页面, 算法再次扫描循环队列, 欲寻找一个被修改过但最近没有被引用过的页面; 虽然淘汰这种页面需写回磁盘, 但依据程序局部性原理, 这类页面不会被立刻再次使用。如果第二步也失败了, 则所有页面已被标记为最近未被引用, 可进入第三步扫描。Macintosh 的虚存管理采用了这种策略, 其主要优点是没有被修改过的页面会被优先选出来, 淘汰这种页面时不必写回磁盘, 从而节省时间。但查找一个淘汰页面可能会经过多轮扫描, 算法的实现开销较大。

UNIX SVR4 使用改进的 Clock 算法, 称双指针 Clock 算法。其实现思想如下: 主存中每个合法的页面(指不锁住, 可淘汰的页面)都有“引用位”相连; 当一个页面被替换进主存时, “引用位”置 0; 当一个页面被引用时, “引用位”置 1; 系统设置了两个时钟指针, 称前指针和后指针。每隔固定时间间隔, 两个指针都扫描一遍: 第一遍, 前向指针依次扫过合法页面并把“引用位”置 0; 再延迟一定时间之后, 后指针依次扫描合法页面, 若该页面的“引用位”为 1, 表明这个期间页面被引用过, 则跳过该页面, 继续扫描, 若该页面的“引用位”为 0, 表明期间页面未被引用过, 那么, 此页面可作为淘汰的候选者。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, 可根据物理空间大小为这两个参数设置默认值, 速度参数可以改变, 以满足条件的变化。当自由的存储空间缩小时, 两个指针移动速度要加快, 以释放更多的页面。

下面给出一个例子, 分别用 Opt、FIFO、LRU 和 Clock 替换算法来计算缺页中断次数和被淘汰的页面, 并对它们的性能作简单的比较。假设采用固定分配策略, 进程分得三个页框, 它在执行中按下列次序引用 5 个独立的页面: 2 3 2 1 5 2 4 5 3 2 5 2。图 4-26 给出了四种算法的计算过程和结果。

从图中可以看出, Opt 共产生 3 次缺页中断, 第一次淘汰 page1, 因为它以后不再使用了; 第二次淘汰 page2, 它要在最远的将来才被再次使用; 第三次淘汰 page4, 它以后不再被使用。LRU 共产生 4 次缺页中断, 它将内存中最长时间内没有被引用的页面淘汰掉。根据局部性原理, 最长时间内没有被引用的页面应该是最近的将来最不可能被引用到的页面, 所以淘汰的页面依次为 page3、page1、page2、page4, 图中顶上一行为最近被访问的页面, 最低下一行为

最近最少被访问的页面。FIFO 共产生 6 次缺页中断, 它认为进入内存时间最长的页面, 目前最不可能被引用, 应该被淘汰掉, 所淘汰的页面依次为 page2、page3、page1、page5、page2、page4, 图中顶上一行为最先进入主存的页面, 最低下一行为最近进入主存的页面。Clock 共产生 5 次缺页中断, 图中 * 表示相应页面的“引用位”为 1, 箭头 → 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”为 1, 所以指针绕过一圈并指向 page2, 故 page5 替换了 page2, 同时 page3 和 page1 的“引用位”被置 0; 第二次引用 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, 因为, 此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 替换了 page5; 当第三次引用 page2 时, 循环队列中三个页面 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 性能最差, Opt 性能最好, 而 Clock 与 LRU 十分接近。

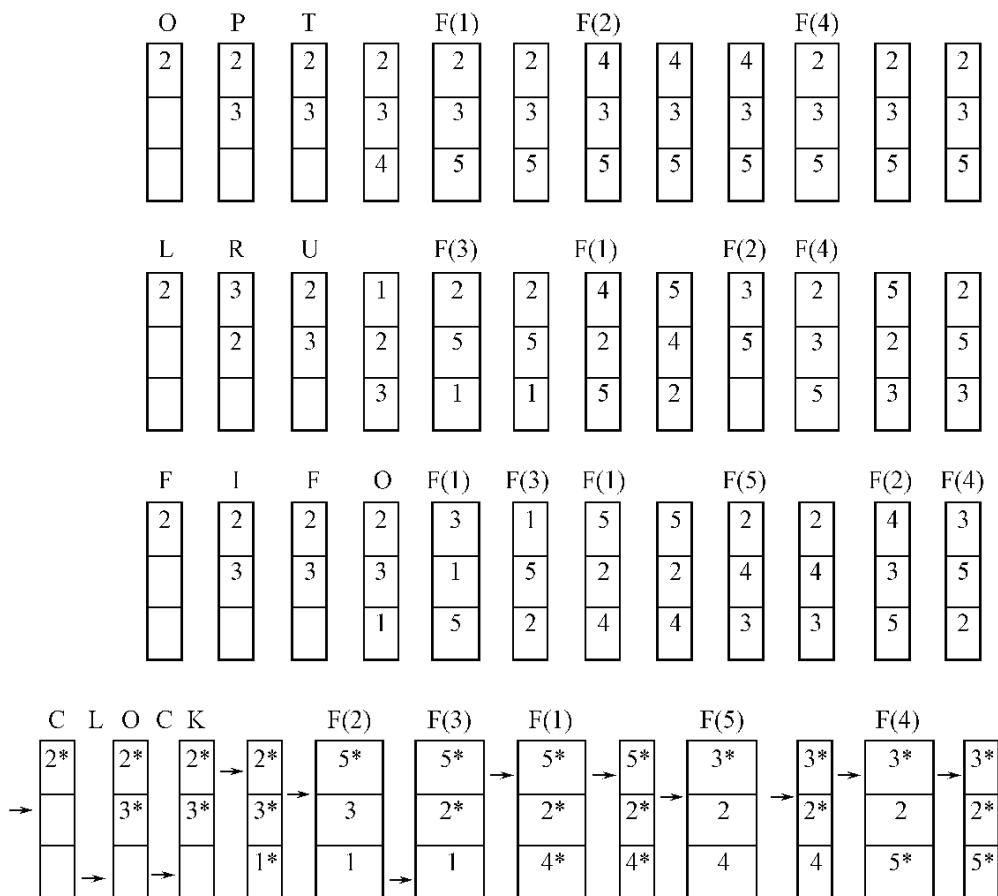


图 4-26 四种算法计算的结果

图 4-27 是用上述四种算法计算的结果曲线。假设分配给进程的页框数是固定的, 运行的 FORTRAN 程序共有 0.25×10^6 次页面引用, 页面大小为 256 个字。在这个实验中, 分配

给进程的页框数分别为 6、8、10、12 和 14。当分配给进程的页框数比较少时, 四种算法的差距明显, 且 FIFO 所产生的缺页中断基本上是 Opt 的 2 倍, Clock 则比较接近于 LRU。

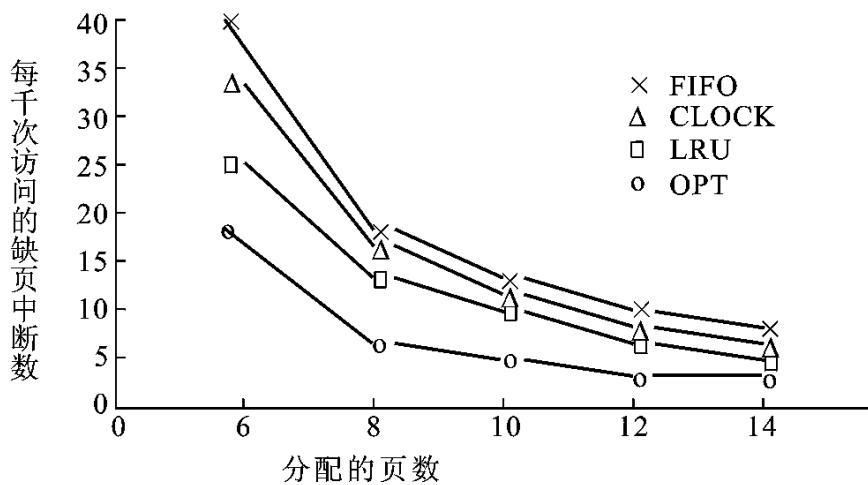


图 4-27 四种算法计算的结构曲线

6. 请求分页虚拟存储管理的几个设计问题

(1) 页面大小

1) 从页表大小考虑。在虚拟空间大小一定的前提下, 不难看出页面大小的变化对页表大小的影响。如果页面较小, 虚拟空间的页面数就增加, 页表也随之扩大。由于每一个作业都必须有自己的页表, 因此, 为了控制页表所占的主存量, 页面的尺寸还是较大一点为好。

2) 从主存利用率考虑。主存是以块为分配单位的, 作业程序一般不可能正好划分为整数倍的页面。于是, 作业的最后一个页面进入主存时, 总会产生内部碎片, 平均起来一个作业将造成半块的浪费。为了减少内部碎片, 页面的尺寸还是小一点为好。

3) 从读写一个页面所需的时间考虑。作业都存放在辅助存储器上, 从磁盘读入一个页面的时间包括等待时间(移臂时间 + 旋转时间)和传输时间, 通常等待时间远大于传输时间。显然, 加大页面的尺寸, 有利于提高 I/O 的效率。

4) 最佳页面尺寸。目前采用页式虚拟存储管理的计算机系统中, 页面大小大多选择在 512 字节到 4K 字节之间。之所以如此, 是从减少内部碎片和页表耗费的存储空间两个角度出发推导出来的。

假定 S 表示用户作业程序的字节数平均长度, P 表示以字节为单位的页面长度, 且有 $S > P$, 而每个页表项需要 e 个字节。则每个作业的页表长度为 S/P , 占用了 Se/P 个字节的页表空间, 在作业的最后一页, 假定浪费的主存平均为 $P/2$ 个字节。则对一个作业而言, 有:

$$\text{浪费的存储字} = \text{页表使用的主存空间} + \text{内部碎片} = Se/P + P/2$$

在页面比较小时页表占用空间多(因 Se/P 较大), 在页面比较大时内部碎片浪费多(因

$P/2$ 较大), 那么, P 的最优值一定在中间的一个点。现在对 P 求一阶导数并令其为 0, 得到方程:

$$-\frac{Se}{P^2} + \frac{1}{2} = 0$$

假如仅考虑碎片浪费的和页表占用的主存, 那么, 从这个方程可以得出最优页面尺寸的为:

$$P = \sqrt{2Se}$$

时, 浪费的存储字节最少, 称 P 为最佳页面尺寸。对于 $S = 128$ KB, 每个页表项 $e = 8B$ 时, 最优页面尺寸是 1448 字节。考虑到其他因素, 如磁盘速度等, 实际可使用 1 KB 或 2 KB 长的页面。大部分商用计算机使用的页面尺寸在 512B 至 4 KB 之间。下面是一些著名操作系统选择的页面尺寸:Atlas 为 512 字(每字 48 位)、IBM370 系列机为 2048 或 4096 字节、VAX 为 512 字节、IBM AS/400 为 512 字节、Motorola 68040(Macintosh) 为 4096 字节、Pentium 为 4096 字节或 4 兆字节、MIPS R4000 提供从 4096 字节至 16 兆字节共 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。

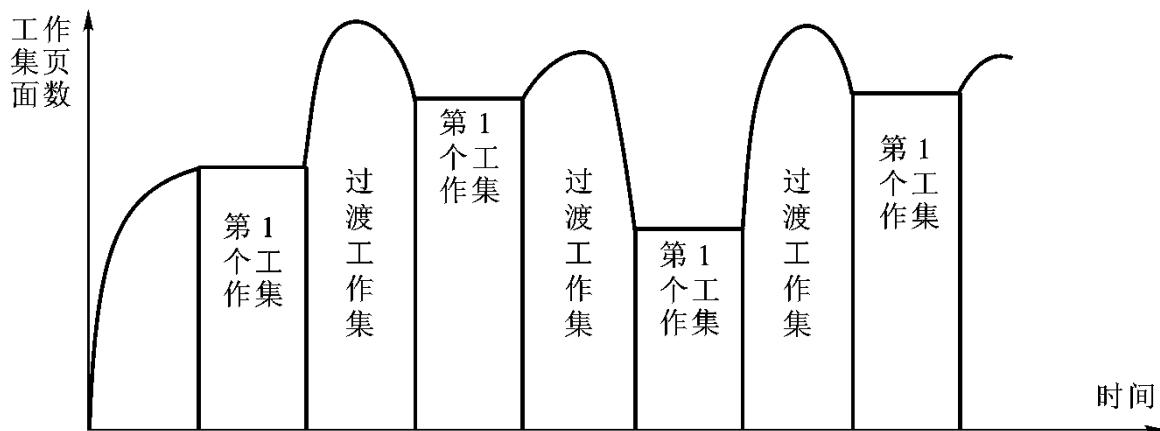
(2) 工作集模型

从充分地共享系统资源这一角度出发, 当然希望主存中的作业数越多越好, 但是, 从保证作业顺利执行、使 CPU 能够有效地得到利用的角度出发, 就应该限制主存中的作业数, 以避免频繁地调入/调出页面, 导致系统的抖动。为此, P.J. Denning 认为, 应该将处理机调度和主存管理结合起来进行考虑, 并在 1968 年提出了工作集(working set)模型, 它对于虚拟存储器的设计有很大影响。Denning 当时提出的工作集概念是“为确保每个进程每一时刻能够执行下去, 在物理存储器中必须有的最少页面数”。但也有的文献所用工作集概念稍有不同是指“在未来的时间间隔内, 一个进程运行时所需访问的页面集”。

从程序的局部性原理可知, 在一段时间内作业的运行只涉及某几个页面, 可以想象, 随着作业的执行, 工作集不断变化, 所包含的页面数时而增多, 时而减少。当执行进入某一个新阶段时, 由于过渡, 这一段时间的工作集所包含的页面会出现剧烈的变动, 如图 4-28 所示。

从图上看, 当作业开始运行而一次一次把页面从辅存调入主存时, 系统将分配给它很多主存块。当对存储的需求暂时趋于平稳状态时, 就形成了第一个工作集。随着时间的推移, 作业的访问将进到下一个工作集, 中间出现一个用曲线表示的过渡工作集。最初曲线是上升趋势, 这是因为程序运行促使新页面的调入, 工作集膨胀。一旦下一个工作集逐渐稳定时, 就可以淘汰不必要的页面, 而形成第二个工作集。所以, 工作集的一次转移, 会出现先上升后下降的曲线。

由此可见, 如果在一段时间内, 作业占用的主存块数目小于工作集, 运行过程中就会不断出现缺页中断, 导致系统的抖动。所以, 为了保证作业的有效运行, 在相应时间段内就应



该根据工作集的大小分配给它主存块,以保证工作集中所需要的页面能够进入主存。推而广之,为了避免系统发生抖动,就应该限制系统内的作业数,使它们的工作集总尺寸不超过主存块总数。

一个进程运行在时刻 $t - \Delta$ 到时刻 t 之间的时间间隔内所访问的页面的集合称为该进程在时刻 t 的工作集,用 $W(t, \Delta)$ 表示。变量 Δ 称为“工作集窗口尺寸”,可以通过窗口来观察进程的行为。通常还把工作集中所包含的页面数目称为“工作集尺寸”,记为 $|W(t, \Delta)|$ 。如果系统能随 $|W(t, \Delta)|$ 的大小来分配主存块的话,就既能有效的利用主存,又可以使缺页中断尽量少地发生,或者说程序要有效运行,其工作集必须在主存中。

现在来考察二元函数 W 的两个变量,首先,工作集 W 是 t 的函数,即随时间不同,工作集也不同。其一是不同时间的工作集所包含的页面数可能不同(工作集尺寸不同);其二是不同时间的工作集所包含的页面可能不同(不同内容的页面)。其次,工作集 W 又是工作集窗口尺寸 Δ 的函数,而且工作集尺寸 $|W(t, \Delta)|$ 是工作集窗口尺寸 Δ 的非递减函数。如图 4-29 所示,其中列出了进程的引用序列, * 表示这个时间单位里工作集没有发生改变。从图中可以看出,工作集窗口尺寸越大,工作集就越大,产生缺页中断的频率就越低。

由于无法预知一个程序在最近的将来会访问哪些页面,只好用最近的过去在 Δ 时间间隔内访问过的页面作为实际工作集的近似。正确选择工作集窗口尺寸的大小对系统性能有很大影响,如果 Δ 过大,甚至把作业地址空间全包括在内,就成了实存管理;如果 Δ 过小,则会引起频繁缺页,降低了系统的效率。对许多程序来说,工作集大小相对稳定的时期和工作集大小快速改变的时期是交替存在的。当一个进程刚开始执行时,随着进程访问新页面,逐渐构造工作集。根据局部性原理,进程最终会稳定在页面的某个集合上,这就是稳定期,而随后的过渡期表明进程向一个新的局部转移。在过渡期中,仍然会有一些上一个稳定期中的页保留在窗口中,这些页在新页面访问时,会使工作集大小产生波动,但经过新页面被访问后,工作集大小会减少直到又包括这个新的局部中的页面。

| 页面访问序列 | 窗口尺寸 | | | |
|--------|-------|----------|-------------|----------------|
| | 2 | 3 | 4 | 5 |
| 24 | 24 | 24 | 24 | 24 |
| 15 | 15 24 | 15 24 | 15 24 | 15 24 |
| 18 | 18 15 | 18 15 24 | 18 15 24 | 18 15 24 |
| 23 | 23 18 | 23 18 15 | 23 18 15 24 | 23 18 15 24 |
| 24 | 24 23 | 24 23 18 | * | * |
| 17 | 17 24 | 17 24 23 | 17 24 23 18 | 17 24 23 18 15 |
| 18 | 18 17 | 18 17 24 | * | * |
| 24 | 24 18 | * | * | * |
| 18 | 18 24 | * | * | * |
| 17 | 17 18 | * | * | * |
| 17 | 17 | * | * | * |
| 15 | 15 17 | 15 17 18 | 15 17 18 24 | * |
| 24 | 24 15 | 24 15 17 | * | * |
| 17 | 17 24 | * | * | * |
| 24 | 24 17 | * | * | * |
| 18 | 18 24 | 18 24 17 | * | * |

图 4-29 进程工作集

可以通过工作集概念来确定驻留集的大小:①监视每个进程的工作集。②定期地从一个进程驻留集中删去那些不在工作集中的页。③仅当一个进程的工作集在主存时,进程才能执行。

正确的策略并不是消除缺页现象,而应使缺页间隔时间保持在合理水平。当此间隔过小时,应增加其页框数,过大则应减少分给进程的页框数,一个更直接的改善系统性能的方法是使用页面故障率(Page Fault Frequency),或者叫做 PFF 分配算法。

前面曾经讨论过,包括 LRU 在内的一大类页面替换算法的故障率随着分配的页框数的增加而减少,如图 4-30 所示。虚线 A 对应的是一个过高的故障率,发生故障的进程将被分配更多的页框以减少故障率;虚线 B 对应的是一个过低的故障率,可以得出结论分配给这个进程的主存太多了,可以收回一些页框。可以看出,PFF 试图把页面故障率保持在一个可以接受的范围内。如果发现主存中进程太多以至于不可能使所有进程的故障率都低于 A,那么,就必须从主存中交换出去某些进程,把它们的页框分配给余下的进程或放进一个空闲页框池中供后面发

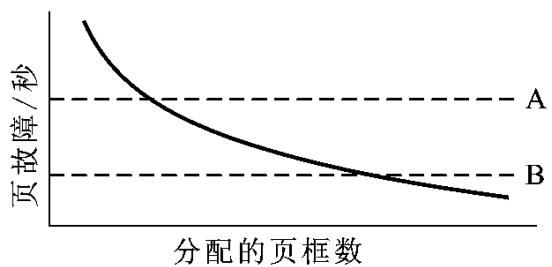


图 4-30 页面故障率是分配的页框数的函数

生页面故障时使用。从主存中移去一个进程的决定实际上是一种负载控制, 它表明即使在分页系统中, 交换仍然是需要的, 不同的只是现在交换是为了降低潜在的对主存的需求, 而不是为了立刻使用收回的主存页框。

(3) 页面交换区

替换算法常常要挑选一个页面淘汰出主存, 但是这个被淘汰出去的页面可能很快又要使用, 被重新装入主存。因而, 操作系统必须把被淘汰的页面内容保存在磁盘的特殊区域, 例如, UNIX/Linux 使用交换区(可以是盘的一个分区称交换设备, 也可以是一个或多个文件称交换文件)临时保存页面。系统初始化时, 保留一定盘空间作交换区, 初始化内容为空, 不能被文件系统使用。当一个进程被启动时, 留出与这个进程一样大的交换空间块, 进程撤销时, 释放占用的交换空间。

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它们。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进行复制。

交换区映射表一般都设计为高效的数据结构, 并驻留在主存的内核区中。但进程的页面被交换出来后, 并不都放在交换区, 通常只有数据页面和堆栈页面保留在交换区。

(4) 锁定主存页

当一个进程执行系统调用拟读入一批数据到其地址空间的缓冲区, 等待 I/O 完成时, 该进程暂停, 其他进程可继续执行, 而此时其他进程产生了缺页中断。如果采用全局替换策略, 有这种可能性, 包含 I/O 缓冲区的页面选中淘汰, 而假如一个 I/O 设备正在与该页面进行 DMA 方式的数据交换, 把该页面淘汰出去造成部分数据写入原页面, 部份数据写入新页面。一种解决方法是锁住正在做 I/O 操作的内存页面, 保证它不被移出, 可以在页表项中增加锁定位来做到这一点。另一种解决方法是在核心内的缓冲区完成 I/O, 然后, 把数据复制到用户的页面。

(5) 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 和 Windows 等许多操作系统所采用, 它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执行写操作时, 会产生一个“写时复制”中断, 操作系统处理这个中断信号, 为该进程分配一个空闲页框, 将要写的页进行复制, 且修改相应的进程页表项, 当进程重新执行写页面操作时指令被顺利执行, 图 4-31 是写时复制的示意。可见操作系统采用写时复制技术后, 就可以延迟到修改时才对共享页面做出副本, 免除了只读页的复制, 从而节省了大量

页面复制操作和副本占用空间。

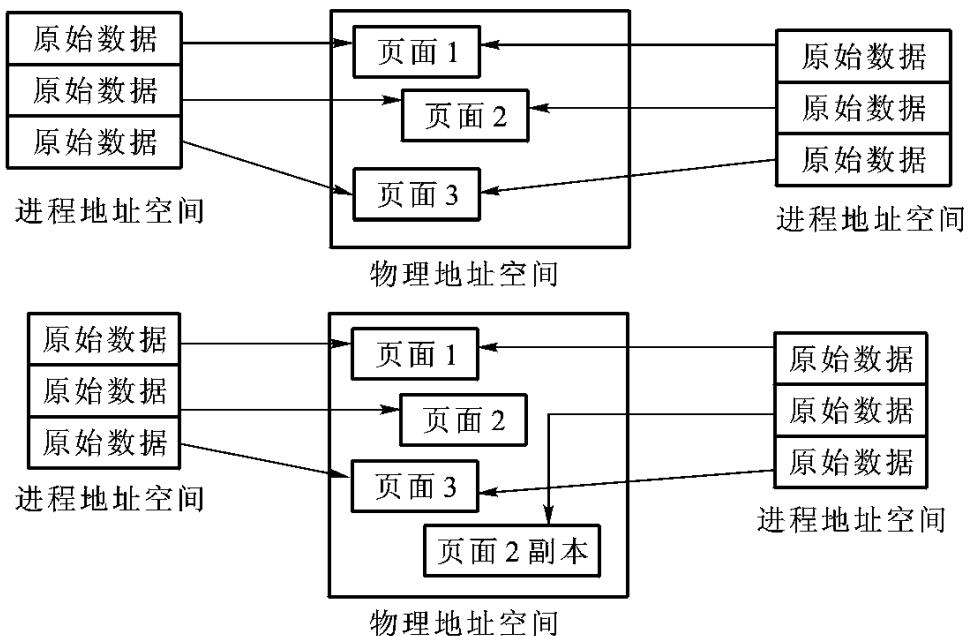


图 4-31 写时复制前(上图),写时复制后(下图)

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,首先把当前需要的一段或几段装入主存,在执行过程中访问到不在主存的段时再把它们动态装入。因此,在段表中必须说明哪些段已在主存,存放在什么位置,段长是多少,哪些段不在主存,它们的副本在辅助存储器的位置。还可设置该段是否被修改过,是否能移动,是否可扩充,能否共享等标志。格式如图 4-32:

| 段号 | 特征 | 存取权限 | 扩充位 | 标志位 | 主存始址 | 段长 | 辅存始址 |
|----|----|------|-----|-----|------|----|------|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

图 4-32 段式虚拟存储管理的

其中:

- 特征位:00(不在内存);01(在内存);11(共享段);
- 存取权限:00(可执行);01(可读);11(可写);

- 扩充位:0(固定长);1(可扩充);
- 标志位:00(未修改);01(已修改);11(不可移动);
- 主存始址和段长:本段在主存的范围;
- 辅存始址:本段在辅存的起始地址。

在作业执行中访问某段时,由硬件的地址转换机构查段表。若该段在主存,则按分段式存储管理中给出的办法进行地址转换得到绝对地址;若该段不在主存中,则硬件发出一个缺段中断。操作系统处理这个中断时,查找主存分配表,找出一个足够大的连续区域容纳该分段。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该分段要求,那么,进行适当移动后,将该分段装入主存。若空闲区总和不能满足要求,则可调出一个或几个分段到辅助存储器上,再将该分段装入主存。

在执行过程中,有些表格或数据段随输入数据多少而变化。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。这只要在该分段后添置新信息就行,添加后的长度应不超过硬件允许的每段最大长度。对于这种变化的数据段,当要往其中添加新数据时,由于欲访问的地址超出原有的段长,硬件产生一个越界中断。操作系统处理这个中断时,先判别一下该段的“扩充位”标志,如可以扩充,则增加段的长度,必要时还要移动或调出一个分段以腾出主存空间。如该段不允许扩充,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 4-33。

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:

第一部分有共享段名、段长、主存起址、状态位(如在不在主存位)、辅存地址、共享进程个数计数器。

第二部分有共享该段的所有进程名、状态、段号、存取控制位(通常为只读)。

由于共享段是供多个进程公用的,对它的主存分配分配要如下进行。当出现第一个要使用某个共享段的进程时,由系统为此共享段分配物理主存区,再将共享段调入该区。同时将共享段主存区始址填入共享段表中对应项的主存始址处,共享进程个数计数器加 1,修改状态位(“在主存位”为 1),填写使用该共享段进程的有关信息(进程名、使用共享段的段号、存取控制等)。而进程段表中共享段的表项指向主存共享段表地址。此后,又有进程使用调入主存的同名共享段时,仅需直接填写共享段表和进程段表,以及把共享进程个数计数器加 1 就行了。当进程不再使用共享段时,应释放该共享段,除了在共享段表中删去进程占用项外,还要把共享进程个数计数器减 1。当共享进程个数计数器为 0 时,说明已没有进程使用此共享段了,系统需回收该共享段的物理内存,并把占用表项也取消。这样做有许多优点:不同进程可以用不同段号使用同一个共享段;由于进程段表中共享段的表项指向主存共享

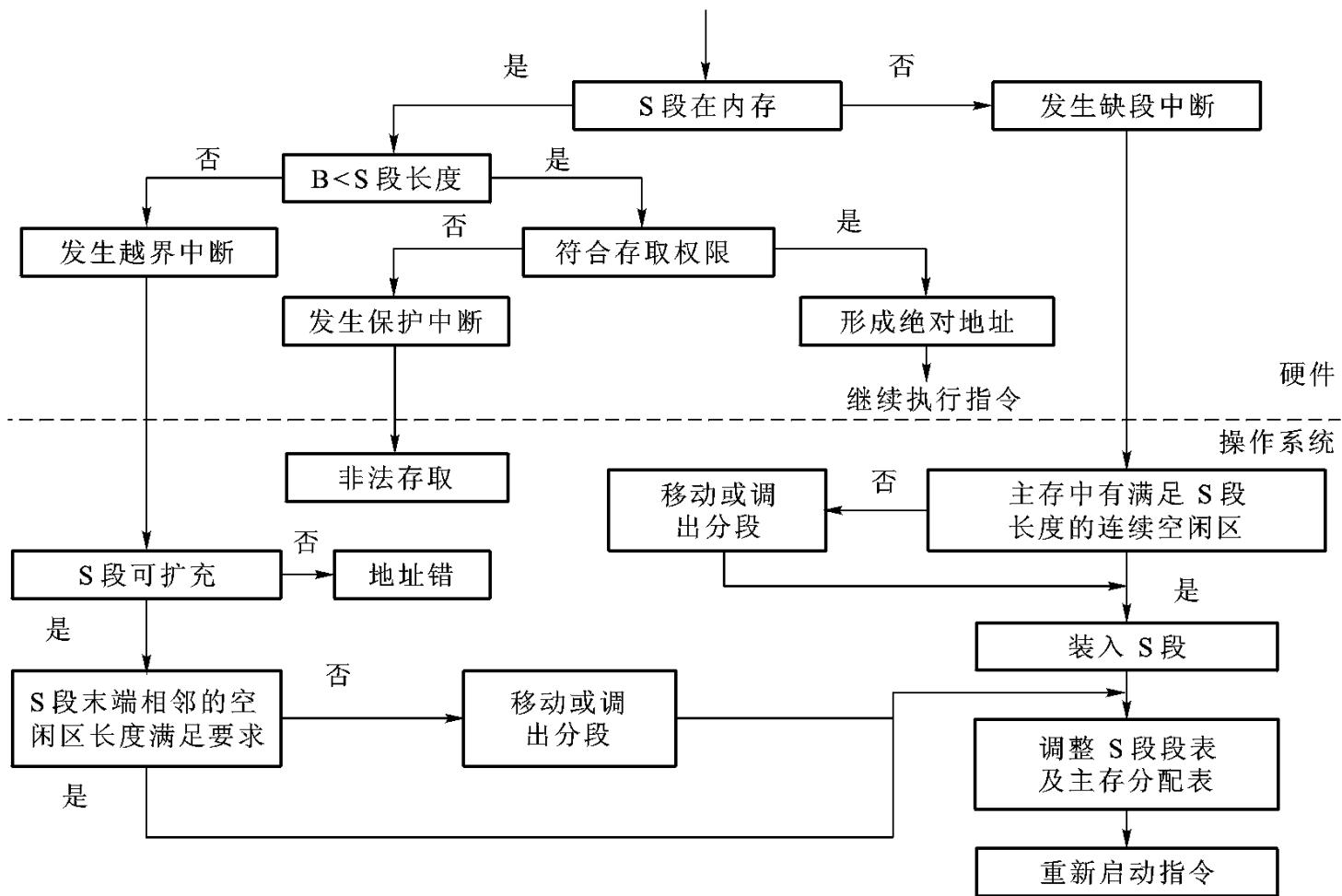


图 4-33 请求分段虚存管理的地址和存储保护

段表地址,每当共享段被移动、调出或再装入时,只要修改共享段表的项目,不必要修改共享段的每个进程的段表。

分段存储器系统中,由于每个分段在逻辑上是独立的,实现存储保护也很方便。一是越界检查。在段表寄存器中存放了段长信息,在进程段表中存放了每个段的段长。在存储访问时,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,还需检查段内地址是否等于或大于段长,将产生地址越界中断,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、读写、只执行等。

4.5.4 请求段页式虚拟存储管理

段式存储是基于用户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,在分页式存储管理的基础上实现分段式存储管理就是段页式存储管理,下面介绍请求段页式虚存管理的基本原理。

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与页框大小相等的页面,于是形成了段页式存储管理的特征。
4. 逻辑地址形式为:

| | | |
|-----------|-------------|-------------|
| 段号(s) | 段内页号(p) | 页内位移(d) |
|-----------|-------------|-------------|

对于用户来说,段式虚拟地址应该由段号 s 和段内位移 d' 组成,操作系统内部再自动把 d' 解释成两部分:段内页号 p 和页内位移 d ,也就是说, $d' = p \times \text{块长} + d$ 。

5. 数据结构

请求段页式虚存管理的数据结构更为复杂,包括作业表、段表和页表三级结构。作业表中登记了进入系统中的所有作业及该作业段表的起始地址;段表中至少包含这个段是否在内存,以及该段页表的起始地址;页表中包含了该页是否在主存(中断位)、对应主存块号。

6. 动态地址转换

请求段页式虚存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 作索引去查快表,如果找到,那么立即获得页 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,完成了地址转换。若查快表失败,就要通过段表和页表来做地址转换了,用段号 s 作索引,找到相应表目,由此得到 s 段的页表的起始地址 s' ,再以 p 作索引得到 s 段 p 页对应的表目,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 生成主存的物理地址,完成地址转换。

上述过程是假设所需信息都在主存的情况下进行的,事实上,会有许多其他情况会产生,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 段在辅存的位置,并将该段页表调入主存;如查页表时,发现 s 段的 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 页在辅存的位置,并将该页调入主存,当主存已无空闲页框时,就会导致淘汰页面。图 4-34 是段页式动态地址转换和存储保护示意。

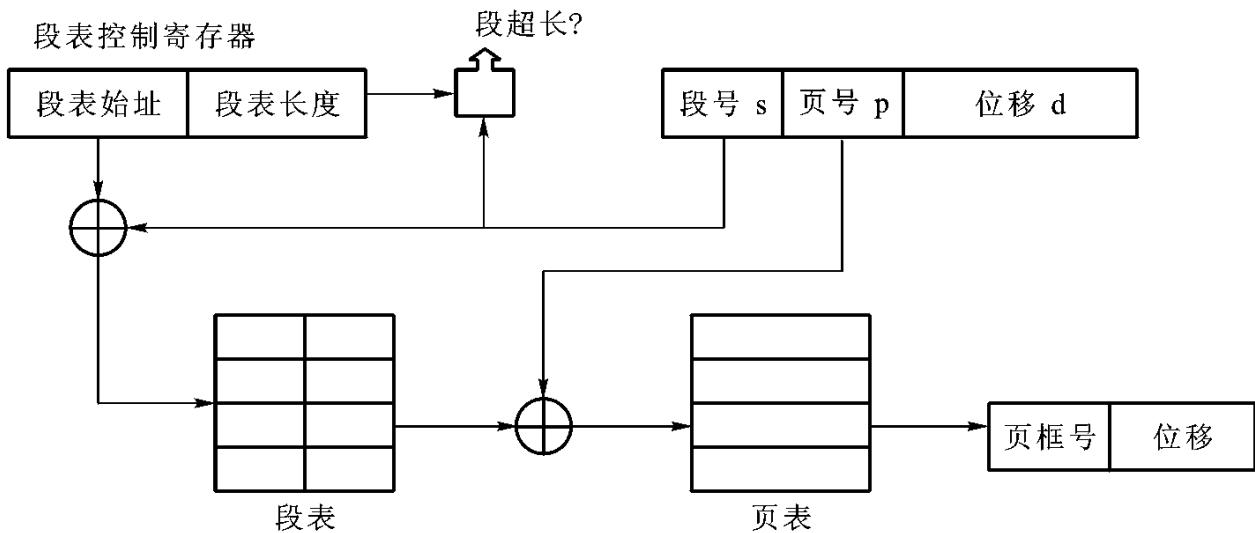


图 4-34 段页式动态地址转换和存储保护

4.6 实例研究: Intel x86/Pentium 存储管理硬件设施

Intel x86/Pentium 系列 CPU 提供三种工作模式: 实地址模式 (real mode), 虚地址模式 (protection mode, 又称保护模式) 和虚拟 8086 模式 (又称虚拟模式)。实地址模式采用段式实存或单一连续分区, 不区分特权级, 不能启用分页机制, 只能寻址 1 MB 地址空间; 保护模式下采用分段机制并可启用分页机制进行地址转换, 共有段式虚存、页式虚存、段页式虚存三种内存管理模式; 虚拟 8086 模式是在保护方式下对实模式的仿真, 允许多个 8086 应用程序同时在 386 以上 CPU 上运行。DOS 在实模式下工作, 而 Windows 和 Linux 等操作系统在保护模式下运行。

4.6.1 Intel x86/Pentium 段机制——段选择符和段描述符

Intel x86/Pentium 上实现存储管理的核心是两张表: 局部描述符表 LDT (local descriptor table) 和全局描述符表 GDT (global descriptor table)。每个进程都有一个自己的 LDT, 它描述每个进程局部于自己的段, 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, 它描述系统段, 包括操作系统自己的基地址、段大小和有关控制信息等。

从逻辑地址(虚地址)到线性地址的转换由段选择符和段描述符组成的分段机制实现, 段选择符软件可处理, 但段描述符软件不可见。为了引用一个段, Pentium 处理器必须把这个段的选择符(selector)装入机器的 6 个段寄存器中的某一个。在运行过程中, 要求段寄存

器 CS 保存代码段的选择符, 段寄存器 DS 保存数据段的选择符。每个选择符是一个 16 位数, 如图 4-35 所示。选择符中有两位指出进程使用处理器的特权级, 1 位指出这个段是局部的还是全局的, 其他 13 位是 LDT 和 GDT 的入口号。由于虚拟地址空间可达 16k 个段, 其中 GDT 映射的一半(8192 个段)是全局虚拟地址空间, 由 LDT 映射的另一半(8192 个段)是局部虚拟地址空间。发生进程切换时, LDT 更新为待执行进程的 LDT, 而 GDT 保持不变。

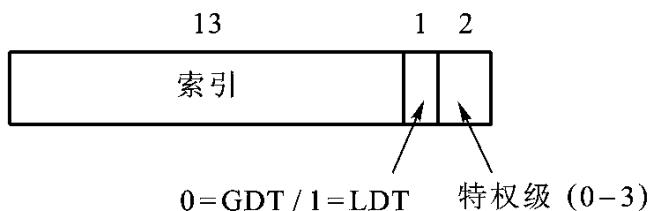


图 4-35 Pentium 的段选符

GDT 和 LDT 拥有相同格式的段描述符, 在选择符被装入段寄存器时, 对应的描述符被从 LDT 和 GDT 中取出装入到微程序寄存器中, 以便快速引用。一个段描述符由 8 个字节组成, 包括段的基址、长度和其他信息, 如图 4-36 所示。

| | | | | | | | | | | |
|----------|---|---|---|----------|---|-----|---|------|---|----------|
| 基址 24-31 | G | D | 0 | 长度 16-19 | P | DPL | S | type | A | 基址 16-23 |
| 基址 0-15 | | | | 长度 0-15 | | | | | | |

图 4-36 Pentium 的代码段描述符(数据段描述符略有区别)

其中:

- 基地址: 共 32 位(分三处合并), 生成内存段的首址, 加上 32 位偏移形成内存地址。对于 286 程序, 基地址的 24~31 位不用, 恒为 0; 所以, 286 只能处理 24 位基址。
- 段长度位: 共 20 位(分两处合并), 限定期描述符寻址的内存段的长度, 注意段长度的计量单位可以是字节或页。
- G 位: 用于描述颗粒大小, 即段长度的计量单位。G = 0 表示长度以字节为单位; G = 1 表示长度以页为单位, 在 Pentium 中页的长度是固定的, 为 4 KB。于是段的长度分别为 220 字节或 220(4 KB = 232 字节)。
- D 位: 当 D = 1 时, 为 32 位段; 当 D = 0 时, 为 16 位段。
- P 位: 表示内存段是否在物理内存中, 若 P = 1, 表示段在内存中, 若 P = 0, 表示段不在内存中。
- Dpl 位: 共 2 位表示段特权级(0~3), 用于保护。0 为内核级; 1 为系统调用级, 2 为共享库级, 3 为用户程序级。Windows 95 只用两级: 0 级和 3 级, 即系统级和用户级。

- S 位:为段内容标志位,当 S = 1 时,表示当前段为应用程序(代码或数据);当 S = 0 时,表示系统段。
- type 类型字段:(3 位),表示内存段类型,如可执行代码段、只读数据段、调用门等等。
- A 位为访问位:表示是否访问过内存段,为淘汰做准备。

4.6.2 Intel x86/Pentium 运行模式选择

Intel 的 286 中的机器状态字 MSW 寄存器中有一位置 1 时,指示从实模式进入保护模式,但该位不能被清 0,要从保护模式切换到实模式可以重新启动机器。

386 以上的机器控制寄存器 CR0 的第 0 位称 PE 位,当 PE 为 1 时,处理器工作在保护模式下;而 PE 为 0 时,处理器工作于实模式下,可以双向切换。CR0 的控制寄存器的第 31 位称 PG 位,当 PG 为 1 时,启用分页机制,控制寄存器 CR3 用来指示页目录表的起址,32 位线性地址通过页目录及页表转换成物理地址;当 PG 为 0 时,则禁止分页机制,32 位线性地址直接寻址物理地址。系统可运行于四种模式:

(1) 系统运行于段页模式时。这时 CR0 的 PE 位 = 1,同时 PG 位 = 1,此时,每个进程可有 $2^{13} = 8192$ 个段,每个段可有 2^{20} 个页,每个页有 4 KB,也就是说段可有 2^{32} 字节大小。

(2) 系统运行于分页模式时。因为 Intel 硬件不存在“不分段”标志,对于线性编址的情况,如果把系统六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),且段描述符基地址设置为“0”,段大小设置为最大 4 GB。这时为单段且也有 CR0 的 PE 位 = 1,同时 PG 位 = 1,系统运行于分页模式,32 位地址空间中单段、分页运行,实际上就是一般的分页。

(3) 系统运行于分段模式时。这时 CR0 的 PE = 1,同时 PG = 0,每个进程可有 $2^{13} = 8192$ 个段,每个段可有 2^{20} 字节大小,这种模式是为了与 286 兼容。

(4) 系统运行于虚拟 8086 模式。只要在保护模式下设置为既不分段(实际仅有一段),又不分页就可以了,这时也有 CR0 的 PE = 1,PG = 0。

4.6.3 Intel x86/Pentium 地址转换

在实模式下,段寄存器中存放段基址,因此,Intel x86/Pentium 系列实模式下 CPU 的地址转换过程比较简单,图 4-37 给出了地址转换过程。

下面来看 Intel x86/Pentium 保护模式下采用分段但不分页时的逻辑地址到物理地址的转换过程,如图 4-38 所示,每一个逻辑地址包括一个 16 位段选择符和一个 32 位(或 16 位)偏移量,不分段时,用户的虚拟内存是 $2^{32} = 4$ GB;而分段时,虚拟地址空间为 $2^{(32+14)} = 64$ TB;物理地址空间使用 32 个地址位,最大有 4 GB。下面来讨论地址转换过程如下:

- 首先根据段选择符第二位 T 选择是查找 LDT 或 GDT;

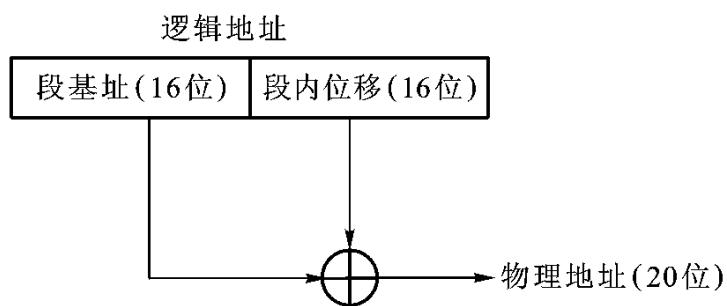


图 4-37 Pentium 系列 CPU 实模式下的地址转换

- 以段选择符索引域的值为索引, 把它拷贝进一个内部寄存器中并且它的低三位被清零, 然后 LDT 或 GDT 表的起始地址被加到它上面, 找到所要访问的段描述符。
- 检查段描述符中的 P 控制位, 如果段不在主存(P 为 0), 就会发生一次陷入(中断)处理; 如果段已经被换出, 就会发生一次陷入(缺段中断)处理; 如果段在主存, 则随后检查偏移量是否超出了段的结尾, 如果是也会发生一次陷入(越界中断)处理。
- 假设段在主存中并且偏移量也在范围内, 就把描述符中 32 位的基地址和偏移量相加形成 32 位线性地址(linear address), 如图 4-38 所示。为了和只有 24 位基地址和 16 位段内位移的保护模式(80286)以及使用 16 位段寄存器来描述 20 位基地址的实模式(8086/8088)兼容, 基址被分成 3 片分布在描述符的各个位置。实际上, 基地址允许各个段的起始地址在 32 位线性地址空间的任何位置。至此段转换已经完成。

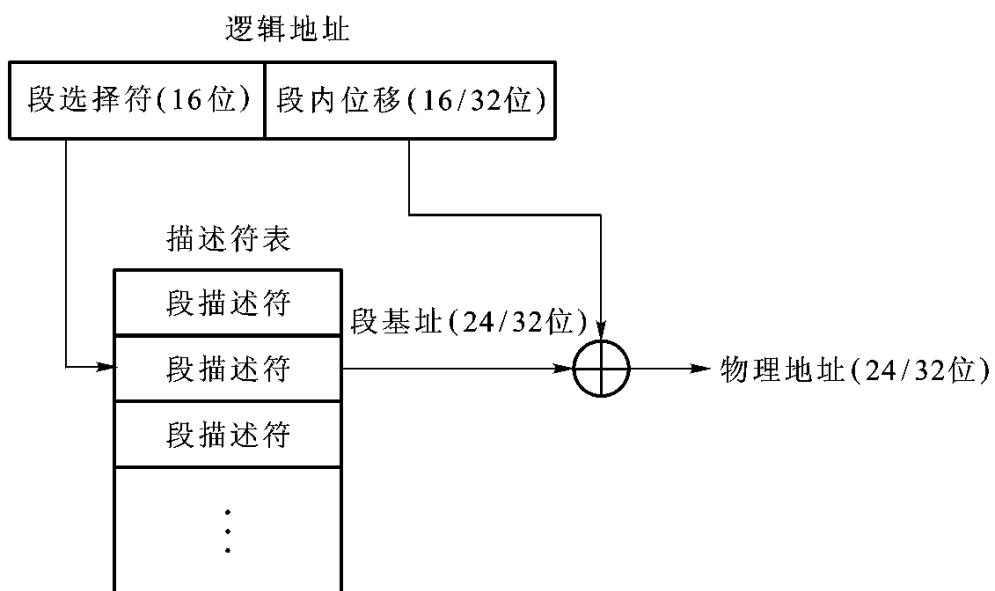


图 4-38 Pentium 系列 CPU(保护模式下)段式不分页的地址转换过程

是否要分页依据 PG 控制位, 若为 0, 说明目前运行于分段模式。那么段转换得到的线性地址就是物理地址并被送往存储器用于读写; 因此, 在分页被禁止时, 就得到了一个纯的

分段解决方案,各个段的基址在它的描述符中。注意,段允许互相覆盖,这是因为用硬件来检查所有的段都互不重叠代价太大,完全可以通过操作系统的软件机制加以解决。

若 PG 控制位为 1,这时以分页模式运行。那么,段转换得到的线性地址将被解释成分页模式的虚地址,需要通过页表找到页框后才映射成物理地址。这里真正复杂的是在 32 位虚地址和 4 K 页面的情况下,一个段可能包括多达一百万个页。因此,Pentium 使用了一种两级页表以在段较小的情况下减少页表的尺寸。

4.6.4 Intel x86/Pentium 页式或段页式地址转换

Intel x86/Pentium 在保护模式下,如果把系统六个段寄存器设置为同一个段选择符,段描述符基地址设置为“0”段大小设置为最大 4 GB,且启动分页机制,这时系统运行于单段、分页模式,可在 32 位地址空间模式运行。页目录(page directory)是一个大小为 4 KB 的内存块,以 4 个字节为一个目录项,因而,每个运行进程都有 1 024 个页目录项入口点,它的地址由全局寄存器 CR3 指出,32 位线性地址的高 10 位(0 ~ 1023)分别对应于页目录项入口点。页目录中的每一个页目录项的值描述了一块大小为 4 KB 的内存块,这个内存块存放部分页表称作页表页(page table)。它和页目录类似,也有 1 024 个页表项(也以 4 个字节为一个页表项)入口点,32 位线性地址的中间 10 位(0 ~ 1 023)分别对应于页表项入口点。页表页的每个表项的指示一块大小为 4 KB 的内存块,这个内存块称作页框(page frame,又称块),其长度等同于页面长,32 位线性地址的低 12 位(0 ~ 4 095)分别对应于页框中的每个字节,其中存放进程的程序或数据。这个方案如图 4-39 所示,线性地址被分成 3 个域:dir、page 和 offset。dir 域被作为索引在页目录中找到指向正确页表页的指针;随后,page 域被作为索引在页表页中找到页框的物理地址;最后,offset 被加到页框的地址上得到需要的字节或字的物理地址。

如图 4-40 所示,页目录项和页表项的结构是类似的,均为 32 位。其中 20 位是页表位置/页架号,其余的包括由硬件设置的供操作系统使用的引用位、修改位、保护位、和其他一些有用的位。

其中:

- G 位,全局页,如该表项是页目录中表项时将忽略。
- D 位,修改位,当该页程序修改后,该位被置位;如该表项是页目录中表项时,处理器不修改此位,并置为 0 保留备用。
- A 位,(读/写)引用位,凡对程序读写均置为已访问(accessed),淘汰时先找 A = 0 的页面。
- PCD 位,禁止 cache。
- PWT 位,同时写 cache 和内存。
- P 位,存在位,页面在内存时,P = 1。

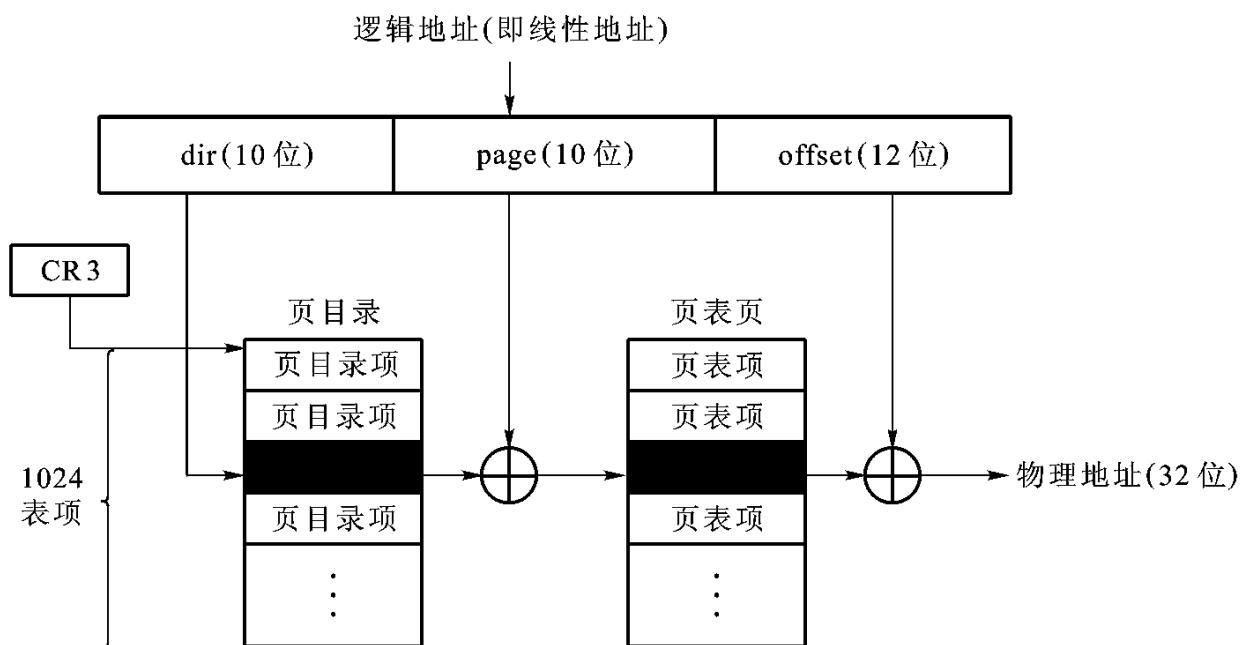


图 4-39 Pentium 系列 CPU(保护模式下)页式不分段地址转换过程

页目录项

| 31 | 12 | 0 |
|------|----------------------------|---|
| 页表起址 | G PS D A PCD PWT U/S R/W P | |

页表项

| 31 | 12 | 0 |
|------|---------------------------|---|
| 页框起址 | G 0 D A PCD PWT U/S R/W P | |

图 4-40 页目录项和页表项的结构

- U/S 位, 用户/管理员 (User/Supervisor) 位, 若 U/S = 0, 该页为一个管理员页面 (OS 页面), 用户不能访问, 否则可在任何处理器特权级下访问。
- R/W 位, 读/写 (Read/Write) 位, R/W = 1 时允许页修改; R/W = 0 时, 不允许页修改。通常程序区的 R/W = 0。

每个页表页有描述 1 024 个 4 K 页框的表项, 因此, 一个页表页可以处理 4 M 的内存。一个小于 4 M 的段有且仅有一个页表页。因此, 通过这种方法, 短的段地址转换只需要访问两个页面。

Intel x86/Pentium 在保护模式下分段且启动分页机制, 则系统在段页式下运行, 图 4-41 便是地址转换过程。为了加快地址转换的速度, Intel x86/Pentium 也设置了相联存储器, 又

称为翻译后援存储器 TLB, 它把最近使用过的 dir/page 暂存起来, 这样就可以不通过页表而快速地把线性地址转化成物理地址。

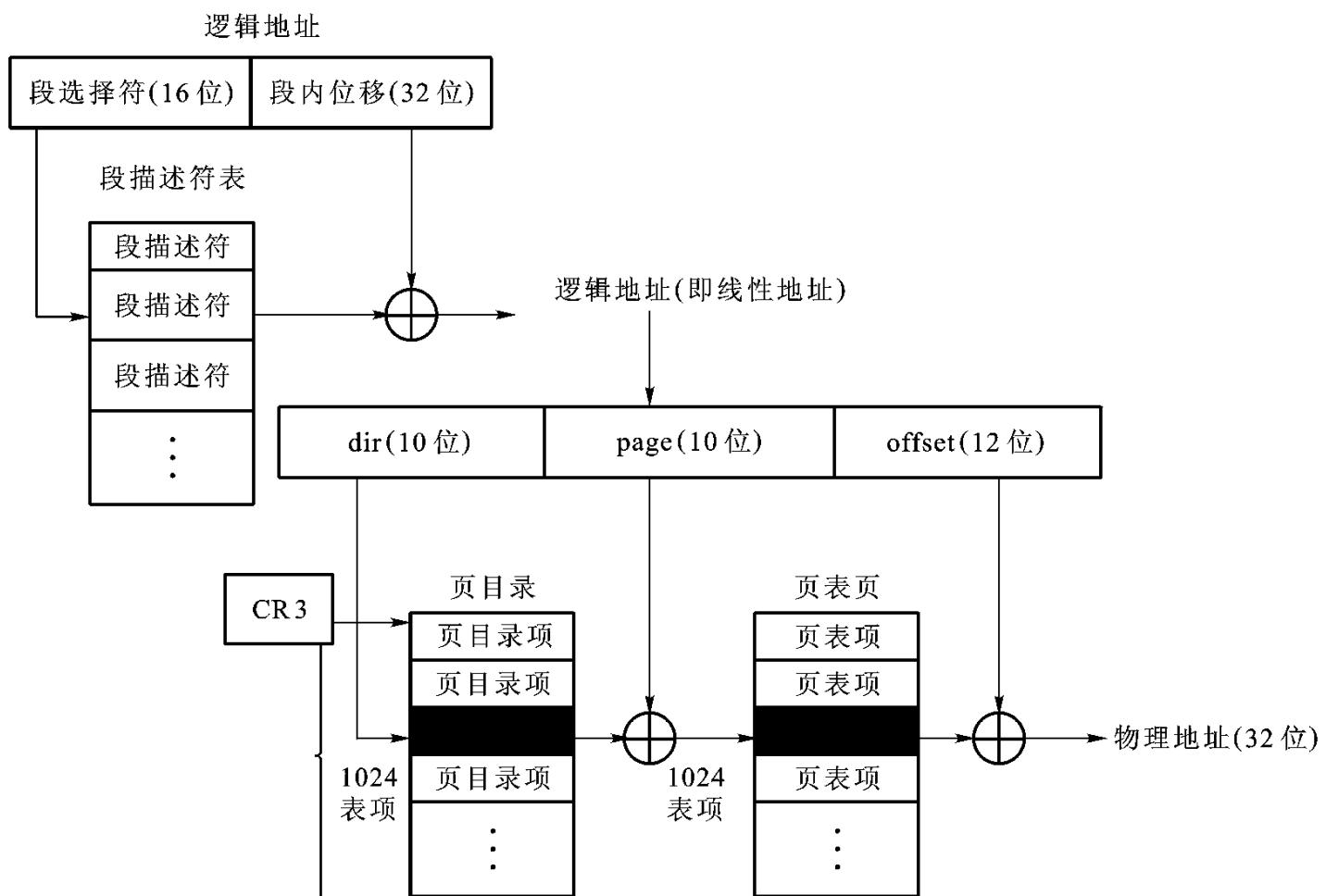


图 4-41 Pentium 系列 CPU(保护模式下)段页式地址转换过程

必须承认, Intel x86/Pentium 的设计非常巧妙, 它实现了互相冲突的目标:段页式存储管理、纯页式存储管理、纯段式存储管理、能够同 80286 以前的保护模式和实模式兼容的存储管理。

4.7 实例研究:Windows 2000/XP 虚拟存储管理

4.7.1 进程地址空间布局

Windows 2000/XP 的内存管理器是执行体中的虚存管理程序 VMM (Virtual Memory Manager)的一个组件,位于 Ntoskrnl.exe 文件中,是 Windows 的基本存储管理系统。它实现内

存的一种管理模式——虚拟内存,为每个进程提供一个受保护的、大而专用的地址空间。系统支持的面向不同应用环境子系统的存储管理也都基于 VMM 虚存管理程序。

Windows 2000/XP 采用“请页式”虚存管理技术,运行于 386 以上的机器上,提供 32 位虚地址,每个进程都有多达 4 GB(2³²B) 的虚地址空间。图 4-42 是进程虚拟地址空间的布局,进程 4 GB 的地址空间被分成两部分:高地址的 2 GB 保留给操作系统使用,低地址的 2 GB 是用户存储区,可被用户态和核心态线程访问。Windows 提供一个引导选项,允许用户拥有 3 GB 虚地址空间,而仅留给系统 1 GB,以改善大型应用程序运行的性能。

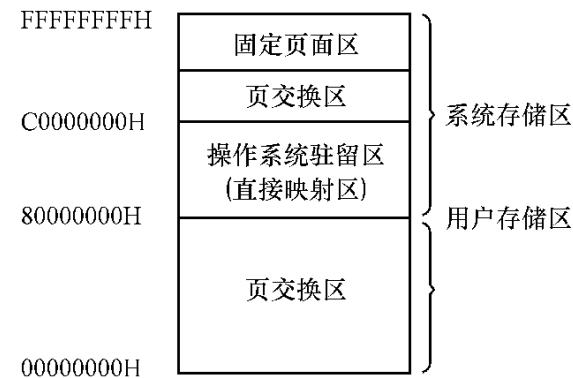


图 4-42 虚拟地址空间布局

系统存储区又分为三部分:上部分固定页面区,页面不可交换以存放系统的关键代码;中部为页交换区,用以存放非常驻的系统代码和数据;下部分为操作系统驻留区,存放内核、执行体、引导驱动程序和硬件抽象层代码,永不失效。为了加快运行速度,这一区域的寻址由硬件直接映射。

4.7.2 用户空间内存分配

下面介绍 Windows 2000/XP 是如何管理应用程序内存空间的,与管理应用程序内存相关的有两个数据结构:虚址描述符和区域对象;三种应用程序内存管理方法:

- 虚页内存分配:最适合于管理大型对象数据或动态结构数组;
- 内存映射文件:最适合于管理大型数据流文件及多个进程之间的数据共享;
- 内存堆分配:最适合于管理大量小对象数据。

1. 虚址描述符

Windows 2000/XP 中一个进程的虚地址空间可以大到 4 GB,这意味着进程的虚地址不是连续的,系统维护了一个数据结构来描述哪些虚拟地址已经在进程中被保留,而哪些没有,这个数据结构叫做“虚地址描述符”VAD(Virtual Address Descriptor)。对每个进程,内存管理器都维护一组 VAD,用来描述进程虚地址空间的状态。为了加快对虚地址的查找速度,VAD 被构造成记录虚址分布范围的一棵平衡二叉树(Self-balancing Binary Tree),如图 4-43 所示。

在 Windows 2000/XP 中,当进程保留地址空间或映射一个内存区域时,内存管理器创建一个 VAD 来保存分配请求所提供的信息。例如,保留的地址范围、该范围是共享的还是私有的、子进程能否继承该地址范围的内容,以及此地址范围内应用于页面的保护限制等。

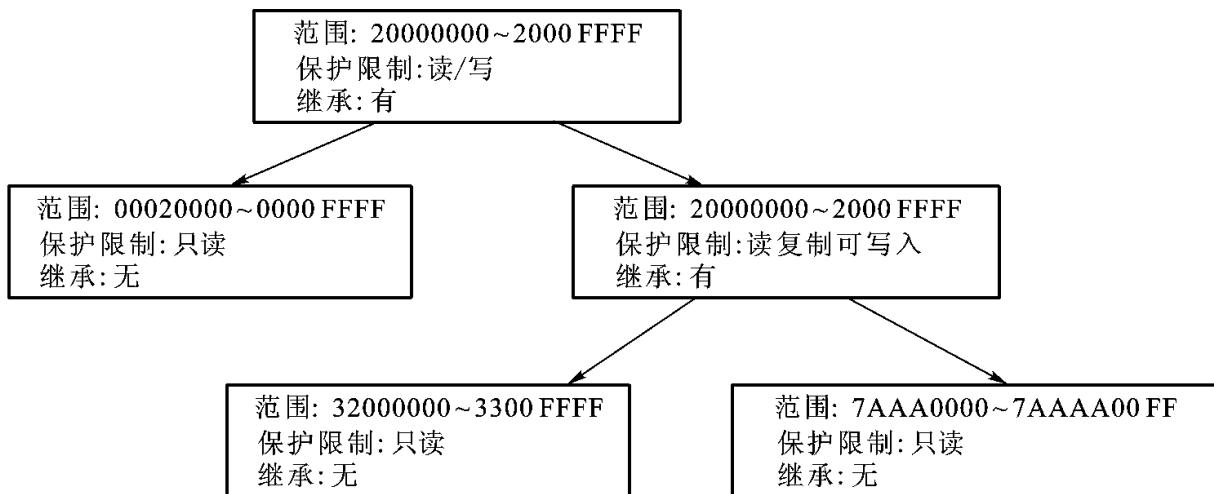


图 4-43 虚地址描述二叉树

当线程首次访问一个地址, 内存管理器必须为包含此地址的页面创建一个页表项。为此, 它找到一个包含被访问的地址的 VAD, 并利用所得信息填充页表项。如果这个地址落在 VAD 覆盖的地址范围以外, 或所在的地址范围仅被保留而未提交, 内存管理器就会知道这个线程在试图使用内存之前并没有分配的内存。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”, 表示可以被两个或多个进程所共享的内存块。其主要作用有以下几点:

- 系统利用区域对象将可执行映像和动态链接库装入内存;
- 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;
- 使用区域对象将一个文件映射到进程地址空间, 然后, 可以像访问内存中一个大数组一样访问这个文件, 而不是对文件进行读写。

图 4-44 显示了区域对象的结构。

一个区域对象代表一个可由两个或多个进程共享的内存块。一个进程中的一个线程可以创建一个区域对象, 并为它起一个名字, 以便其他进程中的线程能打开这个区域的句柄。区域对象句柄被打开后, 一个线程就能把这个区域对象映射到自己或另一个进程的虚地址空间中。

| 对象类型 | 区域 |
|------|-------------------------------------------|
| 对象属性 | 最大尺寸 页保护限制 盘交换区/映射文件 基本/非基本区域 |
| 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 4-44 区域对象

3. 区域对象和视窗

进程的虚拟空间(最大 2^{32} B)可能比区域对象(最大 2^{64} B)小很多。为了让进程能访问更多的物理内存空间, Windows 2000/XP 提供了一个视窗 AWE(又称 Address Windowing Extension)机制, 来为区域对象空间中的一部分空间(进程当前所需要的)保留虚地址, 而所映射的这一部分叫该区域对象的一个视窗。映射一个区域对象的视窗就是使该区域的一部分能在虚地址空间可见。反之, 删除映射一个区域对象的视窗就是把它从进程的虚拟地址空间中删去。这样一来, 进程就可以通过对一个区域对象多次开设和删除视窗, 来达到在较小的虚地址空间(最大 2^{32} B)中处理较大物理内存空间(2^{64} B)的目的。图 4-45 是映射一个区域对象的视窗示意。例如, 一个有 8 GB 物理内存的高端 Windows 2000/XP 数据库服务器上, 应用程序可以用 AWE 来分配和使用将近 8 GB 内存作为数据库缓存。AWE 对于多于 2 GB 或 3 GB 物理内存的系统最有用。因为, 对于 32 位的进程来说, 这是惟一的可直接使用多于 2 GB 或 3 GB 内存的方式。

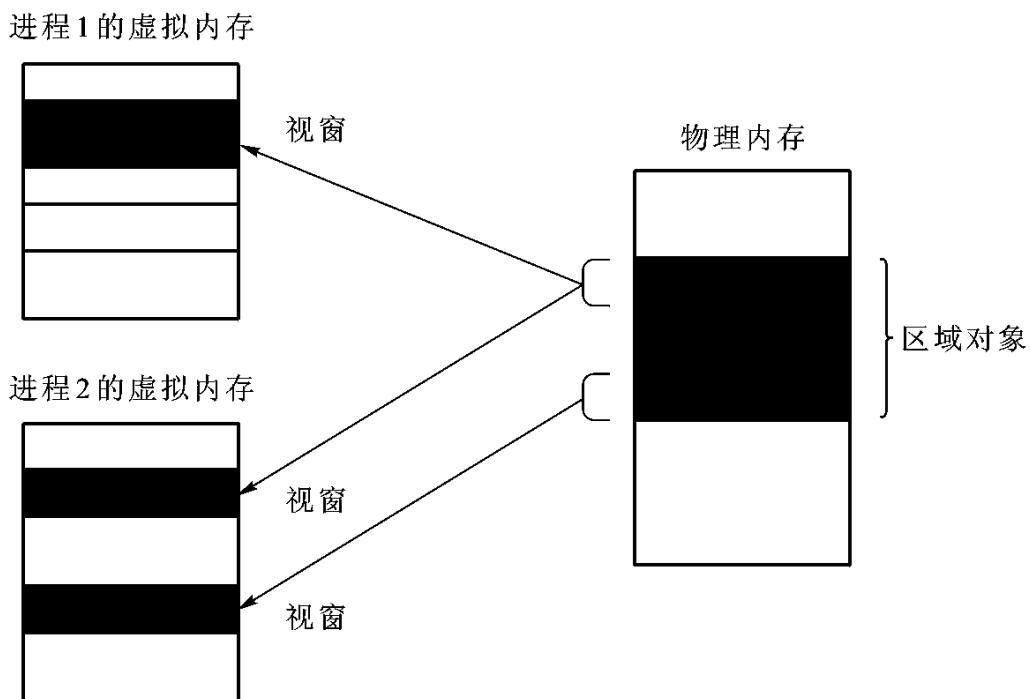


图 4-45 映射区域对象的视窗。

可以按以下三步使用 AWE 函数来分配内存:

步 1 分配将要使用的物理内存。

步 2 创建一个虚地址空间作为视窗来映射区域对象。

步 3 把区域对象映射到视窗。

一个进程可以对一个区对象开一个或多个视窗, 不同进程可以对同一区域对象开设视窗。视窗机制在以下多种场合能发挥作用:

- 当基于内存映射文件的区域对象远远大于虚地址空间时；
- 当基于盘交换区(交换文件)的区域对象远大于交换空间时；
- 用于共享时。

Windows 2000/XP 提供了出色的内存共享机制, 共享的进程通过创建“区域对象”作为共享的内存区域。当两个进程对同一区域对象建立视窗时, 就发生了对区域对象的共享。不同的进程的视窗在区域对象空间中的位置可以相同也可以不同。

4. 应用程序内存管理方法

1) 虚页内存分配

在 Windows 2000/XP 中使用虚拟内存, 要分三个阶段: 保留内存(reserved memory)、提交内存(committed memory)和释放内存(release memory)。

(1) 保留内存

用户使用 Win32 virtual Alloc 可以申请保留一段连续的虚拟地址空间, 使本进程的其他线程或其他进程不能使用这段虚地址, 这样做的目的让用户表明将要用多大的内存, 以便系统节省内存空间和磁盘空间, 而直到需要时才提交。试图访问已保留的内存会造成访问冲突, 因为, 这时内存页面还没有映射到一个可以满足这次访问的物理空间中。保留内存操作不占用和消耗内存空间和盘交换区。在申请保留内存中, 用户进程可以指定起始虚拟地址(lpAddress 参数), 当没有特殊要求。此参数设为 NULL 时, 系统将随便保留一段地址空间, 分配的单位通常为 64 KB; 此时, 还需提供保留地址空间大小(cbSize 参数), 保留或提交(fdwAllocationType 参数); 保护限制信息(fdwProtect 参数), 指定页面不可访问、只读或读/写等。

(2) 提交内存

提交内存指提交物理页面, 在已保留的区域中, 提前页面必须指明将物理存储器提交到何处提交多少。当一段虚地址分配盘交换区配额和在盘交换区中分配空间。然后, 把分配的存储空间映射到保留内存的虚地址空间。提交内存是为了减少盘交换区的空间占用而设立的, 这一过程通常推迟到第一次实际访问该虚址时才提交, 当然也可以在保留内存的同时提交(这时使用 VirtualAllocEx)。提交内存仍然使用 Win32 函数 Virtual Alloc, 与保留内存的区别是参数设置不同, 有关细节可参见 Win32 API 的内存函数。

综合上述的两个阶段, 用户访问虚地址空间时: 首先, 保留内存; 分配虚地址空间, 若访问未保留的虚址将引起地址越界错误。其次, 提交内存; 分配盘交换区空间, 访问未提交的虚址空间将引起地址越界错误。最后, 分配内存; 访问不在内存且已提交的页面将导致实际分配页面。

(3) 释放内存

当进程不再需要被提交的内存或保留的地址空间时, 使用释放函数 Win32 Virtual Free

来回收盘交换区空间或从进程虚址空间中释放虚址,需要的参数有:释放起址(lpAddress);释放长度(cbSize);是否仅释放物理存储(fdwFreeType)。这个函数对减少和控制盘交换区和进程虚址空间的占用是很有用的。

2) 内存映射文件

Windows 2000/XP 内存映射文件用途很多,主要可以用于三种场合:

(1) Windows 执行体使用内存映射把可执行文件 .exe 和动态连接库 .dll 文件装入内存,节省应用程序启动所需时间。

(2) 进程使用内存映射文件来存取磁盘文件信息,这可以减少文件 I/O,且不必对文件进行缓存。

(3) 多个进程可使用内存映射文件来共享内存中的数据和代码。

此外,Windows 2000/XP 高速缓存管理程序使用内存映射文件来读写高速缓存的页。

Win 32 子系统向其客户进程提供了文件映射对象(File - mapping Object)服务。实际上,文件映射对象服务就是 Win32 子系统将区域对象服务通过 Win API 向客户进程所提供的。Win32 映射文件对象等效于一个区域对象。因此,区域对象是实现内存映射文件功能的关键所在。

内存映射文件的使用步骤如下:

步 1 打开文件 进程使用 CreateFile 函数来建立和打开一个文件,该系统调用指定要建立或打开的文件名,访问文件的方式,与其他进程的共享限制等。执行该函数将返回一个文件句柄。

步 2 建立文件映射 进程使用 CreateFileMapping 创建一个文件映射对象,实质是为文件保留虚址而创建一个区域对象。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。执行该函数将返回一个句柄给文件映射对象。

步 3 读写文件视窗 进程通过函数 MapViewOfFile 返回的指针来对文件视窗进行读写操作。由于区域对象可以指向地址空间大得多的文件,要访问一个非常大的区域对象只能通过此函数映射区域对象的一部分——视窗,并指定映射范围。该系统调用的参数包括:文件映射对象句柄,访问方式、视窗起址在映射文件中的位移、视窗映射的虚址空间字节数。

步 4 打开文件映射对象 其他进程使用 OpenFileMapping 打开一个已存在的文件映射对象,以便共享信息或达到通信的目的,参数包括:访问权限(dwDesiredAccess);子进程能否继承该句柄(bInheritHandle)和对象名(lpName)。

步 5 解除映射 访问结束,进程使用 UnmapViewOfFile 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(lpBaseAddress)。

3) 内存堆分配

堆(heap)是保留地址空间中一个或多个页组成的区域,并由堆管理器按更小块划分和

分配内存的技术。堆管理器的函数位于 Ntdll 和 Ntoskrnl.exe 中, 分配和回收内存空间时, 不必像虚页分配一样按页对齐。Win32API 可以调用 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。

进程启动时带有一个缺省的进程堆, 通常有 1 KB 大小。如果需要它会自动扩大。进程也可以使用 HeapCreate 创建另外的私有堆, 使用完就可通过 Heap Destroy 释放申请的堆空间, 也只有另外创建的私有堆才可以在一个进程的生命周期中被释放。

为了从缺省堆中分配内存, 线程调用 GetprocessHeap 函数得到一个指向堆的句柄。然后, 线程可以调用 HeapAlloc 和 HeapFree 函数从堆中分配和回收内存块。

4.7.3 内存管理的实现

1. 进程页表与地址映射

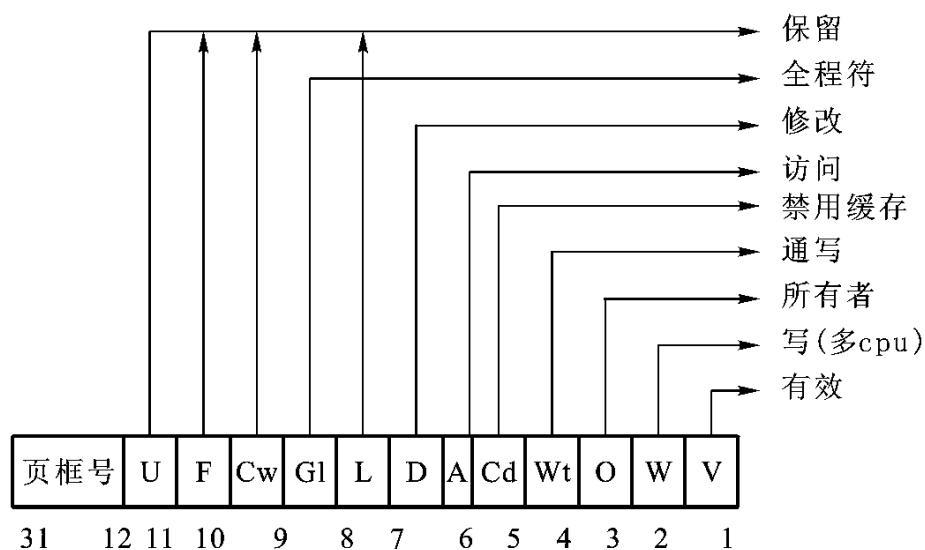
Windows 2000/XP 在 x86 硬件平台上采用二级页表结构来实现进程的逻辑地址到物理地址的转换。一个 32 位的逻辑地址被解释成三个分量, 页目录索引(10 位)、页表页索引(10 位)和位置索引(12 位), 页面大小为 4 KB。

每个进程都拥有一个单独的页目录, 这是内核管理器创建的特殊的页, 用来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, 但需要时也可以在进程之间共享。页表是根据需要创建的, 所以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常常不连续。

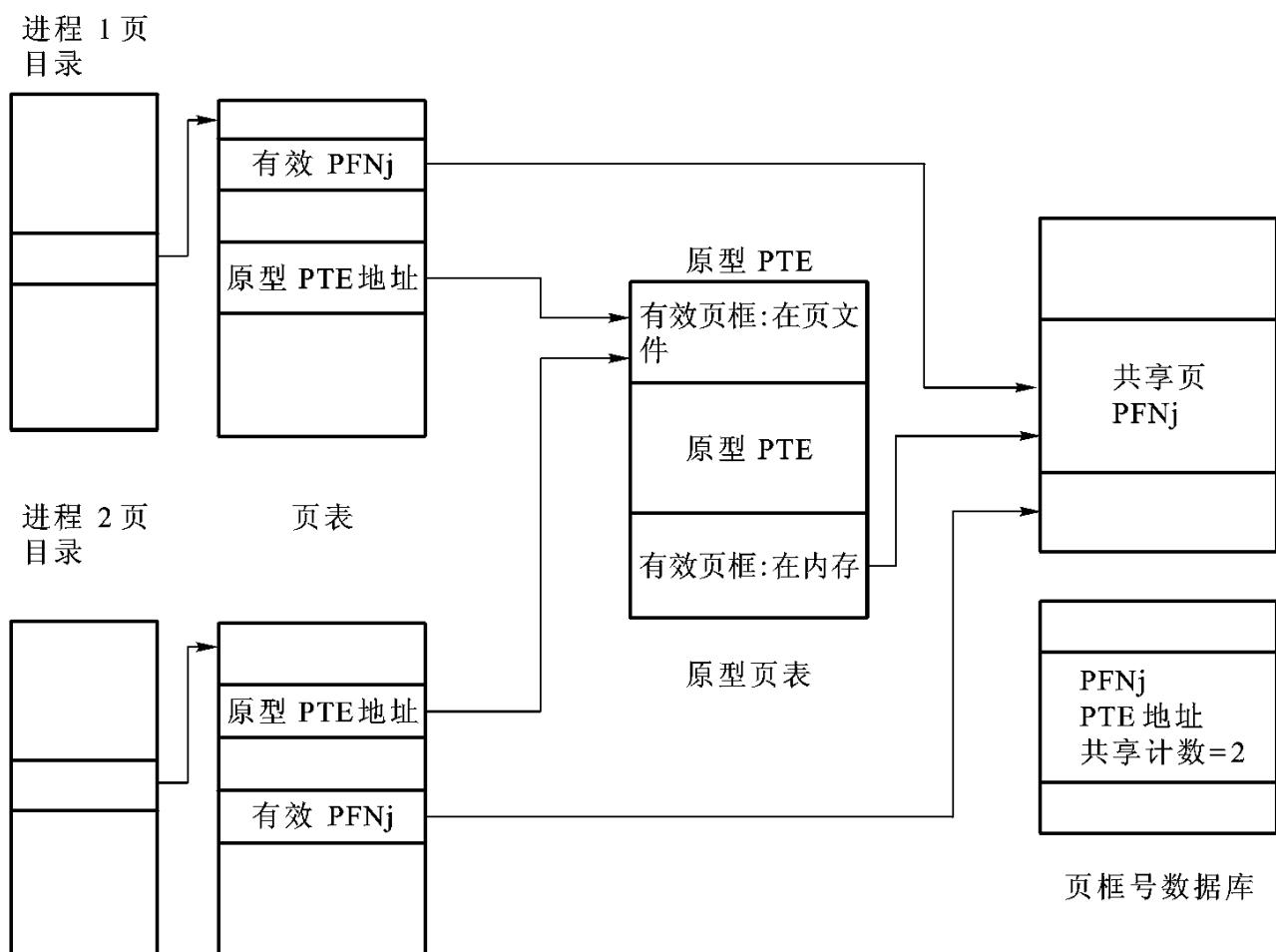
CPU 通过专用寄存器 CR3 来找到页目录存放地址。每次进程切换时, 由操作系统负责把运行进程的页目录的物理地址放入这个寄存器中。页目录是由页目录项 PDE (page directory entry) 组成的, 每个 PDE 有 4 个字节长, 描述了进程所有页表的状态和位置。页表是根据需要来创建的, 故进程的页目录仅指向一部分页表。在 x86 CPU 上, 每个进程需要一张页目录表(有 1024 个 PDE)指出 1024 张页表页, 每张页表页描述 1024 个页面, 合计描述 4 GB 的虚地址空间。其中, 用户进程最多 512 张页表页, 系统进程占用另外 512 页表页并被所有用户进程共享。页表项 PTE (page table entry) 的格式如图 4-46 所示。对于逻辑地址到物理地址的转换过程, 已经在上一节介绍过, 这里不再重复。

2. 原型页表项和区域对象

当两个进程共享一个物理页面——页框时, 内存管理器在进程页表中插入一个称作“原型页表项”prototype PTE (prototype page table Entry) 的数据结构来间接映射共享的页面, 它是进程间共享内存的内部实现机制。由于区域对象所指定的区是被多个进程共享的内存, 当一个区域对象第一次被创建时, 对应于它的一个原型 PTE 同时被创建, 于是多个进程可通过原型 PTE 来共享页框。图 4-47 是通过原型 PTE 来共享内存页框的示意, 当共享页框为



有效时,进程页表项和原型 PTE 均指向包含代码或数据的物理页;而当共享页面无效时,进程页表中的页表项由一个特殊的指针来填充它,该指针指向描述该页框的原型 PTE。此后,



当页面被访问时,内存管理器就可以利用这个页表项中的指针来定位原型 PTE,而原型 PTE 确切地描述了被访问的页框的情况。

原型 PTE 是一个 32 位的数据结构,它与一般的页表项类似,包含页框号、页保护和页状态等信息,但它并不用作地址转换,也不会出现在页表项中。使用它的主要优点是:系统在管理共享页面时,无需更新每个共享此页的进程的页表。例如,一个共享的代码或数据页可能在某个时候被调出内存,当它被再次被调入内存时,只需要更新原型 PTE,使之指向此页面的新的物理位置,而共享此页的进程的页表保持不变。此后,当进程访问该页面时,实际的页表项才得到更新。原型 PTE 位于可交换的系统空间内,它与页表一样可在必要时换出内存。

为了追踪访问有效共享页面的进程数,在“页框号数据库项”内增加了一个计数器。这样,内存管理器就能确定一个共享页面何时已经不再被进程引用,这个页可以被标记为无效,并送到转换链表或写回辅存中。

共享页面可以是原型 PTE 所描述的 6 种状态之一:

- 活动/有效 (active/valid):由于另一个进程访问过此页,页面存在物理内存中;
- 过渡 (transition):所需页在内存的后备链表或修改链表中;
- 修改尚未写入 (modified – no – write):所需页在内存的修改尚未写入链表中;
- 请求零页 (demand zero page):所需页内容应为零页;
- 页文件 (page tile):所需文件驻留在页文件中;
- 映射文件 (mapped file):所需页驻留在映射文件中。

3. 页框号数据库

在 Windows 2000/XP 中,所有内存物理页框组成了页框数据库 (Page Frame Database),每个页框占一项,每项称为一个 PFN 结构 (Page Frame Number)。页框数据库项是定长的,根据页面的情况有不同的状态,个别域针对不同状态又有不同含义,图 4-48 为页框号数据库项的状态。

现对其中几项作简单说明:

- 页表项地址:指向此页框的页表项的虚地址。
- 访问计数:对此页面的访问计数,当页面被首次加入工作集或当该页在内存中被锁定时,访问计数就会增加;当共享计数为 0 或从内存解锁时,访问计数将减少。
- 类型:指该 PNF 可能的状态。
- 标志:区别该页是否被修改过,是否为原型 PTE、是否不奇偶校验错、正在作何种操作。

PFN 在任一时刻可能处于下面 8 种状态之一:



图 4-48 页框号数据库项状态

1. 有效(Valid) 该页框在进程工作集中或是非分页的内核页面,有一个有效页表项指向此页。
 2. 过渡(transition) 该页框不在工作集中且不属于页面调度链表的页面所处的暂时状态,通常该页进行 I/O 时,会处于过渡态。
 3. 后备(Stand by) 该页框被一个进程使用,但已从该进程工作集中删除。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) 该页和后备状态相同,但页面被进程修改过,且其内容还未写到磁盘上。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。所以,该页框被重新使用前应先写回磁盘。
 5. 修改不写入(modified no write) 该页和修改状态相同,但被标记不写回磁盘。在文件系统驱动程序发出请求时,高缓存管理器标记页面为修改不写入状态。
 6. 空闲(free) 该页框是空闲的,但有不确定的数据(未被初始化(清 0))。
 7. 零初始化(zeroed) 该页框是空闲的,并已被初始化为全零,随时可用。
 8. 坏(bad) 该页已发生奇偶错或其他物理故障,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,以便内存管理器可以很快定位特定状态的页框。页框状态的转换如图 4-49 所示。

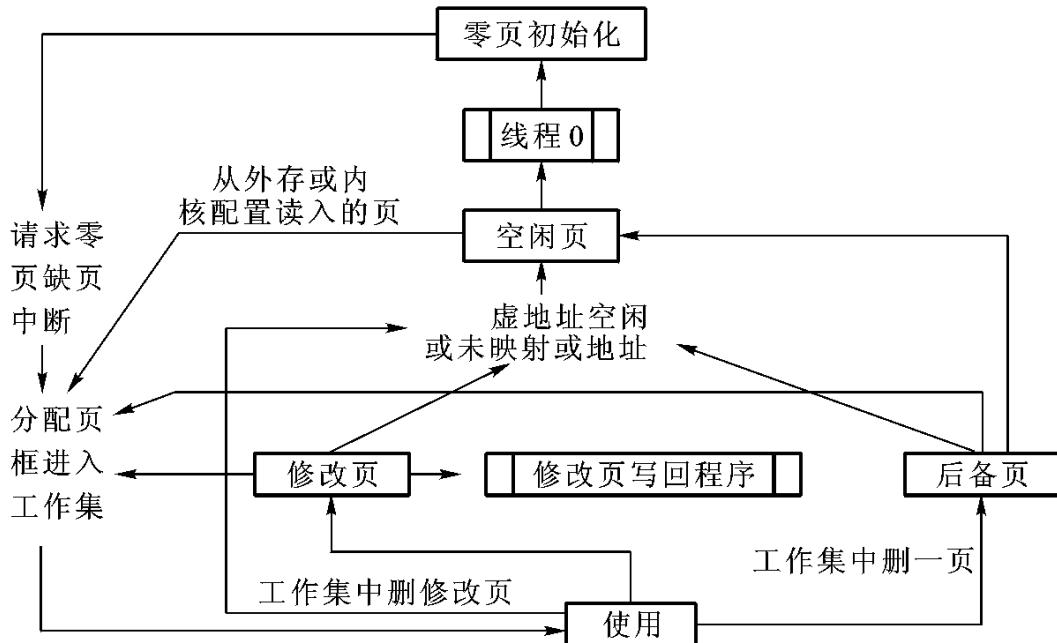


图 4-49 页框的状态及其转换

4. 请求调页

内存管理器在分配页框时,是按照以下次序从非空链表中取得页面进行分配:零页链表→空闲链表→后备链表→修改链表。当发生一次缺页中断时,首先查看所需页是否在后备链表或修改链表中。若在,则将此页从表中移出,收回到进程的工作集中去,不必再分配新的页框;若不在,如果需要一个零初始化页,则内存管理程序总试图在零页链表中取出第一页,若零页链表为空,则从空闲链表中取一页并对它进行零初始化。若需要的不是零初始化页,就从空闲表中取出第一页。如果空闲链表为空,就从零初始化页中取一页。如果以上任一情况中零页链表和空闲表均为空,那么使用后备链表。

任何时候,只要零初始化链表,空闲链表和后备链表的页框数低于允许的最小值(由内核变量 MmMinimum Freepages 指定)时,一个“修改页写回程序”的系统线程被唤醒,将修改链表中的页面写回磁盘,然后这些页框可以放入后备链表。当修改链表太大时,“修改页写回程序”也开始工作,把修改链表中的页面写回页文件中,限制链表的规模过大。如果把修改页写回磁盘后,系统掌握的可用页框还太少,内存管理器就开始把每个进程的工作集调整到最小规模,新淘汰的页被放到修改链表或后备链表中。在从后备或修改链表中移出页框之前,必须先修改仍在指向该页框的相应进程的页表项或原型 PTE 的值为无效,为了做到这一点,页框号数据库项中包含了指向原进程页表项或原型 PTE(共享时)的指针。

5. 页面淘汰算法与工作集管理

Windows 2000/XP 采用请页式和页簇化调页技术,当一个线程发生缺页中断时,内存管理器引发中断的页面及其后续的少量页面一起装入内存。根据局部性原理,这种页簇化策略能减少线程引发的缺页中断次数,减少调页 I/O 的数量。缺省页面读取簇的数量取决于物理内存的大小,当内存大于 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。

Windows 2000/XP 采用局部 FIFO 算法(在多处理器系统中)。采用局部淘汰可防止客户进程损失太多内存;采用 FIFO 算法可让被淘汰的页在淘汰后在物理内存中会停留一段时间,因此,如果马上又用到该页的话,就可很快将该页回收,而无需从磁盘读出。

Windows 2000/XP 对一个进程工作集的定义为“该进程当前在内存中的页面的集合”。当创建一个进程时,系统为其指定最小工作集(最少页框)和最大工作集(最多页框)。开始时所有进程缺省工作集的最小和最大值是相同的。系统初始化时,会计算一个进程最小和最大工作集值,当物理内存大于 32 MB(Windows 2000/XP server 大于 64 MB)时,进程缺省最小工作集为 50 页,最大工作集为 345 页。在进程执行过程中,内存管理器会对进程工作集大小进行自动调整。

当一个进程的工作集降到最小后,如果该进程再发生缺页中断,并且内存并不满,系统就会增加该进程的工作集尺寸。当一个进程的工作集升到最大后,如果没有足够的内存可用,则该进程每发生一次缺页中断,系统都要从该进程工作集中淘汰掉一页,再调入此次页中断所请求的页面。当然,如果有足够内存可用的话,系统也允许一个进程的工作集超过它的最大工作集尺寸。

当物理内存剩余不多时,系统将检查内存中的每个进程,其当前工作集是否大于其最小工作集,是则淘汰该进程工作集中的一些页,直到空闲内存数量足够或每个进程都达到其最小工作集。

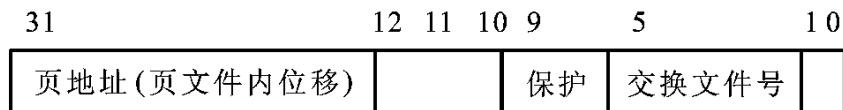
系统定时从进程中淘汰一个有效页,观察其是否对该页发生缺页中断,以此测试和调整进程当前工作集的合适尺寸。如果进程继续执行,并未对被淘汰的这个页发生缺页中断,则该进程工作集减 1,该页框被加到空闲链表中。

综上所述,Windows 2000/XP 的虚存管理系统总是为每个进程提供可能好的性能,而无需用户或系统管理员的干预。尽管这样,系统还提供 Win32 函数 Set process working Set,可让用户或系统管理员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计算出并保存的最大值。

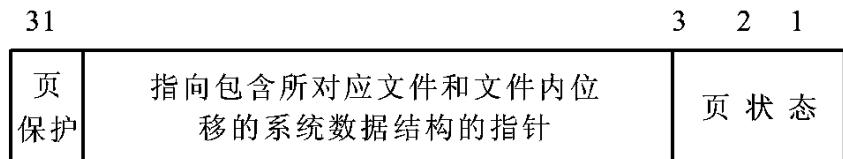
6. 盘交换区管理

Windows 2000/XP 可以支持多达 16 个盘交换文件(又称调页文件),处于文件中的页的

页表项格式见图 4-50(a);当代码或内存映射文件(不占盘交换区)所对应的页因淘汰而暂时不在物理内存时,其表项格式见图 4-50(b)。



(a) 位于盘交换区的页的页表项格式



(b) 当程序代码或内存映射文件所对应的页面被淘汰而暂不在内存时页的页表项格式

图 4-50 无效页表项

7. 其他内存管理机制

(1) 锁内存

可以通过两种方式将页面锁在内存中:

- 设备驱动程序调用核心态函数锁住所需页面;

Win32 应用程序可以通过系统调用锁住进程工作集中的页面,一个进程可以锁住的页面数不能超过它的最小工作集尺寸减 8。

(2) 分配粒度

内存管理按照系统分配粒度(allocation granularity)定义的整型边界,对齐每个保留的进程地址空间区域,通常这一值为 64 KB。进程保留地址空间时,系统能保证区域大小是系统页大小的倍数。例如,x86 系统使用 4 KB 的页,若你试图保留 18 KB 大小的内存区域,x86 系统在实际存储上将分配 20 KB。

(3) 内存页在级保护机制

Windows 2000/XP 提供了内存保护机制,防止用户无意或有意地破坏其他进程或操作系统,共提供 4 种保护方式:

- 区分核心态和用户态,核心态组件使用的数据结构和内存缓冲池只能在核心态下被线程访问,用户态线程不能访问。
- 每个进程只有独立、私有的虚拟地址空间,禁止其他进程的线程访问(除了共享页面或另一进程已被授权限),系统通过虚拟地址映射机制来保证这一点。
- 以页面为单位的保护机制,页表中包含了页级保护标志,如只读、读写等,以决定用户态和核心态可访问的类型,实现访问监控。

- 以对象为单位的保护机制。每个区域对象具有附加的标准存取控制 ACL (Access Control List), 当一个进程试图打开它时, 会检查 ACL, 以确证该进程是否被授权访问该对象。

4.8 实例研究:Linux 虚拟存储管理

4.8.1 Linux 虚拟存储管理概述

本节基于 Pentium 平台讨论 Linux 的虚拟存储管理。在 Linux 中, 每一个用户进程都可以访问 4 GB 的线性虚拟内存地址空间。其中从 0 到 3 GB 的虚拟内存地址是用户空间, 用户进程独占并可以直接对其进行访问。从 3 GB 到 4 GB 的虚拟内存地址是内核态空间, 由所有核心态进程共享, 存放仅供内核态访问的代码和数据, 用户进程处于用户态时不能访问。当中断或系统调用发生时, 用户进程进行模式切换(处理器特权级别从 3 转为 0), 即操作系统把用户态切换到内核态。

所有进程从 3 GB 到 4 GB 的虚拟内存地址都是一样的, 有相同的页目录项和页表, 对应到同样的物理内存段, Linux 以此方式让内核态进程共享代码段和数据段。Linux 采用请求页式技术管理虚拟内存。页表分为三级结构: 页目录 PGD (Page Directory)、中间页目录 PMD (Page Middle Directory) 和页表 PTE (Page Table)。在 Pentium 计算机上它被简化成两层, PGD 和 PMD 合二为一。页目录 PGD 和页表 PTE 都含有 1024 个项, 页面大小为 4 KB。

每一个进程都有一个页目录, 其大小为一个页面, 页目录中的每一项指向中间页目录的一页, 每个活动进程页目录必须在内存中。中间页目录可能跨多个页, 它的每一项指向页表中的一页。页表也可能跨多个页, 每个页表项指向该进程的一个虚页。

当使用 `fork()` 创建一个进程时, 分配内存页面的情况如下: 进程控制块 1 页, 内核态堆栈 1 页, 页目录 1 页, 页表若干页。而使用 `exec()` 系统调用时, 分配内存页面的情况如下: 可执行文件的文件头 1 页, 用户堆栈的 1 页或几页。

这样, 当进程开始运行时, 如果执行代码不在内存中, 将产生第 1 次缺页中断, 让操作系统参与分配内存, 并将执行代码装入内存。此后按照需要, 不断的通过缺页中断调进代码和数据。当系统内存资源不足时, 由操作系统决定是否调出一些页面。

4.8.2 Linux 进程的虚拟地址空间

Linux 将每个用户进程 4 GB 的虚拟地址空间划分成 2 KB 或 4 KB(缺省方式为 4 KB)固定的大小的页面, 采用分页方式进行管理。系统的管理开销相当大, 仅一个进程 4 GB 空间

的页表将占用 4 MB 物理内存。事实上目前也没有应用进程大到如此规模,因此,有必要显式地表示真正被进程使用到的那部分虚拟地址空间。这样一来,虚拟地址空间就由许多个连续虚地址区域构成,Linux 中采用了虚存段 vma(virtual memory area)及其链表来表示一个进程实际用到的虚拟地址空间。

一个 vma 是某个进程的一段连续的虚存空间,在这段虚存里的所有单元拥有相同特征,如属于同一进程、相同的访问权限、同时被锁定、同时受保护等等。但两个 vma 未必连续,并且它们的保护模式也可以不同。对于一个给定的进程,两个 vma 决不会重叠,一个地址最多被一个 vma 所覆盖。vma 段由数据结构 vm_area_struct 描述:

```
struct vm_area_struct
{
    struct mm_struct *vm_mm; /* 指向虚存段所属进程的 mm-struct */
    unsigned long vm_start; /* 虚存段始址 */
    unsigned long vm_end; /* 虚存段末址 */
    pgprot_t vm_page_prot; /* 虚存段读写权限等标志 */
    unsigned short vm_flags; /* 按地址分叉的 vma 的 AVL 树 */
    short 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 */
    /* 按地址顺序排列的 vma 链的下一个节点指针 */
    struct vm_area_struct *vm_next;
    /* 或是在一个结点区双向环链表中,或在内存区映射表中,或未用 */
    struct vm_area_struct *vm_next_share; /* 共享同一个文件的下一个 vma */
    struct vm_area_struct *vm_prev_share; /* 共享同一个文件的前一个 vma */
    /* 用于共享内存 */
    struct vm_operations_struct vm_ops;
    /* 本虚存段封装的操作如 open,close */
    unsigned long vm_offset; /* 相对文件中共享内存起点的位移 */
    struct inode *vm_inode; /* 指向文件 inode 或 NULL */
    unsigned long vm_pte;
};
```

进程通常占用几个 vma,分别用于代码段、数据段、堆栈段等。在 Linux 中对 vma 是如下管理的,每当创建一个进程时,系统便为其建立一个 PCB,称 task_struct 结构。在这个结构中内嵌了一个包含此进程存储管理有关信息的 mm_struct 结构,它被用来描述一个进程的

虚拟内存的总的情况,其定义如下。

```
struct mm_struct
{
    int count;                                /* 进程在内存中有映象的 vma 个数 */
    pgd_t *pgd;                               /* 进程一级页表指针 */
    unsigned long context;                     /* 进程运行的环境 */
    unsigned long start_code, end_code, start_data, end_data;
    /* 代码和数据段的起始地址及结束地址 */
    unsigned long start_brk, brk, start_stack, start_mmap;
    /* 与堆栈段及可用空间有关数据 */
    unsigned long arg_start, arg_end, env_start, env_end;
    /* 参数和环境有关的信息 */
    unsigned long rss, total_vm, locked_vm;    /* 进程占用的总页框数 */
    unsigned long def_flags;                   /* 有关特征位 */
    struct vm_area_struct mmap;               /* 进程 vma 链头指针 */
    struct vm_area_struct *mmap_avl;           /* 进程 vma 的 AVL 树的根节点 */
    struct semaphore mmap_sem;
    /* 对进程的 mm_struct 操作时的信号量 */
};
```

图 4-51 是进程的虚存管理数据结构。从每个进程控制块的内嵌 mm_struct 可以找到内存管理数据结构,从内存管理数据结构的指向 vma 段的链接指针 mmap 就可找到按照升序用 vm_next 链接起来的进程的所有 vma。此外,每个进程都有一个目录 pgd, 存储该进程所使用的内存页面情况,Linux 根据缺页调度原则只分配用到的内存页面,从而避免了页表占用过多的物理内存空间。

在 Linux 中,进程 vma 较少的情况下,可用单链表管理 vma,但插入和删除不太方便。随着进程虚存段的增多超过 AVL-MIN-MAP-COUNT(通常为 32 个)时,为了提高对 vma 的查找速度和操作效率,系统会维护一个 avl 树(Adelson-velskii and Landis 树)。在树中所有的 vm_area_struct 虚存段均有左指针 vm_area_left 指向相邻的低地址虚存段,右指针 vm_area_right 指向相邻的高地址虚存段。struct mm_struct 结构成员 struct vm_area_struct * mmap_avl 表示进程 avl 树的根,vm_avl_height 表示 avl 树的高度。这样管理效率提高了,但系统的开销同时也增大了。

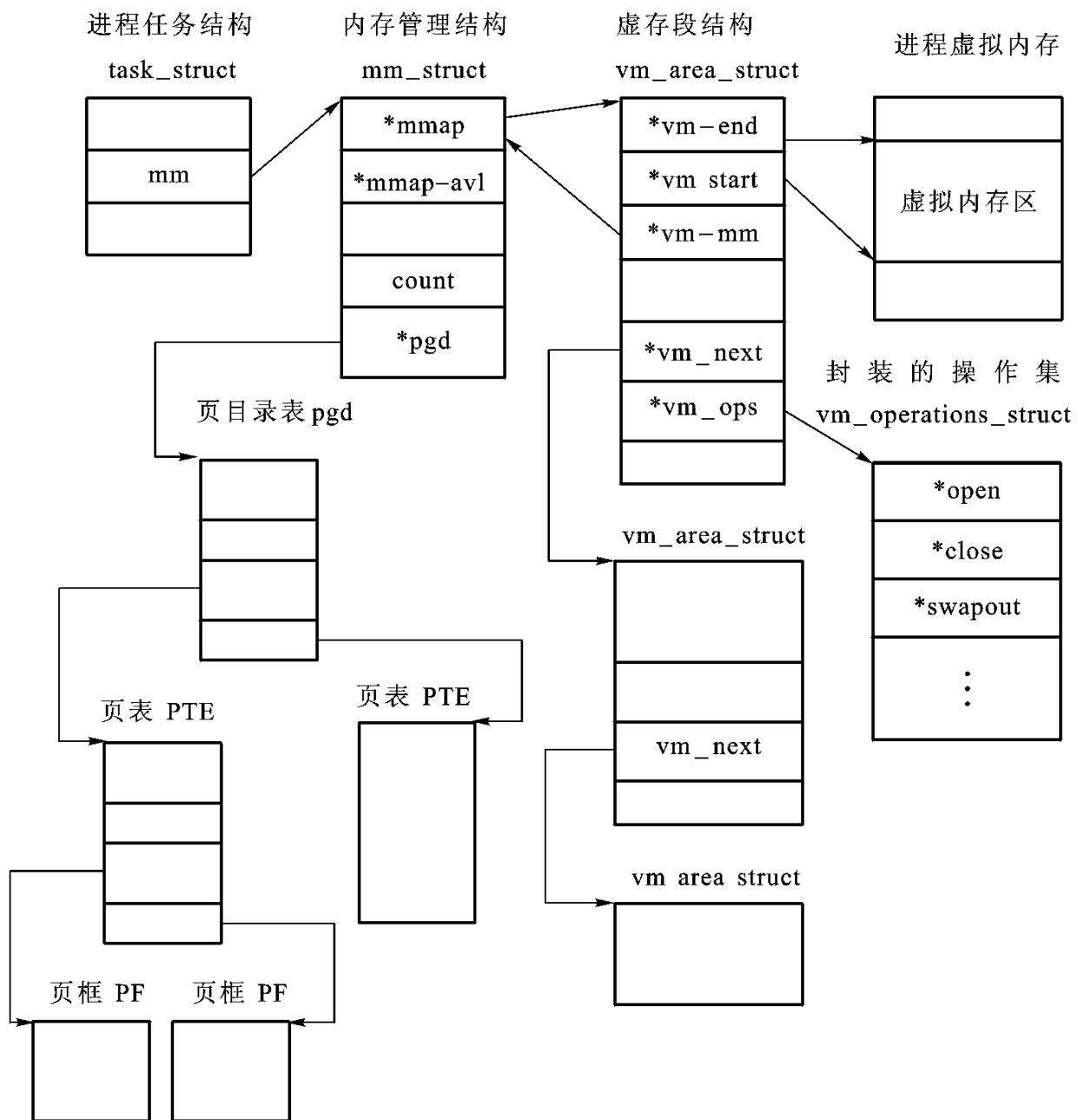


图 4-51 进程虚存管理数据结构

4.8.3 Linux 物理内存空间的管理

在 Linux 操作系统控制下,物理内存划分成页框,其长度与页面相等。系统中的所有物理页框都由 `men-map` 表描述,它在系统初始化时通过 `free-area-init()` 函数创建。`men-map` 本身是由 `men-map-t` 组成的一个数组,每个 `men-map-t` 描述一个物理页框,用于对空闲页框的管理,其定义为:

```
typedef struct page
```

```

{
    struct page * next, * prev;          /* 下一个和前一个空闲页框 */
    struct inode * inode;               /* 页框内存放代码或数据所属文件的 inode */
    unsigned long offset;              /* 页框内存放代码或数据所属文件的位移 */
    struct page * next _ hash;         /* page 快存的 hash 表中, 链表后继指针 */
    atomic_t count;                   /* 访问此页框的进程计数 */
    unsigned flags;                   /* 一些标志位 */
    unsigned dirty;                  /* 页框修改标志 */
    unsigned age;                     /* 页框年龄, 越小越先换出 */
    struct wait_queue * wait;
    struct page * prev _ hash;        /* page 快存的 hash 表中, 链表前向指针 */
    struct buffer_head * buffers;     /* 若页框作为缓冲区, 则指示地址 */
    unsigned long swap _ unlock _ entry;
    unsigned long map _ nr;           /* 页框在 mem _ map 表中的下标 */
}mem _ map _ t;

```

在物理内存的低端, 紧靠 mem _ map 表的 bitmap 以位示图方式记录了所有物理页框的空闲状况, 该表也在系统初始化时由 free _ area _ init() 函数创建。与一般位示图不同, bitmap 分割成 NR _ MEM _ LISTS(缺省时为 6) 组。首先是第 0 组, 每位表示 1 个页框的空闲状况, 置位表示已被占用。接着是第一组, 每位表示 2 个页框的空闲状况, 置 1 表示其中 1 个或 2 个页框已被占用。

Linux 用 free _ area 数组记录空闲物理页框, 该数组由 NR _ MEM _ LISTS 个 free _ area _ struct 结构类型的数组元素构成, 每个元素均作为一条空闲链表头。

```

struct free _ area _ struct
{
    struct page * next , * prev ;      /* 此结构的 next 和 prev 指针与 struct page 匹配 */
    unsigned int * map;
};

static struct free _ area _ struct free _ area[NR _ MEM _ LISTS];

```

与 bitmap 的分配方法一样, 所有单个空闲页框组成的链表链到 free _ area 数组的第 0 项后面, 连续 2^i 个空闲页框被挂到 free - area 数组的第 i 项后面。Linux 采用伙伴(buddy)算法分配空闲块, 块长可以是 2^i 个 ($0 \leq i < \text{NR_MEM_LISTS}$) 页框, 页框的分配由 get _ free _ pages() 执行, 释放页框可以用 free _ pages() 函数执行。

当分配长度是 2^i 页框的块时, 从 free _ area 数组的第 i 条链表开始搜索, 找不到再搜索

第 $i + 1$ 条链表, 以此类推。若找到的空闲块长正好等于需求的块长, 则直接将它从 free_area 中删除, 返回第一个页框的地址。若找到的空闲块大于需求的块长, 则将空闲块前半部分插入 free_area 中前一条链表, 取后半部分, 若还大于, 则继续对半分, 留一半取一半, 直到相等。同时, bitmap 表从第 i 组开始的对应位要置 1。

回收内存时, change_bit() 函数根据 bitmap 表的对应组, 判断回收块的前后邻块是否也为空闲。若空则要合并, 并修改 bitmap 表的对应位, 并从 free_area 的空闲链表中取下该相邻块。这一算法可递归进行, 直到找不到空闲邻块为止, 将最后合并成的最大块插入 free_area 的相应链表中。

4.8.4 用户态内存的申请与释放

用户进程可以使用 vmalloc() 和 vfree() 函数申请和释放大块存储空间, 分配的存储空间在进程的虚地址空间中是连续的, 但它对应的物理页框仍需经缺页中断后, 由缺页中断处理例程分配, 所分配的页框也不是连续的。

可分配的虚地址空间由常量 VMALLOC - START 和 VMALLOC - END 划定范围, 在 3 GB + high_memory + HOLE_8M 以上的端, 由 vmlist 表管理。3 GB 是内核态赖以访问的物理内存始址, high_memory 是安装在机器中实际可用的物理内存的最高地址, 因而, 3 GB + high_memory 也就是虚拟地址空间中看到的物理内存上写。HOLE_8M 则为长度 8 MB 的“隔离带”, 起到越写保护作用。这样, vmlist 管辖的虚地址空间既不与进程用户态 0 到 3 GB 虚地址空间冲突, 也不与进程内核映射到物理空间的 3 GB 到 3 GB + high_memory 的虚地址空间冲突。

尽管 vmalloc() 返回高于任何物理地址的高端地址, 但因为同时更改页表或页目录, 处理器仍能正确访问这些高端连续地址。

vmlist 链表的节点类型 vm_struct 定义为:

```
struct vm_struct
{
    unsigned long flags;           /* 虚拟内存块占用标志 */
    void * addr;                  /* 虚拟内存块的起址 */
    unsigned long size;            /* 虚拟内存块的长度 */
    struct vm_struct * next;       /* 下一个虚拟内存块 */
};

static struct vm_struct * vmlist = NULL;
```

初始时, vmlist 仅一个节点, vmlist.addr 置为 VMALLOC - START。动态管理过程中, vmlist

的虚拟内存块按起始地址从小到大排序,采用首次适应算法查找一个可分配区域,每个虚拟内存块之后都有一个 4 KB 大小的“隔离带”,用以检测访问指针的越界错误。

用户进程申请和释放块连续虚拟内存分别使用 `vmalloc()` 和 `vfree()` 函数,其执行过程大致如下:申请时需给出申请的长度;然后,调用 `set-vm-area` 内部函数向 `vmlist` 索取虚存空间;如果申请成功,将会在 `vmlist` 中插入一个 `vm-struct` 结构,并返回首地址,当申请到虚地址空间后更改页目录和页表。释放时需给出虚拟空间首地址,沿着 `vmlist` 搜索要释放的区域,找到表示该虚拟内存块的 `vm-struct` 结构,并从 `vmlist` 表中删除,同时清除与释放虚存空间有关的目录项和页表项,如果搜遍链表没有找到匹配项则发出一个警告信息并结束。

4.8.5 内存的共享和保护

Linux 中内存共享是以页共享的方式实现的,共享该页的各个进程的页表项直接指向共享页,这种共享不需建立共享页表,节省内存空间,但效率较低。当共享页状态发生变化时,共享该页的各进程页表均需修改,要多次访问页表。

Linux 可以对虚存段中的任一部分加锁或保护,对进程的虚拟地址加锁,实质就是对 `vma` 段的 `vm_flags` 属性与 `VM_LOCKED` 进行或操作。虚存加锁后,它对应的物理页框驻留内存,不再被页面置换程序换出。加锁操作共有 4 种:对指定的一段虚拟空间加锁或解锁(`mlock` 和 `munlock`),对进程所有的虚拟空间加锁或解锁(`mlockall` 和 `munlockall`)。

对进程的虚拟地址空间实施保护操作,就是重新设置 `vma` 段的访问权限,实质就是对 `vma` 段的 `vm_flags` 重置 `PROT_READ`、`PROT_WRITE` 和 `PROT_EXEC` 参数,并重新设定 `vm_page_prot` 属性。与此同时,对虚拟地址范围内所有页表项的访问权限也做调整,保护操作由系统调用 `mprotect` 实施。

4.8.6 交换空间、页面换出和调入

1. 交换空间

计算机的物理内存是影响机器性能的关键因素。相对于以 GB 计算的硬盘空间,内存的容量显得太少,尤其在多任务系统中更是如此。所以,存储管理系统应该设法把暂时不用的内存数据转储到外存中。早期操作系统的解决方法是“对换”,即把暂时没有拥有 CPU 的进程整体性地转存到外存空间,直到进程重新获得 CPU 之后才被整体装回内存。显然这一交换操作会影响效率。70 年代后,按需调页算法得到应用,该算法以页为单位进行转出和调入,大幅度提高了读写效率,Linux 也采用此策略进行虚拟存储管理。

在 Linux 中,内核态内存空间的内容不允许对换,道理很简单,因为驻留该空间的函数和数据结构都用于系统管理,有的甚至是为虚拟存储管理服务的,必须时刻准备着给 CPU

引用。

Linux 采用两种方式保存换出的页面。一是使用整个块设备,如硬盘的一个分区,称作交换设备;另一种是使用文件系统的一个固定长度的文件,称作交换文件。两者统称为交换空间。

交换设备和交换文件的内部格式是一致的。前 4096 个字节是一个以字符串“SWAP_SPACE”结尾的位图。位图的每一位对应于一个交换空间的页面,置位表示对应的页面可以用于换页操作,第 4096 字节之后是真正存放换出页面的空间。这样每个交换空间最多可以容纳 $(4096 - 10) \times 8 - 1 = 32687$ 个页面。如果一个交换空间不够用,Linux 最多允许管理 MAX_SWAPFILES(缺省值为 8) 个交换空间。

交换设备远比交换文件更加有效。在交换设备中,属于同一页面的数据总是连续存放的,第一个数据块地址一经确定,后续的数据块可以按照顺序读出或写入。而在交换文件中,属于同一页面的数据虽然在逻辑上是连续的,但数据块的实际存储位置可能是分散的,需要通过交换文件的 inode 检索,这决定于拥有交换文件的文件系统。在大多数文件系统中,交换这样的页面,必须多次访问磁盘扇区,这意味着磁头的反复移动、寻道时间的增加和效率的降低。

可以使用 swapon 系统调用向内核注册一个交换设备或交换文件。

2. 页交换进程和页面换出

当物理页面不够用时,Linux 存储管理系统必须释放部分候选替换物理页面,把它们的内容写到交换空间。内核态交换线程 kswapd 专门负责完成这项功能,注意内核态线程是没有虚拟空间的线程,它运行在内核态,直接使用物理地址空间。kswapd 不仅能够把页面换出到交换空间,也能保证系统中有足够的空闲页面以保持存储管理系统高效的运行。

kswapd 在系统初启时由 init 创建,然后调用 init_swap_timer() 函数进行设定时间间隔,并马上转入睡眠状态。当定时器时间到后,kswapd 被激活,它首先察看系统中空闲页面 nr_free_pages 是否变得太少,利用两个变量 free_pages_high 和 free_pages_low 进行判断。如果空闲页面数小于 free_pages_high,就要有页面被交换出去;如果空闲页面数小于 free_pages_low,kswapd 不仅换出部分页面,还要把睡眠时间减为平时的一半,以便频繁换出页面;当空闲页面数大于 free_pages_low 时,睡眠时间又会恢复原态。Kswapd 的页面交换利用 do_try_to_free_pages 函数实现,该函数首先调用 kmem_cache_reap 减少内核 cache 中不用的空闲块,然后,循环地依次从三条途径缩减系统使用的物理页面:

- 缩减 page cache 和 buffer cache,调用 shrink_mmap,并采用 clock 算法实现。
- 换出共享内存占用的页面,调用 shm_swap 实现。
- 换出或丢弃进程占用的页面,调用 swap_out,并采用 clock 算法实现。

3. 缺页中断和页面调入

磁盘中的可执行文件映像一旦被映射到一个进程的虚拟空间, 它就开始执行。由于一开始只有该映像区的开始部分被调入内存, 因此, 进程迟早会执行那些未被装入内存的部分。当一个进程访问了一个还没有有效页表项的虚拟地址时, 处理器将产生缺页中断, 通知操作系统, 并把缺页的虚拟地址(保存在 CR2 寄存器中)和缺页时访问虚存的模式一并传给 Linux 的缺页中断处理程序。

系统初始化时首先设定缺页中断处理程序为 `do_page_fault()`, 它根据控制寄存器 CR2 传递的缺页地址, 然后, 进入 `error_code` 处理程序进行分类, 通过 `find_vma` 来找到发生页面失误的虚拟存储区地址所在的 `vm_area_struct` 结构指针。如果没有找到, 那么, 说明进程访问了一个非法存储区, 系统将发出一个信号告知进程出错。然后, 系统检测缺页时访问模式是否合法, 如果进程对该页的访问超越权限, 系统也将发出一个信号, 通知进程的存储访问出错。通过以上两步检查, 可以确定缺页中断是否合法, 进而进程进一步通过页表项中的位 P 来区分缺页对应的页面是在交换空间($P = 0$ 且页表项非空)还是在磁盘中某一执行文件映像的一部分。最后, 进行页面调入操作。

Linux 使用最少使用频率替换策略, 页替换算法在 `clock` 算法基础上作了改进, 使用位被一个 8 位的 `age` 变量所取代。每当一页被访问时, `age` 增加 1。在后台由存储管理程序周期性地扫描全局页面池, 并且当它在主存中所有页间循环时, 对每个页的 `age` 变量减 1。`Age` 为 0 的页是一个“老”页, 已有些时候没有被使用, 因而可用作页替换的候选者。`Age` 值越大, 该页最近被使用的频率越高, 也就越不适宜于替换。

4.8.7 缓冲机制

Linux 虚存管理的缓冲机制主要包括:`kmalloc cache`、`swap cache` 和 `page cache`。

1. swap cache

如果以前被调出到交换空间的页面由于进程再次访问而调入物理内存, 只要该页面调入后未被修改过, 那么它的内容与交换空间中内容是一样的。这种情况下, 交换空间中的备份是有效的。因此, 该页再度换出时, 就没有必要执行写操作。

Linux 采用 `swap-cache` 表描述的 `swap cache` 来实现这种思想。`swap cache` 实质上是关于页表项的一个列表, 表的首地址为:

```
vnsigned long * swap-cache;
```

每一物理页框都在 `swap-cache` 中占有一个表项, 该表项的总数就是物理页框总数。若该页框的内容是新创建的; 或虽曾换出过; 但换出后, 该页框已被修改过时, 则该表项清 0。内容非 0 的表项, 正好是某个进程的页表项, 描述了页面在交换空间中的位置。当一个物理

页框调出到交换空间时,它先查 swap cache。如果其中有与该面对应有效的页表项,那就不需要将该页写出,因为,原有空间中内容与待换出的页面是一致的。

2. page cache

页缓冲的作用是加快对磁盘文件的访问速度。文件被映射到内存中,每次读取一页,而这些页就保存到 page cache 中,Linux 用 hash 表 page – hash – table 来访问 page cache,它是指向由 page 类型节点组成的链表指针。

每当需要读取文件一页时,总是先通过 page cache 读取。如果所需页面就在其中,就通过指向表示该页面的 mem – map – t 的指针;否则必须从申请一个页框,再将该页从磁盘文件中调入。如果可能的话,Linux 还发出预读一页的读操作请求,根据程序局部性原理,进程在读当前页时,它的下一页很可能被用到。

随着越来越多的文件被读取、执行,page cache 会越来越大。进程不再需要的页面应从 page cache 中删除。当系统发现内中的物理页框越来越少时,它将缩减 page cache。

3. kmalloc cache

进程可用 kmalloc() 和 kfree() 函数向系统申请较小的内存块,而这两个函数共同维护了一个称作 kmalloc cache 的缓冲区,其主要目的是加快释放物理内存的速度。

4.9 本 章 小 结

存储器是计算机系统的重要组成部分,存储空间是操作系统管理的宝贵资源,虽然其容量在不断扩大,但仍远远不能满足软件发展的需要。对存储资源进行有效管理,不仅关系到存储器的利用率,而且还对操作系统的性能和效率有很大影响。

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、存储扩充。存储分配指为选中多道运行的作业分配主存空间;地址转换是把逻辑地址空间中的用户程序通过静态重定位或动态重定位转换和映射到分给的物理地址空间中,以便用户程序的执行;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用户进程共享。

内存管理的重要任务之一是解决好存储扩充问题,使得主存利用率得以提高,使得主存中存放尽量多的进程,使得用户程序不受可用物理内存大小的限制。最简单的单道系统中,一旦一个程序能装入主存,它将一直运行直到结束。如果程序长度超出主存实际容量,可以通过覆盖和交换的技术获得解决。更多操作系统支持多个用户进程在主存同时执行,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,又分固定分区和可变分区,并介

绍了可变分区的分配算法包括:最先适应、下次适应、最佳适应、最坏适应和快速适应等分配算法。

采用分区方式管理存储器,每道程序总是要求占用主存的一个或几个连续存储区域,主存中会产生许多碎片。因此,有时为了接纳一个新的作业而往往要移动已在主存的信息,这不仅不方便,而且开销不小。现代计算机都有某种虚存硬设备支持,简单也是常用的虚存是请求分页式虚存管理,于是允许把一个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同时装入主存,而是仅将其中当前使用部分先装入主存储器,其余暂时不用的部分先存放在作为主存扩充的辅存中,待用到这些信息时,再由系统自动把它们装入到主存中,简而言之,它采用了自动的部分装入、部分对换的技术、主存辅存独立编址但统一使用的技术,使得进程的虚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方便。本章中讨论了分页式虚拟存储器的基本原理,包括:页面、页框、逻辑地址、页表、地址转换等概念,也介绍了用于加快地址映射的相联存储器、页表很大时使用的多级页表、某些系统使用的反置页表。通过虚拟存储器,所有的用户访问程序和数据都给出逻辑地址,在运行时由系统转换成物理地址,这叫动态地址转换。于是就允许一个进程位于主存的任何地址,它的位置还可以动态改变。已经设计出许多页面置换算法:Optimal、LRU、二次机会、Clock、LFU 等。Optimal 算法虽好但不可行,两种较好的算法是 LRU 和二次机会。在一个实际的操作系统中,还有一些重要的问题:

- 页面装入策略 决定页面何时被装入主存,有请页式和预调式两种装入策略。
- 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。
- 页面分配策略 根据进程生命周期中分给的页框数是否改变,分为固定分配和可变分配。
- 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。

有些系统采用可变分配与局部替换相结合的方法,取得较好效果。此外,还讨论了工作集模型等,可以通过工作集模型来指导确定进程常驻集的大小,使得进程缺页中断率低,内存空间又得到充分利用。分页式虚存管理不能支持模块化程序设计,实现页面共享也存在问题。

另一种虚拟存储器称段式虚存,它是为满足现代高级语言模块化程序设计所需的二维地址要求和方便用户(程序员)编程和使用而引进的存储管理技术,段划分的基本原则是按用户应用中逻辑上有完整意义的内容进行分段。为了实现段式存储管理,需要建立每个作业的段表,记录用户逻辑段与主存中物理段的对应关系,段式存储管理主存的分配与去配和

可变分区方式十分类似,可以通过直接分配、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这就是段页式虚存管理,并介绍了它们的特点优点,以及实现原理。

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使用需要,采用各种相应的有效存储资源分配策略和保护措施。

习 题 四

一、思考题

1. 简述存储管理的基本功能。
2. 叙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)?有哪些方法可以实现地址转换?
5. 分区存储管理中常用哪些分配策略?比较它们的优缺点。
6. 什么是移动技术?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,试画出分配和释放一个存储区的算法流程
8. 什么是存储保护?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. 试给出几种存储保护方法,各运用于何种场合?
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,说明之;如果是,试设计一种解决死锁的方案。
17. 试论述分页式存储管理中,决定页面大小的主要因素。
18. 叙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硬件/软件设施支撑。
20. 如果主存中某页正在与外围设备交换信息,那么,发生缺页中断时,可以将该页淘汰吗?为什么?出现这种情况时,你能提出什么样的处理办法?
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. 叙述段页式存储器的主要优缺点。
24. 叙述虚存管理与实存管理的主要区别。
25. 叙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. 什么叫“抖动”?试给出一个抖动的例子。
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. 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,主存及辅存容量之和的关系?
30. 分页式存储管理中,试分析大页面与小页面各自的优点。
31. 试述缺页中断与一般中断的区别。
32. 试述虚拟存储器与辅助存储器之间的关系。
33. 在请页分页虚拟存储的替换算法中,对于任何给定的驻留集尺寸,什么样的引用串情况下,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. 解决大作业和小内存的矛盾有哪些途径?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:CPU20%、分页磁盘97.7%、其他外设50%。试问哪些措施可以改善CPU的利用率?
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。
37. 用一个在循环轨道上来回移动的雪犁机来模拟说明:雪均匀地落在轨道上,雪犁机以恒定的速度在轨道上不断地循环,轨道上被扫落的雪从系统中消失。试问这种模拟与哪一种替换算法相似?这种模拟说明了替换算法的哪些行为?

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 个页框的情况下,分别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。

若分配给该作业三个页框,分别采用 FIFO 和 LRU 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。

当分配给该作业的物理块数分别为 3 和 4 时,试计算访问过程中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,按地址排列的内存空闲区为:10 K、4 K、20 K、18 K、7 K、9 K、12 K 和 15 K。对于下列的连续存储区的请求:(1)12 K、10 K、9 K, (2)12 K、10 K、15 K、18 K 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, 哪个空闲区被使用?

5. 给定内存空闲分区, 按地址从小到大为: 100 K、500 K、200 K、300 K 和 600 K。现有用户进程依次分别为 212 K、417 K、112 K 和 426 K, (1) 分别用 first-fit、best-fit 和 worst-fit 算法将它们装入到内存的哪个分区? (2) 哪个算法能最有效利用内存?

6. 一个 32 位地址的计算机系统使用二级页表, 虚地址被分为 9 位顶级页表, 11 位二级页表和偏移。试问: 页面长度是多少? 虚地址空间共有多少个页面?

7. 一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, 在内存有 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、内存、辅存来实现虚拟存储器。如果数据在缓存中, 访问它需要 A_{ns} ; 如果在内存但不在缓存, 需要 B_{ns} 将其装入缓存, 然后才能访问; 如果不在内存而在辅存, 需要 C_{ns} 将其读入内存, 然后, 用 B_{ns} 再读入缓存, 然后才能访问。假设缓存命中率为 $(n-1)/n$, 内存命中率为 $(m-1)/m$, 则数据平均访问时间是多少?

9. 某计算机有 cache、内存、辅存来实现虚拟存储器。如果数据在 cache 中, 访问它需要 20 ns; 如果在内存但不在 cache, 需要 60 ns 将其装入缓存, 然后才能访问; 如果不在内存而在辅存, 需要 12 ms 将其读入内存, 然后, 用 60 ns 再读入 cache, 然后才能访问。假设 cache 命中率为 0.9, 内存命中率为 0.6, 则数据平均访问时间是多少(ns)?

10. 有一个分页系统, 其页表存放在主存里, (1) 如果对内存的一次存取要 $1.2 \mu s$, 试问实现一次页面访问的存取需花多少时间? (2) 若系统配置了联想存储器, 命中率为 80% , 假定页表表目在联想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 | 2300 | 14 |
| 2 | 90 | 100 |
| 3 | 1327 | 580 |
| 4 | 1952 | 96 |

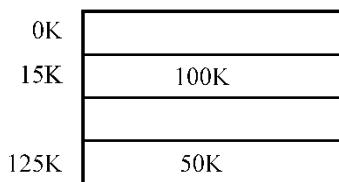
给定地址为段号和位数: 1) [0, 430]、2) [3, 400]、3) [1, 1]、4) [2, 500]、5) [4, 42], 试求出对应的内存物理地址。

12. 某计算机系统提供 24 位虚存空间, 主存为 $2^{18} B$, 采用分页式虚拟存储管理, 页面尺寸为 1 KB。假定用户程序产生了虚拟地址 11123456(八进制), 而该页面分得块号为 100(八进制), 说明该系统如何产生相应的物理地址及写出物理地址。

13. 主存中有两个空间区如图所示,

现有作业序列依次为: Job1 要求 30 K; Job2 要求 70 K; Job3 要求 50 K; 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? 为什么?

14. 设有一页式存储管理系统, 向用户提供的逻辑地址空间最大为 16 页, 每页 2 048 字节, 内存总共有 8 个存储块。试问逻辑地址至少应为多少位? 内存空间有多大?



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 字节,现有一逻辑地址为 2F6AH,且第 0、1、2 页依次存在物理块 10、12、14 号中,问相应的物理地址为多少?

16. 有矩阵:VAR A:ARRAY[1…100, 1…100] OF integer;元素按行存储。在一虚存系统中,采用 LRU 淘汰算法,一个进程有 3 页内存空间,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。

程序 A:

```
FOR i := 1 TO 100 DO
  FOR j := 1 TO 100 DO
    A[i, j] := 0;
```

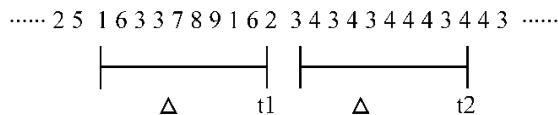
程序 B:

```
FOR j := 1 TO 100 DO
  FOR i := 1 TO 100 DO
    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. 在虚拟页式存储管理中,为解决抖动问题,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. 有一个分页虚存系统,测得 CPU 和磁盘的利用率如下,试指出每种情况下的存在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. 在一个分页虚存系统中,用户编程空间 32 个页,页长 1 KB,主存为 16 KB。如果用户程序有 10 页长,若已知虚页 0、1、2、3,已分到页框 8、7、4、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,每页的装入时间、最后访问时间、访问位 R、修改位 D 如下所示(时间用时钟点数表示):

| page | loaded | last ref | 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、二次机会算法分别淘汰哪一页？

22. 考虑下面的程序：

```
for (i = 0; i < 20, i++)
    for(j = 0; j < 10; j++)
        a[i] = a[i] × j
```

试举例说明该程序的空间局部性和时间局部性。

23. 一个有快表的页式虚存系统，设内存访问周期为 $1 \mu s$ ，内外存传送一个页面的平均时间为 5ms。如果快表命中率为 75%，缺页中断率为 10%。忽略快表访问时间，试求内存的有效存取时间为多少？

24. 假设某虚存的用户空间为 1 024 KB，页面大小为 4 KB，内存空间为 512 KB。已知用户的虚页 10、11、12、13 页分得内存页框号为 62、78、25、36，求出虚地址 0BEBC(16 进制)的实地址(16 进制)是多少？

25. 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内，：

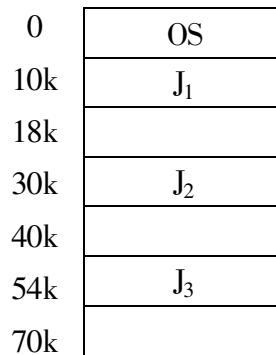
1) 若一次访问主存花 120 ns，那么，访问一个数据的时间是多少？

2) 若增加一个快表，在命中或失误时需有 20 ns 开销，如果快表命中率为 80%，则访问一个数据的时间为多少？

26. 设某系统中作业 J_1, J_2, J_3 占用主存的情况如图。今有一个长度为 20k 的作业 J_4 要装入主存，当采用可变分区分配方式时，请回答：

(1) J_4 装入前的主存已分配表和未分配表的内容。

(2) 写出装入 J_4 时的工作流程，并说明你采用什么分配算法。



27. 考虑下列的段表：

| 段号 | 始址 | 段长 |
|----|------|-----|
| 0 | 200 | 500 |
| 1 | 890 | 30 |
| 2 | 120 | 100 |
| 3 | 1250 | 600 |
| 4 | 1800 | 88 |

对下面的逻辑地址,求物理地址,如越界请指明。1) $<0, 480>$ 2) $<1, 25>$ 3) $<1, 14>$ 4) $<2, 200>$ 5) $<3, 500>$ 6) $<4, 100>$ 。

28. 请页式存储管理中,进程访问地址序列为:10, 11, 104, 170, 73, 305, 180, 240, 244, 445, 467, 366。试问1)如果页面大小为100,给出页面访问序列。2)进程若分得3个页框,采用FIFO和LRU替换算法,求缺页中断率?

29. 假设计算机有2M内存,其中,操作系统占用512K,每个用户程序也使用512K内存。如果所有程序都有70%的I/O等待时间,那么,再增加1M内存,吞吐率增加多少?

30. 一个计算机系统有足够的内存空间存放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时间被浪费掉了?

31. 如果一条指令平均需1ms,处理一个缺页中断另需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。

32. 一台计算机的内存空间为1024个页面,页表放在内存中,从页表中读一个字的开销是500ns。为了减少开销,使用了有32个字的快表,查找速度为100ns。要把平均开销降到200ns需要的快表命中率是多少?

33. 假设一条指令平均需花1μs,但若发生了缺页中断就需2001μs。如果一个程序运行了60秒,期间发生了15000次缺页中断,若可用内存是原来的两倍,这个程序运行需要多少时间?

34. 在分页式虚存管理中,若采用FIFO替换算法,会发生:分给作业页面越多,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。

35. 假设一个任务被划分成4个大小相等的段,每段有8项的页描述符表,若页面大小为2KB。试问段页式存储系统中:(a)每段最大尺寸是多少? (b)该任务的逻辑地址空间最大为多少? (c)若该任务访问到逻辑地址空间5ABCH中的一个数据,试给出逻辑地址的格式。

36. 已知某系统页面长4KB,页表项4B,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?

37. 在请求分页虚存管理系统中,若驻留集为m个页框,页框初始为空,在长为p的引用串中具有n个不同页面($n > m$),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。

38. 在请求分页虚存管理系统中,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2ms,若替换一个已修改过页面的缺页中断处理需另加写盘时间8ms,内存存取周期为1μs。假定70%被替换的页面被修改过,为保证有效存取时间不超过2μs,允许的最大缺页中断率为多少?

39. 若内存下按地址递增次序有三个不邻接的空闲区F1、F2、F3,它们的大小分别是:50K、120K和25

K。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。

40. 有两台计算机 P1 和 P2,它们各有一个硬件高速缓冲存储器 C1 和 C2,且各有一个主存储器 M1 和 M2。其性能为:

| | C1 | C2 | M1 | M2 |
|------|-------|-------|-----------|-------------|
| 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 |

若两台机器指令系统相同,它们的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冲存储器中存取到的概率 P 是 0.7,试问:这两台计算机哪个速度快?当 $P=0.9$ 时,处理器的速度哪个快?

第五章 设备管理

现代计算机系统中配置了大量外围设备。一般来说计算机的外围设备分为两大类：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，它在系统中作为主存储器的扩充，所以，又称辅助存储器；另一类是输入输出型设备（如显示器、卡片机、打印机等），它们把外界信息输入计算机，把运算结果从计算机输出。

设备管理是操作系统中最庞杂和琐碎的部分，普遍使用 I/O 中断、缓冲器管理、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外部设备和 CPU 速度的不匹配所引起的问题，使主机和外设并行工作，提高了使用效率。但是，在另一方面却给用户的使用带来极大的困难，它必须掌握 I/O 系统的原理，对接口和控制器及设备的物理特性要有深入了解，这就使计算机推广应用受到很大限制。

为了方便用户使用各种外围设备，设备管理要达到提供统一界面、方便使用、发挥系统并行性，提高 I/O 设备使用效率等目标。为此，设备管理通常应具有以下功能：

- 外围设备中断处理
- 缓冲区管理
- 外围设备的登记和使用情况跟踪以及分配和去配
- 外围设备驱动调度
- 虚拟设备及其实现

其中，前四项是设备管理的基本功能，最后一项是为了进一步提高系统效率而设置的，许多操作系统都提供了此项功能。每一种功能对不同的系统、不同的外围设备配置也有强有弱。

5.1 I/O 硬件原理

不同人员对于 I/O 硬件有着不同的理解。在电气工程师看来，I/O 硬件就是一堆芯片、电线、电源、马达和其他设备的集合体；而程序员则主要注意它为软件提供的接口，即硬件能够接受的命令、它能够完成的功能，以及能报告的各种错误等。作为操作系统的设计师，立足点主要是针对如何利用 I/O 硬件的功能进行程序设计，提供一个方便用户的实用接口，而非研究 I/O 硬件的设计、制造和维护。

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, 把计算机的主存和外围设备的介质之间的信息传送操作称为输入输出操作。随着计算机技术的飞速进步和应用领域扩大, 计算机的输入输出信息量急剧增加, 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。输入输出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。

按照输入输出特性, I/O 设备可以划分为输入型外围设备、输出型外围设备和存储型外围设备三类。按照输入输出信息交换的单位, I/O 设备则可以划分为字符设备和块设备。输入型外围设备和输出型外围设备一般为字符设备, 它与主存进行信息交换的单位是字节, 即一次交换 1 个或多个字节。所谓块是存储介质上连续信息所组成的一个区域, 块设备每次与主存交换一个或几个块信息, 存储型外围设备一般为块设备。

存储型外围设备又可以划分为顺序存取存储设备和直接存取存储设备。顺序存取存储设备严格依赖信息的物理位置进行定位和读写, 如磁带。直接存取存储设备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的位置, 如磁盘。不同设备的物理特性存在很大差异, 其主要差别在于:

- 数据传输率: 从每秒几十个字符(键盘输入)到每秒几个 KB(磁盘), 相差几万倍;
- 数据表示方式: 不同设备采用不同字符表和奇偶校验码;
- 传输单位: 慢速设备以字符为单位, 快速设备以块为单位, 可相差几千倍;
- 出错条件: 错误的性质、形式、后果、报错方法、应对措施等每类设备都不一样。

这些差异使得不论从操作系统还是从用户角度, 都难以获得一个规范一致的 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具有重要的地位, 为了有效地实现物理 I/O 操作, 必须通过硬、软件技术, 对 CPU 和 I/O 设备的职能进行合理分工, 以调解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, 以及和 CPU 之间联系方式的不同, 可把 I/O 设备的控制方式分为四类, 它们的主要差别在于中央处理器和外围设备并行工作的方式不同, 并行工作的程度不同。中央处理器和外围设备并行工作有重要意义, 它能大幅度提高计算机效率和系统资源的利用率。

1. 询问方式

询问方式又称程序直接控制方式。在这种方式下, I/O 指令或询问指令测试一台设备

的忙闲标志位,决定主存储器和外围设备是否交换一个字符或一个字。下面来看一下数据输入的过程。如图 5-1 所示,假如 CPU 上运行的现行程序需要从 I/O 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的起始地址,然后,向 I/O 设备发读指令或查询标志指令,I/O 设备便把状态返回给 CPU。如果 I/O 忙或未就绪,则重复上述测试过程,继续进行查询;如果 I/O 设备就绪,数据传送便开始,CPU 从 I/O 接口读一个字,再向主存写一个字。如果传送还未结束,再次向设备发出读指令,直到全部数据传输完成再返回现行程序执行。为了正确完成这种查询,通常要使用三条指令:(1) 查询指令,用来查询设备是否就绪;(2) 传送指令,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。需要注意,这种方式的数据传输需要用到 CPU 的寄存器,由于传送的往往是一批数据,需要设置交换数据的计数值,以及数据在主存缓冲区的首址。数据输出的过程类似,不再重复讨论。

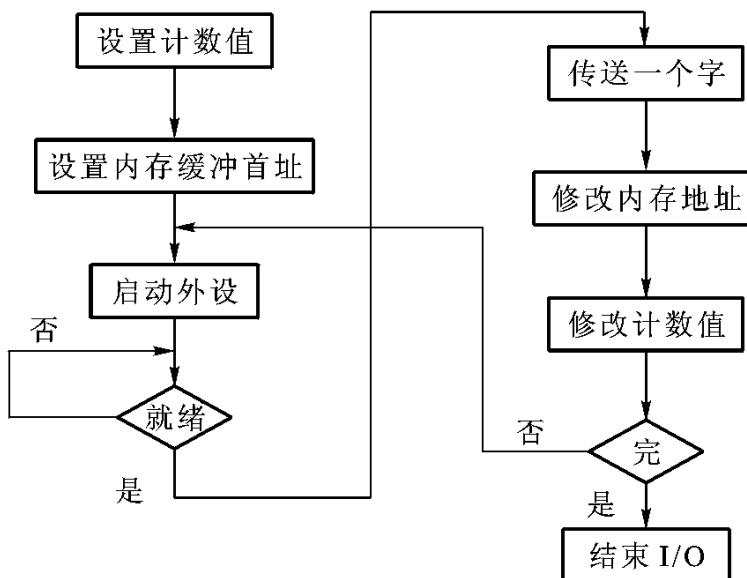


图 5-1 程序查询 I/O

由上述过程可见,一旦 CPU 启动 I/O 设备,便不断查询 I/O 的准备情况,终止了原程序的执行。CPU 在反复查询过程中,浪费了宝贵的 CPU 时间;另一方面,I/O 准备就绪后,CPU 参与数据的传送工作,此时 CPU 也不能执行原程序。可见 CPU 和 I/O 设备串行工作,使主机不能充分发挥效率,外围设备也不能得到合理使用,整个系统的效率很低。

2. 中断方式

中断机构引入后,外围设备有了反映其状态的能力,仅当操作正常或异常结束时才中断中央处理机。这种方式实现了一定程度的并行操作,叫做程序中断方式。仍用上述例子来说明,假如 CPU 在启动 I/O 设备后,不必查询 I/O 设备是否就绪,而是继续执行现行程序,对

设备是否就绪不加过问。直到在启动指令之后的某条指令(如第 K 条)执行完毕,CPU 响应了 I/O 中断请求,才中断现行程序转至 I/O 中断处理程序执行。在中断处理程序中,CPU 全程参与数据传输操作,它从 I/O 接口读一个字(字节)并写入主存,如果 I/O 设备上的数据尚未传送完成,转向现行程序再次启动 I/O 设备,于是命令 I/O 设备再次做准备并重复上述过程;否则,中断处理程序结束后,继续从 K+1 条指令执行。图 5-2 为程序中断方式工作流程。

但是,由于输入输出操作直接由中央处理器控制,每传送一个字符或一个字,都要发生一次中断,因而,仍然消耗大量中央处理器时间。例如,输入机每秒传送 1000 个字符,若每次中断处理平均花 $100 \mu s$,为了传输 1000 个字符,要发生 1000 次中断,所以,每秒内中断处理要花去约 100 ms。但是程序中断方式 I/O,由于不必忙式查询 I/O 准备情况,CPU 和 I/O 设备可实现部分并行,与程序查询的串行工作方式相比,使 CPU 资源得到较充分地利用。

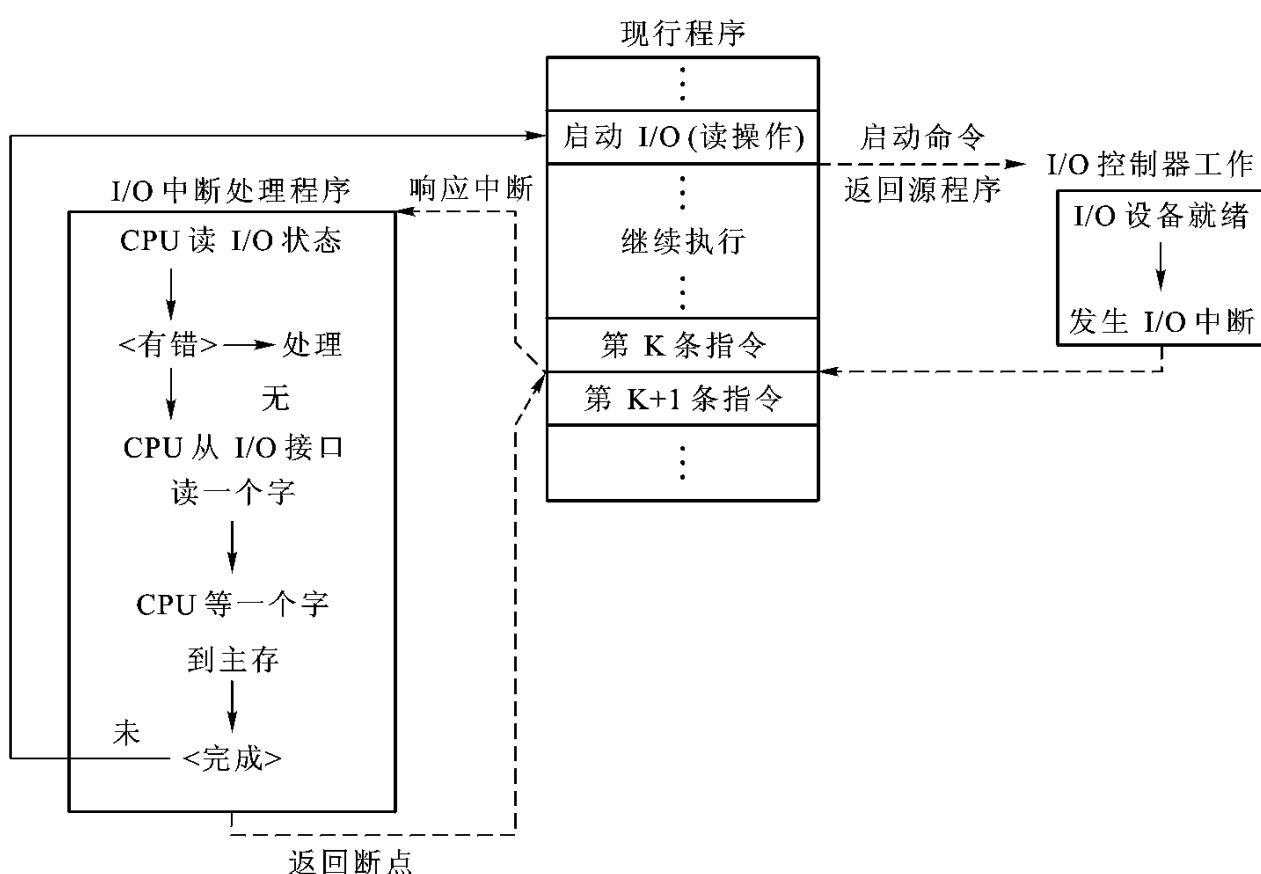


图 5-2 程序中断方式 I/O

3. DMA 方式

虽然程序中断方式消除了程序查询方式的忙式测试,提高了 CPU 资源的利用率,但是在响应中断请求后,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 I/O 设备能直接与主存交换数据而不占用 CPU,那么,CPU 资源的利用率还可提高,这就出现了

直接主存存取 DMA (Direct Memory Access) 方式。

在 DMA 方式中, 主存和 I/O 设备之间有一条数据通路, 在主存和 I/O 设备之间成块地传送数据过程中, 不需要 CPU 干预, 实际操作由 DMA 直接执行完成。图 5-3 为 DMA 方式 I/O 流程。为了实现这种操作, DMA 至少需要以下逻辑部件:

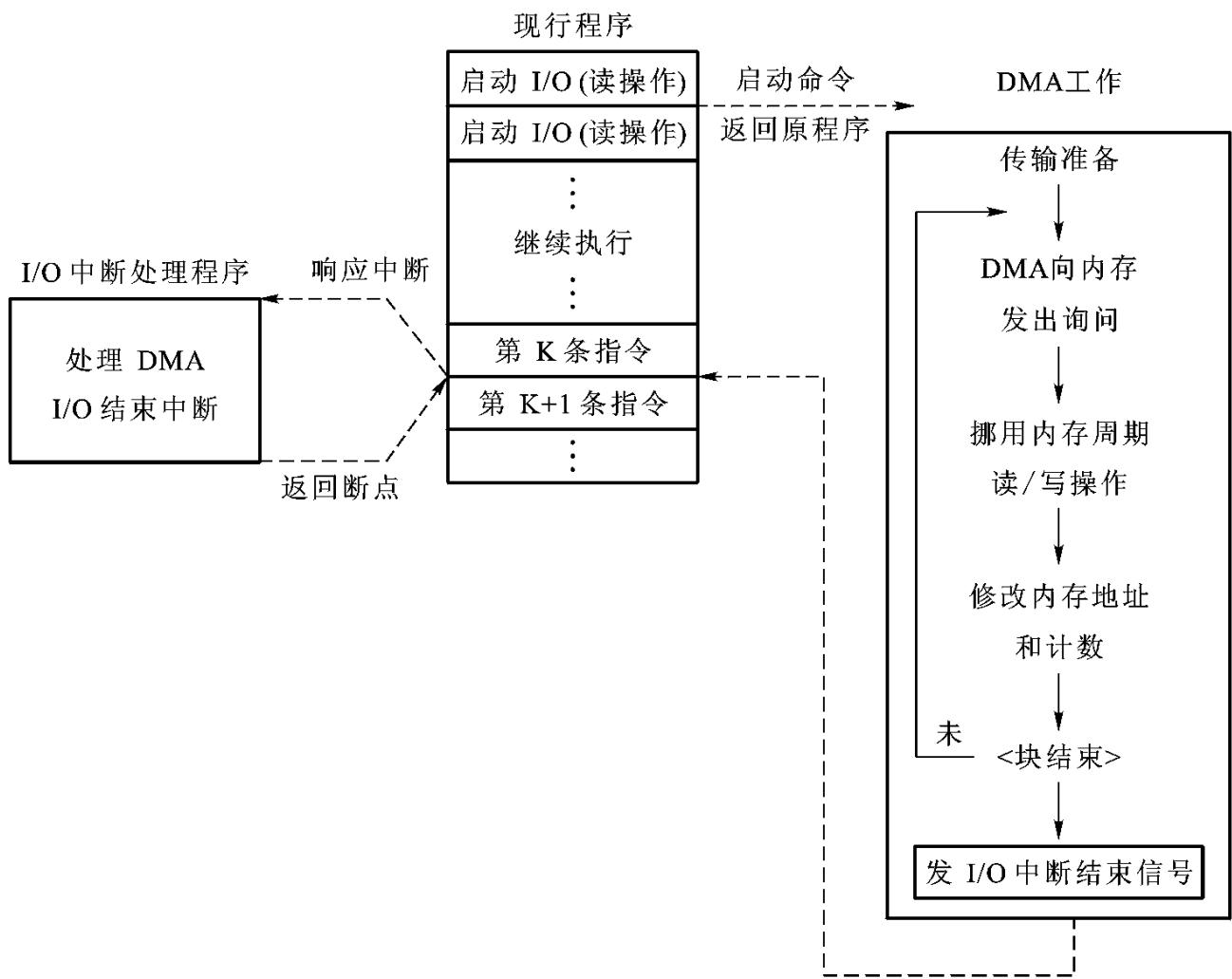


图 5-3 DMA 方式 I/O

- 主存地址寄存器。存放主存中需要交换数据的地址。DMA 传送前, 由程序送入首地址, 在 DMA 传送中, 每交换一次数据, 把地址寄存器内容加 1。
- 字计数器。记录传送数据的总字数, 每传送一个字, 字计数器减 1。
- 数据缓冲寄存器或数据缓冲区。暂存每次传送的数据。DMA 与主存间采用字传送, DMA 与设备间可能是字位或字节传送。所以, DMA 中还可能包括有数据移位寄存器、字节计数器等硬件逻辑。可能有人会提出疑问:为什么控制器从设备读到数据后不立即将其送入主存, 而是需要一个内部缓冲区呢? 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, 不论控制器是否做好接受这些比特的准备。若此时控制器要将数据直接拷贝

到主存中，则它必须在每个字传送完毕后获得对系统总线的控制权。如果由于其他设备也在争用总线的话，则有可能暂时等待。当上一个字还未送入主存前而另一个字已到达时，控制器只能另找一个地方暂存。如果总线非常忙，则控制器可能需要大量的信息暂存，而且要做大量的管理工作。从另一方面来看，如果采用内部缓冲区，则在 DMA 操作启动前不需要使用总线，这样控制器的设计就比较简单，因为，从 DMA 到主存的传输对时间要求并不严格。

- 设备地址寄存器。存放 I/O 设备信息，如磁盘的柱面号、磁道号、块号。
- 中断机制和控制逻辑。用于向 CPU 提出 I/O 中断请求和保存 CPU 发来的 I/O 命令及管理 DMA 的传送过程。

DMA 不仅设有中断机构，而且，还增加了 DMA 传输控制机构。若出现 DMA 与 CPU 同时经总线访问主存，CPU 总把总线占有权让给 DMA，DMA 的这种占有称“周期窃用”，窃取的时间一般为一个存取周期，让设备和主存之间交换数据，而且在 DMA 周期窃取期间，非但不要 CPU 干预，而且 CPU 尚能做运算操作。这样可减轻 CPU 的负担，每次传送数据时，不必进入中断系统，进一步提高了 CPU 的资源利用率。

并非所有的计算机都使用 DMA，反对意见认为 CPU 比 DMA 控制器快的多，当 I/O 设备的速度不构成瓶颈时，CPU 完全可以更快的完成这项工作。特别对于个人计算机来说，如果 CPU 无事可做，而又被迫等待慢速的 DMA 控制器，是完全没有意义的。同时，省去 DMA 控制器还可以节省一些成本。

目前，在小型、微型机中的快速设备均采用这种方案。由于 DMA 方式线路简单，价格低廉，但功能较差，不能满足复杂的 I/O 要求，因而，在中大型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，又减少了 CPU 对 I/O 的干预，已经从字（字节）为单位的干预减少到以数据块为单位的干预。而且，每次 CPU 干预时，并不要做数据拷贝，仅仅需要发一条启动 I/O 指令，以及完成 I/O 结束中断处理。但是，每发出一次 I/O 指令，只能读写一个数据块，如果用户希望一次读写多个离散的数据块，并能把它们传送到不同的主存区域，或相反时，则需要由 CPU 分别发出多条启动 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，它又进一步减少了 CPU 对 I/O 操作的干预，减少对多个不连续的数据块，而不是仅仅一个数据块，及有关管理和控制的干预。同时，为了获得中央处理器和外围设备之间更高的并行工作能力，也让种类繁多，物理特性各异的外围设备能以标准的接口连接到系统中，计算机系统引入了自成独立体系的通道结构。通道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，是计算机技术的一个重要进步。

通道又称输入输出处理器。它能完成主存储器和外围设备之间的信息传送，与中央处理器并行地执行操作。采用通道技术主要解决了输入输出操作的独立性和各部件工作的并行性。由通道管理和控制输入输出操作，大大减少了外围设备和中央处理器的逻辑联系。

从而,把中央处理器从琐碎的输入输出操作中解放出来。此外,外围设备和中央处理器能实现并行操作;通道和通道之间能实现并行操作;各通道上的外围设备也能实现并行操作,以达到提高整个系统效率这一根本目的。

具有通道装置的计算机的主机、通道、控制器和设备之间采用四级连接,实施三级控制。通常,一个中央处理器可以连接若干通道,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。中央处理器执行输入输出指令对通道实施控制,通道执行通道命令(CCW)对控制器实施控制,控制器发出动作序列对设备实施控制,设备执行相应的输入输出操作。

采用输入输出通道设计后,输入输出操作过程如下:中央处理机在执行主程序时遇到输入输出请求,则它启动指定通道上选址的外围设备,一旦启动成功,通道开始控制外围设备进行操作。这时中央处理器就可执行其他任务并与通道并行工作,直到输入输出操作完成。通道发出操作结束中断时,中央处理器才停止当前工作,转向处理输入输出操作结束事件。

按照信息交换方式和加接设备种类不同,通道可分为三种类型:

- 字节多路通道。它是为连接大量慢速外围设备,如软盘输入输出机、纸带输入输出机、卡片输入输出机、控制台打字机等设置的。以字节为单位交叉地工作,当为一台设备传送一个字节后,立即转去为另一台设备传送一个字节。在 IBM 370 系统中,这样的通道可接 256 台设备。

- 选择通道。它用于连接磁带和磁盘快速设备。以成组方式工作,每次传送一批数据,故传送速度很高,但在这段时间只能为一台设备服务。每当一个输入输出操作请求完成后,再选择与通道相连接的另一设备。

- 数组多路通道。对于磁盘这样的外围设备,虽然传输信息很快,但是移臂定位时间很长。如果接在字节多路通道上,那么,通道很难承受这样高的传输率;如果接在选择通道上,那么,磁盘臂移动所花费的较长时间内,通道只能空等。数组多路通道可以解决这个矛盾,它先为一台设备执行一条通道命令,然后自动转换,为另一台设备执行一条通道命令。对于连接在数组多路通道上的若干台磁盘机,可以启动它们同时进行移臂,查找欲访问的柱面,然后,按次序交叉传输一批批信息,这样就避免了移臂操作过长地占用通道。由于它在任一时刻只能为一台设备作数据传送服务,这类似于选择通道;但它不等整个通道程序执行结束就能执行另一设备的通道程序命令,这类似于字节多路通道。数组多路通道的实质是:对通道程序采用多道程序设计技术的硬件实现。

5.1.3 设备控制器

I/O 设备通常包括一个机械部件和一个电子部件。为了达到设计的模块性和通用性,一般将其分开。电子部件称为设备控制器或适配器,在个人计算机中,它常常是一块可以插

入主板扩充槽的印刷电路板;机械部件则是设备本身。

控制器卡上一般都有一个接线器,它可以把与设备相连的电缆线接进来。许多控制器可以控制 2 个、4 个甚至 8 个相同设备。控制器和设备之间的接口越来越多的采用国际标准,如 ANSI、IEEE、ISO 或者事实上的工业标准。依据这些标准,各个计算机厂家都可以制造与标准接口相匹配的控制器和设备。例如 IDE(集成设备电子器件)接口、SCSI(小型计算机系统接口)接口的硬盘。

区分控制器和设备本身是因为操作系统基本上是与控制器打交道,而非设备本身。大多数微型计算机的 CPU 和控制器之间的通信采用单总线模型,CPU 直接控制设备控制器进行输入输出;而主机则采用多总线结构和通道方式,以提高 CPU 与输入输出的并行程度。

控制器与设备之间的接口是一种很低层次的接口。例如,一个磁盘,可以被格式化成为一个每道 16 个 512 字节的扇区,实际从磁盘读出来的是一个比特流,以一个前缀开始,随后是一个扇区的 4096 比特,最后,是一个纠错码 ECC。其中前缀是磁盘格式化时写进的,包括柱面数、扇区数、扇区大小等,以及同步信息。控制器的任务是把这个串行的比特流转换成字节块并在必要时进行纠错,通常该字节块是在控制器中的一个缓冲区中逐个比特汇集而成,在检查和校验后,该块数据将被拷贝到主存中。

CRT 控制器也是一个比特串行设备,它从主存中读取欲显示字符的字节流,产生用来调制 CRT 射线的信号,然后,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以及当整个屏幕被扫描后的垂直方向的折返信号。

不难看出,如果没有控制器,这些复杂的操作必须由操作系统程序员自己编写程序来解决;而引入了控制器后,操作系统只需通过传递几个简单的参数就可以对控制器进行操作和初始化。从而,大大简化了操作系统的功能设计,特别是有利于计算机系统和操作系统对各类控制器和设备的兼容性。

每个控制器都有一些用来与 CPU 通信的寄存器,在某些计算机上,这些寄存器占用内存地址的一部分,称为内存映像 I/O;另一些计算机则采用 I/O 专用地址,每个寄存器占用其中的一部分。设备的 I/O 地址分配由控制器上的总线解码逻辑完成。除 I/O 端口外,许多控制器还通过中断通知 CPU 它们已经做好准备,寄存器可以读写。以 IBM 奔腾系列为例,它向 I/O 设备提供 15 条可用中断。

有些控制器做在计算机主板上,如 IBM PC 机的键盘控制器。对于那些单独插在主板插槽上的控制器,有时上面设有一些可以用来设置 IRQ 号的开关和跳线,以便避免 IRQ 冲突。中断控制器芯片将每个 IRQ 输入并映像到一个中断向量,通过这个中断向量就可以找到相应的中断服务程序。下表给出了 PC 机部分控制器的 I/O 地址、硬件中断和中断向量号。

操作系统通过向控制器寄存器写命令字来执行 I/O 功能。例如 PC 机的软盘控制器可以接收 15 条命令,包括读、写、格式化、重新校准等。许多命令字带有参数,这些参数也要同

时装入控制器寄存器。一旦某个控制器接收到一条命令后, CPU 可以转向进行其他工作, 而让该设备控制器自行完成具体的 I/O 操作。当命令执行完毕后, 控制器发出一个中断信号, 以便使操作系统重新获得 CPU 的控制权并检查执行结果, 此时, CPU 仍旧是从控制器寄存器中读取若干字节信息来获得执行结果和设备的状态信息。

表 5-1 PC 机部分控制器的 I/O 地址、硬件中断号和中断向量号

| I/O 控制器 | I/O 地址 | 硬件中断号 | 中断向量号 |
|---------|---------|-------|-------|
| 时钟 | 040—043 | 0 | 8 |
| 键盘 | 060—063 | 1 | 9 |
| 硬盘 | 1F0—1F7 | 14 | 118 |
| 软盘 | 3F0—3F7 | 6 | 14 |
| LPT1 | 378—37F | 7 | 15 |
| COM1 | 3F8—3FF | 4 | 12 |
| COM2 | 2F8—2FF | 3 | 11 |

下面来小结一下设备控制器的功能和结构。设备控制器是 CPU 和设备之间的一个接口, 它接收从 CPU 发来的命令, 控制 I/O 设备操作, 实现主存和设备之间的数据传输操作, 从而, 使 CPU 从繁杂的设备控制操作中解放出来。设备控制器是一个可编址设备, 当它连接多台设备时, 则应具有多个设备地址。设备控制器的主要功能为:①接收和识别 CPU 或通道发来的命令, 例如, 磁盘控制器能接收读、写、查找、搜索等各种命令;②实现数据交换, 包括设备和控制器之间的数据传输;通过数据总线或通道, 控制器和主存之间的数据传输;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:命令寄存器及译码器, 数据寄存器, 状态寄存器, 地址译码器, 以及用于对设备操作进行控制的 I/O 逻辑。

5.2 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。高效率是不言而喻的, 在改善 I/O 设备的效率中, 最应关注的是磁盘 I/O 的效率。通用性意味着用统一标准的方法来管理所有设备。为了达到这一目标, 通常, 把软件组织成一种层次结构, 低层软件用来屏蔽硬件的具体

细节,高层软件则主要向用户提供一个简洁、规范的界面。I/O 软件设计主要考虑以下 4 个问题:

- 设备无关性。程序员写出的软件在访问不同的外围设备时应该尽可能地与设备的具体类型无关,如访问文件是不必考虑它是存储在硬盘、软盘还是 CD - ROM 上。
- 出错处理。总的来说,错误应该在尽可能靠近硬件的地方处理,在低层软件能够解决的错误就不让高层软件感知,只有低层软件解决不了的错误才通知高层软件解决。
- 同步(阻塞)——异步(中断驱动)传输。多数物理 I/O 是异步传输,即 CPU 在启动传输操作后便转向进行其他工作,直到中断到达。I/O 操作可以采用阻塞语义,发出一条 READ 命令后,程序将自动被挂起,直到数据被送到主存缓冲区。
- 独占性外围设备和共享性外围设备。某些设备可以同时为几个用户服务,如磁盘;另一些设备在某一段时间只能供一个用户使用,如键盘。独占性外围设备和共享性外围设备带来了许多问题,操作系统必须能够同时加以解决。

为了合理、高效地解决以上问题,操作系统通常把 I/O 软件组织成以下四个层次。

- I/O 中断处理程序。
- I/O 设备驱动程序。
- 与设备无关的操作系统 I/O 软件。
- 用户层 I/O 软件。

5.2.2 I/O 中断处理程序

中断是应该尽量加以屏蔽的概念,应该放在操作系统的底层进行处理,系统的其余部分尽可能少地与之发生联系。当一个进程请求 I/O 操作时,该进程将被挂起,直到 I/O 操作结束并发生中断。当中断发生时,中断处理程序执行相应的处理,并解除相应进程的阻塞状态。中断层的主要工作有:处理中断信号和修改进程状态等。

输入输出中断的类型和功能如下:

- 通知用户程序输入输出操作沿链推进的程度。此类中断有程序进中断。
- 通知用户程序输入输出操作正常结束。当输入输出控制器或设备发现通道结束、控制结束、设备结束等信号时,就向通道发出一个报告输入输出操作正常结束的中断。
- 通知用户程序发现的输入输出操作异常,包括设备出错、接口出错、I/O 程序出错、设备特殊、设备忙等,以及提前中止操作的原因。
- 通知程序外围设备上有重要的异步信号。此类中断有注意、设备报到、设备结束等。

当输入输出中断被响应后,中断装置交换程序状态字引出输入输出中断处理程序。输入输出中断处理程序从保存的旧 PSW 中得到产生中断的通道号和设备号,并分析通道状态字,弄清产生中断的输入输出中断事件,处理的一般原则如下:

1) 如果是操作正常结束,那么,系统要查看是否有等待该设备或通道者,若有则释放。例如,接在数组多路通道上的某台行式打印机,当通过通道完成主存到行打印机缓冲器之间的一行信息传送后,虽然行式打印机并未完成打印,而可能发出通道结束中断。此时通道已空闲可以去控制通道上的另一台设备。当行式打印机打印出一行信息,完成了一次输出的所有任务后,它还要发出设备结束中断请求并报告给系统。操作系统分析通道状态字的设备状态字节可知道是“通道结束”还是“设备结束”,从而,释放等待通道者或释放等待设备者。

2) 如果是操作中发生故障或某种特殊事件而产生的中断,那么,操作系统要进一步查明原因,采取相应措施。操作中发生的故障及其处理的方法可能有以下几种:

- 设备本身的故障。例如,读写操作中校验装置发现的错误。操作系统可以通过设备状态字节的“设备错误”位是否为1来发现这类故障。系统处理这种故障时,先向相应设备发命令索取断定状态字节,然后,分析断定状态字节就可以知道故障的确切原因。如果该外围设备的控制器没有复执功能,那么,对于某些故障,系统可组织软复执。如读磁带上的信息,当校验装置发现错误时,操作系统可组织回退,再读若干遍。对于不能复执的故障或复执多次仍不能克服的故障,系统将向操作员报告,请求人工干预。

- 通道的故障。对于这种故障也可进行复执。如果硬件或软复执多次仍出错,那么,系统应将错误情况报告给操作员。

- 通道程序错。由通道识别的各种通道程序错误,例如,通道命令非双字边界;通道命令地址无效;通道命令的命令码无效;CAW格式错;连用两条通道转移命令等,均由系统报告给操作员。

- 启动命令的错误。例如,启动外围设备的命令要求从输入机上读入1000个字符,然而,读了500个字符就遇到“停码”,输入机便停止了。操作系统通过通道状态字节的长度错误位是否为1可判断这类错误,再把处理转交给用户,转向用户程序的中断续元,由用户自己处理。

如果设备在操作中发生了某些特殊事件,那么,在设备操作结束发生中断时,也要将这个情况向系统报告。操作系统从设备状态字节中的设备特殊位为1,可以判知设备在操作中发生了某个特殊事件。对于磁带机,这意味着在写入一块信息遇到了带末点或读出信息时遇到了带标。在写操作的情况下,系统知道磁带即将用完,如果文件还未写完,应立即组织并写入卷尾标,然后,通知操作员换卷以便将文件的剩余部分写在后继卷上。在读操作的情况下,系统判知这个文件已经读完或这个文件在此卷上的部分已经读完,进行文件结束的处理;若只读了一部分,则带标后面是卷尾标,系统将通知操作员换卷,以便继续读入文件。对于行式打印机,这意味着纸将用完,系统可暂停输出,通知操作员装纸,接着继续输出。

3) 如果是人为要求而产生的中断,那么,系统将响应并启动外围设备。例如,要求从控制台打字机输入时,操作员先按“询问键”,随之产生中断请求。如果设备状态字节的“注意”位被置1,系统将响应并启动外围设备。

位”为 1 操作系统就知道控制台打字机请求输入。此时, 系统启动控制台打字机并开放键盘, 接着操作员便可打入信息。

4) 如果是外围设备上来的“设备结束”等异步信号, 表示有外围设备接入可供使用或断开暂停使用。操作系统应修改系统表格中相应设备的状态。

5.2.3 设备驱动程序

设备驱动程序中包括了所有与设备相关的代码, 它的工作是: 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。

每个设备驱动程序只处理一种设备, 或者一类紧密相关的设备。例如, 若系统所支持的不同品牌的所有终端只有很细微的差别, 则较好的办法是为所有这些终端提供一个终端驱动程序。另一方面, 一个机械式的硬拷贝终端和一个带鼠标的智能化图形终端差别太大, 于是只能使用不同的驱动程序。

前面已介绍了设备控制器的功能, 知道每个控制器都有一个或多个寄存器来接收命令。设备驱动程序发出这些命令并对其进行检查, 操作系统中只有硬盘驱动程序才知道磁盘控制器有多少个寄存器, 以及它们的用途。驱动程序知道使磁盘正确操作所需要的全部参数, 包括扇区、磁道、柱面、磁头、磁头臂的移动、交叉系数、步进电机、磁头定位时间等等。

笼统地说, 设备驱动程序的功能是从与设备无关的软件中接收抽象的请求, 并执行之。一条典型的请求是读第 n 块。如果请求到来时驱动程序空闲, 则它立即执行该请求。但如果它正在处理另一条请求, 则它将该请求挂在一个等待队列中。

执行一条 I/O 请求的第一步, 是将它转换为更具体的形式。对磁盘驱动程序来说, 它包含: 计算出所请求块的物理地址、检查驱动器电机是否在运转、检测磁头臂是否定位在正确的柱面等等。简而言之, 它必须确定需要哪些控制器命令以及命令的执行次序。

一旦决定应向控制器发送什么命令, 驱动程序将向控制器的设备寄存器中写入这些命令。某些控制器一次只能处理一条命令, 另一些则可以接收一串命令并自动进行处理。这些控制命令发出后有两种可能。在许多情况下, 驱动程序需等待控制器完成一些操作, 所以, 驱动程序阻塞, 直到中断信号到达才解除阻塞。另一种情况是操作没有任何延迟, 所以驱动程序无需阻塞。后一种情况的例子如: 在有些终端上滚动屏幕只需往控制器寄存器中写入几个字节, 无需任何机械操作, 整个操作可在几微秒内完成。

对前一种情况, 被阻塞的驱动程序须由中断唤醒, 而后一种情况下它根本无需睡眠。无论哪种情况, 都要进行错误检查。如果一切正常, 则驱动程序将数据传送给上层的设备无关软件。最后, 它将向它的调用者返回一些关于错误报告的状态信息。如果请求队列中有别的请求则它选中一个进行处理, 若没有则它阻塞, 等待下一个请求。

5.2.4 与硬件无关的操作系统 I/O 软件

尽管某些 I/O 软件是设备相关的,但大部分独立于设备。设备无关软件和设备驱动程序之间的精确界限在各个系统都不尽相同。对于一些以设备无关方式完成的功能,在实际中由于考虑到执行效率等因素,也可以考虑由驱动程序完成。

下面罗列了一般都是由设备无关软件完成的功能:

- 对设备驱动程序的统一接口
- 设备命名
- 设备保护
- 提供独立于设备的块大小
- 缓冲区管理
- 块设备的存储分配
- 独占性外围设备的分配和释放
- 错误报告

设备无关软件的基本功能是执行适用于所有设备的常用 I/O 功能,并向用户层软件提供一个一致的接口。

(1) I/O 设备的命名方式和映射、设备保护

设备无关软件负责将设备名映射到相应的驱动程序。例如,在 UNIX/Linux 中,一个设备名,如 /dev/tty00 惟一地确定了一个 *i* - 节点,其中包含了主设备号(major device number),通过主设备号就可以找到相应的设备驱动程序。*i* - 节点也包含了次设备号(minor device number),它作为传给驱动程序的参数指定具体的物理设备。类似地,IDE(Intergrated Disk Electronic disk)硬盘为 /dev/hda、/dev/hdb、/dev/hdc 分别为第一和第二块硬盘,hd 为主设备号,而 a 和 b 为次设备号。

与命名相关的是设备保护,检查用户是否有权访问申请的设备。多数个人计算机系统根本就不提供任何保护;多数大型主机系统中,用户进程绝对不允许直接访问 I/O 设备。在 UNIX 中使用一种更为灵活的方法,对应于 I/O 设备的设备文件的保护采用通常的 rwx 权限机制,所以,系统管理员可以为每一台设备设置合理的访问权限。

(2) 文件空间管理和逻辑记录到物理记录的转换

当创建了一个文件并向其输入数据时,该文件必须被分配新的磁盘块。为了完成这种分配工作,操作系统需要为每个磁盘都配置一张记录空闲盘块的表或位图,但分配一个空闲块的算法是独立于设备的,因此,可以在高于驱动程序的层次处理。

不同磁盘的扇区大小可能不同,设备无关软件屏蔽了这一事实并向高层软件提供统一的数据块大小,比如将若干扇区作为一个逻辑块。这样高层软件就只和逻辑块大小都相同

的抽象设备交互, 而不管物理扇区的大小。类似地, 有些字符设备(modem)对字节进行操作, 另一些字符设备(网卡)则使用比字节大一些的单位, 这类差别也可以进行屏蔽。

(3) 缓冲技术

块设备和字符设备都需要缓冲技术。对于块设备, 硬件每次读写均以块为单元, 而用户程序则可以读写任意大小的单元。如果用户进程写半个块, 操作系统将在内部保留这些数据, 直到其余数据到齐后才一次性地将这些数据写到设备上。对字符设备, 用户向系统写数据的速度可能比向设备输出的速度快, 所以, 需要进行缓冲。超前的键盘输入同样也需要缓冲。

(4) 设备状态跟踪, 设备挂起与释放

一些设备, 如 CD-ROM 记录器, 在同一时刻只能由一个进程使用。这要求操作系统检查对该设备的使用请求, 并根据设备的忙闲状况来决定是接受或拒绝此请求。一种简单的处理方法是通过直接用 OPEN 打开相应的设备文件来进行申请。若设备不可用, 则 OPEN 失败。关闭独占设备的同时将释放该设备。

(5) 错误处理

错误处理多数由驱动程序完成, 错误是与设备紧密相关的, 因此, 只有驱动程序知道应如何处理(如重试、忽略、严重错误)。一种典型错误是磁盘块受损导致不能读写。驱动程序在尝试若干次读操作不成功后将放弃, 并向设备无关软件报错, 此后错误处理就与设备无关了。如果在读一个用户文件时出错, 则向调用者报错即可。但如果是在读一些关键系统数据结构时出错, 比如磁盘使用状况位图, 则操作系统只能打印出错信息, 并终止运行。

5.2.5 用户空间的 I/O 软件

1. 库例程

尽管大部分 I/O 软件属于操作系统, 但是有一小部分是与用户程序链接在一起的库例程(库函数)。系统调用, 包括 I/O 系统调用通常通过库例程间接提供给用户。如下面的 C 语言语句:

```
count = write(fd, buffer, nbytes);
```

其中, 所调用的库函数 write 将与用户程序链接在一起, 并包含在运行时的二进制程序代码中, 这一类库例程显然也是 I/O 系统的一部分。它的主要工作是提供参数给相应的系统调用并调用之。

但也有一些库例程, 它们确实做非常实际的工作, 例如, 格式化输入输出就是用库例程实现的。C 语言中的一个例子是 printf 函数, 它的输入为一个格式字符串, 其中可能带有一些变量, 它随后调用 write, 输出格式化后的一个 ASCII 码串。与此类似的 scanf, 它采用与 printf 相同的语法规则来读取输入。标准 I/O 库包含相当多的涉及 I/O 的库例程, 它们都作

为用户程序的一部分运行。

2. spooling 软件

并非所有的用户层 I/O 软件都由库例程构成, spooling 软件就是在核心外运行的用户级 I/O 软件。在本章后面作进一步介绍。

图 5-4 总结了 I/O 系统, 标示出了每一层软件及其功能。从底层开始分别是硬件、中断处理程序、设备驱动程序、设备无关软件, 最上面是用户进程。

该图中的箭头表示控制流, 当用户程序试图从文件中读一数据块时, 需通过操作系统来执行此操作。设备无关软件首先在数据块缓冲区中查找此块, 若未找到, 则它调用设备驱动程序向硬件发出相应的请求, 用户进程随即阻塞直到数据块被读出。当磁盘操作结束时, 硬件发出一个中断, 它将激活中断处理程序。中断处理程序则从设备获取返回状态值并唤醒睡眠的进程来结束此次 I/O 请求, 并使用户进程继续执行。

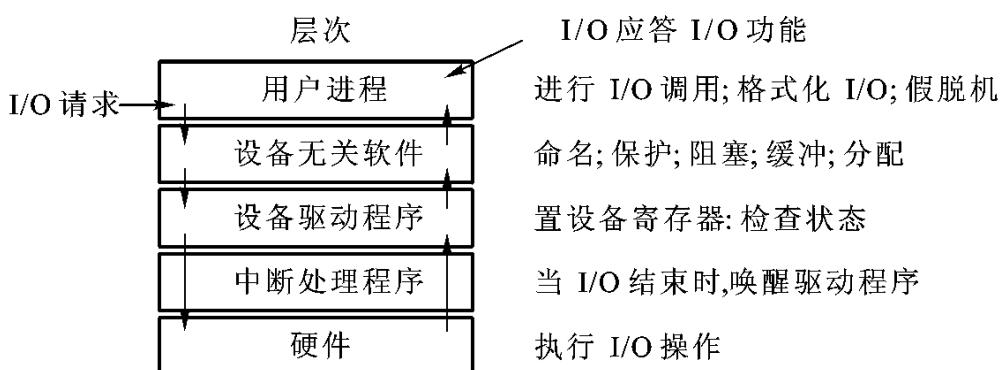


图 5-4 I/O 软件的层次及其功能

5.3 具有通道的 I/O 系统管理

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, 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

通道又称为 I/O 处理机, 具有自己的指令系统, 常常把 I/O 处理机的指令称通道命令。通道命令 CCW(Channel Command Word) 是通道从主存取出并控制 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通道程序, 一条通道命令往往只能实现一种功能。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 系统的通道命令双字长,格式如图 5-5 所示:



图 5-5 IBM370 通道

通道命令字为双字长,各字段的含义如下:

- 命令码,规定了外围设备所执行的操作。通道命令码分三类:数据传输类(读、反读、写、取状态),通道转移类(转移),设备控制类(随设备类不同执行不同控制)。
- 数据主存地址,对数据传输类命令,规定了本条通道命令访问的主存数据区起始(或末尾)地址,而“传送字节个数”指出了数据区的大小。对通道转移类命令,用来规定转移地址。
- 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,32 位至 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 和 33 位均为 0,称无链,表示本条通道命令是通道程序的最后一条;为 01 时,称命令链,表示本命令的操作已是最后一条,后面还有通道命令但为其他命令;32 位为 1 时,称数据链,表示下一条通道命令将沿用本条的命令码但由下一条通道命令指明新的主存区域。34 位为 1 时,该条通道命令执行中,禁止发长度错。35 位为 1 时,能使读型操作实现假读功能。36 位为 1 时,执行到该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。
- 传送字节个数,对数据传输类命令,规定了本次交换的字节个数;对通道转移类命令,规定填一个非 0 数。

2. 通道程序

启动外围设备按指定要求工作,首先要编写出实现指定功能的通道程序。编制通道程序并不困难,关键在于:记住通道命令的格式,不同外围设备有不同的命令码,不能混用。下面是由汇编格式写的一个通道程序的例子。

```

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,

```

inarea DS CL240

该通道程序把磁带上三个不连续的信息块读入主存的连续区域, 其中, * 表示不用主存地址, X 50' 表示使用了“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 I/O 时, 要使用两个主存固定存储单元: 通道地址字 CAW (Channel Address Word) 和通道状态字 CSW (Channel Status Word)。

编好的通道程序放在主存中, 为了使通道能取到通道命令去执行, 在主存的一个固定单元中存放当前启动的外围设备要求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通道地址字。

通道状态字是通道向操作系统报告情况的汇集。通道利用通道状态字可以提供通道和外围设备执行 I/O 操作的情况。IBM 系统中的通道状态字也采用双字表示。其中各字段的含义为:

- 通道命令地址。一般指向最后一条执行的通道命令地址加 8。
- 设备状态。是由控制器或设备产生、记录和供给的信息, 包括注意、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。
- 通道状态。由通道发现、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。
- 剩余字节个数。最后一条通道命令执行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 (Start I/O, SIO), 查询 I/O (Test I/O, TIO), 查询通道 (Test Channel, TCH), 停止 I/O (Halt I/O, HIO) 和停止设备 (Halt Device, HDV), 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, 而根据系统的约定可把通道程序的首地址存放在主存中的通道地址字单元。CPU 执行 I/O 时只是简单地将 I/O 指令发给通道就行了, SIO 指令发出后, 如果条件码为 0, 表示设备已被成功启动, 通道从 CAW 取通道程序首地址开始工作, CPU 可返回去执行计算任务; 如果条件码为 1, 或表示启动成功(对于立即型命令), 或启动不成功, 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, 表示通道或设备忙碌, 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断开, 因而, 启动不成功, 本指令执行结束。

每次执行 I/O 操作, 要为通道编制通道程序, 要为主机编制主机 I/O 程序。CPU 执行驱

动外围设备指令时,同时,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 指令和通道程序对外围设备进行具体的控制。正确执行一次 I/O 操作的步骤可归纳如下:

- 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,决定例外情况处理方法。
- 编写通道程序,完成相应的 I/O 操作。
- 编写主机 I/O 程序,对不同条件码进行不同处理。

下面例子是用 IBM 汇编语言编写的采用双缓冲把磁带上的块记录在行式打印机上输出。

```

START
    BALR    11,0
    USING   *,11
    SSM = X '00'          /* 开中断
    LA     8,READ0
    ST     8,CAW
    SIO    X '0182'        /* 启动磁带机反绕
    BC     7,* - 4         /* 循环直到启动
    TIO    X '0182'
    BC     7,* - 4         /* 测试直到磁带完成反绕

LOOP LA 8,READ1
    ST     8,CAW
    SIO    X '0182'        /* 启动磁带读入缓冲 1
    BC     7,* - 4
    TIO    X '0182'
    BC     7,* - 4         /* 测试直到磁带完成
    LA     8,PRINT1
    ST     8,CAW
    TIO    X '00E'
    BC     7,* - 4         /* 测试直到缓冲 2 打印完
    SIO    X '00E'          /* 启动行式打印机印缓冲 1 的内容

    LA     8,READ2
    ST     8,CAW
    SIO    X '0182'        /* 启动磁带读入缓冲 2
    BC     7,* - 4
    TIO    X '0182'

```

```

BC  7, * - 4          /* 测试直到磁带完成
LA  8, PRINT2
ST  8, CAW
TIO  X '00E'          /* 查询行式打印机
BC  7, * - 4          /* 测试直到缓冲 1 打印完
SIO  X '00E'          /* 启动行式打印机印缓冲 1 的内容
B   LOOP
READ0 CCW   X '07', • , X '20',1
READ1 CCW   X '02', BUFFER1, X '00',512
READ2 CCW   X '02', BUFFER2, X '00',512

```

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 和设备之间是主从关系, 需要相互配合协调才能完成 I/O 操作, 那么, CPU 如何通知通道做什么? 通道又如何被告知 CPU 状态和工作情况呢? 通道方式 I/O 过程可以分成三个阶段:

- 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哪台设备、传输信息的位置、传送个数和信息主存区的地址。然后, 文件系统把存取要求通知设备管理, 设备管理按规定组织好通道程序并将首地址放入 CAW。CPU 向通道发出 SIO, 命令通道工作, 通道根据自身状态形成条件码作为回答, 若通道可用, 则 CPU 传送本次设备地址, I/O 操作开始。这一通信过程发生在操作开始期, CPU 根据条件码便可决定转移方向。

- I/O 操作阶段。启动成功后, 通道从主存固定单元取 CAW, 根据该地址取得第一条通道命令, 通道执行通道程序, 同时将 I/O 地址传送给控制器, 向它发出读、写或控制命令, 控制外围设备进行数据传输。控制器接收通道发来的命令之后, 检查设备状态, 若设备不忙, 则告知通道释放 CPU, 并开始 I/O 操作, 向设备发出一系列动作序列, 设备则执行相应动作。之后, 通道独立执行通道程序中各条 CCW, 直到通道程序执行结束。从通道被启动成功开始, CPU 已被释放可执行其他任务并与通道并行工作, 直到本次 I/O 结束, 通道向 CPU 发出 I/O 操作结束中断, 再次请求 CPU 干预。

- I/O 结束阶段。通道发现通道状态字中出现通道结束、控制器结束、设备结束或其他能产生中断的信号时, 就应向 CPU 申请 I/O 中断。同时, 把产生中断的通道号和设备号, 以及 CSW 存入主存固定单元。中断装置响应中断后, CPU 上的现行程序才被暂停, 调出 I/O 中断处理程序处理 I/O 中断。图 5-6 是通道方式 I/O 的示意。

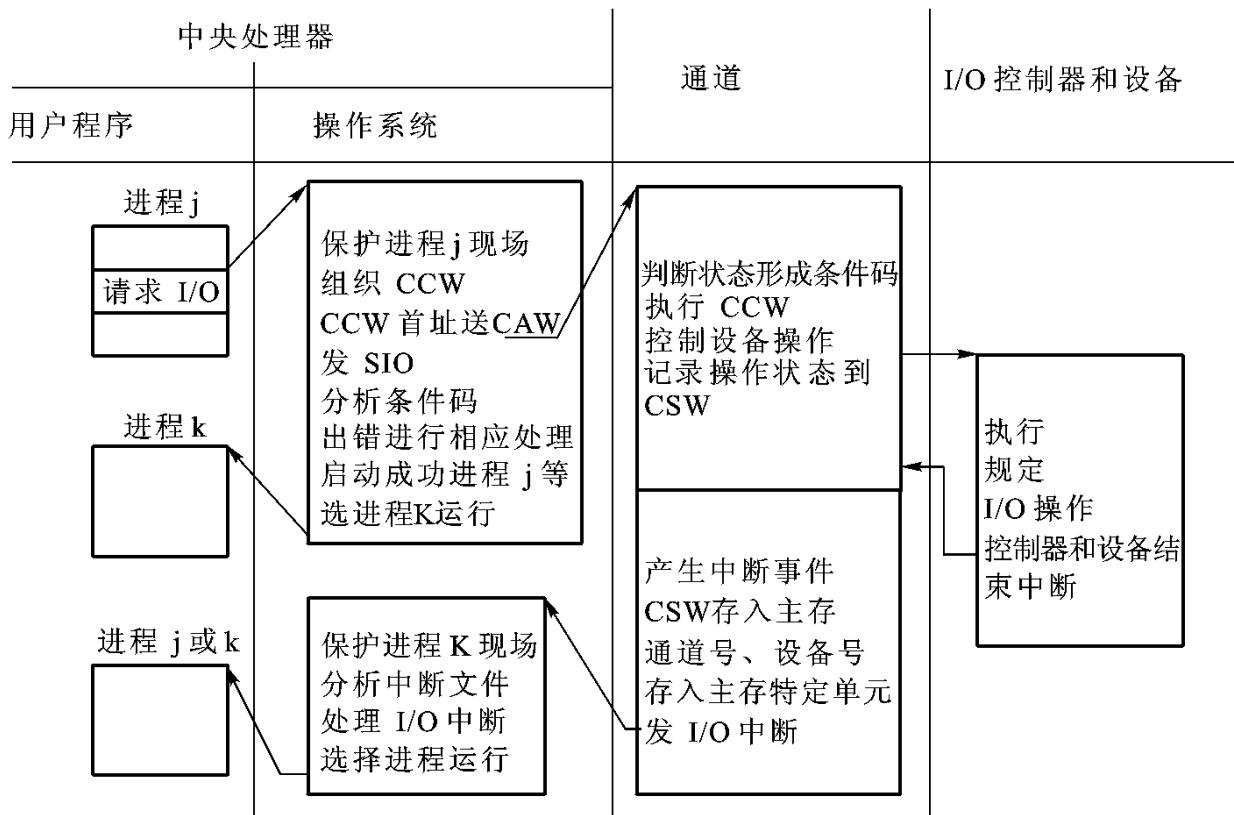


图 5-6 通道方式 I/O

5.4 缓冲技术

为了改善中央处理器与外围设备之间速度不匹配的矛盾,以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 I/O 设备的并行性,减少 I/O 对 CPU 的中断次数和放宽对 CPU 中断响应时间的要求,在操作系统中普遍采用了缓冲技术。缓冲用于平滑两种不同速度部件或设备之间的信息传输,由于硬件实现缓冲成本太高,通常的实现方法是在主存开辟一个存储区称缓冲区,专门用于临时存放 I/O 的数据。

缓冲技术实现的基本思想如下:当一个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,将数据高速送到缓冲区。若为顺序写请求,则不断把数据填到缓冲区,直到它被装满为止。此后,进程可以继续它的计算,同时,系统将缓冲区内容写到 I/O 设备上。当一个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,系统将一个物理记录的内容读到缓冲区中,然后,根据进程要求,把当前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域称为 I/O 缓冲区,在输出数据时,只有在系统还来不及腾

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,仅当缓冲区空而进程又要从中读取数据时,它才被迫等待。其他时间可以进一步提高 CPU 和 I/O 设备的并行性,以及 I/O 设备和 I/O 设备之间的并行性,从而,提高整个系统的效率。

在操作系统管理下,常常辟出许多专用主存区域的缓冲区用来服务于各种设备,支持 I/O 管理功能。常用的缓冲技术有:单缓冲、双缓冲、多缓冲。

5.4.1 单缓冲

单缓冲是操作系统提供的一种简单的缓冲技术。每当一个用户进程发出一个 I/O 请求时,操作系统在主存的系统区中开设一个缓冲区。

对于块设备输入,单缓冲机制如下工作:先从磁盘把一块数据传送到缓冲区,假如所花费的时间为 T ;接着操作系统把缓冲区数据送到用户区,设所花时间为 M ,由于这时缓冲区已空,操作系统可预读紧接的下一块,大多数应用将要使用邻接块,然后,用户进程对这批数据进行计算,共耗时 C 。如果不采用缓冲,数据直接从磁盘到用户区,每批数据处理时间约为 $T + C$,而采用单缓冲,每批数据处理时间约为 $\max(C, T) + M$,通常 M 远小于 C 或 T ,故速度快了很多。对于块设备输出,单缓冲机制工作方式类似,先把数据从用户区拷贝到系统缓冲区,用户进程可以继续请求输出,直到缓冲区填满后,才启动 I/O 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,在输入期间,用户进程被挂起等待一行数据输入完毕;在输出时,用户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没有腾空缓冲区之前,又有第二行数据要输出,用户进程应等待。

如果希望实现输入和输出并行工作,如把输入设备读卡机上的数据输入并加工,再在输出设备打印机上打印出来,必须引入双缓冲技术。

5.4.2 双缓冲

为了加快 I/O 速度,实现输入输出的并行工作和提高设备利用率,需要引入双缓冲工作方式,又称缓冲交换(buffer swapping)。在输入数据时,首先填满缓冲区 1,操作系统可从缓冲区 1 把数据送到用户进程区,用户进程便可对数据进行加工计算;与此同时,输入设备填充缓冲区 2。当缓冲区 1 空出后,输入设备再次向缓冲区 1 输入。操作系统又可以把缓冲区 2 的数据传送到用户进程区,用户进程开始加工缓冲区 2 的数据。两个缓冲区交替使用,使 CPU 和 I/O 设备、设备和设备的并行性进一步提高,仅当两个缓冲区都取空,进程还要提取数据时,它才被迫等待。粗略估计一下,传输和处理一块的时间,如果 $C < T$,输入操作比计算操作慢,这时由于 M 远小于 T ,故在将磁盘上的一块数据传送到一个缓冲区期间(所花时间为 T),计算机已完成了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,这种情况下可以保证块设备连续工作;如果 $C > T$,计算操作比输入操作慢,每当上一块数据计算完毕后,仍需把一个缓冲区中的数据传送到用户区,花费时间为 M ,再对这块数据进行计算,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,这种情况下使得进程没有必要等待 I/O。双缓冲使效率提高了,但复杂性也增加了。

采用双缓冲读卡并打印可以这样进行:第一张卡片读入缓冲区 1,在打印缓冲区 1 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。这样输入和输出处于并行工作状态。

5.4.3 多缓冲

采用双缓冲技术虽然提高了 I/O 设备的并行工作程度,减少了进程调度开销,但在输入设备、输出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,若输入设备的速度高于进程消耗这些数据的速度,则输入设备很快就把两个缓冲区填满;有时由于进程处理输入数据速度高于数据输入的速度,很快又把两个缓冲区抽空,造成进程等待。为改善上述情形,获得较高的并行度,常常采用多缓冲组成的循环缓冲(circular buffer)技术。

操作系统从自由主存区域中分配一组缓冲区,每个缓冲区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,组成了循环缓冲,每个缓冲区的大小可以等于物理记录的大小。多缓冲的缓冲区是系统的公共资源,可供各个进程共享,并由系统统一分配和管理。缓冲区用途分为: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,进行各种操作,必须设计专门的软件,这就是缓冲区自动管理系统。

在 UNIX/linux 系统中,不论是块设备管理,还是字符设备管理,都采用循环缓冲技术,其目的有两个:一是尽力提高 CPU 和 I/O 设备的并行工作程度;二是力争提高文件系统信息读写的速度和效率。UNIX 早期版本为块设备共设立了 15 个 514 字节的缓冲区;为字符设备共设立了 100 个 8 字节的缓冲区,不同版本中的块设备和字符设备缓冲区大小可能不同。每类设备设计了相应的数据结构以及缓冲区自动管理软件,采用了完善的缓冲技术,引入了“预先读”“异步写”“延迟写”方式,提高 CPU 与设备 I/O 的并行性。

图 5-7 是 UNIX 字符设备的 I/O 字符缓存队列,每个字符设备都有一张设备表,它包含有 I/O 字符缓存队列的控制块。其中包括三项内容: $c - cc$ 队列中可用字符计数; $c - cf$ 指向队列中的第一个字符; $c - cl$ 指向队列的最后一个字符。现在来看一下取字符和释放字符缓存的问题以及送字符和申请字符缓存的过程。根据 $c - cf$ 指点,逐一从字符缓存里取出字符 e 和 f, $c - cc$ 做计数调整, $c - cf$ 也依次下移。若发现 $c - cf$ 的最低三位已为 0 时,这说明该字符缓存中的字符已全部取完, $c - cf$ 应指向下一字符缓存的第一个字符,即要把释放的这一个字符缓存中的 $c - next$ 加工送给 $c - cf$ 。反之,根据 $c - cl$ 指点往字符缓存里送字符,

显然,当 `c-cl` 获取下一存放字符的位置时,发现地址的最后三位为 0 时,表示目前的字符缓存已经存满,需要先申请一个新的字符缓存才能把字符存入。于是向系统申请一个新的字符缓存,链入到 I/O 字符缓存队列之尾,然后,再根据调整后的 `c-cl` 送入字符。

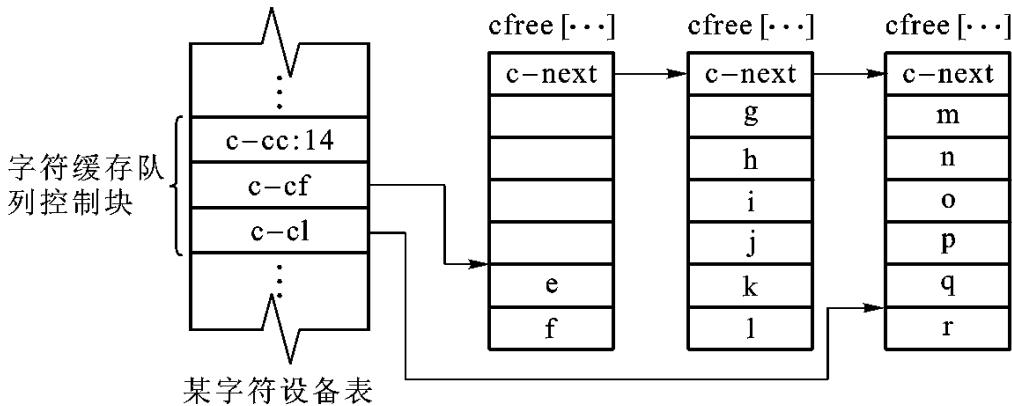


图 5-7 UNIX I/O 字符缓存队列

5.5 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,在繁重的输入输出负载之下,同时会有若干个输入输出请求来到并等待处理。系统必须采用一种调度策略,使能按最佳次序执行要求访问的诸请求,这就叫驱动调度,使用的算法叫驱动调度算法。驱动调度能减少为若干个输入输出请求服务所需的总时间,从而,提高系统效率。除了输入输出请求的优化排序外,信息在辅助存储器上的排列方式,存储空间的分配方法都能影响存取访问速度。

不同的外存储设备具有不同的信息安排方式,本节首先介绍顺序存取存储设备和直接存取存储设备的结构,然后针对磁盘讨论与驱动调度有关的技术。

5.5.1 存储设备的物理结构

顺序存取存储设备是严格依赖信息的物理位置进行定位和读写的存储设备,所以,从存取一个信息块到存取另一信息块要花费较多的时间。磁带机是最常用的一种顺序存取存储设备,由于它具有存储容量大、稳定可靠、卷可装卸和便于保存等优点,已被广泛用作存档的文件存储设备。磁带的存储如图 5-8 所示。

磁带的一个突出优点是物理块长的变化范围较大,块可以很小,也可以很大,原则上没有限制。为了保证可靠性,块长取适中较好,过小时不易区别干扰还是记录信息,过大对产

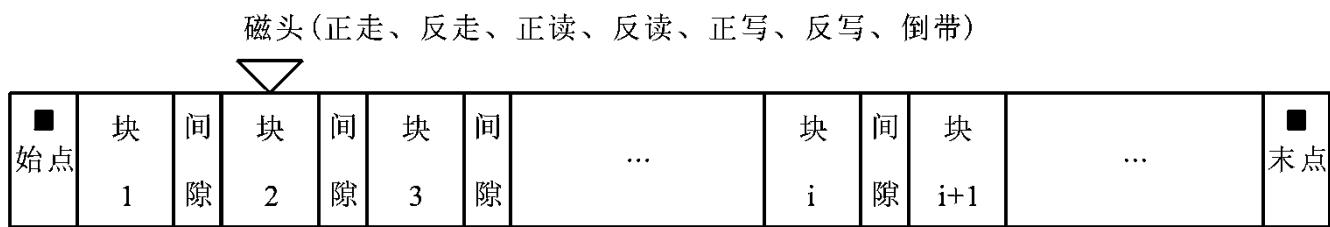


图 5-8 磁带存储示意图

生的误码就难以发现和校正。

磁带上的物理块没有确定的物理地址,而是由它在带上的物理位置来标识。磁头在磁带的始端,为了读出第 100 块上的记录信息,必须正向引带走过前面 99 块。对于磁带机,除了读/写一块物理记录的通道命令外,通常还有辅助命令,如反读、前跳、后退和标识,有一个称做带标的特殊记录块,只有使用写带标命令才能刻写。在执行读出、前跳和后退时,如果磁头遇到带标,硬件能产生设备特殊中断,通知操作系统进行相应处理。

磁盘是一种直接存取存储设备,又叫随机存取存储设备。它的每个物理记录有确定的位置和惟一的地址,存取任何一个物理块所需的时间几乎不依赖于此信息的位置。磁盘的结构如图 5-9 所示,它包括多个盘面用于存储数据。每个盘面有一个读写磁头,所有的读写磁头都固定在惟一的移动臂上同时移动。在一个盘面上的读写磁头的轨迹称磁道,在磁头位置下的所有磁道组成圆柱体称柱面,一个磁道又可被划分成一个或多个物理块。

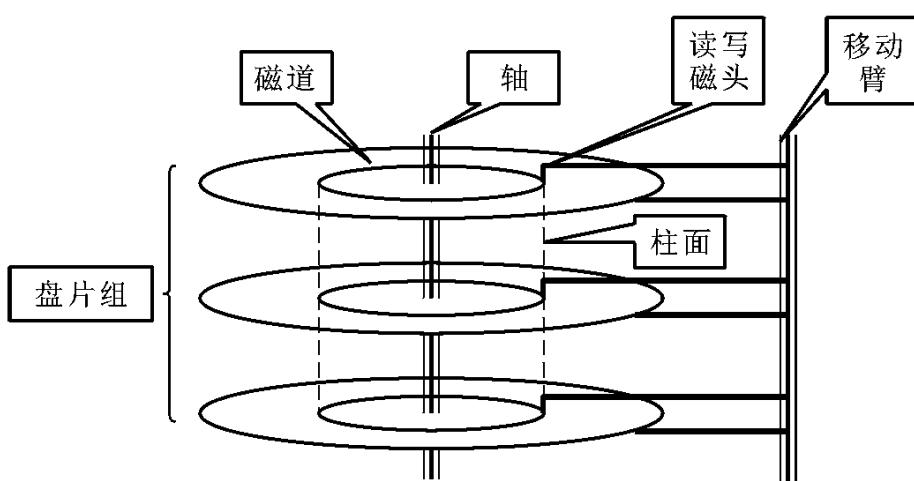


图 5-9 磁盘的结构图

文件的信息通常不是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,这样可使移动臂的移动次数减少,缩短存取信息的时间。为了访问磁盘上的一个物理记录,必须给出三个参数:柱面号、磁头号、块号。磁盘机根据柱面号控制臂作机械的横向移动,带动读写磁头到达指定柱面,这个动作较慢,一般称做“查找时间”,平均需 20 ms 左右。

下一步从磁头号可以确定数据所在的盘,然后,等待被访问的信息块旋转到读写头下时,按块号进行存取,这段等待时间称为“搜索延迟”,平均要 10 ms。磁盘机实现某些操作的通道命令是:查找、搜索、转移和读写。

5.5.2 循环排序

旋转型存储设备上不同记录的存取时间有明显的差别,所以,为了减少延迟时间,输入输出请求的某种排序有实际意义。考虑每一磁道保存 4 个记录的旋转型设备,假定收到以下 4 个输入输出请求。

| 请求次序 | 记录号 |
|------|-------|
| (1) | 读记录 4 |
| (2) | 读记录 3 |
| (3) | 读记录 2 |
| (4) | 读记录 1 |

对这些输入输出请求有多种排序方法:

- 方法 1:如果调度算法按照输入输出请求次序读记录 4、3、2、1,假定平均要用 $1/2$ 周来定位,再加上 $1/4$ 周读出记录。由于当读出记录 4 后需转过 $3/4$ 周才能去读记录 3,所以,总的处理时间等于 $1/2 + 1/4 + 3 \times 3/4 = 3$ 周,即 60 ms。

- 方法 2:如果调度算法决定的读入次序为读记录 1、2、3、4。那么,总的处理时间等于 $1/2 + 1/4 + 3 \times 1/4 = 1.5$ 周,即 30 ms。

- 方法 3:如果知道当前读位置是记录 3,则调度算法采用的次序为读记录 4、1、2、3 会更好。总的处理时间等于 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,这种硬设备叫做旋转位置测定。如果没有这种硬件装置,那么,因无法测定当前记录而可能会平均多花费半圈左右的时间。

循环排序时,还必须考虑某些输入输出的互斥问题。例如,读写磁鼓的记录信息需要两个参数:道号和记录号。如果请求是:

| 请求次序 | 磁道号 | 记录号 |
|-------|-----|------|
| (1) | 道 1 | 记录 2 |
| (2) | 道 1 | 记录 3 |
| (3) | 道 1 | 记录 1 |
| (4) | 道 6 | 记录 3 |
| W (5) | 道 4 | 记录 2 |

那么,请求(1)和(5)、请求(2)和(4),都互相排斥,因为,它们涉及的记录号相同。这样在第一转为请求(1)服务,第二转才能为请求(5)服务,或者反之。对于相同记录号的所有输入输出请求会产生竞争,如果硬件允许一次从多个磁道上读写,就可减少这种拥挤现象,但是,这通常需要附加的控制器,设备中还要增加电子部件。

5.5.3 优化分布

信息在存储空间的排列方式也会影响存取等待时间。考虑 10 个逻辑记录 A, B, ⋯, J 被存于旋转型设备上,每道存放 10 个记录,可安排如下:

| 物理块 | 逻辑记录 |
|-----|------|
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| 5 | E |
| 6 | F |
| 7 | G |
| 8 | H |
| 9 | I |
| 10 | J |

假定要经常顺序处理这些记录,而旋转速度为 20 ms,处理程序读出每个记录后花 4 ms 进行处理。则读出并处理记录 A 之后将转到记录 D 的开始。所以,为了读出 B,必须再转一周。于是,处理 10 个记录的总时间为:10 ms(移动到记录 A 的平均时间) + 2 ms(读记录 A) + 4 ms(处理记录 A) + 9 × [16 ms(访问下一记录) + 2 ms(读记录) + 4 ms(处理记录)] = 214 ms。

按照下面方式对信息优化分布:

| 物理块 | 逻辑记录 |
|-----|------|
| 1 | A |
| 2 | H |
| 3 | E |
| 4 | B |
| 5 | I |
| 6 | F |
| 7 | C |
| 8 | J |
| 9 | G |
| 10 | D |

当读出记录 A 并处理结束后, 恰巧转至记录 B 的位置, 立即就可读出并处理。按照这一方案, 处理 10 个记录的总时间为: 10 ms (移动到记录 A 的平均时间) + $10 \times [2 \text{ ms}$ (读记录) + 4 ms (处理记录)] = 70 ms 。所花时间是原方案的三分之一, 如果有众多记录需要处理, 节省时间更可观了。

5.5.4 交替地址

旋转型存储设备上的任意记录的存取时间主要由旋转速度来确定, 对于给定的装置这一速度是常数。把每个记录重复记录在这台设备的多个区域, 可以显著地减少存取时间。这样读相同的数据, 有几个交替地址, 这种方法也称为多重副本或折迭。

考虑一种设备, 若每道有 8 个记录, 则旋转速度 20 ms , 如果记录 A 存于道 1, 记录 1, 存取记录 A 平均占半周, 即 10 ms 。如果记录 A 的副本存于道 1, 记录 1 和道 1, 记录 5, 那么, 使用旋转位置测定, 总是存取“最近”的副本, 有效的平均存取时间可降为 5 ms 。类似地, 存储更多相同数据记录的副本, 可以把存取时间进一步折半。这一技术的主要缺点是耗用较多存储空间, 有效容量随副本个数增加而减少, 如果每个记录有 n 个副本, 存储空间就被“折迭”了 n 次。

此法成功与否取决于下列因素: 数据记录总是读出使用, 不需修改写入; 数据记录占用的存储空间总量不太大; 数据使用极为频繁。所以, 通常对系统程序可采用这一技巧, 它们能满足上述诸因素。

5.5.5 搜索定位

对于移动臂磁盘设备, 除了旋转位置外, 还有搜索定位的问题。输入输出请求需要 3 个参数: 柱面号、磁道号和记录号。例如, 对磁盘依次有以下 5 个访问请求。

| 柱面号 | 磁道号 | 记录号 |
|-----|-----|-----|
| 7 | 4 | 1 |
| 7 | 4 | 8 |
| 7 | 4 | 5 |
| 40 | 6 | 4 |
| 2 | 7 | 7 |

如果采用先来先服务算法(first - come first - served), 假设当前移动臂处于 0 号柱面, 若按上述次序访问磁盘, 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 的次序处理,这将会节省很多移臂时间。进一步考查 7 号柱面的三个访问,按上述次序,那么,必须使磁盘旋转近 2 圈才能访问完毕。若再次将访问请求排序,按照:

| 柱面号 | 磁道号 | 记录号 |
|-----|-----|-----|
| 7 | 4 | 1 |
| 7 | 4 | 5 |
| 7 | 4 | 8 |

执行,显然,对 7 号柱面的三次访问大约只要旋转 1 圈或更少就能访问完毕。由此可见,对于磁盘一类设备,在启动之前按驱动调度策略对访问的请求优化排序是十分必要的。除了应有使旋转圈数最少的调度策略外,还应考虑使移臂时间最短的调度策略。

移臂调度有若干策略,下面介绍几种移臂调度算法:

(1)“电梯调度”算法(elevator algorithm)是简单而实用的一种算法。如图 5-10 所示,按照这种策略每次总是选择沿臂的移动方向最近的那个柱面,如果同一柱面上有多个请求,还需进行旋转优化。如果沿这个方向没有访问的请求时,就改变臂的移动方向,并使移动频率极小化,然后,处理所遇到的最近的 I/O 请求,这非常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,然后,将当前移动方向和本次停留的位置都登记下来。如果不空,就让请求者等待并把它要求访问的位置登记下来,按照既定的调度算法对全体等待者进行寻查定序,下次按照优化的次序执行。如果有多个盘驱动器的请求同时到达时,系统还必须有优先启动哪一个盘组的 I/O 请求决策。

对于移动臂磁盘还有许多其他驱动调度算法,由于“先来先服务”调度算法下,磁盘臂的移动完全是随机的,不考虑各个 I/O 请求之间的相次对序和移动臂当前所处位置,进程等待 I/O 请求的时间会过长,寻道性能较差。

(2)“最短查找时间优先”算法(shortest seek time first algorithm)。本算法考虑了各个请求之间的区别,总是先执行查找时间最短的那个磁盘请求,从而,比“先来先服务”算法有较好的寻道性能。上面的例子,采用先来先服务算法磁头总共移动了 78 个柱面,而采用本算法磁头总共移动了 40 个柱面,节省了几乎一半的移臂时间。本算法存在“饥饿”现象,随着源源不断靠近当前磁头位置读写请求的到来,使早来的但距离当前磁头位置远的读写请求服务被无限期推迟。

(3)“扫描”算法(scan algorithm)。磁盘臂每次沿一个方向移动,扫过所有的柱面,遇到最近的 I/O 请求便进行处理,直到最后一个柱面后,再向相反方向移动回来。如果一个请求到达时其想访问的柱面刚巧在磁头移动前方,则该请求立即获得服务,反之如果刚好在磁头

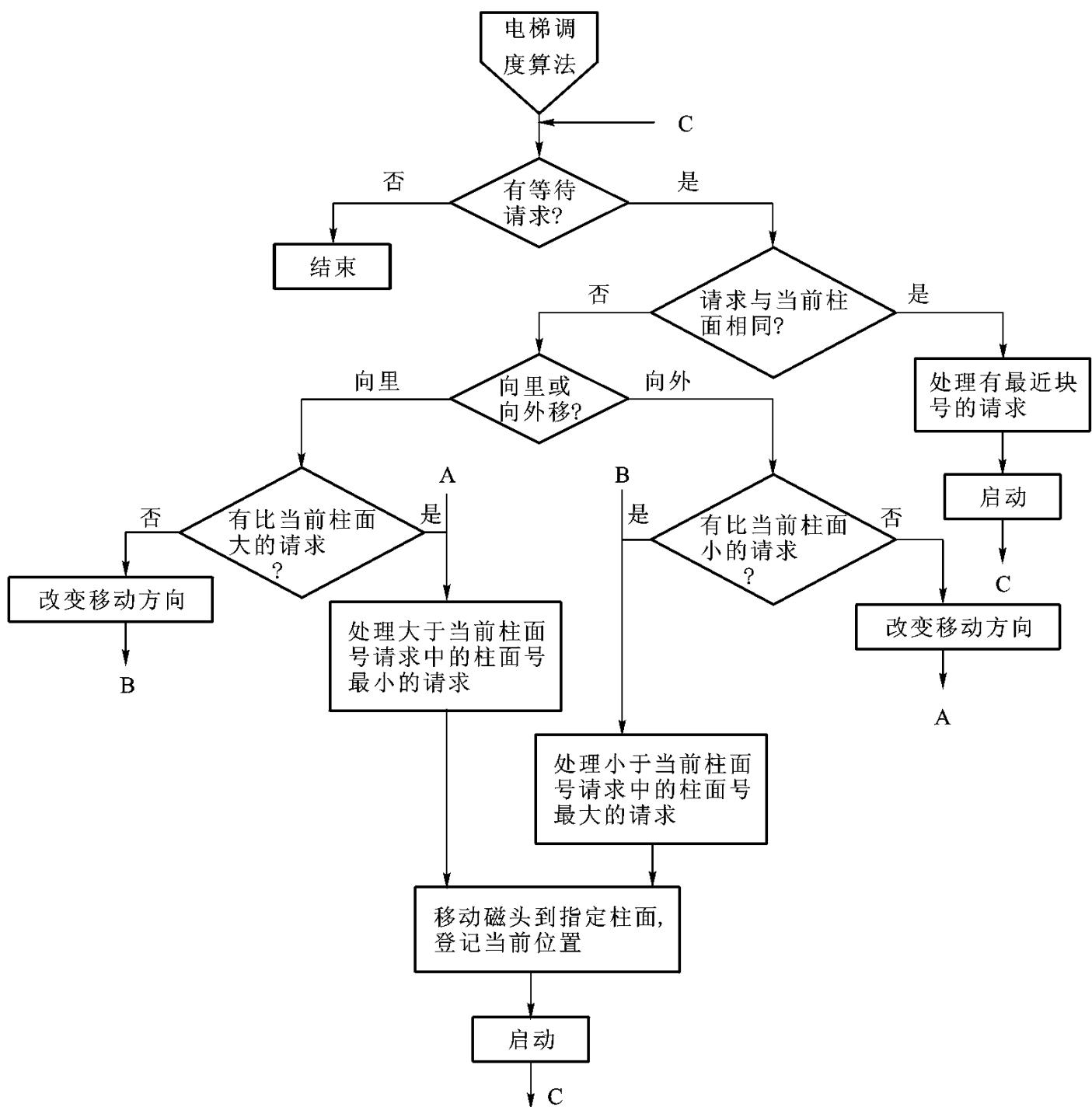


图 5-10 电梯调度算法

后面，则要等到磁头调转方向后才会得到响应。与“电梯调度”算法的不同在于：即使该移动方向暂时没有了 I/O 请求，移动臂也扫描到头。“最短查找时间优先”算法虽有较好的寻道性能，但可能会造成进程“饥饿”状态，而本算法克服了这一缺点。扫描算法偏爱那些最接近里面或靠外的请求，对最近扫描跨过去的区域的响应会较慢。

(4)“分步扫描”算法(N - steps scan algorithm)。一个或多个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”，采用分步扫描可避免这类问题。将 I/O 请求分成组，每

组不超过 N 个请求, 每次选一个组进行扫描, 处理完一组后再选下一组。这种调度算法能保证每个存取请求的等待时间不至太长。当 N 很大时, 接近于“扫描”算法的性能, 当 $N = 1$ 时, 接近于“先来先服务”算法的性能。

(5)“循环扫描”算法(circular scan algorithm)。电梯调度算法虽能杜绝饥饿, 但性能尚待改进, 在磁盘请求对柱面的分布是均匀的情况下, 当磁头到头并转向时, 靠近磁头一端的请求特少, 有许多请求集中分布在远离磁头的一端, 而这些请求的等待时间会较长。循环扫描算法能克服这个缺点。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一种扫描方式。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, 归途中不再服务, 构成了一个循环, 这就减少了处理新来请求的最大延迟。在一柱面上, 移动臂停留至磁盘旋转过一定圈数, 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, 为了在磁盘转动每一圈的时间内执行更多的存取, 必须考虑旋转优化问题。

“电梯调度”和“最短查找时间优先”两种算法, 在单位时间内处理的输入输出请求较多即吞吐量较大, 但是请求的等待时间较长;“电梯调度”种算法使请求的等待时间更长一些。一般说来“扫描”算法较好, 适宜于磁盘负载重的系统, 但它不分具体情况而扫过所有柱面造成性能不够好。“分步扫描”算法使得各个输入输出请求等待时间之间的差距最小, 而吞吐量适中。“电梯调度”算法杜绝饥饿, 性能适中。“循环扫描”算法仅适应不断有大批量柱面均匀分布的输入输出存取请求, 且磁道上存放记录数量较大的情况。

上面讨论的驱动调度算法能减少输入输出请求时间, 但都是以增加处理器时间为代价的。排队技术并不是在所有场合都适用的。这些算法的价值依赖于处理器的速度和输入输出请求的数量。如果输入输出请求较少, 采用多道程序设计后就可以达到较高的吞吐量。如果处理器速度很慢, 处理器的开销可能掩盖这些调度算法带来的好处。

5.5.6 独立磁盘冗余阵列

独立磁盘冗余阵列 RAID (Redundant Array of Independent Disks) 概念早在 1987 年就由美国加里福尼亚大学 Berkeley 分校一个研究小组的论文中提出, 现已得到工业界的认可。作为一种多磁盘数据库设计的标准样式(scheme), 已被广泛地应用于大中型计算机和计算机网络系统。它利用一台磁盘阵列控制器统一管理和控制一组磁盘驱动器, 组成一个速度快、可靠性高、性能价格比好的大容量外存储(磁盘)子系统。RAID 的提出填补了 CPU 速度快与磁盘设备速度慢之间的间隙, 其策略是:用一组较小容量的、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, 再加进冗余技术, 数据能用多种方式组织和分布存储, 于是, 独立的 I/O 请求能被并行处理, 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, 改进了 I/O 性能和系统可靠性。

RAID 样式共有 6 级组成, RAID0 至 RAID5, 最新又扩充了 RAID6 和 RAID7, 它们之间并不隐含层次关系, 而是标明了不同的设计结构, 并有 3 个共同特性:(1) RAID 是一组物理磁盘驱动器, 可以被操作系统看作是单一的逻辑磁盘驱动器;(2) 数据被分布存储在物理驱动器阵列上;(3) 使用冗余磁盘保存奇偶校验信息, 当磁盘出现失误时它能确保数据的恢复。虽然让多个磁头和驱动机构同时操作能达到较高 I/O 数据传输速率, 但使用多台硬设备也增加了故障的概率, 为了补偿可能造成的可靠性下降, RAID 利用存储的奇偶校验信息来恢复由于磁盘故障丢失的数据。RAID0 不支持第三个特性, 采用把大块数据分割成数据条块 (strip), 交替间隔地分布存储, 但未引入冗余磁盘, 适用于性能要求高, 但非要害数据的这类应用。RAID1 采用镜像技术, 适用于做系统驱动器, 存放关键的系统文件。RAID2 和 RAID3 主要采用了并行存取技术, 分别还引入海明 (Hamming code) 校验码和位插入 (bit interleaved) 奇偶校验码改进可靠性, 适用于大数据量 I/O 请求, 如图像和 CAD 这类的应用。RAID4 和 RAID5 主要采用了独立存取技术, 分别还引入块插入 (block interleaved) 奇偶校验码和块插入分布 (block interleaved distributed) 奇偶校验码改进可靠性, 适用于 I/O 请求频繁的事务处理。

1. RAID 0

用户和系统的数据划成条块被分布存储在横跨阵列中的所有磁盘上, 逻辑上连续的数据条块, 按条块并行存取, 在物理上可被循环地依次存储在横向相邻的磁盘驱动器上, 通过一个阵列管理软件进行逻辑地址空间到物理地址空间的映射。与单个大磁盘相比有着明显优点, 它能并行地处理要求位于不同磁盘上数据的不同 I/O 请求。RAID level0 并不能真正划入 RAID 家族, 因为, 它没有引入冗余校验来改进性能, 导致磁盘系统的可靠性差, 容易丢失数据, 故已较少使用。

2. RAID 1

RAID1 与 RAID2 至 RAID5 的主要差别在于实现冗余校验的方法不同。RAID1 简单地采用双份所有数据的办法, 类似于 RAID0, 数据划成条块被分布存储在横跨阵列中的所有磁盘上, 但是, 每个数据条块被存储到两个独立的物理磁盘上, 故阵列中的每个盘都有一个包含相同数据的它的镜像盘。这种数据组织方法的主要可取之处在于:(1) 读请求能通过包含相同请求数据中的任何一个磁盘来提供服务, 两个中的一个所花查找和搜索时间最少;(2) 写操作时, 要求改写对应的两个数据条块, 但这一点可采用并行操作实现, 写操作的性能由并行操作中较慢的一个决定;(3) 出现故障后的恢复很简单, 当一个驱动器出现故障, 数据可以从镜像盘获得。RAID1 的主要缺点是价格太贵, 空间利用率仅有一半, 往往作为存放系统软件、关键数据(金融和军政信息)和要害文件的驱动器;但是当磁盘故障时, 它提供了一个实时的数据备份, 所有的数据信息立即可用。

3. RAID 2

RAID2 和 RAID3 采用了并行存取技术。在并行存取阵列中,所有的磁盘参与每个 I/O 请求的执行,且每个驱动器的移动臂同步工作,使得任何时刻每个磁盘的磁头都在相同的位置。在其他 RAID 样式中,使用了数据条块,RAID2 和 RAID3 的数据条块非常小,常常小到单个字节或一个字。对于 RAID2 纠错码按照横跨的每个数据盘的相应位进行计算,并存储在多只校验盘的相应位的位置。典型地使用海明校验码,它能纠正单位错,发现双位错。

虽然 RAID2 所需磁盘数少于 RAID1,但价格仍然很贵。所需海明校验磁盘的数量与数据盘的多少成比例。例如,若每组有 14 台磁盘,则可以设置 4 台用于海明校验。在执行单个读操作时,所有的磁盘要被同时存取,请求的数据和关联的纠错码都提交给阵列控制器,如果发现有一个二进位错,阵列控制器能立即确认和纠正错误,所以,读操作时间并未变慢;在执行单个写请求时,所有的数据盘和奇偶校验盘必须被存取。

4. RAID 3

RAID3 的组织与 RAID2 相似,其差别是它仅使用一只冗余盘,而不论是多少大的磁盘阵列,以减少用于校验的磁盘个数。RAID3 也使用并行存取技术,数据分割成较小条块分布式存储,它用简单的奇偶校验代替上述复杂的海明校验,仍然按所有数据盘上相同位置的每个二进位的集合进行计算,而奇偶校验磁盘只要一个了。

当出现磁盘故障时,要使用奇偶校验盘的信息,数据可以用剩下的磁盘中的信息来重新构造。数据的构造十分简单,考虑有 5 个磁盘驱动器组成的阵列,若 X0 到 X3 存放数据,X4 为奇偶校验盘,对于第 i 位的奇偶校验位可如下计算:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

假定驱动器 X1 出故障,如果把 $X4(i)$ 和 $X1(i)$ 模 2 加到上面等式的两边,得到

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

因此,在阵列中的任何一只数据盘上的任一数据子块的内容,都能从阵列中剩余下来的其他磁盘的对应数据条块的内容来重新生成,这个原理适用于 RAID3、RAID4 和 RAID5。阵列中有一只磁盘发生故障时,少了一只磁盘后的其他所有数据依然可用,对于读操作来说,丢失的数据通过求异或或计算被重新生成。当数据被写到少了一只磁盘的 RAID 阵列时,必须维护奇偶校验的一致性。返回正常操作时需替换发生故障的磁盘,且故障盘的内容应当被重新生成到新的磁盘上。

5. RAID 4

RAID4 和 RAID5 使用了独立存取技术,在一个独立存取的磁盘阵列中,每个驱动器都可以独立地工作,所以,独立的 I/O 请求可以被并行地得到满足。因此,独立存取阵列适合于有频繁 I/O 请求的应用。同时,也只用一个校验盘。在 RAID4 和 RAID5 中,数据条块划得

较大,逐位(bit-by-bit)二进位奇偶校验数据块按横跨每个数据盘上对应的数据条块来计算,奇偶校验位存储在奇偶校验磁盘上的对应数据条块中。

每当执行一个小数据量写操作时,阵列管理软件不但要修改用户数据,而且,也要修改对应的奇偶校验位。考虑有5个驱动器组成的一个阵列,X0到X3存储数据,X4是奇偶校验磁盘。假如执行一个仅仅涉及磁盘X1上数据的写操作。开始的时候,对每一个二进位 i ,有下列关系式:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

在数据修改之后,对于改变了的二进位用'符号来指出,得到:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

为了计算新的奇偶校验位,阵列管理软件必须读出老的用户数据条块和老的奇偶校验码,因此,每个数据写操作包含了两次读操作和两次写操作,然后,用新数据和新奇偶校验位更新这两个数据条块。

执行一个大数据量写操作时,它涉及数据子块位于所有数据盘上,通过计算新数据的二进位便可很容易地计算出奇偶校验码,因此,奇偶磁盘与数据磁盘能被并行地修改,也就是说,没有额外的读写操作。由于任何写操作均涉及到奇偶校验盘,它也就成了一个瓶颈口。

6. RAID 5

RAID5的组织形式与RAID4类似,差别仅在于奇偶校验码是分布横跨存放在所有的磁盘上,典型的存放方法为循环分布法,设有 n 个磁盘的一个阵列,则开头的 n 个奇偶校验码螺旋式地位于 n 个磁盘上(即每个盘上有一个奇偶校验码),接着按这个模式再次重复。采用螺旋式把奇偶校验码分布横跨存放在所有的磁盘上,能避免RAID4中每次写数据都必须读和写校验盘一次,此时校验盘成了改善性能的关键。

7. RAID 6 和 RAID 7

这是增强型RAID。RAID6中设置了专用快速的异步校验磁盘,具有独立的数据访问通路,比低级RAID性能更好,但价格昂贵。RAID7对RAID6作了改进,该阵列中的所有磁盘都有较高传输速率,性能优异,但价格也很高。

5.5.7 提高磁盘I/O速度的一些方法

通常为磁盘设置高速缓存,这样能显著减少等待磁盘I/O的时间。下面介绍的一些方法也能有效提高磁盘I/O的速度。

提前读。用户经常采用顺序方式访问文件的各个盘块上的数据,在读当前盘块时已能

知道下次要读出的盘块的地址,因此,可在读当前盘块的同时,提前把下一个盘块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了缓冲区,便可直接使用数据,而不必再启动磁盘 I/O,从而,减少了读数据的时间,也就相当于提高了磁盘 I/O 速度。“提前读”功能已被许多操作系统如 UNIX、OS/2、Windows 等广泛采用。

延迟写。在执行写操作时,磁盘缓冲区中的数据本来应该立即写回磁盘,但考虑到该缓冲区中的数据不久之后再次被输出进程或其他进程访问,因此,并不马上把缓冲区中数据写盘,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也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,且分到了该缓冲区时,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了。只要存有输出数据的缓冲区还在队列中,任何访问该数据的进程,可直接从中读出数据,不必再去访问磁盘。这样做,可以减少磁盘的 I/O 时间,相当于提高了 I/O 速度。同样,在 UNIX、OS/2 和 Windows 中也采用了这一技术。

UNIX/Linux 便提供了两种读盘和三种写盘方式:正常读——把磁盘上的块信息读入内存缓冲区;提前读——在读一个磁盘当前块时,把下一个磁盘信息块也读入内存缓冲区;正常写——把内存缓冲区中的信息写到磁盘上,并且写进程应等待写操作完成;异步写——写进程无需等待写盘结束就可返回工作;延迟写——仅在缓冲区首部设置延迟写标志,然后,释放此缓冲区,并把该缓冲区链入空闲缓冲区链表的尾部。当有另外的进程申请到这个缓冲区时,才真正把缓冲区信息写入磁盘。

虚拟盘。指用内存空间去仿真磁盘,又叫 RAM 盘。该盘的设备驱动程序可以接受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内存中。操作过程对用户是透明的,即用户并不会发现这与真正的磁盘操作有什么不同,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,或重新启动系统时,原来保存在虚拟盘中的数据会丢失。因此,该盘常用于存放临时文件。虚拟盘与磁盘高速缓存的主要区别在于:前者内容完全由用户控制,而后者的内容是由操作系统控制的。

5.6 设备分配

5.6.1 设备独立性

现代计算机系统常常配置许多类型的外围设备,同类设备又有多台,尤其是多台磁带机的情况很普遍。作业在执行前,应对静态分配的外围设备提出申请要求,如果申请时指定某

一台具体的物理设备,那么,分配工作就很简单,但当指定的某台设备有故障时,就不能满足申请,该作业也就不能投入运行。例如,系统拥有 A、B 两台卡片输入机,现有作业 J2 申请一台卡片输入机,如果它指定使用卡片输入机 A,作业 J1 已经占用 A 或者设备 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,显然这样做很不合理。为了解决这一问题,通常用户不指定特定的设备,而指定逻辑设备,使得用户作业和物理设备独立开来,再通过其他途径建立逻辑设备和物理设备之间的对应关系,称设备的这种特性为“设备独立性”。具有设备独立性的系统中,用户编写程序时使用的设备与实际使用的设备无关,亦即逻辑设备名是用户命名的,是可以更改的,物理设备名(地址)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转换成物理设备名,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。

设备独立性带来的好处是:用户应用程序与物理外围设备无关,系统增减或变更外围设备时程序不必修改;易于对付输入输出设备的故障,例如,某台行式打印机发生故障时,可用另一台替换,甚至可用磁带机或磁盘机等不同类型的设备代替,从而,提高了系统的可靠性,增加了外围设备分配的灵活性,能更有效地利用外围设备资源,实现多道程序设计技术。

操作系统提供了设备独立性后,程序员可利用逻辑设备进行输入输出,而逻辑设备与物理设备之间的转换通常由操作系统的命令或语言来实现。由于操作系统大小和功能不同,具体实现逻辑设备到物理设备的转换就有差别,一般使用以下方法:利用作业控制语言实现批处理作业的设备转换;利用操作命令实现设备转换;利用高级语言的语句实现设备转换。

5.6.2 设备分配

现代计算机系统可以同时承担若干个用户的多个计算任务,设备管理的一个功能就是为计算机系统接纳的每个计算任务分配所需要的外围设备。从设备的特性来看,可以把设备分成独占设备、共享设备和虚拟设备三类,相应的管理和分配外围设备的技术可分成:独占方式、共享方式和虚拟方式,本节讨论前两种技术。

有些外围设备,如卡片输入机、卡片穿孔机、行式打印机、磁带机等,往往只能让一个作业独占使用,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输入机上装上一叠卡片,卡片上存放着该用户作业要处理的数据,由于作业执行中将随机地读入卡片上的数据进行加工处理,因此,不可能在该作业暂时不使用卡片输入机时,人为地换上另一作业的一叠卡片,让卡片输入机为另一作业服务。只有当某作业归还卡片输入机后,才能让另一作业去占用。

另一类设备,如磁盘、磁鼓等,往往可让多个作业共同使用,或者说,是多个作业可共享的设备。这是因为这一类设备容量大、存取速度快且可直接存取。例如,可把每个作业的信息组织成文件存放在磁盘上,使用信息时可按名查询文件,从磁盘上读出。用户提出存取文

件要求时,总是先由文件管理进行处理,确定信息存放位置,然后,再向设备管理提出驱动要求。所以,对于这一类设备,设备管理的主要工作是驱动工作,这包括驱动调度和实施驱动。

对独占使用的设备,往往采用静态分配方式,即在作业执行前,将作业所要用的这一类设备分配给它。当作业执行中不再需要使用这类设备,或作业结束撤离时,收回分配给它的这类设备。静态分配方式实现简单,能防止系统死锁,但采用这种分配方式,会降低设备的利用率。例如,对行式打印机,若采用静态分配,则在作业执行前把行式打印机分配给它,但一直到作业产生结果时才使用分配给它的行式打印机。这样,尽管这台行式打印机在大部分时间里处于空闲状态,但是,其他作业却不能使用。

如果对行式打印机采用动态分配方式,在作业执行过程中,要求建立一个行式打印机文件输出一批信息,系统才把一台行式打印机分配给该作业,当一个文件输出完毕关闭时,系统就收回分配给该作业的行式打印机。采用动态分配方式后,在行式打印机上可能依次输出了若干个作业的信息,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是属于哪个用户。所以,对某些独占使用的设备,采用动态分配方式,不仅是可行的而且也能提高设备的利用率。

对于磁盘、磁鼓等可共享的设备,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,但这样做会使存储空间利用率降低。

操作系统中,对 I/O 设备的分配算法常用的有:先请求先服务,优先级高者先服务等。此外,在多进程请求 I/O 设备分配时,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的死锁。

为了实现 I/O 设备的分配,系统中应设有设备分配的数据结构:设备类表和设备表。系统中拥有一张设备类表,每类设备对应于设备表中的一栏,其包括的内容通常有:设备类、总台数、空闲台数和设备表起始地址等。每一类设备,如输入机、行式打印机等都有各自的设备表,该表用来登记这类设备中每一台设备的状态,其包含的内容通常有:物理设备名(地址)、逻辑设备名、占有设备的进程号、已分配/未分配、好/坏等。按照上述分配使用的数据结构,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,为了对通道、控制器和每台设备进行管理和控制,要设置: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,它记录配置在系统中的所有物理设备的情况,每个物理设备占有一栏,包括:设备类型、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)、等待获得此部件的进程队列指针,及一次分配后相互勾链的指针,以备分配和执行 I/O 时使用。

5.7 虚拟设备

5.7.1 问题的提出

对于卡片输入输出机、行式打印机之类的设备采用静态分配方式是不利于提高系统效率的。首先,占有这些设备的作业不能有效地充分利用它们。一台设备在作业执行期间,往往只有一部分,甚至很少一部分时间在工作,其余时间均处于空闲状态。其次,这些设备分配给一个作业以后,再有申请这类设备的作业将被拒绝接受。例如,一个系统拥有两台卡片输入机,它就难于接受4个要求使用卡片输入机的作业同时执行,而占用卡片输入机的作业却又在占用的大部分时间里让它闲着。另外,这类慢速设备联机传输大大延长了作业的执行时间。为此,现代操作系统都提供虚拟设备的功能来解决这些问题。

早期,采用脱机外围设备操作,使用一台外围计算机,它的功能是以最大速度从读卡机上读取信息并记录到输入磁带上。然后,把包含有输入信息的输入磁带人工移动到主处理机上。在多道程序环境下,可让作业从磁带上读取各自的数据,运行的结果信息写入到输出磁带上。最后,把输出磁带移动到另一台外围计算机上,其任务是以最大速度读出信息并从打印机上输出。

完成上述输入和输出任务的计算机叫外围计算机,因为它不进行计算,只实现把信息从一台外围设备传送到另一台外围设备上。这种操作独立于主机处理,而在主处理机的直接控制下进行,所以,称做脱机外围设备操作。脱机外围设备操作把独占使用的设备转化为可共享的设备,在一定程度上提高了效率。但却带来了若干新的问题:

- 增加了外围计算机,不能充分发挥这些计算机的功效。
- 增加了操作员的手工操作,在主处理机和外围处理机之间要来回装上和取下输入输出卷,这种手工操作出错机会多,效率低。
- 不易实现优先级调度,不同批次中的作业无法搭配运行。

因此,应进一步考虑能否不使用外围计算机?由于现代计算机有较强的并行操作能力,处理器在执行计算的同时也可进行联机外围设备操作,故只需使用一台计算机就可完成上述三台计算机实现的功能。操作系统将大批信息从输入设备上预先输入到辅助存储器磁盘的输入缓冲区域中暂时保存,这种方式称为“预输入”。此后,由作业调度程序调出执行。作业使用数据时不必再启动输入设备,而只要从磁盘的输入缓冲区域中读入。类似地,作业执行中不必直接启动输出设备输出数据,而只要将作业的输出数据暂时保存到磁盘的输出缓冲区域中,在作业执行完毕后,由操作系统组织信息成批输出。这种方式称为“缓输出”。这

样一来设备的利用率提高了,其次,作业执行中不再和低速的设备联系,而直接从磁盘的输入缓冲区获得输入数据,且只要把输出信息写到磁盘的输出缓冲区就认为输出结束了,也就减少了作业等待输入输出数据的时间,从而,缩短了作业全部的执行时间。此外,还具有能增加多道程序的道数,增加作业调度的灵活性等优点。从上述分析可以看出,操作系统提供了外围设备联机同时操作功能后,系统的效率会有很大提高。与脱机外围设备操作相比,辅助存储器上的输入和输出缓冲区域相当于输入磁盘和输出磁盘,预输入和缓输出程序完成了外围计算机所做的工作。联机的同时外围设备操作又称做假脱机操作,采用这种技术后使得每个作业感到各自拥有独占使用的设备若干台。例如,虽然系统只有两台行式打印机,但是可使在处理机中的5个作业都感到各自拥有一台速度如同磁盘一样快的行式打印机,可以说采用了这种技术的操作系统提供了虚拟设备。Spooling技术是用一类物理设备模拟另一类物理设备的技术,是使独占使用的设备变成可共享设备的技术,也是一种速度匹配技术。操作系统中实现这种技术的功能模块称做斯普林系统。

5.7.2 Spooling 的设计和实现

为了存放从输入设备输入的信息以及作业执行的结果,系统在辅助存储器上开辟了输入井和输出井。“井”是用作缓冲的存储区域,采用井的技术能调节供求之间的矛盾,消除人工干预带来的损失。

图5-11给出了斯普林系统的组成和结构。为了实现联机同时外围设备操作功能,必须具有能将信息从输入设备输入到辅助存储器缓冲区域的“预输入程序”;能将信息从辅助存储器输出缓冲区域输出到输出设备的“缓输出程序”以及控制作业和辅助存储器缓冲区域之间交换信息的“井管理程序”。

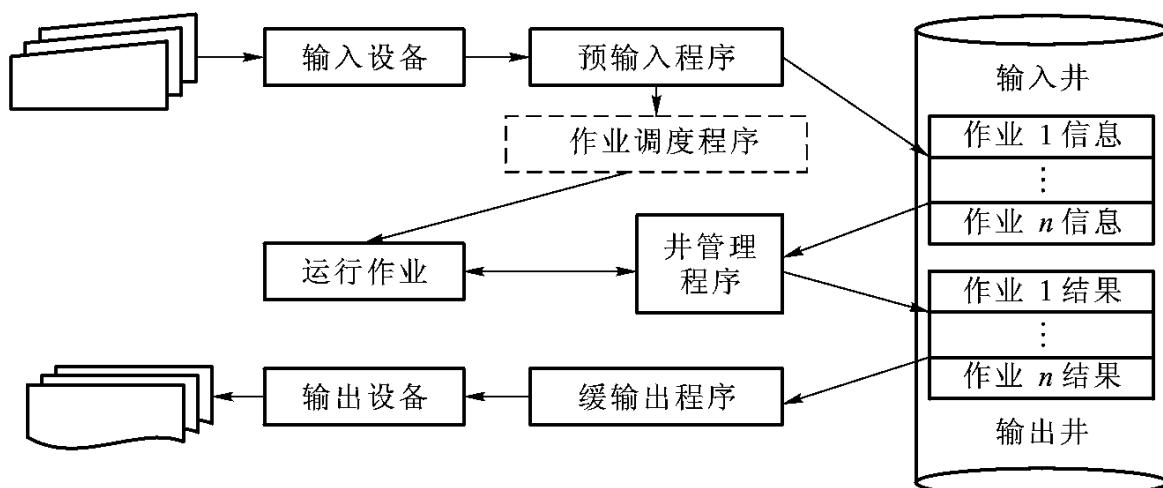


图 5-11 斯普林系统的组成和结构

预输入程序的主要任务是控制信息从输入设备输入到输入井存放，并填写好输入表以便在作业执行中要求输入信息时，可以随时找到它们的存放位置。

系统拥有一张作业表用来登记进入系统的所有作业的作业名、状态、预输入表位置等信息。每个用户作业拥有一张预输入表用来登记该作业的各个文件的情况，包括设备类、信息长度及存放位置等。

输入井中的作业有四种状态：

- 输入状态。作业的信息正从输入设备上预输入。
- 收容态。作业预输入结束但未被选中执行。
- 执行状态。作业已被选中运行，运行过程中，它可从输入井中读取数据信息，也可向输出井写信息。
- 完成状态。作业已经撤离，该作业的执行结果等待缓输出。

作业表指示了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收容状态的作业执行，作业表是作业调度程序进行作业调度的依据，是斯普林系统和作业调度程序共享的数据结构。

1. 预输入程序

通常，由操作员打入预输入命令启动预输入程序进行工作。系统响应预输入命令后，调出预输入程序，它查看作业表及输入井能否容纳新的作业。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和组织作业的信息，如作业名、优先数、处理机运行时间、源程序文件和数据文件等，将获得的作业信息以及预输入表的位置登记入作业表。此后，依次读入作业的信息，在输入井中寻找空闲块存放，把读入的信息以文件的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，用于存放一个或多个逻辑记录。可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，输入的信息被组织成连接文件，文件的第一块信息的位置登记在预输入表中，以后各块用指针连起来，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，文件空间利用率高。第二种是计算方式，假定从读卡机上读入信息并存放到磁盘的井文件空间，每张卡片为 80 个字节，每个磁道可存放 100 个 80 字节的记录，若每个柱面有 20 个磁道，则一个柱面可以存放 2000 张卡片信息。如果把 2000 张卡片作为一叠存放在一个柱面上，于是输入数据在磁盘上的位置为：第一张卡片信息是 0 号磁道的第 1 个记录，第二张卡片信息是 0 号磁道的第 2 个记录，第 101 张卡片信息是 1 号磁道的第 1 个记录，那么，第 n 张卡片信息被存放在：

$$\text{磁道号} = \text{卡片号 } n / 100$$

$$\text{记录号} = (\text{卡片号 } n) \bmod 100$$

用卡片号 n 除以 100 的整数和余数部分分别为其存放的磁道号和记录号。

2. 井管理程序

当作业执行过程中要求启动某台设备进行输入或输出操作时, 操作系统截获这个要求并调出井管理程序控制从相应输入井读取信息或将信息送至输出井内。例如, 作业 J 执行中要求从它指定的设备上读入某文件信息时, 井输入管理程序根据文件名查看其预输入表获得文件起始盘地址, 可以算出每次读请求所需的信息的存放位置。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。当输入井中的信息被作业取出后, 相应的井区应归还。通过预输入管理程序从输入井读入信息和通过设备管理从设备上输入信息, 对用户而言是一样的。

井管理程序处理输出操作的过程与上述类似。用户作业的输出信息一律通过输出井缓冲存放, 并在输出表中登记。缓输出表的格式与预输入表的格式类似, 包括作业名、作业状态、文件名、设备类、数据起始位置、数据当前位置等项。在作业执行当中要求输出数据时, 井输出管理程序根据有关信息查看输出表。如果表中没有这个文件名, 则为第一次请求输出。文件的第一个信息块的物理位置被填入起始位置, 每块信息写入井区前可用计算法计算出块号并将卡片数加 1, 或将后继块位置以连接方式写入信息块的连接字中, 再写到输出井, 并将下一次输出时接受输出信息的位置填入数据当前位置。如果不是第一次请求输出, 则缓输出表中已有登记, 只要从数据当前位置便可得到当前输出信息的井区。

3. 缓输出程序

当计算机的 CPU 有空闲时, 操作系统调出缓输出程序进行缓输出工作, 它查看缓输出表, 是否有输出打印的文件? 文件打印前可能需要组织作业或文件标题, 还可能对从输出井中读出的信息进行一定的格式加工。当一个作业的文件信息输出完毕后, 它占用的井区将被回收以供其他作业使用。

5.7.3 Spooling 应用例子

Spooling 是在多道程序系统中处理独占设备的一种方法, 这种技术已被广泛应用于许多设备和场合: 早期的读卡机、穿卡机; 现在的打印机、网络等。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来进行申请和使用, 但往往一个进程打开它后可能长达几个小时不用, 这期间其他进程又都无法打印。为解决这个问题, 可创建一个特殊的进程叫守护进程 (daemon) 以及一个特殊的目录, 称 spooling 打印目录。在打印一个文件之前, 进程首先产生完整的待打印文件并将其放在 spooling 打印目录下。规定系统中该守护进程是惟一有特权能够使用打印机设备文件的进程, 当打印机空闲时, 守护进程

便启动,打印待输出的文件。通过禁止用户直接使用打印机设备文件便解决了上述打印机空占的问题。

(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,发送文件前先将其放在一特定目录下,而后由网络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 Internet 电子邮件系统 (USENET),该网络将全世界数以千万台计的计算机联接在一起,通过拨号电话线或其他通信网络进行通信。当向某人发送 email 时,用户使用一个类似 send 的程序,它接收要发的信件并将其送入一个固定的 spooling 电子邮件目录下等待以后发送。整个 email 邮件系统在操作系统之外作为一种应用程序运行。

5.8 实例研究:Windows 2000/XP 的 I/O 系统

5.8.1 Windows 2000/XP I/O 系统结构和组件

Windows 2000/XP I/O 系统是 Windows 2000/XP 执行体的组件,存在于 NTOSKRNL.EXE 文件中。它接受来自用户态和核心态的 I/O 请求,并且以不同的形式把它们传送到 I/O 设备。Windows 2000/XP I/O 系统的设计目标如下:高效快速进行 I/O 处理;使用标准的安全机制保护共享的资源;满足 Win32、OS/2 和 POSIX 子系统指定的 I/O 服务的需要;允许用高级语言编写驱动程序;根据用户的配置或者系统中硬件设备的添加和删除,能在系统中动态地添加或删除相应的设备驱动程序;为包括 FAT、CD - ROM 文件系统(CDFS)、UDF 文件系统和 Windows 2000/XP 文件系统(NTFS)的多种可安排的文件系统提供支持;允许整个系统或者单个硬件设备进入和离开低功耗状态,这样可以节约能源。

Windows 2000/XP I/O 系统定义了 Windows 2000/XP 上的 I/O 处理模型,并且执行公用的或被多个驱动程序请求的功能。它主要负责创建代表 I/O 请求的 IRP 和引导通过不同驱动程序的包,在完成 I/O 时向调用者返回结果。I/O 管理器通过使用 I/O 系统对象来定位不同的驱动程序和设备,这些对象包括驱动程序对象和设备对象。内部的 Windows 2000/XP I/O 系统以异步操作方式获得高性能,并且向用户态应用程序提供同步和异步 I/O 功能。

设备驱动程序不仅包括传统的硬件设备驱动程序,还包括文件系统、网络和分层过滤器驱动程序。通过使用公用机制,所有驱动程序都具有相同的结构,并以相同的机制在彼此之间和 I/O 管理器通信。所以,它们可以被分层,即把一层放在另一层上来达到模块化,并可以减少在驱动程序之间的复制。同样,所有的 Windows 2000/XP 设备驱动程序都应被设计

成能够在多处理器系统下工作。

下面首先介绍 Windows 2000/XP I/O 系统的结构和组件以及不同类型的设备驱动程序,然后,描述设备、设备驱动程序和 I/O 请求的关键的数据结构以及设备驱动程序分类和结构。最后,将进一步分析在整个系统中完成 I/O 请求的必要步骤。

Windows 2000/XP 的 I/O 系统由一些执行体组件和设备驱动程序组成,如图 5-12 所示。

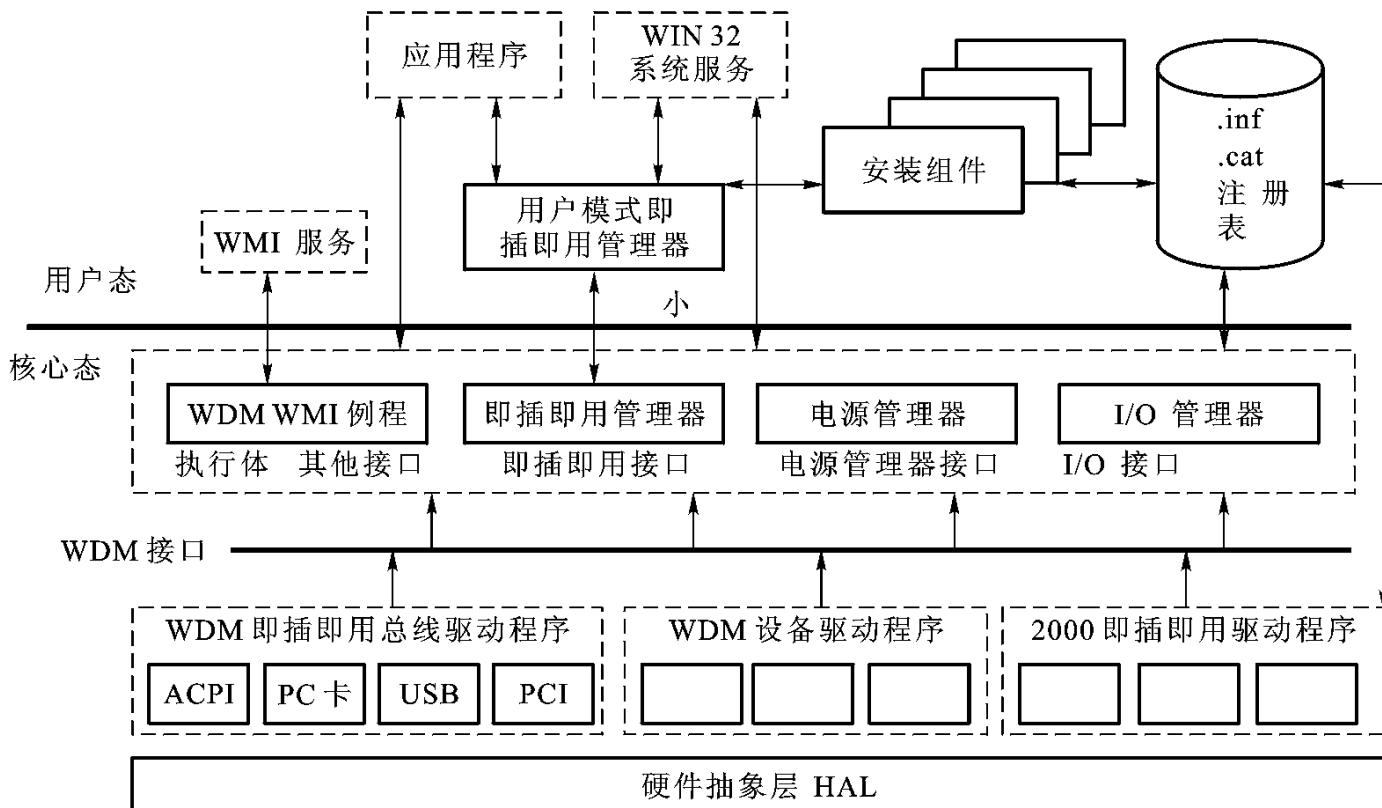


图 5-12 I/O 系统组件

- 用户态即插即用组件 用于控制和配置设备的用户态 API。
- I/O 管理器 把应用程序和系统组件连接到各种虚拟的、逻辑的和物理的设备上,并且定义了一个支持设备驱动程序的基本构架。负责驱动 I/O 请求的处理,为设备驱动程序提供核心服务。它把用户态的读写转化为 I/O 请求包 IRP。
- 设备驱动程序 为某种类型的设备提供一个 I/O 接口。设备驱动程序从 I/O 管理器接受处理命令,当处理完毕后通知 I/O 管理器。设备驱动程序之间的协同工作也通过 I/O 管理器进行。
- 即插即用管理器 PnP(plug and play) 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,并且检测相应硬件设备的添加和删除。
- 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,完成不同电源状态的转换。

• WMI(Windows Management Instrumentation)支持例程 也叫做 Windows 驱动程序模型 WDM(Windows Driver Model)WMI 提供者,允许驱动程序使用这些支持例程作为媒介,与用户态运行的 WMI 服务通信。

• 即插即用 WDM 接口 I/O 系统为驱动程序提供了分层结构,这一结构包括 WDM 驱动程序、驱动程序层和设备对象。WDM 驱动程序可以分为三类:总线驱动程序、驱动程序和过滤器驱动程序。每一个设备都含有两个以上的驱动程序层,用于支持它所基于的 I/O 总线的总线驱动程序,用于支持设备的功能驱动程序,以及可选的对总线、设备或设备类的 I/O 请求进行分类的过滤器驱动程序。

• 注册表 作为一个数据库,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。

• 硬件抽象层(HAL) I/O 访问例程把设备驱动程序与多种多样的硬件平台隔离开来,使它们在给定的体系结构中是二进制可移植的,并在 Windows 2000/XP 支持的硬件体系结构中是源代码可移植的。

大部分 I/O 操作并不会涉及所有的 I/O 组件,一个典型的 I/O 操作从应用程序调用一个与 I/O 操作有关的函数开始,通常会涉及 I/O 管理器、一个或多个设备驱动程序以及硬件抽象层。

在 Windows 2000/XP 中,所有的 I/O 操作都通过虚拟文件执行,隐藏了 I/O 操作目标的实现细节,为应用程序提供了一个统一的到设备的接口。虚拟文件是指用于 I/O 的所有源或目标,它们都被当做文件来处理(例如文件、目录、管道和邮箱)。所有被读取或写入的数据都可以看作是直接读写到这些虚拟文件的流。用户态应用程序(不管它们是 Win32、POSIX 或 OS/2)调用文档化的函数(公开的调用接口),这些函数再依次调用内部 I/O 子系统函数来从文件中读取、对文件写入和执行其他的操作。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。一个典型的 I/O 请求流程的结构如图 5-13 所示。

下面将进一步讨论其中的一些组件,详细地叙述 I/O 管理器,论述不同类型的设备驱动程序和关键的 I/O 系统数据结构,并介绍 PnP 管理器和电源管理器的结构。

1. I/O 管理器

“I/O 管理器”(I/O manager)定义有序的工作框架,在该框架里,I/O 请求被提交给设备驱动程序。在 Windows 2000/XP 中,整个 I/O 系统是由“包”驱动的,大多数 I/O 请求用 I/O 请求包 IRP(I/O Request Packet)”表示,它从一个 I/O 系统组件移动到另一个 I/O 系统组件,快速 I/O 是一个特例,它不使用 IRP。IRP 是在每个阶段控制如何处理 I/O 操作的数据结构。

I/O 管理器创建代表每个 I/O 操作的 IRP,传递 IRP 给正确的驱动程序,并且当此 I/O 操作完成后,处理这个数据包。相反,驱动程序接受 IRP,执行 IRP 指定的操作,并且在完成

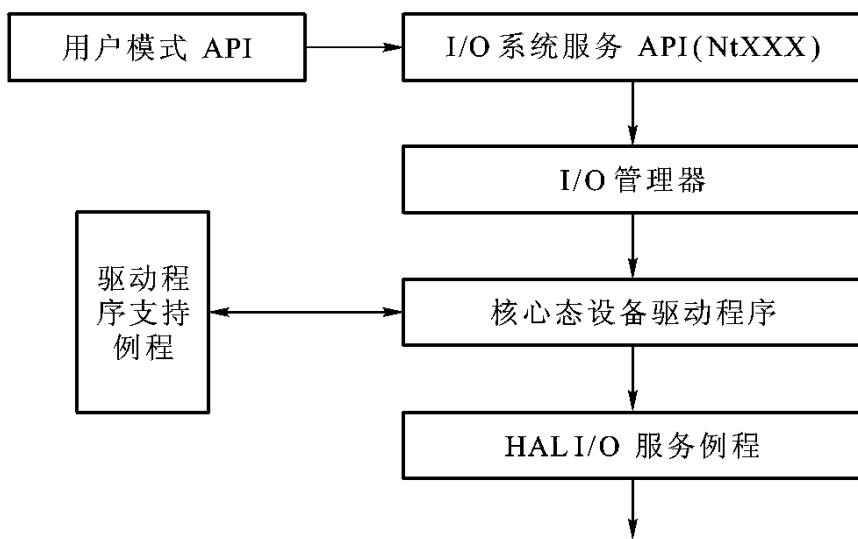


图 5-13 一个典型的 I/O 请求流程

操作后把 IRP 送回 I/O 管理器或为下一步的处理而通过 I/O 管理器把它送到另一个驱动程序。

除了创建并处理 IRP 以外, I/O 管理器还为不同的驱动程序提供了公共的代码, 驱动程序调用这些代码来执行它们的 I/O 处理。通过在 I/O 管理器中合并公共的任务, 单个的驱动程序将变得更加简洁和更加紧凑。例如, I/O 管理器提供一个允许某个驱动程序调用其他驱动程序的函数。它还管理用于 I/O 请求的缓冲区, 为驱动程序提供超时支持, 并记录操作系统中加载了哪些可安装的文件系统。

I/O 管理器也提供灵活的 I/O 服务, 允许环境子系统(例如, Win32 和 POSIX)执行它们各自的 I/O 函数。这些服务包括用于异步 I/O 的高级服务, 它们允许开发者建立可升级的高性能的服务器应用程序。

驱动程序呈现的统一的、模块化的接口允许 I/O 管理器调用任何驱动程序而不需要与它的结构和内部细节有关的任何特殊的知识。驱动程序也可以相互调用(通过 I/O 管理器)来完成 I/O 请求的分层的、独立的处理。

2. PnP 管理器

即插即用 PnP (Plug and Play) 是计算机系统 I/O 设备与部件配置的应用技术。顾名思义, PnP 是指插入就可用, 不需要进行任何设置操作。由于一个系统可以配置多种外部设备, 设备也经常变动和更换, 它们都要占有一定的系统资源, 彼此间在硬件和软件上可能会产生冲突。因此, 在系统中要正确地对它们进行配置和资源匹配; 当设备撤除、添置和进行系统升级时, 配置过程往往是一个困难的过程。为了改变这种状况, 出现了 PnP 技术。

PnP 技术主要有以下特点:PnP 技术支持 I/O 设备及部件的自动配置, 使用户能够简单方便地使用系统扩充设备; PnP 技术减少了由制造商造成的种种用户限制, 简化了部件的硬

件跳线设置,使 I/O 附加卡和部件不再具有人工跳线设置电路;利用 PnP 技术可以在主机板和附加卡上保存系统资源的配置参数和分配状态,有利于系统对整个 I/O 资源的分配和控制;PnP 技术支持和兼容各种操作系统平台,具有很强的扩展性和可移植性;PnP 技术在一定程度上具有“热插入”、“热拼接”功能。

PnP 技术的实现需要多方面的支持,其中包括:具有 PnP 功能的操作系统、配置管理软件、软件安装程序和设备驱动程序等;另外还需要系统平台的支持(如 PnP 主机板、控制芯片组和支持 PnP 的 BIOS 等)以及各种支持 PnP 规范的总线、I/O 控制卡和部件。

PnP 管理器为 Windows 2000/XP 提供了识别并适应计算机系统硬件配置变化的能力。PnP 支持需要硬件、设备驱动程序和操作系统的协同工作才能实现。关于总线上设备标识的工业标准是实现 PnP 支持的基础,例如,USB 标准定义了 USB 总线上识别 USB 设备的方式。Windows 2000/XP 的 PnP 支持提供了以下能力:

- PnP 管理器自动识别所有已经安装的硬件设备。在系统启动的时候,一个进程会检测系统中硬件设备的添加或删除。
- PnP 管理器通过一个名为资源仲裁(resource arbitrating)的进程收集硬件资源需求(中断,I/O 地址等)来实现硬件资源的优化分配;满足系统中的每一个硬件设备的资源需求。PnP 管理器还可以在启动后根据系统中硬件配置的变化对硬件资源重新进行分配。
- PnP 管理器通过硬件标识选择应该加载的设备驱动程序。如果找到相应的设备驱动程序,则通过 I/O 管理器加载,否则,启动相应的用户态进程请求用户指定相应的设备驱动程序。
- PnP 管理器也为检测硬件配置变化提供了应用程序和驱动程序的接口。因此,在 Windows 2000/XP 中,在硬件配置发生变化的时候,相应的应用程序和驱动程序也会得到通知。

Windows 2000/XP 的目标是提供完全的 PnP 支持,但是具体的 PnP 支持程度要由硬件设备和相应驱动程序共同决定。如果某个硬件或驱动程序不支持 PnP,整个系统的 PnP 支持将受到影响。一个不支持 PnP 的驱动程序可能会影响其他设备的正常使用。一些比较早的设备和相应的驱动程序可能都不支持 PnP。在 NT4 下可以正常工作的驱动程序一般情况下在 Windows 2000/XP 中也可以工作,PnP 就不能通过这些驱动程序完成设备资源的动态配置。

为了支持 PnP,设备驱动程序必须支持 PnP 调度例程和添加设备的例程,总线驱动程序必须支持不同类型的 PnP 请求。在系统启动的过程中,PnP 管理器向总线驱动程序询问得到不同设备的描述信息,包括设备标识、资源分配需求等,然后,PnP 管理器就加载相应的设备驱动程序并调用每一个设备驱动程序的添加设备例程。

设备驱动程序加载后已经做好了开始管理硬件设备的准备,但是并没有真正开始和硬

件设备通信。设备驱动程序等待 PnP 管理器向其 PnP 调度例程发出启动设备 (start-device) 的命令, 启动设备命令中包含 PnP 管理器在资源仲裁后确定的设备的硬件资源分配信息。设备驱动程序收到启动设备命令后开始驱动相应设备并使用所分配的硬件资源开始工作。

设备启动后, PnP 管理器可以向设备驱动程序发送其他的 PnP 命令, 包括把设备从系统中卸装, 重新分配硬件资源等。把设备从系统中移开包括的 PnP 命令有 query, remove 等, 重新分配硬件资源涉及的 PnP 命令有 query—stop, stop, start—device 等。

3. 电源管理器

同 Windows 2000/XP 的 PnP 支持一样, 电源管理也需要底层硬件的支持, 底层的硬件需要符合 ACPI (Advanced Configuration and Power Interface) 标准。因此, 支持电源管理的计算机系统的 BIOS (Basic Input and Output System) 必须符合 ACPI 标准, 1998 年底以来的 x86 计算机系统都符合 ACPI 标准。

ACPI 为系统和设备定义了不同的能耗状态, 目前共有六种: S₀ (正常工作)、S₁ 到 S₃ (睡眠)、S₄ (休眠)、S₅ (完全关闭)。每一种状态都有如下指标:

- 电源消耗: 计算机系统消耗的能源。
- 软件运行恢复: 计算机系统回复到正常工作状态时软件能否恢复运行。
- 硬件延迟: 计算机系统回复到正常工作状态的时间延迟。

计算机系统可以在从 S₁ ~ S₅ 的状态之间互相转换, 转换必须先通过状态 S₀。S₁ ~ S₅ 的状态转换到 S₀ 称做唤醒, 从 S₀ 转换到 S₁ ~ S₅ 称做睡眠。

设备也有相应的能耗状态, 设备能耗状态的分类和整个计算机系统是不同的。ACPI 定义的设备能耗分为四种状态: 从 D₀ 到 D₃。其中 D₀ 为正常工作, D₃ 为关闭, D₁ 和 D₂ 的意义可以由设备和驱动程序自行定义, 只要保证 D₁ 耗能低于 D₂, D₂ 耗能低于 D₃ 即可。

Windows 2000/XP 的电源管理策略由两部分组成: 电源管理器和设备驱动程序。电源管理器是系统电源策略的所有者, 因此整个系统的能耗状态转换由电源管理器决定, 并调用相应设备的驱动程序完成。电源管理器根据以下因素决定当前相同的能耗状态: 系统活动状况; 系统电源状况; 应用程序的关机、休眠请求; 用户的操作 (例如用户按电源按钮); 控制面板的电源设置。

当电源管理器决定要转换能耗状态时, 相应的电源管理命令会发给设备驱动程序的相应调度例程。一个设备可能需要多个设备驱动程序, 但是负责电源管理的设备驱动程序只有一个, 设备驱动程序根据当前系统状态和设备的状态决定如何进行下一步操作, 例如, 当设备状态从 S₀ 切换到 S₁ 时, 设备的能耗状态也从 D₀ 切换到 D₁。

除了响应电源管理器的电源管理命令外, 设备驱动程序也可以独立地控制设备的能耗状态。在一些情况下, 当设备长时间不用时, 设备驱动程序就可以减小该设备的能耗。设备驱动程序可以自己检测设备的闲置时间, 也可以通过电源管理器检测。在使用电源管理器

时,设备驱动程序通过调用函数 PoRegisterDeviceForIdleDetection 将相应设备注册到电源管理器中,该函数告诉电源管理器检测设备闲置的超时参数以及发现设备闲置时应该把设备切换到何种能耗状态。驱动程序需要设置两个超时值,一个用于配置计算机节省电能,另一个用于使计算机达到最优性能。调用了 PoRegisterDeviceForIdleDetection 函数后,设备驱动程序需要通过函数 PoSetDeviceBusy 通知电源管理器设备何时被激活。

5.8.2 Windows 2000/XP I/O 系统的数据结构

四种主要的数据结构代表了 I/O 请求:文件对象、驱动程序对象、设备对象和 I/O 请求包(IRP)。

1. 文件对象

文件明显符合 Windows 2000/XP 中的对象标准:它们是两个或两个以上用户态进程的线程可以共享的系统资源,可以有名称,被基于对象的安全性所保护,并且它们支持同步。虽然在 Windows 2000/XP 中的大多数共享资源是基于内存的资源,但是 I/O 系统管理的大多数资源位于物理设备中或者就是实际的物理设备。尽管这有些差异,但在 I/O 系统中的共享资源,像在 Windows 2000/XP 执行体的其他组件中的一样,都作为对象而被操作。

文件对象提供了基于内存的共享物理资源的表示法(除了被命名的管道和邮箱以外,它们虽然是基于内存的,但不是物理资源)。在 Windows 2000/XP I/O 系统中,文件对象也代表这些资源。表 5-2 出了一些文件对象的属性。

表 5-2 文件对象属性

| 属 性 | 目 的 |
|---------------|------------------------------------------|
| 文件名 | 标识文件对象指向的物理文件 |
| 字节偏移量 | 在文件中标识当前位置(只对同步 I/O 有效) |
| 共享模式 | 表示当调用者正在使用文件时,其他的调用者是否可以打开文件来做读取、写入或删除操作 |
| 打开模式 | 表示 I/O 是否将被同步或异步、高速缓存或不高速缓存、连续或随机等 |
| 指向设备对象的指针 | 表示文件在其上驻留的设备的类型 |
| 指向卷参数块的指针 | 表示文件在其上驻留的卷或分区 |
| 指向区域对象指针的指针 | 表示描述一个映射文件的根结构 |
| 指向专用高速缓存映射的指针 | 表示文件的哪一部分由高速缓存管理器管理,以及它们驻留在高速缓存的什么地方 |

当调用者打开文件或单一的设备时, I/O 管理器将为文件对象返回一个句柄。像其他的执行体对象一样, 文件对象由包含访问控制表(ACL)的安全描述体保护。I/O 管理器查看安全子系统来决定文件的 ACL 是否允许进程去访问它的线程正在请求的文件。如果允许对象管理器将准予访问, 并把它返回的文件句柄和给予的访问权限联系起来。如果这个线程或在进程中的另一个线程需要去执行另外的操作, 而不是最初请求指定的操作, 那么线程就必须打开另一个句柄, 它将提示做另外的安全检查。

文件对象表示一个基于内存的共享资源, 它有别于其他的执行体对象。一个文件对象包括的惟一数据结构是对象句柄, 但是文件本身包括将被共享的数据或文本。每次当一个线程打开一个文件句柄时, 就创建了一个新的文件, 其属性由一组新的句柄指定。例如, 属性字节偏移量指的是在文件中下一次将要使用那个句柄做读取或写入操作的位置。每一个为文件打开句柄的线程都有专用的字节偏移量, 即使在底层的文件是被共享的。除了当一个进程复制一个文件句柄给另一个进程, 或当一个子进程从它的父进程那里继承一个文件句柄以外, 文件对象对于进程来说是惟一的。而在这些情况下, 两个进程分开引用二个文件对象的句柄。尽管一个文件句柄对一个进程可能是惟一的, 但在底层的物理资源却不是这样。因此, 当使用任何共享的资源时, 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, 当它打开文件句柄去防止其他的线程在同一时间对该文件写入时, 它应该指定独占的写访问。还可以使用 Win32 的 LockFile 函数, 它可以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一个句柄时, I/O 管理器必须根据文件对象的名称来决定它将调用哪个或哪些驱动程序来处理请求。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可以定位这个信息。下面的系统对象满足这些要求:

- 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并且为 I/O 记录每个驱动程序的调度例程的地址(入口点)。
- 设备对象在系统中代表一个物理的、逻辑的或虚拟的设备并描述了它的特征, 例如, 它所需要的缓冲区的对齐方式和它用来保存即将到来的 I/O 请求包的设备队列的位置。

当驱动程序被加载到系统中时, I/O 管理器将创建一个驱动程序对象, 然后, 它调用驱动程序的初始化例程, 该例程把驱动程序的入口点填放到该驱动程序对象中。初始化例程还创建用于每个设备的设备对象, 这样使设备对象脱离了驱动程序对象。

当打开一个文件时, 文件名包括文件驻留的设备对象的名称。例如, 名称 \Device\Floppy0\myfile.dat 引用软盘驱动器 A 上的文件 myfile.dat。子字符串 kDevice\Floppy0 是 Windows 2000/XP 内部设备对象的名称, 代表那个软盘驱动器。当打开 myfile.dat 文件时, I/O 管理器就创建一个文件对象, 并在文件对象中存储一个 Floppy0 设备的指针, 然后, 给调用

者返回一个文件句柄。此后,当调用者使用文件句柄时, I/O 管理器能够直接找到 Floppy0 设备对象。请记住,在 Win32 应用程序中不能使用 Windows 2000/XP 内部设备名称。相反,设备名称必须出现在对象管理器的名字空间中的一个特定的目录中,这个目录包括到实际的 Windows 2000/XP 内部设备名称的符号链接。在这个目录中,设备驱动程序负责创建链接以使它们的设备能在 Win32 应用程序中被访问。通过使用 Win32 的 QueryDosDevice 和 DefineDosDevice 函数,可以检查、甚至可以用编程的方式来改变这些链接。

如图 5-14 所示,设备对象反过来指向它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 请求时应该调用哪个驱动程序。它使用设备对象找到代表服务于该设备驱动程序的驱动程序对象,然后利用在初始请求中提供的功能码来索引驱动程序对象。每个功能码都对应于一个驱动程序的入口点。

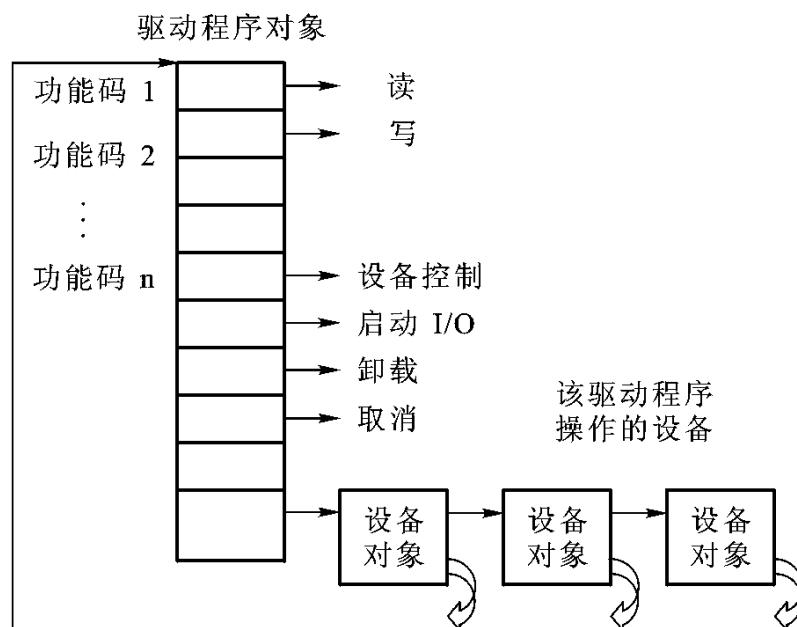


图 5-14 驱动程序对象

驱动程序对象通常有多个与它相关的设备对象。设备对象列表代表驱动程序可以控制的物理设备、逻辑设备和虚拟设备。例如,硬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,相同的硬盘驱动程序被用于访问所有的分区。当一个驱动程序从系统中被卸载时,I/O 管理器就会使用设备对象队列来确定哪个设备由于取走了驱动程序而受到了影响。

3. I/O 请求包

IRP(I/O Request Packet)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 管理器就构造一个 IRP 来表示在整个系统 I/O 进展中要进行的操作。I/O 管理器在 IRP 中保存一个指向调用者文件对象的指针。

IRP 由两部分组成:固定部分(称做标题)和一个或多个堆栈单元。固定部分信息包括:请求的类型和大小、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和随着请求的进展而变化的状态信息。IRP 的堆栈单元包括一个功能码、功能特定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 IRP 都存储在与请求 I/O 的线程相关的 IRP 队列中。如果一个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。

5.8.3 Windows 2000/XP 设备驱动程序

Windows 2000/XP 支持多种类型的设备驱动程序和编程环境,在同一种驱动程序中也存在不同的编程环境,具体取决于硬件设备。这里主要讨论核心模式的驱动程序,核心驱动程序的种类很多,主要分为以下几种:

- 文件系统驱动程序:接受访问文件的 I/O 请求,主要是针对大容量设备和网络设备。
- PnP 管理器和电源管理器设备驱动程序:包括大容量存储设备、协议栈和网络适配器等。
- 为 Windows NT 编写的设备驱动程序:可以在 Windows 2000/XP 中工作,但是一般不具备电源管理和 PnP 的支持,会影响整个系统的电源管理和 PnP 管理的能力。
- Win32 子系统显示驱动程序和打印驱动程序:将把与设备无关的图形 (GDI) 请求转换为设备专用请求。这些驱动程序的集合被称为“核心态图形驱动程序”。显示驱动程序与视频小端口 (miniport) 驱动程序是成对的,用来完成视频显示支持。每个视频小端口驱动程序为与它关联的显示驱动程序提供硬件级的支持。
- 符合 Windows 驱动程序模型的 WDM 驱动程序:包括对 PnP、电源管理和 WMI 的支持。WDM 在 Windows 2000/XP、Windows 98 和 Windows ME 中都是被支持的,因此,在这些操作系统中是源代码级兼容的,在许多情况下是二进制兼容的。有三种类型的 WDM 驱动程序:(1) 总线驱动程序 (bus driver) 管理逻辑的或物理的总线,例如,PCMCIA、PCI、USB、IEEE1394 和 ISA,总线驱动程序需要检测并向 PnP 管理器通知总线上的设备,并且能够管理电源。(2) 功能驱动程序 (function driver) 管理具体的一种设备,对硬件设备进行的操作都是通过功能驱动程序进行的。(3) 过滤器驱动程序 (filterdriver) 与功能驱动程序协同工作,用于增加或改变功能驱动程序的行为。

除了以上驱动程序类型外,Windows 2000/XP 还支持一些用户模式的驱动程序:

- 虚拟设备驱动程序 (VDD) 通常用于模拟 16 位 MS - DOS 应用程序。它们捕获 MS - DOS 应用程序对 I/O 端口的引用,并将其转化为本机 Win32 I/O 函数。因为 Windows 2000/XP 是一个完全受保护的操作系统,用户态 MS - DOS 应用程序不能直接访问硬件,而必须通

过一个真正的核心设备驱动程序。

- Win32 子系统的打印驱动程序将与设备无关的图形请求转换为打印机相关的命令,这些命令再发给核心模式的驱动程序,例如,并口驱动 (Parport.sys)、USB 打印机驱动 (Usbprint.sys) 等。

除了总线驱动、功能驱动、过滤器驱动外,硬件支持驱动程序可以分为以下类型:

- 类驱动程序 (classdriver) 为某一类设备执行 I/O 处理,例如磁盘、磁带或光盘。
- 端口驱动程序 (portdriver) 实现了对特定于某一种类型的 I/O 端口的 I/O 请求的处理,例如 SCSI。
- 小端口驱动程序把对端口类型的一般的 I/O 请求映射到适配器类型。例如,一个特定的 SLCI 适配器。

图 5-15 给出了一个例子,用以说明这些设备驱动程序是如何工作的。文件系统驱动

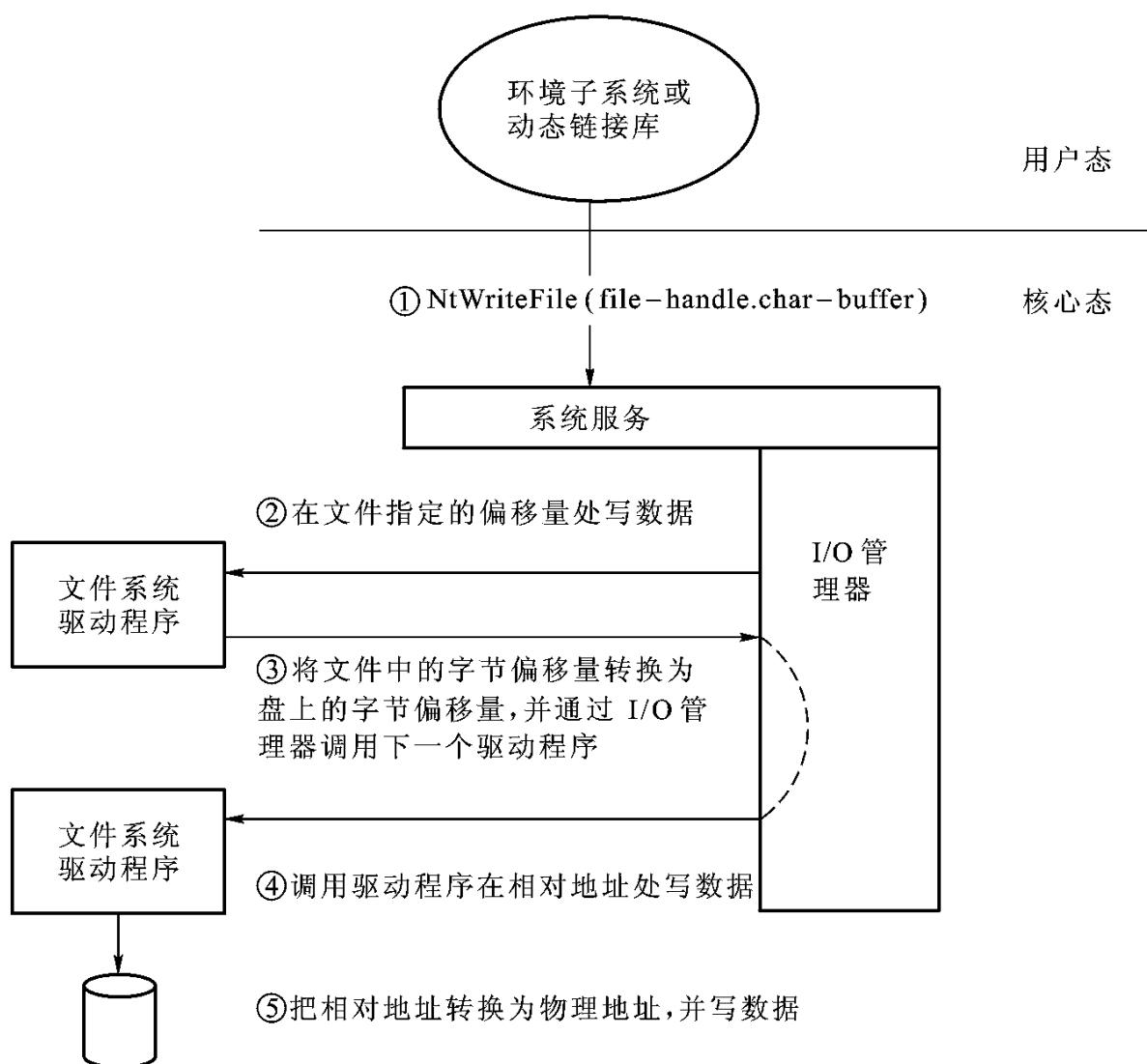


图 5-15 文件系统驱动和磁盘驱动的层次

程序收到向特定文件写数据的请求,它将此请求转换为向磁盘指定的“逻辑”位置写字节的请求,然后,把这个请求传递给一个简单的磁盘驱动程序。这个磁盘驱动程序再依次把请求转换为磁盘上的柱面/磁道/扇区,并且操作磁头来重现数据。

图 5-15 说明了在两层驱动程序之间的工作分界线。I/O 管理器接受了与一个特殊文件的开始部分有关的写请求,并将这个请求传递到文件系统驱动程序,这个驱动程序再把写操作从与文件有关的操作转换为开始位置(磁盘上一个扇区的边界)和要读取的字节数。文件系统驱动程序调用 I/O 管理器把请求传递到磁盘驱动程序,这个驱动程序将请求转换为物理磁盘位置,并且传递数据。

因为所有的驱动程序(包括设备驱动程序和文件系统驱动程序)对于操作系统来说都呈现相同的结构。一个驱动程序可以不经过转换当前的驱动程序或 I/O 系统,就能容易地被插入到分层结构中。例如,通过添加驱动程序,可以使几个磁盘看起来很像非常大的单一的磁盘。在 Windows 2000/XP 中实际上就存在这样一个驱动程序来提供容错磁盘支持。

1. 驱动程序结构

设备驱动程序包括一组调用处理 I/O 请求不同阶段的例程。如图 5-16 所示,主要有六个设备驱动程序例程:

- 初始化例程 当 I/O 管理器把驱动程序加载到操作系统中时,它执行驱动程序的初始化例程,这个例程将创建系统对象。I/O 管理器利用这些系统对象去识别和访问驱动程序。

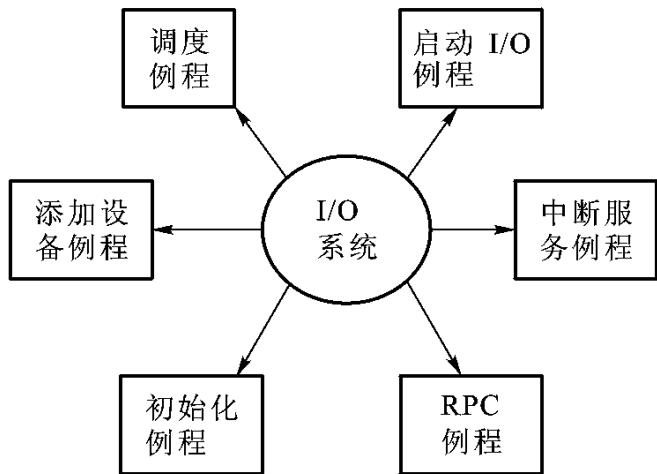


图 5-16 主要的设备驱动例程

- 添加设备例程 用于支持 PnP 管理器的操作。
- 一系列调度例程 调度例程是设备驱动程序提供的主要函数。例如,打开、关闭、读取、写入以及设备、文件系统或网络支持的任何其他功能。当被调用去执行一个 I/O 操作

时, I/O 管理器产生一个 IRP, 并且通过某个驱动程序的调度例程调用驱动程序。

- 启动 I/O 例程 驱动程序可以使用启动 I/O 例程来初始化与设备之间的数据传输。
- 中断服务例程 (ISR) 当一个设备中断时, 内核的中断调度程序把控制转交给这个例程。在 Windows 2000/XP 的 I/O 模型中, ISR 运行在高级的设备中断请求级 (IRQL) 上, 所以, 它们越简单越好, 以避免对低优先级中断产生不希望的阻塞。ISR 将运行在低 IRQL 的延迟过程调用 (DPC) 队列上, 以执行中断处理的剩余部分 (只有用于中断驱动设备的驱动程序才有 ISR, 例如, 文件系统就没有 ISR)。

• 中断服务 DPC 例程 DPC 例程执行在 ISR 执行以后的大部分设备中断处理工作。DPC 例程在低于 ISR 的 IRQ 的时候执行, 从而, 避免对其他中断产生不希望的阻塞。DPC 例程初始化 I/O 完成并启动关于设备的下一个队列的 I/O 操作。

尽管下面的例程没有列出, 但是可以在很多类型的设备驱动程序中找到。

- 一个或多个完成例程 分层驱动程序可能会有完成例程, 通过它一个较低层的驱动程序确定何时完成对一个 IRP 的处理。例如, 当设备驱动程序完成了与文件的数据传输以后, I/O 管理器将调用文件系统的完成例程。该完成例程通知文件系统关于操作的成功、失败或取消, 并且允许文件系统执行清理操作。
- 取消 I/O 例程 如果某个 I/O 操作可以被取消, 驱动程序就可以定义一个或多个取消 I/O 例程。I/O 管理器调用什么样的取消例程, 取决于 I/O 操作在被取消时已进行到什么程度。

• 卸载例程 卸载例程释放任何驱动程序正在使用的系统资源, 以使 I/O 管理器能从内存中删除它们。当系统运行时, 驱动程序可以被加载或卸载。

- 系统关闭通知例程 这个例程允许驱动程序在系统关闭时做清理工作。
- 错误记录例程 当意外错误发生时 (例如, 当磁盘分区被损坏时), 驱动程序的错误记录例程将记录发生的事情, 并通知 I/O 管理器。I/O 管理器把这个信息写入错误记录文件。

2. 同步

驱动程序必须同步执行它们对全局驱动程序数据的访问, 这有两个主要原因:

- 驱动程序的执行可以被高优先级的线程抢先, 或时间片 (或时间段) 到时被中断, 或被其他中断所中断。

- 在多处理器系统中, Windows 2000/XP 能够同时在多个处理器上运行驱动程序代码。

若不能同步执行, 就会导致相应错误的发生。例如, 因为设备驱动程序代码运行在低优先级的 IRQL 上, 所以, 当调用者初始化一个 I/O 操作时, 可能被设备中断请求所中断, 从而导致在它的设备驱动程序正在运行时让设备驱动程序的 ISR 去执行。如果设备驱动程序正在修改其 ISR 也要修改的数据 (例如, 设备寄存器、堆存储器或静态数据) 则在 ISR 执行时,

数据可能被破坏。

要避免这种情况发生,为 Windows 2000/XP 编写的设备驱动程序就必须对它和它的 ISR 对共享数据的访问进行同步控制。在尝试更新共享数据之前,设备驱动程序必须锁定所有其他的线程(或 CPU,在多处理器系统的情况下),以防止它们修改同一个数据结构。

当设备驱动程序访问其 ISR 也要访问的数据时,Windows 2000/XP 的内核提供了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 ISR 的执行。在单 CPU 系统中,在更新一个结构之前,这些例程将 IRQL 提高到一个指定的级别。然而,在多处理器系统中,因为一个驱动程序能同时在两个或两个以上的处理器上执行,以这种技术就不足以阻止其他的访问。因此,另一种被称为“自旋锁”的机制被用来锁定来自指定 CPU 的独占访问的结构。

到目前为止,应该意识到尽管 ISR 需要特别的关注,但一个设备驱动程序使用的任何数据都将面临运行于另一个处理器上的相同的设备驱动程序的访问。因此,用设备驱动程序代码来同步它对所有全局的或共享数据(或任何到物理设备本身的访问)的使用是很危险的。如果数据被 ISR 使用,设备驱动程序就必须使用内核同步例程或者使用一个内核锁。

5.8.4 Windows 2000/XP I/O 处理

在了解了驱动程序的结构和类型以及支持该结构和类型的数据结构之后,现在来看 I/O 请求是如何在系统中传递的。一个 I/O 请求会经过若干个处理阶段,而且根据请求是指向由单层驱动程序操作的设备还是一个经过多层驱动程序才能到达的设备,它经过的阶段也有所不同。处理的不同进一步依赖于调用者是制定了同步 I/O 还是异步 I/O,所以,先了解一下这两种 I/O 类型的处理以及其他几种不同类型的 I/O。

1. I/O 的类型

应用程序在发出 I/O 请求时可以设置不同的选项,例如,设置同步 I/O 或者异步 I/O,设置应用程序获取 I/O 数据的方式等。

(1) 同步 I/O 和异步 I/O

应用程序发出的大多数 I/O 操作都是“同步”的,也就是说,设备执行数据传输并在 I/O 完成时返回一个状态码,然后,程序就可以立即访问被传输的数据。ReadFile 和 WriteFile 函数使用最简单的形式调用时是同步执行的,在把控制返回给调用程序之前,它们完成一个 I/O 操作。

“异步 I/O”允许应用程序发布 I/O 请求,然后当设备传输数据的同时,应用程序继续执行。这类 I/O 能够提高应用程序的吞吐率,因为,它允许在 I/O 操作进行期间,应用程序继续其他的工作。要使用异步 I/O,必须在 Win32 的 CreateFile 函数中指定 FILE_FLAG_

OVERLAPPED 标志。当然,在发出异步 I/O 操作请求之后,线程必须小心地不访问任何来自 I/O 操作的数据,直到设备驱动程序完成数据传输。线程必须通过等待一些同步对象(无论是事件对象、I/O 完成端口或文件对象本身)的句柄,使它的执行与 I/O 请求的完成相同步。当 I/O 完成时,这些同步对象将会变成有信号状态。

与 I/O 请求的类型无关,由 IRP 代表的内部 I/O 操作都将被异步执行,也就是说,一旦一个 I/O 请求已经被启动,设备驱动程序就返回 I/O 系统。I/O 系统是否返回调用程序取决于文件是否为异步 I/O 打开的。可以使用 Win32 的 HasOverlappedToCompleted 函数去测试挂起的异步 I/O 的状态。

(2) 快速 I/O

快速 I/O 是一个特殊的机制,它允许 I/O 系统不产生 IRP 而直接到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

映射文件 I/O 是 I/O 系统的一个重要特性,是 I/O 系统和内存管理器共同实现的。映射文件 I/O 是指把磁盘中的文件视为进程的虚拟内存的一部分。程序可以把文件作为一个大的数组来访问,而无需做缓冲数据或执行磁盘 I/O 的工作。程序访问内存,同时内存管理器利用它的页面调度机制从磁盘文件中加载正确的页面。如果应用程序向它的虚拟地址空间写入数据,内存管理器就把更改作为正常页面调度的一部分写回到文件中。

通过使用 Win32 的 CreateFileMapping 和 MapViewOfFile 函数,映射文件 I/O 对于用户态是可用的。在操作系统中,映射文件 I/O 被用于重要的操作中,例如,文件高速缓存和映像活动(加载并运行可执行程序)。其他重要的使用映射文件 I/O 的程序还有高速缓存管理器。文件系统使用高速缓存管理在虚拟内存中的映像文件数据,从而,为 I/O 绑定程序提供了更快的响应时间。当调用者使用文件时,内存管理器将把被访问的页面调入内存。尽管多数高速缓存系统在内存中分配固定数量的字节给高速缓存文件,但 Windows 2000/XP 高速缓存的增大或缩小取决于可以获得的内存有多少。这种大小的变化是可能的,因为,高速缓存管理器依赖于内存管理器来自动地扩充(或缩小)高速缓存的数量,它使用正常工作集机制来实现这一功能。通过利用内存管理器的页面调度系统,高速缓存避免了重复内存管理器已经执行了的工作。

(4) 分散/集中 I/O

Windows 2000/XP 同样支持一种特殊类型的高性能 I/O,它被称做“分散/集中”(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一个读取或写入操作,从虚拟内存的多个缓冲区读取数据并写到磁盘上文件的一个连续区域里。要使用分散/集中 I/O,文件必须以非高速缓存 I/O 方式打开,被使用的用户缓冲区必须是页对齐的,并且 I/O 必须被异步执行。

2. 对单层驱动程序的 I/O 请求

单层核心态设备驱动程序的同步 I/O 请求处理包括以下六步：

- I/O 请求经过子系统 DLL。
- 子系统 DLL 调用 I/O 管理器的 NtWriteFile 服务。
- I/O 管理器以 IRP 的形式给设备驱动程序发送请求。
- 驱动程序启动 I/O 操作。
- 在设备完成了操作并且中断 CPU 时, 设备驱动程序服务于中断。
- I/O 管理器完成 I/O 请求。

下面进一步看看中断处理和 I/O 完成。

(1) 处理一个中断

在 I/O 设备完成数据传输之后, 它将产生中断并请求服务, 这样 Windows 2000/XP 的内核、I/O 管理器和设备驱动程序都将被调用。

当 I/O 设备中断发生时, 处理器将控制转交给内核陷阱处理程序, 内核陷阱处理程序将在它的中断向量表中搜索定位用于设备的 ISR。Windows 2000/XP 上的 ISR 用两个步骤来典型地处理设备中断。当 ISR 被首次调用时, 它通常只在设备 IRQL 上停留获得设备状态所必需的一段时间, 最后停止设备的中断。然后, 它使一个 DPC 排入队列, 并退出服务例程, 清除中断。过一段时间, 在 DPC 例程被调用时, 设备完成对中断的处理。完成之后, 设备将调用 I/O 管理器来完成 I/O 并处理 IRP。它也可以启动下一个正在设备队列中等待的 I/O 请求。

使用 DPC 来执行大多数设备服务的优点是, 任何优先级位于设备 IRQL 和 Dispatch/DPC IRQL 之间被阻塞的中断允许在低优先级的 DPC 处理发生之前发生。因而, 中间优先级的中断就可以更快地得到服务。

(2) 完成 I/O 请求

当设备驱动程序的 DPC 例程执行完以后, 在 I/O 请求可以考虑结束之前还有一些工作要做。I/O 处理的第三阶段称做“I/O 完成”(I/O completion), 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”(I/O status block)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 系统必须把一些存储在系统内存中的数据复制到调用者的虚拟地址空间中。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, 而此时调用者进程是当前处理器上活动的进程, 调用者线程正在处理器上执行。I/O 管理器通过在线程中执行一个核心态的异步过程调用(APC)来完成这个操作。

APC 在特定线程的描述表中执行, 而 DPC 在任意线程的描述表中执行, 这就意味着 DPC 例程不能涉及用户态进程的地址空间。要注意的是, DPC 具有比 APC 更高的软件中断

优先级。

接下来当线程开始在较低的 IRQL 上执行时, 挂起的 APC 被提交运行。内核把控制权转交给 I/O 管理器的 APC 例程, 它将把数据(如果有)和返回的状态复制到最初调用者的地址空间, 释放代表 I/O 操作的 IRP, 并将调用者的文件句柄(或调用者提供的事件或 I/O 完成端口)设置为有信号状态。现在才可以考虑完成 I/O, 在文件(或其他对象)句柄上等待的最初调用者或其他的线程都将从它们的等待状态中被唤醒并准备再一次执行。

关于 I/O 完成最后要注意的是: 异步 I/O 函数 ReadFileEx 允许调用者提供用户态 APC 作为参数。如果调用者这样做了, I/O 管理器将在 I/O 完成的最后一步为调用者清除这个 APC, 这个特性允许调用者在 I/O 请求完成时指定一个将被调用的子程序。正如在 Platform SDK 文档中对这些函数解释的那样, 用户态 APC 完成例程在请求线程的描述表中执行, 并且只有当线程进入可报警等待状态时才可以被传送。

5.8.5 Windows 2000/XP 高速缓存管理

1. Windows 2000/XP 高速缓存管理器的主要特征

Windows 2000/XP 高速缓存管理器是一组核心态的函数和系统线程, 它们与内存管理器一起为所有 Windows 2000/XP 文件系统驱动程序提供数据高速缓存(包括本地与网络)。下面讨论 Windows 2000/XP 高速缓存管理器, 包括它的关键内部数据结构和函数是如何工作的, 在系统初始化时如何设置, 与操作系统中其他元素如何相互作用, 以及在 Win32 CreateFile 中影响文件高速缓存的五个标志。

Windows 2000/XP 高速缓存管理器提供了一种高速、智能的机制, 用以减少磁盘 I/O 和增加系统的整体吞吐量。基于虚拟块的高速缓存使 Windows 2000/XP 高速缓存管理器能够进行智能预读。依靠全局内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了特殊的快速 I/O 机制减少了用于读写操作的时间, 而且将与物理内存有关的管理工作交给了 Windows 2000/XP 全局内存管理器, 这样减少了代码的冗余, 提高了效率。

Windows 2000/XP 高速缓存管理器的主要特征:

(1) 单一集中式系统高速缓存

Windows 2000/XP 提供了一个集中的高速缓存工具来缓存所有的外部存储数据, 包括在本地硬盘、软盘、网络文件服务器或是 CD - ROM 上的数据。任何数据都能被高速缓存, 无论它是用户数据流(文件内容和在这个文件上正在进行读和写的活动)或是文件系统的元数据(metadata)(例如目录和文件头)。Windows 2000/XP 访问缓存的方法是由被缓存的数据的类型所决定的。

(2) 与内存管理器结合

Windows 2000/XP 高速缓存管理器不知道多少数据存在物理内存, 因为, 它采用将文件视图映射到系统虚拟空间的方法访问数据, 在这过程中使用了标准区域对象 (section object)。访问位于映射视图中的地址时, 内存管理器不在物理内存的逻辑块中分配页面。以后需要内存时, 内存管理器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 (1RP)。取而代之, 它仅仅在内存和被缓存的文件部分所被映射的虚拟地址之间拷贝数据, 并依靠内存管理器去处理换页。这种设计使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

高速缓存管理器一个重要的功能是保证任何访问高速缓存数据的进程都可得到这些数据的最新版本。当进程打开一个文件 (这个文件被缓存了) 而另一个进程直接将文件映射到它的地址空间 (运用 Win32 MapViewOfFile 函数), 问题就产生了。这种潜在的问题不会在 Windows 2000/XP 中出现, 因为, 高速缓存管理器和用户应用程序使用相同的内存管理文件映射服务将文件映射到它们的地址空间。而内存管理器保证每一个被映射文件只有惟一的版本, 它映射文件的所有视图到物理内存页面的单独集合, 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 正在做的任何改变, 进程 2 都可以看到, 而不用等到这些改变被回写。

(4) 虚拟块缓存

大多数操作系统高速缓存管理器 (包括 NetWare, OpenVMS, OS/2 和老的 UNIX 系统) 基于磁盘逻辑块 (logical block) 缓存数据。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 2000/XP 高速缓存管理器用一种虚拟块缓存 (virtual block caching) 方式, 管理器对缓存中文件的某些部分进行追踪。通过内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。这种方式有以下几个主要特点:

- 它使智能的文件预读成为可能。因为, 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。
- 它允许 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)。因为, 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满足 I/O 的需要, 而不调用文件系统。

(5) 基于流的缓存

Windows 2000/XP 高速缓存管理器与文件缓存相对应也设计了字节流的缓存。一个流是指在文件内的字节序列。一些文件系统, 像 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一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 2000/XP 高速缓存管理器被认为是高速缓存文件, 但它实际上缓存的是字节流。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 (recoverablefile 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, 或在系统重起动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。为了避免这个问题, 在改变卷之前, 可恢复文件系统会维护一个日志文件 (log file)。在每一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, 该日志文件进行记录。如果因系统失败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:

- 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。
- 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。
- 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的元数据。
- 高速缓存管理器将被更改的元数据刷新到磁盘上, 更新卷结构。

2. 高速缓存的结构

Windows 2000/XP 系统高速缓存管理器基于虚拟空间缓存数据, 所以, 它管理一块系统虚拟地址空间区域, 而不是一块物理内存区域。高速缓存管理器把每个地址空间区域分成 256 KB 的槽 (slot), 被称为视图 (view)。

文件第一次 I/O (读或写) 操作时, 高速缓存管理器将文件中包含被请求数据的 256 KB 对齐的区域映射为一个 256 KB 视图, 放入到系统缓存空间的一个空闲槽内。例如, 如果从偏移量为 300000 字节区域处开始读入 10 字节数据, 被映射的视图将在偏移量 262 144 处开始 (文件第二个 256 KB 对齐区域) 容量为 256 KB。

高速缓存管理器在文件视图和缓存地址空间的槽之间循环进行映射, 将所请求的第一个视图映射到第一个 256 KB 槽中, 再将第二个视图映射到第二个 256 KB 槽中, 以此类推, 如图 5-17 所示。在这个例子中文件 B 最先被映射, 文件 A 其次, 文件 C 再次, 所以文件 B 被映射的块占据高速缓存的第一个槽。注意文件 B 中只有第一个 256 KB 部分被映射, 这是由于这个文件只有该部分被访问。虽然文件 C 只有 100 KB (比系统缓存视图要小), 但它仍在缓存中占据独立的 256 KB 槽。

高速缓存管理器只映射活跃的视图。然而只有在读或写文件操作时, 视图被标记为活

跃。除非进程用带有 FILE - FLAG - RANDOM - ACCESS 标志的 CreatFile 函数打开一个文件,否则高速缓存管理器在映射文件新视图时,不映射那些未被激活的文件视图。

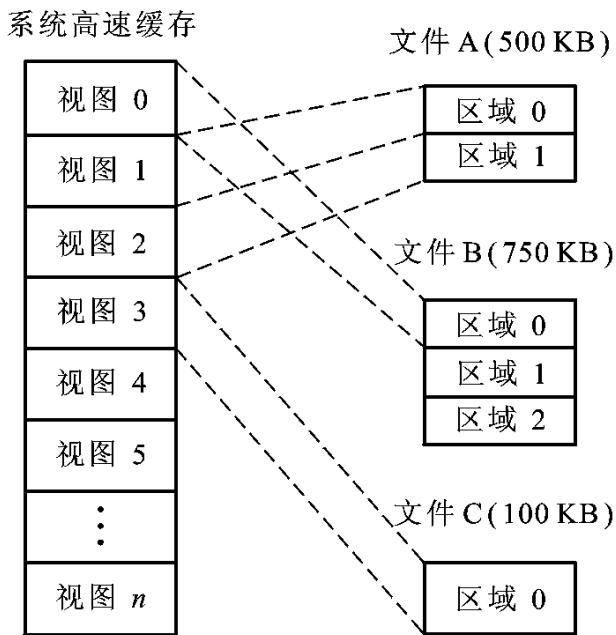


图 5-17 映射到系统高速缓存中的不同大小文件

当高速缓存管理器需要映射一个文件视图但缓存内没有空余的槽时,它将取消最近一个未激活映射视图,并使用这个槽。如果没有视图可用,则返回一个 I/O 错误,说明没有足够的系统资源来完成操作。然而,由于只有在进行读或写操作的视图才被激活,上述情况只有在成千上万的文件被同时访问时才会发生。

3. 高速缓存的大小

Windows 2000/XP 是如何计算系统高速缓存的大小(包括虚拟与物理的高速缓存)的呢?正如大多数与内存管理相关的计算一样,系统缓存的大小依赖于包括内存大小和运行的 Windows 2000/XP 版本等因素。

1) 缓存区的虚拟大小

系统高速缓存虚拟大小是已安装物理内存总量的函数,默认大小为 64 MB。如果系统物理内存多于 4 032 页(16 MB),缓存大小设定为以 128 MB 为基础,物理内存每比 16 MB 多 4 MB,则增加 64 MB 缓存区。利用这种算法,有 64 MB 物理内存的计算机系统虚拟缓存将是: $128\text{ MB} + (64\text{ MB} - 16\text{ MB}) / 4\text{ MB} \times 64\text{ MB} = 896\text{ MB}$ 。

对于 x86 2 GB 系统空间,系统缓存的最小(MmSystemCacheStart)和最大(MmSystemCacheEnd)的虚拟容量为 64/960 MB(开始与结束地址为 0xC1000000 – E0FFFFFF)。如果系统计算出虚拟大小大于 512 MB,缓存就被赋予额外的地址区域,称为缓存附加内存。

2) 缓存的物理大小

Windows 2000/XP 的高速缓存与其他操作系统设计上最大不同是由全局内存管理器来管理物理内存。正因为这样,用来处理工作集的扩展和收缩、管理已修改和未修改链表的代码也被用来控制系统缓存的大小,并动态地平衡进程和操作系统间对物理内存的需求。

系统高速缓存没有自己的工作集,而是与高速缓存数据、页缓冲池、可分页的核心代码以及可分页的驱动程序代码共用一个系统工作集。尽管系统高速缓存只是这个工作集的一个组成部分,在系统内部将它称为“系统高速缓存工作集”(简称为系统工作集)。

通过观察性能计数器和系统变量,可以对比系统高速缓存的物理大小与整个系统工作集的物理大小,同样,也能看到系统工作集上的缺页信息。

4. 高速缓存的数据结构

高速缓存管理器利用下面的数据结构跟踪被缓存的文件:

- 在系统高速缓存的每个 256 KB 的槽由一个 VACB 描述。
- 每个打开的被缓存文件有一个专用的缓存映射,它包含了用于控制文件预读的信息。
- 每个被缓存的文件有一个单独的共享缓存映射结构,它指向系统缓存中包含此文件映射视图的槽。

这些结构及它们的关系将在下一节描述。

(1) 系统范围的高速缓存数据结构

高速缓存管理器通过用虚拟地址控制块 VACB (virtualaddresscontrolblock) 在系统缓存中追踪视图的状态。在系统初始化期间,高速缓存管理器分配一个单独的未分页的内存区域来保存所有用来描述系统高速缓存的 VACB。在变量 CcVacbs 中存储 VACB 数组的地址。每个 VACB 代表系统高速缓存中一个 256 KB 的视图,VACB 结构如图 5-18 所示。

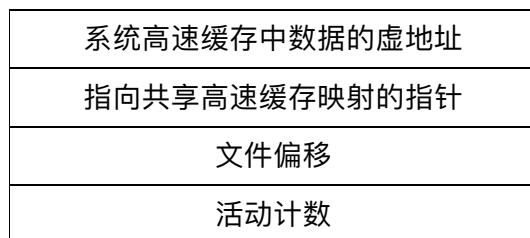


图 5-18 VACB 结构

从图中可以看到,VACB 中的第一个字段存放系统高速缓存中数据的虚拟地址。第二个字段存放被共享缓存映射结构的指针,它指出哪一个文件被缓存。第三个字段标识视图起始处在文件内的偏移(通常在 256 KB 间隔基础上建立)。最后一个字段,VACB 包括引用视图的数目,即有多少读或写操作正在访问该视图。在一个文件进行 I/O 操作时,文件的

VACB 引用数目增加,当 I/O 操作结束时减少。对文件系统的元数据而言,活动数目表示有多少文件系统驱动程序在该视图中有被上锁的内存页面。

(2) 单文件的缓存区数据结构

每个打开的文件句柄都有一个相应的文件对象。如果文件被缓存,文件对象指向一个私有的高速缓存映射结构,它包括最后两次读操作的位置以便高速缓存管理器能完成智能预读任务;另外,文件对象所有私有的缓存映射结构都被链接在一起。

每个被缓存的文件都有一个共享缓存映射结构,它描述被映射文件的状态,包括文件大小和有效数据长度、共享高速缓存映射指向的区域对象(由内存管理器维护。描述文件映射到虚拟内存)、与该文件相关的私有高速缓存映射链表和在系统高速缓存中描述当前文件映射视图的所有 VACB。图 5-19 说明了每个文件的缓存区数据结构之间的关系。

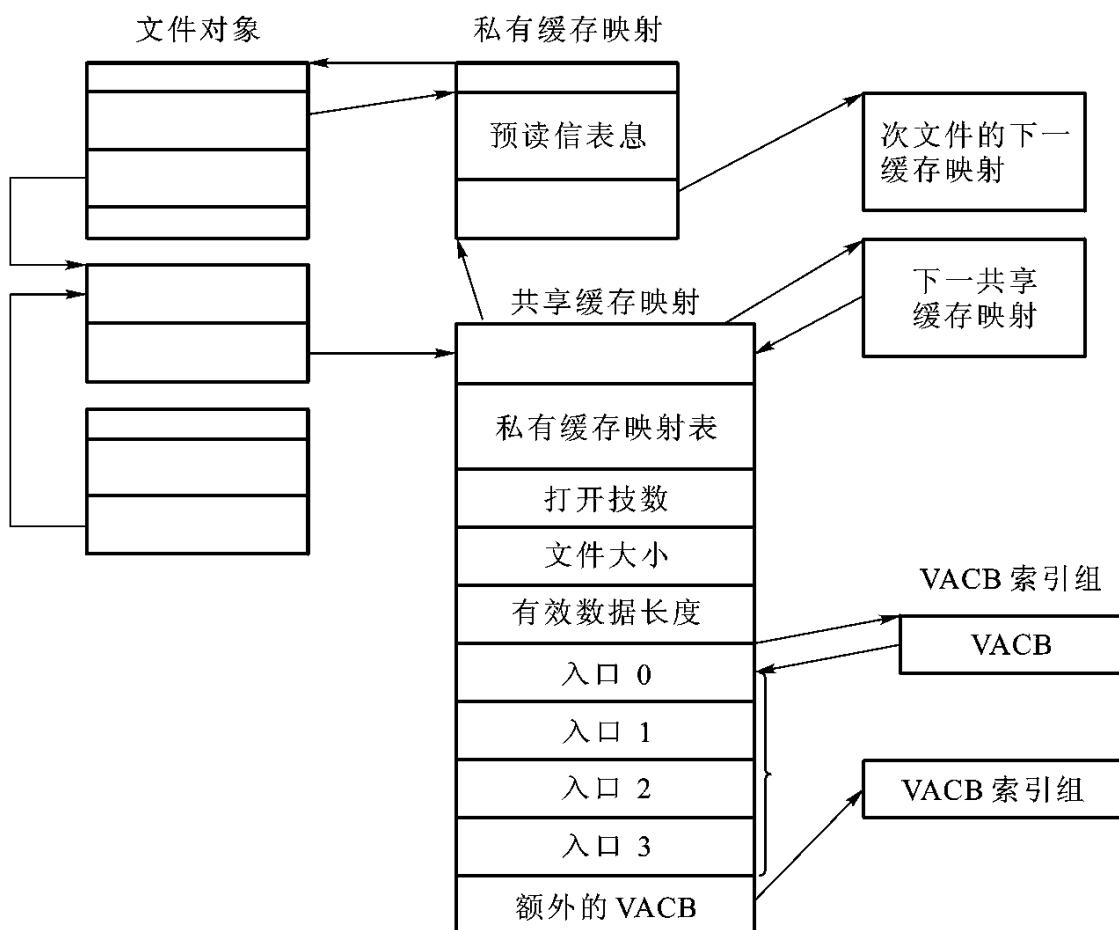


图 5-19 单文件高速缓存数据结构

当请求从一个指定的文进行读取时,高速缓存管理器必须回答两个问题:这个文件在缓存里吗? 如果已经存在,哪一个 VACB 指向被请求的位置?

换句话说,高速缓存管理器必须查明所请求地址上的文件的视图是否被映射到了系统高速缓存。如果没有 VACB 包含所需的文件偏移量,则说明数据没有映射到系统缓存中。

为了追踪给定文件有那些视图被映射到了系统高速缓存中,高速缓存管理器维护了一个指向 VACB 的指针数组,称为“VACB 指针数组”,第一个 VACB 指针数组入口是指文件的第一个 256 KB,第二个入口指第二个 256 KB,以此类推。图 5-20 显示了三个被映射到系统缓存中的不同文件的四个不同部分的情况。

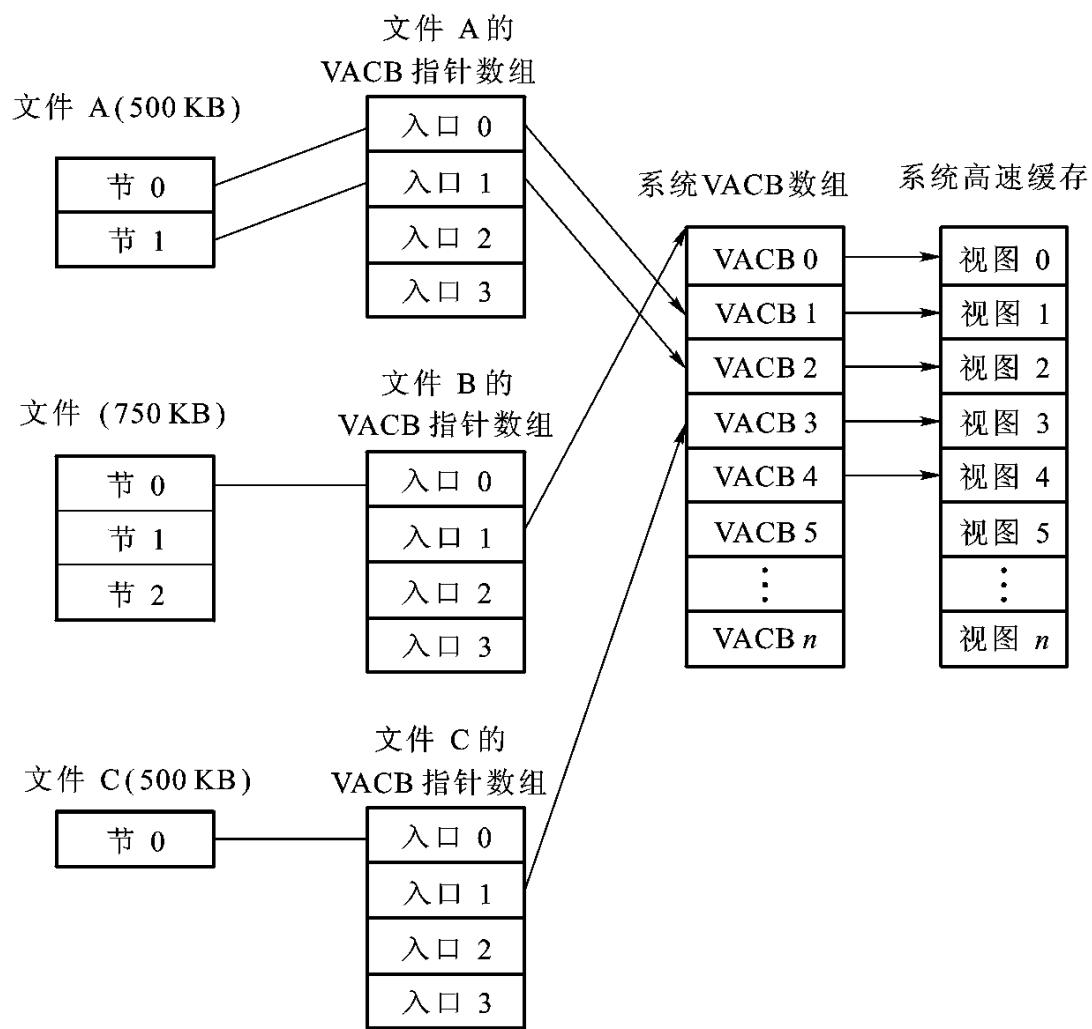


图 5-20 VACB 指针数组

当进程访问一个文件的指定位置时,高速缓存管理器在文件的 VACB 指针数组中查找相应的入口,看被请求的数据是否已经被映射到了缓存中。如果数组的那个入口不为空(因此,含有一个指向 VACB 的指针),则文件中的那一部分已经在缓存中。VACB 会依次指向文件视图被映射到系统高速缓存中的位置。如果该项为空,则高速缓存管理器必须在系统缓存区中找到一个空闲的槽和一个空闲的 VACB 来映射所请求的视图。

出于空间优化的考虑,共享高速缓存映射包含一个有 4 项的 VACB 指针数组。因为每个 VACB 描述 256 KB,这个固定大小的数组最多可以描述 1 MB 的文件。如果文件超过 1MB,则从未分页内存池中分配一个单独的 VACB 指针数组,该数组的大小等于文件的大小。

除以 256 KB(如果有余数,商加 1)。然后,共享高速缓存映射指向这个独立的结构。

为进一步进行优化,如果文件超过 32 GB 大小,从未分页内存池中分配的 VACB 指针数组组成多维指针的稀疏数组(sparse multilevel index array),这时每个指针数组含有 128 项。用下面的公式,可以计算出文件的指针数组需要多少层:

$$(表示文件大小所需的二进制位数 - 18) / 7$$

将公式的结果取整。公式中的 18 是由 VACB 代表 256 KB 而来的,256 KB 等于 2^{18} B。7 来源于数组中的每一级有 128 项,而 2^7 是 128。因此,一个大小为 2^{63} (高速缓存管理器支持的最大数)的文件只需要 7 层。数组之所以稀疏,是因为被高速缓存管理器分配的分支中只有最下层数组才指向 VACB。

这种方案处理稀疏文件很有效,这些文件很大但只含有少量所需数据,所以只需分配足够的数组处理当前文件视图就行了。例如,32 GB 的稀疏文件只有 256 KB 被映射到了高速缓存的虚拟地址空间。这只需分配一个 VACB 和 3 个指针数组,因为,数组只有一个分支被映射了。而这样一个 32 GB(2^{35})的文件只需一个 3 层的数组。如果高速缓存管理器对这个文件不采用多级 VACB 数组优化,它将不得不分配一个有 128 000 项的数组,或相当于分配 1000 个指针数组。

5. 高速缓存的操作

下的介绍高速缓存管理器怎样为文件系统驱动程序实现读写文件数据。记住,在文件 I/O 过程中仅当文件被打开时(例如,用 Win32 CreateFile 函数),高速缓存管理器才被激活。被映射的文件,以及用 FILE _ FLAG _ NO _ BUFFERING 标记打开的文件都不会经过高速缓存管理器。

(1) 回写缓存和延迟写

Windows 2000/XP 高速缓存管理器实现了一个带有延迟写(lazy writing)的回写(write-back)高速缓存,这意味着写入文件的数据首先被存储在高速缓存页面的内存中,然后,再被写入磁盘。因此。写操作允许在短时间内积累,并一次性刷新到磁盘,这可以减少磁盘的 I/O 次数。

高速缓存管理器必须显式地调用内存管理器去刷新高速缓存页面,如果不这样,内存管理器只在对物理内存的需求超过所提供的内存时,才将内存的内容写入磁盘,这种方式适合于反复变化的数据。然而,被缓存的文件数据也可能不是频繁变化的数据。如果一个进程修改了高速缓存的数据,用户期望将修改过的部分及时地反映到磁盘上。

确定高速缓存的刷新频率是十分重要的。如果高速缓存刷新过于频繁,系统将因为不必要的 I/O 操作而降低性能。如果高速缓存刷新过少,用户将面临在系统崩溃时失去已修改数据的风险,或将物理内存用光的危险(因为已修改的页面占用了过多的内存)。

为了平衡这些关系,由高速缓存管理器每秒钟产生一个系统线程——延迟写器——排

列系统缓存中 1/8 的“脏”(dirty)页(已修改页),并将其写入磁盘。如果脏页产生的频率大于延迟写器写入页面的频率,延迟写器将按脏页产生的频率计算出额外需要写入的页面数,将这些页面写入磁盘。

(2) 计算脏页阈值

脏页阈值(threshold)是系统唤醒延迟写系统线程将页面写回到磁盘之前,保存在内存中的系统高速缓存的页面数。该数值在系统初始化时计算,且依赖于物理内存大小和注册表项:HKLM\SYSTEM\CurrentControlSet\Control\SeSSiOnManager\MemoryManagement\LargeSystemCache。在 Windows 2000/XP Professional 中这个值缺省是 0,在 Windows 2000/XP Server 中缺省是 1。可以在 Windows 2000/XP Server 系统图形界面中通过修改文件服务属性来调整这个值。尽管这项服务也存在于 Windows 2000/XP Professional,但它的参数不可以调整。

脏页阈值的计算按系统内存容量是小、中、大,分别为物理页面数除 8、物理页面数除 4、上面两数值的和。当系统最大工作集的大小超过 4 MB 时(经常是这样),计算将被忽略,脏页阈值被设置为系统最大工作集大小减去 2 MB 的页数。

(3) 屏蔽对文件延迟写

在调用 Win32 CreateFile 函数时指定 FILE_ATTRIBUTE_TEMPORARY 标志创建一个临时文件,延迟写器就不会将脏页写回磁盘,除非物理内存严重不足或文件关闭。延迟写器的这种特性改善了系统性能——延迟写器不会立即将最终可能丢弃的数据写入磁盘。应用程序经常在关闭了临时文件后将其删去。

(4) 强制写缓存到磁盘

由于一些应用程序不允许在向磁盘写文件和查看磁盘数据更新之间出现即使很短的延迟,所以高速缓存管理器也支持基于单个文件的通写高速缓存,即数据一经改变被立即写入磁盘。要启动通写高速缓存,需要在调用 CreateFile 函数时设置 FILE_FLAG_WRITE_THROUGH 标志。作为另一种选择,当一个线程需要把数据写入磁盘时,可以使用 Win32FlushFileBuffers 函数显式地刷新一个打开的文件。

(5) 刷新被映射的文件

如果延迟写器必须从映射到其他进程地址空间的视图向磁盘写入数据,情况就有些复杂,高速缓存管理器仅知道它修改过的页面(被其他进程修改的页面只有该进程知道,因为,被修改页在页表项中的修改标志被保存在进程的私有页表中)。为了处理这种情况,当用户映射一个文件时,内存管理器就会通知高速缓存管理器。当该文件在高速缓存内被刷新时(例如,调用了 Win32FlushFileBuffers 函数),高速缓存管理器将缓存中的脏页写入磁盘,然后,检查文件是否被其他进程映射。如果文件也被其他进程映射了,那么,高速缓存管理器把文件区域所对应的整个视图刷新一遍,以便将第二个进程可能改变的页面写入磁盘。如果用户映射了一个也在高速缓存中打开的视图,当该视图被取消映射时,修改过的页被标记

为“脏”以便延迟写线程将来刷新该视图时,将这些脏页写入磁盘。这些过程只有按下列次序进行才能正常起作用:

- 用户取消了视图的映射;
- 进程刷新文件缓冲区。

如果没有遵守这个次序,则无法预测哪些页面会被写入磁盘。

(6) 智能预读

Windows 2000/XP 高速缓存管理器运用空间局部性原理,基于进程当前所读取数据预测其下一步可能读的数据,从而,实现智能预读(intelligent read – ahead)。因为系统缓存是以虚拟地址为基础,而虚拟地址对于一个文件而言是连续的,它们在物理内存中是否连续并不重要。基于逻辑块的高速缓存系统是以磁盘上被访问的数据的相对位置为基础,而文件未必连续存储在磁盘上。所以,对于逻辑块高速缓存的文件预读会更复杂,而且需要文件系统驱动程序和逻辑块高速缓存的紧密配合。

预读有两种类型——虚拟地址预读和带历史信息的异步预读,利用 Cache:ReadAheads/Sec 性能计数器或者 CcReadAheadlos 系统变量可以检查预读的活动。

(7) 虚拟地址预读

当内存管理解决缺页时,它会将被访问页面相近的几个页一起读到内存中,这种方法叫做簇。对于顺序读的应用程序,这种虚拟地址预读(virtual address read – ahead)操作减少了获取数据所需的磁盘读操作次数。内存管理器的这种方法惟一缺点是:由于这种预读方式是在处理缺页的上下文中进行的,所以,它必须同步进行,此时等待页面数据的线程必须处在等待状态。

(8) 带历史信息的异步预读

由内存管理器进行的虚拟地址预读提升了系统的 I/O 性能,但是它只对顺序访问的数据有利。为了将预读的好处扩展到特定的随机访问数据中,高速缓存管理器在文件的私有缓存映射结构中为正在被访问的文件句柄保存最后两次读请求的历史信息,这种方法被称为“带历史信息的异步预读”(asynchronous read – ahead with history)。如果能从调用者明显的随机读取中确定一种模式,高速缓存管理器将这种模式延伸。例如,如果调用者读了第 4000 页,然后是第 3000 页,高速缓存管理器假设调用值下一个请求的页面会是第 2000 页,于是预读第 2000 页。

为了提高预读效率,Win32 CreateFile 函数提供了一个表示顺序文件访问的标志:FILE _ FLAG _ SEQUENTIAL _ SCAN。如果设置了这个标志,高速缓存管理器不会为调用者保存用于预测的历史记录,而是进行顺序预读。由于文件是被读入了高速缓存的工作集,高速缓存管理器取消映射不再活跃的文件视图时,通知内存管理器将属于这些视图的页面放到备用链表或修改链表中(如果页面被修改过),以便它们可以被再次使用。对每次读取的 I/O 操

作, 高速缓存管理器预读的数据是原来的三倍(例如读 192 KB 代替 64 KB)。随着调用者继续读, 高速缓存管理器向前预读附加的数据块, 始终保持着比调用者提前读入一个读操作(当前读取数据的大小)。

高速缓存管理器的预读是异步的, 由于执行预读的线程与读入数据的线程不同, 两者可以同时执行。当请求读取被缓存的数据时, 高速缓存管理器首先访问被请求的虚页面, 并完成这次请求, 然后, 向系统工作线程提出另一次 I/O 请求, 取得额外的数据。接下来, 系统工作线程在后台执行, 读入很可能调用者下一次将请求的数据。当程序继续执行时, 预先读的页面已被调入内存, 所以, 调用者再次请求数据时它已经在内存中了。

尽管, 带历史信息的异步预读技术比普通预读使用了更多的内存, 但它大大改善了程序读大量顺序缓存的数据的性能。Cache: ReadAheads/Sec 性能计数器显示了顺序访问预读的操作。

对于无法预测读取模式的应用程序, 可以在调用 Win32 CreateFile 函数时设置 FILE_FLAG_RANDOM_ACCESS 标志。这个标志通知高速缓存管理器不要试图预测应用程序下一步要读取的位置, 这样就关掉了预读功能。这个标志项还阻止了高速缓存管理器在文件被访问时强制取消其视图映射, 这样程序再次访问文件某个部分时减少了映射/取消映射的活动。

(9) 系统线程

高速缓存管理器通过向公共临界系统工作线程池发送请求来实现延迟写和预读的 I/O 操作。然而, 可供使用的线程数有限制, 对于小型和中型内存的系统, 数目比临界工作系统线程的总数少一个(大内存系统少两个)。在内部, 高速缓存管理器将它的工作请求组织到两张表中(尽管是同一组工作线程为这些表服务):

- 用于预读操作的快速队列;
- 用于延迟写扫描(刷新脏页数据)、后台写和延迟关闭的常规队列。

为了追踪工作线程需要进行的工作项目, 高速缓存管理器创建了自己内部的处理器后备链表。每个处理器有一个定长的包含工作队列项目结构的表。工作队列项目的数量取决于系统大小:小内存系统为 32, 中内存系统为 64, 大内存 Windows 2000/XP Professional 系统为 128, 大内存 Windows 2000/XP Server 系统为 256。

(10) 快速 I/O

任何时候只要有可能, 读写被缓存文件可以用被称为快速 I/O (fast I/O) 的高速机制来处理。快速 I/O 读写一个缓存的文件不需要产生 I/O 请求包(IRP)。有了快速 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的数据, 而不需产生 IRP。

由于 Windows 2000/XP 高速缓存管理器能够追踪哪些文件的哪些块在高速缓存中, 所

以文件系统驱动程序能够利用高速缓存管理器通过简单的拷贝那些在高速缓存中的页面来访问数据, 而不用产生 IRP。

快速 I/O 并不总是发生。例如, 文件的第一次读写时需要设置该文件以供高速缓存(将文件映射到高速缓存中, 设置高速缓存数据结构)。如果调用者指定了异步读写, 快速 I/O 也不能使用。因为在满足将缓冲区与系统高速缓存之间的拷贝而进行的换页 I/O 操作期间, 调用者可能被停止, 这样就不能真正提供所要求的异步 I/O 操作。即使在同步 I/O 操作时, 文件系统驱动程序也可能会判定它不能用快速 I/O 机制, 例如, 如果正在操作的文件有一个被锁定的字节区域(就像调用 Win32 LockFile 和 UnlockFile 函数的结果)。因为, 高速缓存管理器不知道哪个文件的哪个部分被上了锁, 文件系统驱动程序必须检查读写的合法性, 这需要产生 IRP。快速 I/O 的决策树如图 5-21 所示。

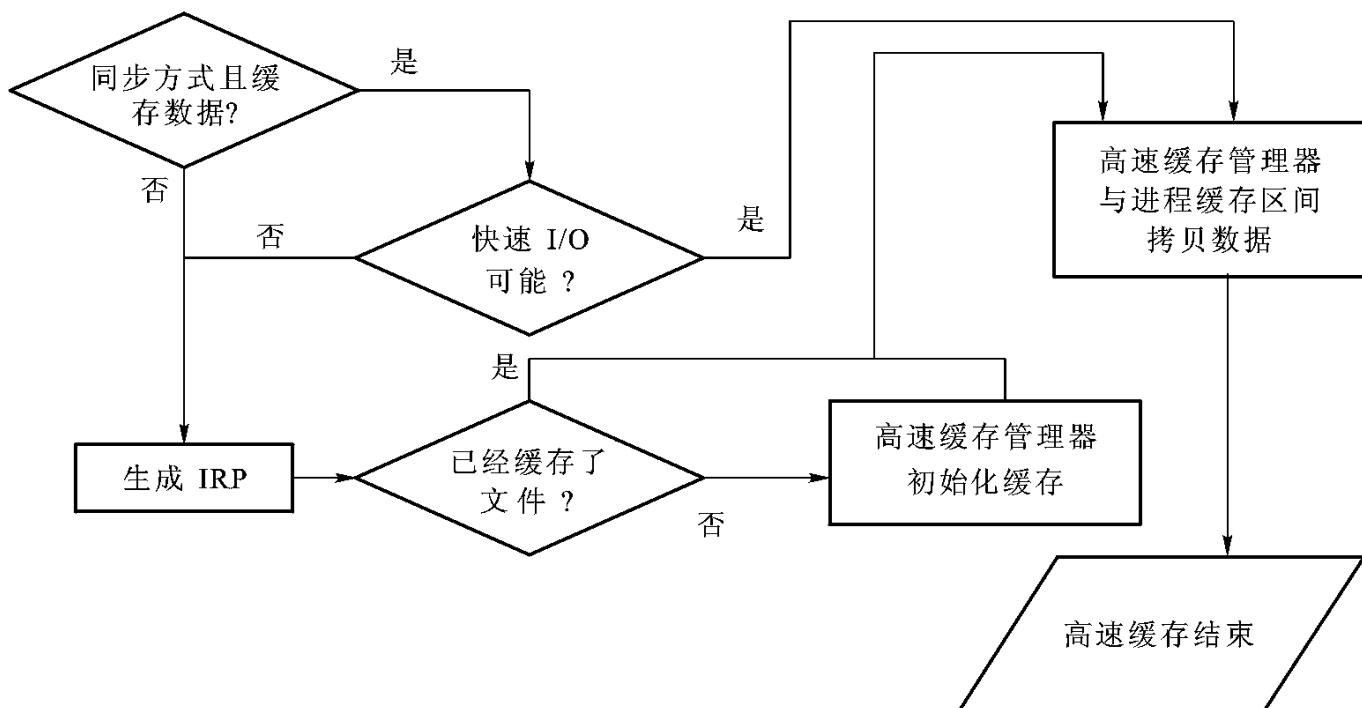


图 5-21 快速 I/O 的决策树

下面是快速 I/O 服务进行读写操作时涉及的几个步骤:

- (1) 线程进行一个读或写的操作;
- (2) 如果文件被缓存而且 I/O 同步时, I/O 请求就会被传送到文件系统驱动程序的快速 I/O 入口点。如果文件没有被缓存, 文件系统驱动程序设置该文件用于高速缓存, 这样下一次就可以使用快速 I/O 来满足读写请求。
- (3) 如果文件系统驱动程序的快速 I/O 例程断定可以使用快速 I/O, 它就调用高速缓存管理器的读或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

序返回到 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)。

(4) 高速缓存管理器将得到的文件偏移转换为高速缓存区的虚拟地址。

(5) 对读操作,高速缓存管理器将数据从系统高速缓存中拷贝到请求数据进程的缓冲区中;对写操作,高速缓存管理器将数据从进程的缓存区中拷贝到系统高速缓存。

(6) 下面的动作之一发生:

- 对于读,更新调用者私有缓存映射中的预读信息。

- 对于写,设置高速缓存中所有被修改页面的“脏”位,以便延迟写器可以将它写入磁盘。

- 对于通写文件,任何修改被刷新到磁盘。

6. 高速缓存支持例程

文件的数据第一次被访问时,文件系统驱动程序负责确定文件的某些部分是否被映射到了系统高速缓存。如果没有,文件系统驱动程序必须调用 CcInitializeCacheMap 函数去设置前面描述过的每个文件的数据结构。

一旦文件设置了高速缓存访问,文件系统驱动程序就可以调用几个函数中的一个来访问文件中的数据。有三种基本的访问缓存数据的方法,每种都适合一种特定的情况:

(1)“拷贝读取”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拷贝用户数据。

(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)“物理内存访问”方法,使用物理地址直接读写高速缓存的数据缓冲区。

文件系统驱动程序必须提供两种版本的文件读操作——有高速缓存的和没有高速缓存的,以避免内存管理器处理缺页时出现无限循环。当内存管理器通过文件系统从文件中取得数据(当然,通过设备驱动程序)来解决缺页时,它必须在 IRP 中设置“没有高速缓存”的标志,指出这是一个无需高速缓存的读操作。

下面介绍这三种缓存访问机制、目的和用法。

(1) 拷贝读取

系统高速缓存位于系统空间,所以,它被映射到每一个进程的地址空间。然而,同所有系统空间页面一样,高速缓存的页面不能在用户态被访问,那样会造成潜在的安全漏洞(例如,一个进程可能没有权利读取某些文件在系统高速缓存中的数据)。因此,用户程序读写缓存中的一个文件时必须依靠核心态的例程的服务,这些核心态的例程可以在系统空间高速缓存的数据缓冲区和用户进程空间应用程序数据缓冲区之间拷贝数据。文件系统驱动程序可以用来完成这些操作的函数被列在了下表 5-3 中。

表 5-3 用于拷贝到缓存和从缓存拷贝的核心态函数

| 函 数 | 描 述 |
|-----------------|---------------------------------------------------------|
| CcCopyRead | 将指定范围的字节从系统高速缓存中拷贝到用户的缓冲区 |
| CcFastCopyRead | CcCopyRead 的更快变种, 但限制在 32 位的文件偏移和同步读 (NTFS 采用, FAT 不用) |
| CcCopyWrite | 将指定范围的字节从用户的缓冲区拷贝到系统高速缓存中 |
| CcFastCopyWrite | CcCopyWrite 的更快变种, 但限制在 32 位的文件偏移和同步读 (NTFS 采用, FAT 不用) |

(2) 映射暂留

正如用户应用程序读写磁盘上的文件一样, 文件系统驱动程序需要读写描述文件自身的数据(元数据或卷结构数据)。由于文件系统驱动程序在核心态运行, 如果高速缓存管理器得到适当的通知, 它们就能够直接修改系统高速缓存中的数据。为了允许这种优化, 高速缓存管理提供了表 5-4 中列出的一些函数。文件系统驱动程序使用这些函数可以找到文件系统元数据在虚拟内存中的位置, 然后, 可以不借助中间缓冲区而直接修改元数据。

表 5-4 用于找到元数据位置的函数

| 函 数 | 描 述 |
|----------------------|--------------------------|
| CcMapData | 映射用于读访问的字节范围 |
| CcPinRead | 映射用于读/写访问的字节范围, 并且将它暂留内存 |
| CcPreparePinWrite | 映射用于写访问的字节范围, 并且将它暂留内存 |
| CcPinMappedData | 暂留一个以前映射的缓存区 |
| CcSetDirtyPinnedData | 告知高速缓存管理器, 数据已经被改变了 |
| CcUnpinData | 释放内存页, 以便它们可以从内存中被除去 |

如果文件系统驱动程序需要读取在高速缓存中的文件系统元数据, 它可以调用高速缓存管理器的映射接口来取得所需数据的虚拟地址。高速缓存管理器找到所有被请求的页面并把它们读入内存, 然后, 将控制权交给文件系统驱动程序。文件系统驱动程序便可以直接地访问这些数据了。

如果文件系统驱动程序需要修改高速缓存页面, 它可以调用高速缓存管理器的暂留服务, 该服务使被修改的页面驻留内存。那些页面实际上没有被锁在内存中(如同设备驱动程序为进行直接内存访问传输而锁定页面那样)。相反, 内存管理器的映射页面写入器看到这些页面是暂留的, 于是不把它们写入磁盘, 直到文件系统驱动程序释放为止。当页面被释放

后, 高速缓存管理器将任何变化刷新到磁盘, 然后释放元数据占用的高速缓存视图。

映射和暂留接口解决了实现文件系统的一个棘手的问题: 缓冲区管理。如果不能直接操纵被缓存的元数据, 在更新卷结构时文件系统就必须估计它所需要的缓冲区最大数目。由于高速缓存管理器允许文件系统在缓存中直接访问和修改它的元数据, 因此不再需要数据缓冲区, 只要简单地更新内存管理器提供的虚拟内存中的卷结构。文件系统受到的惟一限制是可供使用的内存数量。

(3) 物理内存访问

除了用于在高速缓存中直接访问元数据的映射和暂留接口外, 高速缓存管理器还提供了第三种访问缓存数据的接口: 直接存储器存取 (direct memory access, DMA)。DMA 函数用于不借助缓冲区从高速缓存读取或写入高速缓存, 比如网络文件系统在网络上进行传输。

DMA 接口将被高速缓存的用户数据的物理地址返回给文件系统 (而不是虚地址, 虚地址是映射和暂留接口返回的), 这个物理地址用于直接从物理内存向网络设备传输数据。虽然少量的数据 (1 KB 到 2 KB) 能够用一般的基于缓冲区的接口来传输, 但对于大量数据传输, 如网络服务器处理远程系统的文件请求, DMA 接口能够显著地提高性能。

要描述这些对物理内存的访问, 需要使用内存描述链表 (MDL)。下表中 4 个独立的函数构成了高速缓存管理器的 DMA 接口。

表 5-5 构成 DMA 接口的函数

| 函 数 | 描 述 |
|--------------------|---------------------------|
| CcMdlRead | 返回一个描述指定字节范围的 MDL |
| CcMdlReadComplete | 释放 MDL |
| CcMdlWrite | 返回一个描述指定字节范围的 MDL(可能包含 0) |
| CCMdlWriteComplete | 释放 MDL, 将那个字节区范围记上“写” |

7. 写阻塞

Windows 2000/XP 必须确认调度写操作是否会影响系统的性能, 然后, 再安排各项延迟写操作。首先, 它询问现在立刻写入一定数量的字节是否会损害性能, 如果必要则阻塞该项写操作。接下来, 它设置当写操作再次被允许时自动写入字节的回调。一旦获悉将要进行的写操作, 高速缓存管理器便会判断高速缓存中有多少脏页和有多少可以使用的物理内存。如果空闲的物理内存页不足, 高速缓存管理器立即阻塞请求向高速缓存中写数据的文件系统线程。高速缓存管理器的延迟写器会将一些脏页刷新到磁盘, 然后允许被阻塞的文件系统线程继续。当文件系统或网络服务器进行大量写操作时, 这种写阻塞机制防止了系统的

性能由于缺少内存而下降。

写阻塞对于网络重定向程序在低速传输的线路上传送数据也很有用。例如,假设一个本地进程通过 9600 波特率的线路向远程文件系统写大量数据。这些数据直到高速缓存管理器的延迟写器刷新高速缓存时才被写入远程的磁盘。如果重定向程序积累了大量刷新到磁盘的脏页,那么接收者在数据传输结束前可能会接收到一个网络超时。通过使用 CcSetDirtyPageThreshold 函数,高速缓存管理器允许网络重定向程序设置一个可以接受的高速缓存脏页数目界限,以防上述情况的发生。通过限制脏页的数量,重定向程序保证了高速缓存刷新操作不会引起网络超时。

5.9 实例研究:Linux 设备管理

5.9.1 Linux 设备管理概述

在 Linux 操作系统中,输入输出设备可以分为字符设备、块设备和网络设备。块设备把信息存储在可寻址的固定大小的数据块中,数据块均可以被独立地读写,建立块缓冲,能随机访问数据块。字符设备可以发送或接收字符流,通常无法编址,也不存在任何寻址操作。网络设备在 Linux 中是一种独立的设备类型,有一些特殊的处理方法。也有一些设备无法利用上述方法分类,如时钟,它们也需要特殊的处理。

在 Linux 中,所有的硬件设备均当作特殊的设备文件处理,可以使用标准的文件操作。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控制的所有设备具有相同的主设备号,并用不同的次设备号加以区别。网络设备也是当作设备文件来处理,不同的是这类设备由 Linux 创建,并由网络控制器初始化。

Linux 核心具体负责 I/O 设备的操作,这些管理和控制硬件设备控制器的程序代码称为设备驱动程序,它们是常驻内存的底层硬件处理子程序,具体控制和管理 I/O 设备的作用。虽然设备驱动程序的类型很多,它们都有以下的共同特性:

- 核心代码。设备驱动程序是 Linux 核心的重要组成部分,在内核运行。如果出现错误,则可能造成系统的严重破坏。
- 核心接口。设备驱动程序提供标准的核心接口,供上层软件使用。
- 核心机制和服务。设备驱动程序使用标准的核心系统服务,如内存分配、中断处理、进程等待队列等等。
- 可装载性。绝大多数设备驱动程序可以根据需要以核心模块的方式装入,在不需要

时可以卸装。

- 可配置性。设备驱动程序可以编译并链接进入 Linux 核心。当编译 Linux 核心时, 可以指定并配置你所需要的设备驱动程序。
- 动态性。系统启动时将监测所有的设备, 当一个设备驱动程序对应的设备不存在时, 该驱动程序将被闲置, 仅占用了一点内存而已。

Linux 的设备驱动程序可以通过查询 (polling)、中断和直接内存存取等多种形式来控制设备进行输入输出。

为解决查询方式的低效率, Linux 专门引入了系统定时器, 以便每隔一段时间才查询一次设备的状态, 从而, 解决忙式查询带来的效率下降问题。Linux 的软盘驱动程序就是以这样一种方式工作的。即便如此, 查询方式依然存在着效率问题。

一种高效率的 I/O 控制方式是中断。在中断方式下, Linux 核心能够把中断传递到发出 I/O 命令的设备驱动程序。为了做到这一点, 设备驱动程序必须在初始化时向 Linux 核心注册所使用的中断编号和中断处理子程序入口地址, /proc/interrupts 文件列出了设备驱动程序所使用的中断编号。

对于诸如硬盘设备、SCSI 设备等高速 I/O 设备, Linux 采用 DMA 方式进行 I/O 控制, 这类稀有资源一共只有 7 个。DMA 控制器不能使用虚拟内存, 且由于其地址寄存器只有 16 位 (加上页面寄存器 8 位), 它只能访问系统最低端的 16M 内存。DMA 也不能被不同的设备驱动程序共享, 因此, 一些设备独占专用的 DMA, 另一些设备互斥使用 DMA。Linux 使用 dma_chan 数据结构跟踪 DMA 的使用情况, 它包括拥有者的名字和分配标志两个字段, 可以使用 cat/proc/dma 命令列出 dma_chan 的内容。

Linux 核心与设备驱动程序以统一的标准方式交互, 因此, 设备驱动程序必须提供与核心通信的标准接口, 使得 Linux 核心在不知道设备具体细节的情况下, 仍能够用标准方式来控制和管理设备。

字符设备是最简单的设备, Linux 把这种设备当作文件来管理。在初始化时, 设置驱动程序入口到 device_struct (在 fs/devices.h 文件中定义) 数据结构的 chrdev 向量内, 并在 Linux 核心注册。设备的主标识符是访问 chrdev 的索引。device_struct 包括两个元素, 分别指向设备驱动程序和文件操作块。而文件操作块则指向诸如打开、读写、关闭等一些文件操作例行程序的地址。

块设备的标准接口及其操作方式非常类似于字符设备。Linux 采用 blk_devs 向量管理块设备。与 chrdev 一样, blk_devs 用主设备号作为索引, 并指向 blk_dev_struct 数据结构。除了文件操作接口以外, 块设备还必须提供缓冲区缓存接口, blk_dev_struct 结构包括一个请求子程序和一个指向 request 队列的指针, 该队列中的每一个 request 表示一个来自于缓冲区的数据块读写请求。

5.9.2 Linux 硬盘管理

一个典型的 Linux 系统一般包括一个 DOS 分区、一个 EXT2 分区 (Linux 主分区)、一个 Linux 交换分区、以及零个或多个扩展用户分区。Linux 系统在初始化时要先获取系统所有硬盘的结构信息以及所有硬盘的分区信息并用 gendisk 数据结构构成的链表表示, 其细节可以参见/include/linux/genhd 文件。

在 Linux 系统中, IDE 系统 (Inergrated Disk Electronic, 一种磁盘接口) 和 SCSI 系统 (Small Computer System Interface, 一种 I/O 总线) 的管理有所不同。Linux 系统使用的大多数硬盘都是 IDE 硬盘, 每一个 IDE 控制器可以挂接两个 IDE 硬盘, 一个称为主硬盘, 一个称为从硬盘。一个系统可以有多个 IDE 控制器, 第一个称为主 IDE 控制器, 其他称为从 IDE 控制器。Linux 系统最多支持 4 个 IDE 控制器, 每一个控制器用 ide_hwif_t 数据结构描述, 所有这些描述集中存放在 ide_hwifs 向量中。每一个 ide_hwif_t 包括两个 ide_drive_t 数据结构, 分别用于描述主 IDE 硬盘和从 IDE 硬盘。

初始化时, Linux 系统在 CMOS 中查找关于硬盘的信息, 并依次为依据构造上面的数据结构。Linux 系统将按照查找到的顺序给 IDE 硬盘命名。主控制器上的主硬盘的名字为 /dev/hda, 以下依次为 /dev/hdb、/dev/hdc、...。IDE 子系统向 Linux 注册的是的 IDE 控制器而不是硬盘, 主 IDE 控制器的主设备号为 3, 从 IDE 控制器的主设备号为 22。这意味着, 如果系统只有两个 IDE 控制器, blk_devs 中只有两个元素, 分别用 3 和 22 标识。

SCSI 总线是一种高效率的数据总线, 每条 SCSI 总线最多可以挂接八个 SCSI 设备。每个设备有惟一的标识符, 并且这些标识符可以通过设备上的跳线来摄制。总线上的任意两个设备之间可以同步或异步地传输数据, 在数据线为 32 位时数据传输率可以达到 40 MB/秒。SCSI 总线可以在设备间同时传输数据与状态信息。源设备和目标设备间的数据传输步骤最多可以有 8 个不同的阶段:

- 1) BUS FREE: 没有设备在总线的控制下, 总线上无事务发生。
- 2) ARBITRATION: 一个 SCSI 设备试图获得 SCSI 总线的控制权, 这时它把自己的 SCSI 标识符放到地址引脚上。具有最高 SCSI 标识符编号的设备将获得总线控制权。
- 3) SELECTION: 当设备成功地获得了对 SCSI 总线的控制权之后, 必须向它准备发送命令的那个 SCSI 设备发出信号。具体做法是将目标设备的 SCSI 标识符放置到地址引脚上。
- 4) RESELECTION: 在一个请求的处理过程中, SCSI 设备可能会断开连接。目标设备将再次选择源设备。不是所有的 SCSI 设备都支持这个阶段。
- 5) COMMAND: 源设备向目标设备发送 6B、10B 或 12B 命令。
- 6) DATA IN、DATA OUT: 数据在源设备和目标设备之间传输。
- 7) STATUS: 所有命令执行完毕后允许目标设备向源设备发送状态信息, 以指示操作是

否成功。

8) MESSAGE IN、MESSAGE OUT:信息在源设备和目标设备之间传输。

Linux SCSI 子系统包括两个基本组成部分,其数据结构分别用 host 和 device 来表示。Host 用来描述 SCSI 控制器,每个系统可以支持多个相同类型的 SCSI 控制器,每个均用一个单独的 SCSI host 来表示。Device 用来描述各种类型的 SCSI 设备,每个 SCSI 设备都有一个设备号,登记在 Device 表中。

5.9.3 Linux 网络设备

网络设备是传送和接收数据的一种硬件设备,如以太网卡,与字符设备和块设备不一样,网络设备文件在网络设备被检测到和初始化时由系统动态产生。在系统自举或网络初始化时,网络设备驱动程序向 Linux 内核注册。网络设备用 device 数据结构描述,该数据结构包含一些设备信息以及一些操作例程,这些例程用来支持各种网络协议,可以用于传送和接收数据包。Device 数据结构包括以下几个方面的内容:

1) 名称。网络设备名称是标准化的,每一个名字都能表达设备的类型,同类设备从 0 开始编号,如:/dev/ethN(以太网设备)、/dev/seN(SLIP 设备)、/dev/pppN(PPP 设备)、/dev/lo(回流设备)。

2) 总线信息。总线信息被设备驱动程序用来控制设备,包括设备使用的中断 irq、设备控制和状态寄存器的基地址 base address、设备所使用的 DMA 通道编号 DMA channel。

3) 接口标志。接口标志用来描述网络设备的特性和能力,如是否点到点连接、是否接收 IP 多路广播帧等。

4) 协议信息。协议信息描述网络层如何使用设备,其中包括:表示网络层可以传输的最大数据包尺寸;设备支持的协议方案,如 internet 地址方案为 AF_INET;所连接的网络介质的硬件接口类型,Linux 支持的介质类型有以太网、令牌环、X.25、SLIP、PPP、以及 Apple LocalTalk;以及网络设备有关的地址信息。

5) 包队列。等待由该网络设备发送的数据包队列,所有的网络数据包用特定数据结构描述,可以方便地添加或删除网络协议信息头。

6) 支持函数。指向每个设备的一组标准子程序,包括设置、帧传输、添加标准数据头、收集统计信息等子程序。

5.9.4 Linux 设备驱动程序

Linux 设备驱动程序是内核的一部分,由于设备种类繁多、设备驱动程序也有许多种,为了能协调设备驱动程序和内核的开发,必须有一个严格定义和管理的接口。例如,UNIX

SVR4 提出了 DDI/DKI (Device – Driver Interface/Driver – Kernel Interface、设备 – 驱动程序接口/设备驱动程序 – 内核接口) 规范。Linux 的设备驱动程序与外界的接口与 DDI/DKI 类似, 可分为三部分:(1) 驱动程序与系统内核的接口;(2) 驱动程序与系统引导的接口;(3) 驱动程序与设备的接口。按照功能, 设备驱动程序代码可分为以下部分:

- 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。

5.10 本 章 小 结

由于现代计算机外围设备种类繁多、特性各异,使得设备管理成为操作系统中最庞杂和琐碎的部分,其主要任务是控制外围设备和 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、设备与 CPU 的并行性,使得系统效率得到提高,同时,要为用户使用 I/O 设备屏蔽硬件细节,提供方便易用的接口。

首先,讨论了 I/O 硬件,介绍了设备和控制器的关系,控制器是软件处理的部分,应予关注。还介绍了 I/O 系统和 I/O 控制方式,目前,常用的设备和 CPU 之间的数据传送控制方式有四种:程序询问方式、中断控制方式、DMA 方式和通道方式。程序询问方式和中断控制方式仅适用于配置少量外设的场合,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;后者虽改进了上述缺点,但中断次数太多,CPU 累计化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了中断处理速度的限制。DMA 和通道方式较好的解决了所述缺陷,它们均采用了设备与和主存直接交换数据的方法,仅当一块数据传送结束,这两种方式才发出中断信号请求 CPU 干预,把 CPU 从繁杂的 I/O 事务中解放出来。它们的区别是:DMA 要求 CPU 执行设备驱动程序启动设备,并做好传送数据的有关准备工作;通道则完全是一个相对独立的 I/O 控制系统,仅当 CPU 发出 I/O 启动命令后,它便接收控制,完成全部 I/O 操作。

接着,分四层考察了 I/O 软件的功能:设备中断处理程序、设备驱动程序、与设备无关的 I/O 软件、用户层 I/O 软件。中断是应该尽量加以屏蔽的概念,设备中断处理程序放在操作系统的底层,主要工作有:分析中断类型作出相应处理,检查和修改进程状态等,它的任务要尽量少以提高性能;设备驱动程序中包括了所有与设备相关的代码,它的工作是: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,它应当对其上层的软件屏蔽所有硬件细节;与设备无关的 I/O 软件的基本功能是执行适用于所有设备的常用 I/O 功能,并向用户层软件提供一个一致性的接口,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 库例程和 Spooling 程序。

本章对通道作了稍细的讨论。通道又称为 I/O 处理机,采用通道技术主要解决输入输出操作的独立性和设备与 CPU 工作的并行性,能大幅度提高系统整体性能。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,以及 CPU 和通道之间的通信。进一步介绍了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。我们介绍了常用的缓冲技术: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。

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 cache。作为操作系统的辅助存储器,有着繁重的输入输出负载,这些磁盘输入输出请求便是调度的对象,系统必须采用一种调度策略,使能按最佳次序执行要求访问的诸请求,这里讨论了多种磁盘驱动调度算法并比较了它们的优劣,包括: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 cache 是一个缓冲区,通常在主存开辟,作为磁盘块在磁盘与其余主存之间的高速缓冲。由于程序局部性原理,磁盘 cache 的使用可大幅度减少磁盘与主存之间的 I/O 传送的块数。

本章还对独立磁盘冗余阵列 RAID 进行了介绍,它采用一组较小容量的、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,再加进冗余技术,数据能用多种方式组织和分布存储,于是,独立的 I/O 请求能被并行处理,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,改进了 I/O 性能和系统可靠性。

最后讨论了 Spooling 系统,它是能把一个物理设备虚拟化成多个虚拟(逻辑)设备的技术,能用共享设备来模拟独享设备的技术,在中断和通道硬件的支撑下,操作系统采用多道程序设计技术,合理分配和调度各种资源,实现联机的外围设备同时操作。spooling 系统主要有:预输入、井管理和缓输出组成,已被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. 叙述设备管理的基本功能?
2. 简述各种 I/O 控制方式及其主要优缺点。
3. 试述直接内存存取 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? 分别简单说明之。
6. 叙述 I/O 中断的类型及其功能。
7. 叙述 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 CCW 和通道程序?
10. 什么是通道地址字 CAW 和通道状态字 CSW?
11. 叙述采用通道技术时, I/O 操作的全过程。
12. 为什么要引入缓冲技术? 其实现的基本思想是什么?
13. 简述常用的缓冲技术?
14. 什么是驱动调度? 有哪些常用的驱动调度技术?
15. 外围设备分成哪些类型? 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“设备类”, 又能按指定“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”? 什么叫输入井和输出井?
20. 井管理程序有什么功能? 它是如何工作的?
21. 什么叫虚拟设备? 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. 叙述 Spooling 系统和作业调度的关系。
24. 操作系统提供了 Spooling 功能后, 系统在单位时间内处理的作业数是否增加了, 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 如何把独占设备改造成共享设备的?
27. 叙述 Windows 2000/XP I/O 系统的结构和模型。
28. Windows 2000/XP 支持哪些类型的设备驱动程序?

29. Windows 2000/XP 核心态设备驱动程序由哪些例程组成?

30. 设单缓冲情况下,磁盘把一块数据输入缓冲区花费时间为 T ;系统从缓冲区将数据传到用户区花费时间为 M ;处理器处理这块数据花费时间为 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 IBM370 系列机为例,说明大型机系统的 I/O 过程。

33. 目前常用的磁盘驱动调度算法有哪几种?每种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什么好处?

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减少为若干个 I/O 服务的总时间。设磁鼓上分为 20 个区,每区存放一个记录,磁鼓旋转一周需 20 ms,读出每个记录平均需用 1 ms,读出后经 2 ms 处理,再继续处理下一个记录。在不知当前磁鼓位置的情况下:(1)顺序存放记录 1、……,记录 20 时,试计算读出并处理 20 个记录的总时间;(2)给出优先分布 20 个记录的一种方案,使得所花的总处理时间减少,且计算出这个方案所花的总时间。

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移动的总柱面数。假设磁头当前位置下在磁道 100。

3. 上题中,分别按升序和降序移动,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 字节,并依次存放在 50、121、75、80、63 号磁盘块上。现要读出文件的 1569 字节,问访问哪一个磁盘块?

5. 对磁盘存在下面五个请求:

| 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。试分析对这五个请求如何调度,可使磁盘的旋转圈数为最少?

6. 有一具有 40 个磁道的盘面,编号为 0 ~ 39,当磁头位于第 11 磁道时,顺序来到如下磁道请求: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 等三种磁盘驱动调度算法,计算出它们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,并刚刚完成了 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:为完成上述请求,下列算法存取臂移动的总量是多少?并算出存取臂移动的顺序。

- (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。

8. 除 FCFS 外,所有磁盘调度算法都不公平,如造成有些请求饥饿,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. 若磁头的当前位置为 100 柱面,磁头正向磁道号减小方向移动。现有一磁盘读写请求队列,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和电梯调度算法,试计算出各种算法的移臂经过的柱面数?

10. 若磁头的当前位置为 100 柱面,磁头正向磁道号增加方向移动。现有一磁盘读写请求队列,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、最短寻道时间优先和扫描算法,试计算出各种算法的移臂经过的柱面数?

11. 设有长度为 L 个字节的文件存到磁带上,若规定磁带物理块长为 B 字节,试问:(1)存放该文件需多少块?(2)若一次启动磁带机交换 K 块,则存取这个文件需执行 I/O 操作多少次?

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。如果驱动程序接到访求是读出 606 块,计算该信息块的物理位置。

13. 假定磁带记录密度为每英寸 800 字符,每一逻辑记录为 160 个字符,块间隙为 0.6 英寸。今有 1 500 个逻辑记录需要存储,尝试:(1)计算磁带利用率?(2)1 500 个逻辑记录占多少磁带空间?(3)若要使磁带空间利用率不少于 50,至少应以多少个逻辑记录为一组?

14. 假定磁带记录密度为每英寸 800 字符,每一逻辑记录为主 200 字符,块间隔为 0.6 英寸现有 3200 个逻辑记录需要存储,如果不考虑存储记录,则不成组处理和以 8 个逻辑记录为一组的成组处理时磁带的利用率各是多少?两种情况下,3 200 个逻辑记录需占用多少磁带空间?

15. 一个软盘有 40 个柱面,查找移过每个柱面花 6 ms。若文件信息块零乱存放,则相邻逻辑块平均间隔 13 个柱面。但优化存放,相邻逻辑块平均间隔为 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 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 6 ms,用以下算法计算出查找时间:1)FCFS,2)最短查找优先,3)电梯调度(正向柱面大的方向)。

17. 假定在某移动臂磁盘上,刚刚处理了访问 75 号柱面的请求,目前正在 80 号柱面读信息,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) 最短寻找时间优先算法

分别列出实际处理上述请求的次序。

18. 计算机系统中, 屏幕显示分辨率为 640×480 , 若要存储一屏 256 彩色的图像, 需要多少字节存储空间?

19. 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 分别表示任一逻辑磁盘块的柱面号、磁头号、磁道内块号, 则 x 与 a, b, c 可通过如下公式进行转换:

$$x = k \times m \times a + k \times b + c$$

$$a = (x - 1) \text{ DIV } (k \times m)$$

$$b = ((x - 1) \text{ MOD } (k \times m)) \text{ DIV } k$$

$$c = ((x - 1) \text{ MOD } (k \times m)) \text{ MOD } k + 1$$

若某磁盘组为 $n = 200, m = 20, k = 10$, 问:

(1) 柱面号为 185, 磁头号为 12, 道内块号为 5 的磁盘块的逻辑磁盘块号为多少?

(2) 逻辑磁盘块号为 1200, 它所对应的柱面号、磁头号及磁道内块号为多少?

(3) 若每一磁道内的信息块从 0 开始编号, 依次为 $0, 1, \dots, k - 1$, 其余均同题设, 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有效的文件使用和操作方法。文件这一术语不但反映了用户概念中的逻辑结构, 而且和存放它的辅助存储器(也称文件存储器)的存储结构紧密相关。所以, 同一个文件必须从逻辑文件和物理文件两个侧面来观察它。对于用户来说, 可按自己的愿望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”来实现对用户文件信息的存储和检索。可见, 使用者在处理他的信息时, 只需关心所执行的文件操作及文件的逻辑结构, 而不必涉及存储结构。但对文件系统本身来说, 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用户信息的管理, 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、文件内位移就可以进行数据(字节、字段或记录)的读、写、改、删操作; 而作为实现这些功能的文件系统来说, 它的工作与存储硬件紧密相关, 是根据用户的文件操作请求, 转化为对设备(磁盘)上的信息按照所在的位置(设备号、柱面号、磁道块号)进行寻址、读写和控制。所以, 文件系统的功能就是要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、逻辑操作与物理操作之间实现转换, 保证存取速度快、存储空间利用率高、数据可共享、安全可靠性好。面向用户来说, 文件系统的功能应该有:

- 文件的按名存取
- 文件目录的建立和维护
- 实现从逻辑文件到物理文件的转换
- 文件存储空间的分配和管理
- 提供合适的文件存取方法
- 实现文件的共享、保护和保密
- 提供一组可供用户使用的文件操作

为了实现这些功能, 操作系统必须考虑文件目录的建立和维护、存储空间的分配和回收、数据的保密和保护、监督用户存取和修改文件的权限、处理在不同存储介质上信息的表示方式、信息的编址方法、信息的存储次序、以及怎样检索用户信息等问题。

6.1 文 件

6.1.1 文件的概念

早期计算机系统中没有文件管理机构,用户自行管理辅助存储器上的信息,按照物理地址安排信息,组织数据的输入输出,还要记住信息在存储介质上的分布情况,繁琐复杂、易于出错、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了良好的物质基础。多道程序、分时系统的出现,多个用户以及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了文件系统,以适应系统管理和用户使用软件资源的需要。对计算机系统中软件资源的管理形成了操作系统的文件系统。

文件是由文件名字标识的一组信息的集合。文件名字是字母或数字组成的字母数字串,它的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:一个源程序、一批数据、各类语言的编译程序可以各自组成一个文件。操作系统提供文件系统后,首先,用户使用方便,使用者无需记住信息存放在辅助存储器中的物理位置,也无需考虑如何将信息存放在存储介质上,只要知道文件名,给出有关操作要求便可存取信息,实现了“按名存取”。特别,当文件存放位置作了改变,甚至更换了文件的存储设备,对文件的使用者也没有丝毫影响;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,而文件系统能提供各种安全、保密和保护措施,故可防止对文件信息的有意或无意的破坏或窃用。此外,在文件使用过程中可能出现硬件故障,这时文件系统可组织重执或恢复,对于因硬件失效而可能造成的文件信息破坏,可组织转储以提高文件的可靠性。最后,文件系统还能提供文件的共享功能,如不同的用户可以使用同名或异名的同一文件。这样,既节省了文件存放空间,又减少了传递文件的交换时间,进一步提高了文件和文件空间的利用率。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 文件的命名

文件是一种抽象机制,它隐蔽了硬件和实现细节,提供了把信息保存在磁盘上而且便于以后读取的手段,使得用户不必了解信息存储的方法、位置以及存储设备实际运作方式便可存取信息。在这一抽象机制中最重要的是文件命名,当一个进程创建一个文件时必须给出文件名字,以后这个文件将独立于进程存在直到它被显式地删除;当其他进程要使用这一文

件时必须显式地指出该文件名字;操作系统也将根据该文件名字对其进行控制和管理。

各个操作系统的文件命名规则略有不同,即文件名字的格式和长度因系统而异。但一般来说,文件名字由文件名和扩展名两部分组成,中间用“.”分隔开来。它们都是字母或数字组成的字母数字串。操作系统通常还提供通配符,以便于对一组文件进行分类或同时进行操作,通配符“?”代表了文件名字中其所在位置的任何一个可用字符;通配符“*”则代表了文件名字中所在位置的任何一个可用字符串。

例如,在 MS-DOS 系统中,一个文件名长度限于 1 至 8 个字符,扩展名长度限于 0 至 3 个字符;前者用于识别文件;后者用于标识文件特性;两者之间用一个圆点隔开。通常文件名和扩展名均不区分大小写,可用字符包括字母、数字及一些特殊符号,每个操作系统对可用的文件名字符作了一定限制,像 Windows 的文件名和扩展名不能使用 “\”、“/”、“<”、“>”、“|”等字符。扩展名常常用作定义各种类型的文件,系统有一些约定扩展名,例如:COM 表示可执行的二进制代码文件;EXE 表示可执行的浮动二进制代码文件;LIB 表示库程序文件;BAT 表示批命令文件;OBJ 表示编译或汇编生成的目标文件等。

定义文件扩展名是一种习惯,它并不是源于 MS-DOS,尽管一些操作系统的文件名中允许存在多个圆点,如 Windows 98 和 UNIX,但是文件扩展名的定义习惯依然被大多数用户和应用所默认,例如,汇编语言源程序的扩展名应取“.asm”,C 语言编译器要求被编译的源程序扩展名应取“.c”。许多文件系统支持多达 255 个字符的文件名,如 Windows,也有很多操作系统的文件命名需要区分大小写,如 UNIX。

6.1.3 文件的类型

在现代操作系统中,不但信息组织成文件,对设备的访问也都是基于文件进行的,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。

文件可以按各种方法进行分类:如按用途可分成:系统文件、库文件和用户文件;按保护级别可分成:只读文件、读写文件和不保护文件;按信息流向可分成:输入文件、输出文件和输入输出文件;按存放时限可分成:临时文件、永久文件、档案文件;按设备类型可分成:磁盘文件、磁带文件、软盘文件。此外,还可以按文件的逻辑结构或物理结构进行分类,将在下面作进一步讨论。

UNIX 操作系统支持以下几种不同类型的文件:

- 普通文件。一般用户建立的源程序文件、数据文件、目标代码文件及操作系统自身文件、库文件、实用程序文件都是普通文件,它们通常存储在外存储设备上。
- 目录文件。管理和实现文件系统的由文件目录组成的系统文件,对目录文件可进行与普通文件一样的种种文件操作。

- 块设备文件。用于磁盘、光盘或磁带等块设备的 I/O 操作。
- 字符设备文件。用于终端、打印机等字符设备的 I/O 操作。

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 DOS、Windows 等系统中每一行以回车换行结束，整个文件以 CTRL + Z 结束；在 UNIX 等系统中每一行以换行结束，整个文件以 CTRL + D 结束。ASCII 文件的最大优点是可以原样显示和打印，也可以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，它往往有一定的内部结构，组织成字节流，如，可执行文件是指令和数据的流，记录式文件是逻辑记录的流。块设备文件和字符设备文件合称特殊文件，把所有 I/O 设备统一在文件系统下，有利于系统管理，方便了用户使用。

6.1.4 文件的属性

大多数操作系统设置了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：

- 文件基本属性。文件名字、文件所有者、文件授权者、文件长度等。
- 文件的类型属性。如普通文件、目录文件、系统文件、隐式文件、设备文件等。也可按文件信息分为：ASCⅡ码文件、二进制码文件等。
- 文件的保护属性。如可读、可写、可执行、可更新、可删除等，可改变保护、以及档案属性。
- 文件的管理属性。如文件创建时间、最后存取时间、最后修改时间等。
- 文件的控制属性。逻辑记录长、文件当前长、文件最大长，以及允许的存取方式标志，关键字位置、关键字长度等。

文件的保护属性用于防止文件被破坏，称为文件保护。它包括两个方面：一是防止系统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。

为防止系统崩溃造成文件破坏，定时转储是一种经常采用的方法，系统的管理员每隔一段时间，或一日、或一周、或一月、或一个时期，把需要保护的文件保存到另一个介质上，以备数据破坏后恢复。由于需要备份的数据文件可能非常多，增量备份是必须的，为此操作系统专门为文件设置了档案属性，用以指明该文件是否被备份过。操作系统往往也提供备份和转储工具以方便用户转储，如 DOS 的 XCOPY 命令、BACKUP 命令和 RESTORE 命令，Windows 的备份工具，UNIX 的 compress 命令、tar 命令、bar 命令等。第三方公司也提供这样一些备份工具，比较著名的有 arj、lha、winzip 等等。一些应用程序本身也携带备份工具，如数据库管理系统。另外，一些备份工具甚至支持自动定时转储。

必须看到，虽然定时转储的成本较小，但是它不能完全做到百分之百的数据恢复，这对

于实时应用和重要的商业应用来说是不够的。为此又引入了多副本技术,即把重要的数据存放在不同的物理磁盘,乃至不同的联网计算机上,这样系统崩溃造成文件破坏后,可以从另一个文件副本中读取数据。多副本技术的实现同样需要借助于文件的保护属性,且实现开销很大。因此,现代操作系统和数据库系统往往采用磁盘冗余的方法实现多副本,即两个磁盘存放同样的信息,如磁盘镜像技术。采用磁盘信息冗余后,读数据时可以从任何一个磁盘中读,不会造成系统性能降低,但是写数据则需要向两个磁盘各写一次。显然,由操作系统的设备驱动程序直接控制两次写的效率将明显高于应用进程控制两次写磁盘,这就是磁盘镜像往往由操作系统而不是数据库系统来提供的原因。一种更高效的解决方法是计算机系统拥有两个通道,通过两个通道各控制一个磁盘,在硬件上实现同时读写。

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性策略来实现,其基本思想是建立如下的三元组:

(用户、对象、存取权限)

其中:

- 用户。是指每一个操作系统使用者的标识。
- 对象。在操作系统中一般都是文件,因为,操作系统把设备资源也统一到文件层次,如通过设备文件使用设备、通过 socket 关联文件使用进程通信等,当然不排除其他资源。
- 存取权限。定义了用户对文件的访问权,如,读、写、删除、执行等等。一个安全性较高的系统权限划分得较多较细。

实现这一机制需要建立一个庞大的存取控制矩阵,为简化实现代价,可以把用户划分为几类,进一步规定这几类用户对文件和文件目录的存取权限并把它保存在文件目录项的保护属性中。以 UNIX/Linux 为例,它把用户分为文件主、同组用户、其他用户三类,分别定义存取权限可读 r、可写 w、可执行 x,目录项中的文件属性共有 10 位:

— rwxrwxrwx

其中:

- 第 1 位表示文件是普通文件(-),还是目录文件(d)、符号链接文件(l)、设备文件(b/c)。
- 第 2~4 位表示文件主对文件的存取权限。
- 第 5~7 位表示同组用户对文件的存取权限。
- 第 8~10 位表示其他用户对文件的存取权限。

如一个文件的属性是 - rwxr-x--x, 表示该文件是普通文件,文件主对它可读、可写、可执行;同组用户对它可读、可执行;其他用户对它只可执行。

6.1.5 文件的存取方法

从用户使用观点来看,他们关心的是数据的逻辑结构,即记录及其逻辑关系,数据独立于物理环境;从系统实现观点来看,数据则被文件系统按照某种规则排列和存放到物理存储介质上。那么,输入的数据如何存储?处理的数据如何检索?数据的逻辑结构和数据物理结构之间怎样接口?谁来完成数据的成组和分解操作?这些都是存取方法的任务。存取方法是操作系统为用户程序提供的使用文件的技术和手段。

在有些系统中,对每种类型文件仅提供一种存取方法。但象 IBM 系统能支撑许多种不同的存取方法,以适应用户的不同需要,因而,存取方法也就成为文件系统中重要的设计问题了。文件类型和存取方法之间存在密切关系,因为,设备的物理特性和文件类型决定了数据的组织,也就在很大程度上决定了能够施加于文件的存取方法。

1. 顺序存取

按记录顺序进行读/写操作的存取方法称顺序存取。固定长记录的顺序存取是十分简单的。读操作总是读出下一次要读出的文件的下一个记录,同时,自动让文件记录读指针推进,以指向下一次要读出的记录位置。如果文件是可读可写的,再设置一个文件记录写指针,它总指向下一次要写入记录的存放位置,执行写操作时,将一个记录写到文件末端。允许对这种文件进行前跳或后退 N (整数)个记录的操作。对于固定长记录的顺序文件也可采用随机访问,当要读出第 i 个记录时,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,每个记录的长度信息存放于记录前面一个单元中,它的存取操作分两步进行。读出时,根据读指针值先读出存放记录长度的单元,然后,得到当前记录长后,再读出当前记录。

由于顺序文件是顺序存取的,可采用成组和分解操作来加速文件的输入输出。

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个记录。例如,航空订票系统,把特定航班的所有信息用航班号作标识,存放在某物理块中,用户预订某航班时,需要直接将该航班的信息取出。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,一个文件可以看作由顺序编号的物理块组成的,这些块常常划成等长,作为定位和存取的一个最小单位,如一块为 1 024 字节、4 096 字节,视系统和应用而定。于是用户可以请求读块 22、然后,写块 48,再读块 9 等等,直接存取文件对读或写块的次序没有限制。用户提供给操作系统的是相对块号,它是相对于文件开始位置的一个位移量,而绝对块号则由系统换算得到。

3. 索引存取

第三种类型的存取是基于索引文件的索引存取方法。由于文件中的记录不按它在文件中的位置,而按它的记录键来编址,所以,用户提供给操作系统记录键后就可查找到所需记录。

通常记录按记录键的某种顺序存放,例如,按代表键的字母先后次序来排序。对于这种文件,除可采用按键存取外,也可以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。实际的系统中,大都采用多级索引,以加速记录查找过程。

6.1.6 文件的使用

用户通过两类接口与文件系统联系:第一类是与文件有关的操作命令或作业控制语言中与文件有关的 JCL 语句,例如,UNIX 中的 cat, cd, cp, find, mv, rm, mkdir, rmdir 等等,这些构成了必不可少的文件系统人机接口。第二类是提供给用户程序使用的文件类系统调用,构成了用户和文件系统的另一个接口,通过这些接口用户能获得文件系统的各种服务。一般地讲,文件系统提供的基本文件类系统调用有:

- 建立文件。当用户要求把一批信息作为一个文件存放在存储器中时,使用建立文件操作向系统提出建立一个文件的要求。

- 打开文件。文件建立之后能立即使用,要通过“打开”文件操作建立起文件和用户之间的联系。文件打开以后,直至关闭之前,可被反复使用,不必多次打开,这样做能减少查找目录的时间,加快文件存取速度,从而,提高文件系统的运行效率。所以,从系统角度来看,设置打开文件和关闭文件操作能改善性能;从用户角度来看,能以显式提出对文件的使用要求,理解为有了一种权限许可,防止错误使用文件。

- 读/写文件。文件打开以后,就可以用读/写系统调用访问文件,调用这两个操作,应给出以下参数:文件名、主存缓冲地址、读写的记录或字节个数;对有些文件类型还要给出读/写起始逻辑记录号。

- 文件控制。文件打开以后,把文件的读写指针定位到文件头、文件尾、或文件中的任意位置,或执行前跳、后退等各种控制操作,便于用户随机访问文件内容。

- 关闭文件。当一个文件使用完毕后,使用者应关闭文件以便让别的使用者用此文件。关闭文件的要求可以用关闭操作直接向系统提出;也可用隐含方式实现,当使用同一设备上的另外一个文件时,就可以认为先要关闭当前使用文件,然后,再打开新文件。调用关闭系统调用的参数与打开操作相同。

- 撤销文件。当一个文件不再需要时,可向系统提出撤销文件。

6.2 文件目录

6.2.1 文件目录与文件目录项

文件系统怎样实现文件的“按名存取”?如何查找文件存储器中的指定文件?如何有效地管理众多的用户文件和系统文件?文件目录便是用于这些方面的重要手段。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,要求编排的目录便于查找、防止冲突,目录的检索方便迅速。由于文件目录也需要永久保存,所以,把文件目录也组织成文件存放在磁盘上称目录文件。

有了文件目录后,就可实现文件的“按名存取”。每一个文件在文件目录中登记一项,所以,实质上文件目录是文件系统建立和维护的它所包含的文件的清单,每个文件的文件目录项又称文件控制块 FCB(File Control Block),一般应该包括以下内容:

- 有关文件存取控制的信息。如文件名、用户名、文件主存取权限、授权者存取权限、文件类型和文件属性,如读写文件、执行文件、只读文件等。
- 有关文件结构的信息。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。文件的物理结构,如文件所在设备名,文件物理结构类型,记录存放在外存的相对位置或文件第一块的物理块号,也可指出文件索引的所在位置等。
- 有关文件使用的信息。已打开该文件的进程数,文件被修改的情况,文件最大和当前大小等。
- 有关文件管理的信息。如文件建立日期、文件最近修改日期、文件访问日期、文件保留期限、记帐信息等。

有了文件目录后,就可实现文件的“按名存取”。每当建立一个新文件时,系统就要为它设立一个 FCB,其中记录了这个文件的所有属性信息。多个文件的 FCB 便组成了文件目录,文件目录也用文件形式保存起来,这个文件称目录文件。当用户要求存取某个文件时,系统查找目录文件,先找到相对应的文件目录,然后,比较文件名就可找到所寻文件的文件控制块 FCB(文件目录项),再通过 FCB 指出的文件的文件信息相对位置或文件信息首块物理位置等就能依次存取文件信息。

UNIX 采用了一种比较特殊的目录项建立方法。为了减少检索文件访问的物理块数,把文件目录项中的文件名和其他管理信息分开,后者单独组成定长的一个数据结构,称为索引节点(in-node),该索引节点的编号称索引号,记为 i-node。于是,文件目录项中仅剩下 14 个字节的文件名和两个字节的 i-node。因此,一个物理块可存放 32 个文件目录项,系统把

由文件目录项组成的目录文件和普通文件一样对待, 均存放在文件存储器中。

文件存储设备上的每一个文件, 都有一个外存文件控制块(又称外存索引节点)inode与之对应, 这些inode被集中放在文件存储设备上的inode区。文件控制块inode对于文件的作用, 犹如进程控制块proc、user对于每个进程的作用, 集中了这个文件的属性及有关信息, 找到了inode, 就获得了它所对应的文件的一切必要信息。每一个外存的inode结构使用32个字节, 包含如下信息: 文件长度及在存储设备上的物理位置、文件主的各种标识、文件类型、存取权限、文件勾连(链接)数、文件访问和修改时间, 以及inode节点是否空闲。下面列出外存索引节点的部分内容:

- di-mode 文件属性, 如文件类型、存取权限。
- di-nlink 连接该索引节点的目录项数(共享数)。
- di-uid 文件主用户标识。
- di-gid 文件同组用户标识。
- di-size 文件大小(以字节计数)。
- di-add[8]存放文件所在物理块号的索引表。
- di-atime 文件最近被访问的时间。
- di-mtime 文件最近被修改的时间。
- di-ctime 文件最近创建的时间。

外存索引节点inode记录了一个文件的属性和有关信息。可以想象, 在对某一文件的访问过程中, 会频繁地涉及到它, 不断来回于内、外存之间引用它, 当然是极不经济的。为此, UNIX/Linux在系统占用的内存区里开辟了一张表——内存索引节点inode表(又称活动文件控制块表或活动索引节点表), 该表共有100个表目, 每个表目称为一个内存索引节点inode。当需要使用某文件的信息, 而在内存inode表中找不到其相应的inode时, 就申请一个内存inode, 把外存inode的内容拷贝到这个内存inode中, 随之就使用这个内存inode来控制文件的读写。通常, 在最后一个用户关闭此文件后, 内存索引节点inode的内容被写到外存索引节点inode中, 然后, 释放内存inode以供它用。把文件控制块的内容与索引节点号分开, 不仅加快了目录检索速度, 而且, 便于实现文件共享, 有利于系统的控制和管理。

6.2.2 一级目录结构

如图6-1所示, 最简单的文件目录是一级目录结构, 在操作系统中构造一张线性表, 与每个文件有关的属性占用一个目录项就成了一级目录结构。单用户微型机操作系统CP/M的软盘文件便采用这一结构, 每个磁盘上设置一张一

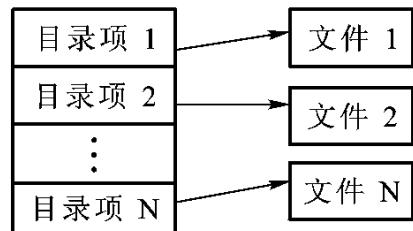


图6-1 一级目录结构示意图

级文件目录表,不同磁盘驱动器上的文件目录互不相关。文件目录表由长度为32字节的文件目录项组成,文件目录项0称目录头,记录有关文件目录表的信息,其他每个文件目录项又称文件控制块。文件目录中列出了盘上全部文件的有关信息。CP/M操作系统中文件目录项包括:盘号、文件名、扩展名、记录数、存放位置等。

一级文件目录结构存在若干缺点:一是重名问题,它要求文件名和文件之间有一一对应关系,但要在多用户的系统中,由于都使用同一文件目录,一旦文件名用重,就会出现混淆而无法实现“按名存取”。如果人为地限制文件名命名规则,对用户来说又极不方便;二是难于实现文件共享,如果允许不同用户使用不同文件名来共享同一个文件,这在一级目录中是很难实现的,为了解决上述问题,操作系统往往采用二级目录结构,使得每个用户有各自独立的文件目录。

6.2.3 二级目录结构

在二级目录中,第一级为主文件目录,它用于管理所有用户文件目录,它的目录项登记了系统接受的用户名及该用户文件目录的地址。第二级为用户的文件目录,它为该用户的每个文件保存一登记栏,其内容与一级目录的目录项相同。每一用户只允许查看自己的文件目录。图6-2是二级文件目录结构示意图。当一个新的用户作业进入系统执行时,要为其开辟一个内存区域存放该作业的文件目录,并把文件目录的起始地址填入主文件目录

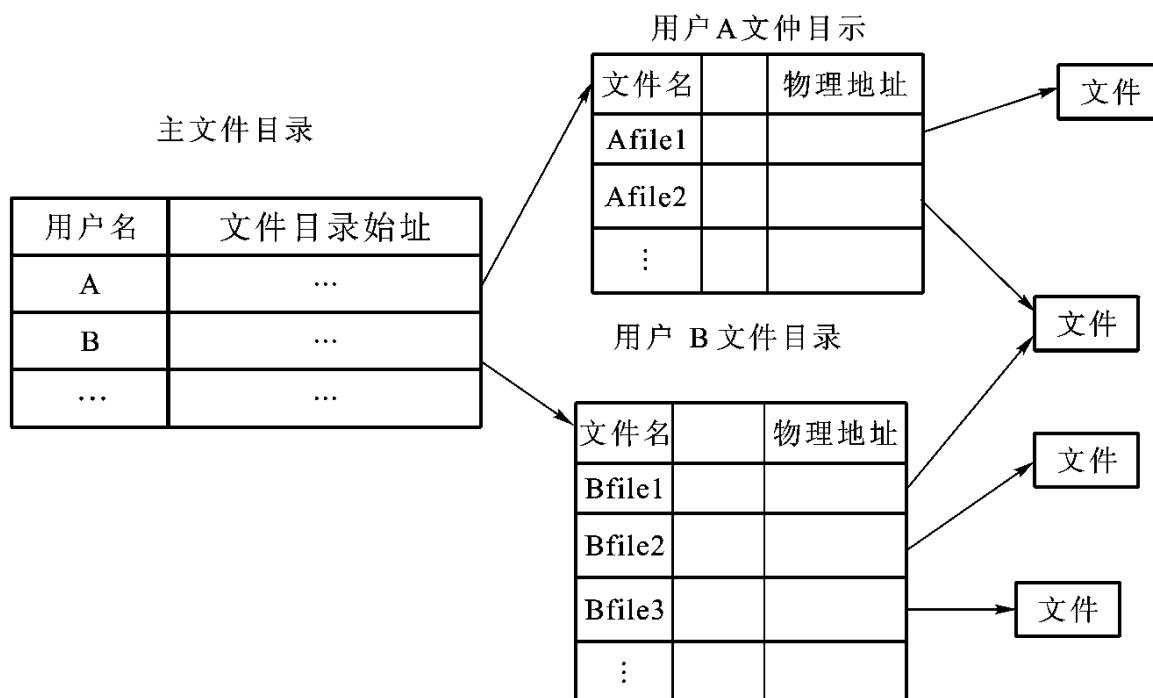


图6-2 二级目录结构示意图

录的一个空目录项中,还需把用户名等信息也填入该目录项。当用户需要访问某个文件时,系统根据用户名从主文件目录中找出该用户的文件目录的物理位置,然后,就可找到文件的文件控制块信息,其余的工作与一级文件目录类似。

采用二级目录管理文件时,因为任何文件的存取都通过主文件目录,于是可以检查访问文件者的存取权限,避免一个用户未经授权就存取另一个用户的文件,使用户文件的私有性得到保证,实现了对文件的保密和保护。特别是不同用户具有同名文件时,由于各自有不同的用户文件目录而不会导致混乱。对于文件的共享,原则上只要把对应目录项指向同一物理位置的文件即可。

6.2.4 树形目录结构

二级目录的推广形成了多级目录。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,形成了层次文件系统。如图 6-3 所示,多级目录结构通常采用树形结构,它是一棵倒向的有根树,树根是根目录;从根向下,每一个树枝是一个子目录;而树叶是文件。树型多级目录有许多优点;较好地反映了现实世界中具有层次关系的数据集合和较确切地反映系统内部文件的分支结构;不同文件可以重名,只要它们不位于同一末端的子目录中,易于规定不同层次或子树中文件的不同存取权限,便于文件的保护、保密和共享等。

在树形目录结构中,一个文件的全名将包括从根目录开始到文件为止,通路上遇到的所有子目录路径。各子目录名之间用正斜线/(或反斜线\)\)隔开,其中,子目录名组成的部分又称为路径名。文件可以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文件一样处理。

MS-DOS2.0 以上版本采用树形目录结构。在对磁盘进行格式化时,在其上建一目录,称为根目录,对于双面盘而言,根目录能存放的文件数为 112 个。但根目录除了能存放文件目录外,还可以存放子目录,依次类推。可以形成一个树形目录结构。子目录和根目录不同,可以看作一般的文件,可以像文件一样进行读写操作,它们被存放在盘上的数据区中,也就是说允许存放任何数目的文件和子目录。

UNIX 操作系统的文件系统也采用树形多级目录结构,在根目录之下有: dev 设备子目录; bin 实用程序子目录; lib 库文件子目录; etc 基本数据和维护实用程序子目录; tmp 临时文件子目录; usr 通用目录。在 usr 下通常包含有一个已安装的文件系统,包括: 小型化的 bin、小型化的 tmp、小型化的库文件 lib 包括文件 include 及各用户的多种文件。此外, UNIX 操作系统自身也在根目录下。图 6-3 为 UNIX 树形目录结构,以方框代表目录文件,未加框者代表一般文件或特殊文件。

目录的查找或检索是文件系统的一项重要工作,每当打开文件时,必须按照申请者给出的文件名字搜索文件目录树中的各级文件目录,直到找出该文件以便实现按名存取。在采

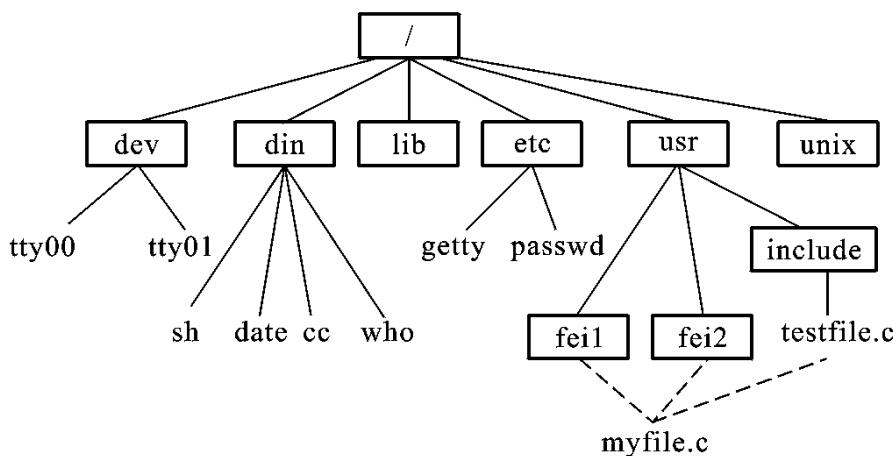


图 6-3 UNIX 树形目录结构

用树形目录结构的文件系统中,要根据用户提供的文件路径名,从根目录开始,逐级查找路径名中的各子目录名,用它们作为索引,逐层搜索各个目录文件,最后找到匹配的文件目录项。

下面是目录查找的一个例子。假设用户进程想要打开文件/usr/fei1/myfile.c,看一下文件系统为进程搜索文件的过程。文件系统搜索这个文件名时,首先,遇到根目录“/”,UNIX 的根目录放在磁盘的固定盘块中,根据内存活动 i-node 表中根目录的活动 i-node,把它作为当前的工作索引节点并将其第一个物理块读入内存缓冲区,接着读入路径的第一个分量字符串 usr。文件系统对根目录文件内容进行搜索,若找不到依次读入第二、第三个物理块、…、进行比较,直到找着 usr 的 i-node 号;查内存的活动 i-node 表中找不到此 i-node,为它在此表中分配一个活动 i-node 区,由于每个 i-node 位于磁盘上分配好了的固定位置,很容易从磁盘上 usr 的 i-node 区中装入它的内容。否则直接查找 usr 内存活动 i-node,通过属性查明 usr 为一个目录文件,经核对符合访问权限,把它作为当前的工作索引节点,接着读入路径的第二个分量字符串 fei1。文件系统读入 usr 目录文件的物理块并对内容进行搜索,直到找着 fei1 的 i-node 号,并为它在内存 i-node 表中分配一个活动 i-node 区,从盘的 fei1 的 i-node 区中装入内容(如果已在内存活动 i-node 表中便省去这一步)。通过属性查明 fei1 为一目录文件,经核对符合访问权限,把它作为当前的工作索引节点,接着读入字符串 myfile.c。文件系统读入 fei1 目录文件的物理块并对内容进行搜索,直到找着 myfile 的 i-node 号,为它在内存 i-node 表中分配一个内存活动 i-node 区,从盘的 myfile.c 的 i-node 区中装入内容(如果已在内存活动 i-node 表中便省去这一步)。中间任何一步出错便返回错误信息,由于路径名分析完毕,这时修改 myfile.c 内存活动 i-node 的有关信息,打开文件操作也就结束。

为了节省目录查找时间,可以使用相对路径,其查找的过程与上述过程类似,但不从根目录而从相对路径指明的当前目录开始查找。每个目录在创建时都自动含有两个特殊的

项“.”项给出当前目录的 i-node 号，“..”项给出其父目录的 i-node 号。所以，查找 ../fei1/myfile.c 的过程只是在工作目录中查找“..”项，找到父目录的 i-node 号，并在其中搜索 fei1 目录。

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，在微型机上则多数采用软盘及硬盘。一盘磁带、一张光盘片、一个硬盘分区或一张软盘片都称为一卷。卷是存储介质的物理单位。对于软盘驱动器、光盘驱动器、磁带驱动器和可拆卸硬盘驱动器等设备而言，由于存储介质与存储设备可以分离，所以，物理卷和物理设备不总是一致的，不能混为一谈。一个卷上可以保存一个文件（叫单文件卷）或多个文件（叫多文件卷），也可以一个文件保存在多个卷上（叫多卷文件）或多个文件保存在多个卷上（叫多卷多文件）。

块是存储介质上连续信息所组成的一个区域，也叫做物理记录。块是主存储器和辅助存储设备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，块的长短常常各不相同；对同一类型的存储介质，块的长短也可以不同。有些外围设备由于启停机械动作的要求或识别不同块的特殊需要，两个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

文件的组织是指文件中信息的配置和构造方式，通常应该从文件的逻辑结构和组织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观点出发，研究用户概念中的抽象的信息组织方式，这是用户能观察到的，可加以处理的数据集合。由于数据可独立于物理环境加以构造，所以，称为逻辑结构。一些相关数据项的集合称做逻辑记录，而相关逻辑记录的集合称做逻辑文件。系统提供若干操作以便使用者构造他的文件，这样，用户不必顾及文件信息的物理构造，而只需了解文件信息的逻辑构造，利用文件名和有关操作就能存储、检索和处理文件信息。显然，用户对于逻辑文件的兴趣远远大于物理文件。然而，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采用的存取方法。

文件的逻辑结构分两种形式:一种是流式文件,另一种是记录式文件。流式文件指文件内的数据不再组成记录,只是依次的一串信息集合,也可以看成是无结构的或只有一个记录的记录式文件。这种文件常常按长度来读取所需信息,也可以用插入的特殊字符作为分界。事实上,有许多类型的文件并不需要分记录,象用户作业的源程序就是一个顺序字符流,硬要分割源程序文件成若干记录只会带来操作复杂、开销增大的缺点。因而,为了简化系统,大多数现代操作系统对用户仅仅提供流式文件,记录式文件往往由高级语言或数据库管理系统提供。

记录式文件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意划分的一个信息单位,记录在文件中的排列可能有顺序关系,但除此以外,记录与记录之间不存在其他关系。在这一点上,文件有别于数据库。早期计算机常常使用卡片输入输出机,一个文件由一叠卡片组成,每张卡片对应于一个逻辑记录,这类文件中的逻辑记录可以依次编号。逻辑记录的概念被应用于许多场合,特别象数据库管理系统中已是必不可少的了。如,某单位财务管理文件中,每个职工的工资信息是一个逻辑记录。整个单位职工的工资信息,即全部逻辑记录便组成了该单位的工资信息文件。

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、检索或更新一个逻辑记录。但从用户使用程序设计语言处理信息的角度来看,还可以把逻辑记录进一步划分成一个或多个更小的数据项。以 COBOL 语言为例,数据项是具有标识名的最小的不可分割的数据单位,数据项的集合构成逻辑记录,相关逻辑记录的集合构成了文件。所以,逻辑文件又可被定义为:与同一实体有关的逻辑记录的集合。仍然以一个单位的工资文件为例,可以很好地说明数据项、逻辑记录和逻辑文件的关系。工资信息逻辑文件中有若干逻辑记录,每一个逻辑记录表示一个职工的工资信息,其中每个记录中的职工编号、姓名、应发工资、扣除工资等都是独立的数据项,当然,扣除工资数据项还可进一步划分为扣除房租、扣除水电费、……,水电费又可进一步划分水费、电费等等更低层次的数据项。在 COBOL 语言的数据描述中要对逻辑记录的数据项的类型、位数和层次等详加说明。一个数据项有一个标识名并赋予它某种物理含义,一个数据项还包含一个值,即数据、数据的类型及其表示法。语言处理程序知道逻辑文件中逻辑记录的组织,知道逻辑记录中每个数据项的物理含义,当从文件系统获得一个逻辑记录后,便可自行分解出数据项进行处理。

通常,用户是按照他的特定需要设计数据项、逻辑记录和逻辑文件的。虽然,用户并不了解文件的存储结构,但是他应该考虑到各种可能的数据表示法,考虑到数据处理的简易性、有效性和可扩充性,考虑到某种类型数据的检索方法。因为,对于用户的某种应用来说,一种类型的表示和组织方法可能比另一种更为适合。所以,在设计逻辑文件时,应该考虑到下列诸因素:

- 如果文件信息经常要增、删、改，那么，便要求能方便地添加数据项到逻辑记录中，添加逻辑记录到逻辑文件中，否则可能造成重新组织整个文件。
- 数据项的数据表示法主要取决于数据的用法。例如，大多数是数值计算，只有少量作字符处理，则数据项以十进制或二进制数的算术表示为宜。
- 数据项的数据应有最普遍的使用形式。例如，数据项“性别”，可用 1 表示男性，2 表示女性。在显示性别这一项时，需要把 1 转换成男性，2 转换成女性。如果性别采用西文的‘M’或‘F’或者中文的“男”和“女”分别表示男性和女性时，这种转换是不需要的。
- 依赖计算机的数据表示法不能被不同的计算机处理，但是字符串数据在两个计算机系统之间作数据交换时，是最容易的数据表示法。因而，这类应用尽可能采用字符串数据。
- 一种高级程序设计语言不可能支持所有的数据表示法。例如 FORTRAN 使用的各个分量应是同类型的向量数组，而 COBOL 却可使用不同类型数据项的记录型数据结构。

2. 成组和分解

现在讨论逻辑记录和块之间的关系。由于逻辑记录是按信息在逻辑上的独立含义划分的单位，而块是存储介质上连续信息所组成的区域。因此，一个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，也可以一个物理块包含多个逻辑记录。如果把文件比作书，逻辑记录比作书中的章节，那么，卷是册而块是页。一本名叫《操作系统教程》的书可以是一册（单卷文件），也可以为多册（多卷文件），当然，也允许《操作系统教程》及《操作系统习题和实习题》两本或多本书装成一册（多文件卷）或多册（多卷多文件）。书中的一个章节占一页或多页，也允许一页中包含若干章节。书和章节相当于文件和逻辑记录，它们是逻辑概念；而册和页相当于卷和块，它们是物理概念，两者不能混淆。

若干个逻辑记录合并成一组，写入一个块叫记录成组，这时每块中的逻辑记录的个数称块因子。成组操作一般先在输出缓冲区内进行，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记录读进输入缓冲区后，把逻辑记录从块中分离出来的操作叫记录的分解。例如，对于穿孔卡片，通常逻辑记录长为 80 个字符，如果把存储介质上的数据块也划分成 80 字节长为一块。那第一张卡片的 80 个字节的数据是一个逻辑记录，也是一个物理记录。在这个例子中，逻辑记录和物理记录是等长的。假定把卡片上的数据写到磁带上，可以规定磁带上的物理记录为 800 字节，这样，每块内就可放 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，可以规定磁盘存储介质的物理记录长为 1 600 字节，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记录长，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，还能减少输入输出操作次数，提高系统效率。记录成组和分解的处理过程如图 6-4 所示，当记录成组和分解处理时，用户的第一个读请求，导致文件管理将包含逻辑记录的整个物理块读入主存系统输入缓冲区，使用户获得所需

的第一个逻辑记录。随后的读请求可直接从输入系统缓冲区取得相继的逻辑记录,直到该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。用户写请求的操作过程相反,开始的若干命令仅将所处理的逻辑记录依次传送到输出缓冲区装配。当某一个写请求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,将该缓冲区的内容写到存储介质的相应块中。

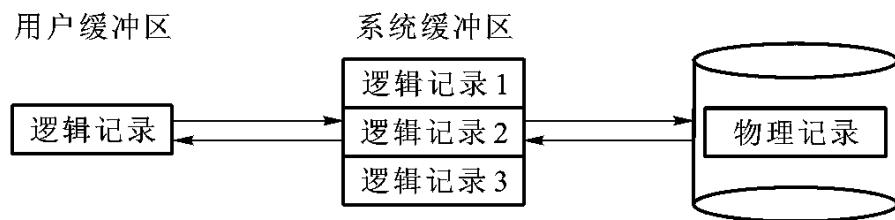


图 6-4 记录成组和分解的处理过程

采用成组和分解方式处理记录的主要缺点是: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输入输出缓冲区。

3. 记录格式和记录键

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到数据处理和各种应用、输入输出传输效率、存储空间利用率和存储设备硬件特点等多种因素。为了存储、检索、处理和加工,必须按可接受的类型和格式把信息提交给操作系统。一个逻辑记录中所有数据项长度的总和称为该逻辑记录的长度。

现将前面已经介绍过的术语,加以小结:

- 逻辑记录——文件中按信息在逻辑上的独立含义划分的一种信息单位。它是可以用一个记录键标识的相关信息的集合,被操作系统看作为一个独立处理的单位,应用程序往往可分解逻辑记录成若干数据项进行相应处理。
- 物理记录——存储介质上连续信息所组成的一个区域,它是主存储器和辅助存储设备进行信息交换的物理单位。综合考虑应用程序的需要、存储设备类型等因素由操作系统设定块长。
- 存储记录——指附加了操作系统控制信息的逻辑记录,它被文件管理看作一个独立处理单位。存储记录除了包含与逻辑记录相同的内容外,还增加了系统描述和处理记录所需要的信息。

系统应将存储记录映射成用户可接受的逻辑记录格式和存储设备上能存储的物理记录格式。图 6-5 表明了三种记录之间的关系。

下面讨论记录的格式和结构。记录格式就是记录内数据的排列方式,显然,这种方式要保证能为操作系统接受。在记录式文件中,记录的长度可以为定长的或变长的,这取决于应

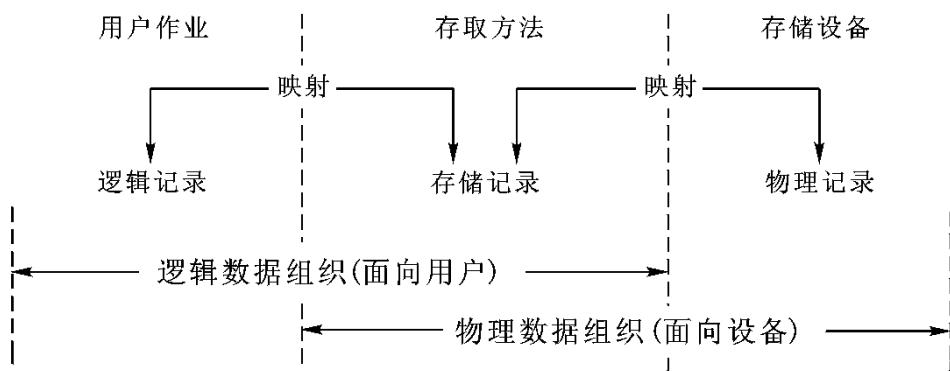


图 6-5 逻辑记录、存储记录和物理记录之间的关系

用程序的需要。用户可以选择下列一种或多种记录格式：

- 格式 F: 定长记录
- 格式 V: 变长记录
- 格式 S: 跨块记录

记录格式主要由用户按所处理数据的性质来选取，用户着眼于逻辑记录结构，然而，输入输出设备的物理特性会影响使用的记录格式。为了处理记录，系统必须获得记录的有关信息。在有些情况下，这些信息来自应用程序，其他场合则由系统自动提供。

定长记录（格式 F）指一个记录式文件中所有的逻辑记录都具有相同的长度，同时所有数据项的相对位置也是固定的。定长记录由于处理方便、控制容易，在传统的数据处理中被普遍采用。定长记录可以成组或不成组，成组时除最末一块外，每块中的逻辑记录数为一常数。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，操作系统能发现并加以处理。

变长记录（格式 V）指一个记录式文件中，逻辑记录的长度不相等，但每个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。
- 包含了可变数目的定长数据项。

虽然可变长记录处理复杂，但却有很大优点，定长记录的记录长度应设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，那么，如采用变长记录则能节省存储空间。为了对变长逻辑记录进行存取，逻辑记录被附加控制信息后，扩展成存储记录。对每个逻辑记录而言，控制信息 RL 为记录长度，指示单个变长记录的字节个数（也包括了控制信息 RL 本身的高度）。对于每个物理记录而言，控制信息 BL 为块长度，指示物理记录中包括控制信息在内的字节总数。

通常，存储介质上的块划分成固定长度后，当处理的可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（格式 S）。在跨块记录的情况下，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不需用户承担,而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。另一个应用例子是在正文编辑处理中,可存放非常长的正文行记录。跨块记录是可变长记录处理的一种延伸,其主要差别是:变长记录处理时,程序员必须知道输入输出缓冲区的大小,而跨块记录处理时却没必要。文件系统将自动分割和装配跨块逻辑记录的信息段,而这些块不会超过输入输出缓冲区的容量。

跨块记录的结构和变长记录完全相同。当应用程序中指明处理跨块记录后,根据需要可将逻辑记录分割成若干段输出,或在输入时将一段或多段再装配成一个逻辑记录。一个完整的逻辑记录可以全部写入一块中,也可以写到相邻的若干块中。前者一段就是一个逻辑记录,后者若干段组成一个逻辑记录。

为了方便文件的组织和管理,提高文件记录的查找效率,通常,对逻辑文件的每个逻辑记录至少设置一个与之对应的基本数据项,利用它可与同一文件中的其他记录区别开来。这个用于标识某个逻辑记录的数据项,称为记录键,也叫关键字,简称键。能惟一地标识某个逻辑记录的记录键,称为主键。例如,

- 工资信息记录中的职员编号;
- 住房信息记录中的住址;
- 订货信息记录中订货号;
- 银行存款信息记录中的存折号;

都可用作相应记录的主键。一个逻辑文件中,主键是最重要的,但它未必是惟一的,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工编号,也允许使用职工姓名作为主键。

如果采用单键记录,即不同的逻辑记录仅设置一个不同的键,它就能惟一地标识这个记录。因而,存取一个记录的信息不必根据内容或记录地址,只需按照记录键就可实现。在很多应用场合,要求多个键对应于同一记录,按其中的任何一个键均可搜索到此记录,这叫多键记录。事实上一个记录中的任一数据项或若干数据项的组合均可作为记录键,总可以为一个记录找到若干个键。除主键外的其他键都称做次键。次键一般不能在若干记录中将特定记录惟一地标识出来。例如,工资信息文件,可将应发工资作为记录键,但月薪 1 000 元的职工却不止一个人,因而,它可当作次键,而不能作为主键使用。多键记录类似于图书馆给图书编索引,可根据书名、书号、作者名和主题分别编目,最终都可找到同一本书。使用多键记录,可以从不同键找到同一个记录,这就引出了有广泛用途的倒排文件的概念。记录键常为一字符串,由使用者提供。最简单的记录键可以和逻辑记录号对应,这时就简化成一个

数字串。

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。用户看到的是逻辑文件，处理的是逻辑记录，按照逻辑文件去存储、检索和加工有关的文件信息，也就是说数据的逻辑结构和组织是面向应用程序的。然而，这种逻辑上的文件总得以不同方式保存到物理存储设备的存储介质上去，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，文件看作为物理文件，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。因而，其优劣直接影响文件系统的性能。

有两类方法可用来构造文件的物理结构。第一类称计算法，其实现原理是设计一映射算法，通常用线性计算法、杂凑法等，通过对记录键的计算转换成对应的物理地址，从而，找到所需记录。直接寻址文件、计算寻址文件、顺序文件均属此类。计算法的存取效率较高，又不必增加存储空间存放附加控制信息，能把分布范围较广的键均匀地映射到一个存储区域中。第二类称指针法，这类方法设置专门指针，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、倒排文件等均属此类。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排列次序完全分开，便于随机存取，便于更新，能加快存取速率。但使用指针要耗用较多存储空间，大型文件的索引查找要耗用较多处理器时间，所以，究竟采用哪种文件存储结构，必须根据应用目标、响应时间和存储空间等多种因素进行权衡折中。下面介绍常用的几种文件的物理结构和组织方法。

1. 顺序文件

将一个文件中逻辑上连续的信息存放到存储介质的依次相邻的块中便形成顺序结构，这类文件叫顺序文件，又称连续文件。显然，这是一种逻辑记录顺序和物理记录顺序完全一致的文件，通常，记录按出现的次序被读出或修改。

一切存于磁带上的文件都只能是顺序文件，此外，卡片机、打印机、纸带机上的文件也属于这一类。这是一种最简单的文件组织形式，在数据处理历史上最早使用。而存储在磁盘或软盘上的文件，也可以组织成顺序文件。顺序文件可由文件目录指出存放该文件信息的第一块存储地址和文件长度。为了改善顺序文件的处理效率，用户常常对顺序文件中的记录按某一个或几个数据项的值从小(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。

顺序文件的基本优点是：顺序存取记录时速度较快。所以，批处理文件、系统文件用得最多。采用磁带存放顺序文件时，总可以保持快速存取的优点。若以磁盘作存储介质时，顺

序文件的记录也按物理邻接次序排列,因而,顺序的盘文件也能象带文件一样进行严格的顺序处理。然而,由于多个程序访问时,在同一时间段内另外的用户程序可能驱动磁头移向了其他文件,因而,不大可能快速地访问到下一个要处理的记录。顺序文件的主要缺点是:建立文件前需要能预先确定文件长度,以便分配存储空间;修改、插入和增加文件记录有困难;对直接存储器作连续分配,会造成空闲块的浪费。

逻辑记录连续地存储在存储介质的相邻物理块上的文件也叫紧凑顺序文件。当插入和修改记录时,往往导致移动大部分记录,为了克服这一严重缺点,对直接存取存储器,可以采用许多顺序文件的变种。扩展顺序文件是在文件内设有空白区域以备预先估计的添加记录使用,这样做虽需较大的存储容量,但在一定程度上适应了文件的扩展性。连接顺序文件中设置溢出区,添加的记录通过连接方法保存到溢出区中,所以对逻辑记录来说是顺序的但文件的存储结构已不一定顺序排列。划分顺序文件是在直接存取设备上模拟顺序文件的一种方法。把一个文件划分成几个能独立存取的顺序子文件。每个文件由数据区和索引区组成,数据区内存放各个子文件,索引区内存放各个子文件的索引,包括名字、地址、长度和状态等,空闲区也由索引指明。划分顺序文件本质上是顺序文件,它有能直接存取各子文件的优点,特别适用于程序库和宏指令库等系统文件的使用。

2. 连接文件

连接结构的特点是使用连接字,又叫指针来表示文件中各个记录之间的关系。如图 6 – 6 所示,文件信息存放在外存的若干个物理块中,第一块文件信息的物理地址由文件目录给出,而每一块的连接字指出了文件的下一个物理块位置。通常,连接字内容为 0 时,表示文件至本块结束。这种文件叫连接文件,又称串联文件,像输入井、输出井等都用此类文件。

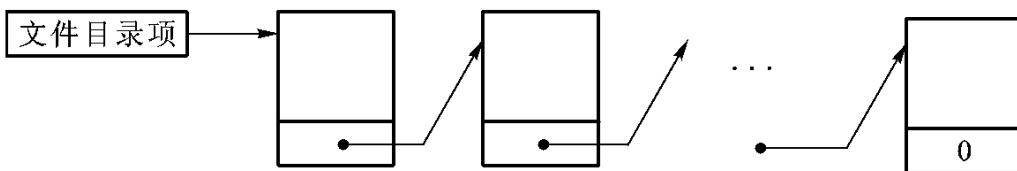


图 6 – 6 连接文件结构示意图

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。使用指针可以将文件的逻辑记录顺序与它所在存储空间的物理记录顺序完全独立开来、即存放信息的物理块不必连续而借助于指针表达记录之间的逻辑关系;此外,必须将连接字与数据信息存放在一起,而破坏了物理块的完整性;由于存取信息须通过缓冲区,待获得连接字后,才能找到下一物理块的地址,因而,仅适宜于顺序存取。连接结构恰好克服了顺序结构不适宜于增、删、改的缺点,对某些操作带来好处,但其他方面又失去了一些性能。

在系统软件中,常常使用某些连接存储结构,它们按记录之间的相对线性位置进行组织。按其插入和删除记录操作的方式不同,可以分成三种情况:

- 堆栈——其所有记录的插入和删除操作只能在同一端进行,这一端称栈顶。堆栈是一个“后进先出”型数据结构,这是因为后进入栈的记录,一定比先进入栈的所有记录先退出栈。由于记录之间没有循环或相交的关系,且记录的逻辑顺序与物理顺序相一致,所以,除栈指针,可以不再设置连接字,这时的存放方式退化为顺序结构。对于不能事先估计栈元素大小的场合,难以采用这种方法,这时就在每个记录中增加指向前一个记录地址的连接字,这就是堆栈的连接存储法。堆栈运算有特殊的名称,把一个新的记录插入栈中,使之成为栈的新顶项叫下推运算;反之,删除栈顶记录叫上推运算,大多数上推需要读取顶项记录以便运算。

- 队列——其记录的插入在后端进行,而删除在前端进行,又叫“先进先出”型数据结构。这是因为从时间上看,先进入队列的记录总比后进入队列的记录先退出队列。为了进行排队及实现入队和出队操作,可采用复杂的连接结构,常用的有:前向、双向和环状指针。前向指针,指每个记录中含有下一个记录的地址,大多数使用的是相对地址。双向指针,指每个记录中除保留有下一个记录的地址外,还含有上一个记录的地址,即同时包含前向和后向指针。使用双向指针的优点是:便于双向搜索;队列中任意记录出队后,便于剩余记录构成连接。环状指针,指首尾两个记录也分别用后向指针和前向指针连接起来的结构。

- 两端队列——左右两端均可进行插入和删除记录操作的队列。

上述连接存储结构在操作系统和语言编辑中用得很多,例如,用作符号运算栈、信息保护栈、进程状态队列、井文件等,但通常不提供给应用程序使用。

3. 直接文件

使用连接文件很容易把数据记录组织起来,但是查找某个记录需遍历链接结构,使得效率很低。另外一种技术,称为散列法或杂凑法(hash法)可以提高效率。在直接存取存储设备上,利用hash法把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直接文件或散列文件。采用hash技术需要建立一张hash表,一个hash表是一个指针数组,数组通过索引访问,而找到的指针便指向数据记录。索引是与数据记录有关的关键字或其变换,例如,有一个描述一座城市人口的文件,如果直接使用年龄作为索引,凡年龄相同的人的资料作为一个数据记录,这样建立的hash表可用来快速查找居民的资料。由于一个城市中有许许多多同龄人,所以hash表中的一个指针指向同龄人的资料记录的链接,搜索这些短的链接结构比搜索全体数据记录要快得多。由于这种存储结构是通过指定记录在介质上的位置进行直接存取的,记录无所谓次序,而记录在介质上的位置是通过对记录的键施加变换来获得相应地址。这种存储结构用在不能采用顺序组织方法、次序较乱、又需在极短时间内存取的场合,对于实时处理文件、操作系统目录文件、存储管理的页表查找、编译程序

变量名表等特别有效。例如,在 Linux 中,常常用 hash 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,仅 Linux 的文件管理就使用 hash 法实现各种缓存,包括:inode cache、directory cache、buffer cache,page cache 和 swap cache。

计算寻址结构中较困难的是“冲突”问题。一般说来,地址的总数和可能选择的关键字之间不存在一一对应关系。因此,不同的关键字可能变换出相同的地址来,这就叫“冲突”。一种散列算法是否成功的一个重要标志是将不同键映射成相同地址的几率有多大,几率越小冲突就越小,则此散列算法的性能也越好。解决冲突会增加相当多的额外代价,因而,“冲突”是计算寻址结构性能变坏的主要因素。解决冲突的办法叫溢出处理技术,这是设计散列文件需要考虑的主要内容。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。

计算机寻址结构的一个特例是:把键作为记录的存取地址,这是一种直接了当的散列方法,又叫直接寻址法。如果键的范围和所使用的实际地址范围相等时,这是一种十分理想的存取方法。但大部分情形下,键值范围大大超过所用的地址范围,因而,这种方法仅在要求快速处理的场合采用。

假定有一个文件系统,采用 hash 法来管理文件控制块,以便加快文件目录的查找过程。下面来讨论该 hash 文件的设计过程,首先,要构造一个 hash 函数,它把文件名转换为其文件控制块所在的盘块地址的索引,根据索引找到相应物理块,然后,再把这个物理块读入内存缓冲区,最后,采用线性比较法逐项查找即可。

步 1 构造 hash 函数。假定有效的文件名为 8 个 ASCⅡ 字符,不足时用空格补足 8 个,超过时截取前面 8 位。构造的一个简单 hash 函数为模 2 加 \oplus ,求已知文件名各个 ASCⅡ 字符值的模 2 加值作为该文件名的文件控制块所在物理块在目录文件中的索引 A,若 a_i 为各文件名字符的 ASCⅡ 字符值,那么,

$$A = (a_1 \oplus a_2 \oplus \cdots \oplus a_8)$$

步 2 建立目录文件。目录文件如图 6-7 那样采用索引结构,建立文件时由步 1 求出文件名的 hash 值 A,凡 A 值相同的文件的文件控制块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。

步 3 查找文件。根据给定的文件名,由步 1 算出该文件名的文件控制块所在物理块号在索引表中的相对位置 A。根据 A 就可以找到该文件控制块所在物理块号,把这个物理块读入内存缓冲区,再用文件名逐个比较,找出要求的文件控制块。

步 4 溢出处理。一个物理块中存放的文件控制块是有限的,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能容纳的文件控制块时,产生了溢出。产生溢出时,系统再申请一个盘区,该区的物理块号将放在 $A + k$ 的索引表目中, k 是一个位移常数,常常选择质数作为位移常数。如第二块盘区也溢出,则申请第三块,块号放在 $A + 2 \times k$ 表目中,依

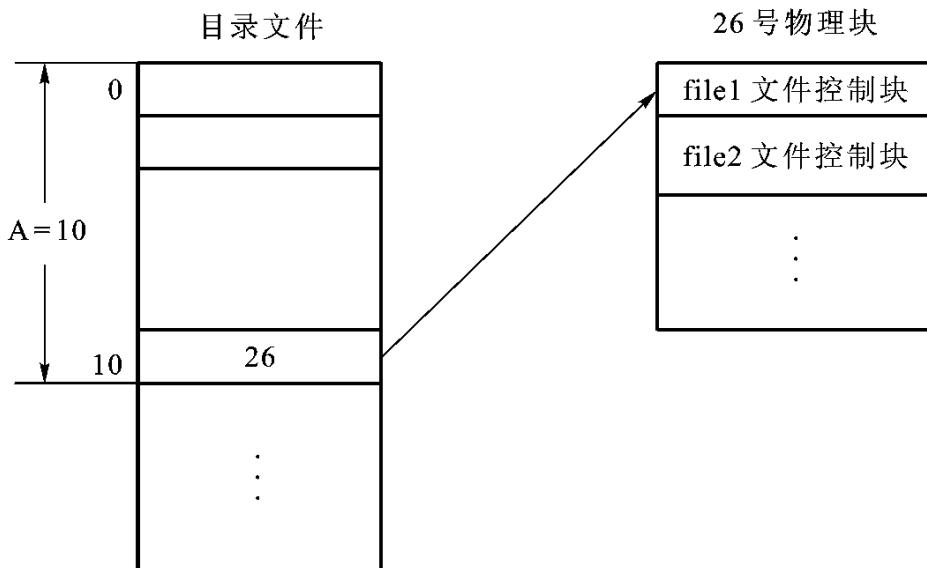


图 6-7 目录文件结构

此类推。而在查找目录时,如第一块找不到可找 $A + k$ 表目中的物理块号,读出后继续比较,依此类推。

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随机存取存储设备上的文件。如图 6-8 所示,系统为每个文件建立了一张索引表,其中,每个表目包含一个记录的键(或逻辑记录号)及其记录数据的存储地址,存储地址可以是记录的物理地址,也可是记录的符号地址,这种类型的文件称索引文件。通常,索引表的地址可由文件目录指出,查阅索引表先找到的是相应记录键(或逻辑记录号),然后,获得数据存储地址。

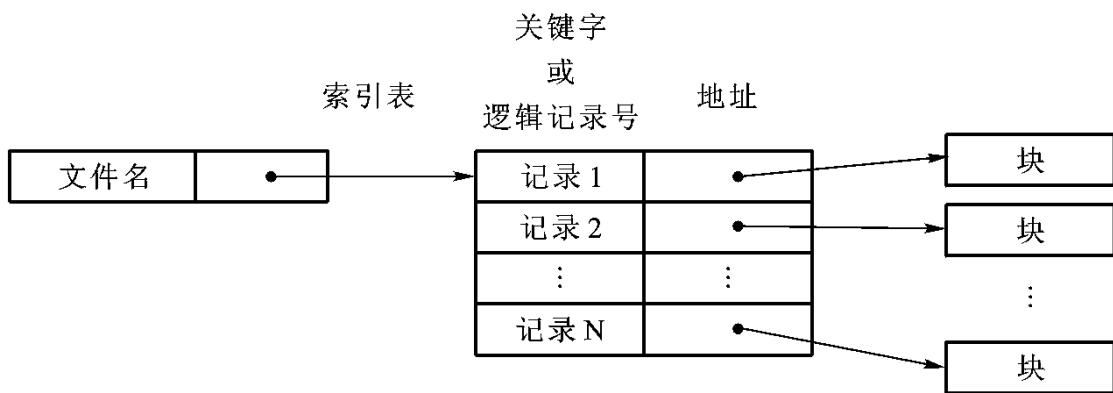


图 6-8 索引文件结构示意图

索引文件在文件存储器上分两个区:索引区和数据区。访问索引文件需两步操作:第一步查找文件索引,第二步以相应键(或逻辑记录号)登记项内容作为物理或符号地址而获得

记录数据。这样,至少需要两次访问辅助存储器,但若文件索引已预先调入主存储器,那么,就可减少一次内外存信息交换。

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外,还克服了它只能作顺序存取的缺点,具有直接读写任意一个记录的能力,便于文件的增、删、改。索引文件的缺点是:增加了索引表的空间开销和查找时间,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。

索引文件中的索引项可以分为两类:一类稠密索引,即对每个数据记录,在索引表里都有一个索引项。因而,索引(表)本身很大,但可以不要求数据记录排序,通过对索引的依次查找就可确定记录的位置或记录是否存在。另一种称稀疏索引,它对每一组数据记录有一索引项。因而,索引表本身较小,但数据记录必须按某种次序排列。注意,虽然稀疏索引本身较小,但是,在查找时又要花出一定代价。因为找到索引之后,只判定了记录所在的组,而该记录是否存在?是组内哪一个记录?还要进一步查找。

索引顺序文件是顺序文件的扩展,其中各记录本身在介质上也是顺序排列的,例如,按字母次序排列,它包含了直接处理和修改记录的能力。索引顺序文件能象顺序文件一样进行快速顺序处理,既允许按物理存放次序(记录出现的次序);也允许按逻辑顺序(由记录主键决定的次序)进行处理。

有时记录数目很多,索引表要占用许多物理块。因此,在查找某键对应的索引项时,可能依次需交换很多块。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 很大时,这是很费时间的操作。提高查找速率的另一种办法是:做一个索引的索引,叫二级索引。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键值(或逻辑记录号)及该索引表的物理地址,也就是说,若干个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物理地址,再搜索一级索引表找出数据记录。当记录数目十分大时,索引的索引也可能占用许许多多块,那样,可以做索引的索引的索引,叫三级索引。有的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完成。

UNIX/Linux 操作系统的多重索引结构稍有不同。如图 6-9 所示,每个文件的索引表规定为 13 个索引项,每项 4 个字节,登记一个存放文件信息的物理块号。由于 UNIX/Linux 文件系统仅提供流式文件,无记录概念,因而,登记项中没有键(或逻辑记录号)与之对应。前面 10 项存放文件信息的物理块号,叫直接寻址,而 0 到 9,可以理解为文件的逻辑块号。如果文件大于 10 块,则利用第 11 项指向一个物理块,该块中最多可放 128 个存放文件信息的物理块的块号叫一次间接寻址,每个大型文件还可以利用第 12 和 13 项作二次和三次间接寻址,因为,每个物理块存放 512 个字节,所以 UNIX/Linux 每个文件最大长度达 11 亿字节。这种方式的优点是与一般索引文件相同,其缺点是多次间接寻址降低了查找速度。对分时

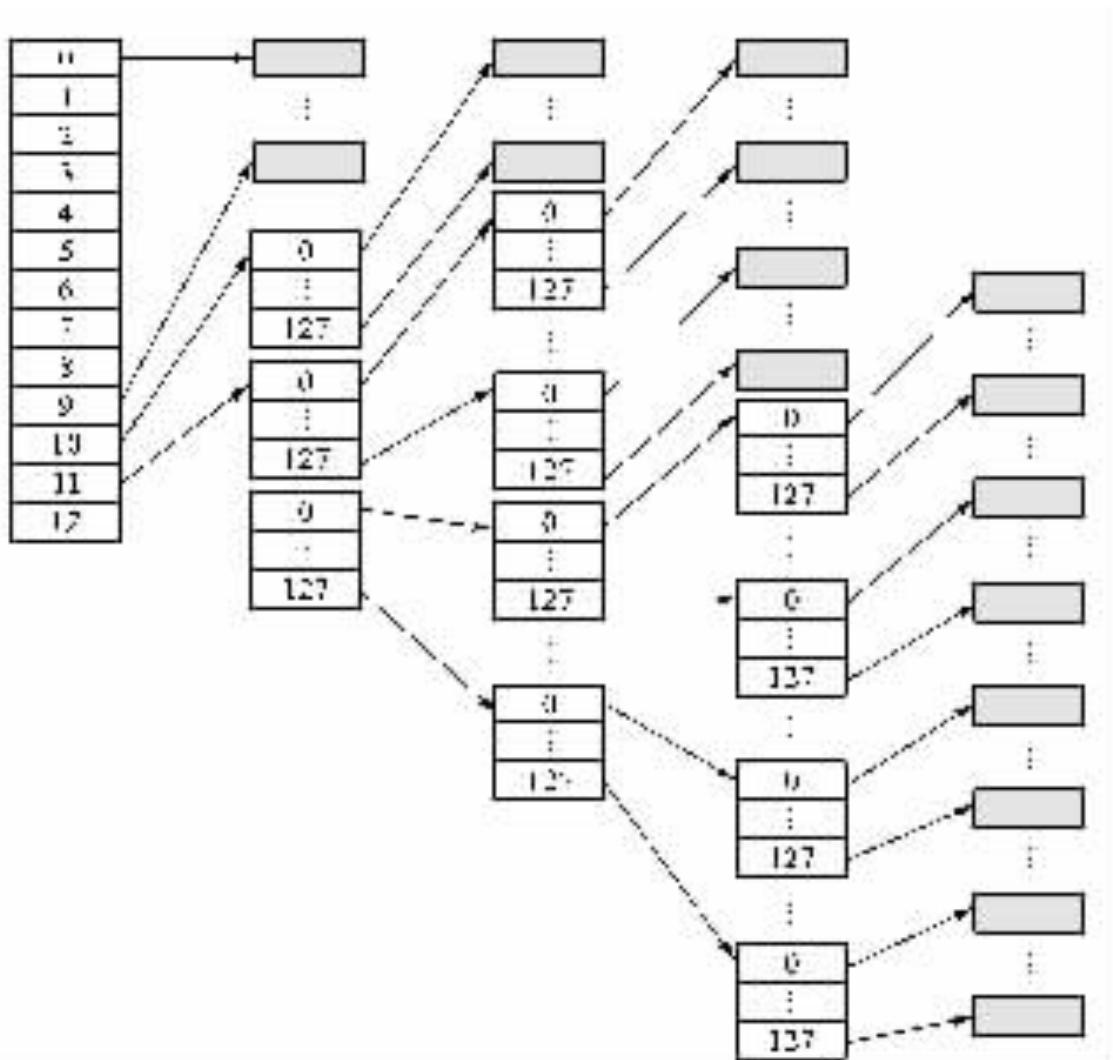


图 6-9 Unix/Linux 的多重索引结构

使用环境统计表明,长度不超过 10 个物理块的文件占总数的 80%,通过直接寻址便能找到文件的信息。对仅占总数的 20% 的超过 10 个物理块的文件才施行间接寻址。

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

文件系统提供给用户程序的一组系统调用,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获得文件系统的各种服务。

文件系统在为用户程序服务时,需要沿路径查找目录以获得有关该文件的各种信息,这往往要多次访问文件存储器,使访问速度大大减慢。若把所有文件目录都复制到主存,访问

速度是加快了,但又增加了主存的开销。一种行之有效的办法是把常用和正在使用的那些文件目录复制进主存,这样,既不增加太多的主存开销,又可明显减少查找时间,系统可以为每个用户进程建立一张活动文件表,当用户使用一个文件之前先通过“打开”操作,把该文件的文件目录复制到指定主存区域。当不再使用该文件时,使用“关闭”操作切断用户进程和该文件目录的联系。同时,若该目录已被修改过,则应更新辅存中对应的文件目录。采用打开文件表的办法之后,每当访问一个文件时,只需先查找活动文件表就可知道文件是否打开,若已打开,就可以对这个文件进行读写操作。这样,一个文件被打开以后,可被用户多次使用,直至文件被关闭或撤销,大大减少访盘次数,提高了文件系统的效率。

1. 建立文件

当用户要求把一批信息作为一个文件存放到存储设备上时,使用建立操作向系统提出建立一个文件的要求。用户使用“建立”系统调用时,通常应提供以下参数:文件名、设备类(号)、文件属性及存取控制信息(如,文件类型、记录大小、保护级别等)。

文件系统完成此系统调用的主要工作是:

- 根据设备类(号)在选中的相应设备上建立一个文件目录,并返回一个用户文件标识,用户在以后的读写操作中可以利用此文件标识;
- 将文件名及文件属性等数据填入文件目录;
- 调用辅存空间管理程序,为文件分配第一个物理块;
- 需要时发出装卷信息(如磁带或可装卸磁盘组);
- 在活动文件表中登记该文件有关信息,文件定位,卷标处理;

在某些操作系统中,可以隐含地执行“建立”操作,即当系统发现有一批信息要写进一个尚未建立的文件中时,就自动先建立文件,完成上述步骤后,接着再写入信息。

2. 打开文件

使用已经建立的文件之前,要通过“打开”文件操作建立起文件和用户之间的联系。打开文件常常使用显式、即用户使用“打开”系统调用直接向系统提出,有的系统可使用隐式,每当读写一个未打开的文件时,意味着由文件系统自动先打开文件。用户打开文件时需要给出文件名和设备类(号)。

文件系统完成此系统调用的主要工作是:

- 在主存活动文件表中申请一个空项,用以存放该文件的文件目录信息;
- 根据文件名查找目录文件,将找到的文件目录信息复制到活动文件表占用栏;
- 若打开的是共享文件,则应有相应处理,如使用共享文件的用户数加1;
- 文件定位,卷标处理;

文件打开以后,直至关闭之前,可被反复使用,不必多次打开,这样做能减少查找目录的

时间,加快文件存取速度,从而,提高文件系统的运行效率。

3. 读/写文件

文件打开以后,就可以用读/写系统调用访问文件,调用这两个操作,应给出以下参数:文件名、主存缓冲地址、读写的字节个数,对有些文件类型还要给出读/写起始逻辑记录号。

文件系统完成此系统调用的主要工作是:

- 按文件名从活动文件表中找到该文件的目录;
- 按存取控制说明检查访问的合法性;
- 根据文件目录指出的该文件的逻辑和物理组织方式将逻辑记录号或字符个数转换成物理块号;
- 向设备管理发 I/O 请求,完成数据传输操作。

4. 关闭文件

当一个文件使用完毕后,使用者应关闭文件以便让其他使用者使用。关闭文件的要求可以通过显式,直接向系统提出;也可用隐式,当使用一个新文件时便可以认为隐含了关闭当前使用的文件的要求。调用关闭系统调用的参数与打开操作相同。

文件系统完成此操作的主要工作是:

- 将活动文件表中该文件的“当前使用用户”减 1;若此值为 0,则撤销此目录;
- 若活动文件表相应表目已被改过,则应先将表目内容写回文件存储器上相应表目中,以使文件目录保持最新状态;
- 卷定位工作。

5. 撤销文件

当一个文件不再需要时,可向系统提出撤销文件。系统调用所需的参数为文件名和设备类(号)。撤销文件时,系统要做的主要工作是:

- 若文件没有关闭,先做关闭工作;若为共享文件,应进行联访处理;
- 在目录文件中删去相应目录项;
- 释放文件占用的文件存储空间。

6.4.2 UNIX 文件系统调用

UNIX 文件系统调用主要包括:文件的打开和关闭,文件的创建和删除,文件的连接和解除连接,文件的读和写,以及文件的随机访问。用户可以通过文件系统提供的系统调用在其程序中对文件进行上述操作,Linux 的文件系统调用种类和功能与 UNIX 大致相同,但实现上有一定差别。下面先介绍 UNIX 文件系统的数据结构及在存储器上的组织安排,然后再介绍文件系统调用。

如图 6-10 所示, 安装在磁盘驱动器上的一个盘组被划分成几个逻辑盘, 每个构成一个文件卷(物理卷), 它又被分成若干柱面、磁道和扇区。每个扇区为 512 个字节(又称一个物

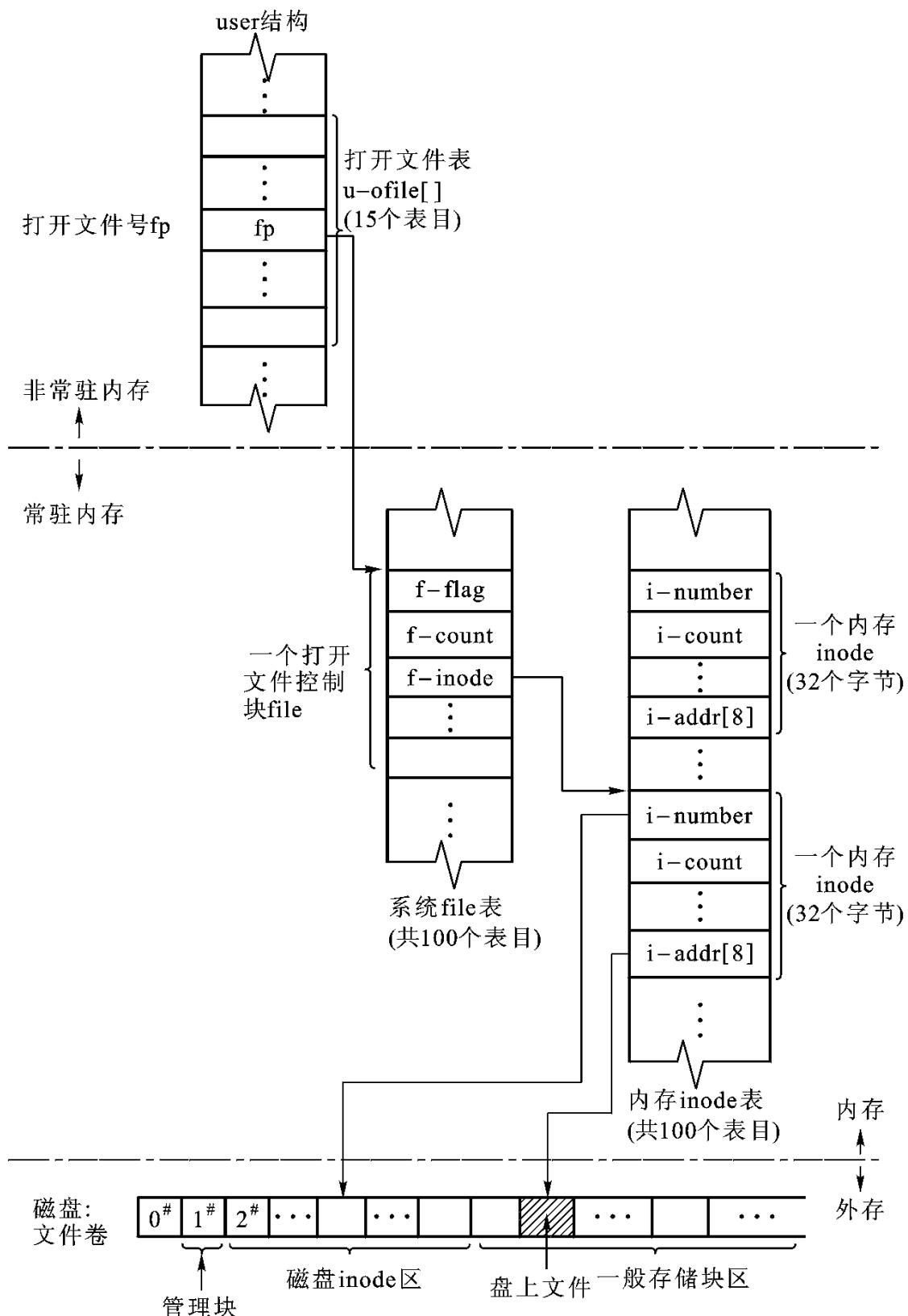


图 6-10 UNIX 文件系统数据结构之间的关系

理块)。系统对物理块从 0 开始统一编号,仅需经过简单换算就可转换成物理地址。其中,0#号块用于系统引导过程;1#号块为专用块(superblock),用于记录文件系统管理用的信息;2#~k+1#共 k 块,用于存放 i 节点(i-node)表;k+2#~n#存放文件信息 n+1#后的若干块作为交换区。UNIX 文件系统的数据结构主要有以下五种:

- 专用块 包括:i 节点表所占盘块数、文件卷盘块数、内存中登记的空闲盘块数(最多 100)、内存中登记的空闲块的物理块号、内存中登记的空闲 i 节点数(最多 100)、内存中登记的空闲 i 节点的编号,以及其他一些文件管理控制信息。可见专用块既有盘位示图的功能,又记录了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,该设备的专用块就要复制到内存专用块中备用,而拆卸文件卷时,修改过的专用块需复制回设备的专用块中。

- i 节点表 存放 UNIX 文件控制块信息,又分外存文件 i 节点表和内存活动 i 节点表,后者解决了频繁访问 i 节点表的效率问题。

- 用户打开文件表 每个进程的 user 结构中保留的整型数组 uOfFile[15]能登记 15 个登记项的用户打开文件表区域。登记的序号为文件描述符 fd,每一登记项登记了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 i 节点。

- 系统打开文件表 为解决多用户共享文件、父子进程共享文件而设置的一个数据结构,内存专门开辟了最多可登记 100 项的系统打开文件表区域。

- 文件卷安装表 用来反映文件卷动态地在根文件设备上装卸情况的数据结构。

下面介绍 UNIX 文件系统调用。

1. 文件的创建和删除

当文件还不存在时,需要创建,或者文件原来已经存在,有时需要重新创建。注意同文件的打开是不同概念,文件打开是指当文件已经存在,需要使用时先执行打开,以便建立用户进程与文件的联系。相应地,文件不再需要时,可以删除之,以便节省存储空间。

(1) 文件的创建

文件创建首先要求文件系统为新的文件建立一个新文件目录项和相应的索引节点 inode,以便随后的写操作为这个新文件输入信息。该系统调用的 C 语言格式为:

```
int fd, mode;
char * filenamep;
fd = creat (filenamep, mode);
```

其中,参数 filenamep 是指向要创建的文件路径名字符串指针;参数 mode 是文件创建者提出的该文件应该具有的存取权限。在文件成功地创建之后,这个权限就记录在相应索引节点的 i_mode 之中。变量 fd 中的内容是当创建成功之后,系统返回给用户的文件描述字,即用户打开文件表项的编号。由此可见,creat 兼有文件的“打开”功能,于是随后的文件写

系统调用,就可以使用这个 *fd* 进行写操作。

例如,用户要创建文件的路径名是 /usr/lib/d2,则用户可用如下的 C 语言程序调用 creat:

```
int fdlib;
fdlib = creat (" /usr/lib/d2 ", 0775);
```

下面简述这一系统调用的执行过程,这里假定文件是首次创建,即在执行之前,文件还未存在:

① 为新文件 d2 分配索引节点和活动索引节点,并把索引节点编号与文件分量名 d2 组成一个新的目录项,记到目录 /usr/lib 中。在这一过程中,需要执行目录检索程序;

② 在文件 d2 所对应的活动索引节点中置初值,包括把存取权限 i_mode 置为 0775,连接计数 i_nlink 置为 1”等等。

③ 为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 d2 所对应的活动索引节点用指针连接起来,最后,把文件描述字返回给调用者。

由于在上述步骤中也执行了文件“打开”功能,因此,在以后操作中,不用再执行“打开”操作就可进行读写。

(2) 文件的删除

删除的主要任务是把指定文件从所在的目录文件中除去。如果没有连接的用户,即如果在执行删除之前 i_link 为 1”,还要把这个文件占用的存储空间释放。文件删除系统调用的形式为: unlink (filenamep), 其中参数与 creat 中的意义相同。在执行删除时,必须要求用户对该文件具有“写”操作权。

2. 文件的打开和关闭

文件在使用前,必须先“打开”,以建立用户进程与文件的联系。其主要任务是把文件的索引节点复制到活动索引节点表中,以便加速对文件的访问。另一方面,活动索引节点表的大小,受到内存容量的限制,这就要求用户一旦当前不再对文件进行操作时,应及时释放相应的活动索引节点,以便让其他进程使用。这就是“关闭”文件的主要功能。

(1) 文件的打开

其调用方式为:

```
int fd, mode;
char * filenamep;
fd = open (filenamep, mode);
```

其中, mode 是打开的方式,它表示时打开后的操作要求,如读(0)、写(1)或又读又写(2)。其余参数的意义与 creat 中的相同。open 的执行过程如下:

① 检索目录,一般来说,要求打开的文件应该是已经创建的文件。它应该登记在文件目录中,否则就出错。在检索到指定文件之后,就把它的外存索引节点复制到活动索引节点表中。

② 把参数 mode 提出的打开方式与活动索引节点中在创建文件时记录的文件访问权限相比较,如果非法,则这次打开失败。

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。通过指针建立这些表项与活动索引节点之间的联系。在完成上述工作之后,把文件描述字,即用户打开文件表中相应文件表项的序号返回给调用者。

需要指出,如果在执行这一调用之前,别的用户已打开了同一文件,则活动索引节点表中已有了这个文件的索引节点,于是在执行这一调用时,不用执行第①步中复制索引节点的工作。而仅把活动索引节点中点的计数器 i_count 加 1”即可。这里 i_count 反映了通过不同的系统打开文件表项来共享同一活动索引节点的进程数目,它是以后执行文件关闭时,活动索引节点能否释放的依据。

(2) 文件的关闭

文件使用完毕,就应该执行关闭系统调用把它关闭,切断用户进程与文件之间的联系。其调用方式为:

```
int fd;
close (fd);
```

显然,要关闭的文件应该是已经打开的,所以,文件描述字 fd 一定存在。 $close$ 的执行过程如下:

① 根据 fd 找到用户打开文件表项,继而找到系统打开文件表项;把用户打开文件表项释放。

② 把对应的系统打开文件表项中的 f_count 减 1”,如果不为“0”,说明进程族中还有进程正在共享这一系统打开文件表项,所以,不用释放系统打开文件表项,而直接返回;否则释放这个系统打开文件表项,并找到与之连接的活动索引节点。

③ 把上述活动索引节点中的 i_count 减 1”,若不为“0”,表明还有其他用户进程正在使用该文件,所以不用释放该活动索引节点而直接返回,否则在把该活动索引节点中的内容复制回文件卷上的相应索引节点中之后,释放该活动索引节点。

由上述过程可见, f_count 和 i_count 分别反映了进程动态地共享一个文件的两种方式。前者反映了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;而后者反映了不同的进程或进程族通过同一个活动索引节点共享一个文件的情况。通过这两种方式,进程之间既可以用相同的位移指针 f_offset ,也可以用不同的 f_offset 来动态地共享同一个文件。

3. 文件的读和写

文件的读和写是文件的最基本操作。“读”是指文件的内容读入到用户进程的变量区中，“写”是指把用户进程变量区中的信息写入到文件存储区中。从文件的什么逻辑位置读入数据，或把数据写入文件的什么逻辑位置均由系统打开文件表中的 `f_offset` 决定。

(1) 读文件

该系统调用的形式为：

```
int nr, fd, count;
char buf [];
nr = read (fd, buf, count);
```

这里 `fd` 是打开系统调用执行之后返回给用户程序的文件描述字；`buf` 是读出信息应送入的用户内存区首地址。`count` 是本次要求传送的字节数，`nr` 是这个系统调用执行之后返回的实际读入字节数。即使在正常情况下，`nr` 指出的字节数也可能小于 `count` 要求的字节数。例如，一旦读到文件末尾时，系统调用就返回，而不管是否已读了用户要求的字节数。如果文件的位移指针已指向文件末尾，又使用了 `read` 系统调用，则返回 0 值。

假定已通过打开系统调用打开了文件 /usr/lib/d2，与它有关的用户打开文件表项，系统打开文件表项和活动索引节点见图 6-11 所示的关系。

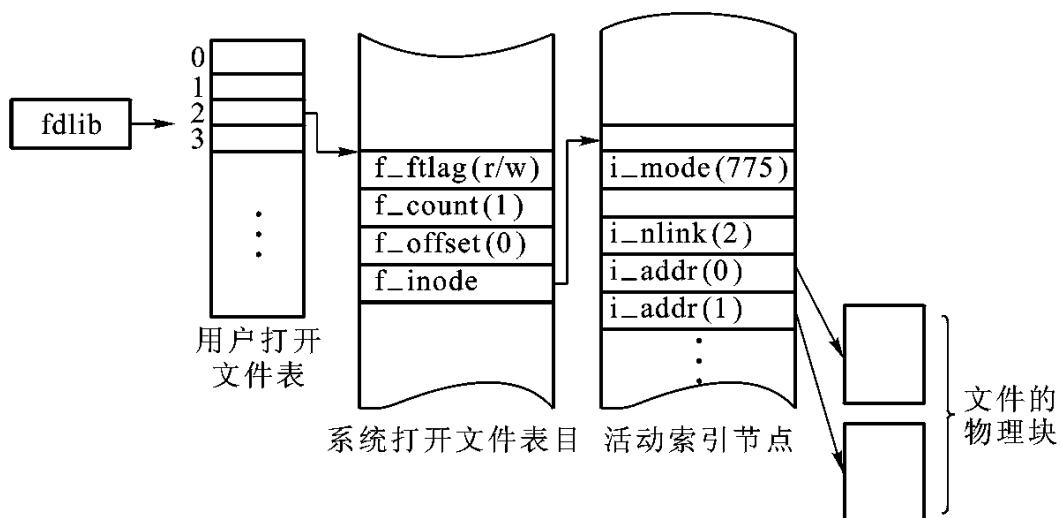


图 6-11 读操作时文件数据结构的关系

现要求读文件 `d2` 的 1500 个字符到指针 `bufp` 指向的用户内存区中，`number` 用来存放实际传送的字节数，则可按如下方式调用 `read`：

```
number = read (fdlib, bufp, 1500);
```

在执行 `read` 系统调用的过程中，系统首先根据 `f_flag` 中记录的信息，检查读操作的合法性。如果合法，则根据当前位移量 `f_offset` 的值，读出要求的字节数，以及活动索引节点

中 *i_addr* 指出的文件物理块存放地址, 把相应的物理块读到块设备缓冲区中, 然后, 再送到 *bufp* 指向的用户内存区中。由此可见, 在执行 *read* 的过程中, 一定要用到块设备管理中的读程序。

(2) 写文件

该系统调用的形式为:

```
    nw = write (fd, buf, count);
```

其中, *fd*, *count* 和 *nw* 的意义类似于 *read*, 只是 *buf* 是信息传送的源地址, 即把 *buf* 所指向的用户内存区中的信息, 写入到文件存储区中。只要情况正常, *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* 清空为零。如果不特别指明, 以后的文件读写操作总是根据 *offset* 的当前值, 顺序地读写文件。为了支持文件的随机访问, 文件系统提供了系统调用 *lseek*, 它允许用户在读、写文件之前, 事先改变 *f_offset* 的指向。这一系统调用的形式为:

```
long lseek;
long offset;
int whence, fd;
lseek (fd, offset, whence);
```

其中, 文件描述字 *fd* 必须指向一个用读或写方式打开的文件, 当 *whence* 是“0”时, 则 *f_offset* 被置为 *offset*; 当 *whence* 是“1”时, 则 *f_offset* 被置为文件当前位置加上 *offset*。

6.4.3 文件卷的安装和使用

文件卷通常又称为文件子系统, 在该卷的空间中不但存放文件和目录信息, 也存放文件属性、空闲区域信息, 以便于文件系统的控制和管理。通常, 存储介质的物理单位为一个卷, 一张格式化后的软盘就是一个卷; 硬盘由于可以从逻辑上进行分区, 每个分区是硬盘上的一个或多个(相邻或不相邻)连续区域, 每个格式化了的分区就是一个卷。不同卷上允许安装不同操作系统(如一个卷装 Windows, 另一个卷装 Linux)。

在 Windows/DOS 操作系统下, 一个盘或分区, 在物理安装和格式化后, 就可以直接使用。使用之前用户不需运行文件卷安装操作, 当更换磁盘即意味着更换了另一个文件子系统时, Windows/DOS 会通过盘符自动识别出来。但还有些操作系统如 UNIX/Linux 却不一样, 每个文件卷需要经过安装后才能使用。下面来讨论 UNIX 操作系统文件卷的动态安装和卸装, 它们的文件系统可分成基本文件系统和可装卸的子文件系统(文件卷)两部分, 见图 6-12。

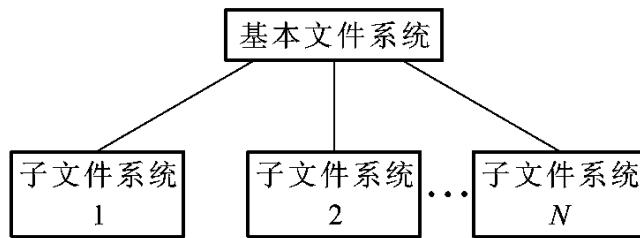


图 6-12 UNIX 文件系统的基本结构

(1) 基本文件系统

基本文件系统固定在根存储设备上,是整个文件系统的基础。通常把硬盘做为根存储设备,系统一旦启动运行,基本文件系统就不能卸装。

(2) 可装卸的子文件系统

存储在可装卸存储介质(如软盘)上的文件系统为可装卸的子文件系统,它可以随时更换。每个用户都可以把自己的文件存放在软盘上,使用时插入软件驱动器,然后,通过系统调用命令将其与基本文件系统安装在一起,也可以用系统调用命令使子文件系统与基本文件系统卸装。

UNIX 文件系统的这种结构,使用灵活,便于扩充和更改。在初始启动时,系统中只有一个安装的文件卷称根文件卷,其上的文件是保证系统正常运行的最小文件集。如汇编语言和 C 语言编译程序、文件卷构造程序、终端命令解释程序等等。这个根文件卷在系统运行过程中是不可卸装的,所以,它是文件系统的基本部分。

其他的文件卷可以根据需要,作为文件子系统动态地安装到一个“已存在”文件卷的某一个节点上。通常,这个节点是为文件卷安装而特地创建的一个空目录。而“已存在”文件卷是指根文件卷,或者是已安装好了的别的文件卷。每个文件卷本身有一个完整独立的目录树结构,所以文件卷的安装就如同树的“嫁接”一样,使安装的文件卷目录树结构与原有的文件卷树形结构融为一体。这一过程如图 6-13 所示,其中(a)为根文件卷中为 rk2 文件卷安装而特地创建一个空目录 usr; (b) 为待安装的文件卷 rk2(例如它可以是一个软盘文件卷); (c) 为动态安装后的文件目录树。

文件卷的动态安装使用系统调用(或命令)mount 完成,上述安装可以使用:

```
mount( "/dev/rk2" , "/usr" , 0 )
```

把软盘 rk2 目录子树安装到根目录下的 usr 子目录下,被安装的这个位置称安装点(mount point)。进程就可以访问安装后的文件子树。

当然,也可以把“安装”上去的文件卷整个地“拆卸”下来,从而恢复“安装”前的状态。文件卷的装卸不同于存储介质(盘、带)的装卸,因为,文件卷卸是从逻辑意义上说的,与文件卷所在的存储介质本身的装卸有本质的不同。即使物理介质本身在工作,但若其上的文件卷

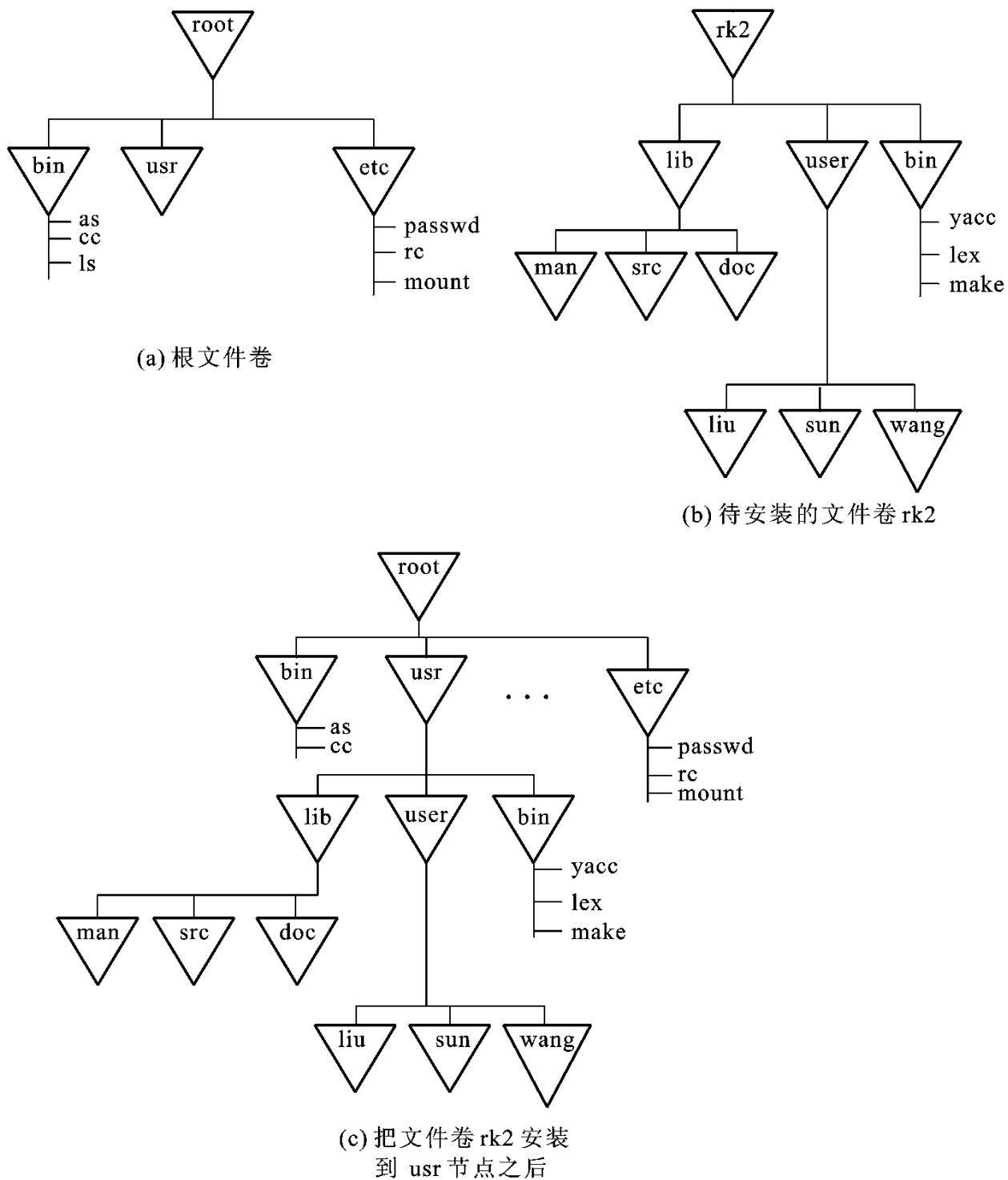


图 6-13 文件卷的动态安装

没有安装好, 系统也无法存取其中的信息。

一旦文件卷安装好, 就要在内核的安装表中登记: 设备号、指向被安装的文件系统专用块的缓冲区指针、指向被安装的文件系统的根 inode 指针 (`/dev/rk2`)、指向安装点超目录的 inode 的指针 (`/usr`) 等, 把这个文件卷的专用块复制到内存专用块中。以后, 当文件操作要进行资源分配和释放时, 就要使用这个内存区。因此, 即使多个文件卷组成了一个统一的树型

结构,但分配和释放资源仍然是各自独立进行的。

文件卷的这种动态装卸有以下优点:

(1) 可以随时扩充文件系统的存储空间。例如,用户又买了一个磁盘时,可以把它构造之后安装到文件目录树结构中。

(2) 可以加强对文件的保护。为了防止一个文件卷上的信息受到有意或无意的破坏,可以把暂时不用的文件卷卸装,从而用户不能访问其上的数据和信息。

6.4.4 文件共享

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的任务所必须,而且还可以节省大量的外存空间,减少由于文件复制而增加的访问外存次数。

文件共享可以有多种形式,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。当一个文件被多个用户程序动态共享使用时,每个程序可以各用自己的读写指针,但也可以共用读写指针。下面用 UNIX 作为例子介绍系统的共享文件方式。

1. 文件的静态共享

有许多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,这种文件在物理上一处存储,从多个目录可到达该文件的多对一关系称做文件链接(file link)。用户在使用文件过程中经常需要在多处使用同一文件,或者多个用户共享同一文件;可以使用不同的路径名来使用同一个文件,也可以使用不同的文件名来使用同一个文件。如果各个用户采用物理拷贝,把同一文件复制到自己的目录下,意味着因冗余而浪费磁盘空间;而且,还有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,其他用户不知道这种修改,最终变成了多个内容不同的文件。在 UNIX/Linux 系统中,两个或多个用户可以通过对文件链接达到共享一个文件的目的。由于这种共享关系不管用户是否正在使用系统,其文件的链接关系都是存在的,因此,称它为静态共享。用文件链接来代替文件的复制可以提高文件资源的利用率,节省文件的物理存储空间。

链接的文件和目录多处出现时,可能由不同的用户或单一用户使用;可能在不同父目录下,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文件(目录)名出现。图 6-3 中,在 fei1 和 fei2 目录下以相同的文件名 myfile.c 出现,但在 include 下虽为同一个文件却以 testfile.c 的文件名出现。

在 UNIX 系统中,是通过文件的 inode 节点来实现文件共享链接的,并且只允许链接到文件而不能链接到目录。文件链接的系统调用形式为:

```
char * oldnamep, * newnamep;
```

```
link (oldnamep, newnamep);
```

其中 oldnamep 和 newnamep 分别为指向已存在文件名字符串和文件别名字符串的指针。这一系统调用的执行步骤如下：

① 检索目录 找到 oldnamep 所指向文件的索引节点 inode 编号。

② 再次检索目录 找到 newnamep 所指文件的父目录文件，并把已存在文件的索引节点 inode 编号与别名构成一个目录项，记入到该目录中去。

③ 把已存在文件索引节点 inode 的连接计数 i_nlink 加 1”。

从上述过程可知，所谓链接，实际上是共享已存在文件的索引节点 inode，完成文件链接的系统调用为：

```
link( "/usr/fei1/myfile.c", "/usr/fei2/myfile.c" );
link( "/usr/fei1/myfile.c", "/usr/include/testfile.c" );
```

执行上述系统调用后，以下三个路径名指的是同一个文件：/usr/fei1/myfile.c, /usr/fei2/myfile.c, 和 /usr/include/testfile.c。

文件的解除链接其调用形式为：unlink (namep)。实际上，UNIX 中解除链接与文件删除执行的是同一个系统调用代码。删除文件是从文件主角度讲的，而解除文件连接是从共享文件的其他用户角度讲的。不论删除还是解除连接，都要删去目录项，把 i_nlink 减 1”。不过，只有当 i_nlink 减为 0”时，才是真正意义上的删除文件。

为了实现文件的静态链接，只要把不同目录的索引节点 i-node 号，指定为同一文件的索引节点即可。但为了反映共享同一文件的用户数，在每个索引节点中设立了一个变量 i_nlink。当文件第一次创建时，i_nlink 等于 1”。以后，每次链接时就把 i_nlink 加 1”。当用户每次删除文件（对于文件主）或解除链接（对其他用户）时，就把该计数值减 1”，直到发现其结果为零时，才释放文件的物理存储空间，从而真正删除这个文件。

2. 文件的动态共享

所谓文件的动态共享，就是系统中不同的用户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用户进程存在时才可能出现，一旦用户的进程消亡，其共享关系也就自动消失。在 UNIX 系统中，文件打开以后，对应的索引节点 i-node 就在活动 i-node 表中，活动 i-node 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，系统在各个进程的 user 结构中设立了一个用户打开文件表，并通过它与各自打开文件的活动的 i-node 相联系。

用户打开文件表与内存活动 i-node 表分离，可以实现不共享文件的当前读写指针的并发共享；用户打开文件表与系统打开文件表分离，可以实现共享文件的当前读写指针的并发共享。当两个程序共享同一文件但读写指针不同时，这两个程序各自的用户打开文件表指向系统打开文件表中的不同表项，而这两个表项又指向内存活动 i-node 表中的同一表项。

当两个程序共享同一文件且共用读写指针时,这两个程序的用户打开文件表指向系统打开文件表中的同一项。

由于 UNIX 系统中的文件采用无结构的字符流序列,因此,文件的每次读写都要由一个位移指针指出要读写的位置。现在的问题是:若一个文件可以为多个进程所共享,那么,应让多个进程共用同一个位移,还是应让各个进程具有各自的读写位移呢?下面分两种情况进行讨论。

在 UNIX 系统中,进程可以动态地创建,被创建的进程完全继承了父进程的一切资源,包括 user 结构中用户打开文件表中指出的打开文件。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,那么,一个进程改变它时,另一个进程能够感觉到它的变化。这样,使父、子进程更容易同步地对文件进行操作。因此,该位移指针宜放在相应文件的活动索引节点中。此时,当用系统调用 fork 函数建立子进程时,父进程的 user 结构被复制到子进程的 user 结构之中,使两个进程的打开文件表指向同一活动的索引节点,达到共享同一位移指针的目的。图 6-14 为 UNIX 共享指针的文件共享。

另一方面,若一个文件为两个以上的用户所共享,必然是每个用户都希望能独立地读、写这个文件,彼此互不干扰。显然,这时不能只设置一个读写位移指针,而必须为每个用户进程分别设置一个读写位移指针。因此,位移指针应放在每个进程用户打开文件表的表目中。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,使两个进程能独立地访问同一文件。图 6-15 为 UNIX 下不共享指针的文件共享。

由此可见,在上述两种动态共享方式中,对读写位移指针的设置位置和数目是不同的。为了解决这一矛盾,系统又设置了一个系统打开文件表。该表共有 100 个表目,为整个系统所公用。在该表的每个表目中,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释放的标志。用户打开文件表、系统打开文件表和活动索引节点表之间的关系已在图 6-14 和图 6-15 中表明。

图 6-15 中,进程 A 和它的子进程通过同一个系统打开文件表表目共享文件 d2,其 f_count 为“2”。而进程 B 自己独立使用一个系统打开文件表表目,其 f_count 为“1”,但它却通过同一个活动索引节点与进程 A 及其子进程共享文件 d2。用户打开文件表表目的序号称为文件描述字,以后进程就是通过文件描述字对相应的文件进行读写操作。下面看一下系统打开文件表目的申请和释放过程。

当一个进程要求打开一个文件时,首先,系统要为它申请一个系统打开文件表表目,并建立该表目与相应文件活动索引节点的联系;然后,把用户打开文件表表目中的一个空闲项指向它。如果在这之后,用户进程通过系统调用 fork 建立一个子进程,系统自动把用户打开

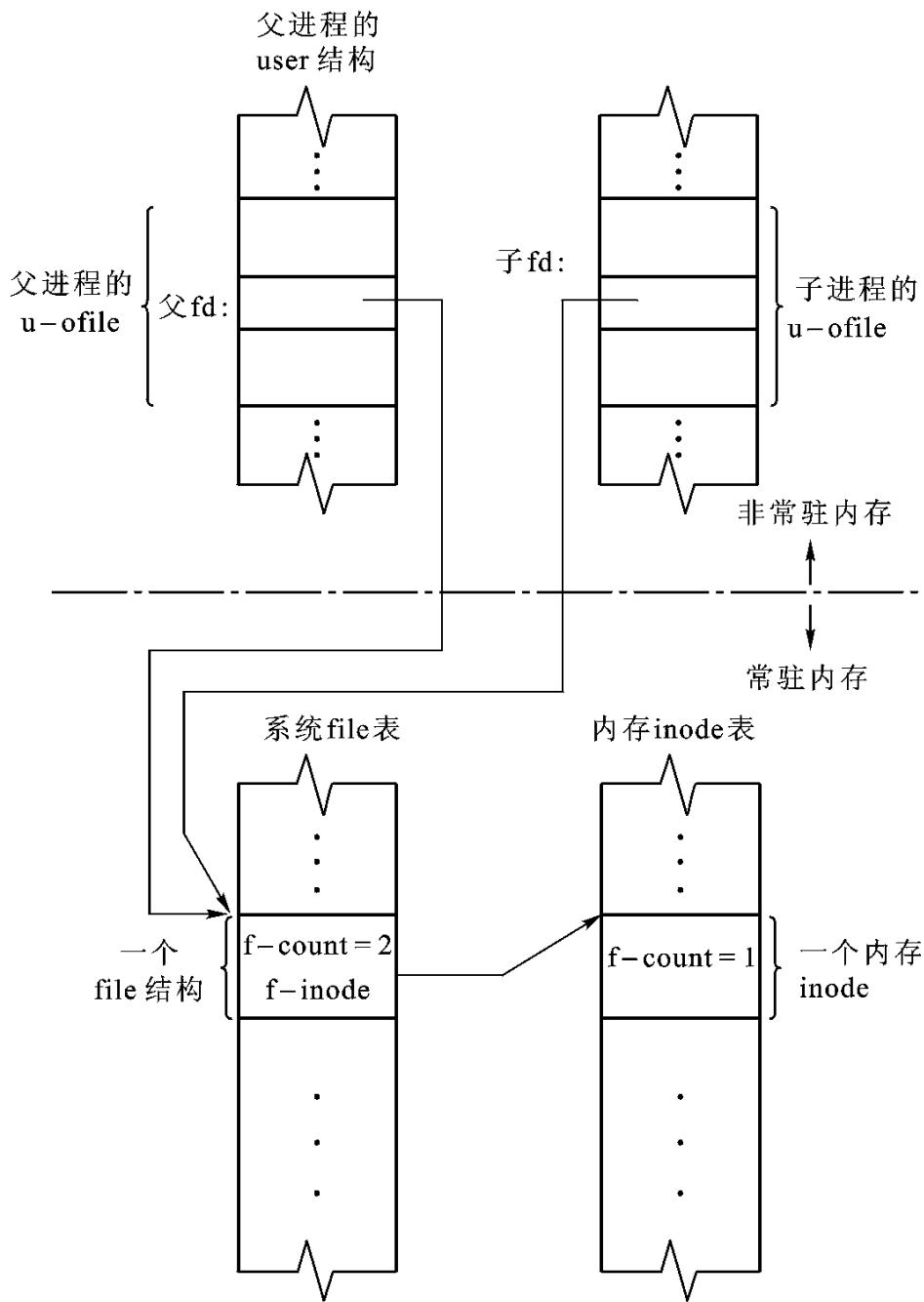


图 6-14 UNIX 共享指针的文件共享

文件表目所指的系统打开文件表目中 `f_count` 加 1”。反之, 当一个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目, 而必须首先判断 `f_count` 的值是否大于 1。如果大于 1, 说明还有进程共享相应的系统打开文件表目, 此时只须把 `f_count` 减 1 即可。由上述过程可知, 进程之间到底采用哪一种方式动态地共享同一文件, 若是由执行“fork”和“打开”系统调用而打开同一名字的文件, 则系统分别为它所分配不同的系统打开文件表目, 并指向同一活动索引节点, 这时, 这两个进程独立地使用各自的读写位移量指针。但如果进程在打开一个文件之后, 又用 fork 系统调用建立了一个子进程, 由于子进程的用户打开文件表表目

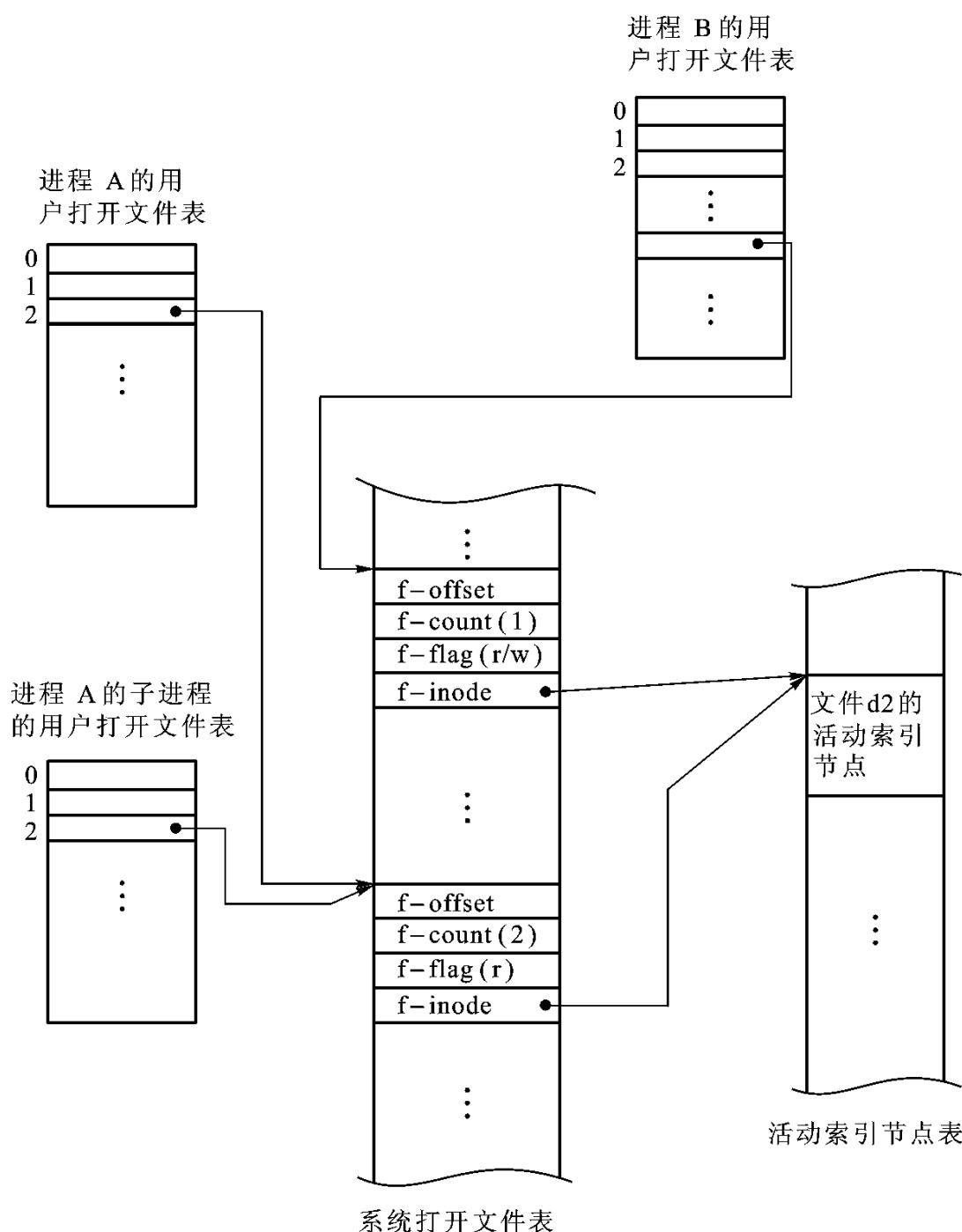


图 6-15 UNIX 不共享指针的文件共享

是由父进程复制得到的, 它自然会指向父进程使用的系统打开文件表表目, 因此, 这两个进程就共用同一个位移量指针来共享一个文件。

这种链接称为硬链接, 只能用于单个文件系统但不能跨文件系统, 可用于文件共享但不能用于目录共享。优点是实现简单、访问速度快。

3. 文件的符号链接共享

另一种链接称符号链接 (symbolic link), 又称软链接, 可以克服上述缺点。符号链接是一

种只有文件名,不指向 inode 的文件。例如,用户 A 为了共享用户 B 的一个文件 F,可以由系统创建一个 LINK 类型的新文件,把新文件写入用户 A 的用户目录中,以实现 A 的目录与 B 的文件 F 的链接。在新文件中只包含被链接文件 F 的路径名,故称这种链接为符号链接,而文件的拥有者才具有指向 i-node 的指针。新文件中的路径名,仅被看作是符号链,当 A 要访问被链接的文件 F 且正要读 LINK 类新文件时,被操作系统截获,它将依据新文件中的路径名去读该文件,于是就实现了用户 A 对用户 B 的文件 F 的共享。符号链接的主要优点是能用于链接计算机网络中不同机器中的文件,此时,仅需提供文件所在机器地址和该机器中文件的路径。这种方法的缺点是:扫描包含文件的路径开销大,需要额外的空间存储路径。

6.4.5 层次式文件系统模型

下面介绍的是由 Madnick 于 1969 年提出来的一个层次式文件系统模型,它不针对某个具体的操作系统,而是介绍文件系统一般的工作原理,从用户发出文件系统调用开始,进入文件系统,直到存取文件存储器上的信息的实现。采用分层方法设计出来的文件系统具有结构清晰、稳定性好、易于维护的优点。这一文件系统模型如图 6-16 所示,大致可以分成下列层次:用户接口、符号文件系统、基本文件系统、存取控制验证、逻辑文件系统、物理文件系统、设备策略模块和 I/O 控制系统。

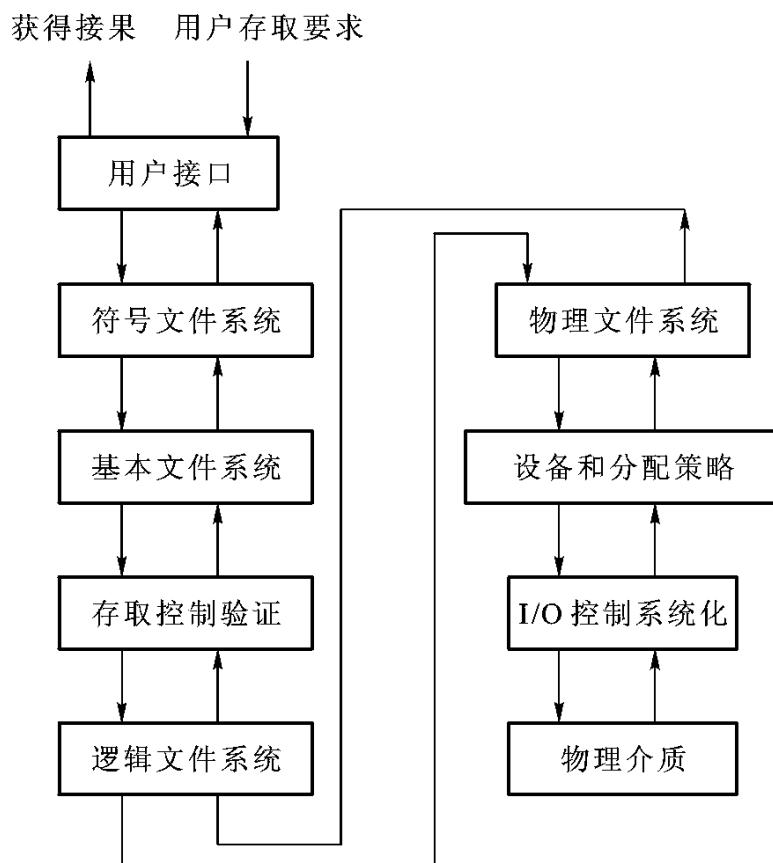


图 6-16 一种文件系统模型

(1) 用户接口接受用户发来的文件系统调用, 进行必要的语法检查, 根据用户对文件的存取要求, 转换成统一格式的内部系统调用, 并进入符号文件系统。

(2) 符号文件系统根据文件名或文件路径名, 建立或搜索文件目录, 获得文件内部惟一标识来代替这个文件, 供后面存取操作使用。

(3) 基本文件系统根据文件内部标识负责把文件说明信息调入到内存的活动文件表中, 这样查找同一表目就不用反复读盘了。如文件已经打开, 则根据本次存取要求修改活动文件表内容, 并把控制传到下一层。

(4) 存取控制验证根据活动文件表相应目录项识别调用者的身份, 验证存取权限, 判定本次文件操作的合法性, 实现文件的存取、共享、保护和保密。如不允许本次访问便发出一个错误条件, 本次文件操作请求失败。

(5) 逻辑文件系统根据文件说明中的文件结构和存取方法等逻辑结构信息, 把指定的逻辑记录地址转换成相应相对的块地址。对于流式文件, 只要把用户指定的逻辑地址按块长计算出相对块号; 对记录式文件, 先把记录号转换成逻辑地址, 再把其转换成相对块号。如本文件适用多种存取方法, 则应设多个例程完成不同的转换算法。

(6) 物理文件系统根据活动文件表相应目录项中的物理结构信息, 将相对块号及块内相对地址转换为文件存储器的物理块号和块内地址。

(7) 设备和分配策略模块负责文件存储空间的分配, 若为写操作, 则动态地为调用者申请物理块; 实现缓冲区信息管理。根据物理块号生成 I/O 控制系统的地址格式。

(8) I/O 控制系统具体执行 I/O 操作, 实现文件信息的存取。这一层属于设备管理功能。

6.4.6 辅存空间管理

在磁盘等大容量辅存空间上, 用户作业运行期间常常要建立和删除文件, 操作系统应能自动管理和控制辅存空间。辅存空间的有效分配和释放是文件系统应解决的一个重要问题。

辅存空间的分配和释放算法是较为简单的, 最初, 整个存储空间可连续分配给文件使用, 但随着用户文件不断建立和撤销, 文件存储空间会出现许多“碎片”。系统应定时或根据命令要求集中碎片, 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。

辅存空间分配常采用以下两种办法:

- 连续分配。文件被存放在辅存空间连续存储区(连续的物理块号)中, 在建立文件时, 用户必须给出文件大小, 然后查找到能满足的连续存储区供使用; 否则文件不能建立, 用户进程必须等待。连续分配的优点是顺序访问时通常无需移动磁头, 文件查找速度快, 管理较为简单, 但为了获得足够大的连续存储区, 需定时进行“碎片”收集。因而, 不适宜于文件

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也无法进行分配。

• 非连续分配。一种非连续分配方法是以块(扇区)为单位,按文件动态要求分配给它若干扇区,这些扇区不一定要连续,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。另一种非连续分配方法是以簇为单位,簇是由若干个连续扇区组成的分配单位;实质上是连续分配和非连续分配的结合。各个簇可以用链指针、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具体文件辅存空间管理方法。

1. 位示图

又称字位映象表,使用若干字节构成一张表,表中每一字位对应一个物理块,字位的次序与块的相对次序一致。字位为‘1’表示相应块已占用,字位为‘0’状态表示该块空闲。微型机操作系统 CP/M、VM/SP、Windows 和 Macintosh 等操作系统均使用这种技术管理文件存储空间。其主要优点是,可以把位示图全部或大部分保存在主存中,再配合现代计算机都具有的位操作指令,可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。删除文件释放空闲区时,把空闲块位置及连续的空闲块数填入空闲区表,如果出现邻接的空闲块,还需执行合并操作并修改登记项。空闲区表的搜索算法有优先适应、最佳适应和最坏适应算法等,这些已经在前面介绍过。

3. 空闲块链

第三种分配方法是把所有空闲块连接在一起,系统保持一个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,使其成为空闲块并将它挂到空闲链上。这种方法效率很低,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。DOS 操作系统采用了这种方案的变种。

4. 成组空闲块链

图 6-17 给出了 UNIX/Linux 采用的空闲块成组连接法。存储空间分成 512 字节一块。为方便讨论,假定文件卷启用时共有可用文件 438 块,编号从 12 至 349。每 100 块划分一组,每组第一块登记下一组空闲块的盘物理块号和空闲总数。其中,50 # – 12 # 一组中,50 # 物理块中登记了一下组 100 个空闲块物理块号 150 # – 51 #。同样下一组的最后一块 150 # 中登记了再下一组 100 个空闲块物理号 250 # – 151 #。注意,最后一组中,即 250 # 块

中第 1 项是 0, 作为结束标志, 表明系统文件空闲块链已经结束。

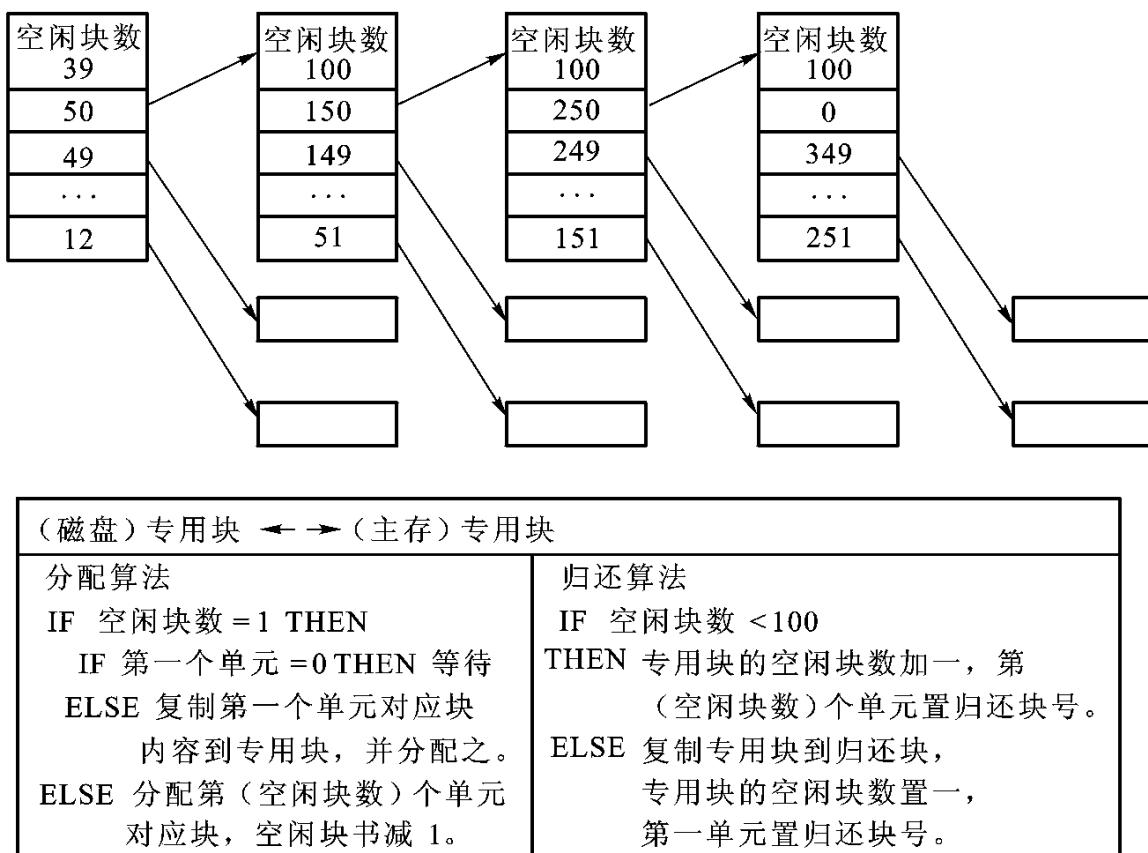


图 6-17 UNIX 系统的空闲块成组连接示意图

当设备安装完毕, 系统就将专用块复制到主存中。专用块指示的空闲块分配完后再有申请要求, 就把下一组空闲块数及盘物理块号复制到专用块中。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, 开始时空闲块是按顺序排列的, 但只要符合分组及组间连接原则, 空闲块可按任意次序排列。事实上, 经过若干次分配、释放操作后, 空闲块物理块号必定不能按序排列了。

6.4.7 内存映射文件

用户进程要使用文件, 通常通过操作系统中提供的一整套文件操作调用, 如打开文件、读写文件和关闭文件等等, 此外, 还要提供缓冲技术和存取方法, 具体做法是把要读/写的文件信息一部分一部分读入进程空间或从进程空间写到文件中。这样处理使得系统调用实现不但管理复杂, 而且开销较大。针对这一点, 最早由 MULTICS 首创通过结合虚存管理和文件管理技术来提供一种新的文件, 称内存映射文件, Windows 2000/XP 等现代操作系统也都实现了这一功能。

内存映射文件(memory-mapped file)允许进程分配一段虚地址空间, 然后, 把某一个盘

文件映射到该地址空间中,反之也可以这样说,指把磁盘中的文件视为进程的虚拟内存的一部分,所以,也叫做“映射文件 I/O。”多个进程读写这个存储区域,而无需做缓冲数据或执行磁盘 I/O 的工作就可以读写文件了。其优点是:方便易用、节省空间、便于共享、灵活高效。通常,内存映射文件功能是由操作系统的 I/O 子系统和存储管理子系统共同实现的。每当一个应用程序访问它的虚拟地址空间的存储区域时,存储管理子系统利用它的页面调度机制从磁盘文件中加载正确的页面。如果应用程序向它的虚拟地址空间写入数据时,存储管理子系统就把更改作为正常页面调度的一部分写回到文件中。

从概念上看,可以设想系统提供了两个新的系统调用,一个叫映射文件 map,它需要有两个参数:一个文件名和一个虚拟地址,把一个文件映射到进程地址空间。另一个叫移去映射文件 unmap,让文件与进程地址空间断开,并把映射文件的数据写回磁盘文件。例如,有一个文件有 64 KB 长,被映射到 512 K 的虚地址处,这样,在 512 K 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 K + 100 地址处写入则修改了文件的 100 字节。当进程开始执行访问文件操作时,例如,从 512 K 开始读出信息,由于此时文件信息尚未调入内存,会引起一个缺页中断,系统便装入文件的页面 0,此后就能顺利进行读操作。类似地,对 512 K + 100 的一个写操作会引起一个缺页中断,系统便装入文件中含有该地址的页面,之后,对存储器的写操作可以开始。假如进程映射文件地址空间中的一个页面被替换算法逐出,它会被写回到磁盘文件的合适位置处。当进程结束后,全部被映射的、被修改的页面写回到磁盘文件中。实际上只需让进程所映射的进程地址空间的页表项中的外存地址指向文件所在的外存块就能很容易实现映射文件和移去映射文件系统调用了。

当多进程共享文件时,使用内存映射文件能节省存储空间,允许每一个进程把同一个文件映射到它自己的虚存空间中。当其中的一个进程修改它自己虚存空间中的数据时,能被其他映射相同文件的进程所看到。其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,而页框中则保存了磁盘文件的拷贝。图 6-18 是内存映射文件示意图,其明显的优点为:进程读写虚存内容相当于执行文件读写操作,在建立映射后,就不再需要使用文件系统调用来读写数据,能大大降低开销;在内存中仅需一个页面副本,既节省空间,又不要缓冲到主存的复制操作。

虽然内存映射文件消除了对 I/O 的需要,编程方便,但也导致了一些问题。首先,系统很难知道输出文件的长度,假设程序只引用了页面 0,其中写了多少个字节系统是不知道的,它所做的只是创建一个同页面同样长度的文件。其次,当一个文件被某个进程映射了,而又被另一个进程以通常的读方式打开了。如果第一个进程修改了一个页面,直到该页面被换出,其修改也不会在磁盘上反映出来。系统必须解决这类数据一致性问题。最后,文件可能很大,甚至比整个进程虚拟地址空间还要大,惟一的办法是安排映射文件系统调用能映射文件的某一部分,而不是整个文件,尽管该方法能够工作,但较之于映射整个文件而言并

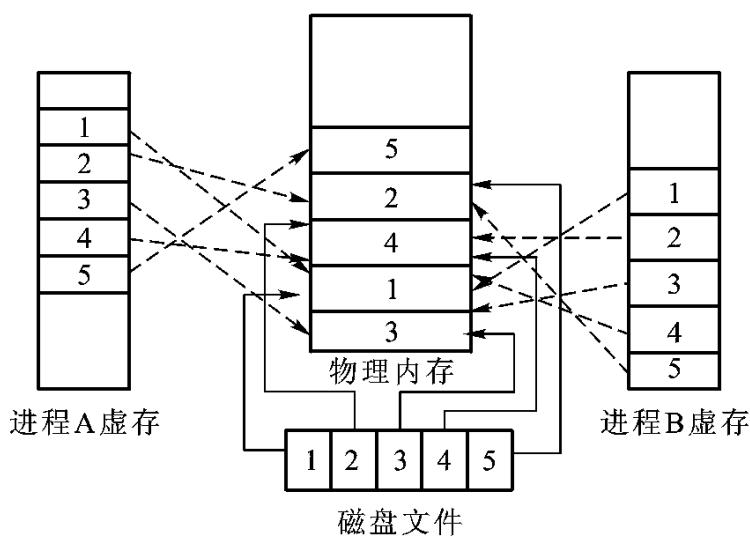


图 6-18 内存映射文件示意图

不令人满意。

6.4.8 虚拟文件系统

传统的操作系统中只设计了一种文件系统,因而,仅能支持一种类型的文件系统。但随着信息技术的发展,对文件系统出现了许多新的要求,例如,要求在 UNIX 系统中支持非 UNIX 文件系统;要求在 Windows 2000/XP 系统中支持新的高性能文件系统的同时支持 FAT 文件系统;Linux 在设计时便瞄准能同时支持多达几十种文件系统;要求现代操作系统都能支持分布式文件系统和网络文件系统;甚至一些用户希望能定制自己的文件系统。解决上述问题可有多种方案,这里介绍了由 AT&T 把 Sun 公司的 vnode/vfs 及其 NFS 技术集成到 UNIX SVR4 中,成为事实上工业标准的虚拟文件系统。

虚拟文件系统要实现以下目标:应同时支持多种文件系统;系统中可以安装多个文件系统,它们应与传统的单一文件系统没有区别,在用户的面前表现为一致的接口;对通过网络共享文件提供完全支持,访问远程结点上的文件系统应与访问本地结点的文件系统一致;可以开发出新的文件系统,以模块方式加入到操作系统中。

虚拟文件系统的主要设计思想体现在两个层次:(1)在对多个文件系统的共同特性进行抽象基础上,形成一个与具体文件系统实现无关的虚拟层,并在此层次上定义与用户的一致性接口;(2)文件系统具体实现层使用类似开关表技术进行文件系统转接,实现各文件系统的具体细节,每个文件系统是自包含的,包含文件系统实现的各种设施,如超级块、i 节点区、数据区以及各种数据结构和对文件的操作函数。

对于一个文件系统来说,最主要的实现要点是 i 节点和文件系统定义。因此,虚拟文件系统首先定义了 v 节点(虚拟节点)和 vfs(虚拟文件系统)。一个 v 节点代表了内核中一个文

件,而 vfs 代表了内核中一个文件系统。

v 节点的抽象信息主要有三部分:数据域(struct vnode),这是与文件系统相关的私有数据,如 v-count、v-data、v-type、v-op 等;虚函数(struct vnodeops),这是 vnodeops 函数的文件系统相关实现,如 vop-open、vop-read、vop-write、vop-mkdir、vop-lookup 等;实用程序及宏,如 vn-open、vn-link 等。vfs 的抽象信息主要有二部分:数据域(struct vfs),这是与文件系统相关的私有数据,如 vfs-next、vfs-data、vfs-op、vfs-fstype 等;虚函数(struct vfsops),这是 vfsops 函数的文件系统相关实现,如 vfs-mount、vfs-root、vfs-unmount、vfs-sync、vfs-statvfs 等。

v 节点和 vfs 定义了多重文件系统框架和接口的抽象层次,把一些文件系统的共同特性抽象出来以形成 v 节点和 vfs。在 UNIX 中 v 节点就是 i 节点,该节点中包含了文件属性信息和文件块索引表。但 i 节点中的文件属性内容与文件分配的方法随文件系统的不同而不同,因此,虚拟文件系统中的 v 节点仅包含共同的内容信息。在一个具体的文件系统中的 i 节点由两部分组成:虚拟 v 节点的内容再加上具体文件系统私有的 i 节点的数据部分。下面列出了 UNIX SVR4 的 v 节点的数据结构定义。

```
Struct vnode
{
    u_short    v_flag;           /* 标志位 */
    u_short    v_count;          /* 访问计数 */
    struct vfs *vfsmountedhere; /* 子文件系统安装点 */
    struct vnodeops *v_op;       /* v 节点操作函数向量 */
    struct vfs *vfs_p;           /* 该 v 节点所属文件系统 */
    struct stdata *v_stream;     /* 指向相关的流设备 */
    struct page *v_page;         /* 驻留页的页表 */
    enum vtype v_type;           /* 文件类型 */
    dev_t      v_rdev;           /* 设备文件的设备 id */
    caddr_t    v_data;            /* 指向公用数据结构的指针 */
}
```

UNIX SVR4 的 v 节点基类的数据域中包含了不依赖于具体文件系统类型的信息,其中一组定义了虚拟文件系统相关接口的虚函数。当进程通过 vfs 接口对某一类型 v 节点调用某一虚函数时,内核会间接调用 v-op,并调用相应的文件系统实现函数。还定义了一组公共的不是虚函数的全局内核函数(或宏)。

vfs 表示一个文件系统,内核为每个活动的文件系统分配一个 vfs 结构,其数据结构如下。

```
struct vnode
```

```

{
    struct vfs    *vfs - next;           /* vfs 链接表指针 */
    struct vfsops  *vfs - op;           /* 操作函数向量 */
    struct vnode   vfs - vnode covered; /* 安装子文件系统的被覆盖的 v 节点 */
    int vfs - fs   type;               /* 文件系统类型索引 */
    caddr - t     vfs - data;          /* 私用数据 */
    dev - t      vfs - dev;            /* 文件系统的设备 id */
}

```

在一个具体的文件系统工作时,文件描述符中包含有一个指向系统活动文件表的指针,而该活动文件表的表目中指出了该文件相应的 v 节点,而 v 节点中包含有与所属文件系统相关操作函数向量和私用数据。

为了正确地将 v 节点和 vfs 的操作定向到对应的具体文件系统的实现上,内核需要一种机制来决定怎样访问每个文件系统的接口函数。UNIX SVR4 使用了虚拟文件系统开关表,这是一个全局表,每一个具体文件系统都占用一个表目,其中包含文件系统(类型)名、初始化程序的地址指针、该 vfs 的操作函数向量等。文件系统的安装通过系统调用 mount 实现,大致过程如下:mount 先通过 lookupn() 来获得安装点目录的 v 节点,确定该 v 节点代表一个目录,且没有其他文件系统在该点安装;mount 查找虚拟文件系统开关表,找出与给定类型名字相同的文件系统(类型)名的表目;内核调用该表目的相关操作对一个特定文件系统执行初始化例程,用来分配文件系统所需的数据结构和资源;内核接着分配一个新的 vfs 结构,且初始化该结构;内核把指向 vfs 结构的指针存放到被覆盖目录们 v 节点的数据域;最后,内核调用虚拟文件系统的 vfs - mount 操作,完成文件系统相关的 mount 调用处理过程。

6.5 实例研究:Linux 文件管理

6.5.1 Linux 文件管理概述

Linux 支持多种不同类型的文件系统,包括 EXT、EXT2、MINIX、UMSDOS、NCP、ISO9660、HPFS、MSDOS、NTFS、XIA、VFAT、PROC、NFS、SMB、SYSV、AFFS 以及 UFS 等。由于每一种文件系统都有自己的组织结构和文件操作函数,并且相互之间的差别很大, Linux 文件系统的实现有一定的难度。为支持上述的各种文件系统,Linux 在实现文件系统时采用了两层结构:第一层是虚拟文件系统 VFS(Virtual File System),它把各种实际文件系统的公共结构抽象出

来,建立统一的以 i-node 为中心的组织结构,为实际文件系统提供兼容性。它的作用是屏蔽各类文件系统的差异,给用户、应用程序和 Linux 的其他管理模块提供统一的接口。第二层是 Linux 支持的各种实际文件系统。

Linux 的文件操作面向外存空间,它采用缓冲技术和 hash 表来解决外存与内存在 I/O 速度上的差异。在众多的文件系统类型中,EXT2 是 Linux 自行设计的具有较高效率的一种文件系统类型它建立在超级块、块组、i-node 和目录项等结构的基础上,本节作简单介绍。

6.5.2 Linux 文件系统安装

同其他操作系统一样,Linux 支持多个物理硬盘,每个物理磁盘可以划分为一个或多个磁盘分区,在每个磁盘分区上就可以建立一个文件系统。一个文件系统在物理数据组织上一般划分成引导块、超级块、i-node 区以及数据区。引导块位于文件系统开头,通常为一个扇区,存放引导程序、用于读入并启动操作系统。超级块由于记录文件系统的管理信息,根据特定文件系统的需要其存储的信息不同。i-node 区用于登记每个文件的目录项,第一个 i-node 是该文件系统的根节点。数据区则存放文件数据或一些管理数据。

一个安装好的 Linux 操作系统究竟支持几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,并在系统初始化时通过函数调用向注册表登记;另一种是把文件系统当作一个模块,通过 kmod 或 insmod 命令在装入该文件系统模块时向注册表登记它的类型。

文件系统数据结构中,file _ systems 指向文件系统注册表,每一个文件系统类型在注册表中有一个登记项,记录了该文件系统类型的名 name、支持该文件系统的设备 requires _ dev、读出该文件系统在外存超级块的函数 read _ super、以及注册表的链表指针 next。函数 register _ filesystem 用于注册一个文件系统类型,函数 unregister _ filesystem 用于从注册表中卸装一个文件系统类型。

每一个具体的文件系统不仅包括文件和数据,还包括文件系统本身的树形目录结构,以及子目录、链接、访问权限等信息,它还必须保证数据的安全性和可靠性。

Linux 操作系统不通过设备标识访问某个具体文件系统,而是通过 mount 命令把它安装到文件系统树形目录结构的某一个目录节点,该文件系统的所有文件和子目录就是该目录的文件和子目录,直到用 umount 命令显式的撤卸该文件系统。

当 Linux 自举时,首先装入根文件系统,然后根据/etc/fstab 中的登记项使用 mount 命令逐个安装文件系统。此外,用户也可以显式地通过 mount 和 umount 命令安装和卸装文件系统。当安装/卸装一个文件系统时,应使用函数 add _ vfsmnt/remove _ vfsmnt 向操作系统注册/注销该文件系统。另外,函数 lookup _ vfsmnt 用于检查注册的文件系统。

执行文件系统的注册/注销操作时, 将在以 `vfsmntlist` 为链表头和 `vfsmnttail` 为链表尾的单向链表中增加/删除一个 `vfsmount` 节点, 具体数据结构如下:

```
static struct vfsmount vfsmntlist = (static struct vfsmount )NULL;
                                         /* 头 */
static struct vfsmount * vfsmnttail = (static struct vfsmount *)NULL;
                                         /* 尾 */
static struct vfsmount * mru_vfsmnt = (static struct vfsmount *)NULL;
                                         /* 当前 */

struct vfsmount
{
    kdev_t mnt_dev;                      /* 文件系统所在的主次设备号 */
    char * mnt_devname;                  /* 文件系统所在的设备名, 如 /dev/hda1 */
    char * mnt_dirname;                 /* 安装目录名 */
    unsigned int mnt_flags;              /* 设备标志, 如 ro */
    struct semaphore mnt_sem;           /* 设备有关的信号量 */
    struct super_block * mnt_sb;        /* 指向超级块 */
    struct file * mnt_quotas[MAXQUOTAS]; /* 指向配额文件的指针 */
    time_t mnt_iexp[MAXQUOTAS];        /* expiretime for inodes */
    time_t mnt_bexp[MAXQUOTAS];        /* expiretime for blocks */
    struct vfsmount * mnt_next;         /* 后继指针 */
};
```

超级用户安装一个文件系统的命令格式是:

mount 参数 文件系统类型 文件系统设备名 文件系统安装目录
文件管理接收 `mount` 命令的处理过程是:

- 步骤 1: 如果文件系统类型注册表中存在对应的文件系统类型, 转步骤 3。
- 步骤 2: 如果文件系统类型不合法, 则出错返回。
否则在文件系统类型注册表注册对应的文件系统类型。
- 步骤 3: 如果该文件系统对应的物理设备不存在或已经被安装,
则出错返回。
- 步骤 4: 如果文件系统安装目录不存在或已经安装有其他文件系统,
则出错返回。
- 步骤 5: 向内存超级块数组 `super_blocks[]` 申请一个空闲的内存超级块。
- 步骤 6: 调用文件系统类型节点提供的 `read_super` 函数读入安装文件系

统的外存超级块,写入内存超级块。

步骤 7: 申请一个 `vfsmount` 节点,填充正确内容后,假如文件系统注册表。

在使用 `umount` 卸装文件系统时,必须首先检查文件系统是否正在被其他进程使用。若正在被使用 `umount` 操作必须等待,否则可以把内存超级块写回外存,并在文件系统注册表中删除相应节点。

6.5.3 虚拟文件系统 VFS

虚拟文件系统 VFS 是物理文件系统与服务之间的一个接口层,它对每一个具体的文件系统的所有细节进行抽象,使得 Linux 用户能够用同一个接口使用不同的文件系统。VFS 只是一种存在于内存的文件系统,在系统启动时产生,并随着系统的关闭而注销。拥有关于各种特殊文件系统的公共接口,如超级块、inode、文件操作函数入口等,特殊的文件系统的细节统一由 VFS 的公共接口来翻译,当然对系统内核和用户进程是透明的。它的主要功能包括:

- 记录可用的文件系统的类型。
- 把设备与对应的文件系统联系起来。
- 处理一些面向文件的通用操作。
- 涉及到针对具体文件系统的操作时,把它们映射到与控制文件、目录以及 inode 相关的物理文件系统。

在引入了 VFS 后,Linux 文件管理的实现层次如图 6-19。当某个进程发出了一个文件系统调用时,内核将调用 VFS 中相应函数,这个函数处理一些与物理结构无关的操作,并且把它重新定向为真实文件系统中相应函数调用,后者再来处理那些与物理结构有关的操作。

实现 VFS 的数据结构主要有以下一些:

- 超级块(super block) 存储被安装的文件系统的信息,对基于磁盘的文件系统来说,超级块中包含文件系统控制块。
- 索引节点(inode) 存储通用的文件信息,对基于磁盘的文件系统来说,一般是指磁盘上的文件控制块,每个 inode 有唯一的 inode 号,并通过 inode 号标识每个文件。
- 系统打开文件表(file) 存储进程与已打开文件的交互信息,这些信息仅当进程打开文件时才存于内核空间中。
- 目录项 dentry(directory entry) 存储对目录的连接信息,包含对应的文件信息。基于磁盘的不同文件系统按各自特定方法将信息存于磁盘上。

VFS 描述系统文件系统使用超级块和 inode 的方式。当系统初始启动时,所有被初始化的文件系统类型都要向 VFS 登记。每种文件系统类型的读超级块 `read_super` 函数必须从

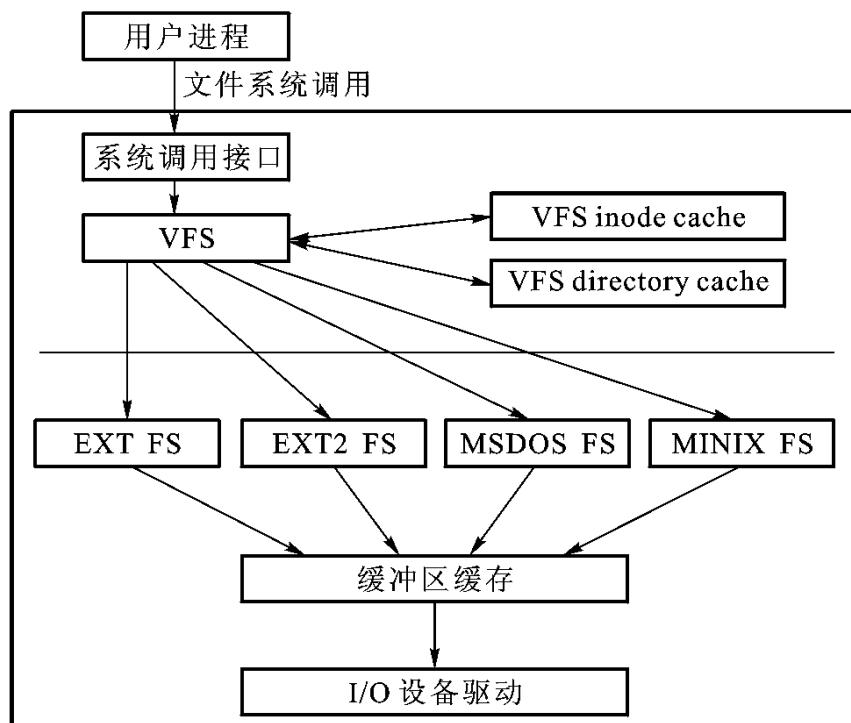


图 6-19 Linux 文件管理的实现层次

磁盘文件系统中读取给定文件系统的数据,识别该文件系统的结构,并且翻译成独立于设备的有用信息,把这些信息存储到 VFS 的 super-block 数据结构中。超级块的数据结构如下:

```

struct super_block
{
    struct list_head s_list;           /* 链接其他文件系统的超级块的链表 */
    kdev_t s_dev;                     /* 该文件系统的主次设备号 */
    unsigned long s_blocksize;         /* 块大小 */
    unsigned char s_blocksize_bits;    /* 以 2 的幂次表示块大小 */
    unsigned char s_lock;              /* 锁定标志,置位表示拒绝其他进程访问 */
    unsigned char s_rd_only;           /* 只读标志 */
    unsigned char s_dirt;              /* 已修改标志 */
    struct file_system_type *s_type;   /* 指向文件系统类型注册表相应项 */
    struct super_operations *s_op;     /* 超级块提供的文件操作函数 */
    struct dquot_operations *dq_op;    /* 超级块提供的磁盘配置操作 */
    unsigned long s_flags, s_magic, s_time; /* 标志、更新时间长度等 */
    struct dentry *s_root;             /* 超级块根节点的 dentry 节点 */
    struct wait_queue *s_wait;         /* 超级块上的等待队列 */
    struct inode *s_ibasket;
    short int s_ibasket_max, s_ibasket_count;
}
  
```

```

struct list - head s - dirty;      /* 超级块所有更改过的 inode 的链接表 */
union
{
    /* 各个物理文件系统超级块的特有结构类型 */
    struct minix - sb - info minix - sb;
    struct ext2 - sb - info ext2 - sb;
    struct hpfs - sb - info hpfs - sb;
    struct ntfs - sb - info ntfs - sb;
    struct msdos - sb - info msdos - sb;
    struct isofs - sb - info isofs - sb;
    struct nfs - sb - info nfs - sb;
    struct sysv - sb - info sysv - sb;
    struct ufs - sb - info ufs - sb;
    struct romfs - sb - info romfs - sb;
    struct smb - sb - info smb - sb;
    struct hfs - sb - info hfs - sb;
    struct adfs - sb - info adfs - sb;
    struct qnx4 - sb - info qnx4 - sb;
    void * generic - sbp;
} u;
};

```

为了保证文件系统的性能, 物理文件系统上超级块必须驻留在内存中, 具体来说, 就是利用 super_block.u 来存储具体的超级块。VFS 超级块包含了一个指向文件系统中的第一个 inode 的指针 s - mounted, 对于根文件系统, 它就是代表根目录的 inode 节点。

文件系统中的每一个子目录和文件对应于一个惟一的 i-node, 它是 Linux 管理文件系统的最基本单位, 也是文件系统连接任何子目录、任何文件的桥梁。VFS inode 可通过 i-node_cache 访问, 其内容来自于物理设备上的文件系统, 并有文件系统指定的操作函数填写。inode 的数据结构如下:

```

struct inode
{
    struct list - head i - hash, i - list, i - dentry;
                                /* 用于 hash、无用链表的 LRU 排列 */
    struct list - head i - dentry;      /* 与 inode 相连的 dentry 节点的链表 */
    unsigned long i - ino;            /* inode 号 */
    unsigned int i - count;          /* inode 节点正在访问数 */
    kdev_t i - dev;                /* 该文件系统的主次设备号 */
}

```

```

umode_t i_mode;           /* 文件类型以及存取权限 */
nlink_t i_nlink;          /* 连接到该 inode 的 link 数 */
uid_t i_uid;              /* inode 的用户 ID */
gid_t i_gid;              /* inode 的组 ID */
kdev_t i_rdev;             /* 该 inode 描述的主次设备号 */
off_t i_size;              /* inode 大小 */
time_t i_atime, i_mtime, i_ctime; /* 访问、修改和创建时间 */
unsigned long i_blksize;      /* inode 节点的块大小 */
unsigned long i_blocks;       /* inode 节点的块数目 */
unsigned long i_version;      /* inode 节点的版本 */
unsigned long i_nrpages;       /* inode 节点所占的页数 */
struct semaphore i_sem, i_atomic_write; /* inode 操作信号量和原语写操作 */
struct inode_operations * i_op; /* inode 的操作函数 */
struct super_block * i_sb;    /* inode 所在的 VFS 超级块 */
struct wait_queue * i_wait;   /* inode 的等待队列 */
struct file_lock * i_flock;   /* inode 的记录锁链表首地址 */
struct vm_area_struct i_mmap; /* inode 内存映象 */
struct page * i_pages;        /* inode 内存页面单向链表 */
struct dquot * i_dquot[MAXQUOTAS];
struct inode i_next, i_prev;   /* inode 链指针 */
struct inode i_hash_next, i_hash_prev; /* inode cache 链指针 */
struct inode * i_bound_to, * i_bound_by;
struct inode * i_mount;        /* 指向下挂文件系统的 inode 的根目录 */
unsigned long i_count;         /* 引用计数, 0 表示空闲 */
unsigned short i_flags, i_writecount;
unsigned char i_lock;          /* inode 的锁定标志 */
unsigned char i_dirt;          /* inode 已修改标志 */
unsigned char i_pipe, i_sock, i_seek, i_update, i_condemned;
union
{
    /* 各个物理文件系统 inode 的特有结构类型 */
    struct pipe_inode_info pipe_i;
    struct minix_inode_info minix_i;
    struct ext2_inode_info ext2_i;
    struct hpfs_inode_info hpfs_i;
    struct ntfs_inode_info ntfs_i;
}

```

```

    struct msdos - inode - info msdos - i;
    struct umsdos - inode - info umsdos - i;
    struct iso95 - inode - info iso95 - i;
    struct nfs - inode - info nfs - i;
    struct sysv - inode - info sysv - i;
    struct affs - inode - info affs - i;
    struct ufs - inode - info ufs - i;
    struct romfs - inode - info romfs - i;
    struct coda - inode - info coda - i;
    struct smb - inode - info smb - i;
    struct adfs - inode - info adfs - i;
    struct qnx4 - inode - info qnx4 - i;
    struct socket socket - i;
    void * generic - ip;
} u;
};

```

同超级块一样, inode.u 用于存储每一个特定文件系统的特定 inode。系统所有的 inode 通过 i_prev, i_next 连接成双向链表, 头指针是 first_inode。每个 inode 通过 i_dev 和 i_ino 唯一地对应到某一个设备上的某一个文件或子目录。i_count 为 0 时表明该 inode 空闲, 空闲的 inode 总是放在 first_inode 链表的前面, 当没有空闲的 inode 时, VFS 会调用函数 grow_inodes 从系统内核空间申请一个页面, 并将该页面分割成若干个空闲 inode, 加入 first_inode 链表。围绕 first_inode 链, VFS 还提供一组操作函数, 主要有: insert_inode_free()、remove_inode_free()、put_last_free()、insert_inode_hash()、remove_inode_hash()、clear_inode()、get_empty_inode()、lock_inode()、unlock_inode()、write_inode() 等。

VFS 的目录中存储了当前目录下的文件和子目录信息, 在 VFS 中目录也被抽象成文件的形式, 每个目录有自己的 inode, 这样 VFS 可采用相同的方法处理文件和目录。VFS 中引入目录 dentry 的主要目的是协助实现对文件的快速定位, 改进文件系统效率, 此外, 目录还起到一定的缓冲作用。Dentry 一般被维护在 cache 中, 这样可以快速找到所需目录, 以便快速的定位文件。下面是目录 dentry 的数据结构:

```

Struct dentry {
    atomic_t d_count;                      /* 当前 dentry 引用数 */
    unsigned int d_flags;                   /* 识别 dentry 状态的标志 */
    struct inode d_inode;                  /* 此 dentry 对应的 inode */
    struct dentry d_parent;                /* 父目录的 dentry 结构 */
}

```

```

struct list_head d_hash;           /* 链入 dentry 的哈希表 */
struct list_head d_lru;          /* 引用数为 0 的 dentry 构成的双向链表 */
struct list_head d_child;        /* 此 dentry 的兄弟 dentry 双向链表 */
struct list_head d_subdirs;      /* 此 dentry 的子目录双向链表 */
struct list_head d_alias;        /* 硬链接时, 指向同一个 inode 的 dentry 链表 */
int d_mounted;                  /* 此 dentry 被安装次数 */
struct qstr d_name;             /* 此 dentry 的名称及哈希值 */
unsigned long d_time;            /* 记录时间用 */
struct dentry_operations *d_op;  /* 一组 dentry 的操作函数 */
struct super_block *d_sb;        /* 此 dentry 超级块指针 */
...
unsigned char d_iname[NAME_INLINE_LEN]; /* 文件名前 16 个字符 */
};

```

6.5.4 文件系统管理的缓冲机制

Linux 既支持多种类型的文件系统, 又保持了很高的性能, 探究其原因, 除了 VFS 以外, 多种复杂的 cache 起到了关键作用。

1. VFS inode cache

随着文件的读出和写入, VFS inode 不断读入内存。从效率角度出发, 为提高对 inode 链表进行线性搜索的速度, VFS 为已经分配的 inode 构造了 cache 和 hash 表。VFS 访问 inode 时, 它首先根据 hash 函数计算出 h, 然后找到对应的 hash_table[h] 指向的双向链表, 通过 i_hash_next 和 i_hash_prev 进行查找。如果从 cache 中找到了指定设备上的 inode, 则该 inode 的 i_count 加 1; 否则申请一个空闲的 inode。

2. VFS directory cache

通过路径名查找该目录文件的 inode 是使用频率极高的操作。为提高此类操作的效率, Linux 维护了表达路径与 inode 对应关系的 VFS directory cache。被访问过的目录将会被存入 directory cache, 这样当同一目录被再次访问时就可以快速获得。Linux 提供 namei() 函数支持这类操作, 数据结构如下:

```

struct hash_list
{struct dir_cache_entry *next, *prev;};
struct dir_cache_entry
{
    struct hash_list h;

```

```

kdev_t dc_dev;
unsigned long dir; /* 父目录的 inode */
unsigned long version;
unsigned long ino; /* 本目录的 inode */
unsigned char name_len;
char name[DCACHE_NAME_LEN];
struct dir_cache_entry * lru_head;
struct dir_cache_entry * next_lru, * prev_lru;
};

```

VFS directory cache 由 level1 _ cache 和 level2 _ cache 组成, 两个 cache 的头指针分别存放在 level1 _ head 和 level2 _ head 中。level1 _ cache 和 level2 _ cache 均采用 dir _ cache _ entry . next _ lru 和 dir _ cache _ entry . prev _ lru 指针构成包含 128 个节点的双向循环链表, 使用 LRU 算法增删节点, 越是最近使用的节点越靠近链尾。level1 _ cache 和 level2 _ cache 各有分工, 新增加的目录存放在 level1 _ cache 尾部; 查询信息命中后则放在 level2 _ cache 尾部。为进一步提高效率, Linux 还为 level1 _ cache 和 level2 _ cache 建立了 hash 表。

3. buffer cache

为加快对物理设备的访问, Linux 维护一组数据块缓冲区, 称为 buffer cache。buffer cache 就是文件组织中所提到的主存缓冲区, 它独立于任何类型的文件系统, 被所有的物理设备所共享。数据块一经使用, 就会在 buffer cache 留下备份, 下次再访问该数据块时, 不必再访问物理设备。

Linux 用结构 buffer _ head 类型封装数据缓冲区, 进行缓冲区数据的读写操作。结构 buffer _ head 给出了设备驱动程序需要的全部信息。操作系统把设备上的数据看作是等长数据块的线性列表, 通过 buffer _ head 的属性值惟一指明向什么设备写第几个数据块。

为提高访问效率, buffer cache 系统被精心设计成一个 hash 表和四类 buffer 链表:

- hash 表, 用于进一步提高访问效率。

```
static struct buffer_head ** hash_table;
```

- 最近最少用链表, 它包括 4 个链表。lru_list[0] 保存数据已经写出的干净的缓冲区; lru_list[1] 保存正在进行写出操作的缓冲区; lru_list[2] 保存超级块和 inode 的缓冲区; lru_list[3] 保存已有新数据但尚未安排写出的缓冲区。不难看出, 此类链表的 buffer _ head 节点都封装了数据缓冲区。

```
static struct buffer_head * lru_list[NR_LIST];
```

- 空闲链表, 它按照 buffer 长度(512、1024、2048、4096、8192 字节)分成 5 个链表, 其中的 buffer _ head 节点封装了数据缓冲区。

```
static struct buffer_head * free_list[NR_SIZES];
```

- 未使用链表,只有 buffer_head 节点,未封装数据缓冲区;

```
static struct buffer_head * unused_list;
```

- 重用链表,只有 buffer_head 节点,未封装数据缓冲区;

```
static struct buffer_head * reuse_list;
```

6.5.5 系统打开文件表和主要文件操作

系统打开文件表记录已经打开的文件,用于和打开该文件的进程交互,和控制文件的读写操作。系统打开文件表是一张以 file 结构作为节点的双向链表,表头指针为 first_file,每个节点对应一个已打开的文件,包含了此文件的 inode、操作函数等。数据结构如下:

```
struct file {
    /* 每个打开文件对应一个 */
    mode_t f_mode;           /* 文件模式 */
    loff_t f_pos;            /* 文件读写指针 */
    struct dentry * f_dentry; /* 文件对应的 dentry 节点 */
    unsigned int f_flags, f_count;
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    unsigned int f_uid, f_gid; /* 文件用户和组 ID */
    struct file * f_next, * f_pprev; /* 打开文件表中前后 file 指针 */
    struct fown_struct f_owner; /* 文件所有者 */
    struct file_operations * f_op; /* 文件支持的操作集 */
    unsigned long f_version;   /* 文件版本 */
    void * private_data;
};
```

Linux 在每一个进程的 task_struct 中设计了两个数据结构:fs_struct 和 files_struct,它们的定义分别如下:

```
struct fs_struct
{
    atomic_t count;           /* 引用计数器 */
    int umask;                /* 新建文件的缺省模式 */
    struct dentry root, pwd;  /* 指向虚拟文件系统的 inode */
};

struct files_struct
{
```

```

atomic_t count;           /* 引用计数器 */
int max_fds;             /* 最多打开文件数 */
fd_set close_on_exec;    /* 系统调用 exec 时关闭的文件的屏蔽字数组 */
fd_set open_fds;          /* 对所有文件描述字 fd 的屏蔽字数组 */
struct file *fd[NR_OPEN];
/* 进程打开文件的 fd 数组, 其元素为指向系统打开文件表中的一个节点指针 */
};

```

图 6-20 是进程文件数据结构间的关系。Fs-struct 用来描述进程工作的文件系统的信息, 包括根目录和当前工作目录的 dentry, 它们安装的文件系统的信息, 以及在 umask 中保存的新建文件的缺省权限; files-struct 包含了进程当前所打开的文件信息。进程在开始时打开三个文件, 它们是 standard input、standard output 和 standard error, 且这三个文件通常从父进程处继承而来。所有通过系统调用的文件操作要么返回文件描述字, 要么使用文件描述字。三个标准文件的文件描述字分别为 0、1 和 2。

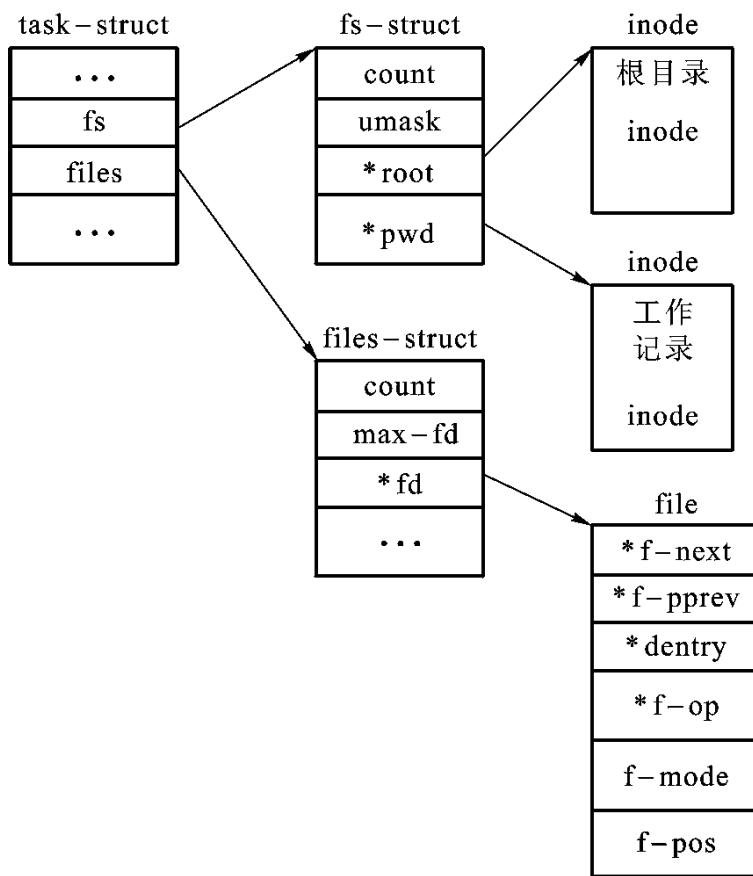


图 6-20 进程文件数据结构间的关系

Linux 引入 fs-struct、file、file_struct 结构后, 可以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 惟一对应一个文件), 二是多个进程的 file 结构共享 i-node(i-node

惟一对应一个文件)。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,两者有了相同的用户打开文件表,都有一项指向同一个 file 结构。后者是通过文件的 link 机制实现的,两个独立进程打开同一文件也属于这种共享。在这种方式中,Linux 专门设计了文件锁,解决两个进程同时读写文件时的互斥访问问题。可以在 include/linux/fs.h 中找到文件锁的数据结构。

下面概略介绍 Linux 的主要文件操作。在 VFS 中,采用 dentry 结构和 inode 节点配合实现文件查找。每个打开的文件都对应一个 dentry 节点,并存放在内核的 dentry - cache 中。当查找一个文件时,利用 dentry * namei 函数,沿文件的路径名,依次查看每一层目录的 dentry 节点是否出现在 dentry - cache 中,如果没有,就通过父目录直接到磁盘上去查找,得到它对应的 inode 节点,再读入内存中,并在 dentry - cache 中新建一个 dentry 节点与得到的已存于内存中的 inode 节点建立联系。之后,系统在文件查找时不用每次再去访问磁盘,直接从核心的内存区中找到文件的 inode 节点,提高了文件查找效率。下面是主要文件操作:

1. 文件的打开

主要函数有:sys - open() 和 sys - creat()。对每个打开的文件,系统会分给它一个唯一的文件 ID 号,进程对文件操作时,只需 ID 号便可找到文件指针。要打开某个文件时,系统先得到一个空的文件 ID 号和一个文件信息节点,然后,由相应文件名通过文件查找找到它的 dentry 和 inode 节点,建立四者的联系,最后,再通过具体的文件系统自身提供的文件打开函数真正地打开指定文件。如果该文件不存在,系统会根据给定参数,先建立文件再把文件打开。

2. 文件的关闭

关闭文件使用函数 sys - close(),系统先释放文件的 ID 号,再释放文件信息节点、dentry 节点、inode 节点,最后,调用具体文件系统的关闭函数关闭该文件。文件被修改时还要进行更新,关闭的同时,要移去其他进程在该文件上留下的记录锁。

3. 文件指针移动

主要函数有:sys - llseek() 和 sys - lseek()。系统根据给定的操作参数对文件的指针进行移动,有两种指针移动方式:一种是从当前文件指针开始;另一种是从文件的末尾开始。针对不同的文件类型,系统提供两种移动函数:一种是当指定文件为符号链接文件时,查到其链接的源文件,并移动文件指针;另一种是当指定文件为符号链接文件时,不进行链接查找,仅移动指定文件的指针。

4. 读文件

主要函数有:sys - read() 和 sys - pread()。这是由具体文件系统实现的功能,先要判断欲读的文件区域是否被其他进程锁住,再决定能否把文件内容读到指定区域。两种读函数

分别为：从文件的当前指针读起、从指定的文件指针读起。

5. 写操作

主要函数有：sys - write() 和 sys - pwrite()。系统先要判断欲写的文件区域是否被其他进程锁住，再决定能否把文件内容写到指定区域。两种写函数分别为：从文件的当前指针写起、从指定的文件指针写起。

6. 文件属性控制

函数 sys - fcntl() 对文件的相关属性进行修改和查询。

7. 文件上锁

同时支持函数 sys - fcntl() 和 sys - flock() 对文件或文件记录上锁。

8. 文件的 I/O 控制

函数 sys - ioctl() 对文件的 I/O 属性进行修改和查询。

9. 各种其他文件操作

Linux 还提供许多函数，用于对文件进行各种操作，主要有：文件信息的获取函数、文件访问权限测试和修改函数、文件 UID 和 GID 修改函数、文件裁减函数、文件的链接和解除链接函数、文件重命名函数、文件目录的创建和删除函数，读文件目录函数、改变当前工作目录函数。

6.5.6 EXT2 文件系统

扩展文件系统 EXT(1992 年)和第二代扩展文件系统 EXT2(1994 年)是专门为 Linux 设计的可扩展的文件系统。在 EXT2 中，文件系统组织成数据块的序列，这些数据块的长度相同，块大小在创建时被固定下来。如图 6-21 所示，EXT2 所占用的磁盘除引导块 (boot block) 外，逻辑分区划分为块组 (block group)，每一个块组依次包括超级块 (super group)、组描述符表 (group descriptors)、块位图 (block bitmap)、i-node 位图 (i-node bitmap)、i-node 表 (i-node table) 以及数据块 (data blocks) 区。块位图集中了本组各个数据块的使用情况；inode 位图则记录了 inode 表中 inode 的使用情况。inode 表保存了本组所有的 inode。inode 用于描述文件，一个 inode 对应一个文件或子目录，有一个唯一的 i-node 号，并记录了文件在外存的位置、存取权限、修改时间、类型等信息。

采用块组划分的目的是使数据块靠近其 i-node 节点，文件 i-node 节点靠近其目录 i-node 节点。从而，将磁头定位时间减到最少，加快磁盘访问速度。

1. EXT2 超级块

EXT2 的超级块用来描述目录和文件在磁盘上的静态分布，包括尺寸和结构。每个块组

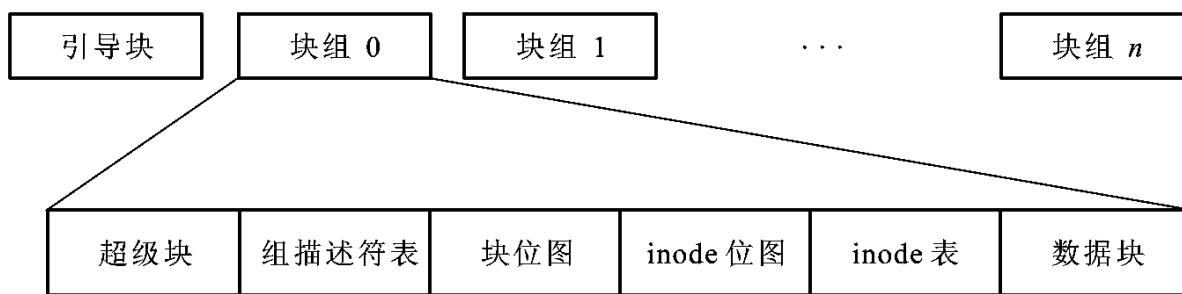


图 6-21 EXT2 文件系统结构

都有一个超级块,一般来说只有块组 0 的超级块才被读入内存超级块,其他块组的超级块仅仅作为恢复备份。EXT2 文件系统的超级块主要包括 inode 数量、块数量、保留块数量、空闲块数量、空闲 inode 数量、第一个数据块位置、块长度、片长度、每个块组块数、每个块组片数、每个块组 inode 数,以及安装时间、最后一次写时间、安装信息、文件系统状态信息等等内容。Linux 中引入了片(fragment)的概念,若干个片可组成块,当 EXT2 文件最后一块不满时,可用片计数。具体的 EXT2 外存超级块和内存超级块数据结构参见 include/linux/ext2_fs.h 中的结构 ext2_super_block 和结构 ext2_sb_info。

2. EXT2 组描述符

每个块组都有一个组描述符,记录了该块组的块位图位置、i-node 位图位置、i-node 节点位置、空闲块数、i-node 数、目录数等内容。具体的组描述符数据结构参见 include/linux/ext2_fs.h 中的结构 ext2_group_desc。

所有的组描述符一个接一个存放,构成了组描述符表。同超级块一样,组描述符表在每个块组中都有备份。当文件系统崩溃时,备份可以用来恢复文件系统,正常情况下只使用块组 0 内的组描述符。

3. EXT2 inode

在 EXT2 中,每个文件或目录都由一个 i-node 来惟一描述,每个块组的 i-node 集中存放在一个 i-node 表中。每个 i-node 有一个惟一的 i-node 号,并记录了文件的类型及存取权限、用户和组标识、修改/访问/创建/删除时间、link 数、文件长度、占用块数及其指针、在外存的位置、以及其他控制信息。具体的数据结构参见 include/linux/ext2_fs.h 中的结构 ext2_inode。

4. EXT2 目录文件

目录是用来创建和保存对文件系统中的文件的存取路径的特殊文件。它是一个目录项的列表(目录文件),其中头两项是标准目录项 .”(本目录)和 ..”(父目录)。目录项的数据结构如下:

```

struct ext2_dir_entry_2
{
    _u32    inode;           /* inode 号 */
    _u16    rec_len;         /* 目录项长度 */
    _u16    name_len;        /* 文件名长度 */
    _u8     file_type;       /* 文件名类型 */
    char   name[EXT2_NAME_LEN]; /* 文件名 */
};

```

5. 数据块分配策略

文件空间的碎片是每个文件系统都要解决的问题, 它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盘的各处, 访问这类文件时, 致使磁头移动急剧增多, 访问盘的速度大幅下降。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 defrag(defragmentation program)。而操作系统能够通过分配策略避免碎片的发生则更加重要, EXT2 采用了两个策略来减少文件碎片:

- 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附近查找。首先试探紧跟文件末尾的那个数据块, 然后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, 而且首先考虑 8 个一族的连续的块;
- 预分配策略: 如果 EXT2 引入了预分配机制(设 EXT2_PREALLOCATE 参数), 就从预分配的数据块取一块来用, 这时紧跟该块后的若干个数据块空闲的话, 也被保留下来。当文件关闭时仍保留的数据块给予释放, 这样保证了尽可能多的数据块被集中成一族。EXT2 文件系统的 inode 的 ext2_inode_info 数据结构中包含两个属性 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 实例研究: Windows 2000/XP 文件系统

6.6.1 Windows 2000/XP 文件系统概述

Windows 2000/XP 支持传统的 FAT 文件系统, 对 FAT 文件系统的支持起源于 DOS, 以后的 Windows 3.x 和 Windows 95 列均支持它们。该文件系统最初是针对相对较小容量的硬盘设计的, 但是随着计算机外存储设备容量的迅速扩展, 出现了明显的不适应。不难看出,

FAT 文件系统最多只可以容纳 2^{12} 或 2^{16} 个簇, 单个 FAT 卷的容量小于 2 GB, 显然, 如果继续扩展簇中包含的扇区数, 文件空间的碎片将很多, 浪费很大。

从 Windows 9x 和 Windows Me 开始, FAT 表被扩展到 32 位, 形成了 FAT32 文件系统, 解决了 FAT16 在文件系统容量上的问题, 可以支持 4 GB 的大硬盘分区, 但是由于 FAT 表的大幅度扩充, 造成了文件系统处理效率的下降。Windows 98 操作系统也支持 FAT32, 但与其同期开发的 Windows NT 则不支持 FAT32, 基于 NT 构建的 Windows 2000/XP 则又支持 FAT32, 此外还支持: 只读光盘 CDFS、通用磁盘格式 UDF、高性能 HPFS 等文件系统。

Microsoft 的另一个操作系统产品 Windows NT 开始提供一个全新的文件系统 NTFS (New Technology File System)。NTFS 除了克服 FAT 系统在容量上的不足外, 主要出发点是立足于设计一个服务器端适用的文件系统, 除了保持向后兼容性的同时, 要求有较好的容错性和安全性。为了有效地支持客户/服务器应用, Windows 2000/XP 在 NT4 的基础上进一步扩充了 NTFS, 这些扩展需要将 NT4 的 NTFS4 分区转化为一个已更改的磁盘格式, 这种格式被称为 NTFS5。NTFS 具有以下的特性:

- 可恢复性: NTFS 提供了基于事务处理模式的文件系统恢复, 并支持对重要文件系统信息的冗余存储, 满足了用于可靠的数据存储和数据访问的要求。
- 安全性: NTFS 利用操作系统提供的对象模式和安全描述体来实现数据安全性。在 Windows 2000/XP 中, 安全描述体 (访问控制表或 ACL) 只需存储一次就可在多个文件中引用, 从而进一步节省磁盘空间。
- 文件加密: 在 Windows 2000 中, 加密文件系统 EFS (Encrypting File System) 能对 NTFS 文件进行加密再存储到磁盘上。
- 数据冗余和容错: NTFS 借助于分层驱动程序模式提供容错磁盘, RAID 技术允许借助于磁盘镜像技术, 或通过奇偶校验和跨磁盘写入来实现数据冗余和容错。
- 大磁盘和大文件: NTFS 采用 64 位分配簇, 从而, 大大扩充了磁盘卷容量和文件长度。
- 多数据流: 在 NTFS 中, 每一个与文件有关的信息单元, 如文件名、所有者、时间标记、数据, 都可以作为文件对象的一个属性, 所以 NTFS 文件可包含多数据流。这项技术为高端服务器应用程序提供了增强功能的新手段。
- 基于 Unicode 的文件名: NTFS 采用 16 位的 Unicode 字符来存储文件名、目录和卷, 适用于各个国家与地区, 每个文件名可以长达 255 个字符, 并可以包括 Unicode 字符、空格和多个句点。
- 通用的索引机制: NTFS 的体系结构被组织成允许在一个磁盘卷中索引文件属性, 从而可以有效地定位匹配各种标准文件。在 Windows 2000 中, 这种索引机制被扩展到其他属性, 如对象 ID。对属性 (例如基于 OLE 上的复合文件) 的本地支持, 包括对这些属性的一般

索引支持。属性作为 NTFS 流在本地存储,允许快速查询。

- 动态添加卷磁盘空间:在 Windows 2000 中,增加了不需要重新引导就可以向 NTFS 卷中添加磁盘空间的功能。

- 动态坏簇重映射:可加载的 NTFS 容错驱动程序可以动态地恢复和保存坏扇区中的数据。

- 磁盘配额:在 Windows 2000 中,NTFS 可以针对每个用户指定磁盘配额,从而提供限制使用磁盘存储器的能力。

- 稀疏文件:在 Windows 2000 中,用户能够创建文件,并且在扩展这些文件时不需要分配磁盘空间就能将这些文件扩展为更大。另外,磁盘的分配将推迟至指定写入操作之后。

- 压缩技术:在 Windows 2000 中,能对文件数据和目录进行压缩,节省了存储空间。文本文件可压缩 50%,可执行文件可压缩的 40%。

- 分布式链接跟踪:在 Windows 2000 中,NTFS 支持文件或目录的惟一 ID 号的创建和指定,并保留文件或目录的 ID 号。通过使用惟一的 ID 号,从而实现分布式链接跟踪。这一功能将改进当前的文件引用存储方式(例如,在 OLE 链接或桌面快捷方式中)。重命名目标文件的过程将中断与该文件的链接。重命名一个目录将中断所有此目录中的文件链接及此目录下所有文件和目录的链接。

- POSIX 支持:如支持区分大小写的文件名、链接命令、POSIX 时间标记等。在 Windows 2000 中,还允许实现符号链接的重解析点,仲裁文件系统卷的装配点和远程存储 分层存储管理(HSM)”。

Windows 2000/XP 还提供分布式文件服务。分布式文件系统(DFS)是用于 Windows 2000/XP 服务器上的一个网络服务器组件,最初它是作为一个扩展层发售给 NT4 的,但是在功能上受到很多限制。在 Windows 2000/XP 中,这些限制得到了修正。DFS 能够使用户更容易地找到和管理网上的数据。使用 DFS,可以更容易地创建一个单目录树,该目录树包括多文件服务器和组、部门或企业中的文件共享。另外,DFS 可以给予用户一个单一目录,这一目录能够覆盖大量文件服务器和文件共享,使用户能够很方便地通过“浏览”网络去找到所需要的数据和文件。浏览 DFS 目录是很容易的,因为不论文件服务器或文件共享的名称如何,系统都能够将 DFS 子目录指定为逻辑的、描述性的名称。

6.6.2 Windows 2000/XP 文件系统模型和 FSD 体系结构

在 Windows 2000/XP 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 6-22 所示。

- 设备驱动程序:位于 I/O 管理器的最底层,直接对设备进行 I/O 操作。

- 中间驱动程序:与低层设备驱动程序一起提供增强功能,如发现 I/O 失败时,设备驱

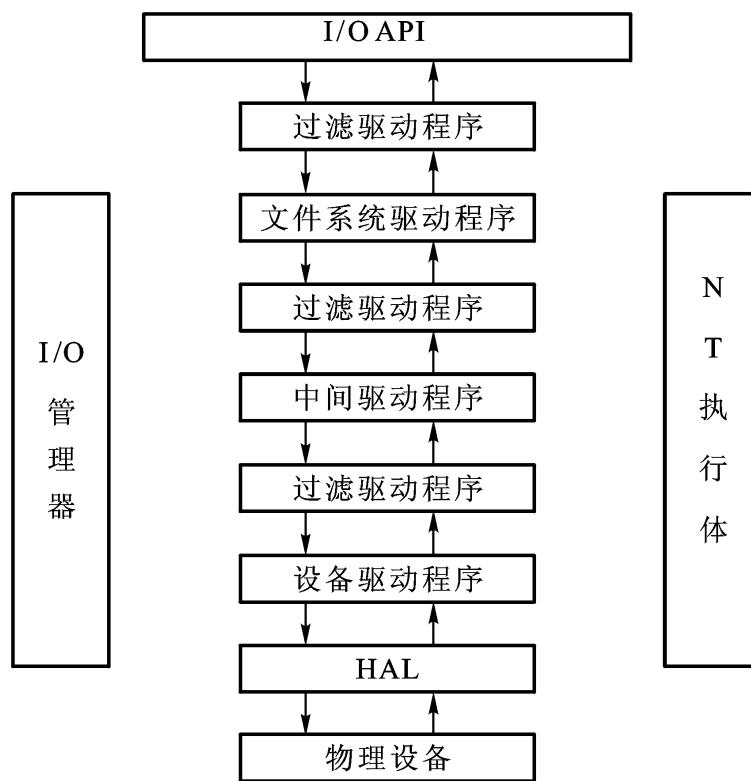


图 6-22 Windows 文件系统模型

动程序只会简单地返回出错信息;而中间驱动程序却可能在收到出错信息后,向设备驱动程序下达重执请求。

- 文件系统驱动程序 FSD(File System Driver):扩展底层驱动程序的功能,以实现特定的文件系统(如 NTFS)。
- 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 I/O 管理器 API 之间。例如,一个网络重定向过滤驱动程序可截取对远程文件的操作,并重定向到远程文件服务器上。

在以上组成构件中,与文件管理最为密切相关的是 FSD,它工作在内核态,但与其他标准内核驱动程序有所不同。FSD 必须先向 I/O 管理器注册,还要与内存管理器和高速缓存管理器产生大量交互。因此,FSD 使用了 Ntoskrnl 出口函数的超集,它的创建必须通过 IFS(Installable File System)实现。

下面简单介绍 FSD 的体系结构。文件系统驱动程序可分为本地 FSD 和远程 FSD,前者允许用户访问本地计算机上的数据,后者则允许用户通过网络访问远程计算机上的数据。

1. 本地 FSD

本地 FSD 包括:Ntfs.sys, Fastfat.sys, Udfs.sys, CDfs.sys 和 Raw FSD 等,参见图 6-23。本地 FSD 负责向 I/O 管理器注册自己,当开始访问某个卷时,I/O 管理器将调用 FSD 来进行卷识别。

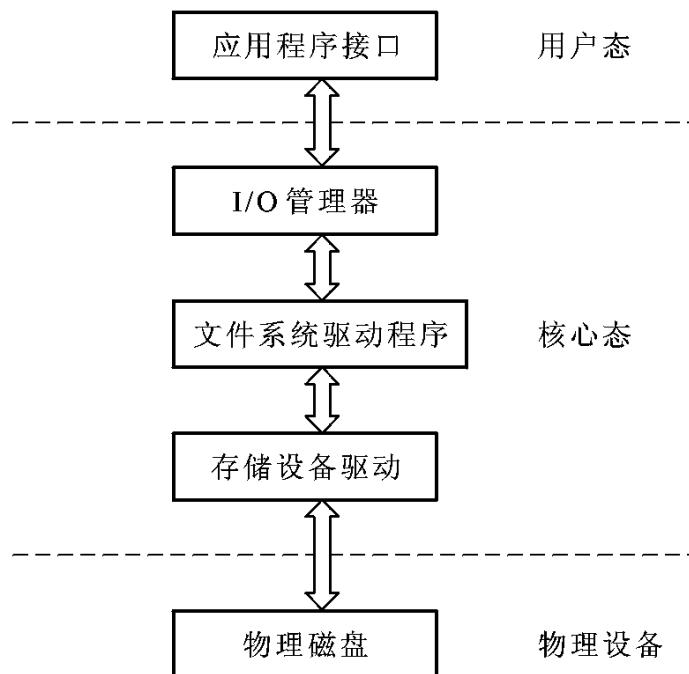


图 6-23 本地 FSD

Windows 2000/XP 支持的文件系统,每个卷的第一个扇区都作为启动扇区预留的,其上保存了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。另外,卷识别常常需要对文件系统作一致性检查。

当完成卷识别后,本地 FSD 还创建一个设备对象以表示所装载的文件系统。I/O 管理器也通过卷参数块 VPB (Volumn Parammeter Block) 为由存储管理器所创建的卷设备对象和由 FSD 所创建的设备对象之间建立连接,该 VPB 连接将 I/O 管理器的有关卷的 I/O 请求转交给 FSD 设备对象。

本地 FSD 常用高速缓存管理器来缓存文件系统的数据以提高性能,它与内存管理器一起实现内存文件映射。本地 FSD 还支持文件系统卸载操作,以便提供对卷的直接访问。

2. 远程 FSD

远程 FSD 由两部分组成:客户端 FSD 和服务器端 FSD。前者允许应用程序访问远程的文件和目录,客户端 FSD 首先接收来自应用程序的 I/O 请求,接着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,并转交给本地 FSD 去执行。图 6-24 是远程 FSD 示意。

对于 Windows 2000/XP 而言,客户端 FSD 为 LANMan 重定向器 (LANMan Redirector),而服务器端 FSD 为 LANMan 服务器 (LANMan Server)。重定向器通过端口/小端口驱动程序的组合来实现。而重定向器与服务器的通信则通过通用互联网文件系统 CIFS (Common Internet File System) 协议进行。

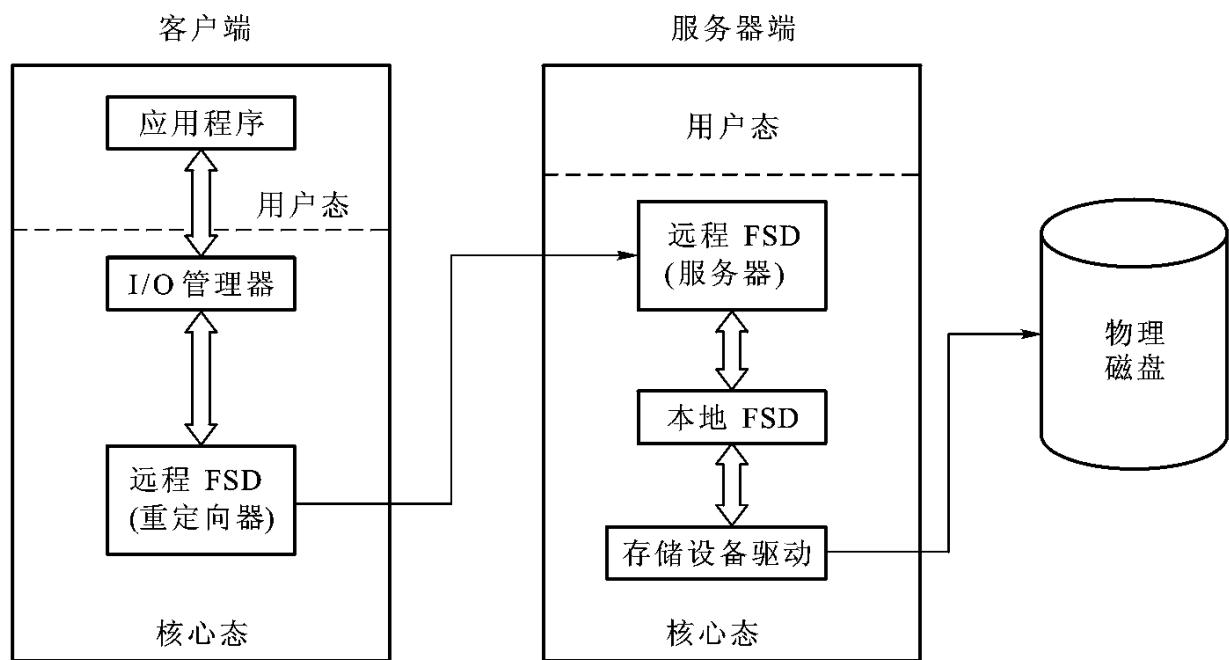


图 6-24 远程 FSD

3. FSD 与文件系统操作

Windows 文件系统的有关操作都是通过 FSD 来完成的, 其作用见图 6-25。有如下几种方式会用到 FSD:

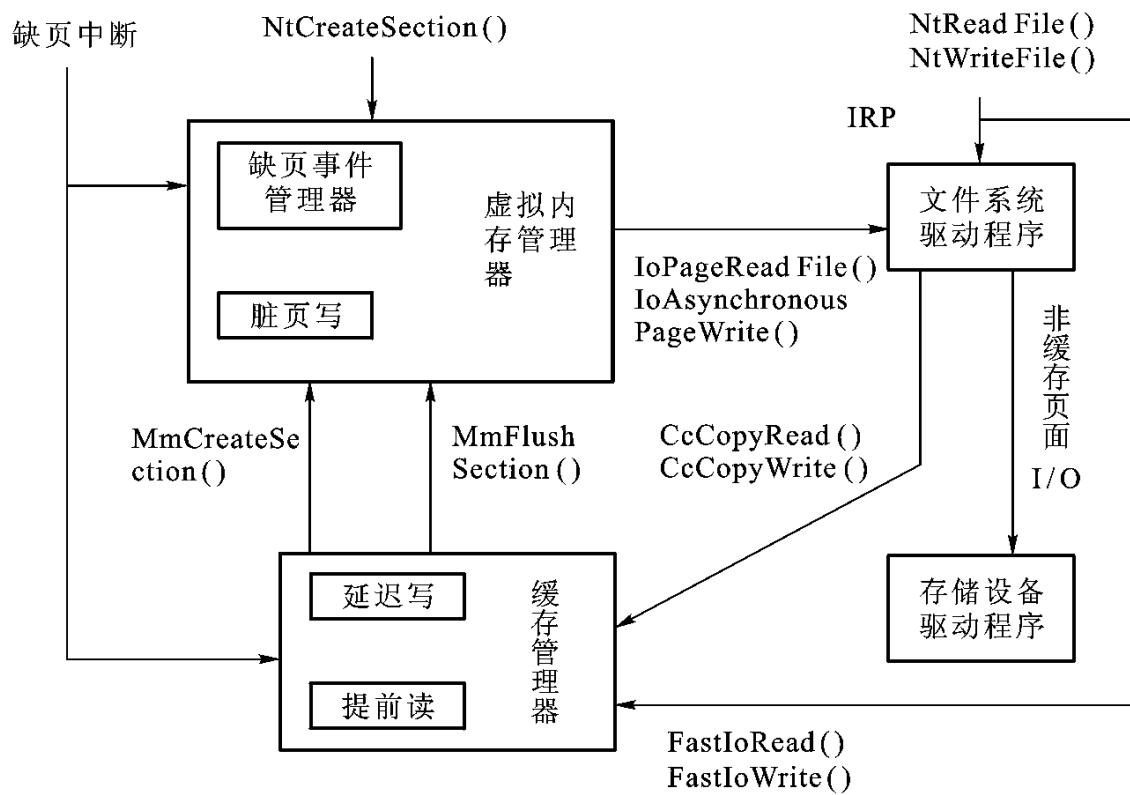


图 6-25 FSD 的作用

显式文件 I/O。应用程序通过 Win32 I/O 函数如 CreateFile、ReadFile 和 WriteFile 等来访问文件。

高速缓存延迟写。高速缓存管理器的延迟写线程定期对高速缓存中已被修改过的页面进行写操作, 这是通过调用内存管理器的 MmFlushSection 函数来完成的, 具体地说, MmFlushSection 通过 IoAsynchronousPageWrite 将数据送交 FSD。

高速缓存提前读。高速缓存管理器的提前读线程负责提前读数据, 提前读线程通过分析已做的读操作来决定提前读多少, 它依赖于缺页中断来完成这一任务。

内存脏页写。内存脏页写线程定期清洗缓冲区, 该线程通过 IoAsynchronousPageWrite 来创建 IRP 请求, 这些 IRP 被标识为不能通过高速缓存, 因此, 它们被 FSD 直接送交磁盘存储驱动程序。

内存缺页处理。以上在进行显式 I/O 操作和高速缓存提前读时, 都会用到缺页中断处理。内存缺页处理 MmAccessFault 通过 IoPageRead 向文件所在文件系统发送 IRP 请求包来完成。

6.6.3 NTFS 文件系统驱动程序

在 Windows 2000/XP 中, NTFS 及其他文件系统如 FAT、HPFS、POSIX 等都结合在 I/O 管理器中, 采用文件系统驱动程序实现。文件系统的实现机制采用面向对象的模型, 文件、目录和系统中其他资源一样, 是作为对象来管理的。文件的命名统一在对象命名空间, 文件对象由 I/O 管理器管理。用户和系统打开文件表在 Window 2000/XP 中表现为每个进程一个进程对象表及其所指向的具体文件对象。如图 6-26 所示, 在 Windows 2000/XP 的 I/O 管理器部分, 包括了一组在核心态运行的可加载的与 NTFS 相关的设备驱动程序。这些驱动程序是分层次实现的, 它们通过调用 I/O 管理器传递 I/O 请求给另外一个驱动程序, 依靠 I/O 管理器作为媒介允许每个驱动程序保持独立, 以便可以被加载或卸载而不影响其他驱动程序。另外, 图中还给出了 NTFS 驱动程序和与文件系统紧密相关的三个其他执行体的关系。

日志文件服务 LFS(log file server)是为维护磁盘写入的日志而提供服务的记录所有影响 NTFS 卷结构的操作, 如文件创建、改变目录结构。此日志文件用于在系统失败时恢复 NTFS 格式化卷。

高速缓存管理器是 Windows 2000/XP 的执行体组件, 它为 NTFS 以及包括网络文件系统驱动程序(服务器和重定向程序)的其他文件系统驱动程序提供系统范围的高速缓存服务。Windows 2000/XP 的所有文件系统通过把高速缓存文件映射到虚拟内存, 然后访问虚拟内存来访问它们。为此, 高速缓存管理器提供了一个特定的文件系统接口给 Windows 2000/XP 虚拟内存管理器。当程序试图访问没有加载到高速缓存的文件的一部分时, 内存管理器调用 NTFS 来访问磁盘驱动器并从磁盘上获得文件的内容。高速缓存管理器通过使用它的

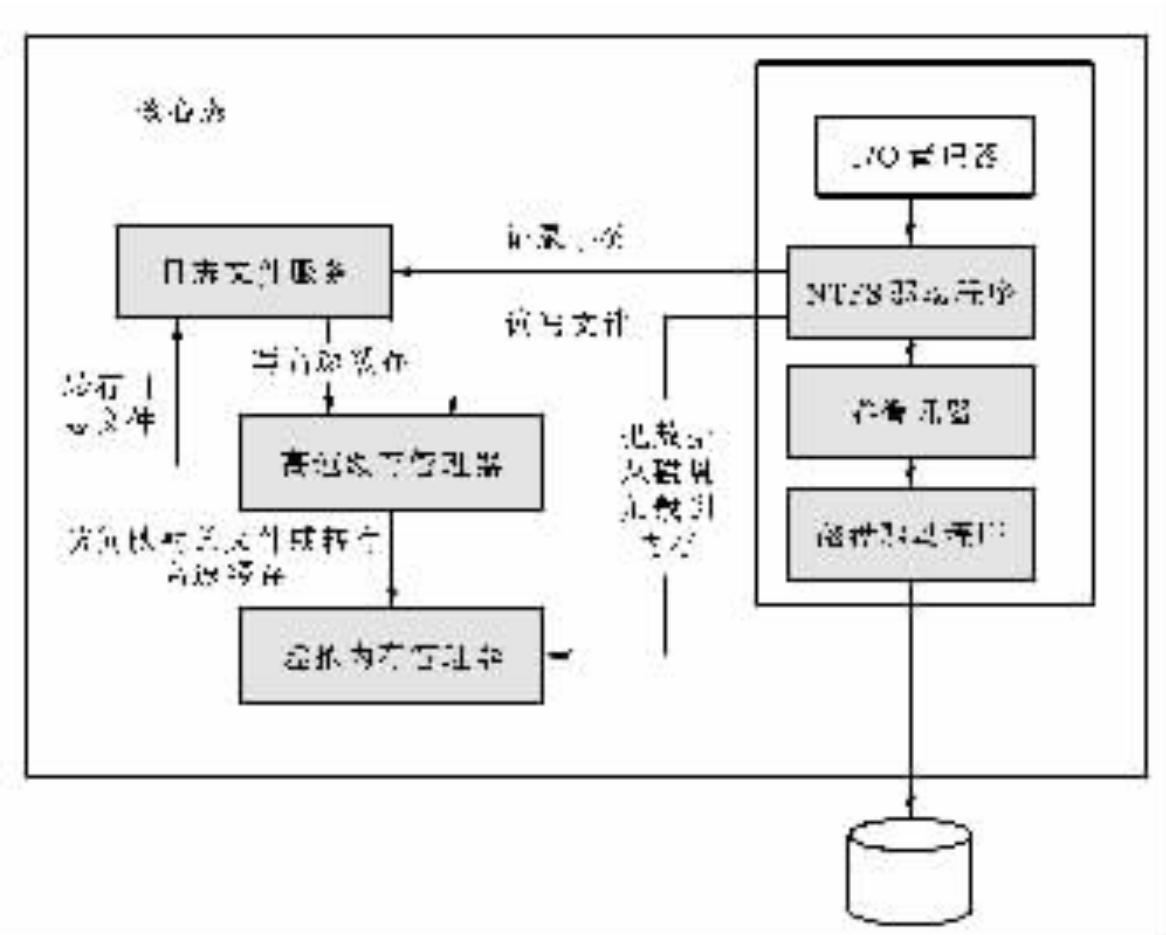


图 6-26 NTFS 及其相关组件

“延迟书写器”(lazy writer)来优化磁盘 I/O。延迟书写器是一组系统线程,在后台运行,调用内存管理器来刷新高速缓存的内容到磁盘上(异步磁盘写入)。

NTFS 把文件作为对象的实现方法允许文件被对象管理器共享和保护,对象管理器是管理所有执行体级别对象的 Windows 2000/XP 组件。应用程序创建和访问文件同对待其他 Windows 2000/XP 对象一样——依靠对象句柄。当 I/O 请求到达 NTFS 时,Windows 2000/XP 对象管理器和安全系统已经验证该调用进程有权以它试图访问的方式来访问文件对象。安全系统把调用程序的访问令牌同文件对象的访问控制列表中的项进行比较。I/O 管理器也将文件句柄转换为指向文件对象的指针。NTFS 使用文件对象中的信息来访问磁盘上的文件。

图 6-27 显示了将文件句柄链接到文件系统在磁盘上结构的数据结构。当 I/O 系统调用 NTFS 时,该句柄已经被转换成指向文件对象的指针。然后,NTFS 跟随几个指针从文件对象中得到文件在磁盘上的位置。一个代表打开文件系统服务的单一调用的文件对象,指向用于调用程序试图读取或写入文件属性的流控制块 SCB(Stream Control Block)。进程同时已打开数据属性和文件的用户定义的属性。SCB 代表单个的文件属性包含有关怎样在文件中查找指定的属性的某些信息。同一个文件的所有 SCB 都指向一个称做文件控制块 FCB(File

Control Block) 的公共数据结构。FCB 包括一个指针, 实际上是一个文件引用, 它指向基于磁盘的主控文件表(MFT) 中该文件记录的指针, NTFS 通过该指针获得文件访问权。

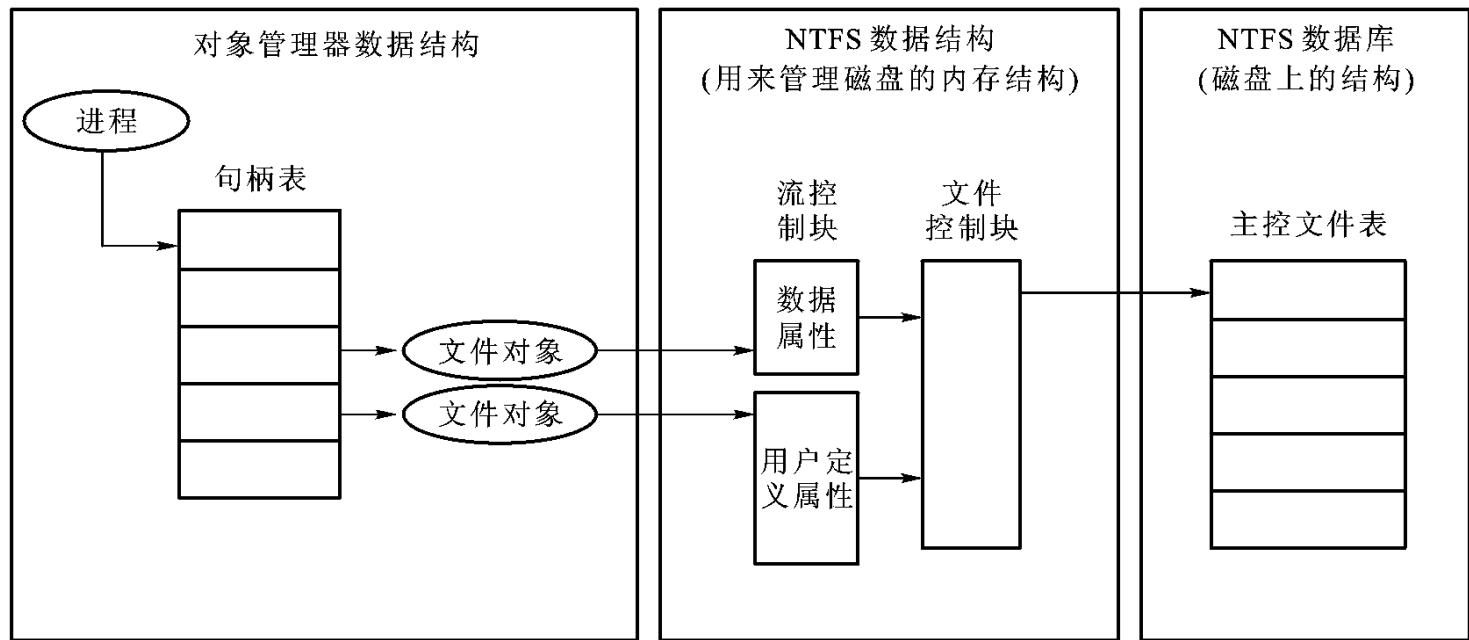


图 6-27 NTFS 数据结构

6.6.4 NTFS 在磁盘上的结构

物理磁盘可以组织成一个或多个卷。卷与磁盘逻辑分区有关, 由一个或多个簇组成, 随着 NTFS 格式化磁盘或磁盘的一部分而创建, 其中镜像卷和容错卷可能跨越多个磁盘。NTFS 将分别处理每一个卷, 同 FAT一样, NTFS 的基本分配单位是簇, 它包含整数个物理扇区; 而扇区是磁盘中最小的物理存储单位, 一个扇区通常存放 512 个字节, 但 NTFS 并不认识扇区。簇的大小可由格式化命令或格式化程序按磁盘容量和应用需求来确定, 可以为 512 B、1 KB、2 KB、…、最大可达 64 KB, 因而, 每个簇中的扇区数可为 1 个、2 个、直至 128 个。

NTFS 使用逻辑簇号 LCN(Logical Cluster Number) 和虚拟簇号 VLN(Virtual Cluster Number) 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, 以方便引用文件中的数据。簇的大小乘以 LCN, 就可以算出卷上的物理字节偏移量, 从而得到物理盘块地址。VCN 可以映射成 LCN, 所以不要求物理上连续。

NTFS 卷中存放的所有数据都包含在一个 NTFS 元数据文件中, 包括定位和恢复文件的数据结构、引导程序数据和记录整个卷分配状态的位图。

主控文件表 MFT(master file table) 是 NTFS 卷结构的中心, NTFS 忽略簇的大小, 每个文件记录的大小都被固定为 1 KB。从逻辑上讲, 卷中的每个文件在 MFT 上都有一行, 其中还包括 MFT 自己的一行。除了 MFT 以外, 每个 NTFS 卷还包括一组“元数据文件”, 其中包含用于

实现文件系统结构的信息。每一个这样的 NTFS 元数据文件都有一个以美元符号 (\$) 开头的名称, 虽然该符号是隐藏的。NTFS 卷中的其余文件是正常的用户文件和目录, 如图 6-28 所示。

| |
|----------------------------------------|
| MFT (\$Mft) /* 记录卷中所有文件的所有属性 |
| MFT 副本 (\$MftMirr) /* MFT 表前9行的副本 |
| 日志文件 (\$LogFile) /* 记录影响卷结构的操作, 用于系统恢复 |
| 卷文件 (\$Volume) /* 卷名, 卷的NTFS版本等信息 |
| 属性定义表 (\$AttrDef) /* 定义卷支持的属性类型, 如可恢复性 |
| 根目录 (/) /* 存放根目录内容 |
| 位图文件 (\$Bitmap) /* 盘空间位图, 每位一簇 |
| 引导文件 (\$Boot) /* Win2000/XP 引导程序 |
| 坏簇文件 (\$BadClus) /* 记录磁盘坏道 |
| 安全文件 (\$Secure) /* 存储卷的安全性描述数据库 |
| 大写文件 (\$UpCase) /* 包含大小写字符转换表 |
| 扩展元数据目录 (\$Ext. metadata Directory) |
| ... |
| 用户文件和目录 |
| ... |

图 6-28 MFT 中 NTFS 元数据文件记录

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, 就可能需要不止一个文件记录。在此情况下, 存放其他文件记录的位置的第一个记录就叫作基文件记录。

当 NTFS 首次访问某个卷时, 它必须装配该卷, 这时 NTFS 会查看引导文件, 找到 MFT 的物理磁盘地址。MFT 自己的文件记录是表中的第一项; 第二个文件记录指向位于磁盘中间的称做“MFT 镜像”的文件。如果因某种原因 MFT 文件不能读取时, 副本就用于定位元数据文件。

一旦 NTFS 找到 MFT 的文件记录, 它就从文件记录的数据属性中获得虚拟簇号 VCN 到逻辑簇号 LCN 的映射信息, 将其解压缩并存储在内存中。这个映射信息告诉 NTFS 组成 MFT 的运行构成放在磁盘的什么地方。然后, NTFS 再解压缩几个元数据文件的 MFT 记录, 并打开这些记录。接着, NTFS 执行它的文件系统恢复操作。最后, NTFS 打开剩余的元数据文件。现在用户就可以访问该卷了。

系统运行时, NTFS 会向另一个重要的元数据文件——日志文件 (log file) 写入信息。

NTFS 使用日志文件记录所有影响 NTFS 卷结构的操作, 包括文件的创建或改变目录结构的任何命令, 例如复制。日志文件被用来在系统失败后恢复 NTFS 卷。

MFT 中的另一项是为根目录保留的。它的文件记录包含一个存放于 NTFS 目录结构根部的文件和目录索引。当第一次请求 NTFS 打开一个文件时, 它开始在根目录的文件记录中搜索这个文件。打开文件之后, NTFS 存储文件的 MFT 文件引用, 以便当它在以后读写该文件时可以直接访问此文件的 MFT 记录。

NTFS 把卷的分配状态记录在位图文件 (bitmap file) 中。该位图文件的数据属性包含一个位图, 它们中的每一位代表卷中的一簇, 标识该簇是空闲的还是已被分配给了一个文件。

另一个重要的系统文件, 引导文件 (bootfile), 存储 Windows 2000/XP 的引导程序代码, 为了引导系统, 引导程序代码必须位于特定的磁盘地址。然而, 在格式化期间, Format 实用程序通过为这个区域创建一个文件记录将它定义为一个文件。创建引导文件使得 NTFS 坚持将磁盘上的所有事物都看成文件的原则。此引导文件以及 NTFS 元数据文件可以通过应用于所有 Windows 2000/XP 对象的安全描述体被分别地保护。使用这个“磁盘上的所有事物均为文件”模式也意味着虽然引导文件目前正被保护而不能编辑, 但引导程序还是可以通过一般的文件 I/O 来修改。

NTFS 还保留了一个记录磁盘卷中所有损坏位置的“坏簇文件”(bad – cluster) 和一个“卷文件”(volume file), 卷文件包含卷名、被格式化的卷的 NTFS 版本和一个位, 当设置此位时表明磁盘已经损坏, 必须用 Chkdsk 实用程序来恢复。

最后, NTFS 保持一个包含属性定义表 (attribute definition table) 的文件, 它定义了卷中支持的属性类型, 并指出它们是否可以被索引, 在系统恢复操作中是否可以恢复。

NTFS 卷中的文件是通过称为文件引用的 64 位值来标识的。文件引用由文件号和顺序号组成。文件号与文件在 MFT 中的文件记录的位置减 1 相对应 (如果文件有多个文件记录, 则对应于基文件记录的位置减 1)。文件引用的顺序号在每次重复使用 MFT 文件记录的位置时会被增加, 它使得 NTFS 能完成内部的一致性检查。

NTFS 将文件作为许多属性/值对的集合来存储, 其中的一个就是它包含的数据 (称为未命名的数据属性)。组成文件的其他属性包括文件名、时间标记、安全描述体以及可能附加的命名数据属性。每个文件的属性在文件中以单独的字节流存储。严格地讲, NTFS 不读取也不写入文件——它只是读取和写入属性流。NTFS 提供了这些属性操作: 创建、删除、读取 (字节范围) 以及写入 (字节范围)。读取和写入服务一般是对文件的未命名属性的操作。然而, 调用程序可以通过使用已命名的数据流句法来指定不同的数据属性。

NTFS 和 FAT 文件系统的文件名长度在 255 个字符以内。文件名可以包括 Unicode 字符、空格和多个句点, 并可以映射到 DOS 和 POSIX 的名字空间。

如果文件很小, 那么它所有的属性和值 (例如, 它的数据) 就放在文件记录中。当属性值

直接存放在 MFT 中时, 该属性叫作常驻属性 (resident attribute)。

每种属性以一个标准头开始, 在头中包含有关属性信息和 NTFS 用一般方法管理属性所需的信息。这个头是常驻的, 它记录了属性值是常驻的还是非常驻的。对于常驻属性, 头中还包含从头到属性值的偏移量和属性值长度。

当一个属性值直接存放在 MFT 中时, NTFS 访问该值的时间将大大缩短。取代了原有的在表中查找一个文件, 然后读出连续分配的单元以找到文件的数据 (例如, 像 FAT 文件系统那样), NTFS 只需访问磁盘一次, 就可立即重新得到数据。

当然, 许多文件和目录不能压缩成 1 KB 固定大小的 MFT 记录。如果一个特定的属性, 例如文件的数据属性, 由于它太大而不能包含在 MFT 文件记录中, 则 NTFS 将在磁盘上分配一个与 MFT 分开的 2 KB 的区域 (对于具有 4 KB 或更大簇的卷来说是 4 KB)。这个区域叫作一个运行 (run) (或者叫作一个区域 (extent)), 它用来存储属性值 (例如, 文件的数据)。如果属性值以后增加了 (例如, 用户向文件中追加了数据), 则 NTFS 将为额外的数据分配另一个运行。值存储在运行中而不是在 MFT 中的属性叫作非常驻属性 (nonresident attributes)。文件系统决定了一个特定的属性是常驻的还是非常驻的; 数据所在的位置对于要进行的访问操作来说是透明的。

当一个属性是非常驻属性时, 对于大文件可能是数据属性, 它的头包含 NTFS 需要在磁盘上查找属性值所必需的有关信息。

在标准属性中, 只有可以增长的属性才可以是非常驻的。对于文件, 可增长的属性是安全描述体、数据和属性列表。标准信息和文件名是常驻的。

一个大目录也可能包括非常驻属性 (或属性的一部分), 当 MFT 文件记录没有足够的空间来存储构成大目录的文件索引。部分索引被存储在索引根属性中, 其余的索引存储在叫作“索引缓冲区”(index buffers)的非常驻运行中。索引根、索引分配以及位图属性在此均使用简化形式表示。标准信息和文件名属性总是常驻的。对目录来说, 头和至少部分索引根属性值也是常驻的。

6.6.5 NTFS 的实现机制

1. 文件引用号

NTFS 卷上的每个文件都有一个 64 位的惟一标识, 称文件引用号 (File Reference Number)。它由两部分组成:一是文件号, 二是文件顺序号。文件号为 48 位, 对于该文件在 MFT 中的位置。文件顺序号随着每次文件记录的重用而增加, 是为 NTFS 进行内部一致性检查而设计的。

2. 文件命名

NTFS 路径名中的每个文件名/目录名的长度可达 255 个字节, 可以包含 Unicode 字符, 多个空格及句点。但是, MS – DOS 文件系统只支持 8 个字符的文件名加上 3 个字符的扩展名, 当 Win32 子系统创建一个文件名时, NTFS 会自动生成一个备用的 MS – DOS 文件名。

POSIX 子系统则需要 Windows 2000/XP 支持的所有应用程序执行环境中的最大的名字空间, 因此 NTFS 的名字空间等于 POSIX 的名字空间。POSIX 子系统甚至可创建在 Win32 和 MS – DOS 中不可见的名称。

3. 文件属性

NTFS 将文件作为属性/属性值的集合来处理, 这一点与其他文件系统不一样。文件数据就是未命名属性的值, 其他文件属性包括文件名、文件拥有者、文件时间标记等。

每个属性由单个的流(stream)组成, 即简单的字符队列。严格地说, NTFS 并不对文件进行操作, 而只是对属性流的读写。NTFS 提供对属性流的各种操作包括: 创建、删除、读取以及写入。读写操作一般是针对文件的未命名属性的, 对于已命名的属性则可以通过已命名的数据流句法来进行操作。

当一个文件很小时, 其所有属性值可存在 MFT 的文件记录中。图 6 – 29 显示了一个用于小文件的 MFT 记录。当属性值能直接存放在 MFT 中时, 该属性就称为常驻属性(resident attribute)。有些属性总是常驻的, 这样 NTFS 才可以确定其他非常驻属性。例如, 标准信息属性和文件名属性就总是常驻属性。标准信息属性包括基本文件属性(如只读、存档); 时间标记(如文件创建和修改时间); 文件链接数等。

| | | |
|------|-----|------|
| 标准信息 | 文件名 | 文件数据 |
|------|-----|------|

图 6 – 29

每个属性都是以一个标准头开始的, 在头中包含该属性的信息和 NTFS 通常用来管理属性的信息。该头总是常驻的, 并记录着属性值是否常驻。对于常驻属性, 头中还包含着属性值的偏移量和属性值的长度。如果属性值能直接存放在 MFT 中, 那么 NTFS 对它的访问时间就将大大缩短。NTFS 只需访问磁盘一次, 就可立即获得数据。

大文件或大目录的所有属性, 就不可能都常驻在 MFT 中。如果一个属性(如文件数据属性)太大而不能存放在只有 1 KB 的 MFT 文件记录中, 那么 NTFS 将从 MFT 之外分配区域。这些区域通常称为一个扩展(extent), 它们可用来存储属性值, 如文件数据。如果以后属性值又增加, NTFS 将会再分配一个扩展, 以便用来存储额外的数据。值存储在扩展中而不是

在 MFT 文件记录中的属性称为非常驻属性 (nonresident attribute)。NTFS 决定了一个属性是常驻还是非常驻的;而属性值的位置对访问它的进程而言是透明的。

当一个属性为非常驻时,如大文件的数据,它的头包含了 NTFS 需要在磁盘上定位该属性值的有关信息。图 6-30 显示了一个存储在两个扩展中的非常驻属性。

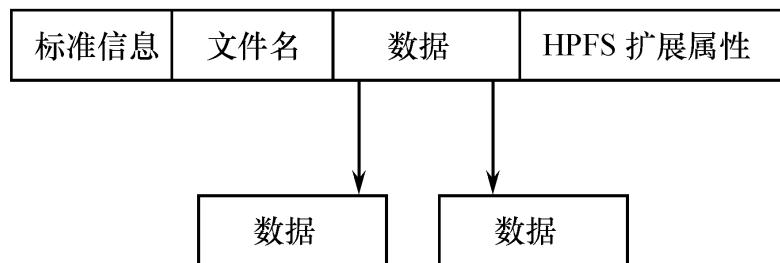


图 6-30

在标准属性中,只有可以增长的属性才是非常驻的。对文件来说,可增长的属性有数据、属性列表等。标准信息和文件名属性总是常驻的。

当一个文件(或目录)的属性不能放在一个 MFT 文件记录中,而需要分开分配时,NTFS 通过 VCN - LCN 之间的映射关系来记录扩展情况。LCN 用来为整个卷中的簇按顺序从 0 到 n 编号,而 VCN 则用来对特定文件所用的簇按逻辑顺序从 0 到 m 进行编号。图 6-31 表示一个非常驻数据属性的扩展所使用的 VCN 与 LCN 编号。

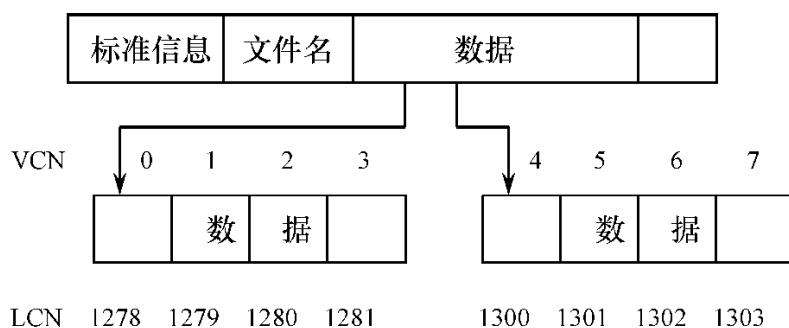


图 6-31

当该文件含有超过 2 个扩展时,则第三个扩展从 VCN8 开始,数据属性头部含有前两个扩展的 VCN 映射,这便于 NTFS 对磁盘文件分配的查询。为了便于 NTFS 快速查找,具有多个扩展文件的常驻数据属性头中包含了图 6-32 所示的 VCN - LCN 映射关系。

虽然数据属性常常因太大而存储在扩展中,但是其他属性也可能因 MFT 文件记录没有足够空间而需要存储在扩展中。另外,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)。这种情况下,一个叫作“属性列表”(attribute)的属性就加进来。属性列表包括文件属性的名称和类

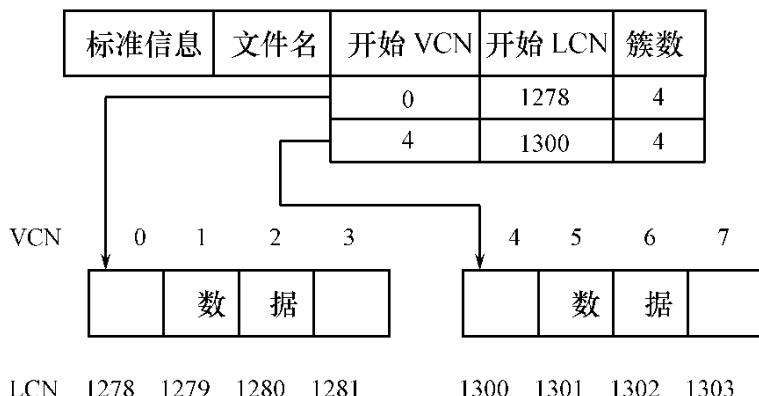


图 6-32

型代码以及属性所在 MFT 的文件引用。属性列表通常用于太大或太零散的文件, 这种文件因 VCN – LCN 映射关系太大而需要多个 MFT 文件记录。具有超过 200 个扩展的文件通常需要属性列表。

4. 文件目录

在 NTFS 系统中, 文件目录仅仅是文件名的一个索引。NTFS 使用了一种特殊的方式把文件名组织起来, 以便于快速访问。当创建一个目录时, NTFS 必须对目录中的文件名属性进行索引。

参见图 6-33, 小文件和小目录的所有属性, 均可以在 MFT 中常驻。小文件的未命名属性可以包括所有文件数据。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。

| 标准信息 | 文件名 | 文件索引 | | | 空 |
|------|-----|------|------|------|---|
| | | 文件 1 | 文件 2 | 文件 3 | |
| | | | | | |

图 6-33

一个大目录也可以包括非常驻属性, 参见图 6-34。在该例中, MFT 文件记录没有足够空间来存储大目录的文件索引。其中, 一部分索引存放在索引根属性中, 而另一部分则存放在叫作“索引缓冲区”(index buffer)的非常驻扩展中。对目录而言, 索引根的头及部分值应是常驻的。

一个目录的 MFT 记录将其目录中的文件名和子目录名进行排序, 并保存在索引根属性中。然而, 对于一个大目录, 文件名实际存储在组织文件名的固定 4 KB 大小的索引缓冲区中。索引缓冲区是通过 B + 树数据结构实现的。B + 树是平衡树的一种, 对于存储在磁盘上的数据来说, 平衡树是一种理性的分类组织形式, 因此使查找一个项时所需的磁盘访问次数减到最少。根索引属性包含 B + 树的第一级(根子目录)并指向包含下一级(大多数是子目

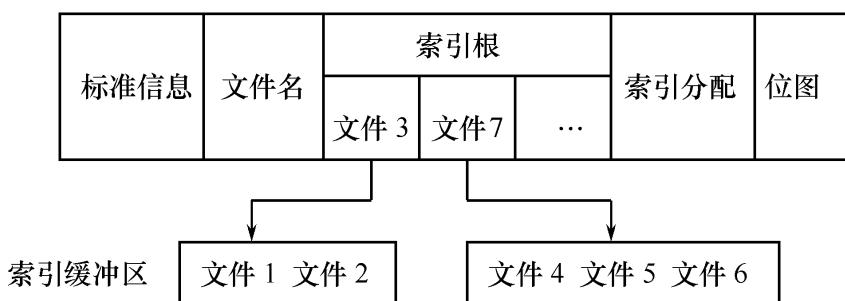


图 6-34

录,也可能是文件)的索引缓冲区中。

上图只显示了根索引属性中的文件名和索引缓冲区。但是索引中的每一项还包括了位于 MFT 中的描述文件所在位置的文件引用以及文件时间和文件大小等信息。NTFS 根据文件的 MFT 记录来复制时间标记和文件大小的信息。这种技术需要将更新信息写在两个地方,因此比较麻烦。但是,这仍是一个提高目录浏览速度的好方法,因为它可以在文件系统不打开目录中任何文件的情况下显示每个文件的时间标记和大小。

索引分配属性包含索引缓冲区的 VCN 到 LCN 的映射,而位图属性跟踪在索引缓冲区中哪些 VCN 是在使用而哪些是空闲的。上图中显示了每个文件项占有一个 VCN,而实际上多个文件项被包装在同一个簇中,每个 4 KB 大小的索引缓冲区可容纳 20 到 30 个文件项。

6.6.6 NTFS 可恢复性支持

NTFS 通过日志记录(logging)来实现文件的可恢复性。所有改变文件系统的子操作在磁盘上运行前,首先被记录在日志文件中。当系统崩溃后的恢复阶段,NTFS 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,从而,保证磁盘上文件的一致性,这种技术称预写日志记录(write-ahead logging)。

文件可恢复性的实现要点如下:

- 日志文件服务 LFS(Log File Service) 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。LFS 分两个区域:重启动区(restart area)和无限记录区域(infinite logging area),前者保存的信息用于失败后的恢复,后者用于记录日志。NTFS 不直接存取日志文件,而是通过 LFS 进行,LFS 提供了打开、写入、向前、向后、更新等操作。

- 日志记录类型 LFS 允许用户在日志文件中写入任何类型的记录,更新记录(update records)和检查点记录是 NTFS 支持的两种主要类型的记录,它们在系统恢复过程中起主要作用。更新记录所记录的是文件系统的更新信息,是 NTFS 写入日志文件中的最普通的记录。每当发生下列事件时: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。检查点记录由 NTFS 周期性写入到日志文

件中,同时还在重启时记录 LSN,在发生系统失败后,NTFS 通过存在检查点记录中的信息定位日志文件中的恢复点。

- 可恢复性的实现 NTFS 通过 LFS 来实现可恢复功能,但这种恢复只针对文件系统的数据,不能保证用户数据的完全恢复。NTFS 在内存中维护两张表:事务表用来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;脏页表用来记录在高速缓存中还未写入磁盘的包括改变 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。要实现 NTFS 卷的恢复,NTFS 要对日志文件进行三次扫描:分析扫描、重做扫描和撤销扫描。

- 可恢复性操作步骤,1) NTFS 首先调用 LFS 在日志文件中记录所有改变卷结构的事务;2) NTFS 执行在高速缓存中的更改卷结构的操作;3) 高速缓存管理器调用 LFS 把日志文件刷新到磁盘;4) 高速缓存管理器把该卷的变化(事务本身)最后被刷新到磁盘。

6.6.7 NTFS 安全性支持

NTFS 卷上的每个文件和目录在创建时创建人就被指定为拥有者,拥有者控制文件和目录的权限设置,并能赋予其他用户访问权限。NTFS 为了保证文件和目录的安全及可靠性,制定了以下的权限设置规则:

- 只有用户在被赋予其访问权限或属于拥有这种权限的组,才能对文件和目录进行访问;

- 权限是累积的,如果组 A 用户对一个文件拥有“写”权限,组 B 用户对该文件只有“读”权限,而用户 C 同属两个组,则 C 将获得“写”权限;

- “拒绝访问”权限优先高于其他所有权限。如果组 A 用户对一个文件拥有“写”权限,组 B 用户对该文件有“拒绝访问”权限,那么同属两个组的 C 也不能读文件;

- 文件权限始终优先于目录权限;

- 当用户在相应权限的目录中创建新的文件或子目录时,创建的文件或子目录继承该目录的权限;

- 创建文件或目录的拥有者,总可以随时更改对文件或子目录的权限设置来控制其他用户对该文件或目录的访问。

在信息交流高度发达的网络时代,很难防止非法用户对某些重要数据的窃取和破坏。高度机密的关键数据,除了设置权限外,还可通过加密技术来保障其安全性。文件的加密指对文件中的内容,按照一定的变换规则进行重新编码,从而得到无法正常显示和阅读的文件。所以,除了上面介绍的对文件和目录设置安全性权限外,对文件内容进行加密是一种十分有效的安全性措施,下面简单介绍 NTFS 的安全性支持——加密文件系统 EFS (Encrpyted File System)。

EFS 加密技术是基于公共密钥的, 它用一个随机产生的文件密钥 FEK (File Encryption Key), 通过加强型的数据加密标准 DESX (Data Encryption Standard) 算法对文件进行加密。DESX 使用同一个密钥来加密和解密数据, 这是一种对称加密算法 (Symmetric Encryption Algorithm), 加解密数据速度快, 适用于大数据量文件。EFS 使用基于 RSA (Rivest Shamir Adleman) 的公共密钥加密算法对 FEK 进行加密, 并把它和文件存储一起, 形成了文件的一个特殊的 EFS 属性字段——数据加密字段 DDF (Data Decryption Field)。解密时, 用户用自己的私钥解密存储在文件 DDF 中的 FEK, 然后, 再用解密后得到的 FEK 对文件数据进行解密, 得到文件的原文。

6.7 本章小结

现代计算机系统都使用高速大容量磁盘存储器作为系统的后援存储器。用户要想通过硬件细节来直接控制读写数据是办不到的。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了的文件的设备, 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, 它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有效的文件使用和操作方法。

文件这一术语不但反映了用户概念中的逻辑结构, 而且和存放它的辅助存储器的存储结构紧密相关。所以, 同一个文件必须从逻辑文件和物理文件两个侧面来观察它, 本章的内容围绕这两方面展开。在介绍了文件的概念、文件命名、文件类型、文件属性、文件存取方式后, 先讨论了文件的逻辑结构, 也就是用户如何来配置信息构造逻辑文件。文件的逻辑结构分两种: 一种是流式文件, 另一种是记录式文件。大多数现代操作系统对用户仅仅提供流式文件, 记录式文件往往由高级语言或数据库管理系统提供, 但 IBM 大型操作系统直接提供记录式文件供用户使用。逻辑上的文件总得以不同方式保存到物理存储设备的存储介质上去, 所以, 文件的物理结构是指逻辑文件在物理存储空间中存放方法和组织关系。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。因而, 其优劣直接影响文件系统的性能。本章中讨论了各种文件的物理结构, 包括: 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工具, 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型目录结构, 目录可以按不同方法组织, 有的把文件名、文件属性、磁盘地址放在一起存放; 有的仅放文件名字, 其他信息放到文件索引 i-node 节点中。

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的任务所必须,而且,还可以节省大量的外存空间,减少由于文件复制而增加的访问外存次数。本章中,讨论了文件的静态共享、文件的动态共享和文件的符号链接共享。

对用户来说,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,主要在 ch7 进行讨论。

用户采用不同的存取方法如顺序存取、直接存取或索引存取来访问文件,系统在组织物理文件时,应根据存储设备特性、使用的存取方法和性能要求来决定存储结构。对于顺序存取的磁盘文件可组织成顺序文件、连接文件或索引顺序文件,但文件若存在磁带上则只能组织成顺序文件。对于随机存取的文件,只能存在磁盘上并可组织成索引文件或直接文件。文件系统实现了按名存取,使用户不必涉及存储设备的物理细节,不必考虑文件的物理结构,由文件系统来完成所有的读写操作。但是为了正确地实现文件存取,保证用户文件的安全可靠,实现不同用户对同一文件的共享,用户必须使用文件系统提供的文件操作和规定的操作次序来请求使用文件。常见的文件操作有:打开文件、建立文件、读写文件、控制文件、关闭文件和删除文件。

本章中还讨论了两种特殊类型的文件即内存映射文件和虚拟文存系统。用户进程都要使用文件,用户用文件操作类系统调用使用文件,不但管理复杂,而且开销较大。针对这一点,许多操作系统提供了一种新的内存映射文件技术,它把进程的虚地址空间与某一个盘文件关联起来,使得进程对文件的存取转化为对关联存储区域的访问,具有方便易用、节省空间、便于共享、灵活高效的优点。为了让一个操作系统同时支持多种文件系统,虚拟文件系统概念被提出来了,它要实现以下目标:应同时支持多种文件系统;多个文件系统应与传统的单一文件系统没有区别,在用户的面前表现为一致的接口;对通过网络共享文件提供完全支持,访问远程结点上的文件系统应与访问本地结点的文件系统一致;可以开发出新的文件系统,以模块方式加入到操作系统中。虚拟文件系统的主要设计思想是两个层次:(1)在对多个文件系统的共同特性进行抽象基础上,形成一个与具体文件系统实现无关的虚拟层,并在此层次上定义与用户的一致性接口;(2)文件系统具体实现层使用类似开关表技术进行文件系统转接,实现各文件系统的具体细节,每个文件系统是自包含的,包含文件系统实现的各种设施,如超级块、i 节点区、数据区以及各种数据结构和对文件的操作函数。

习题六

一、思考题

1. 叙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. 叙述各种文件物理组织方式的主要优缺点。
7. 什么是记录键？它有何用处？
8. 连接文件的连接字可以如下定义：
 - (1) 连接字的内容为(上一块的地址) \oplus (下一块的地址)。
 - (2) 首块连接字内容为(下一块的地址)。
 - (3) 末块连接字内容为(上一块的地址)。
- 其中， \oplus 是模 2 按位加。试述这种连接字的主要特点。
9. 文件系统提供的主要文件操作有哪些？叙述各自的主要功能。
10. 简述 Windows 2000 NTFS 文件系统的主要特点和优点。
11. 简述 Windows 2000 NTFS 文件系统的实现要点。
12. 简述 Windows 2000 NTFS 文件系统的可恢复性支持。
13. 简述 Windows 2000 NTFS 文件系统的安全性支持。
14. 解释：目录文件、文件目录、文件目录项(FCB)。
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，你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，经常按一定顺序访问；
 - (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一个拷贝来实现共享，另一些系统为每个用户提供一个共享文件拷贝，试讨论各自的优缺点。
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. 试说明树型目录结构中线性检索法的检索过程？
28. 试说明采用两分法检索文件目录的检索过程？
29. 什么叫“按名存取”？文件系统如何实现文件的按名存取？

30. 文件目录在何时建立? 它在文件管理中起什么作用?
31. 在 UNIX 中, 当一个进程所访问的一页既不在内存又不在文件系统中时, 该页面可能在什么地方? 存储管理模块是如何把它调入内存的?
32. UNIX/Linux 把文件描述信息从文件目录项中分离出来, 为什么?
33. UNIX/Linux 的 *i* 节点是文件内容的一部分, 对吗? 请说明理由。
34. UNIX/Linux 进程 0 的主要任务是什么?
35. UNIX/Linux 采用空闲块成组连接的方式管理文件空间, 试给出申请和归还一块的工作流程。
36. 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?
37. 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该操作, 为什么?
38. 文件系统提供系统调用或命令 rename 实现文件改名, 同样也可以通过把文件拷贝到新文件并删去原文件来实现文件更名, 试述两种方法的不同。
39. 文件系统根目录的长度是否要加限制, 为什么?
40. 在 UNIX/Linux 中, 用户 2 对用户 1 的一个文件建立了一个链接, 当用户 1 删除了此文件后, 此时用户 2 访问该文件, 其结果如何?

二、应用题

1. 磁带卷上记录了若干文件, 假定当前磁头停在第 *j* 个文件的文件头标前, 现要按名读出文件 *i*, 试给出读出文件 *i* 的步骤。
2. 假定令 $B = \text{物理块长}$ 、 $R = \text{逻辑记录长}$ 、 $F = \text{块因子}$ 。对定长记录(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 位的位示图管理盘空间, 试问:(1)位示图需多少个字? (2)第 *i* 字第 *j* 位对应的块号是多少? (3)并给出申请/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; 用户乙要用到文件 A、D、E、F。已知用户甲的文件 A 与用户乙的文件 A 实际上不是同一文件; 甲、乙两用户的文件 D 和 E 正是同一文件。试设计一种文件系统组织方案, 使得甲、乙两用户能共享该文件系统又不致造成混乱。
5. 在 UNIX 中, 如果一个盘块的大小为 1 KB, 每个盘块号占 4 个字节, 即每块可放 256 个地址。请转换下列文件的字节偏移量为物理地址:(1) 9999; (2) 18000; (3) 420000。
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 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---, 该文件的文件主 $uid = 12$, $gid = 1$, 另一个用户的 $uid = 6$, $gid = 1$, 是否允许该用户执行文件 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 字节, 并依次存放在 50、121、75、80、63 号磁盘块上。若要存取文件的第 1 569 逻辑字节处的信息, 问要访问哪

一个磁盘块?

9. 一个 UNIX/Linux 文件,如果一个盘块的大小为 1 KB, 每个盘块占 4 个字节,那么,若进程欲访问偏移为 263 168 字节处的数据,需经过几次间接?

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 到 9 项为直接寻址方式,后 3 项为间接寻址方式。试描述出文件数据块的索引方式;给出对文件第 n 个字节(设块长 512 字节)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 号逻辑记录读入内存缓冲区的过程。

12. 若某操作系统仅支持单级目录,但允许该目录有任意多个文件,且文件名可任意长,试问能否模拟一个层次式文件系统?如能的话,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. 有一个磁盘组共有 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,现问:(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 5 个字节,则什么时候空白文件目录大于位示图?

15. 某磁盘共有 100 个柱面,每个柱面有 8 个磁头,每个盘面分 4 个扇区。若逻辑记录

与扇区等长,柱面、磁道、扇区均从 0 起编号。现用 16 位的 200 个字(0 ~ 199)来组成位给示图来管理盘空间。现问:(1)位示图第 15 个字的第 7 位为 0 而准备分配某一记录,该块的柱面号、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 68B,而每块大小为 8 KB。问在索引节点中有多大空间给给磁盘块指针?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. 设一个文件由 100 个物理块组成,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 次数(假如头指针和索引表均在内存中):(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。

第七章 操作系统的安全与保护

7.1 安全性概述

计算机安全性涉及内容非常广泛,它既包括物理方面的,如计算机环境、设施、设备、载体和人员,需要采取安全的行政管理上的对策和措施,防止突发性或人为的损害或破坏;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很多。首先,操作系统是一个共享资源系统,支持多用户同时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,涉及到各种安全性问题;其次,随着计算机网络的迅速发展,除了信息的存储和处理外,存在大量数据传送操作,客户机要访问服务器,一台计算机要传送数据给另一台计算机,这一过程中对安全的威胁极大,于是就需要有网络安全和数据信息的保护,防止入侵者恶意破坏;另外,在应用系统中,主要依赖数据库来存储大量信息,它是各个部门十分重要的一种资源,数据库中的数据会被广泛应用,特别是在网络环境中的数据库,这就提出了信息系统——数据库的安全问题;最后,计算机安全中的一个特殊问题是计算机病毒,需要采取措施预防、发现、解除它。

计算机系统的安全性和可靠性是两个概念,可靠性指硬件系统正常持续运行的程度,目标为反故障;安全性是指不因人为疏漏或蓄谋作案而导致信息资源被泄漏、篡改和破坏,目标是反泄密。可靠性是基础,安全性更为复杂。鉴于计算机系统自身的脆弱性和计算机犯罪现象的普遍存在,建造一个安全的计算机信息系统决非易事。一般地说,信息系统的安全模型涉及到: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,由于它的特殊地位,上述计算机安全性问题大部份要求操作系统来保证,所以,操作系统的安全性是计算机系统安全性的基础。本章主要讨论操作系统的安全性,主要包括以下内容:

- 安全策略。主要功能是描述一组用于授权使用其计算机及信息资源的规则。
- 安全模型。主要功能是精确描述系统的安全策略。它是对系统的安全需求,以及如何设计和实现安全控制的一个清晰的全面的理解和描述。
- 安全机制。主要功能是实现安全策略描述的安全问题,它关注的是如何实现系统的安全性。它主要包括:认证机制(Authentication)、授权机制(Authorization)、加密机制

(Encryption)、审计(Audit)等。

美国国防部1985年提出的“计算机可信系统评价准则”中把安全级别分为七个：

- D 最低安全性
- C1 自主存取控制
- C2 较完善的自主存取控制、广泛的审计
- B1 强制存取控制, 安全标识
- B2 良好的安全体系结构、形式化安全模型、抗渗透能力
- B3 全面的访问控制(安全内核)、可信恢复、高抗渗透能力
- A1 形式化认证、非形式化代码、一致性证明

目前流行的几个操作系统的安全性分别为:DOS:D 级;Windows NT 和 Solaris:C2 级;OSF/1:B1;UNIX Ware 2.1:B2 级。

7.2 安全威胁及其类型

为了理解现有的安全威胁(攻击)及其类型,需要对安全要求下一个定义。对计算机系统和网络通信提出了四项安全要求:

- 机密性(confidentiality)。要求计算机系统中的信息只能由被授权者进行规定范围内的访问,这种访问可读或可视,如打印、显示以及其他形式,也包括简单显示一个对象的存在。
- 完整性(integrity)。要求计算机系统中的信息只能被授权用户修改,修改操作包括写、改写、改变状态、删除和创建等。
- 可用性(availability)。防止非法独占资源,每当合法用户需要时,总能访问到合适的计算机系统资源,为其提供所需的服务。
- 真实性(authenticity)。要求计算机系统能证实用户的身份,防止非法用户侵入系统,以及确认数据来源的真实性。

计算机或网络系统在安全性上受到威胁的类型可以这样来刻画:根据计算机系统提供信息的功能,从源端,如从一台计算机中的某个文件或主存的某个区域内容,到目的端,即另一台计算机的一个文件或用户存储区中存在数据的流动。下面的图 7-1a 显示了这种一般的数据流动。图中的其余部分显示了下面的四种普通的威胁类型:

- 切断。系统的资源被破坏或变得不可用或不能用。这是对可用性的威胁,如破坏硬盘、切断通信线路或使文件管理失效。
- 截取。未经授权的用户、程序或计算机系统获得了对某资源的访问。这是对机密性的威胁,如在网络中窃取数据及非法拷贝文件和程序。

- 篡改。未经授权的用户不仅获得了对某资源的访问,而且进行篡改。这是对完整性的攻击,如修改数据文件中的值,修改网络中正在传送的消息内容。
- 伪造。未经授权的用户将伪造的对象插入到系统中。这是对合法性的威胁,如非法用户把伪造的消息加到网络中或向当前文件加入记录。

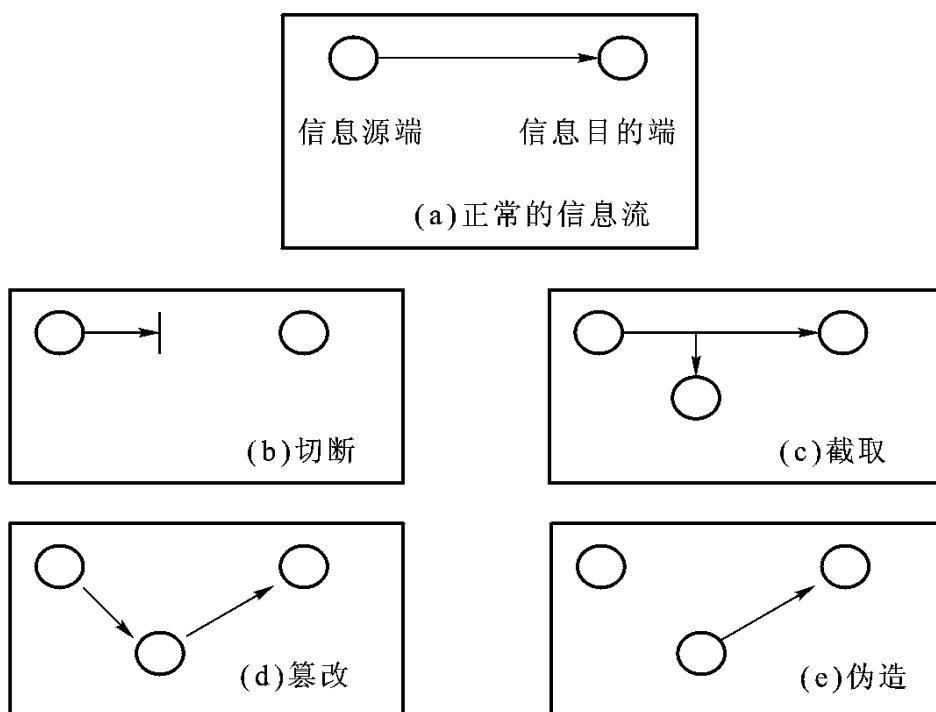


图 7-1 安全威胁

计算机系统的资源分为硬件、软件、数据、以及通信线路与网络等几种。表揭示了每种资源类型所面临的威胁的情况,下面详细说明威胁的本质:

表 7-1 安全威胁和资源

| 可用性 | 机密性 | 完整性/可靠性 | |
|------|---------------------|---------------------------|---------------------------------|
| 硬件 | 设备被偷或破坏,故拒绝服务 | | |
| 软件 | 程序被删除,故拒绝用户访问 | 非授权的软件拷贝 | 工作程序被更改,导致在执行期间出现故障,或执行一些非预期的任务 |
| 数据 | 文件被删除,故拒绝用户访问 | 非授权读数据。通过对统计数据的分析揭示了潜在的数据 | 现有的文件被修改,或伪造新的文件 |
| 通信线路 | 消息被破坏或删除,通信线路或网络不可用 | 读消息;观察消息的流向规律 | 消息被更改、延迟、重排序,伪造假消息 |

1. 硬件(中央处理器、存储器、磁带、打印机、磁盘等)

对计算机系统硬件的威胁主要表现在可用性方面。硬件最容易受到攻击,也最不容易得到自动控制。威胁包括对设备的有意或无意的破坏及偷窃。个人计算机和工作站的急剧增加以及局域网的日益广泛使用增加了在这方面的潜在损失,需要物理上的和行政管理上的安全措施来处理这些威胁。

2. 软件(操作系统、实用程序、应用程序等)

软件所面临的一个主要威胁是对可用性的威胁。软件,尤其是应用软件,非常容易被删除。软件也可能被修改或破坏,从而失效。较好的软件配置管理(如最新版本备份)可以获得高可用性。另一个更难处理的问题是对软件的修改导致程序仍能运行但其行为却发生了变化。计算机病毒和相关的威胁就属于这一类。最后一个问题是软件的保密性,尽管采用了许多措施,但对于软件进行非法拷贝问题仍然未获解决。

3. 数据

硬件与软件的安全性一般与计算机中心的专业人员有关,个别的与个人计算机用户有关。一个更普遍的问题是数据安全性,它包括个人、小组以及企业所控制的文件和其他形式的数据。与数据有关的安全性涉及面广,包括可用性,机密性和完整性。

对于可用性,主要是对数据文件有意和无意的窃取和破坏。机密性方面最受关注的是对数据文件或数据库的未授权访问,这一领域已成为计算机安全性研究和努力的一个重要课题。另一个安全威胁涉及到数据分析,在提供摘要和合计信息的统计数据库的使用上。合计信息的存在可能不会威胁到所涉及的个人的私密性,然而,随着对统计数据库使用的增加,个人信息被暴露的可能性就随之增加。从根本上讲,各组成个体的特性要通过仔细的分析才能识别出来。这里给出一个简单的例子,某表记录了 A、B、C、D 的收入的合计情况。另一个表记录了 A、B、C、D 和 E 的收入的合计情况,两个合计之间的差值将是 E 的收入。当需要更多的联合数据集合时,会导致问题恶化。数据的完整性是很紧要的问题,对数据文件的更改可能造成较小的危害,也可能造成灾难性后果。

4. 通信线路和网络

通信系统是用来传送数据的,与数据相关的可用性、机密性、完整性和真实性对网络安全同样重要,这里的威胁被分为被动的和主动的。图 7-2 给出了针对通信线路和网络的安全威胁的分类。下面详细的描述各种威胁的本质。

被动威胁在本质上是对传输过程进行窃听或截取,攻击者的目的是非法获得正在传输的信息,了解其内容和数据性质。这包括两种威胁:消息内容泄漏和消息流量分析。

- 消息内容泄漏很容易理解,电话交谈,电子邮件消息以及传输的文件中可能含有敏感的或秘密的信息,希望能防止攻击者获得这些传递的信息内容。

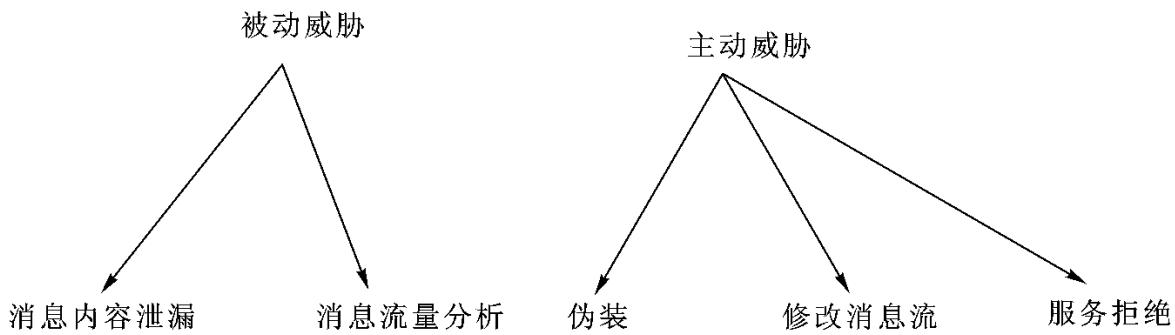


图 7-2 主动和被动威胁

- 消息流量分析则更加巧妙,假设有一个方法屏蔽了消息或其他信息的内容,使得攻击者很难捕获消息或即使获得了消息也不能从中提取信息。屏蔽内容的常用技术是加密。但对具有加密保护的机制,攻击者仍能够观察这些消息的模式,确定通信主机的位置和身份,并能观察到正在交换信息的频率与长度,而这些信息对猜测发生的通信的性质可能是有用的。

被动威胁很难检测出来,因为它不包含对数据流的更改,不干扰网络中信息的流动。然而,防止它却是很方便的,对付被动威胁的关键是在预防,而不是检测。

主动威胁不但截获数据,而且还冒充用户对系统中的数据进行修改、删除或生成伪造数据,可分为伪装,修改信息流和服务拒绝。

当一个实体假装成另外一个不同的实体时,就发生了伪装的情况。伪装威胁通常需要借助其他形式作主动攻击,这种攻击可以通过捕获并重传合法序列进行。修改消息流简单的说就是修改合法消息的某些部分,或者消息被延迟或重排序,从而,产生未预期的结果。服务拒绝就是阻止或禁止对通信设备的正常使用和管理。这种威胁可能有一个特定的目标,例如,强制性把所有消息传到一个特定目标。另一种形式的服务拒绝是破坏整个网络,方法是使网络失效或加入大量消息使得网络超载以降低性能。

主动威胁体现了与被动威胁的相反的特性。尽管被动威胁难于检测,却有预防的方法。主动威胁很难预防,这是因为预防需要在所有时刻对所有通信设备和通路进行物理保护。因而,对于主动威胁只能努力检测出这些威胁,并从威胁导致的破坏和延迟中恢复出来。近年来,广泛使用防火墙作为防范网络主动威胁的手段,它能使网络内部与 Internet 之间或与其他外部网络之间互相隔离,限制网络互访,保护网络内部资源,防止外部入侵。目前常用的防火墙技术主要有:基于对包的 IP 地址校验的包过滤型技术、通过代理服务器隔离的服务代理型技术、建立状态监视服务模块的状态监测型技术等。过滤型防火墙的依据是网络中的分包传输技术,网络上以包为单位进行数据传输,数据包中含有源地址、目标地址、TCP/UDP 源端口和目标端口等。防火墙通过读取数据包中的地址信息来判断这些包是否来自安全的站点,一旦发现有问题,便将数据拒之门外;代理型防火墙位于客户机与服务器

之间,完全阻断了两者的数据交流,因为,数据请求和获得先通过代理服务器,所以,恶意侵害很难伤害内部网络系统,安全性高于过滤型产品;监测型防火墙能够对各层数据进行主动和实时监测,在对数据进行分析的基础上,有效地判断出各层的非法侵入,同时它还带有分布式探测器,不仅检测来自网络外部的攻击,也对来自内部的恶意破坏有防范作用,具有安全性好及功能强的特点。

7.3 保 护

7.3.1 操作系统保护层次

多道程序设计技术的开发使得用户之间可以共享资源。共享的不仅有处理器,而且包括存储器,I/O设备,例如,磁盘和打印机、程序、数据。对资源共享的需要引起了对资源保护的需要。操作系统可能在下列层次提供保护:

- 无保护。当敏感的过程是在独立的时间内运行时,这是合适的。
- 隔离。这种方法意味着每个进程并不感觉到其他进程的存在,当然也没有任何共享资源或通信行为。每个进程具有自己的地址空间,文件和其他对象。
- 共享或不共享。对象(如一个文件或一个内存区)所有者宣布它是公有的或私有的。前者,任何进程可以访问该对象;后者,只有所有者的进程才能访问该对象。
- 通过访问控制的共享。操作系统检查特定用户对特定对象的访问的许可,因此,操作系统就像位于用户和对象之间的护卫,保证只能发生已授权的访问。
- 通过权能的共享。扩展了访问控制的概念,允许动态生成对共享对象的访问权力。
- 限制对象的使用。这种形式的保护不仅限制了对对象的访问,而且还限制存取后对对象的使用方式。如,一用户已被允许浏览一个机密文件,但不能打印它;另一个例子是用户为了推导分析被允许访问一个数据库,从而得到统计结果,但不能确定具体的数据值。

前面各项大致是按实现的困难程度递增的顺序列出的,同样也是按它们所提供的保护的出色程度递增的顺序排列的。一个特定的操作系统可能对不同的对象,用户或应用程序提供不同程度的保护。操作系统需要在资源共享和资源保护之间做出平衡,这将增强需要保护个人用户资源的计算机系统的用途。下面介绍操作系统对一些对象实行保护的机制。

7.3.2 内存储器的保护

在多道程序设计环境中,保护内存储器是最重要的,这里所关注的不仅仅是安全性,还

包括处于活跃状态的各个进程的正确运行。如果一个进程能够不经意地写到另一个进程的存储空间，则后一个进程就可能会不正确地执行。

各个进程的存储空间的分离可以很容易地通过虚存的方法来实现，分段、分页，或两者的结合，提供了管理主存的一种有效的方法。如果进程完全隔离，则操作系统必须保证每个段或页只能由所属的进程来控制和存取，这可以简单地通过要求在页表或段表中没有相同表项来实现。

如果允许共享，则相同的段或页可能出现在不止一个表中。这种类型的共享在一个支持分段或支持分段与分页相结合的系统中最容易实现。在这种情况下，段结构对应用程序是可见的，并且应用程序可以说明某段是共享的或非共享的。在纯粹分页环境中，由于存储器结构对用户透明，要想区分两种类型的存储器就十分困难。

7.3.3 面向用户的访问控制

在数据处理系统中访问控制所采取的方法有两类：与用户有关的和与数据有关的。

在共享系统或服务器上，用户访问控制的最普遍的技术是用户登录，这需要一个用户标识符（ID）和一个口令。如果用户的 ID 是系统所知道的，并且如果用户知道与该 ID 相关的口令，那么，系统将允许该用户登录。这种 ID/口令系统是用户访问控制的一种很差的不可靠的用户控制方法。用户可能会忘记他们的口令，并且会无意或有意地泄漏其口令。黑客们在猜测特殊用户的 ID（如系统管理员的 ID）方面变得越来越老练。所以，ID/口令文件容易遭到攻击。

用户存取控制的问题在通信网络中更重要，因为登录会谈必须通过通信媒体进行，所以窃听是一种潜在的威胁。必须采取有效的网络安全措施。

在分布式环境中用户的访问控制可以是集中式，也可以是分散式的。使用集中式方法时，网络提供登录服务，用来确定允许哪些用户使用网络以及用户可以与哪些人连接。分散式用户访问控制将网络当作一个透明的通信链路，正常的登录过程由目的主机来执行。当然，在网络上传送口令的安全性也是必须解决的。

在许多网络中，可能要使用两级访问控制。可能要为个人主机提供一个登录工具以保护特定主机的资源和应用程序。另外，网络在总体上可以提供保护，限制授权用户的网络访问。这种两级工具是当前的普通情况所需要的，即网络连接了完全不同的主机，并仅仅提供了一种终端主机访问的便利方式。在更统一的主机组成的网络中，某些中心式访问策略可以由一个网络控制中心实施。

7.3.4 面向数据的访问控制

在成功登录以后，用户有权访问一台或一组计算机及应用信息，对于数据库中存有机密

数据的系统来说,这一般还是不够的。通过用户访问控制过程,系统要对用户进行验证。有一个权限表与每个用户相关,指明用户被许可的合法操作和文件访问,操作系统就能够基于用户权限表进行访问控制。然而,数据库管理系统必须控制对特定的记录甚至记录的某些部分的存取。例如,允许管理员查询公司职员名单,但是只有特定的人员才能访问工资信息,这个问题需要更多级细节层次。尽管操作系统可能授权给一个用户存取文件或使用一个应用,在这之后不再有进一步的安全性检查。但数据库管理系统必须针对每个独立访问企图做出决定,该决定将不仅取决于用户身份,而且取决于被访问的特定部分的数据,甚至取决于已公开给用户的信息。

文件或数据库管理系统采用的访问控制的一个通用模型是访问矩阵(access matrix),该模型的基本要素如下:

- 主体(subject)。能够访问对象的实体。一般地,主体概念等同于进程。任何用户或应用程序获取一个对象的访问实际上是通过一个代表该用户或应用程序的进程进行的。
- 客体(object)。被访问的客体(也称对象),如文件,文件的某部分、程序、设备以及存储器段。
- 访问权(access authority)。主体对客体访问的方式。如读、写、执行、删除等。

本章后面将详细介绍安全模型。

7.4 入侵者

7.4.1 入侵技术

两种最引人注目的安全威胁之一是入侵者(另一个是病毒),一般称为黑客(hacker)或计算机窃贼(cracker)。在关于入侵的一个重要研究中,Anderson 标识了三种类型的入侵者:

- 伪装者(masquerader)。一个未经授权使用计算机的个人,他穿过系统的访问控制,使用了一个合法用户的账号。
- 违法行为者(misfeasor)。一个合法的用户,他访问了数据、程序和资源,而这种访问对他来说是未经授权的或虽经授权却错误地使用了他的特权。
- 秘密的用户(clandestine user)。指夺取了对系统的管理控制,并使用这一控制权躲避审核和访问控制,或取消审核的人。

伪装者可能是外部人员;违法行为者一般来说是内部人员;而秘密的用户可能是外部人员也可能内部人员。入侵者攻击的范围包括从良性的到恶性的,有许多人仅仅希望浏览互联网看一看网上世界;还有些个人试图读取特权数据,执行对数据的非法篡改或破坏。

入侵者的目的是获得对系统的访问或提高他访问系统的特权范围。一般来说,这需要入侵者获得已被保护的信息。在多数情况下,这种信息的形式是用户口令,知道了其他用户的口令,入侵者可以登录到系统中,并行使该合法用户所具有的所有特权。

典型情况下,系统必须维护一个口令文件,该文件将记录每个授权用户的口令,如果该文件未经保护,则获得对它的访问并获取口令将很容易。一般的口令文件的保护可以采取下列两种方式之一:

- 单向加密。系统存储的仅仅是加密形式的用户口令。当用户输入口令时,系统对该口令加密并与存储的值相比较。实际上,系统通常执行一个单向(不可逆)的变换,使用口令生成一个用于加密功能的密钥,并产生一个固定长度的输出。
- 访问控制。对口令文件的访问被局限于一个或非常少的几个账号。

如果具有了上述一种或两种对策,入侵者要想得到口令还需要做其他工作,下面列出了获悉口令的技术:

- (1) 尝试标准描述所使用的缺省口令试猜,很多系统管理员都不会去修改默认设置。
- (2) 试遍所有一个到三个字符的短口令。
- (3) 尝试系统联机字典中的单词或可能的口令表中的词试猜。可能口令表在黑客公告板上可得到。
- (4) 收集用户信息,例如,他们的全名,配偶和子女的名字,业余爱好等。
- (5) 尝试用户的电话号码,身份证号和房间号。
- (6) 尝试该地区的所有合法牌照号码。
- (7) 使用特洛伊木马绕过对访问的限制。
- (8) 窃听远程用户和主机系统之间的线路。

前 6 种方法是各种猜测口令的方法,如果入侵者通过猜测口令来尝试登录,则这是一种单调乏味且容易失败的攻击方法。例如,系统可以简单地在三次口令尝试之后拒绝任何登录,入侵者必须重新连接到系统做新一轮尝试。在这些情况下,口令尝试法不切实际。然而,入侵者不会使用这种拙劣的方法,如果入侵者可以通过一个低级特权获得对加密的口令文件的访问,那么,就可捕获该文件,然后轻松地使用该系统特定的加密机制,直到找到更高特权的合法口令。

前面列出的第 7 种攻击方法是特洛伊木马,要防止它是非常困难的。一个低特权的用户提供了一个游戏程序,邀请系统操作员在闲暇时玩。该程序的确在做游戏,但在它的后台隐含着能够将未加密却具有访问保护的口令文件复制到该用户的文件中的代码。因为游戏是在操作员的高特权模式下运行的,所以,就能够访问口令文件。第 8 种攻击方法是线路窃听,这涉及到物理安全性问题,可用链路加密方法来防止。

下面讨论两种主要对策:预防与检测。预防是一个具有挑战性的安全目标,其困难在于

防卫者必须尝试防御所有可能的攻击,而攻击者却可以自由地去发现防御链中最薄弱的环节并在该点实施攻击;检测就是发现攻击,无论是攻击成功之前还是之后。

7.4.2 口令保护

系统的第一条防线是口令系统。事实上,所有多用户系统都要求用户不仅要提供名字或标识符(ID),而且要提供口令。口令的作用是认证登录到系统的个人 ID。ID 在下列情况下提供了安全性:

- ID 用来确定用户是否有权访问系统。在某些系统中,只有在系统上有存档 ID 的用户才能访问系统。
- ID 确定用户权限。只有很少的用户具有管理或超级用户状态,使他们能够读取文件和执行专门由操作系统保护的功能。有些系统具有客人或匿名账号,这些账号的用户比其他用户权限要低。
- ID 可用于自由决定的访问控制。例如,通过列出其他用户的 ID,用户可以允许他们读取该用户所拥有的文件。

1. 口令的脆弱性

为了理解攻击的本质,来讨论 UNIX 系统上广泛使用的一种方案,这种方法不显式存储口令,以免泄漏。其过程如图 7-3a 所示,每个用户选择一个最多 8 个可打印字符作为口令,该口令被转换成一个 56 位的值(使用 7 位的 ASCII 码),作为输入到加密例程的密钥。加密例程 crypt(3) 基于 DES(Data Encryption Standard) 算法使用一个 12 位的“salt”值修改,这个值与将口令分派给用户的时间有关。修改后的 DES 算法用补充 0 后的 64 位组成的数据输入执行。算法的输出接着作为下一次加密的输入,这个过程不断的重复共进行 25 次。最终的 64 位输出转化为 11 个字符的序列。这个密文口令与 salt 的一个明文副本一起被存储在口令文件中,用作相应的用户 ID。

这里的 salt 有三方面的作用:

- 防止口令文件中出现相同的口令。即使有两个用户选择了相同的口令,由于这些口令在不同的时间指定,因此,两个用户的“扩展”口令是不同的。
- 有效地增加了口令的长度,而不要求用户记住两个额外的字符,因此,可能的口令个数增加了 4096 个,从而,加大了猜测口令的难度。
- 防止使用 DES 的硬件实现,硬件实现会降低猜测口令的难度。

当用户试着登录到 UNIX 系统上时,他提供 ID 和口令。操作系统使用 ID 查询口令文件,并得到明文 salt 和口令密文,这些都作为加密例程的输入。如果所得结果与存储的值匹配,则用户登录成功。

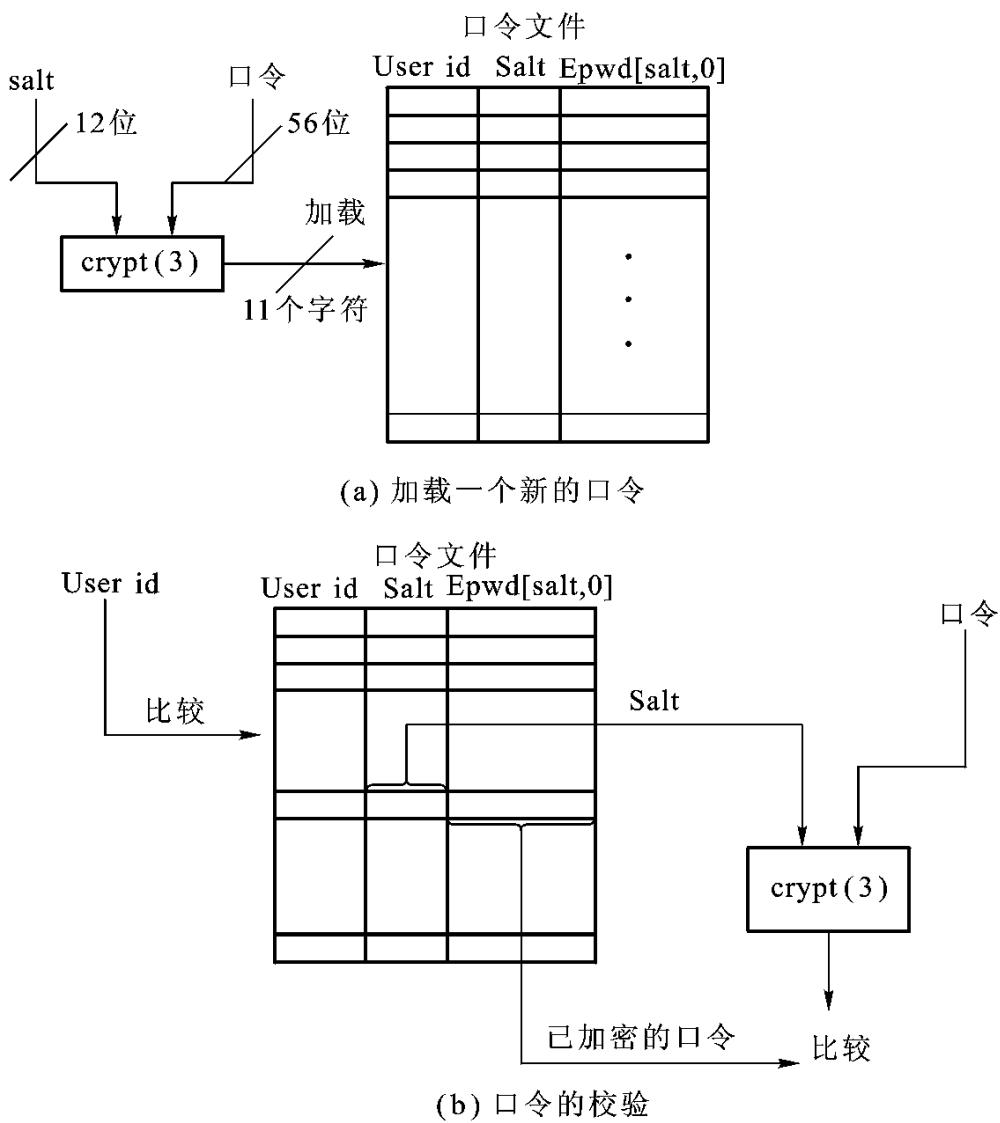


图 7-3 UNIX 口令方案

设计加密例程是用来防止猜口令, DES 的软件实现比硬件实现要慢, 25 次迭代, 使得软件实现需要 25 倍的时间。然而, 自从这个算法问世以来, 已经发生了两方面的变化: 第一, 算法的新实现导致算法执行速度加快。第二, 硬件性能的不断提高, 使得软件算法执行的速度更快。

这样, 对 UNIX 口令方案存在两种威胁。首先, 用户可以使用猜测的账号或通过其他手段获取对某台机器的访问, 然后, 在该机器上运行口令猜测程序。攻击者应该可以检测数百个甚至数千个可能的口令而消耗很少的资源。另外, 如果攻击者能够获取口令文件的一个副本, 则解密程序就可以很随意地在另一台机器上运行, 这样可以在很短的时间尝试数千个可能的口令。

尽管这样的猜测速率是巨大的, 然而, 攻击者通过使用这种笨拙的蛮力技术去尝试字符的所有组合来发现口令仍是不切实际的。事实上, 口令解密高手所依赖的实际情况是, 某些

人选择的是容易猜测的口令。有些用户在选择口令时,使用很短的口令。表 7-2 给出了普度(Purdue)大学的一项研究结果。该研究结果观察了 54 台机器上的口令并进行统计,其中大约有 7000 个用户账号,几乎有 3% 的口令是三个字符或更短。攻击者可以通过穷举所有可能的长度为三或更短的口令而发起进攻。系统的一个简单补救措施是拒绝任何少于,比如说 6 个字符的口令选择,或者要求所有的口令是 8 个字符,大多数用户将不会抱怨这一限制。

表 7-2 统计的口令长度

| 长度 | 数目 | 占总数的比率 |
|-------|-------|--------|
| 1 | 55 | 0.004 |
| 2 | 87 | 0.006 |
| 3 | 212 | 0.02 |
| 4 | 449 | 0.03 |
| 5 | 1260 | 0.09 |
| 6 | 3035 | 0.22 |
| 7 | 2917 | 0.21 |
| 8 | 5772 | 0.42 |
| total | 13787 | 1.0 |

口令长度只是问题的一部分,很多人在选择口令时,使用的是非常容易猜到的口令,例如,他们的名字,所在街道的名称,常见的字典单词等等,这使口令窃取工作变得简单。解密高手仅仅需要对比列出的可能口令来测试口令文件,因为很多人使用易猜的口令,这种策略基本在所有的系统都会获得成功。

2. 口令文件访问控制

阻止口令攻击的一个方法是不让对手访问口令文件,如果口令文件中的口令密文只能被特权用户访问,那么,攻击者不知道特权用户的口令就不能读取口令文件。Spafford. E 指出这种策略中的许多缺点:

- 很多系统,包括大多数的 UNIX 系统易受到突然侵入的影响,一旦某个攻击者通过某种方式获取了访问权,他就可能希望得到大量的口令,以便在不同的登录时使用不同的用户账号,减少被检测到的危险性。或者具有某账号的用户可能希望得到另一个用户的账号去访问特权数据,甚至破坏系统。
- 保护的失败致使口令文件可读,这样会危及所有的账号安全。
- 有些用户具有在其他保护域的其他机器上的账号,并且他们使用相同的口令。这样,如果口令被一台机器上的攻击者获取到,另一台机器可能也会受到威胁。

总的来看,更加有效的策略是强迫用户选择那些不易被别人猜出的口令。

3. 口令选择策略

从上述例子得到的教训是:用户选择的口令不能太短或太容易被别人猜出。在另一个极端,如果用户随意选取 8 个随机字符组成口令,那么,口令解密是很难的,但是对于多数用户来说记住这样的口令也很难。我们的目标是排除那些容易猜测的口令,而同时又选择容易记忆的口令,使得攻击又不容易得逞,可使用 4 种基本方法:

- 用户教育。告诉用户使用难以猜测的口令的重要性,并给用户提供良好选择口令的指导原则。这种用户教育策略在很多计算中心是不会成功的,尤其是当具有大量用户数目或大量人员更新情况时,很多用户会忽略这些建议。

- 计算机生成的口令。这也有问题,如果口令是随机产生的,那么,用户很难记住,于是用户往往把口令写下来。一般来说,计算机生成口令的方法很少为用户接受。

- 口令生效后检测。是指系统周期性的运行它自己的口令解密程序来发现容易猜测的口令,系统取消任何猜出的口令,并告知用户。这种方法有很多缺点,首先,会耗用资源。另外,任何现有有缺陷的口令直到被检出之前都保留在系统中。

- 口令生效前检测。改善口令安全性的最有希望的方法是口令生效前检测,这种方法允许用户选择自己的口令。这种检测技巧是在用户可接受性和口令强度之间达到一种平衡。如果系统拒绝的口令太多,用户将会抱怨选择口令太难。如果系统使用一些简单的算法来定义如何才可接受口令,那么,又降低了猜测口令的难度。下面是两种口令生效前检测方法:一种是所有口令最少为 8 个字符,且其中必须包含大写、小写、数字和标点;另一种是构造一个“坏”口令字典,系统检测用户的口令是否含在“坏”口令字典中。

7.4.3 入侵检测

毫无疑问,最好的入侵预防系统也会失败,系统的第二道防线就是入侵检测,这已经成为近年来研究的重点。主要因为:

- 如果一个入侵很快被检测到,则可以确定入侵者,并在他进行破坏之前将他逐出系统。即使不能及时检测出入侵者,但越早检出,破坏性就越小,系统恢复也越快。
- 有效的入侵检测系统具有威慑力,因而,也可以防止入侵。
- 入侵检测能收集入侵攻击技巧的信息,这些信息可用来加强入侵防范措施。

入侵检测基于的假设是入侵者的行为不同于合法用户的某些行为,当然,无法期望入侵者的攻击和授权用户正常使用资源之间存在着明显区别,相反两者之间存在着相同之处。

尽管入侵者的典型行为和授权用户的典型行为不同,但是也有相同之处。因此,对入侵者行为不精确的解释虽然会捕获更多的入侵者,但也会导致判断错误,将授权用户当作入侵

者。另一方面,对入侵者攻击行为的严格解释来限制错误的肯定判断,又会招致增加错误的否定判断,没有把入侵者检验出来。所以,在入侵检测的实际情况中要在这两方面做出权衡。下面是由 Porras, P.Stat 确定的入侵检测方法:

- 统计异常检测。收集一段时间内与合法用户的行为有关的数据,然后,对观察到的行为进行统计试验,来确定该行为是否是合法的行为。经常采用的方法有:(1)临界值检测。为不同用户的各种事件出现频率定义临界值;(2)基于直方图检测。构造用户行为的直方图,用于检测个人账号的行为变化情况。
- 基于规则的检测。定义一组规则集,用于决定一个已知行为是否是入侵者行为。经常采用的方法有:(1)异常检测。开发规则来检测与以前的使用模式的偏差;(2)攻击验证。使用专家系统的方法来搜寻可疑行为。

统计异常检测对伪装者是有效的,因为他们不太可能模仿所盗用账号的行为模式。另一方面,这种方法对于那些试图访问更高权限资源的合法用户来说就不太合适,对于这种攻击,基于规则的方法能够识别攻击事件序列。实际上,系统可以组合两种方法,以便更有效地对付范围广泛的攻击

入侵检测的基本工具是审计记录,用户必须保存一些正在进行的行为的记录,把它们作为入侵检测的输入,基本上使用两种方法:

- 本地审计记录。用来收集用户行为的信息,由于所有的多用户操作系统都有收集用户行为的软件,好处是不需要另外的收集软件了。但缺点是本地审计记录可能未包含所需要的信息或者不是所需形式的信息。

• 检测专用的审计记录。可以用一种收集工具来生成只包含入侵检测需要的审计记录,好处在于可独立运行,并可对大量系统检测,其不足之处在于需要额外开销。它的一个例子是由 Dorothy Denning 开发的工具,每个审计记录包含主体、行为、客体、例外条件、资源使用、时间戳。考虑由 Smith 发出的一条命令,该命令把可执行文件 GAME 从当前目录复制到 < LIBRARY > 目录中:

COPY GAME.EXE TO < LIBRARY > GAME.EXE

可能会产生下列审计记录:

| 主体 | 行为 | 客体 | 例外条件 | 资源使用 | 时间戳 |
|-------|----|----------------------|--------------|-------------|-------------|
| Smith | 执行 | < library > copy.exe | 0 | CPU = 00002 | 11058721678 |
| Smith | 读 | < smith > GAME.exe | 0 | RECORDS = 0 | 11058721679 |
| Smith | 执行 | < library > copy.exe | write - viol | RECORDS = 0 | 11058721680 |

在这种情况下,复制会异常中止,因为,Smith 没有向 < LIBRARY > 目录中写的许可。由于大多数用户行为由许多基本操作组成,如读文件、写文件、拷贝文件等。按行为来分解用户的操作有以下好处:由于客体受系统保护,使用基本行为可得到影响一个客体的所有行为的集合。因而,系统能进行存取控制,并能检测出主体能访问的客体集的异常;审计记录结构简单、统一,简化了模型和实现。

7.5 病毒(恶意软件)

7.5.1 病毒及其威胁

对计算机系统最一般的威胁是由那些能暴露计算机系统弱点的程序,这里主要讨论应用程序以及实用程序,例如,编辑器和编译器。病毒又称恶意的软件或 malware,病毒是专门用来制造破坏或用尽目标计算机资源的软件,它常常隐藏在合法软件中,或伪造成合法软件,在某些情况下,它通过电子邮件或感染的软磁盘将自己传播到其他的计算机中。

图 7-4 给出了恶意软件的总体分类。这些威胁可分为两类:需要主机程序的和独立运行的。前者在本质上是不能独立于某些应用程序,实用程序或系统程序而存在的程序段,后者是自含式的程序,可以通过操作系统来调度和运行。

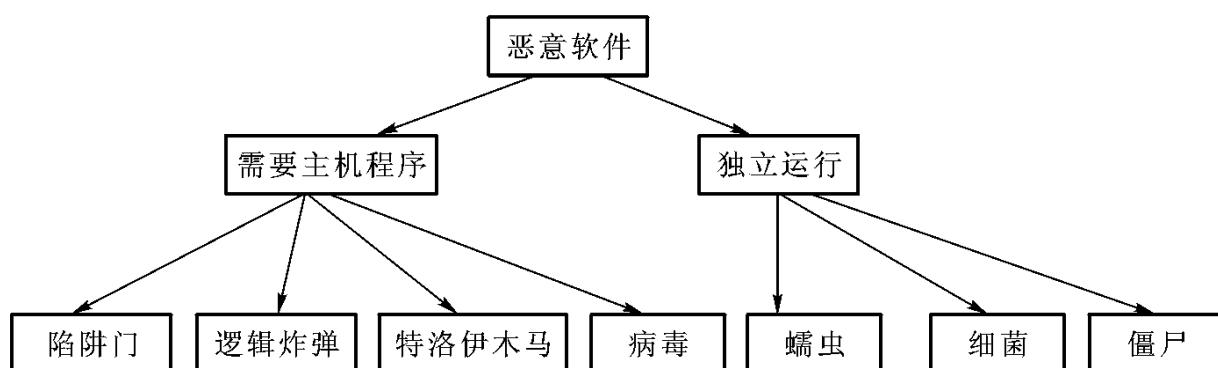


图 7-4 恶意软件分类

(1) 陷阱门(trap door)。陷阱门是秘密进入程序的入口点,用于在没有正常的访问授权的情况下获得了存取权。陷阱门被程序员正当用来调试和测试程序已经有很多年了。当程序员在开发应用程序时,该程序带有一个授权过程,或者具有一个很长的配置,需要用户输入很多值才能运行程序。为了测试程序,开发者希望获得特殊的权限以避免所有必需的配置和认证。程序员可能也希望保证具有一种一旦在授权过程中出了错误就能够激活程序的方法。陷阱门被不道德的程序员用于非授权的访问时,就构成了威胁。由操作系统实现对

陷阱门的控制是很困难的,安全方法应注重程序设计和软件更新上。

(2) 逻辑炸弹(logic bomb)。在病毒和蠕虫出现之前,程序威胁的最古老形式是逻辑炸弹。逻辑炸弹是嵌入到某些合法程序中的代码,满足某项条件时就“爆炸”。引发逻辑炸弹的条件有:某些特定文件、特定的一天,运行应用程序的某个特定用户。有一个著名的例子是,逻辑炸弹检查某个雇员的 ID 号(逻辑炸弹的作者),如果该 ID 没能在两个连续的工薪单中出现就引发。逻辑炸弹将更改或删除数据或整个文件,导致机器异常中止或进行其他破坏活动。

(3) 特洛伊木马(trojan horses)。它是嵌入到一个有用的程序或命令过程中的秘密、潜在的例程,程序的执行会触发该例程的执行,产生非预期的有害后果。特洛伊木马程序可用于间接地完成那些未授权用户无法直接完成的功能。例如,为了获取对共享系统上另一个用户的文件的访问,用户可以创建一个特洛伊木马程序,当它运行时,改变对这个用户文件的激发条件,使文件能被所有用户读取。程序作者把其放在一个公共目录下,并将程序的名字起的像一个有用的程序名字,来引诱其他用户运行这个程序。

(4) 病毒(viruses)。病毒是能够通过修改并感染其他程序的程序,修改的结果包括病毒程序的一个副本,继而可以感染其他的程序。计算机病毒可复制自身,主机上的病毒可暂时控制计算机磁盘操作系统,只要被感染的计算机使用未感染的软件,病毒的一个拷贝就复制进该程序,因此,病毒可通过用户间交换磁盘、在网内传送文件、网络中访问其他计算机上的资源及系统服务等进行传播。

(5) 蠕虫(worms)。网络蠕虫使用网络连接在系统之间进行传播。一旦在一个系统内部激活,像计算机病毒一样进行传播,也能植入特洛伊木马程序,进行破坏性的活动。为了复制自身,网络蠕虫需要使用某种网络工具,包括,电子邮件、远程执行、远程登陆等功能。于是蠕虫的一个新拷贝在远程计算机上运行,再以相同方式向外传播。然而,无论是网络安全方法还是单系统安全方法,只要设计和实现的恰当,都会使蠕虫的威胁达到最小。

(6) 细菌(bacteria)。细菌并不显式破坏文件,其目的是复制自身。一个细菌程序执行同时生成两个拷贝或生成两个新文件,每个都是细菌程序的一个副本。接着这两个副本每个再生成自身两次,如此下去。细菌指数级地再生,逐渐占用所有处理机时间、内存空间或磁盘空间,从而,使用户无资源可用。

(7) 僵尸(corpse)。是一个程序,它能秘密地取代另一台通过 Internet 连接的计算机,使用该计算机发起攻击,使得难以上溯到僵尸的创建者。僵尸多用于拒绝服务式攻击,一般都是针对目标 Web 站点。僵尸被种植到可信的第三方的数百台计算机上,然后,用于控制目标 Web 站点,对 Internet 通信发起无法抵挡的冲击。

7.5.2 病毒的特性

病毒可以做其他程序可做的任何事情,仅有的区别是它将自身复制到另一个程序中,当宿主程序运行时它也秘密的执行。一个病毒可以找到程序的第一条指令,把其变成跳转到内存某一区,然后,把病毒代码的一个副本送到这里,接着让病毒模拟那条被替换的指令,跳回主程序的第二条指令,继续执行主程序。一旦主程序运行时,病毒就感染其他程序,然后,再执行主程序。典型的病毒生命周期分如下4个阶段:

- 潜伏阶段。这时病毒是空闲的,病毒将被某些事件激活,如某个日期,另一个程序或文件的出现,或磁盘的容量超过某个限度。并非所有病毒都具有这个阶段。
- 传播阶段。病毒将自身的一份相同的副本复制到其他程序中,或磁盘上特定系统所在区域中。现在每个受感染的程序已经包含了该病毒的一个副本,它也同样进行传播。
- 触发阶段。病毒被激活,执行它想要执行的功能。与潜伏阶段一样,触发阶段可以由很多不同的系统事件所导致。
- 执行阶段。病毒已经得到执行。功能可能是无害的,如屏幕上显示一条信息;也可能是破坏性的,如删去文件或篡改数据。

大多数病毒都是以一种针对某一特定操作系统的方式运行,因而,病毒是针对特定系统的不足来设计的。

7.5.3 病毒的类型

自从病毒第一次出现以来,在病毒程序编写者和反病毒软件制造者之间就持续不断地进行着对抗。随着适用已有病毒类型的有效的反病毒措施的开发,新的类型的病毒也在不断得到发展。最为重要的病毒类型可以分为以下几类:

- 寄生病毒。传统的并且仍是最普通的病毒形式。寄生病毒将自己加到可执行文件中,当受到感染的程序执行时,它就将自身复制到其他可执行文件中。
- 常驻内存病毒。寄宿在主存中,并作为常驻系统程序的一部分,病毒将感染执行的所有程序。
- 引导扇区病毒。感染主引导记录或引导记录,并且当系统从含有病毒的磁盘引导时,进行传播。
- 秘密病毒。被设计得能够隐蔽自己的一种病毒形式,避免被反病毒软件检测到。
- 多形病毒。每次感染时变异,它使通过病毒的特征来检测病毒变得十分困难。
- 宏病毒。宏病毒充分利用了word和其他办公应用程序,例如Microsoft Excel的特点,即:宏。本质上,宏是一个嵌在字处理文档或其他类型的文件中的可执行程序。自动执行的

宏使生成宏病毒成为可能。这是一种自动调用的宏,无需用户显式输入。一般自动执行的事件有:打开文件、关闭文件、以及启动应用程序。

7.5.4 反病毒的方法

解决病毒威胁的最理想的办法是预防,不要让病毒侵入到系统中。尽管预防能够减少病毒成功攻击的数目,但这个目标一般来说是不太可能达到的。另一个较好的方法就是能够做到如下几点:

- 检测。一旦发生感染,就要确定它的发生并定位病毒。
- 识别。当检测取得成功后,就要识别并清除感染程序中的特定病毒。
- 清除。一旦识别出特定病毒,根除病毒并恢复程序原来的状态。

如果检测成功但没能清除,就可以把被感染程序删去,重新安装一个无毒版本。早期的病毒代码很简单,用通常的杀毒软件就可验证和清除。随着病毒的发展,杀毒软件也越来越复杂,杀毒软件经历了四代:第一代简单扫描。用病毒签名来验证病毒,病毒所有副本的结构和形式相同,这种扫描签名仅限于检测已知病毒;也可以记录程序长度,观察长度是否变化来确定有否病毒。第二代启发式扫描。扫描并不仅依靠签名,而采用启发式规则来搜索可能的病毒感染,例如,找出多形病毒的加密循环,从而找到密钥,依据密钥就可解密病毒,从而清除它并恢复程序。另一种二代方法是完整性检查,对每个程序附加一个检查和,如果病毒感染程序不改变检查和,则完整性检查就可以指出这种改变;对于改变检查和的病毒,可通过加密来检测,加密密钥独立于程序存放,于是病毒不能产生新代码加密。第三代杀毒程序常驻内存,它通过行为而不是被感染程序的结构来验证病毒。优点在于不必对病毒组提供签名,只需验证出感染的行为集。第四代是包含大量杀毒技术的工具包,这些技术包括扫描和活动验证技术。这种包还有存取控制能力,从而,限制了病毒渗透系统的能力,限制了病毒为传递感染的修改文件的能力。

下面讨论一些反病毒技术:

1. 通用解密

通用解密 GD(General Decrypt)技术使得即使对于最复杂的多态病毒,反病毒程序也能够很容易地检测,并且保持很高的扫描速度。当一个包含多态病毒的文件正在运行时,病毒必须对自己进行解密,从而激活自己。为了检测到这类结构,可执行文件运行需要通过一个GD 扫描器。GD 扫描器包含以下部分:

- CPU 仿真器。一个基于软件的虚拟计算机。可执行文件中的指令由这个仿真器解释,而不是在底层的处理器上执行。仿真器包括所有寄存器和其他处理器硬件的软件实现,因此,当程序在仿真器中解释时,不会对底层的处理器产生影响。

- 病毒签名扫描器。可以扫描目标代码,查找已知病毒的模块。
- 仿真控制模块。控制目标代码的执行。

在每次扫描开始时,仿真器开始解释目标代码中的指令,一次解释一条指令。因此,如果代码中包含了用于解密并暴露病毒的解密例程,则该代码被解释。实际上,病毒程序在暴露病毒时所进行的操作是与反病毒程序相同的,控制模块周期地中断解释,并扫描目标代码查找病毒标记。在解释过程中,因为目标代码是在一个完全受控制的环境中进行解释的,它对实际的个人计算机不会产生任何破坏。

设计一个 GD 扫描器最困难的是确定多长时间运行一次解释。典型地,当一个程序运行后,病毒元素很快被激活,但也不总是这样,扫描器模拟一个特定的程序的时间越长,就越容易捕获隐藏的病毒。但反病毒程序必须只占有有限的时间量和资源,否则用户会抱怨不已。

2. 数字免疫系统

数字免疫系统是 IBM 研制的一种综合的反病毒方法。研制该方法的动机是源于基于 Internet 传播的病毒的威胁越来越大。当前,病毒威胁的主要特点是新病毒和新病毒的变种以比较慢的速度传播,反病毒软件通常每月更新一次,并且这足以控制这方面的问题。同时,Internet 在病毒的传播中起着比较小的作用。

为了适应基于 Internet 的能力所带来的威胁,IBM 开发了一个数字免疫系统原型。该系统使应用程序扩展了程序仿真的能力,并提供一个通用的仿真和病毒检测系统。该系统的目标是提供快速响应时间,使得几乎可以在产生病毒的同时就消灭它。当新病毒进入到一个组织中,免疫系统自动捕获到该病毒并进行分析,为它增加保护和防护,除去它并把这个病毒的信息传送到正在运行 IBM 反病毒软件系统,因而,可以使得该病毒在其他地方运行之前就被检测到。图 7-5 给出了数字免疫系统中典型的操作步骤。

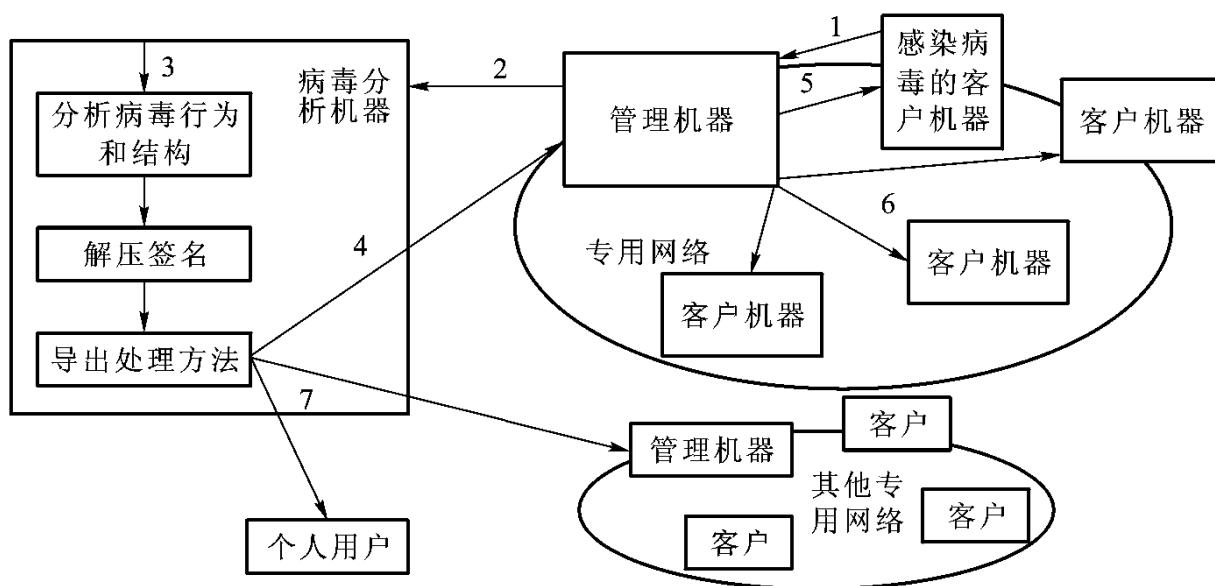


图 7-5 数字免疫系统

7.5.5 电子邮件病毒

恶意软件的最新发展是电子邮件病毒。第一个快速传播的电子邮件病毒,如 Melissa,使用了嵌入在附件中的 Microsoft Word 宏。如果接受者打开电子邮件附件,这个 Word 宏被激活。那么:

1. 电子邮件病毒把它自身发送给用户的电子邮件包中的邮件列表里的每个人。
2. 病毒进行局部破坏。

1999 年末出现了一种强大的电子邮件病毒版本,这个新版本可以仅仅通过打开一个包含该病毒的电子邮件而被激活,而不是通过打开附件。该病毒使用的是电子邮件软件包支持的 Visual Basic 脚本语言。

因此,可以看出,新一代的恶意软件通过电子邮件到达,或者使用电子邮件软件特征在 Internet 上复制自己。病毒只要被激活(或者通过打开电子邮件附件,或者通过打开电子邮件)。就开始把自身传播到被感染的主机所知道的所有电子邮件地址中去。其结果是,以前病毒传播需要几个月甚至几年的时间,现在只需要几小时,下表描述了这一传播时间。这使得反病毒软件很难在产生大规模破坏之前就对该病毒做出反应。最终,对于这种不断增长的威胁,必须为 PC 机中的 Internet 实用程序和应用软件建立更高程度的安全级。

表 7-3 病毒传播时间

| 病毒 | 开始时间 | 类型 | 普遍传播所需时间 | 估计造成的损失 |
|---------------------------|------|------------------------------|----------|-------------------------|
| Jerusalem Cascade Form | 1990 | .exe 文件 | 三年 | 所有毒素在五年多的时间内为 5 000 万美元 |
| Concept | 1995 | Word macro | 四个月 | 5 000 万美元 |
| Melissa | 1999 | E-mail enabled Word macro | 四天 | 3.85 亿美元 |
| Love letter | 2000 | E-mail enabled VBS based | 五个小时 | 高达 150 亿美元 |

7.6 保护的基本机制、策略与模型

7.6.1 机制、策略与模型

现代计算机都把文件(信息)存储在共享设备上,这就产生了信息保护的问题。用户所

拥有的信息,有时候允许其他用户共享,但有的时候,又希望信息是私有的。操作系统如何来建立一个环境,使得用户可以选择让信息私有的,或与他用户共享呢?这就是操作系统的保护和安全性功能。当计算机联网工作后,实现信息保护更加困难。在下面的几节中,主要讨论操作系统中保护文件和其他信息的一些技术。

一个组织的安全策略定义了一组用于授权使用其计算机及信息资源的规则。例如,某组织只允许财务部门的职员使用存储了财务信息的计算机。那么,如何在现代计算机系统中定义和执行安全策略呢?计算机保护机制是实施安全策略的工具,同型号的计算机(使用相同的操作系统)可能采取不同的安全策略,即使他们使用了相同的保护机制。那么,怎样的保护机制系统可以用来支持策略呢?什么才是解决该问题的合理的方法,从而使问题可以在计算机的软硬件范围内被解决?这些都是关系到计算机系统保护和安全问题的例子。

在操作系统中一直强调把策略(policy)与机制(mechanism)明确的分开。策略规定要达到的特定目标,如安全范围内将决定什么时候去完成什么样的信息保护任务?谁的数据需要保护,以及禁止谁访问等;机制是完成任务和特定目标的方法、即如何实现保护,是一组实现不同种类保护方案的算法和代码的集合。这样做的优点是留有灵活性,策略发生变化时整个系统变化小。例如,除了交换信息外,某种特殊的通信策略不允许两个进程共享资源。支持此策略的通信机制就需要支持消息传送,使用的方法可能是把一个进程地址空间内的信息复制到另一进程。系统的安全策略制订了对本组织人员和非本组织人员资源的共享方式。机制是系统提供用于强制执行这安全策略的特定步骤和工具。

举个例子,某学院的计算机系可能有这样一条策略:本科生实验室中的计算机只能给已注册的计算机班的本科生使用。支持此策略的机制需要用户的学生证和计算机系的班级列表。此机制还必须由其他诸如授权认证、实验室的设备来补充完成。建立精确的策略是很困难的,因为,它既需要制订准确的软件需求,又需要制订无任何漏洞的“法令”以控制系统用户的活动。

根据经验,当机制能确保按照它定义的方式工作时,它将被在操作系统中使用。如果机制不能确保按定义的方式工作,那将不能依赖它来实现策略。在现代操作系统中,安全的保护机制可能只在操作系统内实现。一般在操作系统和其保护机制设计及实现后,可由计算机设计者或管理员来选定某些策略。操作系统的一部分用于实现机制,而其他部分——系统软件、应用软件决定了策略。保护机制实现了身份认证功能,使得策略可以验证作为某个实体的用户或远程计算机是否确实是其宣称的那样。认证机制还被用于检查某实体是否拥有访问某些资源的权限。

安全策略是对系统的安全需求,以及如何设计和实现安全控制有一个清晰的、全面的理解和描述。安全模型的目的就是精确的描述系统的安全需求和策略。安全模型有下列特点:

- 它是精确的,无歧义的。

- 它是简单和抽象的，容易理解的。
- 它是一般的，只涉及到安全性质，不过度地抑制系统的功能或其实现。
- 安全模型是安全策略的明显表现。

1. 安全模型的规则

一个系统不如所希望的那样安全有两个原因：一是在安全控制中有漏洞，二是对安全定义有缺陷。第一个问题是软件可靠性问题，它可以通过与设计技术相关的软件工程手段来克服。第二个问题，就是定义系统做什么的问题。对大多数系统来讲，这并不是特别困难，但相对来讲，与安全有关的定义是比较困难的，因为，它必须非常精确。在图 7-6 中描述的规范系统开发过程中，安全模型起着非常重要的，关键性的作用。

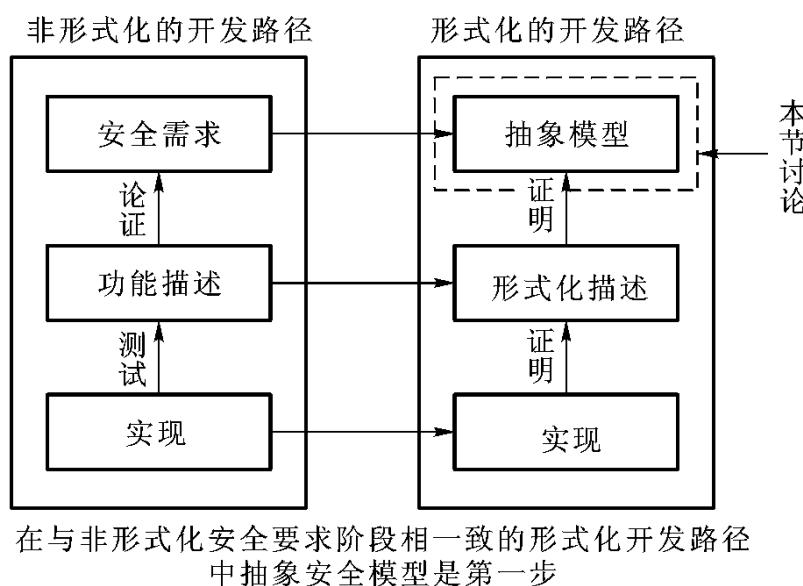


图 7-6 系统开发路径

图 7-7 给出了四种进行规范化系统开发的途径，这些途径用来说明相应模型的实现过程。图 7-7 中(a)与(b)在模型定义前并不涉及任何形式化工作。(a)是假设已经开发了一个形式的或非形式的模型，但是没有实现该系统所依据的有关安全方面的性质。这并不是说根本没有任何描述，只是说明虽有功能方面的描述，但这种描述对模型中所表达的安全需要并没有特别详尽的说明。在(b)中，关于安全性质的非形式描述是模型与实现的中间步骤。这两者都需要非形式论证以及支持相应论证的测试试验。

由于模型与实现之间的细节在层次上有很大的跳跃，所以(a)中相应的论证非常脆弱，这种模型比较含糊。在(b)中，允许开发者进行相应的比较可信的论证，特别是能够使用非形式描述作为设计与实现系统的基准，正如使用功能描述一样。

图 7-7 中(c)与(d)利用了形式化的描述与证明，因此，这两者都需要一个形式化模型。(c)在(b)中相当于非形式描述的地方使用了一个形式描述。虽然(c)中的形式化描述与(b)

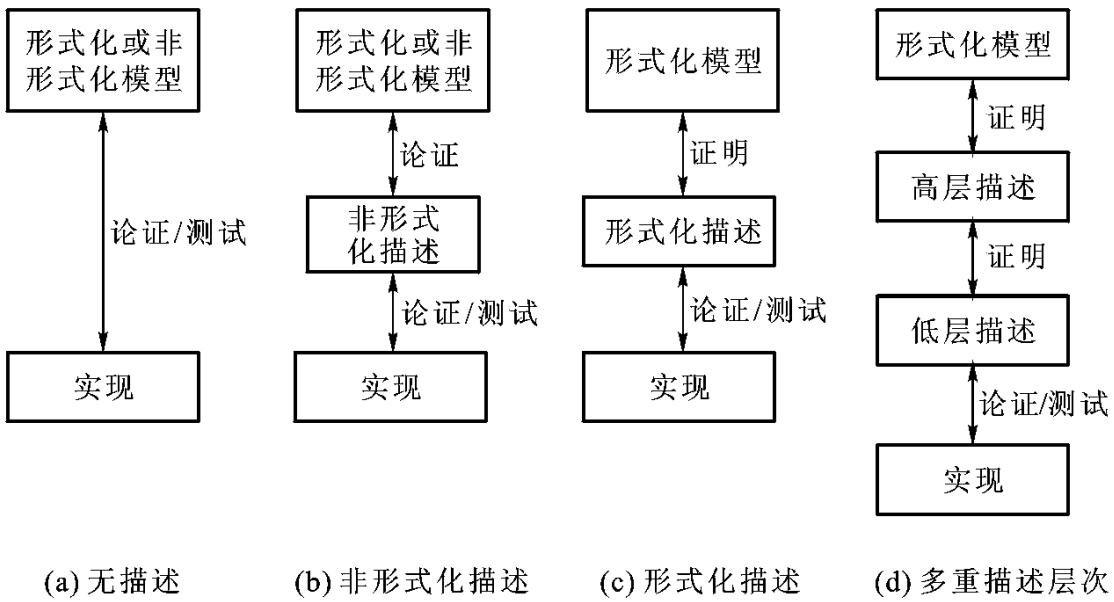


图 7-7 模型的一致性选择

中的非形式化描述具有相同的层次细节,但(c)中形式描述更加精确并无二义性。描述的形式行为为模型相应的数学证明提供了基础,而相应于实现与形式描述间的论证则是非形式的。(c)比(b)明显地增加了保险性,不过,在抽象的形式描述与实现之间仍有较大的间隙。(d)在模型与实现之间有两层甚至多层形式化描述,因此,它缩小了必须有非形式描述补充的间隙。以目前验证技术的发展水平,还不能完全消除在实现层与最低描述层之间的非形式论证过程,而这个非形式步骤在整个从模型到实现之间的论证环节是最薄弱的。

另一方面,由于太多的中间证明会增加错误频率,中间层次的增多会相应地降低安全程度,那么,增加描述层所获得的益处就会迅速下降,甚至最优良的自动验证系统也不能消除所有的人为错误。同时,描述层的增多也明显增加了中间描述与证明的成本。所以,并不是中间描述层越多越好。

2. 安全模型的实际应用

由于建立安全模型的抽象性,人们往往避开这项工作,也很难正确地评价模型与实际系统间的对应关系,特别是当建模工作无助于系统设计时,人们往往就势摆脱建模工作及相应的数学形式。下面将说明安全模型在实际系统开发过程中的作用,主要是要说明它对系统的安全需求的描述是非常有价值的。

当对一个系统的功能进行描述时,通常不能对描述系统的每个可能的行为做足够详细的设计。事实上,即使考虑了所有的异常错误条件,并对系统功能作了非常详尽的描述,也不可能强制实施这些设计。一般情况下,系统的功能描述都允许系统设计者自由地定义在意外情况下应采取的措施,这种意外情况将随着系统的设计而不断出现。对于描述需要满

足的系统的特殊需求方面,功能描述是最有用的。当实现该系统时,设计者还需要进行许多调整,处理意外情况的功能不能违背描述过的精神。

对于安全系统的描述,不应留下某些可供选择的机会。只要功能描述中留下一个漏洞,那么,在系统的任何地方都可能出现安全问题。对系统安全的功能描述不需要非常详细地包罗每一项系统功能,但是,无论省略哪个细节,安全功能描述都必须保证系统不违反安全策略的要求。

安全模型可以帮助人们尽可能精确地描述系统功能,以堵住所有的安全漏洞。作为功能描述的辅助部分,安全模型必须在不限制系统功能的条件下,满足安全需求,必须证明或者说明安全模型能够遵守根据安全模型得到的安全性质。所以,基于这种安全模型而实现的系统将不会有安全方面的漏洞。系统的功能描述指导系统功能的实现,而安全模型指导与安全有关的系统功能的实现。

数学模型或某种建模技术并不是描述系统安全性质的惟一途径。建模既要具备一定的条件,又需付出很大的努力才能完成。如果系统已经建成了,而只是要增强它的安全性,那么,只要有一个非形式化的,自然语言描述的模型就足以满足要求。如果仅仅能对现存系统做少量的修改,那就不能指望这种修改对系统安全性能有很大的改善。这种情况下对系统建立安全模型是不必要的。从概念上说,开发非形式化模型的过程与开发形式化数学模型的过程相同。

幸运的是,对于多数人来说,只是使用模型而不需要自己开发模型。对大多数情况来讲,只要对现存的几种模型稍做修改就能满足系统的要求。在下面介绍的 Bell&Lapadula 模型已经成功地运用在许多基于安全核的系统中。

3. 安全模型的类型和特点

由于安全模型彼此都有显著的不同之处,要想将它们分类不太容易。现在已有屈指可数的几种模型,但真正用于实际的却更少。

状态机模型是将系统描述成一个抽象的数学状态机。在这种模型中,状态变量表征机器的状态;转移函数或称操作规则描述了这些变量是怎样变化的。大部分模型是状态机模型。将系统抽象成一种状态机是相当古老的方法。但这种方法之所以没有在软件开发中起重要作用是因为描述操作系统的所有可能状态变量是不现实的,也是不可能的。而安全模型仅仅涉及与安全有关的状态变量,所以,可对状态机模型进行简化,访问矩阵模型是状态机模型的一种。它将系统的安全状态表示成一个大型矩阵阵列。系统中的每一个主体都拥有一行,每一个客体(对象)都拥有一列。客体可以是程序、内存段、文件、甚至数据库。矩阵中的交叉项表示某主体对某客体的访问模式、即存取权限集。矩阵的列也可以代表主体,这时矩阵中的交叉项表示了相应行主体与相应列主体之间的通信限制。模型的转移函数描述了访问矩阵是如何变化的,本章后面将对访问矩阵模型的实现作进一步讨论。

访问控制负责对实体本身的安全管理,但许多信息泄密不是因为访问控制自身的不完善,而是缺少对信息流动的保护。信息流控制模型 (information flow control) 是访问矩阵模型的一个补充,在一个完善的保护系统中两者缺一不可。它不校验主体对客体的访问模式,而是监管信息可以流通的有效路径,试图控制从一个实体流向另一个实体的信息流,这种控制是根据两个实体的安全属性强制实行的。信息流模型与访问模型之间只有很细微的差别。信息流模型最突出的实用价值是它能够帮助系统发现隐蔽通道,而访问控制却做不到这一点。D.E.Denning 的格阵模型是一种信息流控制模型,它可以用来描述信息流的通道与信息流动策略。

非相干模型。在该模型中,各个主体在不同的域中运行以防止它以某种违反系统安全性的方式干扰其他主体。这种模型还在不断地发展完善。Honeywell 公司正将它用在 Secure Ada Target 的研究项目中。

人们通常很难理解系统模型和系统描述或说明(例如形式化描述)之间的差别。当开发一个特殊的系统模型时,容易疏漏某些细节,结果使模型达不到预期的目的。但是一个好的模型基本特点是:它很容易理解,能用自然语言描述,模型的所有重要方面仅用几页纸或仅用几分钟就能表达清楚。然而,要使一个非数学工作者开发使用一个严密的数学形式的模型可能比较困难,但是,在表达安全模型时,应使熟悉数学符号的人能够很容易理解它。

由于安全模型是用数学语言对系统安全性质的复述,所以,模型必须简单明了。如果模型本身含糊不清,或者说,模型与所要求的安全策略之间没有明显的对应关系,就很难使人相信模型体现了安全策略的要求。由于目前还不能从数学上证明用自然语言描述的安全策略与抽象的安全模型之间的一一对应关系,所以做到“使人相信”就已经很好了。目前流行的几种安全模型的模型与策略之间的对应关系十分清晰,使人一目了然。

4. 状态机模型

由于状态机模型能够模仿操作系统与硬件系统的运行,所以,计算机系统开发人员都十分偏爱它。状态变量是对系统中的每个比特与字节的抽象表示,它们随着系统的运行而不断地变化。存储器、磁盘以及寄存器中的每一个字都是一个状态变量。状态转移函数是对系统调用的抽象表示,它精确地描述了状态的变化情况,虽然还有一些有前途的建模技术,但是状态机的概念已经渗透到了各种技术领域,每个做建模工作的人都能理解状态机模型。

安全模型并不涉及所有的状态变量与转移函数,所以,选择与安全有关的初始状态与转移函数是建模的上上策。开发一个状态机安全模型,需要从一个安全的初始状态起,描述模型的元素(变量,函数,规则等)。只要能够证明初始状态是安全,并且所有的转移函数也是安全的,那么,数学推理就能保证:只要系统从某个安全状态启动,无论按什么顺序调用系统功能,系统将总是保持在安全状态。

开发一个状态机模型一般需要下述步骤:

(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与客体,它们的安全属性,以及主体对客体的访问权。

(2) 定义安全状态需满足的条件。这是一个不变式。它表达了在状态转移期间,状态变量值之间必须保持的关系。

(3) 定义状态转移函数。它描述了状态变量可能发生的变化。有时也称其为操作规则,因为,它的目的是限制系统产生变化的形式,而不是描述所有可能的状态变化。这种规则可能是非常一般的,也可能允许设置一些系统不具备的功能。但是,系统不能修改状态变量,使其具有一些不允许出现的功能。

(4) 证明转移函数能够维持安全状态。为了确信模型与安全状态间的一致关系,必须证明:对每个转移函数,如果系统在某种操作之前处于安全状态,那么,完成相应操作后,系统仍处于安全状态。

(5) 定义初始状态。为每个状态变量选择一个初始值。它将描述系统从怎样的初始状态开始运行。

(6) 用安全状态的定义证明初始状态是安全的。

上述步骤可以看成一点点抽象的描述,它对那些习惯于编写计算机程序而不习惯于对程序进行数学描述的人是有帮助的。

先来看一个军事安全策略的例子:假定区分不同级别,士兵为一般密级、上尉为保密级、上校为机密级、将军为绝密级,他们的文件(对象)分别就为一般密级、保密级、机密级、绝密级。上校可以看上尉的文件,更可以看士兵的文件,但决不能看将军的文件。上尉可以告诉上校他所知道的军事情况,当然也可以直接报告给将军,但决不能把军事情况透露给士兵,因为普通士兵未被授权。当为数据定义了多个级别的安全要求时,称作多级安全性(multilevel security),对多级安全性的一种通用安全策略必须实施:

- 不向上读 进程不能读级别高于它本身的对象,但可以读同级或低级的对象,例如,机密级进程可以读保密级或一般密级的文件,但却不能读绝密级文件。

- 不向下写 进程不能向低于其本身级别的对象写信息,而仅能写入同级或更高级别的对象中,例如,保密进程可以向机密或绝密文件写信息,但却不能写一般密级的文件。

下面来说明安全策略是怎样演变成抽象的安全模型的。先讨论怎样证明模型能够满足安全状态的不变性,之后,考虑一些需要模型满足的附加限制。模型利用自然语言描述安全策略,并假定策略中没有其他限制。

安全策略

- 策略(a) 当且仅当用户的许可证级别高于或等于文件的密级时,才可以阅读该文件。
- 策略(b) 当且仅当用户的许可证级别低于或等于文件的密级时,才可以写该文件。

首先应对策略中的元素做一下处理,然后用抽象术语重新描述上述安全策略。为此,做

如下替换。

| 自然语言术语 | 计算机用术语 |
|--------|--------|
| 人 | 主体 |
| 文件 | 客体 |
| 许可证级别 | 存取类 |
| 密级 | 存取类 |

根据上述抽象替换, 安全策略变成如下形式:

- 性质(a) 仅当主体的存取类高于或等于文件的存取类时, 才可以阅读该文件。
- 性质(b) 仅当主体的存取类低于或等于文件的存取类时, 才可以写该文件。

将许可证级别与密级等同视为存取类, 因为, 两者具有相同的结构与意义。当自然语言描述的安全策略对不同类型的客体具有不同的访问规则时, 有些系统就使用比较特殊的术语。下面介绍开发一个安全模型的步骤。

第一步: 定义状态变量

根据对安全策略的抽象处理, 模型中需要的状态变量如下表所示, 其中还包括一些后面例子所需的状态变量, 其中符号 Φ 表示空集。系统中任意的主体与客体都包括在集合 S 与集合 O 中。二维访问阵列 A 惟一地表达了所有主体对客体的当前访问权。

表 7-4 状态变量

| |
|------------------------------------|
| S = 当前主体集合 |
| O = 当前客体集合 |
| sclass(a) = 主体 s 的存取类 |
| oclass(o) = 主体 o 的存取类 |
| A(s, o) = 访问模式集合, 取下列集合之一 |
| { } = 如果主体 s 能够读客体 o |
| {w} = 如果主体 s 能够写客体 o |
| {r, w} = 如果主体 s 既能够读客体 o, 又能够写客体 o |
| { } = 如果主体 s 既不能读客体 o, 又不能写客体 o |
| content(o) = 客体 o 的内容 |
| subj = 活动主体 |

另外, 还定义了两个在性质(a)与性质(b)中没有直接提到的状态变量: contents(o), 表示每个客体(信息的内容)状态; subj 代表当前活动并且正在调用转移函数的主体的标识符。可以将 subj 看作一个变量, 该变量等于主体当前集合中的某个主体标识符, 它可以在任何

时候变成一个任意值。由于这两个变量在安全性质中没有提及,姑且认为它们与安全无关。但是,在安全模型中包括它们是有一定道理的,在后面就会看到这一点。系统在任意时刻的状态定义为所有状态变量的集合:

$$\text{state} = \{S, O, \text{sclass}, \text{oclass}, A, \text{contents}, \text{subj}\}$$

第二步: 定义安全状态

安全状态的定义就是将性质(a)与性质(b)在数学上解释成一个不变式,见下表:

表 7-5 安全状态的定义

| 不变式 |
|--------------------------------------------------------------------|
| 系统是安全的,当且仅当对于所有 $a \in S, o \in O$, 有 |
| if $r \in A(s, o)$, then $\text{sclass}(s) \geq \text{oclass}(o)$ |
| if $w \in A(s, o)$, then $\text{oclass}(s) \geq \text{sclass}(o)$ |

虽然安全状态的定义是很直观的,但还不能证明我们对原始策略的解释以及对状态变量的定义是否准确地表达了原始安全策略,因此,模型中的安全性质必须简单明了,使人相信它与原始安全策略是一一对应的,这一点很重要。无论是安全性质或安全状态的不变式定义,事实上都没有说明主体是否可以对客体进行读、写操作。换句话说, A 的全部值可能都是空的,根据不变式的定义,系统还是处于安全状态中。不同的安全策略可能需要不同的访问模式,相应地需要系统具有不同的系统功能调用,但是,一般的安全策略对系统的实用性并不附加任何限制。

第三步: 定义转移函数

转移函数可以看成主体对系统服务子例程的过程调用,服务完成后,状态变量就会产生相应的变化。函数中的参数由主体来描述,在系统发生状态变化之前,系统必须对参数进行有效性检测。下表列出了后面将要讨论的状态转移函数。

表 7-6 状态转移函数

| | |
|-----------------------------|-------------------------|
| Creat _ object(o, c) | 在存取类中生成客体 o |
| Set _ access(s, o, modes) | 设置主体 s 对客体 o 的存取方式 |
| Creat/Change _ object(o, c) | 设置客体 o 的存取类别为 c 并建立客体 o |
| Write _ object(o, d) | 将数据写入 contents(o) |
| Copy _ object(from, to) | 复制 from 的内容到 to 的内容中 |
| Append _ data(o, d) | 将数据 d 加到客体 o 的内容里 |

这里用 ‘’号表示状态变量的新值,头两个状态转移函数可定义如下,见下表:

表 7-7 两个状态函数的定义

```

FUNCTION 1: Create _ object(o, c)
    If o ∈ O
        Then 'O = O ∪ {o}
            And
            'oclass(o) = c
            and
            for all s ∈ S, 'A(s, o) = Φ

FUNCTION 2: Set _ access(s, o, modes)
    If o ∈ O and s ∈ S
        And if {[r ∈ modes and sclass(s) ≥ oclass(o)] or r ∉ modes}
            And
            {[w ∈ modes and oclass(s) ≥ sclass(o)] or w ∉ modes}
        then 'A(s, o) = modes

```

这些函数的表达方式很像计算机程序,但它们与计算机程序有如下的重要的区别:

- 转移函数的目的是描述转移函数前后状态变量之间的关系,函数中的 $=$ 号可以看作是数学上的相等关系,虽然它可能表达对新的状态变量的赋值,但没有必要看作是赋值关系。
- 对一个操作来讲,函数并没有表明语句的特殊顺序,应该将它看作是对操作完成后状态变化情况的描述。
- 函数是不可分的,也就是说,它的作用是不可见与不可中断的,在任何时候都可能发生状态变化。在状态转移时,没有任何时间间隔。在对多处理器系统建模时,这种细微性假设非常重要,它提醒你注意,在多道程序系统中,要特别小心,不要忽略异步进程的影响。在形式化描述中比在模型中能更详细地说明这些问题,但基本概念是相同的。
- 最后一点是,函数是对所有许可的状态转移的描述,如果一个状态变量或某个阵列的元素在状态转移前后没有直接的变化,那么,相应的状态变量是不变的。在如下所示的表达式中:

if conditions then

省略了 else 语句,并且在 conditions 为假时,没有说明系统怎样动作,那么,对应状态变量来讲,就不发生任何变化。

Creat _ object 操作是将一个客体加到当前客体集合中去,使其成为 $\{ \}$ 中的一员,同时将该客体的存取类别设置为所要求的值 c。它还将访问控制矩阵中相应于该客体的列设置成全空,因此,没有一个主体能够访问到该客体。Set _ access 操作将对 A 的任意一个元素设置一个新的访问模式,只要这个访问模式与安全状态的要求一致即可,按惯例,对于 $A(a, x)$ 中

$x \neq o$ 的那些列, 使其维持原值。

第四步: 证明转移函数

通常, 只要能够确信构造的转移函数基本上是简单、正确的, 那么, 在进行严格的证明之前, 就可以把绝大部分转移函数构造出来。但是, 比较明智的做法是: 对头几个函数的正确性要进行证明, 它会使人们明确建模的方向是否正确, 以免走弯路。对每个转移函数来讲, 要保证它的安全性, 必须证明下述定理:

$$\text{不变式} + \text{函数} \rightarrow \text{不变式}$$

该定理指出, 当且仅当在某个安全状态(满足不定式)下调用该函数操作后, 系统仍能维持在安全状态(满足不变式)时, 这个转移函数才是安全的。

例如, 在转移函数 Create _ object 中, 如果对访问矩阵中相应于新生成客体的那一列不设置为全空, 即如果去掉 for all $s \in S$, $A(s, o) = \Phi$ 这一条, 那么, 该函数就可能不满足不变式的要求。因为, 假设对某主体 S_i 授予了某种访问模式 mode, 比如 $r = mode$; 那么, 不能保证 $sclass(S_i) \geq oclass(o)$, 由于在这个函数中没有这条保证, 所以, 就不能保证转移函数是安全的。虽然可以利用其他方法来保证在 Create _ object 中对新客体的访问模式不设为全空的, 仍能使得该函数是安全的, 例如, 对新客体设置某个非空访问模式时, 检查相应主体的 $sclass(s)$ 是否满足不变式的要求。但是, 对新客体的访问模式设为全空将会具有更大的灵活性, 也使相应的函数大为简化了。由于在进行 Create _ object 操作时, 可以调用 Set _ access 函数来对新客体设置任何所要求的访问模式, 这样, 在生成新客体时, 如果要求对该客体设置某种访问模式, 那么, 在调用 Create _ object 之后, 可以接着调用 Set _ access。

第五、六步: 定义并证明初始状态

初始状态是非常重要的一个环节, 如果对初始状态定义的不好或者初始状态的安全性得不到保证, 那么, 整个系统的安全性也不会有保障。从数学上讲, 初始状态就是系统中所有状态变量的初始值的一个集合。

$$\{S_0, O_0, Sclass_0, Oclass_0, Contents_0, Subj_0\}$$

为了证明初始状态是安全的, 必须对这些状态变量的初值设置一些限制。满足不变式要求的最简单的初始状态是一个没有任何主体和客体的状态集:

$$\text{初始状态 (1)} : S_0 = \Phi, O_0 = \Phi$$

很明显, 根据不变式的要求, 不必对其他状态变量值做任何说明, 这个初始状态肯定是安全的。当然, 在模型中, 必须增加一个生成主体的函数, 不然的话, 系统永远也不会达到 $S \neq \Phi$ 的状态。

另一个, 也是比较实际的一个初始状态是: 系统允许初始状态中有任意数目的主体和客体, 但它们必须有相同的存取类别:

$$\text{初始状态 (2)} : \text{For all } s \in S_0, o \in O_0$$

$$\begin{aligned} S_{\text{class}_0}(s) &= c_0 \\ O_{\text{class}_0}(o) &= c_0 \\ A_0(s, o) &= \{t, w\} \end{aligned}$$

初始访问矩阵 A_0 允许所有的初始主体对所有的初始客体进行读写访问。这种初始状态是非常一般的, 它对初始主体与初始客体的数目没有任何的限制, 只有一条, 要求所有的初始主体与初始客体具有相同的存取类别。

到目前为止, 构造状态机模型基础的关键是安全状态的概念, 在这里安全的定义完全体现在一个不变式中, 可以在任意时刻对系统状态取样, 并根据这个不变式来判断系统是否安全, 而不必考虑系统在前面的状态中是如何动作的。无疑, 仅仅靠一个不变式并不能够完全描述所要求的安全性质, 这是因为, 安全并不仅仅是系统当前状态的一个性质, 而是一系列相继状态的性质。因此, 需要对安全的定义做一下修改, 使它不仅能够体现单个状态中各个状态变量之间的关系, 还能够体现两个相继状态转移过程中各个状态变量之间的关系。

在这种修正模型中, 虽然每个转移函数可能都满足不变式的要求, 但是, 某个转移函数可能还是不安全的, 因为从原状态到新状态的某些转移是不允许的。为了描述转移的性质, 需要对模型中有关安全的定义附加一些约束。要证明这些约束的正确性, 对约束条件的处理必须像对不变式一样, 即必须保证每个函数都满足约束要求。约束条件与不变式的差别在于, 它仅仅涉及到两个相继状态(转移前与转移后)间的状态变量值之间的关系。对转移函数附加约束有几个原因:

- 不安全的转移。转移前后的状态变量值之间必须维持安全关系。
- 对主体的限制。在某些情况下, 不允许主体调用某些操作。
- 对信息的控制。对处理信息内容的模型来说, 必须控制修改信息内容的转移操作。

下面根据这三条原因来分别说明限制的方法:

(1) 不安全的转移。

现在重新写 Create _ object 如下, 允许改变已存客体的存取类别, 和前面一样, 对修改过存取类别的客体和新生成客体的存取类别设置成全空, 即不允许任何主体对它们具有任何存取模式。因此, 这个函数满足不变式。但我们发现, 这个函数可能会产生严重违反系统安全性的转移, 即可以降低客体的存取类别, 这就可能引起高密级的信息向低密级的用户泄漏。产生这个问题的原因在于, 原来的系统安全策略中并没有对客体存取类别的变化做任何说明。

```

FUNCTION 3: Create/Change _ object(o, c)
    'oclass(o) = c; and
    if o ∈ O then 'o = O ∪ {o}; and
    for all s ∈ S, A(s, o) = Φ

```

然而,如果在安全策略中加入这样一条性质:

- 性质(c):任何一个客体的存取类别都不能向下降的方向转移。

显然,Create/change _ object 就违反了这条原则。由于向下降转移的概念涉及到一个特殊类型的状态转移,所以,需要用一条数学描述将这个性质转化成一条约束条件:

约束 1: For all $o \in O$, $'oclass(o) \geq oclass(o)$

这条约束条件说明了:在新状态中,客体的存取类型只能上升或维持在原来的水平上。当然遵循这个约束条件的函数还必须满足不变式的要求。

(2) 对主体的限制。

对状态转移还需要附加一些其他的约束,即限制主体可以调用的操作。比较普遍的一个约束是:防止主体对那些它们本来不具任何存取权限的客体进行属性修改。例如,“允许主体为自己分配对某一客体的存取权”这样的操作就不合适了。相应于这种约束,模型中的安全性质应该做适当改动,这种改动是为了限制主体对存取模式的修改。

- 性质(d):仅当某主体能够读某客体时,才允许该主体修改另一个主体对该客体的存取权。

相应地可以归纳出一个约束条件,在这里,首先使用了状态变量 subj。

约束 2: For all $o \in O$,

```

if r ∈ A(subj, o)
    then for all s ∈ S, 'A(s, o) = A(s, o)

```

显然,Set _ access 不满足这条约束。

由于所选例子是一个简单模型,只有一个函数能够改变存取模式,所以,才能写一个比较简单的约束条件,它仅适用于 Set _ access 函数所涉及的一个客体。虽然这条约束比较简单,但是它防止了违反安全性质的模型操作,所以,它还是比较可靠的。重要的是要用尽可能的一般方式来描述约束条件,而不要涉及模型中比较特殊的操作,如果在约束条件中包含了一个特殊的操作信息,那么,就有可能漏掉某些情况而产生违反安全需求的后果,特别是过后要改进这个模型时,就有可能忘记为什么在这里要有一条约束条件。

(3) 对信息的控制。

状态机访问模型的一个局限性是:仅仅限制了对存取权的修改,而对客体内容的修改没

有任何限制。由于安全模型的目的是将安全策略形式化,所以,在许多情况下这是可以接收的,它不像对系统功能描述得那么详细。如果想对数据内容进行某种操作,如:

```
FUNCTION 4: Write _ object(o, d)
  If o ∈ O and w ∈ A(subj, o)
    Then 'contents(o) = d
```

`Write _ object` 并没有改变在不变式中所涉及到的任何状态变量,也没有涉及到目前为止所定义过的约束条件,根据给出的模型,这个函数是安全的。从直观上看,它还遵循了提出的安全性质,即除非主体对客体具有写存取权,否则是不允许主体对客体进行写操作的。只有当修改访问矩阵时,满足适当的存取类之间的关系,主体才能获得写存取权。即使去掉函数中 $w \in A(\text{subj}, o)$ 这一条,它还是不违反不变式以及约束条件的要求。但是,很显然,它违反了性质(b)

由于主体仅仅表达了主体对客体可能的访问模式,而没有考虑主体是否实际对该客体进行了读/写访问,所以,这个模型(不是策略)是不充分的。为了解决这个问题,可以附加一条性质:

- 性质(e):仅当某主体对某客体具有写存取权时,该主体才可以修改该客体。

这个性质可以解释成下述约束条件:

约束 3: For all $o \in O$,

```
if wA(subj, o)
  then 'contents(o) = contents(o)
```

模型需要将约束条件作为安全定义的一部分。传统上,在安全状态的定义中并没有提到客体的内容,在模型的转移函数中也没有包括它们,无论在什么场合,安全模型的目的就是体现访问策略,而不考虑实现细节。但是,由于抽象模型化方法已经在越来越多的系统中得到了应用。并且在一个实际系统中可以用许多方法对信息进行读/写操作(可以通过硬件或软件),而这些操作并不能以明显的方式遵循访问矩阵的要求。人们对模型的状态变量所未表达的信息进行的各种操作很不放心,而就倾向于将这些信息以及对这些信息的操作规则附加到模型中去,这就是必须对这些操作的性质予以说明的原因。

正如不能证明关于安全的不变式定义能恰当地表达安全策略一样,并不能证明这些约束条件是很全面的。在一个完备的模型中,很可能罗列了大量的约束条件,有些限制虽然是必要的,但需要很复杂的证明。然而,绝大部分约束条件仅仅表达了模型中转移函数的一些行为,而与安全无关。只需保留能明显影响系统安全性质的约束条件,而去掉那些对系统安全没有什么影响的约束条件。除非有理由证明,去掉某个约束条件会使系统产生安全漏洞。没有什么一般性的工具能够帮助我们查找漏洞,但是,模型越简单,发现漏洞越容易。非常重要的一点是,要认识到对状态转移的约束是定义系统安全需求的不可分割的一部分。

5. Bell&LaPadul 安全模型

Bell&LaPadul 安全模型是首批安全模型之一,也是目前最常使用的模型。它是由 David Bell 和 Leonard LaPadula 开发的,目的是将遵守军事安全策略的计算机操作模型化。这项工作的先期工作是在 Case Western Reserve University 进行的,该模型的最终目的是描述计算机的多级安全操作规则。BLP 模型已经成为多级安全模型最著名的形式,所以,多级安全的概念往往被视为 BLP 模型。还有几个模型也满足多级安全策略,虽然每种模型都试图用不同的方式表达安全策略,但所用的策略却是相同的。

在将多级安全策略形式化时, BLP 模型定义了一个存取类结构,它包括密级与一个类别集合,并且在存取类之间建立了一个偏序关系,称为“支配”。如果采用数学记号,可以将支配操作简单记做 \geq 。在这里,符号 \geq 并不表示数值上的比较关系,“A 支配 B”记做 $A \geq B$ 。为了避免混淆,一般不使用符号 $<$,也不使用符号 $>$ ($A > B$ 意指 A 支配 B,但不等于 B)。用等号 $=$ 来表达传统上的意义,即 $A = B$ 意指 A 与 B 的存取类是相同的。

“ \geq ”操作描述了存取类的一种偏序关系,它有如下数学性质:

- 自反性 $A \geq A$
- 反对称性 如果 $A \geq B$ 并且 $B \geq A$, 那么 $A = B$
- 传递性 如果 $A \geq B$ 并且 $B \geq C$, 那么 $A \geq C$

最容易出现的错误是:如果 $A \not\geq B$,那么, $A \geq B$ 。实际上,并不总能利用支配关系来比较两个存取类,严格地说,称这种情况为 A 与 B 是不相交的,用数学术语来描述就是 A 与 B 不可比。

除了定义偏序关系,再加入两个性质,就使存取类集成为一个格。给定任意两个存取类 A 与 B(不管它们是否可比):

- 在由 A 与 B 支配的所有存取类的集合中,存在惟一的最大下界,它受集合中所有其他存取类的支配。
- 在由 A 与 B 支配的所有存取类的集合中,存在惟一的最小上界,它支配集合中所有其他存取类。

不必关心为什么把这种偏序集称为格,但应该明白,将安全策略表达成一个格是许多安全模型的基本需要。

给定一个访问类的有限集,可以利用格的性质来定义两个重要的存取类,一个是 SYSTEM HIGH,它支配其他所有的存取类,一个是 SYSTEM LOW,它受其他所有存取类的支配。可以简单地把 SYSTEM HIGH 看作具有最高安全级别并具有所有类别的存取类,而把 SYSTEM LOW 看作具有最低存取类并不具任何类别的存取类。当然,还可以对这两个存取类定义其他的值,一般来说这不受什么限制。

BLP 模型的安全策略包括两部分:强制策略与自主策略。自主策略是用一个访问矩阵

来表示的,它包括读、写、运行及控制等存取模式。头几个存取模式是很容易理解的,最后一个,即控制模式,是用来表示主体是否可以将它对某一客体的存取权转移给其他主体。强制策略通过比较主体与客体的存取类属性来控制主体对客体的访问。当对主体分配存取权时,系统不仅要进行自主校验还要进行强制校验。

在 BLP 模型中,大约有 20 个转移函数或称操作规则,这些函数可以分成三种类型:

- 修改访问矩阵。
- 请求并获得对某一客体的访问权。
- 生成与删除客体。

这些函数都经过了证明,它们能够使系统维持在安全状态。BLP 模型没有对客体内容直接进行读/写的函数,它在实现时规定,对客体的任何访问都是以适当的访问请求开始的。读/写访问是根据分配的访问模式进行的。例如,在一个典型的操作系统中,打开一个文件就相应产生了一个访问请求,随后,读/写访问受到在打开文件时产生的访问模式的限制。

除了对强制安全策略进行校验外,转移函数还要遵守一个相当严格的平稳性约束。这约束用来防止客体的访问类受到修改,它是非常必要的,如果一个客体的访问类型能够被修改,那么,已分配的对客体的访问操作可能就不再满足安全状态的定义了。

Bell 与 LaPadula 是根据 Multics 的概念开发 BLP 模型的,并且对该模型提出了一个 Multics 式的解释。虽然如此,这个模型还是可以用到许多其他系统中去。安全模型中的转移函数必须得严格的控制,以确保这些函数能够满足安全性质。

6. 安全操作系统的设计

操作系统由于其在计算系统中的地位和作用,要设计高度安全性的操作系统非常困难。操作系统功能复杂,事务繁忙,要处理中断及上下文转接且必须执行最少的代码以免阻延用户计算;操作系统还得承担整个计算机系统的安全保护责任,这就使得其设计工作更是难上加难。

现在来讨论具有高度安全性的操作系统的应用。先考察标准通用操作系统的基本设计;再考虑隔离。操作系统正是采用隔离同时支持用户域的共享与分割;接下来考察操作系统的“内核”的设计方法,这是提供安全性的有效途径。实际上,存在两种对“核”的不同解释,对这两者都将进行讨论;最后,考虑分层的或环状的结构化的设计。

(1) 基本设计原则

一个操作系统执行与安全性有关的几个功能:

- 用户认证。操作系统必须识别请求访问的每个用户,并核对其身份与实际是否相符。最普通的识别机制是口令比较。
- 存储器保护。每个用户程序必须在分得的存储区域内运行,不得访问未经授权的存储区域。这种保护也许还要控制用户自己对程序空间受限制部分的访问。不同的安全性,

如读、写和执行可能适用于用户的存储空间的各个部分。

- 文件和 I/O 设备的访问控制。操作系统必须保护用户和系统文件以防未经批准的用户的访问。

- 对一般目标的定位和访问控制。一般目标, 如允许并行的和允许同步的机制, 必须提供给用户。而这些目标的使用必须受到控制, 以便一个用户不致对其他用户产生副作用。

- 共享的实施。资源应恰当地为用户所获取。共享则需要保证完整性和一致性。

- 保证公平服务。所有的用户都期望提供 CPU 的使用和其他服务, 而不至于无限期的等下去。硬件记时与调度规则的联合使用保证这种公平性。

- 进程间通信和同步。正在执行的进程有时需要与其他进程通信或协调它们对共享资源的服务。操作系统像进程间的桥梁, 响应进程间异步通信或同步请求, 从而, 提供上述服务。

安全功能渗透于操作系统的工作方式中, 这意味着设计安全操作系统有两方面的事情要做。第一, 必须在操作系统设计的各个方面都考虑安全性。当设计了一部分之后, 必须及时检查它实现或提供的安全程度。第二, 既然安全性体现在整个操作系统中, 对一个设计上不具有或不够安全的操作系统添加安全特点是很困难的。安全性必须是操作系统初始设计的一部分。Saltzer 和 Schroeder 列出了设计安全操作系统的以下原则:

首先, 系统设计必须公开, 假设入侵者不知道系统的工作方式无疑是自欺欺人。

其次, 不可访问应该是缺省属性。至于合法访问被拒绝的错误, 会比未授权访问被容许的错误更快地报告出来。

第三, 检查当前权限。系统应该先检查权限, 确定该访问合法, 然后, 把这个信息保留以备后用, 这种方式是不可取的。许多系统是在打开文件时检查权限, 而不是在以后。这意味着一个用户打开文件几个星期后, 依然有权存取该文件, 即使文件主早就修改了文件的权限。

第四, 给每个进程赋予一个最小的可能权限。如果一个编辑器只有权存取它所编辑的文件(在编辑器启动时指定), 这时即使它带有特洛伊木马病毒, 也不会造成很大的损失。这个原则蕴含着一个小颗粒度的保护方案。

第五, 保护机制要力求简单一致, 嵌入到系统的底层。要想在不安全的系统上改造安全性几乎是不可能的。同正确性一样, 安全性也不是一个附加的特征。

第六, 采取的方案必须可以接受。如果用户觉得保护文件太麻烦, 他们根本就不会去保护, 然而, 在出现问题时他们会不停地抱怨系统的设计者, 这时对他们说“这是你自己的问题”, 用户一般是不会接受的。

(2) 分离(隔离)

有四种办法将一个用户(进程)与其他用户(进程)分离开: 即物理分离, 时间分离, 密码分离和逻辑分离。

- 物理分离。各进程使用不同的硬件设施。例如,敏感计算可用保留的计算机系统完成,非敏感任务则在公用的计算机系统上运行。
- 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 到中午运行非敏感任务。而敏感计算只在中午到下午 5:00 运行。
- 密码分离。安全保存加密的密码,使未经授权的用户不能以可读的形式访问到敏感数据。
- 逻辑分离。也称隔离,当参照监控器把一个用户的目标与其他用户的目标分隔开时,就是使用了逻辑分离。
- 安全操作系统使用这些所有的分离方法。

(3) 核

核是操作系统的一部分,完成低级支撑功能。在标准的操作系统设计中,核实现同步,过程间通信,信息传递及中断处理这样一些操作。核也称为内核或核心,安全核的基本思想是:在一个大型操作系统中,只有相对少的一部分软件负责实施系统安全,提供实现整个操作系统的安全机制。安全核在硬件,操作系统,计算机系统的其他部分之间提供安全接口,并且保证任何逃避安全核的控制检查是不可能的。一般而言,安全核包含在操作系统核内,由于把它做得尽可能小,比较容易做正确性验证,因而,安全核自身是可信软件。

(4) 分层设计

如上所述,核化的计算机系统至少由四层组成:硬件,核,操作系统,用户。其中每一层本身可能包含一些子层。

(5) 环结构

环结构是分层设计的改进和具体化。详细的内容将在后面讨论。

7.6.2 身份认证机制

身份认证是安全操作系统应具备的最基本功能,是用户欲进入系统访问资源或网络中通信双方在进行数据传输之前实施审查和证实身份的操作。因而,身份认证机制成为大多数保护机制的基础。身份认证可分为两种:内部和外部身份认证。外部身份认证涉及验证某用户是否是其宣称的。例如,某用户用一个用户名登录了某系统,此系统的外部身份认证机制将进行检查以证实此用户的登录确实是预想中拥有此用户名的用户。最简单的外部验证是赋予每个账号一个口令,账号可能是广为人知的,例如,它可能被用作一个电子邮件地址,而口令则对不使用此账号的人员保密,此口令作为一个静态实体只能被此账号的拥有者或系统管理员改变。操作系统机制支持这种验证来确保不存在通过某些隐蔽的方式绕过验证机制的可能。

内部身份认证机制确保某进程不能表现为除了它自身以外的进程。若没有内部验证,

某用户可以创建一个看上去属于另一用户的进程。从而,即使是最高效的外部验证机制也会因为把这个用户的伪造进程看成另一个合法用户的进程而被轻易地绕过。

下面来考察一下在 UNIX 工作站网络中创建一套安全认证机制的困难性。最初,分时系统的主机被放置在安全可靠的机房中,用户通过控制台与主机物理连接的通信线路登录到主机上,这种情形在工作站引入 UNIX 系统后发生变化。今天的工作站物理上是放在用户的工作区,操作台已不是在安全区域的独立终端而是物理显示器上的一个窗口。早期分时系统的物理安全性在今天的 UNIX 工作站中以不复存在了。

工作站网络一般是由一个中心组织管理,而工作站的“拥有者”则是普通用户,他用不享有任何特殊管理权限的逻辑意义上的操作台登录到主机。通过给系统管理员提供根登录号,整个系统可实现远程管理而不允许本地用户修改系统文件。然而,设想用户关上了机器电源后又打开它,在这种情况下,传统 UNIX 工作站设计成了在单用户模式下启动而控制台在根模式。因此,任何想拥有根权限的用户都可以用开关电源使机器处于单用户模式,随心所欲地切换许可,然后,启动操作系统到多用户模式而得到根权限。这个缺陷很快得到证实并通过使机器必须以多用户模式启动而得到补救。这个例子说明了人们不能依赖于简单的诸如“此操作系统已被证实是可靠的,那么,此系统亦即可靠”这类假设。

计算机安全性是一个非常复杂的问题,它还包括处于操作系统设计之外的管理策略、道德问题以及物理设施的安全。由于计算机很容易受到安全威胁,保护系统安全的方法比保护该系统的软、硬件安全方法复杂的多。

保护机制可以依赖于计算机间的物理隔离,或者依赖于阻止远程用户通过网络进入计算机的逻辑隔离。大体上,除了假设未授权用户能够使用某种直接(或间接)的设备连入系统来建立某种逻辑通路的现象,本章不讨论物理安全问题。注意,要在一个失控的外部环境中解决问题是很困难的。在这种情形下,入侵者的行为根本不遵循任何特定的“游戏规则”。

1. 用户身份认证

当用户与系统交互时,操作系统是两者进行交互的软件代理。操作系统需要验证这些用户确实是他们所宣称的,这就是用户身份认证。如果系统能清晰地正确验证其用户,那么,许多保护方面的问题就会迎刃而解,但没有任何商业系统是据此假设设计的。如果这是可能的,那么,用户的行为将完全由他本人承担,并且任何其他用户都不能假装成此用户。但一般来说,这种确定的验证是不可能的。

在操作系统中,用户标识符和口令的组合被广泛用来进行用户身份认证。操作系统还可以使用附加手段来确认某用户是否是其宣称的那个用户。这些手段可能会涉及到类似于银行允许用户通过电话周转资金使用的技术。用户可能会被要求提供除密码外的附加信息,这些视认证权限的策略而定。现代操作系统甚至会采用指纹或眼球扫描技术等手段。

最简单的伪装事例是多人合用一个合法的用户名及口令。系统现在还不能区分这些伪

装成一个人的人们,所以,它不能为他们提供安全保证。另一更严重的伪装可能发生在某授权用户把用户名和口令遗留在公共地点或用户名早就为人所知,而口令又被人轻易猜出的情况。

一旦知道远程系统某用户登录名,计算机就可利用此用户名与另一台计算机开始一段对话并模拟出一次合法的远程登录对话,然后,伪装的计算机与验证进程建立起连接,并且迅速而又系统地对已知用户名尝试不同的口令。当验证进程检测到反复的密码提供错误时,它可能检测到这种闯入尝试,从而,在这种错误达到一定次数时终止这次连接。但伪装的机器稍后将可重新建立并继续搜寻此登录名的密码。

2. 网络中的身份认证

诸如用来传递电子邮件的文件传输机制也可被用于侵入机器。文件传输要求计算机能够将信息传递到另一个系统的文件空间中,发送方计算机在存储文件之前要能够获得对接受方机器的访问权限,而接受方机器要准备接收任意的将被保存到其文件系统中的文件。文件传输端口通常包含一则授权机制来证实发送方计算机拥有存储文件的权限。然而,在许多系统中,这种认证机制并不实用,因为,文件传输中的扩展验证增加了文件传输中的开销。

现代网络认证机制在检测接收文件是否含有病毒和蠕虫方面显得尤为重要。病毒和蠕虫的不同在于它们进入机器后的不同反应。病毒是一个隐藏于其他模块的软件模块,通过伪装成修错或升级补丁替换某个已存在模块,病毒可植根于一个文件系统中,它也可能在运行免费游戏或其他软件时被装载,这个隐藏文件将执行它所要做的任务。但它也会执行一些隐蔽的函数,譬如留下给入侵者日后使用的隐藏漏洞或植入破坏系统资源的程序。近几年来,病毒已成为软件业中显著的一部分,特别是因为计算机在两方面的进展:首先是软盘在个人计算机中的广泛使用,软盘是病毒的理想载体,这主要是因为它的容量和运行它的程序;其次,互联网成为病毒的温床,这主要是它提供了各种各样的邮件、新闻组、网页及免费软件。如今,已生产出很多明确以侦测病毒的存在以及根除病毒为目标的产品。

蠕虫不同于病毒之处在于它是一个活动的入侵实体,它可能以文件形式进入某机器,但以后它将独立地运行。一旦某个含蠕虫的文件进入到文件系统中,蠕虫便查找进程管理器中的漏洞以便执行它自己。举个例子,有一个著名的蠕虫程序——“Morris 蠕虫”,这个蠕虫由两个程序组成,即引导程序和真蠕虫程序。引导程序是一个名为 11.c 的 99 行程序,在所攻击的系统中进行编译并执行。运行时,它连接到它原来所在的机器上,下载真蠕虫程序并执行。在设法将自己隐蔽起来后,蠕虫接着查看这台机器的路由表,看看哪些机器连接到这台主机,并试图把引导程序传播到那些机器。

它诞生的目的是通过利用 finger 命令侵入 UNIX 系统。在 UNIX 系统中,命令“finger name@host 打印出一组标准的关于某用户身份证明信息的概要,其中的姓名部分可能是在口令文件中用户名的一部分。因此,假使某人知道该用户的真实姓名,他就可以很容易找出

该用户的登录名。其中的主机部分允许 finger 指令连接到远程主机上以执行 finger 指令去寻找该用户的信息。Finger 指令还打印其他诸如 .plan 的标准文件等。出于对其他用户的礼貌, 用户一般将他们的个人信息存放在 .plan 文件中。但这些文件可被用做猜测口令的基石。虽然 finger 指令对使用电子邮件程序来寻找登录名来说是无用的工具, 但它仍可作为一个被用于收集入侵信息的工具的一个例子。

Morris 蠕虫在远程机器上用一个对 finger 程序的字符串数组来讲过长的名字执行 finger 指令, 结果破坏了 finger 的后台运行栈, 导致当后台完成此命令后, 在为输入的 finger 调用服务之前, 后台再也不能返回它正在执行的程序。取而代之地, 它转移到一段代码, 它目的是为蠕虫运行唤醒远程 shell。这样, 蠕虫就控制了被入侵机器的一个进程, 从而使其能够消耗各种资源引起机器性能低下, 造成可观的破坏。尽管存在病毒、蠕虫及其他隐藏模块, 商业系统仍在使用基本的账号和口令方法来完成外部验证。通过仔细分析有运行病毒、蠕虫和猜测口令的倾向的行为使这些系统得到加强。不幸的是, 这种外部验证机制必须对每个账号考虑所有可能的入侵模型并逐一检查之一这是一项计算量可怕的工作。同时, 审核登录尝试的路线可用于检测归纳性的入侵尝试, 虽然, 对阻止破坏来讲可能太迟了。

3. Kerberos 网络身份认证

Kerberos 网络认证系统是由美国麻省理工学院 (MIT) 在 1980 年为研究计划“Athena”所开发出来的认证服务。适用于所谓的“客户——服务器”模式的开放式网络中。Kerberos 经由值得信赖的中央认证服务器, 对其他服务器提供认证用户的服务, 同时也对用户提供认证其他服务器的服务。Kerberos 可适用于 UNIX TCP/IP 网络中。Kerberos 有如下的需求:

- **安全:**一个网络窃听者应该不能获得必要的信息来假装成另一个用户。更一般地, Kerberos 应该足够强以防止潜在的对手发现脆弱的链路。
- **可靠:**对所有依赖 Kerberos 进行访问控制的服务来说, 无法获得 Kerberos 服务就意味着无法获得所要求的服务。因此, Kerberos 应该是高可靠性的, 应该使用分布式的服务器结构, 一个系统能够对另一个系统进行备份。
- **透明:**理想情况下, 除了需要输入一个口令外, 用户应该没有意识到鉴别服务的发生。
- **可伸缩:**系统应该拥有支持大量客户和服务器的能力, 这意味着需要一个模块化的、分布式结构。

在 Kerberos 中, 它假设在一台计算机(客户机)上的进程利用网络通信希望调用另一台计算机(服务器)上进程的服务。Kerberos 提供一台身份认证服务器及协议来允许客户机和服务器传送验证消息到特定会话中的协助进程中。在协议中遵守如图 7-8 所列的步骤:

步骤 1: 客户机向身份认证服务器请求服务器进程的证书。

步骤 2: 身份认证服务器把一张用用户密钥加密过的令牌和一个会话密钥当作证书返回。那张令牌和会话密钥(及其组合)只有客户机才能读取。令牌中含有客户机证明和用服

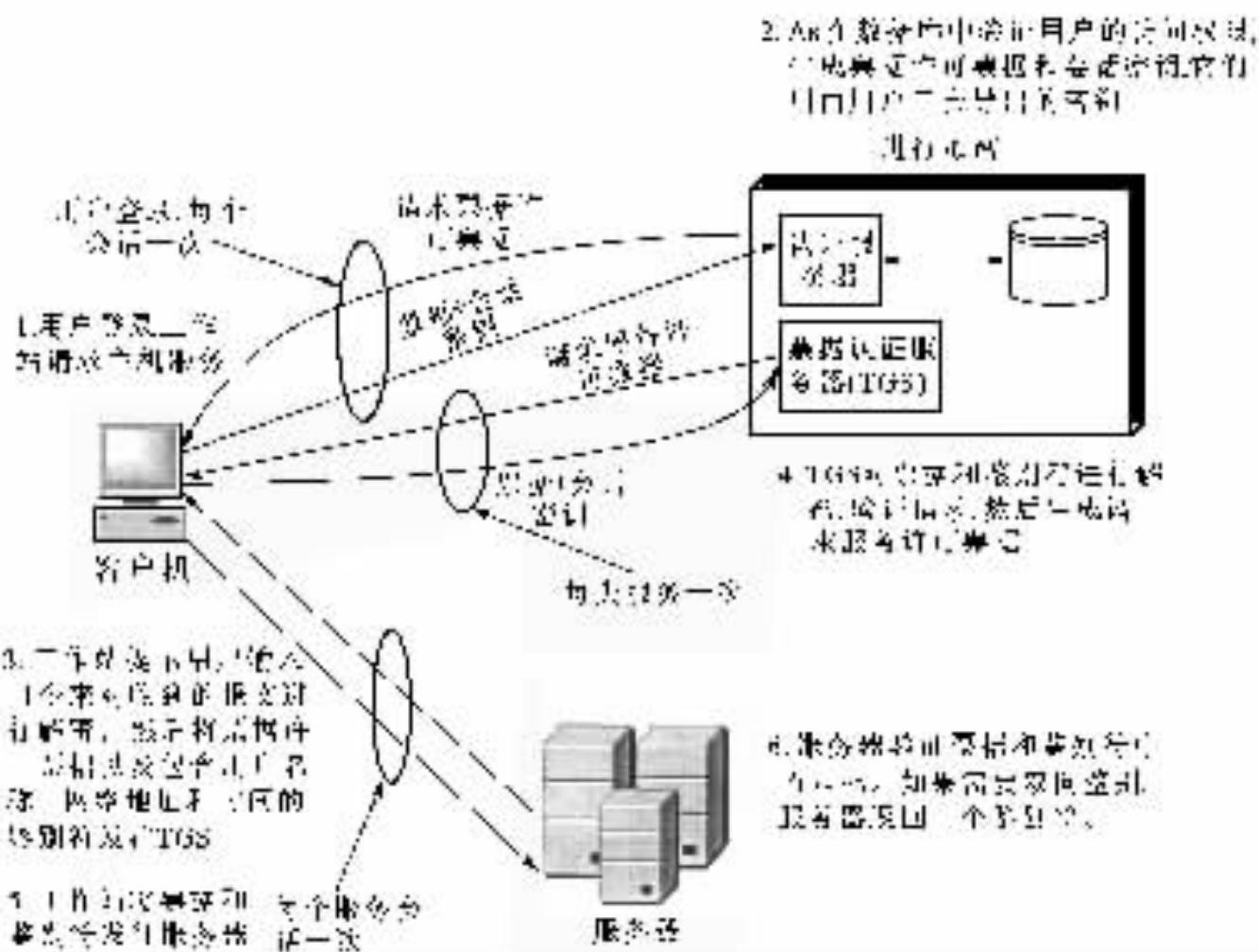


图 7-8 Kerberos 步骤

务器密钥加密的会话密钥的复印件的域, 这意味着能够解释该令牌的域的进程只能是服务器。

步骤 3: 当客户机获得证书后, 它将令牌和会话密钥解密, 同时保留一份会话密钥的拷贝以便它可验证来自服务器的信息。然后, 客户机发送一份加密域不变的令牌复印件给身份认证服务器。

步骤 4: 身份认证服务器对票据和鉴别符进行解密, 验证请求, 接着生成请求服务许可票据。

步骤 5: 身份认证服务器将票据和鉴别符发往服务器。

步骤 6: 服务器对令牌的复印件进行解密以便获取一份客户机证明和会话密钥的安全复印件。

在此协议中, 身份认证服务器必须是可信的, 因为, 它拥有一份客户机证明的安全复印件, 它知道如何进行只有客户机才能解密的信息加密方法, 还可以创建一个独特的会话密钥来代表客户机和服务器间的会话。同时, 身份认证服务器能够为客户机和服务器加密信息, 它可提供给客户机一个“容器”——含有客户机不可读但可传送给服务器的信息的证书。这类类似于拥有一张背面印有代表账号数字的条形码的信用卡, 你事实上根本不能从条形码中读

出什么,但当你把此卡提交给一台自动柜员机(服务器)时,柜员机可从条形码中读出你的账号。信用卡就是一张拥有加密账号数字的令牌(虽然它没有会话密钥)。

当完成步骤1到6后,客户机和服务器进程都拥有一份他们当作可信的身份认证服务器的加密过的信息的会话密钥的拷贝。网络上的入侵者若不知道如何对其进行解密就将不能读出这些加密信息。他也不能改变信息来生成欺骗性的证明文件。另外,服务器拥有一份可信的客户机证明的拷贝以便当客户机向服务器发送消息时,服务器能够验证用户的身份和会话密钥。

7.6.3 授权机制

1. 授权机制和内部授权访问

授权机制确认用户或进程只有在策略许可某种使用时才能够使用计算机的实体(例如资源)。授权机制依赖于安全的认证机制的存在。图7-9显示了经典的计算机系统授权访问的关键点。当一个用户试图访问计算机时,外部访问授权机制首先验证用户的身份,然后,再检查其是否拥有使用本计算机的权限。

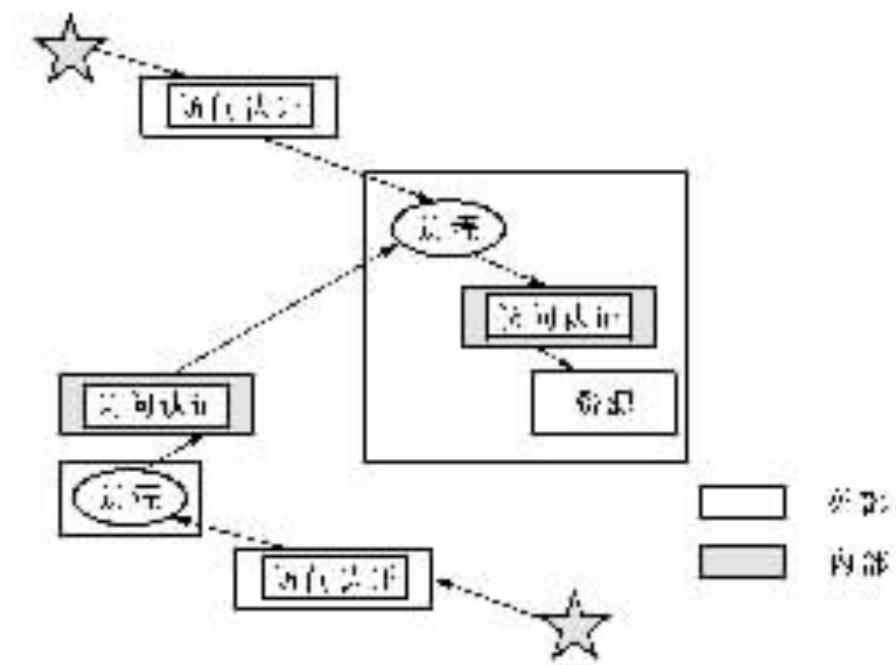


图7-9 经典计算机系统的授权访问

对交互式计算机,通常的做法是为系统保存一个所有授权用户的登录账号记录。一个可证明他(她)身份的用户便有权使用计算机,登录进某机器的用户若正试图登录另一台远程机器的话,他必须先被授权使用本地机器。远程机器可能会使用另一种形式的授权进程。

一旦某用户被授权使用某机器,此机器的操作系统将代表该用户分配一个执行进程。

在登录验证完毕后,用户将可自由使用命令行解释器(shell)进程来使用任意的资源。例如,用户可试图编辑系统的口令文件。如图 7-9 所示,每个使用目标资源的进程必须得到内部授权机制的授权。内外部访问授权机制是不一样的:外部机制授权让用户能进入计算机,内部访问机制授权让进程可存取资源。而内部机制只在用户进入系统后才开始活跃起来。不过,内部机制在每当用户输入一条指令后又会被暂时地唤醒,这是因为大多数指令被存储在文件中,当某个指令执行时,通常伴随文件的读写。

内部访问授权是管理共享资源的工作的一部分。在这里考虑的是保护进程的资源,不被其他进程的行为改变。设想进程 A 拥有资源 W、X、Y、Z,如图 7-10 所示,其中的某些资源与其他进程共享,例如,进程 B 对 W 有读取权限,而 C 对 W 有写的权限,B 对 Y 有写权限,而 C 对 Y 无任何权限,C 对 X 可读,B 对 X 无任何权限,并且 A 对 Z 有私有的所有权限。

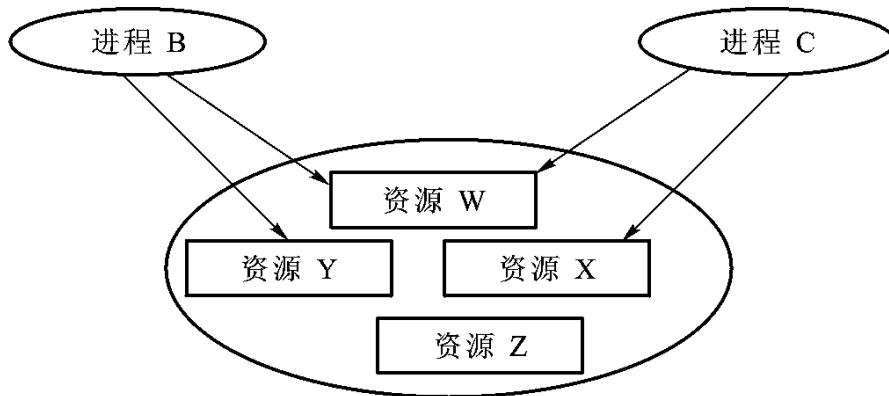


图 7-10 受限资源共享

研究者依其特点对保护问题进行了如下的分类,并试图在此环境中设计策略和机制来阐述它们。

(1) 共享参数:假如其他进程能够无区别地在某进程的地址空间内改变参数,那么,该进程的资源策略将受到破坏。例如,假设某进程在其他进程的地址空间内调用过程,而被调用过程改变了送入该过程的参数,故当调用者再次获得控制时,在它的地址空间内的变量已被调用的过程改变了。

(2) 限制:限制是参数共享的归纳性结果。假设某进程希望把分散的信息集中到某特定环境中,挑战在于包含资源的所有权限以使它们不能把资源传送到指定的进程集合之外。

(3) 分配权限:在一个保护系统中,可以允许一个进程为另一个进程提供使用其资源的特定权限。在某些情况下,第一个进程应能在任何时刻取消这些权限,权限只能暂时由一个进程赋予另一个进程。如果一个进程为另一个进程提供权限就可能产生一个敏感的问题:接收进程把收到的权限送给了其他并没有得到资源所有者许可的进程,在未得到资源所有者的明确许可下,有些保护系统是不允许这种授权繁殖现象的。

2. 资源保护模型

一般来讲,一个系统拥有积极的和消极的两部分。积极的部分,例如,进程或线程,代表了用户的行为;消极的部分,类似于资源,在保护系统中叫做对象。在现代操作系统中,一个进程在不同时刻,依赖于其当前所做的任务,对某对象有不同的权限。例如,一个执行系统调用的进程拥有访问操作系统的权限,它一般也拥有用户所有的权限。举例来说,当使用系统表和操作系统资源时,UNIX 系统在一个二进制文件中用 SetUID 位来允许此文件暂时性地拥有超级用户权限。在任何给定的时刻,某进程拥有的特定的一套权限都遵从于其所在的保护域。因此,任何关于使用某进程的决定的对象必须包括此进程执行的所在保护域的因素。一个主体是一个运行在特定保护域的进程。主体 A 可能是作为一个可执行程序的进程,而主体 Y 可能是作为一个系统调用的同一进程。“访问”的含义是不能概括一个进程是如何控制另一个进程的,这意味着对象的集合包括系统内所有的对象及所有的主体。现在,基本的保护模型可描述为用于指定主体和对象的动态联系的系统主体、对象和机制。

一个保护系统由一套指定保护策略的主体、对象和规则构成,它体现了通过系统保护状态定义的主体的可访问性。系统要保证为每次对象的调用都检查一下保护状态,如图 7-11 中的 X 通过主体 S 进行的检查。内部保护状态只有通过一套执行了外 10.443 28.925 TD(这

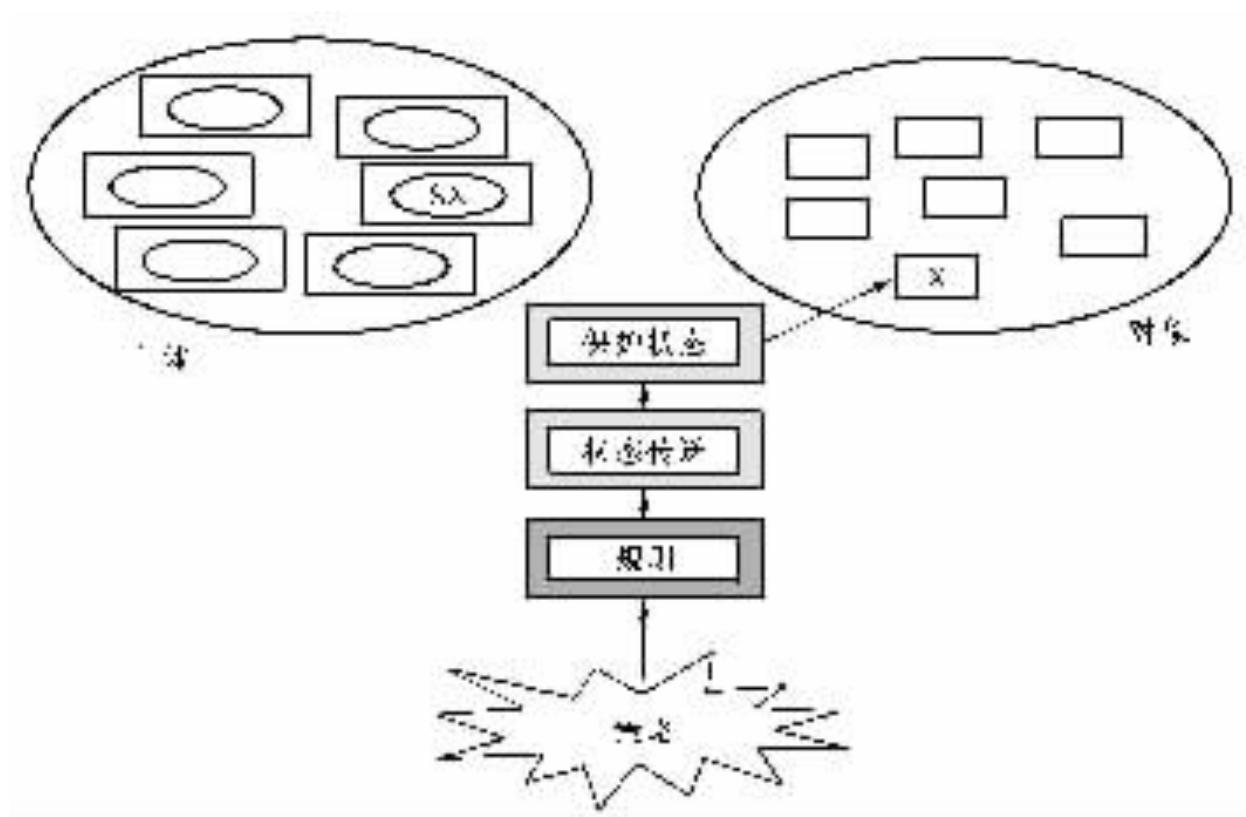


图 7-11 一个保护系统

保护状态可抽象为访问矩阵。在访问矩阵 A 中,用行代表主体(subject),用列代表对象(object,也称客体),所有主体也是对象,因为,进程需要能对其他进程实施控制。访问矩阵 A[S,O]中的每个元素是一个主体 S 对对象 O 可进行的访问操作。矩阵 A 的第 i 行表示主体 Si 对对象的操作权限集,矩阵第 j 列表示对象 Oj 允许主体可进行的操作权限集。访问控制机制可用三元组(S,O,A)来表示,称(S,O,A)为系统的保护状态,状态的变化就是三元组的变化,它由系统提供一套操作来完成。每次访问包含如图 7-12 所示的步骤。

步骤 1: 主体对对象 O 作访问类型 α 初始化。

步骤 2: 保护系统验证 S 并代表 S 产生 (S, α, O) , 由于身份由系统提供, 这个主体不能伪造主体身份。

步骤 3: 对象 O 的检查器查询 $A[S, O]$, 如果 $\alpha \in A[S, O]$ 则访问有效, 若 $\alpha \notin A[S, O]$ 则访问无效。

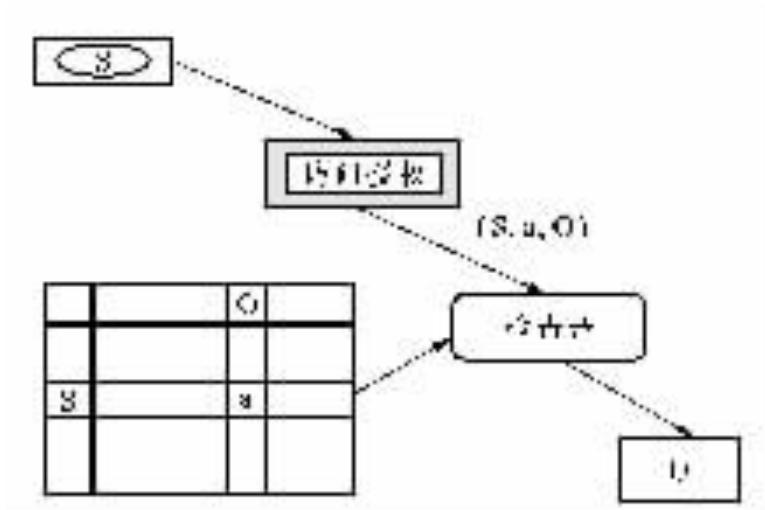


图 7-12

访问矩阵保护机制可用于执行许多不同的安全策略。例如,假设某简单系统是如下构成的:

$$\text{subjects} = \{S_1, S_2, S_3\}$$

$$\text{objects} = \text{subjects} \cup \{F1, F2, D1, D2\}$$

这里 $F1$ 和 $F2$ 表示文件, $D1, D2$ 表示设备。图 7-13 是一个代表了一个系统保护状态的访问矩阵, 每个主体对其自身有控制权。 S_1 对 S_2 有阻止、唤醒和占有的特权, 对 S_3 有占有的特权。文件 $F1$ 可被 S_1 进行读 * 和写 *。 S_2 是 $F1$ 的所有者, S_3 对 $F1$ 有删除权。

举例来讲, 如果 S_2 试图更新对 $F2$ 的访问, 则它先初始化这个访问, 然后, 导致保护系统建立一张表(S_2 , update F2)的纪录。这个纪录被送给 $F2$ 的检查器, 检查器查询 $A[S_2, F2]$, 因为, 此更新存在于 $A[S_2, F2]$ 中, 故这个访问是有效的, 于是此主体被许可更新此文件对象。若 S_2 试图对 $F2$ 执行运行的访问, 则它先初始化这个访问, 再由保护系统产生一个表(S_2 ,

executeF2) 的纪录, 这个纪录被送给 F2 的检查器检查 $A[S_2, F2]$, 因为运行权不在 $A[S_2, F2]$ 中, 故这个访问无效, 同时, 这个侵犯被报告给操作系统。

| | S_1 | S_2 | S_3 | F1 | F2 | D1 | D2 |
|-------|----------------------|----------|----------|----------|----|-----|----|
| S_1 | 控制 阻塞 唤醒 占有 | 控制 占有 | 读* 写* | | 查找 | 占有 | |
| S_2 | | 控制 结束 | 占有 | 更新 | 占有 | 查找* | |
| S_3 | | | 控制 删除 | 执行 占有 | | | |

图 7-13 保护状态

3. 保护状态的改变

保护系统用策略规则来控制用于切换保护状态的手段。就是说, 可通过选用在矩阵中出现的访问类型和定义一套保护状态转换规则来定义策略。作为例子, 这里用一套来自 Graham 和 Denning 的规则来阐明可传送于主体间的权限。

例如, 在图 7-14 中显示的规则实现了一则特定的保护策略。它们是用图 7-13 中所示的访问类型来定义的。在图中, S_0 试图通过执行改变访问矩阵入口 $A[S, O]$ 的命令来改变保护状态。例如, S_0 试图批准 S_3 对 D2 的读的访问权, 仅当此指令的所有者属于 $A[S_0, O]$ 时, 此指令才可执行。这导致读的访问权加入到 $A[S_3, D2]$ 中。这个安全策略例子的目的是为了提出伪装、共享参数和限制问题的。

| 规则 | S_0 的命令 | 权限 | 效果 |
|----|-----------------------------------|--------------------------------------------------|-------------------------------------|
| 1 | transfer $\{\alpha\}$ to $[S, O]$ | $\alpha \in A[S_0, O]$ | $A[S, O] = A[S, O] \cup \{\alpha\}$ |
| 2 | grant $\{\alpha\}$ to $[S, O]$ | $owner \in A[S_0, O]$ $control \in A[S_0, S]$ | $A[S, O] = A[S, O] \cup \{\alpha\}$ |
| 3 | deletefrom $[S, O]$ | or $owner \in A[S_0, S]$ | $A[S, O] = A[S, O] - \{\alpha\}$ |

图 7-14 策略规则

在策略规则中, 符号 * 被称为拷贝标志, 若某进程 S_0 对 O 有使用权且拷贝标志被设为 α 对 O 可访问, 则它可传送使用权 α 给对象 O 再给另一个进程 S 。在图 7-14 中, S_1 可为 F1 通过传输读和读* 权给 S_2 或 S_3 来改变保护状态, 因为, S_1 的读取权上有拷贝标志。对规则 2, 假如 S_0 占有对象 O , 不管有没有拷贝标志, S_0 主体都可批准给对象 O 权限。

拷贝标志及规则的设计目标是防止未知的主体间的权限繁殖, 拷贝标志可从一个主体传送给另一个主体, 或者可传送给一个拷贝标志已清除的主体。这里的传输规则必须是非破坏性的拷贝规则。另一种策略可能要求传输是破坏性的, 即当一个非所有者主体向另一个主体传输访问权后, 第一个主体就失去了它自己的访问权。这样的策略可用于严密地保护所有者主体的权限。删除规则被用于唤醒来自于另一个主体的对象, 在一个主体想删除一个对象之前, 它或者必须控制失去访问权的主体或者成为此对象的占有者。相应于这套规则暗指的策略, 若某主体是另一个主体的占有者, 那么, 它对那个主体也拥有控制权。这个例子阐述了决定某机制是否足够实现某一大类策略, 及此机制和策略的结合能达成一个可接受的解决方案的问题的基本的复杂性。而且, 它也显示了怎样利用规则来建立起预想中的策略。

Graham 和 Denning[1972]指出了在图 7-14 中的规则和其他一些规则定义了一个可用于解决一些本章开头提到的问题的保护系统:

- 伪装。此模型要求该实现能阻止某主体伪装成另一主体。验证模块可复杂到任何安全策略所要达到的, 在此模块验证完此主体后, 它为 S 产生出一个不可伪造的标记来执行一个到 X 的 α 访问, 然后, 把它送入 X 的检查器。这就杜绝了伪装。
- 限制和分配权限。对解决限制问题的模块和规则来说, 它们必须提供一种机制, 在此机制下, 权限被限制于指定的主体集合里。拷贝标志限制了传输过程中的权限繁殖, 同时, 所有权成为批准权限的前提。然而, 还有很多值得考虑的敏感问题。能够限制主体繁殖权限和信息的想法是值得期待的。因为, 读的访问提供了复制信息的能力, 在确信其不含有权限和信息的条件下, 允许一个不可信的子系统提供服务是困难的。通常, 这需要保证可疑的主体是无记忆性的, 即它不具备保存信息或将信息泄漏给其他主体的能力。这意味着只要考虑程序的行为就可完全解决限制问题。若不可信主体不能证实为无记忆性的, 限制问题就无法解决。
- 参数共享。可通过只允许不可信主体对对象的间接访问仔细地检查参数共享。某所有者主体可创建一个“门卫”主体来保护对象不受不可信主体的有害访问。本质上, 这导致用户把访问控制权委托给了“门卫”。最终用户只需要用户有对“门卫”主体的访问权就可获得对对象的明确的占有权。所有者可在任何时刻取消不可信任子系统对“门卫”的访问, 而“门卫”可对不可信主体的每次访问进行验证。
- 特洛伊木马。特洛伊木马问题的产生是因为某进程假定另一进程的权限在代表它运行。本模型把两个使用相同权限的进程区别为不同的主体, 本模型使利用一套恰当的规则来解决特洛伊木马问题成为可能——不同于在上面例子中的解决方法。然而, 许多规则集可能会为特定策略解决问题但并不能保证独立于策略的解决方案。

保护机制的引入会带来管理开销, 并会严重影响系统的运行性能, 基本保护模型要求每

次资源的使用之前必须通过一个检查器,操作系统设计者必须考虑在存在保护机制的情况下,运行性能上的开销是否值得。在某些信息必须保密的环境中(如关于某公司财政中心或涉及国家防御方面的信息),性能上的开销也许算不了什么,这些信息必须受到保护,否则,计算机系统根本起不了什么益处。尽管如此,对操作系统设计者来讲,面临的挑战就是要设计可能好的高效的机制。

4. 内部授权的实现

上面讨论的保护模型描述了一套可用于解决各种安全问题的逻辑组件。什么样的实现对此模型是开销高的呢?如何才能高效地实现访问矩阵?在虚拟存储系统中,实现一个和理论模型一模一样工作的系统的开销是巨大的,所以,只能是大致地实现模型所确定的行为。本节将讨论这类实现策略。

一般的保护机制建立在保存保护状态的方案之上,通过查询这些状态来使运行中的访问生效及改变状态。要实现机制需考虑以下几点:

- 访问矩阵并不是保护状态的惟一可能的代表,但它是大多数实现的基石。
- 访问矩阵必须存在某种安全的中间存储媒体上并只可被选定的进程进行读写。
- 此设计的目标是通过保护检查序列化所有的访问,这种序列化将确认当前的保护状态可用于使此次访问生效。
- 保护机制应能够通过某个主体来验证每个要求的来源而不是通过过程调用接收作为一个参数的主体身份。
- 要实现规则,检查器必须是一个受保护的进程。不得存在其他主体通过检查器,并确定传输机制(如通过共享资源)的可能。

下面将讨论在不同域中一个进程是如何具有不同的访问权,以及保护检查器和访问矩阵的实现。

(1) 保护域与状态隔离

保护系统程序或用户程序不受其他用户程序的破坏,采用的方法是对计算机系统设置不同工作状态或者说处理器具有不同的工作模式,采用两态模式时常称:管态和目态。运行在管态下的程序比运行在目态下的程序有更多的访问权——例如,对内存的访问权和运行扩展指令集的权限,限制用户使用容易造成系统混乱的那些机器指令,达到保护系统程序或其他用户程序的目的。有的计算机处理器可工作在多种状态下,也有的提供了保护环设施,都能更好地进行状态隔离。

这种两层次的域的概念可推广成一个含有 N 个在保护学中被叫做域环结构的同心圆的集合,环结构最早出现在[Organic 1972]中的 Multics 系统构架中。假设保护系统有 N 个保护环构成,在这些环中, R_0 到 R_s 支持操作系统域,而 R_{s+1} 到 R_{n-1} 则被应用程序使用。则 i

$< j$ 表示 R_i 比 R_j 拥有更多的权限。内核中最关键的部分在环 R_0 中运行。其次重要的操作系统层运行在 R_1 中。以此类推, 最安全的用户程序层运行在环 R_{s+1} 中, 越次要安全的程序运行在越往外的环中。在此模型中, 硬件超级用户模式一般当软件运行在编号最低的环(可能只能是 R_0 , 像在 Multics 系统中)中时才会被使用。操作系统的这部分是被最仔细设计和实现的, 由此推测也是被证明是无错的。

在一个环中运行的程序存在于一个分配到那个环上的文件中, 保护机制提供了一种进程可安全地改变域的手段——交叉环。若在 R_i 中的某文件正在执行, 那么, 这个进程可无需特定许可地调用在 $R_j (j >= i)$ 中的任何过程, 因为, 这个调用代表了一个向低保护的域的调用。但是, 当某个进程调用某个外部的环, 操作系统机制必须确保将被作为内部环引用的返回值和参数引用是被许可的。内部环调用可通过仅是让外部环软件从一个环门卫(一个进入的检查器)进入内部环来实现。每次内部环的交叉试图会引起内部授权机制来确认此次调用是否有效, 例如, 通过中断 R_0 的一部分来唤醒到目的环的门卫。

一旦某个进程进行了一个向内的调用, 它就改变了域; 就是说, 它变成了一个不同的主体。明显地, 当一个进程调用了一个内部环的进程, 目标函数保存在内部环中的一个独特的文件中。一种可选的看法是操作系统暂时地放大了此进程的权限以使该进程可运行在内部环中的进程。当该进程返回到外部环中时, 它将再变换它的域并回到它原先的拥有权限。

一般的环结构并不需要支持内部环数据访问, 只要支持过程调用。内部环中的数据只能用一个相应于内部环的过程来访问, 很像某抽象数据类型只允许通过公共接口对其进行引用。

环结构被用于当今的计算机体系结构中。例如, Intel 80386 微处理器采用了一种和上面的讨论有相似之处的四层结构。在 Intel 的例子中有三层指令集。层 2 和 3 的指令是普通的应用程序指令集, 尽管操作系统代码的非关键性的部分也是运行在层 2 的。层 1 的指令包括输入输出指令。层 0 的指令通过使用系统操作全局描述表来操纵内存段, 它也执行设备环境的转换。这种结构及它的后继产品(Intel 80486 及奔腾处理器)意图在于让层 0 支持内存段操作, 而把输入输出操作交给某低安全级的层处理, 某一编号较大的环。操作系统的主体在层 2 中操作, 这里它的段可受到环结构的保护。

VAX/VMS 操作系统利用了处理器的四种模式构成保护环来加强对系统资源的保护和共享。这些处理器模式决定了: 指令执行特权、即处理器这时可执行的指令和存储访问特权、即当前指令可以存取的虚拟内存的位置。如图 7-15 所示四种模式:

- 内核(kernel)态。执行 VMS 操作系统的内核, 包括内存管理、中断处理、I/O 操作等。
- 执行(executive)态。执行操作系统的各种系统调用, 如文件操作等。
- 监管(supervisor)态。执行操作系统其余系统调用, 如应答用户请求。
- 用户(user)态。执行用户程序; 执行诸如编译、编辑、连接、和排错等各种实用程序。

处于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,一用户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,该指令会引起中断,从而,将控制转交给处于高特权状态下的例程。执行返回指令可以通过正常或异常中断返回到断点。采用保护环时具有单向调用关系,如果在域 D_j 中欲调用 D_i 中的例程,则必然有 $j > i$ 。

(2) 空间隔离

现代操作系统常常为不同作业分配不同的地址空间,避免相互干扰。在多道程序环境中,空间隔离十分重要,它能使每个用户进程正确运行。如果一个进程错误地写信息到另一个进程的地址空间,会导致严重的后果。

每个用户进程的内存空间可以通过虚拟存储技术来实现内存保护,分页、分段或段页式,可提供有效的内存隔离。这时操作系统能确保每个页面或每个段只被其所属进程或授权进程访问。隔离技术能保证系统程序和用户程序的安全性,操作系统中还会采用各种技术实现其他资源的保护。

(3) 访问控制表和权能表

由系统中的所有主体和客体组成的访问矩阵会相当大,存储和检索的时空开销也就很大。图 7-16 给出了访问矩阵的一个例子。但在实际的系统中,部分主体和部分对象不发生联系,矩阵是稀疏的,因为,大多数客体只会有几个主体进行访问,而大多数主体也只会访问少数客体。这暗示高效的实现应使用入口索引表而不是将矩阵存在一个巨大的稀疏矩阵数组中。

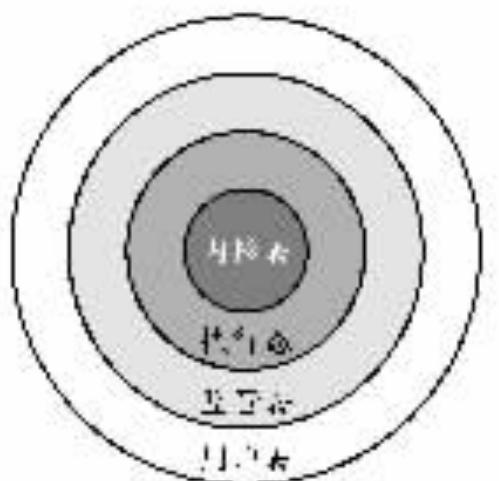


图 7-15 VAX/VMS 操作系统保护环

| | | 客体 | | | 主体 | |
|------|-----|------|------|------|------|------|
| | | 客体 1 | 客体 2 | 客体 3 | 主体 1 | 主体 2 |
| 主体 1 | 读/写 | 读 | | | | 写/读 |
| | | 读/不行 | 读/写 | | - | |
| | | 读/写 | 写 | | - | |

图 7-16 访问矩阵的例子

因此,在实现访问矩阵时,可以用两种方法来分解。

1) 访问控制表

把访问矩阵按列向量分解并存储称为“访问控制表”ACL (Access Control List), 每个列向

量存储为受保护对象(客体)的列表,在任何主体想访问对象的时刻,对象检查器只需简单地查询列表。访问控制表列出了用户和他们所允许对所有可存取对象的存取权。访问控制表可使用缺省值,该值表示允许没有显式具有某权限的用户享有缺省的权限。例如,主体 1 为进程 A, 主体 2 为进程 B, 主体 3 为进程 C; 客体 1 为文件 X, 客体 2 为程序 Y, 客体 3 为内存段 Z, 图 7-17a 为存取控制表的部分内容。

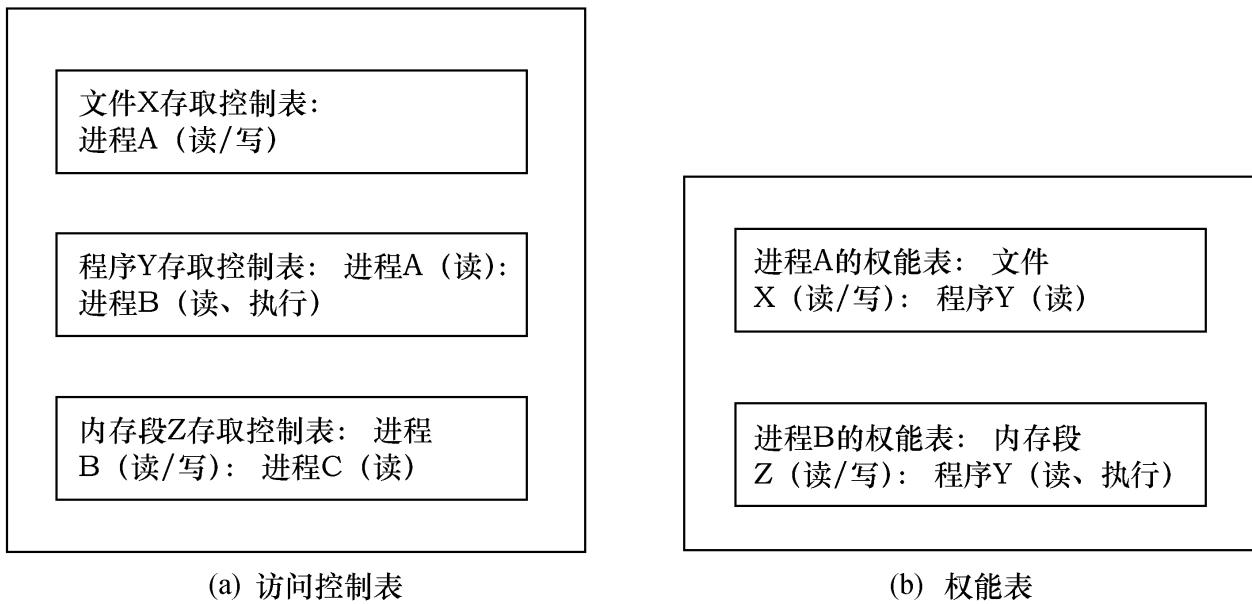


图 7-17 访问控制表和权能表

访问控制表已经使用了很多年,在这种方式下,资源管理者为每种资源配置了一个 ACL。在大多数应用中,主体只有在它打开或被分配资源(而不是每次访问)时才会受到访问权限检查。如果该主体及它的访问权限不在 ACL 中,那么,打开或分配资源就失败了。

UNIX 文件保护机制是 ACL 途径著名的应用。每个 UNIX 用户用一个用户身份证件 (UID) 进行验证。每个用户也可从属于不同的用户组,由组的身份证件 (GID) 注明。某进程的 UID 和 GID 是该进程描述符的一部分,这意味着当这些进程想访问某个文件时系统可方便地进行检查。UNIX 仅用十个字符描述了访问相应文件和目录所需的权限。开头的“d”字符表示此项目是一个目录;“-”字符表示其是一个文件。后面的 9 个字符三个一组地进行解释,第一组表示文件所有者对此文件的使用权限。第二组描述了该文件所属组成员对此文件所持有的权限,第三组描述了其他所有用户拥有的许可(它被叫做“通用”许可位)。如果某个三元组在第一个位置上有一个“r”字符,相应的用户对此文件或目录就有读取的许可;“-”表示该用户不具备读的许可。第二个位置的“w”代表写的许可,第三个位置的“x”代表运行的许可。

当某进程在执行一个可信软件模块时,每个文件也可拥有用于暂时提高此进程的权限的一个 setUID 标志位。当一个进程执行从一个 setUID 位为真的文件中加载的程序时,只要

它在执行该文件中的程序,就假定它就是该文件的 UID 的进程。

Windows 操作系统内核的最底层通过内涵一套完全的 ACL 机制 [Solomon, 1998] 来支持安全操作。内核的主体部分通过用户空间组件指定的保护策略来对每次对象的访问进行检查。无论什么时候,任何线程对系统进行调用,负责访问处理的内核部分都要提交给验证机制一份关于访问企图的描述。通过从 ACL 中得到的信息,验证机制确定线程的身份和访问类型,然后,验证此线程是否被许可访问它想访问的对象。

2) 权能表

把访问矩阵按行分解并存储称为“权能表”CL(Capability List), 它按主体来设立, 记录了授权给该主体对客体及操作的访问和执行权限。权能可能是一张 Kerberos 令牌, 也可能是“对盘 k 上 i 扇区的读取访问”和“对进程 j 的地址空间的虚拟地址的写入访问”。一旦某主体执行一个访问, 保护系统检查列表查看此主体是否拥有权能访问指定的客体。分配访问权给主体就像给某事件令牌一样。如果此主体在它的列表中没有此权能, 那么, 它可能连这个对象的名称都不知道。图 7-17b 为权能表的一个例子, 权能为用户指定其授权可访问的客体及相应存取方式。每个用户有许多权能, 并可把权能转移给别人。由于权能分散在系统中, 它引起比存取控制表更严重的安全性问题。权能应该不可忘记(unforgeable), 一种方法是操作系统以用户名义拥有所有权能, 而且权能必须存于用户不能存取的存储区域中。

权能包括两部分, 对象名和访问权。从权能角度来看, 系统内所有成分都是对象, 拥有权能代表了主体对描述对象的授权, 这是基于权能的系统的关键点。就是说, 当一个主体获得某权能时, 验证就发生了。一旦某权能被发布, 就没有必要让运行检查器和访问矩阵去检查每次访问。

权能的使用暗示了如下必然的属性: 权能使用的值必须从一个巨大的名字空间衍生。这是因为一个使用权能的系统将需要许多不同的权能实例来代表所有主体到所有对象的所有访问。

- 权能必须是独特的并且一旦被分配再也不能被重用。这防止了通过无意地给一个主体提供了原来对某对象的访问而引起的权能的“再生利用”现象。权能必须和似是而非的名字区别开来。例如, 系统不能把普通的整型数字或指针和权能弄混。

- 除了依赖于具体对象的权能, 如, 读、写、执行等外, 往往还有一些用于全部对象的一般权能, 主要有:

- 复制权能 为同一对象创建一新权能。
- 复制对象 创建一复制对象, 使其具有新权能。
- 删除权能 从权能表中删除一项, 相应对象不受影响。
- 删除对象 永久地删除对象和权能。

有两种基本的实现权能的途径。它们或者完全地在操作系统地址空间之内实现, 或者

硬件可实现对权能的特殊支持。作为一个实用的途径,权能有时是通过提供一个很大的空间,然后,从中随机地提取而得到。但这种途径并不能保证绝对的保护;权能不能被保证是独特的。虽然在很多情况下它们的确如此。

某权能在操作系统中可表示为一种类型的标量,概念上它的形式如下:

```
struct capability
{
    type tag;
    long addr;
}
```

的一个纪录。

若 c 是一个权能,那么,它的 tag 域 (c.tag) 就被设成拥有该权能的值;而它的 address 域 (c.addr) 就是该权能的通用地址。某对象只有当它拥有一个 c.addr 指出的访问类型所要的权能才能访问另一个对象。如果本来就有个对象检查器,那就只需要验证 c.tag 被设定拥有权能的值以能够保证此访问有效。

每个主体都要能获得和使用权能,但不允许它们创建权能。若主体的权能都维持在操作系统的空间内,主体就不能不受操作系统干预地创建一个类似于权能的数据结构。然而,主体可不受操作系统支配地利用它的权能去访问对象。既然这样,那么,权能就不会被操作系统独占地管理,数据结构中的 tag 域就可被省去,因为,权能的使用已经暗示了它的类型。例如,在一个分段的虚拟存储系统中,所有的权能被存在一个主体的权能段内。

标记可通过把每个 tag 域联系到每个内存单元的方法在硬件上实现。早期的 Burroughs 计算机结构就是使用硬件权能的例子。单词中的 tag 域可通过超级用户模式指令设成 capability 或 other。在对象保护检查器中,tag 域只可被超级用户模式指令进行读取。

Mach 是 Rochester Intelligent Gateway (RIG) 和 Accent 操作系统的后继产品, Mach 操作系统提供了内核级的权能,三者采用了不同的权能使用方法。IPC 机制中权能的使用是一个如何在现代操作系统中使用权能的好例子 [Accetta, et al. 1986]。

Mach IPC 建立在消息和端口上。消息是一种数据结构,而端口是一个在内核中实现的通信频道并用于从其他线程接收消息。每个端口接收不同特定类型的消息。若某线程试图挂起另一个线程,它把挂起消息送到那个线程的端口,然后,它就拥有了一个可控制第二个线程的权能。

端口是在被使用之前必须请求和分配的受保护的内核对象。若某线程知道一个端口,它就有权向那个端口发送消息,而接收者就会尊重这些消息。再使用一个端口之前,内核必须分配这个端口。端口等同于权能,若某线程拥有权能,它就可向目的端口发送消息;若没有就不能向端口发送信息。

(4) 内存锁和钥匙

早期的系统试图用锁和钥匙的方法提供保护机制,这是一种兼有 ACL 和权能列表属性的简单的机制。在更仔细地考察这些访问矩阵的完全实现之前,先来回顾一下锁和钥匙的途径。

20世纪70年代以来,为实现内存对象的保护检查器人们投入了相当大的努力。早期的保护检查器不支持完整的访问矩阵的通则,作为代替,它们使用了一种较弱的 ACL 形式。

计算机系统可分配的内存单元可以是一个字,一个分区,一个页面或一个段。在20世纪60年代,机器有时在 h 字节的可分配块上使用内存锁。假定每个块可分配一个 k 位的锁的值且每个进程的描述符都包含一个 k 位的钥匙设置(见图 7-18)。当一个内存块被分配给一个进程时,就设定一个和进程钥匙一样的锁。进程钥匙和内存锁只有系统处于超级用户模式时才能设定。在每次内存访问时,硬件通过保存钥匙的值到 CPU 的寄存器中和为内存设定一个锁寄存器对此次访问进行检查。因此,这些内存锁可看作访问控制列表。每当在它块内的数据要被引用,锁寄存器就伴随锁被加载了。如果锁和钥匙是一样的,那么,此次访问被许可;否则,此次访问企图就引起一个违法访问中断。在20世纪60年代的计算机中, $h = 16$, $k = 4$ 。这就意味着要么限制多道程序的道数小于 16,要么重复使用锁和钥匙以使某进程的钥匙能够同时打开另一进程在内存上的锁。

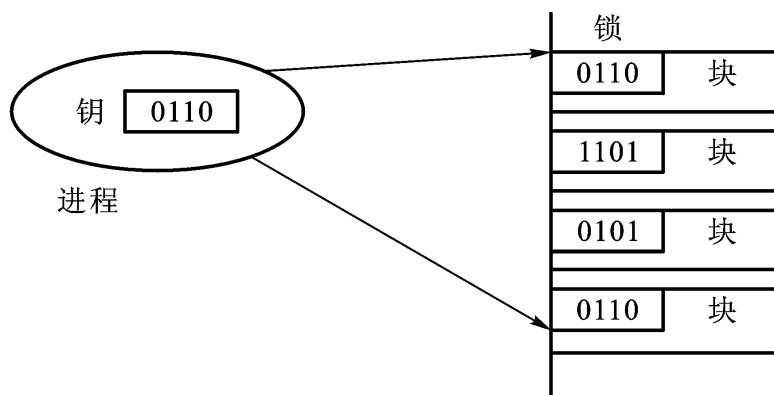


图 7-18 内存锁

这种途径既简单在使用中又有效。但是,它不能分辨不同种类的访问,也不允许操作系统或用户进程的共享,因为,锁和钥匙必须一模一样。此共享问题可通过保留特殊模式给特殊使用而解决。例如,某种钥匙模式可能是在超级用户模式下任何主体的“主人钥匙”。相对应的锁模式可能表示内存可“无保护”由任何主体访问。超级用户访问和共享于是被实现了,虽然,这是一个弱共享机制。

下面是 IBM System/370 系列机器的操作系统采用锁和钥匙方法提供存储保护机制的例子。在这种方法中,主存储器被分割成 2 KB 大小的存储块,每一块都由硬件另外设置了 7

位的存储控制键。其中 2 位指示占用了这个页框的页是否已被引用或修改过, 以供页面替换算法使用。其余各位用作保护机制:4 位二进数组成“存储保护键”和 1 位“取保护位”, 可由操作系统设置, 每个用户作业分得一个与其他作业不同的存储保护键码, 该作业的所有存储块的存储保护键都置成这个存储保护键码。当作业被挑选进入主存运行时, 操作系统把它的存储保护键码放入 PSW 的存储控制键字段(密钥), 这样每当处理器访问主存或 DMA 执行一个页面 I/O 时, 都要核对键键匹配情况, 以决定是否允许访问, 若运行进程试图对键键不同的主存页面进行访问, 则会产生一个主存保护中断。通常用户使用 1~15 存储保护键码, 操作系统使用 0 键, 它可以访问整个主存, 而且它还有权修改和设置密钥和保护键。“取保护位”指示“存储保护键”是适用于写还是适用于读和写, 键键匹配时允许写操作;若“取保护位”为 1, 则执行读操作时也要检查键键匹配与否, 若“取保护位”为 0, 即使键键不匹配, 也允许进行读访问, 目的是为了实现信息共享。

7.6.4 加密机制

加密是将信息编码成像密文一样难解形式的技术。在现代计算机系统中, 加密的重要性在迅速增长。在通过网络互连的计算机系统中, 想要提供一种信息不可达的机制是困难的。因此, 信息被加密成若不解密则其信息内容就不可见的形式, 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, 防止保密数据被窃取与泄密。数据加密技术可分两类:一类是数据传输加密技术, 目的是对网络传输中的数据流加密, 又分成链路加密和端加密。另一类是数据存储加密技术, 目的是防止系统中存储的数据的泄密, 又分成密文存储和存取控制。

1. 加解密算法分类

对付被动攻击的最有效方法是对机器内和网络中所存储及传输的数据进行加密, 使攻击者截获数据后, 无法了解数据的内容; 对付主动攻击的方法是对机器内和网络中所存储及传输的数据进行加密的基础上, 再采用某种鉴别技术, 以便及时发现数据的泄漏及篡改情况。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。定义一个加密函数(算法) encrypt 和一个解密函数 decrypt, 这里

$$\text{decrypt}(\text{key}', \text{encrypt}(\text{key}, \text{plain text})) = \text{plain text}$$

使用 encrypt 函数和加密密钥 key 把明文加密成密文, 用 decrypt 函数和解密密钥 key' 将密文转化为明文。明文(plain text)指被加密的文本; 密文(cipher text)指加密后的文本; 密钥是加解密算法中使用的关键参数。图 7-19 为数据加密的一般模型。

一般而言, 密码系统依其应用可对信息提供下列功能:

- 秘密性。防止非法的接受者窃取信息。

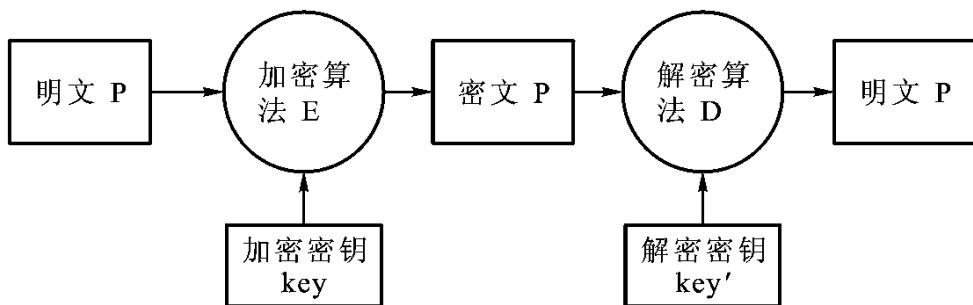


图 7-19 数据加密模型

- 鉴别性。确认信息来源的合法性,也即此信息确实是由发送方传送而非别人伪造。
- 完整性。确认信息没有被有意或无意的更改,及被部分取代,加入或删除等等。
- 防抵赖。发送方在事后,不可否认其传送过的信息。

密码系统为维持其最高安全性,均假设给予破译者最大的知识,即破译者对密码系统有最深切的了解。荷兰人 Kerckhoff 曾对密码系统做了下列假设:一个密码系统的安全性必须仅依赖其解密密钥,亦即在一个密码系统中除了解密密钥外,其余的加/解密算法等,均应假设为破译者完全知道。

只有在这个假设下,破译者若无法破解密码系统,此系统方有可能被称为安全的。密码体制的分类方法很多,通常从以下几个角度对密码体制进行划分:

(1) 根据密码算法所使用的加密密钥和解密密钥是否相同,能否由加密过程推导出解密过程,或者有解密过程推导出加密过程,可将密码体制分为对称密码体制(也叫单钥密码体制,秘密密钥体制,对称密钥密码体制)和非对称密码体制(也叫双钥密码体制,公开密钥体制,非对称密钥密码体制)。

对称技术的优点是具有很高的保密强度,可以达到经受国家级破译力量的分析和攻击。但它的密钥必须是通过安全可靠的途径传递,密钥管理成为影响系统安全的关键性因素,使它难以满足系统的开放性要求。采用非对称密码体制的每个用户都有一对选定的密钥,其中一个可以公开,一个由用户自己秘密保存。非对称密码体制的出现是现代密码学研究的一项重大突破,它的主要优点是可以适应开放性的使用环境,密钥管理问题相当简单,可以方便,安全地实现数字签名和验证,但它的保密强度目前还远远达不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完全不存在缺陷。

(2) 根据密码算法对明文信息的加密方式,可分为序列密码体制和分组密码体制。如果经过加密所得到的密文仅与给定的密码算法和密钥有关,与被处理的明文数据段在整个明文(或密文)中所处的位置无关,就叫作分组密码体制;如果密文不仅与最初给定的密码算法和密钥有关,同时也是被处理的数据段在明文(或密文)中所处的位置的函数,就叫序列密码体制。这两种体制之间还有许多中间类型。

(3) 按照在加密过程中是否注入了客观随机因素,可分为确定型密码体制和概率型密码体制。如果一个加密过程可以描述为:当明文,密钥被确定后,密文的形式也就惟一地确定,就称之为确定型密码体制。

如果一个加密过程可以描述为:当明文,密钥被确定后,密文的形式仍是不确定的,或者说对于给定的明文和密钥,总存在一个很大的密文的集合与之对应,而最后产生出来的密文则是通过客观随机因素在这个密文集合中随机地选出来的,就称之为概率型密码体制。

(4) 按照是否能进行可逆的加密变换,可以分为单向变换密码体制和双向变换密码体制。单向变换是一类特殊的密码体制,其性质是可以容易地把明文转换成密文,而把密文转换成原来的明文却很困难。而双向变换既可以把明文转换成密文,也可以把密文转换成原来的明文。

2. 基本的加解密算法

虽然加解密方法很多,但基本的加解密方法只有两种:易位法和置换法。其他方法都是基于这两种方法形成的。

(1) 易位法

它是通过重新排列明文中的各个字符的位置来形成密文,而字符本身不变。按字符易位时,先设计一个密钥,用它对明文进行易位而形成密文。在密钥中的字符不允许重复,明文按密钥来排序。

如图 7-20 所示,第一行为密钥,按密钥中的字母在英文字母表中的顺序来确定明文排列排列的列号。按此规则密钥中的 A 所对应的序号为 1,而 U 所对应的序号为 8,于是所给明文便转换成相应密文。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| M | E | G | A | B | U | C | K |
| 7 | 4 | 5 | 1 | 2 | 8 | 3 | 6 |
| P | I | e | a | s | e | t | r |
| a | n | s | f | e | r | o | n |
| e | m | i | l | l | i | o | n |
| d | o | 1 | 1 | a | r | s | t |
| o | m | y | S | w | i | s | s |
| B | a | n | k | a | c | c | o |
| u | n | t | s | i | x | t | w |
| o | t | w | o | a | b | c | d |

明文

Please transfer one million dollars to my Swiss
Bank account six two two ...

密文

AFLLSKSOSELAWAIATOOSCTCLNMAN
TESILYNTWRNNTSOWDPAEDOBNO ...

图 7-20 按字符易位加密法

(2) 置换法

最早由 Julius Caeser 提出的置换法十分简单, 只是将明文中的字母按英文字母表的顺序依次向后移动三位生成新的替代字母。这样一来, house 就变成了 krxvh, 让人认不出来。单纯的移动 k 位置换法很容易被破译, 比较好的置换法是进行映像。例如, 把 26 个英文字母映像到 26 个字母的域中, 图 7-21 是映像表的一个例子。利用这个映像, 可将 house 变成为 qifsb。

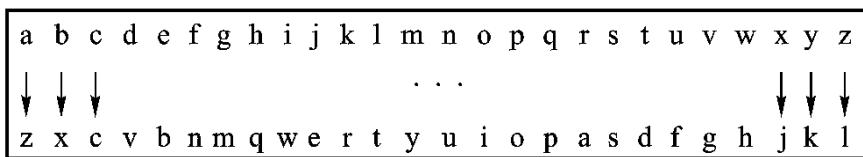


图 7-21 26 个英文字母的一种映象

3. 对称密钥法和公开密钥法

(1) 对称密钥法

数据加密标准 DES(Data Encryption Standard) 是一种对称密钥加密方法, 在 70 年代中期由美国 IBM 公司开发出来的, 且被美国国家标准局公布为数据加密标准的一种分组加密方法。直到今日, 尽管 DES 已历经了 20 个年头, 但在已知的公开文献中, 还是无法完全地、彻底地把 DES 给破解掉。换句话说, DES 这套加密方法至今仍被公认是安全的。

DES 属于分组加密法, 其每次加密或解密的分组大小均为 64 位, 所以, DES 没有密文扩充的问题。无论明文或密文, 当其数据大于 64 位时, 只要将明/密文中每 64 位当成一个分组而加以切割, 再对每一个分组做加密或解密即可。另一方面, DES 所用的加密或解密密钥也是 64 位大小, 但因其中有 8 个位是用来做奇偶校验, 64 位中真正起密钥作用的只有 56 位。而 DES 加密与解密所用的算法除了子密钥的顺序不同之外, 其他的部分则完全相同。DES 的加密过程可分成四步: 1) 对 64 位明文段进行初始易位处理, 2) 使用 56 位密钥进行 16 次迭代处理, 3) 对迭代的 64 位结果进行左 32 位与右 32 位位置互易, 4) 进行初始易位的逆变换, 最后, 产生一个 64 位密文。

由于 DES 的密钥只有 56 位, 仅有 256 个不同的密钥, 就有人提出, 随着计算机速度越来越快, 容易采用尝试法破解 DES 密文, 但无论怎样, DES 在最近几年得到了广泛使用, 特别在商业领域。

(2) 公开密钥法

DES 属于传统加密算法, 要求加解密的密钥是相同的(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, 则密钥又有泄密的可能。解决这一问题的最彻底的办法是不分配密钥, 这种方法就是于 1976 年由 Diffie 和 Hallman 提出的公开密钥法。要求密钥是对称的, 并不是一个必要条件, 可以设计出一个算法, 加密用一

个密钥,而解密用有联系的另一个密钥。另外,也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:

- 网中每个节点都产生一对密钥,用来对它接收的消息加密和解密。
- 每个系统都把加密密钥放在公共的文件中,这是公开密钥,另一个设为私有的,称私有密钥。
- 若 A 要向 B 发送消息,它就用 B 的公开密钥加密消息。
- 当 B 收到消息时,就用私钥解密。由于只有 B 知道其私钥,于是没有其他接收者可以解出消息。

公开密钥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,也不容抵赖。公开密钥法可用于数字签名,以代替传统的手工签名。

• 简单的数字签名 发送者 A 可使用私有解密密钥对明文进行加密,形成的密文传送给接收者 B。B 可利用 A 的公开加密密钥对所得密文进行解密,便得到明文。因为,除了 A 之外,谁也不具有解密密钥,因此,也只有 A 才能送出用他的解密密钥加密过的密文。如果 A 要抵赖,只需出示他的解密密钥加密过的密文,使其无法抵赖。

• 保密数字签名 上述方法解决了数字签名问题,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,可以按下面步骤进行:1)发送者 A 可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 再传送给 B,3)B 收到后,先用自己的私有密钥对密文 2 进行解密,得到了密文 1,4)B 再利用 A 的公开加密密钥对所得密文 1 进行解密,于是得到了明文。

5. 网络加密

防止网络资源被窃取、泄漏、篡改和被破坏的最好方法是网络加密,那么,要决定加密什么,在网络中何处加密?通常有两种网络加密技术:链路加密和端对端加密。

• 链路加密 指对相邻结点之间通信链路上所传输的数据(报文)进行加密,通信链路两端都配置了硬件加密装置,从而,通信链路上流动的信息都是安全的。其缺点是每次包交换都要进行加密与解密,因为,交换时必须从报文头部读出地址,以了解路由,于是消息在每个交换处易受攻击。链路加密常采用序列加密算法,它能有效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文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/解密的开始与结束信号。

• 端对端加密 采用链路加密时,通信线路上传输的是密文,但结点中是明文,因而,这种方式尚不能保证通信的安全性。端对端加密是在主机或前端机中,对用户数据进行加密。在加密数据送到目标主机后,由目标主机或前端机进行解密,这样密文通过整个网络,可以保证在中间结点不会出现明文。但端对端加密方式中,不能对报头进行加密,否则,中间结点无法知道目标地址。但报头不加密也会受到直接或间接的攻击,例如,通过报文中源和目标地址,了解一些机要部门的通信情况。

• 链路加密和端对端加密的组合使用 比较好的网络加密可把上述两种方法结合起来使用。利用端对端加密来使用户数据以密文形式穿越各个中间结点,而链路加密则使报头以密文形式在链路中传输,因而,不容易受到攻击。仅当报头处在交换器的内存中时才是明文。

• 密钥分配 密钥分配是指在两个交换数据的主体间传送密钥而不让外人知道的方法。在实现保密通信时,源系统需要有加密密钥,目标系统需要有解密密钥。应该如何分配、传送和保管这些密钥呢?采用人工方式管理时,可由专人传送密钥,专人保管密钥。在计算机联网后,可通过计算机网络来传送密钥,但要保证传递中不被人截获。为能保密,应先对密钥加密,然后再传递,并将密钥保存在密钥分配中心 KDC(Key Distribution Center)。有的系统有更严厉的管理措施,它不允许各个系统之间任意通信,在通信之前,要先经过访问控制中心 ACC(Access Control Center)的同意。

有多种方法实现密钥分配,对于两个主机 A 和 B 进行保密通信时,可用:(1)由 A 来选择密钥,传送给 B。(2)由第三者选择密钥,传送给 A 和 B。(3)如果 A、B 先后用了一个密钥,其中的一个将新密钥传送给另一个。(4)如果 A 和 B 都与 C 有加密链接,C 可在加密链上向 A 和 B 传送密钥。前两种方法都需要传送密钥。对于链路加密,由于每条链路的加密设备只与链的另一端交换数据,这类方法可行。但对于端对端加密,密钥的传送就很困难。在分布式系统中,任一主机或终端在一段时间内要与大量的其他结点交换数据,这样每个终端就需要大量的密钥,这在广域网分布系统中尤为严重。

第三种方法对链路加密和端对端加密都可行,但若一个攻击者成功获取了一个密钥,那么,所有密钥都可能被截获。第四种方法适用于端对端加密密钥。

7.6.5 审计机制

审计(auditing)作为一种事后追踪手段来保证系统的安全性,是对系统安全性实施的一种技术措施,也是对付计算机犯罪者们的利器。实际上,审计就是对涉及系统安全性的操作做完整的记录,以备有违反系统安全规则的事件发生后能有效地追查事件发生的地点、时间、类型、过程、结果和涉及的用户。必须实时记录的事件类型有:识别和确认机制(如注册和退出)、对资源的某种访问(如打开文件)、删除对象(如删除文件)、计算机管理员所做的操作(如修改口令)等。由于审记会增大系统开销,所以,系统应该确认哪些重要的事件必须加

以审计。

审计过程是一个独立过程,应把它与操作系统的其他功能隔离开来。系统应该生成、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。审计与报警功能相结合,安全效果会更好。

7.7 实例研究:Windows 2000/XP 的安全机制

7.7.1 Windows 2000/XP 安全性概述

Windows 2000/XP 提供了一组全面的、可配置的安全性服务,这些服务达到了美国政府用于受托操作系统的国防部 C2 级要求。1995 年,两个独立配置的 Windows NT Server 和 Windows NT Workstation 3.5 正式得到美国国家计算机安全中心 NCSC(United States National Computer Security Center) 的 C2 级认证。(详细信息请参见 <http://www.radium.ncsc.mil>)。1996 年,Windows NT Server 和 Windows NT Workstation 3.51 的独立和网络配置都通过了英国信息技术安全评估和认证 ITSEC(UK Information Technology Security Evaluation and Certification) 委员会的 F - C2/E3 级认证。这个评价与美国的 C2 级评价等同。(详细信息请参见 <http://www.itsec.gov.uk>)。目前,Windows NT 4.0 和 Windows 2000 正在接受美国 NCSC 和 ITSEC 的评测。

以下是安全性服务及其需要的基本特征:

- 安全登录机制。要求在允许用户访问系统之前,输入惟一的登录标识符和密码来标识自己。
- 谨慎访问控制。允许资源的所有者决定哪些用户可以访问资源和他们可以如何处理这些资源。所有者可以授权给某个用户或一组用户,允许他们进行各种访问。
- 安全审核。提供检测和记录与安全性有关的任何创建、访问或删除系统资源的事件或尝试的能力。登录标识符,记录所有用户的身份,这样便于跟踪任何执行非法操作的用户。
- 内存保护。防止非法进程访问其他进程的专用虚拟内存。另外,Windows 2000/XP 保证当物理内存页面分配给某个用户进程时,这一页中绝对不含有其他进程的脏数据。

Windows 2000/XP 通过它的安全性子系统和相关组件来达到这些需求,并引进了一系列安全性术语,例如活动目录、组织单元、用户、组、域、安全 ID、访问控制列表、访问令牌、用户权限和安全审核。与 Windows NT4 比较,为适应分布式安全性的需要,Windows 2000/XP 对安全性模型进行了相当的扩展。简单地说,这些增强包括:

- 活动目录。为大域提供了可升级的、灵活的账号管理,允许精确地访问控制和管理

委托。

- Kerberos 5 身份验证协议。它是一种成熟的作为网络身份验证默认协议的 Internet 安全性标准,为交互式操作身份验证和使用公共密钥证书的身份验证提供了基础。
- 基于 Secure Sockets Layer 3.0 的安全通道。
- CryptoAPI 2.0 提供了公共网络数据完整性和保密性的传送工业标准协议。

7.7.2 Windows 2000/XP 安全性系统组件

实现 Windows 2000/XP 的安全性系统的一些组件和数据库如下:

- 安全引用监视器(SRM)。是执行体(NTOSKRNL.EXE)的一个组件,该组件负责执行对对象的安全访问的检查、处理权限(用户权限)和产生任何的结果安全审核消息。
- 本地安全权限(LSA)服务器。是一个运行映像 LSASS.EXE 的用户态进程,它负责本地系统安全性规则(例如允许用户登录到机器的规则、密码规则、授予用户和组的权限列表以及系统安全性审核设置)、用户身份验证以及向“事件日志”发送安全性审核消息。
- LSA 策略数据库。是一个包含了系统安全性规则设置的数据库。该数据库被保存在注册表中的 HKEY - LOCAL - MACHINE \ security 下。它包含了这样一些信息: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怎样访问(交互、网络和服务登录方式);谁被赋予了哪些权限;以及执行的安全性审核的种类。
- 安全账号管理服务器。是一组负责管理数据库的子例程,这个数据库包含定义在本地机器上或用于域(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。
- SAM 数据库。是一个包含定义用户和组以及它们的密码和属性的数据库。该数据库被保存在 HKEY - LOCAL - MACHINE \ SAM 下的注册表中。
- 默认身份认证包。是一个被称为 MSV1 _ 0.DLL 的动态链接库(DLL),在进行 Windows 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 数据库中指定的相匹配,如果匹配,返回该用户的信息。
- 登录进程。是一个运行 WINLOGON.EXE 的用户态进程,它负责搜寻用户名和密码,将它们发送给 LSA 用以验证它们,并在用户会话中创建初始化进程。
- 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,是通过把它们发送到 LSASS 进程来验证的。

7.7.3 Windows 2000/XP 保护对象

保护对象是谨慎访问控制和审核的基本要素。Windows 2000/XP 上可以被保护的对象

包括文件、设备、邮件槽、己命名的和未命名的管道、进程、线程、事件、互斥体、信号量、可等待定时器、访问令牌、窗口、桌面、网络共享、服务、注册表键和打印机。

被导出到用户态的系统资源(和以后需要的安全性有效权限)是作为对象来实现的,因此,Windows 2000/XP 对象管理器就成为执行安全访问检查的关键关口。要控制谁可以处理对象,安全系统就必须首先明确每个用户的标识。之所以需要确认用户标识,是因为 Windows 2000/XP 在访问任何系统资源之前都要进行身份验证登录。当一个线程打开某对象的句柄时,对象管理器和安全系统就会使用调用者的安全标识来决定是否将申请的句柄授予调用者。

以下各节从两个角度检查对象保护:控制哪些用户可以访问哪些对象和识别用户的[安全信息](#)。

7.7.4 访问控制策略

当用户登录到 Windows 2000/XP 系统时,Windows 2000 使用名字/口令方案来验证该用户。如果可以接受这次登录,则为该用户创建一个进程,同时有一个访问令牌与这个进程对象相关联。访问令牌包括有关安全 ID(SID),它是基于安全目的,系统所知道的这个用户的标识符。当这个最初的用户进程派生出任何一个额外的进程时,新的进程对象继承了同一个访问令牌。

访问令牌有两种用途:

- 它负责协调所有必需的安全信息,从而,加速访问确定。当与一个用户相关联的任何进程试图访问时,安全子系统使用与该进程相关联的访问令牌来确定用户的访问特权。
- 允许每个进程以一种受限的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。

第二点的主要意义与用户的特权相关。访问令牌指明一个用户可能拥有哪些特权。通常,该标记被初始化成每种特权都处于禁止状态。随后,如果用户进程中的某一个需要执行一个授权操作,则该进程可以允许适合的特权并试图访问。之所以不希望在全系统保留一个用户所有的安全信息,是因为如果这样做,会导致只要允许一个进程的一项特权,就等于允许了所有进程的这项特权。

与每个对象相关联,并且使得进程间的访问成为可能的是安全描述符。安全描述符的主要组件是访问控制表,访问控制表为该对象确定了各个用户和用户组的访问权限。当一个进程试图访问该对象时,该进程的 SID 与该对象的访问控制表相匹配,来确定本次访问是否被允许。

当一个应用程序打开一个可得到的对象的引用时,Windows 2000/XP 验证该对象的安全描述符是否同意该应用程序的用户访问。如果检测成功,Windows 2000/XP 缓存这个允许的

访问权限。

Windows 2000/XP 安全机制的一个重要的特征是假冒的概念,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,服务器就可以临时假冒该客户的身份,使得它可以按该客户的权限发一个访问请求。在访问之后,服务器恢复自己的身份。

7.7.5 访问令牌

图 7-22(a)给出了访问令牌的通用结构,它包括以下参数:

- 安全 ID:在网络的所有机器中它惟一确定一个用户。这通常对应于用户的登录名。
- 组 SID:关于该用户属于哪些组的列表。一组用户 ID 号基于访问控制的目的被标识为一个组。每个组都有一个惟一的组 SID。对对象的访问可以基于组 SID,个人 SID 或它们的组合来定义。
- 特权:该用户可以调用的一组对安全性敏感的系统服务。一个例子是创建标记,另一个是设置备份特权,具有这个特权的用户允许使用备份工具备份它们通常不能读的文件。大多数用户没有这个特权。
- 默认的所有者:如果该进程创建了另一个对象,这个域确定了谁是新对象的所有者。通常,新进程的所有者和派生它的进程的所有者相同。但是,用户可以指定由该进程派生的任何进程默认的所有者是该用户所属的组 SID。
- 默认 ACL:这是适用于保护用户创建的所有对象的初始表。用户可以为他拥有的或所在的组所拥有的任何对象修改 ACL。

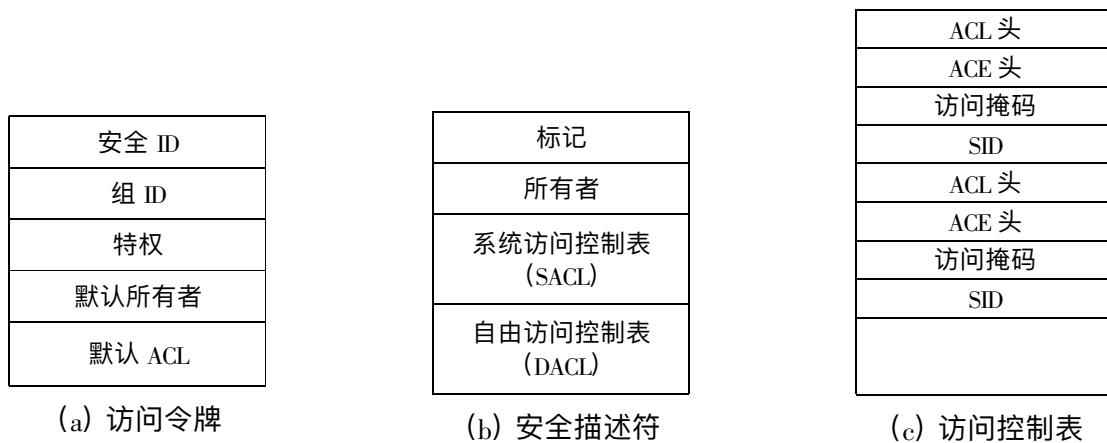


图 7-22 WINDOWS 2000/XP 的安全结构

7.7.6 安全描述符

图 7-22(b)给出了安全描述符的一般结构,它包括以下参数:

• 标记。定义了一个安全描述符的类型和内容。该标记指明是否存在 SACL 和 DACL, 它们是否通过默认机制被放置在该对象中以及描述符中的指针使用的是绝对地址还是相对地址。在网络上传送的对象, 如在 RPC 中传送信息, 也需要相关的描述符。

• 所有者。该对象的所有者可以在这个安全描述符上执行任何动作。所有者可以是一个单一的 SID, 也可以是一组 SID。所有者具有改变 DACL 内容的特权。

• 系统访问控制表。确定该对象上的哪种操作可以产生审核信息。应用程序必须在他的访问令牌中具有相应的特权, 可以读或写任何对象的 SACL。这是为了防止未授权的应用程序读 SACL(从而知道为了避免产生审核信息而不该做什么)或者写 SACL(产生大量的审核, 从而导致没有注意到违法操作)。

• 自由访问控制表(DACL)。确定哪些用户和组可以是哪些操作的访问对象。它由一组访问控制项(ACE)组成。

创建一个对象时, 创建进程可以把该进程的所有者指定成它自己的 SID 或者它的访问令牌中的某组 SID。创建进程不能指定一个不在当前访问令牌中的 SID 作为该进程的所有者。随后, 任何被授权可以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动作后隐蔽自己的踪迹。

由于访问控制表是 Windows 2000/XP 访问控制机制的核心。下面详细介绍它们的结构。每个表由表头和许多访问控制项组成。每项定义一个个人 SID 或组 SID, 访问掩码定义了该 SID 被授予的权限。当进程试图访问一个对象时, Windows 2000/XP 中该对象的管理程序从访问令牌中读取 SID 和组 SID, 然后, 扫描该对象的 DACL。如果发现有一项匹配, 即找到一个 ACE, 它的 SID 与访问令牌中的某 SID 匹配, 那么, 该进程具有该 ACE 的访问掩码所确定的访问权限。

图 7-23 给出了访问掩码的内容。最不重要的 16 位确定了适用于某特定类型的对象的访问权限。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。

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 位被称为标准访问类型:

• Synchronize。允许与该对象相关联的某些事件同步执行。特别地, 该对象可以用在等待函数中。

• Write _ Owner。允许程序修改该对象的所有者。这一点非常有用, 因为对象的所有者经常会改变对该对象的保护(所有者不会被拒绝写 DAC 访问)。

• Write _ DAC。允许应用程序修改 DACL, 因而可以修改在该对象上的保护。

• Read _ Control。允许应用程序查询所有者和该对象安全描述符的 DACL 域。

• Delete。允许应用程序删除这个对象。

访问掩码的高端部分也包含 4 个一般访问类型。这些位为在许多不同的对象类型中设

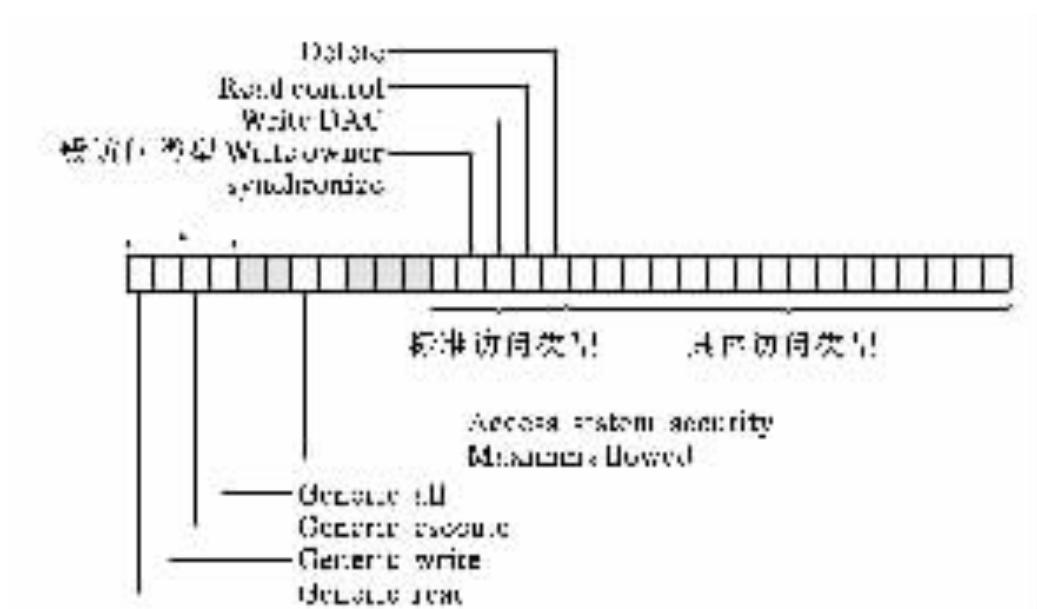


图 7-23 访问掩码

置具体的访问类型提供了一种方便的途径。例如,假如一应用程序希望创建几种类型的对象,并且确保用户可以对这些对象进行读访问(对不同的对象类型,读访问有不同的意义)。为包含没有一般访问位的每种类型的每个对象,应用程序必须为每类对象构造一个不同的 ACE,并且在创建每个对象时,小心地传递正确的 ACE。而创建一个表示一般概念允许读的 ACE,并且把这个 ACE 应用于创建的每个对象,比前面这种方法要方便的多,这就是设置一般访问位的目的,一般访问位包括:

- Generic _ all。允许所有访问
- Generic _ execute。如果是可执行的,则允许执行
- Generic _ write。允许只写访问
- Generic _ read。允许只读访问

一般位还反映了标准访问类型。例如,对于一个文件对象,Generic _ Read 位可以映射到标准位 Read _ control 和 Synchronize,并且可以映射到具体对象位 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 授予 SID 授予 Generic _ Read 权限的文件对象上,就等同于给这些 SID 授予了上面 5 种访问权限,并且好像它们是在访问掩码中单独定义的一样。

访问掩码中剩下的两位也有具体的含义。Access _ System _ Security 位允许为该对象修改审核和警告控制。但是,不仅一个 SID 的 ACE 中的这一位必须设置,并且该 SID 的进程的访问令牌也必须允许相应的特权。

最后,Maximum _ Allowed 位并不是一个真正的访问位,而是用于修改为这个 SID 扫描 DACL 的 Windows 2000/XP 算法。通常 Windows 2000/XP 扫描 DACL,直到它到达一个 ACE,

并且该 ACE 特别允许(置位)或拒绝(未置位)请求进程的访问请求,或者直到它到达了 DACL 的末尾,如果到达了 DACL 的末尾,则访问被拒绝。Maximum _ Allowed 位允许对象的所有者定义授予某个给定的用户一组最大访问权限。假设一个应用程序不知道它将要在某个对象上请求执行的所有操作,则对于请求的访问有三种选择:

试图为所有可能的访问打开该对象。这种方法的缺点是,即使应用程序具有本次会话所要求的所有的访问权限,访问也可能被拒绝。

只有当请求某一具体的方法时才打开一个对象,并且对于每种不同类型的请求为该对象打开一个新的句柄。这通常是一种比较可取的方法,因为这种方法不会有不必要的拒绝,也不会允许比必要的访问多的访问。但是,它增加了额外的开销。

试图为允许这个 SID 的访问打开该对象。其优点是用户不会被人为拒绝访问,但是应用程序会出现多于它所需要的访问。这种情况可能掩盖应用程序中的错误,Windows 2000/XP 安全机制的一个重要特征是应用程序可以为用户定义的对象使用 Windows 2000/XP 安全框架。例如,数据库服务器可以创建自己的安全描述符,并把它们附加在数据库的某一部分。除了普通的读/写访问约束,服务器还可以设置数据库专用的安全机制,如在一个结果集合中滚动或者执行连接操作。服务器负责定义具体权限的含义,并执行访问检查。但是,该检查将使用全系统的用户/组账号和审核日志,并在一个标准上下文环境中发生。可以证明,可扩展的安全模块对于外来文件系统的实现是非常有用的。

7.8 本 章 小 结

一般地说,信息系统的安全模型涉及到: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,所以操作系统的安全性是计算机系统安全性的基础。本章重点讨论了网络系统的安全性和操作系统的安全性。

安全性的需要可以通过检查一个组织所面临的各种安全威胁来评估,切断是对可用性的威胁;截取是对机密性的威胁;篡改是对完整性的威胁;伪造是对合法性的威胁。安全威胁之一是入侵者(黑客),入侵者的目的是获得对系统的访问或提高他访问系统的特权范围,一般来说,这需要入侵者获得已被保护的信息,可以通过种种措施保护口令,让黑客无法侵入系统。如果第一道防线失败,黑客侵入了系统,第二道防线就是入侵检测。常用的入侵检测方法有:统计异常检测和基于规则的检测。安全威胁之二是病毒,它利用现代计算机系统的脆弱性,或者获得未授权的访问,或者破坏系统资源和降低服务性能。

计算机系统的一种重要的安全控制技术是访问控制,访问控制的目标是确保只有授权用户才可以进入和访问特定的系统,并且对特定的数据的访问和修改都局限于授权的程序。

由于对计算机的大多数访问都通过网络和通信机制进行,因而,访问控制机制必须设计成能在分布式网络环境中有效地操作。

操作系统的安全性,主要包括:安全策略(是描述一组用于授权使用其计算机及信息资源的规则)、安全模型(精确描述系统的安全策略,对系统的安全需求,以及如何设计和实现安全控制的一个清晰全面的理解和描述)、安全机制(是实现安全策略描述的安全问题和实现系统的安全性)。在介绍有关的基本概念后,讨论了状态机模型及其开发步骤,以说明安全策略是怎样演变成抽象的安全模型的。然后,着重讨论了常用的安全机制:认证机制;授权机制;审核机制等系统安全性实现问题。分析了设计具有高度安全性的操作系统所涉及的问题,包括:用户认证、存储器保护、文件和 I/O 设备的访问控制、对一般目标的定位和访问控制、资源共享、进程同步和互斥等。

对于文件或数据库管理系统采用的访问控制的一个通用模型是访问矩阵,该模型的基本要素是三元组(主体、客体、访问权),访问矩阵是一种具体实现方法。由系统中的所有主体和客体组成的访问矩阵会相当大,为了节省时空开销,采用对访问矩阵按列或按行分解,这就是:访问控制表和权能表。它们在实际的操作系统中均被广为使用。

安全功能渗透于操作系统的工作和结构中,这意味着设计安全操作系统有两方面的事情要做。一是必须在操作系统设计的各个方面都考虑安全性。当设计了一部分之后,必须及时检查它实现或提供的安全程度。二是操作系统安全性必须是系统初始设计的一部分,而对一个不够安全的操作系统修修补来达到安全性是很困难的。

对付被动攻击的最有效方法是对机器内和网络中所存储及传输的数据进行加密,使攻击者截获数据后,无法了解数据的内容;对付主动攻击的方法是对机器内和网络中所存储及传输的数据进行加密的基础上,再采用某种鉴别技术,以便及时发现数据的泄漏及篡改情况。

本章最后讨论了加密技术,介绍了加密的定义和分类,基本的加密方法。DES 数据加密标准是目前常用的方法,由于使用了对称密钥,存在易被破译和分配密钥的麻烦。从而,公开密钥加密方法被开发出来,ISO 推荐的公开密钥数据加密标准是 RSA 体制。也简单介绍了数字签名技术和网络加密技术(链路加密和端对端加密)。

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. 试叙述计算机安全领域的主要内容。
3. 试叙述操作系统安全性的主要内容。

4. 计算机和网络系统的四项安全要求是什么？
5. 计算机或网络系统在安全性上受到的攻击类型有哪些？叙述其主要内容。
6. 计算机系统的资源分为硬件，软件，数据，以及通信线路与网络等几种。试述每种资源类型所面临的威胁的情况。
7. 什么是被动和主动威胁？它们发生在什么场合？
8. 列举操作系统可能提供的保护层次及其主要内容。
9. 面向用户访问控制的基本技术是什么？如何实现？
10. 面向数据访问控制的基本技术是什么？叙述其实现原理。
11. 讨论访问控制中主体和客体的关系。
12. 何谓入侵者？它分哪几种类型？
13. 口令文件如何保护？叙述其实现原理。
14. 口令文件的主要弱点是什么？如何克服它？
15. 有哪些口令选择策略？你通常采用什么策略来选择口令？
16. 有哪些入侵检测方法？是如何来进行入侵检测的？
17. 解释统计异常入侵检测与基于规则的入侵检测有什么不同？
18. 入侵检测的基本工具是什么？它是如何来为入侵检测服务的？
19. 什么是病毒？叙述其特性和分类？
20. 有哪些反病毒技术？试叙述这些技术的反病毒原理。
21. 解释策略和机制。操作系统中为什么强调要把策略与机制明确的分开？
22. 叙述安全模型的类型及其特点。
23. 什么是状态机模型？开发一个状态机模型一般需要哪些步骤？
24. 叙述 Bell&LaPadul 安全模型。
25. 叙述多级保护系统中“不向上读”规则和“不向下写”规则的重要性。
26. 身份认证机制是保护机制的基础，试叙述分类及其作用？
27. 叙述 Kerberos 网络身份认证系统的工作原理。
28. 解释内部访问授权机制的作用及实现原理。
29. 解释：存取控制矩阵、存取控制表、权能表。
30. 简述操作系统的安全保护技术：状态隔离和空间隔离。
31. 什么是计算机系统的安全性与计算机系统的可靠性？它们有什么关系？
32. 叙述 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？
33. 叙述操作系统中使用的两态模式和环模式保护机制。
34. 解释：密文、明文和密钥。
35. 解释：公钥、私钥和密钥。
36. 讨论常规加密和公钥加密的区别。
37. 试说明 DES 加解密处理过程。
38. 何谓对称和非对称加密算法？说明非对称加密体制。

39. Windows 2000/XP 中提供了哪些安全性服务, 基本特征是什么?
40. Windows 2000/XP 安全性组件有哪些? 叙述其主要功能。
41. 试述 Windows 2000/XP 的安全控制策略。
42. 试述 Windows 2000/XP 访问令牌的结构和功能。
43. 试述 Windows 2000/XP 安全描述符的结构和功能。

二、应用题

1. 有三种不同的保护机制:存取控制表、权能表和 UNIX/Linux 的 rwx 位。下面的各种问题分别适用于哪些机制? (1) Rick 希望建立除 Jennifer 以外,任何人都能读取他的文件。(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 假设用户被分为:教职工、学生、秘书等。

2. 考察下面的保护机制:给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 26 个英文字母表中选 4 个字符形成一个口令,一个攻击者的每秒钟一个的速度试探口令,直到试探结束再反馈给攻击者,那么,找到正确口令的时间是多少?

4. 考虑一个有 5 000 个用户的系统,假如只允许这些用户中的 4 990 个用户能存取一个文件。现问:(1)如何实现? (2)给出更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 二个私有文件,而且这两个用户都需使用共享文件 S。若文件系统对所有用户提供按名存取功能,试画出能保证存取正确性的文件系统目录结构。

6. 在一个带有四个终端的计算机系统中,今有四个学生上机实习,各自从终端上键入程序与数据,并都保存到磁盘上,恰巧他们为各自的文件均取名为 WJ,请问:系统应建立什么样的目录结构才能区别四个学生的程序? 简述系统如何为这四个学生存取各自的程序?

第八章 网络和分布式操作系统

8.1 计算机网络概述

8.1.1 计算机网络的概念

1. 计算机网络的定义和组成

计算机网络是利用各种通信手段,把地理上分散的、具有自治功能的多个计算机系统互连起来的,实现数据通信、资源共享、可互操作和协作处理的系统。

一个计算机网络包含三个组成部分:(1)资源子网——它含有若干台计算机,包括拥有资源的计算机(服务器)和请求资源的计算机(客户机)和各种终端设备;(2)通信子网——它由通信链路和通信处理机组成,如分组交换器、多路转换器、包组装与拆卸设备、网络控制中心和网关等,用于通信控制和通信处理;(3)通信协议——网络中主机和主机、主机和通信子网或通信子网和通信子网各结点之间进行通信的规则。

一个计算机网络系统是计算机和网络硬件、系统软件及应用软件的组合体,除了上述三个组成部分外,还有网络软件,网络软件由网络操作系统、网络支持软件和网络应用软件组成。网络操作系统用于完成网络的资源分配、共享、管理和协调任务;网络支持软件用于支持数据通信、网络互连、维护管理及各种网络活动,网络应用软件在网络操作系统的基础上向用户提供各种网络应用。

2. 计算机网络的功能

计算机网络的主要功能如下:

- 数据通信:计算机联网后,使地理上分散的计算机之间相互通信、交换信息。例如,电子邮件已成为世人接受的最快捷、廉价、方便的通信方式,此外,网上电话,视频会议等许多通信方式正在迅速发展。

- 资源共享:这是计算机网络的主要目的之一,在网络范围内,用户可以共享硬件、软件、信息资源,而不必考虑用户或资源的所在地理位置。

- 支持分布式信息处理:在计算机网络支撑下,许多大型信息处理问题可以借助于分

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,它使分散存储在网络中不同计算机节点上的数据,使用时如同集中存储和处理一样。

- 提高计算机系统的可靠性和可用性:网络中的计算机可以互为备份,让同类资源分布在不同计算机上,一旦某台机器出现故障,网络可以通过不同路由来访问这些资源,或者故障机器的任务可由其他计算机承担,从而,计算机系统的可靠性高。当网络中某些机器负荷过重时,也可以把一些任务分配给网络中空闲计算机去完成,提高了每台计算机的可用性。

3. 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 20 年来得到了迅猛发展。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形网,这类结构称第一代计算机网络。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,也被称作第二代计算机网络。它以通信子网为中心,许多主机和终端设备在通信子网的外围构成了资源子网。通信子网不再仅仅依赖于类似电话通信的电路交换方式,更多地采用适合于数据通信的分组交换方式,网络控制也由集中趋于分散,引入专门的通信控制处理机,大大降低了计算机网络的通信成本,这些特征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,许多计算机厂商制订自己的网络技术标准。如 IBM 公司的 SNA (System Network Architecture) 标准;Dec 公司的 DNA (Digital Network Architecture);Univac 公司的 DCA (Data Communication Architecture);Burroughs 公司的 BNA (Burroughs Network Architecture) 等等,增强了这些公司的产品在世界计算机市场上的竞争力。但是,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相连通,终于促成了网络国际标准的制定。1977 年国际标准化组织 (ISO) 成立了“计算机与信息处理标准化委员会 (TC97)”下属的“开放系统互联分技术委员会 (SC16)”,在现有各种网络标准的基础上,制定了“开放系统互联参考模型 (OSI/RM) 作为网络国际标准,推动了网络技术的进一步发展,开始了所谓的第三代计算机网络。

80 年代中期以来,计算机网络领域最引人注目的事件是美国 Internet 网的飞速发展,它仍属于第三代计算机网络,但有它自己的网络体系结构,没有完全使用 OSI/RM。进入 90 年代后,计算机网络发展更加迅速,目前正在向宽带综合业务数字网 (BISDN) 演变,这就是通常所说的第四代计算机网络。1993 年 9 月美国政府提出的国家信息基础设施计划 NII (National Information Infrastructure),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。

4. 计算机网络的分类

目前世界上出现了各种各样的计算机网络,对其进行分类的方法也很多,可以从不同角度,对计算机网络进行观察和分类。

(1) 按通信介质分类

按通信介质的不同,可以把网络划分为:

- ① 有线网:采用双绞线、同轴电缆、光纤等物理介质来传输数据的网络。
- ② 无线网:让电磁波通过自由空间去传输数据的网络,常用的有微波信道和卫星信道。

(2) 按通信传播方式分类

按通信传播方式,可把网络分为:

- ① 点对点传播网:以点对点方式把计算机连接起来,星形、树形和环形网都采用这种方式。
- ② 广播式传播网:用一个共同的传播介质把各计算机连接起来,主要有:以同轴电缆连接起来的总线形网;以微波、卫星方式传播的广播网。

(3) 按网络作用范围分类

按网络作用范围,可将网络划分成:

- ① 广域网 WAN (Wide Area Network):作用范围达几十公里到几千公里,所以又称远程网。
- ② 局域网 LAN (Local Area Network):通过高速通信线路把许多台计算机连接成网,网络作用范围常在几公里至几十公里以内。
- ③ 城域网 MAN (Metropolitan Area Network):作用范围介于 WAN 与 LAN 之间。
- ④ 内联网 Intranet:它是集 WAN、LAN 和数据服务于一体的一种网络,采用 Internet 技术把计算机连接起来,从而,建立起企业内部的网络(Enterprise Network)。

(4) 按网络使用的数据交换技术分类。

按网络的数据交换技术,可把网络分成:①电路交换网;②报文交换网;③报文分组交换网;④帧中继网;⑤ATM 网。

(5) 按网络的拓扑结构分类

按网络的拓扑结构,可把网络分成:①星型网;②树型网;③环形网;④网状型网;⑤混合型网。

(6) 按网络的控制方式分类

从网络管理者的观点来看,可以把网络分成:

- ①集中式控制网:网络中的所有控制和数据信息必须经过中央交换节点,链路都从中央交换节点向外辐射。
- ②分散式控制网:网络中加入了集中器和复用器,放置在终端集中的地方,它具有一定的交换和控制能力。

③分布式控制网:网络中的任一节点都至少和另外两个节点相连形成了格状网,信息可以从多条路径到达同一节点,因而,可靠性很高。

8.1.2 数据通信基本概念

计算机网络是指通过计算机技术与通信技术的结合来实现信息的传输、交换、存储和处理的系统。现代数据通信系统实质上是一个计算机网络系统,它由数据传输系统和数据处理系统两部分组成。数据传输子系统又叫通信子系统或通信子网,其主要任务是实现不同数据终端设备之间的数据传输;数据处理子系统又叫资源子系统或资源子网,它由许多计算机及终端设备组成,在网络中负责提供信息、接收信息和处理加工信息。

这一节将简单列出数据通信中常用的一些基本概念:

1. 数据通信系统

实质上就是一个计算机网络系统,由数据传输子系统和数据处理子系统组成。数据传输子系统包括:传输线路、调制解调器、多路复用器、交换器组成;数据处理子系统由计算机类设备组成,作为信源和信宿。

2. 信道

是信号传输的通道,通常是一种抽象描述,与传输介质相比,它侧重逻辑上的含义。按传输信号形式,它分成两种:一种是以数字脉冲形式传输数据的信道称数字信道;另一种是把数据先经过调制,变换为模拟的调制信号进行传输的信道称模拟信道;按传输介质类型,它分为:有线信道和无线信道;按使用方式,它分为公用信道和专用信道。

3. 通信线路连接方式

网络中的节点可按以下方式进行连接:

- 点对点连接:一条线路两端连接两个节点的通信方式。
- 分支连接:各个节点都接到一条公共通信线路上,节点之间连接时要通过控制转接站。
- 集线式连接:一群终端处设置一个线路汇集站,用以把终端线路集中起来,再用一条宽带高速线路传输出去。

4. 信道通信方式

根据信号在信道上的传输方向,可以把数据通信方式分为:

- 单工通信:只允许数据按指定的一个方向传输;
- 半双工通信:允许不同时间数据可按指定的不同方向传输;
- 双工通信:允许同时双向传输数据。

5. 数据传输方式

可以分成两种：

- 基带传输：是指把全部介质带宽分配给一个单独的信道，常用于数字信号原封不动的传输。

- 频带传输：是指把全部介质带宽分割成多个信道，每个信道都能传输一种不同的模拟信号。于是要把数字信号调制成音频信号后再发送和传输，到达信宿时再把音频信号解调成原来的数字信号。

6. 数据同步方式

可以分成两种：

- 异步传输：采用起址式传送数据，数据以字符为单位，而且字符发送时间是异步的，即后一字符与前一字符发送的时间无关。

- 同步传输：面向报文或分组为单位传送数据，每组字符必须有一个确定的同步字符领先。

7. 网络复用技术

是多个数据通信合用一条传输线路，以提高线路利用率的一种技术，基本的多路复用技术有：

- 频分多路复用：把传输线的总频带划分成若干个分频带，以提供多条数据传输信道。
- 同步时分多路复用：把多路器轮转一周的时间，分成若干时间片，每个终端都分别对应一个固定的时间片，依次分时共享一条公用传输线。

- 异步时分多路复用：信道的分配与终端有无信号和需求相结合，时间片不再固定划分，以改善多路复用性能。

- 码分多址复用技术：用地址码去调制发送的信号和利用码型正交性选择信号的复用技术。

8. 数据交换技术

通信网络中设置交换中心，用于连接大量终端，以同时对多对要通信的终端建立或拆除通信链路。数据交换技术有：

- 电路交换：发送端和接收端之间实际建立一条物理信道的交换方式，适用于传输模拟信号。

- 报文交换：报文是加上控制信息的待发送数据块，发送端以报文为单位进行发送，交换节点按报文存储转发，每次转发一个相邻的链路，直到目标节点收到报文为止，可用于直接传送数字信息。

- 报文分组交换：与报文交换的区别在于把大的报文数据分割成若干报文组，以报文

组为单位进行数据交换,是对报文交换的一种改进,将不定长的报文分解成定长的分组(packet)进行传输可以提高传输效率。

- 帧中继交换:是在X.25分组交换基础上,简化了差错控制等功能,而形成的一种交换技术。
- 异步传输模式(ATM):又称信元中继交换,以信元为传输单元,并把电路交换和分组交换的优点结合起来的一种数据交换技术。

9. 差错控制技术

指在通信过程中发现、检测、纠正数据传输过程中的错误。差错控制码分检错码和纠错码;差错控制方法分自动控制重发和向前纠错。

10. 信道容量、数据传输率和误码率

数据传输率:指单位时间内传送的二进位数据位数,常用“位/秒(b/s)”、“千位/秒(Kb/s)”或“兆位/秒(Mb/s)”作计量单位。

信道容量:信道允许的最大数据传输率。

误码率:指数据传输中出错数据占被传输数据总数的比例。

8.1.3 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,实现它要解决很多技术问题,如支持多种通信介质、支持不同机器平台、支持不同通信规范、支持多种应用业务等等。在实现计算机网络这种复杂系统时通常采用层次式结构,把一个计算机网络分成若干层,每层完成特定功能,各层协调起来构成整个网络系统。在网络分层结构中, N 层子系统是 $N-1$ 层子系统的使用者,同时也是 $N+1$ 层子系统的功能(服务)提供者。对于 N 层子系统来说, $N+1$ 层子系统直接使用的是 N 层子系统提供的功能,而实际上 $N+1$ 层子系统通过 N 层子系统提供的功能享用了 N 层子系统之内的所有层提供的功能。

采用层次结构的网络系统中,高层数系统仅仅利用了其较低层数子系统提供的功能,而不需了解其实现该功能时所采用的算法和协议等内部细节,反之,较低层数子系统也仅仅是使用从高层送来的参数,这就是层次间的独立性。这种独立性可使得一个层次中的老模块用新模块取代,而只要新老模块是有相同的功能和接口就行。从而,能使网络系统适应性强,便于维护。

所谓网络体系就是为了完成计算机间的通信协作,把计算机间互联的功能划分成有明确定义的层次,规定了同层次进程通信的协议及相邻层之间的接口和服务。计算机网络的同层进程通信协议以及邻层接口便称为网络的体系结构(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,每个层次又应提供哪些协议的精确定义。从上面的介绍可以看到,网络体系结构是从层次结构及功能上来描述计算机网络结构;并不涉及每一层硬件和软件的组成,更不涉及这些硬件和软件本身的实现问题。对于同样的网络体系结构,可采用不同方法设计出完全不同的硬件和软件,相应层次提供完全相同的功能和接口。

作为近代网络发展里程碑的美国 ARPA 网就采用分层方式实现的,确立了通信子网和资源子网两层层次结构,并研究了检错、纠错、路由选择、分组交换和流量控制等多种方法和协议,为网络体系结构的发展提供了宝贵经验。同时,在欧洲,支持商业应用的 X.25 网成为一种较为可行的技术。在 ARPA 之后,国外一些主要计算机生产厂商如 IBM、DEC 等都先后推出了本公司的网络体系结构,但都属专用性的。为了使不同计算机厂家生产的计算机能相互通信,以便建立更大范围的计算机网络,有必要建立一个国际范围的网络体系结构标准,1980 年 12 月国际标准化组织信息处理系统技术委员会下属的开放系统互连分委员会发表了第一个“开放系统互连参考模型(OSI/RM)”的建议书,1983 年春,正式批准为国际标准、即 ISO7492。

2. 通信协议

在计算机网络中,为了保证数据通信双方能正确、自动地进行通信,必须在关于信息传输顺序、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,从而,通信协议也被分为多个层次,每个层内还允许分成若干子层次,协议各层次有高低之分。另外,通信协议应该可靠、有效,否则会造成通信的混乱和中断。

网络通信协议含有三个要素:

(1) 语义 指对构成协议的协议元素含义的解释。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,接着为报文;而第二个协议元素表示正文开始,接着为正文等等。

(2) 语法 用于规定将若干个协议元素和数据结合起来表示一个完整内容时应遵循的格式。例如,传送一份数据报文时,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;最后为奇偶校验码。

(3) 规则 它规定了事件的执行顺序,即通信双方进行发收和应答的次序。

综合上述可见,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时所使用的一种语言。

3. 开放系统互联参考模型(OSI/RM)

为了实现计算机系统的互连,OSI 开放系统互连参考模型把整个网络的通信功能划分

成顺序的 7 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。每个层次完成各自的功能, 通过各层面的接口和功能的组合与其相邻层连接。

图 8-1 是 OSI/RM 的七个层次, 下面介绍七个层次和它们的功能:

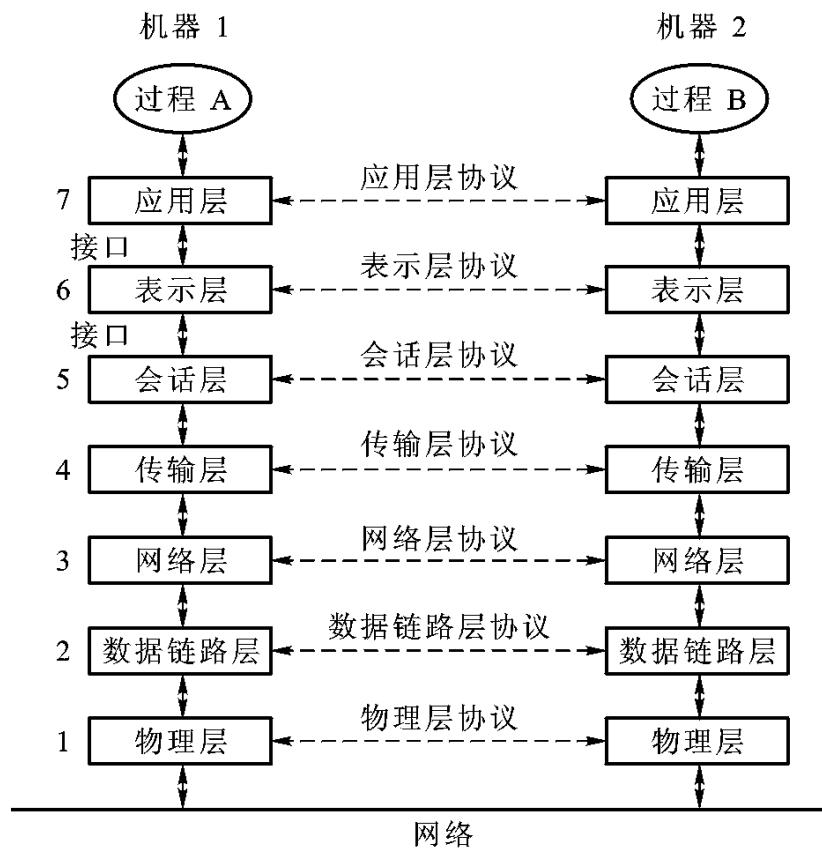


图 8-1 OSI 七层模型中的层和协议

(1) 物理层

物理层为通信提供物理链路, 实现数据链路实体间比特 (bit) 流的透明传输。透明的意思指经物理电路传送后的比特流保持原样, 不发生任何变化。因此, 任意的比特流都可以在物理电路上传输。物理层定义了与传输线路以及接口硬件的机械、电气、功能和规程有关的各种特性, 以便建立、维持和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功能, 可以认为这一层是一个硬件层。现在已经有很多物理层协议, 如 RS-232 是串行通信线的标准。

(2) 数据链路层

数据链路层负责建立、维持和释放数据链路的连接, 数据传输时的流量控制和差错控制。它能保证网络中两个相邻结点间无差错地传送以帧为单位的数据, 提供了校验和纠错机制, 以保证数据的正确传输。每一帧包括一定数量的数据和一些必要的控制信息, 帧是数据通信中的最小语义单位, 根据内容的不同可以分为: 数据帧、命令帧、响应帧等。在传送数据过程中, 借助循环冗余校验码信息, 若接收结点检测到所传数据中有差错, 就要通知发送

方重发这一帧,直到该帧正确无误地到达接收结点为止。在每一帧所包括的控制信息中,有帧头标志信息(同步信息)、地址信息、差错校验信息、帧尾信息,以及流量控制信息。这样,数据链路层就把一条有可能出差错的实际链路,转变为让网络层向下看起来像是一条不出差错的链路。

(3) 网络层

在计算机网络中进行通信的两个计算机之间可能要经过许多结点和链路,也可能要经过多个通信子网进行转发。在网络层,数据传送单位是分组或包,网络层的任务主要是选择合适的路由,使发送站的运输层所传下来的分组能够正确无误地按照地址找到目的站,并交付给目的站的运输层。此外,如果在网络中发送了过多的分组导致网络通路拥挤堵塞时,要进行拥塞控制。具体地说,网络层的功能为:

- 建立和拆除网络连接
- 路由选择和中继
- 网络连接多路复用
- 数据分段和组块
- 传输和流量控制
- 差错检测与恢复

目前广泛使用的网络层协议有:面向连接和面向非连接的。面向连接的协议称作 X·25,它需要在收发之间建立连接,网络从发送者到接收者之间选择一条路由,并在以后的传输中使用。面向非连接的协议称作 IP,一个 IP 信息不需要建立任何连接就可以发送,每个 IP 报文自己寻找路径到达目的地而独立于其他 IP 报文,它并不像 X·25 那样选择记忆内部路径。

(4) 传输层

在传输层,信息的传送单位是报文。当报文较长时,先把它分割成多个分组或包,并编排序列号,然后,再交给网络层(运输层的下一层)进行传输。传输层的主要任务是根据通信子网的特性最佳地利用网络资源,并以可靠和经济的方式,为两个端系统(发送站和接收站)的会话层建立一条传输连接,以透明地传送报文。传输层向上一层(会话层)提供一个可靠的端到端的服务,使会话层看不见传输层以下的数据通信的细节。具体来说,传输层包含以下功能:

- 传输(层)地址到网络地址的映射
- 多路复用与分割
- 传输连接的建立与释放
- 数据分段与重组装
- 组块与分块

可靠的传输连接可以建立在 X.25 或 IP 之上。前一种情况,报文组按一个正确顺序到达;后一种情况由于采用了不同的路由选择,可能后发送的报文组比先发送的早到,而由运输层负责把所有报文组重组,以保证报文组的正确次序。

(5) 会话层

会话层不参与具体的数据传输,它提供对话控制,对正在会话的双方进行跟踪及同步功能控制,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,对它们之间的数据交换进行管理,提供一个面向应用的连接服务。例如,确定是双工工作,还是半双工工作?通信的用户进程该哪一方发送信息?发生意外时,确定应从何处开始重新对话?会话的同步问题等。

(6) 表示层

表示层的任务是处理有关被传送数据的语法表示问题,它将数据从适合于某一用户的抽象语法转换为适合 OSI 系统内部使用的传送语法。由于不同的计算机系统常常采用不同的信息表示方法,在字符编码、数值表示等方面存在差异,如果不解决信息表示上的差别,通信的用户之间就不能互相识别。采用表示层后,用户就可集中注意力考虑通信的内容,不必顾及对方的一些特殊情况。此外,传输数据的加密、压缩和解密也是表示层的任务。

表示层的主要功能可小结如下:

- 数据语法转换
- 数据语法表示
- 为用户执行会话服务提供支持
- 管理多种数据结构集
- 数据加密和数据压缩

(7) 应用层

应用层是 OSI 的最高层,它为应用进程访问 OSI 环境提供了手段和窗口。它要确定应用进程之间通信的性质以满足用户的需要;它负责用户信息的语义表示,并在两个通信者之间进行语义匹配。应用层不仅要提供应用进程所需要的信息交换和远地操作,而且还要作为互相作用的应用进程的用户代理,来完成一些为进行语义上有意义的信息交换所必须的功能。在 OSI 的七个层次中,应用层最为复杂,所包含的应用层协议也最多,有些还正在研究之中,有代表性的应用层协议有:文件传递协议 FTP、存取和管理协议 TAC、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 和事务处理协议 TP 等。

采用 OSI/RM 七层模型的网络中,不同计算机上的应用进程在进行数据通信时,网络上的典型消息结构如图 8-2 所示,其信息流动过程如下:源计算机系统的用户进程把欲发送的数据传送至最高层,由该层在用户数据前面加上该层控制信息,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该层的控制信息,形成次高层的数据单元后,把它

传送至其下一层。以下各层如法炮制,信息按这种方式逐层地向下传送直至最低层,由于该层实现比特流传送,故不必再加控制信息。当比特流经过传输介质到达目标系统时,依次按由低向上次序逐层向上传送,且在每一层都依照相应的控制信息完成指定操作后,再把本层的控制信息去掉,把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,这样便结束了一次通信过程。

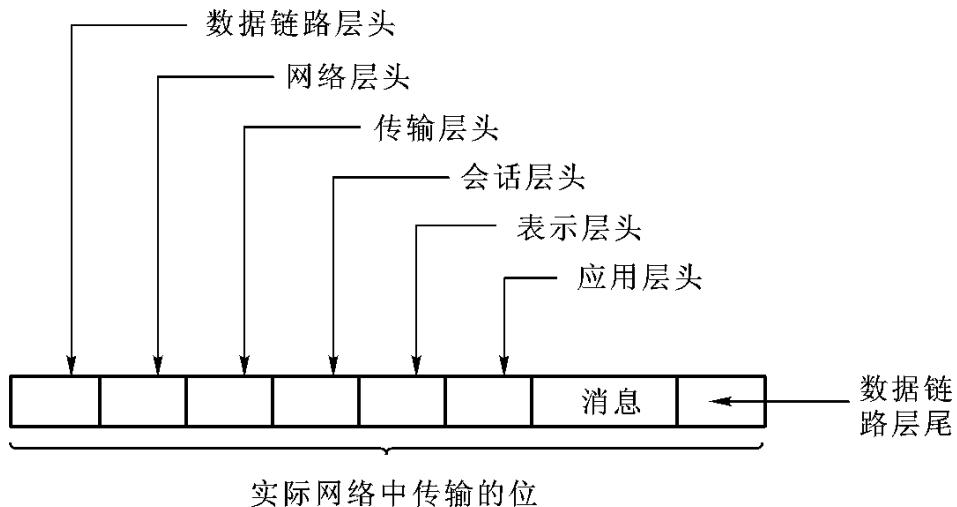


图 8-2 网络上的一个典型消息结构

4. TCP/IP 网络体系结构

除 OSI/RM 外,还流行着一些其他网络体系结构,如 IBM 公司的 SNA、CCITT 建议的公用数据网上的 X·25、Novell 公司的 IPX/ SPX (Internet work packet Exchange/ Sequenced packet Exchange) 网络体系结构、Microsoft 公司的 windows NT 网络体系结构等等。但最有名的是 Internet 网上使用的 TCP/IP,1983 年在 ARPA 网上开发了安装在 UNIX 的 BSD 版上的 TCP/IP 协议并非国际标准,也不如 OSI 那样理论上严谨完整。但由于它简洁、实用,已实际使用了多年,在计算机网络中占有非常重要的地位,成为事实上的工业标准。

传输控制协议 TCP(Transmission Control Protocol) 和网际协议 IP(Internet Protocol) 现已成为 Internet 网的主要通信协议。目前, TCP/IP 泛指以 TCP/IP 为基础的一个协议集,其中最著名的是运输层的 TCP 协议和网络层的 IP 协议,所以,人们常用 TCP/IP 表示 Internet 所使用的网络体系结构。它的主要特点是:适用于多种异构网络的互连、可靠的端—端协议、与操作系统紧密结合、效率高,并有较好的网络管理功能。

TCP/IP 技术是为了容纳物理网络技术的多样化而设计的。TCP 是一个完整的传输协议,位于网际层 IP 之上,提供给进程高可靠的通信能力。TCP 提供基于连接的数据流管道传输,它所采用的基本可靠性技术有:确认和超时重传、流量控制和拥挤控制等。IP 解决了

宽容性问题,由于各种网络技术的帧格式、地址格式等上层协议可见的因素差别很大,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协议的组成如图8-3所示。

TCP/IP与OSI有不少差别。它一共只有三个层次,顶层是各种应用层协议,它相当于OSI的最高三层。与OSI运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没有对更低层次作出规定,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,包含的常用协议如下:

- 简单邮件传送协议 SMTP(Simple Mail Transfer Protocol)。提供ASCII码电子邮件服务。
- 多用途邮件传送协议 MIME(Multipurpose Internet Mail Extensions)。提供非ASCII电子邮件传送服务
- 域名系统 DNS(Domain Name System)。提供主机到IP地址的转换服务
- 远程登录协议 Telnet(Telecommunication Network)。提供使用网络中远程主机的虚终端服务
- 文件传送协议 FTP(File Transfer Protocol)。提供网络内不同计算机之间的文件传送和操作服务
- 网络基本输入输出系统 NetBIOS(Network Basic Input Output System)。提供个人计算机通信服务

(2) 运输层 对应于OSI的运输层,它的主要协议有:

- TCP(Transmission Control Protocol)。提供基于连接的字节流传送服务。
- UDP(User Data gram protocol)。提供数据报传送服务。
- NVP(Network Voice Protocol)。提供声音传送服务。

(3) 网络层 对应于OSI的网络层,该层的重要协议有:

- IP(Internet protocol)。为传输层提供网际传输服务
- ICMP(Internet Control Message Protocol)。因特网控制报文协议,允许主机或路由器报告有关IP服务状况。
- ARP(Address Resolution Protocol)。把IP地址转换成网络物理地址。
- RARP(Reverse ARP)。将网络物理地址转换成IP地址。

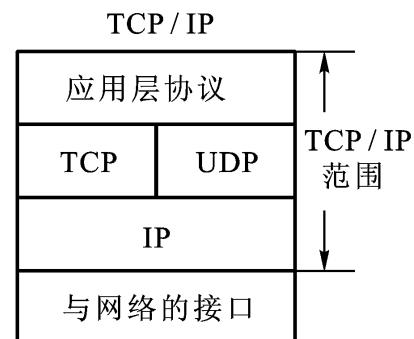


图8-3 TCP/IP协议的层次

8.2 网络操作系统

8.2.1 网络操作系统概述

计算机网络系统除了硬件,还需要有系统软件,两者结合构成计算机网络的基础平台。系统软件中,最重要的是操作系统,它管理硬件资源、控制程序执行、合理组织计算机的工作流程,为用户提供一个功能强大、使用方便、安全可靠的运行环境。

网络操作系统是网络用户和计算机网络之间的一个接口,它除了应该具备通常操作系统所应具备的基本功能外,还应该具有联网功能,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,支持有效可靠安全地数据传输。

早期网络操作系统功能较为简单,仅提供基本的数据通信、文件和打印服务等。随着网络的规模化和复杂化,现代网络的功能不断扩展,性能大幅度提高,很多网络操作系统把通信协议作为内置功能来实现,提供与局域网和广域网的连接。

一个典型的网络操作系统有以下特征:硬件独立性,网络操作系统可以运行在不同的网络硬件上,可以通过网桥或路由器与别的网络连接;多用户支持,应能同时支持多个用户对网络的访问,应对信息资源提供完全的安全和保护功能;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,如微软的 windows NT 网络操作系统可以支持包括:MS – DOS、OS/2、Windows 98、Windows for wrokgroup、UNIX 等多种客户端,极大地方便了网络用户的使用;提供目录服务,以单一逻辑的方式让用户访问可能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、WWW 服务等等;可操作性,这是网络工业的一种趋势,允许多种操作系统和厂商的产品共享相同的网络电缆系统,且彼此可以连通访问。

网络操作系统可分成三种类型:

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,系统的基本单元是一台主机和若干台与主机相连的终端构成,把多台主机连接起来就形成了网络,而信息的处理和控制都是集中的,UNIX 系统是这类系统的典型例子。

2. 客户/服务器模式

这是现代网络的流行模式,网络中连接许多台计算机,其中,一部分计算机称服务器,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。而另外一些计算

机称客户机,它向服务器请求服务,如文件下载和信息打印等。服务器通常配置高,运算能力强,有时还需要专职网络管理员维护。客户机与集中式网络中的终端不同的是,客户机有独立处理和计算能力,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,又可叫分布式处理系统,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形结构,以服务器为中心,与各客户间采用点到点通信方式,各客户间不能直接通信。当今两种主要客户服务器模式为:文件服务器 C/S 模式和数据库服务器 C/S 模式。无论哪一种模式,客户在请求服务器服务时,双方要通过多次交互: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。客户服务器模式的主要优点是: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和服务器两种功能,既可以向其他机器提供服务,又可以向其他机器请求服务,而网络中没有中央控制手段。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式计算模式。Netware Lite 和 Windows for workgroup 是这类网络操作系统的代表。对等模式的主要优点是:平等性、可靠性和可扩展性较好。

8.2.2 几个流行的网络操作系统

网络操作系统有许多产品,一般认为,在高端关键应用场合,以 UNIX 系统为主;在中低端则以 Windows NT 与 Netware 为主,Netware 市场份额较大,但 Windwos NT 发展迅猛前景看好。

1. VINES

VINES 是由 Banyan System 公司开发的网络操作系统产品,是当今企业连网的主导产品之一。最早运行于专用服务器上,现在已推出各种 PC 机的版本。VINES 基于 UNIX system V,系统由工作站和服务器两个模块组成。它的物理层和链路层能接纳许多协议,如 802.X、X.25、HDLC 等,网络层支持 IP、ARP、ICMP 和专有的 VINES 协议 RTP 及 ICP,传输层支持 TCP、UDP 和专有的 VINES 协议 IPC 及 SPP 等。它能对 LAN 和 WAN 提供强有力的支持,主要特点是:安装容易、管理简单;采用 street talk 全局命名服务,故能方便转移资源达到负载平衡;支持多任务、多用户;支持 SMP。

2. Windows NT

Windows NT 是目前被认为最有前途的网络操作系统之一,功能强、性能高,内含软件丰富、互操作性好,具有友好的用户界面。Windows NT 具有抢先式多任务、多线程调度能力,并可支持文件、打印、信息传输与应用服务的多用途 32 位网络操作系统。此外,还具有:支

持 SMP、可在多种服务器平台上运行、先进的容错功能、信息的安全性保证等特点。在 Windows NT 上运行 Microsoft 的 SQL Server 是客户/服务器数据库应用系统的最佳方式。因为,作为同一个公司的产品,SQL Server 能直接利用 NT 的多线程能力,降低了开发成本。

3. Netware

Novell 公司的 Netware 网络操作系统是一个基于客户机/服务器模式的多任务操作系统。Netware 软件分两部分:Netware shell 和文件服务器,分别安装在客户端和服务器上。最新的 Netware 5.0 版增强了网络安全性能、增加网络目录服务、增加接入用户数目、增加和 Internet 连接功能。其主要特点为:(1)开放式网络环境:支持多种通信协议、多种设备驱动程序,构成异构的计算机网络;(2)高性能文件系统:支持 4GB – 32TB 的文件空间,支持 DOS、OS/2、MAC 及 UNIX 文件系统;(3)强化的安全机制:包括登录安全性、访问授权、文件操作权限、目录使用权限;(4)容错技术:如采用磁盘镜像和磁盘复制等;(5)对 Internet 的支持丰富。

4. OS/2 Warp Server

IBM 公司开发的多用途网络操作系统,最早开发的是单机操作系统。1995 年,OS/2 演变为网件 Warp connect3.0,而在 1996 年,发布了 Warp Server4.0,接着又推出了 SMP 版本。综合的 TCP/IP 工具包括 Internet 所有的所有设施,如含有 DHCP 服务器和 DNS 服务器。近期又推出了 Java 的运行时服务,IBM 计划发布综合的全功能 HTTP 服务器。

8.2.3 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。Microsoft 的网络基于对等结构,可以与其他类型网络服务器间进行通信,包括支持微软网络、Netware 网络、基于 TCP/IP 的 UNIX 环境。Windows NT 网络操作系统有以下特点:

- 网络功能做在操作系统中,它的网络驱动程序是执行体中 I/O 子系统的一个组成部分;
- 支持远程打印、电子邮件、文件传输等功能,而不需要用户在机器上再安装任何网络服务器软件;
- 支持多种网络协议,目前流行的多种网络、网络服务器和网络驱动程序,如 Novell、Banyan、Sun NFS 和 VINES 能方便地与 NT 系统进行数据交换;
- 内装网络采用开放式结构,像重定向程序、传输驱动程序和服务器都可被动态装入和卸出,且很多不同的这种部件可以并存。

下面介绍 Windows NT 有关的一些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,它具有集中的安全控制。对域资源(如打印机、目录等)的所有访问都由该域中的某个计算机授权监控,这台计算机称为主域控制

器。使用域的优点是：一是一个域对一个用户来说只有一个密码，此密码可以打开该域的所有被授权使用的资源；二是密码可由用户设定。

每个用户都有一个帐号及其用户信息的数据库记录，权限是允许对网络资源访问的权利（只读、执行、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帐号、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帐号信息。在包含多台服务器的域中，其中有一台是主域控制器，它验证用户对域的登录，维护主安全数据库。域的最小配置为由一台计算机，既运行 NT 服务器，同时又充当域控制器。

工作组（workgroup）是若干合法用户的组合，NT 管理员可把组作为一个整体加以管理，提高管理效率和安全性。一台网络中的计算机要么是工作站（使用但不提供资源），要么是服务器/工作站（既使用又提供资源），而不存在专用的服务器。组可给予执行任务所需的用户数据权限及资源访问权限，工作组的安全性是一种共享级的安全控制，所有访问者只能用一个密码来访问该资源。NT 服务器还使用局部和全局组来简化管理，每个组都有自己的成员和资源访问权限。

域用户管理器是 NT 服务器中提供网络安全的主要程序之一，它负责分配系统级的权力或定义网络采用的监查规则。允许网络管理员完成以下工作：

- 创建、修改、删除域中的用户帐号，控制用户帐号的密码特性
- 决定用户或组的系统权力
- 定义用户环境和网络信息
- 为用户帐号分配登记信息
- 管理域中的组和组中各帐号间的成员关系
- 管理网络中不同域之间的信任关系
- 管理域的安全规则，审核和定义记录的安全事件的类型

2. NT 的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，可以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加以处理。此外，资源管理器还为网络管理员提供目录共享的功能，用户可以通过网络连接到这些共享目录上，并对其上的信息进行访问。此外，资源管理器还要完成对驱动器和目录共享、共享特性的修改、共享目录和文件的访问等管理工作。

NT 服务器使用 NTFS 来提供可靠性和对共享文件的安全支持，NTFS 提供了多种存取权限，用来加强文件系统的安全管理，本地用户也可以设置文件访问权限来限制用户的访问。

3. NT 的文件和目录复制功能

文件和目录复制是 NT 提供的服务之一, 用于在一个或多重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, 主目录保存在一个指定 NT 服务器上, 而主目录上的文件更新会被复制到其他指定计算机上。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. NT 网络服务

NT 提供了强大的网络功能和广泛的 Internet/Intranet 支持, 能完成以下功能:

- DNS 域名服务 把主机 IP 地址解析为给定网络名称的数据库服务, 微软已将 DNS 服务设计成与 Windows Internet 名称服务协同工作以解析主机名称。
- Internet 信息服务 微软将新的 Internet 信息服务器 IIS 的集成版本链到了 NT 的服务器, 该 IIS 作为 NT 安装的可配置选项提供 Internet 访问。
- 多协议路由 NT 能用作连接两个局域网的路由器, 新的多协议路由服务能通过 IP 或 IPX 协议把一个 LAN 连到 WAN 上。
- 动态主机配置协议中继代理 提供集中的、动态的 IP 地址和安全、可靠的 TCP/IP 网络配置。
- NT 浏览器 NT 服务器提供计算机浏览服务, 它用列表方法列出网络资源, 可以指定网络中特定计算机作为浏览器, 而浏览器上集中了所有网络服务器, 从而, 避免了在所有计算机上都要保存一份共享网络资源表, 也减少了建立和维护网络共享资源的网络传输工作量。
- 点对点通道协议及远程访问服务 通过点对点通道协议 (PPTP) 允许远程访问。客户机通过 Internet 与 Windows NT RAS 的连接而访问 WAN 上的资源。此外, NT 也提供拨号登录, 这样, 登录到 RAS 连接的 Windows NT 工作站的用户能利用域控制器连接并引导加密登录认证。
- 分布式组件对象模型 DCOM NT 已将组件对象 COM 的能力从本地应用扩展到包括组网与 Internet 的应用程序。

5. NT 网络环境管理

提供两种网络管理工具: 性能监视器和网络监视器, 以监视服务器的各种参数。

8.3 分布式操作系统

8.3.1 分布式系统概述

分布式计算机系统是由一组松散的计算机系统,经互连网络连接而成的“单计算机系统映像”(Single Computer System Image)。它与计算机网络系统的基础都是网络技术,在计算机硬件连接、系统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。分布式系统与网络系统的主要区别在于:网络系统中,用户在通信或资源共享时必须知道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,可把一个大型程序分布在多台计算机上并行运行。通常,分布式计算机系统满足以下条件:

- 系统中任意两台计算机可以通过系统的安全通信机制来交换信息。
- 系统中的资源为所有用户共享,用户只要考虑系统中是否有所需资源,而无需考虑资源在哪台计算机上,即为用户提供对资源的透明访问。
- 系统中的若干台机器可以互相协作来完成同一个任务,换句话说,一个程序可以分布于几台计算机上并行运行。一般的网络是不满足这个条件的,所以,分布式系统是一种特殊的计算机网络。
- 系统中的一个结点出错不影响其他结点运行,即具有较好的容错性和健壮性。

分布式计算机系统要让用户使用起来像一个“单计算机系统”,那么,如何实现“单计算机系统映像”?实现分布式系统以达到这一目标的技术称透明性,透明的概念适用于分布式的系统的各个方面:

- 位置透明性 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在的位置,资源的名字中也不应包含位置信息;
- 迁移透明性 指资源无需更名就可以从一个结点自由地迁移到另一个结点;
- 复制透明性 指系统可任意地复制文件或资源的多个拷贝,而用户都不得而知;
- 并发透明性 指用户不必也不会知道系统中同时还存在其他许多用户与其竞争使用某个资源;
- 并行透明性 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)自动找出潜在并行模块去分布并执行,而不为用户所察觉,这是分布式系统追求的最高目标。

用于管理分布式计算机系统的操作系统称为分布式操作系统,分布式操作系统

(Distributed Operating System)应该具备四项基本功能：

(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可以通过通信来交换数据。

(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) 并行运算：提供某种程序设计语言，使用户可编写分布式程序，该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，亦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主要优点是：坚定性强、扩充容易、可靠性好、维护方便和效率较高。

为了实现分布式系统的透明性，分布式操作系统至少应具有以下特征：一是有一个单一全局性进程通信机制，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；二是有一个单一全局性进程管理和安全保护机制，进程的创建、执行和撤销以及保护方式不因机器不同而有所变化；三是有一个单一全局性的文件系统，用户存取文件和在单机上没有两样。

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，分布性源于应用的需求，一个应用可以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，因为机器的分散而必须通过通信来实现进程的交互、合作和资源共享，可见分布式系统中通信机制是十分重要的。集中式操作系统中的通信方式，在分布系统中大多已不适用。

目前分布式系统中的进程通信可以分成三种：一是消息传递机制，它类似于单机系统中发送消息（信件）和接收消息（信件）操作；二是远程过程调用 RPC（Remote Procedure）；三是套接字 socket。而这三种通信机制都需依赖于网络的数据传输功能。

1. 消息传递机制

在分布式系统中，进程之间的通信可通过分布式操作系统提供的通信原语完成，最简单的分布式消息传递模型称客户机/服务器模型。一个客户进程请求服务，如读入数据、打印文件，向服务器进程发送一个包含请求的消息，这种消息按通信协议的规定来传递，服务器进程完成请求，作出回答和返回结果。最简单的消息传递模型中，只需两个通信原语：发送和接收。发送原语 send 说明一个目标和消息内容，接收原语 receive 说明消息来源和为消息存储提供一个缓冲区。在设计分布式消息传送机制时，需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

客户为了发送消息给服务器,首先必须知道服务器的地址,这样消息才能到达该机器;其次如果目标机器上有多个进程在运行,那么,消息到达机器后该传递给哪一个进程来处理呢?这就提出了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发送到目标机器上,而进程号用来决定消息应传递给哪个进程。每台机器上的进程编号都可从0开始,不需要互相协调,因为,进程*i*在机器1与机器2上不会发生混淆。BSD UNIX对此作了微小改变,用“机器号和进程本地标识”来代替“机器号和进程号”。进程本地标识是一个16位或32位随机数,一个进程提供一种服务,开始可通过系统调用告诉操作系统内核,它想监听本地进程标识,此后,当一个“机器号和进程本地标识”编址的消息进入时,内核知道消息应该传递给哪个进程。这种方法的缺点是位置不透明性,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。让分布式系统中的每个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,二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,包含目的进程的地址。由于网络中的所有机器都能收到,由内核检查这个进程是否在本机上,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它这个地址,避免下次再广播。这种方法虽然具有位置透明性,但缺点是广播通信给系统增加了较大开销。

第三种方法采用名字服务器寻址法。系统中增加一台机器,提供高层的机器名和机器地址的映射,称作名字服务器。客户机中存放ASCⅡ码形式的服务器名字,每次客户机运行,发送消息给一台服务器时,先发出一个请求消息给名字服务器,询问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了。这种方法的缺点是要增加一台服务器,增加了发送的消息量。

(2) 同步和异步通信原语

分布式进程通信原语可以分成同步和异步两种,通信原语的基本形式为:

- Send(P, Message)
- Receive(Q, Buffer)

其中,P表示接收进程,Q表示要发送进程,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,实际上进程标识要由计算机地址和进程标识符组成。

同步发送时,发送进程要求接收进程做好接收消息的准备,甚至要明确知道接收方已经做好接收消息的准备,而且发送过程在发送完消息后阻塞等待接收方的回答。如果在预定时间内未收到回答,则认为消息已被丢失,应再次重发。同步发送消息的缺点是系统的并行性差,而且无法利用广播消息的功能,而广播功能在很多场合下是很有用的。

分布式系统中消息的发送过程如下:在客户机上的一个进程请求服务器上的某个进程

提供服务。首先,它组织并形成一个消息,其中包含了接收进程标识和欲发送的消息内容。然后,调用分布式操作系统提供的发送原语 `send`,发送的进程被阻塞,直到消息传送完毕后发送进程才能继续执行。执行该原语进入了客户机的消息收发模块,它将信件按网络通信协议的层次自高而下逐层传送,当信件通过网络物理链路传送到目标服务器时,又经过通信协议逐层上传直到接收方的消息收发模块。该模块检查消息中的接收进程标识,把消息投入接收进程的信箱或缓冲区中,然后,发一个信号给接收进程或直接唤醒它进行处理。同样,接收进程调用 `receive` 时,被阻塞直到一条消息收到并放入信箱或缓冲区时才能返回控制权。

异步发送时,发送进程把消息发送出去后,并不阻塞自己并等待对方回答,而是继续执行下去。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。非阻塞通信原语有一个主要的缺点,在消息被发送出去以前,发送进程不能修改发送的消息缓冲区,它不知道传送何时进行,因而,无法知道何时可重新使用缓冲区。有两个办法可以解决它:一是由发送原语把消息拷贝到内部缓冲区,发送进程一旦获得控制权,允许重新自由使用缓冲区。二是当消息被发送出去后,中断发送进程并通知它缓冲区可用。

(3) 缓冲和非缓冲原语

非缓冲原语不使用缓冲区,而是有一个指向特定进程的地址。一个 `receive(addr, &m)` 调用告诉内核,该调用过程正在地址 `addr` 上监听,并准备接收一个发送到该地址上的消息, `m` 指向的消息接收区将保存消息。当消息到达时,接收方内核把它拷贝到消息接收区,然后释放被阻塞的接收进程。

若服务器方先调用 `receive`,客户端后调用 `send`,这种策略会工作得很好。对 `receive` 的调用是告诉服务器的内核,应该使用哪个地址,到达的消息放在哪里。但当 `send` 在 `receive` 之前调用时,对于新来的消息,服务器内核如何知道它的哪个进程正在使用该地址?消息又被拷到哪里呢?答案是不知道。一种简单的策略是:忽略该消息,等到客户端超时,这样服务器就可以在客户重传消息前调用 `receive`。此法虽容易实现,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。

第二种办法是让接收方内核保留每个到达的消息一段时间,直到相应的 `receive` 被调用。每到达一个消息就启动一个计时器,若在相应的 `receive` 被调用前计时器超时,则该消息被丢弃。理论上,保留消息一段时间可定义一个新的数据结构——信箱,准备接收消息的进程请求内核创建一个信箱,并指明一个地址以便查找网络信息包,以该地址标识的消息都放入该信箱中。对 `receive` 的调用只是从信箱中读出一个消息,若没有消息可读则阻塞进程。采用这一技术的原语称缓冲原语。采用这种方法要考虑信箱大小,小了会溢出,大了造成浪费。

在某些系统中,还有一种选择。若目标机上没有空间存放,则不让进程发送消息,让发

送者阻塞,直到消息能被对方接收。

(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。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。有两种方法解决这个问题:第一种方法是尽可能保证传送,使用一个可靠的传输协议,能附带完成检错、确认、重传、重排序,尽可能使消息正确传送。但系统无法保证消息成功发送,完成可靠的通信依赖于用户。第二种方法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 RPC 是目前在分布式系统中被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的方式进行交互,但这个过程调用是用来访问远程计算机上提供的服务的,看上去用户却感觉像在执行本地过程调用一样。

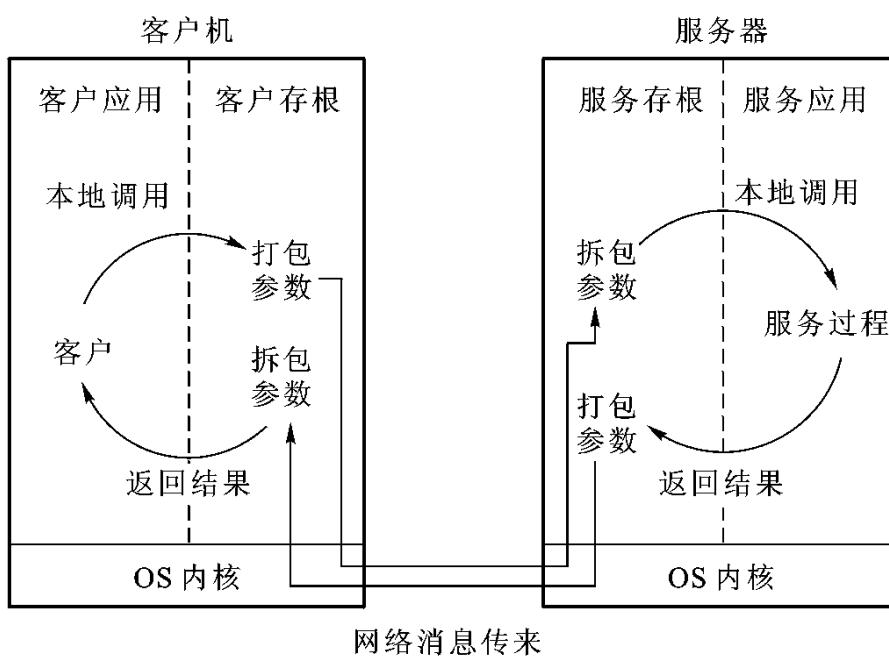


图 8-4 远程过程调用机制

下面我们来讨论远程过程调用的基本原理。在客户/服务器方式下,为了能以相同的方式完成本地过程调用和远程过程调用,可以在客户/服务器上各增设一个客户存根(stub)和服务器存根,它们通常放入程序库中。当客户机上的某进程需要调用服务器上的一个过程时,可以发出一条带有参数的 RPC 命令给客户存根,委托它作为自己的代理。客户存根收到 RPC 命令后,便去执行本次的远程过程调用,它把参数打成消息包,执行 send 原语请求内核把消息发送到服务器去,在发送消息执行后,客户存根调用 receive 原语阻塞自己直到服

务器发来应答。当消息到达服务器后,内核把消息传送给服务器存根,通常,服务器存根会调用 receive 原语阻塞自己等待消息到达。服务器存根拆包从消息中取出参数,然后以一般方式调用服务器进程,该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,服务器存根获得控制权,它把结果打成消息包,再调用 send 原语请求内核把消息发回给调用者。整过过程调用结束后,服务器存根回到 receive 状态,等待下一条消息。消息送回客户机后,内核找到并把消息送给客户存根,客户存根检查并拆开消息包,把取出的结果返回给调用进程,调用进程获得控制权并得到了本次过程调用的结果。图 8-4 说明了远程过程调用的流程。可以把 RPC 执行步骤总结如下:

- 客户进程以普通方式调用客户存根
- 客户存根组织 RPC 消息并执行 Send, 激活内核程序
- 内核把消息通过网络发送到远地内核
- 远地内核把消息送到服务器存根
- 服务器存根取出消息中参数后调用服务器过程
- 服务器过程执行完后把结果返回至服务器存根
- 服务器存根进程将它打包并激活内核程序
- 服务器内核把消息通过网络发送至客户机内核
- 客户内核把消息交给客户存根
- 客户存根从消息中取出结果返回给客户进程
- 客户进程获得控制权并得到了过程调用的结果

通过上述步骤可以将客户进程的本地调用转化为客户存根,再转化为服务器上的本地过程调用。RPC 的实现一般采用图 8-5 所示的通用结构,客户机执行 theClient 进程,它包括客户端应用的代码、clientstub 和传送机制;服务器端有 rpcServer 进程,它包括传送机制、server stub。

上述 RPC 仅适用于同构形分布式系统,即在同种类型计算机上,运行相同的操作系统,采用同一种程序设计语言,那么,对参数表示的要求没有什么问题。反之,对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 设施中增加一种参数表示的转换机制。例如,对于像整数、浮点数、字符和字符串可以提供一个标准格式,这样,任何机器上的本地数据和标准表示只要作相应转换就可以了。比如,用 ASCII 表示字符串、0 和 1 表示布尔数、补码表示整数等等。

3. 套接字(socket)

套接字(socket)是在 UNIX BSD 中首创的网络通信机制。socket 类似于电话扦座,通电话的双方好比相互通信的两个进程,区号是它的网络地址。区内电话交换局的交换机相当于一台主机,交换机分配给每个用户的局内号码相当于 socket 号。任一用户在通话前需占用

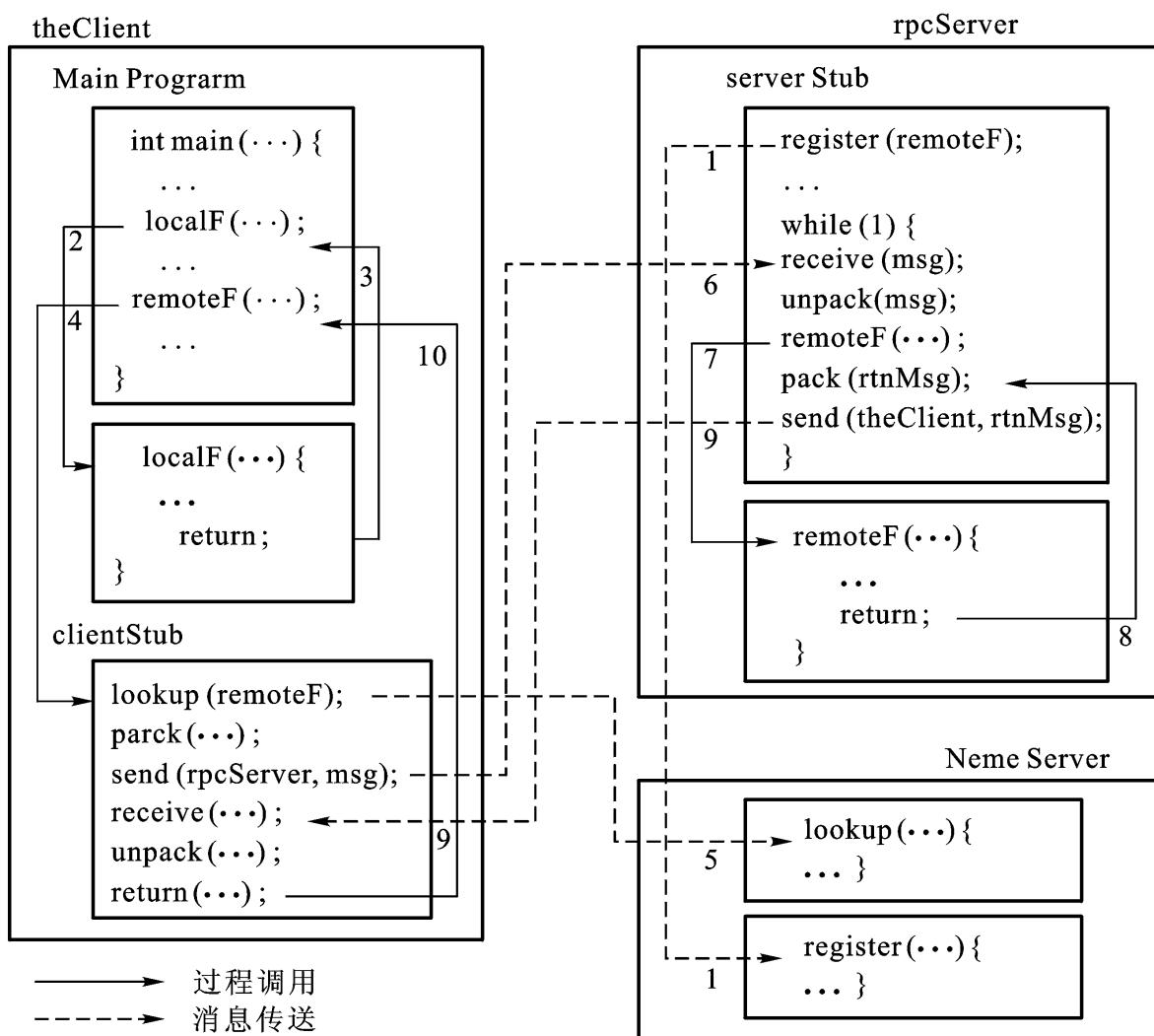


图 8-5 远程过程调用的实现

一部电话机,相当于申请一个 socket。用户通话前应知道对方的号码,相当于对方有一个固定的 socket,然后,向对方拨号呼叫,相当于发出连接请求,若连接成功就可以通话了。双方通话的过程是一方电话机发出信号而对方电话机接收信号,相当于向 socket 发送数据和从 socket 接收数据。通话结束后,一方挂起电话相当于关闭 socket,撤销连接。

Socket 实质上提供了进程通信的端口,进程通信前,双方必须各自创建一个端口。socket 是面向 C/S 模式设计的,针对客户机/服务器提供不同的 socket 系统调用。客户随机申请一个 socket,系统分给它一个 socket 号,而服务器拥有全局公认的 socket,任何客户都可向它发出连接与信息请求。socket 机制利用 C/S 模式巧妙地解决了进程之间的通信问题。

每个套接字支持一种特定类型的网络,在创建 socket 时需指定该类型,常见的类型有:可靠的面向连接的字节流、可靠的面向连接的包流、不可靠的包传递等。创建 socket 时,用一个参数指明所用的协议,对可靠的字节流和包流,常用 TCP/IP;对不可靠的包传递,则采用 UDP。

Socket 的功能由 socket 系统调用来体现,主要有:创建 socket(),指定本地地址 bind(),建立 socket 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 read()、readv()、recvfrom()、recvmsg()。

套接字 socket 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 C/S 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,图 8-6 和图 8-7 分别表示无连接协议和面向连接协议服务的调用时序。服务器如何工作还可分成两种类型:重复服务器和并发服务器。前者面向短时间能处理完的客户请求,由服务器自己处理来到的请求;后者面向长时间才能处理完的客户请求,每接到一个这样的客户请求, fork 一个子进程响应它,自己退回等待状态,监听新的客户请求,形成了一个主服务器和多个从服务器并发工作的局面。本章实例研究中将介绍 Windows 2000/XP 套接字。

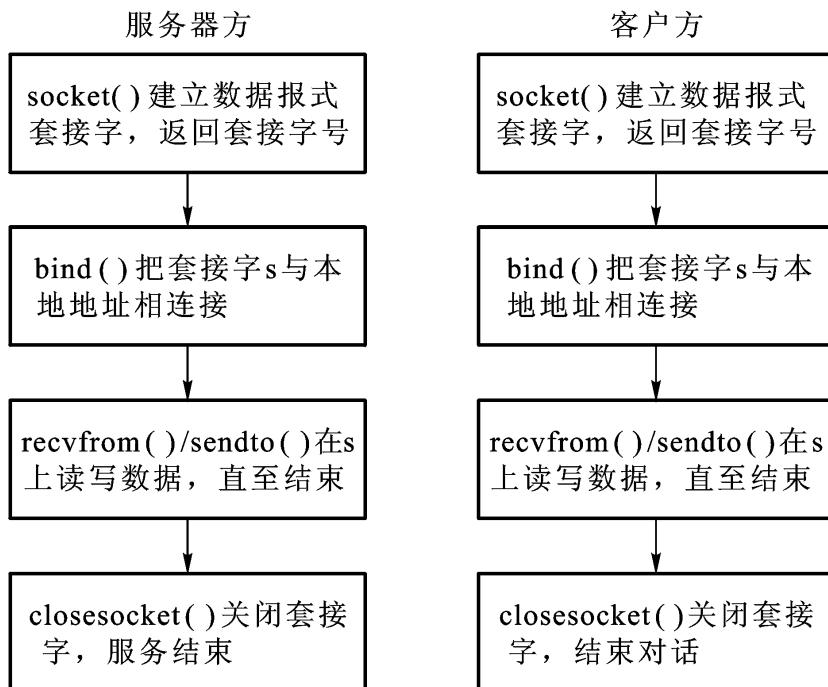


图 8-6 无连接协议的套接字调用时序

8.3.3 分布式资源管理

资源的管理和调度是操作系统的一项主要任务。单机操作系统往往采用一类资源由一个资源管理者来管的集中式管理方式。例如,所有主存空间都由存储管理负责分配、去配;所有行打印机由打印机管理负责打印事务等等。在分布式计算机系统中,由于系统的资源分布在各台计算机上,若一类资源归一个管理者来管的办法会使性能很差。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,那么,不论谁申请存储资源,即使申请的是自己计算机上的资源,都必须发信给存储管理者,这就大大增加了系统开销。如

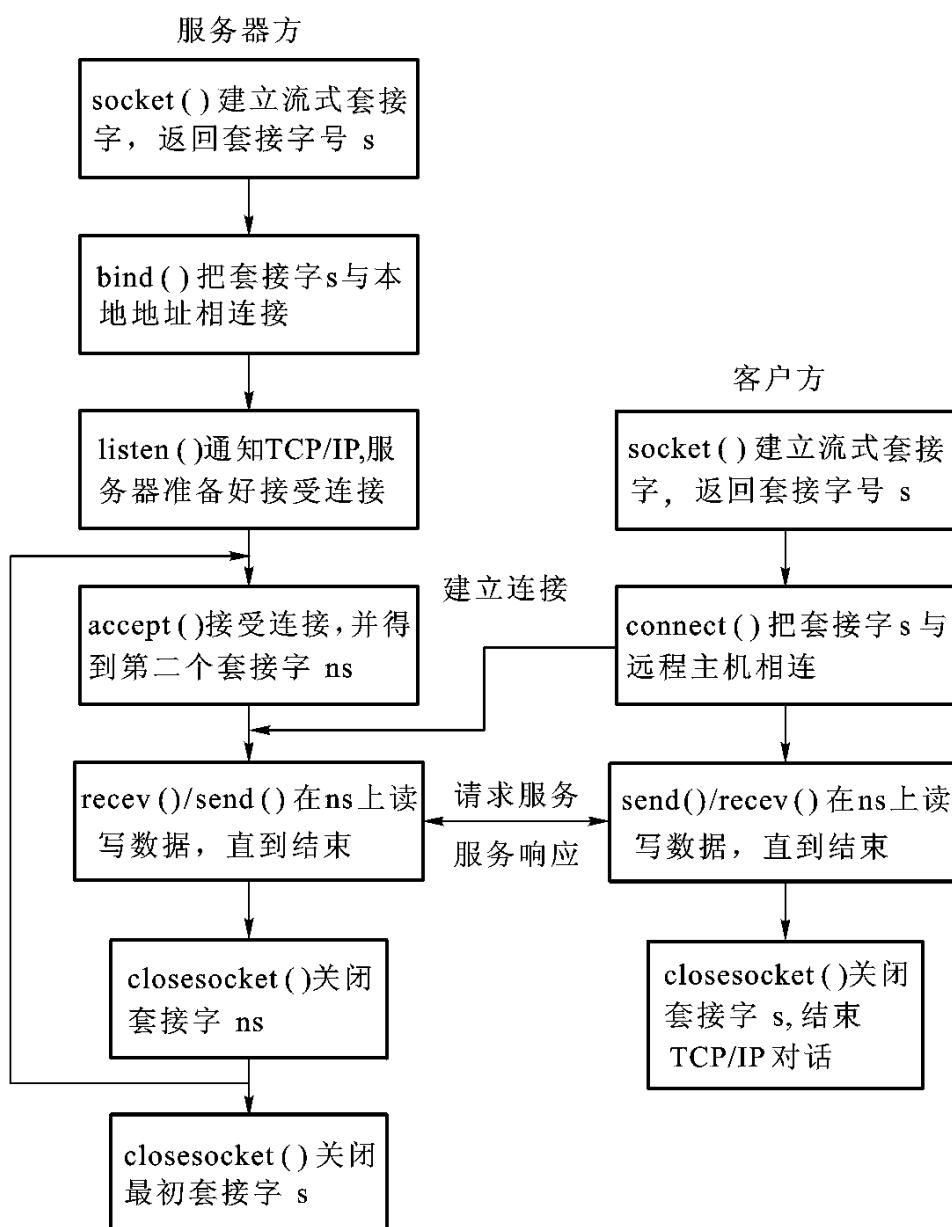


图 8-7 面向连接的套接字调用时序

果存储管理所在那台计算机坏了，系统便会瘫痪。由此可见，分布式操作系统如采用集中式来管理资源，不仅开销大，而且稳定性差。

通常，分布式操作系统采用一类资源多个管理者的方式，可以分成两种：集中分布管理和完全分布管理。它们的主要区别在于：前者对所管资源拥有完全控制权，对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有部分控制权，不仅一类资源存在多个管理者，而且该类中每个资源都由多个管理者共同控制。

集中分布管理中，让一类资源有多个管理者，但每个具体资源仅有一个管理者对其负责，比如，上面提到的文件管理，尽管系统有多个文件管理，但每个文件只依属于一个文件管理者。也就是说，在集中分布管理下，使用某个文件必须也仅须通过其相应的某个文件管理

者。而完全分布管理却不是这样,假如一个文件有若干副本,则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,其他各份副本应禁止使用。因此,当一个文件管理者接收到一个使用文件的申请时,它只有在和管理该文件其他副本的管理者协商后,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。

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,但该类中的一个资源却由惟一的一个管理者来管。当一个资源管理者不能满足一个申请者的请求时,它应当帮助用户去向其他资源管理者申请资源。这样用户申请资源的过程类似在单机操作系统上一样,只要向本机的资源管理者提出申请,他无须知道系统中有多少个资源管理者,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转来的申请的功能。由于资源管理者分布在不同计算机上,系统必须制定一个资源搜索算法,使得资源管理者按此算法帮助用户找到所需资源。设计分布式资源搜索算法应尽量满足以下条件:效率高、开销小、避免饿死、资源使用均衡、具有坚定性。常用的分布式资源搜索算法有三种:

1. 投标算法 该算法的主要规则如下:

- 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,向网络中位于其他结点的每个资源管理者发招标消息。
- 当一个资源管理者接到招标消息时,如果该结点上有所需资源,则根据一定的策略计算出“标数”,然后发一个投标消息给申请者,否则回一个拒绝消息。
- 当申请者收到所有回答消息后,根据一定策略选出一个投标人,并向它发一个申请消息。
- 接到申请消息后,将申请者的名字登记入册,并在可以分配资源时发消息通知申请者。
- 当资源使用完毕后,向分配资源的资源管理者归还资源。

投标与招标的策略可视具体情况而定,如可以用排队等待申请者的个数、投标与招标者距离的远近等作标数来投标,选择标数最小的投标人中标。美国加州大学欧文分校设计的DCS分布式计算机系统便采用了投标算法。

2. 由近及远算法 该算法让资源申请者由近及远地搜索,直到线上具有所要资源的结点为止。算法的规则如下:

- 申请者向它的某个邻结点发一个搜索消息,信中附上对资源的需求及参数 P,其值为申请者编号。
- 接搜索消息后,将发来消息的结点编号和信中参数 p 登记下来,前者定义为它的上邻结点,后者定义为它的前结点。如果接搜索消息的结点具有消息中所要求的资源,那么,它就向它的上邻结点发一个成功消息,并将自己的编号附上;否则它先发一个消息给它的前

结点告知自己是它的后结点。然后,发消息给上邻结点,请继续搜索,消息中带上参数 p,其值为自己的编号。

- 接继续搜索消息后,如果还有未被搜索的下邻结点,那么,就发搜索消息给它,消息中附上的参数 p 是从继续搜索消息中取得的。如果所有下邻结点都已搜索过,但它有后结点,则把继续搜索消息转给它的后结点。如果既没有未被搜索的下邻结点,又没有后结点,则说明全部结点已被搜索过,这时它将向上邻结点发一个失败消息。

- 接成功消息或失败消息后,若接消息者非申请者,则将消息转发给它的上邻结点,否则搜索就此结束。申请者或获得最近能提供所要资源的结点地址,或被告之系统中没有这样的资源。

- 如果一个已被搜索过的结点又收到搜索消息,则将原消息退回,发搜索消息的结点就认为该下邻结点不存在。

3. 回声算法 该算法能用来获得全局知识,也可用于搜索资源,算法的规则如下:

- 资源申请者向它的每一个邻结点发探查消息,消息中附上对资源的需求。

- 若接探查消息的结点是第一次接到这样的探查消息,它就把传来探查消息的邻结点定义为它的对该探查而言的上邻结点,而把其余的邻结点定义为它的下邻结点。若接探查消息的结点不是第一次接到这样的探查消息,它就向传来探查消息的邻结点发一回声消息,消息中参数值为 0。

- 接上邻结点传来的探查消息后,若有下邻结点,则将探查消息复制后分发给各下邻结点,否则向上邻结点发一回声消息,消息中参数 S(称资源参数)取下列值:

$$S = 0 \quad \text{当结点不具备所需资源时}$$

$$S = w * a + 1 \quad \text{当有 } a \text{ 个申请者在等待资源时}$$

式中 w 是一个常数。

- 当一个结点接到它的所有下邻结点发来的回声消息后,它就向它的上邻结点发一回声消息,消息中附上参数 S 及与之对应的结点编号。参数 S 取下列值:

$$S = 0 \quad \text{若 } S_r = 0 \text{ 且所有回声消息中所附参数均为 0}$$

$$S = \min(S_{r1} + 1, \dots, S_{re} + 1, S_r) \quad \text{否则}$$

式中, S_r 为本结点的资源参数; S_{r1}, \dots, S_{re} 为所有回声中所附的非零资源参数。若 S 值被选为 $S_{re} + 1$, 那么回声消息中所附结点编号就是附有资源参数 S_{re} 的回声中所附的结点编号。若 S 值被选为 S_r , 则回声消息中所附结点编号就是本结点的编号。

- 申请者获得所有邻结点发来的回声消息后,将按上一条规则选定 S 的方法选中一个资源提供者,然后,向它发申请消息。
- 当一个结点接到申请消息后,就把申请消息登记下,并在可能时将资源分配给它。
- 使用完毕后通知资源分配者去配。

8.3.4 分布式进程同步

由于在分布式系统中,各计算机相互分散,没有共享内存,因而,在单处理机系统中采用的种种进程同步方式已不再适用。例如,两个进程通过信号量相互作用,它们必须要能访问信号量。如果两个进程运行在同一台机器上,它们就都能共享内核中的信号量,并通过执行系统调用访问。如果它们运行在不同机器上,这种方法就不灵了。采用完全分布式管理方式时,每个资源由位于不同结点上的资源管理共同来管,每个资源管理在决定分配它管理的资源以前,必须和其他资源管理者协商,对于由各计算机共享和要互斥使用的各种资源,都要采用这种管理模式。因此,采用这种管理方式时必须设计一个算法,各资源管理者按此算法共同协商资源的分配。这个算法应满足:资源分配的互斥性、不产生饿死现象、且各资源管理者处于平等地位而无主控者。通常把这种资源分配算法称分布式同步算法,由同步算法构成的机制称分布式同步机制。

分布式系统中各计算机没有共享内存区,导致进程之间无法通过传统公共变量,如锁变量或信号量来进行通信。实现分布式进程同步比实现集中式进程同步复杂得多,由于进程分散在不同计算机上,进程只能根据本地可用的信息做出决策,系统中没有公共的时钟。进程之间通过网络通信联系,时间消息通过网络传递后也会有延迟,可能资源管理程序接到不同机器上的进程同时发来的资源申请,可是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,必须先要解决对不同计算机中发生的事件进行排序的问题,然后,再设计出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的规定,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,各计算机无公共时钟,也无共享存储器,所以,很难确定系统中所发生的两个事件的先后次序。1978年Lamport提出了不使用物理时钟,而对分布式系统中所发生的事件进行排序的方法。Lamport指出时钟同步不需要绝对同步。如果两个进程是无关的,那么,它们的时钟根本无需同步。而且对于相交进程来说,通常并没有必要让进程在绝对时间上完全一致,而只要限定它们执行的先后次序一致就行了。

先定义一个关系称作“先发生”,表达为“ $a \rightarrow b$ ”读作“ a 在 b 之前发生”,意思是指系统中所有进程认为事件 a 先于事件 b 发生,有两三种情况会产生“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 是一个进程发送消息事件, b 为另一个进程接收该消息事件,则 $a \rightarrow b$ 为真;

情况(3):存在某个事件 c ,若有 $a \rightarrow c$ 并且 $c \rightarrow b$,则 $a \rightarrow b$ 为真。

情况(1)说明同一进程中发生的事件间,可按时间上发生的先后次序来确定“先发生”关

系;情况(2)指出消息决不能在发送之前就已接收,也不能在发送的同时接收,因为,传送消息过程会有时间延迟,所以,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。

按上面的方法定义事件的“先发生”关系后,同一进程中两个事件的先后关系可以被明确规定,不同进程中发生的事件间的先后关系,有一部分可以被确定,而另一部分则不能确定。例如,有三个进程 P_1 、 P_2 和 P_3 ,它们分别发生了以下事件:

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 的消息后发消息给 P_3 ;

事件 d : P_3 接收来自 P_2 的消息;

显然,我们有:

$$a \rightarrow b \rightarrow c \rightarrow d;$$

然而,如果假设 P_2 在事件 b 之前发生过某事件 f,例如,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 之间的先后关系是无法确定的。但是,如果两个事件 x 和 y 发生在不同进程中,而且这两个进程也不交换信息,那么 $x \rightarrow y$ 和 $y \rightarrow x$ 都不成立,这两个事件就称为并发事件,简单地说,无法确定这两个事件谁先谁后。

接着,定义逻辑时钟(又叫 timestamping 时间戳),它是指能为系统中的所有活动赋予一个编号的机制,它可以利用一个本地计数器来实现,定义逻辑时钟的实质是把一个系统中的事件映射到一个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 $C(a)$ 小于 $C(b)$,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。

上述关于逻辑时钟的定义指出,如果事件 a 先发生于事件 b,则 a 的逻辑时钟小于 b 的逻辑时钟。但反之却不然,因为,两个并发事件的逻辑时钟的大小没有定义,所以,逻辑时小的事件不见得先发生。

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,下面是一种简单的逻辑时钟函数构造方法。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 1 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 1 个事件,则

$$C(e_r) = 1 + C(e_s')$$

此处, e_s' 是进程 P' 发送这个消息的事件。若 e_r 是 P 的第 r 个事件, 而第 $r - 1$ 个事件是 e_{r-1} , 则

$$C(e_r) = 1 + \max[C(e_{r-1}), C(e_s')]$$

定义了逻辑时钟后, 可以对一个系统中的所有事件人为地排出一个前后关系, 对于由 n 个进程组成的网络系统, 逻辑时钟可用于对由消息传输所组成的事件进行排序。网络中的每个结点 i 都维护了一个本地计数器 C_i , 其功能相当于本地时钟。每次系统发送消息时, 它首先把时钟加 1, 然后, 发送一个消息, 其形式为: (m, T_i, i) (其中, m 为消息内容, T_i 为该消息的逻辑时钟, i 为结点编号)。因为, 分布式系统中的进程可拥有自己的逻辑时钟(各个结点上的本地时钟), 而这些时钟并非是同步运行的。可能出现这种情况: 一个进程发送的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值, 由于发送消息事件必定出现在接收消息事件之前, 故而需要调整接收进程这时的逻辑时钟。所以, 当接收消息时, 接收进程 j 按照上述规则(2)应把它的时钟设为其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, 不能倒退, 所以校正逻辑时钟值是加至少为 1 的正数, 而不是减一个正数。在每个结点, 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 , 如果:

- (1) $T_i < T_j$ 或
- (2) 如果 $T_i = T_j$ 并且 $i < j$

与每个消息有关的时间是附加在消息上的时间戳, 这些时间的顺序是通过上述两个规则确定的。

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来校正系统中发生事件的逻辑时间。如图 8-8 所示, 考虑有三个进程并发工作, 它们运行在不同的机器上, 每台机器都带有自己的时钟, 并且按照它自己的速度计时。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; 并且进程 P_2 中, 则时钟滴答却为 10。每个时钟都按恒定的速率计时, 但不同机器上的晶体振荡有差别, 造成各自的时钟速率不一样。

在时钟滴答 6, 进程 P_0 发送消息 A 给进程 P_1 。该消息花多长时间到达目的地, 取决于你基于哪一个时钟来计算。如果进程 P_1 接收到消息是在时钟滴答 16, 而同时消息 A 携带的时钟滴答为 6, 那么, 进程 P_1 可以认为此消息路上花了 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 花了 16 个时钟滴答, 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。很显然这些时间值是不可能的, 也是必须避免出现的情况, 要对它们加以校正。

Lamport 的时间校正方法是从“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,让每个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,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快于发送时间加 1 之后的时间值。图 8-8(b)可看出,消息 C 到达的时间现在为 61,同样,消息 D 到达的时间为 70。

再加一个小小的附加的条件,此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了,所加条件是:两个事件之间,时钟至少要滴答一次。如果一个进程在极短的时间内快速发送或接收两个消息,那么,必须调整时钟,使这两个事件之间时钟至少要走一个滴答。

在某些情况下,上面的算法还需要满足条件:任何两个事件都不会恰巧在完全相同的时刻发生。为了要满足这个关系,对具有相同时间戳的两个消息则通过它们所在的站点编号来排序,这种规定能避免各通信进程的不同时钟之间的重合问题。

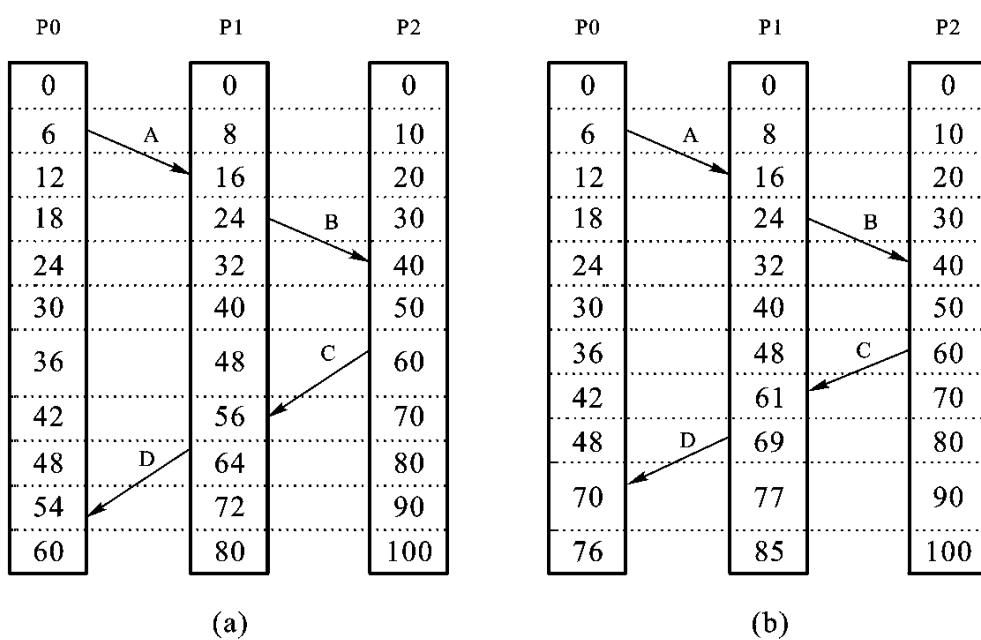


图 8-8 (a) 三个进程,各有自己的时钟

(b) Lamport 算法校正时钟

图 8-9 时间戳算法的操作例子中,有三个结点都通过一个控制时间戳算法的进程来表达。进程 P1 开始时钟值为 0,为了传送消息 a,它把时钟值加 1 并发送(a, 1, 1)。这个消息被结点 2 和 3 的进程收到,由于这两种情况中本地时钟值是 0,则时钟应被设置成值 $2 = 1 + \max[0, 1]$ 。接着 P2 首先将它的时钟增加为 3,再发出下一个消息。当接收到消息后, P1 和 P3 必须把他们的时钟增加到 4。然后,在大致相同的时间,以相同的时间戳, P1 发出消息 b,而 P3 发出消息 j。因为,前面介绍的排序原则,这不会产生混淆,在所有这些事件发生后,消

息的顺序在所有结点上是相同的,依次为 a, x, b 和 j 。

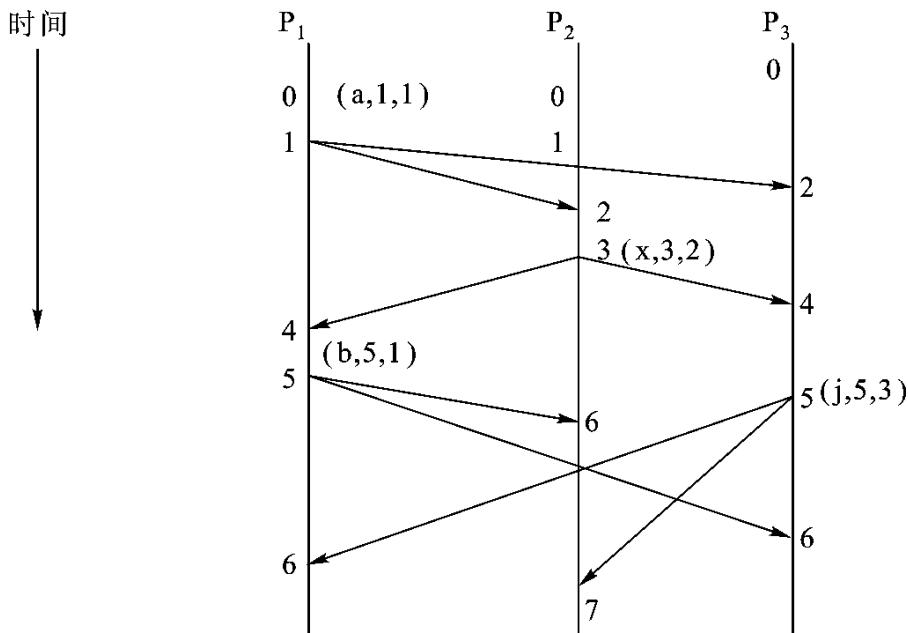


图 8-9 时间戳算法的操作例子

算法在工作时,如果不考虑在两个结点间传输消息时间上的差别,来看一下如图 8-10 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 的消息在结点 2 上比来自 P_4 的消息到得早,但在结点 3 上来自 P_1 的消息比来自 P_4 的消息到得晚。不过,当所有消息在所有结点上都接收完后,消息的顺序在所有结点上是相同的,依次为 a, q 。为什么在结点 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 $T_4 = 1$),但由于结点号 $i < j$,所以, P_1 发出消息的事件先发生, P_4 发出消息的事件后发生,故在结点 3 上,消息次序应为 a 先而 q 后。

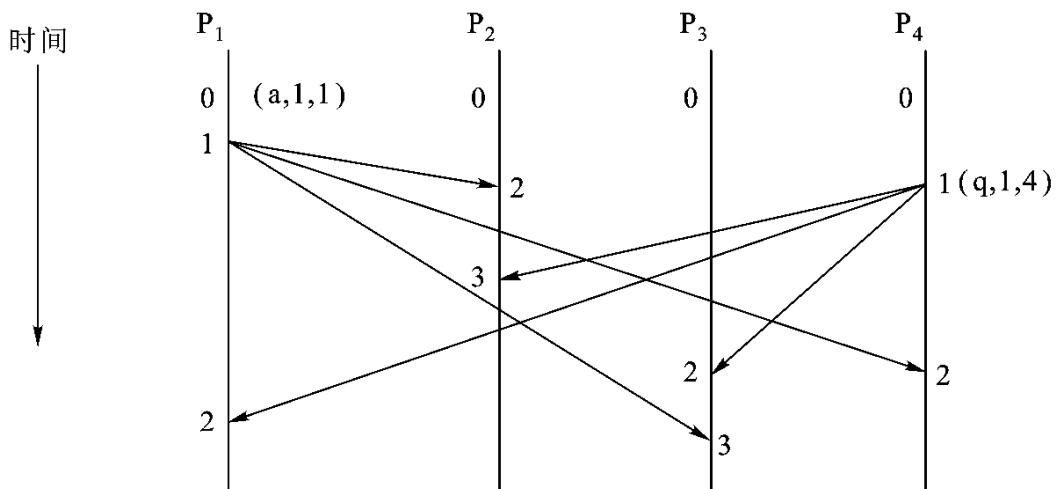


图 8-10 时间戳算法操作的另一个例子

2. 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 FCFS 次序, 对事件进行处理。

Lamport 算法基于以下假设: 分布式系统由 N 个结点组成, 每个结点建立一个数据结构, Lamport 把它叫作队列, 实际上它是一个数组, 用来记录该结点最近收到的消息和该结点自己产生的消息。不妨假定每个结点只有一个进程和仅负责控制一种临界资源, 并处理那些同时到达的请求。数组的下标也就是结点编号($1 \sim N$), 该数组被初始化为:

$$\text{applicationstack} = (\text{release}, 0, i) \quad i = 1, \dots, N$$

该算法用到了三种类型的消息:

- ($\text{request}, T_i, i$) 进程 P_i 发出的访问资源的请求消息
- (reply, T_j, j) 进程 P_j 同意请求进程访问其控制的资源的回答消息
- ($\text{release}, T_k, k$) 占有资源的进程 P_k 释放资源时给各进程的释放消息

每个消息的内容包括: 消息类型、时间戳、结点号。

Lamport 算法描述如下:

(1) 当进程 P_i 要求访问临界资源时, 它向其他各进程分别提交带有本地时间戳的申请消息($\text{request}, T_i, i$), 同时也把此申请消息放入自己的请求队列中的 $\text{applicationstack}[i]$ 位置。注意, 申请事件必定先于发送事件, 所以申请消息的时间戳必先于发送申请消息的时间戳。

(2) 当其他结点进程 P_j 接收到申请消息($\text{request}, T_i, i$)时, 它把这条消息放入自己队列的 $\text{applicationstack}[i]$ 位置。若进程 P_j 在收到($\text{request}, T_i, i$)时, 发生了申请事件且还未发完申请消息, 那么, 要在接收进程 P_j 发完所有申请消息后才发回答消息, 否则立即发回答消息。

(3) 若满足以下条件, 则允许进程 P_i 进入临界区访问资源:

1) P_i 申请请求访问资源的消息、即 applicationstack 中的 request 是队列中最早的申请请求消息, 因为, 消息在所有结点一致排序, 故在任意时刻只有一个进程访问资源。

2) 进程 P_i 已收到了所有其他进程的回答消息 reply , 都同意其访问资源, 而且响应消息上的时间戳晚于(T_i, i)。此条件表明其他进程要么不访问资源, 要么要求访问但其时间戳较晚。

(4) 为了释放该资源, P_i 从自己的请求队列 $\text{applicationstack}[i]$ 位置消去($\text{request}, T_i, i$), 同时再发送一条打上时间戳的($\text{release}, T_i, i$)给所有其他进程。

(5) 当进程 P_j 收到进程 P_i 的释放消息 release 后, 从自己的队列 $\text{applicationstack}[i]$ 中消去 P_i 的请求消息($\text{request}, T_i, i$)。

为了确保互斥, 该算法共需要传送 $3(N - 1)$ 条消息, 包括, $(N - 1)$ 条 request 消息、 $(N - 1)$ 条 reply 消息和 $(N - 1)$ 条 release 消息。

上述算法可用程序描述如下：

各进程间传递的消息采用的数据类型}

```

type message = record
    class: (application, reply, release);   消息分申请消息、回答消息和释放消息}
    source: 1..n;                          发消息进程编号}
    timestamp: integer;                   消息上加盖的时间戳}
    clock: integer;                      发消息时的时间戳}
    valid:Boolean;                       取真值时消息有效}
end;

```

每个进程都定义了下列变量和过程}

```

var
    T:integer;                           逻辑时间, 初值为 1}
    applicationstack:array [1..n] of message;  来自各进程的申请消息, 开始全为无效消息}
                                                回答消息计数}
    replycount:0..n - 1;                申请资源时调用此过程}

procedure Apply;
    var M:message;
        i:integer;
    begin with M do
        begin class:=application;          准备申请消息}
            source:=me;                  {me 为本进程编号}
            timestamp:=T;               镇时间戳}
            valid:=true;
        end;
        T:=T + 1;
        applicationstack[me]:=M;
        replycount:=n - 1;
        For i := 1 to n do
            begin    M.clock:=T;T:=T + 1;
                If i ≠ me then send(M,i)  申请消息发给进程 i, 且不等待}
            end;
        waitfor(Replycount = 0);           等所有回答消息, 应为 n - 1 个}
        for i:= 1 to n do
            waitfor (not Applicationstack[i]→Applicationstack[me]);  删别消息数组中、时间戳的先后}
                                                关系 →”表示前者是有效消息, 且前者排在后者之前}
        get resource;                    可以使用资源了}

```

```

    end;

procedure Receive(var M:message);
进程空闲时,或处于 waitfor 等待态时才调用此过程来接收消息}

var R:message;
begin T:= 1 + max(M.clock,T);

    with M do
        case class of
application:
    begin applicationstack[source]:= M;
        with R do
begin
    class:= reply;
    source:= me;
    clock:= T;
end;
T:= T + 1;
send(R,source);          发回答消息给申请者}
end;
reply:
    replycount:= replycount - 1;
release:
    applicationstack[source].valid:= false;
end;

procedure Release;           释放资源时,调用此过程}

var R:message;
    i:integer;
begin  with R do
        begin class:= release;
        source:= me;
        clock:= T;
end;
T:= T + 1;
applicationstack[me].valid:= false;
for i:= 1 to n do
    if i ≠ me then send(M,i);      向其他进程发释放消息}
end;

```

(2) G. Ricart 算法

Ricart 等人对上述算法作了改进, 试图通过消除 release 消息, 使进程访问资源时, 仅需发送 $2(N - 1)$ 个消息。该算法描述如下:

(1) 当进程 P_i 要求访问临界资源时, 它发送一个打上时间戳的广播消息 ($\text{request}, T_i, i$) 给所有结点的进程。

(2) 当其他结点进程 P_j 接收到消息 ($\text{request}, T_i, i$) 时, 执行以下操作:

1) 如果进程 P_j 既不是资源的申请者, 又不是资源的占有者, 立即返回一个回答消息 (reply, T_j, j) 给进程 P_i 。

2) 如果进程 P_j 是资源申请者, 并满足条件:

$$T_i < T_j \text{ 或 } T_i = T_j \wedge i < j$$

则返回一个回答消息 (reply, T_j, j) 给进程 P_i ; 否则推迟发送 reply 响应。

(3) 当进程 P_i 已收到了所有其他进程的回答消息 reply 时, 便可以访问该临界资源;

(4) 占有资源的进程 P_i 在释放资源时, 对那些曾经接到过它们的申请消息, 但未予回答的进程补送回答 reply 响应。

上述算法可用程序描述如下:

```

type message = record
    class: (application, reply);           消息分申请消息、回答消息}
    source: 1..n;                         发消息进程编号}
    timestamp: integer;                   消息上加盖的时间戳}

    end;

var
    T: integer;                          逻辑时间, 初值为 1}
    Applicationtime: integer;           发申请消息时刻}

    Replydeffered: array [1..n] of Boolean;
        如果 i 号进程申请资源, 本进程暂不回答, 则置 Replydeffered[i] 为真, 初值为假}

    Replycount: 0..n - 1;                回答消息计数}

    Requesting: Boolean;               申请资源时取真}

procedure Apply;          申请资源时调用此过程, 执行后即获得资源}
Var M:message;
begin with M do
    begin class:=application;
        Timestamp:=T;
        Source:=me;
    end;
end;

```

```

applicationtime := T;
T := T + 1;
requesting := true;
replycount := n - 1;
broadcast(M);                                {发广播申请消息}
waitfor(replycount = 0);                      {等待直到 replycount = 0}

end;

procedure Receive(var M:message);
进程空闲时,或处于 waitfor 等待态时,才可调用此过程来接收消息}

begin with M do
  case class of
    application:
      replydeffered[source] := 
        requesting and ((timestamp > applicationtime) or
                         (timestamp = applicationtime and source > me));
    接到 source 的申请消息后,决定要否延迟发回答消息}

      T := 1 + max(T, timestamp);
      If not replydeffered [source] then
        sendreply(source);                     {回答消息发给 source}

    reply :
      replycount := replycount - 1;
    end;

  procedure Release;                           {释放资源时,调用此过程}
    var i:integer;
    begin for i := 1 to n do
      if replydeffered[i] then
        begin replydeffered[i] := false;
          sendreply(i);                      {发释放消息}
        end;
        requesting := false;
      end;
    end;

```

该算法中每个进程的状态转换图如图 8-11 所示。

下面对 Ricart 算法做一个简单小结,当一个进程欲进入临界区时,它就给所有其他进程发送一个带时间戳的请求消息,当它从所有其他进程收到回答后,就可以进入临界区了。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。如果该进程不想进入临界

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,($N - 1$)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。

本算法采用时间戳来确保互斥,它也可以避免死锁,为了证明不死锁,可用反证法:如果可能死锁,在没有消息传送时,就会出现每个进程都发送了一个请求但没有收到相应的回答的情况。由于按请求的顺序来延迟回答,因此,这种情况不会出现。于是具有最早时间戳的请求必定会收到所有相应的回答,从而,不可能发生死锁。饥饿也可以避免,这是因为请求是有序的,由于请求按序满足,每个请求在某时刻会成为最早的,于是其资源请求会得到满足。

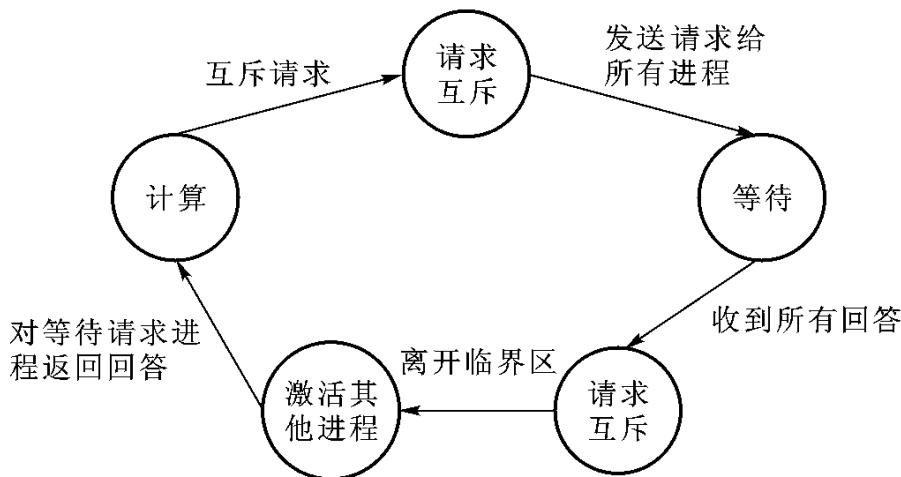


图 8-11 Ricart 算法状态图

(3) 令牌环算法

这是由 Suzuki 在 1982 年提出来的用于分布式系统中完全不同的另一种同步算法,为了实现进程互斥,所有进程构成一个逻辑环(Logical Ring),环中的每一个进程有一个上家和一个下家,这样的逻辑环可以由软件实现,并不意味着任何的物理拓扑结构。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 传递给它的下家进程 $K + 1$,可以按点到点方式进行传递。当一个进程从上家手中得到令牌时,它就检查它欲进入的临界区,如果临界区是开放的,则该进程进入临界区,访问共享资源。当该进程退出临界区时,便把令牌传送给它的下家,不允许使用同一张令牌进入第二个临界区。

当一个进程得到了上家传递来的令牌,又不想进入临界区,就把令牌往下传。这样一来,如果没有进程想进入临界区时,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,由于仅有一张令牌,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。由于令牌以固定顺序移动,存在着上家优先于下家的关系,也不会出现饥饿现象,一旦一个进程想进入临界区,最坏情况下是等待所有其他进程进入后再退出临界区所用时间的总和。

执行令牌算法时,必须能保证逻辑环中任何时候都有令牌在循环。实际上检测令牌丢失是很困难的,因为,网络上令牌两次出现的时间是不定的,一段时间没有发现令牌移动并不意味着它丢失了。但一旦发现令牌丢失,应立即选定一个进程来产生新令牌。此外,逻辑环还应能及时发现环路中某个进程失效和退出,以及通信链路故障,这时应立即撤消该进程,并重构逻辑环。

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,除了因竞争可重复使用资源而产生死锁外,更多地会因竞争临时性资源而引起死锁。虽然,对于死锁的防止、避免和解除等基本方法与单处理机相似,但难度和复杂度要大得多。由于分布式环境下,进程和资源的分布性,竞争资源的诸进程来自不同结点。然而,拥有共享资源的每个结点,通常只知道本结点中的资源使用情况,因而,检测来自不同结点中进程在竞争共享资源是否会产生死锁,显然是很困难的。

分布式系统中的死锁可以分成两类:资源死锁和通信死锁。资源死锁是因为竞争系统中可重复使用的资源,如打印机、磁带机、以及存储器等引起的,一组进程会因竞争这些资源,而由于进程的推进顺序不当,从而,发生系统死锁。在集中式系统中,如果进程 A 发送消息给 B,进程 B 发送消息给 C,而进程 C 又发送消息给 A,那么,就会发生死锁。在分布式系统中,通信死锁指的是;在不同结点中的进程,为发送和接收报文而竞争缓冲区,如果出现了既不能发送,又不能接收的僵持状态。

2. 分布式死锁检测与预防

有两种检测方法:

1) 集中式死锁检测

分布式系统中,每台计算机都有一张进程资源图,描述进程及其占有资源状况,让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结点发送的更新信息,可用以下办法解决更新问题:一是每当资源图中加入或删除一条弧时,相应的变动消息就应发送给检测进程;二是每个进程可以周期性的把自己从上次更新后新添加或删除的弧的信息发送给检测进程;三是在需要的时候检测进程主动去请求更新信息。

上述方法可能会产生假死锁问题,因为,在网络和分布式环境下,如果检测出进程资源图中的环形链,是否系统真的发生了死锁呢?答案是不确的,也可能真的发生了死锁,也可能是假的死锁,其原因是进程所发出的请求与释放资源命令的时序,与执行这两条命令的时序未必一致。下面通过一个例子来说明假死锁的情况,考虑进程 A 和 B 运行在结点 1 上,C 运行在结点 2 上;共有三种资源 R,S 和 T;开始状态如图 8-12(a)(b) 所示:

- A 拥有 S 请求 R,但 R 被 B 占用;
- B 使用 R;
- C 使用 T。

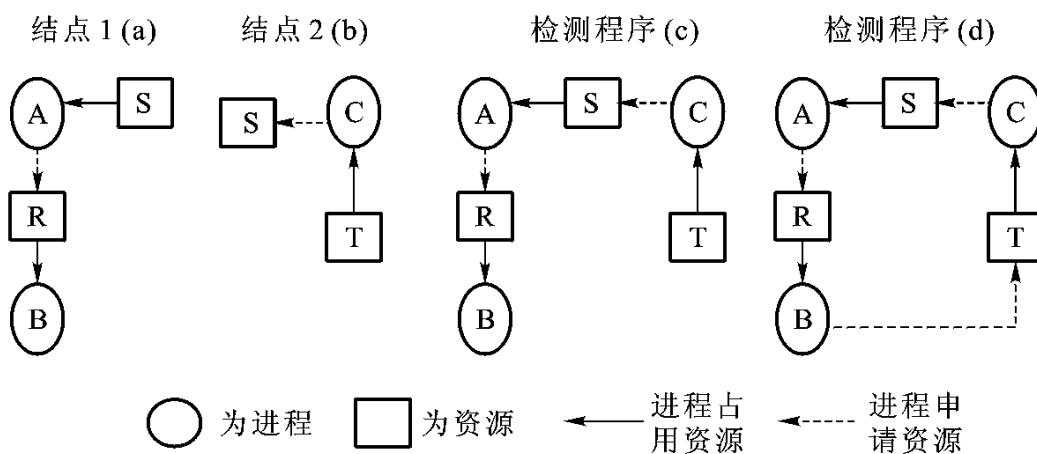


图 8-12 集中式死锁检测

检测进程检测到的状态如图 8-12(c) 所示,这时系统是安全的。一旦进程 B 运行结束,A 就可以得到 R,然后,运行就能结束,并释放进程 C 所等待的 S。但不久之后,进程 B 释放了 R 并同时请求 T,这是一个合法操作。结点 1 向检测进程发送消息声明进程 B 正在等待它的资源 T。假如,结点 2 的消息比结点 1 发送的消息先到达,这就导致了图 8-12(d) 所示的资源图,检测进程错误地得出死锁存在的结论,并中止某进程。由于消息的不完整和延迟使得分布式死锁算法产生了假死锁问题。

可以用 Lamport 算法提供的全局时间来解决假死锁问题。从结点 2 到检测进程的消息是由于结点 1 的请求而发出的,那么,从结点 2 到检测进程的消息的逻辑时钟就应该晚于从结点 1 到检测进程的消息的逻辑时钟。当检测进程收到了从结点 2 发来的有导致死锁嫌疑的消息后,它将给系统中的每台机器发一条消息“我收到了从结点 2 发来的会导致死锁的带有逻辑时钟 T 的消息,如果任何有小于该逻辑时钟的消息要发给我,请立即发送。”当每台机器给出肯定或否定的响应消息后,检测进程会发现从 R 到进程 B 的弧已消失了,因而,系统仍然是安全的。这一方法的缺点是需要全局时间、开销较大。

2) 分布式死锁检测

分布式检测算法无需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,具体实现方法如下:

- (1) 在每个结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,每个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给管理该资源的进程;
- (4) 每个进程应将资源的已分配情况通知所有进程。

由上可见,为实现分布式环境下的死锁检测,通信的开销相当大,而且还可能出现假死锁,因而,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,可以采取破坏产生死锁的四个必要条件之一的方法来实现。为了防止资源死锁,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的全部网络资源。这样,进程在运行中不会再提出资源申请,破坏了“占用和等待”条件。如果网络系统无法满足进程的所有资源要求,索性一个资源也不分配给该进程,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资源号从小到大的次序提出,这样可防止在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。为了避免发生组装型死锁,源结点的发送进程,在发送报文之前,应先向目标结点申请一份报文所需的全部缓冲区,如果目标结点无缓冲区,干脆便一个也不分配,让发送进程等待。为了避免存储转发型死锁,可以为每条链路上的进程配置一定数量缓冲区,且不允许其他链路上的进程使用;或者当结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,它允许通过网络来互连的,使不同机器上的用户共享文件的一种文件系统。它的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。它不是一个分布式操作系统,而是一个相对独立的软件系统,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:

- 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;
- 位置透明性。客户通过文件名访问文件,但并不知道该文件在网络中的位置;同理文件的物理位置变了,但只要文件的名字不变,客户仍可进行访问。

在分布式系统中,区分文件服务(File Service)和文件服务器(File server)的概念是非常重

要的。文件服务是文件系统为其客户提供的各种功能描述,如可用的原语、它们所带的参数和执行的动作。对于客户来说,文件服务精确地定义了他们所期望的服务,而不涉及实现方面的细节。实际上,文件服务提供了文件系统与客户之间的接口。

文件服务器是运行在网络中某台机器上的一个实现文件服务的进程,一个系统可以有一个或多个文件服务器,但客户并不知道有多个文件服务器及它们的位置和功能。客户所知道的只是当调用文件服务中某个具体过程时,所要求的工作以某种方式执行,并返回所要求的结果。事实上,客户不应该也不知道文件服务是分布的,而它看起来和通常单处理机上的文件系统一样。

2. 分布式文件系统的组成

分布式文件系统为系统中的客户机提供共享的文件系统,为分布式操作系统提供远程文件访问服务。分布式操作系统通常在系统中的每个机器上都有一个副本,但分布式文件系统并不一样。它由两部分组成:运行在服务器上的分布式文件系统软件和运行在每个客户机上的分布式文件系统软件。这两部分程序代码在运行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,而它就可以支持多个实际的不同文件系统,分布式文件系统将通过虚拟文件系统(vfs)和虚拟节点(vnode)与本机文件系统交互作用。例如,Sun公司的网络文件系统NFS由以下组成部分:

- 网络文件系统协议。定义了一组客户可能向服务器发送的请求,以请求所用参数和可能返回的应答。
 - 远程过程调用协议。定义了客户和服务器之间所有的交互格式。
 - 扩展数据表达(XDR)。提供与机器无关的通过网络传送数据的方法。
 - 网络文件系统服务器代码。负责处理所有客户机的请求,执行相关文件服务。
 - 网络文件系统客户机代码。通过用户对本机文件系统的系统调用转换成一个远程过程调用,并通过向服务器发送一个或多个RPC请求,来实现客户对远程文件的访问。
 - 安装协议。定义了为客户机安装和卸载文件系统及子目录树的操作和语义。
 - 服务器监听进程。负责监听以及响应客户机的服务请求。
 - 服务器安装进程。负责处理客户机的安装请求。
 - 客户机I/O进程。负责客户机的文件块的异步I/O。
 - 网络锁定管理器和状态监视器。实现对网络中文件的锁定功能。

3. 分布式文件系统体系结构

分布式文件系统的体系结构目前多数采用客户/服务器模式,客户是要访问文件的计算机,服务器是存储文件并且允许用户访问这些文件的计算机。分布式文件系统中需要解决

的另一问题是命名的透明性,大致上有三种办法: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,这样,用户可以自己定制文件名字空间;三是让所有机器上看起来有相同的单一名字空间,这种方法实现难度较大。分布式文件系统中需要解决的第三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。但有些系统中的服务器能提供更多的服务,它不仅响应客户的请求,还对客户机中的高速缓存的一致性作出预测,一旦客户数据变为无效时便通知客户。下面介绍一个实际的分布式文件系统——网络文件系统 NFS。

(1) NFS 的结构

NFS 的基本思想是让任意组合的客户机和服务器共享一个公共的文件系统。在大部分情况下,所有的客户机和服务器都连结在同一个局域网上,但这并不是必需的,NFS 也可以在广域网上运行。

每个 NFS 服务器都输出一个或数个目录供远程客户机访问。一个目录可用,总是意味着它的所有子目录也都是可用的,即输出的总是一个目录树。服务器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前必须先安装它们,客户机安装了远程目录后,这个目录就成了客户机目录层次的一部分,图 8-13 NFS 层次结构。Sun 有一些工作站是无盘的。无盘的客户机总是在引导时将远程的文件系统装入自己的根目录,形成一个完全由远程服务器支持的文件系统。有盘工作站则只将远程目录装在本地目录层次的某个位置,形成一个部分本地、部分远程的文件系统。对运行在客户机上的程序而言,文件是位于本地还是远程机器是没有什么两样的。

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了某个目录,它们就可以通过共享其中的文件来实现通信。例如,客户机 A 可以创建一个文件,而客户机 B 可以读这个文件。共享文件的前提只有一个,即不同的客户机安装了相同的目录。对共享文件的读写与通常的文件访问没有区别。这种简便之处是 NFS 受到欢迎的一个重要原因。

(2) NFS 协议

NFS 的一个主要目标是支持异构型系统,即客户机和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。只有这样,才能写出正确的客户机和服务器程序。NFS 定义了两个客户机/服务器协议来实现这一目标。

协议是一组由客户机送往服务器的请求,以及相应的从服务器送回客户机的应答。只要服务器能够识别和处理协议中的所有请求,它就不需要了解客户机的有关情况。同样,客户机也可把服务器看做是能接收和处理一组特殊请求的过程,至于服务器如何做,则是它们自己的事。

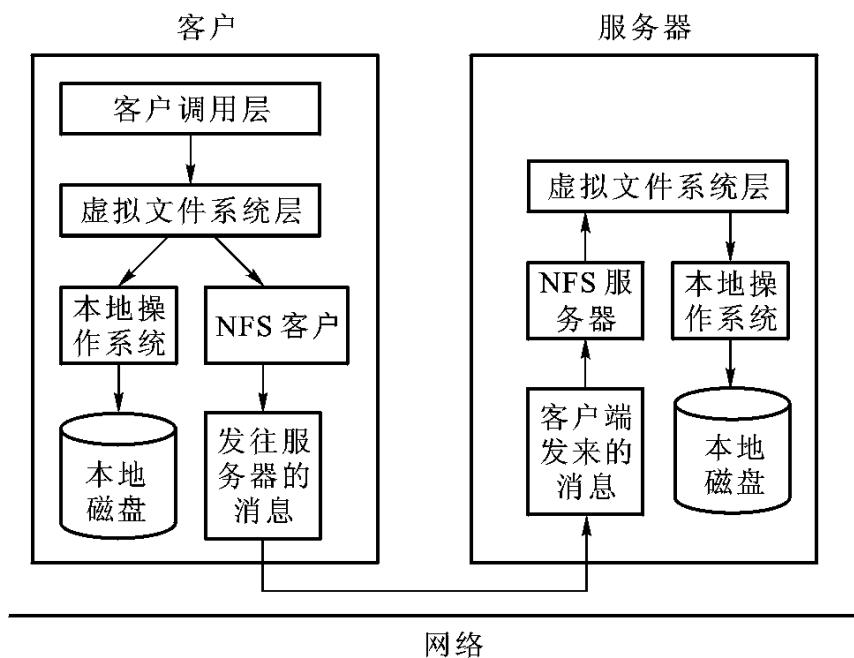


图 8-13 NFS 层次结构

NFS 的第一个协议是处理安装问题。客户机向服务器送出一个路径名, 请求允许将该目录安装在其目录层次的某个位置。实际安装的位置并不包含在消息中, 因为服务器不关心它。如果路径名是合法的, 且该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了相应目录的文件系统类型、所在硬盘、目录的 i 结点号以及安全信息。此后对已安装了的目录内的文件的访问就通过这个句柄进行。

为了避免手工操作, 许多客户机被配置成能在启动时自动安装远程目录。通常, 这类客户机中有一个称为 /etc/rc 的 Shell 文件, 这个文件中包含了安装远程文件系统的命令。客户机启动时, 这个 Shell 文件会被自动执行。

另外, Sun 的 UNIX 操作系统也支持自动安装功能。这个功能是将一组远程目录与一个本地目录对应起来, 但在客户机系统启动时却不安装远程目录, 甚至也不与远程的服务器发生接触。只有当需要打开远程文件时, 操作系统才给相应的若干个服务器发出消息。实际安装的是第一个送回应答消息的服务器上的相应目录。

与通过 /etc/rc 文件实现的静态安装相比, 自动安装有两个优点。首先, 如果 /etc/rc 中给出的某个 NFS 服务器没有开机或出现故障, 客户机就可能引导不起来, 或者出现一些困难、延迟和错误信息。如果用户当时并不需要这个文件服务器, 则所有的安装工作都浪费了。其次, 允许客户机同时试着接通多个服务器, 可以实现某种程度的容错性, 且有助于提高性能。

另一方面, 上述过程也要求所有可供自动安装的文件系统都是一样的。由于 NFS 本身并不支持文件或目录的重复, 用户必须承担保证可能被自动安装的多个文件系统的一致性

的责任。因此,自动安装常用于只读的文件系统,如那些包含系统二进制文件和其他极少改变的文件的文件系统。

NFS 的第二个协议用于文件和目录的访问。客户机可以向服务器发送消息,以操作目录和读写文件。另外,客户机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 支持大部分的 UNIX 系统调用,但不支持 open 和 close。

省略 open 和 close 并不是因为偶然的错误,而是完全有意识的。原则上,在读文件前打开文件和在读完后关闭文件都是不必要的。客户机在读文件时,先向服务器发一个带文件名的消息,要求查找这个文件并返回一个文件句柄,这个句柄实际上就是一个与该文件有关的数据结构。不同于 open 系统调用,这个 lookup 操作并不将任何信息复制到系统的内部表格中。系统调用 read 的参数包括欲读文件的句柄,开始读处在文件中的偏移量及需要读取的字节数。每个这样的消息都是自足的。这种做法的好处是服务器不用记住对它的各种调用之间的相互联系,从而当服务器发生故障并恢复后,也不会丢失与打开文件有关的信息(因为根本就没有这样的信息)。像这样不保留有关文件打开的状态信息的服务器称为无状态的服务器。

但是,NFS 的上述特点却是与 UNIX 的文件语义并不严格一致的。例如在 UNIX 中,可以对打开的文件加锁以阻止别的进程访问该文件,且只有当文件被关闭后,锁才被释放。在 NFS 提供的文件服务中,由于服务器不知道哪些文件是打开的,故无法将锁与打开的文件联系起来。为此,NFS 需要一个单独的处理锁的机制。

NFS 使用了 UNIX 对文件访问方式的保护机制,即相当于文件主,同组同户和其他用户的 rwx 位。早先,每个请求住处都附带有调用者的用户号和组号,NFS 就用它们来检查访问的合法性。实际上,这种做法要求客户机不实施欺骗行为。数年的经验表明,这样的假设是没有意义的。现在,可以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键。当采用这种选项时,未经授权的客户机由于不知道正确的密码而无法弄假。这里,密码系统只用于核对参与通信的双方,而不对涉及到的数据进行重编码。

用于核查的所有密钥及其他信息都通过 NIS(网络信息服务)实现。NIS 先前的名字叫黄页(yellow page),其作用是存储(密钥,值)对。在给定密钥后,它给出相应的值。除了承担解密工作外,它还存放用户与其加密了的口令、机器名与其网络地址等对应关系。

网络信息服务是通过主从方式分布的。为了读到数据,进程可以通过主页,也可以通过它的某个副本。但所有的改变都应该针对主页进行,然后再传给从页。因此,在更新结束后的一个小间隔内,数据是不一致的。

(3) NFS 的实现

NFS 的协议与客户机及服务器的代码是互相独立的,这里讨论的是 SUN 的 NFS 实现。它分成三层,最顶层是系统调用层,处理诸如 open、read、close 等调用。通过对调用的词法进

行分析，并核对参数后，它就调用第二层，即虚拟文件系统（VFS）层。

VFS 层的作用是建立并维持一个打开文件表，它的每一项对应一个打开了的文件，很像 UNIX 中打开文件的 *i* 结点表。在常规的 UNIX 中，*i* 结点是通过（设备，*i* 结点号）惟一地指定的，而 VFS 层则为每个打开文件设立了一个称为 *v* 结点（虚拟 *i* 结点）的项，用于指出文件是本地的还是远程的。

下面通过考察 mount、open、read 等一串系统调用，说明 *v* 结点的使用方式。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有关信息。

(1) 安装（Mount） 安装程序首先分析远程目录，找出远程机器的名字，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，服务器就返回一个该目录的句柄。最后，mount 程序执行 mount 系统调用，将这个句柄交给核心。

核心接到句柄后，为远程目录构造一个 *v* 结点，要求 NFS 客户机代码在其内部表中创建一个 *r* 结点（远程 *i* 结点）来装这个文件句柄，并将 *v* 结点指向 *r* 结点。VFS 层中的每一个 *v* 结点最终都有一个指针指向 NFS 客户机代码中的一个 *r* 结点，或本地操作系统中的 *i* 结点。因此从 *v* 结点中可以看出文件或目录是本地的还是远程的，对远程的还可以找到句柄。

(2) 打开（Open） 打开远程文件时，通过分析路径名，核心会遇到安装远程文件系统的那个目录。当发现该目录不在本地后，通过 *v* 结点就可找到指向 *r* 结点的指针。于是，核心就要求 NFS 客户机程序去打开文件。NFS 客户机程序在远程服务器的相关目录中查找剩下的那部分路径名，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* 结点，并报告 VFS 层，后者接着就在自己的表中加入一个 *v* 结点，并让它指向 *r* 结点。

Open 的调用者将得到一个远程文件描述符，它实际上对应于 VFS 层次中的某个 *v* 结点。注意在服务器那边没有建立任何表格。尽管服务器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，就可以用。在选择了安全选项时，合法性检查中包括密码的核对。

(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* 结点，弄清它是本地的还是远程的，并得到对应的 *i* 结点或 *r* 结点。

出于效率方面的考虑，客户机和服务器之间的数据传送以较大的块（通常是 8 192 字节）进行，尽管有时实际需要的数据量较小。客户机在得到它所需要的 8 K 字节后，马上自动地请求得到下一块，这样当真的需要下一块时，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。

写操作采用的策略是类似的。如果 write 系统调用提供的数据少于 8 912 字节，数据将只在本地累积。只有当整个 8 KB 的块满了以后才送往服务器。当然，当文件被关闭后，所有的数据都将被送往服务器。

用于提高性能的另一种方式是像常规 UNIX 那样利用快速缓存(Cache)技术。服务器会缓存数据以避免磁盘访问,但这个动作不为客户机所见。客户机则维持两个缓冲区,一个用于文件属性(i 结点),另一个用于文件中的数据。无论是需要 i 结点还是数据块,都首先在客户机的缓冲区中找。如果找到,那就不需要通过网络进行传输了。

虽然客户机上的缓冲大大地提高了性能,但它也带来了一些麻烦。假设两个客户机都缓存了同一个文件块,且其中的一个修改了数据。当另一个读数据时,它实际上取得了不正确的旧数据:Cache 已不再一致了。这个问题与我们前面谈过的多处理机中的情况是一样的。但在那个时候,我们是通过让 Cache 监视总线上的写操作并适时更新有关数据来保证一致性的。但对文件 Cache,这种方法就失效了,因为命中文件 Cache 的写操作不会产生任何可供检测的网络通信。即使产生了网络通信,现在也没有合适的能同时侦听所有网络操作的硬件机制。

NFS 在实现上采取了一些措施来减轻文件 Cache 的不一致带来的影响。其一是,给每个 Cache 块设置一个定时计数器,时间满后就抛弃该块。数据块的定时通常是 3 秒,目录块是 30 秒。这样做减少了发生不一致的风险。另外,当打开一个已被缓存的文件时,总是先向服务器发消息去查看文件的最近修改时间。如果最近修改时间在缓存文件的动作之后,就必须丢弃 Cache 中的副本,并从服务器取得新的副本。最后,每经过 30 秒,所有已被修改过的 Cache 块都将被送回服务器。

即便如此,NFS 还是因为没能很好地实现 UNIX 语义,如前节所述,某个客户机对文件的写操作是否为别的客户机所见完全取决于时间方面的因素。进而,当创建新文件时,必须等 30 秒以后,外部世界才能知道这件事。还有其他类似的问题。

从这个例子可以看到,尽管 NFS 提供了一个共享的文件系统,但由于它是通过在 UNIX 系统上打补丁得到的,文件访问的语义就不再是原先那个样子了,而且多个互相协作的程序同时工作时还可能得到与时间有关的不同的结果。而且,NFS 处理的唯一对象是文件系统,诸如进程执行等其他许多问题都根本没有涉及。然而,NFS 的使用还是非常广泛的。

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结点迁移到另一个结点,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. 计算和数据的迁移

(1) 数据迁移(Data Migration)

假如系统 A 中的用户欲去访问系统 B 中文件的数据,可以采用以下两种方法来实现数据传送。第一种方法,是将系统 B 中的整个文件送到系统 A 中,这样,凡是系统 A 中的用户

要访问该文件时,都变成了本地访问。当用户不再需要此文件时,若文件拷贝已被修改,则须把已修改过的拷贝送回系统 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,系统的效率较低。

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该文件中的另一部分,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传回系统 B。Sun 公司的网络文件系统 NFS 和 Microsoft 的 NETBU 便使用了这种方法。

(2) 计算迁移(Computation Migration)

在有些情况下,传送计算要比传递数据效率高。例如,有一个用户应用,它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,便须将驻留在不同系统上的所需文件传送到用户应用驻留的系统中。这样,要传送的数据量相当大,可以采计算迁移来解决这个问题。

计算迁移可以有多种不同的执行方式。它可以通过 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给自己;它也可以发送多个消息给各个驻留了文件的系统,这些机器上的操作系统将创建一个进程来处理相应文件,进程处理完毕后再把结果传递回请求进程。注意,在第二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。上述两种方法,经过网络传输的数据相当少。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(process Migration)

进程迁移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机上运行,也可以被迁移到另一台机器上继续运行。下列原因需要引入进程迁移。1) 负载均衡。分布式系统中,各个结点的负荷经常不均匀,此时,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的系统中去,以改善系统性能;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,应将它们迁移到同一系统中,以减少由于它们之间频繁地交互而加大通信开销。类似地,当某进程在执行数据分析时,如果它们所需的文件远远大于进程,则此时应该把该进程迁移到文件所在驻留的系统中去,能进一步降低通信开销;3) 加速计算。对于一个大型应用,如果始终在一台处理机上执行,可能要化费较多时间,使作业周转时间延长。但如果能为该作业建立多个进程,并把这些进程迁移到步台处理器上执行,会大大加快该作业的完成时间,从而,缩短作业的周转时间;4) 特殊功能和资源的使用。通过进程迁移来利用特殊结点上的硬件或软件功能或资源。此外,在分布式系统中,如果某个系统发生了故障,而该系统中的进程又希望继续下去,则分布式操作系统可以把这些进程迁移到其他系统中去运行,提高了系统的可用性。

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 1)

由谁来发动进程迁移? 2) 如何进行进程迁移? 3) 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了系统负载监视模块, 设定其中一个结点上的为主模块。主模块定时地与各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, 主模块便可启动进程迁移, 向负载沉重的系统发出命令, 让其把若干进程迁移到负载轻的系统中去。当然, 这对用户是透明的, 所有进程迁移工作都由系统完成。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部分成为进程迁移的发动者。

在进程进行迁移时, 应将把系统中的已迁移进程撤消, 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, 所迁移的是进程映象, 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修改。

进程迁移的过程并不复杂, 但需要花费一定的通信开销, 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如下两种办法:一是传送整个地址空间, 把一个进程的所有映象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, 且进程只需要用到一部分程序和数据时, 会造成浪费;二是仅传送内存中的且已修改了的那部分地址空间, 若程序运行时还需要附加的虚存空间部分信息, 则可以通过请求方式予以传送。这样, 所传送的数据量是最少的, 但源系统中仍然必须保存被迁移进程的数据及相关信息, 源系统并未从对该进程的管理中解脱出来。三是预先复制, 进程继续在原结点上执行, 而地址空间被复制到目标结点上, 由于原结点上的某些地址空间内容又被修改过, 所以, 需要有二次迁移, 这种方法能减少进程被冻结的时间。如果被迁移的进程还打开了源系统中的某些文件, 可用两种方法来处理, 一种方法是将已打开的文件随进程一起迁移, 这里存在的问题是:进程有可能仅仅临时迁移过去, 返回时才需要访问该文件;第二种方法是暂时不迁移文件, 仅当迁移后的进程又提出对该文件的访问要求时, 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进程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构, 用于暂时保存这类信息, 还需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, 源系统便可将已收到的相关信息转发至目标系统。

IBM 的 AIX 是一种分布式 UNIX 操作系统, 它提供了一种实用的进程迁移机制。进程迁移的步骤如下:

- 当进程决定迁移自身时, 它先选择一个目标机, 发送一个远程执行任务的消息, 该消息运载了进程映象及打开文件的部分信息;

- 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;
- 这个新进程收集完成其操作所需的环境、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。
- 迁移完成后发消息通知源进程, 源进程就发一个最后完成消息给新进程, 然后删去自己。

8.4 实例研究: Windows 2000 网络体系结构和网络服务

8.4.1 Windows 2000 网络体系结构

1. Windows 2000 网络构架的组件

Windows 2000 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。Windows 2000 网络构架利用 I/O 层次用于支持网络的扩展性, 使得将来的发展能适应计算机网络的发展。当新的协议诞生时, 开发人员可以编写 TDI 传送器在 Windows 2000 中实现此协议。类似地, 新的 API 能够接入现存的 Windows 2000 协议驱动程序中。Windows 2000 的网络构架的各类组件包含:

(1) 网络 API 为应用程序提供一种独立于协议的方式用于网络通信。网络 API 既能在用户态实现, 也可以同时在用户态与核心态实现, 并且有时将其他实现一些特定的程序模型或者额外服务的网络 API 包装在一起。

(2) 传输驱动程序接口 TDI(Transport Driver Interface)客户 是核心态的设备驱动程序, 而设备驱动程序通常实现了网络 API 的核心态部分。TDI 客户从发送至协议驱动程序的 I/O 请求分组(IPR)中获得自己的名称, 这些 IPR 的格式符合 Windows 2000 传输驱动程序接口标准, 这个标准为核心态设备驱动程序定义了公共编程接口。

(3) TDI 传送器 是工作在核心态的协议驱动程序。它们接收从 TDI 客户传来的 IPR, 然后, 处理这些 IPR 中的请求。为了让 TDI 传送器根据不同的协议(例如 TCP, UDP, IPX)将协议头加入 IPR 的数据中, 这一过程可能需要与一个对等实体进行网络通信, 而且需要使用 NDIS 函数与适配驱动程序通信。通过透明的消息操作, 如分段与重组、序列化、确认和重传, TDI 传送器简化了应用程序的网络通信。

(4) NDIS 库(Ndis.sys) 为适配驱动程序提供了封装, 隐藏 Windows 2000 核心态环境下的具体细节。NDIS 库为适配驱动程序提供支持函数, 而且也为 TDI 传送器的使用提供了函数接口。

(5) NDIS 小端口驱动程序(NDIS miniport driver) 是工作在核心态的驱动程序, 它负责将 TDI 传送器接入特定的网络适配器。NDIS 小端口驱动程序被封装在 Windows 2000 NDIS 库中。这种封装提供了与微软的 Consumer Windows 跨平台的兼容性。NDIS 小端口驱动程序并不处理 IRP, 而是将调用表接口注册到 NDIS 库中, 而 NDIS 库含有指向从库中输出给 TDI 传送器的函数指针。NDIS 小端口驱动程序与网络适配器通信时使用 NDIS 库函数, 这些函数被映射到硬件抽象层(HAL)的函数。

Windows 2000 的网络构架的组件与 OSI/RM 参考模型有对应关系, 但并不精确, 存在一些跨层的组件, 例如, TDI 传送器便对应了传输(第 4)层和网络(第 3)层。

2. 网络 API

Windows 2000 实现了多种网络 API 用于支持传统的应用, 以及兼容工业标准。本节将扼要叙述网络 API, 描述应用程序是怎样使用它们的, 讨论的网络 API 有:

- 命名管道和邮件槽
- Windows 套接字(Winsock)
- 远程过程调用(RPC)
- 公共互连网络文件系统(CIFS)
- NetBIOS

(1) 命名管道和邮件槽

命名管道(named pipe)和邮件槽(mailslot)是微软起初为 OS/2 局域网管理器开发的编程 API, 随后移植到 Windows NT。命名管道提供可靠的双向通信, 然而, 邮件槽只提供不可靠的单向通信, 邮件槽的一个优点在于它具有广播能力。在 Windows 2000 中, 以上两种 API 都利用了 Windows 2000 的安全特性, 这样就能让服务器精确地控制哪些客户可以连接它。

名称服务器依照 Windows 2000 通用命名规范(UNC)为命名管道和客户指定名称, UNC 是 Windows 网络中用于定位资源的, 独立于协议的方法。

1) 命名管道的操作

命名管道通信由命名管道服务器和命名管道客户组成。命名管道服务器是创建命名管道、让用户连入的应用程序。命名管道的名字格式为 \\ ServerXPipe \ PipeName。其中 Server 指定了执行命名管道服务器的计算机名, 此计算机名可以是 DNS 名称(例如 mspress.microsoft.com), NetBIOS 名称(mspress), 或者是 IP 地址(255.0.0.0)。格式中的 Pipe 就是字符串 Pipe”, 而 PipeName 是给命名管道指定的惟一名称。此惟一名称可以包含子目录, 例如, \\ MyComputer \ Pipe \ MyServerApp \ ConnectionPipe。

命名管道服务器使用 CreateNamedPipeWin32 函数来创建命名管道。函数的输入参数之一是命名管道名字的指针, 形式为 \\ .\ Pipe \ PipeName。“ \\ .\ ”是 Win32 为本地计算机定义的别名。其他参数则包括一个可选的安全描述符用于保护对命名管道的访问, 一个用

于指定管道以单向或双向方式工作的标志,一个最大的并发连接数的值,以及一个用于指定管道以字节方式还是消息方式工作的标志。

大多数网络 API 只以字节方式工作,发送方发送的一条消息在接收方可能需要多次接收,然后,从碎片中重建完整的消息。以消息方式工作的命名管道简化了接收方的实现,因为,一次发送意味着一次接收,它们是一一对应的。因此,每次接收完成后,接收方总能得到一条完整的消息,它无需记录前前后后的消息碎片。

以一个特定的名字初次调用 CreateNamedpipe 创建了这个名字的第一个实例,同时建立了所有与之相关的命名管道实例的行为。如果再次调用 CreateNamedPipe,则服务器再创建一个实例,但不能超过第一次调用所指定的最大连接数量。在创建了至少一个命名管道的实例之后,服务器执行 ConnectNamedPipe Win32 函数,用来让已有命名管道和客户建立连接。CreateNamedPipe 既可以同步执行,也可以异步执行,此调用直到客户与实例建立了连接或错误产生才算完成。

命名管道客户使用 Win32CreateFile 和 CallNamedPipe 函数连接服务,指定服务器创建的管道名称。如果服务器已执行了 ConnectNamedPipe 调用,则客户的安全配置文件以及它所请求的对管道的访问权限(读、写)都通过命名管道安全描述符进行验证。如果客户被授予访问命名管道的权限,它就会收到一个代表命名管道连接的客户端句柄,此时服务器完成对 ConnectNamedPipe 的调用。

在建立命名管道的连接之后,客户和服务器可以使用 ReadFile 和 WriteFile Win32 函数从管道中读取和写入管道。命名管道支持同步和异步的传输操作。图 8-14 表示了服务器与客户之间通过命名管道的通信。

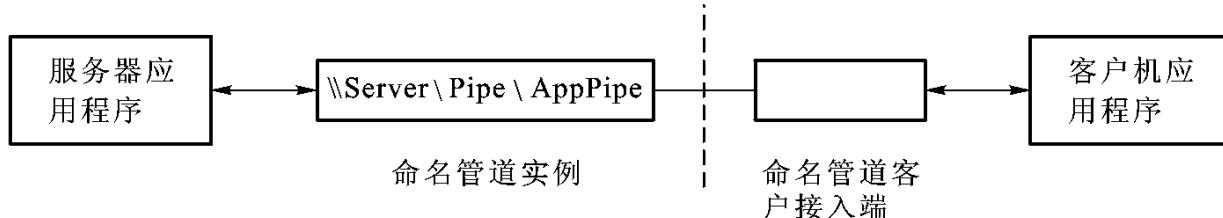


图 8-14 命名管道的通信

命名管道的网络 API 有个特点,就是它允许服务器通过使用 ImpersonateNamedPipeClient 函数扮演客户的角色。

2) 邮件槽的操作

邮件槽提供一种不可靠的,单向广播机制。例如,使用这种通信方式的应用程序可以是时间同步服务,可以每几秒就向域内广播源端报时间。然而,对于网络中的每台计算机并不都需要接收源端时间的消息,因此,使用邮件槽是客户端网络应用的补充。

邮件槽像命名管道一样,与 Win32 API 集成在一起。邮件槽服务器用 CreateMailslot 函数创建一个邮件槽。CreateMailslot 名称的输入格式为 "`*\ Mailslot \ MailslotName`"。其中,邮件槽服务器只能在执行它的机器上创建邮件槽,它的名称能包含子目录,这一点与命名管道相似。CreateMailslot 也可以接收安全描述符用于控制客户对邮件槽的访问。CreateMailslot 返回的句柄会被重叠,即用此函数返回的句柄进行的操作是异步执行的,比如发送和接收消息。

由于邮件槽是单向而且不可靠的,CreateMailslot 并不像 CreateNamedPipe 需要很多参数。在建立了邮件槽之后,服务器仅仅监听到达的客户消息,这一过程使用 ReadFile 函数和此邮件槽的句柄。

邮件槽客户使用类似于命名管道的名称格式,不过针对指定域的广播,它作了相应的改动。为了向一个指定的邮件槽实例发送一条消息,客户调用 CreateFile 时指定了计算机名字。例如"`\Server\ Mailslot \ MailslotName \`"。如果客户所在域中有多个同名的邮件槽服务,但是分布在不同的计算机上,此时客户想要得到一个统一的句柄用来对这些邮件槽进行广播,那它可以以格式"`* \ Mailslot \ MailslotName`"来指定服务器名称。当客户不在此域中,则可以以格式"`\Domain \ Vmailslot\ MailslotName`"进行广播。

在获得客户端的邮件槽句柄之后,客户可以调用 WriteFile 发送消息、但是邮件槽对广播消息有长度小于 425 字节的限制。如果一个消息大于 426 字节,邮件槽就使用点对点的可靠的通信机制,这也就等于失去了广播能力。邮件槽的另一个奇特之处是,长度为 425 或 426 字节的消息会截短为 424 字节。所以,Windows 2000 并不支持 425 和 426 字节的邮件槽消息。图 8-15 说明一个客户对多个邮件槽服务器进行广播的方式。

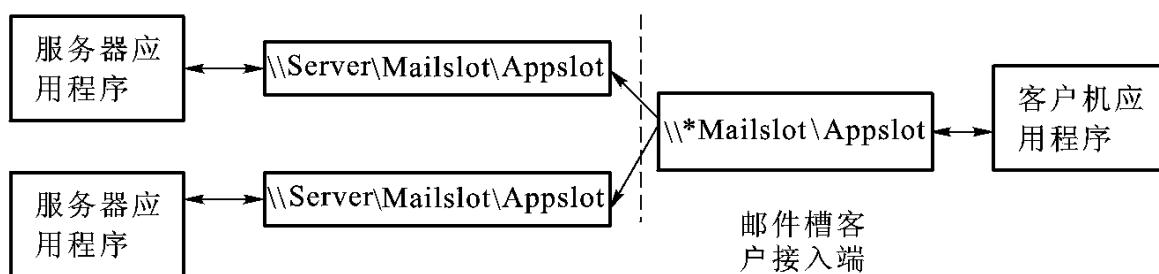


图 8-15 邮件槽的广播

3) 命名管道和邮件槽的实现

由于命名管道和邮件槽与 Win32 紧密地结合在一起,因此,它们都实现在 Kernel32.dll 的 Win32 客户端 DLL 中。应用程序使用 ReadFile 和 WriteFile 函数来发送和接收命名管道和邮件槽的消息,这些函数是基本的 Win32 I/O 例程。用于创建 CreateFile 函数也是标准的 Win32I/O 例程。然而,由命名管道和邮件槽指定的名称确定了由命名管道文件系统驱动程序 (`\Winnt\System32\Npfs.sys`) 和邮件槽文件系统驱动程序 (`\Winnt\System32\Drivers`) 提供的文件系统。

\ Msfs.sys) 所管理的系统名字空间。命名管道文件系统驱动程序创建了一个名为 \ Device \ NamedPipe 的设备对象和一个相应的符号连接 \ ?? \ Pipe, 邮件槽文件系统驱动程序创建了名为 \ Device \ Mailslot 的设备对象和它的符号连接 \ ?? \ Mailslot。以 \\ .\ Pipe \ ... 和 \\ .\ Mailslot \ ... 为格式的名称, 这些名称通常传给 CreateFile, 它们都带有前缀 \\ .\ 。这些前缀会被译成 \ ??, 这样, 名称可以通过设备对象的符号连接进行解析。CreateNamedPipe 和 CreateMailslot 是两个比较特别的函数, 它们使用相应的本地函数 NtCreateNamedPipeFile 和 NtCreateMailslotFile。

当服务器创建了一个命名管道或邮件槽, 或者当它们被客户打开时, 此机器上相应的文件系统驱动程序(FSD)会被调用。FSD 在核心态实现命名管道和邮件槽的主要原因是, 它们集成在对象管理器的名字空间中, 而且能够使用文件对象来表示已打开的命名管道和邮件槽。这种集成给带来以下好处:

- FSD 使用核心态安全性函数为命名管道和邮件槽实现标准的 Windows 2000 安全性。
- 应用程序能够使用 CreateFile 打开命名管道或邮件槽, 因为, FSD 与对象管理器的名字空间集成在一起。
- 应用程序能够使用 Win32 函数, 如 ReadFile 和 WriteFile, 与命名管道和邮件槽交互。
- FSD 依赖于对象管理器对句柄和引用计数的跟踪, 而这些句柄和引用计数属于命名管道和邮件槽的文件对象。
- FSD 利用原有的子目录特性, 可以实现它们自身的命名管道和邮件槽。

因为, 命名管道和邮件槽名字解析使用重定向器 FSD 进行跨网络通信, 它们间接依赖 CIFS 协议, CIFS 使用 IPX, TCP/IP 和 NetBEUI 协议, 所以只要应用程序支持这些协议之一就可以使用命名管道和邮件槽。

(2) Windows 套接字(Winsock)

Windows 套接字是微软根据 BSD 套接字而实现的, 套接字是一类编程 API, 它自 20 世纪 80 年代起成为 UNIX 系统在因特网上的通信标准。Windows 2000 对套接字的支持使 UNIX 网络应用移植到 Windows 2000 上变得相当快捷。Winsock 包括大多数 BSD 套接字的功能, 同时也增加了一些带有微软特色的增强功能, 而这些功能又在不断地改进。Winsock 既支持不可靠的、面向无连接的通信, 也支持可靠的、面向连接的通信。Windows 2000 提供 Winsock 2.2, 它可以包含在或者作为附加软件安装在 ConsumerWindows 的所有版本中。

Winsock 包含以下几个特点: 支持拆分重组和异步模式的应用程序 I/O 操作; 定义了服务质量(QoS)的标准, 当下层网络支持 QoS 的时候, 应用程序可以协商延迟和带宽要求; 扩展性好, 除了必须支持 Windows 2000 所指定的协议以外, Winsock 还能使用其他协议; 除了那些由 Winsock 应用程序所使用的协议定义的名字空间以外, 它还支持集中的名字空间。一个服务器可以在活动目录内发布它的名称, 例如, 使用名字空间扩展, 客户可以在活动目录内

查找服务器的地址。

1) Winsock 的操作

调用初始化函数,完成对 Winsock API 的初始化之后,Winsock 应用的第一步就是创建一个套接字,它表示一个通信端点。一个套接字必须绑定到本机地址,因此,绑定是应用操作的第二步。Winsock 是一种独立于协议的 API,所以,它的地址可以指定为安装在系统上的任何协议的地址,当然,此系统所安装的协议是 Winsock 可操作的。在绑定完成之后,服务器和客户的执行步骤有所不同,对应面向连接和面向无连接的套接字操作步骤也不一样。

一个面向连接的 Winsock 服务器在套接字上执行 listen 操作,指明此套接字可以支持的连接数量。然后,再执行 accept 操作,让客户连接套接字。如果有一个等待的连接请求,accept 调用可以立即返回;否则,只有当连接请求到达时该调用才能完成。当一个连接已经创建之后,accept 函数返回一个新的套接字,它代表此连接的服务器端。此服务器可以使用诸如 receive 和 send 函数来执行发送和接收操作。

面向连接的客户使用 Winsock 的 connect 函数连接到服务器,此函数需要指定一个远程地址。当一个连接建立的时候,客户可以在它的套接字上发送和接收消息。图 8-16 说明了 Winsock 客户与服务器之间的面向连接的通信。

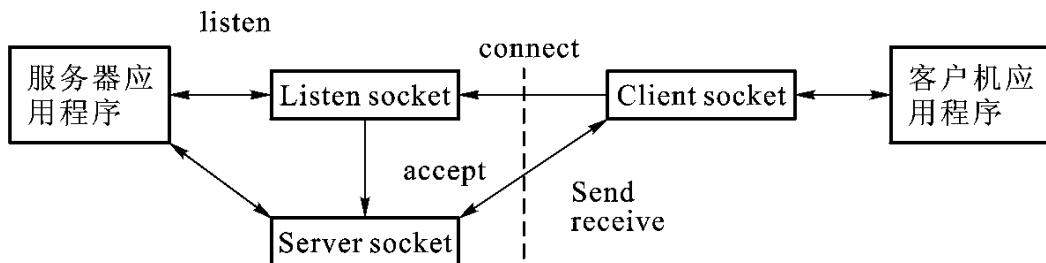


图 8-16 面向连接的 Winsock 操作

面向无连接的服务器在绑定地址之后,与面向无连接的客户没有任何区别:它只需在每条消息中指定远端的地址便可以发送和接收消息。当使用无连接的消息时,称这种消息为数据报,发送方通过下一次接收操作所返回的错误码,可以得知这次发送的消息对方没有收到。

2) Winsock 的可扩展性

一种 Windows 编程观点认为 Winsock API 与 Windows 消息集成在一起是一大特点。Winsock 应用可以利用这一特性执行异步套接字操作,以及通过标准 Windows 消息或回调函数的执行来接收操作完成时的通知。这一功能简化了 Windows 应用的设计,因为,这一类应用不必是多线程或者管理同步对象,这一特性将不再需要执行网络 I/O 操作与从窗口管理器响应用户输入及请求来刷新用户窗口。基于消息版本的 BSD 风格的 Winsock 函数名称一般以前缀 WSA 开头,例如,WSAAccept。

除了支持 BSD 套接字的函数,微软还添加了少许非 Winsock 标准的函数。其中的两个是 AcceptEx 和 TransmitFile,许多 Windows 2000 的 Web 服务器使用这两个函数来获取高性能。AcceptEx 是 accept 函数的一个版本,前者返回客户的地址和第一条消息,而后者仅仅处理与客户建立连接。使用 AcceptEx 函数,Web 服务器可以避免执行多次 Winsock 函数,相反,使用 accept 就不行。

在与客户建立了连接之后,Web 服务器通常发送一个文件给客户,例如网页。由于 TransmitFile 函数的实现与 Windows 2000 缓存管理器集成在一起,因此,客户能够直接从文件系统的缓存中发送一个文件。以这种方式发送一个文件称之为零拷贝,因为服务器没有必要为了发送文件而接触它自身,服务器只需指定一个指向文件的句柄和所传数据的范围。另外,TransmitFile 允许一个服务器在文件之前预先挂起数据或者在文件之后追加数据,这样服务器就可以发送头部信息,这些头部信息包含 Web 服务器的名称,以及一个指明消息大小的域。

3) 扩展 Winsock

在 Windows 2000 上,Winsock 是一种可扩展的 API,因为第三方厂商可以添加传输服务提供者将其他协议接入 Winsock,另外,还有名字空间服务提供者支持 Winsock 名字解析工具。利用 Winsock 的服务提供者接口(SPI)将服务提供杆接,KWinsock。当传输服务提供者在 Winsock 中注册后。Winsock 利用传输服务提供者实现套接字函数,它针对提供者的地址类型实现,例如 connect 和 accept。对于传输服务提供的实现没有任何限制,但是,一般包括核心态中与传输驱动程序之间的通信。

任何一个 Winsock 的客户/服务器应用程序都要求服务器的地址信息对客户来说总是可以得到的,这样客户就能连接到服务器。运行在 TCP/IP 上的标准服务器使用一些众所周知的地址,这样客户很容易连接这些服务器。只要浏览器知道运行 Web 服务器的计算机名称,它就能通过指定公开的 Web 服务器地址(IP 地址加上:80,80 是 HTTP 的端口号)连接 Web 服务器。名字空间服务提供者使某些服务器能以其他方式登记它们的名称。例如,在服务器端,一个名字空间提供者可以在活动目录中登记服务器的地址,而在客户端,它可以在活动目录内查询服务器的地址。名字空间服务提供者实现了标准 Winsock 名字解析函数,如 gethostbyaddr, getservbyname 和 getservbyport, 使用这些函数使得上述功能成为现实。

4) Winsock 的实现

Winsock 的实现如图 8-17 所示。它的应用程序接口由一个 API DLL 组成(Ws2_32.dll),它为应用程序提供 Winsock 函数 Ws2_32.dll 调用名字空间和传输服务提供者实施名称或消息操作。Msafd.dll 库作为微软支持 Winsock 协议的传输服务提供者,它利用 WinsockHelper 库与核心态的协议驱动程序进行协议细节相关的通信。例如,Wshtcpip.dll 是 TCP/IP Helper,Wshnetbs.dll 是 NetBEUI Helper。微软 Winsock 的扩展函数由 Mswsock.dll 实

现,如 TransmitFile, AcceptEx 以及 WSARecvEx 等。Windows 2000 附带 TCP/IP、NetBEUI、AppleTalk、IPX/SPX、ATM 以及 IrDA(红外数据协会)的 Help DLL,还有 DNS(TCP/IP)、活动目录和 IPX/SPX 的名字空间服务提供者。

如同命名管道和邮件槽 API,Winsock 与 Win32 I/O 模型集成在一起,并且利用文件句柄表示套接字。这种特性得益于核心态的文件系统驱动程序的帮助,因此 Msafd.dll 使用辅助函数驱动程序 Afd.sys 实现基于套接字的函数。AFD 是一个 TDI 客户,它通过发送 TDI IRP 到协议驱动程序来执行网络套接字操作,例如发送和接收消息。AFD 并不是针对某一种特定的协议驱动程序设计;然而,Msafd.dll 告诉 AFD 每个套接字所使用的协议名称,这样,AFD 就能够打开代表协议的设备对象。

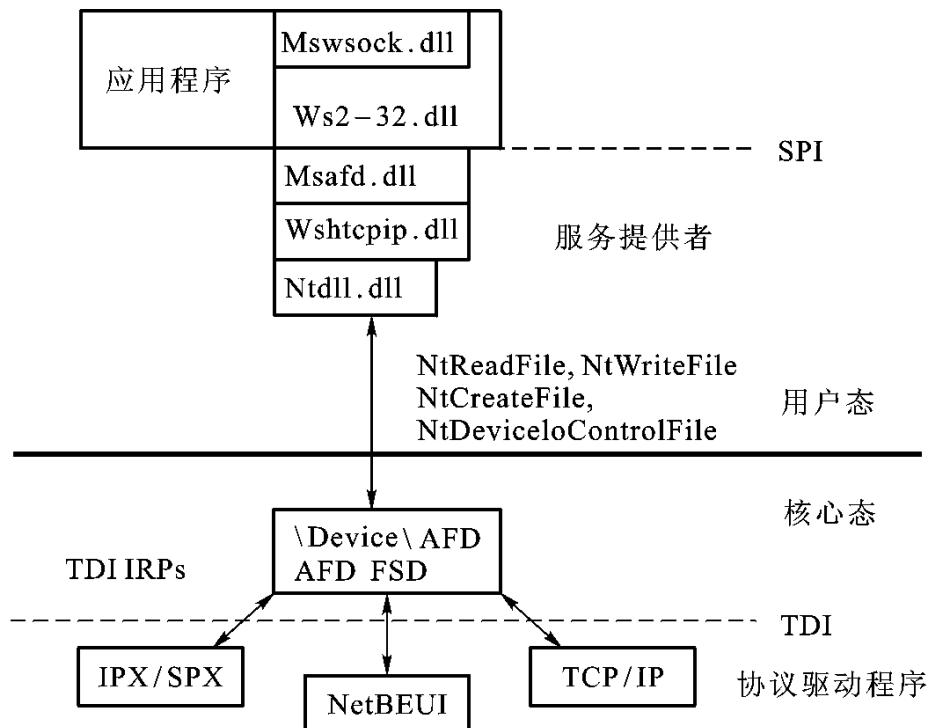


图 8-17 Winsock 的实现

(3) 远程过程调用

远程过程调用(RPC)是一种网络编程标准,它起源于 20 世纪 80 年代早期。开放软件基金会 OSF 使 RPC 成为分布式计算环境(DCE)的分布式计算标准。虽然 SunRPC 是另一个 RPC 的标准,但是微软 RPC 的实现兼容 OSF/DCE 标准。RPC 建立在其他的网络 API 上,这样就能提供另一类编程模型,它将应用程序开发人员从网络编程的细节中解脱出来,即它隐藏了网络的实现。

1) RPC 的操作 RPC 工具让程序员能创建任一个包含若干过程的应用,其中一些过程可以本地执行,而另一些可以跨网络远程执行。它提供一种网络操作的过程式视角,而不是以传输为中心视角,这样也就简化了分布式应用的开发。

传统网络软件的结构以 I/O 处理模型为主。例如,在 Windows 2000 里,应用程序对远程 I/O 的请求会产生网络操作。操作系统将请求转发到重定向器上,重定向器作为远程文件系统,使客户与远程文件系统的交互对客户是不可见的。重定向器将操作传给远程文件系统,远程文件系统完成请求并返回结果之后,本地网卡产生中断。随后,核心处理中断,原先激发的 I/O 操作至此已全部完成,然后,返回结果给调用者。

RPC 的操作方式却完全不同。就像其他结构化应用程序一样,RPC 程序有自己的主程序,它调用过程或过程库来执行一些具体任务。RPC 应用与常规应用的差别在于 RPC 应用中所使用的一些过程库在远程计算机上执行,而其他应用在本地执行过程,如图 8-18 所示。

对于 RPC 应用程序,所有过程在形式上都是本地执行的,程序员无需编写任何代码来传输跨网络的计算机或 I/O 相关的请求,以及选择网络协议,处理网络错误,等待结果等。RPC 软件自动完成了这些工作。Windows 2000 RPC 工具能够在系统中加载的任何传送器之上操作。

为了编写 RPC 应用,程序员决定哪些过程需要本地执行,哪些需要远程执行。例如,假设一个普通的工作站与 Cray 超级计算机或者专用的高速向量机之间存在网络连接,如果程序员编写一个要进行大矩阵群的操作,那么,将数学计算从本机通过 RPC 操作移至远程计算机对于平衡性能负载是非常有意义的。

RPC 的操作以如下方式工作:当应用程序运行时,它调用本地和远程过程。对后者来说,应用程序连接到本地包含桩过程的静态连接库或者 DLL,每个桩过程对应一个远程过程。对一些简单应用,桩过程与应用静态连接,而对一些大组件,桩过程都包含在独立的 DLL 中。在 DCOM 中,后者的连接方式被广泛地采用。桩过程与远程过程有相同的名称,使用相同的接口,然而,桩过程并不执行所要求的操作,它只是将应用程序传入的参数进行参数打包,然后再送至网络上传送。所谓参数打包就是将参数排序后以一种特殊的方式打包成适应网络连接的格式,例如,解析引用,挑选指针所引用的数据结构的一个拷贝等。

桩过程调用 RPC 运行时的过程来定位远程过程所寄居的计算机,确定远程计算机使用哪种传送机制,利用传输软件发送请求给它。当远程服务器收到 RPC 请求之后,它对参数解包,重建原有的过程调用,然后,调用此过程。当服务器完成调用之后,它执行相反的操作将结果返回调用者。

除了上述提到的基于同步函数调用的接口以外,Windows 2000 RPC 还支持异步 RPC。

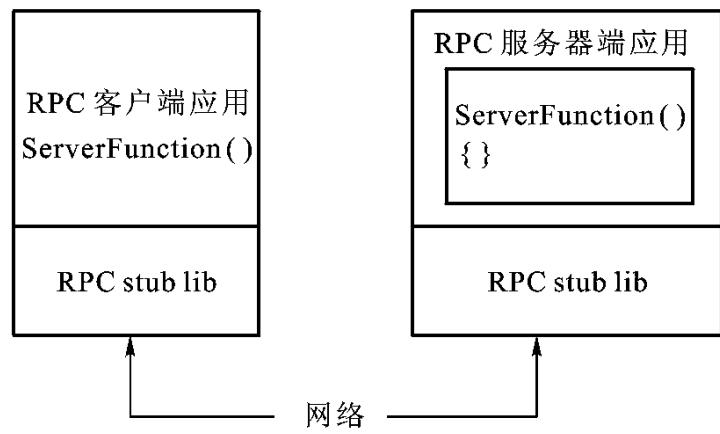


图 8-18 RPC 的操作

异步 RPC 使 RPC 应用程序执行一个函数却不必等它执行完成, 就可以继续应用程序的运行。而应用程序在取得来自服务器的应答之后, 可以在之后的其他代码中执行相应的程序, RPC 运行库产生一个事件对象, 客户通过它与异步调用相关联。客户能够使用标准的 Win32 函数探测函数是否完成, 如 WaitForSingleObject。

除了 RPC 运行库以外, 微软的 RPC 工具还有一个编译器, 叫做微软接口定义语言 (MIDL) 编译器。MIDL 编译器简化了 RPC 应用程序的开发。程序员只需编写一系列常规函数原型(假定是 C 或 C++ 应用函数原型)用于描述远程例程, 然后, 将这些原型存入一个单独的文件。随后, 程序员给原型添加了一些额外的信息, 如在网络中的惟一标识, 它用于例程的打包以及确定版本号, 然后, 再加上一些属性指明参数是输入、输出还是两者都是。这些经过修饰的原型组成了开发人员的接口定义语言文件。

一旦 MIDL 文件创建完毕, 程序员用 MIDL 编译器编译此文件, 生成了客户端和服务器端的桩例程, 另外, 还有一些在应用程序中使用的头文件。当客户端应用连接到桩例程文件的时候, 所有的远程过程引用被解析。然后, 远程过程利用相似的过程在服务器上进行安装。程序员调用已有的 RPC 应用程序时只需编写客户端软件, 将应用程序连接到本地的 RPC 运行时工具就行了。

RPC 运行库使用 RPC 传输提供者接口与传输协议对话, 提供者接口作为 RPC 工具与传送器之间的过度层, 它将 RPC 操作映射到传送器提供的函数。Windows 2000 RPC 工具为命名管道、NetBIOS 和 TCP/IP 实现了传输提供者 DLL。你可以编写新的提供者 DLL 支持其他传送器。

大多数 Windows 2000 的网络服务器是 RPC 应用程序, 即本地进程和远程计算机的进程都可以调用它们。这样, 远程客户计算机可以调用服务器服务来列出共享清单, 打开文件, 写入打印队列, 或在你的服务器上激活用户, 或者它能够调用信使服务器给你直接发送消息等。

服务器名称发布, 是一个服务器在客户能够查询的位置进行注册名称的能力, 它属于 RPC 并且与活动目录集成在一起。如果活动目录没有安装, RPC 名称定位器服务器将退回 to NetBIOS 广播。这种行为保证了与 WindowsNT4 的交互操作, 而且使 RPC 在独立的服务器和工作站上也能发挥作用。

2) RPC 的安全性

Windows 2000 RPC 与安全支持提供者 (SSP) 集成在一起, 服务器可以使用验证或加密的通信。当 RPC 服务器要建立安全通信, 它必须在 SSP 内登记 SSP 相关的主名称。当客户绑定到一个服务器时, 先登记它的安全证书, 然后, 指定服务器的主名称。在绑定的时候, 客户还指定了它所要的验证级别。各种不同的验证级别保证了只有经过授权的客户才能连接到服务器, 确认服务器收到的任何一条来自授权用户的消息, 校验 RPC 消息的完整性, 看其是否被操作过, 甚至对 RPC 消息的数据进行加密。显然, 验证级别越高, 处理开销越大。

SSP 处理网络通信验证与加密的一些细节问题,除了 RPC 以外还有 Winsock。Windows 2000 包括大量内置的 SSP,其中有 Kerberos SSP 实现了 Kerberos 5 验证版本,还有 Secure Channel (SChannel) 实现了 SecureSocketsLayer(SSL),TransportLayerSecurity(TLS) 协议,以及私有通信技术(PCT)。如果没有指定 SSP,RPC 软件将使用命名管道的内置安全性。

RPC 安全性的另一个特点就是,利用 `RpcImpersonateClient` 函数,服务器就有扮演客户安全性标识的能力。在完成扮演客户的操作之后调用 `RpcRevertToSelf` 或者 `RpcRevertToSelfEx`,它恢复为自己的安全性标识。

3) RPC 的实现

图 8-19 描绘了 RPC 的实现方式,一个基于 RPC 的应用程序与 RPC 运行时 DLL 相连接 (\Winnt\System32\Rpcrt4.dll)。除了提供相应的发送、接收打包数据的函数,RPC 运行时 DLL 还为应用程序的 RPC 函数桩提供参数打包和参数解包的函数。RPC 运行时 DLL 包含一些例程支持网络 RPC 的处理,而不仅仅是本地 RPC 的形式。本地 RPC 可以使在同一个系统上的两个进程进行通信,而且 RPC 运行时 DLL 使用核心态的本地过程调用(LPC)工具作为本地网络 API。当 RPC 是基于非本地通信机制的时候,RPC 运行时 DLL 使用 Winsock、命名管道或者消息队列 API。

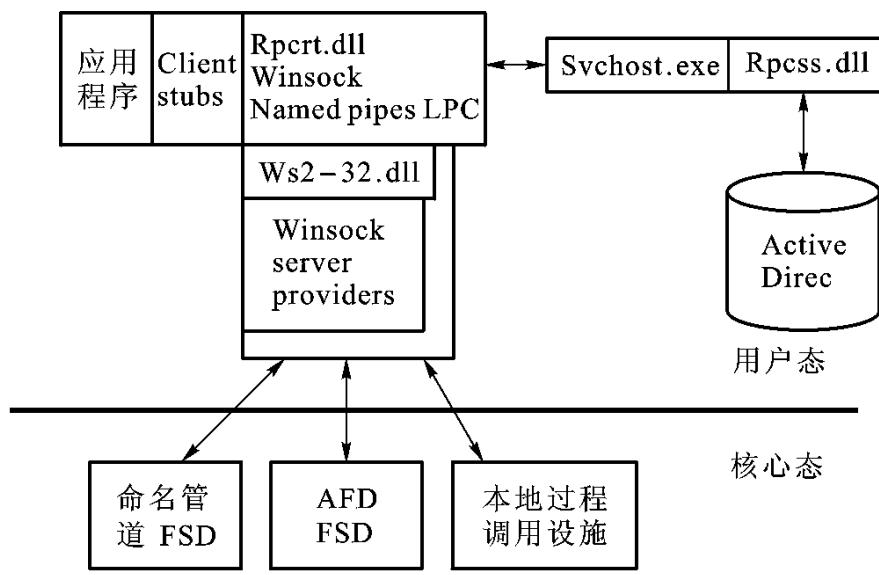


图 8-19 RPC 的实现

对名称注册和查询来说,RPC 应用程序与 RPC 名称服务 DLL 连接 (\Winnt\System32\Rpcns4.dll)。此 DLL 与 RPC 子系统(RPCSS – \Winnt\System32\Rpcss.dll)进行通信,RPC 子系统是以 Win32 的服务的形式实现的。RPCSS 本身也是一个 RPC 应用程序,它与其他子系统上的 RPCSS 通信,处理名称查询和注册。

(4) 通用互联网文件系统

通用互联网文件系统(CIFS)作为服务器消息块(SMB)协议的增强形式,是 Windows 2000

用于互联网文件共享的协议。因为,应用程序通过标准的 Win32 文件 I/O 函数访问远程文件,所以应用程序不直接使用 CIFS 协议,但是 CIFS 负责处理这种远程 I/O 调用请求。CIFS 定义了打印机共享规范,因此,Windows 2000 自身也使用 CIFS 进行打印机共享。尽管 CIFS 自身不是一个 API,在这里提到它是因为文件和打印机共享建立在 CIFS 基础之上,同时它通过 Win32API 向应用程序展现。

CIFS 作为一个公开的微软标准允许第三方与 Windows 2000 文件服务器以及 Windows 2000 文件共享客户之间进行互操作。比如 Samba 共享软件套件使 UNIX 操作系统可以向 Windows 2000 客户提供文件服务,同时使 UNIX 环境的应用程序可以访问 Windows 2000 上的文件。其他支持 CIFS 的平台还有 DEC VMS 和 Apple Macintosh。

Windows 2000 上的文件共享基于重定向器 FSD。重定向器 FSD 运行在客户机上,它与运行在服务器上的服务器 FSD 进行通信。重定向器 FSD 截获指向服务器上文件的 Win32 文件 I/O 调用请求,向服务器的文件系统发送相应的 CIFS 消息去执行客户端的请求。

重定向器 FSD 和服务器 FSD 与 Windows 2000 的 I/O 系统是整合在一起的,这样的实现比起在用户空间的文件服务器实现有很多优点:可以直接与 TDI 传输口和本地 FSD 交互;它们可以和缓存管理器整合在一起,在客户端无缝地缓存服务器上的文件;应用程序可以使用标准的 Win32 文件 I/O 函数访问远程文件,比如 CreateFile, ReadFile 和 WriteFile。

Windows 2000 的重定向器 FSD 和服务器 FSD 遵从标准网络资源命名规范,所有核心态的文件服务器和客户端均遵从这一规范。如果一个远程文件共享用一个驱动器号来连接,网络文件的命名就和本地文件相同。此外,重定向器 FSD 还支持 UNC 命名。

1) CIFS 的实现

和许多其他设备驱动类型一样,Windows 2000 的重定向器 FSD 采用端口/小端口模型。微软提供了名为 \Winnt\System32\Drivers\Rdbss.sys 的重定向器 FSD 端口库。开发者可以针对该库编写自己的小端口驱动程序。重定向器 FSD 端口库隐藏了实现一个重定向器 FSD 的许多细节,比如重定向器 FSD 和缓存管理器、内存管理器、TDI 传输口的整合。CIFS 小端口驱动程序的文件名为 \Winnt\System32\Drivers\Mrxsmb.sys。

Mrxsmb.sys 使用了前面讲述的缓存管理器来缓存文件数据并进行智能预读。CIFS 的小端口驱动程序通过 TDI API 向远程服务器发送 CIFS 命令。它可以使用任何支持 TDI API 的传输方式,比如 NetBEUI、NetBT(基于 TCP/IP 的 NetBIOS)和 TCP/IP。

在充当文件服务器角色的系统上,服务器 FSD(\Winnt\System32\Drivers\Srv.sys)监听发自客户机的 CIFS 命令,同时作为代理接口访问服务器本地 FSD。通过本地 Windows 2000 FSD 实现的文件系统接口,服务器 FSD 实现了零拷贝发送的能力。文件系统接口允许服务器 FSD 获得描述一个内存描述子列表(MDL)。这个列表描述了暂存在文件系统缓存的文件数据。服务器 FSD 可以把 MDL 传给 TDI 传输口,TDI 传输口使用网络适配器驱动程序

进行网络传输。如果没有本地 FSD 缓存管理器的支持,服务器 FSD 就把文件数据拷贝到自己的缓存中,随后把它传给 TDI 传输口。

服务器 FSD 和重定向器 FSD 有相应的 Win32 服务:服务器服务和工作站服务。这些服务运行在服务控制管理器(SCM)进程中,它们提供了对相应驱动程序的管理接口。

2) 分布式文件缓存

如果只有一个客户访问服务器的某一文件,显然,在客户端缓存文件数据是安全的。当两个客户同时访问一个文件时,必须采取一定的措施保证客户和服务器数据文件的一致。Windows 2000 采用机会锁(oplock)的机制来解决分布式缓存的一致性问题。当一个客户访问服务器文件时,它必须首先请求机会锁。服务器授予客户的机会锁的类型决定了客户能采用的缓存方式。有三种主要的机会锁类型:

- I 级机会锁 用于一个客户独占一个文件。拥有这种类型机会锁的客户能在客户端对相应文件进行读写缓存。

- II 级机会锁 表示文件共享锁。拥有 II 级机会锁的客户可以对相应文件进行缓存读,但是任何对文件的写操作会使 II 级机会锁失效。

- 批处理机会锁 是有最大许可权的机会锁。拥有这种类型机会锁的客户不仅能在客户端对相应文件进行读写缓存,同时打开关闭文件不需要额外请求机会锁。批处理机会锁通常用来支持批处理文件的执行,在执行时通常需要反复打开关闭一个文件。

如果一个客户没有机会锁,那么,它就不能在本地进行缓存读或缓存写,而是直接从服务器接收数据或者把所做的所有修改直接发往服务器。

(5) NetBIOS

直到 20 世纪 90 年代,网络基本输入输出系统(NetBIOS)编程 API 是 PC 上使用最广的网络编程 API。NetBIOS 支持可靠的面向连接的通信和非可靠的无连接的通信。Windows 2000 为了运行基于 NetBIOS 的遗留程序,提供对 NetBIOS 的支持。微软建议应用程序开发者不要再使用 NetBIOS,因为,其他 API,比如命名管道和 Winsock,与之相比更灵活更可移植。在 Windows 2000 上是通过 TCP/IP、NetBEUI 和 IPX/SPX 协议来支持 NetBIOS 的。

1) NetBIOS 名称

在 NetBIOS 的命名规范中,所有的计算机和网络服务器被分配了 16 字节长的名称,称为 NetBIOS 名称。NetBIOS 名称有两种类型:UNIQUE 和 GROUP。对于 UNIQUE 类型的 NetBIOS 名称,网络中只能有一个实例;而对于 GROUP 类型的 NetBIOS 名称则可以有多个网络程序拥有该名称。客户可以通过广播向一个 GROUP 的所有成员发消息。在 Windows 的网络服务器中,NetBIOS 名称的第 16 个字节是保留的,表示该名称的资源类型。

为了支持与 Windows NT4 操作系统的互操作,Windows 2000 为每一个域定义了一个 NetBIOS 名称。它是域的 DNS 名称的前 15 个字节。比如名为 mspress.Microsoft.com 的域,它

的 NetBIOS 名称为 mspress。同样,在安装每一台机器时 Windows 2000 需要管理员为它们分配相应的 NetBIOS 名称。

另一个 NetBIOS 中涉及的概念是 LANA 号。每块网络适配器上的每一个 NetBIOS 兼容协议都会分配一个 LANA 号。比如,一台机器有两块网卡,每块网卡都支持 TCP/IP 和 NetBEUI 协议,那么,这台机器就有四个 LANA 号。LANA 号十分重要,因为,基于 NetBIOS 的应用程序必须显式地把服务器名称分配到每一个可接受相应用户连接的 LANA 号上。如果应用程序监听特定名称的用户连接,客户只能通过那些注册了该名称的 LANA 号,即相应网卡上的 NetBIOS 兼容协议来访问应用程序。

一个名为 Windows 因特网名字服务(Windows InternetName Service, WINS)的网络服务器维护 NetBIOS 名称和 IP 地址的映射。如果 WINS 没有安装,NetBIOS 使用名称广播在 Windows 网络中传播 NetBIOS 名称。值得注意的是,NetBIOS 名称排在 DNS 名称之后。计算机名称首先通过 DNS 注册并被解析,只有当 DNS 域名解析失败之后,Windows 2000 才进行 NetBIOS 名称的解析。

2) NetBIOS 的操作

NetBIOS 服务器应用程序通过 NetBIOS API 枚举所在系统的所有 LANA,把表示应用程序的 NetBIOS 名称分配到每个 LANA。如果服务器是面向连接的,应用程序执行 NetBIOS 监听命令,等待客户的连接请求。与客户建立连接后,服务器应用程序执行 NetBIOS 命令进行数据的发送和接收。连接的服务是类似的,只不过服务器应用程序不用建立连接就可以接收消息。

面向连接的客户使用 NetBIOS 命令与服务器建立连接,此后调用其他命令发送和接收数据。一个建立的 NetBIOS 连接是一个会话(session)。如果客户想发送无连接的消息,它只需在发送命令中指明服务器应用程序的 NetBIOS 名称就可以了。

NetBIOS 由一系列命令构成,不过这些命令通过一个单一的 NetBIOS 接口被调用。这种调用方案是由遗留程序造成的。在 MS - DOS 上 NetBIOS 是通过 MS - DOS 中断服务实现的。一个 NetBIOS 应用程序执行一个 MS - DOS 中断,向 MS - DOS 的 NetBIOS 服务器发送一个数据结构。这个数据结构表示了将被执行的 NetBIOS 命令的所有细节。因此,在 Windows 2000 下,Netbios 函数只接受一个参数。它是一个数据结构,包含了应用程序调用 NetBIOS 服务器的所有参数。

3) NetBIOS API 的实现

NetBIOS 函数通过 \Winnt\System32\Netapi32.dll 提供给应用程序。Netapi32.dll 打开一个名为 NetBIOS 仿真驱动程序(NetBIOS emulation driver)的核心态驱动程序的句柄(\Winnt\System32\Drivers\Netbios.sys),并代表应用程序发出 Win32 DeviceIoControl 命令。NetBIOS 仿真驱动程序把收到的 NetBIOS 命令转换成 TDI 命令发送给相应的协议驱动程序。

如果一个应用程序使用基于 TCP/IP 的 NetBIOS, 那么, NetBIOS 仿真驱动程序需要有 NetBT 驱动程序 (\Winnt\kSystem32\Drivers\Netbt.sys)。NetBT 是基于 TCP/IP 的 NetBIOS 驱动程序, 它负责利用 TCP/IP 协议实现 NetBIOS 的语义。最初 NetBIOS 的语义是 NetBEUI 协议固有的, 比如 NetBIOS 依赖 NetBEUI 的消息模式传输和 NetBIOS 名字解析, 所以, 这都需要 NetBT 驱动程序利用 TCP/IP 加以实现。同样, NwLinkNB 驱动程序通过 IPX/SPX 协议实现 NetBIOS 的语义。

3. 网络资源的名字解析

应用程序可以通过两种方法查询和访问远程机器上的资源。一种是使用 UNC 标准, 通过 Win32 函数直接访问远程资源。另一种方法是使用微软网络(WNet) API 枚举所有计算机提供的可共享的计算机和资源。两种方法都使用重定向器访问网络上的资源。正如前面叙述过的, 客户要访问 CIFS 服务器是通过 CIFS 重定向器实现的, 而 CIFS 重定向器包括核心态的重定向器 FSD 和用户态的 Workstaion 服务。微软也提供了访问 Novell NetWare 服务器的共享资源的重定向器, 第三方也可以向 Windows 2000 添加自己的重定向器。在本章将研究当有远程请求时, 决定哪一个重定向器被调用的软件。它们是以下两个组件:

- 多提供者路由器(Multipleproviderrouter, MPR)是一个 DLL。它决定一个应用程序通过 Win32 WNetAPI 访问远程文件时使用何种网络。
- 多 UNC 提供者(MultipleUNCProvider, MUP)是一个驱动程序。它决定一个应用程序通过 Win32I/O API 访问远程文件时使用何种网络。

最后, 将以 Windows 2000 的计算机名字解析系统的核心——域名系统(DNS)结束这一节。

(1) 多提供者路由器

Win32 WNet 函数允许应用程序(包括 Windows 资源管理器、网上邻居应用程序)连接到各类网络资源, 比如文件服务器和打印机, 同时浏览远程文件系统上的各类内容。因为 WNETAPI 的调用可以跨越各类网络, 使用各类网络传输协议, 系统必须使请求正确地经网络发送并理解远程文件系统发回的结果。

提供者(provider)是这样的一类软件, 它们使 Windows 2000 作为一个客户连接到远程网络服务器上。WNet 提供者完成以下一些操作, 包括建立和断开网络连接, 远程打印, 数据传输。内置的 WNet 提供者包括一个 DLL、Workstanon 服务和重定向器。其他网络供应商只需要提供 DLL 和重定向器。

当应用程序调用 WNet 例程时, 调用被直接发送给 MPR.DLL。MPR 接受调用, 确定哪一个提供者可以识别要被访问的资源。MPR 下的每一个提供者 DLL 都提供了被称为提供者接口(provider interface)的一套标准函数集。这个接口可以使 MPR 确定应用程序将要访问哪一个网络并把调用请求发给合适的 WNet 提供者。重定向器 FSD 的提供者是 \Winnt\System32\Ntlanman.dll 这是由注册表 HKLM\SYSTEM\CurrentControlSet\Services\

lanmanworkstation \ NetworkProvJder 主键 ProviderPath 的键值确定的。

当被调用 WnetAddConnection API 函数连接一个远程网络资源的时候, MPR 检查 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Network Provider \ Order \ ProviderOrder 键值, 确定加载哪一个网络提供者。MPR 按照它们在注册表中的次序轮询每一个提供者, 直到有一个相应的重定向器识别这一资源或者所有可用的提供者都被选取。可以通过高级设置对话框改变 ProviderOrder。这个对话框可以通过选取网络和拨号连接应用程序的高级菜单来访问。可以右键点击桌面上网上邻居图标, 选择弹出式菜单的属性或者直接点击开始菜单的设置选项来选取网络和拨号连接应用程序。

WNetAddConnection 函数为每个远程资源分配一个驱动器号或者设备名称。当该函数被调用时, 它把调用传给合适的网络提供者。相应地, 网络提供者在对象管理器名字空间中建立一个符号连接对象。这个对象把定义的驱动器号映射到相应网络的重定向器。

(2) 多 UNC 提供者

多 UNC 提供者(MUP)是一个和 MPR 类似的网络组件。所有针对 UNC 名称的文件或设备的 I/O 请求都由它处理(以“ \\ ”开头的名称表明相应的资源存在于网上)。像 MPR 一样, MUP 接收请求, 确定哪一个本地重定向器可以识别远程资源之一。与 MPR 不同的是, MUP 是一个设备驱动程序, 在系统启动时被载入。

当应用程序企图打开一个 UNC 名称的远程文件或设备时, MUP 驱动程序被激活。当 Win32 客户 DLL Kernel32.dlt(一个提供了文件 I/O API 的 DLL)接收到调用请求时, 该子系统在 UNC 名称前添加 “ \\ ?? \ UNC”的前缀字符串并调用 NtCreateFile 打开这一文件。“ \\ ?? \ UNC”作为符号连接名称被解析成 \\ Device \ Mup, 这是一个表示 MUP 驱动程序的设备名称。

MUP 驱动程序接受调用请求, 此后, 向每一个注册的重定向器发出一个异步的 IRP, 等待它们中的一个能识别要被访问的资源名称并给出回应。当有一个重定向器识别出资源名称后, 它将指出这个名称的哪一部分是它可以确定的。举个例子来说, 比如名为 \\ WIN2KSERVER \ PUBLIC \ insidew2k \ chapl3.doc 的文件, 重定向器识别出它以后会声明字符串 \\ WIN2KSERVER \ PUBLIC 是属于自己的。MUP 驱动程序将缓存这一信息, 所有以那个字符串开头的请求将直接发往那个重定向器, 从而, 省去了轮询过程。MUP 驱动程序的缓存有超时失效的特征, 如果一段时间没被用到, 与一个重定向器相关联的字符串将会失效。如果多个重定向器识别出同一资源, MUP 驱动程序将根据注册表中 ProviderOrder 列出的重定向器的次序决定哪个重定向器优先得到这个 I/O 请求。

(3) 域名系统

域名系统(DNS)是因特网名称转换成 IP 地址的标准。一个网络应用程序如果想把 DNS 名字解析成 IP 地址, 必须通过 TCP/IP 协议向 DNS 服务器发送一个 DNS 查找请求。DNS 服务器使用分布式数据库存储的“名称/IP 对”进行名称到 IP 地址的翻译。每一个服务器负责

一定区域(zone)的翻译。有关 DNS 的详细描述超出了本书的范围,但 DNS 是 Windows 2000 中命名的基础,它是 Windows 2000 最主要的名称解析协议。

Windows 2000 DNS 服务器作为一个 Win32 服务(\ Winnt \ System32 \ Dns.exe)包括在 Windows 2000 的服务器版本中。标准的 DNS 服务器实现使用文本文件作为翻译数据库,但 Windows 2000 DNS 服务器可以改用活动目录存储相应区域的信息。

4. 协议驱动程序

网络 API 驱动程序接受 API 请求,把它们转换为底层网络协议的传输请求,API 驱动程序依赖核心态的传输协议驱动程序进行实际的转换。API 和下层的网络协议是分开的,使得整个网络体系结构十分灵活,它允许每个 API 使用不同的网络协议。Windows 2000 带有 DLC、NetBEUI、TCP/IP 和 NWLink 传输协议的驱动程序。另外一些协议的驱动程序是可选的,比如 Windows 2000 服务器版本安装 ServicesForMacintosh 时安装的 AppleTalk 协议。这里对每一个协议作简单的介绍:

- DLC 协议。IBM 的一些大型机和 HP 的一些网络打印机使用了这一协议。它是一种“原始”的协议,因为,没有一种网络 API 可以使用它。想要使用 DLC 的应用程序必须直接调用 DLC 传输协议设备驱动程序的接口。
- IBM 和微软在 1985 年引入了 NetBEUI,微软把 NetBEUI 作为 LAN 管理器和 NetBIOS API 的默认协议,微软此后增强了 NetBEUI,但这个协议有局限性,它是非路由的,同时在 WAN 上性能很差。NetBIOS 扩展用户接口 NetBEUI(NetBIOS Extended UserInterface),之所以这样命名是因为它和 NetBIOS 紧密集成在一起。微软 NetBEUI 协议驱动程序实现的 NetBEUI 采用 NetBIOS 帧(NBF)格式。Windows 2000 支持 NetBEUI 只是为了和以前的 Windows 操作系统互操作。
- 因特网的飞速发展和对 TCP/IP 协议的依赖使得 TCP/IP 协议成为 Windows 2000 最主要的协议。美国国防部高级研究计划局(DARPA)在 1969 年开发的 ARPANET 是因特网的基础。TCP/IP 有适于 WAN 的特征,比如可路由和在 WAN 上较好的性能。TCP/IP 协议是 Windows 2000 优先使用的协议,也是惟一缺省安装的协议。
- NWLink 协议是由 Novell 的 IPX 协议和 SPX 协议组成的。Windows 2000 中包括 NWLink 协议是为了和 NovellNetWare 服务器互操作。

Windows 2000 的传输口通常实现了上述传输协议的所有相关协议。比如 TCP/IP 驱动程序(\ Winnt \ System32\Drivers \ Tcpip.sys)支持 TCP、UDP、IP、ARP、ICMP 和 IGMP。一个 TDI 传输口通常使用设备对象表示特定的协议。客户获取表示协议的文件对象,使用 IRP 向对应协议发出网络 I/O 请求。TCP/IP 驱动程序创建三个设备对象,表示 TDI 客户会使用的三种协议: \ Device \ TCP、\ Device \ Udp 和 \ Device \ Ip。

网络 API 不必对每一种传输协议使用不同的接口,微软制订了传输驱动程序接口(TDI)

标准,TDI 接口实质上是网络请求如何格式化成 IRP 以及网络地址和通信连接如何分配的规范。支持 TDI 规范的传输协议向 AFD 和重定向器之类的客户提供接口。通过 Windows 2000 设备驱动程序实现的传输协议被称为 TDI 传送器。因为,TDI 传送器是设备驱动程序,它们只处理客户发来的 IRP 格式请求。

在 \Winnt\System32\Drivers\Tdi.sys 库中的支持函数和开发者在自己驱动程序中的定义构成了 TDI 接口。TDI 编程模型和 Winsock 十分相似。一个 TDI 客户执行以下步骤和远程服务器建立连接:

- 客户分配并格式化一个 addressopenTDIIRP 来请求分配一个地址。TDI 传送器返回一个被称为地址对象 (addressobject) 的文件对象。这一步等价于 Winsock 的 bind 函数。
- 客户分配并格式化一个 connection open TDI IRP。TDI 传送器返回一个被称为连接对象的文件对象来表示连接。这一步等价于 Winsock 的 socket 函数。
- 客户用 associate address TDI IRP 把连接对象和地址对象关联在一起(在 Winsock 中没有等价的函数)。
- 接收远程连接的 TDI 客户发出 listenTDI IRP, 该 IRP 指明了服务器端连接对象能支持的最大连接数。同时该 TDI 客户发出 accept IRP。当建立远程连接或者出现错误时, TDI 传送器完成 acceptIRP。这一步等价于 Winsock 的 listen 和 accept 函数。
- 想和远程服务器建立连接的 TDI 客户发出 connect IRP, 该 IRP 指明了连接对象。当建立远程连接或者出现错误时, TDI 传送器完成 connectIRP。这一步等价于 Winsock 的 connect 函数。

TDI 也支持 UDP 等无连接协议使用无连接通信。此外 TDI 还支持这样的方式:TDI 的客户在 TDI 传送器中注册一个事件回调, 当 TDI 传送器从网络上获得数据后, TDI 传送器直接调用事件回调函数。TDI 这种基于事件的回调机制使得 TDI 传送器能够通知它的客户相关的网络事件。使用事件回调的客户在接受网络数据时不必预先分配缓冲等资源, 因为, 它们可以访问 TDI 协议驱动程序缓冲中的数据。

5. NDIS 驱动程序

当一个协议驱动程序在网络上读写相应协议格式的消息时, 必然要使用网络适配器。不可能指望协议驱动程序了解市场上每一块网络适配器的细微差别(有专利的网络适配器就有几千种)。只有靠网络适配器供应商提供驱动程序, 利用驱动程序在他们的硬件上传输和接收网络消息。1989 年, 微软和 3COM 联合开发了网络驱动程序接口规范 (NDIS), 它允许协议驱动程序以与设备无关的方式和网络适配器驱动程序通信。遵守 NDIS 的网络适配器驱动程序被称为 NDIS 驱动程序或 NDIS 小端口驱动程序。Windows 2000 中的 NDIS 版本是 NDIS 5。

在 Windows 2000 中, NDIS 库 (\Winnt\System32\Drivers\Ndis.sys) 实现了 TDI 传送器

和 NDIS 驱动程序的边界。和 Tdi.sys 一样, NDIS 库是一个辅助库, NDIS 驱动程序的客户使用这个库格式化发向 NDIS 驱动程序的命令。NDIS 驱动程序通过 NDIS 库接收请求或者发回响应。

微软的网络体系结构的设计目标之一就是使适配器供应商能够很容易地开发 NDIS 驱动程序并很容易地在 ConsumerWindows 和 Windows 2000 之间进行移植。除了提供 NDIS 边界辅助例程, NDIS 库为 NDIS 驱动程序提供了完整的执行环境。如果没有 NDIS 库对它们的封装, 它们将不能被调用。因此, NDIS 驱动程序不是真正的 Windows 2000 驱动程序。NDIS 驱动程序被隔离层完全地包装, 不能直接接收和处理 IRP。NDIS 库从 TDI 服务器接收 IRP, 把它们转换成对 NDIS 驱动程序的调用。在驱动程序完成上次调用之前, NDIS 库可能向驱动程序发出新的调用, NDIS 库保证 NDIS 驱动程序不必担心这种可重入性问题。没有可重入性问题意味着在编写 NDIS 驱动程序时不必考虑复杂的同步问题。在多处理器并行处理的环境下, 这可能是一个非常棘手的问题。

尽管 NDIS 库对 NDIS 驱动程序的序列化简化了开发, 但序列化会影响多处理器可扩展性。在多处理器环境, 标准的 NDIS 4 驱动程序 (NDIS 库的 WindowsNT4 版本) 对某些操作的可扩展性不是很好。在 NDIS 5 中, 微软允许开发者对操作反序列化。NDIS 5 驱动程序可以向 NDIS 库表明它们不想被序列化, 那么, NDIS 库将把收到的 IRP 请求直接发给驱动程序。这时 NDIS 驱动程序必须负责对请求的排队和并发管理。在多处理器环境中, 这种反序列化可获得更好的性能。

(1) NDIS 小端口的变体

NDIS 模型也支持被称为 NDIS 中间驱动程序 (NDIS intermediatedriver) 的混合 TDI 传送器 NDIS 驱动程序。这类驱动程序处于 TDI 传送器和 NDIS 驱动程序之间。对于 NDIS 驱动程序, NDIS 中间驱动程序看上去像一个 TDI 传送器, 对于 TDI 传送器, NDIS 中间驱动程序看上去像一个 NDIS 驱动程序。因为, NDIS 中间驱动程序处在协议驱动程序和网络驱动程序中间, 它可以处理系统所有网络通信的数据分组。支持网络适配器故障容错和负载均衡的软件, 比如微软网络负载均衡提供者, 就要依靠 NDIS 中间驱动程序。QoS 中的分组调度器也是 NDIS 中间驱动器的一个例子。

(2) 面向连接的 NDIS

NDIS 5 引入了新的一类 NDIS 驱动程序——面向连接的 NDIS 小端口驱动程序。由此 Windows 2000 天生具备了对面向连接的网络硬件 (比如 ATM) 的支持, Windows 2000 网络体系结构对连接的管理和建立也有了标准。面向连接的 NDIS 使用许多和标准 NDIS 相同的 API, 但面向连接的 NDIS 驱动程序通过已建立的网络连接发送数据分组, 而不只是简单地把数据分组放到网络媒介上。

除了支持面向连接的网络媒介的 NDIS 小端口驱动程序, NDIS 5 也定义了一些驱动程序

支持面向连接的小端口驱动程序。

- 通话管理器是一种 NDIS 驱动程序,向面向连接的客户提供通话建立和挂断服务。通话管理器使用面向连接的小端口驱动程序和其他网络实体,比如网络交换机或其他通话管理器,交换信号信息。一个通话管理器支持一个或多个信号协议,比如 ATM 用户网络接口 3.1(UNI)。

- 一个整合的小端口通话管理器(MCM)是一个面向连接的小端口驱动程序。它向面向连接的客户提供了通话管理器服务。MCM 实质上是内置通话管理器的 NDIS 小端口驱动程序。

一个面向连接的客户使用通话管理器或者 MCM 的通话建立和挂断服务,使用面向连接的 NDIS 小端口驱动程序的发送和接收服务。一个面向连接的客户可以向网络协议栈的高层提供自己协议的服务,或者它可在无连接协议和面向连接的介质之间充当仿真层。LAN 仿真(LANE)就是一个例子,它向上层协议隐藏了 ATM 面向连接的特征,表现为一种无连接的媒介。

8.4.2 Windows 2000 的层次化网络服务

下面讨论 Windows 2000 中建立在 API 及组件之上的网络服务,简略介绍远程访问、活动目录、网络负载平衡、文件复制服务(FRS)以及分布式文件系统(DFS)。另外,Windows 2000 支持几种基于 TCP/IP 协议的扩展特性的服务,它们包括网络地址翻译(NAT)、网际协议安全性(1PSec)以及服务质量(QoS)。

1. 远程访问

Windows 2000 支持远程访问,它允许远程访问的客户连接远程访问服务器并访问网络资源,例如文件、打印机以及网络服务。这样,客户就好像与远程访问服务器的网络连在一起。Windows 2000 提供两种远程访问类型:

- 客户使用拨号远程访问经过电话线或其他电信设施接入远程访问服务器,电信介质用于建立一条客户与服务器之间的临时物理或虚拟连接。

- 虚拟专用网络(VPN)的远程访问使 VPN 客户与服务器建立一条点到点的虚拟连接,这种连接一般建立在 IP 网络上,如因特网。

远程访问与远程控制不同,因为,远程访问起到代理连接 Windows 2000 网络的作用,而远程控制软件必须在服务器上运行应用程序,用于给客户提供一个用户接口。

2. 活动目录

活动目录是轻量目录访问协议(LDAP)中目录服务的 Windows 2000 的实现方式。活动目录基于存储资源对象的数据库,这些对象由 Windows 2000 网络中的应用程序定义。例如,

Windows 2000 域的成员以及结构, 包括用户账号和密码信息都存储在活动目录中。

Windows 中称对象类及其属性为模式, 活动目录模式中的对象以层次化布局。就像注册表内的逻辑组织一样, 其中容器对象能够储存其他对象。活动目录支持大量的 API, 它们能让客户在活动目录的数据内访问对象:

- LDAPC API 是 C 语言版本的 API, 它使用 LDAP 网络协议。C 或 C++ 应用程序可以直接使用此 API, 其他语言编写的应用程序需要通过翻译层访问这些 API。
- 活动目录服务接口 (ADSI) 是活动目录的 COM 接口, 它屏蔽了 LDAP 的编程细节。ADSI 支持多种语言, 如 MicrosoftVisual Basic, C 以及 MicrosoftVisual C ++ 。 ADSI 也可以由 Windows 脚本主机 (WSH) 应用程序使用。
- 消息 API(MAPI)兼容 MicrosoftExchange 客户软件和 OutlookAddressBook 客户应用程序。
- 安全账号管理器 (SAM) API 建立在活动目录的最上层, 它提供登录验证分组的接口, 如 MSV1_0(\ Winnt \ System32 \ Msvl _ 0.dll, 它沿用以前 NT LAN 管理器的验证) 和 Kerberos (\ Winnt \ System32 \ Kdsc. d11)。
- WindowsNT4 网络 API(NetAPI)用于 NT4 的客户通过 SAM 验证获取访问活动目录的权利。

活动目录是作为一个数据库文件实现的, 文件名称是 \ Winnt\Ntds\ Ntds.dit, 此文件在域中的域控制器上复制。活动目录的目录服务是在局部安全授权子系统 (Lsass) 进程中执行的 Win32 服务。它管理数据库, 使用实现数据库的磁盘结构 (on-disk) 的 DLL, 除此之外还提供用于保护数据库完整性的、基于事务的更新。活动目录数据库的存储还基于可扩展的存储引擎 (ESE) 数据库, 这种技术用于 MicrosoftExchangeServer5.5 版本的客户/服务器消息传递与群件系统。图 8-20 描述了活动目录的体系结构。

3. 网络负载平衡

Windows 2000 AdvancedServer 中的网络负载平衡服务是基于 NDIS 中间驱动程序。网络负载平衡允许建立一个可以多达 32 台计算机的集群, 这些计算机在网络负载平衡服务中称为集群主机。集群服务器维护一个虚拟 IP 地址, 并公开给客户访问, 客户的请求能够送至集群中的所有计算机。然而, 只有一台集群主机能够响应请求。网络负载平衡的 NDIS 驱动程序能够在有效的集群主机中, 分布式地、高效地分配客户空间。这样, 每台主机能够处理一部分的客户请求, 每个客户请求总能被一台且只有一台主机所应答。集群主机一旦决定处理某个客户请求, 它就将此请求送给 TCP/IP 协议驱动程序, 最终到达服务器应用程序; 而其他集群主机什么也不用做。如果一台集群主机不再运行, 则集群中的其余主机得知那台主机不再参与处理请求后, 就会将那台主机所获得的请求再发送给其余的主机。另外, 任何一台集群主机可以添加到一个集群中去替换一台已有的主机, 并且它能够做到无缝地开始处理客户请求。

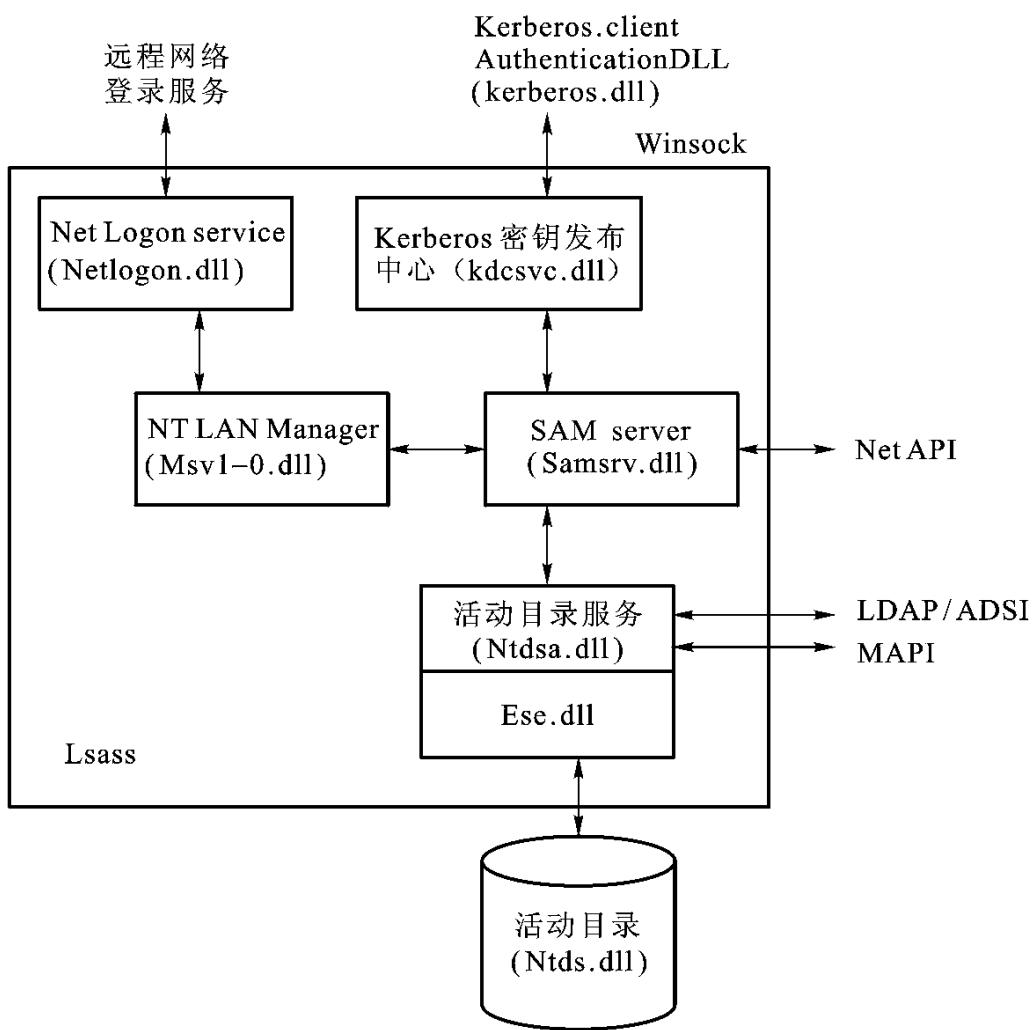


图 8-20 活动目录的体系结构

网络负载平衡并不是一种集群技术的通用解决方案,因为,与客户交互的服务器应用程序必须拥有某些特性:首先,它必须基于 TCP/IP;其次,它必须能够在网络负载平衡集群内的任何一台系统中处理客户请求。第二个要求的含义是一个应用程序,它必须能够得知共享状态,这样就能为客户请求提供服务,应用程序本身必须能够管理共享状态——网络负载平衡服务并不包括自动的在集群主机中发布共享状态的服务。网络负载平衡中理想化的应用程序应包含用于提供静态内容的网站服务器、Windows 媒体服务器以及终端服务。

4. 文件复制服务

Windows 2000 Server 包含文件复制服务(FRS)。复制域控制器 \ SYSVOL 目录的内容是它的主要功能, \ SYSVOL 目录是 Windows 2000 域控制器保存登录脚本和组策略的地方(组策略允许管理员为域中的计算机定义使用及安全策略)。另外, FRS 可以用于复制系统之间的分布式文件系统(DFS)的共享资源。FRS 还提供分布式多主控复制, 它能使任何一台服务器执行复制操作。复制的目录或文件一旦改变, 这些变化会广播到其他域控制器上。

FRS 中的基本概念是复制集, 它由两个或两个以上的系统组成, 这些系统能够根据管理制定的调度与拓扑, 相互复制目录树中的内容。只有 NTFS 卷上的目录可以进行复制, 因为 FRS 依赖于 NTFS 的更改历程, 它需要侦测目录中的文件与复制集中的变化。FRS 是基于多主控复制, 它能够理论上支持成百上千台系统, 并把它们作为复制集的一部分, 复制集中的计算机能够在任意的网络拓扑中连接(例如环状、星型或者网状)。计算机也可以是多个复制集中的成员。

FRS 是 Win32 的一个服务(\Winnt\System32\Ntffs.exe), 它使用经过验证的并附带通信加密的 RPC。另外, 活动目录包含它自身的复制能力, FRS 使用活动目录 API 检索来自活动目录的 FRS 配置信息。

5. 分布式文件系统

分布式文件系统(DFS)服务器位于工作站服务器的顶层, 它将文件共享共同连入一个单一的名字空间。文件能够在同一台或者多台不同的计算机上共享, 而且 DFS 可以为客户提供位置透明的资源访问。DFS 名字空间的基础是一个文件共享必须在 Windows 2000Server 中定义。

除了具有统一的网络资源名字空间的功能以外, DFS 还提供 DFS 复制集的特性, 它基于 FRS 复制集。管理员能够从两个或两个以上的共享中创建 DFS 复制集, 这样 FRS 就能够在复制集中的共享之间复制数据, 并保持其内容同步。通过随机选择复制集的成员来完成客户对复制集的数据请求, DFS 提供负载平衡的有限形式。另外, 当一个成员失效时, DFS 利用对工作成员或复制集中成员的路由请求获得高可靠性。

DFS 架构中的组件如图 8-21 所示。DFS 的服务器端实现由一个 Win32 服务(\Winnt\System32\Dfssvc.exe)和一个设备驱动程序(\Winnt\System32\Drivers\Ds.sys)组成。DFS 服务器在注册表(非活动目录系统)或者在活动目录中, 负责输出 DFS 拓扑管理接口以及维护 DFS 拓扑。当 DFS 收到客户请求时, 驱动程序进行拓扑查询, 然后将客户定向到请求的文件所在的系统。

在客户端, DFS 的实现依赖于 MUP 驱动程序和 NetWare 与 CIFS 重定向器的支持。当客户发出访问 DFS 名字空间中的文件 I/O 请求时, 通过使用合适的重定向器, 客户的 MUP 驱动程序与 DFS 服务器便可以进行通信。

6. TCP/IP 的一些扩展特性

另一些 Windows 2000 的网络服务依赖一些附加驱动程序, 扩展了 TCP/IP 协议驱动程序的基本网络特性, 这些利用专用接口的附加驱动程序与 TCP/IP 协议集成在一起。这些服务包括网络地址翻译(NAT)、网际协议安全性(1PSec)以及服务质量(QoS)。

(1) 网络地址翻译

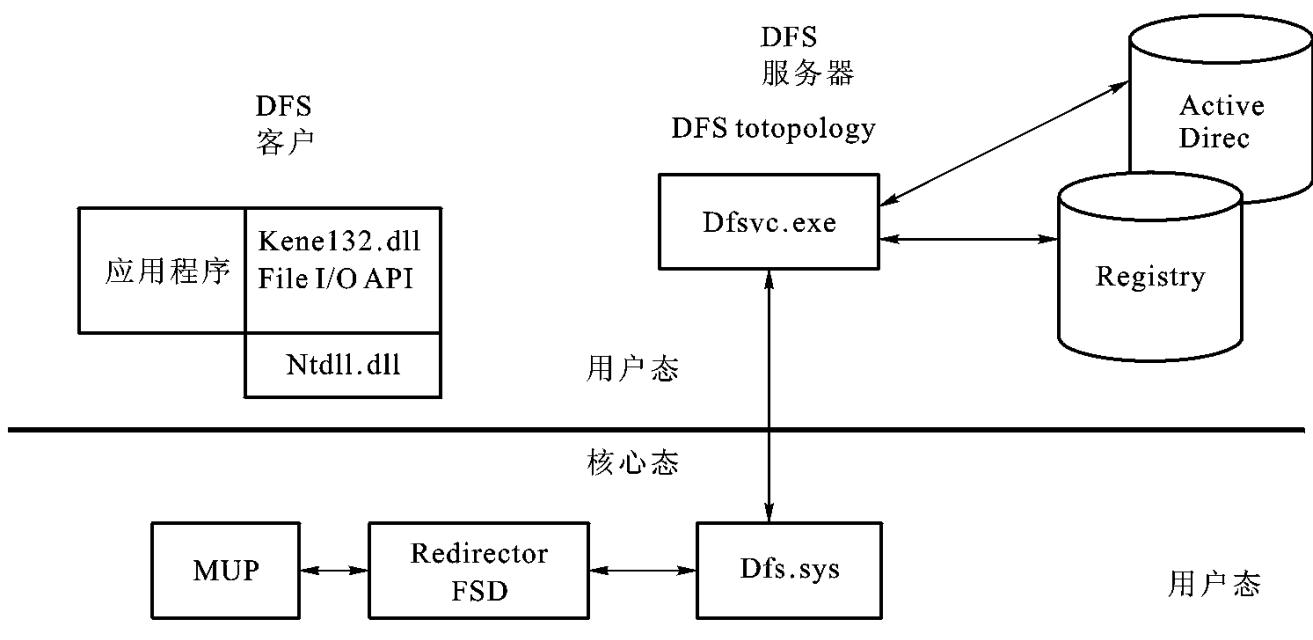


图 8-21 DFS 的组成

网络地址翻译(NAT)是一个路由服务,它允许多个本地IP地址映射成单个IP地址。如果没有NAT,局域网中的每台计算机必须分配一个公共的IP地址用于因特网通信。NAT使得局域网中的某一台计算机可以分配一个公共IP地址,而其他计算机通过那台计算机访问因特网。NAT完成局域网地址与公共IP地址之间的翻译,并且将分组从因特网路由至指定的局域网中的计算机。

Windows 2000 上的NAT组件由一个与TCP/IP栈相连接的NAT设备驱动程序和一个管理员用于定义地址翻译的编辑器组成。NAT能够作为一种协议被安装在带路由及远程访问的MMC插件上,或者是在使用网络及拨号连接工具来配置因特网连接共享时,将NAT的协议安装在Windows 2000上。

(2) 网际协议安全性

网际协议安全性(IPSec)与Windows 2000 TCP/IP栈集成在一起,它提供对IP数据的保护以防止窃听与伪造,并且防御基于IP的攻击。这两个目标通过基于密码技术的保护服务、安全性协议以及动态密钥管理服务得以实现。基于IPSec的通信包括如下一些特点:

- 验证服务确认IP消息的源及其一致性。
- 一致性保证了IP数据在传输过程中不会在无法得知的情况下被篡改。
- 使用加密技术保证了只有消息有效的接收者能够对消息的内容进行解密。
- 反再生性保证了每个分组都是惟一的,而且不可重用。这一特性防止窃听人员针对一个捕获的消息进行回复以达到建立一个会话或者得到未经授权的数据访问许可。

Windows 2000 中的IPSec依赖活动目录中配置的组策略,它使用活动目录 Kerberos 5 的验证方法来验证加入IPSec消息交换的计算机。IPSec 使用基于 Windows 2000 CryptoAPI,用

于加密、解密 IPSec 消息数据与密码的证书服务的私钥、公钥对。

IPSec 的实现由一个 IPSec 设备驱动程序 (\Winnt\System32\Drivers\klpsec.sys) 组成。此驱动程序与 TCP/IP 协议驱动程序集成在一起。在用户空间内,策略代理从活动目录中获得 IPSec 的配置信息,然后将 IPSec 过滤信息、(IPSec 通信应该使用的 IP 地址过滤器)传给 IPSec 驱动程序,并将安全性设置信息传给因特网密钥交换(IKE)模块。IKE 模块等待来自 IPSec 驱动程序安全性关联请求,并仲裁此请求,然后将结果返回给 IPSec 驱动程序使用。以上过程发生在验证与加密阶段。

3) 服务质量 (QoS)

如果没有采用任何特殊的方式,IP 通信网提供基于先进先服务策略。应用程序无法控制其消息的优先级,而且会经历突发的网络现象,这种现象指应用程序偶然地获得高吞吐率以及低延迟,然而,其他时候它们总是工作在糟糕的网络环境下。尽管这类服务在多数情况下是可以接受的,一个网络应用程序数量不断增加的环境需要更可靠的服务级别,或者说需要服务质量 (QoS) 的保证。视频会议、媒体流和企业资源计划 (ERP) 都是需要良好的网络性能的几个例子。QoS 允许应用程序指定最低带宽和最大延迟,这些特性只要求发送方与接收方的网络软件与硬件支持 QoS 标准即可,例如 IEEE 802.1p,它是定义 QoS 分组格式与 OSI 第二层设备(交换机和网络适配器)的响应机制的工业标准。

Windows 2000 的 QoS 是基于微软定义的一些 Winsock API,它们使应用程序能够请求在 Winsock 上通信的 QoS。例如,应用程序使用 WSCInstallQOSTemplate 用于安装 QoS 模板,它能够指定所需的带宽和延迟。第二个 API 是通信控制 (TC) API,它让管理程序更精确的控制此计算机所连的网络通信量。

Windows 2000 QoS 实现的核心是资源预定配置协议 (RSVP) Win32 服务 (\Winnt\System32\Rsvp.exe)。RSVPWinsock 服务提供者 (\Winnt\System32\RsvpMil) 通过 RPC 将应用程序请求传到 RSVP 服务。随后,RSVP 服务使用 TC API 控制通信量。TC API 在 \Winnt\System32\Traffic.dll 中实现,它将 I/O 控制命令发送给通用分组分类器 (GPC) 驱动程序 (\Winnt\System32\Drivers\Msgpc.sys)。GPC 驱动程序与 QoS 分组调度器 NDIS 直通驱动程序通信 (\Winnt\System32\Drivers\Psched.sys),它控制自计算机到网络的分组流,这样所允诺给指定应用程序的 QoS 级别就能够达到,同时也保证了合适的 QoS 头能够置于需要 QoS 的分组内。

8.5 实例研究:Linux 网络体系结构

Linux 的网络体系结构如图 8-22 所示,采用分成 4 层的层次结构实现 TCP/IP 协议簇。

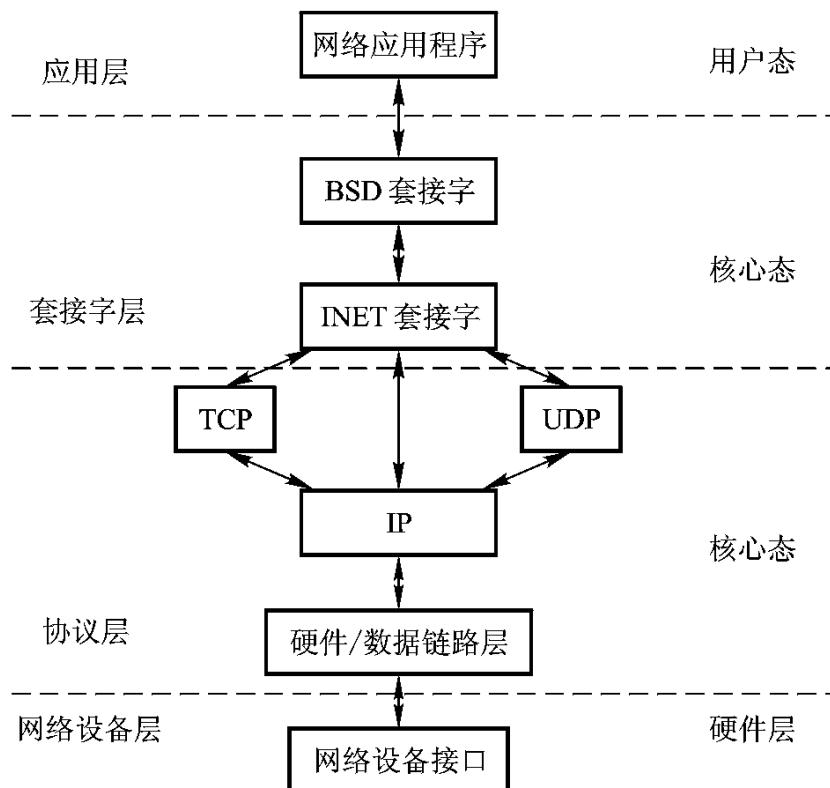


图 8-22 Linux 的网络体系结构

- **BSD 套接字** 包含了 BSD socket 编程接口的内容,有了这一层的支持,使用 UNIX 的 BSD socket 通用接口编写的程序都可以在 Linux 中运行。Linux 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 messages)、顺序数据包(sequenced packets)和数据包(packets)。
- **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。
- **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 TCP/IP 协议中网络互连实现的内容。
- **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。

8.6 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功

能的操作系统称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,在计算机网络上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,可以把它看作是网络用户和网络系统之间的接口。本章中先讨论了计算机网络的概念,包括:计算机网络的定义、组成、功能、分类、计算机网络系统、计算机网络的体系结构和通信协议,对于计算机网络赖以传送信息的数据通信系统也作了简单介绍。开放系统互连参考模型OSI/RM 是为异构型计算机协同工作而设计的标准化网络通信体系结构,具有七个功能层,并为每一层的标准开发出了实现标准,最广泛使用的网络通信体系结构是 TCP/IP。

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,它除了应该具备通常集中式操作系统所应具备的基本功能外,还应该增加联网功能,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。这里讨论了网络操作系统的特点、分类,以网络操作系统 Windows NT 为例,介绍了网络管理功能,主要有: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,本章中先介绍分布式操作系统的定义、特点和功能,然后,着重讨论了分布式操作系统主要功能的实现问题,包括: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的交互和合作,进程通信是分布式系统的关键机制。分布式进程通信可以分成三种: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发送消息和接收消息操作,但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被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的方式进行交互,但这个过程调用是用来访问远程计算机上提供的服务的,调用者和被调用者就好象运行在同一台机器上一样;套接字提供了进程通信的端点,它的实现原理与电话通信十分相似,利用 C/S 模式巧妙地解决了进程之间的通信问题。

分布式操作系统采用一类资源多个管理者的方式,可以分成两种:集中分布管理和完全分布管理。它们的主要区别在于:前者对所管资源拥有完全控制权,对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有部分控制权,不仅一类资源存在多个管理者,而且该类中每个资源都由多个管理者共同控制。集中分布管理介绍了两种资源搜索算法:投标算法和由近及远算法。完全分布管理较为复杂,由于在分布式系统中,各计算机相互分散,没有共享内存,因而,需要设计出适用的分布式同步算法。本章中介绍了

Lamport 提出的不使用物理时钟, 而对分布式系统中所发生的事件进行排序的方法, 定义了逻辑时钟, 然后, 讨论了 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。

分布式文件系统是分布式系统的重要组成部分, 它允许通过网络来互连的, 使不同机器上的用户共享文件的一种系统。本章以 Sun 公司的网络文件系统 NFS 为例, 介绍了分布式文件系统。

分布式操作系统支持分布式进程迁移, 这需要进程状态从一台机器转移到另一台机器上, 目的是使该进程能在目标机器上运行。进程迁移能用于均衡系统负载、降低通信开销、加快应用计算。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(1)由谁来发动进程迁移? (2)如何进行进程迁移? (3)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? 它由哪些组成部分?
2. 叙述计算机网络系统的主要组成。
3. 叙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, 叙述每代的主要标志。
5. 如何对计算机网络进行分类? 对每种类型作简单描述。
6. 什么是数据通信? 数据通信系统有哪些部分组成?
7. 解释数据通信中的基本概念: 信道容量、数据传输率和误码率。
8. 解释数据通信中的基本概念: 基带传输/频带传输、异步传输/同步传输、频分多路复用/时分多路复用。
9. 什么是通信协议? 说明它包含的三个要素。
10. 解释开放系统互连参考模型 OSI/RM。
11. 试说明层次式结构的网络中, 数据通信时信息的流动过程。
12. 解释 TCP/IP 网络体系结构。
13. 叙述网络操作系统的主要特征和分类。
14. 叙述网络操作系统与网络管理和控制有关的功能。
15. 叙述分布式计算机系统应满足的基本条件。
16. 分布式系统与网络系统的主要区别是什么?
17. 分布式操作系统应具有哪些基本功能?
18. 什么是分布式文件系统? 试述它的组成。

19. 叙述分布式操作系统的特性及优点。
20. 什么是 RPC? 说明其工作原理。
21. 叙述分布式系统中的集中分布资源管理方法。
22. 叙述分布式系统中的完全分布资源管理方法。
23. 叙述下列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? 说明它的用途。
26. 试描述 Lamport 分布式同步算法。
27. 试描述 Ricart 分布式同步算法。
28. 试描述令牌分布式同步算法。
29. 对令牌算法, 证明能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? 叙述防止和检测方法。
34. 比较同步通信和异步通信的优劣?
35. 什么是 C 线程包? 主要作用是什么?
36. 简述 Windows 2000 的网络构架和各类组件。
37. 讨论下列 Windows 2000 网络 API 的工作原理:
 - 命名管道和邮件槽
 - Windows 套接字 (Winsock)
 - 远程过程调用 (RPC)
 - 公共互连网络文件系统 (CIFS)
 - NetBIOS
38. 试述 Windows 2000 中下列网络服务的主要功能:
 - 远程访问
 - 活动目录、
 - 网络负载平衡、
 - 文件复制服务 (FRS)
 - 分布式文件系统 (DFS)

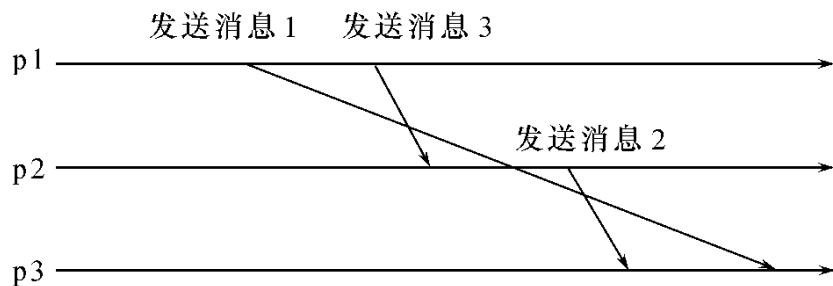
二、应用题

1. 在一个分布式系统中, 如果 P1 的逻辑时钟在它向 P2 发送消息时为 10, 进程 P2 收到来自 P1 的消息时逻辑时钟为 8, 下一个局部事件 P2 的逻辑时钟为多少? 如果接收时 P2 的逻辑时钟为 16, 下一个局部事件 P2 的逻辑时钟为多少?
2. 在图 8-8(b)中, 加入一条与消息 A 并发的新消息, 而且它不发生在 A 之前, 也不发生在 A 之后。
3. 由于在网络中可以通过不同路由发送消息, 因而, 可能出现后发送的消息先到达的乱序情况(如

图)。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来自不同进程的乱序消息(如图)进行排序。



参 考 文 献

1. William Stallings. Operating Systems Internals and Design Principles. Fourth edition. Prentice – Hall International Inc, 1998
2. Abraham Silberschatz (Bell Labs), Peter Baer Galvin (Cor. Tech. , Inc.). Operating System Concepts. Fifth edition. John Wiley & Sons Inc, 1997
3. Andrew S. Tanenbaum, Maarten van Steen. Distributed Systems Principles and Paradigms. Prentics Hall, 2002
4. Abraham Silberschatz, Peter Baer Galvin. Applied Operating System Concepts. John Wiley & Sons Inc, 1998
5. Andrew S. Tanenbaum. Computer Network. Third Edition. Prentics – Hall International Inc, 1996
6. Jim Hoskins. Exploring IBM Technology and Products. MAXIMUM PRESS, www. maxpree. com, 1998
7. Pfleeger. Security in computing, 1997
8. 尤晋元, 史美林等. Windows 操作系统原理. 北京:机械工业出版社 , 2001
9. 孟静. 操作系统教程. 高等教育出版社, 2001
10. 曾平, 李春葆. 操作系统 – 习题与解析. 清华大学出版社, 2001
11. Jim Mauro, 冯锐等. Solaris 内核结构. 机械工业出版社, 2001
12. 彭晓明、王强. Linux 核心源代码分析. 人民邮电出版社, 2000
13. 冯锐、邢飞等. Linux 内核源代码分析. 机械工业出版社, 2000
14. 雷澎等. Linux 的内核与编程. 机械工业出版社, 2000
15. 李善平等. Linux 操作系统及试验教程. 机械工业出版社, 1999
16. Andrew S. Tanenbaum, 陈向群等. 现代操作系统. Prentice Hall Inc, 1999
17. 屠祁, 屠立德. 操作系统基础. 清华大学出版社, 1999
18. William Stallings. 网络安全基础教程:应用与标准. 清华大学出版社影印, 1999
19. William Stallings. 密码学与网络安全:原理与实践. 清华大学出版社影印, 1999
20. 张效祥等. 计算机科学技术百科全书. 清华大学出版社, 1998
21. Tanenbaum. Andrew S, Woodhull. Albert. S, 操作系统:设计与实现. 第 2 版. 王鹏, 尤晋元等译校. Prentice Hall. Inc, 1997
22. 汤子瀛, 哲凤屏, 汤小丹. 计算机操作系统. 西安电子科技大学出版社, 1996
23. 孙钟秀. 分布式计算机系统. 国防工业出协社, 1987
24. 李永锡, 周兴社. 计算机操作系统原理. 西北工业大学出版社, 1987
25. 尤晋元. UNIX 操作系统教程. 西北电讯工程学院出版社, 1985