面向21世纪计算机专业本科系列教材

操作系统原理

何炎祥　熊前兴　编著

华中科技大学出版社
（华中理工大学出版社）

内 容 简 介

本书结合当今操作系统的设计并考虑操作系统的发展方向，全面介绍了现代操作系统的基本概念、设计原理以及在构造过程中可能面临的种种问题及其解决方法；介绍了操作系统设计的一些重要的新进展，如线程、实时系统、多处理器调度、进程迁移、分布计算模式、中间件、微核和安全性等。为便于理解，还选择了3个有代表性的操作系统：Windows NT，UNIX和MVS作为实例贯穿全书。

全书共分12章，分别是操作系统概述，进程描述与控制，并发控制 —— 同步与互斥，死锁处理，内存管理，处理机调度，I/O管理与磁盘调度，文件管理，分布计算，分布式进程管理，操作系统的安全性，排队分析。

本书概念清晰，内容丰富，取材新颖，强调理论与实践的结合，并配有《操作系统学习与解题指南》，以满足教学需要。它既可作为大专院校计算机及相关专业的教科书，又适合于计算机爱好者自学，还可作为有关工程技术人员的参考书。

面向21世纪计算机教材出版指导委员会

主任　陈火旺（中国科学院院士、国防科技大学教授、博士生导师）
沈绪榜（中国科学院院士、华中理工大学教授、博士生导师）
邹寿彬（华中理工大学副校长、教授、博士生导师）

委员　（以姓氏笔画为序）
王长胤（空军雷达学院教授、博士生导师）
韦　敏（华中理工大学出版社社长、副教授）
卢开澄（清华大学教授、博士生导师）
卢正鼎（华中理工大学教授、博士生导师）
张　峰（华中理工大学出版社总编辑、教授）
何炎祥（武汉大学教授、博士生导师）
苏锦祥（郑州大学教授、博士生导师）

秘书　沈旭日（华中理工大学出版社副编审）

面向21世纪计算机专业本科系列教材编委会

主任　何炎祥（武汉大学教授、博士生导师）
卢正鼎（华中理工大学教授、博士生导师）

委员　（以姓氏笔画为序）
卢炎生（华中理工大学教授、博士生导师）
肖德宝（华中师范大学教授、博士生导师）
陈　珉（武汉测绘科技大学教授）
张　峰（华中理工大学出版社总编辑、教授）
贺贵明（武汉水利电力大学教授、博士生导师）
姜新祺（华中理工大学出版社副总编辑、副编审）
熊前兴（武汉交通科技大学教授、博士生导师）

秘书　沈旭日（华中理工大学出版社副编审）

图书在版编目(CIP)数据

操作系统原理/何炎祥 编著
武汉：华中科技大学出版社, 2001年5月
ISBN 7-5609-2872-2 / TP316

Ⅰ. 操Λ
Ⅱ. 何Λ
Ⅲ. 通信理论-高等学校-教材
Ⅳ. TN

操作系统原理　　何炎祥　熊前兴编著

责任编辑：沈旭日　　封面设计：刘　卉
责任校对：章　红　　责任监印：张正林

出版发行：华中科技大学出版社
武昌喻家山 邮编：430074 电话：(027)87545012

经销：新华书店湖北发行所

录排：华中科技大学惠友科技文印中心
印刷：湖北省新华印刷厂

开本：787×960　1/16　　印张：　　字数：　000
版次：2001年10月第1版　　印次：2001年10月第1次印刷　　印数：1— 000
ISBN 7-5609-2872-2 / TP316　　定价：18.80 元

面向21世纪计算机教材出版指导委员会

主任　陈火旺（中国科学院院士，国防科技大学教授、博士生导师）
沈绪榜（中国科学院院士，华中理工大学教授、博士生导师）
邹寿彬（华中理工大学副校长、教授、博士生导师）

委员　（以姓氏笔画为序）
王长胤（空军雷达学院教授、博士生导师）
韦　敏（华中理工大学出版社社长、副教授）
卢开澄（清华大学教授、博士生导师）
卢正鼎（华中理工大学教授、博士生导师）
张　峰（华中理工大学出版社总编辑、教授）
何炎祥（武汉大学教授、博士生导师）
苏锦祥（郑州大学教授、博士生导师）

秘书　沈旭日（华中理工大学出版社副编审）

面向21世纪计算机专业本科系列教材编委会

主任　何炎祥（武汉大学教授、博士生导师）
卢正鼎（华中理工大学教授、博士生导师）

委员　（以姓氏笔画为序）
卢炎生（华中理工大学教授、博士生导师）
肖德宝（华中师范大学教授、博士生导师）
陈　珉（武汉测绘科技大学教授）
张　峰（华中理工大学出版社总编辑、教授）
贺贵明（武汉水利电力大学教授、博士生导师）
姜新祺（华中理工大学出版社副总编辑、副编审）
熊前兴（武汉交通科技大学教授、博士生导师）

秘书 沈旭日（华中理工大学出版社副编审）

华中科技大学出版社读者信息反馈卡

非常感谢您使用我社出版的教材。为了提高服务质量、出版水平，我们准备了此卡。卡中所提供的信息将作为我们改进工作的依据，所涉及的个人资料将予以保密。

☞ 个人资料

姓名：_____ 所在单位性质：_____

邮政编码：_____ 通讯地址：_____

1．学历：☐ 博士 ☐ 硕士 ☐ 本科 ☐ 高职 ☐ 专科 ☐ 其他

2．职业：☐ 学生 ☐ 教师 ☐ 技术主管 ☐ 干部 ☐ 其他

☞ 购书行为

1．购书用途：☐ 教材 ☐ 教学参考 ☐ 自学 ☐ 应考 ☐ 其他

2．购书原因：☐ 教师指定 ☐ 他人推荐 ☐ 宣传资料 ☐ 前言及内容简介
☐ 目录索引 ☐ 作者 ☐ 出版社 ☐ 封面与装帧
☐ 部分章节内容 ☐ 价格 ☐ 其他

3．学习时数（作教材、教参）：☐ 80 ☐ 72 ☐ 64 ☐ 54 ☐ 36 ☐ 其他

☞ 对所购书的评价

所购书名：_____

1．封面设计：☐ 满意 ☐ 基本满意 ☐ 不太满意 ☐ 不满意

2．印装质量：☐ 满意 ☐ 基本满意 ☐ 不太满意 ☐ 不满意

3．技术内容：☐ 满意 ☐ 基本满意 ☐ 不太满意 ☐ 不满意

4．文字质量：☐ 满意 ☐ 基本满意 ☐ 不太满意 ☐ 不满意

☞ 改进意见与建议

1．您认为本书有哪些地方需要改进?

a. 哪些章节需要精简? _____

b. 哪些内容需要更新? _____

c. 哪些内容需要增补? _____

2．希望与建议

a. 您希望获得什么样的服务?

b. 您现在最需要什么教材?

c. 您有什么好的教材需要出版?

✂

请将此卡寄至：

[4] [3] [0] [0] [7] [4]

湖北武汉市 华中科技大学出版社 计算机编辑室

沈 旭 日 （收）

E-mail: xuri@public.wh.hb.cn

总　序

自1946年世界上第一台电子数字计算机ENIAC诞生以来，计算机硬件系统经过了电子管、晶体管、小规模集成电路和大规模集成电路等几个阶段，正遵循着摩尔定律高速地发展：1998年，速度最快的个人PC微处理器是Intel 450MHz的Xeon, 1999年速度最快的已达800MHz；1997年2.1GB的磁盘容量已经很不错了，1999年则已突破10GB……软件方面，无论是操作系统、数据库系统，还是编程语言、应用软件，更是频繁地更新换代，令人眼花缭乱。

与此同时，作为计算机与通信技术结合的产物——计算机网络得到了迅速发展，特别是Internet技术的广泛应用，使得计算机网络的规模越来越大，网上主机数目一直保持每3年增长10倍的速率，Internet上的数据流量则保持着平均每半年就翻一番的增长速率，信息网络已交叉纵横整个世界，将偌大的世界连成了一个“地球村”。

计算机技术日新月异的进步，对现有的计算机专业的教学模式提出了挑战，同时也带来了前所未有的机遇。深化面向21世纪的教学改革，寻求一条行之有效的途径，培养跨世纪的高素质的科技人才，已是当务之急。如果说教学内容、课程体系的改革是教学改革的重点和难点，那么，教材建设则是其不可或缺的重要组成部分。华中理工大学出版社敏锐地抓住了这一点，在其倡议和组织下，我们经过研究、讨论和对教学经验进行总结，规划了这套“面向21世纪计算机教材”。为了满足各级各类学校人才培养的需要，这套教材计划包括计算机专业类教材和非计算机专业类教材，从

层次上则可划分为研究生层次、本科生层次、高职高专层次、中职中专层次、中小学层次等若干个子系列，将陆续分批出版。

当今世界，信息革命方兴未艾，知识经济已见端倪，教育观念正面临从注重以知识为主体向以能力为主体的转变。我们在对教材进行规划和评审时，尤其注重把提高学生素质、培养学生的应用能力和创新能力作为首要的评价标准，同时注意教材的特色和教学的实用性，反映最新的教学和科研成果，体现时代特征。

限于水平和经验，这批教材的编写、出版还存在不足，希望使用教材的学校、教师和学生以及其他读者积极提出批评意见，以便我们及时更新、修订，以满足读者要求。

面向21世纪计算机教材出版指导委员会主任

陈火旺（中国科学院院士，国防科技大学教授）

沈绪榜（中国科学院院士，华中理工大学教授）

邹寿彬（华中理工大学教授，副校长）

2000年2月10日

面 向 21 世 纪 计 算 机 专 业 本 科 系 列 教 材

序

人们已普遍认识到：21世纪是信息时代，以计算机为核心的信息技术是21世纪科技发展的大趋势。那么，作为计算机专业人才培养基地的大学计算机专业，如何适应这种发展，培养出符合时代要求、具有创新能力的人才呢？这是近年来计算机教育界讨论的热门话题，也是我们长期思考并努力探索的课题。

教材是人才培养的基础。在华中理工大学出版社的倡议和委托下，我们自1998年下半年起就开始讨论、筹划编写一套适应21世纪人才培养需要的计算机本科专业系列教材。在此基础上，我们组织了武汉大学、华中理工大学、华中师范大学、武汉测绘科技大学、武汉水利电力大学、武汉交通科技大学等院校的部分教师共同编写了这套“面向21世纪计算机系列教材”，以期总结我们在教学内容和课程体系改革方面的体会和做法，在适应21世纪的教材建设方面作出自己的努力。

值得欣慰的是，在教材的编写过程中，全国计算机专业教学指导委员会、中国计算机学会教育委员会联合推出了“计算机学科教学计划2000”（简称“2000教程”），这就更增强了我们编好这套教材的信心。在编写过程中，我们吸收了其中与我们内容相异的新内容。因此，也完全可以说，这套教材是与“2000教程”完全配套的教材。

我们这套系列教材的编写计划分为两个阶段：第一阶段，在2000年内出版“2000教程”中所涉及的所有专业课和部分专业基础课教材；第二阶段，在2000年以后出版与这套教材相配套的实践课和实验课教材，以及教学辅导书。

我们希冀这套教材具有以下特点：

1. 基础性和先进性相结合。与其他学科相比，计算机学科的一个显著特点就是知识内容更新更快，这对教学内容的选取、课程知识结构的构建提出了挑战。基于大学教育应努力实现知识、能力、素质三者辨证统一的目标，我们把编写的重点放在基础知识、基本技能和基本方法上，希望提高学生的理论素养和分析问题、解决问题的能力；与此同时，注重介绍最新的技术和方法，以拓展学生的知识面，激发他们学习的积极性和创新意识。

2. 理论性与应用性相结合。理论是规律的表现形式，良好的理论素养是应用的前提，而掌握理论的目的就是应用。在教材的编写过程中，我们注意了理论的系统性，在讲深讲透主要知识的基础上，各门课程知识点的选取做到尽量广一些；融理

论性和应用性于一体，在阐述理论的同时，尤其注意理论方法的讲授，以培养学生应用理论和技术的能力。此外，精心设计了比较多的习题，以加强应用能力和创造能力的培养。

3. 时代性和实用性相结合。力求精简旧的知识点，增加新的知识点，使整个知识建立在“高”、“新”平台上，体现教材的时代特征。但是，并不片面追求“高”、“新”，而是实事求是地充分考虑一般高校目前所拥有的教学设备、师资条件，注重教材的实用性。我们以为，教材建设不可能毕其功于一役，而必须根据学科的发展和客观环境以及条件的变化不断努力和改进。需要说明的是，与“2000教程”相比，我们根据人的认识规律和教学安排的需要，将有些课程进行了划分或合并，以便于教师根据需要灵活安排。

4. 科学性与通俗性相结合。概念原理、新技术的阐述力求准确、精练；写作风格上尽量通俗易懂、深入浅出、图文并茂，增加可读性，便于学生自学。

如果说科学技术快速发展是21世纪的一个重要特征的话，那么，教学改革将是21世纪教育工作永恒的主题，是需要不断探索的课题。我们要达到以上目标，还需要不断地努力实践和完善。欢迎使用这套教材的教师、学生和其他读者提出宝贵意见。

最后，衷心感谢参加这套教材编写的所有作者所贡献的成果和辛勤的汗水，对为这套教材的编写提供支持的有关学校、院系的领导和老师表示诚挚的谢忱！感谢华中理工大学出版社为本系列教材的出版所付出的艰辛和努力！

面向21世纪计算机专业本科系列教材编委会主任

何炎祥　（武汉大学教授）

卢正鼎　（华中理工大学教授）

1999年11月20日

前　言

计算机系统存在着极大的多样性，不仅机型、容量和速度存在差异，而且应用程序和系统支持环境也不尽相同；此外，整个计算机系统正以极快的速度发展着，作为计算机系统中的十分关键的系统软件——操作系统，也不断有新的关键技术诞生。因此，本书将结合当今操作系统的设计并考虑操作系统的发展方向，着重讨论操作系统设计的基本概念、基本原理和典型技术，同时，介绍构造操作系统过程中可能面临的种种问题及其解决办法；介绍操作系统设计中的一些非常重要的新进展，包括线程、实时系统、多处理器调度、分布式系统、中间件、进程迁移和安全性等。其目的是尽可能清晰、全面地向读者展现当代操作系统的设计原理与基本实现技术，以便读者深入了解现代操作系统,为今后进行较深层次的软件研制与系统开发打下坚实的基础。

为了帮助读者更好地理解操作系统的概念、原理和方法，更好地将理论与实际设计相结合，我们选择了以下3个具有相关性、代表性和典型性的操作系统作为例子。

• Windows NT：一个单用户、多任务操作系统，第一个采用面向对象设计原理设计的重要操作系统。它运行在个人计算机、工作站和服务器上。

• UNIX：一个多用户、多任务、分时交互式操作系统。它开始是为微型计算机设计的，现在已广泛应用于其他机型。

• MVS：IBM大型计算机的顶级操作系统，是到目前为止最为复杂的操作系统。它提供了批处理和分时处理的功能。

本书由以下12章组成。

第1章操作系统概述：对本书的内容进行了概述。

第2章进程的描述和控制：讨论了进程的概念以及操作系统对进程进行控制的数据结构，还讨论了与进程相关的线程等内容。

第3章并行控制 —— 互斥和同步：着重讨论在单一系统中并行处理的关键技术——互斥和同步机制。

第4章死锁处理：描述了死锁和饥饿的性质并讨论了解决它们的方法。

第5章内存管理：提出了多种内存管理方法，并讨论了支撑虚拟内存所需的硬件结构和操作系统用来管理虚拟内存的软件方法。

第6章处理机调度：讨论了各种不同的进程调度方法，包括实时调度策略。

第7章I/O管理和磁盘调度：论述了操作系统对输入/输出的控制，尤其是对系统

性能影响较大的磁盘I/O的调度和控制。

第8章文件管理：对文件的组织、存储、使用和保护等方面的内容进行了综述。

第9、10章分布计算和分布式进程管理：描述了分布式操作系统的一些关键设计领域，包括Client/Server结构，用于消息传递和远程过程调用的分布式通信机制、分布式进程迁移、中间件以及解决分布式互斥和死锁问题的原理与技术。

第11章操作系统的安全性：简要讨论了保证计算机系统和网络安全性的相关理论和方法。

第12章排队分析理论：扼要介绍了排队分析的基本原理，及其在操作系统设计中的应用。

本书是参照全国高校计算机专业教学指导委员会、中国计算机学会教育委员会和全国高等学校计算机教育研究会联合推出的《计算机学科教学计划2000》，从传授知识和培养能力的目标出发，结合作者多年从事教学与科研的实践，根据本课程教学的特点、重点和难点编写的。本课程的先导课程是“高级语言程序设计”，“数据结构”，“计算机组成原理”等。本课程的教学参考学时为72学时，也可根据需要对其内容进行取舍，将学时压缩到54学时。

为了满足教学和自学的需要，我们还编写了配套教材《操作系统原理学习与解题指南》，供读者选用。

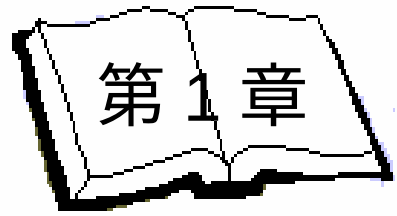
本书的第1~4，6，10，12章由何炎祥编写，第5，7~9，11章由熊前兴编写，何炎祥统编了全书。在本书的编写过程中，得到了武汉大学计算机学院和武汉理工大学计算机学院的领导和同事们的关心，华中科技大学出版社为本书的出版给予了大力支持，文中还引用了一些专家学者的研究成果和公司的产品介绍，在此一并表示感谢。

限于水平，书中错误难免，敬请读者赐教。

编著者

2001年7月于武昌珞珈山

目　录



操作系统概述

操作系统(Operating Systems，简称 OS)是控制应用程序执行和充当人机交互界面的软件。OS 常可以实现 3 个目标和执行 3 种功能：

① 方便。OS 使计算机应用起来更方便。

② 高效。OS 使计算机系统的资源以一种高效的形式得以应用。

③ 可扩展。建造 OS 时，应使它具有进行扩充和发展，并且在引进新系统组件的同时，又不会干扰现有服务的能力。

1.1 操作系统的作用

1.1.1 作为人机交互界面

如图 1.1 描述的一样，计算机用以给用户提供应用的软、硬件可看做是层次化、分级的形式。终端用户通常不必考虑计算机本身的设计，只是借助应用程序与计算机系统交互。这些应用程序可通过程序设计语言来表达，并且可被应用程序员升级。如果一个人想通过一系列机器指令来开发应用程序，而这些应用程序又全权控制计算机硬件，那么，等待他的将是一个十分庞杂的工作。为简化这项工作，一系列系统程序则应运而生。在这些系统程序中，有些被称为实用程序，它们频繁地执行程序创建、文件管理、输入/输出(I/O)设备控制等功能。程序员可以利用它们来开发应用程序，而应用程序在运行时又调用实用程序执行某项

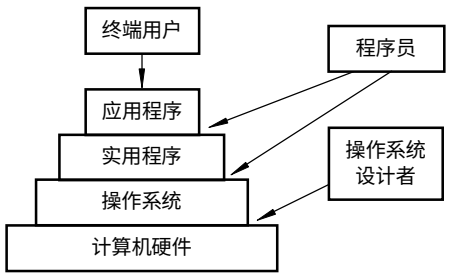


图 1.1 计算机系统的层次视图

功能。最重要的系统程序是 OS。由 OS 定义的软、硬件和数据，给程序员提供了方便的界面，使程序员和应用程序更容易获取和使用计算机系统中的资源、工具和服务。

简言之，OS 为如下领域提供服务：

① 程序创建。OS 提供多种工具（如编辑器、调试器）和服务来帮助程序员编程。特别要说明的是，这些服务以实用程序的形式存在，它们并不是 OS 的一部分，而只是通过 OS 获得。

② 程序执行。将指令和数据装入主存、I/O 设备和文件初始化，以及其他资源的准备等，这些工作是由 OS 完成的。

③ I/O 设备的访问。每个 I/O 设备都有用来操作自身的指令系列和控制信号，OS 则可使程序员用简单的读/写操作来使用和控制 I/O。

④ 控制对文件的访问。就文件而言，控制不仅能识别 I/O 设备属性(磁盘驱动器，打印驱动器)，而且能识别存储介质上的文件结构。当涉及到多用户系统时，OS 能提供控制访问文件的保护机制。

⑤ 系统访问。当涉及一个共享或公用的系统时，OS 承担整体控制系统的访问和具体系统资源的访问。它必须使资源和数据不受无权用户的干扰，而且必须解决资源使用中的冲突问题。

⑥ 查错和纠错。大量的错误不能在系统运行时出现，包括内部和外部硬件错误，例如存储器错误，或设备失误、无效；各种软件错误，例如，算术溢出、试图使用禁用存储器，OS 不能满足应用程序要求等。每种情况下，OS 都必须作出响应，并在最小限度影响执行程序的条件下清除错误，或向应用程序报错，或重试一次，或终止出错程序。

⑦ 簿记。好的 OS 应能搜集各种资源的使用统计数据和监听执行情况(如响应时间)等。这些信息对将来完善计算机所需的条件，以及提高系统执行性能是十分有用的。

1.1.2　作为资源管理者

是否能说 OS 控制运行、存储和处理数据呢？从某种观点来看是对的：OS 通过管理计算机资源来控制计算机基本功能的实现。但这个控制是以不寻常的方式运作的。通常认为一个控制部件应位于其被控物的外部，或至少是不同于且分离于被控物的，OS 就不是这样，作为一个控制部件它至少有两个方面不寻常：

- 以通常的计算机软件的方式起作用，即它是一些程序的集合。
- 频繁放弃控制，且必须依靠处理器来重新获取控制。

OS 实际上只不过是一个计算机程序。它与一般程序的关键区别是程序的内容。OS 可指示处理器使用其他系统资源，加速其他程序的执行。但这时，处理器必须停

止执行 OS 程序。这样，OS 就会为处理器而放弃控制，去做另一些“有用”工作，待到重新掌握控制权后再为处理器的下一项工作做准备。

图 1.2 显示出一些受 OS 控制的主要资源。OS 的一部分在内存中。包括内核，它包含了 OS 中使用最频繁的和在给定时间内正被使用的 OS 的一部分。内存余下的部分存储了其他用户程序和数据。主存的分配是由 OS 和处理器中的存储管理硬件决定的，如图 1.2 所示，OS 决定什么时间 I/O 设备能够被处于执行中的程序段用到，还控制和管理对文件的使用。处理器本身是资源，OS 决定处理器有多少时间花在执行特定用户程序上。

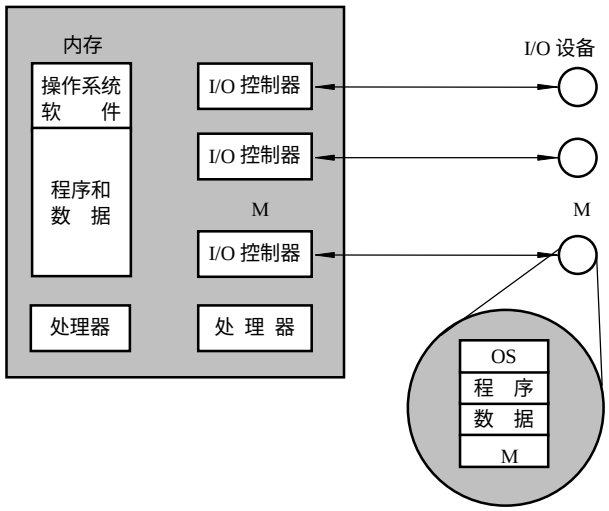


图 1.2　作为资源管理器的操作系统

1.1.3　推动操作系统发展的因素

操作系统随时间而演变主要基于以下因素。

1. 硬件升级以及新的硬件类型

例如，早期的 UNIX 和 OS/2 不含分页部件，因它们的运行不需分页硬件，最近的版本已增加了分页能力；同样，图形终端和页式终端代替逐行显示的终端会影响 OS 的设计，这样的终端允许用户通过屏幕上的窗口，在同一时间内提出多个申请，这就需要 OS 有更多更复杂的支持。

2. 新服务

为响应用户要求和系统管理者的需要，OS 要扩大服务范围。当发现现有的工具很难为用户保持良好的服务性能时，将新的测量和管理工具添加到 OS 中；在出现一个新请求，如要求在显示器上应用 Windows 操作系统时，就需要有相应的支撑机

制。

3. 修补

OS 中的错误会不断被发现，因此就必须进行修补。当然，在修补的同时也有可能引入新的错误。

显而易见，OS 在创建中应以组件为基础，应具有清晰定义的各组件之间的接口，应有很好的文件支持。

1.2　操作系统的演变

为了理解 OS 的关键需求和 OS 主要特征的重要性，回顾一下 OS 的发展是十分有益的。

1.2.1　串行处理系统

最早的计算机，从 20 世纪 40 年代末到 50 年代中期，程序员直接与硬件接触，根本没有 OS。计算机运行在一个集成了指示器、各种开关、一些输入设备以及一个打印机的控制台之上。用机器代码编写的程序由输入设备、读卡机载入，在因错误而导致程序被挂起时，出错位置由指示灯显示。程序员可以通过检测寄存器和主存来寻找出错原因。如果程序正常执行完毕，则结果会输出到打印机上。

早期的这种系统存在两个问题：

(1) 上机安排

上机请求要用一个订单来预约机器时间。典型问题是，如果一个用户事先约定了一个小时，而最终执行程序时只用了 45min，就浪费了 15min 的时间；如果在约定的一个小时内完成不了，则程序就会在执行一小时后被迫停止。

(2) 启动时间

启动时间包括向存储器中装载编辑器和高级语言源程序、存储编译过的程序(目标程序)、然后连接装配目标程序与公共函数等所花的时间。其中的每一步都有可能引起纸带机或卡片机的安装和拆卸。如果这一过程出现错误，则用户又不得不回到准备程序的开始。这样，就有相当的时间被浪费在程序启动上。

OS 的这种工作模式被定义为串行处理模式，它表明了用户串行获取计算机的事实。随着时间的推移，各种系统软件工具应运而生，包括标准函数库、链接器、装配器、反编辑器、I/O 驱动程序等，它们对所有用户都是开放的。显然，这些软件工具可以提高串行处理的效率。

1.2.2　简单批处理系统

早期计算机十分昂贵，因而最大限度地利用它就显得很重要。早期计算机的上机安排和启动时间所造成的时间花费是不可接受的。

为了改善上述情况，产生了批处理系统的概念。第一个批处理系统产生于 20 世纪 50 年代中期，是由 General Motors 开发，用于 IBM 701 计算机上。这个概念后来被 IBM 的顾客改进并应用在 IBM 704 中。到 60 年代初期，一些业主为他们的计算机系统开发了批处理系统。IBSYS 即 IBM 公司为 7090/7094 计算机配置的操作系统，由于它对其他系统的广泛影响而尤为著名。

简单批处理系统的中心思想是，通过应用一种被称为监视器的软件，使用户不必再直接接触机器，而是先通过卡片机和纸带机向计算机控制器提交作业，由监视器将作业组织在一起构成一批作业，然后将整批作业放入由监视器管理的输入设备上，每当一个程序执行完毕返回监视器时，监视器已自动装入下一个程序。

下面从监视器和处理器这两个方面入手来考虑这些设计是如何实现的。从监视器角度来看，是它决定事件的顺序。因此，大部分监视器必须总是处于主存中，这部分称为驻留监视程序，其内存结构如图 1.3 所示。剩下的监视程序可作为用户程序的一部分，在各种作业开始时根据需要而装入。监视器一次从输入设备读取一个作业(典型的是从读卡机和磁带驱动器上读入)。当前作业被放置在用户程序区，这时，此作业就开始拥有控制权。在作业完成后，控制权又回到监视器，监视器马上读入下一作业。每项作业的结果都被打印出来传给用户。

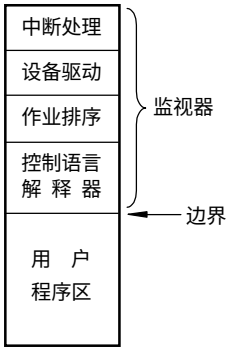


图 1.3　驻留监视器的内存结构

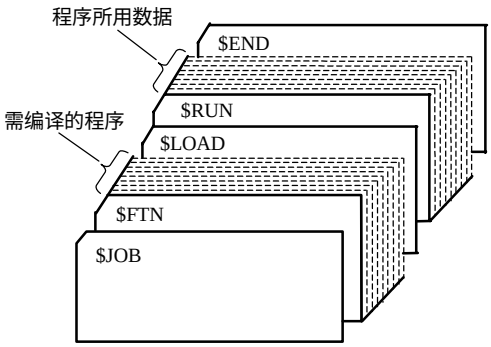


图 1.4　一个简单批处理系统的卡片叠

从处理器的角度考虑：在某段时间内，处理器从包含监视器的主存(或内存)中执行指令，这些指令使得下一个作业被读入主存的其他部分；一旦一个作业读入主存，处理器就会收到监测器的一条分支指令，指示处理器继续在用户程序的开始处执行；接着，处理器执行用户程序直至程序结束或遇到错误。这两种情况的任何一种都会促使处理器从监视器程序中取下一条指令。因此，“控制权被传递给作业”表明处理器正在用户程

序中存取和执行指令；“控制权返回控制器”表明处理器正在存取和执行监视器的指令。显然，监视器掌握调度权。一批作业排成队列，被尽可能快地执行，就可以使处理器没有闲置时间。

那么，作业是如何启动的呢？实际上，也是在监视器的控制之下进行的。对于每一作业，指令被包含在作业控制语言(JCL)的原语形式中，这是一种特殊的程序语言，用来给监视器提供指令。图 1.4 展示了一个由卡片输入作业的简单示例。在此例中，用户将用 FORTRAN 语言书写的程序和一些数据，以及作业控制指令(它们用“$”符号打头来标明)提交给系统。为执行这个作业，监视器读入$FTN 卡片，从它的存储介质(常为纸带)上装入适当的编译器。编译器将用户程序翻译成目标代码，存入存储器或介质存储器中。如果用户程序被存入存储器，则操作过程即为“编译→装入→运行”。如果要存入磁带，就需要$LOAD 卡，由监视器读此卡，监视器在编译操作完成之后重新获得控制权。监视器唤醒加载器代替编译器，将目标程序装入内存，并将控制权传递给程序。从这个意义上讲，一大片主存可由不同的子系统共享，尽管在同一时间只有一个系统是常驻内存且正在被执行的。

在用户程序执行期间，一条输入指令将读取一个数据卡片。用户程序的输入指令引发 OS 的一个子程序，该子程序检查并判定用户程序中是否出现读取一个 JCL 卡的指令。如果是的，则将控制权返还监视器。在完成一个用户作业的执行之后，监视器就继续扫描输入卡片，直至碰到下一个 JCL 卡。这样，系统就不会受具有过多或过少数据卡程序的干扰。

可以看出监视器或批处理系统是一个简单的计算机程序。它依靠处理器从主存的不同区域取指令来交替获得和放弃控制权。为此，其他硬件支持也是需要的，例如：

① 存储器保护。用户程序在执行时，不能修改包含监视器的主存。如果有这样的企图，则处理器硬件应能发现这一错误并及时将控制权转交监视器。监视器将终止该作业，打印出错信息，装入下一个作业。

② 计时器。计时器用来防止一个作业长期占有整个系统。计时器设置在每项作业的开始。如计时器超时，就发生中断，并将控制权返还监视器。

③ 特权指令。有些指令被指定为特权指令，只能被监视器执行。如果处理器在执行用户程序时遇到这样一个指令，就发出一个出错中断。在特权指令中有 I/O 指令，所以监视器获得了所有 I/O 设备的控制权。这就可以防止用户程序意外地从下一个作业中读取作业控制指令。如果一个用户程序想执行 I/O 操作，则必须请求监视器为它执行。如果处理器在执行用户程序时遇到特权指令，控制器硬件就会认为是出错并将控制权交给监视器。

④ 中断。早期的计算机没有这项功能。这项功能使 OS 在放弃和取得控制权时变得更加复杂。

当然，没有这些支持也可创建操作系统，但可能会使问题变得一团糟，所以即

使是最原始的批处理操作系统，也有这些硬件支持。作为一个例外，世界上应用得最广泛的操作系统，如 PC-DOS/MS-DOS，就是既没有主存保护又没有特权 I/O 指令的操作系统。由于这个系统只用于单用户个人电脑，所以问题并不严重。

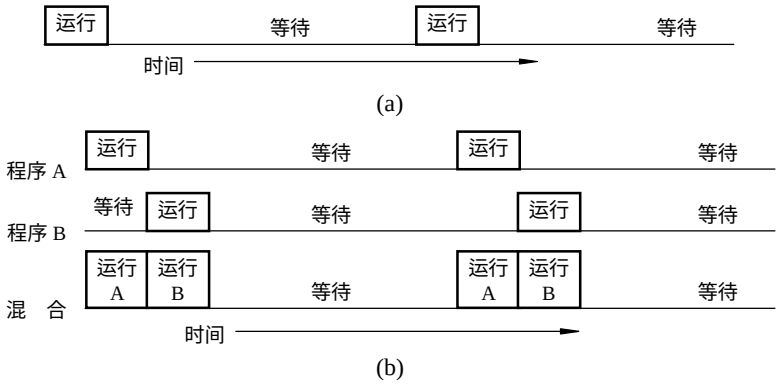
对于批处理系统，计算机的时间随用户程序的执行和监视器的执行而变化。这有两个代价：某些主存给了监视器，一些机器时间被监视器占用。这些都是额外的代价，即使有这种额外代价，简单批系统仍提高了计算机的使用效率和应用水平。

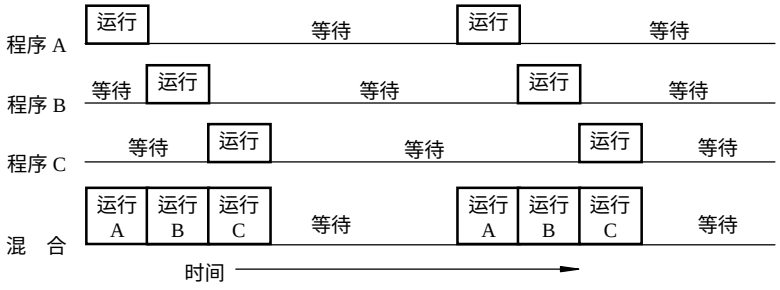
1.2.3　多道程序批处理系统

即使有简单批处理系统提供的自动作业队列，处理器也常常闲置。问题在于 I/O 设备的执行速度比处理器慢。图 1.5 列举了一个有代表性的例子。该例子中的程序是处理一个文件的记录和执行，平均每秒钟执行 100 条机器指令。在这个例子中，计算机要花去 96%的时间等待 I/O 设备来完成数据传递。图 1.6(a)说明了这种情形。处理器花一定时间执行程序直至遇到一个 I/O 指令，这时，它必须等待直至 I/O 指令结束。

读一个记录	0.0015s
执行 100 条指令	0.0001s
写一个记录	0.0015s
总　共	0.0031s
CPU 利用百分比=0.0001/0.0031=0.032=3.2%	

图 1.5　系统利用率例子





(c)

图 1.6　多道程序设计

(a) 单道程序设计　(b) 两个程序的多道程序设计　(c) 3 个程序的多道程序设计

这种低效并不是不可避免的，只是必须要有足够的存储空间来存储操作系统和用户程序。设想一个空间中有一个操作系统和两个用户程序，当一个作业等待 I/O 时，处理器可转向另一个作业，这样就不必等待 I/O，如图 1.6(b)所示。而且，可通过扩展内存空间来装下 3 个、4 个或更多的程序，并在它们之中进行切换，如图 1.6(c)所示。这个设想就是多道程序设计基本思想，它是现代操作系统的主旋律。

为说明多道程序设计的好处，可观察基于 Turner 上的一个例子。设想一台电脑配备 256KB 的可用内存空间(未被 OS 占用的)，一个磁盘，一个终端和一台打印机。3 个程序 JOB_1、JOB_2 和 JOB_3，同时被提交执行，如表 1.1 所列。假设 JOB_2 和 JOB_3 请求占用处理器，JOB_3 可持续占用键盘和打印机。在一个简单的批处理环境中，这些作业将排成队列执行。如果 JOB_1 在 5min 内执行完，则 JOB_2 必须等待 5min 后再执行 15min，JOB_3 就必须在 20min 后才能开始执行，在它提交申请后的 30min 执行完毕。平均资源使用、响应时间已经在表 1.2 的单道程序栏中表示出来，其作业执行情况如图 1.7 所示。

表 1.1　某些程序执行属性

	JOB_1	JOB_2	JOB_3
作业类型	偏重计算	偏重 I/O	偏重 I/O
执行时间	5 min	15 min	10 min
所需内存	50KB	100KB	80KB
是否需要磁盘	No	No	Yes
是否需要终端	No	Yes	No
是否需要打印机	No	No	Yes

表 1.2　多道程序设计在提高资源利用率方面产生的效果

	单道程序设计	多道程序设计
处理机使用	17%	33%
内存使用	30%	67%

磁盘使用	33%	67%
打印机使用	33%	67%
经过时间	30 min	15 min
吞吐率	6 jobs/h	12 jobs/h
平均响应时间	18 min	10 min

现在设想这 3 个作业在一个多道程序 OS 下同时运行。因资源之间几乎没有竞争，所有作业都共存于计算机中，能够以最少的时间运行完毕(假设 JOB_2 和 JOB_3 有足够的处理器时间来控制输入/输出操作)。JOB_1 仍然需要 5min，但在 JOB_1 完成时，JOB_2 已完成 1/3，而 JOB_3 已完成 1/2。3 个作业将在 15min 之内全部完成，资源利用率的提高是明显的，如表 1.2 所示，其作业执行情况如图 1.8 所示。

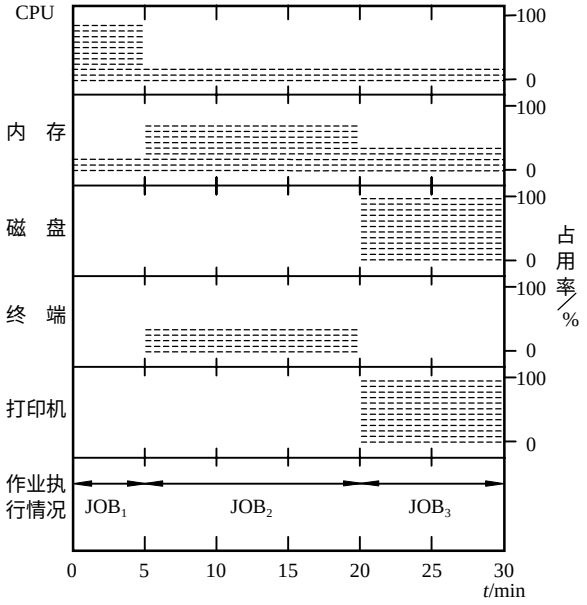


图 1.7　单道程序设计的作业执行情况

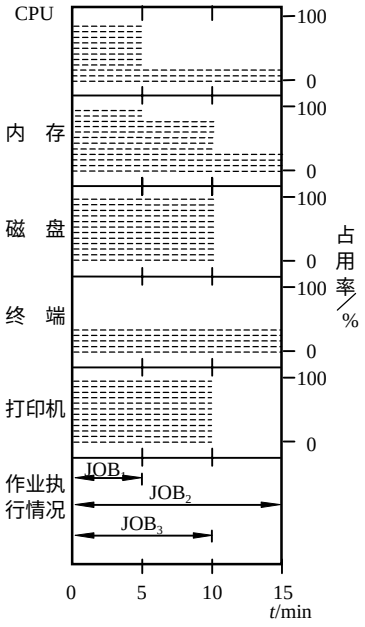


图 1.8　多道程序设计的作业执行情况

就像在简单批处理系统中一样，一个多道程序批处理系统是一种必须依靠某种计算机硬件的程序。对多道程序最显著的、有用的附加特征是支持 I/O 中断和 DMA 的硬件。有了 I/O 中断驱动和 DMA，处理器可以为一个作业发出 I/O 请求，而且在设备控制器执行这一 I/O 请求的同时，处理器又可继续执行另一个作业。当 I/O 操作完成后，处理器被中断，控制权传给操作系统中的中断控制程序，接下来 OS 将控制权传给下一个作业。

多道程序操作系统比单道程序操作系统要完善得多。为了使多个程序处于就绪

状态，必须将它们放入主存中，需要一些存储器管理表列。另外，如果有多道作业准备运行，则处理器必须决定运行哪一个，这就要有调度算法，这些概念将在后面讨论。

1.2.4　分时系统

利用多道程序设计技术，批处理将十分高效。然而，对许多作业，例如，事务处理，提供一个用户与计算机直接作用的交互作用模式是必要的。

今天，交互计算可以由一个微型计算机来实现。在计算机大而昂贵的 20 世纪 60 年代，这种选择是不现实的，这是分时系统得以发展的动因。多道程序能够用来控制多道相互作用的作业，这项技术被称为分时技术(Time Sharing)，它反映了处理器的时间是由多个用户共享的事实。分时系统的基本思想是让多个用户同时通过终端使用系统，而操作系统则在系统内部处理用户程序。这样，如果有 n 个用户在同一时间请求设备，则为单个用户服务的速度只是高效速度的 1/n，而且还不算 OS 的时间。然而，考虑到人的反应相对慢一些，在一个精心设计的系统中，还是可以使计算机上的多个用户都觉得这台计算机是专门为自己服务的。

批处理多道程序和分时系统都使用了多道程序设计技术。其关键的不同之处列在表 1.3 中。

表 1.3　批处理多道程序设计与分时系统

	批处理多道程序设计	分时系统
策略目标	使处理机使用率最高	使响应时间最小
指令源	作业控制语言提供	在终端上键入命令

最早的分时系统之一是便携式分时系统 CTSS，它是由一个称为 MAC 的项目组在 MIT 开发的。此系统最早是在 1961 年为 IBM 709 而开发的，后来被移植到 IBM 7094 上。

与后来的系统相比，CTSS 十分初级，它的基本操作很容易解释。该系统要在 36 位的 32KB 主存上运行，其中包含 5KB 的驻留监控程序。当控制权被指定给交互用户时，用户的程序和数据装入剩余的 27KB 主存中。系统时钟大约每 0.2s 产生一次中断。在每个时钟中断里，OS 获得控制权，而后将处理器指定给另一个用户。这样，在有规则的间隙中，当前用户被另一个用户所取代。为保证老用户以后有再占有的权利，老用户程序和数据在新用户程序和数据读入之前被写到磁盘上。此后，老用户的主存空间在下一次运行时将再恢复。

为减少磁盘事故，用户存储器只有当写入程序要覆盖它时才写出。有关规则如图 1.9 所示。假设有如下 4 个请求存储器的用户：JOB_1(15KB)、JOB_2(20KB)、JOB_3(5KB)、JOB_4(10KB)。最初，监视器装入 JOB_1，给它控制权，如图 1.9(a)所示；

其次，监视器决定把控制权给 JOB_2，因 JOB_2 比 JOB_1 需要更多的主存空间，故 JOB_1 必须先写出，然后装入 JOB_2 如图 1.9(b)所示；其三，JOB_3 被装入运行，然而，因为 JOB_3 比 JOB_2 小，JOB_2 的一部分可保留在内存中，减少了磁盘的写时间如图 1.9(c)所示；其四，监测器把控制权再传给 JOB_1，当 JOB_1 重新被装入内存时如图 1.9(d)所示；JOB_2 多出的部分必须被写出，当 JOB_4 装入时，存储器中的 JOB_1 和 JOB_2 的一部分仍被保留，这时，JOB_1 或 JOB_2 有一个被激活，需要时再装入一部分。此例中，JOB_2 接着运行。这就需要将 JOB_4 和 JOB_1 常驻的部分写出，并读入 JOB_2 的相关部分。

CTSS 与今天的分时系统相比很初级，但它很起作用。它十分简单，这就缩小了监视器的规模；因为作业被装入时总是在相同的位置，因而无需特别的加载技术。在 7094 上运行时，CTSS 最多能支持 32 个用户。

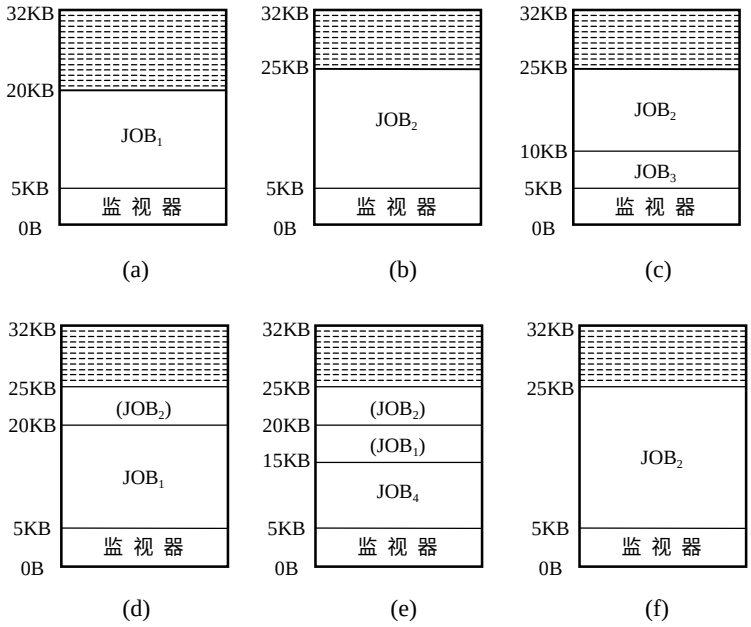


图 1.9　CTSS 操作示意图

分时和多道程序给 OS 带来了许多新问题。如果存储器中有多个作业，则这些作业必须受到保护以免相互干扰(例如，更改彼此的数据)。若有多个相互作用的用户，则文件系统也必须被保护起来，只允许有特许权的用户访问特定的文件。有些资源(如打印机和物理存储设备)，必须得到有效的控制和管理，这些问题都是需要研究的。

1.3　操作系统的主要成就

操作系统是现有软件系统中最复杂的软件之一。到目前为止，操作系统已取得了 5 项主要成就：进程、内存管理、信息的保护与安全性、调度与资源管理、系统结构。它们涉及到现代操作系统在设计和实现方面的基本问题。

1.3.1　进程

进程是操作系统结构的基础。进程这个术语最早是在 20 世纪 60 年代由 Multics 的程序设计员使用的，它在某种程度上比作业还要大众化。对进程有很多种定义方法，其中包括：

① 一个执行的程序。

② 能分配给处理器并在其上执行的实体。

在下面的叙述中，进程这个概念将变得更为清楚。

计算机系统发展的 3 条主线导致实时性与同步性问题的出现，这些问题的产生有助于进程概念的完善，这 3 条主线是：批处理多道程序操作系统、分时系统以及实时事务处理系统。

① 批处理多道程序。其功能是，使处理器、I/O 设备（包括存储设备）能同时工作，使系统得到最大效率。关键技巧在于：在响应 I/O 处理完毕的信号时，处理器在驻留在主存内的各种程序中进行切换。

② 分时系统。其理论基础在于，如果计算机用户能够通过某种终端与计算机直接作用的话，则工作效率会更高。这里的关键在于，系统能够响应多个用户的请求，同时，考虑耗费等因素。如果一个典型的用户平均每分钟需要 2s 的处理时间，那么，每分钟就可以允许有接近 30 个用户同时共享同一个系统而没有明显的干扰，当然，OS 的开销时间必须要考虑。

③ 实时事务处理系统。在这种情形下，一些用户会对数据库进行查询和更新，航空预定系统就是一例。实时事务处理系统与分时系统的主要区别在于前者局限于一个或较少的应用程序，而分时系统的用户却可以涉及程序的开发、作业的执行和不同应用程序的应用。这两种情形下，系统响应时间都是很快的。

系统程序设计员用于开发早期的多道程序和多用户系统的原则是中断。任何作业在遇到规定的事件(例如，I/O)时就会被挂起，处理器保存某些内容(例如，程序计数器和其他寄存器)，然后转到中断处理程序，中断处理程序将确定中断、处理中断，然后继续执行用户被中断的作业和其他一些作业。

设计系统软件以协调各种组件间的运作是很困难的。在同一时刻有许多任务，而每个任务又包括大量必须依次执行的步骤，因此，对所有事件的可能的组合顺序进行分析是不可能的。由于系统缺乏协作的方法，常会产生一些错误。这些错误很难诊断，因为必须将它们与应用程序的出错及硬件出错区分开。即使发现了错误，也很难确定原因，因为出现错误的精确环境是很难再现的。通常，有 4 种原因会导致这样的错误：

① 不合适的同步。一个进程常常因等待某事件发生而必须挂起。例如，一个程序执行初始化 I/O 读操作后，必须等待直到缓冲区中有所需的数据。不合适的信号机制可能会导致信号丢失或收到多个信号。

② 失败的互斥。有些情况下，不止一个用户或程序企图同时使用共享资源。例如，在一个航空定票系统中，两个用户可能分别试图读取数据库，如果有空余座位，就订下座位并更新数据库。如果访问没有得到控制，就可能出现错误。必须有一种互斥机制，即在同一时刻只允许一个进程对数据区域进行更新。这种互斥的实现很难在所有可能的事件顺序下被证明是正确的。

③ 非确定的程序操作。一个特定程序的运行结果通常是由该程序的输入决定而

不依赖于一个共享系统中其他程序的运行。但是，如果多个程序共享存储器并都被处理器调入内存，就有可能修改共享存储器区从而互相干扰。这样，各程序的调度次序就会影响其他程序的输出结果。

④ 死锁。在某些情况下，可能有两个或多个程序都在互相等待彼此占用的资源。例如，两个程序可能都需要两个 I/O 设备。每个程序都控制了其中一个设备而等待另一个程序释放另一个设备。

解决这些问题所必需的基本条件是，要有一个系统的方法去监视和控制处理器中运行的程序。进程的概念提供了这个基本条件。进程由以下 3 部分组成：

① 一个可执行的程序；

② 该程序所需的相关数据(变量、工作空间，缓冲区等)；

③ 该程序的执行上下文(Context)。

其中，最后一项是必不可少的。操作系统中所有用来管理进程和处理器执行进程的信息都包括在执行上下文中。这个上下文包括寄存器的内容、进程的优先级以及进程是否等待 I/O 事件的完成等信息。

图 1.10 显示了进程可能实现的一种方法。两个进程 A 和 B 存在于内存中。每个进程在进程表列中都有记录，并且由操作系统保存。进程表列保存了每个进程的表项，包括指向进程在内存中存储块的指针。这个表项可能还包括进程执行上下文

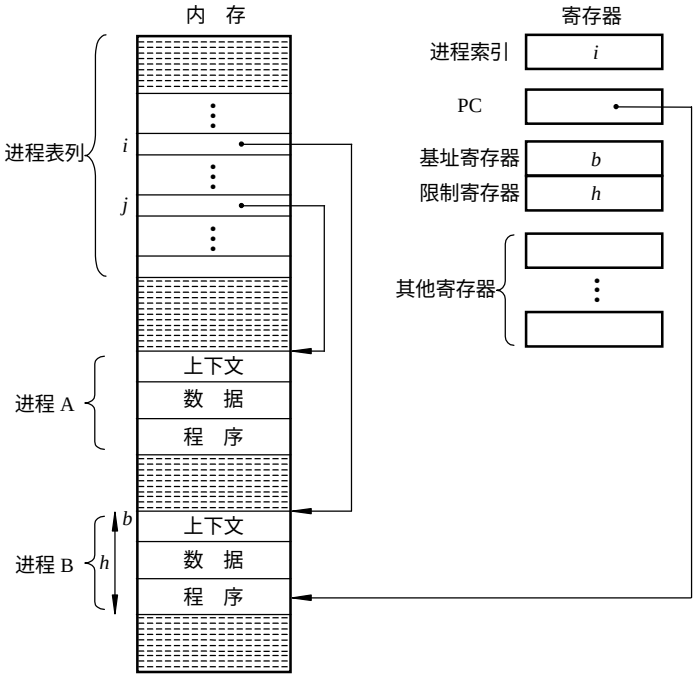


图 1.10　典型的进程实现

的一部分。执行上下文的剩余部分则同进程自身一起存放。进程索引寄存器存储当前处理器的进程在进程表列中的索引。程序计数器指向进程中下一条执行指令。基址(Base)和限制(Limit)寄存器定义了进程在内存中占有的区域。程序计数器和所有数据的访问都被解释为基本寄存器的相对值，并且不能超过限制寄存器的值。这样就防止了进程间的相互干扰。

在图 1.10 中进程 B 正在执行。进程 A 已被暂时中断。A 被中断时，所有寄存器的内容都记录在其执行上下文中。以后，处理器可以进行上下文切换而重新执行进程 A。上下文切换包括存储 B 的上下文和恢复 A 的上下文。

这样，进程可以看做是一个数据结构。进程既可以被执行，又可以等待执行。进程的整个状态都保存在其上下文中。可以扩展进程的上下文以允许加入新的信息。

1.3.2　存储器管理

用户需要一个计算环境，以支持组件编程和灵活使用数据。系统管理员需要高效和有序地存储分配控制机制。为了满足这些要求，操作系统有如下 5 条存储管理原则：

① 进程隔离。操作系统必须防止独立进程之间的相互干扰。

② 自动分配和管理。程序应该能够按要求的等级动态分配到存储区。这个过程对程序员是透明的。这样，程序员就不必关心存储区的限制，由操作系统根据任务的需求分配存储器，其效率也得到提高。

③ 支持组件编程。存储器共享使得一个程序具有对另一程序的存储空间进行寻址的潜在可能性。有时这是需要的，有时则对绝大多数程序甚至操作系统本身构成极大威胁。

④ 长时间存储。很多用户和应用程序要求长时间存储信息。

⑤ 保护和存取控制。

通常，操作系统用虚拟存储器(Virtual Memory)和文件系统来满足这些需要。虚拟存储器允许程序以逻辑方法来寻址，而不用考虑物理上可获得的内存大小。当一个程序执行时，事实上，只有一部分程序和数据可以保存在内存中，其他部分则存储在磁盘上。在以后的章节里，我们将看到，这种将存储空间分为物理空间和逻辑空间的方法为操作系统提供了强大的支持。

文件系统将信息存储在称为文件的一个命名了的对象中，实现了长期存储。文件是一个对程序员很方便的概念，并且对操作系统而言是存取控制和保护的单元。

图 1.11 描述了对存储系统的两种看法。从用户的角度出发，处理器同操作系统一起为用户提供了一个“虚拟处理器(Virtual Processor)”，它访问虚拟存储空间。这种存储可以是线性地址空间或一个段的集合(段是可用长度的连续地址块)。不论是哪种情况，编程指令都可以对虚拟内存中的程序和数据进行访问。进程分隔可以通过

给每个进程一个惟一的、不可重叠的虚拟内存空间来实现。进程共享则可以通过将两个虚拟存储空间重叠来获得。文件存储在长期存储介质上。文件可以被复制到虚拟内存中。

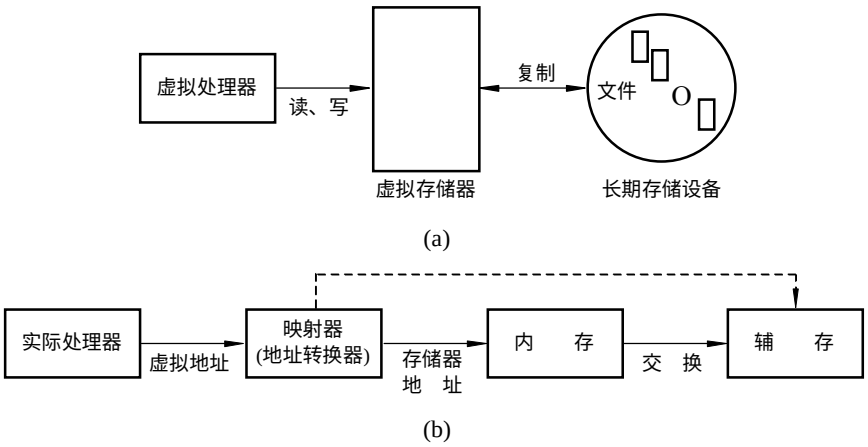


图 1.11　看待存储系统的两种不同观点

(a) 用户观点　(b) 操作系统设计者观点

从设计者的角度来看存储器，存储器由(机器指令)可直接寻址内存和通过将数据块加载到内存来间接访问的辅存组成。地址转换硬件——映射器(Mapper)插在处理器和存储器之间。程序用虚拟地址进行访问，虚拟地址映射到实际的内存地址。如果一个访问并不在实际内存中，那么实际内存中的一块内容就会被交换到辅存，从而可以将一个需要的数据块交换进内存。在这个过程中，产生这个地址访问的进程将被挂起。

1.3.3　信息保护和安全性

随着分时系统的广泛应用以及近来计算机网络的发展，人们对信息保护的关注也越来越强烈。

美国国家标准局(National Bureau of Standards)颁布了一个文件，指出了一些安全领域要警惕的恐怖活动：

① 有组织地、故意地企图从竞争对手处获取经济或市场信息。

② 有组织地、故意地企图从政府机构获取经济信息。

③ 非故意地获取经济或市场信息。

④ 非故意地获取私人信息。

⑤ 通过非法访问计算机数据，故意获取资金数据、经济数据、法律执行数据以及私人信息。

⑥ 政府侵犯私人权利。

为防止这些活动，可以将一些通用工具安装到计算机和操作系统中以支持各种保护和安全机制。通常，主要关注对计算机系统和存储在其中的信息的存取控制。以下是 4 种保护策略：

① 不共享。在这种情况下，进程完全相互分隔开，每个进程对所分配到的资源都具有排它控制权。在这种策略中，为了共享程序或数据的安全，相关进程需要将程序的数据复制一份到自己的虚拟内存中。

② 共享原始程序或数据文件。一个物理上存在的程序可以在多个虚拟地址空间中出现。为了防止在同一时刻，多个用户相互干扰，对可写的数据文件共享需要特殊的加锁机制。

③ 无存储子系统。在这种情况下，进程被分为多个子系统组。例如，一个客户(Client)进程调用一个服务(Server)进程去完成一些任务。服务进程受到保护不会让客户进程发现完成该任务的算法。而客户进程也受到保护，不会让服务进程保留该任务的任何信息。

④ 控制信息的分布。在某些系统中，定义了一些安全类。用户和应用程序都要进行安全检查，而数据和其他资源(I/O 设备等)都有一个安全级别。在这种安全策略中，哪些用户对哪些级别具有访问权均有明确规定。这种模式在军事和商业应用中非常重要。

同操作系统有关的安全和保护工作可分为以下 3 类。

① 访问控制。管理用户对整个系统、子系统和数据的访问，并且管理进程对系统中各种资源和目标的访问。

② 信息流控制。管理系统中的数据流以及传递给用户的数据流。

③ 确认。它同以下两点相关：

- 所提供的访问及信息流控制机制。
- 这些机制所支持的访问及安全策略。

1.3.4　调度和资源管理

操作系统的核心任务之一就是管理各种可获得的资源以及合理地调度它们。任何资源分配和调度策略都必须考虑以下 3 个因素：

① 公平性。通常希望给竞争资源的进程以平等的访问权限。

② 不同敏感性。操作系统应该区分具有不同服务请求的不同类型的任务。对资源而言，操作系统既应该为了满足全局的需要统筹分配和调度，又应该根据具体情况动态地分配和调度。例如，如果一个进程正在等待一个 I/O 设备，则操作系统应在该 I/O 设备空闲出来后，立刻调度该进程执行。

③ 效率。在公平和效率的限制下，操作系统最好能有最大化吞吐量、最小化响

应时间，对于分时系统，则应尽可能为更多的用户服务。

图 1.12 展示了在多道程序环境中，操作系统的关键组件。操作系统保存了一些队列，每个队列都是一组等待某些资源的进程。短程队列是由在主存中并处于就绪状态的进程组成。这些进程中的任何一个进程都可以作为处理器的下一个选用者，这由分派程序决定。

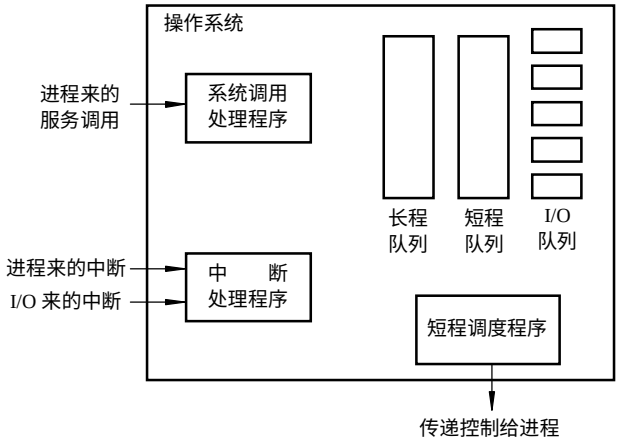


图 1.12　多道程序操作系统的关键组件

长程队列是一列等待使用系统的新进程。操作系统通过将一个进程从长程队列转移到短程队列，向系统增加任务。每个 I/O 设备都有一个 I/O 队列。等待使用某个设备的所有进程都排在该设备队列中。

当中断发生时，操作系统得到处理器的控制权。一个进程可能会求助于操作系统提供的一些服务。在这种情况下，服务调用处理程序成为进入操作系统的入口。

1.3.5　系统结构

随着操作系统性能的增强，以及基础硬件复杂性的增加，操作系统的大小和复杂性也不断增加。CTSS 分时系统在 MIT 于 1963 年投入使用时，大约有 32 000 个 36 位字的存储容量。一年以后出现的 IBM OS/360，有超过一百万条的机器指令。1972 年底，MIT 和贝尔试验室共同开发的 Multics 系统有超过两千万条的机器指令。确实，近年来出现了一些在小系统上运行的较小的操作系统，但随着硬件技术的发展和人们需求的增长，这些系统也越来越复杂了。简单的 PC-DOS 已让路于复杂的功能强大的 OS/2。

操作系统的大小及其处理任务的难度导致了许多问题，比如，系统总会有许多潜在的错误(Bugs)，以及性能没有达到所希望的要求。为了有效管理系统资源和控

制操作系统的复杂性，人们开始极大地重视操作系统的软件结构。

一个明显的观点是软件必须组件化，这有助于组织软件、限制诊断任务并定位错误；组件间的接口应尽可能简单，这使得系统改进更为容易。有了简洁的接口，就会将改变一个组件后对其他组件产生的影响降到最低程度。

对大型操作系统，仅仅组件化编程还是不够的。现在越来越多地用到体系结构分层和信息抽象技术。现代操作系统的体系结构分层是根据其复杂性以及抽象的水平来分离功能的。可以将系统看成一个分层结构，每层完成操作系统要求的一个功能子集，每层都依赖紧挨着的较低一层的功能，并且为较高层提供服务。定义不同的层就是为了当某一组件改变时不会影响其他层的内容，将一个问题分解为许多容易解决的子问题。

表 1.4 所示的是一个层次操作系统的模式。

表 1.4　操作系统设计和层次结构

层	名　称	对　象	操作举例
13	外壳	用户程序设计环境	shell 语言中的语句
12	用户进程	用户进程	quit, kill, suspend, resume
11	目录	目录	create, destroy, attach, detach, search, list
10	设备	外设：打印机、显示器等	create, destroy, open, close, read, write
9	文件系统	文件	create, destroy, open, close
8	通信	管道	create, destroy, open, close, read, write
7	虚拟存储器	段、页	read, write, fetch
6	局部辅存	数据块、设备通道	read, write, allocate, free
5	进程原语	进程原语、信号量、就绪队列	suspend, resume, wait, signal
4	中断	中断处理程序	invoke, mask, unmask, retry
3	过程	过程、调用栈、显示	mark stack, call, return
2	指令集	演算栈、微程序解释器	load, store, add, subtract, branch
1	电子线路	寄存器、门电路、总线等	clear, transfer, activate, complement

• 第 1 层由电路组成，其中的对象是寄存器、门电路和总线等。对这些对象的操作是一些动作，如清除寄存器或读取内存单元等。

• 第 2 层是处理器的指令集。这一层的操作是那些机器语言指令集所允许的一些指令，如 add、subtract、load 和 store 等。

• 第 3 层加入了过程的概念，包括调用返回指令等。

• 第 4 层是中断，使处理器保存当前内容并调用中断处理程序。

这 4 层并不是操作系统的一部分，但它们组成了处理器硬件。然而，操作系统中的一些元素，如中断处理程序，已在这一层出现。

• 第 5 层中，进程作为程序的执行在本层出现。为了支持多进程，对操作系统的基本要求包括具有挂起和重新执行进程的能力。这就要求保存寄存器的值以便从一个进程切换到另一个进程。

• 第 6 层管理计算机的辅存。这一层的主要功能有读/写扇区、进行定位以及传输数据块。第 6 层依靠第 5 层的调度操作。

• 第 7 层为进程创建逻辑空间。这一层将虚拟空间组织成块，并在主、辅存间调度。当一个所需块不在主存中时，本层将逻辑地要求第 6 层传输。

• 第 8 层处理进程间的信息和消息通信。其最有力的工具之一是管道(Pipe)。管道是进程间数据流的一个逻辑通道。它也可用来将外部设备和文件同进程连起来。

• 第 9 层支持长期存储文件。

• 第 10 层是利用标准接口，以提供对外部设备的访问。

• 第 11 层负责保存系统资源和对象的外部和内部定义间的联系。外部定义是应用程序和用户可以使用的名字。内部定义是能够被操作系统的低层部分用来控制一个对象的地址或其他指示符。

• 第 12 层支持所有管理进程所必须的信息。包括进程虚拟地址空间、与该进程有相互作用的进程和对象表列、创建该进程时传递的参数等。

• 第 13 层在操作系统同用户间提供一个界面。它被称为“外壳(shell)”，这是因为它将用户和操作系统具体实现区分开并使操作系统就像一个功能的集合。这个外壳接受用户命令，解释后根据需要创建并控制进程。

1.4　操作系统举例

本书主要介绍操作系统的设计原理与实现技术，并选择以下 3 个系统作为例子：

① Windows NT。这是一个单用户、多任务操作系统，它运行在各种 PC 机和工作站上。它包含了操作系统最新的各种技术。

② UNIX。这是一个多用户操作系统。开始是为小型机设计的，后来广泛用于从微机到巨型机的各种机型。

③ MVS。这是 IBM 大型机的最新操作系统，是到目前为止最复杂的操作系统之一。它提供了批处理和分时能力。

1.4.1　Windows NT

1. 历史

Windows NT 的最久远的历史是 Microsoft 公司为最初的 IBM PC 设计的 MS-DOS 或 PC-DOS。最早的版本 DOS 1.0，在 1981 年 8 月发布。它由 4000 行汇编

代码组成，运行于 8KB 内存中，使用 Intel 8086 微处理器。随着 Intel 不断推出功能越来越强大的 CPU 以及 PC 领域各种新技术的问世，Microsoft 不断推出新的 DOS 版本。但 DOS 并不能充分发挥一些新的 CPU 的强大功能。

早在 20 世纪 80 年代早期，Microsoft 就开始研究图形用户界面(GUI)以方便使用。到 1990 年，Microsoft 已经有了 GUI 的一个版本 Windows 3.0。Windows 3.0 界面友好，达到 Macintosh 机的水平，但仍需要 DOS 的支持。随后，Microsoft 开发出自己的 Windows NT。Windows NT 与 Windows 3.x 建立在完全不同的概念之上，它能充分发挥当前处理器的能力，提供了完全的多任务、单用户环境。

2. 单用户多任务

Windows NT 代表了个人计算机操作系统的新潮流，还有 OS/2、Macintosh System 7。这些操作系统的一个显著特征是，尽管他们只是支持单一用户，但它们支持多任务。主要有以下两个因素促使个人计算机需要多任务：

① 随着处理器速度增加、内存容量增大以及对虚拟内存的支持，应用程序变得越来越复杂并相互依靠。用户可能希望同时运行文字处理程序、图像处理程序和表格程序以创建一个文档。如果没有多任务，仅希望创建一幅图画，然后粘贴到正在处理的文件中，就必须进行如下步骤：打开图像处理程序；创建图像并将其保存在文件或剪贴板中；关闭图像处理程序；打开字处理程序；最后，加入图像。

如果要改动图像，则必须先关闭字处理程序，再打开图像处理程序并按以上步骤重做一遍。这样，单任务环境就显得太不灵活，对用户也不太友好。在多任务环境中，用户根据需要打开每个应用程序，并使它们全处于打开状态。信息很容易在应用程序间传递。每个应用程序可以有一个或多个窗口。用户使用一种指示设备如鼠标，就可以很方便地在这个环境中进行切换。

② Client/Server(简写为 C/S)计算方式的发展。在 C/S 计算中，一台 PC 机或工作站(client)和一个主系统(server)一起用来完成一个特定的应用程序。这两者连接在一起并且每个都被分配了适合其能力的一部分任务。C/S 计算可以从一个由 PC 机和服务器组成的局域网获得，也可以通过将一个用户系统同大型机连起来获得。一个应用程序可以包括一台或多台 PC 机和一个或多个服务器。为了提供合理的响应，操作系统必须支持精密的时间通信硬件、相关通信协议和数据传输体系。

3. 说明

很多因素都影响 Windows NT 的设计。它提供了早期 Windows 产品中相同类型的 GUI，包括窗口的使用、下拉菜单和点击交互方式。它内部结构受到 Mach 操作系统的启发，而 Mach 又建立在 UNIX 基础上。

图 1.13 描述了 Windows NT 的总体结构。高度组件化给 Windows NT 带来相当的弹性。Windows NT 能够运行于许多硬件平台上，并支持为许多其他操作系统编写的应用程序。

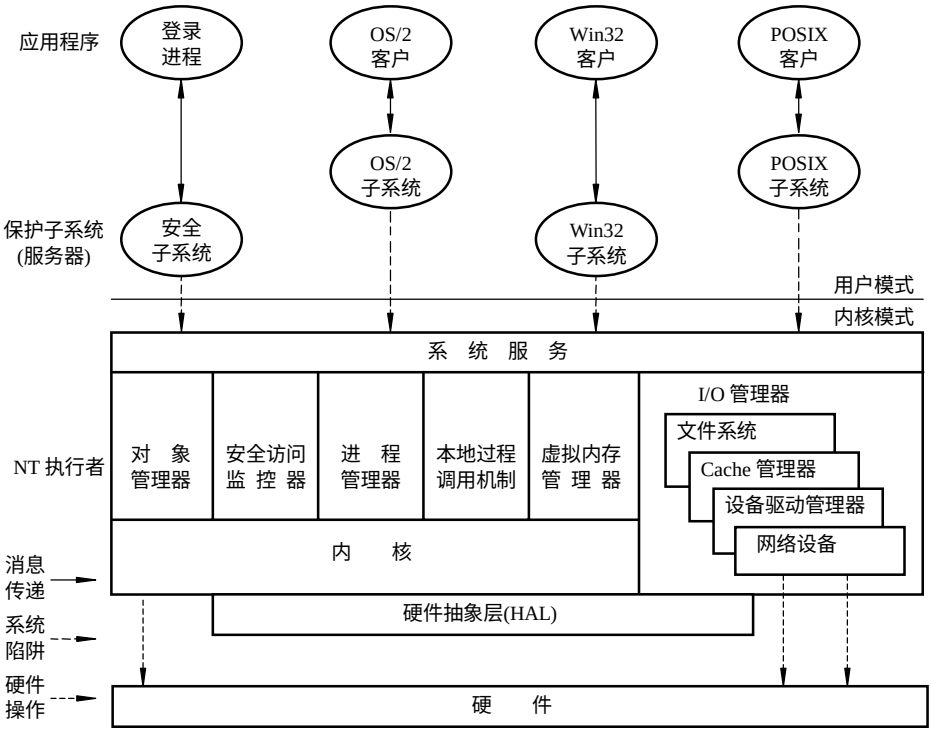


图 1.13　Windows NT 结构

同绝大多数操作系统一样，Windows NT 将应用软件与操作系统软件分开。后者运行于特权模式或内核模式下。内核模式可访问系统数据和硬件。其余软件运行于用户模式下，有限制地访问系统数据。内核模式软件是 Windows NT 的执行者。

不论是运行在单处理器或多处理器，还是运行在 CISC 或 RISC 系统上，大多数 Windows NT 都以同一角度看待低层硬件。为了达到这一独立性，操作系统由以下 4 层组成：

① 硬件抽象层(Hardware Abstraction Layer，简称 HAL)。将硬件命令、响应等映射到一个特定的平台(如 Inter 486、奔腾处理器或 Alpha 处理器)上。HAL 使每一个机器的系统总线、DMA 控制器、中断控制器、系统时钟和存储控制器对内核来说都是相同的。

② 内核(Kernel)。由操作系统最常用、最基础的构件组成。内核管理调度、上下文的切换、中断处理以及多处理器同步。

③ 子系统(Subsystem)。包括利用内核提供的基础服务的各种特殊功能组件。

④ 系统服务。提供与用户模式软件的接口。

有一个子系统——I/O 管理器可绕过 HAL 直接同硬件相互作用。这对于提高 I/O 操作的效率和吞吐量是很有必要的。

Windows NT 的威力来自于其能为其他系统编写的应用程序提供支持。Windows NT 提供这一支持的方法是通过保护子系统(Protected Subsystem)实现的。保护子系统是 Windows NT 同终端用户交互的部分。保护子系统提供了一个图形或命令行用户界面来定义操作系统的界面。另外，每个保护子系统为特殊的操作环境提供了应用程序接口(Application Programming Interface，简称 API)。这意味着为一个特定环境编写的程序可能会毫无改变地运行于 Windows NT 中，因为它们所接触的操作系统接口与其编程的环境是一样的。例如，基于 OS/2 的应用程序不必改动就可以运行于 Windows NT 中。在很多情况下，只需对程序重新编译一次。

C/S 体系结构是分布计算的通用模式。这一模式也适用于单个系统内部，就像 Windows NT 一样。

每个服务器由一个或多个进程实现，每个进程等待客户对其服务的请求，例如，存储器服务、进程创建服务、处理器调度服务等。客户可以是应用程序或其他操作系统组件，它通过发送消息提出请求。消息通过执行者传送到适当的服务器。服务器进行所需的操作，并通过执行者向客户返回所需的信息。

4. 线程

Windows NT 的一个重要方面是它对进程里面线程的支持。线程包含了一些传统上与进程有关的功能。我们将在第 3 章详细介绍进程和线程及其区别。

5. 对称多处理

至今为止，几乎所有的单用户个人计算机和工作站都只有一个 CPU。随着对性能要求的不断增加以及 CPU 花费的降低，越来越多的系统采用了多个处理器。为了获得最大的效率和可靠性，一个称为对称多处理(Symmetric Multiprocessing，简称 SMP)的操作模式是必要的。有了 SMP，每个进程或线程可分配给任意一个处理器。

现代操作系统一个关键要求就是充分开发利用 SMP。Windows NT 对 SMP 的支持有以下性质：

① 操作系统的进程可以在任何可获得的处理器上运行。

② Windows NT 支持单个进程中多个线程的运行。相同进程中的多个线程可以在不同处理器上同时运行。

③ 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要求。

④ Windows NT 提供了方便的机制，以在进程间共享数据和资源，以及灵活的进程间通信能力。

6. Windows NT 对象

Windows NT 很重视面向对象设计(OOD)的概念。这种方法很容易在进程间共享数据和资源，并且对于未授权的访问提供对资源的保护。Windows NT 中使用的主要面向对象概念有：

① 封装(Encapsulation)。每个对象由一个或多个数据项(称作属性，Attribute)，

以及一个或多个可以对数据进行处理的过程(称作服务，Service)组成。访问一个对象中的数据的惟一方法是激活该对象的服务。这样，对象的数据就被保护起来，以防止非授权的使用或错误使用。

② 对象的类(Class)和实例(Instance)。一个对象的类就是列出了该对象的属性和服务并定义了该对象的一些性质的模板。这种方法简化了对象的创建和管理。

Windows NT 中并非所有实体都是对象。只有当数据对用户模式访问是公开的或数据访问是共享或受限制时才使用对象。通过对象代表的实体有：文件、进程、线程、信号量(Semaphore)、时钟和视窗(Windows)。Windows NT 通过内核中的对象管理器创建和管理所有的对象。

一个对象可以通过打开其句柄对另一对象进行访问。句柄实质上是一个指向对象的指针。在 Windows NT 中，对象可以命名或不命名。当进程创建一个不命名对象时，对象管理器返回一个句柄，并且这个句柄是访问该对象的惟一方法。对命名对象，其他进程可以使用这个名字去获得这个对象的句柄。命名和不命名对象的一个重要区别是，命名对象通常有安全信息(以访问令牌的形式)，安全信息可以用来限制对对象的访问。

Windows NT 并不是一个完全的面向对象操作系统。它也不是用面向对象语言实现的，其完全驻留在执行者组件中的数据结构并不是用对象描述的。

1.4.2　UNIX System V

1. 历史

UNIX 开始是由贝尔实验室开发并最初于 1970 年在 PDP-7 上实现的。贝尔实验室和其他一些部门在 UNIX 上的研发工作，导致一系列 UNIX 版本的产生。第一个具有里程碑意义的工作是将 UNIX 系统由 PDP-7 转移到 PDP-11 上，这预示着：UNIX 是一个适合所有计算机的操作系统。下一个具有重要里程碑意义的工作是用 C 语言重写 UNIX，这是前所未有的。通常认为由于 OS 要处理实时事件，故只能用汇编语言编写，现在，几乎所有 UNIX 都由 C 语言实现。

1974 年，有关 UNIX 的报导第一次在一份科学杂志上刊出并引起广泛注意。第一个在贝尔实验室以外可以广泛获得的 UNIX 是 1976 年的第 6 版。1978 年的第 7 版可被看做是当今 UNIX 系统的祖先。非 AT&T 系统的 UNIX 是由加州大学 Berkeley 分校开发的 UNIX BSD，可运行于 VAX 机器上。1982 年，贝尔实验室将 AT&T 中的几个 UNIX 版本合并成一个单一的系统 UNIX System Ⅲ，并在市场上推出。

本书将 UNIX System V 作为 UNIX 例子。UNIX System V 是主流的商业版本，可在从 32 位微处理器到超级计算机上运行。

2. 说明

图 1.14 所示的是一个一般 UNIX 的体系结构，其底层硬件被操作系统软件包围。

操作系统通常称做系统内核或内核，以便同用户和应用程序区分开来，UNIX 的这一部分就是在本书中要介绍的。当然，UNIX 还有一些用户服务接口也被视为操作系统的一部分，这些可归到外壳中。外壳还包括接口软件和 C 编译器部分(编译器、汇编编译器、加载程序)。这一层以外就是用户应用程序和 C 编译器的用户界面。

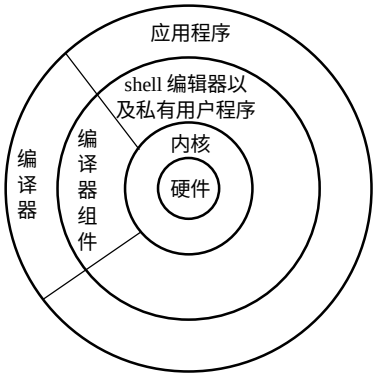


图 1.14　一般的 UNIX 体系结构

图 1.15 则进一步分析了内核。用户程序能够直接或通过库程序调用操作系统服务。系统调用接口是内核与用户的界面，并允许高层软件访问特定的内核功能；操作系统中包含的原语操作直接与硬件起作用。在这两个界面间，系统被分为两个部分，一个主要从事进程控制，另一个从事文件管理和 I/O 控制。进程控制子系统负责内存管理、进程调度以及进程同步和线程间通信。文件系统在内存与外部设备之间以字符流或块的方式交换数据。

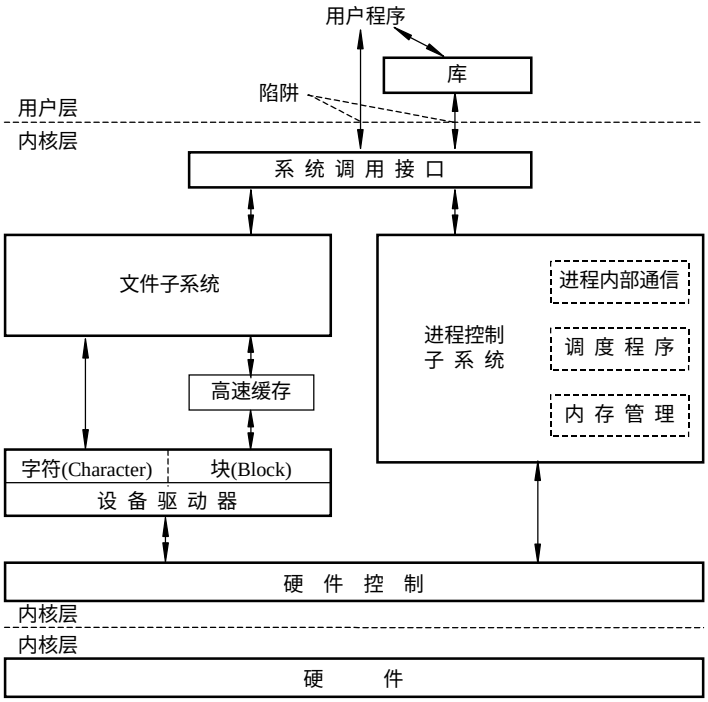


图 1.15　UNIX 系统内核分块图

1.4.3　MVS

1. 历史

在1964年，IBM发布了一个新的计算机家族System/360，其各种不同型号都是兼容的。运行在这些机器上的第一个操作系统是OS/360。最初的OS/360是一个多道程序批处理系统。OS/360最完整的版本MVT(Multiprogramming with a Variable number of Tasks)是OS/360各演化版本中最灵活的，发布于1969年。MVT只允许同时运行15个任务。

IBM在1972年发布了OS/SVS(Single Virtual Storage)作为一个过渡操作系统。其增加的最显著的功能是支持虚拟内存。SVS共建立了16MB的虚地址空间，此空间由操作系统和所有激活的任务共享，但很快就不够用了。

为了满足不断增长的内存需要，IBM发布了MVS。在SVS中虚拟地址是24位的，因此为16MB。而在MVS中，每个任务都有16MB的虚拟内存。操作系统将虚拟内存块映射到实际内存，并能够对每个任务独立的虚拟内存进行跟踪。实际上，每个任务通常只需要使用8MB不到的虚拟内存，剩余的可以由操作系统使用。

即使每个任务都有单独的虚拟内存，24位地址空间对某些用户还是很快又不够用了。IBM使用一种称为扩展寻址(Extended Addressing，简称EA)的机制，将其底层处理器扩展成能处理31位地址。为了充分利用此新硬件，MVS的新版本——MVS/XA于1983年问世。有了MVS/XA，每个任务地址空间增长到最大2GB。但是，2GB对处于某些环境的某些应用程序仍被认为不够用。所以，IBM研制了企业系统体系(Enterprise System Architecture，简称ESA)和增强型操作系统MVS/ESA。在MVS/ESA中每个任务仍可得到2GB地址空间。所不同的是，对特殊任务，共有15个额外的2GB地址空间可用于存放数据。这样，最大可寻址虚拟内存变为每个任务都有32GB。

2. 说明

MVS 也许是曾经出现过的最大、最复杂的一个操作系统。它最少需 2MB 存储空间，但一个较理想的配置需要 6MB。MVS 的设计主要由以下 4 个因素决定：

① 支持批处理和相互作用的任务。

② 对每个任务或用户提供最高达32GB的虚拟内存。

③ 紧密耦合多处理(Tightly Coupled Multiprocessing)。这种体系结构由共享同一内存的多个处理器组成。

④ 精巧的资源分配和监控措施以获得对系统存储器、多处理器以及复杂的I/O管道结构的有效利用。

在OS/2和UNIX System V中不需要面对多个处理器，而MVS需要。当有多个处理器，每个处理器可以执行任一进程，同时又支持在不同处理器上运行的进程间通信时，这种操作系统的复杂性要比单处理器机器上的操作系统复杂性明显大得多。

图1.16是MVS主要模块的简单视图。外壳包括对用户和负责维护及协调系统的系统管理员可视的服务和界面。它除了生成和编译程序所需的程序以外，还有一个任务管理子系统，它有以下功能：

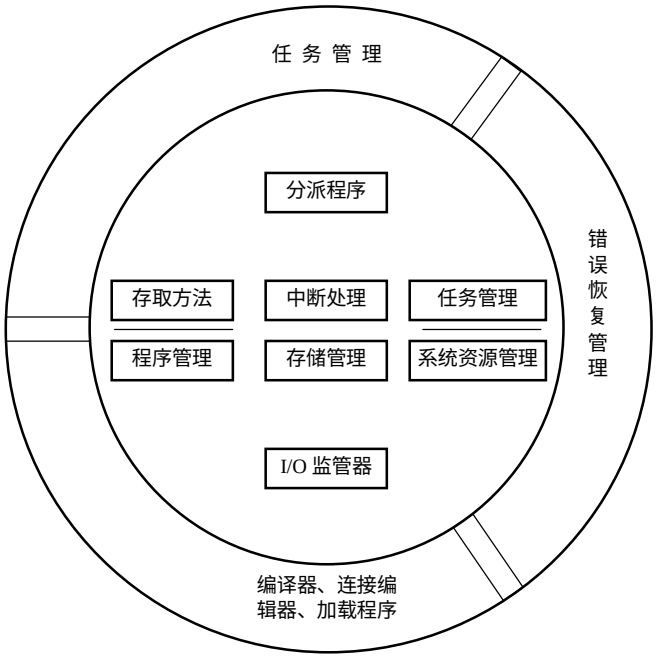


图 1.16　MVS 的组成部分

① 解释(从控制台来的)操作命令并发送合适的信息。

② 读任务输入数据或写任务输出数据。

③ 为任务分配I/O设备，并通知操作员在任务执行前准备好一些物理数据单元。

④ 将作业转换成可以被任务管理器处理的任务。

此外，外壳还包括一个恢复错误的详尽管理子系统，该子系统假设任务的错误是独立的，不会影响操作系统的其他部分，因此错误可以诊断出来。如果可能的话，被错误影响的任务将能够重新执行。

操作系统的内核由一组主要的组件或子系统组成，它们与硬件、外壳以及彼此间发生相互作用。包括：

① 分派程序。可以被看做是处理器的管理者，它的功能是扫描就绪队列并调度其中一个进程来执行。

② 中断处理。System/370支持众多中断。任何中断都会使当前进程挂起而且控制转移到相应的中断处理程序。

③ 任务管理。一个任务实际上就是一个进程。任务管理器负责任务的创建和删

除、优先级的控制、对任务队列的管理以及事件的同步。

④ 程序管理。这个组件负责将包含在一个程序执行中的必要步骤连接在一起。这个组件能够被JCL命令或因响应用户编译和执行程序的请求而驱动。

⑤ 存储管理。这个组件负责管理虚存和实存。

⑥ 系统资源管理(SRM)。这个组件负责进程中的资源分配。

⑦ 存取方法。一个应用程序通常采用一种存取方法以实现一个I/O操作。存取方法是应用程序与I/O监管器之间的接口。存取方法减轻了应用程序的负担，使其不用与管道程序、创建I/O监管器所需的控制块以及处理终止的情况相联系。

⑧ I/O监管器。I/O监管器在硬件一级实现I/O操作系统的初始化和完成I/O操作。

其他操作系统没有SRM的功能。资源的概念包括处理器、实际存储器和I/O通道。SRM积累关于处理器、管道使用以及各种关键数据结构的统计数据。其目的是根据对性能的监管和分析提供最佳的性能。开始时，设置了一些性能参数作为SRM参考；SRM再根据系统的使用情况动态修改任务性能特性。SRM还向操作员提供报告，使那些训练有素的操作员能优化系统配置和参数设置以改进对用户的服务。

1.5　操作系统的主要研究课题

操作系统的主要研究课题及其关系如图1.17所示。

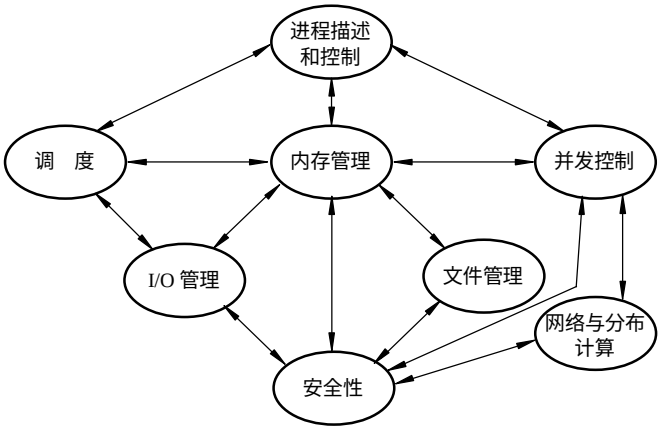


图 1.17　OS 研究课题

习　题　一

1.1　假设有一个支持多道程序设计的计算机系统，其中每个作业都有完全相同的属性。对一

个作业，在一段计算周期T中，一半的时间用于I/O，另一半时间用于处理器操作。每个作业总共运行N段周期。有几个定义如下：

• 周期(Turnaround time)=完成一个作业实际用的时间；

• 吞吐量(Throughput)=在每一时间段T中完成的平均作业数；

• 处理器使用率(Processor utilization)=处理器处于激活态(非等待)时间的百分比。

计算当有1个、2个或4个作业并行执行时的周期、吞吐量和处理器使用率，假设时间段T按以下任一种方式分布：

(1) I/O在前半段，处理器运行于后半段；

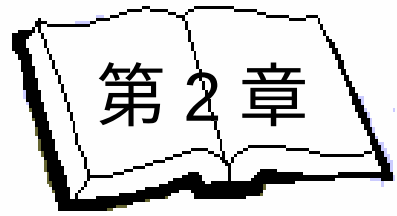
(2) 将T分为4段，I/O在第1、4段，处理器运行于第2、3段。

1.2　现定义等待时间为一个作业停留在短程队列(见图1.12)中的时间，请给出周期(Turnaround Time)、处理器繁忙时间(Processor Busy Time)以及等待时间之间的相关表达式。

1.3　偏重I/O型的程序用于等待I/O的时间较长，而使用处理器的时间较短；而偏重处理器的程序则相反。假设有一种短程调度算法满足最近使用处理器较少的进程。解释为什么这种算法对偏重I/O的程序有利，但也不会总是拒绝给予偏重处理器型程序使用处理器的时间。

1.4　某计算机用Cache、内存和磁盘来实现虚拟内存。如果某数据在Cache中，访问它需要t_A(ns)；如果在内存但不在Cache中，则需要t_B(ns)的时间将其装入Cache然后开始访问；如果不在内存中，需要t_C(ns)将其读入内存，然后用t_B(ns)读入Cache。如果Cache命中率为$(n-1)/n$，内存命中率为$(m-1)/m$，则平均访问时间是多少？

1.5　试举出几种为了使分时系统或多道程序批处理系统最佳化而使用的调度策略，并比较它们的优缺点。



进程描述与控制

从为单用户服务的系统(如 Windows NT)到可同时为上千个人服务的大型机系统(如 MVS)，所有的多道程序设计操作系统都建立在进程的概念之上。操作系统所应满足的主要要求都与进程有关：

① 操作系统必须同时执行多个进程以最大程度地利用处理器，同时，必须对每个进程提供合适的响应时间。

② 操作系统必须按一定的策略为进程分配资源(例如，某些函数和应用程序应有较高优先级)，同时必须避免死锁。

③ 操作系统应该支持进程内部通信和用户创建进程。

由于进程是操作系统的核心，因此，我们对操作系统的学习从进程开始。

2.1 进程状态

处理器的主要功能就是执行驻留在内存中的指令。为了提高效率，处理器可以同时执行多个进程。这样，从处理器的角度来看，其对全部指令的执行是按照一定次序的，这个次序是通过改变程序计数器(PC)或指令指示器(IP)里的值来排定的。随着时间的变化，PC或IP会指向不同程序中的不同代码段。从单个程序角度看，其执行由一系列代码段的执行组成。单个程序的执行被称为进程(Process)或任务(Task)。

单个进程要以通过列出按顺序执行的指令集来描述，这种方法称做进程的跟踪。

现在来看一个简单例子。图2.1显示在内存中有3个进程。为简化讨论，我们假设没有用到虚拟存储器，所有3个进程全部都存储在内存中。另外，还有一个分派程序(Dispatcher Program)负责把处理器分配给进程。图2.2列举了3个独立进程开始的一部分指令，其中，α为进程A的起始地址；β为进程B的起始地址；γ为进程C的起始地址。分别有进程A和C开始的12条指令，以及进程B的4条指令，并假设进程B的第4

条指令包括I/O操作。

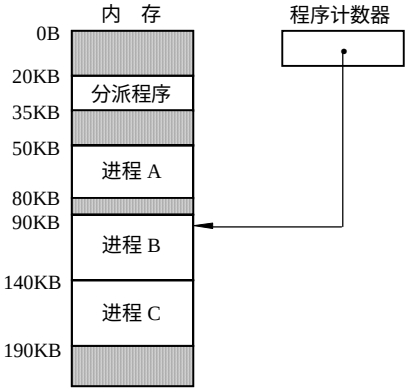


图 2.1　内存中 3 个进程的例子

(a)	(b)	(c)
$\alpha+0$	$\beta+0$	$\gamma+0$
$\alpha+1$	$\beta+1$	$\gamma+1$
$\alpha+2$	$\beta+2$	$\gamma+2$
$\alpha+3$	$\beta+3$	$\gamma+3$
$\alpha+4$		$\gamma+4$
$\alpha+5$		$\gamma+5$
$\alpha+6$		$\gamma+6$
$\alpha+7$		$\gamma+7$
$\alpha+8$		$\gamma+8$
$\alpha+9$		$\gamma+9$
$\alpha+10$		$\gamma+10$
$\alpha+11$		$\gamma+11$

图 2.2　图 2.1 中进程的轨迹

(a) 进程A的轨迹　(b) 进程B的轨迹

(c) 进程C的轨迹

现在从处理器的角度来看一下进程的轨迹。图2.3显示3个进程的执行情况，这3个进程由52个指令周期组成。这里假设操作系统在每个进程连续执行6个指令周期后会产生中断。这就防止了进程独占处理器时间的情况发生。如图2.3所示，操作系统在进程A的前6个指令执行后，产生超时(Time Out)中断，这时，分派程序开始执行，它将处理器分配给进程B(实际上，进程和分派程序的指令数并不会如图2.3所示的那么少，这里只是为了说明的方便)。进程B执行4条指令后，由于I/O操作必须等待，因此，处理器停止执行B，并由分派程序分配给C。经过超时中断，处理

1	α+0	7	δ+0	17	δ+0	29	δ+0	41	δ+0
2	α+1	8	δ+1	18	δ+1	30	δ+1	42	δ+1
3	α+2	9	δ+2	19	δ+2	31	δ+2	43	δ+2
4	α+3	10	δ+3	20	δ+3	32	δ+3	44	δ+3
5	α+4	11	δ+4	21	δ+4	33	δ+4	45	δ+4
6	α+5	12	δ+5	22	δ+5	34	δ+5	46	δ+5
----	超时	13	β+0	23	γ+0	35	α+6	47	γ+6
		14	β+1	24	γ+1	36	α+7	48	γ+7
		15	β+2	25	γ+2	37	α+8	49	γ+8
		16	β+3	26	γ+3	38	α+9	50	γ+9
		---	I/O请求	27	γ+4	39	α+10	51	γ+10
				28	γ+5	40	α+11	52	γ+11
				----	超时	----	超时	----	超时

δ为分派程序的起始地址

图 2.3　图 2.1 所示的 3 个进程混合执行过程

器又被分配给 A。A 超时中断后，B 仍在等待 I/O 操作结束，因此，又轮到进程 C。

2.1.1　进程产生和终止

操作系统的一个主要职责就是控制进程的执行。为有效地设计操作系统，必须了解进程的运行模型。

最简单的模型是基于这样一个事实：进程要么正在执行，要么没有执行。这样，一个进程就有两种状态：运行(Running)和非运行(Not-running)，如图2.4(a)所示。操作系统在产生一个进程之后，就将该进程加入到非运行系统中。这样，操作系统可以知道存在的进程数，而进程则等待机会执行。每隔一段时间，正在运行的进程就会被中断运行，分派程序将选择一个新进程运行，原先运行的进程由运行状态变为非运行状态，另一个进程则由非运行状态变为运行状态。这个模型尽管很简单，但已经能够帮助我们认识到操作系统设计的一些复杂性了：每个进程必须以某种方式代表，以便操作系统能够对其进行跟踪，也就是说，必须有一些与进程相关的信息，包括当前的状态和内存中的地址等；那些非运行状态的进程要放在一个排序队列中，以等待运行。图2.4(b)提出了一种方法，即设计一种进程队列，队列中的每项都是一个指向进程的指针。还有一种方法是队列由一些连接在一起的数据块组成，每个数据块代表一个进程。

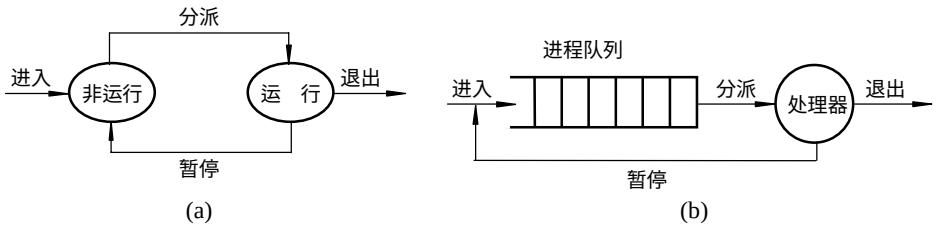


图 2.4　两状态进程模型

(a) 状态转换图　(b) 进程队列图

分派程序的行为可以用队列图示的方式描述。一个进程在中断执行后，就被放入等待进程队列中。如果进程结束或运行失败，就会被注销(Discarded)即退出系统。无论遇到哪种情况，分派程序都会选择一个新进程运行。

在完善简单的两状态进程模型之前，有必要讨论进程的产生和终止。无论采用哪种模型，进程的生命周期都同产生和终止联系在一起。

1. 进程产生

当有一个新进程要加入当前进程队列时，操作系统就产生一个控制进程的数据结构(将在2.2节讨论)并且为该进程分配地址空间。这样，新进程就产生了。

通常有4种事件会导致新进程产生，如表2.1所示。一是在一个批处理环境中，为了响应一个任务的要求而产生进程。二是在一个交互式环境中，当一个新用户企图登录时也会产生进程。在这两种情况下，进程的产生均由操作系统负责。三是操

作系统代用户程序产生进程，如果用户想要打印一个文件，则操作系统就会产生一个新进程对打印进行管理。四是由已存在的进程产生另一个进程。

表 2.1　进程产生的原因

原　　因	说　　明
由批处理任务	通常在磁带或磁盘上，有一个供操作系统使用的批处理作业流。当操作系统准备好接受新任务后，它将会读作业控制命令的下一个序列
交互式环境中的新用户登录所致	在终端上的用户登录系统
OS产生	OS可以代用户程序产生进程以实现某种功能，使用户不必等待
已存在的进程生成	用户程序可以产生多个进程

传统上，进程都由操作系统产生并且对用户或应用程序是透明的，这在当今的操作系统中仍很普遍。然而，允许一个进程产生另一个进程是很有用处的。例如，一个应用程序进程可以产生一个进程用来接收应用程序产生的数据，并将那些数据组织起来放入一个表中，以便后面分析利用。新进程同原来的应用程序并行运行，当有新数据产生时就被激活。这一种安排对应用程序的结构化非常有益。

当一个进程生成另一个进程时，生成进程称为父进程(Parent Process)，而被生成进程称为子进程(Child Process)。通常情况下，这些相关进程需要相互通信并且互相协作。

2. 进程终止

导致进程终止的事件大致有14种，如表2.2所示。

表 2.2　进程终止的原因

原　　因	说　　明
正常结束	进程执行一个OS调用以表示其运行完毕
超时限制	进程运行超过了指定的最长时间
内存不足	进程需要的内存量大于系统可提供量
超界	进程试图对不允许接近的区域进行操作
保护错误	进程试图使用不允许使用的资源或文件，或者使用方式不对。例如，试图对一个只读文件进行写操作
算术错误	进程试图进行不允许的计算，例如，除以零或存储一个比硬件所能容纳最大数还要大的数等
超越时限	进程等待时间超过了某事件发生的指定时间
I/O失败	输入/输出过程中产生错误
非法指令	进程试图执行一个不存在的指令(通常是分支程序进入数据区域并企图执行这些数据)

续表

原　　因	说　　明

特权指令	进程试图执行一个保留给OS使用的指令
错误使用数据	数据类型出错或者数据未初始化
操作员或OS干预	由于某些原因，操作员或OS终止了进程(例如，发生死锁)
父进程终止	当一个父进程终止后操作系统会自动终止所有的子孙进程
父进程需要	父进程拥有终止所有子孙进程的权限

在任何计算机系统中，进程必须有一种方法以表明其运行结束。一个批处理任务可以包含一条“Halt”指令或执行OS提供的终止调用。在前一种情况下，“Halt”指令会产生一个中断告诉OS进程已完成。在交互程序中，应有一个用户动作显示进程结束。例如，在一个分时系统中，当用户退出登录或关闭终端时，该用户的进程就会被终止。在个人计算机或工作站上，用户可能退出应用程序。所有这些动作将导致OS产生终止相应进程的请求。

另外，一些错误会导致终止进程。一个进程也可被父进程终止或随着父进程终止而终止。

2.1.2　进程状态模型

如果所有进程都已准备好执行，则图2.4(b)建议的排队原则是有效的，即队列按先进先出(First-in First-out，简称FIFO)排列，处理器依次运行进程(只要不阻塞队列，每个进程都有一定的时间运行，然后回到队列中)。但即使是最简单的例子，这种实现方式也是不能满足要求的。一些非运行状态的进程预备好执行，而其他的一些却由于等待I/O完成而被阻塞。分派程序不能仅仅只在队列中选择等待时间最久的进程，应该扫描整个队列，寻找未阻塞的等待时间最长的进程。

一种自然的方法是将非运行状态又分为两种状态：就绪(Ready)和阻塞(Blocked)，如图2.5所示。为了便于说明，我们又增加了两种有用的状态。这样就有了以下5种状态。

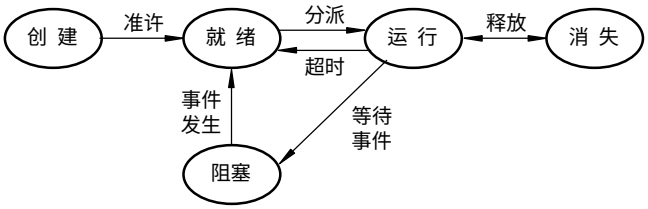


图 2.5　进程状态模型

① 运行(Running)：进程当前正处于运行的状态。假设系统只有一个处理器，因此，在同一时刻只有一个进程可以处于该状态。

② 就绪(Ready)：进程已准备好，一旦有处理器就可运行。

③ 阻塞(Blocked)：进程在等待某些事件发生后才能运行，例如，等待I/O操作完成。

④ 创建(New)：进程刚产生，还没有被OS提交到可运行进程队列中。

⑤ 消失(Exit)：进程被OS从可运行进程队列中释放出来。

创建和消失状态对进程管理非常有用。创建状态对应刚定义的进程。OS在定义一个新进程时，首先做一些相关的杂务工作，然后建立所有管理进程所需的表格。这时，进程处于创建阶段。这意味着，操作系统执行了创建进程的相关操作但还没有允许自己执行该进程。例如，OS可能因为系统性能或内存的限制而限制进程个数。

进程从系统中退出时，会先终止然后进入消失状态。这时，进程没有资格运行，然而，在与该任务有关的表格或其他信息抽取完后，OS也不必再保存与进程有关的信息。表2.3提供了更详细的状态转换信息。

表 2.3　进程状态转换

原状态	转换后的状态				
	创建	运行	就绪	阻塞	消失
	OS根据作业控制请求；分时系统用户登录；进程产生子进程而创建进程	×	×	×	×
创建	×	×	OS准备运行新的进程	×	×
运行	×	×	超时；OS服务请求；OS响应具有更高优先级的进程；进程释放控制	OS服务请求；资源请求；事件请求	进程完成，进程夭折
就绪	×	被分派程序选择为下一个即将执行的进程	×	×	被父进程终止
阻塞	×	×	事件发生	×	被父进程终止

现在依次讨论每一种状态转换的可能：

① 无→创建。为执行一个程序而产生一个新进程，表2.1中列举的任何一个原因都会导致这一事件发生。

② 创建→就绪。OS在可以执行其他进程时，将进程从创建状态转换为就绪状态。大多数系统对存在的进程数或提供给已有进程的虚拟内存的空间设置一些限制，以免进程过多而降低系统性能。

③ 就绪→运行。当要选择一个新进程运行时，OS从处于就绪状态的进程中选

择一个进程运行。关于选择策略将在第6章中讨论。

④ 运行→消失。如果当前运行的进程完成或夭折，则该进程就会被OS终止。

⑤ 运行→就绪。由于某种原因导致这种状态转换，例如，在一个有多种优先级进程的OS中，进程A已在运行，而进程B虽拥有较A高的优先级却处于阻塞状态，如果B等待的事件已发生，并转换为就绪状态，则OS会中断A的执行，转而执行B，这时，A就回到就绪状态。

⑥ 运行→阻塞。当一个进程提出某些请求而必须等待时，该进程就进入阻塞状态。对OS的请求通常是以系统调用的形式，也就是运行OS的一段代码，而OS有可能不能立即执行；也有可能是对资源(如文件)、虚拟内存等提出要求而不能立即得到。或者进程开始某个操作(如I/O)，而只有当该操作完成后，进程才能继续。如果进程间有通信，一个进程有可能因为必须等待其他进程的输入或传来的信息而被阻塞。

⑦ 阻塞→就绪。当某阻塞进程所等待事件发生后，该进程变为就绪状态。

⑧ 就绪→消失。这一转换未在图2.5中标出。在某些系统中，父进程能随时终止子进程。并且，若父进程终止，则其所有子进程将一起终止。

⑨ 阻塞→消失。同就绪→消失。

表2.4列出了每个进程在5种状态间的转换。图2.6(a)提出了一种排队规则。现在有两个队列：一个就绪队列和一个阻塞队列。所有进程在被提交到系统后，都处于就绪队列中。每次，OS从就绪队列中挑一个进程运行。如果没有任何优先级调度，则这个队列可能是一个先进先出(FIFO)队列。当一个运行进程停止执行时，它可能被终止或放入就绪或阻塞队列。当有一个事件发生时，所有阻塞队列中等待该事件的进程被移入就绪队列。

表 2.4　对图 2.3 中进程状态的跟踪

时间段	进程A	进程B	进程C
1～6	运行	就绪	就绪
7～12	就绪	就绪	就绪
13～18	就绪	运行	就绪
19～24	就绪	阻塞	就绪
25～28	就绪	阻塞	运行
29～34	就绪	阻塞	就绪
35～40	运行	阻塞	就绪
41～46	就绪	阻塞	就绪
47～52	就绪	阻塞	运行

这后一种管理方法意味着，每一个事件发生时，OS都必须扫描整个阻塞队列，搜寻等待该事件发生的进程。在大的操作系统中，可能有成千上万个进程在那个队列中。因此，如果每个事件都对应有一个等待队列，则效率会高一些。这样，当某

事件发生时，与其相对应的队列就被移入就绪队列，如图2.6(b)所示。

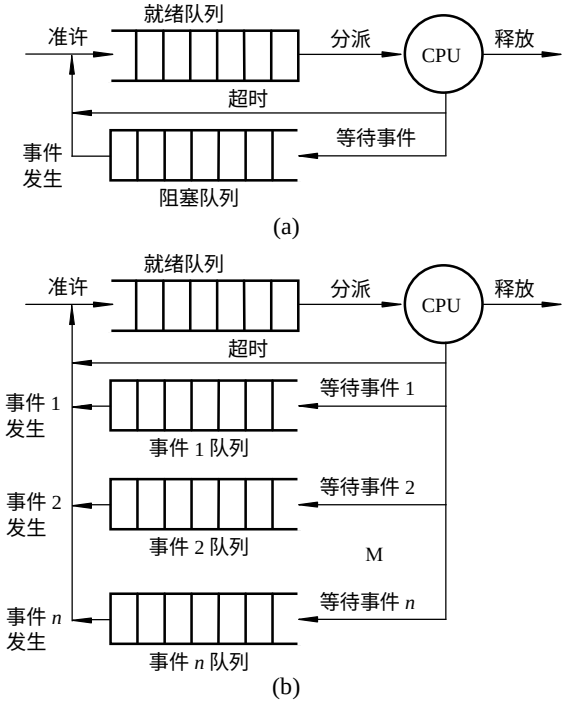


图 2.6　图 2.5 中的队列模型

(a) 单一阻塞队列　(b) 多阻塞队列

最后，需指出的是：如果分派程序通过优先级调度策略调度，那么有多个就绪队列将更方便。每个队列有一个优先级，这样OS可以很容易地确定拥有最高优先级且等待时间最长的就绪进程。

2.1.3　进程挂起

1. 进程挂起的状态

上面讨论的3个基本状态(就绪、运行、阻塞)提供了一种系统规范进程行为的方式。很多操作系统都只有这3种状态。然而，如果系统中没有虚拟内存，每个要执行的进程全部装到内存中的话，我们也有理由再增加一些额外的状态。提出这些复杂机制的理由是I/O操作比计算要慢得多，虽然在内存中有多个进程，当一个进程等待时，处理器又转而执行另一个进程，但处理速度比I/O还是要快得多，在内存中所有进程都等待I/O的现象是很普遍的。这样，即使同时运行多个程序，处理器在大多数时间里仍处于空闲状态。图2.6所示的队列模型并没有完全解决这个问题。

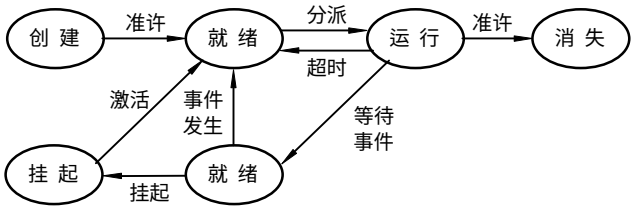
解决方法之一是将主存进行扩充，以容纳更多的进程。这种方法仍有两点不足：

第一，这必将导致费用的增加；第二，现在程序内存占用量的增加足以抵消内存价格的下降，更大的内存可能只是导致更大的进程而不是更多的进程。

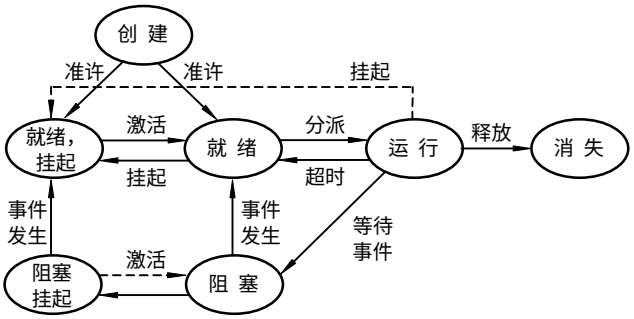
另一种解决方法是交换(Swapping)，即将内存中一部分进程转移到磁盘中。当内存中没有进程处于就绪状态时，OS将其中一个阻塞进程转移到磁盘上的挂起队列中(挂起队列中的进程是那些暂时移出内存或称为挂起的进程)。接下来，OS从挂起队列中移入另一个进程或响应一个新进程的请求供处理器执行。

然而，交换实际上是一个I/O操作，所以，也许会使事情变得更糟。但是磁盘I/O通常是系统I/O操作中最快的(同磁带或打印机相比)，所以交换通常会提高系统性能。

有了交换，就必须在进程行为模式中增加一个新状态：挂起状态(Suspend State)，如图2.7(a)。当内存中的所有进程都阻塞时，OS可以将其中一个置于挂起状态并交换到磁盘中，然后调入另一进程。这时OS有两种选择，它既可以接受一个新进程，又可以调入一个原先挂起的进程。事实上从减轻整个系统负担的角度出发，调入一个挂起的进程比接受一个新进程要好。



(a)



(b)

图 2.7　有挂起状态的进程转换图

(a) 带有一个挂起状态　(b) 带有两个挂起状态

因此，必须对早先的设计进行改进。这里每个进程都有两个独立的概念：是否在等待某事件发生(是否阻塞)以及是否被交换出了内存(是否挂起)。这样，两两组合，我们就得到以下4种状态：

① 就绪。进程在内存中并可以被执行。

② 阻塞。进程在内存中，但必须等待某事件的发生。

③ 阻塞挂起。进程在辅存中并且等待某事件的发生。

④ 就绪挂起。进程在辅存中但只要被调入主存就能执行。

需要指出的是，目前的讨论都假设没有使用虚拟内存，并且每个进程要么全部在内存中，要么全部不在。而在虚拟内存调度中，可以执行一个只部分在内存中的进程。虚拟内存的使用会降低交换的需要。

现在分析图2.7(b)和表2.5中所示的新增的状态转换。图2.7(b)中所示的虚线是有可能但不一定必要的转换。新转换中重要的有以下几种：

表 2.5　有挂起状态的进程状态转换

原状态	转换后状态						
	创建	运行	就绪	阻塞	就绪挂起	阻塞挂起	消失
	OS 根据作业控制请求；分时系统用户登录；进程产生子系统而创建进程。	×	×	×	×	×	×
创建	×	×	OS 准备运行新进程	×	OS 准备运行新进程	×	×
运行	×	×	超时；OS服务请求；OS优先响应具有更高优先级的进程；进程释放控制	OS 服务请求；资源请求；事件请求	用户请求进程挂起；OS请求进程挂起	×	进程完成；进程夭折
就绪	×	分派程序选择作为下一个即将执行的进程	×	×	OS 将进程交换出内存以留出空间	×	进程被父进程终止
阻塞	×	×	事件发生	×	×	OS 将进程交换出内存以留出空间；OS或其他进程要求进程挂起	进程被父进程终止
就绪挂起	×	×	OS 将进程交换进内存	×	×	×	进程被父进程终止
阻塞挂起	×	×	×	OS 将进程交换进内存	事件发生	×	进程被父进程终止

① 阻塞→阻塞挂起。如果没有就绪进程，那么至少应有一个阻塞进程被交换到辅存，为非阻塞进程留出空间。只要OS确定正在运行或就绪进程需要更多内存，这种转换即使在有就绪进程时也可以发生。

② 阻塞挂起→就绪挂起。当一个处于阻塞挂起状态的进程所等待事件发生后，它就进入就绪挂起状态。

③ 就绪挂起→就绪。当内存中没有就绪进程时，OS将调入一个就绪挂起的进程继续执行。处于就绪挂起状态的进程也许比所有处于就绪状态的进程的优先级高。OS设计者也许会觉得先执行优先级高的进程比直接交换更重要。

④ 就绪→就绪挂起。通常，OS更愿意挂起阻塞进程而不是就绪进程。然而，如果挂起就绪进程是惟一有效留出内存空间的方法，也只能那样做。另外，如果OS确定高优先级的阻塞进程很快变为就绪时，它可能会选择挂起低优先级的就绪进程。

还有几种转换值得考虑：

⑤ 创建→就绪和就绪挂起。当一个进程生成后，它既可以加入到就绪队列，又可以加入到就绪挂起队列。不管是进入哪种状态，OS都必须建立一些表格以管理该进程并分配地址空间。在这一阶段，如果内存没有足够的空间，就必须使用创建→就绪挂起转换。

⑥ 阻塞挂起→阻塞。这种转换看起来没什么作用，将一个未就绪进程交换进内存有什么意义呢？一个可能的情况是：一个进程终止了，空出一些内存空间，阻塞挂起队列中有一个进程的优先级比所有就绪挂起队列中的进程高，而OS认为所引起的阻塞事件会很快发生。在这种情况下，OS会将该阻塞进程调入。

⑦ 运行→就绪挂起。通常，进程在运行时间到后会移入就绪状态。然而，如果在阻塞挂起队列中某进程拥有较高优先级，且阻塞事件发生而被OS优先选择，则OS会直接将运行进程移入就绪等待队列以空出空间。

2. 挂起进程的其他用途

到目前为止，我们都将挂起进程同不在内存中的进程等同看待。实际上，若一个进程不在内存中，则不论它是否在等待事件发生，都是不能立即执行的。

现通过以下性质的描述来重新定义挂起进程的概念：

① 挂起进程不能被立即执行。

② 挂起进程可以等待某事件发生，也可以不等待。这样，阻塞情况与挂起情况就是相互独立的，一个引起阻塞的事件发生并不能导致挂起程序可以执行。

③ 进程若被某代理者(Agent)置于挂起状态以阻止其执行，该代理者可以是进程自己、父进程或操作系统。

④ 进程只能被代理者的命令移出挂起状态。

表2.6列出了挂起进程的原因。其中之一就是已经讨论过的交换进程为就绪进程留出空间及缓解虚拟内存系统中对内存需要的压力。操作系统还可能为了其他目的而将进程挂起。如果OS发现或怀疑系统出了问题，就会挂起进程。例如，发生死锁，这将在第4章讨论。如果发现通信线路出了毛病，则操作员会要求OS将使用该线路的进程挂起直到修好为止。

表 2.6　进程挂起的原因

原　因	说　明
交换	OS需要释放足够的内存空间以调入就绪进程执行
其他OS原因	OS可能会挂起一个后台进程或怀疑引起问题的进程
交互用户要求	用户可能为了调试或与资源连接而要求挂起进程
间断	一个进程可能是周期性地执行的，那么它在等待下一次执行时如被挂起
父进程请求	父进程有时希望挂起某个后代进程以检查或修正挂起进程，或协调多个后代进程的运行

另一个原因与交互式用户的要求有关。当用户在调试程序时，会挂起进程，检查和改变程序、数据，或者用户需要一些后台程序来跟踪已有程序并进行某些统计工作。用户当然希望这些后台程序能够自然地启动和终止。

进程间断会导致交换决定的作出。例如，某进程需要定时激活但绝大多数时间内又处于空闲状态，那么应在两次使用之间将其交换出内存。

最后，父进程可能希望将一个后代进程挂起。例如，进程A生成进程B去读一个文件，而进程B在读文件时遇到了错误，并告诉进程A，这时进程A会挂起进程B并查明原因。

2.2　进程描述

在计算机系统中，操作系统调度进程、分配资源、响应用户程序服务的请求。可以认为操作系统管理整个系统资源的使用和控制整个系统的事件。

图2.8描述了这个概念。在一个多道程序运行环境内，有多个进程(P_1，…，P_n)生成并存放在虚存中。在执行过程中，每个进程必须拥有一定的资源，包括处理器、I/O设备、内存等。其中，P_1正在运行，P_1至少有一部分在内存中，并控制了两个I/O设备；P_2也在内存中，但由于等待一个I/O设备而阻塞；P_n被交换出内存处于挂起状态。

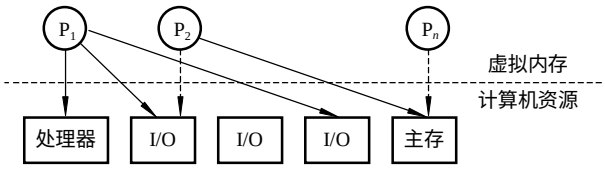


图 2.8　进程和资源

现在的问题是，操作系统靠什么来控制进程和为进程管理资源。

2.2.1　操作系统控制结构

为了管理进程和资源，操作系统必须掌握每一个进程和资源的当前状态信息。一般的方法是直接提供信息，操作系统为每个被管理者建立并维护一个信息表，如图2.9所示。操作系统保存了不同类别的表：内存表、I/O表、文件表和进程表。尽管细节上可能会有所不同，但几乎所有操作系统都会用这4类表来保存信息。

① 内存表(Memory Tables)用来跟踪主(实)存和辅(虚拟)存。内存表必须包含以下信息：

• 主存分配给进程的情况。

• 辅存分配给进程的情况。

• 主存或虚拟内存中所有的段保护性质，如哪些进程可以对特定存储区域进行操作。

• 所有用来管理虚拟内存的信息。

存储管理的详细内容将在第5章讨论。

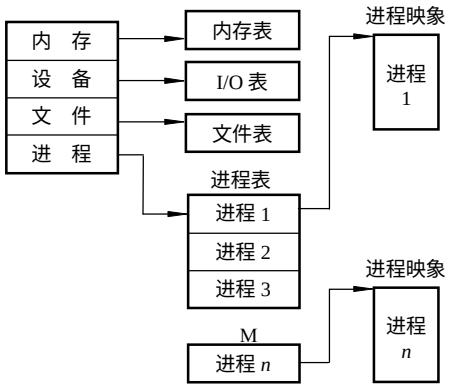


图 2.9　操作系统控制表的一般结构

② I/O表用来管理I/O设备和通道。在任一时刻，I/O设备都有可能空闲或分配给了某进程。如果I/O操作在进行中，则操作系统需要知道I/O操作的状态以及I/O传输在主存中的源或目的地址。

③ 文件表提供当前存在文件中的信息：文件在辅存中的位置、当前状态及其他性质。这些信息中很多由文件管理系统维护和使用，操作系统并不参与。在有些系统中，操作系统亲自参与文件管理。

④ 进程表用来管理进程。

在继续讨论前，有两点需要指明：第一，图2.9中有4个明显的表集，因为对内存、I/O和文件的管理是为了执行进程，因此，在进程表与这些资源之间应该有某些直接或间接的索引；第二，操作系统又是如何知道到底要创建哪些表格呢？显然，操作系统必须清楚基本的环境。因此，在操作系统初始化时，它必须接受某些定义系统基本环境的配置数据。这由操作系统以外的程序在人工辅助下完成。

2.2.2　进程控制结构

为了控制进程，操作系统必须知道进程存储在哪儿，以及进程的一些属性。

首先，介绍进程的具体物理组成。一个进程最少应包括一个要执行的程序或程序集。与程序有关的是一个数据集，包括局部、全局变量及定义的常量。这样，进程最少必须由足够的内存组成以容纳程序和数据。另外，程序的执行通常都需要栈

以备调用过程时使用或在过程间传递数据。最后，还有一些被操作系统用来控制进程的属性。通常这些属性的集合称为进程映像(Process Image)(见表2.7)。

表 2.7　进程映像的基本元素

用户数据	用户空间中可修改部分。可以包括程序、数据、用户自定义栈、可修改程序等
用户程序	要执行的程序
系统栈	每个进程都有一个或多个后进先出(LIFO)栈，用来存储参数、过程和系统调用的调用地址
进程控制块	操作系统用来控制进程的信息(见表2.8)

进程映像的存储依赖于所使用的存储器管理调度方式。最简单的情况是，将进程映像作为一个连续块存储，这个块保存在辅存中。为了管理进程，映像中的一小部分包括操作系统当前要用到的信息必须保存在内存中。如果要执行进程，则整个进程映像必须装入内存。这样，操作系统必须知道进程在磁盘或内存中的存储地址。

在大多数现代操作系统使用的存储器管理调度方式中，进程映像是由一组不必连续存储的块组成。根据调度类型不同，这些块可能是变长的(称为段)或固定长的(称为页)或混合的。不论哪种情况，都允许操作系统仅调入进程的一部分。在任一时刻，进程映像只有一部分在内存中，其余的在辅存中。因此，进程表必须指明每个进程映像的每一个段和/或页的地址。

在图2.9中，有一个基本进程表，表中每个进程都有一个条目与之对应，每个条目又至少包括一个指向进程映像的指针。如果进程映像包含多个块，则这个信息将包含在基本进程表中或者通过交叉索引包含在内存表中。

2.2.3　进程属性

对一个精巧的多道程序运行系统，进程管理需要许多信息。系统不同组织信息的方式亦不同，这将在2.5节介绍几个这方面的例子。现在，抛开具体细节来讨论对操作系统可能有用的信息类型。

表2.8列出了操作系统管理进程所需的信息类型，这些信息都收集在进程控制块(PCB)中。

在几乎所有操作系统中，每个进程都用一个惟一的数字编号作为进程标识，这个编号可能仅仅是一个基本进程表中的一个索引。如果没有数字编号，就必须有一种映射机制，使操作系统根据进程标识去定位合适的表格。很多操作系统控制的其他表格都是通过进程标识来对进程表进行交叉引用的。

表 2.8　进程控制块的典型元素

进程标识	
• 本进程的标识符 • 创建本进程的进程(父进程)的标识符 • 用户标识符	
处理器状态信息	
用户可视寄存器	这是通过处理器执行的机器语言可访问的寄存器。通常有8到32个，在某些RISC系统中超过100个
控制和状态寄存器	用来控制处理器运行的各种寄存器，包括： • 程序计数器(Program Counter)　保存下一条指令的地址 • 状态码(Condition Code)　最近的算术或逻辑操作结果(如符号，进位，溢出等) • 状态信息(Status Information)　包括中断开/关符号、执行模式等
栈指针	每个进程都有一个后进先出(LIFO)的系统栈
进程控制信息	
调度和状态信息	这是OS用来实现调度功能的信息，通常有以下几条： • 进程状态　定义进程在调度中的状态(如运行、就绪、等待等) • 优先权　可以用一个或多个字段来描述调度优先权。在一些系统中需要多个值(如缺省，当前，最高允许值等) • 调度相关信息　这同具体使用的调度算法有关 • 事件　进程重新开始执行前必须等待的事件
数据结构	一个进程可能会与其他进程连在一起形成队列或其他结构。例如，所有处于等待状态的同一优先级进程可被连成一个队列。进程控制块(PCB)中可能会包括指向其他进程的指针以支持多进程组成的结构
进程间通信	与两独立进程通信相关的有各种标志、信号和信息。这些信息可以包含在PCB中
进程特权	进程被授予一些特权，规定可被使用的存储区和可以使用的指令类型。另外，还有可以调用的系统工具和服务
存储器管理	包括指向本进程描述虚拟内存空间的段/页的指针
资源的拥有和使用	指示进程拥有的资源。还包括使用处理器和拥有资源的历史，这个信息对调度程序有用

进程状态信息包括处理器中的一些寄存器的内容。当进程执行时，这些信息在寄存器中；在进程中断后，所有寄存器的信息都保存起来以备重新执行时恢复。各种控制和状态寄存器被用来控制处理器的运行。其中大多数对用户都是不可见的。在所有处理器中都有程序计数器或指令寄存器，还有一个或多个寄存器称作程序状态字(Program Status Word，简称PSW)，用以保存状态码及其他状态信息。

进程控制块中第三类信息叫进程控制信息，它是操作系统用来控制和协调各进程的信息。

图2.10给出了一种进程映像结构。其中，进程映像是连续存储的，但在具体实现中，可以根据存储器管理方式和OS控制结构方式来存储。

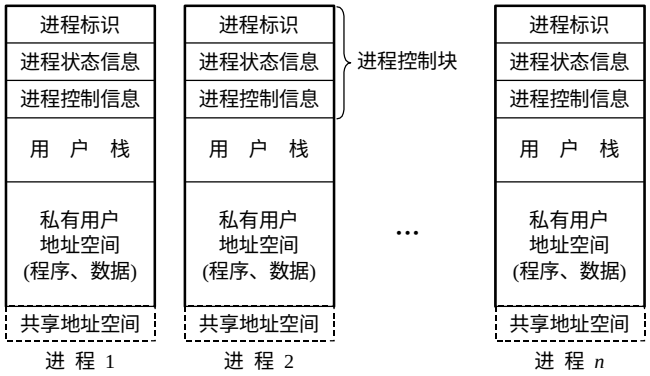


图 2.10　虚拟内存中的用户进程

由于进程控制块中有结构信息，包括可以用其他进程控制块连接的指针。这样，上节介绍的队列就可以实现了。图2.6(a)就可以像图2.11所示的那样实现了。

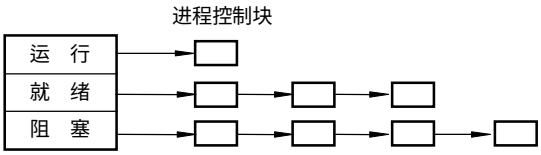


图 2.11　进程队列结构

2.3　进程控制

2.3.1　执行模式

我们必须清楚处理器同操作系统有关的执行模式，以及处理器同用户程序有关的执行模式之间的区别。大多数处理器都至少支持两种执行模式。某些特殊指令和特殊存储区只能被更高特权级别的模式使用。

较低特权模式称为用户模式(User Mode)，用户程序只能在这一级别执行。较高特权模式指系统模式(System Mode)、控制模式(Control Mode)或内核模式(Kernel Mode)。内核是操作系统中最核心功能的集合。表2.9列出了操作系统内核的主要功能。

使用两种模式的原因是：有必要对操作系统和操作系统中的关键表格进行保护，以免被用户程序破坏。在内核模式下，软件对处理器和所有指令、寄存器、存储器拥有全部控制权。这种级别的控制对用户程序是不合适，也是不必要的。

那么，处理器如何知道其正处于何种模式，并且模式又是如何转换的呢？通常，

在PSW中有一位用来显示当前处理器处于的模式，这一位会因响应某些事件而改变。例如，当用户调用一个操作系统服务时，只需执行一条指令，当前模式就被变为内核模式。

表 2.9　操作系统内核的主要功能

进程管理	• 进程创建和终止 • 进程调度和分派 • 进程转换 • 进程同步和对进程间通信的支持 • 对进程控制块的管理	I/O管理	• 缓冲区管理 • 给进程分配I/O通道和设备
存储器管理	• 为进程分配地址空间 • 交换 • 分页和分段管理	支持功能	• 中断处理 • 计账 • 监控

2.3.2　进程创建

一旦操作系统因表2.1中所列举的任一原因决定创建一个进程时，它将进行如下操作：

① 给新进程一个编号。这时，在基本进程表中加入一个新的条目。

② 为进程分配空间。这包括进程映像的所有元素。这样，OS必须知道私有用户地址空间(程序和数据)和用户栈所需空间太小。这个值可以根据创建的进程类型的缺省大小或作业创建时用户的要求来确定。如果一个进程由另一个进程创建，则父进程可将所需值传给操作系统。如果已有的地址空间需和新进程共享，还必须建立合适的链接。最后，还必须为PCB分配空间。

③ 初始化PCB。进程标识部分包括含该进程以及相关进程的ID。除了程序计数器(设置为程序入口地址)和系统栈指针(设置为进程栈的边界)外，进程状态信息部分大都初始化成0。而进程控制信息部分则应按照对该进程要求的缺省值和属性设置初始值。如进程状态通常设为就绪或就绪挂起。除非有明确的要求，优先级都设缺省值为最低级。除了明确要求或从父进程继承外，进程开始不应拥有资源(I/O设备，文件)。

④ 设置合适的链接。如果操作系统用链表管理每个调度队列，那么，新进程就必须链入就绪或就绪挂起队列。

⑤ 生成其他一些数据结构。

2.3.3　进程切换

在某时刻，一个正在运行的进程被中断，操作系统就将另一个进程置为运行状

态，并对其进行控制。那么，什么事件导致进程切换呢？内容切换同进程切换之间有什么区别？操作系统为了实现进程切换必须对各种数据结构进行怎样的处理？这些问题应很好掌握。

当操作系统掌握控制权时，切换随时会发生。表2.10所示的是可能将控制权交给操作系统的事件。首先，我们来看一下系统中断。在很多系统中，有两种系统中断。其中之一被简单称为中断，另一个称为陷阱(Trap)。前者是由与当前运行进程无关的外部事件引起的，如I/O操作完成；而后者则同当前运行进程产生的错误和意外有关，如对文件的非法操作。

表 2.10　中断进程执行的事件

事　　件	起　　因	用　　途
中断	来自当前指令执行的外部	对外部事件响应
陷阱	同当前指令执行有关	处理错误或意外情况
监管调用	明确的请求	调用操作系统的功能

对一般的中断，控制权被转移给中断处理程序。对“陷阱”，操作系统要确定错误是否“致命”。如果这样，则当前执行的进程被移入消失状态，并发生进程切换。如果不是这样，则OS会根据错误的性质和自身的设计进行处理。它要能尝试进行恢复或仅仅将错误通知用户。OS既有可能进行进程切换，也有可能继续执行当前进程。

最后，来自执行程序的监管调用(Supervisor Call)会激活操作系统。例如，一个用户进程中一条指令请求一个I/O操作，这个调用会导致处理器转向执行一段操作系统代码。通常，对系统的调用会使进程转入阻塞状态。

2.3.4　上下文切换

在中断周期中，处理器要检查是否有中断发生，如果有中断等待处理，则处理器应进行如下工作：

① 将当前运行程序的上下文保存；

② 将程序计数器设为中断处理程序的起始地址。

接着处理器进入取指令周期，开始为中断服务。

所有可能会被中断处理的执行所改变及重新执行过程所需的信息都必须保存下来。因此，PCB中的处理器状态信息都是要保存的。

PCB中的其他信息会怎样呢？如果这次中断以后必须接着切换其他进程，就必须对PCB作相应处理。然而，在大多数操作系统中，中断的发生并不一定导致进程交换。有可能当中断处理执行完后，当前运行进程接着执行。如果是这样的话，所要做的是当中断发生时保存处理器状态信息，而当控制重新回到程序时，恢复它们。

很显然，上下文切换同进程切换在概念上是不同的。上下文切换发生时，可能并不改变当前处于运行态的进程状态。

当前运行进程的状态要改变时，操作系统将进行以下步骤完成进程切换：

① 保存处理器内容。

② 对当前运行进程的PCB进行更新，包括改变进程状态和其他相关信息。

③ 将这个进程的PCB移入适当的队列(就绪、因事件的堵塞、就绪挂起等)。

④ 挑选其他进程执行。

⑤ 对挑选进程的PCB进行更新，包括将其状态改为运行。

⑥ 对存储器管理数据结构进行更新。

⑦ 将被选中进程上次移出时的处理器状态进行恢复。

这样，包含状态改变的进程切换，比上下文切换需要进行更多的处理。

2.3.5　操作系统的运行

操作系统的运行方式同普通软件一样。也就是说，操作系统也是一个由处理器执行的程序。

操作系统经常放弃控制权，它必须依靠处理器来允许其恢复控制。

如果操作系统只是程序的集合并且它同其他程序一样地被处理器执行，那么，操作系统是进程吗？如果是，它是如何进行控制的呢？

操作系统的运行方式如图2.12所示。

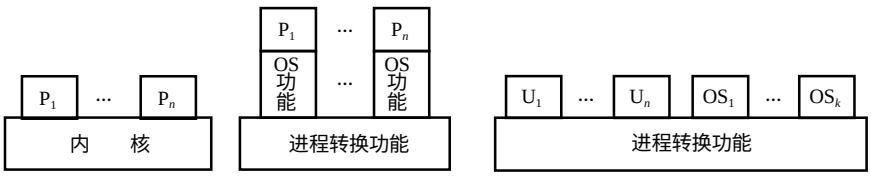


图 2.12　操作系统与用户进程间的关系

(a) 在进程以外运行　(b) 在用户进程中运行　(c) 作为独立进程运行

1. 在进程以外运行

一种非常传统并且在很多旧式操作系统中，经常使用的方法是在进程以外执行操作系统的内核，如图2.12(a)所示。在这种方法中，若当前运行进程发生中断或系统调用，则保存该进程的处理器状态。然后内核得到控制权。操作系统有自己的内存区和系统栈。操作系统进行合适操作后可以恢复中断进程执行或调用其他进程。这里，关键的一点是，进程的概念只适用于用户程序。操作系统代码在特权模式下作为一个独立实体运行。

2. 在用户进程中运行

这种方法经常在小型机或微型机的操作系统上使用。它是将几乎所有的操作系统软件作为用户进程的内容执行。操作系统是各种供用户调用功能的集合，并且在用户进程环境下执行，如图2.12(b)所示。在任一时刻，操作系统都管理*n*个进程映像。每个进程映像不仅包括图2.10中的区域还包括供内核程序使用的程序、数据和栈等区域。

图2.13给出了一种典型的进程映像结构。独立的内核栈用来管理进程处于内核模式下的调用和返回。操作系统代码和数据处于共享地址空间并由所有用户进程共享。

当中断、陷阱或监管调用发生时，处理器转换为内核模式，操作系统得到控制权。为了实现这一点，处理器的上下文被保存起来并发生上下文转换。然而，运行仍发生在当前用户进程的内部。这样，进程转换并没有发生，而只在同一进程中发生上下文转换。

如果当自身任务完成后，操作系统发现当前进程应继续运行，那么在当前进程中上下文转换将恢复中断的程序。这是这种方法的优势之一。一个程序中断后又重新开始执行，这一过程不会带来两个进程转换的麻烦和开销。如果操作系统发现应进行进程转换，控制权就会转到进程转换例程。

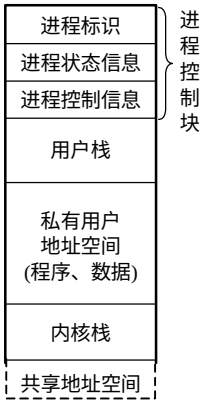


图 2.13　进程映像：操作系统在用户进程中执行

3. 作为独立进程运行

如图2.12(c)所示，这种方式将操作系统作为系统进程的集合来实现。同其他方式一样，内核部分的软件将在内核模式下执行。然而，主要的内核功能都是作为独立进程组织的。有一小部分的进程转换代码在所有进程外执行。

这种方法有多个优点。这里提出一种设计原则：鼓励操作系统组件化，而且尽可能使各组件清晰、简洁。另外，操作系统的一些并不关键的功能作为独立进程实现。将操作系统作为一系列进程实现，在多处理器或多计算机环境中是很有用的。在这种环境中，一些操作系统服务能够被转移到其他处理器中，这样可以提高整个系统的性能。

2.3.6　微核

最近，微核(Microkernel)的概念引起人们的关注。微核是一个操作系统的核心，它为组件扩展提供了基础。微核这个定义是很模糊的。关于微核有许多问题，不同的设计者有不同的答案。这些问题包括：核多小才被称为微核？如何在抽象硬件功能的同时设计设备驱动程序以获得最佳性能？对非核心操作的运行是在内核还是在用户空间？是保留已有子系统代码(如UNIX的一个版本)还是从头开始？等等。

在理论上，这种内核方式被设计成具有高度的弹性和组件化。Windows NT广泛

宣传自己使用了微核技术，并声称其设计不仅组件化而且还支持便携性。其内核周围有许多紧密的子系统，因此，在各种平台上实现Windows NT的功能变得很简单。

微核所应遵循的原理是：只有那些绝对必要的操作系统核心功能才能放在内核中。次重要的服务和应用功能则建立在微核之外。传统上作为操作系统一部分的许多属性，现在都成为外围子系统，这包括文件系统、用户界面和安全设施等。

2.4　线程和SMP

线程(Thread)是近年来操作系统中出现的一个非常重要的技术。当代操作系统如Windows NT/2000，OS/390，Sun Solaris, Linux, Mach 等均采用了线程机制。线程的引入，进一步提高了程序并发执行的程度，从而进一步提高了系统的吞吐量。

2.4.1　线程及其管理

1. 线程的引入

首先回顾一下进程的有关知识。

进程引入的目的是为了程序并发执行，进程有两个基本属性：

① 进程是一个拥有资源的独立单元。一个进程被分配一个虚拟地址空间以容纳进程映像(Process Image)，一个进程的映像主要包括 4 个部分：用户数据、用户程序、系统堆栈和 PCB。有时，进程也被分配主存并控制其他资源，如 I/O 通道、I/O 设备和文件。

② 进程是一个被处理机独立调度和分配的单元。

上述两个属性构成了程序并发执行的基础。为了使进程并发执行，操作系统还需进行一系列操作：创建进程、撤消进程、进程切换。所有这些，操作系统必须为之付出较多的时间开销。正因为如此，在系统中不宜设置过多的进程，进程切换的频率也不能太高。这也限制了并发程度的进一步提高。

如何既能提高程序的并发程度，又能减少操作系统的开销呢？能否将前述的进程的两个基本属性分离开来，分别交由不同的实体来实现呢？为此，操作系统设计者引入了线程，让线程去完成第二基本属性的任务，而由进程只完成第一基本属性的任务。这样，线程成为进程中的一个实体，是被系统独立调度和分配的基本单位，线程自己基本上不拥有系统资源，只拥有在运行中必不可少的一点点资源(例如：程序计数器、一组寄存器、堆栈)，但它可与同一进程内的其他线程共享进程所拥有的全部资源。

引入线程还有一个好处，就是能较好地支持对称多处理器系统(Symmetric Multiprocessor，简称 SMP)。

2. 线程的定义及特征

关于线程有许多不同的定义：线程是进程内的一个执行单元；线程是一个独立的程序计数器；线程是进程内的一个可调度实体；线程是执行的上下文(Context of Execution)，其含义是执行的现场数据和其他调度所需的信息，等等。

因此，可将线程定义为：线程是进程内的一个相对独立的、可独立调度和指派的执行单元。

在有些操作系统中，将线程叫轻型进程(Light-weight Process)；而把传统的进程叫重型进程(Heavy-weight Process)。

根据线程的概念可知，线程具有以下性质：

① 线程是进程内的一个相对独立的可执行单元。

② 线程是操作系统中的基本调度单元，在线程中包含调度所需的信息。

③ 一个进程中至少应有一个线程，当然也可有多个线程，这是因为进程已经不再是被调度的单元。

④ 线程并不拥有资源，而是共享和使用包含它的进程所拥有的所有资源。正因为共享进程资源，所以进程内的多个线程之间的通信需要同步机制。

⑤ 线程在需要时也可创建其他线程。线程有自己的生命期，也有状态变化。

进程和线程的区别与联系：

① 调度。线程是调度和指派的基本单位，而进程是资源拥有的基本单位。在同一进程中，线程的切换不会引起进程切换。在不同的进程中进行线程切换，如一个进程内的线程切换到另一个进程中的线程时，将会引起进程切换。

② 拥有资源。线程不拥有系统资源，但可以访问其隶属进程的系统资源，从而获得系统资源。

③ 并发性。在引入线程的操作系统中，不仅进程之间可以并发执行，而且同一进程内的多线程之间也可并发执行，从而使操作系统具有更好的并发性，大大提高系统的吞吐量。

④ 系统开销。进程切换时的时空开销很大，但线程切换时，只需保存和设置少量寄存器内容，因此开销很小。另外，由于同一进程内的多个线程共享进程的相同地址空间，因此，多线程之间的同步与通信非常容易实现，甚至无须操作系统内核的干预。

3. 线程的状态和管理

不同的操作系统，线程的状态设计不完全相同，但下述 3 个关键的状态是共有的。

① 就绪(Ready)状态：线程已具备执行条件，等待程序分配给一个 CPU 运行。

② 运行(Running)状态：线程正在某一个 CPU 内运行。

③ 阻塞(Blocked)状态：线程正在等待某个事件发生。

值得强调的是：

① 线程中不具有进程中的挂起(Suspend)状态。为什么呢？因为挂起的作用是将资源从内存移到外存，而线程不管理资源，它不应有将整个进程或线程自己从主存中移出的权限，而且可能还有其他线程在使用进程的共享空间。

② 对具有多线程的进程状态，若一个线程被阻塞是否导致整个进程被阻塞呢？很清楚，若如此，则将失去线程的灵活性和优越性。因此，多数操作系统并不阻塞整个进程，该进程中其他线程依然可以参与调度。

③ 由于进程已经不再是调度的基本单位，所以不少系统对进程的状态只划分为两种状态：活动(可运行)和非活动(不可运行)状态，如 Windows NT/2000。

线程使用线程控制块(TCB, Thread Control Block)来描述其数据结构，而 TCB 则利用线程对象加以描述。在多线程操作系统中，PCB 也用进程对象加以描述。

线程的状态转换是通过相关的控制原语来实现的。常用的原语有：创建线程、终止线程、线程阻塞等。

2.4.2　多线程的实现

多线程机制是指 OS 支持在一个进程内执行多个线程的能力。用线程的观点来看，MS-DOS 支持一个用户进程和一个线程。UNIX 支持多个用户进程，但一个进程只有一个线程；一个 Java 运行引擎只有一个进程但有多个线程；Windows NT、Solaris、Mach 支持多进程多线程。

线程虽在许多系统中实现，但实现的方式并不完全相同。有的系统实现的是用户级线程(User-level Threads，简称 ULT)，例如 Infromix 数据库系统；有一些系统实现的是内核级线程(Kernel-level Threads，简称 KLT)，例如 Mach 操作系统；还有一些系统同时实现了这两种类型的线程，例如 Sun 的 Solaris 操作系统。

1. 用户级线程

用户级线程 ULT 由用户应用程序建立，由用户应用程序负责对这些线程进行调度和管理，操作系统内核并不知道有用户级线程的存在。因而这种线程与内核无关。这就是通常所说的“纯 ULT 方法”。MS-DOS 和 UNIX 属于此类。

这种纯 ULT 方法的优点是：

① 应用程序中的线程开关(即转换)的时空开销比内核级线程的开销要小得多。

② 线程的调度算法与操作系统的调度算法无关。

③ 用户级线程方法可适用于任何操作系统，因为它与内核无关。

缺点是：

① 在一个典型操作系统中，有许多系统请求正被阻塞着。因此，当线程执行一个系统请求时，不仅仅本线程阻塞，而且该进程中的所有线程也被阻塞。

② 在纯 ULT 策略下，多线程应用没办法利用多处理器的优点，因为每次只有一个进程的一个线程在一个 CPU 上运行。

2. 内核级线程

内核级线程 KLT 中的所有线程的创建、调度和管理全部由操作系统内核负责。一个应用进程可按多线程方式编制程序，当它被提交给多线程操作系统运行时，内核为它创建一个进程和一个线程，线程在运行中还会创建新的线程。这就是通常所说的“纯 KLT 方法”。Windows NT 属于此类。

这种内核级线程的优点是：

① 内核可调度一个进程中的多个线程使其同时在多个处理器上并行运行，从而提高程序的执行速度和效率。

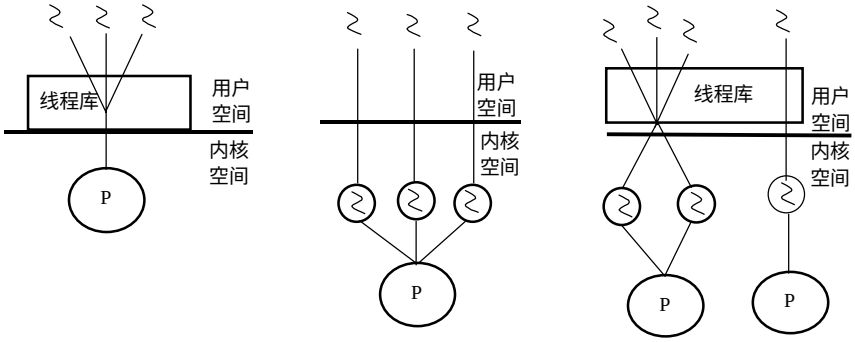
② 当进程中的一个线程被阻塞时，进程中的其他线程仍可运行。

③ 内核过程本身也都可以以线程方式实现。

缺点是：同一进程中的线程切换要有两次模式转换(用户态→内核态→用户态)，因为线程调度程序运行在内核态，而应用程序运行在用户态。

3. KLT 和 ULT 结合

由于纯 KLT 和纯 ULT 各有自己的优点和缺点，因此，如果将两种方法结合起来，则可得到两者的全部优点，将两种方法结合起来的系统称之为多线程的操作系统。内核支持多线程的建立、调度与管理。同时，系统中又提供使用线程库(一个线程应用程序的开发和运行环境)的便利，允许用户应用程序建立、调度和管理用户级的线程。图 2.14 示出了 3 种多线程模型。



(a)　　(b)　　(c)

图 2.14　用户级线程和内核级线程

(a) 纯用户级　(b) 纯内核级　(c) 结合方式

注：≳ 表示用户级线程；Ⓚ表示内核级线程；Ⓟ表示进程。

2.4.3　进程与线程的关系

传统的资源分配和调度单元的关系是一对一的。最近人们对同一进程中多个线程的系统，也就是多对一的关系很感兴趣。表2.11列出了线程与进程之间的多种关系。

表 2.11　线程与进程间的关系

线程：进程	描　　述	例　　子
1：1	每个线程的执行就是一个进程	UNIX System V
n：1	每个进程定义一个地址空间并动态拥有资源； 同一进程可产生多个线程并运行	OS/2，MVS，MACH
1：*n*	一个线程可以在多个进程间转移	Ra
n：*n*	包含*n*：1和1：*n*的性质	TRIX

2.4.4　SMP

随着计算机技术的发展和硬件价格的下降，计算机设计者发现了越来越多的并行计算方法。他们通过多种方法和技术，将多个处理器连在一起，让它们共同完成某项任务，从而大大提高系统的性能和可靠性。

按照多处理器之间的通信方式可将多处理器系统划分为两种类型：

第一种为共享存储器多处理系统(Shared Memory Multiprocessor)。其特点是多个处理器共享一个共享存储器，每个处理器都能存取共享存储器中的程序和数据，并且处理器之间借助于共享存储器进行通信。它有两种结构：一种为主/从式结构；一种为对称多处理器结构。

第二种为分布式存储器系统(Distributed Memory)。其特点是每个处理器都有自己的专用主存储器，计算机之间的通信通过专用线路或网络。通常所说的群集(Clusters)即为此类系统。

本节只讨论对称多处理器系统 SMP。

1. SMP 的特点

SMP 的特点是：

① 系统中有多个处理器，且每个处理器的地位是相同的。内核可在任何一个处理器上执行，每个处理器都可自主地调度进程和线程池中的进程和线程。进程或线程可以并行执行。

② 所有处理器共享主存储器。

SMP 的组织结构如图 2.15 所示。其中，每个处理器都拥有自己的私有 Cache。私有 Cache 的使用带来了一些新的、在设计上需要解决的问题。因为私有 Cache 中有共享主存中的一部分映像，如果一个 Cache 中的数据发生了改变，则数据必须反映到其他 Cache 中，此即“Cache 一致性”问题。

2. SMP 系统设计问题

一个 SMP 操作系统被设计成透明地管理多处理器和其他资源，以便让用户感觉就像在一个单处理器上使用系统。多处理器操作系统应该具有多道程序的功能再加上多处理系统的特性。

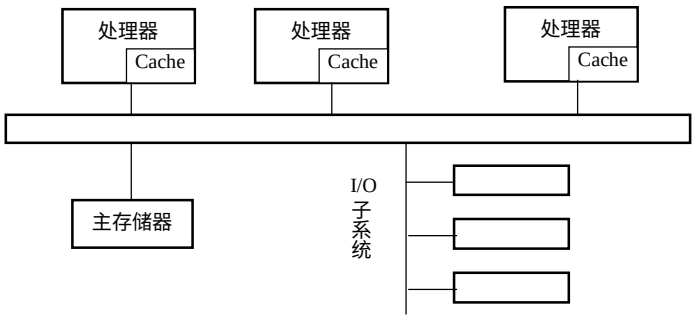


图 2.15　SMP 的组织结构

SMP 操作系统设计的关键是：

① 同时并发执行多进程和多线程。为了允许多个处理器去同时执行同样的内核代码，必须能重入内核程序。由于多个处理器执行内核的相同或不同部分，内核表和管理结构必须正确地管理，以防死锁式非法操作。

② 调度。任一个处理器都可独立调度，因此，必须避免冲突。调度中必须解决的主要问题有：互斥问题、处理器的分配问题、负荷均衡问题、全局或局部的可运行队列的维护问题、调度方法、应用特性与系统性能等问题。

③ 同步。由于有许多活动的进程或线程存取共享的地址空间和 I/O 资源，所以必须提供互斥和同步机制。

④ 存储器管理。主要涉及存储器硬件和存储管理结构的数据一致性和互斥等问题。

⑤ 可靠性和容错能力。操作系统应在碰到某一处理器错误时，依然能正确地运行，这时调度程序和操作系统的其他部分应能识别处理器的失效并重构管理表。

2.5　系统举例

2.5.1　UNIX System V

在UNIX中除了两个基本系统进程外，其余都是由用户程序产生的。

1. 进程状态

UNIX中共有9种进程状态，如表2.12所示。图2.16是UNIX进程状态转换图。这个图同图2.7很相似，两个UNIX睡眠状态对应两个阻塞状态，其不同之处总结如下。

① UNIX中用两个运行状态来区分是运行用户模式还是内核模式。

② 有两个状态被区别开：在内存中就绪和被抢占(Preempted)。它们本质上是同一状态，因此，用虚线将它们连起来。这种区别是为了强调被抢占状态出现的方式。

当一个进程处于内核模式，过一段时间，内核运行完毕并准备将控制转移给用户程序时，内核也许会决定用就绪队列中有较高优先级的进程代替当前进程。在这种情况下，当前进程就被置为被抢占状态。然而，为了调度，那些被抢占进程与在内存中的就绪进程组成同一个队列。

表 2.12　UNIX 进程状态

进程状态	说　明
用户运行态	在用户模式下运行
核心运行态	在内核模式下运行
内存中就绪	只要内核调度就能执行
内存中睡眠	只有等到某事件发生后才能执行，进程处于内存中
就绪且交换	进程处于就绪状态，但是，在其被内核调度去执行之前必须被交换进内存
睡眠且交换	进程在等待某事件发生，并被交换到了辅存上
被抢占状态	进程已在从内核模式向用户模式转化，但是，内核抢先剥夺它的运行，并进行了一项内容转换，调度了另一个进程
创建状态	进程刚被创建，还不能运行
僵死状态	进程已不存在，但它留下一个含有状态码和一些信息的记录，供父进程收集

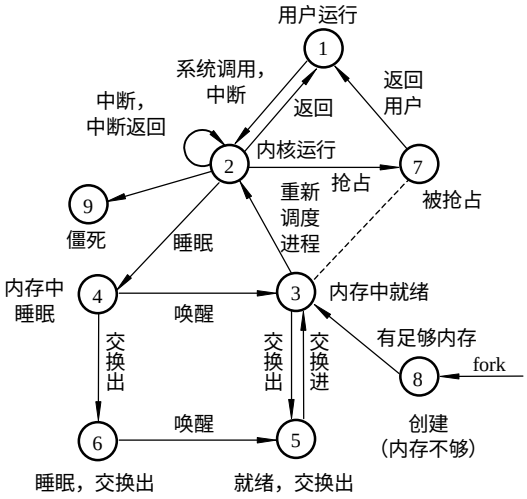


图 2.16　UNIX 进程状态转换图

抢占只能发生在当一个进程正要从内核模式转移到用户模式的时刻。当进程处于内核模式时，是不能被抢占的。这使得UNIX不适合实时处理。

UNIX中的两个惟一的进程是，进程0在系统启动时产生，被预先定义为在启动时加载的数据结构；进程0又生成进程1，称为init进程，所有系统中其他进程都以进程1为祖先。

2. 进程描述

UNIX中的进程是一套非常复杂的数据结构，为操作系统提供了各种管理和调度的信息。表2.13总结了UNIX进程映像的元素并将其组织成3类：用户级上下文、寄存器内容和系统级上下文。

表 2.13　UNIX 进程映像

用户级上下文	
进程内容	程序中可执行的机器指令
进程数据	进程可以访问的数据
用户栈	容纳参数、本地变量，以及指向在用户模式下执行的函数的指针
共享存储空间	与其他进程共享的存储空间，用于进程间通信
寄存器内容	
程序计数器	即将执行的下一条指令的地址
处理器状态寄存器	存放被抢占时的硬件信息；内容和形式依赖于硬件结构
栈指针	指向内核或用户栈的顶部
通用寄存器	与硬件相关
系统级上下文	
进程表项(见表2.14)	定义一个进程的状态；该信息通常被操作系统访问
U(user)区(见表2.15)	存放进程表项的一些扩充信息，它只能被运行在内核态的进程所存取
进程区表	定义虚地址到物理地址的映射
内核栈	进程在核心态执行时使用的栈

表 2.14　UNIX 进程表项

属　性	说　明
进程状态	当前进程的状态
指针	指向U区和进程在内存或外存的位置
进程的大小	操作系统为进程分配存储空间的依据
用户标识符	标识实际用户的ID
进程标识符	标识一个进程的ID
事件描述符	记录使进程进入睡眠状态的事件
优先级	用于进程调度
信号	记录发给进程的信号
计时器	包括进程执行时间，核心资源利用情况和用户设定时间，用来向进程发警报
P-link	指向就绪队列中下一个链接的指针
存储状态	显示进程映像在内存中还是已被交换出内存

表 2.15　UNIX U 区

属　　性	说　　明
进程表项指针	指向同U区相对应的进程表项
用户标识符	真正和有效的用户标识符
信号句柄数组	对系统中定义的信号类型，存放进程在收到该信号后如何反应的信息
控制终端	如果消失，为该进程指示登录进终端
错误域	记录在一次系统调用过程中遇到的错误
返回值	系统调用的结果
I/O参数	给出要传输的数据量，源(或目标)数据的地址，文件的输入、输出偏移量
文件参数	描述进程的文件系统环境(当前路径和当前根目录)
用户文件描述表	记录该进程已打开的所有文件
限制值	对进程的大小及其能“写”的文件大小的限制

用户级上下文可以在编译目标文件中直接生成。用户程序被分为正文区和数据区，正文区是只读的并保存程序指令。当程序运行时，处理器使用用户栈来完成过程调用和返回以及参数传递。共享内存区是一被其他进程所共享的数据区。一个共享内存区只有一个物理上的拷贝，但由于使用了虚拟内存，故使得对于多个共享进程，共享内存区似乎就在其地址空间内。

3. 进程控制

如前所述，UNIX System V遵循了如图2.12(b)中的模式，其中，操作系统大部分是在一用户进程的环境中执行的，即需要两种状态，用户状态和核心状态。

在UNIX中，进程是通过内核系统调用(fork)的方式产生的。当某进程发出一个fork请求时，操作系统执行以下的功能：

① 在进程表中为新进程分配一个位置。

② 分配一个惟一的进程标识符(ID)给这个子进程。

③ 除了共享内存外，拷贝一个父进程的映像。

④ 增加父进程所拥有的文件数量，表现在存在另一进程现在也拥有这些文件。

⑤ 设置子进程的状态为准备运行状态。

⑥ 返回子进程的ID给父进程，将一零值赋给子进程。

所有这些工作都在父进程的核心状态里完成，当内核完成了这些功能后，它可以做以下工作之一，作为调度例程的一部分：

① 停留在父进程，在父进程的fork调用时，控制返回到用户模式。

② 将控制交给子进程，如同父进程一样，子进程在代码的相同处开始执行，也就是说，在fork调用返回时开始执行。

③ 将控制交给另一进程，父子进程均被置于就绪状态。

要将这一进程产生方法形象化可能比较困难，因为父子进程都在执行代码段的

同一部分，所不同的是：当发生fork调用返回时，要对返回的参数进行检测。如果值为零，则为子进程，此时将执行一分支程序到适当的用户程序以便继续执行。若其值非零，则为父进程，且主流程将继续执行下去。

2.5.2　Windows NT

对各种不同的操作系统环境提供支持的需求促进了Windows NT(简称NT)进程设计的产生。不同操作系统所支持的进程在很多方面都有差异，这些差异包括：

① 进程如何命名？

② 进程中是否提供线程？

③ 进程如何表示？

④ 如何保护进程资源?

⑤ 进程间的通信与同步采取何种机制?

⑥ 进程之间如何关联?

于是，NT内核所提供的本质进程结构和服务相对来说就比较简单，并可用于一般的用途。NT进程的重要特点如下：

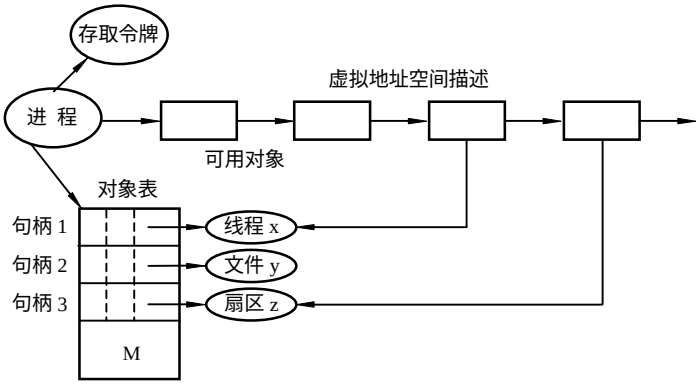
① NT进程以对象形式执行；

② 一个可执行进程可以包含一个或多个线程；

③ 无论是进程还是线程对象都有内置的同步机制；

④ NT内核不维持与它所创建的进程间的任何联系，包括父子进程间的联系。

图2.17展示了一进程与其控制或使用资源之间相互联系的方式。存取令牌控制着进程是否能改变其自身属性。若进程不拥有对其存取令牌打开的句柄，而且该进程又试图打开这样一个句柄，则安全系统将决定这是否是被允许的，并由此决定该进程是否可改变其自身属性。



2.17　NT 进程及其资源

与此进程相关的还有一系列定义当前分配给这一进程的虚拟地址空间的块，进

程不能直接修改这些结构，而必须依靠对其提供存储分配服务的虚拟内存管理器来实现。

最后，进程包含通过句柄与其他对象相连的对象表。此对象所包含的每一个线程都存在一相应的句柄。图2.17显示了一个单一的线程。除此以外，进程还可访问文件对象和共享内存段的段对象。

1. 进程和线程对象

NT的面向对象结构促进了一般进程机制的发展。NT利用了两种与进程有关的对象：进程和线程。进程是一种与拥有资源的用户作业或应用(比如存储和打开文件)通信的实体；线程是一种线性执行并不可被打断的可调度工作单元，这使得处理器可以转到另一线程。

每一个NT进程由一对象表示，其一般结构如图2.18(a)所示。每一进程都是由许多属性定义的，且封装了许多它可能执行的动作或服务。当接受到适当信息时，进程会履行一种服务，传送给提供这种服务的进程对象的消息是激发这种服务的惟一方式。

	进　　程
对象类型	
对象体属性	进程ID 存取令牌 基本优先级 缺省处理器关系 配额限制 执行时间 I/O计数器 VM操作计数器 异常/调试端口 消失状态
服　务	创建进程 打开进程 查询进程信息 设置进程信息 当前进程 终止进程 分配/释放虚拟内存 读/写虚拟内存 保护虚拟内存 锁/开锁虚拟内存 查询虚拟内存 刷新虚拟内存

(a)

	线　　程
对象类型	
对象体属性	客户ID 线程上下文 动态优先级 基本优先级 线程处理器关系 线程执行时间 报警状态 挂起计数 模拟令牌 终止端口 线程消失状态
服　务	创建线程 打开线程 查询线程信息 设置线程信息 当前线程 终止线程 获得上下文 设置上下文 挂起 恢复 线程报警 测试线程报警 登记终止端口

(b)

图 2.18　Windows NT 进程和线程对象

(a) 进程对象　(b) 线程对象

当NT创建一新进程时，它会使用对象类或对象类型。这可以说是一种NT进程定义产生新的对象实例的模板。进程对象在创建时，它将被赋予属性值。表2.16列出了一进程对象的每一属性的简短定义。

表 2.16　Windows NT 进程对象属性

属　性	说　明
进程ID	一个惟一确定进程的值(或编号)
访问令牌	包含关于登录用户安全信息的可执行对象
基本优先级	一个进程的线程的基本执行优先级
缺省处理器关系	进程的线程能够运行缺省的一组处理器
配额限制	一个用户进程所能使用的在分页或分页系统中的最大内存值、分页文件空间以及处理器时间
执行时间	进程中所有线程已经执行的时间
I/O计数器	记录进程的线程执行过的I/O操作的类型和数量的变量
VM操作计数器	记录进程的线程执行虚拟内存操作的类型数量的变量
异常/调试端口	当进程的线程产生异常时，进程管理器向进程传送信息的通信管道
消失状态	进程终止的原因

一个NT进程至少必须包含一个执行线程。该线程可能接着创建其他线程。在多处理器系统中，来自于同一进程的多个线程可能并发执行。

图2.18(b)描述了一线程对象的对象结构，表2.17定义了该线程对象的属性。

表 2.17　Windows NT 线程对象属性

属　性	说　明
客户ID	当线程调用一个服务时，它是一个惟一的定义该线程的值(或编号)。
线程上下文	定义一个线程执行状态的一组寄存器值以及其他易变的数据。
动态优先级	在任一给定时刻，线程的执行优先级
基本优先级	线程动态优先级的最低限
线程处理器关系	线程所能运行的一组处理器，它是线程的进程与处理器关系的一个子集或全集
线程执行时间	线程累计执行时间
报警状态	一个显示了该线程是否该执行一个异步过程调用的状态
挂起计数	线程的执行被挂起的次数
令牌	一个临时访问令牌，允许线程代表一个其他进程执行一个操作
终止端口	一个进程间的通信管道，通过它，进程管理器在线程终止时发送一个消息
线程消失状态	线程终止的原因

注意：在线程的某些属性与进程相似的情况下，线程的属性是源于进程的。举例来说，在一多处理器系统中，线程处理器类是可执行该线程的处理器的集合；该

集合与进程处理器类相等或为其子集。

线程对象有一个属性是线程上下文，其信息能使线程被挂起并恢复，而且，当线程被挂起时，靠转变其上下文就可以转变该线程的行为。

2. 多线程的进程

NT支持进程间的并发，因为不同进程间的线程可以并发执行。而且，同一进程的多个线程可以分配给独立的处理器并且并发执行。多线程的进程也可获得并发性而不需要总是使用多个进程。同一进程内的线程之间通过共享内存交换信息，并可以访问该进程的共享资源。

面向对象的多线程的进程是一种提供服务器应用的有效方法，图2.19描述了它的一般概念。一个单一的服务器进程可向许多客户端提供服务。每个客户端的请求都将触发服务器内部一新线程的创建。

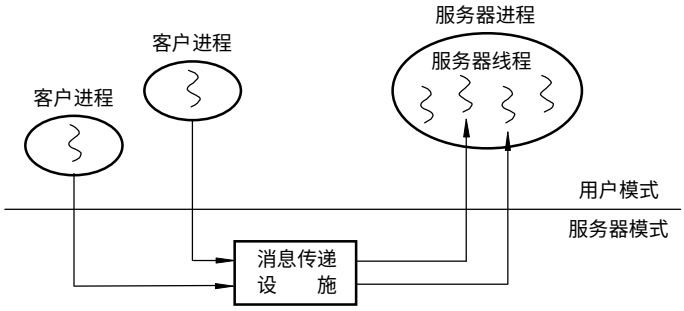


图 2.19　Windows NT 多线程服务器

3. 操作系统子系统的支持

利用NT进程与线程特性来仿真相应操作系统的进程与线程机制是每一个操作系统子系统的责任。

通过观察进程的创建过程，可以了解操作系统是如何支持进程与线程的。进程的创建开始于操作系统应用中对新进程的请求。创建进程的请求从一应用程序发给相应的受保护子系统，该子系统又发送一进程请求NT执行。由NT创建一进程对象并将该对象的句柄返回给子系统。当NT创建一进程时，它不会自动创建线程。而在Windows 32和OS/2系统中，新进程的创建总是伴随着线程的创建。对于这些操作系统而言，子系统要再次调用NT进程管理器为新进程创建线程，16位Windows和POSIX不支持线程。因此，就这些系统而言，子系统从NT获得新进程的一个线程，就可激活该进程，但仅仅返回进程信息给应用程序。

当一新进程在Windows 32或OS/2中被创建时，它继承了很多创建进程的属性，然而，在NT环境中，进程并非直接被创建。客户进程发出其进程创建请求给操作系统子系统，该子系统中一进程又发出一进程请求给NT执行。因为希望的结果是新进程继承客户端进程的特征，而非服务器进程的特征，所以NT使子系统能够识别该新进程的父

进程。它将继承其父进程的存取令牌、配额限制、基本优先级和缺省处理器类。

在NT中，进程之间没有预先定义的联系，而POSIX和OS/2都增加了一种级别联系。除了初始进程，每一进程都由另一进程所创建并被认为从属于该创建进程。使用对象句柄，操作系统子系统可以维持进程间的这种联系。

2.5.3　MVS

1. MVS 任务

MVS操作系统利用了高度结构化的进程环境。在MVS中，虚拟内存被划分成称之为地址空间的大区域。大致说来，单一的地址空间对应于单一的应用。在地址空间中，会产生许多任务。MVS任务就是MVS中调度的单元。

通常，同时存在的一百或更多的地址空间中的每一个都有很多任务。因此，MVS能够经常地并发管理成千上万的任务。

图2.20所示为一个采用某种方式在一地址空间内产生多个任务的例子。该应用支持来自许多终端的查询事务。程序分为4个模块：主程序和3个用于3种形式查询的程序(用户、预订和生产)。当应用程序装入系统时，地址空间开始于某些属于MVS的内存(空间)和加载的主程序如图2.20(a)所示。

然后，一终端用户进入一用户查询。于是，主程序发出一创建新任务的请求以执行用户模块，该模块被装入地址空间，如图2.20(b)所示。当该事务执行时，第二个终端发出一预订请求，该模块作为一个单独的任务被装入，如图2.20(c)所示。接着，当这两个任务仍在执行时，第三个终端又发出一预订请求。此时，预订模块的代码已在地址空间中，并可被用于不止一个任务。然而，每个任务为处理局部变量和进行堆栈处理，需要各自私有的内存区域，如图2.20(d)所示。

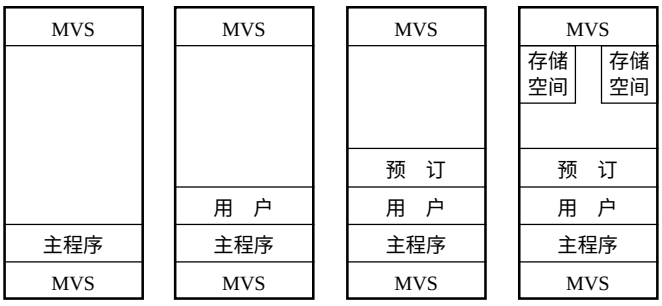


图 2.20　MVS 应用程序的地址空间布局

显而易见，MVS只利用了3种任务状态：准备、活动和等待。然而，整个地址空间可被换到一辅存中。这样，该地址空间的所有任务将被挂起。

每个任务是由任务控制块(TCB)表示的。除此以外，还有一种可调度单元——服

务请求。它是一种系统任务，且有两种(形式)。一些系统任务为某一具体应用提供服务，并在该应用的地址空间中执行；而其他涉及交叉地址空间操作的任务不在任何用户地址空间，而在系统的保留地址空间内执行。综合来说，这些系统任务可以被认为是MVS操作系统的内核。每个系统任务都有其自己的服务请求块(SRB)，当调度下一任务时在所有就绪的SRB和TCB中进行的调度选择。

这样，在MVS中，找到了两个显著的现代操作系统的概念：进程中的线程(概念)和作为进程集合的操作系统的概念(见图2.12(c))。

2. 与 MVS 任务相关的控制结构

表2.18列出了MVS用于管理用户和系统任务的主要控制结构。这些结构集中地包含了许多在进程控制块中提到的内容。该表中使用全局服务请求来管理不在用户地址空间内执行的系统任务。其余结构都与地址空间管理有关。

表 2.18　MVS 进程控制结构

表　项	说　明
全局服务请求块(GSRB)	代表一个系统任务，该任务独立于任何用户地址空间，包括： • 例程指派的ASCB地址 • 指向例程的入口指针 • 传给例程的参数地址 • 优先级
地址空间控制块(ASCB)	包含广泛地址空间信息。包括： • 用来创建ASCB队列的指针 • 该地址空间是否已交换出 • 调度优先级 • 分配给该地址空间的实际和虚拟地址空间 • 在该空间中就绪的TCB数目 • 指向ASXB的指针
地址空间扩展块(ASXB)	包含有关地址扩展空间的信息，包括： • 该地址空间中TCB个数 • 指向该地址空间TCB分派队列的指针 • 指向该地址空间局部SRB分派队列的指针 • 指向中断处理保护区的指针
局部服务请求块(LSRB)	代表一个运行特定用户地址空间的系统任务，包括： • 例程指派的ASCB地址 • 指向例程的入口指针 • 传给例程的参数地址 • 优先级
任务控制块(TCB)	代表一个执行中的用户程序，包含管理任务所需的信息： • 处理器状态信息 • 指向该任务程序部分的指针

地址空间控制块(ASCB)和地址空间扩展块(ASXB)，包含了有关地址空间的信息。它们的区别在于，当在系统保留的地址空间中执行MVS时，需要ASCB的信息；当MVS在一个指定的用户地址空间内处理和执行时，需要ASXB包含的信息。局部SRB和TCB包含了地址空间内可调度实体的信息。

图2.21显示了这些结构是如何组织的，并给出了MVS所遵守的调度规则。全局SRB保持在系统队列区的降序优先级队列中，它是一个不能被交换的主存区域。因此，全局SRB在主存中始终是可用的。当一处理器可用时，MVS首先在该队列中搜索一处于就绪状态的SRB。如果没找到，MVS会接着寻找一至少含有一个处于就绪状态的TCB或者SRB的地址空间。为此，ASCB的队列保持在系统队列区中。每个地址空间都被分配总的优先级别；ASCB可按降序排列。

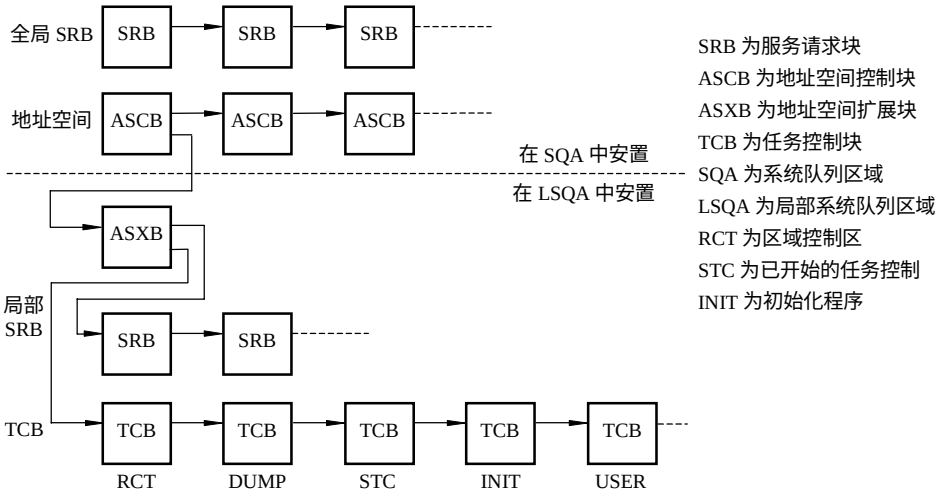


图 2.21　MVS 分派队列

一旦选择了一个地址空间，MVS就可与该地址空间里的数据结构一起工作，而该地址空间也被归于局部系统队列区域。MVS选择具有最高优先级的SRB，不然，就选择最高优先级的TCB进程调度。在此例中，显示了一批量工作的TCB队列结构。它由以下任务构成：

① 区域控制任务。负责为所有在地址空间中运行的任务管理该地址空间。

② 清空任务。如果地址空间非正常终止，则负责清空该地址空间。

③ 启动控制任务。启动地址空间的程序TCB。

④ 初始任务。负责装入批量工作流。

⑤ 用户任务。应用程序本身就是由一个或多个用户任务构成。

将控制信息分成全局和局部信息的优点在于，有关地址空间的尽可能多的信息

可以被换出地址空间，从而节省了主存。

2.6 小　结

现代操作系统中最基本的组件就是进程。操作系统的重要功能就是创建、管理和终止进程。当进程处于活动状态时，操作系统必须保证每一进程都分配到处理器执行时间，还要协调它们的活动，管理冲突请求，并分配系统资源给这些进程。要履行其进程管理职能，操作系统必须维持对每一进程的描述。每个进程都是由一个进程映像来表示的，它包括进程执行的地址空间和一个进程控制块。后者包含了操作系统管理该进程所需的全部信息，包括其目前的状态，分配给它的资源、优先级及其他有关数据。

在进程生命周期中，它会在很多状态之间转换。这些状态中最重要的是就绪、运行和阻塞。就绪进程是指目前并未执行但一旦得到操作系统调度就准备运行的进程。运行进程是指当前正被处理器执行的进程。在一多处理器系统中，可有不止一个进程处于该状态。阻塞进程是等待某事件完成(比如，I/O操作)的进程。

运行进程可能被中断或执行操作系统的访管而中止，所谓中断是指发生在进程之外并可被处理器所识别的事件。在这两种情况下，处理器都将执行一切换操作，将控制转交给操作系统例程。在完成所需工作后，操作系统可能恢复被中止的进程或切换到另一进程。

一些操作系统区分了进程和线程的概念，前者与资源的拥有有关，而后者与程序的执行有关，这可提高效率和编程的便利性。

习　题　二

2.1　表2.19显示了VAX/VMS操作系统的进程状态。问：

(1) 存在这么多明显的等待状态，你能给出一个合理的解释吗？

(2) 为什么以下状态没有常驻和换出版本：页面错等待、冲突页等待、公共事件等待、空闲页等待和资源等待？

(3) 画出状态转换图并标明引起每一转换的动作或发生的事件。

2.2　Pinkert和Wear定义了下列进程状态：执行(运行)、活动(就绪)、阻塞和挂起。一进程如在等待使用某资源的许可时，该进程就处于阻塞(状态)，若它在等待对其已获得资源上的某操作完成，则该进程处于挂起(状态)。在很多操作系统中，这两种状态合在一起作为封锁状态。比较这两种形式定义的优缺点。

2.3　就图2.7(b)的7状态进程模型，画出与图2.6(b)相似的排队流程图。

表 2.19　VAX/VMS 进程状态

进程状态	进程所处的情况
当前正在执行	正在执行进程
可计算(驻留)	就绪并且在内存中驻留
可计算(交换出)	就绪，但是被交换出了内存
页面错等待	进程访问了一个并不在内存中的页，并且必须等待该页被读入
冲突页等待	进程访问了一个共享页，该页是引起另一进程的页面错等待的原因，或者是一个当前正被读入或写出的私有页
公共事件等待	等待共享事件标志(事件标志是单个比特位的进程间信号量机制)
空闲页等待	等待内存中的一个空闲页以加入到分配给这个进程的内存页集合中
睡眠等待(驻留)	进程将自己放入等待状态
睡眠等待(交换出)	睡眠进程被交换出内存
局部事件等待(驻留)	进程在内存中并且等待一个局部事件标志(通常是I/O结束)
局部事件等待(交换出)	进程处于局部事件等待状态并被交换出内存
挂起等待(驻留)	进程被其他进程置于等待状态
挂起等待(交换出)	挂起的进程被交换出内存
资源等待	进程在等待各种系统资源

2.4　考虑图2.7(b)所示的状态转换图。假定现在操作系统要调度一进程，且存在就绪状态和就绪挂起状态的进程，而且至少有一处于就绪挂起状态的进程拥有比任何处于等待状态进程更高的调度优先级。两种极端的策略是：

(1) 总是调度一处于就绪状态的进程以使交换达到最少；

(2) 总是优先考虑具有最高优先级的进程，即使这样可能引起不必要的交换。

请提出一种折衷策略以平衡优先级与执行情况的关系。

2.5　VAX/VMS操作系统利用了4个处理器存取模式来加强进程之间系统资源的保护和共享。这种存取模式决定了以下内容：

① 指令执行特权。处理器可以执行的指令。

② 存储访问特权。当前指令可以存取的虚拟内存的位置。

4种模式如下：

① 内核。执行VMS操作系统的内核，包括内存管理、中断处理和I/O操作。

② 执行。执行许多操作系统服务调用，包括文件记录(磁盘和磁带)管理例程。

③ 管态。执行其余操作系统服务，如应答用户请求。

④ 用户态。执行用户程序，以及诸如编译、编辑、链接和调试程序的实用程序。

处于较低特权状态下执行的进程经常需要调用在更高特权状态下执行的过程；举例来说，一用户程序需要一操作系统的服务。使用CHM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新的存取状态的例程。执行REI指令将返回(从异常或中断返回)。问：

(1) 许多操作系统有两种状态，内核和用户。使用4种状态代替这两种状态有何优缺点？

(2) 你能举出多于4种状态的实例吗？

2.6　在题2.5中提及的VMS方案通常被认为是一种环保护结构，如图2.22所示。事实上，简单的内核/用户方案，正如2.3节描述的那样，是一种双环结构。Silberschatz和Galvin指出这种结构的一个问题：

这种环(层次)结构的主要不足在于，其不能允许我们坚持需知原则。特别地，如果在域D_j中的某对象必须是可访问的，而在域D_i中则不行，那么一定有$j<i$，但这就意味着每个在D_i中可访问的段在D_j中也可访问。

(1) 请更清楚地解释上面引语中所提到的问题。

(2) 请给出一种使得操作系统能够解决此问题的方法。

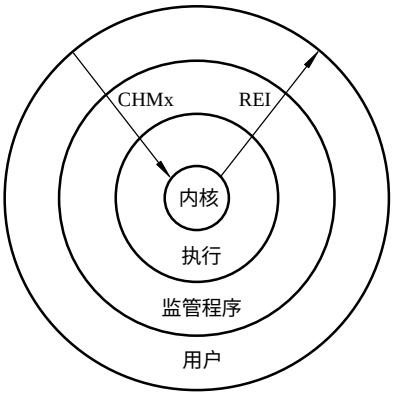


图 2.22　VAX/VMS 存取模式

2.7　图2.6(b)示意在某一时刻，一进程可以在仅有一个事件的队列中。

(1) 如果想让一进程同时等待不止一个事件，这可能吗？请举例说明。

(2) 在这种情况下，该如何修改图中的队列结构以支持这一新的特点？

2.8　在很多早期的计算机中，由寄存器值引起的中断被存储在与给定中断信号有关的固定位置。试问在何种环境下，该技术是一种实用技术？请解释为什么这种技术通常是不方便的？

2.9　在一进程中使用多线程的两大优点是：

(1) 在已存在的进程中创建一新线程所需的工作量要比创建一新进程的工作量少；

(2) 同一进程内各线程间的交流被简化了。

那么，是否在同一进程中的两线程之间的上下文切换要比在不同进程中的两线程间的上下文切换所需工作量也要少呢？

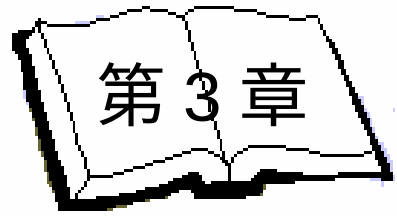
2.10　如2.5节所述，UNIX并不适合实时应用，因为在内核模式下执行的进程不可能被抢占。请对此加以详细描述。

2.11　如果删除会话和与进程有关的用户接口(键盘，鼠标，显示器屏幕)，在OS/2中与进程相关的概念数可从3个减至2个。这样，某一时刻的进程就处于前台的状态，为了进一步结构化，进程可划分成线程。问：

(1) 这种方法丧失了何种优势？

(2) 如果继续进行这种修改，你将在何处分配资源(内存，文件等)，是在进程级还是在线程级？

2.12　一进程完成其任务并退出，需要何种基本操作来清除它并继续另一个进程？



并发控制——互斥与同步

进程和线程是现代操作系统的重要概念，操作系统设计的核心是进程管理，主要涉及下述 3 个方面：

① 多道程序，即单个处理机中多个进程的管理。大部分个人电脑、工作站、单处理器系统上的操作系统，例如 Windows 98、OS/2 和 Machintosh System 7 都支持多道程序，共享单处理机系统，UNIX 也支持多道程序。

② 多进程，即多个处理机中多个进程的管理。多处理机一般用于较大的系统，诸如 MVS 和 VMS 之类的操作系统支持多进程。近来，服务器和工作站也开始支持多进程，Windows NT 就是一个为这种环境而设计的操作系统。

③ 分布式进程，即运行于分布式计算机系统中的多个进程的管理。

上述 3 个方面和操作系统设计的共同基础是并发，并发包含了大量的设计问题，包括进程间通信、竞争和共享资源，多个活跃进程的同步和处理机时间的分配等。这些问题并非仅存在于多进程和分布式进程系统中，也存在于单处理机多道程序的系统中。

在以下 3 种情况下可能出现并发：

① 多个应用。多道程序用于多个作业或多个应用程序动态地共享处理机时间。

② 结构化应用。如果多个应用程序遵循结构化程序设计和模块化设计的原则，那它们就可以作为并发进程高效地实现。

③ 操作系统结构。如果将结构化的优点应用到操作系统中，那么，这些操作系统本身也可作为一个并发进程集高效地实现。

本章首先介绍并发的概念和并发进程的实现。支持并发的基本要求是互斥，即若一个进程正在执行，则其他所有进程将被排斥执行；本章的第二部分讨论实现互斥的可能途径，所有方法都是通过软件加以解决的，并且要用到“忙碌-等待”(Busy Waiting)技术。由于这些方法过于复杂，而且“忙碌-等待”方法也不理想，这就要寻找不包含“忙碌-等待”的方法，这些方法还需要得到操作系统

的支持或用编译器实现。本章讨论了 3 种方法：信号量(Semaphores)、管程(Monitors)和消息传递(Message passing)。

我们还用两个经典问题描述这些概念并比较所提到的方法，它们是生产者/消费者(Producer/Consumer)问题和读者/写者(Readers/Writers)问题。

3.1　并发原理

在一个单处理机多道程序的系统中，进程被分隔插入到处理机时间中交替执行，使进程表面看起来是同时执行的，如图3.1(a)所示，尽管这不是真正的并行，并且进程间的调度将带来额外的开销，但插入执行的主要好处是提高了执行效率，在多处理机系统中不仅可以插入执行，还可以重叠执行，如图3.1(b)所示。

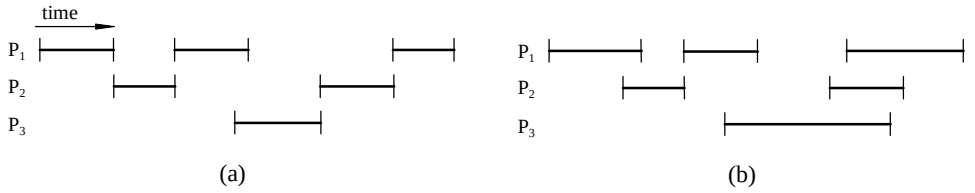


图 3.1　并发进程的执行

(a) 插入执行　(b) 插入和重叠执行

插入执行和重叠执行看起来好像是两种根本不同的执行方式，事实上，它们都是并发执行方式。以单处理机系统为例，多道程序中进程执行的相对速度无法预测的问题依赖于以下几方面：其他活动进程的状况，操作系统处理中断的方式，操作系统的调度规则。此外，还存在如下困难：

① 全局变量的共享充满危险。如果两个进程使用同一全局变量，并对此变量进行读/写操作，则不同的读/写顺序将产生矛盾的结果。

② 资源的最优分配对操作系统而言是困难的。例如，进程A申请特定的I/O通道，并在使用该通道前被挂起，如果操作系统封锁了该通道，并禁止其他进程使用，则将降低系统效率。

③ 由于结果通常不可再现，故使程序错误的标记变得困难。

上述困难在多处理机系统中同样存在，因为这里也有进程执行的相对速度不可预测的问题，多处理机系统还必须处理多个进程同时执行带来的问题。

一个简单的例子：

```
procedure echo;
var out,in:character;
begin
    input (in, keyboard);
    out：=in;
    output (out, keyboard);
end.
```

这个过程是一个字符回显程序的基本部分，每次敲击键盘得到输入，每一输入字符都先存储在变量in中，再将它传给变量out，并输出出来，任何程序都可重复调用这个过程来得到输入并将其输出到显示器上。

对于一个支持单用户的单处理机多道程序系统，用户可以从一个应用转到另一个应用，每一应用都使用同一键盘进行输入，用同一显示器进行输出。因为每一应用都必须使用过程echo，将其存入所有应用的全局共享存储区作为共享进程是有益的，这样只保留了echo的一个副本，从而节省了空间。

考虑下面的执行结果：

① 进程P_1调用过程echo，在得到函数input的结果后被中断，此时，输入的字符x存储在变量in中。

② 进程P_2调用过程echo，它执行输入并在显示器上输出字符y。

③ 进程P_1再次执行，这时x由于被覆盖而丢失，in中保存的是y，y将传给out而输出。

显然第一个字符丢失而第二个字符显示了两次。问题的实质在于共享变量in可被多个进程访问。如果一个进程修改全局变量，然后被中断，另一进程可能在第一个进程使用此变量值前被中断，假定echo仍是一个全局过程，只要一次只有一个进程执行该过程，则上述的执行结果就发生变化：

① 进程P_1调用过程echo，在得到函数input的结果后被中断，此时，最先输入的字符x存储在变量in中。

② 进程P_2调用过程echo，因为P_1仍在使用过程echo，尽管P_1被挂起，但P_2仍被封锁在过程echo之外，所以P_2被挂起直到得到过程echo的使用权为止。

③ 随后，进程P_1再次执行并完成echo的执行，字符x被输出。

④ 当P_1退出echo后，解除对P_2的封锁，当P_2再次执行时，过程echo将被成功地调用。

这个例子说明必须保护共享全局变量和其他共享的全局资源，这也是控制代码访问共享变量的惟一方法。如果加上以下规则，一次只能有一个进程进入到echo，并且一旦进入echo，其他进程就不能访问echo，直到该过程完成为止，那么，可以避免出现上述类型的错误。如何加上这种规则是本章讨论的主要问题。

这里假定问题存在于单处理机多道程序的操作系统中，以上例子说明在单处理机系统中也会出现并发问题。在多处理机系统中同样存在共享资源的保护问题，也需要同样的解决方法，首先假设对于共享的全局变量没有任何控制的访问机制。

① 进程P_1和P_2各自在不同的处理机上执行，并且都调用过程echo。

② 以下事件以相同次序并行发生。

PROCESS P_1	PROCESS P_2
M	M
input(in, keyboard)	input(in, keyboard)
out：=in	out：=in
output(out, keyboard)	output(out, keyboard)
M	M

结果是输入到P_1中的字符在显示前丢失，而输入到P_2中的字符被P_1和P_2输出。如果允许一次只能有一个进程进入echo的规则，则会出现以下结果：

① 进程P_1和P_2各自在不同的处理机上执行，P_1调用过程echo。

② 当P_1在执行过程echo时，P_2调用echo。因为P_1还在echo中(不管P_1挂起还是在执行)，P_2将被封锁进入echo，所以P_2被挂起直到得到过程echo的访问权。

③ 随后，进程P_1完成echo并退出。一旦P_1退出echo，P_2就再次激活并执行echo。

在单处理机系统中，出现这个问题的原因是，进程中的中断可在任何地方中止指令的执行，在多处理机系统中除了有这个因素外，两个进程的同时执行并试图访问同一全局变量也将引起这个问题。但是，解决这两种类型问题的方法是相同的，即有效控制对共享资源的访问。

并发的存在对操作系统的设计和管理提出了以下要求。

① 操作系统必须能跟踪大量活跃进程(这可使用进程控制块完成)。

② 操作系统必须为每一活跃进程分配资源，这些资源包括：处理机时间、内存、文件、I/O设备。

③ 操作系统必须保护每一进程的数据和物理资源不被其他进程侵犯，这包括与存储器、文件和I/O设备相关的技术。

④ 进程执行的结果与其他并发进程执行时的相对速度无关。

3.1.1　进程间的相互作用

并发进程存在于多道程序环境，多道程序包括多个独立的应用、多进程应用和操作系统中多进程结构的使用。本小节对于所述种种可能的情况讨论进程间相互作用的方法。

根据进程觉察到其他进程的存在程度可将进程间的相互作用分为3类，如表3.1

所示。

表 3.1　进程间相互作用

觉察程度	相互关系	一进程对其他进程的影响	潜在的控制问题
进程互不觉察	竞争	• 一进程独立于其他进程 • 进程执行时间将受影响	• 互斥 • 死锁 • 饥饿
进程间接觉察	通过共享 合　作	• 一进程依赖于从其他进程得到信息 • 进程执行时间将受影响	• 互斥 • 死锁 • 饥饿 • 数据一致性
进程直接觉察	通过通信 合　作	• 一进程依赖于从其他进程得到信息 • 进程执行时间将受影响	• 死锁 • 饥饿

1. 进程互不觉察

这些独立的进程并不协同工作。最好的例子是多个独立进程的多道程序，它们可以是大量的作业或是相互作用的段或是二者的混合。尽管进程并不协同工作，但操作系统还是要解决竞争资源的问题。例如，两个独立的应用可能要访问同一文件或打印机，操作系统必须处理好这些访问。

2. 进程间接觉察

这些进程并不是通过名字觉察到对方，而是通过共享访问，比如一个I/O缓冲区。这些进程通过共享方式进行合作。

3. 进程直接觉察

这些进程可通过名字直接通信并设计紧密的协同工作方式。

需要指出的是，实际情况并不像表3.1中所示的那样可以截然区分，一些进程可能同时表现出竞争和协作两个方面，而且分别检查上述3种情况并在操作系统中加以实现的效率较高。

3.1.2　进程间的相互竞争

并发进程在竞争使用同一资源时将产生冲突。这种情况可简单地描述为，两个或更多的进程在执行时要访问某一资源，每一进程都没有觉察到其他进程的存在，并且每一进程也不受其他进程执行的影响。这就要求进程留下其所用资源的状态，这些资源包括I/O设备、存储器、处理机时间和时钟。

进程在相互竞争资源时并不交换信息，但是，一个进程的执行可能影响到同其竞争的进程，特别是如果两个进程要访问同一资源，那么一个进程可通过操作系统分配到该资源，而另一个将不得不等待。在极端的情况下，被阻塞的进程将永远得不到访问权，从而不能成功地终止。

进程间的竞争面临3个控制问题：

(1) 互斥(mutual exclusion)

假设两个或两个以上进程要访问同一不可共享的资源(例如打印机)，那么，在执行期间，每一进程将发命令给I/O设备，然后收到状态信息，发送数据并/或接收数据。这种资源称为临界资源(critical resource)，访问临界资源的那一部分程序称为程序的临界段(critical section)。在任一时刻只允许一个程序在其临界段中是至关重要的。这里不能简单地依赖操作系统来理解并执行这种约束，因为具体的需求并不是显而易见的。例如，打印机在打印整个文档的时候，需要有独立的进程来控制打印机，要不然就要插入相互竞争的其他进程的文档，以致出现夹杂不清的状况。

(2) 死锁(deadlock)

互斥的实现产生两个额外的控制问题，一个是死锁，考虑两个进程P_1、P_2和两个临界资源R_1、R_2，假设每个进程都要访问这两个资源，这就可能出现以下情况。操作系统把R_1分配给P_2，R_2分配给P_1，每一进程都在等待另一资源，每一进程都不释放它所占有的资源，除非它已得到另一资源并执行完临界段，故最终所有的进程将死锁。

(3) 饥饿(starvation)

互斥的实现所产生的另一个控制问题是饥饿。假设有3个进程P_1、P_2和P_3，每一个都要周期地访问资源R。考虑以下情况，P_1占有资源，P_2和P_3都被延迟、等待资源，当P_1离开临界段时，P_2和P_3都可分配到R，假设P_3得到R，在P_3完成其临界段之前，P_1又申请得到R，并且如果P_1和P_3重复得到R，则P_2就不可避免地得不到该资源，尽管这里并没有死锁，这时称P_2处于饥饿状态。

竞争的控制不可避免地涉及到操作系统，因为是操作系统分配资源，另外，进程自身也必须能以某种方式表达互斥的要求，例如，在使用某一资源之前先将其封锁。任何方法都要得到操作系统的支持，例如，提供加锁的工具。图3.2使用抽象

```
program mutualexclusion;
const n=…; (*number of processor*);
procedure P(i:integer);
begin
  repeat
    entercritical(R);
    〈critical section〉;
    exitcritical(R);
     〈remainder〉
  forever
end;
begin (*main program*)
  parbegin
    P(1);
    P(2);
    …
    P(n)
  parend
end.
```

图 3.2　互斥

语言描述了互斥机制。有几个进程互斥执行，每一进程包含一个使用资源R的临界段

和一个与R无关的剩余段。为实现互斥提供了两个函数：entercritical和exitcritical。每一函数可看作是对可竞争资源名字的声明，在有其他进程进入到其临界段时，任一为竞争同一资源而需要进入临界段的进程将等待。

3.1.3　进程间的相互合作

1. 通过共享合作

进程间通过共享合作并不需要清楚地觉察到对方。例如，多个进程可以访问共享变量、共享文件或数据库，进程不需要参照其他进程就可使用和修改共享数据，但要知道其他进程也可访问同一数据，因此，进程必须合作以管理好共享数据。这种控制机制必须保证共享数据的完整性。

因为数据存于资源(设备、存储器)上，所以出现了互斥、死锁、饥饿等控制问题，惟一的不同是，数据项有两种不同的访问方式，读和写，并且只有写操作必须互斥执行。

除上述问题之外还有一个新的要求，即数据相关性。例如，假设有两个数据项a和b。它们要保持关系a=b，也就是说，任何程序在修改一个值时也必须修改另一个以保持这种关系。现在有以下两个进程：

P_1:　a:= a+1;　　P_2:　b:= 2∗b;

　　b:= b+1;　　a:= 2∗a;

如果起始数据是一致的，则每一进程分别执行后，共享数据仍是一致的。考虑以下两个进程在各个互斥数据上并发执行的情况：

a:= a +1;

b:= 2∗b;

b:= b+1;

a:= 2∗a;

以上执行序列结束后，就不再有a=b。这个问题可通过将每个进程的整个序列声明为临界段加以解决，但严格地说这里并未涉及到临界资源。

因此，可以看出在共享合作中临界段概念的重要性，这里同样可使用图3.2中的抽象函数entercritical和exitcritical。如果用临界段解决数据的完整性，那就没必要声明任何特殊的资源或变量，在这种情况下，可以认为在并发进程中共享的声明都标记了必须互斥的临界段。

2. 通过通信合作

在以上讨论的两个例子中，每一进程都是在不含其他进程的孤立环境中。进程间的工作影响是间接的。在两个例子中用到的都是共享，在竞争的例子中，它们在没有觉察其他进程的情况下共享资源。在第二个例子中，它们共享变量并且不能清楚地觉察到其他进程，但要保持数据的完整性。进程在通过通信合作时，通信为其

提供了在不同的活动中同步或合作的途径。

通常通信可描述为包括某种形式的消息，收、发信息的原语，它们是作为程序设计语言的一部分或由操作系统的内核提供。

因为在消息传递中不存在共享，所以这种形式的合作不需要互斥，但是还存在死锁和饥饿问题。举一死锁的例子，两个进程可能由于都在等待对方的通信而阻塞。饥饿的例子是，假设有3个进程P_1、P_2和P_3，P_1不断地同P_2或P_3通信，P_2和P_3也要同P_1通信，可能会出现P_1和P_2重复地交换信息，而P_3因为等待同P_1通信而阻塞，这里没有死锁，因为P_1保持活跃而P_3饥饿。

3.1.4　互斥的要求

并发进程的成功完成需要有定义临界段和实现互斥的能力，这是任何并发进程方案的基础，任何支持互斥的工具和方法都要满足以下的要求。

① 实现互斥，即在任一时刻，在含有竞争同一共享资源的临界段的进程中，只能有一个进程进入自己的临界段。

② 执行非临界段的进程不能受到其他进程的干扰。

③ 申请进入临界段的进程不能被无限期延迟，也不允许存在死锁和饥饿。

④ 当没有进程进入临界段时，任何申请进入临界段的进程必须允许无延迟地进入。

⑤ 没有进程相对速度和数目的假设。

⑥ 进程进入到临界段中的时间有限。

这里有许多实现互斥要求的方法。一种方法是对并发进程不提供任何支持，因此，无论它们是系统程序或应用程序，进程都要同其他进程合作以解决互斥，它们从程序设计语言和操作系统得不到任何支持，这些称为软件方法。尽管软件方法容易引起较高的进程负荷和较多的错误，但它可使我们对并发的复杂性有较深的理解。另一种方法是使用特殊的机器指令，这可以降低开销。第三种方法是在操作系统中支持分级。下面就讨论这些方法。

3.2　互斥——用软件方法实现

软件方法可在共享主存的单处理机或多处理机系统中实现并发进程，这些方法通常假定基于内存访问级别的一些基本的互斥，即对内存中同一位置的同时访问(读/写)将被排序，而访问的顺序并不预先指定，除此之外不需要硬件、操作系统和程序设计语言的任何支持。

3.2.1　Dekker 算法

1. 初次尝试

Dekker算法的优点在于它描述了并发进程发展过程中遇到的大部分共同问题。

任何互斥都必须依赖于一些硬件上的基本约束，其中最基本的约束是任一时刻只能有一个进程访问内存中某一位置。下面以一个“小屋协议”为例来描述这个问题。

这个小屋本身和它的入口都非常小，以致在给定的任何时刻，只有一个人可以呆在屋内，屋内有一块只能写一个号码的小黑板。

协议内容为，一个希望进入临界段的进程(P_0或P_1)，爬进小屋并查看黑板，若黑板上写着自己的编号，就离开小屋并去执行它的临界段，若黑板上写着他人的编号，它就离开小屋并等待，然后不时地进屋查看黑板，直至允许进入自己的临界段。这种做法称之为忙碌-等待(Busy-waiting)。因为等待进程在进入临界段之前没做任何有意义的工作，实际上它必须不断地“闲逛”并查看小黑板，故在等待进入临界段期间，它浪费了大量的处理机时间。

在进程进入其临界段并执行完临界段后，它必须返回小屋并在黑板上写上其他进程的编号。

如果我们将这一问题用程序形式描述，则可写成如下进程形式：

var turn: 0..1;　{turn为共享的全局变量}

两个进程的程序为

PROCESS 0	PROCESS 1
M	M
while turn≠0 **do** {nothing}	**while** turn≠1 **do** {nothing};
〈critical section〉;	〈critical section〉;
turn: =1;	turn: =0;
M	M

这种方法保证了互斥，但它也有两点不足：首先，进程必须严格地交替执行它们的临界段，因此，执行的速度由进程中较慢的一个决定，如果P_0每小时只进入临界段一次，而P_1每小时进入临界段1000次，则该协议将迫使P_1的工作节拍同P_0保持一致；一个更严重的问题是，如果一个进程发生意外(例如在去小屋的途中被熊吃掉)，则另一个进程会永远死锁，不管进程是否在其临界段中发生意外都将导致这种情况发生。

2. 第 2 次尝试

初次尝试的主要毛病在于它只记住了允许哪个进程进入其临界段，但未记住每个进程的状态。事实上，每个进程都有进入临界段的钥匙，这样当其中一个被熊吃掉后，另一个仍可进入自己的临界段；每一个进程都有自己的小屋，每一个进程都

可以查看另一个的黑板，但不能修改。当一个进程想进入其临界段时，它将不时地查看另一个进程的黑板直到发现上面写着“false”为止，这就意味着另一进程并不在其临界段内，于是它迅速爬进自己的小屋，将黑板上的“false”改为“true”，进程现在就可以执行其临界段了，当进程执行完自己的临界段后，再将自己黑板上的“true”改为“false”。

现在，共享的全局变量是：

```
var flag: array[0..1]of boolean;
```

它被初始化为false，两个进程的程序为

```
PROCESS 0                          PROCESS 1
    M                                  M
while flag[1]do {nothing};         while flag[0]do {nothing};
flag[0]: =true;                    flag[1]: =true;
〈critical section〉;               〈critical section〉;
flag[0]: =false;                   flag[1]: =false;
    M                                  M
```

这时，如果进程在临界段之外，包括置flag的代码发生意外，那么其他进程将不会被阻塞。事实上，其他进程只要想进入自己的临界段就可以进。因为另一进程的标志flag永远是false。但是，如果进程在其临界段中发生意外，或在刚把flag置为true还未进入其临界段之前，那么其他进程将被永远地阻塞。

从某种意义上说，这种解决方法比第一种更差，因为它甚至不能保证互斥。考虑如下序列：

P_0进入while语句并发现flag[1]=false

P_1进入while语句并发现flag[0]=false

P_0置flag[0]为true并进入临界段

P_1置flag[1]为true并进入临界段

此时，所有进程都在它们的临界段中，程序发生错误。出现这个问题的原因在于，这种解决方法依赖于进程执行的相对速度。

3. 第 3 次尝试

因为一个进程可以在其他进程检查状态后，而在其他进程还未进入临界段前可以改变自己的状态，所以第2次尝试失败了，也许可以通过交换两行代码的次序来解决这个问题。

```
PROCESS 0                          PROCESS 1
    M                                  M
flag[0]: =true;                    flag[1]: =true;
while flag[1] do {nothing};        while flag[0] do {nothing};
```

〈critical section〉;
flag[0]: =false;
M

〈critical section〉;
flag[1]: =false;
M

如果一个进程在其临界段内，包括置flag的代码发生意外，则其他进程被阻塞，如果一个进程在其临界段外发生意外，则其他进程不会被阻塞。

先从进程P_0的角度来看看这种方法能否保证互斥，一旦P_0将flag[0]置为“true”，P_1就不能进入自己的临界段直到P_0进入然后离开临界段。在P_0置flag时，P_1可能已进入自己的临界段。在这种情况下，P_0将被阻塞在while语句上，直到P_1离开临界段。从P_1的角度也可进行同样的推理。

这种方法保证了互斥但产生了另一个问题，如果所有的进程都置flag为“true”，它们都认为对方已进入临界段，这就导致了死锁。

4. 第 4 次尝试

在第3次尝试中，一个进程在置自己的状态时，并不考虑其他进程的状态。由于每个进程都坚持要进入临界段就出现了死锁。可以把每个进程弄得差异大一点来解决这一问题。进程置自己的标志flag以表明想进入临界段，但它要时刻准备着按其他进程的状态来重置自己的标志flag。

PROCESS 0
M
flag[0]: =true;
while flag[1] **do** {nothing};
begin
flag[0]: =false;
〈delay for a short time〉;
flag[0]: =true;
end;
〈critical section〉;
flag[0]: =false;
M

PROCESS 1
M
flag[1]: =true;
while flag[0] **do** {nothing};
begin
flag[1]: =false;
〈delay for a short time〉;
flag[1]: =true;
end;
〈critical section〉;
flag[1]: =false;
M

这种方法已很接近正确解法，但仍有小缺陷，采用同第3次尝试中相同的推理可证明它能保证互斥，然而在如下的事件序列中：

P_0置flag[0]为true	P_1置flag[1]为true
P_0检查flag[1]	P_1检查flag[0]
P_0置flag[0]为false	P_1置flag[1]为false
P_0置flag[0]为true	P_1置flag[1]为true

这个序列可以一直扩展下去，并且任意一个进程都不能进入临界段。严格地说，

这并不是死锁，因为两个进程执行的相对速度的任何改变都将打破循环并将允许其中一个进入临界段，尽管这种序列不可能保持很长，但它毕竟是一种可能的情况，因此，放弃第4种方法。

5. 一个正确的解决方法

通过数组flag可以能观察到所有进程的状态，但这还是不够的，故应采取措施解决以上由于“互相礼让”产生的问题，第1次尝试中的变量turn可用于此目的，这个变量可以指出哪个进程坚持进入临界段。

设计一个“指示”小屋，小屋内的黑板标明“turn”，当P_0想进入其临界段时，置自己的flag为“true”，这时它去查看P_1的flag，如果是“false”，则P_0就立即进入自己的临界段，反之P_0去查看“指示”小屋，如果turn=0，那么它知道自己应该坚持并不时去查看P_1的小屋，P_1将觉察到它应该放弃并在自己的黑板上写上“false”，以允许P_0继续执行。P_0执行完临界段后，它将flag置为“false”以释放临界段，并且将turn置为1，将进入权交给P_1。

图3.3给出了按上述思想设计的Dekker算法，其证明留做练习。

```
var flag: array[0..1]of boolean;
    turn: 0..1;
procedure P0;
begin
  repeat
    flag[0]:=true;
    while flag[1] do if turn=1 then
                  begin
                    flag[0]:=false;
                      while turn=1 do {nothing};
                    flag[0]:=true;
                  end;
    〈critical section〉;
    turn:=1;
    flag[0]:=false;
    〈remainder〉
  forever
end;
procedure P1;
begin
  repeat
    flag[1]:=true;
    while flag[0] do if turn=0 then
                  begin
                    flag[1]:=false;
                      while turn=0 do {nothing};
                    flag[1]:=true;
                  end;
    〈critical section〉;
    turn:=0;
    flag[1]:=false;
    〈remainder〉
  forever
end;
begin
  flag[0]:=false;
  flag[1]:=false;
  turn:=1;
  parbegin
    P0; P1
  parend
end.
```

图 3.3　Dekker 算法

3.2.2　Peterson 算法

Dekker算法解决了互斥的问题，但是，其复杂的程序难于理解，其正确性难于

证明。Peterson给出了一个简单的方法。和上述方法一样，数组flag表示互斥执行的每一进程，全局变量turn用于解决同时产生的冲突，图3.4给出了这个算法。

```
var flag: array[0..1] of boolean;
    turn: 0..1;
procedure P0;
begin
  repeat
    flag[0]: =true;
    turn: =1;
    while flag[1] and turn=1 do {nothing};
    〈critical section〉;
    flag[0]: =false;
    〈remainder〉
  forever
end;
procedure P1;
begin
  repeat
    flag[1]: =true;
    turn: =0;
    while flag[0] and turn=0 do {nothing};
    〈critical section〉;
    flag[1]: =false;
    〈remainder〉
  forever
end;
begin
  flag[0]: =false;
  flag[1]: =false;
  turn: =1;
  parbegin
    P0; P1
  parend
end.
```

图 3.4　两进程的 Peterson 算法

容易证明这种方法保证了互斥。对于进程P_0，一旦它置flag[0]为true，P_1就不能进入其临界段。如果P_1已经在其临界段中，那么flag[1]=true并且P_0被阻塞进入临界段。另一方面，它也防止了相互阻塞，假设P_0阻塞于while循环，这意味着flag[1]为true，而且true=1，当flag[1]为false或turn为0时，P_0就可进入自己的临界段了。下面是3种极端的情况。

① P_1不想进入其临界段。这是不可能的，因为这意味着flag[1]=true。

② P_1等待进入临界段。这也是不可能的，因为若turn=1，P_1就能进入其临界段。

③ P_1反复访问临界段，从而形成对临界段的垄断。这不可能发生，因为P_1在进入临界段前通过将turn置为0，把机会让给了P_0。

这是一个两进程互斥的简单解决方法，进一步可将Peterson算法推广到多个进程的情况。

3.3　互斥——用硬件方法解决

3.3.1　禁止中断

在单处理机中并发进程不能重叠执行，它们只能被插入，而且进程将继续执行直到它调用操作系统服务或被中断，所以，为保证互斥，禁止进程被中断就已足够。

通过操作系统内核定义的禁止和允许中断的原语就可获得这种能力，进程可按如下方式实现互斥。

```
repeat
    〈disable interrupts〉;
    〈critical section〉;
    〈enable interrupts〉;
    〈remainder〉
forever.
```

因为临界段不能被中断，互斥就得到保证。这种方法的代价较高，而且执行效率也会显著地降低，因为处理机受到不能插入的限制。第二个问题是这种方法不能用于多处理机系统。对于含有不止一个处理机的计算机系统，在同一时间通常有一个以上的进程在执行。在这种情况下，禁止中断亦不能保证互斥。

3.3.2　使用机器指令

1. 特殊的机器指令

在多处理机系统中，多个处理机共享一个主存，这里并没有主/从关系，也没有实现互斥的中断机制。

在硬件水平上，对同一存储区的访问是互斥执行的。以此为基础，设计者设计了几个机器指令以执行两个原子操作，例如，读/写或读测试，因为这些操作都是在一个指令周期内完成的，所以不会被其他指令打断。

本小节将给出两个最一般的指令：test and set指令和exchange指令。

test and set指令可定义如下：

```
function testset (var i:integer):boolean;
begin
  if i=0 then
    begin
      i:=1;
      testset:=true
    end
  else testset:=false
end.
```

这个指令首先测试i的值。如果i的值为0, 则将i赋值为1, 并且返回true。反之, i的值不变并返回false。这里将整个testset函数作为一个原语执行, 即这里不存在中断。

图3.5(a)给出了基于这种指令的互斥协议，共享变量bolt初始化为0，发现bolt值为0的惟一进程可进入其临界段，所以其他想进入临界段的进程进入忙碌-等待状况。当

一个进程离开其临界段时，它重置bolt为0。此时，处于等待状态的惟一的一个进程得到临界段的访问权。哪个进程刚好紧接着执行testset指令，哪个进程将被选中。

exchange指令定义如下：

```
procedure exchange (var r: register; var m: memory);
var temp;
begin
   temp:=m;
   m:=r;
   r:=temp
end.
```

可以用这个指令交换寄存器和内存的内容。在此指令执行期间，任何访问同一内存地址的指令将被阻塞。

图3.5(b)给出了基于这个指令的互斥协议，共享变量bolt初始化为0，发现bolt为0的惟一进程进入其临界段，它通过将bolt置为1排斥所有其他进程访问临界段。当一个进程离开临界段时，它重置bolt为0以使其他进程获得临界段的访问权。

```
program mutualexclusion;
const n=…; (*number of processes*);
var bolt: integer;
procedure P(i:integer);
begin
  repeat
    repeat {nothing} until testset(bolt);
    〈critical section〉;
    bolt:=0;
    〈remainder〉
  forever
end;
begin (*main program*)
  bolt: = 0;
  parbegin
    P(1);
    P(2);
    …
    P(n);
  parend
end.
```

```
program mutualexclusion;
const n=…; (*number of processes*);
var bolt: integer;
procedure P(i:integer);
var key_i: integer;
begin
  repeat
key_i :=1;
repeat exchange (key_i, bolt) until key_i=0;
    〈critical section〉;
    exchange (key_i, bolt);
    〈remainder〉
  forever
end;
begin (*main program*)
  bolt:=0;
parbegin
    P(1);
    P(2);
    …
    P(n);
  parend
end.
```

图 3.5　用硬件方法解决互斥问题

(a) Test and Set指令　(b) Exchange指令

2. 机器指令方法的特性

使用特殊的机器指令实现互斥有许多优点：

① 可用于含有任意数量进程的单处理机或共享主存的多处理机；

② 比较简单，所以易于验证；

③ 可支持多个临界段，每个临界段用各自的变量加以定义。

下面是一些严重的缺陷：

① 由于采用忙碌-等待，所以在进程等待进入临界段时，将耗费处理机时间；

② 有可能产生饥饿。当一个进程离开临界段并且有多个进程等待时，等待进程的选择是随意的，因此，不可避免地出现某些进程得不到访问权的情况。

③ 有可能产生死锁。在一个单处理机系统中，进程P_1执行特殊指令（例如，testset，exchange）并进入其临界段，这时，拥有更高优先级的P_2执行并中断P_1。如果P_2又要使用P_1占用的资源，那么互斥机制将拒绝该要求，这样，它就陷入忙碌-等待循环，但P_1也永远不能终止，因为它比另一预备进程P_2的优先级低。

由于软件和硬件方法都存在缺陷，故需要寻找其他机制。

3.4 信　号　量

现在寄希望于操作系统和程序设计语言提供对并发的支持。本节开始讨论信号量，后两节介绍管程和消息传递。

1965年，Dijkstra在关于并发进程的论文中论述了解决并发进程的主要好处，Dijkstra将操作系统看做是并发进程集，并描述了支持合作的可靠机制，如果处理机或操作系统提供这些机制，那么用户就能容易地使用它们。

基本的原则如下：两个或更多的进程可通过单一的信号量(Semaphore)展开合作，即进程在某一特定的地方停止执行直到得到一个特定的信号量。通过信号量，任何复杂的合作要求都可被满足。为了使用信号量s发送一个信号，进程要执行原语signal(s)。为了得到一个信号，进程要执行原语wait(s)。如果没有得到相应的信号，进程将被挂起直到得到信号为止。

为加深印象，可将信号量看做一个整型变量，并且定义以下3种操作：

① 信号量的初始化值非负。

② wait操作减小信号量的值，如果信号量的值为负，则执行wait操作的进程被阻塞。

③ signal操作增加信号量的值，如果信号量的值不为正数时，被wait操作阻塞的进程此时可以解除阻塞。

除了以上3种操作，再没有别的途径可以查看或操作信号量了。

图3.6给出了较为正式的信号量原语定义。wait和signal都是原子操作，它们不能被中断。图3.7给出了二元信号量(Binary Semaphore)的定义，它的定义比一般信号量的定义更加严格。一个二元信号量仅能取值为0或1。二元信号量的实现更为简单，而且可以证明它与一般信号量具有同等的表达能力。

```
type semaphore = record
                   count:integer;
                   queue: list of process
                 end;
var s: semaphore;
wait(s):
    s.count:=s.count–1;
    if s.count＜0
     then begin
       将该进程置入s.queue中;
       阻塞该进程
     end;
signal(s):
    s.count:=s.count+1;
    if s.count≤0
     then begin
       将进程P从s.queue中移出;
       将进程P置入就绪队列中
     end;
```

图 3.6　信号量原语的定义

```
type binary semaphore = record
                          value: (0,1);
                          queue: list of process
                        end;
var s:binary semaphore;
waitB(s):
   if s. value = 1
     then
       s.value=0
     else begin
          将该进程置入s.queue中;
          阻塞该进程
        end;
signalB(s):
    if s. queue为空
     then
       s.value:=1
     else begin
          将进程P从s.queue中移出;
          将进程P置入就绪队列中
        end;
```

图 3.7　二元信号量原语的定义

不论是采用一般信号量还是二元信号量，进程都将排队等候信号量，但这并不意味着进程移出的顺序与队列次序相同。一个最公平的规则是先进先出(FIFO)，阻塞时间最长的进程将最先从等待队列中移出。在其他进程拥有更高的优先级时，惟一的要求是进程不能在信号量队列中无限等待。

3.4.1　用信号量解决互斥问题

图3.8给出了一种使用信号量解决互斥问题的简明方法(与图3.2相比)。设用P(i)标识n个进程，wait(s)只在其临界段前执行。如果s的值为负数，则该进程被挂起。如果s的值为1，则其值减为0并且进程立即进入临界段，这时s的值不再为正数，所以其他进程就不能进入临界段。

```
program mutualexclusion;
const n=…; (* number of processes *);
var s:semaphore (: =1);
procedure P(i:integer);
begin
  repeat
    wait(s);
    〈critical section〉;
    signal(s);
    〈remainder〉
  forever
end;
begin (* main program *)
  parbegin
    P(1);
    P(2);
    …
    P(n);
  parend
end.
```

图 3.8　使用信号量实现互斥

信号量的初始值为1，因此第一个进程执行完wait操作后可以立即进入其临界段，并置s的值为0。其他任何欲进入临界段的进程将被阻塞并置s的值为–1，这类进程的每一次失败尝试都会使s的值减小。当最初进入临界段的进程离开时，s的值减小，并且一个被阻塞进程从阻塞于该信号量的队列中移出。在操作系统的下一次调度时，它将进入其临界段。

使用信号量的互斥算法也可以用小屋模型来描述。除了黑板外，小屋中还有一个大冰箱。某进程进入小屋后执行wait操作将黑板上的数减1，这时，如果黑板上的值非负，它就进入临界段，反之它就进入冰箱内冬眠。这时，就允许另一进程进入小屋。当一个进程完成其临界段后，它进入小屋执行signal，将黑板上的值加1，这时如果黑板上的值为非正数，它就从冰箱中唤醒一个进程。

对于在任一给定时刻允许超过一个进程进入其临界段的要求，图3.8所示的程序也可以很好地满足。在这种情况下，信号量将被初始化为一特定值，因此，在任一时刻，s.count的值可以解释如下。

- s.count≥0:　s.count是执行wait(s)操作而不会被阻塞的进程数
  (这期间没有执行任何signal(s)操作)
- s.count＜0:　s.count是阻塞在s.queue队列中的进程数

3.4.2　用信号量解决生产者/消费者问题

生产者/消费者问题是并发进程中最常见的问题。该问题可描述如下：一个或更多的生产者生产出某种类型的数据(记录、字符)，并把它们送入缓冲区，惟一的一个消费者一次从缓冲区中取走一个数据，系统要保证缓冲区操作不发生重叠，即在任一时刻只能有一方(生产者或消费者)访问缓冲区。下面用几种方法来描述信号量的能力和问题。

首先，假定缓冲区是无限大的线性数组，生产者和消费者函数的定义如下：

```
producer:                          consumer:
repeat                             repeat
  produce item v;                    while in≤out do {nothing};
  b[in]:= v;                         w:=b[out];
  in:= in+1                          out:=out+1;
forever.                             consume item w
                                   forever.
```

图3.9给出了缓冲区b的结构，生产者按自己的节拍生产数据并将其存入缓冲区。每做一次，缓冲区的索引in的值增加一次，消费者也以同样的方式工作，但消费者不会读空缓冲区，所以消费者在消费时要先确认缓冲区是否为空，若缓冲区为空，则消费者必须等待。

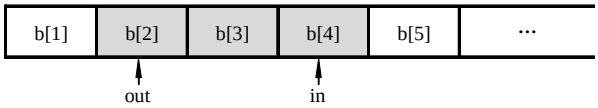


图 3.9　生产者/消费者问题的有限缓冲区

现在用二元信号量来解决此问题。图3.10所示的方法是首次尝试，这里可以简单地用整型变量n(=in–out)来记录缓冲区中的数据项数，这比in和out更为简明。信号量s用来实现互斥。在缓冲区为空时，用信号量delay强迫消费者等待。

```
program producerconsumer;
var     n:integer;
        s:(* binary *)semaphore (:=1);
        delay: (* binary *) semaphore (:=0);
procedure producer;
begin
  repeat
    produce;
    waitB(s);
    append;
    n:= n+1;
    if n=1 then signalB(delay);
    signalB(s)
  forever
end;
procedure consumer;
begin
  waitB(delay);
  repeat
    waitB(s);
    take;
    n:= n–1;
    signalB(s);
    consume;
    if n=0 then waitB(delay)
  forever
end;
begin (* main program *)
  n:= 0;
  parbegin
    producer; consumer
  parend
end.
```

图 3.10　解决生产者/消费者问题的一次尝试

这个方法简明易懂。在任何时候生产者都可向缓冲区中添加数据，在添加数据前，生产者执行waitB(s)，然后执行signalB(s)以防止在添加过程中，别的消费者或生产者访问缓冲区。在进入到临界段时，生产者将增加n的值，如果n=1则在此次添加数据前缓冲区为空，于是生产者执行signalB(delay)并将这个情况通知消费者。消费者最初执行waitB(delay)来等待生产者生产出第一个数据，然后取走数据并在临界段中减小n的值。如果生产者总保持在消费者前面，那么消费者就不会因为信号量delay而阻塞，因为n总是正数，这样生产者和消费者都能顺利地工作。

这个方法也存在缺陷。当消费者消耗空缓冲区时，它必须重置信号量delay，然后等待直到生产者再往缓冲区中添加数据为止。这就是语句**if** n=0 **then** waitB(delay)的用途。考虑表3.2所示的情况，在第6行消费者在执行waitB时发生意外，消费者的确已耗尽缓冲区并置n为0(见表3.2的第4行)，但是，由于生产者在第6行所示的消费者检测n之前已将n的值增加了，结果导致signalB没有与前面相对应的waitB；第9行中n的值为–1就意味着消费者消费了一个缓冲区中不存在的数据项，这并不仅仅是改变消费者临界段中相关条件的问题，因为这将导致死锁(例如第3行)。

表 3.2　图 3.10 程序中可能出现的情况

	动　　作	n	delay
1	初始化	0	0
2	Producer: critical section	1	1
3	Consumer: waitB(delay)	1	0
4	Consumer: critical section	0	0
5	Producer: critical section	1	1
6	Consumer: **if** n=0 **then** waitB(delay)	1	1
7	Consumer:critical section	0	1
8	Consumer: **if** n=0 **then** waitB(delay)	0	0
9	Consumer:critical section	–1	0

解决这个问题的一个方法是引入一个附加变量，如图3.11所示，它在消费者的临界段中设置。这样，就不会出现死锁了。

如图3.12所示，使用一般信号量可以得到另一种解决方法，变量n是一个信号量，它的值等于缓冲区中的数据项数。现在假设对程序进行改动，交换signal(s)和signal(n)的位置，这就使signal(n)在生产者的临界段中执行而不会被消费者或其他生产者中断。上述改动并不影响程序的正确性，因为在任何情况下，消费者在执行前都必须等候所有的信号量。

如果再交换wait(n)和wait(s)的次序，这就会产生一个致命的错误，当缓冲区为空时(n.count=0)，如果消费者进入到临界段中，那么生产者就不能向缓冲区中添加数据，从而系统陷入死锁。这个例子说明了使用信号量要注意细节以及进行正确设计的困难性。

```
program producerconsumer;
var     n:integer;
        s:(* binary *) semaphore (:=1);
        delay: (* binary *) semaphore (:=0);
procedure producer;
begin
  repeat
    produce;
    waitB(s);
    append;
    n:=n+1;
    if n=1 then signalB(delay);
    signalB(s)
  forever
end;
procedure consumer;
var m:integer; (* m为局部量*)
begin
  waitB(delay);
  repeat
    waitB(s);
    take;
    n:= n−1;
    m:= n;
    signalB(s);
    consume;
    if m=0 then waitB(delay)
  forever
end;
begin (* main program *)
  n:=0;
  parbegin
    producer; consumer
  parend
end.
```

图 3.11　解决生产者/消费者问题的一个正确方法

```
program producerconsumer;
var     n: integer (:=0);
        s: semaphore (:=1);
procedure producer;
begin
  repeat
    produce;
    wait(s);
    append;
    signal(s)
    signal(n)
  forever
end;
procedure consumer;
begin
  repeat
    wait(n);
    wait(s);
    take;
    signal(s);
    consume;
  forever
end;
begin (* main program *)
  parbegin
    producer; consumer
  parend
  end.
```

图 3.12　用信号量解决带有无限缓冲区的生产者/消费者问题

最后，给生产者/消费者问题添加一个新的也是现实的约束，即缓冲区有限。如图3.13所示将缓冲区看做一个环，并且指针的值必须对缓冲区的容量取模。

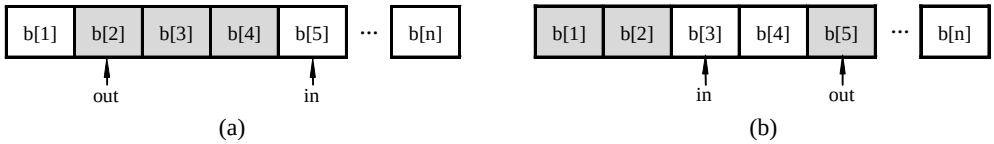


图 3.13　生产者/消费者问题的有限环形缓冲区

生产者/消费者函数定义如下(其中 in 和 out 的初始值为 0)：

```
producer
repeat
   produce item v;
      while((in+1) mod n=out) do {nothing};
         b[in]:=v;
         in=(in+1) mod n
forever;
consumer:
   repeat
      while in=out do {nothing};
         w:=b[out];
         out:=(out+1) mod n;
         consume item w
forever;
```

图3.14给出了使用一般信号量解决互斥的方法。

```
program boundedbuffer;
const       sizeofbuffer =…;
var         s:semaphore (:=1 );
            n: semaphore (:= 0);
            e:semaphore (:=sizeofbuffer);
procedure producer;
begin
  repeat
    produce;
    wait(e);
    wait(s);
    append;
    signal(s)
    signal(n)
  forever
end;
procedure consumer;
begin
  repeat
    wait(n);
    wait(s);
    take;
    signal(s);
    signal(e);
    consume
  forever
end;
begin (* main program *)
   parbegin
    producer; consumer
  parend
end.
```

图 3.14　使用信号量解决带有限缓冲区的生产者/消费者问题

3.4.3　信号量的实现

wait和signal操作都必须作为原子操作实现。显见，用硬件方法或固件方法都可解决这一问题，还有其他解决方法。该问题本质也是一种互斥。在任一时刻只有一个进程可用wait或signal操作信号量，因此，诸如Dekker算法和Peterson算法等都可以用，另外，还可用机器指令支持的互斥实现，例如，在图3.15(a)所示的方法中使用了test and set

指令，其信号量仍然是记录类型，只是增加了一个s.flag的分量。尽管wait和signal操作执行的时间较短，但因包含了忙碌-等待，故忙碌-等待占用的时间是主要的。

```
Wait(s):
    repeat{nothing}until testset(s.flag);
    s.count:=s.count–1;
    if s.count＜0
     then begin
       将该进程置入s.queue中;
       阻塞该进程(同时置s.flag为0)
      end
     else s.flag:=0;
signal(s):
repeat{nothing }until testset(s.flag);
    s.count:=s.count+1;
    if s.count≤0
     then begin
       将进程P从s.queue中移出;
       将进程P置入就绪队列
      end
    s.flag:=0;
```

(a)

```
Wait(s):
    屏蔽中断;
    s.count:=s.count–1;
    if s.count＜0
     then begin
       将该进程置入s.queue中;
       阻塞该进程并允许中断
      end
     else允许中断;
signal(s):
屏蔽中断;
    s.count:=s.count+1;
    if s.count≤0
     then begin
       将进程P从s.queue中移出;
       将进程P置入就绪队列
      end
    允许中断;
```

(b)

图 3.15　信号量的两种实现方法

(a) testset指令　(b) 中断

对单处理机系统而言，可以在wait和signal操作期间屏蔽中断，如图3.15(b)所示，而且这些操作的执行时间相对较短，所以这种方法是可行的。

3.4.4　用信号量解决理发店问题

理发店问题是使用信号量实现并发的另一个例子，它同操作系统中的实际问题非常相似。假设理发店有3个理发椅，3个理发师和1个可容纳4个人的等候室，此外还有1个供其他顾客休息的地方，如图3.16所示，理发店中的顾客总数不能超过20。顾客如果发现理发店中人已经满了(超过20)，就不进入理发店。在理发店内，顾客坐在沙发上，如果沙发坐满了，就站着。理发师一旦有空，就为在沙发上等候时间最长的顾客服务。这时，如果有站着的顾客，那么，站立时间最长的顾客就坐到沙发上。顾客理完发后，可向任何一个理发师付款，但理发店只有一本现金登记册，在任一时刻，

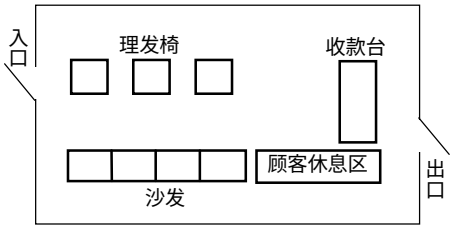


图 3.16　理发店

只能记录一个顾客的付款。理发师的时间就用在理发、收款和在理发椅上睡眠以等候顾客这几件事情上。

1. 一个不公平的理发店

图3.17为采用信号量解决理发店问题的一次尝试，这里假定所有的信号量队列都采用FIFO规则。

```
program barbershop1;
var     max-capacity: semaphore (:=20);
            sofa: semaphore (:=4);
            barber-char, coord:semaphore (:=3);
            cust-ready, finished, leave-b-chair, payment, receipt: semaphore (:=0);
procedure customer;
begin
  wait(max-capacity);
  enter shop;
  wait(sofa);
  sit on sofa;
  wait(barber-chair);
  get up from sofa;
  signal(sofa);
  sit in barber chair;
  signal(cust-ready);
  wait(finished);
  leave barber chair;
  signal(leave-b-chair);
  pay;
  signal(payment);
  wait(receipt);
  exit shop;
  signal(max-capacity)
end;
procedure barber;
begin
  repeat
    wait(cust-ready);
    wait(coord);
    cut hair;
    signal(coord);
    signal(finished);
    wait(leave-b-chair);
    signal(barber-chair);
  forever
end;
procedure cashier;
begin
  repeat
    wait(payment);
    wait(coord);
    accept pay;
    signal(coord);
    signal(receipt);
  forever
end;
begin (* main program *)
  parbegin
    customer; …50 times; …customer;
    barber; barber; barber;
    cashier
  parend
end.
```

图 3.17　一个不公平的理发店

程序中包含20名顾客、3名理发师和收款过程。以下讨论不同的信号量操作的用途和位置。

① 理发店和沙发容量：理发店和沙发容量分别由信号量max-capacity和sofa表示，每进一名顾客，信号量max-capacity就减1，每走一名顾客，就相应地加1，如果

顾客发现理发店已满，顾客进程就被wait函数挂起在信号量max-capacity上，同样wait和signal操作控制着顾客是坐进还是离开沙发。

② 理发椅的容量：这里有3个理发椅，1个椅子上只准坐1个人，信号量barber-chair保证了在任一时刻理发室内正在理发的人数不会超过3个人。

③ 确认顾客在理发椅上：信号量cust-ready用来唤醒睡眠中的理发师，让顾客坐到理发椅上。没有这个信号量，理发师将永远不会睡醒，因而，也不可能给顾客理发。

④ 控制顾客坐在理发椅上：顾客一旦坐进理发椅就不能离开直到理发师用信号量finished告诉他，已经理完发。

⑤ 限制一个顾客到一个理发椅上：信号量barber-chair用来限制坐在理发椅上的顾客不超过3个，但barber-chair本身并不能完全做到这一点，当理发师理完发和执行signal/finish时，下一个顾客进入理发室，但前一顾客可能因与别人闲聊还未离开理发椅，信号量leave-b-chair解决了这一问题，它禁止新顾客进来直到前一顾客离开。

⑥ 付款和收款：顾客执行signal(payment)表示已付款，然后执行wait(receipt)等待开发票。

⑦ 理发师和收款过程：理发店为省钱而不设专门的收款员。每个理发师在不理发时就执行收款的任务，信号量coord保证了在任一时刻理发师只干一项工作。

表3.3总结了该程序中每个信号量的用途。

表 3.3　图 3.17 程序中信号量的用途

信号量	wait操作	signal操作
max-capacity	顾客等待进入理发店	有顾客离开表示等待的顾客可进入理发店
sofa	顾客等待坐上沙发	有顾客离开沙发表示等待的顾客可坐沙发
barber-chair	顾客等待空的理发椅	表示理发椅为空
cust-ready	理发师等待顾客来理发	顾客通知理发师可开始理发
finished	顾客等待理完发	理发师告诉顾客头发已理完
leave-b-chair	理发师等待直到顾客离开理发椅	顾客告诉理发师他已离开理发椅
payment	收款员等待顾客付款	顾客告诉收款员他已付款
receipt	顾客等待收款发票	收款员表示已收款
coord	等待理发师去理发或收款	表示理发师空闲

2. 一个公平的理发店

图3.17所示的解决方法仍旧留下了一些问题。本节将解决其中的一个问题，其余的留作练习。

图3.17所示的方法中有一个定时不合理的问题，使得顾客之间出现不平等。假设有3个顾客正坐在3个理发椅上，其余顾客在wait(finished)上等待，如果其中1个理发师干得非常快或1个顾客几乎秃顶，若以他们为定时标准，则有的顾客还未理完发就被赶出理发店，反之，先理完发的顾客又不得不在理发椅上等待。

解决这个问题要用到更多的信号量。如图3.18所示，给每一名顾客一个独一无

```
program barbershop2;
var     max-capacity: semaphore (:=20);
        sofa: semaphore (:= 4);
        barber-char, coord:semaphore (:=3);
        mutex1,mutex2: semaphore (:=1);
        cust-ready, leave-b-chair, payment, receipt: semaphore (:=0);
        finished: array[1..20] of semaphore (:= 0);
        count: integer;

procedure customer;
var custnr: integer;
begin
  wait(max-capacity);
  enter shop;
  wait(mutex1);
  count:=count+1;
  custnr:=count;
  signal(mutex1);
  wait(sofa);
  sit on sofa;
  wait(barber-chair);
  get up from sofa;
  signal(sofa);
  sit in barber chair;
  wait(mutex2);
  enqueue1(custnr);
  signal(cust-ready);
  signal(mutex2);
  wait(finished[custnr]);
  leave barber chair;
  signal(leave-b-chair);
  pay;
  signal(payment);
  wait(receipt);
  exit shop;
  signal(max-capacity)
end;

    procedure barber;
    var b-cust: integer;
    begin
  repeat
    wait(cust-ready);
    wait(mutex2);
    dequeuel(b-cust);
    signal(mutex2);
    wait(coord);
    cut hair;
    signal(coord);
    signal(finished[b-cust]);
    wait(leave-b-chair);
    signal(barber-chair);
  forever
end;

        procedure cashier;
begin
  repeat
    wait(payment);
    wait(coord);
    accept pay;
    signal(coord);
    signal(receipt);
  forever
end;

begin (* main program *)
  count:=0;
  parbegin
    customer; … 20 times; …customer;
    barber; barber; barber;
    cashier
  parend
end.
```

图 3.18　一个公平的理发店

二的顾客号，信号量mutexl保护着全局变量count以使顾客能取到惟一的一个顾客号。信号量finished重新定义为一个含20个信号量的数组。一旦顾客坐进理发椅，他就执行wait(finished[custnr])以等待各自的信号量。如果理发师已为顾客理完发，理发师就执行signal(finished[b-cust])来放走这位顾客。

理发师是如何知道顾客号的？实际上，顾客在用信号量 cust-ready 通知理发师之前已将其号码放到队列 enqueuel 中，当理发师准备理发时，执行 dequeuel(b-cust) 将队列头存放的号码放到了理发师的局部变量 b-cust 中。

3.5 管　　程

信号量为实现互斥和进程间的协调提供了一个功能强大而灵活的工具，然而，使用信号量来编制正确的程序是困难的。其困难在于wait和signal操作可能遍布整个程序并且很难看出它们所引起的全部后果。

作为程序设计语言中的一种结构，管程(Monitor)提供了与信号量相同的表达能力，但它更容易控制，Hoare中给出了它的第一个正式定义。许多程序设计语言中都已包含了管程结构，包括并发pascal、pascal-plus、modula-2和modula-3。

3.5.1 带信号量的管程

管程是一种软件模块，它包括一个或多个过程、初始化语句和局部数据。管程最主要的特点如下：

① 只能通过管程中的过程而不能用其他外部过程访问其局部数据变量。

② 进程通过调用管程的过程而进入管程。

③ 每一时刻只能有一个进程在管程中执行，任何其他调用管程的进程将被挂起直至管程可用为止。

前两个特点同对象在面向对象软件中的特性一样。事实上面向对象的操作系统或程序设计语言都可将管程看作是具有特殊性质的对象加以实现。

管程在任一时刻只允许一个进程调用，这样就具有实现互斥的能力，任一时刻管程中的局部数据也只能被一个进程访问。因此，一个共享的数据结构可以放在管程中加以保护，如果管程中的数据代表某种资源，那么，管程就支持对这种资源访问的互斥。

为了将管程用于并发进程中，管程必须拥有同步手段。管程可以使用条件变量支持同步，这些变量保存在管程中并且只能在管程内部访问。用以下两个函数操作条件变量。

① cwait(c)：调用进程在条件c上挂起，管程现在可被其他进程使用。

② csignal(c)：在条件c上被cwait挂起的进程被再次执行。如果有这样的进程，就选择其中一个执行，否则就什么都不做。

注意：管程的wait/signal操作与信号量中的操作不同，如果在管程中的进程发信号并且没有一个任务等待条件变量，这个信号就丢失了。

图3.19给出了管程的结构，尽管进程可调用任一进程而进入管程，仍可将管程看做是单入口的，这样就保证了在任一时刻只有一个进程在管程中，其他欲进入管程的进程在等待管程的队列中挂起。一旦进程进入管程，就可以执行cwait(x)，在条件x上暂时挂起，然后被放入一个队列，等到条件改变后再进入管程。

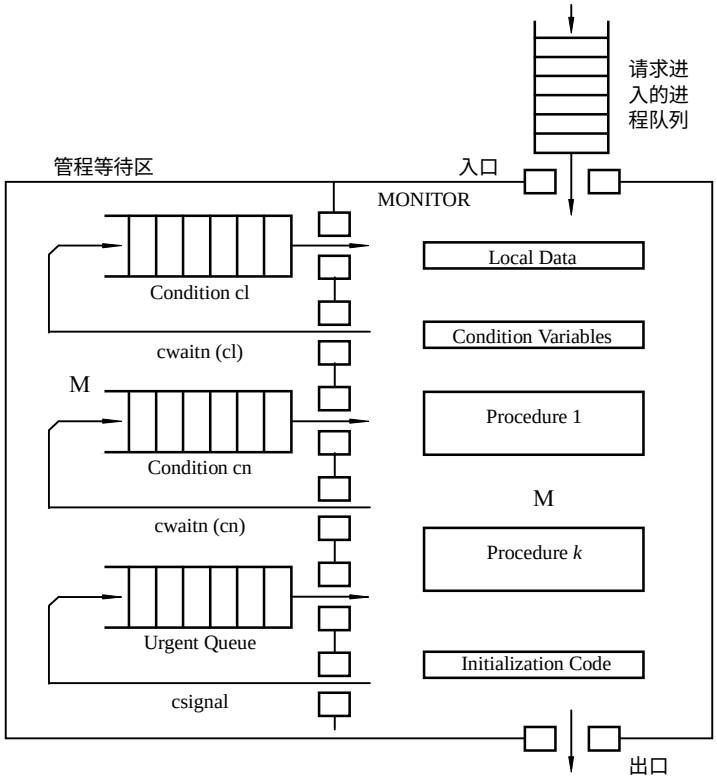


图 3.19　管理的结构

如果一个在管程中执行的进程检测到条件变量x已发生改变，就执行csignal(x)，通知相应的进程条件已改变。

3.5.2　用管程解决生产者/消费者问题

现在回过头来看看有限缓冲区生产者/消费者问题，图3.20给出了使用管程的解决方法。管程模块控制着缓冲区。管程包括两个条件变量：当缓冲区中还有空位置

时，变量notfull为true，如果缓冲区中至少有一个字符存在，则变量notempty为true。

```
program producerconsumer
  monitor boundedbuffer;
    buffer: array [0..N] of char;                {N个数据项大小的缓冲区}
    nextin, nextout: integer;                    {缓冲区指针}
    count: integer;                              {缓冲区中数据项的个数}
    notfull, notempty: condition;                {用于同步}
  var i:integer;
  procedure append (x:char);
    begin
      if count=N then cwait(notfull);            {缓冲区满；避免上溢}
      buffer[nextin]:=x;
      nextin:=nextin+1 mod N;
      count:=count+1;                            {数据项数加1}
      csignal(notempty);                         {唤醒等待的消费者}
    end;
  procedure take (x:char);
    begin
      if count=0 then cwait(notempty);           {缓冲区空；避免下溢}
      x:=buffer[nextout];
      nextout:=nextout+1 mod N;
      count:=count–1;                            {数据项数减1}
      csignal(notfull);                          {唤醒等待的生产者}
    end;
  begin
      nextin:=0; nextout:=0; count:=0            {缓冲区初始化}
   end;
procedure producer;
var x:char;
begin
  repeat
    produce(x);
    append(x);
  forever
end;
procedure consumer;
var x:char;
begin
  repeat
    take(x);
    consume(x);
  forever
end;
begin (*main program*)
  parbegin
    producer; consumer
  parend
end.
```

图 3.20　使用管程解决带有有限缓冲区的生产者/消费者问题

生产者只能通过管程中的过程append向缓冲区添加字符，生产者不能直接访问缓冲区。生产者首先检查条件notfull来确定缓冲区中是否有空位置，如果没有进程就在这个条件上挂起。其他进程(生产者或消费者)就可以进入管程。当缓冲区出现空位置时，进程就从挂起队列中移出并继续执行。在缓冲区放入字符后，进程修改条件变量notempty。消费者函数也可同样表述。

这个例子比较了管程和信号量。对于管程而言，管程本身的结构就实现了互斥，生产者和消费者同时访问缓冲区是不可能的，然而为了防止进程向已满的缓冲区存数据或从空缓冲区中取数据，程序设计者必须在管程中使用类似于cwait和csignal操作。对于信号量而言，实现互斥和同步都是程序设计者的责任。

3.6　消息传递

进程间的相互作用必须满足两个条件：同步和通信，进程需要同步来实现互斥，进程间的协同需要交换信息，一个能解决上述两个问题的方法是消息传递(message passing)。消息传递还有一个优点是，它能在分布式系统、共享存储器的多处理机系统以及单处理机系统中实现。

3.6.1　消息传递原语

消息传递系统设计所涉及的主要内容见表3.4。实际的消息传递函数通常是成对的原语：

- send (destination, message)
- receive (source, message)

表 3.4　消息传递系统设计所涉及的主要内容

通信和同步		
同步	**寻址**	**消息格式**
发送	直接	内容
阻塞	发送	长度
无阻塞	接收	固定
接收	明确	可变
阻塞	隐含	**排队规则**
无阻塞	间接	先进先出
测试到达	静态	优先权
	动态	
	所有权	

以上是进程进行消息传递所需的最小集，进程根据目的地址(destination)以消息

的形式向另一进程传递信息，进程通过执行receive原语接收信息，其中包括发送进程的源地址和消息。

3.6.2　用消息传递实现同步

两个进程间的消息通信就意味着其中包含某种程度的同步，这就是接收方在发送方发送信息之前，接收不到任何消息。下面讨论进程在执行发送和接收原语后的状态。

首先考虑send原语。进程在执行send原语时有两种可能：一种是发送进程被阻塞直到消息被接收，另一种是发送进程不被阻塞。

进程发出receive原语后也有两种可能：

① 如果消息已发送来，则消息被接收并继续执行；

② 如果没有发送来的消息，则进程或者被阻塞以等待消息或者放弃接收。

所以，无论是发送方还是接收方都可能被阻塞。下面有三种最一般的组合，任何特定系统都实现了其中的一种或两种：

① 阻塞发送，阻塞接收。发送方和接收方都将被阻塞直到传递完消息，这种组合要求在进程间有严格的同步。

② 无阻塞发送，阻塞接收。发送方可持续发送消息，而接收方被阻塞直到接收到消息，这是最有用的一种组合，它允许发送进程以尽可能快的速度向多个目的地发送消息，接收进程在消息到达前将被阻塞。

③ 无阻塞发送，无阻塞接收。无论哪一方都不需要等待。

对许多并发任务而言，无阻塞发送是最自然不过的，例如，用此方法向打印机发请求，它允许请求进程不间断地向打印机发出请求。无阻塞发送的一个潜在危险是一些错误可能导致进程重复地产生消息，因为没有阻塞来约束该进程，这些消息将耗费大量的系统资源，包括处理机时间和缓冲区空间。另一方面，无阻塞发送加重了程序员确认消息是否收到的负担，进程必须发送应答消息确认已收到消息。

对于receive原语，无阻塞的形式看起来是并发任务的自然选择。一般说来，接收进程在继续执行前需要得到它所希望的消息，但是，如果一个消息丢失或发送进程在发送消息时发生意外，则接收进程将被阻塞，这个问题可以采用无阻塞的receive原语解决。这种方法的危险在于，当一个进程刚接收完一个消息后，另一消息又送到，后一个消息将丢失。其他的方法允许进程在发出receive原语前检测是否有正在等待的消息，还可以在一个receive原语中标明多个发送方。

3.6.3　寻址方式

显然，发送原语必须标明哪一个进程将接收消息。同样，接收进程要通知发送

进程消息已收到。标识进程的方法可分为两类：直接寻址和间接寻址。

1. 直接寻址

对直接寻址方式，send原语中明确标明了目的进程。处理receive原语有两种方式：第一种方式接收进程预先知道发送消息的源进程，这对并发进程的协同是有效的；另一种方式则不可能预先知道源进程，例如，打印服务进程，它可从任何其他进程接收请求信息，对这种应用更有效的方法是隐含寻址。

2. 间接寻址

对间接寻址来说，消息并不是直接由发送方传给接收方，而是通过一个共享的数据结构，包括临时存放消息的队列，这种队列通常称为邮箱式端口。两个进程进行通信时，一个进程向邮箱式端口发送消息，而另一个进程则从邮箱式端口中取回消息，如图3.21所示。

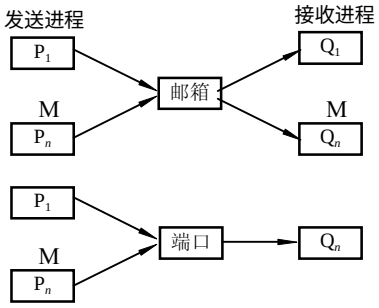


图 3.21　间接的进程通信

这种方法有很大的灵活性。发送方和接收方的关系可以是一对一、多对一、一对多或者多对多。一对一关系允许在两个进程间建立私人信箱，这就将他们的相互作用同其他进程隔离开。多对一的关系允许有多个发送方和一个接收方，可用于客户/服务器模式。一对多关系允许有一个发送方和多个接收方，这对于广播应用是有益的。

进程与邮箱的关系可以是静态的或动态的，端口通常与一个特定的进程相关联，一个一对一关系常定义为静态的和永久的。当有多个发送方时，发送方与邮箱的联系就是动态的，这就要用到connect和disconnect原语。

端口通常由接收进程产生并为其所有，所以，当进程受损时，端口也要遭到破坏。一般的邮箱则由操作系统生成，当邮箱随进程的完成而取消时，邮箱可看作属于产生它们的进程。当邮箱要用显示命令取消时，邮箱的所有权就属于操作系统。

3.6.4　消息格式

消息格式依赖于消息系统的用途和消息系统是在单个计算机上运行还是在分布式系统中运行，一些操作系统选择了较短的定长消息以减小处理和存储的开销，如果有大量的数据需要传递，那么可以将数据存入文件并把文件作为消息传递，一种更为灵活的方法是允许变长消息。

图3.22给出了支持变长消息的操作系统中的消息格式，消息分为两部分：消息头用来保存有关消息的信息，消息体保存着实际的消息内容。消息头中包含有源进程的

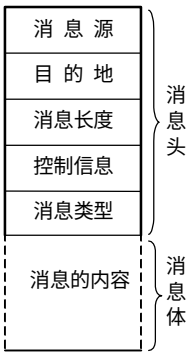


图 3.22　典型的消息格式

标识，消息的目的地址，和用于描述不同类型消息的长度域和类型域，还可以有其他的控制信息。例如，用于创建消息链表的指针域、记录消息发送次序的序列号以及优先级域等等。

3.6.5　排队规则

最简单的排队规则是先进先出，但这远远不够。例如，一些消息比另一些消息更紧迫。一个改进的方法是由发送方或接收方基于消息的类型标明消息的优先权；另一个改进方法是允许接收方检查消息队列并选择要接收的消息。

3.6.6　用消息传递实现互斥

图3.23给出了使用消息传递实现互斥的一种方法。假设使用阻塞receive原语和无阻塞send原语，并发进程集共享一个邮箱mutex，将邮箱初始化为仅包含一个空消息，欲进入临界段的进程首先要接收相应的消息，如果邮箱为空则该进程被阻塞，一旦进程得到消息它就执行其临界段，然后再将消息放回邮箱，这样消息就如同令牌一样在进程之间传递。这种解决方法是，假设有多于一个进程并发执行receive操作，则有：

- 如果仅有一个消息，那么它只可传递给一个进程，其他进程将被阻塞。
- 如果邮箱为空，则所有的进程将被阻塞。当消息可用时，仅有一个阻塞进程被激活并得到消息。

几乎在所有的消息传递中都有这些假设。

```
program mutualexclusion;
const n= …; (*进程个数*);
procedure P(i：integer);
var msg:message;
begin
  repeat
    receive (mutex, msg);
    〈critical section〉;
    send (mutex, msg);
    〈remainder〉
  forever
end;
begin (*main program*)
  create-mailbox(mutex);
  send (mutex,null);
  parbegin
    P(1);
    P(2);
    …
    P(n);
  parend
end.
```

图 3.23　使用消息实现互斥

图3.24给出了有限缓冲区生产者/消息者问题的一种解决方法，它是应用消息传递的又一个例子。利用与图3.24相似的算法并使用消息传递所支持的互斥就可以解决这一问题。实际上，图3.24还利用了信号量不具备的传递数据的能力。这里使用了两个邮箱。生产者将生产出的数据作为消息送到邮箱mayconsume，只要邮箱中有消息，消费者就可以消费。mayconsume就用做缓冲区，这个缓冲区中的数据组织成

消息队列，缓冲区的大小由全局变量capacity确定。最初，邮箱mayproduce充满了与缓冲区容量相同数量的空消息，mayproduce中的消息随生产者生产而减少，随消费者消费而增加。

```
const
  capacity=…;  {缓冲区大小}
  null=…;      {空消息}
var i:integer;
procedure producer;
  var pmsg: message;
    begin
      while true do
        begin
          receive (mayproduce, psmg);
          pmsg:=produce;
          send (mayconsume, pmsg)
        end
    end;
procedure consumer;
  var csmg: message;
  begin
    while true do
      begin
        receive (mayconsume, cmsg);
        consume (csmg);
        send (mayproduce, null)
      end
  end;
{父进程}
begin
  create mailbox (mayproduce);
  create mailbox (mayconsume);
  for i=1 to capacity do send (mayproduce, null);
  parbegin
    producer;
    consumer
  parend
  end.
```

图 3.24　使用消息解决带有界缓冲区的生产者/消费者问题

这个方法非常灵活，可以允许有多个生产者和消费者，系统还可以是分布式的，在这种分布式环境中，所有的生产者进程和邮箱 mayproduce 在一个节点上，所有的消费者进程和邮箱 mayconsume 在另一个节点上。

3.7　读者/写者问题

为做好同步和并发的设计工作，需要将现实问题同经典问题联系起来，并根据其解决经典问题的能力来测试这些设计，有些问题是设计中的共同问题并含有极高的教学价值，这些问题比较常见而且重要，其中有已讨论过的生产者/消费者问题；还有将要介绍的另一经典问题：读者/写者(readers/writers)问题。

readers/writers问题的定义如下：一些进程共享一个数据区。数据区可以是一个文件、一块内存空间或一组寄存器。readers进程只能读数据区中的数据，而writers进程只能写。

解决该问题必须满足的条件如下：

① 任意多个readers可以同时读；

② 任一时刻只能有一个writers可以写；

③ 如果writers正在写，那么readers就不能读。

首先从两个方面认清这个问题：一般的互斥问题和生产者/消费者问题。在readers/writers问题中，readers不能写数据，writers不能读数据。更一般的问题是，允许任何一个进程都可读或写数据。在这种情况下，可以将进程访问数据区的任何部分作为临界段，然后使用一般的解决互斥的方法。可以找到比解决一般问题的方法更为高效的解决办法。对于大部分问题，一般的方法速度太慢以至不可容忍，例如，假设共享数据区是图书馆的目录，一般读者可以查阅目录以找到所需要的书，一些图书管理员可以修改目录。对于一般解法，每一次对目录的访问都将当作临界段，在任一时刻只能有一个用户查阅目录，这显然是不能容忍的，同时还要防止writers互相干扰，而且在写的过程中要禁止读。

能否将生产者/消费者问题简单地看做是readers/writers问题在仅有一个writer和reader时的特例呢？答案是否定的，生产者并不仅仅是一个writer，它还要读队列指针以判断在哪里放置数据项，还要判断缓冲区是否已满，同样，消费者也不仅仅是reader，因为它还必须移动指针以表示它已从缓冲区中取走了一个数据。

下面讨论解决这个问题的两种方法。

3.7.1　读者优先

图3.25给出了使用信号量的解决方法，信号量wsem用来实现互斥，只要有writer在访问共享数据区，其他writers或readers就不能访问数据区。reader进程同样利用wsem实现互斥。为了适用于多个reader，当没有reader读时，第一个进行

```
program readersandwriters;
var readcount: integer;
    x,wsem: semaphore (:=1);
procedure reader;
begin
  repeat
    wait(x);
      readcount:=readcount+1;
      if readcount=1 then wait(wsem);
    signal(x);
    READUNIT;
    wait(x);
      readcount:=readcount–1;
      if readcount=0 then signal(wsem);
      signal(x)
  forever
end;
procedure writer;
begin
  repeat
    wait(wsem);
      WRITEUNIT;
    signal(wsem)
  forever
end;
begin
  readcount:=0;
  parbegin
    reader;
    writer
  parend
end.
```

图 3.25　使用信号量解决读者/写者问题的方法(读者优先)

读的reader要测试wsem，当已经有reader在读时，随后的reader无须等待就可读取数据区，全局变量readcount用来记录reader的数目，信号量x用于保证readcount修改的正确性。

3.7.2　写者优先

在前一种解法中，readers优先，一旦有一个reader访问数据区，只要还有一个reader在进行读操作，reader就可以保持对数据区的控制，这就容易导致writers饥饿。

图3.26给出了writers优先的解决方法。只要有writer申请写操作，就不允许新的readers 访问数据区。在以前定义基础上，writers增加了以下的信号量和变量。

① 信号量rsem用于在有writer访问数据区时，屏蔽所有的readers；

② 变量writercount控制rsem的设置；

③ 信号量y控制writecount的修改。

```
program readersandwriters;
var readcount,writecount: integer;
    x,y,z,wsem,rsem: semaphore (:=1);
procedure reader;
begin
  repeat
    wait(z);
      wait(rsem);
        wait(x);
          readcount:=readcount+1;
          if readcount=1 then wait(wsem);
        signal(x);
      signal(rsem);
    signal(z);
    READUNIT;
    wait(x);
      readcount:=readcount−1;
      if readcount=0 then signal(wsem);
    signal(x)
  forever
end;
procedure writer;
begin
  repeat
    wait(y);
      writecount:=writecount+1;
      if writecount=1 then wait(rsem);
    signal(y);
    wait(wsem);
      WRITEUNIT;
    signal(wsem);
    wait(y);
      writecount:=writecount−1;
      if writecount=1 then wait(rsem);
    signal(y);
  forever
end;
begin
  readcount, writecount:=0;
  parbegin
    reader;
    writer
  parend
end.
```

图 3.26　使用信号量解决读者/写者的方法(写者优先)

readers也必须添加一个信号量，在信号量rsem上不允许有较长的排队等待，要不然，writers就不能跳过这个队列，所以，只允许一个reader在rsem上排队，其余的

readers在等待rsem之前先在另一信号量z上排队。表3.5总结了各种可能性。

表 3.5　图 3.26 中进程队列的状态

状　　态	操　　作
系统中只有readers	• 置wsem　• 没有排队
系统中只有writers	• 置wsem和rsem　• writers在wsem上排队
存在readers和writers且readers优先	• reader设置wsem　• writer设置rsem　• 所有的writers在wsem上排队 • 只有一个reader在rsem上排队　• 其余readers在z上排队
存在readers和writers且writers优先	• writer设置wsem　• writer设置rsem　• writers在wsem上排队 • 只有一个reader在rsem上排队　• 其余readers在y上排队

另一种解决方法是使用消息传递，如图3.27所示。这个方法基于Theaker和Brookes论文中的算法。有一个可访问共享数据区的控制进程，其他欲访问数据区的进程向控制进程发请求消息，收到“OK”应答消息后就可访问，在访问完成后发出“finished”消息，控制进程拥有3个信箱，每个信箱装一种类型的消息。

```
procedure readeri;
  var rmsg: message;
  begin
    repeat
      rmsg:=i;
      send(readrequest, rmsg);
      receive(mboxi, rmsg);
      READUNIT;
      rmsg:=i;
      send(finished, rmsg)
    forever
  end;
procedure writerj;
  var rmsg: message;
  begin
    repeat
      rmsg:=i;
      send(writerequest,rmsg);
      receive(mboxj,rmsg);
      WRITEUNIT;
      rmsg:=i;
      send(finished,rmsg)
    forever
end;
```

```
procedure controller;
  begin
    repeat
      if count＞0 do begin
        if not empty(finished) then begin
          receive(finished,msg);
          count:=count+1
          end
        else if not empty(writerequest) then begin
          receive(writerequest,msg);
          writer.id:=msg.id;
          count:=count–100
          end
        else if not empty(readrequest) then begin
          receive(readrequest,msg);
          count:=count–1
          send(msg.id,"OK")
          end
      end;
      if count=0 do begin
        send(writer.id,"OK");
        receive(finished,msg);
        count:=100
      end;
      while count＜0 do begin
        receive(finished,msg);
        count:=count+1
      end
    forever
  end.
```

图 3.27　使用消息传递解决读者/写者问题

控制进程通过在读请求之前先为写请求服务将优先权给writers，另外为实现互斥，要使用变量count并将其初始化为比最大可能的readers数还大的数，这里取值为100。

控制进程的行为如下：

① 如果count＞0，则没有正在等待的writer，可能有也可能没有已激活的reader。先为所有的“finished”消息服务，以清除活跃的reader，然后依次为写请求和读请求服务。

② 如果count=0，则惟一的请求代表一个写请求，允许进行写操作并等待“finished”消息。

③ 如果count＜0，则一个writer已发出请求并正在等待清除所有的活跃readers，所以只为“finished”消息服务。

3.8 小　结

现代操作系统的中心问题是多道程序、多进程和分布式进程，并发是这些问题的基础，同时也是操作系统设计的基础，当多个进程并发执行时，无论是在多处理机系统还是在单处理机系统中都会出现解决冲突和进程间协同的问题。

并发进程可通过多种方式相互作用，互相透明的进程要为使用资源展开竞争。这些资源包括处理机时间、I/O设备的访问权等。当共享某一对象时，进程就间接地觉察到对方的存在，例如，共享一块内存或一个文件，进程也可直接认识对方并通过交换信息进行协同，在这些相互作用中的主要问题是互斥和死锁。

对于并发进程，互斥是一个必要的条件，在任一时刻，只能有一个进程可以访问某一给定的资源或执行某一给定的函数，互斥可用来解决一些冲突，例如，竞争资源；也可用于进程同步以使它们能协同工作，例如，生产者/消费者模型，一个进程向缓冲区添加数据，另一些进程则从缓冲区取走数据。

现在已有了不少解决互斥问题的算法，其中最著名的是Dekker算法。软件方法开销较大，出错的几率高；另一种方法是使用特殊的机器指令解决互斥问题，这种方法虽然减小了开销，但仍有不足，因为它用到了忙碌-等待技术。

还有一种解决互斥的方法是在操作系统内部提供支持，两个最常用的技术是信号量和消息传递。信号量和消息传递都能方便地实现互斥，而消息传递还可用于进程间通信。

习　题　三

3.1　一个并发程序拥有两个进程P和Q，其中A，B，C，D和E是原语：

```
procedure P;
begin
  A;
  B;
  C;
end.
```

```
procedure Q;
begin
  D;
  E;
end.
```

试给出该程序在并发执行时所有可能的执行路径(参照图3.1)。

3.2　是否在任何情况下忙碌-等待都比阻塞等待方法的效率低？试解释之。

3.3　以下程序并发执行，且数据只有在调入寄存器后才能加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;
  parbegin
    total;total
  parend;
  writeln(tally);
end.
```

(1) 给出该并发程序输出的tally值的下限和上限。

(2) 假定并发执行的进程数为任意个，(1)中的结果会作何变化？

3.4　证明Dekker算法的正确性：

(1) 证明它可以实现互斥。

(2) 证明它可以避免死锁。

3.5　面包店算法(bakery algorithm)如下，它是解决互斥问题的另一软件方法。

```
var choosing: array [0..n–1] of boolean;
    number: array [0..n–1] of integer;
repeat
  choosing[i]=true;
  number[i]:=1+max(number[0],number[1],…，number[n–1]);
  choosing[i]:=false;
for j:=0 to n–1 do
```

```
    begin
      while choosing[j] do {nothing};
      while number[j]≠0 and (number[j],j)＜(number[i],i) do {nothing};
  end;
    〈critical section〉;
     number[i]:=0
    〈remaider〉;
  forever.
```

数组choosing和number的初始值分别为false和0，每个数组中的第*i*个元素只能由进程*i*进行读/写，而其他进程只能读。(a,b)＜(c,d)的定义为：(a＜c) **or** (a=c **and** b＜d)

(1) 用自然语言描述该算法。

(2) 该算法是否能避免死锁，为什么？

(3) 该算法是否能实现互斥，为什么？

3.6　证明以下实现互斥的软件方法并不依赖于存储器访问一级的互斥执行。

(1) bakery算法(面包店算法)；

(2) peterson算法。

3.7　在图3.14的程序中将以下语句进行交换会产生什么后果？

(1) wait(e); wait(s)

(2) signal(s); signal(n)

(3) wait(n); wait(s)

(4) signal(s); signal(e)

3.8　在本章的生产者/消费者问题中，我们只允许缓冲区中最多有*n*–1个数据项。

(1) 为什么要这个限制？

(2) 修改该算法以消除该限制。

3.9　回答以下与公平的理发店问题相关的问题(见图3.18)：

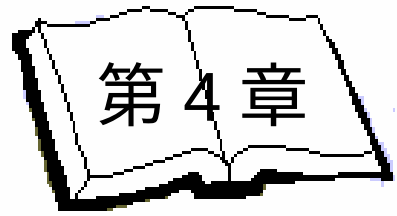
(1) 理发师是否必须亲自向其服务的顾客收款？

(2) 理发师是否一直使用同一理发椅？

3.10　证明管程与信号量的功能等价：

(1) 用信号量实现管程。

(2) 用管程实现信号量。



死锁处理

本章讨论支持并发进程的两个问题：死锁(Deadlock)和饥饿(Starvation)。首先讨论死锁和饥饿的基本概念，然后讨论解决死锁的 3 种途径：死锁预防、死锁检测和死锁避免，最后讨论同步和死锁的经典问题——哲学家用餐问题的解决方法。

4.1 死锁概述

死锁是多个进程为竞争系统资源或彼此间通信而引起的永久性的阻塞现象。不像并发进程管理中的其他问题，死锁没有一个通行的解决方法。

所有的死锁都涉及到两个或更多的进程由于竞争资源引起的冲突。图4.1以抽象的形式描述了两个进程和两个资源之间的冲突。图中的纵横坐标轴分别代表两个进程的执行情况，水平虚线或垂直虚线代表了惟一的一个正在执行的进程占用的时间段。假设进程在执行过程中的某一时刻需要与其他进程互斥地使用R_1和R_2。例如，在某一时刻进程P_1占用了资源R_1，进程P_2占用了资源R_2，而且每个进程都请求得到另一资源，这时就出现死锁。

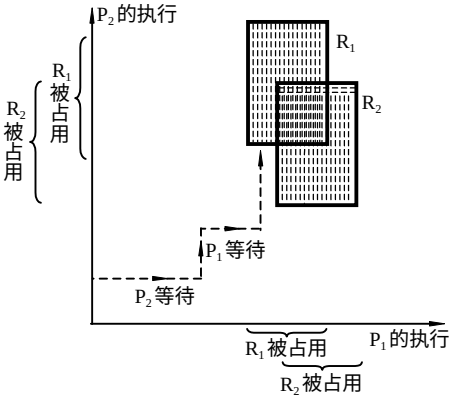


图 4.1 进程 P_1 和 P_2 的执行

4.1.1 可重用资源

资源可分为两大类：可重用资源和消耗型资源。可重用资源在任一时刻只能由一个进程使用而且使用后的资源完好如初，可被重新使用。可重用资源包括处理机、

I/O通道、存储器、设备和文件、数据库、信号量等数据结构。

下面是一个使用可重用资源而发生死锁的例子。两个进程P_1和P_2竞争必须互斥访问的磁盘文件D和磁带机T，程序重复地执行以下操作：

P_1	P_2
repeat	**repeat**
…	…
Request(D);	Request(T);
…	…
Request(T);	Request(D);
…	…
Release(T);	Release(D);
…	…
Release(D);	Release(T);
…	…
forever	**forever**

如果每个进程都占有一个资源并请求得到其他资源就会发生死锁，表面看起来这是程序错误而不是操作系统设计的问题，然而并发程序设计极具挑战性，其复杂的程序逻辑将产生死锁问题，解决死锁的一个方法是在系统设计中给资源定序。

另一个死锁的例子与内存分配相关，假设系统的可用空间为200KB，并有以下请求序列：

P_1	P_2
…	…
申请80KB	申请70KB
…	…
申请60KB	申请80KB

当两个进程都执行到它们的第二个申请时就发生死锁，如果事先不知道要申请的内存空间数量，而要通过在系统设计时添加限制的方法解决这类问题则是困难的。事实上解决这个问题最好的方法是使用虚拟存储器。

4.1.2　消耗型资源

消耗型资源是可以生产和消耗掉的资源，通常对某一特定类型的消耗型资源并没有数量的限制。当资源被进程占用后，它就不存在了。消耗型资源包括中断、消息、I/O缓冲区中的信息等。

下面是使用消耗型资源而发生死锁的例子：

P_1	P_2
…	…
Receive(P_2,M);	Receive(P_1,Q);
…	…
Send(P_2,N);	Send(P_1,R);

如果Receive阻塞就会发生死锁。这里，同样是由设计错误导致的死锁，这样的错误可能细小并且难于检测。实际应用中导致死锁的事件组合很少出现，所以一个程序在运行相当长的时间甚至几年，问题才会出现。

解决各种类型的死锁并没有一个通行的有效方法。表4.1总结了解决死锁方法的关键部分：死锁预防、死锁检测和死锁避免。

表 4.1　解决死锁的方法

解决方法	资源分配策略	不同的方案	主要优点	主要缺点
死锁预防	保守，对资源严加限制	一次申请所有的资源	• 在进程的活动较为单一时性能较好 • 无须抢占	• 效率低 • 启动进程慢
		抢占	• 对那些状态易于保存和恢复的资源较为方便	• 导致过多的不必要的抢占 • 容易导致循环重启
		资源定序	• 易于实现在编译期间的检查 • 无须执行时间，在系统设计阶段问题就已解决	• 无用抢占 • 不允许增加资源请求
死锁检测	非常自由，只要有可能就会满足资源请求	周期地执行以检测死锁	• 不会推迟进程的启动 • 在线处理比较便利	• 丢失了固有的抢占
死锁避免	介于死锁预防和死锁检测之间	寻找到至少一个安全路径	• 无须抢占	• 必须知道下一步的资源请求 • 进程可能被长时间阻塞

4.1.3　产生死锁的条件

产生死锁有4个必要条件：

① 互斥。任一时刻只能有一个进程使用某一资源。

② 占用并等待。进程占用部分资源后等待分配其他的资源。

③ 非抢占。资源不能从占用它的进程中夺走。

然而，在许多情况下又非常需要以上条件。例如，为保证结果的一致性和数据库的完整性就需要互斥，一般情况下资源不能随意被抢占特别是数据资源。抢占还必须有撤回恢复机制的支持，以使进程能回到原始状态并重新执行。

④ 循环等待。在一个进程链中，每个进程至少占有一个其他进程所必需的资源，如图4.2所示。

前3个条件是导致死锁的必要而非充分条件。事实上前3个条件也潜藏着第4个条件。如果存在前3个条件，就可能有一系列事件形成不可解除的循环等待。这种不可解除的循环等待事实上就是死锁。由于前3个条件使得这种循环链不可解开。

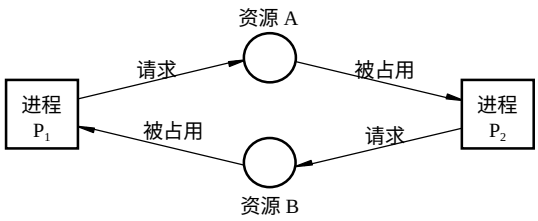


图 4.2　循环等待

4.2　死锁处理

4.2.1　死锁预防

死锁预防就是预先防止死锁发生的可能性。预防死锁的方法可分为两类：间接方法和直接方法。间接方法是防止出现产生死锁的前3个条件，直接方法是防止第4个条件即循环等待的出现。下面分别讨论与这4个条件相关的技术。

1. 互斥

一些资源例如文件允许多个读访问，只有写操作是互斥执行的。即使在这种情况下如果有多个进程要进行写操作，也将会产生死锁。可用第3章介绍的解决互斥问题的思想和技术来解决有关由于互斥条件不满足而产生的死锁问题。

2. 占用并等待

为了破坏占用并等待条件，进程在执行前请求分配其所需的资源，并且阻塞该进程直至所有请求同时满足。这个方法有两点不足，首先进程为等待满足所有的资源需要等待很长时间，而事实上只需其中一部分资源进程就可执行；其次分配给一个进程的资源可能在相当长的时间内不被使用，而在此期间其他进程也无法使用。

3. 非抢占

有几种破坏非抢占条件的方法。首先，如果拥有某种资源的进程在申请其他资源遭到拒绝时，它必须释放其占有的资源，如果以后有必要，它可再次申请上述资源。另一方法为，当一进程申请的资源正被其他进程占用时，可通过操作系统抢占这一资源，但当两个进程的优先级相同时，这个方法不能防止死锁。在实践中，后一种方法仅适于那些状态容易保留和恢复的资源，例如，处理机。

4. 循环等待

通过将不同类型的资源定序可破坏循环等待条件。如果一个进程已得到类型为R的资源，那么以后它只能申请在R次序之后的资源。

为了解其具体工作过程，为每种资源类型建立索引，于是，如果R_i在R_j之前，那么$i<j$。假设有两个进程A和B，A申请R_i和R_j，并且B申请R_j和R_i，那么，这种情

况导致死锁是不可能的，因为它意味着$i<j$且$j<i$。

以上预防占用并等待和循环等待的方法效率低下，它们降低了处理速度。

4.2.2　死锁检测

死锁预防的方法是非常保守的。它们是通过限制对资源的访问和对进程加以种种约束来解决死锁问题的。死锁检测则走向另一个极端。它对访问资源和进程没有任何约束，只要有可能资源就分配给发出请求的进程。操作系统只是周期地执行检测循环等待条件的算法。

可以在每次申请资源时进行死锁检测，也可以少检测一些，这取决于系统发生死锁的频繁程度。每次在申请资源前进行死锁检测有两个好处，它可以更早检测到死锁，而且其算法相对简单。但是，这种频繁的检测也耗费了相当多的处理机时间。

一旦检测到死锁，就要有恢复的方法，以下是一些可能的方法，它们以复杂度增加的次序排列：

① 杀死所有的死锁进程。这是操作系统中最常用的方法。

② 所有的死锁进程都退回到原来已定义的检测点，然后重新执行它们。这要求系统有撤回和重启机制。这种方法的危险性在于原来的死锁可能再次出现。而并发进程的不确定性常常使死锁可能不再发生。

③ 逐个杀死死锁进程直至死锁消除。杀死进程的顺序根据开销最小的原则确定。每杀死一个进程后都要调用检测算法检测死锁是否存在。

④ 逐个抢占其他进程的资源直到死锁不再存在。同第③点一样也要根据开销来选择抢占哪个进程的资源，然后在每次抢占之后执行检测算法。资源被抢占的进程必须撤回到它拥有该资源之前的某一点。

第③点和第④点中的选择标准如下：

- 已耗用处理机时间最少的进程；
- 已产生的输出最少的进程；
- 为执行完毕还需时间最长的进程；
- 已分配的资源最少的进程；
- 最低优先级的进程。

上述标准中，有些标准难于度量，预测进程还需要多长时间才能完成尤其困难，而且除了优先级以外，对整个系统而言，开销的大小也不明确。

4.2.3　死锁避免

另一个只在细节上与死锁预防方法不同的途径是死锁避免。死锁预防是通过限制对资源的申请至少破坏死锁4个条件中的一个条件来实现的。这可以通过防止前3个必要条件(互斥，占用并等待，非抢占)中的一个产生来实现，也可以直接地防止出

现循环等待。但此方法也使资源利用率和进程执行效率大大降低。死锁避免虽然允许出现3个必要条件，但是，它是通过有效的选择来确保系统不出现死锁的。死锁避免比死锁预防允许更多的进程并发执行。死锁避免的方法对资源分配的判断是动态的，这就有出现死锁的潜在危险，所以，死锁避免需要知道未来的资源请求信息。

死锁避免有以下两种方法：

① 如果进程对资源的申请可能导致死锁，就不启动这个进程

② 如果进程对资源的申请可能导致死锁，就不给进程分配该资源。

1. 避免启动新进程

对于一个有n个进程和m个不同类型资源的系统，定义以下向量和矩阵：

$\text{Resource}=\begin{bmatrix}R_1\\ \text{M}\\ R_m\end{bmatrix}$，该向量表示系统拥有的资源数量。

$\text{Available}=\begin{bmatrix}AV_1\\ \text{M}\\ AV_m\end{bmatrix}$，该向量表示系统当前可供使用的资源数量。

$$\text{Claim}=\begin{bmatrix}C_{11}\Lambda\ C_{n1}\\ \text{M}\quad\ \text{M}\\ C_{1m}\quad C_{nm}\end{bmatrix}=(C_{1*}\cdots C_{n*})$$

$$\text{Allocation}=\begin{bmatrix}A_{11}\Lambda\ A_{n1}\\ \text{M}\quad\ \text{M}\\ A_{1m}\quad A_{nm}\end{bmatrix}=(A_{1*}\Lambda\ A_{n*})$$

矩阵Claim给出了每个进程对每类资源需求的最大数量，即C_{ij}表示“进程i对资源j的需求量”。这个信息必须预先通知死锁避免算法。矩阵Allocation给出了当前分配给每个进程的每类资源的数量。其中，元素A_{ij}表示“当前分配给进程i的资源j的数量”。于是，有以下关系：

① $\sum_{k=1}^{n}A_{k*}+\text{Available}=\text{Resource}$，即所有资源要么空，要么已分配。

② 对所有的k，$C_{k*}\leqslant\text{Resource}$，即任一进程都不能申请超出系统拥有的资源数量。

③ 对所有的k，$A_{k*}\leqslant C_{k*}$，即进程实际分配的资源数量不能超过它原先声明的需求量。

有了以上定义，下面给出死锁避免的规则：

执行一个新进程P_{n+1}，当且仅当 $\text{Resource}\geqslant\sum_{k=1}^{n}C_{k*}+C_{(n+1)*}$，即当且仅当当前所有进程和新进程的最大资源需求量之和能被系统满足时，才能启动一个新的进程。这个方法并不是最优的，因为它考虑的是最坏的情况，即所有的进程都使用它们最

大的资源需求量。

2. 避免分配资源

避免分配资源也称作Banker(银行家)算法，该算数最先由Dijkstra提出。首先定义状态和安全状态两个概念。设有一个有固定进程资源数的系统，系统的状态是指资源分配给进程的情况，故状态包括Resource和Available两个向量，Claim和Allocation两个矩阵。安全状态是指进程至少能以一种顺序执行完毕而不会导致死锁的状态。

下面的例子说明了这些概念。图4.3(a)给出了一个包含4个进程和3种资源的系统的状态。资源R_1，R_2和R_3分别含有9、3和6个例示。当前，资源已分配给4个进程，只有R_2和R_3有1个空闲例示，问题在于这个状态安全吗？为解答这个问题先来看一个更深入的问题：利用现有资源，这4个进程都能执行完毕吗？显然P_1不可能，P_1只占有R_1的1个例示，却申请R_1的两个例示，R_2的两个例示和R_3的两个例示。然而将R_3的1个例示分配给P_2，则P_2分配到最大需求，于是P_2可以完成。P_2完成后它所占用的资源将成为可用资源，这时的状态如图4.3(b)所示。现在每个进程就都能执行完毕了。假设先执行P_1，P_1完成后状态如图4.3(c)所示，下一步完成P_3后状态如图4.3(d)，最后完成P_4。所以图4.3(a)中的状态是安全状态。

(a)

	P_1	P_2	P_3	P_4
R_1	3	6	3	4
R_2	2	1	1	2
R_3	2	3	4	2

Claim

	P_1	P_2	P_3	P_4
R_1	3	6	3	4
R_2	2	1	1	2
R_3	2	3	4	2

Allocation

R_1	0
R_2	1
R_3	1

Available

(b)

	P_1	P_2	P_3	P_4
R_1	3	0	3	4
R_2	2	0	1	2
R_3	2	0	4	2

Claim

	P_1	P_2	P_3	P_4
R_1	1	0	2	0
R_2	0	0	1	0
R_3	0	0	1	0

Allocation

R_1	6
R_2	2
R_3	3

Available

(c)

	P_1	P_2	P_3	P_4
R_1	0	0	3	4
R_2	0	0	1	2
R_3	0	0	4	2

Claim

	P_1	P_2	P_3	P_4
R_1	0	0	2	0
R_2	0	0	1	0
R_3	0	0	1	2

Allocation

R_1	7
R_2	2
R_3	2

Available

(d)

	P_1	P_2	P_3	P_4
R_1	0	0	0	4
R_2	0	0	0	2
R_3	0	0	0	2

Claim

	P_1	P_2	P_3	P_4
R_1	0	0	0	0
R_2	0	0	0	0
R_3	0	0	0	2

Allocation

R_1	9
R_2	3
R_3	4

Available

图 4.3　安全状态的判定

死锁避免方法保证了系统永远处于安全状态。具体方法如下，当进程发出资源

请求时，假设满足其请求并相应修改系统状态，再判断系统是否处于安全状态，如果是就满足该请求，如果不是就阻塞该进程，直到满足其请求后系统仍处于安全状态为止。

考虑图4.4(a)定义的系统状态。假设P_2申请R_1的1个例示和R_3的1个例示，且系统满足其要求，于是有状态如图4.4(a)所示，这是一个安全状态，所以满足这个请求是安全的。回到图4.4(a)所示的状态，假定P_1申请R_1和R_3各一个例示，且满足该申请，于是有状态如图4.4(b)所示。这个状态是不安全的，因为每个进程都还要R_1的1个例示，而这是不能满足的。因此，根据死锁避免规则，P_1的要求被拒绝而且P_1被阻塞。

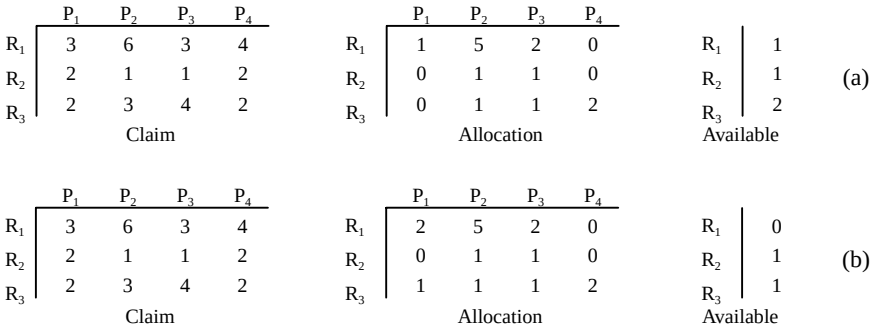


图 4.4　非安全状态的判定

需要指出的是，图4.4(b)所示的状态并不是死锁状态，它仅仅有发生死锁的潜在危险。例如，P_1在需要这些资源之前可能先释放R_1的1个例示和R_3的一个例示，这时系统就会进入安全状态。所以，死锁避免仅仅是在某种程度上预防死锁的发生。

图4.5所示的算法是由Krakowiak和Beeson给出的。系统状态由数据结构state描述，request[∗]是用于定义进程*i*所需资源例示数的向量。首先，进行检查以确认资源申请没有超过进程最初声明的需求量。如果资源申请合法，下一步就决定是否满足该申请，即是否有足够的可用资源。如果没有，则进程将被挂起。如果有，则最后一步检查满足该请求后系统是否是安全的。为此，资源将暂时分配给进程*i*，于是就可以用图4.5(c)所示的算法检测其安全与否。

死锁避免的优点在于它不像死锁检测那样必须抢占并撤回进程，而且它比死锁预防的限制少。然而死锁避免也有不少缺点和限制：

① 进程要预先声明资源的最大需求量；

② 进程必须是独立的，即进程的执行顺序不受同步要求的约束；

③ 资源和进程的数目必须固定。

```
type state=record
        resource, available: array [0..m–1] of integer;
        claim, allocation: array [0..n–1，0..m–1] of integer
      end
```

(a)

```
if allocation [i,*]+request [*]＞claim [i,*] then
  〈error〉                    {total request＞claim}
else
  if request [*]＞available [*] then
    〈suspend process〉
else
    〈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
```

(b)

```
function safe (state:S):boolean;
var currentavail: array[0..m–1] of integer;
    rest: set of process;
begin
currentavail:=available;
rest:={all processes};
possible:=ture;
while possible do
  find a Pk in rest such that
    claim[k,*]–allocation [k,*]＜currentavail;
  if found then                {simulate execution of P}
    currentavail:=currentavail+allocation[k,*];
    rest:=rest–{Pk}
  else
    possible:=false
  end
end;
safe:=(rest=null)
end.
```

(c)

图 4.5　死锁避免算法

(a) 全局数据结构　(b) 资源分配算法　(c) 银行家算法

4.2.4　采用综合方法处理死锁

如表4.1所示，处理死锁的方法各有其优点和缺点，在不同的环境中使用不同的方法就比只用一种方法要好得多，例如，

① 对资源进行分类。

② 对不同类的资源使用资源定序的方法，这样可以防止不同资源类因循环等待而出现的死锁。

③ 对每一类资源，分别使用其最佳的处理方式。

例如，可将资源作如下分类：

① 可交换空间。每个用户作业的辅助存储空间。

② 作业资源。可分配的设备和文件。

③ 主存。以页/段的形式给用户作业分配内存。

④ 内部资源。系统使用的资源，例如，I/O通道。

利用混合方法对每一类资源处理如下：

① 可交换空间。由于最大存储空间的请求通常是知道的，因此，可使用死锁避免中的预分法。

② 作业资源。由于从进程控制块中可获取资源请求的信息，所以，可以采用死锁避免的方法。通过资源定序来预防死锁也是一个可行的方法。

③ 主存。由于作业总可以被交换出来，而且主存可以抢占，因而可以采用抢占主存的方法防止死锁。

④ 内部资源。使用资源定序的方法预防死锁。

4.3　哲学家用餐问题

哲学家用餐问题可以描述如下：5个哲学家聚在一起思考问题和用餐。他们共用一个由5把椅子围起的圆桌，每把椅子归一个哲学家使用。桌子中间是一些饭菜，桌子上还有5个碟子和5根筷子。当一个哲学家思考问题时，他并不影响他的同事。当哲学家需用餐时，就去拿最靠近他的两根筷子，一个哲学家一次只能拿一根筷子，当然不能拿已在临近位置上同事手中的筷子。当一个哲学家拿到两根筷子后，他就用餐而不放下筷子。当用餐完毕后，就把手中的筷子放回原处后再思考问题，如图4.6所示。

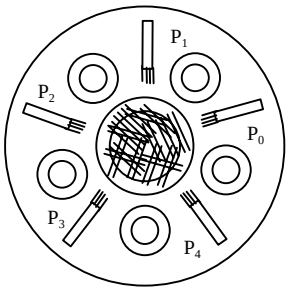


图 4.6　哲学家用餐问题

这个问题最早由Dijkstra提出并解决的，它是一个死锁和饥饿的典型问题，也是一大类并发控制所面临的问题。

图4.7给出了使用信号量解决此问题的一种方法。每个哲学家先拿其左边的筷子然后再拿右边的筷子。哲学家用完餐后再将筷子放回原来的位置。这种解决方法将导致死锁。例如，当所有的哲学家都想用餐时，他们都会拿起各自左边的筷子。

```
program diningphilosophers;
var       fork: array[0..4] of semaphore (:=1);
            i:integer;
procedure philosopher (i:integer);
begin
  repeat
    think;
    wait (fork[i]);
    wait (fork[(i+1) mod 5]);
    eat;
    signal (fork [(i+1) mod 5]);
    signal (fork[i])
  forever
end;
begin
  parbegin
  philosopher (0);
  philosopher (1);
  philosopher (2);
  philosopher (3);
  philosopher (4);
  parend
end.
```

图 4.7　哲学家用餐问题的第一个解法

这时他们的右边就不再有筷子，于是所有的哲学家只有挨饿。

为避免死锁，可以只允许不超过4个哲学家同时用餐。这样就至少有一个哲学家可以得到两根筷子。图4.8给出了另一种使用信号量的解决方法。这个方法可避免死锁和饥饿。

```
program diningphilosophers;
var     fork: array[0..4] of semaphore(:=1);
        room:semaphor(:=4);
        i:integer;
procedure philosopher (i:integer);
begin
  repeat
    think;
    wait (room);
    wait (fork[i]);
    wait (fork [(i+1) mod 5]);
    eat;
    signal (fork [(i+1) mod 5]);
    signal (fork [i]);
    signal (room)
  forever
end;
begin
  parbegin
    philosopher (0);
    philosopher (1);
    philosopher (2);
    philosopher (3);
    philosopher (4);
  parend
end.
```

图 4.8　哲学家用餐问题的第二个解法

4.4　系统举例

4.4.1　UNIX System V

UNIX提供了许多机制来进行进程间通信和同步。这里讨论其中最重要的机制：管道(Pipe)、消息(Message)、共享内存(Shared Memory)、信号量(Semaphores)和信号(Signals)。

管道、消息和共享内存提供了一种在进程间交换数据的方法，而信号量和信号用来触发某些行为。

1. 管道

UNIX对操作系统的发展最有意义的贡献之一就是管道。管道就是一个环状缓冲区，它允许两个进程以生产者/消费者的模式进行通信。因此，它是一个先进先出队列，由一个进程写，另一个进程读。

当产生一个管道时，它被给予固定的字节大小，一个进程要写管道时，如果有足够的空间，写要求就立即被执行；否则，该进程被阻塞。类似地，一个读进程要读比当前管道中多的字节时，它也被阻塞；否则，读要求立即被执行。操作系统实

施互斥机制，就是一次只有一个进程能访问管道。

2. 消息

消息就是有相应类型的一个文本段。UNIX提供msgend和msgrw操作来请求进程传递消息，每个进程都带有一个消息队列，相当于一个邮箱。

消息发送者检查每个发送消息的消息类型，这可被接受者用来作为选择标准。接受者可按FIFO顺序或按类型来检索消息，一个进程向一个满队列发送消息时，它就被挂起。进程读空队列时也会被挂起。如果一个进程要读某一类型的消息而失败了，它不会被挂起。

3. 共享内存

UNIX中进程通信的最快方法是通过共享内存，它是一个被大量进程共享的虚拟内存块。进程就像读/写其他虚拟内存空间一样，用同样的机器指令来读/写共享内存，是否允许进程只读或读/写共享内存，这要根据每个进程来决定，互斥的限制不是共享内存的一部分，而必须由使用共享内存的进程来提供。

4. 信号量

UNIX中的信号量系统请求是第3章中定义的wait和signal原语的一般化，因此，可以同时执行几个操作，并且增减值可以大于1。

一个信号量由以下元素组成：

① 信号量的当前值；

② 操作信号量的上一个进程的进程ID；

③ 等待比当前信号量值要大的进程数；

④ 等待信号量值为0的进程数。

信号量带有挂起在该信号量上的进程队列。

实际上信号量以集合的形式产生，一个信号量集包含一个或多个信号量，semctl系统请求允许同时设置集合内所有信号量值，sem-op系统请求带有一个信号量操作列表，每一个操作对应集合内的一个信号量。当请求时，内核一次执行一个所指示的操作。对每个操作，其实际功能由sem-op值决定，有以下几种可能：

① 若sem-op为正数，则内核增加信号量的值，并唤醒所有等待增加信号量值的进程。

② 若sem-op为0，内核就检查信号量值。如果信号量值为0，就继续执行其他操作，否则，它将等待该信号量的进程数置为0，挂起信号量值为0的事件的进程。

③ 若sem-op为负数，并且其绝对值小于或等于信号量值，内核就将sem-op (负数)加到信号量值上，如果结果为0，内核就唤醒所有等待信号量值为0的进程。

④ 若sem-op为负数，并且其绝对值大于信号量值，内核就将信号量值增加的事件的进程挂起。

信号量在进程同步和合作方面相当灵活。

5. 信号

信号就是告诉一个进程发生了同步事件的软件机制。信号类似于硬件中断，但没有优先级。也就是说，所有信号被同等处理；同时发生的信号一次一个的发送给进程，而没有特定的顺序。

进程可相互间发送信号，内核可以在内部发送信号。一个信号通过将对信号要送到的进程的进程表中一个域的修改来传送。由于每个信号都被看做一个单位，给定类型的信号不能排队，只是在进程被唤醒后才处理信号，或者是在进程要从系统请求返回时，进程可以执行一些错误行为(如：终止)来响应信号，或者忽略该信号。

表4.2列出了UNIX中所定义的信号。

表 4.2　UNIX 的信号

值	名	描　　述
01	SIGHUP	挂起；内核认为某进程的用户正在做无用的工作时就发送此信号给该进程
02	SIGINT	中断
03	SIGQUIT	退出；由用户发送，用来中止进程和转储内存信息
04	SIGILL	非法指令
05	SIGTRAP	跟踪；触发进程跟踪代码的执行
06	SIGIOTIOT	IOT指令
07	SIGEMTEMT	EMT指令
08	SIGFPT	浮点异常
09	SIGKILL	杀死；终止进程
10	SIGBUS	总线错误
11	SIGSEGV	违反分段；在某进程想要访问超出其虚拟地址空间的位置时就发出
12	SIGSYS	系统请求的坏变元
13	SIGPIPE	写没有读进程的管道
14	SIGALARM	警钟；在一个进程想要在一段时间后接收一个信号时发出
15	SIGTERM	软件终止
16	SIGUSR1	用户定义信号1
17	SIGUSR2	用户定义信号2
18	SIGCLD	一个后代被杀死
19	SIGPWR	电源故障

4.4.2　Windows NT

Windows NT将线程的同步作为对象结构的一部分。Windows NT用来实现同步的机制是同步对象家族，包括：进程、线程、文件、事件、事件对、信号、计时器和变体(Mutant)。前3个对象类型除用来同步外，还有其他用途。剩下的对象类型是

专门为支持同步而设计的。

每个同步对象都可以处于信号态，或者是非信号态。线程可挂起在对象的非信号态；当对象进入信号态时，该线程被释放，其机制非常简单：一个线程通过对象处理向Windows NT发出一个等待请求。当一个对象进入信号态时，Windows NT就释放所有等待该同步对象上的线程对象。

表4.3总结了引起每个对象类型进入信号态的事件及其对等待线程的影响，事件对是与客户-服务器相互作用相关的对象，它只能被该客户线程和该服务器线程访问。客户或服务器可以将对象置为信号态，从而引起其他线程被传送信号。对所有的其他同步对象，可以有大量线程等待在该对象上。

表 4.3　Windows NT 同步对象

对象类型	定　义	置为信号态的时间	对等待线程的影响
进　程	一个程序调用，包括运行该程序所需的地址空间和资源(数据等)	上一个线程终止时	释放所有等待线程
线　程	进程内一个可执行的实体	线程终止时	释放所有等待线程
文　件	一个打开文件或I/O设备的例示	I/O操作完成时	释放所有等待线程
事　件	对一个系统事件已发生的通知	线程设置事件时	释放所有等待线程
事件对	通知一个专用客户线程已将一个消息拷贝到Win32服务器上或相反	专用客户或服务器设置事件时	释放其他专用线程
信　号	用来调整使用一个资源的线程数的计数器	信号值降为0时	释放所有等待线程
计时器	记录时间的计数器	设置到达时间或期望时间片时	释放所有等待线程
变　体	在Win32和OS/2环境中提供互斥的机制	所属线程或其他线程释放变体时	释放一个线程

变体对象用来实现对资源访问的互斥，它一次只允许一个线程对象访问资源，因此，它就像一个二元信号量。当变体对象进入信号态时，只释放一个等待该变体的进程，在对象进入同步态时释放所有等待线程。

4.4.3　MVS

MVS根据所用的资源提供两种实行互斥的机制：排队和加锁。排队用在用户控制的资源如文件上，而加锁是对于MVS系统资源而言的。

1. 排队

排队机制用来控制对共享数据资源的访问。所需资源可以是整个磁盘卷、一个或多个文件、文件中的记录、程序控制段，或者是主存内任何工作区。如果资源是只读类的，进程就请求共享控制；如果可以修改数据，就要请求互斥控制。表4.4列出了在有排队请求时MVS确定行为的规则。该策略采用读者/写者模式，写者有优先

权。

表 4.4　MVS 排队机制的决策表

请求类型	请求时资源的状态		
	自由的	共享的	互斥
共享	授予	授予，除非互斥请求排队排队	排队
互斥	授予		排队

除越权访问外，用户可以规定要测试资源的可用性但不需要资源控制，或者用户可以规定只有在资源立即可用时才将资源控制权分配给请求者。这可以用来预防死锁，或者通知操作员出现了问题。

2. 加锁

MVS对系统资源的访问采取加锁方法来保证互斥。一个锁就是公用虚拟存储器中的一块区域，它通常被永久地保留在主存中。它包含一些用来指示该锁是否被用的位，如果被用，是谁占有资源等。锁有两种类别和两种形式。类别如下：

① 全局。涉及所有地址空间。

② 本地。涉及某地址空间的所有任务。

形式如下：

① 旋式锁。处理器执行等待锁的测试指令，这是忙等。

② 挂起锁。等待任务被挂起。

旋式锁用在运行时间较短的临界段上，挂起锁用在较长的临界段上。挂起一个被拒绝进程有益处，这之后可以调度其他进程。本地锁一般是挂起锁，而全局锁既可以是旋式锁，也可以是挂起锁。

为预防死锁，可按分层结构来组织锁，这就是前面所讨论的用来防止循环等待条件的策略。表4.5列出了MVS中锁的分层，处理器只能请求比它当前所拥有的锁更高层的锁。

表 4.5　MVS 锁

锁　名	类　别	型　式	描　　述
RSMGL	全局锁	旋式锁	真正的存储管理全局锁——串行化RSM全局资源
VSMFIX	全局锁	旋式锁	虚拟存储管理固定子池锁——串行化VSM全局队列
ASM	全局锁	旋式锁	辅助存储管理锁——在地址空间层串行化ASM资源
ASMGL	全局锁	旋式锁	辅助存储管理全局锁——在全局层串行化ASM资源
RSMST	全局锁	旋式锁	真正的存储管理锁——不知道当前所拥有的地址空间锁时，在地址空间层串行化RSM控制段

续表

锁　名	类　别	型　式	描　　述
RSMCM	全局锁	旋式锁	真正的存储管理公用锁——串行化RSM公用区域资源(如页表)
RSMXM	全局锁	旋式锁	真正的存储管理内存锁——当串行化需要另一个地址空间时，在地址空间层串行化RSM控制段
RSMAD	全局锁	旋式锁	真正的存储管理地址空间锁——在地址空间层串行化RSM控制段
RSM	全局锁	旋式锁	真正的存储管理锁(共享/互斥)——在全局层串行化RSM功能和资源
VSMPAG	全局锁	旋式锁	虚拟存储管理分页子池锁——为VSM分页子池串行化VSM工作区
DISP	全局锁	旋式锁	全局调度锁——串行化ASVT和ASCB调度队列
SALLOC	全局锁	旋式锁	空间分配锁——串行化接收过程
IOSYNCH	全局锁	旋式锁	I/O管理同步锁——串行化IOS资源
IOSUCB	全局锁	旋式锁	I/O管理单元控制段锁——串行化对UCB的访问和修改。每个UCB只有一个IOSUCB
SRM	全局锁	旋式锁	系统资源管理锁——串行化SRM控制段及其相应数据
TRACE	全局锁	旋式锁	跟踪锁(共享/互斥)——串行化系统跟踪缓冲压
CPU	全局锁	旋式锁	处理器锁——提供系统识别(合法)的屏蔽

4.5 小　　结

死锁是由于进程间相互竞争系统资源或通信而引起的一种阻塞现象。如果操作系统不采取特别的措施，这种阻塞将永远存在。例如，可以杀死一个或多个进程或强迫他们撤回。死锁可能涉及到可重用资源和消耗型资源，消耗型资源在被进程占用时即消失，例如，消息和I/O缓冲区中的信息。可重用资源是不因使用而受到破坏的资源，例如，I/O通道和存储器。

处理死锁的方法通常有3种：死锁预防、死锁检测和死锁避免。死锁预防通过破坏产生死锁的3个必要条件而保证不会出现死锁。在操作系统随时满足资源请求时就要用到死锁检测。操作系统检测到死锁并采取措施消除死锁。死锁避免是通过对资源请求是否有可能导致死锁的分析来消除发生死锁的可能性。

习　题　四

4.1　设当前的系统状态如下：

available

R_1	R_2	R_3	R_4
2	1	0	0

	current allocation				maximum demand				still needs			
process	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	0	2	0	0	1	2				
P_2	2	0	0	0	2	7	5	0				
P_3	0	0	3	4	6	6	5	6				
P_4	2	3	5	4	4	3	5	6				
P_5	0	3	3	2	0	6	5	2				

(1) 计算各个进程的“still needs”。

(2) 系统是否处于安全状态，为什么？

(3) 系统是否死锁，为什么？

(4) 哪些进程可能死锁？

4.2　(1) 假设3个进程共享4个资源，每个进程一次只能预定或释放一个资源，每个进程最多需要两个资源，证明这样做不会发生死锁。

(2) 假设*N*个进程共享*M*个资源，每个进程一次只能预定或释放一个资源，每个进程最多需要*M*个资源，所有进程总共的需求少于*M*+*N*个资源，证明此时不会发生死锁。

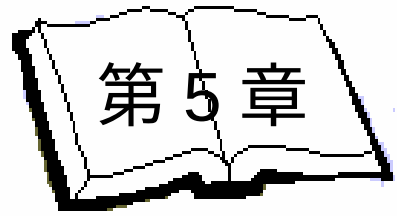
4.3　假设有两种类型的哲学家，一种类型的哲学家总是首先拿其左边的筷子，称为“lefty”，另一种总是首先拿其右边的筷子，称为“righty”，lefty的行为已在图4.7中定义过，righty的行为定义如下：

```
beign
  repeat
    think;
    wait (fork [(i+1) mod 5]);
    wait (fork [i]);
    eat;
    signal (fork [i]);
    signal (fork [(i+1) mod 5])
  forever
end;
```

证明以下结论(在餐桌上至少各有一名lefty和righty型的哲学家)：

(1) 任何一种座位排列都能避免死锁。

(2) 任何一种座位排列都能避免饥饿。



内存管理

冯·诺依曼的存储机制要求任何一个程序必须装入内存才能执行，现代计算机系统就是基于这种机制之上的。在计算机系统中，内存管理在很大程度上影响着这个系统的性能，这使得存储管理成为人们研究操作系统的中心问题之一。

现代计算机系统中的存储器通常由内存(Primary Storage)和外存(Secondary Storage)组成。CPU 直接存取内存中的指令和数据，内存的访问速度快，但容量小、价格贵；外存不与 CPU 直接交互，用来存放暂不执行的程序和数据，但可以通过启动相应的 I/O 设备进行内、外存信息的交换，外存的访问速度慢，但容量大大超过内存的容量，价格便宜。虽然随着硬件技术和生产水平的迅速发展，内存的成本急速下降，但是，内存容量仍是计算机资源中最关键且最紧张的资源。因此，对内存的有效管理仍是现代操作系统中十分重要的问题。

5.1 概　　述

5.1.1 基本概念

CPU 能直接存取指令和数据的存储器是内存，也称主存、实存，它的结构组织和实现方法将很大程度地决定整个计算机系统的性能，它是现代计算机系统操作的中心。如图 5.1 所示，CPU 和 I/O 系统都要和内存打交道。

内存用来存放内核、程序指令和数据，计算机当前要用到的每一项信息都存放在内存的特定单元中。内存是一个由字或字节构成的大型一维数组，每一个单元都有自己的地址，通过对指定地址单元进行读/写操作来实现对内存的访问。

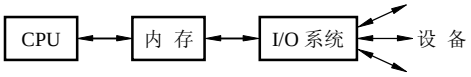


图 5.1　内存在计算机系统中的地位

1. 存储器的层次

尽管内存的访问速度大大高于外存，还是不能与高速的 CPU 相匹配，从而影响整个系统的处理速度。为此，往往把存储器分为 3 级，采用高速缓存器来存放 CPU 近期要用的程序和数据，如图 5.2 所示。高速缓存器由硬件寄存器构成，其存取速度比内存快，但成本远远高于内存，因此，在一个实际系统中的高速缓存器的容量不会很大。往往把某段程序或数据预先从内存调入高速缓存，CPU 直接访问高速缓存，从而减少 CPU 访问内存的次数，提高系统处理速度。

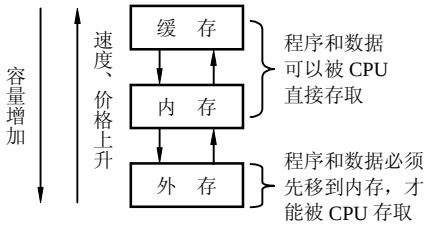


图 5.2　3 级存储器结构

在图 5.2 所示的 3 级存储器结构中，从高速缓存到外存，其容量愈来愈大，如缓存的容量为 128KB～256KB，而内存可达到 256MB；访问数据的速度则愈来愈慢，价格也愈来愈便宜，如 IBM 的缓存的最大传输速度为每字 120ns～225ns，内存的传输速度为每字 1μs。由于本章主要介绍内存管理，因此，不对缓存作详细介绍。

2. 存储管理

在单道程序系统中，内存一次只调入一个用户进程，并且该进程可以使用除操作系统占用的所有内存空间，存储管理就是分配和回收内存区。

在多道程序系统中，多个作业可以同时装入内存，因而对存储管理提出了一系列要求：如何有效地将内存分配给多个作业，如何共享和保护内存等，即要求存储管理具有内存空间管理、地址转换、内存扩充、内存保护和共享等功能。

(1) 内存空间管理

内存空间管理负责记录每个内存单元的使用状态，负责内存的分配与回收。内存分配有静态分配和动态分配两种方式。静态分配不允许作业在运行时再申请内存空间，在目标模块装入内存时就一次性地分配了作业所需的所有空间；而动态分配允许作业在运行时再请求分配附加空间，在目标模块装入内存时只分配了作业所需的基本内存空间。

(2) 地址转换

程序在装入内存执行时对应一个内存地址，而用户不必关心程序在内存中的实际位置。用户程序一旦编译之后每个目标模块都以 0 为基地址进行编址，这种地址称为相对地址或逻辑地址，而内存中各个物理存储单元的地址是从统一的基地址顺序编址，此地址又称为绝对地址或物理地址。当内存分配确定后，需要将逻辑地址转换为物理地址，这种转换过程称为地址转换，也叫重定位。

(3) 内存扩充

内存的容量是受实际存储单元限制的，而运行的程序又不受内存大小的限制，

这就需要有效的存储管理技术来实现内存的逻辑扩充。这种扩充不是增加实际的存储单元，而是通过虚拟存储、覆盖、交换等技术来实现的。内存扩充后，就可执行比内存容量大得多的程序。

(4) 内存的共享和保护

为了更有效地使用内存空间，要求共享内存，例如，当两个进程都要调用 C 编译程序时，操作系统只把一个 C 编译程序装入内存，让两个进程共享内存中的 C 编译程序，这样可以减少内存空间占有，提高内存的利用率。再者，内存共享可实现两个同步进程访问内存区。

当多道程序共享内存空间时，就需要对内存信息进行保护，以保证每个程序在各自的内存空间正常运行；当信息共享时，也要对共享区进行保护，防止任何进程去破坏共享区中的信息。

5.1.2　虚拟存储器

在实存储器存储管理方式中，要求作业在运行前先全部装入内存，这是一种对内存空间的严重浪费，因为实际上有许多作业在每次运行时，并非用到其全部程序。再者，作业装入内存后，就一直驻留在内存中直到作业运行结束，其中有些程序运行一次后就不再需要运行了，却长期占据着内存资源，使得一些需要运行的作业无法装入内存运行，从而降低了内存的利用率，减少了系统的吞吐量，为此，引入了虚拟存储器。

虚拟存储器，并不是以物理方式存在的存储器，而是具有请求调入和交换功能、能从逻辑上对内存容量进行扩充、在作业运行前可以只将一部分装入内存便可运行(给用户提供了一个比真实的内存空间大得多的地址空间)的以逻辑方式存在的存储器。虚拟存储器并不是实际的内存，它的大小比内存空间大得多，其逻辑容量由内存和外存容量之和所决定，其运行速度接近于内存速度，而每位的成本都又接近外存。虚拟存储技术将内存和外存有机地结合起来，把用户地址空间和实际的存储空间区分开来，在程序运行时将逻辑地址转换为物理地址，以实现动态定位，所以实现虚拟存储技术的物质基础是，一要有相当容量的辅助存储器以存放所有并发作业的地址空间；二要有一定容量的内存来存放运行作业的部分程序；三要有动态地址转换机构(DAT)，实现逻辑地址的转换。具体实现和管理虚拟存储器的技术主要有：分页(Paging)、分段(Segmentation)以及段页式(Segmentation with Paging)存储管理技术等。

虚拟存储器最显著的特性是虚拟性，在此基础上它还有离散性、多次性和交换性等基本特征。

(1) 虚拟性

虚拟性是指在逻辑上扩大了内存容量，使得用户所看到的内存地址空间远大于实际内存的容量。例如，实际内存只有 4MB，而用户程序和数据所用的空间都可以

达到 20MB 或者更多。

(2) 离散性

离散性是指内存在分配时采用的是离散分配方式。一个作业分多次装入内存，在装入时占用的内存也不是彼此连续的，而有可能是散布在内存的不同地方，从而避免内存空间的浪费。如将作业装入一个连续的内存区域中，就会需要事先为它一次性地申请足够大的内存空间，就会使一部分内存空间暂时处于空闲状态，而采用离散分配方式，在需调入某些程序和数据时再申请内存空间，就避免了内存空间的浪费。

(3) 多次性

多次性是指一个作业不是全部一次性地装入内存，而是分成若干部分，当作业要装入时，只需将当前运行的那部分程序和数据装入到内存，以后运行到其他部分时，再分别把它们从外存调入内存。这是任何其他存储管理方式都不具备的特征。

(4) 交换性

交换性是指在一个进程运行期间，允许将那些暂不使用的程序和数据，从内存调至外存的交换区，以腾出尽量多的内存空间使其他运行进程调入内存使用，并且被调出的程序和数据在需要时再调入内存中。这种换出、换进技术能有效地提高内存利用率。

5.1.3　重定位

内存有物理内存和逻辑内存两个概念。物理内存是由系统实际提供的存储单元所组成，相应地由内存中的一系列存储单元所限定的地址范围称为内存空间，也称物理空间或绝对空间；而逻辑内存不考虑物理内存的大小和信息存放的实际地址，只考虑相互关联的信息之间的相对位置，其容量只受计算机地址位的限制。若处理器有 32 位地址，则它的虚拟地址空间为 2^{32}，约 4000MB，程序中的逻辑地址范围叫做逻辑地址空间，也称为地址空间。

在多数情况下，一个作业在装入时分配到的存储空间和它的地址空间是不一致的，因此，作业在 CPU 上运行时，其所要访问的指令和数据的实际地址和地址空间的地址不同，这种把地址空间中使用的逻辑地址转换为内存空间中的物理地址的地址转换叫做重定位。

用户程序和数据装入内存时，需要进行重定位。图 5.3 表示了程序 A 装入内存前后的情况。在地址空间 100 号单元处有一条指令 LOAD 1, 500，它实现把 500 号

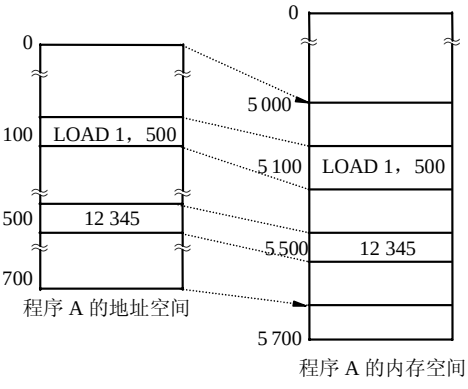


图 5.3　程序装入内存时的情况

单元中的数据 12 345 装到寄存器 1 中去。如果现在将程序 A 装入到内存单元 5 000~5 700 的空间中，而不进行地址转换，则在执行内存中 5100 号单元中的“ LOAD 1，500 ”指令时，系统仍然会从内存的 500 号单元中取出数据，送到寄存器 1 中，很显然，数据出了错。

由图 5.3 可以看出，程序 A 的起始地址 0 不是内存空间的物理地址 0，它与物理地址 5 000 相对应。同样，程序 A 的 100 号单元中的指令放在内存 5 100 号单元中，程序 A 的 500 号单元中的数据放在内存 5 500 号单元中。因此，正确的方法是，CPU 执行程序 A 在内存 5 100 号单元中的指令时，要从内存的 5 500 号单元中取出数据(12 345)送至寄存器 1 中，就是说，程序装入内存时要进行重定位。

根据地址转换的时间及采用技术手段的不同，把重定位分为静态重定位和动态重定位两种。静态重定位是在目标程序装入到内存区时由装配程序来完成地址转换；而动态重定位是在目标程序执行过程中，在 CPU 访问内存之前，由硬件地址映射机构来完成将要访问的指令或数据的逻辑地址向内存的物理地址的转换。

1. 静态重定位

静态重定位是由专门设计的重定位装配程序来完成的。对每个程序来说，这种地址转换只在装入时一次完成，在程序运行期间不再进行重定位。假设目标程序分配的内存区起始地址为 FA，每条指令或数据的逻辑地址为 LA，则该指令或数据映射的内存地址 MA＝FA＋LA。例如，经过动态重定位，原 100 号单元中的指令放到内存 5 100 号单元，该指令中的相对地址 500 相应变成 5 500，以后程序 A 执行时，CPU 是从绝对地址 5 500 号单元中取出数据 12 345，装入到寄存器 1 中。如果程序 A 被装入到 8 000～8 700 号内存单元中，那么，上述那条指令在内存中的形式将是 LOAD 1,8500。依此类推，程序中所有与地址有关的量都要作相应变更。

这种静态重定位方式的优点是，无需增加地址转换机构，在早期的多道程序系统中大多采用此方案。它的主要缺点是，程序的存储空间是连续的一片区域，程序在执行期间不能移动，因而就不能实现重新分配内存，这不利于内存的利用；其次，用户必须事先确定所需的存储量，若所需的存储量超过可用存储空间时，则必须采用覆盖技术；其三，每个用户进程很难共享内存的同一程序的副本，需各自使用一个独立的副本。

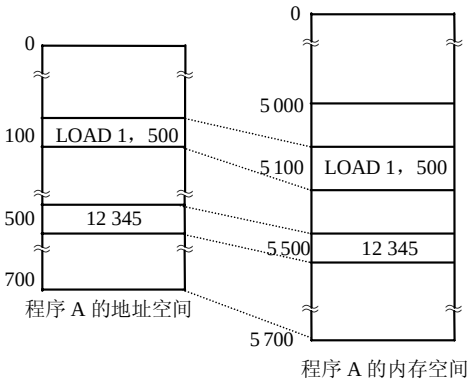


图 5.4　动态重定位示意图

2. 动态重定位

动态重定位是靠硬件地址转换来完成的。通常由一个重定位寄存器(RR)和一个逻辑地址寄存器 LR 组成，其中，RR 存放

当前程序分配到存储空间的起始位置；LR 中存放的是当前被映射的逻辑地址，映射得到的物理地址 MA＝LR＋RR。动态映射过程如图 5.4 所示，只要改变 LR 的内容，就可以改变程序的内存空间，实现程序在内存中移动。由于这种地址转换是在作业执行期间随着每条指令的数据访问自动地、连续地进行的，所以称之为动态重定位。

动态重定位的优点主要是，内存的使用更加灵活有效，几个作业共享一程序段的单个副本比较容易，并且有可能向用户提供一个比内存空间大得多的地址空间，因而无需用户干预，而由系统来负责全部的存储管理，但它需附加硬件支持，且实现存储器管理的软件比较复杂。

5.2　存储管理的基本技术

最基本的 4 种存储管理技术是分区法、可重定位分区法、覆盖技术、交换技术，下面对它们作简要介绍。

5.2.1　分区法

分区管理是满足多道程序设计的一种最简单的存储管理方法。内存划分成若干个大小不同的区域，除操作系统占用一个区域之外，其余由多道环境下的各并发进程共享。其基本原理是给每一个内存中的进程划分一块适当大小的存储块，以连续存储各进程的程序和数据，使各进程能并发进行。按照分区的划分方式，又分为两种常见的分配方法：固定分区法和动态分区法。

1. 固定分区法

固定分区法就是把内存固定划分为若干个不等的区域，在整个执行过程中保持分区长度和分区个数不变。

图 5.5 为固定分区管理的示意图。为了便于内存分配，系统对内存的管理和控制采取分区说明表这样的数据结构，每一分区对应表中的一位，其中包括分区号，分区大小，起始地址和分区状态，并且内存的分配与释放、内存保护以及地址转换等都通过分区说明表来进行。

分区号	大小	起址	状态
1	20KB	20KB	已分配
2	32KB	32KB	已分配
3	64KB	64KB	已分配
4	128KB	128KB	未分配

(a)

0	操作系统
20KB	作业 A
32KB	作业 B
64KB	作业 C
128KB	
256KB	

(b)

图 5.5　固定分区的存储分配

(a) 分区说明表　(b)存储空间分配情况

当某个用户程序装入内存时，就向系统提出要求分配内存的申请，以及需要多大内存空间，这时系统按照用户的申请去检索分区说明表，从中找出一个满足条件的空闲分区给该程序，并修改分区说明表中的状态栏，将状态置为“已分配”；如果找不到足够的分区，则拒绝为该用户程序分配内存。

当一个用户程序执行完后，就不再使用分给它的分区，于是释放相应的内存空间，系统根据分区的起始地址或分区号在分区说明表中找到相应的表项，并将其状态置为“空闲”。

固定分区法管理方式虽然简单，但内存利用率不高。如图 5.5(b)所示，若有作业 D 提出内存申请，需要 64KB 空间，系统将分区 4 分给它，这样，分区 4 就有 64KB 的空间浪费了，因为作业 D 占用这个分区后，不管还有多大的剩余空间，都不能再分配给别的作业使用了。

2. 动态分区法

动态分区分配是根据进程的实际需要，动态地为它分配连续的内存空间，就是说，各个分区是在相应作业装入内存时建立的，其大小恰好等于作业的大小。为了实现分区分配，系统中设置了相应的数据结构来记录内存的使用情况，常用的数据结构形式有以下两种：

(1) 空闲分区表

图 5.6 给出了空闲分区表的例子，内存中每一个未被使用的分区对应该表的一个表项，一个表项中包括分区序号、分区大小、分区的起始地址以及该分区的状态。当分配内存空间时，就从中进行查找，如找到合乎条件的空闲区就分配给作业。

序　　号	分区大小	分区始址	状　　态
1	64KB	44KB	空闲
2	24KB	156KB	空闲
3	48KB	200KB	空闲
4	30KB	300KB	空闲
5	40KB	480KB	空闲
…	…	…	…

图 5.6　空闲分区表

(2) 空闲分区链

空闲分区链是使用链指针把所有的空闲分区链接成一条链，为了实现对空闲分区的分配和链接，在每个分区的起始部分设置状态位、分区大小和链接各个分区的前向指针，由状态位指示该分区是否分配出去了；同时，在分区尾部还设置有一后向指针，用来链接后面的一个分区；分区的中间部分是用来存放作业的空闲内存空间，当该分区分配出去后，状态位就由“0”置为“1”，如图 5.7 所示。

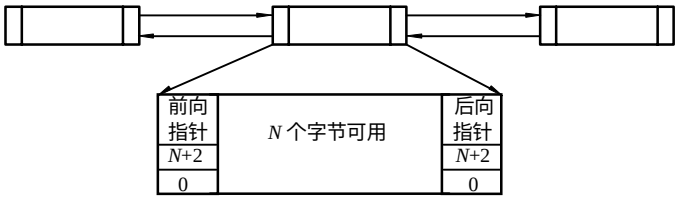


图 5.7　空闲链结构

采取动态分区法时，开始整个内存中只装有操作系统，当作业申请内存时，系统就查表找一个空闲区，该空闲区应足以放下这个作业，如果大小恰好一样，则把该分区分给这个作业使用，然后在空闲分区表或空闲链中再做相应的修改；如果这个空闲区比作业需要的空间大，则将该区分为两部分，一部分给作业使用，另一部分置为较小的空闲区，当作业完成后就释放占有的分区，系统会设法将它和邻接的空闲区合并起来，使它们成为一个连续的更大的空闲区。

5.2.2　可重定位分区法

不管是使用固定分区法还是动态分区法，都必须把一个系统程序和用户程序装入一个连续的内存空间中。虽然动态分区法比固定分区法的内存利用率要高，但由于各作业申请和释放内存的结果，在内存中经常可能出现大量的分散的小空闲区。内存中这种容量太小、无法被利用的小分区称做“碎片”或“零头”，大量的碎片的出现降低了内存中作业的道数，还造成了内存空间的大量浪费。为了使分散、较小的空闲区得到合理的使用，可以定时或在分配内存时就把所有的碎片合并成为一个连续区，也就是说移动某些已分配区的内容，使所有作业的分区紧凑地连在一起，而把空闲区留在另一端，这就是紧凑技术。

紧凑过程中作业在内存中要移动，因而所有关于地址的项均得做相应的修改，如基址寄存器、访问内存的指令、参数表和使用地址指针的数据结构等，采用动态重定位技术可以较好地解决这个问题。

动态重定位经常是用硬件来实现的。硬件支持包括一对寄存器，其中一个用于存放用户程序在内存中的起始地址，即基址寄存器；另一个用于表示用户程序的逻辑地址的最大范围，即限长寄存器。如图 5.8 所示，开始时作业 3 的起始地址是 64KB，其作业大小是 24KB，当执行作业 3 时，系统就把它的起始地址放入基址寄存器中，把其作业大小放入限长寄存器中。由于系统对内存进行了紧凑，作业 3 移动到新位置，起始地址变为 28KB。再执行作业 3 时，就把 28KB 和 24KB 分别放入基址寄存器和限长寄存器中。

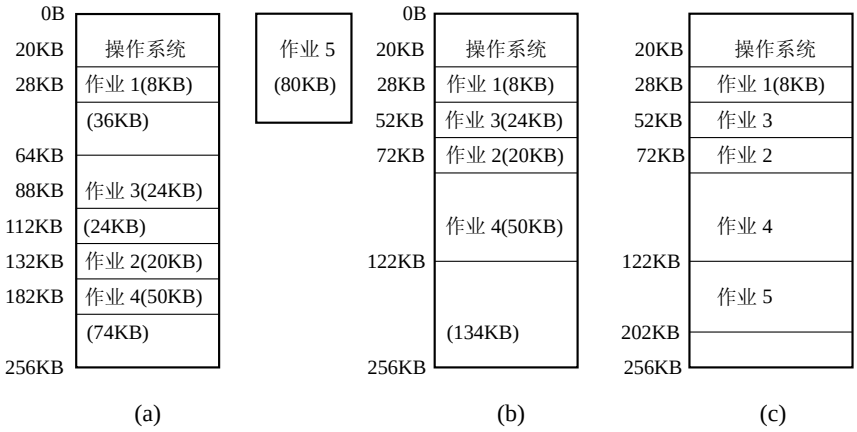


图 5.8　可重定位分区的紧缩

(a) 初始状态　(b) 移动之后　(c) 分配作业 5 之后

动态重定位的实现如图 5.9 所示。从图中可以看出，作业 3 装入内存后，其内存空间中的内容与地址空间中的内容是一样的，就是说，将用户的程序和数据完全地装入内存中。当调度该作业在 CPU 上执行时，操作系统就自动将该作业在内存的起始地址(64KB)装入基址寄存器，将作业的大小(24KB)装入限长寄存器。当执行 LOAD 1，3000 这条指令时，操作对象的相对地址首先与限长寄存器的值进行比较，如果前者小于后者，则表示地址合法，在限定范围内；然后，将相对地址与基址寄存器中的地址相加，所得的结果就是真正访问的内存地址，如果该地址大于限长寄存器的内容，则表示地址越界，发出相应中断，进行处理。

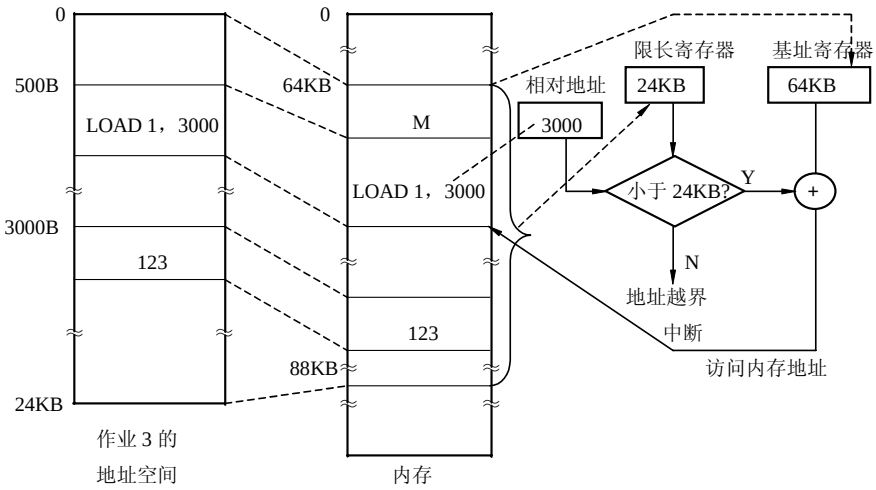


图 5.9　动态重定位的实现过程

5.2.3　覆盖技术

覆盖技术主要用在早期的操作系统中，因为在早期的单用户系统中内存的容量一般少于 64KB，可用的存储空间受到相当限制，某些大作业不能一次全部装入内存中，这就发生了大作业和小内存的矛盾。为此，引入“ 覆盖 ”管理技术，就是把大的程序划分为一系列的覆盖，每个覆盖是一个相对独立的程序单位，把程序执行时并不要求同时装入的内存的覆盖组成一组，称为覆盖段，这个覆盖段分配到同一个存储区域，这个存储分配区域称为覆盖区，它与覆盖段一一对应，并且为了使覆盖区能为相应覆盖段中每个覆盖在不同时刻共享，其大小应由其中最大的覆盖来确定。

覆盖技术要求程序员必须完成把一个程序划分成不同的程序段，并规定好它们的执行和覆盖顺序，操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖。图 5.10 所示为这种覆盖技术的例子，从图中可以看出，程序段 B 不会调用 C，程序段 C 也不会调用 B，因此，程序段 B 和 C 不需要同时驻留在内存，它们可以共享同一内存区；同理，程序段 D、E、F 也可共享同一内存区。整个程序段被分为两个部分，一个是常驻内存部分，称为根程序，它与所有的被调用程序段有关，因而不能被覆盖，图中的程序 A 是根程序；另一部分是覆盖部分，被分为两个覆盖区，一个覆盖区由程序段 B、C 共享，其大小为 B、C 中所要求容量大者，另一个覆盖区为程序段 D、E、F 共享，两个覆盖区的大小分别为 50KB 与 40KB。这样，虽然该进程正文段所要求的内存空间是 190KB，但由于采用了覆盖技术，只需 110KB 的内存空间就可执行。

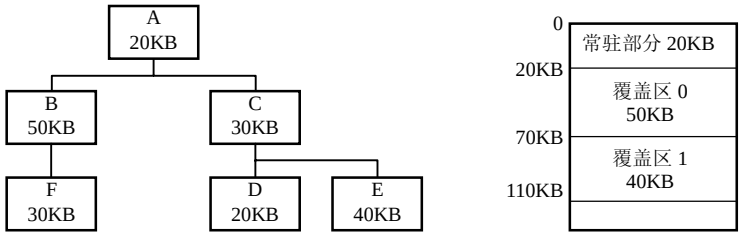


图 5.10　覆盖示例

5.2.4　交换技术

交换技术就是把暂时不用的某个程序及数据部分或全部从内存移到外存中去，以便腾出必要的内存空间，或把指定的程序或数据从外存读到相应的内存中，并将控制权转给它，让其在系统上运行的一种内存扩充技术。与覆盖技术相比，交换不要求程序员给出程序段之间的覆盖结构，而且交换主要是在进程或作业之间进行的，

而覆盖则主要在同一个作业或进程中进行；另外，覆盖只能覆盖与覆盖程序段无关的程序段。

交换进程由换出和换进两个过程组成，换出是把内存中的数据和程序换到外存的交换区，而换进则是把外存交换区中的数据和程序换到内存的分区中。

数据和程序的交换是一种很有效的管理技术，通常可以解决如下问题：

① 作业要求增加存储空间，而分配请求受到阻塞；

② CPU 时间片用完；

③ 作业等待某一 I/O 事件发生，如等待打印机资源等；

④ 紧凑存储空间，需把作业移动到存储空间的新位置上。

交换技术最早是用在 MIT 的兼容分时系统(CTSS)中，任何时刻在该系统的内存中只有一个完整的用户作业，当其运行一段时间后，或由于分配给它的时间片用完，或由于需要其他资源而等待，系统就把它交换到外存上，同时把另一个作业调入内存让其运行。这样，可以在存储容量不大的小型机上实现分时运行，早期的一些小型分时系统多数都是采用这种交换技术。

5.3　分页存储管理

无论是分区技术还是交换技术均要求作业存放在一片连续的内存区域中，这就造成了内存中的碎片问题。可以通过移动信息，利用紧缩法使空闲区变成连续的较大一块来解决问题。另外，分页管理也是解决碎片问题的一种有效办法，它允许程序的存储空间是不连续的，用户程序的地址空间被划分为若干个固定大小的区域，称为 “ 页 ”。页面的典型大小为 1KB；相应地，也可将内存空间分成若干个物理块，页和块的大小相等。

5.3.1　基本概念

1. 页面和物理块

在分页存储管理中，将一个进程的逻辑地址空间划分成若干个大小相等的部分，每一部分称为页面或页，并且每页都有一个编号，即页号，页号从 0 开始依次编排，如 0，1，2，……同样，将内存空间也划分成与页面大小相同的若干个存储块，即为内存块或页框。相应地，它们也进行了编号，块号从 0 开始依次顺序排列为 0 号块，1 号块，2 号块，……在为进程分配内存时，以块为单位将进程中的若干页分别装入多个可以不相邻的块中。

页面和物理块的大小是由硬件即机器的地址结构所决定的。在确定地址结构时，若选择的页面较小，一方面可使内存碎片小，并减少了内存碎片的总空间，有利于提高内存的利用率；另一方面，也会使每个进程要求较多的页面，从而导

致页表过长，占用大量内存；此外，还会降低页面换出换进的效率。若选择的页面较大，虽然可减少页表长度，提高换进换出的效率，但又会使页内碎片增大。因此，页面的大小应该选择得适中，通常页面的大小是 2 的幂，大约在 512B 到 4KB 之间。

在分页存储管理中，表示地址的结构如图 5.11 所示。它由两部分组成，前一部分表示该地址所在页面的页号 P，后一部分表示页内位移 d，即位移量。地址长度为 32 位，其中 0 到 11 为位移量，即每页的大小为 4KB，12 到 31 位为页号，表示地址空间最多允许容纳 1M 个页面。对于某一特定机器来说，它的地址结构是一定的，如果给定的逻辑地址空间中的地址是 A，页面大小为 L，则页号和位移量 d 可按下式求得：

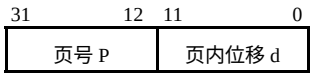


图 5.11　分页技术的地址结构

$$P = \mathrm{INT}\lfloor A/L \rfloor$$

$$d = [A] \quad \mathrm{MOD} \quad L$$

其中，INT 是向下整除的函数，MOD 是取余函数，例如，设某系统的页面大小为 1KB，A=3456，则 P=3，d=384，用一个数对(P，d)来表示，就是(3，384)。

2. 页表

在分页系统中，允许将进程的各个页面离散地装入内存的任何空闲块中，这样，就出现作业的页号连续而块号不连续的情况，但应能在内存中找到每个页面所对应的物理块，为此，系统又为每个进程设立一张页面映象表，简称页表。在进程地址空间内的所有页(0~n–1)依次在页表中有一个页表项，其中记载了相应页面在内存中对应的物理块号。进程执行时，按照逻辑地址中的页号去查找页表中对应项，可找到该页在内存中的物理页号。如图 5.12 所示，页表的作用是实现从页号到物理块号的地址映射，页号 2 对应内存的 5 号块。

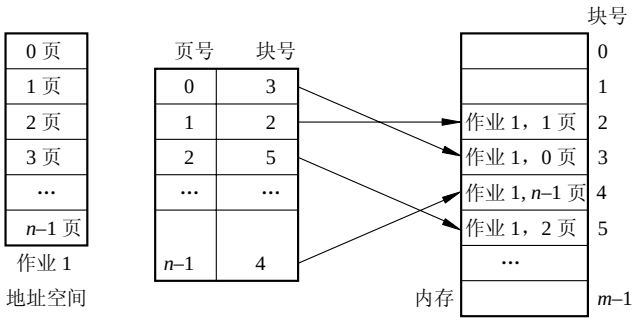


图 5.12　分页存储管理系统

3. 分页系统中的地址转换

分页系统最基本的任务是实现逻辑地址到物理地址的转换，由于页内地址与物理地址是一一对应的，因此，地址转换机构实际上只是将逻辑地址中的页号转换为

内存中的物理块号；又因为页面映射表的作用就是用于实现从页号到物理块号的转换，因此，地址转换任务是借助于页表来完成的。

(1) 基本的地址转换

通常，页表都放在内存中，页表的功能可以由一组专门的寄存器来实现，一个页表项用一个寄存器。由于寄存器具有很高的访问速度，因而有利于提高地址转换的速度。但由于寄存器成本很高，故只适用于较小的系统中。大多数现代计算机的页表都可能很大，页表项的总数可达到几千到几十万个，显然不可能都使用寄存器来实现，因此，页表大多数驻留在内存中。在系统中设置一个页表寄存器 PTR，用来存放页表在内存中的起始地址和页表的长度。一般进程在没执行时，页表的起始地址是存放在本进程的 PCB 中的，当调度程序调度到某一进程时，才将它们装入到页表寄存器。因此，在单处理机环境下，虽然系统中可以运行多个进程，但只需一个页表寄存器。

当进程要访问某个逻辑地址中的数据时，分页地址转换机构会自动地将相对地址分为页号和页内地址两部分，然后再以页号为索引去检索页表。查找操作由硬件执行，在执行检索之前，先将页号与页表长度进行比较，如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址空间，这一错误将产生一地址越界中断，若未出现错误，则将页表始址与页号和页表长度的乘积相加，便得到该表项在页表中的位置，从而得到该页的物理块号，将物理块号装入物理地址寄存器中，与此同时，再将有效地址寄存器中的页内地址直接送入物理地址寄存器的块内地址字段中。这样，便完成了从逻辑地址到物理地址的转换。图 5.13 示出了分页系统的地址转换机构。

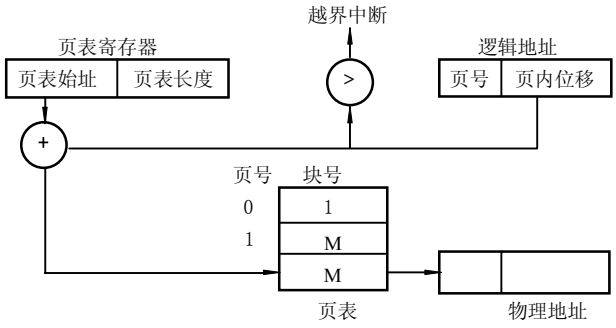


图 5.13　分页系统的地址转换机构

(2) 快表的地址转换

页表存放在内存中，使得 CPU 每要存取一个数据，都要两次访问内存。第一次是访问内存中的页表，从中找到该页的物理块号，由此块号与页内位移量 d 形成物理地址；第二次访问内存时，才是从第一步所得地址中读/写数据。这将使计算机的

处理速度降低近 1/2，为了提高地址转换速度，可在地址转换机构中增设一个具有并行查寻能力的特殊高速缓冲存储器，又称“联想存储器”或“快表”。在 IBM 系统中名为 TLB(Translation Lookaside Buffer)，用以存放当前访问的那些页表项。此时的地址转换过程是，在 CPU 给出有效地址后，由地址转换机构自动地将页号送入高速缓冲存储器，并将此页号与高速缓冲存储器中的所有页号进行比较，若其中有与此相匹配的页号，则表示所要访问的页表项在快表中，就直接读出该页所对应的物理块号，并送物理地址到寄存器中；如对应的页表项不在快表中，还需要再访问内存中的页表，待找到所要访问的页表后，再把页表项中读出的物理地址送入地址寄存器中，同时，还得重新修改快表，将此页表项存入快表中的一个寄存器单元中。但是，如果联想寄存器已满，则操作系统必须找到一个老的但是已被认为不再需要的页表项将它换出。图 5.14 示出了具有快表的地址转换机构。

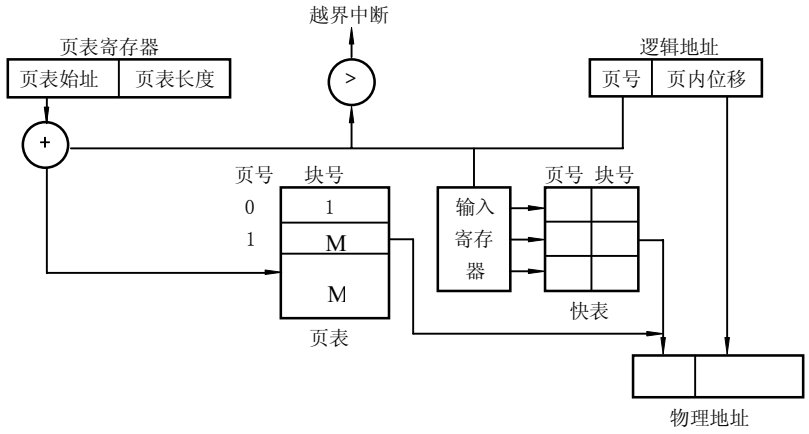


图 5.14　具有快表的地址转换机构

5.3.2　纯分页系统

分页存储管理中，以物理块为单位给作业分配存储空间。例如，一个作业的地址空间有 *m* 页，那么，只要分配给它 *m* 个物理块，每一页分别装入一个物理块内即可，并不需要这些物理块是相互邻接的。所谓纯分页系统就是指在调度一个作业时，必须把它的所有页一次装入到内存的物理块中，如果当时物理块不足，则该作业必须等待，直到有足够的物理块为止，这时系统可再调度另外的作业。

在分页系统中，作业的一页可以分配到存储空间中的任何一个可用的物理块中，作业的地址空间原本是连续的，分页并装入内存后存放在不相邻接的页框内，为了系统的正确运行，在执行每条指令时进行了从逻辑地址到物理地址的转换。为此，利用页表指出该页在内存中的页号，在多道程序设计系统中，为了便于管理和保护，系统为每个装入内存的作业建立一张相应的页表，它的起始地址及大小装入特定的

页表寄存器中。除此之外，根据每页内容的不同，可以设置不同的存取限制。所以，在页表的表项中除了包含指向所在页框的指针外，还包含一个存取控制字段，这个表项也称为页描述子。

为了将逻辑地址转换为物理地址，地址转换机构由两部分组成，页号和页内位移，这也组成了逻辑地址。这个页号首先和页表寄存器表中的当前页表大小比较，如果页号太大，则表明其访问越界，系统产生相应中断。如果页访问是合法的，则先由页表的起始地址和页号计算出相应的页描述子的位置；然后，取出该页描述子，并校验其存取控制字段，以保证这次访问是合法的；最后，将页描述子中的页框号与逻辑地址中位移相拼，形成最终访问内存的物理地址。图 5.15 给出了纯分页系统中由逻辑地址转换成物理地址的过程和一个作业的分配情况及相应页表的内容。

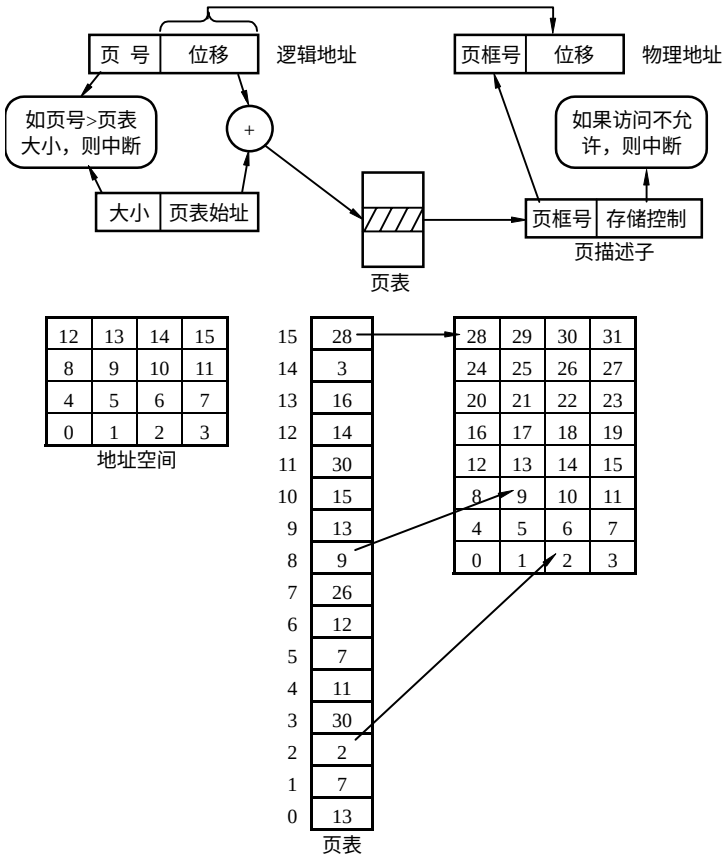


图 5.15　纯分页系统的地址转换过程

这也是上一节地址转换机构中基本的地址转换，为了提高系统的性能，同样在纯分页系统中可以利用快表的地址转换，这在上面已经提及，不再赘述。

5.3.3　请求式分页系统

在纯分页系统中，要求运行的作业一次必须全部装入内存，而往往在许多系统中，可用的存储空间并不总能同时满足所有就绪作业对内存的要求。为此，在纯分页技术的基础上提出了请求式分页系统(Requested Paging System)。其基本思想是，作业在运行之前，只把当前需要的一部分页面装入内存，当需要其他页面时，才自动选择一些页交换到辅存，同时调入所需的页到内存中。这样，减少了交换时间和所需内存的容量，能支持比内存容量大的作业在系统中进行。

自动选择页面交换的机构，称之为虚拟存储系统(Virtual Memory System)，此系统把内存和外存两级存储器结合成一个统一的整体，称为一级存储器，如图 5.16 所示，现在每个作业可用的存储器通常只受到计算机的最大虚拟地址空间的限制。

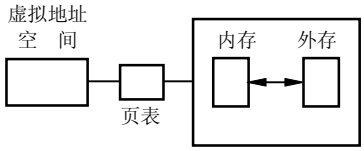


图 5.16　一级存储器

在虚拟存储系统中所用页表的页描述子有了新的扩充，其中包括 3 个标识位，分别表示页面是否存在于内存中、是否被访问过和被修改过。

在请求式分页系统中，其地址转换类似于纯分页系统的地址转换，如果访问的页已在内存中，则与纯分页系统的地址转换过程类似；如果发现缺页，则首先将逻辑地址分为页号和页内位移。开始假设描述子在快表中，同时根据页表寄存器来确定当前页表的始址，并计算出相应页描述子的地址；如果在快表中没有找到这个页描述子，则通过访问内存中的页表来提取它，如果页描述子中页面存在位为 1，则表示该页在内存中，否则必须确定它在外存的位置并把它从外存中调入内存。然而，由于内存空间有限，它往往已被页面占满，这就要求把其他的页从内存中交换出去，这是一个很复杂的过程，通常的作法是一旦发生缺页，由硬件发出中断信号，再由操作系统的相应软件负责调入其所缺的页面。

5.3.4　硬件支持及缺页处理

为了实现请求式分页，系统必须有一定的硬件支持，这包括页表、快表硬件设施，除此之外，还需要依赖内存管理单元来完成地址转换的任务，并且当出现缺页时，在硬件方面，还须增加缺页中断响应机构。

1. 页表

分页系统中的从逻辑地址到物理地址的转换是通过页表来实现的。页表是数组，每一表目对应进程中的一个虚页，页表表目通常为 32 位，它不仅包含该页在内存的基地址，还包含有页面、内存块号、改变位、状态位、引用位和保护权限等信息。当改变位为“ 0 ”时，表示页面没有被修改过，可以淘汰该页；当改变位为“ 1 ”时，

表示相应的页面中内容已被修改过，且淘汰该页时，必须写回磁盘的交换区或相应的文件中。状态位为“ 0 ”表示相应的页面不在内存中；为“ 1 ”表示相应的页面在内存中。引用位为“ 1 ”表示该页被访问过；该位为“ 0 ”表示该页尚未被访问。保护权限允许该页进行任何类型的访问，如果该项只有 1 位，则该位为“ 0 ”表示该页可读/写，该位为“ 1 ”表示只读、不可写，如果该项有 3 位，则每一位分别说明读、写、执行的权限。

2. 快表

快表是为了加快地址转换速度而使用的一个联想高速缓存，使用快表进行地址转换的速度比页表要快很多倍。快表由硬件实现，每个表目对应一个物理页框，包含如下信息：页号、物理块号、数据快表、指令快表和保护权限。

3. 内存管理单元

内存管理单元 MMU 主要是用来完成地址转换，它具有如下功能：将逻辑地址分为页号和页内位移，在页表中先找到与该页对应的物理块号，再将其与页内位移一起组成物理地址；管理硬件的页表地址寄存器；当页表中的状态位为“ 0 ”时，即该页不在内存中，或页面访问越界，或出现保护性错误时，内存管理单元会出现页面失效异常，并将控制权交给操作系统内核的相应程序处理；设置页表中相应的引用位、改变位、状态位和保护权限等。

4. 缺页中断机构

当发现要访问的页面不在内存中时，则立即产生中断信号，这时缺页中断机构将响应中断进行相应处理。

由于页面中断的独特性，缺页中断的处理过程是由硬件和软件共同实现的。系统中提供硬件寄存器或其他机构，在出现页面故障时，要保存 CPU 状态，并且还需要使用返回指令，在出现页面中断处恢复该指令的处理，图 5.17 为中断处理算法流程图。

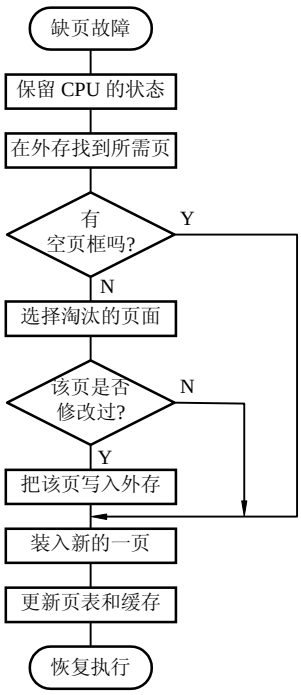


图 5.17　页面中断处理算法流程图

采用虚拟存储技术要注意缺页中断的特殊性和频繁性，如果出现缺页中断的频率太高，则 CPU 花在页面交换的时间会很多，将使系统的性能急剧下降，这就是所谓的系统“ 抖动 (Thrashing) ”。

5.3.5　页的共享和保护

数据共享在多道程序系统中是很重要的，分页存储管理技术使每个作业分别存

储在内存的不连续的存储块中，这种灵活性允许两个和多个作业共享程序库中的例程或公共数据段的同一副本，共享的方法是使这些相关进程的逻辑空间中的页指向相同的内存块。

对于数据页面的共享，实现起来比较简单，因为这个共享的数据页面可以在作业地址空间中的任一页面上，然而，对于库例程的共享却不是这样的，它必须把共享的例程安排到所有共享它的作业地址空间中相同页号的页面中，因为一个作业在运行前必须链接好，而链接后一个例程的所占页号就确定了。如果其他作业要共享该例程，则必须具有与该例程相同的页号才能正常运行。

实际上，并非所有页面都可共享，故实现信息共享的前提是提供附加的保护措施，对共享信息加以保护。例如，某一子程序为一个用户所专有，其他用户只能执行而不能读/写。页面保护是分页系统中必须小心处理的一个问题，即使在纯分页系统中，也常在页表的表格中设置存取控制字段，以防止非法访问，一般设置只读(R)、读写(RW)、读和执行(RX)等权限。如果一进程试图去执行一只允许读的内存块，就会引起操作系统的一次非法访问性中断，系统会拒绝进程的这种尝试，从而保护该块的内容不被破坏。

分页存储管理可以实现对内存进行离散式分配，是目前大型机操作系统中广泛应用的一种存储管理技术。对于纯分页系统而言，它有效地解决了内存的碎片问题，由于分页存储管理中作业的程序段和数据可以在内存中不连续存放，从而可以充分利用内存的每一帧，这有可能让更多的作业同时投入运行，提高处理机和存储器的利用率，但是，由于纯分页系统要求运行的作业必须一次全部装入内存，当作业要求的存储空间大于当前可用的存储空间时，作业只有等待，这样作业地址空间受到内存实际容量的限制，并且要对每个作业建立和管理相应的页表，还要增加硬件实现地址转换，这些都增加了系统时间和空间上的开销。对于请求式分页系统而言，它与纯分页系统一样，消除了内存碎片，同样增加了系统时间和空间的开销，但是，由于每个作业只要求部分装入就可以投入运行，这就大大增加了作业的利用空间，提高了内存的利用率，使作业地址空间不受内存容量大小的限制，可是由于缺页时要进行页面交换也会引起系统“抖动”。

5.4　分段存储管理

用户一般希望把信息按其内容或函数关系分成段，每段有其自己的名字，且可以根据名字来访问相应的程序或数据段，为此形成了分段存储管理。它把程序的地址空间分成若干个不相等的段，每段可以定义一组相对完整的逻辑信息，在进行存储分配时以段为单位，这些段在内存中可以离散分配。

5.4.1　基本概念

1. 分段

在分段存储管理方式中，段是一组逻辑信息的集合。例如，把作业按照逻辑关系加以组织，划分成若干段，并按这些段来分配内存。这些段是主程序段 MAIN、子程序段 P、数据段 O 和堆栈段等。

每个段都有自己的名字和长度，为了实现简单，通常用一个段号来代替段名，每个段都从 0 开始编址，并采用一段连续的地址空间，段的长度由相应的逻辑信息组长度决定，所认各段长度可以不同。

分段系统中的逻辑地址由段号 s 和段内地址 d 组成，其地址结构如图5.18所示。在该地址结构中，允许一个作业最多有64K 个段，每段的最大长度为64KB。通常，规定每个作业的段号从0开始顺序编排：如0段，1段，2段……

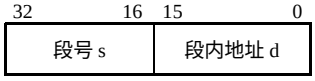


图 5.18　分段系统中的地址结构

不同机器中指令的地址部分有些差异，如有些机器指令的地址部分占 24 位，段号占 8 位，段内地址占 16 位。

2. 分页和分段的主要区别

分页和分段存储管理系统虽然在很多地方有相似之处，例如，它们在内存中都是离散的，都要通过地址映射机构将逻辑地址映射到物理内存中，但二者在概念上完全不同，主要有以下几点：

① 页是信息的物理单位，分页是为了实现离散分配，以减少内存的外零头，提高内存利用率，便于系统管理；而段是信息的逻辑单位，每一段在逻辑上是相对完整的一组信息，如一个函数、过程、数组等，分段是为了更好满足用户需要。

② 分页式存储管理的作业地址空间是一维的，地址编号从 0 开始顺次递增一直排到末尾；而分段式存储管理作业地址空间是二维的，要标识一个地址，除给出段内地址外，还必须给出段号。

③ 页的长度由系统确定，是等长的；而段的长度由具有相对完整意义的信息长度确定，是不固定的。

5.4.2　基本原理

所谓分段管理，就是管理由若干段组成的作业，并且按分段来进行存储分配，由于分段的作业地址空间是二维的，所以分段的关键在于如何把分段地址结构变成一维的地址结构。和分页管理一样，可以采用动态重定位技术来进行地址转换。起初，系统作业建立一张段表，每个表目至少有 4 个数据：段号、段长、内存始址和存取控制。其中，段长指明段的大小，内存始址指出该段在内存中的位置，存取控制说明对该段访问的限制(RWX)。

段地址转换和分页地址转换的过程基本相同，其大致过程如图 5.19 所示。

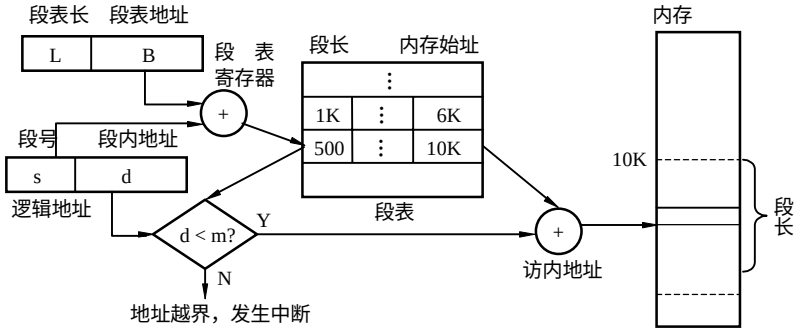


图 5.19　分段地址转换

① 将逻辑地址分为两部分，段号 s 和段内地址 d。

② 将该进程的段表地址寄存器中的段表内存地址 B 与段号 s 相加，得到该进程在段表中相应表项的索引值，从该表项中得到该段的长度以及该段在内存中的起始地址。

③ 将段内地址 d 与段长 m 进行比较，如果 d 大于等于 m，则表示越界，系统将发出越界中断，进而终止程序执行，如果小于 m 则表示地址合法，从而将段内地址 d 与该段的内存始址相加得到所要访问单元的内存地址。

5.4.3　硬件支持和缺段处理

和分页系统一样，为了实现分段式存储管理，系统必须有一定的硬件支持，以完成快速请求分段的功能，这包括段表机制、地址转换机构。同样当出现缺段的问题时，需要有缺段中断机构来进行及时地处理。

1. 段表机制

在分页系统中的主要数据结构是页表，相应地，在请求分段管理中是段表，在段表项中，除了段号(名)、段长、段在内存的始址外，它还包含下列信息：

① 存取方式，用于说明本段的存取属性是只读，还是可读/写，可执行；

② 访问字段，用于记录该段被访问的频繁程度；

③ 改变位，用于表示该页进入内存后，是否被修改过；

④ 状态位，用于表示该段是否调入内存；

⑤ 增补位，用于表示本段在运行过程中，是否允许扩展，如该段已被增补，则在写回外存时，务必要另外选择外存空间，这是请求分段式管理中所特有的字段；

⑥ 外存起址，用于表示本段在外存中的起始地址，也就是起始盘的块号。

2. 地址转换机构

图 5.20 为分段存储管理中从逻辑地址到物理地址转换的流程图。由于被访问的

段并非都在内存中，所以在进行地址转换时，当发现缺段时就必须先把所缺的段调入内存，然后修改段表，再利用段表进行地址转换，并且在进行地址转换过程中，如发现分段越界则进行相应的处理等。所以，在地址转换机制中又增加了缺段中断处理，分段越界中断处理和分段保护中断处理等。

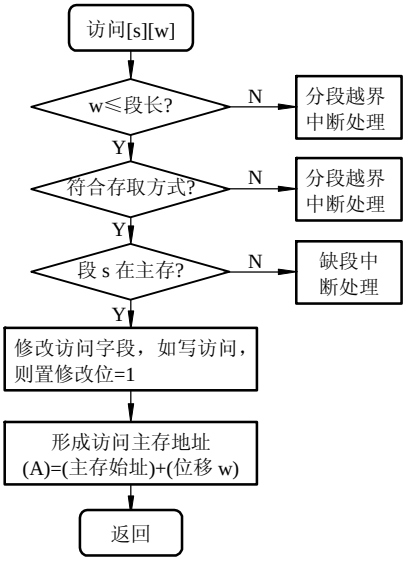


图 5.20　请求分段式的地址转换过程

3. 缺段中断机构

在请求分段系统中，每当进程要访问的段还没有调入内存时，就由缺段中断机构产生一个缺段中断信号，然后，操作系统将由缺段中断处理程序把所需的段调入内存。与缺页中断机构一样，缺段中断机构在一条指令的执行期间产生和处理中断，并且在一条指令执行期间可能产生多次缺段中断。与页不同，段是不定长的，故对缺段中断的处理将比缺页中断的处理复杂得多。请求分段系统中的中断处理过程如图 5.21 所示。

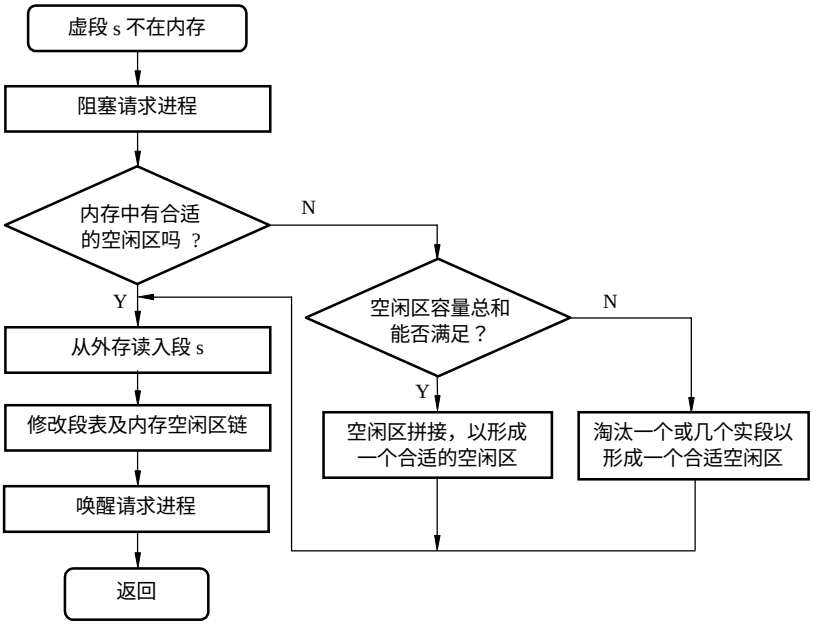


图 5.21　请求分段系统中的中断处理过程

5.4.4　段的共享和保护

段是按逻辑意义来划分的，可以按名存取，所以，段式存储管理可以方便地实现内存信息共享，并进行有效的内存保护。

1. 段的共享

段的共享是指两个以上的作业，使用某个子程序或数据段，而在内存中只保留该信息的一个副本，图 5.22 给出分段系统中共享段的例子。如果用户进程或作业需要共享内存中的某段数据或程序，则只要用户使用相同的段名，就可在新的段表中填入已在内存之中段的起始地址，并设置一定的访问权限，从而实现段的共享。为此，系统应建立一张登记共享信息的表，该表中有共享段段名、装入标志位、内存始址和当前使用该共享段的作业等信息，在作业需要共享段时，系统首先按段名查找共享段信息表，若该段已装入内存，则将内存始址填入该作业的段表中，实现段的共享。

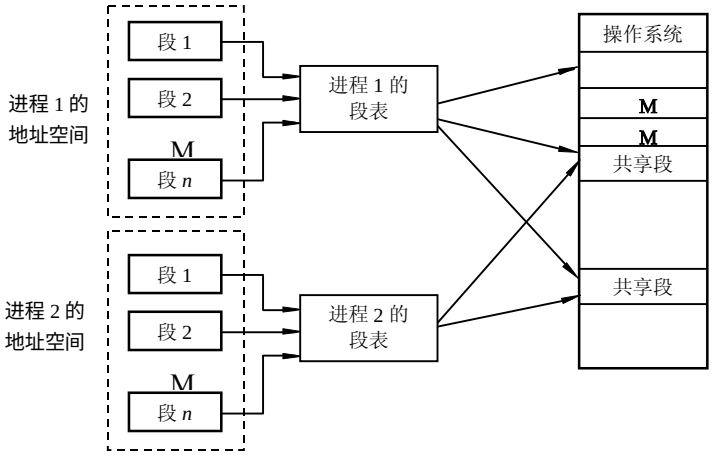


图 5.22　分段系统中段的共享

2. 段的保护

段的保护是为了实现段的共享和保证作业正常运行的一种措施。分段存储管理中的保护主要有地址越界保护和存取方式控制保护。地址越界保护是利用段表中的段长和逻辑地址中的段内相对地址相比较，如段内地址大于段长，则发出地址越界中断，系统便会对段进行保护。这样，各段都限定了一定的空间，并且每个作业也只在自己的地址空间中运行，不会发生一个用户作业破坏另一用户作业空间的危险。不过，有的系统中允许段动态增长，在此系统中段内相对地址大于段长是允许的，为此，段表中设置相应的增补位，以指示该段是否允许动态增长。

分段式存储管理与前面介绍的页式存储管理相比，具有许多优点：

第一，它提供了内外存统一管理的虚存，与请求式分页管理一样，只把当前使用的段调入内存，而其余信息存储在外存，这样可为用户提供超过内存容量的地址空间。不同的是，段式虚存每次调入的是一段有意义的信息，而页式虚存只是调入固定大小的页，这页的信息并不一定完整，从而需要调入多个页面才能把所需完整信息调入内存。

第二，段式管理中允许段长动态增长，这对不断增长新数据的段来说具有很大的优越性。

第三，分段式存储管理便于对具有完整逻辑功能的信息段进行共享。

第四，它还便于实现动态链接。

尽管分段存储管理具有很多优点，但是，它比其他方式需要更多的硬件支持，并且段的长度不等和段的共享与动态增长以及链接等功能都会使系统的复杂性大大增加；另外，段的长度还受内存可用区大小的限制；还有一个缺点就是与分页管理一样，若替换算法选择不恰当就有可能产生抖动现象。

5.5　段页式存储管理

分页存储管理能有效地提高内存的利用率，分段存储管理能很好地满足用户的需要，段页式存储管理则是分页和分段两种存储管理方式的结合，它同时具备了两者的优点。

5.5.1　基本概念

段页式存储管理既方便使用又提高了内存利用率，是目前用得较多的一种存储管理方式，它主要涉及如下基本概念：

① 等分内存。它把整个内存分成大小相等的内存块，称为页，并从 0 起依次编号，称为页号。

② 作业或进程的地址空间。这里采用分段的方式，按程序的逻辑关系把进程的地址空间分成若干段，每一段有一个段名。

③ 段内分页。按照内存页的大小把每一段分成若干个页，每段都从零开始为自己段的各页依次编以连续的页号。

④ 逻辑地址结构。一个逻辑地址的表示由 3 部分组成，段号 s、页号 p 和页内地址 d，记作 v=(s，p，d)，如图 5.23 所示。

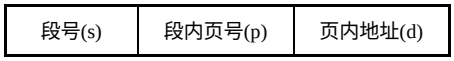


图 5.23　段页式存储管理中的逻辑地址结构

⑤ 内存分配。内存以页为单位分配给每个进程。

⑥ 段表、页表和段表地址寄存器。为

了实现从逻辑地址到物理地址的转换，系统要为每个进程或作业建立一个段表，并且还要为该作业段表中的每一段建立一个页表。这样，作业段表的内容是页表长度和表地址，为了指出运行作业的段表地址，系统有一个段表地址寄存器，它指出作业的段表长度和段表起始地址。图 5.24 示出了段表、页表和内存的关系。

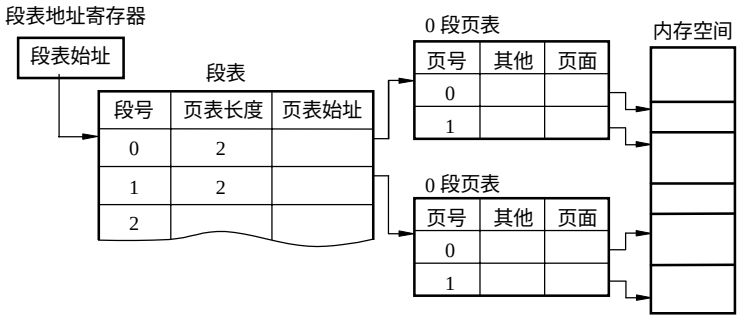


图 5.24　段表、页表与内存的关系

在段页式存储管理系统中，面向物理实现的地址空间是页式划分的，而面向用户的地址空间是段式划分的，也就是说，用户程序被逻辑划分为若干段，每段又分成若干页面，内存划分成对应大小的块，进程映像交换是以页为单位进行的，从而使逻辑上连续的段存入在分散内存块中。

5.5.2　地址转换

在段页式系统中，一个程序首先被分成若干程序段，每一段赋予不同的分段标识符，然后，将每一分段又分成若干个固定大小的页面。段页式系统中地址转换过程如图 5.25 所示。

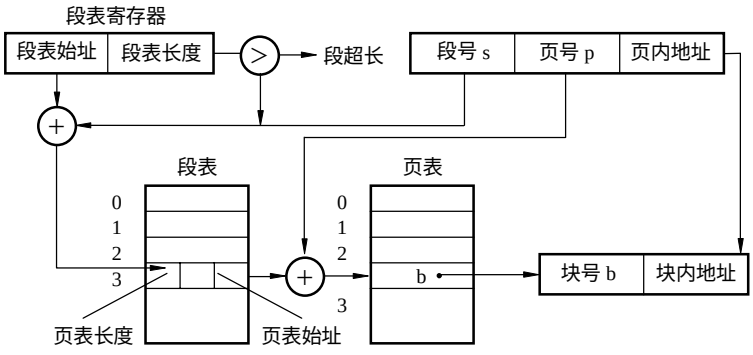


图 5.25　段页式系统中的地址转换机构

段页式系统中的地址转换过程大致如下：

① 地址转换硬件将段表地址寄存器的内容和逻辑地址中的段号相加，得到访问该作业段表的入口地址；

② 将段 s 表中的页表长度与逻辑地址中的页号 p 进行比较，如果页号 p 大于页表长度，则发生中断，否则正常进行；

③ 将该段的页表基地址与页号 p 相加，得到访问段 s 的页表和第 p 页的入口地址；

④ 从该页表的对应的表项中读出该页所在的物理块号 f，再用块号 f 和页内地址 d 组成访问地址；

⑤ 如果对应的页不在内存，则发生缺页中断，系统进行缺页中断处理，如果该段的页表不在内存中，则发生缺段中断，然后由系统为该段在内存建立页表。

5.5.3　管理算法

在地址转换过程中，软、硬件应密切配合，这在分页和分段式存储管理中已体现出来，段页式存储管理也是如此，如图 5.26 所示，其中的中断处理模块的主要功能如下：

① 链接障碍中断。这个模块的功能是实现动态链接，即给每个段一个段号，在相应段表和现行调用表中，为其设置表目，并利用段号改造链接间接字；

② 缺段中断。这个模块的功能是在系统的现行分段表中建立一个表目，并为调进的段建立一张页表，在其段表的相应表目中登记此页表的始址；

③ 缺页中断。这个模块的功能是在内存中找出空闲的存储块，如果没有找到，则调用交换算法，交换内存中的一页到外存，然后调进所需页面到内存，并修改相应的页表表目。

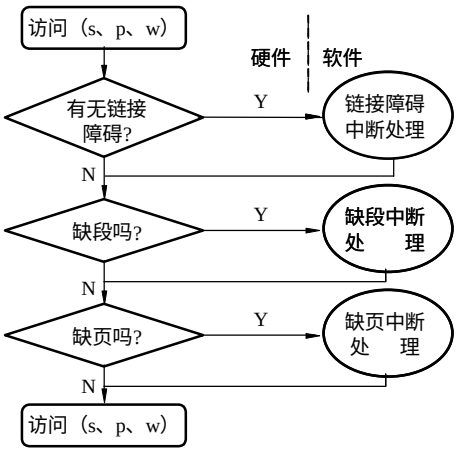


图 5.26　地址转换过程中硬、软件的相互作用

段页式存储管理是分段技术和分页技术的结合，因而，它具备了这些技术的综合优点，即提供了虚存的功能；又因为它以段为单位分配内存，所以无紧缩问题，也没有页外碎片的存在；另外，它便于处理变化的数据结构，段可以动态增长；它还便于共享和控制存取访问权限。

但是，它也有缺点，段页式存储管理增加了软件的复杂性和管理开销，也增加了硬件成本，需要更多的硬件支持；此外，各种表格要占用一定的存储空间；并且与分页和分段一样存在着系统抖动现象；还有，和分页管理一样仍然存在着页内碎

片的问题。

5.6　虚拟内存的置换算法

在进程运行过程中若发生缺页，而此时内存中又已无空闲空间时，为了保护系统的正常运行，系统必须从内存中调整一个页面到交换区中，如果此时这个页面已经被修改过，则它在磁盘上的副本已是最新的，因此，不需要写回，调入的页可以直接覆盖被淘汰的页。虽然可以随机选取一个页面淘汰，但是，如果一个经常被使用的页面被淘汰掉了，就带来了不必要的额外开销，所以，一个好的页面置换算法应具有较低的页面更换频率，从理论上讲，应将那些以后不再会访问的页面换出，或把那些在较长时间内不再访问的页面调出，下面介绍几种常用的置换算法。

5.6.1　先进先出页面置换算法

最早最简单的页面置换算法是先进先出(FIFO)算法，这种算法的实质是，总是选择在内存中停留时间最长的页淘汰，即先进入内存的页先被换出内存。该算法实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指针，使它总是指向最老页面，每次淘汰的均是这个最老的页面。

假定系统为某进程分配了 3 个物理块，并考虑有以下的页面走向：7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1，1，7，0，1。对于这个给定的页面走向，设有 3 个内存块，最初都是空的。下面访问 2，发生缺页，因为 3 个内存块中都有页面，所以要淘汰最先进入的页面 7，接着对页面 0 的访问，由于它已在内存，所以不发生缺页。这样顺次做下去，就产生图 5.27 所示的情况，每出现一次缺页，就示出淘汰后的情况，总共有 15 次缺页。

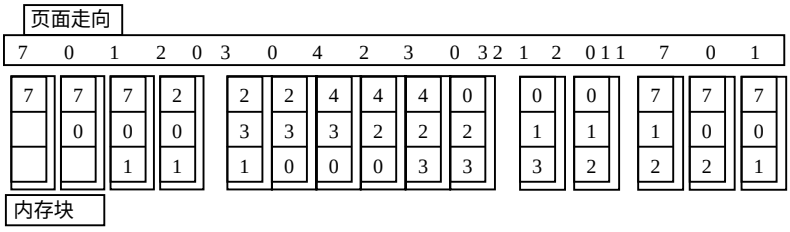


图 5.27　利用 FIFO 页面置换算法时的置换图

FIFO 页面置换算法容易理解和容易进行程序设计，然而它的性能并非总是很好，只有当按线性顺序访问地址空间时才是理想的，否则效率不高。

5.6.2　最佳页面置换算法

最佳(Optimal，OPT)页面置换算法是1966年由Belady在理论上提出的一种算法。它的实质是，当调入一个新的页面必须预先淘汰某个老页面时，所选择的老页面应是将来不再被使用，或者是在很久以后才被使用的页面。采用这种页面置换算法，能保证有最少的缺页率，但无法实现，因为它需要人们预先知道作业整个运行期间的页面走向情况。例如，对于上面给定的页面走向来说，最佳页面置换算法仅出现 9 次缺页中断，如图 5.28 所示。

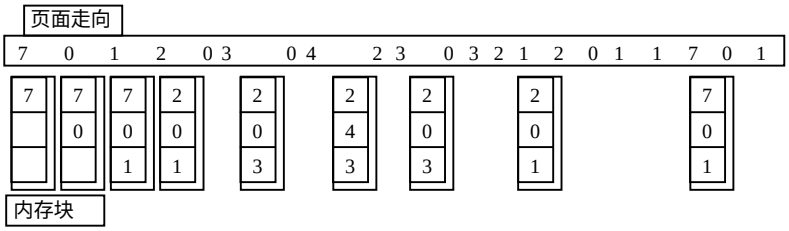


图 5.28　利用 OPT 页面置换算法时的置换图

5.6.3　最近最少使用页面置换算法

由于 FIFO 页面置换算法所依据的条件是各个页面调入内存的时间，实际上，页面调入的先后并不能反映页面的使用情况，所以，FIFO 页面置换算法的性能较差，而最近最少使用(Least Recently Used，LRU)页面置换算法则是根据页面调入内存后的使用情况，选择最近最少使用的页面予以淘汰。

LRU 算法与每个页面最后使用的时间有关，该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当要淘汰一个页面时，就选择现有页面中 t 值最大的那个页面，这种算法需要实际硬件的支持，一般利用一个寄存器记录该页面最后被访问的时间，在淘汰页面时，选择该时间值最大的页面；也可以用一个栈来保留页号，每当访问一个页面时，便将该页面找出放在栈顶，这样栈顶总是放有目前刚使用过的页，而栈底放着目前用得最少的页。

在图 5.29 中，应用 LRU 算法产生 12 次缺页，其中，前 5 次缺页情况和 OPT 算法一样。因为 OPT 是依据以后各页面的使用情况，而 LRU 算法则是根据各页以前的使用情况来判断，访问到第 4 页时，LRU 算法就与 OPT 算法不同。

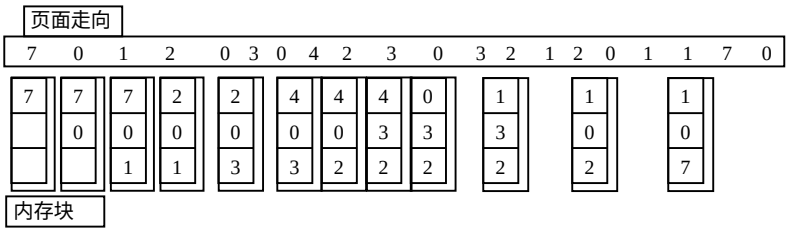


图 5.29　利用 LRU 页面置换算法时的置换图

5.6.4　第二次机会页面置换算法

FIFO 算法可能会把经常使用的页面置换出去，为此，对该算法做一个简单的修改，对最老页面的 R 位进行检查。如 R 位是 0，那么这个页既老又没用，可以被立刻置换掉，如果是 1，就清除掉这个位，并将该页放到链表的尾端，修改它的装入时间使它就像刚装入一样，然后继续搜索。这就是第二次机会(Second Chance)页面置换算法，如图 5.30 所示。图中，页面 1～8 按照进入内存的时间顺序保存在链表中。

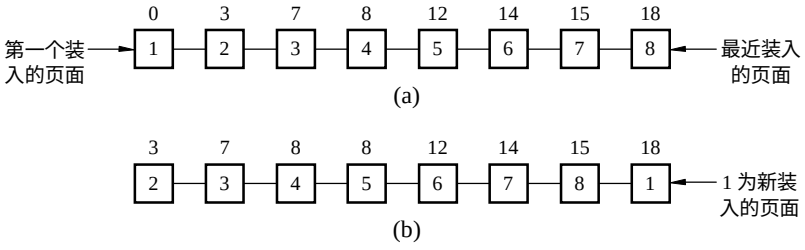


图 5.30　利用第二次机会算法的置换图

(a) 页面按先进先出的方法排列

(b) 在时间 25 发生缺页并且页面 1 的 R 位已经设置时的页面链表

假设在时间 25 发生了一次缺页，此时最老的页面是时间 0 进入内存的页面 1，如果该页面 1 的 P 位是 0，则将它淘汰出内存，或者把它写回磁盘；如果 P 位为 1，则将页面 1 放在链表尾部并重新设置“装入内存时间”为 25，然后清除 R 位，这时对页面的搜索将从页面 2 处开始。

Second Chance 算法所做的是寻找一个从上一次对它检查以来没有被访问过的页面，如果所有的页都被引用了，就是 FIFO 算法。并且设想所有页面的 R 位均为 1，操作系统将一个接一个地把各个页移到链表尾部，与此同时清除这些页的 R 位，最后又回到页 1，这时它的 R 位已为 0 了，因此，页面 1 将被淘汰，这个算法就结束

了。

5.6.5　时钟页面置换算法

由于 Second Chance 算法经常要在链表中移动页面，既降低了效率，又引起许多不必要的操作。一个更好的办法是把所有的页面保存在一个类似钟表面的环形链表中，用一个指针指向最老的页面，就如表针指向某一时刻一样，如图 5.31 所示。

当发生页面故障时，算法首先检查表针指向的页面，如果该页面的 R 位是 0 就淘汰掉这页，并把新页插入这个位置，再把表针前移一个位置；如果 R 位是 1 就清除 R 位并把表针前移一个位置，重复这个过程直到找到一个 R 位为 0 的页为止，它和第二次机会算法的区别仅是实现的方法不同。

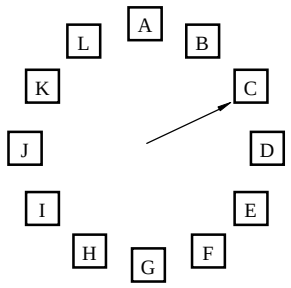


图 5.31　利用时钟页面替换算法的置换图

5.6.6　其他页面置换算法

还有许多用于页面置换的算法，如最近未使用(Not Used Recently，简称 NUR)页面置换算法，页面缓冲算法(Page Buffering Algorithm)，这里对它们仅作简要介绍。

1. 最近未使用置换算法

NUR 算法与 LRU 算法类似，它较 LRU 算法而言，易于实现，开销也比较少。实质是在存储分块表的每一表项中增加了一个引用位，操作系统定期地将它们置为 0，当某一页被访问时，由硬件将该位置 1，在一段时间之后，通过检查这些位可确认自上次置换后哪些页面已被使用过，哪些页面还没有使用过，并把该位是 0 的页淘汰，其算法流程如图 5.32 所示。

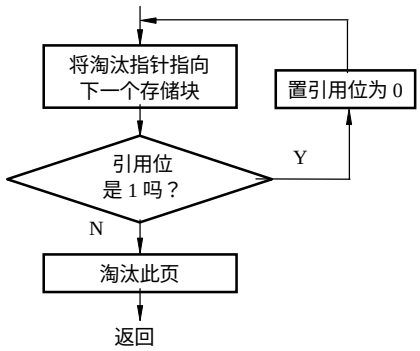


图 5.32　NUR 算法

2. 页面缓冲算法

虽然 LRU 和时钟算法都比 FIFO 算法好，但它们都需一定的硬件支持，开销较大，而且置换一个已修改的页比置换未修改的页的开销要大，而页面缓冲算法则既可改善系统的性能，又可采用一种较简单的置换策略。VAX/VMS 操作系统便是使用的页面缓冲算法。该算法规定将一个被淘汰的页放入两个链表中的一个，即如果页面未被修改，就将它直接放入空闲链表中，否则便放入已修改页面的链表中。

注意：这时页面在内存并不做物理上的移动，而只是将页表中的表项移到上述

两个链表之一中。

空闲页面链表实际上是一个空闲物理链表，其中的每个物理块都是空闲的，因此，可装入程序或数据。当需要读入一个页面时，便可利用空闲物理块链表中的第一个物理块来装入该页，当换出一个未被修改的页时，实际上并不将它换出内存，而是把它所在的物理块挂在自由页链的末尾。类似地，在置换一个已修改的页面时，也将其所在的物理块挂在修改页面链表的末尾。利用这种方式可使已被修改的页面和未被修改的页面都留在内存中，当该进程以后再次访问这些页面时，只需花费较小的开销，就可使这些页面又返回到该进程的驻留集中，当被修改页面达到一定数量时，例如，64 个页面，再将它们一起写回到磁盘，从而显著地减少了磁盘 I/O 的操作次数。

5.7　系统举例

5.7.1　UNIX 系统中的存储管理技术

UNIX 采用了请求分页存储管理技术和交换技术。当进程运行时不必将整个进程的映像都保留在内存中，而只需保留当前要用的页面，当进程访问到某些尚未在内存的页面时，系统把这些页面装入内存，这种策略使进程的逻辑地址空间映射到机器的物理地址空间具有更大的灵活性。下面将简要介绍 UNIX 的交换技术和请求分页技术。

1. 交换

在 UNIX 系统中，内存资源十分紧张，为此引入了交换策略，将内存中处于睡眠状态的某些进程调到外存交换区中，而将交换区中的就绪进程重新调入内存，为实现此种策略，系统内核应具有交换空间的管理、进程换出和进程换入这三个功能。

(1) 交换空间的管理

由于进程在交换区驻留的时间很短，交换操作很频繁，因此，在交换空间的管理中，速度是关键性问题。为此内核应为被换出的进程分配连续空间，而较少考虑存储空间的碎片问题。在 UNIX 系统中分配交换空间使用的数据结构是驻留在内存的交换映射表，此映射表是一个结构数组，每一个结构都包含两个信息(空闲区的始址和空间区的大小)，映射表中的表项按照空闲区的地址从小到大的顺序排列，采用一定的算法对交换空间进行分配，如采用最优适应算法，此算法的空闲区是按位置排列的，空闲块地址小的，在表中的序号也小。当一个进程要换出内存并需在交换区分配一片空间时，便在各空闲区中找一其大小可以满足要求的空闲区。只要找到第一个足以满足要求的空闲区就停止查找，并把它分配出去。当进程换入内存并释放所用的交换区时，要在 map 表的合适位置进行登记，保证表项是按地址排列的。如果该释放区与前面或后面的空闲区相邻接，则把它们合并成一个大区，并修改相

应表目。

(2) 进程的换出

如果内核需要内存空间，就可以把一个进程换到交换区中，当内核决定一个进程适合换出时，先将该进程的每个区的引用计数减 1，然后选出其值为 0 的进程换出。再者，内核还须调用 malloc 过程来申请交换空间，并对该进程加锁，以避免交换进程再次选中该进程予以换出。此后便可开始用户地址空间和交换空间之间的数据传送，交换进程不必将进程的全部虚地址空间写到交换区上，而只需将已分配给该进程的那部分内存中的内容拷贝到交换区上，如图 5.33 所示。若数据传送过程中未出现错误，则调用 mfree 过程释放进程所占内存，并对该进程的进程表项做相应的修改。

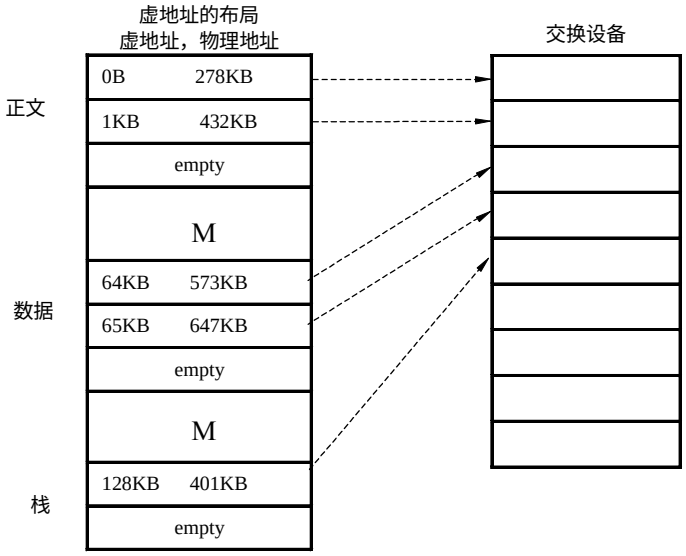


图 5.33　将内存中的内容拷贝到交换区

(3) 进程的换入

引用计数为 0 的进程即为交换进程。每当睡眠进程被唤醒去执行换入操作时，便去查看进程表中所在进程的状态，从中找出“ 就绪且换出 ”状态的进程，把其中换出时间最久的进程作为换入进程，再根据该进程的大小，调用 malloc 为其申请内存，当申请内存成功时，直接将进程换入，否则需先将内存中的某些进程换出，腾出足够的内存空间后再将进程换入。当交换进程成功地换入一个进程后，又重复上述被唤醒的过程，陆续地将一些进程换入，直到所有的“ 就绪且换出 ”的进程已被全部换入或者已无法获得足够的内存来换入进程。

2. 请求分页

UNIX 系统中采用的请求式分页存储管理方式与前面讲过的原理相同。为实现请求分页存储管理，UNIX 系统配置了 4 种数据结构，现对这些数据结构作一简单介绍。

(1) 页表

页表用于将逻辑页号映射为物理页号，页表项包含该页的内存地址、读、写或执行的保护位和一些附加信息位，如有效位、访问位、修改位、复制写位和年龄位。有效位指示该页内容是否有效；访问位指示该页最近是否被进程访问过；修改位说明该页内容是否被修改过；年龄位记录该页成为一进程工作集中一页的时间，即一个页面作为进程工作集中的成员已存在了多久；复制写位说明修改该页时，是否已为之建立一新的拷贝。

(2) 磁盘块描述字

每一个页表项对应一个磁盘块(简称盘块)描述字，其中记录了各虚页的磁盘拷贝的块号。磁盘块描述字对逻辑页面的磁盘副本进行说明，如图 5.34 所示，由图中可以看出，共享一个分区的诸进程可存取共同的页表项和磁盘块描述字，页面内容可以在交换设备的特定块，或者是可执行文件中，也可以不在交换设备上。如果页面在交换设备上，则磁盘块描述字包括该逻辑设备号和页面所在的盘块号；如果页面在一个可执行文件中，则磁盘块描述字包括该页在文件中的逻辑块号。

图 5.34　页表项和磁盘块描述字

(3) 页面数据表

页面数据表用于描述每个物理页，通过页号进行索引，包含页面状态，访问该页面的进程数目，逻辑设备和该页所在的盘块号以及指向在空闲页面链表上和在页面散列队列中的其他页框数据表项的指针。

(4) 交换使用表

交换使用表描述了交换设备上每一页的使用情况，每个在交换设备上的页面在交换使用表中都占用一项，该项表明了有多少表指向交换设备上的一个页面。

图 5.35 示出页表项、磁盘块描述字、页面数据表和交换用计数表之间的关系。其中，一个进程的虚拟地址 1493KB 映射到一个页表项，它指向物理页面 794；该页表项的磁盘块描述字说明该页的副本存于交换设备 1 的 2743 块中。与物理页面 794 对应的页框数据表项也说明了该页存于交换设备 1 的 2743 块中。该虚拟页面的交换用计数值为 1，说明只有一个页表项指向它在盘上的交换副本。

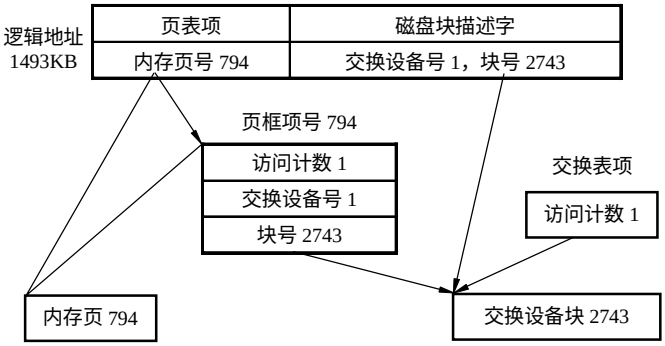


图 5.35　请求分页数据结构间的关系

3. 换页进程

为了支持请求式分页存储管理算法，UNIX 系统设置了一个核心进程，即换页进程，该进程的主要任务是增加内存中所有有效页的年龄和将内存中长期不用的页换出。

(1) 增加有效页的年龄

计算内存页年龄是指每当内存空闲页数低于某规定的下限时，内核便唤醒换页进程，后者检查内存中每一个活动的、非上锁的区，增加所有有效页的年龄。内存中的页有不可换出和可换出两种状态，前者指该页在内存中的年龄尚未到达规定值，后者指年龄已到规定值。一个页可计数的最大年龄取决于具体的实现方法，对于只设置了两位的年龄域，其年龄只能取值 0、1、2 和 3，当有效页的年龄到达时，便可将它换出内存。每当有进程访问了某个页面时，便将该页的年龄置 0。因此，仅当一个页面被连续检查了 3 次，且中间未曾被访问过时，其年龄才可能增至 3，而成为可被换出的页，图 5.36 示出了一个有效页同时被进程访问和被换页进程检查的情况。图中的数字表明，该页的年龄在被换出进程检查两次后，因又被进程访问而为 0，后又被检查一次又因进程访问而降为 0，

图 5.36　某有效页的年龄变化

最后换出进程查得该页已被连续检查三次而将其换出。

(2) 将内存长期不用的页换出

如果多个进程共享一个分区，则相应页面可被多个进程同时使用，只要页面被进程使用，它就应留在内存中，仅当它不被任何进程使用时，它才适于换出。换页进程分两步将页面换出，第一，由换页进程从内存有效页中找出应被换出的页，并将它们链入要换出的页面表中；第二，当换出页面链表已满时，换页进程将它们一起换到设备上。

4. 缺页

在 UNIX 系统中可出现两类缺页，有效缺页和保护性缺页。当出现缺页时，缺页处理程序可能要从盘上读一个页面到内存，并在 I/O 执行期间睡眠。

(1) 有效缺页处理

有效缺页处理，对于进程虚拟地址空间之外的页面和虽在其虚拟地址空间内但当前未在内存的页面，它们的有效位是 0，表示缺页。如果一个进程试图存取这样的一个页面，则导致有效缺页，内核调用有效缺页处理程序，进行相应处理。

其大致处理工作是，如果页面地址在虚拟地址空间之外，则向被中断进程发“ 段越界 ”信号，如果访问的页面不在内存中，则按常规缺页中断处理，为该页分配内存页，把它调入内存。

(2) 保护性缺页处理

第二种类型的缺页是保护性缺页，它是由进程对有效页面存取的权限不符合规定而引起的。例如，某进程试图对正文段进行改写。导致保护性缺页的另一个原因是一个进程想写一个页面，而该页面的复制写位在执行 fork 期间已经置上。内核必须确定导致保护性缺页的原因是上述哪一种。当发生保护性缺页事件后，硬件向其处理程序提供发生事件的虚拟地址，然后由处理程序进行处理，其主要处理工作是，如果与其他进程共享该内存页，则重新分配一个内存页，并把它复制过来，如果没有进程在使用该内页，则淘汰它。

5.7.2　Linux 系统中的存储管理技术

Linux 是多用户多任务操作系统，存储资源可被多个进程有效共享。Linux 的内存管理主要体现在对虚拟内存的管理上，而对虚拟内存管理又体现在大地址空间、进程保护、内存映射、物理内存分配和共享虚拟内存上。Linux 为了实现虚拟内存的这些功能需要各种机制的支持，如地址映射机制，内存分配和回收机制，缓存和刷新机制，请页机制和交换机制，内存共享机制，这几种机制的关系如图 5.37 所示。

内存管理程序首先通过映射机制把用户程序的逻辑地址映射到物理地址，在用户程序运行时如果发现程序中要用的虚地址没有对应的物理内存时，就发出请页要

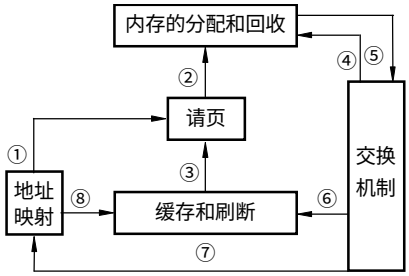


图 5.37　虚存实现机制间的关系

求①;如果有空闲的内存可供可分配,就请求分配内存②(用到了内存的分配和回收)、并把正在使用的物理页记录在页缓存中③(使用了缓存机制)。如果没有足够的内存可供分配，那么就调用交换机制，腾出一部分内存④⑤。另外在地址映射中要通过 TLB(翻译后援存储器)来寻找物理页⑧，交换机制中也要用到交换缓存⑥，并且把物理页内容交换到交换文件中后也要修改页表来映射文件地址⑦。

Linux 中存储管理技术采用的是段页式虚存技术。它将一个进程中的程序、数据、堆栈分成若干“ 段 ”来处理，每段有一个 8 字节的段描述符，指出该段的起始地址、长度和存取权限等，这些段描述符的集合，构成段表，在 CPU 中通过一个寄存器，指出段表的起始位置。

为了便于段长的动态变化，避免段长动态增加时造成周围段内容的移动，Linux 内存管理采用请求页式技术实现。每段又分为若干页，将需要的内容以页面为单位调入内存块中，暂不执行的页面仍留在外存交换区内，以保证比实际内存需求大得多的进程使用内存。

Linux 把指令代码的正文段、数据、堆栈分为若干段，这些段都各有含义。它们符合程序本来的二维结构，为了有效地扩充内存，使进程能在内外存交换。由此，采用了请求式分页存储管理技术，这样既满足了小内存运行大程序的要求，还便于段的增长，由于一段分为若干页后，一段就不必占用一个连续的内存区，而可以把页面见缝插针地放入其内存块中，但是段页式虚存技术的地址转换需要查找段表和页表，势必使读/写速度降低。为了弥补这一缺点，Linux 还采用了“ 快表 ”的方法，把页表内容放在高速缓存 Cache 中，可以加快地址转换的过程。

除此之外,Linux 存储管理还具有如下特点,系统中新建子进程只占两页的内存，一页作新进程的 PCB，一页作新进程的核心栈，此外新建子进程从父进程处仅复制页表；并且 Linux 对数据段和堆栈段采用写时复制的方法。

5.8　小　　结

存储管理在操作系统中占有重要地位，其目的是方便用户和提高内存的利用率。存储管理的基本任务是管理内存空间；进行虚拟地址到物理地址的转换；实现内存的逻辑扩充；完成内存的共享和保护。随着计算机技术的逐步发展，存储器的种类越来越多，按照其容量、存取速度以及在操作系统中的作用，可分为 3 级存储器：高速缓存、内存和外存。

前面介绍的存储管理技术各具特点，在存储分配方式上有静态和动态、连续和非连续之分。所谓静态重定位是指在目标程序运行之前就完成了存储分配，如分区式存储管理和静态分页管理。动态重定位是指在目标程序运行过程中再实现存储分配，如动态分页管理、段式和段页式管理。连续性存储分配要求给作业分配一块地址连续的内存空间，非连续性的是指作业分得的内存空间可以是离散的、地址不连续的内存块，如分页式管理。在将逻辑地址转换成物理地址时，固定分区采用的是静态重定位，其他采用的是动态重定位。静态重定位是由专门设计的重定位装配程序来完成的；而动态重定位是靠硬件地址转换机构来实现的。在实现内存的逻辑扩充方面，分区和静态分页管理采用的覆盖技术和交换技术，请求式分页、段式和段页式管理采用的是虚拟存储技术。覆盖技术就是让不会同时调用的子模块共同使用同一内存区；交换技术就是将作业不需要或暂时不需要的信息交换到外存，腾出内存空间供调入其他所需信息使用；虚拟存储技术是通过请求调入和替换功能，对内外存进行统一管理，为用户提供了似乎比实际内存容量大得多的存储器，这是一种性能优越的存储管理技术。在完成信息共享和保护方面，分区管理不能实现共享，分页管理实现共享较难，但是，分段和段页式管理就能容易地实现共享。分区管理采用的是越界保护技术，其他的均采用越界保护和存取权控制保护相结合的方法。

在进程运行过程中，当实际内存不能满足需求时，为释放内存块给新的页面，需要进行页面置换，有很多种页面置换算法供使用。FIFO 是最容易实现的，但性能不是很好；OPT 仅具有理论价值；LRU 是 OPT 的近似算法，但是实现时要有硬件的支持和软件开销。多数页面置换算法，如最近未使用置换算法等都是 LRU 的近似算法。

在大型通用系统中，往往把段页式存储管理和虚拟存储技术结合起来，形成带虚拟存储的段页式系统，它兼顾了分段存储管理在逻辑上的优点和请求式分页存储管理在存储管理方面的长处，是最通用、最灵活的系统。

UNIX 采用了交换和请求式分页存储管理的技术，页面淘汰采用 LRU 算法，为了实现交换和请求式分页，系统设立了很多数据结构，便于各分区的共享和保护。Linux 操作系统则采用的是段页式虚拟存储技术。

习　题　五

5.1　解释下列概念：物理地址，逻辑地址，逻辑地址空间、内存空间。

5.2　什么叫重定位？区分静态重定位和动态重定位的根据是什么？

5.3　什么是虚拟存储器？它有哪些基本特征？

5.4　什么是分页？什么是分段？二者有何主要区别？

5.5　纯分页技术和请求式分页技术之间的根本区别是什么？

5.6　请画出页式动态地址转换机构，并说明地址转换过程。

5.7　某虚拟存储器的用户编程空间共 32 个页面，每页为 1KB，内存为 16KB，假定某时刻一用户页表中已调入内存页面的页号和物理块号的对照表如下：

页　　号	物理块号
0	7
1	10
2	4
3	5

则逻辑地址 OA5C(H)所对应的物理地址是多少？

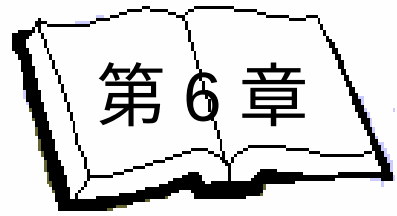
5.8　何谓系统的“抖动”现象？你认为应该采取何种措施来加以避免？

5.9　为什么与分页技术相比分段技术更容易实现程序和数据的共享和保护？

5.10　考虑下述页面走向：

4、3、2、1、4、3、5、4、3、2、1、5

当内存块数量分别为 3 和 4 时，试问 LRU、FIFO、OPT 这 3 种置换算法的缺页次数各是多少(假设所有内存块起初为空)？



处理机调度

在一个多道程序系统中，主存同时存在多个进程。每个进程在两种状态之间转换：要么占用处理机，要么等待 I/O 执行或等待其他事件发生。处理机忙于执行一个进程而其他进程只有等待。

调度是多道程序的关键。事实上，有 4 种类型的调度(见表 6.1)。其中，I/O 调度将在第 7 章讨论。另外 3 种就是处理机调度。

本章先对这 3 种处理机调度进行分析。其中，长程调度和中程调度主要取决于多道程序的度，本章只集中讨论短程调度。然后，介绍短程调度中用到的各种算法，再讨论多处理机调度，并介绍在设计多处理机线程调度时有什么不同之处。最后将讨论实时调度，首先介绍实时进程的特性，然后介绍进程调度的本质及实时调度的方法。

表 6.1　调度类型

调度类型	调度内容
长程调度	决定欲增加执行的进程池
中程调度	决定增加部分或全部位于内存中的进程数
短程调度	决定哪个就绪进程被处理机执行
I/O 调度	决定哪个进程未完成的 I/O 请求被可用 I/O 设备处理

6.1　调度类型

处理机调度的目的是使处理机在满足系统要求的响应时间、吞吐量和处理机利用率的前提下及时地运行进程。在许多系统中，调度被分成 3 种：长程、中程和短程调度。

图 6.1 是进程状态转换图。当产生一个新进程时，就执行长程调度，将新进程加到一组活动进程中。中程调度是替换功能的一部分，它将一个新进程的部分或全部调入内存以便执行。短程调度真正决定哪一个就绪进程将在下次执行。图 6.2 为重新组织了的进程状态转换图，它可以说明调度作用的嵌套关系。

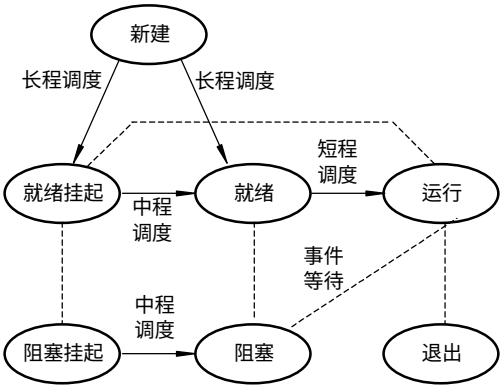


图 6.1　调度和进程状态转换

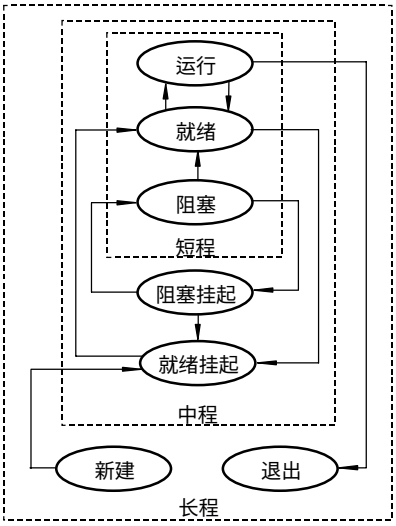


图 6.2　调度层次

由于调度决定了哪些进程将等待、哪些进程被执行，所以直接影响到系统的执行效率。图 6.3 给出了一个进程在状态转换中所涉及的队列。从根本上讲，调度就是要使队列延迟最小，并优化系统的执行效率。

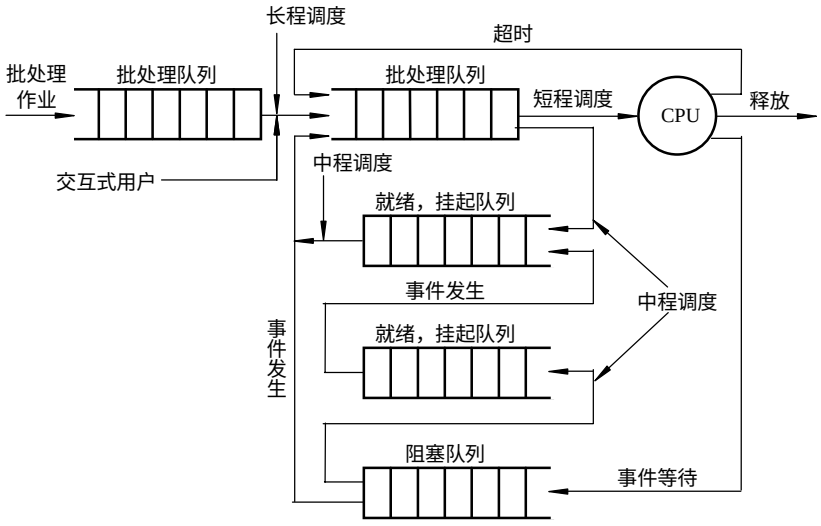


图 6.3　调度排队图

6.1.1　长程调度

长程调度决定哪些程序被系统接纳处理，因此，它控制多道程序的度。一旦一个作业或用户程序被接纳，就成为一个进程并被加到短程调度的队列中。在有些系统中，一个新创建的进程开始处于换出状态，于是它就被加到中程调度的队列中。

在一个批处理系统或一个一般操作系统的批处理部分，新递交的作业先驻留在磁盘上，进入批处理队列。需要时长程调度就会从该队列中创建进程。这时，长程调度还必须决定系统能承担一个还是多个进程，哪一个或哪一些作业被接纳并且变成进程。

什么时候创建新进程主要取决于多道程序的度。创建的新进程越多，每个进程执行时间所占百分比就越小。因此，长程调度限制多道程序的度以便为当前进程提供满意的服务。每当一个作业结束，调度会决定增加一个或多个新作业。另外，当处理机空闲时间超过阈值时，也会执行长程调度。

决定接纳哪个作业可用简单的先来先服务算法，同时它可用来管理系统运行。

在一个分时系统交互式程序中，用户被接纳进入系统时就会产生一个请求进程。分时用户不是简单地排队等待系统运行，操作系统会接纳所有的授权进程直到系统饱和。这时连接请求会收到一个系统已满请稍后再试的信息。

6.1.2　中程调度

中程调度是替换功能的一部分，这在 5.6 节已讨论过。

6.1.3　短程调度

长程调度相对而言执行得较少，并且它只是粗略决定是否启用一个新进程，启用哪一个进程。中程调度较频繁地进行替换。短程调度执行得最频繁，它决定下一步执行哪个进程。

当一个可能导致当前进程中断或者使另一个进程抢占正在运行的进程的事件发生时，就会执行短程调度。例如，有以下事件发生时，就执行短程调度：时钟中断；I/O 中断；OS 请求；信号。

6.2　调度算法

6.2.1　短程调度标准

短程调度的主要目标是，以使系统性能得到优化的方法来分配处理机时间。

1. 通常使用的标准

通常使用的标准可分为两类：面向用户的标准和面向系统的标准。

① 面向用户的标准与单个用户或进程关心的系统性能有关，如交互式系统的响应时间。响应时间是指提交一个请求到系统响应的时间，这是用户所关心的。我们希望调度策略能对多个用户提供“优质”服务，就响应时间而言，必须定义一个阈值，比如说 2s。这样，调度策略的目标之一就是，尽可能容纳更多的响应时间不超过 2s 的用户。

② 面向系统的标准是为了使系统高效地运行。例如，吞吐量，它是指完成进程的速率，希望它越大越好。但这主要是为系统效率考虑，很少为用户设备考虑。因此，它主要与系统管理员有关，与用户数量无关。

面向用户的标准对所有系统都很重要，但面向系统的标准在单用户系统中并不重要。在单用户系统中，处理机的高效使用或高吞吐量并不像系统对用户应用程序的响应那么重要。

2. 与性能相关的标准

根据所面向的对象是否与性能相关，可将其分为与性能有关的标准和与性能无关的标准。与性能有关的标准是可定量的，如响应时间和吞吐量。与性能无关的标准是定性的，如预测性。从某种程度上说，后者可用一个工作负载函数来衡量，但不像测量吞吐量和响应时间那样直接。

表 6.2 总结了主要的调度标准。它们是相互独立的，不能同时优化。例如：提

表 6.2　调度标准

面向用户的与性能有关的标准	
响应时间	对于一个活动进程，这是指发出请求到收到响应的时间间隔。通常，一个进程在它处理请求时就可以产生一些结果给用户。因此，从用户角度看，它比轮转时间要好。调度策略应尽量降低响应时间，并且使其响应时间可接受的活动用户数最大化。
轮转时间	这是指提交一个进程到其完成的时间片段，包括互斥运行时间，等待资源(包括处理机)的时间。这是对批处理作业的一个适合衡量。
期限	当指定进程到达期限时，调度策略应综合其他目标以满足尽可能多的期限要求。
面向用户的其他标准	
预测性	不管系统的负载如何，一个给定的作业应运行同样的时间，有同样的耗费。响应时间或轮转时间变化很大，意味着系统负载变化很大，或系统有必要调整其不稳定性。
面向系统的与性能有关的标准	
吞吐量	调度策略应使单位时间内完成的进程数尽可能多。这是对完成的工作量进行衡量。显然，这取决于进程的平均长度，同时受调度策略的影响。
处理机利用率	这是指处理机忙所占的时间百分比。对共享系统，这个标准很重要。在单用户系统和其他系统（如实时系统）中，这个标准不如其他标准重要。
面向系统的其他标准	
公平性	在没有用户的指导或其他系统支持的情况下，应同等对待每一个进程，并且不应有进程饥饿。
优先权	若给进程分配优先权，调度策略先调度高优先权进程。

资源平衡	调度策略应使系统资源忙，使用不紧张资源的进程有优先权。这个标准涉及中程调度和长程调度。

供满意的响应时间需要频繁切换进程，这样会加大系统开销并减小吞吐量。因此，在设计调度管理策略时应均衡各方面的要求，根据其性质和对系统的使用对各种要求进行加权。

在大多数交互式 OS 中，不管是单用户还是分时系统，都要求有适当的响应时间。这将在本章附录中作深入研究。

现在讨论其他的调度策略，并假定只有一个处理机的情况。

6.2.2　优先权的使用

调度可基于优先权。在很多系统中，每个进程都有一个优先权，高优先权的进程比低优先权的进程优先运行。

图 6.4 说明了优先权的使用。为了便于理解，简化了排队图，省略了大量的阻塞队列和挂起状态队列。用一个按优先权降序排列的队列代替单等待队列 RQ_0，RQ_1，…，RQ_n，其中优先权［RQ_i］＞优先权［RQ_j］，$i<j$。当进行调度选择时，先从最高优先权的就绪队列(RQ_0)开始。如果队列中有一个或更多的进程，则按其他调度算法选一个进程。如果 RQ_0 空，则检查 RQ_1，如此下去。

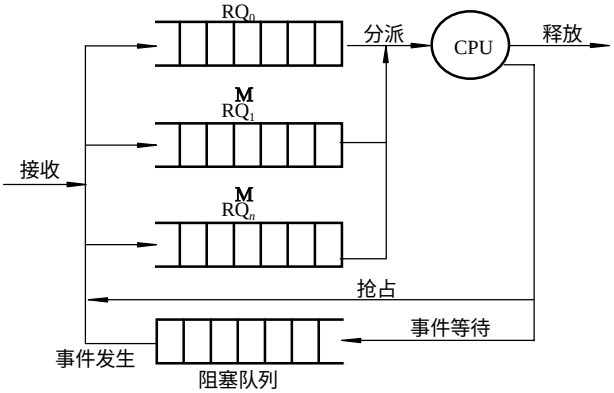


图 6.4　调度优先级

完全按优先权调度会出现的问题是，低优先权进程可能饥饿，当有高优先权的进程流持续到达时，就会出现这种情况。为解决这个问题，进程优先权可随其年龄或执行历史改变，稍后会给出一个例子。

6.2.3　调度策略

表 6.3 归纳了本小节所要讨论的各种调度策略的特性。前两个是按选择函数和

决定模式对调度策略进行分类的。

表 6.3　各种调度策略的特性

调度策略	先来先服务 FCFS	时间片轮转 RR	最短进程优先 SPN	最短剩余时间优先 SRT	最高响应比优先 HRRN	多级反馈队列 FB
选择函数	max(w)	常数	min(t_s)	min(t_s−t_e)	$\max\left(\frac{t_w+t_s}{t_s}\right)$	(见文中)
决定模式	非抢占	抢占(时间片)	非抢占	抢占(到达时)	非抢占	抢占(时间片)
吞吐量	不定	时间片很小时，吞吐量可能很低	高	高	高	不定
响应时间	可能很高，尤其是进程执行时间变化很大时	对短进程提供好的响应时间	对短进程提供好的响应时间	提供较好的响应时间	提供较好的响应时间	不定
耗费	最小	低	可能高	可能高	可能高	可能高
对进程的影响	对偏重于 I/O 的进程不利	公平对待	对长进程不利	对长进程不利	平衡性好	有利于偏重 I/O 的进程
饥饿	无	无	可能	可能	无	可能

① 选择函数决定哪个就绪进程在下一次执行。函数可基于优先权、资源需求或进程的执行特性。对于后者，以下 3 个量比较重要：

• t_w 为到目前为止，进程位于系统内的时间，包括等待和执行的时间。

• t_e 为到目前为止，进程执行的时间。

• t_s 为进程所要求的总服务时间，包括 t_e。

例如，选择函数 $f(w)=w$ 意味着采用的是先进先出(FIFO)的策略。

② 决定模式指出选择函数被执行的确切时间，分为两类：

• 非抢占式。在这种情况下，只要进程处于运行态，它就一直执行直到终止、I/O 阻塞或请求其他的 OS 服务。

• 抢占式。OS 中断当前运行进程，将其变成就绪状态。抢占式的决定可以在到达一个新进程、将阻塞进程变为就绪状态的中断发生时，或时钟中断期间进行。

抢占策略比非抢占的开销要大，但它给所有进程提供了更好的服务，因为它防止了一个进程独占处理机时间过长。另外，它可以通过有效的现场切换机制(尽量使用硬件)和一个大的主存以容纳多的程序来降低其开销。

下面以表 6.4 所示的进程集作为范例，讨论不同的调度策略。

表 6.4 所示的进程可看成批处理的作业，

表 6.4　供讨论调度策略所用的进程集

进程	到达时间/s	服务时间/s
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

服务时间是其所需的总执行时间。另外，认为这些进程要求交替使用处理机和 I/O。在 I/O 中，服务时间是指一次循环中的处理时间，在排队模型中，这两种情况的服务时间就是服务时间。

最简单的调度策略是先来先服务(FCFS)，或先进先出(FIFO)。当一个进程处于就绪状态，它就进入就绪队列。若当前进程停止运行，就从就绪队列中选最“老”的进程运行。

图 6.5 给出了该进程集一次循环的执行模式(采用不同的调度策略)，表 6.5 给出了一些主要结果。每个进程的完成时间是确定的，从而可以得到轮转时间。在排队模型中，轮转时间指排队时间或总时间，也就是一个项目位于系统内的时间(等待时间加上服务时间)，较实用的是标准化轮转时间，就是轮转时间和服务时间的比值，这个值指出了一个进程的相对延迟。一般来说，进程执行的时间越长，所允许的绝对延迟时间就越长。标准化轮转时间最小为 1.0；大于 1.0 表示服务的降级。

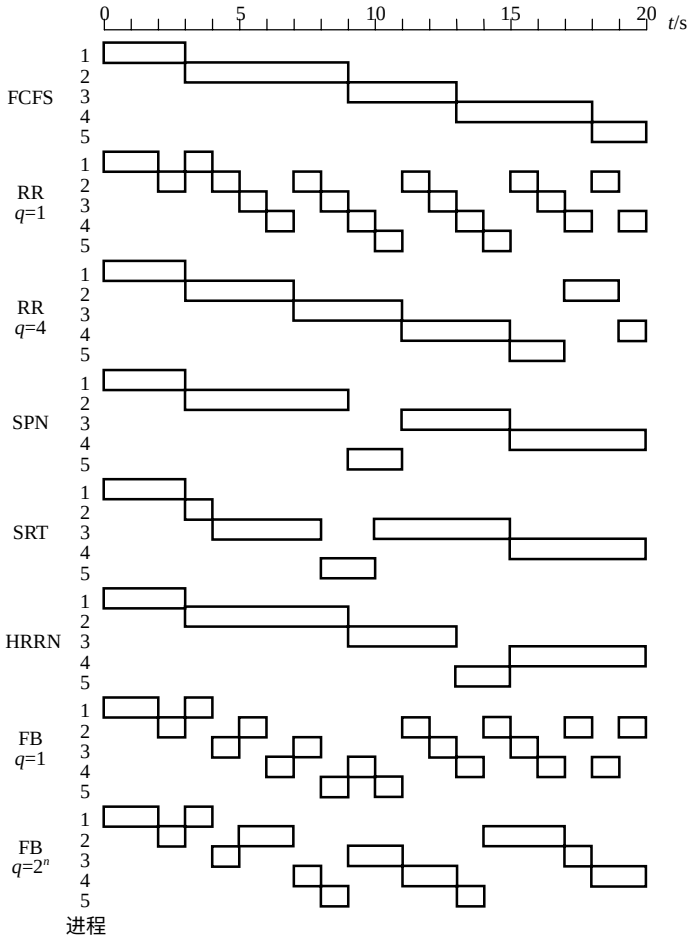


图 6.5　调度策略比较

表 6.5　调度策略的比较

	进　　程	1	2	3	4	5	
	到达时间/s	0	2	4	6	8	平均值
	服务时间/s	3	6	4	5	2	
FCFS	完成时间/s	3	9	13	18	20	
	轮转时间/s	3	7	9	12	12	8.60
	t_q/t_s	1.00	1.17	2.25	2.40	6.00	2.56
RR(q=1)	完成时间/s	4	19	17	20	15	
	轮转时间/s	4	17	13	14	7	11.00
	t_q/t_s	1.33	2.83	3.25	2.80	3.50	2.74
RR(q=4)	完成时间/s	3	19	11	20	17	
	轮转时间/s	3	17	7	14	9	10.00
	t_q/t_s	1.00	2.83	1.75	2.80	4.50	2.58
SPN	完成时间/s	3	9	15	20	11	
	轮转时间/s	3	7	11	14	3	7.60
	t_q/t_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT	完成时间/s	3	15	8	20	10	
	轮转时间/s	3	13	4	14	2	7.20
	t_q/t_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN	完成时间/s	3	9	13	15	20	
	轮转时间/s	3	7	9	9	12	8.00
	t_q/t_s	1.00	1.17	2.25	1.80	6.00	2.44
FB(q=1)	完成时间/s	4	20	16	19	11	
	轮转时间/s	4	18	12	13	3	10.00
	t_q/t_s	1.33	3.00	3.00	2.60	1.50	2.29
FB（q=2^n）	完成时间/s	4	17	18	20	14	
	轮转时间/s	4	15	14	14	6	10.60
	t_q/t_s	1.33	2.50	3.50	2.80	3.00	2.63

注：t_s 表示服务时间；t_q 表示轮转时间；t_q/t_s 为标准化轮转时间。

1. 先来先服务(FCFS)策略

FCFS 策略更适于长进程，如表 6.6 所示。

进程 C 的标准化等待时间是不能容忍的：它位于系统内的总时间是它所需服务时间的 100 倍，无论何时，一个长进程到后，一个短进程就会出现较长的等待。另一种极端情况尚可容忍，进程 D 的轮转时间差不多是 C 的两倍，但它的标准化等待时间小于 2.0。

表 6.6　FCFS 策略适于长进程的例子

进　　程	到达时间/s	服务时间/s	开始时间/s	完成时间/s	轮转时间/s	t_q/t_s
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	199
平均值					100	26

FCFS 策略的另一个问题是它有利于偏重 CPU 的进程，不利于偏重 I/O 的进程。考虑这样一个进程集，其中，一个进程基本上使用 CPU(偏重 CPU)，有许多偏重于 I/O。当偏重于 CPU 的进程正在运行时，所有偏重于 I/O 的进程必须等待，其中有些可能处于 I/O 队列中(阻塞状态)，甚至在偏重于 CPU 的进程还在执行时就移到了就绪队列中。这样，大多数或所有的 I/O 设备都空闲，实际上它们还有许多工作要做。在当前运行进程离开运行态时，就绪的偏重 I/O 的进程很快就通过运行态，被阻塞在 I/O 事件上。如果偏重于 CPU 的进程也被阻塞，那么处理机就空闲了。因此，采用 FCFS 策略往往不能充分利用处理机和 I/O 设备。

FCFS 策略在单处理机系统中不是一个好方法，但常将它与优先权策略结合起来，以提供一个有效的调度算法。因此，调度时可以有多个队列，每个队列有一个优先权，队列内按 FCFS 策略调度，如后面要讨论的“多级反馈队列调度”。

2. 时间片轮转(Round-Robin，RR) 策略

解决 FCFS 策略不利于短作业的一个简单方法就是基于时钟抢占式调度，如时间片轮转(RR) 策略。周期性产生一个时钟中断。中断时，将当前运行进程放入就绪队列中，按 FCFS 策略选下一个就绪进程运行。这种方法也叫做“时间片轮转”，其中每个进程在被抢占前运行一段时间。

对 RR 策略，关键在于设计时间片的长度。如果时间片太短，那么，短进程相对而言很快通过系统。然而，这在处理时钟中断和进行调度及分派时有开销。因此，要避免使用太短的时间片。一个有效的原则是：时间片应比典型交互所需时间稍长。如果比它小，大部分进程就至少需要两个时间片。图 6.6 描绘了这种要求对响应时钟的影响。如果时间片大于所有进程所需的运行时间，那么，RR 策略就退化为 FCFS 策略。

图 6.5 和表 6.4 显示了分别使用 1 和 4 个时间单位的时间片的结果。最短进程 5 在两种情况下都得到了改善。图 6.7 给出了以时间片大小为参数的函数曲线。这里，时间片的大小对执行效率影响不大，但在一般情况中要仔细选择时间片大小。

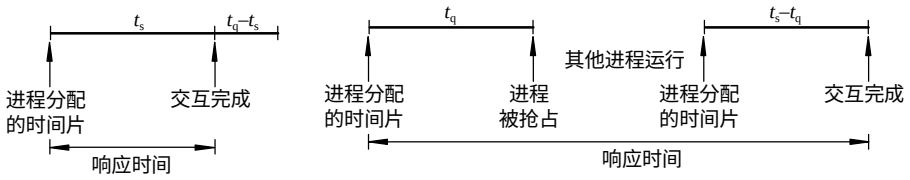


图 6.6　抢占时间片大小的影响

(a) 时间片大于典型交互所需时间　(b) 时间片小于典型交互所需时间

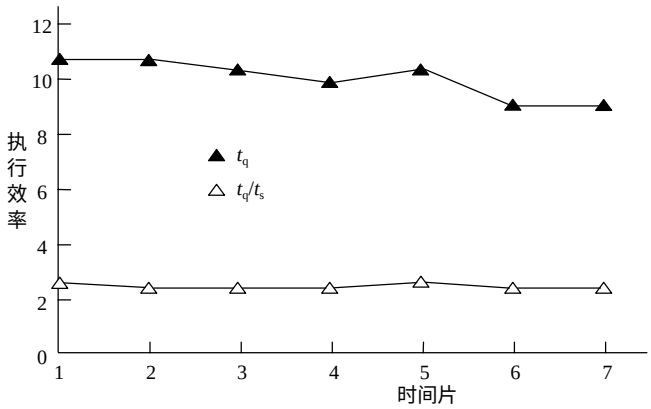


图 6.7　RR 策略的性能

RR 策略对于一般目的的分时系统或事务处理系统非常有效。但有一个不足，就是它对偏重 CPU 的进程和偏重 I/O 的进程有不同的处理结果。通常，一个偏重 I/O 的进程比偏重 CPU 的进程需要较少的 CPU 时间(两次 I/O 操作之间的时间)。如果存在这两种进程，就会出现下面的情况：一个偏重 I/O 的进程使用处理机一小段时间后，被阻塞等待 I/O；等到 I/O 操作完成之后，它进入就绪队列。相反，一个偏重 CPU 的进程执行时通常占用一个完整的时间片，并且立即进入就绪队列。这样，偏重 CPU 的进程获得的处理机时间多一些，从而引起偏重 I/O 进程的执行效率低，I/O 设备利用率不高，响应时间变化大。

可用虚拟时间片轮转(VRR)策略来避免上述缺陷，如图 6.8 所示。新进程到达后，进入基于 FCFS 策略的就绪队列；正运行的进程的时间片结束时也进入就绪队列。当一个进程因 I/O 阻塞时，它进入 I/O 队列。这与原方法一样。新加入的特性是附加一个 FCFS 策略队列来收集从 I/O 等待中释放的进程。调度时，附加队列比就绪队列有更高的优先权。当一个进程从附加队列调入时，其运行时间不应超过基本时间片减去它上次从就绪队列调入运行的总时间。实验表明，这种方法比 RR 策略在公平性上要好些。

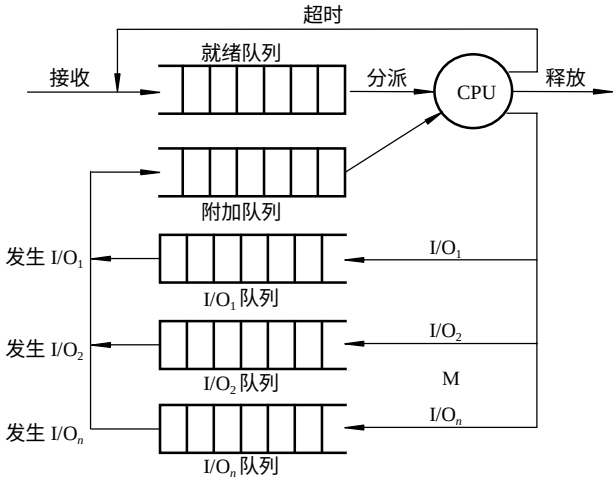


图 6.8　VRR 调度策略的排队图

3. 最短作业优先(Shortest Process Next，SPN)策略

最短作业优先(SPN)策略是一种非抢占式的方式，它首先运行期望运行时间最短的进程。这样，短进程就排到了长进程的前面。

从图 6.5 和表 6.5 的结果可以看出，进程 5 比在 FCFS 中较早得到服务，响应时间也有所改善。但是，响应时间变化幅度加大了，尤其是长进程，这就导致了可预测性的下降。

SPN 策略的一个缺陷是必须预先估计每个进程所需的处理时间。对批处理作业，系统可能要求每个程序员估计处理时间并提交给系统。如果估计时间小于运行时间，会造成系统中途终止该作业的运行。

SPN 策略的一个风险是当能持续提供较短进程时，长进程可能会饥饿。另一方面，虽然 SPN 策略有利于短作业，但它不适于分时系统或事务处理环境，因为它不是抢占式的。对于在 FCFS 策略中所讨论的例子，尽管进程 A、B、C、D 执行顺序相同，但仍对短进程 C 不利。

4. 最短剩余时间优先(Shortest Remaining Time，SRT)策略

最短剩余时间优先(SRT)策略是抢占式的 SPN 策略，其中，总是优先运行期望剩余时间最短的进程。当一个进程进入就绪队列时，它的剩余时间可能比当前运行进程的要短。于是，每当新进程进入就绪队列时，就会进行抢占式调度。和 SPN 策略一样，根据执行时间来选择进程执行可能会引起长进程的饥饿。

SRT 策略一方面不像 FCFS 策略那样有利于长进程，也不像 RR 策略要产生额外中断，从而减少了开销；另一方面，它要记下已执行的时间，又增加了开销。因为一个短作业比当前运行的长作业有优先权，因此，SRT 策略轮转时间性能比 SPN

策略好。

在前例中，3 个最短进程都获得了立即服务，其标准化轮转时间均为 1.0。

5. 最高响应比优先(Highest Response Ratio Next，HRRN) 策略

在表 6.5 中用到了标准化轮转时间。对每个进程，我们希望最小化这个比值，同时也希望所有进程的平均值最小。虽然这是一个经验标准，但我们希望接近它。考虑下面的响应比：

$$\mathrm{HRR}=\frac{t_{\mathrm{w}}+t_{\mathrm{s}}}{t_{\mathrm{s}}}$$

其中，t_{w}=等待处理机的时间；t_{s}=期望的服务时间。

响应比 HRR 等于标准化的轮转时间。在一个进程第一次进入系统时，其响应比 HRR 为最小值 1.0。

这样，调度策略就变为：在当前进程完成或挂起时，从就绪进程中选择具有最高响应比的进程。这个策略的好处在于它考虑了进程的等待时间，有利于短作业，但是，由于长作业的响应比随等待时间的增加而增大，因此最终也将获得处理。

和 SRT 策略及 SPN 策略一样，在用 HRRN 策略之前要事先估计期望的服务时间。

6. 多级反馈队列(Feedback,FB) 策略

如果未指明进程长度，那么 SPN，SRT 和 HRRN 策略都无法使用。另一种有利于短作业的策略是使那些已运行很长时间的作业处于不利地位。换句话说，如果不知道还需要多少时间来运行某进程，那么，就考虑已执行的时间。

该策略如下所述。调度是抢占式的，并采用动态分配优先权。当一个进程第一次进入系统时，它被置入队列 RQ_0(见图 6.4)。在它第一次执行完回到就绪态时，被置入队列 RQ_1。依此类推，每一次执行完，它就被置入下一级具有较低优先权的队列中，从而使一个短进程可以很快完成，无需进入较深层的队列，较长的进程将逐级下传。因此，较新的、较短的进程比较老的、较长的进程有利。在每个队列内部，除了最低优先权的队列外都采用 FCFS 策略，在最低优先权的队列中，进程不能继续下传而是返回队列尾部自行完成。因此，这个队列采用的是 RR 策略。

图 6.9 用一个进程通过各种队列的路径解释了 FB 策略。该策略称为多级反馈，是指操作系统分配给处理机一个进程，当进程阻塞或被抢占时，将它加入到其他优先权的队列中。

这种策略有许多变形。一种简单的方法是按时间间隔抢占，与 RR 策略类似，前例中就是用一个单位时间(见图 6.5 和表 6.5)，这种情况有些类似时间片为一个单位时间的 RR 策略。

这个简单策略的一个突出问题是长进程的轮转时间惊人地变长。事实上，当系统中总有新作业到达时，长作业可能会饥饿。为避免这种情况，可以针对不同的队

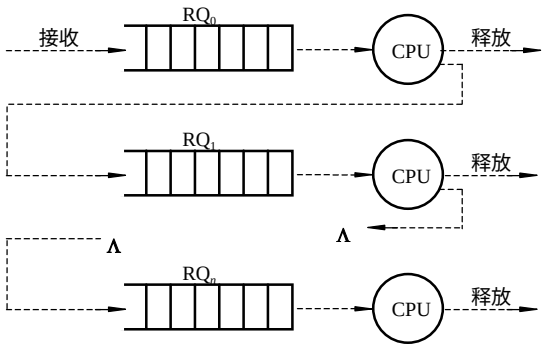


图 6.9　FB 调度策略

列采用不同的抢占时间：RQ_1 中进程允许执行一个单位时间然后被抢占；RQ_2 中进程允许执行两个单位时间后被抢占，如此类推。这样，RQ_i 中进程在被抢占前允许执行 i 个单位时间，如图 6.5 和表 6.5 所示。

即使允许在低优先级的队列中的进程占较多的时间，一个长进程仍可能饥饿。一个解决方法是一个进程在当前队列中等待了一定时间后将其提升到优先级较高的一级队列中。

6.2.4　性能比较

在选择调度策略时要考虑其性能，但不可能进行绝对的比较。因为性能取决于多方面的因素，包括各进程服务时间的可能分布，调度和切换机制的效率，以及 I/O 命令的特性和 I/O 子系统的性能。尽管如此，还是要从中抽取一般的结论。

这里主要利用基本的排队公式，并假设进程到达率满足泊松分布，服务时间满足指数分布。

首先，在服务时间相互独立时，选择下一个服务对象应满足下面的关系：

$$\frac{t_q}{t_s}=\frac{1}{1-\rho}$$

其中，t_q 为轮转时间(位于系统中的总时间，等于等待时间加上执行时间)；t_s 为平均服务时间(处于运行态的平均时间)；ρ=处理机利用率。

基于优先权的调度，其中每个进程被赋予了一个与估计运行时间无关的优先权，那么，它和 FCFS 策略有相同的平均轮转时间和平均标准化轮转时间，而且抢占或非抢占对这些平均值无影响。

除了 RR 和 FCFS 策略外，其他的调度策略在选择时需要估计运行时间，然而无法建立一个精确的分析模型，因此，只能通过与 FCFS 策略比较，了解基于服务时间的调度算法的性能。

如果调度基于优先权，并且进程优先权是以运行时间为基础的，则会出现不同的情况。表 6.7 列出了一些公式，表明了有两个优先权的进程，并且每个优先权有不同的运行时间的结果。这些结果可以推广到多个优先权的情况，抢占式与非抢占的公式有所不同。在抢占式调度算法中，当有一个高优先权进程就绪时，低优先权的进程被抢占，停止运行。

表 6.7　有两个优先权的单服务器队列公式

假设　1. 泊松到达率；　　2. 优先权为 j 的项目在优先权为 j+1 的项目前运行；
3. 运行时进程不被中断；　　4. 优先权相等时按 FCFS 调度策略；
5. 没有项目丢失。

(a) 一般公式	(b) 非中断：	(c) 抢占：
$\lambda = \lambda_1 + \lambda_2$ $\rho_1 = \lambda t_{s1}; \rho_2 = \lambda t_{s2}$ $\rho = \rho_1 + \rho_2$ $t_s = \frac{\lambda_1}{\lambda} t_{s1} + \frac{\lambda_1}{\lambda} t_{s2}$ $t_q = \frac{\lambda_1}{\lambda} t_{q1} + \frac{\lambda_2}{\lambda} t_{q2}$	服务时间呈指数分布 $t_{q1} = 1 + \frac{\rho_1 t_{s1} + \rho_1 t_{s2}}{1 - \rho_1}$ $t_{q2} = 1 + \frac{t_{q1} - 1}{1 - \rho_1}$	服务时间呈指数分布 $t_{q1} = 1 + \frac{\rho_1 t_{s1}}{1 - \rho_1}$ $t_{q2} = 1 + \frac{1}{1 - \rho_1}(\rho_1 t_{s1} + \frac{\rho t_s}{1 - \rho})$

例如，系统中具有两个优先权的进程到达的数量相同，较低优先权的平均服务时间是较高优先权的 5 倍。我们希望有利于短进程，图 6.10 给出了整体结果。通过给短进程高优先权，来改善平均标准化轮转时间。采用抢占式算法可以获得这种效果，且总性能未受很大影响。

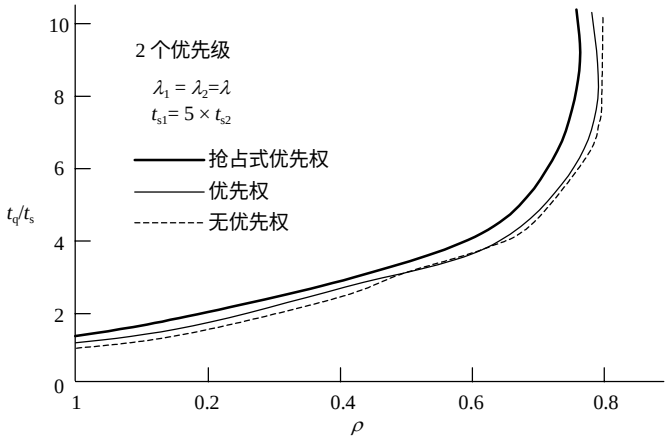


图 6.10　整体标准化响应时间

但是，当我们分开考虑两个优先权时却出现了很大的差异。图 6.11 所示为高优

先权短进程的情况。图中，上边的曲线是没用优先权的情况，我们只是借此简单分析运行时间较短的一些进程的相对性能；其他两条曲线则是优先权较高的两个进程的性能。当系统采用非抢占式优先权调度策略时，对进程性能的改善比抢占时更明显。

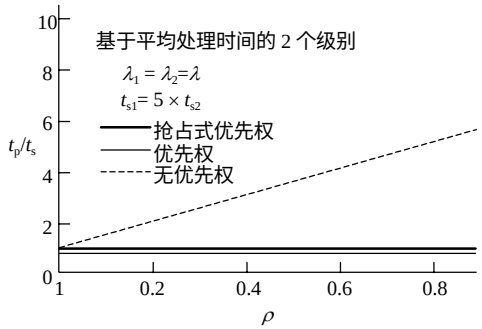


图 6.11　高优先权短进程的标准化响应时间

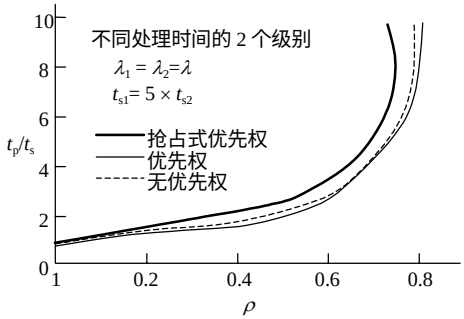


图 6.12　低优先权长进程的标准化响应时间

图 6.12 表明了对低优先权长进程作类似分析的结果。正如所料，这些进程在基于优先权的调度下会出现性能递减的情况。

6.2.5　模拟模型

模拟模型是在特定假设下，运用于特定的进程集的，有一定局限性。尽管如此，还是可以从中获得一些有用的结论。

假设有 5000 个进程，到达率λ=0.8，平均服务时间 t_s=1，于是处理机利用率$\rho=\lambda t_s$=0.8。

将这 5000 个进程按其执行时间分为 10 个组，每个组 500 个进程。其中，最短的 500 个进程被分在第一个组，剩下的进程中最短的 500 个分到第二组中，依此类推。这样，就可以分析不同的调度策略对不同长度的进程的运行效果。

模拟实验发现 FCFS 策略性能很差，1/3 进程的标准化轮转时间是其服务时间的 10 倍多，并且它们是最短的进程。再看等待时间，因为 FCFS 策略的调度与进程所需的服务时间无关，所以像我们料想的那样，所有进程的等待时间都是一致的。以一个时间单位为时间片的 RR 策略，除了那些服务时间不到一个时间片的短进程外，所有进程的轮转时间都在 5 个时间片左右，这对所有进程都比较公平，SPN 策略性能比 RR 策略好。作为 SPN 策略优化的 SRT 策略性能比 SPN 策略优(除最长的 7%进程外)。FCFS 策略较适合长进程，SPN 策略适合短进程。HRRN 策略执行效率介于 FCFS 策略和 SPN 策略之间。多级反馈队列调度对短进程较好。

6.2.6　公平分享调度策略

前面所讨论的调度策略都是将所有的就绪进程作为一个整体，从中选取下一个要执行的进程。

但是，在多用户系统中，如果单个用户的应用程序或作业可分解成多个进程和线程，那么，此时就需要一个不同于传统调度策略所用的收集进程的方法。用户关心的并不是某一具体进程如何执行，而是构成一个应用的所有进程是如何执行的，于是基于这些进程组的调度也很重要。这种策略一般称为公平分享调度(Fair-Share Scheduling，FSS)策略，并且可推广到用户组的情况。例如，在分时系统中，希望同一部门的用户属于同一组，调度策略应尽量给每一组相同的服务。这样，如果一个部门有许多用户登录该系统，那么，可看到该部门用户的响应时间明显低于其他部门用户的响应时间。

“公平分享”隐含了调度的基本原理。每个用户被赋予某种权，它规定了该用户所占全部资源的份额。并且所有用户共享一个处理机。这种策略采用线性工作方式。因此，如果用户 A 的权是 B 的两倍，那么，在长时间运行中，用户 A 能做 B 所做的 2 倍的工作。公平分享调度(FSS)的目的是监视用户使用资源的情况，使超过其份额的用户获得较少的资源，而少于其份额的用户获得更多的资源。

下面介绍一个在 UNIX 系统中实现的策略。FSS 考虑相关进程组的执行历史，根据执行历史来决定调度，系统将用户分成一系列公平分享的用户组并为每组分配部分处理机资源，如有 4 个用户组，则每组占用 25%的处理机时间，在效果上看，每组用户工作在一个虚拟系统中，该系统的运行效率成比例地低于整个系统的运行效率。

基于优先权的调度，要考虑每个进程的优先级别、最近使用 CPU 的情况和该进程所在组最近使用 CPU 的情况。优先权值越大级别越低，对组 k 中的进程 j，有公式如下：

$$P_j(i)=\mathrm{Base}_j+\frac{\mathrm{CPU}_j(i-1)}{2}+\frac{\mathrm{GCPU}_k(i-1)}{4\times W_k}$$

$$\mathrm{CPU}_j(i)=\frac{U_j(i-1)}{2}+\frac{\mathrm{CPU}_j(i-1)}{2}$$

$$\mathrm{GCPU}_k(i)=\frac{\mathrm{GU}_k(i-1)}{2}+\frac{\mathrm{GCPU}_k(i-1)}{2}$$

其中，$P_j(i)$=在时间段 i 开始处进程 j 的优先权；Base_j=进程 j 的基本优先权；$U_j(i)$=在时间段 i 进程 j 占用 CPU 的时间；$\mathrm{GU}_k(i)$=k 组中所有进程在时间段 i 中占用 CPU 的时间；$\mathrm{CPU}_j(i)$=时间段 i 中进程 j 的指数加权平均 CPU 占用时间；$\mathrm{GCPU}_k(i)$=时间段 i 中组 k 中所有进程的指数加权平均 CPU 占用时间；W_k=组 k 的权，$0\leqslant W_k\leqslant 1^{\Sigma W_k}=1$。

每个进程都有一个基本的优先权，它随进程使用 CPU 和进程所在组使用 CPU 的增加而下降。两种情况下，CPU 使用的平均运行时间都采用指数加权平均。在组使用处理机时，其平均值由该组的权决定。组的权越大，它对使用时间的影响就越小。

如图 6.13 所示，某一组中进程 A 和另一组中进程 B、C，每组权为 0.5。假设所有进程都只使用 CPU 并且已就绪，所有进程的基本优先权为 60。使用 CPU 的情况按下述衡量：处理机 1 秒钟中断 60 次；每次中断，当前正在执行的进程的 CPU 使用域与对应组的 CPU 使用域都增加，每秒结束时重新计算优先权。

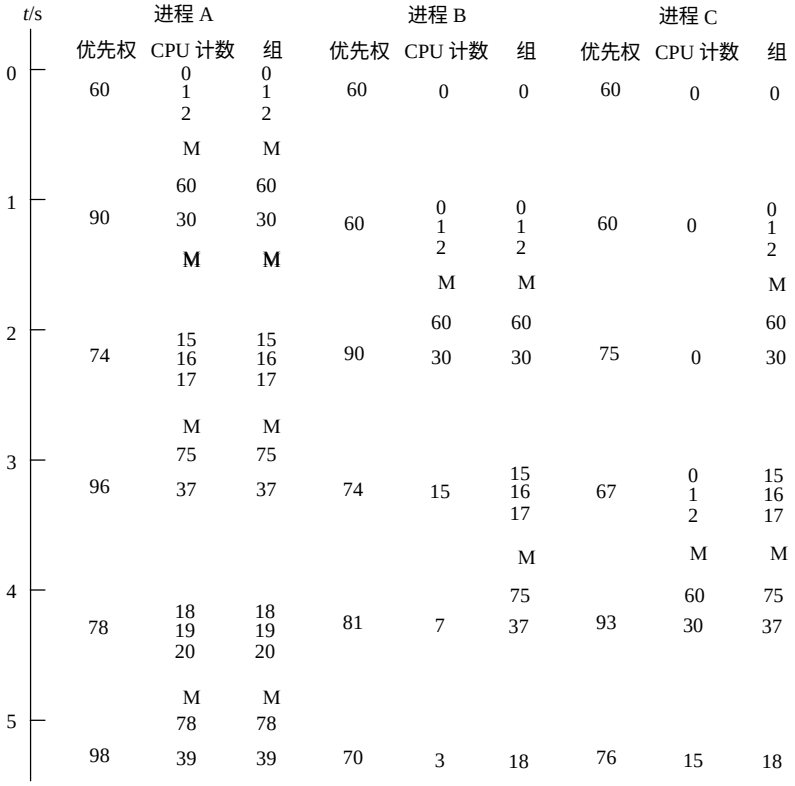


图 6.13　公平分享调度举例——3 个进程，2 个组

图 6.13 中，进程 A 首先调度，在第 1 秒结束时，它被抢占，进程 B 和 C 有更高的优先权，调度进程 B，第 2 秒结束时，进程 A 优先权最高，调度顺序为 A，B，A，C，A，B，如此下去。这样，每组进程各占 50%的 CPU 时间，A 占 50%，B 和 C 共占 50%。

6.3　多处理机调度

一个计算机系统有多个处理机时，其调度会有一些新特点。在此先简单介绍多处理机概念，然后分析进程级调度和线程级调度的显著区别。

多处理机可分为：

① 松耦合多处理机。由一群相对独立的系统构成，每个处理机都有自己的内存和 I/O 通道。

② 特定功能多处理机。如一个 I/O 处理机。在这种情况下，有一个控制者——全局处理机，其他的特定功能处理机受其控制并为它服务。

③ 紧耦合多处理机。由一系列共享主存的处理机组成，并由一个操作系统集中控制。

本节主要讨论最后一种多处理机的调度。多处理机一般是用来提高多道程序的性能与可靠性的。

① 性能：一个运行多道程序操作系统的多处理机应比单处理机性能好，并且应比多个单处理机系统花费少。

② 可靠性：在紧耦合处理机系统中，如果所有处理机都高性能地工作，那么一个处理机出错只会使整个系统工作性能降低，而不会造成系统完全崩溃。

最近几年，多进程和多线程的应用越来越广，把这些应用称为并行应用，其好处是使软件的设计变得简单了，并且可以在不同处理机上对应用的不同部分同时进行处理，从而提高性能，但这种对多处理机的应用使调度更复杂了。

6.3.1　粒度

描述多处理机的一个好方法是考虑系统中进程的同步粒度，或同步的频率。可以把粒度分成 5 个层次，如表 6.8 中所列。

表 6.8　同步粒度与进程

粒度大小	描　　述	同步间隔(指令条数)
细	单指令流的固有并行性	＜20
中	单个应用中的并行处理或多任务处理	20~200
粗	多道程序环境中并发进程的多进程处理	200~2000
超粗	通过网络节点的分布式处理以形成一个单一计算环境	2000~1M
独立	大量无关进程	>2000

1. 独立并行

在独立并行进程中，进程间没有明显的同步。每个进程都代表一个独立的应用或作业。分时系统就是独立并行的例子。每个用户都运行一个特定的应用，多处理机系统提供与单处理机多道程序系统一样的服务。由于有不只一个处理机可用，系统的平均响应时间减少了。这样就有可能使用户获得类似使用工作站的性能，但他使用的仅仅是一台 PC 机，如果能共享文件或消息，那么，这些分散的系统必须互联起来，通过网络构成一个分布式系统。另一方面，单个的多处理机共享系统一般要比分布式系统开销小，因为它能节省磁盘空间和其他外设的使用。

2. 粗粒度和超粗粒度并行

如果使用粗粒度和超粗粒度并行，那么进程间必须有同步，但这种同步是在一个非常粗的级别上进行的，这种情况可看作多道程序单处理机上运行并发进程集，可用多处理机来支持并且不需改变用户软件。

一般，需要通信或同步的任何并发进程集都可从多处理机结构中获益，如果进程间交互频繁，则分布式系统可提供较好的支持，但若交互行为再频繁点，那么网中的通信耗费会迅速增加，这种情况下，多处理机系统可提供更有效的支持。

3. 中粒度并行

单个应用可以在一个进程中作为多线程的形式有效地实现，这时，一个应用的潜在并行性由程序员明确指出，这通常需要在这种应用的多线程中存在相当程度的协调和交互，从而导致中粒度的同步。

尽管独立、超粗和粗粒度并行可由多道程序单处理机或者由无特定调度要求的多处理机支持，但我们仍要重新讨论线程的调度，因为应用程序中的大量线程交互频繁，对一个线程的调度将影响整个应用程序，本节后面部分再讨论这个问题。

4. 细粒度并行

细粒度并行比线程并行要复杂得多。尽管在高并行应用中已进行了大量的研究工作，但细粒度并行仍是一个专业化的领域，对其研究较少。

6.3.2　设计要点

多处理机调度中有 3 个相关联的要点：

① 分配进程给处理机；

② 在单个处理机上运行多道程序；

③ 进程的实际分派。

1. 分配进程给处理机

(1) 静态分配和动态分配

假设多处理机的结构是相同的，也就是说在获得主存和 I/O 设备上没有哪个处理机有特定的物理优先权，那么，最简单的分配方法是将所有的处理机看做一个共享资源，按需分配进程，但问题在于分配是静态的还是动态的。

如果一个进程在被激活直到完成运行，一直都被分配到一台处理机上，那么，每台处理机都有一个专用的队列，这种分配方法的好处在于调度耗费小，因为一旦一个处理机被分配，那么它永远被分配，另外，专用处理机允许使用一种叫做组或群调度的策略，这在后面讨论。

静态分配的一个不利之处在于有可能某个处理机长期空闲，其等待队列空，而另一个处理机却有一个很长的等待队列，为避免出现这种情况，可使用公共等待队列，所有进程都进入这个全局队列，并且可以被任一可用的处理机调度。因此，一个进程在其整个生命期内，在不同时间可以被不同的处理机执行，在一个紧耦合共享存储器系统中，所有进程的现场信息都可以被所有处理机使用，因此，调度某个进程的耗费应独立于调度它的处理机。

(2) 主/从结构和对称结构

还有两种分配进程的方法：主/从结构和对称结构。在主/从结构中，操作系统的主要内核功能总是运行在一个特定的(主)处理机上，其他从处理机只执行用户程序，主处理机负责调度作业，一旦一个进程被激活，如果从处理机需要服务(如 I/O 请求)，则它必须向主处理机发送一个请求，并等待其允许。这种方法相当简单，并且很容易解决冲突问题，因为只有一个处理机能对所有的内存与 I/O 资源进行控制。这种方法的缺点是

① 主处理机发生故障会使整个系统崩溃；

② 主处理机会成为性能瓶颈。

在对称结构中，操作系统可以在任何处理机上运行，并且每个处理机都从可用进程池中选择进程，进行自身的调度，这种方法使操作系统变得复杂。操作系统必须保证两个处理机不会同时选择同一个进程，而且进程不会从队列中丢失，必须有相应的解决资源竞争的同步问题的方法。

这两种结构各有好处，其中后者可提供一系列而不是一个处理机执行内核进程。而前者提供一种简单的方法来管理不同的内核进程和其他进程。

2. 在单处理机上使用多道程序设计

每个进程被静态地分配给一个处理机，并且在其生命期内不变，这会产生一个新问题：该处理机是否具有多道程序设计的功能？当一个进程经常由于等待 I/O 而被阻塞，或由于并发/同步的要求将一个进程静态分给一个处理机，都有可能造成很大的浪费。

在一个处理粗粒度或独立同步粒度的传统多处理机中，要想获得更高的性能，则每个单独的处理机都应该能在一些进程中选择。但在多处理机系统中处理中粒度应用时，情况就不那么明显了。如果处理机很多，那么所有处理机都很忙是不太可能的。我们希望对应用提供最好的平均性能，因为由大量线程构成的应用的运行效率很低，除非它的所有线程都可同时运行。

3. 进程的实际分派

关于多处理机调度的最后一个设计要点是选择哪一个进程运行，从多道程序单处理机系统中可看出，使用优先权或执行历史的调度策略比简单的 FCFS 策略的性能要好得多。在多处理机中，这些调度策略并不复杂。在传统的进程调度中，一个较简单的方法可以有较高的效率，但在线程调度中出现了一些新特点，这些特点比优先权或执行历史更重要，下面将进行讨论。

6.3.3　进程调度策略

在最传统的多处理机系统中，进程并不依赖于处理机，所有处理机共用一个队列，或者由于使用优先权，存在优先权不同的多重队列，所有进程都进入一个可供所有处理机调用的公共进程池，无论哪种情况，都可将其看做一个多服务器排队模型。

考虑一个双处理机系统，其中每个处理机以一个单处理机系统调度进程的一半概率进行调度，比较 FCFS、RR 和最 SRT。在 RR 中，假设时间片比现场切换时间要长，比实际服务时间要短，则结果在很大程度上依赖于服务时间，考察变化系数 C_s：

$$C_s = \frac{\delta_s}{t_s}$$

其中，δ_s 为服务时间的标准方差；t_s 为平均服务时间。

C_s 为 1 意味着指数级服务时间，实际进程的服务时间变化幅度要比这大一些。但 C_s 大于 10 的情况并不常见。

图 6.14(a)比较了 RR 和 FCFS 的吞吐量。可以看出在双处理机系统上不同的调度策略的差别很小。在双处理机系统中，某个进程服务时间很长并不会产生很大的影响，其他进程可用另一个处理机，图 6.14(b)也给出了类似的结果。

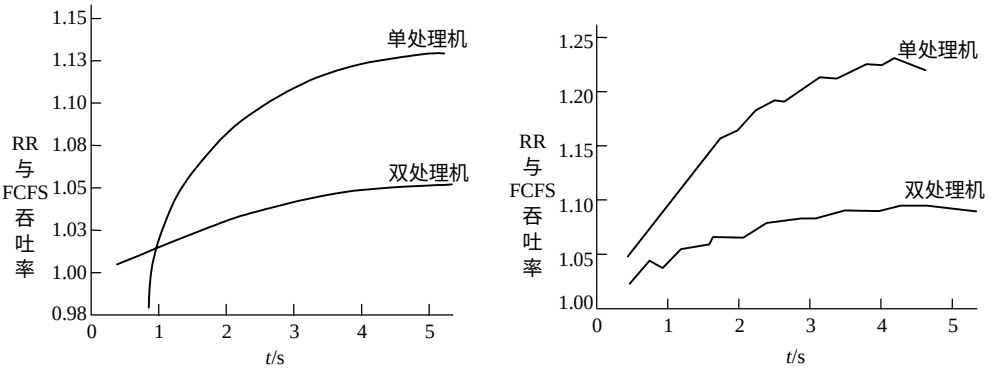


图 6.14　单处理机与双处理机调度性能比较

(a) 第一种情况　(b) 第二种情况

在两个处理机系统中，使用特定的调度策略并不像在单处理机系统中那么重要，如果处理机数目更多，这就更明显。因此，在一个多处理机系统中使用一个简单的 FCFS 策略或 FCFS 策略和静态优先权结合的策略就可以了。

1. 线程调度

由于使用线程，执行的概念就与进程分开了，一个应用程序可用一系列线程的形式完成，它们之间相互配合，并在相同的地址空间同时运行。

在单处理机系统中，可以使用线程来构造程序并扩大 I/O 的利用率，由于选择线程的难度与选择进程一样，因此，不需花费太大的代价。在多处理机系统中，可发现线程的真正好处：在一个应用中能实现真正的多线程并行，如果不同的线程在不同的处理机上同时运行，就有可能获得最佳性能，但对于线程间要相互通信的应用而言，在线程管理和调度上的小小差异都会对性能产生显著影响。

在多处理机上调度线程和分配处理机有以下 4 种方法较突出：

① 负载共享。并不是将进程分配给某一特定的处理机。系统有一个全局就绪队列，处理机空闲时就从该队列中选择一个线程。

② 群调度。相关联的线程集被一个处理机集同时一对一调用。

③ 专用处理机分配。在程序执行期间，每个程序都被分配与其线程数相等的处理机，程序结束时，将所有的处理机归还，以便其他程序使用。

④ 动态调度。程序的线程数可随程序的执行而改变。

2. 负载共享

负载共享是最简单的方法，它直接继承了单处理机系统的许多好处，其优点如下：

① 对处理机平均分配负载，保证在有作业没完成时不会有空闲的处理机。

② 不需要集中调度者，一旦有处理机空闲，操作系统就在该处理机上运行调度程序以选出下一个线程。

③ 使用全局队列，按优先权、执行历史或预测处理命令进行调度。

对 3 种负载共享方法的分析如下：

① 先来先服务(FCFS)。当作业调入后，它的所有线程都依次放到共享队列的尾部，处理机空闲时，选出下一个就绪线程执行直到其完成或阻塞。

② 最小号线程优先。共享队列按优先级排列，最高优先权的作业号最小，具有相等优先权的作业按 FCFS 排列，和 FCFS 一样，被调度的进程运行到完成或阻塞为止。

③ 估计最小号线程优先。给未完成且线程数最少的作业最高优先权，到达作业的线程数比当前运行作业线程数要少，那么它就抢占已调度的作业。

通过模拟可得：FCFS 策略比另两种策略要好，进一步分析还可发现群调度策略的某些形式在总体上要比负载共享要好。

负载共享的缺点如下：

① 集中队列占据了部分内存，对它的访问要求互斥。因此，在许多处理机同时都要工作时，就会成为瓶颈。如果处理机很少，那么这个问题并不明显，但如果有几十个甚至几百个处理机，那么这个潜在的瓶颈问题就很明显了。

② 被抢占进程不可能再在同一台处理机上运行，如果每个处理机都有局部高速缓存，其效率都不会很高。

③ 如果所有进程都被放入一个公共线程池，那么同一程序的所有线程不可能同时获得处理机。如果一个程序的各线程间存在程度相当高的协调，那么进程切换也会影响性能。

尽管自身调度有许多不足，但仍是目前多处理机系统中使用最普遍的方法之一。

一个改进负载共享的方法已被用于 Mach 操作系统中。该系统中每个处理机都拥有一个本地队列，同时也存在一个全局队列。本地队列中的线程都等待特定处理机。有一个处理机检查本地队列并给那些特定线程绝对优先权。

3. 群调度

一系列进程同时被一系列处理机调度，称为群调度，它有如下好处：

① 如果紧耦合的进程并发执行，那么同步阻塞减少了，进程切换也减少了，从而提高了效率。

② 减少了调度耗费。因为一个决定会同时影响许多处理机和进程。

在C_m^*多处理机系统中，使用了并发调度。并发调度是建立在调度一系列相关联任务的基础上的。

群调度用于组成一个进程的多个线程的同时调度。群调度对于中粒度和细粒度的并行应用是必要的。因为当这些应用中的一部分线程不运行而其他的线程要运行时，性能就会显著下降。此外，它对任何并行应用都有好处。

群调度的使用产生了对处理机分配的需求。假设有 N 个处理机和 M 个应用，其中每个应用有小于 N 个的线程，那么每个应用都应分配 N 个处理机时间的 $1/M$。这种策略效率很低，如果有两个应用，一个有 4 个线程，另一个只有 1 个线程。若按应用统一分配时间则浪费 37.5%的处理机资源，因为当单线程的进程运行时，有 3 个处理机空闲了(见图 6.15(a))，如果有几个单线程应用，那么它们同时运行将提高处理机的利用率。若按线程权分配调度，则有 4 个线程的应用被赋予 4/5 的时间，而单线程应用占 1/5 的时间，于是将处理机的时间浪费减少到 15%(见图 6.15(b))。

4. 专用处理机分配

群调度的一个极端形式是将一群处理机分配给一个应用，使它们在该应用的生

命期内归该应用专用。也就是说，当一个应用被调度后，它的所有线程都分配给同一个处理机直到该应用运行完毕。

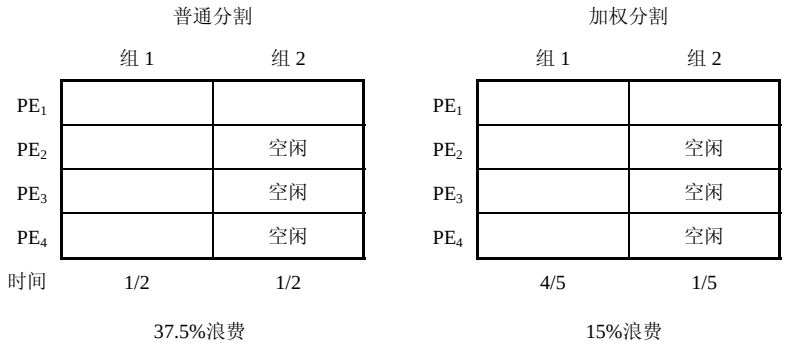


图 6.15　4 个线程和 1 个线程的群调度举例

这种方法看上去非常浪费处理机时间。如果一个应用的某线程由于等待 I/O 被阻塞，同时又要与另一线程同步，那么该线程所在的处理机将空闲。但也有些结果能支持这种方法：

① 在高并行的多处理机系统中，有几十个甚至上百个处理机，每个都代表整个系统的部分花费，处理机利用不再是效率的惟一标准。

② 在程序的生命期中完全避免进程切换会导致该程序的加速变化。

例如，在一个有 16 个处理机的系统上运行两个应用，一个是矩阵乘法，另一个是快速傅立叶变换。每个应用都将其问题分成大量的任务，这些任务被映射成该应用的线程，编程时允许所用的线程数可变。实际上，这些任务由应用来定义并排队。从队列中取出任务并映射成可供使用的线程。如果一个应用的任务数大于线程数，那么多出来的任务将保留在队列中直到某些线程完成后再选出。

同时运行 24 个线程的应用，执行矩阵运算的速度比单线程应用快 1.8 倍，执行傅立叶转换则快 1.4 倍。数据表明，当应用的线程数大于 8 时其效率显著下降。在进程数大于处理机个数时也存在同样的情况，进一步说，线程数越多性能越糟，因为有线程以相当高的频率抢占与再调度，这种过多的抢占将导致许多资源的利用率不高，包括等待、挂起、进程退出临界段的时间，进程切换的时间，以及高速缓存的低效。

专用处理机分配和群调度在调度时都涉及到处理机分配。多处理机系统上的处理机分配类似于单处理机系统上的内存分配。在某个时候给一个程序分配多少个处理机非常类似于某时刻给进程分配多少内存页的问题。活动工作集类似于虚拟存储工作集。活动工作集是指为了使应用执行速度可接受，在处理机上必须同时调度的最小活动线程数。和内存管理策略一样，对活动工作集中所有元素的调度失败将导致处理机抖动，这在一些需要服务的线程调度引起其他线程调度失败时会发生。类

似地，处理机碎片是指有一些处理机没有工作，但又无法满足任何等待的应用程序，群调度和专用处理机分配可以避免这些问题。

5. 动态调度

有些应用允许线程数动态改变，从而操作系统可以调整负载来提高效率。

有一种方法可以使操作系统和应用都能进行调度。操作系统负责作业的处理机分配，每个作业使用处理机，将其可运行的任务集映射成线程，由应用决定运行哪个子集，如同进程抢占时挂起某个线程一样，这种方法并不适合所有的应用。

这种方法中，操作系统的调度仅限于处理机分配，并按下述策略进行。当一个作业需要一个或更多的处理机时(无论该作业是第一次到达还是其需求发生了变化)，如果有空闲处理机，则满足请求。否则，

① 如果该作业是刚到达的，则将目前拥有多个处理机的作业的一个处理机分给它；

② 如果都无法满足，则它一直等到有可用处理机或撤消请求(例如，它不再需要另外的处理机)为止。

在释放一个或更多处理机时(包括作业完成离开)，扫描没满足请求的队列，给当前没有处理机的作业分配处理机，再次扫描队列，将剩余处理机按 FCFS 方式分配。

对适合于动态调度的任务来说，这种方法比群调度和专用处理机分配要好。

6.4　实时调度

最近几年，实时计算已成为一项重要的计算机科学和工程设计的方法。操作系统，特别是调度，可能是实时操作系统最重要的组件，实时操作系统的应用例子有实验控制、植物生长控制、机器人控制、空中交通控制、远程通信，以及军事命令和控制系统。下一代系统将包括自动挖土机、控制机器人的活动关节、人工智能、空间站以及海底探索等。

实时计算不仅要求答案正确而且要求在一指定时间内完成。可以通过定义一个实时进程或任务来定义一个实时操作系统。总体上说，实时操作系统中有一些任务是实时的，需要紧急处理，这些任务试图控制外部设备或对外部事件作出反应。因此，一个实时任务必须与其相关的事件保持同步。于是对一特定任务经常给出一期限，该期限指出其开始时间或完成时间，这种任务可分为强、弱两种类型，一个强实时任务必须在规定时间内完成，否则会造成无法预料的损失或系统的致命错误。一个弱实时任务有一个相关期限，但并不是必须的，即使超过期限，也会继续调度直到完成。

实时任务有期段型和非期段型之分，一个非期段型任务有一个它必须开始或完成的期限，或者对两者都限制，而对于期段型任务，其要求可表述为“在期段 T 内一次处理一个事件”。

6.4.1　实时操作系统的特性

实时操作系统在 5 个一般领域有独特的要求：决定性、响应性、用户控制、可靠性和弱失效操作。

1. 决定性与响应性

一个操作系统在某种程度上要决定其执行操作的时间，若有多个进程竞争资源和处理机时间，则没有系统能够完全决定。在实时操作系统中，一个进程如果要求服务，则必须通过外部事件和时序来请求，一个操作系统满足请求的程度取决于：中断响应的速度，以及系统在规定时间内解决所有请求的能力。

衡量系统决定性能力的有效方法是，衡量从一个高优先权设备中断到服务开始的最大延迟，在非实时操作系统中，这个延迟可能是几十到几百毫秒，而实时操作系统只能是 1μs 或 1ms。

一个与之有关但又不同的特征是：响应性。决定性是指系统得知中断前的延迟时间，而响应性是指系统在得知中断后多长时间内对这个中断进行服务。响应包括：

① 初始处理中断和开始常规中断服务(ISR)的时间。如果运行需要现场切换，那么延迟将比当前进程切换的 ISR 要长。

② 执行 ISR 的时间，这依赖于硬件平台。

③ 中断嵌套的影响，如果一个 ISR 能被另一个中断所中断，那么该服务被延迟。

决定性和响应性一起决定了对外部事件的响应。对响应时间的要求在实时操作系统中是非常重要的，因为这些系统必须满足个人、设备及系统外部数据流的时序要求。

2. 用户控制

一般来说，用户控制在实时操作系统中要比其他操作系统应用得要广。在一个典型的非实时操作系统中，用户或者根本无法控制操作系统的调度功能，或者只能提供较宽的指导，如将用户分成多个优先级别的组。但在实时操作系统中，有必要允许用户详细控制任务优先级。用户应能区别强、弱任务，指出每类的相对优先级别。实时操作系统也应允许用户指定分页或进程交换的特性，哪些进程应一直驻留内存，使用哪种磁盘传输算法，如何确定不同优先级别的进程等等。

3. 可靠性

可靠性对实时操作系统比对非实时操作系统要重要得多，非实时操作系统的暂

时失效可通过重新启动系统来解决，多处理机非实时操作系统中一个处理机失效会引起服务降级，直到失效的处理机被更换或修好，但实时操作系统必须实时响应和控制事件，性能下降也许会导致可怕的后果。

4. 弱失效操作

在其他方面，实时操作系统与非实时操作系统的区别是它们的度，即便是实时操作系统也必须设计对不同失效模式的响应，弱失效操作就是指系统在失效时尽量保护其数据和能力。例如，在一个典型的 UNIX 系统中，当它发现内核中出现数据错误时，它就发出一条出错信息并将内存中的内容导入磁盘，以便以后进行错误分析，接着结束系统的运行。与此相反，实时操作系统是在运行时修改错误或降低其影响。典型的情况是，它将通知用户或用户进程：它要纠错并且降级运行，在关闭事件时，保证文件和数据的一致性。

弱失效操作的一个重要方面就是稳定性。若一个实时操作系统是稳定的，那么如果它不可能满足所有任务的期限时间要求，就尽可能满足最重要、优先权最高的任务的期限时间要求。

为满足上述要求，实时操作系统应包含如下特性：

① 快速现场切换；

② 尺寸小；

③ 能迅速响应外部中断；

④ 多任务并存，并有如信号量、信号、事件等进程间通信工具；

⑤ 使用专门的线性文件来收集数据；

⑥ 基于优先权的抢占调度；

⑦ 最小化禁止中断的时间间隔；

⑧ 简单地延迟任务一段时间或停止/重新开始任务；

⑨ 特殊的警告和超时。

实时操作系统的核心是短程任务调度。在设计调度时，公平性和最小化平均响应时间并不重要，重要的是使所有强实时性任务都在规定时间内完成，同时应尽量在规定时间内完成更多的弱实时性任务。

大多数实时操作系统不能直接处理期限，它们一般设计成尽量响应实时任务。这样在期限到来时，迅速调度一个任务，从而使实时应用要求在各种条件下迅速响应，响应时间为毫秒级或纳秒级，在一些前沿应用中要求更严格。

图 6.16 给出了一种可能，在一个使用简单的时间片轮转的抢占式调度算法中，一个实时任务必须加入到就绪队列中等待下一个时间片，如图 6.16(a)所示的，调度时间对实时任务来说是无法容忍的。在非抢占性调度中，可以用优先权调度机制，给每个实时任务较高的优先权。就绪的实时任务在当前进程阻塞或运行完成时马上被调度(如图 6.16(b))。如果一个优先权进程处于关键段中，则会引起几秒钟

的延迟，这种方法也无法接受，一个较好的方法是将优先权和时钟中断结合起来。在固定的时间间隔内进行抢占，抢占时，如果有一个更高优先权的任务在等待，那么当前运行任务被抢占。这个任务甚至可以位于内核，这样延迟可以减至几毫秒(见图 6.16(c))。这种方法仅适合于一些实时应用，并不适合于那些更迫切的应用，在这种情况下，通常使用一种称为立即抢占的方法，立即抢占中，操作系统对中断立即响应，除非系统死锁。这样，实时任务的调度延迟将减至 100μs 或更少。

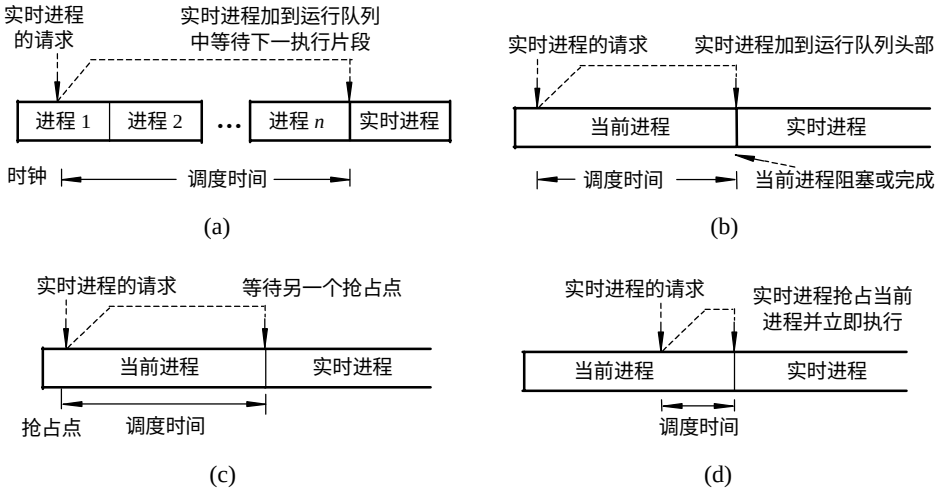


图 6.16　实时进程的调度

(a) 时间片段轮转抢占调度　　(b) 优先权驱动的非抢占调度

(c) 优先权驱动，在抢占点抢占调度　(d) 立即抢占调度

6.4.2　实时调度

实时调度是计算机科学中最活跃的领域之一。在本小节，我们讨论各种不同的实时调度并介绍两类流行的调度算法。

不同的调度算法取决于一个系统是否有调度分析能力；如果有，那么是静态的还是动态的；是根据分析结果产生调度，还是根据运行时分派哪些任务。基于这些考虑，算法可分为：

① 静态表驱动方法。对可行的调度进行静态分析，分析结果就是一个调度，它决定一个任务何时必须执行。

② 静态优先权驱动抢占方法。同样也是进行静态分析，但不产生调度，分析结果用来分配任务优先权。

③ 动态计划方法。在运行时(动态地)决定可行性而不是在执行开始时(静态地)决定优先权，一个到达任务在其期限能满足时才被接受运行，可行性分析的结果之

一就是一个调度或者是用来决定何时分派该任务的方案。

④ 动态尽力方法。不进行可行性分析，系统尽力满足所有期限并放弃所有已超过期限的进程。

静态表驱动调度适合于阶段性的任务，分析的输入包括阶段到达时间，执行时间，阶段结束期限，以及每个任务的相对优先权。要建立一个能满足所有阶段性任务要求的调度。这是一种可预测但不灵活的方法，因为任何任务要求的变化都会导致重建调度。

静态优先权驱动抢占方法利用了在许多非实时多道程序系统中所用的优先权驱动抢占调度机制。在一个非实时操作系统中，有许多因素影响优先权。例如，一个分时系统的进程优先权的变化取决于它是偏重处理机使用还是偏重 I/O 使用。在实时操作系统中，优先权的分配取决于各任务的期限。例如，单一比率算法是根据阶段来分配优先权的。

在动态计划调度中，到达的任务在未执行前，先检查新到达的任务和以前执行过的任务，如果新到达任务能满足期限，已调度的任务不会超过期限，则重新产生一个包括这个新任务的调度。

动态尽力方法是目前许多商业实时操作系统中都采用的一种方法。当任务到达时，系统根据其特性为其分配一个优先权，再使用期限调度的某种形式。如果任务是阶段性的，就不可能进行静态调度分析。在这种调度中，直到期限到达或某任务完成，才可能知道时间限制是否被满足，这也是这种调度的弊端所在，其好处在于容易实现。

6.4.3　期限调度

目前，大多数实时操作系统都有一个设计目标：尽快开始实时任务。因此，它们强调快速中断处理和任务发送。但实际上，这不是一个衡量实时操作系统的特别有用的手段。实时应用总体上不关心最快速度，只关心能否在规定时间内完成(或开始)任务。无论是动态资源竞争与请求、进程的超载，还是硬件或软件故障都不应过早或过晚产生结果。

近几年，有一些更好的实时调度方法，但都要了解每个任务的额外信息。一般，要了解每个任务的如下信息：

① 就绪时间。任务变成就绪的时间。对于一个阶段性的任务，这是可以预先知道的时间序列，对非阶段性任务，该时间可能预先知道，或者仅当任务就绪时操作系统才知道。

② 开始期限。一个任务必须开始的时间。

③ 完成期限。一个任务必须完成的时间。一个典型的实时应用必须有开始期限或者完成期限，但不一定全有。

④ 处理时间。从执行任务开始到任务完成所需的时间。有些情况下可以提供这个时间；有些情况下操作系统衡量的是一个指数平均值；在某些情况下，甚至不使用这个时间。

⑤ 资源需求。任务执行时所需要的一系列资源(不包括处理机)。

⑥ 优先权。这是衡量任务相对重要性的手段。强实时任务可以有一个完全优先权，超过其期限系统会失败。系统继续运行时，强实时任务和弱实时任务都可指定相应的优先权，作为调度的依据。

⑦ 子任务结构。一个任务可分解成一个主子任务和一个可选子任务，只有主子任务才有强期限。

实时调度中考虑期限时，以下几个方面要考虑：接下来调度哪个任务，抢占哪个任务。可以看到，如果使用开始期限或完成期限，则可使超过期限的任务数最少，这个结论适用于单处理机系统和多处理机系统。

另一个关键的设计问题是抢占。设定了开始期限，非抢占调度就比较好。这时，允许实时任务的响应性能在完成主要的或关键的执行部分后阻塞自己，允许满足其他的实时开始期限，如图 6.16(b)所示。对带有完成期限的系统，用抢占策略较好(见图 6.16(c)、(d))。假如任务 X 正在运行而任务 Y 已就绪，则有可能同时满足 X 和 Y 的完成期限的惟一方法是抢占 X，执行 Y 至完成，再继续执行 X 至完成。

对于带有完成期限的阶段性任务调度，例如，系统收集并处理来自 A 和 B 的数据，收集 A 的数据的期限是每 20ms 一次，收集 B 数据的期限是每 50ms 一次。花费 10ms 处理 A 的数据，25ms 处理 B 的数据，总结如表 6.9 所示。

表 6.9　带有完成期限的阶段性任务调度举例

进　程	到达时间/ms	执行时间/ms	完成期限/ms
A_1	0	10	20
A_2	20	10	40
A_3	40	10	60
A_4	60	10	80
A_5	80	10	100
…			
B_1	0	25	50
B_2	50	25	100
…			

计算机每 10ms 进行一次调度。假设在此环境下，用优先权调度。图 6.17 前两个时序图给出了结果。如果 A 优先权较高，那么任务 B 的第一个例示在其期限到达时，只有 20ms 的处理时间，于是失败了。如果 B 优先权较高，那么 A 将超过其第一个期限，最后的时序图是最早期限优先权调度。在 t=0 时，A_1 和 B_1 同时到达。由

于 A_1 期限较早，执行 A_1，当 A_1 完成时，B_1 拥有处理机。t=20ms 时，A_2 到达，由于 A_2 期限比 B_1 早，中断 B_1，执行 A_2 到完成。在 t=30ms 时继续 B_1，t=40ms，A_3 到达。但 B_1 期限比 A_3 早，B_1 继续执行到 t=45ms 时完成，A_3 获得处理机执行到 t=55ms 时完成。

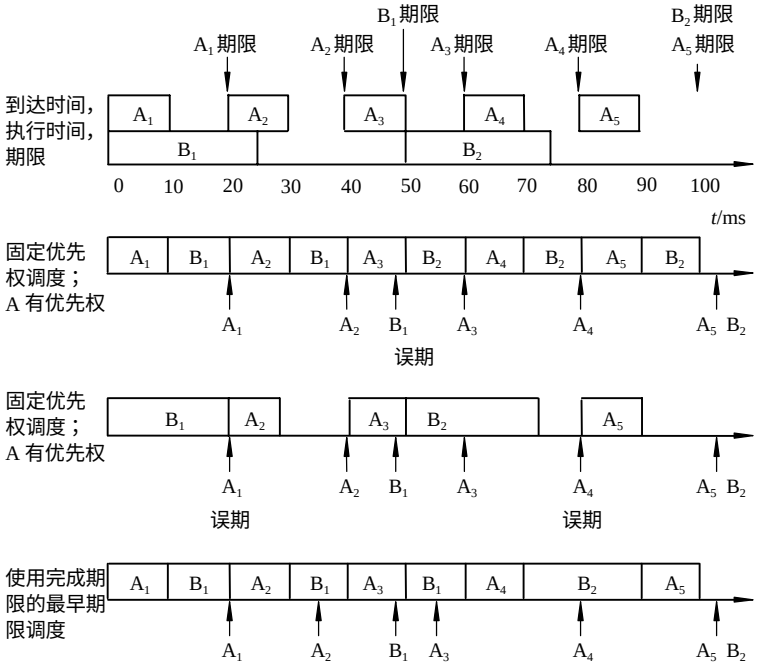


图 6.17　带完成期限的阶段性实时任务调度

例中，在抢占点给最早期限的任务优先权，所有系统要求都满足。由于任务是静态可预测的，使用的是静态表驱动方法。

下面讨论带开始期限的阶段性实时任务调度。图 6.18 的上半部分给出了一个有 5 个任务，每个任务的执行时间为 20ms 的例子的到达时间和开始期限。总结如表 6.10 所示。

表 6.10　带开始期限的阶段性实时任务调度举例

进　程	到达时间/ms	执行时间/ms	开始期限/ms
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

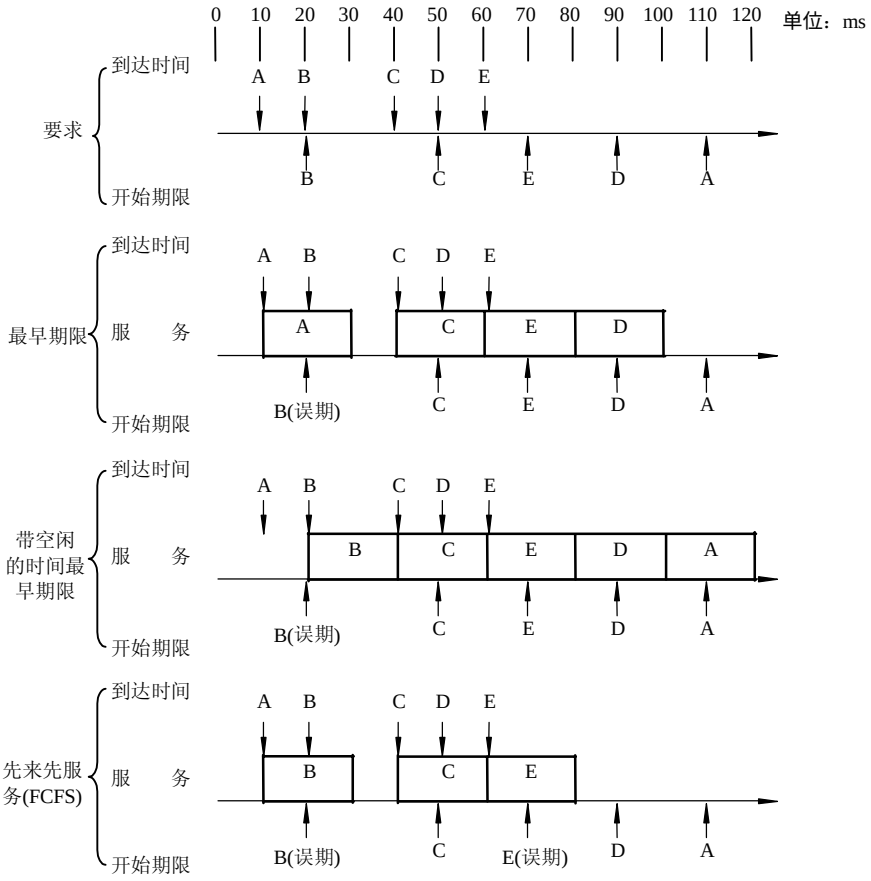


图 6.18　带开始期限的阶段性实时任务调度

最简单的方法是执行开始期限最早的就绪任务，上例中，尽管 B 要求立即服务但被拒绝，这在处理阶段性任务时是很危险的，尤其是带开始期限的。如果能在任务就绪前就知道其期限，那么就有可能提高其性能。于是，可以总是调度那些期限最早的进程，即使该进程不是就绪进程，这又有可能引起处理机空闲，甚至是在有就绪进程时。例中，系统不调度 A 即使它是惟一的就绪进程，结果是处理机利用率并不是最高，但所有期限都能满足。最后为比较，我们给出了 FCFS 调度结果，这时，任务 B 和 E 没满足期限(见图 6.18)。

6.4.4　比率单调调度

解决阶段性任务的多任务调度冲突问题的一个较好的方法是比率单调调度(RMS)。RMS 根据任务的阶段来分配优先权。

图 6.19 给出了阶段性任务的有关参数，任务阶段 T 是指任务的一个例示到达与下一个例示到达的时间差，任务的比率就是其阶段的倒数。例如，一个任务阶段为 50ms，则其比率为 20Hz。一般，任务阶段的结束时间也是该任务的强期限。执行(或计算)时间 t_C 指任务每次出现时所需的执行时间。在单处理机系统中，任务的执行时间不会超过任务阶段($t_C \leqslant T$)，如果一个任务一直执行到结束，那么它没有由于资源缺乏而被拒绝，那么该任务的处理机利用率$\rho=t_C/T$。例如，一个任务阶段为 80ms 而执行时间为 55ms，则其处理机利用率为 55/80=0.6875。

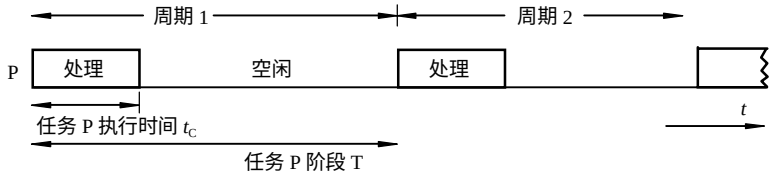


图 6.19　阶段性任务时序图

对于 RMS，最高优先权的任务就是阶段最短的任务，第二优先权的任务就是第二短阶段的任务，如此下去。若有不止一个任务可执行时，首先运行阶段最短的那个。如果将优先权作为其比率的函数，结果就是一个单调递增函数(见图 6.20)，因此，称其为比率单调调度。

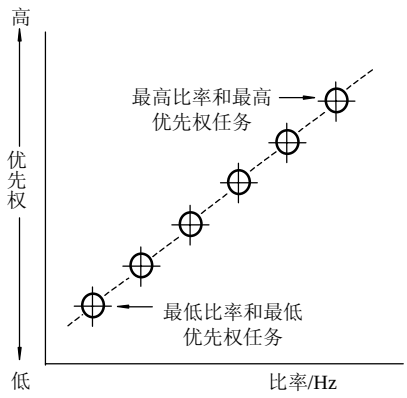


图 6.20　RMS 任务

衡量一个阶段性调度算法效率的方法是，看其满足所有任务的强期限的程度。假设有 n 个任务，每个都有一固定的阶段和执行时间，那么要满足所有的期限，必须满足下面的条件：

$$\frac{t_{C_1}}{T_1}+\frac{t_{C_2}}{T_2}+\Lambda+\frac{t_{C_n}}{T_n}\leqslant 1 \tag{6.1}$$

即所有任务的处理机利用率之和不超过 1。公式(6.1)对成功调度的任务数作了限制。对某些特定算法，限制可降低。对 RMS，有下面的不等式：

$$\frac{t_{C_1}}{T_1}+\frac{t_{C_2}}{T_2}+\Lambda+\frac{t_{C_n}}{T_n}\leqslant n(2^{1/n}-1) \tag{6.2}$$

表 6.11 给出上面所限制的值。随着任务数的增加，调度限制趋向于 $\ln 2\approx 0.693$。

例如，有 3 个阶段性的任务，其中，

• 对于任务 P_1，t_{C_1}=20，T_1=100，ρ_1=0.2。

• 对于任务 P_2，t_{C_2}=40，T_2=150，ρ_2=0.267。

• 对于任务 P_3，t_{C_3}=100，T_3=350，ρ_3=0.286。

这 3 个任务的总利用率为 0.2+0.267+0.286 =0.753，RMS 3 个任务的调度上限为

$$\frac{t_{C_1}}{T_1}+\frac{t_{C_2}}{T_2}+\frac{t_{C_3}}{T_3}\leqslant 3(2^{1/3}-1)=0.779$$

由 3 个任务的总利用率小于上限(0.753＜0.779)可知，如果用 RMS，所有任务都调度成功。

表 6.11　RMS 上限值

n	$n(2^{1/n}-1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
M	M
∞	ln2≈0.693

不等式(6.1)中的上限用于最早期限调度。因此，有可能获得更高的处理机利用率，从而最早期限调度可容纳更多的阶段性任务。RMS 已被广泛用于工业应用中，原因如下：

① 实际应用中性能差别不大，不等式(6.2)中上限较保守，实际中利用率可高达 90%。

② 大多数强实时操作系统中也有弱实时组件，如某些非关键显示及重建自身测试，这样可以将 RMS 调度强实时任务没用的处理机时间用来执行较低优先权的任务。

③ RMS 能保证稳定性，一个系统在由于超载或转换错误而不能满足所有期限时，必须确保关键任务的期限得到满足。在静态优先权分配中，只需对关键任务赋较高优先权，在 RMS 中可使关键任务阶段较短或修改关键任务的 RMS 优先权。最早期限调度中，阶段性任务的优先权随阶段而改变，这就使关键任务满足其期限很难实现。

6.5　系统举例

6.5.1　UNIX System V

UNIX 采用多级反馈调度，每个优先级队列内采用时间片轮转方法。系统采用 1 秒钟抢占，也就是，如果运行进程在 1 秒钟内未阻塞或完成，它就被抢占。优先权基于进程类型和执行历史。用到如下公式：

$$P_j(i)=\text{Base}_j+\frac{\text{CPU}_j(i-1)}{2}+\text{nice}_j$$

$$\text{CPU}_j(i)=\frac{U_j(i)}{2}+\frac{\text{CPU}_j(i-1)}{2}$$

其中，$P_j(i)$为进程 j 在阶段 i 开始时的优先权，其值越小优先权越高。Base_j 为进程 j 的基本优先权。$U_j(i)$为进程 j 在阶段 i 对 CPU 的使用。$\text{CPU}_j(i)$为进程 j 在阶段 i 的指数加权平均 CPU 利用率。nice_j 为用户控制的调节因子。

每秒钟重新计算每个进程的优先权，产生一个新调度。基本优先权是为了将所有进程分为固定的不同优先级别。CPU 和 nice 因子是用于防止一个进程从它所在的优先级中移出。按优先级降序排列如下：

① 交换；

② 阻塞 I/O 设备控制；

③ 文件控制；

④ I/O 设备控制特征；

⑤ 用户进程。

这种分层结构能有效使用 I/O 设备。在用户进程级，用执行历史来降低偏置 CPU 的进程，使偏重 I/O 进程的要求得到满足。这样可以提高效率。使用时间片抢占策略，也能满足分时要求。

如图 6.21 所示，进程 A、B、C 同时创建，并且有相同的基本优先权 60，系统时钟中断为 60 次/s，并且有一个运行进程计数器。假设没有进程阻塞，也没有进程就绪(与图 6.13 所示的比较)。

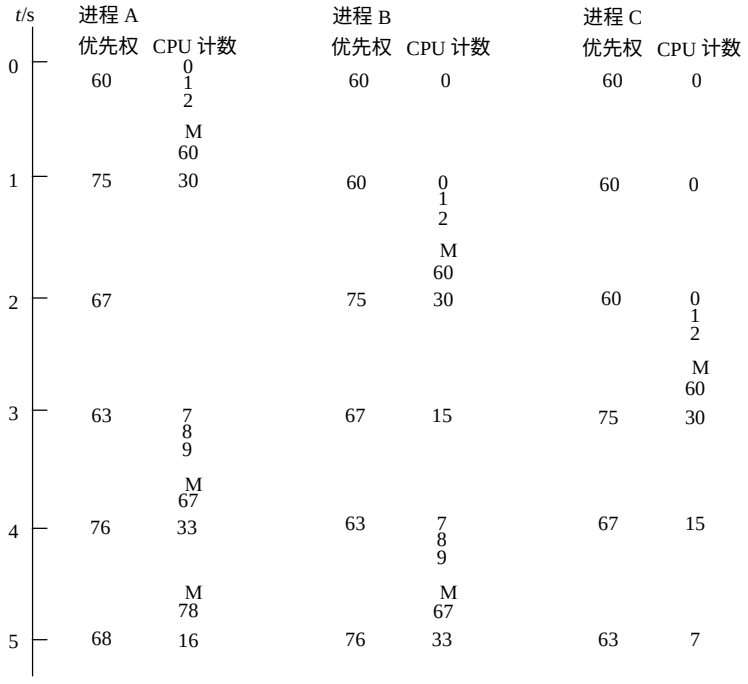


图 6.21　UNIX 进程调度举例

6.5.2　Windows NT

Windows NT(简称 NT)的调度目标与 UNIX 有所不同。UNIX 关心的是对用户组的公平响应服务，以共同分享系统。而 NT 尽量在高交互环境中响应单用户的需求或者作为一个服务器。和 UNIX 一样，NT 采用多优先权抢占策略。在 NT 中，优先权较灵活，每个优先权采用时间片轮转，有些优先权可以根据当前线程的活动动态地进行改变。

1. 进程和线程优先权

NT 中优先权分为两类：实时的和可变的。每类有 16 个级别，要求立即响应的线程处于实时类，包括通信和实时任务。

由于 NT 用优先权驱动抢占调度，实时优先权的线程在其他线程前执行。在单处理机系统中，当其优先权高于当前运行线程的某个线程就绪时，低优先权线程被抢占，处理机交给高优先权者。

这两类对优先权的处理有所不同(见图 6.22)。在实时优先权类中，所有线程优先权固定不变。活动进程按时间片轮转。在可变优先权类中，线程优先权被赋一初值，但可以改变。因此，在每个级别有一个 FIFO 队列，一个进程可移到可变优先权类的其他队列中，但 15 级的线程不能提到 16 级或实时类的级别中。

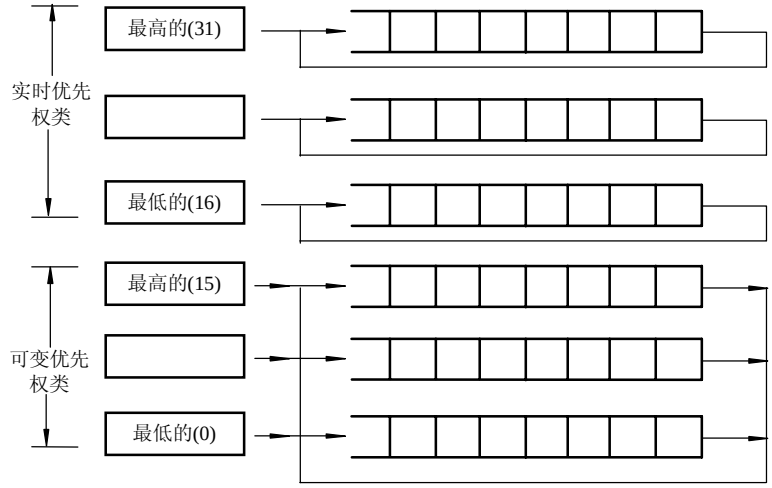


图 6.22　Windows NT 线程分派优先权

可变优先权类中的线程初始优先权由两个量决定：进程基本优先权和线程基本优先权。进程的一个属性是进程基本优先权，可以从 0 到 15。进程的每个线程有一个线程基本优先权，指出相对该进程的线程基本优先权。线程基本优先权可以等于其进程的基本优先权，也可在其进程的上下两个级别内。例如，进程基本优先权为

4，它的一个线程基本优先权为–1，那么该线程初始优先权为 3。

可变优先权类中有一个线程被激活，它的实际优先权(也就是线程的动态优先权)可在限制范围内变化。动态优先权不会降到比线程基本优先权低，也不会超过 15。如图 6.23 所示，进程有基本优先权 4，该进程的每个线程初始优先权只能是 2 到 6。每个线程的动态优先权可在 2 到 15 中变动。如果线程用完其时间片而中断，NT 就降低其优先权。如果线程中断，等待一 I/O 事件，NT 就提升其优先权。因此，偏重 CPU 的线程优先权较低，而偏重 I/O 的线程优先权较高，对偏重 I/O 的线程，交互式等待(如，等待键盘输入或显示)的进程比其他 I/O 进程级别高，故交互式进程在可变优先权类中有最高优先权。

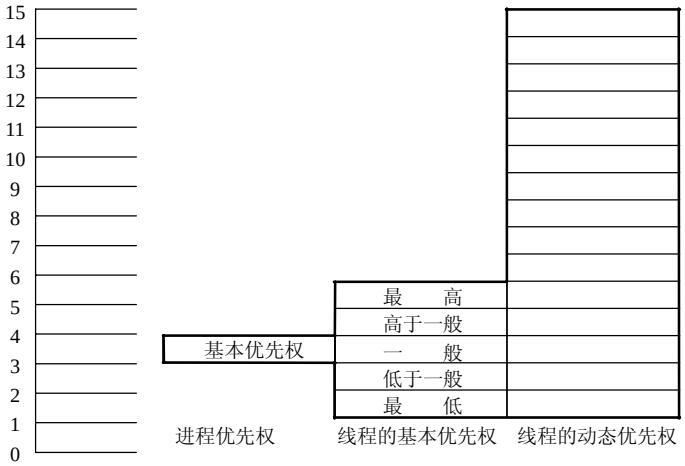


图 6.23　Windows NT 优先权关系举例

2. 多处理机调度

NT 运行在单处理机上时，最高优先权的线程总是活动的，除非它等待某事件。如果有不止一个线程有最高优先权，那么处理机就分享，按时间片轮转。在多处理机系统中若有 N 个处理机，那么(N–1)个最高优先权线程是活动的，互斥地在(N–1)个处理机上运行，剩下的低优先权线程分享剩余的一个处理机。例如，有 3 个处理机，2 个最高优先权线程运行在 2 个处理机上，所有的其他线程分享剩余的一个处理机。

以上策略受处理机和线程之间的亲密度影响。如果线程已就绪，但惟一可用的处理机不在其亲密集中，那么该线程被迫等待，调度下一个可用线程。

6.5.3　MVS

如前所述，MVS 的进程环境包括以下的可控制实体。

① 全局服务请求块(全局 SRB)：用来控制不在用户地址空间执行的系统任务，全局 SRB 代表了相互地址操作的系统任务。

② 地址空间控制块(ASCB)和地址空间扩展块(ASXB)：用来控制对应于一个应用或作业的地址空间，每个地址空间的工作包括系统任务和用户任务集。

③ 服务请求块(SRB)：用来控制不在用户地址空间执行的系统任务，它对该地址空间的一个任务发出的服务请求进行响应。

④ 任务控制块(TCB)：代表一地址空间的一个任务。

SRB 所代表的任务是不可抢占的，如果这种任务被中断，那么在中断处理后接受控制。与之相对，由 TCB 控制的任务可以抢占。如果它被中断，则中断处理完后，将控制转给调度者，调度另外的任务运行。

全局 SRB 保存在一个优先权递增的队列中，该队列位于系统队列区域中，该区域不能交换。因此，全局 SRB 总可以在内存中找到。当处理机可用时，MVS 首先搜索该队列以获得就绪 SRB。如果没找到，MVS 就找寻至少有一个就绪 TCB 或 SRB 的地址空间。为此，ASCB 队列要保存在系统队列区内，每个地址空间都有一整体优先权，这样，ASCB 就可按优先权降序排列。一旦选择了一个地址空间，MVS 就用该地址空间的结构(也就是局部系统队列区)，MVS 选择最高优先权的 SRB，若失败则调度最高优先权的 TCB。

MVS 中的调度优先级有 256 级，全局 SRB 调度为最高级。在给定的调度优先级内，按 FCFS 队列组织地址空间。调度优先级组成集合类，每一个集合又分为 16 个级。一般，一个地址空间被分配给一类并一直保存在该类中，在每类中，最高 6 级各自被指定一个固定优先级，低的 10 级组成一个平均等待时间群(MTTW)。在 MTTW 群中，地址空间优先权由各地址空间平均等待间隔时间决定。等待间隔短的有高优先权，长时间使用 CPU 而不涉及 I/O 的优先权较低。

当地址空间创建时，它被分配给一个执行组，要求满足该地址空间的优先权。执行组中的地址空间要通过一系列执行阶段。执行阶段允许一个事务或作业在不同时候得到不同的控制，衡量时段的方法是积累服务，控制参数改变的方法是从一个执行阶段转到另一个执行阶段。定义了一个执行组，可指定一个或多个执行阶段。每个执行阶段由其时段和优先权区分。地址空间在阶段 1 开始，并在该阶段保持其优先权直到该阶段的处理时间用完。每个执行阶段的时段由服务单元指出。一个服务单元就是一个 CPU 时间段，它根据 CPU 模式、进程数、操作系统环境的不同而不同；其范围是 1～11ms。

因此，整个工作负载分成执行组——首先根据相似类型或地址空间；其次根据不同工作负载组件的优先权。在地址空间内，按事务期望运行时间进行分化。例如，一个分时地址空间可定义如下。

·阶段 1：最高优先级，持续时间为 1000 个服务单元。

• 阶段 2：次高优先级，持续时间为 4000 个服务单元。

• 阶段 3：最低优先级，持续时间不限。

在这种结构下，我们希望较短的命令在阶段 1 内完成，给予它最高优先级最好的访问系统服务。较长的命令在阶段 2 完成，也有较好的服务，这种命令比最短命令要复杂，其中有些可以并行执行。最长的命令可能触发一个编译器或一个应用程序，它将运行在最低服务环境中。

6.6 小　　结

对于进程的执行，操作系统必须作出 3 个决定：长程调度、中程调度以及短程调度，本章主要讨论短程调度。

对于短程调度的设计可面向用户，也可面向系统。用户主要关心响应时间，而系统关心吞吐量和处理机利用率。

短程调度的算法有：先来先服务(FCFS)、时间片轮转、最短进程优先、最短剩余时间优先、最高响应比优先、多级反馈队列等。调度算法的选择取决于期望的性能及实际应用。

在多处理机系统中，处理机可共享内存，各调度算法的性能差别不大。

实时进程和任务要与外部事件交互，要满足一定的时限，实时操作系统就是要处理实时进程，关键在于满足时限。

附录　响应时间

响应时间是系统对一个输入的反应时间。在一个交互系统中，可定义为用户最后敲打键盘和计算机开始显示的时间。对于不同的应用程序，定义有所不同。一般，它是系统响应一个请求执行特定任务的时间。

理想情况下，响应时间越短越好。然而，通常响应时间越短，代价越大，代价来源于两方面：

① 计算机运行能力。计算机速度越快，响应时间就越短，当然，代价越大。

② 竞争环境。一些进程响应时间快了，另一些则慢了。

这样，对于一给定响应时间的评价必须相对于它的代价而言。

表 6.12 列出了 6 种响应时间。设计的困难来自于当要求响应时间低于 1s 时。这种 1s 内的响应时间来自于系统控制或和实时的外部行为交互，如装配线等。

这个快的响应时间是交互应用中的关键，这也被许多研究所证实。这些研究表明当一个计算机和用户交互，这种交互无需任一方等待，生产率显著地提高，因而

计算机上的工作量下降，性能趋于改善。过去通常认为一个不到 2s 的相对慢一点的反应在大多数应用中是可以接受的，因为人们总是想着下一个任务。然而，现在却表明当快速的响应时间得到时，生产率也提高了。

对于响应时间的报告结果是基于在线的处理分析。一个处理包括一个用户终端命令和系统的回答。这是在线用户工作的基本单元，可以分为两个时间顺序。

① 系统响应时间。用户输入命令到终端显示完整回答的时间长度。

② 用户响应时间。用户收到完整回答到下一命令输入的时间。通常指思考时间。

表 6.12　响应时间的幅度

> 15s	这排除了对话交互的时间。对于某些应用，某些用户可能满足于坐在一个终端旁花 15s 以上时间等待一个简单的回答。然而，对于一个忙人来说，15s 以上的等待是不能容忍的。如果这种延迟发生了，系统应该设计成用户可以在以后的时间里做其他的事。
> 4s	这对于一个谈话来说一般是太长了，它要求操作者保留信息在低级内存(操作者内存，不是计算机的)。这样，这种延迟限制了实时任务和数据通信任务。
2~4s	大于 2s 的延迟限制了要求高度集中的终端操作。在终端上等待 2~4s 对于一个专心于完成工作的用户来说太长了。
< 2s	当终端用户要保存所有回答信息时，响应时间必须很短。信息越具体，对响应时间小于 2s 的要求越高。对于精细的终端事务，2s 是一个重要的界限。
< 1s	某些思考密集型工作，如图形应用，需要非常短的响应时间以使用户的兴趣和注意力长时间保持。
< 1/10s	按键并回显或按鼠标键并反映要求即时性(小于 0.1s)，用鼠标交互要求极快的响应。

习　题　六

6.1　考虑下列进程：

进程名	到达时间/s	执行时间/s
1	0	3
2	1	5
3	3	2
4	9	5
5	12	5

根据表 6.5 和图 6.5 对此进行分析，并给出分析结果。

6.2　同题 6.1 对下列情况进行分析：

进程名	到达时间/ms	执行时间/ms
A	0	1
B	1	100
C	2	1
D	3	100

6.3　证明：在非抢占式调度算法中，SPN 具有最小平均等待时间。

6.4　在一个非抢占式单处理机系统中，就绪队列在时间 t_0 时有 3 个进程，它们分别在 t_1，t_2，t_3 时刻到达，估计运行时间为 t_{r_1}，t_{r_2}，t_{r_3}。图 6.24 表明了它们的响应比随时间线性变化。利用这个例子找出一个响应比调度的方法，叫做最大响应比最小化调度，它使一组给定的进程的最大响应比最小，不考虑后续到达。

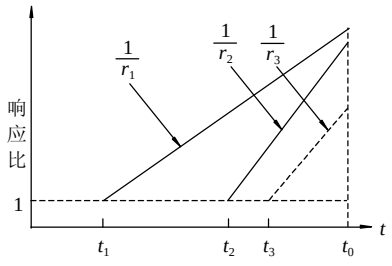


图 6.24　响应比作为时间的函数

提示：首先决定哪一个进程最后调度。

6.5　证明：最大响应比最小化调度算法使一组给定进程的最大响应比最小。

提示：考虑将达到最大响应比的进程和之前的所有进程。考虑相同组的进程调度，观察最后一个执行的进程的响应比。注意这一组现在可能包括整个组中其他的进程。

6.6　定义响应时间 t_R 是一个进程等待和服务的平均时间。证明：对于 FIFO，$t_R=t_s/(1-\rho)$，设平均服务时间为 t_s。

6.7　一个处理机被所有的就绪队列中的进程所复用，进程以无限的速度运行(这是一个理想的时间片轮转调度，它使用的时间片和要求服务的平均时间相比非常小)。证明：对于一个无穷的泊松分布输入，一个进程的平均响应时间 t_{Rx} 和服务时间 t_x，有

$$t_{Rx}=t_x/1-\rho$$

提示：考虑系统在给定输入下的平均队列长度。

6.8　在一个排队系统中，新作业在服务前必须等待。等待时，它的优先级从 0 开始随时间以 α 斜角线性增加，作业一直等待到优先级等于正在被服务作业的优先级，然后，加入被服务作业共享处理机，同时优先级以 β 线性增加。该算法称为自私时间片轮转，因为被服务进程优先级不断增加来垄断处理机，试根据图 6.25，证明：作业服务时间为 t_x，平均响应时间为 t_{Rx}，且

$$t_{Rx}=\frac{t_s}{1-\rho}+\frac{t_x-t_s}{1-\rho'}$$

$$\rho=\lambda t_s \quad \rho'=\rho\left(1-\frac{\beta}{\alpha}\right) \quad 0\leqslant\beta\leqslant\alpha$$

假设到达和服务时间分别服从 $\frac{1}{\lambda}$ 和 t_s 的指数分布。

提示：考虑整个系统及两个子系统。

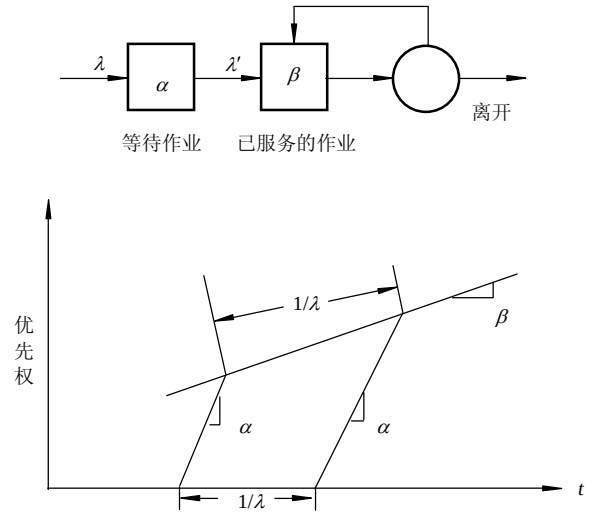
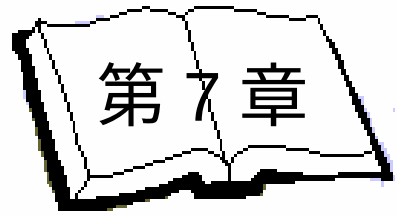


图 6.25　自私的时间片轮转法

6.9　一个交互系统用时间片轮转调度，试图给那些小进程足够的响应时间。当结束所有就绪进程一轮后，系统用最大响应时间除以进程数作为下一轮的时间片，这可行吗？

6.10　哪种进程最适合于多级反馈队列调度——偏重处理机型或偏重 I/O 型？简述之。

6.11　在基于优先级的调度中，仅当没有优先级更高的进程时，才有一个进程获得调度。假设没有其他调度信息，且优先级在创建进程时产生，以后不变。在此系统中，使用 Dekker 的互斥方法很危险，为什么？考虑会有什么不希望的事情发生，如何发生。



I/O 设备管理

在计算机系统中，使用了大量的完全不同的 I/O 设备，它们的特点和操作方式也完全不一样。但是，所有的 I/O 设备都通过设备管理程序来管理。设备管理是计算机操作系统中最繁杂的且与硬件紧密相关的部分。正如 A.M.Lister 所说“ 从传统上看，I/O 被以为是 OS 设计中最逊色的领域之一，在这个领域中规范化很困难而且特定方法太多 ”。要把大量设备精减成一个单一模块的 I/O 系统，必须要全面地适应已有设备的需求，从简单的鼠标到键盘、打印机、图形显示终端、硬盘驱动器、CD-ROM 驱动器，以至于网络，同时也必须考虑到未来的存储和输入技术。

7.1 I/O系统硬件

7.1.1 I/O 设备

计算机所管理的 I/O 外部设备可以分为以下 3 类：

① 用户可读设备，用于用户与计算机通信。例如，视频显示终端，它是由显示器、键盘、还有鼠标和打印机组成。

② 机器可读设备，用于电子装置与计算机通信。例如，磁带、磁盘、控制器和感应器等。

③ 通信设备，用于与远程设备通信。例如，modem，ISDN 终端。

所有这些设备其属性和类别有很大的区别，其主要区别在于：

① 数据传输速度。它们的数据传输速度有相当大的区别。例如，键盘约 0.01Kb/s，激光打印机约 100Kb/s，图形显示则约为 30 000Kb/s。

② 应用。被接入设备的使用影响着 OS 和支持工具中的软件和策略。例如，磁盘需要文件管理软件才能处理文件请求，在虚拟内存模式下，磁盘对页面的转储需要虚拟内存硬件和软件；更进一步，这些磁盘应用都关注调度策略。又例如，一个

终端设备可能是一般用户使用，也可能是超级用户使用，当然要求不同的权限和优先级。

③ 控制的复杂性。打印机只需要一个简单的控制接口，磁盘则复杂得多。

④ 传输单元，设备分字符设备和块设备。所谓字符设备即指传输单元为字符流或字节流的设备，而块设备则指传输单元为数据块的设备。字符设备属于无结构设备，如交互式终端、打印机等，其特征为，数据传输速度较低，不可寻址，在 I/O 时常采用中断驱动方式。块设备的特征是信息的存取以数据块为单位，是一种有结构设备，如磁盘等，其特征是：传输速度较高，可寻址、可随机读写，磁盘的 I/O 常采用 DMA 方式。

⑤ 数据描述。不同设备使用不同的数据编码模式，例如，字符代码。

⑥ 错误条件。对于不同的错误种类，出错的报告方法，出错的后果和影响范围是不同的。

这些差异使得不论从操作系统的角度还是从用户进程的角度都难以获得一个规范的、一致的 I/O 的解决方案。

7.1.2　设备控制器

I/O 设备一般由机械和电子两部分组成，通常将这两部分分开处理，以提供更加模块化、更加通用的设计。电子部分称作设备控制器或适配器(Device Controller 或 Adapter)。机械部分就是设备本身，控制器通过电缆与设备内部相连。很多控制器可以连接 2 个、4 个，甚至 8 个相同的设备。如果控制器和设备之间的接口采用的是标准接口(符合 ANSI、IEEE、ISO 或者企业标准)，那么各厂商就可以制造各种适合这个接口的控制器或设备。例如：许多厂商生产与 IBM 磁盘控制器接口相适应的磁盘驱动器。

图 7.1 描述了一个微机的多 I/O 总线的组织。

从图 7.1 不难发现，设备并不直接与 CPU 进行通信，而是与设备控制器通信。因此，在设备与控制器之间应有一接口。该接口中有 3 类信号：数据信号、控制信号、状态信号。

设备控制器是一个可编址设备。当它仅控制一台设备时，它只有一个惟一的设备地址；若控制器连接多台设备时，则应具有多个设备地址，使每一个地址对应一台设备。设备控制器的复杂性因设备而异，相差甚大，可把设备控制器分成两大类：一类是用于控制字符设备的；另一类是用于控制块设备的。

例如，磁盘控制器的任务是把串行的位流转换为字节块，并进行必要的校验工作。首先，控制器按位进行组装，然后存入控制器内部的缓冲区中形成以字节为单位的块，在对块验证检查并证明无错误后，再将它复制到主存中。

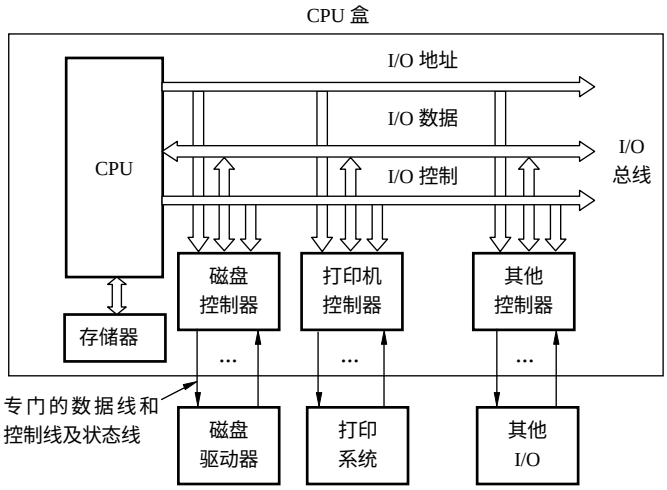


图 7.1　微机的 I/O 组成

CRT 终端控制器的任务是从内存中读出待显示字符的字节，并产生用来调制 CRT 电子束的信号，以便将结果写到屏幕上。控制器还产生使 CRT 电子束在完成一行扫描后做水平回扫的信号，以及整个屏幕扫描结束再做垂直回扫的信号。如果没有 CRT 控制器，操作系统程序员只能编写程序去模拟显像管的扫描。如果有控制器，则操作系统只要用几个参数对其初始化即可。这些参数包括每行字符数，每屏的行数，并使控制器实际驱动电子束。

每个控制器有几个寄存器用来与 CPU 通信，因此，需要提供寻址的机制。用于 I/O 功能的地址集合称为 I/O 空间。计算机的 I/O 空间除包括所有设备的寄存器外，还包括一些在主存中用以映射设备(如图形终端)的帧缓冲区。每个寄存器在 I/O 空间中都有一个定义好的地址。在有些计算机中，这些寄存器地址是常规存储器地址空间的一部分。这种模式称为存储器映像(Memory-mapped)I/O。例如，Motorola 680x0 就采用这一方法，而其他计算机中则使用专用的 I/O 地址空间，每个控制器分配其中的一部分，不同的 I/O 控制器分配不同的 I/O 地址，对应不同的中断向量，如 Intel 80x86 就是采用这种模式。

设备控制器的主要作用有：

① 接收和识别 CPU 发来的多种不同命令；

② 实现 CPU 与控制之间、控制器和设备之间的数据交换；

③ 记录和报告设备的状态。

设备控制器的组成如图 7.2 所示。其中的 I/O 逻辑用于对设备的控制，它通过一组控制线与处理机交互，处理机利用该逻辑向控制器发送 I/O 命令；I/O 逻辑对收到的命令进行译码。每当 CPU 要启动一个设备时，就将启动命令发送给控制器；同时

又通过地址线把地址送给控制器，由控制器的 I/O 逻辑对收到的地址进行译码，再根据所译出的命令对所选设备进行控制。

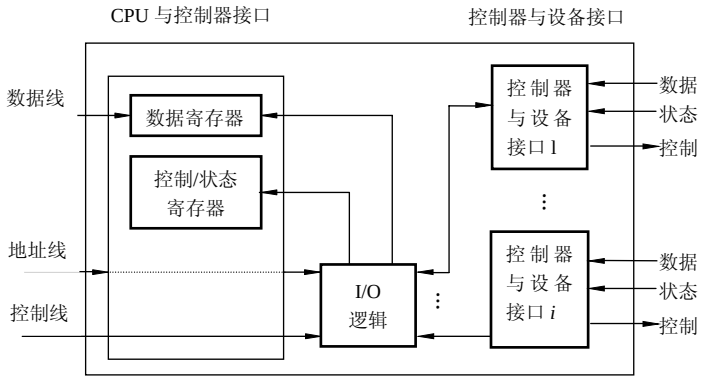


图 7.2　设备控制器的组成

7.1.3　I/O 技术

1. I/O 技术

目前，操作系统中的 I/O 技术大致有以下 3 种：

(1) 程序 I/O

处理器根据用户进程编制的 I/O 语句或指令，向 I/O 设备块(包括设备和设备控制器)发出一个 I/O 命令。在 I/O 处理完成之前，进程处于“忙等待”状态，当 I/O 处理完成后，进程继续执行。

(2) 中断驱动 I/O

处理器根据进程中的 I/O 要求，发出一个 I/O 命令来启动 I/O 设备。这时，如果进程不需等待 I/O 完成，则进程继续执行后续指令序列；否则，该进程被挂起，直到中断到来才解除挂起状态，然后继续其他工作。

(3) 直接存储器存储(DMA)

一个 DMA 模块直接控制主存和 I/O 设备块之间的数据交换。处理器仅向 DMA 模块发送一个数据传输请求，并且仅仅在整个数据块传送完毕后发送中断给处理器，显然，DMA 在传输数据块过程中无需 CPU 的任何干预。

随着计算机系统的发展，单个部件也越来越复杂。最明显的例子就是 I/O 技术的发展，I/O 技术的发展阶段如下所述：

① 处理器直接控制边缘设备。如简单的处理器控制的设备。

② 增加一个控制器或 I/O 模块。处理器使用无中断的编程控制 I/O。处理器开

始和外部设备接口的特殊细节脱离。

③ 使用了如阶段②的设置，但增加了中断。处理器不必花费时间等待 I/O 操作执行，因此提高了效率。

④ I/O 模块通过 DMA 直接控制内存。除了在传输开始和结束的时候，它都可以不通过处理器把数据块移入或移出内存。

⑤ I/O 模块由一个单独处理器处理，有专门用于 I/O 的指令集。中央处理单元(CPU)控制 I/O 处理器在主存中执行 I/O 程序。I/O 处理系统无需 CPU 干预直接取出和执行这些指令。这使得 CPU 形成一个 I/O 活动序列，并且只有在整个序列执行完时才能中断。

⑥ I/O 模块有本地存储器，事实上，有其自己的计算机。在这种体系结构中，大量的 I/O 设备仅由很小一部分的 CPU 控制。这种体系结构普遍应用于交互终端的通信。I/O 处理器管理包括控制终端在内的大部分工作。

可以看出，越来越多的 I/O 模块不需要 CPU 就可执行。中央处理器逐渐与 I/O 相关任务脱离，改善了性能。在最后两个阶段，随着 I/O 模块这个概念的产生，I/O 功能发生了巨大的变化。

注意：对于阶段④至阶段⑥中提到的所有模块，DMA 都是适用的，因为各种模块都可对主存进行直接控制。同样的，阶段⑤中的 I/O 模块又称为 I/O 通道(I/O Channel)，阶段⑥中的 I/O 模块为 I/O 处理器(I/O Processor)。这两个术语也可通用于这两个阶段。

随着 I/O 功能的发展，I/O 模块已是一个 I/O 计算机，控制着许多设备控制器，整个控制器又控制着许多设备。

2. DMA

DMA 是一种优于中断方式的 I/O 控制方式，其特点为：数据传输的基本单位是数据块，即 CPU 与 I/O 设备之间，每次至少传送一个数据块；所传送的数据是从设备直接送入内存的，或者相反；仅在传送一个或多个数据块的开始和结束时，向 CPU 发中断信号，请求 CPU 干预，整块数据的传送是在控制器的控制下完成的。

磁盘控制器通过 DMA 读取磁盘数据的过程是，当 CPU 要读取一数据块时，CPU 向控制器发一条读命令，并向控制器提供读出块的磁盘地址、读出块被送往内存的起始地址和要传输的字节数，控制器将整个块从设备写入其内部缓冲区进行校验，然后，将第一个字节或字拷贝到由 DMA 存储地址指定的内存地址处。接着，控制器按刚刚传送的字节数对 DMA 地址和 DMA 计数器分别进行步增和步减操作，重复这一过程，直到 DMA 计数变成 0，此时，控制器发中断信号，操作系统开始工作，但操作系统不必将块拷贝到内存，因为它已经被写入内存了。

图 7.3 描述了一个 DMA 控制器组成的示意图。其中，数据计数器 DC 用于存放本次 CPU 要读/写的字(节)数；数据寄存器 DR 用于暂存从设备到内存或内存到设备

的数据；内存地址寄存器 MAR 用于存放数据从设备传送到内存的目的地址，或由内存传到设备的内存源地址；控制逻辑，由 DMA 接受或产生控制和状态信号。

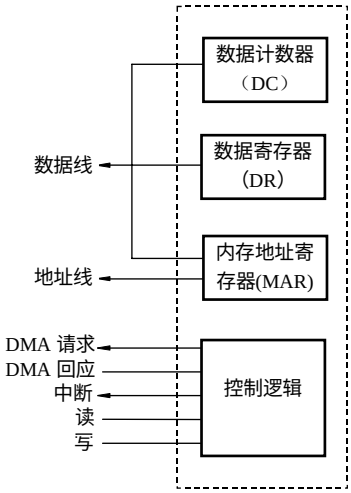


图 7.3　DMA 控制器组成

DMA 可以有很多配置方案(见图 7.4)。在图 7.4(a)中，所有模块共享同一系统总线。DMA 模块作为代理 CPU，使用编程控制 I/O 在存储器和 I/O 模块间通过 DMA 模块交换数据，这种配置虽然廉价，但效率不高，传送一个字要两个总线周期。

把 DMA 和 I/O 函数集成，可以大量削减需要总线周期的数目。如图 7.4(b)所示，在 DMA 模块和一个或多个 I/O 模块间建立一个路径。DMA 可以成为 I/O 模块的一部分，也可以成为单独模块，控制一个或多个 I/O 模块。更进一步，可以通过一个 I/O 总线将 I/O 模块和 DMA 模块连接起来（见图 7.4(c))。这可以减少 I/O 的接口数，提供更易扩展的配置。在图 7.4(b)、(c)中，系统总线处于 DMA 模块和 CPU 主存之间，DMA 使用系统总线与主存交换数据，与 CPU 交换控制信号。DMA 与 I/O 模块之间的数据交换不在系统总线上。

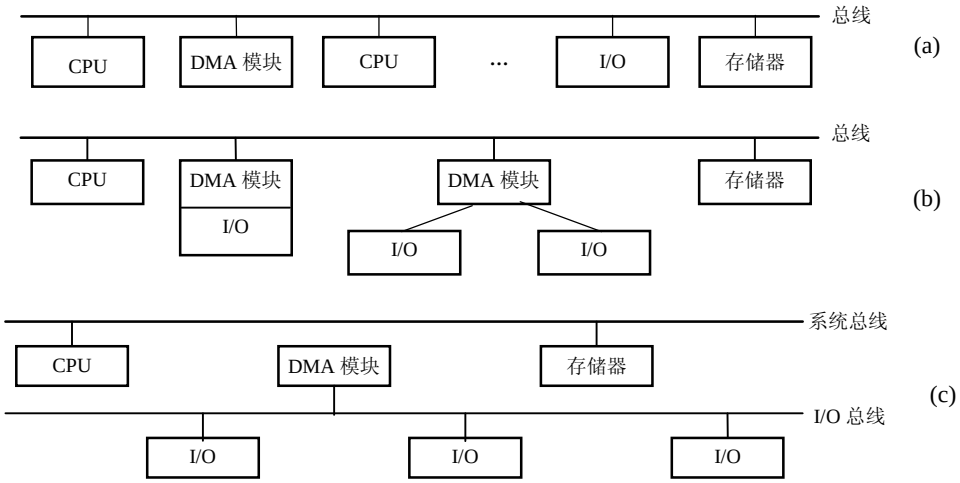


图 7.4　可能的 DMA 构型

(a) 单总线，分离 DMA　(b) 单总线，集成 DMA-I/O　(c) I/O 总线

3. I/O 通道

在大型计算机中，虽然 CPU 和 I/O 设备之间的设备控制器大大减轻了 CPU 的

负担，但在外部设备太多时，CPU 依然负担较重。为此，设计了一个专门负责外设 I/O 的处理器，置于 CPU 和设备控制器之间，称这个 I/O 处理器为 I/O 通道。设计目的是：建立独立的 I/O 操作，使数据的传送独立于 CPU，并尽量使有关 I/O 操作的组织、管理及结束也独立，以保证 CPU 有更多时间进行数据处理。或者说，是想使一些原来由 CPU 处理的 I/O 任务转由通道来承担，从而把 CPU 从繁杂的 I/O 任务中解脱出来。在设置通道后，CPU 只需向通道发一条 I/O 指令。当通道完成了规定的 I/O 任务后，才向 CPU 发出中断信号。

I/O 通道同 CPU 一样，有运算和控制逻辑，有累加器、寄存器，有自己的指令系统，也在程序控制下工作。它的程序是由通道指令组成的，称为通道程序。I/O 处理器和 CPU 共享主存储器。在微型计算机中，其 I/O 处理器并不完全具有前述 I/O 通道的所有功能，因此，就称为 I/O 处理器。

在大型计算机中常有多个 I/O 通道，而在一般的微型计算机中则可以配置 1~2 个 I/O 处理器(或更多)。这些 I/O 处理器和中央处理器共享主存储器和总线(微型机中采用总线结构)，在大型机中就可能出现几条通道和中央处理器同时争相访问主存储器的情况。为此给通道和中央处理器规定了不同的优先次序，当通道和中央处理器同时访问主存时，存储器的控制逻辑按优先次序予以响应。通常中央处理器被规定为最低优先级。而在微型计算机中，系统总线的使用是在中央处理器控制下，当 I/O 处理器要求使用总线时，向中央处理器发出请求总线的信号，中央处理器就把总线使用权暂时转让给 I/O 处理器。

通道通过执行通道程序，并与设备控制器一起共同实现对 I/O 设备的控制。例如：当 CPU 要完成一组相关的读(或写)操作及有关控制时，只需向 I/O 通道发出一条 I/O 指令，以给出其所要执行的通道程序的首址和访问的 I/O 设备，通道接到该指令后，通过执行通道程序便可完成 CPU 指定的 I/O 任务。

通道程序是由一系列的通道指令(或称为通道命令)所构成。通道程序由 CPU 按数据传送的不同要求自动形成，通道程序常常只包括少数几条指令。CPU 生成的通道程序存放在主存储器中，并将该程序在主存中的起始地址通知 I/O 处理器。在大型计算机中，通道程序的起始地址常存放在一个称为通道地址字(CAW)的主存固定单元中，而在微型计算机中，常将此起始地址存放在主存中的 CPU 与 I/O 处理器的通信区中。每一条通道指令称为通道命令字 CCW。

通道作为处理器，也有说明处理器状态的通道状态字 CSW。CSW 中包含有该通道及与之相连的控制器和设备的状态、以及数据传输的情况。

一般 I/O 通道有 3 种类型：

(1) 字节多路通道

在这种通道中，通常都含有许多非分配型子通道，其数量可从几十到数百个，每一个子通道接一台 I/O 设备。这些子通道按时间片轮转方式共享主通道，当每一

个子通道控制其 I/O 设备完成一个字节的交换后，便立即腾出字节多路通道(主通道)给第二个子通道使用；当第二个子通道也交换完一个字节后，又依次地把主通道让给第三个子通道；依此类推。当所有子通道轮转一周后，又返回来由第一个子通道去使用字节多路通道。这样，只要字节多路通道扫描每个子通道的速率足够快，而连接到子通道上的设备的速率不是太高时，就不会丢失信息。

(2) 数组选择通道

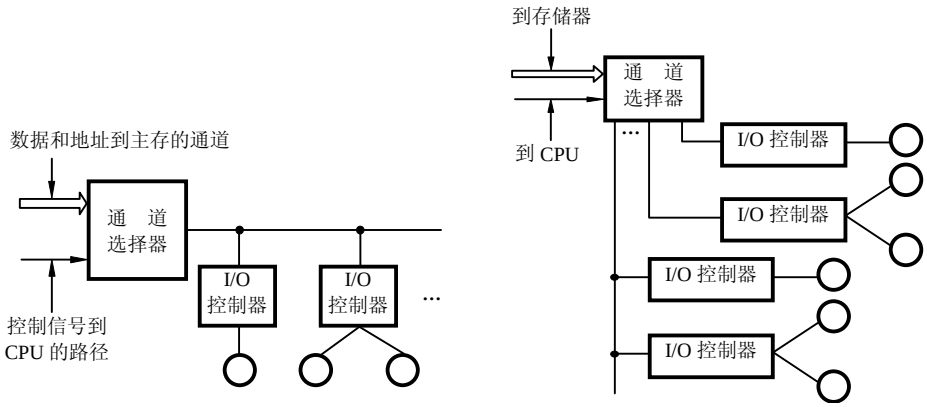
字节多路通道不适于连接高速设备，这推动了按数组方式进行数据传送的数组选择通道的出现。这种通道虽然可以连接多台高速设备，但由于它只含有一个分配型子通道，在一段时间内只能执行一道通道程序、控制一台设备进行数据传送，致使当某台设备占用了该通道后，便一直由它独占，即使无数据传送(通道被闲置)，也不允许其他设备利用该通道，直至该设备传送完毕并释放该通道。可见，这种通道利用率很低。

(3) 数组多路通道

数组选择通道虽有很高的传输速率，但每次只允许一个设备传输数据。数组多路通道将数组选择通道的高传输速率和字节多路通道的各子通道(设备)分时并行操作的优点相结合，从而形成了一种新通道。它含有多个分配型子通道，因而，这种通道既具有很高的数据传输速率，又能获得令人满意的通道利用率。也正因为如此，该通道被广泛地用于连接多台高中速的外围设备，其数据传送按数组方式进行。

I/O 通道方式的发展，既可进一步减少 CPU 的干预，又可实现 CPU、通道和 I/O 设备 3 者的并行工作，从而更有效地提高了整个系统的资源利用率。

图 7.5 说明了两种常见的 I/O 通道。



(a)　　(b)

图 7.5　I/O 通道体系结构

(a) 选择器通道　(b) 多路通道

7.2　I/O软件

I/O 软件的总体目标是，按分层的思想构造软件，较低层的软件要使较高层的软件独立于硬件，较高层的软件则要向用户提供一个友好、规范、清晰的界面。I/O 软件设计的具体目标是：

① 设备独立性。让应用程序独立于具体使用的物理设备。例如，在编写使用软盘、硬盘上文件的程序时，无需为每一种具体的设备类型而修改程序。甚至不必重新编译就能将程序移到它处运行。

② 统一命名。它与设备独立性相关。一个文件或一个设备的名字应该是一个简单的字符串或一个整数，它不应依赖于设备。在 UNIX 系统中，所有磁盘都能以任意方式集成到文件系统层次中，因此，用户不必知道哪个名字对应于哪台设备。例如，一个软盘可以安装到目录/usr/ast/backup 下，这样拷贝一个文件到/usr/ast/backup/Monday 就是将文件拷贝到软盘上。用这种方法，所有文件和设备都使用相同的方式(路径名)进行定位。

③ 同步/异步传输。大多数物理 I/O 是异步的——CPU 启动传输后便转去做其他工作，直到中断发生。如果 I/O 操作是异步的，在发出读/写命令后，程序将自动被挂起，直到缓冲区中的数据准备好或使用完毕。

④ 出错处理。一般来说，错误应尽可能地在接近硬件的层上处理。如果控制器发现了一个读错误，自己没法纠正，那么设备驱动程序应当予以处理。很多错误是偶然性的，例如，磁盘读/写头上的灰尘导致读/写错误时，重复该操作，错误就会消失。只有在低层软件处理不了的情况下，才将错误上交高层处理。

⑤ 设备共享与独占。有些 I/O 设备，如磁盘，能够同时让多个用户使用，但有些 I/O 设备，如打印机等设备，必须由单个用户独占使用，直到该用户使用完，另一个用户才能使用。独占设备的引入也带来了各种各样的问题，操作系统必须妥善管理共享设备和独占设备，以避免出现问题。

根据 I/O 软件的设计目标，将 I/O 软件组织成以下 4 个层次：中断处理程序、设备驱动程序、与设备无关的操作系统软件和用户层软件。

下面由低到高逐层讨论 I/O 软件，最后介绍缓冲技术。

7.2.1　中断处理程序

在现代计算机系统中，对 I/O 设备的控制，广泛采用中断驱动(Interrupt-driven)方式，即当某进程要启动某个 I/O 设备工作时，便由 CPU 向相应的设备控制器发出一条 I/O 命令，然后立即返回继续执行原来的任务。设备控制器便按照该命令的要求去控制 I/O 设备，此时，CPU 与 I/O 设备并行操作。例如，在输入时，当设备控

制器收到 CPU 发来的读命令后，便准备接收输入数据，一旦数据进入数据寄存器，控制器便通过控制线向 CPU 发送一中断信号，由 CPU 检查输入过程中是否出错，若无错，便向控制器发送取走数据的信号，然后通过控制器及数据线将数据写入指定内存单元。可见，中断驱动方式可以成倍地提高 CPU 的利用率。

无论是哪种 I/O 设备，其中断处理程序的处理基本相同，其步骤为：

① 唤醒被阻塞的驱动进程。当中断处理程序开始执行时，必须唤醒被阻塞的驱动进程。某些系统，是根据信号量执行 PV 操作；某些系统，是对管程中的条件变量执行 Signal 操作；还有一些系统，是向阻塞的进程发一个消息。

② 保护被中断进程的 CPU 环境。通常由硬件自动将处理机状态字 PSW 和程序计数器 PC 中的内容保存在堆栈中，然后，对被中断的进程的 CPU 现场进行保留(压栈)，另外所有的 CPU 寄存器也需要保留。

③ 分析中断原因，转入相应的中断处理程序。由 CPU 来确定引起本次中断的中断设备，并发送一个应答信号给发中断请求的进程，使之撤消该中断请求信号。然后，将中断处理程序的入口地址送入程序计数器 PC 中，处理机转向中断处理程序。

④ 进行中断处理。执行中断处理程序中规定的功能。

⑤ 恢复现场。当中断处理完毕后，从中断栈中取出中断前保留的信息并恢复到对应部件中，使被中断的程序得以继续执行。

7.2.2　设备驱动程序

所有与设备相关的代码放在设备驱动程序中。由于驱动程序与设备硬件密切相关，故应为每一类设备配置一种驱动程序，或为一类密切相关的设备配置一个驱动程序。例如，系统支持若干不同品牌的终端，但它们只有很细微的差别，则最好是仅设计一个终端驱动程序。另一方面，若系统支持的终端性能差别很大，如笨拙的硬拷贝终端与带有鼠标的智能位映像图形终端，则必须为它们分别设计不同的终端驱动程序。

1．设备驱动程序的功能

设备驱动程序的功能有：

① 将接收到的来自它上一层的与设备无关的抽象请求转为具体请求。

② 检查用户 I/O 请求的合法性，了解 I/O 设备的状态，传递有关参数、设置设备的工作方式。

③ 发出 I/O 命令，启动分配到的 I/O 设备，完成指定的 I/O 操作。

④ 及时响应控制器或通道发来的中断请求，并调用相应的中断处理程序处理。

⑤ 对于有通道的计算机系统，驱动程序还应能根据用户 I/O 请求构成通道程序。

2. 设备驱动程序的处理过程

一般地说，设备驱动程序的任务是接收来自它上面一层的与设备无关软件的请

求，并执行这个请求。一个典型的请求是“ 读第 *n* 块 ”，如果请求到来时驱动程序是空闲的，则立即开始执行该请求；若驱动程序正在执行一个请求，则将新到来的请求插到一个等待处理 I/O 请求队列中。

以磁盘为例，实现一个 I/O 请求的第一步是将这个抽象请求转换成具体的形式，对于磁盘驱动程序来说，要计算请求块实际在磁盘的位置，检查驱动器的电机是否正在运转，确定磁头臂是否定位在正确的柱面上，等等。

一旦明确应向控制器发哪些命令，它就向控制器中写入这些命令。一些控制器一次只能接收一条命令，另一些控制器则可接收一个命令链表，然后自行控制命令的执行，不再求助于操作系统。

在设备驱动程序发出一条或多条命令后，系统有两种处理方式。多数情况下，设备驱动程序必须等待控制器完成操作，所以驱动程序阻塞自己，直到中断信号将它解除阻塞为止。有的情况可以通过操作完成，驱动程序不需阻塞，例如，某些终端的滚屏操作，只要把几个字节写入控制器的寄存器中即可。对前一种情况，被阻塞的驱动程序将由中断唤醒，而后一种情况是驱动程序不进入睡眠状态。上述任一种处理方式，在操作完成后，都必须检查是否有错。若一切正常，则设备驱动程序负责将数据传送到与设备无关的软件层。最后，它向调用者返回一些用于错误报告的状态信息。若还有其他未完成的请求在排队，则选择一个启动执行。若队列中没有未完成的请求，则该驱动程序等待下一个请求。

7.2.3　与设备无关的 I/O 软件

虽然一些 I/O 软件是与设备相关的，但大部分软件是与设备无关的，设备驱动程序与设备独立软件之间的确切界限依赖于具体系统，因为对于一些本来应按照设备独立方式实现的功能，出于效率和其他原因，实际上还是由设备驱动来实现的。与设备无关软件的基本任务是实现一般设备都需要的 I/O 功能，并且向用户层软件提供一个统一的接口。与设备无关软件层通常应实现的功能为：设备驱动程序的统一接口、设备命名、提供一个与设备无关的块大小、缓冲、块设备的存储分配、分配和释放独占设备、错误报告等。

与设备无关的 I/O 软件系统称为 I/O 子系统。I/O 子系统执行着与设备无关的操作。同时 I/O 子系统为用户应用程序提供一个统一的接口。下面讨论 I/O 子系统所需完成的主要功能。

1. 设备命名

如何给文件和 I/O 设备这样的对象命名是操作系统要解决的一个问题。与设备无关的软件(即 I/O 子系统)就负责把设备的符号名映射到相应的设备驱动程序。显然，设备的命名越简单越好。

设备命名后，所有设备的名字的集合称做设备的名字空间。设备的名字空间说

明了不同设备是如何被标识和引用的。UNIX 系列有 3 种不同的名字空间。

(1) 主次设备号

内核采用数字方法来命名设备，主次设备号是内核使用的一种设备命名法：用设备类型描述加上两个称之为主、次设备号的数字来标识设备。

(2) 内部号与外部号

内部设备号标识设备驱动程序，并且是设备开关表中的索引号，外部设备号构成用户可见的设备表示，它存储在设备特殊文件的 *i* 节点(基本文件目录数据块)中。

对大多数系统来说，内部号和外部号是不一样的，于是内核维护一个由外部号到内部号的映射表格。

(3) 设备文件与路径名

要使用路径名，就要引入设备文件的概念，把设备作为文件来处理。UNIX 为文件和设备提供了一致的接口，并把设备称作特殊文件，每个特殊文件与特定的设备相关联，它可以放在文件系统中任何位置，但习惯上所有的设备都位于/dev 目录或其子目录之下。

从用户的角度来看，设备文件和普通文件没有太大的区别。但从系统内部看，两者有着很大区别：设备文件在磁盘上没有数据块，但设备文件在文件系统中有一个永久的 *i* 节点，通常是在根文件中，在 *i* 节点的 di-mode 中标明文件类型是 IFBLK(块设备)或 IFCHR(字符设备)，而在 di-rdev 的域中存放着该设备的主设备号和次设备号，于是内核可以根据用户设备的路径名得出内部设备名(主设备号、次备设号)。

2. 设备保护

与设备命名机制密切相关的是设备保护。系统如何防止无权存取设备的用户存取设备呢？在大多数大型计算机系统中，用户进程对 I/O 设备的访问是完全禁止的。UNIX 系统则使用更灵活的模式，对应于 I/O 设备的特殊文件通常用“ rwx ”位进行保护，系统管理员可以为每一个设备设置适当的存取权限。

3. 与设备无关的块

不同的磁盘可以采用不同的扇区尺寸，与设备无关软件的一个任务是向较高层软件屏蔽并给上一层提供大小统一的块尺寸，例如，将若干扇区合并成一个逻辑块。这样，较高层的软件只与抽象设备打交道，不考虑物理扇区的尺寸而使用等长的逻辑块。对其他字符设备也是如此。

4. 设备分配

一些设备，如磁盘驱动器，在任一时刻只能被单个进程使用。这就要求操作系统对设备使用请求进行检查，并根据申请设备的可用状况决定是接收该请求还是拒绝该请求。一个简单的处理这些请求的方法是，要求进程直接通过 OPEN 打开设备的特殊文件来提出请求。若设备不可用，则 OPEN 失败。并在关闭独占设备的同时释放该设备。

5. 出错处理

出错处理是由设备驱动程序完成的。大多数错误是与设备密切相关的，因此，只有驱动程序知道应如何处理(是重试、忽略，还是报警)。一种典型的错误是，由于磁盘块受损而不能再读，驱动程序将尝试重读一定次数，若仍有错误，则放弃读并通知与设备无关的软件。从此，如何处理这个错误就与设备无关了。如果在读一个用户文件时出现错误，则将错误信息报告给调用者。若在读一些关键的系统数据结构时出现错误，比如磁盘的空闲块位图，则操作系统只能打印一些错误信息并终止执行。

7.2.4　用户空间的 I/O 软件

虽然大部分 I/O 软件都包合在 OS 内核之中，但也有一小部分 I/O 软件是由与用户程序连接在一起的库过程构成，它们可能完全运行在 OS 之外。例如，下列一个 C 程序调用了 write 库过程，并包含在运行时的二进制程序代码中：

```
count=write(fd, buffer, nbytes);
```

显然，write 库过程是 I/O 系统的组成部分。

尽管这些过程所做的工作主要是参数置换，然而确有一些 I/O 过程实现了真正的操作，特别是一些 I/O 格式的库过程。以 C 语言中的 printf 为例，它以一个格式串和一些变量作为输入，构造一个 ASCII 字符串，然后通过 write 系统调用输出这个串。对输入而言，与之类似的过程是 gets，它读出一行并返回一个字符串。标准的 I/O 库包含了许多涉及 I/O 的过程，它们都是作为用户程序一部分运行的。

上面描述的是第一类用户空间 I/O 软件，第二类用户空间 I/O 软件为 spooling(simultaneous peripheral operation on line)系统，即假脱机系统。下面讨论 spooling 的有关技术。

spooling 系统是多道程序设计系统中处理独占 I/O 设备的一种方法。系统中独占型设备的数量是有限的，往往不能满足诸多进程的需求，成为系统中的“瓶颈”资源，使许多进程由于等待某些独占设备而被阻塞。另一方面，得到独占设备的进程，在其整个运行期间，往往是占有这些设备，并不是经常地使用这些设备，因而使这些设备的利用率很低。为克服这个缺点，人们常通过共享设备来模拟独占型设备的动作，使独占型设备成为共享设备，从而提高了设备利用率和系统的效率，这种技术被称为虚拟设备技术，实现这一技术的硬件和软件系统被称为 spooling 系统，或称为假脱机系统。它分输出 spooling 和输入 spooling。

spooling 不仅可以应用在打印机上，还可以应用在其他场合。

打印设备是独占型设备，但通过 spooling 技术，可让该打印设备成为共享设备。进程要求打印输出时，spooling 系统并不是把某台打印机分配给该进程，而是在某共享设备(磁盘)上的输出 spooling 存储区中，为其分配一块存储空间，同时为该进程

的输出数据建立一个文件(文件名可缺省)。该进程的输出数据实际上并未从打印机上输出，而只是以文件形式输出，并暂时存放在输出 spooling 存储区中。这个输出文件实际上相当于虚拟的行式打印机。各进程的输出都以文件形式暂时存放在输出 spooling 存储区中，并形成了一个输出队列，由输出 spooling 控制打印机进程，依次将输出队列中的各进程的输出文件最后实际地打印输出。

由此可以看出 spooling 系统的作用如下：

(1) 实现了虚拟设备功能

宏观上，虽然是多个进程在同时使用一台独占型设备，而对每一个进程而言，都可以认为自己是独占了一个设备——一个逻辑设备。spooling 系统实现了将一个独占型设备变成多台独占型的虚拟设备。

(2) 将独占型设备变成共享设备

实际分给用户进程的不是打印设备，而是共享设备中的一个存储区(或文件)，即虚拟设备，实际的打印机由守护进程依次按某一策略逐个地打印输出 spooling 存储区中的数据。

(3) 提高了 I/O 效率

目前不但大、中型计算机的操作系统中使用 spooling 进行输入、输出工作，而且在许多微型计算机上也使用了 spooling 技术。在 spooling 系统设计中，为了弥补独占设备(如打印机)与共享设备间数据传输速度的差异，需要使用缓冲区技术，所以应注意同步与互斥的问题。

7.2.5　缓冲技术

系统为达到如下目的需要使用缓冲技术：

(1) 缓和 CPU 与 I/O 设备间速度不匹配的矛盾

由于外部设备数据的传输率与 CPU 的处理速度严重不匹配，故限制了与处理机连接的外设台数，且在中断时易造成数据丢失，极大地制约了计算机系统性能的进一步提高，限制了系统应用范围。

通常的程序都是时而进行计算，时而产生输出。如果没有缓冲，则程序在输出时，CPU 必须停下来等待；然而在计算阶段，打印机又空闲无事。显然，在打印机或控制器中置一缓冲区，用于快速地暂存程序的输出数据，以后由打印机“ 慢慢地 ”从中取出数据打印，就可使 CPU 与 I/O 设备并行工作。

(2) 减少 CPU 的中断频率，放宽对中断响应的限制

如果在 I/O 控制器中增加一个 100 个字符的缓冲区，则由中断原理可知，I/O 控制器对 CPU 的中断一直要等到存放 100 个字符的缓冲区装满以后才向处理器发一次中断，若不使用缓冲，则每个字符输入完毕均会产生一个中断。

(3) 提高 CPU 和 I/O 设备之间的并行性

缓冲的引入可显著地提高 CPU 和设备的并行操作程度，提高系统的吞吐量和设备的利用率。例如，在 CPU 和打印机之间设置了缓冲区后，可使 CPU 与打印机并行工作。

根据系统设置的缓冲区的个数，可以把缓冲技术分为单缓冲、双缓冲和循环缓冲以及缓冲池几种，如图 7.6 所示。

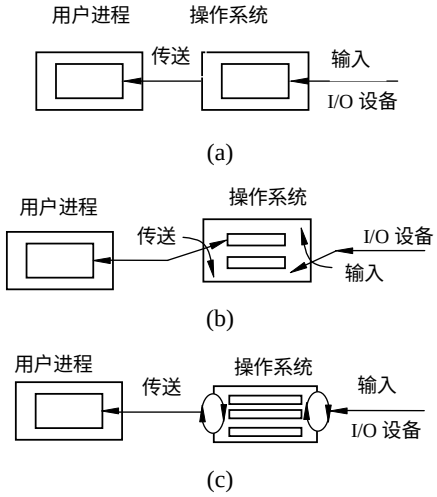


图 7.6　常见的缓冲技术
(a) 单缓冲　(b) 双缓冲　(c) 循环缓冲

1．单缓冲

单缓冲是 OS 提供的一种最简单的缓冲。当用户进程发出一个 I/O 请求时，OS 便在主存中分配一个缓冲区。

对于块设备，其输入过程如下：

先从磁盘把一块数据输入到缓冲区，设其所花费的时间为 t；然后，由操作系统将缓冲区的数据传送到用户区，设其所花时间为 t_m；接下来，由 CPU 对这一块数据进行计算，计算时间假定为 t_c，则系统对每一整块数据的处理时间为 $\max(t_c,t)+t_m$。通常 t_m 远小于 t 或 t_c，如果没有单缓冲区，数据直接进入用户区，则每一块数据的处理时间近似为 $t+t_c$。

对于字符设备其输入过程如下：

字符设备输入时，缓冲区用于暂存用户输入的一行数据。在输入期间，用户进程被挂起以等待一行数据输入完毕；在输出时，用户进程将一行数据送入缓冲区后，继续执行计算。当用户进程已有第二行数据输出时，若第一行数据尚未提取完毕，则用户进程应阻塞。

对于单缓冲，缓冲区属于临界资源，即不允许多个进程同时对一个缓冲区进行操作。因此，单缓冲虽然能匹配设备和 CPU 的处理速度，但无法实现设备与设备之间的并行操作。

2. 双缓冲

如图 7.6(b)所示，双缓冲能够提供两个缓冲区。在设备输入时，先将数据输入第一缓冲区，装满后便转向第二缓冲区。此时操作系统可从第一缓冲区中移出数据，接着由 CPU 对数据进行计算。在双缓冲时，系统处理一块数据的时间可粗略地认为是 $\max(t_c,t)$。如果 $t_c<t$，则可使块设备连续输入；如果 $t_c>t$，则可使 CPU 不必等待设备输入。对于字符设备，若采用行输入方式，则采用双缓冲工作方式时，通常用户进程不会被阻塞，从而消除了用户的等待时间，即用户在输完第一行后，在 CPU 执行第一行中的命令时，用户可继续向第二缓冲区输入下一行数据。

为了实现两台外设、打印机和终端之间的并行操作，可以设置双缓冲区。有了两个缓冲区后，CPU 就可以把输出到打印机的数据放入其中一个缓冲区，让打印机慢慢打印；再从另一个为终端设置的缓冲区中读取所需要的输入数据。

很明显，双缓冲只是一种说明设备与设备、CPU 与设备并行操作的简单模型，并不能用于实际系统中的并行操作。这是因为计算机中的外围设备较多，双缓冲区也很难匹配设备和处理机的处理速度。

3. 循环缓冲

由于双缓冲并不能真正解决实际系统中的并行操作，于是引入了多缓冲。通过增加缓冲区的个数，可使并行程度得到明显提高。

多缓冲是把多个缓冲区连接起来组成两部分：一部分专门用于输入；另一部分专门用于输出。一般将多缓冲区组织成循环缓冲形式。对于用作输入的循环缓冲，通常提供给输入进程和计算进程使用，输入进程不断向空缓冲区输入数据，计算进程则从中提取数据用于计算。

循环缓冲包括以下两部分。

(1) 多个缓冲区

循环缓冲含有多个缓冲区，每个缓冲区的大小相同。缓冲区可分成 3 种类型：空缓冲区 R，用于存放输入数据；已装满数据的缓冲区 G，其中的数据提供给计算进程使用；现行工作缓冲区 C，这是计算进程正在使用的缓冲区。循环缓冲的组成如图 7.7 所示。

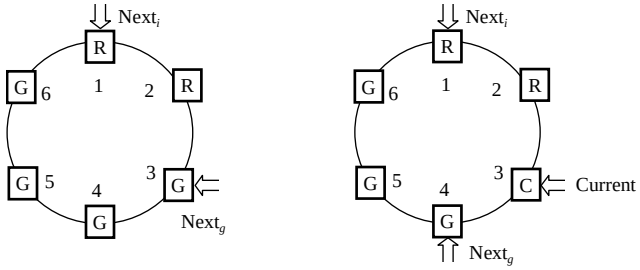


图 7.7　循环缓冲

(2) 多个指针

对用于存放输入数据的多缓冲区，应设置这样 3 个指针：Next$_g$，指示计算进程下一个可用的缓冲区 G；Next$_i$，指示输入进程下次可用的空缓冲区 R；Current，指示计算进程正在使用的缓冲区单元，开始时，它指向第一个单元，随计算进程的使用，它将逐次地指向第 2 个单元，第 3 个单元…… 直至缓冲区的最后一个含有数据的单元。

4. 缓冲池

上述的循环缓冲区仅适用于某特定的 I/O 进程和计算进程，因而属于专用缓冲。当系统较大时，将会有许多这样的循环缓冲，这不仅要消耗大量的内存空间，而且其利用率也不高，为了提高缓冲区的利用率，目前广泛流行公用缓冲池，池中的缓冲区可供多个进程共享。

(1) 缓冲池的结构

缓冲池由多个缓冲区组成，一个缓冲区由两部分组成：一部分是用来标识该缓冲区和用于管理的缓冲首部，另一部分是用于存放数据的缓冲体，这两部分有一一对应的映射关系。缓冲首部由设备号、数据块号、缓冲区号、互斥标识位、连接指针组成。

对于既可用于输入又可用于输出的公用缓冲池，至少应含有以下 3 种类型的缓冲区：空(闲)缓冲区；装满输入数据的缓冲区；装满输出数据的缓冲区。为了管理上的方便，可将相同类型的缓冲区链成一个队列，于是可形成以下 3 个队列：

① 空缓冲队列 emq。这是由空缓冲区所链成的队列。其队首指针 F(emq)和队尾指针 L(emq)分别指向该队列的首缓冲区和尾缓冲区。

② 输入队列 inq。这是由装满输入数据的缓冲区所链成的队列，其队首指针 F(inq)和队尾指针 L(inq)分别指向该队列的首、尾缓冲区。

③ 输出队列 outg。这是由装满输出数据的缓冲区所链成的队列，其队首指针 F(outq)和队尾指针 L(outq)分别指向该队列的首、尾缓冲区。

除了上述 3 种队列外，系统或用户进程从这 3 种队列中申请和取出缓冲区，并用得到的缓冲区存数或取数，存、取数操作完毕后再将缓冲区放入相应的队列，这些缓冲区称工作缓冲区。在缓冲池中具有 4 种工作缓冲区：用于收容输入数据的工作缓冲区 hin；用于提取输入数据的工作缓冲区 sin；用于收容输出数据的工作缓冲区 hout；用于提取输出数据的工作缓冲区 sout。

缓冲池的工作缓冲区，如图 7.8 所示。

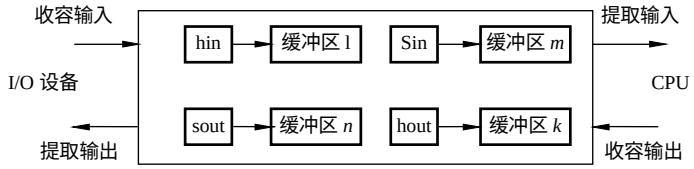


图 7.8　缓冲池中的工作缓冲区

(2) 缓冲池的管理

一般，管理缓冲池的步骤如下：

① take-buf(type)：从 3 种缓冲区队列中按一定的规则取出一个缓冲区。

② add-buf(type, number)：把缓冲区按一定的规则插入相应的缓冲队列。

③ get-buf(type, number)：申请缓冲区。

④ put-buf(type, work-buf)：将缓冲区放入相应缓冲区队列。

其中，type 为缓冲队列类型, number 为缓冲区号，work-buf 为工作缓冲区类型。

缓冲池的工作过程如下：

首先，输入进程调用 get-buf (em, number)过程从空白缓冲区队列中取出一个缓冲区 number，将该空白缓冲区作为输入缓冲区 hin。当 hin 装满了输入数据之后，系统调用 put-buf (in, hin)将该缓冲区插入输入缓冲区队列 in 中。

接着，当进程需要输出数据时，输出进程经过缓冲管理程序调用 get-buf(em, number)，从空白缓冲区队列可取出一个空白缓冲区 number 作为收容输出缓冲区 hout，当 hout 中装满了输出数据之后，系统再调用 put-buf(out, hout)过程将该缓冲区插入输出缓冲区队列 out。

最后，用 get-buf 和 put-buf 过程提取缓冲区的输入和输出数据。get-buf(out, number)从输出缓冲队列中取出装满数据的缓冲区 number，将其作为 sout。当 sout 中数据输出完毕时，系统调用 put-buf(em, sout)将该缓冲区插入空白缓冲队列。put-buf(in, number)从输入缓冲队列中取出装满数据的缓冲区 number 作 sin，当 CPU 从中取完数据后，系统调用 put-buf(em, sin)将该缓冲区(已变空)释放并插入到空白缓冲队列 em 中。

下面描述过程 get-buf 和 put-buf。

假设互斥信号量 MS(type)，初值为 1；描述资源数目的信号量 RS(type)，初值为 n(n 为 type 到长度)，则既可实现互斥，又可实现同步的 get-buf 和 put-buf 两过程描述如下：

```
Procedure Get-buf(type)
begin
    Wait(RS(type));
    Wait(MS(type));
    B(number):=Take-buf(type);
    Signal(MS(type));
end
Procedure Put-buf(type, number)
begin
    Wait(MS(type));
    Add-buf(type, number)
    Signal(MS(type))
    Signal(RS(type));
  End
```

7.3　磁盘调度

近 30 年来，处理器和主存速度的增加远远把磁盘抛在了后面。因为磁盘的相对低速，使得磁盘子系统的性能变得至关重要，许多人也在积极探索提高磁盘子系统性能的方法，改进调度策略，降低查找时间。

7.3.1　调度策略

磁盘调度策略有很多，常见的有随机调度、先进先出、进程优先级、后进先出等。

如表 7.1 所示，磁盘调度策略通常用吞吐量、平均响应时间、响应时间的可预期性(或变化幅度)来决策。

表 7.1　磁盘调度策略

	名　称	描　述	注　释
根据请求对象选择	RSS	随机调度	用于分析
	FIFO	先进先服务	最简单
	PRI	优先级	控制磁盘队列管理的外部
	LIFO	后进先服务	最大的位置和资源利用
根据被请求项目选择	SSTF	最短服务时间优先	高利用、小队列
	SCAN	扫描	好一些的服务分配
	C-SCAN	循环扫描	低一些的服务变化
	N-Step-SCAN	同一时间扫描 N 个记录	服务保证
	FSCAN	SCAN 循环的开始 N=队列的 N-Step-SCAN	装载敏感

下面介绍常用的几种磁盘调度策略。

1. 先进先服务策略

顾名思义，它是将各进程对磁盘请求的等待队列按提出请求的时间进行排序，并按此次序给予服务的一种策略。这个策略对各进程是公平的，它不管进程优先级多高，只要是新来到的访问请求，都被排在队尾。例如，假定磁盘有 200 个磁道，磁盘请求队列是随机的。被请求的磁道依次为 55，58，39，18，90，160，150，38，184，开始时读写头位于 100 号磁道。这样磁头总共要移动 628 个磁道，这个调度可用图 7.9 表示。

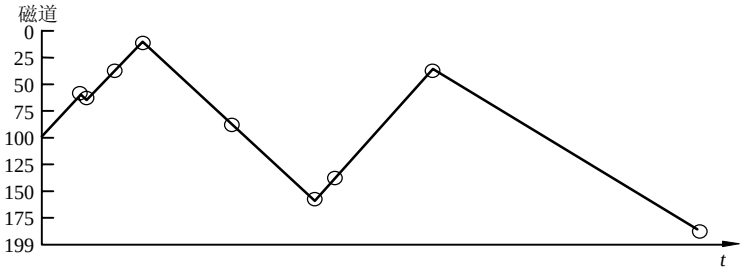


图 7.9　先来先服务(FCFS)的磁盘调度

当用户提出的访问请求比较均匀地遍布整个盘面，而不具有某种集中倾向时(通常是这样的)，FCFS 策略导致了随机访问模式，即无法对访问进行优化。在对盘的访问请求比较多的情况下，此策略将降低设备服务的吞吐量和提高响应时间，但各进程得到服务的响应时间的变化幅度较小。

FCFS 适用于访问请求不是很多的情况，其算法最简单。

2．最短服务时间优先策略

这是选择请求队列中柱面号最接近于磁头当前所在的柱面的访问要求，作为一个服务对象的一种策略。

如果对上述请求队列，使用 SSTF 策略时，最接近当前磁头所在位置 100 的请求是 90 号磁道，尔后是 58,39,38,18,150,184 号磁道。该策略可以得到比较好的吞吐量(与 FCFS 相比)和较低的平均响应时间。其缺点是对用户的服务请求的响应机会不是均等的，对中间磁道的访问请求将得到最好的服务，对内、外两侧磁道的服务随偏离中心磁道的距离增加而越来越差，因而，导致响应时间的变化幅度很大，在服务请求很多的情况下，对内、外边缘磁道的请求将会无限地被延迟，因而有些请求的响应时间将不可预期。图 7.10 给出了最短服务时间优先的磁盘调度情况。

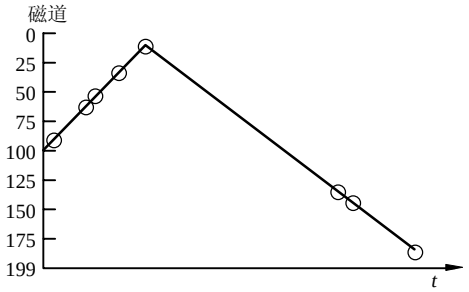


图 7.10　最短服务时间优先的磁盘调度

3．扫描策略

扫描策略（SCAN）也叫电梯策略(Elevator Algorithm)。大多数电梯保持按一个方向移动，直到没有请求为止，然后改变方向。它需要软件维护一个二进制位——当前方向位：向上或向下。当一个请求处理完毕后，软件检查该位，如果向上，则电梯舱移到下一个更高的等待请求。如果更高的位置没有请求，就掉转方向。如果方向改为向下，同时存在一个低位置请求，则移向该位置。

SCAN 策略的磁盘调度策略也是如此。

SCAN 策略是选择请求队列中，把磁臂前进方向最接近于磁头当前所在柱面的访问要求作为下一个服务对象。也就是说，如果磁臂目前向内移动，那么下一个服务对象，应该是在磁头当前位置以内的柱面上的诸访问请求中之最近者，……这样依次地进行服务，直到没有更内侧的服务请求，磁臂才改变移动方向，转而向外移动，并依次服务于此方向上的访问请求，如此由内向外，由外向内，反复地扫描访问请求，依次给以服务。如果仍以上面请求序列为例，在使用 SCAN 之前，不仅要知道磁头移动的当前位置，而且还要知道磁头移动方向，如果此时磁头向 200 号磁道移动，则先服务 150、160、184 号磁道上的请求，再移到 200 道，然后磁头反向向内移动，服务 90、58、55、39、38、18 号磁道上的请求，图 7.11 表示 SCAN 策略用于磁盘调度情况，此策略基本上克服了 SSTF 策略的服务集中于中间磁道和响应时间变化比较大的缺点。而具有 SSTF 策略的优点，即吞吐量比较大，平均响应时间较小，但是，由于是摆动式的扫描方法，两侧磁道被访问的频率仍然低于中间磁道，只是不像上述 SSTF 策略那样严重而已。

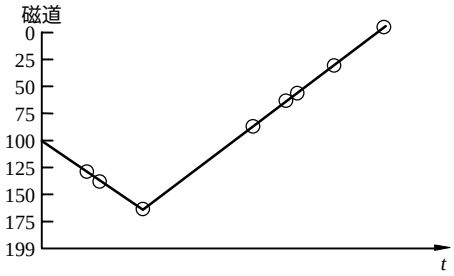


图 7.11　扫描策略用于磁盘调度

4. 循环扫描策略

对 SCAN 算法稍作改进可以进一步减少响应时间。方法是：总是按同一方向移动磁臂，处理完最高编号柱面上的请求后，磁臂移动到具有读/写请求的最低编号的柱面，然后继续向上移动。

循环扫描策略与扫描策略的不同之处在于循环扫描是单向反复地扫描。当磁臂向内移动时，它对本次移动开始前到达的各访问要求，自外向内地依次给以服务，直到对最内柱面上的访问要求满足后，磁臂直接向外移动，使磁头停在所有新的访问要求的最外边的柱面上。然后再对本次移动前到达的各访问要求依次给以服务。

如果仍以前面的访问请求序列为例，那么从当前磁头位置 100 号磁道出发，其以后的服务次序为 150、160、184、18、38、39、55、58、90 号磁道。图 7.12 表示了循环扫描策略用于磁盘调度的情况。

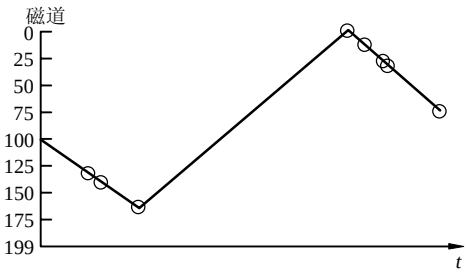


图 7.12　循环扫描策略用于磁盘调度

磁盘调度策略很多，各有利弊。如何选择相应调度策略与磁盘的使用环境因素有关。表 7.2 比较了上述四种策略的性能。

表 7.2　磁盘调度策略的性能比较

(a)FIFO 从磁道 100 开始		(b)SSTF 从磁道 100 开始		(c)SCAN 从磁道 100 开始 按增加磁道数方向		(d)C-SCAN 从磁道 100 开始 按增加磁道数方向	
下次访问 磁道	访问的磁 道数	下次访问 磁道	访问的磁 道数	下次访问 磁道	访问的磁 道数	下次访问 磁道	访问的磁 道数
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
平均 寻道数	55.3	平均 寻道数	27.5	平均 寻道数	27.8	平均 寻道数	35.8

磁盘调度的另一个趋势是使多个磁盘一起工作，特别对高端系统很有用。一个令人感兴趣的配置是让 38 个驱动器并行工作。当并行读操作时，一次有 38 位灌入计算机，每个驱动器一位，它们形成一个 32 位的字及 6 位校验位。将 1、2、4、8、16 和 32 位作为等价位，该 38 位字可被编码为海明码。利用该方法，如果某个驱动器停止工作，则每个字丢失一位，由于海明码可以恢复丢失的位，所以系统可以继续运行，这种设计称为廉价冗余磁盘陈列(Redundant Array of Inexpensive Disks，简称 RAID)。RAID 的特点是：

① RAID 是一个物理磁盘集合，但被 OS 认为是一个逻辑盘。

② 数据分布存放在不同磁盘上。

③ 具有较强的纠错能力。

RAID 分 6 个级别：level0~level5。

7.3.2　磁盘高速缓存

Cache 存储器是在主存和处理器之间插入的一个更快、更小但更昂贵的存储器，其作用是减少主存的平均存取时间。同样的原理也可应用到磁盘存储器。一个磁盘 Cache 是一个在主存中为磁盘扇区开辟的一个缓冲区。这个 Cache 包含有磁盘的一部分扇区的拷贝。当输入/输出请求操作某一特定扇区时，就会首先检查磁盘 Cache 是否有该扇区的拷贝。如果有，这个 I/O 请求就直接对该 Cache 操作；否则，被请求的扇区就首先被写进 Cache 中(从磁盘)，然后再针对 Cache 操作。

关于磁盘 Cache，有几个设计问题值得说明。

(1) Cache 中的数据传送

当一个 I/O 请求能由 Cache 提供服务时，Cache 中的数据必须传送到请求进程中。有两种方法传送数据：一是通过共享存储器将数据所在的指针传送到请求进程；二是传送 Cache 中的数据到用户进程的存储空间中。

很显然，前一种方法能节省存储器到存储器的时间，也允许其他进程共享读/写该 Cache 数据。

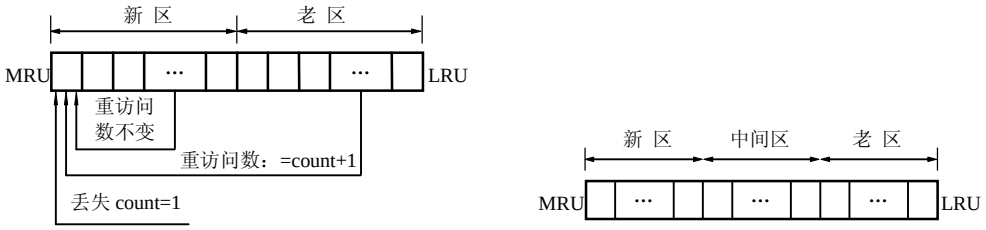
(2) Cache 中数据的替换策略

当一个新的扇区装入 Cache 中时，一个原先在 Cache 中的扇区数据必须替换出去。经常使用的算法是最近最少使用算法(Least Rencently Used, LRU)。Cache 中没有被使用时间最长的那个块将被替换。可以说，Cache 中有一个块堆栈。最近访问最多的块在栈顶。当 Cache 中的一个块被使用时，此块将从栈中当前位置移到栈顶。当一个新块从辅存中读出时，位于栈底的块将会移出，新块将置于栈顶。当然，事实上并不是将这些块在主存中移动；而是用一个栈指针和 Cache 相联系。

替换 Cache 中最少被访问的块的算法是最少使用算法(Least Frequently Used, LFU)。LFU 给每块加上一个计数器，当读出一个新块时，计数器赋值 1；访问块一次，其计数器增加 1。当需要替换时，就替换计数器数值最小的块。LFU 比 LRU 更正确，因为这个选择过程利用了更多的关于块的信息。

一个简单的 LFU 算法有如下问题。对于有些块而言，整个访问次数相对不多，但在一段时间内会因为局部性重复访问，从而产生很高的访问次数，但实际上，该计数值并未反映出此块将来被访问的情况。在这段时间结束之后，由于该计数值高居不下，将会使 LFU 的替换选择变得很差。

为了克服 LFU 的这个困难，提出一个新的技术——基于频率的替换(Frequency-Based Replacement)。先考虑一个简单的例子(见图 7.13(a))。块组织成堆栈形式，栈顶部的一部分作为“新区”保留。当发生一个读 Cache 时，被访问的块移到栈顶。



(a)　　(b)

图 7.13　基于频率的替换

(a) FIFO　(b) 使用 3 个区

如果此块已在“新区”中，它的计数并不增加，否则增加 1。在一个足够大的“新区”中，这种方法保证刚才提到的那种块的计数值维持不变。选择那些不在“新区”中且计数值最小的块进行替换；当相等情况出现时，替换最近最少使用的块。

这种方法只在 LRU 上取得很小的改进。仍有问题如下：

① 当发生一个 Cache 错误时，计数值为 1 的新块会移入“新区”中。

② 只要块在“新区”中，计数值保持为 1。

③ 当块从“新区”移出时，它的计数值依然是 1。

④ 如果某块现在不是相当迅速地被重复访问，那么就很有可能因为它是“新区”中计数值最小的块，而被替换掉。换句话说，即使它们访问的频率相当高，也不会有足够的时间让它们在移出“新区”之前有很高的计数值。

进一步地说，就是把栈分成 3 个部分：新、中间、老。在“新区”中，计数值并不增加，只有“老区”中的块才能被替换。假定有一个足够大的中间部分，这样给了经常被访问块的一个机会，能够在可能被替换之前使计数值足够大。模拟研究表明这种方法比简单的 LRU 或 LFU 要好得多。

不管什么替换策略，替换可以按需要产生或是可以预先计划好。前者只要在需要一个槽时才替换一个扇区(槽(slot)是指一个缓冲区，每个槽包含一个磁盘扇区)。在后面的情况中，一次可以释放多个槽。原因在于需要写回扇区。如果一个扇区读到 Cache 中，而且只读，那么当它被替换时，没有必要把它写回磁盘。然而，如果扇区已被更新，就有必要在替换它时将它写回。同时需要使写回的寻道时间最小。

(3) Cache 的失效率

一般而言，在同样的替换策略下，Cache 越大，失效率越少。因此，必须设计好 Cache 的大小和替换策略，减少失效率。

7.4　系统举例

7.4.1　UNIX System V

在 UNIX 中，每个 I/O 设备都和一个特殊的文件相联系，由文件系统管理，对其的读/写操作和对用户数据文件的操作一样。这种方法为用户和进程提供了一个统一的接口。为了从设备上进行读或写，将对与设备相关的特殊文件进行读/写请求。

图 7.14 说明了 I/O 设备的逻辑结构。文件子系统对存放在辅存上的文件进行管理。同时，因为设备被看成文件，它也作为进程的接口。

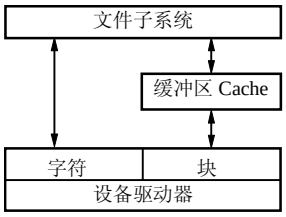


图 7.14　UNIX I/O 结构

在 UNIX 中有两种类型的 I/O：缓冲 I/O 和无缓冲

I/O。缓冲 I/O 使用系统缓冲，而无缓冲 I/O 使用 DMA。在后者中，传输直接在 I/O 模块和进程 I/O 区中发生。对于缓冲 I/O，使用了两类缓冲：系统缓冲 Cache 和字符队列。

1. 缓冲高速缓存(Buffer Cache)

UNIX 中缓冲 Cache 实际上是一个磁盘 Cache。对于磁盘的 I/O 操作通过缓冲 Cache 处理。缓冲 Cache 和用户进程空间之间的数据传输经常通过使用 DMA 发生。因为缓冲 Cache 和进程 I/O 都在主存中，DMA 机制利用这来执行内存之间的拷贝。这不会用完任何处理器周期，但会消费总线周期。

为了对缓冲 Cache 进行管理，系统使用了 3 种表列：

① 自由表列。Cache 中用于分配的所有槽的表列。

② 设备表列。和每个磁盘相联系的所有缓冲的表列。

③ 驱动器 I/O 表列。在一个特定设备上执行或等待 I/O 的缓冲区的表列。

所有缓冲区应该在自由表列或驱动器 I/O 表列中。一旦一个缓冲区和一个设备相连接，即使它在自由表列中，也将和这个设备相连直到它被重用，从而和另一个设备相连接。这些表列作为指针和每个缓冲建立联系，而不是物理上分离的表列。

当访问某个设备上的一个物理块号时，操作系统首先检查此块是否在缓冲 Cache 中。为了减少寻找时间，设备表列被组织成快表(Hash Table)形式，图 7.15 描述了缓冲 Cache 的一般结构。一个固定长度的快表中存放着指向缓冲 Cache 的指针。每一个对于(设备号、块号)的访问都被映射成快表中的一个特定入口。入口中的指针指向链中的第一个缓冲区。每个缓冲区有一个指针(Hash Pointer)指向链中的下一个缓冲区。因此，对于所有(设备号、块号)访问都映射到同一快表入口，一旦要找的块在缓冲 Cache 中，则缓冲区在快表入口的链表中。这样，对于缓冲 Cache 的搜索长

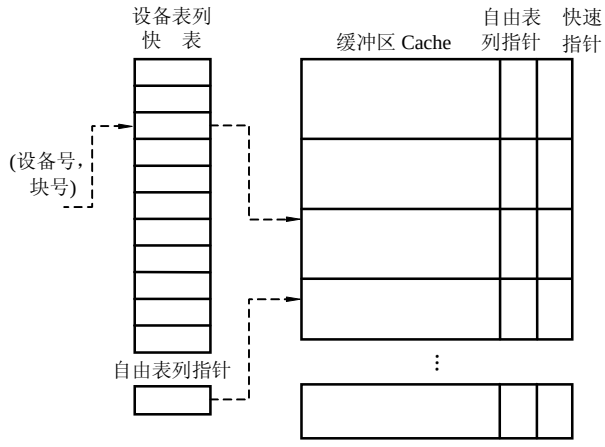


图 7.15　UNIX 缓冲区 Cache 结构

度就减小至 N 的因数，N 是快表的长度。

块的替换中使用的是 LRU：在一个缓冲区分配给一个磁盘块后，除非其他所有缓冲区最近都被访问过，此缓冲区不会被分配给另一个块。

2. 字符队列(Character Queue)

面向块的设备，如磁盘和磁带，可以被缓冲 Cache 支持。另一种缓冲则适用于面向字符的设备，如终端、打印机等。字符队列可以由 I/O 设备写、处理器读或者由处理器来写、I/O 设备读。这两种情况都使用了前面介绍的生产者/消费者模型。因此，字符队列只能读一次；一旦一个字符读过了，它就消失了。这和缓冲 Cache 相关，缓冲 Cache 可以读很多次，使用的是前面的读者/写者模型。

3. 无缓冲 I/O(Unbuffered I/O)

无缓冲 I/O 是进程进行 I/O 的最快方法，它在设备和进程空间之间使用 DMA，进行无缓冲 I/O 的进程在主存中被锁起来，不能被换出。通过锁死高端内存，减少了交换的机会，但也降低了整个系统的性能。同时，I/O 设备也固定于一个进程，在传输中，不得为其他进程所用。

4. UNIX 设备

UNIX 识别下面 5 种设备：磁盘驱动器、磁带驱动器、终端、通信线、打印机。

表 7.3 说明了何种 I/O 适用何种设备。UNIX 中经常使用的磁盘驱动器是块设备，可能有较高的吞吐率。因此，I/O 可以是无缓冲或是缓冲 Cache。磁带驱动器相似于磁盘驱动器，二者均使用相同的 I/O 方法。

表 7.3　UNIX 中的设备 I/O

设　　备	无缓冲 I/O	缓冲 Cache	字符队列
磁盘驱动器	√	√	
磁带驱动器	√	√	
终端			√
通信线			√
打印机	√		√

由于终端包含相对缓慢的字符交换，终端 I/O 使用字符队列。通信线要求数据字节的串行处理，因此，可由字符队列处理。字符队列能否用于打印机取决于打印机的速度。慢速打印机可以使用字符队列，而快速打印机使用无缓冲 I/O 或缓冲 Cache。然而，因为打印机中的数据不会被重用，缓冲 Cache 的头(Overhead)是不必要的。

7.4.2　Windows NT I/O 分析

Microsoft Windows NT I/O 系统是 Windows NT 执行体的组件，并且存在于

ntoskrnl.exe 文件中。它接受 I/O 请求(来自用户态和核心态的调用程序)，并且以不同的形式把它们传送到 I/O 设备。在用户态 I/O 函数和实际的 I/O 硬件之间有几个分立的系统组件，包括文件系统驱动程序、过滤器驱动程序和低层设备驱动程序。

Windows NT I/O 系统的设计目标如下：

① 加快单处理器或多处理器系统的 I/O 处理。

② 使用标准的 Windows NT 安全机制保护共享的资源。

③ 满足 Microsoft Win32、OS/2 和 POSIX 子系统指定的 I/O 服务的需要。

④ 提供服务，使设备驱动程序的开发尽可能地简单，并且允许许用高级语言编写驱动程序。

⑤ 允许在系统中动态地添加或删除设备驱动程序。

⑥ 为包括 FAT、CD-ROM 文件系统(CDFS)和 Windows NT 文件系统(NTFS)的多种可安装的文件系统提供支持。

⑦ 为映像活动、文件高速缓存和应用程序提供映射文件 I/O 的能力。

1. I/O 系统结构和管理器

图 7.16 描述了 Windows NT 的 I/O 系统结构。

在 Windows NT 中，程序在虚拟文件中执行 I/O。虚拟文件是指用于 I/O 的所有源或目标，它们都被当作文件来处理(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被看做是直接到这些虚拟文件的简单的字节流。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，这些函数再依次地调用内部 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。

Windows NT 的 I/O 管理器如图 7.17 所示。

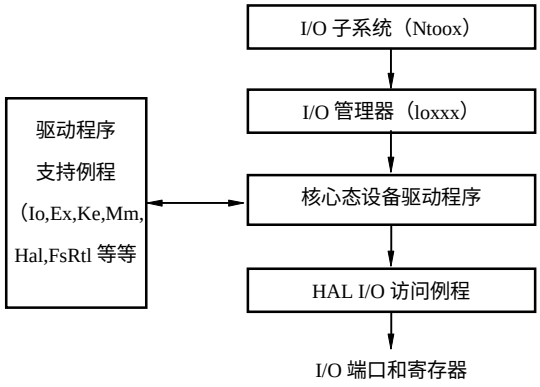


图 7.16　I/O 系统结构

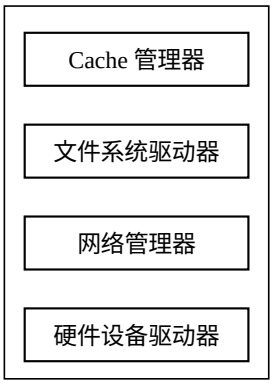


图 7.17　Windows NT I/O 管理器

I/O 管理器定义了有序的工作框架(或模型)。在该框架里，I/O 请求被提交给设

备驱动程序。I/O 系统是包驱动的。大多数 I/O 请求用 I/O 请求包 IRP 表示，它从一个 I/O 系统组件移动到另一个 I/O 系统组件。IRP 是在每个阶段控制如何处理 I/O 操作的数据结构。

如图 7.17 所示，I/O 管理器由以下 4 部分组成。

① Cache 管理器。它管理整个 I/O 子系统的 Cache，它为所有的文件系统和网络部件在主存中提供 Cache 服务，并且根据可用的物理存储器的变化动态地增大或减小 Cache。Cache 管理器提供了两个增强性能的服务：

•延迟写入。系统记录变化在 Cache 中而不是磁盘中，当处理器较空闲时，Cache 管理器写入变化到磁盘中。

• 延迟提交。系统 Cache 被提出的事务信息，待以后作为一个后台进程写入到文件系统日志中。

② 文件系统驱动器。I/O 管理器对待一个文件系统就像对待一个设备一样，并路由一定的信息给设备适配器对应的驱动程序。

③ 网络驱动器。Windows NT 内置了用于分布式应用的网络能力和支持。

④ 硬件设备驱动器。该驱动器通过动态链接库(DLL)的入口去存取硬件设备的注册信息。

2. I/O 函数

除了通常的打开、关闭、读/写函数外，Windows NT I/O 系统还提供了一些高级的特性，例如，异步 I/O 和快速 I/O 等。

(1) 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，程序可以立即访问这个被传输的数据。它们的最简单的形式——Win32 Readfile 和 Writenfile 函数，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 IO 允许应用程序发布 I/O 请求，然后，在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 CreateFile 函数中指定 FILE_FLAG_OVERLAPPEDS 标志。当然，在发出异步 I/O 操作之后，线程不访问任何来自 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 请求的完成同步。当 I/O 完成时，这些同步对象将会变成有信号态。

与 I/O 请求的类型无关，由 IRP 代表的内部 I/O 操作都被异步执行；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是为异步 I/O 打开的。

(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，是由 I/O 系统和内存管理器共同产生的，它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，则内存管理器就把更改信息作为正常页面调度的一部分写回到文件中。

(4) 分散/集中 I/O

Windows NT 同样支持一种特殊种类的高性能 I/O，它被称作分散/集中(Scatter/Gather)。它可通过 Win32 ReadFileScatter 和 WriteFileScatter 函数来实现。这些函数允许应用程序，从在虚拟内存中的一个以上的缓冲区，发布单一的读取或写入到磁盘上文件的一个连续区域的命令。若要使用分散/集中 I/O，则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 必须异步执行(重叠)。

3. 设备驱动程序

Windows NT 支持几种类型的设备驱动程序，图 7.18 列出了 Win32 I/O 函数和

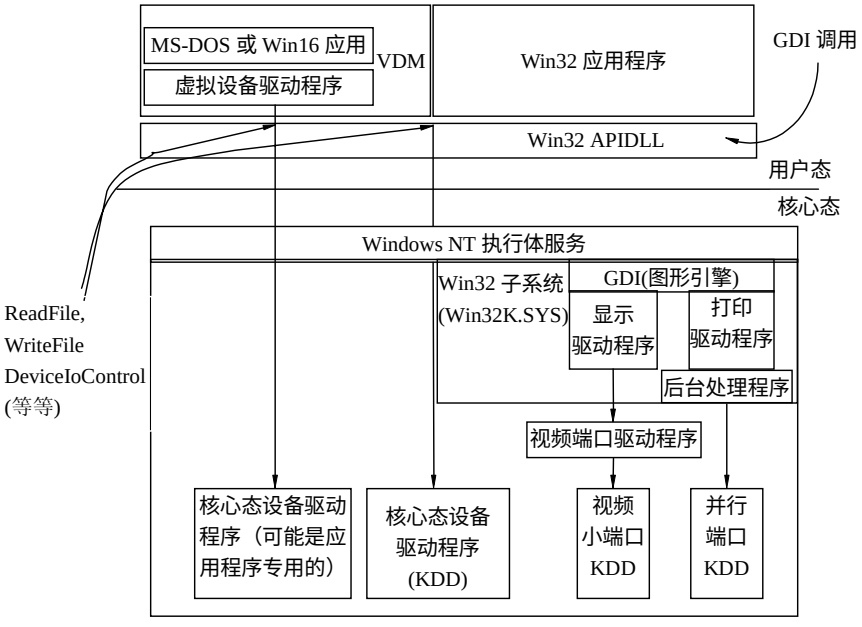


图 7.18　设备驱动程序的类型

3 个主要种类的设备驱动程序之间的关系。

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 NT 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态设备驱动程序来访问硬件。

Win32 子系统显示驱动程序和打印驱动程序将与设备无关的图形(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它关联的显示驱动程序提供硬件级的支持。

核心态设备驱动程序是能够直接控制和访问硬件设备的惟一驱动程序类型。图 7.19 描述了核心态设备驱动程序之间的关系。

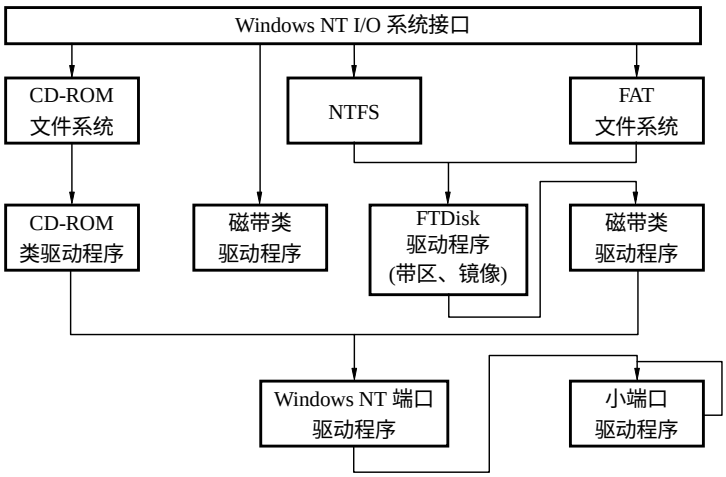


图 7.19　驱动程序结构

7.5　小　　结

I/O 接口是计算机系统与外部世界的接口。由于现代操作系统设备的多样性和复杂性，使得设备管理成为操作系统中最复杂、最具多样性的部分。为了保证易用性，设备管理模块应为用户提供一个透明的、易于扩展的接口，使得用户不必了解具体设备的物理特性，以便于设备的追加和更新。另外，设备管理模块还应该设计成尽量提高设备与设备之间、设备与 CPU 之间的并行操作度以及设备利用率，使得整个系统获得最佳效率。

围绕着上述目的，本章分两个大类讨论了 I/O 系统原理：一是 I/O 硬件组成原

理；二是 I/O 软件组成原理。

I/O 硬件的基本概念如下：

① 设备一般分为字符设备和块设备。字符设备传输的信息为字节流或字符；而块设备则为数据块。

② I/O 设备一般由机械部分和电子部分组成。通常将这两部分开处理，机械部分是设备本身，电子部分为设备驱动器或适配器。

③ I/O 技术常见的有 4 种，它们是程序直接控制方式、中断控制方式、DMA 方式和通道方式。程序直接控制方式和中断控制方式只适用于简单的、外设很少的计算机系统。DMA 方式和通道方式较好地解决了前两种方式占用 CPU 时间多的缺点。后两种方式采用外设与内存直接交换数据的方式，只是到数据传输结束时，才发中断信号给 CPU，大大减轻了 CPU 负担。

I/O 软件组成如下：

① 中断处理程序。当发生中断时，系统分析中断原因，并转入相应的中断处理程序。

② 设备驱动程序。所有与设备相关的代码放在设备驱动程序中。

③ 与外设无关的 I/O 软件。通常称之为 I/O 子系统，它执行与设备无关的操作。另外还重点讨论了缓冲技术。

④ 在用户空间运行的 I/O 库程序和 spooling 技术。

另外本章也较详细地讨论了磁盘调度策略。最后，以两个典型的系统 UNIX System V 和 Windows NT 为例简单介绍了这两个系统如何管理 I/O 设备的思想。

习　题　七

7.1　设备按传输的数据单元分成几类？各有何特点？举例加以说明。

7.2　试画出微型机和主机中常采用的I/O系统组织结构图。

7.3　设备控制器有什么作用？试画出设备控制器的组成图。

7.4　试说明I/O技术的发展过程和主要推动原因。

7.5　简述DMA控制方式和通道控制方式的工作流程。

7.6　I/O软件设计的目标是什么？以下各项工作是在4个I/O软件层的哪一层完成的？

(1) 为一个磁盘读操作计算磁盘扇区、寻道和定位磁头的时间；

(2) 维护一个最近使用的块的高速缓存；

(3) 向设备寄存器写命令；

(4) 检查用户是否有权使用设备；

(5) 将二进制整数转换成ASCII码以便打印。

7.7　引入缓冲的主要原因是什么？比较单缓冲、双缓冲、循环缓冲以及缓冲池的结构、管理

方法以及优缺点。

7.8　一个局域网以如下方式使用：用户发了一个系统调用，请求将数据包写到网上；然后，操作系统将数据拷贝到一个内核缓冲区中，再将数据拷贝到网络控制器板上。当所有数据都安全存入到控制器中时，再将它们从网上以10Mb/s的速率传送，在每一位被发送后，将它们以1b/ms的速率保存到网络控制器。当最后一位到达时，中断目标CPU，内核将新到达的数据包拷贝到内核缓冲区中进行检查。一旦指明该数据包是发送给哪个用户进程的，内核就将数据拷贝到该用户进程空间。如果假设每一个中断及其相关处理过程花费l ms，数据包有1024B(忽略头标)，拷贝1B花费1 ms，将数据从一个进程转储到另一个进程的最大速率是多少？

7.9　什么叫设备无关性？如何实现设备独立性？

7.10　试说明spooling系统的概念及优点。

7.11　为什么打印机的输出文件在打印前通常都假脱机输出到磁盘上？

7.12　磁盘上请求以10、22、20、2、40、6、38柱面的次序到达磁盘驱动器。寻道时每个柱面移动需要6 ms，问：以下各算法的寻道时间是多少？并画出寻道顺序图。

(1) FCFS

(2) SSTF

(3) SCAN

(4) C-SCAN

7.13　试说明设备驱动程序具有哪些特点？具有哪些功能？通常应完成哪些工作？

7.14　什么是磁盘缓冲？实现磁盘缓冲有什么好处？

7.15　简述Windows NT I/O管理器结构。

文件管理

文件管理是操作系统的基本功能之一，在操作系统中实现这一基本功能的程序系统(部分)称为文件系统，它主要是进行信息的组织、管理、存取和保护。本章将讨论文件的组织方式、存取的机制、可执行文件的结构，以及文件存储空间的管理等问题。

8.1 文件与文件系统

8.1.1 文件及其分类

1. 文件

文件的概念是在信息的物理存储及其信息表示方式需要的基础上引入的。在讨论文件时经常使用以下几个相关术语：域(Field)、记录(Record)、文件(File)以及数据库(Database)。

域是数据的基本元素。每个域有个值，如雇员的姓、日期等。各域由各自的长度和数据类型分类。由用户和程序给域赋值。域由设计决定是固定长度或变长的。在变长的域中，域有 2 或 3 个子域：存放的实际值、姓名和域的长度。在有些变长的域中，域的长度由域间的特殊分界符号表示。大多数文件系统不支持可变长的域。

记录是相关域的集合，可以看成是将一个单元供应用程序使用。例如，一个雇员记录有姓名、社会保险号、工作分类和雇佣日期等。同样的，依赖于设计，记录可以是定长或变长的。当记录的一些域是变长的，或域的数目是可变的，记录就是可变长度的。每个域都有个域名。不论什么情况下，记录总有长度域。

文件是相关记录的集合。用户和应用程序把文件当成单个实体，由文件名来访问文件。文件有惟一的文件名，可以生成或是删除。访问控制常常在文件层上进行，

就是说，在共享系统中，给予或不给予用户程序以访问文件的权限。在一些更复杂系统中，访问控制在记录或是域的层次上。

数据库是相关数据的集合。数据库中数据元素间的关系是明显的。数据库可供若干不同的应用程序来使用。一个数据库可以包括与一个组织或项目相关的所有信息，如商业或科学研究。数据库由一种或多种文件组成。通常会有一个单独的数据库管理系统，虽然此系统使用了一些文件管理程序。

为了便于对文件的管理，将文件自身分为文件说明和文件体两个部分，所有的文件都具有 3 个基本特征：

① 文件体的内容丰富，可以是源程序、可执行代码、数据、表格、语言或图像等。

② 无论何种内容的文件都遵循按名存取的规则，用户无需了解存取内容在存储介质上的物理位置。

③ 文件具有可重用性和可保存性。

2. 文件的分类

文件一般按其用途和属性来归类。

按用途把文件划分为用户文件，系统文件和库文件 3 种：

① 用户文件，由用户建立，并由文件拥有者进行读/写和执行。

② 库文件，由系统为用户提供的实用程序、标准子程序、动态重链接库等。如 ACLEDIT.DLL 是 Windows NT 中提供的访问控制清单编辑库文件。用户可调用，但无权修改。

③ 系统文件，由系统建立的文件，如操作系统、编辑系统、编译系统等。这类文件只允许通过系统调用来执行，不允许读/写与修改。

如果按文件的属性来划分，文件又可分为：

① 可执行文件，用户可执行该程序，但不能修改。

② 只读文件，允许文件主和文件的授权者读出文件但不准改写文件内容。

③ 可读/写文件，文件主和文件授权者可以读/写文件内容。

④ 非保护文件，可以给任一用户读/写或执行。

文件的权限由文件所有者或系统授予。

如果按文件的组织方式划分，文件可划分为两类：

① 一般文件，也叫普通文件，它按一般的文件格式进行组织，如字符流文件。

② 特殊文件，如目录文件(由目录信息构成的文件)。在某些操作系统中，把 I/O 设备也定义为特殊文件。

8.1.2　文件系统及其功能

1. 文件系统的体系结构

文件系统是操作系统中实现对文件的组织、管理和存取的一组系统程序，或者说它是管理软件资源的软件，对用户来说它提供了一种便捷地存取信息的方法：按文件名存取信息，无须了解文件存储的物理位置。从这种意义上讲文件系统是用户与外存的接口。

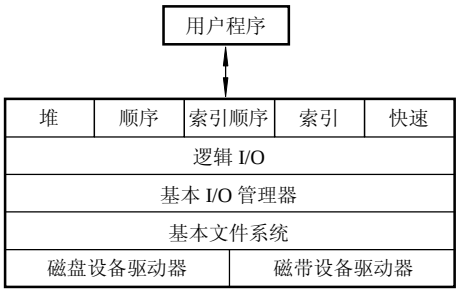


图 8.1　文件系统软件体系结构

文件系统软件的体系结构如图 8.1 所示，图中，

• 最底层的设备驱动器直接和外围设备控制器或通道进行通信。设备驱动器开始一个 I/O 操作，处理一个 I/O 请求。

• 基本文件系统(Basic File System)，或物理 I/O 层(physical I/O level)，它是与计算机系统外部环境的主要接口。它处理磁盘或磁带间交换的数据块。它关心的是这些块在辅存设备上的位置和在主存中的缓冲。它并不关心数据的内容和文件的结构。

• 基本 I/O 管理器(Basic I/O Supervisor)，负责所有文件 I/O 的初始化和文件的终止。此层处理设备 I/O、调度和文件状态。基本 I/O 管理器在被选文件的基础上，选择在什么设备上执行文件 I/O；也管理调度磁盘和磁带访问以优化性能。I/O 缓冲区和辅存都在此层。

• 逻辑 I/O(Logical I/O)作为文件系统的一部分，允许用户和应用程序访问记录。基本文件系统处理数据块，而逻辑 I/O 模块则处理文件记录。逻辑 I/O 提供通用的记录 I/O 能力，同时维护文件的基本数据。

• 最接近用户的层称为存取方法(Access Method)，图 8.1 中示出了一些最常用的方法。它在应用和文件系统以及保存数据的设备之间建立一个标准的接口。不同的存取方法反映出不同的文件结构和不同的访问处理数据的方法。

2. 文件系统的主要功能

① 实现按文件名存取文件信息，完成从文件名到文件存储物理地址映射。这种映射是由文件的文件说明(如文件头)中所记载的有关信息决定的。对用户而言，不必去了解文件存取的物理位置和查找方式，它对用户是透明的。

② 文件存储空间的分配与回收。当建立一个文件时，文件系统根据文件块的大小，分配一定的存储空间，当文件被删除时，系统将回收这一空间，以提高空间的利用率。

③ 对文件及文件目录的管理。这是文件系统最基本的功能，包括文件的建立、读、写、删除，文件目录的建立与删除等。

④ 提供(创建)操作系统与用户的接口。一般来说，人们把文件系统视为操作系统对外的窗口。用户通过文件系统提供的接口(窗口)进入系统。不同的操作系统会提

供不同类型的接口，不同的应用程序，往往会使用不同的接口，常见的接口有：

• 菜单式接口。它是用户与文件系统交互的接口，系统以不同的形式给出选项，用户通过鼠标点击，进入选项。另外，命令接口也类同，不过用户进入方式需键入命令。

• 程序接口。它是用户程序与文件系统的接口，是用户直接介入的接口，通过此接口，用户程序可获取系统的支持和服务，在用户程序中写入系统调用即可。

⑤ 提供有关文件自身的服务。如文件的安全性、文件的共享机制等。

8.2　文件的结构及存取方式

文件的结构是指文件的组织形式，文件的结构有两种，一种是逻辑结构，另一种是物理结构。

从用户观察和使用文件的角度出发所定义的文件组织形式，通常称为文件的逻辑结构。从系统的角度考察文件在实际存储设备上的存放形式，称为文件的物理结构，这一结构直接关系到存储空间的利用率。

8.2.1　文件的逻辑结构及存取方式

按文件的逻辑结构分，可将文件分为无结构的字符流式文件和有结构的记录式文件。

1. 字符流式文件

字符流式文件是由字符序列组成的文件，其内部信息不再划分结构，也可以理解为字符是该文件的基本信息单位。

这种文件的管理简单，要查找信息的基本单位困难，正因为如此，这种结构仅适于那些对文件的基本信息单位查找、修改不多的文件。常用的源程序文件、生成的可执行文件均可采用这种结构。

设计这种文件结构时应尽量考虑到，当用户对文件信息进行操作时，给定的文件逻辑结构应能使文件系统尽快查找到所需的信息，且尽量减少对存储其他信息的变动。

另外文件的逻辑结构应有利于节省存储空间。

2. 记录式文件

这是一种有结构文件。它把文件内的信息划分为多个记录，用户以记录为单位来组织信息。

记录是一个具有特定意义的信息单位，它由该记录在文件中的相对位置、记录名以及该记录对应的一组键、属性及属性值组成。

一个记录可以有多个键名，每个键名可对应于多项属性。根据系统设计的要求，记录可以是定长的，也可以是变长的。记录的长度可以短到一个字符，也可以长到一个文件。

按照记录式文件中记录的排列方式不同，记录式文件结构可分为：

① 连续结构。文件按记录生成的顺序连续排列的逻辑结构，适用于所有文件。记录的顺序并不反映在文件的内容中，即记录的顺序与记录的内容无关。这种结构便于记录的增加和修改，但不利于文件的搜索。

字符流式的文件也是一个典型的连续结构，所不同的是它的记录是一个特定的长度(一个字符)。

② 顺序结构。给定某一顺序规则，将文件的记录按满足规则的键的顺序排列起来，形成顺序结构的文件。这种结构文件按键查找，增删操作，十分方便。如用户 E-mail 地址的记录，可按用户名的字母顺序来排列以组成记录，这便是顺序记录。

③ 多重结构。多重结构可以用记录的键和记录名组成行列式的形式表示。对于由 n 个记录(每个记录含有 m 个键的文件)组成的一个 $m\times n$ 阶行列式，若第 j 个记录 R_j，含有第 i 个键 K_i，则第 i 行第 j 列的值为 1，否则其值为 0(K_i 可属于不同的记录 R)。然后将行列式中为 0 的项去掉，并以键 K_i 为队首包含 K_i 的记录为队列元素构成一个记录队列，m 个键会构成 m 个队列，此队列组成了 n 个记录文件的多重结构，如图 8.2 所示。

④ 转置结构。转置结构是在多重结构基础上加以改进的，即按上述方式组成 $m\times n$ 阶行列式后，将含有相同键 K_i 的所有记录指针连续地放在目录中的该键的位置下，如图 8.3 所示。无疑，这种结构适于按键查找。

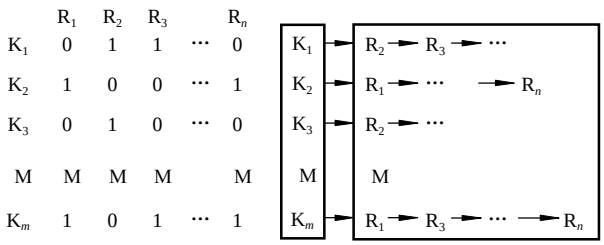


图 8.2　文件多重结构

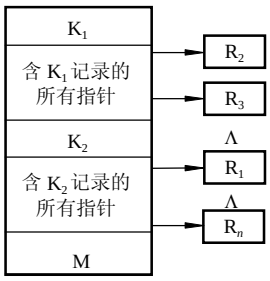


图 8.3　文件转置结构

3. 文件存取方式

文件存取方式是指用户的逻辑存取方式，从逻辑存取到物理存取之间有一个复杂的映射，逻辑存取常用的方式有：顺序存取、随机存取和按键存取 3 种，究竟采取哪种方式，这与文件的逻辑结构需存取的内容，目的有关。

(1) 顺序存取

按照文件的逻辑地址依次存取，对记录式文件，便是按照记录的排序顺序存取。如在读文件时，读完第 i 个记录 R_i，指针自动下移到第 i+1 个记录，指向 R_{i+1}。

在写操作时，写入一个记录，写指针自动移至所写文件的尾部，写完后，指针下移到新的尾部。这种操作，指针可反绕到前面，这种方法，是基于磁盘的模式。

(2) 随机存取

随机存取也称直接存取或立即存取，用户按照记录的编号进行文件存取，这种存取方式，根据存取的命令，把读/写指针直接移到读/写处进行操作。

一般用户给出可读记录的逻辑号或块号，文件系统将这逻辑号转换成相应的物理块号(一般一块为 512KB，1024KB 等)。

(3) 按键存取

按键存取是根据给定记录的键进行存取，这种存取方法大多适用于多重结构的文件，对于给定的键系统，首先搜索该键在记录中的位置，一般从多重结构的队列表中可以得到，找到键所在位置后，进一步搜索包含该键的记录。在含有该键的所有记录中查找所需记录。

当搜索到所需记录的逻辑位置后，再将其转换到相应的物理地址进行存取，如图 8.4 所示。

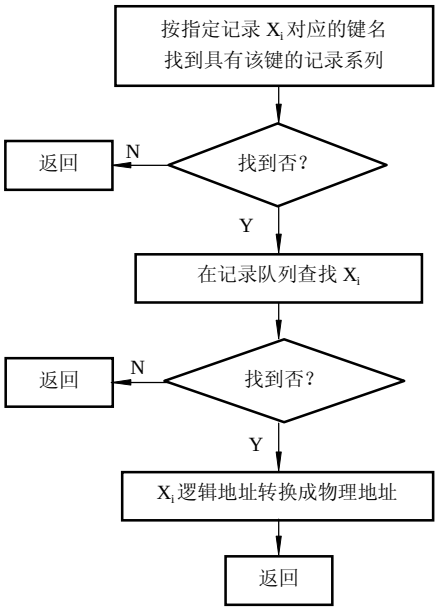


图 8.4　按键存取方法示意图

搜索是按键存取的关键，不同逻辑结构的文件，其搜索方法和效率各不相同，常用的搜索算法有线性搜索、散列二分法等。

8.2.2　文件的物理结构及存储设备

1. 文件的物理结构

文件的物理结构是指文件在存储器上的存放方式，以及它与文件的逻辑结构之间的关系，这种物理结构实际上是文件的存储结构。文件的存储设备不同，相应存储上的文件的结构也应有所不同。存储设备(如磁盘)通常将存储空间划分为相同大小的物理块。不同的操作系统，其块的大小不同，有的盘块大小是 1024KB，有的是 512KB，也有的为 256KB，信息的传输以块为单位。显然，字符流文件(等记录长文

件)，在每个物理块中存放了长度相等的信息，而对记录文件来说，由于文件记录不一定是等长的，因此，每个物理块上存放的文件信息长度可能会不同。其逻辑块到物理块的映射无疑也会较复杂。

通常文件物理结构有顺序文件、链接文件、索引文件 3 种。

(1) 顺序文件

按文件的逻辑记录顺序把文件放在连续的存储块中。文件系统为每个文件都建立一个文件控制块 FCB，它记录了文件的有关信息。对于顺序文件，只要从 FCB 中得到需存放的第一个块的块号和文件长度(块数)，便可确定该文件存放在存储器中的位置，如图 8.5 所示。

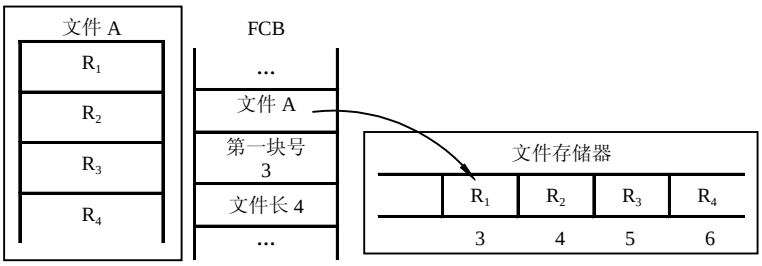


图 8.5　顺序文件的存放方式

这种存放方式的优点是，实现简单，存取速度快，常用于存放系统文件等固定长度的文件；缺点是，文件长度不便于动态增加，因为一个文件末尾处的空块可能已分配给其他文件。一旦增加记录，便会导致大量移动。另外，在反复删除记录后，便会产生“碎片”，导致存储空间利用不充分。

(2) 链接文件

一个逻辑上连续的文件，可以存放在不连续的存储块中，而每个块之间用单向链表链接起来。其中，每个物理块设有一个指针，指向其后续连接的另一个物理块，从而使得存放同一文件的物理块链接成一个串联队列。这种文件结构称为链接文件。如图 8.6 所示，逻辑上连续的文件 05，06，07，08 块，依次存放在离散的物理块 21，18，28，5 物理块上。

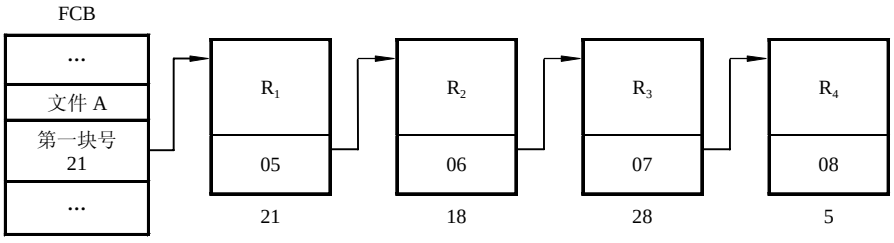


图 8.6　链接文件的存放方式

只要从 FCB 找到该文件的第一个块号，便可知道要存放的第一个逻辑块(如 05 块)存放在第 21 个物理块中，在该块的尾部存放了第二个逻辑块(如 06 块)的物理块号 18……依次类推，若在两个逻辑块之间插入和删除某块，只要调整其指针即可。

链接文件的优点是不要求对整个文件分配连续的空间，从而解决了空间碎片问题，提高了存储空间利用率，也克服了顺序文件不易扩充的缺点。链接文件的缺点是随机存取文件末尾的记录时，必须从头到尾依次存取，其存取速度较慢。链接指针要占去一定的存储空间。

(3) 索引文件

索引文件是由系统为每个文件建立一张索引表，表中标明文件的逻辑块号所对应物理块号，索引表自身的物理地址由 FCB 给出，索引表结构如图 8.7 所示。存取文件时必须先读索引表，使操作速度变慢。为了弥补这个缺点，通常在读/写文件之前，先将盘上的索引表存入内存缓冲区、以便加快速度。

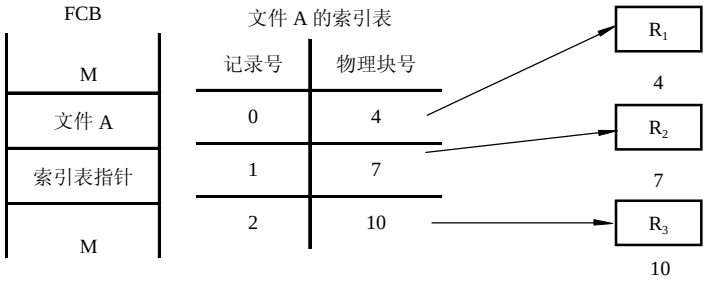


图 8.7　索引表结构

索引文件克服了连续文件和链接文件的不足，它既能方便迅速地实现随机存取，又能满足文件动态增删的需要。

对于上述索引情况，如果索引表很大，超过了一个物理块，则系统势必要像处理其他文件一样，来处理索引表的物理存放方式，这样不利于索引表的动态增删。解决的办法是采用多重索引的方式，也就是说，当索引表所指的物理块超过一块时，再增加一个寻找索引块的索引表，此级索引表所指的物理块中存放的不是文件信息，而是装有这些信息的物理块地址，如这级索引表中有 n 个物理地址，每一个可指向存有 n 个块物理地址的二级索引，故可存入的文件长度为 $n \times n$ 块，如图 8.8 所示。

不过，大多数文件不需要进行多重索引，也就是说，这些文件所占用的物理块号可以放在一个物理块内。如果对这些文件也采用多重索引，显然会降低文件的存取速度。因此，在实际系统中，总是把索引表的前几项按直接寻址方式设计，也就是这几项所指的物理块中存放的是文件信息；而索引表的后几项存放多重索引，也就是间接寻址方式。如果索引表较短，就可利用直接寻址方式找到物理块号而节省

存取时间。

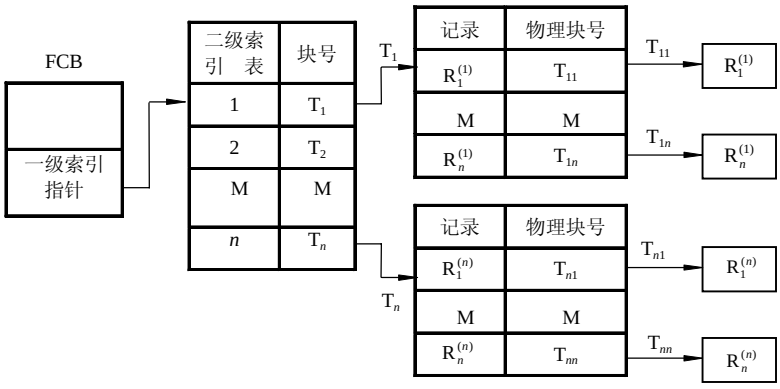


图 8.8　多重索引表

索引结构既适用于顺序存取，也适用于随机存取。索引结构由于使用了索引表而增加了存储空间的开销，且在存取文件时至少需要访问存储器两次，一次是访问索引表，另一次是根据索引表提供的物理块号访问文件信息。文件信息和索引表信息都在存储设备上，无疑会降低文件的存取速度，为了解决速度问题，当对某个文件进行操作之前，系统预先把索引表放入内存。这样，文件的存取就可直接在内存通过索引表进行，进而减少了访问磁盘的次数。

2. 文件的存储设备

文件的存储设备分为不可重复使用的和可重复使用的两类。

不可重复使用的文件存储设备也称为 I/O 式字符设备，如打印纸等。

可重复使用的文件存储设备有磁带、磁盘、光盘等，也称块设备。文件存储设备的特性决定了文件存取方法，下面仅介绍两种典型的存储设备特性及存取方法。

(1) 顺序存取设备

顺序存取设备通常是指那些容量大、价格低的存储设备。磁带是一种典型的顺序存储设备，数据以块的形式存放，只有在前面物理块被存取访问之后，才能存取后续的物理块，块与块之间用间隙分开，这个间隙用于控制磁带机，以正常速度读取数据，在读完一块数据后自动停滞。显然，间隙间的数据块越大，读/写速度越快。所以，每块一般要包含一个以上的记录。磁带中的每个物理记录都有一个惟一的标识号，即物理记录号。

磁带设备的数据传送率主要取决于信息密度(字符数/英寸)、磁带带速(英寸/s)和块间间隙。

如果从磁带上随机存取一条记录，或一个物理块，若该块离磁头当前位置太远，

则系统将花费很长的时间移动磁头，即花大量时间在寻找记录上，一旦找到记录块后，真正读取数据的时间却很短。一般读取块记录时间是整个存取时间的一半，因此，磁带文件不适合随机存取，最适合顺序存取文件。

(2) 直接存取设备

光盘、磁盘都是一种可直接存取的存储设备(磁盘又分为硬盘和软盘)。

① 磁盘

磁盘是一种可直接存取(按地址存取)的存储设备，它把信息记录在盘片上，每个盘片有正反两面。硬盘是将若干张盘片固定在一根轴上组成一个盘组，沿一个方向高速旋转。每个盘面有一个读/写磁头，所有的读/写磁头都被固定在惟一的移动臂上同时移动，如图 8.9 所示。

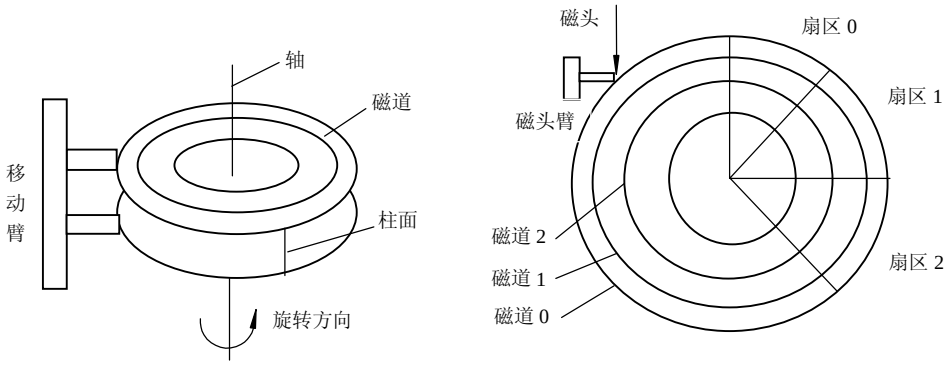


图 8.9　磁盘结构示意图

把所有的读/写磁头按从上到下顺序编号，此号称为磁头号。每个盘面上有多个磁道，各个盘面对应磁头位置的磁道在同一个圆柱面上，这些磁道组成了一个柱面。每个盘面上的许多磁道，形成不同的柱面。由外向里把盘面上磁道的编号作为柱面号。移动臂通过移动读/写磁头，当磁头移动到某一位置时，所有的读/写磁头都在同一柱面上访问所有磁道。另外每个盘面又被划分成相等的扇区，每个扇区将各个磁道分成相等的小段，叫磁盘块，每个块上存放相等字节数的信息(如每个盘块放 512B 或 1024B)。

② 只读型光盘

光盘存储器是利用光学原理存取信息的存储设备。它的主要特点是存储容量大，价格低廉，而且存放信息的光盘可从光驱中取出，单独长期保存。因此，目前计算机系统中大都使用了光盘存储器。

光盘存储器按其功能不同，可分为只读型(CD-ROM)、一次写入型(WORM)和可擦写型 3 种。

只读光盘存储器由两部分组成，即只读光驱和光盘片，只读光驱通过连线接在SCSI、IDE 以及 EIDE 接口上。用户的光盘片插入光驱内即可输入数据。

只读光盘片上的信息一般是由厂家用某种生产工艺预先刻录的。光盘片是一张直径为 5.25 英寸(1 英寸=25.4mm)的圆形塑料片，盘面上的信息是由一系列宽度为0.3~0.6μm、深度约为 0.12μm 的凹坑组成，凹坑以螺旋线的形式分布在盘面上，有坑为 1，无坑为 0，由于凹坑非常微小，约 1μm^2，其线密度一般为 1000B/mm，道密度为 600~700 道/mm，一张 5.25 英寸的光盘上可存放 680MB 信息。

厂家生产的过程是，先制母盘、再刻录、喷镀、注塑。

光盘片上数据的存储格式与磁盘数据存储格式有许多相似之处。光盘采用的是ISO9600 标准。

光盘的一个物理扇区有 2352B，除了同步、首标、校验码外，可存放数据 2048B，其一个扇区的数据格式如图 8.10 所示。

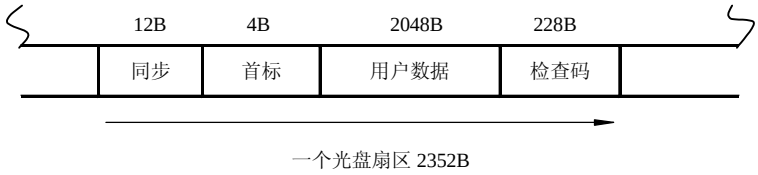


图 8.10　光盘扇区格式

光盘片在光驱中是以恒定线速度旋转的，盘面上的光道是螺旋形的，采用时间为地址，不同于磁盘，磁盘是以恒定角速度运转，磁盘上磁道是环形同心圆，采用面号、道号、扇区号为地址。

光盘和磁盘都采用分层目录结构，但查找文件的方式不完全一样。光盘采用一种称为路径表的方法，通过它可直接访问任何一个子目录，路径表与目录结构的关系如图 8.11 所示。

3. 文件结构、存储设备与存取方式

综上所述，文件的物理结构，必须适应文件的存储设备，而不同的存储设备的特性，又决定了其上的文件的存取方式，下面以磁盘和磁带存储设备为例，简要说明 3 者的关系：

① 磁盘上的文件结构为连续时，其存取方式一般为顺序或随机。

当文件为连续方式时，存取方式通常为顺序的。

② 磁带上的文件结构为连续时，其存取方式一般为顺序存取。

当其上文件为索引文件时，存取方式可为顺序、随机两种形式。

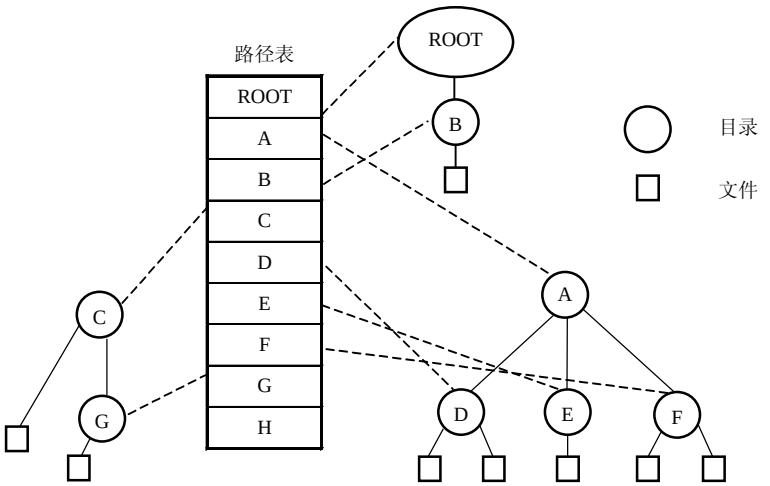


图 8.11　路径表与目录结构

8.3　文件管理

8.3.1　文件目录结构

1. 文件目录

文件系统为程序和用户提供了按文件名存取文件的机制，而将文件名转换为存储地址，以及对文件实施控制管理则需通过文件目录来实现。

文件目录的管理和文件存储空间的管理已成为文件管理的重要内容。本节将讨论文件目录的管理。

一个文件由文件说明和文件体组成。文件说明部分包括文件的基本信息、存取控制信息和文件使用信息。

① 基本信息包括：

• 文件名，用于标识一个文件的符号名。每个文件必须具有惟一的文件名，这样，用户可以按文件名进行文件操作。

• 文件物理位置，标明文件内容在外存上的存储位置。包括存放文件的设备名、文件在外存中的起始地址、文件的长度。

• 文件结构，指示文件的逻辑结构和物理结构。它决定了文件的寻址方式。

② 存取信息包括：各类用户的存取权限，实现文件的共享及保密。

③ 使用信息包括：文件的创建日期和时间，以及当前使用的状态信息。

文件系统将这些说明部分的全部信息集中起来，以一个数据结构的形式表示，称此结构为文件控制块 FCB (File Control Block)。

文件目录由文件控制块组成。文件系统在每个文件建立时都要为它建立一个文件目录。文件目录用于文件描述和文件控制，实现按名存取和文件信息共享与保护，随文件的建立而创建，随文件的删除而消亡。

不同的操作系统有不同的文件目录。文件系统将若干个文件目录组成一个独立的文件，这种仅由文件目录组成的文件称之目录文件。它是文件系统管理文件的手段，文件系统要求文件目录以及目录文件占有空间少，存取方便。

概括起来，目录包括了文件的基本信息、地址信息、存取控制信息和使用信息，如表 8.1 所示。

表 8.1　文件目录的组成

基本信息	文件名	生成者(用户或程序)选择名字，名字在此目录中是惟一的
	文件类型	如文本、二进制、加载模块等
	文件结构	支持不同结构的系统
地址信息	卷	暗示文件存储的设备
	起始地址	辅存中的起始物理地址(如磁柱、磁道、块数)
	使用的大小	文件现在的大小(字节、字或块)
	分配的容量	文件最大的容量
存取控制信息	所有者	具有文件控制权的用户。所有者可以允许或禁止其他用户的存取，也可以改变这些权限
	存取信息	包含每个授权用户的用户名和密码
	允许的工作	管理读、写、执行和网络上传输
使用信息	生成的数据	文件第一次放置在目录中的数据
	生成者标识	通常但不一定是当前所有者
	最后读到的数据	最后一次读记录中的数据
	最后读者的标识	最后一次读数据者的标识
	最后修改的数据	最后一次更新、插入或删除的数据
	最后修改者的标识	最后一次修改者的标识
	最后备份的数据	最后一次文件备份的数据
	现在的使用情况	关于文件当前活动的信息，如打开文件的进程，文件是否由进程上锁，文件是否在主存中被更新，而没有在磁盘上更新

下面以 UNIX 文件目录为例加以说明。

UNIX 系统的文件目录由目录项和索引节点两部分组成。目录项占 16B，其中 14B 为文件名，2B 为指向文件说明信息的索引节点的指针，每个索引节点占 64B，包括文件属性、文件共享目录数、时间、文件存放块号、文件长度等说明信息，如

图 8.12 所示。

2. 文件目录结构

文件目录是由文件说明组成的，若干个文件目录组成一个专门的目录文件，目录文件的结构如何，关系到文件的存取速度和文件的共享及安全特性，因此，下面将介绍几种常用的文件目录结构。

文件目录结构是指专门的目录文件的组织形式。常用的目录结构有单级目录，二级目录和多级目录。

(1) 单级目录

文件系统在每个存储设备上仅建立一个目录文件的目录结构，称为单级目录(或称一级目录)。目录文件中的每一目录项(或称一条记录)对应一个文件目录，它包含相对的数据项(文件名、物理地址、说明信息)，如图 8.13 所示。

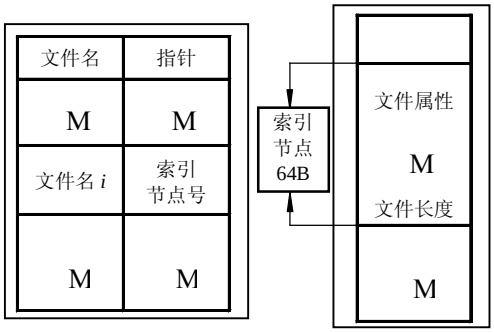


图 8.12　UNIX 系统的文件目录

图 8.13　单级文件目录结构

单级目录的优点是结构简单，通过管理其目录文件，便可实现对文件信息的管理。通过物理地址指针，在文件名与物理存储空间之间建立对应关系，实现按名存取文件。

单级目录的特点是：

① 搜索范围宽。搜索一文件有时涉及到整个目录文件中的所有目录项，开销大、速度慢。

② 不允许文件重名。在一个目录文件中，不允许两个不同的文件具有相同的名字，这在多用户环境中是不合适的。

(2) 二级目录结构

二级目录结构将存储在设备上的目录文件分成两级：第一级为系统目录(称主目录 MFD)，它包含了用户目录名和指向该用户目录的指针；第二级为用户目录(称 UFD)，它包含了该用户所有文件的文件目录，该文件目录和上述单级的目录一样，包含了相应文件的名字，物理地址等，如图 8.14 所示。图中，系统目录列出了两个用户目录名：User 1 和 User *N*。每个用户有一个用户目录，由系统目录中的用户目

录指针指示。用户 User l、User *N* 都可以使用文件名 XQX。

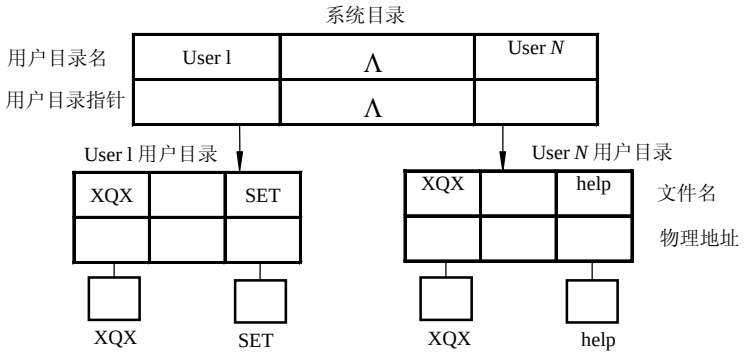


图 8.14　二级目录结构

从二级目录结构可以清楚看到，每一个文件都是由系统目录中的用户目录和用户目录中的文件名标识，这种标识具有惟一性。因为文件系统在建立系统目录时，对不同用户目录会有不同标识，在不同的用户目录中，即使有相同的文件名也无妨，这就解决了单级目录中用户之间文件不能重名的问题。

在二级目录中，访问一个文件首先在系统目录中按用户目录名查找相应的用户目录，然后再在相应的用户目录中按文件名查找文件的物理地址。

在这种方式的基础上扩展，便产生了多级目录结构。

(3) 多级目录结构

采用树型数据结构方法，便形成一种树型的结构目录。

这种文件目录的第一级系统目录为树的根节点，定义为根目录，文件目录的第二级和以下各级目录均为树的分支节点(非终节点)，均定义为子目录，只有树的叶节点(终节点)才为文件。例如，Netware 网络操作系统便是采用这种方式对服务器上的磁盘文件进行组织和管理的。它将服务器上的文件系统分成若干级：卷、目录、子目录、文件，构成一个树状的目录结构。

• 卷是指目录结构的第一级，或者称为树的根，目录结构的起点。在文件服务器上的卷可理解为一个逻辑设备，它可映射(对应)为若干个硬盘，而一个硬盘也可以划分为多个卷。

• 卷的下一级一般为目录(也可以直接为文件)，称为树枝。它是提供有关文件信息的一张表，目录的下一级可以是多层的子目录，不论是目录还是子目录，均视为目录。

• 目录的下一级即为文件，称为树叶。

Netware 的文件目录结构复杂，为适应多级管理的需要，它在文件服务器上往往

建立了多卷目录结构，即建多个卷，每个卷上又建了多级目录，每个目录下建立多个文件。具体表现形式：

文件服务器名 ——┬ 卷 1：目录/子目录　1/…/子目录 *n*/文件
　　　　　　　　M
　　　　　　　　└ 卷 *n*：目录/子目录　1 /…/子目录 *n*/文件

服务器名、卷名、目录名、子目录名都由若干个字符表示，这一连串的字符，确定了文件在磁盘上的位置。

Netware 系统在安装时，在建立 SYS 卷的过程中，自动建立一组目录表，称为基础目录结构。

从根目录经各级子目录到达文件的通路上的所有子目录名称为文件的存取路径。

在多级目录结构中，要访问一个文件必须从根目录开始，逐级查找各级子目录，直到文件。无疑这样查找速度较慢。文件系统中的一个进程运行时所访问的文件，往往仅局限在某个范围里，因此，查找不一定要从根目录开始，有必要为系统建立一个当前目录，查找文件时从该目录开始，称这一目录为工作目录，不同的文件系统，都可以设置这一目录。当用户不另外指定缺省目录时，系统从该目录起进行查找。

3. 文件目录与文件共享

系统中有许多公用的文件，被若干用户使用，如果每个用户都在系统内保留这些可用文件的副本，无疑会造成存储空间的浪费，也达不到文件共享的目的。

为了有效的实现文件共享，文件系统在建立文件目录的过程中，采用了以下两种方法，使文件只需保存一个副本，达到多个用户共享的目的。

(1) 绕道法(交叉法)

绕道法查找共享文件的方法是每个用户从各自当前目录开始，向上返回到共享文件所在路径的交叉节点，然后沿交叉节点顺序向下访问到共享文件。

用户需要指定所要共享文件的路径，如图 8.15 所示。

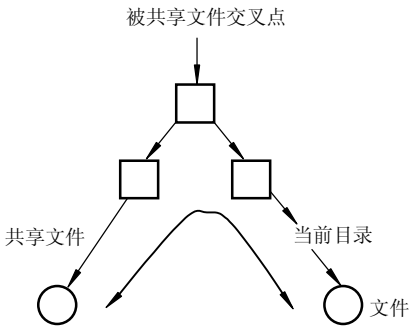


图 8.15　绕道法

(2) 基本文件目录表法

为了有效实现系统文件的共享，文件系统需建立一基本文件目录 BFD，它包括了文件的结构、物理块号、存取控制和管理信息。另外，需增加符号文件目录表 SFD，包括用户给定的符号名和系统文件赋予的文件说明信息的内部标识符。

主目录(MFD)记录了文件名和系统给定的惟一标识，如图 8.16 所示。

为实现文件有效共享，将可用文件直接登入基本文件目录 BFD，在 BFD 中记录了共享文件的物理块号及系统给出的惟一的内部标识，同时主目录 MFD 的物理地址

也登入 BFD。在每个用户的符号文件目录中都列入需共享的文件名及它的内部标识。

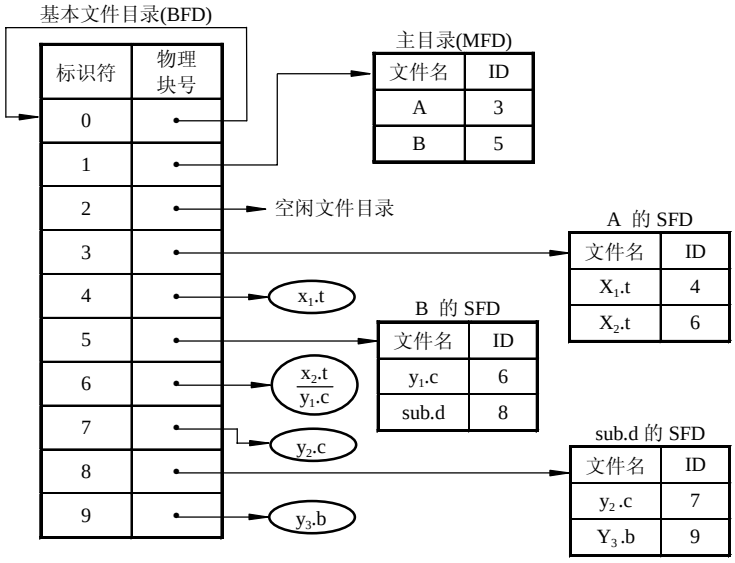


图 8.16　文件目录表

当系统访问查找某一共享文件时，只需通过用户内部标识找到主目录，从主目录找到用户的 SFD，再从 SFD 找到共享文件名及对应的内部标识，然后，从 BFD 内部标识查到共享文件的块号，达到共享的目的。

8.3.2　文件目录管理

如上所述，文件的目录是以目录文件的形式存放的，当存取一个文件时，往往需要访问多级文件目录，如果对每一级目录访问都需要到文件存储设备上去搜索，势必占用过多的 CPU 时间，且降低处理速度，人们曾一度设想在系统启动时，把全部目录文件读入内存，由系统直接在内存实施对各级目录的搜索，减少 I/O 设备的开销，这样有利于提高访问速度，但是这种方法需要的内存容量大，显然是不可取的。

一般来说，系统只把当前正在使用的那些文件的目录表复制到内存中，为此，系统提供两种特殊操作，其一是把有关的目录文件复制到内存指定区，通常称为打开文件(Open)；其二是提供用户不再访问的有关文件的目录文件删除的操作，通常称为关闭文件(Close)。

下面以按照 BFD、MFD、SFD 方式组织的多级文件目录为例说明对于 Open 操作的实现过程：

① 系统首先把主目录 MFD 中与待打开文件相关联的表目复制到内存。如图 8.16 所示，如果拟打开的文件为 $y_2.c$，则将 MFD 中的第二项(B)复制到内存。

② 根据复制项中 ID 对应的标识符，将它所指明的基本文件目录表(BFD)的有关表目项，包括控制存取信息、下级目录 SFD 的块号信息等复制到内存。

③ 搜索 SFD 以找到与待打开文件相应的目录表项，例如，搜索 SFD 中的 y_1．c，sub.d，若 y_1.c 不是待打开文件，找到的表目仍是子目录名 sub.d，则根据子目录对应的标识符 ID，继续复制 BFD 中目录的 ID=8 的表目项，包括有效控制信息、结构信息及下级目录物理块号等。根据上述说明信息搜索 sub.d 子目录的 SFD 便可找到待打开的文件名 y_2.c。

④ 根据以上所搜索到的文件名所对应的标识 ID=8，把相应的 BFD 的表目项复制到内存。这样，待打开的文件的说明信息就已复制到内存中。由复制的文件说明，系统显然可以方便地得到文件的有关物理块号。从而，系统可对文件进行有关操作。

经过上述操作打开的文件，称为活动文件，内存中存取活动文件的 SFD 表目的表称为活动名字表，每个用户一张。

内存存取所有活动文件的 BFD 是一个整体活文件表。

8.4　文件存储空间的分配与管理

由文件的存储结构可知，文件信息的交换都是以块为单位进行的。因此，将文件存储设备称为块设备，这里介绍的存储空间的管理实际上是对文件块空间而言的，具体说是指空闲块的组织与回收。

一般来说，空闲块空间的分配常常有两种方式，一种静态分配，即在文件建立时一次性分给所需的全部空间；另一种是动态分配，即根据动态增加文件的长度进行动态分配，甚至每次可分配一块。

另外在分配区域大小上，可以连接分配在一个完整分区（以块或簇为单位），常使用包含文件名、起始地址、长度的文件分配表 FAT 等。

8.4.1　文件存储空间的分配

文件空间分配常采用：连续分配、索引分配、链接分配 3 种方法。

1. 连续分配

连续分配方式是将文件存放在辅存的连续存储区中。在这种分配算法中，用户必须在分配前说明被建文件所需的存储区大小。然后系统查找空闲区的管理表格，判断是否有足够大的空闲可使用。如果有，就分给其所需的存储区；如果没有，该文件将不能建立，用户进程必须等待，如图 8.17 所示。

连续分配的优点是查找速度快。目录中关于文件的物理存储区的信息也比较简单，只需要起始块号和文件大小。其主要特点是容易产生碎片。

显然这种方法不适合文件动态增长和减少的情况，也不适合用户事先未知文件有多大的情况。

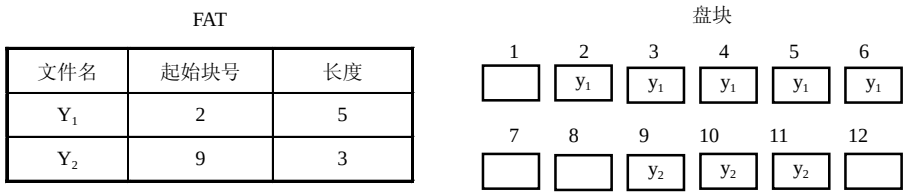


图 8.17　文件的连续分配

2. 索引分配

索引分配方法主要是利用文件分配表 FAT 给每个文件分配一个指出该文件的索引表所在的物理块号的表目，索引表自身所在的块是独立的，并不在分配给文件的磁盘块中，即索引块与文件盘块是分离的。文件索引的每个表目的设置有两种情况：一种是直接给出索引文件各物理块；另一种是设置文件的起始块和长度，这有利于连续分配，也有利于节省索引表空间、提高效率，如图 8.18 所示。

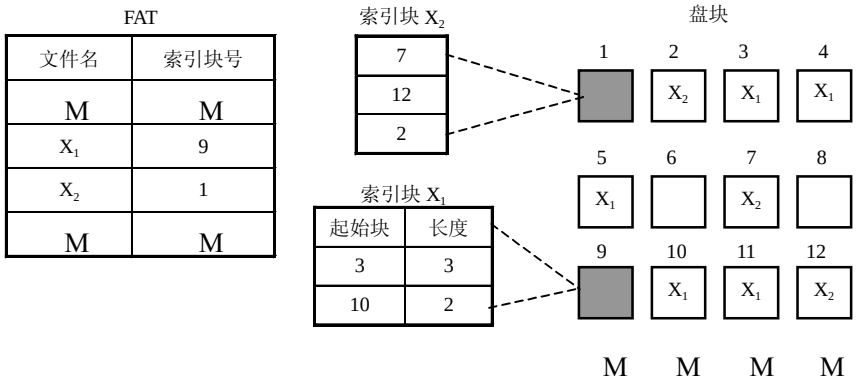


图 8.18　文件的索引分配法

3. 链接分配

链接分配文件空间的方法适用于文件长度需动态增减，或用户对其文件的应用不十分明确的情况，一般分配非连续的辅存空间。采用链接表方法链接存储空间，链接空间的大小大多以区或段为单位。

(1) 以扇区为链接单位

这是给需动态变化的文件分配若干磁盘扇区，这些扇区在磁盘上可以不连续，而同一文件所分配的各扇区按其上文件逻辑记录的次序用链指针链接起来。当文件需增加时从空闲扇区表中分给所需扇区，修改链指针链接到文件链上去，同样当文

件缩短时，将对释放出的扇区，修改链指针，放回空闲扇区表，如图 8.19 所示。

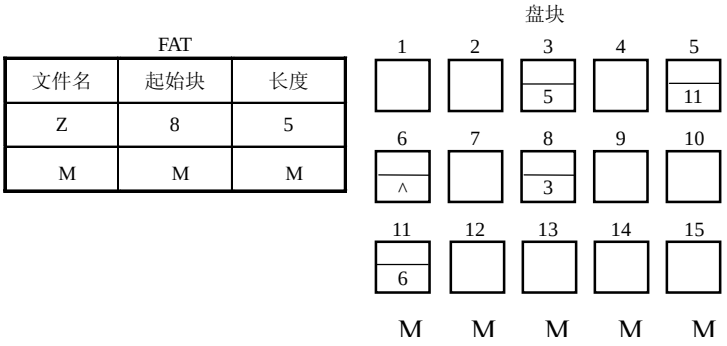


图 8.19　文件的链接分配

由上可知，这种方法的最大优点是有利于消除碎片，但查找逻辑连续记录时间长，链指针占存储空间和链维护开销大。

(2) 以区段(或簇)为单位分配

这不是以扇区为单位进行分配，而是以区段(或称簇)为单位进行分配的。区段是由若干个连续扇区所组成。一个区段往往由整条或几条磁道所组成，文件所属的各区段可以用链指针、索引表等方法来管理。当为文件动态地分配一个新的区段时，该区段应尽量靠近文件的已有区段号，以减少查寻时间。

8.4.2　磁盘空间管理

文件的磁盘存储空间的管理包括磁盘空间块的分配和回收。磁盘空间的有效管理有助于提高整个操作系统的效率，它涉及到盘块大小的划分及管理的方式。

1. 盘块

盘块是操作系统传输数据的基本单位，盘块大，I/O 操作传输数据量多，传输性能好，但也会造成盘空间的浪费。既要提高传输率，又要减少盘空间的浪费，是文件系统追求的目标，盘块是重要因素之一。

(1) 逻辑块

逻辑磁盘是文件系统中一个抽象的存储概念。系统将逻辑磁盘视为一些有固定大小可随机存取的逻辑块的线性序列。磁盘驱动程序将逻辑块映射到物理介质上。一般情况下，一个物理磁盘被分成物理上连续的几个分区，每个分区就是一个逻辑磁盘，又称磁盘分区。

通常所说的磁盘分区就是将每一个分区定义为一个盘，此盘就是一个逻辑磁盘。

(2) 盘区

磁盘分区是将磁盘上一组连续的柱面空间组成一体，定义为一个盘区。其上可

有一个独立的文件系统。不同类的文件系统可占有不同的盘，各自定义自己盘块的大小。为适应当今分布式文件系统和远程网络文件系统的应用需要，操作系统必须能支持多种文件系统(如 Windows NT 可支持 4 种文件系统)。为此，一个磁盘系统可以拥有多个盘区，一台机器有多个文件系统。

这些不同的文件系统可通过虚拟的文件系统接口，给用户一个统一的界面。

扇区是盘空间计量单位，不同系统其扇区大小不一定相同，如 Netware 中每一个扇区定义为 512B。扇区在盘空间管理中，用于计算装载模块的大小。如在操作系统 Netware 中：

装载模块大小= (文件所占用的扇区数–1)×每扇区的字节数
+文件在最后一个扇区中的字节数–文件头长度

从被加载文件的文件头中可以得到有关扇区的信息。Netware 中的 server.exe 文件头信息如下：

00—01H	5A4DH	有效 EXE 文件的标志；
02—03H	01B4H	文件在最后一个扇区中的字节数；
04—05H	0067H	文件所占用的扇区数；
06—07H	0000H	重定位项数为零；
08—09H	0020H	文件头的长度为 20 节（512B）；
0C—0D	FFFFH	程序在内存的低端运行；
0E—0FH	0000H	堆栈段的段偏移为 0000H；
10—11H	DDB4H	堆栈段指针的初始值为 DDB4H；
14—15H	0000H	程序的第一条指令的偏移量为 0000H，即 IP=0000H；
16—17H	0000H	程序的第一条指令的段偏移为 0000H；
1A—1BH	0000H	程序中无覆盖。

从文件头取得有关的区值，计算如下：

装载模块大小= ([04–05H]–1)×200+[02–03H]–([08–09H])×10H
= (67H–1)×200+01B4H–200H
= CBB4H

2. 磁盘块大小

① 磁盘块大小，可以理解为磁盘分配的单位，它规定了文件系统的分配粒度和磁盘 I/O 粒度，盘块大，有利于增加系统性能，不同的文件系统块大小也不同，FFS 可大于等于 4KB，NTFS 可大到 64KB。

② 片断。是盘块的组成单位。为了提高存储空间利用率，减少盘空间的浪费，文件系统将盘块再细分为一个或多个片断，其大小在文件系统建立时被确定，最小不能小于扇区大小。有些文件系统规定只有文件的最后一块才使用连续的片断。一块中剩下的片断可用于其他文件的最后一块。

有些系统并没有用到片断，而是以扇区为单位，如 Netware 的文件系统。

3. 盘块管理

盘块管理常用盘图，链表和 *i* 节点等手段，因文件系统而异。

(1) 盘图法

盘图也称字位映像图，是一种常用的方法，它用位(bit)的值 0、1 来表示磁盘上相应物理块是否被分配，bit 值为 1 表示对应物理块被分配，为 0 表示对应物理块为空闲。

对应一串连续的 bit 值，按字节构成一张表，此表可以把一个完整磁盘的使用情况记载下来，且存入主存，查找此表会方便地找到连续的磁盘空闲块。

Windows NT 就是使用盘图来管理磁盘空间的。

(2) 链接法

① 链接索引块。这是一种常用的方法，它首先是选择若干空闲物理块建立索引表块，假设这样块的大小为 1KB，可以设 512 个表目，每个表目占用 16 位，以此表示一个空闲物理块的块号，则每个表目对应一个空闲物理块。而后将这些含有空闲块号的索引块之间用链接方式链接起来，即每个索引块的第 0 个表目作为链表的指针，指向下一个索引块，或链尾标志。而链表的头指针放在特殊指定的块中，如图 8.20 所示。

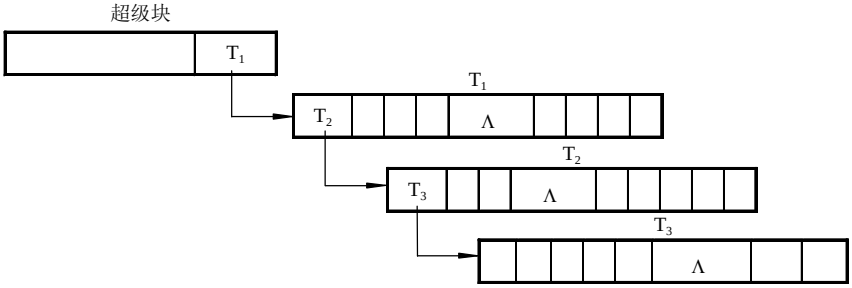


图 8.20　链接索引块

② 分配与回收空闲块。为了操作方便，通常将索引链表中的链头指针所指向的索引块的表目中留出空项(其他索引块表目项全填满)，当文件系统分配盘空间时从链表头的索引块的块尾开始，直到该索引块的第 0 个表目，如果该索引块仅剩下第 0 个表目，则该表目的内容读到特定块链头指针中，然后将原链头指针指向的索引块 T，分给请求分配空闲块的文件。

空闲块的回收则相反，仅将释放的空闲块块号加到链头指针指出的索引表块的尾部表目中即可。

这种方法的优点是，在进行空闲块分配时，只需将链表索引块读入内存，这比

盘图方法所占内存要小。

8.5　系统举例——Windows NT

无论哪种操作系统，它都要以某种特定方式呈现在用户的面前，这种方式便是文件或文件夹，因为人们常常认为文件系统是操作系统中最直接可见的部分。

因为不同的操作系统在设计时，对它所管理的可执行文件信息采用了各自不同的存储、打印、显示形式，增加了诸多的辅助部分，如文件首部、信息块表、信息块、辅助信息等等，并用不同数据结构予以描述，以此支持系统文件的执行，通常人们把这种特定的数据结构称为文件的格式。如 OS/2 采用了 LX 格式，16 位 Windows 操作系统采用了 NE(New Executable)分段可执行文件格式，Windows 3.x 操作系统采用了 LE(Linear Executable)线性可执行文件格式。Windows NT 和 Windows 98 采用 PE(Portable Executable)移动可执行文件格式。

本节以 Windows NT 的可执行文件为例，介绍可执行文件格式及其管理机制。

8.5.1　PE 可移动执行的文件格式

PE(Portable Executable)是一种可移动执行的文件格式，以其简单、灵活的特点，得到了所有基于 Win32 系统的普遍应用。

1. PE 文件格式

PE 文件格式由 5 部分组成：即 MS-DOS 首部、PE 首部、信息块表、信息块和辅助信息，如图 8.21 所示。

图 8.21　Windows NT PE 文件格式图

2. WINSTUB 的设置

PE 文件格式的可执行文件保留了一个称之为文件头的首部信息，它是 PE 文件格式的重要组成部分，它概括了该文件的基本信息，包括文件的代码和数据的长度及地址、堆栈大小、文件适应的操作系统等。在这个文件首部前，设置了一个特别的文件首部 WINSTUB(MS-DOS 文件的首部)。在 Windows NT 可执行文件的首部前还保留着 128B 的首部信息，原因是：

① 为了表明具有 PE 文件格式的文件必须具备的运行环境(即 Windows NT 或 Windows 95/98)，一旦 PE 类文件在非此类 OS 下运行，如在 MS-DOS 下运行，系统自动显示“This program requires Microsoft Windows”之类的信息，以提示用户该程序需要在 Windows NT 环境下运行。如果该文件的确是 PE 格式文件，则系统自然跳过 WINSTUB 首部，而进入 PE 真正文件首部。

② PE 首部的起始地址隐含在 MS-DOS 首部之中，该首部的最后 4 个字节的 LONG e_lfanew 字段便是真正 PE 首部的相对偏移。

下面以 Windows NT 中的 SETUP.EXE 为例来说明。

SETUP.EXE 的 WINSTUB 的数据结构如下：

```
Typedef struct_IMAGE_POS_HEADER
  {// Dos. EXE header
WORD e_magic ;          // Magic number
WORD e_eblp ;           // Bytes on last page of file
WORD e_cp ;             // Pages in file
WORD e_cparhdr ;        // Size of header in paragraghs
WORD e_minalloc ;       // Minimum extra paragraphs needed
WORD e_maxalloc ;       // Maximum extra paragraphs needed
WORD e_ss ;             // Initial (relative) SS value
WORD e_sp ;             // Initial SP value
WORD e_csum ;           // Checksum
WORD e_ip ;             // Initial IP value
WORD e_cs ;             // Initial (relative) CS value
WORD e_lfarlc ;         // File address of relocation table
WORD e_ovno ;           // Overlay number
WORD e_res [4]          // Reserved words
WORD e_oemid ;          // OEM identifier (for e_oeminfo)
WORD e_oeminfo ;        // OEM informa tion ; e_oemid specific
```

```
WORD e_res2 [10] ;          // Reserved words
LONG e_lfanew ;             // File address of new exe header
} IMAGE_DOS HEADER, * PIMAGE_DOS_HEAD_ER;
```

相应的 SETUP.EXE 的 WINSTUB 的数据：

```
OC32：0100 4D 5A 90 00 03 00 00 00- 04 00 00 00 FF FF 00 00
OC32：0110 B8 00 00 00 00 00 00 00- 40 00 00 00 00 00 00 00
OC32：0120 00 00 00 00 00 00 00 00- 00 00 00 00 00 00 00 00
OC32：0130 00 00 00 00 00 00 00 00- 00 00 00 00 80 00 00 00
OC32：0140 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68
OC32：0150 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F
OC32：0160 74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20
OC32：0170 6D 6F 64 65 2E 0D 0A-24 00 00 00 00 00 00 00 00
```

(SETUP.EXE)PE 首部的起始地址隐含在 WINSTUB 中，e_lfanew 字段就是真正 PE 首部的相对偏移(e_lfanew 为 80H)，要得到内存中 PE 首部指针，只需用基地址加上 e_lfanew 即可。

8.5.2　PE 文件首部

从 WINSTUB 的作用得知，隐含在 WINSTUB 中的相对偏移地址可以找到 PE 文件的首部，系统将 PE 文件首部的结构定义为一个完整的 IMAGE-NT-HEADERS 项。即由

```
typedef struct_IMAGE-NT_HEADERS{
DWORD Signature
IMAGE-FILE-HEADER FileHeader;
IMAGE-OPTIONAL-HEADER Optional-Header;
}IMAGE-NT-HEADERS,*PIMAGE-NT-HEADERS;
```

该结构由一个双字标志项 DWORD Signature 和两个子结构项：

```
IMAGE-FILE-HEADER FileHeader
IMAGE-OPTIONAL-HEADER Optional Header
```

组成。其各部分的功能如下：

1. 双字标志项

DWORD Signature 双字标志项是文件格式的标识符，表明带有该文件首部的文件采用了何种文件格式。

如果根据 WINSTUB 中的 e-lfanew 找到的标志是“NE”，而不是“PE\0\0”，这说明该文件是 NE 格式的。同样，如果为 LE 标志，则指出该程序是 Windows3.x 的设备驱动程序，如果为 LX 标志则表示为 OS/2 的执行文件。

只有标志项为“PE\0\0”时才为 Windows NT 文件格式。

2. PE 文件首部的第一个子结构 IMAGE-FILE-HEADER

这是 PE 文件首部的第二部分，其形式如下：

```
Typedef struct IMAGE-FILE-HEADER{
WORD Machine;
WORD NumberofSections;
DWORD TimeDateStamp;
DWORD PointerToSymbolTable;
DWORD NumberofSymbols;
WORD SizeofOptionalHeader;
WORD Characteristics;
}IMAGE FILE HEAD 为 ER,*PIMAGE-FILE-HEADER;
```

这个结构紧跟在标志项之后，结合 Windows NT 中的 SETUP.EXE 实例，可以看出该结构中的信息主要特征为：

① Windows NT 的可执行文件所适应的 CPU，包括 386、486、586、MIPS、R3000、R4000、DEC Alpha、AXP 等。

② 指明了有关重定位项信息，即文件中有无重定位项，文件是可执行文件(0x0002)，还是动态连接库(0x2000)等。

3. PE 文件首部的第二个子结构 IMAGEAGE-OPTIONAL HEADER

这是 PE 文件首部的第三部分，对于 PE 文件而言，这一部分是必不可少的。其数据结构如下：

```
Typedef struct IMAGE-OPTIONAL-HEADER {
//
//Standard fields.
//
WORD    Magic;
BYTE    MajorLinkerVersion;
BYTE    MinorLinkerVersion;
DWORD    SizeofCode;
DWORD    SizeoflnitializedData;
DWORD    SizefUninitializedData;
DWORD    AddressofEntryPoint;
DWORD    BaseofCode;
DWIRD    BaseofData;
//
```

```
//NT additional fields,
//
DWORD   ImageBase;
DWORD   SectionAlignment;
DWORD   FileAlignment;
WORD    MajorOperating System Version;
WORD    MinorOperating System Version;
WORD    MajorImageVersion;
WORD    MinorImageVersion;
WORD    MajorSubsystem Version;
WORD    MinorSubsystemVersion;
DWORD   Reservedl;
DWORD   Sizeoflmage;
DWORD   SizeofHeaders;
DWORD   CheckSum;
WORD    Subsystem;
WORD    DIICharacteristics;
DWORD   SizeofStackReserve;
DWORD   SizeofHeapCommit;
DWORD   SizeofHeapReserve;
DWORD   SizeofHeapCommit;
DWORD   LoaderFlags;
DWORD   NumberofRva AndSizes;
IMAGE-DATA-DIRECTORY
Data Directory[IMAGE-NUMBEROF-DIRECTORY-ENTRIES];
{IMAGE-OPTIONL-HEADER,*PIMAGE-OP-TIONAL-HEADER;
```

这个子结构是 PE 文件头中的非可选项，也是 PE 文件首部中最重要的部分，它定义有关.EXE 文件的装载、运行、支持环境、状态等关键信息。下面结合 Windows NT 中的 SETUP.EXE，介绍这些信息的内涵。

(1) 有关.EXE 文件块的信息

① .EXE 文件代码的扇区数(即.text 块数)和代码块的起始地址。装载程序开始执行的地址(RVA)一般在.text 块中。RVA(Relative Virtual Address)为相对虚拟地址，即 PE 格式文件中的某个项相对于文件内存映像地址的偏移量(如虚拟地址 0x10000，映像地址 0x10464，则 RVA 为 0x0464，显然虚拟地址加上 RVA 便得到模块的映像地址或基地址)，可见 PE 格式中可执行文件的任何偏移地址都是直接给出的。

② .EXE 直接映像的内存地址。文件一旦被装入到这个地址，在运行之前代码就不需要变动，在 Windows NT 生成的可执行文件中，该内存地址缺省值为 0x10000；对于 DLLs，缺省值为 0x400000。在 Chicago 中，不能用来装入 32 位可执行文件，因为该地址处于进程共享的线性地址区域，故 Microsoft 将 Win32 可执行文件的缺省基地址改变为 0x400000。这对于以 0x10000 为映像地址的老程序，在 Chicago 中装入时无疑要花去的时间会长一些，因为装载程序需对它进行重定位。

③ 装入.EXE 文件总长度(除.text 外, 即从 data 块到文件最后)。此外还有 PE 文件头中数据目录中的数据项数(流行工具软件为 16 项)，进、出口函数表的地址及长度。

(2) 版本及 DLL 调用状态信息

① OS 版本号，生成 .EXE 文件的连接器版本号，.EXE 自定义版本号，运行.EXE 文件的所需子系统版本号。

② .EXE 文件执行用户界面使用的字符类型，包括 GUI、Windows 字符系统，或 OS/2 字符系统或 POSIX 字符系统。

③ 初始函数 DLL 调用的状态信息，包括为 DLL 在首次被装入一个进程地址空间的调用信息，在一个线程建立或终止时的调用信息，或 DLL 退出时的调用信息。

此外还有为提高可靠性而设置的文件 CRC 校验和。

④ 对组成块的数据要求。按 PE 文件格式组成的.EXE 文件的各类块的数据量必须是某一特定数据(扇区字数)的倍数。这样做的目的是为了确保文件的每一块均从磁盘扇区的开头存放(类似于 NE 文件格式中的对齐因子做法)，PE 格式块的数量不像 NE 格式中的段数量那么大，这样可减少存放空间的浪费。

(3) 有关装载信息和计算装载模块大小以及内存空间的信息

① 装载.EXE 文件各部分(数据块到最后一个块)所需空间；

② .EXE 文件首部及块的大小，预留、初始化堆栈内存大小(缺省时为 1MB)；

③ 初始化堆栈的大小(缺省为 1 页)；

④ 初始化进程队列的虚拟内存大小；

⑤ 进程队列中，内存大小的初始设定(缺省值为 1 页)。

8.5.3 块表数据结构及辅助信息块

1. 块表与信息块

由 PE 文件格式可知，在 PE 首部与原始信息块之间存在一个块表，块表包含了每个信息块在映像中的信息，信息块在映像中是按起始地址 RVA 顺序排列。

操作系统管理这些信息块是通过块表来进行的，而块表这一数据结构是以 IMAGE-SECTION-HEADER 数组的形式存在于系统之中，该数据的大小，在 PE 文件首部的第二个子结构 IMAGE-SECTION-HEADER 的 DWORD Section Alignment 字段中已给出了。

2. 块表的数据结构

PE 可执行文件的块表是一自定义的结构体类型。结构体“IMAGE-SECTION-HEADER”由一共同体 Misc 和若干个体内分量(域表)组成，具体形式如下：

```
typedef struct-IMAGE-SECTION-HEADER
BYTE Name[IMAGE-SIZEOF-SHORT-NAME];
union {
DWORD   PhysicalAddress;
DWORD   VirtualSize
Misc;
DWORD   VirtualAddress;
DWORD   SizeofRawData;
DWORD   PointerToRawData;
DWORD   PointetTorRelocations;
DWORD   PointerToLinenumbers;
WORD    NumberofRelocations;
WORD    NumberofLinenumbers;
DWORD   Characteristics;
} IMAGE-SECTION-HEADER,*PIMAGE-SECTION-HEADER;
```

它包含了 3 方面的内容：

(1) 表征了块表所指向块的块名

块名是一个 8 位的 ANSI 名，大多数块名以“.”开头，例如，

.text　　文本块名；

.data　　外来 DLL 函数信息块名；

.rdata　　初始化数据块名；

.rsrc　　资源块名；

.reloc　　重定位块名(当块名超过 8 字节时，无终止符 NULL 标志)。

PE 文件格式对块命名的要求，不同于汇编语言中直接用段命名，也不同于 C++ 编译器中用“#pragma dtat-seg”或“pragma cade-seg”给块命名。

(2) 大小、地址、指针及相关数据

上述块表中有两个长度 Virtual size 与 size of Raw Data，这两者是有区别的。前者保存的是代码或数据的真实长度，这个长度是对齐以前的长度，后者是代码或数据(行)经调整后的值。

有关块的地址，块表中用 DWORD VirtualAddress 字段表示，它说明了装入该块的 RVA 地址。若要计算出一个给定块的内存中的起始地址，则应将该块的

VirtualAddress 加上文件映像的基地址。在 Microsoft 工具中，第一个块的 RVA 缺省值为 0x1000。

此外，PE 可执行文件的块表中有 3 个指针：DWORD PointerToRawData、DWORD PointerToRelocations 和 DWORD pointerToLinenumbers。

第一个指针定义为原始数据指针，表示程序经编译或汇编生成的原始数据在文件中的偏移。

第二个指针在 PE 可执行文件中无意义，被设置为 0。因为当连接器生成可执行文件时，已经将许多块连接起来了，只剩下重定位的基地址和输入函数(Imported Function)需要在装入时决定。而在 OBJ 文件中，该指针是块的重定信息在文件中的偏移。

第三个指针是行号表在文件中的偏移，行号表将源文件中的行号与该行生成的代码地址联系起来，现代的 debug 格式中，如 Codeview 格式，行号信息是作为 debug 信息的一部分存储的。

(3) 有关块属性的标志

在 PE/COFF 格式中，DWORD Characteristics 字段表明了块属性(如数据的可读、可写等特征)，即它定义了属性的代码。

PE 文件格式中一些重要块标志可分析归纳为以下几种情况：

0x00000020　包含代码块。

0x00000040　包含已初始化的数据块。

0x00000080　包含未初始化数据(块如.bass)。

0x00000200　包含注解及其他一些类型信息块。

0x00000800　内容不能放入最终的可执行文件中的块。

0x02000000　可被丢弃块(因为它一旦被装入之后，进程就不再需要它。常见的可丢弃块为基地址重定位.reloc)。

0x10000000　共享块 (如果在 DLL 中设置这类共享块，则表示其中的数据可被所有使用该 DLL 的进程共享。数据块的缺省设置为不能共享，即每一个进程使用 DLL 时，都要有单独的数据拷贝)。

另外，共享块可使内存管理器为该块设置页面映像，以使所有进程使用 DLL 时均指向内存中的同一物理页。为实现块的共享，在连接的时候使用 SHARED 属性。

0x20000000　可执行块。通常当块设置 0x0000020 标志时，也设置该标志。

0x40000000　可读块。可执行文件中的块一般都设置该标志。

0x80000000　可写块。.data 和.bass 设置该标志，如果可执行文件中的块没有设置该标志，则装载程序就会将内存映像页标记为只读或执行。

3. 主要块的调用机制

下面给出.text, .reloc, .rdata, .data, .bss, .rsrc 等常见块的描述，重点分析代码

块.text 及重定位块.reloc 块的运行机制。

(1) .text 块

代码.text 是在编译或汇编结束时产生的一种代码块。由于 PE 文件运行在 32 位方式下，不受 16 位段的约束，因此，不同文件产生的代码不再分成单独的块，而链接器将所有目标文件的代码块连接成一个大的.text 块。如果使用 Borland C++，则其编译器将编译产生的代码存于名为 CODE 区域中，产生的 PE 文件的代码块不称.text，而直接命名为 CODE 块。

PE 可执行文件的调用机制不同于一般可执行文件(如 NE 可执行文件)的调用方法。PE 执行文件的调用方法有其独到之处，在 PE 文件中，当其他模块调用动态链接库中的某一功能(如 USER32.DLL 中的 GetMessage)时，编译出的 CALL 指令不能直接调用 DLL 中的功能，而是转换成控制指令：

JMP DWORD PTR [XXXXXXXX]

或称微码(thunk)。并将其存于.text 块中，而 JMP 指令通过双字变量 PTR 的值间接进入相邻的.rdata 块中，在.rdata 块的双字中包含了操作系统的某些功能函数的真实目标地址，装载程序依据此目标函数的地址，并将该函数插补到执行映像中，所需信息均存放在.rdata 块中，如图 8.22 所示。

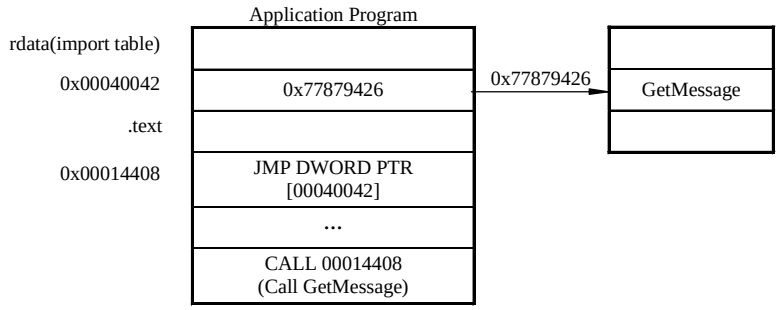


图 8.22　.text 与 DLL

由图 8.22 可知，当 PE 文件中多处调用某一特定 DLL 功能时，装载程序不必逐一修改其他调用指令，而只需将所有被调用目标功能地址正确放入.rdata 块中即可。这与 NE 文件中的方式明显不同，在 NE 文件中，每一个段中都包含一个重定位表，如果在该段中调用一个 DLL 功能 20 次，装载程序就必须将这一功能的地址在该段中重复写 20 次。一般来说在 Microsoft 系统中，当连接外部 DLL 库函数时，库管理程序 LIB32 生成输入库以及微码(连接程序无需知道微码的生成)。JMP DWORD PTR [XXXXXXX]，再通过输入库的.text 块进入可执行文件，可见输入库实际上是连接到 PE 文件的另一些代码及数据。

(2) .reloc 块

.reloc 块保存基地址重定位表(简称为重定位表)，它的作用主要体现在装载程序不能按链接器所指定的地址装载文件，需对文件中指令或已初始化的变量值进行调整(或者说进行重定位)时，就必须用到某一基地址重定位表，此表包含了调整所需要的数据(当然，在装载程序能够正常装载文件时，.reloc 块中的重定位表无疑可忽略，如果希望装载程序总能按链接器假定的地址装入，则可通过\FIXED 选择项跳过该块信息)。

值得注意的是：

① 这样做的目的是为节省执行文件的空间，但可能导致该执行文件在其他 Win32 系统中不能执行。如 Windows NT 下的可执行文件的基地址是 0x10000,如通过 FIXED 而选择，通知链接器去掉重定位(即.reloc 块信息)，则执行文件不能在 chicago 系统下运行，因 chicago 中的地址 0x10000 已被其他程序选用，因此，在此情况下不应去掉.reloc 块。

② 编译器生成的 JMP 和 CALL 指令使用的是相对于该指令的偏移地址，而不同于 32 位段中的真实偏移地址。如果要把包含这些指定的文件装到某个非链接器指定的地址时，无疑这些指令无需改变(因为它们使用的是相对地址)，无需重定位信息，只有那些真正使用了 32 位偏移地址数据的指令才需要重定位，才涉及到.reloc 块。如定义全程变量：

int i；int *ptr=&i；

若链接器假定的基地址是 0x10000，变量 i 的地址为 0x12004,那么链接器会在内存中“*ptr”处写入 0x12004。如果由于某种原因，装载程序将该文件从基地址 0x70000 处装入，那么，i 的地址就变成 0x72004。.reloc 块中存入链接器假定的装入地址和实际的装入地址之间的差值。

(3) 其他数据块

其他数据块包括.data 块、.bss 块及.rsrc 块。

同.text 块默认为代码块一样，.data 块为初始数据，这些数据包括编译时被初始化的 global 全程变量和 static 固定变量，还包括文字串等，链接器将 OBJ 及 LIB 文件的.data 块结合成一个大的.data 块，局部变量 Local 放在一个线程的堆栈中，不占用.data 块或.bss 块的空间。

.bss 块存放未初始化的全程变量 global 和固定变量 static，同样，链接器将 OBJ 和 LIB 文件中的.bss 块结合成一个大的.bss 块，.bss 块的原始系统数据偏移字段被置为 0，表明此块在文件中不占空间，TLINK 不产生此块，只扩展.data 块虚拟长度。

.rsrc 块是一个功能单一的模块，包含了块的全部资源。

.rdata 块包含了其他外来 DLL 的函数及数据的信息。

.edata 块是 PE 文件输出函数和数据的列表，以供其他模块引用。它与 NE 文件

的入口表、驻留名表及非驻留名表的综合功能等价。

8.6 小　结

本章主要讨论了文件、文件系统的基本概念。文件是一组赋名的相关联的字符流的集合或者说文件是相关联的记录的集合。文件的逻辑结构有无结构的字符流式和有结构的记录式两种。文件的物理结构是文件在外存上的存储组织方式，基本类型有连续文件、链接文件和索引文件。从逻辑结构到物理结构的映射是复杂的。

文件系统是 OS 重要组成部分，它实现按名存取文件、文件存储空间的分配与回收、文件及其目录的管理，提供用户与 OS 接口。

文件目录是用来组织文件和检索文件的关键数据结构。每个目录项记录了一个文件的说明和控制信息。树型（层次）目录是使用最广的目录结构，它能有效地提高对目录的检索速度，允许文件重名，便于实现文件共享。

本章还介绍了文件的存储空间分配与回收的方法。

本章最后给出了 Windows NT 可执行文件的结构、文件头、信息块、辅助信息块等，分析了可执行文件常用块的调用机制。

习　题　八

8.1　什么是文件、文件系统？文件系统的功能是什么？

8.2　文件按其用途和性质可分成几类，各有何特点？

8.3　文件的逻辑结构有几种形式？

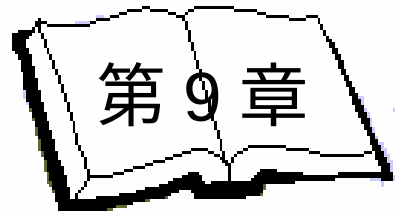
8.4　常见的文件的物理结构有几种？它与文件的存取方式有什么关系？

8.5　什么是文件目录？文件目录中包含哪些信息？

8.6　二级目录和多级目录的好处是什么？

8.7　常用的文件空间分配方法有哪些？试分别加以说明。

8.8　Windows NT可执行文件的结构是怎样的？



分布计算

随着微型计算机和个人计算机的计算能力不断提高、价格不断下降，已出现了分布数据计算(Distributed Data Processing，简称 DDP)的趋势。DDP 使处理器、数据和数据计算系统的其他方面分散到一些结构中。分散可以使系统更好地满足用户需要，提供更短的响应时间，而且与集中式方法相比减小了通信开销。一个 DDP 系统不仅包括计算函数，而且也包含数据库、设备控制、交互(网络)控制的分布式结构。

许多结构都主要依赖于个人计算机对数据的集中计算。个人计算机用于支持大量用户友好界面的应用，如字处理和图形处理，此外，还包括数据库和用于数据库管理或信息系统的复杂软件。

操作系统和支撑软件的分布式能力的不断提高，推动了相关应用的发展：

① 通信体系结构。它支持单独的计算机网络软件，支持分布式应用，如电子信函、文件传输和远程终端访问。每台计算机都有其自己的操作系统。只要所有机器支持同一通信协议，就有可能组成计算机和操作系统的异构环境。

② 网络操作系统。这是由客户机所组成的网络配置。网络通常由单用户工作站和一个或多个“服务器”构成。服务器提供网络中的服务，如文件存储和打印机管理。网络操作系统是对本地操作系统的一个简单连接，使得客户机和服务器可以相互作用。用户清楚地知道组成网络的各种设备，而且用显示的方式使用它们。

③ 分布式操作系统。这是为分布式计算机系统(多机系统)配置的一个操作系统。用户把它看成一个普通的集中式操作系统，它允许用户透明地访问多个机器上的资源。

第一个用于商业的通信体系结构是，IBM 在 1974 年提出的系统网络体系结构(System Network Architecture，即 SNA)。网络操作系统发展较晚，但已有大量商业产品涌现出来。分布式系统的研究和开发中的前沿领域是分布式操作系统。虽然已有一些商用系统，但完整功能的分布式操作系统仍在实验阶段。

本章主要介绍分布计算的两个重要内容：客户/服务器结构和分布进程通信。

9.1　客户/服务器计算

近年来，用于信息系统中典型的计算机模型是客户/服务器计算模型。这个计算模型快速代替了占优势的主机集中计算模式和分布式数据计算模式。

9.1.1　什么是客户/服务器计算

表9.1给出了客户/服务器(Client/Server)计算的一些定义，图9.1是进一步的说明。一个客户/服务器环境由客户和服务器组成。客户机常常是单用户PC或是工作站，它们可以为终端用户提供友好的用户界面。基于客户的工作站使用了图形界面，十分方便。

表 9.1　客户/服务器计算的定义

名　称	定　义
客户/服务器计算	① 任何满足如下条件的应用可称之为客户/服务器计算：动作的请求者在一个系统，而动作的提供者可能在另一个系统中。而且，大多数客户/服务器设计方案是多对一的，即多个客户对一个服务器提出请求。 ② 将一个应用分成一些任务，将每个任务分配到可以最有效执行的平台上。处理的表示保存在用户机(客户)上，而数据的管理则保存在服务器上。所有的数据处理可以全部在客户上，也可以分到客户和服务器上，这取决于应用和使用的软件。服务器通过网络与客户连接。服务器软件接收来自客户软件对数据的请求，返回结果给客户。客户对数据进行操作，显示结果给用户。
客户/服务器计算模型	由一个客户或请求者，提交一个协同处理的请求，服务器处理这个请求并返回结果给客户。在此模型中，应用的处理被分散到(并不是必须地)客户和服务器上，由客户初始化，而且部分由客户控制，并不是主从式，相反，客户和服务器协作执行一个应用。
客户/服务器	① 并行执行的软件进程之间的一种交互模型。客户进程发送请求给一个服务器进程，服务器进程返回这些请求的结果。服务器进程按它们特有的处理方式给客户提供服务，使客户进程得以从复杂事务中解脱出来，完成其他有用的工作。客户进程和服务器进程之间的交互是协作式的事务交换，其中客户是主动的，服务器是被动的。 ② 一种满足如下条件的网络环境：服务器节点控制数据，其他节点能够访问数据，但不能更新数据。

客户/服务器环境中的每个服务器为客户提供一组共享用户设备。现在，最普通的服务器是数据库服务器，通常用来控制关系数据库。服务器使得客户能够共享访问同一个数据库，使数据库管理的性能更高。

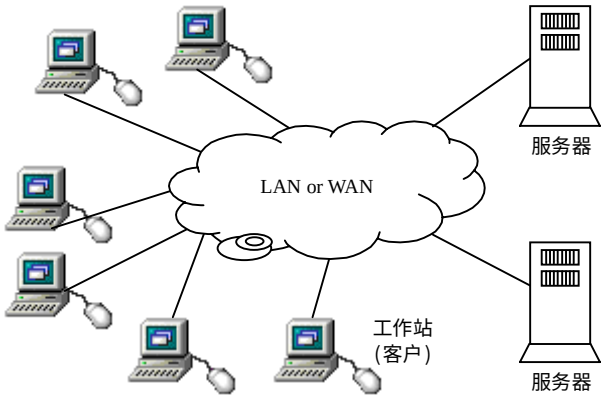


图 9.1　客户/服务器环境

除了客户和服务器外，第三个不可缺少的部分是网络。客户/服务器计算是一种分布计算。用户、应用和资源根据商业需求是分布式的，它们通过LAN、WAN，或是Internet网络连接起来。

一个客户/服务器模型和其他分布计算方式的不同之处如下：

① 用户十分依赖于自己系统中的友好应用。这使得在客户/服务器配置中，用户要对应用的执行时间和使用方式进行控制。

② 在应用分散的同时，十分强调集中共享数据库和一些网络管理的功能。

③ 网络是操作的基础。因此，网络管理和网络安全在组织、设计和使用信息系统中有着十分重要的地位。

9.1.2　客户/服务器模式的应用

客户/服务器体系结构的特征是应用级任务在客户和服务器之间分配(见图9.2)。当然，在客户和服务器中，最基本的软件是运行于硬件平台上的操作系统。应用与服务器的平台和操作系统是不同的。事实上，在一个环境中，平台和操作系统有许多种。只要客户和服务器在通信协议上相同，支持相同的应用，那么，这些低层的差异并不重要。

正是通信软件使客户和服务器得以相互作用。这种软件有TCP/IP、OSI、SNA等。

这些支撑软件构成分布式应用的基础。理想情况下，应用的功能分散到客户和服务器上，优化了平台和网络，同时优化了用户执行不同任务的性能和共享资源的能力。

一个成功的客户/服务器环境的关键因素是用户如何和系统一体化交互。因此，客户机上用户界面的设计是十分重要的。在大多数客户/服务器系统中，十分强调提供一个图形用户界面(Graphical User Interface，简称为GUI)。这个用户界面易用易学、

功能强大而且伸展性很强。

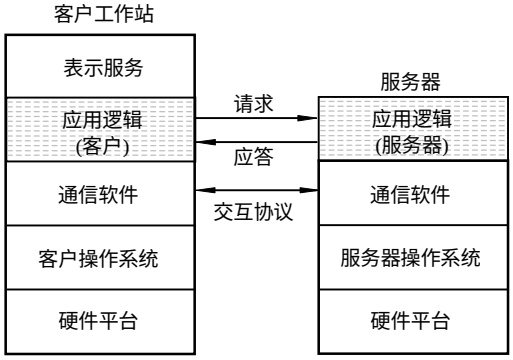


图 9.2　客户/服务器体系结构

1. 客户/服务器应用举例

下面通过一个例子来说明使用关系数据库的客户/服务器应用。在这个环境中，服务器实质上是数据库服务器，客户在事务中发出一个数据库请求，接受数据库响应。

图9.3表明这样一个系统的结构：服务器上有一个复杂的数据库管理系统软件模块，负责维护数据库，客户机上可有多种不同的应用，用软件(如结构化查询语言(SQL))使两者连接在一起。

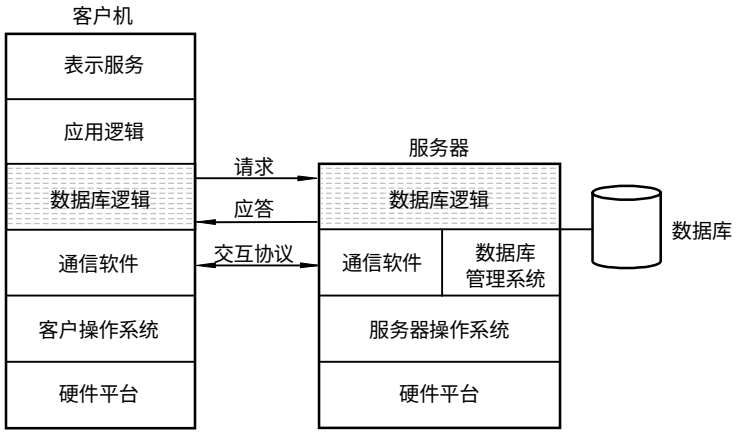


图 9.3　基于客户/服务器体系结构的数据库应用

图9.3表明所有应用逻辑在客户端，而服务器只关心管理数据库。这种配置的正确性依赖于应用的风格和用途。例如，假设主要目的是为记录查询提供在线访问，图9.4(a)显示其工作的方式(设服务器装有1 000 000条记录的数据库，用户想通过一些查询条件来查找记录)。

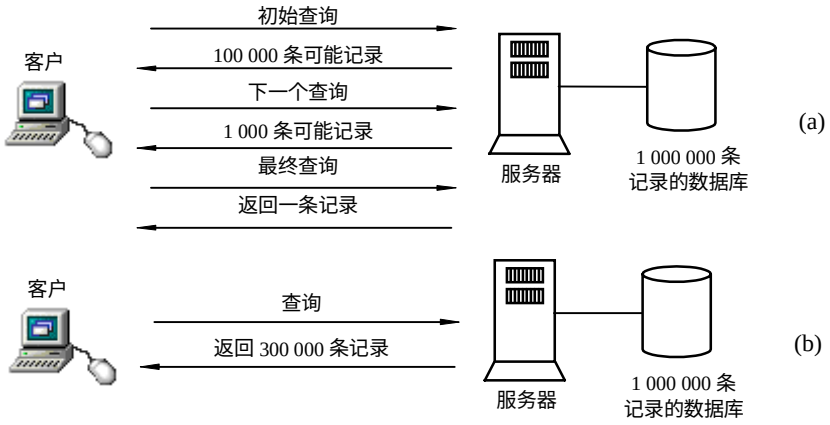


图 9.4　客户/服务器的使用

(a) 正确的客户/服务器使用　(b) 不正确的客户/服务器使用

上述的应用适用于客户/服务器结构，有两个原因：

① 有大量数据库的排序和查询工作。这要求一个大容量的磁盘，一个高速CPU和高速I/O接口。对于单用户工作站和PC而言，这种需求是不必要的，也过于昂贵。

② 不会在网络上有大量的通信。

如图9.4(b)所示，这是一种不正确的客户/服务器使用，一个查询请求产生了300 000条记录的传输，通信的数据量太大。

2. 客户/服务器应用分类

图9.5给出了一些不同的客户/服务器应用。

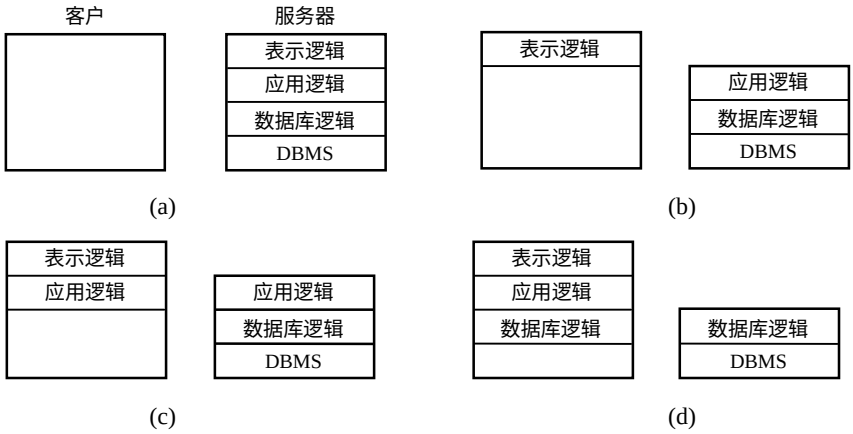


图 9.5　客户/服务器应用分类

(a) 基于主机的处理　(b) 基于服务器的处理　(c) 协同处理　(d) 基于客户的处理

① 基于主机的处理。这种处理不是真正的客户/服务器计算，它更像传统的主机环境，处理几乎都在主机上，用户界面只是无声终端。

② 基于服务器的处理。这是最基本的客户/服务器配置方式，是服务器负责提供图形用户接口，所有的处理都在服务器上完成。

③ 基于客户的处理。其所有应用处理都在客户机上，数据路由和数据库逻辑函数在服务器上。这种结构是使用最普遍的客户/服务器方式。

④ 协同处理。在协同处理配置中，充分利用客户、服务器和数据的分布性，使应用处理得以优化。这种配置比较复杂，但长远来看，这种配置可以提供更高的用户生产率和更高的网络效率。

当然，数据和应用处理的分布取决于数据库信息的特点、要支持应用的类型，等等。

3. 客户/服务器应用要求——文件高速缓存一致性

当使用一个文件服务器时，文件I/O的性能显著低于本地文件访问的水平，这是网络导致的延迟。为了减少性能损失，要使用文件高速缓存来保存近来访问的文件记录。因为局部性，一个本地文件高速缓存的使用可以减少远程服务器访问的数量。

图9.6说明了一个典型使用文件高速缓存的分布式机制。当一个进程产生一个文件访问请求时，先将请求传给进程工作站的高速缓存。如果不能满足，要么将请求传给文件存放的本地磁盘，要么传给存放文件的文件服务器。

如果高速缓存经常包含远程数据的备份，则称高速缓存是具有一致性的。当远程数据改变而对应的本地高速缓冲备份没有丢弃时，就有可能出现高速缓存不一致的情况。如果一个客户改变了一个文件而其他客户又使用这个文件，就有可能出现这个情况。其中的问题有两个：如果客户在改变后立即将其写到服务器的文件中去，那么拥有这个文件的高速缓存备份的相关客户的所有数据将过时；如果客户延缓写回到服务器，则问题会更糟，这种情况下，服务器的文件都是过时的，新的读文件请求会读到过时的数据。保证本地高速缓存中的备份跟上远程数据改变的问题，称为高速缓存一致性(Cache Consistency)问题。

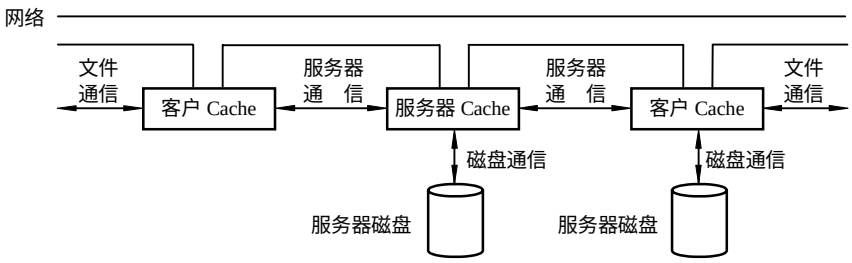


图 9.6　分布式文件高速缓存

最简单的解决高速缓存一致性的方法是使用文件上锁技术，以防止多个客户同时访问一个文件。这是以性能和扩展性为代价得到的一致性。一个更好的方法是：可以有任意数目的远程进程打开一个文件来读，生成自己的客户高速缓存，但当一个进程打开文件请求写访问，而其他进程读这个文件时，服务器采取两个步骤，首先，通知写进程，虽然它有一个高速缓存，但必须在更新之后立即写回所有改变了的块；其次，通知所有的读进程，文件不再能被高速缓存。

9.1.3　中间件

为了真正从客户/服务器方式中受益，开发者必须开发一套工具，以提供可以针对所有平台而言都是一致的、访问系统资源的方法和风格。这样，编程人员可以不再考虑各种PC机或是工作站之间的差异，而且不管数据存放在何处都可以使用一样的访问方法。

最普通的实现方法是通过在应用层之上、通信软件和操作系统层之下，使用标准的编程接口和协议，这就是所谓的中间件(Middleware)。中间件有许多种，从相当简单到十分复杂，但有一个共同点：它们都掩藏了不同网络协议和操作系统之间的复杂细节和差异。

图9.7描述了一个客户/服务器结构中中间件的作用。它的作用与其所在的环境十分相关。

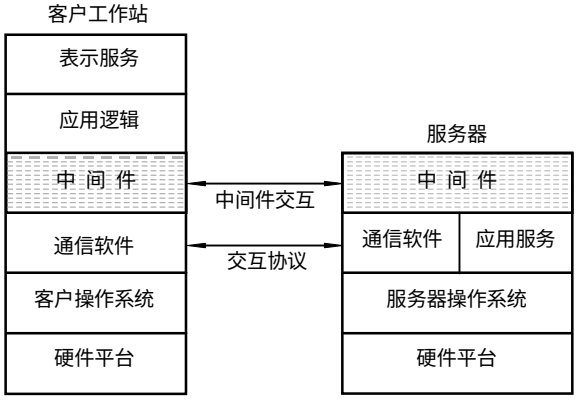


图 9.7　客户/服务器结构中中间件的作用

注意：中间件有两个，一个在客户端，一个在服务器端。中间件的基本目的是使客户端的应用程序或用户，不用关心服务器的差异，直接去请求服务器的服务。例如，SQL提供一个标准化的方法，使本地或远程用户或应用程序可以用SQL访问一个关系数据库。然而，许多关系数据库厂家，虽然也支持SQL，但在SQL中增加了他们自己的“方言”，从而使该厂家的产品区别于其他厂家的产品，同时也产生了不兼容的问题。

假设有一个用于人事部门的分布式系统。其基本雇员数据，如雇员的姓名、地址等都存放在Gupta数据库中；而工资信息存放在Oracle数据库中。当一个用户要对记录进行访问时，用户无须关心记录存放在哪个数据库中。中间件就提供了这样一个软件层，使得对不同系统有一种规范的访问方法。

图9.8从逻辑上而不是从实现上描述了中间件的作用。中间件使分布式客户/服务器计算成为可能。这样，用户就可以将整个分布式系统看成一组可用的应用和资源，不必关心数据的位置或应用的位置。所有的应用操作都建立在一个规范的应用编程接口上(Application Programming Interface，即API)。中间件负责把客户请求路由到正确的服务器上去。

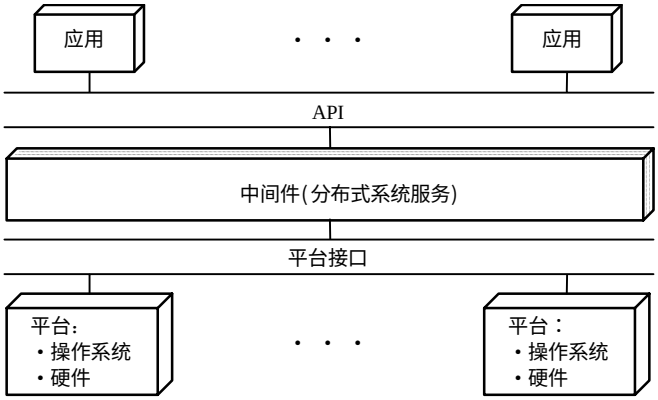


图 9.8　中间件的逻辑视图

图9.9为应用示例。例中，中间件用于实现网络和操作系统的兼容性。一

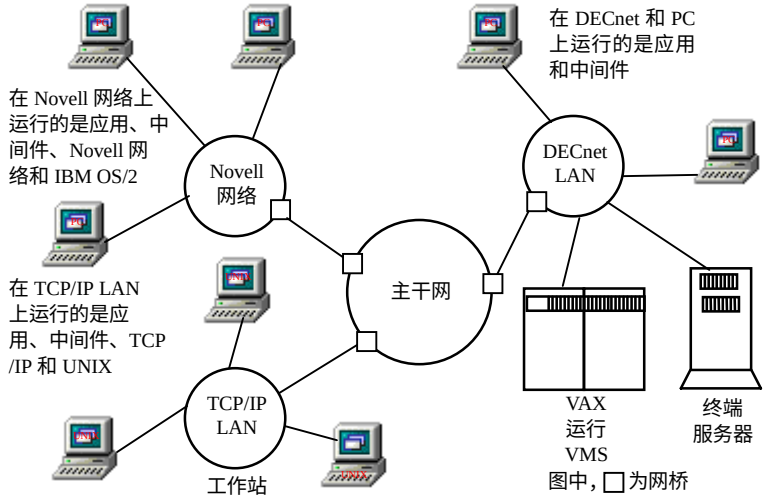


图 9.9　中间件应用举例

个主干网连接DECnet、Novell和TCP/IP网络。中间件保证了所有网络用户对这3个网上的任何应用和资源都可以透明地访问。

虽然，中间件产品有许多种，但这些产品都基于两种机制：消息传递和远程过程调用。

9.2　分布式消息传递

在一个分布式计算系统中计算机并不共享存储器，因此，在这类系统中使用消息传递机制。

9.2.1　分布式消息传递的方法

图9.10所示的是一个最普通的用于分布式消息传送的模型，称为客户/服务器模型。一个客户进程请求一些服务(如读文件、打印)，向服务器进程发送一个包含请求的消息。服务器进程完成请求，做出回答。最简单的形式中，只需要两个函数：发送(Send)和接收(Receive)。Send函数说明发送的目标和消息内容；Receive函数说明消息的来源，为消息的存储提供一个缓冲区。

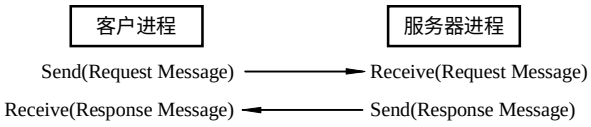


图 9.10　分布式消息传递图

图9.11说明了消息传递的一种实现方法。进程使用消息传递模块，这是操作系统的一部分。服务请求用原语和参数来表示。原语(primitive)说明要执行的函数、传递数据和控制信息的参数。原语的实际形式取决于操作系统，它可以是一个过程调用，或者本身也是一个消息。

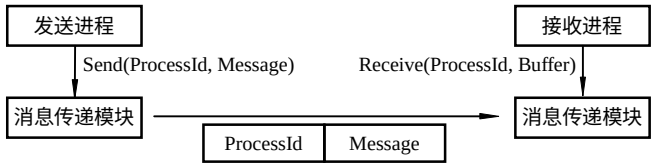


图 9.11　基本的消息传递原语

Send原语用来发送消息。它的参数是目标进程的标识符和消息的内容。消息传递模块构造了一个数据单元，其中包括进程标识符和消息内容这两个元素。这个数据单元通过一些通信设备，如OSI通信结构，发送到目标进程所在的机器。当数据单元被目标系统接收时，它被通信设备路由到消息传递模块。此模块检查进程的ID域

并将消息存到相应的缓冲区中。

接收进程通过Receive原语通知消息传送模块，它想接收消息。消息传送模块在收到一个消息时，向目标进程发出某种接收信号，并将接收到的消息存放到缓冲区中。

9.2.2　消息传递的可靠性

一个可靠的消息传递方式是尽可能地保证传送，能使用一个可靠的传输协议或相似的逻辑，能进行检错、确认、重传和重排序。因为传送受到保证，所以发送进程就不必知道消息是否已被传送了。然而，给发送进程一个确认是十分有用的，如果传送不能完成(如网络出错或系统崩溃)，则发送进程会注意到这一点。

另一方面，消息传送可以只简单地把消息传送到通信网络上，并不报告成功或是失败。这样做大大减小了复杂度，以及处理和通信的花费。对于那些需要确认的应用程序或用户，它们可以用Reply(回答)消息来实现这一点。

9.3　远程过程调用

远程过程调用(Remote Procedure Call，即RPC)是现在广为应用的分布式系统中的一种通信技术。这种技术允许不同机器上的程序通过简单的过程调用/返回算法来相互作用，就好像这两个程序在同一台机器上一样。它具有如下优点：

① 过程调用已经广为接受、使用和理解。

② 远程过程调用使远程界面成为一组特定类型的操作。因此，界面可以很清楚地描述，分布式系统也易于检查类型错误。

③ 因为说明了一个标准化和精确定义的界面，所以，应用程序的通信代码可以自动生成。

④ 开发者能够编写出客户方和服务器方的模块，这些模块无须改动和重新编码就能在不同计算机和操作系统上运行。

远程过程调用可以看成是一个可靠的分块消息传送。图9.12所示是总体结构，

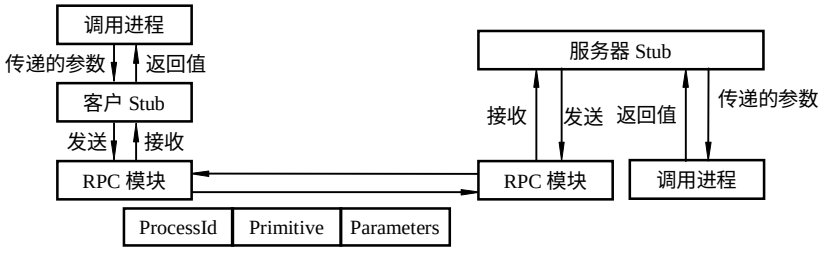


图 9.12　远程过程调用机制

图9.13是其内部程序逻辑。调用程序用自己机器上的参数进行一个普通的过程调用，即

CALL　P(X，Y)

其中，P为过程名；X为传送的参数；Y为返回值。

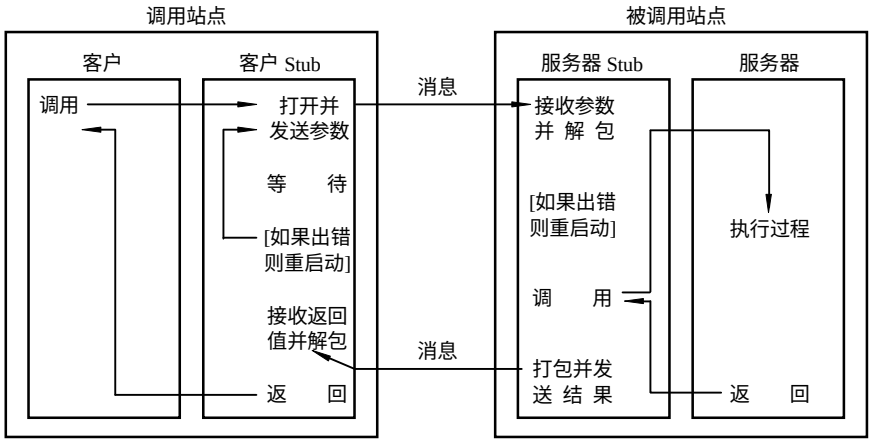


图 9.13　远程过程调用逻辑

对用户而言，这个对其他机器上的远程过程调用可以是透明或不透明的。过程P要包括调用者的地址空间或者在调用时动态链接它。此过程会生成一条消息，标识正在调用的过程和参数。它会用Send原语发送这条消息，并等待回答。当回答收到时，过程返回调用程序，提供返回值。

在远程机构中，另一个程序与调用过程相关联。当消息到达时，它进行检查，并且生成一个本地的CALL P(X，Y)。这个远程过程在本地调用，它的参数和堆栈状态等都和本地过程调用一样处理。

1. 参数传递

大多数编程语言允许参数以值的方式传递(传值：call by value)，或是以一个指向地址的指针来传递(传地址：call by reference)。传值对一个远程过程调用来说是简单的，参数被简单地备份到消息中，发送给远程系统。相比之下，传地址则难于实现。它需要每个对象都有一个系统惟一的指针，开销较大。

2. 参数表示

如果调用和被调用程序所在机器是同构的，或是在相同操作系统上用相同编程语言编写的，则不会产生任何问题。但如果这些方面存在差异，那么，在数字和文本的表示上就会存在差异。最好的解决方法是提供一个标准的形式，如整数、浮点数、字符或字符串。这样任何机器上的本地参数都可以转化成标准表示或从标准表示中得到。

9.4 小　结

目前，计算机已不再总是单独工作，大多已成为计算机网络的一部分。为了支持分布式系统，已开发出了一系列设备，如通信体系结构、网络操作系统和分布式操作系统。

通信体系结构是硬件和软件的构造集合，用来支持系统间的数据交换和分布式应用。开放系统互连(OSI)模型是为异构计算机协同工作而设计的标准化通信结构。OSI模型有7个功能层，每个功能层都开发出了实现标准。最广泛使用的通信结构是TCP/IP协议。

网络操作系统实际上不是操作系统，而是在网络上支持服务器的一个分布式系统软件集。服务器提供网络中的服务或应用，如存储和打印机管理。每个计算机有它自己的操作系统。网络操作系统只是对本地机操作系统的一个简单连接，以允许机器和服务器交互。通常，一个普遍通信结构可支持这些网络应用。

分布式操作系统是被一个计算机网络共享的操作系统。对于用户，像一个普通的集中式操作系统，但它为用户提供了对若干机器资源的透明访问。一个分布式操作系统可以依赖于具有基本通信功能的通信结构。

客户/服务器模式是实现信息系统和网络潜力的重要技术之一，它能够显著改善系统的生产率。客户/服务器计算中，其应用被分散到单个用户工作站和个人计算机上。同时，服务器上的资源对所有用户是可用的。所以，客户/服务器计算是集中计算和非集中计算的混合形式。通常，客户系统提供一个图形用户界面，使用户能够很轻松地掌握不同的应用。服务器支持共享设备，如数据库管理系统。实际应用按一定规则分散到客户和服务器上。

任何分布式系统中的关键机制都是进程间通信。经常使用的技术有两个：消息传递和远程过程调用。消息传递是在一个系统中生成消息，应用相同类型的会晤和同步规则去传递和接收消息。使用远程过程调用时，不同机器上的两个程序通过使用过程调用/返回语法和语义进行交互。调用者和被调用者就好像运行在同一台机器上一样。

习　题　九

9.1　简述分布式操作系统与网络操作系统的异同。

9.2　比较客户/服务器结构与集中式结构的特点。

9.3　中间件的主要作用是什么？

9.4　什么是远程过程调用(RPC)？试述实现RPC的基本原理。

分布式进程管理

本章先讨论分布式操作系统所用的关键机制——进程迁移，它已成为分布式操作系统中的热门课题；接着，讨论不同系统上的进程如何协调它们的行为；最后，讨论分布式进程管理中的两个主要问题：互斥和死锁。

10.1　进程迁移

进程迁移就是将一个进程的状态，从一台机器(源机)转移到另一台机器(目标机)上，从而使该进程能在目标机上执行。这个概念主要来自于对大量互连系统间负载平衡方法的研究，但其应用已超出了这个领域。

10.1.1　进程迁移的动机

进程迁移在分布式操作系统中很重要，主要有以下几个原因：

① 负载共享。通过将进程从负载较重的节点迁移到负载较轻的节点，使系统负载达到平衡，从而提高整体执行效率。经验表明，这是可行的，但要注意设计负载平衡算法。一般，分布式系统实现负载平衡所需的通信量越多，系统的执行效率就越低。

② 减少通信开销。可以将相互间紧密作用的进程迁移到同一节点，以减少它们相互作用期间的通信耗费。同样，一个进程在对比它自身要大的某一文件或文件集进行数据分析时，将进程迁移到数据所在处比移动数据要好些。

③ 可获得性。运行时间较长的进程在出现错误时可能需要迁移，而这些错误可以预知，如果操作系统提供这种通知的话。一个想继续的进程可以迁移到另外的系统，或者确信它能在以后某一时刻在当前系统中重新开始。

④ 利用特定资源。一个进程可以迁移到某特定节点上，以利用该节点上独有的

硬件或软件功能。

10.1.2　进程迁移的机制

在设计一个进程迁移机制时要考虑许多问题，其中包括：

① 由谁来激发迁移？

② 进程的哪一部分被迁移？

③ 如何处理未完成的消息和信号？

1. 迁移激发

由谁激发迁移取决于迁移机制的目的。若其目的在于负载平衡，那么，通常由操作系统中掌管系统负载的组件决定什么时候进行迁移。该组件负责抢占或通知将被迁移的进程，并决定进程迁移到哪儿，组件需要同其他类似组件通信，以掌握其他系统的负载情况；若其目的在于获得特定资源，那么，可由进程自行决定何时进行迁移。对后一种情况，进程必须了解分布式系统的分布情况；对前一种情况，整个迁移作用以及多系统的存在，对进程都可以是透明的。

2. 迁移什么

当迁移一个进程时，必须在源系统上破坏该进程，并在目标系统上建立它。这是进程的移动，并不是过程的复制。因此，进程映像，至少包括进程控制块，必须移动。另外，这个进程与其他进程间的任何链接(例如，那些附带的消息和信号)也必须更新。图10.1对此作了解释。进程3从机器S迁移到机器D上，该进程包含的所有链接标识符(由小写字母表示)保持原状，由操作系统负责迁移进程控制块和更新链接映射。进程从一台机器迁移到另一台机器对迁移进程及其通信进程来说都是不可见的。

进程控制块的移动是很简单的。从执行的角度看，困难在于进程地址空间的传送和进程打开文件的传送。考虑第一个问题——进程地址空间的传递，假设用虚拟存储策略(分段或分页)，有两种方法：

① 迁移时传送整个地址空间，这当然是最简单的方法。这样，在以前系统中就再也找不到与该进程相关的任何信息了。但是，如果地址空间非常大，而进程并不需要地址空间中的大部分，那么这种方法就可能太昂贵了，显然没有必要。

② 仅传送那些在主存中的地址空间部分，在需要时再传送虚拟地址空间中的段，这可使传送的数据量最少，但需要源机在进程生存期间不断修改段表或页表。

如果进程并不使用太多的非内存地址空间(例如，进程仅暂时到另一台机器上对一个文件进行操作，然后返回)，那么第二种方案比较好。如果要逐渐访问那些不在内存的地址空间，那么零碎地传送地址空间的各段，就没有迁移时一次传送所有地址空间有效了。

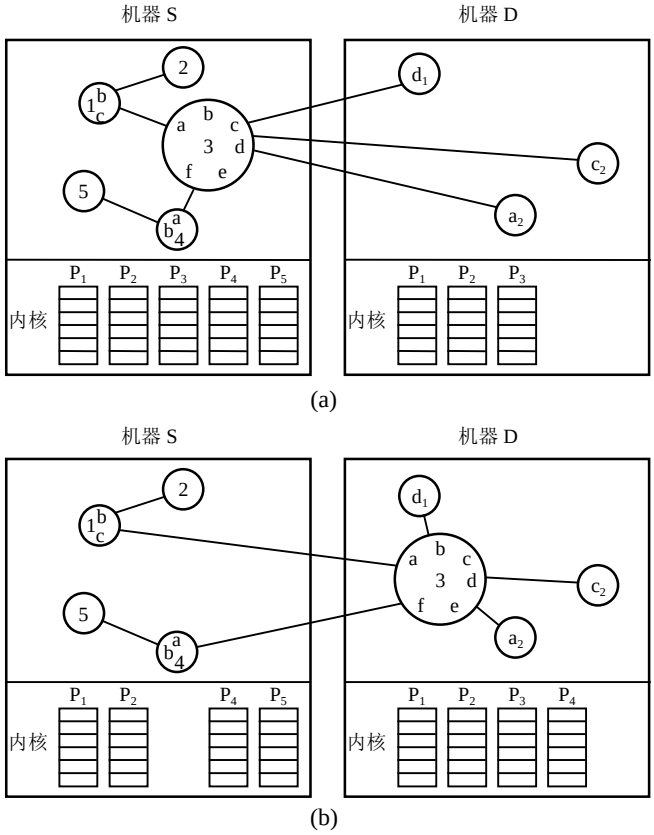


图 10.1　进程迁移举例

(a) 迁移前　(b) 迁移后

许多情况下，不可能预先知道是否需要较多的非内存地址空间。但是，如果进程的结构用线程来表示，并且迁移的基本单位是线程而不是进程，那么第二种策略看起来较好。确实，由于进程余下的线程被留在后面，同样也需要访问进程的地址空间，因此，第二种策略几乎被淘汰。Emerald操作系统实现了线程的迁移。

对打开的文件作类似考虑。如果文件最初与将被迁移的进程在同一系统中，且该文件被该进程加了互斥访问锁，那么将文件与进程一起传送就比较好。但有可能进程仅暂时迁移并且直到返回都不需要这个文件。因此，仅在迁移进程有访问需求时才传送整个文件较好。

3. 消息和信号处理

对于那些未完成的消息和信号，可通过一种机制进行处理：在迁移进行时，暂时存储那些完成的消息和信号，然后将它们直接送到新的目的地，有必要在初始位

置将前向信息维持一段时间，以确保所有的未完成消息和信号都被传送到目的地。

4. 一种迁移方案

下面讨论在IBM的AIX操作系统上的一种迁移机制，它是自我迁移的典型代表。AIX操作系统是一种分布式的UNIX操作系统，在LOCUS操作系统上可得到类似的机制。实际上AIX系统是对LOCUS的改进。

迁移按以下事件序列进行：

① 当进程决定迁移自身时，它选择一个目标机，发送一个远程消息，该消息带有进程的部分映像及打开文件信息。

② 在接收端，内核服务进程生成一个后代，将这个信息交给它。

③ 这个新进程收集完成其操作所需的数据、环境、自变量及栈信息。如果程序文本是不纯的，那么它是被拷贝过来的；如果是纯的，那么它就是全局文件系统所要求的页面。

④ 迁移完成时通知源进程，源进程就发送一个最后完成消息给新进程，然后破坏自己。

当另一个进程激发迁移时有类似的操作序列，其不同之处在于要迁移的进程必须被挂起，使它可以在非运行态被迁移。

在上述方案中，迁移是一个自动的过程，它用许多步来移动进程映像。当迁移是由另一个进程激发，而不是自我迁移时，其方案就是将进程映像及其整个地址空间拷贝到一个文件里，利用文件传送机制将这个文件拷贝到另一台机器上，然后在目标机上按该文件重新生成进程。

10.1.3　进程迁移的协商

进程迁移涉及到进程迁移的决定，在某些情况下，由单个实体作决定。例如，如果进程迁移的目标在于负载平衡，则由负载平衡组件监管一些机器上的相对负载情况，在有必要保持负载平衡时执行迁移；如果使用自我迁移机制以使进程可以访问特定资源或较大的远程文件，那么由进程自己作决定。但是，有些系统允许指定的目标系统参与作决定，主要是为了保持对用户的响应时间。例如，如果进程迁移到用户所在系统，那么对用户的响应时间可能明显下降。虽然，这样做可以提供更好的整体平衡性。

Charlotte采用的就是协商机制，由Starter进程负责迁移策略(什么时候将哪个进程迁移到什么地方)，它也负责长程调度和内存分配。因此，Starter可以在这3方面进行协调，每个Starter进程可以控制一簇机器，Starter从它控制的每台机器的内核获取当时的详细负载统计信息。

必须由两个Starter进程联合来决定迁移，如图10.2所示，并按以下步骤进行：

① 由控制源系统S的Starter决定将进程P迁移到特定的目标系统D，它发送一个

消息给D的Starter要求传送。

② 如果D的Starter准备接受进程P，就回复一个肯定的确认。

③ S的Starter通过服务请求的方式将这个决定传给S的内核(如果Starter在S上运行)，或者将消息传给S机上的KernJob(KJ)进程，将来自远程进程的消息转换成服务请求。

④ 于是S的内核将进程P传给D，包括对P的年龄、处理器、通信负载的统计。

⑤ 如果D资源不足，它可以拒绝接受，否则，D的内核就把来自S的信息传给它的控制Starter。

⑥ Starer通过一个MigrateIn请求与D协商决定迁移策略。

⑦ D保留必要的资源以避免死锁和流控问题，然后给S返回一个接收信息。

图10.2显示了两个进程A和B，有链与P连接。按上述步骤，S所在的机器1必须发给机器0和2一个链接更新消息，以保持A和B与P之间的链接。之后，通过P的任一链接发给P的消息，都将直接发给D，这些消息可与上述各步同时进行交换。

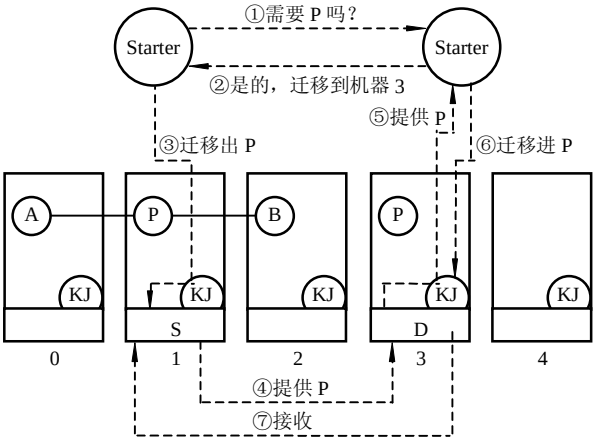


图 10.2　进程迁移的协商

在第⑦步及所有链接更新后，S将P的所有现场情况收集到一个消息中，发给D。

机器4同样在运行，但它与这次迁移无关，因此，在这段时间里它与其他系统没有通信。

10.1.4　进程驱逐

协商进程允许一个系统将迁移到其上的进程驱逐出去。例如，如果一个工作站空闲，则可能有一个或多个进程迁移到其上，一旦该工作站上的用户要使用，就有

必要将迁移来的进程驱逐出去，以给用户提供足够的响应时间。

Sprite提供了这种驱逐能力。在Sprite中，每个进程在其生存期内都好像仅在一个主机上运行。这台主机是该进程的“家节点”(Home Node)，如果某进程被迁移，它就成为目标机上的“外来进程”(Foreign Process)。目标机任何时候都可驱逐外来进程，于是外来进程有可能被迫迁移回其“家节点”。

Sprite的驱逐机制包含如下部分：

① 每个节点都有一个根据当前的负载来决定什么时候接受新的外来进程的监管进程。如果监管进程检测到其工作站上的用户要使用该工作站的行为，它就对每个外来进程激发一个驱逐过程。

② 如果一个进程被驱逐，它就迁移回其家节点，在其他节点可以使用时该进程又可进行迁移。

③ 尽管驱逐所有进程需要一段时间，但被标记驱逐的所有进程可以被立即挂起。允许一个被驱逐进程在等待驱逐期间运行，可以减少该进程的冻结时间，也可以保持驱逐操作进行时主机对其他事务的处理能力。

④ 将被驱逐进程的整个地址空间传送回家节点，通过在以前的主机恢复迁移进程的内存映像，可以减少迁移进程的时间。这就需要该主机牺牲一些资源，以备被驱逐进程以后使用。

10.1.5　抢占及非抢占进程的迁移

抢占进程的迁移，包括传送一个部分运行的进程，或者是已生成但还未运行的进程。迁移非抢占进程的方法较简单，仅传送那些尚未运行的进程，因此，不要求传送进程的状态。这两种迁移，都必须将进程运行的环境信息传给远程节点，这包括用户当前的工作目录和进程继承的权限。

非抢占进程的迁移在负载平衡中很有用。其优点是可以避免频繁的进程迁移，其不足在于对负载分配的突变反应不灵敏。

10.2　分布式全局状态

10.2.1　全局状态及分布式快照

在紧耦合系统中存在的所有同步问题，如互斥、死锁、饥饿，在分布式系统中同样存在。由于系统没有全局状态，故这方面的设计策略更复杂。也就是，操作系统或任一个进程都不可能知道分布式系统中所有进程的当前状态。一个进程可以通过访问内存中的进程控制块而得知本地系统上所有进程的当前状态。对于远程进程，只能通

过消息原语接收状态信息，但这些状态信息只表示远程进程某一时刻的状态。

由于分布式系统的特性引起时间滞后，使其与同步相关的问题更加复杂化，因此，用进程/事件图(图10.3和10.4)来解释这个问题。在这些图中，每个进程都用一个水平轴表示时间，其上的点表示事件(如，内部进程事件，发送消息，接收消息)。点外的方块表示在该点获得的本地进程状态的快照。箭头表示两进程间的消息。

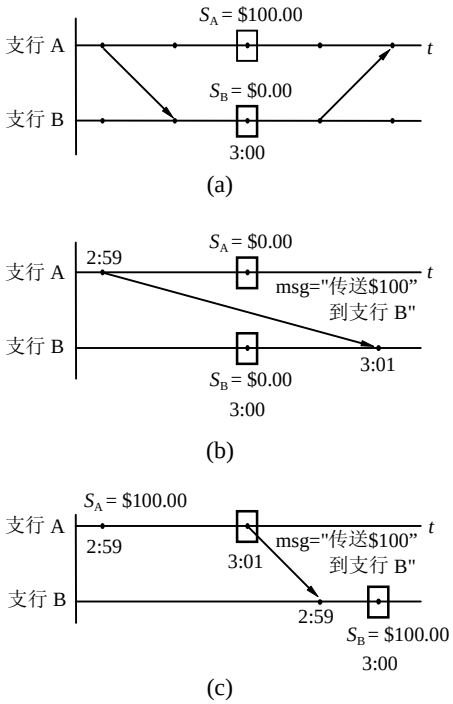


图 10.3　确定全局状态的例子

(a) 总和=$100.00

(b) 总和=$0.00

(c) 总和=$200.00

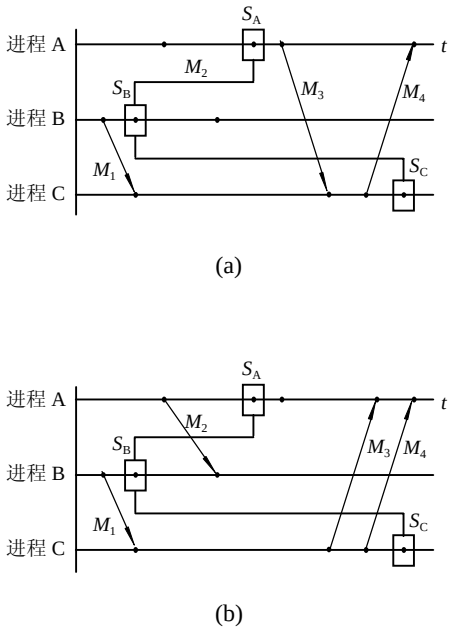


图 10.4　不一致和一致的全局状态

(a) 不一致的全局状态

(b) 一致的全局状态

例如，一个用户的银行记录分布在两个支行中，为统计该用户的总存款数，必须统计其在每一支行的存款量。假设在下午3点作统计，图10.3(a)表示两记录中的100美元的平衡情况。但是，图10.3(b)的情形也是可能的：支行A正传送该款项给支行B，就得到错误的结果0美元。这时，若检查所有正传送的消息，就可以解决这个问题。支行A保留所有传出存款的记录，记录传送目标的确认结果，并将当前存款量和传送记录都包含在支行存款量中。这时，观察者就会得到一个离开支行A向B传送的用户款项。由于该款项尚未到达B，因此，将其加入到总数量中，任何传送并已收到的

款项仅计算一次。

图10.3(c)所示的策略并不安全，如果两支行的时钟并不完全同步，支行A下午3点用户存款为100美元，在支行A时钟为下午3:01时传送给支行B，若这时B时钟却为2:59，即在2:59收到，则款项在3:00就被计算了两次。

考虑到这些问题，为了寻求一个合适的解决方案，首先对下面术语给出定义：

① 通道。如果两个进程交换信息，那么它们之间就有一个通道。我们把通道看成是传送消息的路径或方法。为了方便，通道只是单向的。因此，如果两个进程交换消息，就需要两条通道，一个负责一个方向的消息传送。

② 状态。一个进程的状态就是该进程所带通道发送和接收的消息序列。

③ 快照。一张快照记录一个进程的状态，每张快照包括上次快照后所有通道上发送和接收的所有消息的记录。

④ 全局状态。所有进程的联合状态。

⑤ 分布式快照。快照集，一个进程一个。

面对的问题是，由于消息传送的时间滞后，不能决定真正的全局状态。我们希望通过从所有进程收集快照来决定全局状态。例如，在快照时，图10.4(a)的全局状态表示在通道(A，B)、(A，C)、(C，A)上都有一个消息在传送，消息2(M_2)和4(M_4)是正确的，但是，消息3(M_3)不正确。分布式快照，表示已收到该消息但还没有发送。

我们希望分布式快照记录一个一致的全局状态。如果对于每个记录，接收消息的进程状态都有该消息的发送记录在发送消息进程的进程状态中，那么全局状态是一致的，如图10.4(b)的信息所示。如果一个进程记录了一条消息的接收，但相应的发送消息进程没有记录消息已发送的信息，就会发生不一致的全局状态。

10.2.2　分布式快照算法

在记录一致的全局状态的分布式快照算法中，假设消息按序发送，并且没有丢失。可靠的传输协议(如TCP)可以满足这些要求，算法用到一个专门的控制消息，叫做marker。

一些进程在发送更多消息前，记录其状态并通过所有输出的通道发送给marker，从而激发该算法。每个进程P都按如下步骤进行。第一次收到marker(来自进程q)，接收进程p执行如下步骤：

① 记录其当前局部状态S_p。

② 将从q到p的输入通道记录为空状态。

③ 将marker沿所有输出通道传送给其所有邻居。

以上几步必须自动执行。也就是说，在以上3步执行结束之前，p不能发送或接收任何消息。

在记录了状态后，当p从另外的输入通道(来自进程r)收到一个marker时，接受进

程p就执行下面的步骤：

① 将从r到p的状态记为：从p记录其状态S_p到它接收来自r的marker期间，p从r收到的消息序列。

② 一旦所有输入通道都收到marker，算法在这个进程上中止。

对算法讨论如下：

① 通过发送一个marker，使任何进程都可开始该算法。实际上，几个节点各自独立地记录状态，算法仍有效。

② 若每个消息(包括marker)都在有限时间内传送，则算法将在有限时间内终止。

③ 这是一个分布式算法，每个进程都记录自己的状态和所有输入通道的状态。

④ 一旦所有状态都记录了(算法在所有进程都中止了)，算法就得到一致的全局状态。每个进程将它记录的数据从每个输出通道发送出去，并且每个进程将它接收到的状态数据从每个输出通道向前传送，这样每个进程都可得到一致的全局状态。另一种方法是，初始进程向所有进程索要全局状态。

⑤ 算法不受影响，也不被进程采用的任何其他的分布式算法影响。

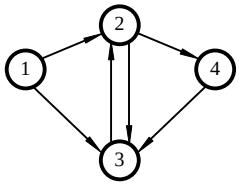


图 10.5　进程和通道图

下面讨论算法的用法。如图10.5所示，一个节点代表一个进程，两节点间的连线表示一个单向通道，箭头表示方向。假设快照算法在运行，每个进程在每条输出通道有9条消息发送，进程1在发送6条消息后要记录全局状态，进程4在发送3条消息后要记录全局状态。中止后，从每个进程收集快照，结果如图10.6所示。进程2在记录状态前从它的每条输出通道各发送4条消息，分别给进程3和进程4。它在记录状态前从进程1收到4条消息，将消息5和6留在通道中。检查快照的一致性：每条消息或者在目标进程被接收，或者已被标记为在通道中传送。

进程1	进程3
输出通道	输出通道
2发送1，2，3，4，5，6	2发送1，2，3，4，5，6，7，8
3发送1，2，3，4，5，6	进入通道
进入通道	1接收1，2，3存储4，5，6
进程2	2接收1，2，3存储4
输出通道	4接收1，2，3
3发送1，2，3，4	进程4
4发送1，2，3，4	输出通道
进入通道	3发送1，2，3
1接收1，2，3，4存储5，6	进入通道
2接收1，2，3，4，5，6，7，8	2接收1，2存储3，4

图 10.6　快照举例

分布式快照算法有效、灵活，它可将任何集中式算法用于分布式环境，这是因为任何集中式算法的基础都是知道全局状态。也可在分布式算法中设一个检查点，以便在失败时撤消和恢复。

10.3　分布式进程管理——互斥

在第3章和第4章中曾提到的并发进程执行的相关问题，其中两个主要问题就是互斥和死锁。这两章主要讨论在单机系统中只有一个主存而有多个进程的环境下解决这个问题的方法。为解决分布式操作系统和没有共享主存的多处理器中发生的相关问题，就需要新的解决方法。互斥和死锁算法必须依靠消息的交换，而不能依靠对主存的访问。本小节讨论分布式操作系统中的互斥和死锁问题。

10.3.1　分布式互斥

当两个或多个进程竞争系统资源时，有必要利用互斥机制，假设两个或更多的进程都要使用单一非共享资源，如打印机。执行时，每个进程向I/O设备发命令，收到状态信息，发送数据，或接收数据。把这样的资源看做是临界资源，程序中使用这类资源的部分看做是程序的临界段。在某一时刻仅允许一个程序位于其临界段。

进程的并发运行要求能够定义临界段和实行互斥，要支持互斥必须满足以下要求：

① 在某一时刻，仅允许拥有同一资源或共享临界段的进程中的一个进入临界段。

② 在非临界段运行结束的进程，不能影响其他进程。

③ 访问临界段的进程，不能无限期地被延迟，即无死锁和饥饿。

④ 没有进程在临界段时，应允许任何要求进入临界段的进程进入，并且无延迟。

⑤ 对进程执行速度或进程数无任何假定。

⑥ 一个进程在其临界段中只逗留有限时间。

图10.7描述了一个可用来在分布式环境中解决互斥问题的模型。假设系统由网络互连，并且假设由操作系统中的某一进程来进行资源分配。每个这样的进程都控制一些资源并为一些用户进程服务。其任务就是设计一个算法，使多个并发进程相互合作以实行互斥。

互斥算法可以是集中式的，也可以是分布式的。在集中式算法中，将有一个节点作为控制节点，控制对所有共享对象的访问。当任一进程要求访问临界段时，它就发送一个Request消息给本地资源控制进程，资源控制进程就发送一个Request消息给控制节点。当其共享对象可用时，控制节点就返回一个Reply(允许)消息；

当进程使用完资源后，就发送一个Release消息给控制节点。这个集中式算法有两个特点：

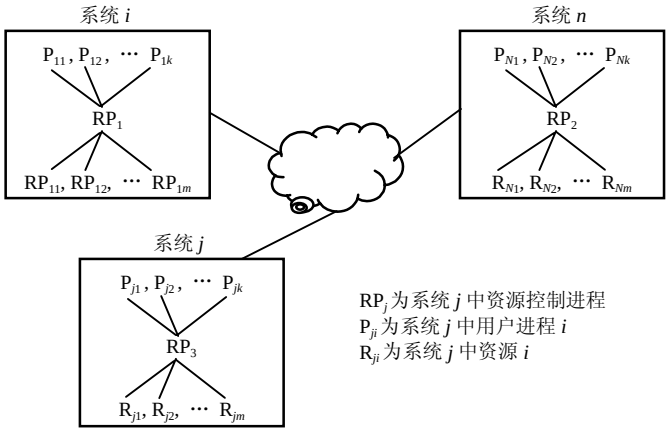


图 10.7　分布式进程管理中互斥问题模型

① 只有控制节点才能分配资源。

② 所有必要信息都集中在控制节点，包括所有资源的性质和位置，以及每个资源的分配状态。

集中式算法简便，对互斥的实施很明了：直到资源被释放，控制节点才满足对该资源的要求。但是，这种方法有几个缺点，如果控制节点失效，互斥机制就无用了。另外，资源的分配和回收都需要与控制节点交换消息，因此，控制节点就成为了系统性能的瓶颈。

由于集中式算法存在上述问题，有必要设计分布式算法，分布式算法应有如下特性：

① 所有节点有同等的信息量。

② 每个节点只有整个系统的部分信息，必须根据这些信息作决定。

③ 所有节点共同负责作最后决定。

④ 所有节点对最后决定所起的作用相同。

⑤ 一般，一个节点失效，不会引起整个系统的失效。

⑥ 没有调整事件计时的统一的系统共同时钟。

需要对第②点和第⑥点作进一步说明：

对第②点，一些分布式算法允许节点之间相互交流信息。即使这样，在某一时刻，仍有一些信息可能被滞留在传递中，未能及时到达其他节点。因此，由于消息传递的滞后性，当前节点的信息可能不是最新的。

对第⑥点，由于系统通信的延迟，要求所有系统有一个统一的时钟是不可能的；

使所有本地时钟都与中央时钟完全同步也很困难，经过一段时间后，有些本地时钟可能偏移。

由于通信的延迟，再加上没有统一的时钟，就使得在分布式系统中实行互斥机制比在集中式系统中要困难得多。在研究分布式互斥算法前，我们先讨论解决统一时钟问题的一般方法。

10.3.2　分布式系统的事件定序——时戳方法

大多数互斥和死锁的分布式算法的基本操作是对事件进行定序。没有统一时钟或没有同步的局部时钟是主要问题，可用下述方法表示，系统i中的事件a在系统j中的事件b之前，并且希望在网络中得到公认，但是，由于以下两个原因而难以实现：第一，一个事件的实际发生与在系统中观察到它的时间之间可能有延迟；第二，缺少同步使不同系统中的时钟有偏差。

为了解决这个问题，用“时戳”对分布式系统中的事件定序，而不需要物理时钟。这种方法很有效，在绝大多数互斥和死锁算法中都用到它。

首先，对事件下一定义。讨论一个系统的行为、一个事件，如一个进程进入或离开临界段，但是，在分布式系统中，进程间通过消息相互作用，因此，有必要将事件与消息联系起来。事件与消息联系起来很简单。例如，一个进程在它想进入临界段或要离开临界段时可以发送一条消息，这里，只将事件与发送消息联系起来，而不与接收消息连接起来。因此，每一个进程发送一条消息，在消息离开该进程时就可定义一个事件。

时戳方法就是对由消息传送的事件定序。网络中的每个系统i都有一个本地计数器C_i，它相当于一个时钟。每次系统传送一条消息就将其计数器加1，消息按

$$(m，\ T_i，\ i)$$

形式传送，其中，m为消息的内容；T_i为消息的时戳；i为该站点的编号。

当该消息达到时，接收系统j将它的时钟置为

$$C_j=1+\max[C_j，\ T_i]$$

在每个站点，事件的顺序按下面的规则决定。对i处的消息x和j处消息y，如果满足下面一个条件，就称x先于y：

① $T_i<T_j$

② $T_i=T_j$并且$i<j$

消息事件的时间就是该消息所带的时戳，这些时间按上面两个规则来决定顺序，也就是说，时戳相同的两个消息事件的顺序按它们的编号排列。由于这个规则的运用独立于站点，因此，这个方法可以避免通信进程由不同时钟偏移所引发的任何问题。

图10.8给出了该算法运行的一个例子，其中有3个站点。每个站点由一个控制

时戳算法的进程表示。进程P_1在时钟值0时开始，为传送消息a，它将时钟加1传送(a，1，1)，第一个1表示时戳，第二个表示站点。该消息由站点2和3处的进程接收。在这两种情况下，当地时钟都是0，并且被置为2=1+max[1，0]。P_2要传送下一个消息时，首先将其时钟增加到3；接收这条消息后，P_1和P_3必须将它们的时钟增到4。接下来，P_1发消息b，同时P_3发消息j，并且有相同的时戳，根据定序规则，并不会产生冲突。所有事件发生后，所有站点的消息定序都相同，都是{a，x，b，j}。

该算法并不注重两个系统间通信时间的差别，如图10.9所示，其中P_1和P_4发送的消息有相同时戳，在站点2处来自P_1的消息比来自P_4的到得早，但是，在站点3来自P_1的消息比来自P_4的要晚。然而，在所有站点都收到消息后，消息事件定序在各处都一样，都是{a，q}。

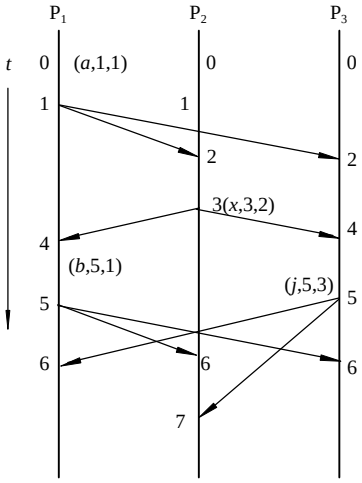


图 10.8　时戳算法运行举例

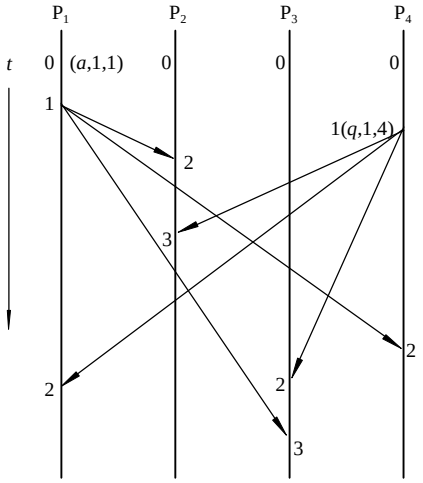


图 10.9　时戳算法运行举例

注意：此方法所得的序列并不一定与实际时间顺序相同。由于算法基于时戳，因此，哪一个事件真正先发生并不重要，重要的是调用算法的所有进程都遵循这一事件定序规则。

在刚刚讨论的例子中，每条消息都是从一个进程传向另外所有的进程。如果有些消息不按这种方式传送，就不可能在所有位置得到相同的消息序列。在这种情况下，就要汇集部分序列。但是，我们主要在分布式算法中利用时戳来解决互斥和死锁。在这种算法中，一个进程通常发一条消息(带时戳)给其他所有进程，时戳用来决定消息如何处理。

10.3.3　分布式互斥算法

1. Lamport 算法

最早提出的分布式互斥方法是Lamport算法，它基于分布式队列的概念，该算法基于以下假设：

① 分布式系统由N个站点组成，每个站点有惟一编号，从1到N。每个站点都有一个进程负责进行互斥访问资源，也负责处理那些同时到达的请求。

② 按发送的顺序接收从一个进程到另一个进程的消息。

③ 每条消息都在有限时间内正确地送到目的地。

④ 网络是全互连的，这就意味着每个进程都可以直接向其他任何进程发送消息，而不需要中间进程转发消息。

假设②和③可通过一个可靠传输协议来实现。为了简便，假设每个站点仅控制一个资源。

我们试图将该算法一般化，使它也能在集中式系统中运行。如果由一个中央进程来管理资源，则可以将输入请求排序，按先进先出方式满足请求。要在分布式系统中得到这样的算法，所有站点必须有这种队列的拷贝。可用时戳来确保所有站点都符合资源要求的顺序。现在的问题是：由于网络传送消息需要一定时间，就可能发生多个站点对哪个消息是队列的头产生不一致的看法。如图10.9所示中，在消息a到达P_2和消息q到达P_3时，还有另外的消息在传送。因此P_1和P_2认为消息a是队列头，而P_3和消息P_4认为消息q是队列头，这就不满足互斥的要求，于是增加以下规则：如果一个进程要根据它的队列进行分配时，就需要从所有其他站点收到一个消息，以确保没有比它自己队列头更早的消息还在传送之中。

算法所用的数据结构可以是一个队列，也可以是一个数组，若是数组，则每个站点占其中一个数据项：项q[j]表示来自P_j的消息。数组初始化为

$$q[j]=(\text{Release}，0，j)\quad j=1，\cdots，N$$

本算法中用到以下3种消息。

① (Request，T_i，i)：P_i请求访问资源。

② (Reply，T_j，j)：P_j允许对它所控制的资源进行访问。

③ (Release，T_k，k)：P_k释放以前分配给它的一个资源。

Lamport算法如下：

① 当P_i请求访问资源时，它发送一个请求(Request，T_i，i)，时戳是当前本地时钟值。它将此消息放入自身数组的q[i]中，然后，将这个消息传给所有其他进程。

② P_j收到请求(Request，T_i，i)后，将这个消息放入自身数组的q[i]中，并传送(Reply，T_j，j)给所有其他进程。这一步就是为了满足上述规则，以确保在作决定时没有更早的消息在传送。

③ P_i满足下面两个条件就可访问一个资源(进入其临界段)：

• 数组q中P_i的请求消息是数组中最早的请求消息。由于消息在所有站点的顺序一致，这个规则允许在任一时刻仅有一个进程访问资源。

• 本地数组中所有其他消息都比q[i]中的消息晚。这个规则确保P_i已知道所有在它当前请求前面的请求。

④ P_i通过发送(Release,T_i, i)消息来释放所占用的资源。该消息也置入其自身的数组项中，并传递给所有其他进程。

⑤ 当P_i接收到(Release,T_j, j)消息，它用该消息替换q(j)的当前内容。

⑥ 当P_i接收到(Reply,T_j, j)消息，它用该消息替换q(j)的当前内容。

很容易证明此算法满足互斥要求：公平，不死锁，不饥饿。

① 互斥：通过时戳机制对消息定序，再根据这个顺序来处理进入临界段的请求。一旦P_i要进入临界段，系统中就不可能有其他的请求消息在其前传送，因为那时P_i已从其他所有站点收到了一个消息，这些消息比它的请求消息要晚，Reply消息机制可以确保这一点。

② 公平：严格按照时戳顺序满足请求。因此，所有进程有相等的机会。

③ 无死锁：一旦P_i退出其临界段，它就发送Release消息，从而可在所有其他站点将P_i的Request消息删去，允许其他进程进入临界段。

为保证互斥，该算法需要3(N–1)条消息：(N–1)条Request消息，(N–1)条Reply消息，(N–1)条Release消息。

2. Ricart 算法

Ricart对Lamport算法进行了一些简化，希望将Release消息删去以优化原始算法。Ricart算法的假设与以前相同，但不一定保证按照进程发送的顺序接受信息。

同前，每个站点都有一个进程负责控制资源的分配。该进程有一个数组q并遵循以下规则。

① 当P_i请求访问资源时，它发出一个请求(Request，T_i，i)。时戳为当前本地时钟之值。它将这条消息放入自身数组q[i]中，然后，将消息发送给所有其他进程。

② 当P_j收到请求(Request，T_i，i)后，按如下规则处理：

• 如果P_j正处于临界段中，就延迟发送Reply允许消息(见规则4)；

• 如果P_j并不等待进入临界段，就发送(Reply，T_j，j)给所有其他进程；

• 如果P_j等待进入其临界段，并且到来的消息在P_j的Request后，则将到来的消息放入其数组的q[i]中，并延迟发送Reply消息；

• 如果P_j等待进入其临界段，但是到来的消息在P_j的Request前，就将到来的消息放入其数组的q[i]中，并发送(Reply，T_j，j)给P_i。

③ 如果P_i从所有其他进程都收到了Reply允许消息，它就可以访问资源(进入临

界段)。

④ 当P_i离开临界段时，它给每个挂起的Request发送一个Reply允许消息，从而释放资源。

该算法中每个进程的状态转换图如图10.10所示。

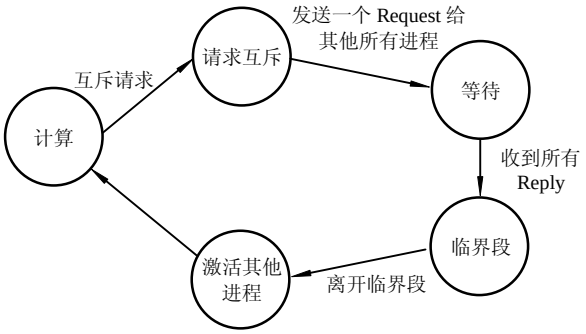


图 10.10　Ricart 算法的状态图

下面做一简单小结：当一个进程想要进入临界段时，它就给所有其他进程发送一条带时戳的Request消息，当它从所有其他进程收到Reply后，它就可以进入临界段。当一个进程收到另一个进程的Request时，它必须酌情发送一个Reply。如果该进程不想进入临界段，它就马上发送Reply。若它想要进入临界段，它就将自己的Request的时戳与收到Request的时戳进行比较，如果后者较迟，它就延迟发送Reply；否则马上发送Reply。

在此算法中，需要2(N–1)条消息：(N–1)条Request消息，表示P_i要进入临界段；(N–1)条Reply消息，以允许其请求的访问。

本算法利用时戳来确保互斥，可以避免死锁。证明不会产生死锁，可以用反证法：如果发生死锁，则在没有消息传送时，就会出现每个进程都发送了一个Request但没收到相应的Reply的情形。由于按Request的顺序来延迟Reply，因此，这种情形不会出现。于是具有最早时戳的Request必定会收到所有相应的Reply，从而不可能发生死锁。

饥饿也可以避免，这是因为Request是有序的。由于Request按序满足，每个Request在某时刻会成为最早的，于是会得到满足。

3. 令牌方法

一些学者提出了许多不同的方法，例如，通过在参与的进程中传递令牌来达到互斥。令牌在某一时刻只能被一个进程得到，握有令牌的进程无需请求就可进入临界段。当一个进程离开临界段时，它就将令牌传给另一个进程。

下面讨论一种有效的令牌方法。这个算法需要2个数据结构：令牌从一进程传到

另一个进程，它实际上是一个数组token，其第k个元素记录上一次令牌传到进程P_k的时戳；每个进程也有一个数组Request，其第j个元素记录上次从P_j收到Request的时戳。

算法过程如下：最初，将令牌随机分配给一个进程，即将该进程的token present置为true。当一个进程要进入临界段时，如果它拥有令牌就可进入；否则，它向所有其他进程广播带时戳的Request请求消息，等待直到它得到令牌。当进程P_i离开临界段时，它必须将令牌传给其他进程，它按i+1，i+2，…，1，2，…，i–1的顺序搜索Request数组，找到第一个满足P_j最新要求的、时戳比其token中记录的P_j上一次拥有令牌的时间值要大的进程P_j，也就是满足：Request[j]＞token[j]的P_j。

图 10.11 描述了令牌传递算法，该算法包括两部分：第一部分处理对临界段的

```
if not token-present then begin clock:=clock+1;              [Prelude]
                                broadcast(request, clock, i);
                                wait(access, token);
                                token-present:=True
                          end;
endif;
token-held:=True;
<critical section>
token(i):=clock;                                             [Postlude]
token-held:=false;
for j:=i+1 to n, 1 to i–1 do
  if (request(j)＞token(j))∧token-present
    then begin
      token-present:=False;
      send(j, access, token)
    end
  endif;
```

(a)

```
When received(request, t, j) do
  request (j):=max (request(j), t);
  if token-present∧not token-held then
  <text of postlude>
  endif
enddo;
```

(b)

记号说明：

send(j, access, token)　将access消息和令牌token传给进程j

broadcast(request, clock, i)　将进程i的请求消息和时戳clock传给所有其他进程

received(request, t, j)　从进程j收到请求消息及时戳t

图 10.11　令牌传递算法(进程 P_i)

(a) 第一部分　(b) 第二部分

使用，由前序、临界段、后序组成；第二部分处理对一个请求的接收，时钟是作时戳用的本地计数器。Wait(access，token)操作使进程等待直到“access”型的消息(该消息包括一个令牌值)已收到，然后将这个值放到数组token中。

该算法满足下面两种情况之一：

① 如果请求进程没有令牌，则需要N条消息(N–1条广播请求，1条传送令牌)。

② 如果进程已拥有令牌就不需要消息。

10.4　分布式死锁

在第4章中，将死锁定义为一个进程集的永久阻塞，这些进程或者竞争系统资源或者相互通信。这个定义对分布式系统和单个系统同样有效。与互斥一样，死锁在分布式系统中会引起比共享内存系统更为复杂的问题。由于没有一个站点能精确地知道整个系统的当前状态，并且两个进程的消息可能遇到不可预测的延迟，因此，死锁处理在分布式系统中相当复杂。

有两种类型的死锁：资源分配引起的死锁和消息通信引起的死锁。在资源分配引起的死锁中，这些进程都要访问资源，如数据库中数据对象或服务器上的I/O资源，若每个进程所请求的资源都被同一集合的其他进程占有，就会产生死锁。在通信引起的死锁中，消息是进程所等待的资源，如果一个进程集中每个进程都在等待同一进程集中其他进程的消息，但该进程中没有进程曾发送过消息，就会产生死锁。

10.4.1　资源分配中的死锁

在第4章中曾提到，只有满足以下所有条件时，才会出现资源分配的死锁：

① 互斥。一次只有一个进程能使用资源。

② 占有并等待。一个进程在占有分配资源的同时等待其他资源。

③ 非抢占。不能从一个进程中将其占有资源抢占过来。

④ 循环等待。存在一个封闭的进程链，每个进程至少占有链中下一个进程所需的一个资源。

处理死锁的算法主要旨在预防循环等待的信息，或者检测实际或者潜在性的死锁发生。在分布式系统中，资源分布在不同的站点，对资源的访问由控制进程来控制，这些控制进程对系统当前全局状态并不完全了解，因此，必须根据局部信息来决定。于是，就需要新的死锁算法。

分布式死锁处理遇到的困难在于“死锁幻象”(Phantom Deadlock)。图10.12对死锁幻象进行了解释。$P_1 \rightarrow P_2 \rightarrow P_3$表示$P_1$等待$P_2$占有资源，而$P_2$等待$P_3$占有的资源。假设

在开始时，P_3占有资源R_a，P_1占有资源R_b，再假设现在P_3先发送一个消息释放R_a，然后发送消息请求R_b。如果第一个消息比第二个先到达循环检测进程，就出现图10.12(a)所示的结果，它正确反映了资源请求关系。但是，如果第二个消息比第一个先到，就显示死锁(见图10.12(b))。这是一个错误的检测，并不是真正死锁，是由缺少全局状态而引起的。在集中式系统中不存在这样的情况。

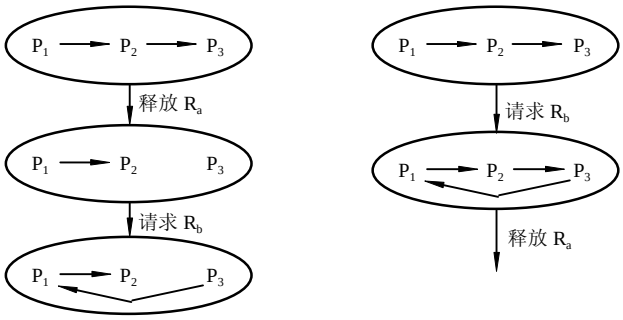


图 10.12　死锁幻象

(a) 请求前释放到达　(b) 释放前请求到达

10.4.2　死锁预防

第4章讨论的死锁预防方法中有两种可以用在分布式环境中：

① 对资源类定义一个线性序列可破坏循环等待条件。如果一个进程分配了R类的资源，那么它接下来只能请求处于R后的资源。这种方法的缺点在于资源并不是按它们使用的顺序请求。因此，资源可能比需要占有的时间要长。

② 一个进程一次请求它所需要的全部资源，通过阻塞该进程直到所有请求同时被满足为止，这可破坏占有并等待的条件。这种方法在以下两方面并不有效。第一，一个进程可能被阻塞很长一段时间以等待所有的请求被满足，而实际上它在部分资源上就可运行；第二，分配给一个进程的资源可能在相当长的一段时间内都不能使用，而在这期间又不能分给其他进程。

这两种方法都需要进程事先就知道它所需要的资源，这并不总是可能的，例如，在数据库的应用中，往往会动态加入一些新项。下面讨论两个不需要这种预见知识的算法，由于我们是围绕数据库讨论，所以只对事务而不是对进程来讨论。

这里介绍的方法利用了时戳。每个事务在生存期内都带有其产生的时戳，从而使事务有了严格的顺序。如果资源R已在事务T_1中用过而被另一个事务T_2请求，就可以通过比较两者的时戳来解决冲突，这个比较可避免循环等待条件的形成。对这个基本方法作两种变化，就得到了wait-die方法和wound-wait方法。

假设T_1目前占有R，T_2发出一个请求。对于wait-die方法，图10.13(a)表示了R处的资源分配所用的算法：两个事务的时戳由$e(T_1)$和$e(T_2)$表示，如果T_2较老，它就被阻塞直到T_1释放R，或者是主动释放或者是在请求其他资源时就被“杀死”(killed)；如果T_1较年青，T_2就重新开始，但其时戳和以前相同。

在发生冲突时，老事务有优先权，被杀死的事务可以恢复，并且带着以前的时戳，这样，老事务会变得更老，从而获得更高的优先权。因此，不需要知道所有资源的分配状态，所需要的只是事务请求资源的时戳。

wound-wait将占有资源的年青事务杀死，满足年老事务的请求，如图10.13(b)所示。与wait-die方法相比，它使年老事务不用等待那些分配给年青事务的资源。

if e (T_2) < e (T_1) **then** halt T_2 ('wait')	**if** e (T_2)＜e (T_1) **then** kill T_1('wound')
else kill T_2 ('die')	**else** halt T_2 ('wait')
endif	**endif**
(a)	(b)

图 10.13　死锁预防方法

(a) wait-die 方法　　(b) wound-wait 方法

10.4.3　死锁避免

死锁避免就是动态确定如果满足一个资源分配请求，是否会引起死锁。由于以下原因分布式死锁避免并不实际可行：

① 每个节点必须记录系统的全局状态，这就需要大量的存储和通信耗费。

② 检测安全全局状态的进程必须是互斥的，否则，两个节点各自考虑一个不同进程的资源请求，同时得出满足请求是安全的结论，但实际上同时满足这两个请求会发生死锁。

③ 对有大量进程和资源的分布式系统而言，检查安全状态，需要相当大的开销。

10.4.4　死锁检测

有了死锁检测，就允许进程先获得它想要的资源，在其后再确定是否死锁。如果检测到死锁，则可选一个进程，释放其资源，从而破坏死锁。

1. 分布式死锁检测策略

分布式死锁检测的困难在于，每个站点仅知道自己的资源，而死锁可能涉及分布式资源。根据系统控制是集中的、分层的还是分布式的，相应有几种策略，如表10.1所示。

表 10.1　分布式死锁检测策略

策　略	优　点	缺　点
集中式策略	• 算法简单，易于实现 • 中央站点有全部信息并且可以解决死锁	• 通信开销大，每个站点必须将状态信息传给中央站点 • 中央站点易于失效
分布式策略	• 单个站点失效影响不大 • 没有站点统一进行死锁检测	• 死锁难于处理，因为可能会有几个站点检测到相同的死锁，而不知死锁涉及的其他站点 • 由于时序问题，难于设计
分层策略	• 单个站点失效影响不大 • 如果大多数潜在死锁已定位，那么死锁解决活动有限	• 很难指出潜在死锁；要明确指出潜在死锁，则耗费比分布式系统要大

集中式策略，由一个站点来负责死锁检测。在所有Request和Release消息送到控制特定资源的进程的同时，也送给这个中央进程。由于这个中央进程了解所有情况，因此在某种意义上可检测死锁。这种方法需要大量的消息，并且中央进程不能失效，另外，有可能检测出死锁幻象。

分层策略，按树的结构组织，其中有一处作为树的根。在每个节点处汇集所有节点的资源分配信息。这就允许在比根节点低的层次上检测死锁。特别是，涉及一组资源的死锁可被那些有冲突资源的节点的共同祖先节点检测出来。

分布式，需要所有进程合作来检测死锁。通常，这就意味着要交换相当数量的带时戳的信息。因此，其开销相当大。

2. 分布式死锁检测算法

现在讨论一种分布式死锁检测算法。该算法用于一个分布式数据库系统，其中每个站点只有部分数据库信息，并且事务可在每处初始化。每个事务最多只有一个未满足的资源请求。如果一个事务需要多于一个的数据对象，则只有在第一个数据对象被满足后才能请求第二个数据对象。

一个站点中的所有数据对象都有两个参数：惟一标号D_i和变量Lock-by(D_i)，后面这个变量在数据对象没有任何事务加锁时为空；否则，其值就是加锁事务的标号。

一个站点中的所有事务j都有4个参数：

① 惟一标号T_j。

② 变量Held-by(T_j)，如果事务T_j在运行或者处于就绪态，就置为空；否则，其值就等于占有事务(T_j)所请求数据对象的事务标号。

③ 变量Wait-by(T_j)，如果事务T_j不等待其他事务，就置为空；否则，其值就等于阻塞事务序列头的事务标号。

④ 队列Request Q(T_j)，它包括对T_j占有的数据对象的所有尚未满足的请求。队列的每个元素都形如(T_k，D_k)，其中T_k是请求事务，D_k是T_j占有的数据对象。

例如，假设事务T_2等待T_1占有的一个数据对象，而T_1又等待T_0占有的一个数据对象。于是相关参数的值如下：

事　　务	Wait-for	Held-by	Request-Q
T_0	nil	nil	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	nil

这个例子中的重要之处在于Wait-for(T_i)和Held-by(T_i)的不同。在T_0释放T_1所需数据对象以前，T_1和T_2都不能进行；在T_0释放该数据后，T_1可以执行，随后释放T_2所需的数据对象。

图10.14描述了用于死锁检测的算法。当一个进程请求对一个数据对象加锁时，与该数据对象相应的服务进程或者满足或者拒绝该请求。若请求没被满足，服务进程就返回占有该数据对象的事务标号。

```
{数据对象Dj收到lock-request(Ti)}
begin
  if Locked-by(Dj)=nil then send granted
  else
    begin
      send not granted to Ti;
      send Locked-by (Dj) to Ti
    end
end

{事务Ti对数据对象Dj发出加锁请求}
begin
  send lock-request(Ti) to Dj;
  wait for granted/not granted;
  if granted then
    begin
      Locked-by(Dj):=Ti;
      Held-by(Ti):= ∅;
    end
  else{假定Dj已被事务Tj使用}
    begin
      Held-by(Ti):=Tj;
      Enqueue(Ti, Request-Q(Tj));
      if Wait-for(Tj)=nil then Wait-for(Ti):=Tj
      else Wait-for(Ti):=Wait-for(Tj);
      update(Wait-for (Ti), Request-Q(Ti))
    end
end.
```

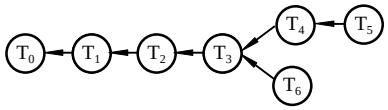
```
{事务Tj收到更新的消息}
begin
  if Wait-for (Tj)≠Wait-for(Ti)
    then Wait-for(Tj):=Wait-for(Ti);
  if Wait-for(Tj)∩Request-Q(Tj)=nil
    then update(Wait-for(Ti), Request-Q(Tj))
  else
    begin
      DECLARE   DEADLOCK;
      {initiate deadlock resolution as follows}
      {Tj被选择作为夭折的事务}
      {Tj释放它所占用的全部数据对象}
      send-clear(Tj, Held-by(Tj));
      allocate each data object Di held-by Tj to
      the first requester Tk in Request-Q(Tj);
      for every transaction Tn in Request-Q (Tj)
      requesting data object Di held-by Tj do
        Enqueue (Tn, Request-Q(Tk));
    end
end.

{事务Tk收到clear(Tj,Tk)消息}
begin
  purge the tuple having Tj as the requesting transaction
from Request-Q(Tk)
end.
```

图 10.14　分布式死锁检测算法

当请求事务收到允许响应时，它就对该数据结构加锁；否则，请求事务将其Held-by变量修改为占有该数据对象的事务标号，并将自己的标号加到占有事务的Request-Q中，将Wait-for变量或者修改为占有事务的标号(如果那个事务不在等待态)，或者修改为占有事务的Wait-for变量值，这样Wait-for变量值就被置为最终阻塞执行的事务标号。最后，请求事务向其Request-Q中所有事务发送一个更改消息，修改所有的Wait for变量。当一个事务收到一个更改消息时，它就修改其Wait-for变量。

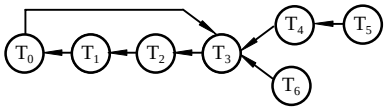
一个事务在收到一个更改消息时，就修改其Wait-for变量，以反映它最初等待的事务正被另一个事务阻塞，然后就进行死锁检测，看是否有它正在等待的事务在等待它，如果没有，它就向前传递更新消息；如果有，就发送一个清空消息给占有它所需数据对象的事务，然后将它所占有的每个数据对象分配给它的Request-Q中第一个请求者，再将剩下的请求者加入到新事务的队列中。

图10.15描述了该算法的运行情况。当T_0请求T_3占有的一个数据对象时，就会产生一个循环。T_0发出一个更新消息，从T_0向T_1、T_2传播。这时，T_3发现其Wait-for和Request-Q的交集不空，就发送一个清空消息给T_2，于是T_3从Request-Q(T_2)中删去，释放其占有的数据对象，激活T_4和T_6。



事　务	Wait-for	Held-by	Request-Q
T_0	nil	nil	T_1
T_0	T_0	T_0	T_2
T_0	T_1	T_1	T_3
T_0	T_2	T_2	T_4,T_6
T_0	T_3	T_3	T_5
T_0	T_4	T_4	nil
T_0	T_5	T_3	nil

(a)



事　务	Wait-for	Held-by	Request-Q
T_0	T_0	T_3	T_0
T_1	T_0	T_0	T_1
T_2	T_0	T_1	T_2
T_3	T_0	T_2	T_4,T_6,T_0
T_4	T_0	T_3	T_3
T_5	T_0	T_4	nil
T_6	T_0	T_3	nil

(b)

图 10.15　分布式死锁检测算法举例

(a) T_0 向 T_3 提出请求前的系统状态　(b) T_0 向 T_3 提出请求后的系统状态

10.4.5　消息通信中的死锁

1. 互相等待

当一组进程中的每一个成员都在等待组内另一个成员的消息，又没有任何消息

在传送时，就会发生消息通信死锁。

为了更详细地解释这种情况，需定义一个进程的依赖集(DS)：对一个停止等待一个消息的进程P_i，其DS(P_i)由P_i所希望得到的消息的所有进程构成，如果任何期望消息都到达，P_i就可以运行。另一种说法是，只有在所有期望消息到达后，P_i才能运行。前一种说法较普遍，我们在此讨论的也是这种说法。

有了前面的定义，进程集S的死锁可定义如下：

① S中所有进程都停止，正等待消息。

② S中所有的进程的独立集都包含在S中。

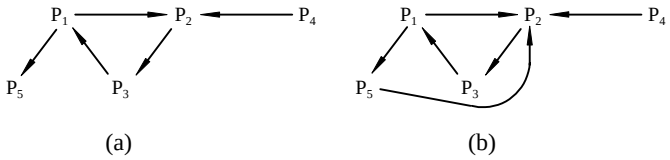
③ S中的成员间没有传送消息。

④ S中的所有进程都被死锁。

消息死锁和资源死锁有所不同。在资源死锁中，如果进程依赖图中存在封闭环，就表明存在死锁。如果一个进程占有另一个进程所需资源时，那么后面这个进程就依赖于前一个进程。在消息死锁中，产生死锁的条件是S中任何成员的后继都在S中，也就是说S是一个群。

图10.16对此作了解释。在图10.16(a)中，P_1等待来自P_2或P_5的消息；P_5不等待，于是P_5可以发消息给P_1，从而释放链(P_1，P_5)和(P_1，P_2)。图10.16(b)中加入了一个依赖：P_5等待P_2的消息。这样，图10.16(b)就成了一环路，于是存在死锁。

和资源死锁一样，消息死锁也可预防和检测。



(a)　　(b)

图 10.16　消息通信中的死锁

(a) 不死锁　(b) 死锁

2. 消息缓冲分配不合理

对正在传送消息的存储缓冲的分配不合理也会引起消息传送系统的死锁。这种死锁在包交换网中居多。下面先在数据网中讨论这个问题，然后在分布式操作系统中讨论。

数据网中最简单的死锁形式是直接的存储转发死锁，如果一个包交换节点用公共缓冲池来缓冲包，就可能产生死锁。图10.17(a)表示节点A的全部缓冲区都被要发给B的包占据，B处缓冲区都被要发给A的包占据。这时A和B都不能接受包，因为它们的缓冲区被占满了。这样两个节点在任何链路上都不能传送或接收数据了。

直接的存储转发死锁是可避免的，只要不将全部缓冲区都分配给一条链就可以了。可以将缓冲区分成大小固定的块，每块对应一条链，就可以预防死锁。

另一种是间接的存储转发死锁，如图10.17(b)所示。对于图中的每个节点，在一个方向上的队列都是满的，并且还有不是传给下一节点的包。预防这种死锁的一个简单方法就是用结构化的缓冲池(见图10.18)。缓冲池分层：第0层缓冲池没有限制，任何输入包都可以存放进去；第1层到第*N*层缓冲池(其中N是网络路径的最大跳数)。按下述方法预留：第*k*层缓冲池留给跳数至少为*k*的包。这样在负载较重时，缓冲池按从第0层到第*N*层的顺序逐渐装满。如果从第0层到第*k*层的缓冲池都被装满了，则跳数小于等于*k*的包被丢掉。这种方法同样可以预防直接的存储转发死锁。

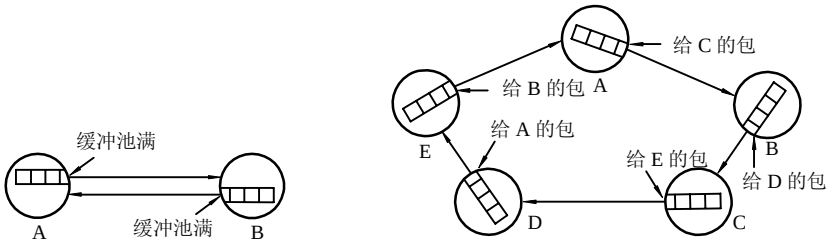


图 10.17　存储转发死锁举例

(a) 直接的存储转发死锁　(b) 间接的存储转发死锁

在利用消息进行通信的分布式操作系统中也会出现相同的问题。如果Send操作是非阻塞的，就需要缓冲区来存放输出消息。可以把存放从进程*x*到进程*y*的消息的缓冲区看成*x*和*y*之间的通信通道。如果通道容量有限(有限缓冲)，那么Send操作可能挂起进程。也就是说，如果缓冲区大小为*n*并且有*n*条消息正在传送(还没有被目标进程接收)，此时如果再执行Send就会阻塞发送进程，直到有消息被接收，让出缓冲区为止。

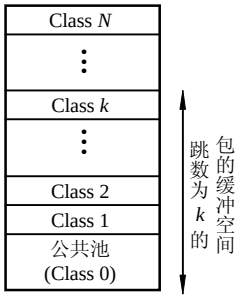


图 10.18　死锁预防的结构化缓冲池

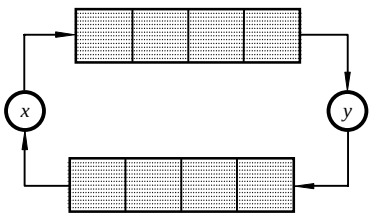


图 10.19　分布式系统中的通信死锁

图10.19描述了有限通道引起死锁的情况。图中有两个通道，每条通道的容量为4，如果每条通道上都有4条消息在传送，并且*x*和*y*都想在接收消息前再传送消息，

于是x和y都被挂起，从而产生死锁。

如果能对系统每对进程间传送的消息加以限制，就可以分配给所有通道其所需的缓冲。这可能造成大量浪费，并且要预先知道消息数。如果不能预先知道请求，或者浪费太大，那么就要优化分配。在一般情况下，这个问题无法解决。

10.5 小　结

分布式操作系统支持进程迁移。进程迁移就是将一个进程的状态从一台机器传送到另一台机器，从而使该进程能在目标机上运行。进程迁移可用来保持负载平衡，减少通信量，提高执行效率，提高可用性，且允许进程访问特定远程资源。

在分布式系统中，可以构造全局状态信息来解决资源竞争问题，从而协调进程。由于消息传送中时延可变性和不可预测性，不同进程都必须在事件顺序上取得一致。

分布式进程管理要处理互斥和死锁问题。互斥和死锁的处理在分布式系统中要比单机系统复杂得多。

习　题　十

10.1　通过以下两点证明分布式快照算法的正确性：

(1) 算法不会记录不一致的全局状态(提示：反证法)。

(2) 算法正确记录了每个通道的状态，也就是说，对每个通道，如从进程r到进程p的通道，通道状态(r，p)就是由r发送的消息序列，它可达到S_r，但p不一定接收到S_p，如图10.4(b)中所示的通道状态(C,A)是{M_3，M_4}(提示：假设消息按其发送的顺序接收)。

10.2　如图10.9所示，考虑这样一个时间点：站点2收到消息a但未收到消息q，站点3收到消息q但未收到消息a，于是各站点不一致。讨论Lamport算法是如何解决这个问题的，它是否会引起10.3节中讨论的其他互斥算法的困难？

10.3　问：Lamport算法是否考虑了P_i能自己保存Reply消息的传送情况？

10.4　对Ricart算法，

(1) 证明其实现了互斥。

(2) 如果消息没按其发送顺序到达，则算法不保证按它们请求的顺序执行临界段。这可能引起饥饿吗？

10.5　在令牌传递互斥算法中，时戳是否用来重设时钟和修正偏离？如果不是，时戳的作用是什么？

10.6　对令牌传送互斥算法，证明：

(1) 保证了互斥；

(2) 避免了死锁；

(3) 公平。

10.7　假设分布式系统有一个逻辑时钟，如果P_1的时钟在它向P_2发送消息时为12，进程P_2收到来自P_1的消息时时钟为8，则下一局部事件P_2的逻辑时钟为多少？如果接收时P_2逻辑时钟为15，那么下一局部事件时P_2逻辑时钟又为多少？

10.8　在10.3节中，分布式互斥的6个要求之一是进程在非临界段停止时不影响其他进程，假设有另外一个进程发送消息给已停止进程，并等待回答，这种死锁可避免吗？

10.9　参见图10.11(a)，解释为什么第2行不能被简单写成“request(j):=t”。

10.10　对图10.14定义的分布式死锁检测算法，证明只检测真正的死锁，并能在有限时间内检测出(提示：假设事务T_i发出一个请求，产生了循环)。

操作系统的安全性

计算机作为一种工具已成为高科技领域不可缺少的物质基础，它在给各行各业带来巨大经济效益的同时，也带来了潜在的严重的不安全性、危险性和脆弱性。一个有效可靠的操作系统意味着应具有良好的保护性能，即使安全性并不是系统设计所考虑的主要目标，也应该提供必要的保护措施，以防止因用户软件的缺陷而损害系统。系统的安全机制已成为操作系统不可分割的一部分。为此本章初步探讨操作系统的安全性。

11.1　安全性概述

11.1.1　安全性的内涵

1. 安全性

近年来，“操作系统安全”一词的内涵一直在不断地演化，在现实生活里，操作系统可能会遇到两方面的威胁：一方面是被动的，也可以认为是偶然发生的，如由各种失误、设备和设施失常、自然灾害等所致；另一方面是主动的，即是出于某种目的故意攻击使系统受到损害。

对于前者，人们一般可以采用一定的预防措施以尽量避免事故的发生或者减少事故发生后的损失，对于后者，人们必须通过加强系统的安全性去防范。所以操作系统安全的本质就是为数据处理系统采取技术和管理上的安全保护措施，以保护计算机硬件、软件和数据不因偶然的因素或者人为恶意的攻击而遭破坏，使整个系统能够连续正常可靠地运行。

系统的安全性包括以下几方面的内容：

① 保护系统内的各种资源免遭自然与人为的破坏；

② 估计到操作系统存在的各种威胁，以及它存在的特殊问题；

③ 开发与实施卓有成效的安全策略，尽可能减少系统所面临的各种风险；

④ 准备适当的应急措施，使系统在遭到破坏或攻击时能尽快恢复正常；

⑤ 定期检查各种安全管理措施的实施情况。

不同的计算机操作系统有不同的安全要求，但总的来说系统应具有如下特性：

(1) 保密性(Security)

保密性是计算机系统安全的一个重要方面，它指系统不受外界破坏、无泄露，对各种非法入侵和信息窃取具有防范能力。计算机系统的资源仅能由授权用户存取。

(2) 完整性(Integrity)

完整性技术是保护计算机系统内部程序与数据不被非法删改的一种手段，它分为软件完整性和数据完整性。在某些情况下，源代码的泄露给攻击操作系统者提供了破坏系统的途径，非法用户对软件的蓄意修改也会破坏程序的完整性；同样，数据完整性也显得非常重要，它是让存储在计算机系统中的、计算机系统间传输的数据不受非法修改或意外事件的破坏，以此保证数据整体的完整。

(3) 可用性(Availability)

可用性意味着计算机系统内的资源仅由授权用户使用，并且系统内的各种资源应随时可供授权用户使用，不应该将授权用户可合法存取的对象保护起来不让其使用。

2. 对计算机系统安全性的主要威胁

对计算机系统安全性的威胁主要来自以下 3 个方面：

(1) 偶然无意

有许多威胁是由偶然原因造成的,如设备的机能失常、不可避免的人为错误、软件的机能失常、电源故障等偶然事件所致的威胁。

(2) 自然灾害

计算机的脆弱性使计算机不能受重压或强烈的震荡，更不能受强力冲击。所以各种自然灾害对计算机系统构成了严重的威胁；另外，计算机设备对环境的要求也很高，如温度、湿度和各种化学物品的浓度均可对计算机系统造成威胁。

(3) 人为攻击

计算机存在着各种弱点，非法用户往往利用它的弱点达到他们的目的。从事工业、商业或军事情报收集工作的间谍对相应领域的计算机系统会造成很大的威胁，这些威胁都具有明显的主动性，所以他们是计算机系统安全的主要威胁源。

11.1.2　操作系统的安全性

1. 操作系统的安全性

操作系统的安全性是计算机系统安全性的关键。因为操作系统是计算机系统的核心软件，允许多道程序及资源共享，所以，非法用户攻破操作系统的防范就可以

得到计算机系统保密信息的存取权。因此，一个操作系统应具有如下功能：

① 用户身份鉴别。操作系统必须能识别请求存取的用户，并判断它的合法性，用户名和口令机制是一种最常见的身份鉴别机制。

② 内存保护。每个用户的程序都必须运行在某个非授权用户所不能存取的内存区域中。该保护将用户的存取限制在程序的内存空间某一部分上，人们可以将不同的安全性应用到用户内存空间的各个部分中。

③ 文件及 I/O 设备存取控制。操作系统必须防止非授权用户对系统资源的存取，并且对 I/O 设备也必须进行相应的保护。

④ 对一般实体进行分配与存取控制，并对其实行一定的控制与保护。这里所说的实体也叫系统的被保护体，如内存、文件、硬件设备、数据及保护机制本身。

⑤ 共享约束。资源必须对合法用户开放，共享要求系统具有完整性和一致性。

⑥ 在考虑操作系统安全机制的同时，也要确保系统用户享有公平的服务，而不出现永久的等待服务；还要确保操作系统为进程同步与异步通信提供及时的响应。

操作系统安全性在功能上的描述如图 11.1 所示。由图可知，安全性遍及整个操作系统的设计和结构中，所以在设计操作系统时应多方面地考虑安全性的要求。Saltzer,J. 和 Schroeder,M.曾提出了保密安全操作系统设计的原则：

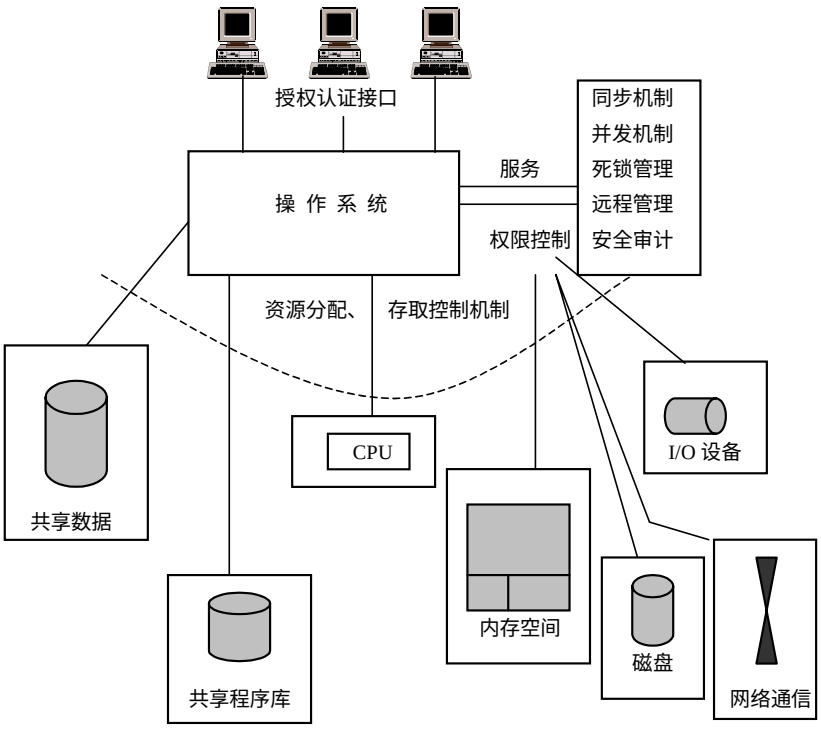


图 11.1　传统操作系统中的安全机制

① 最小权限。每个用户及程序应使用尽可能小的权限工作。这样，由入侵者或者恶意攻击者所造成的破坏会降到最低限度。

② 机制的经济性。保护系统的设计应是小型化的、简单明确的。这样，安全系统就能被完全检测其可信性。

③ 开放式设计。保护机制必须是独立设计的，它必须能防止所有潜在攻击者的攻击；也必须是公开的，仅依赖于一些保密信息。

④ 完整的策划。每个存取都必须被检查。

⑤ 权限分离。对实体的存取应该不只是依赖于某一个条件，如用户身份的鉴别加上密钥。这样，侵入保护系统的用户将不会拥有对系统全部资源的存取权。

⑥ 最少通用机制。可共享实体提供了信息流的潜在通道，系统为防止这种共享的威胁采取了物理或逻辑分离的措施。

在运用这些原则进行安全系统设计时，既要考虑系统能有效地防止来自各个方面的威胁，也要尽可能降低成本。

2. 操作系统的安全保护

Rushby 和 Randel 指出，一个计算机系统的安全性应该从如下几个方面考虑：

① 物理分离。进程使用不同的物理实体。例如，将打印机的输出按要求的不同分为不同的安全级别。

② 时间分离。将具有不同安全要求的进程安排在不同的时间运行。

③ 逻辑分离。用户的操作不受其他进程的影响，同时操作系统能对程序进行存取控制，使得程序不能存取其允许范围以外的实体。

④ 密码分离。进程以一种其他进程不了解的方式隐藏数据和计算。

以上所列出的策略是按其实现的复杂度递增及所提供的安全递减的次序列出的。在实际系统中，这几种分离策略可以相互结合，前两种分离非常直接有效，但是，将导致资源的利用率下降，所以，在实际设计操作系统时既要建立安全有效的保护机制，又要减轻系统的负担，还要保证具有不同安全需要的进程能并发执行。

一个操作系统可以在任何层次上提供如下保护：

① 无保护。此系统适合于敏感进程运行于独立的时间环境。

② 隔离。它是指并发进程相互独立地运行在它自己的地址空间，并且还有自己的文件及其他实体。操作系统限制每个进程绝对不影响其他进程的实体。

③ 完全共享和无共享。它要求设置严格的共享属性，公用的实体对所有的用户开放，而私有的实体只被它的所有者使用。

④ 存取权限的保护。它是指在特定用户和特定实体上实施存取控制权限，设置

存取控制表。在存取控制的保护中，操作系统检查每个存取的有效性，保证只有合法的存取才能发生。

⑤ 权能共享的保护。它是存取权限保护的扩展，这种形式的保护允许操作系统为实体动态地建立共享权限，共享的程度依赖于属主或主体、实体等。

⑥ 实体的使用限制。这种形式的保护不仅限制对实体的存取，而且限制存取后的使用。例如，用户可能被允许查看某个文件，却不能复制它；又如，用户能存取数据库中的数据，但不能查看特定的数据项。

这些保护措施也是按其实现难度、保护性能的递增顺序排列的，操作系统可以结合上述的安全措施形成多种层次、多种程度的安全机制。

11.1.3　安全性级别

美国国防部把计算机系统的安全从低到高分为 4 等(D、C、B、A)和 8 级(D1、C1、C3、B1、B2、B3、A1、A2)，从最低级(D1)开始，随着级别的提高，系统的可信度也随之增加，风险也逐渐减少。

1. D 等——最低保护等级

这一等只有一个级别 D1，又叫安全保护欠缺级，符合该级别的系统几乎没有什么安全性，它不对用户进行身份验证，任何人都可以使用计算机系统中的任何资源，不受任何限制，并且硬件系统和操作系统非常容易被攻破。常见的达到 D1 级的操作系统有：DOS、Windows 3.X、Windows 95/98(工作在非网络环境下)、APPLE 的 SYSTEM 7.X。

2. C 等——自主保护等级

这一等分为两个级别(C1 和 C2)，它主要提供自主访问控制保护，并具有对主体责任和其初始动作审计的能力。

C1 级称为自主安全保护级。它支持用户标识与验证、自主型的访问控制和系统安全测试；它要求硬件本身具备一定的安全保护能力，并且要求用户在使用系统之前一定要先通过身份验证。自主安全保护控制允许网络管理员为不同的应用程序或数据，设置不同的访问许可权限。C1 级保护的不足之处是用户能将系统的数据随意移动，也可以更改系统配置，从而具有与系统管理员相同的权限。常见的达到 C1 级的操作系统有：UNIX、Xenix、Novell3.X、Novell 4.X 或更高版本以及 Windows NT 等。

C2 级称为受控安全保护级，它更加完善了自主型存取控制和审计功能。C2 级针对 C1 级的不足之处做了相应的补充与修改，增加了用户权限级别，用户权限的授权以个人为单位，授权分级的方式使系统管理员按照用户的职能对用户进行分组，统一为用户组指派能够访问某些程序或目录的权限。审计特性是 C2 级系统采用的另一种提高系统安全的方法，它将跟踪所有的与安全性有关的事件和网络管理员的工

作。通过跟踪安全相关事件和管理员责任，审计功能为系统提供了一个附加的安全保护。常见的达到 C2 级的操作系统有：UNIX、Xenix、Novell3.X、Novell 4.X 或更高版本以及 Windows NT 等。

3. B 等——强制保护等级

这一等分为 3 个级别(B1、B2、B3)，它主要要求客体必须保留敏感标记，可信计算机利用它去施加强制访问控制保护。

B1 级为标记安全保护级。它的控制特点包括非形式化安全策略模型，指定型的存取控制和数据标记，并能解决测试中发现的问题。B2 级支持多级安全，多级安全是指将安全保护措施安装在不同级别中，这样对机密数据提供了更高级的保护。

B2 级为结构化保护级。它的控制特点包括形式化安全策略模型，并兼有自主型与指定型的存取控制，同时加强了验证机制，使系统能够抵抗攻击。B2 级要求为计算机系统中的全部组件设置标签，并且给设备分配安全级别。

B3 级为安全域级。它满足访问监控要求，能够进行充分的分析和测试，并且实现了扩展审计机制。B3 级要求用户工作站或终端设备，必须通过可信任的途径连接到网络中，并且它还使用了硬件保护方式，实现了安全系统的存储区数据的保护。

4. A 等——验证保护等级

它使用形式化安全验证方法，保证使用强制访问控制和自主访问控制的系统，能有效地保护该系统存储和处理秘密信息和其他敏感信息。A 等又分为两级(A1、A2)。

11.2　安全保护机制

本节将从进程支持、内存及地址保护、存取控制、文件保护和用户身份鉴别等方面介绍操作系统的安全保护机制。

11.2.1　进程支持

对操作系统安全性的基本要求是，当受控路径执行信息交换操作时，系统能够使各个用户彼此隔离。所有现代操作系统都支持一个进程代理一个用户的概念，并且在分时和多道程序运行的系统中，每个用户在自己的权限内都可能会有几个同时运行的进程。

由于多道程序运行是多用户操作系统安全性的中心问题，所以进程的快速转换是非常重要的。在同时支持多个用户的系统中，进程转换的开销会对系统性能产生重大影响。如果这种影响很大，那么软件开发人员将尽可能地避开进程的转换。使进程转换次数减至最小的一个常用方法是，对于几个同时请求的用户使用同一个进

程。但是，多用户复用一个进程将大大地降低系统的安全性，这时硬件的进程隔离机制将不再适用于用户隔离。

为描述和控制进程的活动，系统为每个进程定义了一个数据结构，即进程控制块 PCB，系统创建一个进程的同时就为它设置了一个进程控制块，用它去对进程进行控制和管理，进程任务完成了，系统回收其 PCB，该进程就消亡了。系统将通过 PCB 而感知相应的进程，进程控制块 PCB 是进程存在的惟一标志。进程控制块 PCB 包含了进程的描述信息和控制信息。

11.2.2　内存及地址保护

多道程序中的一个最明显的问题是防止一道程序在存储和运行时影响到其他程序。操作系统可以在硬件中有效使用硬保护机制进行存储器的安全保护。现在最常用的是界址、界限寄存器、重定位、特征位、分段、分页和段页式机制。

1. 界址

最简单的内存保护机制是将系统所用的存储空间和用户空间分开。界址则是将用户限制在地址范围的一侧的方法。在这种方法中，界址被预先定义为内存地址，以便操作系统驻留在界址的一边而用户使用另一边的空间，如图 11.2 所示。这种实现限制太强，固定的空间被留作操作系统使用，若所需空间小于这一数目，则留下的空间将被大大地浪费掉，而操作系统不能超出界址的范围。

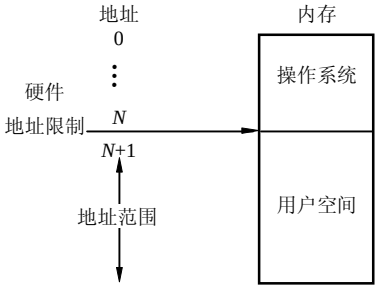


图 11.2　固定界址

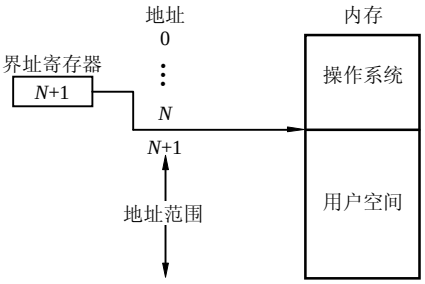


图 11.3　可变界址寄存器

另一种实现方法是使用硬件寄存器，通常称作界址寄存器，它包含了操作系统结束处的地址。在用户程序每次产生一个修改数据的地址时，该地址自动地和界址进行比较，若地址比界址大，也就是说地址在用户区，则执行指令，若小于界地址，说明在操作系统区，则产生错误。界址寄存器的使用如图 11.3 所示。

界址寄存器仅在一个方向实施保护，操作系统能保护单用户，但界址不能保护一个用户免遭其他用户的侵犯。并且，用户不能识别程序的特定区域是否是不可侵入的。

2. 重定位

重定位(Relocation)是将程序的起始地址视为从 0 开始，并在程序装入到内存时改变所有和实际地址有关的地址的过程。一般，只需将常数重定位因子加到程序的每个地址上，这个因子则是赋给程序的内存起始地址。界址寄存器可以作为硬件重定位设备，每个程序的地址加上界址寄存器的内容，这样，不仅做到了重定位而且保证用户程序不能存取比界址地址小的单元，从而保证了操作系统的安全性。

3. 基址/界限寄存器

在两个或多个用户情况下，任何一方都不能预先知道程序将被装入到内存的什么地址去执行，系统通过重定位寄存器提供的基址来解决这一问题。程序中所有的地址都是起始于基地址的位移，由此可见，基地址寄存器提供了向下的界限，而向上的地址界限由谁来提供呢？系统引进了界限寄存器，其内容作为向上的地址界限。于是每个程序的地址被强制在基址之上，界限地址之下。通过这种方法，程序的地址完全局限于基址及界限寄存器之间的空间内。借助于一对基址/界限寄存器，完全可以保护用户而免遭其他用户干扰，也可避免受任何其他用户程序错误的影响，用户地址空间内的不正确地址也只能影响到程序本身，如图 11.4 所示。

还有一种方法是，使用两对基址/界限寄存器，一对用于程序的指令，对其指令进行重定位和检查；另一对用于数据空间，对所存取的数据重定位和检查。如图 11.5 所示，两对寄存器的使用虽不能保证程序不出错，但能将数据操纵指令的影响限制在数据区，并且能将程序分为两个可分别重定位的块。相仿也可以引入 3 对寄存器(一对用于指令，一对用于只读数据，另一对用于修改数据)。

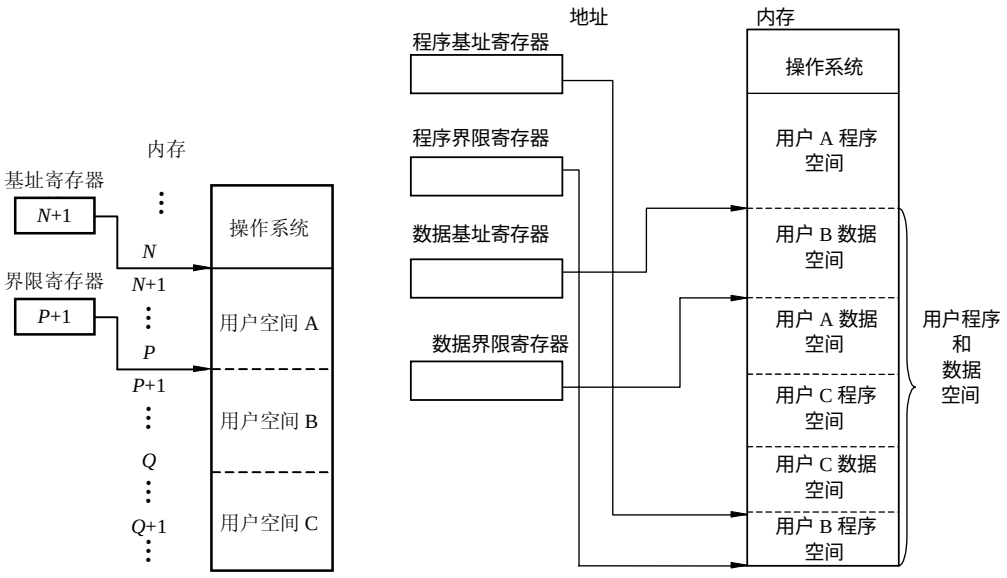


图 11.4　基址/界限寄存器对

图 11.5　两对基址/界限寄存器

特征位
字　符

R	010011
RW	013700
X	…
W	…
R	40912A
RW	C33FFE

R=只读
X=只执行
RW=读/写

图 11.6　特征位结构的例子

4. 特征位结构

下面介绍内存地址保护的另一种方法——使用特征位结构，即在机器内存的每个字中都有一个或多个附加位表示该字的存取权限，这些存取位仅能被特权指令设置。每次指令存取该单元时都对这些位进行检查。例如，在某一内存单元可能被保护为可执行权限，而另一单元可能被保护为只读权限，还有的单元被保护为可写权限。如图 11.6 所示，两个相邻单元具有不同的存取权限，这样，通过使用附加特征位将不同类的数据项分开，形成数据在内存中的保护。这一保护技术已被 BURROUGHS 公司的 B6500-7500 系统、IBM SYSTEM/38 系统使用。

5. 分段、分页和段页式

一般，程序可以被划分为许多具有不同存取权限的块，每块具有一个逻辑实体，它的所有代码和数据之间存在联系。这样的块称为段，且给予惟一的名字，段中的代码或数据项被编址〈段名，偏移地址〉。从逻辑上讲，程序员将程序看做一系列段的集合，段可以分别重定位，允许将任何段放在任何可用的内存单元内。操作系统通过在段表中查找段名以确定其实际的内存地址，无疑用户程序并不知道程序所使用的实际内存地址，它也无需知道与〈段名，偏移量〉有关的实际地址。这种地址隐藏对操作系统来说是很有意义的：

其一，操作系统可以将任何段移到任何内存单元中。由于所有地址引用都是通过操作系统查找段表进行转换，因此，在段移动时，操作系统只需修改段表中的地址即可。

其二，若段当前未使用的话，可以将其移出主内存，并存入辅存中，这样可以让出存储空间。

其三，每个地址引用都传送给操作系统处理，以便使系统有时间进行安全保护检查。

程序分段是通过硬件和软件共同完成的，因此，很容易将特定级别的保护和特殊的段联系起来，并且让操作系统和硬件在每次存取该段时检查该保护。从保护的角度来看，程序分段是通过地址引用进行保护检查的，且允许不同层次的数据项被赋予不同的保护级别，这样，多个用户都可共享存取段，但具有不同的存取权限。

这仅是从内存及地址保护来看分段方法的优点，实际上，程序分段的实现在有效性方面存在两大缺陷。首先，段名不适合于指令编码，因此，操作系统对表中段

名的查找会降低程序运行速度；其次，由于段的尺寸不一样会使内存中出现碎片，一段时间后，单从内存管理看，分段管理未使用的空间碎块就会使内存的利用率严重下降，当然，这可以采用紧凑内存空间的方法解决。

和程序分段相对应的是分页。从保护的角度来看，分页可能有一个严重的缺陷，它和分段不同，分段有可能将不同的段赋予不同的保护权限(如只读或只执行)，可以在地址转换中很方便地解决保护问题，而使用分页由于没有必要将页中的项看做整体，因此，不可能将页中的所有信息置为只读或只执行属性。

为了获得分段在逻辑上的优点和分页在管理存储空间方面的优点，采用了段页式存储管理的方法。该方法允许程序员将程序划分为一系列逻辑段，然后将每个段划分为固定大小的页，这种方法保留了段的逻辑整体特性并允许为段提供不同的保护，无疑，每个地址都增加了一次转换(参见第 5 章)。

11.2.3　存取控制

在计算机系统中，安全机制的主要目的是存取控制，它包含 3 个方面的内容：首先是授权，即确定可给予哪些主体存取实体的权力；其次是确定存取权限；最后是实施存取权限。

存取控制实现以实体保护为目标，内存保护是实体保护的特殊情况。随着多道程序概念的提出，可共享实体种类及数目在不断增加，需要保护的实体也不断地扩充，例如，内存、外设上的文件和数据，内存中的执行程序，文件目录，硬设备，数据结构，操作系统使用的表，指令、口令及用户身份鉴别机制，保护机制本身等。存取是保护机制的关键，对于一般实体而言，存取点的数目可能会很大，因此，所有的存取不可能通过某一中心授权机制进行，并且存取的种类也不只限于简单的读、写或者执行操作，而且存取都是通过程序来完成的，所以程序可被看做是存取的媒介。

存取控制机制对实体保护实现的具体过程是检查每个存取，拒绝超越存取权限的行为，并防止撤权后对实体的再次存取；实施最小权限原则，其描述的是主体为完成某些任务必须具有最小的实体存取权限，并且不能对额外信息进行存取；存取验证除了检查是否存取外，还应检查在实体上所进行的活动是否适当。下面将进一步讨论几种一般实体的保护机制。

1. 目录

用目录机制实现实体保护是一种简单的保护工作方式。这像文件目录一样，每个文件有一个惟一的文件主，它拥有大多数存取权限，并且还可以对文件授权、撤权。

为了防止伪造存取文件，系统不允许任何用户写入文件目录。因此，操作系统必须在文件主命令的控制下维护所有的文件目录，禁止用户直接对目录存取，但用

户可以通过系统进行合理的目录操作。目录保护机制的实现很简单，系统中的每个用户使用一张表，表中列出了该用户允许存取的所有实体，某些共享实体对用户都是可存取的(如一些子程序库、用户公用表)，每个用户目录都必须记录它，无论用户是否打算存取。图 11.7 所示的是文件目录的一个例子。

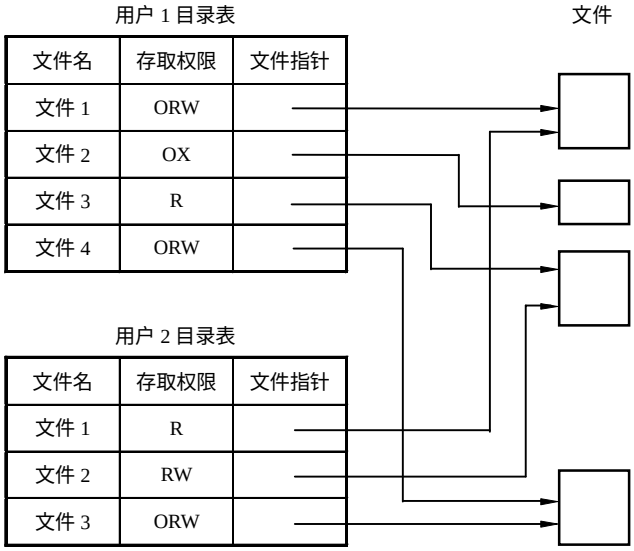


图 11.7　目录控制表

此外，授权和撤权也是存取控制中的一个重要的问题。若文件主 A 传递给用户 S 读文件 1 的权限，则必须在 S 的目录中记录文件 1。这就是说，A 和 S 具有相同的可信任级别。A 也可以收回对 S 的存取权限，这时，操作系统只需简单地删除 S 对文件 1 的存取权限(只要删除该目录下一个记录即可)。若 A 要撤消所有用户对文件 1 的存取权限，则操作系统必须在每个目录中搜索文件 1 的记录，这在大型系统中将是非常费时的。并且如果 S 又将存取权限传递给其他用户，则有可能会造成撤权的不彻底的后果。

别名问题是目录存取控制中值得注意的另一个问题，它使目录机制受到限制。例如，A、B 两个文件主可能有不同的文件都被命名为文件 1，同时可以被 S 存取，而在 S 的目录中不能有两个相同的文件名的实体。这样，S 就不能惟一地标识来自 A(或 B)的实体，解决的方法之一是将文件主名变为文件名的一部分，如 A:文件 1，B:文件 1 的形式；方法之二是允许 S 使用 S 目录下惟一的名字命名，对于实体文件 1 来说，S 可能具有不同的对文件 1 的存取权限集，一个使用文件 A，另一个使用文件 1。允许使用别名就会产生多次请求的可能，而每次请求并非一样。因此，用目录的方法对大多数实体保护机制来说是可行的。

2. 存取控制表

存取控制表是用来记录所有可存取该实体的主体和存取方式的一类数据结构。每个实体都对应一张存取控制表，表中列出所有的可存取该实体的主体和存取方式。这和目录表是不同的，目录表是由每个主体建立的，而存取控制表则是由每个实体建立的，如图 11.8 所示。

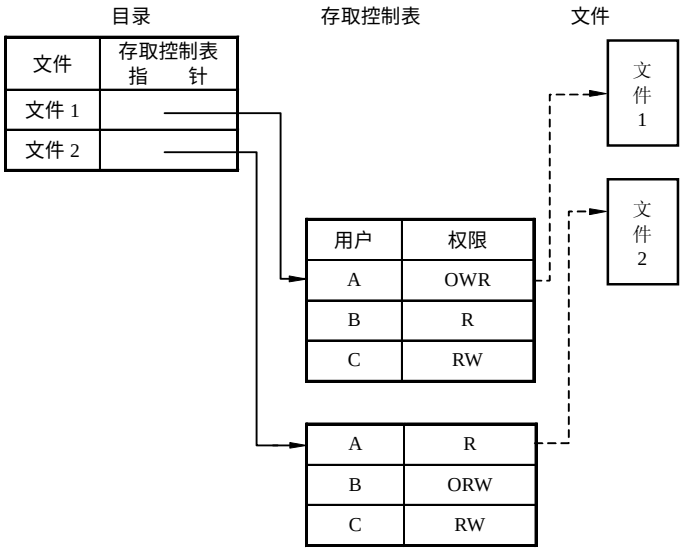


图 11.8　存取控制表

例如，若主体 A 和 B 同时存取实体文件 1，操作系统仅需维护文件 1 的存取控制表，表示 A 和 B 具有对文件 1 的存取权限。存取控制表可以对用户具有一般的缺省记录，这样，特定用户可以具有显式的权限，而所有其他的用户则具有缺省的权限集，通过这种方式，公共文件或程序就可以被系统中所有可能的用户所共享而不必在每个用户的目录下去记录该实体，使得公共实体的控制表非常小。UNIX 系统中采用的就是这种方式建立的存取权限，表中的每个用户属于不同的类，即文件主、同组用户、其他用户。每个文件都有一个文件主，由他创建这个文件，此外，某些用户可能和文件主有密切关系，同属于一个用户组，可以共享该文件，具有类似的存取权限，除了这两种身份之外的所有用户都属于其他用户。

3. 存取控制矩阵

将目录看做是单个主体能存取的实体表，而将存取表看做是记录能存取单个实体的主体表。所有的数据在这两种表达中都是等价的，区别只在于它们应用的场合不同。

如果将存取控制表用另一种形式表示，则可以形成一张行用于表示主体，列表

示实体的存取矩阵，每个记录则表示该主体对实体的存取权限集。不难看出，这样的存取控制矩阵是个稀疏矩阵，大多数主体对大多数实体并无存取权限，它可被表示为一个三维向量〈主体，实体，权限〉，如图 11.9 所示。通常搜索一个数目很大的三维向量是非常低效的，因此，实际实现中很少采用这种方法。

文件 / 用户	文件 1	文件 2	文件 3	文件 4	文件 5
用户 1	ORW		W	R	
用户 2	RW		R		X
用户 3		R		OX	

图 11.9　存取控制矩阵

4. 权能

权能是一个提供给主体对实体具有特定权限的不可伪造的标志。它的原理是主体可以建立新的实体并指定在这些实体上所允许的操作。例如，用户可以创建文件、数据段或者子进程这样的新实体，而且可在这些实体上指定可接受的操作类型(读、写或执行)，也可建立全新的实体(如新的数据结构)并定义以前未定义的存取类型(如授权、传递等)。它是一张允许主体在某一实体上完成特定类型存取的凭证，并且该凭证是不可伪造的，系统不是直接将此凭证发给用户，而是由操作系统以用户的名义拥有所有凭证。只有在通过操作系统发特定请求时，用户才建立权能，每个权能同样也标识可允许的存取。

对某一实体而言，它可能具有转移或传播的存取权限。具有这种权限的主体可以将其权能副本传递给其他实体。每个这样的权能同样具有一张许可存取类型的表。例如，进程 A 能将权能的副本传递给 B，B 于是能将副本传递给 C。B 为了防止权能的继续传播(这就防止了存取权限的滥用)可在它发送给 C 的权能中略去转移的权限，这时 B 仍然可以将特定的存取权限传送给 C，但并没有授予 C 具有转移权能的权限。

进程运行在它的作用域中，这些作用域指存取的实体集，如程序、文件、I/O 设备等。当进程运行时可能会调用一个子过程，将它能存取的某些实体作为参数传递给该子过程，而子过程的作用域不一定和调用它的过程的作用域一样，实际上，调用过程可能仅将实体的一部分传递给子过程，而子过程可能具有其调用过程不能存取的其他实体，调用过程也只能将它的存取权限的一部分传递给子过程。例如，一个过程可能只将对某一特定数据值的读权限传送给子过程，而未将修改权限传送给子过程。由于每个权能都标识了作用域中的单个实体，因此，权能的集合定义了作用域。当进程调用子过程并将特定实体传送给子过程时，操作系统形成一个由当前

过程的所有权能组成的栈，并且为子过程建立新的权能。

从使用的角度来讲，权能是一种在程序运行期间直接跟踪主体对实体存取权限的方法。权能也可以被备份到一张诸如存取矩阵或存取控制表中，每次进程请求使用新的实体时，操作系统检查实体及主体控制表，判断该实体是否是可存取的，若是，则操作系统就为该实体建立权能。

权能必须存放在内存中，而普通用户不能存取。一种实现方法是将权能存于用户段表未指向的段中，或是将它们置于一对基址/界限寄存器所保护的内存区域中；另一方法是给权能赋予某一特征位。在程序运行期间，只获取被当前进程所存取的实体权能，这样便提高了对存取实体权能检查的速度。权能的一个缺陷是它能被收回，若发行主体收回权能，就不能在所收回权能下进行存取。因此，在权能被回收时，操作系统应能跟踪到应该删除的权能，彻底予以回收，并且删除那些不再活跃的用户的权能。

5. 面向过程的存取控制

存取控制的目标是，既控制对某一实体的存取主体又控制对该实体的存取方式。对大多数操作系统来说，对读和写的存取控制相对简单一些，但对更为复杂的控制则很难实现。借助于面向过程的保护，即借助一个对实体进行存取控制的过程(例如，对操作系统所提供的用户身份的鉴定)，在实体周围形成一个保护层，达到仅允许特定的存取之目的。

面向过程的保护特性是过程可以通过可信任接口请求存取实体，例如，无论用户或是通用操作系统例程都不允许直接存取一个合法用户表。惟一的存取方法可以通过 3 个过程实现：增加一个用户，删除一个用户以及检查特定的名字是否对应于一个合法用户。这些过程可以通过它们自身的检查保证调用是合法的。面向过程的保护实现了信息隐藏，对实体控制的实施手段仅由控制过程所知，但当实体频繁调用时，将会影响系统速度。

对存取控制的研究已经由简单到复杂正在逐步进行，随着所提供的机制将会变得更为灵活，而实际操作系统中在采用存取控制时，仍需在它的简单性和功能性之间权衡。

11.2.4　文件保护

文件系统是操作系统的又一个重要部分，文件系统的结构体现了操作系统的特点，也极大地影响操作系统的性能，是操作系统设计中的基础与难点。文件是操作系统处理的资源，文件存放在磁盘、磁带这些可随机存取的辅助存取介质上。因此，对文件系统的安全保护机制就分为对文件系统本身和对文件存储载体的安全保护。

1．基本保护

所有多用户操作系统都必须提供最小的文件保护机制，以防止用户有意或无意

对系统文件和其他用户文件的存取或修改，这是最基本的保护。随着用户数目增多，保护模式的复杂性也随之增加。

2．全部/无保护

在最初的操作系统中，文件被默认为是公用的，任何用户都可以读、修改或删除属于其他用户的文件，这是一种基于信任的保护原则，用户希望其他人不读或修改自己的文件，同样其他的用户也是这样，而且用户只能通过名字存取文件，并假定用户仅知道那些他们能合法存取的文件的文件名。然而，对某些特定文件是敏感的，一般借助于口令的方式进行文件的保护，将口令用于控制所有存取(读、写、删除)，或仅控制写及删除存取，这种干预是通过系统操作员来完成的。

3．组保护

组保护主要用于标识一组具有某些相同特性的用户。通常的做法是，将系统中所有用户分为 3 类：普通用户、可信用户、其他用户。UNIX 操作系统、VMS 系统中都采用这种形式的保护方法。所有授权的用户都被分为组，一个组可以是一些在一个相同项目上工作的成员、部门、班级或单个用户，共享是选择组的基础，组中成员具有相同的兴趣并且假定和组中其他成员共享文件，一个用户不能属于多个组。

在建立文件时，普通用户为本用户、组中其他用户以及其他用户定义存取权限。一般而言，存取权限的选择是一个有限集，如{读、写、执行、删除}。对于特殊文件，普通用户可能只允许其他用户具有只读存取权限，组中用户具有允许读和写的存取权限，而用户本身具有所有的权限。这种分组的方法实现很简单，使用两个标识符，通常是数字，标识每个用户的 ID 及组 ID。这些标识符保存在每个文件的文件目录记录中，并在用户录入系统时由操作系统得到。因此，检查用户是否可以存取某一文件只需检查其组 ID 是否和要存取的文件所属的组 ID 一致。

这种保护模式避免了全部或无保护模式的缺点，并且也容易实现，但也存在着缺陷。首先，单个用户不能属于两个组，否则会产生二义性。例如，用户 1 与 2 属于同一个组，而用户 1 又与 3 在另一个组，若用户 1 将某一个文件设定为组内只读，那么，它的组内只读属性到底对两组中的哪个用户有效呢？因此，操作系统限制每个用户一般只能属于特定的组。其次，为克服每一组的限制，特定的人可能会有多个账号，若允许的话，其效果则类似于多用户。例如，用户 1 得到两个账号 A1 和 B1，它们分别属于两个不同的组，在 A1 下开发的任何文件、程序和工具仅在它们可用于整个系统时才可被 B1 使用。这种方法将会导致账号猛增、文件冗余，限制了普通用户的文件保护，而且使用不便。并且，文件只可在组内或其他用户间共享，用户可为每个文件标识共享对象。

4. 单许可保护

单许可保护是指允许用户对单个文件进行保护，或者进行临时权限设置。下面的保护方法将允许对单个文件进行保护。

(1) 口令保护机制

它是用户将口令赋给一个文件，对该文件的存取只限于提供正确口令的用户，口令可用于任何存取或是只用于修改。

文件口令同样也带来了类似于身份鉴别口令(见 11.2.5 节)一样的问题，如口令丢失、泄露、取消。一旦用户的口令丢失则必须重新设置新口令；口令泄露后，文件就不安全了，若用户改变口令则可以重新保护文件，并且必须将新口令通知所有其他合法用户。当用户撤回某一用户对某一文件的存取权时，也必须改变口令，这和泄露所产生的问题类似。

(2) 临时获得许可权

UNIX 操作系统提供了另一种许可模式，它将基本保护设为 3 个层次：文件主，同组用户，其他用户。其中重要的是设置用户标识 SUID 许可，若对要执行的文件设置这种保护，则保护级别为文件主而非执行者。例如，用户 1 拥有一个文件并允许用户 2 执行，那么，用户 2 在执行该文件时应当拥有用户 1 的保护权限，而非自己的保护权限。

这种独特的许可方式很有用，它允许用户建立一个只能通过特定过程才能存取的数据文件。如，用户 1 有一个应用程序，并且设置用户标识 SUID，使得只有他自己才有存取权，如果用户 2 运行该程序，则仅在程序运行期间临时获得用户 1 的许可权，当用户 2 退出该程序时，重新获得的是自身的存取权，而放弃用户 1 给予的存取权。借助于 SUID 特性，口令改变程序应为系统拥有，因此，它有权存取整个系统的口令表。改变口令的程序也应具有 SUID 的保护，以便在普通用户执行它时以一种限制的方式代表该用户修改口令文件。

5．单实体及单用户的保护

由于上面描述的保护模式存在着限制，将单许可保护方式进行扩展，就形成了单实体及单用户保护。利用存取控制表或存取控制矩阵可以提供非常灵活的保护方式。但是它也存在着缺点，用户需要允许不同的用户存取不同的数据集，这样的用户必须指定每个用户所存取的数据集，随着新的用户加入，该用户的特殊存取权必须由其他相应的用户指定。

DEC 公司的 VAX VMS/SE 操作系统提供存取控制表，用户可为任何文件建立存取控制表，指定谁能存取该文件，以及所拥有的存取类型。对位于不同组中的用户，系统管理员使用普通标识符建立一个有效组，允许其中的成员存取某个文件。如，在一个软件开发小组中，用户 1 属于第一组，用户 2 属于第二组，用户 3 和 4 属于第三组，系统管理员可以定义一个普通标识符 SOFT-PROJ，它仅包含这四个用户。用户可以允许普通标识符中的成员存取某一文件，而不让组中的其他用户存取该文件，存取控制表指定用户、组和普通标识符，这就形成了一种实体保护。在 VMS 中，可将存取控制表应用于特定设备或设备类型，仅允许用户存取特定的打印机、

电话线路、调制解调器、限制网络存取等。IBM MVS 操作系统增加了资源存取控制设施，为其数据集提供了类似的保护方式，系统允许文件主给文件赋予缺省保护，然后为特定用户赋予存取保护类型。

11.2.5　用户身份鉴别

操作系统的许多保护措施都是基于鉴别系统中的合法用户的，身份鉴别是操作系统中很重要的一个方面，也是用户获得权限的关键。现在最常见的用户身份鉴别机制是口令。

1. 使用口令

(1) 口令的使用

口令是计算机和用户双方知道的某个关键字，是一个需要严加保护的对象，它作为一个确认符号串只能由用户和操作系统本身识别。

口令是相互认可的编码单词，并保证只被用户和系统知道。在某些场合，由用户选择口令，而在另外一些场合，则由系统分配给用户，并且口令的长度及格式也随着系统的不同而不同。

口令的使用很简单，用户输入某一标识，如姓名或 ID 号，这一标识可以公开或者很容易猜到，因为它并没有提供系统安全。系统于是请求输入口令，若口令与口令文件中该用户的口令匹配，则系统认可该用户，否则，系统会提示再次输入口令。

(2) 口令的安全性

由于口令所包含的信息位很少，因此，它作为一种保护是很有限的，下面介绍几种保证口令安全性的方法：

① 使穷举攻击不可行。在穷举攻击中，攻击者尝试所有可能的口令，可能的口令数目依赖于特定计算机系统的实现。若口令由 A~Z 这 26 个字母和 0~9 这 10 个数字组成，其长度在 1~8 个字符之间，一个字符可以有 36 种可能，则 8 个字符有 36^8 种可能。于是整个系统会有 $36^1+36^2+\cdots+36^8$ 种可能。这个数目很大，测试会花去大量的系统时间，实际上是不可行的。

再者，搜索单个特定口令并不需要尝试所有这些口令，如果口令分布均匀，则查找任何特定的口令的期望搜索次数为口令数目的一半，所以在选择口令时，应尽量使口令分布不均匀。但是，即使口令分布不均匀，也由于人的思考记忆方式和心理因素，常常选择那些简单并有规律的口令，这样攻击者就利用一般人的习惯有目的地尝试去获取口令，为了使穷举攻击不可行，一般要选择一些使人难以猜到的口令。

② 限制明文系统口令表的存取。为了验证口令，系统必须采用将记录和实际的口令相比较的方式，攻击者可能攻击的目标是口令文件，借助于系统口令表可以准确无误地确定口令。

在某些系统中，口令表是一个文件，实际上是一个由用户标识及相应口令组成

的列表。显然，不能让任何人都能访问到该表，为此，系统采用了不同的安全方法来保证。保护口令表的安全设施是使用强制存取控制，限制它仅可为操作系统存取，这样仍然不太安全，因为并非所有操作系统模块都需要或应该使用该表，一个较好的方法是让系统口令表只允许那些需要存取该表的模块存取。

③ 加密口令文件。加密口令文件表较为安全，这样，读文件内容对入侵者将毫无用处，一般使用传统加密及单向加密这两种加密口令的方法。

使用传统加密方法，整个口令表被加密，或只加密口令列。当接收到用户口令时，所存取的口令被解密，使用这种方法在某一瞬时会在内存中得到用户口令的明文，任何人只要得到对所有内存的存取权就可获得它，显然这是一个很明显的缺陷。

另一个较安全的方法则是使用单向加密，加密方法相对简单，解密则是用加密函数。在用户输入口令时，口令也被加密，然后将两种加密形式进行比较，若两种形式相同，则鉴别成功。使用单向加密的口令文件可以以一种可见的形式保存。事实上，在操作系统中，除非在口令表上安装特殊的存取控制机制，任何用户都可以读口令表，口令表的后备拷贝也将不再是一个问题。

以一种伪装的形式保存口令表将在很大程度上保证安全性。存取仍然被限制为那些具有合法需要的进程。但是，保密表的内容和表的存取控制提供两层安全，若某人成功地侵入了外层安全圈，他仍然无法获取有用信息。

(3) 口令的挑选准则

选择合适的口令是很重要的，一般，选择的口令应使他人很难猜到。由于不同环境的安全需求也影响着口令的选择，所以对口令的选择有如下准则：

① 增加组成口令的种类。不仅仅使用英文字母 A~Z，增加字符种类可增加口令的破译的难度。

② 挑选长口令。当口令的字符数超过 5 时，其组合数将呈爆炸型增长，口令越长，被破译的可能性就越小。

③ 避免使用常规名称、术语和单词，尽量选择不太规范的单词。挑选非单词，会使攻击者在穷举搜索上花费较大精力。

④ 定期更换口令，也可以使用一次性口令。定期更换口令将使得攻击者通过原来的口令表或强攻击加密口令表等方法获取口令的难度增大。一次性口令用于身份鉴定是相当保密的，因为口令在每次使用后就作了更换，攻击者即使截取到口令也是无用的。

(4) 身份鉴定的缺陷

在口令身份鉴定中，假定知道口令的拥有者，则口令有可能被猜中，通过询问得到，或在该用户使用时窃取，那么，口令的真实性需要有更令人信服的证据。

在常规系统中，证据是单方面的，系统需要用户特定的标识，而用户必须信任系统，因此，入侵者可以采用假冒登录和屏幕提示，待程序建立后，故意离开计算

机，等待合法用户登录，此时，假登录程序记录用户输入口令及标识，并保存在入侵者指定的文件中，同时提示用户登录错误而退回到系统真正的登录过程，这种攻击是一种特洛依木马，用户不会怀疑他的口令和标识已经泄露和被窃取。

要防止这类攻击，用户应当确信进入系统的路径每次都被初始化。此外，用户也要确认系统的合法性，在此之前不会输入保密数据或口令。比如，系统可以提示用户最近登录的时间，用户在输入口令之前先证实日期和时间的正确性，确认在这之前登录中没有口令被截取，这时用户才输入该时间标志和口令。

2. 其他身份鉴别机制

有些物理设备可以用来辅助身份鉴别，这些物理设备包括笔迹鉴别器、声音识别器及视网膜识别器。这些都是利用不可伪造的用户特征去鉴别用户的。尽管这些设备很昂贵且还处于实验阶段，但在极端保密的环境下仍然十分有用。

更为普通的安全环境中需要将登录和特征结合起来，可以将用户限制在特定物理位置的终端或特定的时间内使用机器。对一个非法用户，无任何正当理由企图存取某一文件，在系统检测到这些违章存取尝试时，系统会断开和用户的会话并将其挂起，直到安全管理员清除为止。

身份鉴别是操作系统中相当重要的一个方面，因为准确的用户身份鉴别是用户获取权限的关键。大多数操作系统及计算机系统管理员，为防止非法用户存取系统资源，都采取了可行但却是极为严密的安全措施。

11.3　病毒及其防御

11.3.1　病毒概述

1. 病毒的特点

计算机病毒(简称病毒)是一种可传染其他程序的程序，它通过修改其他程序使之成为含有病毒的版本或可能的演化版本、变种或其他繁衍体。病毒可经过计算机系统或计算机网络进行扩散，也可通过存储媒介进行扩散。一旦病毒嵌入了某个程序，就称该程序染了计算机病毒了，并且这个受感染的程序可以作为传染源继续感染其他程序。

病毒主要有如下的特点：

① 病毒是一段可执行程序，具有依附性。当它依附在系统内某个合法的可执行程序上时便可执行。

② 病毒具有传染性。一旦一个程序染上了病毒，就能传染整个系统。

③ 病毒具有潜伏性。计算机病毒往往是先潜伏下来，等条件满足时才触发它，

并且这种潜伏性使病毒的威力大为增强。一旦触发条件满足，就会激活病毒的表现模块，使其实施传染操作。

④ 病毒具有破坏性。计算机病毒的破坏性有轻有重，轻者只是影响系统的工作效率，重者将会导致系统崩溃。

⑤ 病毒具有针对性。计算机病毒的发作还需要一定的环境，一种病毒并不是对所有操作系统都能传染。例如，攻击 UNIX 操作系统的病毒就不能破坏 DOS 系统，反之亦然。

2. 病毒的结构

病毒大多由 3 部分组成：引导模块、传染模块和表现模块。

病毒的引导模块负责将病毒引导到内存，对相应的存储空间实施保护，以防止被其他程序覆盖，并且修改一些必要的系统参数，为激活病毒做准备。

病毒的传染模块负责将病毒传染给其他计算机程序。它是整个病毒程序的核心，由两部分组成：一是传染条件的判断部分，另一是传染部分。

病毒的表现模块也分为两部分：一是病毒触发条件判断部分，另一是病毒的具体表现部分。

3. 病毒的分类

按照病毒的特点，可以有很多分类方法：

① 按攻击对象分类，可以分为攻击 IBM PC 机及其兼容机的病毒，攻击 MACINTOSH 系列机的病毒和攻击 UNIX 操作系统的病毒。

② 按攻击的机种分类，可以分为攻击微型机的病毒，攻击小型机的病毒和攻击工作站的病毒。

③ 按链接方式分类，可以分为操作系统病毒，源码病毒，入侵型病毒和外壳型病毒。

④ 按寄生方式分类，可以分为寄生在磁盘的引导区和寄生在可执行文件中的病毒。

11.3.2　病毒的防御机制

病毒的显著特性是传染性，病毒之所以对计算机系统构成威胁，主要是由于它的传染性，病毒的传染性主要与操作系统有关。

1．病毒的作用机理

病毒的作用机理首先是，检查是否要感染计算机，如要，就对计算机进行感染，然后，再去执行下面的命令。如不需要感染，就跳过感染过程去执行下面的指令，也就是去判断触发条件是否满足，如果不满足，就去执行正常的系统命令，如果满足，病毒的表现模块就会运行起来，产生一系列的症状。病毒的整个工作过程如图 11.10 所示。

2．操作系统与病毒传染性的关系

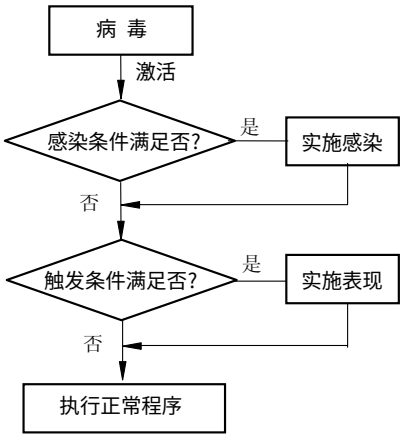


图 11.10　病毒的工作过程

病毒也是一段计算机程序，所以它的运行离不开操作系统的支持。病毒从一个系统向其他系统传染有很多途径，比较普遍的一个途径是经过磁盘传染。病毒要想感染一块磁盘，首先要把自己复制到磁盘上，但它不能以独立的文件形式存入盘中，而必须嵌入某个合法程序中，这样它才能隐藏起来而不被发现。病毒向磁盘的存储过程与操作系统是有密切关系的，一般是调用系统的磁盘操作功能来实施自身的复制，不同操作系统的磁盘操作功能是不同的，所以一般来讲，针对某种操作系统的病毒不传染其他互不兼容的操作系统。

3．病毒的防御机制

病毒的防御机制是针对病毒的运行特征而采取的层层设防、级级设防措施。病毒的传染和运行的机理在上面已有了讨论，可见，在病毒感染或运行过程中的任何一环都可以采取防御措施。

病毒防御措施通常与系统的存取控制、实体保护等安全机制配合起来，再由专门的防御程序模块来完成。防御的重点在操作系统敏感的数据结构、文件系统、数据存储结构和 I/O 设备驱动结构上。操作系统的敏感数据结构包括系统进程表、关键缓冲区、共享数据段、系统记录、中断矢量表和指针表等关键数据结构，病毒会试图篡改甚至删除其中的数据和记录，这样会使得系统运行出错。针对病毒的各种攻击，病毒防御机制采用了存储映像、数据备份、修改许可、区域保护、动态检疫等方式。

① 存储映像，是保护操作系统关键数据结构在内存中的映像，以防止病毒破坏或便于系统恢复。

② 数据备份，类似于映像，主要是针对文件和存储结构，将系统文件、操作系统内核、磁盘主结构表、文件主目录及分配表等建立副本，保存在磁盘上以作后备。

③ 修改许可，是一种认证机制，在用户操作环境下，每当出现对文件或关键结构的写操作时，都提示用户要求确认，这也是防止病毒感染传播的一种基本手段。

④ 区域保护，是借助禁止许可机制，对关键数据区、系统参数区、系统内核禁止写操作。由于一般用户不具有对这些区域的写权限，故很容易对某些实体设置标记，并不让用户知道，从而将用户限制在允许的区域内操作。

⑤ 动态检疫，是将主动性引入可防御机制，在系统运行的每时每刻，它都监视某些敏感的操作或者操作企图，一旦发现则给出提示并予以记录，它可以与病毒检测软件配合，发现病毒的随机攻击。

上述机制都是嵌入到操作系统模块中，或者以系统驻留程序实现的，对操作系统的设计来说起到了一种弥补作用，但并不能完全解决问题。

11.3.3　特洛伊木马程序及其防御

1. 特洛伊木马

特洛伊木马是一段程序，表面上在进行合法操作，实际上却进行非法操作，受骗者是程序的用户，能入侵者是这段程序的开发者。

特洛伊木马程序能成功入侵的关键是入侵者要写一段程序进行非法行为，程序的行动方式不会引起用户的怀疑，该程序应当完成某些有趣的或有用的功能，以诱导他人去使用，并且必须由受骗者去运行这段程序，最后，入侵者还要能以某种方式获取他所需的信息。

1984 年被科恩命名为“病毒“的程序，是一种通过系统和系统网络将自身传播蔓延的特殊特洛伊木马程序。它将病毒程序传染一个系统，一旦程序安装到系统上后，就会使系统受到严重的破坏，而且很难将病毒清除。

2．对特洛伊木马的防御

如果不依赖于一些强制手段，想防止特洛伊木马的破坏几乎是不可能的。但是可以采取对存取控制灵活性的限制。过程控制、系统控制、强制控制和检查软件商的软件等措施都可以降低特洛伊木马攻击成功的可能性，但是，不管是哪种技术均有它的局限性。

3．特洛伊木马与隐蔽信道

特洛伊木马程序攻击系统的一个关键标志是，通过一个合法的信息信道进行非法通信，这些信道一般是用于交互进程通信的，如文件、通信道或者是共享内存。虽然强制访问控制通过访问能够防止利用这种信道进行非法通信，但是，一个系统往往还允许进程之间利用许多其他途径进行通信，而这些途径通常是不用于通信的，并且也不受强制访问控制的保护，人们称这些通道为隐蔽信息信道，即隐蔽信道。它也可称作泄露路径，因为信息可以经过它在不经意中泄露出去，比较严重的情况是由特洛伊木马程序通过这种信道而产生的信息泄露。

系统内到处充满着隐蔽信道，只要它可以由一个进程修改而由另一个进程读取，那它就是一个潜在的隐蔽信道。隐蔽信道有两类：存储信道和时间信道。存储信道是一种这样的信道，一个进程对某客体进行写操作，而另一个进程可以观察到写的结果；时间信道是一个进程对系统性能产生的影响可以被另一个进程观察到，并且可以利用实时时钟这样的时间坐标进行测量。

11.4　加密技术

加密技术在计算机安全中得到了广泛的应用。加密就是将可理解的消息变成不可理解的消息，从而隐藏其真正含义的过程。加密可通过编码或密码进行。编码系统就是用事先定义的表或字典将每个消息或消息的一部分替换成无意义的词或词组，最简单的编码就是将字母表中的每个字符用另一个字符代替。密码就是用一个加密算法将消息转化成不可理解的密文。由于密码技术安全性较高，因此，在现代计算机和网络安全设备中就用到了这些技术。

本节仅讨论加密技术，首先讨论加密的传统方法，然后讨论一种公开密钥加密方法，最后介绍密钥的管理。

11.4.1　传统加密方法

1．传统加密过程

加密就是把明文信息利用某种加密算法加以编码以防止未授权的访问，从而实现其保密的过程。图 11.11(a)描述了传统加密的过程。明文就是指用户可理解的消息，加密后被转换成无法阅读的信息，也就是密文。加密过程包含一个算法和一个密钥。加密密钥是一个惟一的保密数字，用于对数据进行加密防止未授权者阅读。算法根据当时所用的特定密钥产生不同的输出，改变密钥会改变算法的输出。

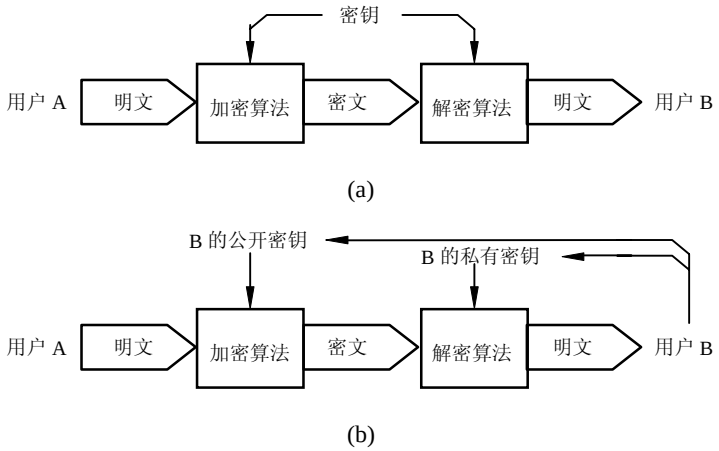


图 11.11　加密

(a) 传统加密　(b) 公开密钥加密

加密完明文后就发送密文。在接收端，用一个解密算法及一个与加密相同的密钥将密文重新转换成初始明文。传统加密的安全性主要依赖于几个因素：首先，加

密算法应该是强有力的，因此，只根据密文来解密出明文是不可能的。除此之外，传统加密的安全性还依赖于密钥的保密性，而不是算法的保密性。所以，有了密文和加密/解密算法也不可能解密出一个消息。换句话说，我们不需保密算法，只需保密密钥。

传统加密的这种特性使它被广泛运用。不需保密算法就意味着制造商可以用较低的耗费来实现数据加密算法。传统加密的安全性问题主要在密钥的安全管理。

2．数据加密标准(DES)

使用得最广的加密方法是基于数据加密标准(DES)的方法，DES 是由美国国家标准局(NBS)在 1977 年研制的一种数据加密、解密的标准方法。在 DES 中，数据被分为 64 位一组，每组使用 56 位的密钥来加密。解密时使用相同的步骤和相同的密钥。

DES 的应用越来越广泛。但 DES 的安全性也引起争议，主要是密钥的长度，有些人认为太短。下面，先大致介绍一下 DES 的历史。

NBS 在 1973 年提出了国家加密标准，从而产生了 DES。那时，IBM 正对 Lucifer 算法进行最后阶段的研究。Lucifer 方案在当时是最好的，因此 IBM 为 NBS 开发了基于 Lucifer 的数据加密算法。DES 与 Lucifer 本质相同，但有一个根本区别：Lucifer 的密钥是 128 位，而 DES 的密钥是 56 位的。

破解密文有两种基本的方法：一种是对加密算法所依赖的数学基础进行分析，然后对其攻击，一般认为 DES 可免予这种攻击；另一种是用所有可能的密钥尝试，也就是用所有可能的 56 位密钥来解密，直到得出明文。DES 密钥只有 56 位，于是只有 2^{56} 个不同的密钥——这个数不够大，而且在计算机速度越来越快时就更觉得太小了。

但由于该方法对信息进行了完完全全地随机处理，即使破译者拿到了部分原文本也无法将其完全破译，所以 DES 在最近几年仍被广泛应用，美国政府部门和大多数银行以及货币交换部门都采用此方法保护敏感的计算机信息。

11.4.2　公开密钥加密方法

传统加密的一个主要困难是要以一种保密的方式来分配密钥。对这一点最彻底的解决方法就是不分配密钥，也就是采用公开密钥加密，如图 11.11(b)所示，它是在 1976 年被首次提出的。

在传统加密方法中，加密和解密所用的密钥是相同的，这并不是一个必要条件，有可能设计出一个算法用一个密钥加密，用另一个不同的但有联系的密钥来解密；另外也有可能设计出一个算法，即使已知加密算法和加密密钥也无法确定解密密钥。因此，就需要以下技术：

① 在网中每个端系统都产生一对密钥，用来对它将接收的消息加密、解密。

② 每个端系统都将加密密钥放在公共寄存器或文件中(也就是公开密钥)，解密

密钥设为私有。

③ 如果 A 要向 B 发送消息，A 就用 B 的公开密钥加密消息。

④ 当 B 收到消息时，就用私有密钥解密。由于只有 B 知道其私有密钥，因此，没有其他接收者可以解密出密文。

公开密钥加密解决了密钥的分配问题，所有人都可以访问公开密钥，私有密钥在本地产生，不需分配。只要系统控制其私有密钥，就可以保证输入通信的安全性。任何时候系统都可以改变其私有密钥，并用其对应的公开密钥代表旧的公开密钥。

与传统加密相比，公开密钥加密的一个主要缺点是算法比较复杂。因此，在硬件的规模和耗费相当时，公开密钥加密的效率较低。

表 11.1 对传统加密和公开密钥加密的一些重要特性进行了归纳。

表 11.1　传统加密和公开密钥加密

传统加密		公开密钥加密
工作条件	• 加密和解密所用的是相同算法和相同密钥 • 发送者和接收者必须共享算法和密钥	• 加密/解密的算法，一对密钥，一个用作加密，一个用作解密 • 发送者和接收者各有一对匹配的密钥
安全性条件	• 密钥必须保密 • 没有其他信息，解密消息是不可能的 • 知道算法和密文无法确定密钥	• 两个密钥中一个要保密 • 没有其他信息，解密消息是不可能的 • 知道算法，一个密钥和密文无法确定另一个密钥

11.4.3　密钥的管理

如果用传统加密方法加密，则两个主体必须有相同的密钥，并且密钥受到保护，以免被其他人访问。另外，如果密钥被攻击者知道了，就要改变密钥以减少泄露的数据量。因此，加密系统的强度与密钥分配技术有关。密钥分配是指在两个交换数据的主体间传送密钥而不让其他人知道的方法。对于两个实体 A 和 B，实现密钥分配的方法有多种：

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 和 B 先后用了同一个密钥，则由一个主体将新密钥传送给另一个。

④ 如果 A 和 B 都与 C 有加密链接，则 C 可在加密链上向 A 和 B 传送密钥。

第①种方法和第②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节点交换数据，故这类方法可行。但对于端对端加密，密钥的传送就很困难。在分布式系统中，任一给定主机或终端在一段时间内要与大量的其他主机和终端交换数据。这样每个终端就需要大量的密钥。这个问题在广域分布系统中尤其严重。

第③种方法对链路加密和端对端加密都可行，但如果一个攻击者成功获取了一个密钥，那么所有密钥都可被获取。

第④种方法适于提供端对端加密密钥。图 11.12 给出了一种满足第④种方法的端对端加密的实现。在这种方法中，要验证以下两种类型的密钥：

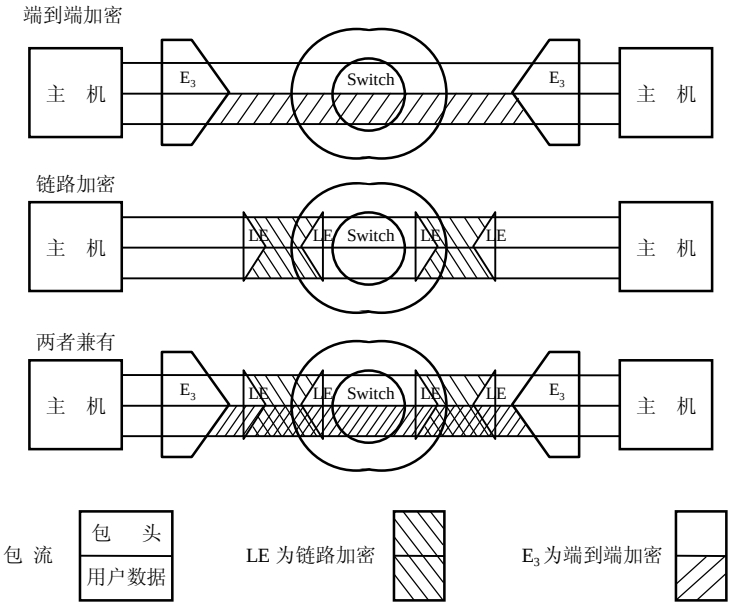


图 11.12　端到端及链路加密

• 暂时密钥。当两个端系统进行通信时，它们之间建立起一个逻辑连接。在逻辑连接期间，所有的用户数据都用一个暂时密钥加密，连接结束时，再取消暂时密钥。

• 永久密钥。永久密钥是在实体间分配暂时密钥时所用的密钥。

建立连接的步骤(见图 11.13)如下：

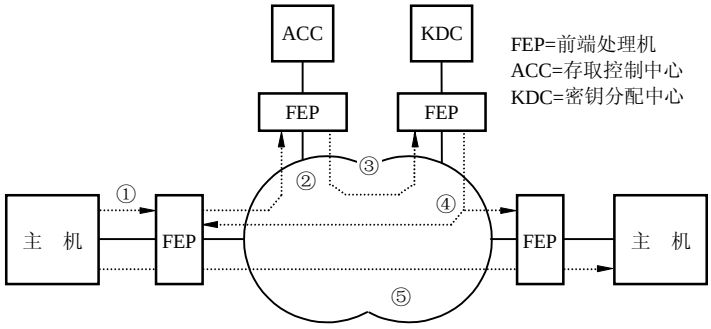


图 11.13　网络交叉的端到端加密

图中，①主机发送包请求连接
② 前端缓存包，向 ACC 要段密钥
③ ACC 同意请求，向 KDC 发命令
④ KDC 将段密钥分布到两个前端
⑤ 发送缓存包

① 当一台主机要与另一台主机建立连接时，它就发送一个连接请求包。

② 前端处理机保存该包，并将其送到存取控制中心以获得建立连接的许可。

③ 前端处理机与存取控制中心间的通信由永久密钥加密。该密钥只能被存取控制中心和前端处理机使用，存取控制中心对应每个前端处理机和每个密钥分配中心都有一个惟一的永久密钥。如果存取控制中心允许连接，就发送一个消息给密钥分配中心，请求产生暂时密钥。

④ 密钥分配中心产生暂时密钥，然后传送给两个相应的前端处理机。

⑤ 发出请求的前端处理机释放连接请求包，在两个端系统间建立连接，两个端系统间的所有数据交换都要由其前端处理机用暂时密钥加密。

对上述方法可作几种改动，存取控制和密钥分配的功能可合成一个系统，将这两个功能分开可提供较高的安全性。如果允许两个端系统任意通信，那么就不需要存取控制：当两个系统建立连接时，其中一个向密钥分配中心请求一个暂时密钥。另外，前端处理机的功能也不必由单独的装置提供，而可以合入主机中，使用前端处理机的好处是它可以使网络负载最小化。对于主机而言，前端处理机是一个包交换节点；对于网络而言，前端处理机是一台主机。

密钥分配的另一种方法是公开密钥加密。公开密钥加密与传统密钥加密相比，其缺点在于算法要复杂得多。因此，在硬件规模和耗费相当时，公开密钥方法效率较低，公开密钥可用作图 11.13 中所示的永久密钥，而传统密钥用作暂时密钥。

11.5　安全操作系统的设计

操作系统能统一管理系统内的一切硬件和软件资源，使系统能自动、协调、高效地工作，因此，操作系统往往成为攻击的目标。正因为操作系统是计算机系统中安全性的基石，所以在设计操作系统时安全性也成为考虑的一个重点。从操作系统设计者的角度来看，安全操作系统的开发要经历如下 3 个阶段：第一是模型化阶段，在开始建立一个安全操作系统之前，设计者必须知道安全性是什么，进而构造一个需保密的模型并研究达到安全性的不同方法；第二是设计阶段，在选择了安全模型之后，设计者需选择一种实现该模型的方法，必须保证从设计者到用户都相信设计准确地表达了模型，而代码准确地表达了设计；第三是实现阶段，安全操作系统的实现目前有两种方法，一种是专门针对安全性而设计的操作系统，另一种是将安全

的特性加到目前的操作系统之中。

11.5.1　安全模型

安全模型用来描述计算机系统和用户的安全特性，是对现实社会中一种系统安全需求的抽象描述。一个计算机系统要达到安全应有 3 个要求：保密性、完整性和可用性，这 3 个基本要求描述了用户对计算机系统的存取方式的要求。这里，将讨论安全操作系统应具有的特性，使用模型描述计算机系统的安全特性。由于存取是计算机系统安全需求的核心，存取控制则是这些模型的基础。假设某些存取控制策略规定了用户是否可以存取特定实体，并假定这些策略是独立于任何模型建立的，也就是说，策略决定了用户是否应存取特定实体，而模型仅是实现策略的机制。

计算机的安全模型设计的目的有两个：一是模型在确定一个保密系统应采取的策略方面要达到保密性和完整性；二是对抽象模型的研究将使人们了解到保护系统的特性。

1. 单层模型

在这种安全模式中，存取策略为每个用户和实体简单地指定权限，即用户将对实体的存取策略简单地设置为“允许”或“禁止”。实现这种模式有两种不同的表达方法。

(1) 监督程序模型

监督程序是用户之间的通道，如图 11.14 所示，用户通过调用监督程序对某一实体进行特定的存取。监督程序响应用户的请求，依据所需的存取控制信息决定是否准许存取。

监督程序的实现虽然很容易，但存在着两个缺陷：

① 由于监督程序进程被频繁地调用，它将成为系统的一个瓶颈。

② 它仅对直接的存取进行控制，无法控制间接存取。例如，下面的程序段，

```
if   profit ≤0
     then delete   file A
     else begin
          write   file A, ″message″;
          close   file A
     end.
```

以上程序借助于文件 A 的存在可以判断 profit 的信息，由此用户可以不存取 profit 只通过文件 A 而达到存取 profit 的目的，可见这种间接存取的方式监控程序是无法控制的。

(2) 信息流模型

为了弥补监督程序的缺点，提出了信息流模型，它的作用类似于一个智能筛选

程序，它控制所允许存取的特定实体的信息发送，其信息流模式的安全性如图 11.15 所示。

下面给出信息流模型的一个例子，其信息流是从 B 到 A 的：

A:=B；

A:=B+C；

A:=A+B；

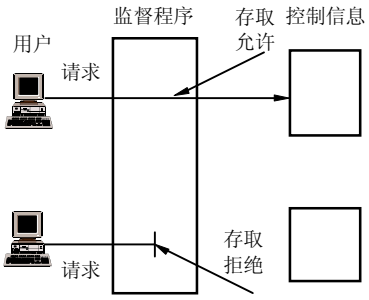


图 11.14　存取的监督程序模型

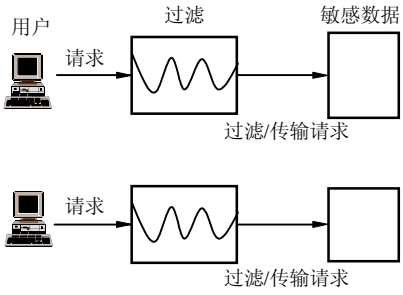


图 11.15　信息流模型

但在下面的例子中，信息流则是难确定的，用户不用直接存取 B 则可知道 B 是否为 0，因为 B 的有关信息可以由 A 推出：

if　B=0　then　A:=0　else　A:=1

若某一模块中的敏感数据被不应存取该敏感数据的用户所调用，则信息流模型可用来描述这种潜在的存取，这个过程可由编译程序实现，并且对程序中每条语句的信息流分析证明，来自一个模块的非敏感数据的输出将不会受该模块对敏感数据存取的任何影响，用户也将不会从调用的模块中得到任何非授权信息。信息流分析能保证操作系统模型在对敏感数据的存取时不会将数据泄露给所调用的模块。

单层模型有一定的局限性，在现代安全操作系统设计中，提出了多级安全模型的概念，信息流模型在其中得到了更深入的应用。如著名的 Bell-LaPadula 模型和 Biba 模型，前者主要用于实现保密，后者主要用于保证信息的完整性。

2. 多层网络模型

在前面的两个模型中，安全性具有二元性：实体有敏感或不敏感，用户有授权存取或未授权存取。然而，通常需要为实体及用户考虑更多的敏感层次。安全网络模型突破了二元敏感性，以适应那些在不同敏感层次上并行处理信息的系统。

安全网络模型是一种广义的操作系统安全模型，其元素形成一种网络数学结构，它基于部队安全需要和保密信息级别的处理。在部队中，信息一般被分为不保密、秘密、机密以及绝密，这些不相交的级别描述了信息的敏感性。绝密信息是最敏感的，而不保密信息是不敏感的。人们使用敏感数据工作时的安全原则是最小权限原

则，主体应存取完成工作所需的最少实体。因此，人们不应对所有绝密信息进行存取，他们仅能存取那些完成其工作所必须的信息。信息存取由“需知(Need-To-Know)”准则限制，仅允许需要此数据来完成其工作的主体存取某一敏感数据。

每块分类信息都和一个或多个隔离组有关，隔离组描述了信息的主体对象。隔离组被用来施行“Need-To-Know”约束，使得人们能进行存取的信息是仅和其工作有关的。

〈级别；隔离组〉组合叫信息的分类。对敏感信息进行存取必须被批准，通常采用存取权限中的许可证机制，许可证是表示某人可以存取特定敏感级别以上的信息并且他必须知道敏感信息所属的特定类。主体许可证和信息分类的形式一样，若引入敏感实体和主体上的关系“≤”，则对于实体 O 和主体 S 有如下的关系：

当且仅当　　级别 O≤级别 S 并且隔离组 O∈隔离组 S

O≤S

这里，关系“≤”用来限制敏感性及主体能存取的信息内容，只有当主体的许可级别至少与该信息的级别一样高，并且主体必须知道信息分类的所有隔离组时才能够存取。例如，分类信息〈机密；USSR〉可被拥有〈绝密；USSR〉或〈机密；USSR，密码学〉的人存取，但不能被拥有〈绝密；密码学〉或是拥有〈秘密；USSR〉的人存取。这种安全模型是一种称为网格的更为一般模式的代表，下面就介绍网格模型。

网格是一种其元素在关系运算符操作下的数学结构。网格中的元素按半定序≤排列，它具有传递性和非对称性，即对任意三个元素 A, B, C 有，

传递性　　若 A≤B 及 B≤C　　则　　A≤C

非对称性　若 A≤B 及 B≤A　　则　　A=B

这种安全模型同时强调了敏感性和需知性需求，前者是层次需求，后者是非层次的约束需求，网格表达了自然的安全级别递增，因此，敏感级别的网格表达是一个可用于多种不同的计算环境的安全模型。

综上所述，研究计算机的安全模型有两个目的：第一，模型在确定一个保密系统应采取的策略方面是重要的，例如，Bell-LaPadula 模型和 Biba 模型能表示为达到保密性或完整性需强加的特定条件；第二，对对象模型的研究将使我们了解到保护系统的特性，这些特性都是保护系统的设计者应知道的。

下面研究安全操作系统的设计，这些设计遵循一系列保护系统模型所建立的策略。

11.5.2　安全操作系统的设计

操作系统的设计是非常复杂的，因为它要处理许多任务，易发生中断及进行低层文本切换操作，与此同时，还必须尽可能地缩小系统开销以提高用户的存取速度，如果再考虑安全因素，则更增加了系统设计的难度。下面将在一个较高的安全层次

上研究操作系统的设计。首先讨论安全操作系统设计的基本原则，然后介绍几种实现操作系统安全的方法。

1. 通用操作系统安全设计的原则

在通用操作系统中，除了前面讲述的内存保护、文件保护、存取控制和用户身份鉴别外，还需考虑共享约束、公平服务保证、进程间的通信与同步，这在有关章节中已细述过。一般来说，安全功能遍及整个操作系统的设计和结构中，它表现在安全操作系统设计中的两个方面：一方面，必须在操作系统设计的每个方面考虑安全性，在设计了一个部分之后，必须检查它所强制或提供的安全性尺度；另一方面，安全性必须成为操作系统初始设计的一部分。这也是人们进行操作系统安全设计时必须遵循的原则。

2. 几种实现操作系统安全的方法

基于前面讲过的操作系统的设计原则，下面将介绍操作系统在安全方面成功实现的方法，即考虑隔离(最少通用机制的逻辑扩展)，内核化的设计(最少权限及机制经济性方面的考虑)，分层结构(开放式设计及完整的策划)等。

(1) 分离/隔离

将一个进程和其他进程分离的方法有 4 个：物理分离、暂时分离、密码分离和逻辑分离。使用物理分离，进程可使用不同的资源；暂时分离用于进程处于不同的时间段；加密被用于密码分离，使得非授权用户不能以可读的形式存取敏感数据；逻辑分离也叫隔离，它是由进程(如监督程序)提供给用户，并将某一用户的实体和其他用户实体分开的一种手段。安全计算系统应使用所有这些形式的分离，常见的方式有两种：虚拟存储空间和虚拟机方式。

虚拟存储空间最初是为提供编址和内存管理的灵活性而设计的，实际上它同时也提供了一种安全的机制。多道程序操作系统必须将用户之间进行隔离，仅允许严格控制下的用户交互作用，大多数操作系统采用系统副本的方式为多个用户提供单个环境，采用虚拟存储空间机制提供逻辑分离。每个用户的逻辑地址空间通过用户存储映像机制与其他用户的逻辑地址分离开，用户程序看似运行在一个单用户的机器上。IBM MVS 操作系统便提供了这种多虚拟存储空间机制。

如果将虚存概念扩充，系统通过给用户提供逻辑设备、逻辑文件等多种逻辑资源，就形成了虚拟机的隔离方式。系统的硬件设备在传统上都是在操作系统的直接控制下，而给每个用户提供虚拟机就使每个用户不但拥有逻辑内存，而且还拥有逻辑 I/O 设备、逻辑文件等其他逻辑资源。虚存只给用户提供了一个与实际内存分离的，且比实际内存要大的内存空间，而虚拟机则给用户提供了一个完整的硬件特性概念，一个本质上完全不同于实际机器的机器概念。虚拟硬件资源也同样在逻辑上从其他用户处分离出来。

可见，两种方式都是将用户和实际的计算机系统分离开来，因而减少了系统的

安全漏洞，改善了用户间及系统硬件间的隔离性能，当然，与此同时也增加了这些层次上的系统开销。

(2) 内核机制

内核是操作系统中完成最低层功能的部分，在标准的操作系统中，内核实现的操作包含了进程调度、同步、通信、信息传递及中断处理。安全内核是负责整个操作系统安全机制的部分，它提供硬件、操作系统及计算系统其他部分间的安全接口，通常，安全内核包含在操作系统内核中，而安全功能在内核中分离。

另外，在内核中加入了用户程序和操作系统资源之间的一个接口层。内核的存在并不能保证它包括所有的安全功能。安全内核的设计和使用在某种程度上依赖于设计的方法，可以将安全内核设计为操作系统的附加部分或是整个操作系统的基本部分，一种方法是在操作系统内核中加入安全功能，另一种方法是先设计安全内核功能。

对于前一种方法，在一个已实现的操作系统中，与安全有关的活动可能会在许多不同场合完成，安全性可能和每次内存存取、I/O 操作、文件或程序存取有关。在模块化的操作系统中，这些独立的活动可能由不同的模块来处理。因此，必须在每个这样的模块中加入和安全有关的功能，这种加入可能会破坏已有系统的模块化特性，而且使加入安全功能后的内核安全验证变得很困难，于是，设计者可能会设法从已实现的操作系统中分离出安全功能，建立单独的安全内核，因此，有了第二种方法。

第二种方法是先设计安全内核，再以它为前提设计操作系统是一种更为合理的方法。目前较多的操作系统使用这种方法进行设计。在这种基于安全的设计中，安全内核为接口层，它位于系统硬件的顶端，管理所有操作系统硬件，存取并完成所有保护功能，它依赖于支持它的硬件，允许操作系统处理大多数和安全无关的功能。这样，安全内核可以很小且很有效，由此在计算系统中至少存在硬件、安全内核、操作系统和用户 4 层运行区域，安全内核必须保证每个区域的保密性和完整性，并管理如下 4 种基本的相互作用：

① 进程激活。在多道程序环境中，进程的激活和挂起经常发生，由一个进程切换到另一个进程要求完全改变寄存器、重定位映像、文件存取表、进程状态信息及其指针等安全敏感信息。

② 运行区域切换。运行在某一区域中的进程为了得到更多的敏感数据或服务，通常调用其他区域的进程。

③ 内存保护。由于每个区域都包括了保存在内存的代码及数据，故安全内核必须管理内存的引用，以确保每个区域的保密性及完整性。

④ I/O 操作。在某些系统中，有一种将每个字符进行转换的 I/O 操作软件，该软件在最外层连接用户程序，在最内层连接 I/O 设备，因此，I/O 操作可以覆盖所有

区域。

(3) 分层结构

前面讲过，基于内核的操作系统由 4 层组成：硬件、内核、操作系统及用户。每层都由子层及其本身组成。安全操作系统的设计思想可以用一系列同心圆予以描述，其中最敏感的操作位于最内层，越接近中心进程的可信度进程则越高。分层结构如图 11.16 所示。

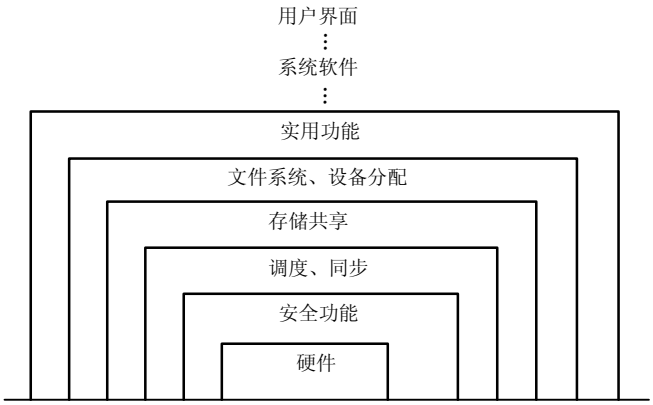


图 11.16　分层操作系统

在这种设计中，有些保护功能是在安全内核之外实现的，如用户身份鉴别等，这些可信模块必须能提供很高的可信度。由于可信度和存取权限是分层的基础，单个安全功能可在不同层的模块中实现，每层上的模块完成具有特定敏感度的操作，如图 11.17 所示。

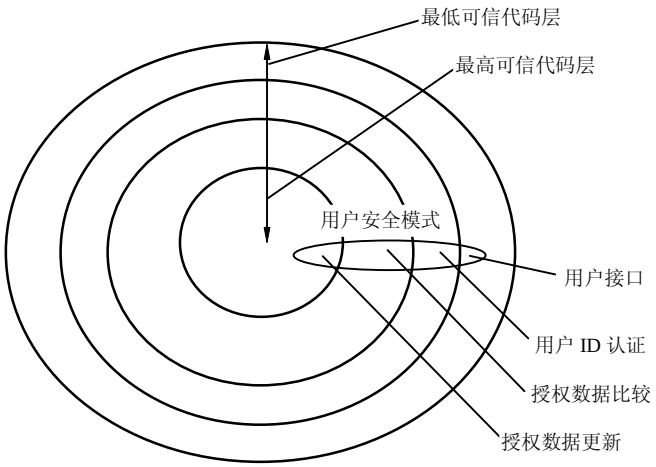


图 11.17　不同层上的模块操作

分层结构设计方法是一种较好的操作系统设计方法，每层使用几个作为服务的中心层，并为其外层提供特定的功能支持，安全操作系统的设计也可采用这种方式，并在各个层次中考虑系统的安全机制。

(4) 环结构

如果将上述的分层描述成围绕系统硬件的环，则可构成一种称为环的运行保护模式。MULTICS 操作系统在每步设计中都采用分层设计，在运行中则是通过环结构来实现的，处于不同环区域的进程拥有不同的权限。

每个运行进程在特定的环区域上运行。可信度高的进程则在编号较小的环区域上操作，环是重叠的，即运行在环 I 上的进程包括了所有环 J 的权限，其中 J>I，环的编号越小，进程能存取的资源则越多，操作所要求的保护也就越少。每个数据区或过程称为段，段由 3 个编号保护，B_1，B_2，B_3，其中 $B_1 \leqslant B_2 \leqslant B_3$。这 3 个编号〈$B_1$，$B_2$，$B_3$〉叫做环域；〈$B_1$，$B_2$〉叫做存取域，〈$B_2$，$B_3$〉叫做调用域或调用点。从 B_1 到 B_2 为进程能自由存取段的环集合，大于 B_2 小于 B_3 的环则为进程仅能在特定的入口点调用段的环。例如，内核段可能具有存取域(0，4)，意思是在 0~4 层的进程可以自由运行，某一用户段可能拥有存取域(4，6)，表示它通常只由用户进程存取。环域表示在某一段中的可信度，可信度高的段具有起始号较小的存取域，可信度低的段则具有起始号较大的存取域，可信度低的段也很少被可信度高的内核进程调用。

以上讨论了用于安全操作系统设计中的 3 个基本原则：隔离、安全内核及分层结构。下面将介绍安全操作系统设计的实例。

11.6　系统举例——Windows 2000 的安全性分析

Windows 2000 的安全模型影响着其整个操作系统，该安全模型如图 11.18 所示。

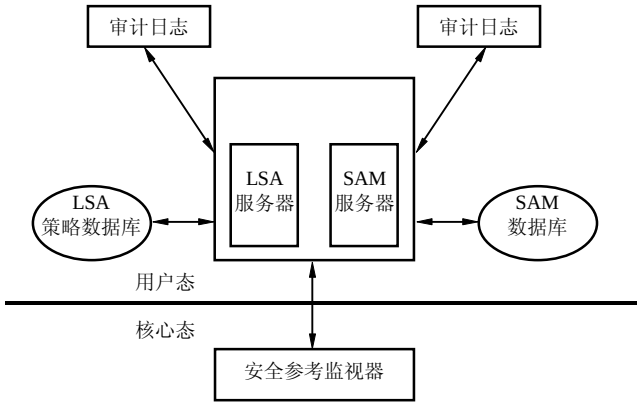


图 11.18　Windows 2000 的安全模型

Windows 2000 是一个安全性很强的操作系统，近来已经通过了桔皮书标准的评估，该系统达到了 C2 级的级别，该级别在安全性方面具有如下的基本特征：

① 安全登录机构。它要求在允许用户访问系统之前，输入惟一的登录标识符和口令来标识自己。

② 安全的访问控制。它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。

③ 安全审核。它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，以便跟踪任何执行非法操作的用户。

④ 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 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数据。

Windows 2000 是通过安全性子系统和相关的组件来实现这些特征的，它的安全机制的核心是其安全子系统，此安全子系统控制了安全的各个方面，是一个集成的子系统，决定了 Windows 2000 操作系统，其组件和数据库由以下几部分组成：

• 本地安全权限(LSA)服务器。它是一个用户态进程，负责本系统安全规则、用户身份验证以及向“事件日志”发送安全性审核消息。

• 安全引用监视器(SRM)。该组件以核心模式运行，负责执行对对象安全访问的检查、处理权限和产生安全审核消息。

• LSA 策略数据库。这是一个包含了系统安全性规则设置的数据库。

• 安全账号管理器服务器。这是一个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的用户名和组。

• SAM 数据库。这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。

• 默认身份认证包。这是一个动态链接库，在进行身份验证的进程的描述表中运行，这个动态链接库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，如匹配则返回该用户的信息。

• 登录进程。这是一个用户态进程，负责搜索用户名和密码，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。这是一个响应网络登录请求的 SERVICES.EXE 进程内部的用户态服务。

保护对象是访问控制和审核的基本要素，在 Windows 2000 操作系统中被保护的对象有：文件、设备、邮件槽、进程、线程、事件、访问令牌、Windows 桌面、窗口站、服务、互斥体、信号量、注册表键、管道和打印机等。Windows 2000 以两种方式的访问控制作用于对象：一种是自主访问控制，这是大多数人在该操作系统使用保护时所采用的保护机制，通过这种方式对象的所有者决定其他任何人对对象的访问；另一种方式是特权访问控制，当对象的所有者不在时，它可以确保他人能够访问受保护的对象。

Windows 2000 的安全审核记录流程如图 11.19 所示，SAM 经连接到 LSA 的 LPC 发送这些审核事件，“事件记录器”将审核事件写入“安全日志”中，除了由 SRM

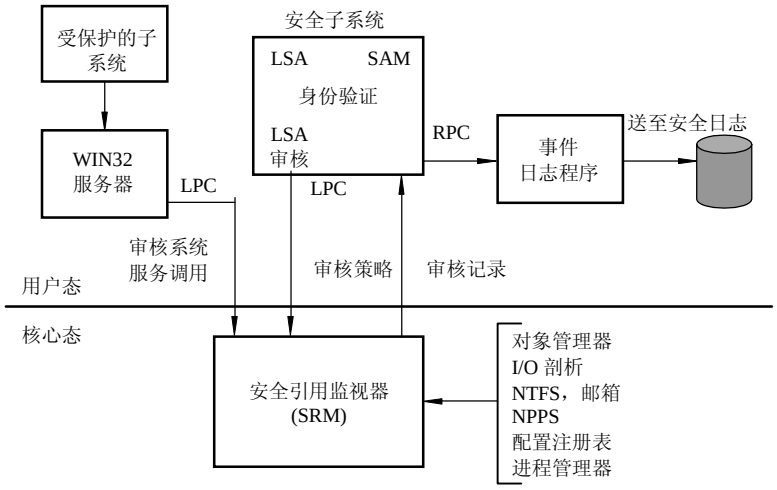


图 11.19　安全审核记录流程图

传递的审核事件之外，LSA 和 SAM 二者都产生 LSA 直接发送到“事件记录器”的审核记录，当接收到审核记录之后，它们被放到队列中以便发送到 LSA，而不会被成批地提交，可以使用两种方式中的一种从 SRM 中把审核记录移到安全子系统。但是，如果审核记录比较小，它就被作为一条 LPC 消息来发送，审计记录从 SRM 的地址空间复制到 LSASS 进程的地址空间；如果审核记录较大，则 SRM 使用共享内存让 LSASS 可以使用该消息，并在 LPC 消息中简单地传送一个指针。

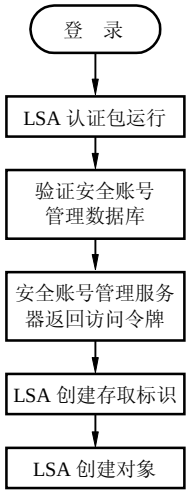


图 11.20　登录过程

登录是为确认用户身份而设计的，它是抵御非法存取的第一道防线。Windows 2000 的登录是由登录进程、LSA、一个或多个身份验证包和 SAM 的相互作用来组成的。身份验证包是执行身份验证检查的动态链接库。用户登录过程如图 11.20 所示。

总之，Windows 2000 具有很高的安全性能，它具有一组全面的、可配置的安全性的服务，该服务已经达到美国 C2 级的安全标准，并且它还提供了安全函数的一个扩展数组以适应政府机关和商业安装的关键需求。

11.7　小　　结

目前人们对计算机操作系统的安全性越来越重视，操作系统安全的本质就是为数据处理系统从技术和管理上采取安全保护措施，以保护计算机硬件、软件和数据不因偶然的因素或恶意的攻击而遭到破坏，使整个系统能够正常可靠地运行。

操作系统的安全保护有许多方式：进程支持、内存及地址保护、存取控制、文件保护、用户身份鉴别、界址、基址/界限寄存器等，而特征位结构，分页和分段以及段页式也是用于编址及保护设计的机制。通用操作系统上的文件保护模式一般基于类似于用户-组-所有者这样的 3 或 4 层格式，这种格式便于直接实现，但它把存取控制的粒度限制在少数几个层次上。存取控制通常通过以单实体或单用户为基础的存取控制矩阵来实现，尽管这种机制很灵活，但是在实现上难以有效的进行。用户身份鉴别则是在非法用户通过计算机网寻求共享服务时变得更为严重的一个安全问题。传统的身份鉴别设施则是口令。明文口令文件使得计算机系统存在严重的脆弱性。通常，这些文件或是被重点保护或是被加密。但是，一个更为严重的问题则是建立一个使用户口令足够保密的管理过程。此外，在不信任的环境下还必须使用协议来进行相互的身份鉴别。

有一种令人担忧的威胁是来自病毒及类似的软件机制，它们寻找系统软件的弱点，从而获取非法访问权，或者降低系统服务效率。病毒是一段计算机程序，它的运行离不开操作系统的支持，病毒对操作系统的安全性造成了极大的威胁，特洛伊木马程序就是一个典型的入侵程序。虽然可以采用过程控制、系统控制、强制控制和检查软件等措施降低特洛伊木马的攻击，但也有它的局限性。

加密是计算机安全中的一个重要的技术，加密就是把可理解的消息转化成不可理解的消息。大多数加密采用传统方法，即两个实体共享一个加密/解密密钥。传统加密的主要问题在于密钥的分配和保护；另外还有公开密钥加密方法，其中每个进程有一对密钥，一个用来加密，一个用来解密，加密密钥公开，解密密钥私有。

操作系统是计算机系统中安全性的基石，所以，设计操作系统时安全性是应考虑的一个重点。

习　题　十一

11.1　一个计算机系统的基本特性是什么？

11.2　安全操作系统应该具有哪些功能？

11.3　假设从26个字母表中选4个字母形成一个口令，一个攻击者以每秒钟一个字母的速度试探每个口令。试回答：

(1) 直到试探结束才反馈给攻击者，那么找到正确口令的时间是多少?

(2) 只要输入了不正确字母就反馈给攻击者，找到正确口令的时间又是多少?

11.4　若两个用户共同存取某一段，他们必须使用相同的名字才能存取。他们的保护权限也必须一样吗，为什么？

11.5　使用分段及分页地址转换的一个问题是要使用I/O。假定用户希望将某些数据由输入设备读入内存，为了保证数据传输过程中的有效性，通常将要放入数据处的实际内存地址提供给I/O设备，由于将实际地址传送给I/O，因此，在非常快速的数据传输过程中不再需要进行费时的地址转换。这一方法所带来的安全问题是什么？

11.6　目录是应控制存取的一种实体，为什么不允许用户直接修改他人的目录？

11.7　描述下列4种存取控制机制：①在运行过程中易于确定授权存取权限；②易于为新实体增加存取权限；③易于通过主体删除存取权限；④易于建立一个新实体，使得所有主体都具有缺省存取权限。

怎样适合下列情况：

(1) 主体存取控制表(即，每个主体的一张表表示该主体能存取的所有实体)；

(2) 实体存取控制表(即，每个实体的一张表表示所有能存取该实体的主体)；

(3) 存取控制矩阵；

(4) 权能。

11.8　试述计算机病毒运行机理？

11.9　阐述计算机病毒与操作系统的关系？

11.10　给出一些流量分析可以破坏安全性的例子，描述端到端加密与链路加密结合时仍有可能进行流量分析的情形。

11.11　试述计算机操作系统的安全模型？

11.12　Windows 2000 操作系统的安全性是怎样实现的？

排队分析理论

本书参考了排队分析理论并得出了基于排队分析的结果。确实，排队分析理论是计算机和网络人员的重要工具之一，它可对许多问题作出回答，如：

① 磁盘 I/O 使用增加时，文件查寻时间如何变化?

② 处理机速度和系统中用户数量都翻倍时，响应时间变化吗?

③ 时间共享系统在拨号装置上应有多少条线?

④ 在线咨询中心需要多少台终端，操作员有多少空闲时间?

虽然排队论的数学基础很复杂，但应用排队论进行效率分析一般都相当简单，只需要基本的统计概念(平均值和标准方差)和排队论概念。

本章只是简单介绍排队分析理论。

12.1 为什么进行排队分析

对改变设计后所产生的影响，尤其是对系统效率的影响，进行分析是很重要的。在交互式或实时应用中，系统效率主要由响应时间反映。在其他情况中，主要考虑产出率。但都要根据现有负载信息，或者对负载进行估计来制订方案，具体方法是：

① 事后进行分析；

② 根据现有经验对要出现的环境制订一个简单方案；

③ 根据排队论构造一个分析模型；

④ 设计并运行一个模拟模型。

一般不用第①种方法，第②种方法虽然看上去较合理，但无法确定将来需要什么，因此，无法提出精确的模型。这种方法的问题在于大多数负载变化的系统行为与我们的主观想象不一样。如果一个系统中有共享设备，那么该系统的效率呈指数

级增加，如图 12.1 所示。

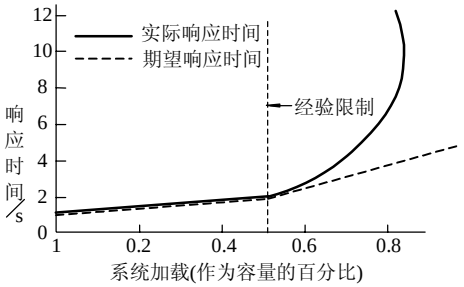


图 12.1　期望响应时间和实际响应时间比较

于是，就需要更精确的估计工具。第③种方法是利用分析模型，该分析模型表示成一组方程，解此方程组就可得到想知道的参数值(响应时间、产出率等)。对于计算机、操作系统、网络问题，以及实际问题，基于排队论的分析模型比较符合实际情况。排队论的不足在于要进行大量的简化假设，以便列出参数方程。

第④种方法是利用一个模拟模型。有了模拟程序设计语言，分析者就可以详细地模拟现实。但大多数情况不需要模拟模型，而且现有的方法也可能有错。因此，不管模拟模型设计得如何好，其结果总要受输入的限制。另外，尽管排队论有许多假设，但其结果与设计得较好的模拟模型结果仍很接近，而且利用排队分析对良定义问题只需几分钟就可完成，而模拟需要几天、几星期甚至更久。

12.2　排队模型

12.2.1　单服务器模型

图 12.2 给出了一个最简单的排队系统。系统核心是一个服务器，它为到达系统的项目服务。如果服务器空闲，就可立即为一个项目服务；否则，就将该项目加入等待队列中。服务完成后，项目离开系统。如果系统中有等待项目，就立即从中调一个给服务器。

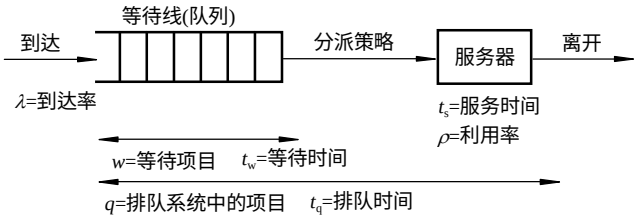


图 12.2　单服务器队列的排队系统结构和参数

图 12.2 同时给出了排队模型中的参数。项目以平均到达率λ进入系统。在任意

给定时间，队列中有一定数量的等待项目(0 个或更多)，平均等待的项目数是 w。项目平均等待时间是 t_w，t_w 是所有输入项目包括不等待项目的平均等待时间。服务器对项目的平均服务时间是 t_s，它表示将一个项目送到服务器到该项目离开服务器的平均时间。利用率是一段时间内，服务器处于忙状态的时间百分比。还有两个衡量整个系统的参数：系统中平均项目数 q，包括正在服务的项目和等待项目；项目在系统中的平均时间 t_q，包括等待时间和服务时间。

假设等待队列无限长，那么系统不会丢失任何项目，项目被延迟直到被服务。在这种情况下，离开率等于到达率，也就是系统中的流量。随着到达率的增加，利用率也在提高，相应拥塞也增加，等待队列变长，等待时间也变长。ρ=1 时，服务器处于饱和状态，其所有时间都在工作。

理论上，系统能处理的最大到达率为$\lambda_{max}=1/t_s$。

如果 ρ=1 时等待队列无限增长，就会接近系统饱和。实际上由于响应时间及缓冲大小的要求，到达率的最大值一般为理论值的 70%～90%。

下面，对模型作些假设：

① 项目数。假设有无穷个项目，这就意味着系统不会因为项目数而改变到达率。如果项目数有限，由于当前系统中有项目，从而使到达的项目减少，就进一步降低了到达率。

② 队列大小。假设队列无穷大，于是等待项目数可无限增加。在有限队列中，系统可能会丢失某些项目。实际上，任何队列都是有限的，在许多情况中，这对分析并无大碍。

③ 调度规则。如果服务器空闲，并且有多个项目等待，那么就要决定为哪个项目服务。最简单的规则是先进先出，另外还有后进先出或基于服务时间的规则。基于服务时间的规则很难进行模型分析。

表 12.1 归纳了图 12.2 中所用的概念及其他有用参数，为了表示参数的变动情况，引入了标准方差。

表 12.1　本章用到的参数

参数	含　义	参数	含　义
λ	每秒钟到达项目数的平均值	σ_{tq}	t_q 的标准方差
t_s	每个到达项目的平均服务时间	w	等待服务的平均项目数
σ_s	服务时间的标准方差	t_w	一个项目等待服务的平均时间
ρ	利用率，设备忙所占的时间百分比	t_d	等待项目的平均等待时间(不包括等待时间为 0 的项目)
q	系统中的(等待的和正在被服务的)平均项目数	σ_w	w 的标准方差
t_q	一个项目在系统中的平均时间	m	服务器的个数
σ_q	q 的标准方差	$m_x(r)$	第 r 百分布数；使 x 在不超过 r% 的时间来出现的 r 值

12.2.2　多服务器模型

图 12.3 给出了一个简单模型，其中有多个服务器共享一个等待队列。如果有一个项目到达系统并且有多于一个的服务器空闲，那么该项目可立即得到服务。假设所有服务器都是一样的，有多个服务器空闲，则由哪一个服务并无很大区别。如果所有服务器都忙，就形成一个等待队列。一旦有服务器空闲，就按调度规则调度等待队列中的一个项目。

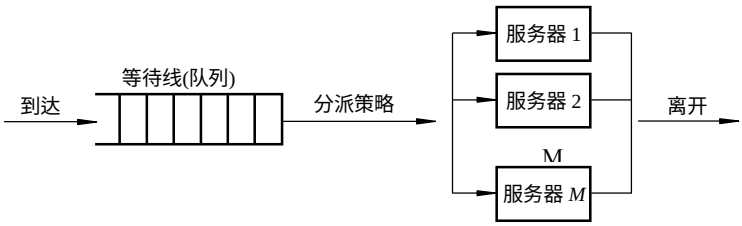


图 12.3　多服务器队列的排队系统结构

除了利用率外，图 12.2 中所有参数在多服务器系统中含义相同。如果有 M 台相同的服务器，每台服务器的利用率是ρ，那么整个系统的利用率为 $M\rho$。理论上，最大利用率为 $M\times100\%$，最大到达率为 $\lambda_{\max}=M/t_s$。

12.2.3　基本排队关系

首先作一些简化假设，这可能会使模型有效性降低，但大多数情况中，所得结果还算准确。有一些排队关系如表 12.2 所示。

表 12.2　一些基本的排队关系

基本排队关系	系　　统	基本排队关系	系　　统
$\rho=\lambda t_s$	单服务器系统	$t_q=t_w+t_s$	
$\rho=\dfrac{\lambda t_s}{M}$	多服务器系统	$q=w+\rho$	单服务器系统
$q=\lambda t_q$		$q=w+M\rho$	多服务器系统
$w=\lambda t_w$			

12.2.4　假设

排队分析的基本任务是：将到达率和服务时间作为输入，得出等待项目、等待时间、排队的项目、排队时间等信息。

我们想知道这些输出的平均值(w, t_w, q, t_q)，另外还要了解它们的变动性(σ_q，σ_{t_q}，σ_w，σ_{t_w})，以及其他方法。

回答这些问题需要了解到达率和服务时间的分布。但是，就算知道了服务时间的分布，但由于公式太复杂了，也难以解决问题。为使问题易处理，作些简化假设。

最重要的是假设到达率服从泊松分布，另外，假设服务时间呈指数分布或者服务器为不间断服务。有了这些假设，如果知道到达率和服务时间的平均值和标准方差，就可得到许多有用结果。

用一个简单概念来概括排队模型中所作的主要假设，就是 *X*/*Y*/*N*，其中 *X* 指相继到达间隔时间的分布，*Y* 指服务时间的分布，*N* 指服务器的数目，最普通的分布表示如下：

G=一般独立的到达或服务时间

M=负指数分布

D=确定型到达或固定的服务时间

于是，*M*/*M*/1 表示到达率为泊松分布，服务时间为负指数分布，单服务器的模型。

12.3　单服务器队列

表 12.3(a)给出了 *M*/*G*/1 模型的一些公式，其中到达率呈泊松分布。利用因子 *A* 可简化公式。*A* 中最重要的部分是服务时间的标准方差与平均值的比，*A* 中不包含其他有关服务时间的信息。有两种特殊情况：标准方差等于平均值时，服务时间为指数分布，这是最简单的情况。表 12.3(b)给出了σ_{q}、$\sigma_{t_{\mathrm{q}}}$、σ_{w}和$\sigma_{t_{\mathrm{w}}}$简化后的形式，以及其他一些相关参数。另一种特殊情况是标准方差为 0，也就是不间断服务，相应公式如表 12.3(c)所示。

那么，最可能的$\sigma_{t_{\mathrm{s}}}/t_{\mathrm{s}}$值是多少？考虑 4 种情况：

① $\sigma_{t_{\mathrm{s}}}/t_{\mathrm{s}}$值为 0。这是不间断服务，这种情况很少。如果所有消息等长就适合这种情况。

② $\sigma_{t_{\mathrm{s}}}/t_{\mathrm{s}}$值小于 1。这比指数分布的服务要好，用 *M*/*M*/1 表得出的队列大小和排队时间比实际值要长。

③ $\sigma_{t_{\mathrm{s}}}/t_{\mathrm{s}}$值接近 1。这是最一般的情况，也就是服务时间是随机分布的。例如，订飞机票、文件查询、共享局域网及包交换网。

④ $\sigma_{t_{\mathrm{s}}}/t_{\mathrm{s}}$值大于 1。用 *M*/*G*/1 模型，而不是 *M*/*M*/1。这种模型的一般情况有两个峰值，在两个峰值之间有较宽扩展。例如，系统中有许多短进程、许多长进程，中间进程很少的情况。

对到达率作相同分析。对于呈泊松分布的到达率，相继到达间隔时间为指数分布，标准方差与平均值的比值为 1。如果观察到的比值比 1 小得多，那么，到达率

可能是平均分布(变化不大)，泊松分布数假设将过高估计队列大小和延迟。另一方面，如果比值比 1 大，那么，到达是成群的，并且更容易引起阻塞。

表 12.3　单服务器队列公式

<table>
<tr><td colspan="2">假定：1.泊松到达率　　2.分派策略对基于服务时间的项目无影响
3.标准方差公式假设 FIFO 分派策略　　4.无项目丢失</td></tr>
<tr><td>(a) 一般服务时间(M/G/1)
$A=\frac{1}{2}[1+(\frac{\sigma_{t_s}}{t_s})^2]$　有用参数
$q=\rho+\frac{\rho^2 A}{1-\rho}$
$w=\frac{\rho^2 A}{1-\rho}$
$t_q=t_s+\frac{\rho t_s A}{1-\rho}$
$t_w=\frac{\rho t_s A}{1-\rho}$</td><td rowspan="2">(b)指数服务时间(M/M/1)
$q=\frac{\rho}{1-\rho}$
$w=\frac{\rho^2}{1-\rho}$
$t_q=\frac{t_s}{1-\rho}$
$t_w=\frac{\rho t_s}{1-\rho}$
$\sigma_q=\frac{\sqrt{\rho}}{1-\rho}$
$\sigma_{t_q}=\frac{t_s}{1-\rho}$
$\sigma_{t_q}=\frac{t_s}{1-\rho}\sqrt{\frac{\rho}{3}-\frac{\rho^4}{12}}$
$\Pr[q=N]=(1-\rho)\rho^N$
$\Pr[q\leqslant N]=\sum_{i=0}^{N}(1-\rho)\rho^j$
$\Pr[t_q\leqslant t]=1-e^{-(1-\rho)t/t_s}$
$m_{t_q}(r)=t_q\times\log_e(\frac{100}{100-r})$
$m_{t_w}(r)=t_w\times\log_e(\frac{100\rho}{100-r})$</td></tr>
<tr><td>(c) 不间断服务时间(M/D/1)
$q=\frac{\rho^2}{2(1-\rho)}+\rho$
$w=\frac{\rho^2}{2(1-\rho)}$
$t_q=\frac{t_s(2-\rho^2)}{2(1-\rho)}$
$t_w=\frac{\rho t_s}{2(1-\rho)}$
$\sigma_q=\frac{1}{(1-\rho)}R$
$R=\sqrt{\rho-\frac{3\rho^2}{2}-\frac{5\rho^3}{6}-\frac{\rho^4}{12}}$</td></tr>
</table>

12.4　多服务器队列

表 12.4 列出了多服务器队列中的一些主要公式，只有在 *M*/*M*/*N* 模型中才可能得到该模型的拥塞统计，其中，*N* 个服务器的服务时间都是呈指数分布。

表 12.4　多服务器队列的公式(M/M/N)

假　定	公　式
1. 泊松到达率 2. 指数服务时间 3. 所有服务器负载均衡 4. 所有服务器有相同的平均服务时间 5. FIFO 分派策略 6. 无项目丢失	$K=\dfrac{\sum_{N=0}^{M-1}\frac{(M\rho)^N}{N!}}{\sum_{N=0}^{M}\frac{(M\rho)^N}{N!}}$ 有用参数 所有服务器都忙的概率 $B=\dfrac{1-K}{1-\rho K}$ $q=B\dfrac{\rho}{1-\rho}+M\rho$ $w=B\dfrac{\rho}{1-\rho}$ $t_{\mathrm{q}}=\dfrac{B}{M}\dfrac{t_{\mathrm{s}}}{1-\rho}+t_{\mathrm{s}}$ $t_{\mathrm{w}}=\dfrac{B}{M}\dfrac{t_s}{1-\rho}$ $\sigma_{t_{\mathrm{q}}}=\dfrac{t_{\mathrm{s}}}{M(1-\rho)}\sqrt{B(2-B)+M^2(1-\rho)^2}$ $\sigma_{\mathrm{w}}=\dfrac{1}{1-\rho}\sqrt{B\rho(1+\rho-B_\rho)}$ $\mathrm{PR}[t_{\mathrm{w}}>t]=\mathrm{B}\mathrm{e}^{-M(1-\rho)t/t_{\mathrm{s}}}$ $t_{\mathrm{d}}=\dfrac{t_{\mathrm{s}}}{M(1-\rho)}$

12.5　队　列　网

在分布式环境中，分析者面临的不是孤立的队列。通常，待分析的问题包含几个相互连接的队列，如图 12.4 所示。图中，节点代表队列，连线代表流量。

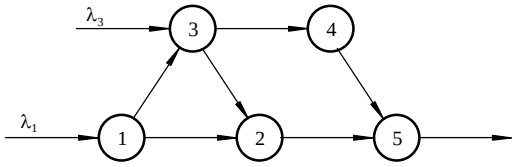


图 12.4　队列网举例

这种队列网中，有两种情况可使问题复杂化：

① 流量的分割和汇聚，如图中节点 1 和节点 5 所示。

② 一前一后或按序排列的队列，如节点 3 和节点 4 所示。

超出这两种情况的队列问题则无法分析。但如果流量呈泊松分布，服务时间呈指数分布，那么就有简单的解决方法了。本节先讨论所列出的这两种情况，然后给

出队列分析方法。

12.5.1　信息流的分割和汇聚

假设到达一个队列的信息平均到达率为λ，并且有两条通道 A 和 B，一个项目被服务完要离开队列时，则以概率 P 使用通道 A，以概率$(1–P)$使用通道 B。一般输出流 A 和 B 的分布与输入分布不相同。但如果输入为泊松分布，那么两个输出流也为泊松分布，平均离开率分别为 $P\lambda$ 和$(1–P)\lambda$，如图 12.5(a)所示。

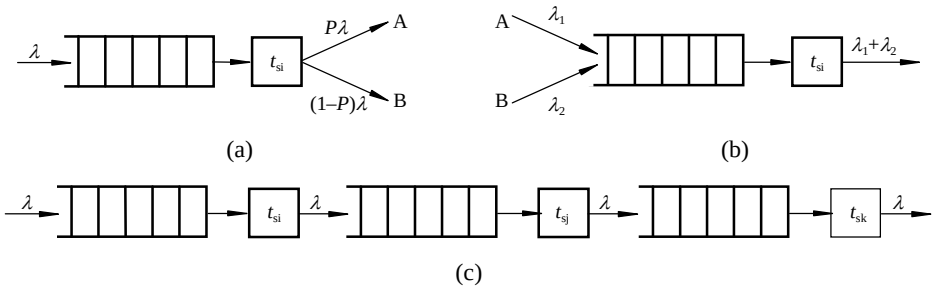


图 12.5　排队网元素

(a) 流量分割　(b) 流量汇集　(c) 简单的一前一后的队列

信息的汇聚与此相似，如果平均到达率分别为 λ_1 和 λ_2 的两个泊松分布流汇聚就得到平均到达率为 $\lambda_1+\lambda_2$ 的泊松分布流，如图 12.5(b)所示。

这些结果可扩充到分割或两个以上的分支及两个以上达到流汇聚的情况。

12.5.2　一前一后的队列

图 12.5(c)给出了一个线性单服务器队列集。除第一个队列外，每个队列的输入都是上一个队列的输出。假设第一个队列的输入为泊松分布，每个队列的服务时间为指数分布并且等待队列为无限，那么，每个队列的输出是一个与输入相同的泊松分布流。当这个泊松分布流进入下一个队列中时，在第二个队列中延迟相同的时间，就好像初始流没经过第一个队列直接进入第二个队列中一样。因此，队列间是相互独立的，并且可以一次只分析一个。线性系统的平均总延迟等于每个阶段的平均延迟之和。

这个结果可扩展到线性系统中每个节点都是多服务器队列的情况。

12.5.3　Jackson 定理

Jackson 定理可用来分析队列网，该定理基于 3 个假设：

① 队列网由 m 个节点组成，每个节点都能提供相互独立的指数分布服务。

② 系统外项目都以泊松分布的形式到达。

③ 一个项目在一个节点服务后，就以固定的概率进入另一个节点或退出系统。

Jackson 定理指出：在这样的队列网中，每个节点都是独立的排队系统，其输入

是由分割、汇聚和顺序排队决定的泊松分布流，因此，每个节点都可用 *M*/*M*/1 或 *M*/*M*/*N* 模型分析，将每个节点的延迟加起来就得到系统延迟，但无法得出系统延迟的标准方差。

Jackson 适于包交换网，可以将包交换网模型化成一个队列网。每个包代表一个独立的项目。假设每个包都是独立传送，并且在从源端到目的端的路径上的每一个包交换节点，包按排队传输。一个队列的服务时间就是包的传输时间，与包的长度成正比。

这种方法的缺点是，不满足定理的一个条件，也就是服务分布不独立。由于每条传输链上的包长度相等，每个队列的到达进程与服务进程有关。但已证明，由于分割和汇聚的均衡效果，独立服务时间的假设也有效。

12.5.4　包交换网中的应用

在一个包交换网中，通过传输链将节点连接起来，其中，每个节点都是零个或多个系统的交界，其功能是作为信息的源和目的。提供给网络的外部工作负载为

$$\gamma = \sum_{j=1}^{N}\sum_{k=1}^{n}\gamma_{jk}$$

其中，γ=每秒钟的总负载(按包计)；γ_{jk}=源 j 与目的 k 间的工作负载。

由于一个包可流过源和目的之间的多条链，故内部总负载比所提供的负载要高，即

$$\lambda = \sum_{i=1}^{L}\lambda_i$$

其中，λ=网中所有链路的总负载；λ_i=链 i 的负载。

内部负载由包通过网络的实际路径决定。假设有这样一个路由算法，独立链上的负载λ_i 由提供的负载γ_{jk} 决定。对任意路径，都可用这些参数来确定包通过的平均链数。所有路径的平均长度为

$$q_i=\lambda_i t_i$$

其中，t_i 由每个队列的延迟决定。如果将所有链上的平均项目数加起来，就可以得到网中所有队列的等待包总数的平均值，已证明公式所得的结果比实际值要大。因此，网中等待包的个数为γT。组合这两个结果，得

$$T = \frac{1}{\gamma}\sum_{i=1}^{L}\lambda_i t_i$$

为确定 T 的值，就要确定每个延迟 t_i。由于假设每个队列都可看作独立的 M/M/1 模型，于是很容易得

$$t_i = \frac{t_{s_i}}{1-\rho_i} = \frac{t_{s_i}}{1-\lambda_i t_{s_i}}$$

链 i 的服务时间 t_{si} 就是链上的数据输出率，它等于每秒服务的位数乘以平均包长度，分别用 C_i 和 $1/\mu$ 表示，则有

$$t_i = \frac{\frac{1}{\mu C_i}}{1-\frac{\lambda_i}{\mu C_i}} = \frac{1}{\mu C_i - \lambda_i}$$

从而可得到包通过网络的延迟时间

$$t = \frac{1}{\gamma}\sum_{i=1}^{L}\frac{\lambda_i}{\mu C_i - \lambda_i}$$

12.6　其他排队模型

本章重点讨论了一种排队模型，但实际上还有许多排队模型，它们的基础依据是：

① 处理阻塞项目的方法；

② 信息源的数目。

当一个项目到达服务器，发现服务器忙，或者进入一个多服务器系统发现所有服务器都忙时，就被阻塞。对阻塞项目的处理有多种方法：可以将项目加到队列中，等待服务器空闲，这种方法就是电话模型中的“丢失呼叫延迟”，实际上呼叫并未丢失；另外一种方法是不提供等待线，这就有两种情况：一是项目可以等待任意长一段时间后再试，这就是“丢失呼叫清除”；二是项目不停地要求服务，这就是“丢失呼叫保持”。丢失呼叫延迟模型最适合大多数计算机和数据通信问题；丢失呼叫清除最适合电话交换系统。

信息模型的第二个关键问题是信息源是有限还是无限。对于无限源模型，达到率是固定的。对有限源模型，达到率依赖于已有的源数。如果 L 个源的达到率都是 λ/L，那么所有排队装置空闲时，达到率为 λ；如果占据了 K 个，那么达到率为 $\lambda(L–K)/L$，无限源模型容易处理。源数为系统容量的 5~10 倍时，可设为无限源。

12.7　小　　结

排队分析属随机服务系统理论。主要是研究各种排队系统的概率性，寻求最优的适合要求的服务方案。操作系统中的资源分配、作业调度等等都用到排队分析方法。本章简单介绍了排队分析的基本概念，常用的排队模型，以及排队分析在计算机系统、网络系统中的应用。

参 考 文 献

1 Agarwal A, Horowitz M, Hennessy J. An Analytical Cache Model. ACM Transactions on Computer Systems, May 1988.

2 Almasi G, Gottlieb A. Highly Parallel Computing. Redwood City, CA: Benjamin/Cummings, 1989.

3 Ananda A, Tay B, and Koh E. A Survey of Asynchronous Remote Procedure Calls. Operating Systems Review, April 1992.

4 Andleigh UNIX System Architecture. Englewood Cliffs, NJ: Prentice Hall, 1990.

5 Andrianoff S. A Module on Distributed Systems for the Operating System Course.Proceedings, Twenty-First SIGCSE Technical Symposium on Computer Science Education, SIGSCE Bulletin, February 1990.

6 American National Standards Institute. American National Standard Dictionary for Information Systems. X3-N1-990.

7 Artsy Y. Designing a Process Migration Facility: The Charlotte Experience. Computer, September 1989.

8 Axford T. Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design. New York: Wiley, 1988.

9 Bach M. The Design of the UNIX Operating System. Englewood Cliffs, NJ: Prentice Hall, 1986.

10 Bacon J. Concurrent Systems. Reading, MA: Addison Wesley, 1994.

11 Barbosa V. Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs. IEEE Transactions on Software Engineering, November 1990.

12 Back L. System Software. Reading, MA: Addison Wesley, 1990.

13 Bellovin S. Packets Found on an Internet. Computer Communications Review, July 1993.

14 Ben-Ari M. Principles of Concurrent Programming.Englewood Cliffs, NJ: Prentice Hall, 1982.

15 Ben-Ari M. Principles of Concurrent and Distributed Programming. Englewood Cliffs, NJ: Prentice Hall, 1990.

16 Bernstein P. Transaction Processing Monitors. Communications of the ACM, November 1990.

17 Bernstein P. Middleware: An Architecture for Distributed System Services. Report CRL 93/6. Digital Equipment Corporation Cambridge Research Lab, 2 March 1993.

18 Berson A. Client/Server Architecture. New York: McGraw-Hill, 1992.

19 Bic L, Shaw A. The Logical Design of Operating Systems, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

20 Black D. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. Computer, May 1990.

21 Brazier F, Johansen D. Distributed Open Systems. Los Alamitos, CA: IEEE Computer Society Press, 1994.

22 Brent R. Efficient Implementation of the First Fit Strategy for Dynamic Storage Allocation. ACM Transactions on Programming Languages and Systems, July 1989.

23 Brown R, Denning P, Tichy W. Advanced Operating Systems. Computer, October 1984.

24 Cabrear L. The Influence of Workload on Load Balancing Strategies.USENIX conference Proceedings,Summer 1986.

25 Carr R. Virtual Memory Management.Ann Arbor, MI: UMI Research Press, 1984.

26 Carriero N, Gelernter D. How to Write Parallel Programs: A Guide for the Perplexed. ACM Computing Surveys, September, 1989.

27 Casavant T, Singhal M. Distributed Computing Systems. Los Alamitos, CA: IEEE Computer Society press, 1994.

28 Castillo C, Flanagan E, Wilkinson N. Object Oriented Design and Programming. AT&T Technical Journal, November/December 1992.

29 Chandy K, Lamport L. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, February 1985.

30 Chandras R. Distributed Message Passing Operating Systems. Operating Systems Review, January 1990.

31 Cooper J. Computer and Communications Security: Strategies for the 1990s. New York: McGraw-Hill, 1990.

32 Datta A, Ghosh S. Deadlock Detection in Distributed Systems. Proceedings, Phoenix Conference on Computers and Communications, March 1990.

33 Datta A, Javagal R, Ghosh S. An Algorithm for Resource Deadlock Detection in Distributed Systems. Computer Systems Science and Engineering, October 1992.

34 Davies D, Price W. Security for Computer Networks. New York: Wiley, 1989.

35 Davis W. Operating Systems: A Systematic View, Third Edition,Reading, MA: Addison-Wesley, 1987.

36 Deitel H. An Introduction to Operating Systems, Second Edition.Reading, MA: Addison-Wesley, 1996.

37 Denning P, Brown R. Operating Systems. Scientific American, September 1988.

38 Dewire D. Client/Server Computing. New York: McGraw-Hill, 1993.

39 Douglas F, Ousterhout J. Progress Migration in Sprite: A Status Report. Newsletter of the IEEE Computer Society Technical Committee on Operating Systems, Winter 1989.

40 Eager D, Lazowska E, Zahnorjan J. Adaptive Load Sharing in Homogeneous Distributed Systems. IEEE Transactions on Software Engineering,May 1986.

41 Feitelson D, Rudolph L. Distributed Hierarchical Control for Parallel Processing. Computer, May 1990.

42 Feitelson D, Rudolph L. Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control. Proceedings,1990 International Conference on Parallel Processing,August 1990.

43 Feldman L. Windows NT: The Next Generation. Carmel, IN: Sams, 1993.

44 Finkel R. The Process Migration Mechanism of Charlotte. Newsletter of the IEEE Computer Society Technical Committee on Operating Systems, Winter 1989.

45 Foster I. Automatic Generations of Self Scheduling Programs. IEEE Transactions on Parallel and Distributed Systems, January 1991.

46 Gibbons P. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. IEEE Transactions on Software Engineering, January 1987.

47 Gingras A. Dining Philosophers Revisited. ACM SIGCSE Bulletin, September 1990.

48 Gopal I. Prevention of Store-and-Forward Deadlock in Computer Networks, IEEE Transactions on Communications, December 1985.

49 Grosshans D. File Systems:Design and Implementation. Englewood Cliffs, NJ: Prentice Hall, 1986.

50 Haldar S, Subramanian D. Fairness in Processor Scheduling in Time Sharing Systems. Operating Systems Review, January 1991.

51 Hoare C. Communicating Sequential Processes.Englewood Cliffs, NJ: Prentice Hall, 1985.

52 Hofri M. Proof of a Mutual Exclusion Algorithm. Operating Systems Review, January 1990.

53 Horner D. Operating Systems: Concepts and Applications. Glenview, IL: Scott, Foresman, 1989.

54 Inmon W. Developing Client/Server Applications in an Architect Environment. Boston, MA: QED, 1991.

55 Johnson R. MVS Concepts and Facilities. New York: McGraw-Hill, 1989.

56 Johnson B, Javagal R, Datta A, et al. A Distributed Algorithm for Resource Deadlock Detection. Proceedings, Tenth Annual Phoenix Conference on Computers and Computing, March 1991.

57 Jones S, Schwarz P. Experience Using Multiprocessor Systems—A Status Report. Computing Surveys, June 1980.

58 Kay J, Lauder P. A Fair Share Scheduler. Communications of the ACM, January 1988.

59 Kessler R, Hill M. Page Placement Algorithms for Large Real Indexed Caches. ACM Transactions on Computer Systems, November 1992.

60 Khalidi Y, Talluri M, Williams D, et al. Virtual Memory Support for Multiple Page Sizes. Proceedings, Fourth Workshop on Workstation Operating Systems, October 1993.

61 Klein D. Foiling the Cracker: A Survey of, and Improvements to, Password Security. Proceedings, UNIX Security Workshop II, August 1990.

62 Krakowiak S, Beeson D.Principles of Operating Systems. Cambridge, MA: MIT Press, 1988.

63 Krishna C, Lee Y, et al. Special Issue on Real-Time Systems. Proceedings of the IEEE, January 1994.

64 Kurzban S, Heines T, Kurzban S. Operating Systems Principles, 2nd ed. New York: Van Nostrand Reinhold, 1989.

65 Lamport L. The Mutual Exclusion Problem. Journal of the ACM, April 1986.

66 Lamport L. The Mutual Exclusion Problem Has Been Solved. Communications of the ACM, January 1991.

67 Lee Y, adn Krishna C, et al. Readings in Real-Time Systems. Los Alamitos, CA: IEEE Computer Society Press, 1993.

68 Leffler S, McKusick M, Karels M, et al. The Design and Implementation of the 4.3BSD UNIX Operating System. Reading, MA: Addison Wesley, 1988.

69 Leibfried T. A Deadlock Detection and Recovery Algorithm Using the Formalism of a Directed Graph Matrix. Operating Systems Review, April 1989.

70 Leland W, Ott T. Load Balancing Heuristics and Process Behavior. Proceedings, ACM Sig-Metrics Performance 1986 Conference, 1986.

71 Leutenegger S, Vernon M. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. Proceedings,

Conference on Measurement and Modeling of Computer Systems, May 1990.

72 Lister A, Eager R. Fundamentals of Operating Systems, 4th ed.London: Macmillan Education Ltd, 1996.

73 Livadas P. File Structures: Theory and Practice. Englewood Cliffs, NJ: Prentice Hall, 1990.

74 Maekawa M, Oldehoeft A, Oldehoeft, R. Operating Systems: Advanced Concepts. Menlo Park, CA: Benjamin Cummings, 1987.

75 Majumdar S, Eager D, Bunt R. Scheduling in Multiprogrammed Parallel Systems. Proceedings, Conference on Measurement and Modeling of Computer Systems, May 1988.

76 Martin J. Principles of Data Communication. Englewood Cliffs, NJ: Prentice Hall, 1988.

77 Milenkovic M. Operating Systems: Concepts and Design. New York: McGraw-Hill, 1996.

78 Morse S. Client/Server Is Not Always the Best Solution. Network Computing, Special Issue on Client/Server Computing,Spring 1993.

79 Moskowitz R. What Are Clients and Servers Anyway? Network Computing, Special Issue on Client/Server Computing, Spring 1993.

80 Nutt G. Centralized and Distributed Operating Systems. Englewood Cliffs, NJ: Prentice Hall, 1994.

81 Ousterhout J, et al. The Sprite Network Operating System. Computer, February 1988.

82 Panwar S, Towsley D, Wolf J. Optimal Scheduling Policies for a Class of Queues with Customer Deadlines in the Beginning of Service. Journal of the ACM, October 1988.

83 Pfleeger C. Security in Computing. Englewood Cliffs, NJ: Prentice Hall, 1989.

84 Pinkert J, Wear L. Operating Systems: Concepts, Policies, and Mechanisms. Englewood Cliffs, NJ: Prentice Hall, 1989.

85 Pizzarello A. Memory Management for a Large Operating System. Proceedings, International Conference on Measurement and Modeling of Computer Systems, May 1989.

86 Plambeck K. Concepts of Enterprise Systems Architecture. IBM Systems Journal, No. 1, 1989.

87 Ramamritham K, Stankovic J. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. Proceedings of the IEEE, January 1994.

88 Rashid R, et al. Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. IEEE Transactions on Computers, August 1988.

89 Rashid R, et al. Mach: A System Software Kernel. Proceedings, COMPCON Sprint'89, March 1989.

90 Raynal M. Algorithms for Mutual Exclusion. Cambridge, MA:MIT Press, 1986.

91 Raynal M. Distributed Algorithms and Protocols. New York: Wiley, 1988.

92 Raynal M, Helary, J. Synchronization and Control of Distributed Systems and Programs. New York: Wiley, 1990.

93 Ritchie D. The Evolution of the UNIX Time-Sharing System. AT&T Bell Laboratories Technical Journal, October 1984.

94 Silberschatz A, Galvin P. Operating System Concepts. Readings, MA: Addison Wesley, 1996.

95 Singhal M, Shivaratri N. Advanced Concepts in Operating Systems. New York: McGraw-Hill, 1996.

96 Smith J. A Survey of Process Migration Mechanisms. Operating Systems Review, July 1988.

97 Smith J. Implementing Remote fork() with Checkpoint/restart. Newsletter of the IEEE Computer Society Technical Committee on Operating Systems, Winter 1989.

98 Spafford E.Observing Reusable Password Choices. Proceedings, UNIX Security Symposium III, September 1992.

99 Stallings W. Data and Computer Communications, 4th ed. Englewood Cliffs, NJ: Prenticed Hall, 1994.

100 Stallings W. Network and Internetwork Security: Principles and Practice. Englewood Cliffs, NJ: Prentice Hall, 1995.

101 Stankovic J, Ramamrithan K. Hardd Real-Time Systems. Washington, DC: IEEE Computer Society Press, 1988.

102 Stankovic J, Ramamritham K, et al. Advances in Real-Time Systems. Los Alamitos, CA: IEEE Computer Society Press, 1993.

103 Stone H. High-Performance Computer Architecture, 2nd ed. Reading, MA: Addison Wesley, 1990.

104 Suzuki I, Kasami T. An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks. Proceedings of the Third International Conferences of Distributed Computing Systems, October 1982.

105 Tanenbaum A, Renesse R. Distributed Operating Systems. Computing Surveys, December 1995.

106 Tanenbaum A. Operating System Design and Implementation. Englewood Cliffs, NJ: Prentice Hall, 1996.

107 Tanenbaum A. Modern Operating Systems. Englewood Cliffs, NJ: Prentice Hall, 1997.

108 Tay B, Ananda A. A Survey of Remote Procedure Calls. Operating Systems Review, July 1990.

109 Tevanian A, et al. Mach Threads and the Unix Kernel: The Battle for Control. Proceedings, Summer 1987 USENIX Conference, June 1987.

110 Tevanian A, Smith B. Mach: The Model for Future Unix. Byte, November 1989.

111 Theaker C, Brookes G. A Practical Course on Operating Systems. New York: Springer-Verlag, 1993.

112 Tilborg A, Koob G, et al. Foundations of Real-Time Computing: Scheduling and Resource Management. Boston: Kluwer Academic Publishers, 1994.

113 Tucker A, Gupta A. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. Proceedings, Twelfth ACM Symposium on Operating Systems Principles, December 1989.

114 Turner R. Operating Systems: Design and Implementations. New York: Macmillan, 1996.

115 William S, Operating Systems (second edition) Prentice Hall International Inc., New Jersey, 1997.

116 何炎祥等. 计算机操作系统原理及其习题解答. 北京：海洋出版社, 1993.

117 何炎祥. 分布式操作系统设计。 北京：海洋出版社, 1993.

118 何炎祥. 并行程序设计方法. 北京：学苑出版社, 1995.

119 何炎祥等. 高级操作系统. 北京：科学出版社, 1999.

120 金玉琢等. 多媒体计算机技术基础及应用. 北京：高等教育出版社, 1999.

121 何炎祥等. 操作系统原理学习与解题指南. 武汉：华中科技大学出版社, 2001.

122 何炎祥等. 操作系统原理. 上海：上海科学技术文献出版社, 2000.

123 何炎祥，陈莘萌. Agent 和多 Agent 系统的设计与应用. 武汉：武汉大学出版社, 2001.