

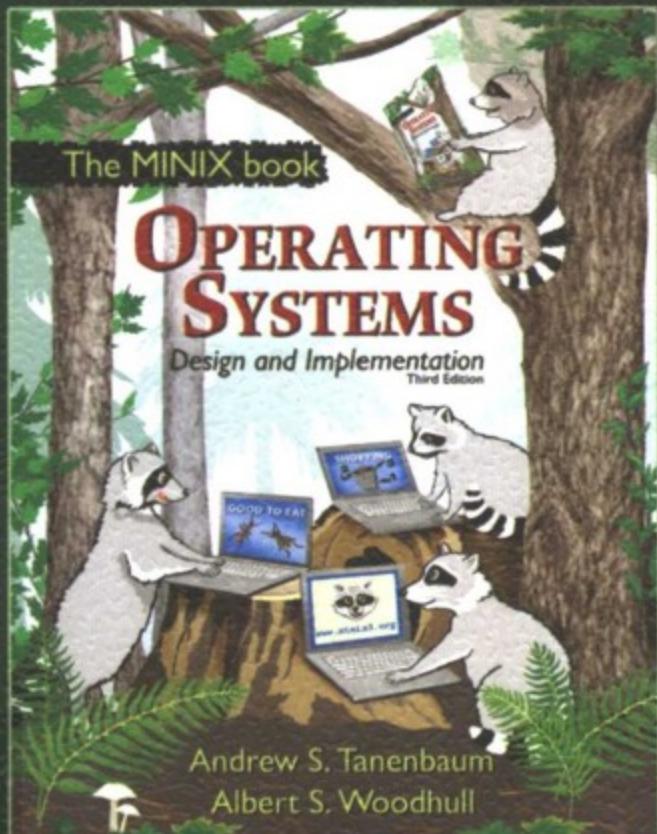
Tanenbaum's

操作系统设计与实现

(第三版) 上册

Operating Systems

Design and Implementation, Third Edition



[美] Andrew S. Tanenbaum 著
Albert S. Woodhull
陈渝 谌卫军 译
向勇 审校



电子工业出版社

Publishing House of Electronics Industry

<http://www.phei.com.cn>

TP316
398
:1
2007

国外计算机科学教材系列

操作系统设计与实现

(第三版)(上册)

Operating Systems

Design and Implementation, Third Edition

[美] Andrew S. Tanenbaum 著
Albert S. Woodhull

陈渝 谌卫军 译

向勇 审校



内 容 简 介

本书是关于操作系统的权威教材。大多数关于操作系统的图书均重理论而轻实践，而本书则在这两者之间进行了较好的折中。本册详细探讨了操作系统的基本原理，包括进程、进程间通信、信号量、管程、消息传递、调度算法、输入/输出、死锁、设备驱动程序、存储管理、调页算法、文件系统设计、安全和保护机制等；此外，还详细讨论了一个特殊的操作系统MINIX 3（一个与UNIX兼容的操作系统），并提供了该系统的源代码（见本书下册），以便于读者仔细研究。这种安排不仅可让读者了解操作系统的基本原理，而且可让读者了解到这些基本原理是如何应用到真实的操作系统中去的。

本书适用于高校计算机专业的学生，也可供程序设计人员、工程技术人员、系统架构师等相关人员参考。

Simplified Chinese edition Copyright © 2006 by PEARSON EDUCATION ASIA LIMITED and Publishing House of Electronics Industry.

Operating Systems: Design and Implementation, Third Edition, ISBN: 0131429388 by Andrew S. Tanenbaum and Albert S. Woodhull. Copyright © 2006.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书中文简体字翻译版由电子工业出版社和Pearson Education培生教育出版亚洲有限公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2006-0890

图书在版编目（CIP）数据

操作系统设计与实现·上册：第3版 / (美)塔嫩鲍姆 (Tanenbaum, A. S.) 等著；陈渝，谌卫军译。
北京：电子工业出版社，2007.3
(国外计算机科学教材系列)

书名原文：Operating Systems: Design and Implementation, Third Edition
ISBN 978-7-121-03381-0

I. 操… II. ①塔… ②陈… ③谌… III. 操作系统—教材 IV. TP316

中国版本图书馆CIP数据核字(2006)第130349号

责任编辑：谭海平

印 刷：北京市海淀区四季青印刷厂

装 订：北京牛山世兴印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×1092 1/16 印张：29.5 字数：830千字

印 次：2007年3月第1次印刷

定 价：49.80元

凡所购买电子工业出版社的图书有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系电话：(010) 68279077。邮购电话：(010) 88254888。

质量投诉请发邮件至zlt@phe.com.cn，盗版侵权举报请发邮件至dbqq@phe.com.cn。

服务热线：(010) 88258888。

出版说明

21世纪初的5至10年是我国国民经济和社会发展的重要时期，也是信息产业快速发展的关键时期。在我国加入WTO后的今天，培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡，是我国面对国际竞争时成败的关键因素。

当前，正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期，为使我国教育体制与国际化接轨，有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材，以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验，翻译出版了“国外计算机科学教材系列”丛书，这套教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时，我们也适当引进了一些优秀英文原版教材，本着翻译版本和英文原版并重的原则，对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上，我们大都选择国外著名出版公司出版的高校教材，如Pearson Education培生教育出版集团、麦格劳-希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者，如道格拉斯·科默(Douglas E. Comer)、威廉·斯托林斯(William Stallings)、哈维·戴特尔(Harvey M. Deitel)、尤利斯·布莱克(Uyless Black)等。

为确保教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士，也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括对所选教材进行全面论证；选择编辑时力求达到专业对口；对排版、印制质量进行严格把关。对于英文教材中出现的错误，我们通过与作者联络和网上下载勘误表等方式，逐一进行了修订。

此外，我们还将与国外著名出版公司合作，提供一些教材的教学支持资料，希望能为授课老师提供帮助。今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

主任	杨芙清	北京大学教授 中国科学院院士 北京大学信息与工程学部主任 北京大学软件工程研究所所长
委员	王 珊	中国人民大学信息学院院长、教授
	胡道元	清华大学计算机科学与技术系教授 国际信息处理联合会通信系统中国代表
	钟玉琢	清华大学计算机科学与技术系教授、博士生导师 清华大学深圳研究生院信息学部主任
	谢希仁	中国人民解放军理工大学教授 全军网络技术研究中心主任、博士生导师
	尤晋元	上海交通大学计算机科学与工程系教授 上海分布计算技术中心主任
	施伯乐	上海国际数据库研究中心主任、复旦大学教授 中国计算机学会常务理事、上海市计算机学会理事长
	邹 鹏	国防科学技术大学计算机学院教授、博士生导师 教育部计算机基础课程教学指导委员会副主任委员
	张昆藏	青岛大学信息工程学院教授

译 者 序

Andrew S. Tanenbaum是著名的计算机科学家、教育家，荷兰皇家科学艺术院院士，也是IEEE会士和ACM会士，目前供职于荷兰阿姆斯特丹Vrije大学。他在操作系统、分布式系统和计算机网络等领域都有很深的造诣，曾多次获奖，包括1994年度ACM Karl V. Karlstrom杰出教育家奖、1997年度ACM CSE计算机科学教育杰出贡献奖、2002年度TAA优秀教材奖和2003年度TAA McGuffey奖。

20世纪80年代，出于教学工作的需要，Tanenbaum教授开发了一个小巧、完整、开放源代码、UNIX兼容的操作系统MINIX，使学生可以通过剖析这个“麻雀虽小，五脏俱全”的系统，来研究其内部的运作机理。为了便于学习，他还出版了相应的教材，即本书的第一版。经过20多年的发展，MINIX系统在许多方面得到了改进，如对现代主流硬件设备的支持、对POSIX标准的支持、微内核系统结构等；与之相对应，本书也不断推陈出新，在2006年出版了第三版。

本书的最大特点就是理论与实践的完美结合。在多年的教学实践中，我们深刻地体会到，对于操作系统这样一门实用性和实践性很强的课程，如果只是单纯地介绍它的基本原理和基本概念，很难有非常理想的教学效果。一个连进程的创建函数都没有用过的人，很难想象他能对进程与线程之间的区别有真正的了解。同样，一个没有分析过内存分配源代码的人，也很难对虚拟存储管理有太多深入的理解。而本书的出现则弥补了这个缺陷，在理论与实践之间，搭起了一座桥梁。本书涵盖了操作系统课程的所有内容，包括进程管理、存储管理、文件系统和设备管理等。对于每一个章节，在组织结构上采用了从浅到深、从抽象到具体、从宏观到细节的讲授方式。首先从总体上介绍操作系统的根本原理和基本概念，然后结合MINIX 3系统，深入探讨这些基本原理的具体实现过程，最后再以源代码的形式给出了所有的实现细节。通过这种自顶向下、逐步求精的学习过程，使读者能够做到融会贯通。在面对抽象、枯燥的理论时，能够用技术实现来加以印证、加深理解；在面对复杂、繁琐的源代码时，能够用理论思想来进行指导。相信这样的一种学习模式，对于读者深入掌握操作系统的原理、设计与实现，是大有裨益的。

本书的另一个特点是实用性。如果说MINIX 1和MINIX 2还主要是用于教学目的，那么MINIX 3则完全不同。它的设计目标是一个实用的、具有高可靠性、灵活性和安全性的系统，能够运行在一些资源有限或者是嵌入式的硬件平台上。系统采用微内核结构，内核代码仅有4000行左右，而设备驱动程序等模块则作为普通的用户进程运行，这种结构大大提高了系统的可靠性，读者只要加以修改，就可以移植到自己的硬件平台上。

基于上述原因，我们认为翻译本书、把它介绍给国内的读者是一件非常有意义的事情，衷心希望我们付出的劳动能对国内的操作系统教学和实践有所帮助和促进。

本书的第2章、第3章由陈渝翻译并统稿，第1章、第4章、第5章、第6章由谌卫军翻译，向勇对全书进行了审校。在整个翻译过程中，清华大学计算机系和软件学院的师生给予了帮助，并且在计算机系和软件学院的本科生的操作系统课程中进行了试用，许多学生提出了很好的建议，在此向他们表示衷心的感谢。

译者

2006年11月于清华园

本书定稿前的部分内容得到了广州海洋大学计算机系主任高升教授的审阅——编者注。

作者简介

Andrew S. Tanenbaum

Andrew S. Tanenbaum 分别在麻省理工学院和加州大学伯克利分校获得学士与博士学位。现任荷兰阿姆斯特丹 Vrije 大学计算机科学教授并领导着一个计算机系统研究小组。到 2005 年 1 月卸任为止，他担任计算与成像高级学院院长一职已有 12 年。

Tanenbaum 过去的研究领域包括编译器、操作系统、网络和局域分布式系统，而现在的研究方向则主要为计算机安全，尤其是操作系统、网络以及分布式系统的安全。在所有这些研究领域，Tanenbaum 发表了超过 100 篇论文，并出版了 5 本书籍。

Tanenbaum 教授还编写了大量软件。他是 Amsterdam Compiler Kit（一种广泛使用的、用于编写可移植编译器以及 MINIX 的工具集）的主要开发者，而该系统则是 Linux 诞生的灵感与基础。与他的博士生及程序员一起，他帮助设计了 Amoeba 分布式操作系统（一个基于微内核的、高性能局域分布式操作系统）。此后，他是 Globe（一个可处理 10 亿用户的广域分布式操作系统）的设计者之一。所有这些软件现在均可在互联网上免费获得。

他的博士生在毕业后均取得了很大的成绩，他为此感到非常骄傲。

Tanenbaum 教授是 ACM 会士、IEEE 会士以及荷兰皇家科学艺术院院士。他还是 1994 年度 ACM Karl V. Karlstrom 杰出教育家奖的获得者，1997 年度 ACM/SIGCSE 计算机科学教育杰出贡献奖的获得者，以及 2002 年度优秀教材奖的获得者。2004 年，他被推选为荷兰皇家学会的五位新学会教授之一。他的主页地址为 <http://www.cs.vu.nl/~ast/>。

Albert S. Woodhull

Albert S. Woodhull 在麻省理工学院获得学士学位，在华盛顿大学获得博士学位。在入读麻省理工学院时，他希望能成为一位电气工程师，但最后却成了一位生物学家。20 多年来，他一直工作于麻省 Amherst 的 Hampshire 自然科学院。当微型计算机慢慢多起来的时候，作为使用电子检测仪器的生物学家，他开始使用微型计算机。他给学生开设的仪器检测方面的课程逐渐演变为计算机接口和实时程序设计课程。

Woodhull 博士对教学和科学技术的发展有浓厚的兴趣，在进入研究生院之前，他曾在尼日利亚教过两年中学，近年来他曾几次利用自己的假期在尼加拉瓜讲授计算机科学。

他对计算机作为电子系统，以及计算机与其他电子系统的相互配合很感兴趣。他最喜欢讲授的课程有计算机体系结构、汇编语言程序设计、操作系统和计算机通信。他还为开发电子器件及相关软件担任顾问。

在学术之外，Woodhull 还有不少兴趣，包括各种户外活动、业余无线电制作和读书。他还喜欢旅游和学习别国语言。他的主页放在一台运行 MINIX 的机器上，地址为 <http://minix1.hampshire.edu/asw/>。

序 言

大多数关于操作系统的图书均重理论而轻实践，而本书则在这两者之间进行了较好的折中。本书详细探讨了操作系统的基本原理，包括进程、进程间通信、信号量、管程、消息传递、调度算法、输入/输出、死锁、设备驱动程序、存储管理、调页算法、文件系统设计、安全和保护机制等；此外，还详细讨论了一个特殊的操作系统 MINIX 3（一个与 UNIX 兼容的操作系统），甚至提供了该系统的源代码，以便于读者仔细研究。这种安排不仅可让读者了解操作系统的基本原理，而且可让读者了解到这些基本原理是如何应用到真实的操作系统中去的。

自本书第一版于 1987 年推出以来，操作系统课程的教学方式已有了较小的革新。在此之前，大多数课程都只讲授理论知识。随着 MINIX 的出现，许多学校开始开设实验课程，以便学生仔细研究操作系统，了解其内部工作机理。对于这种教学方式，我们相当欣慰并希望能继续加强这种趋势。

MINIX 在推出以来的 10 年里经历了许多变化。最初的代码是为基于 8088 的有着 256 KB 内存和两个磁盘驱动器的 IBM PC 机（没有硬盘）开发的，它基于 UNIX V7。随着时间的推移，MINIX 在许多方面已得到了改进，例如它支持有着大内存和多个硬盘的 32 位保护模式的机器，而且它不再基于 UNIX V7，而基于国际 POSIX 标准（IEEE 1003.1 和 ISO 9945-1）。与此同时，UNIX 中添加了许多新特性，在我们看来，这些新添加的特性可能很多，但在某些人看来这些新特性还不够，因而导致了 Linux 的出现。此外，MINIX 还被移植到了许多其他的平台，包括 Macintosh, Amiga, Atari 和 SPARC。本书的第二版于 1997 年出版并已在大学中广泛使用。

MINIX 仍在得到人们的欢迎，这可以通过 Google 上对 MINIX 的搜索次数看出。

本书的第三版已有了很大的变化。我们修订了关于原理的几乎所有内容，并添加了许多新内容，但主要变化是对新操作系统 MINIX 3 的讨论以及本书中包含的新代码。尽管仍松散地基于 MINIX 2，但 MINIX 3 在几个主要的方面几乎完全不同。

设计 MINIX 3 的目的基于如下事实：操作系统正变得越来越大、越来越慢以及越来越不可靠。与其他电子设备如电视、手机以及 DVD 播放器相比，操作系统变得越来越容易崩溃，加之具有许多特性和选项，实际上几乎没有人能完全理解它们，或将它们管理得很好。当然，计算机病毒、蠕虫、间谍软件以及其他形式的恶意程序也已变得越来越猖獗，这对操作系统无疑会造成较大的影响。

在很大程度上，这些问题均是由当前操作系统中的基本设计缺陷引起的：模块性的缺失。整个操作系统一般会由几百万行 C/C++ 代码组成，这些代码被编译到了一个在内核态下运行的巨大可执行程序中。在几百万行代码中，即使只有一行代码存在缺陷，也会导致系统发生故障。使所有这些代码均正确是不可能的，尤其是当 70% 的代码是由设备驱动程序组成时，因为这些设备驱动程序是由第三方编写的，而这已超出了操作系统编写者的控制范围。

通过使用 MINIX 3，我们证明了在设计操作系统时，并不是只有整体式设计这一种方法。MINIX 3 内核仅包含有约 4000 行可执行代码，而不是 Windows, Linux, Mac OS X 或 FreeBSD 的数百万行代码。系统中的其他内容，包括所有的设备驱动程序（除时钟驱动程序外），是一个模块化的用户模式进程的小集合，对于每个进程，我们只关注其作用以及与之通信的其他进程。

尽管 MINIX 3 仍在改进，但我们相信这个将操作系统构建为高度封装的用户模式进程的模型完全可以在将来用于构建更为可靠的系统。MINIX 3 适用于小型 PC 机（如那些可在第三世界国家

找到的机器以及嵌入式系统，这些机器的资源均有限）。无论如何，这种设计可使得学生更易于了解操作系统的工作原理，而不必去试图研究巨大的整体式操作系统。

本书附带的 CD-ROM 是一张可引导 CD。读者可将该 CD-ROM 放到驱动器中，重新启动计算机，之后 MINIX 3 会在几秒钟后显示登录提示符。读者可登录为 root 用户来启动系统，而不用将 MINIX 3 首先安装到硬盘中。当然，MINIX 3 也可安装到硬盘上。详细的安装信息请参阅附录 A。

像上面建议的那样，MINIX 3 正在快速发展，新版本正在频繁地推出。要下载当前的 CD-ROM 映像文件来刻录光盘，请进入 MINIX 的官方网站 www.minix3.org。该网站还包含有大量的新软件、文献，以及关于 MINIX 3 开发的新闻。关于 MINIX 3 的讨论以及频繁问及的问题，读者可进入 USENET 新闻组 *comp.os.minix* 了解详情。没有新闻阅读器的人们可按网站 <http://groups.google.com/group/comp.os.minix> 上的说明操作。

MINIX 3 除了可安装到硬盘上来运行外，也可在几个 PC 模拟器上运行。网站的主页上列出了一些模拟器。

使用本书作为教材的教师，可通过 Prentice Hall 出版公司在当地的机构来获得习题解答（索取方式请参阅书后的“教学支持说明”）。此外，本书有其自己的配套网站，网址为 www.prenhall.com/tanenbaum。

在本书的编写过程中，我们有幸得到了许多人的帮助。Ben Gras 和 Jorrit Herder 在时间仓促的情况下，完成了新版本操作系统的大部分编程工作；此外，他们还阅读了本书的初稿并做了许多有用的注释。在此，我们要对他们深表谢意。

Kees Bot 曾为 MINIX 2 做了许多工作，这为我们开发 MINIX 3 奠定了很好的基础。Kees 为 MINIX 的最初版本至版本 2.0.4 编写了大量的代码，修复了缺陷，并回答了许多问题。Philip Homburg 也编写了大量的代码，并在其他方面也给予了我们大量的帮助，尤其是在初稿的审读方面。

当然，MINIX 的推出还得到了其他许多人的帮助，在此我们一并表示致谢。

一些人阅读了部分初稿并提出了修订建议。在此要特别感谢 Gojko Babic, Michael Crowley, Joseph M. Kizza, Sam Kohn Alexander Manov 和 Du Zhang。

最后，我们要感谢我们的家庭。Suzanne 已是第 16 次在我埋头写作时给予我支持，Barbara 已是第 15 次，而 Marvin 已是 14 次。他们的支持和爱心对我非常重要（AST）。

Al's Barbara 已是第二次在我写作时给我支持。若没有她的支持、耐心，我们是不可能完成这项工作的。我的儿子 Gordon 一直是一位有耐心的倾听者，能得到儿子的理解与支持，对我而言无疑十分欣慰。最后，教父 Zain 的生日恰好就是 MINIX 3 的发布日期，也许某一天他会对此感到高兴的（ASW）。

Andrew S. Tanenbaum
Albert S. Woodhull

目 录

第1章 引言	1
1.1 什么是操作系统	3
1.1.1 操作系统作为扩展机	3
1.1.2 操作系统作为资源管理器	3
1.2 操作系统的发展历史	4
1.2.1 第一代计算机 (1945-1955): 真空管和插接板	5
1.2.2 第二代计算机 (1955-1965): 晶体管和批处理系统	5
1.2.3 第三代计算机 (1965-1980): 集成电路和多道程序	6
1.2.4 第四代计算机 (1980-): 个人计算机	10
1.2.5 MINIX 3 的历史	11
1.3 操作系统概念	13
1.3.1 进程	14
1.3.2 文件	15
1.3.3 命令解释器	17
1.4 系统调用	18
1.4.1 进程管理的系统调用	20
1.4.2 信号管理的系统调用	22
1.4.3 文件管理的系统调用	24
1.4.4 目录管理的系统调用	27
1.4.5 保护的系统调用	29
1.4.6 时间管理的系统调用	30
1.5 操作系统结构	30
1.5.1 整体结构	31
1.5.2 分层结构	33
1.5.3 虚拟机	33
1.5.4 外核	35
1.5.5 客户 - 服务器模型	36
1.6 剩余各章内容简介	37
1.7 小结	37
习题	37
第2章 进程	39
2.1 进程介绍	39
2.1.1 进程模型	39
2.1.2 进程的创建	40

2.1.3	进程的终止	41
2.1.4	进程的层次结构	42
2.1.5	进程的状态	43
2.1.6	进程的实现	44
2.1.7	线程	45
2.2	进程间通信	48
2.2.1	竞争条件	48
2.2.2	临界区	49
2.2.3	忙等待形式的互斥	50
2.2.4	睡眠和唤醒	53
2.2.5	信号量	55
2.2.6	互斥	57
2.2.7	管程	57
2.2.8	消息传递	60
2.3	经典 IPC 问题	62
2.3.1	哲学家进餐问题	62
2.3.2	读者 - 写者问题	65
2.4	进程调度	66
2.4.1	调度介绍	66
2.4.2	批处理系统中的调度	69
2.4.3	交互式系统中的调度	72
2.4.4	实时系统调度	76
2.4.5	策略与机制	76
2.4.6	线程调度	77
2.5	MINIX 3 进程概述	78
2.5.1	MINIX 3 的内部结构	78
2.5.2	MINIX 3 中的进程管理	80
2.5.3	MINIX 3 中的进程间通信	83
2.5.4	MINIX 3 中的进程调度	85
2.6	MINIX 3 中进程的实现	86
2.6.1	MINIX 3 源代码的组织	86
2.6.2	编译及运行 MINIX 3	88
2.6.3	公共头文件	90
2.6.4	MINIX 3 头文件	95
2.6.5	进程数据结构和头文件	101
2.6.6	引导 MINIX 3	107
2.6.7	系统初始化	110
2.6.8	MINIX 的中断处理	114
2.6.9	MINIX 3 的进程间通信	121
2.6.10	MINIX 的进程调度	124

2.6.11	与硬件相关的内核支持	126
2.6.12	实用程序和内核库	129
2.7	MINIX 3 的系统任务	131
2.7.1	系统任务综述	132
2.7.2	系统任务的实现	134
2.7.3	系统库的实现	136
2.8	MINIX 3 的时钟任务	138
2.8.1	时钟硬件	139
2.8.2	计时程序	140
2.8.3	MINIX 3 中的时钟驱动程序总览	142
2.8.4	MINIX 3 中的时钟驱动程序的应用	144
2.9	小结	145
	习题	146

第3章	输入/输出系统	150
3.1	I/O 硬件原理	150
3.1.1	I/O 设备	150
3.1.2	设备控制器	151
3.1.3	内存映射 I/O	152
3.1.4	中断	153
3.1.5	直接存储器存取	154
3.2	I/O 软件的原理	155
3.2.1	I/O 软件的目标	155
3.2.2	中断处理器	156
3.2.3	设备驱动程序	157
3.2.4	与设备无关的 I/O 软件	158
3.2.5	用户空间的 I/O 软件	160
3.3	死锁	161
3.3.1	资源	161
3.3.2	死锁的原理	162
3.3.3	鸵鸟算法	165
3.3.4	死锁的检测和恢复	166
3.3.5	死锁的预防	166
3.3.6	避免死锁	168
3.4	MINIX 3 中的 I/O 概述	171
3.4.1	MINIX 3 中的中断处理器和 I/O 访问	171
3.4.2	MINIX 3 的设备驱动程序	173
3.4.3	MINIX 3 中与设备无关的 I/O 软件	176
3.4.4	MINIX 3 中的用户级 I/O 软件	176
3.4.5	MINIX 3 的死锁处理	177
3.5	MINIX 3 中的块设备	177

3.5.1	MINIX 3 中的块设备驱动程序概述	177
3.5.2	通用块设备驱动程序软件	180
3.5.3	驱动程序库	182
3.6	RAM 盘	183
3.6.1	RAM 盘硬件和软件	184
3.6.2	MINIX 3 中的 RAM 盘驱动程序概述	185
3.6.3	MINIX 3 中 RAM 盘驱动程序的实现	186
3.7	磁盘	188
3.7.1	磁盘硬件	188
3.7.2	RAID	189
3.7.3	磁盘软件	190
3.7.4	MINIX 3 中的硬盘驱动程序简介	194
3.7.5	MINIX 3 中硬盘驱动程序的实现	196
3.7.6	软盘处理	203
3.8	终端	204
3.8.1	终端硬件	205
3.8.2	终端软件	208
3.8.3	MINIX 3 中的终端驱动程序简介	213
3.8.4	设备无关终端驱动程序的实现	224
3.8.5	键盘驱动程序的实现	236
3.8.6	显示驱动程序的实现	241
3.9	小结	246
	习题	247

第 4 章	存储管理	251
4.1	基本的存储管理	251
4.1.1	单道程序存储管理	252
4.1.2	固定分区的多道程序系统	252
4.1.3	重定位和存储保护	254
4.2	交换技术	255
4.2.1	基于位图的存储管理	257
4.2.2	基于链表的存储管理	257
4.3	虚拟存储管理	259
4.3.1	虚拟页式存储管理	260
4.3.2	页表	263
4.3.3	关联存储器 TLB	266
4.3.4	反置页表	268
4.4	页面置换算法	269
4.4.1	最优页面置换算法	270
4.4.2	最近未使用页面置换算法	270
4.4.3	先进先出页面置换算法	271

4.4.4 第二次机会页面置换算法	271
4.4.5 时钟页面置换算法	272
4.4.6 最近最久未使用页面置换算法	273
4.4.7 LRU 算法的软件模拟	273
4.5 页式存储管理中的设计问题	275
4.5.1 工作集模型	275
4.5.2 局部与全局分配策略	277
4.5.3 页面大小	279
4.5.4 虚拟存储器接口	280
4.6 段式存储管理	281
4.6.1 纯分段系统的实现	283
4.6.2 段页式存储管理: Intel Pentium	284
4.7 MINIX 3 进程管理器概述	287
4.7.1 内存布局	288
4.7.2 消息处理	291
4.7.3 进程管理的数据结构和算法	292
4.7.4 FORK, EXIT 和 WAIT 系统调用	296
4.7.5 EXEC 系统调用	297
4.7.6 BRK 系统调用	300
4.7.7 信号处理	300
4.7.8 其他的系统调用	306
4.8 MINIX 3 进程管理器的实现	306
4.8.1 头文件和数据结构	306
4.8.2 主程序	309
4.8.3 FORK, EXIT 和 WAIT 的实现	312
4.8.4 EXEC 的实现	314
4.8.5 BRK 的实现	317
4.8.6 信号处理的实现	317
4.8.7 其他系统调用的实现	323
4.8.8 内存管理工具	326
4.9 小结	327
习题	327
第 5 章 文件系统	331
5.1 文件	331
5.1.1 文件的命名	332
5.1.2 文件的结构	333
5.1.3 文件的类型	334
5.1.4 文件的访问	336
5.1.5 文件的属性	336
5.1.6 文件的操作	337

5.2	目录	338
5.2.1	简单的目录系统	338
5.2.2	层状目录系统	339
5.2.3	路径名	340
5.2.4	目录的操作	342
5.3	文件系统的实现	342
5.3.1	文件系统的布局	342
5.3.2	文件的实现	344
5.3.3	目录的实现	347
5.3.4	磁盘空间管理	351
5.3.5	文件系统的可靠性	354
5.3.6	文件系统的性能	359
5.3.7	日志结构的文件系统	362
5.4	文件系统的安全性	363
5.4.1	安全环境	364
5.4.2	通常的安全攻击	367
5.4.3	安全性的设计原则	368
5.4.4	用户认证	368
5.5	保护机制	371
5.5.1	保护域	371
5.5.2	访问控制列表	373
5.5.3	权能	375
5.5.4	秘密通道	377
5.6	MINIX 3 文件系统概述	379
5.6.1	消息	380
5.6.2	文件系统的布局	381
5.6.3	位图	383
5.6.4	i 节点	384
5.6.5	块高速缓存	386
5.6.6	目录和路径	387
5.6.7	文件描述符	389
5.6.8	文件锁	390
5.6.9	管道和设备文件	391
5.6.10	一个例子：READ 系统调用	392
5.7	MINIX 3 文件系统的实现	392
5.7.1	头文件和全局数据结构	393
5.7.2	表格管理	395
5.7.3	主程序	401
5.7.4	对单个文件的操作	404
5.7.5	目录和路径	410
5.7.6	其他的系统调用	412

5.7.7 I/O 设备接口	414
5.7.8 附加的系统调用支持	418
5.7.9 文件系统的实用程序	419
5.7.10 其他的 MINIX 3 组件	420
5.8 小结	420
习题	421
第 6 章 阅读材料和参考文献	424
6.1 推荐的进一步阅读材料	424
6.1.1 介绍和概论	424
6.1.2 进程	426
6.1.3 输入 / 输出	426
6.1.4 存储管理	427
6.1.5 文件系统	427
6.2 按字母顺序排列的参考文献	428
索引	436

第1章 引言

- 1.1 什么是操作系统
- 1.2 操作系统的发展历史
- 1.3 操作系统概念
- 1.4 系统调用
- 1.5 操作系统结构
- 1.6 剩余各章内容简介
- 1.7 小结

众所周知，软件是计算机系统的灵魂，没有软件的计算机就像一个漂亮的花瓶，虚有其表。有了软件，计算机可以存储、处理和检索信息，可以播放音乐和电影，可以发送电子邮件、搜索 Internet，可以做许多有意思的事情。计算机软件大致可以分为两类，即系统软件和应用软件。系统软件负责管理计算机本身的运作，而应用软件则负责完成用户所需要的各种功能。最基本的系统软件是操作系统（Operating System, OS），它负责管理计算机的所有资源并提供一个可以在其上编写应用程序的平台。本书讨论的就是操作系统这种软件，我们将以 MINIX 3 系统为模型，来阐述操作系统的概念、设计原理，以及如何来实现一个真正的系统。

现代计算机系统中包含有各种不同类型的硬件设备，如处理器（一个或多个）、内存、磁盘、打印机、键盘、显示器、网络接口以及其他输入/输出设备。总之，这是一个复杂的系统。在这种情形下，如果要编写程序来管理所有的这些组件并正确地使用它们（暂且不论性能的优化），则将是一件非常困难的事情。因此，对于每一个程序员来说，当他在编写程序的时候，如果要考虑到该方面的所有问题，例如磁盘驱动器的工作原理、在读取一个磁盘数据块时哪些环节容易发生错误等，那么许多程序基本上无法编写。

在许多年以前，人们就已经认识到必须找到某种方法，让程序员从复杂、繁琐的硬件操作中解脱出来。经过不断探索和改进，目前的做法是在裸机上引入一层软件，让它来管理系统的各个部件，并给上层的用户提供一个易于理解和编程的接口〔或者称为虚拟机（virtual machine）〕。这样的一层软件就是操作系统。

操作系统的定位如图 1.1 所示。在计算机系统的底层是硬件。在许多情形下，硬件本身又可以分为两层或多层。最底层是物理设备，包括集成电路芯片、线路、电源、阴极射线管等。至于物理设备的内部结构和工作原理，则属于电气工程的范畴，本书不予详述。



图 1.1 计算机系统由硬件、系统程序和应用程序组成

接下来是微体系结构层（microarchitecture level）。在这一层中，各个物理设备被组织成一些功能单元，如中央处理器（Central Processing Unit, CPU）内部的一些寄存器、涉及算术逻辑单元的

数据流程等。在每个时钟周期，一个或两个操作数将被从寄存器中取出，并在算术逻辑单元中进行运算（如加法运算或逻辑“与”运算），运算的结果保存在一个或多个寄存器中，这样就完成了一次数据流程。在有些机器上，数据流程的运转是由软件来控制的，这种软件称为微程序（microprogram）。而在另外一些机器上，这个过程是由硬件电路直接来控制的。

数据流程的目的是为了执行一组指令，有些指令可以在一个数据流程周期中完成，而有些则可能需要多个周期。指令在执行时，可以使用寄存器或其他的运算部件。一般来说，这些指令的功能和用法是公开的，汇编语言程序员可以看到。硬件和指令在一起，就组成了指令集体体系结构（Instruction Set Architecture, ISA）。这一层通常也称为机器语言。

机器语言通常有 50~300 条指令，其中多数用来完成数据传送、算术运算和数值比较等操作。在这个层次上，可以通过向特殊的设备寄存器（device register）写入特定的数值来控制输入/输出设备。例如，为了读取磁盘数据，可以将磁盘地址、内存地址、需要读取的字节数和数据传送方向（读或写）等值写入到相应的寄存器中。当然，为了叙述方便，这里进行了简化。实际上，为了完成一次磁盘的读取操作，往往需要更多的参数，而且当操作完成后，设备返回的状态也比较复杂。另外，对于许多输入/输出（Input/Output, I/O）设备，在编写控制程序的时候还要考虑到时间的因素。

操作系统的一个主要功能就是将硬件的这些复杂性封装起来，给程序员提供一个更方便的编程接口。例如，从概念上讲，“从文件中读取一个数据块”就比“移动磁头，等待旋转延迟”之类的细节来得简单、方便。

操作系统之上是其他的系统软件，包括命令解释器（shell）、窗口系统、编译器、编辑器及类似的独立于应用的程序。这些程序通常在计算机出厂时就已经安装好了，或者是和操作系统装在同一个盒子里，用户在购买后可以自行安装。但需要指出的是，这些程序本身并不是操作系统的组成部分，这一点很重要。所谓操作系统，一般是指在内核态（kernel mode）或称管态（supervisor mode）下运行的软件，它受到硬件的保护，用户不能随便去篡改它的内容。当然，在一些老的或低端的微处理器上，可能没有硬件保护机制。而对于编译器和编辑器这些系统软件，它们运行在用户态（user mode）。如果用户不喜欢某一个编译器，他^①完全可以另起炉灶，自己去写一个！但如果是时钟中断处理程序，那么用户就不能这么做，因为它是操作系统的一部分，硬件保护机制会阻止用户去对它进行修改。

不过，在有些系统中，这种区别也不是很明显。例如，在有些嵌入式系统中，可能根本就没有内核态；而在一些解释系统中，如 Java，它们使用解释器而不是硬件来划分各个功能组件。尽管如此，对于传统的计算机，操作系统仍然指的是运行在内核态的软件。

在许多系统中，有一些运行在用户态的程序，但它们却是为操作系统服务的，或者是用来实现一些特权功能的。例如，在系统中通常会有一个允许用户去修改其密码的程序。这个程序并不是操作系统的一部分，也不运行在内核态，但是它实现的功能非常敏感，必须用某种特殊的方法来进行保护。

在 MINIX 3 等操作系统中，也采用了类似的思路。有一些在传统意义上属于操作系统的功能，也运行在用户空间中。因此，在这种情形下，要想在操作系统和其他系统软件之间画上一条清晰的边界，是比较困难的。诚然，运行在内核态的所有代码都属于操作系统，但有些运行在用户态的程序也可以视为操作系统的一部分，至少跟它是密切相关的。例如，在 MINIX 3 中，文件系统就是运行在用户态的一个 C 程序。

系统软件之上是各种应用软件。这些软件是用户购买或自行开发的，用来解决特定的问题，如字处理、表格处理、工程计算或者在数据库中存储信息等。

^① 这里的“他”指的是“他”或“她”，后文均如此，不再一一说明。

1.1 什么是操作系统

大多数计算机用户都曾经使用过操作系统，但如果要精确地给出操作系统的定义却很困难。这可能是因为操作系统具有两个互不相关的功能：扩展机和资源管理。由于说者的不同，听者有时听到的是操作系统的一个功能，而有时听到的是另一个功能，这就使得听者容易感到迷惑。下面我们分别对这两个功能进行讨论。

1.1.1 操作系统作为扩展机

如前所述，对多数计算机而言，在机器语言一级的体系结构（指令集、存储组织、I/O 和总线结构）上编程是比较困难的，尤其是输入输出操作。例如，在许多基于 Intel 处理器的 PC 机上，采用的是 NEC PD765 兼容的磁盘控制器芯片，我们以它为例来讨论软盘的输入输出过程。PD765 有 16 条命令，每条命令都将一定长度的数据（1 到 9 个字节）装入到某个设备寄存器中。这些命令主要用来读写数据、移动磁头臂、格式化磁道、初始化、检测磁盘状态、复位、校准控制器和驱动器等。

最基本的命令是读操作和写操作，它们都需要 13 个参数，这些参数被封装在 9 个字节中。具体内容包括欲读取的磁盘块地址、每条磁道的扇区个数、物理介质的数据记录格式、扇区间的空隙大小以及对已删除数据地址标识的处理方法等。如果读者对这些稀奇古怪的名词术语感到困惑不解，这就对了。因为对于不熟悉的人来说，硬件就是这样晦涩难懂。当读写操作完成时，控制器芯片将返回 23 个状态及出错信息，它们被封装在 7 个字节中。不仅如此，程序员还必须时刻注意电机的开关状态。如果电机关闭，那么在读写数据前要先启动它（有较长的一段启动延迟时间）。但又不能让它长时间处于开启状态，否则将会损坏软盘。这样一来，就使程序员陷入了两难的境地：如果经常关闭电机，那么每次启动都需要较长的延迟时间；如果不关闭电机，又可能会损坏软盘，造成数据的丢失。所以只能在这两者之间折中。

显然，对于通常的程序员来说，肯定不愿意涉及太多的软盘编程细节（硬盘也一样，它与软盘不同，但同样很复杂），他需要的是一种简单的、高度抽象的接口。对于磁盘而言，一种典型的抽象是：磁盘内包含了一组文件，每个文件都有一个文件名。在访问一个文件之前，首先要打开这个文件，然后才能对它进行读写操作。在使用完文件之后，还要关闭文件。以上就是磁盘的抽象，至于底层的实现细节，如数据的记录格式、电机的当前状态等，对程序员来说是透明的，是无须了解的。

负责将硬件细节与程序员隔离开来，并提供一个简单、方便的文件访问方式的程序，当然就是操作系统。除了磁盘硬件之外，它还隐藏了许多其他的低层特性，如中断、时钟、存储管理等。对于每一种硬件，操作系统都提供了一个简单、好用的抽象接口。

从这个角度来看，操作系统的功能就是为用户提供一台等价的扩展计算机，或称虚拟机（virtual machine），它比底层硬件更容易编程。但操作系统是如何做到这一点的呢？这正是本书要详细讨论的内容。简单地说，就是操作系统提供了各式各样的服务，用户程序可以通过称为系统调用的特殊指令来使用这些服务。在本章的稍后部分，我们将会看到一些常用的系统调用。

1.1.2 操作系统作为资源管理器

上述的虚拟机模型是一种自顶向下的观点。反之，如果按照自底向上的观点，则可以把操作系统视为一个复杂的系统的管理者。现代计算机都包含处理器、存储器、定时器、磁盘、鼠标、网络接口、打印机以及其他的各种设备，从这个角度来看，操作系统的任务就是在相互竞争的程序之间，如何有序地控制这些硬件设备的分配。

举例来说，假设在一台计算机上，同时有三个程序在运行，它们都试图在一台打印机上打印各自的计算结果。在这种情形下，最终打印出来的结果可能是这样的：头几行的内容来自于程序1，接下来的几行来自于程序2，再接下来的几行又来自于程序3，等等，最终是一片混乱。而操作系统能够避免这种混乱，使打印变得有序。它的基本思路是先把各个程序的输出结果保存到磁盘上的缓冲区中。当一个程序结束时，再把该程序的输出内容从磁盘文件送到打印机，从而完成打印。与此同时，其他程序可以继续产生新的输出结果，它们不知道这些输出其实并没有立即送往打印机，而是暂存在缓冲区中。

当一台计算机（或网络）有多个用户时，由于用户之间可能会相互干扰，因此必须更好地管理和保护存储器、I/O设备和其他各种资源。此外，在不同的用户之间，不仅需要共享硬件设备，有时还需要共享信息（文件、数据库等）。总之，从资源管理器的角度来说，操作系统的主要任务是跟踪资源的使用状况、满足资源请求、提高资源利用率，以及协调不同程序和用户对资源的访问冲突。

资源管理主要包括两种形式的资源共享：时间上的资源共享和空间上的资源共享。所谓时间上的资源共享，指的是各个程序或用户轮流使用该资源，开始是一个程序在使用，然后是另一个程序在使用，等等。例如，假设在系统中只有一个CPU，而且有多个程序想运行。在这种情形下，操作系统会把CPU分配给其中的一个程序，然后等它运行了足够长的时间后，再换上第二个程序去运行，然后是第三个程序。当每个程序都轮流运行了一遍以后，又把CPU分配给第一个程序，让它再次运行。操作系统的任务就是决定如何来分享该资源：下一个轮到谁？它可以运行多长时间？另外一个例子是打印机的共享。当有多个打印任务在队列中等待打印时，操作系统必须做出决策，选择其中的哪一个去执行。

所谓空间上的资源共享，指的是每个程序不是轮流去使用资源，而是把资源划分为若干份，然后分配给各个程序。例如，将整个内存空间划分为若干块，分别用来存放各个并发运行的程序，使每个程序都能驻留在内存中（这样才能去轮流使用CPU）。假定内存空间足够大，能容纳多个程序，那么这种多个程序同时分享内存的方式显然比单个程序独占内存的方式更有效。当然，这种做法会带来一系列的问题，如公平性、内存保护等，这些问题是由操作系统来解决的。另外一个例子是硬盘。在许多系统中，只有一块硬盘，它同时存放了不同用户的许多文件。对于操作系统的资源管理模块而言，它的任务通常包括分配磁盘空间、跟踪记录磁盘块的使用情况等。

1.2 操作系统的发展历史

操作系统经历了漫长的发展过程。在以下的几个小节中，我们将对此进行简要的回顾。由于在历史上操作系统与计算机体系结构存在非常紧密的联系，因此我们将按照计算机的升级换代历程来讲述它们的操作系统的发展变化。

第一台真正的数字计算机是英国数学家Charles Babbage（1792–1871）设计的。Babbage投入了毕生的精力和大部分财产去建造他的“分析机”，但却未能让它成功地运行起来。因为这台机器是纯机械式的，而当时的技术条件不可能生产出符合精度要求的轮子、齿轮和轮齿等零部件。不用说，这种分析机是没有操作系统的。

有趣的是，Babbage认识到他的分析机可能需要软件，于是雇用了一位年轻的女子——英国著名诗人拜伦的女儿Ada Lovelace为他工作。Ada由此成为世界上第一位程序员。Ada程序设计语言就是用她的名字命名的。

1.2.1 第一代计算机 (1945–1955): 真空管和插接板

从 Babbage 之后一直到第二次世界大战，数字计算机几乎没有什么进展。在 20 世纪 40 年代中期，哈佛大学的 Howard Aiken、普林斯顿高等研究院的 John von Neumann（冯·诺依曼）、宾夕法尼亚大学的 J. Presper Eckert 和 John Mauchley、德国的 Konrad Zuse 等人先后都成功地建造出了计算机。最早的机器使用的是机械继电器，速度非常慢，周期时间的计量单位是秒。后来这些继电器被真空管所取代。那时候的机器非常庞大，往往使用数以万计的真空管，占据了整个房间，然而其运算速度却比现在最便宜的个人计算机还要慢几百万倍。

在计算机出现的早期，每台机器都由一个专门的小组来设计、制造、编程、操作和维护。编程全部采用机器语言，通过在一些插接板上的硬连线来控制其基本功能。当时没有编程语言（甚至连汇编语言都没有），操作系统更是闻所未闻。机器的使用方式是程序员提前在墙上的机时表上预约一段时间，然后到机房将他的插接板插到计算机里，在接下来的几小时里计算自己的题目，并祈祷机器中的两万多个真空管不要发生故障，能够把程序算完。这个阶段的上机题目基本上都是数值计算问题，例如计算正弦和余弦函数表。

到 20 世纪 50 年代早期，出现了穿孔卡片，这时不再使用插接板，而是将程序写在卡片上，然后读入计算机，但其他过程则依然如故。

1.2.2 第二代计算机 (1955–1965): 晶体管和批处理系统

20 世纪 50 年代发明的晶体管极大地改变了计算机的状况。这时的计算机已经比较可靠，厂商可以成批地进行生产并销售给客户，而客户则期望能用它们来完成一些有用的工作。这时也是第一次将设计人员、生产人员、操作员、程序员和维护人员分开。

在这个时期，计算机（现在称之为 **主机**，mainframe）被安装在空调房间里，由训练有素的职业操作员来运行。由于其价格昂贵，每台要几百万美元，因此仅有少数大公司、主要的政府部门和大学才买得起。运行一个作业（job，即一个或一组程序）时，程序员首先将程序写在纸上（用 FORTRAN 或汇编语言），然后用穿孔机制成卡片，并将这些卡片交给操作员。然后程序员就可以去放松一下，喝点咖啡，等待计算结果出来。

当计算机完成了当前的作业后，操作员会从打印机上撕下这次的输出结果并送到输出室，这样程序员就可以从那里拿到自己的计算结果，然后，操作员再从卡片上读入另一个作业。如果作业运行时需要用到 FORTRAN 编译器，操作员还要把它从文档柜中取出来，并读入计算机。当操作员在机房里走来走去忙活这些事情的时候，大量的机时就被浪费掉了。

由于当时计算机非常昂贵，所以很快人们就开始想办法来减少机时的浪费。通常的解决方案是**批处理系统**（batch system）。它的基本思路是：在作业输入室收集满满一盘子的作业，然后用一台比较小、比较便宜的计算机（如 IBM 1401，它比较适合于读卡片、复制磁带和打印输出，但不适合于做数值运算）将它们读到磁带上，然后再用一台比较昂贵的计算机（如 IBM 7094）来完成真正的计算，如图 1.2 所示。

在收集到一批作业之后，输入磁带被送到机房里并装到磁带机上。然后操作员会装入一个特殊的程序（现代操作系统的前身），它将把磁带上的第一个作业读入并运行，输出结果被写入到另一盘磁带上，而不是打印出来。每当一个作业结束后，操作系统就会自动读入下一个作业并运行。当这一批作业全部结束后，操作员会取下输入和输出磁带，把输入磁带换成下一批作业，然后把输出磁带拿到一台 1401 机器上进行脱机打印（即打印机未与主机相连）。

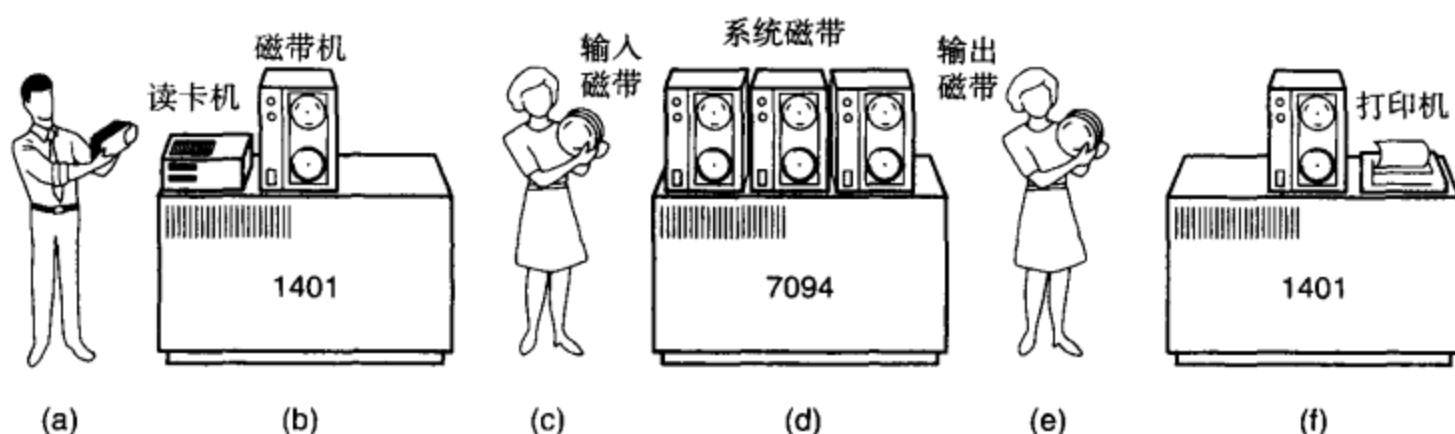


图 1.2 早期的批处理系统: (a)程序员将卡片拿到 1401 处; (b)1401 将一批作业读到磁带上; (c)操作员将输入磁带送至 7094 处; (d)7094 进行计算; (e)操作员将输出磁带送至 1401 处; (f)1401 打印输出结果

典型的输入作业结构如图 1.3 所示。它由一张 \$JOB 卡片开始, 该卡片标识出所需的最大运行时间(分钟)、收费账号以及程序员的名字。随后是一张 \$FORTRAN 卡片, 用于通知操作系统从系统磁带上装入 FORTRAN 语言编译器。在此之后是待编译的源程序, 然后是 \$LOAD 卡片, 用于通知操作系统装入刚编译好的目标程序(编译好的目标程序通常写在暂存磁带上, 需要主动去把它装入)。接下来是 \$RUN 卡片, 告诉操作系统使用随后的数据来运行该程序。最后, \$END 卡片标识作业的结束。这些原始的控制卡片是现代作业控制语言和命令解释器的先驱。

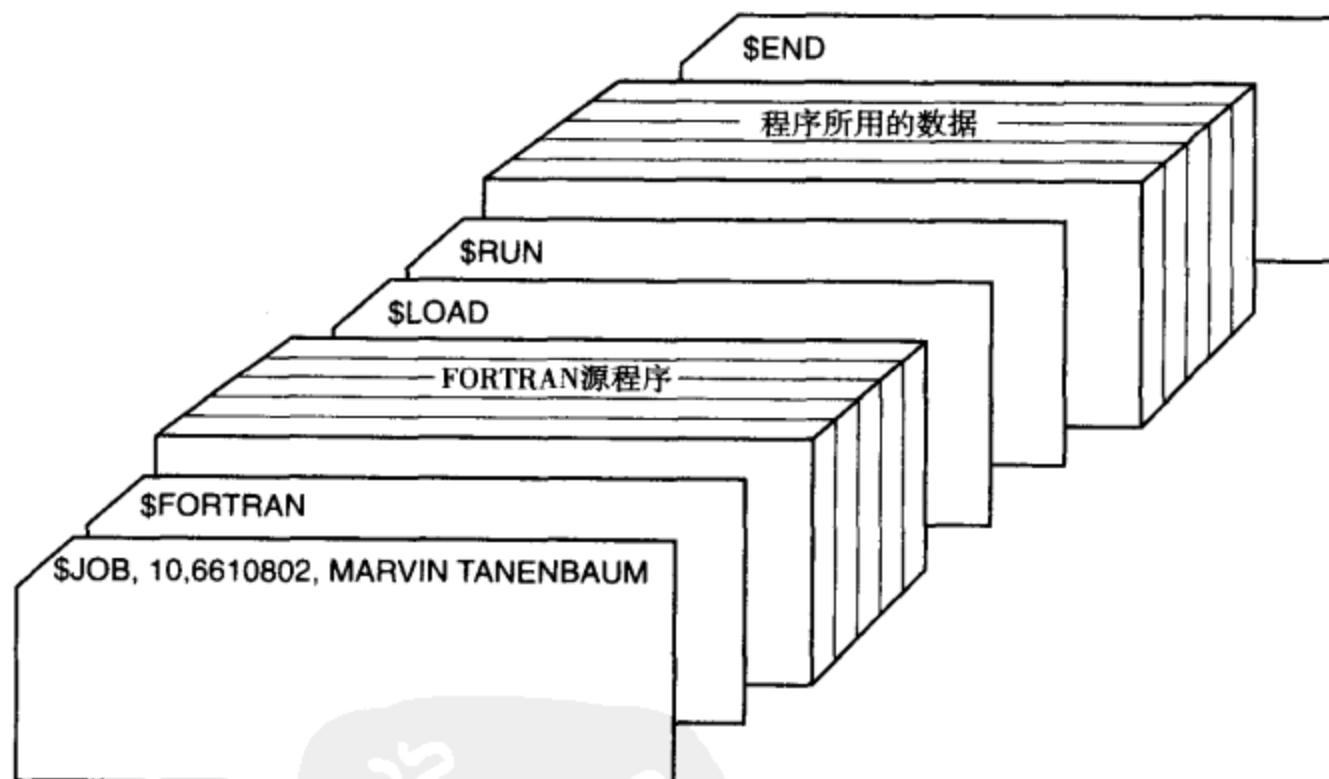


图 1.3 典型 FMS 作业的结构

第二代计算机主要用于科学和工程计算, 例如在物理和工程中经常出现的偏微分方程的求解。这些问题大多用 FORTRAN 语言和汇编语言来编程。典型的操作系统有 FMS (FORTRAN Monitor System) 和 IBSYS (IBM 为 7094 机配备的操作系统)。

1.2.3 第三代计算机 (1965–1980): 集成电路和多道程序

在 20 世纪 60 年代初期, 多数计算机厂商都有两条完全不同且互不兼容的生产线: 一条是面向字的大规模计算机, 如 IBM 7094, 主要用于科学和工程中的数值计算; 另一条是面向字符的商用计算机, 如 IBM 1401, 主要在银行和保险公司中用于磁带的归档和打印。

对于计算机厂商来说，开发和维护两条完全不同的产品线是很昂贵的。此外，许多新的计算机用户在开始时只需要一台小的计算机，后来随着应用的增多，可能又需要一台较大的计算机，并且要求它与原来的机器具有相同的体系结构，这样，原有的程序就都能够运行了。

为了解决这些问题，IBM公司推出了System/360。360是一个软件兼容的计算机系列，在该系列中，既有与1401相当的低档机，又有比7094功能更强的高档机。这些计算机只在价格和性能（最大存储器容量、处理器速度、I/O设备数量等）上有差异。所有的计算机都有相同的体系结构和指令集，因此在一台机器上编写的程序可以运行在所有的计算机上（起码在理论上可行）。此外，360被设计成既可用于科学计算，又可用于商业计算。这样，一个系列的计算机便可以满足所有用户的需要。在随后的几年里，IBM陆续推出了360的后续机型，如370, 4300, 3080, 3090和Z系列。

360是第一种采用（小规模）集成电路的主流机型。与采用晶体管制造的第二代计算机相比，其性能价格比有了很大提高。360很快就获得了成功，其他计算机厂商也很快采纳了这种兼容机系列的思想。时至今日，这些计算机仍在世界各地的一些计算中心使用，主要用来管理巨型数据库（如航空订票系统），或者是作为访问流量较大的WWW服务器，以便每秒钟可处理几千次的访问请求。

“单一系列”思想的最大优点同时也是它的最大缺点。它要求所有的软件，包括操作系统**OS/360**，都要能够在所有的机器上运行——从比较小的计算机（通常用来代替1401，进行读取卡片、复制磁盘等工作）到非常大的计算机（通常用来代替7094，进行天气预报和其他繁重的计算工作）；从只能带很少外设的机器到能带很多外设的机器；从商业应用到科学计算，等等。总之，这个系列的计算机要能够适用于所有不同的应用领域。

在这种情形下，要想编写出一个软件来同时满足所有这些相互冲突的需求，对任何人来说，均是非常困难的。其结果就是一个庞大且极其复杂的操作系统，其规模比FMS大2~3个数量级。它里面包含有数千名程序员编写的数百万行的汇编语言代码，同时也包含有成千上万处错误。这就导致IBM不断地发行新版本来更正这些错误，而新版本在改正原有错误的同时又引入了新的错误，所以错误的总数基本保持不变。

OS/360的设计者之一Fred Brooks后来写过一本诙谐幽默而又不乏真知灼见的书（Brooks, 1995），描述了他在开发OS/360过程中的经验。由于篇幅原因，这里不可能复述该书的内容，但只要看一下该书的封面，就可知一二。它的封面画的是一群史前动物陷入一个泥潭而不能自拔的情形。2004年出版的Silberschatz和Galvin的著作，也用相同的方法表达了类似的观点，他们都把操作系统比喻为恐龙。

虽然存在着上述的种种问题，第三代操作系统（包括OS/360和其他公司的类似产品）的确很好地满足了大多数用户的要求。同时，它们也使第二代操作系统中缺乏的几项关键技术得到了广泛应用。其中最重要的就是**多道程序**（multiprogramming）。在7094机上，若当前作业因等待磁带或其他I/O操作而暂停时，CPU就只能处于空闲状态，直至该I/O结束。对于CPU繁忙的科学计算问题，I/O操作比较少，因此浪费的时间也很少。而对于商业数据处理，I/O操作的等待时间通常占到80%~90%，因此必须采取某种措施来减少CPU时间的浪费。

解决的办法就是将内存划分为几个分区，每个分区存放不同的作业，如图1.4所示。当一个作业正在等待I/O操作完成时，另一个作业就可以去使用CPU。按照这个思路，如果在内存中存放足够的作业，那么CPU的利用率就可以接近100%。但这又带来了一个新问题，即如何保护内存中的各个作业，使它们不会相互妨碍和攻击。这需要用到特殊的硬件机制，幸运的是，360和其他第三代计算机都配有此类硬件。

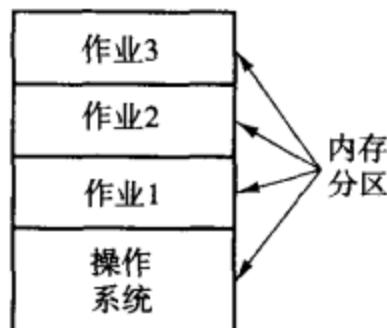


图 1.4 内存中有三个作业的多道程序系统

第三代计算机的另一个新特性是，当一张卡片被拿到机房后，能够很快把其中的作业读入磁盘。这样，当一个作业运行结束后，操作系统就能将一个新作业从磁盘中读出，并装入刚刚空出来的内存分区去运行。这种技术称为假脱机技术（Simultaneous Peripheral Operation On Line, SPOOLING），它同样可以用于输出。在使用了假脱机技术后，就不再需要 IBM 1401 了，也不需要再将磁带搬来搬去。

第三代操作系统相当适合于大型的科学计算和大规模的商务数据处理，但从本质上说，它们仍旧是批处理系统。许多程序员很怀念第一代计算机的使用方式，那时他们可以独占一台机器几个小时，可以即时地调试自己的程序。而对于第三代计算机系统，一个作业从提交到取回计算结果往往需要几个小时的时间。因此，当程序员在编写程序的时候，如果敲错了一个逗号，从而导致编译失败，那么就要等待半天的时间，才能去改正这个错误。

程序员们都希望能有快速的响应时间，这种需求导致了分时系统（timesharing）的出现。它实际上是多道程序的一个变体，不同之处在于每个用户都有一个联机的终端。在分时系统中，假设有 20 个用户登录在线，其中有 17 个用户在思考问题、聊天或喝咖啡，则 CPU 可轮流分配给剩下那 3 个需要得到服务的作业。当人们在调试程序时，通常只会发出简短的命令（如编译一个 5 页的过程^①），而很少执行费时的长命令（例如，将一个上百万条记录的文件进行排序）。因此计算机能够为许多用户提供快速的交互式服务，与此同时，当 CPU 偶尔空闲时还能在后台运行一些比较大的批处理作业。第一个真正意义上的分时系统 CTSS（Compatible Time Sharing System）是由 MIT 的科研人员在一台改装过的 7094 机上开发成功的（Corbató et al., 1962），但直到第三代计算机广泛采用了必需的硬件保护机制后，分时系统才逐渐流行开来。

在 CTSS 系统研制成功之后，MIT、贝尔实验室和通用电气公司（在当时是一家主要的计算机厂商）决定开发一种“公用计算服务系统”，即一台能同时支持数百个分时用户的计算机。这个想法来源于电力供应系统：当人们需要电能时，只要将电气设备连接到墙上的插座即可。这个系统称为 MULTICS（MULTiplexed Information and Computing Service），它的设计者雄心勃勃，试图用一台巨大的机器来满足整个波士顿地区所有用户的计算需求。在当时看来，这种想法完全是一种科学幻想，谁能想到时隔 30 年后只花几千美元就能买到一台计算能力远远超过他们的 GE-645 的个人计算机呢？这就好像我们现在设想去构造一条横跨大西洋的超音速海底铁路隧道一样！

MULTICS 的成功是比较复杂的。在一台计算能力仅比 Intel 80386 台式机稍微强一点的机器上，它能同时支持数百个用户。不过说实在的，这并不像听起来那样了不起。因为当时的人们知道如何去编写小的、高效的程序，而现在人们似乎已经失去这种技能了，这也许就是科技进步所必须付出的代价。MULTICS 未能统治全世界的原因有多个，其中一个原因是它采用了 PL/I 编程语言。PL/I 编译器的发布比预计晚了几年，而且当它最终推出时，几乎无法正常工作。另外，以当时的技术条件来说，MULTICS 制定的目标有点太高了，就像 19 世纪 Charles Babbage 的分析机那样。

^① 在本书中，术语“过程”、“子程序”和“函数”是等价的。

MULTICS引入了计算机领域的许多概念的雏形，但其研制难度却超出了所有人的预料。在开发过程中，贝尔实验室退出了该项目，通用电气公司（GE）也退出了计算机领域。然而，MIT坚持了下来，并最终让MULTICS成功地运转起来。Honeywell公司在收购了GE的计算机业务后，把MULTICS作为一个商业产品进行销售，它被安装在全球80多家大公司和大学中。MULTICS的用户数目不是很多，但都非常忠诚。例如，通用汽车公司、福特公司、美国国家安全局等，它们都是在20世纪90年代末期才关闭了自己的MULTICS系统。最忠实的用户当属加拿大国防部，它在2000年10月才关闭了自己的MULTICS系统。尽管在商业上不是很成功，但MULTICS对随后的操作系统有着巨大的影响，且关于它的信息相当丰富（Corbató et al., 1972; Corbató and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; Saltzer, 1974）。它还有一个Web站点 www.multicians.org，至今依然活跃，上面登载了该系统的详细信息，如它的设计者和用户等。

“计算服务”这个术语早已不再使用，但它的思想近年来又重获新生。一个最简单的例子是，在一家公司或一间教室内，把所有的PC机和工作站（一种高端PC机）通过局域网（Local Area Network, LAN）连接到一台文件服务器上，在这台服务器上存放了所有的程序和数据。这样，系统管理员只需要安装和维护一组程序和数据即可。当一台PC机或工作站发生故障时，可以很方便地重新安装本地软件，而不用担心本地数据的恢复和保存问题。在一些异构环境下，可以使用一种中间件软件，来帮助本地用户去访问远程服务器上的文件、程序和数据库。有了中间件以后，对于个人PC机和服务器来说，远程的联网计算机就像在本地一样，而且众多不同类型的服务器、PC机和工作站之间的差异性，也被中间件屏蔽掉了，它提供了一个统一的用户界面。万维网就是一个例子。Web浏览器在显示文档的时候，采用的是一种统一的风格。例如，对于用户在浏览器中看到的一篇文档，它的文本可能来自于某个服务器，它的图片可能来自于另一个服务器，而它的表单样式可能来自于第三个服务器。并在企业和大学中，经常使用一个Web界面去访问另一座大楼甚至另一个城市中的某台计算机，在上面访问数据库或运行程序。对于分布式系统而言，中间件就好比是它的操作系统，但实际上，它并不是一个真正的操作系统。不过这部分内容不属于本书的范畴。关于分布式的详细内容，请参阅 Tanenbaum and Van Steen (2002)。

第三代计算机的另一个主要成就是小型机的崛起，以1961年DEC公司的PDP-1为起点。PDP-1计算机的内存只有4K个字，每个字18比特，售价12万美元（不到IBM 7094的5%），在当时这种机型非常热销。对于某些非数值的计算，它几乎和7094一样快。PDP-1开辟了一个全新的产业，很快就有了一系列的机型（但与IBM系列机不同，它们互不兼容），其顶峰为PDP-11。

贝尔实验室一位曾参加过MULTICS研制的计算机科学家Ken Thompson，在一台无人使用的PDP-7机器上开发了一个简化的单用户版MULTICS，他的工作导致了后来的UNIX操作系统的诞生。很快，UNIX就在学术界、政府部门和商业公司流行开来。

关于UNIX操作系统的具体历史有专门的著作（如Salus, 1994），这里不再详述。由于UNIX的源代码公开，许多组织都开发了他们自己的UNIX版本，这些版本互不兼容，造成了很大的混乱。其中有两个主要的版本，一个是AT&T的**System V**，另一个是加州大学伯克利分校的**BSD**（Berkeley Software Distribution）。另外，它们都有一些小的变体，如FreeBSD，OpenBSD和NetBSD。为了使相同的程序能在所有的UNIX系统上运行，IEEE制定了一个标准，称为**POSIX**，大多数UNIX版本现在都支持该标准。POSIX定义了一组最小的系统调用接口，所有兼容的UNIX版本都必须支持这组函数接口。实际上，一些其他类型的操作系统现在也支持POSIX接口，如Windows。关于如何来编写POSIX兼容的软件，请参阅相关书籍（IEEE, 1990; Lewine, 1991）和在线文档，如www.unix.org上的文章“Single UNIX Specification”。在本章的后面，当我们谈到UNIX时，指的是POSIX

兼容的所有 UNIX 版本。对于这些系统，虽然它们的内部实现机理是不同的，但它们都支持 POSIX 标准，因而对于程序员来说，它们都是类似的。

1.2.4 第四代计算机（1980-）：个人计算机

大规模集成电路的发展导致在每平方厘米的硅片上可以集成数千个晶体管，于是基于微处理器的个人计算机时代就来临了。从体系结构上看，个人计算机（最开始称为微机）与 PDP-11 并无二致，但价格上却相去甚远。打个比方来说，对于公司的一个部门或大学的一个院系，可能具备实力去配备一台小型机。但微机的出现，却使每个人都有可能拥有自己的计算机。

在微机发展的初期，曾经有不同系列的机器。Intel 在 1974 年推出了 8080，这是第一个通用的 8 位微处理器。许多公司生产的整机都使用了 8080（或兼容的 Zilog Z80）和 CP/M（Control Program for Microcomputers）操作系统。CP/M 是 Digital Research 公司的产品，也是当时的主流系统，人们在它上面开发了很多应用程序。在长达 5 年的时间内，它一直统治着个人计算机的操作系统市场。

摩托罗拉（Motorola）也推出了一款 8 位的微处理器 6800。公司的一群工程师，建议对 6800 进行改进，但遭到了高层的拒绝，于是他们离开了摩托罗拉，创立了一家新公司 MOS Technology，并制造出 6502 CPU。6502 被用在一些早期的 PC 机中，如 Apple II。在家庭和教育市场，Apple II 成为 CP/M 系统的有力竞争者。但 CP/M 是如此地成功，以至于许多 Apple II 用户购买了 Z-80 协处理器附加卡，来运行 CP/M 操作系统（6502 本身与 CP/M 不兼容）。CP/M 卡的销售商是一家小公司，叫做微软（Microsoft），它的另一个业务领域是 BASIC 解释器，这种解释器运行在许多 CP/M 微机上。

下一代微处理器是 16 位系统。Intel 在 20 世纪 80 年代初推出了 8086，IBM 使用 Intel 的 8088（以 8086 为核心，外接 8 位数据线）来构造它的 IBM PC 机。微软为 IBM PC 机配备了一个软件包，包括微软的 BASIC 和 DOS 操作系统。DOS（Disk Operating System）最早是由另一家公司开发的，后来微软把它买了下来，并雇用原作者来对它进行改进。改进后的系统被命名为 MS-DOS（MicroSoft Disk Operating System），它很快就统治了 IBM PC 机市场。

CP/M、MS-DOS 和 Apple DOS 都是命令行系统，也就是说，用户通过在键盘上键入命令，来控制系统的运行。数年前，斯坦福研究院的 Doug Engelbart 发明了图形用户界面 GUI（Graphical User Interface），包括一些窗口、图标、菜单和鼠标。苹果公司的 Steve Jobs 看到了其中的机会，认为可以用它来实现真正用户友好的 PC 机。也就是说，这种 PC 机适合于对计算机一无所知且不想学习的用户。1984 年初，苹果公司推出了 Macintosh 机，它使用了摩托罗拉公司的 16 位 CPU 68000 和 64 KB 的只读内存（Read Only Memory, ROM）来支持 GUI。随着时间的推移，Macintosh 机也在不断地发展，后继的摩托罗拉 CPU 都是真正的 32 位系统。不过后来苹果公司还是更换了 CPU，使用了 IBM 的 PowerPC CPU，这种 CPU 具有 RISC 32 位（后来是 64 位）体系结构。2001 年，苹果公司对操作系统进行了一次较大变动，推出了 Mac OS X，该系统在 Berkeley UNIX 的基础上，构造了一个全新的 Macintosh GUI 版本。2005 年，苹果公司又宣布，它将转而使用 Intel 的处理器。

为了与 Macintosh 竞争，微软公司开发了 Windows 操作系统。最早的 Windows 只是运行在 16 位 MS-DOS 上的一个图形环境，也就是说，它并非一个真正的操作系统，而更像是一个漂亮的外壳。当前的 Windows 版本都是从 Windows NT 演变过来的，NT 是一个全 32 位操作系统，它在某些层次上与以前的版本兼容，但其内核完全经过了重写。

另一种主要的操作系统是 UNIX，包括由它派生出来的其他系统。UNIX 主要用于工作站和其他高档计算机（如网络服务器），尤其是采用了高性能 RISC 芯片的计算机。在 Pentium 计算机上，许多学生和企业用户逐渐倾向于用 Linux 来取代 Windows 系统（在本书中，将使用术语“奔腾”来表示整个 Pentium 系列产品，包括低端的 Celeron、高端的 Xeon 以及兼容的 AMD 微处理器）。

虽然许多UNIX用户，尤其是一些有经验的程序员，更喜欢命令行式的交互界面，但几乎所有的UNIX系统都支持一种叫做**X Window**的窗口系统。这套系统是由MIT开发的，它支持基本的窗口管理，用户可以使用鼠标来创建、删除和移动窗口，或者调整窗口的大小。对于一个完整的GUI，如**Motif**，它可以运行在X Window系统之上，给UNIX用户提供一个类似于Macintosh或Microsoft Windows的图形用户界面。

20世纪80年代中期开始出现了一种有趣的发展趋势，即运行**网络操作系统**(network operating system)和**分布式操作系统**(distributed operating system)(Tanenbaum and Van Steen, 2002)的个人计算机网络的崛起。在网络操作系统中，用户知道其他计算机的存在。他可以登录到一台远程计算机上，并将文件从一台机器复制到另一台机器。每台计算机都运行它自己的本地操作系统，而且有它自己的本地用户。一般来说，各台机器之间是相互独立的。

网络操作系统与单处理器的操作系统本质上没有区别。它们需要一个网络接口控制器以及相应的底层驱动软件，此外还需要一些程序来进行远程登录和远程文件访问，但这些额外的事物并未改变操作系统的本质结构。

与之相反，在用户看来，分布式操作系统就像普通的单处理器系统，尽管它实际上由多个处理器组成。用户不会感知到他们的程序是在哪个处理器上运行、他们的文件被存放在哪里，所有这些均由操作系统自行高效地完成。

真正的分布式操作系统并不仅仅是在单处理器系统的基础上增添一小段代码。事实上，分布式系统与集中式系统有本质的区别。例如，在分布式系统中，通常允许一个应用程序在多台处理器上同时运行，因此需要更复杂的处理器调度算法来获得最好的并行性能。

另外，网络中的通信延迟往往会导致不完整、过时甚至错误的信息，而分布式算法必须在这种环境下运行。这和单处理器系统是完全不同的，在单处理器系统中，操作系统能够掌握整个系统的所有信息。

1.2.5 MINIX 3的历史

在UNIX发展的早期(Version 6)，在AT&T的许可下，人们可以方便地拿到它的源代码并加以研究。澳大利亚新南威尔士大学的John Lions甚至专门写了一本小册子来逐行地解释UNIX源代码(Lions, 1996)。这本小册子已被许多大学的操作系统课程用做教材(需经AT&T的同意)。

当AT&T发布UNIX Version 7时，开始认识到了它的商业价值。因此在Version 7的许可证中，禁止人们在课程中研究UNIX源代码，以免其商业利益受到损害。许多大学为了遵守该规定，就在课程中略去UNIX的内容而只讲操作系统理论。

但如果只讲理论，会使学生对实际的操作系统产生一种片面的认识。例如，在书本中作为重点来讲述的内容，如进程调度算法，在实际中其实并没有那么重要。而在实际系统中非常重要的内容，如I/O系统和文件系统，又因为缺乏理论性而被忽略。

为了扭转这种局面，本书的作者之一(Tanenbaum)决定从头开始编写一个全新的操作系统。从用户使用的角度来看，该系统与UNIX完全兼容；但是从内部实现来看，它与UNIX是完全不同的，它没有借用AT&T的任何一行代码，这样就不受其许可证的限制，可以被班级和个人用来学习。就像医学院的学生解剖青蛙一样，读者可以通过剖析一个真实的操作系统，来研究其内部的运作机理。这个系统称为**MINIX**，它于1987年正式发布。它带有完整的源代码，任何人都可以去研究和修改。它的名字代表“mini-UNIX”的意思，因为它非常简洁，一般程度的读者都能读懂它。

除了合法性之外，MINIX与UNIX相比还有另一个优点：它比UNIX晚出现十年，因此其代码采用了一种更加模块化的组织结构。例如，从最早的版本开始，MINIX的文件系统和存储管理器就

不是操作系统的一部分，而是作为一个用户程序运行。在当前的版本（MINIX 3）中，这种模块化的思想扩展到了设备驱动程序，除了时钟驱动程序之外，所有的驱动程序都作为用户程序来运行。另一个不同之处在于 UNIX 的设计侧重于效率，而 MINIX 则侧重于可读性（数百页的程序通常被认为是可读的）。在 MINIX 的源代码中，有数千行的注释。

MINIX 最初被设计成与 UNIX Version 7（V7）兼容。V7 比较简单、优雅，所以我们使用它来作为目标原型。V7 在业界有很好的声誉，有人甚至认为它不仅超过了之前的所有版本，而且超过了之后的所有版本。随着 POSIX 的出现，MINIX 在保持向后兼容的同时，开始向 POSIX 标准靠拢。这种渐进的思想在计算机界相当普遍，没有一家厂商希望在引进新系统的同时使客户原先的所有程序作废。本书介绍的 MINIX 版本——MINIX 3 基于 POSIX 标准。

和 UNIX 一样，MINIX 也是用 C 语言编写的，因此很容易移植到其他机器上。MINIX 的最初版本主要在 IBM PC 机上实现，后来被移植到其他一些硬件平台上。MINIX 一直恪守“Small is Beautiful”的原则，因此最早的版本甚至不需要硬盘就能运行（在 MINIX 刚刚出现的 20 世纪 80 年代中期，硬盘还是一种昂贵的新鲜事物）。随着功能和规模的增长，MINIX 也发展到需要一个硬盘才能运行，但只要求一个大小为 200 MB 的分区即可（对于嵌入式应用，仍然不需要硬盘）。与之相比，即使是一些比较小的 Linux 系统，也需要 500 MB 的磁盘空间，而为了安装一些通用的应用软件，还需要几个 GB 的空间。

对于多数 IBM PC 用户来说，运行 MINIX 和运行 UNIX 是非常类似的。所有的基本程序，如 *cat*, *grep*, *ls*, *make* 和 *shell* 程序，都与 UNIX 中的对应程序有相同的功能。与操作系统本身一样，这些实用程序都是由作者、他的学生以及其他志愿人员从头重写的，没有使用任何 AT&T 的代码。

对于 MINIX 的最初版本，我们又继续开发了十年左右，在 1997 年发布了 MINIX 2，同时推出了本书的第二版。MINIX 的版本 1 和版本 2 之间的差别还是比较大的，例如，从 8088、16 位、实模式和软盘升级到了 386、32 位、保护模式和硬盘。

此后，开发过程继续进展下去，虽然有点慢，但很有系统性。到了 2004 年，Tanenbaum 认为整个系统已经变得过于庞大且不可靠，因此决定继续从事以前的 MINIX 方面的工作。他与阿姆斯特丹 Vrije 大学的学生和程序员一起，推出了 MINIX 3。系统采用了新的设计，内核也进行了重新构造，缩减了系统的规模，强化了模块化和可靠性。新版本既可用于 PC 机，也可用于嵌入式系统。尤其是对嵌入式应用来说，简洁、模块化和可靠性是非常关键的。开发小组内的一些人提议给系统换一个全新的名字，但最终我们还是决定把它称为 MINIX 3，因为 MINIX 这个名字已经广为人知。事实上，这也是一种通行的做法。例如，当苹果公司决定放弃它自己的操作系统 Mac OS 9，并且用 Berkeley UNIX 的一个变体来取而代之时，使用的名称是 Mac OS X，而不是 APPLIX 或类似的新名字。再比如，Windows 系列操作系统一直在不断向前发展，但 Windows 这个名字却始终保持不变。

与 Windows, Linux, FreeBSD 和其他操作系统相比，MINIX 3 内核只有不到 4000 行的可执行代码，而其他系统一般都有几百万行。我们认为保持一个精简的内核是非常重要的，因为内核错误的破坏性要远远大于用户程序中错误的破坏性，而内核代码越多，其中的错误也就越多。研究表明，每 1000 行可执行代码中能够检测到的错误数量为 6~16 个（Basili and Perricone, 1984）。而且这还只是检测出来的错误数量，而实际的错误数量可能更多。另一项研究（Ostrand et al., 2004）表明，一个软件即使已经发布了十多个版本，但平均仍有 6% 的文件中包含有错误。而且在到达某个点之后，错误率就会趋于稳定，而不是逐渐逼近于零。有人曾经用一个非常简单的自动模型检查器来测试 Linux 和 OpenBSD 的稳定内核版本，结果发现了几百个内核错误，这些内核错误主要出现在设备驱动程序中（Chou et al., 2001; Engler et al., 2001）。这就是为什么在 MINIX 3 中，我们把设备驱动程序移出内核的原因。如果设备驱动程序在用户态下运行，那么对系统造成的破坏就会比较小。

本书自始至终以 MINIX 3 作为例子。不过，对于 MINIX 3 系统调用的大部分内容（代码本身除外），同样适用于其他的 UNIX 系统。在阅读本书时，读者应记住这一点。

有些读者可能对 MINIX 和 Linux 之间的关系比较感兴趣。在 MINIX 发布后不久，便出现了一个相应的 USENET 新闻组 *comp.os.minix*，在数周内便有多达 40 000 个用户订阅。其中多数人都想往 MINIX 中加入一些新的特性以使它更大、更好（嗯，至少是更大吧）。每天都有数百人提出自己的建议、想法甚至是代码。而 MINIX 的作者在几年内始终坚持不采纳这些建议，目的是使 MINIX 保持足够的简洁，以便于学生理解；同时保持规模的小巧，使它能运行在一些低档的、学生可以买得起的计算机上。事实上，对于一些看不上 MS-DOS 的人来说，MINIX 系统及其源代码的存在，甚至是促使他们去购买一台 PC 机的动因。

在这些提建议的人中，有一个芬兰学生 Linus Torvalds。Torvalds 在他的 PC 机上安装了 MINIX 系统，并且认真钻研了它的源代码。后来，Torvalds 想在他自己的 PC 机上阅读 USENET 新闻组（如 *comp.os.minix*），免得每次都得跑到学校去。但是他需要的一些特性在 MINIX 中没有，于是他就写了一个程序来完成这些功能。但不久他又需要一个不同的终端驱动程序，于是他又写了一个相应的程序。接下来，他想要下载和保存新闻组中的文章，于是又写了一个磁盘驱动程序和一个文件系统。到 1991 年 8 月，他已经完成了一个基本的内核。在 1991 年 8 月 25 日，他把这个消息公布在 *comp.os.minix* 上。这个公告吸引了其他人来帮助他，在 1994 年 3 月 13 日，Linux 1.0 正式发布了，这标志着 Linux 的诞生。

Linux 已经成为开放源代码运动的重要标志之一，它正在许多环境中挑战 UNIX 和 Windows 系统。其他一些开源软件，如 Apache Web 服务器和 MySQL 数据库，在商业领域与 Linux 配合默契。Linux, Apache, MySQL 和开源的 Perl 和 PHP 编程语言，经常一起用在 Web 服务器上。有时人们把它们的首字母抽取出来，缩写为 LAMP。关于 Linux 和开源软件的历史，请参阅 DiBona et al. (1999), Moody (2001) 和 Naughton (2000)。

1.3 操作系统概念

操作系统与用户程序之间的接口由操作系统提供的“扩展指令”集来定义。这些扩展指令传统上被称为系统调用（system call），它们的实现方法多种多样。为了真正地理解操作系统的运作机制，我们有必要仔细地研究一下这个接口。当然，对于不同的操作系统来说，它们提供的系统调用也是各不相同的（尽管基本概念都是大致相同的）。

在介绍系统调用的时候，有两种方法：一种是泛泛而笼统地进行介绍（如操作系统有“读文件”的系统调用）；另一种是选择一个特定的系统，然后讲述该系统的系统调用（如 MINIX 3 有一个 `read` 系统调用，它有三个参数：一个指定所操作的文件，一个指定读取数据的存放地址，最后一个指定读多少个字节）。

本书将采用第二种方法，这样可以更细致地观察操作系统的内部操作。1.4 节中将详细介绍 UNIX（包括 BSD 的不同版本）、Linux 和 MINIX 3 中的基本系统调用。为简洁起见，我们只讲述 MINIX 3，但相应的 UNIX 和 Linux 系统调用一般均基于 POSIX 标准。在讲述实际的系统调用之前，我们先来简要地综述一下 MINIX 3，以获得一些大体上的认识，也就是说，一个操作系统到底是什么样的。这个综述的内容同样也适用于 UNIX 和 Linux 系统。

MINIX 3 的系统调用大致可以分为两类：与进程有关的系统调用和与文件系统有关的系统调用。我们将逐一进行介绍。

1.3.1 进程

在 MINIX 3 及所有操作系统中，一个重要的概念就是进程（process）。从本质上来说，一个进程就是一个正在执行的程序。每个进程都有自己的地址空间，也就是一组内存地址，从某个最小值（通常是0）到某个最大值，进程可以读写其中的内容。地址空间中包括可执行程序、程序的数据和它的栈。与每个进程相关的还包括一组寄存器，如程序计数器、栈指针和其他硬件寄存器，以及运行该程序所需的所有其他信息。

进程的概念将在第2章中详细讨论，就目前而言，为了对进程有一种直观的感觉，我们可以考虑一下分时系统的工作原理。每隔一定的周期，操作系统就会暂停当前进程的执行，转而启动另一个进程。这样做的原因是，比如说，在过去1秒钟内，第一个进程已经运行完分配给它的时间片，所以要暂停它的运行。

假设一个进程被暂时挂起，那么后来当它需要重新运行的时候，就要求此刻的状态与先前暂停时的状态完全相同。这就意味着当我们要挂起一个进程时，必须把它的所有信息都要保存在某个地方。例如，假设进程以只读方式打开了若干个文件，每个文件都有一个指针来指示当前的读写位置。当一个进程被暂时挂起时，所有的这些指针都要保存下来，这样，当进程被重新调度执行后，如果它使用 `read` 调用读取数据，就能读到正确的数据。在许多操作系统中，一个进程的所有信息（除了它的地址空间中的内容）均存放在操作系统的一张表中，该表称为进程表（process table），它实际上是一个结构数组（或链表），系统中的每个进程都要占用其中的一项。

因此，对于一个被挂起的进程，它主要包括两部分的内容。一是进程的地址空间，称为内核映像（core image，之所以起这个名字，是为了纪念以前的磁心存储器），二是相应的进程表项，包含寄存器值及其他信息。

在与进程管理有关的系统调用中，最主要的是进程的创建和终止。考虑一个典型的例子：命令解释器（shell）进程，它负责从终端读入命令。假设用户刚刚键入一条命令，要求编译一个程序，那么 shell 必须首先创建一个新进程来运行编译器。当该进程完成编译后，它就会执行一条系统调用来终止自己。

在 Windows 和其他图形用户界面的操作系统中，可以通过双击桌面上的图标来启动一个程序，这等效于在命令行中键入该程序的名字。作为命令解释器，GUI 方式的确方便好用，但由于篇幅有限，我们不打算详述这方面的内容。

一般来说，一个进程能够创建一个或多个其他的进程（称为子进程），而且这些子进程又可以创建它们自己的子进程，这样就得到了一棵进程树，如图 1.5 所示。有时，一组相关的进程需要相互合作，共同完成某项任务，这样它们就需要相互通信以协调各自的进展，这种通信称为进程间通信（interprocess communication），我们将在第 2 章中对此进行详细讨论。

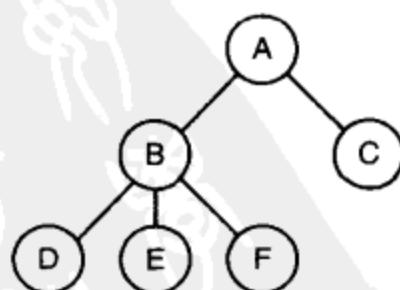


图 1.5 一棵进程树。进程 A 创建了两个子进程 B 和 C。进程 B 创建了三个子进程 D，E 和 F

与进程有关的其他系统调用包括请求更多的内存（或释放不再使用的内存）、等待一个子进程结束、加载并执行另一个程序等。

有时，我们需要向一个正在运行的进程传送信息，而该进程并不是无所事事，专门等待消息的到来。例如，一个进程需要与另一台计算机上的某个进程进行通信，这时它可以通过网络向远程进程发送消息。为了确保消息或消息的应答不会丢失，发送者可以要求它所在的操作系统在指定的若干秒后给它一个通知，这样，如果它尚未收到应答消息，就可以重发消息。在设定了这个定时器后，该程序就可以去做其他的事情。

在经过了指定的时间后，操作系统向这个进程发送一个**警报信号**。该信号将使进程暂停执行当前的程序，将寄存器的值保存在栈中，然后运行一个特殊的信号处理程序，例如重传一条丢失的消息。当信号处理程序结束后，再恢复进程的运行上下文，回到被信号打断的地方继续往下执行。从某种意义上来说，信号有点类似于硬件中断，只不过它是一种纯软件机制。除了定时器超时之外，还有很多其他原因可以导致信号的产生。例如，硬件检测出的许多陷阱（trap），如执行非法指令或访问非法地址，都会被转换成信号，发送给相应的“犯了错误”的进程。

MINIX 3 的每一个合法用户都有一个由系统管理员分配的**用户标识号（UID）**。对于系统中的每一个进程，都记录有启动它的用户的 UID。子进程的 UID 与其父进程的相同。此外，用户可以是某个组的成员，每个组都有一个**组标识号（GID）**。

系统中有一个特殊的用户——**超级用户（superuser）**，他拥有特殊的权力，许多保护规则对他无效。在大型系统中，只有系统管理员知道超级用户的口令。但很多普通用户，尤其是一些学生，总是费尽心机地去寻找系统的安全破绽，以使自己无须口令便可成为超级用户。

我们将在第 2 章中讨论进程、进程间通信等相关问题。

1.3.2 文件

另一大类系统调用与文件系统有关。如前所述，操作系统的一个主要功能就是屏蔽磁盘和其他 I/O 设备的具体细节，给程序员提供一个简洁方便且与设备无关的文件模型。显然，系统需要如下一些系统调用：创建文件、删除文件、读文件和写文件。在读一个文件之前先要打开它；在读完之后还要关闭它，因此需要相应的系统调用来完成此类功能。

为了更好地组织文件，MINIX 3 使用了**目录（directory）**的概念。例如，对于一个学生来说，他可能会为选修的每门课程创建一个目录（用来存放该课程的编程练习），并创建一个目录来存放电子邮件，然后再创建另一个目录来存放他的 WWW 主页。这样就需要用到相应的系统调用来创建和删除目录，以及把一个现有文件放到某个目录中或者从某个目录中删除一个文件。目录项可以是文件或其他目录。这样就得到了一个层次化的模型，也就是文件系统，如图 1.6 所示。

进程和文件都被组织成树状结构，但它们之间的相似性仅此而已。进程树的层次一般都不会很深（很少超过三层），而文件树的层次一般为四层、五层甚至更多。进程之间的层次结构是暂时性的，通常只存在几分钟；而目录层次则可以长期存在，几年都没有问题。在所有者和保护方面，进程和文件也是有区别的。一般来说，只有父进程可以控制和访问子进程，但对于文件和目录来说，除了所有者之外，其他用户也可以对它们进行访问。

目录层次结构中的每一个文件都可以用一个从根目录开始的**路径名**来标识，这种绝对路径名中包含了从根目录到该文件的所有中间目录，相互之间用斜杠“/”隔开。在图 1.6 中，文件 CS101 的路径名是 /Faculty/Prof.Brown/Courses/CS101。起始的斜杠表示这是一个从根目录开始的绝对路径。在 Windows 操作系统中，也有路径名的概念，但它使用反斜杠“\”来作为目录之间的分隔符，因此上述文件的路径名为 \Faculty\Prof.Brown\Courses\CS101。在本书中，我们将使用 UNIX 的路径名规范。

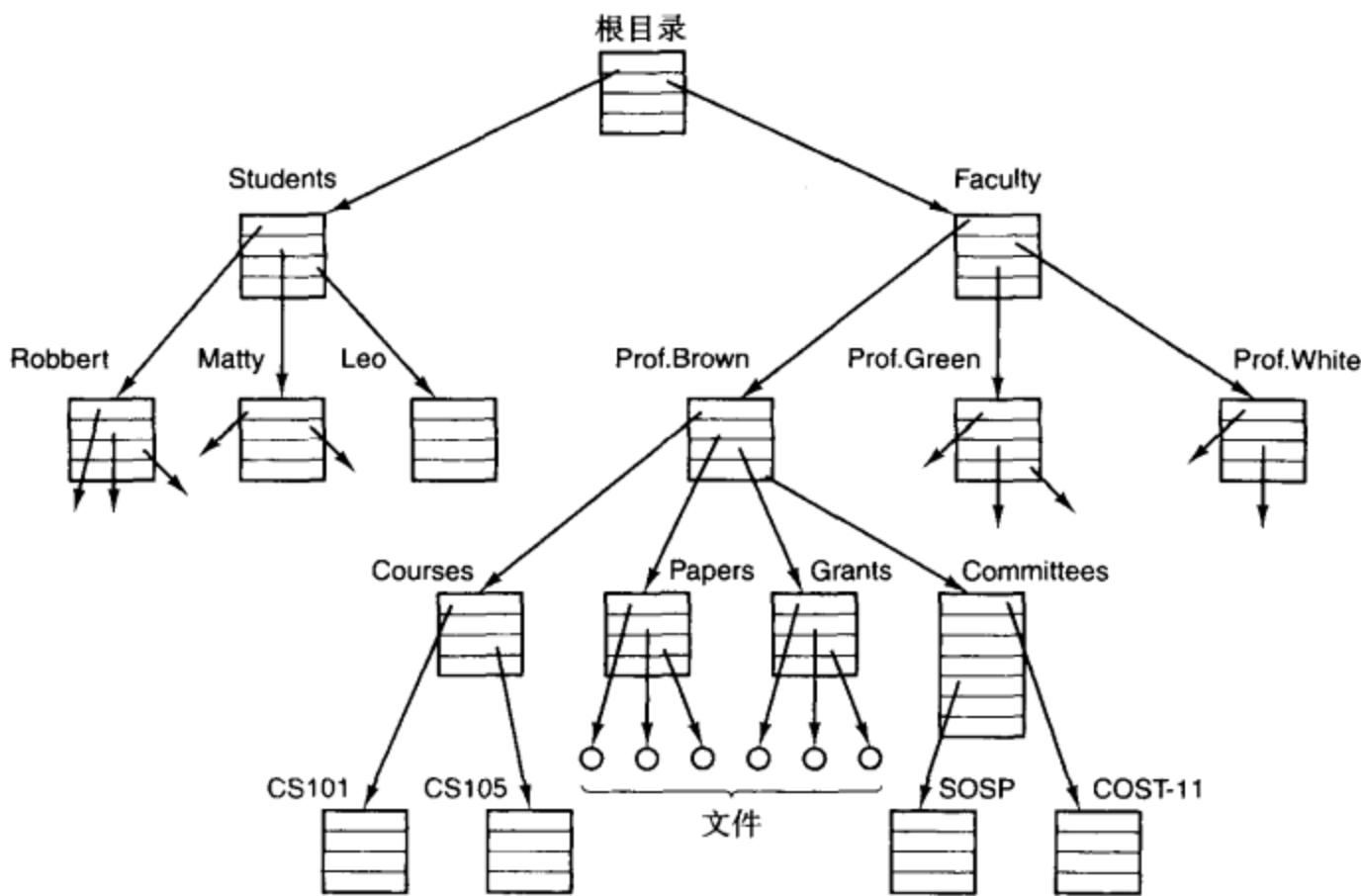


图 1.6 某大学一个系的文件系统

在任意一个时刻，每个进程都会有一个当前的工作目录（working directory），在这种目录下，路径名不是用斜杠开始的。例如，在图 1.6 中，如果当前工作目录为 /Faculty/Prof.Brown，那么路径名 Courses/CS101 和刚才的绝对路径名 /Faculty/Prof.Brown/Courses/CS101 是完全等效的。进程可以通过系统调用来改变它的当前工作目录。

MINIX 3 中的文件和目录通过一个 11 位的二进制码来保护。保护码包括三个 3 位的域，分别用来描述文件的所有者、同组用户和其他用户，剩下的 2 位以后再讨论。每个域有 1 位标识读权限、1 位标识写权限和 1 位标识可执行权限，这三位即为 **rwx** 位。例如，保护码 *rwxr-x--x* 表示文件的所有者可以进行读、写和执行操作；同组用户可以读和执行，但不能写；而其他用户只能执行，不能读写。对目录来说，*x* 表示搜索权限，短横线（-）表示不具备相应权限。

在对一个文件进行读写操作之前，先要把它打开，此时，系统就会进行访问权限检查。如果访问权限是许可的，系统将返回一个整数，称为文件描述符（file descriptor），用于后续的操作。如果访问权限不够，就返回一个错误码（-1）。

MINIX 3 中的另一个重要概念是文件系统的挂装（mount）。多数个人计算机都配有一个或多个 CD-ROM 驱动器。为了对此类可移动介质（如 CD-ROM、DVD、软盘、Zip 驱动器等）提供一种简洁的访问方式，MINIX 3 允许将光驱上的文件系统挂装到主文件树上。如图 1.7(a)所示，在执行 mount 系统调用之前，在硬盘上有一个根文件系统；在 CD-ROM 上也有一个文件系统，这两个文件系统是相互独立、没有关联的。

然而，CD-ROM 上的文件系统无法访问，因为无法确定它上面的文件的路径名。MINIX 3 不允许使用驱动器名或数字作为前缀的路径名，这种表示方式是设备相关的，操作系统应尽量避免。在 MINIX 3 中，程序员可以用 mount 系统调用将 CD-ROM 上的文件系统挂装到根文件系统中的任何位置。在图 1.7(b)中，CD-ROM 上的文件系统被挂装在目录 *b* 下，这样就可以去访问文件 /*b/x* 和 /*b/y*。如果目录 *b* 下原先就存有文件，那么在新的文件系统挂装期间，这些文件就无法访问，因为目录 *b* 现在指向的是 CD-ROM 的根目录（这种情况通常不会造成太多不便，因为文件系统总是被挂装在一个空目录下）。如果在一个系统中包含有多个硬盘，那么可以把它们都挂装到同一棵树中。

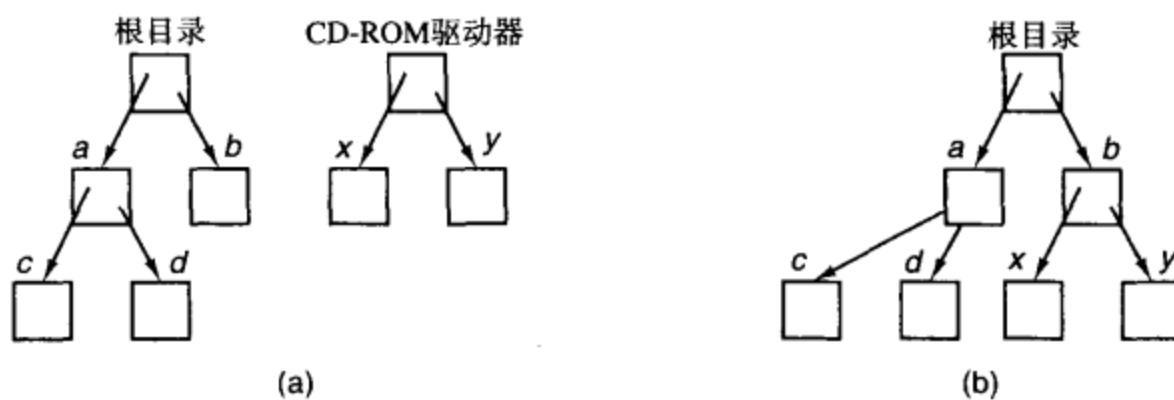


图 1.7 (a)在挂装前, CD-ROM 驱动器上的文件不可访问; (b)在挂装后, 它们是文件层次结构的一部分

MINIX 3 的另一个重要概念是设备文件 (special file)。提供设备文件的目的是使 I/O 设备使用起来更类似于文件。这样, 对设备的读写操作就可以使用与普通文件相同的系统调用。设备文件分为两类: 块设备文件 (block special files) 和字符设备文件 (character special files)。块设备文件描述的是以随机访问的数据块为单元的设备, 如磁盘。在打开一个块设备文件后, 可以直接去访问它的某一个数据块, 如第 4 个数据块, 而不用考虑其文件系统的内部结构。类似地, 字符设备文件指那些以字符流方式进行操作的设备, 如打印机、调制解调器等。一般来说, 设备文件都保存在 /dev 目录下。例如, /dev/lp 可能是一台行式打印机。

最后我们来了解一下与进程和文件都有关的管道 (pipe)。管道是一种用来连接两个进程的虚拟文件, 如图 1.8 所示。当进程 A 和进程 B 想要通过管道来交流信息时, 必须先创建一个管道。然后, 如果进程 A 要给进程 B 发送数据, 它就把这些数据写入管道中, 就好像它是一个输出文件。而进程 B 就可以从管道中读取这些数据, 就好像它是一个输入文件。这样, 在 MINIX 3 中, 进程之间的通信就像是普通文件的读写。一个进程判断其输出是普通文件还是管道的唯一方法, 是调用一条特殊的系统调用。

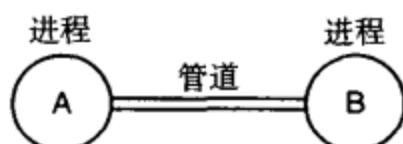


图 1.8 通过管道连接的两个进程

1.3.3 命令解释器

操作系统是实现系统调用的代码, 而编辑器、编译器、汇编程序、链接程序以及命令解释器等都不是操作系统的组成部分, 虽然它们也是非常重要、非常有用的。在本小节中, 我们将大概介绍一下 MINIX 3 的命令解释器: shell。尽管它本身不是操作系统的一部分, 但它使用了大量的操作系统特性, 因而可以把它作为一个很好的范例, 来学习各种系统调用的使用方法。同时, 它也是终端用户与操作系统之间的主要接口, 除非用户使用了一个图形用户界面。命令解释器有多个版本, 如 csh, ksh, zsh 和 bash 等, 它们都来源于最初的 sh, 并且都支持以下描述的功能。

当用户登录进入系统时, 同时将启动一个 shell, 它以终端作为标准的输入和输出。一般来说, 它首先会显示一个系统提示符, 如美元符号, 提示用户 shell 正在等待接收一条命令。这时, 如果用户键入

```
date
```

那么 shell 将创建一个子进程并使其运行 date 程序。在子进程运行期间, shell 将等待它结束。当子进程结束后, shell 再次显示系统提示符并等待下一个输入。

用户可以将标准输出重定向到一个文件，如

```
date > file
```

同样，也可以将标准输入进行重定向，如

```
sort <file1 >file2
```

这条命令将调用 *sort* 程序，将 *file1* 中的数据进行排序，并把排序的结果输出到 *file2*。

通过使用管道，可以将一个程序的输出作为另一个程序的输入，如

```
cat file1 file2 file3 | sort >/dev/lp
```

这条命令调用 *cat* 程序将三个文件进行合并，并把结果送到 *sort* 程序进行排序。然后，*sort* 的输出又被重定向到文件 */dev/lp*，而这正是打印机的设备文件名。

如果用户在一条命令的后面加上一个“&”符号，那么 shell 将不等待其结束而直接显示系统提示符。因此，

```
cat file1 file2 file3 | sort >/dev/lp &
```

将启动 *sort* 程序作为后台任务执行，这样，当这次排序工作还在进行的时候，用户就可以去做其他的事情，而不必等待其结束。shell 还有许多其他的特性，由于篇幅所限，这里不再详述，具体内容请参阅一些 UNIX 系统的入门书籍，如 Ray and Ray (2003) 和 Herborth (2005)。

1.4 系统调用

在大致了解了 MINIX 3 的进程和文件后，下面我们来看一下操作系统与应用程序之间的接口，也就是系统调用。虽然此处的讨论基于 POSIX (国际标准 9945-1)，因而是适用于 MINIX 3, UNIX 和 Linux 的，但实际上，大多数现代操作系统都有功能相似的系统调用，尽管在实现细节上可能不尽相同。由于发出一个系统调用的方式往往与具体的机器有关，而且通常用汇编语言来实现，所以为了能在 C 程序中使用系统调用，通常需要另外再构造一个函数库。

请记住以下事实：任何单 CPU 计算机一次只能执行一条指令。如果一个进程正在用户态下运行一个程序，然后它需要一个系统服务，如读取文件数据，那么它就必须执行一个陷阱或系统调用指令，把控制权交给操作系统。操作系统通过检查此次调用的参数，判断出该进程所需要的服务类型，然后去执行相应的服务功能并把控制权交还给用户进程，从系统调用后面的那条指令开始执行。从某种意义上来说，发出一个系统调用类似于发出一个特殊的函数调用，两者的区别仅在于，发出系统调用后，将进入内核或其他的特权操作系统组件，而函数调用则不会这样。

为了弄清楚系统调用的内在机理，我们以 *read* 系统调用为例。它有三个参数：第一个参数指定所访问的文件，第二个参数指定所用的缓冲区，第三个参数指定要读取的字节数。在 C 程序中调用该系统调用的方法如下：

```
count = read(fd, buffer, nbytes);
```

该系统调用会把实际读到的字节数返回给 *count* 变量。在正常情形下，这个值与 *nbytes* 相等，但有时可能会小一些。例如，在读文件时碰上了文件结束符，从而提前结束此次读操作。

如果由于参数无效或磁盘访问错误等原因，使得此次系统调用无法完成，则 *count* 被置为 -1，同时把一个错误码放在全局变量 *errno* 中。我们在编写程序时，应该经常检查系统调用的返回值，以确认此次调用是否已被正确执行。

MINIX 3 总共有 53 条系统调用，被分为 6 大类，如图 1.9 所示。除此之外，还有少量专用的系统调用，这里不予介绍。在以下几个小节中，我们将简要地介绍图 1.9 中的每一个系统调用，看看它们的功能是什么。由于个人计算机的资源管理功能非常有限（起码与用户众多的大型机相比是如此），因此，这些系统调用所提供的服务在很大程度上决定了操作系统应提供的主要功能。

进程管理	<code>pid = fork()</code>	创建一个与父进程相同的子进程
	<code>pid = waitpid(pid, &statloc, opts)</code>	等待一个子进程结束
	<code>s = wait(&status)</code>	waitpid 的老版本
	<code>s = execve(name, argv, envp)</code>	替换一个进程的内核映像
	<code>exit(status)</code>	终止进程的执行并返回 status
	<code>size = brk(addr)</code>	设置数据段的大小
	<code>pid = getpid()</code>	返回调用进程的标识号
	<code>pid = getpgrp()</code>	返回调用进程的组标识号
	<code>pid = setsid()</code>	创建一个新的会话并返回其组标识号
	<code>l = ptrace(req,pid,addr,data)</code>	用于调试
信号	<code>s = sigaction(sig, &act, &oldact)</code>	定义针对信号的处理操作
	<code>s = sigreturn(&context)</code>	从信号返回
	<code>s = sigprocmask(how, &set, &old)</code>	检查或修改信号屏蔽码
	<code>s = sigpending(set)</code>	获得阻塞信号集合
	<code>s = sigsuspend(sigmask)</code>	替换信号屏蔽码并挂起进程
	<code>s = kill(pid, sig)</code>	给进程发送一个信号
	<code>residual = alarm(seconds)</code>	设置警报时钟
	<code>s = pause()</code>	将调用进程挂起直到下一个信号
文件管理	<code>fd = creat(name, mode)</code>	创建一个新文件（已过时）
	<code>fd = mknod(name, mode, addr)</code>	创建一个普通文件、设备文件或目录的 i 节点
	<code>fd = open(file, how, ...)</code>	打开一个文件进行读、写或读写
	<code>s = close(fd)</code>	关闭一个文件
	<code>n = read(fd, buffer, nbytes)</code>	从文件中读数据到缓冲区
	<code>n = write(fd, buffer, nbytes)</code>	将缓冲区中的数据写入文件
	<code>pos = lseek(fd, offset, whence)</code>	移动文件指针
	<code>s = stat(name, &buf)</code>	获取文件的状态信息
	<code>s = fstat(fd, &buf)</code>	获取文件的状态信息
	<code>fd = dup(fd)</code>	为打开文件分配一个新的文件描述符
	<code>s = pipe(&fd[0])</code>	创建一个管道
	<code>s = ioctl(fd, request, argp)</code>	对设备文件进行控制操作
	<code>s = access(name, amode)</code>	检查一个文件的可访问性
	<code>s = rename(old, new)</code>	修改文件的名字
	<code>s = fcntl(fd, cmd, ...)</code>	文件加锁及其他操作
目录及文件系统管理	<code>s = mkdir(name, mode)</code>	创建一个新目录
	<code>s = rmdir(name)</code>	删除一个空目录
	<code>s = link(name1, name2)</code>	创建一个新的目录项 name2，指向 name1
	<code>s = unlink(name)</code>	删除一个目录项
	<code>s = mount(special, name, flag)</code>	挂装一个文件系统
	<code>s = umount(special)</code>	卸装一个文件系统
	<code>s = sync()</code>	将缓冲区中的数据块回写到磁盘
	<code>s = chdir(dirname)</code>	改变当前工作目录
	<code>s = chroot(dirname)</code>	改变根目录

图 1.9 MINIX 的系统调用。*fd* 为文件描述符，*n* 为字节个数

保护	<code>s = chmod(name, mode)</code> <code>uid = getuid()</code> <code>gid = getgid()</code> <code>s = setgid(uid)</code> <code>s = setgid(gid)</code> <code>s = chown(name, owner, group)</code> <code>oldmask = umask(complmode)</code>	改变文件的保护位 获取调用进程的 uid 获取调用进程的 gid 设置调用进程的 uid 设置调用进程的 gid 改变文件的所有者及其所在的组 改变模式屏蔽码
时间管理	<code>seconds = time(&seconds)</code> <code>s = stime(tp)</code> <code>s = utime(file, timep)</code> <code>s = times(buffer)</code>	获取当前时间，以 1970 年 1 月 1 日为起点 设置当前时间，以 1970 年 1 月 1 日为起点 设置文件的“上次访问”时间 获取用户和系统所使用的时间

图 1.9 MINIX 的系统调用。*fd* 为文件描述符，*n* 为字节个数（续）

需要指出的是，POSIX 函数调用与系统调用之间的映射并不是一一对应的。在 POSIX 标准中，定义了许多函数，每一个兼容的系统都必须支持这些函数。但它并未明确规定这些函数到底是系统调用、库函数，还是其他的什么东西。在有些情形下，POSIX 函数以库函数的形式出现在 MINIX 3 中。而在其他情形下，由于几个函数之间的差别非常细微，所以通常把它们包含在同一条系统调用中。

1.4.1 进程管理的系统调用

图 1.9 中的第一类系统调用主要用于进程管理。先来看一下 `fork` 系统调用。在 MINIX 3 中，`fork` 是创建一个新进程的唯一途径。它实际上创建的是原进程的一个副本，包括文件描述符、寄存器的值等，所有内容都是完全相同的。在调用 `fork` 后，原进程和新进程（即父进程和子进程）各自执行，互不相关。在执行 `fork` 时，两个进程的所有对应变量都具有相同的值，但由于父进程和子进程的地址空间是相互独立的，因此在 `fork` 执行之后，如果其中一个进程的变量值发生了变化，并不会影响到另一个进程（代码段是不可修改的，由父、子进程共享）。在正常情形下，如果 `fork` 函数的返回值为 0，则表明当前进程是子进程；如果返回值为一个正整数，则表明当前进程是父进程，而该整数即为子进程的标识号 **PID**。因此，尽管在调用 `fork` 函数后，父进程和子进程的内容是完全相同的，但是通过这个函数的返回值，还是可以将父进程和子进程区分开来。

在多数情形下，在执行完 `fork` 之后，子进程需要执行一段与父进程不同的代码。例如，对于一个 `shell`，它首先从终端读取一条命令，然后创建一个子进程来执行该命令，并等待子进程执行完毕，然后再读取下一条命令。为了等待子进程结束，父进程会执行一个 `waitpid` 系统调用，该调用将使父进程阻塞，直到子进程结束（如果有多个子进程，只要任何一个子进程结束即可）。通过把第一个参数设置为 -1，`waitpid` 也可以等待某个特定的子进程，或某个比较老的子进程。当 `waitpid` 结束时，它的第二个参数 `statloc` 指向的是子进程的终止状态值所在的内存单元（正常结束或异常结束，以及正常结束时的返回值）。此外，`waitpid` 的第三个参数还可以设置不同的选项。`waitpid` 取代了先前的 `wait` 系统调用，`wait` 系统调用虽然已经过时，但为了保持向后的兼容性，还是把它留着。

现在来看一下 `shell` 是如何使用 `fork` 系统调用的。当用户键入一条命令时，`shell` 首先创建一个新进程。这个子进程必须执行该用户命令，这是通过 `execve` 系统调用来实现的，它将用第一个参数所指定的可执行文件来替换当前的内核映像（实际上，真正的系统调用是 `exec`，但是有几个不同的库函数都调用了它，并且使用了不同的参数和稍微变化的函数名。为方便起见，我们把这些库

函数都视为系统调用)。一个高度简化的 shell 框架如图 1.10 所示, 它演示了 fork, waitpid 和 execve 等系统调用的用法。

```
#define TRUE 1

while (TRUE){
    type_prompt();
    read_command(command, parameters);
    /* 无限循环 */
    /* 在屏幕上显示提示符 */
    /* 从终端读取输入 */

    if (fork() != 0){
        /* Parent code. */
        waitpid(-1, &status, 0);
        /* 创建子进程 */
        /* 等待子进程退出 */
    } else{
        /* Child code. */
        execve(command, parameters, 0);
        /* 执行命令 */
    }
}
```

图 1.10 一个简化的 shell。在本书中, TRUE 被定义为 1

在一般情形下, execve 有三个参数: 待执行的文件名、指向参数数组的指针和指向环境数组的指针。系统提供了若干个不同的库函数, 如 *exec1*, *execv*, *execle* 和 *execve*, 它们通过不同的方式来忽略或设定这些参数。在本书中, 我们用名字 **exec** 来表示在所有这些库函数中都要用到的那个系统调用。

考虑如下一条命令:

```
cp file1 file2
```

它的功能是为文件 *file1* 制作一个副本 *file2*。在 shell 创建一个子进程后, 子进程将查找并执行程序 *cp*, 同时向它传递执行的参数: 源文件名和目标文件名。

cp 程序的主函数格式如下:

```
main(argc, argv, envp)
```

其中 *argc* 是命令行中的参数个数(包括程序名在内)。对于上述例子, *argc* 为 3。

第二个参数 *argv* 是一个指向数组的指针, 该数组的第 *i* 个元素就是命令行中的第 *i* 个字符串。对于上述例子, *argv[0]* 为 “*cp*”, *argv[1]* 为 “*file1*”, *argv[2]* 为 “*file2*”。

第三个参数 *envp* 是一个环境指针, 所谓环境就是一个字符串数组, 其中每个元素形如 *name = value*, 用来将各种环境信息传递给程序, 如终端类型、主目录名等。在图 1.10 中, 不需要向子进程传递任何环境信息, 因此 *execve* 的第三个参数为空。

如果读者觉得 **exec** 的用法很复杂, 请不必太过担忧。实际上在所有的 POSIX 系统调用中, 它是最复杂的一个, 其他的系统调用都要比它简单。例如, 当一个进程完成任务后, 可以用 **exit** 系统调用来结束运行。这个系统调用只有一个参数, 也就是退出的状态值(0~255)。这个值通过 **waitpid** 系统调用中的 *statloc*, 返回给父进程。*statloc* 的低字节存放结束状态, 0 表示正常退出, 其他值表示不同的错误类型。*statloc* 的高字节包含子进程的退出状态(0~255)。例如, 假设父进程执行

```
n = waitpid(-1, &statloc, options);
```

那么父进程将会被阻塞, 直到它的某一个子进程运行结束。如果子进程退出时用 4 作为 *exit* 的参数值, 则当父进程被唤醒时, *n* 中存放的是该子进程的 PID, 而 *statloc* 的值为 0x0400(在本书中将使用 C 语言的语法, 用 0x 来表示一个十六进制常量)。

在 MINIX 3 中，进程的内存空间被分为三个部分：代码段（text segment，即程序代码）、数据段（data segment，即变量）和栈段（stack segment）。数据段从下往上增长，而栈从上向下增长，如图 1.11 所示。在这两者之间是空闲的地址空间。栈的增长是随着程序的执行自动进行的，而数据段的扩展则需要通过 brk 系统调用来显式地完成，brk 有一个参数来指定数据段的结束地址，它可以比当前值大（表示扩展数据段），或是比当前值小（表示缩小数据段）。当然，这个参数必须小于栈指针，否则栈和数据段将会重叠，这是不允许的。

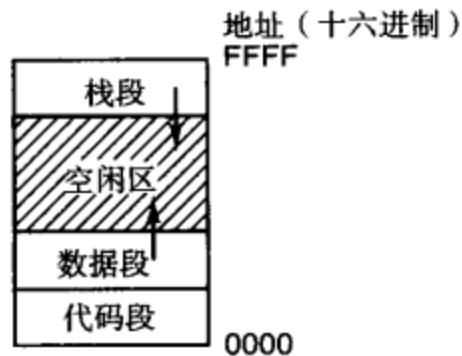


图 1.11 进程有三个段：代码段、数据段和栈段。在本例中，这三个段都在同一个地址空间中，但系统也支持相互独立的代码空间和数据空间

出于程序员的方便考虑，系统还提供了一个库函数 `sbrk` 来改变数据段的大小，它只有一个参数，即数据段的增加量（以字节为单位，负数表示缩小数据段）。它的工作原理是：获取数据段的当前大小（该值由 `brk` 返回），然后根据用户给定的参数，计算出新的大小，再通过系统调用来申请所需的空间。`brk` 和 `sbrk` 都不是 POSIX 标准的内容，程序员最好使用 `malloc` 库函数来申请动态内存空间。

下一个系统调用 `getpid` 也非常简单，它返回调用进程的进程标识号 PID。如前所述，在调用 `fork` 时，只有父进程能够获得子进程的 PID。如果子进程想要知道它自己的 PID，就必须使用 `getpid`。类似的系统调用包括：`getpgrp` 返回调用进程的组标识号，`setsid` 创建一个新的会话，并将进程组的 PID 设置为调用者的 PID。会话与 POSIX 的可选特性——作业控制有关，MINIX 3 不支持作业控制，因此这里不做进一步的讨论。

最后一个系统调用是 `ptrace`，它主要用来对被调试的程序进行控制。通过 `ptrace`，调试器可以读、写被控进程的地址空间，并进行其他方式的管理。

1.4.2 信号管理的系统调用

在多数情形下，进程间的通信是事先预计好的，但有时可能需要处理不可预知的通信问题。例如，假设用户无意中启动了一个文本编辑器来显示一个大文件的全部内容，但随即他意识到该操作并不需要，这时就需要一种方法来终止编辑器的工作。在 MINIX 3 中，用户可以通过按 CTRL-C 组合键做到这一点，也就是说，向编辑器发送一个信号。当编辑器收到这个信号时，就会停止文件的显示。信号还可用来报告硬件捕获到的某些陷阱，如非法指令或浮点运算溢出等。另外，超时也可以通过信号来实现。

对于一个尚未声明愿意接收信号的进程，如果此时它收到一个信号，那么该进程将会被杀死。为了避免这种结局，进程可以用 `sigaction` 系统调用来声明它准备接收某种类型的信号，并提供两个参数：一个是信号处理程序的地址，另一个是内存单元，用于保存该信号的原先处理程序的地址。在执行完 `sigaction` 系统调用后，如果进程收到相关类型的信号（例如由 CTRL-C 产生的信号），那么就会把进程的当前状态压入栈中，然后调用相应的信号处理程序。信号处理的时间可以任意长，

也可以调用任何系统调用，但在通常情形下，它都会比较短。当信号处理程序结束后，就调用 `sigreturn` 函数，返回到被此次信号所打断的指令，继续往下执行。`sigaction` 替换了原先的 `signal` 系统调用，但出于兼容性的考虑，在系统中仍然保留了 `signal`，只是把它变成了一个库函数。

在 MINIX 3 中，信号可以被阻塞。被阻塞的信号一直被挂起，直到阻塞解除。在这段时间内，它不会被传递，但也不会丢失。`sigprocmask` 系统调用允许进程通过提交一幅位图的方式，来定义一组被阻塞的信号集合。与之相对应，进程也可以使用系统调用 `sigpending` 来查询当前因阻塞而挂起的信号集。最后，`sigsuspend` 系统调用允许进程去设置阻塞信号的位图并将其挂起。

除了使用函数来捕获信号之外，程序也可以使用常量 `SIG_IGN` 来忽略指定类型的信号，或者使用 `SIG_DFL` 来恢复默认的信号处理程序。默认的处理方式随信号而异，可以是撤销该进程，或者是忽略该信号。为了说明 `SIG_IGN` 的用法，请看以下命令，其功能是让 shell 创建一个后台进程：

```
command &
```

对于这个后台进程来说，它不希望被 `SIGINT` 信号打扰（该信号由 `CTRL-C` 组合键产生），因此 shell 程序在执行 `fork` 之后、`exec` 之前，需要执行

```
sigaction(SIGINT, SIG_IGN, NULL);
```

和

```
sigaction(SIGQUIT, SIG_IGN, NULL);
```

这两条命令来忽略 `SIGINT` 和 `SIGQUIT` 信号（`SIGQUIT` 信号由 `CTRL-\` 组合键产生，它与 `SIGINT` 信号的作用基本相同。不同之处在于，如果它未被捕获或忽略，则将产生被撤销进程的内核映像转储）。对于前台进程（即命令中不带 `&`），这些信号不能被忽略。

按 `CTRL-C` 组合键并不是发送信号的唯一途径，使用 `kill` 系统调用也可以向一个进程发送信号（前提是发送方和接收方具有相同的用户标识号 `UID`，也就是说，无关的进程之间不能发送信号）。再来看一下后台进程的例子，假设一个后台进程已被启动，但随后发现它应被终止，此时 `SIGINT` 和 `SIGQUIT` 都已被禁用，所以在这种情形下，只能另想办法。解决方案就是使用 `kill` 程序，该程序将使用 `kill` 系统调用来发送一个信号。当后台进程收到信号 9（`SIGKILL`）时，该进程将被撤销。`SIGKILL` 不能被捕获或忽略。

对于许多实时应用，当一个进程在运行一段时间间隔后，需要发生一次中断，以进行其他的一些处理，例如在不可靠的通信线路上重传一个丢失的数据包。为了处理此类情形，系统提供了 `alarm` 系统调用。`alarm` 的参数指定了一个时间间隔，以秒为单位。然后，一旦进程的运行时间超过了该时间间隔，就会收到一个 `SIGALRM` 信号。在任意时刻，一个进程只能设定一个警报时钟。例如，假设进程先设定了一个 10 s 的警报时钟，然后在 3 s 后又设定了一个 20 s 的警报时钟，那么其中只有一个有效，即在第二个 `alarm` 调用之后的 20 s，将会发送一个 `SIGALRM` 信号。如果 `alarm` 的参数为 0，则所有挂起的 `SIGALRM` 信号都被取消。如果一个 `SIGALRM` 信号未被捕获，那么就采用默认的处理方法，将相应的进程撤销。

在某些情形下，一个进程在信号到达之前无须进行任何操作。例如，有一个测试阅读速度和理解能力的计算机辅助教学系统，它先是在屏幕上显示一些文本，然后调用 `alarm`，让系统在 30 s 后给自己发送一个信号。那么当学生正在阅读课文时，程序无须执行任何操作。这时，它可以采用执行空循环的方法来等待信号的到来，但这样的话，就浪费了 CPU 时间。而在系统中，可能还有其他的进程或用户需要使用 CPU。因此，更好的做法是使用 `pause` 系统调用，它将通知 MINIX 3 把当前进程挂起，直至信号到来。

1.4.3 文件管理的系统调用

许多系统调用与文件系统有关。在本小节中，我们只讨论对单个文件进行操作的系统调用；下一节将讨论对目录和文件系统进行操作的系统调用。`creat` 系统调用的功能是创建一个新文件，其参数包括文件的文件名和保护模式（这个系统调用的名字是`creat`而不是`create`的原因随着时间的流逝已无从考证）。例如，

```
fd = creat("abc", 0751);
```

这个例子将创建一个名为 *abc* 的文件，其保护模式为 0751（在 C 语言中，以 0 开头的整数代表一个八进制数常量）。0751 的低 9 位分别指明了文件所有者、同组用户和其他用户的访问权限，即文件所有者为 7，表示可读、可写、可执行；同组用户为 5，表示可读、可执行；其他用户为 1，表示只可执行。

`creat` 系统调用在创建文件的同时，还以可写的方式将其打开。用户可以使用返回的文件描述符 *fd*，来对该文件进行写操作。如果对一个现有的文件进行 `creat` 操作，那么该文件的内容将被清空，文件的长度变为 0。当然，前提条件是此次操作的权限是许可的。在当前系统中，`creat` 已经过时了，因为使用 `open` 系统调用也可以创建新文件。但为了保持兼容性，还是把 `creat` 保留了下来。

设备文件的创建使用的是 `mknod`，而不是 `creat`。典型的用法为

```
fd = mknod("/dev/ttyc2", 020744, 0x0402);
```

这将创建一个名为 */dev/ttyc2*（2 号控制台常用的文件名）的文件，并将其模式代码设置为八进制数的 020744（这表示该文件是字符设备文件，保护模式为 *rwxr--r-*）。第三个参数的高字节指定其主设备号为 4，低字节指定其次设备号为 2。主设备号可以取任何值，但对于名为 */dev/ttyc2* 的文件，次设备号应当为 2。`mknod` 只能被超级用户使用。

在读写一个文件之前，首先要使用 `open` 系统调用将其打开。`open` 的第一个参数指定文件的路径名，可使用绝对路径名或相对于当前工作目录的相对路径名；第二个参数指定打开的方式，如 *O_RDONLY*、*O_WRONLY* 或 *O_RDWR*，分别表示只读、只写和可读可写。在打开一个文件后，就可以使用它返回的文件描述符 *fd* 对文件进行读写操作。在使用完一个文件后，还要用 `close` 系统调用把它关闭，这样它的文件描述符就可以空出来，给其他文件使用。

最常用的系统调用当属 `read` 和 `write`。我们在前面已看到过 `read`，而 `write` 的参数与之完全相同。

多数程序对文件的读写操作都是顺序进行的，但有时也需要随机地去访问文件的任意部分。每一个文件都有一个指针来指明其当前的访问位置。在顺序读写时，该指针通常指向下一个即将读写的字节。我们可以使用 `lseek` 系统调用来直接修改文件指针的值，这样，随后的 `read` 或 `write` 操作就可以在文件的任意位置进行，甚至可以超越文件的末尾。

`lseek` 有三个参数：第一个是文件描述符，第二个是文件的位置，第三个指明该文件位置是相对于文件开头、当前位置还是文件末尾。`lseek` 的返回值是文件指针被修改之后的绝对位置。

对于每一个文件，MINIX 3 记录了如下信息：文件类型（普通文件、设备文件、目录等）、文件大小、最后修改时间等。程序可以通过 `stat` 或 `fstat` 系统调用来获取这些信息，它们的区别仅在于 `stat` 是通过文件名来指定文件，而 `fstat` 则使用文件描述符。显然，`fstat` 比较适合于已打开的文件，尤其是像标准输入和标准输出这种文件名不可知的情形。`stat` 和 `fstat` 的第二个参数是一个指针，指向一个用来存放文件信息的数据结构，该结构如图 1.12 所示。

```

struct stat{
    short st_dev;           /* i 节点所属的设备 */
    unsigned short st_ino;   /* i 节点号 */
    unsigned short st_mode;  /* 模式字 */
    short st_nlink;         /* 链接数 */
    short st_uid;           /* 用户标识号 */
    short st_gid;           /* 组标识号 */
    short st_rdev;          /* 设备文件的主/次设备号 */
    long st_size;           /* 文件大小 */
    long st_atime;          /* 最后访问时间 */
    long st_mtime;          /* 最后修改时间 */
    long st_ctime;          /* 对 i 节点的最后修改时间 */
};

```

图 1.12 stat 和 fstat 系统调用用于存放返回信息的数据结构。在实际代码中，有些类型使用了符号名

在操作文件描述符的时候，有时可以使用 dup 系统调用。例如，假设一个程序需要关闭标准输出（文件描述符为 1），并用一个普通文件来作为标准输出，然后调用一个函数向标准输出写入一些信息，最后再恢复原先的状态。为了实现这些功能，可以先关闭文件描述符 1，再打开一个新文件，这时该文件就成为标准输出（假设标准输入，即文件描述符 0 正在使用），但如果这样的话，就无法恢复原先的标准输出。

解决办法是先调用

```
fd = dup(1);
```

该操作将为标准输出分配一个新的文件描述符 fd，并使之对应于标准输出文件。然后，就可以将标准输出关闭，并打开一个新文件，使用该文件来作为标准输出。当需要恢复原先的标准输出时，先关闭文件描述符 1，然后再执行

```
n = dup(fd);
```

这时，最小的文件描述符 1 被定向到 fd 所指的文件。最后将 fd 关闭就恢复了最初状态。

dup 系统调用有一个变体，它允许将一个未使用的文件描述符定向到某个已打开的文件。其调用方法为

```
dup2(fd, fd2);
```

此处 fd 指向一个打开的文件，fd2 是一个未使用的文件描述符，当执行完这条语句后，fd2 将指向 fd 所指向的文件。例如，假设 fd 指向标准输入（文件描述符 0），fd2 为 4，那么在此调用后，描述符 0 和 4 都指向标准输入。

如前所述，MINIX 3 中的进程间通信可以使用管道。如果一个用户键入

```
cat file1 file2 | sort
```

那么 shell 将创建一个管道并将第一个进程的标准输出信息写入到管道中，于是第二个进程的标准输入便可以从该管道中读取。pipe 系统调用创建一个管道并返回两个文件描述符，一个用于写，另一个用于读。pipe 的调用格式为

```
pipe(&fd[0]);
```

这里 fd 是由两个整数组成的数组，fd[0] 存放读操作的文件描述符，fd[1] 存放写操作的文件描述符。一般来说，在本条语句之后会调用一个 fork 来创建一个子进程，然后父进程关掉用于读的文件描述

符，子进程关掉用于写的文件描述符（或者相反），这样便可以做到一个进程向管道中写数据，另一个进程从管道中读数据。

图 1.13 给出了一个函数的框架，它创建了两个进程，并通过管道的方法，将进程 1 的输出导向进程 2（在实际代码中，还要进行错误检查和参数处理）。函数的基本过程如下：首先创建一个管道，然后执行 fork 系统调用，将父进程作为管道中的进程 1，子进程作为进程 2。由于待运行的两个进程 *process1* 和 *process2* 并不知道它们是管道的一部分，所以必须对文件描述符进行操作，使得进程 1 的标准输出和进程 2 的标准输入都指向管道。父进程首先关掉从管道读的文件描述符和标准输出，然后执行 dup，使文件描述符 1 可被用于向管道写入。注意，dup 总是返回最小的可用文件描述符，在这里即为 1。然后程序关闭另一个管道文件描述符。

```
# define STD_INPUT 0           /* 标准输入的文件描述符 */
# define STD_OUTPUT 1          /* 标准输出的文件描述符 */
pipeline(process1, process2)  /* 指向程序名的指针 */
char *process1, *process2;
{
    int fd[2];

    pipe(&fd[0]);            /* 创建一个管道 */
    if (fork() != 0){
        /* 父进程执行如下语句 */
        close(fd[0]);         /* 进程 1 不需要从管道读 */
        close(STD_OUTPUT);     /* 准备新的标准输出 */
        dup(fd[1]);           /* 将标准输出指向 fd[1] */
        close(fd[1]);          /* 不再需要此文件描述符 */
        execl(process1, process1, 0);
    } else {
        /* 子进程执行如下语句 */
        close(fd[1]);          /* 进程 2 不需要向管道写入 */
        close(STD_INPUT);       /* 准备新的标准输入 */
        dup(fd[0]);             /* 将标准输入指向 fd[0] */
        close(fd[0]);           /* 不再需要此文件描述符 */
        execl(process2, process2, 0);
    }
}
```

图 1.13 建立一个两进程管道的框架

在 exec 系统调用之后，父进程将保留文件描述符 0 和 2，而文件描述符 1 则用于向管道中写入。子进程的代码与父进程类似。execl 的参数是重复的，这是因为它的第一个参数是待执行的文件名，而第二个参数是执行文件的第一个参数，对于多程序来说，该参数即为执行文件的文件名。

下一个系统调用 ioctl 适用于所有设备文件。例如，它可以用于块设备驱动程序，像 SCSI 驱动程序就用它来控制磁带机和 CD-ROM。不过，它的主要用途还在于字符设备文件，尤其是终端。POSIX 定义了许多库函数，最终都转化为相应的 ioctl 调用。例如，库函数 tcgetattr 和 tcsetattr 都使用 ioctl 来改变终端模式和各种属性。

传统意义上的终端模式有三种：熟模式（cooked）、生模式（raw）和 cbreak 模式。其中，熟模式是最常用的终端模式。在该模式下，字符的删除和终止能正常地工作，CTRL-S 组合键和 CTRL-Q 组合键能用来停止和恢复终端输出，CTRL-D 组合键表示文件结束，CTRL-C 组合键产生一个中断信号，而 CTRL-\ 组合键产生一个退出信号并强制进行内核映像转储。

在生模式下，所有上述功能都被取消，每一个字符都是在未加处理的情况下直接发送给程序的。用户输入的任何一个字符，都会立即发送给程序。而不像熟模式那样，要等到终端输入了一整行以后才发送给程序。全屏幕编辑器通常使用这种模式。

cbreak 模式介于上述两者之间，用做编辑的删除键和终止键被禁用，**CTRL-D** 组合键也被禁用。但 **CTRL-S**、**CTRL-Q**、**CTRL-C** 和 **CTRL-** 组合键则仍然有效。与生模式一样，单个字符不等一行结束就送给程序（如果禁止行内编辑功能，则没必要等待接收到完整的一行，因为用户不可能像在熟模式下那样改变主意并删除它）。

POSIX 并没有采用生模式、熟模式和 **cbreak** 模式这几个术语。POSIX 中的正規模式（canonical mode）对应于熟模式。在这种模式下，定义了 11 个特殊字符，输入也以行为单位进行。在 POSIX 的非正規模式（noncanonical mode）下，字符的读操作由两个因素来决定，一是可接受的最小字符个数，二是指定的时间，以 0.1 s 为单位。POSIX 标准具有很大的灵活性，可以通过设置一些标志位，使得非正規模式使用起来很像生模式或 **cbreak** 模式。尽管未被 POSIX 采用，但熟模式、生模式等术语更具描述性，因此我们将继续使用它们。

ioctl 有三个参数。例如，假设程序调用 **tcsetattr** 函数来设置终端参数，那么它最终将被转换为以下的系统调用：

```
ioctl(fd, TCSETS, &termios);
```

第一个参数指定一个文件，第二个参数指定操作的类型，第三个参数指定一个 POSIX 数据结构的地址，其中包含了控制字符数组以及各种标志位。还有其他一些操作码，其功能包括：推迟对终端参数所做的修改，直到全部输出被送出；抛弃未读取的输入信息；返回参数的当前值。

access 系统调用可以检查文件访问操作是否具有相应的权限。之所以需要这个系统调用，是因为有些程序可以用另一个用户的 UID 去运行，这种 SETUID 机制将在稍后介绍。

rename 系统调用可以改变一个文件的名字，其参数为旧文件名和新文件名。

最后，**fctl** 系统调用用于对文件进行控制，它有点类似于 **ioctl**。**fctl** 有若干个选项，最常用的是文件加锁。它可以对一个文件的一部分进行加锁和解锁，也可以检测文件的某个部分是否被上锁。**fctl** 本身并不包含任何与锁操作有关的语义，这要由程序员自行定义。

1.4.4 目录管理的系统调用

在本小节中，我们将介绍一些与目录和文件系统有关的系统调用。首先是 **mkdir** 和 **rmdir**，它们分别用来创建和删除空目录。下一个 **link**，它允许同一个文件具有两个或多个名字，通常位于不同的目录下。**link** 的一种典型应用是，在一个开发小组的多个成员之间，可以利用它来共享同一个文件。对于每一个成员来说，在他们自己的目录下，都可以看到该文件，而且文件名也可以不同。共享一个文件不同于给每个人复制一份副本，对于前者，只存在一个文件，因此任何人所做的修改其他人都可以看见；而对于后者，由于每个人都有自己独立的一份，因此他所做的修改只有自己看得见，不会影响到其他人。

图 1.14(a) 中的例子解释了 **link** 的工作原理。有两个用户 *ast* 和 *jim*，每个用户都有自己的目录和一些文件。如果 *ast* 现在执行一个程序，里面包含了以下的系统调用：

```
link("/usr/jim/memo", "/usr/ast/note");
```

则 *jim* 目录下的文件 *memo* 将以文件名 *note* 出现在 *ast* 的目录下。此后，*/usr/jim/memo* 和 */usr/ast/note* 指的是同一个文件。

理解 link 的工作原理有助于掌握其功能。UNIX 中的每个文件都由一个唯一的编号，即 i 节点 (i-node) 号来标识它。i 节点号是 i 节点表的索引值，系统中的每一个文件都有一个 i 节点，里面存放了文件的所有者、文件在磁盘上的存储位置等信息。目录实际上也是文件，只不过它的内容并非通常的数据，而是位于该目录下的每个文件的文件名与它的 i 节点号之间的对应关系，即 <i 节点号，文件名> 这样的组合。在 UNIX 的早期版本中，每个目录项有 16 个字节，其中 2 个字节用于 i 节点号，14 个字节用于文件名。在现代操作系统中，为了支持长文件名，需要更复杂的目录结构，但基本原理还是一样的，每个目录项描述的还是 <i 节点号，文件名> 这样的组合。在图 1.14 中，文件 mail 的 i 节点号为 16，其他文件的 i 节点号如图所示。link 所做的事情只是创建一个新的目录项，它有一个新文件名，但它的 i 节点号则是被链接的那个文件的 i 节点号。在图 1.14(b) 中，两个目录项具有相同的 i 节点号 70，所以它们指向的是同一个文件。如果用 unlink 系统调用删除其中的一个目录项，那么另一个目录项依然存在，相关的文件也依然存在。如果这两个目录项都被删除，那么 UNIX 发现没有任何目录项指向该文件（在 i 节点中有一个域，记录指向该文件的目录项数），于是就把该文件从磁盘上删除。

/usr/ast	/usr/jim
16 mail 81 games 40 test	31 bin 70 memo 59 f.c. 38 prog1

(a)
(b)

/usr/ast	/usr/jim
16 mail 81 games 40 test 70 note	31 bin 70 memo 59 f.c. 38 prog1

图 1.14 (a) 将 /usr/jim/memo 链接到 ast 目录之前的两个目录；(b) 链接之后的情形

如前所述，mount 系统调用可以将两个文件系统合并成一个。通常的情形是，在硬盘上先有一个根文件系统，包含了常用命令的可执行文件及其他常用文件。然后用户可以在 CD-ROM 驱动器中插入一张存有文件的光盘。

接下来，就可以使用 mount 系统调用将 CD-ROM 上的文件系统挂装到根文件系统下，如图 1.15 所示。执行挂装操作的 C 语句为

```
mount ("/dev/cdrom0", "/mnt", 0);
```

其中，第一个参数是 CD-ROM 驱动器 0 的设备文件名，第二个参数是文件树中的挂装点，第三个参数将文件系统的访问方式设定为只读或可读写。



图 1.15 (a) 挂装前的文件系统；(b) 挂装后的文件系统

执行完 mount 后，CD-ROM 驱动器 0 上的文件就可以使用路径名（相对于根目录或当前工作目录）来访问，而与具体的物理设备无关。实际上，第 2 个、第 3 个、第 4 个驱动器都可以挂装到文件树中的任何位置。使用 mount 命令，用户可以将各种可移动存储介质集成到一个统一的文件层次结构中，而不必关心文件是存放在哪一台设备上。在本例中只涉及到 CD-ROM，实际上硬盘或硬

盘的一部分（称为分区或次设备）都可以这样挂装。当一个文件系统不再需要时，可以用 `umount` 系统调用将其卸装。

MINIX 3 在内存中开辟了一个块缓冲区（block cache）来保存最近访问过的数据块，这样就可以避免重复地从磁盘上读数据。如果缓冲区中的某个数据块被修改过，而且在它被写回到磁盘之前系统发生了崩溃，那么整个文件系统就可能会受到损坏。为了避免这种现象，必须定期地将缓冲区中的数据块写回磁盘，从而降低在系统崩溃时数据丢失的风险。在 MINIX 3 中，这项工作是由 `sync` 系统调用完成的，它将把缓冲区中被修改过的数据块写回磁盘。当 MINIX 3 启动后，一个名为 `update` 的程序就会被启动，作为后台进程运行，每隔 30 s 它会执行一次 `sync` 操作，刷新缓冲区的内容。

与目录操作有关的另外两个系统调用是 `chdir` 和 `chroot`。前者用来改变当前的工作目录，后者用来改变根目录。如果先调用

```
chdir("/usr/ast/test");
```

然后再打开文件 `xyz`，那么最后的结果就是打开 `/usr/ast/test/xyz` 文件。`chroot` 的功能与此类似。当一个进程改变了它的根目录后，所有绝对路径名都将从这个新的根目录开始。那么为什么要这样做呢？答案是为了安全！例如，对于 **FTP**（File Transfer Protocol）和 **HTTP**（HyperText Transfer Protocol）服务器程序，它可以使用 `chroot` 来修改根目录，这样，远程用户就只能访问新的根目录下的文件。只有超级用户才有权限执行 `chroot`，但即使是超级用户，也很少去使用它。

1.4.5 保护的系统调用

在 MINIX 3 中，每一个文件都有一个 11 位的保护模式码，其中的 9 位用来描述文件所有者、同组用户和其他用户的读-写-执行权限。`chmod` 系统调用可以改变文件的保护模式。例如，如果我们要使一个文件对文件所有者之外的所有用户只读，可以调用

```
chmod("file", 0644);
```

保护模式码的另外两位，即 02000 和 04000，分别是 SETGID 位（设置组标识）和 SETUID 位（设置用户标识）。如果用户执行了一个标有 SETUID 位的程序，那么在这个进程运行期间，用户的有效 UID 被临时设置为文件所有者的 UID。这个特性常被用来使一般用户可以去执行一些通常只有超级用户才能执行的功能，如创建目录。创建目录的系统调用是 `mknod`，通常只有超级用户才能使用。但普通用户也需要去创建一个新目录，因此解决的办法就是另外设置一个 `mkdir` 程序，在该程序中去调用 `mknod`。通过把 `mkdir` 程序的所有者设置为超级用户，同时将其保护模式设为 04755，普通用户就可以具有以受限方式执行 `mknod` 的权利。

当进程执行一个标有 SETUID 位或 SETGID 位的文件时，它便获得了一个有效 UID 或 GID，而这个有效 ID 不同于它的真实 ID。对于一个进程来说，它有时需要知道自己的 ID 是什么，包括真实 ID 和有效 ID，这时就可以使用 `getuid` 和 `getgid` 这两个系统调用，前者返回真实 UID 和有效 UID，后者返回真实 GID 和有效 GID。为了使用上的方便，系统对这两个系统调用进行了封装，提供了四个库函数，分别是 `getuid`, `getgid`, `geteuid` 和 `getegid`。前两个返回真实的 UID/GID，后两个返回有效的 UID/GID。

一般用户不能改变他们的 UID，除非执行了一个标有 SETUID 位的程序。但超级用户则不同，他可以使用 `setuid` 系统调用来自设置自己的有效和真实 UID，也可以使用 `setgid` 来设置有效和真实 GID，还可以使用 `chown` 来改变文件的所有者。总之，超级用户不受系统保护规则的约束。这就是为什么有那么多的学生千方百计、乐此不疲地想成为超级用户的原因。

在权限管理方面，还有两个系统调用可以被一般用户进程执行。第一个是 `umask`，它可以设置一个内部掩码，用于文件创建时对指定的保护模式码进行掩蔽。例如，假设执行了

```
umask(022);
```

在此之后，`creat` 和 `mknod` 指定的保护模式码都要与 022 的反码（0755）相与，所得的结果才是最终的保护码。因此，假设有如下的调用：

```
creat("file", 0777);
```

首先要把指定的模式码 0777 与 022 的反码相与，得到的结果为 0755。因此，对于文件 `file`，它最终的保护模式码就是 0755。由于掩码可以被子进程继承，所以如果在登录后 shell 执行了一次 `umask` 操作，那么此次登录期间的所有用户进程都将受到该掩码的约束，即它们所创建的文件都不能被其他用户（包括同组用户）修改。

对于一个所有者为 root 的程序 A，如果它设置了 SETUID 位，那么它可以访问任何文件，因为它的有效 UID 为超级用户。但这样的话就会带来一些安全问题。例如，对于一个普通的用户来说，他本来没有权限去访问某一个文件，但如果他执行了程序 A，就有可能通过它来间接地访问该文件，因为他的有效 UID 已经是超级用户了。因此，对于程序 A 来说，当它要访问一个文件时，需要知道调用它的那个用户是否拥有该文件的访问权限。但如果它只是简单地去尝试一下，看能否打开、访问这个文件，这是没有什么意义的，因为它肯定能够访问该文件。

所以，对于程序 A 来说，它需要知道调用它的用户的真实 UID（而不是有效 UID）是否具有访问该文件的权限。这时，就可以用到 `access` 系统调用。`access` 的 `mode` 参数如果为 4，表示检查读权限；如果为 2，表示检查写权限；如果为 1，表示检查执行权限。此外，还可以把这几个值组合起来使用。例如，如果 `mode` 为 6，那么只有当真实的 ID 用户对文件具有可读可写权限时，才会返回 0（成功），否则返回 -1。当 `mode` 为 0 时，仅检查文件是否存在，同时检查从根目录直到该文件的所有目录是否允许搜索。

对于类 UNIX 操作系统，它们的保护机制基本上是相似的，但也有一些区别和矛盾，从而容易造成一些安全漏洞。详细内容请参阅 Chen et al. (2002)。

1.4.6 时间管理的系统调用

MINIX 3 有 4 个用于时间管理的系统调用。`time` 系统调用返回当前的时间，以秒为单位，从 1970 年 1 月 1 日零时开始计时。`stime` 用来设置当前的系统时间（仅由超级用户执行）。`utime` 允许文件的所有者（或超级用户）来修改存储在文件 i 节点中的时间，例如 `touch` 命令就使用 `utime` 将文件的时间设为当前时间。

最后是 `times` 系统调用，它返回进程的账户信息。例如，进程总共执行了多少 CPU 时间，其中有多少是自己用的，多少是操作系统代表它使用的（即执行系统调用）；它的所有子进程总共执行了多少时间，其中有多少是用户使用的，多少是系统使用的，等等。

1.5 操作系统结构

至此我们已经了解了操作系统的外部特性（即编程接口），现在开始观察其内部的组成结构。在以下几个小节中，我们将讨论五种不同的操作系统体系结构，以便对每种结构的特点有一个初步的了解。这五种结构并不包括一切，但通过对它们的研究我们将对实际操作系统的设计建立一些基本的概念。这五种结构分别是整体结构、分层结构、虚拟机、外核和客户 - 服务器结构。

1.5.1 整体结构

整体结构是最常用的一种组织方式，但常被人们形容为“一锅粥”，它的结构实际上就是“无结构”。整个操作系统是一组函数的集合，其中每个函数在需要的时候可以去调用任何其他的函数。当使用这种技术时，系统中的每个函数都有一个定义完好的接口，包括它的入口参数和返回值，而且相互之间的调用不受任何约束。

在整体式系统中，为了构造最终的目标操作系统程序，开发人员首先将一些独立的函数或文件进行编译，然后用链接程序把它们链接在一起成为一个单独的目标程序。从信息隐藏的观点来看，它没有任何程度的隐藏——所有的函数都是相互可见的（与此相对的是结构化程序设计，这种设计将整个系统划分为若干个模块，信息被隐藏在这些模块的内部，外部程序只能通过一些预先指定的入口点来调用这些模块）。

即使在整体式系统中，也存在一些程度很低的结构化。操作系统提供的服务（系统调用）的请求过程是这样的：先将参数放入预先确定的地方，如寄存器或栈，然后执行一条特殊的陷阱指令，即访管程序调用（supervisor call）指令或内核调用（kernel call）指令。

这条指令将把CPU从用户态切换到内核态，并将控制权交给操作系统（大多数CPU有两种状态，即内核态和用户态。内核态供操作系统使用，在该状态下可以执行所有的指令；用户态供用户程序使用，在该状态下不能执行I/O操作和其他的一些操作）。

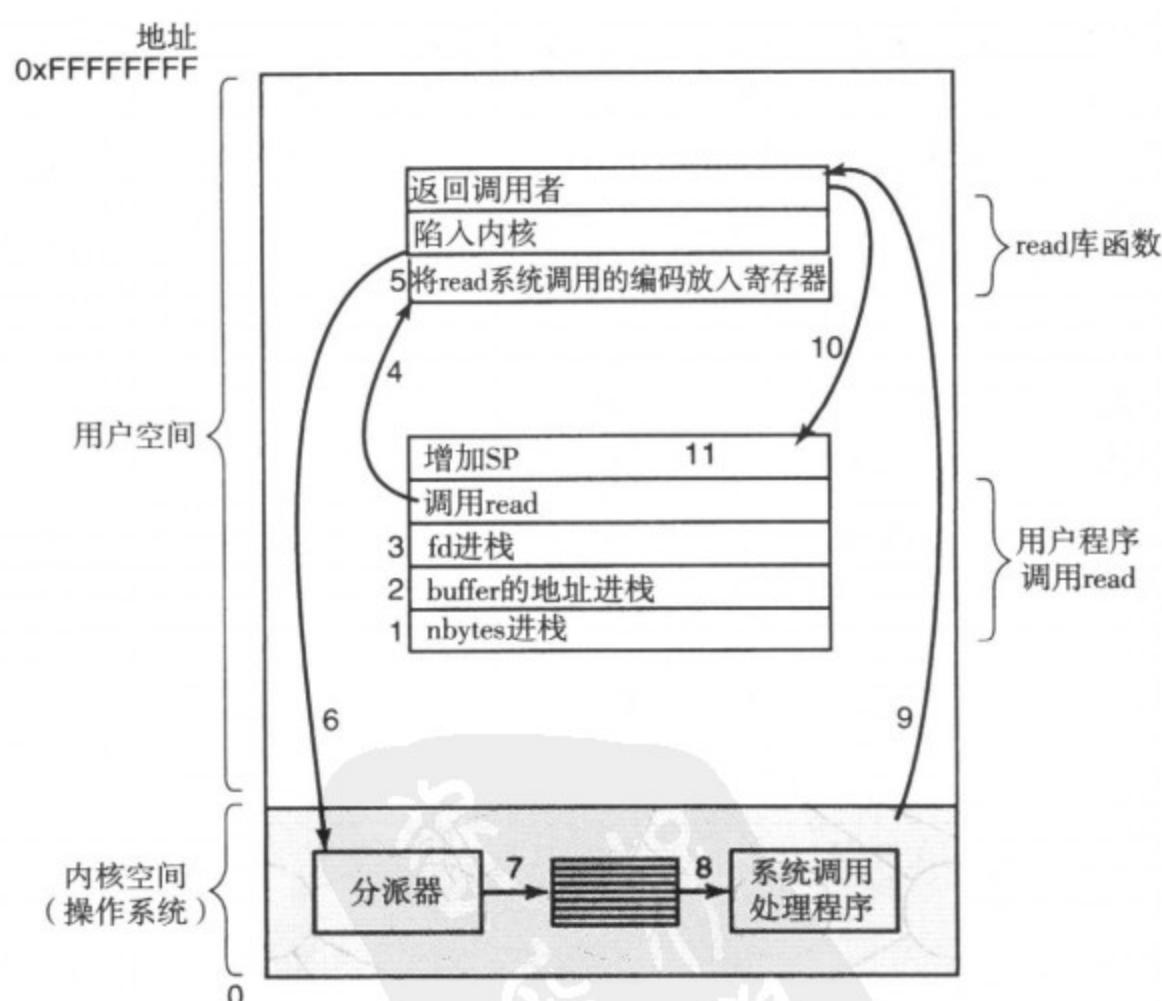


图 1.16 系统调用 `read(fd, buffer, nbytes)` 执行的 11 个步骤

下面我们来看一下系统调用的执行过程。前面我们曾经讲过了 `read` 函数的用法：

```
count = read(fd, &buffer, nbytes);
```

首先，在用户程序中，当它需要去读取文件的时候，就调用 `read` 库函数（由它再去调用 `read` 系统调用）。这样，就把此次函数调用的参数压入栈中，如图 1.16 中的步骤 1~3 所示。可以注意到，C

和 C++ 编译器是按照相反的顺序将各个参数入栈的，这主要是基于历史的原因（与 *printf* 函数有关，它的第一个参数，即格式字符串，必须出现在栈顶）。本次函数调用的第一个和第三个参数都是传值引用，而第二个参数传递的是地址（& 表示取地址运算符）。接下来就是调用相应的库函数（第 4 步），这条指令是一个普通的函数调用指令。

接下来我们就进入了 *read* 库函数，它一般是由汇编语言写的，其主要功能就是把本次系统调用的编号放在操作系统所期待的地方，如某个寄存器（第 5 步）。然后执行一条 TRAP 指令，从用户态切换到内核态，并跳转到内核的某个固定地址开始执行（第 6 步）。这段内核代码将检查此次系统调用的编号，并把它分派到相应的系统调用处理程序。通常的做法是去查询一张系统调用表，每个表项存放了相应的处理程序的起始地址，表项的下标即为系统调用的编号（第 7 步）。这样，我们就跳转到了正确的处理程序并开始运行（第 8 步）。当处理程序完成任务后，控制流可能回到了用户空间中的 *read* 库函数，即 TRAP 指令的下一条指令（第 9 步）。然后，该函数将执行一条普通的函数返回指令，返回到调用它的用户程序（第 10 步）。

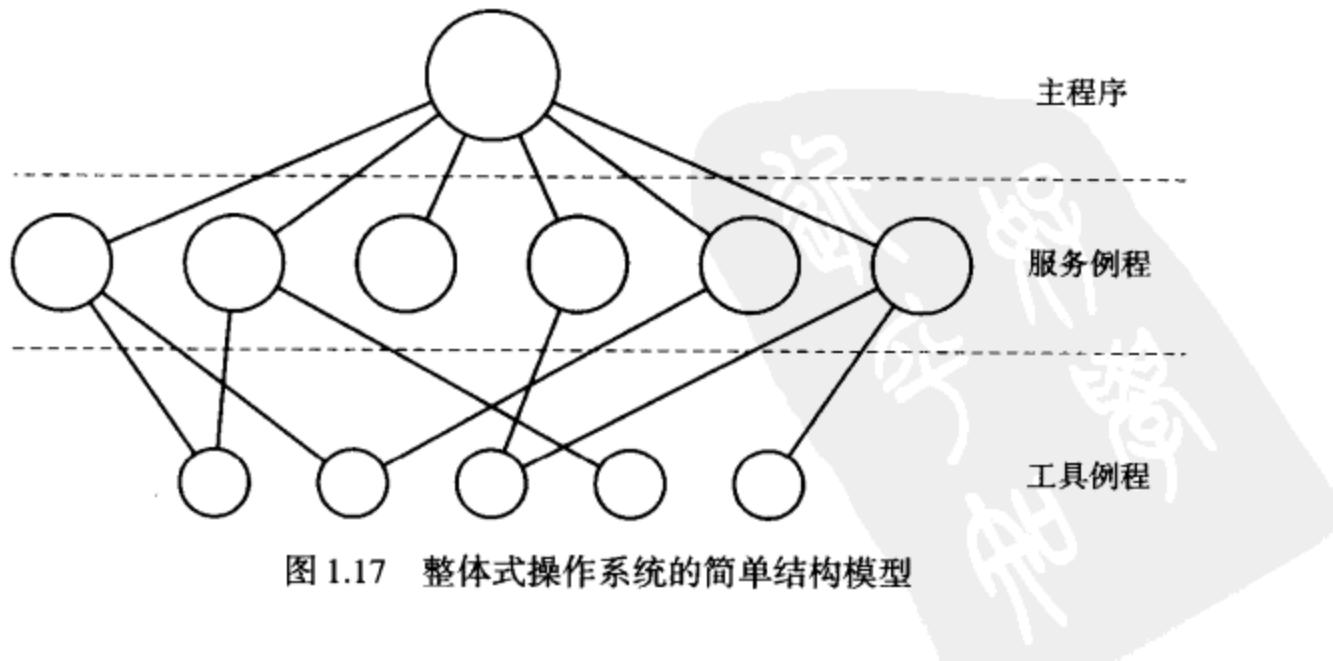
在用户程序中，先要把栈顶的内容清除掉（里面有三个参数，现在已经没什么用了），这也是普通的函数调用在收尾时所做的事情（第 11 步）。假定栈的增长方向向下，代码将增加栈指针的值，使它正好跳过先前入栈的这三个参数。这样，整个的处理过程就结束了，从磁盘文件读入的数据已经保存在 *buffer* 中，用户程序就可以接着去做其他的事情。

在上述的第 9 步中，我们曾经说“控制流可能回到了用户空间中的 *read* 库函数”，我们之所以说“可能”，这是有原因的。因为此次系统调用可能会阻塞调用进程。例如，假设用户程序不是去访问磁盘文件，而是去读取键盘输入，但此时还没有任何字符被键入，这样，系统就会把调用者阻塞起来，然后去看一下有没有其他的进程需要运行。后来，当键盘输入发生后，被阻塞的进程将重新开始运行，从而完成上述的第 9~11 步。

这种组织方式提出了操作系统的一种基本结构：

1. 一个主程序，用来调用被请求的服务例程。
2. 一组服务例程，用来实现相应的系统调用。
3. 一组工具函数，用来帮助服务例程的实现。

在这种模型中，每一个系统调用都由一个服务例程来完成。而工具函数则负责一些辅助性的工作，如从用户程序获取数据。这样的一个三层模型如图 1.17 所示。



1.5.2 分层结构

将图 1.17 所示的方法进行进一步的推广，就得到了分层结构，即把整个操作系统组织成一个层次结构，每一层软件都是在它的下层软件的基础上构造起来的。按照这种模型构造的第一个操作系统是 E. W. Dijkstra 和他的学生在荷兰 Eindhoven 技术学院开发的 THE 系统（1968 年）。THE 系统是为荷兰制造的 Electrologica X8 计算机配备的一个简单的批处理系统，X8 的内存只有 32K 个字，每个字 27 位。

THE 系统分为六层，如图 1.18 所示。第 0 层负责处理器的分配，当发生中断或定时器到期时由该层软件进行进程切换。在第 0 层之上，系统由一序列的进程组成，每个进程在具体实现时，无须考虑进程之间对单一处理器的竞争访问。换句话说，第 0 层掩盖了 CPU 的使用细节，向上提供了基本的多道程序功能。

层次	功能
5	操作员
4	用户程序
3	输入/输出管理
2	操作员 - 进程间通信
1	内存和磁鼓管理
0	处理器分配和多道程序

图 1.18 THE 操作系统的结构

第 1 层负责存储管理，为进程分配内存空间，如果内存不够用，则在 512K 个字的磁鼓上存放进程的部分页面。在第 1 层之上，进程无须再考虑存储问题，它不用担心自己是在内存中还是在磁鼓上，因为第 1 层的软件对此进行了封装，它能够保证当进程需要访问某一页时，该页面肯定在内存中。

第 2 层负责处理进程与操作员控制台之间的通信。在这一层之上，可以认为每个进程都有它自己的操作员控制台。第 3 层负责管理 I/O 设备和相关的信息流缓冲区。在第 3 层之上，每个进程都能与性能良好的抽象 I/O 设备打交道，而不必考虑物理设备的具体细节。第 4 层是用户程序，它们无须考虑进程管理、存储管理、控制台和 I/O 管理等环节。系统操作员进程位于第 5 层。

MULTICS 对层次化概念进行了进一步的推广，它没有采用层次结构，而是将系统组织成一系列的同心圆环，内层的环比外层的环具有更高的特权级。当外层环的一个函数想要调用内层环中的某个函数时，它必须执行一条类似于系统调用的 TRAP 指令，并对调用参数进行严格的合法性检查。在 MULTICS 中，操作系统出现在每个用户进程的地址空间中。为确保安全，在硬件上存在着相应的保护机制，能够对单个例程（实际上是内存中的段）的读、写和执行权限进行保护。

THE 的分层方案实际上只是在设计上提供了一些方便，系统的各个部分最终仍然被链接成一个完整且单一的目标程序。但是在 MULTICS 中，上述的环形方案在系统运行时是实际存在的，并且由硬件来保障。环形方案的优点是易于扩展，便于构造用户子系统。例如，在一个系统中，教授可以写一个程序来检查学生编写的程序并打分，为了安全起见，他可以将自己的程序放在第 n 个环中运行，而将学生的程序放在第 $n+1$ 个环中运行，这样学生就无法篡改教授给出的成绩。奔腾硬件支持 MULTICS 的环形结构，但目前的主流操作系统都没有用到这一点。

1.5.3 虚拟机

IBM OS/360 的最早版本是纯粹的批处理系统，然而许多 360 的用户希望使用分时系统，于是公司内外的一些研究小组决定开发一些分时系统。IBM 官方研制的分时系统 TSS/360，一直未如期

推出。当它终于发布时，却因为规模庞大、运行缓慢，几乎没有什么人用。在耗费了约 5000 万美元的研制费用后，该系统最终还是被放弃了（Graham, 1970）。但 IBM 设在麻省剑桥的一个研究中心开发了一个完全不同的系统，最终被 IBM 作为产品。目前该系统仍然在 IBM 的大型主机上广泛应用。

该系统最初称为 CP/CMS，后来更名为 VM/370（Seawright and MacKinnon, 1979）。它基于如下的观察，即一个分时系统应该提供两个特性：(1)多道程序；(2)一个扩展机，它提供了比裸机更方便的编程接口。VM/370 的实质就是将此二者完全分离开来。

系统的核心是一个虚拟机监控程序，它在裸机上运行并具备多道程序功能。它向上层提供了若干台虚拟机，如图 1.19 所示。与其他操作系统不同，这些虚拟机不是那种具有文件等良好特性的扩展机，而仅仅是裸机硬件的精确拷贝，包含有内核态/用户态、I/O 功能、中断以及真实硬件所具有的所有内容。

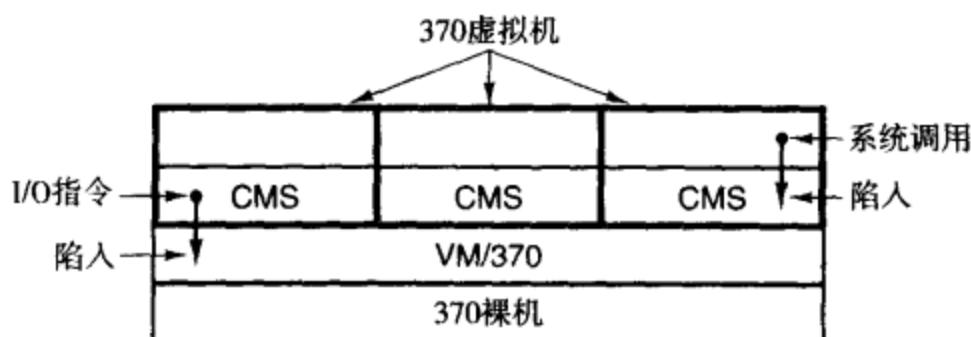


图 1.19 带 CMS 的 VM/370 结构

由于每台虚拟机都与裸机完全一样，所以在每台虚拟机上都可以运行任何一种操作系统。一般来说，在不同的虚拟机上运行的是不同的操作系统。例如，有些虚拟机运行的是 OS/360 的某个后续版本，从事一些批处理或事务处理的工作；而另一些虚拟机则为分时用户运行一个单用户、交互式的系统 CMS（Conversational Monitor System，会话监控系统）。

当一个 CMS 程序执行一条系统调用时，该调用将陷入到其虚拟机的操作系统，而不是 VM/370，这就像在真正的计算机上一样。然后 CMS 发出正常的硬件 I/O 指令来读取它的虚拟磁盘或者做其他的事情来完成此次系统调用。这些 I/O 指令被 VM/370 捕获，随后 VM/370 执行这些指令，作为对真实硬件模拟的一部分。通过将多道程序功能和虚拟机功能分开，每一部分都能更简单、更灵活并易于维护。

虚拟机的思想如今已被广泛采用，例如，在奔腾 CPU 上运行老的 MS-DOS 程序。在设计奔腾芯片的硬件和软件时，Intel 和 Microsoft 都意识到要让老软件能够在新硬件上运行，于是 Intel 在奔腾芯片上提供了一个虚拟 8086 模式。在此模式下，奔腾机就像一台 8086 计算机一样，包括 1 MB 内存、16 位寻址方式。

虚拟 8086 模式被 Windows 和其他操作系统用于运行老的 MS-DOS 程序。这些程序在虚拟 8086 模式下启动，如果它们执行的是一般的指令，那么就直接在裸机上运行。但是，如果程序试图陷入操作系统，请求一个系统调用，或者试图直接去访问 I/O 设备，这时就会陷入到虚拟机监控程序。

在这种情形下，有两种设计方法。第一种方法是，MS-DOS 本身被装入虚拟 8086 模式的地址空间，虚拟机仅仅将该陷入传回给 MS-DOS，就像在真正的 8086 上发生的过程一样。后来当 MS-DOS 试图自行执行 I/O 操作时，该操作将被虚拟机监控程序捕获并由它来完成。

第二种方法是虚拟机监控程序仅仅捕获第一条陷入，然后自己执行 I/O 操作。因为它知道所有的 MS-DOS 系统调用，并由此知道每条陷入需要做的事情是什么。这种方法不如第一种方法纯，因为它仅仅正确地模拟了 MS-DOS，而不能模拟其他操作系统。相比之下，第一种方法可以正确地模

拟其他的操作系统。但另一方面，这种方法的速度很快，因为它不再需要启动 MS-DOS 来执行 I/O 操作。另外，在虚拟 8086 模式中运行 MS-DOS 的另一个缺点是，MS-DOS 需要频繁地对中断屏蔽位进行操作，而模拟这些操作是很费时的。

需要指出的是，上述这两种方法都不同于 VM/370，因为它们模拟的并不是完整的奔腾硬件而只是一个 8086。在 VM/370 系统中，可以在虚拟机上运行 VM/370 本身，而在奔腾系统中，不可能在虚拟 8086 上运行 Windows，因为 Windows 不能在 8086 芯片上运行。其最低版本也需要在 80286 上运行，而奔腾芯片不提供对 80286 的模拟。

在市面上已经有一些商品化的虚拟机产品。对于一个提供 Web 主机服务的公司，它可以有两种做法，一是在一个非常快的服务器上（该服务器可能有多个 CPU）运行多个虚拟机，每个虚拟机作为一个 Web 站点的主机；二是维护许多台小的计算机，每台计算机作为一个 Web 站点的主机。显然，第一种方式比第二种方式要经济得多。VMWare 和微软的 Virtual PC 都是一些商品化的虚拟机。这些程序使用主机系统上的一个大文件来作为它们模拟的磁盘。为了提高系统的性能，它们往往需要去分析目标操作系统的可执行代码。这种系统在教学领域也很有用，例如，选修了 MINIX 3 课程的学生，可以在一台 Windows、Linux 或 UNIX 主机上安装 VMWare 虚拟机，然后在该虚拟机上运行 MINIX 3 操作系统，并完成相应的课后练习。这样就不用担心由于新的操作系统的安装，对 PC 机上的其他软件和数据造成破坏。事实上，许多讲授其他课程的教授，非常不愿意和操作系统课程共享实验室的机器，因为操作系统课程的实验往往比较“危险”，稍不小心就可能把磁盘上的数据给毁掉了。

虚拟机的另一个应用领域是 Java 虚拟机。当 Sun 公司发明 Java 语言的时候，它同时发明了一个名为 JVM（Java Virtual Machine）的虚拟机（一种计算机体系结构）。Java 编译器将输入的源程序编译成 JVM 代码，然后就可以在一个 JVM 解释器上运行。JVM 解释器既可以用软件实现，也可以用硬件来实现。这种方法的优点是，生成的 JVM 代码可以通过 Internet 发送到任何一台装有 JVM 解释器的计算机，并在那里运行。反之，如果编译器生成的是 SPARC 或奔腾的二进制代码，那么这种代码就没有那么好的平台独立性。另一个优点是，如果 JVM 解释器的实现方法得当，那么它可以检查输入的 JVM 程序的安全性，并在一种保护环境下执行这些程序，这样就可以防止这些程序偷走数据或对主机造成任何破坏。

1.5.4 外核

在 VM/370 中，每个用户进程获得真实计算机的一个精确副本。在奔腾芯片的虚拟 8086 模式中，每个用户进程获得的是另一套硬件（8086）的精确副本。由此再进一步，MIT 的研究人员构造了一个系统，每个用户都可以获得真实计算机的一个副本，但只能占用部分系统资源（Engler et al., 1995; Leschke, 2004）。例如，一台虚拟机可能占用第 0 个到第 1023 个磁盘块，而另一台虚拟机可能占用第 1024 个到第 2047 个磁盘块，等等。

在内核态下运行的最底层软件是一个称为外核（exokernel）的程序，其任务是为虚拟机分配资源并确保资源的使用不会发生冲突。每台用户层的虚拟机可以运行它自己的操作系统，就像 VM/370 和奔腾的虚拟 8086 一样，不同之处在于它们各自只能使用分配给它的那部分资源。

外核方案的优点在于它省去了一个映射层。在其他设计方案中，每台虚拟机认为它有自己独立的磁盘，磁盘块的编号从 0 到最大，因此虚拟机监控程序必须维护一些表格来完成磁盘地址以及其他资源的重映射。有了外核之后，这种重映射就不再需要了。外核只需记录每台虚拟机被分配了哪些资源。这种方法的另一个好处是以较少的开销将多道程序（在外核中）与用户操作系统代码（在用户空间中）分离开来，因为外核需要做的工作仅是使各个虚拟机互不干扰。

1.5.5 客户-服务器模型

VM/370 将传统操作系统的大部分代码（用于扩展机的实现）分离出来，放在更高的层次上，即 CMS，由此使系统得以简化。但 VM/370 本身仍然非常复杂，因为模拟许多虚拟的 370 硬件不是一件简单的事情（尤其还有性能的问题）。

现代操作系统的一个趋势是将这种把代码移到更高层次的思想进一步发展，从操作系统中去掉尽可能多的东西，只留下一个最小的内核。通常的方法是将大多数的操作系统功能由用户进程来实现。为了获得某项服务，如读取文件的某个数据块，用户进程（现在称客户进程，client process）将此请求发送给一个服务器进程（server process），服务器进程随后完成此次操作并将应答信息送回。

该模型如图 1.20 所示，内核的全部工作就是处理客户与服务器间的通信。操作系统被分割成许多部分，每一部分只处理一方面的功能，如文件服务、进程服务、终端服务或内存服务。这样，每一部分便变得更小、更易于管理。而且，由于所有的服务器都是以用户进程的形式运行的，而不是运行在内核态，所以它们不直接访问硬件。这样处理的结果是，假如在文件服务器中发生错误，那么文件服务可能崩溃，但不会导致整个系统的崩溃。

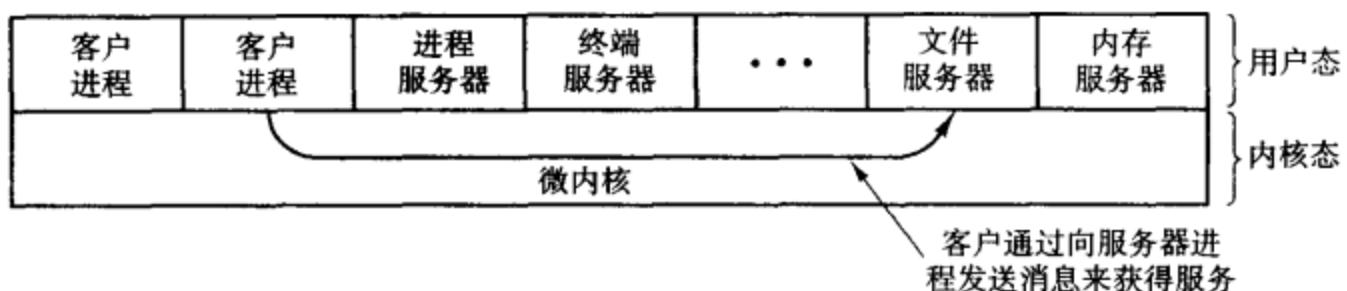


图 1.20 客户-服务器模型

客户-服务器模型的另一个优点是它适用于分布式系统（见图 1.21）。如果一个客户通过发送消息与服务器通信，那么客户无须知道这条消息是在本机就地处理还是通过网络送给远地机器上的服务器。对于客户而言，无论是哪一种情形，结果都是一样的，即客户发出一个请求，然后收到一个应答。

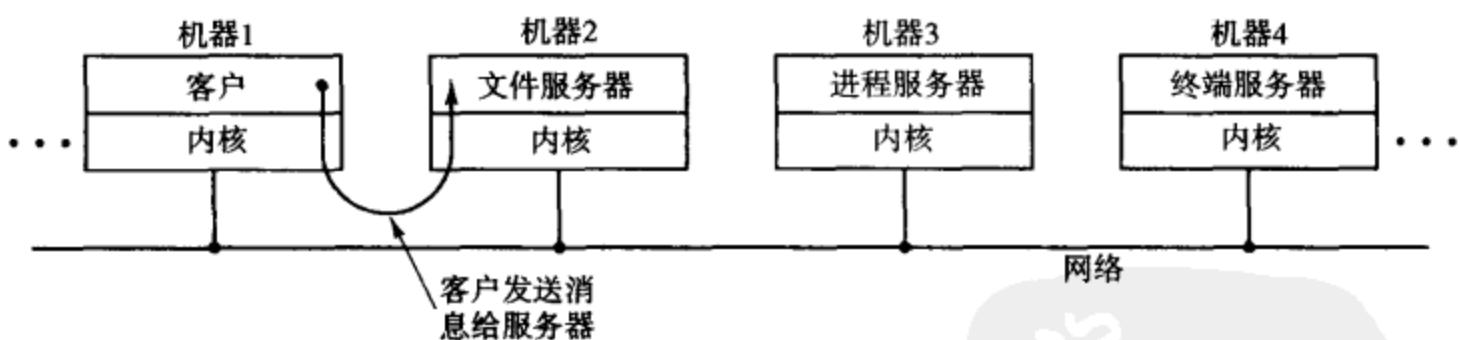


图 1.21 一个分布式系统中的客户-服务器模型

图 1.21 所描绘的内核只处理客户与服务器之间的消息传递，但实际的系统并不完全如此。有些操作系统功能（如向物理 I/O 设备寄存器写入命令字）靠用户空间的程序是很难完成的。解决这个问题的方法有两种：一是设立一些运行于内核态的专用服务器进程（如 I/O 设备驱动程序），它们能访问所有的硬件设备，但仍然通过平常的消息机制与其他进程通信。在 MINIX 的早期版本中，使用了类似的机制，即驱动程序被编译进内核，但是作为一个单独的进程运行。

另一种方法是在内核中建立一套最小的机制（mechanism），而将策略（policy）留给用户空间中的服务器进程。例如，内核可能将发往某个特定地址的消息理解为：取出该消息的内容，并将其装入某个磁盘的 I/O 设备寄存器，从而启动一次读盘操作。在这个例子中，内核甚至不对消息的内

容进行合法性检查，而只是将它们机械地复制到磁盘设备寄存器（显然，必须使用某种方案，将此类消息的发送限定在一些授权的进程中）。MINIX 3采用的就是这种方案，驱动程序运行在用户空间，然后通过特殊的内核调用来请求读、写 I/O 寄存器，或是访问内核信息。机制与策略分离是一个重要的概念，它在操作系统的许多地方经常用到。

1.6 剩余各章内容简介

操作系统一般由四大部分组成：进程管理、I/O 设备管理、存储管理和文件管理。MINIX 3也分为这四个部分。第 2 章到第 5 章将分别讨论这四个主题，第 6 章列出了阅读材料和参考文献。

进程管理、I/O 管理、存储管理和文件系统这几章的组织结构大致相同，首先阐述该主题的基本原理，然后介绍 MINIX 3 中的相应内容（这些内容同样适用于 UNIX）。最后详细地讨论 MINIX 3 中的具体实现。如果读者只是对操作系统的基本原理感兴趣，那么可以略过实现部分的内容，这不会造成任何的不连贯。但是，如果读者希望了解一个真实的操作系统（MINIX 3）是如何工作的，就应该阅读所有的章节。

1.7 小结

我们可以从两种观点来看待操作系统：资源管理器的观点和扩展机的观点。从资源管理器的观点来看，操作系统的任务是高效地管理整个系统的各个部分；从扩展机的观点来看，其任务是为用户提供一台比物理计算机更易于使用的虚拟计算机。

操作系统的功能很强大，从早期的代替操作员手工操作的系统，一直到现在的多道程序系统。

操作系统的功能很强大，从早期的代替操作员手工操作的系统，一直到现在的多道程序系统。操作系统的功能很强大，从早期的代替操作员手工操作的系统，一直到现在的多道程序系统。对于 MINIX 3，它的系统调用分为六大类。第一类与进程的创建和终止有关；第二类处理信号；第三类针对文件读写；第四类进行目录管理；第五类对信息进行保护；第六类用于时间管理。

操作系统有若干种构造方式，包括整体结构、分层结构、虚拟机、外核和客户 - 服务器模型。

习题

1. 操作系统的两个主要功能是什么？
2. 内核态和用户态的区别是什么？对于一个操作系统来说，这种区别为什么很重要？
3. 什么是多道程序？
4. 什么是假脱机？你认为将来的高档个人计算机会将假脱机作为标准特性吗？
5. 在早期的计算机中，每一个字节数据的读写都是由 CPU 直接进行处理的（那时没有 DMA，即直接存储器访问）。这种组织结构对多道程序技术有什么影响？
6. 为什么分时系统未被第二代计算机广泛采用？
7. 下列哪种指令只能在内核态下执行？
 - (1) 屏蔽所有中断。
 - (2) 读时钟日期。
 - (3) 设置时钟日期。
 - (4) 改变内存映像图。
8. 请指出个人计算机操作系统与大型主机操作系统的不同之处。

9. 给出一个理由，说明为什么源代码不公开的商业操作系统（如 Windows）要比一个开放源代码的操作系统（如 Linux）具有更好的品质。然后再给出一个理由，说明为什么开放源代码的操作系统（如 Linux）要比一个源代码不公开的商业操作系统（如 Windows）具有更好的品质。
10. 一个 MINIX 文件的所有者的 UID = 12, GID = 1, 该文件的权限模式码为 *rwxr-x--*。另一个用户的 UID = 6, GID = 1, 如果他试图去执行该文件，结果会如何？
11. 既然超级用户的存在将导致许多安全问题，为什么还要使用这个概念？
12. UNIX 的所有版本都支持文件的两种命名方式，即绝对路径名（相对于根目录）和相对路径名（相对于当前工作目录）。能否去掉其中的一种，只使用一种统一的命名方式？如果要这样做，你建议保留哪一种？
13. 在分时系统中为什么需要进程表？假设在一个 PC 机系统中，只有一个进程存在，该进程占有整个的系统资源，直至它运行结束。在这种情形下，是否还需要进程表？
14. 块设备文件和字符设备文件的本质区别是什么？
15. 在 MINIX 3 系统中，用户 2 对用户 1 的一个文件建立了一个链接，然后用户 1 删除了此文件。如果此时用户 2 去访问该文件，结果会如何？
16. 管道是一种必备的工具吗？如果没有了管道，系统的许多功能会失效吗？
17. 现代消费电子产品，如立体声音响和数码相机，通常会有一个显示屏幕，可以输入命令并显示相应的结果。在这些设备内部有一个原始的操作系统。这种命令处理功能与个人计算机软件上的哪一部分是类似的？
18. Windows 操作系统没有 fork 系统调用，但它也能创建新进程。请你猜测一下，Windows 用来创建一个新进程的系统调用的语义是什么。
19. 为什么 chroot 系统调用仅限于超级用户使用？（提示：信息保护问题。）
20. 分析一下图 1.9 中的系统调用列表。你认为执行速度最快的系统调用是哪一个？为什么？
21. 假设一台计算机的执行速度为 10 亿条指令 / 秒，一个系统调用需要 1000 条指令，包括陷阱指令和所有的上下文切换指令。请问，在保留一半的 CPU 容量来运行应用程序的情形下，该系统每秒钟能执行多少次系统调用？
22. 在图 1.16 中有一个 mknod 系统调用，但却没有 rmnod 调用。这是否意味着我们在创建新节点时必须非常小心，因为没有办法去删掉它们。
23. 在 MINIX 3 中为什么要设立一个在后台始终不停地运行的 update 程序？
24. 在什么情形下忽略 SIGALRM 信号是有意义的？
25. 客户 - 服务器模型在分布式系统中很流行。它能用在单机系统中吗？
26. Pentium 处理器的初始版本不支持虚拟机监控程序。一台机器需要具备什么样的本质特征才能使之虚拟化？
27. 编写一个程序或一组程序，来测试所有的 MINIX 3 系统调用。对每一个系统调用，尝试使用不同的参数，包括错误的参数，以验证它们能否被检测出来。
28. 编写一个类似于图 1.10 的 shell，要求其中包括足够多的代码，使之能正常地工作，这样才能对它进行测试。可以加入一些特性，如输入输出的重定向、管道和后台作业等。

第2章 进 程

2.1 进程介绍
2.2 进程间通信
2.3 经典 IPC 问题
2.4 进程调度
2.5 MINIX 3 进程概述
2.6 MINIX 3 中进程的实现
2.7 MINIX 3 的系统任务
2.8 MINIX 3 的时钟任务
2.9 小结

本章将详细阐述操作系统的一般设计和构造方法，以及 MINIX 3 操作系统的特定设计与构造。进程是操作系统中最核心的概念：它是对正在运行程序的一个抽象。操作系统的其他部分都围绕此概念展开，所以操作系统的设计师（及学生）应该很好地理解进程。

2.1 进程介绍

所有现代的计算机都能同时做几件事情。当一个用户程序正在运行时，计算机还能够同时读取磁盘，并向屏幕或打印机输出文本信息。在一个多道程序系统中，CPU 在程序间切换，使每道程序运行几十或几百毫秒。然而严格地说，在一个瞬间，CPU 只能运行一道程序。在 1 s 期间，它可能运行了多道程序，这样就给用户一种并行的错觉。有时人们所说的伪并行（*pseudoparallelism*）就是这种情形，以此来区分多处理机（multiprocessor）系统的真正的硬件并行（该系统有两个或多个 CPU 共享同一物理内存）。人们很难对多个并行的活动进行跟踪。因此，经过多年的努力，操作系统的设计师发展了一种概念模型（顺序进程，*sequential processes*），使得并行更容易处理。有关该模型、它的使用及其影响正是本章的主题。

2.1.1 进程模型

在该模型中，计算机上所有可运行的软件，通常包括操作系统，被组织成若干顺序进程，简称进程（processes）。一个进程就是一个正在执行的程序，包括程序计数器、寄存器和变量的当前值。从概念上说，每个进程都拥有它自己的虚拟 CPU。当然，实际上真正的 CPU 在各进程之间来回切换。为了理解这种系统，考虑在（伪）并行情况下运行的进程集，要比试图跟踪 CPU 如何在程序间来回切换简单得多。正如在第 1 章所示，这种快速的切换称为多道程序设计（multiprogramming）。

在图 2.1(a)中，可以看到一个计算机在内存中多道运行 4 个程序。在图 2.1(b)中，可看到 4 个进程各自拥有自己的控制流程（即自己的程序计数器），并且每个都独立地运行。当然，实际上只有一个物理程序计数器，所以在每个程序运行时，它的逻辑程序计数器被装入实际的程序计数器中。随着时间的推移，当程序结束时，物理程序计数器被保存到内存中该进程的逻辑程序计数器中。在图 2.1(c)中，在观察一段足够长的时间后，所有的进程都有所进展，但在一个给定的瞬间仅有一个进程真正在运行。

由于 CPU 在各进程之间来回切换，每个进程执行运算的速度是不确定的，而且当同一进程再次运行时，其运算速度通常也不可再现，所以在对进程编程时绝不能对时序做任何固定的假设。例如，考虑一个 I/O 进程用流式磁带机恢复已备份的文件，它执行一个 10 000 次的空循环以等待磁带机达到正常速度，然后发出命令读取第 1 个记录。如果 CPU 决定在空循环期间将处理机调度给其他进程，则磁带机进程可能在第 1 条记录通过磁头之后还未被再次调度。当一个进程具有此类严格的实时要求时，也就是一些特定事件一定要在所指定的若干毫秒中发生时，那么必须采取特殊措施

来保证它们一定在这段时间中发生。然而，通常大多数进程并不受 CPU 多道程序或其他进程相对速度的影响。

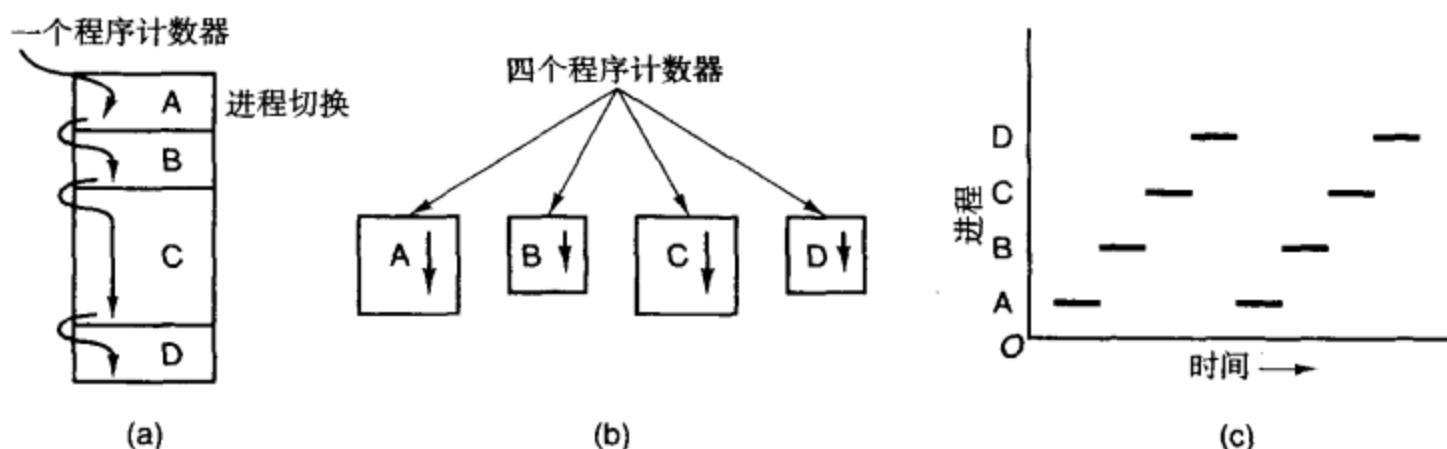


图 2.1 (a) 4 个程序的多道程序设计; (b) 4 个独立的顺序进程的概念模型; (c) 在任意瞬间只有一个程序处于活跃状态

进程和程序之间的区别是很微妙的，但却至关重要。一个类比可以使读者更容易理解这一点。想象一位有一手好厨艺的计算机科学家正在为他的女儿烘制生日蛋糕。他有制作生日蛋糕的食谱，厨房里有所需的原料：面粉、鸡蛋、糖、香草汁等。在这个比喻中，做蛋糕的食谱就是程序（即用适当形式描述的算法），计算机科学家就是处理器（CPU），而做蛋糕的各种原料就是输入数据。进程就是厨师阅读食谱、取来各种原料以及烘制蛋糕的一系列动作的总和。

现在假设计算机科学家的儿子哭着跑了进来，说他被一只蜜蜂蛰了。计算机科学家就记录下自己照着食谱做到哪儿了（保存进程的当前状态），然后拿出一本急救手册，按照其中的指示处理蛰伤。这里可看到处理器从一个进程（做蛋糕）切换到另一个高优先级的进程（实施医疗救治），每个进程拥有各自的程序（食谱和急救手册）。当蜜蜂蛰伤处理完之后，计算机科学家又回来做蛋糕，从他离开时的那一步继续做下去。

这里的关键思想是：一个进程是某种类型的一个活动，它有程序、输入、输出及状态。单个处理机被若干进程共享，它使用某种调度算法决定何时停止一个进程的工作，并转而为另一个进程提供服务。

2.1.2 进程的创建

操作系统需要一种方式来确定所有必需的进程都存在。在一些非常简单的系统中，或在那种只为运行一个应用程序而设计的系统中（如实时地控制一个设备），在系统启动时，以后需要的所有进程都存在是可能的。但是在通用系统中，需要某种方法在运行时按照需要来创建和终止进程。下面开始探讨这一问题。

进程的创建有四个主要的原因：

1. 系统初始化。
2. 正在运行的一个进程执行了创建进程的系统调用。
3. 用户请求创建一个新进程。
4. 批处理作业的初始化。

启动操作系统时，通常会创建若干个进程。其中一些是前台进程，也就是同用户（人类）交互并替他们完成工作的那些进程。另外一些是后台进程，它们不与特定的用户相联系，而是具有某些专用的功能。例如，一个后台进程可能用来接收请求访问这台机器上网页的请求消息，在请求消息

到来时就会醒来并处理这个请求。这些处于后台用来处理网页、打印之类活动的进程称为**守护进程**(daemon)。在大型系统中，通常会有很多守护进程。在MINIX 3中，*ps*程序可以用来显示当前运行的所有进程。

除了在系统启动阶段可创建进程以外，在启动之后也可以创建进程。通常一个正在运行的进程会发出系统调用创建一个或多个进程协助其完成工作。若所从事的工作可以很容易地划分成若干个相关但未相互作用的进程，则创建新的进程特别有效。例如，当编译一个大程序时，*make*命令调用C编译器把源文件转换成目标代码，然后调用*install*命令将程序复制到目标地址，设置所有者和访问权限等。在MINIX 3中，C编译器实际上是一些在一起工作的不同程序集合，其中包括预处理器、C语言解析器、汇编语言代码生成器、汇编器、链接器等。

在交互式系统中，用户可以通过键入命令启动程序。在MINIX 3中，虚拟终端允许用户启动一个程序，例如一个编译器，在编译器运行的时候，可以切换到另外一个终端启动一个新的程序，例如去编辑一个文档。

最后一种创建进程的情形仅在大型机的批处理系统中应用。用户在这种系统中（可能是远程地）提交批处理作业。在操作系统认为有资源运行另一个作业时，它创建一个新的进程，并运行其输入队列中的一个作业。

从技术上看，在所有这些情形中，新进程都是由于一个已经存在的进程执行了进程创建的系统调用而创建的。这个进程可以是一个运行的用户进程、一个由键盘或鼠标启动的系统进程或一个批处理管理进程。这个进程所做的工作是，执行一个用来创建新进程的系统调用。这个系统调用通知操作系统创建一个新进程，并且直接或间接地指定在该进程中运行的程序。

在MINIX 3系统中，只有一个系统调用用来创建新的进程：*fork*。这个系统调用会创建一个与调用进程相同的副本。在调用了*fork*后，这两个进程（父进程和子进程）拥有相同的内存映像、相同的环境字符串和相同的打开的文件。这就是所有的情形。通常子进程会执行一个*execve*或一个类似的系统调用，以修改其存储映像并运行一个新的程序。例如，当一个用户在*shell*中键入*sort*命令时，*shell*就创建一个子进程，这个子进程运行*sort*命令。之所以要安排两个步骤来创建进程，是为了在*fork*和*execve*命令之间，完成对标准输入文件、标准输出文件和标准出错文件重定向等文件描述符操作。

在MINIX 3和UNIX中，在进程创建之后，父进程和子进程拥有各自独立的地址空间。如果其中某个进程在其地址空间修改了一个字，则这一改变对另一进程是不可见的。子进程的初始地址空间是父进程地址空间的一个副本，但是这是两个不同的地址空间。可写的地址空间是不共享的（同其他的UNIX实现一样，MINIX 3可以在父子进程间共享代码段，因为它是不可修改的）。但是，一个新建的进程有可能共享像打开文件之类的资源。

2.1.3 进程的终止

进程在创建之后，就开始运行并执行其工作。但永恒是不存在的，进程也一样。迟早这一进程将会终止，通常可能是由于下列原因：

1. 正常退出（自愿）。
2. 出错退出（自愿）。
3. 严重错误（非自愿）。
4. 被其他进程杀死（非自愿）。

多数进程由于完成了它们的工作而终止。当编译器完成了给定程序的编译之后，它执行一个系统调用，通知操作系统它的工作已经完成。在 MINIX 3 中该系统调用是 `exit`。面向屏幕的程序也支持自愿退出。例如，编辑器通常提供一个键组合，用户用来通知进程保存工作文件，删除它打开的所有临时文件，然后退出。

进程终止的第二个原因是进程发现了严重错误。例如，如果用户键入命令

```
cc foo.c
```

来编译程序 `foo.c`，但是这个文件不存在，那么编译器就会简单退出。

进程终止的第三个原因是由于进程引起的错误，通常是由于程序中的错误所致。例如，执行了一条非法指令，引用了不存在的内存，或除数是零，等等。在 MINIX 3 中，进程可以告诉操作系统，它希望自行处理某些类型的错误，在这类错误中，进程会收到信号（被中断），而不是在这类错误出现时而终止。

第四种终止进程的原因是，某个进程执行了系统调用通知系统杀死某个进程。在 MINIX 3 中，这一系统调用是 `kill`。当然，“杀手”需要得到确定的授权才能进行杀死操作。在一些系统中，当进程终止时，不论是否是自愿的，由该进程所创建的所有进程也都立即被杀死。不过，MINIX 3 中没有采用这种工作方式。

2.1.4 进程的层次结构

在某些系统中，当进程创建了另一个进程后，父进程和子进程就以某种形式继续保持关联。子进程自身可以创建更多的进程，组成一个进程的层次结构。与动物和植物的有性繁殖不同，进程只有一个父进程（但可以有零个、一个、两个或多个子进程）。

在 MINIX 3 中，进程和它的所有子进程及其后裔共同组成一个进程组。当用户从键盘发出一个信号时，该信号被送给当前与键盘相关的进程组中的所有成员（它们通常是当前窗口所创建的所有进程）。这个过程是信号相关的。如果一个信号发送给了一个进程组，那么每个进程就可以捕获该信号、忽略该信号或采取默认的动作，即被该信号杀死。

作为如何使用进程树的一个简单例子，我们来看看 MINIX 3 在启动时是怎样对自己进行初始化的。在引导映像中有两个特殊的进程：**再生服务器**（reincarnation server）和 **init**。再生服务器进程的任务是启动或重启驱动（drivers）和服务器（server）。它初始时处于阻塞态，等待消息告诉它创建什么。

与此相对应，`init` 进程执行 `/etc/rc` 脚本，并向再生服务器发送命令启动引导映像中不存在的驱动和服务器。这个过程使所有的驱动和服务器进程都是再生服务器进程的子进程，因此，如果其中某些进程终止的话，再生服务器进程将会收到通知，并重新启动它们。通过这一机制，MINIX 3 操作系统能够允许进程崩溃，因为一个新的进程将被重新启动。实际上，替换一个驱动比替换一个服务器要容易得多，因为这将在系统中引起较小的反响（这一机制并不总是工作得很好，它还处于改进阶段）。

当 `init` 进程完成这些之后，它读取配置文件 `/etc/ttymtab` 中的内容来查看存在哪个终端和虚拟终端。`init` 进程为每个终端产生（`fork`）一个 `getty` 进程，显示登录提示符，然后等待输入。当一个名字输入之后，`getty` 以这个名字为参数运行（`exec`）一个 `login` 进程。如果用户登录成功，`login` 进程将运行用户的 `shell` 进程。所以 `shell` 是 `init` 进程的一个子进程。用户命令创建 `shell` 的子进程，即 `init` 的孙进程。这一系列的事件是进程树如何使用的例子。再生服务器进程和 `init` 进程的代码已经超出了本书的范围，不在此列出。`shell` 的代码也不在此论述。读者可以在其他地方找到这些代码，但是现在读者对进程树应有一个基本的概念。

2.1.5 进程的状态

尽管每个进程是一个独立的实体，有它自己的程序计数器、堆栈、打开的文件、定时器(alarms)和内部状态，但进程之间经常需要交互、通信以及和其他进程同步。一个进程的输出结果可能作为另一个进程的输入。在这种情况下，数据需要在进程间交换，在shell命令

```
cat chapter1 chapter2 chapter3 | grep tree
```

中，第一个进程运行`cat`，将三个文件链接并输出。第二个进程运行`grep`，它从输入中选择所有包含单词“tree”的那些行。根据这两个进程的相对速度（具体取决于这两个程序的相对复杂度和各自所分配到的CPU时间），`grep`可能已做好了可以运行的准备，但还没有输入，于是它就必须被阻塞直到输入到来。

当一个进程在逻辑上不能继续运行时，就会被阻塞，典型的例子是它在等待可以使用的输入。还可能有这样的情况，即一个概念上能够运行的进程被迫停止，其原因是操作系统调度另一个进程占用处理机。这两种条件是完全不同的。第一种情况下，挂起是程序自身所固有的（在用户命令行被键入之前，我们无法执行它）；第二种情况则是由系统引起的（没有足够的CPU，所以不能使每个进程都有一台它私用的处理机）。在图2.2中，可看到进程三种状态的转换图。这三种状态是

1. 运行态 (Running，在该时刻实际占用处理机)。
2. 就绪态 (Ready，可运行，因为其他进程正在运行而暂时被挂起)。
3. 阻塞态 (Blocked，除非某种外部事件发生，否则不能运行)。

前两种状态在逻辑上很类似。这两种状态下的进程都可以运行，只是在后者中，暂时没有CPU分配给它。第三种状态与前两种状态不同，该状态的进程不能运行，即使CPU空闲也不行。

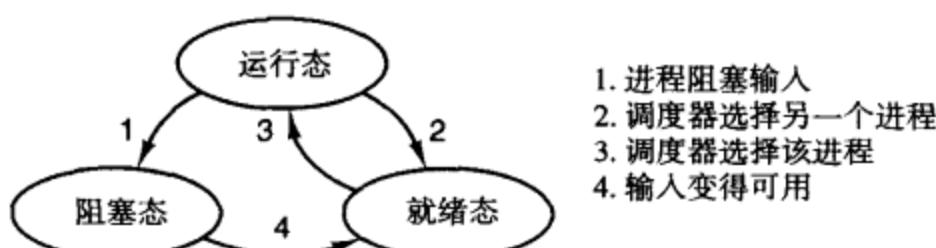


图2.2 一个进程可处于运行态、阻塞态和就绪态，这些状态的转换如图所示

这三种状态之间有四种可能的转换关系，如图所示。在进程发现自己无法继续运行下去时则发生转换1。在某些系统中，进程需要执行一个系统调用——`block`或`pause`，来进入阻塞态。在其他系统中，包括MINIX 3，当一个进程从管道或特殊文件（例如终端）读取数据时，如果没有可用的输入，则进程自动从运行态转换为阻塞态。

转换关系2和3是由进程调度器引起的，进程调度器是操作系统的一部分，进程甚至感知不到调度器的存在。在调度器认为运行进程占用处理机的时间已经足够长，决定让其他进程占用处理机时，发生转换2。在系统已让其他进程享有了它们应有的CPU时间而重新轮到该进程来占用处理机时，发生转换3。调度器的主要内容是决定哪个进程应当运行，以及它应运行多长时间。这是很重要的一点。目前已经设计了许多种算法，它们力图从整体上平衡系统的效率，并公平地对待各个进程。这将在本节的后面进行讨论

当一个进程等待的一个外部事件发生时（例如到达一些输入），发生转换4。如果此时没有其他进程运行，则立即触发转换3，该进程便开始运行。否则该进程将处于就绪态，等待调度器把CPU分配给它。

使用进程模型可使我们易于想象系统内部的操作状况。一些进程运行着一些程序，这些程序执行用户键入的命令。另一些进程是系统的一部分，它们的任务是处理下列工作：执行文件服务请求、管理运行磁盘驱动器和磁带机的细节等。当发生一个磁盘中断时，系统可能会做出这样的决定，即停止运行当前进程，而转向磁盘进程，该进程在此之前因等待该中断而处于阻塞态。这里说“可能”，是因为这取决于正在运行进程和磁盘驱动进程的相对优先级。但是关键的一点是，这样可以不再考虑中断，而是考虑用户进程、磁盘进程、终端进程等。这些进程在等待时总是处于阻塞态。所等待的事件发生时，它们被解除阻塞，并成为可被调度运行的进程。

这个观点引出了图 2.3 所示的模型。这里最低层是操作系统的调度器，在它上面有许多进程。所有关于中断处理、启动和中止进程的具体细节被隐藏在调度器中。实际上，它是一段非常短小的程序。操作系统的其他部分被简洁地组织成进程形式。图 2.3 所示的模型被 MINIX 3 使用，但是其中的调度器应不仅仅被理解为对进程的调度安排，同时也包括中断处理和所有的进程间通信。不过，作为近似描述，它展示了其基本结构。

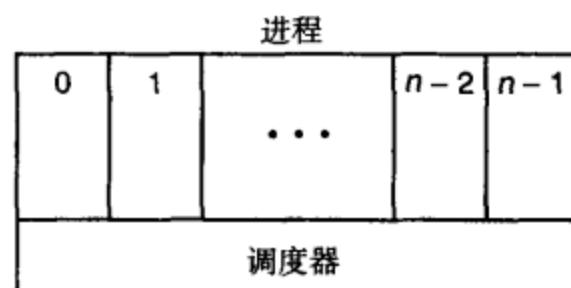


图 2.3 进程结构的操作系统的最低层处理中断与调度，该层的上面是顺序进程

2.1.6 进程的实现

为了实现进程模型，操作系统维持着一张表格（一个结构数组），即进程表（process table）。每个进程占用一个进程表项（一些作者把这些表项称为进程控制块）。该表项包含了进程的状态、它的程序计数器、栈指针、内存分配状况、打开文件状态、统计和调度信息、定时器和其他信号，以及进程由运行态到就绪态切换时所必须保存的其他信息。这样进程随后能够被再次启动，就像从未被中断过一样。

在 MINIX 3 中，进程通信、内存管理和文件管理是由系统中的几个独立模块分别处理的，所以进程表被分为几个部分，各模块维护它们各自所需要的那些域。图 2.4 示出了一些重要的域。与本章有关的域位于第一列，给出其他两列仅仅是为了让读者对系统的其他部分需要哪些信息有一大致了解。

在了解进程表之后，就可以对一台有单个 CPU、多个 I/O 设备的计算机如何维持多个顺序进程的假象做更多的解释。接下来的内容从技术上说是对图 2.3 中的 MINIX 3 调度器如何工作的一个描述，但多数现代的操作系统从本质上来说都差不多。每类 I/O 设备（例如软盘、硬盘、定时器、终端）都有一个靠近内存底部的位置，称为中断向量。它包含中断服务器的入口地址。假设当一个磁盘中断发生时，用户进程 23 正在运行，则中断硬件将程序计数器、程序状态字以及可能的一个或多个寄存器压入（当前）堆栈，计算机随即跳转到磁盘中断向量所指的地址处。这是硬件做的操作，从这里开始，软件就接管了一切。

中断服务过程的工作从把当前进程全部寄存器值存入进程表项开始。当前进程号及一个指向其表项的指针被保存在全局变量中，以便能够快速地找到它们。随后，将中断存入的那部分信息从堆栈中删除，并将栈指针指向一个被进程处理器（process handler）所使用的临时堆栈。一些动作，诸如保存寄存器值和设置栈指针等无法用 C 语言描述，所以由一个短小的汇编语言例程来完成。当该例程结束后，它调用一个 C 过程来完成特定中断类型剩下的工作。

内核	进程管理	文件管理
寄存器	正文段指针	UMASK 掩码
程序计数器	数据段指针	根目录
程序状态字	bss 段指针	工作目录
栈指针	退出状态	文件描述符
进程状态	信号状态	真实 ID
当前调度优先权	进程标识号	有效 UID
最大调度优先权	父进程	真实 GID
Scheduling ticks left	进程组	有效 GID
配额大小	子进程的 CPU 时间	控制 tty
使用的 CPU 时间	真实 UID	用于 read/write 的保存区
消息队列指针	有效 UID	系统调用参数
挂起的信号位	真实 GID	各种标志位
各种标志位	有效 GID	
进程名字	代码段共享所需文件信息	
	信号位图	
	各种标志位	
	进程名字	

图 2.4 MINIX 3 进程表的某些域，这些域分布在内核、进程管理器和文件系统中

MINIX 3 中的进程间通信通过消息完成，所以下一步是构造一条发给磁盘进程的消息，这时磁盘进程正被阻塞并等待该消息。这条消息通知说发生了一条中断，以此将它和那些由用户进程发送的消息加以区分。那些消息发出读磁盘块之类的请求。现在磁盘进程的状态由阻塞态转换到就绪态，然后，中断服务器调用调度器。在 MINIX 3 中，不同的进程有不同的优先级，以此向 I/O 设备服务例程提供比用户进程更好的服务。如果当前磁盘进程是优先级最高的就绪进程，则它将被调度运行。如果被中断进程具有与它相等或更高的优先级，则它将被再次调度运行，而磁盘进程只好等待一段时间。

不论哪种情况，被汇编语言中断代码所调用的 C 过程现在返回，汇编语言代码为新的当前进程装入寄存器值和内存映射并启动它运行。图 2.5 中总结了中断处理和调度的过程。值得注意的是，各系统在细节上略有不同。

1. 硬件压栈程序计数器等。
2. 硬件按中断向量下载新的程序计数器。
3. 汇编语言程序存储寄存器值。
4. 汇编语言程序设置新的栈。
5. C 语言中断服务例程运行。
6. 消息传递代码对等待的就绪任务进行标识。
7. 调度器决定哪个进程是下一个将运行的进程。
8. C 程序段返回汇编代码。
9. 汇编语言程序开始运行当前进程。

图 2.5 当一个中断发生后作为操作系统最低层的调度器的工作步骤

2.1.7 线程

在传统的操作系统中，每个进程中只存在一个地址空间和一个控制流（thread）。事实上，这几乎就是一个进程的定义。然而，有些情况下需要在相同的地址空间中有多个控制流并行地运行，就

像它们是单独的进程一样（只是它们共享相同的地址空间）。这些控制流通常被称为线程（thread），有时也称为轻量进程（lightweight process）。

观察进程的一种方式是把进程视为一组相关资源的集合。进程有一个存放程序正文和数据以及其他资源的地址空间。这些资源包括打开的文件、子进程、未处理的定时器（pending alarms）、信号处理器（signal handlers）和审计信息（accounting information）。通过以进程的形式把它们放在一起，我们就可以更方便地管理它们。

进程具有的另外一个概念是它是一个执行流（thread of execution），通常简称为线程（thread）。线程有一个程序计数器，用来跟踪下一条将要执行的指令。它有寄存器，存储当前使用的变量。它有堆栈，它存储着执行的历史，其中每一栈帧（frame）保存了没有返回的过程调用。尽管线程必须在进程中执行，但线程和它的进程是可以分别对待处理的两个不同概念。进程用来集合资源，而线程是CPU中调度的实体。

线程给进程模型增加的是，允许在同一个进程环境中多个执行流，这些流在很大程度上相对独立。在图2.6(a)中，可看到三个传统的进程。每个进程有自己的地址空间和单一的控制流。与此相反，在图2.6(b)中，可看到一个进程有三条控制流。尽管这两种情况都有三个线程，但在图2.6(a)中各线程在不同的地址空间中操作，而在图2.6(b)中所有三个线程共享同一个地址空间。

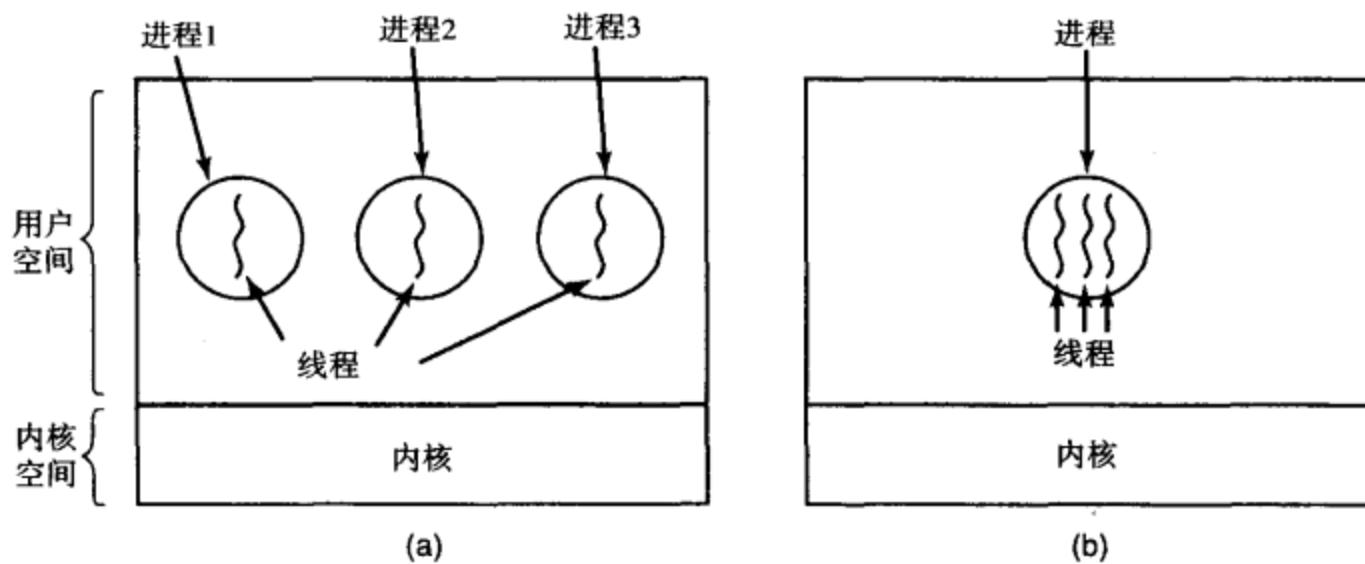


图2.6 (a)三个进程各有一个线程；(b)一个进程有三个线程

作为一个在何处使用多线程的一个例子，考虑一个网络浏览器进程。许多Web页面都包含有多幅很小的图像。对Web页面上的每一幅小图像，浏览器都必须与页面的驻留站点建立一条单独的连接以索取该图像。这样就有大量的时间浪费在建立和释放这些连接上。通过在浏览器内设立多个线程，便可以同时请求传输多幅图像。在多数情况下这样做显著地提高了性能，因为对于小图像，限制因素在于建立连接的时间，而不在于传输线的速率。

当同一地址空间中有多个线程时，图2.4中的几个域就不再针对进程，而是针对线程，于是就需要一张单独的线程表，每个线程占用一项。针对每个线程的信息包括程序计数器、寄存器值及状态。需要程序计数器是因为线程可以像进程一样被挂起和恢复运行。需要寄存器值是因为当线程被挂起时，它的寄存器值必须被保存下来。最后，线程像进程一样，可处于运行态、就绪态或阻塞态。图2.7列出了每个进程项和每个线程项。

在有些系统中，操作系统感知不到线程的存在。换言之，线程完全在用户空间进行管理。例如，当一个线程将被阻塞时，它在停止之前选择并启动它的后续线程。目前有几个用户级的线程软件包用得很普遍，包括POSIX的P-threads和Mach的C-threads软件包。

每个进程项	每个线程项
地址空间	程序计数器
全局变量	寄存器
打开文件	堆栈
子进程	状态
定时器	
信号和信号处理程序	
统计信息	

图 2.7 第一列列出了同一个进程中的所有线程共享的项，第二列列出了每个线程私有的一些项

在另外一些系统中，操作系统知道每个进程中存在多个线程，所以当一个线程阻塞时，操作系统会选择下一个运行的线程，它可能来自同一个进程，或者其他进程。为了进行调度，内核必须有一张线程表，其中列出了系统中的所有线程，这与进程表类似。

尽管这两种选择看起来是等价的，但它们的性能相差甚远。在用户空间管理的线程其切换速度比需要内核调用的情况快得多。这一事实对将线程管理放在用户空间提供了有力的支持。而另一方面，当线程完全在用户空间管理时，若一个线程阻塞（如等待 I/O 或处理页面故障），则内核将整个进程阻塞，因为它甚至不知道其他线程的存在。这一事实又对将线程放在内核进行管理提供了有力的支持（Boehm, 2005）。最后的结果是两种系统都被使用，同时还提出了各种混合方案（Anderson et al., 1992）。

不论线程是放在内核还是用户空间，都会带来一大堆必须解决的问题，而且这些问题相当程度上改变了编程模型。首先来考虑对 fork 系统调用的影响。如果父进程有多个线程，那么子进程是否也应该有这些线程呢？如果不是，那么它可能无法正常工作，因为可能所有的线程都是必不可少的。

然而，如果子进程获得了与父进程一样多的线程，当一个线程在一条 read 调用发生阻塞时，比如键盘，这时是否两个线程都阻塞在键盘上呢？当键入一行内容时，是否两个线程都得到一份副本？还是只有父进程得到？抑或是只有子进程得到？对于打开网络连接也存在同样的问题。

另一类问题与多线程共享许多数据结构有关。若一个线程关闭一个文件而另一个线程正在读该文件，将有什么后果呢？假设一个线程注意到内存不够并开始申请更多的内存，但此时发生线程切换，新运行的线程也注意到这个问题并再次申请内存。那么这里是申请一次还是两次呢？在几乎所有设计时未考虑线程的系统中，其库例程（例如申请内存的例程）都不可重入，如果前一个调用尚在激活时就进行第二次调用，则必然会引起崩溃。

另一个问题就是错误报告。在 UNIX 中，一条系统调用执行完之后，其状态将放在一个全局变量 *errno* 中。如果一个线程执行系统调用，在它读取 *errno* 之前，另一个线程也执行一条系统调用并清除了 *errno* 原先的值，则情况会怎样呢？

下面考虑信号（signal）。有些信号在逻辑上是针对线程的，有些则不是。例如，当一个线程调用 alarm 系统调用时，则将产生的信号传递给执行调用的线程是很合理的。当内核能够感知到线程时，通常它可以保证由正确的线程获得该信号。当内核感知不到线程时，线程软件包必须自己以某种方式跟踪定时器。对于用户级线程还有另一个麻烦：一个进程（例如在 UNIX 中）某一时刻只能定一个时间定时器，而若干线程各自独立地调用 alarm 系统调用时，则将发生混乱。

其他信号，例如键盘引发的 SIGINT 信号，不是特定于线程的。那么应该由谁来捕捉它们呢？是一个专门的线程、所有的线程还是一个新创建的线程呢？这些方法都存在问题。进一步考虑，如果一个线程修改了信号处理程序而未通知其他线程，将发生什么情况呢？

线程引起的最后一个问题是堆栈管理。在许多系统中，当发生堆栈溢出时，内核只是自动地提供更多的堆栈空间。当一个进程有多个线程时，它必须也有多个堆栈。如果内核感知不到所有这些

堆栈，则发生堆栈故障时它不能自动将其扩展，实际上，它甚至可能意识不到内存故障与堆栈增长有关。

这些问题当然并不是无法克服，但它们确实表明仅仅向一个现有系统中引入线程而不进行彻底的重新设计是根本不行的。起码系统调用的语义要重新定义，库例程也要重写。而且所有这些改变必须保持向后兼容，以保证现存的只有一个线程的进程的可用性。关于线程的其他信息，请参阅 Hauser 等（1993）和 Marsh 等（1991）。

2.2 进程间通信

进程经常需要与其他进程通信。例如，在一个 shell 管道中，第一个进程的输出必须传送到第二个进程，这样沿着管道传递下去。因此，在需要通信的进程之间，最好使用一种结构较好的方式，而不要使用中断。在随后的几节中，我们来分析进程间通信（IPC）问题。

进程间通信有三方面的内容。第一方面的内容已经在上面提到过：一个进程如何向另一个进程传送信息。第二方面的内容是必须要保证两个或多个进程在涉及临界活动时不会彼此影响（设想两个进程都试图攫取最后 100 KB 内存的情况）。第三方面的内容是当存在依赖关系时确定适当的次序：如果进程 A 产生数据，进程 B 打印数据，则 B 必须等到 A 产生了一些数据后才能开始打印。从下一节开始，我们将讨论这三个问题。

值得一提的是，在这些问题中有两个问题对于线程也适用。第一个关于传递信息的问题在线程中很容易解决，因为它们拥有相同的地址空间（在不同地址空间的线程的通信问题落到了进程间通信这个标题上）。另外两个关于互不妨碍和排序恰当的问题对于线程适用得很好：同样的问题存在并且同样的解决方案奏效。下面将以进程为例介绍，但是注意所有的问题和解决方案也适用于线程。

2.2.1 竞争条件

在有些操作系统中，协作进程可能共享一些彼此都能够读写的公用存储区。这个共享存储区可能在主存中（可能在一个内核数据结构中），也可能是一个共享文件，这里共享内存的位置并不影响通信的本质及其带来的问题。为了理解实际中进程间通信如何工作，考虑一个简单但很普遍的例子，即一个假脱机打印程序。当一个进程需要打印文件时，它将文件名放在一个特殊的打印机假脱机系统（spooler）目录下。另一个打印机守护进程（Printer daemon）周期性地检查是否有文件需要打印，如果有，则将其打印出来并将该文件名从 spooler 目录下删掉。

设想 spooler 目录有许多槽，编号依次为 0, 1, 2, …，每个槽存放一个文件名。同时设有两个共享变量：out，指明下一个被打印的文件；in，指向目录中下一个空闲的槽。这两个变量可以被保存在一个所有进程都可访问的两个字的文件中。某一时刻，0 到 3 号槽空闲（其中的文件已经被打印完毕），4 到 6 号槽被占用（其中存有待打印的文件名）。几乎在同一时刻，进程 A 和进程 B 都决定将一个文件排队打印，这种情况如图 2.8 所示。

在 Murphy 法则^①生效时，可能发生以下的情况。进程 A 读到 in 的值为 7，将它存在一个局部变量 next_free_slot 中。此时发生一次时钟中断，CPU 认为进程 A 已运行了足够长的时间，决定切换到进程 B。进程 B 也读取 in，同样得到值为 7，于是它将要打印的文件名存入 7 号槽，并将 in 的值更新为 8。然后它继续执行其他操作。

最后进程 A 接着从上次中止的地方再次运行，它检查变量 next_free_slot，发现其值为 7，于是将打印文件名存入 7 号槽，这将把进程 B 存在那里的文件名删掉。然后它将 next_free_slot 加 1，得

① 若某件事可能出错，则它一定会出错。

到值为8，就将8存到变量 *in* 中。此时 spooler 目录内部是一致的，所以打印机守护进程将发现不了任何错误，但进程B却永远得不到任何打印输出。类似这样的情况，即两个多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序，就称为竞争条件（race condition）。调试包含有竞争条件的程序是一件很头痛的事。大多数的运行结果都很好，但在极少数情况下会发生一些无法解释的奇怪事情。

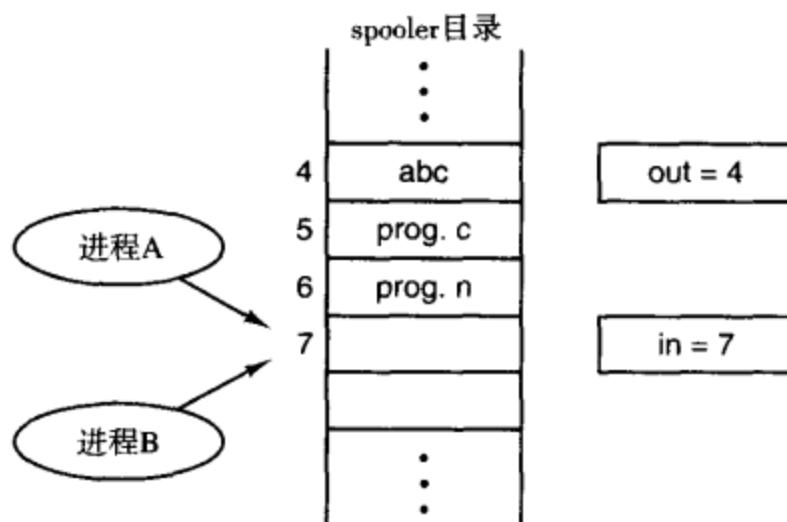


图 2.8 两个进程试图在同一时刻访问共享内存

2.2.2 临界区

如何避免部分条件呢？实际上凡涉及到共享内存、共享文件以及共享其他资源的情况都会引发与前面类似的问题。要避免这种错误，关键是要找出某种途径防止多个进程同时访问共享数据。换言之，这里需要的是互斥（mutual exclusion），即以某种手段确保当一个进程在使用一个共享变量或文件时，其他进程不能做同样的操作。前述问题的症结就在于，在进程A对共享变量的使用未结束之前进程B就使用它。为实现互斥而选择适当的原语是任何操作系统的主要设计内容之一，也是在这里将详细加以讨论的内容。

避免竞争条件的问题也可以用一种抽象的方式进行描述。在一部分时间内，进程忙于做内部计算或其他一些不会引发竞争条件的操作。在某些时候进程可能会访问共享内存或共享文件。这里把对共享内存进行访问的程序片段称为临界区或临界段（critical region 或 critical section）。如果能够进行适当的安排，使得两个进程不可能同时处于临界区，则能够避免竞争条件。

尽管这样的要求防止了竞争条件，但它还不能保证使用共享数据的并发进程能够正确和高效地进行操作。对于一个好的解决方案，需要具有以下四个条件：

1. 任何两个进程不能同时处于临界区。
2. 不应对 CPU 的速度和数目做任何假设。
3. 临界区外的进程不得阻塞其他进程。
4. 不得使进程在临界区外无休止地等待。

所期望的情形如图 2.9 所示。在这里，进程 A 于时刻 T_1 进入临界区，稍后，在时刻 T_2 进程 B 试图进入临界区，但是失败了，因为另外一个进程已经进入了临界区，并且临界区内一次只能进入一个进程。结果，B 被挂起直到时刻 T_3 进程 A 离开临界区为止，从而 B 立即进入临界区，最后在时刻 T_4 离开临界区，于是又回到了临界区内没有进程的初始状态。

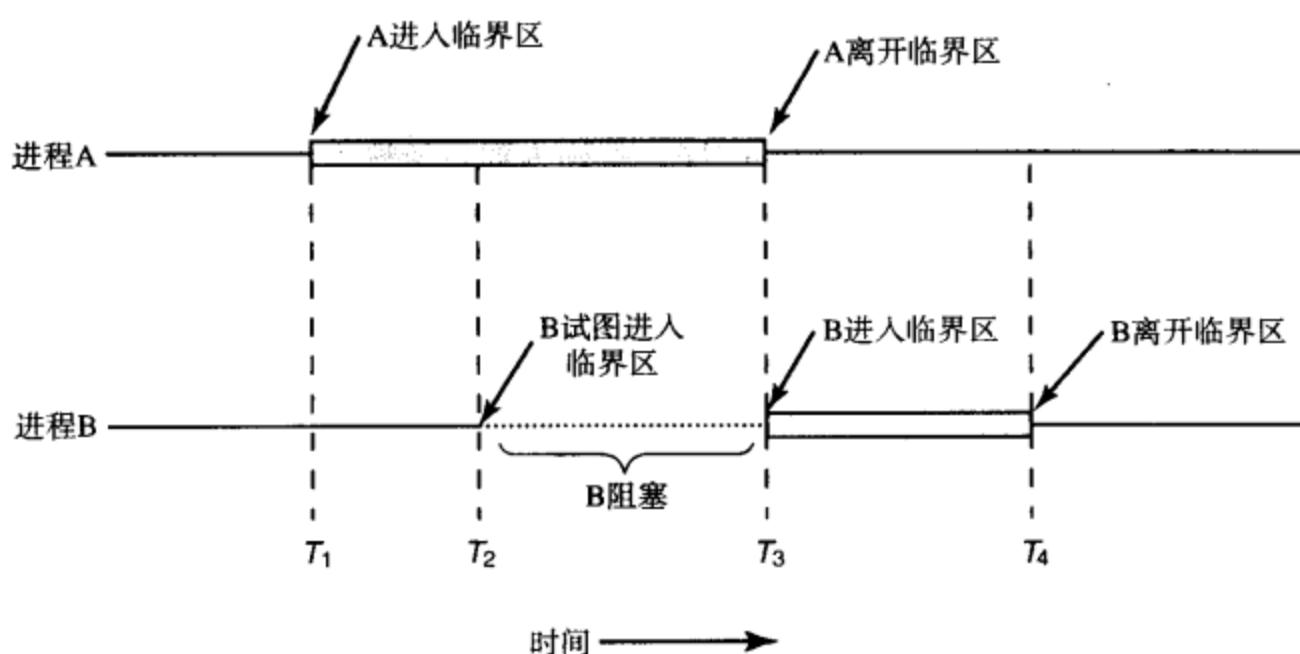


图 2.9 使用临界区实现的互斥

2.2.3 忙等待形式的互斥

本节将看几种实现互斥的方案。在这些方案中，当一个进程在临界区中更新共享内存时，其他进程将不会进入其临界区，也不会带来任何麻烦。

关闭中断

最简单的解决方案是使每个进程在进入临界区后先关中断，在离开之前再开中断。中断被关掉后，时钟中断也被屏蔽。CPU 只有在发生时钟或其他中断时才会进行进程切换，因此关中断之后 CPU 将不会被切换到其他进程。于是，一旦进程关中断之后，它就可以检查和修改共享内存，而不必担心其他进程的介入。

这种方案不好，因为把关中断的权力交给用户进程是不明智的。设想一下，若一个进程关中断之后不再开中断，其结果将会如何呢？系统可能会因此终止。而且，如系统有两个或多个共享内存的处理器，则关中断仅仅对执行本指令的 CPU 有效，其他 CPU 仍将继续运行，并可以访问共享内存。

另一方面，对内核来说，它在执行更新变量或列表的几条指令期间将中断关掉是很方便的。因为当就绪进程队列状态不一致时发生中断，会导致竞争条件。所以可得出结论：关中断对于操作系统本身是一项很有用的技术，但对于用户进程则不是一种合适的通用互斥机制。

锁变量

作为第二种尝试，需要寻找一种软件解决方案。设想有一个共享（锁）变量，初值为 0。当一个进程想进入其临界区时，它首先测试这把锁。如果锁的值为 0，则进程将其置为 1 并进入临界区。若锁已经为 1，则进程将一直等待到值变成 0。于是，0 就表示临界区内没有进程，1 就表示已经有某个进程进入了临界区。

遗憾的是，这种想法也含有与 spooler 目录一样的纰漏。假设一个进程读锁变量的值并发现它为 0，而恰好在它将其置为 1 之前，调度运行了另一个进程并将锁变量置为 1。当第一个进程能再次运行时，它同样也将锁置为 1，则此时同时有两个进程处于临界区中。

可能读者会想，先读取锁变量，紧接着在改变其值之前再检查一遍它的值，这样便可以解决问题。但这实际上无济于事。如果第二个进程恰好在第一个进程完成第二次检查之后修改锁变量，则同样还会发生竞争条件。

严格交替法

互斥的第三种方法示于图 2.10。几乎与本书中的所有其他程序一样，这里的程序段用 C 编写。之所以选择 C 是由于实际的操作系统普遍用 C 编写（偶尔也有用 C++ 编写的），而基本上不用类似的 Java 语言。对于编写操作系统而言，C 语言是强大、有效、可预知和具有关键特征的语言。而对于 Java 而言，它不可以预知，因为它在关键时刻会用完存储器，而且可能在不恰当的时候调用空间回收处理。在 C 语言中，这种情况不会发生，因为 C 语言没有空间回收实现。有关 C、C++、Java 和其他四种语言的比较可参阅 Prechelt (2000)。

```

while(TRUE){
    while(turn! = 0) /* loop */;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
}

while(TRUE){
    while(turn! = 1) /* loop */;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
}

(a)                                (b)

```

图 2.10 临界区问题的一种解法：(a)进程 0；(b)进程 1。两种情况下均要注意 while 语句终止时的分号

在图 2.10 中，整型变量 *turn* 的初值为 0，它用于跟踪轮到哪个进程进入临界区并检查或更新共享内存。一开始进程 0 检查 *turn*，发现它是 0，于是进入临界区。进程 1 同样也发现它是 0，于是执行一个等待循环不停地检测它是否变成了 1。持续地检测一个变量直到它具有某一特定值就称为忙等待 (busy waiting)。忙等待是应该避免的，因为它会浪费 CPU 时间。只有在有理由预期等待时间很短时才使用忙等待。一个适用忙等待的锁称为自旋锁 (spin lock)。

当进程 0 离开临界区时，它将 *turn* 置为 1，以允许进程 1 进入其临界区。假设进程 1 很快便离开了临界区，则这时两个进程都处于临界区之外，*turn* 的值被置为 0。现在进程 0 很快便执行完了其整个循环，退出临界区，并将 *turn* 置为 1。此时 *turn* 的值为 1 并且两个进程都在临界区外执行。

此时，进程 0 结束了其非临界区的操作并回到循环的开始，但很遗憾，这时它不能进入临界区，因为 *turn* 的值为 1，而进程 1 还在忙于非临界区的操作。这说明，轮流进入临界区在一个进程比另一个慢很多的情况下并不是一个好办法。

这种情况违反了以上描述的条件 3：进程 0 被一个临界区之外的进程阻塞。再回到 spooler 目录的问题，如果现在将临界区与读写 spooler 目录相联系，则进程 0 有可能因为进程 1 在做其他事情而被禁止打印另一个文件。

实际上，该方案要求两个进程严格地轮流进入它们的临界区。如果用假脱机方式打印文件，那么任何一个进程都不可能在一轮中打印两个文件。尽管该算法的确避免了所有的竞争，但由于它违反了条件 3，所以不能作为一个很好的备选方案。

Peterson 解决方案

荷兰数学家 T. Dekker 将轮换法和锁变量及警告变量的思想相结合，最早提出了一个不需要严格轮换的软件互斥解法。关于 Dekker 的算法，请参阅 Dijkstra (1965)。

在 1981 年，G. L. Peterson 发现了一种简单得多的互斥算法。这使得 Dekker 的方法不再有任何新意。Peterson 的算法示于图 2.11。该算法由两个用 ANSI C 写的过程组成。ANSI C 要求为所定义和使用的所有函数提供函数原型，但为了节省篇幅，在本例和后边的例子中将不给出函数原型。

```

#define FALSE 0
#define TRUE 1
#define N 2           /* 进程数 */

int turn;             /* 轮到谁了? */
int interested[N];   /* 所有值初始为 0 (FALSE) */

void enter_region(int process)    /* 进程号为 0 或 1 */
{
    int other;           /* 另一个进程的进程号 */

    other = 1 - process; /* 另一个进程 */
    interested[process] = TRUE; /* 标识出希望进入临界区 */
    turn = process;       /* 设置标志位 */
    while(turn == process && interested[other] == TRUE) /* 空语句 */
}

void leave_region(int process)    /* process: 即将离开临界区的进程 */
{
    interested[process] = FALSE; /* 标识将离开临界区 */
}

```

图 2.11 完成互斥的 Peterson 方案

在使用共享变量（即进入其临界区）之前，各进程使用其进程号 0 或 1 作为参数来调用 *enter_region*，该调用在需要时将使进程等待，直到能安全地进入。进程在完成对共享变量的操作之后，将调用 *leave_region*，表示操作已完成，若其他进程希望进入临界区，则现在可以进入。

我们来看这个方案是如何工作的。起初没有任何进程处于临界区，现在进程 0 调用 *enter_region*，它通过将其数组元素置位和将 *turn* 置为 0 来标识它希望进入临界区。由于进程 1 并不想进入临界区，所以 *enter_region* 很快便返回。如果进程 1 现在调用 *enter_region*，它将在此处挂起直到 *interested[0]* 变成 *FALSE*，该事件只有当进程 0 调用 *leave_region* 退出临界区时才会发生。

现在考虑两个进程几乎同时调用 *enter_region* 的情况。它们都将自己的进程号存入 *turn*。但只有后一个被保存进去的进程号才有效，前一个是无效的。假设进程 1 后存，则 *turn* 为 1。当两个进程都运行到 *while* 语句时，进程 0 将循环 0 次并进入临界区，而进程 1 将不停地循环，并不得进入临界区。

TSL 指令

现在来看一种需要硬件支持的方案。许多计算机，特别是那些为多处理器设计的计算机，都有一条指令

TSL RX, LOCK

(测试并上锁) 其工作原理如下所述：它将一个存储器字读到寄存器中，然后在该内存地址上存一个非零值。读数和写数操作保证是不可分割的，即该指令结束之前其他处理机均不允许访问该存储器字。执行 TSL 指令的 CPU 将锁住内存总线以禁止其他 CPU 在本指令结束之前访问内存。

为了使用 TSL 指令，需要使用一个共享变量 *LOCK* 来协调对共享内存的访问。当 *LOCK* 为 0 时，任何进程都可以使用 TSL 指令将 *LOCK* 置为 1 并读写共享内存。当操作结束时，进程用一条普通的 MOVE 指令将 *LOCK* 重新置为 0。

如何用这条指令来防止两个进程同时进入临界区呢？解决方案示于图 2.12 中。其中示出了使用四条指令的汇编语言子例程。第一条指令将 *LOCK* 原来的值复制到寄存器中并将 *LOCK* 置为 1，

随后这个原先的值与0相比较。如果它非零，则说明先前已被上锁，从而程序将回到开头并再次测试。经过或长或短的一段时间后，它将变成0（当前处于临界区中的进程退出临界区时），于是子例程返回，并上锁。清除这个锁很简单，程序只需将0存入LOCK即可，不需要特殊的指令。

enter_region:	TSL REGISTER,LOCK	复制LOCK到寄存器，并将LOCK置为1
	CMP REGISTER,#0	LOCK等于0吗？
	JNE ENTER_REGION	如果不等于0，已上锁，再次循环
	RET	返回调用程序，进入临界区
leave_region:	MOVE LOCK,#0	置LOCK为0
	RET	返回调用程序

图 2.12 用 TSL 指令上锁和清除锁

现在就有一种很明确的解法了：进程在进入临界区之前先调用 *enter_region*。这将导致忙等待，直到锁空闲为止。随后它获得锁变量并返回。在进程从临界区返回时它调用 *leave_region*，这将把LOCK置为0。与临界区问题的所有解法一样，进程必须在正确的时间调用 *enter_region* 和 *leave_region*，解法才能奏效。如果一个进程有欺诈行为，则互斥将会失败。

2.2.4 睡眠和唤醒

Peterson解法和TSL解法都是正确的，但它们都有忙等待的缺点。这些解法在本质上是这样的：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则进程将忙等待，直到许可为止。

这种方法不仅浪费CPU时间，还可能引起预料不到的结果。考虑一台计算机有两个进程H和L的情况，H优先级较高，L优先级较低，并且它们共享临界区。调度规则规定，只要H处于就绪态它就可以运行。在某一时刻，L处于临界区中，此时H变到就绪态准备运行（如一条I/O操作结束）。现在H开始忙等待，但由于当H就绪时L不会被调度，也就无法离开临界区，所以H将永远忙等待下去。这种情况有时被称为优先级反转问题（priority inversion problem）。

现在来看看几条进程间通信原语，它们在无法进入临界区时将阻塞，而不是忙等待。最简单的是睡眠（sleep）和唤醒（wakeup）。sleep系统调用将引起调用进程阻塞，即被挂起，直到另一进程将其唤醒。wakeup调用有一个参数，即要被唤醒的进程。另一种方法是sleep与wakeup各有一个参数，即一个用于匹配sleep和wakeup的内存地址。

生产者-消费者问题

这里考虑生产者-消费者问题（也称为有界缓冲区问题），作为如何使用这些原语的一个例子。两个进程共享一个公共的固定大小的缓冲区。其中的一个是生产者，负责将信息放入缓冲区；另一个是消费者，负责从缓冲区中取出信息（该问题也可被推广到m个生产者、n个消费者的情况，但出于简单起见，这里只考虑一个生产者、一个消费者的情况）。

麻烦之处在于当缓冲区已满，而此时生产者还想向其中放入一个新的数据项的情况。解决办法是让生产者睡眠，待消费者从缓冲区中取走一个或多个数据项时再唤醒它。同样，当消费者试图从缓冲区中取数据而发现缓冲区为空时，它就睡眠，直到生产者向其中放入一些数据时再将其唤醒。

这种方法听起来很简单，但它包含与前边 spooler 目录问题一样的竞争条件。为了跟踪缓冲区中的数据项数，需要一个变量 *count*。如果缓冲区最多存放N个数据项，则生产者将首先检查 *count* 是否达到N，若是，则生产者睡眠；否则生产者向缓冲区中放入一个数据项并将 *count* 的值增1。

消费者的实现与此类似：首先看 *count* 是否为 0，若是则睡眠；否则从中取走一个数据项并将 *count* 的值减 1。每个进程同时也检测另一个是否应睡眠，若不应睡眠则唤醒之。生产者和消费者的实现代码示于图 2.13。

```
#define N 100                                /* 缓冲区内的槽数 */
int count = 0;                                /* 缓冲区内数据项个数 */

void producer(void)
{
    int item;

    while(TRUE){                                /* 无限循环 */
        item = produce_item();                  /* 产生下一个数据项 */
        if(count == N) sleep();                /* 缓冲区满，进入睡眠 */
        insert_item(item);                    /* 将一个数据项放入缓冲区 */
        count = count + 1;                   /* 缓冲区内数据项个数增 1 */
        if(count == 1) wakeup(consumer);      /* 缓冲区为空？ */
    }
}

void consumer(void)
{
    int item;

    while(TRUE){                                /* 无限循环 */
        if(count == 0)sleep();                /* 缓冲区空，进入睡眠 */
        item = remove_item();                /* 从缓冲区中取走一个数据项 */
        count = count-1;                     /* 缓冲区中数据项个数减 1 */
        if(count == N - 1) wakeup(producer);  /* 缓冲区满？ */
        consume_item(item);                 /* 打印数据项 */
    }
}
```

图 2.13 含有竞争条件的生产者 - 消费者问题

为了在 C 语言中表示诸如 *sleep* 和 *wakeup* 的系统调用，这里将用库函数的调用形式来表示。尽管它们不是标准C库的一部分，但在实际上具有这些系统调用的所有系统中都应具有这两种库函数。图中未示出的 *enter_item* 和 *remove_item* 过程用来记录将数据项放入缓冲区和从缓冲区中取出数据项。

现在回到竞争条件的问题。这里有可能会出现竞争条件，其原因是对 *count* 的访问未加限制。有可能出现以下情况：缓冲区为空，消费者刚刚读取 *count* 的值发现它为 0。此时调度器决定暂停消费者并启动运行生产者。生产者向缓冲区中加入一个数据项，将 *count* 加 1。现在 *count* 的值变成了 1。它推断认为由于 *count* 刚才为 0，所以消费者此时一定在睡眠，于是生产者调用 *wakeup* 来唤醒消费者。

遗憾的是，消费者此时在逻辑上并未睡眠，所以唤醒信号丢失。当消费者下次运行时，它将测试先前读到的 *count* 值，发现它为 0，于是去睡眠。这样生产者迟早会添满整个缓冲区，然后睡眠。这样一来两个进程都将永远睡眠下去。

这里问题的实质在于发给一个未睡眠进程的唤醒信号被丢失了。如果它没有丢失，则一切都很正常。一种快速的弥补方法是修改规则，加上一个唤醒等待位（wakeup waiting bit）。当向一个清醒的进程发送一个唤醒信号时，将该位置位。随后，当进程要睡眠时，如果唤醒等待位为1，则将该位置0，而进程仍然保持清醒。

尽管在本例中唤醒等待位解决了问题，但很容易就可以构造出一些例子，其中有两个或更多的进程，这时一个唤醒等待位就不够用了。这里可以再进行补充，加入第二个唤醒等待位，或者甚至更多，但原则上讲这并未解决问题。

2.2.5 信号量

信号量是E. W. Dijkstra在1965年提出的一种方法，它使用一个整型变量来累计唤醒次数，以供以后使用。在他的建议中，引入了一个新的变量类型，称为信号量（semaphore）。一个信号量的值可以为0，表示没有积累下来的唤醒操作；或者为正值，表示有一个或多个被积累下来的唤醒操作。

Dijkstra建议设两种操作：down和up（分别为一般化的sleep和wakeup）。对一信号量执行down操作是检查其值是否大于0。如果是这样，则将其值减1（即用掉一个保存的唤醒信号）并继续。如果值为0，则进程将睡眠，而且此时down操作并未结束。检查数值、改变数值以及可能发生的睡眠操作均作为一个单一的、不可分割的原子操作（atomic action）完成。即保证一旦一个信号量操作开始，则在操作完成或阻塞之前其他的进程均不允许访问该信号量。这种原子性对于解决同步问题和避免竞争条件是非常重要的。

up操作递增信号量的值。如果一个或多个进程在该信号量上睡眠，无法完成一个先前的down操作，则由系统选择其中的一个（例如随机挑选）并允许其完成它的down操作。于是，对一个有进程在其上睡眠的信号量执行一次up操作之后，该信号量的值仍旧是0，但在其上睡眠的进程却少了一个。递增信号量的值和唤醒一个进程同样也是不可分割的。不会有进程因执行up操作而阻塞，正如在前面的模型中不会有进程因执行wakeup操作而阻塞一样。

在Dijkstra最早的论文中，他使用p和v而不是down和up，但因为对于不讲荷兰语的读者来说采用什么记号并无大的干系，所以这里使用down和up。它们在Algol 68中首次被引入。

用信号量解决生产者-消费者问题

图2.14中示出了用信号量解决丢失的唤醒问题。最重要的是它采用一种不可分割的方式来实现。通常是将up和down作为系统调用实现，而且操作系统只需在执行以下操作时短暂地关掉中断，这些操作包括：检测信号量、修改信号量以及在需要时使进程睡眠。由于这些动作只需要几条指令，所以关中断不会带来什么副作用。如果使用多个CPU，则每个信号量应由一个锁变量进行保护。通过TSL指令来确保同一时刻只有一个CPU在对信号量进行操作。读者必须搞清楚使用TSL来防止几个CPU同时访问信号量，与生产者或消费者使用忙等待来等待对方腾出或填充缓冲区是完全不同的。信号量操作仅需几个微秒，而生产者或消费者则可能需要任意长的时间。

该解决方案使用了三个信号量：full用来记录满的缓冲槽数目，empty用来记录空的缓冲槽总数，mutex用来确保生产者和消费者不会同时访问缓冲区。full的初值为0，empty的初值为缓冲区内槽的数目，mutex的初值为1。两个或多个进程使用的初值为1的信号量保证同时只有一个进程可以进入临界区，它被称为二进制信号量（binary semaphores）。如果每个进程在进入临界区前都执行一个down操作，并在退出时执行一个up操作，则能够实现互斥。

```

#define N 100           /* 缓冲区内槽数 */
typedef int semaphore;   /* 信号量是一种特殊的整型变量 */
semaphore mutex = 1;    /* 控制对临界区的访问 */
semaphore empty = N;    /* 记录缓冲区内空的槽数 */
semaphore full = 0;     /* 记录缓冲区内满的槽数 */

void producer(void)
{
    int item;

    while(TRUE){          /*TRUE 为常量 1 */
        item = produce_item(); /* 产生一个需放入缓冲区的数据项 */
        down(&empty);      /* 递减空槽数 */
        down(&mutex);      /* 进入临界区 */
        insert_item(item);  /* 将一个新数据项放入缓冲区 */
        up(&mutex);        /* 离开临界区 */
        up(&full);         /* 递增满槽数 */
    }
}

void consumer(void)
{
    int item;

    while(TRUE){          /* 无限循环 */
        down(&full);       /* 递减满槽数 */
        down(&mutex);      /* 进入临界区 */
        item = remove_item(); /* 从缓冲区中取走一个数据项 */
        up(&mutex);        /* 离开临界区 */
        up(&empty);        /* 递增空槽数 */
        consume_item(item); /* 对数据项进行操作 */
    }
}

```

图 2.14 使用信号量的生产者 - 消费者问题

在有了一些进程间通信原语之后，我们需要回头再观察一下图 2.5 中的中断顺序。在使用信号量的系统中，隐藏中断机构的最自然的方法是为每一个 I/O 设备设置一个信号量，初值设为 0。在启动一个 I/O 设备之后，其管理进程对相关联的信号量执行一个 `down` 操作，于是立即被阻塞。当中断到来时，中断处理程序随后对相关联的信号量执行一个 `up` 操作，这将使相关的进程再次处于就绪态。该模型中，图 2.5 中的第 6 步具体化为在设备的信号量上执行 `up` 操作，所以在第 7 步中，调度器将能够运行设备管理程序。当然，如果这时有几个进程就绪，则调度器下次可以选择一个最为重要的进程来运行。在本章以后我们将看到如何进行调度。

在图 2.14 的例子中，实际上通过两种不同的方式来使用信号量，它们之间的区别是很重要的。信号量 `mutex` 用于互斥。它用于保证任一时刻只有一个进程读写缓冲区和相关的变量。互斥是避免混乱所必需的。

信号量的另一种用途是同步（synchronization）。信号量 `full` 和 `empty` 用来保证一定的事件顺序发生或不发生。在本例中，它们保证当缓冲区满时生产者停止运行，以及当缓冲区空时消费者停止运行。这种用法与互斥是不同的。

2.2.6 互斥

如果不需要信号量的计数能力，有时可以使用信号量的一个简化版本，称为互斥（mutex）。互斥仅仅适用于管理共享资源或一小段代码时。由于互斥实现起来既容易又有效，这使得互斥信号体在实现用户空间线程包时非常有用。

互斥是一个可以处于两态之一的变量：解锁和加锁。这样，只需要一个二进制位来表示它，但实际上，常常使用一个整数来表示，0表示解锁，其他值则表示加锁。互斥适用两个过程。当一个进程（或线程）需要进入临界区时，它调用 *mutex_lock*，如果此时互斥是解锁的（即临界区是可用的），则此调用成功，调用进程可以进入临界区。

另一方面，如果该互斥已经加锁，调用者被阻塞，直到在临界区中的进程完成操作并调用 *mutex_unlock* 退出为止。如果多个进程在互斥上阻塞，则随机选择一个进程并允许它获得锁。

2.2.7 管程

有了信号量之后，进程间通信看来很容易了，对吗？答案是否定的。仔细看图 2.14 中向缓冲区放入数据项以及从中删除数据项之前的 *down* 操作。假设将生产者代码中的两个 *down* 操作交换一下次序，即使 *mutex* 的值在 *empty* 之前被减 1，而不是在其之后。如果缓冲区完全是满的，生产者将阻塞，*mutex* 值为 0。这样一来，当消费者下次试图访问缓冲区时，它将对 *mutex* 执行一个 *down* 操作，由于 *mutex* 的值为 0，则它也将阻塞。两个进程都将永远地阻塞下去，无法再做有效的工作，这种糟糕的状况称为死锁（deadlock）。我们将在第 3 章中详细地研究死锁。

指出这个问题是为了说明使用信号量时要非常小心！一处很小的错误将导致很大的麻烦。这就像用汇编语言编程一样，甚至更糟，因为这里出现的错误都是竞争条件、死锁以及其他不可预测和不可重现的行为。

为了更易于编写正确的程序，Hoare (1974) 和 Brinch Hansen (1975) 提出了一种高级的同步原语，称为管程（monitor）。他们两人提出的方案略有不同，如下所述。管程是由过程、变量及数据结构等组成的集合，它们组成一个特殊的模块或软件包。进程可在任何需要时调用管程中的过程，但它们不能在管程外的过程中直接访问管程内的数据结构。这一规则在诸如 Java 之类的面向对象语言中是很常见的，尽管对象可以追溯到 Simula 67，但在当时并不常用。图 2.15 展示了用一种想象的类 Pascal 语言描述的管程。

```
monitor example
    integer i;
    condition c;

    procedure producer(x);
    :
    end;

    procedure consumer(x);
    :
    end;
end monitor;
```

图 2.15 一个管程

管程有一个很重要的特性，这使得它们能有效地完成互斥：任意时刻管程中只能有一个活跃进程。管程是一种编程语言的组件，所以编译器知道它们很特殊，并可以采用与其他过程调用不同的方法来处理它们。典型地，当一个进程调用管程中的过程时，前几条指令将检查在管程中是否有其他的活跃进程。如果有，调用进程将挂起，直到另一个进程离开管程。如果没有，则调用进程便进入管程。

对进入管程实现互斥由编译器负责，但通常的做法是用一个互斥或二进制信号量。因为是由编译器而非程序员来安排互斥的，所以出错的可能性要小得多。在任意时刻，写管程的人无须关心编译器是如何实现互斥的，他只需知道将所有的临界区转换成管程中的过程即可，而绝不会有两个进程同时执行临界区中的代码。

尽管如上边所看到的，管程提供了一种实现互斥的简便途径，但这还不够。这里还需要一种办法，使得进程在无法继续运行时被阻塞。在生产者-消费者问题中，很容易将针对缓冲区满和缓冲区空的测试放到管程的过程中，但是生产者在发现缓冲区满时如何阻塞呢？

解决方法在于引入条件变量（condition variable）以及相关的两个操作：`wait` 和 `signal`。当一个管程过程发现它无法继续时（如生产者发现缓冲区满时），它在某些条件变量上执行 `wait`，如 `full`。这个动作引起调用进程阻塞。它也允许另一个先前被挡在管程之外的进程现在进入管程。

另一个进程，如消费者，可以通过对其伙伴正在其上等待的一个条件变量执行 `signal` 操作来唤醒正在睡眠的伙伴进程。为避免管程中同时有两个活跃进程，需要一条规则来通知在 `signal` 之后该怎么办。Hoare 建议让新唤醒的进程运行，而挂起另一个进程。Brinch Hansen 则建议要求执行 `signal` 的进程必须立即退出管程。换言之，`signal` 语句只可能作为一个管程过程的最后一条语句。这里将采纳 Brinch Hansen 的建议，因为它在概念上更简单，并且更容易实现。如果在一个条件变量上正有若干进程等待，则对该条件变量执行 `signal` 操作，调度器将在其中选择一个使其恢复运行。

另外，还有 Hoare 或 Brinch Hansen 都没提到的第三种方法。该方法让发送信号者继续运行，并且只有在发信号者退出管程之后，才允许等待的进程开始执行。

条件变量不是计数器，它们并不像信号量那样积累信号供以后使用，所以如果向一个其上没有等待进程的条件变量发送信号，则该信号将丢失。`wait` 操作必须在 `signal` 之前。这条规则使得实现简单了许多。实际上这不是一个问题，因为用变量很容易跟踪每个进程的状态。一个原本要执行 `signal` 的进程通过检查这些变量便可以知道该操作是不需要的。

图 2.16 中用类 Pascal 语言给出了使用管程解决生产者-消费者问题的解法框架。使用类 Pascal 语言的优点在于它清晰、简单，并且严格符合 Hoare/Brinch Hansen 模型。

读者可能会觉得 `wait` 和 `signal` 操作看起来很像前面提到的 `sleep` 和 `wakeup`。它们确实很相似，但存在一点很关键的区别：`sleep` 和 `wakeup` 之所以失败，是因为当一个进程想睡眠时另一个进程试图去唤醒它。使用管程将不会发生这种情况。对管程过程的互斥保证了这样一点：如果管程中的过程发现缓冲区满，它将能够完成 `wait` 操作而不用担心调度器可能会在 `wait` 完成之前切换到消费者进程。消费者进程甚至在 `wait` 完成且生产者被标志为不可运行之前根本不允许进入管程。

尽管类 Pascal 语言是一种理想的语言，但还是有一些真正的编程语言支持管程，不过它们不一定是 Hoare 和 Brinch Hansen 所设计的模型。其中一种语言是 Java。Java 是一种面向对象的语言，它支持用户级线程，还允许将方法（过程）划分为类。只要将关键词 `synchronized` 加入到方法声明中，Java 保证一旦某个线程执行该方法，就不允许其他线程执行该类中的任何其他方法。

Java 中的同步方法与经典管程有本质差别：Java 没有条件变量。取而代之的是，Java 提供了两个过程：`wait` 和 `notify`，它们分别和 `sleep` 和 `wakeup` 等价，不同的是，它们在同步方法内部使用，不受竞争态条件的影响。

```

monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;

```

图 2.16 一个使用管程的生产者-消费者问题解法概述。在某时刻仅有一个管程过程活跃，缓冲区有 N 个槽

通过临界区互斥的自动化，管程使并行编程比用信号量更容易保证正确性。但它也有缺点。图 2.16 中之所以使用类 Pascal 而不像其他例子那样使用 C 语言并不是没有原因的。正如早先所说，管程是一个编程语言概念。编译器必须要识别出管程并用某种方式对互斥做出安排。C、Pascal 及多数其他语言都没有管程，所以指望这些编译器来实现互斥规则是不可靠的。实际上，编译器如何知道哪些过程属于管程内部哪些不属于管程内部，这也是一个问题。

上述语言同样也没有信号量，但增加信号量是很容易的：需要做的就是向库里加入两段短小的汇编程序代码，以执行 up 和 down 系统调用。编译器甚至不用知道它们的存在。当然，操作系统必须知道信号量，如果有一个基于信号量的操作系统，那么仍旧可以使用 C 或 C++（如果愿意，甚至可以使用 FORTRAN）来写用户程序。如果使用管程，就需要一种带有管程的语言。

与管程和信号量有关的另一个问题是，它们都被设计用来解决访问一块公共存储器的一个或多个 CPU 上的互斥问题。通过将信号量放在共享内存中并用 TSL 指令来保护它们，可以避免竞争。

对于一个具有多个CPU、各CPU拥有自己的私有内存、由一个局域网相连的分布式系统，这些原语将失效。得出的结论是：信号量太低级，而管程在少数几种编程语言以外无法使用。同时，这些原语均未提供机器间的信息交换方法，所以还需要其他的东西。

2.2.8 消息传递

前边提及的其他东西就是消息传递（message passing）。这种进程间通信的方法使用两条原语 `send` 和 `receive`。它们像信号量一样，是系统调用，而不像管程那样是语言组件。因此，它们可以很容易地被加入到库例程中。例如，

```
send(destination, &message);
```

和

```
receive(source, &message);
```

前一个调用向一个给定的目标发送一条消息，后一个调用从给定的源（或者是任意源，如果接收者不介意）接收一条消息。如果没有消息可用，则接收者可能一直阻塞到一条消息到达。或者它也可以立即返回，并带回一个错误码。

消息传递系统的设计要点

消息传递系统面临许多信号量和管程所未涉及的问题和设计难点。特别是对于通信进程位于网络中不同机器的情况。例如，消息有可能被网络丢失。为了防止消息丢失，发送方和接收方可以达成一致：一旦信息被接收到，接收方马上回送一条特殊的应答（acknowledgement）消息。如果发送方在一段时间间隔内未收到应答，则进行重发。

现在考虑消息本身被正确地接收，而应答信息丢失的情况。发送者将重发信息，这样接收者将接收到两次。对于接收者来说，区分新消息和一条重发的老消息是非常重要的。通常采用在每条原始消息中嵌入一个连续的序号来解决该问题。如果接收者收到一条消息，它具有与前一条消息一样的序号，则它就知道这条消息是重复的，可以忽略。

消息系统还需要解决进程命名的问题，这样才能明确在 `send` 和 `receive` 调用中所指定的进程。身份认证也是一个问题：客户如何知道它是在与一个真正的文件服务器通信，而不是一个冒充者？

另一方面，对发送者和接收者在同一台机器上的情况，也存在若干设计问题，其中一点是性能。将消息从一个进程复制到另一个进程通常比信号量操作和进入一个管程要慢。为了使消息传递变得高效，人们已经做了许多工作。例如，Cheriton (1984) 建议限制信息的大小，使其能装入机器的寄存器中，然后便可以使用寄存器进行消息传递。

用消息传递解决生产者-消费者问题

现在我们来看如何用消息传递而不是共享内存来解决生产者-消费者问题。图2.17中给出了一种解法。假设所有的消息都有同样的大小，并且尚未接收到的消息由操作系统自动进行缓冲。在该图中，共使用了 N 条信息，这就类似于一块共享内存缓冲区中的 N 个槽。消费者首先将 N 条空消息发送给生产者。当生产者向消费者传递一个数据项时，它取走一条空消息并送回一条填充了内容的消息。通过这种方式，系统中总的消息数保持不变，所以消息可以存放在预知数量的内存中。

如果生产者的速度比消费者快，则所有的消息最终都将被填满，于是生产者将阻塞，以等待消费者取用后返回一条空消息。如果消费者速度快，则正好相反：所有的消息均为空，等待生产者来填充它们，消费者阻塞以等待一条填充过的消息。

```

#define N 100                                /* 缓冲区中的槽数 */

void producer(void)
{
    int item;
    message m;                            /* 消息缓冲区 */

    while(TRUE){
        item = produce_item( );           /* 产生一些数据放入缓冲区 */
        receive(consumer, &m);          /* 等待一条空消息到达 */
        build_message(&m, item);        /* 构造一条消息供发送 */
        send(consumer, &m);             /* 向消费者发送一数据项 */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for(i = 0; i<N; i++) send(producer, &m); /* 发送 N 条空消息 */
    while(TRUE){
        receive(producer, &m);          /* 收到一条包含数据的消息 */
        item = extract_item(&m);        /* 从消息中提取数据 */
        send(producer, &m);             /* 回送空消息作为应答 */
        consume_item(item);            /* 使用数据项进行操作 */
    }
}

```

图 2.17 用 N 条消息的生产者 - 消费者进程

消息传递可以有许多变体。对于初学者，让我们来看看如何对消息编址。一种方法是为每个进程分配一个唯一的地址，按进程为消息指定地址。另一种方法是引入一种新的数据结构，该结构称为信箱（mailbox）。信箱就是用来对一定数量的消息进行缓冲的地方，典型的情况是在信箱创建时确定消息的数量。当使用信箱时，`send` 和 `receive` 调用中的地址参数使用信箱，而不是进程。当一个进程试图向一个满的信箱发消息时，它将被挂起，直至信箱内有消息被取走而为新消息腾出空间。

对于生产者 - 消费者问题，生产者和消费者均应创建足够容纳 N 条消息的信箱。生产者向消费者信箱发送包含数据的消息，消费者则向生产者信箱发送空消息。当使用信箱时，缓冲机制是很清楚的：目标信箱容纳那些被发送但尚未被目标进程接收的消息。

使用信箱的另一种极端情况是彻底去掉缓冲。采用这种方法时，如果 `send` 在 `receive` 之前执行，则发送进程被阻塞，直到 `receive` 发生，执行 `receive` 时，消息可以直接从发送者复制到接收者，不用任何中间缓冲。类似地，如果 `receive` 先被执行，则接收者阻塞直到 `send` 发生。这种策略常被称为聚合（rendezvous）原则。与带有缓冲的消息方案相比，这种方案实现起来更容易一些，但却降低了灵活性，因为发送者和接收者一定要以步步紧接的方式运行。

在 MINIX 3 中的进程间的通信采用聚合的方式，使用固定大小的消息通信。用户进程也用这种方式与操作系统组件进行通信，编程者可能看不到这些，因为这由库程序区仲裁系统调用。

MINIX 3（和 UNIX）中用户进程间采用管道进行通信。采用信箱的消息系统和管道机制之间的区别实际在于管道没有预先设定消息的边界。换言之，如果一个进程向管道写入 10 条 100 字节的消息，而另一个进程从管道中读取 1000 个字节，则读进程将一次性地获得所有 10 条消息。而在一个真正的消息系统中，每个 `read` 操作将只返回一条消息。当然，如果进程能够达成一致：总是从管道中读写固定大小的消息，或者每条消息都以一个特殊字符（如换行符）结束，则不会有任何问题。

通常在并行程序设计系统中使用消息传递。例如，一个著名的消息传递系统是 MPI（消息传递接口），它广泛用于科学计算当中。有关该系统的更多信息，可参阅 Gropp 等（1994）和 Snir 等（1996）。

2.3 经典 IPC 问题

操作系统文献中有许多应用多种同步方法来解决进程间通信问题的讨论。以下几节将讨论两个较为著名的问题。

2.3.1 哲学家进餐问题

在 1965 年，Dijkstra 提出并解决了一个他称之为哲学家进餐的同步问题。从那时起，每个发明新的同步原语的人都希望通过解决哲学家进餐问题来展示其同步原语的精妙之处。这个问题可以简单地描述如下：五个哲学家围坐在一个张圆桌周围，每个哲学家的前面都有一碟通心面，由于面条很滑，所以要两把叉子才能夹住。相邻两个碟子之间有一把叉子。餐桌的布局如图 2.18 所示。

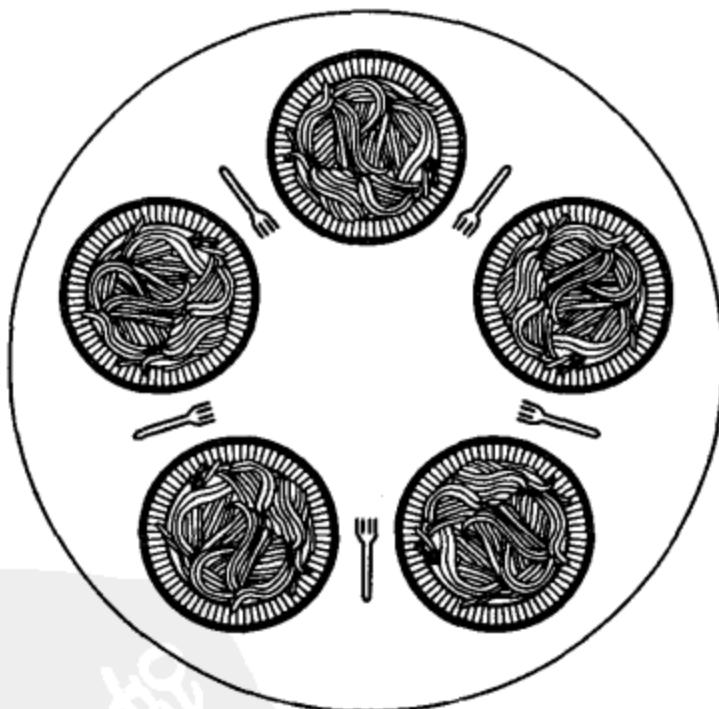


图 2.18 哲学家进餐图

哲学家的生活包括两种活动，即吃饭和思考（这只是一个抽象，即对本问题而言其他活动都无关紧要）。当一个哲学家觉得饿时，他就试图分两次去取他左边和右边的叉子，每次拿一把，但不分次序。如果成功地获得了两把叉子，他就开始吃饭，吃完以后放下叉子继续思考。这里的问题就是：为每一个哲学家写一段程序来描述其行为，要求不能死锁（要求拿两把叉子是人为规定的，这里也可以将意大利面条换成中国菜，用米饭代替通心面，用筷子代替叉子）。

图 2.19 给出了最浅显的解法。过程 *take_fork* 将一直等到所指定的叉子可用，然后将其取用。遗憾的是，这种解法是错误的。设想所有五个哲学家都同时拿起左面的叉子，则他们都拿不到右面的叉子，于是发生死锁。

```
# define N 5                                /* 哲学家数目 */

void philosopher(int i)                      /* i: 哲学家号码，从 0 到 4 */

{
    while(TRUE){
        think();                            /* 哲学家正在思考 */
        take_fork(i);                      /* 取左叉 */
        take_fork((i + 1) % N);           /* 取右叉；% 为取余 */
        eat();                             /* 吃面 */
        put_fork(i);                      /* 放回左叉 */
        put_fork((i + 1) % N);           /* 放回右叉 */
    }
}
```

图 2.19 哲学家进餐问题的一种不正确解法

我们可以将程序修改一下，规定在拿到左叉后，查看右面的叉子是否可用。如果不可用，则先放下左叉，等一段时间后再重复整个过程。尽管与前一种的原因不同，但这种解法也是错误的。可能在某一个瞬间，所有的哲学家都同时启动这个算法，拿起左叉，看到右叉不可用，又都放下左叉，等一会儿，又同时拿起左叉，如此这样永远重复下去：所有的程序都在运行，但却无法取得进展，这种情况就称为饥饿（starvation）（即使问题不发生在意大利或中国餐馆，也被称为饥饿）。

现在读者可能会想：如果哲学家在拿不到右叉时等待一段随机的时间，而不是等待相同的时间，则长时间处于上述死锁状态的机会就很小了。这种想法是对的，而且在大多数应用中，稍后再试的方法也不成问题。例如，在使用以太网的局域网当中，一台计算机只有在检测到其他计算机都没有在发送数据包的时候发送数据包。但是由于传输延迟，两台由电缆相连的电脑可能交迭发送数据包，从而出现数据包碰撞。当检测到数据包碰撞时，每台电脑等待一段随机的时间后重试。实际上，这一方案工作得很好。但在一些应用中人们希望一种完全正确的方案，它不能因为一串靠不住的随机数字而失效（想想核电站中的安全控制系统）。

对图 2.19 中的算法可进行下列改进，它既不会发生死锁又不会发生饥饿：使用一个二进制信号量对 *think* 函数之后的五条语句进行保护。在哲学家开始拿叉子之前，先对信号量 *mutex* 执行 *down* 操作。放回叉子后，再对 *mutex* 执行 *up* 操作。从理论上讲，这种解法是可行的。但从实际角度来看，这里有性能上的局限：同一时刻只能有一个哲学家进餐。而五把叉子实际允许两个哲学家同时进餐。

图 2.20 中的解法不仅正确，而且对于任意多个哲学家的情况都能获得最大的并行度。其中使用一个数组 *state* 来跟踪一个哲学家是在吃饭、思考还是正在试图拿叉子：一个哲学家只有在两个邻居都不在进餐时才允许进入到进餐状态。第 *i* 位哲学家的邻居由宏 *LEFT* 和 *RIGHT* 定义。换言之，若 *i* 为 2，则 *LEFT* 为 1，*RIGHT* 为 3。

哲学家进餐问题的解决方案使用了一个信号量数组，每个信号量分别对应一个哲学家，这样，当所需的叉子被占用时，想进餐的哲学家可以阻塞。注意，每个进程将过程 *philosopher* 作为主代码运行，而其他过程 *take_forks*、*put_forks* 和 *test* 只是普通的过程，而非单独的进程。

```

#define N      5           /* 哲学家数目 */
#define LEFT    (i + N - 1)%N /* i 的左邻号码 */
#define RIGHT   (i + 1)%N   /* i 的右邻号码 */
#define THINKING 0          /* 哲学家正在思考 */
#define HUNGRY   1          /* 哲学家想取得叉子 */
#define EATING   2          /* 哲学家正在吃饭 */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
/* 信号量是一个特殊的整型变量 */
/* 记录每个人状态的数组 */
/* 临界区互斥 */
/* 每个哲学家一个信号量 */

void philosopher(int i)          /* i: 哲学家号码, 从 0 到 N - 1 */
{
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i)            /* i: 哲学家号码, 从 0 到 N - 1 */
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
    /* 进入临界区 */
    /* 记录下哲学家 i 饥饿的事实 */
    /* 试图得到两把叉子 */
    /* 离开临界区 */
    /* 如果得不到叉子就阻塞 */
}

void put_forks(i)                /* i: 哲学家号码, 从 0 到 N - 1 */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
    /* 进入临界区 */
    /* 哲学家进餐结束 */
    /* 看一下左邻居现在是否能进餐 */
    /* 看一下右邻居现在是否能进餐 */
    /* 离开临界区 */
}

void test(i)                     /* i: 哲学家号码, 从 0 到 N - 1 */
{
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
        state[i] = EATING;
        up(&s[i]);
    }
}

```

图 2.20 哲学家进餐问题的解决方案

2.3.2 读者-写者问题

哲学家进餐问题对于多个竞争进程互斥地访问有限资源(如I/O设备)这一类问题的建模十分有用。另一个著名的问题是读者-写者问题(Courtois et al., 1971),它为数据库访问建立了一个模型。例如,设想一个飞机订票系统,其中有许多竞争的进程试图读写其中的数据。多个进程同时读是可以接受的,但如果一个进程正在更新数据库,则所有其他进程都不能访问数据库,即便是读操作也不行。这里的问题是如何对读者和写者进行编程。图2.21给出了一种解法。

```

typedef int semaphore;           /* 运用你的想象力 */
semaphore mutex = 1;           /* 控制对 'rc' 的访问 */
semaphore db = 1;               /* 控制对数据库的访问 */
int rc = 0;                     /* 正在读或想要读的进程数 */

void reader(void)
{
    while(TRUE){                /* 无限循环 */
        down(&mutex);          /* 排斥对 'rc' 的访问 */
        rc = rc + 1;             /* 又多了一个读者 */
        if(rc == 1)down(&db);    /* 如果这是第一个读者,那么…… */
        up(&mutex);             /* 恢复对 'rc' 的访问 */
        read_data_base();        /* 访问数据 */
        down(&mutex);          /* 排斥对 'rc' 的访问 */
        rc = rc - 1;             /* 读者又少了一个 */
        if(rc == 0)up(&db);      /* 如果这是最后一个读者,那么…… */
        up(&mutex);             /* 恢复对 'rc' 的访问 */
        use_data_read();         /* 非临界区操作 */
    }
}

void writer(void)
{
    while(TRUE){
        think_up_data();        /* 非临界区操作 */
        down(&db);              /* 排斥访问 */
        write_data_base();        /* 修改数据 */
        up(&db);                /* 恢复访问 */
    }
}

```

图2.21 一个读者与写者问题的解决方案

该解法中,第一个读者对信号量db执行down操作。随后的读者只是递增一个计数器rc。当读者离开时,它们递减这个计数器,而最后一个读者则对db执行up操作,这样就允许一个阻塞的写者(如果存在)访问数据库。

该解法在此处隐含了一条值得讨论的微妙规则。设想当一个读者在使用数据库时,另一个读者也来访问数据库,由于同时允许多个读者同时进行读操作,所以第二个读者也被允许进入,同理第三个及随后更多的读者都被允许进入。

现在假设一个写者到来,由于写操作是排他的,所以该写者不能访问数据库,而是被挂起。随后,又有读者出现,这样只要有一个读者活跃,随后而来的读者就都被允许访问数据库。这种策略

的结果是只要有读者陆续到来，它们一来就被允许进入，而写者将一直被挂起直到没有一个读者为止。假如每 2 秒钟来一个读者，而其操作时间为 5 秒钟，则写者将永远不能访问数据库。

为了防止这种情况，程序可以略做如下改动：当一个读者到来而正有一个写者在等待时，则读者被挂在写者后边，而不是立即进入。这样，写者只需等待它到来时就处于活跃状态的读者结束，而不用等那些在它后边到来的读者。这种解法的缺点是并发性较低，从而性能较差。Courtois 等人给出了一个写者优先的解法。详细内容请参阅他的论文。

2.4 进程调度

在前几节的例子中，我们经常遇到两个或多个进程（如生产者和消费者）在逻辑上均可以运行的情况。当计算机是多道程序设计系统时，通常就会有多个进程竞争 CPU。当多个进程处于就绪态而只有一个 CPU 时，操作系统就必须决定先运行哪一个进程。操作系统中做出这种决定的部分称为调度器（scheduler），它使用的算法称为调度算法（scheduling algorithm）。

许多进程调度的处理方式对进程和线程都适用。这里首先讨论进程调度问题，随后我们将考察一些线程调度专有的问题。

2.4.1 调度介绍

在早期以磁带上的卡片映像作为输入的批处理系统时代，调度算法很简单：依次运行磁带上的下一个作业。对于分时系统，调度算法要复杂一些，因为经常有多个用户等待服务，而且同时可能存在多个批处理流（如保险公司处理索赔）。读者可能会想在个人电脑上只有一个活动的进程。毕竟一个在文字处理软件上输入文档的用户不大可能同时编译一个后台程序。但是，还是会有后台任务，例如电子邮件守护进程发送和接收邮件。读者也可能会想近年来计算机速度已是如此之快，CPU 不再是什么稀缺的资源了。但是新的应用倾向于需要更多的资源，处理数字图像和观看实时录像就是如此。

进程行为

几乎所有的进程都交替地突发（磁盘）I/O 请求，如图 2.22 所示。典型地，CPU 不停顿地运行一段时间，然后发出一个系统调用读写文件。当系统调用结束时，CPU 又开始计算，直到它需要读更多的数据或写更多的数据为止。请注意，某些 I/O 活动可以视为计算。例如，当 CPU 向视频 RAM 复制数据以更新屏幕时，因为使用了 CPU，所以这是计算操作而非 I/O 操作。从这个意义上讲，当一个进程进入阻塞态等待外部设备完成工作而被阻塞时，才是 I/O 的执行过程。

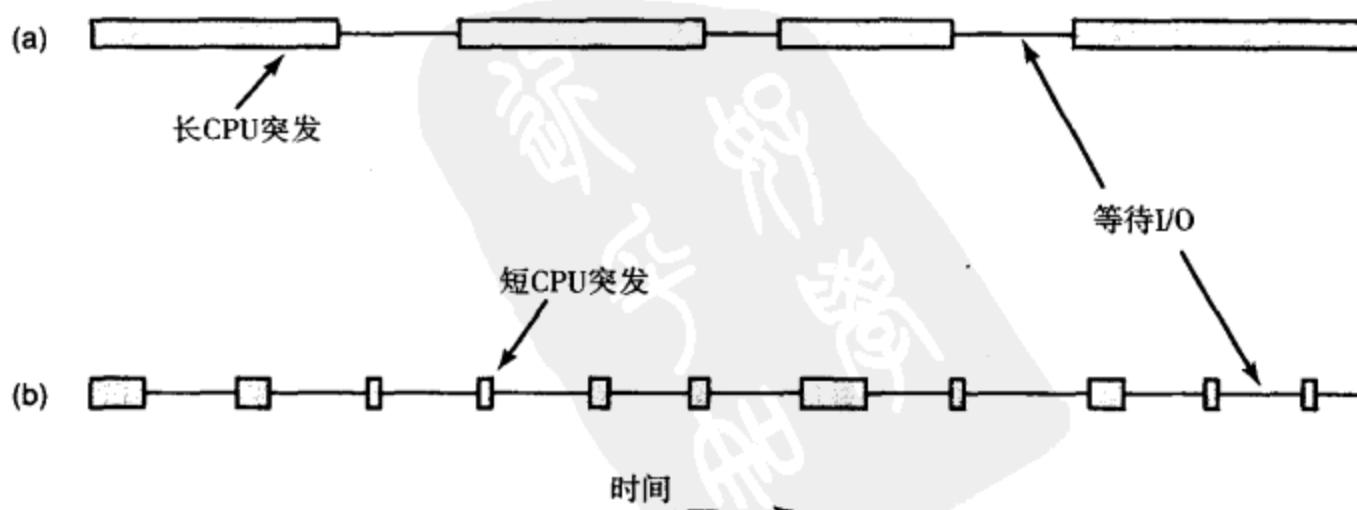


图 2.22 突发的 CPU 使用与周期性的等待 I/O 交替：(a)一个计算密集型进程；(b)一个 I/O 密集型进程

在图2.22中值得一提的是，有些进程（例如图2.22(a)中的进程）大部分时间花费在运算上，而有些进程（例如图2.22(b)中的进程）大部分时间花费在等待I/O操作上。前者称为**计算密集型**（compute-bound），而后者称为**I/O密集型**（I/O-bound）。计算密集型进程通常具有较长的CPU运算时间，因而具有较少频度的I/O操作。I/O密集型进程之所以是I/O密集型的，是因为它们在I/O请求之间需要较少的CPU运算，而不是因为它们拥有较长的I/O请求时间。在I/O请求后无论处理数据所花时间是多还是少，都花费同样多的时间读取磁盘块。

有必要指出，随着CPU变得越来越快，进程倾向于变成I/O密集型。这一现象之所以发生，是因为CPU发展的速度比磁盘存取发展的速度要快得多。结果，如何调度I/O密集型进程在将来可能是一个更加重要的课题。这里的基本思想是，如果需要运行I/O密集型进程，那么就应该让它尽快地得到机会，以便发出磁盘请求保持磁盘忙碌。

什么时候调度

调度有可能在很多情况下发生。首先，如下两种情况下调度肯定会发生：

1. 当一个进程退出时。
2. 当一个进程在I/O或信号量上阻塞时。

在这些情况下，刚刚运行的进程不再就绪，所以必须挑选另外一个进程运行。

另外还有其他三种情况，在这些情况下，尽管调度在逻辑上不是必需的，但还是经常发生：

1. 当一个新进程创建时。
2. 当一个I/O中断发生时。
3. 当一个时钟中断发生时。

当一个新的进程创建时，重新评估优先级是理所当然的。在一些情况下，父进程会为子进程申请一个不同的优先级。

当产生I/O中断时，这意味着I/O设备完成了它的工作，这时一些被I/O阻塞的进程就可能转为就绪态能够运行了。

时钟中断提供了一个判断当前运行进程是否运行了足够长时间的机会。根据如何处理时钟中断可以把调度算法分为两类。一类是非抢占式调度算法（non-preemptive scheduling algorithm），这种算法挑选一个进程运行，并一直运行到阻塞（可能是I/O阻塞或等待另外一个进程）或自愿退出。另一类是抢占式调度算法（preemptive scheduling algorithm），与前者不同，它挑选一个进程运行，这个进程所运行的最大时间是固定的。如果到了这个最大时间间隔进程仍在运行，进程将会被挂起，调度器将会挑选另外一个进程运行（如果这个进程存在的话）。抢占式调度器在这个时间间隔最后，需要一个时钟中断来获得对CPU的控制权。如果没有时钟，那么非抢占式中断就是唯一选择。

调度算法的分类

毫不奇怪，不同的环境下需要不同的调度算法。这是因为不同的应用领域（和不同种类的操作系统）有不同的目标。换句话说，在不同的系统中，调度器的优化目标是不同的。以下三种情况值得区分：

1. 批处理。
2. 交互式。
3. 实时。

在批处理系统中，没有用户在终端不耐烦地等待结果。因此，非抢占式调度算法和具有较长时间间隔的抢占式调度算法都是适用的。这种方法减少了进程切换，因而增强了系统性能。

在交互式环境中，为了不让一个进程霸占 CPU 使得其他进程得不到服务，抢占是必须的。尽管不会有进程故意不停地运行，但是由于错误，一个进程还是可能无限期地排斥其他进程。为了防止这种情况，仍然需要抢占。

说来也奇怪，在具有实时要求的系统中，有时不需要抢占，因为这些进程不会长时间地运行，而是运行一段时间就阻塞。和交互式系统不同，实时系统只运行那些用来推进现有应用的程序。而交互式系统是通用的，可能运行一些不友好的甚至是恶意的程序。

调度算法的目标

为了设计一个调度算法，应当首先明确一个好的调度算法必须做什么。一些目标是根据环境（批处理、交互式或实时）设定的，而另外一些目标是在各种情况下都适用的。图 2.23 中列举了一些调度算法设计目标，我们将按列举的顺序讨论。

所有系统
公平——给每个进程公平的 CPU 份额
策略强制执行——执行所规定的策略
平衡——保持系统所有的部分都忙碌
批处理系统
吞吐量——最大化每小时作业数
周转时间——最小化从提交到完成的时间间隔
CPU 利用率——保持 CPU 始终忙碌
交互式系统
响应时间——快速响应请求
均衡性——满足所有用户的需求
实时系统
满足截止时间——避免丢失数据
可预测性——在多媒体系统中避免失真

图 2.23 不同情况下调度算法的一些目标

公平在所有的情况下，都是重要的。可比较的进程应该获得可比较的服务。相对于处于同等地位的进程而言，给予一个进程更多的 CPU 时间是不公平的。当然，不同种类的进程应该得到不同的处理。可以考虑一下在核反应堆计算机中心安全控制与发放薪水处理之间的差别。

和公平稍微有关的是系统策略的强制执行。如果局部策略是安全地控制进程以便可以随时运行，即使这意味着晚 30 秒钟发薪，调度器也必须保证这一策略得到实施。

另一个共同目标是尽量保持计算机各部分忙碌。如果 CPU 和所有的 I/O 设备保持不停地运行，那么每秒钟的工作量就要比有一些设备不工作的情况下要大。例如，在批处理系统中，调度器控制着哪个作业被装入内存运行。让一些计算密集型进程和一些 I/O 密集型进程一起在内存中运行要比先加载和运行完毕计算密集型进程，再加载运行 I/O 密集型进程更好一些。在后一种情况下，当计算密集型进程一起运行时，它们会竞争 CPU 资源，而磁盘空闲。之后，当 I/O 密集型作业到来时，它们互相争夺磁盘资源，而 CPU 处于空闲。因此最好把这两种进程恰当地混合让系统一次运行。

公司运行批作业（处理保险索赔）的计算机中心的管理者们从三个维度看待计算机系统的性能：吞吐量、周转时间和 CPU 利用率。吞吐量是系统每秒钟所完成的作业数量。全面考虑的情况

下，每秒钟完成50个作业要比每秒钟完成40个作业要好。周转时间是作业提交到处理完毕所经过的平均时间间隔，它衡量了一个用户等待一个输出的平均时间。这里的规则是：时间越短越好。

一个最大化吞吐量的调度算法，不一定能最小化周转时间。例如，给了一些作业，有些作业需要较长的运行时间（长作业），而有些作业需要较短的运行时间（短作业）。如果调度器只挑选短作业运行，那么系统可能得到较大的吞吐量，但这是以牺牲长作业的周转时间为代价的。如果短作业以一个稳定的速率不断到达，那么长作业可能永远得不到处理，这就意味着在达到较高吞吐量的情况下，长作业的周转时间是无限大的。

在批处理系统中，另外一个重要因素是CPU利用率，因为在批处理系统的大型机上，CPU仍是主要的开支。所以如果CPU没有全部利用起来，那么计算机中心的管理者们就会感到内疚。尽管如此，CPU利用率并不是一个好的度量参数。真正有价值的是系统每小时完成的作业数（吞吐量），以及取回作业需要等待的时间（周转时间）。把CPU利用率用做度量依据，就像是以汽车引擎每秒钟的转数来评价汽车一样。

在交互式系统中，特别是分时系统和服务器中，则用不同的适用目标。最重要的一个设计目标是最小化响应时间，即命令从提交到得到结果所需的时间间隔。在一个有后台进程运行（如从网络中读取和存储电子邮件）的个人电脑中，启动程序或打开文件的用户请求应当优先于后台进程。让所有交互的请求优先处理将会得到一个较好的服务。

一个相关的问题是**均衡性**。用户对一件事情需要多长时间通常有一种固有的（但通常是不正确的）看法。当一个被认为是复杂的请求花很长时间时，用户可以接受；但如果被认为是很简单的一个请求也花了很长时间时，用户就要变得恼火了。例如，如果点击图标通过模拟调制解调器呼叫互联网服务提供商，要花费45 s建立连接，则用户可能把它作为一个无法更改的事实接受。相反，如果点击图标撤销这个连接也要花45 s，那么在等了30 s时用户大概就会喋喋不休地咒骂，45 s时可能已口吐白沫了。之所以这样，是因为用户一般认为拿起听筒并建立通话连接所需的时间要比挂掉电话所花的时间长。在一些情况下（如本例），调度器对响应时间无能为力，但在另外一些情况下，尤其是当延迟由不当地安排进程顺序引起的时候，调度器还是可以大有作为的。

与交互式系统相比，实时系统有不同的特性，因此也有不同的调度目标。实时系统的特点是它有必须或至少满足的截止时间。例如，计算机正在控制一台以一定频率生成数据的设备，如果不能及时地运行数据收集进程，则可能造成数据丢失。所以，实时系统中最重要的是满足所有（或绝大多数）的截止时间。

在实时系统中，特别是那些涉及多媒体的实时系统，可预测性非常重要。偶尔的一次不能满足截止时间要求并不严重，但是如果声音进程运行没有周期性，那么音质将急剧下降。视频也有同样的问题，但耳朵比眼睛对抖动更敏感。为了防止抖动，进程调度必须高度可预知和有规律性。

2.4.2 批处理系统中的调度

现在我们从一般调度问题转向特定调度算法。这一节将讨论批处理系统中用到的调度算法，接下来的章节还会讨论交互系统和实时系统。值得指出的是，一些算法既适合批处理系统也适合交互式系统。稍后将讨论这些问题。本节的重点是只适用于批处理系统的调度算法。

先到先服务

在所有的调度算法中，最简单的可能当属非抢占式先到先服务（first-come first-served）算法。在这个算法中，进程按照它们请求CPU的顺序使用CPU。基本上，有一个就绪进程的单一队列。早上，当一个作业从外部进入系统，它就马上运行并可以运行任意长时间。当其他作业到来时，就被

插入队列尾端。当运行的进程阻塞时，队列里的第一个进程调度运行，当这个阻塞的进程转为就绪时，它就像一个新到来的进程一样插入队尾。

这个算法的优点是易于理解，编程实现简单。就难于得到的体育或音乐会门票的分配问题，这对那些愿意在早上两点就去排队的人们也是公平的。在这个算法中，一个单向链表记录所有就绪的进程。加入一个新作业或解除阻塞一个进程，只需要把它们插入队尾。还有比这更简单的么？

不过，先到先服务也有明显的缺点。假设有一个计算密集型进程一次运行 1 s，另外有很多 I/O 密集型进程占用很少的 CPU 但每个都要读 1000 次磁盘来完成作业。I/O 密集型进程运行 1 s，然后去读一个磁盘块，这时所有的 I/O 进程开始运行并读磁盘。当计算密集型进程读到它需要的磁盘块时，它又运行 1 s，继而是 I/O 密集型进程接连快速运行。

最终结果是，I/O 密集型进程每秒读一个磁盘块，要花 1000 s 才能完成作业。如果使用一个每 10 ms 抢占一次计算密集型进程的调度算法，I/O 密集型进程就会需要 10 s 完成而不是 1000 s，而且这也没有对计算密集型进程造成太大影响。

最短作业优先

现在来看另外一种适用于运行时间可以预知的批作业的非抢占式调度算法。例如一家保险公司，因为每天都做类似的工作，所以人们可以相当精确地预测一个处理 1000 起索赔的作业需要多长时间。当输入队列中有若干个同等重要的作业将被启动时，调度器应使用**最短作业优先**算法。如图 2.24 所示，这里有 4 个作业 A, B, C, D，运行时间为 8 min, 4 min, 4 min, 4 min。若按图中的次序运行，则 A, B, C 和 D 的周转时间为 8 min, 12 min, 16 min 和 20 min，平均为 14 min。



图 2.24 一个最短作业优先调度的例子：(a)以初始的顺序运行 4 个作业；(b)以最短作业优先调度运行这 4 个作业

现在考虑使用最短作业优先算法。如图 2.24(b)所示。现在周转时间为 4 min, 8 min, 12 min 和 20 min，平均为 11 min。可以证明最短作业优先是最优的。考虑有 4 个作业的情况，其运行时间为 a, b, c, d 。第 1 个作业在时间 a 结束，第 2 个作业在时间 $a + b$ 结束，依次类推。平均周转时间为 $(4a + 3b + 2c + d)/4$ ，显然 a 对平均值影响最大，所以它应是最短作业，其次是 b ，再次是 c ，最后的 d 只影响它自己的周转时间。对于任意数目作业的情况，道理完全一样。

值得一提的是，最短作业优先算法只有在所有作业同时启动时才是最优的。作为一个反例，考虑从 A 到 E 五个作业，运行时间为 2, 4, 1, 1 和 1，它们的到达时间分别是 0, 0, 3, 3 和 3。起初，因为其他三个进程还没到来，只有进程 A 和 B 可以运行。使用最短作业优先算法，如果作业以 A, B, C, D, E 的顺序运行，则平均等待时间是 4.6。然而，如果作业以 B, C, D, E, A 的顺序运行，则平均等待时间是 4.4。

最短剩余时间优先

最短作业优先的抢占式版本是**最短剩余时间优先** (shortest remaining time next) 算法。使用这种调度算法，调度器总是挑选其剩余时间最短的那些进程运行。同样，在这里运行时间必须预知。当一个新作业到来时，它所需的总时间与当前运行进程的剩余时间进行比较，如果新作业需要比当前进程更少的时间来完成，那么当前进程被挂起，而新作业启动运行。这种方式可以使新到来的短作业得到较好的服务。

三级调度

从一定的角度来看，批处理系统允许三个层次的调度处理，如图 2.25 所示。当作业到来时，它们首先被放入存储在磁盘上的输入队列中。准入调度器（admission scheduler）决定哪些作业允许进入系统，其他的作业在被选中之前就保存在输入队列中。一个典型的准入控制算法是找一些计算密集型进程和 I/O 密集型进程混合在一起运行。另外一种方案是，短作业可以很快准入，而长作业必须等待。准入调度器可以随意地在输入队列中保留作业，如果它愿意的话，还可以让后来的作业先进入系统。

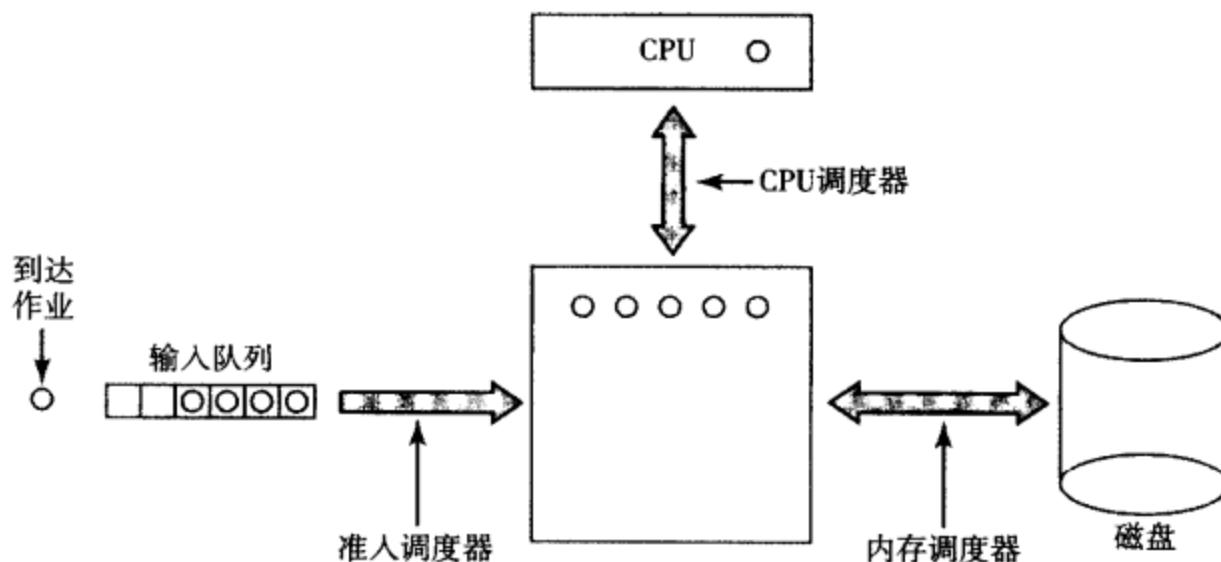


图 2.25 三级调度

一旦作业被允许进入系统，就可以为该作业创建进程并争夺 CPU 时间。然而，很有可能由于进程的数目过多，内存中没有足够的空间来容纳它们。在这种情况下，其中一些进程必须换出到磁盘上去。第二级调度器决定哪个进程留在内存，而哪个进程存入磁盘。这个调度器称为内存调度器，因为它决定了哪个进程留在内存而哪个进程换出到磁盘。

为了让磁盘里的进程也能得到服务，必须经常对上述决定进行重新评估。然而，由于进程由磁盘调度的内存代价很高，所以评估不应该超过每秒一次，或许更少一些。如果内存中的内容交换过于频繁，则会浪费大量的磁盘带宽，减慢文件 I/O 速度。

为了从整体上优化系统性能，内存调度器必须仔细考虑需要在内存中保留多少进程，这个数目称为多道程序的道数（degree of multiprogramming），并要考虑是什么类型的进程。如果调度器能够知道哪些进程是计算密集型的，哪些是 I/O 密集型的，它就可以尽量在内存中混合保留这两种进程。作为一个非常粗糙的估计，如果某类进程计算占大约 20% 的时间，为了让 CPU 保持忙碌，在内存中保留 5 个这样的进程大概是一个合适的数目。

为了进行决策，内存调度器周期性地评估磁盘上的每个进程，以便决定是否将它调入内存。可以用来做出决定的有以下标准：

1. 换入或换出已经经过了多长时间？
2. 最近占用了多少 CPU 时间？
3. 进程占用的空间有多大？（小的进程不这样考虑。）
4. 进程有多重要？

第三级调度实际在内存中选取下一个将要运行的进程。这通常称为 CPU 调度器，也是通常人们在谈论“调度器”时所指的那个。任何合适的抢占式或非抢占式调度算法都可以在 CPU 调度器中使用，这包括上面已经介绍的算法，以及下一节中将要介绍的算法。

2.4.3 交互式系统中的调度

下面看一下交互式系统中用到的调度算法。这些调度算法也可以用在批处理系统的调度器中。这里尽管三级调度不大可行，但两级调度（内存调度和CPU调度）是可行的也是常见的。下面我们将重点介绍CPU调度器和一些常见的调度算法。

时间片轮转调度

现在来看具体的调度算法。一种最古老、最简单、最公平且使用最广的算法是时间片轮转（round robin）调度。每个进程被分配一个时间段，称为它的时间片（quantum），即该进程允许运行的时间。如果在时间片结束时进程仍在运行，则CPU将被抢占并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。时间片轮转调度很容易实现。调度器所要做的就是维护一张就绪进程列表，如图2.26(a)所示；进程用完它的时间片后，就被移到队列的末尾，如图2.26(b)所示。



图2.26 时间片轮转调度：(a)就绪进程列表；(b)进程B用完时间片后的就绪进程列表

时间片轮转调度中有趣的一点是时间片的长度。从一个进程切换到另一个进程需要一定的时间（保存和装入寄存器值及内存映射，更新各种表格和队列，清空和重新调入高速缓存，等等）。假如进程切换（process switch）——有时称为上下文切换（context switch），需要1 ms，其中包括切换内存映射、清空和重新调入高速缓存等。再假设时间片设为4 ms。在这些参数设置下，做完4 ms有用的工作之后，CPU将花费1 ms来进行进程切换。CPU 20% 的时间浪费在了管理开销上，显然，这一开销太大了。

为了提高CPU效率，可以将时间片设为100 ms。这时浪费的时间只有1%。但是，在一个分时系统中，如果有10个交互用户几乎同时按下回车键，将发生什么情况呢？这时10个进程被挂在就绪队列中，如果CPU空闲，则立即启动第1个进程，第2个进程在大约100 ms之后启动，依次类推。假设所有其他进程都用完了它们的时间片，则最后一个进程不得不等待1 s才能获得运行机会。多数用户无法忍受一条简短命令要1 s才能做出响应。

另外一个因素是，如果时间片设置得比平均CPU突发时间长，抢占将会很少发生，取而代之的是，进程在用完时间片之前先被阻塞，从而引起进程切换。去除抢占可以提高系统性能，这样进程切换将只在逻辑需要时才发生，亦即当进程因为等待某事阻塞而不能运行时发生。

结论可以归结如下：时间片设得太短会导致过多的进程切换，降低了CPU效率；而设得太长又可能引起对短的交互请求的响应变差。将时间片设为20~50 ms通常是一个比较合理的折中。

优先级调度

时间片调度做了一个内在的假设，即所有的进程同等重要。而拥有和操作多用户计算机系统的人对此常有不同的看法。在一所有大学里，等级顺序可能是教务长、教授、秘书、勤务人员，最后是学生。这种将外部因素考虑在内的需要就导致了优先级调度（priority scheduling）。这里的基本思想很清楚：每个进程被赋予一个优先级，率先运行优先级最高的就绪进程。

即使在只有一个用户的PC机上，也可能有多个重要程度不同的进程。例如，一个在后台收发电子邮件的守护进程应被赋予一个较低的优先级，而在屏幕上实时播放电影的进程则应赋予较高的优先级。

为了防止高优先级进程无休止地运行下去，调度器可能在每个时钟节拍时（即每一个时钟中断）降低当前进程的优先级。如果这个动作导致其优先级低于次高优先级，则将进行进程切换。或者给每个进程设定一段它能够连续使用CPU的时间片，一旦这段时间用完，就运行次高优先级的进程。

优先级可以为静态或动态。在一台军用计算机上，将军启动的进程优先级为100，上校启动的进程优先级为90，少校启动的进程优先级为80，上尉启动的进程优先级为70，中尉启动的进程优先级为60，依次类推。或者在一个商业计算中心，高优先级作业的每小时费用为100美元，中等优先级作业的每小时费用为75美元，低优先级作业的每小时费用为50美元。UNIX系统中有一条命令*nice*，它允许用户为了照顾别人而自愿降低其进程的优先级，但从未有人使用过它。

优先级也可以被系统动态地确定，以达到某种目的。例如，有些进程为I/O密集型，其多数时间用来等待I/O结束。当这样的进程需要CPU时，它应被立即分配CPU，以便启动下一个I/O请求，这样就可以在另一个进程计算的同时执行I/O操作。使这类进程长时间等待CPU只会造成它无谓地长时间占用内存。使I/O密集型进程获得较好服务的一种简单算法是将其优先级设为 $1/f$ ， f 为该进程在上一时间片中用的比重。设一个进程的时间片为50 ms，如果进程只用了1 ms，那么它的优先级为50，而运行25 ms才被阻塞的进程的优先级则为2，全部时间都在计算的进程的优先级则为1。

很方便就可以将一组进程按优先级分成若干类。在各类之间采用优先级调度，而同类进程内部采用时间片轮转调度。图2.27显示了一个有4类优先级的系统，调度算法如下：只要存在优先级为第4类的就绪进程，就按照时间片轮转法使其运行一个时间片，此时不理会较低优先级的进程；若第4类进程为空，则运行第3类进程；若第4类、第3类均为空，则按时间片调度算法运行第2类进程。如果对优先级不经常进行调整，则低优先级进程很可能会饿死。

MINIX 3操作系统使用了一个类似于图2.27的调度算法，在默认的配置中，有16种不同的优先级。在MINIX 3中，操作系统的组件以进程的方式运行。MINIX 3把任务（I/O驱动）和服务器（内存管理器、文件系统和网络）设置在最高优先级类别中，每个任务或服务的初始优先级在编译时确定。慢速设备上的I/O会被授予比快速设备上的I/O甚至服务器更低的优先级。用户进程通常比系统进程具有较低的优先级，但所有进程的优先级在运行过程中可以改变。

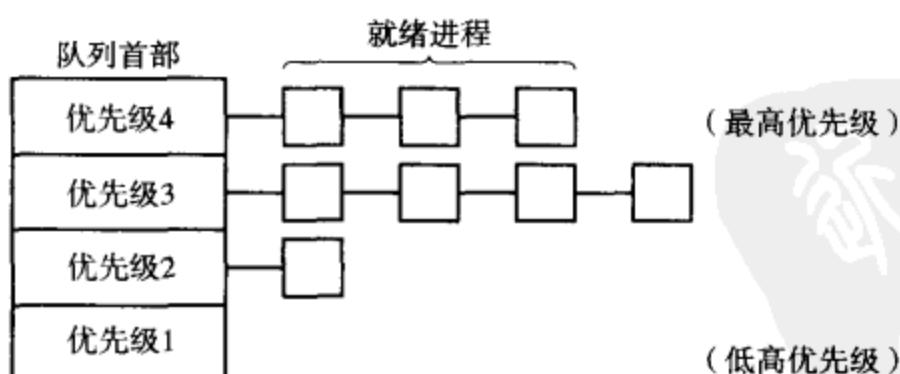


图2.27 一个有四类优先级的调度算法

多重队列

CTSS (Corbató et al., 1962) 是最早使用优先级调度的系统之一。但是CTSS存在进程切换速度太慢的问题，其原因是IBM 7094内存中只放得下一个进程，每次切换都需要将当前进程换出到磁盘，并从磁盘上读入一个新进程。CTSS的设计者很快便认识到为CPU密集型进程设置较长的时

间片，比频繁地分给它们很短的时间片要高效（减少交换次数）。另一方面，如前所述，给进程长时间片又会影响响应时间。他们的解决办法是设立优先级类。属于最高优先级类的进程运行1个时间片，属于次高优先级类的进程运行2个时间片，再次一级的进程运行4个时间片，依次类推。当一个进程用完分配的时间片后，它被移到下一类。

作为一个例子，考虑一个进程需要连续计算100个时间片。它最初被分配1个时间片，然后被换出。下次它将获得2个时间片，接下来分别是4, 8, 16, 32和64。当然，最后一次它只使用64个时间片中的37个便可以结束工作。该进程需要7次交换（包括最初的装入），而如果采用纯粹的时间片轮转则需要100次交换。而且，随着进程优先级的不断降低，它的运行频度逐渐放慢，从而为短的交互进程让出CPU。

对于那些刚开始运行一段长时间而后来又需要交互的进程，为了防止其优先级降低过快，可以采取这样的策略：只要终端上有回车键按下，则属于该终端的所有进程都被移到最高优先级，这样做的原因是认为此时进程即将需要交互。但可能有一天，一台负载重的机器上有几个用户偶然发现，只需坐在那里每过随机的几秒钟敲一下回车键就可大大提高响应速度，于是他又告诉所有的朋友。这件事的结论是：实践中行得通比理论上可行要困难得多。

已经有许多其他算法可用来将进程划分为优先级类。例如，在伯克利制造的著名的XDS 940系统中（Lampson, 1968），有4个优先级类，分别是终端、I/O、短时间片和长时间片。当一个等待终端输入的进程最终被唤醒时，它转到最高优先级类（终端）。当一个等待一块磁盘数据的进程就绪时，它将转到第2类。当进程在时间片用完时仍为就绪时，它被放入第3类。但如果一个进程已经多次用完时间片而从未因终端或其他I/O阻塞，它将被转入最低优先级类。许多其他系统也使用类似的算法，以侧重交互用户和交互进程，而牺牲后台进程。

最短进程优先

由于最短作业优先在批处理系统中常常伴随产生最短平均响应时间，所以如果它同时能够被用于交互进程，那将是非常好的。在某种程度上，它的确可以做到这一点。交互进程通常遵循下列模式：等待命令，执行命令，等待命令，执行命令，如此不断反复。如果将每一条命令的执行视为一个独立的作业，则可以通过首先运行最短的作业来使响应时间最短。唯一的问题是如何从当前的就绪进程中找出最短的那个。

一种办法是根据进程过去的行为进行推测，并执行估计运行时间最短的一个。假设某终端上每条命令的估计运行时间为 T_0 ，现在假设测量到其下一次运行时间为 T_1 ，可以用这两个值的加权和来改进估计时间，即 $aT_0 + (1 - a)T_1$ 。通过选择 a 的值，可以决定是尽快忘掉老的运行时间，还是在一段长时间内还记住它们。当 $a = 1/2$ 时，可以得到如下序列：

$$T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2$$

很容易看到，三轮过后， T_0 在新的估计值中所占的比重下降到了 $1/8$ 。

这种通过将当前测量值和先前估计值进行加权平均而得到下一个估计值的技术有时称为老化（aging）。它适用于许多预测值必须基于先前值的情况。老化算法在 $a = 1/2$ 时特别容易实现，只需将新值加到当前估计值上然后除以2（即右移一位）。

保证调度算法

一种完全不同的调度算法是向用户做出明确的性能保证，然后去实现它。一种很实际并很容易实现的保证是：若读者工作时有 n 个用户登录，则读者将获得CPU处理能力的 $1/n$ 。类似地，在一个有 n 个进程运行的单用户系统中，若所有的进程都平等，则每个进程将获得 $1/n$ 的CPU时间。

为了实现这个保证，系统必须跟踪每个进程从创建起已使用了多少 CPU 时间。然后它计算每个进程应获得的 CPU 时间，即自从创建以来的时间除以 n 。由于每个进程实际获得的 CPU 时间是已知的，所以很容易计算出真正获得的 CPU 时间和应获得的 CPU 时间之比。比率为 0.5 说明一个进程只获得了应得时间的一半，而比率为 2.0 则说明它获得了应得时间的 2 倍。于是该算法随后转向比率最低的进程，直到该进程的比率超过次最低进程为止。

彩票调度算法

尽管向用户做出承诺并履行它是一个好主意，但实现起来却很困难。不过有另一种算法可以给出类似的可预见的结果，而且实现起来简单许多，这种算法称为彩票调度法（lottery scheduling）（Waldspurger and Weihl, 1994）。

其基本思想是为进程发放针对系统各种资源（如 CPU 时间）的彩票。当调度器需做出决策时，随机选择一张彩票，持有该彩票的进程将获得系统资源。对于 CPU 调度，系统可能每秒钟抽 50 次彩票，每次的中奖者获得 20 ms 的运行时间。

George Orwell 对此的解释为：“所有进程都是平等的，而某些进程则需要更多的机会。”更重要的进程被给予更多的额外彩票，以增加其中奖机会。如果共发出 100 张彩票，而一个进程持有 20 张，它就有 20% 的中奖概率，对于长时间运行，它将获得大约 20% 的 CPU 时间。彩票算法与优先级调度完全不同，在后者中很难说清楚优先级为 40 说明了什么，而在前者中则很清楚：进程拥有多少彩票份额，它就将获得多少资源。

彩票调度法有几点有趣的特性。例如，如果一个新进程创建并得到一些彩票，则在下次抽奖时，它中奖的机会与其持有的彩票数成正比。换言之，彩票调度的反应非常迅速。

合作进程如果愿意，可以交换彩票。例如，一个客户进程向服务器进程发送一条消息并阻塞，它可以把所有的彩票都交给服务器进程，以增加其下一次被运行的机会。当服务器进程结束后，它又将彩票交还给客户进程以使其能够再次运行。实际上，在没有客户时，服务器根本不需要彩票。

彩票调度能用来解决其他算法难以解决的问题。例如，一个视频服务器，其中有若干个进程在不同的帧频下将视频信息传送给各自的客户。假设其分别需要 10 帧/秒、20 帧/秒和 25 帧/秒的速度，则通过为这些进程分别分配 10 张、20 张和 25 张彩票，它们将自动地按照正确的比例 10 : 20 : 25 分配 CPU 资源。

公平分享调度

到目前为止我们一直假设每个进程都基于其自身调度，而没有考虑进程的拥有者是谁。结果，如果用户 1 启动了 9 个进程而用户 2 启动了 1 个进程，使用轮转调度或相同优先级调度，那么用户 1 将得到 90% 的 CPU 时间，而用户 2 只得到了 10% 的 CPU 时间。

为了防止这种情况发生，一些系统在调度进程时首先考虑进程的拥有者是谁。在这个模型中，每个用户分得 CPU 时间的一部分，调度器以保证这一策略实施的方式挑选进程。因此，如果两个用户都得到了获得 50% 的 CPU 时间的保证，那么无论一个用户拥有多少进程，它都得到应得的 CPU 份额。

举例来说，考虑一个只有两个用户的系统，每个用户都得到了获取 50% 的 CPU 时间的保证。用户 1 有 A, B, C, D 四个进程，用户 2 只有一个进程 E。如果采用轮转调度，则一个满足约束条件的调度序列可能是

A E B E C E D E A E B E C E D E...

另一方面，如果授予了用户 1 的 CPU 时间是用户 2 的两倍，则调度序列可能是

A B E C D E A B E C D E...

当然还有许多其他可能可以进一步探讨，这取决于如何定义公平的含义。

2.4.4 实时系统调度

实时系统是那些时间因素非常关键的系统。例如，一个或多个外设发出信号，计算机必须在一段固定时间内做出适当的反应。一个实例是，计算机用 CD-ROM 放音乐时从驱动器获得二进制数据，并且必须在很短的时间内将其转换成音乐。如果这中间计算花的时间太长，音乐听起来就会失真。其他实时系统还包括医院里特护病房的监控系统、飞行器中的自动驾驶仪以及自动化工厂中的机器人控制等。在这些系统中，迟到的响应即使正确，也和没有响应一样糟糕。

实时系统通常分为硬实时（hard real time）系统和软实时（soft real time）系统。前者意味着存在必须满足的时间限制；后者意味着偶尔超过时间限制是可以容忍的。在这两种系统中，实时性的获得是通过将程序分成许多进程，而每个进程的行为都预先可知。这些进程通常生存周期都很短，往往在一秒内便运行结束。当检测到一个外部事件时，调度器按满足它们最后期限的方式调度这些进程。

实时系统要响应的事件进一步分为周期性（每隔一段固定的时间发生）和非周期性（在不可预测的时间发生）。一个系统可能必须响应多个周期的事件流。根据每个事件需要多长的处理时间，系统可能根本来不及处理所有事件。例如，有 m 个周期性事件，事件 i 的周期为 P_i ，其中每个事件 i 需要 C_i 秒的 CPU 时间来处理。则只有满足条件

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

时，才可能处理所有的负载。满足该条件的实时系统称为是可调度的（schedulable）。

举例来说，一个软实时系统处理三个事件流，其周期分别为 100 ms, 200 ms 和 500 ms。如果事件处理时间为 50 ms, 30 ms 和 100 ms，则这个系统是可调度的，因为 $0.5 + 0.15 + 0.2 < 1$ 。如果加入周期为 1 s 的第四个事件，则只要其处理时间不超过 150 ms，该系统仍将将是可调度的。这个运算的隐含条件是上下文切换的开销很小，可以忽略。

实时调度算法可以是静态的或动态的。前者在系统启动之前完成所有的调度决策，后者在运行时做出调度决策。要使用静态调度算法，必须预先知道足够多的需要做的工作和必须满足的约束时间信息。动态调度算法没有这个限制。

2.4.5 策略与机制

截至目前，这里隐含地假设了系统中的所有进程分属不同的用户，并且相互间竞争 CPU。尽管通常就是这样，但有时会有这样的情况：一个进程有许多子进程，这些子进程在其控制之下运行。例如，一个数据库管理系统可能有许多子进程，每个子进程可能处理不同的请求，或者每个子进程实现不同的功能（请求的语法分析、访问磁盘等）。主进程完全可能知道哪个子进程最重要（或最紧迫），哪个最不重要。但遗憾的是，以上讨论的调度算法中没有一个从用户进程接受有关的调度决策信息。这就导致调度器很少能够做出最优的选择。

解决该问题的方法是将调度机制（scheduling mechanism）和调度策略（scheduling policy）分开。亦即将调度算法以某种形式参数化，而参数可以由用户进程来填写。再来看数据库的例子。假设内核使用优先级调度算法，但提供一条系统调用，一个进程可以使用它来设置或改变其子进程的

优先级。这样，尽管父进程本身并不进行调度，但它可以控制子进程如何被调度的细节。在这里，机制位于内核，而策略则由用户进程设定。

2.4.6 线程调度

当若干个进程都有多个线程时，就有两个层次的并行：进程级和线程级。在这样的系统里，调度的处理有很大的差别，这取决于系统支持的是用户级线程还是内核级线程，还是两者都支持。

首先来考虑用户线程。因为内核感觉不到线程的存在，所以还像以前一样操作，选取一个进程，例如A进程，允许它在自己的时间片内取得控制权。A进程内部的线程调度器决定哪个线程运行，例如A1。因为没有时钟中断来使线程多道运行，故这个线程可以运行任意长时间，当它用完了进程的整个时间片后，内核将挑选另外一个进程运行。

当进程A再一次投入运行时，线程A1将继续运行，在A1运行结束前，它可能总是消耗掉进程A的全部时间片。不过，它这种不合群的行为并不影响其他进程。不管A进程内部发生什么，其他进程都会得到调度器所分配的合适时间份额。

现在来考虑进程A内的线程计算工作比较少的情况，例如，50 ms的时间片中只有5 ms的计算工作。结果，每个线程运行一小段时间，然后把CPU控制权交给了线程调度器。这样在内核切换到进程B之前，可能的运行序列是A1, A2, A3, A1, A2, A3, A1, A2, A3, A1。这种情形如图2.28所示。

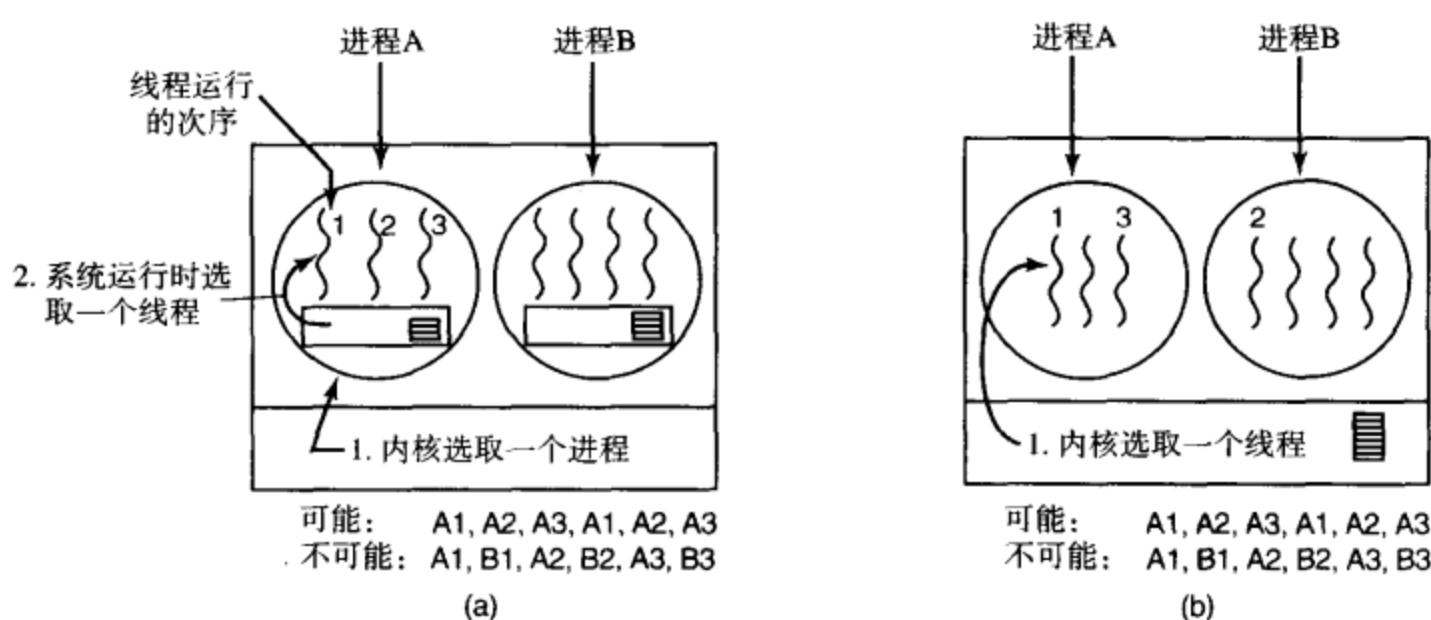


图 2.28 (a)在每个进程时间片为 50 ms、每个线程运行占用 5 ms 的情况下，用户级线程的可能调度情况；(b)在与(a)同样的情况下，内核级线程的可能调度情况

系统运行时可以使用以前描述的任意一种调度算法。实际上，轮转调度和优先级调度是最常用的两种。存在的唯一约束是，没有时钟中断来打断运行了足够长时间的线程。

现在来考虑内核级线程调度的情况。此时内核挑选线程运行，它不需要考虑这个线程属于哪个进程，但如果想的话它也可以知道。线程被赋予一个时间片，如果线程运行时间超过了这个时间片，将会被强制挂起。如果时间片是 50 ms，但运行 5 ms 就阻塞，则 30 ms 内线程运行的可能顺序是 A1, B1, A2, B2, A3, B3。同样是这些参数，在用户线程状态下得到这一序列是不可能的。这种情形如图 2.28(b)所示。

用户级线程和内核线程的一个主要区别是它们表现出来的性能。用户线程进行线程切换只需要几条指令，而内核线程需要完整的上下文切换，修改内存映射，使高速缓存失效，这将比用户线程切换慢几个数量级。另一方面，在使用内核线程时，一个线程的阻塞不会导致整个进程阻塞，而如果在用户线程下则会导致整个线程阻塞。

内核知道从进程 A 内的一个线程切换到进程 B 内的一个线程比切换到 A 内的另外一个线程有更大的开销，因为前者将进行内存映射切换并清除高速缓存，当内核考虑线程切换时可以考虑这个因素。例如，假设有两个其他方面同等重要的线程，一个线程和刚刚运行被阻塞的线程属于同一进程，另外一个属于其他进程，那么将优先考虑前者。

另一个重要因素是用户线程可作为特定应用定制的调度器。例如，考虑一个 Web 服务器，内部有一个分派线程负责接收请求并把这些请求分派给工作线程。假设有一个工作线程阻塞，分派线程和另外两个工作线程处于就绪态。那么下一个线程将运行哪一个呢？由于运行时系统了解每个线程的作用，它可以很容易地挑选出分派线程运行，从而使阻塞的线程能够运行。在工作线程经常被磁盘 I/O 阻塞的环境中，这一策略可以使并行最大化。在内核线程中，内核从来不了解每个线程的工作（尽管它可以为每个线程指定不同的优先级）。不过，应用定制的线程调度器一般比内核更能满足应用需要。

2.5 MINIX 3 进程概述

在研究了进程管理、进程间通信以及进程调度的原理之后，现在我们来看这些原理在 MINIX 3 中的应用。MINIX 3 与 UNIX 不同，UNIX 的内核是一个不分模块的单块程序，而 MINIX 3 本身就是一组进程的集合。它们相互之间以及与用户进程之间使用进程间通信机制（简单的消息传递）来通信。这种设计使得 MINIX 3 的结构更加模块化和灵活。例如，这使得将整个文件系统替换成另一个完全不同的文件系统变得很容易，而无须重新编译内核。

2.5.1 MINIX 3 的内部结构

作为开始，首先我们大致了解一下整个 MINIX 3 系统。MINIX 3 被组织成四层，每一层执行一套定义明确的功能。这四层示于图 2.29。

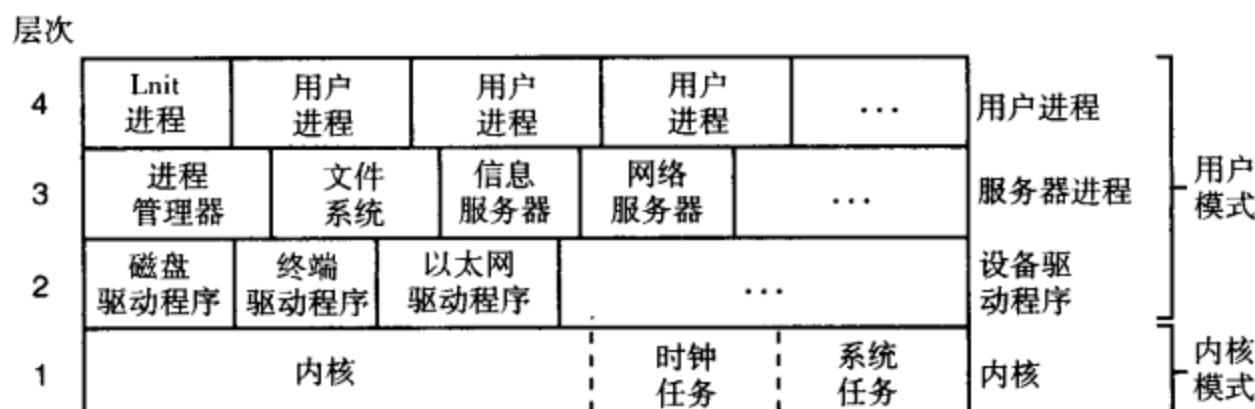


图 2.29 MINIX 3 的四层结构

内核最底层进行进程调度，并负责进程在图 2.2 所示的就绪态、运行态和阻塞态间的转换。内核还负责处理所有进程间的消息。消息处理需要核实目标的合法性，定位内存中的发送和接收缓存，把消息从发送缓存复制到接收缓存。内核的另一部分支持对 I/O 端口和中断的访问，在现代处理器中这需要使用特权内核模式（kernel mode）指令，这些指令在用户进程中是没有的。

除了内核本身之外，这一层还有两个类似设备驱动程序功能的模块。从能够与产生时钟信号的硬件交互的角度讲，时钟任务（clock task）是一个 I/O 设备驱动程序，但是不像磁盘或通信总线驱动程序，它只提供内核接口，用户不能访问。

第1层的主要功能是为上层驱动程序和服务器提供一组特权内核调用。这包括读写I/O端口、跨地址空间复制数据等。这些调用由系统任务（system task）实现。尽管系统任务和时钟任务被编译进了内核地址空间，但它们作为单独的进程调用，并有自己的调用堆栈。

大部分内核程序和所有的时钟任务及系统任务程序用C语言编写。内核中有一小部分程序是用汇编语言编写的。汇编语言编写的部分负责中断处理、进程切换的底层上下文管理机制（保存和恢复寄存器等）、MMU硬件的底层操作部分。总体上讲，汇编语言负责内核直接与硬件交互的最底层部分，这部分不能用C实现。如果把MINIX 3移植到新的体系结构上，那么这部分代码需要重写。

内核以上的其他三层也可以视为一层，因为内核从根本上同样对待它们。每一层只限于使用用户模式指令，并且每一层由内核调度执行。它们都不能直接对I/O端口进行操作。此外，它们也不能访问分配给它们之外的内存。

然而，进程都潜在具有一定的特权（例如进行系统调用的能力）。这是第2层、第3层和第4层内的进程的真正不同之处。第2层内的进程拥有最多的特权，第3层内的进程的特权少一些，而第4层内的进程没有特权。例如，处于第2层内的称为设备驱动程序（device drivers）的进程，可以请求系统任务代表它们从I/O端口读数据或向I/O端口写数据。每种类型的设备，包括磁盘、打印机、终端和网络接口，都需要一个设备驱动程序。如果还有其他类型的I/O设备，每种也需要一个驱动程序。设备驱动程序也可能发出其他内核调用，例如请求刚刚读取的数据副本到另一个进程的地址空间。

第3层包含了服务器，即向用户进程提供有用服务的进程。有两个服务器是必不可少的。进程管理器（Process Manager, PM）执行所有涉及启动或终止进程的MINIX 3系统调用，例如fork, exec和wait等，还负责执行与信号有关的系统调用，例如alarm和kill等，这些调用能够改变进程的执行状态。进程管理器还负责管理内存，例如发出brk系统调用。文件系统（File System, FS）负责执行文件系统的调用，如read, mount和chdir等。

理解内核调用（kernel call）和POSIX系统调用（system call）的区别非常重要。内核调用是由系统服务提供的以使驱动程序和服务器完成工作的低层函数。一个典型的内核调用是读硬件的I/O端口。相反，POSIX系统调用是由POSIX规范定义的高层调用，如read, fork和unlink等，这些调用供第4层的用户程序使用。用户程序内包含了许多POSIX调用，但不包含内核调用。实现这些调用的机制是相似的，并且内核调用可以视为系统调用的一个子集。

在第3层中除了进程管理器和文件系统外，还有其他一些服务。它们执行MINIX 3特定的功能。有把握认为进程管理器和文件系统的功能在其他任何操作系统上都可以找到。信息服务器（information server, IS）负责提供其他驱动程序和服务器的调试和状态信息的工作，而这一工作在像MINIX 3这样的专门为实验而设计的系统中是非常重要的，而在用户不能修改的商业操作系统中，这一工作没那么重要。再生服务器（reincarnation server, RS）启动或重启那些不与内核一起加载到内存的设备驱动程序。另外，如果驱动程序在操作过程中失败，那么再生服务器检测到这个失败，并杀死这个驱动程序（如果它没有死的话），重启一个驱动程序的新的副本，从而使整个系统具有更好的容错能力。这一功能在大多数操作系统中是没有的。在一个网络操作系统上，第3层中还可能有网络服务器（network server, inet）。服务器不能直接进行I/O操作，但它们可以通过驱动程序来请求I/O。服务器也可以通过系统任务与内核通信。

就像在第1章开始讲到的那样，操作系统负责两件事情：管理资源和通过实现系统调用提供一个扩展的计算机。在MINIX 3操作系统中，资源管理主要由第2层的驱动程序来做，当需要操作I/O或系统中断时，还可以得到内核层的帮助。系统调用的解释由第3层的进程管理器和文件系统来做。文件系统作为一个文件服务器设计仔细，改动很少就可以移到一个远程的计算机上。

为了增加新的服务器，系统并不需要重新编译。进程管理器和文件系统可以由网络服务器和其他服务器补充（或代替），这可以通过当MINIX 3启动时或启动后向系统添加另外的服务器来实现。设备驱动程序通常在系统启动时开始运行，但也可以稍后启动运行。设备驱动程序和服务器都被编译并以一般可执行文件的形式存储在磁盘上，但是当启动以后，如果需要的话，它们可以操作特权指令（它们被赋予必要的优先权）。一个称为服务（service）的用户程序提供与再生服务器的接口。尽管驱动程序和服务器作为独立的进程存在，但与用户进程不同的是，当系统处于活动状态时驱动程序和服务器进程不会终止。

第2层和第3层内的驱动程序和服务器通常称为系统进程。系统进程是操作系统的一部分。它们不属于某一个特定的用户，并且大部分（如果不是全部的话）会在系统第1个用户登录前启动。系统进程和用户进程的另一个不同之处是，系统进程比用户进程具有更高的执行优先级。事实上，通常驱动程序比服务器具有更高的执行优先级，但这也不是绝对的。MINIX 3中的执行优先级是视情况而定的，一个慢设备的驱动程序很可能授予比必须快速响应的服务器更低的优先级。

最后，第4层包括了所有的用户进程，如shell程序、编辑器、编译器和用户的*a.out*程序。许多用户进程随着用户的登录、工作和退出而创建和终止。一个运行系统中通常有一些进程在系统引导时启动，并一直运行。*init*进程就是其中的一个，下一节将介绍这个进程。另外还可能有一些守护进程在运行。守护进程（daemon）是周期性运行或总是等待某个事件（例如网络上一个包的到达）的后台进程。从某种意义上说，守护进程是单独启动并作为一个用户进程运行的服务器。像真的在启动阶段创建的服务进程一样，可以给守护进程设置一个比一般用户进程高的优先级。

有必要解释一下任务（task）和设备驱动程序（device driver）这两个术语。在MINIX的旧版本中，所有的设备驱动程序和内核编译在一起，这给了它们访问内核数据结构和互相访问的能力。它们还都可以直接访问I/O端口。我们把这些设备驱动程序称为任务以区分那些纯粹独立的用户空间进程。在MINIX 3中，设备驱动程序完全在用户空间实现。唯一的例外是时钟任务，从驱动程序可以由用户程序通过设备文件访问的角度讲，它不是一个设备驱动程序。在正文中，使用术语“任务”时，专指时钟任务或系统任务。当称呼用户空间的设备驱动程序时，“任务”一词全部换成了“设备驱动程序”。然而，程序源代码中的函数名、变量名以及注释还没有更新。所以当读者在学习中看MINIX 3源代码时，实际指“设备驱动程序”的地方可能用的是“任务”一词。

2.5.2 MINIX 3 中的进程管理

MINIX 3中的进程遵从本章前面所描述的通用进程模型。进程可以创建子进程，子进程又可以创建更多的子进程，这样便构造出一棵进程树。实际上，整个系统中所有的用户进程都属于以*init*（见图2.29）为根节点的一棵进程树。当然，服务器和驱动程序是其中的特例，因为它们其中的一些必须在所有用户进程（包括*init*）启动前启动。

MINIX 3 的启动

操作系统是如何启动的呢？在下面将用几页的篇幅概述MINIX 3的启动顺序。要了解其他操作系统的启动过程，可以参考Dodge et al. (2005)。

在大多数具有磁盘设备的计算机中，都会设置好启动盘的搜索顺序。如果在第1个软盘驱动器中有软盘，那么这张软盘将作为启动盘。如果没有软盘，而第1个CD-ROM驱动器中有一张光盘（CD-ROM），那么这张光盘将作为启动盘。如果既没有软盘又没有光盘，那么第1个硬盘将作为启动盘。这个顺序可以在计算机接通电源之后，及时进入BIOS而重新设置。其他的设备，尤其是可移动存储设备也可能得到支持用来作为启动盘。

当计算机开机时，如果启动设备是一个软盘，那么硬件从引导盘上将第1道第1个扇区读入内存并从那里开始执行。在软盘上，第1扇区包含了引导程序（bootstrap）。引导程序很小，因为它必须能容纳在一个扇区（512字节）里。MINIX 3引导程序装入一个更大的程序boot，由boot装入操作系统。

相比之下，硬盘启动需要一个中间步骤。硬盘被分成若干个分区（partition），整个硬盘的第1个扇区包括一段小程序和磁盘分区表（partition table），通常这两者合在一起称为主引导记录（master boot record）。程序部分被执行以读入分区表并选择活动分区（active partition）。活动分区的第1个扇区有一个引导程序，它随后被装入并执行以查找并启动程序boot，这与从软盘引导完全相同。

在计算机发展史中，光盘的出现比硬盘和软盘要晚一些，当对光盘引导的支持出现时，它能够加载不只一个扇区。支持从光盘启动的计算机可以马上把一大块的数据加载到内存。通常从光盘载入的恰恰是可引导软盘的一个副本，这个副本被放在内存中并作为RAM disk。进行完第1步之后，控制权交给RAM disk，引导就像软盘是启动设备一样继续进行。在一台仅有光盘驱动器但不支持从光盘启动的老式计算机上，可以将可引导软盘映像（bootable floppy disk image）复制到软盘，然后这张软盘用来作为启动盘。当然光盘驱动器中必须有光盘，因为可引导软盘映像期望这样。

不管哪种情况，MINIX 3的boot程序将在软盘或硬盘分区上找一个包含多个部分的文件，并将各部分装到内存的适当位置。这一文件就是引导映像（boot image）。其中最重要的部分是内核（包括时钟任务和系统任务）、进程管理器、文件系统。另外，必须有一个磁盘驱动程序作为引导映像的一部分加载。在引导映像中还有很多其他的程序，这其中包括再生服务器、RAM disk、控制台、日志驱动程序和init程序。

这里必须特别强调引导映像的各个部分是独立的程序。当必需的内核、进程管理器和文件系统加载以后，其他许多部分可以分别加载。有一个例外是再生服务器。它必须是引导映像的一部分。它可以把一个初始化后加载的普通进程赋予特定的优先级和权限，使得它们成为系统进程。它也可以根据驱动程序的名字重新启动崩溃的驱动程序。如前所述，至少有一个磁盘驱动程序。如果根文件系统（root file system）要复制到RAM盘，还需要有内存驱动程序，否则，内存驱动程序可以稍后加载。控制台和日志驱动程序在引导映像中是可选的。尽早加载它们是为了在进程启动过程中在控制台显示信息和记录日志。init进程当然可以稍后加载，但是由于它控制着系统的初始配置信息，所以最好在引导映像中直接包括它。

这个启动过程并不简单。所有那些属于磁盘任务和文件系统范围的操作在这两部分被激活之前都要由boot完成。在后边将回到MINIX 3如何启动这一主题。现在我们只需知道一旦装入操作完成，内核便开始运行。

在其初始化阶段，内核先启动系统任务和时钟任务，再启动进程管理器和文件系统。然后进程管理器和文件系统合作加载作为引导映像一部分的其他服务器和驱动程序。当所有这些都开始运行并完成初始化之后，它们将阻塞，等待执行某种操作。MINIX 3调度按优先级区分的进程。只有包含在引导映像中的所有任务、驱动程序和服务器都阻塞之后，第1个用户进程init才开始运行。和引导映像一起加载或在启动过程中加载的系统组件如图2.30所示。

进程树的初始化

Init是第1个用户进程，也是作为引导映像的一部分加载的最后一个进程。读者可能会想，一旦init进程运行以后，就会生成一棵像图1.5那样的进程树，但并非完全这样。在一个传统的操作系统中，这可能是正确的，但MINIX 3操作系统有所不同。首先，在init进程运行之前，就有一些系统进程已经运行了。在内核中运行的时钟任务和系统任务很特别，它们在内核外是不可见的。它们没有PID，也不作为任何进程树的一部分。进程管理器是用户空间内运行的第一个进程，它被赋

予 PID 为 0，它既不是其他任何进程的子进程，也不是其他任何进程的父进程。再生服务器被作为其他所有在引导映像中启动的进程的父进程（例如驱动程序和服务器）。这里的逻辑是，当其他这些程序需要重启时，应该通知的是再生服务器进程。

组件	描述	通过什么加载
kernel	内核+时钟和系统任务	(在引导镜像中)
pm	进程管理器	(在引导镜像中)
fs	文件系统	(在引导镜像中)
rs	重启服务器和驱动程序	(在引导镜像中)
memory	RAM 盘驱动程序	(在引导镜像中)
log	缓存日志输出	(在引导镜像中)
tty	终端和键盘驱动程序	(在引导镜像中)
driver	磁盘 (bios 或软盘) 驱动程序	(在引导镜像中)
init	所有用户进程的父进程	(在引导镜像中)
floppy	软盘驱动程序 (如果从硬盘启动)	/etc/rc
is	信息服务器 (用于调试转储)	/etc/rc
cmos	读取 CMOS 时钟来设置时间	/etc/rc
random	随机数产生器	/etc/rc
printer	打印机驱动程序	/etc/rc

图 2.30 MINIX3 中一些重要的系统组件，其他的诸如网络驱动程序和 inet 服务器组件也可能存在

稍后我们将会看到，即使在 *init* 启动运行以后，MINIX 3 中进程树的创建与传统概念也有所不同。在类 UNIX 系统中，*init* 被赋予的进程标识符为 1。在 MINIX 3 中，尽管 *init* 不是运行的第一个进程，但是仍为它保留着进程标识符 1。像引导映像中所有其他用户进程一样（进程管理器除外），*init* 进程被作为再生服务器进程的一个子进程。在标准的类 UNIX 系统中，*init* 进程首先执行 /etc/rc 脚本。这一脚本启动其他不在引导映像中的驱动程序和服务器。所有通过 rc 脚本启动的进程都是 *init* 进程的一个子进程。首先启动的程序中有一个实用程序称为服务 (*service*)。正如所料，服务本身作为 *init* 进程的一个子进程，但是还是与传统的概念有所不同。

服务是再生服务器进程的用户接口。再生服务器启动一个普通程序并将它转化为系统进程。它启动 *floppy*（如果在启动系统过程中使用的话）、*cmos*（用来读取实时时钟）和 *is*，即信息管理器，用来管理当在控制台键盘上按下功能键 (F1, F2 等) 时所产生的调试转储信息。再生服务器进程收养所有的系统进程（进程管理器除外）作为它的子进程。

当 *cmos* 设备驱动程序启动以后，rc 脚本就可以初始化实时时钟了。此时，所有需要的文件都必须在根设备上能够找到。需要的驱动程序和服务器起初位于 */sbin* 目录下，其他启动需要的命令位于 */bin* 目录下。当初始启动步骤完成以后，就可以加载诸如 */usr* 之类的文件系统了。rc 脚本的一个重要功能是检查可能由上次系统崩溃所引起的文件系统错误。检查非常简单，当系统关闭是通过执行 *shutdown* 命令时，在登录历史文件 */usr/adm/wtmp* 中将会记录一条信息。*shutdown -C* 命令检查 *wtmp* 文件中最后一条记录是否是关闭信息。如果不是，将假定上次是非正常关机，执行 *fsck* 以检查所有的文件系统。*/etc/rc* 的最后一项任务是启动守护进程。这可以通过辅助脚本命令完成。如果察看 *ps aux* 命令的输出，它显示了进程的 PID 及父进程的 PID (PPID)，将会看到像 *update* 和 *usyslogd* 这样的守护进程位于 *init* 的子进程中的第一持久进程集合 (first persistent processes) 中。

最后，*init* 读取 */etc/ttymtab* 文件，该文件列出了所有可能的终端设备。那些可以作为登录终端（在标准发行版本中，只包括主控制台和最多三个虚拟控制台，但是还可以添加串口线和伪终端）的

设备都在 */etc/ttymtab* 的 *getty* 域有对应的一项。而且 *init* 为每个这样的终端创建一个子进程。通常每个子进程都执行文件 */usr/bin/getty*, 打印出一条信息, 然后等待输入一个用户名。如果某个终端需要特殊的处理(如一条拨号线路), 则 */etc/ttymtab* 可以指定一条命令(如 */usr/bin/stty*), 在执行 *getty* 之前执行该命令并对线路进行初始化。

当用户输入用户名登录时, */usr/bin/login* 被调用, 并使用这个用户名作为参数。*login* 进程判断是否需要密码, 如果需要则输出密码提示符, 然后验证密码。在成功登录之后, */bin/login* 执行用户的 *shell*(默认是 */bin/sh*, 在 */etc/passwd* 文件中还可以指定其他的 *shell*)。*shell* 等待用户键入命令, 并为每条命令创建一个新的进程。采用这种方式时, 各个 *shell* 都是 *init* 的子进程, 而用户进程则是 *init* 的孙子进程, 并且所有的用户进程都是一棵进程树的组成部分。实际上, 除了编译进内核的任务和进程管理器外, 所有的系统进程和用户进程构成一棵进程树。但是与传统的UNIX系统进程树不同, *init* 进程不是进程树的树根, 根据进程树的结构也不能判断系统进程的启动顺序。

MINIX 3 用于进程管理的两个主要的系统调用是 *fork* 和 *exec*。*fork* 是创建一个新进程的唯一途径。*exec* 允许一个进程执行一个指定的程序, 当一个程序被执行时, 将按照文件头中指定的大小为其分配一部分内存。尽管在进程运行期间, 数据段、栈段和空闲未使用部分的分布可以不时地改变, 但进程分配到的内存总量将得到保证。

一个进程的所有信息被保存在进程表中, 进程表划分成内核、内存管理器和文件系统三部分, 分别拥有它们各自所需要的那些域。当出现一个新进程(通过 *fork*)或者一个老进程终止(通过 *exit* 或信号)时, 内存管理器首先更新它那部分进程表, 然后向文件系统和内核发送消息, 以通知它们进行相应的操作。

2.5.3 MINIX 3 中的进程间通信

MINIX 3 提供了三条原语来发送和接收消息, 它们均通过 C 库例程调用。其中,

```
send(dest, &message);
```

用来向进程 *dest* 发送一条消息,

```
receive(source, &message);
```

用来接收一条来自进程 *source* (或任何地方) 的消息,

```
sendrec(src_dst, &message);
```

用来发送一条消息, 并等待同一个进程的应答。

以上调用中第 2 个参数是消息数据的本地地址。内核中的消息传递机制将消息从发送者复制到接收者。应答消息(对于 *sendrec*)将覆盖原先的消息。原则上该内核机制可以替换为另一套机制以实现分布式系统, 即在网络上将消息从一台机器复制到另一台机器上。但在实践中这很复杂, 因为有时消息的内容可能是一个指向大型数据结构的指针, 于是分布式系统也必须提供网络上数据本身的复制功能。

每个任务、驱动程序或服务器进程只允许与一些特定的进程通信。后面将会阐述保证这一机制的细节。通常消息流在图 2.29 所示的结构中是向下传递的, 消息可以在同层或相邻层进程间传递。第 4 层的用户进程可以向第 3 层的服务器发送消息, 第 3 层的服务器可以向第 2 层的驱动程序发送消息。

当一个进程发送消息到目标进程而目标进程并不在等待消息时, 发送进程将阻塞, 直到目标进程调用 *receive* 为止。换句话说, MINIX 3 使用这种方法避免了缓冲那些发送出去但未被目标进程

接收的消息所带来的问题。这种方法的好处在于，它很简单，避免了缓存管理（包括缓存溢出的问题）。此外，因为所有的消息长度固定，并且在编译时确定下来，缓存溢出问题这个常见的错误就从结构上消除了。

限制进程间消息传递的根本原因是，如果允许进程A通过**send**或**sendrec**向进程B发送消息，那么可以允许进程B通过**receive**从进程A接收消息，但是不能允许B向A发送消息。显然，如果进程A尝试向进程B发送消息而阻塞，同时进程B尝试向进程A发送消息而阻塞，这将导致死锁。两个进程完成操作，不是需要像I/O设备这样的硬件资源，而是需要对方发出**receive**调用。死锁的情形将在第3章详细介绍。

有时需要不阻塞的消息发送。还有另外一个重要的消息传递原语，它通过C库例程

```
notify(dest);
```

调用，当一个进程需要通知另外一个进程有重要的事件发生时，可以使用这一调用。**notify**调用是不阻塞的，也就是说不管接收者是否正在等待消息，发送者都能够继续执行。因为通知操作不阻塞，所以可以用它来避免死锁。

可以通过消息机制来发送通知，但所传递的消息非常有限。通常情况下，消息仅仅包含了发送者的身份和一个由内核加上的时间戳。但有时这已足够了。例如，当有一个功能键按下（F1到F12以及shift+F1到F12）时，使用**notify**调用。在MINIX 3中，功能键用来发出调试转储信息。网络驱动程序就是一个例子，它仅产生调试转储信息，不需要与控制台驱动程序进行任何通信。所以当dump-Ethernet-stats（转储网络状态）键按下时，从键盘驱动程序向网络驱动程序发送一个通知的含义是明确的。在另外一些情况下，一条通知还不够，但是目标进程可以在收到通知后向发起者请求更多的信息。

通知消息（notification messages）之所以很简单，有一条原因。因为**notify**调用是不阻塞的，当接收者没有调用**receive**时也可以发出通知。消息的简单性在于，不能被接收的通知可以很容易地存储起来，以便当接收者调用**receive**时可以收到这个通知。实际上，这只需要一个位就够了。通知为系统进程间的通信而设计，在这里，进程总数很少。每一个系统进程有一个挂起通知的位图，每一位代表一个系统进程。当进程A向进程B发送通知而进程B未在**receive**调用上阻塞时，消息传递机制将进程B中的挂起通知位图中代表进程A的那一位置位。当进程B调用**receive**时，第1步操作时检查挂起通知位图。通过这种方式，它可以收到很多进程的通知。被置位的这一位足以重建通知的内容：它说明了发送者，当内核发送消息时，消息传递代码将加上一个时间戳。时间戳的主要目的是检查定时器是否已过期，所以时间戳比发送者开始发送通知时的时间晚一些也没关系。

这里可以对通知机制进一步改进。在有些情况下，通知消息还使用附加域。当生成通知向接收者说明有中断时，一个所有中断源的位图也包含在通知中。当通知来自系统任务时，接收者所有挂起信号的位图也包括在通知当中。这里的问题是，当通知必须发送给一个进程而该进程没有等待消息时，这些附加的信息该如何存储呢？答案是把位图结构存储在内核数据结构中。保存它们不需要附加的副本。如果通知必须推迟且只发送一个位，当接收者最终调用**receive**重新生成通知消息时，知道通知发送者就足以确定在消息中包含什么附加信息。对于接收者而言，通知的发送者还表明了通知是否包含附加信息，如果包含，该如何解释附加信息。

还有其他一些与进程间通信有关的原语，将在本节后面的部分介绍。不过它们不如**send**、**receive**、**sendrec**和**notify**重要。

2.5.4 MINIX 3 中的进程调度

中断系统能使多道程序操作系统持续不断地工作。当进程请求输入时，它们将阻塞以允许其他进程执行。当输入可用时，当前运行进程被磁盘、键盘或其他硬件中断。时钟也产生中断，这种中断使正在运行的未请求输入的用户进程最终放弃CPU，以使其他进程获得运行的机会。MINIX 3 最底层软件的任务就是通过将中断转换成消息来对其加以隐藏。就进程而言，当一个I/O设备完成一个操作时，它向某个进程发送一条消息，将其唤醒并使之就绪。

中断也可以由软件产生，在这种情况下的中断称为陷阱。系统库把上面谈到的 send 和 receive 操作转换成软中断指令，软中断和硬件产生的中断的效果是一样的。执行软中断指令的进程马上阻塞，内核被激活来处理中断。用户程序不直接引用 send 和 receive，当调用图 1.9 列举的系统调用时，可以是直接调用或通过库例程，内部将使用 sendrec 调用并产生软中断。

每当一个进程被中断（不管是常规的 I/O 设备还是时钟）或执行软中断指令时，都有机会重新确定哪个进程最需要运行机会。当然，在一个进程终止时也要执行该操作，但在类似 MINIX 3 这样的系统中，由 I/O 操作、时钟或消息传递引起的中断远远多于进程终止而引起的中断。

MINIX 3 调度器使用一个多级排队系统。一共定义了 16 个队列，但是重新编译以定义更多或更少的队列也是可以的。最低优先级队列只由 IDLE 进程使用，IDLE 进程在系统没有其他任务时运行。用户进程启动时默认的优先级比最低优先级要高一些。

服务器进程处于比用户进程具有更高优先级的队列当中，而驱动程序进程处于比服务器进程更高优先级的队列当中，系统和时钟任务处于最高优先级队列当中。在某一时刻并不一定所有的队列都在用。开始进程只在几个队列当中。一个进程可以被系统移到一个具有不同优先级的队列当中，用户也可以通过 nice 命令在一定限制内改变进程优先级。额外的优先级供试验使用，当向 MINIX 3 中添加了新的驱动程序时，也可以改变默认优先级以得到更好的性能。例如，如果需要添加一个向网络发送数字音频或视频的服务器进程时，可以赋予这个服务器进程一个比当前服务器进程更高的初始优先级，或者降低当前服务器进程或驱动程序进程的优先级，以使新的服务器进程性能更好。

除了进程所在队列所决定的优先级以外，还使用其他机制以使一些进程比其他进程更有优势。时间片，即进程在被抢占前所允许运行的最大时间间隔，在所有的进程中并不相同。用户进程有一个较小的时间片。驱动程序进程和服务器进程通常可以运行到阻塞，为了防止故障，它们是可抢占的，但具有一个大的时间片。它们可以运行大的但仍是有限的时钟节拍，如果它们用完了时间片就会被抢占，这样可以使系统不至于挂起（hang）。在这种情况下，运行超时的进程仍处于就绪态，但被放到它所在队列的尾端。如果一个用完了时间片的进程，仍然是上一次运行的进程，则可以认为它卡在了一个循环中，从而出现了阻止其他低优先级进程运行的征兆。此时，将把该进程放到一个较低优先级队列的队尾以降低它的优先级。如果这个进程又运行超时，而还有其他进程得不到运行，它的优先级将再一次降低。最终使其他的进程都得到机会运行。

一个被降低优先级的进程仍有机会提高它的优先级。如果一个进程用完了它的时间片，但没有妨碍其他进程运行，它的优先级将得到提高，直到该进程所允许的最大优先级为止。这样的一个进程显然需要它所得的时间片，但它并没有不顾及其他进程。

此外，使用一个经改进的轮转调度算法调度进程。如果进程转为非就绪的时候没有用完它的时间片，这说明它在 I/O 上阻塞，当该进程再次转为就绪时就被放到队首，并分配上次所剩余的时间片。这样是为了给用户进程更快的 I/O 响应。一个用完了时间片的进程以纯时间片轮转调度的方式被放到队尾。

任务通常有最高的优先级，驱动程序次之，服务器再次，最后是用户进程。这样，只有当所有的系统进程都无事可做时，用户进程才得到运行。系统进程不会被用户进程阻止运行。

当选择进程运行的时候，调度器首先检查最高优先级队列中的进程，如果一个或多个处于就绪态，那么队首的进程将运行。如果没有进程就绪，那么下一个优先级队列将会被同样地检查，依次类推。因为驱动程序进程响应服务进程的请求，服务器进程响应用户进程的请求，最终，所有高优先级进程都完成被请求的工作。如果没有进程就绪，将选择 *IDLE* 进程。这将使 CPU 处于低功耗模式，直到下一个中断产生。

在每一个时钟节拍，都将检查当前进程是否运行完了分配给它的时间片。如果是，调度器将它放到队尾（如果它是队列中的唯一进程，那么什么也不用做）。然后，如前所述，将选择下一个进程运行。只有当高优先级队列中没有进程且前面的那个进程在其队列中唯一时，它将再次得到运行。否则，具有最高优先级非空队列队首的进程将运行。重要的驱动程序和服务器被分配一个较大的时间片，在被时钟抢占之前，它们通常已经阻塞。但是当出错时，将会降低它们的优先级以防止整个系统处于停滞状态。如果是一个关键服务器出错，可能其他什么也不能做，但至少可以平和地能关闭系统、防止数据丢失并收集帮助调试的有用信息。

2.6 MINIX 3 中进程的实现

现在来讨论实际代码，先介绍一下将要用到的几个术语。“过程”、“函数”以及“例程”这几个词可以混用。变量名、过程名以及文件名将用斜体，如 *rw_flag*。当一条语句用变量名、过程名或文件名开头时，它将使用大写，但真正的名字都使用小写字母。也有一些例外，被编译进内核的任务将以大写表明，如 *CLOCK*, *SYSTEM* 和 *IDLE*。系统调用将用小写字母表示，如 *read*。

由于本书和软件一直在不断演变，而不是在同一天交付出版，所以在代码引用、打印出的代码清单以及 CD-ROM 上的软件之间可能会有少许出入。但这种差异一般只影响一两行代码。本书印刷出来的源代码已经进行了简化，从中删去了那些与本书未讨论的编译选项相关的部分。MINIX 3 网站 (www.minix3.org) 上有最新的版本，它具有新的特性及其配套软件和文档。

2.6.1 MINIX 3 源代码的组织

本书所讨论的 MINIX 3 的应用是运行在 IBM 兼容机上的，它们有一颗 32 位的微处理器（如 80386, 80486, 奔腾, 奔腾 Pro, II, III, 4, M 或 D），即所有这些 Intel 的 32 位处理器。在该标准平台上 C 语言源代码的完整路径是 */usr/src*（路径名中的 “/” 指向一个目录）。其他平台下源程序的路径有可能不同。在本书中，都是用一个顶层目录为 *src/* 的路径字符串来引用 MINIX 3 的源程序文件的。该目录下有一个重要的子目录 *src/include/*，在该子目录下存放了所有的主要 C 头文件的副本。在本书中用 *include/* 指代本目录。

源代码目录树下的每一个目录都包含一个名为 **Makefile** 的文件，它用来完成 UNIX 标准下的 *make* 命令。**Makefile** 文件控制它所在目录中的文件的编译，有时也会包括它的一些子目录中的文件。*make* 命令的执行比较复杂，对它的完整讲解超出了本节的范畴，但是可以简单地说 *make* 使得涉及到大量源代码文件的程序编译更加高效与方便。*make* 使得所有必要的文件都得到编译。它检查先前编译过的模块看它们是否是最新的，并重新编译那些自从上次编译之后其源代码已经被改动过的模块。这避免了重新编译那些本来不需要再编译的文件，从而节省了时间。最后，*make* 将编译好的各个独立的模块组合成一个可执行的程序，有可能还将它彻底安装好。

src/ 目录树的全部或者部分也可以被重新分配位置，因为每个源目录内的 **Makefile** 都是使用 C 源程序所在目录的相对路径。例如，根设备是一个 RAM 驱动器时为了提高编译速度，读者也许会想要在文件系统的根目录下建一个源程序目录 */src/*；如果读者正在开发一个专用的版本，则可以以其他的名字做一个 *src/* 的副本。

C头文件的路径是一种特殊情况。在编译期间，每个 *Makefile* 都会到 */usr/include/* 目录（或者在非 Intel 平台下的相应目录）去查找头文件。但是，用来重编译系统的 *src/tools/Makefile* 则会到 */usr/src/include* 目录（在 Intel 系统上）下查找头文件的副本。在重新编译系统之前，整个 */usr/include/* 目录树将被删除，而 */usr/src/include/* 则被复制到 */usr/include/*。这样做是为了使 MINIX 3 开发所用到的所有文件都在同一个地方。这样也使维护众多源文件和头文件的副本更加容易，因为这些文件是用来实验 MINIX 3 系统的不同配置的。但是，如果读者在这类实验中想要编辑某个头文件，则一定要确保自己所编辑的是 *src/include* 目录中的副本，而不是 */usr/include/* 中的文件。

在此还要顺便向 C 语言入门者介绍一下在 *#include* 语句中怎样包含一个文件名。每一个 C 编译器都会到一个默认的头文件目录中去查找 *include* 文件。大多数情况下，这个目录是 */usr/include/*。当一个 *include* 文件是在一个小于号和一个大于号之间（即“*<...>*”）被引用时，编译器则到默认头文件目录或者一个指定的目录中去查找该文件，例如，

```
#include <filename>
```

包含了 */usr/include/* 目录中的一个文件。

也有很多程序定义了一些不想在系统内共享的本地头文件。这样的头文件可能会有一个与标准头文件相同的名字，以代替或补充标准头文件来使用。当文件名在普通的引号内（“*"..."*”）被引用时，该文件将首先在其源文件（或某个特定的子目录）所在目录中查找，如果没找到，再到默认目录中查找。这样，

```
#include "filename"
```

将读取一个本地文件。

include/ 目录包含了许多符合 POSIX 标准的头文件，它又包含三个子目录：

- sys/* 包含 POSIX 头文件。
- minix/* 包含操作系统使用的头文件。
- ibm/* 包含 IBM PC 特有定义的头文件。

为了支持对 MINIX 3 和在 MINIX 3 环境下运行程序的扩展，在 CD-ROM 上同时提供了放在 *include/* 目录下的其他文件和子目录，这些内容也可在 MINIX 3 网站上找到。例如，*include/arpa/* 和 *include/net/* 目录及其子目录 *include/net/gen/* 支持网络扩展。这些对于编译基本的 MINIX 3 系统并不是必需的，而且这些目录中的文件也未列在附录 B 中。

除 *src/include/* 之外，*src/* 目录还包含了其他三个重要的子目录，它们也包含了操作系统源代码：

- kernel/* 第 1 层（调度、消息、时钟和系统任务）。
- drivers/* 第 2 层（磁盘、控制台、打印机等的驱动程序）。
- servers/* 第 3 层（进程管理器、文件系统、其他服务器）。

还有另外三个源代码目录在本书中未列出，也不加以讨论。但对于构造一个能够发挥作用的系统，它们是很重要的：

- src/lib/* 库例程（如 *open*, *read*）的源代码。
- src/tools/* 用于构建 MINIX 3 系统的 *Makefile* 和脚本。
- src/boot/* 引导和安装 MINIX 3 的代码。

MINIX 3 的标准发布版还包含了很多本文没有讨论的其他源代码。除了进程管理器和文件系统源代码外，系统源程序目录 *src/servers/* 还包含了 *init* 程序和再生服务器 *rs* 的源代码，它们都是运行

MINIX 3 系统所必需的。网络服务器的源代码在 *src/servers/inet/* 目录中。而 *src/drivers/* 中则包含了本文没有探讨的设备驱动程序的源代码，包括可选设备，如磁盘驱动程序、声卡、网卡。由于 MINIX 3 是一个用于教学的操作系统，这意味着要对它经常做修改，所以有一个 *src/test/* 目录包含有一些被设计用来对新编译好的 MINIX 3 系统进行完整测试的工具。操作系统当然要支持在其上运行的命令（程序）。所以有一个很大的命令目录 *src/commands/*，其中包含工具程序（如 *cat*, *cp*, *date*, *ls*, *pwd* 以及 200 多个其他程序）的源代码。最初由 GNU 和 BSD 工程开发的一些主要的开源应用程序的源代码也在这里。

本书在讲解 MINIX 3 时删除了很多可选项（这是因为，不可能在一本书中包含所有的东西，也不可能在一个学期的课程中把所有东西都装进学生的脑子里）。该版用 *Makefile* 编译时没有涉及不必要的文件（一个标准的 *Makefile* 要求必须呈现所有的文件，即使其不被编译）。删除了这些文件以及条件语句，这使得阅读代码更加容易。

一个目录下的文件将放在一起讨论，这样就不会发生混淆。这些文件在附录 B 中按照其在文中讨论的顺序罗列，这样更易于掌握。此时使用一对书签可能会比较有用，这样读者就可以在正文和附录列表中来回翻看。为将列表的大小保持在合理范围内，没有将每个文件的代码打印出来。通常情况下，在正文中详细讨论的函数都列入了附录 B；而只是一带而过的则没列进去，但可以在 CD-ROM 和 Web 站点上找到其完整代码，上面同样提供了函数索引、定义以及源代码内的全局变量。

附录 C 包含了一个在附录 B 中描述的文件的字母表，分为头文件、驱动程序文件、内核文件以及进程管理器文件几部分。附录、Web 站点和 CD-ROM 使用了所列各对象在源代码中的行号作为索引。

第 1 层代码位于 *src/kernel/* 下。该目录下的文件支持进程管理，这是图 2.29 中所示的 MINIX 3 结构的最低层。该层包括完成以下功能的函数：系统初始化、中断、消息传递以及进程调度。与此密切相关的是两个编译进了同一个 binary 的模块，但是它们是作为两个独立的进程来运行的，即在较高层在内核服务和进程间提供接口的系统任务，以及为内核提供时钟信号的时钟任务。在第 3 章中，我们将研究 *src/drivers* 目录的子目录中的一些文件，它们支持图 2.29 中第 2 层的多种设备驱动程序。在第 4 章将讨论 *src/servers/pm/* 目录下的进程管理器文件。最后在第 5 章将学习 *src/servers/fs/* 下的文件系统。

2.6.2 编译及运行 MINIX 3

可以在 *src/tools/* 目录下运行 *make* 命令来编译 MINIX 3，其中有些选项对应不同的安装方式。可以执行不带参数的 *make* 以查看这些选项。最简单的方法是 *make image*。

执行 *make image* 时，*src/include/* 目录下的头文件将在 */usr/include/* 目录下产生一个新的副本。然后 *src/kernel/* 目录中的源文件以及 *src/servers/* 和 *src/drivers/* 中的一些子目录将被编译成 *object* 文件。*src/kernel/* 中所有的 *object* 文件再链接成一个可执行程序 *kernel*。*src/servers/pm/* 目录中的所有 *object* 文件也将链接成一个可执行程序 *pm*，*src/servers/fs/* 目录中的 *object* 文件则链接为 *fs*。图 2.30 中所列的作为引导映像的组成部分的其他程序也在它们各自的目录中被编译和链接。这包括 *src/servers/* 的子目录中的 *rs* 和 *init* 以及 *src/drivers/* 子目录中的 *memory/*, *log/* 和 *tty/*。图 2.30 中指定的“driver”组件指各类磁盘驱动程序中的任何一种。此处讨论的是将 MINIX 3 系统配置为使用标准的 *at_wini* 驱动程序从硬盘驱动器中引导，*at_wini* 驱动程序在 *src/drivers/at_wini/* 目录中被编译。其他的驱动程序可以添加进去，但是绝大多数驱动程序需要编译进引导映像。网络支持也是如此，无论是否使用网络，基本的 MINIX 3 系统都是如此。

为了安装一个能被引导的可运行的MINIX 3系统，名为*installboot*的程序（其源代码在*src/boot/*中）将名字添加到*kernel, pm, fs, init*以及其他引导映像组件中，并添加一些数据以使其长度为磁盘扇区大小的整数倍（以便可以更容易地独立加载该部分），并将其连接为一个文件。这个新的文件就是引导映像文件，可以将其复制到软盘或者硬盘分区的*/boot/*或者*/boot/image/*目录中，然后，引导监控程序可以加载该引导映像并将控制权交给操作系统。

图 2.31 描述了链接后的程序被分离和加载后的内存分配情况。内核被载入到内存低端，而引导映像的其他部分则被加载到高于 1 MB 的地址处。当用户进程运行时，内核以上的可用内存将首先被使用。如果此处并不适合新程序，那么该程序将被加载到内存中高于 *init* 的区域内。当然，具体细节还依赖于系统的配置。例如，图中所展示的例子适合拥有能够装载 512 B 到 4 KB 大小的磁盘数据块的块缓冲的 MINIX 3 文件系统。这是一个中等数值，如果有适当的内存可用，推荐使用更多。如果大量地缩减缓冲区数目，则有可能使整个系统占用不到 640 KB 的内存，从而还可以为几个用户进程留出空间。

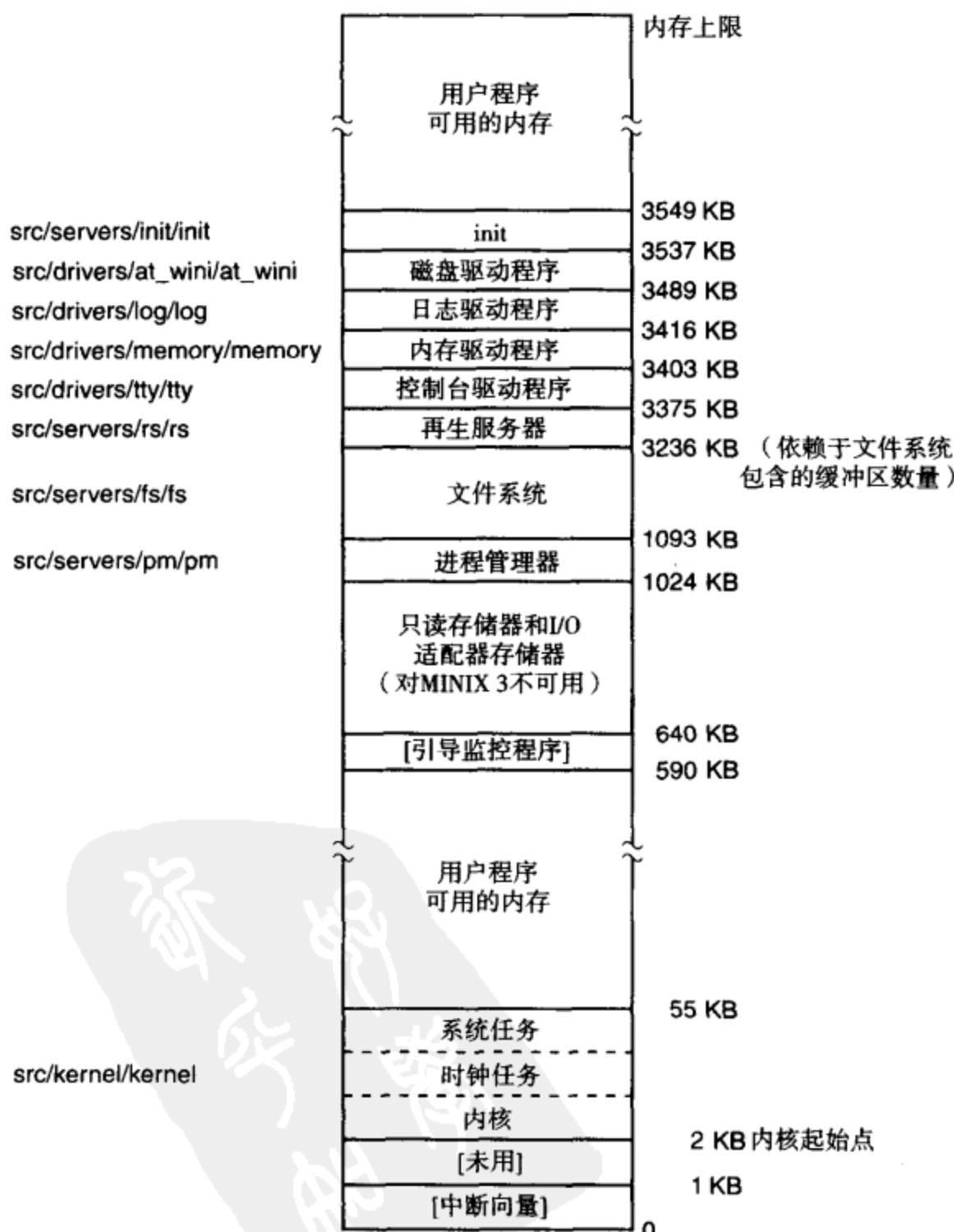


图 2.31 MINIX 3从磁盘加载到内存后的内存分配情况。内核、服务器和驱动程序都独立地编译、链接，并示于左边。图中的尺寸只是近似大小，未按比例画出

一定要明白 MINIX 3 是由一些完全独立的程序组成的，而这些程序之间只是靠传递消息来通信。目录 *src/servers/fs/* 中的 *panic* 程序与目录 *src/servers/pm/* 中的 *panic* 程序并不冲突，因为最终它们是被链接进了不同的可执行文件。通常情况下操作系统的三大块通用的程序只有 *src/lib/* 目录中的库程序。这种模块结构使得文件系统很容易改变，而不会影响到进程管理器。还使得很容易将整个文件系统移植到一台不同的机器上充当文件服务器，通过在网络上发送消息来与用户机器通信。

MINIX 3 的模块化特点的另一个例子是，加入网络支持对进程管理器、文件系统以及内核没有丝毫影响。无论是以太网络驱动程序还是 *inet* 服务器都可以在引导映像加载后激活；在图 2.31 中可以看到由 */etc/rc* 启动的进程，它们可以加载进图 2.31 所示的“用户程序可用的内存”区域中的一块。一个启动了网络功能的 MINIX 3 系统可以用做远程终端或者一个 FTP 和 Web 服务器。只有当读者想要来自网络的访问登录到该 MINIX 3 系统时，本文中所描述的一些部分才需要做些改动：如控制台驱动程序 *tty*，它需要使用伪终端配置选项来重新编译以允许远程登录。

2.6.3 公共头文件

include/ 及其子目录包含了许多文件，其中定义了常量、宏以及类型。POSIX 标准需要其中的许多定义，并且指定了在 *include/* 及其子目录 *include/sys/* 下其所需的每一个定义应放在哪一个文件中。这些目录下的文件是头文件（header），或称为包含文件，用后缀 *.h* 标识。这些文件通过 C 源程序中的 *#include* 语句引用，该语句是 C 语言的特性之一。头文件使得更容易维护一个大系统。

编译用户程序时可能用到的头文件主要放在 *include/* 目录下，而传统上 *include/sys/* 目录放那些编译系统程序和实用程序所用的头文件。这种规定并不是绝对的，而且一个典型的编译过程，无论是编译用户程序还是编译操作系统的一部分，将同时包含这两个目录下的文件。这里首先讨论编译标准的 MINIX 3 系统所用到的文件，先看 *include/* 目录，再看 *include/sys/* 目录。下一节将讨论 *include/minix/* 和 *include/ibm/* 目录下的所有文件。顾名思义，*include/ibm/* 用于 IBM 类型的机器上的 MINIX 3 实现。

首先考虑的是真正通用的头文件，它们不直接被任何 MINIX 3 系统的 C 源文件引用，而是被其他的头文件包含。MINIX 3 系统的每一个主要部件都有一个主要的头文件，如 *src/kernel/kernel.h*, *src/servers/pm/pm.h* 以及 *src/servers/fs/fs.h*，它们包含在这些部件的每一个编译版本中。每一个设备驱动程序的源代码都包含一个相似的文件 *src/drivers/drivers.h*。每个头文件都根据 MINIX 3 系统相应部分的需要进行剪裁，但每个文件都含有一个如图 2.32 所示的起始部分。在本书的其他部分还将讨论这些主控头文件，这里只是强调不同目录下的头文件是一起使用的。在本节和下一节中还将提到图 2.32 中引用的每一个头文件。

```
#include <minix/config.h>
#include <ansi.h>
#include <limits.h>
#include <errno.h>
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/syslib.h>
#include "const.h"

/* 必须是第一个头文件 */
/* 必须是第二个头文件 */
```

图 2.32 一个主控头文件的一部分，它保证了将所有 C 源文件都需要的那些头文件包括在内。注意引用了两个 *const.h* 文件，一个来自 *include/* 树，另一个来自本地目录

讨论从 *include/* 目录下的第 1 个头文件 *ansi.h* 开始 (0000 行)。这是编译 MINIX 3 系统任何一部分都要处理的第 2 个文件，第 1 个文件是 *include/minix/config.h*。*ansi.h* 的作用是测试编译器是否符合 ISO 规定的标准 C 的要求。标准 C 也称为 ANSI C，因为在获得国际承认之前它最早是由美国国家标准局开发的。一个标准 C 编译器定义了若干宏，它们可在被编译的程序中测试。*_STDC_* 就是一个这样的宏，而且它被一个标准编译器定义为 1，就像是 C 语言预处理器读入了如下的一行：

```
#define _STDC_ 1
```

随当前的 MINIX 3 版本发行的编译器遵从标准 C，但老版本的 MINIX 是在该标准被采纳之前开发的。不过仍然可以用一个传统的 C 编译器 (Kernighan & Ritchie) 来编译 MINIX 3。MINIX 3 希望能够很容易地被移植到新的平台上，允许使用老的编译器正是该目标的一部分。如果使用标准 C 编译器，则将处理在 0023 到 0025 行之间的语句

```
#define _ANSI
```

ansi.h 使用不同的方式定义了若干宏，这具体依赖于是否定义了宏 *_ANSI*。这是特性测试宏 (feature test macro) 的一个例子。

另一个特性测试宏是 *_POSIX_SOURCE* (0065 行)，这是 POSIX 要求的，如果定义了其他与 POSIX 兼容的宏，那么也要确保该宏被定义。

在编译一个 C 程序时，数据结构、参数以及函数返回值在引用它们的代码生成之前必须是已知的。在一个复杂系统中，对这些函数定义排序以满足该要求是很困难的，所以 C 允许在定义之前使用函数原型来声明一个函数的参数和返回值。*ansi.h* 中最重要的宏是 *_PROTOTYPE*。它允许用如下形式写一个函数原型：

```
_PROTOTYP(return-type function-name, (argument-type argument, ...))
```

如果使用一个 ANSI 标准 C 编译器，则将由 C 预处理器将其转换成

```
return-type function-name(argument-type, argument, ...)
```

如果使用一个老式 (即 Kernighan & Ritchie) 编译器，则将其转换成

```
return-type function-name()
```

在结束 *ansi.h* 的讨论前还要提及另一个特性。整个 *ansi.h* 文件被包括在语句

```
#ifndef _ANSI_H
```

和

```
#endif /* _ANSI_H */
```

之间。在 *#ifndef _ANSI_H* 的下面一行，紧接着定义 *_ANSI_H*。在一次编译中，一个头文件应该仅被包含一次。这种写法保证了一个文件被包含多次时内容的正确性。接下来将会看到在 *include/* 目录下的所有头文件都使用这种技术。

还有相关的两点需要指明。第一，头文件目录下的文件中的所有 *#ifndef ... #define* 语句包含的文件名的前面要加下划线。另外一些具有相同名字的头文件可能存在子目录中，而且在那里会使用同样的机制，但是文件名前不会有下划线。这样包含主要头文件目录中的一个文件并不能防止执行一个本地目录下的头文件。第二，注意 *#ifndef* 后的 */* _ANSI_H */* 并非是必需的。该

内容只是为了帮助识别嵌套的 `#ifndef ... #endif` 和 `#ifdef ... #endif` 部分。尽管如此，在编写此类内容时仍需注意，如果出现错误，则比没有它们更糟糕。

include/ 目录下第二个间接包含在每一个 MINIX 3 源文件中的头文件是 *limits.h* (0100 行)。其中定义了许多基本的大小值，既有语言中的数据类型，如整数所占的位数，也有操作系统的限制，如文件名长度。

注意，为了方便，在附录 B 中一个新的文件是从下一个 100 的整数倍行开始的。所以别指望 *ansi.h* 包含 100 行 (00000 到 00099 行)。在这种方法下，对于一个文件的轻微改动 (可能) 不会影响到后面的文件的重新排列。还需要注意当一个新的文件列入时，将标有三行的头信息 (没有行号)，它们由一行 + 号、文件名以及另一行 + 号组成。00068 到 00100 行展示了一个这样的例子。

errno.h (0200 行) 也被所有的主控头文件包含。它包含了系统调用失败时从全局变量 *errno* 返回给用户程序的错误码。*errno* 也用来标识一些内部错误，如试图向一个不存在的任务发送消息。在内部调用一个可能产生错误的函数去查看一个全局变量的效率是很低的，但是函数必须总是返回一个整数，例如，在 I/O 操作中被传送的字节数。在 MINIX 3 中，若返回值为负，则表示它是一个系统内的错误码，但在返回用户程序之前必须使它们为正值，这里的技巧在于每个错误码都在类似下面一行 (0236 行) 的语句中定义：

```
#define EPERM(_SIGN 1)
```

操作系统各部分的主控头文件定义宏 *_SYSTEM*，但在一个用户程序被编译后从不定义 *_SYSTEM*。如果 *_SYSTEM* 被定义，则 *_SIGN* 定义为 “-”，否则 *_SIGN* 无定义。

下面一组文件未被包含在所有的主控头文件中，但被 MINIX 3 所有三个部分的许多源文件使用。最重要的是 *unistd.h* (0400 行)。该头文件定义了许多常量，其中多数是 POSIX 需要的。它也包含了许多 C 函数的原型，其中包括所有用于进行系统调用的 C 函数原型。另一个用得广泛的文件是 *string.h* (0600 行)，它包含许多用于串操作的 C 函数原型。*signal.h* (0700 行) 定义了标准信号名，一些 MINIX 3 操作系统专用的信号也在那里进行了定义。MINIX 3 的操作系统功能是由相互独立的进程处理的，而不是在单体内核中处理的。上述事实要求在系统组件之间建立一种特殊的类似于信号的通信机制。*signal.h* 也包含一些信号相关的函数原型。随后我们将看到，信号处理牵涉到 MINIX 3 的所有部分。

fcntl.h (0900 行) 定义了文件控制操作使用的许多参数。例如，它允许在 *open* 调用中使用宏 *O_RDONLY* 来代替数值 0 作为参数。尽管该文件主要由文件系统引用，但它的定义在内核和内存管理器中也多次用到。

正如在第 3 章中分析设备驱动程序时我们将看到的那样，操作系统的控制台和终端接口是很复杂的，因为各种不同类型的硬件必须通过一种标准的方式与操作系统和用户程序交互。头文件 *termios.h* (1000 行) 定义了控制终端类型的 I/O 设备所用到的常量、宏和函数原型。最重要的数据结构是 *termios* 结构，它包含的内容有：标识各种操作模式的标志位、设置输入输出速率的变量以及放置特殊字符的数组（比如 *INTR* 和 *KILL* 字符）。这个结构体是 POSIX 所需要的，该文件中定义的许多宏和函数原型同样也是 POSIX 需要的。

然而，正如 POSIX 标准始终遵循的那样，它并未提供可能用到的全部内容。在文件的后半部分，1140 行以后，提供了对 POSIX 的扩展。其中有些扩展的意义是很明显的。比如定义 57 600 或以上的波特率，以及对终端显示窗口的支持。正如任何合理的标准都是开放的那样，POSIX 标准并不排斥扩展。但在 MINIX 3 环境下想写一个可移植到其他环境的程序时，就必须注意避免使用那

些局限于 MINIX 3 的特征。这一点很容易做到。在该文件和其他定义面向 MINIX 3 的扩展的文件中，这些扩展是通过以下语句控制的：

```
#ifdef _MINIX
```

如果 `_MINIX` 未定义，那么编译器将根本不会感知到 MINIX 扩展。它们将被完全忽略。

看门狗定时器由 `timers.h` (1300 行) 支持，它被包含于内核的主头文件之中。它定义了 `timer` 结构体和一个用于操作定时器链表的函数原型。在 1321 行有一个 `tmr_func_t` 的类型定义。该数据类型是一个函数指针。在 1332 行可以看到它的用法：在 `timer` 结构体内，作为定时器链表中的一个元素，在定时器到期时调用该 `tmr_func_t` 指向的那个函数。

再介绍一下附录 B 中没有列出的 `include/` 目录下的四个文件。`stdlib.h` 定义了除最简单的 C 程序之外几乎所有 C 程序编译时都需要用到的类型、宏和函数原型。在编译用户程序时，这是使用得最频繁的头文件之一，尽管在 MINIX 3 系统源文件中，它只被内核中的一小部分文件引用到。对于每一个从学写著名的“Hello World!”程序开始接触 C 语言的人，`stdio.h` 是最熟悉的了。同 `stdlib.h` 一样，在系统文件中，几乎没用到过它，但几乎每一个用户程序都离不开它。`a.out.h` 定义了可执行文件在磁盘上的存储格式。结构体 `exec` 也是在这里定义，进行 `exec` 调用加载一个新的程序镜像时，进程管理器会用到该结构体内的信息。最后，`stddef.h` 定义了一些公用的宏。

现在来看 `include/sys/` 目录。如图 2.32 所示，MINIX 3 系统各主要部分的主控头文件在包含 `ansi.h` 后随即都包含 `sys/types.h` 文件 (1400 行)。`sys/types.h` 中定义了许多 MINIX 3 使用的数据类型。使用这里提供的定义可以避免对特定情况下所使用的基本数据结构的理解错误而导致的故障。图 2.33 显示了其中定义的几种数据类型，以及它们分别在 16 位和 32 位处理器上编译时所占位数的差异。注意，所有的类型名都以“_t”结尾，这不仅仅是一种习惯，而且是 POSIX 标准的规定。这是保留后缀的一个例子，而且“_t”不用于非数据类型的其他任何符号名。

类型	16位MINIX	32位MINIX
<code>gid_t</code>	8	8
<code>dev_t</code>	16	16
<code>pid_t</code>	16	32
<code>ino_t</code>	16	32

图 2.33 16 位和 32 位系统中的各种类型及所占的位数

MINIX 3 现在自然还是运行在 32 位微处理器上，但是 64 位机将是未来的主流。如果需要，可以生成一个硬件并未提供的类型。在 1471 行，`u64_t` 类型被定义为一个结构体 `struct {u32_t[2]}`。目前的应用中，此类型还不太需要，但是它很有用，比如，所有的磁盘和分区都按 64 位存储（偏移量和大小），则允许使用容量非常大的磁盘。

MINIX 3 定义了许多类型，它们最终会被编译器编译为相对较小的普通类型的数。这是为了使代码更易读，例如，声明为 `dev_t` 类型的变量代表标识设备 I/O 的主副设备号。而对于编译器来说，将其声明为一个 `short` 类型能够工作得同样良好。还要说明的是，还定义了很多首字母分别为大小写的配对类型，比如 `dev_t` 和 `Dev_t`。对于编译器，首字母大写的所有类型都等价于 `int` 类型；它们用在某些函数原型中，这些原型必须使用与 `int` 类兼容的类型以支持 K&R 编译器。`types.h` 对此做了详细解释。

另一个值得介绍的是以

```
#if _EM_WSIZE == 2
```

开头的条件代码部分（1502到1516行）。先前曾指出，大多数条件代码都从源代码中移走。该例却被保留了，所以可以指出使用条件性定义的方法。所使用的宏 `_EM_WSIZE` 是编译器定义的特性测试宏的又一个例子。它以字节为单位告知目标系统一个字的大小。`#if ... #else ... #endif` 序列语句是一次性获取所有定义的一种方法，它使得代码无论是在16位还是在32位系统上使用时都能被正确编译。

MINIX 3 系统还广泛使用了 `include/sys/` 目录下的其他文件。文件 `sys/sigcontext.h` (1600行) 定义了用于在执行一个信号处理程序前后保存和恢复系统操作的结构体，它能被内核及进程管理器使用。`sys/stat.h` (1700行) 定义了图 1.12 中所示的由 `stat` 和 `fstat` 系统调用返回的结构体，以及 `stat`, `fstat` 及其他操作文件属性的函数的原型。该文件被文件系统和进程管理器多次引用。

接下来我们将要讨论的文件不像前面的这些被广泛地引用。`sys/dir.h` (1800行) 定义了 MINIX 3 的目录项结构。它只被直接引用过一次，但包含它的文件在文件系统中被广为使用。它的重要之处在于，它定义了一个文件名最长有多少个字符(60)。而 `sys/wait.h` (1900行) 头文件定义了用于 `wait` 和 `waitpid` 系统调用的宏，它们本身在内存管理器中实现。

还应该介绍一下 `include/sys/` 目录下的其他一些文件，尽管附录 B 中并未包含它们。MINIX 3 通过一个调试程序提供了对跟踪可执行文件和用调试器分析内核转储（core dump）的支持。而 `sys/ptrace.h` 定义了使用 `ptrace` 系统调用的各种可能操作。`sys/svrctl.h` 定义了用于 `svrctl` 的数据结构和宏，`svrctl` 并不是系统调用，但是与此类似。`svrctl` 经常在系统启动时用来协调服务器级进程。`select` 系统调用允许在多个通道等待输入，例如伪终端等待网络连接。该调用所需的定义在 `sys/select.h` 中。

之所以将对 `sys/ioctl.h` 及其相关文件的讲解放在最后是有原因的，因为不同时关注另一个目录下的文件 `minix/ioctl.h` 是不可能完全理解这些文件的。`ioctl` 系统调用用来进行设备控制操作。通过与现代计算机连接而进行控制的设备的数目越来越多，它们都需要各式各样的控制。实际上，本书中讲述的 MINIX 3 和其他版本的 MINIX 之间的主要差别在于：为了突出本书的意图，文中描述了一个只带有少量 I/O 设备的 MINIX 3。其余许多设备可被加进来，如网络接口、CD-ROM 驱动器、声卡等。

为了增加可执行性，使用了一些小文件，它们各自包含一组定义。它们包含在 `sys/ioctl.h` (2000行) 中，功能与图 2.32 中的主控头文件类似。在附录 B 中只列了其中之一，即 `sys/ioc_disk.h` (2100行)。它和其他被 `sys_ioctl.h` 包含的文件位于 `include/sys/` 目录中，因为它们被认为是“发布的接口”，也就是说程序员可以使用它们来编写任何运行于 MINIX 3 环境下的程序。尽管如此，它们中的每一个还都要依赖于由 `minix/ioctl.h` (2200行) 所提供的宏，并包含它们。在编写程序时 `minix/ioctl.h` 不能被它自己使用，这就是它放在 `include/minix/` 中而不是放在 `include/sys/` 中的原因。

这些文件定义的宏的作用是，它们定义了每一个函数所需要的各个参数是如何被包装成一个32位的整数传递给 `ioctl` 的。例如，磁盘设备需要五类操作，这可以在 2110 到 2114 行的 `sys/ioc_disk.h` 中看到。字母 “d” 表示告诉 `ioctl` 该操作是针对磁盘设备的，操作码是一个 3 到 7 的整数，读或写操作的第 3 个参数表示被传送数据的结构的大小。在 2225 到 2231 行的 `minix/ioctl.h` 文件展示了这样一个过程：8 位字符码被放在左边 8 位，表示结构大小的 13 位最低有效位向左扩展为 16 位，然后与小整数操作码进行逻辑与运算。而 32 位数字中的 3 位最高有效位则被编码为返回值类型。

虽然这看起来是很重的一份工作,但该过程是在编译阶段完成的,从而产生了运行期间对于系统调用的高效率接口,因为实际上传输的都是些宿主机CPU的最普通的数据类型。但是,它确实使人们想起了Ken Thompson在早期UNIX的源代码中所写的一句著名的注释:

```
/* You are not expected to understand this */
```

*minix/ioctl.h*还包含2241行处的*ioctl*系统调用的原型。很多情况下,这个调用都不由程序员直接进行,因为POSIX在*include/termios.h*中定义的函数原型已经取代了老的*ioctl*库函数的许多功能,包括对终端、控制台及类似设备的处理。但该系统调用仍然是需要的。实际上,控制终端的POSIX函数通过函数库被库转换成了*ioctl*系统调用。

2.6.4 MINIX 3 头文件

*include/minix/*和*include/ibm/*目录包含了MINIX 3特有的头文件。*include/minix/*中的文件对任何平台上的MINIX 3实现都是需要的,尽管其中一些文件中有与平台相关的替代定义。前面已经解释过文件*ioctl.h*。*include/ibm/*中定义了IBM类型机器上的MINIX 3实现所特有的数据结构和宏。

先从*minix/*目录开始。在前一节中我们曾看到*config.h*(2300行)被MINIX 3所有部分的主控头文件所包含,并且它是编译器实际上处理的第一个文件。在许多情况下,当因硬件差异或对操作系统的使用方式不同而需要对MINIX 3的配置进行修改时,只需要编辑该文件并对系统重新编译即可。如果更改了这个文件,建议需要再修改一下2303行的内容来帮助解释做此修改的目的。

用户可设置参数均放在该文件的第一部分,但其中一些参数并不宜在此修改。在2326行包括了另一个头文件*minix/sys_config.h*,一些参数的定义可以从该文件获得。程序员们曾认为这样做是个好主意,因为系统中一些文件只需*sys_config.h*中的基本定义,而不用*config.h*文件中的其他部分。但事实上,*config.h*文件中有很多名字都不是以下划线开头的,它们很容易与通常所用的名字冲突,例如*CHIP*和*INTEL*就很可能出现在从其他系统迁移到MINIX 3上的软件中。而*sys_config.h*中的所有名字都是以下划线开头的,从而降低了冲突的可能性。

在*sys_config.h*中*MACHINE*实际上被配置为*_MACHINE_IBM_PC*,2330到2334行列出了所有可选的*MACHINE*值。早期的MINIX版本曾被移植到Sin,Atari和Macintosh平台上,所以完整的源代码包含了各种可选的硬件配置项。MINIX 3的绝大多数源代码是平台无关的,但是一个操作系统总会有些系统相关的代码。所以需要指出的是,因为MINIX 3还是刚出不久,至于其在其他非Intel平台上的移植还需要做很多工作。

*config.h*文件中的其他定义允许在特殊安装中进行其他需求的定制,例如,对于磁盘缓存来说,用于文件系统的缓存容量是越大越好,但是一个大容量的缓存需要占用大量的内存。在2345行处配置的128个数据块的缓存是最小值,刚刚能满足在RAM小于16 MB的系统上安装MINIX 3的要求;对于拥有充足内存的系统来说可以分配更多的缓存。如果想要使用调制解调器或者通过网络远程登录,那么*NR_RS_LINES*和*NR_PTYS*(2379行和2380行)的定义应增加,并重新编译系统。*config.h*的最后部分定义了其他必要的参数,但是不能修改。这里的很多定义都只是*sys_config.h*中定义的常量换了名字而已。

sys_config.h(2500行)包含了一些定义,可能会被系统程序员用到,比如新式设备驱动程序开发者。不太需要修改这个文件,除了*_NR_PROCS*(2522行)。它控制进程表的大小,如果读者想使用MINIX 3作为一个网络服务器以便多个远程用户或者服务器进程能并发执行,那么需要增加此常量的值。

下一个文件是 *const.h* (2600行)。通过它可以理解头文件的另一种普遍用法。在这里可以发现一些常量定义，在编译一个新内核时它们一般不会改变，但却在许多地方用到。通过这些定义可以防止发生变量在多处定义不一致的错误，而且这种错误很难发现。源代码目录树中还有另外几个文件也命名为 *const.h*，但其使用很有限。只在内核中使用的定义都包含在 *src/kernel/const.h* 中，而只在文件系统中使用的定义都放在 *src/servers/fs/const.h* 中，进程管理器使用的则在 *src/servers/pm/const.h* 中。只有那些在 MINIX 3 中不止一处用到的定义才放在 *include/minix/const.h* 中。

const.h 中有几处定义值得注意。*EXTERN* 被定义为一个扩展的 *extern* 宏 (2608行)。对于在头文件中定义并被两个以上的文件包含的全局变量，可声明为 *EXTERN* 类型，形式如下：

```
EXTERN int who;
```

如果变量采用方式

```
int who;
```

声明并被包含于两个以上的文件中，则有些链接程序将会把它认做变量多重定义错误。而且，C 语言参考手册 (Kernighan 和 Ritchie, 1988) 明确地禁止这种语法形式。

为了避免这类问题，必须在除一处之外的所有其他地方都写成

```
extern int who;
```

在包含了 *const.h* 之处 *EXTERN* 将被替换成 *extern*，而在定义该变量之前明确地将 *EXTERN* 重新定义为空串，那么这一问题便解决了。实现方法是：在 MINIX 3 的每一部分中，将全局定义放在一个指定文件 *glo.h* 中。例如 *src/kernel/glo.h*，该文件被间接地包含在每一个编译版本中。在每个 *glo.h* 中有如下语句：

```
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif
```

并且在 MINIX 3 的每一部分的 *table.c* 文件中，在 *#include* 之前都有一行

```
#define _TABLE
```

这样，当头文件被包含进 *table.c* 并展开成为其中的一部分时，*extern* 绝对不会出现在该文件中的任何地方（因为在 *table.c* 中 *EXTERN* 被定义为空串），并且全局变量的空间只被预留在一个地方，即 *table.o* 中。

如果读者是 C 程序设计的新手，对上述内容不太理解，那么其中的细节也并不特别重要。这只是对于 Ken Thompson 早期的著名阐述的一种优雅转述。对有些链接程序而言，头文件的多次包含可能会导致一些问题，因为它可能引起被包含变量的多重声明。上述 *EXTERN* 用法只是增强 MINIX 3 移植性的一种途径，因为这样一来，即使在链接程序不支持变量多重定义的机器上，MINIX 3 也能够被链接成功。

PRIVATE 被定义为 *static* 的同义词。对于那些不会在定义文件以外被引用的过程和数据，它们往往被声明为 *PRIVATE*，以使其名字在文件之外不可见。作为一条通用的规则，所有的变量和过程都应被声明成在尽可能局部的范围内有效。如果可能，*PUBLIC* 被定义为空串，*kernel/proc.c* 中的一个例子将有助于讲述清楚。声明

```
PUBLIC void lock_dequeue(rp)
```

经 C 预处理器处理后成为

```
void lock_dequeue(rp)
```

按照 C 语言的作用域规则，该语句意味着函数 *lock_dequeue* 从该文件输出，并能被其他连接到内核的任何文件的任何位置调用，当然也可以被内核在任意地方调用。该文件中声明的另一个函数是

```
PRIVATE void dequeue(rp)
```

经预处理后成为

```
static void dequeue(rp)
```

这个函数只能被同文件里的代码所调用。*PRIVATE* 和 *PUBLIC* 并不是必需的，但使用它们是为了尽量消除 C 语言作用域规则的副作用（C 语言中一个符号名默认情形下输出到文件之外，但实际上应该正好相反）。

const.h 的其余部分定义了在整个系统中使用的数值常量。其中有一段专门定义与机器或配置相关的内容。例如在全部的源码中，内存大小的基本单位是 **click**（块）。块的具体大小取决于处理器结构。对于 Intel 兼容芯片、Motorola 68000、Sun SPARC 等体系结构，*click* 定义在 2673 到 2681 行，它们的块大小是 1024 位。该文件中还包括了宏 *MAX* 和 *MIN*，因此

```
z = MAX(x, y);
```

的功能是将 *x* 和 *y* 中的较大值赋给 *z*。

type.h（2800 行）是另一个通过主控头文件被包含在每一个编译版本中的文件。它包括了许多重要的类型定义，以及相关的数量值。

前面的两个结构定义了两个不同的内存映射类型，一个是本地内存区域（进程的数据区内），一个是远程内存区域，例如 RAM disk（2828 到 2840 行），先介绍一下设计内存的一些概念比较合适。正如刚刚提到的，块是内存的基本度量单位；Intel 处理器平台的 MINIX 3 的块是 1024 字节。内存用参数 **phys_clicks** 来度量，内核通过它来访问系统任何地方的任何内存区域，而进程则通过 **vir_clicks** 来度量。一块 *vir_clicks* 内存引用常常是指分配给特定进程的一段内存的基地址，而内核常常要在虚拟地址（例如进程基地址）和物理地址（RAM 基地址）之间变换。一个进程能够引用所有 *vir_clicks* 中的内存弥补了这样做带来的不便。

读者可能会认为同样的单元也可以用来描述任何内存类型的大小，但事实上使用 *vir_clicks* 描述某个进程的内存单元大小还有一个优点，这就是当使用这个单元时，还会进行检查，从而确保没有对该进程所占有的内存区域以外的范围进行读写。这是现代 Intel 处理器的保护模式的一个主要特征，如奔腾系列。在早期的 8086 和 8088 处理器上，该特性的存在给设计早期的 MINIX 版本带来过很大麻烦。

此处定义的另一个重要结构是 *sigmsg*（2866 到 2872 行）。当捕获一个信号时，内核必须安排响应该信号的进程在下次得以执行时，它将允许信号处理器（signal handler）运行，而不是在它被打断的地方继续执行。进程管理器完成了管理信号的绝大部分工作，当捕获一个信号时，里程管理器给内核传送一个这样的结构。

kinfo 结构体（2875 到 2893 行）用来向系统的其他部分传达有关系统的信息。进程管理器在创建它的进程表的部分时使用的正是该结构。

进程间通信的数据结构和函数原型在 *ipc.h*（行 3000）内定义。该文件中最重要的定义是 3020 到 3032 行的 *message*。当然可以将 *message* 定义为一个包含若干字节的数组，但在编程实践中最好

将其处理为一个结构，其中包含有若干不同的可能的消息类型的联合。这里定义了七种消息格式：*mess_1* 到 *mess_8* (*mess_6* 已经废弃)。一条消息包括如下的几个部分：*m_source* 域，标识谁发送了该消息；*m_type* 域，标识消息类型（例如，给系统任务的 *SYS_EXEC*）以及数据域。

七种消息类型示于图 2.34。在图中的各类消息中，头两个和后两个比较相似。单就数据元素的大小来讲是这样，但是数据类型则存在很多不同。这主要用在 Intel CPU 上，它的字长为 32 位，*int*, *long* 和指针类型数据也都是 32 位，但在其他种类的硬件平台上，这种情况并不是必需的。定义七种消息类型将使得更易于在不同的体系结构上重新编译。

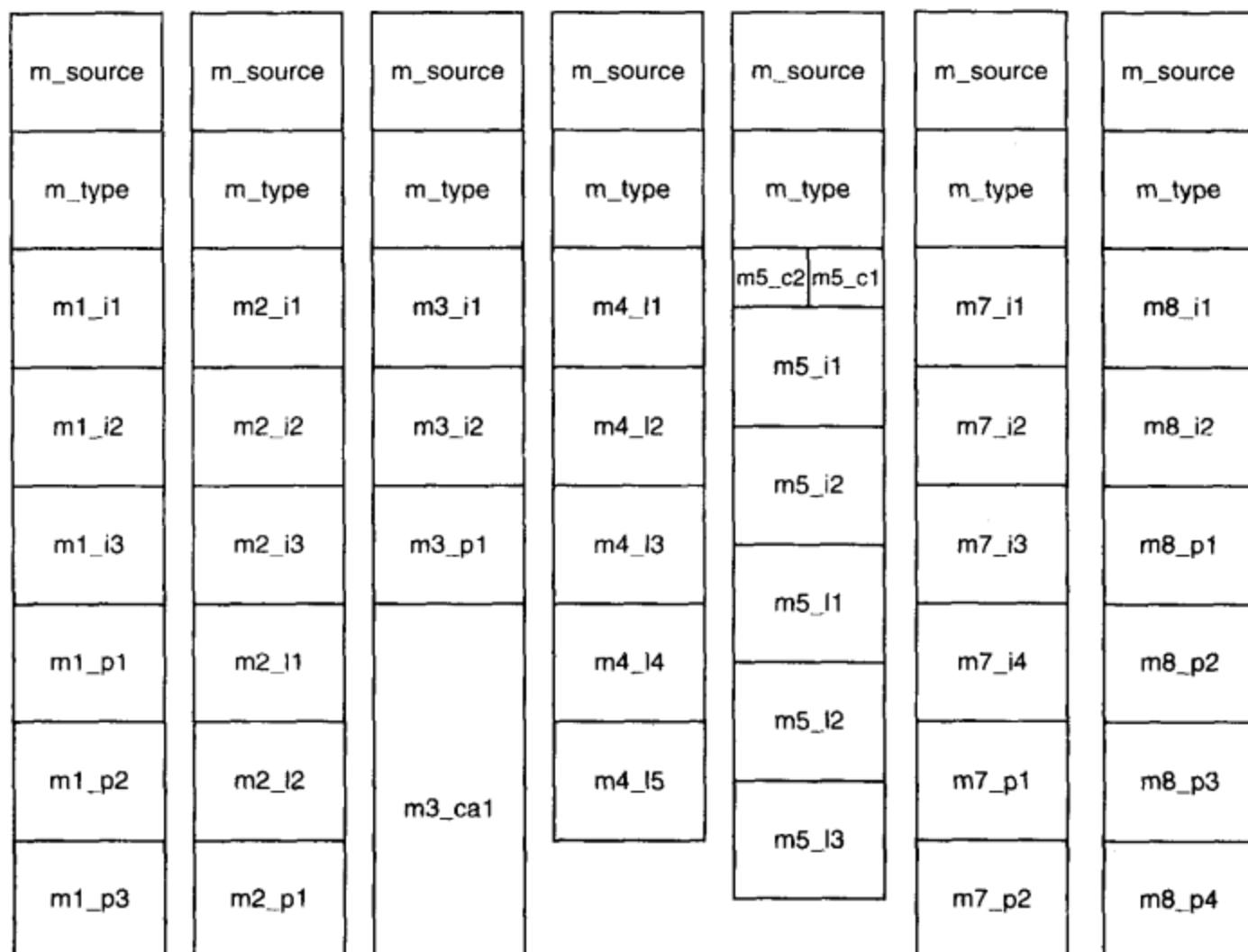


图 2.34 MINIX 3 中所用的七种消息。消息大小随机器体系结构的不同会有所不同。本图展示的是类似奔腾系列的拥有 32 位指针的 CPU 的情况

当需要发送一条消息而其中包含 3 个整数和 3 个指针（或 3 个整数和 2 个指针）时，应使用图 2.34 中的第一种格式。对其他格式依次类推。如何给第一种格式中的第一个整数赋值呢？假设这里的消息为 *x*，则 *x.m_u* 指向该消息结构的联合部分。使用 *x.m_u.m_m1* 来指向 6 种格式中的第一种。最后，用 *x.m_u.m_m1.mli1* 来指向该结构中的第一个整数，这种使用方法相当麻烦，所以在定义了消息之后，一般都定义一个短一点的名字作为其相应的宏。于是用 *x.m1_i1* 来代替 *x.m_u.m_m1.mli1*。这种短名字的格式都由以下几部分组成：字母 *m*，格式编号，下划线，一个或两个类型字母（用来指明该域是一个整数、指针、长整数、字符、字符串或是函数）以及一个序号，在一条消息中包含多个相同类型的域时，可用该序号对它们加以区分。

在讨论消息格式时，顺便指出操作系统及其编译器通常对某些事物有共同的“理解”，比如一个结构的布局等，而且这种共识简化了系统的构造。在 MINIX 3 的消息中，*int* 域有时用来存放 *unsigned* 数据类型。在某些情况下这会导致溢出，但编码时，MINIX 3 编译器只是机械地在 *unsigned* 类型和 *int* 类型之间来复制，而不进行溢出检查。一种更规范的写法是将每一个 *int* 域写成 *int* 和

unsigned 的联合。在消息中对 *long* 也同样处理，有些 *long* 类型的域被用来传送 *unsigned long* 型数据。这样做是不是有点骗人的感觉呢？有人也许这么认为，但如果要将 MINIX 3 移植到一个新平台上，很显然必须对消息的确切格式给予足够的注意，而且编译器的特性是另一个必须注意的因素。

ipc.h 文件中定义了前文所述的消息传递原语的原型（3095 到 3101 行）。除了重要的 *send*, *receive*, *sendrec* 和 *notify* 原语外，还定义了其他的原语。它们很少被用到，可以说它们是 MINIX 3 发展史上早期版本的残留物。老式的计算机程序可挖掘出这些内容，但在将来的版本中，它们会消失。然而，如果不进行解释，想必现在很多读者都会对其迷惑不解。非阻塞的 *nb_send* 和 *nb_receive* 调用大都被 *notify* 替换了，而且它被认为是更好地解决发送或检查一个未阻塞的消息的方法。*echo* 原型没有源和目的地址域。该原语对于生成代码没有任何帮助，但是它可以检测出发送和接收消息所用的时间。

include/minix/ 下还有另一个被 MINIX 3 所有用户空间组件的主控头文件所包含的文件广泛使用的文件，即 *syslib.h*（3200 行）。这个文件不包含在内核的主控头文件 *src/kernel/kernel.h* 中，因为内核不需要库函数访问它自己。*syslib.h* 包含了在操作系统内访问其他操作系统服务所需调用的 C 库函数原型。

本书不详细讨论 C 库，但其中有许多是标准的，且对任何 C 编译器都可用。然而，*syslib.h* 引用的 C 函数显然是特定于 MINIX 3 的；并且，将 MINIX 3 移植到一个带有不同编译器的新系统将需要移植这些库函数。幸运的是，这并不难，因为这些函数只是抽取出函数调用的参数，将其插入到一条消息结构中，然后发送该消息，接着从应答消息中将结果抽取出来。这中间许多库函数的定义仅有十几行甚至更少的 C 代码。

值得注意的是，该文件中的四个使用字节或字数据类型访问 I/O 端口进行输入输出的宏和 *sys_sdevio* 函数原型，四个宏引用的都是 *sys_sdevio* 函数原型（3241 到 3250 行）。它为设备驱动程序提供了一种方法来请求通过内核读或写 I/O 端口，而这也是 MINIX 3 将所有的驱动程序都移到用户空间所需具备的最基本的一部分。

有一些本应在 *syslib.h* 中定义的函数却定义于一个独立的文件 *sysutil.h*（3400 行）中，因为它们的代码被编译为一个独立的库。其中有两个函数原型需要说明一下。第一个是 *printf*（3442 行）。如果读者曾使用过 C 语言编程，则一定知道 *printf* 是一个标准的库函数，几乎在所有的程序中都会用到。

但事实上，这并非我们平时所用的 *printf*。该版本是一个并不能用于系统组件的标准库函数。另外，标准的 *printf* 会输出到标准输出设备上，输出内容包括格式化浮点数。使用标准输出需要请求浏览文件系统，但打印消息时，如果出现了问题或者系统组件需要显示一个错误信息，则希望它能在没有其他系统组件的辅助下来完成这些。而实行一个支持所有显示格式规范的标准 *printf* 会使得代码膨胀，且没有太大的意义。因此 *printf* 的精简版本，即只供操作系统组件使用的版本，被编译进了系统公用库内。它可以被编译器在平台相关的地方找到；对于 32 位 Intel 系统，是在 */usr/lib/i386/libsysutil.a* 中。当系统文件、进程管理器或者操作系统的其他部分链接库函数时，该版本先于标准库函数被搜索并发现。

下一行是 *kputc* 的原型。它被 *printf* 的系统版本调用以完成在控制台显示字符。但是，还涉及到很多微妙的事务。*kputc* 在多处定义。在系统公用库内有一个副本，这是默认使用的版本。但是系统的其他部分还定义了它们自己的版本。下一章我们学习控制台接口时会看到其中之一。日志驱动程序（此处不予详细介绍）也定义了一个专用版本。甚至内核自己也定义了一个 *kputc*，但这属于特殊情况。内核并不使用 *printf*。专门的打印函数 *kprintf* 是内核的一部分，当内核需要打印时将使用到它。

当一个进程需要执行一条MINIX 3系统调用时，它向进程管理器（缩写为PM）或文件系统（缩写为FS）发送一条消息。每条消息中含有所要求的系统调用序号。这些序号在下一个文件*callnr.h*（3500行）中定义。有些序号并未用到，它们是为未实现的系统调用或目前被库函数替换了的系统调用保留的。在文件的末尾定义了一些不符合图1.9的调用序号。*svrctl*（前面曾提到过），*ksig*，*unpause*，*revive*和*task_reply*只被操作系统自己使用。系统调用机制是实现这些的一个好方法。事实上，因为它们不能被外部程序使用，这些所谓的“系统调用”可能会在MINIX 3的新版本中进行修改，而不会破坏用户程序。

另一个文件是*com.h*（3600行）。其文件名的一种说法是它代表common，另一种是说它代表communication。这个文件定义了在服务器和设备驱动程序间通信所用到的常用定义。在3623到3626行定义了任务序号。为了将其与进程号分开，任务号是负数。3633到3640行定义了引导映像所要加载的进程的进程号。注意这是进程表中的各项编号，不要将它与进程号（PID）混淆。

*com.h*的下一部分还定义了如何构造一个消息来执行*notify*操作。该进程号用于生成消息的*m_type*域的值。该文件中定义的*notifications*和其他类型消息是通过将一些基本数值组合而建立起来的，这些数值使用一个代表特定类型的较小的数来划分类型。文件剩余部分是各种宏的描述，它们将各种含义明确的标识符转换为含义不明显的标识消息类型和域名的数字。

附录B还列举了*include/minix/*目录下的其他一些文件。*devio.h*（4100行）定义了支持用户空间访问I/O端口的类型和常量，以及一些宏，这些宏使得编写描述端口及其数值的程序更加简单。*dmap.h*（4200行）定义了一个结构及其数组，皆命名为*dmap*。这张表用于将主设备号和支持它们的函数联系起来。还定义了内存设备的主副设备号和其他重要设备驱动程序的主设备号。

*include/minix/*包含了其他一些并未列入附录B的特殊头文件，但编译内核时必须包含它们。其中的*u64.h*对64位整数算法操作提供了支持，对于高性能磁盘驱动的地址操作，它是必需的。在UNIX、C语言、奔腾系列处理器以及MINIX中，早期是没有想到这方面的。今后的MINIX 3版本可能会使用一种嵌入了对拥有64位寄存器的处理器的64位整数支持的语言来编写；到那时，*u64.h*的定义将会有用武之地。

还要介绍三个文件。*keymap.h*定义了若干结构，这些结构用来实现不同语言所需的字符集对应的特殊键盘布局。它也被那些生成和加载这些表格的程序使用。*bitmap.h*提供了一些宏，它们使得置位、重置位和测试位更加容易。最后，*partition.h*定义了MINIX 3用来定义磁盘分区的信息，或者是通过磁盘大小及字节绝对偏移，或者是通过柱面、磁头、扇区等地址。*u64_t*类型用来表示磁盘大小和偏移，并允许使用大容量硬盘。该文件并未描述磁盘上分区表的布局，描述这些内容的文件位于下一个目录内。

*include/ibm/*是这里讨论的最后一个特殊头文件目录，它下面有两个文件包含与IBM PC系列机相关的信息。因为C语言只知道内存地址，并且不能访问I/O地址，所以库包含了用汇编语言编写的读写I/O端口的程序。这些程序在*ibm/portio.h*中定义（4300行）。*inb*（输入一个字节）到*outsl*（输出一个long类型的字符串）包括所有可能的字节、整数、长数据类型、独自的或者作为一个字符串的输入和输出程序。内核中低层的程序可能还需要启用或者关闭CPU中断，这也是C语言所不能执行的操作。函数库提供了汇编代码来完成此任务，在4325到4326行声明了*intr_disable*和*intr_enable*。

该目录中的下一个文件是*interrupt.h*（4400行），它定义了中断控制器和微机系统的BIOS要用到的内存和端口地址。最后，*ports.h*（4500行）定义了更多的I/O端口。该文件定义了时钟芯片要用到的地址以及读写键盘接口所需的地址。

include/ibm/ 中的一些其他文件未在附录 B 中列出，但是也很基本并值得一提。如果读者想深入了解内存的使用以及磁盘分区表，那么强烈建议读者读一下 *bios.h*, *memory.h* 和 *partition.h*。而 *cmos.h*, *cpu.h* 和 *int86.h* 提供了关于端口、CPU 标志位以及 16 位模式下的 DOS 服务的其他信息。最后，*diskparm.h* 定义了格式化软盘所需的数据结构。

2.6.5 进程数据结构和头文件

现在来看 *src/kernel/* 下的代码。前两节中的讨论是围绕一个典型的主控头文件进行的，这里先来观察内核的主控头文件 *kernel.h* (4600 行)。它一开始先定义了三个宏。第一个 *_POSIX_SOURCE* 是 POSIX 标准自行定义的一个特性测试宏 (feature test macro)。所有这类宏都必须以一个下划线 “_” 开头。定义 *_POSIX_SOURCE* 的作用是保证所有 POSIX 要求的符号和那些显式地允许但并不要求的符号将是可见的，同时又隐藏掉任何属于 POSIX 非官方扩展的附加符号。本书已经提到过下面的两个定义：宏 *_MINIX* 将为 MINIX 3 所定义的扩展而重载 *_POSIX_SOURCE* 的效果；而在编译系统代码时，如果要做与用户代码不同的事情，比如改变错误码的符号，则可以对 *_SYSTEM* 宏进行测试。*kernel.h* 随后包含 *include/* 及其子目录 *include/sys/* 和 *include/minix/* 下的其他头文件，*include/ibm/* 目录包括图 2.32 中的所有文件，这些文件在前两节中已经做过讨论。最后将包含本地目录 *src/kernel/* 下的另外 6 个头文件，它们的名字在引用字符里面。

通过在所有其他内核源文件中写这样一行简单的语句：

```
#include "kernel.h"
```

便可以使所有源文件共享大量重要的定义。由于头文件的包含次序有时非常重要，*kernel.h* 同时还一次性地保证了这种次序的正确性。这等于将头文件“做对一次，然后便可忽略具体细节”的思想带到了一个新高度。文件系统和进程管理器也存在类似的主控头文件。

下面继续看 *kernel.h* 中的本地头文件。首先看一下 *config.h* 这个文件，它被归入系统文件 *include/minix/config.h*，必须在其他本地文件之前被包含进来。正如在公共头文件目录 *include/minix/* 下有 *const.h* 和 *type.h* 一样，在 *src/kernel/* 目录下也有 *const.h* 文件和 *type.h* 文件。*include/minix/* 下的文件之所以被放在那里是因为系统的许多部分，包括在系统控制下运行的程序，都需要它们。*src/kernel/* 下的文件提供了仅为编译内核所需的定义。FS 和 PM 以及其他源文件目录也包含有 *const.h* 和 *type.h* 文件，同样这些文件定义了仅为系统相应部分使用的常量和类型。在主控头文件中包含的另外两个文件 *proto.h* 和 *glo.h* 在主 *include/* 目录下没有对应的文件。但可以发现它编译 FS 和 PM 时使用的对应文件。*kernel.h* 中包含的最后一个本地头文件是 *ipc.h*。

因为这还是它第一次进入本书的讨论范围，注意在 *kernel/config.h* 的开始处有个 *#ifndef ... #define* 序列语句来防止某个文件被包含多次，我们先前曾看到过此类情况。但要注意的是，此处定义的宏是没有下划线的 *CONFIG_H*。这样它就与 *include/minix/config.h* 中的 *_CONFIG_H* 宏区分开来了。

config.h 的内核版本在同一个地方收集那些不太可能需要更改的定义的编号，除非读者学习 MINIX 3 是为了了解一个操作系统是如何工作的，或者想要将这个操作系统在一台传统的通用计算机上运行起来，才需要更改这些定义。然而，假如读者想制作一个 MINIX 3 的小版本来控制科学操作或者家用的手机，则要在 4717 到 4743 行选取合适的系统调用。除去不需要的功能还降低了对内存的要求，因为处理每个内核调用所需要的代码是使用 4717 到 4743 行定义的条件编译的。如果一些函数被取消，那么执行它的代码也会从系统二进制代码中删除。例如，手机不需要产生 (fork) 新进程，因而该部分代码可以从执行文件中删除，这便使得内存占用更小。该文件中定义的其他大

多数常量都用来控制基本的参数。例如，当处理中断时，将使用专门的一个大小为 `K_STACK_BYTES` 的堆栈。它的值在文件的 4772 行设置。该堆栈的空间为汇编语言文件 `mpx386.s` 保留。

在 `const.h` (4800 行) 中定义了一个宏 (4814 行)，用于将内核的内存空间的与基地址相关的虚拟地址转换为物理地址。在内核的其他地方还定义了一个 C 函数 `umap_local`，利用 `umap_local` 函数，内核可以为系统中的其他组件做地址转换，然而对于核心内部的地址转换，宏 `vir2phys` 的效率更高。这个文件中也包含了另外一些宏，其中几个完成位模式操作。包括一些位操作。Intel 系列内置了一个重要的安全机制，它正是由此处定义的两个宏来激活的。**进程状态字 (PSW)** 是个 CPU 寄存器，其中 **I/O Protection Level (IOPL)** 位定义了是否允许访问中断系统和 I/O 端口。4850 到 4851 行定义了不同的 PSW 值，它们决定了普通进程和特权进程对中断和 I/O 端口的访问权限。这些值将被压入栈中作为将进程切换到执行状态这一过程所需的部分数据。

在下一个文件中，我们将考虑 `type.h` (4900 行)，其中定义了一个名为 `memory` 的结构 (4925 到 4928 行)，些结构使用了地址和存储区大小两个域来保证唯一标识一块内存区域。

`type.h` 定义了其他一些实现 MINIX 3 所需的结构和函数原型。例如，`kmessages` 用于来自内核诊断的消息，`randomness` 用于随机数发生器。`type.h` 还包括了几个机器相关的类型定义。为了使代码长度更短、可读性更强，本书删去了条件性代码和其他类型 CPU 相关的数据类型定义。但是读者应该能够认识像 `stackframe_s` 这样的结构体 (4955 行 4974 行)，它定义了寄存器值是如何被保存到栈中的，主要针对 32 位处理器。对于其他平台，`stackframe_s` 结构体将根据其 CPU 使用的寄存器结构来定义。另一个例子是 `segdesc_s` 结构体 (4976 到 4983 行)，这是保护机制的一部分，它用来防止进程访问不属于它的内存区域。对于有些 CPU，可能根本就不存在 `segdesc_s` 结构，这主要与内存保护机制有关。

对于这类结构，另一个要指出的关键问题是，确保所需的数据项齐全是必需的，然而仍可能不足以达到性能优化。`stackframe_s` 必须以汇编语言代码来处理。如果把此结构定义为可让汇编语言高速读写，那么将减少进程上下文切换的时间。

下一个文件 `proto.h` (5100 行) 提供了一些函数的原型，这些函数需要在其被定义的文件之外可用。它们都使用了前一节中提到的 `_PROTOTYPE` 宏。这样，MINIX 3 内核既可以使用传统的 C 编译器（由 Kernighan 和 Ritchie 定义），例如 MINIX 3 初始提供的编译器，又可以使用一个现代的 ANSI 标准 C 编译器，例如 MINIX 3 版本中所带的编译器。这其中的许多函数原型是与系统相关的，包括中断和异常处理例程以及用汇编语言写的一些函数。

文件 `glo.h` (5300 行) 包含了内核的全局变量。宏 `EXTERN` 的作用已在对文件 `include/minix/const.h` 的讨论中说明。它通常被展开成 `extern`。注意 `glo.h` 中的许多定义都以该宏开头。当该文件被包含在定义了宏 `_TABLE` 的 `table.c` 中时，宏 `EXTERN` 的定义被强行取消。这样当 `glo.h` 被包含在 `table.c` 的编译中时，按这种方式定义的变量的实际存储空间将被保留。将 `glo.h` 包含在其他 C 源文件中将使 `table.c` 中的变量定义为内核中的其他模块所感知。

此处一些内核信息相关的结构在启动时被使用。`aout` (5321 行) 保存了一个数组的地址，这个数组存储的是 MINIX 3 的所有系统映像组件的头信息。注意这些是物理地址，也就是说，与处理器的整个地址空间相关。我们将会看到，`aout` 本身的物理地址在 MINIX 3 启动时会从引导监控程序传递到内核，所以内核的启动程序能够从监控程序的内存地址中得到 MINIX 3 的所有组件地址。`kinfo` (5322 行) 是另一个重要的信息块，它在 `include/minix/type.h` 中定义，同引导监控程序通过 `aout` 来将引导映像中所有有关处理器的信息传递给内核一样，内核将有关它自己的信息放到了 `kinfo` 的各个域中，而系统的其他部分也会需要了解这些信息。

`glo.h` 的下一部分包括了与进程控制和内核执行有关的一些变量。`prev_ptr`, `proc_ptr` 和 `next_ptr` 分别指向之前、当前、之后的进程的进程表项。`bill_ptr` 也指向进程表项；它用来表示哪个进程当前

被审计其使用的时钟节拍。当一个用户进程调用文件系统且文件系统正在运行时，则 *proc_ptr* 指向该文件系统进程。而 *bill_ptr* 则指向进行调用的用户进程，因为文件系统所用的 CPU 时间片长度将从调用者拥有的系统时间中扣除。虽然并未明确听说过哪个进程为其他进程所用时间片买单，但确实是这样。下一个变量 *k_reenter* 用于对内核代码的嵌套执行次数计数，比如当内核本身而不是用户进程发生中断时。这很重要，因为与重新进入内核相比，从用户进程向内核进程或者相反过程的开销是不容忽视（或者说代价昂贵）的。当一个中断服务完成时，它需要确定控制权是应该继续保留在内核手中还是要重启一个用户空间进程。有些关闭或者开启中断的函数也会检测该变量的值，比如 *lock_enqueue*。如果这种函数执行时，中断已经被关闭，那么在不希望重启该中断时，它不会被重新启动。最后，该部分还有一个丢失时钟节拍的计数器。一个时钟节拍怎么会丢失以及如何处理这种情况等问题将会在讨论时钟任务时予以介绍。

glo.h 中定义的最后几个变量也在此处声明，因为它们必须在整个内存代码间都能被感知，但是它们被声明为了 *extern* 而非 *EXTERN*，因为它们是 **初始化变量**（initialized variables），这是 C 语言的一个特性。*EXTERN* 宏与 C 模式的初始化是不兼容的，因为一个变量只能被初始化一次。

运行于内核空间的任务目前只有时钟任务和系统任务，它们在 *t_stack* 中都有自己的栈。在中断处理期间，内核会使用一个单独的栈，但是并非在此声明，因为它只可以被汇编语言层次的中断处理代码所读写，而且不必是全局的。*kernel.h* 包含的最后一个文件是 *ipc.h*（5400 行），它在每次编译时都要用到。它定义了用于进程间通信的各种变量。这将在稍后学习到使用它们的文件 *kernel/proc.c* 时再进行讲解。

还有另外一些内核头文件，虽然不如包含在 *kernel.h* 中的那些用得多，但还是用得很广泛的。第一个是 *proc.h*（5500 行），它定义了内核进程表。一个进程的完整状态是由内存中的进程数据加上它的进程表项中的信息所定义的。当一个进程没有执行时 CPU 寄存器内容就存储在这里，恢复执行时则重新存储。这就是为什么虽然一个 CPU 在任何时刻都只能执行一个进程的指令，但是多进程看上去却是在并行执行并且相互间进行通信。现场数据切换时，花在内核保存以及重保存进程状态数据上的时间是必要的，然而很显然的是，在这些时间里进程实际上暂停了操作。为此该结构要设计得高效一些。正如在 *proc.h* 的开头，访问这些结构以及头文件 *sconst.h* 的程序都是用汇编语言写的一样，定义进程表中的偏移量的代码使用的也是汇编代码。这样在 *proc.h* 中更改定义也必然需要在 *sconst.h* 中更改。

在进行更深入的探讨之前，应该提一下，因为 MINIX 3 的微内核结构，此处所讨论的进程表是与进程管理器和文件管理器中的表类似的，它们包含了与 MINIX 3 中这些部分的函数相关的每一个进程的表项。同样，这三个表也都与拥有单体内核的操作系统的进程表等价，但是当前讨论的进程表指的只是内核的进程表。其他的会在后续章节中介绍。

进程表中的每一项都被定义为一个 *proc* 进程（5516 到 5545 行）。每一项包括进程寄存器、栈指针、状态值、内存映射、栈限制、进程号、计数值、alarm 时间以及消息信息。每一个进程表项的第一部分是 *stackframe_s* 结构体。内存中的进程通过将它的栈指针赋予进程表项的地址来投入运行，并且将该结构弹出到 CPU 寄存器。

关于一个进程的状态，除了 CPU 寄存器和内存中的相应数据外，还有其他一些信息。在 MINIX 3 中，每个进程在其进程表项中都有一个指向 *priv* 结构的指针（5522 行）。此结构定义了消息允许的源（source）和目的（destination），以及其他一些特权。细节部分将在稍后讨论。现在需要注意的是，每一个系统进程都有一个指向其一个副本的指针，而用户的权限与指向同一副本的用户进程的指针相同。在一组标志位 *p_rts_flags*（5523 行）中还有一个字节大小的域。以下会描述这些位的意义。任何一个位置 1 表示一个进程是不可运行的状态，而置 0 则表示就绪。

进程表的每一项为内核所需信息提供了存储空间。例如，*p_max_priority* 域（5526行）表示一个进程首次处于就绪待运行时需要放入哪个调度队列。因为如果某进程阻碍了其他进程运行，那么它的优先级会降低，还有一个*p_priority* 域被初始化为与 *p_max_priority* 相等。每次进程就绪时，实际上是由 *p_priority* 决定使用那个队列的。

每个进程所用时间在 5532 和 5533 行处的变量 *clock_t* 中记录。该信息对于内核必须是可访问的，而且不能存储在进程自己的内存空间中，尽管逻辑上可以这样。*p_nextready*（5535行）用于将进程与调度队列连接起来。

下一个域存储与进程间的消息相关的信息。当进程由于目标未在等待而不能完成 *send* 时，发送者将被放置在目标的 *p_caller_q* 指针指向的队列中（5536行）。这样，当目标最终进行 *receive* 时，它就可以很容易地找到要向它发送消息的进程。*p_q_link*（5537行）域用来将队列成员连接在一起。

在 5538 到 5540 行保留的存储空间使得传递消息的 *rendezvous* 方法成为可能。当一个进程执行了一个 *receive* 但没有被接受的消息时，它将阻塞，而它想要接收信息的进程号被存储在 *p_getfrom* 中。同样地，当一个进程执行了 *send* 而没有接受者需要时，*p_sendto* 将存储目标进程号。消息地址缓存在 *p_messbuf* 中。每条进程表项的倒数第二个域是 *p_pending*（5542行），它使用了一个位映射来跟踪那些没有被发送给进程管理器的信号（因为进程管理器并未在等待该进程）。

最后，进程表项的最后一个域是一个字符数组 *p_name*，用来存储进程名称。该域不是内核管理内核所需要的。MINIX 3 提供了多种由键盘特定键激活的调试内存信息转储（debug dumps）信息。有些允许使用每一个进程的名字与其他数据来观察所有进程的信息。对每一个进程使用一个有意义的名字将使理解及调试内核更加容易。

进程表项之后定义了用于其元素的各种常量。这些可以在 *p_rts_flags* 中设置的各个标志位在 5548 到 5555 行定义及描述。如果某项未被使用，则其 *SLOT_FREE* 被置位。*fork* 之后，*NO_MAP* 被置位以防子进程还未建立内存映射就开始运行。*SENDING* 和 *RECEIVING* 表示进程在发送或接收消息时阻塞。*SIGNALLED* 和 *SIG_PENDING* 表示信号已被接收，而 *P_STOP* 为追踪提供支持。*NO_PRIV* 用来防止新系统进程还未完全建立就开始执行。

接下来（5562 到 5567 行）定义了调度队列和 *p_priority* 的允许值。在该文件的当前版本中，用户进程被授予了访问最高优先级队列的权限；这也许是早期在用户空间测试驱动程序的遗留物，而 *MAX_USER_Q* 也许本应该调整为较低的优先级（一个较大的数字）。

然后定义了几个宏，它们允许进程表重要部分的地址在编译期间被定义为常量，为避免运行期的快速读写，在运行期很多宏进行计算及测试。提供宏 *proc_addr*（5577行）的原因是在 C 中不可能有副下标。数组 *proc* 逻辑上应该从 *-NR_TASKS* 到 *+NR_PROCS*，遗憾的是，在 C 语言中，它必须从 0 开始，所以 *proc[0]* 指优先级最低的任务，依次类推。为更简单地保持表项与进程的对应关系，可以写为

```
rp = proc_addr(n);
```

rp 等于进程 *n* 的进程表项的地址，或正或负。

进程表本身定义为一个 *proc* 结构体的数组 *proc[NR_TASKS + NR_PROCS]*（5593行）。*NR_TASKS* 在 *include/minix/com.h* 中定义（3630行），而常量 *NR_PROCS* 则在 *include/minix/config.h* 中定义（2522行）。它们共同设置了进程表的大小。如果需要，*NR_PROCS* 可以更改，以创建一个能够处理更多进程的系统（例如，在一个大型服务器上）。

最后，定义了一些宏来提高访问速度。进程表被频繁读写，而在一个数组中计算地址需要用到速度很慢的乘法操作，所以提供了一个指向进程表元素的数组 *pproc_addr*（5594行）。*rdy_head* 和

rdy_head 和 *rdy_tail* 这两个数组用来维护调度队列。例如，通过 *rdy_head[USER_Q]* 指向默认用户队列的第一个进程。

正如在对 *proc.h* 的讨论之初曾提到的，还有另一个文件 *sconst.h*（5600 行），如果在进程表中进行了改动，那么它必须与 *proc.h* 同步。*sconst.h* 定义了用于汇编代码的一些常量，以汇编器所使用的形式进行表达。这些常量都是相对进程表项中 *stackframe_s* 部分的偏移，其表示方式为汇编器可使用的一种格式。因为汇编代码不由 C 编译器处理，所以将其放在单独的文件中将更简单。同样，由于这些定义均与机器相关，将其隔离出去将简化 MINIX 3 向另一处理器的移植的过程。该处理器将需要另一个版本的 *sconst.h*。注意，许多偏移被表示为前一值加上字母 W，这里 W 的值在 5401 行被设为字长。这样处理允许同一个文件被编译成 16 位或 32 位版本的 MINIX 3。

这里有一个潜在的问题。头文件被假设为允许一个人提供一套正确的定义，随后可以在许多地方使用而不必过多地注意细节。显然，类似 *sconst.h* 和 *proc.h* 中的重复定义违反了该原则。当然，*sconst.h* 只是一个特例，但在这种情况下，若修改 *sconst.h* 或 *proc.h*，则必须保证它们之间的一致性。

在对进程表的讨论中曾简要提及的系统特权结构体 *priv* 是在 *priv.h* 的 5718 到 5735 行定义的。首先是一组标志位 *s_flags*，然后是 *s_trap_mask*, *s_ipc_from*, *s_ipc_to* 和 *s_call_mask* 域，它们定义了哪些系统调用需要被初始化、哪些进程消息需要接收或者发送以及哪一个内核调用被允许。

priv 结构体不是进程表的一部分，尽管每个进程表项都有一个指针指向它的某个实例。只有系统进程拥有一个私有副本；而用户进程全部指向同一个副本。这样，对于一个用户进程，结构的剩余域是不相关的，因为共享它们没有什么意义。这些域代表未决的通知、硬件中断、信号以及定时器的位映射。在此处为系统进程提供指向信息是很有必要的。用户进程可以让进程管理器管理代表它们的通知、信号、定时器。

priv.h 的组织与 *proc.h* 的相同。在定义了 *priv* 结构体之后，又接着定义了一些重要的宏，它们分别用于标志位、编译期间需要用到的重要地址、运行期间所用的地址。在定义了一个指针数组 *ppriv_addr[NR_SYS_PROCS]*（5762 到 5763 行）之后又定义了 *priv* 结构体数组 *priv[NR_SYS_PROCS]*。该指针数组提供快速访问，类似于对进程表项提供快速访问的指针数组。在 5738 行定义的 *STACK_GUARD* 值是一种很容易识别的格式。稍后我们再看它的用法，另外读者也可以上网查找以了解该值的历史渊源。

priv.h 中的最后一条是一个测试，用以确保 *NR_SYS_PROCS* 被定义为大于引导映像中的进程号的数。当测试条件为真时，#error 行将会打印一条消息。尽管其他的 C 编译器表现可能会不同，但使用标准的 MINIX 3 编译器也会忽略这个编译测试。

F4 键激发一个调试转储信息，它展示了优先级表中的一些信息。图 2.35 展示了包含一些典型进程的优先级表的一些行。标志项的意思是：P 代表抢占（preemptable），B 代表可审计（billable），S 代表系统（system）。trap 栏内 E 代表回显（echo），S 代表发送（send），R 代表接收（receive），B 代表二者皆有（both），N 代表通知（notification）。位图中的一位表示了 32（*NR_SYS_PROCS* 的定义为 32）个系统进程中的一个进程的允许情况。这个次序对应于 id 的域（因篇幅原因，该图中只显示了 16 位）。所有的用户进程共享 id 0，即最左端的位。位图表示了像 *init* 这样的用户进程只能向进程管理器、文件系统以及再生服务器发送消息，而且必须为 *sendrec*。根据该图，服务器和驱动程序可以使用任何 ipc 原语，且除内存外的其他东西都可发送给其他进程。

另一个被许多源文件包含的头文件是 *protect.h*（5800 行）。该文件中几乎所有的内容都与支持保护模式的 Intel 处理器（80286、80386、80486、奔腾系列）体系结构的细节有关。对这些芯片的详细描述不属于本书的范围。简单地说，这些芯片都包含一些指向内存中描述符表（descriptor

tables) 的内部寄存器。描述符表定义了系统资源是如何使用的，并防止进程访问属于其他进程的内存区域。

--nr-	-id-	-name-	-flags-	-traps-	-ipc_to mask-----
(-4)	(01)	IDLE	P-BS-	-----	00000000 00001111
[-3]	(02)	CLOCK	---S-	--R--	00000000 00001111
[-2]	(03)	SYSTEM	---S-	--R--	00000000 00001111
[-1]	(04)	KERNEL	---S-	-----	00000000 00001111
0	(05)	pm	P--S-	ESRBN	11111111 11111111
1	(06)	fs	P--S-	ESRBN	11111111 11111111
2	(07)	rs	P--S-	ESRBN	11111111 11111111
3	(09)	memory	P--S-	ESRBN	00110111 01101111
4	(10)	log	P--S-	ESRBN	11111111 11111111
5	(08)	tty	P--S-	ESRBN	11111111 11111111
6	(11)	driver	P--S-	ESRBN	11111111 11111111
7	(00)	init	P-B--	E--B-	00000111 00000000

图 2.35 优先级表的部分调试转储信息。时钟任务、文件服务器、tty 和 init 进程的优先级分别是典型的任务 (task)、服务器 (server)、设备驱动程序 (device drivers) 和用户进程 (user processes) 的优先级。位图截短显示了 16 位

32 位的 Intel 系列处理器提供了四种特权级 (privilege levels)，MINIX 3 使用其中的三种，它们的符号定义位于 5843 到 5845 行。内核的最中心部分，即运行于中断处理期间的部分和切换进程的部分运行在 *INTR_PRIVILEGE* 特权级。在该特权级上，进程可以访问全部的内存空间和 CPU 的全部寄存器。系统任务运行在 *TASK_PRIVILEGE* 特权级。该特权级允许它们访问 I/O，但不能使用那些修改特殊寄存器 (如指向描述符表的寄存器) 值的指令。服务器进程和用户进程运行在 *USER_PRIVILEGE* 特权级。运行在该特权级的进程不能执行某些指令，如访问 I/O 端口、改变内存分配状况或改变处理器运行级别等。

特权级的概念对于熟悉现代 CPU 体系结构的读者很容易理解，但对于那些通过汇编语言来学习体系结构或是只学习过低档微处理器的读者，可能从未遇到过此类运行级别的限制。

kernel/ 目录中还有一个文件 *system.h* 没有介绍，本文将在后面讨论系统任务时予以介绍，系统任务以一个独立进程运行，尽管它被编译进了内核。现在我们主要看些头文件并深入了解一些 *.c 文件 (C 语言源文件)。首先介绍 *table.c* (6000 行)。对其进行编译得不到可执行代码，但是编译后的目标文件 *table.o* 包含了所有的内核数据结构。在 *glo.h* 和其他头文件中已经看过了很多数据结构。在 6028 行 *#include* 之前定义了宏 *_TABLE*。前面曾说过，这样定义导致 *EXTERN* 成为空串定义，并为用 *EXTERN* 声明的数据分配存储空间。

除了在头文件中声明的变量，全局数据还存储在其他两个地方。在 *table.c* 中直接定义了一些，在 6037 到 6041 行的内核所需的栈空间中进行了定义，而任务所需的栈空间的全部则被保留为 6045 行的数组 *t_stack[TOT_STACK_SPACE]*。

table.c 的剩余部分定义了一些进程属性相关的变量，比如标志位组、调用陷阱和定义消息传递或通知可以发给谁的屏蔽位信息 (如图 2.35 所示，6048 到 6071 行)。接下来是用来定义对于各种进程所允许的内核调用的掩码。进程管理器和文件服务器可以结合。再生服务器可以访问所有的内核调用，但不是为它自己使用，而是因为作为其他系统进程的父进程，它只能将其特权集的子集传递给它的子进程。除了 RAM 驱动器需要经常访问内存外，其他驱动程序都接收了一组系统调用屏蔽 (注意，6075 行的“system services manager”应该称为“reincarnation server”，该名字在发展过程中已经改变，但是还有一些地方使用的是过去的名称)。

最后在 6095 到 6109 行定义了 *image* 表。将它放在这里而不放在一个头文件中，其原因是前述防止多重声明的 *EXTERN* 技术对初始化的变量不起作用，也即任何时候都不能采用如下的写法：

```
extern int x = 3;
```

image 表提供了初始化所有从引导映像加载的进程所需的细节部分。在系统启动时将会用到它。作为此处包含信息的一个例子，考虑 6096 行的 “qs” 域。它展示了分配给每个进程的时钟数。在通常情况下，用户进程，例如 *init* 的子进程，运行 8 个时钟，而如果有必要，时钟和系统任务可以运行 64 个时钟。事实上，在阻塞前它们并不能运行这么长时间，但是不像用户空间服务器和驱动程序，它们即使阻碍了其他进程的运行机会，其优先级也不会降低。

如果在引导映像中新添一个进程，那么在 *image* 表中也要加入新的一行。在将 *image* 的大小与其他常量进行匹配时会发生“不可用”及“不允许”这样的错误。在 *table.c* 测试部分的最后是使用一点小技巧产生一个错误信息。数组 *dummy* 在此声明了两次。在每次声明中，*dummy* 的大小不存在，导致产生编译器错。因为 *dummy* 声明为 *extern*，此处不为其分配空间。因为在代码的其他地方是不能引用它的，这就用不着麻烦编译器了。

另一个分配全局空间的地方是在汇编代码文件 *mpx386.s* 的末尾。尽管为了查看它需要在列表中跳过开头好几页，可是在此讨论它依然是合适的，因为这里正在研究全局变量这一主题。在 6822 行，汇编指令 *.sect .rom* 在内核数据段的最前边放置一个魔数（依靠它来标识一个合法的 MINIX 3 内核）。还使用了 *.sect bss* 汇编指令和 *.space* 伪指令来为内核栈保留存储空间。伪指令 *.comm* 在栈顶标记了一些字以便它们可以直接执行。在稍后讨论完 MINIX 3 的引导步骤后，会再返回到 *mpx386.s* 继续讲解。

2.6.6 引导 MINIX 3

现在我们可以来看执行代码了。但在此之前，先要搞清楚 MINIX 3 是怎么被装入内存的。当然，MINIX 3 是从硬盘上装入的，但是，进程并非完全一样而且事件发生的次序也与磁盘类型有关。确切地说，这依赖于磁盘是否被分区。图 2.36 示出了软盘和做过分区的硬盘的布局。

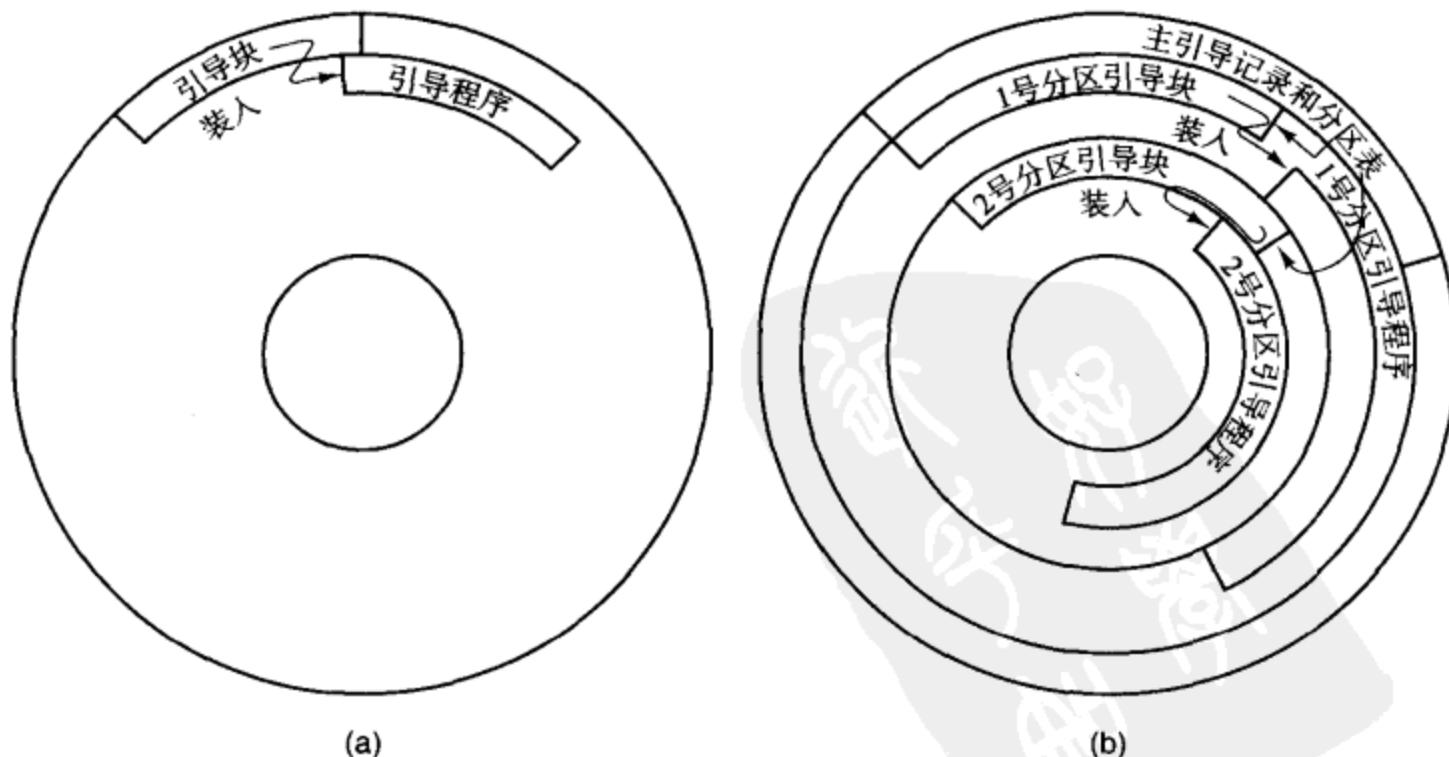


图 2.36 引导过程中所用的磁盘结构：(a)未分区磁盘，第 1 个扇区是引导块；(b)已分区磁盘，第 1 个扇区是主引导记录，又称主引导程序

当系统启动时，硬件（实际是 ROM 中的一个程序）读取引导磁盘的第一个扇区，并执行从那里得到的代码。在一个未分区的 MINIX 3 软盘上，第一个扇区是一个引导块，由它装入引导程序，如图 2.36(a)所示。对于已分区的硬盘，第一个扇区上的程序（称为 MINIX 系统主引导程序）首先将自身定位到一个不同的内存区域，取同样位于第一个扇区中的分区表，并装入和执行活动分区的第一个扇区，如图 2.36(b)所示（通常有且只有一个分区被标识为活动状态）。一个 MINIX 3 分区与一个未分区的 MINIX 3 软盘结构相同，其中有一个装入引导程序的引导块。已分区与未分区磁盘的引导块代码是相同的。因为主引导程序将其自身重定位，所以引导块的代码可以写入，然后从与主引导程序原本加载的地方的相同地址处执行。

真实情况可能较图示的略复杂一些，因为一个分区可能包含有子分区。这种情况下，分区的第一个扇区就是另外一个包含子分区表的主引导记录。但最终控制权将传到一个引导扇区，即一个不再被进一步细分的设备的第一个扇区。对于软盘，其第一个扇区总是一个引导扇区。MINIX 3 确实允许将一张软盘分区，但只有第一个分区可以引导。这张软盘上没有单独的主引导记录，而且也不允许有子分区。这样便可以使用完全相同的方法来安装已分区的和未分区的软盘。软盘分区的主要用途在于它可以方便地将一张安装盘分成一个将复制到 RAM 盘的根镜像，以及一个在不需要时可被卸下来的可安装部分。这样便可以空出软盘驱动器以继续安装过程。

将 MINIX 3 引导扇区写入硬盘时（写入操作由一个特殊的程序 *installboot* 完成，该程序添加一个扇区号以便在其分区或子分区中找到 *boot* 程序），要对它进行修改。在 MINIX 3 中，引导程序的标准位置是在它的同名目录中，也就是 */boot/boot*。但是可以在任何地方进行添加该引导扇区到其加载之处的操作。这个操作是非常必要的，因为在装入操作系统之前无法通过目录和文件名来定位一个文件。

boot 是 MINIX 3 的次级装入程序，它不仅可以装入操作系统，而且作为一个监控程序，它允许用户改变、设置和保存不同的参数。*boot* 从它所在分区的第 2 个扇区中寻找一套可用的参数。像标准的 UNIX 一样，MINIX 3 保留每个硬盘设备的前 1 KB 作为一个引导块。但其中只有一个 512 字节的扇区被 ROM 引导程序或主引导扇区装入，这样，另外 512 字节可以用来保存设置信息。这些信息控制引导操作，并且被传到操作系统本身。默认的设置显示一个只有一个选择项的菜单，即引导 MINIX 3。但该设置信息可以被改变，以显示一个更复杂的菜单。这样，便允许引导其他操作系统（通过装入并执行其他分区的引导扇区）或使用不同的选择项来引导 MINIX 3。默认设置也可以被修改，以旁路掉该菜单而直接引导 MINIX 3。

boot 并不是操作系统的一部分，但它很精巧，可以使用文件系统的数据结构来找到操作系统镜像。*boot* 寻找一个参数 *image = boot* 的特定文件，默认情况下，该文件在 */boot/image* 下，若存在该文件则加载之，但是如果有一个 *minix* 目录且其中有最新文件，则加载该最新文件。许多操作系统都有一个为引导映像预定义的文件名。但 MINIX 3 鼓励用户对其进行改动并生成新的实验版本。这些都要求进行实验的用户应该有某种方法对多个版本进行选择，因为只有这样才能在一次实验失败后退回到上一个正确的版本。

此处我们不再花费过多篇幅讲解引导监控程序的细节。这是一个极其精巧的程序，几乎是一个操作系统的微缩版。它同 MINIX 3 一起运行，但 MINIX 3 正常关闭后，该引导监控程序会重新获得控制权。如果读者想了解更多有关方面的内容，在 MINIX 3 网站上提供了描述其源代码的链接。

MINIX 3 引导映像（也称为系统映像）是好几个程序文件的组合：包括内核、进程管理器、文件系统、再生服务器、设备驱动程序以及 *init*，如图 2.30 所示。此处讲述的 MINIX 3 的引导映像中只配置了一个磁盘驱动器，但可能会有多个，而只有其中一个被标记为活动的（active）。同所有的

执行代码程序一样，引导映像中的每一个文件都包含一个头文件，由它标识在加载完可执行代码及初始化的数据后应为那些未初始化的数据及栈预留多大的空间，以使下一个程序能在合适的地址进行加载。

可用于装入 MINIX 3 的引导监控程序和组件程序的内存区域取决于硬件。同样，某些机器体系结构可能需要对可执行代码内部的地址做一些调整，以将其校准到程序装入的真正地址。Intel 处理器的分段体系结构无须进行该操作。

装入过程的细节随机器类型而不同。重要的是操作系统可以按某种方式装入内存。接下来，在 MINIX 3 启动之前需要做少量的准备工作。首先，在加载镜像时，引导程序从镜像中读取一些字节。它们能告诉引导程序一些关于镜像的属性，最关键的是它被编译为运行在 16 位机还是 32 位机上的属性。然后启动系统所需的一些其他信息将对内核可用。MINIX 3 镜像组件的头文件 *a.out* 被抽取为引导内存空间的一个数组，其基址被传送给内核。当 MINIX 3 结束时，它会将控制权返回给引导监控程序，所以监控程序运行到的当前地址也要被保存。稍后我们会看到，这些信息保存到了栈中。

其他一些信息，如 **引导参数**，必须由引导监控程序传送给操作系统。有些是内核所需要的，而有些是不需要但仍然以消息方式进行传送的，例如，所加载的引导映像的名字。这些条目都是以 *string = value* (属性名 = 属性值) 对的格式表示的，而指向这些对的表的地址通过栈进行传输。图 2.37 显示了一组典型的由来自 MINIX 3 命令行的 **sysenv** 命令所刻画的引导参数。

```
rootdev=904
ramimagedev=904
ramsize=0
processor=686
bus=at
video=vga
chrome=color
memory=800:92540,100000:3DF0000
label=AT
controller=c0
image=boot/image
```

图 2.37 典型 MINIX 3 系统启动时传递给内核的引导参数

此例中需要看的一个重要条目是参数 *memory*；在图示的情况下，它指示引导监控程序决定有两个内存段可以由 MINIX 3 使用。一个由 16 进制地址 800 (十进制是 2048) 开始，大小为十六进制数 0x92540 (十进制为 599 360) 个字节，另一个从 100000 (十进制是 1 048 576) 开始，包含 0x3df00000 (64 946 176) 个字节。这是除早期机器外的兼容机的典型情况。原始的 IBM PC 在内存可用区域的顶部设置了一块只读区域，在 8088 CPU 上最大为 1 MB。而在现代的兼容机上内存范围通常比早期的 PC 要大，但是为了兼容，它们仍在与原老式 PC 机相同的地址上保留一块只读内存。这样，可读写的内存就不连续了，在低 640 KB 和高 1 MB 的区域内夹入了一个块 ROM (只读存储器)。如果可能的话，引导监控程序会将内核加载到低段内存，而将服务器、驱动程序和 *init* 加载到高于 ROM 的内存段。这主要是为文件系统考虑的，因为这样做可以有一大块缓存可用，而不需要跳过只读区。

此外，需要指出操作系统并不仅仅可以从本地硬盘装入。无盘工作站可以通过网络从远地硬盘装入操作系统，当然这要求 ROM 中具有网络软件。尽管具体细节各不相同，但大体上类似。ROM 代码必须能够从网络上获得一个包含完整操作系统的文件。如果 MINIX 3 通过这种方式装入，则

操作系统被装入内存中的初始化过程几乎无须修改。当然，这需要一个网络服务器以及一个经过修改的、能够从网络访问文件的文件系统。

2.6.7 系统初始化

早期的 MINIX 版本如果需要与旧的处理器兼容，则可被编译成 16 位模式，MINIX 3 保留了 16 位模式的代码。但是，此处讨论的版本以及在 CD-ROM 中发布的版本针对的机器均只是配有 80386 或者更高级的处理器的 32 位机。它不支持 16 位模式，创建 16 位版本需要抛弃一些特性。另外，32 位可执行程序比 16 位的要大，而且独立的用户空间驱动程序不能共享以前包含驱动程序的单个二进制文件中的代码。这两种模式使用相同的 C 源代码，根据编译器本身是 16 位或 32 位而产生相应的输出。由编译器本身定义的一个宏确定 *include/minix/sys_config.h* 文件中宏 *_WORD_SIZE* 的值。

MINIX 3 执行代码的第一部分是用汇编语言写的，并且针对 16 位或 32 位编译器必须使用不同的源文件。初始化代码的 32 位版本位于文件 *mpx386.s* 中。对应的 16 位系统则位于文件 *mpx88.s* 中。这两者同时也都包含对其他低层内核操作的汇编语言支持。16 位和 32 位的选择在文件 *mpx.s* 中自动完成。*mpx.s* 很短，其内容示于图 2.38。

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

图 2.38 如何选择不同的汇编语言源文件

mpx.s 示出了 C 语言预处理器中 `#include` 语句的一种非常规用法。通常 `#include` 用来包含头文件，但也可以用来选择源代码的适当部分。使用 `#if` 语句来做到这一点将需要把两个很大的文件 *mpx88.s* 和 *mpx386.s* 放在一个单独的文件中。这样做不仅不实用，而且会浪费磁盘空间。因为在特定的安装中这两个文件中的一个可能根本就用不到，而且可以被归档或删除。在随后的讨论中，我们将以 32 位的 *mpx386.s* 为例。

由于这是首次接触可执行代码，所以先讲一下在本书中是如何对可执行代码进行讨论的。同时理解编译一个大的 C 程序时使用多个源文件是很困难的，所以本书通常一次只讲述一个文件。附录 B 中文件的顺序正是本书中讲述这些文件的次序。首先从 MINIX 3 系统各部分的入口点开始并跟随执行主线前进。当遇到调用一个支持函数时，将首先简略讲一下调用的意图，但并不对其进行详细的内部描述，而是直到被调用函数的定义处才详细地对其进行讨论。重要的从属函数通常定义在其被调用的同一个文件中，其前边是发出调用的高层函数。但小的或通用的函数有时被集中放在单独的文件中。而且本书不会试图探讨每个函数的内部特征，包含这些函数的文件也未列在附录 B 中。

同时，出于移植性的考虑，与机器相关和无关的代码尽量分开放在单独的文件中。为了使代码更容易理解且减小列表大小，绝大多数与非 Intel 32 位系统平台相关的选择性代码都被删除而未列入附录 B 中。所有文件的完整版本放在了 CD-ROM 的源代码目录中，在 MINIX 3 网站上也可以查到。

组织代码费了很多功夫，实际上在本书的写作过程中为了方便读者重写了许多文件。然而大程序有许多分支，有时理解一个主函数需要读懂它所调用的函数，所以有时使用一些纸片作为书签，偏离讨论次序而以另一种顺序来观察问题，可能会对理解函数有所帮助。

在讲完代码组织方式之后，现在言归正传。MINIX 3 的启动牵涉到几次控制权转移，它们发生在 *mpx386.s* 中的汇编语言例程和 *start.c* 及 *main.c* 中的 C 语言例程之间。本书将按执行次序进行讨论，尽管这需要从一个文件跳到另一个文件。

一旦引导进程将操作系统装入内存，控制权便转到标号 *MINIX*（在 *mpx386.s* 中，6420 行）。第一条指令是一条跃过几个字节数据的跳转指令。这几个字节数据包括先前提到过的引导监控程序标志（6423 行）。此刻，该标志已经在为目标服务；当引导监控程序将内核加载到内存时，它会读取这些标志。因为它曾被指定为该位置，所以它们会定位于此。该标志由引导监控程序用来标识内核的不同特征，其中最重要的是 16 位或 32 位系统标志。引导监控程序总是从 16 位模式开始，但在需要时会将 CPU 切换到 32 位模式。这个切换发生在控制权传给标号 *MINIX* 之前。

理解栈的状态有助于理解接下来的代码。监控程序通过将一些参数入栈而传递给 MINIX 3。首先，监控程序将变量 *aout* 的地址压入栈中，*aout* 保存了一个数组的地址，该数组存储了引导映像的组件程序的头信息。然后将其大小及引导参数的地址依次入栈，全是 32 位的属性值。最后是监控程序的代码段地址和监控程序执行到的位置，MINIX 3 结束后返回时要用到它们。这些都是 16 位特性，因为监控程序在 16 位保护模式下执行。然后通过指令

```
mov    ebp, esp
```

（6436 行）将栈指针复制到 *ebp* 寄存器，所以它能够通过偏移量找回监控程序存储在栈里（6464 到 6467 行）的这些现场数据。注意，因为 Intel 处理器的栈是向下生长的，*8(ebp)* 指的是从这个位置一直到 *12(ebp)* 依次存储的值。

汇编代码需要做许多工作，包括建立一个栈帧以便为 C 编译器编译的代码提供适当的环境，复制处理器所使用的表格来定义存储器段，建立各种处理器寄存器，等等。待这些工作结束后，初始化过程通过调用（6481 行）C 函数 *cstart*（在 *start.c* 中，下一节将介绍）继续进行。注意在汇编语言代码中用 *_cstart* 引用该函数。这是因为 C 编译器编译的所有函数在符号表中其名字前都有一个下划线，而且当编译好的分立模块被链接时，链接程序会查找这样的名字。由于汇编器并不自动地添加这个下划线，所以汇编语言程序员必须显式地加上这个下划线以保证在目标文件中能够找到相应的符号名。

cstart 调用另一个例程来初始化全局描述符表（Global Descriptor Table），这是 Intel 32 位处理器实现保护模式的内核数据结构；以及中断描述符表（Interrupt Descriptor Table），它用来为每种可能的中断类型选择执行代码。在从 *cstart* 返回之后，*lgdt* 和 *lidt* 指令（6487 和 6488 行）通过向其对应的寻址寄存器装入相应的值来将这些表格激活。指令

```
jmpf    CS_SELECTOR:csinit
```

初看起来像是空操作，因为好像在其所处的地方有一个 *nop* 指令序列一样。它把控制转到当时控制所在的位置，但这是初始化过程的一个重要部分。这个跳转指令强制使用刚刚被初始化的结构。在对处理器寄存器做进一步的操作之后，*MINIX* 以 6503 行的一个跳转（不是调用）操作结束，这样便跳转到了内核的 *main* 入口点（在 *main.c* 中）。*mpx386.s* 中的初始化代码在该处结束。该文件剩余部分的代码用来启动或重启动一个系统任务、进程、中断处理程序或其他的支持例程。出于效率的原因，它们均用汇编语言编写，下一节还将对它们进行讨论。

现在来看高层的 C 初始化函数。这里最基本的策略是用高层的 C 代码做尽可能多的操作。可以发现，这里已经有了两种版本的 *mpx* 代码，如果将 C 代码可能执行的操作都由 C 代码承担，则将省去两大段汇编代码。*cstart* 做的第一件事（在 *start.c* 中，6920 行）是调用 *prot_init* 来建立 CPU 的

保护机制和中断表。然后将引导参数复制到内存的内核部分，并使用函数 *get_value* (6997行) 扫描它们来查找参数名并返回相应的字符串值。它还确定显示器的型号、处理器类型、总线类型、是否16位模式、处理器操作模式（实模式还是保护模式）等。所有的信息都保存在适当的全局变量中，这是为了使内核代码的所有部分在需要时都能够访问到它们。

main 函数（在 *main.c* 中，7130行）完成初始化，然后开始系统的正常运行。它调用 *intr_init* 来配置中断控制硬件。该操作之所以放在这里是因为此前必须知道机器类型（因为 *intr_init* 调用完全依赖于硬件，所以该过程放在一个独立文件中）。该调用中的参数(1)指示 *intr_init* 这是在为 MINIX 3 执行初始化，MINIX 3 终止并将控制权返回给引导监控程序时可以通过使用参数(0)再次初始化硬件，使其回到原始状态。*intr_init* 保证在初始化完成之前的任何中断都不会生效。稍后我们将会讲述它是如何实现此功能的。

main 代码的主要部分用来建立进程表和特权表，这样当调度到第一批任务和进程时，它们的内存镜像和寄存器及特权信息将能够被正确地设置。进程表的所有表项都被标志为空闲；用于加快进程表访问的 7150 到 7154 行的循环用来初始化 *pproc_addr* 数组，该数组用来加快进程表的访问。7155 到 7159 行的循环清除了特权表以及与进程表及其项数组类似的 *ppriv_addr* 数组。对于进程表和特权表而言，在它们的一个域中放入一个特定值可以将其标记为未被使用。但是每一个表项，无论是否被使用，都需要使用一个索引号来初始化。

顺便提及 C 语言的一个小特性：在 7153 行处的代码

```
(pproc_addr + NR_TASKS)[i] = rp;
```

还可以写为

```
pproc_addr[i + NR_TASKS] = rp;
```

因为在 C 语言中 *a[i]* 只是 **(&a + i)* 的另一种写法。所以如果给 *a* 或 *i* 加一个常量将会得到同样的效果。对某些 C 编译器而言，给数组加一个常量将比给下标加一个常量产生稍微快一些的代码。不管是否是这样，此处不再介绍。

现在我们来看 7172 到 7242 行的长循环，它使用运行引导映像所需的必要信息来初始化进程表（注意，在 7161 行还有一个只描述了任务和服务器的旧版本）。所有这些进程在启动时必须存在，而且在正常执行过程中不应该终止。在循环的开始处，*ip* 被设置为 *table.c* 中创建的镜像表项的地址（7173 行）。因为 *ip* 是一个指向结构体的指针，所以该结构体的元素可以使用类似 *ip->proc_nr* 的标记来访问，见 7174 行。在 MINIX 3 源代码中广泛使用这种记法。同样，*rp* 是一个指向进程表项的指针，而 *priv(rp)* 指向特权表的表项。在这个长循环中，进程表和特权表的初始化很多都是由两个过程组成的：先从镜像表中读取一个数值，然后将其存入进程表或者特权表。

7185 行对内核部分的进程进行测试，如果为真，则在任务栈的基地址存储为特定的 *STACK_GUARD* 模式。这样可以稍后检查以确保栈没有溢出。然后为每一个任务建立初始的栈指针。每个任务都需要自己私有的栈指针。因为在内存中，栈是向低地址生长的，所以初始栈指针可以通过当前基地址和任务栈的大小相加来得到（7190 到 7191 行）。但是有一个例外：*KERNEL* 进程（有些地方也标识为 *HARDWARE*）一直都是非就绪态，永远不作为一个普通进程运行，也不需要栈指针。

二进制的引导映像部件同其他任何 MINIX 3 程序一样进行编译，编译器在每个文件的开头生成一个头文件，如 *include/a.out.h* 中所定义的那样。引导加载程序在 MINIX 3 启动之前将每个头文件复制到它的内存区，而此时监控程序将控制权交给 MINIX：以前曾看到过，在 *mpx386.s* 中头文件区域的物理地址传递给栈中的汇编代码。在 7202 行，其中一个头文件复制到当地的 *exec* 结构体 *ehdr*，

并使用 *hdrindex* 作为头文件数组的索引。然后数据段和文本段的地址被转换为 *clicks* 并进入这个进程的内存映射区域（7205 到 7214 行）。

在进一步继续之前，应提一下某些指针。首先，在 7178 行的 *hdrindex* 总是被赋予 0。进程全部编译进了内核文件，而有关栈请求的信息都在镜像表中。由于任务被编译进内核并且可以调用内核中的任意代码和访问内核中任意处的数据，所以一个独立任务的大小是没有什么意义的。内核和每一个任务都读取 *aout* 中数组的同一元素，而一个任务的表征其大小的域则按内核大小进行赋值。任务从镜像表中获取它们的栈信息，在编译 *table.c* 时进行初始化。处理完所有的内核进程后，该循环每执行一次，*hdrindex* 就加 1（7196 行），所以所有的用户空间进程都可以从它们的头文件中得到正确的数据。

还要说明的一点是，进行数据复制的函数没有必要与指定的源和目标顺序一致。研读这个循环代码时，要小心潜意识的混淆。标准 C 的库函数 *strncpy* 的参数中目标地址在最前面：*strncpy(to, from, count)*。这类似于分配操作，左侧的参数指定了要被赋值的变量，右侧的参数是代表所赋的值的表达式。为了调试和其他目的，在 7179 行使用该函数将进程名复制到进程表项中。作为对比可以看到，函数 *phys_copy* 使用相反的约定 *phys_copy(from, to, quantity)*。在 7202 行使用 *phys_copy* 复制用户空间进程的头文件。

下面继续关于进程表初始化的讨论，在 7220 和 7221 行，设置了程序计数器的初始值和处理器状态字。任务的处理器状态字与设备驱动程序和服务器是不同的，因为任务拥有高优先级，可以访问 I/O 端口。而如果是一个用户空间进程，它的栈指针则被初始化。

进程表中有一个入口不需要（也不能够）被调度。进程 *HARDWARE* 只是为了记账操作的目的。它记录中断服务所用的时间。所有其他进程由 7234 到 7235 行的代码放置在适当的队列中。函数 *lock_enqueue* 在修改队列之前关闭中断，修改完后再打开它们。当什么都没运行时是不需要这样的，但这是标准方法，而且没有必要为一种只出现一次的情况额外编写代码。

对进程表中的每个表项进行初始化的最后一步是在 7241 行调用 *alloc_segments*。这是一个与机器相关的过程，它将各进程使用的内存段的位置、大小及运行特权级设置到适当的域中。对于旧式的不支持保护模式的 Intel 处理器，它只定义段地址。对一种内存分配方法相异的处理器类型，必须重写 *alloc_segments*。

一旦对所有任务、服务器和 *init* 初始化了进程表，系统基本上就可以运行了。变量 *bill_ptr* 标明对哪个进程进行 CPU 使用的计费，它需要一个初值。该初值在 7250 行赋值，此时，*IDLE* 是一个合适的选择。此时内核已经可以正常工作，如图 2.2 中所示的控制和调用进程的执行。

虽然并非系统所有的其他部分都已能够正常运行，但是它们都已能作为独立的进程运行，并被标记为就绪和等待运行。当它们运行时会初始化自身。内核还要做的就是调用 *announce* 来声明它已就绪，然后调用 *restart*（7251 到 7252 行）。在许多 C 程序中 *main* 是一个循环，但在 MINIX 3 内核中，它的工作只到初始化结束为止。7252 行中对 *restart* 的调用将启动第一个任务，控制权从此不再返回到 *main*。

_restart 是 *mpx386.s* 中的一个汇编语言例程。实际上，*_restart* 不是一个完整的函数，它是一个更大的过程的中间入口。下一节将详细讨论该过程。这里仅指出 *_restart* 引发一个上下文切换，这样就将运行 *proc_ptr* 所指向的进程。当 *_restart* 首次执行后，可以说 MINIX 3 正在运行，即它在执行一个进程。每当系统任务、服务器进程或用户进程得到了运行机会然后又被挂起时，都会反复执行 *_restart*，无论挂起原因是等待输入还是在轮到其他进程运行时将控制权转交给它们。

当然，首次执行 *_restart* 时，只完成了内核的初始化。在 MINIX 3 的进程表中有三个部分。读者也许会问在进程表的主要部分还没建立时，进程怎么能够运行呢？完整的答案要等到下一章，简

要地说，是因为引导映像中所有进程的指令指针初始化为指向进程的初始代码，并且很快就会阻塞。最后，内存管理器和文件系统便可以运行它们的初始代码，而其进程表部分随后会建立起来。最后 *init* 将为每个终端创建一个 *getty* 进程，这些 *getty* 进程将一直阻塞，直到终端上有键入的输入，此时第一个用户便能登录。

至此已经从三个文件跟踪了 MINIX 3 的启动过程，两个使用 C 语言，一个使用汇编语言。*mpx386.s* 文件中包含了附加的用于中断处理的代码，这些代码将在下一节中说明。但在继续之前，我们先简短地了解一下这两个 C 文件中的其余例程。在 *start.c* 中剩下的函数是 *get_value* (6997 行)，它用于在内核环境下查找项，这是镜像参数的一个副本。为了使内核简单，此处是一个重写过的标准库函数的简化版本。

在 *main.c* 中还有三个其他程序。*announce* 显示版权信息以及 MINIX 3 是运行于实模式还是 16 位或 32 位的保护模式，如下所示：

```
MINIX 3.1 Copyright 2006 Vrije Universiteit, Amsterdam, The Netherlands
Executing in 32-bit protected mode
```

当读者看到这条信息时就知道内核初始化已经完成。*prepare_shutdown* (7272 行) 使用 *SIGKSTOP* 信号（使用信号通知系统进程的方式不同于通知用户进程）通知所有的系统进程。然后它设置一个定时器，并允许所有的系统进程设置执行清理的时间，该时间需在调用最后一个程序 *shutdown* 之前。*shutdown* 正常情况下会将控制权交给 MINIX 3 引导监控程序。为此，中断控制器会重新保存到 BIOS，而 BIOS 在 7338 行通过调用 *intr_init()* 进行设置。

2.6.8 MINIX 的中断处理

中断硬件的细节与系统相关，但任何系统都必须具有与 Intel 32 位 CPU 功能等价的部件。由硬件设备产生的中断是一些电信号，它们首先由中断控制器进行处理。中断控制器是一片集成电路，它能够检测到许多此类电信号并在处理器的数据总线上为其生成唯一的数据格式。这些数据格式之所以必须唯一是因为：对所有这些设备，处理器本身只能有一个输入，所以它无法辨认需要服务的具体设备。使用 32 位处理器的 PC 机通常有两片中断控制器芯片，其中每一片可以处理 8 个输入，但其中有一片为从片，它的输出线连到主片的一条输入线，这样一共可以挂接 15 个不同的外部设备，如图 2.39 所示。在这 15 个输入中，有一些输入是专有的。例如，时钟输入线 (IRQ 0) 不连接任何可以插入新适配器的插槽。而另外一些输入连接到插槽上，这个插槽可以插入任何设备。

图 2.39 中，中断信号出现在右侧的 *IRQ n* 信号线上。连到 CPU 的 INT 管脚的连接线通知 CPU 发生了中断。从 CPU 发出的 INTA (中断应答) 信号使负责中断的控制器芯片将数据放在系统数据总线上，并通知处理器应执行哪个服务例程。在系统初始化期间，当 *main* 调用 *intr_init* 时，对中断控制器进行编程。这种编程内容决定了对于各条输入线的信号将向 CPU 送出什么样的数据，同时也决定了中断控制器操作所用的其他参数。放在总线上的数据是一个 8 位 (二进制) 的数值，它用做对一个表格的索引，该表格最多可包含 256 项。MINIX 3 的表格中含有 56 个表项，其中实际用到 35 项，其余 21 项保留供将来的 Intel 处理器或 MINIX 3 扩展使用。在 32 位的 Intel 处理器上，这张表中包含中断门描述符，每个中断门描述符是一个含有若干域的 8 字节结构。

对中断有几种可能的响应方式；在 MINIX 3 使用的一种模式中，中断门描述符中最重要的一个域指向服务例程可执行代码段和其中的起始地址。CPU 执行被选中的描述符所指向的代码，其结果与执行如下的汇编语言指令完全相同：

```
int    <nnn>
```

唯一的差别在于，对于硬件中断，<nnn>来自中断控制器芯片的一个寄存器，而不是程序内存中的一条指令。

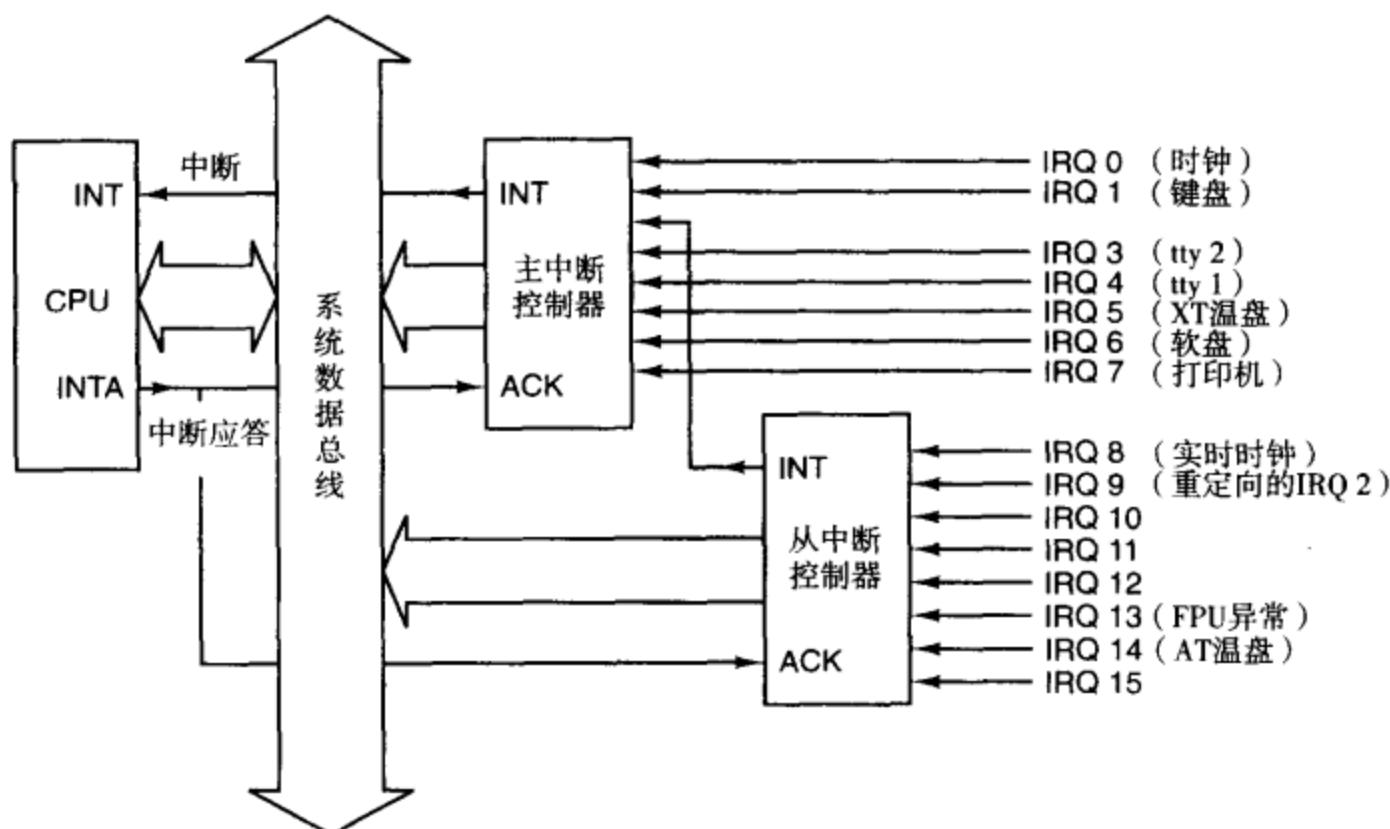


图 2.39 一台 32 位 Intel PC 上的中断处理硬件

32 位 Intel 处理器响应中断时的任务切换机制很复杂，改变程序计数器以执行另一个函数只是其中很小的一部分。当 CPU 在一个进程运行期间接收到一个中断时，它将建立一个新堆栈供中断服务器使用。该堆栈的位置由任务状态段（Task State Segment, TSS）中的一项决定。整个系统中有这样一个结构，它在 *cstart* 调用 *prot_init* 时进行初始化，并且在每个进程启动时被修改。其结果是每个中断创建的新堆栈总是从被中断进程的进程表项中 *stackframe_s* 结构的结尾处开始。CPU 自动地将几个关键寄存器值压入新堆栈，包括用来恢复被中断进程本身堆栈及其程序计数器的那些寄存器。当中断处理例程代码开始运行时，它使用进程表中的这个区域作为自己的堆栈，同时也保存了返回被中断进程所需的大部分信息。中断处理例程将其余寄存器的内容压栈，填充栈帧结构，然后切换到一个由内核提供的堆栈，以便在进行中断服务过程中使用它。

中断服务例程的结束操作如下：从内核栈切换回进程表项中的栈帧结构（它不必是最后一次中断所创建的那个），显式地弹出非硬件压栈的寄存器值，并执行一条 *iretd*（从中断返回）指令。*iretd* 恢复中断前的状态，恢复被硬件压栈的寄存器，并切换回中断前使用的堆栈。由此可知，中断停止一个进程，而中断服务结束则重启一个进程。被启动的进程可能不是最近被停止的那个进程。与一般的汇编语言教材中所讲述的较简单的中断机制不同，在中断期间，并不在被中断进程的工作堆栈上保存任何内容。进一步而言，由于在中断之后堆栈被重新创建于一个已知的位置（由 TSS 决定），所以多进程的控制得以简化。启动另一个进程所需的是：将堆栈指针指向该进程的栈帧结构，弹出先前显式压栈的寄存器值，然后执行一条 *iretd* 指令。

当接收到一个中断时，CPU 关掉所有的中断，这样就保证了进程表项中的栈帧不会溢出。这个过程是自动进行的，但也存在可以关中断和开中断的汇编指令。当使用位于进程表项之外的内核栈时，中断保持关闭状态。存在一种机制，当在使用内核栈时，能够使异常（CPU 检测到错误后做出的一种响应）处理程序运行。异常和中断类似，但是异常是不可屏蔽的。为了使异常得到响应，必须有一种处理中断嵌套的方式。在这种情况下并不创建新的堆栈，相反，CPU 把那些为继续运

行被打断的代码所必需的寄存器压到现有的栈中。内核运行的时候不应该发生异常，如果发生将会崩溃（panic）。

当执行内核代码时，若遇到 `iretd` 指令，则使用的返回机制比返回用户进程要简单一些。处理器能够通过检查从堆栈弹出的代码段选择符来决定如何处理 `iretd` 指令。

前面提到的特权级控制在运行进程和执行内核代码（包括中断服务例程）时，会对接收到的中断做出不同响应。当被中断代码的特权级和中断后即将执行代码的特权级相同时，使用较简单的机制。然而，通常情况下被中断代码的特权级比中断服务器的特权级要低，在这种情况下，使用较复杂的机制。该机制用到 TSS 和一个新堆栈。一个代码段的特权级记录在代码段选择符中，并且因为这是中断过程中压栈的内容之一，所以从中断返回时可以检查其内容，以决定 `iretd` 指令将执行什么操作。

当中断服务期间创建一个新堆栈供使用时，硬件提供另外一种服务。硬件检查并保证这个新堆栈起码能够装得下需放在其中的最小的信息集合。这样就保护了特权级更高的内核代码不会因为进行系统调用的用户进程堆栈空间不够而崩溃。专门运行多进程操作系统的处理器特意将这种机制固化在处理器中。

如果不清楚 32 位 Intel CPU 的内部工作原理，则会对这种操作过程感到困惑。一般情况下，应尽量避免描述这类细节，但如能理解发生中断和执行 `iretd` 指令时所发生的处理细节，则对理解内核如何控制进入和离开图 2.2 中的“运行”态是非常有帮助的。硬件完成其中许多工作，这极大地简化了程序员的工作，并使得系统更加高效，但却使得通过读软件代码来理解其内部操作变得更加困难。

在讲解了中断机制以后，下面将回到 `mpx386.s` 中来，看一下 MINIX 3 内核中实际处理硬件中断的代码。每个中断都有一个入口。从 `_hwint00` 到 `_hwint07`（6531 到 6560 行）的所有入口点的源代码看起来像是调用 `hwint_master`（6515 行），从 `_hwint08` 到 `_hwintl5`（6583 到 6612 行）的入口点像是调用 `hwint_slave`（6566 行）。每个人口点好像在该调用中传递一个参数，指明需要服务的设备。实际上这些不是调用，而是宏。由宏 `hwint_master` 定义的代码的 8 份副本很类似，只有参数 `irq` 不同。同样，`hwint_slave` 的 8 份副本也很类似。这似乎有些浪费，但这些类似的代码非常紧凑。每个宏展开后均不到 40 字节。在中断服务时速度是最重要的，而且这样做省掉了装入参数、调用子例程和检查参数的开销。

下面将继续把 `hwint_master` 当做单个函数来看待，而不把它视为在 8 个不同地方展开的宏。回忆一下在 `hwint_master` 开始执行之前，CPU 已经在被中断进程的进程表项的 `stackframe_s` 结构中建立了一个新堆栈。若干关键寄存器值已经保存在那里，而且所有的中断都处于关闭状态。`hwint_master` 的第一个动作是调用 `save`（6516 行），该子例程将以后重启动被中断进程所需的所有其他寄存器值压栈。`save` 其实可以作为内联代码写成宏的一部分以加快速度，但这样做将使宏的大小增加一倍以上，而且 `save` 也需要被其他函数使用。可以看到，`save` 在堆栈上做了一个小把戏。当返回 `hwint_master` 时，将使用内核栈而不是进程表项中的栈帧结构。

在这里要用到 `glo.h` 中所声明的两个表。`_irq_handlers` 包含了钩子（hook）信息，其中包括处理例程的地址。将要服务的中断号转化成 `_irq_handlers` 内的地址，这个地址作为 `_intr_handle` 的参数压栈，然后调用 `_intr_handle`。后面还会介绍 `_intr_handle` 的代码。现在需要说明的是，它不仅为调用的中断调用服务例程，还置位或复位 `_irq_actids` 数组中的标志来说明这次服务中断的尝试是否成功，它给队列中的其他表项一个运行并从列表中移除的机会。在对 `_intr_handle` 的调用返回之前是否接受其他中断，取决于中断处理程序的需要。可以通过检查 `_irq_actids` 中的相应入口来决定。

如果 *_irq_act_ids* 数组中的某一入口值非零，那么说明 IRQ 对应的中断服务器尚未完成。如果是这样，中断控制器将不再响应来自同一 IRQ 线的其他中断（6722 到 6724 行）。这通过屏蔽中断控制器对某一输入的响应来实现。当 CPU 接收到中断信号后且在没有恢复之前，CPU 对所有中断的响应能力在内部被关闭了。

对于不熟悉汇编语言的读者来说，简要介绍一下汇编代码或许很有帮助。6521 行的指令

```
jz 0f
```

没有用几个字节来指定跳转的地址。*0f* 不是一个十六进制数，也不是一个普通标号。普通标号不允许以数字开头。这实际上是 MINIX 3 汇编器指定一个局部标号（local label）的方式，*0f* 表示跳转到前面的下一个数字标号 *0*（6525 行）处。在 6526 行上的代码使中断控制器继续正常操作，此时当前中断的中断信号线可能被屏蔽。

一个有趣但可能令人费解的一点是，在 6525 行的标号 *0* 也出现在了同一文件的其他地方：在 *hwint_slave* 的 6576 行。实际情况可能比第一眼看上去的更复杂，因为这些标号出现在了宏里，这些宏将在汇编器处理之前被展开。因此实际上汇编器将看到代码中有 16 个标号 *0*。在宏中定义的标号可能会增殖，这就是汇编语言中为什么提供局部标号的原因。当处理局部标号时，汇编器将使用给定方向上的最近匹配的标号，其他地方出现的标号将被忽略。

_intr_handle 是硬件相关的，详细代码将在叙及 *i8259.c* 时讨论。但是现在简要说明一下它如何工作还是适宜的。*_intr_handle* 扫描一个链接起来的数据结构，每个数据结构中有处理设备中断需要调用的函数地址，以及设备驱动程序的进程号。它是一个链表是因为每个 IRQ 线可以由多个设备共享。每个设备的驱动程序负责测试自己的设备是否需要服务。当然，像 IRQ 0 上的时钟中断不需要这一步，因为 IRQ 0 线硬连接在产生时钟信号的芯片上，其他设备不可能触发这一 IRQ。

中断处理程序应该以一种可以很快返回的方式处理。如果没有工作需要做或者中断服务器迅速完成，则处理程序返回 *TRUE*。处理程序可以进行一些操作，例如从一个输入设备读入数据，并把数据传送到缓存中，相应的驱动程序在下次运行的时候可以读取。然后处理程序向设备驱动程序发送一个消息，以使设备驱动程序能够作为一个正常进程调度。如果工作没有完成，处理程序返回 *FALSE*。*_irq_act_ids* 数组中有一个成员是位图，位图记录了链表中所有的处理程序的结构，当且仅当所有的处理程序返回 *TRUE* 时，结果才为 0。如果不是这种情况，6522 到 6524 行的代码将在 6536 行的代码重新启用中断控制器前，关闭该 IRQ。

这一机制保证了在它们所有的设备驱动程序完成工作之前，链表中共享同一中断请求线的中断处理程序不会被激活。显然还需要一种启用 IRQ 的方法。后面将会看到的 *enable_irq* 函数提供了这一功能。每个设备驱动程序必须保证在完成工作之后调用 *enable_irq*。显然，*enable_irq* 应该首先复位 *_irq_act_ids* 中驱动程序的 IRQ 所对应元素本身的位，然后检测该元素的所有位是否都已复位。只有在所有位都复位时，中断控制器的 IRQ 才能重新启用。

上面讨论的这些简单的形式仅仅对时钟驱动适用，因为时钟是编译进内核的唯一中断驱动的设备。其他进程中中断处理程序的地址在内核上下文是没有意义的，内核中的 *enable_irq* 函数在一个独立进程本身的内存空间内是不能调用的。对于用户空间的驱动程序，即除时钟驱动之外的所有响应硬件触发的中断的设备驱动程序，一个通用的处理函数 *generic_handler* 的地址存储在钩子列表中。这个函数的源代码在系统任务文件中，但是由于系统任务和内核一起编译，而且这段代码作为中断的响应执行，它不能视为系统任务的一部分。钩子列表中的其他元素还包括设备驱动程序相关的进程号。当调用 *generic_handler* 时，它向对应的设备驱动程序发送一条消息，从而使驱动程序的特定处理函数运行。系统任务还支持上述事件的相反序列。当用户空间设备驱动程序完成

任务时，它发出一个 `sys_irqctl` 内核调用，从而使系统任务代表这个驱动发出 `enable_irq` 调用，为下一个中断做准备。

现在看一下 `hwint_master`。注意，它以一条 `ret` 指令（6527行）结束。这里有一个不很明显的小把戏。当一个进程被打断时，这时使用的是内核堆栈，而不是在 `hwinit_master` 调用前由硬件建立的进程表中的堆栈。在这种情况下，通过 `save` 对堆栈的操作将使 `_restart` 的地址保存在内核栈中，从而使任务、驱动程序、服务器或用户进程可以继续运行。但返回的可能不是（实际上非常可能不是）中断发生时所执行的进程。这取决于由硬件相关的中断服务例程产生的消息的处理是否会在进程调度队列中引起变化。在硬件中断的情况下，几乎肯定是这种情况。中断处理程序产生向驱动程序发出的消息，而设备驱动程序通常位于比用户进程更高的优先级队列中。这就是造成有多个进程同时运行的假象的核心机制。

在结束之前，再讲一下当内核代码正在执行时发生中断的情况，此时正在使用的是内核栈，`save` 将 `restart1` 的地址保存在内核栈中。在这种情况下，内核先前正在执行的操作将在 `hwint_master` 末尾的 `ret` 指令之后继续下去。这是对嵌套中断处理的描述，在 MINIX 3 中不会发生，因为内核堆栈运行时中断是关闭的。然而，如前所述，处理异常需要这一机制。当所有响应异常的内核例程完成之后，`_restart` 将会执行。当执行内核代码的时候响应异常时，几乎毫无疑问的是，最后一个不同于被打断的进程的新进程将会投入运行。内核中对异常的响应会引起崩溃，将会发生的是尝试关闭系统，并可能会造成一些损坏。

除了必须将主和从中断控制器全部重新启用外，`hwint_slave`（6566行）的行为很像 `hwint_master`。因为从中断控制器接收到中断后，会使这两者被屏蔽中断。

现在来看 `save`（6622行），对于它，前面已经提到过几次。它的名字指出了它的若干功能之一，即将一个中断进程的上下文保存在 CPU 提供的堆栈上，该堆栈是进程表中的一个栈帧结构。`save` 用变量 `_k_reenter` 来计算和确定中断的嵌套级数。如果当前中断发生时一个进程正在运行，则 6635 行的指令

```
mov    esp, k_stktop
```

切换到内核栈，并且随后的指令将 `_restart` 的地址压栈。如果内核栈已在使用时发生中断，则将 `restart1` 的地址压栈（6642行）。当然这种情况不允许发生，但还是存在这样一种机制来处理异常。不论哪种情况，所使用的堆栈与入口时起作用的堆栈可能不同，而且调用它的例程中的返回地址被压在已压栈寄存器值的下边。这些都造成一条普通的 `return` 指令不足以返回到调用者。位于 6638 和 6643 行的结束 `save` 操作的两个出口点的指令

```
jmp    RETADR-P_STACKBASE(eax)
```

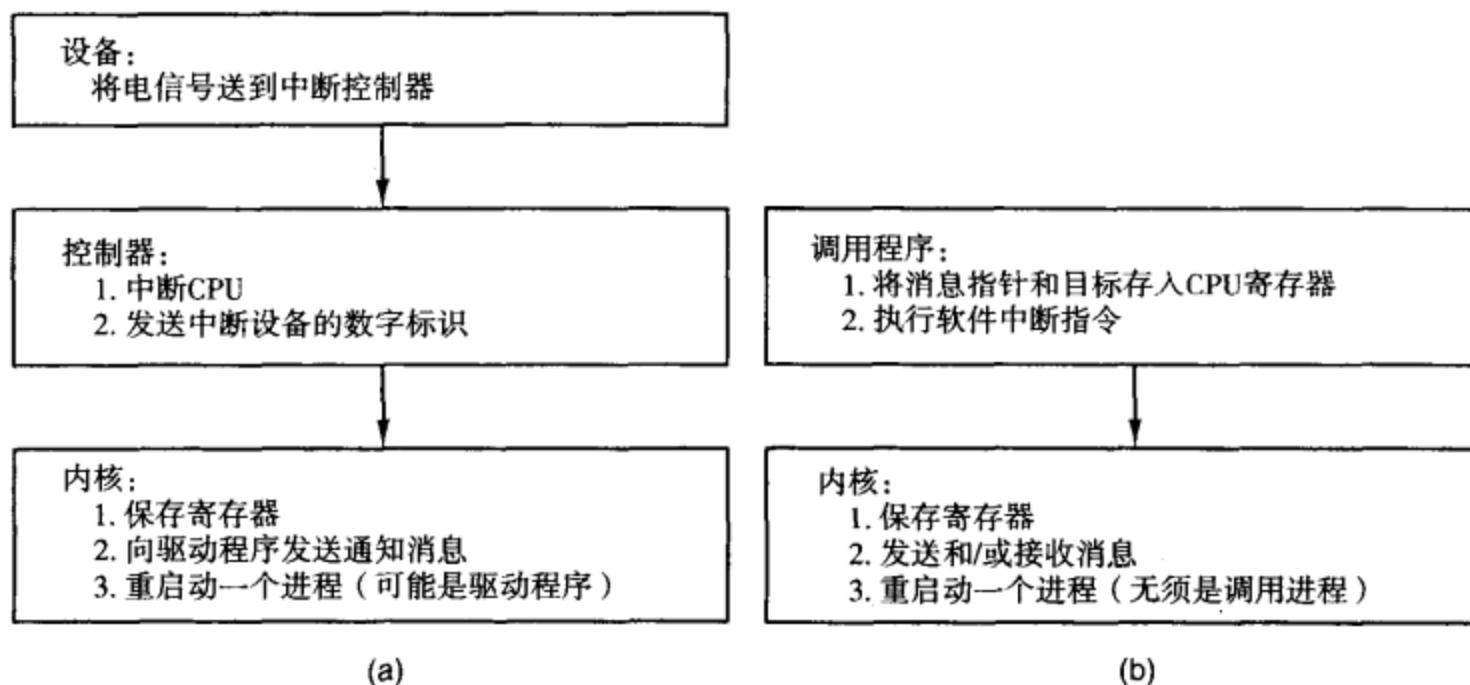
使用调用 `save` 时压进栈的地址。

内核重入会引起许多问题，消除它会使代码的许多地方变得简单。在 MINIX 3 中，`_k_reenter` 变量还有一个用途，尽管内核代码执行的时候普通中断不会发生，但发生异常还是可能的。到目前为止，必须记住在正常的操作当中，6634 行的 `jump` 语句不会发生，但发生异常时需要它。

作为旁白，这里必须承认对内核重入的消除是在 MINIX 3 的发展中代码超前文档的一个例子。在某些方面，做文档比编代码更难。编译器或程序会最终揭示代码中的错误，却没有这样一种机制来更正源代码中的注释。在 `mpx386.s` 的开头部分有很长一段注释，但很遗憾，它是错误的。6310 到 6315 行的那部分应当说，只有在检测到异常时，内核重人才会发生。

`mpx386.s` 中的下一个过程是 `_s_call`，它从 6649 行开始。在讲述其内部细节之前，先看看它是如何结束的。其结束处没有用 `ret` 或 `jmp` 指令。实际上，程序在 `_restart`（6681行）处继续执行。`_s_call`

是中断处理机制在系统调用中的对等物。在软件中断即执行一条 `int <nnn>` 指令后，控制权转到 `_s_call`。对软件中断的处理类似硬件中断，不同之处在于中断描述符表的索引硬编码进了 `int <nnn>` 指令的 `nnn` 部分，而不是由中断控制器芯片提供。这样当进入 `_s_call` 时，CPU 已经切换到了进程表（由 TSS 提供）中的栈，并且几个寄存器值已经被压入该栈。直到执行 `_restart`，对 `_s_call` 的调用最终以一条 `iretd` 指令结束。而且，正如硬件中断一样，该指令将启动 `proc_ptr` 此时指向的进程。图 2.40 对硬件中断和使用软件中断机制的系统调用做了比较。



现在来看 `_s_call` 的细节。它还有另一个名字 `_p_s_call`，这是 16 位 MINIX 3 版本的残迹，16 位版本的 MINIX 3 分别具有保护模式和实模式操作例程。在 32 位版本中，对这两个标号的调用都到达这里。使用系统调用的程序员用 C 写的函数调用看起来与其他函数调用一样，不论调用的是一个本地定义的函数还是一个 C 库中的例程。支持一条系统调用的库函数代码构造一条消息，将消息的地址和目标进程标识号装入 CPU 的寄存器，然后调用一条 `int SYS386_VECTOR` 指令。如上所述，其结果是将控制转到 `_s_call` 的起始处，同时几个寄存器值已被压入进程表中的一个堆栈上。像硬件中断的处理一样，此时所有中断都关闭。

`_s_call` 代码的第一部分看起来像 `save` 函数的在线展开，并且将其余必须保存的寄存器值保存下来。正如在 `save` 中一样，执行一条

```
mov esp, k_stktop
```

指令，然后切换到内核栈（软件中断与硬件中断的相似之处还包括它们都关中断）。其后将调用 `_sys_call` (6672 行)，下节中将对其进行讨论。现在只需知道它将构造一条消息，然后发送该消息，这将进一步引发调度器运行。因此，当 `_sys_call` 返回时，`proc_ptr` 有可能指向引发本系统调用的进程之外的另一个进程。然后执行将又回到 `restart`。

我们已经看到有几种方法使执行线路到达 `_restart` 函数 (6681 行)：

1. 在系统启动时从 `main` 调用它。
2. 在硬件中断发生后从 `hwint_master` 或 `hwint_slave` 跳转到它。
3. 在系统调用发生后从 `_s_call` 调用它。

图 2.41 简要说明了控制是如何在进程和内核之间通过 `_restart` 传递的。

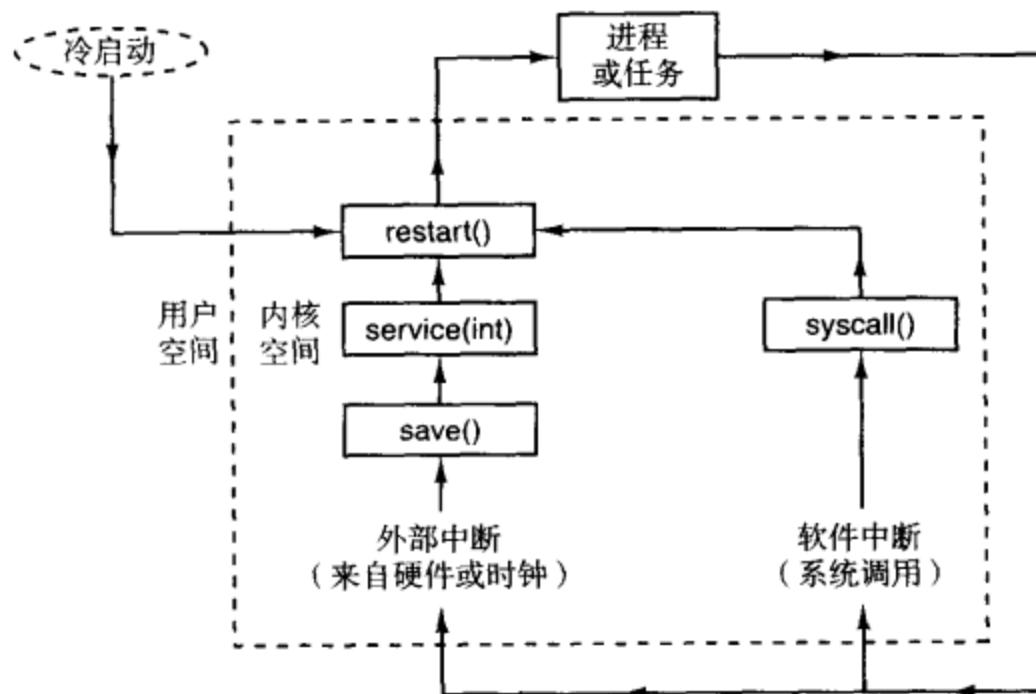


图 2.41 在系统启动、中断和系统调用后到达的公共点是 *Restart*。最需要 CPU 的进程（这可能是并且通常是与被中断的进程不同的进程）下一个运行。图中没有给出在内核运行时发生中断的情况

在以上所有情况中调用 *_restart* 时都要关中断。到 6690 行时已经最终选择了下一个运行的进程，因为中断是关闭的，下一个运行进程将不会被改变。进程表的结构是精心设计的，它以一个栈帧结构开始，因此指令

```
mov esp, _proc_ptr
```

就使 CPU 的栈指针寄存器指向该栈帧，指令

```
lldt P_LDT_SEL(esp)
```

随后从该栈帧中装入处理器的本地描述符表寄存器。这使得处理器为使用将运行进程的存储器段做好准备。接下来的指令将把该地址保存在下一个运行进程的进程表项中，下一次中断的堆栈将建立在该表项中。然后，随后的一条指令把这个地址保存在 TSS 中。

执行内核代码（包括中断服务代码）时会发生中断，因为将使用内核栈，所以 *_restart* 的第一部分便不再需要，而中断服务结束将使内核代码继续执行。但实际上在 MINIX 3 中内核是不可重入的，普通的中断不会以这种方式发生。然而关闭中断并不能使处理器失去检测异常的能力。标号 *restart1* (6694 行) 标示出当内核代码执行时发生异常（这时不希望看到的）后应从何处继续执行。在这一点上，*k_reenter* 被减 1，以记录一层可能的嵌套中断已被处理，然后其余指令将处理器恢复到下一进程的最后一次执行所处的状态。倒数第二条指令修改栈指针，这样使调用 *save* 时压栈的地址被忽略。如果最后一个中断是在进程执行时发生的，那么最后一条指令 *iretd* 将结束这一系列操作，返回到下一个进程应该执行的地方，恢复其剩余的寄存器，包括栈段寄存器和栈指针。但是如果遇到经由 *restart1* 的 *iretd*，则使用的就不是一个栈帧，而是内核栈。而且这种情况也就不再是向一个被中断进程的返回，而是在内核代码执行期间所发生中断的完成。在 *iretd* 执行期间，当代码段描述符从堆栈弹出时 CPU 将检测到这一点，而且在这种情况下，*iretd* 的全部动作是使内核栈继续保持使用。

现在到了介绍一下异常的时候了。异常（exception）是由 CPU 内部的各种错误引起的。异常并不总是坏事，它们可以支持操作系统提供一种服务，例如为进程提供更多的内存，或者将当前换出的内存页面换入等，尽管标准的 MINIX 3 并未实现这些服务。异常也可能由编程错误引起。在

内核中，异常是非常严重的，也是崩溃的基础。当异常在用户程序中发生时，这个程序可能终止，然而操作系统应该能够继续运行。异常的处理和中断机制相同，使用中断描述符表中的描述符。该表中的表项指向 16 个异常处理入口。这些入口从 *_divide_error* 开始，以 *_copr_error* 结束，位于 *mpx386.s* 尾部的 6707 到 6769 行。根据是否将一个错误码压栈，这些入口分别跳转到 *exception* (6774 行) 或 *errexception* (6785 行)。此处汇编代码的处理与已经看到的很相似，寄存器被压栈并且调用 C 例程 *_exception* (注意其中的下划线) 来处理该事件。异常导致的结果各不相同，有的被忽略，有的导致系统崩溃 (panics)，有的导致向进程发送消息。在下一节中，我们将讨论 *_exception*。

还有另外一个人口点像中断一样处理，即 *_level0_call* (6714 行)。代码必须在 0 特权级运行时使用它，特权级 0 是最高特权级。该入口点与中断和异常的入口点一起放在 *mx386.s* 中，这是因为也被 *int <nnn>* 指令所调用。和异常处理例程一样，它也调用 *save*，因此跳转到这里的代码最终也将以一条 *ret* 指令结束，这将最终引起 *_restart* 调用。它的用法将会在后面的章节中，当遇到需要在通常不可用的特权级（甚至对于内核也不可用）运行的代码时，加以介绍。

最后，在该汇编语言文件的末尾预留了一些数据存储空间。这里定义了两个不同的数据段。在 6822 行声明的数据段

```
.sect .rom
```

保证该存储空间被分配在内核数据段的最开始处，这是只读内存段的开始。编译器在这里放置一个魔数，以便 *boot* 能够验证其装入的文件是一个合法的内核镜像。在编译整个系统时，许多字符串常量将存储在这里。另一个数据存储区域

```
.sect .bss
```

(6825 行) 的声明，在内核的普通未初始化变量区为内核栈预留了空间，同时在这之上为异常处理函数使用的变量预留了地址空间。服务器进程和普通的用户进程在可执行文件被链接时预留堆栈空间，并在执行时依靠内核来适当地设置栈段描述符和堆栈指针。内核也必须为它们自己进行这种处理。

2.6.9 MINIX 3 的进程间通信

MINIX 3 中的进程使用消息进行通信，这里使用到了进程聚合 (rendezvous) 的原理。当一个进程执行 *send* 操作时，内核的最底层检查目标进程是否在等待从发送者（或任意发送者）发来的消息。如果是，则该消息从发送者的缓冲区复制到接收者的缓冲区，同时这两个进程都被标记为就绪态。如果目标进程未在等待消息，则发送者被标记为阻塞，并被挂入一个等待将消息发送到接收进程的进程队列中。

当一个进程执行 *receive* 操作时，内核检查该队列中是否存在向它发送消息的进程。若有，则消息从被阻塞的发送进程复制到接收进程，并将两者均标记为就绪；若不存在这样的进程，则接收进程被阻塞，直到一条消息到达。

在 MINIX 3 中，操作系统的各个部分作为独立的进程运行，有时这种聚合的方法工作得并不好。提供的 *notify* 原语正好适用这些情况。*notify* 发送一条基本 (bare-bones) 消息。如果接收者不在等待消息，发送者也不会阻塞。然而，通知也不会丢失。当目标进程下次运行时，挂起的通知将先于普通消息处理。通知可以用于那些使用普通消息将会引起死锁的情况。前面提到过进程 A 向进程 B 发送消息，同时进程 B 向进程 A 发送消息而导致阻塞的情况，这种情况是必须要避免的。但是如果其中一条消息是一个不会阻塞的通知的话，则不会有任何问题。

在大多数情况下，通知仅仅把它的发起者告诉接收者，仅此而已。有时这样就足够了，但在两种特殊情况下通知还需传递附加信息。在所有的情况下，目标进程都可以向通知的发起者发送消息，请求更多的信息。

进程间通信的高层代码在 *proc.c* 中。内核的任务是将一个硬件中断或软件中断转换为一条消息，前者由硬件产生，后者则是一个对系统服务的请求（即系统调用）向内核传递的途径。这两者很类似，以至于可以用同一个函数处理，但将其分成两个专门的函数会更高效。

需要注意一下这个文件开头的一条注释和两个宏定义。为了操作链表，广泛使用了指向指针的指针，7420到7436行的注释说明了它们的使用好处及用法。有两个重要的宏定义。*BuildMess* (7458 到 7471 行) 尽管从名字来看很通用，但实际上只是用来构造 *notify* 用到的消息。它调用的唯一函数是 *get_uptime*，它读取时钟任务所维持的一个变量，让这个通知中包含一个时间戳。对一个名为 *priv* 的函数的显式调用是 *priv.h* 中所定义的一个宏

```
#define priv(rp) ((rp)->p_priv)
```

的展开。另外一个宏 *CopyMess* 是一个对 *klib386.s* 中汇编语言例程 *cp_mess* 的编程者友好的接口。

还需要再解释一下宏 *BuildMess*。宏 *priv* 用于两种情形。如果通知的发起者是 *HARDWARE*，则它携带一个负载，即目标进程挂起中断位图的副本。如果发起者是 *SYSTEM*，则负载是挂起信号的位图。因为这些位图位于目标进程的 *priv* 表中，因此随时可以访问它们。当消息发送时，如果目标进程没有阻塞等待这些通知，那么通知可以稍后交付处理。对于一般消息而言，可能需要某种缓存以存储尚未交付处理的消息。为了存储一个通知，所需要的只是一个位图，位图中的每一位代表一个可能发送通知的进程。当通知不能发送时，接收者位图中代表发送进程的那一位置位。当接收者调用 *receive* 时，它将检查位图，如果发现位图中的某一位置位，将重建消息。这一位说明了消息的发起者，如果发起者是 *HARDWARE* 或 *SYSTEM*，则还添加了附加的内容。当消息重建时，唯一还需要的是一个时间戳。从时间戳的用途来看，时间戳不必显示这个通知是什么时候尝试发出的，显示什么时间交付处理已经足够了。

proc.c 中第一个函数是 *sys_call* (7480 行)。它将一个软中断（用来发出系统调用的 *int SYS386_VECTOR* 指令）转化为消息。有很多可能的消息发送者和接收者，这个系统调用可能需要发送消息或接收消息或两者都需要。这里需要做一系列的检查。在 7480 到 7481 行，从系统调用的第一个参数中抽出函数编码（*SEND*, *RECEIVE* 等）和标志。首先检查调用进程时允许发出这个调用。7501 行使用的 *iskerneln* 是文件 *proc.h* (5584 行) 定义的宏。然后检查的是指定的源进程或目标进程是否是一个有效的进程，再检查消息的指针指向内存中的一个有效区域。MINIX 3 特权定义了一个给定的进程允许向哪些进程发送消息，这就是下一个检查 (7537 到 7541 行)。最后的检查是验证目标进程正在运行并且没有启动终止过程 (7543 到 7547 行)。在通过了所有的检查之后，将调用函数 *mini_send*, *mini_receive* 或 *mini_notify* 之一来做真正的工作。如果函数是 *ECHO*，将会使用一个目标进程和源进程相同的 *CopyMess* 宏。如前所述，*ECHO* 函数仅用来测试。

在 *sys_call* 中发生检查错误的可能性很小，但检查很容易实现，因为最终只需要在代码中进行一些小整数的比较。操作系统中这种最基本的检查不可能常发生的错误是可取的。每秒钟这些代码将在它们运行的计算机中执行很多次。

函数 *mini_send*, *mini_rec* 和 *mini_notify* 是 MINIX 3 中标准消息传递机制的核心，需要详细研究。

mini_send (7591 行) 有三个参数：调用进程、目标进程以及指向消息所在缓冲区的指针。进行完 *sys_call* 所做的所有检查以后，仅仅还需要一个检查，既检测发送死锁。7606 到 7610 行的代码检查发送进程和目标进程是否正在同时向对方发送消息。*mini_send* 所进行的关键检查在 7615 到

7616行，检查目标进程是否阻塞在receive上，可以通过进程表项的p_rts_flags域中的RECEIVING位知道。如果目标进程正在等待，则下一个问题是“它在等谁？”。如果它正在等待发送进程或是ANY，则使用CopyMess宏复制消息，接收进程通过复位RECEIVING位取消阻塞。然后调用enqueue给接收进程一个运行机会（7620行）。

另一方面，如果接收进程没有阻塞，或阻塞了但在等待来自另外一个进程的消息，则将执行7623到7632行的代码阻塞发送进程并将发送进程移出运行队列。所有等待向同一进程发送消息的进程用链表链接在一起，目标中的p_callerq域指向位于链表头的进程的进程表项。图2.42(a)中的例子展示了当进程3不能向进程0发送消息时的情形。如果接下来进程4也不能向进程0发送消息，将有图2.42(b)中的情形。

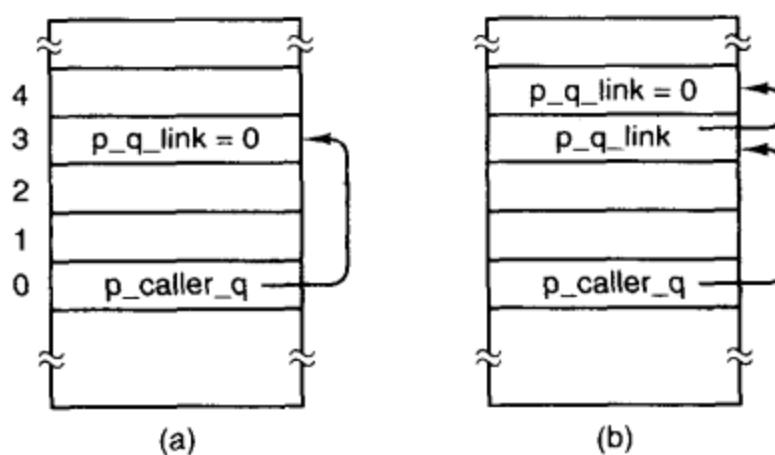


图2.42 试图向进程0发送消息的进程被排队

当函数sys_call的参数是RECEIVE或BOTH时，将调用mini_receive（7642行）。如前所述，通知比一般的消息具有更高的优先级。然而，通知绝非是send调用的正确回复，需要通过检查位图的SENDREC_BUSY标志是否置位来判断是否有挂起的通知。如果发现挂起的通知，将标记为不再挂起并将通知交付处理（7670到7685行）。通知交付用到proc.c顶端定义的两个宏BuildMess和CopyMess。

有人可能会想，既然时间戳是通知消息的一部分，那么它应该传达有用的信息，例如，如果接收者在一段时间内不能接收，那么时间戳应该反映出它处于未交付处理状态有多长时间了。但是通知消息在通知交付处理时产生并加上时间戳，而不是在它发送时。然而，在交付处理时构造通知消息是有原因的。当不能马上处理时，代码不需要保存通知消息。所需要的全部就是记住当交付时应该产生通知。我们不可能得到比这更经济的存储了：每个挂起的通知一个位。

通常情况是，需要的是当前的时间。例如，使用通知向进程管理器发送一个SYN_ALARM消息，如果不是在通知交付时产生时间戳，那么进程管理器将需要在检查它的定时器队列之前，询问内核当前时间。

注意一次只能发送一条消息，mini_send在发送通知之后在7684行返回。但是调用mini_send的进程并不阻塞，所以它可以在通知之后马上调用receive。如果没有通知的话，将会检查队列察看是否有其他类型挂起（7690到7699行）的消息。如果发现了这样一条消息，则调用CopyMess宏处理消息，然后通过调用enqueue（7694行）将消息的发起者置为非阻塞态。在这种情况下，receive调用者不会阻塞。

如果没有通知或其他类型的消息，则调用者通过调用dequeue（7708行）函数阻塞。

mini_notify（7719行）用来完成一个通知。它和mini_send类似，所以可以简要介绍。如果消息的接收者阻塞，并等待接收消息，那么将通过BuildMess构造消息然后交付处理。接收者的RECEIVING标志被关闭，然后调用enqueue（7738到7743行）。如果接收者没有等待消息，那么它的

s_notify_pending 位图中的一个位将被置位，它指示了有一个通知挂起并表明了发送者的身份。然后接收者可以继续它的任务。如果在上一个通知处理之前需要向同一个接收者发送另外一个通知，那么接收者位图中的一位将有效地重写，从同一个发送者的多条通知合并为一个通知消息。这种设计在提供异步消息传递的同时消除了缓存管理的需要。

如果 *mini_notify* 调用由软中断及随后对 *sys_call* 的调用引起，这时中断将会关闭。但是时钟任务或系统任务，或在将来可能会加入 MINIX 3 的其他任务，可能需要在中断未关闭的情况下发送通知。*lock_notify* (7758 行) 是对 *mini_notify* 的一个安全包裹。它首先检查 *k_reenter* 变量来察看中断是否已经关闭，如果是这样，它将直接调用 *mini_notify*。如果处于打开状态，那么首先通过 *lock* 调用关闭中断，然后调用 *mini_notify*，最后通过 *unlock* 调用打开中断。

2.6.10 MINIX 的进程调度

MINIX 3 使用一种多级调度算法。进程被赋予一个与图 2.29 所示结构相关的初始优先级，但是有更多的层次并且进程的优先级在执行过程中可以改变。在图 2.29 中，第 1 层的时钟和系统任务获得最高的优先级。第 2 层的设备驱动程序获得低一些的优先级，它们的优先级不完全相同。第 3 层的服务器进程的优先级比驱动程序低一些，但其中一些优先级比另一些要低。用户进程启动时优先级比所有系统进程优先级都低，并且初始值相等，通过 *nice* 命令可以提高或降低某个进程的优先级。

调度器维护 16 个可运行进程队列，但是任一时刻并不一定所有的队列都在使用。图 2.43 展示了当内核完成初始化并开始运行时，即在 *main.c* 文件中在 7252 行调用 *restart* 函数时，优先级队列和进程的情况。*rdy_head* 数组中的每一项对应一个队列，这一项指向相应队列的头的进程。同样，*rdy_tail* 数组中的每一项对应队列尾的进程。这两个函数都使用 *EXTERN* 宏在 *proc.h* (5595 到 5596 行) 中定义。在系统初始化过程中，初始进程的排队情况由 *table.c* (6095 到 6109 行) 中的 *image* 表决定。

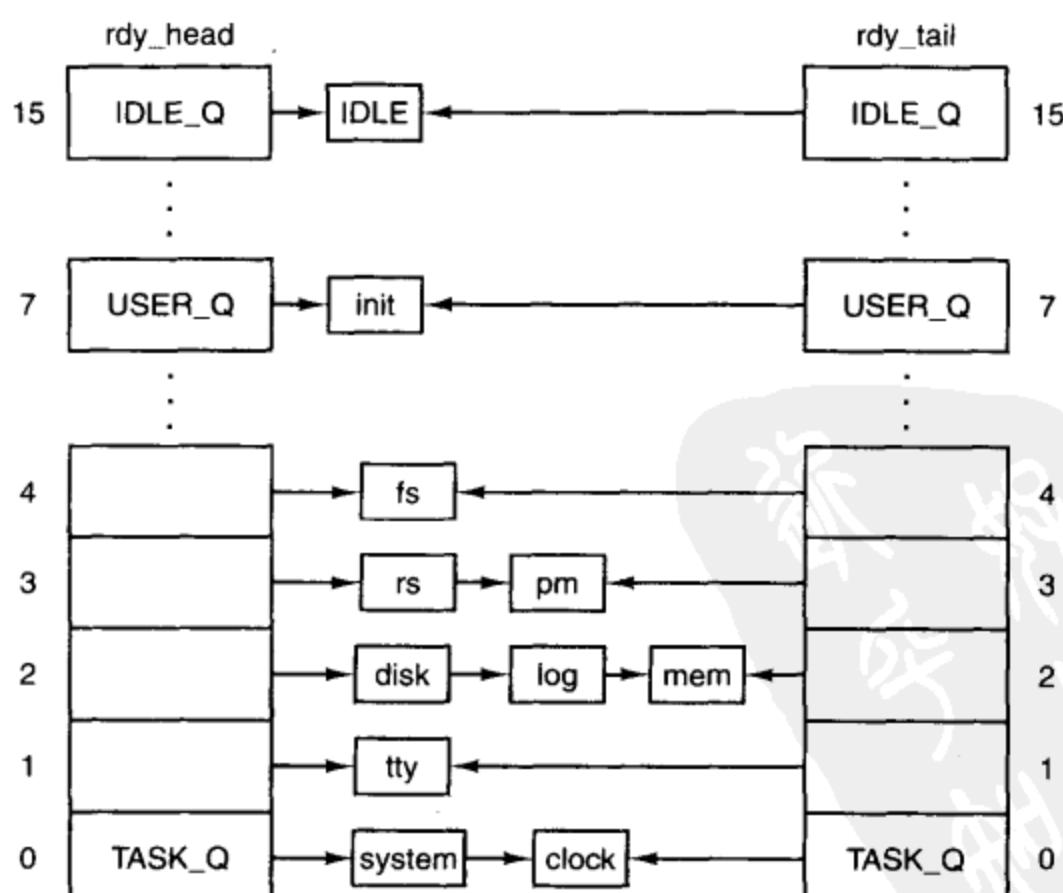


图 2.43 调度器维护 16 个队列，每个队列具有一个优先级，图中示出了 MINIX 3 启动后的初始进程队列情况

每个队列内部采用时间片轮转调度算法。如果一个运行的进程用完了它的时间片，则它被移到队列尾部并分配一个新的时间片。然而当一个阻塞的进程被唤醒时，如果在阻塞前有没有用完的时间片，则它将被放到队首。它并不会得到一个新的时间片，而只是得到阻塞前所剩余的时间片。数组 *rdy_tail* 的存在使得向队列尾部加入一个进程变得简单。当一个运行的进程被阻塞或者被一个信号杀死时，进程将被移出调度队列。队列中仅有可运行进程。

有了上面描述的队列结构以后，调度算法就非常简单：找到最高非空优先级队列，选取队列首部的进程。进程 *IDLE* 总是处于就绪态，并且位于最低优先级队列中。如果所有的高优先级队列都空，则 *IDLE* 进程将会运行。

在最后一部分可以看到许多对 *enqueue* 和 *dequeue* 的引用。现在来分析一下它们。*enqueue* 使用一个指向进程表项的指针（7787行）作为参数调用。它使用一个指向变量的指针作为参数调用另外一个函数 *sched*，该参数说明了这个进程应该位于哪个队列，是放在队列首还是队列尾。现在有了三种可能。这些是经典的数据结构的例子。如果选择的队列为空，则数组 *rdy_head* 和 *rdy_tail* 中各添加一个指针指向该进程，链接域 *p_nextready* 获取一个表示后面没有其他进程的特殊指针值：*NIL_PROC*。如果进程需要加入队列首，则它的 *p_nextready* 获得 *rdy_head* 的当前值，然后 *rdy_head* 指向新进程。如果进程需要加入队列尾，则当前队列尾的进程的 *p_nextready* 指针指向新进程，新进程作为队列尾，它的 *p_nextready* 指针指向 *NIL_PROC*。最后，调用 *pick_proc* 来选择下一个将要运行的进程。

当需要把一个进程转换为非就绪态时，调用 *dequeue*（7823行）函数。一个要阻塞的进程必定处于运行态，所以将要被移出的进程位于它所在队列的首端。然而，可能存在向一个不处于运行态的进程发送了的信号，所以需要遍历队列来查找这个牺牲品，很可能在队列首找到它。当找到以后，需要调整所有的指针来把它移出队列。如果它正在运行，则会调用 *pick_proc*。

该函数还有另外有趣的一点。因为内核中运行的任务共享一个硬件定义的堆栈区域，因此需要偶尔检查这个堆栈的完整性。在 *dequeue* 函数的开头做了一个检查，察看将要移出队列的进程是否运行在内核空间。如果是的话，将检查堆栈区域尾端的特色模式（distinctive pattern）没有被覆盖（7835到7838行）。

下面来看一下 *sched* 函数，这个函数决定一个新就绪的进程应该放到哪个队列上，是放在这个队列的首端还是尾端。每个进程的进程表项内记录有它的时间片、剩余时间片、优先级和所允许的最大优先级。7880到7885行检查进程是否用完了它的整个时间片。如果没有，它将以上次剩余的时间片重新启动。如果时间片已经用完，则检查这个进程是否可以在其他进程没有运行的情况下，连续运行两次。这种情况被视为可能的无限循环或至少是一个足够长的循环，并处以+1的处罚值。然而，如果整个时间片用完，并且其他进程获得了运行机会，那么处罚值为-1。当然，如果两个或多个进程一起在循环中执行，那么这样是没有用的。如何检测这样的情况是一个开放的问题。

下面决定选用的队列。队列0是最优优先级队列，队列15是最低优先级队列。可能有人会说情况应该正好相反，但是这种方式是与UNIX中使用的传统的nice值是一致的，在这里一个正的nice值说明进程需要降低优先级。内核进程（时钟和系统任务）是不受影响的，但是所有其他的进程优先级都可以降低，即通过加上一个正的处罚值来移动到一个更大编号的队列上。所有的进程以它的最高优先级启动，所以在被处以正的处罚值之前，负的处罚值不起作用。优先级还有一个最低限制，即普通进程不会和 *IDLE* 进程处于同一队列。

下面看一下 *pick_proc* 函数（7910行）。这个函数的主要任务是设置 *next_ptr* 的值。队列中所有可能影响选择下一个运行进程的改变都将引起 *pick_proc* 重新调用。在当前进程阻塞时，将调用 *pick_proc* 来重新调度CPU。从本质上讲，*pick_proc* 是调度器。

pick_proc 函数很简单。检查每一个队列。首先检查 *TASK_Q*, 如果这个队列中有进程就绪, 那么 *pick_proc* 设置 *proc_ptr* 指针并立即返回。否则, 检查下一个优先级队列, 一直到 *IDLE_Q* 为止。通过改变指针 *bill_ptr* 的值, 来对赋予进程的 CPU 时间收费 (7694 行)。这会保证系统根据最后运行的用户进程所做的工作对它收费。

proc.c 中的其他过程是 *lock_send*, *lock_enqueue* 和 *lock_dequeue*。它们都使用 *lock* 和 *unlock* 来调用它们的基本函数, 其方式与 *lock_notify* 相同。

总之, 调度算法维护多个优先级队列。最高优先级上的进程总是下一个运行。时钟任务监视所有进程所用的时间。如果一个用户进程用完了它的时间片, 则被放到所在队列的末尾, 在竞争的用户进程间完成了一个简单的时间片轮转调度。任务、驱动程序和服务器预期一直运行到阻塞, 并享有较大的时间片。但是如果它们运行了太长的时间, 则它们也可能被抢占。这种情况不应该频繁发生, 但这是一种防止出现问题的高优先级进程锁上系统的机制。一个妨碍其他进程运行的进程也可以暂时移动到一个低优先级队列中。

2.6.11 与硬件相关的内核支持

有几个函数用 C 语言编写, 但与特定硬件相关。为了便于将 MINIX 3 移植到其他平台, 这些函数被隔离出来放在本节讨论的 *exception.c*, *i8259.c* 和 *protect.c* 中, 而不是和受它们支持的高层代码放在相同的文件中。

exception.c 包含异常处理程序 *exception* (8012 行)。该例程被 *mpx386.s* 中的异常处理代码的汇编语言部分调用 (作为 *_exception* 调用)。由用户进程引起的异常被转换成信号。用户自己编写的程序是可能有错的, 但由操作系统本身引起的异常则表明发生了严重错误, 并产生了不可恢复的故障。数组 *ex_data* (8022 到 8040 行) 确定发生这类错误时应打印的错误信息, 或是对每种异常应向用户进程发送的信号。早期的 Intel 处理器并不能产生所有这些异常, 该数组每项的第 3 个域指出能产生各个异常的最低处理器型号。这个数组提供并已实现了 MINIX 3 的 Intel 系列处理器进展的一个有趣的总结。如果严重错误是由所使用的处理器不支持某种中断而引起的, 则 8065 行将打出一条更换 CPU 的建议信息。

硬件相关的中断支持

i8259.c 中的三个函数用于系统初始化过程, 以对 Intel 8259 中断控制器芯片进行初始化。8119 行的宏是一个哑函数 (只有当 MINIX 3 为 16 位 Intel 平台编译时才需要真函数)。*intr_init* (8124 行) 完成中断控制器的初始化。有两步保证了在系统初始化完成之前不会发生中断。首先在 8134 行调用 *intr_disable*。这是一个对库中汇编语言函数的 C 语言调用, 该读函数执行了一条语句: *cli*, 它关闭了 CPU 对中断的响应。然后向每个中断控制器的寄存器中写入一个字节序列, 其作用是阻止控制器对外部输入的响应能力。在 8145 行中写入的字节全部为 1, 但是控制由从控制器向主控制器的级联输入的一位为 0 (如图 2.39 所示)。字节中的 0 位打开输入, 1 位关闭输入。8151 行中向第二个寄存器写入的字节全为 1。

i8259 中断控制芯片中存储有一个表, 它对每一种可能的输入 (图 2.39 中的右手边信号) 产生一个 8 位索引, CPU 用这个索引来找到正确的中断门描述符。这个表格在计算机启动的过程中由 BIOS 初始化, 这些值几乎都可以留下。因为需要中断的驱动程序在启动的过程中, 可以根据需要修改。每个驱动程序可以通过请求中断控制器中的一位复位来打开自己的中断输入。*intr_init* 的参数 *mine* 用来判断 MINIX 3 是正在启动还是正在打开。这个函数既可以用于启动初始化, 也可用于 MINIX 3 关闭时恢复 BIOS 设置。

硬件初始化完成之后，*intr_init*的最后一步是把 BIOS 的中断向量复制到 MINIX 3 向量表中。*8259.c* 中第二个函数是 *put_irq_handler*（8162 行）。在初始化期间，每个必须响应中断的进程调用该函数，以便将处理程序地址装入中断表 *irq_handlers* 中，这个中断表在 *glo.h* 中使用 *EXTERN* 宏定义。在现代计算机中，15 条中断线经常不够用（因为有可能有 15 个以上的 I/O 设备），所以有可能两个设备共享一条中断线。本书中所描述的 MINIX 3 所支持的基本设备不会发生这种情况，但是需要支持网络接口、声卡或更多复杂的 I/O 设备时，它们可能需要共享中断线。为了支持共享，中断表就不仅仅是一个地址表。*irq_handlers[NR_IRQ_VECTORS]* 是一个指向 *irq_hook* 结构的指针数组，这一结构在 *kernel/type.h* 中定义。这个结构中包含一个指向相同结构的指针，所以可以建立一个链表，并以 *irq_handlers* 中的一个元素作为链表头。*put_irq_handler* 向这样的一个链表中加入一个入口。这样的一个入口中最重要的元素是指向中断处理程序的指针，该函数在中断产生（例如 I/O 请求完成）时开始执行。

需要说一下 *put_irq_handler* 的一些细节。注意在扫描整个链表的 *while* 循环（8176 到 8180 行）开始前，变量 *id* 设置为 1。每次循环 *id* 都左移一位。8181 行的检查把链表的长度限制在 *id* 的长度之内，对于 32 位系统来说链表中可以有 32 个处理函数。在正常情况下，扫描将以找到链表尾结束，在这里可以加入新的处理函数。当完成这些以后，*id* 同时也被存入链表中新元素的同名域中。*put_irq_handler* 同时设置全局变量 *irq_use* 中的一位，以记录对应 IRQ 中存在处理函数。

如果读者全面了解 MINIX 3 把驱动程序放入用户空间的设计目标，那么前面关于中断处理函数如何调用的讨论可能会使读者感到困惑。在钩子结构中所存储的中断处理函数的地址，只有在指向内核地址空间的函数时才是有效的。内核地址空间中唯一的中断驱动的设备是时钟。那些拥有自己的地址空间的设备驱动程序怎么办呢？

答案就是，系统任务处理它。实际上，这是大多数涉及内核与用户空间进程通信问题的答案。用户空间由中断驱动的设备驱动程序在需要注册中断处理函数时，向系统任务发出一个 *sys_irqctl* 调用。然后系统任务调用 *put_irq_handler*，然而存入中断处理函数域的不是驱动程序地址空间的中断处理函数地址，而是作为系统任务一部分的 *generic_handler* 的地址。*generic_handler* 使用钩子结构中的进程号域来定位驱动程序的 *priv* 表项。驱动程序的挂起中断位图中与该中断相应的位设置为 1，然后 *generic_handler* 向驱动程序发送一个通知。通知标记为来自 *HARDWARE*，消息中为驱动程序包含了挂起中断位图。因此，如果一个驱动程序需要对来自不止一个源的中断做出响应，那么它可以判断当前通知对应的中断是哪一个。实际上，因为位图随通知一起发送，所以通知为驱动程序提供了所有挂起中断的信息。钩子结构中的另一个域是策略域，它决定了中断是马上打开还是继续保持关闭。在后一种情况下，在当前中断服务完成以后，应该由驱动程序发出一个 *sys_irqenable* 内核调用。

MINIX 3 的设计目标之一是支持运行时 I/O 设备配置。下一个函数 *rm_irq_handler* 移去一个处理函数，如果设备驱动程序被移去（可能由另外一个代替），那么这一步是必要的。它的作用与 *put_irq_handler* 的作用相反。

这个文件中的最后一个函数 *intr_handler*（8221 行）从 *mpx386.s* 中的 *hwint_master* 和 *hwint_slave* 宏调用。位图数组 *irq_actids* 中的每个元素与正在服务的中断对应，用来跟踪链表中每个处理函数的当前状态。对于链表中的每一个函数，*intr_handle* 置位 *irq_actids* 中的对应位，并调用处理函数。如果处理函数没有任务可做，或马上完成任务返回，则它返回“true”并将 *irq_actids* 中的对应位清零。在 *hwint_master* 和 *hwint_slave* 宏的末尾，将中断的整个位图作为一个整数检查，并据此决定在另外一个进程启动前是否允许打开中断。

Intel 保护模式支持

protect.c 包含与 Intel 处理器保护模式相关的例程。全局描述符表（Global Descriptor Table, GDT）、局部描述符表（Local Descriptor Table, LDT）和中断描述符表（Interrupt Descriptor Table, IDT）都位于内存中，它们提供对系统资源的受保护的访问。**GDT** 和 **IDT** 被 CPU 中特殊的寄存器所指向，GDT 表项指向 LDT。GDT 对所有进程均可用并记录了操作系统使用的内存区域的段描述符。通常对每个进程有一个 LDT，它记录了该进程所使用内存区域的段描述符。描述符是一个包含许多内容的 8 字节结构，但段描述符中最重要的是描述一个内存区域的基地址和限长域。IDT 也由 8 字节的描述符构成，其中最重要的部分是当中断被激活时所执行代码的地址。

prot_init (8368 行) 由 *start.c* 中的 *cstart* 调用，以建立 GDT (8421 到 8438 行)。IBM PC 机的 BIOS 要求它按特定的方式排序，所有对它的索引都定义在 *protect.h* 中。每个进程 LDT 的空间都在进程表中分配。每个包含有两个描述符，分别是代码段和数据段描述符（记住这里讨论的段由硬件定义）。这些段与操作系统中的段不同，操作系统中将硬件定义的数据段进一步分为数据段和栈段。从 8444 到 8450 行，每个 LDT 的描述符放在 GDT 中。这些描述符实际上由函数 *init_dataseg* 和 *init_codeseg* 建立。当进程内存镜像改变时，LDT 中的表项被初始化（例如，执行 *exec* 系统调用时）。

另一个需要进行初始化的处理器数据结构是任务状态段（Task State Segment, TSS）。该数据结构在文件的开头定义 (8325 到 8354 行)，并为处理器寄存器和其他任务切换时应保存的信息提供空间。MINIX 3 只使用某些域的信息，这些域定义了当发生中断时在何处建立新堆栈。在 8460 行对 *init_dataseg* 的调用保证它可以用 GDT 进行定位。

为了理解在最底层上 MINIX 3 是如何工作的，可能最重要的是理解异常、硬件中断或 *int <nnn>* 指令如何激活执行那些事先写好的服务代码。这些事件通过中断门描述符表处理，数组 *gate_table* (8383 到 8418 行) 由编译器用异常和硬件中断处理例程的地址来初始化，随后它被 8464 到 8468 行的循环用来对该表进行初始化，具体通过调用 *int_gate* 函数完成。

有充分的理由解释为什么按照描述符方式来组织数据，其原因源于硬件细节，以及维持高级处理器与 16 位的 286 处理器之间的兼容性这两个方面。幸运的是，通常可以将这些细节留给 Intel 处理器的设计者。对其中大部分而言，C 语言允许回避具体的细节，但在实现一个真正的操作系统时必须面对某些方面的细节。图 2.44 展示了一种段描述符的内部结构。在 C 程序中基地址可被引用为简单的 32 位无符号整数。注意，这些基地址被分成三部分，其中两部分被许多 1 位、2 位或 4 位的二进制组合隔开。20 位的限长被分成 16 位和 4 位两部分。限长可解释为字节数，或者是以 4096 字节为一页的页面数，这具体依赖于 *G*（粒度）位。其他描述符，例如那些用来指定中断处理方式的描述符，具有不同的结构，但其结构也同样复杂。第 4 章将更加详细地讨论这些描述符。

protect.c 中定义的大部分其他函数，用于在 C 程序中使用的变量和类似于图 2.44 所示的供机器使用的复杂格式的描述符数据之间进行转换。*init_codeseg* (8477 行) 和 *init_dataseg* (8493 行) 在操作上很类似，都用于将传送给它们的参数转换成段描述符。而它们则调用下一个函数 *sdesc* (8508 行) 来完成此项工作。*sdesc* 函数的工作就是对图 2.44 所示的繁琐的结构进行具体处理。在系统初始化期间并不使用 *init_codeseg* 和 *init_data_seg*。此外，当启动新进程时，为了给进程分配适当的存储器段使用，系统任务将调用它们。*seg2phys* (8533 行) 只被 *start.c* 调用，它执行 *sdesc* 的逆操作，即从一个段描述符中析取出段基地址。*phys2seg* (8556 行) 已经不再需要了，内核调用 *sys_sectl* 现在用来访问远程内存段，例如 PC 在 640 KB 到 1 MB 空间所保留的区域。*int_gate* (8571 行) 在为中断描述符表构造表项时，执行一个类似于 *init_codeseg* 和 *init_dataseg* 的函数。

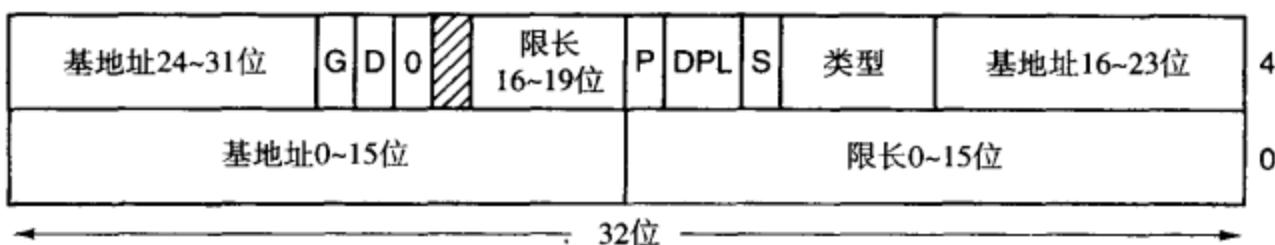


图 2.44 Intel 处理器的段描述符格式

现在来看一下 *protect.c* 的函数 *enable_iop* (8589 行)，它能进行一个不大规范的操作。它改变 I/O 操作的特权级，从而允许当前进程执行 I/O 读写的指令。对于该函数目的的描述远比函数本身复杂，该函数仅仅置位调用进程栈帧表项中的一个字的两位，当进程下次执行时，这个字将加载到 CPU 状态寄存器。不需要撤销此操作的函数，因为它只作用于调用进程。这一函数当前没有使用，也没有提供给用户空间函数激活它的方法。

protect.c 中的最后一个函数是 *alloc_segments* (8603 行)。该函数被 *do_newmap* 调用，在初始化过程中还被内核中的 *main* 例程调用。这一定义是非常依赖于硬件的。它获得记录在进程表项中的段的分配，并操作奔腾处理器用来在硬件级支持段保护的那些寄存器和描述符。如 8629 到 8633 行的那些多次赋值是 C 语言的特性。

2.6.12 实用程序和内核库

最后，内核有一个支持函数库，可以通过编译用汇编语言写的 *klib.s* 和几个位于文件 *misc.c* 中的 C 实用程序包含进来。先看汇编语言文件。*klib.s* (8700 行) 是一个与 *mpx.s* 类似的短文件，*mpx.s* 的功能是根据 *WORD_SIZE* 的定义来选择适当的机器版本。这里将讨论的代码在 *klib386.s* (8800 行) 中。其中有大约 20 几个实用例程，之所以用汇编语言是出于效率的原因，或是因为它们根本就无法用 C 来编写。

_monitor (8844 行) 使系统能够返回到引导监控程序。从引导监控程序的角度来看，整个 MINIX 3 是一个子例程，并且当 MINIX 3 启动时，返回引导监控程序的地址被留在其堆栈中。*_monitor* 只需恢复各段选择符和 MINIX 3 启动时保存下来的栈指针，然后像从其他子例程那样返回即可。

Int86 (8864 行) 支持 BIOS 调用。BIOS 用来提供可选择的磁盘驱动程序，这里不做讨论。*Int86* 把控制权交给引导监控程序，引导监控程序进行一个从保护模式到实模式的转移以执行 BIOS 调用，然后回到保护模式，以便返回 32 位的 MINIX 3。引导监控程序还返回在 BIOS 调用过程中所经过的时钟滴嗒数。讨论时钟任务时将会看到这一数值如何使用。

尽管可以使用 *_phys_copy* (如下所示) 来进行消息的复制，然而有一个更快的专门过程 *_cp_mess* (8952 行) 被用于此目的，它的调用方式为

```
cp_mess(source, src_clicks, src_offset, dest_clicks, dest_offset);
```

其中，*source* 为发送者的进程号，它被复制到接收者缓冲区的 *m_source* 域。源地址和目标地址通过一个 click 数来指定，典型情况下是包含该缓冲区的段地址以及一个相对于该 click 的偏移量。这种指定源和目标地址的方式比 *_phys_copy* 所用的 32 位地址更加高效。

定义 *_exit*, *_exit* 和 *_exit* (9006 到 9008 行) 是因为编译 MINIX 3 时可能用到的一些库例程需调用标准 C 函数 *exit*。从内核退出是一个没有意义的概念，从内核退出后便无处可去。因此，这里不能使用标准的 *exit*。这里所用的解决办法是开中断，然后进入一个无休止的循环。最终一个

I/O 操作或时钟将发出中断，由此恢复正常系统的操作。入口点 `__main` (9012 行) 是处理编译操作的另一种方法。该方法在编译一个用户程序时很合理，但在内核中则没什么用处。它指向一条汇编语言的 `ret` (从子例程返回) 指令。

`_phys_insw` (9022 行), `_phys_insb` (9047 行), `_phys_outsw` (9072 行) 和 `_phys_outsb` (9098 行) 提供了对 I/O 端口的访问方法。在 Intel 硬件中，I/O 端口使用与内存分开的地址空间，并使用与内存访问不同的指令。这里所使用的 I/O 指令 `ins`, `insb`, `outs` 和 `outsb` 被设计用来有效地操作数组 (字符串)，既可以是 16 位的字，也可以是 8 位字节。每个函数中的其他指令还设立了一些参数，这些参数当在物理地址指定的缓冲区和端口之间移动一定数量的字节或字时使用。这一方法为服务磁盘提供了足够的速度，服务磁盘必须比简单的一次一个字或一次一个字节的 I/O 操作要快。

一个简单的机器指令就可以打开或关闭 CPU 对所有中断的响应能力。`_enable_irq` (9126 行) 和 `_disable_irq` (9162 行) 的代码则更为复杂。它们工作在中断控制器芯片级，以启用和禁止单个硬件中断。

`_phys_copy` (9204 行) 在 C 程序中使用如下语句进行调用：

```
phys_copy(source_address, destination_address, bytes);
```

它将物理内存中任意处的一个数据块复制到任意的另外一处。其中，两个地址都是绝对地址，也就是地址 0 确实表示整个地址空间的第一个字节，并且三个参数均为无符号长整数。

为安全起见，所有将被程序使用的地址空间应首先擦除干净，以抹除上个程序占用空间时留下的数据。这一操作通过调用 MINIX 3 的 `exec` 调用，然后调用 `klib386.s` 中的函数 `phys_memset` (9248 行) 来实现。

接下来的两个小函数专用于 Intel 处理器。`_mem_rdw` (9291 行) 返回内存中任意处的一个 16 位的字。其结果用 0 扩展后放在 32 位长的 `eax` 寄存器中。`_reset` 函数 (9307 行) 使处理器复位，这通过将一个空指针装入处理器的中断描述符表寄存器，然后执行一个软件中断来实现，它与硬件复位效果一样。

`idle_task` (9318 行) 在不需要做其他工作时调用。它写成了一个无限循环的形式，但是并不仅仅是一个忙循环 (虽然可以用这种方式来实现同样的效果)。`idle_task` 利用了 `hlt` 指令，`hlt` 指令把处理器置于节电状态，直到收到中断。然而，`hlt` 是一条特权指令，当特权级不是 0 时执行它将会引起异常。因此 `idle_task` 把一个包含 `hlt` 指令的子程序压栈，然后调用 `level0` 函数 (9322 行)。这一函数取回 `halt` 子程序的地址，并把它复制到一个保留的存储区域内 (在 `glo.h` 中声明，但实际在 `table.c` 中保留)。

`_level0` 把任何事先存入该区域的地址都视为一个中断服务例程的函数部分，并在最高特权级 (0 级) 运行。

最后两个函数是 `read_tsc` 和 `read_flags`。`read_tsc` 通过执行一条读时间戳计数器指令 `rdtsc` 来读取一个 CPU 寄存器的值。时间戳计数器对 CPU 周期计数，并用来做时间基准或调试。MINIX 3 汇编语言并不支持这条指令，而是用 16 进制操作码编码的方式产生。最后，`read_flags` 读取处理器的标志，并把它们以 C 变量的形式返回。编程人员累了，把这个函数的目的注释写错了。

本节讨论的最后一个文件是 `utility.c`，它提供了三个重要函数。当内核中发生了非常严重的错误时，将调用 `panic` (9429 行)。它打印出一条信息并调用 `prepare_shutdown`。当内核需要打印消息时，它不能使用标准库函数 `printf`，这里定义了一个特殊的 `kprintf` (9450 行)。这里并不需要库函数版本中支持的所有格式选项，但是这里提供了大部分的功能。因为内核不能使用文件系统来访问文件或设备，它把所有字符传递给了另外一个函数 `kputc` (9525 行)。这一函数把每个字符都加入一

个缓冲区。后面，当 *kputc* 收到 *END_OF_KMESS* 时，它将通知处理这些消息的进程。进程在 *include/minix/config.h* 中定义，既可以是日志驱动，也可以是控制台驱动。如果是日志驱动，那么消息也会同时传递给控制台。

2.7 MINIX 3 的系统任务

制作独立于主系统部件的内核外部进程的结果是，它们被禁止执行实际的 I/O 操作，也不能改动系统表及完成其他操作系统一般都有的功能。例如，*fork* 系统调用是由进程管理器来处理的。当一个新的进程被创建后，内核必须能感知它以便调度。那么一个进程管理器是如何通知内核的呢？

解决这个问题的方法是让内核为驱动程序和服务器提供一组服务。这些对普通用户进程可用的服务，允许驱动程序和服务器执行实际 I/O 操作、存取内核表以及其他它们所需要的功能，而不需要在内核内部。

这些专门的服务是由 **系统任务** 进行处理的，见图 2.29 中的层 1。尽管它被编译进了内核的二进制程序中，但它实际上是一个独立的进程并这样被调度：系统任务接收所有来自驱动程序和服务器对于这些服务的调用请求，因为系统任务是处于内核空间的一部分，所以它可以调度并使用它们。

前述章节中讲述了一个系统任务提供的服务的例子。而在对中断处理的讨论中则描述了一个用户空间里的设备驱动程序是如何使用 *sys_irqctl* 发送一个消息通知系统任务来请求安装一个中断处理程序的。用户空间的驱动程序无法访问保存有中断服务器的内核数据结构，而系统任务则可以。进一步讲，因为中断服务器必须位于内核地址空间，其地址正是由系统任务提供的函数 *generic_handler* 的地址。这个函数通过向设备驱动程序发送确认信息来对中断予以响应。

这是一个澄清术语的好地方。在传统的使用单体内核的操作系统中，**系统调用**（*system call*）指所有对于内核提供服务的调用。而在现代的类 UNIX 操作系统中，POSIX 标准描述对进程可用的系统调用。也许有一些对于 POSIX 的非标准的扩展，当然程序员利用系统调用一般将会引用那些在 C 库中定义的函数，而这可以带来程序接口的容易使用。也存在有些对于程序员看起来不同于“系统调用”的独立的库函数实际上使用的仍是同样的内核入口。

在 MINIX 3 中，情况就不同了；操作系统的组件是在用户空间中运行的，尽管它们有着作为系统进程的特权。尽管对于图 1.9 中所列的由 POSIX 定义的系统调用来说，依然使用“系统调用”来指代，但是用户进程不再直接向内核请求服务。在 MINIX 3 中用户进程发出的系统调用将被转换为发往服务器进程的消息。服务器进程相互之间、服务器进程与设备驱动程序之间以及服务器进程与内核之间通过消息进行通信。作为此阶段的主体的系统任务将接收所有对于内核服务的请求，笼统而言，可以称这些请求为系统调用，但是严格地说应该称之为**内核调用**。用户进程不能产生内核调用。很多情况下，用户进程首创的系统调用将会引起服务器进程生成一个同该系统调用类似名字的内核调用。这往往是因为被请求的服务中有些部分只能由内核来完成。例如用户进程产生的 *fork* 系统调用，它将被发送到进程管理器，但它只能完成其中的一部分工作。*fork* 系统调用要求改变进程表的内核部分，为此进程管理器将发送 *sys_fork* 调用给系统任务，而它则可以操作内核空间的数据。并非所有的内核调用都仅跟一个系统调用有如此清晰的对应关系。例如，来自设备驱动程序的用来读写 I/O 端口的 *sys_devio* 内核调用。图 1.9 中半数以上的系统调用会激活一个设备驱动程序从而产生一个或多个 *sys_devio* 调用。

从技术上讲，还应该划分出第三类调用（除去系统调用和内核调用）。用于进程间通信的诸如 *send*, *receive*, *notify* 等消息原语可以被认为是类系统调用。在本书中的剩余章节将会多次这样称呼

它们，它们确实调用了系统。但是有时它们又应该被称为既非“系统调用”也非“内核调用”的其他名字，即也可能使用其他名字。有时会用IPC原语以及陷阱，所有这些都可能在一些源代码的内容中看到。可以把信息原语想象为通信系统中的载波。为使无线电波更加有用，调制是必要的；而消息类型和消息结构中的其他组件则使得消息调用可以传输信息。而有一些情况下，未调制的无线电波则是有用的；例如用无线电指向标来引领飞机进入机场。Notify消息原语与此类似，比起它的原始内容，它传送很少的信息。

2.7.1 系统任务综述

系统任务接收28种消息，如图2.45所示。其中每一种可以被视为一个内核调用，尽管如此，还是应该明白，有些情况下多个宏被定义为不同的名字，而事实上都只是表中的同一条消息。而有时表中的多种消息其实都是被同一个过程进行处理的。

消息类型	来自	含义
sys_fork	PM	已派生一个进程
sys_exec	PM	在EXEC调用后设置栈指针
sys_exit	PM	一个进程已退出
sys_nice	PM	设置调度优先级
sys_privctl	RS	设置或改变权限
sys_trace	PM	执行PTRACE调用的一个操作
sys_kill	PM, FS, TTY	在KILL调用后发送信号到一个进程
sys_getksig	PM	PM正检测挂起信号
sys_endksig	PM	PM已完成处理信号
sys_sigsend	PM	发送一个信号到一个进程
sys_sigreturn	PM	一个信号完成后清理
sys_irqctl	Drivers	启用、禁用或配置中断
sys_devio	Drivers	读写I/O端口
sys_sdevio	Drivers	从I/O端口读字符串或将字符串写到I/O端口
sys_vdevio	Drivers	执行I/O请求的一个向量
sys_int86	Drivers	执行一个实模式BIOS调用
sys_newmap	PM	设置一个进程内存映射
sys_segctl	Drivers	添加段和获取选择器（远离数据访问）
sys_memset	PM	写字符到内存区
sys_umap	Drivers	将虚拟地址转换为物理地址
sys_vircopy	FS, Drivers	使用纯虚拟寻址复制
sys_physcopy	Drivers	使用物理寻址复制
sys_virvcopy	Any	VCOPY请求的向量
sys_physvcopy	Any	PHYSCOPY请求的向量
sys_times	PM	获得正常运行时间和进程时间
sys_setalarm	PM, FS, Drivers	调度一个同步alarm
sys_abort	PM, TTY	崩溃：MINIX不能继续
sys_getinfo	Any	请求系统信息

图2.45 系统任务接收的消息类型。Any表示任何系统进程；用户进程不能直接调用系统任务

系统函数的主程序与其他任务是同结构的。在进行了必要的初始化之后，它进入一个循环：获取一条消息，然后将其分派给适当的服务进程，最后发送一条回复消息。一些通用支持函数可以在主文件 *system.c* 中找到，但是主循环体分派给了 *kernel/system* 文件夹中的一个独立文件的一个进程，以负责处理每一个内核调用。在对系统任务的应用进行探讨时，我们会对这个进程及如此组织的原因进行介绍。

首先对每一个内核调用的功能进行简单介绍。图 2.45 中所列消息可以分为几类。第一种是关于进程管理的。*sys_fork*, *sys_exec*, *sys_exit* 和 *sys_trace* 很明显与标准 POSIX 系统调用有密切联系。尽管 *nice* 不是一个 POSIX 请求的系统调用，但这条命令实际上导致了一个 *sys_nice* 内核调用以改变进程的属性。该类中唯一可能不太相同的是 *sys_privctl*，它被再生服务器（RS）所使用，再生服务器是指负责转换进程使其从普通用户进程变为系统进程而开始运行的 MINIX 3 组件。*sys_privctl* 改变了进程特权，例如，允许某个进程进行内核调用。当驱动程序和不属于启动映像的服务器被 */etc/rc* 脚本启动时，将用到 *sys_privctl*。MINIX 3 驱动程序也可以在任何时刻启动（或重启）；此时将会用到特权转变。

第二类系统内核调用是有关信号的。*sys_kill* 是有关用户可访问（和误命名）系统调用之终止进程。而该类中的其他消息如 *sys_getksig*, *sys_endksig*, *sys_sigsend* 和 *sys_sigreturn* 则被进程管理器用以得到关于操作信号的内核的帮助。

sys_irqctl, *sys_devio*, *sys_sdevio* 和 *sys_vdevio* 内核调用是 MINIX 3 中的特殊调用。它们提供了用户空间驱动程序所必要的支持。在这一部分的开始提到了 *sys_irqctl*。它的功能之一是设置一个硬件中断处理程序并为用户空间驱动程序启用该中断。*sys_devio* 允许一个用户空间的驱动程序要求系统任务从一个 I/O 端口读或者写。这显然是必要的；而且这种方式显然比在内核空间运行的驱动程序开销要大。接下来两个内核调用提供了高层次的 I/O 设备支持。当一系列字节或者字（例如一个字符串）需要从一个单独的 I/O 端口地址读取或者写入时（例如访问一串口时），*sys_sdevio* 将被使用。*sys_vdevio* 用于发送一个 I/O 请求向量给系统任务。此向量指一个含有（端口号，值）对的序列。在本章开头曾介绍了用于初始化 Intel i8259 中断控制器的 *intr_init* 函数。从 8140 到 8152 行，一系列的指令写了一系列的字节值。对于两块 i8259 芯片的每一块，都有一个控制端口用于设置模式，而另一个端口则用于接收初始化序列中的 4 字节序列数据。当然，这些代码是在内核里面执行的，所以不需要系统任务的支持。但是如果这是由用户空间进程来进行的，那么只用一条消息向包含有 10 个（端口号，值）对的缓冲传输此地址要比使用 10 条消息分别传送一个要写入的端口地址和写入值高效得多。

图 2.45 中的第三类系统内容调用则是在不同方面涉及到内存的内核调用。首先是 *sys_newmap*，它在当内存被进程改变时被进程管理器调用，所以该进程表的内核部分可以被更新。*sys_segctl* 和 *sys_memset* 则提供了一个简单的办法来为一个进程去访问它的数据空间的外部的内存。正如在对有关 MINIX 3 系统的启动的讨论中所提到过的，从 0x0000 到 0xffff 的内存区域是为 I/O 端口保留的。有些设备使用此内存区域的一部分进行 I/O 操作，例如，显卡将其芯片上用于存储待显示数据的内存映射到此处。*sys_segctl* 则被设备驱动程序用于获取一个段选择器来定位于此范围的内存。另一个内核调用 *sys_memset* 则用于一个服务器进程需要将数据写入一块并不属于它的内存区。当一个新的进程被启动时，进程管理器使用它来清理外部内存，以免这个新的进程读取另一个进程遗留的数据。

下一类内核调用则是关于内存复制的。*sys_umap* 将虚拟地址转换为物理地址。*sys_vircopy* 和 *sys_physcopy* 使用虚拟或者物理地址进行内存区复制。而其他两个内核调用 *sys_vircopy* 和

`sys_physvcopy`则是前两个的向量版本，利用一个向量化的I/O请求，它们允许对系统任务建立一个请求，以进行一系列的内存复制操作。

`sys_times`显然与时间有关，它对应POSIX定时器系统调用`times`。`sys_setalarm`则与POSIX`alarm`系统调用有关，但这是一种远程联系。POSIX调用大多数情况下被进程管理器处理，而进程管理器则维护着一个代表用户进程的定时器的序列。当需要为一个进程在内核中设置一个定时器时，进程管理器将调用`sys_setalarm`。这只在由进程管理器管理的队列头有变动时才被执行，而没有必要在每一个来自用户进程的`alarm`调用之后进行。

图2.45中所列的最后两个内核调用用于系统控制。`sys_abort`会在一个正常的关闭系统请求或者崩溃（panic）之后由进程管理器生成，也可以由tty设备响应用户按下的Ctrl-Alt-Del组合键而产生。

最后，`sys_getinfo`是一个用于处理各种不同范畴的对于内核信息的请求的分发器。如果读者在MINIX 3 C源文件中查找，则会发现事实上很少有按照它本身的名字对其进行的调用，但是当读者进一步在其头文件中查找它时，将发现在文件`include/minix/syslib.h`中`sys_getinfo`这个名字被赋予了不下13个宏。例如，

```
sys_getkinfo(dst) sys_getinfo(GET_KINFO, dst, 0, 0, 0)
```

它被用于在系统启动期间返回`kinfo`结构体（在`include/minix/type.h`的2875行2893行定义）给进程管理器使用。其他时候也会用到同样的信息，比如用户命令`ps`需要知道进程表的内核部分的位置以及所有进程的状态信息。那么这时候它就会要求进程管理器，而进程管理器又会使用`sys_getkinfo`的变体`sys_getinfo`来获得这个信息。

在结束对内核调用的综述之前，有必要说明一下`sys_getinfo`并非唯一一个由在头文件`include/minix/syslib.h`中定义的不同名字的宏调用的内核调用。像`sys_sdevio`通常是被`sys_insb`、`sys_insw`、`sys_outsb`和`sys_outsw`诸宏之一来调用的。名字如此设置是为了易于分辨该操作是输入还是输出、其数据类型是字节还是字。同样，`sys_irqctl`常被`sys_irqenable`、`sys_irqdisable`等这样的宏来调用。这些宏使得对于人们阅读代码时理解其含义更加容易，还可以帮助编程者自动产生常量参数。

2.7.2 系统任务的实现

系统任务是由`kernel`/主目录中的一个头文件`system.h`和一个C源文件`system.c`编译而成的。另外还有一个专门由`kernel/system`/子目录中的源文件创建的库。这样组织是有原因的，尽管如前所述的MINIX 3是一个具有通用性的操作系统，但它也具有支持便携设备上使用嵌入式系统的潜在能力。在这种情况下，一个可分解的操作系统版本比较合适。例如，对于没有磁盘的设备就不需要文件系统。如文件`kernel/config.h`中内核调用部分的编译选项就可以选择或者放弃。可以在编译期的最后阶段对支持每一个内核调用的代码进行链接，以使得创建一个定制的系统变得更加容易。

将各内核调用的支持放在不同的文件中可使得软件的维护更加简单，但是这样也会造成文件之间的一些冗余，如果将这些文件全部列出来将会使本书增加40页左右的篇幅。因此本书将其列在了附录B中，而在本书正文中只是介绍了`kernel/system`/文件夹中的一部分文件。另外，在本书的附带光盘中以及MINIX 3的网站上可以看到这些文件的全部内容。

首先看一下头文件`kernel/system.h`（9600行）。它提供了一个有关图2.45中绝大多数内核调用的函数原型。并且还有`do_unused`的原型，这个函数在产生了不支持的内核调用时被调用。图2.45中所列的一些消息类型对应此处的宏定义，在9625到9630行处。这些也是一个函数用以处理多个调用的情况。

在学习 *system.c* 文件中的代码之前，首先要注意调用向量 *call_vec* 以及 9745 到 9749 行的宏映射的定义。*call_vec* 是一个函数指针的数组，通过使用消息类型（用数字表示）作为数组索引的方法，来提供分派函数服务特定消息的机制。宏映射是初始化此数组的一种便捷方法。把宏用一个非法参数来扩展的定义方式将导致把数组的大小表示为负数，这样当然会导致编译错误。

系统任务的顶层是 *sys_task* 进程。在初始化一个函数指针的数组的调用完成之后，*sys_task* 进入一个循环以等待消息，然后对消息做一些有效性验证，通过验证的消息将被分派给相应的函数，由这个函数处理该消息对应的调用，并有可能产生回复消息。在 MINIX 3 的运行期间，这样的过程会往复循环（9768 行到 9796 行）。其中的验证过程包括对该调用者核对 *priv* 表的项以确定其是否有权进行此调用以及该调用的类型是否有效。而向相应函数的分派过程可参见 9783 行。*call_vec* 数组内的索引值即调用号，被调用的函数地址存放于数组内的该单元，函数接收的参数则是指向此消息的指针，返回值是状态编号。例如返回 *EDONTREPLY* 状态代表无返回信息，否则返回信息在 9792 行处被返回。

正如在图 2.43 中所显示的，当 MINIX 3 启动时，系统任务处于最高优先级队列的头部，所以系统任务的初始化功能可以用于初始化中断钩子数组和 *alarm* 定时器列表（见 9808 到 9815 行）。前边曾经提到过，任何情况下，系统任务都用于启用代表设备驱动程序的中断，而该驱动程序对中断做出响应，所以用系统任务来准备该表是行得通的。当有其他进程请求同步 *alarm* 时，系统任务将建立定时器，所以在此进行定时器列表初始化也比较合适。

接下来，在 9822 到 9824 行，*call_vec* 数组中所有的项都被程序 *do_unused* 所填充，这样当发生不被支持的内核调用时 *do_unused* 将被调用。而从文件的 9827 到 9867 行，则是映射 (*map*) 宏的扩展，其中每一个都向 *call_vec* 的适当项装载了一个函数地址。

system.c 文件的剩余部分由被声明为 *PUBLIC* 的函数组成，并且这些函数有可能被多个服务于内核调用的程序或者内核的其他部分使用，例如，第一个函数 *get_priv*（9872 行）被支撑系统调用 *sys_privctl* 的函数 *do_privctl* 使用，同时，当构造一个启动映像进程的进程表项时，它又会被内核本身调用。也许这个名字有些让人产生误解。*get_priv* 不对已分配的特权信息进行检索，它只是查找到一个 *priv* 结构并将其分配给调用者。有两种情形——每个系统进程在 *priv* 表中均有它们自己的项。如果其中之一失效，那么进程将不再成为一个系统进程。所有的用户进程都共享表中同样的项 (*entry*)。

get_randomness（9899 行）被用做随机数发生器生成种子数字，随机数发生器是 MINIX 3 中的一个字符设备。最新的奔腾系列处理器包含了一个外部时钟计数器并且提供了一组可以读取它的汇编语言指令。如果可用，则使用它，否则将调用一个读取时钟芯片上的寄存器的函数。

在将要用信号通知的进程的位图 *s_sig_pending* 的某位置位之后，*send_sig* 向该系统进程发送一个通知。置位操作见 9942 行。需要注意的是，因为 *s_sig_pending* 位图是 *priv* 结构的一部分，所以这种机制只能用于通知系统进程。所有的用户进程共享一个公共的 *priv* 表项，因此其类似于 *s_sig_pending* 位图的域不能被用户进程共享使用。在调用 *send_sig* 之前，需要确认目标是系统进程。而调用或者来自 *sys_kill* 调用的结果，或者是当 *kprintf* 发送字符串时来自内核。早期的情况下，调用者确定目标是否是系统进程，而在后来的情况下，内核仅向已配置的输出进程（即控制台驱动程序或者日志驱动程序）输出，而二者都是系统进程。

再下一个函数 *cause_sig*（9949 行）用来向用户进程发送信号。当系统进程 *sys_kill* 以一个用户进程为目标时，该函数将被调用。它之所以位于 *system.c* 中，是因为在由用户进程触发的异常发生时，它也会被内核直接调用以对此异常做出处理。至于 *send_sig*，则必须在接收者的位图中为待处理的信号置位，但是在用户进程中，该置位信息不是在 *priv* 表中，而是在进程表里面。目标进程必须通过对 *lock_dequeue* 进行调用以置为非就绪态，并且更新标记（同样位于进程表中）以表示它将

要被信号通知。接着会有消息发送，但不是发往目标进程，而是送给进程管理器，由它来负责用信号通知一个进程的各个方面，而被通知进程应是可以被一个用户空间系统进程所处理的。

接下来的三个函数均是支持 `sys_umap` 内核调用的。通常情况下，进程处理虚拟地址时需要涉及到特殊段的基址。但有时它们还需要知道某个内存区域的绝对（物理）地址，例如，当一个请求需要进行分属两个不同段的内存区域副本时。指定一个虚拟地址大致有三种方法。一种标准方法是针对文本段、数据段、栈段，它们被分配给进程并记入进程表项中。在此过程中，由虚拟内存向物理内存的转换是通过调用 `umap_local` (9983 行) 来完成的。

对内存进行引用的第二种方法是针对分配给进程的除文本段、数据段、栈段之外的内存区域，对此进程负有相应的职责。显卡驱动程序和以太网卡驱动程序便是这类例子。显卡和网卡需要映射从 0xa0000 到 0xfffff 的 I/O 设备保留地址的内存。另一个例子则是内存驱动程序，它管理 RAM 盘并且通过 `/dev/mem` 和 `/dev/kmem` 设备可以提供任何内存部分的项。对于这种内存地址的由虚拟地址到物理地址的转换请求则通过 `umap_remote` 进行处理 (10 025 行)。

最后，则是有关被 BIOS 使用的内存。这通常包括最低处的 2 KB 内存，这低于 MINIX 3 所加载的地址，以及从 0x90000 到 0xfffff 的区域，这包括 MINIX 3 加载地址之上的 RAM 区域以及 I/O 设备的保留区域。这也是通过 `umap_remote` 进行处理的，但使用的是第三个函数 `umap_bios` (10 047 行)，并且要确保对于该引用内存确系此区域的检查。

`system.c` 中定义的最后一个函数是 `virtual_copy` (10 071 行)。该函数的绝大部分是 C 语言的 `switch`，用来决定使用上述进行虚拟地址到物理地址转换的三个 `umap_*` 函数之一。这是为源地址和目标地址的转换而做的。实际的复制是通过调用 `klib386.s` 中的一个汇编语言程序 `phys_copy` 来完成的 (10 121 行)。

2.7.3 系统库的实现

每一个名字为 `do_xyz` 形式的函数的源代码都在 `kernel/system/do_xyz.c` 文件中。`kernel/` 路径下的 `Makefile` 包含一行代码：

```
cd system && $(MAKE) -$(MAKEFLAGS) $@
```

它将 `kernel/system/` 内的所有文件编译进了 `kernel/` 主路径下的库文件 `system.a` 中。当控制返回到主 `kernel` 路径时，`Makefile` 的另一行代码将使得每次内核目标文件被链接时都首先查找这个本地库文件。

在本书附录 B 中列举了 `kernel/system/` 路径下的两个文件。之所以选择它们，是因为它们代表了系统任务提供的两类通用的支持。一类支持是对于代表任何需要此类支持的用户空间系统进程的内核数据结构的访问。本书将以 `system/do_setalarm.c` 作为此类的一个例子进行讲述。另外一个通用的支持是对于特殊的系统调用，它们绝大多数是由用户空间进程进行管理的，但需要在内核空间执行一些操作。`system/do_exec.c` 便是其中之一。

`sys_setalarm` 内核调用稍微有点类似在内核中断处理一节中讨论过的 `sys_irqenable`。当引发一个 IRQ 时，`sys_irqenable` 对将要调用的中断处理器设置地址。这个处理器是系统任务中的一个函数 `generic_handler`。它生成一个通知消息给需要对此中断响应的设备驱动程序。`system/do_setalarm.c` (10 200 行) 包含了管理定时器的代码，管理的方法类似于中断管理。`sys_setalarm` 内核调用为一个需要接收同步 `alarm` 的用户空间系统进程初始化一个定时器，并且它还提供了一个函数，此函数在定时器到期时被调用以通知用户空间进程。它还可以通过在它的请求消息的终止时间位传输一个 0 来要求取消一个先前的调度 `alarm`。在 10 230 到 10 232 行中从消息中提取信息的操作

很简单。最重要的是定时器需要知道逝去的时间以及需要知道此时间的进程。每一个系统进程在 *priv* 表中都有它自己的定时器结构。从 10 237 到 10 239 行定义了此结构，并且加入了进程号和定时器终止时所要执行的函数 *cause_alarm* 的地址。

如果定时器已被激活，则 *sys_setalarm* 在它的回复消息中返回剩余时间。如果返回 0 则表示尚未被激活。有几种可能需要考虑。其一，定时器可能在之前已经成为无效定时器，即通过将它的 *exp_time* 位设为 *TMR_NEVER* 值将其标记为非活动状态。具体到 C 代码则是只要设为一个大整数，所以直接测试此值是作为检验期满时间是否已过的一部分操作内容。定时器可以显示一个已经过去的时间。虽然这不太可能发生，但这是很容易检测的。定时器还可以显示一个将来的时间。在此两种情况下，其返回值都是 0，否则将返回剩余时间（10 242 到 10 247 行）。

最后，是关于定时器的设置和复位。这是通过在定时器结构体中的恰当位段写入期望的预置时间并调用另一个函数来完成的。当然，定时器复位不需要存储任何值。关于定时器的复位和设置函数的代码在时钟任务的源文件中，将会很快进行介绍。但是既然系统任务和时钟任务都被编译进了内核映像中，所有声明为 *PUBLIC* 的函数都是可访问的。

在 *do_setalarm.c* 文件中还定义了另一个函数。这就是 *cause_alarm*，这是一个看门狗函数，它的地址存储于每一个定时器里，所以当定时器到期时可以调用它。它本身很简单。它向进程号也存储于定时器结构体的进程发送一个通知消息。这样内核里的同步 *alarm* 就被转换为一个消息并发送给请求 *alarm* 的系统进程。

顺便指出，请注意前面讨论定时器的初始化时曾提到对于系统进程所请求的同步 *alarm*（在本节也是这样）。如果这些在这一节并未完全阐述清楚，或者如果读者想要知道什么是同步 *alarm* 或者有关非系统进程的定时器的情况，那么这些问题将会在下一部分讨论时钟任务时进行更详细的介绍。操作系统中各部分之间存在如此多的相互联系，以至于几乎不可能将所有主题依次进行讲解，而不用偶尔提到一些还不曾介绍又不适合讲解的内容。若不是在分析一个实际的操作系统，也许可以绕过这些琐碎的细节。如果这样，有关操作系统原理的理论讲解也许根本不用提及系统任务。在介绍理论的书籍中只需挥舞双臂滔滔不绝地讲解理论而忽略掉很多具体问题，比如操作系统用户空间的部件对于中断及 I/O 端口等特权资源的受限的、受控制的访问。

目录 *kernel/system/* 中的将要具体介绍的最后一个文件是 *do_exec.c*（10 300 行）。*exec* 系统调用的绝大部分工作是由进程管理器完成的。进程管理器为包含参数及环境信息的进程设置一个栈，然后将栈指针通过 *sys_exec* 传送给内核，这是由 *do_exec* 处理的（10 618 行）。栈指针被设置在进程表的内核部分，如果该正在被 *do_exec* 处理的进程在使用附加段，那么在 *klib386.s* 中定义的 *phys_memset* 函数将被调用，以擦除这个内存区域中可能留有的先前应用中的残留数据（10 330 行）。

exec 调用会导致一些轻微的异常。调用它的进程发送一个消息给进程管理器，然后阻塞。对于其他的系统调用，导致的回复消息会解除阻塞，但是对于 *exec* 则没有回复消息，因为新加载的内核映像不期待一个回复（*reply*）。因此，在 10 333 行，*do_exec* 自己为进程解除了阻塞态。下一行使用防止可能的竞争条件的 *lock_enqueue* 函数使得新映像就绪以待运行。最后，保存命令字符串以便在用户调用 *ps* 命令或者按下从进程表中显示数据的功能键时标识进程。

在结束关于系统任务的讨论之前，再看一下它在处理典型服务操作时所扮演的角色，比如为 *read* 系统调用的响应提供数据。当一个用户执行了 *read* 调用后，文件系统将检查它的缓冲区以确定它是否已存有一个需要的数据块，如果没有，它会向相应的磁盘驱动程序发送一条消息以将该数据块载入到缓冲区。然后文件系统发送一条消息给系统任务以通知将该数据块复制到用户进程。最坏情况下，读取一个数据块需要 11 条消息；最好情况下则需要 4 条。在图 2.46 中给出了这两种情况。在图 2.46(a) 中，消息 3 请求系统任务执行 I/O 操作；消息 4 是确认信号。当一个硬件中断发生

时，系统任务通过消息 5 告诉等待中的驱动程序这个事件。消息 6 和消息 7 分别是请求复制数据到文件系统的消息以及回复。消息 8 告诉文件系统数据已经准备好，然后消息 9 和消息 10 是将数据由缓冲区复制给用户的请求消息和回复。最后，消息 11 是一个给用户的回复。在图 2.46(b)中，数据已经存在于缓冲区中，消息 2 和消息 3 分别是将数据复制给用户的请求和回复。这些消息是 MINIX 3 中的开销的源头和为了追求高度模块化设计所付出的代价。

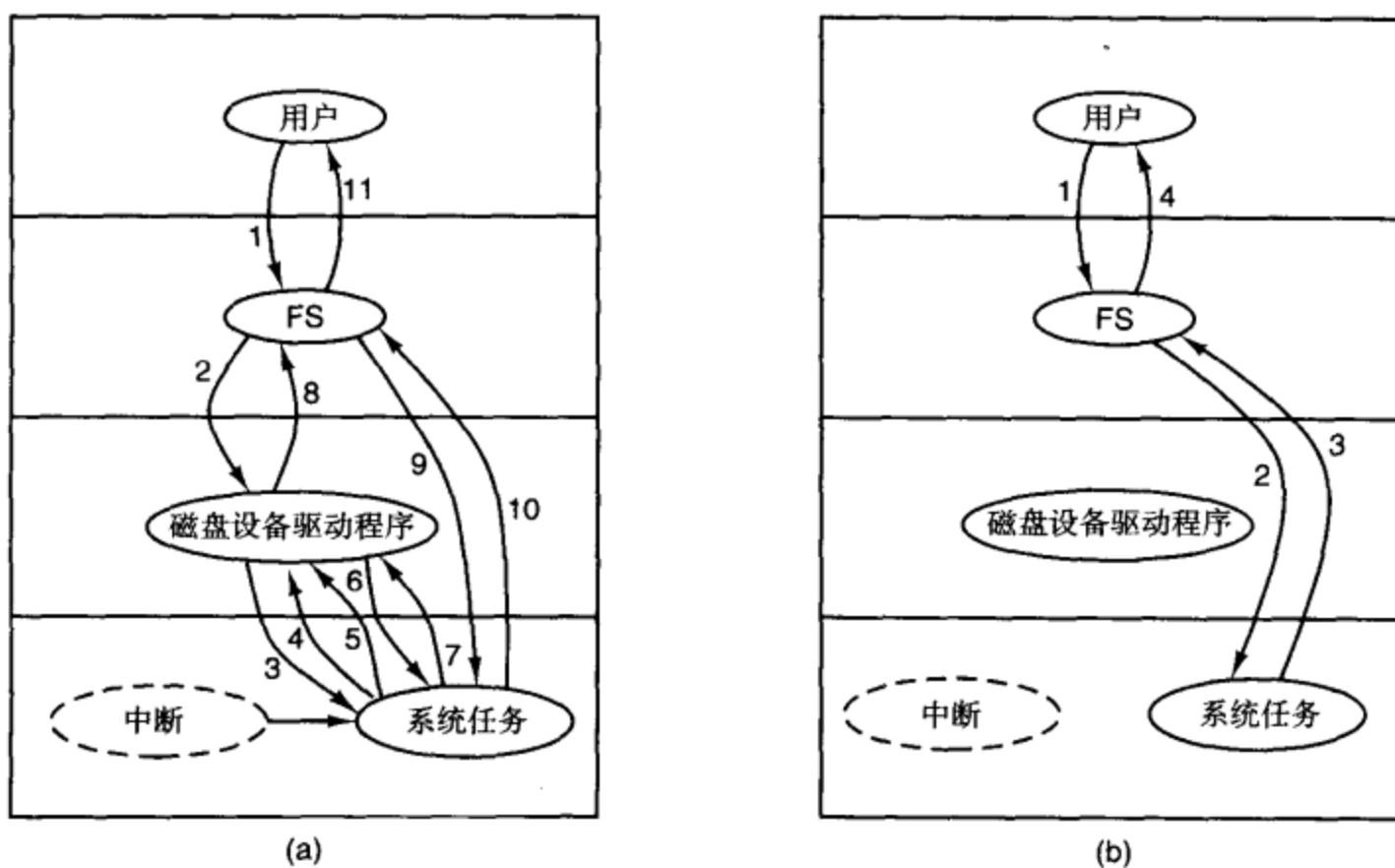


图 2.46 (a)读取数据块请求最坏情况需 11 条消息；(b)读取数据块请求最好情况需 4 条消息

复制数据的内核调用很可能是 MINIX 3 中使用得最频繁的操作。前面已经讲述了系统任务中用来完成此工作的函数 *virtual_copy*。改进使用消息传递机制带来的效率低下问题的一种方法就是将多个请求打包成一条消息。*sys_virvcopy* 和 *sys_physvcopy* 内核调用正是用来完成此事的。调用它们的消息的内容是一个指针，该指针指向一个向量，而该向量记载了需要在内存中进行复制的多个数据块的地址。*sys_virvcopy* 和 *sys_physvcopy* 都是由 *do_vcopy* 来支持的，*do_vcopy* 执行一个循环，抽取源地址、目标地址和数据块长度并反复调用 *phys_copy* 直到所有的复制完成。在下一章中，我们将看到磁盘驱动程序有能力通过一个请求完成多个传输。

2.8 MINIX 3 的时钟任务

由于种种原因，时钟（也称为定时器）对于任何分时系统的运行都是很必要的。比如，它们维护每天的计时，防止一个进程独占 CPU。MINIX 3 的时钟任务与设备驱动程序有些类似，驱动程序由硬件设备的中断所驱动。但是，时钟既不是像磁盘那样的块设备，也不是像终端那样的字符设备。事实上，在 MINIX 3 中，时钟的接口并非由 */dev/* 路径下的某个文件提供。进一步说，时钟任务在内核空间执行，它不能被用户空间进程直接访问。它可以访问所有的内核函数和数据，但是用户空间进程只能通过系统任务对它进行读写。在本节中我们将首先看一下通用的时钟硬件及软件，然后看看在 MINIX 3 中它们是怎么实现的。

2.8.1 时钟硬件

在计算机中使用了两类时钟，但都与人们日常使用的表或者钟不同。其中简单点的时钟是连接在110 V或者220 V的电线上，在每个电压周期产生一个中断，大概为50 Hz或60 Hz。在现代的计算机上这种时钟基本上已经不再使用了。

另一种时钟由三个部分组成：一个晶体振荡器，一个计数器，一个支持寄存器，参见图2.47。当一个石英晶体被适当切割并加上电压后，它将产生高度准确的周期性信号，一般在5~200 MHz，依据选择的石英而有所不同。在任何计算机中至少能找到一个这样的电路，它为计算机的各类电路提供同步信号。这个信号馈到计算机使它趋于零计数。当计数器倒数到零时产生一个CPU中断。主频高于200 MHz的计算机常常使用一个低频率的时钟和一个分频电路。

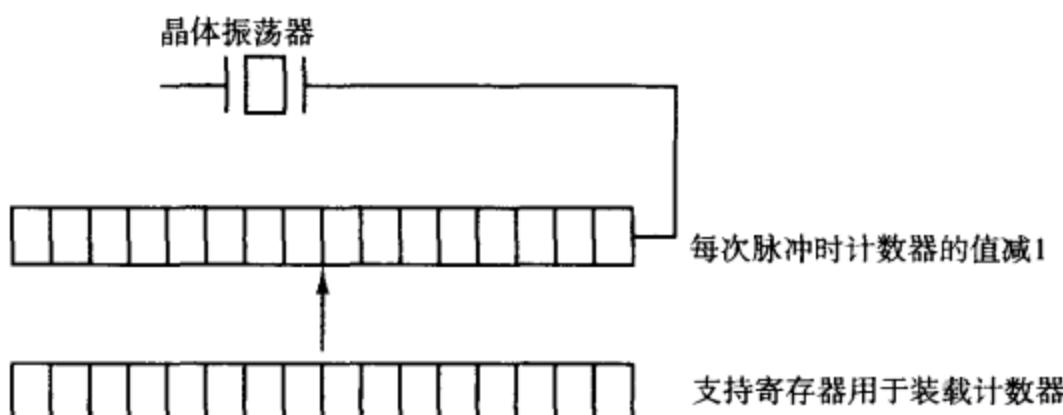


图2.47 可编程时钟

典型的可编程时钟有多种执行模式。在单次模式 (one-shot mode) 下，当时钟被启动时，它将支撑寄存器中的值复制到计数器中，然后晶振每来一个脉冲，它的计数值减1，当计数器减至零时，它就引发一个中断并停止，直到某个程序再次显式地启动它。在方波模式 (square-wave mode) 下，在计数到零并引发了一个中断之后，寄存器将自动地把其中的值重新复制到计数器中，重新开始计数，这种周期性的中断称为时钟节拍 (clock ticks)。

可编程时钟的优点是其中断频率可以由软件控制。如果使用的是1 MHz的晶振，那么计数器则每毫秒产生一个脉冲。对于16位的寄存器来说，中断产生的间隔可以编程为从1 μ s到65 536 ms。可编程时钟芯片通常含有两到三个独立的可编程定时器，还有其他一些可选择的模式（例如，是由高到低计数还是由低到高计数，屏蔽中断，等等）。

为防止计算机关机后所存储的当前时间丢失，绝大多数计算机都有一个使用备用电池的计时器，使用那种用于数字时钟中的低压电路来运行。该电池时钟可以在机器启动时被读取。如果没有备用的时钟，那么软件会询问用户当前日期和时间。还有一种标准协议用来从远程主机通过网络系统来得到当前时间。任何一种情况下，时间都随后被转换为国际通用协调时间（Universal Coordinated Time，简称UTC，以前称为格林尼治时间：GMT），也就是距离1970年1月1日零点间隔的秒数，这跟UNIX和MINIX 3是一样的，或者与其他标准时间点的间隔秒数是一样的。运行的系统对时钟节拍进行计数，每过1秒钟，计数器加1。MINIX 3（以及大多数UNIX系统）不考虑闰秒，闰秒自从1970年以来一共有过23次。这并不是严重的纰漏。通常情况下，应用程序由手动来设置系统时钟和备用时钟这两个时钟的同步。

在此还应该提到，在早期的IBM兼容机中有一个独立的时钟电路来为CPU提供时钟信号，通过内部的数据总线和其他部件。这就是人们讲到CPU主频速度时所指的时钟，在早期个人计算机上用MHz为单位，现在的个人计算机则使用GHz衡量。晶振和计数器的基本电路是相同的，但是对它们的要求则有很大不同，以至于现在的计算机都有独立的时钟来保证CPU控制和计时。

2.8.2 计时程序

所有的时钟电路都用来产生确知间隔的中断。涉及到计时的任何东西都必须由软件来完成，即时钟驱动程序。不同的系统中时钟驱动程序的确切职责也有所不同，不过通常包括如下部分：

1. 维护日常时间。
2. 防止进程运行时间超过它们的允许运行时间。
3. 审计 CPU 的使用。
4. 处理用户进程发出的 `alarm` 系统调用。
5. 为系统自己的某些部分提供看门狗时钟。
6. 执行测试统计（profiling），监视并获取统计数据。

上述第一个功能，即维护日常时间（也称为实际时间）比较容易，前面讨论过，这只需在每一个时钟节拍将计数器加 1，不过其中有一点需要注意，即在日常时间计数器中的位数。对于一个 60 Hz 的时钟，32 位的计数器最大只能存储稍多于两年的时间，所以系统使用 32 位的计数器显然不够存储自从 1970 年 1 月 1 日以来的节拍数。

可以采取三个办法来解决这个问题。第一种就是使用一个 64 位的计数器，不过这样做使得维护一个计数器太不经济，因为这样的话 1 秒钟要会增加很多次计算。第二种方法是用秒为单位来维护日常时间，而不是用节拍数，在每秒内使用一个辅助计数器来计节拍数。因为 2^{32} 秒比 136 年还长，所以这种方法可以在进入 22 世纪时依然正常工作。

第三种方法是依然以节拍数来计数，但是是从系统启动那一刻开始的，而不是从某个固定的外部时刻开始。当备用时钟被读取或者用户输入真实时间时，系统启动时间就可以从当前真实时间值计算出来并用任何方便的形式存储在内存中。当需要当前时间时，存储的时间值加上计数器的时间值就可以得到当前的时间。以上这三种办法如图 2.48 所示。

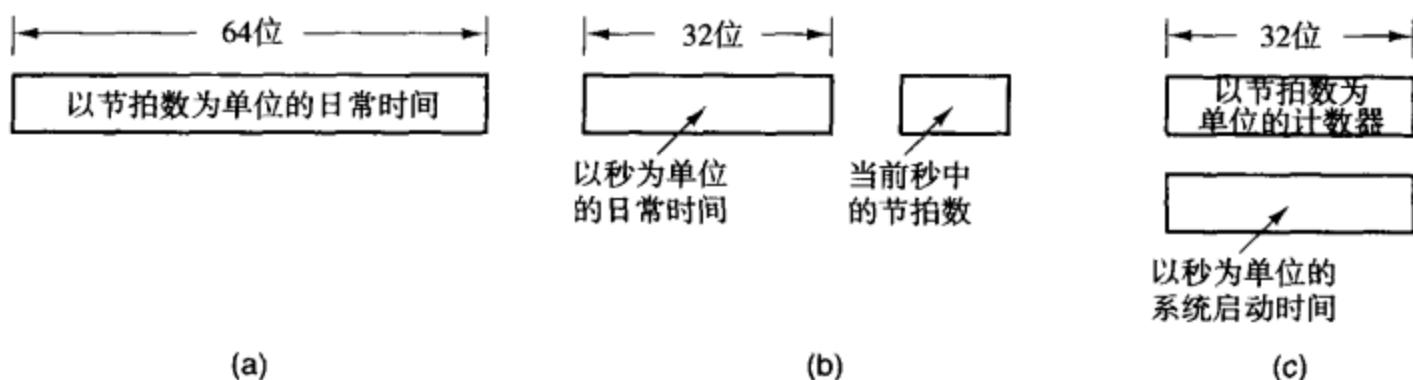


图 2.48 日常时间维护的三种方法

时钟的第二个功能是防止某个进程执行过长的时间。每当一个进程被启动运行时，调度器就会初始化一个计数器为该进程的时钟节拍数值。每来一个时钟中断，该值减 1，当减为零时，时钟驱动程序将会通知调度器启动另一个进程。

第三个时钟功能是记录进程已使用的 CPU 资源。最准确的方法是当启动一个进程的同时，启动另外一个定时器，即非系统主定时器。当进程停止时，可以从该定时器读出该进程运行了多久，为更加准确，该辅助定时器在每次中断发生时应该保存其值然后恢复。

一种有些粗略但是较简便的计数方法是在全局变量区为当前运行的进程维护一个进程表项的指针。每来一个时钟节拍，当前进程表项的一个字段就加 1。这样的话，每一个时钟节拍就被转换成了该进程运行的节拍数表示的时间。这种方法的一个小问题是如果在该进程运行期间发生了太多的

中断，那么对于所有的节拍该进程的运行时间都按一个全时钟周期进行计算，尽管在这个时钟内它并未能做多少工作。但是对中断期间的 CPU 进行精确计数花销太大而很少采用。

在 MINIX 3 以及其他的一些系统中，进程可以要求操作系统在间隔一定时间后给它一个 alarm。该类 alarm 通常是个信号、中断、消息或者其他类似的形式。网络活动便是这种请求 alarm 的一个应用，在网络活动中，发送一个数据包之后如果在一定时间内得不到确认信息，则必须重新发送。另一个应用则是计算机辅助教学，如果学生在一定时间内没有做出回答则系统会给出答案。

如果一个时钟驱动程序拥有足够的时钟，那么它可以为每个请求分配一个独立的时钟。但事实并非如此，它必须利用一个物理时钟模拟出多个虚拟的时钟。一种方法便是维护一张表，在这张表中保存有代表各未完成的定时器的信号时间，还有一个代表下一个进程所需时间的变量。一旦真实时间被更新，驱动程序则检查最邻近的信号是否发生了，如果发生了，该表将查找下一个将要发生的信号。

如果有多个将要发生的信号，那么通过这种将各未完成的时钟请求连接到一起来模拟多个时钟的方法是很有效的，正如在图 2.49 中所展现的那样。列表的每一项显示了在先前的一个信号发生后到后续信号发生之前要等多少个时钟节拍。如果表中存有此信息，那么这张表会查找后续信号。在这个例子中，待处理的信号的发生时间是在时钟节拍为 4203, 4207, 4213, 4215 和 4216 时。

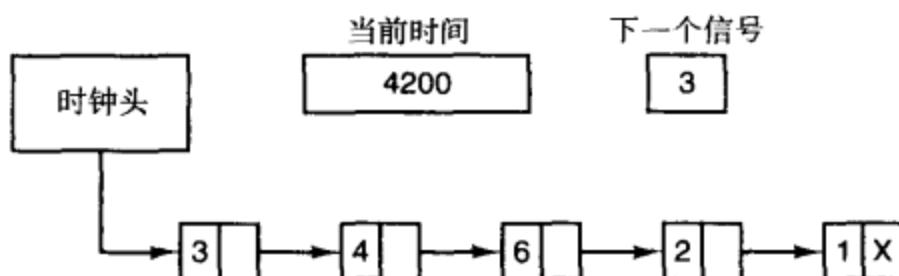


图 2.49 使用单个时钟模拟多个定时器

在图 2.49 中，刚刚有一个定时器到时。下一个中断将在 3 个节拍之后发生，3 已被载入。每来一个节拍，后续信号的计数值将减 1，当减到零时，线形表中对应第 1 个条目的信号将发生，同时该条目也从表中被删除。然后后续信号将被置为表头项的值，本例中为 4。在很多情况下，使用绝对节拍数比相对节拍数要方便，MINIX 3 使用的正是这种方法。

注意在一个时钟中断期间，时钟驱动程序有几件事情要做。包括继续推进日常时间，(对当前进程所剩时钟) 消耗时钟数并检查是否到零，对 CPU 使用计数，以及减小 alarm 计数器的值。所有这些操作都被仔细地安排，以快速运行，因为在 1 秒内这些操作要反复执行很多次。

操作系统的一些部分也需要设置定时器。这些定时器被称为看门狗时钟(定时器)。在稍后介绍硬盘驱动器时会看到，唤醒调用就是每当磁盘控制器接收到一个命令时被调度，所以当一个命令彻底失败时可以尝试恢复。软盘驱动器使用定时器来等待磁盘马达加速到一定速度，如果一段时间内无事可做，则关闭马达。一些带有移动打印头的打印机可以以 120 字符/秒 (8.3 ms/字符) 的速度打印，但是无法在 8.3 ms 内将打印头移动到左边空白处，所以中断驱动器必须延迟直到机械架返回。

时钟驱动程序处理看门狗时钟的机制与对待用户信号的机制相同。唯一的不同点在于，当定时器触发时，不是引发一个信号，而是由时钟驱动程序调用一个调用者提供的进程。该进程是调用者的代码的一部分，这会带来一个 MINIX 3 设计中的问题，因为其目的之一就是将驱动程序从内核地址空间移出。简单地说就是利用内核空间内的系统任务，它能设置代表一些用户空间进程的 alarm，并且在定时器到时间时通知这些进程。这些内容将会在后面部分详细讲解。

列表中最后一项功能是测试统计(profiling)。一些操作系统提供了一种机制，在这种机制下用户程序可以使系统为它的程序计数器绘制统计图，从而该用户程序可以知道它的时间开销都花在了

哪里。当启用测试统计时，驱动程序会对每一个节拍都进行检查，以察看当前进程是否正被测试统计，若是，则计算当前进程对应的域号（一段地址范围）。然后将该域值加1，这种机制也可以用来测试统计操作系统本身。

2.8.3 MINIX 3 中的时钟驱动程序总览

MINIX 3 时钟驱动程序包含在 *kernel/clock.c* 文件中。可以认为它包含三个功能部分。第一，与下一章将要介绍的设备驱动程序类似，它也有一个任务管理机制，它循环运行，等待消息然后将每个消息分派到执行实际操作的子程序执行。然而，这种结构在时钟任务中过于落后。消息机制开销很大，需要大量的上下文切换。所以对于时钟只有当有大量的工作要做时才会使用该机制。在这里仅接收一类消息，也仅有一个子程序来处理该消息，工作正在进行时不进行消息回复。

时钟程序的第二个主要部分是每秒被激活 60 次的中断处理程序。它维护基本的日常时间，更新记录系统启动以来的时钟节拍数的变量，并将其与下次定时器到时的时间进行比较。寄存当前进程本次已使用的以及一共所使用了的时间的计数器也由它来更新。如果中断处理程序检测到一个进程用完了它的运行时间数或者一个定时器到时间了，它将发送一个返回主循环任务的消息。否则不发送任何消息。此处的策略就是对于每个时钟节拍，中断处理程序用尽可能快的速度做尽可能少的事。开销较大的主任务只在有大量工作要做时才被激活。

时钟程序的第三部分是一系列起支持作用的子程序，但它们不是被中断处理程序或者主循环任务来调用以响应时钟中断的。这些子程序中有一个程序被编码为 *PRIVATE*，它在进入主循环任务之前被调用。其工作是初始化时钟，将数据写入时钟芯片以使它能够产生需要间隔的中断。该初始化程序还将中断处理程序的地址放在合适的地方，以便时钟芯片向中断控制芯片触发了 8 号中断时能够找到，并使能它以响应输入的中断。

clock.c 中的其他子程序则被声明为 *PUBLIC*，因而可以在内核代码的任何地方进行调用。事实上，它们中没有一个是被 *clock.c* 自身调用的。大多数情况下是由系统任务调用它们来为与时间有关的系统调用服务。这些子程序所完成的功能包括读取启动时间计数器以通过时钟节拍来计时，还有读取时钟芯片本身的寄存器，这样可以按照要求的 ms 单位进行计时。另外还有一些子程序用来设置或重置定时器。最后，当 MINIX 3 关闭时也会调用一个子程序，由它重置 BIOS 所需要的硬件时间参数。

时钟任务

时钟任务的主循环只接收一类消息，即 *HARD_INT*，它来自中断处理程序。其他任何消息都是错误。还有，它并非接收每一个时钟节拍的中断产生的消息，尽管每次接收到一个消息都会调用的子程序名字为 *do_clocktick*。当接收到一个消息时，只有进程调度器需要时或者有一个定时器已到期时，*do_clocktick* 才会被调用。

时钟中断处理程序

时钟中断处理程序在每次时钟芯片内的计数器到达零并产生一个中断时开始运行。这正是基本的计时功能所要完成的。在 MINIX 3 中用如图 2.48(c)所示的方法来维护时间。而在 *clock.c* 中只有节拍计数器在系统启动后被维护，系统启动时间则记录在其他地方。时钟程序只提供当前节拍数来辅助相应系统调用计算真实时间。更深层次的处理是由某个服务器来完成的。这与 MINIX 3 将功能性模块移到用户空间来运行的策略是一致的。

中断处理程序中的本地计数器每接收到一个中断就会被更新。当中断被禁止时时钟节拍会丢失，在有些情况下可以纠正这个影响。可以使用一个全局变量来计算丢失的节拍数，再加上主计数

器，然后每次中断处理程序被激活时将该变量清零。在讲述应用时，我们将会介绍一个关于它的使用情况的例子。

中断处理程序还会影响进程表中的用于记账及进程控制的变量，仅当当前进程超过了下一个被调度的定时器的限定时间或者正在运行的进程的时钟节拍数已经减小到零时，一条消息才将被发送到时钟任务。中断服务中进行的操作都是些简单的整数运算、比较、逻辑与或非，或者通过C编译器可以很容易地转换为基本机器运算。最坏情况下会有五个加或减运算和六个比较运算，以及一些逻辑运算和完成一些中断服务所需要的分配。特殊情况下甚至没有子程序调用开销。

看门狗时钟

后面还有几页包含关于怎样为用户空间进程提供看门狗时钟的问题，通常认为这是提供给用户的，也是用户代码的一部分的进程，它在定时器到期时被执行。显然，MINIX 3 中不能这样做。但是可以使用一个同步警报将内核与用户空间联系起来。

现在正是介绍同步警报的意思的时候。一个信号可能到达或者一个传统的看门狗可能被激活，而这个看门狗与当前正在运行的进程的部分没有任何关系，所以这样就导致了不同步。同步警报是以消息的形式被传递的，这样只有当接收者执行了 `receive` 操作后，该消息才能被接收。之所以称为同步，是因为只有在接收者也期望它时该消息才能被接收。如果该 `notify` 方法用来向一个接收者通知一个警报消息，那么发送者则不必阻塞，接收者也不必关心是否错过了一个警报消息。因为如果接收者并不是在等待该 `notify` 消息，那么它会被保存起来。使用一个位段，其中每位代表一个可能的通知源。

看门狗时钟使用了 `priv` 表中任何一个元素都有的一个 `timer_t` 类型的变量 `s_alarm_timer`。每一个系统进程在 `priv` 表中都有一个对应的项，为了设置一个定时器，用户空间的系统进程进行 `sys_setalarm` 调用，这是由系统任务来处理的。系统任务被编译进了内核空间，从而能够初始化一个代表正调用的进程的定时器。初始化过程会将定时器到期时所要执行的进程的地址放入正确的区域，然后将该定时器插入到定时器列表中，就像图 2.49 所示的那样。

当然，将要执行的进程也必须位于内核空间。这没有问题。系统任务包含看门狗函数 `cause_alarm`，它在到期时产生一个 `notify`，从而为用户引发一个同步警报。这个警报可以调用用户空间的看门狗函数。在内核里这是一个真实的看门狗，但是对于请求定时器的进程来说，这只是一个同步警报。它不同于使定时器在目标地址空间执行一个进程。虽然这样做的开销会大一些，但是比中断要简单。

上面所讲的是可行的：这里曾提到系统任务能够为一些用户空间进程设置警报。该机制仅仅为系统进程描述了它们的任务。每一个系统进程都有一个 `priv` 结构副本，但是单独的一份副本可以被所有的非系统（用户）进程所共享。而 `priv` 表中的部分是不能共享的，比如未决的通知及定时器的位图就不能被用户进程所使用。解决办法是，进程管理器使用一种类似系统任务为系统进程管理定时器的方法来管理代表用户进程的定时器。每一个进程在进程表中的进程管理器部分都有它自己的 `timer_t` 成员变量。

当一个用户进程产生一个 `alarm` 系统调用来请求设置一个警报时，它会被进程管理器处理：设置一个定时器并将其插入到定时器列表中。当进程管理器的定时器列表中的第 1 个定时器到期时，进程管理器请求系统任务发送一个通知。当定时器列表的链头发生改变时，要么是因为第 1 个定时器到期或者被取消，要么是因为收到一个新的、必须置于当前链头的之前的请求。这是用来支持 POSIX 标准的 `alarm` 系统调用。此程序是在进程管理器的地址空间执行的，执行时，请求 `alarm` 的用户进程发送一个信号而不是回复确认信号。

毫秒计时

clock.c 中提供了一个程序进行微秒计时。很多 I/O 设备要求短至数微秒的延迟。使用警报和消息传递接口无法做到这一点。用来产生时钟中断的计数器可以直接读取，它大概每 $0.8 \mu\text{s}$ 减去 1，这样如果每秒 60 次减至零，则大概 16.67 ms 一次。为了对 I/O 计时有用，需要有一个运行在内核空间的程序轮询此计数器，但是将驱动程序从内核空间移出需要做很多的工作。目前该功能只用来作为随机数发生器的源数字。也许在一个速度更快的系统中它能起到更大的作用，但这是将来的项目。

时钟服务小结

图 2.50 总结了 *clock.c* 直接或者间接提供的各种服务。一些函数声明为 *PUBLIC*，因而可以被内核或者系统任务调用。而其他的所有的服务都只能被由系统任务最终控制的系统调用间接使用。其他系统进程可以直接请求系统任务，但是用户进程则必须请求进程管理器，而它事实上也依赖于系统任务。

服务	访问	响应	客户
get_uptime	函数调用	节拍	内核或系统任务
set_timer	函数调用	无	内核或系统任务
reset_timer	函数调用	无	内核或系统任务
read_clock	函数调用	计数	内核或系统任务
clock_stop	函数调用	无	内核或系统任务
Synchronous alarm	系统调用	通知	服务器或驱动程序，通过系统任务
POSIX alarm	系统调用	信号	用户进程，通过 PM
Time	系统调用	消息	任何进程，通过 PM

图 2.50 时钟驱动程序支持的时间相关服务

内核或系统任务能够在不需要消息开销的情况下获取当前时间、设置或者重置一个定时器。内核或系统任务还可以调用 *read_clock*，该调用用来读取定时器芯片上的计数器，以约 $0.8 \mu\text{s}$ 为单位来获取时间。而当 MINIX 3 关机时则会调用 *clock_stop* 函数，它存储 BIOS 时钟速率。一个系统进程（无论是驱动程序还是服务器）都可以请求一个同步警报，该警报将激活内核空间的一个看门狗函数，并向发送请求的进程发送一个确认信息。用户进程则首先通过进程管理器请求一个 POSIX 警报，由管理器再请求系统任务激活一个看门狗。定时器到期时，系统任务通知进程管理器，进程管理器再将信号传递给用户进程。

2.8.4 MINIX 3 中的时钟驱动程序的应用

系统任务没有使用主要的日期结构体，而是使用了一些变量来记录时间。变化中的真实时间（10 462 行）是最基本的。它对所有的时钟节拍计数。全局变量 *lost_ticks* 在 *glo.h* 中定义（5333 行）。它是提供给运行在内核空间的函数使用的，因为内核空间由于禁用中断足够长时会丢失一个或多个时钟节拍。目前它主要由文件 *klib386.s* 中的 *int86* 函数使用。*int86* 使用引导监控程序来管理对于 BIOS 的控制，而监控程序返回在返回到内核之前 BIOS 调用忙期间 *ecx* 寄存器内对时钟节拍的计数值。之所以能够这样做，是因为尽管在 BIOS 请求处理时时钟芯片并未触发 MINIX 3 时钟中断处理程序，但是引导监控程序在 BIOS 的帮助下能够保持对时间的跟踪。

时钟驱动程序还访问其他几个全局变量。它使用 *proc_ptr*, *prev_ptr* 和 *bill_ptr* 分别引用当前运行的进程、之前运行的进程以及取得时间片段的进程的进程表项。在这些进程表项里它访问多个变量，包括计算时间的 *p_user_time* 和 *p_sys_time*，还有计算进程剩余时钟节拍数的 *p_ticks_left*。

MINIX 3启动时，会调用所有的驱动程序。绝大多数驱动程序都是先进行初始化然后尝试接收一个消息就进入阻塞态。时钟驱动程序 *clock_task* (10 468行) 也是如此。首先它调用 *init_clock* 将可编程时钟发生器的频率初始化 60 Hz，收到一条消息后，如果是 *HARD_INT* (10 486行) 则调用 *do_clocktick*，而对于其他种类的消息都作为错误对待。

并不是任何一个时钟节拍都会调用 *do_clocktick* (10 497行)，所以它的名字并未确切地描述其功能。当中断处理器确认有些重要的事情要做时才会调用它。条件之一是当前进程用完了它的时间片段。如果这个进程是可抢占的（系统任务和时钟任务都不是），那么在调用 *lock_enqueue* (10 510 到 10 512行) 之后会立即调用 *lock_enqueue* 来从该进程的队列中将其清除，再将其置为就绪态并重新调度它。另一种会调用 *do_clocktick* 的情况是一个看门狗时钟到期了。在 MINIX 3 中定时器及其链表的使用如此之广，以至于专门为它们创建了一个函数库来支持它们。这个函数库 *tmrs_exptimers* 在 10 517 行被调用，它用来为所有到期的定时器执行看门狗功能，并使其无效。

init_clock (10 529行) 只在时钟任务启动时被调用一次。有几个地方会被人们指出并声称“这是 MINIX 3 开始运行的地方！”。这是一个候选；时钟对于一个多重任务的抢占式系统是基本的要求。*init_clock* 往时钟芯片中写入三个字节数据以设置它的运行模式以及设置主寄存器合适的计数值。然后它设置它的进程号、IRQ 以及使中断能够正确执行的处理程序地址。最后，启用中断控制器芯片以接收时钟中断。

下一个函数 *clock_stop* 则执行与时钟芯片的初始化相反的工作。它声明为 *PUBLIC*，但并不是在 *clock.c* 的任何地方都调用它。放在这里讲述它是因为它与 *init_clock* 有明显的相似之处。它只在 MINIX 3 关闭时才被系统任务调用，然后控制权返回给引导监控程序。

一旦 *init_clock* 运行（或者更准确地说，是 16.67 ms 之后），就会产生第 1 个时钟中断，在 MINIX 3 运行期间每秒产生 60 次时钟中断。*clock_handler* (10 556行) 的代码也许是 MINIX 3 系统中执行最频繁的部分。因此，*clock_handler* 被设计为速度至上。唯一的一个子程序调用是在 10 586 行；仅在运行于 IBM PS/2 系统上时才需要它。在 10 589 到 10 591 行，对当前时间（按节拍数）进行更新。然后用户和审计时间再被更新。

有可能被询问的处理程序在设计时进行了判断。在 10 610 行进行了两个测试，如果其中之一为真则时钟任务会被通知。由时钟任务调用的 *do_clocktick* 函数重复这两项测试以决定需要做什么。这样做是必要的，因为处理函数所用的 *alarm* 调用不能传递任何可以区分不同条件的信息。在此将该问题留给读者去考虑，以便进行选择以及对其进行评价。

clock.c 的其余部分包括的有用函数都已经提到过。*get_uptime* (10 620行) 返回真实时间值，它只对 *clock.c* 中的函数可见。*set_timer* 和 *reset_timer* 使用定时器库中的其他函数来负责操作定时器链的细节。最后，*read_clock* 读取并返回时钟芯片的向下计数寄存器的当前值。

2.9 小结

为了屏蔽中断的影响，操作系统提供了一个由并发运行的顺序进程所构成的概念模型。进程之间可以使用进程间通信原语来相互通信，比如信号量、管程、消息等。使用这些原语是为了保证任一时刻不会同时有两个进程进入临界区。进程的状态可以是运行态、就绪态或阻塞态，当它或另一个进程执行一条进程间通信原语时可能会引起状态的改变。

进程间通信原语可用来解决诸如生产者-消费者、哲学家进餐、读者-写者等问题。但即使使用了这些原语，也必须注意避免错误和死锁。目前已经有许多种调度算法，包括时间片轮转、优先级调度、多级队列以及策略驱动的调度等。

MINIX 3 支持进程概念，并提供了用于进程间通信的消息。消息是不加以缓冲的，所以一条 `send` 操作只有在接收者正在等待它时才能成功。同样，一条 `receive` 操作只有在消息已经可用时才能成功。这两种操作的调用进程在操作不成功时都将阻塞。MINIX 3 还使用 `notify` 原语对消息机制提供非阻塞的补充。如果试图向不可用的接收者发送一个 `notify`，则会导致一位被置位，它将在稍后完成了接收时触发通知。

考虑一个进行 `read` 操作作为信息流的例子。当用户进程发个消息给文件系统来请求读取数据时，如果该数据并未在文件系统缓存中，那么文件系统就会请求驱动程序从磁盘中读取。然后文件系统阻塞以等待该数据，当磁盘中断产生时，文件系统被唤醒，然后给磁盘驱动程序一个回复，接着会有一个给文件系统的回复。这样，文件系统任务请求系统任务从它的缓冲中将刚才放入的被请求的数据复制给用户。图 2.46 描述了这个过程。

中断可能引发任务切换。当一个进程被中断时，该进程的进程表项中将建立一个堆栈，再次启动该进程所需的全部信息都放在这个新堆栈上。通过以下操作可以再次启动任何一个进程：将栈指针指向其进程表项，然后执行一个指令序列来恢复 CPU 寄存器值，最后以一条 `iretd` 指令结束。调度器决定堆栈指向哪个进程表项。

内核本身在运行时也可能发生中断。CPU 检测到中断后将使用内核栈，而不是进程表中的堆栈。这样便允许中断嵌套。当后来的中断服务例程结束后，在它之前执行的中断服务例程就可以一直运行到结束。在所有中断都被处理之后，则可以重启一个进程。

MINIX 3 调度算法使用多优先级队列。通常情况下系统进程运行在最高优先级队列，而用户进程在最低优先级队列，但是优先级是按照一个进程接一个进程来安排的，一个循环连续运行的进程的优先级可能会临时降低，当一个进程有机会运行时，其他进程的优先级会被保存。可以使用 `nice` 命令改变那些有定义限制的进程的优先级。进程之间轮询运行以获得每秒改变多次的时间片。然而，一个进程被阻塞后一直到其重新就绪，它将按照其未使用的时间片被放在队列头部。这是为了对于 I/O 操作给予快速响应。尽管如此，如果系统进程运行时间过长，那么它们也可被抢占。

内核镜像包含了系统任务，它使得用户空间进程同内核的通信更加容易。它通过执行代表服务器或者驱动程序的私有操作来支持它们。在 MINIX 3 中，时钟任务也被编译进了内核。它不是普通意义上的设备驱动程序。用户空间进程不能作为一个设备去访问时钟。

习题

1. 现代操作系统为何都是多进程的？
2. 进程的三种状态各是什么？分别简要描述。
3. 假设你正在设计一种先进的计算机体系结构，它使用硬件而不是中断来完成进程切换，则 CPU 需要哪些信息？请描述用硬件完成进程切换的工作过程。
4. 目前的计算机上，中断处理程序至少有一小部分用汇编语言编写，为什么？
5. 重画图 2.2，添加两个状态：新建和终结。进程创建时被初始化为“新建”状态，退出时为“终结”状态。
6. 书中认为图 2.6(a)的模型不适用于在内存中进行高速缓存的文件服务器，为什么？可否使每个进程拥有自己的高速缓存？
7. 进程与线程的本质区别是什么？
8. 在使用线程的系统中，是每个线程有一个堆栈还是每个进程有一个堆栈？说明原因。
9. 什么是竞争条件？

10. 举一个一起出行的两人买飞机票时可能发生的竞争条件的例子。
11. 写一个 shell 程序，其功能是产生一个内容为一个整数序列的文件。要求它先读取文件中的最后一个整数，将其加1，然后将这个新整数追加到文件的末尾。在系统前台和后台同时运行它并使用相同的文件名。在因竞争而造成的故障发生之前它能运行多久？此处的临界区是什么？请对其进行修改以防止发生竞争。提示：使用 `In file file.lock` 来将数据文件加锁。
12. 对于前一习题，语句
`In file file.lock`
是一种很有效的加锁机制吗？说明原因。
13. 两个进程在一台共享内存（即共享同一个存储器）的多处理机上运行时，图 2.10 所示的采用变量 *turn* 的忙等待方案还奏效吗？
14. 现有一台计算机，它没有 TEST AND SET LOCK 指令，但有一条指令可以按原子操作方式将一个寄存器的值和一个存储器字进行交换。能否利用该指令写一个与图 2.12 中 *enter_region* 类似的例程？
15. 给出一个框架，来描述一个可禁用中断的操作系统如何实现信号量。
16. 请说明如何只利用二进制信号量和普通的机器指令来实现计数信号量（即可以拥有任意大的数值的信号量）。
17. 在 2.2.4 节中描述了一个高优先级进程 H 和低优先级进程 L 的情况。它最终导致 H 陷入死循环。若采用时间片调度而不是优先级调度，还能发生这种情况吗？请进行讨论。
18. 管程内部的同步机制使用条件变量和两个特殊操作 WAIT 和 SIGNAL，一种更一般的同步方式是只有一条原语 WAITUNTIL，它以任意的布尔表达式作为参数。例如 `WAITUNTIL x < 0` 或 `y + z < n`，这样便不再需要 SIGNAL 原语。很显然这个方案比 Hoare 和 Brinch Hansen 的方案更通用，但它从未被采用过，为什么？提示：从实现考虑。
19. 一个快餐厅有四种职员：(1)领班，他们负责接收顾客点的菜单；(2)厨师，他们负责准备饭菜；(3)打包工，他们负责将饭菜装在袋子里；(4)出纳员，他们负责收钱并将食品袋交给顾客。每个职员可被视为一个进行通信的串行进程，他们采用的进程间通信方式是什么？将该模型与 MINIX 3 中的进程加以比较。
20. 假设有一消息传递系统使用信箱。当向满信箱发消息或从空信箱收消息时，一个进程不会阻塞。相反，它会获得一条返回的错误码。该进程对错误码的处理方式是重试，直到成功为止。这种方案会带来竞争吗？
21. 在图 2.20 所示的哲学家进餐问题的解法中，为什么过程 *take_forks* 将状态变量置为 *HUNGRY*？
22. 考虑图 2.20 中的过程 *put_forks*。假设变量 *state[i]* 在对 *test* 的两次调用之后被置为 *THINKING*，而不是在调用之前。对于 3 位哲学家的情况，这个改动有什么影响？对于 100 位哲学家的情况呢？
23. 从何种类型的进程可以在何时被启动的角度来看，读者-写者问题可以通过几种方式进行形式化。根据优先哪几类进程的不同，请详细地描述该问题的三种变体。对每种变体，请说明当一个读者或写者能够访问数据库时情况将会怎样，以及当一个进程对数据库访问结束后又将会怎样。
24. CDC 6600 计算机使用一种有趣的称为处理器共享的时间片调度算法，它可以同时处理多达 10 个 I/O 进程，每条指令结束后都进行 **进程切换**，这样进程 1 执行指令 1，进程 2 执行指令 2，等等。进程切换由特殊的硬件完成，所以没有系统开销。如果在没有竞争的条件下一个进程需要 *T* 秒钟才能完成，那么当有 *n* 个进程共享处理器时需要多长时间才能完成？

25. 时间片调度器通常维护一个由所有就绪进程组成的队列，每个进程在队列中出现一次。如果一个进程在队列中出现两次以上，情况将会怎样？你能设想出这种情况出现的原因吗？
26. 对某系统进行监测后表明平均每个进程在I/O阻塞之前的运行时间为 T 。一次进程切换需要的时间为 S ，这里 S 实际上为开销。对于采用时间片长度为 Q 的时间片调度法，对以下各种情况给出CPU利用率的计算公式：
- (a) $Q = \infty$
 - (b) $Q > T$
 - (c) $S < Q < T$
 - (d) $Q = S$
 - (e) Q 趋近于0
27. 有5个待运行任务，各自的预计运行时间分别是9, 6, 3, 5和 X 。采用哪种运行次序将使平均响应时间最短？（答案取决于 X 。）
28. 有5个批处理任务A到E几乎同时到达一个计算中心。其预计运行时间为10 min, 6 min, 2 min, 4 min 和 8 min。其优先级（由外部设定）分别为3, 5, 2, 1 和 4，这里5为最高优先级。对于下列每种调度算法，计算其平均进程周转时间，进程切换开销可忽略：
- (a) 时间片轮转
 - (b) 优先级调度
 - (c) 先来先服务（按照次序 10, 6, 2, 4, 8）
 - (d) 最短作业优先
- 对于(a)，假设系统具有多道处理能力，每个作业均获得公平的CPU份额；对于(b)到(d)，假设某一时刻只有一个作业运行，直到结束。所有的作业都是完全的CPU密集型作业。
29. 在CTSS上运行的一个进程需要30个时间片方能结束，那么它需要被换入多少次？包括第一次，即开始运行之前。
30. 使用一个参数 $a = 1/2$ 的老化算法来预测运行时间。从最早到最近的前4次执行时间为40 ms, 20 ms, 40 ms 和 15 ms，则下次运行时间预计为多长？
31. 由图2.25可以看到批处理系统中的三层调度算法，这种方法能否用到无新到达任务的交互式系统中呢？怎么做？
32. 假设图2.28(a)中的线程按照下面的顺序执行：A的一条，B的一条，A的一条，B的一条，依次类推。请问开始的四次调度顺序有多少种可能？
33. 一个软实时系统有4个周期性事件，其周期分别为50 ms, 100 ms, 300 ms 和 250 ms。假设其处理分别需要35 ms, 20 ms, 10 ms 和 x ms，则该系统可调度所允许的 x 值最大是多少？
34. MINIX 3在执行期间维护一个变量 $proc_ptr$ ，它指向当前进程的进程表项。为什么？
35. MINIX 3对消息不加缓冲。请解释这样的设计会对时钟和键盘中断带来什么问题？
36. 在MINIX 3中当一条消息被发送给一个睡眠进程时，将调用过程 $ready$ 来将该进程挂入适当的调度队列中。该过程首先要关中断。为什么？
37. MINIX 3中的 $mini_rec$ 过程包含一个循环。请解释为什么需要该循环。
38. MINIX 3使用图2.43所示的调度方法，其中不同类型的进程有不同的优先级。优先级最低的进程（用户进程）使用时间片调度法，而系统任务和服务器进程则允许一直运行到阻塞。请问优先级最低的进程是否会发生饥饿？为什么？
39. MINIX 3适用于实时应用吗？例如数据日志记录。若不适用，可对其做什么改动？

40. 假设有一个提供信号量的操作系统。请实现一个消息传递系统，写出发送和接收消息的过程。
41. 一个主修人类学、辅修计算机科学的学生参加了一个课题，调查非洲狒狒是否能被教会理解死锁。他找到一处很深的峡谷，在上边固定了一根横跨峡谷的绳索，这样狒狒就可以攀住绳索越过峡谷。同一时刻可以有几只狒狒通过，只要它们朝着相同的方向。但如果向东和向西的狒狒同时攀在绳索上，则将发生死锁（狒狒将被卡在中间），因为它们无法在绳索上从另一只的背上翻过去。如果一只狒狒想越过峡谷，它必须看当前是否有其他的狒狒正在逆向通过。使用信号量写一个避免死锁的程序来解决该问题。
42. 继续前一习题，但现在要避免饥饿。当一只想向东去的狒狒到了绳索跟前，但发现有其他的狒狒正在向西越过峡谷时，它将一直等到绳索可用为止。但在至少有一只狒狒向东越过峡谷之前，不允许再有狒狒开始从东向西越过峡谷。
43. 使用管程而不是信号量来解决哲学家进餐问题。
44. 在 MINIX 3 的内核中增加一些代码来跟踪从进程（任务） i 发送到进程（任务） j 的消息个数。按下 F4 键时打印出该矩阵。
45. 修改 MINIX 3 的调度器以跟踪每个用户进程最近使用的 CPU 时间。在无任务或服务器进程运行时，要求选择运行使用 CPU 最少的用户进程。
46. 修改 MINIX 3，使每一个进程可以显式地通过一个带有 pid 和 $priority$ 参数的新系统调用 `setpriority` 来设置它的子进程的调度优先级。
47. 修改 `mpx386.s` 中的宏 `hwint_master` 和 `hwint_slave`，使得当前由 `save` 函数执行的操作改变由内联代码执行。这样做使代码增大了多少？你能测出性能提高了多少吗？
48. 请解释在你的 MINIX 3 系统上 `sysenv` 命令所带的各项参数。如果无法接触到运行中的 MINIX 3 系统，那就解释图 2.37 中的各项。
49. 在讨论进程表的初始化时我们曾提到过，如果在数组中而不是在索引中添加一个常量，那么有些 C 编译器能产生较好的代码。请写一对 C 程序以检验这种假说。
50. 修改 MINIX 3 来收集端到端发送的消息统计，写一个有效的程序收集并打印该统计信息。

第3章 输入/输出系统

- 3.1 I/O硬件原理
- 3.2 I/O软件的原理
- 3.3 死锁
- 3.4 MINIX 3中的I/O概述
- 3.5 MINIX 3中的块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端
- 3.9 小结

操作系统的主要功能之一是控制所有的输入/输出 (Input/Output, 简称 I/O) 设备。它必须向设备发出命令，捕获中断并进行错误处理，它还要提供一个设备与系统其余部分之间的简单易用的接口。该接口应对所有设备尽可能地一致 (设备无关性)。I/O 部分的代码占据了整个操作系统的相当部分。本章研究操作系统如何管理输入 / 输出。

本章内容安排如下：首先介绍 I/O 硬件的基本原理，然后概述 I/O 软件。I/O 软件可以分为若干层次，每一层都有一个定义良好的接口。这其中的每一层都将得到讨论，以了解各层的功能以及它们是如何结合在一起的。

随后讨论死锁。我们将给死锁下一个准确的定义，讲述死锁产生的原因，提供对死锁进行分析的两种模型，并讨论几种死锁预防的算法。

接下来简要地浏览一下 MINIX 3 中的 I/O，包括中断、设备驱动程序、设备相关 I/O 和设备无关 I/O。之后，我们将详细探讨几个 I/O 设备：磁盘、键盘和显示器。对于每种设备，我们将探讨其硬件和软件。

3.1 I/O 硬件原理

不同的人对 I/O 硬件有不同的理解。在电气工程师看来，I/O 硬件就是一堆芯片、电线、电源、马达和其他组成硬件的物理部件。对程序员而言，主要注意 I/O 硬件提供给软件的接口，如硬件能够接收的命令、能够完成的功能以及能报告的各种错误等。本书主要关注 I/O 设备的编程，而不是硬件的设计、制造和维护，因此重点放在如何对硬件进行编程上，而不是其内部的工作原理。然而，许多 I/O 设备的编程常常与其内部操作密切相关。以下三节将介绍与 I/O 设备编程有关的一般性背景知识。

3.1.1 I/O 设备

I/O 设备大致可以分为两类：**块设备** (block device) 和**字符设备** (character device)。块设备将信息存储在固定大小的块中，每个块都有自己的地址。数据块的大小通常在 512 字节到 32 768 字节之间。块设备的基本特征是每个块都能够独立于其他块而读写。磁盘是最常见的块设备。

如果仔细观察，那么会发现按块编址的设备和不按块编址的设备之间没有明确的界限。磁盘是公认的按块编址的设备，因为无论磁盘臂当前处于什么位置，总可以寻址到其他柱面并等待所需要的数据块旋转到磁头下面。现在考虑一个用来对磁盘进行备份的磁带机。磁带包含按顺序排列的块。如果命令磁带机读第 N 个数据块，那么它总能通过倒带和前进来定位到所需要的数据块。该操作和磁盘的寻址类似，只是花费的时间更长。不过，重写磁带中间位置的某个数据块可能做得到，也可能做不到。把磁带用做随机存取的块设备是有可能的，但会勉为其难，因为我们通常并不这样使用磁带。

另一类 I/O 设备是字符设备。字符设备发送或接收的是字符流，而不考虑任何块结构。字符设备无法编址，也不存在任何寻址操作。打印机、网络接口、鼠标（用做指点设备）、老鼠（用于心理学实验室实验）以及大多数与磁盘不同的设备均可被视为字符设备。

这种分类方法并不完美。有些设备就符合这种分类。例如，时钟是无法寻址的，也不产生或接收字符流。时钟的全部功能就是按照预先定义的时间间隔发出中断。

但是，块设备和字符设备的模型具有足够的一般性，可以作为使处理不同I/O设备的操作系统软件具有设备无关性的基础。例如，文件系统仅仅控制抽象的块设备，而把与设备有关的部分留给较低层软件，即设备驱动程序（device driver）去处理。

I/O设备在速度上覆盖了巨大的范围，这给软件在数据率上跨越这么多量级的情况下保持性能良好造成了很大的压力。图3.1列出了某些常见设备的数据率，这些设备中的大多数倾向于随着时间的推移变得越来越快。

设备	数据率
键盘	10 B/s
鼠标	100 B/s
56K调制解调器	7 KB/s
扫描仪	400 KB/s
数字便携式摄像机	4 MB/s
52倍速CD-ROM	8 MB/s
火线(IEEE 1394)	50 MB/s
USB 2.0	60 MB/s
XGA显示器	60 MB/s
SONET OC-12网络	78 MB/s
千兆以太网	125 MB/s
串行ATA磁盘	200 MB/s
SCSI超宽4磁盘	320 MB/s
PCI总线	528 GB/s

图3.1 某些典型设备、网络和总线的数据率

3.1.2 设备控制器

I/O设备通常由一个机械部件和一个电子部件组成。为了提供更加模块化和更加通用的设计，一般可以将这两部分分开。电子部件称为设备控制器（device controller）或适配器（adapter）。在个人计算机中，它常常是一块可以插入主板扩展槽的印制电路板。机械部件则是设备本身，参见图3.2。

控制器卡上一般都有一个连接器，与设备本身相连的电缆可以插入这个连接器。许多控制器可以控制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标准接口，如ANSI, IEEE, ISO或者事实上的标准，那么各公司就可以制造与该接口匹配的控制器和设备。例如许多公司生产符合IDE（集成设备电子器件）接口或SCSI（小型计算机系统接口）接口的硬盘驱动器。

之所以区分控制器和设备本身，是因为操作系统大多与控制器打交道，而非设备本身。大多数个人计算机服务器的采用如图3.2所示的总线模型来进行CPU和控制器之间的通信。大型机通常采用其他模型，这种模型带有被称为I/O通道的专门用于I/O的计算机，它能够减轻主CPU的部分工作负担。

控制器与设备之间的接口通常是一种很低层次的接口。例如，磁盘可以按每个磁道1024个扇区、每个扇区512字节进行格式化。然而，实际从驱动器读出来的是一串比特流，它以一个前导符（preamble）开始，随后是一个扇区的4096比特，最后是一个校验和，也称为纠错码（Error-Correcting Code, ECC）。前导符是磁盘格式化时写进去的，它包括柱面数和扇区数、扇区大小之类的数据，还包含同步信息。

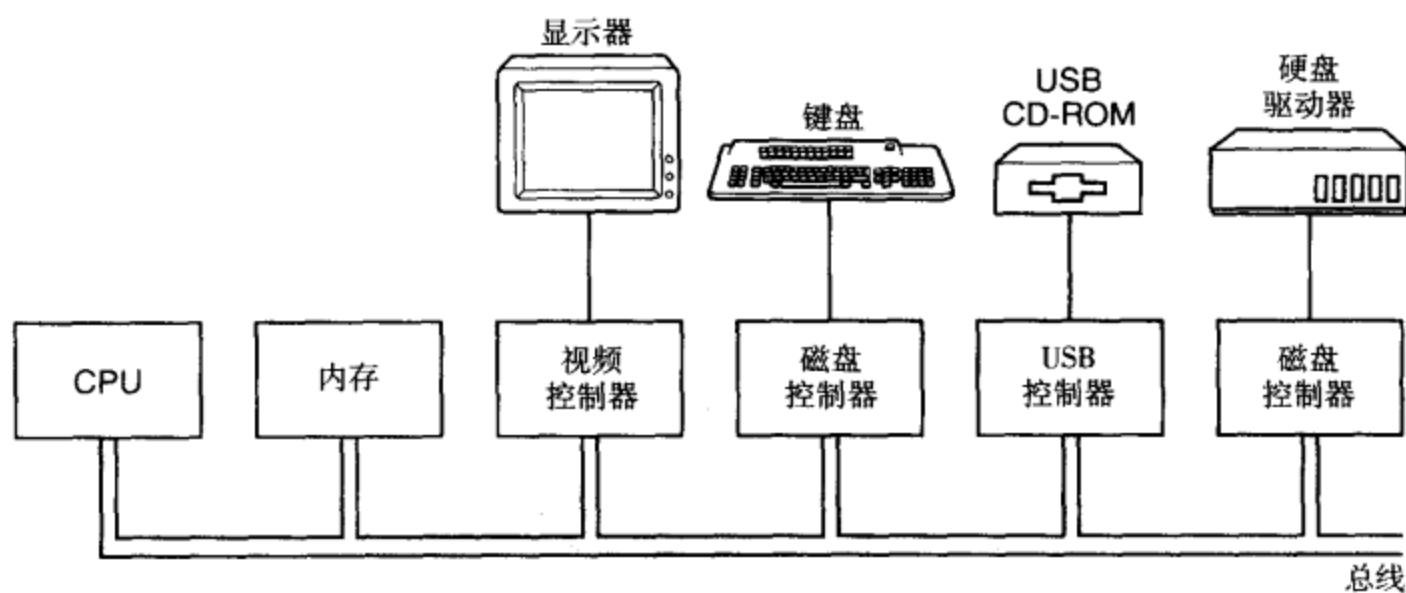


图 3.2 CPU、内存、控制器和 I/O 设备的连接模型

控制器的任务是将这个串行的比特流转换成字节块，并完成必要的纠错工作。通常，字节块是在控制器的缓冲区中逐个比特汇集而成的。在对校验和进行校验并证明数据块没有错误后，字节块将被复制到主存中。

在同样低的层次上，CRT 终端控制器也是一个比特串行设备。它从内存中读取欲包含待显示字符的字节流，然后产生用来调制 CRT 电子束的信号。控制器还产生使 CRT 电子束在完成一次行扫描后做水平回扫的信号，并且产生使 CRT 电子束在当整个屏幕扫描结束后的垂直回扫的信号。在 LCD 屏上，这些信号选择各个像素并控制它们的亮度，模拟 CRT 中电子束的效果。如果没有视频控制器，那么操作系统程序员只能直接对扫描编程。有了控制器，操作系统只需通过一些参数，如每行的字符数、每行的像数和每屏的行数等，对控制器进行初始化，并让控制器实际驱动显示。

某些设备（尤其是磁盘）的控制器正变得相当复杂。例如，现代磁盘控制器内部的存储器容量通常有几 MB。因此，在执行读取操作时，只要磁头臂到达了正确的柱面，控制器就开始读取并存储数据，即使磁头臂还未到达正确的扇区。这种缓存的数据对于满足后续请求是相当便利的。此外，即使获得了请求的数据，控制器也可以继续缓存来自后续扇区的数据，因为这些数据以后可能需要。按这种方式可处理许多磁盘读取操作，而根本不需要磁活动。

3.1.3 内存映射 I/O

每个控制器都有一些用来与 CPU 通信的寄存器。通过写入这些寄存器，操作系统可以命令设备发送数据、接收数据、开启或关闭，或者执行某些其他操作。通过读取这些寄存器，操作系统可以了解设备的状态、是否准备好接受一个新的命令等。

除了这些控制寄存器以外，许多设备还有一个操作系统可以读写的数据缓冲区。例如，在屏幕上显示像素的常见方法是使用一个视频 RAM，这个视频 RAM 基本上只是一个数据缓冲区，可供程序或操作系统写入数据。

于是问题出现了：CPU 如何与控制寄存器和设备的数据缓冲区进行通信？存在两种可选的方法。在第一种方法中，每个控制寄存器被分配一个 I/O 端口（I/O port）号，这是一个 8 位或 16 位的整数。使用一条专门的 I/O 指令，例如

```
IN REG, PORT
```

CPU 可以读取控制寄存器 PORT 的内容并将结果存入到 CPU 寄存器 REG 中。与此类似，使用

```
OUT PORT, REG
```

CPU可以将REG的内容写入到控制寄存器中。大多数早期的计算机，包括几乎所有的主机，如IBM 360及其所有后续机型，都是以这种方式工作的。

在这种方案中，内存和I/O的地址空间是不同的，如图3.3(a)所示。

在其他一些计算机上，I/O寄存器是内存地址空间的一部分，如图3.3(b)所示，这种方案被称为内存映射I/O，这是由PDP-11小型机引入的。每个控制寄存器被分配唯一的一个内存地址，并且不会有内存分配这一地址。通常分配给控制寄存器的地址位于地址空间的顶端。图3.3(c)描述的是一种混合方案，这一方案具有内在映射I/O的数据缓冲区和控制器专用的I/O端口。奔腾处理器使用的就是这一体系结构。在IBM PC兼容机中，除了0到64 KB的I/O端口之外，640 KB到1 MB的地址保留给设备的数据缓冲区。

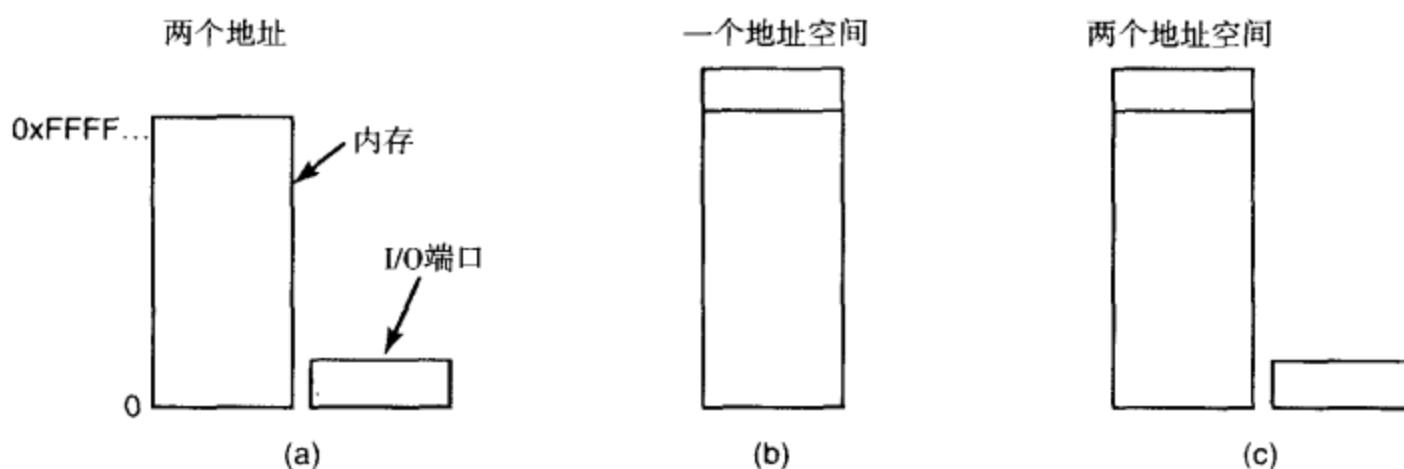


图3.3 (a)单独的I/O和内存空间；(b)内存映射I/O；(c)混合方案

这些方案是怎样工作的呢？在各种情况下，当CPU想要读入一个字时，不论是从内存中读入还是从I/O端口中读入，它都要将需要的地址放到总线的地址线上，然后在总线的一条控制线上设置读信号。还要用到第二条信号线来表明需要的是I/O空间还是内存空间。如果是内存空间，内存将响应请求。如果是I/O空间，I/O设备将响应请求。如果只有内存空间[如图3.3(b)所示]，那么每个内存模块和每个I/O设备都会将地址线和它所服务的地址范围进行比较，如果地址落在这一范围内，它就会响应请求。因为绝对不会有地址既分配给内存又分配给I/O设备的情形，所以不会存在歧义和冲突。

3.1.4 中断

通常，控制寄存器有一个或多个状态标志位，这些标志位可以被测试以确定输出操作是否完成或者输入设备中是否有新数据可用。CPU可以执行一个循环，一直测试状态标志位，直到设备准备好接收或提供新数据。这种方法称为轮询检测（polling）或者忙等待（busy waiting）。这个概念在2.2.3节中作为一种处理临界区域的方法出现过，在那种情况下，它在大多数环境下应避免使用。在I/O领域，为了等待外界接收或产生数据可能不得不等待很长时间，轮询检测是不可接受的，除非是针对不是运行多进程的很小的专用系统。

除了状态标志位，许多控制器还可以在读完或写完它们的寄存器时，使用中断通知CPU。CPU怎样处理中断已经在2.1.6节中讨论过。在I/O环境中，所有需要知道的就是大多数接口设备提供一个输出，这个输出逻辑上和寄存器的“操作完成”或者“数据就绪”的状态标志位是一样的，但这个输出将用于驱动一条系统总线的IRQ（Interrupt ReQuest）线。因此，当一个启用的中断操作完成时，它将中断CPU并开始运行中断处理器。这段代码通知操作系统I/O已经完成，然后操作系统将检查状态标志位以确定一切正常，并获取结果数据或者开始重试。

中断控制器的数量是有限的，奔腾系列的PC机只有15条中断供I/O设备使用。有些控制器直接制作在系统的主板上，如IBM PC机的磁盘和键盘控制器。在一些比较老的系统上，设备使用的

IRQ 通过和控制器关联的开关或跳线来设置。如果用户买了一个新的插件板，将不得不手动设定 IRQ 以避免和已存在的 IRQ 冲突。用户很少能正确完成，这就导致了工业界发展了即插即用(Plug'n Play) 技术，它可以使 BIOS 在启动时为设备自动分配 IRQ 以避免冲突。

3.1.5 直接存储器存取

不管一个系统是否有内存映射 I/O，它的 CPU 都需要寻址设备控制器以便与它们交换数据。CPU 可以从 I/O 控制器每次请求一个字节的数据，但是对于像磁盘这样能产生大块数据的设备，这样做会浪费 CPU 的时间，所以经常采用一种被称为直接存储器存取(Direct Memory Access, DMA) 的不同方案。只有硬件具有 DMA 控制器时操作系统才能使用 DMA，而大多数系统都有 DMA 控制器。有时 DMA 控制器集成到磁盘控制器和其他控制器中，但是对于这样的设计，每个设备都需要一个单独的 DMA 控制器。更普遍的是，只有一个 DMA 控制器可用(例如在主板上)，由它调控多个设备的数据传送，而这些数据传送经常是同时发生的。

不管 DMA 控制器物理上处于什么地方，它都独立于 CPU 访问系统总线，如图 3.4 所示。它包含一些可以被 CPU 读写的寄存器。控制寄存器指定使用的 I/O 端口、传输方向(从 I/O 设备读或者向 I/O 设备写)、传输单位(每次一个字节或每次一个字)以及在一次突发传送中要传送的字节数。

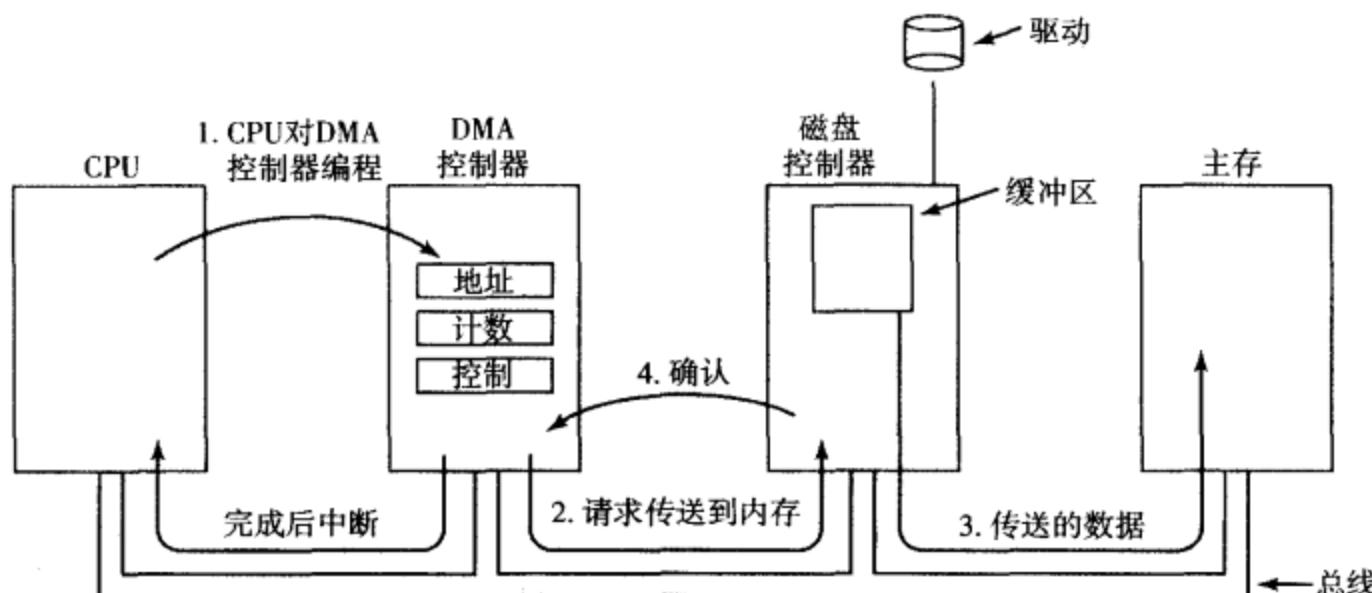


图 3.4 DMA 传送操作

为了解释 DMA 是怎样工作的，我们首先看一下未使用 DMA 时磁盘读操作是怎样发生的。首先控制器从驱动器串行地按位读取一个块(一个或多个扇区)，直到整个块被读入控制器的内部缓冲区。接着，控制器计算校验和以确保没有读错误发生。然后控制器产生一个中断。当操作系统开始运行时，通过执行一个循环，它能从控制器的缓冲区每次一个字节或一个字地读取磁盘块，每次循环反复地从控制器设备寄存器读取一个字节或一个字，将其存进内存，增加内存地址，减少要读的数据个数，直到减为零为止。

使用 DMA 时，过程是不同的。首先，CPU 通过设置 DMA 控制器的寄存器对它进行编程，所以 DMA 控制器知道将数据传送到什么地方(图 3.4 中的第 1 步)。DMA 控制器还向磁盘控制器发出命令，告诉它从磁盘读取数据到其内部缓冲区并验证校验和。当有效的数据在磁盘控制器的缓冲区里时，DMA 就可以开始了。

DMA 控制器通过在总线上发出一个读请求到磁盘控制器来开始传送(第 2 步)。这个读请求看起来和其他的读请求是一样的，而且磁盘控制器不知道也不关心它是来自 CPU 还是 DMA 控制器。在典型情况下，要写的内存地址在总线的地址线上，所以当磁盘控制器从它的内部缓冲区取下一个

字时，它知道把这个字写到什么地方。写到内存是另外一个标准总线周期（第3步）。当写操作完成时，磁盘控制器向DMA控制器发出一个应答信号，这也是通过总线完成的（第4步）。DMA增加要使用的内存地址，减少字节计数。如果字节计数仍然大于零，就重复第2步到第4步，直到计数达到零。这时，DMA控制器将产生中断。当操作系统启动时，不需要再复制数据块到内存，因为它已经在那了。

也许读者想知道为什么控制器从磁盘读入字节后不立即将其存储在主存。换句话说，为什么它需要一个内部缓冲区？有两个原因。首先，通过内部缓冲，磁盘控制器能在开始传送之前验证校验和。如果校验和是错误的，将发出一个错误的信号并且不传送到内存。

第二个原因是，一旦磁盘传输开始，从磁盘读出的位流以恒定的速率到达，而不管控制器是否已准备好接收它们。如果控制器试图将数据直接写到内存，那么它必须为要传送的每个字获取系统总线的控制权。如果由于其他设备使用总线而导致总线繁忙，则控制器只能等待。如果下一个磁盘字（disk word）在前一个还没有存储前到达，则控制器只能把它存在某个地方。如果总线非常忙，则控制器可能需要存储很多字，而且还有大量的管理工作。如果块被放入内部缓冲区，则在DMA开始前不需要使用总线，这样控制器的设计就可以更加简化，因为对DMA到内存的传送没有严格的时间要求。

并不是所有的计算机都使用DMA。反对的根据是主CPU通常要比DMA控制器快得多，并且能更快地完成工作（当限制因素不是I/O设备的速度时）。如果CPU没有其他工作要做，那么让（快速的）CPU等待（慢速的）DMA控制器完成工作是无意义的。而且，去除DMA控制器而使CPU用软件做所有的工作可以省钱，这一点在低端（嵌入式）计算机上十分重要。

3.2 I/O 软件的原理

在讨论了I/O硬件之后，现在看一下I/O软件。首先将看一下I/O软件的目标，然后再从操作系统的角度看一下I/O的不同实现方法。

3.2.1 I/O 软件的目标

I/O软件设计的一个关键概念是设备无关性，其含义就是应该能够写出可以访问任意I/O而不需要事先指定设备的程序。例如，读取一个文件作为输入的程序，无须为每种不同的设备进行修改就能在软盘、硬盘以及CD-ROM上读取文件。类似地，用户应该能够输入如下命令：

```
sort <input> output
```

并且无论输入来自软盘、IDE硬盘、SCSI硬盘或者键盘，输出被送到任意类型的磁盘或者屏幕，上述命令都可以工作。尽管这些设备实际上差别很大，需要非常不同的命令序列来读或写，但这一事实所带来的问题将由操作系统负责处理。

与设备无关性紧密相关的是统一命名法（uniform naming）这一目标。文件或设备名应该很简单，是一个字符串或一个整数，且完全不依赖于设备。在UNIX和MINIX 3中，所有的磁盘可以以任何方式集成到文件系统层次结构中去，因此用户也不必知道哪个名字对应哪个设备。例如，软盘可以被挂载（mount）到目录/usr/ast/backup下，这时复制一个文件至/usr/ast/backup就是将把文件复制到软盘上。使用这种方式，所有的文件和设备都使用相同的方式寻址——通过路径名。

I/O软件的另一个重要问题是错误处理。一般来说，错误应该在尽可能接近硬件的地方得到处理。在控制器发现了一个读错误时，若它能够处理，则它就应该尽量纠正这个错误。如果控制器

处理不了，则设备驱动程序应该处理，这时可能只需要重读一次数据块就可以解决问题。许多错误是暂时性的，如磁头被灰尘阻滞导致的读错误，只需重复一次操作便可以消除。只有在低层软件处理不了的情况下才通知高层软件。在许多情况下，错误恢复可以在低层透明地得到解决，而高层软件甚至不知道存在这一错误。

另外一个关键问题是同步（阻塞）与异步（中断驱动）传输。大多数物理 I/O 是异步传输，即 CPU 在启动传输后便转向其他工作，直到中断到达。如果 I/O 操作是阻塞的，那么用户程序就更容易编写——在 `Receive` 系统调用后，程序将自动挂起，直到缓冲区中的数据准备好。使实际上是中断驱动的操作变为在用户程序看来是阻塞的操作，这应该是操作系统的工作。

I/O 软件的另一个问题是缓冲（buffering）。通常数据离开设备后不能直接存放到最终目的地。例如，当从网络来一个数据包时，在把该数据包存储在某个地方并检验以后，操作系统才知道要把它放在什么地方。而且一些设备具有严格的实时约束（例如，数字音频设备），所以数据必须事先存放到输出缓冲区中，从而消除缓冲区填满速率和缓冲区清空速率之间的相互影响，以避免缓冲区欠载。缓冲涉及大量的复制工作，并且通常对 I/O 性能有重大影响。

这里将提到的最后一个概念是共享设备和独占设备的问题。某些设备可同时被多个用户使用，如磁盘。多个用户在同一磁盘上同时打开多个文件是没有问题的。其他设备则只能供一个用户专用，只有当前用户使用完毕后，其他用户才能使用，如磁带机。让两个或更多用户随机地将交叉混杂的数据块写入相同的磁带注定是不能工作的。独占（非共享）设备的引入也带来了各种问题，如死锁。同样，操作系统必须能以一种避免问题发生的方法来处理共享设备和独占设备。

I/O 软件通常被组织为四个层次，如图 3.5 所示。在接下来的部分我们将依次讨论每一层，从底层开始。本章的重点在设备驱动（第 2 层）上，但是下面将概述 I/O 软件的其余部分以说明 I/O 系统的各层是怎样结合到一起的。

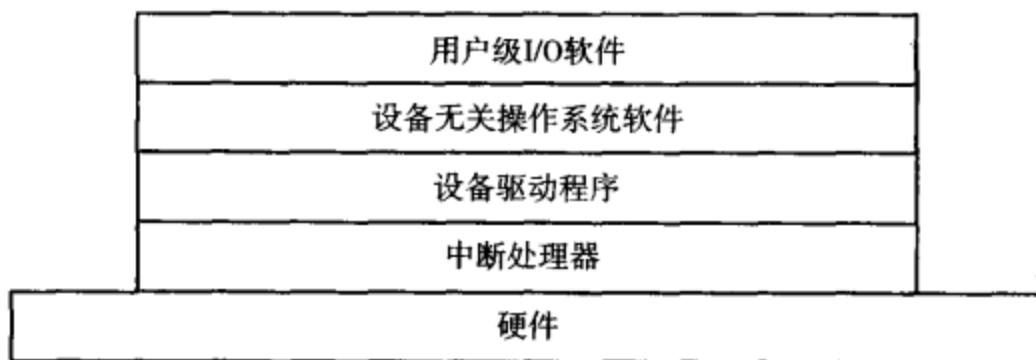


图 3.5 I/O 软件系统的层次

3.2.2 中断处理器

中断是令人不愉快的事情，但是无法避免，应当将其深深隐藏在操作系统内部，以便操作系统中尽可能少的部分知道它们。隐藏它们的最好方法就是让启动 I/O 的驱动程序阻塞起来，直到 I/O 操作完成且产生一个中断。驱动程序可以通过在一个信号量上执行 `down` 操作、在一个条件变量上执行 `wait` 操作、在一条消息上执行 `receive` 操作或者某些类似的操作来使自己阻塞。

当中断发生时，中断处理器将做它必须要做的全部工作以便对中断进行处理。然后，它可以将启动中断的驱动程序解除阻塞。在一些情况下，它只是在一个信号量上执行 `up` 操作。其他情况下，它将对管程中的一个条件变量执行 `signal` 操作，还有一些情况，是向被阻塞的驱动程序发一条消息。在所有这些情况下，中断的最终结果是使先前阻塞的驱动程序现在能够运行。如果驱动程序被构造为独立进程，有它们自己的状态、堆栈和程序计数器，那么这个模型会运转得最好。

3.2.3 设备驱动程序

本章前面的内容提到每个设备控制器都有用于给它命令的寄存器，或者读出它的状态的寄存器，或者二者皆有。寄存器的数量和命令的性质在不同设备之间有着根本性的不同。例如，鼠标驱动程序必须从鼠标接收信息，以识别鼠标移动了多远的距离以及当前哪一个键被按下。相反，磁盘驱动程序则必须知道扇区、磁道、柱面、磁头、磁盘臂移动、电动驱动器、磁头定位时间及所有其他保证磁盘正常工作的机制。显然，这些驱动程序是有很大区别的。

因此，每个连接到计算机上的I/O设备需要一些设备特定的代码来控制它。这样的代码被称为**设备驱动程序**，它一般由设备的制造商编写并刻录在CD-ROM上和设备一起交付。因为每个操作系统都需要自己的驱动程序，所以设备制造商通常要为一些流行的操作系统提供驱动程序。

每个设备驱动程序通常只处理一种类型的设备，或者一类紧密相关的设备。比如，即使系统支持几个不同品牌的鼠标，而只有一个鼠标驱动程序也可能是一个好主意。另外一个例子是，一个磁盘驱动程序通常可以处理多个不同大小、不同速度的磁盘，也可能处理CD-ROM。另一方面，鼠标和硬盘是如此不同，以至于不同的驱动程序是必需的。

为了访问设备硬件（意味着访问控制器的寄存器），驱动程序传统上是系统内核的一部分。这种方法提供了最好的性能和最坏的可靠性，因为任何设备驱动程序的一个缺陷都可能导致整个系统崩溃。MINIX 3为了提高可靠性没有采用这个模型。正如我们将要看到的，现在MINIX 3中的每个设备驱动程序是一个单独的用户模式的进程。

正如前面提到的，操作系统通常把驱动器分为**块设备**（比如磁盘）和**字符设备**（比如键盘和打印机）。大多数操作系统都定义了一个所有块设备都必须支持的标准接口，并且还定义了另一个所有字符设备都必须支持的标准接口。这些接口由许多过程（procedures）组成，操作系统的其余部分可以调用它们让驱动程序为它工作。

概括地说，设备驱动程序的工作是接受一个来自其上方与设备无关的软件的抽象请求，并负责执行这个请求。对于磁盘驱动程序，一个典型的应用是读取数据块 n 。如果在请求到来时驱动器是空闲的，那么它将立即执行请求。然而，如果它在处理一个请求，那么它通常会将新的请求加入未处理的请求的队列，以便尽可能快地处理。

实际上，执行I/O请求的第一步是检查输入参数是否有效并且如果无效就返回一个错误。如果请求是有效的，那么下一步是把抽象值转化为具体形式。对于磁盘驱动程序来说，这意味着计算出请求的块于磁盘上实际在什么地方，检查驱动器的电机是否运行，确定磁盘臂是否定位在合适的柱面，等等。总之，驱动程序必须决定需要什么控制器操作，按什么顺序。

一旦驱动程序决定向控制器发出某个命令，它就开始通过写入控制器的设备寄存器来发送命令。简单的控制器一次只能处理一个命令。更复杂的控制器能够接受一串命令，然后控制器自己就能执行这些命令，而不再需要来自操作系统的进一步帮助。

命令发出后，会出现两种情况中的一种。在多数情况下，设备驱动程序必须等待，直到控制器为其做某些事情，所以驱动程序将阻塞其自身直到中断来解除阻塞。然而，在另外一些情况下，操作可以无延迟地完成，所以驱动程序不需要阻塞。在一些图形卡上滚动屏幕只需要写少许字节到控制器的寄存器中，由于不需要机械运行，所以整个操作能在几微秒内完成，这便是后一种情形的例子。

在前一种情况下，被阻塞的驱动程序将被中断唤醒。在后一种情况下，驱动程序根本就不会睡眠。不管是哪种情况，操作完成之后驱动程序都必须检查错误。如果一切正常，驱动程序可以把数据（例如刚刚读出的一个磁盘块）传送给与设备无关的软件。最后，它向调用者返回一些用于错误

报告的状态信息。如果还有其他请求在队列中，那么其中的一个将被选择启动。如果队列中不再有请求，驱动程序将阻塞以等待下一个请求。

处理读写的请求是驱动程序的主要功能，但也可能有其他的需求。比如，驱动程序可能需要在系统启动或者设备第一次使用时初始化设备。还可能有管理电源需求、处理即插即拔设备或者记录事件的需要。

3.2.4 与设备无关的 I/O 软件

尽管某些 I/O 软件是设备相关的，但大部分 I/O 软件都是与设备无关的。设备驱动程序和设备无关软件之间的密切界限依赖于具体系统，因为对于一些本应该以设备无关方式完成的功能，出于效率和其他原因，实际上是由驱动程序完成的。图 3.6 所描述的是典型的由设备无关软件实现的功能。在 MINIX 3 中，大多数设备无关软件属于文件系统。尽管文件系统将在第 5 章中讨论，但这里仍简单地介绍一下设备无关软件，以便提供关于 I/O 的一些观点和更好地展示驱动程序放置在什么地方。

设备驱动程序的统一接口
缓冲
错误报告
分配和释放专用设备
提供与设备无关的块大小

图 3.6 与设备无关的 I/O 软件的功能

设备无关软件的基本功能是执行对所有设备公共的 I/O 功能，并向用户层软件提供一个统一的接口。下面将详细介绍以上问题。

设备驱动程序的统一接口

操作系统的一个主要问题是如何使所有 I/O 设备和驱动程序看起来或多或少是相同的。如果磁盘、打印机、键盘等接口方式都不相同，那么每次一个新的外围设备出现时，都必须为新设备修改操作系统。图 3.7(a)形象地说明了每个设备驱动程序有不同的与操作系统的接口的情况。相反，图 3.7(b)描述了一种不同的设计，在这种设计中所有驱动程序具有相同的接口。

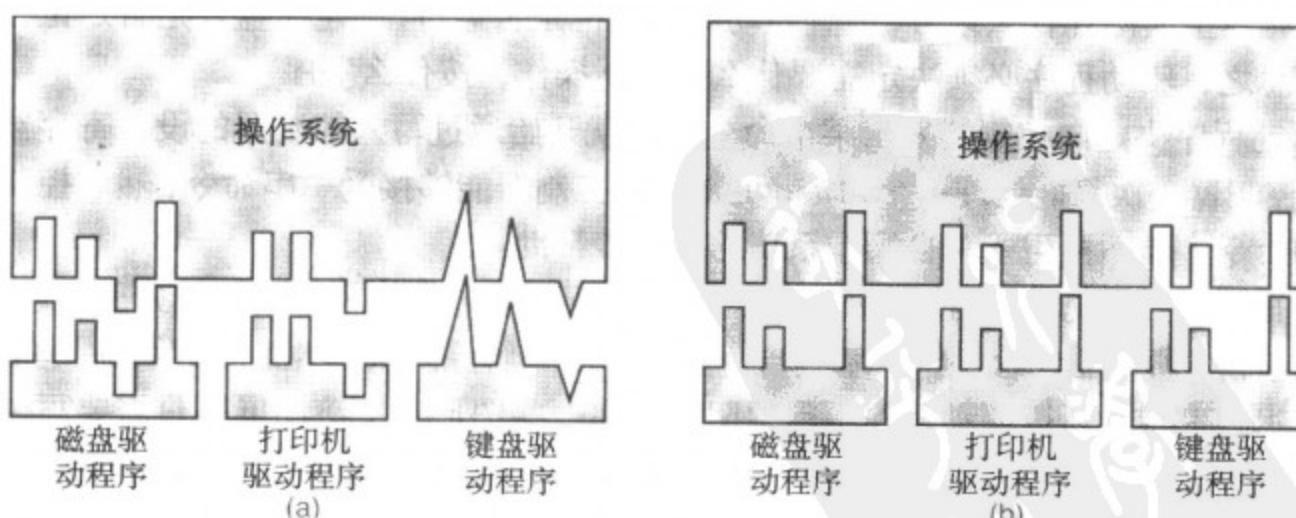


图 3.7 (a)没有标准的驱动程序接口；(b)具有标准的驱动程序接口

在有标准接口的情况下，倘若符合驱动程序接口，那么接入一个新的驱动程序就变得更加容易。这还意味着驱动程序的开发者知道他们应该做什么（比如，他们必须提供什么函数和应该调用

哪些内核函数)。实际上，并非所有的设备都是绝对一致的，通常只存在少数设备类型大体上几乎是相同的。例如，即使是块设备和字符设备也有许多函数是共同的。

使用统一接口的另一方面是如何给I/O设备命名。与设备无关的软件要负责将符号化的设备名映射到相应的驱动程序。例如，在UNIX和MINIX 3中一个设备名，如`/dev/disk0`，唯一地确定了一个特殊文件的i节点，这个i节点包含了主设备号(major device number)，通过主设备号就可以找到相应的设备驱动程序。i节点也包含次设备号(minor device number)，它作为参数传给驱动程序，用来指定要读或写的具体单元。所有设备都有主设备号和次设备号，而且所有驱动程序都是通过使用主设备号来选择驱动程序而得到访问的。

与命名密切相关的是保护。系统如何避免用户对设备的未授权访问呢？在UNIX、MINIX 3和后来的Windows版本中，如Windows 2000和Windows XP，设备作为命名对象出现在文件系统中，这意味着针对文件的常规保护规则也适用于I/O设备。系统管理员为每个设备设定合适的权限(比如，在UNIX中的`rwx`位)。多数个人计算机系统根本就不提供任何保护，所有进程都可以为所欲为。在多数大型主机系统中，用户进程绝对不允许访问I/O设备。UNIX中使用了一种更为灵活的方法。对应于I/O设备的设备文件的保护采用通常的`rwx`权限机制，所以系统管理员可以为每一台设备设置合理的访问权限。

缓冲

无论是对于块设备还是字符设备，缓冲都是一个问题。对于块设备，硬件通常坚持一次读写整个数据块，但是用户程序可以读写任意大小的单元。如果用户进程写半个块，那么操作系统通常将在内部保留这些数据，直到其余数据被写，这时数据块才离开系统内部写到磁盘上。对字符设备，用户向系统写数据的速度可能比设备输出的速度快，这就使得需要缓冲。在需要之前到达的键盘输入也需要缓冲。

错误报告

错误在I/O上下文中比在其他上下文中常见得多。当错误发生时，操作系统一定要尽可能处理它们。很多错误是设备特定的，所以只有驱动程序知道应该做什么，比如重试(retry)、忽略(ignore)或者panic(崩溃)。磁盘块导致的典型错误是它已经被损坏而不能再读。在驱动程序设法读取块一定次数后，它将放弃并通知设备无关软件。从现在开始起提及的如何处理错误是设备无关的。如果读取一个用户文件时发生错误，那么驱动程序会将错误报告给调用者。然而，如果读一个关键的系统数据结构时发生错误(比如包含表明空闲块位图的块)，那么操作系统可能不得不显示错误并终止运行。

分配和释放专用设备

某些设备，比如CD-ROM刻录机，在任意给定的时刻只能被一个进程使用。操作系统需对设备使用的请求进行检验，并根据被请求的设备是否可用来决定接受或拒绝请求。处理这些请求的一个简单方法是要求进程直接在代表设备的设备文件上执行open操作。如果设备是不可用的，那么open操作就会失败。于是关闭这样一个专用设备，然后释放它。

与设备无关的块大小

并非所有磁盘的扇区大小都是相同的。应该由与设备无关的软件来隐藏这一事实并向高层提供一个统一的块大小，比如，把几个块视为一个逻辑块。这样，高层软件只处理那些都使用同样逻辑块大小的抽象设备，逻辑块大小和物理扇区大小无关。类似地，一些字符设备一次一个字节地交付

它们的数据（如调制解调器），而其他的设备则以较大的单位交付它们的数据（如网络数据）。这些差异也可以被隐藏起来。

3.2.5 用户空间的 I/O 软件

尽管大多数 I/O 软件都在操作系统内部，但也有一小部分是与库和用户程序链接在一起构成的，甚至整个程序都运行在操作系统以外。系统调用，包括 I/O 系统调用，通常是由库过程实现的。当一个 C 程序包含调用

```
count = write(fd, buffer, nbytes);
```

时，库过程 `write` 将与程序链接在一起，并被包含在运行时出现在内存中的二进制程序中。所有这些库过程的集合显然也是 I/O 系统的一部分。

这些过程所做的工作只不过是为系统调用把它们的参数放在合适的位置，其他的 I/O 过程实际完成真实的工作。特别是输入输出的格式化是由库过程完成的。C 语言中的一个例子是 `printf`，它以格式串和可能的一些变量作为输入，构造一个 ASCII 字符串，然后调用 `write` 来输出这个串。作为 `printf` 的一个例子，考虑语句

```
printf("The square of %3d is %6d\n", i, i*i);
```

该语句格式化一个字符串，这个字符串是这样组成的，先是 14 个字符的串“the square of”，随后是表示 i 值的三个字符的串，然后是 4 个字符的串“is”，再后是表示 i^2 值的 6 个字符的串，最后是一个换行。

对输入而言，类似过程的一个例子是 `scanf`，它读取输入并将其存放到采用与 `printf` 同样语法的格式串描述的变量中。标准的 I/O 库包含许多涉及 I/O 的过程，它们都是作为用户程序的一部分运行的。

并非所有的用户级 I/O 软件都是由库过程组成的。另一个重要的分类是假脱机系统（spooling system）。假脱机（spooling）是多道程序系统中处理专用 I/O 设备的一种方法。考虑一种典型的假脱机设备：打印机。尽管在技术上很容易让任何用户进程打开代表打印机的字符设备文件，但是如果一个进程打开它，然后很长时间都不使用，那么其他进程也不能打印任何东西。

另一种方法是创建一个特殊的进程，称为守护进程（daemon），以及一个特殊目录，称为假脱机目录。为了打印一个文件，一个进程首先要生成需要打印的整个文件并把它放在假脱机目录里。由守护进程打印该目录下的文件，该进程是允许使用打印机设备文件的唯一进程。通过保护设备文件来防止用户直接使用，可以解决某些进程不必要地长期空占打印机的问题。

假脱机不仅用于打印机，也用于其他各种情况。例如，电子邮件通常使用守护进程。当一条信息被提交后，它被放到邮件假脱机目录下。随后邮件守护进程尝试发送它。在某些给定的瞬间，一些特别的目的地可能是临时不可到达的，所以守护进程把信息放在假脱机目录里，并标上状态信息以标志这条消息过一段时间后重新尝试发送。守护进程也可能把信息返回给发送者，并告知交付延迟，或者在延迟几小时或几天后，告知这条消息不能交付。所有这些都是在操作系统外运行的。

图 3.8 总结了 I/O 系统，给出了所有层次和每一层的主要功能。从底层开始，这些层依次是硬件、中断处理器、设备驱动程序、与设备无关的软件和用户进程。

图 3.8 中的箭头表明了控制流。比如，当一个用户程序试图从一个文件读一个块时，操作系统被调用来执行这个请求。与设备无关的软件在缓冲区的高速缓存中查找有关的块。如果需要的块不在其中，则调用设备驱动程序，向硬件发出一个请求，从磁盘读取该块。然后，进程被阻塞直到磁盘操作完成。

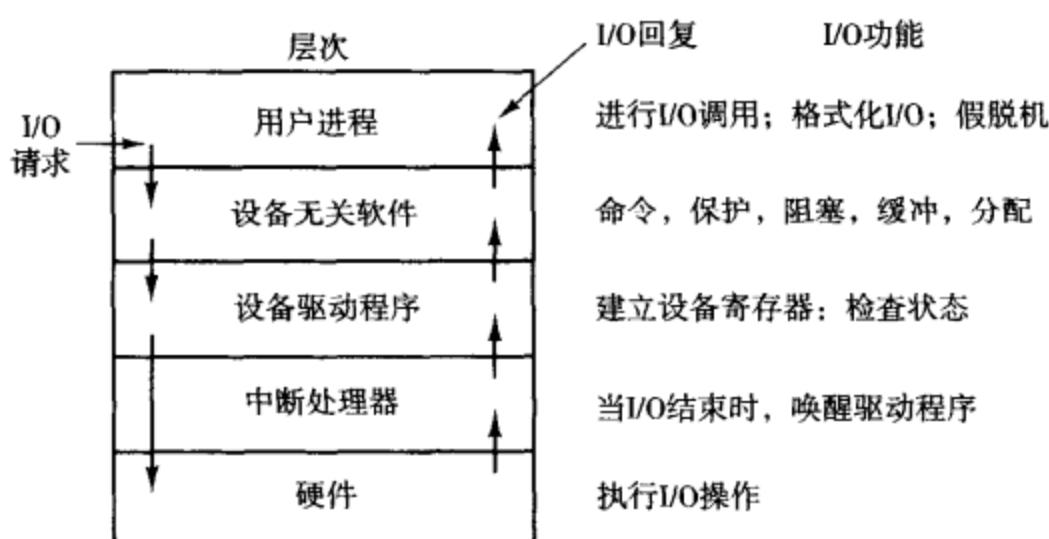


图 3.8 I/O 系统的层次以及每一层的主要功能

当磁盘操作完成时，硬件产生一个中断。中断处理器就会被运行以查明发生了什么，也就是说，此时哪个设备需要关注。然后，中断处理器从设备读取状态并唤醒休眠的进程来结束这次 I/O 请求，并让用户进程继续运行。

3.3 死锁

计算机系统中有许多独占资源，它们一次只能被一个进程使用。常见的例子有打印机、磁带机以及系统内部表中的表项。让两个进程同时向打印机输出会导致混乱。让两个进程同时使用同一文件系统表项会导致系统崩溃。正因为如此，所有的操作系统都具有（临时）授权一个进程排他地访问某一资源的能力，这个资源包括软件和硬件。

在许多应用中，一个进程需要独占地访问几种资源而不只是一种。例如，如果有两个进程想要分别刻录扫描的文档到 CD 上。进程 A 请求使用扫描仪，并被授权使用。进程 B 按不同的方式编程，首先请求 CD 刻录机，也被授权使用。现在 A 请求 CD 刻录机，但是这个请求被拒绝直到 B 释放它。遗憾的是，进程 B 不但不会释放 CD 刻录机，还会去请求扫描仪。这时，两个进程同时被阻塞，并且一直处于这样的状态。这种情况称为死锁。

除了请求专用 I/O 设备外，其他情况也可能导致死锁。例如，在一个数据库系统中，为了避免竞争状态，可将若干记录加锁。若进程 A 对记录 R1 加锁，进程 B 对记录 R2 加锁，随后这两个进程又各自试图把对方的记录也加锁，这时也将导致死锁。因此，软、硬件资源都有可能出现死锁。

本节将详细研究死锁，了解死锁是如何产生的，学习防止或避免死锁的方法。尽管这里讨论的是操作系统环境下的死锁问题，但是这些也会发生在数据库系统以及其他一些计算机科学的环境中，所以这些内容实际上可以应用到各种多进程系统中。

3.3.1 资源

进程对设备、文件等获得独占的访问权时有可能会发生死锁。为了使关于死锁的讨论尽可能地通用化，我们将这种需要排它使用的对象称为资源。资源可以是硬件设备（如磁带机）或一组信息（如数据库中加锁的记录）。计算机中通常有多种资源。有些类型的资源有若干个相同的实例，如三台磁带机。当某一资源有可互换副本时（称为可替代资源），其中任一个都可以用来满足对资源的请求。简而言之，资源是在任何时刻只能被单个进程使用的任何对象。

资源分为两类：可抢占资源（preemptable resource）和不可抢占资源（nonpreemptable resource）。可抢占资源可从拥有它的进程处抢占而没有任何副作用，存储器是一类可抢占资源。例如，一个系统有 64 MB 用户内存和一台打印机，如果两个 64 MB 的进程都要进行打印，进程 A 请求并得到了打印机，然后开始计算要打印的值。在它未完成计算任务之前，它的时间片用完并被换出。

然后进程 B 开始运行并请求打印机，但未成功。此时有潜在的死锁危险，因为进程 A 拥有打印机，而进程 B 占有内存，没有另外一个进程的资源，两个进程中的任何一个都不能继续执行。然而幸运的是，通过将进程 B 换出、进程 A 换入就可以抢占进程 B 的内存。于是，进程 A 继续运行并执行打印任务，然后释放打印机。这个过程将不会发生死锁。

与此相反，不可抢占资源是无法在不导致相关计算失败的情况下将其从占有它的进程处剥夺的。在一个进程已经开始刻盘时，如果突然将 CD 刻录机分配给另一进程，那么将划坏 CD 盘。在任何时刻 CD 刻录机都是不可抢占的。

总的来说，死锁与不可抢占资源有关，有关可抢占资源的潜在死锁通常可以通过在进程间重新分配资源而化解。所以，这里将重点放在不可抢占资源上。

使用一个资源需要的事件顺序可以用抽象的形式表示如下：

1. 请求资源。
2. 使用资源。
3. 释放资源。

如果请求时资源不可用，那么请求进程被迫等待。在一些操作系统中，资源请求失败时，进程自动被阻塞，在资源可用时再被唤醒。在其他系统中，资源请求失败将返回一个错误码，由请求进程等待一段时间后重试。

3.3.2 死锁的原理

死锁的规范定义如下：

如果一个进程集合中的每一个进程都在等待只能由本集合中的其他进程才能引发的事件，那么该组进程是死锁的。

由于所有的进程都在等待，所以没有一个进程能引起任何能够唤醒本集合中其他进程的事件，于是所有进程都将永远地等待下去。在这一模型中，假定进程只含有一个线程，并且阻塞的进程不能由中断唤醒。无中断条件使死锁的进程不会被时钟中断等唤醒，从而引发释放该集合中的其他进程的事件。

在大多数情况下，每个进程所等待的事件是本集合中其他进程正在占用的资源的释放。换言之，这组死锁进程的每个进程都在等待另一个死锁进程所占有的资源。但因为所有进程都无法运行，因而它们中的任何一个资源都无法释放，而且所有进程都不能被唤醒。至于进程数量和占用及请求的资源的数量和种类并不重要。这个结果对于任何类型资源（包括软件和硬件）都会发生。

死锁的条件

Coffman et al. (1971) 总结出了死锁发生的四个必要条件：

1. 互斥条件。每一资源要么被分配给了一个进程，要么就是可用的。
2. 占有和等待条件。已分配到了一些资源的进程可以请求新的资源。

3. 不可抢占条件。已分配给一个进程的资源不能强制性地被抢占，只能被占有它的进程显式地释放。
4. 环路等待条件。死锁发生时，系统中必然有一条由两个或两个以上的进程组成的环路，该环路中的每一个进程都在等待环路中下一个进程占用的资源。

所有这四个条件都是死锁发生所必需的，如果其中的任何一个条件不成立，死锁就不会发生。

在一系列论文中，Levine (2003a, 2003b, 2005)指出在文献中有各种被称为死锁的情况，并且Coffman等人的条件仅适用于被称为资源死锁很合适的情况。文献中包含有不满足这些条件的“死锁”的例子。比如，如果四辆车同时到达一个十字路口，并且尽量遵守每辆车应该给它右边的车让路的规则，那么没有一辆车能够运行，但这不是一个进程已经占用了一个独一无二的资源的情况。而且，这个问题是一个能够通过由警察从外界强加的关于优先权的决定来解决的“调度死锁”问题。

值得注意的是，每个条件都与系统的一种可选的策略相关。一种资源能否同时分配给不同的进程呢？一个进程能否在占有一个资源的同时请求另一个资源呢？资源能否被抢占呢？环路等待是否存在呢？在后面我们将会看到怎样通过破坏上述条件来预防死锁。

死锁模型

Holt (1972) 提出了用有向图建立以上四个条件的模型。图中有两类节点：圆形表示的进程和方形表示的资源，从资源节点到进程节点的弧表示该资源已被请求、授权并被进程占用。在图 3.9(a) 中，资源 R 正被进程 A 占用。

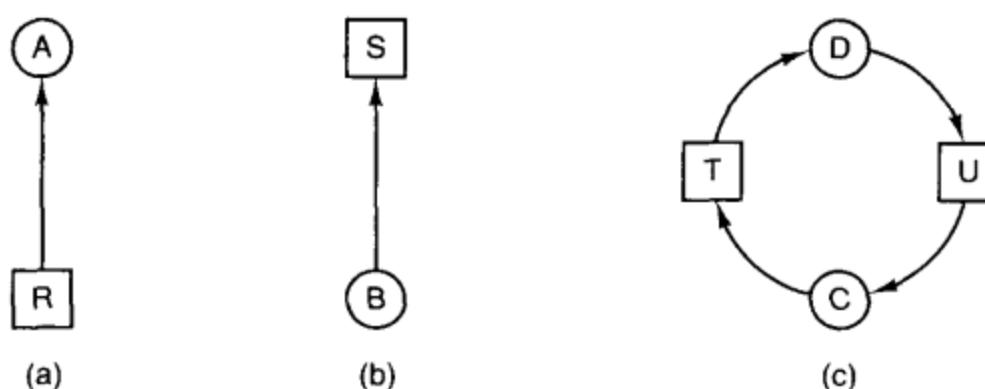


图 3.9 资源分配图：(a)占有一个资源；(b)请求一个资源；(c)死锁

从进程节点到资源节点的弧表示进程当前正请求该资源，并处于阻塞等待状态。在图 3.9(b) 中，进程 B 正在等待资源 S。图 3.9(c) 表示了死锁状态：进程 C 等待资源 T，资源 T 被进程 D 占用，进程 D 又在等待被进程 C 占用的资源 U。于是两个进程都将等待下去。图中的环表示涉及到这些进程和资源的死锁。本例中的环为 C-T-D-U-C。

再看资源图的用法。假设有三个进程 A, B, C 及三个资源 R, S, T。三个进程对资源的请求和释放如图 3.10(a)~(c) 所示。操作系统可以随时选择任何一个非阻塞的进程来执行，因此它可以選擇 A 运行直至 A 完成所有的工作，然后运行 B，最后运行 C。

上述顺序不会导致死锁（因为不存在对资源的竞争）。但执行过程也没有任何并行性。在进程的执行过程中，除了请求和释放资源，还要做计算或者输入输出工作。当进程串行执行时，不可能出现一个进程等待 I/O 时，另一个进程使用 CPU 计算的情形。所以严格的串行操作有可能不是最优的。另一方面，如果进程根本不执行 I/O 操作，那么最短作业优先调度将优于时间片轮转调度，所以在某些情况下，串行执行可能是最优的。

如果假设进程操作包含 I/O 和计算，那么时间片轮转法是一种合理的调度算法。资源请求顺序可能如图 3.10(d) 所示。若按此顺序执行，图 3.10(e) 至图 3.10(j) 是相应的资源图。在发生请求(4)之

后, 进程 A 被阻塞等待 S, 如图 3.10(h)所示, 随后的两步中 B 和 C 也将阻塞, 最终导致一个环路和死锁, 如图 3.10(j)所示。从这时起, 系统将冻结。

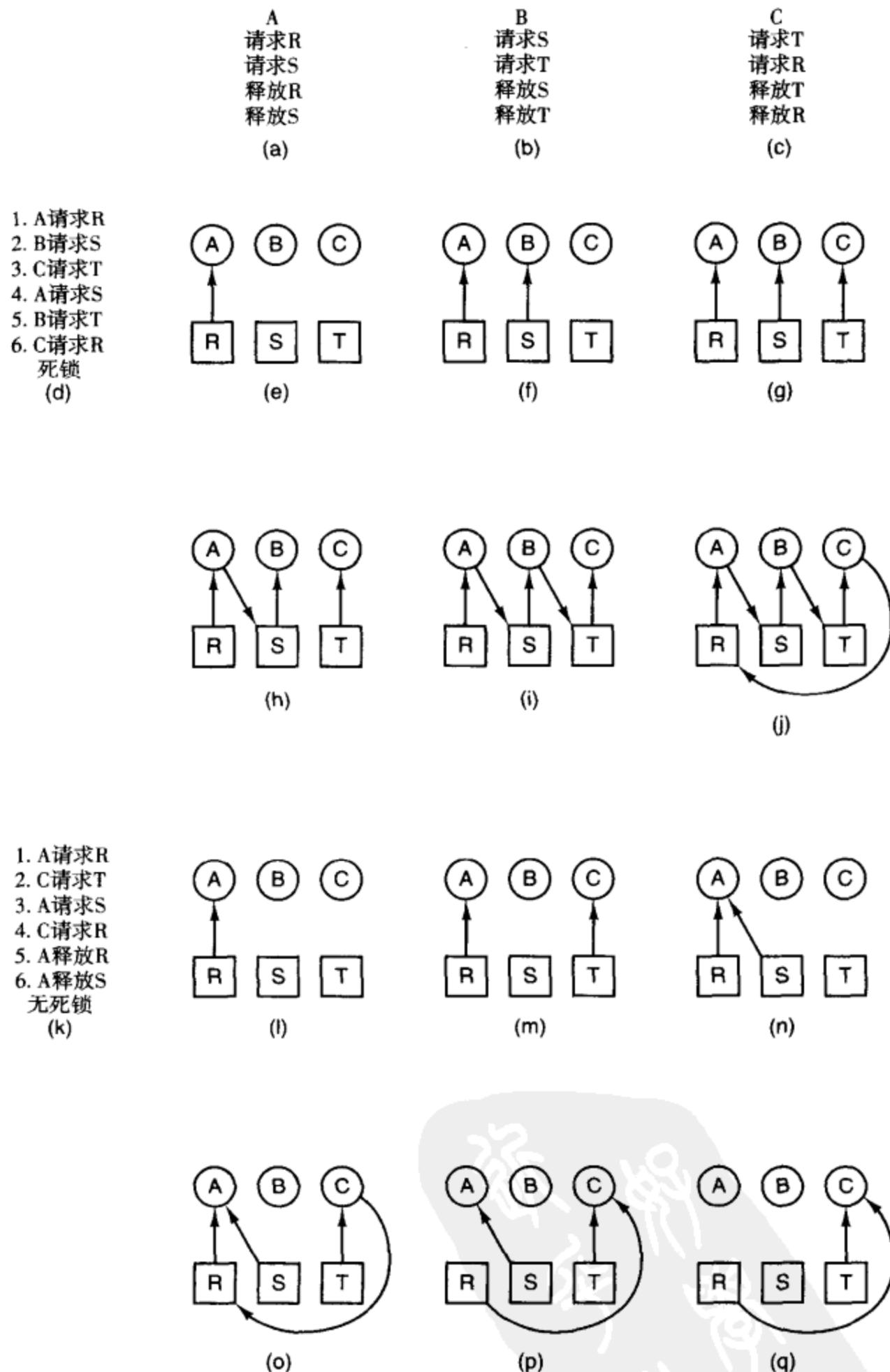


图 3.10 一个死锁是如何产生以及如何避免的例子

然而, 正如前面提到的, 操作系统没有必要按某种特定次序来执行这些进程。特别是对于一个可能导致死锁的资源请求, 操作系统可以简单地不批准该请求, 并将进程挂起(即不参与调度), 直

到它处于安全状态。在图3.10中，假设操作系统知道可能导致死锁，它可以不把资源S分配给B，于是B被挂起。假设只运行进程A和C，则资源请求和释放过程如图3.10(k)所示，而不是如图3.10(d)所示。该过程的资源利用情况如图3.10(l)至图3.10(q)所示，其中无死锁发生。

第(q)步执行完后，可将资源S分配给B，因为A已经结束并且C已得到所需的全部资源。即使B最终因请求T而阻塞，也不会发生死锁，B只需等C结束。

在本章后面还会研究一个具体的算法，以做出对资源进行分配而不会死锁的决策。此处需要说明的是，资源图可作为分析一给定的请求/释放序列是否将导致死锁的一种工具。只需按请求和释放的顺序一步步地执行，每一步之后都检查其中是否包含环路。如果是，则发生死锁，反之，则不发生死锁。虽然这里的例子中只涉及同一类资源只包含一个实例的情况，但资源图完全可以推广到同种资源含有多个实例的情况(Holt, 1972)。然而，Levine(2003a, 2003b)指出，当涉及可替换资源时，问题将变得很复杂。如果即使图的一个分支不是环路的一部分，也就是说，如果一个没有死锁的进程占有某个资源的一个副本，那么死锁也可能不发生。

概括而言，处理死锁有四种策略：

1. 忽略该问题，也许你忽略它，它也会忽略你。
2. 检测死锁并恢复。让死锁发生，检测它们并采取行动解决问题。
3. 谨慎地对资源进行动态分配，动态避免死锁。
4. 通过破坏上述四个必要条件之一，来防止死锁发生。

下面四小节将依次讨论这四种方法。

3.3.3 鸵鸟算法

最简单的方法就是鸵鸟算法：把头埋到沙子里，假装根本没有问题发生。每个人对该方法的看法不同。数学家认为这种方法根本不能接受，不管花多大代价也要彻底防止死锁的发生；工程师们则想要了解死锁发生的频率、系统因各种原因崩溃的频率以及死锁的严重程度。如果死锁平均每5年发生一次，而系统每周会因硬件故障、编译器错误或操作系统错误而崩溃一次，那么大多数工程师不会以性能损失或者易用性损失的代价来消除死锁。

为了使这个对比更加具体，假设UNIX(和MINIX 3)潜在地受到某些死锁的威胁，不过这些死锁从没有被检测到过，因此理所当然地自动解除了。系统中进程的总数目由进程表项的数量制约。因此进程表项是有限的资源，如果一个fork调用由于进程表满而失败，那么一种合理的办法是等待一段随机的时间后重试。

现假设MINIX 3系统的进程表有100项，有10个进程在执行，每一个都要创建12个子进程。在每个进程创建9个子进程后，原来的10个进程和新创建的90个进程将用完进程表的所有表项。那么，这10个进程都将进入一个无休止的循环——执行fork，从而导致失败，即发生死锁。当然，发生这类事件的概率是很小的，但它的确会出现。难道会为了消除这种状况就放弃进程和fork调用吗？

打开文件的最大数目受i节点表大小的限制，所以当i节点表满时会发生类似的问题。磁盘上的交换空间是另一种有限的资源。实际上，几乎操作系统中的每一种表格都代表了一种有限的资源。只是由于可能出现n个进程请求1/n的资源，接着每个进程再试图请求另外一份资源的情况，难道就该为此抛掉这一切吗？

大多数操作系统，包括UNIX, MINIX 3 和 Windows，处理这一问题的办法仅仅是忽略它，其假设前提是大多数用户宁可在极偶然的情况下发生死锁，也不愿接受只能创建一个进程、只能打开一个文件等限制。如果可以不花什么代价就能够解决死锁，那么就没有问题了。问题是，这种代价通常很大，而且常常会给进程带来许多不便的限制，正如我们马上将要看到的。于是我们不得不在方便性和正确性之间做出令人不愉快的权衡，要充分考虑哪一个、对谁最重要。在这些条件下，很难找到通用的解决办法。

3.3.4 死锁的检测和恢复

第二种技术是死锁检测和恢复。采用这种技术时，系统只需监视资源的请求和释放。每次资源被请求或释放时，资源图被更新，同时检测是否存在环路。如果存在一个环，则撤销环中的一个进程，如果仍不能消除死锁，则撤销另一个进程。如此往复直至环路被消除。

一种更粗略的方法甚至不维护资源图，而是周期性地检测是否有进程连续阻塞超过一定时间，如1小时。一旦发现这样的进程，则将其撤销。

检测和恢复这种策略常用于大型主机，特别是批处理系统。因为在批处理系统中，取消一个进程然后重新启动它通常是可以接受的。但必须要将该进程修改了的文件恢复到初始状态，并消除它所导致的所有副作用。

3.3.5 死锁的预防

第三种死锁策略是对进程施加适当的限制，以使得死锁在结构上是不可能的。Coffman et al. (1971) 提出的四个条件为某些可能的解决方案提供了线索。

首先考虑破坏互斥使用条件。如果资源不被单一进程独占，那么死锁肯定不会发生。然而，允许两个进程同时使用打印机会造成混乱，这是显然的。通过采用假脱机技术可以允许多个进程同时产生输出。在该模型中，唯一真正请求物理打印机的进程是打印机守护进程，由于它决不会请求其他的设备，所以不会因打印机而发生死锁。

但是，假脱机技术并不适用于所有的设备（进程表自身就不适合使用假脱机技术）。而且，在使用假脱机技术时对磁盘空间本身的竞争也可能导致死锁。例如，若两个进程分别占用了可用的假脱机磁盘空间的一半用于输出，而任何一个也没有能够完成输出，那会怎么样呢？假设守护进程被设计为不等全部输出放入假脱机目录下就开始打印，如果在头一轮打印之后输出进程决定等待几个小时，那么打印机就会闲置。为了避免这种现象，一般把打印机守护进程设计成完整的输出文件就绪后才开始打印。在这种情形下，就会有两个进程，每一个都完成了一部分的输出，但不是它们的全部输出，于是无法继续进行。没有一个进程能够进行，结果在磁盘上出现了死锁。

Coffman等人表述的第二个条件似乎更有希望。只要禁止已拥有资源的进程再等待其他资源便可以消除死锁。一种实现方法是规定所有进程在开始执行前请求所需的全部资源。如果所需的资源全部可用，那么就将它们分配给这个进程，于是它肯定能够运行到结束。如果有一个或多个资源正被使用，则不进行分配，而让进程等待。

这种方法的一个直接的问题是，许多进程直到运行时才知道它需要多少资源。另一个问题是，这种方案中资源不能够最优化地使用。例如，一个进程先从输入磁带上读取数据，进行一小时的分析，最后将输出写到输出磁带上，同时将其在绘图仪上绘出。如果所有资源一定要提前请求，那么这个进程就把输出磁带机和绘图仪控制在一个小时内。

破坏占有和等待条件的另一个略微不同的方案是，要求当一个进程请求资源时，先暂时释放其当前占用的所有资源，然后再尝试一次来获得所需的全部资源。

破坏第三个条件(不可抢占)比第二个更困难。若一个进程已分配到一台打印机并正在进行打印输出,如果因为它需要的绘图仪无法得到而强制性地将打印机抢占掉,这往好处说是棘手的,往坏处说就是不可能的。

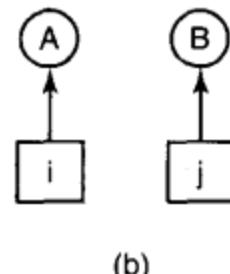
现在只剩下一个条件了。消除环路等待有几种方法。一种是简单地使用一个规则,保证每一个进程在任何时刻只能占用一个资源,如果需要第二个,它必须先释放第一个。对于一个需要把一个大文件从磁带机上读入并输出到打印机的进程,这种限制是不可接受的。

另一种避免环路等待的方法是给所有资源提供一个全局编号,如图3.11(a)所示。现在的规则是,进程可以在它们需要时请求资源,但是所有请求必须按照编号的顺序提出。进程可以先请求扫描仪,然后请求磁带机,但不可以先请求绘图仪,后请求扫描仪。

按此规则,资源分配图中肯定不会出现环路。现在看一下在两个进程的情况下,这种方法为何可行,如图3.11(b)所示。只有在A请求资源j及B请求资源i的情况下才会发生死锁。设i和j是不同的资源,它们将具有不同的数值。若*i*>*j*,则不允许A请求j,因为这个编号小于A已有资源的编号;若*i*<*j*,则不允许B请求*i*,因为这个编号小于B已有资源的编号。不论哪种情况都不可能发生死锁。

1. 图像操作员
2. 扫描仪
3. 绘图仪
4. 磁带机
5. CD-ROM

(a)



(b)

图3.11 (a)对资源排序编号;(b)一个资源分配图

对于多进程的情况,同样的逻辑依然成立。对于每一种情况,总有一个被分配的资源编号是最高的。占用该资源的进程不可能请求其他已被占用的各种资源,它或者将执行完毕,或者最坏的情况是请求编号更高的资源,而编号更高的资源肯定是可获得的。最终它将结束并释放它的资源,这时其他占有最高编号资源的进程也可以执行完。简言之,存在一种所有进程都可以执行完的情形,因而没有死锁发生。

该算法的一个变体是摈弃必须按升序请求资源的限制,而仅要求不允许进程请求编号比当前所占有资源编号低的资源。若一个进程起初请求9号和10号资源,随后将其释放,它实际上相当于从头开始,所以没有必要阻止它现在请求1号资源。

尽管对资源编号的方法消除了死锁的问题,但几乎找不出一种使每个人都满意的编号次序。当资源包括进程表项、假脱机磁盘空间、加锁的数据库记录及其他抽象资源时,潜在的资源及各种不同用途的数目会变得如此之大,以至于使编号法根本无法使用。此外,如Levine(2005)指出的那样,编号资源不可替代,即在这种规则下,不能访问资源的可用副本。

预防死锁的各种方法如图3.12所示。

条件	方法
互斥	一切都使用假脱机技术
占有和等待	在开始时请求所有资源
不可抢占	抢占资源
环路等待	对资源按数值编号

图3.12 死锁预防方法总结

3.3.6 避免死锁

从图 3.10 中，我们可以看到不是通过对进程随意强加一些规则来避免死锁的，而是通过对每一次资源请求进行认真的分析来判断它是否能安全地分配的。问题是，是否存在一种算法总能做出正确的选择从而避免死锁呢？答案是肯定的，但条件是必须事先获得一些特定的信息。本小节将讨论几种通过仔细分配资源来避免死锁的方法。

单一资源的银行家算法

Dijkstra (1965) 提出了一种能够避免死锁的调度算法，称为银行家算法。它的模型基于一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度。银行家不必为了能在同一时间借给每一个客户满额的货款而在手头上保存足够的现金。在图 3.13(a)中可以看到四个客户 A, B, C, D，每个客户都有一个贷款额度(如每人1000美元)，银行家知道不可能所有客户同时都需要最大贷款额，所以他只保留 10 个单位的资金来为客户服务，而不是 22 个单位。他也相信每个客户在达到自己贷款的最高额度时都能尽可能快地偿还贷款(这是一个小城镇)，所以最后他知道他能为所有的请求提供服务(在这个类比中，我们将客户比做进程，将贷款比做磁带机，将银行家比做操作系统)。

已使用			最大			已使用			最大			已使用			最大		
A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
0	0	6	1	1	6	1	2	6	1	2	6	1	2	4	4	7	7
0	5		1	5		2	4		2	5		4	7		4	7	
0	4		2	4					2	4							
0	7		4	7					4	7							
可用: 10			可用: 2			可用: 1											
(a)			(b)			(c)											

图 3.13 三种资源分配状态: (a)安全; (b)安全; (c)不安全

每一幅图表示一种系统资源分配的状态，也就是一个客户的列表，反映了已有贷款(已经分配的磁带机)和可用最大额度(随后一次需要的磁带机的最大数量)。如果存在一个其他状态的序列，那么这些状态将使得所有的客户获得达到信誉额度的贷款(所有的进程获得它们所有的资源并终止)。

客户们各自做自己的生意，在某些时刻需要贷款(相当于请求资源)。在某一特定时刻，具体情况如图 3.13(b)所示。这个状态是安全的，由于保留着两个单位，银行家能够拖延除了 C 以外的其他请求，因而可以让 C 先完成，然后释放 C 所占的 4 个资源。有了这 4 个资源，银行家就可以给 D 或 B 分配所需的贷款单位，依次类推。

考虑向 B 又提供了一个他所请求的贷款单位的情形，如图 3.13(c)所示。我们可以得到如图 3.13(c)的状态，这个状态是不安全的。如果所有的客户突然都请求最大的限额，而银行家又无法满足其中任何一个的要求，那么就会产生死锁。不安全状态不一定会引起死锁，因为客户不一定需要其最大额度贷款，但银行家不敢抱这种侥幸心理。

银行家算法在每个资源请求发出时对这个请求进行考察，检查满足这一请求时是否会达到安全状态。如果能达到，那么就满足该请求；如果不能达到，那么就推迟这一请求。为了看状态是否安全，银行家看他是否有足够的资源来满足某一个客户。如果有，那么这笔贷款被认为是能够收回的，并且最接近最大限额的客户将被检查，依次类推。如果所有的贷款最终都被收回，那么该状态是安全的，最初的请求可以批准。

资源轨迹

上面的算法描述了单一资源的情况（例如仅有磁带机或仅有打印机，而不是多种资源）。在图3.14中，我们可以看到一个处理两个进程和两种资源（打印机和绘图仪）的模型。横轴表示进程A的指令执行过程，纵轴表示进程B的指令执行过程。进程A在 I_1 处请求一台打印机，在 I_3 处释放，再 I_2 处请求一台绘图仪，在 I_4 处释放。进程B在 I_5 到 I_7 之间需要绘图仪，在 I_6 到 I_8 之间需要打印机。

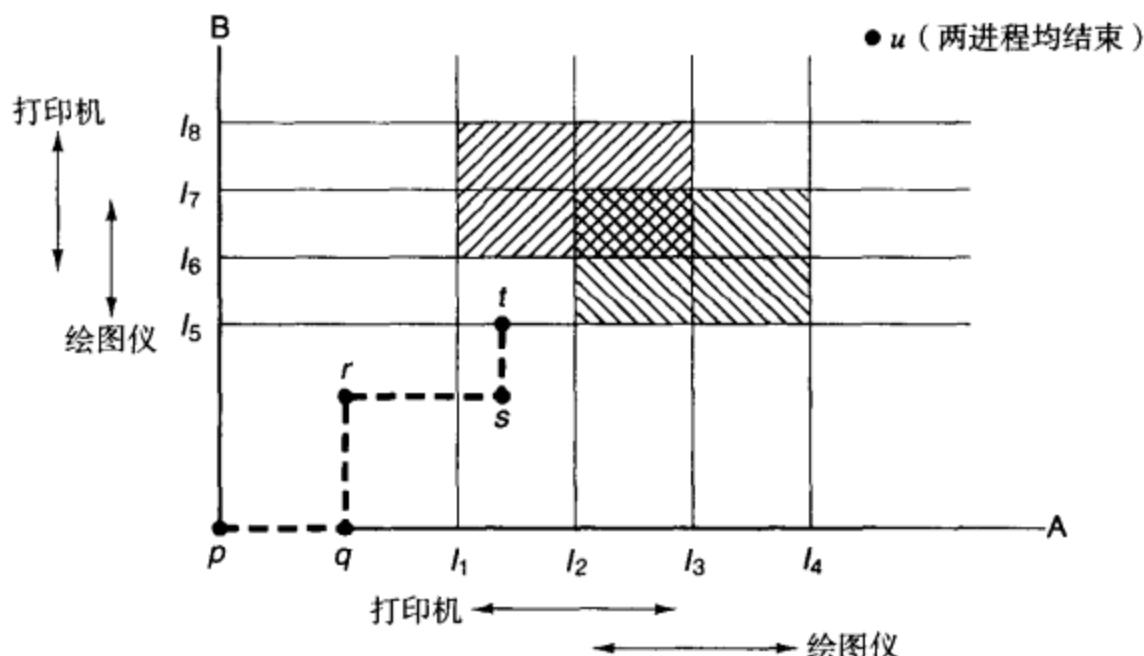


图3.14 两个进程的资源轨迹图

图中的每一点都示出了两个进程的连接状态。初始点为 p ，没有进程执行任何指令。如果调度器选择A先运行，那么在A执行一段指令后到达 q ，此时B没有执行任何指令。在 q 点，如果轨迹沿垂直方向移动，则表示调度器选中B运行。在单处理机情况下，所有路径都只能是水平或垂直方向的，不会出现斜向的。而且，运动方向一定是向右或向上，而不会是向左或向下，因为进程的执行不可能后退。

当进程A由 r 向 s 移动并穿过 I_1 线时，它请求并获得打印机。当进程B到达 t 时，它请求绘图仪。

图中的阴影部分是特别值得注意的，用从左下到右上斜线填充的区域表示在该区域中两个进程都拥有打印机，而互斥使用的规则决定了不可能进入该区域。另一种斜线表示的区域表示两个进程都拥有绘图仪，且同样不可进入。

如果系统一旦进入由 I_1, I_2 和 I_5, I_6 组成的矩形区域，那么最后一定会到达 I_2 和 I_6 的交叉点，此时就会发生死锁。在该点处，A请求绘图仪，B请求打印机，而且这两种资源均已被分配。整个矩形区域都是不安全的，因此绝对不能进入这个区域。在 t 处唯一的办法是运行进程A直到 I_4 ，过了 I_4 后则可以按任何路线前进，直到终点 u 。

这里需要明确的是，在点 t ，进程B请求资源。系统必须决定是否分配。如果系统把资源分配给B，系统进入不安全区，最终产生死锁。要避免死锁，应该将B挂起，直到A请求并释放绘图仪。

多种资源的银行家算法

资源轨迹图的方法很难应用到具有任意数目的进程、任意数量的资源种类且每种资源有多个实例的一般情况。然而，银行家算法可以被推广来解决这个问题。图3.15给出了其工作原理。

	进程	磁带机	绘图仪	打印机	CD-ROM
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

已分配的资源

	进程	磁带机	绘图仪	打印机	CD-ROM
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

仍需要的资源

图 3.15 多种资源的银行家算法

在图 3.15 中可以看到两个矩阵。左边的矩阵显示出对 5 个进程分别已分配的各种资源数，右边的矩阵则显示了使各进程运行结束还需要的各种资源数。与单一资源的情况一样，各进程在执行前给出其所需的全部资源量，所以系统的每一步都可以计算出右边的矩阵。

图 3.15 最右边的 3 个向量分别表示总资源 E、已分配资源 P 和剩余资源 A。由 E 可知系统中共有 6 台磁带机、3 台绘图仪、4 台打印机和 2 台 CD-ROM。由 P 可知当前已分配了 5 台磁带机、3 台绘图仪、2 台打印机和 2 台 CD-ROM。该向量可通过将左边矩阵的各列相加得到，剩余资源向量可通过从资源总数中减去已分配资源数得到。

检查一个状态是否安全的算法描述如下：

1. 查找右边矩阵中是否有一行，其未被满足的资源数均小于或等于 A。如果不存在这样的行，则系统将死锁，因为任何进程都无法运行结束。
2. 若找到这样一行，则可以假设它获得所需的资源并运行结束，将该进程标记为结束，并将资源加到向量 A 上。
3. 重复以上两步，直到所有的进程都标记为结束。若能达到这种状态，则初始状态是安全的；或者直到发生死锁，则初始状态是不安全的。

如果在第 1 步中同时存在若干进程均符合条件，则不管挑选哪一个进程运行都没有关系，因为可用资源或者将增多，或者保持不变。

现在回到图 3.15 的例子，当前状态是安全的。假设进程 B 现在请求一台打印机，可以满足它的请求，而且所得到的系统状态仍然是安全的（进程 D 可以结束，然后是 A 或 E 结束，剩下的进程相继结束）。

假设进程 B 获得两台可用的打印机中的一台以后，E 试图获得最后的一台打印机，若分配给 E，可用资源向量将减到 (1 0 0 0)，这时将导致死锁。显然 E 的请求不能立即满足，必须延迟一段时间。

银行家算法最早由 Dijkstra 于 1965 年发表。从那之后几乎每本操作系统的专著都会详细地描述它，许多论文的内容也围绕该算法，但很少有作者指出该算法虽然很有意义但缺乏实用价值，因为很少有进程能够在运行前就知道其所需资源的最大值，而且进程数不是固定的，往往在不断地变化（如用户的登录或退出），况且原本可用的资源也可能突然间变得不可用（如磁带机可能会坏掉）。因此，在实际中，如果有，也只有极少的系统使用银行家算法来避免死锁。

总之，前面描述的死锁预防方案过于严格，这里的死锁避免算法又需要通常无法得到的信息。如果读者能想到一种理论上和实际中都适用的通用解法，就可以在计算机科学的杂志上发表一篇论文。

尽管避免和预防在通常情况下都不是很有希望,但对于一些特殊的应用,有许多有名且很好的特殊目的的算法。例如,在许多数据库系统中,一个频繁发生操作就是请求对一些记录加锁,然后更新所有加锁的记录。当有多个进程同时运行时,就有可能发生死锁。为了解决这个问题,将使用特殊的技术。

最常用的一种方法是**两阶段加锁法**。在第一阶段,进程试图将其所需的全部记录加锁,一次锁一个记录。若成功,就开始第二阶段,完成更新然后释放锁。在第一阶段并没有做实际的工作。

如果在第一阶段某一进程需要的记录已被加锁,则该进程释放它所有已加锁的记录,然后重新开始第一阶段。从某种意义上说,该方法有点类似于提前或者至少是在一些不可逆的操作之前请求所有资源。在两阶段加锁的一些版本中,如果在第一阶段遇到了已加锁的记录,那么并不会释放锁然后重新开始。在这些版本中,可能会产生死锁。

不过这一策略并不通用。例如,在实时系统和过程控制系统中,由于一个进程缺少一个可用资源就半途中断它,并重新开始执行,这是不可接受的。如果一个进程已经在网络上读写消息、更新文件或任何不能安全重复的操作,那么重新运行进程也是不可接受的。该算法仅适用于那些由于程序员仔细安排了程序而使得在第一阶段可以随时停止并重新执行的程序。但许多应用都不能按这种方式来设计。

3.4 MINIX 3 中的 I/O 概述

MINIX 3 的 I/O 结构如图 3.8 所示。其中,上边四层对应于图 2.29 中所示的 MINIX 3 的 4 层结构。以下章节将对每一层进行简介,重点放在驱动程序上。中断处理已在第 2 章做过介绍,设备无关的 I/O 将在第 5 章中讨论。

3.4.1 MINIX 3 中的中断处理器和 I/O 访问

许多设备驱动程序在启动一些 I/O 设备后阻塞,等待某种消息到达。这种消息往往由该设备的中断处理器产生。另外一些设备驱动程序不启动物理的 I/O(如从 RAM 盘中读数据,然后写到一个内存映射的显示器),也不使用中断,而且也不等待从 I/O 设备发来的消息。前一章中已详细讨论了内核里的中断产生消息和导致任务切换的机制,不再赘述。这里将概要地讨论驱动程序里的中断和 I/O。细节将放在讨论各种设备代码时叙述。

对于磁盘设备,输入输出通常仅仅是命令设备执行某个操作,然后等待该操作结束。大部分工作由磁盘控制器完成,而需要中断处理器做的工作很少。如果所有的中断处理都这么简单,那么工作就容易多了。

然而,有时低层中断处理器有更多的事情要做。消息传递机制的代价是较高的,所以对于中断本身很频繁且每次中断所处理的 I/O 操作量又很少的情况,更好的办法是让中断处理器做更多的工作,而推迟向驱动程序发送消息,直到后续中断到来,这时驱动程序有更多的工作需要完成。在 MINIX 3 中,对于大多数 I/O 这是不可能的,因为内核中的低层中断处理器是对几乎所有设备通用的程序。

在上一章中,我们看到时钟是一个例外。因为它和内核一起编译,所以时钟能够拥有自己的完成额外工作的中断处理器。大多数的时钟节拍除了维护系统时间外几乎无事可做。在这种情况下,无须在每次时钟中断时都向时钟任务发送消息。时钟中断处理器将增加一个变量 *realtime*,这可能增加在 BIOS 调用期间对时钟节拍计数的校正。中断处理器对用户时间和系统时间做一些附加的简单算术递增计数,对当前进程递减 *ticks_left* 计数器,并检查定时器是否到期。只有当前进程的时间片用完或者定时器到期时才会向时钟任务发送消息。

在 MINIX 3 中，时钟中断处理器是独特的，因为时钟是唯一运行在内核空间的中断驱动设备。事实上，时钟硬件是集成到 PC 上的，时钟中断线并不会连接到用来插入附加 I/O 控制器的插槽上的任何端口，所以安装时钟升级包来替代时钟硬件和厂商提供的驱动程序是不可能的。所以，时钟驱动程序被编译到内核里并能访问内核空间的任意变量是合理的。但是 MINIX 3 的一个主要设计目标就是使得其他任何设备驱动程序获得这样的访问都是不可能的。

运行在用户空间的设备驱动程序不能直接访问内核内存或者 I/O 端口。尽管允许一个中断服务器调用执行在用户空间的上下文中的服务器是可能的，但这违背了 MINIX 3 的设计原则。这将比让一个用户空间的进程调用内核空间的函数更危险。在那种情况至少可以保证内核空间的函数是由一个有能力的、有安全意识的操作系统的设计者写的，这个设计者可能就是读过这本书的某个人。但是内核不应该相信用户程序提供的代码。

用户空间的设备驱动程序可能需要若干不同层次的 I/O 访问。

1. 驱动程序可能需要访问其正常数据空间以外的内存。管理 RAM 盘的内存驱动，就是一个需要这种访问的驱动的例子。
2. 驱动程序可能需要读写 I/O 端口。这些操作的机器级指令仅在内核态下可用。正如我们即将看到的，硬盘驱动程序就需要这种访问。
3. 驱动程序可能需要响应可预测中断。比如，硬盘驱动程序向磁盘控制器发送命令，这将导致在操作完成后发生中断。
4. 驱动程序可能需要响应不可预测中断。键盘驱动程序属于这一类。这本来可以考虑作为前面一项的子类，但是不可预测性使事情变得复杂。

所有这些情况都由系统任务处理的内核调用来支持。

第 1 种情况（访问额外的内存段）充分利用了 Intel 处理器提供的硬件分段。尽管一个普通的进程只能访问自己的代码段、数据段和栈段，但是系统任务允许其他段被用户空间的进程定义和访问。因此，内存驱动程序能够访问作为 RAM 盘使用的保留区域以及其他指定的专用访问区域。终端驱动程序按同样的方式访问视频显示适配器上的内存。

对于第 2 种情况，MINIX 3 提供内核调用来使用 I/O 指令。系统任务代表缺少权限的进程完成实际的 I/O 操作。本章随后的部分将讨论硬盘驱动程序是如何使用这一服务的。这里先进行一下概述。磁盘驱动程序可能必须要向一个单一的输出端口写指令来选择磁盘，然后从另外一个端口读来检验设备是否就绪。如果通常响应很快，那么可以通过轮询检测来完成。需要写的端口和数据或者接收的读取数据存放的位置都由内核调用来指定。这要求读取端口的调用是非阻塞的，而事实上，内核调用也是非阻塞的。

一些避免设备失效的保证措施是有用的。一个轮询检测循环可能包括一个计数器，如果在一定次数的重复后设备仍未就绪，那么这个计数器会终止循环。总的来说，这不是一个好的想法，因为循环执行的时间和 CPU 的速度相关。解决这个问题的一种方法是用一个与 CPU 时间相关的值来启动计数器，可能在系统启动时使用一个全局变量来初始化。MINIX 3 的系统库提供了一种更好的方法，因为系统库提供了 *getuptime* 函数。它利用内核调用来获取由时钟任务维护的自从系统启动以来的时钟节拍的计数器。使用该信息来跟踪一次循环所用的时间的代价是每次重复中的额外的内核调用的费用。另一种可能是让系统任务设定看门狗定时器。但为了从定时器接收通知，需要一个阻塞的 *receive* 操作。如果需要快速响应，那么这不是一种好的解决方法。

硬盘也使用各种用于 I/O 的内核调用，这使得向系统任务发送一列要写的数据和端口或者是需要改变的变量成为可能。这是很有用的。将要讨论的硬盘驱动程序需要向 7 个输出端口写一连串的

字节值来初始化一个操作。其中的最后一个字节是一个命令，当磁盘控制器在完成这个命令后会产生一个中断。所有这些都可以由一个内核调用来完成，从而极大地减少了需要的消息数量。

现在我们讨论上面列表中的第3种情况：响应一个可预测的中断。正如在系统任务讨论中注意到的那样，当一个中断代表一个用户空间程序（使用 `sys_irqctl` 内核调用）被初始化时，中断处理器例程总是 `generic_handler`，即一个被定义为系统任意一部分的函数。这个例程把中断转化为对中断所代表的进程的通知消息。因此设备驱动程序在内核调用向控制器发出命令后必须初始化 `receive` 操作。当接到通知时，设备驱动程序继续做中断服务必须完成的工作。

尽管在这种情况下，中断是可以预测的，但还是应该谨慎地防止有时某些事情会发生故障。为了对中断可能被错误地触发这种情况做好准备，进程可以请求系统任务设定看门狗定时器。看门狗定时器也产生通知消息，因此 `receive` 操作能够得到一个通知，不管是由于中断发生还是定时器到期。虽然一个通知不能传递很多信息，但这并不是一个问题，因为通知消息指出了它的起源。虽然两个通知都由系统任务产生，但是中断的通知看起来是来自硬件，而定时器到期的通知看起来是来自时钟。

还有另外一个问题。如果一个中断以一种及时的方式被接收，并且看门狗定时器已经被设定，那么在将来某一时刻定时器的到期将会被另外一个 `receive` 操作检测到，这可能是在驱动程序的主循环里。一种解决方法是当来自硬件的通知被接收时，让内核调用禁用定时器。另外一种方法是，如果下一个 `receive` 操作是来自于不可预测时钟的消息，那么这样的消息可以被忽略，并再次调用 `receive` 操作。虽然不太可能，但是磁盘操作会在一个不可预测的长延时之后发生，并只在看门狗超时以后产生中断，这一点是可以想象的。同样的解决方法也可以在这里应用。当超时发生时，内核调用将禁用中断，或者并不是处理中断的 `receive` 操作会忽略任何来自硬件的消息。

当中断首先被启用时，内核调用能够为中断设定一个策略，现在是提及这件事情的时候。这个策略只是一个标志，这个标志决定中断是应该自动地被重新启用还是应该保持禁用一直到它所服务的设备驱动程序调用内核调用重新启用它。对于磁盘驱动程序，在中断以后可能还有大量的工作要做，因此最好使中断禁用直到所有的数据复制完毕。

列表中的第4种情况是最有疑问的。键盘支持是提供了输出和输入的 `tty` 驱动程序的一部分。此外，许多设备被支持。所以输入可能来自本地的键盘，但也可能来自通过串口线或者网线连接的远程用户。而且若干进程可以同时运行，每个进程为不同的本地或远程终端产生输出。当不知道中断何时发生时，如果同一个进程可能需要对其他的输入输出源响应，就不能仅使用一个阻塞的 `receive` 调用来接收来自单一源的输入。

MINIX 3 使用若干技术来处理这些问题。终端驱动程序处理键盘输入的主要技术是使中断响应尽可能快，以便使得字符不会丢失。可能的最小工作量是用来从键盘硬件获取字符到缓冲区。另外，当数据从键盘取出以响应一个中断且在返回中断以前，数据一旦被缓冲，键盘就会再次被读。中断产生通知消息，它不会阻塞发送者，这可以防止数据的丢失。虽然非阻塞的 `receive` 操作只是用于在系统崩溃时处理消息，但也是可以调用的。看门狗定时器也可以用于激活检查键盘的例程。

3.4.2 MINIX 3 的设备驱动程序

MINIX 3 系统中的每一类 I/O 设备都有一个单独的 I/O 设备驱动程序。这些驱动程序是完整的进程，每个都有其自己的状态、寄存器、堆栈等。设备驱动程序采用 MINIX 3 中所有进程使用的标准消息传递机制与文件系统进行通信。一个简单的设备驱动程序放在一个单独的源文件里。对于 RAM 盘、硬盘、软盘都各自有一个源文件来支持每种类型的设备，而支持所有块设备类型的通用例程则放在 `driver.c` 和 `drvlib.c` 中。这种把与设备相关部分和与设备无关部分分开的方法能够更灵活

地处理不同类型的硬件配置。尽管使用了一些公共的源码，但为了支持快速数据转移并把驱动器相互隔离，每种磁盘类型的驱动程序各自作为独立的进程运行。

终端驱动程序代码的组织方法与此类似，与硬件无关的代码放在 *tty.c* 中，而支持各种设备的代码则分别放在各自独占的文件中，包括内存映射的控制台、键盘、串口线以及虚拟终端等。但对终端而言，是一个单独的进程支持所有这些设备。

对于磁盘和终端这样的设备组，不仅有源文件，同时还有头文件。*driver.h* 支持所有的块设备驱动程序，*tty.h* 提供对所有串口设备的通用定义。

MINIX 3 的设计原则是用作为完全独立的用户空间的进程来运行操作系统的组件。这种设计原则是高度模块化和相当有效率的。这也是 MINIX 3 和 UNIX 在本质上的少数区别之一。在 MINIX 3 中进程通过向文件系统进程发送消息来读取一个文件，而文件系统则向磁盘驱动程序发送一条消息来请求其读取所需的数据块。磁盘驱动程序使用内核调用来请求系统任务完成实际的 I/O 操作并在进程之间复制数据。这一顺序（比现实稍微简化）如图 3.16(a) 所示。采用消息传递机制，可以强制系统的各部分按一种标准的方式同其他部分进行交互。

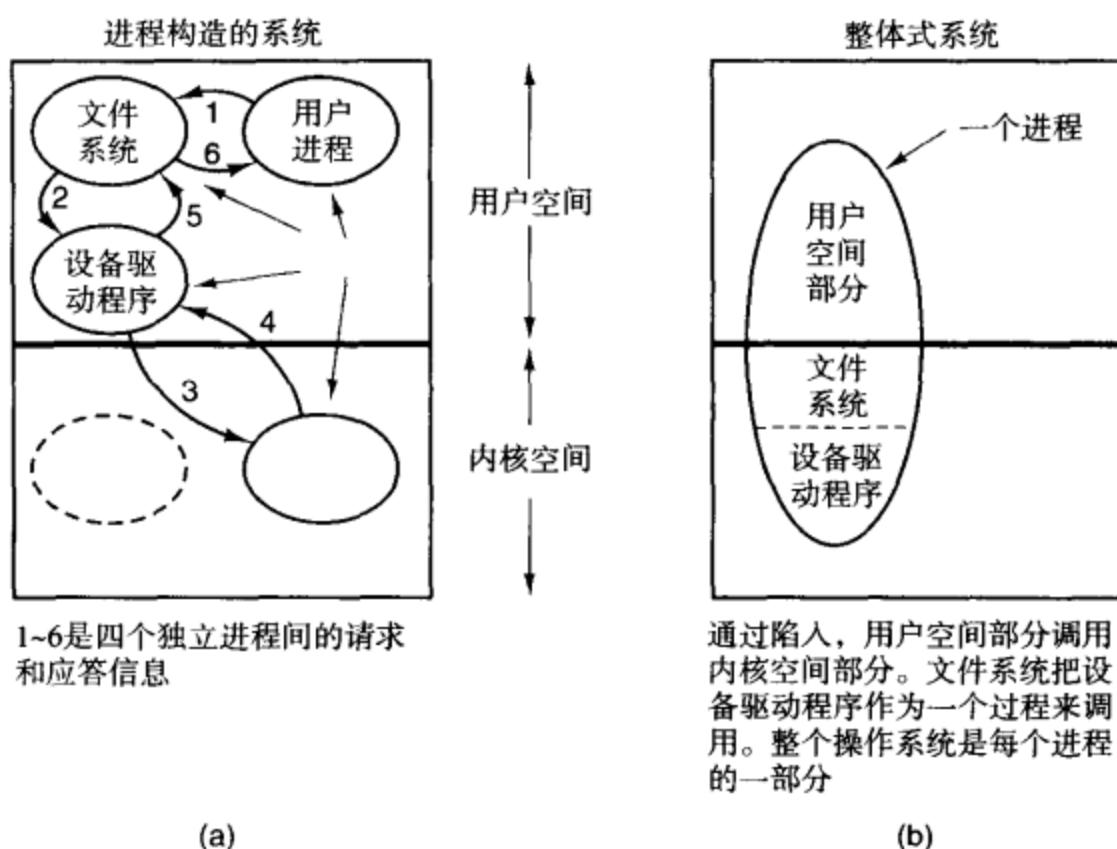


图 3.16 构造用户系统通信的两种方法

在 UNIX 中，所有进程都有两个部分：用户空间部分和内核空间部分，如图 3.16(b) 所示。当执行系统调用时，操作系统以一种特殊的方式从用户空间部分切换到内核空间部分，这种结构是 MULTICS 设计的遗留物。在 MULTICS 中，这种切换是普通的过程调用，而不是像 UNIX 那样陷入后将用户部分状态保存起来。

UNIX 中的设备驱动程序只是能被进程的内核空间部分调用的内核过程。当驱动程序需要等待一个中断时，它调用一个内核过程，从而使自己睡眠直到某一中断处理器将它唤醒。注意这里睡眠的只是用户进程本身，因为内核部分和用户部分是一个进程的完全不同的两部分。

在操作系统的设计师中，关于整体式系统（如 UNIX）和按进程构造的系统（如 MINIX 3）的争论永无休止。MINIX 3 的方法更加结构化（更模块化），各部分之间的接口更清晰，也更容易扩展到各进程运行于不同计算机的分布式系统上。而 UNIX 的方法则更高效，因为过程调用比消息传递要快得多。MINIX 3 被分成许多进程，因为可以相信随着计算机性能的提高，为了得到清晰的软

件，使系统稍微慢一些也是值得的。使操作系统的大部分运行于用户空间而造成的性能损失一般在5%~10%的范围内。需要注意的是，一些操作系统的设计者不认为牺牲一点速度来换取更模块化更稳定的系统是值得的。

本章将讨论RAM盘、硬盘、时钟和终端的驱动程序。标准的MINIX 3配置也包括软盘和打印机的驱动程序，但此处不予详细讨论。发布的MINIX 3软件中包含有其他一些设备的驱动程序的源码，如RS-232串行线、CD-ROM、各种以太网网卡以及声卡。这些程序可以在任何时候被编译并启动。

所有这些驱动程序接口与MINIX 3系统中的其他部分是一样的：将请求的消息发送给驱动程序。消息中包含的若干域用来存放操作码（例如READ或WRITE）和它的参数。驱动程序则试图执行收到的请求并返回应答消息。

块设备的请求和应答消息的域结构如图3.17所示。请求消息包括包含发送数据或者存放接收数据的缓冲区地址。应答包括相应的状态信息以使得请求的进程能够验证请求是否被正确地执行。字符设备的域与此基本类似，但各驱动程序之间可能略有不同。发给终端驱动程序的消息包含一个数据结构的地址，该数据结构中保存有终端的各种配置信息，如行编辑键、删除字符键和行删除键等。

请求		
域	类型	含义
m.m_type	int	操作请求
m.DEVICE	int	用于次设备
m.PROC_NR	int	进程请求I/O
m.COUNT	int	字节数或者控制码
m.POSITION	long	设备上的位置
m.ADDRESS	char*	请求进程中的地址

应答		
域	类型	含义
m.m_type	int	总是DRIVER_REPLY
m.REP_PROC_NR	int	与请求中的PROC_NR相同
m.REP_STATUS	int	已传送字节数或错误号

图3.17 文件系统向块设备驱动程序发送的消息的域和返回的应答消息的域

每个设备驱动程序的作用是接收其他进程发来的请求并执行之，通常请求来自文件系统。所有的块设备驱动程序都是首先获取消息，然后执行和返回应答信息。这个结果意味着，为简单起见，这些驱动程序是完全串行的，不包含任何内部的多道程序设计。当发出一个硬件请求时，驱动程序执行一个receive操作，指明它只接收中断消息，而不是新的工作请求。任何新来的请求消息只是保持等待直到当前的工作结束为止（聚合原则）。终端驱动程序略有不同，因为单个任务要为若干设备服务，所以可以在从串行线上读数据的同时接收从键盘输入的新请求。但每一个设备都必须执行完上一个请求后才能开始另一个新的请求。

所有块设备驱动程序中的主程序都如图3.18所示。当系统启动时，所有的驱动程序被启动，轮流进行内部数据结构的初始化，然后驱动程序试图获取消息并被阻塞。当收到一条消息后，先将调

用者标识保存下来，然后调用过程来执行请求，对不同操作有不同的过程可供调用。当工作结束后，则向调用者发回应答，然后驱动程序返回循环的顶部来等待下一个请求。

```

message mess;                                /* 消息缓冲区 */

void io_driver( ) {
    initialize ( );                          /* 只在系统初始化期间做一次 */
    while (TRUE) {
        receive(ANY &mess);                /* 等待一个请求 */
        caller = mess.source              /* 消息来源 */
        switch(mess.type) {
            case READ:                   rcode = dev_read(&mess); break;
            case WRITE:                  rcode = dev_write(&mess); break;
            /* 这里还有其他分支，如 OPEN, CLOSE 及 IOCTL */
            default:                    rcode = ERROR;
        }
        mess.type = DRIVER_REPLY;
        mess.status = rcode;             /* 执行结果代码 */
        send(caller, &mess);           /* 向调用者发送应答消息 */
    }
}

```

图 3.18 一个 I/O 设备驱动程序的主过程的框架

设备驱动程序中的每一个 *dev_XXX* 过程执行一种该驱动程序能做的操作。它返回一个状态码，表明发生了什么。该状态码包含在应答消息内的 *REP_STATUS* 域中，为零或正值时指明传输的字节数，若其值为负，则表示这是一个错误码。该值有可能与请求的字节数不等。如遇到文件尾时，所得到的字节数就可能小于请求的字节数。在终端上，即使请求的数量再多，最多也只能返回一行的字符数（除非是在生模式）。

3.4.3 MINIX 3 中与设备无关的 I/O 软件

在 MINIX 3 中，所有与设备无关的软件都包含在文件系统进程中。因为 I/O 系统与文件系统的联系非常紧密，所以它们也可以合并到一个进程中。文件系统执行的功能如图 3.6 所示，其中不包括对独占设备的请求和释放。在 MINIX 3 的目前配置中不包含对独占设备的支持。但在将来需要时可以很容易地加到相关设备的驱动程序中。

除了处理与驱动程序、缓冲、数据块分配的接口，文件系统还处理对 i 节点、目录、挂装的文件系统的保护和管理等。文件系统将在第 5 章详细讨论。

3.4.4 MINIX 3 中的用户级 I/O 软件

本章前边给出的总体框架在这里也适用。为了执行系统调用和所有 POSIX 标准要求的 C 函数，系统提供了库过程，例如格式化输入和输出函数 *printf* 和 *scanf*。标准的 MINIX 3 配置包括一个假脱机守护进程 *lpd*，它处理所有通过 *lp* 命令向它提交的打印文件。发布的标准 MINIX 3 软件也提供了许多支持网络操作的守护进程，这需要在启动时启用网络服务和网卡的驱动程序。重新编译带有虚拟终端和串口线支持的终端驱动程序，可以增加对来自远程终端和通过串口线（包括调制解调器）的网络的登录支持。网络服务器以和内存管理器、文件系统相同的优先级运行，而且像它们一样作为用户进程运行。

3.4.5 MINIX 3 的死锁处理

对于本章前面提到的各种类型的死锁，MINIX 3 沿用和 UNIX 相同的方式：仅仅简单地忽略问题，这完全是 UNIX 的继承。尽管如果某人想把一台工业标准 DAT 磁带机装到 PC 上，并为它开发软件，这不会带来任何特别的问题，但是，通常 MINIX 3 不包含任何独占的 I/O 设备。简而言之，可能发生死锁的唯一环节是在使用隐含的共享资源时，例如进程表项、i 节点表项等，没有一种已知的死锁算法能够解决这种非显式请求资源的问题。

实际上，上面的叙述并不完全正确。接受用户进程可能产生死锁的风险是一方面，但是在操作系统本身内也确实存在一些仔细考虑以避免发生死锁的地方。主要的一点就是进程之间的消息传递交互。例如，用户进程只被允许使用 `sendrec` 通信方法，所以用户进程永远也不会查找，因为它在没有进程想向它发送消息时完成一个 `receive` 操作。服务器只使用 `send` 或者 `sendrec` 来和设备驱动程序交互，而设备驱动程序只用 `send` 或者 `sendrec` 和内核层的系统任务交互。在很少的情况下，必须要服务器在它们之间交互，比如进程管理器和文件系统之间的交互，就像进程初始化进程表中的它们的那部分，交互的顺序被认真地设计以避免死锁。同样，在消息传递系统的最低层也有一个检查以确保一个进程想要完成 `send` 操作时，目的进程没有在尝试做同样的事情。

除了上面的限制，在 MINIX 3 中提供了新的 `notify` 消息原型以处理那些消息一定要向“上游”方向发送的情况。`notify` 是非阻塞的，接收者暂时不可用时通知会被保存。在本章中考察 MINIX 3 的设备驱动程序实现时，将会看到 `notify` 被广泛地使用。

锁（locks）是可以防止死锁的另一种机制。即使没有操作系统的支持也可以对设备和文件加锁，文件名可以作为真正的全局变量来使用，其存在与否可以被所有其他进程感知。在 MINIX 3 和大多数类 UNIX 系统中都存在一个特殊的目录 `/usr/spool/locks/`，供进程创建加锁文件，以标识其正在使用的资源。MINIX 3 文件系统也支持 POSIX 风格的建议文件锁（advisory file locking）。但是这些机制都不是强制性的，它们取决于进程的良好行为，而且没有什么可以阻止一个进程使用由另一个进程加锁的资源。这与资源的抢占并不完全一样，它并不阻止第一个进程继续使用该资源，即不存在互斥访问。进程的这种不正常行为可能会导致一片混乱，但不会导致死锁。

3.5 MINIX 3 中的块设备

MINIX 3 支持若干不同的块设备，所以这里首先讨论所有块设备的共性部分。然后将讨论 RAM 盘、硬盘和软盘。其中的每一个都由于有不同的原因而很值得研究。RAM 盘是一个很好的用来研究的例子，它具有除了实际 I/O 以外的块设备的所有共性，因为这个“盘”实际上只是内存的一部分。RAM 盘的简单性使其成为一个好的出发点。硬盘展示了真实的磁盘驱动程序的特征。有人会误认为软盘比硬盘容易一些，但事实上不是这样。这里不会讨论软盘的所有细节，但将指出软盘驱动程序的复杂之处。

在讨论块设备之后，我们将讨论终端（键盘+显示器）的驱动程序，它在所有的系统中都很重要，而且是典型的字符设备。

以下每一部分都将描述相关的硬件、驱动程序背后的有关软件原理、实现的概述以及代码本身，这种结构将使这部分内容对那些对代码本身细节不感兴趣的读者也颇有价值。

3.5.1 MINIX 3 中的块设备驱动程序概述

前面提到过，所有 I/O 设备驱动程序的主程序都有类似的结构。MINIX 3 总是至少有两种设备驱动程序被编译到系统中：一个是 RAM 盘驱动程序，另一个是若干可能的硬盘驱动程序中的一个

或者是软盘驱动程序。通常，有三个块设备包括软盘驱动程序和 IDE (Integrated Drive Electronics) 硬盘驱动程序。每个块设备的驱动程序会被单独编译，但所有这些驱动程序会共享公共的源码库。

在 MINIX 的老版本中，有时会出现一个单独的 CD-ROM 驱动程序，如果必要的时候可以被添加。单独的 CD-ROM 驱动程序现在已经陈旧了。过去它们必须支持不同驱动器厂商的所有接口，尽管笔记本电脑上的某些 CD-ROM 是通过 USB 接口连接的，但现代 CD-ROM 驱动器通常还是连接到 IDE 控制器上。MINIX 3 的硬盘驱动程序的完整版本包括对 CD-ROM 的支持，但是对 CD-ROM 的支持已经从正文中描述的和附录 B 中列出的驱动程序中去除了。

当然，每个块设备驱动程序都要做一些初始化。RAM 盘驱动程序要保留一些内存，硬盘驱动程序要确定硬盘的硬件参数，等等。所有磁盘驱动程序在相应的硬件初始化时都被调用。在完成必需的工作后，每个驱动程序将调用包含公共主循环的函数。该循环将一直执行下去，不会返回到调用者。在主循环中，接收一条消息，调用能执行每条消息所操作的函数，然后产生一条应答消息。

各驱动程序进程调用的公共主循环在 *drivers/libdriver/driver.c* 和这个目录的其他文件被编译时也会被编译，然后目标文件 *driver.o* 的一个副本被链接到每个设备驱动程序的可执行文件中。它使用的技术是使各驱动程序向主循环传递一个参数，该参数是指向一个表的指针。该表中包含了完成各具体操作的函数指针，于是以后就可以间接调用这些函数。

如果驱动程序和一个单一的可执行文件一起编译，则只需要一个主循环的副本。事实上，这个代码首先是为 MINIX 的早期版本写的，在早期的 MINIX 中，所有的驱动程序是被编译在一起的。MINIX 3 的重点在于使操作系统的各组件尽可能地独立，但是各独立的程序使用公共源码仍然是增加稳定性的一种好方法。假如能一次就使代码正确，那么它对所有的驱动程序都是正确的。或者，在一次使用中发现的缺陷也存在于没有注意到的其他使用中。因此，共享源码需要更彻底的测试。

在 *drivers/libdriver/drplib.c* 中定义的许多其他函数也潜在地对多磁盘的驱动程序有用，并且链接 *drplib.o* 文件使得这些函数可以调用。所有的功能都已经在一个单独的文件中提供，但并不是每个磁盘驱动程序都需要所有这些功能。比如，内存驱动程序要比其他设备驱动程序更简单，它只链接 *driver.o* 文件。*at_wini* 链接 *driver.o* 和 *drplib.o* 文件。

图 3.19 给出了主循环的框架，其格式类似于图 3.18。语句

```
code = (*entry_points->dev_read) (&mess);
```

是间接函数调用，尽管各驱动程序执行从同一源文件编译的主循环，但每个驱动程序调用不同的 *dev_read* 函数。但有些其他的操作（如 *close*）非常简单，可供多个设备共享。

任何设备驱动程序都有六种可能的操作可以请求，它们对应于图 3.17 所示的消息结构中 *m.m_type* 域的各种可能值，它们分别是

1. OPEN
2. CLOSE
3. READ
4. WRITE
5. IOCTL
6. SCATTERED_IO

有编程经验的读者对这些操作中的大部分可能很熟悉。在设备驱动程序层，大多数操作与同名的系统调用相关。例如 *READ* 和 *WRITE* 的意义就非常清楚。对于每个这样的操作，一个数据块被

从设备读到启动调用的进程的内存，或者正好相反。*READ*操作通常在数据传送完后才向调用进程返回，但是对于*WRITE*而言，操作系统有可能将数据暂存在缓冲区中，随后再真正将其传送到设备，这时*WRITE*系统调用将立即向调用进程返回。这对调用进程很有利，因为它随后可以再次使用操作系统暂存的写出数据的缓冲区。设备的*OPEN*和*CLOSE*的含义与用于文件操作的系统调用*open*和*close*类似：*OPEN*操作将验证设备是否可用，当不可用时则返回一条错误消息；*CLOSE*将确保任何被缓冲的调用进程写的数据完全传输到设备上它们的最终目标。

```

message mess;                                /* 消息缓冲区 */

void shared_io_driver(struct driver_table *entry_points){
/* 每个任务调用本过程之前先完成初始化 */
    while(TRUE){
        receive(ANY, &mess);
        caller = mess.source;
        switch(mess.type){
            case READ:   rcode = (*entry_points->dev_read)(&mess); break;
            case WRITE:  rcode = (*entry_points->dev_write)(&mess); break;
            /* 这里还有其他分支，如 OPEN, CLOSE 及 IOCTL */
            default:     rcode = ERROR;
        }
        mess.type = DRIVER_REPLY;
        mess.status = rcode;                      /* 执行结果代码 */
        send ( caller, &mess);
    }
}

```

图 3.19 一个使用间接调用的 I/O 驱动程序的主过程

读者对*IOCTL*可能不大熟悉。许多I/O设备都有一些操作参数，经常会需要对这些参数进行检查，也可能要修改。*IOCTL*的任务就是做这些工作。常见的一个例子是改变通信线路的传输速率和奇偶校验方式。对于块设备，*IOCTL*操作不太常用，在MINIX 3中，检查或者改变磁盘设备的分区是用*IOCTL*操作完成的（尽管也可以通过读写数据块完成）。

*SCATTERED_IO*操作无疑是这些操作中最少见的。除了个别非常快的磁盘设备（如RAM盘）外，如果每次只请求读写一块，那么其I/O性能很难提高。一个*SCATTERED_IO*允许文件系统请求读写多个块。对于*READ*来说，要求多读的块可能并不是由执行该调用的进程所请求的，而只是因为操作系统试图预测将来对数据的请求。这种请求并不一定由设备驱动程序实现。对每一个块的请求可以用一个标志位进行修改，它通知驱动程序该请求是可选的。实际上文件系统可以声明：“最好将这些数据全部都拿来，但实际上并非马上就需要”。设备可以据此见机行事。例如软磁盘驱动程序将返回整条磁道上的数据块，相当于“我把这些数据给你，但移动到另一条磁道上操作太费时了，在你需要时再通知我。”

当数据必须被写时，则不存在可选的问题，每次写操作都是强制的。但操作系统可能会缓冲许多写请求，随后一次写出，这比每次处理单个请求的效率要高。在*SCATTERED_IO*请求中，将请求读写块的请求排序，这比随机地处理请求更有效率。而且在MINIX 3中，只调用一次驱动程序来传输多个块减少了需要发送的消息数量。

3.5.2 通用块设备驱动程序软件

所有块设备驱动程序需要的定义都放在 *drivers/libdriver/driver.h* 中。这个文件中最重要的是 *driver* 结构（10 829 到 10 845 行），其中保存了各驱动程序执行具体 I/O 操作的函数地址。该文件还定义了 *device* 结构（10 856 到 10 859 行）。其中保存了与分区相关的最主要信息：基地址和大小，它们都以字节为单位。采用这种格式使得对基于内存的设备无须做任何转换，由此最大程度地提高了响应速度。而对于真正的磁盘，由于有很多因素影响延迟访问，因而转换到扇区地址并不会增加很多麻烦。

所有块设备驱动程序共享的主循环及其他公用函数的源码都在 *driver.c* 中。在具体硬件执行完必要的初始化后，每个驱动程序都调用 *driver_task*，同时向其传入一个 *driver* 结构作为调用参数。在获得一个供 DMA 操作使用的缓冲区地址后进入主循环（11 071 到 11 120 行）。

在主循环内的 *switch* 语句中，前五种消息类型 *DEV_OPEN*, *DEV_CLOSE*, *DEV_IOCTL*, *DEV_CANCEL* 和 *DEV_SELECT* 将使用 *driver* 结构传来的地址进行间接调用。而 *DEV_READ* 和 *DEV_WRITE* 消息都将直接调用 *do_rdwt*; *DEV_GATHER* 和 *DEV_SCATTER* 消息都将直接调用 *do_vrdwt*。不论是直接还是间接，*driver* 数据结构都被 *switch* 中的所有调用作为一个参数传递，所以，所有的被调用函数在需要时都可以进一步使用。*do_rdwt* 和 *do_vrdwt* 会做一些预处理，但然后它们也将间接调用设备相关的例程。

对于其他情况，如 *HARD_INT*, *SYS_SIG* 和 *SYN_ALARM*，会响应通知（*notifications*）。这些也会导致间接调用，但是在完成上，每个都要执行一个 *continue* 语句。这会使控制返回循环的顶部，跳过清除和应答消息两步操作。

按消息中的请求进行处理后，根据设备本身的特性需要做一些清理操作。例如，对于软盘，可能会启动一个定时器，以便在一段时间内下一个请求未达到的情况下关闭驱动器的电机。间接调用也可用于此目的。在清理操作之后，将构造一个应答消息并传送给调用者（11 113 到 11 119 行）。服务于某种消息类型的例程可能会返回一个 *EDONTREPLY* 值来禁止应答消息，但当前的驱动程序没有一个使用这个选项。

在进入主循环后，各驱动程序执行的第一个操作是调用 *init_buffer*（11 126 行），它会分配一个用于 DMA 操作的缓冲区。这段初始化操作之所以必要，是由于最初的 IBM PC 机硬件的一个特殊之处，即 PC 机硬件要求 DMA 缓冲区不得越过 64 KB 边界。也就是说，1 KB 大小的 DMA 缓冲区应从 64 510 而不是 64 514 开始，因为后者正好越过 65 536 的 64 KB 边界。

这种讨厌的规则源于 IBM PC 使用一种老式的 DMA 控制器芯片 Intel 8237A，它含有一个 16 位的计数器。由于 DMA 使用绝对地址，而不是相对于段寄存器的偏移地址，所以需要一个更大的计数器。在老式的只能寻址 1 MB 的机器上，DMA 地址的低 16 位被装入 8237A，高 4 位则被装入一个 4 位的锁存器。新一些的机器使用 8 位的锁存器，并能寻址 16 MB。当 8237A 由 0xFFFF 变到 0x0000 时，并不向锁存器进位，从而导致 DMA 地址在内存中突然减掉了 64 KB。

可移植的 C 程序不能为一个数据结构指定绝对的内存地址，所以无法防止编译器将缓冲区放在一个不可用的位置。解决办法是分配一片大小为所需缓冲区两倍的内存 *buffer*（11 044 行），并保留一个指针 *tmp_buf*（11 045 行）供对这片内存的实际访问。*init_buffer* 首先尝试将 *tmp_buf* 指向 *buffer* 的开头，然后检查它在遇到 64 KB 边界之前是否能够提供足够的空间。如果不行，则 *tmp_buf* 就递增真正所需空间的字节数。这样，在 *buffer* 的某一段总会造成一些地址的浪费，但也保证了不会由于 64 KB 边界的限制而造成缓冲区失败。

更新的IBM PC系列计算机使用更好的DMA控制器，所以这部分代码可以简化。而且，如果所用的机器肯定不存在上述问题，那么只请求一小部分内存便可以满足要求。如果确实这样做了，假定判断有误，会发生什么情况呢？对于1 KB大小的DMA缓冲区，在使用老式的DMA控制器时，出错的概率是1/64。每次内核代码被修改，导致编译后的内核大小发生变化时，问题会以同样的概率出现。当这个错误在下一个月或次年发生时，很可能归咎于最后一次修改的代码。由于类似的不可预知的硬件“特征”，可能需要花几周的时间来查找那些隐藏很深的错误（对于那些与本例类似的、在技术手册中从未提到的错误，可能会花更多的时间）。

do_rdwt (11 148行) 是 *driver.c* 中的下一个函数。它轮流调用两个与设备相关的函数，*driver* 结构中的 *dr_prepare* 和 *dr_transfer* 域分别指向这两个函数。下面将按照C语言的记法，用 **function_pointer* 表示 *function_pointer* 指向的函数。

在检查请求中的字节计数是正数后，*do_rdwt* 调用 **dr_prepare*。该操作填写正在被访问的分区或子分区的 *device* 结构中的磁盘基地址和大小。对于不支持分区的内存驱动程序，它只是检查次设备号是否有效。对于硬盘，它使用次设备号来获取这个次设备号所指定的分区或子分区的大小。**dr_prepare* 一定会成功，因为只有当 *open* 操作指定了一个无效设备时 **dr_prepare* 才会失败。接下来将填写 *iovec_t* 结构（在 *include/minix/type.h* 中的 2856 到 2859 行定义），即 *iovec1*，这个结构指定了局部缓冲区的虚拟地址和大小，或者系统任务需要把数据复制到何处。同样是这个结构，当调用针对多个块时，它被作为一组请求中的一个元素。变量的地址和同类型变量数组的第一个元素都可按完全相同的方法处理。然后是一个间接调用 **dr_transfer*，它可以完成需要的数据复制和I/O操作。所有处理传送的例程都期望接收一组请求。*do_rdwt* 中的最后一个参数是 1，它指定了一个队列中只有一个元素。

正如我们将要在下一节中对磁盘的讨论中看到的，按照请求的接收顺序来响应是没有效率的，并且这个例程允许一个特殊的设备以对设备最好的方式来处理请求。这里隐藏了许多各设备执行方式之间可能的变化。对于RAM盘，如果正在访问的次设备是 */dev/ram*, */dev/mem*, */dev/kmem*, */dev/boot* 或 */dev/zero*（当然没有复制需要访问 */dev/null*），那么 *dr_transfer* 指向一段代码，这段代码使用内核调用来请求系统任务，把数据从物理内存的一部分复制到另一部分。对于真正的磁盘，*dr_transfer* 指向的代码也必须要请求数据传送的系统任务。但是在复制操作之前（对于读）或者在复制操作之后（对于写），内核调用也必须要请求系统任务来完成实际的I/O，通过向磁盘控制器的寄存器写字节来选择磁盘上的位置和大小以及传输的方向。

在执行数据传输的例程过程中，*iovec1* 结构中的 *iov_size* 域将改变。返回值为负，表明出错；返回值为正，则表明请求的字节数和成功传送的字节数之间的差值。即使没有发生字节传送也并不表示一定是出错，它只是表明到了设备的结尾。当返回主循环时，错误码或者字节计数放在由 *driver_task* 返回的应答消息的 *REP_STATUS* 域里。

下一个函数 *do_vrdwt* (11 182行) 处理所有散布的I/O请求。请求一个散布I/O的消息使用 *ADDRESS* 域来指向一个 *iovec_t* 结构的数组，其中的每一项指定了缓冲区地址和要传送的字节数。在MINIX 3中，这样一个请求只能对磁盘上相邻的块调用，设备上的初始偏移和这个操作是读还是写都包含在消息里。所以一个请求中的全部操作或者都是读，或者都是写，而且将按照它们在设备上的数据块顺序排序。在11 198行完成一个检查，看这个代表内核空间的I/O任务的调用是否已完成；这是MINIX 3发展的早期阶段的一个痕迹，是在为使所有磁盘驱动程序能够在用户空间运行而被重写以前的一个阶段。

这部分代码从根本上和 *do_rdwt* 执行简单的读写操作是十分类似的。它们会间接调用相同的设备相关例程 (**dr_prepare*) 和 (**dr_transfer*)。为了处理多个请求，主循环在内部让 **dr_transfer*

完成所有的工作。这种情况下，最后一个参数不再是1，而是`iovec_t`数组的元素的个数。在循环终止以后，请求的队列被复制回发出请求的地方。数组中每个元素的`io_size`域将表明那个请求要传送的字节数，尽管总字节数没有放在`driver_task`构造的应答消息中直接传回，但是调用进程能够从这个数组中提取出总数。

在`driver.c`中的随后几个例程都是为以上操作提供通用支持的。`*dr_name`可以用于返回设备的名字。对于没有给定名字的设备，`no_name`函数返回字符串“noname”。有些设备可能不需要特定的服务，如RAM盘在响应`DEV_CLOSE`请求时并不要求做任何特别的操作，这时就使用`do_nop`函数，它仅仅根据请求的类别返回不同的码值。其他的函数`nop_signal`, `nop_alarm`, `nop_prepare`, `nop_cleanup`和`nop_cancel`是类似的不需要这些服务的设备的哑例程（dummy routine）。

最后，`do_ioctl`（11 216行）执行块设备的`DEV_IOCTL`请求。`DEV_IOCTL`请求只能是读(`DIOCGETP`)或写(`DIOCSETP`)分区信息，除此之外都会出错。`do_ioctl`调用设备的`*dr_prepare`函数来验证设备是否有效，并得到一个指向`device`结构的指针，该结构描述了按字节计数的分区基地址和大小。对于读请求，它调用设备的`*dr_geometry`函数来获得该分区的最后柱面号、磁头及扇区信息。在每种情况下都需要调用内核调用`sys_datacopy`来请求系统任务在驱动程序的内存空间和请求进程的内存空间之间复制数据。

3.5.3 驱动程序库

文件`drvlib.h`和`drvlib.c`包含一些与系统有关的代码，这些代码支持IBM PC及兼容机的磁盘分区。

分区机制允许单个存储设备被分成若干子设备，它主要用于硬盘，但MINIX 3还提供对软盘的分区。进行磁盘分区的理由是：

1. 大容量磁盘单位价格便宜。如果有两个或更多的操作系统在使用不同的文件系统，则使用一个大硬盘分区要比为每个操作系统安装较小的硬盘更经济。
2. 操作系统对于能够处理的设备的大小是有限制的。这里讨论的MINIX 3版本能够处理4 GB的文件系统，但是老的版本只能局限于256 MB，任何多于256 MB的磁盘空间都会被浪费。
3. 一个操作系统可能使用两个或更多的不同文件系统。例如，标准的文件系统用于普通文件，另一个不同结构的文件系统用做虚拟内存的交换区。
4. 将一个系统的文件的一部分放在一个独立的逻辑设备上可能方便一些。将MINIX 3根文件系统放在一个小的设备上将使其更便于备份，而且有助于在引导时将其复制到RAM盘中。

磁盘分区的支持是与平台有关的，这种特定性与硬件没有关系。分区支持独立于设备，但如果一台设备上有多个操作系统运行，则它们必须就分区表的格式达成一致。在IBM PC上，该标准由MS-DOS的`fdisk`命令确定。其他操作系统，如MINIX 3, Windows以及Linux也使用该命令，以便与MS-DOS共存。当MINIX 3被移植到另一种类型的机器上时，应该使用与在这种新硬件上运行的其他操作系统兼容的分区表格式。所以在MINIX 3中支持IBM计算机分区的源码部分放在`drvlib.c`中，而不是放在`driver.c`中，原因有如下两个：首先，并不是所有的磁盘类型都支持分区。正如前面提到的，内存驱动程序链接了`driver.o`但不需要使用编译到`drvlib.o`中的函数。其实，这是为了更容易地把MINIX 3移植到其他硬件上。替换一个小的文件比编辑一个许多部分都要针对不同环境进行条件编译的大文件要更容易。

从硬件设计人员那里继承过来的基本数据结构定义在`include/ibm/partition.h`中，它通过`#include`语句包含在`drvlib.h`中（10 900行）。其中包含每个分区的柱面-磁头-扇区格式的信息，以

及标识文件系统类型的代码和一个分区是否可引导的激活标志。对文件系统进行了检查之后，其中多数信息便不再被 MINIX 3 需要。

第一次打开块设备时，将调用 *partition* 函数 (*drvlib.c*, 11 426 行)。其参数包括一个 *driver* 结构（这样它便可以调用与设备相关的函数）、一个初始的次设备号以及另一个参数，该参数标识分区类型是软盘、主分区或次分区。*partition* 函数调用设备相关的 **dr_prepare* 函数来验证设备是否有效，并获得前面一节提到的 *device* 结构中的基地址和大小。然后它调用 *get_part_table* 来判断分区表是否存在。若分区表存在则将其读入，若分区表不存在则结束操作。否则，用最初调用时指定的分区类型所使用的分区编号规则计算第一个分区的次设备号。在使用主分区的情况下，分区表是经过排序的，所以分区的顺序与其他操作系统使用的顺序相同。

在这里将第二次调用 **dr_prepare*，这次使用刚计算出来的第一个分区的设备号。如果子设备有效，则对表中所有项执行一个循环，每次检查从设备上的表中读出的值是否超越先前获得的整个设备的基地址和范围。如果出现不一致，则修改内存中的表格，以取得一致。这似乎不太合适，但由于分区表可能被不同的操作系统修改，使用另一个操作系统的程序员可能很聪明地试图利用分区表做一些不可预知的事情。或者由于其他的原因，磁盘上的分区表中可能会有垃圾信息，所以要更相信使用 MINIX 3 计算出来的数值。这里的思想是，安全总比出错要好。

还是在这个循环中，对于设备上所有标识为 MINIX 3 的分区，递归调用 *partition* 函数以获取其子分区信息。如果分区被标识为一个扩展分区，则调用 *drvlib.c* 中的下一个函数。

extpartition (11 501 行) 实际上与 MINIX 3 本身无关，所以这里不进行详细讨论。其他的一些操作系统使用扩展分区，比如 Windows。它们使用链表而不是固定大小的数组来支持子分区。为简单起见，MINIX 3 对主分区使用了相同的子分区机制。然而，对扩展分区提供的最小支持是 MINIX 3 命令能读写其他操作系统的文件和目录。这些操作很容易，但提供对挂装或者和主分区一样的方式来使用扩展分区的支持将复杂得多。

get_part_table (11 549 行) 调用 *do_rdwt* 从保存分区表的设备（或子设备）上获取一个扇区。如果它用来获取一个主分区，则偏移参数为 0；如果用来获取一个次分区，则偏移为非 0。它检查分区表的魔数 (0xaa55)，并返回一个真或假的状态，以表明是否找到了有效的分区表。若找到一个分区表，则将其复制到由输入参数指定的表地址处。

最后，*sort* 函数 (11 582 行) 从最低扇区号开始将分区表中的表项排序。标识为不包含分区的表项则不参与排序，所以被排在最后，即使其最低扇区号为 0。这种排序只是简单的冒泡排序，对于一个只有 4 项的表，没有必要使用特别优化的算法。

3.6 RAM 盘

现在回到独立的块设备驱动程序，并对其中的几个进行详细研究。第一个要研究的是存储器驱动程序，利用它可以访问存储器的任何部分，它的主要用途是可以保留一部分存储器以像普通磁盘一样来使用，一般情况下称之为 RAM 盘驱动程序。RAM 盘并不提供永久存储，但是一旦文件被复制到这一区域，就能够以极快的速度进行访问。

当在一个仅有可移动存储设备的计算机上初次安装一个操作系统时，RAM 盘非常有用，无论这个可移动设备是软盘驱动器、CD-ROM 还是其他什么设备。通过把根文件设备放在 RAM 盘上，就可以随意地挂装和卸载可移动设备，从而可以将数据转移到硬盘上。因为不能卸载根设备，所以如果把根文件系统放在软盘上，就不能将文件保存在软盘上。RAM 盘还可以用来作为“活动” CD-ROM，这样就可以在不复制任何文件到硬盘的情况下，为了测试或者演示目的运行一个操作系

统。另外，把根文件系统放到 RAM 盘可以使系统具有很大的灵活性：软盘和硬盘的任意组合都能挂装上去。在发行的 CD-ROM 中包含了 MINIX 3 以及其他许多操作系统。

接下来马上就会发现：存储器驱动程序提供了许多除 RAM 盘之外的其他功能。通过它就可以直接对任何内存空间进行随机访问，逐字节地访问或者将任何大小的内存作为一个块访问。这种方式使得它不像块设备而更加类似于字符设备。存储器驱动程序提供的其他字符设备有 `/dev/zero` 和 `/dev/null`，以及在 sky 中的大位桶（great bit bucket）。

3.6.1 RAM 盘硬件和软件

RAM 盘的思想很简单。块设备是具有两个操作命令的存储介质，即写数据块和读数据块。通常这些数据块存储在旋转存储设备上，例如软盘和硬盘。RAM 盘则简单得多，它使用预先分配的主存来存储数据块。RAM 盘具有快速存取的优点（没有寻道和旋转延迟），适于存储需要频繁存取的程序和数据。

有些系统支持可挂装的文件系统，有些则不支持（如 MS-DOS 和 Windows），这里简略地指出它们的区别。利用挂装文件系统，可使得根设备总是位于固定的位置，可移动的文件系统（即磁盘）可以挂装到文件树上，从而构成一个完整的文件系统。一旦挂装完毕，用户就不必关心文件在哪个设备上。

相反，对于像 MS-DOS 这样的系统，用户必须指定每个文件的位置，或者以 `B:\DIR\FILE` 格式显式地指出，或者使用一定的默认值（当前设备、当前目录等）。在仅有两个软盘驱动器时，管理的负担还可以承受。但对于有几十个磁盘的大型计算机系统而言，要在任何时刻跟踪所有的设备，其负担是无法忍受的。请注意，类 UNIX 操作系统可以运行在家庭和办公等场合的小型计算机上，也可以运行在诸如 IBM Blue Gene/L（在写这本书之前世界上最快的计算机）的巨型计算机上，而 MS-DOS 仅在小系统上运行。

图 3.20 给出了实现 RAM 盘的思想。根据为 RAM 盘分配的内存大小，RAM 盘被分为 n 块，每块的大小和实际磁盘块的大小相同。当驱动程序接收到一条读写数据块的消息时，它只计算被请求的块在 RAM 盘对应存储区中的位置，并读出或写入该块，而不读写实际的软盘或硬盘。最终调用系统任务以完成最后的传输。数据的传输通过调用内核中一个名为 `phys_copy` 的汇编语言过程来实现，该过程以硬件所能实现的最高速度把数据复制到用户程序或从用户程序复制出来。

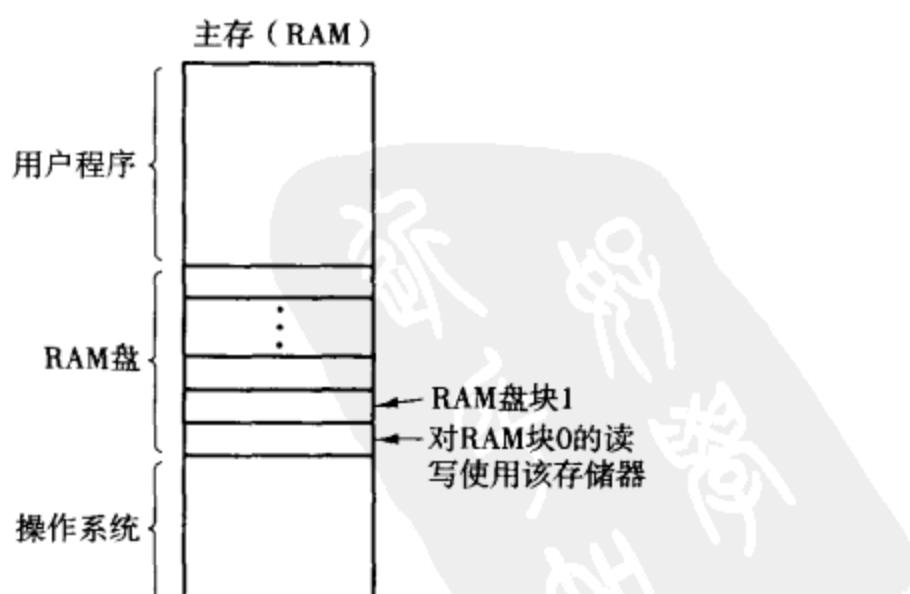


图 3.20 一个 RAM 盘

一个RAM盘驱动程序可以支持将存储器中的若干区域当做RAM盘来使用，每个RAM盘用次设备号来区分。一般情况下这些存储区相互分开，但正如下一小节将看到的，在某些情况下，使它们相互重叠可能会更方便。

3.6.2 MINIX 3 中的 RAM 盘驱动程序概述

在 MINIX 3 中，RAM 盘驱动程序实际上是互为一体的 6 个紧密相关的驱动程序，传给驱动程序的每条消息都要指定下列次设备之一：

0: /dev/ram	2: /dev/kmem	4: /dev/boot
1: /dev/mem	4: /dev/null	5: /dev/zero

其中，第 1 个文件 */dev/ram* 是真正的 RAM 盘，驱动程序内部既不指定它的大小，也不指定它的起始地址，这些是在 MINIX 3 启动时由文件系统确定的。如果启动参数指定根文件系统将被放置到 RAM 盘上，但是 RAM 盘的大小没有指定，则默认情况下将创建一个与根文件系统映像设备大小相同的 RAM 盘，以便将根文件系统复制到 RAM 盘。通过指定启动参数，可以创建一个容量大于根文件系统的 RAM 盘；如果不把根文件系统复制到 RAM 盘，则在为系统操作留下足够内存的前提下，RAM 盘的容量可以取内存可容纳的任意值。一旦 RAM 盘大小确定，则系统就寻找一块足够大的内存，并在内存管理器初始化时将其从内存池中移出。这种策略使得不必重新编译操作系统即可增加或减少 RAM 盘的容量。

随后的两个次设备分别用于读写物理内存和内核内存。当打开 */dev/mem* 并进行读操作时，读出的是起始于绝对地址零的物理内存中的内容（实模式下的中断向量）。普通用户程序永远不会执行这个操作，但与系统调试有关的程序可能会使用这种功能。打开并对 */dev/mem* 执行写操作将修改中断向量。显然，只有那些对该操作的结果十分清楚的熟练用户才可以非常谨慎地执行这个操作。

设备文件 */dev/kmem* 和 */dev/mem* 相似，不过其第 0 字节是内核数据存储区的第 0 字节，而其绝对地址随 MINIX 3 内核代码的大小而变。它也是主要用于调试以及非常特殊的程序。注意，这两个次设备覆盖的存储区是重叠的，如果能够精确地知道内核在内存中如何放置，则可以打开 */dev/mem* 文件，将文件指针定位到内核数据区的起点，并会发现这里的数据和从文件 */dev/kmem* 开头读出的数据相同。但如果重新编译内核并改变其大小，或者如果在 MINIX 3 的后续版本中内核被放在内存的其他位置，那么只有把指针定位到 */dev/mem* 的另一个位置时才能发现和 */dev/kmem* 起始处相同的内容。对这两个文件必须采取保护措施，以保证超级用户之外的其他用户不能使用它们。

下一个文件 */dev/null* 是一个接收数据并把数据抛弃掉的设备文件。在执行 shell 命令时，如果不再需要程序产生的结果，则可以使用它。例如，

```
a.out >/dev/null
```

上面的命令会执行 *a.out* 程序，但是丢弃输出结果。RAM 盘驱动程序对这个次设备采用了一种高效的处理方式：将其长度置为零，这样便没有复制数据的动作。对这个设备进行读操作马上就会返回一个 EOF (End of File, 文件的结尾) 标记。

如果在 */dev* 目录下查看这些文件的目录表项，就会发现正如刚才所说的那样，只有 */dev/ram* 是一个块设备，其他的都是字符设备。存储器驱动程序块设备还有一个，那就是 */dev/boot*。如果从设备驱动程序的观点来看，和 */dev/ram* 一样，这个设备是在 RAM 中实现的另一个块设备。但是，它初始化时复制一个文件附加到引导映像后面，而不像 */dev/ram* 一样，开辟一块空内存。支持这个设备是为了以后使用，就像在这本书中所说的，在 MINIX 3 中没有使用它。

存储器驱动程序支持的最后的设备是另外一个字符设备文件`/dev/zero`。使用它可以方便地得到字符零。向设备`/dev/zero`写数据类似于向`/dev/null`设备写数据，它会将数据抛弃掉。但是如果读设备`/dev/zero`，则会返回零，返回的零的数目没有什么限制，从一个单独的字符到整个磁盘都可以。

在驱动程序级别上，处理`/dev/ram`, `/dev/mem`, `/dev/kmem` 和 `/dev/boot` 的代码是相同的，区别仅在于它们各自对应不同的存储区，这些存储区由数组 `ram_origin` 和 `ram_limit` 指示，这两个数组均由次设备号进行索引。文件系统在一个更高的级别上管理这些设备，将这些设备解释为字符设备或者块设备，因此能够加载`/dev/ram` 和 `/dev/boot` 并管理这些设备上的目录和文件。如果设备是一个字符设备，则文件系统只能读写字符流（虽然从`/dev/null` 设备读取流只能得到 EOF）。

3.6.3 MINIX 3 中 RAM 盘驱动程序的实现

和其他磁盘驱动程序一样，RAM 盘驱动程序的主循环在文件 `driver.c` 中，和设备相关的支持存储器设备的代码在文件 `memory.c` 中（10 800 行）。当存储器驱动程序被编译时，源文件 `drivers/libdriver/driver.c` 编译生成目标文件 `drivers/libdriver/driver.o`，源文件 `drivers/memory/memory.c` 编译生成目标文件 `drivers/memory/memory.o`，然后这两个目标文件被链接到一起。

花费一些时间考虑主循环是怎么编译的是很值得的。`driver` 结构体的声明放在文件 `driver.h`（10 829 到 10 845 行）中，在这里定义了一个数据结构但是没有创建。11 645 到 11 660 行声明并定义了一个 `m_dtab` 结构体，结构体的每一部分填充了一个指向函数的指针。这些函数中有一些函数的代码是在文件 `driver.c` 编译时产生的，例如所有的 `nop` 函数；其他函数的代码在文件 `memory.c` 编译时产生，例如 `m_do_open` 函数。注意在存储器驱动程序中有 7 个函数没有做任何操作或者仅仅做了很少的操作，最后两个函数指针被赋值为 `NULL`（意味着这些函数从来不会被调用，所以甚至不必为这些函数生成空代码）。所有这些都证实了 RAM 盘操作并不复杂。

存储器驱动程序不需要定义数目众多的数据结构体。数组 `m_geom[NR_DEVS]`（11 627 行）用 64 位无符号整数以字节为单位存放了 6 个存储器设备的基地址和大小，所以目前 MINIX 3 能够支持足够大的 RAM 盘。接下来的一行定义了一个非常独特的在其他驱动程序中都不存在的结构体 `m_seg[NR_DEVS]`。它只是一个整型数组，但是借助这些整数可以找到段描述符。存储器驱动程序和普通的用户进程不同，因为它们除了可以访问自己的代码段、数据段以及栈段（每个进程都拥有这些空间并可以访问自己拥有的这些空间）外，还有能力访问其他的存储空间。这个数组包含了访问进程外面指定存储区域所需要的信息。变量 `m_device` 仅仅包含了当前使用的次设备在这个数组中的索引。

为了在 MINIX 3 启动时使用 `/dev/ram` 作为根设备，存储器驱动程序必须尽早初始化。后面定义的 `kinfo` 和 `machine` 结构体在系统启动时会记录从内核获取到的初始化存储器驱动程序所必需的数据。

在可执行代码开始之前定义的另一个数据结构是 `dev_zero`，这是一个大小为 1024 字节的数组，当对 `/dev/zero` 设备进行读操作时用来提供数据。

主过程 `main`（11 672 行）调用一个函数进行一些局部初始化后进入主循环。主循环完成取消息，将其分派到相应的过程，然后发出应答。执行完毕后并不返回到 `main`。

下一个函数 `m_name` 不是很重要，被调用时会返回字符串“`memory`”。

对于一个读或写操作，主循环执行三个调用：一个准备设备，一个做实际的数据传输，一个结束操作。对于存储器设备，首先调用 `m_prepare`，它检查请求的次设备是否合法，然后返回一个结构的地址，该结构中存放着所请求 RAM 区的基地址和长度。第二个调用的是 `m_transfer`（11 706 行），它执行所有的工作。观察 `driver.c` 文件就会发现，所有对数据读写的调用都被转换为对多个相邻块

数据读写的调用，如果只需要一个块，则请求被解释为数目只有一个的多重块。所以传送给驱动程序的请求只有两种类型：如果请求读一个或者多个块，则传送一个请求 *DEV_GATHER*，而 *DEV_SCATTER* 则对应一个或者多个块的写操作。因此，获取次设备号以后，*m_transfer* 进入一个循环，循环次数与传送请求的数目相同，在循环内部会根据设备的类型做不同的选择。

首先是 */dev/null*，如果是 *DEV_GATHER* 请求则立即返回，如果是 *DEV_SCATTER* 请求则直接跳到分支结构的最后。这样做可以使得像其他任何写操作一样，返回传送的字节数（虽然对于 */dev/null* 设备而言这个数目是零）。

所有的设备类型查找请求的数据在内存中的实际位置的操作都是类似的。首先会比较请求的偏移量和设备的大小，以确认请求的数据是在为设备分配的存储区域内。然后借助内核调用将数据复制到用户存储空间，或者从用户存储空间复制数据到设备存储空间。在这里有两块代码用来完成这些任务。*/dev/ram*, */dev/kmem* 和 */dev/boot* 使用虚地址，故需要借助 *m_seg* 数组获得存储区域的段地址，然后产生一个 *sys_vircopy* 系统调用（11 640 到 11 652 行）。*/dev/mem* 使用物理地址，对应 *sys_physcopy* 系统调用。

剩下的操作就是对 */dev/zero* 的读写操作，读操作会使用前面介绍的 *dev_zero* 数组。向目标位置复制数据需要借助内核调用，如果此时根据需要生成字节零，则或者以极其低效的方法逐个字节地将数据从存储器驱动程序复制到系统任务中，或者借助另外的代码在系统任务中产生字节零。后一种方法会增加内核空间代码的复杂度，在 MINIX 3 中这种情况是要尽量避免的。因此对 */dev/zero* 的读操作使用了缓冲区，而没有在需要时临时生成字节零。

存储器设备不需要第三步来结束一次读写操作，在 *m_dtab* 中相应的内容为调用 *nop_finish*。

m_do_open (11 801 行) 打开一个存储器设备，其主要任务是调用 *m_prepare* 来检查访问的设备是否合法。在较早版本的 MINIX 中，从关于代码的注释中可以发现一个比较有趣的现象。用户进程中用来打开 */dev/mem* 或者 */dev/kmem* 的调用被授予了访问 I/O 端口的能力。奔腾系列的 CPU 实现了四个特权级，用户程序一般情况下运行在最低特权级。当一个进程尝试执行一条当前特权级别所不允许的指令时，CPU 将产生一个一般保护异常。由于只有 root 级别的用户才能够访问存储器驱动程序，提供一个方法来绕过这个规定被认为是安全的。无论如何，由于内核调用允许通过系统任务进行 I/O 访问，这个潜在的危险“特性”在 MINIX 3 中是存在的。在注释中还指出，如果 MINIX 3 所在的硬件将 I/O 映射到存储区域，则这个特性需要被重新考虑。虽然没有被广泛使用，但内核代码中的 *enable_iop* 函数展示了一个可行的实现方法。

下一个函数 *m_init* (11 817 行) 仅当第一次调用 *mem_task* 时被调用一次。这个例程使用了许多系统调用，仔细研究它会进一步了解 MINIX 3 中驱动程序是怎样通过系统任务提供的服务与内核空间交互的。首先，使用 *sys_getkinfo* 内核调用获得内核 *kinfo* 数据的一份副本，利用这个数据就可以将 */dev/kmem* 的基地址和大小正确地设置到 *m_geom* 数据结构中相应的位置。另一个内核调用 *sys_sectl* 将 */dev/kmem* 的物理地址和大小转换为段描述符信息，这样就可以将内核存储区作为虚拟地址空间使用。如果一个引导设备的映像被编译进系统引导映像中，那么用来记录 */dev/boot* 基地址的区域将不再为空。为了访问设备的存储区域，和 */dev/kmem* 的初始化过程一样，*/dev/boot* 设备同样初始化了必要的信息。接下来 */dev/zero* 设备使用的用来提供数据的数组被明确地初始化为零。这个步骤或许不是必需的，因为 C 编译器将任何新创建的静态变量都初始化为零。

最后 *m_init* 使用 *sys_getmachine* 内核调用从内核中获取另一个数据 *machine*，它标记了各种可能的硬件构架。在这里有用的信息为 CPU 是否支持保护模式。根据 MINIX 3 是运行在 8088 还是 80386 模式下，将 */dev/mem* 的长度分别设置为 1 MB 或 4 GB - 1。这些长度值是 MINIX 3 支持的最大长度，和机器上安装的 RAM 多少无关。在这里仅仅设置设备的大小，编译器统一将基地址设

置为零。由于 `/dev/mem` 直接访问物理（不是虚拟）内存，故没有必要使用 `sys_segctl` 内核调用设置段描述符。

在停止讨论 `m_init` 之前，还应该介绍一下另一个在代码中不是很明显的内核调用。在存储器驱动程序初始化过程中执行的许多操作对于 MINIX 3 的正常工作是很重要的，故在这个过程中伴随了许多相关的测试，如果某一个测试失败，则会引发 `panic`。在这种情况下，`panic` 是一个位于库中的例程，它最终引起一个 `sys_exit` 内核调用。内核和（正如我们将要看到的）进程管理器以及文件系统都有自己的 `panic` 例程。库中提供的例程可以供设备驱动程序以及其他小的系统组件使用。

比较特别的是，刚才讨论的 `m_init` 函数并不初始化一个很重要的设备 `/dev/ram`。该设备的初始化在另一个函数 `m_ioctl` 中（11 863 行）。实际上，对于 RAM 盘而言，只定义了一个 `ioctl` 操作，即 `MIOCRAMSIZE`，文件系统使用这个操作设置 RAM 盘的大小。这个步骤几乎不需要什么内核提供的服务。在 11 887 行出现的 `allocmem` 调用是一个系统调用，但并不是内核调用。进程管理器处理这个过程，它维护所有必要的信息以找到一个可变的存储区域。尽管如此，在最后还需要一个内核调用。在 11 894 行，`sys_segctl` 内核调用用来将 `allocmem` 调用返回的物理地址及大小转换为段信息，以便进一步对其进行访问操作。

`memory.c` 中的最后一个函数是 `m_geometry`。存储器设备没有机械驱动器的柱面、磁道和每条磁道上的扇区等几何结构，但如果某一个对存储器设备的请求使用了这些参数，则这个函数假定设备具有 64 个磁头、每个磁道含有 32 个扇区，然后根据这些信息计算指定大小的区域包含多少柱面。

3.7 磁盘

除了嵌入式平台，任何现代计算机都拥有磁盘驱动器。故在这里需要研究磁盘，首先从硬件开始，然后探讨一些磁盘软件中通用的东西。接下来会研究 MINIX 3 是怎么控制磁盘的。

3.7.1 磁盘硬件

所有实际的磁盘都组织成许多柱面，一个柱面上的磁道数和垂直放置的磁头个数相同。磁道又被分成许多扇区，每条磁道上扇区数目的典型范围是：对软盘每条磁道为 8~32 个扇区，在某些硬盘上则可多达几百个扇区。最简单的设计是每条磁道具有相同的扇区数，每个扇区包含相同数目的字节。然而，略加思索就可以知道，物理上靠近磁盘外边沿的扇区比靠近磁盘内边沿的扇区要长一些。不过读写每个扇区的时间是一样的。很明显，在最里面的柱面上的数据密度要高一些，一些磁盘的设计要求读写内部磁道时改变磁头的驱动电流，这由磁盘控制器硬件处理，对用户（或操作系统的实现者）不可见。

内圈磁道和外圈磁道数据密度的不同意味着会牺牲一些磁盘容量，也意味着存在更复杂的系统。有人尝试设计过一种当磁头处于外部磁道时磁盘旋转速度更快的软盘，这样外圈磁道就可以具有更多的扇区，从而增加磁盘的容量。目前可用 MINIX 3 的系统还不支持这种磁盘。然而，现代大容量硬盘中外圈磁道具有的扇区数比内圈磁道更多，这就是 IDE（Integrated Drive Electronics）驱动器，它内置的电子器件进行的复杂处理屏蔽了具体细节，对于操作系统来说，它们仍然呈现出简单的几何结构，每条磁道具有相同的扇区。

驱动器与控制器电子器件和机械硬件同等重要。磁盘控制器的主要元件是一块专用的集成电路，实际上是一个小的微型计算机。以前，它需要被放置到一个板子上，然后将这个板子插到计算机的主板上，不过在现代系统中磁盘控制器被直接放置到主板上了。对于一块现在的硬盘，磁盘控制器的控制线路可能比软盘的还要简单，这是因为硬盘驱动器本身具有一个功能强大的电子控制器。

对于磁盘驱动程序有重要意义的一个设备特性是：控制器可以同时控制两个或多个驱动器进行寻道，这称为重叠寻道（overlapped seek）。当控制器和软件等待一个驱动器完成寻道时，控制器可以启动另一个驱动器进行寻道。许多控制器可以在对一个或多个其他驱动器寻道的同时在一个驱动器上进行读写操作，但是软盘控制器不能同时读写两个驱动器（读写数据要求控制器在毫秒级的时间内传输比特流，所以一个传输就基本占用了所有的计算能力）。对于具有集成控制器的硬盘则情况不同，在一个有多个硬盘的系统中，这些驱动器可以同时操作，至少在磁盘和控制器缓冲区之间的数据传输可以并行进行。然而，在控制器和系统内存之间只能同时执行一个传输，同时执行两个或多个操作的能力极大地降低了平均读写时间。

在阅读现代硬盘规格时需要注意的一点是，驱动程序软件所指定和使用的几何参数可能与物理格式不同。事实上，如果查看一块大硬盘的推荐设置参数，将会发现无论磁盘的大小是什么，都是16 383柱面，16磁头，每磁道63个扇区。这组参数对应的磁盘容量为8 GB，但用于所有该容量以及比这个容量大的硬盘。最初的IBM PC ROM BIOS的设计者使用6位来保存扇区数目，使用4位来保存磁头数目，使用14位保存柱面数目，每个扇区大小为512字节，故对应的容量为8 GB。如果尝试在一台非常老的机器上安装一个比较大的硬盘，则即使磁盘容量非常大也只能访问8 GB空间。为突破这一限制可以使用逻辑块寻址（logical block addressing, LBA），在这一方法中，磁盘扇区从零开始连续编号，而不管磁盘的几何规格。

现代磁盘的几何参数是虚拟的。现在磁盘的表面被划分为20个或者更多的环带，内层的环带相比外层的环带每磁道拥有更少的扇区数目。故无论扇区在磁盘的什么部位，它们都近似拥有相同的物理长度，这就使得磁盘表面被更加有效地利用。在内部，集成的控制器利用环带、柱面、磁头以及扇区对磁盘进行编址。但是这些细节对于用户而言是不可见的，也很少有发布的规范来描述这些细节。总之，除非使用了一台非常老的不支持逻辑块寻址技术的计算机，否则不需要使用柱面、磁头、扇区来对磁盘进行寻址。同时，为1983年生产的PC-XT型计算机购买一个最新容量为400 GB的磁盘是没有任何意义的，因为只能使用不到8 GB的容量。

在此还应该弄清楚一个关于磁盘容量规格的模糊观点。计算机专业人士习惯于使用2的幂来计算存储器的容量，即 $1\text{ KB} = 2^{10} = 1024$ 字节， $1\text{ MB} = 2^{20} = 1024^2$ 字节，等等。这样的话， 1 GB 就应该是 1024^3 或者 2^{30} 字节。但是磁盘生产商习惯于使用Gigabyte来表示 10^9 ，这样的话，在名义上磁盘的容量就会增加。因此上面提到的8 GB的容量限制在磁盘制造商的表达中就成了8.4 GB。最近有一种趋势，使用名词Gibibyte（GiB）来表示 2^{30} 。然而，在本书中，为了保持原来的意义，同时为了反对那些因广告目的而改变传统用法的做法，作者将仍然使用诸如MB和GB这样的名词来表示它们原来的意义。

3.7.2 RAID

虽然现代磁盘的速度比老型磁盘快多了，但是CPU性能提高的幅度要远远大于磁盘性能提高的幅度。在过去的一些年中，许多人意识到并行磁盘输入/输出应该非常有帮助。这促使了称为RAID的一种新型I/O设备的诞生，RAID是Redundant Array of Independent Disks的首字母缩写。实际上，RAID的设计者（在伯克利）原来使用RAID代表Redundant Array of Inexpensive Disks（廉价磁盘冗余阵列），以与SLED（Single Large Expensive Disk）设计相比较。不过，当RAID在业界流行以后，磁盘生产商改变了缩写词的意义，因为名字中的“廉价”不利于他们卖出比较昂贵的磁盘。RAID背后的基本思想是将一个装满了磁盘的盒子安装到计算机（通常是一个大型服务器）上，用RAID控制器替代磁盘控制器，将数据复制到整个RAID，然后继续常规的操作。

独立的磁盘能够以多种方法整合到一起使用，在这里不会详尽地描述所有这些方法，同时 MINIX 3 还不支持 RAID，但是作为操作系统的介绍至少应该提及一些可能的方法。RAID 能被用来提高磁盘访问速度，或者使数据更加安全。

在此举一个简单的例子。考虑一个仅由两个磁盘组成的 RAID。当多扇区数据要被写到磁盘上时，RAID 控制器将扇区 0, 2, 4 等发送到第 1 块磁盘上，将扇区 1, 3, 5 等发送到第 2 块磁盘上。控制器负责将数据分成两份，两个磁盘同步进行写操作，故写操作速度将变为原来的两倍。读取数据时，两个驱动器同步进行读操作，控制器负责将数据重新按照正确的顺序组合到一起，这样系统的其他部分仅仅看到读取的速度提高了一倍。这种技术称为条带 (striping)。这就是简单的 0 级 RAID 例子。实际使用时会使用四个或者更多的磁盘驱动器。如果经常进行大块的数据读写，则这种方式工作得很好。如果一次只读写一个扇区，则就变成了典型的磁盘请求操作，这时没有任何优势。

前面的例子展示了多个磁盘驱动器组合在一起是怎么提高访问速度的。那么怎么提高可靠性呢？1 级 RAID 和 0 级 RAID 类似，不过数据是双份的。同样，最简单情况下可以使用由两个磁盘驱动器组成的磁盘阵列，所有的数据同时存放到两者上面。这不会提高速度，但是冗余度会提高一倍。当读取数据时如果出现错误，只要另一个能够正常读取就不需要重试。控制器只要保证正确的数据被传送给系统即可。然而，在写操作出现错误时，如果不进行重试，则不是一个很好的主意。如果错误频繁发生同时出错时不重试，则速度会明显加快，这时或许应该警惕设备马上就会彻底失效。典型情况下，用于 RAID 的磁盘是支持热插拔的，即在系统不关闭的情况下就可以替换它们。

多个磁盘更加复杂的组合能够同时提高速度和可靠性。作为例子，考虑一个由 7 个磁盘组成的阵列。字节能够分成由 4 位组成的半字节，每一位被记录到 4 个磁盘中的一个上，剩下的 3 个用来记录错误校正码。如果一个磁盘损坏了，需要通过热插拔更换一个新磁盘，则这个损坏的磁盘等价于一个损坏的位，从而能够在维护的同时保持系统正常运行。通过使用 7 个磁盘的代价，带来了可靠的 4 倍性能提高，而且没有停机时间。

3.7.3 磁盘软件

本节将笼统地浏览一下与磁盘驱动程序有关的内容。首先考虑读写一个磁盘块需要多长时间。需要的时间由下面三个因素决定：

1. 寻道时间（将磁头臂移动到相应的柱面所需的时间）。
2. 旋转延迟（相应的扇区旋转到磁头下面所需的时间）。
3. 实际的数据传输时间。

对于大多数磁盘，和另外两个时间参数相比，寻道时间要大得多，因此减小平均寻道时间将极大地提高系统性能。

磁盘设备可能会出错，所以常将某种错误校验（即校验和或者循环冗余校验的信息）与各扇区的数据同时记录在盘上，甚至磁盘格式化时记录的扇区地址也有校验数据。当检测到出错时，软盘控制器可以报告出错信息，但是必须由软件决定出错后该怎么办。硬盘控制器通常承担大部分出错处理工作。

尤其对于硬盘，一条磁道上连续扇区的传输是非常快的，因此读取比请求数据更多的数据并把它缓存在内存中对加速磁盘存取非常有效。

磁盘臂调度算法

如果磁盘驱动程序每次接收一个请求并按照接收顺序执行，即先到先服务（First-Come, First-Served, FCFS），则很难优化寻道时间。然而当磁盘负载很重时，可以采用其他策略。很有可能当

磁盘臂为一个请求寻道时，其他进程也产生了其他磁盘请求。许多磁盘驱动程序都维护一张表格，按柱面号索引，每一柱面的全部请求用一个链表链在一起，链表头指针存放在表格相应的表项中。

有了这种数据结构，就可以改进先来先服务的调度算法。为了说明如何实现，考虑一个具有40个柱面的磁盘。假设一个请求到达，请求读柱面11上的一个数据块，当对柱面11寻道时，又顺序到达了新请求要求寻道1, 36, 16, 34, 9和12，则它们被安排进入请求等待表，每一个柱面对应一个单独的链表。图3.21显示了这些请求。

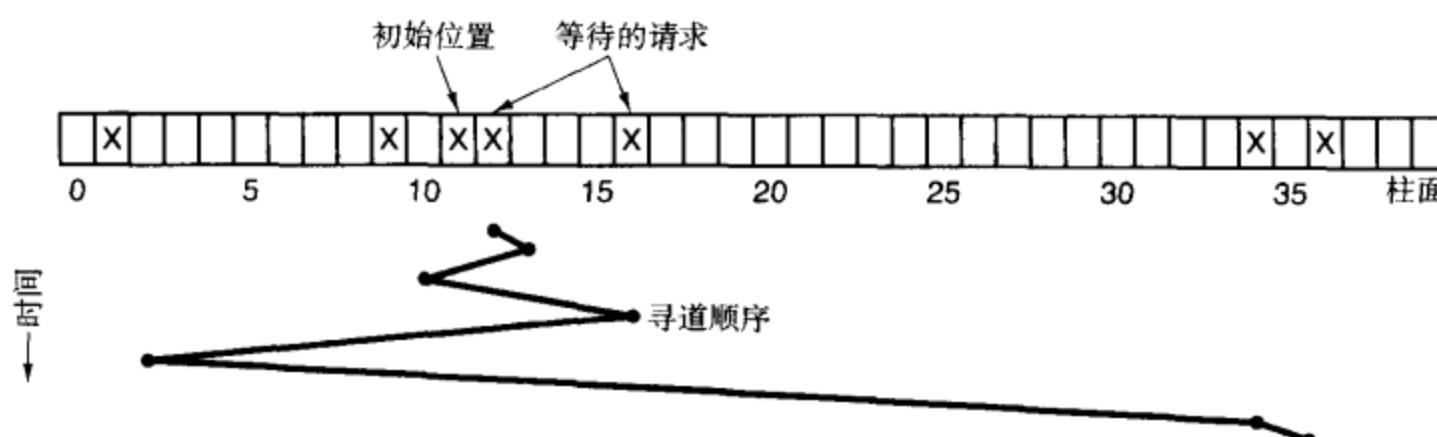


图3.21 最短寻道(SSF)磁盘调度算法

当前请求(请求柱面11)结束后，磁盘驱动程序需要选择下一个请求。若使用FCFS，则首先选择柱面1，然后是36，依次类推。这个算法中磁盘臂分别需要移动10, 35, 20, 18, 25和3个柱面，总共需要移动111个柱面。

为了减少寻道时间，也可以总是选择和磁盘臂最接近的柱面请求。对于图3.21所示的请求，被选择请求的顺序如图3.21下方的折线所示，依次为12, 9, 16, 1, 34和36。按照这个顺序，磁盘臂分别需要移动1, 3, 7, 15, 33和2个柱面，总共需要移动61个柱面。这个算法——**最短寻道算法**(Shortest Seek First, SSF)和FCFS算法相比，把柱面移动数减小了一半。

遗憾的是，SSF算法存在一个问题。假设正在处理图3.21所示的请求时，有其他请求不断到达。例如，磁盘臂移到16柱面以后，到达一个对柱面8的请求，那么它的优先级将比柱面1要高。如果接着又到达了一个对柱面13的请求，磁盘臂将移到柱面13而不是柱面1。对于一个负载很重的磁盘，磁盘臂趋向于大部分时间停留在柱面中部区域，对两端的极端区域请求的处理不得不等待，直到由于负载的统计波动使得中部区域没有请求为止。远离柱面中部区域的请求得到的服务很差。这时，获得最小响应时间的目标和公平性之间存在冲突。

高层建筑也得进行这种折中处理，高层建筑中的电梯调度问题和调度磁盘臂很相似。电梯请求不停地到来，随机地呼唤电梯到各个楼层，控制电梯的微处理器能够很容易地跟踪用户按下按钮的顺序，并使用FCFS算法为人们提供服务，也可以使用SSF算法。

然而，大多数的电梯使用另一种算法来调和效率和公平性的冲突。电梯保持按一个方向运动，直到在那个方向上没有更远的请求为止，然后改变方向。这个算法在磁盘领域和电梯领域都被称为**电梯算法**(elevator algorithm)。它需要软件维护一个二进制位，即当前的方向：向上或是向下。当一个请求结束之后，磁盘或电梯的驱动程序检查该位，如果是向上，磁盘臂或电梯舱则移至下一个更高的等待请求。如果更高的位置没有请求，就反转方向位。如果方向位设置为向下，同时存在一个低位置的请求，则移向该位置。

图3.22显示了与图3.21相同的7个请求使用电梯算法的情况。假设初始方向位为向上，则各柱面获得服务的顺序是12, 16, 34, 36, 9和1，磁盘臂分别移动1, 4, 18, 2, 27和8个柱面，总共移动

60个柱面。在该例中，电梯算法比SSF算法稍微好一点，尽管通常它不如SSF。电梯算法的一个优良特性是对任意的一组请求，移动磁盘臂次数的上界是固定的：正好是柱面数的两倍。

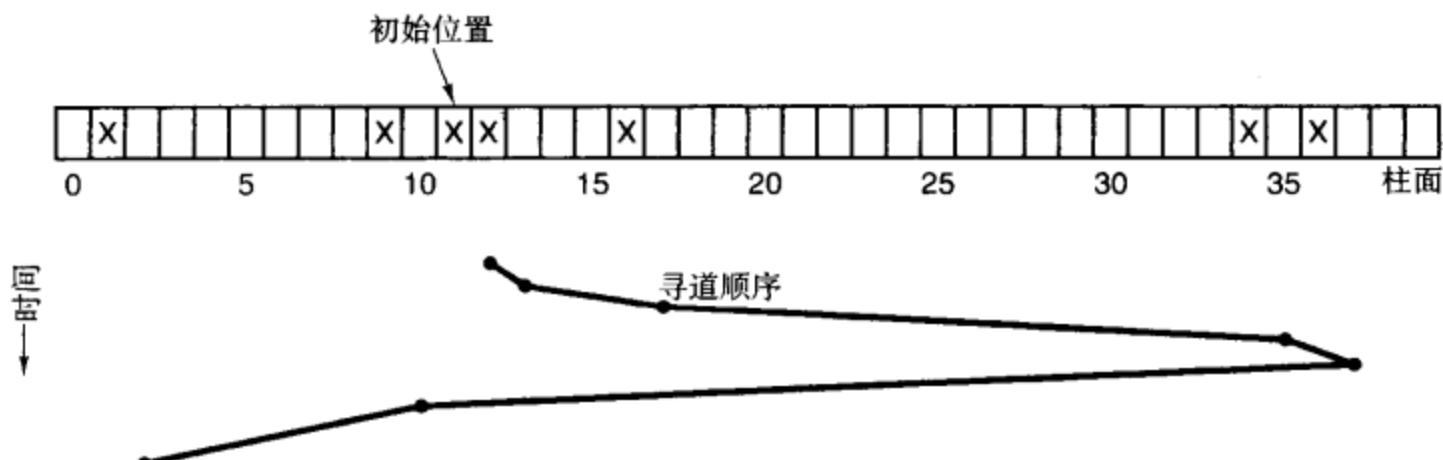


图 3.22 调度磁盘请求的电梯算法

对这个算法进行略微改进就可以进一步减小响应时间 (Teory, 1972)，方法是总按一个方向移动磁盘臂，处理完编号最高柱面上的请求后，磁盘臂移动到具有读写请求的编号最低的柱面，然后继续向上移动。这实际上等于将最低柱面视为最高柱面之上的相邻柱面。

一些磁盘控制器提供了供软件检查当前磁头下方扇区号的方法。对于这种磁盘控制器，还可进行另一种优化。如果有两个或多个读写同一柱面的请求正在等待处理，驱动程序可以请求读写下一个要通过磁头的扇区。注意，当一个柱面有多条磁道时，因为选择磁头既不需要移动磁盘臂也没有旋转延迟，控制器可以快速选择任意磁头，所以连续的读写请求可以申请不同的磁道。

对于现在的硬盘，其数据传输率比软盘快得多，所以必须采用某种自动高速缓冲机制。典型地，任何读扇区的请求将使得这个扇区和同一磁道上的其他扇区都被读进来，读进多少扇区取决于控制器中有多少可用的高速缓冲空间。目前缓冲区的大小通常为 8 MB 或者更多。

当有多个驱动器时，每个驱动器都有一个单独的请求等待表。一旦任何一个驱动器空闲，则启动一个寻道请求将磁盘臂移到下一个请求的柱面（假设控制器允许重叠寻道）。当前数据传输结束后，将检查是否有一个驱动器位于正确的柱面上，如果存在一个或多个这样的驱动器，则启动下一个数据传输。如果没有，则驱动程序对刚结束数据传输操作的驱动器发出新的寻道命令并等待，直到下一次中断到来时检查哪一个磁盘臂首先到达了目标位置。

错误处理

RAM 盘不需要考虑寻道和旋转优化：在任意时刻，不做任何物理移动就可以读写所有的数据块。RAM 盘比实际磁盘简单的另一个地方是出错处理。RAM 盘永远正常工作，而实际磁盘却不能，它们可能会出现各种各样的错误。常见的错误有：

1. 程序性错误（例如请求读写不存在的扇区）。
2. 暂时性校验和错（例如由磁头上的灰尘引起）。
3. 永久性校验和错（例如磁盘块的物理损坏）。
4. 寻道出错（例如磁盘臂应定位在第 6 柱面，但却到了第 7 柱面）。
5. 控制器错（例如控制器拒绝接受命令）。

磁盘驱动程序应尽可能地处理这些错误。

当驱动程序通知控制器对不存在的扇区进行寻道、使用不存在的磁头或者对不存在的存储器传送数据时，将发生程序性错误。多数控制器检查发送给它们的参数，并在参数非法时给出信息。理

论上这些错误不会发生，但是当控制器指出发生了这些错误时，驱动程序如何处理呢？对于家用系统，最好的办法是停止运行，打印一条类似“求助于程序员”的消息，以便对错误进行跟踪并加以修正。对于一个在世界上数以千计的地方使用的商用软件产品，这种方法并不合适，也许唯一能做的就是终止出错的当前磁盘请求，并且希望这种错误不会发生得太频繁。

暂时性校验和错是由于空气中的灰尘进入了盘面和磁头之间。多数情况下，可以通过重复操作几次来消除这种错误。如果它一直存在，则只能将相应的数据块标记为坏块（bad block），并且不再使用。

避免坏块的一种方法是写一个非常特殊的程序，它将坏块清单作为输入，并小心地创建一个文件，使之包含所有的坏块。一旦这个文件创建完毕，磁盘分配程序就认为这些数据块已被占用并永远不会把它们分配出去。只要不去读坏块文件，就不会发生任何问题。

永远不读坏块文件说起来容易做起来难。许多磁盘采用一次将一条磁道的内容复制到后备磁带或磁盘上的方法进行备份。如果使用这种方式，则坏块就会引起麻烦。如果后备程序知道坏块文件的文件名并且不复制它，那么按一次复制一个文件的方式虽然要慢一些，但可以解决这个问题。

坏块文件不能解决的另一个问题是：如何处理必须位于磁盘中固定位置的文件系统数据结构中存在的坏块。几乎每一文件系统至少都有一个数据结构的位置必须是固定的，以便很容易地找到它。在一个可分区的文件系统中，通过重新分区可以避开坏道，但软盘或硬盘开始几个扇区中的永久性错误一般意味着整个磁盘无法使用。

“智能”控制器保留了几条用户程序一般不能使用的磁道。当格式化磁盘时，控制器可以确定哪些是坏块，并用保留的磁道代替损坏的磁道。存放坏道和保留磁道之间对应关系的表格保存在控制器的内部存储器和磁盘中。这种替换对驱动程序透明，只是对于精心设计的电梯算法，如果每次请求读写柱面3时，控制器实际使用的是柱面800，那么其性能会大大降低。磁盘记录表面的制造技术已经比过去有所提高，但还不是尽善尽美，不过对用户隐藏缺陷的技术也得到了提高。控制器也管理在使用过程中发生的新错误，并在错误不能恢复时永久地分配一个替换块。对于这样的磁盘，驱动程序软件很少会感觉到坏块的存在。

寻道错由磁盘臂的机械问题引起。控制器在内部跟踪磁盘臂的位置。为了寻道，控制器向磁盘臂电机发出一系列脉冲，每个脉冲移动一个柱面，从而将其移动到新的柱面。当磁盘臂到达目标柱面后，控制器读取实际的柱面号（在驱动器格式化时被写上去），如果磁盘臂处于不正确的位置，就会发生寻道错，这时就需要一些校正操作。

大多数硬盘能自动纠正寻道错，但许多软盘控制器（包括IBM PC）仅设置一个出错位，而把其他工作留给驱动程序。驱动程序处理这个错误的方法是发出一条recalibrate命令，把磁盘臂移到最外面，并在控制器内部将当前柱面号复位为0，如果不能解决问题，驱动器就必须进行修理了。

正如所看到的：控制器实际上是一个专用的小计算机，它有完备的软件、变量、缓冲区，偶而还出现故障。有时一个非常的事件序列，例如一个驱动器上的中断和另一个驱动器上的recalibrate命令同时发生可能引发一个故障，致使控制器陷入一个循环或无法继续正在做的工作。控制器设计者通常考虑到最坏的情况，在芯片上提供了一个引脚，当触发该引脚时，强迫控制器忘记当前的工作并自行复位。如果其他努力都无法奏效，磁盘驱动程序可以设置一个比特来触发该信号对控制器复位。如果还不成功，那么驱动程序就只能打印出错信息并放弃。

每次一道缓冲

对一个新柱面寻道所花的时间通常比旋转延迟多得多，而且往往比传输时间长很多。也就是说，一旦驱动程序需要把磁盘臂移到某个位置，那么读一个扇区还是读一条磁道都差不多。特别是对于控制器提供旋转检测（rotational sensing）的情况，这种效应就更加明显。因为这时驱动程序能

够知道哪个扇区正位于磁头下面，并启动一个对下一扇区的请求，从而使得在一次旋转中读一条磁道成为可能（平均来讲，通常花旋转半周的时间加上一个扇区的时间仅仅能读一个扇区）。

有些磁盘驱动程序通过维护一个秘密的每次一道（track-at-a-time）的高速缓冲来充分利用这一特性，并且不被独立于设备的软件所感知。如果需要的扇区位于高速缓冲里，则不需要磁盘数据传输。每次一道缓冲的一个缺点（除了软件复杂性和需要缓存空间）是：从高速缓冲到调用程序之间的数据传输必须由 CPU 使用一个循环完成，而不是让 DMA 硬件来做这项工作。

一些控制器将这个过程更进一步，在它们自己内部的存储器中实现每次一道缓冲，而对驱动程序透明，这样在控制器和存储器之间的数据传输就能使用 DMA。如果控制器按这种方式工作，就没必要让磁盘驱动程序也做这项工作。注意，在控制器和驱动程序中由一个命令读写整条磁道是合适的，但不能放在设备无关软件中，因为它将磁盘视为数据块的线性序列，而不考虑它如何划分成磁道和柱面。只有控制器确实知道真实的磁盘几何参数分布情况。

3.7.4 MINIX 3 中的硬盘驱动程序简介

硬盘驱动程序是我们已经看到 MINIX 3 不得不处理各种类型硬件的第一部分。在开始讨论驱动程序细节以前，简单考虑一下一些硬件的不同可能引起的问题。

“PC”确实是一个不同计算机的家族。在不同的家族成员中不仅使用的处理器不同，而且在基本硬件上也有很大的差别。MINIX 3 是在具有奔腾系列 CPU 的新系统上开发的，这也是它的运行平台，但是即使如此仍然存在不同。例如，比较老的奔腾系统使用最初为 80286 处理器设计的 16 位 AT 总线。AT 总线的一个特点是其经过巧妙的设计从而使得原来的 8 位外部设备仍然能够使用。后来的系统在保留 AT 总线插槽情况下为外围设备增加了 32 位的 PCI 总线。最新的设计已经完全放弃了对 AT 总线的支持，仅仅提供 PCI 总线。有理由相信使用计算机很长时间的用户会期望在 MINIX 3 上能够组合使用 8 位、16 位以及 32 位的外围设备。

对每一类总线，都有一个与之相应的 I/O 适配器（I/O adapter）系列。在较老的系统中是一些单独的电路板，它们被插在系统主板上。而在比较新的系统中许多标准适配器，特别是磁盘控制器，已经被集成到了系统主板芯片中。本身而言，这对于编程者不会产生什么问题，因为一般情况下集成的适配器和可移除设备具有相同的软件接口。同时，集成的控制器一般情况下可以被禁用。这使得我们能够使用功能更强大的附加设备，例如 SCSI 控制器，来取代内置的设备。为了达到这样的可伸缩性，操作系统不应限制只能使用一种适配器。

在 IBM PC 家族中，和其他大多数计算机系统一样，与总线设计同时配套的还有基本 I/O 系统（BIOS ROM）中的固件。其目的是在操作系统和硬件特殊性之间建立一个连接二者的桥梁。有些外设在自己的外设板上提供扩充 BIOS 的 ROM 芯片。操作系统的实现者面临的困难是：在 IBM 类型的计算机（当然指早期的）中的 BIOS 是为 MS-DOS 操作系统设计的，不支持多道程序设计，运行于 16 位实模式，这是 80x86 CPU 家族可以使用的各种操作模式中最普通的一种模式。

于是，为 IBM PC 设计新操作系统的实现者面临几个选择：第一项选择是使用 BIOS 中的设备驱动程序支持，还是从头开始编写新的驱动程序。由于 BIOS 在许多方面不适于 MINIX 的需要，所以在 MINIX 的初始设计中做出选择并不困难。当然，为了启动 MINIX 3，引导监控程序仍要使用 BIOS 实现系统的初始装载——无论是从硬盘还是从软盘，实际上这是无法选择的。一旦系统被装入，包括设备驱动程序，就可以比 BIOS 做得更好。

这里面临的第二项选择是：没有 BIOS 的支持，如何使设备驱动程序适用于不同系统中的各种硬件。为了使讨论更具体，考虑 MINIX 3 设计所基于的现代 32 位奔腾系统中能够使用的两种根本不同的硬盘控制器：集成的 IDE 控制器和附加的 PCI 接口的 SCSI 控制器。如果为了更好地利用以

前的硬件，同时使 MINIX 3 能够在以前可以运行老版本 MINIX 的硬件环境中，则存在至少四种基本类型的硬盘控制器：最初的 8 位 XT 类型控制器、16 位 AT 类型控制器，以及 IBM PS/2 系列计算机中两种不同类型的控制器。有几种不同的方法解决这个问题。

1. 为需要支持的每种硬盘控制器重新编译一个操作系统版本。
2. 在内核中编译几个不同的硬盘驱动程序，由内核在启动时自动决定使用哪一个。
3. 在内核中编译几个不同的硬盘驱动程序，提供一种方法使用户决定使用哪一个。

正如后面即将看到的，这些方法并不相互排斥。

第一种方法从长远来看是最好的。对于使用特定配置的系统，没有必要为从来不使用的驱动程序浪费磁盘和内存空间，然而，对软件发行者而言这却无法接受。准备四套不同的启动盘，并告诉用户如何使用这些盘片，这样做代价很高，也十分困难。因此，值得考虑其他方案，至少对于初始安装而言是这样。

第二种方法是由操作系统检测外部设备，通过读写每块卡上的 ROM 或者写和读 I/O 端口来识别每一块卡。这是可行的（在新的 IBM 类型的系统中要比老的系统工作得好多了），但是这种方法对于非标准 I/O 设备不是很适合。有些情况下，读写 I/O 端口来对设备进行识别可能会激活其他的设备，使其获得控制权而导致系统无法工作。这种方法也使每个设备的启动代码复杂化，并且还是不能工作得很好。使用这种方法的操作系统一般必须提供某种重载机制，如在 MINIX 3 中使用的典型机制。

第三种办法正是 MINIX 3 使用的方法，它允许在系统引导映像中包含多个驱动程序。MINIX 3 引导监控程序允许在启动时读各种引导参数（boot parameter），可以手工输入这些参数，也可以将其永久存放于磁盘上。在启动时如果发现引导参数格式为

```
label = AT
```

则在 MINIX 3 启动时强制使用 IDE 硬盘驱动程序（at_wini）。这依赖于 at_wini 驱动程序分配了这个标号。标号在引导映像编译阶段被分配。

为了减少支持多个硬盘驱动程序所导致的问题，MINIX 3 还做了其他两件事。第一是提供了一个支持在 MINIX 3 和 ROM BIOS 硬盘之间接口的驱动程序，这个驱动程序几乎可以保证在所有的系统下都可以工作，通过使用引导参数

```
label = BIOS
```

来选择，不过一般这是可求助的最后一种办法。在 80386 或更高级的处理器上，MINIX 3 运行于保护模式，但是 BIOS 永远运行于实模式（8086）。无论何时调用 BIOS 中的例程都会切换出保护模式，然后重新回到先前的模式，这是很慢的。

MINIX 3 处理驱动程序的另一策略是尽可能推迟初始化工作。这样，如果在某些硬件配置中没有硬盘驱动程序可用，则仍然可以从软盘启动，以完成一些有用的工作。只要不访问硬盘，MINIX 3 就不会有任何问题。在用户友好方面这算不上一个大的突破，但是考虑这种情况：由于不恰当地配置了一些从不使用的设备，如果系统引导时所有的驱动程序立即初始化，那么系统将全部处于停滞状态。把设备初始化推迟到需要的时刻，则当用户试图去解决问题时，系统可以使用任何可以工作的部件继续运行。

另一方面，大家在以困难的方式学习这门课程：早期的 MINIX 版本一启动就试图初始化硬盘，如果没有硬盘，系统就挂起。这是很糟糕的，在没有硬盘的情况下，虽然存储能力要受限制，性能也要降低，但是 MINIX 应该能够在没有硬盘的系统中流畅地运行。

在本节和下一节的讨论中，我们选择了AT类型的硬盘驱动程序为例，在发布的标准MINIX 3中这是默认的设备驱动程序。这是一个多功能的设备驱动程序，可以处理从早期80286系统中使用的硬盘控制器到上GB存储容量的EIDE（Extended Integrated Drive Electronics，扩展的集成驱动电子线路）控制器。现代的EIDE控制器也支持标准的CD-ROM驱动器。不过，为了使得讨论简化，有关能够支持CD-ROM的扩展特性的代码被抽出并列在了附录B中。在本节所讨论的硬盘操作的一般内容也适用于系统所支持的其他硬盘驱动程序。

硬盘任务的主循环是已经讨论过的相同的共享代码，可以执行9种标准请求。因为在硬盘上总有分区和子分区，所以`DEV_OPEN`请求需要执行大量的工作。当打开设备时（第一次被访问时），必须首先读入这些代码。如果支持CD-ROM驱动器，由于它们属于可移动介质，故执行`DEV_OPEN`时，必须检查介质是否存在。对于CD-ROM，`DEV_CLOSE`操作还有以下含义：它需要打开光驱弹出光盘。可移动介质的其他复杂性在软驱中更普遍，所以将在下一节讨论。对于CD-ROM，`DEV_IOCTL`操作设置当执行`DEV_CLOSE`时介质应被弹出的标志。`DEV_IOCTL`操作也可以用来读写分区表。

像先前看到的那样，每一个`DEV_READ`请求、`DEV_WRITE`请求、`DEV_GATHER`请求和`DEV_SCATTER`请求都分成两个阶段来处理：准备和传输。对于硬盘，`DEV_CANCEL`请求和`DEV_SELECT`请求被忽略。

硬盘驱动程序自己不进行任何调度算法，该项工作是由文件系统完成的，它将集中/分散(gather/scatter)I/O的请求组合起来成为请求队列。来自文件系统缓存的诸如`DEV_GATHER`以及`DEV_SCATTER`的请求申请多个块（在MINIX 3中默认配置为4KB），但驱动程序也能够处理大小为扇区尺寸(512字节)任意倍的请求。无论如何，正如所看到的，所有硬盘驱动程序的主循环将把对一个单独块数据的请求转换为请求向量中的一个元素。

在一个请求队列中读操作和写操作不会被混合到一块，请求也不能够被标记为可选。请求队列中的元素都针对连续的磁盘扇区，并且文件系统在将队列交给驱动程序之前会对其进行排序，故对于一个完整的请求队列而言，能够在磁盘上指出最开始的请求位置就足够了。

对于一个请求队列，一般认为驱动程序最低限度能够正确处理其第一个读写请求，如果请求失败则会返回。最后由文件系统决定怎么处理出错，一般情况下文件系统会尝试完成写操作，但是对于读操作，文件系统仅仅返回调用进程尽可能多的数据。

文件系统通过使用分布I/O可以实现类似Teory版本的电梯算法，在一个分布式I/O请求中请求列表按照块号排序。对于现代硬盘，调度算法的第二步在控制器中实现。这些控制器具有“智能”，能够缓冲大量的数据，使用内部的程序算法来按最有效的顺序获取数据，而不是按照请求的到达次序来获取数据。

3.7.5 MINIX 3 中硬盘驱动程序的实现

微机上使用的小型硬盘有时称为“温彻斯特”硬盘。关于这个名字的来源有几个不同的故事。很明显它是IBM一个开发磁盘技术项目的代号。在这个项目中，读/写磁头飞行于薄薄的空气垫上，当磁盘停止旋转时落在记录介质上。这个名字的一种解释是，早期的型号有两个数据模块：30MB固定硬盘和30MB可移动硬盘。可以设想它使开发人员想起了温彻斯特30-30火枪，一个在美国西部传说中的角色。但无论名字来源何处，其基本技术是一样的，尽管今天典型的微型计算机上的磁盘比起14英寸磁盘体积要小得多，容量要大得多（14英寸磁盘是20世纪70年代开发“温彻斯特”技术时的典型磁盘）。

MINIX 3 中 AT 风格的驱动程序在 *at_wini.c* (12 100 行) 中。这是一个面向高级设备的复杂驱动程序，有好几页宏定义了控制器寄存器、状态位和命令、数据结构和原型。像其他的块设备驱动程序一样，把一个 *driver* 结构 *w_dtab* (12 316 到 12 331 行) 初始化为指向实际完成这项工作的函数指针。这些指针大多数在 *at_wini.c* 中定义，但是由于硬盘不需要特殊的清除操作，所以 *dr_cleanup* 指向了 *driver.c* 中公用的 *nop_cleanup*，和其他不需要特殊清除操作的设备驱动程序共享该函数。一些其他和驱动程序无关的函数初始化被指向 *nop_* 函数。入口函数 *at_winchester_task* (12 336 行) 调用一个由硬件决定的初始化过程，然后调用 *driver.c* 中的主循环，并将 *w_dtab* 的地址传递过来。*libdriver/driver.c* 文件中的 *driver_task* 主循环永远运行，把调用分派到 *driver* 表中指向的各种函数。

因为这里正在处理实际的机械电子存储设备，所以要 *init_params* (12 347 行) 做一定量的工作来初始化硬盘驱动器。有关硬盘的各种参数保存于定义在 12 254 到 12 276 行的数组 *wini* 中。无论是内置的 IDE 控制器还是 SATA (Serial AT Attachment) 控制器，支持的 *MAX_DRIVES* (8) 个驱动程序（包括 4 个传统的 IDE 驱动器以及 4 个 PCI 总线上的驱动器）中的每一个在该数组中都有一个元素。

作为推迟初始化策略的一部分，由于在真正需要使用设备之前进行设备初始化可能会失败，所以在内核初始化时调用的 *init_params* 并不做任何需要访问磁盘的工作。它做的主要工作是把有关磁盘的逻辑配置信息复制到 *wini* 数组中。这些信息是 ROM BIOS 从 CMOS 存储器中提取的，奔腾类计算机在这些存储器中存放配置信息。当机器第一次接通电源时，在 MINIX 3 第一部分的装载过程开始以前，执行 BIOS 中的功能。从 BIOS 复制信息的代码在 12 366 到 12 392 行。在这里使用了大量的常量，例如，在文件 *include/ibm/bios.h* 中定义的 *NR_HD_DRIVES_ADDR*。这个文件未在附录 B 中列出，但是在 MINIX 3 的 CD-ROM 中能够找到它。取不出这项信息并不是致命的，如果磁盘是新型的，那么第一次访问这些信息时可以从磁盘直接得到。从 BIOS 获取到初始的数据以后，对每个磁盘的附加信息可以通过调用下一个函数 *init_drive* 来完成。

在带有 IDE 控制器的老系统中，即使控制器可能被集成到主板上，磁盘函数也与 AT 风格的外围板卡类似。新的磁盘控制器使用类似于 PCI 设备的函数，不再使用 AT 总线的 16 位数据总线，而通过 32 的数据通道与 CPU 连接。幸运的是，一旦初始化完成，对程序员而言所有的磁盘控制器接口都是相同的。为了做到这一点，如果有必要会调用 *init_params_pci* (12 437 行) 函数以获取 PCI 设备的参数。在此不会详细描述这个例程的细节，但是需要指出几点：首先，在 12 361 行使用的启动参数 *ata_instance* 用来设置变量 *w_instance* 的值。如果在启动参数中没有明确指出这个参数的值，则默认情况下会被设置为零。如果这个参数被设置为一个大于零的数，则 12 365 行的测试代码使得查询 BIOS 以及对标准 IDE 驱动程序的初始化会被跳过。这样就只有在 PCI 总线发现的设备能够注册。

第二点需要指出的是，PCI 总线上发现的控制器会被标记为控制设备 *c0d4* 到 *c0d7*。如果 *w_instance* 非零，则设备标记号在 *c0d0* 到 *c0d3* 之间的设备会被忽略，除非某一个 PCI 总线控制器将自己标记为“兼容的”。兼容的 PCI 总线控制器在驱动程序中标记为 *c0d0* 到 *c0d3*。对于大多数 MINIX 3 用户而言，可以忽略这些兼容性问题。如果一台计算机拥有的驱动器的数目小于 4 个（包括 CD-ROM 驱动器），则一般情况下会使用典型配置。无论它们是连接到 IDE 控制器上还是 PCI 控制器上，无论使用的是典型的 40 针并行控制器还是较新的串行控制器，驱动器都会使用 *c0d0* 到 *c0d3* 标记自己。但是用来创建这些抽象的程序是比较复杂的。

调用通用的主循环以后，直到试图访问硬盘以前，不做任何事情。当第一次尝试访问磁盘时，会收到一个请求 *DEV_OPEN* 操作的消息，再间接调用 *w_do_open* 函数 (12 521 行)。*w_do_open* 调用 *w_prepare* 来确定请求的设备是否合法，然后调用 *w_identify* 来确定设备的类型，并初始化在数

组 *wini* 中的其他一些参数。最后，使用在数组 *wini* 中的计数器检测从 MINIX 3 启动以来是否第一次打开设备。检测完毕后，计数器加 1。如果是第一次 *DEV_OPEN* 操作，就调用 *partition* 函数（在 *drvlib.c* 中）。

下一个函数 *w_prepare*（12 577 行）接收一个设备的次设备号或使用分区的整型参数 *device*，并返回一个指向 *device* 结构的指针，指出设备的基址和大小。在 C 语言中，使用一个标识符来命名一个结构并不妨碍使用相同的名字来命名一个变量。一个设备是驱动器分区还是子分区可以由次设备号确定。一旦 *w_prepare* 完成了它的工作，其他任何读写磁盘的函数就不需要关心它们和分区的关系。就像看到的那样，当执行 *DEV_OPEN* 请求时，将调用 *w_prepare*，该函数也是所有数据传输中使用的准备/传输循环的一个步骤。

各种软件兼容的 AT 型磁盘已经使用了很长一段时间，*w_identify*（12 603 行）函数不得不把这段时间内引入的各种不同设计区分开来。第一步是首先在所有这一类磁盘控制器都应有一个地址检查是否存在一个可读写的端口。这是提到的用户空间驱动程序访问 I/O 端口的第一个例子，其操作比较容易描述。对于一个磁盘驱动程序，用一个在 12 201 到 12 208 行定义的指令结构完成输入/输出，这个结构中是一系列字节值。接下来会更详细地描述它，在此注意这个结构中的两个字节：一个是 *ATA_IDENTITY*，代表一个让 ATA（AT Attached）驱动程序标记自己的命令；另一个是一些模式位，用来选择驱动程序。然后 *com_simple* 被调用。

这个函数隐藏了以下所有细节：构建一个由 7 个 I/O 端口地址以及将要被写入的字节组成的向量，将这些信息发送给系统任务，等待一个中断，检查返回的状态。它测试驱动程序是否可用，使用位于 12 629 行的 *sys_insw* 内核调用读取一个 16 位的字符串。解析这个信息是一个非常复杂的过程，在此不做详细讨论。可以说这是一个非常庞大的信息，包括一个标识磁盘模型的字符串以及设备的物理柱面、磁头和扇区参数（报告的“物理”配置可能并不是真正的物理配置，但只能接受驱动器报告的参数）。磁盘信息也指出了磁盘是否允许进行逻辑块寻址（Logical Block Addressing，LBA）。如果能，则驱动程序可以忽略柱面、磁头和扇区参数，使用绝对的扇区号访问磁盘，从而简化访问磁盘的操作。

正如以前已经提到的那样，*init_params* 函数有可能没有从 BIOS 表中复原逻辑磁盘配置信息。如果确实如此，那么在 12 666 至 12 674 行的代码将试图根据它从驱动器本身读的参数来创建一合适的参数集。其思路是根据原始 BIOS 数据结构中对应域所允许的位数，把最大的柱面、磁头和扇区数分别取值为 1023, 255 和 63。

如果 *ATA_IDENTITY* 命令失败，可能仅仅意味着磁盘是不支持这个命令的过时模型。在这种情况下，仅仅可以得到以前 *init_params* 所读出的逻辑配置值。如果这些值有效，则把它们复制到 *wini* 中的物理参数域，否则返回错误信息，磁盘不能使用。

最后，MINIX 3 使用一个 *u32_t* 变量来对地址按字节计数，这限制了一个分区的大小不能超过 4 GB。不过用来记录分区地址和大小的 *device* 结构体（在文件 *drivers/libdriver/driver.h* 的 10 856 到 10 858 行定义）使用了 *u64_t* 类型的数字，同时使用 64 位的乘法来计算驱动器的大小（12 688 行），随后整个驱动器的地址以及大小被记录到 *wini* 数组中。接着调用 *w_specify*，如果必要的话，会调用两次，将这些参数回传给磁盘控制器（12 691 行）。随后会进一步调用内核调用：*sys_irqsetpolicy* 调用（12 699 行）保证当磁盘控制器发出一个中断然后中断被处理以后，会自动重新打开中断，以准备下一个中断；之后 *sys_irqenable* 调用实际打开一个中断。

w_name（12 711 行）返回一个指向设备名的字符串指针，它们是“AT-D0”，“AT-D1”，“AT-D2”或“AT-D3”之一。如果需要产生一条错误消息，那么这个函数能够指出消息是哪个设备产生的。

由于某些原因，驱动程序可能与 MINIX 3 系统不兼容。函数 *w_io_test* (12 723 行) 在第一次尝试打开一个设备驱动程序时对新驱动程序进行测试。这个例程以比正常操作等待时间更短的方式尝试去读取驱动器上的第一个块，如果测试失败，则驱动程序被永久标记为不可用。

w_specify (12 775 行) 除了向控制器传递参数外，还通过对零柱面寻道来重新校准驱动器（如果是以前的设备）。

do_transfer (12 814 行) 所做的正如它的名字的含义一样，它将指令集和为了传输一块数据（可能为 255 个磁盘扇区）所需要的所有字节值组合在一起，然后调用 *com_out* 将命令发送给磁盘控制器。根据磁盘寻址方式不同，即使用柱面、磁头、扇区寻址还是通过 LBA 寻址，将数据做相应的格式化。在 MINIX 3 内部将磁盘块按照线性编址，故如果支持 LBA，则最开始的三个字节是通过如下方法得到的：将扇区数向右移动适当的位数，然后使用掩码获得 8 位值。扇区数是一个 28 位数字，故最后的掩码操作使用一个 4 位的掩码 (12 830 行)。如果磁盘不支持 LBA 模式，则根据磁盘使用的参数计算柱面、磁头以及扇区数目 (12 833 到 12 835 行)。

在这里，代码隐含了一个未来可以改进的地方。以 28 位作为扇区数目的 LBA 寻址限制了 MINIX 3 只能够有效地使用最大 128 GB 的磁盘（可以使用更大的磁盘，但是 MINIX 3 只能够访问最开始的 128 GB）。程序员已经考虑使用最新的 **LBA48** 方式来寻址，不过还没有实现，LBA48 模式使用 48 位来对磁盘块编址。在 12 824 行有一个测试，以查看 LBA48 是否被启用。在目前描述的 MINIX 3 版本中，这个测试结果永远失败。这样处理较好，因为即使测试成功，也没有代码用来完成相应功能。注意，如果想要自己改变 MINIX 3 使得能够使用 LBA48，则不能只是改变这里的一些代码，而是需要对好多地方的代码进行修改，以能够处理 48 位寻址。或许在发行了针对 64 位处理器的 MINIX 3 版本后，这会更加容易。但是如果 128 GB 已经不能满足需要，则 LBA48 模式能够提供最大 128 PB (Petabytes) 的容量。

在此，我们简要看一下一个在比较高的级别数据传送是怎么被完成的。首先调用的是 *w_prepare*，这个函数在前面已经讨论过。如果传送操作请求多个块（即一个 *DEV_GATHER* 或者 *DEV_SCATTER* 请求），则随后马上会调用位于 12 848 行的 *w_transfer*。如果传送操作只请求一个单独的块（一个 *DEV_READ* 或者 *DEV_WRITE* 请求），则创建一个只含有一个元素的分散/集中向量，然后调用 *w_transfer*。与此同时，*w_transfer* 使用 *iovec_t* 请求。请求向量中的每一个元素由缓冲区地址和缓冲区大小组成，大小必须是磁盘扇区大小的倍数。所有其他需要的信息作为一个参数传递给调用的函数，影响整个请求向量。

所做的第一件事情是一个简单的测试，以查看传送最开始请求的磁盘地址是否相对扇区边界对齐 (12 863 行)。接着进入函数的外层循环，对于请求向量中的每一个元素循环一次。在循环中，和前面多次讨论的一样，在函数真正工作之前进行一系列测试。首先，将请求向量中每一个元素所包含的 *iov_size* 域加在一起，以计算请求的总字节数。检查这个结果以确认请求的总大小是扇区大小的倍数。其他测试检查请求的起始位置不在或者不超过设备的结尾，如果请求结束部分会超过设备的结尾，则请求的大小会被裁剪。至此所有的计算都是基于字节的，但是在 12 876 行使用 64 位运算计算磁盘上块的位置。注意，在此虽然名字是块，但是实际上是一些磁盘块，即大小为 512 字节的扇区，而不是在 MINIX 3 中使用的一般情况下大小为 4096 字节的“块”。完成这些以后还要做一些调整。每个驱动器一次能够传送的数据大小都有一个上限，如果有必要会根据这个值再次检查请求。验证磁盘已经被正确初始化（如果必要会再次对磁盘做初始化操作）后，对一块数据的请求由 *do_transfer* 函数来完成 (12 887 行)。

传输请求产生以后将会进入内层循环，对每一个扇区循环一次。无论是读操作还是写操作，每个扇区都会产生一个中断。对于读操作，中断意味着数据已经就绪并能够传送了。在 12 913 行的

`sys_insw` 内核调用让系统任务重复地读取指定 I/O 端口，将数据传送到特定进程数据空间中的虚拟地址中。对于写操作，顺序正好相反。下面几行的 `sys_outsw` 内核调用将一串数据交给控制器，当数据传送到磁盘上以后，从控制器发出一个中断。无论是读操作还是写操作，都要使用 `at_intr_wait` 来接收一个中断，例如在 12 920 行对于写操作就有一个。虽然会等待中断，但这个函数提供了一种方法使得异常发生或者中断不会到来时结束等待状态。`at_intr_wait` 也会读取磁盘控制器的状态寄存器，从而返回各种各样的代码。在 12 933 行有一个检测。无论是读操作还是写操作，一旦出现错误，就有一个 `break` 能够跳过相应区域，同时记录结果，指针和计数器会调整到下一个扇区。如果允许再重试一次，则内层循环下一次会重试同一个扇区。如果磁盘控制器报告一个坏扇区，则 `w_transfer` 立即中止。对于其他错误，仅仅增加一个计数器的值，如果没有达到设定的最大错误数目，`max_errors` 函数会继续执行。

下一个要讨论的函数是 `com_out`，它负责向磁盘控制器发送命令。不过在查看这部分代码之前，我们要先看一下在软件看来控制器是什么样子的。磁盘控制器由一组寄存器来控制，在一些系统中这些寄存器可以被映射到存储器中，不过在 IBM 兼容系统中，它们表现为 I/O 端口。在此，我们会查看这些寄存器，同时讨论一下如何使用它们（一般情况下为 I/O 控制寄存器）。在 MINIX 3 中这显得更加复杂，因为驱动程序运行在用户空间，从而不能执行读写寄存器的指令。在此，我们会观察在这个限制下如何使用内核调用来达到目的。

一个标准的 IBM-AT 系列硬盘控制器使用的寄存器如图 3.23 所示。

寄存器	读函数	写函数
0	数据	数据
1	错误	写预补偿
2	扇区数量	扇区数量
3	扇区数 (0~7)	扇区数 (0~7)
4	柱面数低位 (8~15)	柱面数低位 (8~15)
5	柱面数高位 (16~23)	柱面数高位 (16~23)
6	驱动器/磁头选择 (24~27)	驱动器/磁头选择 (24~27)
7	状态	命令

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = 柱面/磁头/扇区模式
 1 = 逻辑块寻址模式
 D: 0 = 主驱动器
 1 = 从驱动器
 HSn: CHS模式：CHS模式中选择的磁头
 LBA模式：选择块的24~27位

(b)

图 3.23 (a)IDE 硬盘控制器使用的控制寄存器，圆括号中的数字表示在 LBA 模式中各个寄存器于逻辑块地址中所占的位；(b) 驱动器 / 磁头选择寄存器的各个组成部分

前面已经多次提到对 I/O 端口读写，但是前面一直默认将其视为与存储器地址类似。事实上，I/O 端口一般情况下与存储器地址不同。首先，具有相同 I/O 端口地址的输入输出寄存器并不是同

一个寄存器，因此写入某一特定地址的数据不能在以后由读操作取出来。例如，对于图3.23中的最后一个寄存器地址，当对其读时为磁盘控制器状态，当对其写时为向磁盘控制器发命令。一般读写I/O设备寄存器会引发一个独立于数据传输细节的动作。AT磁盘控制器中的命令寄存器就是这样的。在使用时，通过把数据写入低编号的寄存器来选择读出或写入的磁盘地址，然后把操作码写入命令寄存器。写入命令寄存器的数据决定了要执行的操作是什么。将操作码写入命令寄存器将启动具体的操作。

也存在这种情况，即对于不同的操作模式，寄存器或寄存器中的某些域的使用是不同的。按图中给出的例子，向第6寄存器的第6位即LBA位写入0或写入1，将分别选用CHS（柱面/磁头/扇区）模式或LBA（Logical Block Addressing，逻辑块寻址）模式。向寄存器3, 4, 5写入的数据或从寄存器3, 4, 5读出的数据以及第6寄存器的低4位数据，由于LBA位的不同其解释是不同的。

在此通过调用*com_out*（12 947行）来研究命令是如何发送到控制器的。在建立好一个*cmd*结构体（如前所述，使用*do_transfer*）后会调用这个函数。在改变任何寄存器的内容以前，通过读状态寄存器来确定控制器不忙。这项工作是通过检查`STATUS_BSY`位而完成的。在这里速度很重要。一般磁盘控制器总是准备好的或在很短的时间内即可准备好，所以采用了忙等待的方法。在12 960行调用了*w_waitfor*来测试`STATUS_BSY`状态位。*w_waitfor*使用内核调用以要求系统任务读一个I/O端口，故*w_waitfor*能够用来测试状态寄存器中的一位。它执行循环测试直至条件为真或者预定义的超时时间到。循环被设计为如果磁盘准备好，则快速返回，因此，如果控制器准备好，那么将会在最短时间内返回真。如果是暂时性失效，则经过一段时间的延迟后返回真，如果在超过限定时间后还没有准备好，则返回假。在讨论*w_waitfor*时将进一步讨论超时问题。

一个控制器可以控制多个驱动器，所以一旦确定控制器准备好，就可以通过向控制器写一个字节来选择驱动器、磁头和操作模式（12 966行），然后再次调用*w_waitfor*。磁盘驱动器执行命令时，有时会失败或不能正常地返回一个出错代码。毕竟驱动器是机械设备，内部有可能发生各种机械故障。所以，作为一项保险措施，*sys_setalarm*内核调用用来使系统任务调度器安排一个对唤醒例程的调用。然后，通过首先向各种寄存器写入参数再向命令寄存器写入命令代码来发出命令。这项工作由*sys_voutb*内核调用完成，它向系统任务发送一系列由“值，地址”组成的数据对。系统任务依次将每一个值按照指定的地址设置到相应的I/O端口上。内核调用*sys_voutb*使用的数据向量由*pv_set*宏生成，这个宏的定义位于文件*include/minix/devio.h*中。向命令寄存器中写入操作码的动作使得一个操作开始。当动作完成时，会发生一个中断，同时发送一个通知消息。如果命令时间超时，则设置的警报时间就会超时，从而会发生一个同步警报通知以唤醒磁盘驱动器。

下面几个函数很短。如果在等待磁盘中断或者等待磁盘就绪时发生超时，则调用*w_need_reset*（12 999行）。*w_need_reset*的工作仅仅是对*wini*数组中每个驱动器的*state*变量做标志，使得下一次访问时强制进行初始化。

w_do_close（13 016行）对常规的硬盘几乎不做任何工作。当支持的设备为CD-ROM时，就需要一些代码来完成一些额外工作。

调用*com_simple*来向控制器发出一些没有数据传输阶段、可以立即终止的命令。属于这一类的命令包括取磁盘标识、设置一些参数和重新校准。*w_identify*就是一个例子。在调用之前，必须正确初始化指令结构。注意，在调用*com_out*之后，马上调用了*at_intr_wait*，最后*receive*会导致阻塞，直到通知中断发生的消息到达。

注意，在*com_out*要求系统任务写寄存器以设置和执行一个命令之前，调用了*sys_setalarm*内核调用。正如在本节多次讲到的，接下来的*receive*操作一般情况下会收到一个表示中断的通知。如果设置了一个警报但是没有中断发生，则接下来会触发`SYN_ALARM`消息。这时将会调用位于

13 046行的*w_timeout*。接下来要执行的动作取决于*w_command*中记录的当前动作。发生超时时有可能前一个操作已经结束，这时*w_command*中的值为*CMD_IDLE*，意味着磁盘完成了它的操作，这种情况下什么都不会做。如果命令没有完成，并且操作是读请求或是写请求，那么减小I/O请求的大小可能会有帮助。这项工作分两步完成，首先把可以请求的最大扇区数目减小为8，再减小到1，对于所有的超时，打印一条消息，在下一次试图访问磁盘时，调用*w_need_reset*强制重新初始化所有的驱动器。

当需要复位时，就调用*w_reset*（13 076行）。这个函数使用库函数*tickdelay*，它设置一个看门狗定时器，然后等待它被触发。经过使驱动器从以前的操作中恢复过来的初始延迟以后，选通磁盘控制器中的某一位，也就是把它提到逻辑电平1一段时间，然后恢复为逻辑电平0。执行这个操作以后，调用*w_waitfor*给驱动器一段合理的时间以便发信号使其准备好。如果复位不成功，就打印一条消息，返回出错状态。

发往磁盘的有关数据传输的命令一般通过产生一个中断而终止，该中断向磁盘任务发送一条消息。事实上，当每一个扇区被读出或写入时，都产生一个中断，函数*w_intr_wait*（13 123行）在一个循环中调用*receive*，如果接收到一个*SYN_ALARM*消息，则调用*w_timeout*。在这个函数中需要查看的唯一一个其他消息是*HARD_INT*。如果接收到这个消息，则会去读取状态寄存器，同时会调用*ack_args*以重新初始化中断。

*w_intr_wait*不直接被调用，当需要等待一个中断时调用下一个函数*at_intr_wait*（13 152行）。*at_intr_wait*接收到一个中断后会对驱动器状态位做快速检查。如果状态忙位、写失败位以及错误位都被清除，则一切正常，否则会进行更加详细的检查。如果根本不能读取寄存器，则进入崩溃时间。如果问题是一个坏扇区，则返回一个特定的错误，所有其他错误返回一个一般错误码。所有这些情况都会设置*STATUS ADMBSY*位，稍后由用户对其进行复位操作。

至此我们已经在几个地方见到调用函数*w_waitfor*（13 177行），以便在磁盘控制器状态寄存器的某一位上执行忙等待。这个函数适用于以下场合：被等待的位会在最开始的测试中被复位，同时假定测试很快会发生。为了追求速度，在以前的MINIX版本中使用一个宏来直接读取I/O端口。当然，在将驱动程序作为用户空间进程看待的MINIX 3环境下，这种方法是不可行的。这里的解决方法是，在进行最开始的测试之前，使用一个具有最小开支的*do...while*循环。如果测试的位被清除，则立即从循环中返回。为了能够处理可能出现的失败，在循环内部通过跟踪时钟设置了一个超时检测。如果超时发生，则调用*w_need_reset*。

*w_waitfor*使用的*timeout*参数在12 228行定义为300个时钟节拍，或者5秒。另一个相似的参数是*WAKEUP*（12 216行），其值取为31秒，时钟任务用它安排唤醒事件。考虑到普通进程在其被迫放弃CPU以前仅仅可以运行100 ms，那么这些参数对于忙等待而言，是很长的一段时间。但是，这些数值是基于已公布的AT类计算机硬盘接口标准的，这些标准指出了磁盘旋转到一定的速度所允许的最长时间为31秒。但实际上，这是最坏情况下的规范，在大多数系统中，仅仅在刚加电时或在很长时间不活动以后，才需要启动旋转加速，至少对于硬盘是这样的。对于CD-ROM或其他经常需要旋转加速的设备，考虑这一点非常重要。

在文件*at_wini.c*中还有一些其他的函数。函数*w_geometry*返回被选中硬盘设备逻辑上的最大柱面数、磁头数和扇区数。这些值是实际值，而不是像为RAM盘驱动程序那样构造出来的值。*w_other*函数是针对不认识命令和*ioctls*的一个拦截器。事实上，在目前的MINIX 3版本中没有使用它，或许应该从附录B的清单中删除它。如果在不期望中断的情况下发生了一个中断，则调用函数*w_hw_int*。前面提到的情况下，当中断发生以前，如果超时被触发，则会发生这种情况。这将引

发处于阻塞态正在等待中断的 `receive` 操作，但是对应的中断通知可能会被下一个 `receive` 操作接收。这时唯一要做的事情是调用 `ack_irqs`，以重新允许中断（13 297行）。它遍历所有的已知驱动器，使用 `sys_irqenable` 内核调用以确保中断被打开。最后，在文件 `at_wini.c` 结尾有两个奇怪的小函数：`strstatus` 和 `strerr`。它们使用在前面 13 313 到 13 314 行定义的宏将错误代码连接成字符串。这些函数在目前描述的 MINIX 3 中没有使用。

3.7.6 软盘处理

和硬盘驱动程序相比，软盘驱动程序更长，也更复杂。因为软盘的结构和硬盘相比要简单，所以这好像有点不合情理。但是，简单的机构具有一个简单的控制器，因此，操作系统就必须考虑更多的内容。可移动介质这一事实也增加了复杂性。在这一节中，我们将会讨论一些在处理软盘时必须考虑的问题。但在此处我们不会讨论 MINIX 3 的软盘驱动程序代码的细节。事实上，在附录 B 中没有列出软盘驱动程序，其重要的部分和硬盘是相似的。

对于软盘驱动程序，不必关心的一件事是支持多种控制器类型，而对于硬盘，这却是不得不处理的。尽管初始的 IBM PC 不支持当前使用的高密度软盘，但是单一的软件驱动程序可以支持 IBM 家族中各种计算机上的软盘控制器。这种和硬盘形成鲜明对照的情形可能是因为，对于软盘而言，没有像硬盘那样有提高性能的压力。在计算机系统中，软盘很少作为工作介质，且和硬盘相比，其速度和容量太有限了。对于新软件的发布和备份，软盘还是很重要的，但是由于网络和大容量的可移动存储设备变得越来越普遍，因此现在的个人计算机标准配置中很少带有软驱了。

软盘驱动程序不使用 SSF 算法或电梯算法，它是严格顺序执行的。在接收另一个请求以前，要完成已经接收的请求。在初始 MINIX 的设计中，由于 MINIX 是为个人计算机而设计的，大多数时间里仅仅有一个活跃的进程，所以设计者感到正在处理一个磁盘请求时，另一个磁盘请求到达的可能性很小。把各个请求排成一个队列需要显著地增加软件复杂性，但又不会带来什么益处。现在，软盘除了被配有硬盘的系统用来传入传出数据之外很少使用，这使其益处进一步降低。

这就是说，软盘和其他块设备一样也能处理分散的 I/O 请求。然而，对软盘驱动程序而言，请求数组比硬盘驱动程序要小，最大值为软盘上每条磁道的扇区数。

软盘驱动器的简单性使得软盘驱动程序复杂化。廉价的、缓慢的、低容量的软盘驱动器不值得配置硬盘所使用的复杂的集成控制器，因此驱动程序软件就不得不处理一些在硬盘中被隐藏于硬盘驱动器的操作。作为一个由软盘驱动器简单性而引起的驱动程序复杂性的例子，考虑在寻道过程中如何把读写磁头定位到特定的磁道。没有硬盘会要求驱动程序软件明确地调用寻道操作。对于硬盘而言，对程序员可见的柱面、磁头、扇区等几何结构和物理几何结构并无对应关系。事实上物理结构可能很复杂，典型情况下会有好多条带（柱面组），外部条带比内部条带要包含更多的扇区，然而，这对于用户而言是不可见的。作为对磁盘按柱面、磁道、扇区寻址的另一种方法，硬盘可以接受按磁盘上绝对扇区号编址的逻辑块地址（LBA）。即使采用柱面、磁道、扇区编址，由于磁盘的集成控制器计算把磁头移到何处，如果需要执行寻道操作，只要不访问任何不存在扇区，就可以使用任何几何结构。

然而对于软盘，寻道（SEEK）操作需要明确地编程，如果寻道失败，就必须提供一个例程来执行重校准（RECALIBRATE）操作，强迫磁头回到零柱面。这使得控制器有可能经过一定的步数把磁头移到需要寻道的位置。对于硬盘也需要相同的操作，但硬盘控制器执行了这些操作，而不需要驱动程序软件的指导。

使软盘驱动程序复杂化的一些因素是：

1. 可移动介质。
2. 多种磁盘格式。
3. 电机控制。

一些硬盘控制器也支持可移动介质，CD-ROM 驱动器就是一个例子。但是即使没有设备驱动程序软件的支持，驱动器控制器一般也能处理一些复杂的问题。然而对于软盘而言，则没有这些内置的支持，而实际上软盘更需要这些支持。软盘最常用的用处是安装软件和备份文件，经常需要从驱动器中取出软盘或把软盘插入驱动器。把想写入某张软盘的数据写入了另一张软盘会很不幸。设备驱动程序应尽最大的努力来防止这种情况的发生，虽然有时这是不可能的。因为，并不是所有的驱动器硬件都支持检测从上次访问磁盘以后驱动器是否被打开过。可移动介质可能引起的另一个问题是，如果系统试图访问一个没有插入软盘的驱动器，那么系统可能会挂起。如果能检测驱动器门是否打开，则可以解决这个问题。然而，并不总是能进行这种检测，所以如果对软盘的操作不能在一个合理的时间内终止，就必须提供一些措施实施超时处理并返回报错信息。

可移动介质可以被其他的介质代替。对于软盘，有许多不同的格式，IBM 兼容硬件支持 3.5 英寸和 5.25 英寸软盘驱动器；软盘可以按多种格式格式化，从 360 KB 到 1.2 MB（5.25 英寸软驱）或 1.44 MB（3.5 英寸软驱）。

MINIX 3 支持 7 种不同的软盘格式。对于不同格式引起的问题，有两种解决方法。一种方法是把每一种可能的格式视为一个特殊的驱动器，并为设备提供多个次设备号，老版本的 MINIX 就是这样实现的。在设备目录中，将会发现 14 个不同的设备，从第一个驱动器的 /dev/pc0（一张容量为 360 KB 的 5.25 英寸软盘）到第二个驱动器的 /dev/PS1（一张容量为 1.44 MB 的 3.5 英寸软盘）。注意，各种不同的组合是很麻烦的。因此，MINIX 3 提供了第二种方法。当用第一个 /dev/fd0 或第二个 /dev/fd1 首次访问软驱时，软盘驱动程序测试在驱动器中所访问的软盘，以确定其格式。一些格式的柱面比较多，另一些格式每道的扇区数比其他的格式多，通过逐步读更大的磁道和扇区数就可确定软盘的格式。通过一个淘汰过程，软盘格式即可确定下来。这种方法比较花费时间，不过在现代计算机中只有容量为 1.44 MB 的 3.5 英寸的磁盘可能被发现，故这种格式会首先被检测。另一个可能的问题是，如果一个磁盘有坏扇区，则有可能被识别错。可以使用工具程序来测试磁盘，不过在操作系统中自动做这些测试将会非常慢。

软驱的最后一个复杂问题是电机控制。如果不旋转，软盘就不能读写。硬盘被设计成连续运行几千小时也不会失效，但是电机永远旋转会使软驱和软盘很快失效。如果访问驱动器时电机未开，就需要发出命令启动电机，然后在等待半秒后再试图读写数据。开关电机是很慢的，所以 MINIX 3 每次使用完驱动器后，都继续使电机开几秒钟，如果在这段时间内再次使用，则把定时器再延长几秒。如果在这段时间内没有使用软驱，那么就关闭电机。

3.8 终端

数十年来，用户一直使用由键盘和显示器组成的设备与计算机进行通信，键盘用来做用户输入，显示器用来做计算机输出。多年以来，这两种设备被集成为独立式的设备（称为终端），它们使用电缆来和计算机连接。现在银行业和旅游业中使用的大型机有时候仍然使用这些终端，典型情况下它们通过调制解调器连接到大型机上，尤其当它们远离大型机时。当然，随着个人计算机的出现，键盘和显示器变成了独立的外设而不再是一个设备，不过由于它们是紧密相关的，在此仍然使用组合的名字“终端”，并把它们放到一起讨论。

从历史上讲，终端有大量不同的型号。我们需要终端驱动程序来屏蔽这些细节，从而对于不同型号的终端，就可不必重新编写操作系统和用户程序的设备无关部分。在下面几节中，我们首先一般性地讨论终端硬件和软件，然后讨论 MINIX 3 软件。

3.8.1 终端硬件

从操作系统的观点来看，根据操作系统如何和终端通信以及它们实际的硬件特性，终端可分成三类：第一类为内存映射（memory-mapped）终端，包括键盘和显示器，二者都直接与计算机相连。所有的个人计算机中键盘和显示器都采用了这种模式。第二类为使用RS-232标准的串行通信线，一般还经由调制解调器构成串行接口的终端。一些大型机中仍然使用这种模式，不过个人计算机也有串行线接口。第三类为通过网络连接到计算机上的终端。这个分类可以参考图 3.24。

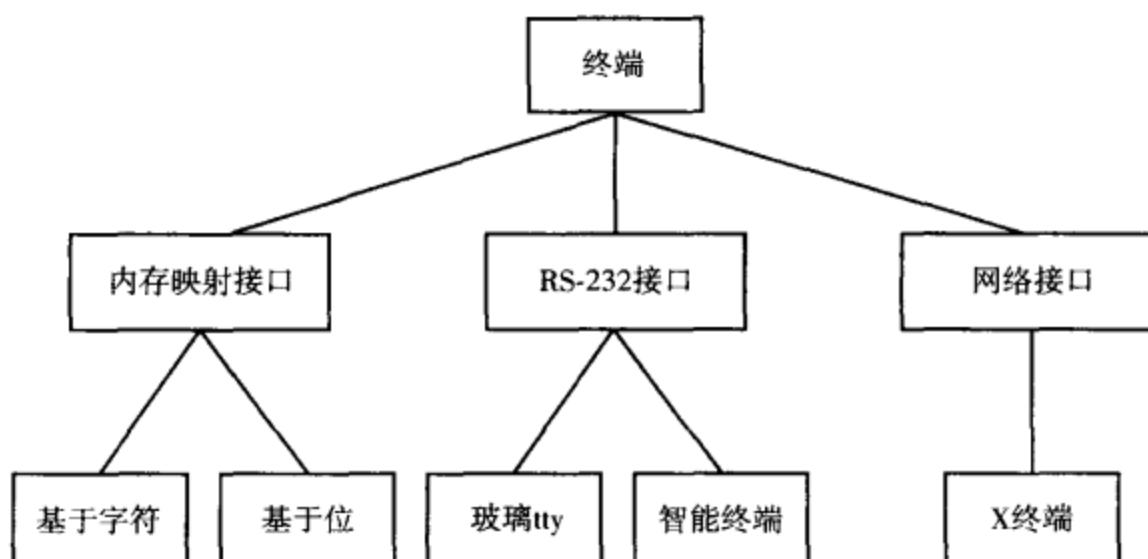


图 3.24 终端类型

内存映射终端

图 3.24 中的第一类终端是内存映射终端。这些终端是计算机整体的一部分，尤其对于个人计算机。它们由一台显示器和一个键盘组成。内存映射显示器使用称为视频 RAM（video RAM）的特殊存储器，视频 RAM 是计算机地址空间的一部分，我们可通过和其他地址空间一样的方式对它进行访问（见图 3.25）。

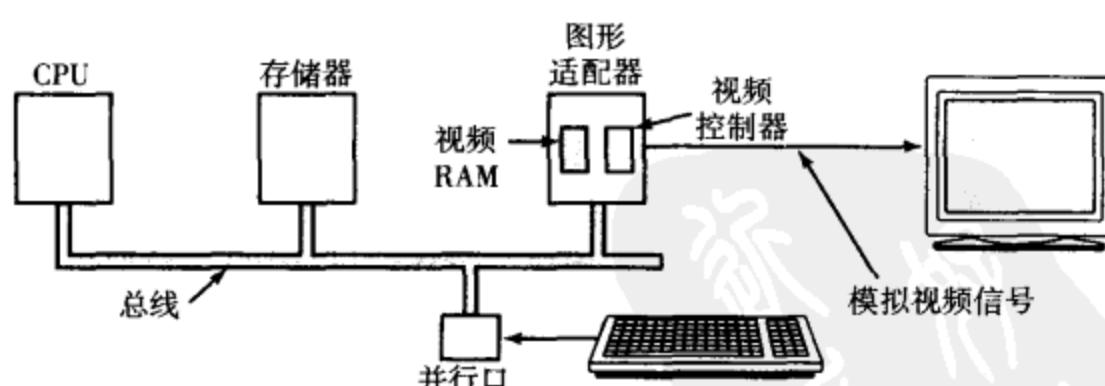


图 3.25 内存映射终端直接写入视频 RAM

视频 RAM 卡上有一个芯片称为视频控制器（video controller）。这个芯片从视频 RAM 中取出字符，产生用于驱动显示器（监视器）的视频信号。显示器一般有两种类型：CRT 监视器或者平板显示器。CRT 监视器产生电子束在屏幕上水平扫描，从而产生水平线。典型的屏幕从上到下有 480 至 1200 行，每行有 640 至 1920 个点。这些点称为像素（pixel）。视频控制器调节电子束的强度，决

定一个像素是亮的还是暗的。彩色监视器有三个电子束，分别对应于红色、绿色和蓝色，且各自可以独立调节。

平板显示器的内部工作原理与此大不相同，不过一个CRT兼容的平板显示器接受和CRT相同的同步信号以及视频信号，并使用它们来控制每一个像素位置的液晶单元。

一台简单的单色显示器可以把一个字符显示在宽度占9个像素、高度占14个像素（包括字符间的空白）的方框内，共显示25行，每行80个字符。此时这些显示器有350行扫描线，每行扫描线有720个点，每秒每帧会被重绘45~70次。视频控制器被设计成首先从视频RAM中取80个字符，产生14行扫描线，再取80个字符，再产生14行扫描线，这样一直工作下去。事实上，大多数视频控制器显示每个字符的每行扫描线时，都取一次字符，以便在控制器中不需要缓冲。每个字符的9列宽14行高的位组合保存在视频控制器的视频ROM中（也可以使用RAM以支持用户字体）。ROM按12位编址，8位来自字符代码，4位指定扫描线。ROM中每个字节的8位控制8个像素，字符间的第9个像素永远为空。因此，对于屏幕上的每行文本，需要 $14 \times 80 = 1120$ 次针对视频RAM的访问。对于字符生成器ROM，也需要访问相同的次数。

最初的IBM-PC有几种屏幕模式，在最简单的模式中，控制台使用字符映射显示器。在图3.26(a)中，可看到视频RAM的一部分。在图3.26(b)中，屏幕上的每个字符在RAM中占两个字节，低字节是显示字符的ASCII码，高字节为属性字节，用于指定颜色、反显、闪烁等。在这种模式下，满屏25行80列字符需4000字节的视频RAM。所有现代显示器仍然支持这种操作模式。

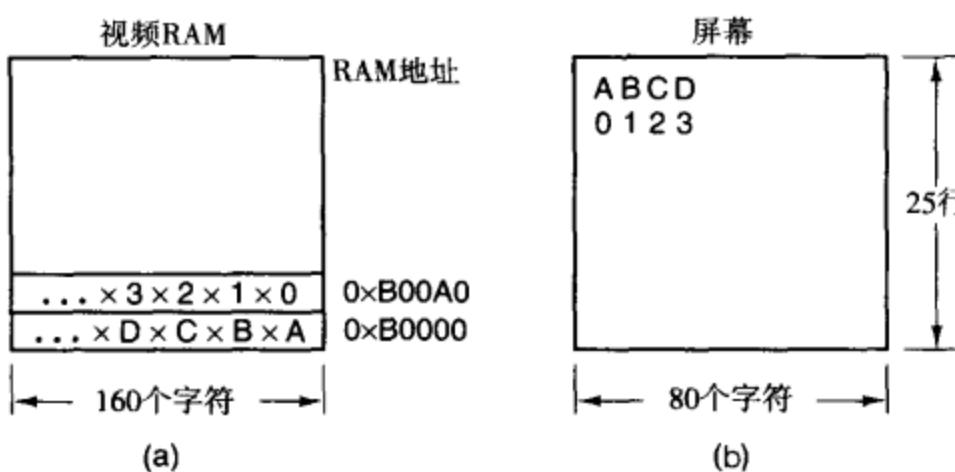


图3.26 (a) IBM单色显示器的视频RAM图像；(b)和(a)相对应的屏幕。×为属性字节

除了每个像素是独立控制的之外，位图模式显示器使用相同的原理。对于单显这种最简单的配置，每个像素对应视频RAM中的一位，在另一个极端，每个像素用24位的数来表示，红色、绿色和蓝色各8位。一个24位像素的 768×1024 彩色显示器需2 MB的RAM来存放图像。

对于内存映射显示器，键盘与显示器是完全分开的，它可能通过一个串行口或并行口和计算机相连。对于每一个键动作，产生CPU中断，键盘驱动程序通过读I/O端口取得键入的字符。

在IBM-PC中，键盘包括一个内嵌的微处理器，通过特殊的串行口和主板上的一个控制芯片通信。任何时刻按键或释放键，都产生一个中断，而且键盘仅仅提供键码，而不是ASCII码。当按A键时，键码(30)被存放于I/O寄存器中。输入字符是大写、小写、CTRL-A、ALT-A、CTRL-ALT-A还是其他的组合则由驱动程序确定。因为驱动程序知道哪些键被按下还没有释放（例如Shift键），因此它有足够的信息完成这项工作。虽然键盘接口把全部工作都交给了软件，但这也提供了很大的灵活性。例如，用户程序可能对一个数字是来源于最上面一行的键还是旁边的数字小键盘感兴趣。原则上，驱动程序可以提供这项信息。

RS-232 终端

RS-232 终端是包括一个键盘和一台显示器的设备，通过一次传输一位的串行口与计算机通信（见图 3.27）。这些终端使用 9 针或 25 针连接器，其中 1 针用于发送数据，1 针用于接收数据，1 针接地，其他各针用于各种控制功能，实际上大多数并未使用。为了向 RS-232 终端发一个字符，计算机必须一次传输一位，为了给字符定界，在字符前面加一个起始位，后接一个或两个终止位。可以在终止位前插入一个提供基本校验的奇偶位，但是通常仅在与大型主机系统通信时才需要这种技术。一般传输速率为 14 400 b/s 或 56 000 b/s，前面的速率一般用于传真，而后面的速率用于传送数据。RS-232 终端通常用于和远程计算机通信，两者之间使用调制解调器及电话线相连。

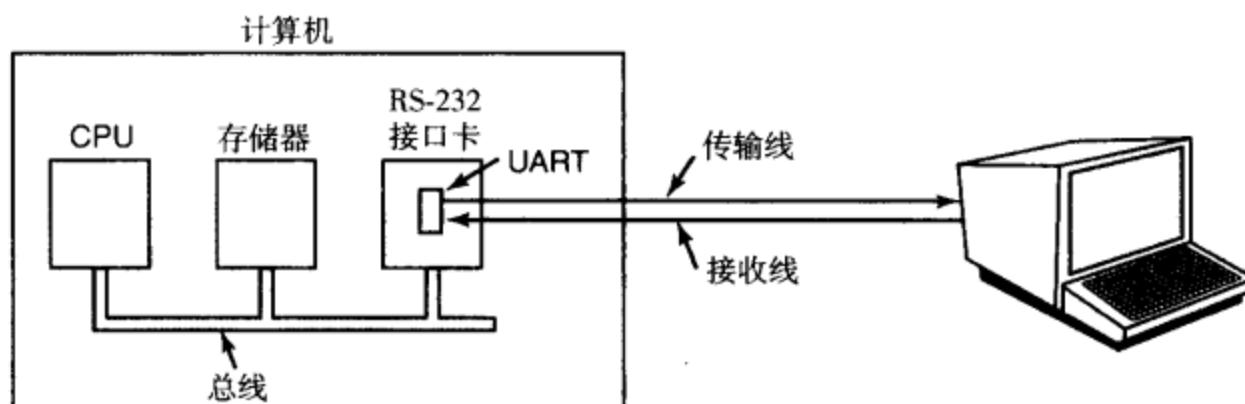


图 3.27 一个 RS-232 终端通过一条一次传输一位的通信线路和计算机通信。计算机和终端完全独立

计算机和终端在内部都是对整个字符进行操作的，但又必须通过串行线路以一次传输一位的方式来通信，因此开发了芯片来实现字符到串行口和串行口到字符的转换，该芯片称为 **UART**（通用异步收发器，Universal Asynchronous Receiver Transmitter）。UART 通过把 RS-232 接口卡插入总线和计算机相连，如图 3.27 所示。在现代计算机中，UART 和 RS-232 接口经常是主板芯片的一部分。板载的 UART 或许能够被禁用，以使用一个插在总线上面的调制解调器接口卡，或许它们两者可以共存。调制解调器也具有 UART（虽然它可能与其他功能整合在一起形成一个多功能芯片），不过它的通信通道借助于一条电话线而不是串行电缆。然而在计算机看来，所有的 UART 都是相同的，无论连接介质是专用的串行电缆还是一条电话线。

RS-232 终端正逐渐消失，而由个人计算机代替，但是它们仍然用于一些老的大型系统中，特别是银行、飞机订票等类似的系统中。不过使用一台远程计算机来模拟终端的终端软件仍然被广泛使用。

为了显示出一个字符，终端驱动程序把字符写入接口卡，这个字符被缓冲在接口卡中，然后通过 UART 在串行线上一位一位地发送出去。即使传输速率为 56 000 b/s，发送一个字符也至少需要 140 μ s。由于传输速率很低，一般情况下驱动程序向 RS-232 卡输出一个字符后就阻塞，等待字符传输完毕 UART 可以接受下一个字符时产生中断。正如其名称所暗示的，UART 可以同时发送和接收字符。当接收到一个字符时也产生中断，一般情况下 UART 能够缓冲少量的字符。当接收到中断时，终端驱动程序必须检查一个寄存器以确定中断源。一些接口卡有 CPU 和存储器，并能操纵多条线路，从而承担了许多原先由 CPU 完成的 I/O 工作。

正如前边提到的，RS-232 终端可以进一步分成几类。最简单的是硬拷贝终端。通过键盘键入的字符传输至主机，主机传出的字符打印在纸上。这些终端已过时并很少见到。

哑 CRT 终端也按这种方式工作，只是用屏幕代替了纸，因为在功能上和硬拷贝 tty 是一样的，这些终端常被称为“玻璃 tty”（术语 tty 是 Teletype 的缩写，Teletype 是一家公司，过去该公司是计算机终端企业的先驱。现在 tty 代表任何终端）。玻璃终端也已过时。

智能终端事实上是微缩的专用计算机，它们有CPU、存储器和软件，软件一般在ROM中。从操作系统的观点来看，玻璃终端和智能终端的不同在于，后者可以理解特殊的转义字符序列，例如通过发送ASCII ESC字符(033)后接各种其他的字符可以把光标移至屏幕上的任何位置、在屏幕中间插入文本等。

3.8.2 终端软件

键盘和显示器从大多数意义上讲是独立的设备，故在此分别讨论它们（它们并不是完全独立的，因为键入的字符必须在显示器上显示出来）。在MINIX 3中，键盘和显示器是同一任务的不同部分；在其他的系统中，它们可能分成不同的驱动程序。

输入软件

键盘驱动程序的基本工作是收集从键盘输入的信息，当用户程序读终端时，把它传输给用户程序。对于键盘驱动程序的设计可以采用两种思想：第一，驱动程序的工作仅仅是接收输入，并且不经任何修改就向上层传递。一个从终端读的程序得到原始的ASCII码序列（为用户程序提供键码太原始，也过分依赖于机器）。

这种思想非常适用于像*emacs*这样复杂的屏幕编辑器的需要。*emacs*允许用户把任意动作捆绑到任意字符或字符序列上。然而，如果用户键入了`dste`而不是`date`，然后键入三个退格键和`ate`键来对此进行修正，最后键入一个回车键，则意味着用户程序将收到键入的全部11个ASCII码。

大多数程序不需要这么多细节，它们仅仅希望得到正确的输入，而不是如何生成它的完整的序列。基于这一点，产生了第二种思想：驱动程序处理行内的编辑，仅仅向用户程序传输正确的一行。第一种思想是面向字符的，第二种思想是面向行的。这里把它们分别称为生模式（Raw Mode）和熟模式（Cooked Mode）。POSIX标准使用了一个不太形象的术语规范模式（Canonical Mode）来描述面向行的模式。在大多数系统中规范模式指的是定义好的配置。非规范模式（Noncanonical Mode）指的是原始模式，不过许多终端行为细节能够被改变。POSIX兼容的系统提供了几个库函数以支持选择哪种模式，同时能够改变终端配置。在MINIX 3系统中，`ioctl`系统调用支持这些函数。

键盘驱动程序的第一个任务是收集字符。如果每次击键产生一个中断，则驱动程序可以在中断过程中获得字符。如果中断被低层软件变成了消息，那么可以把最新获得的字符放入消息中，也可以放在内存的一个小缓冲区中，再由消息告诉驱动程序某件事情发生。如果一条消息只能发送给一个等待的进程并且存在键盘驱动程序忙于处理前一个字符的可能性，则后一种方法实际上更加安全。

一旦驱动程序接收到了字符，它就必须开始处理它。如果键盘传过来的是键码而不是应用软件使用的字符代码，那么驱动程序就必须使用一个表格，对其进行转换。并不是所有的IBM“兼容机”都使用标准键码，所以如果驱动程序希望支持这些机器，则必须利用不同的表格进行不同的映射。一种简单的办法是在驱动程序中编辑一个表格，以进行键盘提供的代码和ASCII码（美国信息交换标准代码）之间的映射，但是这对于非英语的用户是不令人满意的。在不同的国家，键盘的安排是不同的，即使对于西半球的大多数人而言，ASCII码集也是不够的。西班牙语、葡萄牙语和法语需要英语中不使用的标点符号和重音字符。为了满足键盘布局的灵活性以提供支持不同语言的需要，许多操作系统提供了可装载的键位表（Keymap）或代码页（Code Page），从而可以选择键码和传输给应用程序的代码之间的映射。这可以在系统启动时实现，也可以在启动后实现。

如果终端工作于规范模式（熟模式），则必须保存字符直至累积到一行，这是因为用户以后可能决定删除其中一部分。即使终端工作在生模式，程序也可能还未请求输入，所以字符也必须缓冲

起来以便允许用户提前输入(那些不允许用户提前键入的系统设计者应该受到惩罚,或者更加严重的是强迫他们使用他们自己的系统)。

一般有两种字符缓冲的方法。在第一种方法中,驱动程序包含一个中心缓冲池,每个缓冲区大约可存放10个字符。每个终端有一个数据结构与之相联系,在该数据结构中除了其他内容外,包含一个缓冲链指针,而缓冲链中的各缓冲区存放从该终端上输入的各个字符。用户键入的字符越多,用于输入的缓冲区就越多,挂在链上的缓冲区也就越多。当字符传送给用户程序时,缓冲区被删除并且放回到中心缓冲池中。

另一种方法是直接在终端数据结构中实施缓冲,这时没有中心缓冲池。因为常见的情况是用户输入了一条需要花费一段时间执行的命令(例如一个编译过程),然后用户又提前键入了几行命令,所以为了安全起见,驱动程序应该为每一个终端分配例如200个字节的空间。在拥有100台终端的大型分时系统中,为提前键入而始终分配20KB的缓冲空间显然是不必要的浪费,所以分配具有大约5KB空间的中心缓冲池可能就足够了。但是,为每个终端指定一个专用缓冲区可使得驱动程序比较简单(没有链接表管理),在具有一两个终端的计算机上,用户更偏爱这种方法。图3.28显示了这两种方法的不同。

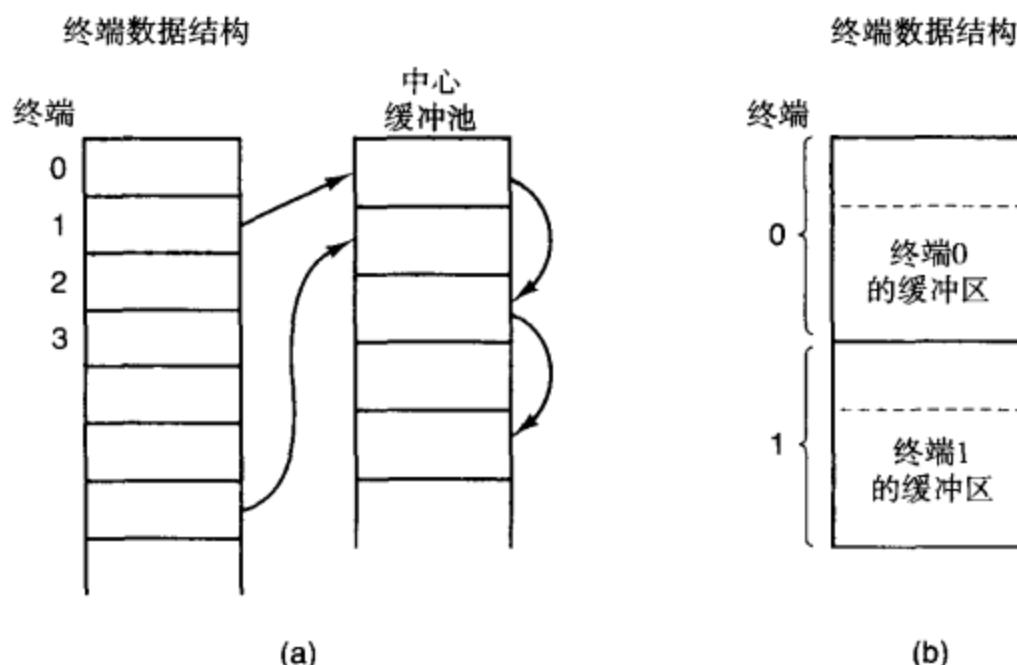


图3.28 (a)中心缓冲池;(b)每个终端的专用缓冲

尽管键盘和显示器在逻辑上是分立的设备,但是许多用户已习惯于在屏幕上看见他们刚刚键入的字符。某些(老式的)终端通过自动(用硬件)把刚键入的内容显示出来而做到了这一点,然而这样做不但在输入口令时有所妨害,而且极大地限制了复杂编辑器和应用程序的灵活性。幸运的是,个人计算机键盘在输入时不显示任何内容,而由软件来显示输入的内容,这个过程称为回显(Echoing)。

当用户击键时,程序可能正在写屏幕,这使得回显过程复杂化了。至少,键盘驱动程序要计算出新输入字符的显示位置,使其不被应用程序的输出所覆盖。

在每行80个字符的终端上,键入超过80个字符时,回显也变得复杂了。根据应用程序,转入下一行可能是合适的。某些驱动程序只是通过丢弃超出80列的所有字符而将每行截断到80个字符。

另一个问题是Tab键的处理。大多数键盘都有一个Tab键,但是几乎没有终端能处理Tab键的输出。通常由驱动程序来计算光标当前定位在什么位置,它既要考虑应用程序的输出,又要考虑回显的输出,并且需要计算要回显的正确的空格字符个数。

现在讨论设备等效性问题。从逻辑上说，在每一文本行的末尾，需要一个回车键把光标移到第一列，然后用一个换行键前进到下一行。让用户在每一行末尾键入回车和换行这两个键是不合适的（虽然有些终端有产生这两个字符的键，但应用程序仅有 50% 的可能需要它）。驱动程序通常把输入的内容转换成操作系统使用的内部标准。

如果标准格式仅仅存储换行（UNIX 相关系统约定），那么回车应被转换为换行。如果内部格式存储回车和换行，那么驱动程序应该在键入回车时生成一个换行符，键入换行时生成一个回车符。无论在内部如何转换，为了使屏幕能正确更新，终端可能要求回显回车和换行。因为一台大型计算机可能和大量不同的终端相连，所以需要键盘驱动程序把不同的回车/换行组合转换成内部系统标准并正确处理回显。

一个相应的问题是回车换行的定时问题。在一些终端上，显示一个回车或换行比显示字符或数字的时间长。如果终端内的微处理机不得不复制一大块文本来实现滚动一行，那么换行可能会慢一些。如果机械打印头不得不回到打印纸的左边，那么回车会慢一些。在这两种情况下，都需要驱动程序向输出流中插入填充字符（Filler Character，即假的空字符）或者等待足够长的时间，以使终端来得及响应。延迟时间一般与终端的速度有关，例如，4800 b/s 或更低的速率可能不需要延迟，但在 9600 b/s 或更高的速率，则可能需要一个填充字符。如果硬件支持 Tab，特别是硬拷贝，则输出 Tab 后也需要一个延迟。

当操作在规范模式时，许多输入字符有特殊的含义。图 3.29 显示了 POSIX 需要的特殊字符和 MINIX 3 识别的附加字符。其默认值都是和程序使用的代码和文本输入不相冲突的控制字符，但如果需要，除最后两个以外，它们都可以由 *stty* 命令重新设置。老版本的 UNIX 对这些字符中的大多数使用不同的默认值。

字符	POSIX名字	说明
CTRL-D	EOF	文件结束
	EOL	行结束（未定义）
CTRL-H	ERASE	后退一个字符
CTRL-C	INTR	中断进程（SIGINT）
CTRL-U	KILL	删除输入的一行
CTRL-\	QUIT	强制内核转储（SIGQUIT）
CTRL-Z	SUSP	挂起（被 MINIX 忽略）
CTRL-Q	START	开始输出
CTRL-S	STOP	停止输出
CTRL-R	REPRINT	重新显示输入（MINIX 扩充）
CTRL-V	LNEXT	下一字符（MINIX 扩充）
CTRL-O	DISCARD	抛弃输出（MINIX 扩充）
CTRL-M	CR	回车（不可改变）
CTRL-J	NL	换行（不可改变）

图 3.29 在规范模式下特殊处理的字符

ERASE 字符使用户删除刚刚输入的字符。在 MINIX 3 中，这是退格符（CTRL-H），它并不会把这个字符插入到字符队列中，而是从队列中移走前一字符。为了从屏幕上去掉一个字符，它应顺序回显三个字符：退格、空格和退格。如果前一字符为制表符，则需要跟踪在制表符前光标的位置才能将制表符删除。在大多数系统中，退格仅仅删除当前行的字符，不能删除回车回到上一行。

当用户在输入的一行中的起始部分发现一个错误时，删除整行重新输入是很方便的。*KILL*字符（在MINIX 3中为CTRL-U）删除当前行。MINIX 3使删除的行从屏幕上消失。但是在一些系统中，由于某些用户喜欢看到从前的一行，因此其处理是在后面放一回车和换行并回显之。所以如何处理*KILL*字符是一个喜好的问题。与*ERASE*字符一样，*KILL*通常也不可能从当前行进一步回退。当一个字符块被删除时，如果使用了缓冲，那么让驱动程序把缓冲区退还给缓冲池可能值得，也可能不值得。

有时*ERASE*或*KILL*字符必须作为普通数据输入。*LNEXT*字符起转义字符（Escape Character）的作用。在MINIX 3中，其默认值为CTRL-V。作为一个例子，在老版本的UNIX中，经常使用@作为*KILL*，但是Internet邮件系统使用形式为*linda@cs.washington.edu*的地址，对于老习惯感觉比较舒服的人可能会重定义*KILL*为@，但是在电子邮件地址中需要输入@，这可以通过键入CTRL-V@来实现。CTRL-V自身可以通过键入CTRL-V CTRL-V来输入。发现CTRL-V以后，驱动程序设置一个标志，表示对下一个字符不进行特殊处理。*LNEXT*字符自身不输入到字符序列中。

为了使用户能让屏幕图像停下来而不致滚出窗口以外，提供了用户冻结屏幕和以后重启屏幕的控制码。在MINIX 3中，分别是*STOP*（CTRL-S）和*START*（CTRL-Q），它们不被存储，但用来清除和设置终端数据结构中的一个标志。每当试图输出时，就检查这个标志。如果设置了这个标志，则不执行输出。通常回显也随程序输出一起被禁止。

杀死一个正在被调试的失控程序经常是有必要的，使用*INTR*（CTRL-C）和*QUIT*（CTRL-\）可以做到这一点。在MINIX 3中，CTRL-C将SIGINT信号发送到从该终端启动的所有进程。实现CTRL-C是相当需要技巧的，困难之处在于从驱动程序取得信息送给系统处理信号的那部分，而后者毕竟还没有请求这个信号。CTRL-\和CTRL-C相类似，但发送的是SIGQUIT信号。如果该信号未被捕获到或者被忽略，则强制进行内核转储。

无论键入哪个键，驱动程序都应回显一个回车换行，丢掉所有的输入以允许重新启动。历史上，DEL在许多UNIX系统中通常作为*INTR*的默认值。由于许多系统将DEL作为编辑时退格键的替代键，因此现在将*INTR*指定为CTRL-C。

另一个特殊字符为*EOF*（CTRL-D），在MINIX 3中，它使任何一个针对该终端的未完成的读请求以缓冲区中可用的任何字符来满足，即使缓冲区是空的。在一行的开头键入CTRL-D将使得程序读到0个字节，按照惯例该字符被解释为文件结尾，并且使大多数程序按照它们在处理输入文件时遇到文件结尾的同样方法对其进行处理。

某些终端驱动程序允许比在此讨论的更加精美的行内编辑。它们具有特殊的控制字符用于删除一个字，向前或向后跳过几个字符或字，定位到正在输入行的行首或行尾等。把所有这些功能加到终端驱动程序中会使它变得很大，而且在生模式下使用精美的屏幕编辑器时这无论如何也是一种浪费。

为使程序控制终端参数，在标准库中POSIX需要几个函数，最重要的两个函数为*tcsetattr*和*tcgetattr*。*tcgetattr*取回如图3.30中显示的数据结构的一份副本，该结构为*termios*，它包含用来改变特殊字符、设置模式和修改终端其他特性的所有信息。程序可检查当前的设置，当需要时对这些设置进行修改，然后调用*tcsetattr*把这个结构写回终端任务中。

POSIX并未规定这些需求是通过库函数实现还是通过系统调用实现，MINIX 3提供了一个系统调用*ioctl*，它通过以下的形式调用：

```
ioctl(file_descriptor, request, argp);
```

使用它可检查和修改许多I/O设备的配置。这个调用被用来实现`tcgetattr`和`tcsetattr`。变量`request`指出了是读还是写`termios`结构，在后一种情况下，则指出了该请求是立即起作用还是等到当前输出队列中的内容全部输出时才起作用；变量`argp`是指向调用程序中`termios`结构的一个指针。这种在应用程序和驱动程序之间的特殊通信方式是为了保持和UNIX系统之间的兼容性，而不是为了固有的美感。

```
struct termios{
    tcflag_t c_iflag;           /* 输入模式 */
    tcflag_t c_oflag;           /* 输出模式 */
    tcflag_t c_cflag;           /* 控制模式 */
    tcflag_t c_lflag;           /* 局部模式 */
    speed_t c_ispeed;          /* 输入速度 */
    speed_t c_ospeed;          /* 输出速度 */
    cc_t c_cc[NCCS];           /* 控制字符 */
};
```

图 3.30 `termios` 结构。在 MINIX 3 中 `tc_flag_t` 的类型为 short, `speed_t` 的类型为 int, 而 `cc_t` 的类型为 char

顺序讨论一下`termios`结构。4个标志字提供了很大的灵活性。在`c_iflag`中的各位控制处理输入的各种方法。例如`ICRNL`位使得输入中的`CR`字符转换为`NL`字符，这个标志在MINIX 3中默认被设置。`c_oflag`存放影响输出处理的各位，例如`OPOST`位允许输出处理。MINIX 3默认也设置了该位和`ONLCR`位，这两位使输出的`NL`字符转换为`CR NL`序列。`c_cflag`是控制标志字，MINIX 3的默认设置使一条线路接收8位字符，而且如果用户在一条线路注销，则使调制解调器挂起。`c_lflag`是局部模式标志域，`ECHO`位允许回显（登录时可以关闭回显，以提供登录时的安全性）。最重要的位为`ICANON`位，它使终端工作于规范模式，在关闭`ICANON`位情况下，还存在几种可能的设置。如果其他的设置都是默认值，则进入常规的`cbreak`模式。在这种模式下，输入的字符不必等待满行就传给了应用程序，但是，`INTR`, `QUIT`, `START` 和 `STOP` 还能够起作用。通过清除标志中相应的位，所有这些都可以禁止，其结果为传统的生模式。

各种可以改变的特殊字符，包括那些MINIX 3扩充的字符，都存放在`c_cc`数组中。该数组中也存放了两个使用在非规范模式中的参数，存储在`c_cc[VMIN]`中的数值`MIN`规定了`read`调用读出的最少字符数。`c_cc[VTIME]`中的值`TIME`设置了这些调用的时间极限。`MIN`和`TIME`的交互使用示于图 3.31 中，图中展示了请求`N`个字节的调用。如果`TIME = 0`且`MIN = 1`，那么其行为类似于常规的生模式。

	<code>TIME = 0</code>	<code>TIME > 0</code>
<code>MIN = 0</code>	从0到 <code>N</code> 个字节，无论可得到多少都立即返回	立即启动定时器，如果有输入则返回第1个字节，或者由于超时返回0个字节
<code>MIN > 0</code>	至少返回 <code>MIN</code> 个字节，最多返回 <code>N</code> 个字节，可能无限阻塞	输入第1个字节后启动输入字节定时器，未超时时如输入了 <code>N</code> 个字节就返回 <code>N</code> 个字节，超时时至少返回1个字节。可能无限阻塞

图 3.31 `MIN` 和 `TIME` 决定在非规范模式下当调用读操作时如何返回。`N` 是请求的字节数
输出软件

输出比输入简单，但RS-232终端的驱动程序和内存映射终端的驱动程序基本上是不同的。一般用于RS-232的方法是为每一终端设置缓冲，缓冲可以来源于同时作为输入缓冲的缓冲池，也可

以是像输入一样指定的专用缓冲。当程序向终端写时，首先把输出复制到缓冲。同样，回显输出也复制到缓冲。当所有的输出复制到缓冲以后（或缓冲满），输出第一个字符，驱动程序睡眠，当中断发生时输出下一个字符，如此进行下去。

对于内存映射终端，有一个更加简单的可行方案。需要打印的字符在某一时刻从用户空间中取出，直接放入视频 RAM 中。对于 RS-232 终端，每一个字符通过传输线送至终端。对于内存映射终端，一些字符需特殊处理，其中包括退格、回车、换行和响铃（CTRL-G）。一个内存映射终端的驱动程序必须在软件中跟踪视频 RAM 中的当前位置，以便在那里打印可打印字符，然后向前移动输出位置。空格、回车、换行都需要相应地更新位置。Tab 也需要特殊处理。

当在屏幕底部行的末尾输出换行时，屏幕必须上滚。为了分析滚动如何实现，可以看一下图 3.26。如果视频控制器永远从 0xB0000 开始读 RAM，那么在字符模式下滚动屏幕的唯一办法是从 0xB00A0 复制 24×80 个字符到 0xB0000（每个字符需两个字节），这是很费时间的。如果在位图模式下将会更加糟糕。

幸运的是，在这里大多数情况下硬件提供了一些帮助。大多数的视频控制器包含一个寄存器，这个寄存器控制在视频 RAM 中从何处开始取屏幕上的最顶行的字节。通过设置该寄存器指向 0xB00A0 来取代 0xB0000，以前的第二行移到了最顶行，且整个屏幕上滚一行。驱动程序必须做的其他工作仅仅是复制新的最底行的内容，当视频控制器指向了 RAM 的最高端后，它仅仅返回到最低地址并开始继续取数据。在位图模式下也存在类似的硬件支持。

驱动程序必须处理的另一个问题是内存映射终端的光标位置。硬件一般也用一个寄存器来提供帮助，该寄存器告诉光标去向何处。最后还有一个响铃问题，这通过向扬声器送一正弦波或方波来产生。扬声器是计算机中和视频 RAM 不同的部分。

值得注意的是，内存映射终端驱动程序所面临的许多问题（滚动、响铃等），RS-232 终端中的微处理器也会遇到。从微处理器的观点来看，它是带有内存映射显示器系统中的主处理器。

屏幕编辑和许多其他的复杂程序需要的屏幕更新方式要比仅仅把文本从底部上滚的方式更加复杂。为了支持这些功能，许多终端驱动程序支持大量的转义序列。许多终端驱动程序支持自己特殊的转义序列集，但是制定一个标准，使一个系统的软件能适应另一个系统是非常有用的。美国国家标准委员会（ANSI）定义了一套标准的转义序列，MINIX 3 支持 ANSI 标准的转义序列的一个子集，该子集示于图 3.32 中。对于许多常用操作，这已经足够了。当驱动程序看到开始转义序列的字符时，它设置一个标志并等待转义序列的其他部分进入系统。当全部进入以后，驱动程序必须在软件中实现转义序列指定的操作。插入和删除文本需要在 RAM 中移动大量字符。硬件除了滚动和显示光标外不能提供任何帮助。

3.8.3 MINIX 3 中的终端驱动程序简介

终端驱动程序包含在 4 个 C 文件中（如果支持 RS-232 和伪终端，则为 6 个文件），它们一起无疑构成了 MINIX 3 中最大的驱动程序。终端驱动程序既处理键盘，也处理显示器，二者都很复杂，同时又包括两个可选择终端，因此终端驱动程序是分成几部分来解释的。和调度器比较，终端驱动程序要比它大 30 倍，大多数人一定会对此感到惊讶（看一看大量关于操作系统的书籍，这些书籍中关于调度器的内容几乎是把所有 I/O 的内容加在一起的 30 倍，这一定会使这些人更吃惊）。

终端驱动程序接收十几种消息类型，其中最重要的有：

1. 从终端读（来自代表用户进程的 FS）。
2. 向终端写（来自代表用户进程的 FS）。

3. 为 `ioctl` 设置终端参数(来自代表用户进程的 FS)。
4. 键盘中断发生(有键被按下或者释放)。
5. 终止上一个请求(当信号发生时, 来自文件系统)。
6. 打开设备。
7. 关闭设备。

其余的消息类型用做特殊用途, 例如当功能键被按下时产生诊断显示, 或者触发崩溃转储。

转义序列	含义
<code>ESC [nA</code>	上移n行
<code>ESC [nB</code>	下移n行
<code>ESC [nC</code>	左移n行空格
<code>ESC [nD</code>	右移n个空格
<code>ESC [m; nH</code>	移动光标至(y = m, x = n)
<code>ESC [sJ</code>	从光标处开始清屏(0至行尾, 1到行首, 2全部)
<code>ESC [sK</code>	从光标处开始清除一行(0至行尾, 1至行首, 2全部)
<code>ESC [nL</code>	在光标处插入n行
<code>ESC [nM</code>	在光标处删除n行
<code>ESC [nP</code>	删除光标处的n个字符
<code>ESC [n@</code>	在光标处插入n个字符
<code>ESC [nm</code>	允许重显n(0=正常, 4=加粗, 5=闪烁, 7=反显)
<code>ESC M</code>	如果光标在顶行则回滚屏幕

图 3.32 终端输出驱动程序接受的 ANSI 转义序列。`ESC` 表示 ASCII 码转义字符(0x1B)。`n`, `m` 和 `s` 为可选的数值参数

除了不需要 `POSITION` 域外, 读写消息和示于图 3.17 中的消息具有相同的格式。对于磁盘, 程序必须指定它想读哪一块, 对于键盘, 却没有这项选择。程序永远取键入的下一个字符, 键盘不支持寻道。

`ioctl` 系统调用支持 POSIX 函数 `tcgetattr` 和 `tcsetattr`, 用来检查和修改终端属性(特性), 一种较好的编程实践是使用这两个函数和 `include/termios.h` 中的其他函数, 让库函数调用 `ioctl` 系统调用。有一些 MINIX 3 需要的控制操作在 POSIX 中没有提供, 装入可选择的键位表就是一个例子。对于这些问题, 程序员必须明确地使用 `ioctl`。

通过 `ioctl` 系统调用发往驱动程序的消息包括一个功能请求代码和一个指针。对于 `tcsetattr` 函数, 执行的是一个具有 `TCSETS`, `TCSETSW` 或 `TCSETSF` 请求类型和一个指向类似图 3.30 的 `termios` 结构指针的 `ioctl` 调用。所有这些调用把当前的属性集换成一个新的属性集。它们的区别在于: `TCSETS` 请求立即起作用, `TCSETSW` 请求直到所有的输出被传送后才起作用, `TCSETSF` 等待输出完成并抛弃掉还没有读的输入。`tcgetattr` 转换为一个具有 `TCGETS` 请求类型的 `ioctl` 调用, 并返回调用者一个填入 `termios` 结构的消息。因此, 可使用该调用检查当前设备的状态。有一些 `ioctl` 调用并不完全对应于 POSIX 定义的函数, 它传输的是另外几种结构的指针。例如, `KIOCSMAP` 请求用于装入一个新键位表, 于是需传输的是一个指向 `keymap_t` 的指针, 这是一个 1536 字节的结构(128 键 × 6 个转换键, 每个键码为 16 位)。图 3.39 概括了标准的 POSIX 调用是如何转换成 `ioctl` 系统调用的。

终端驱动程序使用的一个主要的数据结构为 `tty_table`。它是一个 `tty` 结构的数组, 每个终端一个。标准 PC 仅有一个键盘和一台显示器, 但 MINIX 3 可支持多达 8 个虚拟终端, 这取决于显示适

配器卡的存储空间大小。这样就允许使用控制台的用户登录多次，并且显示输出和键盘输入可从一个用户切换到另一个用户。对于虚拟控制台，按 ALT-F2 选择第二个虚拟控制台，按 ALT-F1 返回第一个虚拟控制台，也可使用 ALT 和箭头键。除此之外，串行线路也支持通过 RS-232 和调制解调器相连的两个远程用户，还支持用户通过网络相连接的伪终端。驱动程序被编写成很容易增加额外的终端。书中的源码示例的标准配置支持两个虚拟控制台，不支持串行线和伪终端。

tty_table 中的每个 *tty* 结构既跟踪输入，也跟踪输出。对于输入，它存放了一个所有已经键入但还没有被程序读出的字符队列，还存放了请求读但还未接收到字符的信息，以及超时信息。因此，如果没有键入字符也能请求输入，而且可以使任务不永久阻塞。对于输出，它存放了没有完成的写请求的参数，其他的域存放着各种通用变量。例如，前面讨论的 *termios* 结构，该结构影响输入输出的许多特性。在 *tty* 结构中还有一个字段，指向一类特殊设备所需的信息，但这些信息不是 *tty_table* 中所有设备都需要的。例如，控制台驱动程序和硬件有关的部分需要使用屏幕上和视频 RAM 中的当前位置，以及当前显示的属性字节，但对于 RS-232 传输线，则不需要这些信息。每种设备类型的私有数据结构也是从中断服务例程接收输入数据的缓冲区位置。慢速设备，例如键盘，不像快速设备那样需要很大的缓冲区。

终端输入

为了更好地理解终端驱动程序是如何工作的，我们首先看一下在终端上键入的字符是如何从系统传到需要它们的程序中的。虽然这一节旨在引入一个概述，但通过给出的行号参考信息能够帮助读者找到每一个使用的函数。或许这是一个很好的漫游信息，输入练习代码在文件 *tty.c*, *keyboard.c* 以及 *console.c* 中，这些文件都是大文件。

当用户在系统控制台上登录时，系统为其创建一个 shell，它用 */dev/console* 作为标准输入、标准输出和标准错误输出位置。shell 启动并试图通过调用库过程 *read* 读标准输入，这个过程把一条包含文件描述符、缓冲地址和字节数的消息发送到文件系统。这条消息如图 3.33 中的(1)所示。发送消息以后，shell 阻塞，等待应答（用户进程仅仅执行 *sendrec* 原语，这个原语把一个 *send* 和一个从进程发往的 *receive* 结合起来）。

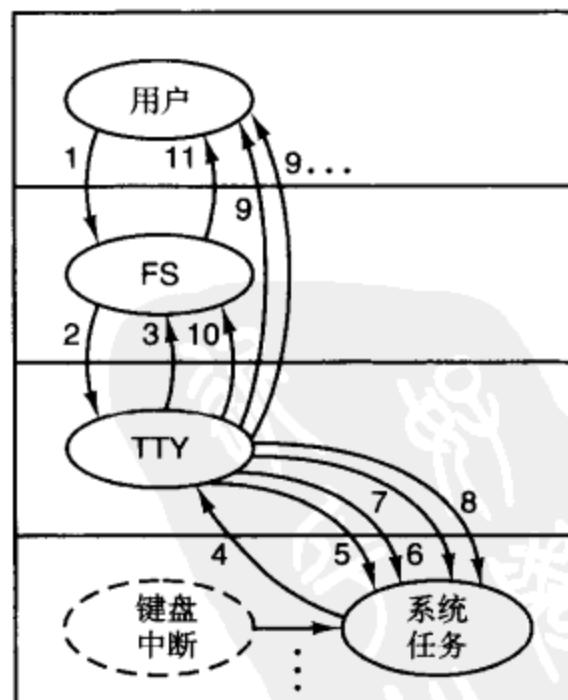


图 3.33 当还未输入字符时来自终端的读请求。FS 是文件系统，TTY 是终端驱动程序。TTY 为每次击键接收一条消息，并对输入的扫描码排队。之后，它们被中断并装入一个 ASCII 码缓冲，以便复制到用户进程

文件系统取消消息，并找到对应文件描述符的 i 节点，这个 i 节点对应一个字符设备文件 /dev/console，并且包含终端的主次设备号。终端的主设备号为 4；对于控制台，次设备号为 0。

文件系统在设备映射图 (device map) *dmap* 中查找终端驱动程序 TTY 的数目。然后，如图 3.33 中的(2)所示，向终端驱动程序 TTY 发一条消息。通常用户至此还没有键入，因此终端驱动程序不能满足这个请求。它立即发回一个应答，使文件系统解除阻塞，报告没有任何可用字符，这正如图中的(3)所示。文件系统在 *tty_table* 的控制台数据结构中记录一个进程等待终端（例如键盘）输入，然后去处理下一个请求。当然，用户 shell 会继续阻塞直至请求的字符到达。

当最终在键盘上键入字符以后，它引起两个中断，其中一个在按下键时发生，另一个在释放键时发生。在此很重要的一点是，PC 键盘不产生 ASCII 码，每当按下一个键时产生一个扫描码，释放对应键时产生一个不同的扫描码。对于同一个键，“按下”时产生的扫描码和“释放”时产生的扫描码的低七位是相同的。不同是最高的符号位，当键被按下时为 0，释放时为 1。这个规则对于像 CTRL 和 SHIFT 这样的转换键也是适用的。虽然最终这些键并不向用户进程返回任何 ASCII 码，但它们也会产生扫描码以指示哪个键被按下了（如果有必要，驱动程序能够区分出左边的 SHIFT 键和右边的 SHIFT 键），每个键仍要引起两个中断。

键盘中断为 IRQ 1。这个中断线在系统总线上不可访问，也不能和其他的 I/O 适配器共享。当 *hwint01* (6535 行) 调用 *intr_handle* (8221 行) 时，不需要遍历很长的分支就能够找到需要通知的 TTY。由于 *system/do_irqctl.c* 中的 *generic_handler* 产生消息，故在图 3.33 中显示系统任务产生通知消息(4)，不过这个例程直接被下层的中断处理例程调用。系统任务进程本身并不活跃。一旦获得了 HARD_INT 消息，*tty_task* (13 740 行) 就会引发 *kbd_interrupt* (15 335 行)，它最终调用 *scan_keyboard* (15 800 行)。*scan_keyboard* 产生三个内核调用(5, 6, 7)，以使得系统任务读写一些 I/O 端口，最终返回扫描码，然后被添加到环形缓冲区中。然后设置 *tty_events* 标记，以指示缓冲区含有字符而非空。

从这个观点上讲，并不需要消息。每当 *tty_task* 的主循环开始另一个循环时，针对每一个终端设备检查 *tty_events* 标记，对于每一个设置了这个标记的设备调用 *handle_events* (14 358 行)。*tty_events* 标志指出了各种活动（虽然大多数为输入），因此 *handle_events* 永远调用和设备有关的用于读写的函数。对于来自键盘的输入，将调用 *kb_read* (15 360 行)，该函数跟踪指示按下和释放 CTRL、SHIFT 和 ALT 键的键码，把键码转换为 ASCII 码。*kb_read* 转而调用 *in_process* (14 486 行)，它处理 ASCII 码，它要考虑特殊字符和可能设置的不同标志，包括规范模式是否有效。虽然一些代码例如 BACKSPACE 有其他的功能，但是其一般处理结果是在控制台输入队列 *tty_table* 中添加字符。通常 *in_process* 还要在显示器上回显相应的 ASCII 码。

当输入了足够多的字符时，终端驱动程序调用另一个内核调用(8)，令系统任务将数据复制到 shell 指定的地址中。这个操作也不是通过消息传递来实现的，正因为如此，在图 3.33 中通过虚线(9)表示。这样的线显示了若干条，这是因为在用户请求完全满足以前，可能有多个这样的操作。当这个操作最后完成时，终端驱动程序向文件系统发一条消息，告诉它工作已完成(10)，文件系统响应这条消息，向 shell 发一条消息来使其解除阻塞，如图中的(11)所示。

怎样才算输入了足够多的字符呢？这取决于终端模式。在规范模式，当收到换行、行结束或文件尾代码时，请求就完成了。为了对输入进行合适的处理，输入的一行不能超过输入队列的大小。在非规范模式，读可以请求大量的字符，在表明操作已完成的消息返回文件系统前，*in_process* 可能不得不传输多次字符。

注意，终端驱动程序把实际的字符直接从它自己的地址空间复制到 shell 空间中，它并不首先通过文件系统。对于块 I/O，数据确实通过了文件系统，以维护一个最近使用数据块的缓冲。如果请求块正好在缓冲中，那么请求可以直接由文件系统满足，而不需要做任何磁盘 I/O。

对于键盘I/O，缓冲毫无意义，而且文件系统到磁盘驱动程序的请求总可以在几百毫秒内得到满足，因此使文件系统等待没有什么损失。键盘I/O可能需几个小时才能完成，甚至永远不能完成（在规范模式，终端驱动程序等待一个完整的行；在非规范模式，可能等待的时间更长，这取决于MIN和TIME的设置）。因此，使文件系统阻塞直至终端输入请求满足是不合适的。

此后，用户可能提前键入了一些字符，这些字符在请求（前面的中断和事件4）以前就已准备好。在这种情况下，事件1, 2, 5和11在读请求以后相继发生，3则不发生。

对早期MINIX版本熟悉的读者可能记得在那些版本中TTY驱动程序（以及所有其他的驱动程序）和内核编译在一起。每个驱动程序在内核空间拥有自己的中断处理。对键盘驱动程序来讲，中断处理器自己能够缓冲一定数目的扫描码，同时可以进行一些前期处理（丢弃大多数键释放时产生的扫描码，只有像SHIFT这样的转换键的释放扫描码被缓冲）。中断处理本身并不向TTY驱动程序发送消息，因为TTY可能没有阻塞在receive上，故不能在任何时候接收消息。替代方法是，时钟中断周期性地唤醒TTY驱动程序。使用这种技术可以避免丢失任何键盘输入。

前面曾经介绍过处理诸如磁盘控制器产生的期待中断和处理诸如键盘产生的不可预知中断的不同。但是在MINIX 3中为了处理不可预知中断几乎不需要做任何特殊处理。为什么这是可行的呢？需要牢记的是，早期版本MINIX所针对的计算机与现在设计所面对的计算机之间性能存在很大差别。CPU时钟频率提高了，执行一条指令所需要的时钟循环数目减少了。为了使用MINIX 3，所推荐的最低处理器为80386。一台较慢的80386执行指令的速度大约为原始IBM PC速度的20倍。一台100 MHz的奔腾机的执行速度大约是较慢80386的25倍。故可以认为CPU速度已足够快。

另一件需要注意的事情是，与计算机标准相比，键盘输入非常慢。以每分钟100个字的速度，打字员每秒的输入不超过10个字符，即使对于快速的打字员，对于在键盘上输入的每个字符，也会向终端任务发一条消息。对于其他输入设备，可能具有更高的数据速率，如一个连到56 000 b/s调制解调器的串行口可能比打字员快1000倍或更多。以这样的速度，在两个时钟节拍之间，调制解调器大约收到120个字符。但是考虑到在调制解调器链路上的数据压缩，连到调制解调器上的串行口必须能处理至少两倍的字符。

不过对于串行端口需要考虑的一件事情是，传送的是字符而不是扫描码，故即使对于一个老的没有缓冲的UART，每一次按键都只会发生一个中断而不是两个。在较新的个人计算机上，配置的UART一般具有一个大小为16个字符的缓冲区，或许缓冲区有128个字符那么大。这种情况下没有必要针对每一个字符产生一个中断。例如，具有16个字节缓冲的UART可能被配置为当缓冲区中字符数达到14个时产生一个中断。以太网能够以比串行线更高的速率传送字符，不过以太网适配器能够对整个包进行缓冲，故每个包只需产生一个中断。

在结束终端输入概述时，我们总结一下当终端驱动程序被第一个读请求激活且接收到键盘输入以后被再次激活时发生的事件（参见图3.34）。在第一种情况下，当消息进入终端驱动程序请求从键盘读字符时，主过程tty_task（13 740行）调用do_read（13 953行）来处理这个请求，如果没有足够的缓冲字符，那么do_read把调用参数存入tty_table的键盘条目中。

然后它调用in_transfer（14 416行）取得任何已在等待的输入，再调用handle_events（14 358行），该函数接着调用kb_read（15 360行）[通过函数指针(*tp->tty_devread)]，并再次调用in_transfer试图再取得几个字符。kb_read调用了几个在图3.34中没有显示的例程来完成它的工作，其结果是把立即可得到的都复制给用户，如果什么也得不到，就什么也不复制。如果in_transfer或handle_events完成了读操作，当所有字符已被传输时向文件系统发一个消息，因此文件系统可以使调用者解除阻塞。如果读还未完成（没有字符或没有足够的字符），那么do_read报告文件系统，告诉它是否应该挂起初始调用者，或者如果请求的是非阻塞读操作，那么取消该读请求。

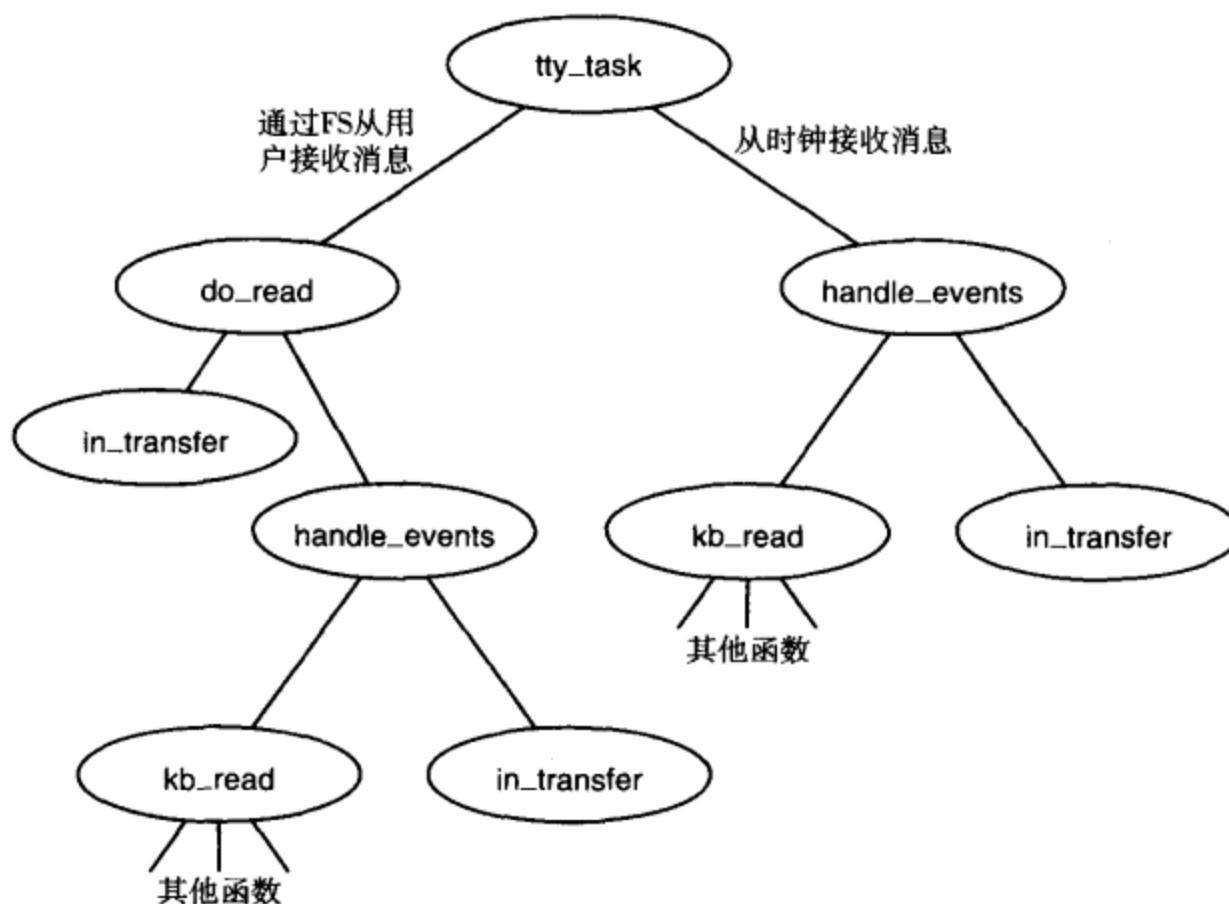


图 3.34 终端驱动程序中的输入处理。树左侧的分支适用于处理读字符的一个请求, 树右侧的分支在一个用户已请求输入前一个键盘消息发送到驱动程序时采用(图 3.X 将被修改)

图 3.34 的右边总结了键盘中断以后终端任务被唤醒时发生的事件。当键入一个字符时, 中断“处理器”`kbd_interrupt`(15 335 行)调用`scan_keyboard`, 它再调用系统任务进行输入/输出(此处的“处理器”加了引号, 这是因为当中断来到时它不是一个真正的中断处理器, 它是被系统任务中`generic_handle`发送给`tty_task`的消息激活的)。`kbd_interrupt`将接收到的字符码放入键盘缓冲`ibuf`, 同时设置一个标志指示控制台设备发生了一个事件。当`kbd_interrupt`返回控制到`tty_task`时,`continue`指令会导致开始下一次主循环。所有设备的事件标记都会被检查, 然后对每一个设置了标记的设备调用`handle_events`。如果是键盘, 就像接收到初始读请求一样, `handle_events`调用`kb_read`和`in_transfer`。在 FS 处接到第一条消息后, 图右侧所示的事件可能发生若干次, 直到有足够的字符能满足`do_read`所接收的请求为止。如果 FS 在第一个请求结束之前试图启动同一设备的另一个请求以读取更多字符, 则将返回一个错误。当然, 各个设备都是独立的, 远程终端上的用户的读请求和控制台上的用户的读请求是分开进行处理的。

图 3.34 中未给出的由`kb_read`调用的其他函数包括: 把硬件产生的键码(扫描码)转换为 ASCII 码的`map_key`(15 303); 跟踪转换键(如 SHIFT 键)状态的`make_break`(15 431 行); 处理各种复杂情况的`in_process`(14 486 行), 这些情况包括, 用户试图用退格键覆盖错误输入、其他特殊字符以及不同输入模式下的可用选项。`in_process`也调用`tty_echo`(14 647 行), 所以键入的字符也会显示在屏幕上。

终端输出

一般来说, 控制台输出比终端输入简单, 这是因为操作系统拥有控制权, 不需要考虑意外的输出请求。另外, 由于 MINIX 3 控制台是内存映射显示器, 所以向控制台输出就显得特别简单。输出的基本操作是把数据从一个内存区复制到另一个内存区, 不需要任何中断。另一方面, 显示管理的所有细节工作, 包括转义序列的处理, 都必须由驱动程序软件处理。和上一节讲述键盘输入时一

样，这里将跟踪向控制台发送字符时涉及的每个步骤。假设在例子中正被写入的是当前活动的显示器，如果考虑虚拟控制台设备，那么情况将会有点复杂。这种情况将在后面讨论。

一个进程在试图显示时，一般要调用 `printf`。`printf` 调用 `write` 向文件系统发送一条消息。该消息包含一个指向待显示字符序列的指针（不是字符序列本身）。接着文件系统向终端驱动程序发送一条消息，终端驱动程序取出字符并将其复制到视频 RAM 中。图 3.35 绘出了输出时涉及到的主要过程。

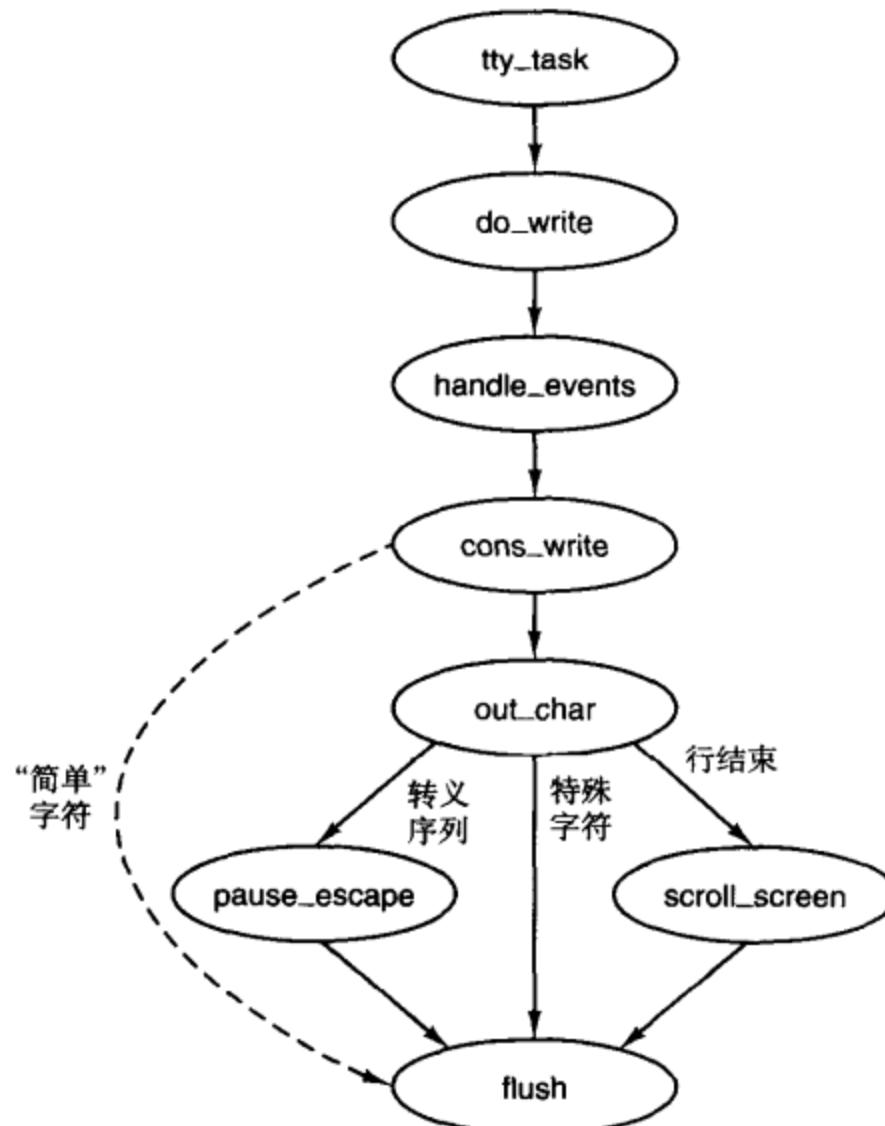


图 3.35 终端输出中使用的主过程。虚线表示由 `cons_write` 直接复制到 `ramqueue` 的字符

当终端驱动程序收到一条请求在屏幕上输出的消息时，它调用 `do_write` (14 029 行) 来存储 `tty_table` 中控制台的 `tty` 结构参数。然后调用 `handle_events` (只要 `tty_events` 标志被置位，就调用这个函数)。每次调用时这个函数都会调用由它的参数指定的设备的输入和输出例程。在讨论控制台显示时，这意味着任何处于等待状态的键盘输入将被首先处理。在有输入等待时，则待回显字符被添加到任何正在等待输出的字符后面。然后执行一个对 `cons_write` 的调用 (16 036 行)。`cons_write` 是一个内存映射显示过程，它使用 `phys_copy` 把字符块从用户进程复制到局部缓冲区。由于局部缓冲区只有 64 字节，所以这个过程和以下各步骤可能要重复若干次。当局部缓冲区满时，每个 8 位字节被传送到另一个缓冲区 `ramqueue` 中。`ramqueue` 是一个 16 位的字数组，交替地用屏幕属性字节的当前值填充。屏幕属性字节决定了字符的前景色、背景色以及其他属性。如果可能，字符就被直接传送到 `ramqueue` 中；但某些字符如控制字符或转义序列中的字符，需要经过特殊处理；当一个字符的屏幕位置超出了屏幕的实际宽度，或者 `ramqueue` 已满时，也需要进行特殊处理。在这些情况下就要调用 `out_char` (16 119 行) 来传送字符并执行各种相应的附加动作。例如，在定位屏幕的最后一行时，如果接收到一个换行符，那么 `scroll_screen` (16 205 行) 将被调用；而对于一个转义

序列，系统将调用 *parse_escape* 来处理这些字符。一般地，*out_char* 调用 *flush* (16 259 行) 把 *ramqueue* 中的内容复制到视频显示内存中，其中使用了一个名为 *mem_vid_copy* 的汇编语言例程。在最后一个输出字符被送入 *ramqueue* 中后，也要调用 *flush* 以确保所有要输出的字符都被显示出来。*flush* 的最终执行结果是使 6845 视频控制器在正确的位置上显示光标。

从逻辑上讲，从用户进程取出的字符可以在每次循环遍历中逐个写入视频 RAM。然而，在奔腾类处理器的保护内存环境中，把在 *ramqueue* 中等待的字符用 *mem_vid_copy* 成块地复制到视频 RAM 的效率要高得多。有趣的是，这项技术却是运行在没有内存保护的处理器上的早期 MINIX 3 版本中引入的。早期的 *mem_vid_copy* 必须考虑时序问题——老式的视频卡只允许在 CRT 电子束垂直回扫屏幕为空时写视频存储器，以避免屏幕上出现杂乱无章的“雪花”现象。由于性能牺牲过大，MINIX 3 现在已不再支持这些过时的设备。不过，成块复制 *ramqueue* 的思想却使现代的 MINIX 3 在其他方面得到了益处。

控制台可以使用的视频 RAM 区由 *console* 结构中的 *c_start* 和 *c_limit* 域限定。光标的当前位置存放在 *c_column* 域和 *c_row* 域中。坐标原点(0, 0)位于屏幕的左上角，硬件由这里开始填充屏幕。每次视频扫描从 *c_org* 中指定的地址开始，连续扫描 80×25 个字符 (4000 字节)。也就是说，6845 芯片从视频 RAM 的偏移 *c_org* 处取出一个字显示在屏幕左上角(0, 0)处，由属性字节控制字符的颜色和闪烁等，然后 6845 芯片取出下一个字显示在(1, 0)处。这个过程一直持续到到达(79, 0)处，然后从屏幕第二行的(0, 1)处重新开始。

计算机刚启动时，屏幕被清空，写入视频 RAM 的输出字符从视频 RAM 区的偏移 *c_start* 处开始存放；*c_org* 被赋予和 *c_start* 相同的初值。这样第一行就显示在屏幕的顶部。当第一行满或 *out_char* 检测到一个换行符，进而需要开始一个新行时，输出被写到偏移量等于 *c_start* 加 80 处。当 25 行都被写满时，就需要进行卷屏操作。有些程序也需要进行向下卷屏操作。例如一个文本编辑器在光标已位于最顶部并仍然向上运动时，就需要把整个文本块向下移动。

卷屏的方式有两种。在软件卷屏方式下，显示在(0, 0)位置上的字符永远被写入视频存储器的第一个位置，*c_start* 指向相对位置为 0 的字的位置。通过赋予 *c_org* 和 *c_start* 相同的值使得视频控制芯片从这个位置开始显示。当屏幕需要卷动时，视频 RAM 中相对位置 80 处的字，也就是第二行的第一个字符，被复制到相对位置 0 处；相对位置 81 处的字被复制到相对位置 1 处；等等。而扫描顺序没有改变，仍然把视频存储器位置 0 处的数据显示在屏幕位置(0, 0)处。这样屏幕上就产生了向上卷动一行的效果。这种方式下 CPU 的开销是需要移动 $80 \times 24 = 1920$ 个字。在硬件卷屏方式下，数据本身并不在存储器中移动，而是通过改变视频控制器的起始扫描位置使它从视频 RAM 中的不同位置开始显示，比如可以设置为从第 80 个字开始显示。设置的过程是把 *c_org* 的内容加上 80，把所得结果保存起来，并把这个值写入视频芯片中相应的寄存器。这种方式要求或者控制器有足够的智能在视频 RAM 区中卷动，在到达底端（由 *c_limit* 指定的地址）时能回卷到视频 RAM 的起始地址（由 *c_start* 指定）读取数据，或者拥有比存储正好一屏内容所必需的 80×2000 个字更多的视频 RAM。

老式的显示适配卡显示内存较少，但控制器能够回卷并进行硬件卷屏；较新的适配卡一般有比一屏文本的容量大得多的显示内存，但控制器不能回卷。这样，一块拥有 32 768 字节显示内存的适配卡就可以存储 204 行、每行 160 字节的数据，并且在遇到不能回卷的问题前可以执行 179 次硬件卷屏。但是最终仍然需要一次内存复制操作，把最后的 24 行数据移到视频内存的位置 0 处。不管采用哪种方法，都要向视频 RAM 中复制一个空行，以保证在屏幕最底部的新行是空行。

如果配置了虚拟控制台，那么一块适配卡上可用的存储器将被各个控制台平均分配。每个控制台设备使用的范围由各自的 *c_start* 域和 *c_limit* 域设定。配置虚拟控制台将对卷屏产生影响。对于

存储能力足以支持虚拟控制台的适配卡，尽管从概念上讲仍然可以使用硬件卷屏，但实际上使用得更多的是软件卷屏。每个控制台可用的存储器越少，软件卷屏就使用得越频繁。如果配置了最大数目的虚拟控制台，就达到了极限状态。这时所有的卷屏操作都由软件完成。

相对于视频RAM起始地址的光标位置可以由 *c_column* 和 *c_row* 计算出来。但单独为光标位置设置一个 *c_cur* 域会更快些。当需要显示一个字符时，就把它写入视频RAM的 *c_cur* 处，然后更新 *c_cur* 和 *c_column*。图 3.36 总结了 *console* 结构中影响当前位置和起始显示原点的域。

域	意义
<i>c_start</i>	该 <i>console</i> 的视频内存的起始位置
<i>c_limit</i>	该 <i>console</i> 的视频内存的界限
<i>c_column</i>	当前列 (0~79)，第0列在左边
<i>c_row</i>	当前行 (0~24)，第0行在顶部
<i>c_cur</i>	光标在视频RAM中的偏移
<i>c_org</i>	由6845基址寄存器指向的RAM中的位置

图 3.36 *console* 结构中与当前屏幕位置相关的域

对影响光标位置的字符（如换行符、退格符等）进行处理时需要调整 *c_column*, *c_row* 和 *c_cur* 的值。这些工作由 *flush* 在结束前执行 *set_6845* 调用完成。这个调用将设置视频控制器中相应的寄存器。

终端驱动程序支持转义序列，允许屏幕编辑器和其他交互式程序用灵活的方式来更新屏幕。所支持的序列是 ANSI 标准的一个子集，这个子集应该是完备的，使得为其他硬件和操作系统编写的程序能方便地移植到 MINIX 3 上。转义序列分为两类：一类不包含可变参数，另一类则可能包含参数。第一类中 MINIX 3 唯一支持的是 ESC M，它反向搜索屏幕，并且把光标上移一行；如果光标已经位于第一行，就把屏幕下卷一行。另一类转义序列可以包含一个或两个数值参数。这一类中所有的序列都以 ESC [开头，其中 “[” 是控制序列引入符 (control sequence introducer)。图 3.32 给出了 ANSI 标准定义的可由 MINIX 3 识别的转义序列表。

解析转义序列并不简单。在 MINIX 3 中有效的转义序列可以只有两个字符，如 ESC M；而可以接受两个两位数的数值参数的序列最多可以有 8 个字符长，如 ESC [20;60H，它使得光标移到第 20 行和第 60 列。在接受一个参数的序列中，参数可能被省略；而在接受两个参数的序列中，可能会省略一个参数，或者两个参数都省略掉。当有一个参数被省略或超出了有效范围时，就用默认值替换。默认值等于最低有效值。

以下是构造一个把光标移到屏幕左上角的转义序列的几种方法：

1. ESC [H 是可以接受的，因为在没有输入参数时，就用最低有效值代替。
2. ESC [1;1H 正确地把光标移动到第一行、第一列处。
3. ESC [l;H 和 ESC [;1H 都省略了一个参数，和第一个例子中一样，用默认值 1 替代。
4. ESC [0;0H 也具有同样的作用，因为每个参数都小于最小有效值，用最小值代替。

这些例子并不是推荐任意使用无效参数，而是想说明解析这些序列的代码并不简单。

MINIX 3 使用了一个有限状态自动机解析转义序列。*console* 结构中的 *c_esc_state* 变量在正常状态下值为 0。当 *out_char* 检测到一个 ESC 字符时，*c_esc_state* 的值被置为 1，后面的字符由 *parse_escape* (16 293 行) 进行处理。如果后续字符是控制序列引入符，就进入状态 2；否则认为序列已结束，调用 *do_escape* (16 352 行)。在状态 2，只要后续字符是数值字符，就把参数计算过程

中上一次的参数值（初始值为0）乘以10，加上当前字符的数值作为当前参数值。参数值保存在一个数组中，如果检测到一个分号，处理过程就移到数组的下一个单元中（MINIX 3中的数组只有两个元素，但原理是一样的）。如果遇到一个不是分号的非数值字符，就判断序列已结束，接着同样调用 *do_escape*。在 *do_escape* 的入口处，根据当前字符决定采取什么动作和如何解释参数，是使用默认值还是使用字符流中输入的参数。图 3.44 详细地绘出了整个过程。

可加载的键位映射表

IBM PC 键盘并不直接产生 ASCII 码。从标准 PC 键盘的左上角开始，每个键依次用一个数来标识——“ESC”键为1，“1”键为2，依次类推。这样每个键都被赋予一个数，包括转换键，如左 SHIFT 键为42，右 SHIFT 键为54。当按下一个键时，MINIX 3 把接收到的对应的数作为扫描码。当松开一个键时也产生一个扫描码，但松开时产生的扫描码最高位被置位（即该键对应的数加上128）。这样就可以分辨出一个键是被按下还是被释放。通过检测转换键是否被按下并且仍未松开，可以产生大量的组合键。在一般的场合，两键的组合对于双手打字来说最容易操作，如 SHIFT-A 或 CTRL-D。但在某些特殊场合也使用三键组合（或更多）。比如 CTRL-SHIFT-A 或众所周知的 PC 用户用来复位并重新启动系统的 CTRL-ALT-DEL 组合键。

十分复杂的PC键盘为用户提供了很大的灵活性。一个标准键盘定义了47个普通字符键（26个字母，10个数字，11个标点）。如果愿意使用类似 CTRL-ALT-SHIFT 这样的三键组合，则可以支持有 376 (8×47) 个成员的字符集。但这是在不加任何限制的情况下，现在假定不区分左右转换键，也不使用数字键盘和功能键。实际上，并不只限于使用 CTRL, ALT 和 SHIFT 作为转换键，如果需要编写一个支持这种系统的驱动程序，那么可以从普通键集合中去掉一些键，把它们作为转换键。

支持这种键盘的操作系统根据按下的键和有效的转换键，用键位映射表（keymap）决定把什么字符码传给程序。MINIX 3 的键位映射表可以视为一个 128 行、6 列的数组。行代表可能的扫描码值（选择这个行数是为了容纳日文键盘，美国和欧洲键盘没有这么多键），各列分别代表无转换键、SHIFT 键、CTRL 键、左 ALT 键、右 ALT 键以及任一个 ALT 与 SHIFT 键的组合。用这种方式可以产生 720 个 $[(128 - 8) \times 6]$ 字符编码，能定义一个完备的键盘。这就要求表中的每个表项都是 16 位长。对于美国键盘，ALT 列和 ALT2 列是一样的。在其他语言的键盘上，ALT2 也被称为 ALTGR，许多键位映射表使用这个键作为转换键支持三符号的键。

编译时在 MINIX 3 内核中包含了一个标准键位映射表（由 *keyboard.c* 中的行

```
#include keymaps/us-std.src
```

决定），但也可以用

```
ioctl(0, KIOCSMAP, keymap)
```

把其他不同的键位映射表装入到内核地址 *keymap* 处。一个完整的键位映射表占用 1536 个字节 ($128 \times 6 \times 2$)。其他附加的映射表以压缩格式存储。一个程序可以调用 *genmap* 来生成一张新的压缩的映射表。在编译时，*genmap* 为某个特定的映射表包含 *keymap.src* 代码，这样这张映射表就被编译到 *genmap* 中。正常情况下，*genmap* 在编译之后立即执行，把生成的映射表以压缩格式输出到一个文件中；然后删除二进制的 *genmap*。命令 *loadkeys* 读入一个压缩的映射表，把它在内存中展开，然后调用 *ioctl* 把映射表传送到内核存储区。MINIX 3 可以在启动时自动执行 *loadkeys*，也可以由用户在任何时候调用它。

映射表的源代码中定义了一个初始化过的大型数组。为了节省篇幅，附录B没有打印一个用源码形式列出的映射表文件。图3.37用表格形式给出了*src/kernel/keymaps/us-std.src*中若干行的内容，说明了映射表的若干方面。IBM PC 键盘上没有键产生值为0的扫描码。扫描码为1，即ESC键对应的表项说明按下SHIFT或者CTRL键时返回值不变，但ALT和ESC同时按下时将返回一个不同的值。编译到表中各行里的值是由*include/minix/keymap.h*中定义的宏决定的：

```
#define C(c) ((c) & 0x1F)      /* 到控制编码的映射 */
#define A(c) ((c) | 0x80)      /* 设置8位(ALT) */
#define CA(c) A(C(c))        /* CTRL-ALT */
#define L(c) ((c) | HASCAPS)  /* 增加“大写锁定已生效”属性 */


```

前三个宏对被引用字符扫描码中的位进行操作，把生成的所需扫描码返回给应用程序。最后一个宏在16位值的高字节中设置HASCAPS位。这个标志位指示需要检测大写锁定变量的状态，并且扫描码在返回前可能需要修改。在图中，扫描码为2, 13和16的表项说明了典型的数字、标点和字母键是如何被处理的。在此可以观察到扫描码28的一个不同特征——ENTER键一般产生扫描码CR(0x0D)，这里用C('M')表示。由于在UNIX文件中换行符的编码是LF(0x0A)，并且有时需要直接输入换行符，所以这个映射表提供了一个CTRL-ENTER组合来产生这个编码，即C('J')。

扫描码	字符	常规	SHIFT	ALT1	ALT2	ALT+SHIFT	CTRL
00	none	0	0	0	0	0	0
01	ESC	C('T')	C('T')	CA('T')	CA('T')	CA('T')	C('T')
02	'1'	'1'	'1'	A('1')	A('1')	A('!')	C('A')
13	'='	'=	'+'	A('=')	A('=')	A('+')	C('@')
16	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
28	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
29	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
59	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

图3.37 一个建立映射表源文件的若干表项

扫描码29对应一个转换键码，并且无论是否还按下了其他什么键，它都应该能被识别出来，因此总是返回CTRL。按下功能键不返回普通的ASCII码值。扫描码59对应的行中用符号（在*include/minix/keymap.h*中定义）给出了F1键和其他转换键组合时的返回值。这些值是F1: 0x0110, SF1: 0x1010, AF1: 0x0810, ASF1: 0x0C10和CF1: 0x0210。图中的最后一个表项对应扫描码127，在映射表数组末尾附近的表项中十分典型。绝大多数在欧洲和美洲使用的键盘没有那么多的键对应所有可能的扫描码，表中的这些表项都用0填充。

可装载字体

早期的PC只能用存储在ROM中的字符点阵在屏幕上显示字符。现代PC系统中使用的显示器的视频显示卡上则提供了可以加载定制的字符发生器（character generator）的RAM。MINIX 3 提供了一个*ioctl*操作来支持定制的可装入字符发生器：

```
ioctl(0, TIOCSFON, font)
```

MINIX 3 支持 80 列 × 25 行的显示模式，字体文件包含 4096 个字节。每个字节代表一行 8 个像素，如果某一位为 1，对应的像素就被点亮，映射一个字符需要 16 个这样的行，即 16 个字节。不过视频适配卡中使用了 32 个字节来映射一个字符，用来在 MINIX 3 目前还不支持的模式中提供更高的

分辨率。命令 *loadfont* 把这些字体文件转换到由 *ioctl* 调用引用的 8192 字节的 *font* 结构中，同时用 *ioctl* 调用来装载这些字体。和键位映射表一样，字体可以在启动时加载，也可以在正常操作中的任何时候加载。不过，任何视频适配器都有一个存储在 ROM 中的标准字体作为可使用的默认字体。MINIX 3 并不需要把一种字体编译到它本身里，内核中唯一需要的对字体的支持是进行 *TIOCSFON* *ioctl* 操作的代码。

3.8.4 设备无关终端驱动程序的实现

在这一节中，我们将要仔细地研究终端驱动程序的源代码。在研究块设备时，我们已经看到支持几个不同设备的多个任务可以共享一套共同的基本软件。终端设备的情况也是类似的，不同的是，这里一个终端驱动程序要支持几个不同类型的终端设备。这里我们从设备无关代码开始。后面几节将研究键盘和内存映射控制台显示的设备相关代码。

终端驱动程序数据结构

文件 *tty.h* 中包含了在实现终端驱动程序的 C 文件中使用的定义。由于这个驱动程序支持多种不同的设备，所以必须使用次设备号来区分在一个特定的调用中被支持的是哪个设备，这些定义在 13 405 到 13 409 行。

tty.h 中定义的 *O_NOCTTY* 和 *O_NONBLOCK* 标记（它们是 *open* 调用的可选参数）在文件 *include/fcntl.h* 中已经定义过了。在这里再重写一次是为了避免需要再包含一个文件。*devfun_t* 和 *devfunarg_t* 类型（13 423 行）用来定义指向函数的指针，以提供在磁盘驱动程序代码的主循环中看到的相类似的间接调用机制。

这个文件中声明的许多变量都具有前缀 *tty_*。*tty.h* 中最重要的是 *tty* 结构的定义（13 426 到 13 488 行和 13 424 行）。每个终端设备都有一个这样的结构与之对应（把控制台显示和键盘共同作为一个单一的终端）。*tty* 结构中的第一个变量 *tty_events* 被用做一个标志，它在某个中断引起变化需要终端任务来处理设备时被置位。

tty 结构中的其余部分分别按照处理输入、输出、状态和关于未完成操作的信息等功能组织在一起。在输入部分，*tty_inhead* 和 *tty_intail* 定义了缓冲接收到的字符队列。*tty_incount* 对队列中的字符个数计数，*tty_eotct* 对字符的行数计数，这将在下面解释。除了用来建立间接调用指针的终端初始化例程以外，所有针对特定设备的调用都是间接调用。*tty_devread* 域和 *tty_icancel* 域中保存了指向执行读和输入取消操作的设备指定代码的指针。*tty_min* 用来和 *tty_eotct* 进行比较，当后者与前者一致时，就认为一个读操作结束。在规范输入模式中，*tty_min* 被置为 1，用 *tty_eotct* 对输入的行数计数。在非规范输入模式中，用 *tty_eotct* 对字符计数，*tty_min* 则由 *termios* 结构中的 *MIN* 域决定。这样，根据所处的模式，通过比较两个变量可以判断出一行是否已准备好，或者何时达到最小字符数。*tty_tmr* 是针对这个 *tty* 的一个计时器，被 *termios* 中的 *TIME* 域使用。

由于输出排队是由设备相关代码处理的，所以 *tty* 的输出部分没有变量说明，全部由指针组成，这些指针指向写、回显、送中断信号和取消输出等设备相关函数。在状态部分，标志 *tty_reprint*，*tty escaped* 和 *tty_inhibited* 说明最后一个字符具有特殊含义。例如，当遇到一个 CTRL-V (LNEXT) 字符时，*tty escaped* 被置为 1，指示下一个字符的任何特殊含义应被忽略。

结构中的下一部分保存了 *DEV_READ*, *DEV_WRITE* 和 *DEV_IOCTL* 操作的进度数据。每个操作中包含了两个进程。管理系统调用的服务器（正常时为 FS）在 *tty_incallee* (13 458 行) 中指定。服务器为另一个需要进行 I/O 操作的进程调用 *tty* 任务，这个客户进程在 *tty_inproc* (13 459 行) 中指定。如图 3.33 所示，在执行一个 *read* 时，字符直接由终端任务传送到初始调用者内存空间中的缓

冲区。缓冲区由 *tty_inproc* 和 *tty_in_vir* 定位。接下来的两个变量 *tty_inleft* 和 *tty_incum* 对仍需传送和已传送的字符计数。*write* 系统调用也需要类似的变量集。对于 *ioctl* 操作可能有在请求进程和任务之间的直接数据传送，因此需要定义一个虚拟地址，但不需要标志操作进度的变量。一个 *ioctl* 请求可能被延迟，比如直到当前输出完成；但在时间合适时，一次操作即可完成请求。

最后，*tty* 结构中还包含了一些不在其他集合中的变量，包括指向在设备级处理 *DEV_IOCTL* 和 *DEV_CLOSE* 的函数的指针，一个 POSIX 格式的 *termios* 结构，一个支持面向窗口屏幕显示的 *winsize* 结构。结构的最后部分提供了存储输入队列本身的数组 *tty_inbuf*。注意，这是一个 *u16_t* 数组，而不是 8 位 *char* 型字符数组。虽然应用程序和设备使用 8 位编码的字符，C 语言却需要输入函数 *getchar* 能使用更大的数据类型，以便能返回在所有 256 个可能值之外的符号值 *EOF*。

tty_table，即 *tty* 结构的数组，是用 *extern* 宏说明的（13 491 行）。由 *include/minix/config.h* 中的定义 *NR_CONS*, *NR_RS_LINES* 和 *NR_PTYS*，每个启用的终端都有一个元素与之对应。在本书讨论的配置中，有两个控制台被启用，但 MINIX 3 可以被重新编译，以增加额外的虚拟控制台、1 到 2 条串行线以及最多 64 个伪终端。

tty.h 中还有一个 *extern* 定义。*tty_timers*（13 516 行）是由定时器使用的保存 *timer_t* 域链表头的指针。许多文件包含了 *tty.h* 头文件，*tty_table* 和 *tty_timers* 的存储空间在编译 *tty.c* 时分配。

在 *tty.h* 的 13 520 到 13 521 行定义了两个宏 *buflen* 和 *bufend*。终端驱动程序代码经常使用这两个宏把数据复制到缓冲区，或者从缓冲区复制出数据。

设备无关终端驱动程序

主要的终端驱动程序和设备无关的支持函数都在 *tty.c* 中定义。接下来的是一些宏定义。如果一个终端没有初始化，那么指向该设备的设备相关函数的指针将被 C 编译器置为 0。这样就可以定义一个 *tty_active* 宏（13 687 行），在发现一个空指针时返回 *FALSE*。当然，如果设备的初始化代码的一部分工作是初始化指针，使间接存取成为可能，那么它本身是不能被间接存取的。13 690 到 13 696 行定义的条件宏使得对 RS-232 或伪终端设备的初始化调用在没有配置这些设备时都同样地调用一个空函数。在这一节中，*do_pty* 也可能类似地被禁止。这样，如果不需要某些设备，就可以把这些设备对应的代码完全忽略掉。

由于每个终端可配置的参数很多，并且在一个网络系统中可能有许多终端，所以 13 720 到 13 727 行说明了一个 *termios_defaults* 结构，并用默认值（所有这些均在 *include/termios.h* 中定义）初始化。当需要初始化或重新初始化一个终端时，就向该终端的 *tty_table* 中复制一个这样的结构。图 3.29 中给出了特殊字符的默认值。图 3.38 给出了各种标志的默认值。接下来的行类似地声明了 *winsize_defaults* 结构。它应由 C 编译器全部初始化为 0。这是一个合适的默认动作，因为它意味着“窗口大小未知，使用 */etc/termcap*”。

域	默认值
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTEN ICANON ECHO ECHOE

图 3.38 默认的 *termios* 标志值

在可执行代码开始之前最后定义的部分是前面在 *tty.h* 中声明为 *extern* 的全局变量的 PUBLIC 声明（13 731 到 13 735 行）。

终端驱动程序的入口点为 `tty_task` (13 740行)。在进入主循环之前，对每个已配置的终端调用一次 `tty_init` (13 752行)。初始化键盘和终端需要的关于主机的信息通过 `sys_getmachine` 内核调用来获得，然后初始化键盘硬件。对应的例程调用是 `kb_init_once`。这样命名是为了将它与另一个随后会调用的用来对每一个虚拟控制台进行部分初始化的初始化例程区分。最后一个单独的0被打印以测试输出系统，同时将直到第一次使用还没有初始化的任何东西踢出去。虽然从源码中看到的是对 `printf` 的调用，但是它不同于用户程序使用的 `printf`，它是一个使用了控制台驱动程序中局部函数 `putk` 的特殊版本。

13 764 到 13 876 行的主循环原理上与其他任务的主循环是一样的——它接收一条消息，根据消息类型执行一条 `switch` 语句调用适当的函数，并产生一条返回消息。但是这里有一点复杂。首先，最后的中断可能会导致额外的字符可能被读取到，或者将要向一个输出设备写的字符可能已经准备好了。在试图接收一条消息之前，主循环总是检查每个终端的 `tp->tty_events` 标志。如果必要，就调用 `handle_events` 处理未完成的事务。只有在没有需要立刻处理的事件时，才产生一个调用来接收消息。

在 `tty.c` 文件开头的注释中用来显示消息类型的图表展示了最经常使用的类型。一些需要从终端驱动程序得到特殊服务的消息类型在这里没有显示。这些不是针对某一个特殊的设备的。`tty_task` 主循环在检查设备相关消息之前检查这些以同时处理它们。首先检查 `SYN_ALARM` 消息，如果存在这个消息则调用 `expire_timers` 来引发一个看门狗例程，接下来就是一个 `continue` 语句。事实上所有下面将会讨论的其他几种情况都是在 `continue` 语句后面。随后我们马上会讨论这些内容。

接下来检测的消息类型是 `HARD_INT`。这大多数是由本地键盘某个键被按下或者被释放引起的。这也意味着如果串口在配置中启用（在这里研究的情况没有启用），那么数据已经从串口获取，但是这里把条件码放在文件里来说明串口输入是怎样被处理的。消息中的一位用来确定中断源。

接下来检查 `SYS_SIG`。系统进程（驱动程序和服务器）将会因为等待消息而阻塞。普通信号仅由激活的进程来接收，所以标准的 UNIX 发信号方法对系统进程不起作用。`SYS_SIG` 消息用于向一个系统进程发信号。一个向终端驱动程序发出的信号可能意味着内核正在关闭 (`SIGKSTOP`)，终端驱动程序正在被关闭 (`SIGTERM`)，或者内核需要向终端打印消息 (`SIGKMESS`)，对于这些情况会分别调用合适的程序。

最后一组与设备无关的消息是 `PANIC_DUMPS`, `DIAGNOSTICS` 和 `FKEY_CONTROL`。我们将在这些消息的对应功能被调用时，详细地讨论这些消息。

现在该讨论 `continue` 语句了：在 C 语言中，`continue` 语句短路一个循环，使控制返回到循环的顶部。所以，如果迄今为止所提到的各种消息类型被检测到，那么它们一旦被服务，控制就会返回到主循环的顶部，在 13 764 行，对事件的检查是重复的，而 `receive` 会被再次调用来等待一个新消息。在输入的情况下较特殊，故尽可能快地做好响应的准备是很重要的。而且，如果在循环的第一部分中任何一个消息类型测试成功，那么在第一个 `switch` 以后就没有必要对任何一个消息类型再进行测试。

上面讨论了终端驱动程序必须处理的复杂性。第二个复杂性在于终端驱动程序必须为一些设备服务。如果中断不是来自硬件，那么消息中的 `TTY_LINE` 域可以用来确定是哪个设备应该响应这个消息。次设备号通过一系列的比较被解码，通过这种方法，`tp` 指向 `tty_table` 中正确的表项 (13 834 到 13 847 行)。如果该设备是一个伪终端，那么就调用 `do_pty` (在 `pty.c` 中) 并且重新开始主循环。在这种情况下，`do_pty` 生成应答消息。当然，如果没有启用伪终端，那么对 `do_pty` 的调用将使用前面定义的哑宏。人们希望永远不要发生试图访问不存在设备的情况，但加上一个检查过程比确保

系统中其他任何地方都没错要容易得多。在设备不存在或没有配置时，就产生一条带有 *ENXIO* 错误消息的应答消息，并且控制返回到循环的顶部。

终端驱动程序的其余部分和在其他驱动序的主循环部分看到的相类似。根据消息的类型进行一个 **switch** 选择（13 862 到 13 875 行），调用请求对应的函数，如 *do_read*, *do_write* 等。每种情况被调用的函数产生应答消息，而不是把构造消息所需要的信息传回主循环。只有在没有收到有效的消息类型时，主循环的结束部分才产生一条 *EINVAL* 错误信息。因为应答消息是在终端任务内部的许多不同地方发出的，所以可以调用一个公共的例程 *tty_reply* 来处理构造应答消息的细节。

如果 *tty_task* 收到的消息是一个有效的消息类型而不是一个中断的结果，也不是来自一个伪终端，那么主循环最后的 **switch** 将把它分发给函数 *do_read*, *do_write*, *do_ioctl*, *do_open*, *do_close*, *do_select* 和 *do_cancel* 中的一个。每个函数调用的参数是指向 *tty* 结构的指针 *tp* 和消息的地址。在分别考察这些函数之前，要先进行一些一般性的讨论。由于 *tty_task* 可能要为多个终端设备服务，所以这些函数必须很快地返回，以便主循环能继续响应其他请求。

然而，*do_read*, *do_write* 和 *do_ioctl* 可能无法立即处理完所有的请求任务。为了允许 FS 为其他调用服务，需要立即应答。如果请求不能立即完成，那么就在应答消息的状态域中返回一个 *SUSPEND* 码。这对应于图 3.33 中的消息 3，它挂起发起调用的进程，同时释放 FS。图中消息 10 和 11 在操作结束后发送。如果完全可以满足请求，或者发生了一个错误，那么就在返回给 FS 的消息的状态域中返回传送的字节数或错误码。在这种情况下，FS 立即向进行原始调用的进程发送一条消息，唤醒该进程。

读终端与读磁盘设备是完全不同的。磁盘驱动程序向磁盘硬件发出一条指令，数据最终就会被返回，除非发生机械或电子故障。计算机可以在屏幕上显示一条提示信息，但却无法强迫一个人坐在键盘前开始输入。因此，计算机根本无法保证有人坐在那里。为了能够尽快得到返回，在有输入到达时，*do_read*（13 953 行）一开始就会保存一些信息，使得请求可以在稍后完成。首先要进行一些错误检查。当设备仍然在等待输入以满足前一个请求时，或消息中的参数无效时（13 964 到 13 972 行），将发生一个错误。如果通过了检查，13 975 到 13 979 行就把关于请求的信息复制到设备的 *tp->tty_table* 表项的适当域中。最后一步把 *tp->tty_inleft* 设置为所需的字符数非常重要。这个变量用来决定读请求何时得到满足。在规范模式，*tp->tty_inleft* 每返回一个字符则减 1，接收到行结束符时，这个值突然减小到 0。在非规范模式，处理有所不同，但在任何情况下，只要调用被满足，*tp->tty_inleft* 就被置为 0，而不论是由于超时还是接收到请求的最小字符数。当 *tp->tty_inleft* 达到 0 时，送出应答消息。正如我们将要看到的那样，应答消息可以在好几个地方产生。有时还需要检查一个读进程是否仍在等待一个应答，非 0 时的 *tp->tty_inleft* 可以用于这个目的。

在规范模式，终端设备一直在等待输入，直到已接收到调用中请求的字符数，或达到了一行或一个文件的末尾。13 981 行通过检查 *termios* 结构中的 *ICANON* 位判断终端是否处于规范模式。如果这一位没有置位，那么就检查 *termios* 的 *MIN* 和 *TIME* 值以决定采取什么动作。

在图 3.31 中，我们已可以看到 *MIN* 和 *TIME* 是怎样相互作用的，以提供一个读调用可能采取的不同动作。*TIME* 在 13 983 行中检测。0 值对应于图 3.31 中的左边一列，在这种情况下不需要进一步的检测。如果 *TIME* 不为 0，那么就检测 *MIN*。如果为 0，那么就调用 *settimer* 启动定时器。即使没有收到一个字节，在延迟一段时间后定时器也将结束 *DEV_READ* 请求。这里 *tp->tty_min* 被置为 1，使得在超时前如果接收到一个或几个字节就立即终止调用。这里还没有对可能的输入进行检查，所以可能有不止一个字符在等待满足请求。在这种情况下，发现输入后将立即返回 *read* 调用中指定个数的字符。如果 *TIME* 和 *MIN* 都不为 0，那么定时器将有不同的含义。在这种情况下，定时器用来作为字符间的定时器。它只在收到第一个字符后启动，并在接收到每个后续字符之后重新

启动。*tp->tty_eotct* 在非规范模式下对字符计数，如果在 13 993 行中为 0，那么就还没有接收到字符，并且字节间定时器被禁用。

在任何情况下，14 001 行都调用 *in_transfer* 把所有已在输入队列中的字节直接传送到读进程中。接着进行一个 *handle_events* 调用，这个调用可能向输入队列中送入更多的数据，然后再次调用 *in_transfer*。这里需要解释一下这个显得重复的调用。虽然到目前为止，我们所有的讨论都是从键盘输入的角度来考虑的，但 *do_read* 是在代码的设备无关部分，它也要为通过串行线连接的远程终端的输入服务。上一次的输入可能已经填满了 RS-232 输入缓冲区，从而导致输入被禁止。对 *in_transfer* 的第一次调用没有重新启动输入流，但对 *handle_events* 的调用可以起到这个作用。事实上，对 *in_transfer* 的再次调用仅是一个意外收获，重要的是确保远程终端被允许再次发送。这些调用都可能满足请求并向 FS 发送应答消息。*tp->tty_inleft* 被用做一个标志来判断应答是否已发送，如果在 14 004 行中它仍为 0，那么 *do_read* 就自己产生并发送应答消息。这些由 14 013 到 14 021 行完成（这里假定 *select* 系统调用未被使用，并因此未调用 14 006 行的 *select_retry*）。

如果原始调用者指定了一个不可阻塞读，那么 FS 将被通知向原始调用者发送一个 *EAGAIN* 错误码。如果调用是一个普通的可阻塞读，那么 FS 将收到一个 *SUSPEND* 码，释放 FS 但通知它保持原始调用者阻塞。在这种情况下，终端的 *tp->tty_inrepcode* 域被置为 *REVIVE*。如果 *read* 稍后被满足，那么这个值将放在给 FS 的应答消息中，以指示原始调用者被置为需要唤醒的睡眠状态。

do_write (14 029 行) 和 *do_read* 很相似，但更简单一些，因为在处理一个 *write* 系统调用时需要考虑的情况较少。用与 *do_read* 中相类似的检查来检测上一次写是否未在进行中以及参数是否有效，然后把请求的参数复制到 *tty* 结构中。接着调用 *handle_events*，并检查 *tp->tty_outleft* 以判断是否完成工作 (14 058 到 14 060 行)。如果是，那么 *handle_events* 已经发送了一条应答消息，不需要再做什么。如果没有，那么就产生一条应答消息，消息参数依赖于原始的 *write* 调用是否以不可阻塞模式进行。

下一个函数 *do_ioctl* (14 079 行) 很长，但不难理解。*do_ioctl* 的函数体是两条 *switch* 语句。第一条语句确定由请求消息中的指针指向的参数的大小 (14 094 到 14 125 行)。如果大小不为 0，就检验参数的有效性。这里不能检测内容，但可以检查指定地址处开始的所需大小的结构是否能装入它被指定放置的段。函数的余下部分是根据 *ioctl* 操作请求类型进行选择的 *switch* 语句 (14 128 到 14 225 行)。

遗憾的是，用 *ioctl* 调用支持 POSIX 要求的操作意味着必须为 *ioctl* 操作构造 POSIX 要求的名字，而不是复制。图 3.39 给出了 POSIX 要求的名字与用 MINIX 3 的 *ioctl* 调用的名字之间的关系。一个 *TCGETS* 操作为用户的 *tcgetattr* 调用服务，并且只是简单地返回终端设备的 *tp->tty_termios* 结构的一份副本。下面的 4 个请求类型共享代码。*TCSETSW*, *TCSETSF* 和 *TCSETS* 请求类型对应于用户调用的 POSIX 定义的函数 *tcsetattr*，并且基本动作都是把新的 *termios* 结构复制到终端的 *tty* 结构中。对于 *TCSETS* 调用，复制动作立即执行；对于 *TCSETSW* 和 *TCSETSF* 调用，则可能在输出结束后执行。复制通过一个 *sys_vircopy* 内核调用从用户取得数据，接下来在 14 153 到 14 156 行调用 *setattr*。如果调用 *tcsetattr* 时使用了一个修饰符请求把动作延迟到当前输出完毕后执行，并且如果在 14 139 行中对 *tp->tty_outleft* 的检测表明输出没有结束，那么就把请求的参数放在终端的 *tty* 结构中供以后处理。*tcdrain* 被翻译为一个 *TCDRAIN* 类型的 *ioctl* 调用，可把一个程序挂起，直到输出结束。如果输出已经结束，则它不再执行什么动作。如果没有结束，还必须把信息保存在 *tty* 结构中。

POSIX函数	POSIX操作	IOCTL类型	IOCTL参数
tcdrain	(无)	TCDRAIN	(无)
tcflow	TCOFF	TCFLOW	int=TCOFF
tcflow	TCON	TCFLOW	int=TCON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(无)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(无)	TCSBRK	int=duration

图 3.39 POSIX 调用和 IOCTL 操作

POSIX 的 *tcflush* 函数根据它的参数丢弃未读入的输入和（或）未发送的输出。该函数被直接翻译为 *ioctl* 调用，由一个为所有终端服务的函数 *tty_icancel* 和（或）由 *tp->tty_ocancel* 指向的设备相关函数的调用组成（14 159 到 14 167 行）。*tcflow* 也被类似地直接翻译为一个 *ioctl* 调用。为了挂起或重启输出，它向 *tp->tty_inhibited* 中写入一个 *TRUE* 或 *FALSE*，然后设定 *tp->tty_events* 标志位。为了挂起或重启输入，它使用由 *tp->tty_echo*（14 181 到 14 186 行）指向的设备指定回显例程向远程终端发送合适的 *STOP*（正常为 *CTRL-S*）或 *START*（*CTRL-Q*）码。

余下的由 *do_ioctl* 处理的大部分操作都由一行代码调用一个合适的函数处理。对于 *KIOCSMAP*（装入键位映射表）和 *TIOCSFON*（装入字体）操作，要进行检验以保证该设备确实为控制台，因为这些操作不应用于其他终端。如果使用了虚拟终端，那么所有的控制台都将使用相同的键位映射表和字体，硬件不允许用任何其他方便的方法。窗口大小操作在用户进程和终端任务之间复制 *winsize* 结构。不过要注意 *TIOCSWINSZ* 操作对应的代码下面的注释。当一个进程改变了它的窗口大小时，在某些版本的 UNIX 下，内核应该向进程组发送一个 *SIGWINCH* 信号。POSIX 标准中并不要求这个信号，在 MINIX 3 中也没有实现。但是任何想要使用这些结构的人都应该考虑在这里添加代码以初始化这个信号。

do_ioctl 的最后两种情况支持 POSIX 要求的 *tcgetpgrp* 和 *tcsetpgrp* 函数。这两种情况没有对应的动作，只是永远返回一个错误。这并没有错误。这些函数支持作业控制（job control），能够从键盘上挂起和重启一个进程。POSIX 不要求作业控制，在 MINIX 3 中也不支持。不过，即使不支持作业控制，POSIX 也需要这些函数来保证程序的可移植性。

do_open（14 234 行）执行的是一个简单的基本动作——它增加设备的 *tp->tty_opencnt* 变量的值，以便检验设备是否是打开的。不过，在这之前首先要进行一些检测。对于普通终端，POSIX 指定第一个打开终端的进程为会话主导进程（session leader）。当一个会话主导进程终止时，将收回该组中其他进程对该终端的访问权。监控程序需要能够输出出错信息，并且如果错误信息输出未被重定向到一个文件中，那么它应当被送到一个不能被关闭的显示器上。

为此，MINIX 3 中存在一个称为 */dev/log* 的设备。在物理上它与 *dev/console* 是相同的设备，但它由一个独立的次设备号所标识，而且处理方式也不同。它是一个只写设备，如果 *do_open* 试图以读方式打开它，则返回一个 *EACCESS* 错误（14 246 行）。*do_open* 执行的另一个检测是测试

O_NOCTTY 标志。如果该标志未被置位且设备不是 */dev/log*, 则该终端成为一个进程组的控制终端。这是通过把调用方的进程号放入 *tty_table* 表项的 *tp->tty_pgrp* 域来完成的。然后, 增加 *tp->tty_opencnt* 变量的值并发送应答消息。

一个终端设备可能被打开不止一次, 而下一个函数 *do_close* (14 260行) 除了减少 *tp->tty_opencnt* 变量的值外不执行其他动作。14 266行的检测防止了该函数试图关闭设备 */dev/log*。如果这个操作是最后一次关闭操作, 那么就调用 *tp->tty_icancel* 取消输入。由 *tp->tty_ocancel* 和 *tp->tty_close* 指向的设备相关例程也被调用。然后 *tty* 结构中的各个域被置回它们的默认值, 并且发送应答消息。

最后一类消息类型由 *do_cancel* 处理 (14 281行)。当一个试图读或写时被阻塞的进程接收到一个信号时, 将引起这个动作。这时需要检查三种状态:

1. 终止时进程可能在读。
2. 终止时进程可能在写。
3. 进程可能被 *tcdrain* 挂起直到它的输出结束。

对每种情况进行检测, 并根据需要调用通用的 *tp->tty_icancel* 或由 *tp->tty_ocancel* 指向的设备相关例程。在最后一种情况下, 唯一要做的事情就是重置 *tp->tty_ioreq* 标志来指示 *ioctl* 操作现在已经完成。最后, *tp->tty_events* 标志被置位并发送一条应答消息。

终端驱动程序支持代码

现在我们已经研究了 *tty_task* 中主循环调用的顶层函数, 下面将从 *handle_events* 开始 (14 358行), 以学习支持它们的代码。前面提到, 每执行一次终端驱动程序的主循环, 都要检查每个终端设备的 *tp->tty_events* 标志, 如果它表明需要处理某个特定设备的事件, 就调用 *handle_events*。*do_read* 和 *do_write* 也调用 *handle_events*。例程 *handle_events* 必须工作得很快。它重置 *tp->tty_events* 标志, 然后调用由指针 *tp->tty_devread* 和 *tp->tty_devwrite* (14 382 到 14 385行) 指向的设备相关例程进行读和写。

这些函数都是无条件地调用的, 因为这里无法检验一次读或写是否会引起标志的置位——这里有一个设计上的选择, 即对每个设备检查两个标志的代价要高于每次激活一个设备时做两次调用的代价。另外, 在大部分时间里从终端接收到的字符是需要回显的, 所以两个调用都是需要的。如同在讨论处理 *do_ioctl* 调用的 *tcsetattr* 时所提到的那样, POSIX 可能会推迟对设备的控制操作直到当前的输出完成。所以在调用设备相关函数 *tty_devwrite* 后最好立即处理 *ioctl* 操作。这些在 14 388行中完成, 如果有一个等待的控制请求, 就调用 *dev_ioctl*。

由于 *tp->tty_events* 标志由中断置位, 并且字符从高速设备中迅速到达, 因此可能出现在对设备相关的读和写例程以及 *dev_ioctl* 的调用结束时, 另一个中断又使得标志置位的情况。因此, 要将输入信息由缓存中断程序放置的初始位置移出, 并且赋予较高的优先权。这样, 只要在循环结束处 (14 389行) 检测到 *tp->tty_events* 标志置位, *handle_events* 就重复地调用设备相关例程。当输入流停止时 (也可以是输出, 但输入更可能这样重复地请求), 就调用 *in_transfer* 把字符从输入队列中传送到调用一个读操作进程的内部缓冲区中。无论是传送了请求的最大字符数还是到达了一行的末尾 (在规范模式中), 如果传送结果结束了请求, 那么 *in_transfer* 本身发送一条应答消息。如果这样, 那么在返回到 *handle_events* 时 *tp->tty_left* 的值为 0。这里还会做进一步检查, 如果传送的字符数达到请求的最小数目, 那么就发送一条应答信息。对 *tp->tty_inleft* 的检测防止了重复消息的发送。

下一步将研究 *in_transfer* (14 416行), 这个函数把数据从驱动程序存储空间中的输入队列移动到请求输入的用户进程的缓冲区中。然而, 直接块复制是不行的。输入队列是一个环形缓冲区,

并且需要检查字符以判断是否已达到文件末尾，或者检查是否处于规范模式下，传送只持续到一行结束。另外，输入队列中各单元的大小为16位，而接收缓冲区是一个8位字符的数组。这样就需要使用一个中间局部缓冲区。字符在放入局部缓冲区时被逐个检查，当缓冲区被填满或者输入队列被移空时，就调用 `sys_vircopy` 把局部缓冲区中的内容移到接收进程的缓冲区中（14 432到14 459行）。

tty 结构中有三个变量 `tp->tty_inleft`, `tp->tty_eotct` 和 `tp->tty_min`, 用于确定 `in_transfer` 是否需要完成一些任务，前两个变量用于控制主循环。正如前面所提到的，`tp->tty_inleft`一开始被设置为一个 `read` 调用所请求的字符数。正常情况下，每传送一个字符，它就减1。但是当到输入的结束时，它会突然减少到0。不管 `tp->tty_inleft` 何时变为0，都会向读进程发送一条应答消息，因此该变量也可以作为一个标志来指示是否已经发送应答消息。所以，在14 429行的检查中，检测到 `tp->tty_inleft` 为0，就可以作为不发送应答消息就中止执行 `in_transfer` 的充分理由。

检测的下一部分是比较 `tp->tty_eotct` 和 `tp->tty_min` 两个变量。在规范模式中，这两个变量与完整的输入行有关；而在非规范模式中，它们与字符有关。只要一个“行中止”字符或一个字节被放入输入队列，就增加 `tp->tty_eotct` 的值；当一行或一个字节从队列移出时，`in_transfer` 递减 `tp->tty_eotct` 的值。换言之，`tp->tty_eotct` 对已被终端驱动程序接收但还未传给一个读请求进程的行或字节进行计数。`tp->tty_min` 则指出了完成一个读请求所必须传送的最小行数（规范模式）或字符数（非规范模式）。在规范模式中它的值总是为1，在非规范模式中，它可以是从0到 `MAX_INPUT`（MINIX 3中为255）之间的任何一个值。14 429行中第二部分的检测功能，使得在规范模式下如果未接收到一个整行，`in_transfer` 就立即返回。传送并不马上执行，而是等到一行结束之后，这样就可以修改队列的内容。例如，用户在按下ENTER之前接着输入了ERASE或KILL字符。在非规范模式中，如果无法取得所需的小字符数，就立即返回。

几行之后，`tp->tty_inleft` 和 `tp->tty_eotct` 被用来控制 `in_transfer` 的主循环。在规范模式中，传送一直持续到队列中不再有完整的行。在非规范模式中，`tp->tty_eotct` 用来对等待的字符计数。`tp->tty_min` 用来控制是否进入循环，但不用来决定何时结束循环。进入循环后，或者传送所有可用的字符，或者传送原始调用所需的字符数，这取决于哪一个更小。

输入队列中的字符大小为16位。实际传给用户进程的是低8位。图3.40描述了高位字节是怎样使用的。3位用来标志字符是否被转义（CTRL-V），是否表示文件结束，或者是否表示行结束的几个代码之一。4位用于计数字符回显时占据的屏幕空间。14 435行检查 `IN_EOF` 位（图中的D）是否被置位。检测是在内循环顶部进行的，因为文件结束符（CTRL-D）本身不传给读进程，也不计入字符计数。在传送每个字符时，用一个掩码使高8位为0，只有低8位中的ASCII值被送到局部缓冲区（14 437行）。

0	V	D	N	c	c	c	c	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- V: IN_ESC, 转义标志符 (CTRL-V)
- D: IN_EOF, 文件结束符 (CTRL-D)
- N: IN_EOT, 行中止符 (NL或其他)
- cccc: 回显的字符数
- 7: 位7, 如果ISTRIP被置位则为0
- 6-0: 6-0: 位0-6, ASCII码

图3.40 输入队列的字符码的域

标志输入结束的方法不止一种，但一般使用设备相关的输入例程来确定接收到的字符是否为换行符、CTRL-D或其他此类字符，并对这些字符进行标记。`in_transfer` 只需要检查这个标记，即在

14 454 行的 *IN_EOT* 位（图 3.40 中的 *N*）。如果检测到，就减少 *tp->tty_eotct* 的值。在非规范模式中，每个字符在放入输入队列中时都这样计数，并且每个字符都在这时候被标记 *IN_EOT* 位，所以 *tp->tty_eotct* 实际上就是队列中未移走的字符的计数。*in_transfer* 的主循环在两种不同模式下的唯一区别在 14 457 行。这里如果发现一个字符标志为行中止符，则 *tp->tty_inleft* 被置为 0，但只有在规范模式下才这样做。这样，当控制返回到循环的顶部时，规范模式中循环在一个行中止符之后终止，但在非规范模式中行中止符被忽略。

当循环结束时，一般会有一个部分被填满的局部缓冲区等待传送（14 461 到 14 468 行）。然后，如果 *tp->tty_inleft* 达到 0，那么就发送一条应答消息。在规范模式下这种情况很普遍，但如果处于非规范模式并且传送的字符数少于全部的请求，那么将不发送应答。如果还记得在调用 *in_transfer* 函数时所看到的（在 *do_read* 和 *handle_events* 中），当 *in_transfer* 函数在返回时传送了多于 *tp->tty_min* 所指定的字符数时，在 *in_transfer* 函数调用之后的代码将发送一条应答消息，那么读者会觉得很迷惑，因为这和刚才讲的结论不同。讨论下一个函数时我们将会看到，在一个不同的环境中调用 *in_transfer* 函数时，为什么 *in_transfer* 那时不能无条件地发送应答消息。

接下来要讨论的函数是 *in_process*（14 486 行）。它由设备相关的软件调用来完成对所有输入都要进行的共同的处理。它的参数包括一个指向源设备的 *tty* 结构的指针、一个指向待处理的 8 位字符数组的指针和一个计数器。计数器被返回给调用者。*in_process* 是一个很长的函数，但并不复杂。它把 16 位的字符加入到输入队列中，这些字符稍后将由 *in_transfer* 函数处理。

in_transfer 函数提供了几种不同的处理方法。

1. 普通字符被加入到输入队列中，并扩展为 16 位。
2. 影响以后处理的字符，通过修改标志位来表示影响，但不放入队列。
3. 控制回显的字符立即生效，不放入队列。
4. 具有特殊标记的字符在放入缓冲区时，在高位字节加入了 *EOT* 位之类的代码。

先来看一下一个完全正常的环境：在一个设置为标准 MINIX 3 默认属性的终端上，一个普通字符，例如“x”（ASCII 码 0x78），在一个短行中间输入，其中没有转义序列。当从输入设备接收到时，这个字符占用了图 3.40 中的 0 到 7 位。在 14 504 行，如果 *ISTRIP* 被置位，那么该字符的最高有效位第 7 位被重置为 0，但 MINIX 3 中的默认值不清除这一位，而是允许输入全部的 8 位。不过这不会影响“x”。MINIX 3 的默认值是允许输入的扩充处理，所以能够通过对 *tp->tty_termios.c_lflag*（14 507 行）中 *IEXTEN* 位的检测。但接下来的检测在这里的假设之下不能通过：没有字符转义（14 510 行）。输入本身不是字符转义字符（14 517 行），并且该输入不是 *REPRINT* 字符（14 524 行）。

下面几行的检测判断出字符既不是特殊 *_POSIX_VDISABLE* 字符，也不是 *CR* 和 *NL* 字符。最后，一个正的结果：处于规范模式，这是正常的默认值（14 324 行）。然而，“x”既不是 *ERASE* 字符，也不是 *KILL*, *EOF*(*CTRL-D*), *NL* 或 *EOL* 字符之一，所以直到 14 576 行，仍然没有对它进行任何处理。这里检测到 *IXON* 被默认置位，允许使用 *STOP* (*CTRL-S*) 和 *START* (*CTRL-Q*) 字符，但在接下来的测试中却没有发现匹配。在 14 597 行发现 *ISIG* 位被默认置位，允许使用 *INTR* 和 *QUIT* 字符，但同样找不到匹配。

实际上，对于一个普通字符，第一个值得注意的处理发生在 14 610 行，这里对输入队列是否已满进行检测。如果已满，那么字符将在这里被丢弃，因为这时处于规范模式下，用户看不到它回显在屏幕上（*continue* 语句丢弃该字符，因为它重新开始外循环）。然而，由于在说明时假设了完全正常的情况，所以这里假定缓冲区还没有满。下一个检测判断是否需要一个特殊非规范模式处理

(14 616行),如果失败,就向前跳到14 629行。这里由于`tp->tty_termios.c_lflag`中的`ECHO`位是默认置位,所以调用`ECHO`向用户显示字符。

最后,在14 632到14 636行,字符被放入输入队列。这时要增加`tp->tty_incount`的值,但是由于该字符为普通字符,没有`EOT`位标志,所以`tp->tty_eotct`不变。

如果刚送入队列的字符填满了队列,那么循环中的最后一行就调用`in_transfer`。然而在一般条件下,对于这个例子假设`in_transfer`即使被调用也将不执行任何动作,这是因为(假设队列被正常服务,并且当前一行输入结束时,前一次输入被接受)`tp->tty_eotct`为0,`tp->tty_min`为1。在`in_transfer`开头的检测(14 429行)使得它立即返回。

在一般情况下用一个一般字符处理过程遍历了一遍`in_process`后,现在再回到`in_process`的开头看一看特殊情况下将会如何。首先讨论转义字符,它允许一个普通字符把一种特殊影响传递给用户进程。如果转义字符有效,那么`tp->tty_escaped`将被置位,当检测到这个标志时(在14 510行),该标志立即被重置,并且`IN_ESC`位,即图3.40中的V位,被添加到当前的字符中。这使得字符在回显时被特殊处理——转义控制字符用“^”加字符的形式显示,以使它们可见。`IN_SEC`位还可以防止字符被识别为其他特殊字符。

下面几行处理转义字符本身,即`LNEXT`字符(默认为`CTRL-V`)。当检测到`LNEXT`字符时,`tp->tty_escaped`标志被置位,并且调用两次`rawecho`输出一个“^”,后面接一个退格符。这提醒用户是在转义字符有效的键盘上操作,当后续字符被回显时将覆盖“^”。`LNEXT`字符是影响后续字符的一个例子(在这里,只是紧接着的字符)。它不放在队列中,并且在两次`rawecho`调用后重新启动循环。这两次检测的次序很重要,这使得可以在一行中输入`LNEXT`两次而把第二个副本作为实际数据传给一个进程。

`in_process`处理的下一个特殊字符是`REPRINT`字符(`CTRL-R`)。当`in_process`发现接下来是一个`reprint`调用时(14 525行),将重新显示当前的回显输出。`REPRINT`本身被丢弃,对输入队列没有影响。

考虑每个特殊字符的处理细节是很单调乏味的,并且`in_process`的源代码是很简单明了的,这里将只提几个地方。一个是在放入输入队列的16位值的高位使用特殊位,从而区分有相似作用的一类字符就显得很容易。因此,`EOT`(`CTRL-D`),`LF`和可选的`EOL`字符(默认时没有定义)都有`EOT`位标志,即图3.40中的D位(14 566到14 573行),这使得后面的识别更容易。

最后将说明前面提到的`in_transfer`的特殊行为。虽然在前面看到在`in_transfer`调用中似乎在返回时总是产生一条应答消息,但实际上应答消息并不是每次终止时都产生。回忆当输入队列已满时(14 639行),`in_process`对`in_transfer`的调用在规范模式下没有作用。但如果在非规范模式下则需要处理,14 618行将为所有的字符打上`EOT`标志,这样每个字符都在14 636行中被`tp->tty_eotct`计数。在返回时,由于在非规范模式下输入队列已满,将进入调用`in_transfer`时的主循环。在这种情况下,当`in_transfer`终止时不应该发送消息,因为在返回`in_process`后很可能读入了过多的字符。实际上,虽然在规范模式下单个`read`的输入受输入队列大小的限制(MINIX 3中为255个字符),但是在非规范模式下,一个`read`调用必须能够传送POSIX要求的`_POSIX_SSIZ_MAX`个字符。在MINIX 3中这个值为32 767。

下面是`tty.c`中的几个支持字符输入的函数。`tty_echo`(14 647行)对一些字符进行特殊处理,但对大多数字符只是在用于输入的同一设备上显示输出。在输入被回显的同时,一个进程可能正好要向同一个设备输出,这时如果键盘用户正试图从键盘退格,就有可能引起混乱。为了处理这种情况,在产生正常的输出时,设备相关输出例程总是把`tp->tty_reprint`标志设置为`TRUE`。这样,处理退格的函数就能够判断出是否产生了混合输出。由于`tty_echo`也使用设备输出例程,所以`tp->`

tty_reprint 的当前值在回显时由局部变量 *rp* (14 668 到 14 701 行) 保存起来。不过, 如果刚刚开始一个新的输入行, 那么 *rp* 将被置为 *FALSE* 而不是保持原值, 以保证 *tp->tty_reprint* 在 *echo* 终止时被重置。

读者可能已经注意到 *tty_echo* 将返回一个值。例如, 在 *in_process* 中 14 629 行的调用中:

```
ch = tty_echo(tp, ch)
```

echo 返回的值包括回显字符所用的屏幕上的空格数, 如果是 *TAB* 字符, 那么这个值可以一直到 8。这个值被存放在图 3.40 中的 *cccc* 域。普通字符在屏幕上占据一格, 但一个控制字符(除 *TAB*, *NI*, *CR* 或 *DEL* (0x7F) 外) 回显时, 该字符被显示为“*^*”加一个可打印的 ASCII 字符的形式, 在屏幕上占用两个位置。而另一方面 *NL* 或 *CR* 不占用空间。当然, 实际的回显必须由设备相关例程完成, 并且一个字符必须被送往设备时, 就如同在 14 696 行中对普通字符那样, 用 *tp->tty_echo* 执行一个间接调用。

下一个函数 *rawecho* 用来旁路 *echo* 所进行的特殊处理。它检测 *ECHO* 标志是否被置位, 如果是, 就不经过任何特殊处理将字符送往设备相关的 *tp->tty_echo* 例程。这里使用了一个局部变量 *rp*, 以防止 *rawecho* 对输出例程的调用改变 *tp->tty_reprint* 的值。

当 *in_process* 发现一个退格符时, 将调用下一个函数 *back_over* (14 721 行)。它对输入队列进行操作, 如果在队列中可以回溯, 那么就删除队列中的前一个字符。如果队列为空或上一个字符为行中止符, 那么就不能回溯。这里也检测在讨论 *echo* 和 *rawecho* 时提到的 *tp->tty_reprint* 标志。如果为 *TRUE*, 那么就调用 *reprint* (14 732 行) 在屏幕上放置一个输出行的纯副本。然后查询上一个显示字符的 *len* 域(图 3.40 中的 *cccc* 域)来确定需要在显示器上删除多少个字符, 并且对每个字符, 都由 *rawecho* 发出一个退格 - 空格 - 退格字符序列, 从屏幕上删除不需要的字符。

下一个函数是 *reprint*。除了被 *back_over* 调用外, 用户也可以通过按下 *REPRINT* 键 (CTRL-R) 来引用该函数。14 764 到 14 769 行的循环在输入队列中逆向查找最后一个行中止符。如果找到的是位于队列最末尾的位置, 那么 *reprint* 什么也不做就返回, 否则回显 CTRL-R, 即在屏幕上显示两字符序列“*^R*”, 然后移到下一行, 重新显示队列中从上一个行中止符到末尾的部分。

现在已经讨论到 *out_process* 函数 (14 789 行)。和 *in_process* 一样, 它也由设备相关的输出例程调用, 但要简单一些。它由 RS-232 和伪终端设备指定输出例程调用, 但不能被控制台例程调用。*out_process* 工作在字节的环形缓冲区上, 但不从缓冲区中删除字符。它对缓冲区数组所做的唯一变动是, 如果 *tp->tty_termios.oflag* 中的 *OPOST* (能够输出处理) 和 *ONLCR* (将 *NL* 映射为 *CR-NL*) 都被置位, 就在缓冲区中的 *NL* 字符前插入一个 *CR* 字符。MINIX 3 中这两位都默认为置位。*out_process* 的任务是保持设备的 *tty* 结构中变量 *tp->tty_position* 的更新。制表符和退格符使得情况要复杂一些。

下一个例程是 *dev_ioctl* (14 874 行)。当用 *TCSADRAIN* 或 *TCSAFLUSH* 选项调用 *do_ioctl* 时, 它都支持执行 *tcdrain* 函数和 *tcsetattr* 函数。在这些情况下, *do_ioctl* 在输出没有结束时不能立即完成动作, 所以请求的信息被保存在 *tty* 结构的一部分中以延迟 *ioctl* 操作。无论 *handle_events* 何时运行, 它都要在调用设备相关的输出例程后检查 *tp->tty_ioreq* 域, 并且如果有操作不能确定, 就调用 *dev_ioctl*。*dev_ioctl* 检查 *tp->tty_outleft* 来判断输出是否结束, 如果是, 和 *do_ioctl* 在不延迟时执行的动作相同。为 *tcdrain* 服务的唯一动作就是重置 *tp->tty_ioreq* 域并向 FS 发送应答消息, 通知它唤醒原始调用进程。*tcsetattr* 在 *TCSAFLUSH* 选项下调用 *tty_icancel* 取消输入。对 *tcsetattr* 的两个选项, *termios* 结构被复制到设备的 *tp->tty_termios* 结构中, *termios* 结构的地址在对 *ioctl* 的原始调用中已经被传入。然后调用 *setattr*, 接下来和 *tcdrain* 一样, 通过发送一条应答消息来唤醒被阻塞的原始调用。

下一个过程是 *setattr* (14 899行)。如同我们已经看到的那样，它由 *do_ioctl* 或 *dev_ioctl* 调用来改变一个终端设备的属性，而 *do_close* 则调用它把属性重置为默认设置。*setattr* 总是在把一个新的 *termios* 结构复制到一个设备的 *tty* 结构中后被调用，这是因为只复制参数是不够的。如果被控制的设备正处于非规范模式，那么第一个动作就是给当前输入队列中的所有字符都标上 *IN_EOT* 位，与处于非规范模式下这些字符刚被输入到队列中时所要做的一样。直接这样做 (14 913 到 14 919 行) 要比检测字符是否已经设置该位容易一些。因为没有办法知道哪个属性刚被改变，哪个还保持原值。

下一个动作是检测 *MIN* 和 *TIME* 值。在规范模式下 *tp->tty_min* 总是为 1，这在 14 926 行置位。在非规范模式下，两个值的组合允许 4 种不同的操作模式，如图 3.31 所示。在 14 931 到 14 933 行中，*tp->tty_min* 首先被设置为 *tp->tty_termios.cc[VMIN]* 传过来的值，如果它为 0 且 *tp->tty_termios.cc[VTIME]* 不为 0，那么它将被修改。

最后，*setattr* 确保如果 XON/XOFF 控制被禁止，则输出就不停止。如果输出速度被设置为 0，就发送一个 *SIGHUP* 信号，并且执行一个由 *tp->tty_ioctl* 指向的设备相关例程的间接调用，来完成只能在设备层完成的工作。

下一个函数 *tty_reply* (14 952 行) 在前面的讨论中已经被多次提到。它的动作十分简单：构造并发送一条消息。如果因为某种原因应答失败，就会发生混乱。接下来的函数也同样简单。*sigchar* (14 973 行) 请求 MM 发送一个信号。如果 *NOFLSH* 标志没有置位，那么已在排队的输入将被删除——接收到的字符或行的计数被置为 0，队列的头指针和尾指针重合。这是默认动作。当即将捕获一个 *SIGHUP* 信号时，*NOFLSH* 可以被置位，以在捕获信号后恢复输入和输出。*tty_icancel* (15 000 行) 根据 *sigchar* 描述的方法无条件地丢弃等待的输入，并且另外再调用由 *tp->tty_icancel* 指向的设备相关函数，取消可能留在设备本身或被低层代码缓冲的输入。

在 *tty_task* 第一次启动时为每个设备调用一次 *tty_init* (15 013 行)。*tty_init* 设置为默认值。开始时，指向 *tty_devnop* 的指针和一个什么也不做的空函数被设置到变量 *tp->tty_icancel*, *tp->tty_ocancel*, *tp->tty_ioctl* 和 *tp->tty_close* 中。然后 *tty_init* 根据不同类型的终端（控制台、串行线或虚拟终端）调用设备相关的初始化函数。它们设置间接调用的设备相关函数的实际指针。前面讲过，如果根本就没有配置某一特定类型的设备，那么将生成一个立即返回的宏，这样就不需要为没有配置的设备编译代码了。调用 *scr_init* 初始化控制台驱动程序，并且也为键盘调用它的初始化例程。

下面的三个函数支持定时器。看门狗定时器用一个指向定时器到期时运行的函数指针来初始化。*tty_timed_out* 是终端任务为大多数定时器设定的函数。它设置事件标志位，来强制处理输入输出。*expire_timers* 处理终端驱动程序的定时器队列。回忆可知，这是 *tty_task* 的主循环在收到一个 *SYN_ALARM* 消息时调用的函数。库例程 *tmrs_exptimers* 用于在定时器的链表中来回移动，使失效或调用任何已经到期的看门狗函数。从库函数返回时，如果队列仍是活跃的，则会为另一个 *SYS_ALARM* 调用 *sys_setalarm* 内核调用。最后，*settimer* (15 089 行) 设定定时器以确定在非规范模式下何时从 *read* 调用返回。它以一个指向 *tty* 结构的指针 *tty_ptr* 和一个代表 *TRUE* 或者 *FALSE* 的整数 *enable* 作为参数来调用。库函数 *tmrs_settimer* 和 *tmrs_clrtimer* 用于启用和禁止一个定时器，这由 *enable* 这个参数决定。当定时器启用时，看门狗函数总是前面描述过的 *tty_timed_out*。

tty_devnop 的描述 (15 125 行) 比它的执行码还长，因为它什么也没有。它是一个无操作函数，被间接地定位在一个不需要服务的设备中。前面已经看到，在为一个设备调用它的初始化例程之前，在 *tty_init* 中 *tty_devnop* 被用做设置输入到各个函数指针的默认值。

tty.c 中的最后的内容需要一些解释。*select* 是单进程在多个 I/O 于不可预测的时间需要服务时使用的系统调用。一个典型的程序是，需要关注本地键盘和远程系统的通信程序，它可能使用调制解调器来连接。*select* 调用允许打开几个设备文件并监视它们，看它们何时可以读出或写入而没有

阻塞。如果没有 `select`, 则需要使用两个程序来处理双向通信, 一个作为主程序来处理一个方向的通信, 另一个作为从程序处理另一个方向的通信。`select` 是那种拥有它很好但又会严重地使系统复杂的一个例子。MINIX 3 的一个设计目标是足够地简单, 以使得能在适度的时间内适度的努力下理解系统, 并且不得不设置一些限制。由于这个原因, 这里将不会讨论 `do_select` (15 135 行) 以及这里的支持例程 `select_try` (14 313 行) 和 `select_retry` (14 348 行)。

3.8.5 键盘驱动程序的实现

现在我们转而分析一下支持由一个 IBM PC 键盘和一个内存映射显示器组成的 MINIX 3 控制台的设备无关代码。支持它们的物理设备是完全独立的: 在一个标准的桌面系统中显示器使用一块插在底板上的适配卡 (至少有 6 种不同的基本型号), 而键盘由设计在主板上的电路支持, 通过该电路与键盘内部的一个 8 位单片机接口。两个子设备需要完全独立的软件支持, 即文件 `keyboard.c` 和 `console.c`。

操作系统把键盘和控制台视为同一个设备 `/dev/console` 的两个部分。如果显示卡上有足够的存储空间, 就可以把虚拟控制台 (virtual console) 支持编译进去。这样, 除 `/dev/console` 外, 还可以有其他的逻辑设备, 如 `/dev/ttyc1`, `/dev/ttyc2` 等。来自一个设备的输出只在任何给定的时刻送往显示器, 并且对任何被激活的控制台都只有一个键盘用来输入。在逻辑上键盘对于控制台是有作用的, 但只有两个相对次要的方面说明了这一点。首先, `tty_table` 包含了一个控制台的 `tty` 结构, 为输入和输出提供了单独的域, 例如, `keyboard.c` 和 `console.c` 中的函数指针 `tty_devread` 和 `tty_devwrite` 在启动时被赋值。不过, 只有一个 `tty_priv` 域, 并且它只指向控制台的数据结构。其次, 在进入主循环前, `tty_task` 调用每个逻辑设备依次来进行初始化。为 `/dev/console` 调用的程序位于 `console.c` 中, 从那里调用键盘的初始化代码。不过这种隐含的层次关系也可能被倒过来。在处理 I/O 设备时总是先看输入再看输出, 现在仍然保持这种模式, 在这一节中讨论 `keyboard.c`, 而把 `console.c` 的讨论留到后面。

和已经看到的大部分源文件一样, `keyboard.c` 的开头也是几条 `#include` 语句。不过其中有一条比较特殊。`keymaps/us-std.src` (在 15 218 行中包含) 不是一个普通的头文件, 这是一个 C 源文件, 它使得默认键位映射表被编译为 `keyboard.o` 中的一个已初始化的数组。由于篇幅所限, 键位映射表源文件未在附录 B 中列出, 但图 3.37 中列出了一些有代表性的表项。在 `#include` 语句之后的是定义各种常量的宏。第一组用于和键盘控制器的低层交互。其中许多是在这些交互中有意义的 I/O 端口地址或位组合。下一组包含了一些特殊键的符号名。15 249 行中用符号 `KB_IN_BYTES` 定义了环形的键盘输入缓冲区的大小, 其值为 32。接下来定义了缓冲区本身和管理缓冲区变量。由于这些缓冲区中只有一个必须要处理, 因此必须要注意确保在虚拟终端变化以前来处理它的所有内容。

下一组变量用来保存在解释一个按键时所需要保留的各种状态。它们的使用各不相同。例如, 每次当 Caps Lock 键按下时, `caps_down` 标志的值 (15 266 行) 在 `TRUE` 和 `FALSE` 之间切换。当 Shift 键被按下时, `shift` 标志 (15 264 行) 被置为 `TRUE`, 当 Shift 键被松开时, 该标志被置为 `FALSE`。当接收到一个转义扫描码时, `esc` 变量被置位。在收到后续字符时, 它总是被重置。

`map_key0` (15 297 行) 被定义为一个宏。它返回一个扫描码对应的 ASCII 码, 忽略修饰符。这等价于键位映射表数组中的第一列。它的同类函数是 `map_key` (15 303 行), 该函数进行扫描码到 ASCII 码的完全映射, 包括处理和普通字符同时按下的 (多重) 修改键。

键盘中断服务例程是 `kbd_interrupt` (15 335 行), 它在一个键被压下或松开时调用。它调用 `scode` 从键盘控制器芯片获取扫描码。当一个键被松开引起中断时, 其扫描码的最高有效位被置位, 在这种情况下该码被忽略, 除非它是修改键之一。然而, 为要尽可能快地为中断服务, 就要做尽可能少

的工作,出于这样的考虑,所有原始的扫描码被放在环形缓冲区中,并且当前控制台的`tp->tty_events`标志被置位(15 350行)。为了方便讨论,这里还是像前面一样,假定未调用`select`,并且`kbd_interrupt`在这以后会立即返回。图3.41给出了在缓冲区中的一短行输入的扫描码,其中包含两个大写字符,它们的前导码是按下一个Shift键的扫描码,而后续码是松开Shift键的扫描码。初始化时保存有键按下和释放对应的编码。

42	35	163	170	18	146	38	166	38	166	24	152	57	185
L+	h+	h-	L-	e+	e-	l+	l-	l+	l-	o+	o-	SP+	SP-
54	17	145	182	24	152	19	147	38	166	32	160	28	156
R+	w+	w-	R-	o+	o-	r+	r-	l+	l-	d+	d-	CR+	CR-

图3.41 在输入缓冲区中由键盘输入的一行文本的扫描码,下面一行是对应的按下的键。其中,L+,L-,R+和R-分别代表按下和松开左、右Shift键。松开一个键的扫描码比按下同一个键时大128

当`tty_task`接收到来自键盘的`HARD_INT`时,并不会执行完整的主循环。13 795行的`continue`语句使一次新的主循环立即开始(13 764行,这要稍微简单一些,一些条件码被放在列表中,它们表明如果启用了串口线驱动程序,那么它的用户空间的中断处理器也能被调用)。当执行到循环的顶部时,会发现终端设备的`tp->tty_events`标志被置位,并且使用控制台`tty`结构内`tp->tty_devread`域中的指针调用设备相关例程`kb_read`(15 360行)。

`kb_read`从键盘的环形缓冲区中取出扫描码,并把ASCII码放在它的局部缓冲区中,缓冲区应足够大,以保存来自数字键盘的某些扫描码产生的转义序列。然后它调用硬件无关代码中的`in_process`,并把字符放入输入队列。15 379行的`icount`值被减少。对`make_break`的调用以整数形式返回ASCII码。例如小键盘和功能键等特殊键,它们的值大于0xFF。按下数字键盘将产生`HOME`和`INSRT`之间的扫描码(0x101到0x10C,在`include/minix/keymap.h`中定义),并且通过`numpad_map`数组转换为图3.42所示的三字符转义序列。

键	扫描码	ASCII码	转义序例
Home	71	0x101	ESC [H
上箭头	72	0x103	ESC [A
Pg Up	73	0x107	ESC [V
-	74	0x10A	ESC [S
左箭头	75	0x105	ESC [D
5	76	0x109	ESC [G
右箭头	77	0x106	ESC [C
+	78	0x10B	ESC [T
End	79	0x102	ESC [Y
下箭头	80	0x104	ESC [B
Pg Dn	81	0x108	ESC [U
Ins	82	0x10C	ESC [@

图3.42 由数字键盘产生的转义编码。普通键的扫描码被转换为ASCII码,而特殊键被转换为大于0xFF的“伪ASCII”码

这些序列接着被传送到`in_process`中(15 392到15 397行)。更高的扫描码不传给`in_process`,但要检测ALT-LEFT-ARROW,ALT-RIGHT-ARROW和ALT-F1到ALT-F12这些扫描码。如果发现

其中之一，就调用 *select_console* 切换虚拟控制台。类似地，也会对 CTRL-F1 到 CTRL-F2 进行特殊处理。CTRL-F1 显示功能键的映射（后面会有详细的讨论）。CTRL-F3 切换终端屏幕的硬件滚屏和软件滚屏。CTRL-F7, CTRL-F8 和 CTRL-F9 分别产生与 CTRL-\, CTRL-C 和 CTRL-U 相同效果的信号，但那些不能通过 *stty* 命令改变的信号除外。

make_break (15 431 行) 把扫描码转换为 ASCII 码，然后更新跟踪修改键状态的变量。首先，它检测用来在 MS-DOS 下 PC 用户强制重启动的 CTRL-ALT-DEL 键组合。注意最好在较低的层次来完成检测这个注释。然而，MINIX 3 内核空间的中断处理器的简单性使得在底层检测 CTRL-ALT-DEL 不可能，因为当发出一个中断通知时，扫描码还没有读到。

由于希望能正常关机，所以要通过 *sys_kill* 这个内核调用向 *init*（所有进程的父进程）发送一个 *SIGKILL* 信号 (15 448 行)，而不是试图启动 PC BIOS。*init* 应该在返回到可以完全重启系统或重启 MINIX 3 的引导监控程序之前接收这个信号，把它解释为开始一个按次序关机的进程命令。

当然，我们希望这个过程每次都能完成是不现实的。大部分用户清楚突然关机的危险，也知道除非在系统真正出现问题而失去控制时才应该按 CTRL-ALT-DEL 键。当系统发生混乱时已无法按次序向另一个进程发送信号。这就是为什么在 *make_break* 中有一个静态变量 *CAD_count* 的原因。大多数系统崩溃时，中断系统仍能工作，所以键盘输入仍然可以被接收，终端驱动程序仍保持运行。这里 MINIX 3 利用了计算机用户的可预期行为，即在某些东西看起来不能正常工作时用户会反复击键。如果试图杀死 *init* 失败，并且用户按了 CTRL-ALT-DEL 键两次以上，就直接调用 *sys_abort* 返回到监控程序而不再调用 *init*。

make_break 的主要部分不难看懂。变量 *make* 记录扫描码由按下还是释放一个键产生，接着被调用的 *map_key* 把 ASCII 码返回给 *ch*。接下来是处理 *ch* 的一个 *switch* 语句 (15 460 到 15 499 行)。这里要考虑两种情况，一种是普通键，一种是特殊键。对于普通键没有可以匹配的情况，在默认情况下如果 *make* 为真则返回键扫描码 (15 498 行)。如果由于某种原因一个普通键在放开时被接收，那么就用 -1 代替，而调用者 *kb_read* 将忽略这个键。而一个特殊键，例如 *CTRL*，会在 *switch* 语句中合适的地方被检测出来，对于 *ctrl* 是在 15 461 行。在这种情况下对应的变量为 *control*，用来记录 *make* 的状态，用 1 代替要返回的（和被忽略的）字符扫描码。对 *ALT*, *CALOCK*, *NLOCK* 和 *SLOCK* 键的处理要复杂一些，但是对所有这些特殊键的作用效果是相似的：一个变量记录了当前的状态（对只在按下时有效的键）或保持前一个状态（对锁定键）。

另外还要考虑一种情况，就是 *EXTKEY* 扫描码和 *esc* 变量。它不会和键盘上的 *ESC* 键混淆，返回的 ASCII 码是 0x1B。从键盘上按任何一个键或组合键都无法产生单独的 *EXTKEY* 扫描码；它是 PC 键盘的扩展键前缀 (extended key prefix)，是一个两字节扫描码的第一个字节，表明按下一个不是初始 PC 扩充键的一部分但具有相同扫描码的键。在很多情况下，软件对这两个键的处理是等同的。例如，普通的 “/” 字符和数字键盘上灰色的 “/” 键，就是这样的例子。在其他情况下，则可能需要区别这些字符。例如，许多非英语键盘布局对左右 *ALT* 键做不同的处理，来支持必须产生三个不同字符扫描码的键。两个 *ALT* 键都产生相同的扫描码 (56)，但当右 *ALT* 键按下时前面要加上 *EXTKEY* 码。当返回 *EXTKEY* 码时，*esc* 标志被置位。在这种情况下，*make_break* 从 *switch* 内部返回。在正常返回前旁路了在每种情况下把 *esc* 置为 0 (15 458 行) 的最后一步。这使得 *esc* 只对紧接着收到的扫描码有效。如果读者熟悉 PC 键盘在普通应用下的各种复杂情况，则会熟悉这种情况，但也会感觉有点奇怪，因为 PC BIOS 不允许读一个 *ALT* 键的扫描码，并且为扩展扫描码返回一个不同的值，而 MINIX 3 却允许这样做。

set_leds (15 508 行) 打开或关闭 PC 键盘上表示 *Num Lock*, *Caps Lock* 或 *Scroll Lock* 键是否被按下的指示灯。一个控制字节 *LED_CODE* 被写到输出端口，通知键盘写到端口的下一个字节是指

示灯控制字节，三个指示灯的状态用该字节中的三位进行编码。当然，这些操作由内核调用来完成，而内核调用请求系统任务向输出端口写。下面两个函数支持这个操作。调用 `kb_wait` (15 530行) 用来决定键盘是否已准备好接收一个命令序列；调用 `kb_ack` (15 552行) 用来检验该命令是否被响应。这些命令都使用忙等待，即持续地读下去，直到出现一个期望的扫描码。对于处理大多数的I/O操作，这并不是一种值得推荐的技术。但是打开或关闭键盘上的指示灯并不经常发生，所以如果效率稍低一些并不会浪费很多时间。注意这两个函数也都可能失败，如果发生了这种情况，可以从函数的返回码判断出来。通过限制重试的次数，即循环中的计数器，来处理超时设定。但是设置键盘上的灯并不是很重要的工作，并不值得去检查这两个调用的返回值，所以 `set_leds` 直接继续执行。

因为键盘是控制台的一部分，所以它的初始化例程 `kb_init` (15 572行) 从 `console.c` 中的 `scr_init` 调用，而不是直接从 `tty.c` 的 `tty_init` 中调用。如果可以使用虚拟控制台（即 `include/minix/config.h` 中的 `NR_CONS` 大于 1），则对每个逻辑控制台调用一次 `kb_init`。下一个函数 `kb_init_once` (15 583行) 只被调用一次，像它的名字暗示的那样。它打开键盘上的指示灯，并扫描键盘以确保没有剩余的按键被读入。然后它初始化两个数组 `fkey_obs` 和 `sfkey_obs`，这两个数组用来将功能键绑定到响应它们的例程。一切都初始化完毕后，它调用两个内核调用 `sys_irqsetpolicy` 和 `sys_irqenable`，来设定键盘的IRQ并把它配置成自动重新启用，所以当有键按下或释放时总会向 `tty_task` 发出通知消息。

尽管下面有更多的机来讨论功能键如何工作，但在这里描述 `fkey_obs` 和 `sfkey_obs` 数组是很合适的。每个数组都有 12 个元素，因为现代 PC 键盘有 12 个功能键。第一个数组是供没有修改的功能使用的，而第二个数组则在检测到转换的功能键时使用。它们由 `obs_t` 类型的元素构成，`obs_t` 是一个保存了一个进程号和一个整数的结构。这个结构和这些数组在 `keyboard.c` 的 15 279 和 15 281 行中声明。初始化在这个结构的 `proc_nr` 成员中保存一个特殊值，形像地表示为 `NONE`，表明它未被使用。`NONE` 不在有效的进程号的范围内。注意这个进程号不是 `pid`，它用于识别进程表中的项。这个术语可能有些让人糊涂。但通知是发给一个进程号而不是 `pid` 的，因为进程号用于索引 `priv` 表，这个表决定是否允许一个进程接收通知。结构中的整数成员也初始为 0，用于对事件计数。

下面三个函数都非常简单。`kbd_loadmap` (15 610行) 几乎微不足道。它由 `tty.c` 的 `do_ioctl` 调用把一个键位映射表从用户进程空间复制到内核空间，覆盖由在 `keyboard.c` 开头包含的键位映射表源文件编译成的默认映射表。

从第一次发布起，MINIX 就已经提供了各种系统信息或者其他特殊的动作以响应在系统终端按下 F1, F2 等功能键。这并不是一个其他操作提供的一般服务，但是 MINIX 一直计划成为一个教学工具。这里鼓励用户修补它，所以用户需要额外的帮助来调试。在许多情况下，由按下一个特殊功能键所产生的输出甚至在系统崩溃时也都是可用的。图 3.43 总结了这些键和它们的作用。

这些键可以分为两类。前面提到的 CTRL-F1 到 CTRL-F12 的键组合由 `kb_read` 来检测。这些触发事件由终端驱动程序来处理。这些事件不是必须要显示的。事实上，目前只有 CTRL-F1 提供信息显示，它列出功能键的绑定。CTRL-F3 切换终端屏幕的软件和硬件滚屏，以及其他原因信号 (`cause signals`)。

功能键自己或者和 Shift 键一起被按下可以产生终端驱动程序不能处理的事件。这会导致向服务器和驱动程序发出通知消息。因为服务器和驱动程序能够被加载、启用并在 MINIX 3 已经运行以后禁用，而在编译时静态地绑定这些键是不能让人满意的。为了启用运行期变化 (`run-time changes`)，`tty_task` 接受 `FKEY_CONTROL` 类型的消息。`do_fkey_ctl` (15 624行) 服务这些请求。请求类型是 `FKEY_MAP`, `FKEY_UNMAP` 或者 `FKEY_EVENTS`。前两个使一个进程注册或者释放消息的位图中指定的键，第三个消息类型返回属于调用者的已经被按下的键的位图，并重置这些键的

events 域。服务器进程，即信息服务器（information server 或 IS），初始化引导映像中的进程的设置，并且也促成响应的产生。但是每个驱动程序也能注册响应一个功能键。典型地，以太网驱动程序就这样做，即作为一个转储来显示数据包统计，这在解决网络问题时是很有帮助的。

键	目的
F1	内核进程表
F2	进程内存映射
F3	引导映像
F4	进程特权
F5	引导监控程序参数
F6	IRQ钩子和策略
F7	内核消息
F10	内核参数
F11	定时细节（如果启用）
F12	调度队列
SF1	进程管理器进程表
SF2	信号
SF3	文件系统进程表
SF4	设备和驱动程序映射
SF5	打印键映射
SF9	以太网统计（只针对RTL8139）
CF1	显示键映射
CF3	在软件/硬件终端滚屏间切换
CF7	发送SIGQUIT，作用和CTRL-\相同
CF8	发送SIGINT，作用和CTRL-C相同
CF9	发送SIGKILL，作用和CTRL-U相同

图 3.43 *func_key()* 检测的功能键

func_key (15 715 行) 从 *kb_read* 中调用，以判断是否按下了表示局部处理的特殊键。这个检查在每次接收到扫描码后进行任何其他处理之前完成。如果它不是一个功能键，那么在控制返回 *kb_read* 以前最多可以做三种比较。如果功能键已经注册，就会通知消息发送给合适的进程。如果这个进程只注册了这一个键，那么通知本身已经足够使进程知道应该做什么。如果进程是信息服务器或者另外一个注册了若干键的进程，则需要一个对话，进程必须要向终端驱动程序发出一个 *FKEY_EVENTS* 请求，这个请求将由 *do_fkey_ctl* 来处理，它会告诉调用进程哪个键已经被激活。然后调用进程为每个按下的键分配处理例程。

scan_keyboard (15 800 行) 在硬件接口层工作，从 I/O 端口读写字节。从 15 809 到 15 810 行的序列通知键盘控制器读入了一个字符，该序列读入一个字节，把它的最高有效位置为 1 并再写一次，再把该位重置为 0 再重写一次。这样就防止了后续的读操作读入相同的数据。在读键盘时没有状态检测，但在任何情况下都应该没有问题，因为 *scan_keyboard* 只在响应中断时被调用。

keyboard.c 中的最后一个函数是 *do_panic_dumps* (15 819 行)。系统崩溃时此函数将被调用，它为用户提供了用功能键显示调试信息的机会。15 830 到 15 854 行的循环是使用忙等待的另一个例子。它重复地读键盘直到键入了一个 ESC。当然没有人可以断言在崩溃后等待命令重启时需要一种更有效的技术。在循环内部调用很少使用的非阻塞的接收操作 *nb_receive*，用于允许交替地接受信息，如果可能，还要为输入测试键盘，这被认为是如下消息中建议的选项中的一个：

```
Hit ESC to reboot, DEL to shutdown, F-keys for  
(按ESC重启, 按DEL关机, 按功能键获得调试转储)
```

这条消息在进入这个函数时打印。接下来将看一下实现 *do_newkness* 和 *do_diagnostics* 的代码。

3.8.6 显示驱动程序的实现

如果有足够的内存, 那么可以将 IBM PC 显示器配置为若干个虚拟终端。这一节将探讨控制台的设备相关代码, 还要研究使用键盘和显示器低层服务的调试转储例程 (debug dump routine)。这些例程支持与控制台使用者有限的交互, 即使在 MINIX 3 系统的其他部分不工作时, 它们还能在系统几乎完全崩溃之后提供有用的信息。

把控制台输出到 PC 内存映射屏幕上的硬件特定代码位于 *console.c* 中。*console* 结构在 15 981 到 15 998 行中定义。在某种意义上, 这个结构是 *tty.c* 中定义的 *tty* 结构的扩展。在初始化时, 控制台 *tty* 结构的 *tp->tty_priv* 域被赋予一个指向自己的 *console* 结构的指针。*console* 结构中的第一项是一个指向对应的 *tty* 结构的指针。*console* 结构中的成员对应于一个视频显示器: 记录光标行列位置的变量, 显示内存的起始地址和界限, 由控制器芯片的基指针指向的内存地址和光标的当前地址。其他的变量用来管理转义序列。因为字符最初被接收时是 8 位字节的形式, 必须和属性字节组合并以 16 位字的形式传送到视频内存, 所以在 *c_ramqueue* 中建立了一个待传送的块, 这个块是一个足够大的数组, 用来存放一行 80 列 16 位的字符-属性对。每个虚拟控制台需要一个 *console* 结构, 它存放在数组 *cons_table* 中 (16 001 行)。和处理 *tty* 以及其他结构一样, 一般通过一个指针 (例如 *cons->c_tty*) 来引用 *console* 结构中的元素。

cons_write 函数的地址存放在每个控制台的 *tp->tty_devwrite* 入口处 (16 036 行)。它只在一个地方被调用, 即 *tty.c* 中的 *handle_events*。*console.c* 中的其他大部分函数的存在都是为了支持这个函数。当客户进程调用一个 *write* 后第一次调用 *cons_write*, 待输出的数据位于客户进程的缓冲区中, 可以用 *tty* 结构中的 *tp->tty_outproc* 和 *tp->out_vir* 域找到这个数据。*tp->tty_outleft* 域记录了还有多少字符需要传送, *tp->tty_outcum* 被初始化为 0, 表示什么都没有传送。这是进入 *cons_write* 时的一般情况, 因为在正常情况下, 一旦该函数被调用, 它就传送原始调用请求的所有数据。不过, 如果用户希望进程能慢下来, 以便在屏幕上查看数据, 那么他可以在键盘上键入一个 *STOP* (CTRL-S) 字符, 这将引起 *tp->tty_inhibited* 标志置位。即使 *write* 调用还没有结束, *cons_write* 在该标志置位时也立即返回。在这种情况下, *handle_events* 将继续调用 *cons_write*, 当 *tp->tty_inhibited* 最终被重置时, 通过用户键入 *START* 字符 (CTRL-Q) 可以让 *cons_write* 继续中断的传送。

cons_write 的第一个参数是一个指向特定控制台的 *tty* 结构的指针, 所以首先初始化 *cons*, 这是指向这个控制台的 *console* 结构的指针 (16 049 行)。然后, 因为 *handle_events* 不管什么时候运行总是调用 *cons_write*, 所以第一个动作就是检测是否真的有事要做。如果没有就快速返回 (16 056 行)。接下来就进入了 16 061 到 16 089 行的主循环。这个循环在结构上与 *tty.c* 中 *in_transfer* 的主循环很相似。通过调用 *sys_vircopy* 从客户进程的缓冲区中获得数据, 并填入一个可以保存 64 个字符的局部缓冲区, 接着更新指向数据源的指针和计数器, 然后局部缓冲区中的每个字符连同稍后将由 *flush* 传送到屏幕的属性字节一起传送到 *cons->c_ramqueue* 数组中。

可以用不止一种方法来执行从 *cons->c_ramqueue* 传送字符, 如图 3.35 所示。可以调用 *out_char* 来传送每个字符, 如果字符是一个可见字符, 没有一个转义序列, 没有超过屏幕宽度, 并且 *cons->c_ramqueue* 数组没有满, 那么 *out_char* 的特殊服务就变得多余。如果不需要 *out_char* 的完整服务, 那么该字符连同属性字节 (由 *cons->c_attr* 获得) 被直接放入 *cons->c_ramqueue* 中, 并且增加 *cons->c_rwords* (队列的索引)、*cons->c_column* (跟踪屏幕上的当前列) 和指向缓冲区的指针 *tbuf*。这个

把字符直接放入`cons->c_ramqueue`中的动作对应于图3.35中左边的虚线。如果需要，就调用`out_char`（16 082行）。它进行所有的登记，另外还在必要时调用`flush`最终把字符传送到屏幕存储器。

只要`tp->tty_outleft`指示仍有字符等待传送，并且`tp->tty_inhibited`仍然没有置位，那么从用户进程缓冲区到局部缓冲区再到队列的传送就重复进行。当传送停止时，不论是因为`write`操作结束还是`tp->tty_inhibited`被置位，都要再次调用`flush`把队列中余下的字符传送到屏幕存储器。如果操作完成（检测中`tp->tty_outleft`是否为0），就调用`tty_reply`（16 096到16 097行）发送一条应答消息。

除了从`handle_events`调用`cons_write`外，待显示的字符还由终端驱动程序的硬件无关部分的`echo`和`rawecho`送往控制台。如果控制台是当前的输出设备，就由`tp->tty_echo`指针调用下一个函数`cons_echo`（16 105行）。`cons_echo`通过调用`out_char`后再调用`flush`完成所有的工作。来自键盘的输入逐个字符地到达，而输入的人希望看到没有可察觉延迟的回显，所以把字符送入输出队列并不能满足要求。

`out_char`（16 119行）检测是否有一个转义序列，如果有，就调用`parse_escape`并立即返回（16 124到16 126行）；否则，就进入一个`switch`语句，检查与特殊情况的匹配：空字符、退格符、响铃符等。对这些情况的处理大都容易理解。换行和制表符是最复杂的，因为它们涉及到光标在屏幕上的位置变化，并且可能需要卷屏。最后要检测`ESC`码。如果有，就对`cons->c_esc_state`标志置位（16 181行），对`out_char`的调用转换为对`parse_escape`的调用，直到序列结束。最后，对可打印字符进行默认处理。如果超出了屏幕宽度，则屏幕可能需要卷屏，这时将会调用`flush`。在一个字符被放入输出队列前，要检测队列是否已满，如果已满，就调用`flush`。把一个字符放入队列需要进行与前面在`cons_write`中看到的同样的登记处理。

下一个函数是`scroll_screen`（16 205行）。`scroll_screen`处理屏幕最后一行已满时的向上卷屏，以及在光标定位命令试图把光标移到屏幕最顶行之外时发生向下卷屏。对每个方向的卷动，有三种可能的处理方法，分别用来支持不同的视频卡。

现在我们将研究向上卷动的情况。开始时，`chars`被赋予屏幕尺寸减去一行的值。软件卷屏由一个单独的`vid_vid_copy`调用完成，把`chars`个字符向内存低端移动，移动的距离是一行中的字符数。`vid_vid_copy`是可回卷的，即如果请求移动的内存块溢出了赋予视频显示内存的上边界，就从内存块的低端取出溢出部分，把它移动到高于被移到低端部分的地址处，即把整个块视为一个环形数组。这个调用看起来简单，但执行却相当慢。即使`vid_vid_copy`是汇编语言程序（定义在`drivers/tty/vidcopy.s`中，未在附录B中列出），这个调用也需要CPU移动3840个字节，因此对于汇编语言这也是一项繁重的工作。

软件卷屏永远不会作为默认方法，只有在硬件卷屏不能工作或由于某种原因不能采用时才选择软件方式。一个原因可能是因为希望使用`screendump`命令把屏幕内存存入一个文件或者在远程终端工作时看主终端显示。当使用硬件卷屏时，`screendump`可能会产生不可预知的结果，因为屏幕内存的起始地址和显示器可见部分的开始位置可能不一致。

在16 226行中`wrap`变量被检测，它是一系列复合检测的第一部分。对于比较老式的支持硬件卷屏的显示器，`wrap`为`TRUE`，如果检测失败，那么16 230行将进行简单的硬件卷屏，视频控制芯片使用的原始指针`cons->c_org`被更新为指向显示器左上角显示的第一个字符。如果`wrap`为`FALSE`，那么复合检测将继续测试卷屏操作将要移动的内存块是否超出了分配给该控制台的内存界限。如果是，就再次调用`vid_vid_copy`执行一个回卷移动，把内存块移动到控制台的分配内存的起始处，并且修改原始指针。如果没有重叠，就将控制传递给旧的视频控制器一直使用的简单硬件卷屏方法。这一步包括调节`cons->c_org`，然后把新的原点放入控制器芯片对应的寄存器中。这个调用后面还要调用一次，因为还有一个为获得“卷屏”效果而清空屏幕最下一行的调用。

向下卷屏的代码与向上卷屏的代码很相似。最后，调用 *mem_vid_copy* 清空由 *new_line* 指向的屏幕最下（或最顶部）一行。然后调用 *set_6845* 把新的原点从 *cons->c_org* 中写到对应的寄存器中，*flush* 将确保所有的改动在屏幕上可见。

前面已经好几次提到 *flush* (16 259行)。它使用 *mem_vid_copy* 把队列中的字符传送到视频内存中，更新某些变量，并保证行和列的数值是合理的。例如，如果一个转义序列试图把光标移到一个为负列数的位置上，那么 *flush* 将调整它们。最后将计算光标应处的位置，并与 *cons->c_cur* 比较。如果它们不一致，并且如果现在正在处理的视频内存属于当前的虚拟控制台，那么就调用 *set_6845* 把控制器的光标寄存器设置为正确的值。

图 3.44 绘出了对转义序列的处理如何表示为一个有限状态自动机。这是由 *parse_escape* (16 293行) 实现的，如果 *cons->c_esc_state* 不为 0，就在 *out_char* 的开头调用它。*out-char* 检测到一个 ESC，并且将 *cons->c_esc_state* 设置为 1。当接收到下一个字符时，*parse_escape* 就在指向参数数组起始位置的指针 *cons->c_ese_intro* 中放入一个 ‘\0’，把 *cons->c_esc_parmv[0]* 放入 *cons->c_esc_parmp* 中，并且参数数组本身都置为 0，准备进一步的处理。接着检查 ESC 后面的第一个字符——有效值为 “[” 或 “M”。在第一种情况下，“[” 被复制到 *cons->c_esc_intro* 中，状态转换到 2。在第二种情况下，调用 *do_escape* 来执行这个动作，并且转义状态被重置为 0。如果 ESC 之后的第一个字符不是有效值中的一个，它将被忽略，后继的字符仍然正常显示。

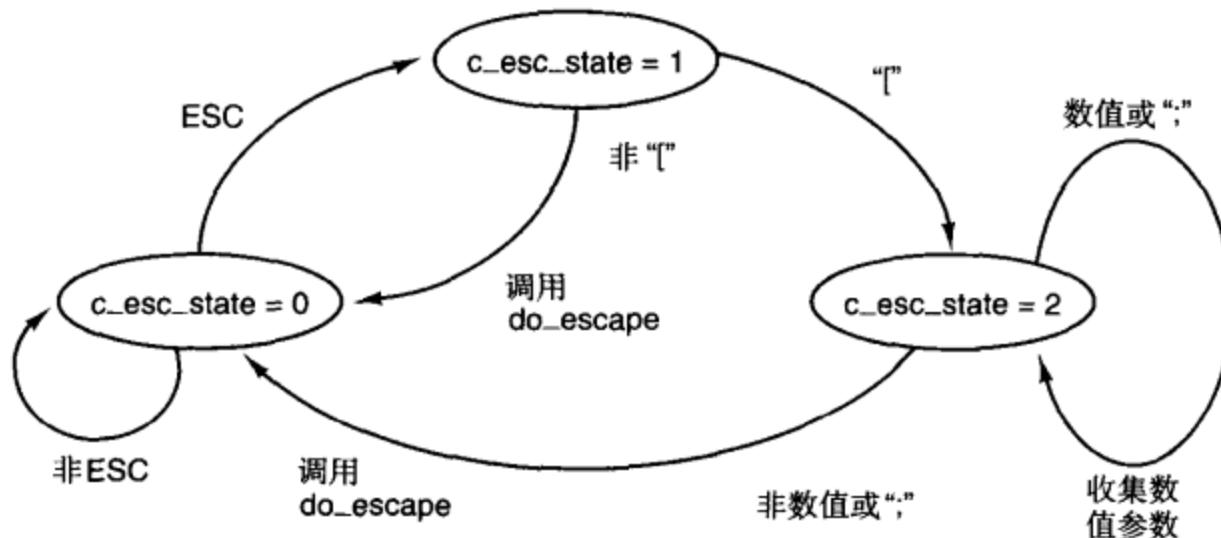


图 3.44 处理转义序列的有限状态机

当检测到一个 ESC [序列时，输入的下一个字符由转义状态 2 的代码处理。这里有三种可能性。如果该字符是一个数值字符，那么它的值被取出，并与 *cons->c_esc_parmp* 所指向的当前位置上的值乘以 10 相加，初始值为 *cons->c_esc_parmv[0]* (被初始化为 0)。转义状态没有变化。这使得可以输入一系列十进制数并累计为一个很大的数值参数，虽然 MINIX 3 现在最大能识别的值为 80，由将光标移到任意位置的序列使用 (16 335 到 16 337 行)。如果该字符为分号，即还有一个参数，那么指向参数字符串的指针就向前移动，这样后续的数值可以在第二个参数中累加 (16 339 到 16 341 行)。如果想要改变 *MAX_ESC_PARMS* 来为参数分配一个大一些的数组，那么也不需要改变这段代码用来在输入额外的参数后累计额外的数值。最后，如果字符既不是一个数字也不是分号，就调用 *do_escape*。

尽管 MINIX 3 对转义序列的识别相对来说并不多，但 *do_escape* (16 352 行) 是 MINIX 3 系统源代码中较长的一个函数。不过，这段代码并不难理解。在执行了一个对 *flush* 的初始调用以保证视频显示被完全更新之后，有一个简单的选择。这个选择取决于紧接着 ESC 之后的字符是否是一个特殊的控制序列引入符。如果不是，那么只有一个有效的动作，即如果该序列是 ESC M 就把光

标上移一行。注意对“M”的检测是在一个switch中的默认动作中完成的，作为有效性检查并等待其他不使用ESC [格式的序列的附加部分。在许多转义序列中，检查`cons->c_row`变量来决定是否需要卷屏的动作很典型。如果光标已经位于第0行，就用`SCROLL_DOWN`来调用`scroll_screen`，否则光标被上移一行。对`cons->c_row`减1然后调用`flush`完成光标上移一行的操作。如果发现了一个控制序列引入符，那么就执行16 377行在else之后的代码。这里要对MINIX 3现在唯一能识别的控制序列引入符“[”进行检测。如果序列有效，那么在转义序列中的第一个参数（没有输入数值参数时为0）被赋给`value`（16 380行）。如果该序列无效，那么除了跳过接下来的switch（16 381到16 586行）并且在从`do_escape`返回之前把转义状态重置为0外，什么也不做。更令人感兴趣的是序列有效的情况，即进入switch语句。这里不讨论所有的情况，只研究转移序列决定的动作中具有代表性的几种。

开始的5个没有数值参数的序列由IBM PC键盘上的4个箭头键和Home键产生。头两个，ESC [A和ESC [B，与ESC M很相似，除了它们可以接受一个数值参数并且可以向上或向下移动多于一行的距离，而且如果参数指定的移动超过了屏幕的界限，它们并不卷动屏幕。在这些情况下，`flush`捕获移出界限的请求并把移动限制在对应的最后一行或第一行。下面两个序列，即ESC [C和ESC [D，把光标向左右移动，它们也类似地受到`flush`的限制。当由箭头键产生时，没有数值参数，这时就默认移动一行或一列。

下一个序列ESC [H可以有两个数值变量，例如，ESC [20;60H。这两个参数指定的是绝对位置而不是相对位置，并且被从以开始1计数的数值转换为从0计数的数值，这样是为了得到正确的解释。Home键产生默认的序列（无参数）把光标移动到(1, 1)处。

下面的两个序列ESC [s J和ESC [s K，清除全部屏幕或当前行的一部分，这将取决于输入的参数。每种情况下都计算字符计数。例如，对于ESC [1J，`count`中保存着从屏幕的开始到光标位置的字符数，这个计数和一个可以是屏幕的起始位置`cons->c_org`或当前的光标位置`cons->c_cur`的位置参数`dst`，一起用做调用`mem_vid_copy`的参数。带参数调用这个过程将使得它用当前背景色填充指定的区域。

下面的4个序列插入或删除在光标处的行或空格，它们的动作不需要详细解释。最后一种情况ESC [n m（注意n代表一个数值参数，但“m”是一个文字）对`cons->c_attr`起作用，即当写到视频存储器时交错存放在字符码之间的属性字节。

下一个函数`set_6845`（16 594行）用来在需要时更新视频控制芯片。6845具有16位的内部寄存器，一次可以对8位编程，写一个寄存器需要4次I/O端口写操作。这些操作能通过设立一个（端口，值）对的数组和调用`sys_voutb`内核调用完成，`sys_voutb`又调用执行I/O的系统任务。图3.45中给出了6845视频控制芯片的部分寄存器。

寄存器	功能
10 - 11	光标大小
12 - 13	重画屏幕的起始地址
14 - 15	光标位置

图3.45 部分6845寄存器

下一个函数是`get_6845`（16 613行），返回可读的视频控制器寄存器的值。它也使用内核调用完成它的工作。它没有在当前的MINIX 3代码的任何位置被调用，但它可能对将来的系统增强（比如增加图形支持）有用。

beep 函数(16 629行)在必须输出CTRL-G字符时被调用。它利用了PC内建的对声音的支持,向扬声器发送方波来使其发出声音。声音的初始化代码大多数是只有汇编程序员才喜爱的I/O端口操作。代码中令人感兴趣的部分是使用设置定时器的功能来关闭蜂鸣。作为一个具有系统特权的进程(即*priv*表中的一项),终端驱动程序可以使用库函数*tmr_settimers*来设定定时器。这是在16 655行完成的,而下一个函数*stop_beep*,被指定作为在定时器到期时运行的函数。这个定时器被放在终端任务自己的定时器队列里。接下来的内核调用*sys_setalarm*请求系统任务来设定内核中的定时器。当它到期时,终端驱动程序的主循环就检测到一个SYN_ALARM消息,*tty_task*调用*expire_timers*来处理所有的属于终端驱动程序的定时器,其中的一个就是由蜂鸣来设定的。

下一个例程*stop_beep*(16 666行)的地址被放在由*beep*初始化的定时器的*tmr_func*里。它在给定的时间后停止发声,并且重置*beeping*标志,它用来防止多余的对发声例程的调用产生任何效果。

scr_init(16 679行)由*tty_init*调用NR_CONS次。每次调用的参数是一个指向*tty*结构的指针,它是*tty_table*的一个元素。在16 693到16 694行,*line*用做*cons_table*数组的索引,它首先被计算出来,然后进行有效性检测。如果有效,就用来初始化指向当前控制台表表项的指针*cons*。这里可以用指向设备的主*tty*结构的指针来初始化*cons->c_tty*,并且*tp->tty_priv*可以指向该设备的*console_t*结构。接着,调用*kb_init*来初始化键盘,然后建立指向设备相关例程的指针,*tp->tty_devwrite*指向*cons_write*,*tp->tty_echo*指向*cons_echo*,*tp->tty_ioctl*指向*cons_ioctl*。16 708到16 731行从BIOS取出CRT控制器基址寄存器的I/O地址,决定视频内存的地址和大小,并根据使用的视频控制器的类型设置*wrap*标志(用来确定如何卷屏)。16 735行,系统任务在全局描述符表中初始化视频内存的段描述符。

接下来是虚拟控制台的初始化。每次调用*scr_init*时,参数是不同的*tp*值,这样16 750到16 753行就用不同的*line*和*cons*为各虚拟控制台提供它自己的可用视频内存部分。接着每个屏幕被清空,准备好开始,最后控制台0被选为第一个活动控制台。

一些例程代表终端驱动程序本身、内核或者其他系统的组件来显示输出。第一个例程*kputc*(16 775行)只是调用*putk*,它是一个一次输出一个字节的文本输出例程,这将在下面讨论。这个例程在此处使用是由于在系统组件内提供*printf*函数的库例程被链接到同名的字符打印例程,但是终端驱动程序的其他函数需要名为*putk*的函数。

do_new_kmess(16 784行)用于打印来自内核的消息。实际上,“消息”并不是用在这里最合适的词;这里消息并不意味着进程间的通信。这个函数是为了在终端上显示文本以向用户报告信息、警告或者错误。

内核需要一个特殊的机制来显示信息。而且,它必须是鲁棒的,这样才能在所有MINIX 3的组件运行前的启动阶段或者在内核混乱时即另外一个系统的大部分都不可用的时间使用。内核向环形的字符缓冲区写文本,这个缓冲区是还包含要写的下一个字节的指针和还需要处理的文本的大小的结构的一部分。当有新的文本时,内核向终端驱动程序发出SYS_SIG消息,当*tty_task*的主循环运行时调用*do_new_kmess*。在事情进行得并不这样顺利时(比如系统崩溃时),SYS_SIG将被主循环检测到,主循环包括*do_panic_dumps*中的非阻塞*read*操作,这已经在*keyboard.c*中见过,并且还会调用*do_new_kmess*。在任意情况下,内核调用*sys_getkmessages*重新得到一个内核结构的副本和需要显示的字节,这是通过一个字节一个字节地传给*putk*完成的,然后用一个空字节最后一次调用*putk*,来强制它马上输出。一个局部静态变量用于跟踪消息间缓冲区中的位置。

do_diagnostics(16 823行)有一个和*do_new_kmess*相似的功能,但是*do_diagnostics*用于显示来自系统进程的消息,而不是内核的。*DIAGNOSTICS*可以由*tty_task*的主循环或者*do_panic_dumps*

中的主循环来接收，并且不管在哪种情况都会调用 *do_diagnostics*。这个消息包括一个指向调用进程中的缓冲区的指针和消息大小的计数。没有局部缓冲区被使用，而是重复调用 *sys_vircopy* 内核调用，一次获得一个字节的文本。这样保护了终端驱动程序；如果发生了错误，一个进程开始产生过多的输出，那么缺缓冲区后导致字符泛滥成灾。字符通过调用 *putk* 一个接一个地被输出，最后是一个空字节。

putk (16 850行) 能够代表链接到终端驱动程序的任何代码来打印字符，并且可以被刚被讨论过的那些函数用于代表内核或其他系统组件来输出文本。它只是为每个接收到的非空字符调用 *out_char*，然后为字符串结尾的空字节调用 *flush*。

console.c 中余下的例程短而简单，这里会很快地概述一下。*toggle_scroll* (16 869行) 的动作与它的名称一样，它切换决定使用硬件还是软件卷屏的标志，还在当前光标位置显示一条信息来确认已选择的模式。*cons_stop* (16 881行) 在关机或重启之前重新将控制台初始化为重启监控程序指定的状态。*cons_org0* (16 893行) 只有在F3键强制切换卷屏模式或准备关机时使用。*select_console* (16 917行) 选择一个虚拟控制台。它用新的索引调用，并调用 *set_6845* 两次，使视频控制器显示视频内存的正确部分。

下两个例程高度依赖于硬件。*con_loadfont* (16 931行) 把一个字体装入图形适配器，以支持 *ioctl TIOCSFON* 操作。它调用 *ga_program* (16 971行) 执行一系列对 I/O 端口的写，使得一般不可被CPU寻址的视频适配器字体内存变为可见。然后调用 *phys_copy* 把字体数据复制到这块内存区，并且引用另一个序列把图形适配器设置回一般操作模式。

最后一个函数是 *cons_ioctl* (16 987行)。它只完成一个功能，即设定屏幕的大小，并且只被使用从 BIOS 获取的值的 *scr_init* 调用，如果有必要调用真实的 *ioctl* 来改变 MINIX 3 的屏幕大小，那么提供新尺寸的代码就会被写。

3.9 小结

输入/输出是一个经常被忽略但是十分重要的话题。任何操作系统中都有相当重要的部分与 I/O 操作有关。同时许多情况下操作系统的问题都是由输入/输出设备驱动程序造成的。通常情况下驱动程序是设备生产商的程序员写的。传统的操作系统设计需要能够让驱动程序访问控制临界资源，例如中断、输入/输出端口以及属于其他进程的存储区。MINIX 3 利用限制特权级将驱动程序隔离为独立的进程，这样如果驱动程序中存在一个缺陷就不至于使得整个系统崩溃。

这里由研究 I/O 硬件开始，分析了 I/O 设备和 I/O 控制器的关系，这是软件必须处理的问题。然后研究了 I/O 软件的 4 个层次：中断例程、设备驱动程序、设备无关 I/O 软件以及在用户空间运行的 I/O 库和假脱机。

这里研究了死锁的问题以及如何处理它。当一组进程都拥有对某些资源的互斥存取权，并且每个进程还要求仍属于该组中另一个进程的资源时，会发生死锁。所有进程都被阻塞，没有一个可以再次运行。通过结构化系统可以避免发生死锁。例如，在任何时刻一个进程只允许拥有一个资源。通过检测每个资源请求是否可能导致死锁(不安全状态)以及拒绝或延迟那些可能引起麻烦的请求，也可以避免死锁。

MINIX 3 中的设备驱动程序是作为用户空间独立的进程来实现的。这里研究了 RAM 磁盘驱动程序、硬盘驱动程序以及终端驱动程序。这些驱动程序都有一个主循环获取请求并进行处理，最终送回应答消息，报告发生的事件。RAM 磁盘、硬盘、软磁盘驱动程序都使用一个相同的主循环副本，并共享相同的函数，不过每一个驱动程序在编译和链接时使用自己的例程库。每一个设备驱动

程序都位于自己的地址空间中。使用系统控制台、串行线和网络连接的几种不同终端都由一个单一的终端任务支持。

设备驱动程序和中断系统有许多不同的关系。能很快完成工作的设备如RAM磁盘和内存映射显示器根本不使用中断。硬盘驱动程序在任务代码本身中完成了大部分工作，中断处理器只返回状态信息。中断在任何时候都是可用的，如果想要等待一个中断，可以使用 `receive`。键盘中断在任何时候都有可能发生。对于终端驱动程序，由中断产生的消息在驱动程序的主循环中被接收并同时被处理。当一个键盘中断发生时，输入处理的第一步会尽快完成以随时准备处理随后的中断。

MINIX 3 驱动程序具有受限的特权级别，因此不能自主地操作中断或者访问输入/输出端口。中断由系统任务管理，当中断发生时，系统任务会发送一个消息以通知驱动程序。对输入/输出端口的访问以类似的方式由系统任务接管。驱动程序不能直接读写输入/输出端口。

习题

1. 一台1倍速的DVD读取器能够以1.32 MB/s的速度传送数据。如果DVD驱动器使用USB 2.0连接，在保证不丢失数据前提下，DVD驱动器的最高传输速度为多少？
2. 许多磁盘在每个扇区结束时有一个ECC，如果ECC错误，将会触发什么动作？这个动作是由什么硬件或者软件来完成的？
3. 什么是存储器映射I/O？为什么在一些情况下要使用它？
4. 尝试解释什么是DMA。为什么要使用它？
5. 虽然DMA不使用CPU，但是最大传输速率仍然受限。考虑从磁盘读取数据块，举出三个最终会限制传输速率的扇区。
6. CD质量的音乐要求每秒对声音信号进行44 100次采样。假如存在一个定时器以这样的速率产生中断，在主频为1 GHz的CPU上要耗费1 μs的时间来处理一个中断。在保证不丢失数据的前提下，最低需要多快的时钟频率？在此假定对于一个中断需要执行的指令数目是恒定的，即如果时钟频率降低一半，则中断处理时间会增加一倍。
7. 中断的一种替代方法是轮询。是否存在一种情况，轮询比中断更加有效？如果有，尝试举出一个例子。
8. 为什么磁盘控制器都有内部缓存且型号越新缓存的容量越大？
9. 每一个设备驱动程序与操作系统之间都有两个不同的接口。一个接口是供操作系统调用驱动程序的一组函数，另一个接口是驱动程序调用操作系统的函数集合。针对每一种接口举出一个合适的例子。
10. 为什么操作系统设计者都尽可能地提供设备无关的输入/输出。
11. 以下的工作各在4个I/O软件层的哪一层完成？
 - (a) 为一个磁盘读操作计算磁道、扇区、磁头。
 - (b) 维护一个最近使用的块的缓冲。
 - (c) 向设备寄存器写命令。
 - (d) 检查用户是否有权使用设备。
 - (e) 将二进制整数转换成ASCII码以便打印。
12. 为什么打印机的输出文件在打印前通常都假脱机输出在磁盘上？
13. 举出一个现实生活中死锁的例子。

14. 考虑图 3.10。假设在步骤(o)中 C 需要 S 而不是 R, 这将会导致死锁吗? 如果既需要 S 又需要 R 呢?
15. 仔细观察图 3.13(b)。如果 D 再多请求一个单位, 将会导致一个安全的状态还是不安全的状态? 如果请求来自 C 而不是 D 呢?
16. 图 3.14 中的所有轨迹都是垂直的或是水平的。你能想象出可能出现的对角线轨迹的情形吗?
17. 假设图 3.15 中的进程 A 请求最后一台磁带驱动器。这会导致死锁吗?
18. 一台计算机有 6 台磁带机被 n 个进程竞争, 每个进程可能需要两台磁带机。那么 n 为多少时, 系统没有死锁的危险?
19. 一个系统可以处于既不死锁也不安全的状态吗? 如果可以, 举出例子; 如果不可以, 请证明所有状态均处于死锁或安全两种状态之一。
20. 一个使用信箱的分布式系统有两条 IPC 原语: SEND 和 RECEIVE。后一个原语指定从哪个进程接收消息, 并且, 如果该进程没有可用的消息, 那么即使消息可能正在等待其他进程, 该原语也阻塞。不存在共享资源, 但是进程因为其他原因需要经常通信。请讨论可能会发生死锁吗?
21. 在一个电子转账系统中, 有数百个相同进程以如下方式工作: 每一进程读取一个输入行, 指定一定数目的款项、贷方账号、借方账号, 然后锁定两个账号, 传送这笔钱, 完成后再解锁。由于许多进程并行运行, 所以存在这样的危险: 锁定 x 将无法锁定 y , 因为 y 已被一个正在等待 x 的进程锁定。设计一个方案来避免死锁。在没有完成事务处理前不要释放账号记录(换言之, 锁定一个账号后立即又将其释放, 因为另一个账号被锁定了, 这种方法是不允许的)。
22. 银行家算法在一个有 m 个资源类和 n 个进程的系统中运行。在 m 和 n 都很大的情况下, 为检查状态是否稳定而进行的操作次数正比于 $m^a n^b$ 。 a 和 b 的值为多少?
23. 考虑图 3.15 中的银行家算法。假如进程 A 和进程 D 改变它们的请求, 现在它们分别还需要资源(1, 2, 1, 0)和(1, 2, 1, 0)。它们现在的请求是否能够被满足? 系统是否能够保持在安全状态?
24. Cinderella 和 Prince 要离婚。为分割财产, 他们商定了以下算法: 每天早晨各人发函给对方律师要求财产中的一项。由于邮递信件需要一天的时间, 他们商定如果发现在同一天两人要求了同一项财产, 第二天他们将发信取消这一要求。他们的财产包括狗 Woofer、Woofer 的狗屋、金丝雀 Tweeter 和 Tweeter 的鸟笼。由于这些动物喜爱它们的房屋, 所以又商定任何将动物和它们房屋分开的方案都无效, 且整个分配从头开始。Cinderella 和 Prince 都非常想要 Woofer。于是他们分别去度假, 并且每人都用一台 PC 处理这一谈判工作。当他们度假回来时, 发现电脑仍在谈判, 为什么? 可能发生死锁吗? 可能发生饥饿(永远等待)状态吗?
25. 假如一个磁盘每磁道拥有 1000 个大小为 512 字节的扇区, 每个柱面上有 8 个磁道, 一共有 10 000 个柱面, 旋转时间为 10 ms, 磁道之间的寻道时间为 1 ms。那么可以承受的最大突发传输速率为多大? 这样的突发可以持续多长时间?
26. 一个局域网以如下方式使用: 用户发出一条系统调用, 请求将数据包写到网上, 然后操作系统将数据复制到一个内核缓冲区中, 再将数据复制到网络控制器板上。当所有数据都安全存储于控制器中时, 把它们在网上以 10 Mb/s 的速率传送, 在每一位被发送后 1 μ s, 接收的网络控制器将它保存。当最后一位到达时, 目标 CPU 被中断, 内核将新到达的数据包复制到内核缓冲区中进行检查。一旦它指明该数据包是发送给哪个用户进程的, 内核就将数据复制到该用户进程空间。如果假设每一个中断及其相关处理过程费时 1 ms, 数据包有 1024 字节

- (忽略头标), 复制一个字节费时 $1\ \mu s$, 将数据从一个进程注入另一个进程的最大速率是多少? 假设发送者一直被阻塞直到接收端的工作完成并且返回一条确认消息。为简便起见, 假设返回确认消息的时间可以忽略不计。
27. 图 3.17 的消息格式被用来向块设备的驱动程序发送请求消息。如果可以, 消息中的哪些域在字符设备中可以省略?
28. 磁盘请求以 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, 计算以下寻道时间:
- 先到先服务。
 - 下一个最邻近柱面。
 - 电梯算法(起始移动向上)。
- 所有情况下磁头臂起始都位于柱面 20。
29. 一个 PC 销售商在访问阿姆斯特丹西南部的一所大学时, 竭力推销他的产品, 声称他的公司通过努力使 UNIX 版本速度非常快。例如, 磁盘驱动程序使用了电梯算法, 并将一个柱面内的多个请求按扇区次序排队。学生 Harry Hacker 被打动并买了一台。
回家后他写了一个程序随机地读分布在磁盘上的 10 000 个块。令他惊讶的是, 测试结果与先到先服务方式得出的结果一样。商人是否在骗人?
30. 一个 UNIX 进程包含两部分: 用户部分和内核部分。内核部分类似于子例程还是共行例程 (coroutine)?
31. 一个电脑的时钟中断处理器每一时钟节拍要占用 2 ms(包括所有的进程切换开销), 时钟以 60 Hz 的频率运行, 那么 CPU 时间用于时钟处理的部分是多少?
32. 教材中给出了两个看门狗定时器的例子: 软盘马达的定时启动和允许在硬拷贝终端上使用回车。请给出第三个例子。
33. 为什么 RS-232 终端是中断驱动, 而内存映射终端不是中断驱动?
34. 考虑一个终端如何工作: 驱动程序输出一个字符, 然后阻塞。当字符被打印完毕后, 发生一条中断并且传送一条消息给阻塞的驱动程序, 它输出下一个字符然后再次阻塞。如果传送一条消息, 输出一个字符, 然后阻塞共需时间为 4 ms, 那么这一方法在波特率为 110 的传输线上能很好地工作吗? 在 4800 波特的传输线上呢?
35. 一个位图终端包含 1200×800 个像素。为了滚动一个窗口, CPU(或者控制器)必须移动上面的文本的所有行, 这通过将其所有比特从视频 RAM 的一个部分复制到另一个部分来实现。如果一个窗口高 66 行, 宽 80 个字符(共 5280 个字符), 每个字符宽 8 像素, 高 12 像素。假设将字符输出到屏幕需要 $50\ \mu s$, 如果以每字节 500 ns 的速率进行复制, 则滚动整个窗口需要多长时间? 如果每行都是 80 个字符, 那么终端的波特率是多少? 对于每像素占 4 比特的彩色终端, 计算其波特率(假设现在将一个字符输出到屏幕需要 $200\ \mu s$)。
36. 操作系统为什么提供转义字符, 例如 MINIX 中的 CTRL-V?
37. 在接收到一个 CTRL-C (SIGINT) 字符后, MINIX 驱动程序放弃了当前为终端排队的所有输出。为什么?
38. 许多 RS-232 终端具有这样的转义序列: 删除当前传输行并将其下边的所有行上移。你认为在终端内部如何实现这一特性?
39. 在最初 IBM PC 的彩色显示器上, 除了 CRT 电子束垂直回扫以外的任何时间向视频 RAM 中写数据都会导致屏幕上的“雪花”。一个屏幕为 25×80 字符, 每个字符占用 8×8 像素。每行的 640 像素在电子束的一次水平扫描中绘出。该过程历时 $63.6\ \mu s$, 屏幕每秒钟刷新 60 次,

每次刷新均需要一个垂直回扫以使电子束回到屏幕顶端。该过程中可供写视频RAM的时间占多少？

40. 为 IBM 彩色显示器或其他位图显示器写一个图形驱动程序，它应该接收如下命令：设置并清除单个像素、移动屏幕上的矩形以及你认为有趣的其他特性。用户程序与驱动程序的接口方式为打开 */dev/graphics* 文件并向其中写入命令。
41. 修改 MINIX 软盘驱动程序以完成每次一道高速缓冲。
42. 实现一个软盘驱动程序使其作为一个字符设备而非块设备来工作，以便旁路掉文件系统的块高速缓冲。在此方式下，用户可从磁盘读大量数据，这些数据通过 DMA 方式直接传到用户空间，以大幅度提高性能。该驱动程序主要针对那些不需要通过文件系统而是机械地从磁盘读取比特的程序，例如文件系统检查工具。
43. 实现 MINIX 中缺少的 UNIX PROFIL 系统调用。
44. 修改终端驱动程序，使得除了一个用来删除前一个字符的特殊键外，另外再有一个键能够删除前一个单词。
45. MINIX 3 系统中已经加入了一种支持可移动介质的新硬盘设备。该设备在更换介质时必须旋转加速，而且这个时间相当长。预期在系统运行过程中介质更换很频繁。这样 *at_wini.c* 文件中的 *waitfor* 例程便无法满足需要。请设计一个新的 *waitfor* 例程，若所等待的位组合在 1 s 的忙等待后仍未出现，则进入这样一个阶段：该阶段下磁盘任务将睡眠 1 s，测试端口，并回去继续睡眠 1 s，直到出现相应的位组合，或者到达预先设置的 *TIMEOUT* 超时。



第4章 存储管理

- 4.1 基本的存储管理
- 4.2 交换技术
- 4.3 虚拟存储管理
- 4.4 页面置换算法
- 4.5 页式存储管理中的设计问题
- 4.7 MINIX 3进程管理器概述
- 4.8 MINIX 3进程管理器的实现
- 4.9 小结

存储器是一种很重要的资源，必须仔细管理。随着科技的发展，存储器容量也越来越大。例如，与20世纪60年代初世界最大的计算机IBM 7094相比，现在一台普通家用计算机的存储器容量已是它的2000倍。但与此相对应，应用程序的规模也越来越大。用Parkinson定律来说，就是“存储器有多大，程序就会有多长”。在这一章中，我们将讨论操作系统是如何来管理存储器的。

从一个程序员的角度来说，他梦想拥有的存储器应该是容量无限大的，速度无限快的，而且是非易失型的（nonvolatile），即掉电后数据不会丢失。嗯，既然话已至此，干脆再加上一条：价格还要尽量便宜。遗憾的是，在目前这个阶段，同时满足这么多条件的存储器还仅仅是一个

梦想，仅凭现有的科技手段还无法让我们梦想成真。因此，绝大多数的计算机采用了一种折中的方法，也就是说，建立了一个存储器层次结构。在这个层次结构的最顶层，是CPU内部的一些寄存器，它们的访问速度是最快的，但容量不是很大，价格也比较昂贵。第二层是高速缓存（cache）。第三层是主存储器（内存），它的访问速度适中，价格也适中，容量一般在几百兆字节。以上这三种存储器都是易失型的，即在断电后，其内容全部会丢失。第四层是磁盘，它的访问速度较慢、价格较便宜，存储容量一般在几十或几百GB，而且是非易失型的。操作系统作为一个系统软件，其任务就是协调好这些不同类型的存储器的使用。

在操作系统中，负责管理这个存储器层次结构的那一部分程序，称为存储管理器（memory manager）。它的主要任务是：第一，记录存储器的使用情况，即哪些部分正被使用，哪些部分还空闲着。第二，当进程需要存储空间时，就分配给它；然后当它运行结束后，再把存储空间收回回来。第三，如果内存太小，容不下所有的进程，那么就需要把内存中暂时不能运行的进程送到磁盘上，然后再把磁盘上的另一个进程装入内存，这个交换的过程也要由存储管理器来管理。在大多数的操作系统中，存储管理器都位于内核之中（但MINIX 3除外）。

本章我们将讨论一些不同类型的存储管理方案，在这些方案中，既有非常简单的，也有非常复杂的。我们将按照从简单到复杂的顺序，循序渐进地来进行介绍。

就像我们在第1章中指出的那样，在计算机领域，历史总是在重演。早期的小型机软件有点像大型机软件，而早期的个人计算机软件又有点像小型机软件。时至今日，在掌上型电脑、个人数字助理（PDA）和嵌入式系统中，历史又在重演。在这些系统中，简单的存储管理方案仍然得到了广泛应用，因此，我们在这里讨论这些存储管理方案是有意义的。

4.1 基本的存储管理

存储管理系统可以分为两类，一类是需要在内存和磁盘之间，把进程换进换出的；另一类是不需要这种换进换出的。后者比较简单，因此我们将首先介绍。在本章稍后，我们将讨论进程的换进换出和页面置换。这里先请大家记住一点，进程的换进换出和页面置换都是由于内存不足造成的，

由于在内存中无法同时容纳所有的程序和数据,因此只能设计出各种策略来解决这个问题。如果内存的容量足够大、足够用,那么各种存储管理方案之间的差别也就无足轻重了。

另一方面,如上所述,软件规模的增长速度与存储容量的增长速度是差不多的,因此,有效的存储管理还是有用的。在20世纪80年代,许多大学的分时系统运行在4 MB的VAX机上,还带着许多的用户,而且这些用户对于能够使用这样的系统,或多或少还是满意的。而现在微软的单用户Windows XP系统,推荐的内存容量为128 MB。此外,目前的发展趋势是多媒体,这就需要更多的内存。因此,即便是在下一个十年,一个好的存储管理器还是很有必要的。

4.1.1 单道程序存储管理

单道程序存储管理是最简单的一种存储管理方法。它的基本思路是,把整个内存划分为两个区域,即系统区和用户区。然后每一次把一个应用程序装入到用户区去运行,由它和操作系统来共享整个内存。而且从装入开始一直到它运行结束,在这段时期内,该程序始终独占着整个用户区。在具体实现单道程序存储管理方案时,主要有三种实现方式,如图4.1所示。在图4.1(a)所示的方式中,操作系统被放在了随机存取存储器(Random Access Memory, RAM)的最低端;在图4.1(b)所示的方式中,操作系统被放在了内存地址的最高端,而且是放在了只读存储器(Read-Only Memory, ROM)里面。在图4.1(c)所示的方式中,操作系统被分成两部分,一部分是设备驱动程序,被放在内存高端的ROM中;另一部分则放在了内存地址的最低端。对于这三种实现方式,第一种主要用在早期的大型机和小型机中,现在已经很少使用了;第二种主要用在一些掌上型电脑和嵌入式系统中;第三种主要用在早期的个人计算机系统中,如MS-DOS。其中,存放在内存高端的ROM中的系统内容,称为基本输入输出系统(Basic Input Output System, BIOS)。

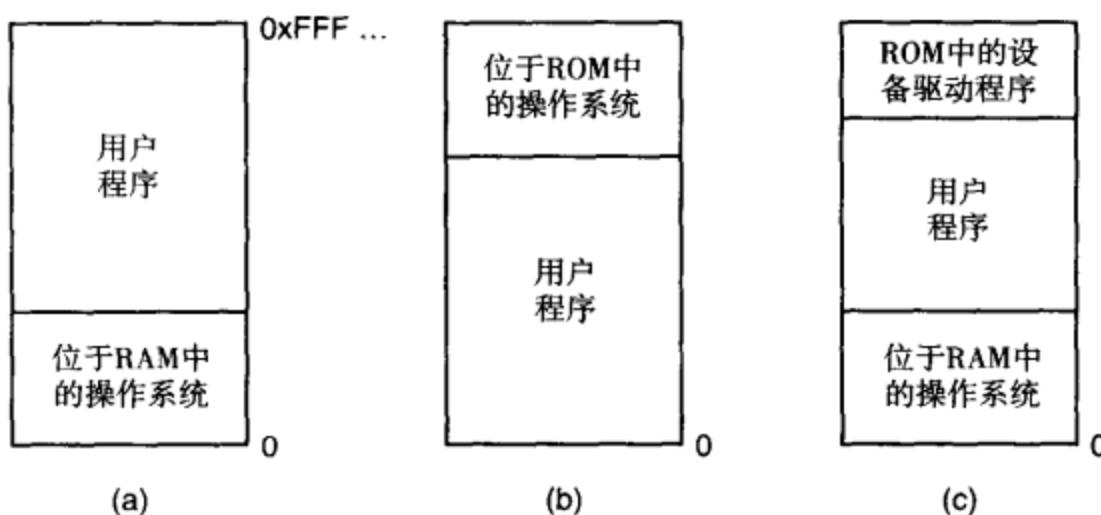


图4.1 单道程序存储管理的三种实现方案。其他方案也是存在的

在单道程序存储管理方式下,每一次只能运行一个程序。当用户输入一条命令后,操作系统就会把相应的程序从磁盘装入内存并运行之。当该进程运行结束后,操作系统就会显示一个提示符,等待用户输入新的命令。当操作系统收到新的命令后,就会把一个新的程序装入内存,从而把旧的那个程序覆盖掉。

4.1.2 固定分区的多道程序系统

在单道程序存储管理中每次只能运行一个进程,所以除了一些简单的嵌入式系统以外,已经没什么人再用它了。大多数现代系统都能够同时运行多个进程。所谓多个进程同时运行,就是说,当一个进程由于等待I/O操作被阻塞时,另一个进程可以去使用CPU。所以多道程序增加了CPU的利

用率。对于网络服务器来说，它总是能够同时运行多个进程（每个进程服务于不同的客户），但现在大多数的客户（如桌面）计算机都具备了这种能力。

为了实现多道程序技术，最简单的办法就是把内存划分为 n 个分区，每个分区的大小可以相等，也可以不等。例如，在系统启动的时候，由管理员来手工地划分出若干个分区。

当一个新进程到来时，需要根据它的大小，把它放置到相应的输入队列中去，然后等待合适的空闲分区。一方面，这个分区必须比进程要大，这样才能装得下它；另一方面，这个分区又必须尽可能地小，这样的话，在装入了该进程后，所浪费的内存资源就会尽可能地少（在本方案中，分区设置是固定的，在一个分区内未被进程使用的空间即被浪费掉了）。图4.2(a)显示了固定分区和多个输入队列的示意图。

多个输入队列方式存在一个很明显的缺点，也就是说，它可能会出现如下情形：小分区的输入队列是满的，而大分区的输入队列却是空的。例如，在图4.2(a)中，在分区1的输入队列中，有3个进程在等待，而分区3的输入队列却是空的。也就是说，一方面，有很多个小的进程在等着进入内存，而另一方面，在内存中却存在着大量的空闲空间。在本例中有300 KB的空闲空间，如果能把它平均分给这三个进程，那么它们就都能进入内存了。为了改变这种不合理的状态，人们提出了单个输入队列的方法，如图4.2(b)所示。也就是说，对于所有的用户分区，只设置一个统一的输入队列。当一个新的进程到来时，就把它加入到这个输入队列中。然后，当某个分区变得空闲时，可以采用两种办法来选择合适的进程。一种办法是选择离队首最近的、能够装入该分区的进程。但是这样的话，如果选中的进程是一个比较小的进程，那么就会浪费大量的内存空间。所以另外一种方法是先搜索整个队列，从中选择能够装入该分区的最大进程，从而尽可能地减少所浪费的空间。显然，这种算法不利于那些比较小的进程，因为这种算法会认为它们不值得拥有整个分区。但实际上，对于这些比较小的进程（通常是一些交互式进程），更应该给它们提供良好的服务，而不是反过来去歧视它们。

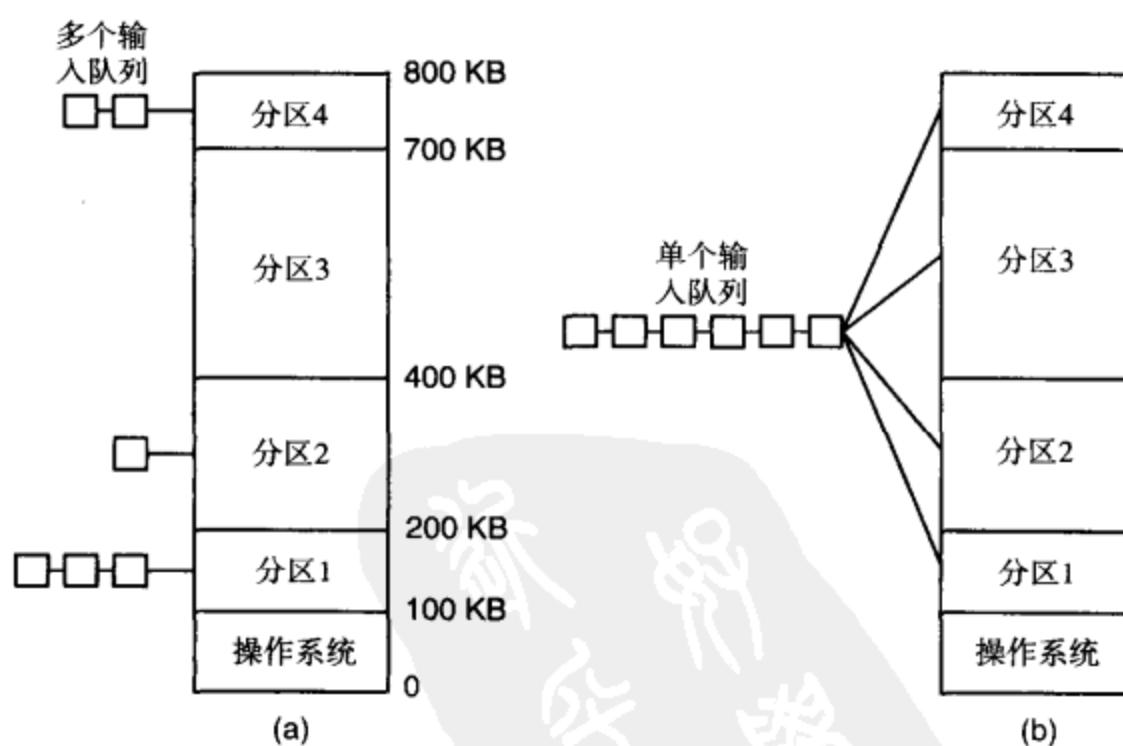


图 4.2 (a)各分区具有独立输入队列的固定内存分区；(b)仅有单个输入队列的固定内存分区

一种解决办法是始终保留至少一个小分区。这样，一些小的进程就可以直接运行，而无须去和其他进程竞争大块的内存分区。

另一种方法是制定一条规则，规定一个进程被忽略的次数不能超过 k 次。每当跳过一个进程时，该进程就得到1分。当它得到 k 分后，就不能再跳过它。

这种固定分区的方法，在IBM大型机的OS/360上使用了许多年，操作员早晨开机的时候，手工地划分出若干个分区，并确定每个分区的起始位置和大小等参数。然后，在系统的整个运行期间，这些参数就固定下来，不再改变。这套系统称为**固定数量任务的多道程序**（Multiprogramming with a Fixed number of Tasks, MFT）。它的优点是易于理解，也易于实现。当一个新进程到来时，把它放到输入队列中，然后等待合适的空闲分区。此时将进程装入该分区运行，直到其运行结束。不过，时至今日，很少有操作系统支持这种模型，即便是在大型机的批处理系统中。

4.1.3 重定位和存储保护

多道程序技术引发了两个很重要的问题：地址重定位和存储保护。如图4.2所示，不同的作业将在不同的地址区间运行。当一个程序被链接时（即把主程序、用户编写的函数和库函数组合到同一个地址空间中），链接器必须知道程序将在内存的什么地址开始运行。

例如，假设一个程序的第一条指令是去调用一个起始地址为100的函数，这个地址是链接器在生成二进制文件时分配的。如果程序被装入分区1（起始地址为100 KB），那么该指令在执行时，将跳转到绝对地址为100的地方，而这个地址存放的是操作系统的代码或数据，并不是它想要访问的那个函数。实际上，本次调用的目标地址应该是 $100\text{ KB} + 100$ 。换句话说，如果程序被装入分区2（起始地址为200 KB），那么函数调用的目标地址应该是 $200\text{ KB} + 100$ ，等等。像这样的问题，就是内存地址的**重定位**问题。

为了解决这个问题，一种方案是当一个程序被装入内存时，直接对指令代码进行修改，一次性地实现从文件内的相对地址到内存中的绝对地址的转换。例如，如果程序被装入分区1，那么就把它的每个地址加上100 KB；如果程序被装入分区2，那么就把每个地址加上200 KB，等等。为了在装入程序的时候执行这种操作，链接器必须在可执行文件中包含一个链表或位图，列出各个需要重定位的地址单元的位置，也就是说，需要告诉装载程序，哪些地方是需要修改的地址，哪些地方是不能修改的操作码、常量数据等。OS/MFT采用的就是这种方法。

在装入程序时进行地址重定位，这种方法并没有解决存储保护问题。由于系统中的程序使用的是绝对地址而不是相对于某个寄存器的相对地址，因此对于一个恶意的程序来说，它总是能生成一条指令，去访问内存中任何它想访问的地址。在多用户系统中，我们不希望一个进程去读写其他进程的内存空间。

IBM采用的保护360机器的办法是将内存划分为2 KB的块，并为每个块分配一个4位的保护码。另外，在CPU的程序状态字（Program Status Word, PSW）中包含一个4位的密钥。当一个进程在运行时，如果它访问的内存单元的保护码与PSW中的密钥不符，那么360的硬件就会引起一个陷入。由于只有操作系统才能修改保护码和密钥，因此这种办法能有效地阻止一个用户进程去破坏其他进程或操作系统的运行。

地址重定位和存储保护的另一种解决方案，就是在机器中增加两个特殊的硬件寄存器，即**基址**（base）寄存器和**边界**（limit）寄存器。当一个进程被调度执行时，就把该进程所在的分区的起始地址，放在基址寄存器中，并且把这个分区的长度，放在边界寄存器中。然后，在这个进程的运行过程中，当它需要访问内存单元的时候，硬件就会自动地把相应的内存地址加上基址寄存器的值，从而得到真正的目标地址。因此，如果基址寄存器的值为100 KB，那么指令CALL 100的执行结果就是 $CALL 100\text{ KB} + 100$ 。这样，对于指令本身来说，就不用做任何修改。在存储保护方面，对于每一次的内存访问，都要把该地址与边界寄存器的值进行比较，以防止它去访问分区以外的内存区域。当然，在这种方案中，一个前提条件就是必须用硬件把基址寄存器和边界寄存器保护起来，不能让用户程序随便去修改它们，否则，这种方法就失去了意义。

这种方法的一个缺点是，在每一次内存访问中，都必须增加一次加法和比较操作。比较操作的速度比较快，但加法操作的速度比较慢（需要进位传播）——除非使用特殊的加法电路。

CDC 6600（世界上第一台巨型机）使用了这种方案。早期的 IBM PC 中的 Intel 8088 CPU 使用了这种方案的一个较弱版本——有基地址寄存器，但没有边界寄存器。时至今日，已经没有计算机再采用这种方案了。

4.2 交换技术

在批处理系统中，采用固定分区的存储管理方案是简单而高效的。当一个作业需要运行时，先是在输入队列中等待，等到有了一个足够大的空闲分区后，就把它装入该分区去运行。在作业运行期间，它始终位于内存中，直到运行结束。只要在内存中能够有足够的作业，使 CPU 始终保持繁忙的状态，那么就没有理由去使用某种更为复杂的存储管理方案。

但在分时系统或面向图形的个人计算机中情形就不同了，有时会出现内存不够用的情形，无法同时容纳所有当前活动的进程。此时，就必须把多出来的进程暂时保存在磁盘上，并在需要的时候把它动态地调入内存。

在这种情形下，根据硬件条件的不同，可以采用两种不同的存储管理方法。最简单的策略称为 **交换** (swapping) 技术，它把各个进程完整地调入内存，运行一段时间，然后再放回到磁盘上；另一种策略称为 **虚拟存储器** (virtual memory)，在这种方式下，进程即使只有一部分内容存放在内存中，它也能运行。下面我们将讨论交换技术；在 4.3 节中我们将讨论虚拟存储器。

交换技术的基本原理如图 4.3 所示。开始时只有进程 A 在内存，随后进程 B 和 C 被创建，或者是从磁盘上换入。在图 4.3(d) 中，A 被交换到磁盘上，然后 D 进入，接着 B 离开，最后 A 被重新调入。由于进程 A 现在是在不同的位置，它所包含的地址都必须重定位，或者是在它换入时由软件来完成，或者是在程序运行时由硬件来完成。

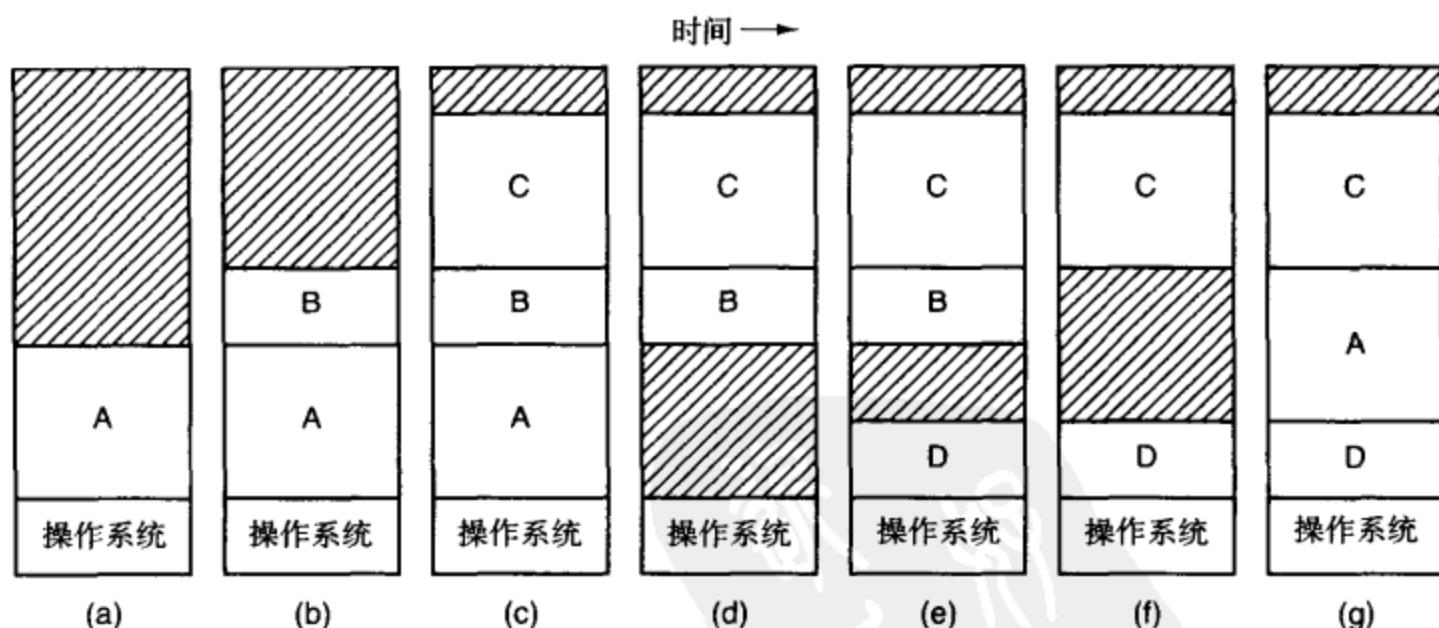


图 4.3 当进程进出内存时，内存分配的变化情况。阴影区域表示未使用的内存

图 4.2 所示的固定分区与图 4.3 所示的可变分区的主要区别是：在固定分区中，分区的个数、位置和大小都是固定不变的；而在可变分区中，这些参数都是随着进程的进进出出而动态变化的。由于分区的大小和个数不再固定，而是根据进程的实际需要来划分的，因此这种方式提高了内存的利用率，但同时也使内存的分配、回收和管理变得更加复杂。

随着进程的换入换出，在内存中会造成一些不连续的黑洞（即比较小的空闲分区），这时可以采用**内存紧缩**（memory compaction）技术，也就是说，把所有的进程都尽可能地往内存地址的低端移动，相应地，那些空闲的小分区就会往地址的高端移动，从而在地址高端形成一个较大的空闲分区。不过，我们通常不进行这种操作，因为它需要耗费大量的CPU时间。例如在一个有1GB内存、每秒可以复制2GB的计算机上，把全部内存紧缩一次需要0.5s。这看起来好像不是很多，但是对于一个正在观看视频影像的用户来说，还是能感觉到明显的停顿。

那么当一个进程被创建或换入时，应该给它分配多大的内存空间呢？如果进程的大小是固定不变的，那么分配方法很简单：进程需要多少，操作系统就分配多少。

然而，如果进程的数据段是可以增长的（例如，许多程序设计语言都支持动态内存分配），那么当一个进程试图增长时，问题就出现了。如果该进程邻接一个空洞，那么就可以把这个空洞分配给它；但如果进程邻接的是另一个进程，那么在这种情形下，或者是把需要扩展的进程移动到另一个足够大的空闲区中；或者是把一个或多个进程交换出去以生成一个足够大的空闲区。如果这两种方式都行不通，那么该进程只好等待或被杀死。

如果预计大多数进程在运行时都要增长，那么可以在进程被换入或移动时为它分配一点额外的内存，从而减少进程的移动或换入换出的次数，减少系统的开销。当然，在把一个进程换出到磁盘上时，应该只交换进程实际占用的内存内容，而没有必要换出额外的内存空间，因为这是无意义的，纯粹是浪费时间。在图4.4(a)中我们可以看到有两个进程，系统为它们分配了额外的空间以便于增长。

如果进程有两个可增长的段，例如，一个是数据段，作为堆空间来使用，用于存放动态分配和释放的变量；另一个是栈段，用来存放普通的局部变量和返回地址，那么可以使用另一种安排，如图4.4(b)所示。在图中我们可以看到，每个进程的栈段位于内存地址的顶端并向下增长，而位于代码段上面的数据段其增长方向是向上的。栈段和数据段之间的内存区间，既可以用于栈，也可以用于数据段。如果这段内存区间用完了，那么对于该进程而言，它或者被移动到足够大的空闲区中；或者被交换出内存，直到出现一个足够大的空闲区；或者被杀死。

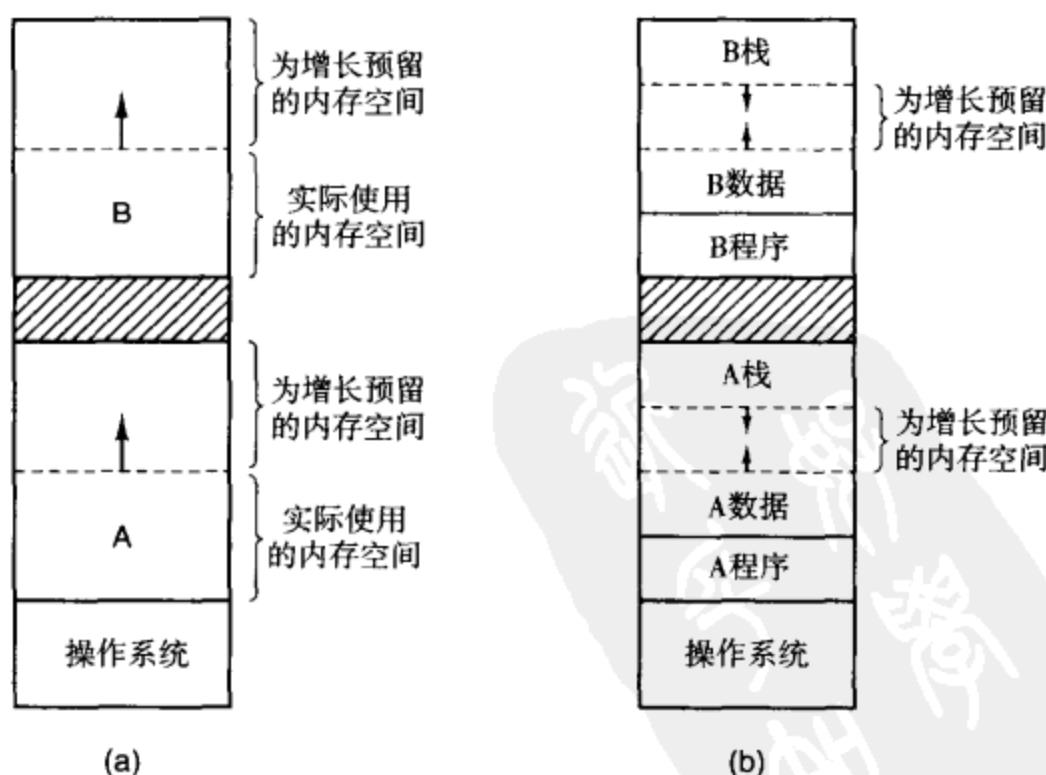


图4.4 (a)为能够增长的数据段预留空间；(b)为能够增长的数据段和栈段预留空间

4.2.1 基于位图的存储管理

如果内存是动态分配的，那么操作系统就必须管理好它。一般来说，主要有两种方法来记录内存的使用状况：位图法和空闲链表法。在这一小节和下一小节中，我们将分别对这两种方法进行讨论。

在位图法中，内存被划分为很多个分配单元，每个单元可能特别小，只有几个字；也可能特别大，包含几千个字节。每个分配单元都对应于位图中的某个数据位，0表示该单元是空闲的，1表示该单元已经被占用。图 4.5 显示了部分的内存和相应的位图。

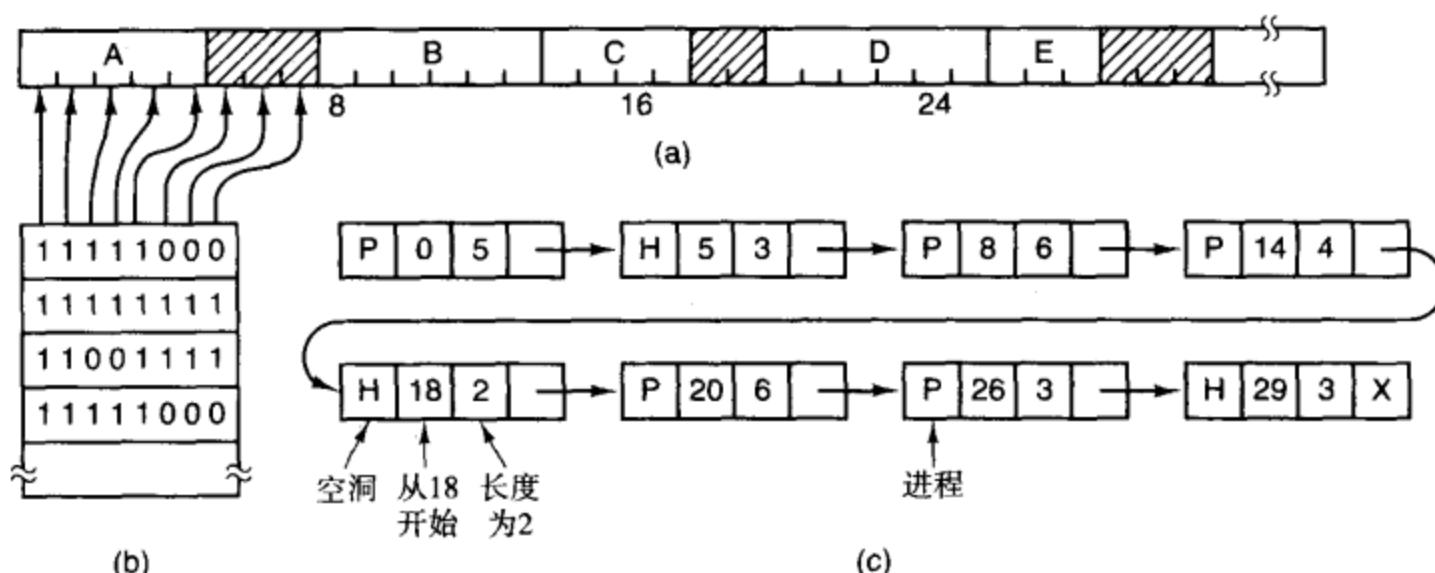


图 4.5 (a)一段内存，有 5 个进程和 3 个空洞，刻度表示内存的分配单元，阴影表示空闲分区(在位图中为 0); (b)对应的位图; (c)用链表来表示相同的信息

分配单元的大小是一个很重要的设计问题。分配单元越小，位图越大。不过，即便是 4 个字节来作为一个分配单元，32 位的内存也需要位图中的一位。而 $32n$ 位内存将使用第 n 个映射位，因此位图将只占用 $1/33$ 的内存。如果分配单元比较大，那么位图比较小，但如果进程的大小并非分配单元的整数倍，那么在最后一个存储单元中会有空间被浪费掉。

位图法为存储空间的管理提供了一种简单的方法，其大小仅取决于内存和分配单元的大小。它的缺点主要是，当决定把一个大小为 k 个分配单元的进程调入内存时，内存管理器必须搜索位图以找出一串 k 个连续的 0。而这种操作是比较慢的（因为一个串可能会跨越多个字的边界）。这是反对使用位图法的一个理由。

4.2.2 基于链表的存储管理

内存管理的另一种方法是建立一个链表，来描述已分配和空闲的内存分区。对于每一个分区，它可能存放了某个进程，也可能是两个进程之间的空闲区。例如，假设当系统运行到某个时刻，内存的使用情况如图 4.5(a)所示，那么相应的链表为图 4.5(c)。链表中的每一个结点，分别描述了一个内存分区，包括它的起始地址、长度、指向下一个结点的指针以及分区的当前状态，H 表示它是空闲区 (Hole)，P 表示它已经被某个进程 (Process) 占用了。

在这个例子中，链表是按照内存地址从低到高排序的。这样做好处是当一个进程运行结束或被置换出去时，可以很方便地来更新链表。对于一个即将运行结束的进程，它一般有两个邻居（除非它位于内存的最低端或最高端），这两个邻居可能是进程也可能是空闲区，这样就有四种组合，如图 4.6 所示。在图 4.6(a)中，进程 X 的左邻和右舍，都是一个进程，即 A 和 B，它们各自占据着一个分区在运行。在这种情形下，当进程 X 运行结束、释放它所占用的分区后，只要把它的状态修改

为空闲即可，即把它从 P 变为 H。图 4.6(b)和图 4.6(c)是类似的，在进程 X 的左边或右边有一个进程，但在它的另一侧是一个空闲的分区。在这种情形下，当进程 X 运行结束后，需要把它所占用的内存分区与右边或左边的空闲分区进行合并，形成一块更大的空闲分区。与此相对应，在链表中需要把两个链表结点合二为一。在图 4.6(d)中，进程 X 的左右两边都是空闲分区，在这种情形下，当进程 X 运行结束后，需要把这三块空闲分区进行合并。另外，为了便于链表结点的合并，可以把链表设计为双向链表，这样从当前结点就可以很方便地访问它的上一个结点。

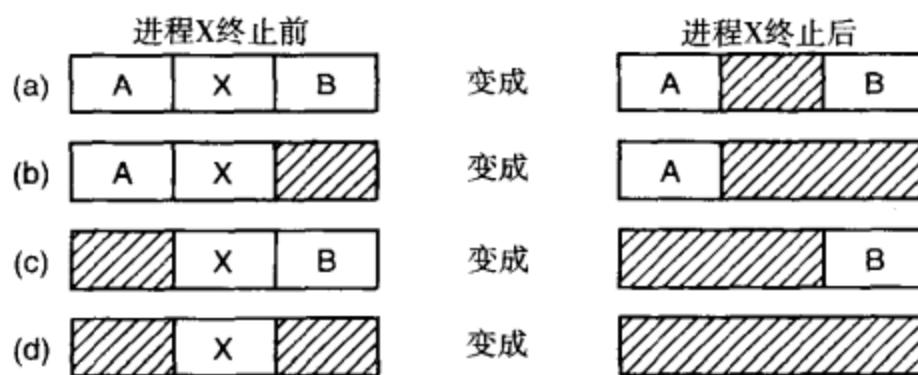


图 4.6 进程 X 终止时与左右邻居合并的 4 种方式

在基于链表的存储管理中，当一个新的进程来到时（或者一个已经存在的进程被置换进内存时），需要为它分配内存空间，即为它寻找某个空闲分区，该分区的大小必须大于或等于进程的大小。通常的分配算法有最先匹配法、下次匹配法、最佳匹配法、最坏匹配法和快速匹配法。

最先匹配法 (first fit)是最简单的一种算法，它的基本思路是：假设新进程的大小为 M，那么从链表的首结点开始，将每一个“空闲”结点的长度与 M 进行比较，看看是否大于或等于它，直到找到第一个符合要求的结点。然后把它所对应的空闲分区分割为两个小的分区，一个用来装入这个进程，另一个是剩余的空闲区域。与之相对应，链表结点也要一分为二，分裂成两个结点，并修改相应的内容，包括分区的状态、起始地址和长度。这种算法查找的结点很少，因而速度很快。

下次匹配法 (next fit)与最先匹配法的思路是类似的，只不过每一次当它找到一个合适的空闲结点时，就会把当前位置记录下来。然后等下一次又有一个新进程到来时，就从这个位置开始继续往下找，而不是像最先匹配法那样，每次都是从链表的首结点开始找。当然，如果现在已经走到了链表的结尾，就要再回到开头继续查找。这种算法查找的结点也很少，因此速度也很快。但它的缺点是，较大的空闲分区不容易保留，从模拟的结果来看 (Bays, 1977)，它的性能要略逊于最先匹配法。

最佳匹配法 (best fit)的基本思路是：搜索整个链表，将能够装得下该进程的最小空闲分区分配出去。

我们来看一个例子，在图 4.5 中，如果新进程的大小为 2，那么最先匹配法将分配起始地址为 5 的那个空闲分区，而最佳匹配法将分配起始地址为 18 的空闲分区。

最佳匹配法要比最先匹配法慢，因为它必须搜索整个链表。而且出人意料的是，这种算法在性能上比最先匹配法和下次匹配法还要差，浪费了更多的内存。原因在于：由于每次选择的都是与进程大小最接近的空闲分区，因此在分割以后剩余的空闲区域将会很小，甚至无法使用，从而造成浪费。

为了克服最佳匹配算法把空闲区切割得太小的缺点，人们又提出了一种**最坏匹配法 (worst fit)**。也就是说，在每次分配的时候，总是将最大的那个空闲区切去一部分，分配给请求者。它的依据是当一个很大的空闲区被切割了一部分以后，可能仍然是一个比较大的空闲区，从而避免了空闲区越分越小的问题。这种算法基本上不会留下小的空闲分区，但是那些较大的空闲分区也没有保留下来，

这样，如果有一个大的进程到来，就有可能找不到合适的空闲分区。模拟结果表明，这种算法的性能也不是很理想。

对于以上这四种算法，可以通过将进程链表和空闲链表分离的方法，来加快链表的查找速度。这样，当这些算法在分配内存空间时，只需去查找空闲链表即可，而不必考虑进程链表。不过，这种做法虽然加速了分配过程，但却使内存的回收过程变得更加复杂，速度更慢。如果一个进程所占用的内存分区被释放，那么首先要把它从进程链表中摘下来，然后插入到空闲链表中。

如果进程链表和空闲链表是分离的，那么我们可以把空闲链表按照从小到大的顺序进行排序，以提高最佳匹配法的速度。这样，当最佳匹配法搜索到第一个合适的空闲分区时，它就知道这个空闲区是能容纳该进程的最小空闲分区，因此是最佳的。而后面的链表结点就没有必要再去搜索了。另外，在这种排序方式下，最先匹配法和最佳匹配法的速度一样快，而下次匹配法则变得毫无意义。

另外，我们还可以对空闲链表进行一点优化。也就是说，我们可以使用空闲分区本身来存储链表信息，而不必像图 4.5(c)那样另外再使用一组数据结构来构造链表。具体来说，每个空闲分区的第一个字可以用来存放空闲区的大小，第二个字可以用来作为指向下一个结点的指针。这样一来，图 4.5(c)中的那些链表结点（每个结点需要 3 个字和 1 个状态位 P/H），就不再需要了。

最后一种算法叫快速匹配法 (quick fit)，它的基本思路是：对于一些常用的请求大小，为它们分别设置各自的链表。例如，在这种算法中，可能会有一个长度为 n 的表，第一个表项是一个指针，指向一个空闲链表的首结点，其中每个结点描述的是大小为 4 KB 的空闲分区。第二个表项指向的是长度为 8 KB 的空闲链表，第三个表项指向的是长度为 12 KB 的空闲链表，等等。对于 21 KB 这样的空闲分区，它既可以放在 20 KB 的链表中，也可以放在一个专门用来存放特殊大小的空闲链表中。在快速匹配法中，寻找指定大小的空闲区是非常快的，但就像所有按空闲区大小排序的算法那样，它的缺点是：当一个进程运行结束或被置换出内存时，如果我们要去查看它的左邻右舍，看看能否进行合并，那么这个操作将非常费时且开销很大。但如果不行并，内存很快就会分裂成大量的小空洞，成为进程无法使用的碎片。

4.3 虚拟存储管理

如果程序太大了，超过了空闲内存的容量，没有办法把它全部装入到内存，这时应该怎么办？在很多年前，当人们首次碰上这个问题时，通常采用的解决方案是覆盖技术。也就是说，把该程序划分为若干个部分，称为覆盖块 (overlay)，然后只把那些当前需要用到的指令和数据保存在内存中，而把其余的指令和数据保存在外存中。例如，覆盖块 0 首先运行，当它运行结束时再调用另一个覆盖块去运行。有一些覆盖系统比较复杂，允许多个覆盖块同时在内存中。对于存放在磁盘上的覆盖块，当需要用到它们时会由操作系统动态地换入内存。

虽然覆盖块的交换操作是由操作系统来完成的，但必须由程序员来手工地把一个大的程序划分为若干个小的功能模块，并确定各个模块之间的调用关系。这种事情既费时又费力，增加了编程的复杂度。不久就有人找到了一种办法，可以把全部的工作都交给计算机来完成。

这种方法就是虚拟存储器 (virtual memory) (Fotheringham, 1961)。它的基本思路是：程序的代码、数据和栈的总大小可以超过实际可用的物理内存的大小。操作系统把当前需要用到的那些部分保留在内存中，而把其余部分保存在磁盘上。例如，一个大小为 512 MB 的程序，可以运行在一台内存空间只有 256 MB 的机器上。采用的方法就是在进程运行的每个时刻，经过精心挑选，只保留一半的程序代码和数据在内存中，然后在需要的时候，再把各个程序片段在内存和磁盘之间来回交换。

虚拟存储器也可以运行在多道程序系统中, 将许多程序的零零碎碎的片段同时存放在内存中。如果一个程序在运行时需要把存放在磁盘上的另一部分代码调入内存, 这时它就会启动一次I/O操作。而当它正在等待I/O操作完成时, CPU就可以空闲下来, 交给另外一个进程去使用。

4.3.1 虚拟页式存储管理

大部分虚拟存储系统采用的是一种称为分页(paging)的技术。在一台计算机上, 程序员在编写程序时, 将使用一组内存地址。例如, 在程序中可能有如下的一条指令:

```
MOVE REG, 1000
```

这条指令的功能是把地址为1000的内存单元的内容复制到REG寄存器中(或者反过来, 这取决于不同的计算机型号)。该地址可以通过索引、基地址寄存器、段寄存器和其他方式产生。

这些由程序产生的地址称为虚拟地址(virtual address), 它们构成了一个虚拟地址空间(virtual address space)。如果计算机没有使用虚拟存储机制, 那么虚拟地址就是最终的物理地址, 它被直接放在地址总线上, 从而可以对相应地址的内存单元进行读写操作。如果计算机使用了虚拟存储机制, 那么虚拟地址不是被直接放在地址总线上, 而是被送到存储管理单元(Memory Management Unit, MMU)中, 由它负责把虚拟地址映射为物理地址, 如图4.7所示。

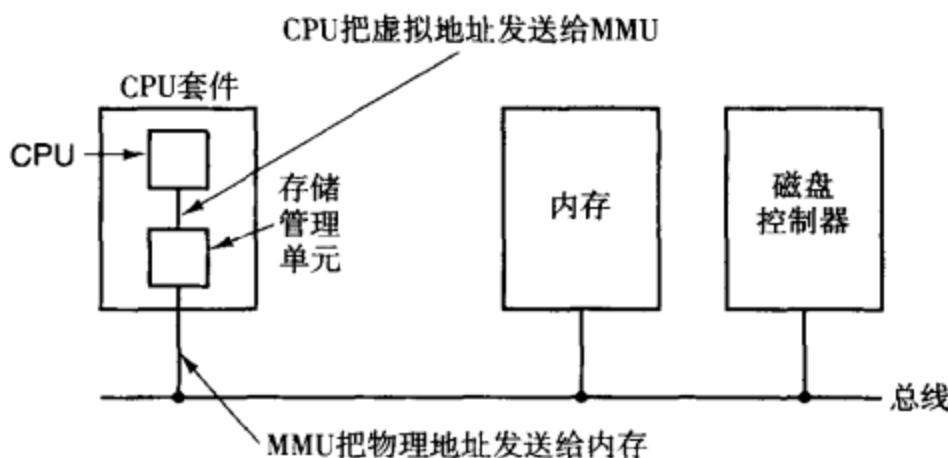


图4.7 MMU的位置和功能。当今的MMU一般集成在CPU芯片内部, 但从逻辑上讲, 它可以是单独的一个芯片

图4.8是一个非常简单的例子, 它演示了地址映射的过程。在这个例子中, 计算机的地址为16位, 从0一直到64 KB, 这些都是虚拟地址。也就是说, 虚拟地址空间的大小是64 KB。而对于实际的物理内存来说, 它的大小只有32 KB。因此, 当我们在编写程序的时候, 虽然程序的大小可以达到64 KB, 但这种程序无法完全装入内存去运行。因此, 我们必须在磁盘上保存该程序的内存映像(最多可达64 KB), 然后在需要的时候再把不同的程序片断装入内存。

在虚拟页式存储管理中, 一方面, 把物理内存划分为许多个固定大小的内存块, 称为物理页面, 或者是页框(page frame)。另一方面, 把虚拟地址空间也划分为大小相同的块, 称为虚拟页面, 或者简称为页面(page)。页面的大小要求是2的整数次幂, 一般在512个字节到1 M字节之间。在本例中, 页面的大小是4 KB。由于虚拟地址空间是64 KB, 物理内存是32 KB, 因此我们将得到16个虚拟页面和8个物理页面。当一个用户程序需要在内存和磁盘之间换入换出时, 不是以整个程序为单位, 而是以页面为单位。

假设程序在运行时, 需要去访问地址0。例如, 执行如下的指令:

```
MOV REG, 0
```

虚拟地址 0 被发送给 MMU，然后 MMU 就会发现这个虚拟地址位于第 0 个页面中（第 0 个页面所包含的地址为 0~4095），而第 0 个虚拟页面存放在内存的第 2 个物理页面中（8192~12 287），它的起始地址是 8192，然后再加上页内的偏移地址 0，所以最后得到的物理地址就是 8192。然后 MMU 就会把这个真正的物理地址发送到地址总线上，从而去访问相应的内存单元。这样，这一次的内存访问就顺利地完成了。事实上，第 0 个虚拟页面中的每一个地址（0~4095）都能映射到相应的物理地址（8192~12 287）。

类似地，如果去执行另一条指令

```
MOV REG, 8192
```

那么该指令将被转换为

```
MOV REG, 24576
```

这是因为虚拟地址 8192 位于第 2 个虚拟页面中，而该页面存放在第 6 个物理页面中（相应的物理地址为 24 576~28 671）。作为第三个例子，虚拟地址 20 500 位于第 5 个虚拟页面中（虚拟地址为 20 480~24 575），页面内的偏移地址是 20。而第 5 个虚拟页面存放在第 3 个物理页面中，它的起始地址为 12 288，所以最后的物理地址为 $12\ 288 + 20 = 12\ 308$ 。

经过适当的设置后，MMU 能够将 16 个虚拟页面映射到相应的 8 个物理页面之一，但这并没有解决虚拟地址空间比物理内存大的问题。由于我们只有 8 个物理页面，因此，在图 4.8 中，只有 8 个虚拟页面能够被映射到物理内存。而其他的虚拟页面，则无法被映射，在图中用 X 来表示。在实际的硬件上，会有一个有效位（present/absent bit）来描述每个虚拟页面是否在内存中。

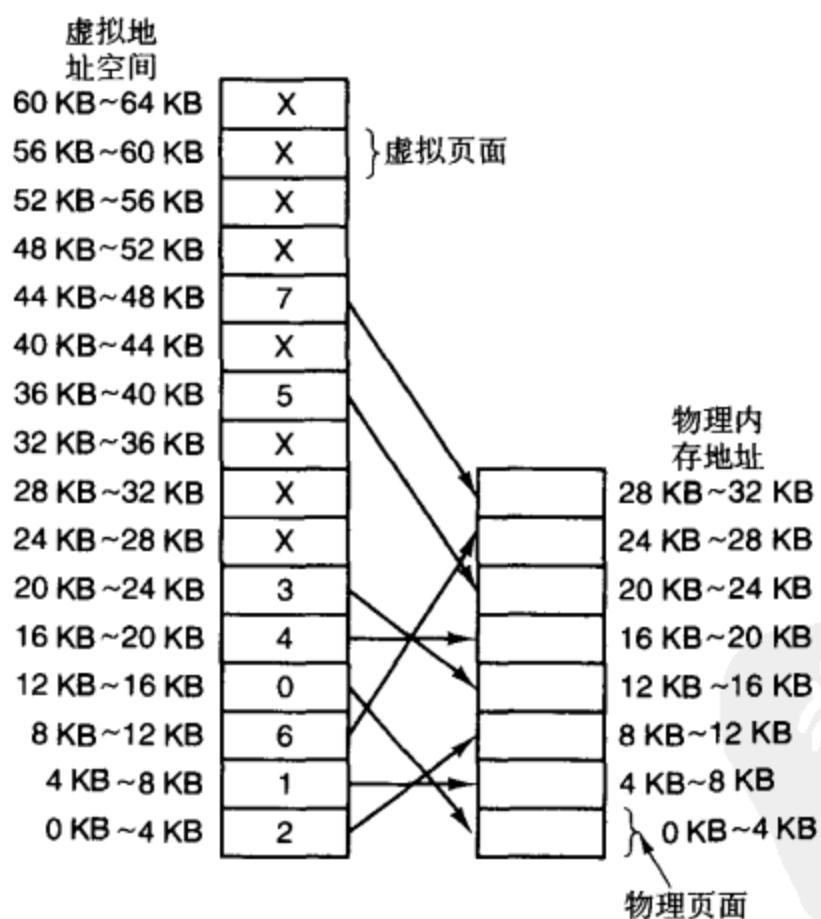


图 4.8 虚拟地址与物理内存地址之间的映射关系存放在页表中

如果程序访问了一个未被映射的页面，例如，它执行了如下的指令：

```
MOV REG, 32780
```

也就是说，对虚拟地址 32 768 进行访问（这个虚拟地址位于第 8 个虚拟页面中，而且页内偏移地址是 12），那么 MMU 在进行地址映射的时候就会发现，第 8 个虚拟页面并未在内存中（在图中用 X

标识），因此无法完成此次地址映射。这时，MMU 就会引发一个缺页中断（page fault），把这个问题提交给操作系统去处理。操作系统将从内存中挑选一个使用不多的物理页面，把它的内容写回到磁盘，从而腾出了一个空闲页面，然后把引发缺页中断的那个虚拟页面装入该空闲页面中，并对地址映射关系进行更新。最后，回到被中断的指令重新开始执行。

例如，假设操作系统选中了物理页面 1，把它的内容保存在磁盘上，然后把第 8 个虚拟页面装入到物理页面 1（起始地址为 4 KB）。接下来，要对 MMU 的地址映射进行两处修改：一是将虚拟页面 1 的表项设置为未映射，这样，如果将来有程序需要去访问虚拟页面 1 中的地址，就会引发缺页中断；二是将虚拟页面 8 中的表项设置为 1，这样，当重新执行被中断的指令时，就会把虚拟地址 32 780 映射到相应的物理地址 4108。

下面我们来看一下 MMU 的内部结构，了解一下它的工作原理，以及为什么页面的大小必须是 2 的整数次幂。在图 4.9 中，我们可以看到一个虚拟地址 8196（它的二进制形式是 001000000000100）。在进行地址映射时，采用的是图 4.8 中的地址映射关系（页表）。这个 16 位的虚拟地址被划分为两部分：4 位的页号和 12 位的偏移量。4 位的页号可以表示 16 个页面，12 位的偏移量可以寻址 4096 个字节。

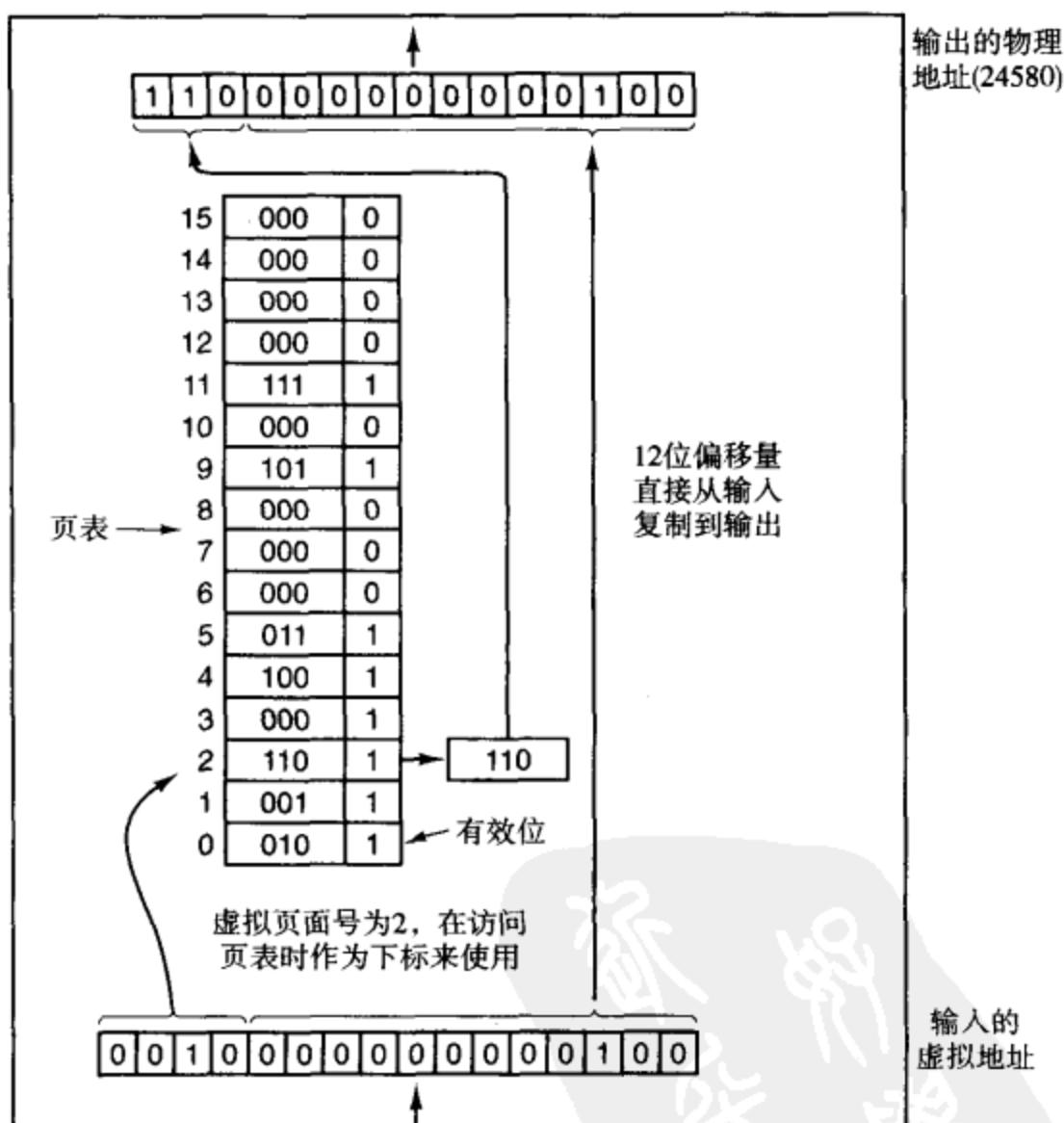


图 4.9 MMU 的内部操作（带有 16 个 4 KB 的虚拟页面）

在进行地址映射时，使用虚拟页面号作为索引去访问页表（page table），从而得到相应的物理页面号。如果有效位的值为 0，则将引发一个缺页中断，陷入到操作系统；如果有效位的值为 1，则将页表中查到的物理页面号复制到输出寄存器的高三位中，再加上输入的虚拟地址中的 12 位偏移量，就构成了 15 位的物理地址。输出寄存器的内容随即被作为物理地址送到内存总线。

4.3.2 页表

从理论上来说，虚拟地址到物理地址的映射就像我们刚才所描述的那样，虚拟地址被分成虚拟页面号（高位）和偏移量（低位）两部分。例如，假设地址为16位，页面大小为4 KB，那么高4位用于指定虚拟页面号（总共有16个虚拟页面），低12位用来作为页内偏移量（0~4095）。当然，虚拟页面号也可以是3位、5位或其他的位数，不同的划分表示不同的页面大小。

在地址映射时，使用虚拟页面号作为索引去访问页表，从而得到相应的物理页面号。然后用物理页面号来取代虚拟页面号，与虚拟地址中的偏移量进行组合，从而得到最终的物理地址。

页表的用途就是将虚拟页面映射为相应的物理页面。从数学的角度来说，页表是一个函数，它的输入是虚拟页面号，输出是物理页面号。通过这个函数可以把虚拟地址中的虚拟页面号替换成物理页面号，从而形成物理地址。

在页表的具体实现上，除了以上的简单描述之外，还要考虑两个重要的问题：

1. 页表可能会非常大。
2. 地址映射必须十分迅速。

第一个问题是因为在现代计算机中普遍使用了32位的虚拟地址，即虚拟地址空间为4 GB。假设页面的大小为4 KB，那么总共有 $4\text{ GB}/4\text{ KB} = 100\text{ 万个页面}$ ，因此页表必须有100万个表项。此外，由于每个进程都有自己独立的虚拟地址空间，因此每个进程都有它自己的页表。这样，页表的数量和规模就非常庞大。

第二个问题是因为在每一次内存访问时，都必须进行虚拟地址到物理地址的映射。一条典型的指令通常包括一个操作字和一个内存运算符。因此在每一条指令的执行过程中，都要去访问页表一次、两次甚至多次。假设执行一条指令需要1 ns，则页表的查找时间必须在250 ps内完成，以避免成为系统的主要瓶颈。

在设计和实现计算机时，必须考虑到这种大而快速的页面映射需求。一般来说，这个问题在高档的计算机中比较突出，但对于强调性能/价格比的低档机来说，它也是一个重要的问题。在本节和以下几节中，我们将详细讨论页表的设计技术和一些已经实用化的解决方案。

最简单的一种设计方案（至少在理论上）是使用一个页表，它由一组快速的硬件寄存器组成。每一个虚拟页面对应于一个表项，并用虚拟页面号作为索引，如图4.9所示。在启动一个进程时，操作系统把位于内存中的页表装入寄存器。这样，在进程运行期间就不用因为页表而访问内存。这种方法的优点是直观，而且在地址映射时不需要访问内存。缺点是价格比较昂贵（如果页表很大）。另外，在每次进程切换时都要装入位于内存中的页表，这也会降低系统的性能。

另外一个极端是把页表全部放在内存中。这时，在硬件上仅仅需要一个指向页表起始地址的寄存器。在进程切换时，只要重新装入这个寄存器的内容即可。这种方法的一个严重缺点是，在执行每一条指令时，都需要访问一次或多次内存，以读取所需的页表项。因此这种方法很少以它最单纯的方式使用，下面我们将讨论它的一些改进方案，它们具有更好的性能。

多级页表

多级页表的基本思路是，虽然进程的虚拟地址空间很大，但是当进程在运行时，并不会用到所有的虚拟地址，所以没有必要把所有的页表项都保存在内存中。图4.10是一个二级页表的简单例子。在图4.10(a)中，假设虚拟地址为32位，页面的大小为4 KB。如果是通常的页式存储管理，将把虚拟地址划分为两部分：虚拟页面号20位，页内偏移地址12位。但是在二级页表的结构下，还需要把虚拟页面号再进一步地划分为两部分：10位的字段PT1和10位的字段PT2。

多级页表的秘诀在于避免把所有的页表一直保存在内存中，尤其是那些暂时用不上的页表。例如，假设一个进程需要 12 MB 的内存空间，最底端的 4 MB 用于程序代码，接下来的 4 MB 用于数据，最顶端的 4 MB 用做栈。而对于数据段以上、栈以下的一大块空间，都是没有用到的地址空间。

在图 4.10(b) 中，我们可以看到相应的二级页表方案。在图的左边是第一级页表，有 1024 个页表项，对应于 10 位的 PT1 字段。当一个虚拟地址发送给 MMU 时，它首先把 PT1 字段抽取出来，并用它作为索引，去访问第一级页表。这 1024 个页表项中的每一个都描述了 4 MB 的地址空间，因为总的虚拟地址空间大小是 4 GB（即 32 位），它被分为 1024 块，因此每一块的大小是 4 MB。

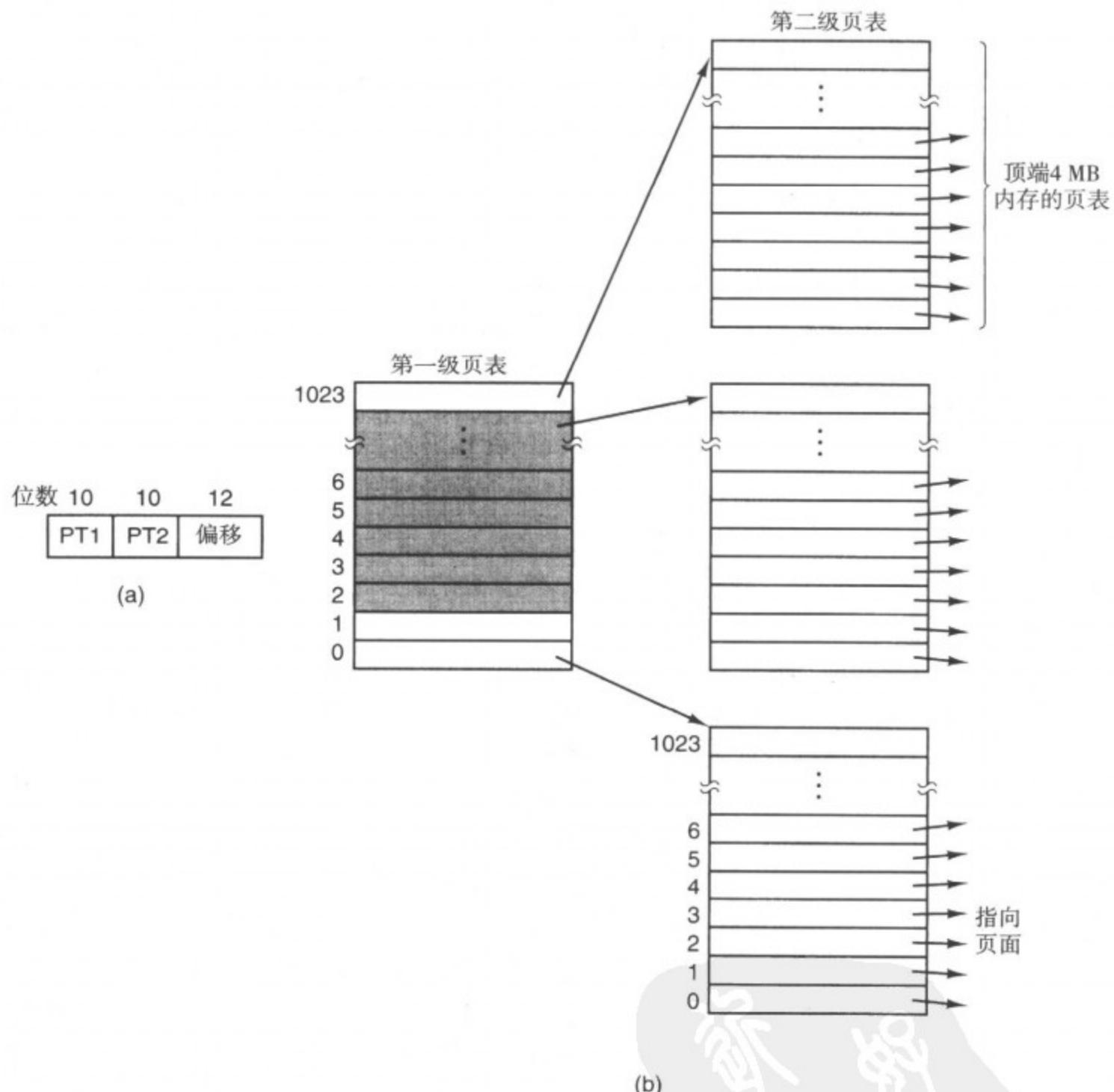


图 4.10 (a)包含有两个页表字段的一个 32 位地址；(b)二级页表

从第一级页表中，可以取出一个第二级页表所在的物理页面号。第一级页表的表项 0 存放的是程序代码的页表的起始地址，表项 1 存放的是数据段的页表的起始地址，表项 1023 存放的是栈的页表的起始地址，其他的表项（阴影部分）未被使用。当找到了所需的第二级页表之后，就可以使用 PT2 字段来作为索引，去访问相应的页表项，从而找到最终的物理页面号。

例如，假设有一个 32 位的虚拟地址 0x00403004（十进制的 4 206 596），该地址位于数据段内的 12 292 字节处。首先，将该地址除以 4 MB，结果为 1，余数为 12 292，因此 PT1 字段为 1。再

将 12 292 除以 4 KB，结果为 3，余数为 4，因此 PT2 字段为 3，页内偏移量为 4。然后，MMU 将使用 PT1 来访问第一级页表的表项 1，它对应于虚拟地址 4 MB~8 MB。在找到相应的第二级页表后，再使用 PT2 来访问它的表项 3，这对应于这 4 MB 空间内的地址 12 288~16 383（也就是绝对地址 4 206 592~4 210 687）。在这个页表项中，存放的就是最终的物理页面号，它包含有虚拟地址 0x00403004。如果该页面未在内存中，则页表项中的有效位的值为 0，此时将触发一次缺页中断；如果该页面正在内存中，那么就把得到的物理页面号与页内偏移地址 4 进行组合，得到最终的物理地址。这个地址将被放在地址总线上，并发送给物理内存。

在图 4.10 中，值得注意的是，虽然地址空间包含了 100 万个页面，但实际上只需要 4 个页表：一个第一级页表和三个第二级页表（0~4 MB，4 MB~8 MB 和顶端的 4 MB）。在第一级页表中，有 1021 个表项的有效位都为 0，如果对它们进行访问，将引起缺页中断。这时，操作系统将注意到该进程试图访问一个不恰当的地址，因此将采取恰当的行动，如给进程发送一个信号将它杀死。在这个例子中，所选择的各种长度都是整数（代码、数据和栈都是 4 MB 长），并且选择 PT1 与 PT2 等长，但在实际中其他的值当然也是可能的。

图 4.10 的二级页表可以扩充为三级、四级或更多级。更多的级别带来了更多的灵活性，但算法的复杂性也会更高。一般的页表都不会超过二级。

页表项的结构

让我们把目光从大的页表结构转移到单个页表项的详细内容。对于不同的 CPU 来说，它们的页表项的具体安排是各不相同的，这里只讨论一些共性信息。在图 4.11 中，我们给出了一个页表项的示例。页表项的长度因机器而异，但一般使用的是 32 位，即 4 个字节。其中最重要的字段是物理页面号，因为地址映射的最终目标就是把虚拟页面号映射为相应的物理页面号。下一个字段是有效位，有效位表示该页面现在是否在内存中。如果这一位的值等于 1，则表示该页面现在位于内存中，也就是说，这个页表项是有效的，可以使用，它里面存放了这个页面所对应的物理页面号。如果这一位的值等于 0，则表示这个页面现在还在外存中，也就是说，这个页表项是无效的，如果这个时候去访问它，将会导致缺页中断。

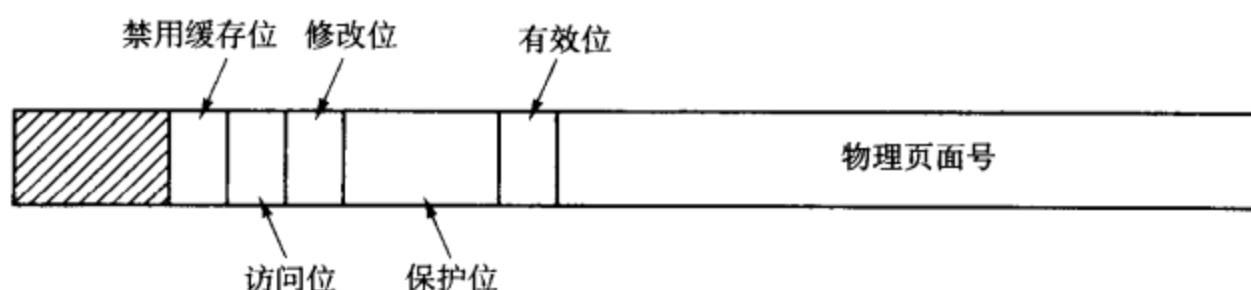


图 4.11 一个典型的页表项

接下来是保护位，表示允许对这个页面做何种类型的访问，是只读、可读写，还是可执行等。例如，在最简单的情形下，可以把这个字段设置为 1 位，如果它的值等于 0，表示可读/可写；如果它的值等于 1，表示只读。稍微复杂一点的实现方法可以使用 3 个相互独立的位，每一位分别表示是否允许对该页面进行读、写和执行操作。

修改位和访问位用来记录页面的使用情况。如果一个页面的内容被修改了，那么 CPU 就会自动地把它的修改位设置为 1。这一位的主要功能是，当操作系统需要回收一个物理页面时，先要检查这个修改位。如果该位为 1，即这个页面的内容曾经被修改过（称为“脏页面”），那么就必须把它写回到磁盘中；如果该位为 0，即这个页面的内容未被修改过（称为“干净页面”），那么

就可以把它覆盖掉，因为我们在磁盘上有它的备份。修改位有时也称为脏位，它反映了页面的当前状态。

如果一个页面被访问过，不管是读操作也好，写操作也好，那么它的访问位就会被硬件设置为1。这个信息主要用在页面置换算法中，也就是说，当一个缺页中断发生时，如果内存空间不够用，那么操作系统需要选择一个页面把它置换出去。如果有两个页面，一个曾经被访问过，另一个从未被访问过，显然后者更应该被置换出去。访问位在好几个页面置换算法中发挥了重要的作用，后面我们还会介绍。

最后一位可以禁止一个页面被缓存，对于那些被映射到设备寄存器而不是常规内存的页面来说，这个特性是非常重要的。如果操作系统正在紧张地循环等待某个I/O设备对它刚发出的命令做出响应，那么对于硬件来说，它必须直接从设备中读取命令字，而不是去访问一个旧的被缓存的副本。通过设置这个位，就可以禁止缓存。当然，如果机器具有独立的I/O地址空间，而不是使用内存映射的I/O地址，那么就不需要这个位。

请注意，如果一个页面被置换出内存，保存在磁盘上，那么它在磁盘上的起始地址并不会存放在它的页表项中。原因很简单，页表中的信息只是用于把虚拟地址转换为相应的物理地址。而操作系统在处理缺页中断时需要用到的信息，则是保存在操作系统内部的表格中，负责地址映射的硬件并不需要这些信息。

4.3.3 关联存储器 TLB

在大多数页式存储管理方案中，由于页表比较大，所以通常把它保存在内存中，但这种设计对性能有很大的影响。例如，假设一条指令的功能是把一个寄存器的内容复制到另一个寄存器。如果不使用页式存储管理，那么在执行这条指令的时候，只需要访问一次内存，即从内存取出该指令并交给CPU；但如果采用页式存储管理，那么内存的访问次数就不只一次，因为还需要去访问页表。由于系统的运行速度主要取决于CPU的取指令和内存的数据输出，这样，如果在每一次内存访问时，都要去访问两次页表，那么就会使系统的性能降低 $2/3$ 。这样的系统是没有人愿意使用的。

计算机设计者们很早就意识到这个问题，并提出了一个解决方案。它的基本思路来自于对程序运行过程的一个观察结果，也就是说，对于绝大多数的程序，它们在运行时，倾向于集中地访问一小部分的页面。这样，对于它的页表来说，只有一小部分的页表项会被经常地访问，而其他的页表项则很少使用。这种现象就是程序局部性原理的一个具体表现，我们后面还会介绍。

根据这个观察的结果，人们给计算机增加了一种特殊的快速查找硬件，即TLB（Translation Lookaside Buffer）或者称为关联存储器（associative memory），用来存放那些最常用的页表项。这种硬件设备能够直接把虚拟页面号映射为相应的物理页面号，而不需要去访问内存中的页表，这样就缩短了页表的查找时间。TLB通常位于MMU中，包含了少量的表项，一般不超过64个。在图4.12给出的例子中，有8个表项。每个表项包含了一个页面的信息，包括虚拟页面号、访问位、保护码（读/写/可执行权限），以及该页面所在的物理页面号。这些字段与页表中的字段有着一一对应的关系。另外还有一个位，标明该表项是否有效。

图4.12所示的TLB的当前状态可能来自于一个正在执行某个循环的进程，该循环的代码跨越了虚拟页面19、页面20和页面21，因此这三个页面的保护码是可读和可执行。当前正在使用的数据（不妨认为是一个数组）在页面129和页面130上，页面140包含的是用于数组访问的下标。最后，栈位于页面860和页面861上。

下面我们来看一下TLB是如何工作的。当一个虚拟地址到来时，MMU首先会到TLB中去查找，看看这个虚拟页面号是否出现在TLB中。这个查找的速度非常快，因为它是以并行的方式进

行的，同时与所有的页表项进行比较。如果能够找到而且此次访问是合法的，那么就直接从 TLB 中把相应的物理页面号取出来。如果能够找到该虚拟页面号，但违反了访问权限，例如，对一个只读的页面进行写操作，那么将引发一个保护中断。

有效位	虚拟页面号	修改位	保护码	物理页面号
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

图 4.12 用于加快页表查找的 TLB

如果在 TLB 中未找到该虚拟页面号，那么就要采用通常的办法去访问内存中的页表，把相应的物理页面号取出来。但是事情并没有到此结束，硬件还会从 TLB 中把某一个表项驱逐出来，然后把刚刚访问的这个页表项添加到 TLB 中。这样，如果过一会儿又要去访问这个页面，就能够在 TLB 中找到它了。当一个表项被驱逐出 TLB 时，要把它修改位复制到内存的页表项中，因为它的值可能会发生变化。

软件 TLB 管理

到目前为止，我们一直假设每台支持虚拟页式存储的机器都有能被硬件识别的页表，外加一个 TLB。在这种设计中，TLB 的管理和 TLB 未命中时的处理完全由 MMU 硬件来完成，只有当页面不在内存中时才会陷入到操作系统。

在过去，这个假设是正确的。但是在一些现代的 RISC 机中，包括 SPARC, MIPS, HP PA 和 Power PC，几乎所有的页面管理工作都是由软件来完成的。在这些机器中，TLB 表项是由操作系统来负责装入的。如果发生 TLB 未命中的情形，MMU 不是到页表中找到并装入所需的页面信息，而是产生一个 TLB 中断，把问题交给操作系统。由操作系统负责找到相应的页面，并从 TLB 中淘汰一个表项，装入新访问的这个表项，然后重新执行被中断的指令。所有这些工作必须用很少的几条指令完成，因为 TLB 未命中的发生频率要远远高于缺页中断的发生频率。

令人惊奇的是，如果 TLB 的容量足够大（比如 64 个表项），那么使用软件来管理 TLB 在性能上是可以接受的。这样做的优点是使 MMU 的设计更为简单，从而在 CPU 芯片上腾出相当大的一块面积，可以用于高速缓存和其他部件，以进一步提高系统的性能。软件 TLB 管理的详细内容请参阅 Uhlig et al. (1994)。

对于使用软件来管理 TLB 的机器，人们设计了许多方法来提高它们的性能。一种方法研究了如何来降低 TLB 的未命中率，以及在 TLB 未命中时，如何尽可能地降低系统的开销 (Bala et al., 1994)。为了降低 TLB 的未命中率，操作系统有时可以预测出哪些页面即将被访问，并把它们预先装入 TLB。例如，当一个客户进程向位于同一台机器上的服务器进程发送一个消息时，服务器进程很可能会马上运行。根据这个信息，当系统在处理客户发出的 send 系统调用时，可以同时去查找服务器进程的代码、数据和栈所在的页面，并将它们提前映射进来，以避免 TLB 中断的发生。

无论是硬件方式还是软件方式，在处理一次 TLB 未命中时，通常的做法是去访问内存中的页表，查找相应的页表项。如果采用软件方式来实现，一个问题就是用来存放页表的页面本身可能就不在 TLB 中，这样就会在处理过程中再一次引发 TLB 中断。为了解决这个问题，可以在内存的固定位置设置一个比较大（如 4 KB）的软件缓冲区，用来存放最近访问过的一些 TLB 表项，而这个缓冲区所在的页面始终位于 TLB 中。这样，就可以大大降低 TLB 未命中的次数。

4.3.4 反置页表

我们前面所讲的各种页表方案都是根据进程的虚拟页面号来组织的，用虚拟页面号来作为访问页表的索引。在这种情形下，如果虚拟地址的位数为 32 位，那么虚拟地址空间的大小就是 2^{32} 个字节。如果页面的大小是 4 KB，那么每个进程的页表项的个数就是 100 万。假设每个页表项的长度为 4 个字节，那么一个页表就要占用 4 MB 的内存空间。在一些比较大的系统中，这个规模还是可以接受的。

然而，随着 64 位计算机变得越来越普及，形势发生了很大的变化。虚拟地址空间的大小变成了 2^{64} 个字节。如果页面的大小是 4 KB，那么页表项的个数就是 2^{52} 。假设每个页表项的长度为 8 个字节，那么一个页表就要占用 255 个字节。这显然是一个天文数字，在目前及不远的将来都是不可能实现的。因此，对于这种 64 位的地址空间来说，要想实现页式存储管理，就必须对页表的结构进行改进。

一种解决方案就是反置页表（inverted page table），也就是说，根据内存中的物理页面号来组织页表，用物理页面号来作为访问页表的索引。有多少个物理页面，就在页表中设置多少个页表项。例如，假设虚拟地址为 64 位，页面大小为 4 KB，物理内存的大小为 256 MB，那么总的页表项的个数等于 256 MB 除以 4 KB，即 65 536。在每一个页表项中，记录了在相应的物理页面中所保存的是哪一个进程的哪一个虚拟页面。

反置页表的方法节省了大量的内存空间，尤其是当虚拟地址空间远远大于内存空间时，但它有一个很大的缺点，即从虚拟页面号到物理页面号的转换变得更加复杂。因为我们在访问内存单元时，所给出的都是虚拟地址和虚拟页面号，而反置页表是根据物理页面号的顺序来存放的，所以在给定了一个虚拟页面号以后，必须搜索整个页表，才能找到它所对应的物理页面号。例如，假设有一个进程 n ，它要去访问第 p 个虚拟页面，那么对于硬件来说，它无法像访问通常的页表那样，以 p 为索引去查找页表。它只能逐一地去搜索整个页表，看哪一个表项描述的是 (n, p) 。另外，在每一次内存访问的时候，都要进行这样的搜索，这显然对系统的性能是非常不利的。

摆脱这种困境的方法是使用 TLB。如果在 TLB 中能够存放所有经常要访问的页面，那么地址映射就会像通常的页表一样快。不过，如果 TLB 未命中，那么就必须对整个反置页表进行搜索。为了加快这个搜索过程，可以使用虚拟地址来建立一个哈希表。如图 4.13 所示，如果两个虚拟页面具有相同的哈希值，那么它们就被链在一起。如果这个哈希表的表项个数与物理页面的个数一样多，那么平均下来，每条链表上只会有一个页面，这样就极大地加快了映射的速度。在找到了相应的物理页面号之后，就把新的（虚拟页面号，物理页面号）对插入到 TLB 中，然后重新执行被中断的指令。

当前，反置页表主要用在一些 IBM、Sun 和 HP 的工作站上，随着 64 位机的普及，它的应用将会变得更加普遍。除了反置页表之外，处理大规模虚拟地址空间的其他方法可以参阅下列文献：Huck and Hays (1993); Talluri and Hill (1994); Talluri et al. (1995)。在实现虚拟存储器时的一些硬件问题可以参阅 Jacob and Mudge (1998)。

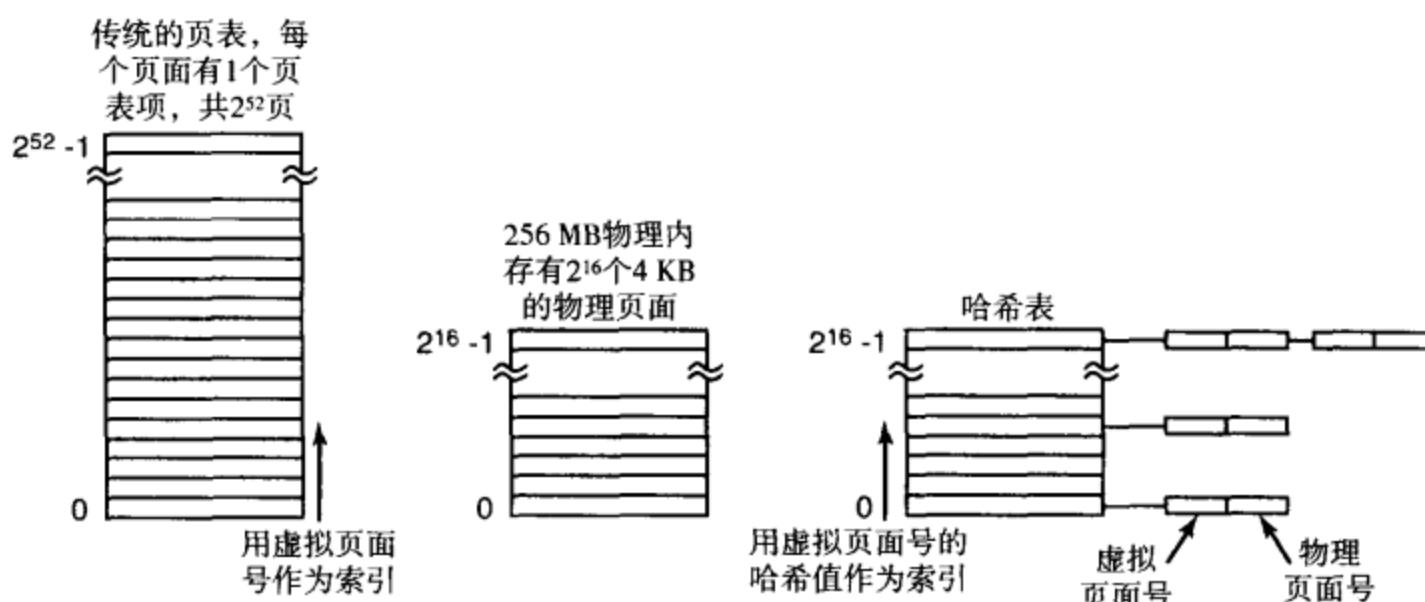


图 4.13 传统页表与反置页表的比较

4.4 页面置换算法

当一个缺页中断发生时, 需要从磁盘上调入相应的页面。如果此时内存已经满了, 就需要从内存中选择一个页面, 把它置换出去, 从而把它所占用的空间腾出来, 装入新的页面。如果被选中的页面曾经被修改过, 那么就必须把它写回到磁盘, 以更新磁盘上的页面副本。如果该页面未被修改过(例如, 该页面是程序代码), 那么我们可以直接抛弃该页面, 用新的页面来覆盖它, 因为它在磁盘上有一份完全相同的副本。

那么如何来挑选被置换的页面呢? 当然, 最简单的办法就是从内存中随机地选择一个页面, 把它置换出去。但这显然不是一种令人满意的方法, 因为如果选中的是一个经常要访问的页面, 那么当它被置换到外存以后, 过一段时间后, 恐怕又得把它再换进来, 而这种页面的换进换出并不是免费的, 相反需要一定的系统开销。因此对于一个好的页面置换算法来说, 应该尽可能地减少页面的换进换出次数, 或者说, 尽可能地减少缺页中断的次数。关于页面置换算法, 无论是在理论研究上, 还是在实际应用中, 人们都做了大量的工作, 提出了各种各样的算法。下面我们将介绍其中一些比较重要的算法。

需要指出的是, “页面置换”问题也出现在计算机科学的其他领域。例如, 大多数计算机都有一个或多个内存的高速缓存, 用来存放最近访问过的32字节或64字节的内存块。如果高速缓存已经满了, 就必须选择其中的某一个数据块, 把它移出去。这个问题和页面置换问题是完全一样的, 只不过它的时间规模更短(它必须在几纳秒内完成, 而不是页面置换中的几毫秒)。之所以要求这么短的时间, 是因为它需要的数据块来自于内存, 而内存的访问速度比较快, 它不像磁盘访问那样, 需要磁头定位和旋转延迟。

第二个例子是Web浏览器。浏览器会在磁盘的缓存区中保留一些以前访问过的Web页面的副本。一般来说, 缓存区的最大容量是事先确定的, 因此, 如果浏览器使用得较多的话, 缓存区可能就会满。当用户去访问一个Web页面时, 首先检查一下缓存区, 看是否已经有了该页面。如果有的话, 再看一下它是否与Web上的页面一致, 如果是的话, 就直接去使用缓存区中的这个副本; 如果Web上的页面已经被刷新, 或者在缓存区中没有找到该页面, 那么就从Web上下载一份最新副本。如果缓存区已满, 就必须做出一个决策, 从中挑选一个页面, 把它删除。这个问题与页面置换问题也是类似的, 只不过缓存区中的Web页面从来不会被修改, 因此无须把它们写回到Web服务

器中。而在虚拟页式存储管理中，内存中的页面既可能是干净的(未被修改过)，也可能是脏的(已被修改过)。

4.4.1 最优页面置换算法

最优页面置换算法易于描述但无法实现。它的基本思路是：当一个缺页中断发生时，对于内存中的每一个虚拟页面，计算在它的下一次访问之前，还需要等待多长的时间，用指令数表示。例如，有的页面在下一条指令中就要访问到(下一条指令就位于该页面中)，而有的页面可能要等到10条、100条或者1000条指令后才会被访问，因此，我们可以给每一个页面标注一个指令数，表示要执行完这么多条指令之后，才会第一次去访问该页面。

最优页面置换算法的基本思路就是选择等待时间最长(标注的指令数最大)的那个页面，把它作为被置换的页面。如果一个页面要过800万条指令后才会被访问，另一个页面要过600万条指令后才会被访问，显然应该选择前者，把它置换出去。这样就可以把缺页中断的发生时间尽可能地往后推延。就像我们人类一样，计算机也会把自己不喜欢的事件尽量往后拖延。

这个算法唯一的缺点就是它是无法实现的，因为操作系统根本就无从知道每一个页面还要等待多长的时间，才会被再次访问(在最短作业优先调度算法中，我们也碰到过类似的问题，操作系统怎么知道哪一个作业是最短的呢？)。不过，最优页面置换算法也不是一无是处，它可以用做其他算法的性能评价的依据，也就是说，如果要判断一个页面置换算法好不好，可以拿它和最优页面置换算法进行比较，看有多大的差距。具体来说，可以在一个模拟器上运行某个程序，并且记录下每一次的页面访问情况，有了这些数据以后，在第二次运行这个程序的时候，就可以采用最优算法，计算出最少的缺页次数，然后把它和其他算法的运行结果做一个比较。例如，如果某一个操作系统使用的页面置换算法在性能上仅比最优算法低1%，那么无论投入多大的努力去改进该算法，最后的性能提高都不会超过1%。

为了避免可能的混淆，读者必须清楚，这个页面访问情况的记录只涉及到刚刚被测试的程序，因此从中导出的页面置换算法也只是针对该程序的。虽然这种方法对评价页面置换算法很有用，但它在实际系统中没有什么用处。下面我们将介绍一些实际使用的算法。

4.4.2 最近未使用页面置换算法

为了使操作系统能够收集每个页面被使用的统计信息，在大多数支持虚拟存储的计算机中，每个页面都有两个状态标志位，即访问位R和修改位M。当一个页面被访问时(包括读操作和写操作)，它的R位被设置为1；当一个页面被修改时(即该页面被写入数据)，它的M位被设置为1。这两个标志位存放在页面的页表项中，如图4.11所示。请注意，这些位要求在每一次内存访问时都要进行更新，因此必须由硬件来自动进行置位。如果某个位被设置成1，那么它就一直保持为1，直到操作系统使用相应的指令把它复位为0。

如果在硬件上没有这两个位，也可以在软件上进行模拟。当一个进程被启动时，把所有的页面都标记为不在内存中。这样，一旦有一个页面被访问，就会引发一个缺页中断，这时操作系统就可以去设置R位(在它的内部表格中)，并修改其页表项，使之指向正确的页面，并将属性设为只读模式，然后重新执行被中断的指令。如果该页面随后被写入新内容，又会引发一个页面中断，这时，操作系统就可以去设置这个页面的M位，并把它的属性改为可读/可写。

R位和M位可以用来构造一个简单的页面置换算法：当一个进程被启动时，操作系统会把所有的页面的这两个位都设置为0。然后，在进程运行过程中，R位会被定期地(例如在每次时钟中

断时)清零,这样,就能把最近未被访问过的页面与最近被访问过的页面区别开来(所谓最近,指的是从上一个时钟中断到当前的这一段时间)。

当一个缺页中断发生时,操作系统会去检查所有的页面,并根据当前的R位和M位的值,把它们分为4类:

第0类:未被访问,未被修改。

第1类:未被访问,已被修改。

第2类:已被访问,未被修改。

第3类:已被访问,已被修改。

对于第1类,表面上看似乎是不可能的,但实际上,对于一个第3类的页面,如果在上一个时钟中断中它的R位被清零,而且到目前为止它还未被访问,这时它就变成了第1类。时钟中断之所以不去清除M位,是因为将来该页面被置换出去的时候,需要根据这个信息来决定是否需要把这个页面的内容写回到磁盘。

最近未使用算法(Not Recently Used, NRU)随机地从编号最小的非空类中挑选一个页面,把它淘汰出去。这个算法的隐含意思是,如果有两个页面,一个曾经被修改过,但在最少一个时钟周期内(通常是20 ms)未被访问;另一个未被修改过,但在最近一个时钟周期内肯定被访问过,对于这样的两个页面,算法认为应该淘汰前者。NRU算法的主要优点是易于理解、实现起来效率较高,而且它的性能尽管不是最好的,但也还是够用的。

4.4.3 先进先出页面置换算法

另一种低开销的页面置换算法是先进先出(First-In, First-Out, FIFO)算法。为了说明它的工作原理,我们来看一个例子。假设有一家超市,它的货架恰好能摆放 k 种不同的商品。有一天,某家公司推荐了一种新的方便食品——快速、冷冻干燥并可用微波炉加工的乳酸酪,这个产品非常成功,该超市很希望能进货。但由于超市的空间有限,所以我们必须将一种旧商品下架,以便腾出地方来存放这种新产品。

一种办法是,在超市中寻找进货时间最长的那种商品(比如说,120年前就有卖的东西),并将它下架处理,理由是已经没有人喜欢它了。这实际上相当于超市有一个按进货时间排列的所有商品的链表,新的商品加入到链表的末尾,而位于链表开头的商品将被撤掉。

同样的思路也可以用在页面置换算法中。操作系统维护着一个链表,在这个链表中,记录了所有位于内存中的虚拟页面,从链表的排列顺序来看,链首页面的驻留时间最长,链尾页面的驻留时间最短。当发生一个缺页中断时,就把链首的页面淘汰出局,并且把新的页面添加到链表的末尾。FIFO算法的性能不是很好。仍以超市为例,算法可能会淘汰一种不常用的商品,但也可能会淘汰面粉、盐或黄油等日常用品。同样,当它用在计算机上时也会引起类似的问题,被它淘汰出去的页面可能是经常要访问的页面。基于这个原因,FIFO算法很少被单独使用。

4.4.4 第二次机会页面置换算法

为了克服FIFO算法的缺点,人们对它进行了改进,即对于最古老的那个页面,去检查它的R位。如果R位是0,说明这个页面老且无用,应该立刻淘汰出局;如果R位是1,则说明该页面曾经被访问过,因此就再给它一次机会。具体来说,先把R位清零,然后把这个页面放到链表的尾端,并修改它的装入时间,就好像它刚刚进入系统一样,然后继续往下搜索。

这个算法称为第二次机会（second chance），如图 4.14 所示。在图 4.14(a)中，我们看到页面 A 到 H 被保存在一个链表中，并按照它们抵达内存的时间排好了序。

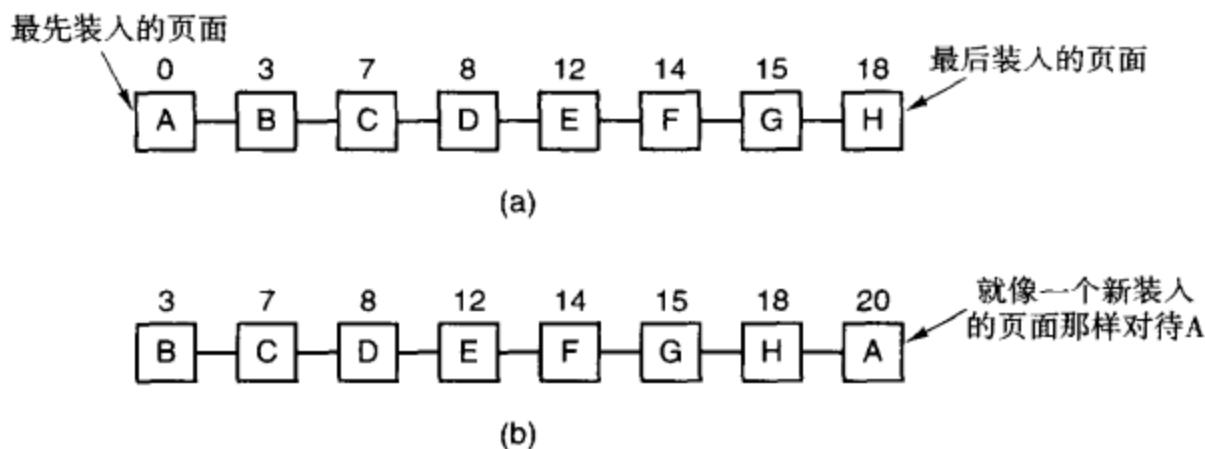


图 4.14 第二次机会页面置换算法: (a) 页面按先进先出的顺序排列; (b) 在时间 20 处发生一次缺页中断且 A 的 R 位为 1 时的页面链表。页面上方的数字表示它们的装入时间

假设在时间 20 处发生了一个缺页中断，这时最古老的页面是 A，它的到达时间为 0，即进程刚刚启动的时候。如果 A 的 R 位是 0，它将被淘汰出局。这时要根据它的修改位的值来进行处理，或者是把它写回到磁盘（修改位的值为 1），或者只是简单地把它抛弃（修改位的值为 0）；另一方面，如果 R 位的值为 1，则把 A 放到链表的尾部并将它的“装入时间”设置为当前时间（20），同时它的 R 位将被清零。然后从下一个页面即页面 B 开始继续往下搜索。

第二次机会算法的实质就是寻找一个比较古老且从上一次时钟中断以来尚未被访问的页面。如果所有的页面都被访问过了，它就退化为纯粹的 FIFO 算法。具体来说，假设在图 4.14(a)中，所有页面的 R 位都被设置为 1，那么操作系统将一个接一个地把各个页面移到链表的末尾，并清除被移动的页面的 R 位。最后算法又将回到页面 A，此时它的 R 位已经被清除了，因此 A 将被淘汰，所以这个算法总是可以结束的。

4.4.5 时钟页面置换算法

虽然第二次机会算法是一个较合理的算法，但它经常需要在链表中移动页面，这样做既降低了效率，又是不必要的。一种更好的办法是把各个页面组织成环形链表的形式，类似于时钟的表面，如图 4.15 所示。然后把一个指针指向最古老的那个页面，或者说，最先进来的那个页面。

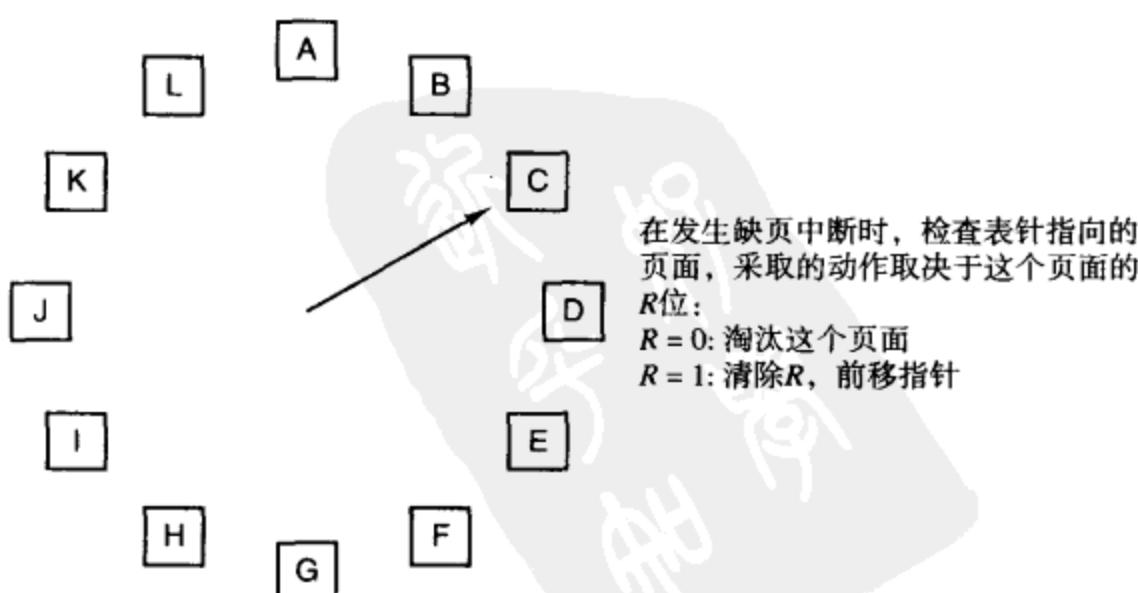


图 4.15 时钟页面置换算法

当一个缺页中断发生时，检查指针所指向的那个页面，如果它的R位的值等于0，则淘汰该页面；如果R位的值等于1，则把它清零，并把指针指向下一个页面。就这样一直进展下去，直到发现一个其访问位的值等于0的页面并把它淘汰掉。这个算法称为时钟页面置换算法，它和第二次机会算法的功能是完全一样的，只是在具体实现上有所不同。

4.4.6 最近最久未使用页面置换算法

最近最久未使用(Least Recently Used, LRU)算法的基本思路是，当一个缺页中断发生时，从内存中选择最久未被使用的那个页面，把它淘汰出局。这种算法实质上是对最优页面置换算法的一个近似，它的理论依据就是程序的局部性原理。也就是说，在最近的一小段时间内，或者说，在最近的几条指令中，如果某些页面被频繁地访问，那么在将来的一小段时间内，它们还可能会再一次被频繁地访问。反过来说，如果在过去一段时间内，某些页面长时间未被访问，那么在将来，它们还可能会长时间得不到访问。对于最优页面置换算法，它寻找的是将来长时间内得不到访问的那个页面，但将来的访问情况是不知道的，所以LRU算法的策略就是根据程序的局部性原理，利用过去的、已知的页面访问情况，来预测将来的情况。

尽管LRU算法在理论上是可行的，但是系统的开销比较大。为了完全实现LRU算法，系统必须维护一个页面链表，把最近刚刚使用过的页面作为首结点，把最久没有使用的页面作为尾结点。然后在每一次内存访问的时候，都对这条链表进行更新。也就是说，从链表中找到相应的页面，把它摘下来，移动到链表的开头，成为新的首结点。这种操作的开销是非常大的，即使能设计出相应的硬件，也不能完全解决问题。

不过，还是有其他一些方法，在特殊硬件的帮助下，能够实现LRU算法。我们先考虑一种最简单的方法。它要求在硬件上有一个64位的计数器C，在每条指令执行完后C的值会自动加1。另外，在每个页表项中，必须有一个足够大的字段来存放该计数器的值。在每一次内存访问后，C的当前值会被存放在被访问页面的页表项中(作为它的时间戳)。当发生一个缺页中断时，操作系统会去检查页表中所有计数器的值，从中找出一个最小值，该页面就是最近最久未使用的页面。

我们再来看一下第二个硬件LRU算法。在一台机器上，有n个物理页面，LRU硬件可以维护一个 $n \times n$ 位的矩阵，初始化时全部设置为0。如果第k个物理页面被访问，硬件首先把第k行全部都设置为1，再把第k列全部都设置为0。在任何时刻，其二进制值最小的那一行就是最近最久未使用的，第二小的那一行就是下一个最久未使用的，依次类推。这个算法的工作原理可以用图4.16中的例子来说明，在这个例子中，有4个物理页面，它们的访问次序为

0 1 2 3 2 1 0 3 2 3

在页面0被访问后，矩阵的状态如图4.16(a)所示。在页面1被访问后，矩阵的状态如图4.16(b)所示，依次类推。

4.4.7 LRU算法的软件模拟

虽然上述LRU算法在理论上都是可行的，但实际上，很少有计算机配备了这些硬件。因此，对于操作系统的设计师来说，这些算法都是无意义的，毕竟，巧妇难为无米之炊。我们真正需要的是一个能用软件来实现的解决方案，如最不经常使用(Not Frequently Used, NFU)算法。对于每一个页面，算法将设置一个软件计数器，它的初值为0。在每次时钟中断时，操作系统将对内存中的所有页面进行扫描，把每个页面的R位(其值为0或1)的值加到它的计数器上。也就是说，算法

使用这个计数器来记录页面被访问的频繁程度。当发生一个缺页中断时，计数器值最小的那个页面将被淘汰出局。

页面			
0	1	2	3
0	0	1	1
1	0	0	0
2	0	0	0
3	0	0	0

页面			
0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

页面			
0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

页面			
0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

页面			
0	1	2	3
0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

页面			
0	1	2	3
0	0	0	0
0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

页面			
0	1	2	3
0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

页面			
0	1	2	3
0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

页面			
0	1	2	3
0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

图 4.16 使用矩阵的 LRU 算法，页面的访问次序为 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

NFU 的主要问题是它从来不忘记任何事情。例如，对于一个多遍扫描的编译器，在第一遍扫描中频繁用到的页面，在程序进入第二遍扫描时其计数器的值可能仍然很高。事实上，如果第一遍扫描的执行时间恰好是各遍扫描中最长的，那么对于第二遍以后访问的页面来说，它们的计数器的值可能总是比第一遍扫描中的页面要小，其结果就是，操作系统将删掉有用的页面，而不是不再使用的页面。

幸运的是，只要对 NFU 做一个小小的修改就能让它更好地模拟 LRU。修改工作分为两部分：一是先把计数器的值右移一位，然后再把 R 位的值加进来；二是把 R 位加到计数器的最左端而不是最右端。

修改以后的算法称为老化 (aging) 算法，它的工作原理如图 4.17 所示。假设在第一个时钟节拍后，页面 0 到页面 5 的 R 位的值分别是 1, 0, 1, 0, 1, 1，换句话说，在时钟节拍 0 到时钟节拍 1 期间，页面 0, 2, 4, 5 被访问过，它们的 R 位被设置为 1，而其他页面的 R 位仍然是 0。我们把对应的 6 个计数器先右移 1 位，然后把 R 位插入其左端，得到的结果如图 4.17(a) 所示。图中后面的 4 列是在接下来的 4 个时钟节拍后这 6 个计数器的值。

当一个缺页中断发生时，计数器值最小的那个页面将被淘汰。如果一个页面已经连续 4 个时钟节拍未被访问过，那么在它的计数值最左边应该有 4 个连续的 0。类似地，如果一个页面已经连续 3 个节拍未被访问，则它的计数值最左边有 3 个 0。两者相比较，显然前者的计数值更小，所以应该先淘汰它。

老化算法与 LRU 算法存在两点区别。考虑图 4.17(e) 中的页面 3 和页面 5，它们都已经连续两个时钟节拍未被访问了，而在之前那个节拍中它们都被访问过，因此它们的计数值的前三位都是 001。根据 LRU 算法，如果在当前时刻需要淘汰一个页面，那么就应该在这两个页面中选一个。但现在的问题是，我们不知道在时钟节拍 1 到时钟节拍 2 这个期间，这两个页面的访问顺序是什么，哪一个先访问，哪一个后访问。由于每个时钟周期的访问信息只使用一个数据位来记录，其值非 0 即 1，

这就使我们无法区分在一个周期内的先后顺序。因此，我们所能做的只是淘汰页面3，因为页面5在此前的两个节拍中也被访问过，而页面3则没有。

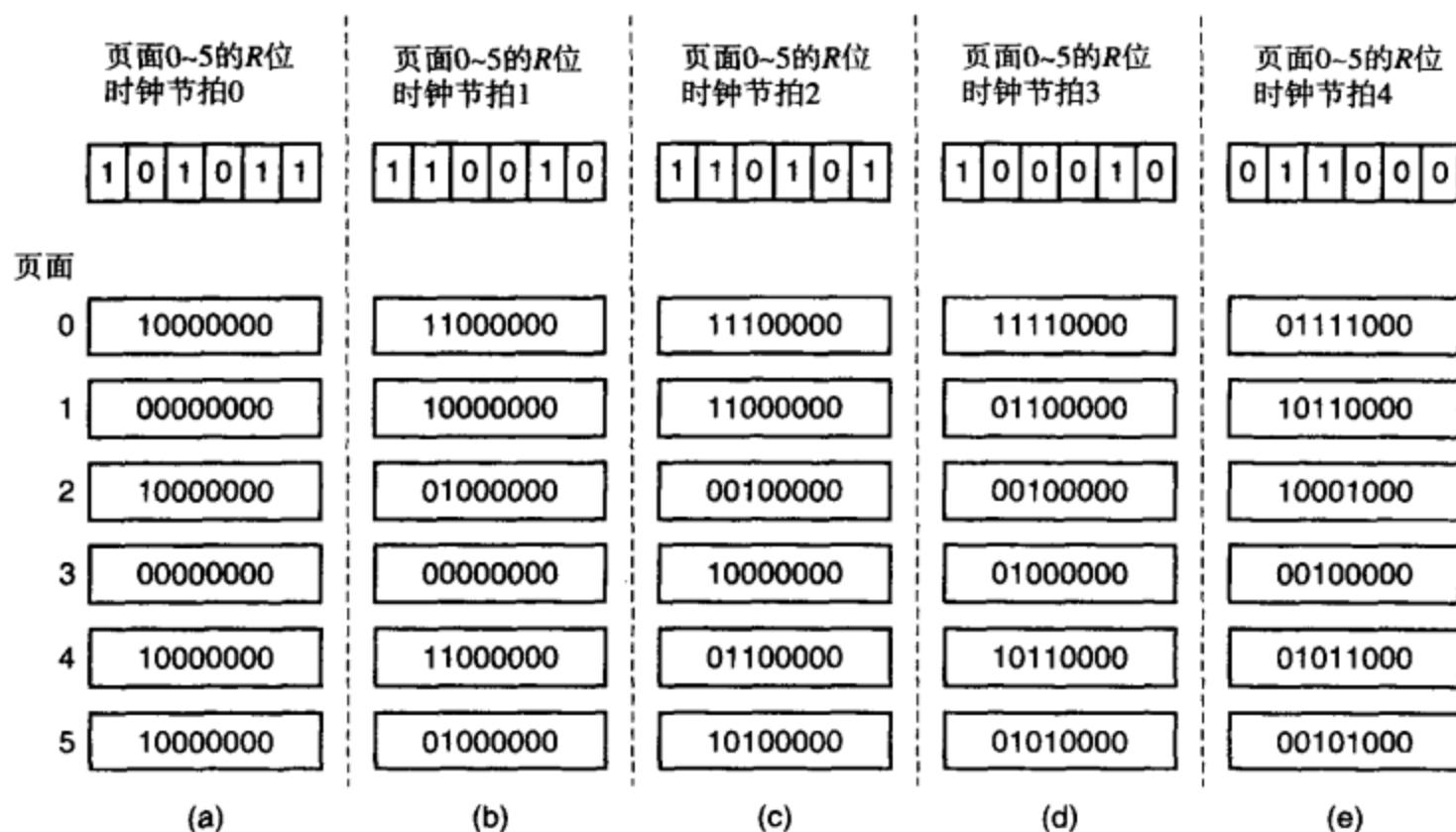


图 4.17 用软件来模拟LRU的老化算法。图中所示的是6个页面在5个时钟节拍的情况下，5个时钟节拍分别由(a)~(e)表示

两种算法的第二个区别是老化算法的计数器只有有限位，在本例中是8位。这样，如果两个页面的计数值都是0，那么我们只能在这两个页面中随机选一个。而实际上，可能一个页面的上次访问时间是9个节拍以前，另一个页面是在1000个节拍以前，而我们却无从知晓这些信息。不过，在实际应用中，如果时钟节拍是20 ms，那么8位一般是够用的。如果一个页面已经有160 ms未被访问，那么它可能就不那么重要。

4.5 页式存储管理中的设计问题

在前面几个小节中，我们解释了页式存储管理的工作原理，并讨论了几个基本的页面置换算法，但仅仅知道这些基本原理是远远不够的。为了设计一个系统，必须知道更多的东西，才能使它工作得更好。打个比方来说，如果读者知道了象棋中的车、马和其他棋子的走法，这并不意味着你已经成为了一名象棋高手。在以下的几个小节中，我们来看一些具体的设计问题，为了使页式存储管理达到一个较好的性能，操作系统的设计者就必须仔细考虑这些问题。

4.5.1 工作集模型

在纯粹的虚拟页式存储管理中，当一个进程被启动执行时，它的所有页面都还在外存。当CPU试图去取出第一条指令时，就会引发缺页中断，于是操作系统就会把包含有第一条指令的那个页面装入内存。接下来，在程序运行时，一般很快就会去访问全局变量和栈，这样又会引发一些缺页中断。过了一会儿，进程所需要的页面大都位于内存中，因此可以安心地运行一段时间，缺页中断的次数也比较少。以上这种策略称为请求调页(demand paging)，因为页面不是预先被装入内存的，而是根据需要随要随调。

在请求调页方式下,我们可以很容易地编写一段测试程序,系统地去访问一个大的地址空间中的所有页面。由于没有足够的内存空间能够容纳所有这些页面,因此这段程序将会触发大量的缺页中断。但幸运的是,绝大多数的进程都不是这样工作的。它们都会表现出一种访问的局部性(l locality of reference),也就是说,在进程运行的任何一个阶段,它只会去访问一小部分的页面。例如,对于一个多遍扫描的编译器,在每一遍扫描时只会访问一小部分页面,而且不同的扫描访问的是不同的页面。局部性原理的概念在计算机科学中得到了广泛的应用,关于它的发展历史请参阅 Denning (2005)。

一个进程当前正在使用的页面集合称为它的工作集(working set)(Denning, 1968a; Denning, 1980)。如果进程的整个工作集都在内存中,那么它将会很顺利地运行,而不会造成太多的缺页中断。这种局面会一直持续下去,直到它进入另一个执行阶段(如编译器的下一遍扫描),这时进程的工作集会发生剧烈的变动,它的运行也会进入一个调整期。如果分配给一个进程的物理页面数太少,不能包含整个工作集,这时,进程将会造成很多的缺页中断,需要频繁地在内存与外存之间置换页面,从而使得进程的运行速度变得很慢,我们把这种状态称之为抖动(Denning, 1968b)。由于指令的执行速度非常快,通常只有几纳秒,而磁盘的读写操作非常慢,通常需要10 ms,所以当进程在运行时,如果频繁地出现缺页中断,频繁地去访问磁盘,那么对于它的运行速度来说,将会造成很大的影响。

在多道程序系统中,经常需要把进程转移到磁盘上(即把它的所有页面从内存中移走),从而让其他进程有机会去使用CPU。问题是,当一个进程被再次调入内存时应该如何处理?从技术的角度来说,什么都不用做,该进程将会不断地产生缺页中断,直到它的工作集全部被装入。但问题是,如果每次装入一个进程都要产生20个、100个甚至1000个缺页中断,那么速度是非常慢的,同时也浪费了相当多的CPU时间,因为操作系统在处理一个缺页中断时,需要花费几毫秒的时间。

因此,许多页式存储管理系统都试图记录每个进程的工作集,并确保在进程运行之前,它的工作集就已经在内存中了。这种方法称为工作集模型(Denning, 1970),它旨在大大减少缺页中断的次数。在一个进程运行之前就预先装入它的页面,这种技术也称为预先调页(prepaging)。需要指出的是,随着时间的变化,进程的工作集的内容也会发生变化。

如前所述,大多数程序在运行时都不会均衡地去访问它们的地址空间,而是会集中地去访问一小部分页面。内存访问的类型包括取指令、读数据或存数据。一个进程的工作集可以用一个二元函数 $w(k, t)$ 来表示,其中 t 指的是当前的执行时刻, k 称为工作集窗口,它是一个定长的页面访问窗口。工作集 w 等于在当前时刻 t 之前的 k 次内存访问的所有页面所组成的集合。 k 越大,意味着工作集窗口越宽,往回看的页面访问数越多,从而工作集中的页面数越多。因此 $w(k, t)$ 是 k 的一个单调非递减函数。图4.18描述了工作集的大小与 k 之间的函数关系。

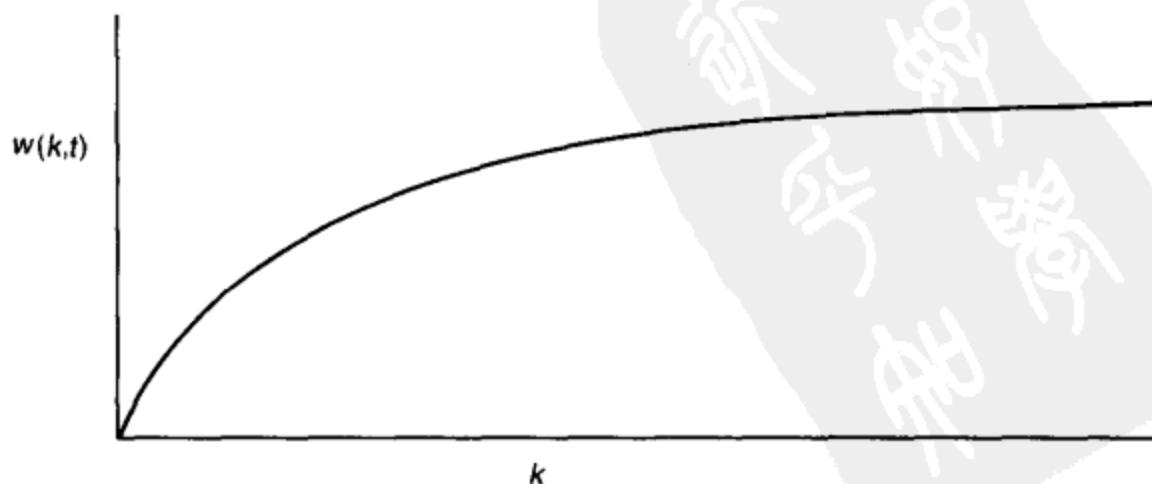


图4.18 工作集是最近 k 次内存访问所用到的页面集合。函数 $w(k, t)$ 是 t 时刻工作集的大小

从图 4.18 中的曲线来看，开始时工作集的增长速度很快，后来就慢慢平缓下来。这说明大多数程序会随机地访问一小部分页面，但随着时间的增长，这个集合就会慢慢稳定下来。举例来说，假设有一个程序正在执行一段循环语句，这段代码占用了 2 个页面，它所用到的数据分别属于 4 个页面。由于是循环语句，因此每执行 1000 条指令，可能会访问所有的这 6 个页面。但其他的页面一般不会去访问，比如说，针对另外某个页面的访问可能会出现在 100 万条指令之前，可能是在程序初始化的时候。由于工作集具有这种渐进式、缓慢变化的特点，因此它的内容不会对 k 的取值很敏感。换句话说，我们可以取不同的 k 值，而工作集的内容不会有任何变化。这样一来，我们就可以把这个特点用在预先调页技术中。也就是说，假设一个进程开始时被中止运行，存放在磁盘上，后来又要把它调入内存重新运行。这时，我们就可以根据进程在中止前的工作集内容，猜测出当它重新开始运行后，可能会去访问哪些页面。这样，在开始运行该进程之前，我们可以先把这些页面装入内存，从而减少缺页中断的次数。

为了实现工作集模型，操作系统必须知道哪一些页面属于工作集。一种办法就是使用上面讨论过的老化算法，对于任何一个页面，如果在它的计数值的高 n 位中包含有一个 1，那么这个页面就是工作集的一个成员。如果一个页面连续 n 个时钟周期未被访问，就把它从工作集中删除。对于不同的系统，参数 n 的值可能是不同的，这必须通过一些实验来确定。不过，系统的性能对这个参数值并不是特别敏感。

有关工作集的信息可以用来提高时钟页面置换算法的性能。本来，如果指针指向的页面的 R 位是 0，那么就淘汰该页面。而现在的改进之处就是检查这个页面是否属于当前进程的工作集，如果是，就跳过该页面。这个算法称为 **wsclock**。

4.5.2 局部与全局分配策略

在前面几个小节中，我们曾经讨论过几种页面置换算法。另外一个与之相关的问题（到目前为止我们一直在回避这个问题）是，在相互竞争的进程之间如何来分配内存。

我们来看一下图 4.19(a)。在该图中，有三个进程 A、B 和 C 在运行。假如 A 发生了缺页中断，那么页面置换算法在寻找最近最久未使用的页面时，是只考虑分配给 A 的 6 个页面呢，还是考虑内存中的所有页面？如果只考虑 A 的页面，那么年龄值最小的页面是 A5，于是我们将得到图 4.19(b) 所示的情形。

另一方面，如果考虑的是内存中的所有页面，那么年龄值最小的页面 B3 将被选中，于是我们将得到图 4.19(c) 所示的情形。图 4.19(b) 的算法称为局部（local）页面置换算法，而图 4.19(c) 的算法称为全局（global）算法。局部算法实际上对应于为每个进程分配固定大小的内存空间，而全局算法可以在所有进程之间动态地分配物理页面，因此分配给各个进程的物理页面数是随时间变化的。

在通常情形下，全局算法的性能比局部算法好。如果在进程的运行期间，工作集大小发生较大的变化，那么全局算法的优势就会更加明显。如果采用局部算法，那么当进程的工作集增大时，即使内存中还有大量的空闲页面，也可能导致抖动现象；而如果工作集收缩了，那么又会浪费内存空间。对于全局算法，系统必须不断地确定应该给各个进程分配多少个物理页面。一种办法是监视由年龄位指出的工作集的大小，但这种办法并不足以阻止抖动。因为工作集的大小可以在几微秒内改变，而年龄位只是一个粗略量度，它跨越了若干个时钟节拍。

另一种方式是设置一个算法来为进程分配物理页面。例如，定期地计算出正在运行的进程数目，并为它们分配相等的份额。假设系统中有 12 416 个空闲物理页面和 10 个进程，那么每个进程将获得 1241 个空闲页面，剩下的 6 个被放入一个公用池中，用于缺页中断。

年龄	
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)		(b)		(c)	
A0		A0		A0	
A1		A1		A1	
A2		A2		A2	
A3		A3		A3	
A4		A4		A4	
A5		A6		A5	
B0		B0		B0	
B1		B1		B1	
B2		B2		B2	
B3		B3		B3	
B4		B4		B4	
B5		B5		B5	
B6		B6		B6	
C1		C1		C1	
C2		C2		C2	
C3		C3		C3	

图 4.19 局部与全局页面置换算法: (a)初始状态; (b)局部页面置换; (c)全局页面置换

这个算法貌似公平,但实际上,给一个 10 KB 的进程和一个 300 KB 的进程分配同样大小的内存空间,这是完全不合理的。或许,我们可以按照进程大小的比例来为它们分配页面,这样,300 KB 的进程所得到的份额就是 10 KB 进程的 30 倍。此外,最好给每一个进程都规定一个最小的物理页面数,这样不管多么小的进程都可以运行。例如,在某些机器上,一条双操作数的指令可能会涉及到 6 个页面,因为指令本身、源操作数和目的操作数都可能会跨越页面边界。如果在一个程序中包含这种指令,而且只给它分配了 5 个页面,那么该程序根本就无法运行。

在全局算法下,当一个进程刚开始运行时,可以先根据进程的大小给它分配一定数量的物理页面,然后在进程的运行过程中,再动态地调整它的内存空间的大小。在具体实现上,可以使用缺页率 (Page Fault Frequency, PFF) 算法。该算法用于控制分配集的大小,即什么时候应该增加、什么时候应该减少分配给进程的物理页面数。它并不关心当一个缺页中断发生时,应该把哪一个页面置换出去。

对于大多数的页面置换算法,包括 LRU 在内,当分配给进程的物理页面数增加时,缺页率将会下降,如图 4.20 所示。PFF 算法就是基于这个假设。

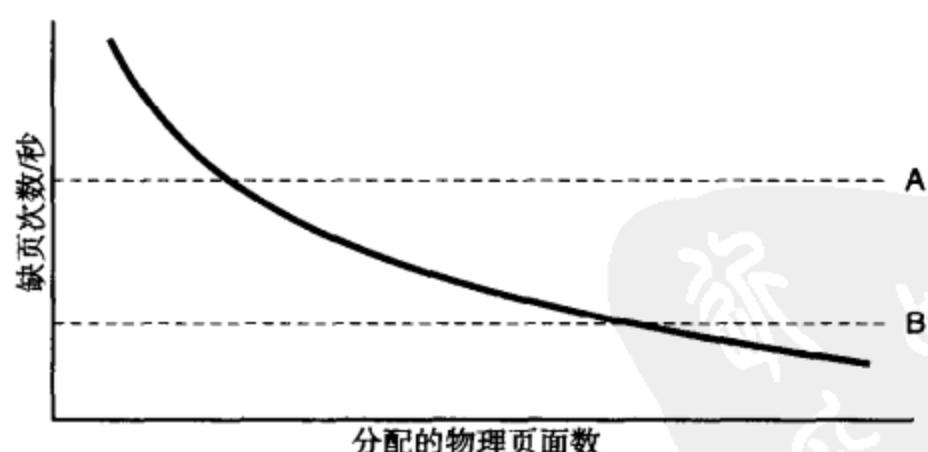


图 4.20 缺页率与分配的物理页面数之间的函数关系

缺页率的计算很简单:直接统计在一秒钟内出现的缺页次数。为了更加准确,也可以统计过去几秒钟内缺页率的一个平均值。在图 4.20 中,横坐标是分配给进程的物理页面数,纵坐标是它的缺页率,然后对于缺页率设置了两个界线,一个是下界 B,一个是上界 A。如果系统发现某个进程的缺页率太高,超出了上界 A,则可能会出现抖动现象,因此就要给这个进程分配更多的物理页面,把它的缺页率降下来。反过来,如果系统发现某个进程的缺页率太低,低于下界 B,则说明这个进

程得到的物理页面数太多了。所以就要从它的手里拿走一些页面。总之，缺页率算法试图去控制每个进程的缺页率，使它保持在一个合理的范围内。这样，既降低了进程的缺页率，又不会造成内存空间的浪费。

如果系统发现内存中的进程太多，无法使所有进程的缺页率都低于A，那么就必须把一些进程置换出去，从而腾出它们所占用的内存空间，并把这些物理页面分配给其余的进程，或是把它们放进一个空闲页面池，以用于将来的缺页中断。这种将进程移出内存的做法，实际上是一种负载控制（load control），它表明即使是在虚拟页式存储管理中，交换仍然是需要的，只不过交换的目的是为了减少潜在的内存需求，而不是像原来那样，立刻就要使用刚刚收回的内存块。将进程交换出去以缓解内存的负载压力，这种做法令人想起了二级调度，即一个调度器用于决定把哪些进程留在磁盘上，把哪些进程调入内存；而另一个调度器则是通常的CPU调度器。显然，我们可以把这种思想运用到存储管理中。

4.5.3 页面大小

在设计一个虚拟页式存储管理系统时，页面的大小是一个很重要的参数，也是一个可以自定义的参数。例如，即使在硬件设计上只支持512字节的页面，操作系统也可以通过把两个相邻的页面合并来支持1KB大小的页面。也就是说，把第0和第1页、第2和3页、第4和第5页等，视为一个页面单元。在内存分配时，每次也是分配两个连续的512字节页面。

页面大小的选择需要平衡各种相互矛盾的因素，因此，没有一个全局性的最优解。有的因素要求比较小的页面大小。比如说，页面越小，那么内碎片就越少。我们知道，在页式存储管理中，系统会自动地把一个进程划分为大小固定的页面，但对于进程的各个段来说，如代码段、数据段和栈段，它们一般都不会正好是页面大小的整数倍，也就是说，在最后一个页面内，可能没有装满，可能存在一些空闲的地方，即内碎片（internal fragmentation）。从统计规律来说，内碎片的大小一般是半个页面。因此，假设内存中有n个段，每个页面的大小为p个字节，那么总的内碎片大小就是 $np/2$ 。显然，页面越小，内碎片就越少。

另外，小页面还有一个好处。例如，假设有一个程序，它被分成8个阶段顺序执行，每阶段需要4KB内存。如果页面的大小是4KB，那么在程序的整个运行过程中，只需要4KB内存即可。也就是说，在程序的不同阶段，分别装入不同的内容。如果页面大小是16KB，那么在程序的整个运行过程中，始终占用着16KB内存。而这是没有必要的，因为当程序在执行第1阶段时，完全没有必要把第2~4阶段的内容装入内存。而当程序在执行第2阶段时，第1阶段的内容已经没什么用了，而第3、第4阶段的内容还暂且用不上。如果页面大小是32KB，那么浪费的空间就更多了。总之，如果页面比较大，那么就会使更多的、未被使用的代码或数据出现在内存中。

以上讨论的是小页面的优点，但有一些因素又要求页面比较大。比如说，如果页面越小，那么对于同一个程序来说，它就需要越多的页表项，所以页表就越庞大。一个32KB的程序只需要4个8KB的页面，却需要64个512字节的页面。另外，在内存与磁盘之间的数据传输通常是以页面为单位的，多数时间用在磁头定位和旋转延迟，因此，传输一个小页面的时间和传输一个大页面的时间是差不多的。这样一来，如果页面越小，那么需要传送的次数就越多，因而系统的开销就越大。例如，如果要装入64个512字节的页面，则可能需要 $64 \times 10\text{ ms}$ ；但如果要装入4个8KB的页面，则只需要 $4 \times 10.1\text{ ms}$ 。

在有些机器上，每次CPU从一个进程切换到另一个进程时，必须把新进程的页表装入硬件寄存器。在这种情形下，页面越小意味着页表项越多，页表越大，因此装入页表寄存器所需要的时间就越长。

对于最后这一点，可以进行数学上的分析。假设进程的平均大小是 s 个字节，页面的大小是 p 个字节，每个页表项需要 e 个字节。那么每个进程需要的页面数大约是 s/p ，占用的页表空间是 se/p 个字节。由于内碎片的原因，在最后一个页面内浪费的内存是 $p/2$ 。因此，由页表和内碎片造成的总开销是

$$\text{总开销} = se/p + p/2$$

如果页面比较小，那么上式中的第一项（页表大小）就越大；如果页面比较大，那么上式中的第二项（内存碎片）就越大。因此，最优值一定在中间某个地方。通过对 p 求导并令其等于零，我们得到方程

$$-se/p^2 + 1/2 = 0$$

从这个方程可以推出最优页面大小的计算公式（只考虑内碎片和页表所需的内存），其结果是

$$p = \sqrt{2se}$$

如果 $s = 1 \text{ MB}$, $e = 8$ 个字节，那么最优的页面大小是 4 KB 。

大多数计算机使用的页面大小在 512 字节到 1 MB 之间，典型的值是 1 KB，但现在 4 KB 或 8 KB 更常见。随着内存容量越来越大，页面的大小也越来越大。但两者之间的对应关系并不是线性的，事实上，即使内存空间增大了 4 倍，页面的大小也很少增加 2 倍以上。

4.5.4 虚拟存储器接口

到目前为止，我们一直假设虚拟存储器对进程和程序员是透明的，也就是说，他们所看到的是一个大的虚拟地址空间和一个较小的物理内存。在许多系统上这是对的，但在一些高级系统中，程序员可以对内存映射进行调整，并用一些非传统的方式来使用它，从而增强程序的行为。在本小节，我们将简单地介绍一些这方面的内容。

之所以允许程序员去管理他们的内存映射，一个原因是为了让两个或多个进程可以共享同一段内存空间。如果程序员能够对他们的内存区域进行命名，那么一个进程就有可能把自己的一块内存区域的名字告诉另一个进程，使该进程也可以把它映射进来。这样一来，如果两个（或多个）进程共享相同的页面，那么进程之间的高速数据共享就成为可能，也就是说，一个进程负责将数据写入共享内存区，而另一个进程则负责从中读出数据。

页面共享还可以用来实现高性能的消息传递系统。一般来说，在传递消息时，是把数据从一个地址空间复制到另一个地址空间，开销非常大。如果进程可以管理它们的页面映射，那么消息传递就可以这样来实现：发送消息的进程把包含该消息的页面从页面映射中删除，而接收进程则把这些页面映射进来。这样一来，只有页面的名字需要复制，而数据不用动。

还有一种高级的存储管理技术是分布式共享存储器（distributed shared memory）（Feeley et al., 1995; Li and Hudak, 1989; Zekauskas et al., 1994）。它的基本思路是允许网络上的多个进程共享同一组页面，这些页面可以（但不是必须）组成一个共享的线性地址空间。如果一个进程访问了一个未被映射的页面，就会引发缺页中断。中断处理程序（可以在内核中也可以在用户空间）随即找到持有该页面的机器，并向这个机器发送消息，让它删除对这个页面的映射，并把它从网络上发送过来。当这个页面到达后，就把它映射进来，并回到被中断的指令重新执行。

4.6 段式存储管理

到目前为止，我们讨论的虚拟存储器都是一维的，也就是说，虚拟地址是从0一直到某个最大地址，一个接一个。但是对于许多问题来说，拥有两个或多个独立的虚拟地址空间可能要比单个空间好得多。例如，编译器在编译过程中会建立许多表格，其中包括：

1. 供打印清单使用的源文本。
2. 符号表，包含变量的名字和属性。
3. 包含整型和浮点型常量的表。
4. 语法树，包含程序的语法分析结果。
5. 编译器内部用于函数调用的栈。

在这些表格中，前4个将随着编译的进行不断地增长，而最后一个在编译过程中以一种不可预计的方式增长和缩小。在一维存储器方式下，这5个表格将被存放在虚拟地址空间的连续数据块中，如图4.21所示。

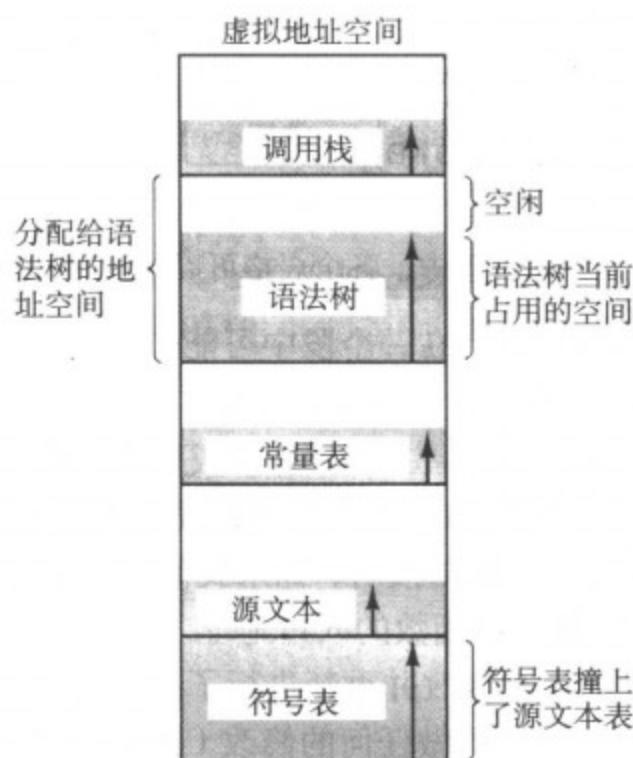


图4.21 在一维地址空间中，如果有多个动态增长的表，则一个表可能会撞上另一个

假设有一个程序，其他内容都很正常，但变量的个数非常多。这样，分配给符号表的地址空间可能会被装满，而其他表格中却还有大量的空间。在这种情形下，编译器当然可以简单地打印出一条信息说，由于变量太多编译不能继续进行。但在其他表中还有大量空间时，这样做似乎并不恰当。

另一种做法是罗宾汉式的劫富济贫，从空闲空间较多的表中拿出一些空间给空间很少的表。这种处理是可以做到的，但在具体实现时，它有点类似于自己去实现覆盖技术：在最好情形下，可能会觉得很麻烦；在最坏情形下，可能吃力不讨好，一点效果也没有。

我们真正需要的是能够把程序员从管理表格的收缩和扩张工作中解放出来的办法，就像虚拟存储器把程序员解放出来，不用再去操心如何把程序划分成一个个的覆盖块。

一种直观并且通用的方法是在机器上提供多个互相独立的地址空间，称为段（segment），在每个段的内部，是一个一维的线性地址序列，从0开始，一直到某个最大值。一般来说，每个段的长度是不相等的，而且在执行过程中，段的长度可以动态变化。例如，对于一个栈来说，当有数据被压入栈时，栈的长度就增加；当有数据被弹出栈时，栈的长度就缩短。

由于每个段都是一个独立的地址空间，因此它们可以各自独立地增长或缩减，而不会相互影响。如果某个段中的栈需要更多的空间，那么它可以直接往上增长，因为在这个地址空间中，在它的上方没有任何其他东西阻挡。当然，段也有可能会被装满，但这种情况发生可能性非常小，因为一个段通常是很大的。在这种段式或二维存储器中，一个地址包含两部分的内容：段号和段内偏移地址。图 4.22 演示了在段式存储管理中，如何来实现前面讲过的编译表格的例子。

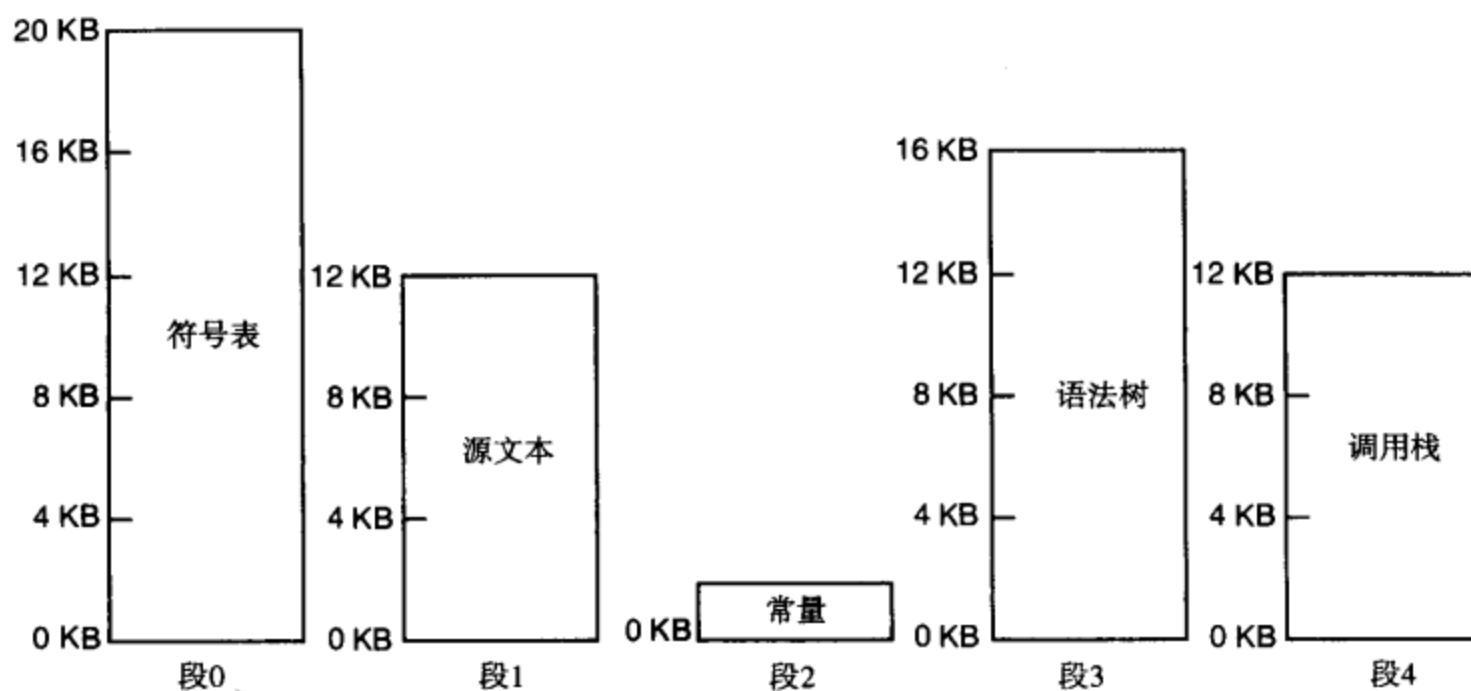


图 4.22 在段式存储管理中，各个表格可以相互独立地增长或缩减

需要强调的是，段是一个逻辑实体。在一个段中可能包含一个或多个函数，或者是一个数组，或者是一个栈，或者是一些数值变量，但它一般不会同时混杂有不同类型的内容。

除了能够简化处理动态增长或缩减的数据结构，段式存储管理还有其他一些优点。如果每个函数都位于不同的段中，并且起始地址都是 0，那么当我们需要把单独编译好的各个函数链接在一起时，就会非常简单、方便。在经过编译和链接后，如果我们要去调用第 n 个段中的那个函数，只要使用二维地址 $(n, 0)$ ，就可以跳转到该函数的入口地址。

如果第 n 个段中的那个函数随后被修改并重新进行了编译，而且新版本比旧版本要长，那么并不会影响到其他的函数，其他的函数不用做任何的修改（因为大家的起始地址都没有变）。而在一维地址中，函数被一个挨一个紧紧地放在一起，中间没有空隙，因此，如果修改了一个函数的长度，将会影响到其他函数的起始地址。这样一来，所有相关的函数调用都要进行修改，以适应这些新的起始地址。如果一个程序包含有上百个函数，那么相应的开销可能是相当大的。

分段也有助于在几个进程之间共享函数和数据，一个典型的例子就是共享库（shared library）。运行高级窗口系统的现代工作站常常要把非常大的图形库编译进几乎每个程序。在段式系统中，可以把图形库放到一个单独的段中由各个进程共享，这样就不需要在每个进程的地址空间中都留有一份。虽然在页式存储管理中也能够实现共享库，但是要复杂得多，而且它们实际上都是通过模拟分段来实现的。

如前所述，段是一个逻辑实体，如函数、数组或栈，因此，不同的段应该具有不同的保护模式。例如，一个函数段可以设定为只可执行，不允许对它进行读操作或写操作；一个浮点数组可以设定为可读可写但不可执行，如果试图跳转到这个段去执行，将会被系统捕获。这些保护机制能够帮助程序员发现程序中的错误。

与页式存储管理相比，段式存储管理中的保护更有意义，因为程序员知道每个段所包含的内容。一般来说，在一个段中只会包含一种类型的信息，如函数或栈，而不会把它们混杂在一起，既

有函数又有栈。在这种情形下，我们可以针对特定的段类型，来为它设置适合的保护模式。段式和页式存储管理的比较参见图 4.23。

从某种程度上来说，一个页面中的内容是随机的，程序员甚至觉察不到分页的存在。虽然我们可以在页表的表项中放入几个数据位，从而设定页面的访问权限，但是为了做到这一点，我们必须知道在整个地址空间中，所有的页面边界在哪里。然而，当初正是为了避免这一类琐碎的管理工作，人们才发明了页式系统。而在段式系统中，用户知道内存中的每一个段，包括它的类型和内容，因此可以对各个段分别进行保护，同时也不用去关心它们的具体管理细节。

考虑的问题	页式存储管理	段式存储管理
程序员需要知道正在使用这种技术吗？	不需要	需要
有多少个线性地址空间？	一个	多个
地址空间可以超过物理内存的大小吗？	可以	可以
数据和函数可以区别开来、分别进行保护吗？	不可以	可以
能够很容易地容纳长度经常变化的表吗？	不能	能
用户之间的函数共享是否方便？	否	是
为什么要发明这项技术？	为了在物理内存空间不变的情况下，得到更大的线性地址空间	为了能将代码和数据划分为逻辑独立的地址空间，为了帮助实现共享和保护

图 4.23 页式和段式存储管理的比较

4.6.1 纯分段系统的实现

在具体实现上，段式和页式存储系统是完全不同的：页面是定长的而段不是。如图 4.24(a)所示，在刚开始时，在物理内存中包含有 5 个段。假设现在要把段 1 淘汰出去，再把比它小的段 7 调进来，取代它的位置。这样就得到了图 4.24(b)所示的内存状态。在段 7 与段 2 之间是一个空闲区域，即一个空洞。随后段 4 被段 5 取代，如图 4.24(c)所示；段 3 被段 6 取代，如图 4.24(d)所示。这样一来，当系统运行一段时间后，内存将会被划分为许多块，其中有些是段，有些是空洞，这种现象称为跳棋盘（checkerboarding）或外碎片（external fragmentation）。由于外碎片通常都比较小，无法再装入新的段，因而就造成了内存空间的浪费。不过，这个问题可以通过内存紧缩技术来解决，如图 4.24(e)所示。

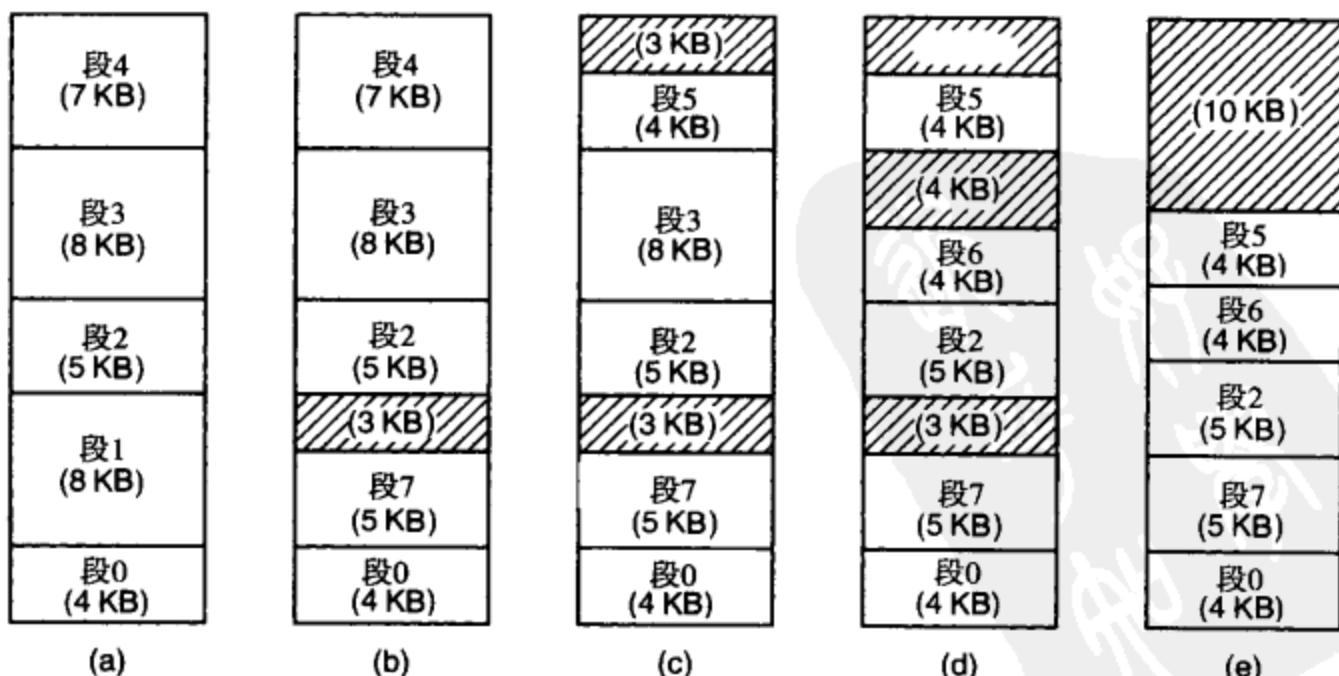


图 4.24 (a)~(d) 外碎片的形成；(e)通过内存紧缩技术消除外碎片

4.6.2 段页式存储管理: Intel Pentium

Pentium 支持 16K 个段, 每个段最多可以容纳 2^{32} 个字节的虚拟地址空间。操作系统可以对 Pentium 进行设置, 使它支持纯段式、纯页式或段页式存储管理。大多数操作系统, 如 Windows XP 和各种类型的 UNIX, 使用的是纯页式模型, 每个进程只有一个段, 长度为 2^{32} 个字节。但也有一些操作系统, 如 OS/2, 充分使用了 Pentium 的寻址功能, 从而能够访问一个更大的地址空间。

Pentium 虚拟存储器的核心是两张表, 局部描述符表 (Local Descriptor Table, LDT) 和全局描述符表 (Global Descriptor Table, GDT)。每个程序都有自己的 LDT, 但 GDT 只有一个, 为计算机上的所有程序所共享。LDT 描述的是每个程序自己的段, 包括代码段、数据段、栈段等, 而 GDT 描述的是系统的段, 包括操作系统本身。

为了访问某个段, Pentium 程序必须先把这个段的选择符 (selector) 装入到机器的 6 个段寄存器之一。在程序执行时, CS 寄存器存放的是代码段的选择符, DS 寄存器存放的是数据段的选择符, 而其他的段寄存器就稍微次要一点。每一个选择符都是一个 16 位的整数, 如图 4.25 所示。

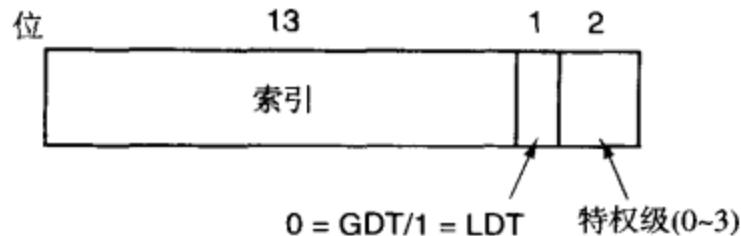


图 4.25 Pentium 的选择符

在选择符中有一位用来指出这个段是局部的还是全局的, 也就是说, 它是在 LDT 中还是在 GDT 中。另外 13 位则用做 LDT 或 GDT 的表项号, 因此, 这些表的长度不会超过 8K, 最多只能存放 8K 个段描述符。剩余的两位与保护有关, 我们将在后面讨论。在 LDT 和 GDT 中, 描述符 0 是禁止使用的, 因此我们可以在一个段寄存器中装入 0, 来表示该段寄存器暂且不指向任何段。此时, 如果硬是要使用这个段寄存器, 将会引发陷阱中断。

当一个选择符被装入某个段寄存器时, 存放在 LDT 或 GDT 中的相应的描述符就会被取出, 存放在微程序寄存器中, 以便于快速地访问。每个描述符由 8 个字节组成, 包括段的基地址、长度和其他信息, 如图 4.26 所示。

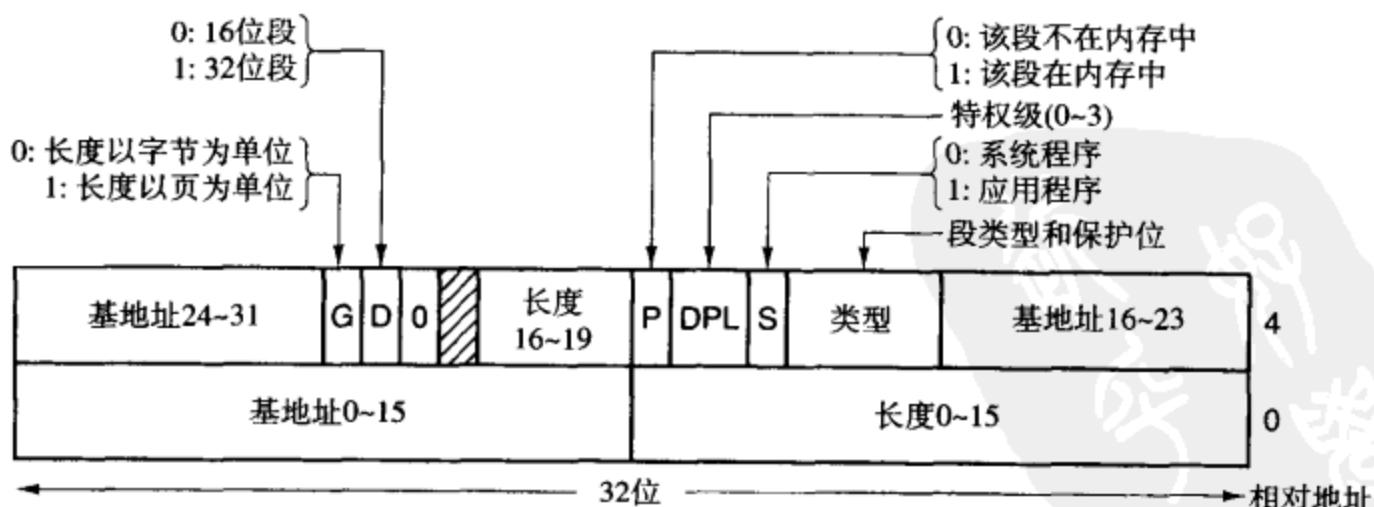


图 4.26 Pentium 代码段的描述符, 数据段略有不同

选择符的格式是经过精心设计的, 它使描述符的定位变得非常方便。首先, 根据选择符中的第二位, 可以选定相应的表格, LDT 或是 GDT; 其次, 把选择符复制到一个内部寄存器, 并把它的

低3位清0；最后，将LDT或GDT的起始地址加上去，这样就得到了目标描述符所在的位置。例如，选择符72描述的是GDT的第9个表项，它的起始地址为GDT + 72。

现在让我们来看一下，一个虚拟地址（选择符，段内偏移）是如何被转换为相应的物理地址的。只要微程序知道正在使用的是哪个段寄存器，它就能根据该寄存器的值（即某个选择符），从内部寄存器中找到相应的描述符。如果目标段不存在（即选择符为0），或者已经被换出，就会发生一次陷阱中断。

接下来，再检查一下，看段内偏移量是否超出了段的末尾，如果是也会引发一次陷阱中断。从理论上来说，在描述符中应该有一个长度为32位的字段，用来描述段的长度。但实际上我们只有20位可以使用，因此Pentium的做法是：如果gbit（粒度）字段为0，则limit（长度）字段存放的是普通的段长，以字节为单位，最大1MB；如果gbit为1，则limit字段中存放的段长以页面为单位。Pentium的页面大小是固定的4KB，因此20位的长度字段所能存放的最大段长是 $20 \times 4\text{ KB} = 2^{32}$ 字节。

如果目标段在内存中且段内偏移也在范围内，那么Pentium就会把描述符中的32位基地址与段内偏移量相加，形成所谓的线性地址（linear address），如图4.27所示。为了和只有24位基地址的286兼容，Pentium的基地址被划分为3段，分布在描述符的不同位置。实际上，有了base（基地址）字段后，一个段可以把它的起始地址设置在32位线性地址空间中的任何一个位置。

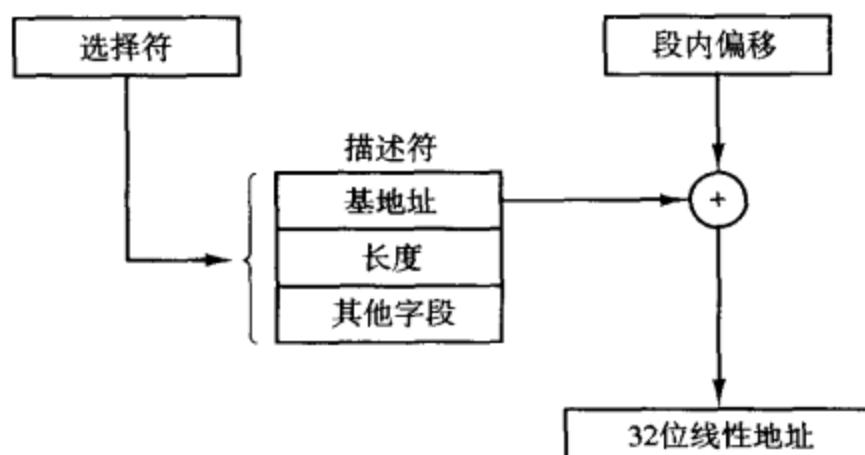


图4.27 一个虚拟地址（选择符，段内偏移）到物理地址的转换过程

如果页式存储管理被禁用（通过全局控制寄存器中的某一位），那么线性地址就直接被解释为物理地址，并送往内存用于读写。在这种情形下，我们就得到了一个纯粹的段式存储管理系统，每个段的基地址存放在它的描述符中。顺便提一句，段与段之间可能会互相重叠。这是因为如果要验证任意两个段之间都没有重叠，那么这实在是太麻烦了，时间开销太大，所以系统不进行此类检查。

如果在系统中启用分页机制，那么线性地址就被解释成一个虚拟地址，并在页表的帮助下，映射为相应的物理地址。具体过程请参见前面的例子。这里的麻烦在于：在32位虚拟地址和4KB页面的情形下，一个段可能会包含多达100万个页面，因此Pentium使用了一种两级地址映射，以减少页表的大小。

对于每一个正在运行的程序，都有一个页面目录（page directory），包含1024个32位的目录项。页面目录的起始地址存放在一个全局寄存器中。每个目录项都指向一个页表，页面中包含有1024个32位的页表项，页表项负责指向物理页面。这种方案如图4.28所示。

在图4.28(a)中，我们看到一个线性地址被分为三个字段：*dir*, *page* 和 *offset*。*dir*（目录）字段用做页面目录的索引，利用它可以找到相应的目录项，里面存放了目标页表的起始地址。在找到相应的页表后，再以*page*（页面）字段为索引去查找页表，从中得到目标物理页面的起始地址。最

后，把页内偏移量 *offset* 加上这个起始地址，就得到了最终的物理地址，然后就可以根据这个地址去访问相应的内存单元（字节或字）。

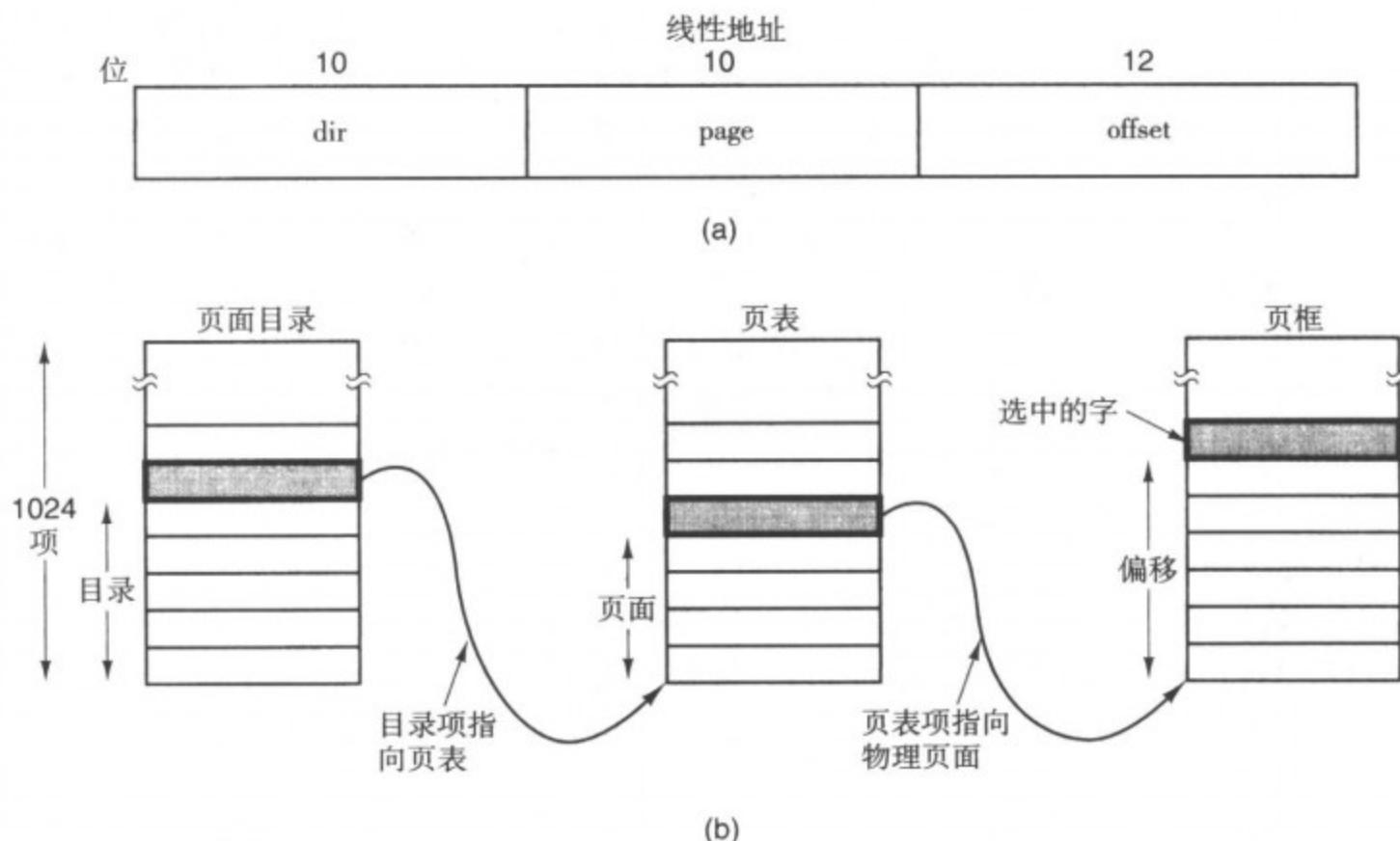


图 4.28 线性地址到物理地址的映射

每个页表项都是一个 32 位的整数，其中 20 位用于存放物理页面号，其余的位包括：访问位、修改位、保护位和其他一些有用的信息位。如前所述，访问位和修改位是由机器硬件来自动置位的，主要用于操作系统的页面置换等功能。

每个页表包含 1024 个表项，每个表项指向一个 4 KB 的物理页面，因此一个页表可以处理 4 MB 的内存。如果一个段的大小不超过 4 MB，那么在页面目录中只有一个目录项，它指向一个仅有的页表。在这种情形下，系统的开销只有两页（一个用来装页面目录，一个用来装页表）。而如果采用一级页表的方法，那么系统的开销可能是上千个页面。

为了减少页表的访问次数，提高地址映射的速度，在 Pentium 中有一个小的 TLB，能够把最近使用过的 *dir-page* 组合映射到相应的物理页面。只有当 TLB 未命中时，才会去执行图 4.28 所示的映射机制，并更新 TLB。只要 TLB 的命中率比较高，那么性能总还是不错的。

略微思考一下我们就会发现，在使用页式存储管理时，如果把描述符中的基址字段设置为一个非 0 的整数，这是毫无意义的。实际上，引入这个字段的真正原因，是为了支持纯段式管理，以及与 286 兼容（286 只有纯段式管理，没有分页机制）。

还有一点值得注意，如果一个应用程序不需要分段，它只需要一个以页面为单位的 32 位的地址空间，这也是可以做到的。我们可以把所有的段寄存器都设置为同一个选择符，而对于它所对应的那个描述符，其基地址为 0，长度设置为最大值。在这种情形下，指令中用到的偏移地址就等价于线性地址，而且只用到了一个地址空间（即为纯粹的页式存储管理）。实际上，当前 Pentium 上的所有操作系统都采用这种机制，只有 OS/2 系统曾经是一个例外，它使用了 Intel MMU 体系结构的全部功能。

不管怎么说，我们不得不称赞 Pentium 的设计者。他们要设计的这个系统具有多重的、相互矛盾的目标，既要支持纯段式，又要支持纯页式，还要支持段页式，同时还要与 286 兼容。不仅

如此，还要求系统的性能好、速度快。面对如此大的挑战，他们拿出来的这个设计的确非常简洁、干净。

在讨论完Pentium虚拟存储器的完整体系结构之后，我们再来简单地看一下它里面的保护机制。Pentium 支持 4 个保护级别，第 0 级别表示权限最高，第 3 级表示权限最低，如图 4.29 所示。对于一个正在运行的程序，在任意时刻，它总是位于其中的某一个特权级，用 PSW 中的一个 2 位的字段来表示。系统中的每个段也有一个特权级。

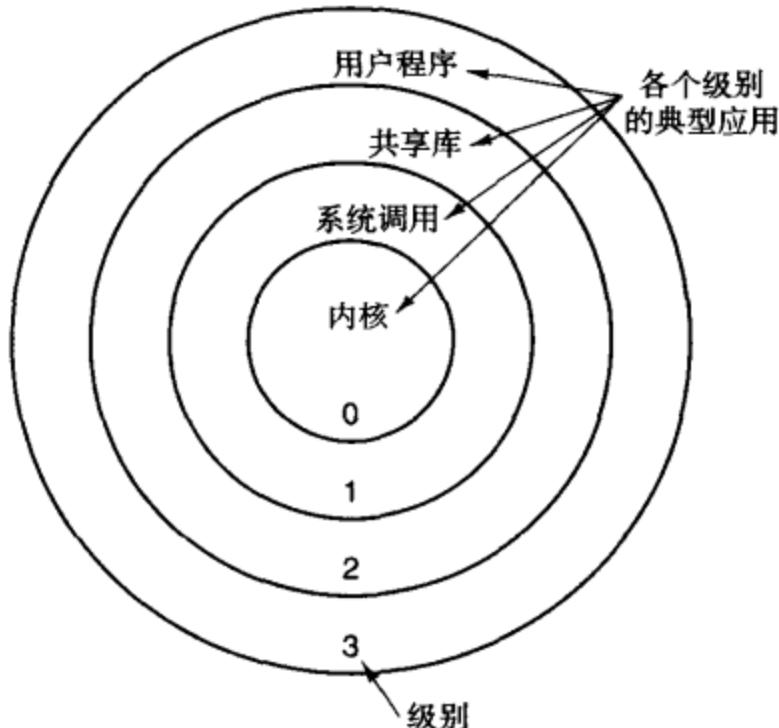


图 4.29 Pentium 中的保护机制

如果一个程序在运行时，只使用与之同级的段，那么一切都会很正常。如果它去访问更高一级的数据，这也是允许的。但如果试图去访问更低一级的数据，那么这是非法的，将会引发陷阱中断。如果是去调用不同级别（更高或更低）的函数，这是允许的，但是要通过一种严格控制的方法。为了执行越级调用，CALL 指令必须包含一个选择符而不是地址，这个选择符将指向一个称为调用门（call gate）的描述符，由它给出被调用函数的起始地址。因此，想随便跳转到某个不同级别的代码段的中间，这是不可能的，只能去使用一些正式规定的人口点。

上述机制的一种典型应用如图 4.29 所示。在 0 级是操作系统内核，负责处理 I/O、存储管理和其他一些关键事务；在 1 级是系统调用处理程序，用户程序可以通过调用这里的函数来执行系统调用。但并非这里的所有函数都能调用，系统定义了一组指定的、受保护的函数调用接口；在 2 级是一些库函数，可以由多个运行程序所共享，用户程序可以调用这些库函数，读取它们的数据，但不能去修改它们；最后，用户程序运行在级别 3 上，受到的保护最少。

陷阱和中断采用的机制与调用门差不多，它们也是使用描述符而不是绝对地址。这些描述符指向特定的、即将被执行的函数。图 4.26 中的类型字段用于区分代码段、数据段和不同类型的调用门。

4.7 MINIX 3 进程管理器概述

MINIX 3 的存储管理非常简单，它不支持页式存储管理。另外，这里的讨论也不包括交换技术。不过，在系统的源代码中有交换的相关代码，如果物理内存比较少，也可以启用这部分代码。实际上，现代计算机的内存都比较大，很少需要用到交换技术。

本章我们将讨论一个用户空间服务器：进程管理器（Process Manager, PM）。进程管理器负责处理与进程管理有关的系统调用，其中有些调用与存储管理密切相关，如 fork, exec 和 brk 等。另外一些调用则与信号处理、进程属性的设置和检查（如用户和组）、CPU 的使用时间、实时时钟的设置和查询等相关。

当我们在谈到进程管理器中与存储管理有关的内容时，我们将把它称为“存储管理器”。也许在将来的版本中，进程管理和存储管理将会彻底地分开，但在 MINIX 3 中，这两个功能模块合并同一个进程中。

PM 维护着一个空闲空间列表，根据内存地址从低到高排列。当需要内存空间时（一般是由于执行了 fork 或 exec 系统调用），系统将采用最先匹配法，从空闲列表中找到第一个符合要求的空闲区。当一个进程被装入内存后，在它的整个运行期间，它都将呆在原地不动，既不会被置换出去，也不会被移动到另一个地方，而且分配给它的内存空间也不会增加或缩小。

这种存储管理策略可能不是很常见，所以这里有必要稍微解释一下。它主要是受到以下三个方面因素的影响。首先，作为一个教学用的操作系统，我们希望整个系统是易于理解的，我们不希望把系统弄得很复杂。一般来说，源代码有 250 页左右基本上就足够了；其次，MINIX 系统最初是为 IBM PC 机设计的，在早期的 IBM PC CPU 上，甚至都没有 MMU，因此就无法实现分页机制。最后，由于同时代的其他计算机也没有 MMU，因此采用这种简单的存储管理策略后，可以很方便地把它移植到其他机器上，如 Macintosh, Atari, Amiga 等。

当然，也有人会质疑以上的这三个因素现在还有多大的意义？第一个因素当然还是有意义的，虽然经过这么多年的发展，系统已经增大了很多。另外，还有其他一些新的因素也能支持我们的观点。与早期的 IBM PC 相比，现代 PC 机的存储容量已经翻了一千多倍。虽然应用程序的规模也变大了，但一般来说，在大多数计算机上，内存都是足够的，不需要再去使用交换技术和分页技术。而且从某种程度上来说，MINIX 3 定位于一些低端系统，如嵌入式系统。如今的数码相机、DVD 播放器、立体声音响和手机等产品，都使用了操作系统，但它们都不支持交换和分页技术。MINIX 3 是面向这些领域的一个合理选择，因此它没有把交换和分页作为一个优先考虑的问题。尽管如此，我们也正在做一些这方面的尝试，看看能不能用最简单的方法，在虚拟存储管理方面做一些事情。具体的进展请随时留意我们的 Web 网站。

还有一点需要指出的是，在存储管理的实现上，MINIX 3 不同于其他的许多操作系统。也就是说，PM 不是系统内核的一部分，相反，它是运行在用户空间的一个进程，通过标准的消息机制来与内核通信。PM 在系统中的位置如图 2.29 所示。

把 PM 移出内核，这是将策略（policy）与机制（mechanism）分离的设计思路的一个具体例子。哪一个进程应该被放在内存的哪个位置，这个决定（策略）是由 PM 做出的，而对进程的内存映射进行设置和修改的工作（机制）是由内核中的系统任务来完成的。这种划分使我们可以方便地对内存的管理策略（算法等）进行修改，而不用担心会影响到操作系统的底层实现。

大部分的 PM 代码用来处理与进程创建有关的系统调用，如 fork 和 exec，而不仅仅用来管理进程列表和空闲空间列表。下一小节将介绍内存的布局，随后的几个小节将简单地看一下 PM 是如何来处理进程管理相关的系统调用的。

4.7.1 内存布局

MINIX 3 程序可以被编译为使用组合的 I 和 D 空间（combined I and D space），即进程的各个部分（代码、数据和栈）共用同一个内存块，作为一个整体来申请和释放。在 MINIX 的早期版本中，这是一种默认设置。不过，在 MINIX 3 中，默认的做法是把程序编译为使用独立的 I 和 D 空间。

(separate I and D space)。在这种方式下，能够更有效地使用内存，但也会使事情变得更复杂。为了叙述方便，我们将按照从简单到复杂的原则，先讨论简单的组合模型，然后再来讨论独立空间模型。

在 MINIX 3 中有两种情况需要分配内存。一是当一个进程执行 `fork` 时，为子进程分配所需要的内存空间；二是当一个进程通过 `exec` 系统调用修改它的内存映像时，旧的映像所占用的空间被返回给空闲列表，然后要为新的映像分配内存空间。新、旧映像所占用的内存地址可能是不同的，这取决于在什么地方能找到合适的空间区。当一个进程结束运行时（可能是主动退出，也可能是被某个信号杀死），它所占用的内存空间即被释放。此外，如果一个系统进程也来请求内存，如存储驱动程序需要为 RAM 盘申请内存空间，那么也可能会从用户进程手里剥夺一些内存，但这主要是发生在系统初始化的时候。

图 4.30 显示了 `fork` 和 `exec` 在执行过程中内存的分配情况。在图 4.30(a)中我们看到在内存中有两个进程：A 和 B。如果 A 执行了 `fork`，我们将得到图 4.30(b)所示的情形。子进程是 A 的一个完全相同的副本。如果这个子进程又调用 `exec`，执行了文件 C，那么内存的状态如图 4.30(c)所示，子进程的映像被 C 所取代。

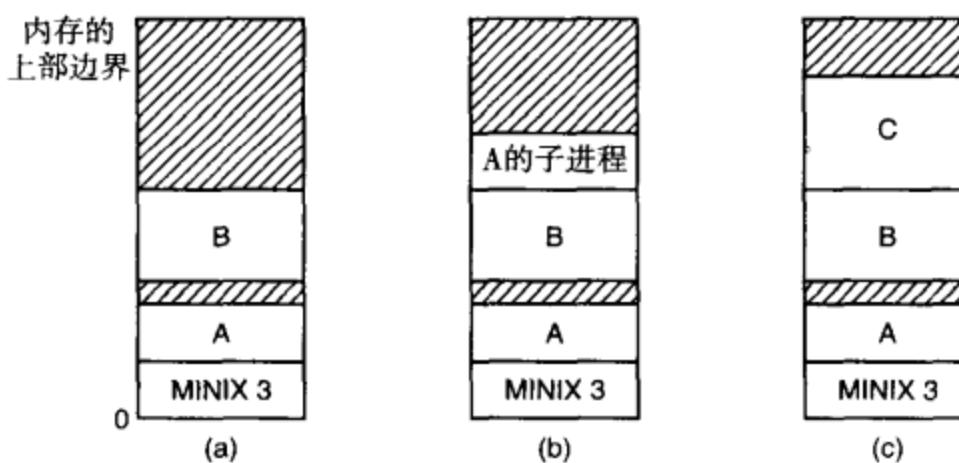


图 4.30 内存分配：(a) 初始情形；(b) 执行 `fork` 后的情形；(c) 子进程执行 `exec` 后的情形。图中的阴影部分表示空闲的内存区，进程使用的是组合的 I 和 D 地址

请注意，分配给子进程的旧内存空间先被释放，然后再为 C 分配新的内存空间，所以从图中可以看到，C 使用了子进程原先占用的那块内存。这样，在执行完一系列的 `fork` 和 `exec` 对之后，所有新生成的进程都是相邻的，中间没有空隙。如果我们在旧内存尚未释放之前就分配新内存，那么就可能会产生很多的空隙。

当 `fork` 或 `exec` 被执行后，系统就会分配一定数量的内存空间给新进程。如果是 `fork`，那么分配的内存空间的大小与父进程完全相同；如果是 `exec`，那么 PM 将从可执行文件的文件头中，获取进程的大小信息。一旦分配完毕，系统就不会再给进程分配任何内存。

上面讨论的内容都是针对使用组合的 I 和 D 空间的程序，而对于使用独立的 I 和 D 空间的程序，可以利用存储管理的一个增强模式，即共享代码（shared text）。当这样的一个进程在执行 `fork` 时，只需要为新进程分配数据段和栈段所需要的内存空间即可。而代码段则可以共享，因为这些代码已经位于父进程的地址空间中。当进程执行 `exec` 时，系统将查找进程表，看是否有另外一个进程已经在使用这个可执行文件。如果是的话，那么只需要为新进程分配数据段和栈段空间即可，而代码段可以共享。在共享代码方式下，进程的结束就变得更加复杂了。当一个进程运行结束时，数据段和栈段所占用的空间肯定要被释放，但代码段则不一定，系统先要对进程表进行搜索，如果发现没有其他进程正在使用这段代码，才能把它所占用的空间释放。因此，如果一个进程在启动时装入了自己的代码段，然后当它运行结束时，该代码段正在被其他进程所共享，那么在这种情形下，这个进程在结束时释放的内存就会比它在启动时得到的内存要少一些。

图 4.31 显示了一个程序以文件的形式存放在磁盘上的方式，以及当它被调入内存、成为 MINIX 3 进程后的内存布局。磁盘文件的文件头包含了进程映像各部分的大小以及总的大小等信息。如果一个程序被编译为使用组合的 I 和 D 空间，那么在它的程序头部，有一个字段指明了代码部分和数据部分的总长度，这些部分将被直接复制到内存映像中。映像中的数据部分还要再加上文件头中的 *bss* 字段的值，*bss* 字段主要用于未初始化的静态数据，这些数据将被初始化为 0。总共需要分配的内存数量由文件头中的 *total* 字段来说明。假如，如果一个程序有 4 KB 的代码段、2 KB 的数据段（包括 *bss* 字段），以及 1 KB 的栈，那么在文件头中将设定：需要分配的内存总量是 40 KB。也就是说，在数据段和栈段之间的空闲区域是 33 KB。另外，在磁盘的程序文件中还可以包含一个符号表，它主要用于调试，并不需要装入内存。

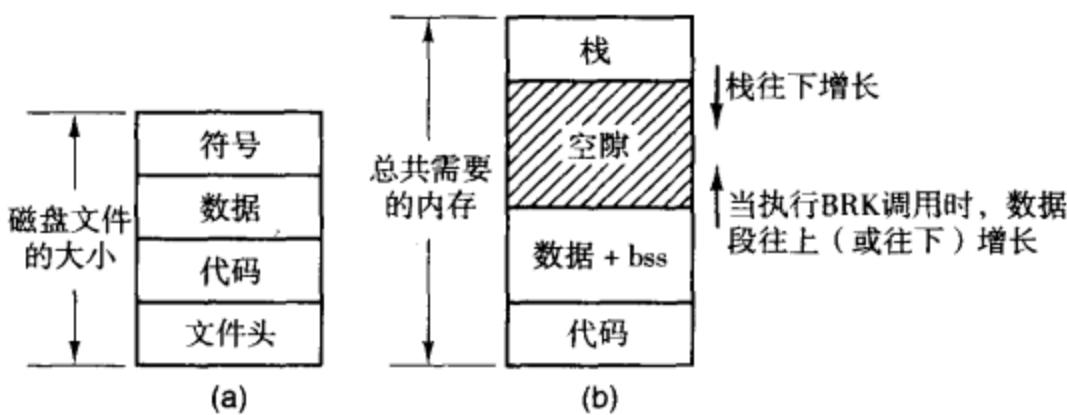


图 4.31 (a)一个保存在磁盘文件中的程序；(b)一个进程的内存布局。在这两个图中，最低的磁盘或内存地址都位于图的底端，最高地址都位于顶端

如果程序员知道，当可执行文件 a.out 在执行时，数据段和栈段的增长不会超过 10 KB，他可以使用下面的命令：

```
chmem = 10240 a.out
```

该命令将修改文件头字段，使得在执行 exec 时，PM 为这个程序分配的内存空间大小为：代码段长度 + 数据段长度 + 10 240 字节。在上面的例子中，由于代码段是 4 KB、数据段是 2 KB，因此总共需要 16 KB。在这 16 KB 中，最高端的 1 KB 用于栈，接下来的 9 KB 是空隙，它可以根据需要，用于栈和数据段的增长。

对于使用独立的 I 和 D 空间的程序（在文件头中有一个标志位，由链接器来设置），文件头中的 total 字段只对组合的数据和栈空间有用。例如，假设一个程序有 4 KB 代码、2 KB 数据和 1 KB 栈，total 字段为 64 KB，那么这个程序将被分配 68 KB 的空间（4 KB 指令空间，64 KB 栈和数据空间），其中留出了 61 KB 的空间供数据段和栈的扩展使用。数据段的边界只有通过 brk 系统调用才能修改，brk 所做的全部工作就是去检查新的数据段是否超越了当前的栈指针。如果没有，就去修改一些内部表格以反映此次变动。请注意，这个操作只是对最初分配给进程的内存区域进行调整，而不是向操作系统申请额外的内存空间。如果新的数据段闯入了栈区间，那么本次调用将以失败告终。

这里需要指出一个容易混淆的地方，当我们使用“段”这个词时，指的是操作系统定义的一个内存区域。Intel 处理器有一组内部的段寄存器和段描述符表，它们为“段”提供了硬件支持。Intel 硬件设计者所理解的段的概念类似于 MINIX 3 中定义和使用的段，但又不完全相同。在本书中所谈到的段，都是指 MINIX 3 数据结构所描述的内存区域。而当我们在谈到硬件时，将明确地使用“段寄存器”或“段描述符”。

推而广之，硬件设计者在进行硬件设计时，总希望为运行在他们机器上的操作系统提供各种支持，他们用来描述寄存器和处理器其他方面的术语，通常反映了他们的一种预见，即这些特性将来

可能会被如何使用。一般来说，这些特性对操作系统的实现者的确是有帮助的，但是在具体的使用方式上，可能和硬件设计者的初衷有所不同。这样，在描述操作系统或底层硬件的某些方面时，同一个词可能会有不同的含义，从而容易导致误解。

4.7.2 消息处理

就像 MINIX 3 的所有其他组件一样，进程管理器也是消息驱动的。在系统初始化完成后，进程管理器就进入它的主循环，包括等待消息、执行消息中包含的请求以及发送应答等。

进程管理器可以接收两种类别的消息，对于内核与系统服务器（如 PM）之间的高优先级通信，使用的是系统通知消息。这些是比较特殊的情形，将在本章后面的实现部分加以讨论。进程管理器收到的大部分消息，来源于用户进程所启动的系统调用。对于这个类别，图 4.32 中的列表给出了合法的消息类型、它们的输入参数以及在应答消息中返回的值。

消息类型	输入参数	返回值
fork	(无)	子进程的 PID (对子进程是 0)
exit	退出状态	成功时无应答
wait	(无)	状态
waitpid	(无)	状态
brk	新的长度	新的长度
exec	指向起始栈的指针	(成功时无应答)
kill	进程标识和信号	状态
alarm	等待的秒数	剩余时间
pause	(无)	(成功时无应答)
sigaction	信号编号、动作、老动作	状态
sigsuspend	信号掩码	(成功时无应答)
sigpending	(无)	状态
sigprocmask	如何、设置、老设置	状态
sigreturn	上下文	状态
getuid	(无)	用户 ID, 有效 UID
getgid	(无)	组 ID, 有效 GID
getpid	(无)	PID, 父进程的 ID
setuid	新用户号	状态
setgid	新组号	状态
setsid	新会话号	进程组
getpgrp	新组号	进程组
time	指向当前时间的存放位置的指针	状态
stime	指向当前时间的指针	状态
times	指针，指向进程和子进程时间的缓冲区	自启动以来的运行时间
ptrace	请求, 进程号, 地址, 数据	状态
reboot	方式 (停机、重启或崩溃)	(成功时无应答)
svrctl	请求, 数据 (取决于函数)	状态
getsysinfo	请求, 数据 (取决于函数)	状态
getprocnr	(无)	Proc 号
memalloc	大小, 指向地址的指针	状态
memfree	大小, 地址	状态
getpriority	进程标识, 类型, 值	优先级 (nice 值)
setpriority	进程标识, 类型, 值	优先级 (nice 值)
gettimeofday	(无)	时间, 运行时间

图 4.32 与 PM 通信时用到的消息类型、输入参数和返回值

`fork`, `exit`, `wait`, `waitpid`, `brk` 和 `exec` 显然与内存的分配和回收密切相关；`kill`, `alarm`, `pause`, `sigaction`, `sigsuspend`, `sigpending`, `sigmask` 和 `sigreturn` 等都与信号有关，但它们也可能会影响到内存，因为当一个信号杀死某个进程时，该进程所占用的内存空间将被释放。7个`get/set`调用与内存管理没有什么关系，但与进程管理有关。其他的系统调用，可以归于文件系统，也可以归于PM。之所以把它们放在这里，是因为文件系统已经足够大了。基于同样的原因，`time`, `stime` 和 `times` 也被放在这里，`ptrace` 也是，它主要用于调试。

`reboot`对整个操作系统都有影响，但它的首要任务是以一种受控的方式发送信号，以结束系统中的所有进程，因此把它放在PM中是十分合适的。类似的还有`svrctl`，它主要是用来启用或关闭PM中的交换技术。

读者可能注意到，刚才提到的两个调用，即`reboot`和`svrctl`，并没有包含在图1.9中。类似的还有图4.32中的`getsysinfo`, `getprocnr`, `memalloc`, `memfree`和`getsetpriority`。对于通常的用户进程来说，一般不会去使用这些调用，而且它们也不属于POSIX标准。之所以提供这些调用，是因为在MINIX 3这样的系统中需要。在一个整体式内核的系统中，这些调用所提供的功能可以通过内核函数来实现。但在MINIX 3中，一些通常被认为是操作系统的组件却运行在用户空间，因此就需要增加一些额外的系统调用。实际上，有些调用的功能非常简单，只是提供了一个接口，转而去执行相应的内核调用。

如同第1章所述，系统提供了库函数`sbrk`，但却没有系统调用`sbrk`。这个库函数把数据段的当前长度加上函数参数所指定的增量或减量，并以此为参数去调用`brk`。同样，对于库函数`geteuid`和`getegid`，并不存在单独的系统调用。实际上，在调用`getuid`和`getgid`时，既能返回用户和组的有效标识，又能返回它们的真实标识。同样，`getpid`返回的是调用进程及其父进程的PID。

消息处理中的一个核心数据结构是在`table.c`中声明的`call_vec`表。它包含了一些函数指针，用来指向不同类型消息的处理函数。当一条消息到达PM时，主循环会抽取出消息的类型，并把它放在全局变量`call_nr`中。然后用这个值作为索引，去访问`call_vec`，从而找到相应的消息处理函数。然后调用这个函数来执行系统调用。它的返回值被放在应答消息中送回给调用者，以报告此次调用是成功还是失败。这种机制类似于图1.16中的第7步所使用的指针表格，只不过一个运行在用户空间，另一个运行在内核空间。

4.7.3 进程管理的数据结构和算法

进程管理器用到了两个核心数据结构：进程表和空闲链表。我们将逐个讨论。

从图2.4可以看出，有些进程表字段是内核需要的，有些字段是进程管理器需要的，还有一些是文件系统需要的。在MINIX 3中，操作系统的这三部分都有自己的进程表，仅包含了自己需要的字段。为了简单起见，各个表项之间是精确对应的，因此，PM的第 k 个表项与文件系统的第 k 个表项所描述的是同一个进程。为了保持同步，在进程创建或结束时，这三个部分都要去更新它们的表，以反映最新的情况。

有些进程在内核之外是不为所知的，或者是因为它们被编译进内核，如`CLOCK`和`SYSTEM`任务；或者是因为它们是位置保持者，如`IDLE`和`KERNEL`。在内核的进程表中，这些项用负数来表示。这些表项不存在于进程管理器或文件系统进程表中，因此，严格来说，上面所说的第 k 个表格项之间的精确对应关系有一个前提，即 k 必须大于或等于0。

内存中的进程

PM的进程表称为`mproc`，其定义位于`src/servers/pm/mproc.h`中。它包含了与进程内存分配有关的所有字段，还有其他一些额外信息。其中最重要的字段是`mp_seg`数组，它包含三个数组元素，

分别用于代码段、数据段和栈段。每个数组元素是一个结构体类型，包含虚拟地址、物理地址和段长等信息，单位都是 click 而不是字节。在不同的实现中，click 的大小也是不同的。早期的 MINIX 版本把它设置为 256 个字节，而在 MINIX 3 中，一个 click 表示 1024 个字节。所有段的起始地址必须在 click 的边界上，而且段长必须是它的整数倍。

内存分配的记录方法如图 4.33 所示。在该图中，一个进程有 3 KB 代码、4 KB 数据、1 KB 的空隙和 2 KB 的栈，总共分配的内存空间是 10 KB。假设这个进程没有独立的 I 和 D 空间，那么在图 4.33(b) 中我们可以看到这三个段各自的虚拟地址、物理地址和长度字段。在这个模型中，代码段总是空的，而数据段包含了代码和数据。当进程去访问虚拟地址 0 时（包括跳转到该地址或读取该内存单元，也就是说，把它视为指令空间或数据空间），实际上访问的是物理地址 0x32000（十进制是 200 KB），这个地址位于第 0xc8 个 click 中。

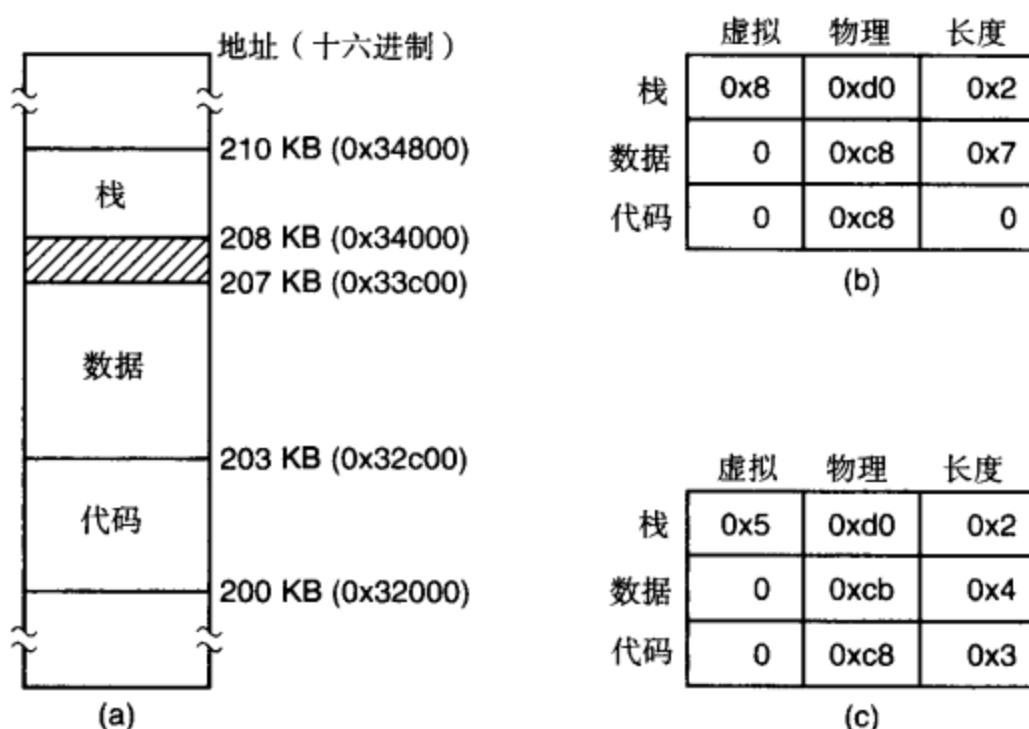


图 4.33 (a) 内存中的一个进程；(b) 使用组合的 I 和 D 空间时的内存表示；(c) 使用独立的 I 和 D 空间时的内存表示

请注意，在虚拟地址空间中，栈的起始地址取决于分配给进程的内存总量。如果我们需要更大的动态分配空间（即扩大数据段和栈段之间的空隙），可以使用 *chmem* 命令去修改文件头，这样当这个文件下次被执行时，栈将从一个更高的虚拟地址开始。如果栈的长度增加了一个 click，那么栈表项就应该从三元组 (0x8, 0xd0, 0x2) 变成三元组 (0x7, 0xcf, 0x3)。注意在这个例子中，如果分配给进程的内存空间总量没有变化，那么当栈的长度增加时，栈段与数据段之间的空隙就会变小。

8088 硬件没有栈边界陷入，在 32 位处理器上，MINIX 对栈的定义方式使得当栈在覆盖数据段之前不会引发陷入。因此，在下一次 *brk* 系统调用之前，栈表项都不会被修改。只有当 *brk* 系统调用发生时，操作系统才会去读入 SP 寄存器的值，并重新计算栈表项。如果在机器硬件上带有栈陷入功能，那么一旦栈指针跨越了它的边界，就可以立即去更新栈表项。不过，在 32 位 Intel 处理器上的 MINIX 3 并没有这样做，下面我们将讨论其中的原因。

前面我们曾经提到，硬件设计者努力提供的功能并不一定恰好能满足软件设计者的需要。即使是在 Pentium 的保护模式下，当栈指针跨越了它的段边界时，MINIX 3 也不会发生陷入。虽然在保护模式下，Intel 硬件能够检测出内存段的越界访问（由段描述符定义，如图 4.26 所示），但在 MINIX 3 中，数据段描述符和栈段描述符总是一样的。MINIX 3 的数据段和栈段各自使用这个空间

的一部分，因此两者都可以扩展到它们之间的空隙中。而且只有 MINIX 3 才能管理这种情形，CPU 则无法做到，因为对于它来说，这段空隙既属于数据段，也属于栈段，是它们地址空间中的合法区域。当然，硬件能够检查到非常大的错误，例如，对于该进程地址空间以外的区域进行访问。以上这种机制能够保护一个进程免受其他进程的攻击或非法访问，但如果在进程的内部发生错误的内存访问，则无能为力。

这里做出了一个设计决策。我们知道有人不同意放弃共享的硬件定义的段，这种段允许 MINIX 3 动态地重分配空隙区域。另一种做法是用硬件来定义不重叠的栈段和数据段，这样可能会在一定程度上提供更高的安全性，但却会使 MINIX 3 花费更多的内存空间。关于这种方法，我们也提供了源代码，任何感兴趣的人都可以去试一试。

图 4.33(c) 显示了在使用独立的 I 和 D 空间的情形下，图 4.33(a) 的内存布局。这里，代码段和数据段的长度都不为零。图 4.33(b) 或 (c) 中的 *mp_seg* 数组主要用于将虚拟地址映射成物理地址。在给定一个虚拟地址和它所属的空间后，先要检查一下，看这个虚拟地址是否合法（即是否落在段的内部），如果是的话，再来看一下相应的物理地址是多少。例如，内核函数 *umap_local* 负责为 I/O 任务实现这种映射，也为与用户空间的数据通信来实现这种映射。

共享代码段

当一个进程在执行时，它的数据段和栈段的内容可能会不断地变化，但代码段的内容是不变的。此外，在有的时候，几个进程需要执行相同的一个程序，例如，几个用户在同时执行 shell 程序。在这种情形下，我们可以使用 **共享代码段** 技术，来提高内存的利用率。当 *exec* 在装入一个进程时，它首先会打开相应的可执行文件，并读入文件头。如果该进程使用的是独立的 I 和 D 空间，那么就去 *mproc* 的每一个表项中搜索 *mp_dev*, *mp_ino* 和 *mp_ctime* 字段，这些字段包含了其他进程正在执行的各个程序的设备号、i 节点号和修改时间。如果发现即将执行的这个可执行文件与内存中某个进程所执行的程序是完全相同的，那么就没有必要去分配内存空间，装入该程序的另一份副本。我们可以把新进程的内存映射中的 *mp_seg[T]* 初始化为指向已装入的代码段的起始地址。而对于数据段和栈段，则需要为它们分配相应的内存空间。这个过程如图 4.34 所示。如果没有找到运行相同程序的进程，或者该进程使用的是组合的 I 和 D 空间，那么就按照图 4.33 的方式来分配内存，然后把新进程的代码和数据从磁盘复制到内存。

除了段信息之外，*mproc* 还保存了进程的其他一些信息，如进程及其父进程的编号（PID）、用户号（UID）和组号（GID）（包括真实的和有效的）、信号方面的信息以及进程的结束状态——如果该进程已结束而父进程尚未执行对它的 *wait* 操作。此外还有一些字段是给定时器用的，用于 *sigalarm* 和统计子进程所使用的用户和系统时间。在 MINIX 的早期版本中，由内核来负责这部分工作，但是在 MINIX 3 中，这部分的内容改由进程管理器来管理。

空闲链表

进程管理器的另一个重要的数据结构是 **空闲链表** *hole*，其定义在 *src/servers/pm/alloc.c* 中。它按照内存地址的递增顺序列出了内存中的各个空闲区域。不过，数据段和栈段之间的空隙不认为是空闲区，因为它们已经被分配给了进程。每个空闲区表项有三个字段：空闲区的起始地址（以 click 为单位）、空闲区的长度（以 click 为单位）以及一个指针，指向下一个链表结点。该链表是一个单向链表，因此，从当前结点可以很方便地找到下一个结点，但是反过来，如果要从当前结点找到上一个结点，就没这么容易，需要从链表的首结点开始逐一搜索，直到找到相应的空闲区。由于空间限制，*alloc.c* 这个源文件没有包含在书后的附录中，而是包含在随书的 CD-ROM 中。但是用来定义空闲链表的代码很简单，如图 4.35 所示。

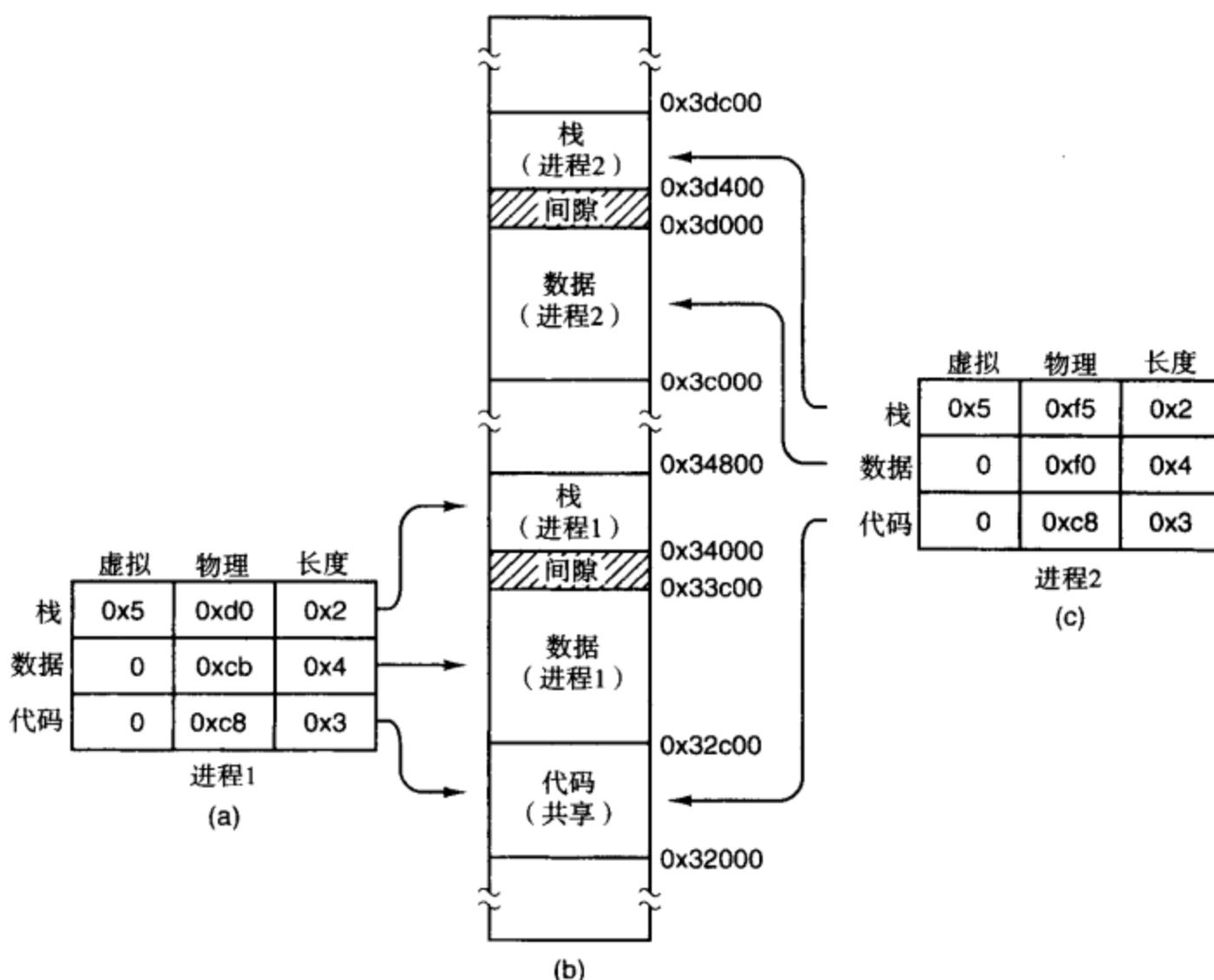


图 4.34 (a) 使用独立的 I 和 D 空间的进程的内存映射; (b) 第 2 个进程启动后的内存布局, 该进程执行的是与共享代码相同的程序; (c) 第 2 个进程的内存映射

```
PRIVATE struct hole {
    struct hole * h_next;           /* 指针, 指向链表的下一个结点 */
    phys_clicks h_base;            /* 空闲区的起始地址 */
    phys_clicks h_len;             /* 空闲区的大小 */
}hole[NR_HOLES];
```

图 4.35 空闲链表是一个 hole 结构体数组

在存储管理中,之所以使用 click 而不是字节来作为基本单位,主要是因为它的效率更高。在 16 位模式下,使用 16 位的整数来描述内存地址,因此,如果一个 click 是 1024 个字节,那么系统能够支持的最大内存为 64 MB。而在 32 位模式下,系统能够支持的最大内存为 $2^{32} \times 2^{10} = 2^{42}$ 字节,即 4 TB 或 4096 GB。

针对空闲链表的主要操作是分配一块指定大小的内存和归还一块已经分配的内存。在分配内存时,从低地址开始搜索空闲链表,直到找到一个足够大的空闲区(即最先匹配法)。随后,将空闲区一分为二,一块分配给进程,另一块是剩余的空闲区。当然,在极少的情形下,空闲区的大小可能正好等于进程所需要的大小,这时只要直接将该空闲区从链表中删除即可。这种分配方案简单快速,但可能会存在着内碎片(由于分配的最小单位是 click,因此在最后一个 click 中可能没有装满,在最坏情形下会浪费 1023 个字节的空间)和外碎片的问题。

当一个进程运行结束后,它的数据段和栈段所占用的内存区域将被归还给空闲链表。如果进程使用的是组合的 I 和 D 空间,那么它的所有内存都将被释放;如果进程使用的是独立的 I 和 D 空间,

而且在搜索进程表后发现没有其他进程在共享其代码，那么它的代码段也将被释放。在共享代码段的情形下，由于代码段和数据段未必相邻，所以归还的可能是两块内存区域。对于其中的每一块区域，如果它的左邻右舍也是空闲区，则需要将它们合并，这样就不会出现两个空闲区相邻的情形。随着系统的运行，各个空闲区的个数、位置和大小都将不断变化。当所有的用户进程都终止时，系统中的所有可用内存将重新变为空闲，等待下一次分配。当然，这些空闲内存并不一定是一块完整的连续区域，因为在物理内存中，有些区域是操作系统不能访问的。例如，在 IBM 兼容机中，需要为只读存储器（ROM）和 I/O 传输保留一段内存地址空间，因此可用的内存区域被分为两块，一块是 640 KB 以下，另一块是 1 MB 以上。

4.7.4 FORK, EXIT 和 WAIT 系统调用

在创建和撤销进程时必须为它分配或释放内存，还必须更新进程表，包括由内核和 FS 保存的部分，这些操作由 PM 来协调。进程的创建是由 fork 来完成的，相应的执行步骤如图 4.36 所示。

1. 检查进程表是否已满。
2. 试着为子进程的数据段和栈段分配内存。
3. 把父进程的数据和栈复制到子进程的内存中。
4. 找到一个空闲的进程表项并把父进程的表项内容复制进去。
5. 在进程表中填入子进程的内存映射。
6. 为子进程选择一个进程号。
7. 告诉内核和文件系统该进程的情况。
8. 向内核报告子进程的内存映射。
9. 向父进程和子进程发送应答信息。

图 4.36 执行 fork 系统调用时所需要的步骤

在执行 fork 调用时，如果中途停止，则是很困难也是很不方便的，所以 PM 维护了一个计数器，记录了当前存在的进程个数，这样就能够方便地知道是否有空闲的进程表项。如果说有的话，就尝试着为子进程分配内存。如果该程序具有独立的 I 和 D 空间，只需要为它请求新的数据段和栈段的空间。如果这一步也成功了，那么这一次的 fork 调用肯定能成功。然后就找到一个空闲的进程表项，把新分配的内存填写进来，选择一个 PID 并向系统的其他部分通报这个新进程的创建。

当以下两个事件同时发生时，一个进程才会完全终止。(1)进程自己退出（或被一个信号杀死）；(2)它的父进程执行了 wait 系统调用，等待进程的执行结果。如果一个进程已经退出或被杀死，但它的父进程尚未执行 wait 操作，那么该进程将进入一种挂起状态，有时称为僵死状态（zombie state）。这种进程不再参与调度，它的警报定时器将被关闭（如果原来是开的），它的内存也会被释放，但系统并不会把它从进程表中删除。僵死是一种临时状态，一般不会持续太长的时间。当父进程最终执行 wait 时，将释放相应的进程表项，并通知文件系统和内核。

上述方案有一个问题：对于一个即将结束的进程，如果它的父进程已经被撤销了怎么办？如果不采取特殊的处理，那么该进程将永远处于僵死状态。对于这种情况，系统将会修改进程表，使这种孤儿进程成为 init 的子进程。在系统启动时，init 进程会读取 /etc/ttymtab 文件以获得所有终端的清单，并创建一个注册进程来处理每个终端。然后它将阻塞起来，等待进程运行结束。通过这种方法，可以使孤儿进程很快被清理掉。

4.7.5 EXEC 系统调用

当从终端输入一条命令时，shell 会 fork 出一个新进程，然后由它来执行这条命令。本来，我们可以定义一个系统调用，来完成 fork 和 exec 这两个操作，但在具体实现时，这两个操作还是被分开了，原因在于：为了易于实现 I/O 重定向。当 shell 在执行 fork 时，如果标准输入是重定向的，那么子进程在执行命令前先要关闭标准输入，然后再打开新的标准输入，这样新创建的进程就可以继承重定向过的标准输入。标准输出的处理方法也是类似的。

exec 是 MINIX 3 中最复杂的系统调用，它必须用新的内存映像来替换当前的内存映像，包括设置新的栈。当然，新的映像必须是一个可执行的二进制文件，或者是一个脚本文件，可以用另一个程序来解释，如 shell 和 perl。在脚本文件的情形下，被装入的内存映像即为解释程序的二进制文件，并使用脚本文件的文件名来作为参数。在本小节，我们主要讨论比较简单的情形，即 exec 装入的是一个二进制可执行文件。稍后，当我们在讨论 exec 的实现时，会介绍一些与脚本执行有关的处理步骤。

exec 通过图 4.37 所示的一系列步骤来完成其工作。

1. 检查权限——文件是否可执行？
2. 读取文件头以获得各个段的长度和总长度。
3. 从调用者处获取参数和环境。
4. 分配新内存并释放不再需要的旧内存。
5. 把栈复制到新的内存映像中。
6. 把数据段（可能还有代码段）复制到新的内存映像中。
7. 检查并处理 setuid, setgid 位。
8. 设置相应的进程表项。
9. 告诉内核，该进程已准备运行。

图 4.37 执行 exec 系统调用时所需要的步聚

在上述步骤中，每一步都由一些更小的步骤组成，其中有些可能会失败，例如内存空间可能不够。各个步骤的执行顺序经过了仔细的选择，只有当 exec 确定会成功时，才会去释放旧的内存映像，这样就能够避免出现旧的内存映像已被释放，而新的内存映像又无法创建的尴尬境地。通常 exec 不会返回，但如果它失败了，调用进程必须再次得到控制权，并收到一个错误提示。

关于图 4.37 中的几个步骤，这里再多解释一下。第一个问题：是否有足够的空间？为了回答这个问题，首先要确定该进程需要多大的内存空间，数据段和栈的大小是确定的，而对于代码段，则需要检查一下，看看该进程是否可以和内存中的其他进程共享代码段。在确定了进程所需的空间大小后，就去搜索空闲链表，看看在释放旧的内存之前，是否有足够的物理内存。注意，我们不能先去释放旧的内存映射，否则的话，如果在旧的内存被释放后才发现内存不够用，那么我们将会处于进退两难的境地。

不过，上述的测试方法过于严格，它可能会拒绝一些原本可以成功的 exec 调用。例如，假设执行 exec 系统调用的进程占用了 20 KB，且它的代码段未被其他进程所共享。再假设有一个 30 KB 的空闲区域，而新的映像需要 50 KB。根据上述方法，即先测试后释放，我们发现只有 30 KB 的空间可用，因此会拒绝这个调用。但如果我们先释放进程原有的 20 KB，那么这个调用或许能够成功。这取决于这 20 KB 是否与 30 KB 的空间相邻，从而可以合并为一个 50 KB 的大空闲区。

如果新进程使用的是独立的 I 和 D 空间，还有另外一种做法，即分别搜索两个空闲区，一个用于代码段，一个用于数据段，而且这两个段不需要是相邻的。

另一个比较细微的问题是可执行文件是否能放进虚拟地址空间。这个问题的原因在于内存的分配单位不是字节，而是长度为1024字节的click。由于整个内存管理都以click为单位，因此每个click只能属于一个段，不能出现一半是数据、一般是栈的情形。

下面通过一个例子来说明该限制可能带来的麻烦。我们知道，在16位Intel处理器（8086和80286）上，地址空间被限制为64 KB。如果click的大小为1024，那么总共允许64个click。假设一个具有独立的I和D空间的程序，其代码段的大小为40 000字节，数据段的大小为32 770字节，栈段的大小为32 760字节。那么根据系统的要求，数据段需要占用33个click，其中最后一个click只用了2个字节，但整个click还是隶属于数据段的一部分。栈空间需要32个click。这样一来，这两部分空间加起来总共需要65个click，这就超出了系统的上限。但实际上，如果光看这两个段所需要的字节数，那么并没有超出64 KB的范围。从理论上讲，这个问题在所有click大于1字节的机器上都存在，但实际上，在Pentium系列的处理器上，由于它们支持比较大的段（4 GB），因此这个问题很少出现。

另一个重要的问题是如何来设置栈的初始状态。通常用来调用exec的库函数是execve，它的形式是

```
execve(name, argv, envp);
```

这三个参数都是指针类型，其中，*name*指向即将被执行的文件名；*argv*指向一个指针数组，每个数组元素指向一个参数；*envp*也指向一个指针数组，每个数组元素指向一个环境字符串。

在实现exec时，最简单的做法是直接把上述三个指针放在一条消息中，然后把它传给进程管理器，让PM自己去获取文件名和另外两个数组，这样PM不得不一个接一个地去取回各个参数和环境字符串。如果是这样，那么对于每一个参数和字符串，至少需要一条消息来处理，因为PM事先无法知道各个参数或串的长度。

为了避免使用很多的消息来读取各个参数和环境串，MINIX采用了一种完全不同的策略。也就是说，由execve库函数自己来构造整个初始栈，然后把它的起始地址和长度传给PM。在用户空间中构造这个新栈是非常快的，因为对参数和环境串的访问都是局部的内存访问，不涉及到其他地址空间。

为了更清楚地解释这个机制，我们来看一个例子。假设用户向shell键入了

```
ls -lf.c g.c
```

shell将对这条命令进行解释，并调用库函数

```
execve("/bin/ls", argv, envp);
```

这两个指针数组的内容如图4.38(a)所示。接下来，在shell地址空间中的函数execve将创建初始栈，如图4.38(b)所示。这个栈将在exec的执行过程中被原封不动地复制到进程管理器。

当栈最后被复制到用户进程时，它不是被放在虚拟地址0，而是放在所分配的内存空间的顶端，而内存空间的大小是由可执行文件头中的total字段来决定的。例如，我们假设总长度是8192字节，这样程序可以访问的最后一个字节位于地址8191。进程管理器负责重新定位栈中的各个指针，这样当栈中被存入新的数据后，栈的状态如图4.38(c)所示。

在exec调用执行结束、程序开始运行时，栈的内容与图4.38(c)完全相同，栈指针的当前值是8136。不过，还有一个问题需要解决。对于可执行文件来说，其主程序可能是这样声明的：

```
main(argc, argv, envp);
```

对于C编译器来说，main只是一个普通的函数，它并不知道main有何特别之处，因此在编译该函数时，以为该函数的三个参数将按照标准C函数调用的习惯（即最后一个参数放在最前面）被传递

到栈中，所以就按这个假设来访问这三个参数。也就是说，这三个参数（一个整数和两个指针）将被压入栈中，位于返回地址之前。但实际上，图 4.38(c)中的栈的内容显然并非如此。

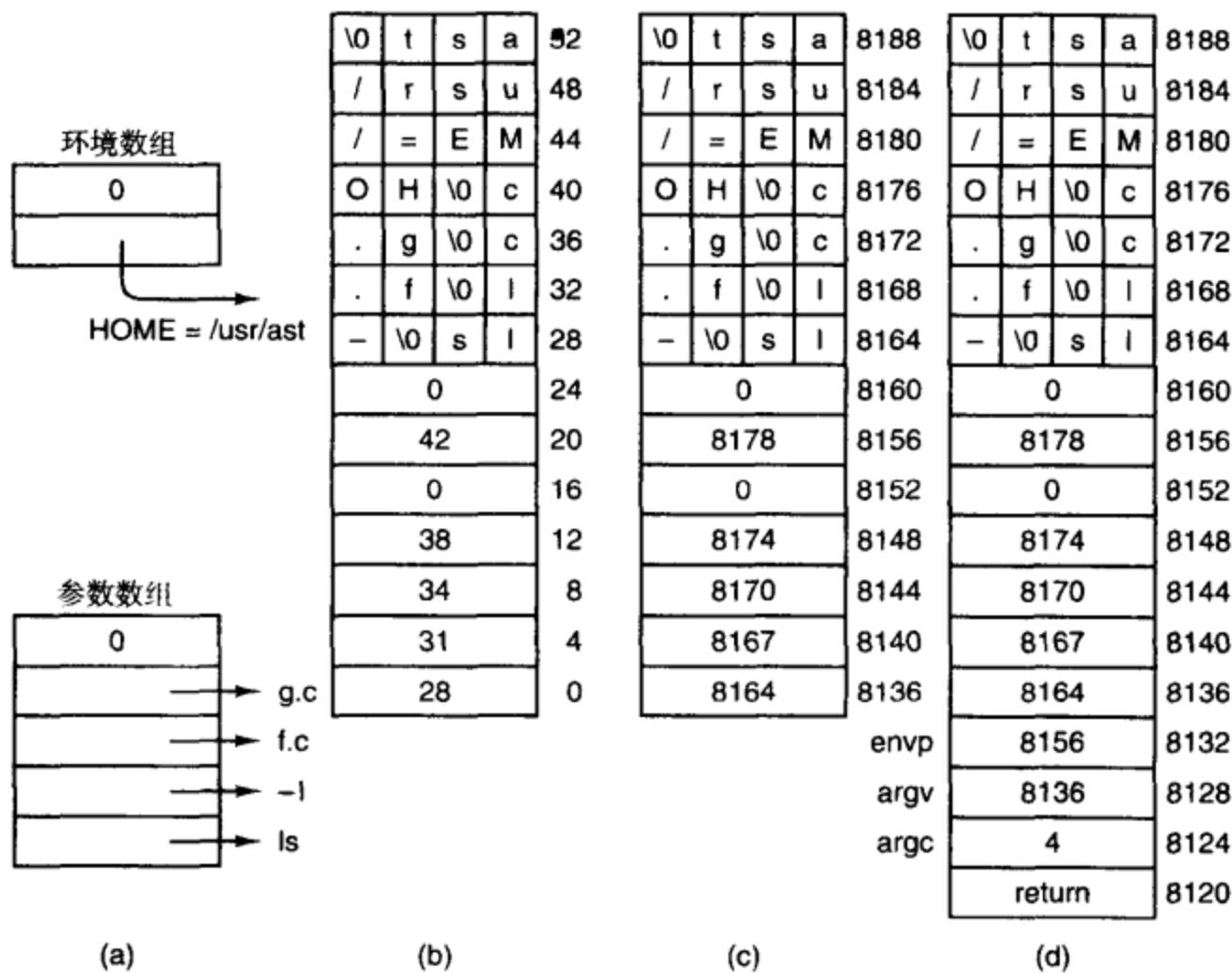


图 4.38 (a)传递给 *execve* 的两个数组; (b) *execve* 创建的栈; (c)PM 重定位后的栈; (d)当 *main* 函数开始执行时所看到的栈

解决办法就是让程序不从 *main* 开始执行，而是把一小段特别的汇编语言函数（称为 C 运行起始函数，C run-time start-off, crtso）链接到代码段的地址 0 处，使它能够首先被执行。这段函数的任务就是把三个字（即上述的三个参数）推入栈中，然后使用标准的函数调用指令去调用 *main* 函数。这样，当 *main* 开始运行时，栈的内容如图 4.38(d)所示。而这种状态正是 *main* 函数所需要的，因此它就被欺骗了，以为它是以通常的方式被调用的。

如果程序员在 *main* 结束时忘了去调用 *exit*，那么控制流就会回到运行起始函数 crtso。与前面一样，编译器只是把 *main* 视为一个普通的函数，因此在它的最后一条语句后面生成通常的函数返回指令。这样，*main* 函数将返回到它的调用者：crtso，由它来调用 *exit* 退出。对于一个 32 位的 crtso，其主要的代码如图 4.39 所示。其他未包含进来的代码，包括环境的初始化、装入一些寄存器、设置一个标志位（表示浮点协处理器是否存在）等。完整的源代码请参见文件 *src/lib/i386/rts/crtso.s*。

```

push ecx          ! environ 入栈
push edx          ! argv 入栈
push eax          ! argc 入栈
call _main        ! 调用 main ( argc, argv, envp )
push eax          ! 把退出状态压入栈中
call _exit
hlt              ! 如果 exit 失败则强制陷入
  
```

图 4.39 crtso 的主要内容

4.7.6 BRK 系统调用

库函数 `brk` 和 `sbrk` 用来调整数据段的上边界。前者的参数是绝对长度（以字节为单位），并以此来调用 `brk` 系统调用；后者的参数是相对于当前长度的正的或负的增量，并由此计算出新的数据段长度，然后调用 `brk` 系统调用。如前所述，库函数与系统调用之间并不是一一对应的，比如说，不存在 `sbrk` 这样的一条系统调用。

一个有趣的问题是：“`sbrk` 如何知道数据段的当前长度呢？”答案是：有一个变量 `brksize`，它存放的就是数据段的当前长度，所以 `sbrk` 可以去访问它。在初始化这个变量时，需要考虑到代码段和数据段的初始大小（组合的 I 和 D 空间）或者只需考虑数据段的大小（独立的 I 和 D 空间）。这个变量的名字及其相应的符号是依赖于具体编译器的，所以在源文件目录的任何头文件中都找不到它的定义，它被定义在库中，即在文件 `brksize.s` 中。不过，它的具体位置依赖于具体的系统，一般来说它与 `crt0.o` 位于相同的目录中。

对于进程管理器来说，执行 `brk` 是非常简单的，它的工作包括：检查各个部分是否仍在地址空间中，修改各种表格，并通知系统内核。

4.7.7 信号处理

在第 1 章中，信号（signal）被描述成一种信息传递机制，而收到消息的进程未必在等待输入。在系统中定义了一组信号，每个信号都有一个默认的动作——或者杀死收到信号的进程，或者忽略这个信号。当然，如果只有这两种可能，那么信号处理将是很容易理解和实现的。但实际上，进程可以通过系统调用来改变信号的响应方式。进程可以请求忽略所有的信号（除了特殊的 `sigkill` 信号），进程也可以不使用信号的默认处理方式，具体来说，它可以请求将一个信号处理函数（signal handler）与某个信号绑定在一起（`sigkill` 除外），这样，当该信号发生后，就会被捕获下来，并跳转到相应的信号处理函数去执行。对于程序员而言，操作系统对信号的处理可以分为两个不同的阶段：一个是准备阶段，此时进程可以去修改它对信号的响应方式；另一个是响应阶段，此时信号已经发生，相应的动作也已经被激活执行。刚才讲了，这个动作可能是去执行一段用户编写的信号处理函数。实际上还有第三个阶段，如图 4.40 所示。当一个用户编写的信号处理函数运行结束时，一个特殊的系统调用会去清理并恢复收到信号的进程的正常操作。当然，程序员并不需要知道第三个阶段，他在编写一个信号处理函数时，与编写其他的函数是差不多的，操作系统会负责具体的细节，包括处理函数的启动和终止、栈的管理等。

准备阶段：给信号准备相应的程序代码
响应阶段：信号已收到，相应的动作被执行
清理阶段：恢复进程的正常操作

图 4.40 信号处理的三个阶段

在准备阶段，进程可以在任何时间使用若干个系统调用，去改变它对信号的响应方式。其中最常用的是 `sigaction`，用它可以去设置信号的处理方式，包括忽略某些信号、捕获某些信号（用进程内部、用户自定义的信号处理函数来替换默认处理）或者恢复对某个信号的默认处理方式。另一个系统调用 `sigprocmask` 可以阻塞一个信号，把它暂时放在一个队列中，只有当进程在后来某个时候解除了对这个信号的阻塞后，存放在队列中的信号才会被唤醒执行。用户可以在任何时候使用这些调用，甚至是在一个信号处理的过程中。在 MINIX 3 中，信号处理的准备阶段完全由 PM 处理。因为所有需要的数据结构都位于 PM 的进程表中。对于每一个进程，都有几个 `sigset_t` 变量，这些都

是位图，每一个可能的信号由它们中的一个数据位来表示。一个这样的变量定义了一组需要忽略的信号集，另一个变量定义了一组需要捕获的信号集，依次类推。对于每一个进程，还有一个 *sigaction* 结构体数组，每个数组元素代表一个信号。如图 4.41 所示，在 *sigaction* 结构体中，一个变量用于保存用户自定义的信号处理函数的起始地址；另一个 *sigset_t* 类型的变量用于设定被阻塞的信号。用来保存处理函数地址的字段，也可以存放一些特殊的值，表明该信号即将被忽略或以默认的方式来处理。

```
struct sigaction{
    __sighandler_t sa_handler;           /* SIG_DFL, SIG_IGN, SIG_MESS,
                                             或指向函数的指针 */
    sigset_t sa_mask;                   /* 在程序执行期间需要阻塞的信号 */
    int sa_flags;                      /* 特殊的符号位 */
}
```

图 4.41 *sigaction* 结构体

注意，一个系统进程，如进程管理器本身，无法捕获信号。系统进程使用了一个新的处理函数类型 *SIG_MESS*，告诉 PM 通过 *SYS_SIG* 通知消息的方式把信号向后传。对于 *SIG_MESS* 类型的信号，不需要信号的清理过程。

当一个信号发生时，可能会涉及到 MINIX 3 系统中的多个部分。响应阶段起始于 PM 中，它使用刚刚谈到的数据结构，判断出哪个进程应该得到这个信号。如果该信号应该被捕获，那么就必须把它传递给目标进程。这时，就需要保存当前进程的各种状态信息以便将来能够恢复正常运行。这些信息被保存在接收进程的栈上，因此必须检查是否有足够的栈空间。因为这是进程管理器的领地，所以这个检查由它来执行，随后 PM 将调用内核中的一个系统任务把这些信息压入栈中。系统任务还要去处理进程的程序计数器 PC，从而使进程能够执行信号处理函数的代码。当信号处理函数结束时，将执行一个 *sigreturn* 系统调用。通过这个调用，PM 和内核共同参与，去恢复进程的信号上下文和寄存器，使进程可以恢复正常运行。如果信号不需要捕获，默认的操作将被执行，这可能需要调用文件系统来生成一个内核映像转储（core dump，即把进程的内存映像写到一个文件里，用于程序员来查找错误），或者杀死进程，这涉及到进程管理、文件系统和内核。最后，由于一个信号可能需要传送给一组进程，所以 PM 可以去管理这些操作的重复执行。

MINIX 3 中的信号定义在 *include/signal.h* 中，这是 POSIX 标准所要求的文件。这些信号列在图 4.42 中。POSIX 规定的必选信号在 MINIX 3 中都定义了，但有些可选信号没有包含进来。例如，POSIX 定义了一些与作业控制有关的信号，即把一个正在运行的程序放在后台运行或把一个后台程序调回前台。MINIX 3 不支持作业控制，但对于那些使用了这些信号的程序，可以方便地把它们移植到 MINIX 3 上。也就是说，如果这些信号出现在系统中，它们将会被忽略。作业控制的初衷是提供一种方式，使得用户在启动一个程序后，可以转而去做别的事情。而在 MINIX 3 中，采用的是另外一种做法。在启动一个程序后，用户可以敲击 ALT+F2 切换到另一个虚拟终端，去做其他的事情。虚拟终端是一种不那么优雅的窗口系统，但有了它以后，就不再需要作业控制和相关的信号了。另外，MINIX 3 还定义了一些内部使用的非 POSIX 信号和一些 POSIX 名词的同义词，以便于与旧的源代码兼容。

在一个传统的 UNIX 系统中，信号可以通过两种方式产生：由 *kill* 系统调用产生或由内核产生。在 MINIX 3 中，有些原本是内核来做的事情现在由用户空间中的进程来完成。图 4.42 显示了 MINIX 3 中的所有信号以及它们的来源。*sigint*, *sigquit* 和 *sigkill* 这三个信号可以通过按下特殊的按键组合来触发。*sigalarm* 由进程管理器来管理，*sigpipe* 由文件系统产生。*kill* 程序可以用来向任何

进程发送任何一个信号。有些内核信号依赖于硬件支持。例如，8086/8088 处理器不支持对非法指令操作码的检测，但 286 及以上的处理器都有这个能力，如果在它们上面执行非法指令，将会发生陷入。这种服务是硬件提供的，而操作系统的实现者必须提供一些代码，用于在陷入发生时产生相应的信号。在第 2 章我们看到在 *kernel/exception.c* 中包含了这种代码，它能处理不同的情形。因此，如果 MINIX 3 运行在 286 或更高的处理器上，那么在处理一条非法指令时，将发出一个 **sigill** 信号。但如果系统运行在早期的 8088 处理器上，则不会产生这个信号。

信号	描述	产生者
SIGHUP	挂断	KILL 系统调用
SIGINT	中断	TTY
SIGQUIT	退出	TTY
SIGILL	非法指令	内核 (*)
SIGTRAP	跟踪陷入	内核 (M)
SIGABRT	异常退出	TTY
SIGFPE	浮点异常	内核 (*)
SIGKILL	杀死进程（不能被捕获或忽略）	KILL 系统调用
SIGUSR1	用户自定义的信号 1	不支持
SIGSEGV	段违例	内核 (*)
SIGUSR2	用户自定义的信号 2	不支持
SIGPIPE	向无人读取的管道写入数据	FS
SIGALRM	警报时钟，时间到	PM
SIGTERM	kill 发出的软件终止信号	KILL 系统调用
SIGCHLD	子进程结束或停止	PM
SIGCONT	如果已停止则继续	不支持
SIGSTOP	停止信号	不支持
SIGTSTP	交互式停止信号	不支持
SIGTTIN	后台进程想要读	不支持
SIGTTOU	后台进程想要写	不支持
SIGKMESS	内核消息	内核
SIGKSIG	内核信号未决	内核
SIGKSTOP	内核关机	内核

图 4.42 POSIX 和 MINIX 3 定义的信号。（*）标记的信号依赖于硬件的支持。（M）

标记的信号在 POSIX 中没有定义，但 MINIX 3 为了与老的源码兼容定义了该信号。内核信号是 MINIX 3 特有的信号，由内核产生，用于向各个系统进程通报系统事件的发生。一些过时的名字和同义词没有在这里列出

硬件在某种条件下能够陷入，这并不意味着操作系统的实现者一定能够充分利用这个特性。例如，在 286 以上的 Intel 处理器上，有好几种违反内存完整性的操作都会导致异常，位于 *kernel/exception.c* 中的代码会把这些异常转换为 **sigsegv** 信号。对于违反硬件设定的栈的界限和其他段的界限，将会产生不同的异常，因为它们可能需要进行不同的处理。然而，在 MINIX 3 中，由于内存的使用方式的原因，使得硬件无法检测到所有可能发生的错误。硬件为每个段都定义了一个基地址和一个长度，而进程的栈段和数据段被组合在同一个硬件段中。硬件定义的数据段基地址和 MINIX 3 的数据段基地址是相同的，但是硬件定义的数据段边界比 MINIX 3 中使用的边界要高。换句话说，硬件定义的数据段指的是在栈空间为 0 的情形下，MINIX 3 能够用于数据的最大内存容量。同样，硬件定义的栈段指的是在数据段的长度为 0 的情形下，MINIX 3 的栈能够使用的最大内存容量。尽管硬件能够检测到某些违法访问，但它无法检测到最有可能发生的栈溢出，即栈空间被扩展到了数据段所在的区域。原因很简单：就硬件寄存器和描述符表来说，数据段和栈段是重叠的。

一种可以想到的解决方法是：在内核中加入一些代码，每一次当进程得到CPU去运行时，对进程的寄存器内容进行检查。如果发现了违反MINIX 3所定义的数据段或栈段的完整性的情形，就发出一个 `sigsegv` 信号。不过，这种做法是否值得目前还不清楚。硬件能立刻俘获一个违法访问，而软件检查可能需要执行几千条指令后才能发现，此时，对于一个信号处理函数来说，已经不能为错误恢复做什么了。

不论信号来自什么地方，进程管理器的处理方式都是差不多的。对于每一个即将收到信号的进程，需要进行一系列的检查以确定这个信号是否可行。信号的发送并不是随意的，不能想发就发，在信号的发送者与接收者之间存在着一些限制。只有当发送者是超级用户，或者它的真实或有效用户号等于接收者的真实或有效用户号时，此次信号发送才是可行的。此外还有其他一些限制条件，比如僵死的进程不能接收信号。另外，如果一个进程已经调用 `sigaction` 来忽略某个信号，或者调用 `sigprocmask` 来阻塞某个信号，那么就不能向该进程发送这个信号。阻塞信号与忽略信号不同，在收到一个被阻塞的信号后，该信号将被保存下来。这样，当接收进程取消了相应的阻塞时，就会对该信号进行处理。最后，如果接收信号的进程没有足够的栈空间，那么它将被杀死。

如果上述的所有条件都满足了，信号才能被发送出去。如果进程尚未设置去捕获这个信号，那么就不需要向进程传递任何信息。在这种情形下PM将执行这个信号的默认动作，通常是杀死进程，可能也会产生一个内核映像文件。对于少数信号，它们的默认动作是忽略信号。图 4.42 中标记为“不支持”的信号是 POSIX 要求定义的，但是在 MINIX 3 中将被忽略。

捕获一个信号意味着执行进程自己的信号处理代码，它的起始地址保存在进程表的 `sigaction` 结构中。在第 2 章我们曾经看到当一个进程被中断时，进程表项中的栈帧如何收到所需要的信息，以便于将来进程的重新执行。通过修改用户进程的栈，使得当信号处理函数运行结束时，将执行 `sigreturn` 系统调用。这个系统调用不会在普通的用户代码中发出，它的调用方式比较特别，需要内核把它的起始地址放到栈上，而且要刻意安排，使得该地址成为信号处理函数结束时的返回地址。`sigreturn` 恢复该进程的原始栈帧，使它能够从被信号中断的地方恢复运行。

虽然发送信号的最后一个阶段是由系统任务来完成的，但是由于数据是从进程管理器传到内核的，所以在这里总结一下执行的过程是很恰当的。捕获一个信号的过程类似于进程的上下文切换，即一个进程被剥夺 CPU，然后让另一个进程去运行。这里也是一样，在信号处理函数运行结束后，进程应该像什么都没有发生一样继续往下执行。不过，这里的问题可能更复杂一点，原因在于：为了将进程恢复到某个初始状态，需要记录它的所有 CPU 寄存器的当前内容。但由于进程表中的空间有限，只能存放一份这样的副本。解决的方法如图 4.43 所示。图中(a)部分是一个进程在中断发生后，它的栈和进程表项中的部分内容。在进程被挂起时，所有 CPU 寄存器的内容都被复制到进程表项中的栈帧内，这就是信号发生时的状态。由于信号的发送者是不同于接收者的另一个进程或任务，因此，此时信号的接收进程还无法运行。

在准备处理信号的过程中，进程表中的栈帧被复制到目标进程的栈中，作为一个 `sigcontext` 结构，从而将它保存起来。然后把一个 `sigframe` 结构放在栈中，这个结构包含了在信号处理函数结束后 `sigreturn` 将要使用的信息。还包含了调用 `sigreturn` 的库函数的起始地址（返回地址 1）和被中断的进程将来要恢复运行的地址（返回地址 2）。不过，后面我们会看到，返回地址 2 在正常运行时并没有用到。

程序员在编写信号处理函数时可以把它视为一个普通的函数，但是在调用它时却不能使用通常的 `call` 指令。位于进程表栈帧中的指令指针（即程序计数器）被修改，这样当 `restart` 把接收信号的进程投入运行时，将会跳转到该信号处理函数去执行。图 4.43(b)所示是这项准备工作已完成，且信

号处理函数开始运行时的情形。请记住，信号处理函数也是一个普通的函数，因此当它结束时，“返回地址 1”将被弹出，然后 `sigreturn` 将被执行。

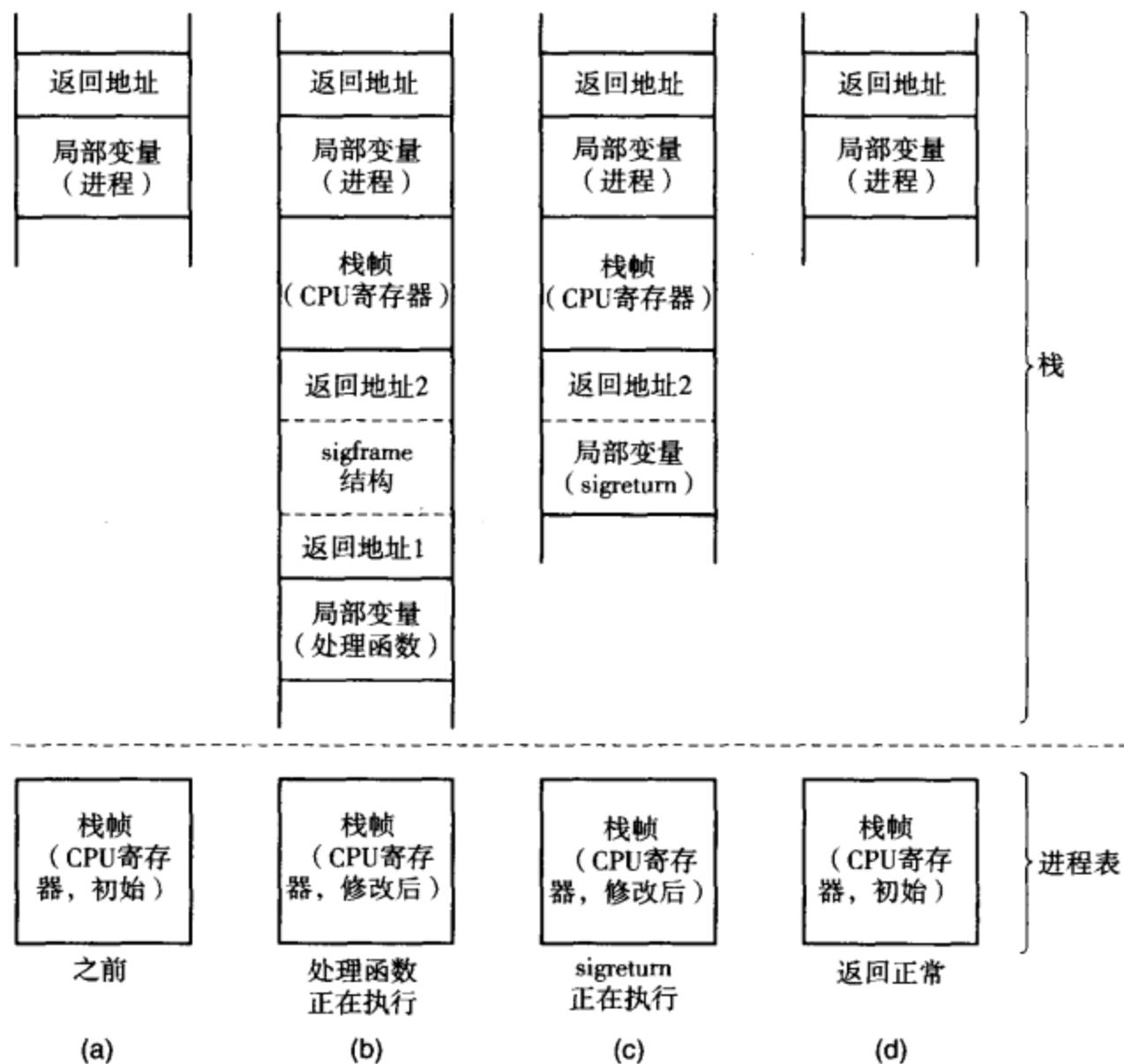


图 4.43 在处理一个信号的不同阶段，进程的栈（上图）和它的进程表中的栈帧（下图）：(a)当进程刚刚被剥夺 CPU 时的状态；(b)信号处理函数开始执行时的状态；(c) `sigreturn` 在执行时的状态；(d) `sigreturn` 执行完后的状态

图 4.43(c)显示的是 `sigreturn` 正在执行时的情形。`sigframe` 结构的剩余部分现在是 `sigreturn` 的局部变量。`sigreturn` 的部分工作是调整它自己的栈指针，使它看起来将要像一个普通函数那样结束，它使用“返回地址 2”来作为它的返回地址。不过，`sigreturn` 实际上并不是这样结束的。与其他系统调用一样，当它结束时允许内核中的调度程序来决定重启哪一个进程。最后，接收信号的进程将被重新调度，并从这个地址开始执行，因为这个地址同样也存放在进程的初始栈帧中。之所以把该地址放在栈中，是因为用户可能想要用调试器来跟踪程序的运行。当一个信号处理函数被跟踪时，这样做能欺骗调试器并给它一个关于栈内容的合理解释。在每个阶段，栈的内容看起来都像一个普通进程的栈，即局部变量位于返回地址之上。

`sigreturn` 的任务是把各个部分恢复到接收信号之前的状态，并进行相应的清理工作。最重要的是使用保存在目标进程的栈中的副本，将进程表中的栈帧恢复到它原来的状态。`sigreturn` 运行结束时的状态如图 4.23(d)所示，从中可以看出正在等待恢复执行的进程的状态与它刚被中断时的状态相同。

大部分信号的默认动作是杀死接收信号的进程，包括默认情形下不能被忽略的信号、接收进程未提出请求去处理、阻塞或忽略的信号等，这件事情是由进程管理器来负责的。如果父进程正在等

待（即执行 `wait` 操作），那么被杀死的进程将被清理并从进程表中删除；如果父进程没有等待，那么它将进入僵死状态。对于某些信号（如 `SIGQUIT`），PM 还会把进程的内核映像文件写入当前目录。

有时可能会出现这样的情形：一个进程由于正在等待 I/O 操作（如读文件）而处于阻塞状态，这时另一个进程给它发送了一个信号。在这种情形下，如果该进程没有声明去捕获这种信号，那么它将被杀死，这和普通的进程是一样的。但如果信号被捕获了，那么问题就出现了：在信号中断被处理完之后应该怎么办？进程是继续等待呢，还是去执行下一条语句？

在 MINIX 3 中是这样处理的：系统调用将被终止并返回错误代码 `EINTR`，这样进程可以知道调用是被一个信号中断的。确定一个进程是否被阻塞并不是一件很简单的事，进程管理器必须请求文件系统进行检查。

POSIX 建议可以使用这种处理方法，但并不要求，它也允许 `read` 操作在接收到一个信号时返回它在接收到信号时已经读到的字节数。而返回 `EINTR` 可以使我们设置一个时钟警报并捕获 `sigalarm` 信号，这是一种简单的实现超时的方法。例如，如果用户在指定的时间内没有响应，那么就终止 `login`，并挂断调制解调器。

用户空间定时器

信号最经常的用法是生成一个警报，在一定的时间后唤醒某个进程。在通常的操作系统中，警报完全是由系统内核来管理的，或者是由运行在内核空间的时钟驱动器来管理的。而在 MINIX 3 中，负责向用户进程发出警报的是进程管理器。我们的思路是尽量减轻内核的负载，并简化内核空间的代码。一般认为，在每 h 行的源代码中，不可避免地会出现 b 个程序错误，因此，内核中的代码越少，那么错误的个数也就会越少。当我们把更多代码放在用户空间后，即使错误的总数保持不变，那么这些错误对系统造成的破坏程度也肯定比在内核中要小。

那么我们在处理警报的时候，能否完全不依赖于内核空间的代码呢？至少在 MINIX 3 中，答案是否定的。警报首先由内核空间的时钟任务来管理，它维护了一条定时器链表，或者说队列，如图 4.29 所示。在每个时钟中断中，将位于队列最开头的定时器的过期时间与当前时间进行比较，如果过期了，那么时钟任务的主循环就被激活，它将向请求警报的进程发送一个通知。

MINIX 3 的创新之处在于内核空间的定时器仅用于系统进程，而进程管理器会维护另外两个定时器队列，用来处理用户进程的警报请求。进程管理器在向时钟发出警报请求时，每次只考虑队列首部的那个定时器。因此，如果用户进程提出了一个新的定时器请求，而该请求并未添加在队列的首部，那么此时 PM 并不急于向时钟发出请求（当然，警报请求也是通过系统任务来发出的，因为时钟任务并不会直接和其他进程通信）。在一个时钟中断后，如果检测到了一个过期的警报，进程管理器就会收到一个通知。然后它就会检查它自己的定时器队列，并向相应的用户进程发送信号。如果在链表中还有其他的警报请求，就向时钟提出下一个申请。

到目前为止，听起来好像系统内核并没有省太多的事情，但还有其他的一些考虑。首先，在一个特定的时钟节拍内，可能会有多个定时器同时到期。当然，两个进程同时请求警报的情形是不太可能的，不过，虽然在每次中断中，时钟都会去查看是否有定时器过期，但在有些时候，中断可能会上被关闭。例如，在调用 PC BIOS 时可能会错过不少的中断，这意味着时钟任务所维护的时间可能会跳越几个节拍，从而使得多个定时器过期，需要同时处理。在这种情形下，如果由进程管理器来处理，那么内核空间的代码就不需要遍历它自己的链表，重新清理并发出多个通知。

第二，警报可能会被取消。用户进程可能会在它设定的定时器过期之前，取消这次定时。或者，在有的时候，进程需要等待某个可能永远也不会发生的事件，为了避免出现永远等待的情形，进程可能会设置一个定时器。但是当该事件的确发生时，这个定时器就可以被取消了。显然，如果

定时器的取消操作由进程管理器来完成，而且在它自己的队列上完成，那么内核代码的负担就大大减轻了。对于内核空间的队列来说，它只有在两个时刻需要注意：一是队列首部的定时器过期；二是进程管理器修改了它的队列首结点。

如果我们对警报处理的相关函数进行快速的浏览，那么定时器的实现机制可能就更容易理解了。由于警报处理涉及到众多的函数，包括进程管理器中的函数和内核中的函数，因此，如果我们逐一地去讨论每一个函数，那么就会只见树木，不见森林，缺乏一个整体性的、宏观的了解。

当 PM 为某个用户进程设置一个警报时，可以用 *set_alarm* 来初始化定时器。定时器结构包括几个字段：过期时间、请求本次警报的进程以及一个指针，该指针指向一个待执行的函数。对于警报，该函数通常是 *cause_sigalarm*。然后，系统任务被要求设置一个内核空间的警报。当这个定时器过期时，内核中的看门狗进程 *cause_alarm* 将被执行，并向进程管理器发送一个通知。这涉及到几个函数和宏定义，但最终这个通知被 PM 的 *get_work* 函数收到，并在 PM 的主循环中被检测为 *SYN_ALARM* 消息类型，然后再调用 PM 的 *pm_expire_timers* 函数。接下来，在进程管理器的空间中，又用到了几个函数。库函数 *tmrs_exptimers* 将启动看门狗 *cause_sigalarm* 去执行，而后者调用了 *checksig*，它又调用了 *sig_proc*。接下来，*sig_proc* 将决定是去杀死该进程，还是给它发送 *SIGALRM*。最后，在发送信号的时候，需要内核空间的系统任务帮忙，因为需要对进程表中的数据以及目标进程的栈空间中的数据进行操作，如图 4.43 所示。

4.7.8 其他的系统调用

PM 还负责处理其他一些比较简单的系统调用。*time* 和 *stime* 处理实时时钟，*times* 调用可以获得进程的统计时间信息。之所以在这里对它们处理，主要是因为把它们放在 PM 中是比较方便的（在第 5 章介绍文件系统时，我们将讨论另外一个与时间相关的调用 *utime*，因为这个调用负责把文件的修改时间保存在 i 节点中）。

库函数 *getuid* 和 *geteuid* 都调用 *getuid* 系统调用，在它的返回信息中包括这两个值。类似地，*getgid* 系统调用也返回真实值和有效值，以供 *getgid* 和 *getegid* 函数使用。*getpid* 也是类似的，它返回的是当前进程的编号及其父进程的编号，而 *setuid* 和 *setgid* 都可以在一个调用中同时设置真实值和有效值。在这一组调用中还有两个调用，*getpgrp* 和 *setsid*。前者返回进程的组号，后者把它设置为当前的进程号。这 7 个调用是 MINIX 3 中最简单的系统调用。

ptrace 和 *reboot* 系统调用也由进程管理器来处理，前者支持程序的调试，后者影响到系统的很多方面。把它放在 PM 中是合适的，因为它的第一个动作就是杀死除 init 以外的所有进程，然后调用文件系统和系统任务来完成它的工作。

4.8 MINIX 3 进程管理器的实现

在了解了进程管理器的基本工作原理后，让我们看看它的代码本身。PM 全部是用 C 编写的，代码很直观，而且包含有很多注释，所以大部分内容都不需要冗长而深入的介绍。我们将首先简单地看一看头文件，然后是主程序，最后是一些文件，里面包含了上面所讨论的各组系统调用。

4.8.1 头文件和数据结构

PM 源程序目录中的几个头文件和内核目录中的文件具有相同的名字，这些名字在文件系统中还将再一次见到。这些文件在它们各自的上下文中有相似的函数。之所以采用这种平行的结构，是为了使整个 MINIX 3 系统的组织结构更容易理解。在 PM 中还有许多头文件，它们的名字是独一无

二的。与系统的其他部分一样，当PM版本的*table.c*被编译时，会为全局变量保留存储空间。在本小节我们将看到所有的头文件以及*table.c*。

和MINIX 3的其他主要模块一样，PM也有一个主头文件*pm.h*（17 000行）。在每一个编译中都会把它包含进来，而它自己则包含了每个目标模块都需要的位于*/usr/include*及其子目录下的所有系统级的头文件。*kernel/kernel.h*中包含的大部分文件在这里也被包含进来。PM还需要*include/fcntl.h*和*include/unistd.h*中的一些定义。另外，PM自己的*const.h*, *type.h*, *proto.h*和*glo.h*版本也被包含进来。在内核中我们看到过类似的组织结构。

const.h（17 100行）定义了PM用到的一些常量。

*type.h*目前没有使用，它只是以一个框架的形式存在，使PM和MINIX 3的其他部分具有同样的组织结构。*proto.h*（17 300行）收集了在整个PM中需要用到的函数原型。在它的第17 313行和17 314行，定义了一些哑函数，当我们把交换模块编译进MINIX 3时，需要用到这些函数。把这些宏定义放在这里的优点是：当我们需要编译一个无交换模块的版本时，就会很方便。否则的话，我们就必须对许多其他的源文件进行修改，把所有调用了这些函数的地方都要删掉。

PM的全局变量在*glo.h*中（17 500行）声明。在内核中用过的关于*EXTERN*的技巧也用在了这里，也就是说，除了文件*table.c*以外，*EXTERN*这个宏通常被扩展为*extern*。而在文件*table.c*中，它将变成一个空串，这样，使用*EXTERN*来声明的变量将被分配存储空间。

这些变量中的第一个变量*mp*，是指向一个*mproc*结构的指针。对于一个其系统调用正在被处理的进程，该结构描述了进程表中的PM部分。第二个变量*procs_in_use*记录了当前正在使用的进程表项数，这样在*fork*系统调用中就能够很方便地知道此次创建新进程能否成功。

消息缓冲区*m_in*用于请求消息。*who*是当前进程的索引，它与*mp*的关系是

```
mp = &mproc[who];
```

当一条消息到达时，系统调用号被抽取出来放在*call_nr*中。

当一个进程异常终止时，MINIX 3会把该进程的映像写入一个内核映像转储（core dump）文件，*core_name*定义了这个文件的名字。*core_sset*是一个位图，表明哪些信号应该产生内核映像文件。*ign_sset*也是一个位图，表明哪些信号应该被忽略。注意*core_name*被定义为*extern*而不是*EXTERN*。数组*call_vec*同样被声明为*extern*。之所以要采用这种方式来声明这两个变量，我们将在讨论*table.c*时再做解释。

进程表的PM部分位于另一个文件*mproc.h*中（17 600行）。大部分的字段都用注释进行了恰当的描述。其中有几个字段和信号处理有关，例如，*mp_ignore*, *mp_catch*, *mp_sig2mess*, *mp_sigmask*, *mp_sigmask2*和*mp_sigpending*等，它们都是位图，其中每个位代表一个可以送往进程的信号。类型*sigset_t*是一个32位的整数，因此MINIX 3最多能支持32个信号，但当前只定义了22个。信号1是最低（最右）位。在任何情形下，POSIX都要求提供标准的函数，以添加或删除这些位图所表示的信号集成员，因此所有必要的操作都可以在程序员不知道细节的情形下完成。数组*mp_sigact*对信号处理是非常重要的，每一种信号类型都有一个相应的数组元素，每个元素是一个*sigaction*结构（定义在*include/signal.h*中），每个*sigaction*结构由三个字段组成：

1. *sa_handler*字段定义信号的处理方式，是按照默认方式、忽略还是用专门的程序来处理。
2. *sa_mask*字段的类型是*sigset_t*，它定义了当这个信号正在被用户的自定义程序所处理时，哪一些信号将被阻塞。
3. *sa_flags*字段是一组用于信号处理的标志位。

这个数组使得信号的处理具有很大的灵活性。

mp_flags 字段用来保存一些杂七杂八的标志位。这个字段是一个无符号整数，在低端CPU上它是 16 位，在 386 及以上的 CPU 上则是 32 位。

进程表中的下一个字段是 *mp_procargs*。当一个新进程启动时，系统将创建一个类似于图 4.38 的栈，而且把指向新进程的 *argv* 数组起始地址的指针存放在那里，这将被 *ps* 命令使用。例如，对于图 4.38 的例子，值 8164 将被保存在这里。这样，对于 *ps* 来说，它可能会显示如下的命令行：

```
ls -l f.c g.c
```

mp_swapq 字段主要用在交换模块中，用来指向一组正在等待被换出的进程队列。*mp_reply* 字段用于存放应答消息。在 MINIX 的早期版本中，只提供了一个这种字段，定义在 *glo.h* 中，当 *table.c* 被编译时，它也会被阻塞起来。在 MINIX 3 中，为每个进程提供了一个建造应答消息的空间。在每个进程的进程表项中提供一个应答的地方，这使得当 PM 不能马上发出应答消息时，能够继续去处理另一个到来的消息。PM 不能一次同时处理两个请求，但如果有必要的话，它可以推迟消息的应答。然后，当它完成一次请求后，再试图把所有未决的应答消息都发送出去。

进程表项中的最后两项可以认为是一种点缀。*mp_nice* 用于存放进程的 *nice* 值，用户可以使用它来降低自己进程的优先级，比如说，暂停当前正在运行的进程，让出 CPU，让另一个更重要的进程先运行。不过，在 MINIX 3 中，这个字段主要在内部使用，用来给不同的系统进程（服务器和驱动程序）赋予不同的优先级。*mp_name* 字段主要用于调试，它可以帮助程序员在一个内存转储中找到某个进程表项。然后可以使用一个系统调用在进程表中搜索某个进程的名字并返回相应的进程号。

最后，请注意进程表的 PM 部分被定义为一个长为 *NR_PROCS* 的数组（17 655 行）。大家如果回忆一下，可以知道在 *kernel/proc.h* 文件中，进程表的内核部分被定义为一个长为 *NR_TASKS+NR_PROCS* 的数组（5593 行）。如前所述，对于用户空间的操作系统组件（如进程管理器）来说，它们并不知道有哪些进程被编译进系统内核。

下一个文件是 *param.h*（17 700 行），它包含了一些宏，用于请求消息中的许多系统调用参数，还有 12 个宏是用于应答消息中的字段，3 个宏是用于文件系统的相关消息。例如，如果语句

```
k = m_in.pid;
```

出现在一个含有 *param.h* 的文件中，那么 C 预处理器会把它转换为（17 707 行）

```
k = m_in.m1_i1;
```

在介绍可执行代码之前，我们先来看一下 *table.c*（17 800 行）。在编译这个文件时，将会为我们 *glo.h* 和 *mproc.h* 中看到的各种 *EXTERN* 变量和结构保留存储空间。语句

```
#define _TABLE
```

将使 *EXTERN* 成为一个空串，这种技巧我们在内核代码中曾经看到过。如前所述，在 *glo.h* 中，*core_name* 被声明为 *extern*，而不是 *EXTERN*。原因在于，在这里 *core_name* 在声明的时候带有一个初始化字符串，而如果使用 *extern* 就不能对它进行初始化。

table.c 的另一个主要成分是数组 *call_vec*（17 815 行）。它也是一个带有初始值的数组，因此在 *glo.h* 中不能被声明为 *EXTERN*。当一个请求消息到达时，其中的系统调用号将被取出作为 *call_vec* 的索引，从而找到处理这个系统调用的函数。对于无效的系统调用号，将会执行 *no_sys*，它仅仅返回一个错误码。值得注意的是，虽然 *_PROTOTYPE* 宏用在了 *call_vec* 的定义中，但这并不是用来声明一个原型，而是定义了一个带有初始值的数组。只不过由于这个数组是一个函数数组，因此我们使用了 *_PROTOTYPE* 宏，以便于兼容经典 C（Kernighan & Ritchie）和标准 C。

关于头文件的最后一点评论：由于 MINIX 3 仍在不断向前发展，所以有些地方的边界可能有些模糊。例如，*pm/* 目录下的一些源文件包含了内核目录下的某些头文件。如果你对此不太了解，可能会找不到一些重要的定义。按道理来说，在 MINIX 3 系统中，同时被多个主要模块所使用的那些定义，应该把它们统一放在 *include/* 目录下的头文件中。

4.8.2 主程序

PM 的编译和链接是独立于内核和文件系统的，因此它有自己的主程序。当内核完成初始化后，该主程序即开始运行。主程序的入口点在 *main.c* 中的第 18 041 行，它首先调用 *pm_init* 来完成自己的初始化，然后就在第 18 051 行进入一个循环。在这个循环中，它将调用 *get_work* 来等待请求消息的到来。然后通过 *call_vec* 表，调用某个 *do_XXX* 函数来完成本次请求。最后，如果需要的话就发送一个应答消息。读者对这种结构应该很熟悉了，它和 I/O 任务所用到的结构是完全一样的。

当然，上面的描述进行了一些简化。在第 2 章曾经提到，通知消息（notification message）可以被发送给任何一个进程，这种消息的特征是 *call_nr* 字段里面存放的是特殊的值。在 18 055 行到 18 062 行，对于 PM 可能收到的两种通知消息进行了测试，并对这两种情形采用了特殊的操作。另外，在 18 064 行，在真正执行本次请求（18 067 行）之前，需要对 *call_nr* 的有效性进行测试。虽然不太可能出现无效请求的情形，但最好还是要做一下测试。一则这个测试的开销很小，二则万一碰上无效的请求，那么后果就会比较严重。

另外一点需要说明的是第 18 073 行的 *swap_in* 调用。我们在介绍 *proto.h* 时曾经说过，就本书所描述的 MINIX 3 而言，交换模块未被包含进来，因此相关的函数都被定义为哑函数，没有实质的内容。但如果我们在编译系统的时候，把所有的源代码都包含进来，包括交换模块，那么在这个地方就必须进行一个测试，看看一个进程能否被交换进来。

最后，虽然 18 070 行的注释表明该处是发回一个应答消息，但也是一种简化。*setreply* 调用会在我们前面谈到过的当前进程的进程表项中构造一个应答，然后在 18 078 行到 18 091 行的循环中，进程表中的所有表项都会被检查一遍，所有能发的应答都会被发出去，只留下那些暂时还不能发出的应答。

函数 *get_work*（18 099 行）和 *setreply*（18 116 行）分别处理实际的接收和发送。前者用了一点小技巧，使得来自于 PM 本身的消息看起来就像是来自于内核。而后者并不会真正发送应答，就像我们刚才所说的，它只是把应答设置好，呆会儿自有人把它发送出去。

进程管理器的初始化

main.c 中最长的函数是 *pm_init*，其功能是对 PM 进行初始化。当系统开始运行后，该函数就没有什么用了。在系统中，虽然各个设备驱动程序和服务器是被单独编译的并作为单独的进程运行。但在装入系统时，有些程序是由引导监控程序来装入的，作为引导映像（boot image）的一部分。我们知道，对于任何一个操作系统而言，如果在启动时没有 PM 和文件系统的支持，那么是比较困难的。因此，这些组件最好由引导监控程序把它们统一装入到内存。另外，有些设备驱动程序也需要作为映像的一部分被装入。当然，从 MINIX 3 的角度来说，它希望各个驱动程序之间能够相互独立，但有时无法完全做到这一点。例如，在系统启动时，磁盘驱动程序必须尽早被装入内存，否则就会影响到后续的操作。

pm_init 的主要工作是初始化 PM 的各种表格，使得所有预先装入的进程都能够运行。如前所述，PM 维护了两个重要的数据结构：**空闲链表**（空闲内存表）和进程表的一部分。先来看空闲链表。由于内存的初始化是比较复杂的，如果我们先来看一下 PM 被启动时的内存组织结构，那么对于后面的叙述可能就会更容易理解了。

在 MINIX 3 的引导映像本身被装入内存之前，引导监控程序会确定内存的布局。从引导菜单上，用户可以按下 ESC 键，从而看到各种引导参数。其中一行显示的是空闲内存的块数，可能是这个样子：

```
memory = 800:923e0, 100000:3df0000
```

（在 MINIX 3 启动后，用户可以使用 *sysenv* 命令或 F5 键看到上述信息。当然具体的数字可能会有变化。）

这条信息显示了两块空闲内存区域，换言之，也有两块已被使用的内存区域。内存地址在 0x800 以下的区域，用于存放 BIOS 数据，并被主引导记录和引导块使用。至于它是如何被使用的，这并不重要，我们只要知道，在引导监控程序开始执行时，这块内存区域是不能使用的。从 0x800 开始的空闲内存区域，是 IBM 兼容机的基本内存。在本例中，从地址 0x800 (2048) 开始，有一块大小为 0x923e0 (599 008) 字节的空闲区。在这块区域上面，从 640 KB 到 1 MB 的“上部内存区域”，是普通程序不能访问的，它是为 ROM 和 I/O 专用内存而保留的。最后，从地址 0x100000 (1 MB) 开始，是一块大小为 0x3df0000 字节的空闲区域，这块区域通常称为“扩展内存”。以上信息表明，这台计算机的内存容量是 64 MB。

在上面的例子中，我们可以注意到，在基本内存区域 (640 KB 以下)，空闲空间的大小不超过 638 KB。MINIX 3 引导监控程序会把它自己装入到这段区间内的尽可能高的位置，它可能需要 52 KB。这样，真正剩下来的只有 584 KB 左右的空闲空间。这里我们要插一句话，在具体应用中，内存的使用可能比这个例子要复杂得多。例如，一种运行 MINIX 系统但又无须移植到 MINIX 的方法，就是使用一个 MS-DOS 文件来模拟一个 MINIX 磁盘。这种技术需要在启动 MINIX 3 的引导监控程序之前，先装入 MS-DOS 的一些模块。如果这些模块被装入的内存区域没有和其他的已占用的区域相邻，那么引导监控程序在报告空闲内存时，又会增加两块空闲区域。

当引导监控程序把引导映像装入内存时，关于这个映像各个部分的信息就会显示在控制台屏幕上。图 4.44 是其中的一部分显示内容。在本例中（本例是一个典型例子，但它和你所看到内容可能并不完全一致，因为它来源于 MINIX 3 的一个发布前的版本），引导监控程序把内核装入到起始地址为 0x800 的空闲内存区域。PM、文件系统、服务器等其他一些未列出的模块，被装入到起始地址为 1 MB 的空闲区域。这种设计选择是比较随意的，事实上，在 588 KB 地址以下，还有不少的空闲空间，可以装入一些模块。不过也不是所有模块都能装得下，例如，如果在编译 MINIX 3 时，把它设置为支持大的块缓冲区（如本例所示），那么文件系统就太大了，没有办法装入到内核下面的空闲区。当然，最简单的办法，就是把内核之外的所有模块都装入到扩展内存中，而且这样做也不会损失什么，因为一旦系统正在运行，用户进程也被启动，那么 588 KB 以下的那块内存区域就能够被使用。

cs	ds	代码段	数据段	bss	栈段	
0000800	0005800	19552	3140	30076	0	内核
0100000	0104c00	19456	2356	48612	1024	pm
0111800	011c400	43216	5912	6224364	2048	fs
070e000	070f400	4352	616	4696	131072	rs

图 4.44 引导监控程序显示的部分内存使用情况

在初始化 PM 时，首先用一个循环语句来遍历进程表，把每个表项中的定时器关闭，这样就不会被虚假的警报所干扰。然后，将一些全局变量进行初始化，这些变量定义了将被忽略或引发内核

映像转储的默认信号集合。接下来，对内存使用信息进行处理。在第 18 182 行，系统任务得到我们刚才看到的引导监控程序所显示的内存字符串。在本例中，有两个（基地址：大小）对，显示了两块空闲内存区域。函数调用 *get_mem_chunks* (18 184 行) 将 ASCII 码字符串中的数据转换为二进制数据，然后把基地址和大小输入到 *mem_chunks* 数组 (18 192 行)。这个数组的元素是如下定义的：

```
struct memory { phys_clicks base; phys_clicks size;};
```

mem_chunks 还不是空闲链表，它只是一个小的数组，用来收集相应的信息，为空闲链表的初始化做好准备。

接下来，需要查询内核并把内核的内存使用信息转换为以 click 为单位。然后调用 *patch_mem_chunks* 把内核的内存使用情况从 *mem_chunks* 数组中减去。*mem_chunks* 并不是完备的，引导映像中的普通进程所使用的内存将由 18 201 行到 18 239 行中的循环来说明，该循环对各个进程表项进行了初始化。

引导映像中的各个进程的属性信息存放在映像表中，该表是在 *kernel/table.c* 中声明的 (6095 到 6109 行)。在进入主循环之前，第 18 197 行的 *sys_getimage* 内核调用给进程管理器提供了一份映像表的副本 (严格来说，它并不是一个内核调用，它只是在 *include/minix/syslib.h* 中定义的十多个宏定义之一，用来为 *sys_getinfo* 内核调用提供一个易于使用的接口)。内核进程在用户空间是不为所知的，进程表的 PM (和 FS) 部分不需要为内核进程进行初始化。实际上，内核进程表项所需要的空间也没有预留，每个表项有一个负的进程编号 (进程表索引)，它们将被 18 202 行的测试所忽略。另外，也不必为内核进程来调用 *patch_mem_chunks*。

系统进程和用户进程需要被添加到进程表中，虽然它们的处理方法略有不同 (18 210 行到 18 219 行)。引导映像中的唯一一个用户进程是 *init*，因此需要测试 *INIT_PROC_NR* (18 210 行)，其他进程都是系统进程。系统进程比较特别：它们不能被换进换出；每个系统进程在内核的 *priv* 表中有一个专用的表项；它们具有特别的权限。对于每一个进程，必须为它们设置一个合适的信号处理默认操作 (对于系统进程和 *init* 进程，这个默认操作有所不同)。然后，使用 *get_mem_map* 函数，从内核中获取每个进程的内存映射。该函数最终会执行 *sys_getinfo* 内核调用。接下来再调用 *patch_mem_chunks* 函数来调整 *mem_chunks* 数组 (18 225 行到 18 230 行)。

最后，发送一条消息给文件系统，使得在进程表的 FS 部分中，为每个进程初始化一个表项 (18 233 行到 18 236 行)。该消息只包含了进程号和 PID，这对于初始化 FS 进程表项来说，已经足够了，因为系统引导映像中的所有进程都是属于超级用户的，因而可以被设置为相同的默认值。每条消息使用一个 *send* 操作来发出，不需要应答。在发出消息后，进程的名字被显示在控制台上 (18 237 行)：

```
Building process table: pm fs rs tty memory log driver init
```

在这个显示中，*driver* 代表的是默认的磁盘驱动程序，我们可以把多个磁盘驱动程序编译进引导映像中，然后在引导参数中使用一个 “*label =*” 语句，把其中的一个设置为默认的驱动程序。

PM 自己的进程表项是一个特例。当主循环结束后，PM 对它自己的表项进行了一些修改，然后向文件系统发送了一条最终的消息，在消息中使用 *NONE* 符号值来作为进程号。这条消息通过 *sendrec* 调用发出，然后进程管理器将阻塞自己，等待应答。当 PM 正在循环地执行初始化代码时，文件系统已经执行了一个 *receive* 循环 (24 189 行到 24 202 行)。在收到一条进程号为 *NONE* 的消息后，文件系统知道所有的系统进程都已经初始化好了，因此它就退出循环，并发出一个同步消息，唤醒 PM。

现在文件系统就可以继续往下，去进行它自己的初始化工作，而PM的初始化已经基本上完成了。在18 253行，*mem_init*被调用。该函数根据*mem_chunks*数组中搜集的信息，对空闲内存区链表和相关的变量进行初始化，当系统开始运行后，这些变量将被用于存储管理。通常意义上的存储管理起始于控制台上的一条打印消息，该消息列出了总的内存容量、MINIX 3已经使用的内存大小和空闲空间大小。

```
Physical memory: total 63996 KB, system 12834 KB, free 51162 KB.
```

下一个函数是*get_nice_value*(18 263行)，它的作用是去确定引导映像中的每个进程的“nice级别”。对于每个进程，在*image*表中有一个*queue*值，用于设置进程调度时的优先级队列。这些优先级从0到15，0表示最高优先级进程，如*CLOCK*；15表示最低优先级进程，如*IDLE*。不过，在UNIX系统中，传统意义上的“nice级别”是一个可正可负的值。因此对于用户进程，*get_nice_value*会把内核的优先级投影到一个以0为中心的刻度上。这项工作用到了*include/sys/resource.h*(未列出)中定义的一些常量，如*PRIO_MIN*和*PRIO_MAX*，它们的值分别是-20和+20。这些值被调整在*MIN_USER_Q*和*MAX_USER_Q*之间(在*kernel/proc.h*中定义)，这样，如果需要减少或增加调度队列的个数，那么*nice*命令仍然是有效的。位于用户进程树的根节点的*init*进程，它的*nice*值为0，位于第7个优先级调度队列中。这个值可以被它的子进程所继承。

*main.c*中的最后两个函数前面已经提到过了。*get_mem_chunks*(18 280行)只会被调用一次，它的输入是引导监控程序返回的内存使用信息，即一个ASCII字符串，包含了十六进制的(基址：长度)对，然后它会把这些信息转换为click单元个数，并把它们存放*mem_chunks*数组中。*patch_mem_chunks*(18 333行)负责继续构造空闲空间链表，它会被调用多次，一次是内核本身，一次是*init*，此外，在*pm_init*的主循环中，在初始化每一个系统进程时，也会去调用一次该函数。它负责纠正原始的引导监控程序信息。它的任务比较简单，因为它得到的数据已经是以click为单位。对于每一个进程，*pm_init*将收到该进程的代码段和数据段的基地址和大小。在分配内存时，对于每个进程，将某个空闲区域的最后一个元素的起始地址，加上代码段和数据段的长度之和。然后，再把该空闲区域的大小减去相同的数量，以表明分配给该进程的内存空间已被占用。

4.8.3 FORK, EXIT 和 WAIT 的实现

fork, *exit* 和 *wait* 系统调用是由*forkexit.c*文件中的*do_fork*, *do_pm_exit* 和 *do_waitpid* 函数来实现的。*do_fork*(18 430行)根据图4.36所示的步骤执行。请注意，对*procs_in_use*的第二次调用(18 445行)为超级用户保留了最后几个进程表项。在计算子进程需要多大的内存时，数据段和栈段之间的空隙也被包括在内，但不包括代码段。一种可能是因为父进程的代码段被共享；另一种可能是进程具有组合的I和D空间，但它的代码段长度为零。在计算完成之后，调用*alloc_mem*来申请内存。如果成功，子进程和父进程的基地址都将从click转换为绝对字节数，然后调用*sys_copy*发送一条消息给系统任务，由其完成复制工作。

现在需要在进程表中找到一个空项，早先在*procs_in_use*中的测试保证了此次寻找肯定能够成功。在找到一个空项后，首先把父进程的表项内容复制到这里，然后再修改其中的*mp_parent*, *mp_flags*, *mp_child_utime*, *mp_child_stime*, *mp_seg*, *mp_exitstatus* 和 *mp_sigstatus* 等字段。有些字段需要特殊处理，例如对于*mp_flags*字段，只有其中的某些位能够继承。*mp_seg*字段是一个数组，它的数组元素分别描述了代码段、数据段和栈段。如果进程具有独立的I和D空间，可以共享代码段，那么它的代码部分将指向父进程的代码段。

下一步是为子进程分配一个进程号，这是通过函数调用 `get_free_pid` 来实现的。这个函数并不像我们想象的那么简单，后面我们还会讨论这个问题。

`sys_fork` 和 `tell_fs` 分别用于通知内核和文件系统，一个新进程已经被创建，使它们能够去更新各自的进程表（所有以 `sys_` 开头的函数都是库函数，它们通过向内核中的系统进程发送消息来请求图 2.45 中的某个服务）。进程的创建和撤销都是从 PM 开始的，并在结束时传播到内核和文件系统。

对子进程的应答消息是在 `do_fork` 的结尾处发出的。而对父进程的应答（其中包含有子进程的 PID）则像通常的请求应答一样，是由 `main` 中的循环发出的。

PM 处理的下一个系统调用是 `exit`。函数 `do_pm_exit` (18 509 行) 负责接收这个调用，但它的大部分工作是通过调用 `pm_exit` 来完成的。之所以要如此划分，是因为 `pm_exit` 还被用来处理被信号终止的进程。这两项工作是相同的，但参数不同，所以这样的一种划分是比较方便的。

`pm_exit` 做的第一件事情是：如果进程有一个定时器在运行，那么就停止它。然后，子进程所使用的时间被添加到父进程的账号中。接下来，通知内核和文件系统该进程已经不再运行 (18 550 行和 18 551 行)。`sys_exit` 内核调用发送一条消息给系统任务，让它去清除该进程在内核进程表中所占用的表项，然后释放内存。接下来调用 `find_share` 函数，判断该进程的代码段是否被另一个进程共享，如果没有，就调用 `free_mem` 释放这个代码段。紧接着调用同一个函数去释放数据段和栈段。这样我们就连续 3 次调用了 `free_mem` 函数，另一种做法是只调用一次，然后判断能否释放所有的内存，但我们认为这样做是不值得的。接下来，如果父进程正在等待，就调用 `cleanup` 去释放进程表项；如果父进程未在等待，该进程就进入僵死状态，也就是说，把 `mp_flags` 字段中的 `ZOMBIE` 位设置为 1，然后向父进程发送一个 `SIGCHLD` 信号。

不管进程是被彻底消灭还是进入僵死状态，`pm_exit` 的最后一个动作是去遍历整个进程表，搜索刚刚被终止的进程的子进程 (18 582 行到 18 589 行)。如果能够找到，就把它们变成 `init` 的子进程。如果 `init` 正在等待并且一个子进程进入了挂起状态，那么就调用 `cleanup` 来处理这个子进程。这种情形如图 4.45(a) 所示。在图中我们看到，进程 12 要结束了，而它的父进程 7 正在等待它，因此 `cleanup` 将被调用以清除 12，这样 52 和 53 将成为 `init` 的子进程，如图 4.45(b) 所示。现在的情形是，已经结束的进程 53 成为某个正在执行 `wait` 操作的进程的子进程，所以它也将被清理。

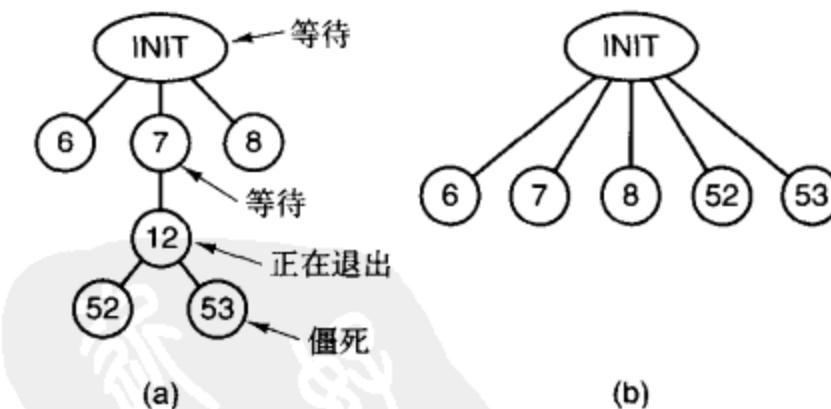


图 4.45 (a) 进程 12 即将结束时的情形；(b) 进程 12 结束以后的情形

当父进程执行 `wait` 或 `waitpid` 时，控制流就转移到 18 598 行的函数 `do_waitpid`。这两个调用的参数不太相同，所执行的操作也不同，但在 18 613 行到 18 615 行可以对内部变量进行设置，使得 `do_waitpid` 能够完成两个调用中的任何一个。从 18 623 行到 18 642 行的循环会扫描整个进程表，检查这个进程是否有子进程。如果有，再看看它们是否处于僵死状态。如果找到了一个僵死的子进程 (18 630 行)，`do_waitpid` 将把它清理掉并返回 `SUSPEND` 返回码。如果找到了一个被跟踪的子进程，`do_waitpid` 将修改正在构造的应答消息，标明该进程已经停止，然后返回。

如果执行 `wait` 的进程没有子进程，它将得到一个错误返回（18 653 行）；如果它有子进程，但其中没有一个处于僵死状态或被跟踪状态，那么系统将检查在 `do_waitpid` 被调用时是否设置了一个标志位，表明父进程不想等待。如果没有（通常的情形），18 648 行将设置一个位表明它正在等待，然后父进程将被挂起，直到某个子进程结束。

当一个进程已经结束并且它的父进程正在等待它时，不管这两个事件发生的次序如何，函数 `cleanup` 都将被调用（18 660 行），执行最后的欢送仪式。这时它要做的工作其实不多：把父进程从 `wait` 或 `waitpid` 中唤醒并告诉它被终止的子进程的 PID 以及退出状态和信号状态。此时文件系统已经释放了子进程的内存，内核已经停止了对它的调度，并释放了它在进程表中的表项。这样，这个子进程就永远地离去了。

4.8.4 EXEC 的实现

`exec` 的代码遵循着图 4.40 中的工作步骤，它被包含在 `exec.c` 文件的 `do_exec` 函数中（18 747 行）。PM 首先进行一些合法性检查，然后从用户空间取出即将执行的文件的名字（18 773 行到 18 776 行）。就像我们在图 4.38 中所看到的，实现 `exec` 的库函数在老的内核映像中构造了一个栈，这个栈被装入 PM 的内存空间中（18 782 行）。

接下来的几个步骤被放在一个循环中（18 789 行到 18 801 行），不过，对于通常的二进制可执行文件，这个循环只会执行一遍。我们先来看这种情形。在 18 791 行，发向文件系统的一个消息使得当前目录切换到用户的目录，这样，文件的路径名将被解释为相对于用户目录，而不是 PM 的当前工作目录。然后调用 `allowed`，判断此次运行是否合法，如果是，就打开该文件，返回相应的文件描述符；如果不是，就返回一个负数，然后调用 `do_exit` 退出。如果文件是存在的，而且是可执行的，PM 就调用 `read_header`，读取各个段的长度。对于一个通常的二进制文件，`read_header` 函数的返回码将使得 PM 在 18 800 行退出循环。

现在我们来看一下，如果可执行文件是一个脚本，那么应该如何处理。与大多数类 UNIX 操作系统一样，MINIX 3 也支持可执行脚本。`read_header` 测试文件的头两个字节，看它们是不是特别的符号序列 “#!”，如果是就返回一个特殊的代码，表示这是一个脚本文件。脚本的第一行一般指定了它的解释器，可能还指定了相应的标志位和选项。例如，某个脚本文件的第一行是

```
#! /bin/sh
```

这表明该脚本文件应该用 Bourne shell 来解释。再如

```
#! /usr/local/bin/perl -wT
```

这表明，该脚本文件应该用 Perl 来解释，并且设置了警告标志位，当出现问题时会显示提示信息。有了脚本文件以后，就使得 `exec` 的工作变得更加复杂。在运行一个脚本时，`do_exec` 需要装入内存的并不是脚本文件本身，而是相应的解释器的可执行代码。当发现一个脚本后，就在循环语句的末尾（18 801 行）调用 `patch_stack`。

`patch_stack` 所做的事情可以用一个例子来阐述。假设一个 Perl 脚本被调用，并带有一些命令行参数，例如

```
perl_prog.pl file1 file2
```

`patch_stack` 将创建一个栈来执行 perl 可执行代码，就好像用户输入的命令是

```
/usr/local/bin/perl -wT perl_prog.pl file1 file2
```

如果能够成功，该命令行的第一部分将被返回，也就是说，解释器的可执行文件的路径名将被返回。然后循环体将再次执行，这次将读入文件头，并获取该文件的各个段的大小。在脚本文件的第一行，不允许使用另一个脚本文件来作为它的解释器，这就是为什么要使用变量 *r* 的原因。它只能被增加一次，只有一次机会去调用 *patch_stack*。如果在第二轮循环中，代码表明碰上了一个脚本文件，那么在 18 800 行的测试语句就会跳出循环。脚本的代码用一个符号 *ESCRIP*T 来表示，该符号实际上是一个负数（在 18 741 行定义）。在这种情形下，18 803 行的测试将会导致 *do_exit* 返回一个错误码，表明该问题可能是由于一个无法执行的文件或命令行太长了。

为了完成 *exec* 操作，还有其他的一些工作要做。*find_share* 检查新的进程能否与某个正在运行的进程共享代码段（18 809 行）。*new_mem* 负责为新的映像分配内存空间并释放旧的内存。在 *exec-ed* 程序执行前，这两个内存映像和进程表都必须已经准备就绪。在 18 819 行到 18 821 行，可执行文件的索引节点、文件系统和修改时间都被保存在进程表中。然后像图 4.38(c) 那样，把栈设置好，并复制到新的内存映像。接下来，调用 *rw_seg*（18 834 行到 18 841 行），将代码段（如果不是共享代码段的话）和数据段从磁盘复制到内存映像。如果 *setuid* 或 *setgid* 位被置位，需要通知文件系统把有效的 ID 信息保存到进程表项的 FS 部分（18 845 行到 18 852 行）。在文件表的 PM 部分，一个指向新程序参数的指针被保存起来，这样 *ps* 命令就能够显示命令行。信号的位掩码被初始化，FS 收到通知关闭所有在 *exec* 后应该关闭的文件，命令的名字被保存起来，用于 *ps* 的显示或调试时的显示（18 856 行到 18 877 行）。一般来说，最后一个步骤是去通知内核，但如果启用了代码跟踪，那么就必须发出一个信号（18 878 行到 18 881 行）。

在描述 *do_exec* 的工作时，我们提到了 *exec.c* 中提供的一组支持函数。*read_header*（18 889 行）不仅能够读入文件头并返回段的大小，而且还能验证该文件是否是一个有效的 MINIX 3 可执行文件。18 944 行的常量 *A_I80386* 的值在编译时用一个 *#ifdef* … *#endif* 序列来确定。对于 Intel 平台上的 32 位 MINIX 3 可执行程序，必须把这个常量包含在它们的头文件中。如果 MINIX 3 将被编译并运行在 16 位模式下，那么该常量为 *A_I8086*。如果你想进一步了解相关的内容，可以去阅读 *include/a.out.h*，看看为其他 CPU 定义的值。

函数 *new_mem*（18 980 行）检查是否有足够的内存空间来存放新的内存映像。如果代码段是共享的，它就只为数据段和栈段查找一块足够大的空闲区域；否则，如果代码段不是共享的，那么就必须找到一块更大的空闲区域，能够把代码段、数据段和栈段全部装在里面。一种改进的思路就是查找两个独立的空闲区域，一个用于代码段，另一个用于数据段和栈段，原因是这两个区域不需要相邻。而在早期的 MINIX 版本中，三个段必须邻接在一起，所以只能存放在同一块空闲区中。如果内存空间是够用的，那么旧的内存将被释放，新的内存将被分配。如果没有足够的内存，*exec* 将失败。在新的内存分配后，*new_mem* 将更新内存映射（在 *mp_seg* 中），并通过 *sys_newmap* 内核调用来通知内核。

new_mem 剩下的工作是把 *bss* 段、空隙和栈段清零（*bss* 是数据段的一部分，存放了所有未初始化的全局变量）。这项工作是通过系统任务 *sys_memset*（19 064 行）来完成的。虽然许多编译器能自动生成将 *bss* 段清零的代码，但我们的这种做法更稳妥一点，即使编译器不能生成这种代码，系统也能正常工作。数据段和栈段之间的空隙也被清零，这样当数据段被 *brk* 系统调用扩充时，新获得的内存将全部包含零。这不仅是为了方便程序员，使他不必再将新变量初始化为零，在多用户操作系统上这还是一个安全措施，因为之前使用这块内存的进程可能在其中存放了一些重要数据，它不希望这些数据被其他进程看到。

下一个函数是 *patch_ptr*（19 074 行），它把图 4.38(b) 中的指针重定位为图 4.38(c) 的形式。这项工作很简单：检查栈，找到所有的指针，然后把基地址加到每个指针上。

接下来的两个函数是一起工作的，前面我们曾经谈到过它们的用途。当一个脚本文件被 `exec` 执行时，真正需要执行的其实是它的解释器的可执行代码。`insert_arg`(19 106行)负责把字符串插入栈中的PM副本，这项工作是在`patch_stack`(19 162行)的指导下完成的，它负责在脚本文件的第一行找到所有的字符串，然后调用`insert_arg`。当然，相关的指针也需要被校正。`insert_arg`的任务是很简单的，但也有很多的事情需要考虑，如果测试不够严密，可能就会出错。这里顺便再补充两句，在处理脚本文件时，必须仔细核查各种各样的问题。毕竟，脚本可能是由用户来编写的，而所有的计算机专业人员一致认为，用户通常就是问题的主要来源。当然，这仅仅是一个玩笑。实际上，对于一个脚本和一个编译好的可执行文件，它们之间的差别主要在于：我们可以充分信任编译器，它能检测出源代码中的许许多多的错误，而脚本文件却无法这样来验证。

图4.46是一个例子，有一个shell脚本`s.sh`，它对一个文件`f1`进行操作。在调用这个脚本时，命令行形如

```
s.sh f1
```

该脚本文件的第一行表明它的解释器是 Bourne shell：

```
#!/bin/sh
```

图4.46(a)是从调用者空间复制过来的栈，图4.46(b)显示了`patch_stack`和`insert_arg`如何对它进行转换。这些图都对应于图4.38(b)中的情形。

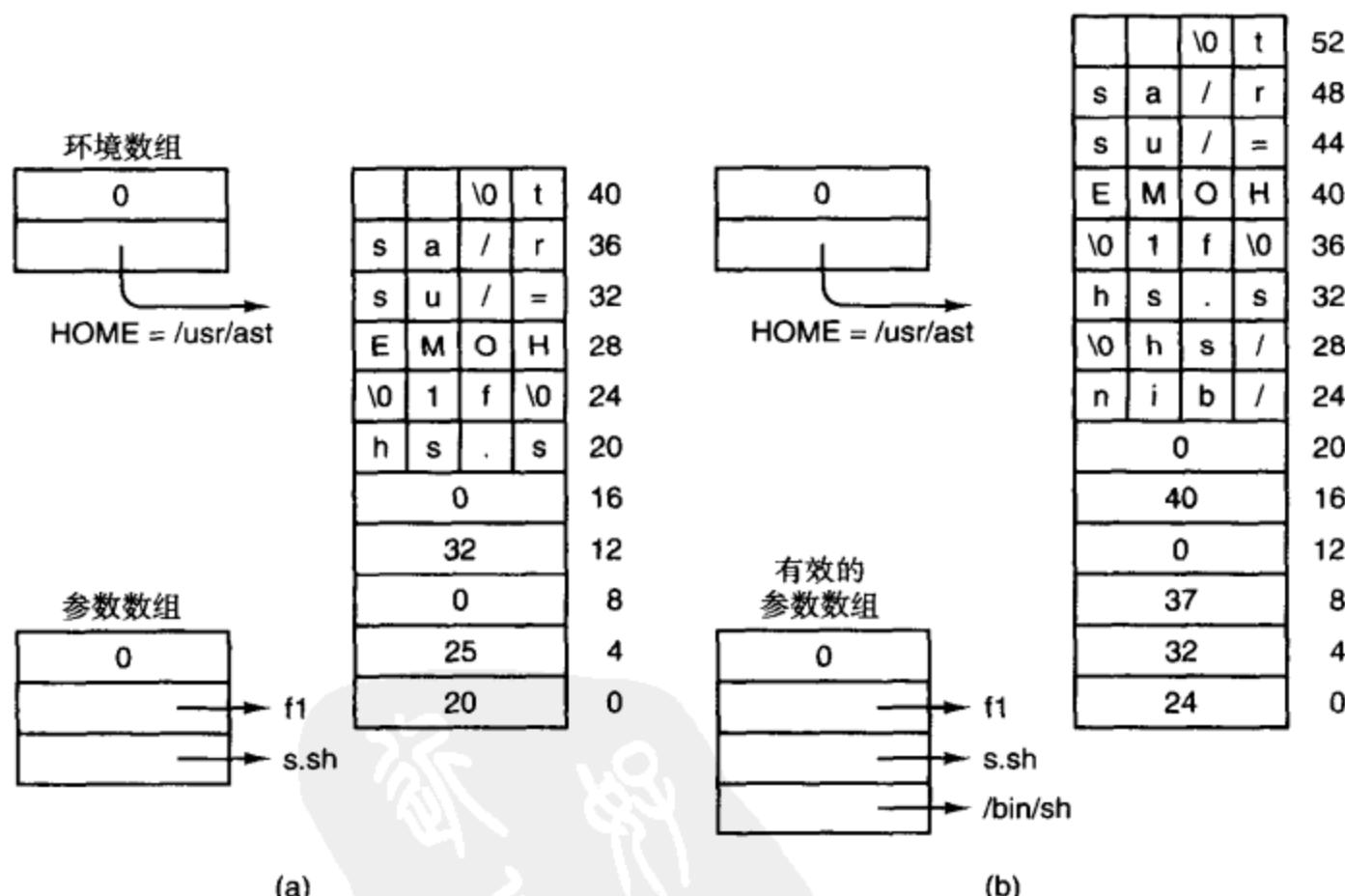


图4.46 (a) 传递给 `execve` 的数组以及当脚本被执行时所创建的栈；(b) 在经过 `patch_stack` 处理之后的数组和栈。脚本的名字被传递给解释该脚本的程序

`exec.c` 中定义的下一个函数是 `rw_seg`(19 208行)。在每一次 `exec` 中，它会被调用一次或两次。数据段肯定要装入，而代码段可能需要也可能不需要装入。在这里我们使用了一个技巧，可以让文件系统把整个段直接装入用户空间，而不需要一个数据块一个数据块地读入文件然后把它们复制到用户空间。实际上，这个调用被文件系统以一种略微特殊的方式解码，使它看起来就像是用户进程

自己去读取一个整段。只有文件系统的读函数的头几行知道这里面做了特殊的处理，这个策略使装入速度得到了明显的提高。

exec.c 中的最后一个函数是 *find_share* (19 256 行)，它把即将执行的文件的 i 节点、设备和修改时间与内存中的现有进程进行比较，看看能否与其中的某个进程共享代码段。这个搜索比较简单，只是对 *mproc* 中的相应字段进行比对。当然，这次搜索应该忽略刚刚被创建的这个进程本身。

4.8.5 BRK 的实现

正如我们刚刚看到的，MINIX 3 使用的内存模型是非常简单的：当一个进程被创建时，将获得一块用于存放数据和栈的连续内存区域。一旦内存分配完毕，进程既不会被移动，也不会被交换出内存；既不会增长，也不会缩小。唯一能够发生的变化是数据段可能会从下往上吃掉一些空隙，而栈段则可能会从上往下吃掉一些空隙。在这些情形下，位于 *break.c* 中的 *brk* 调用的实现是特别简单的。它首先检查新的大小是否可行，随后更新相应的表格以反映这些变化。

顶层的函数是 *do_brk* (19 328 行)，但大部分工作是由 *adjust* (19 361 行) 来完成的。后者负责检查数据段和栈段是否冲突。如果是，*brk* 调用就不能执行，但进程并不会立刻被杀死。在进行测试前，一个安全因子 *SAFETY_BYTES* 被添加到数据段的顶上，这样做的目的是：当我们不得不做出栈增长越界的结论时，在栈中仍会有足够的空间，使进程能够继续运行一小段时间。这样，它就能收回控制权（带着一个错误信息），打印出合适的错误信息，然后自然地结束。

注意，*SAFETY_BYTES* 和 *SAFETY_CLICKS* 是在函数的中间 (19 393 行) 用一个 `#define` 语句定义的。这种用法比较少见，在通常情形下，这种定义一般出现在文件的开头或单独的头文件中。相关的注释指出，程序员发现确定安全因子的大小是非常困难的，因此这种定义方式无非是为了引起注意，也许还是为了鼓励进一步的实验。

数据段的基地址是固定的，所以如果 *adjust* 需要调整数据段，只能去更新它的长度字段。栈从一个固定的终点向下增长，因此如果 *adjust* 注意到作为参数传递给它的栈指针已经超出了栈段（到了更低的地址），那么长度和起点都将被更新。

4.8.6 信号处理的实现

与信号处理有关的 POSIX 系统调用有 8 个，它们被列在图 4.47 中。这些系统调用和信号一起，都是在 *signal.c* 中处理的。

系统调用	用途
<i>sigaction</i>	修改对信号的响应方式
<i>sigprocmask</i>	改变阻塞信号集合
<i>kill</i>	向另一个进程发送信号
<i>alarm</i>	经过一定延迟后向自己发送ALRM信号
<i>pause</i>	挂起自己直到被信号唤醒
<i>sigsuspend</i>	改变阻塞信号集合，然后PAUSE
<i>sigpending</i>	检查未处理的（阻塞的）信号
<i>sigreturn</i>	在信号处理函数执行完后清理现场

图 4.47 与信号处理有关的系统调用

sigaction 调用支持 *sigaction* 和 *signal* 函数，这两个函数使进程能够改变自己对信号的响应方式。*sigaction* 是 POSIX 要求的，在多数情形下是首选的调用。而 *signal* 库函数是标准 C 要求的，如

果一个程序要移植到非 POSIX 系统上，那么就要使用这个函数。*do_sigaction* 的代码（19 544 行）首先检查信号编号的合法性，并验证此次调用不是去试图改变对 *sigkill* 信号的响应方式（19 550 行到 19 551 行）（忽略、俘获或阻塞 *sigkill* 是不允许的，因为它是用户得以控制自己的进程以及系统管理员得以控制他的用户的最后手段）。在调用 *sigaction* 时会带有两个指针 *sig_osa* 和 *sig_nsa*，分别指向两个 *sigaction* 结构体变量。前者存放的是本次调用执行前有效的旧的信号属性；而后者存放的是新的属性信息。

第一步是调用系统任务把当前的属性复制到 *sig_osa* 所指的结构中。具体做法是把 *NULL* 指针放在 *sig_nsa* 中，然后去调用 *sigaction*。在这种情形下 *do_sigaction* 会立刻返回（19 560 行）。如果 *sig_nsa* 不是 *NULL*，那么定义了新的信号动作的结构将被复制到 PM 的空间中。

19 567 行到 19 585 行的代码根据新动作的类型，即忽略信号、使用默认处理函数或捕获信号，来修改相应的位图，即 *mpCatch*, *mpIgnore* 或 *mpSigpending*。如果需要捕获信号，或者使用了特殊代码 *SIG_JGN* 或 *SIG_DFL*（如果你了解 POSIX 标准中的信号处理部分，就会明白它们的含义），那么可以使用 *sigaction* 结构中的 *saHandler* 字段来传递一个函数指针给即将执行的函数。此外还有一个特殊的 MINIX 3 专用的代码 *SIGMESS*，下面我们会介绍。

库函数 *sigaddset* 和 *sigdelset* 被用来修改信号位图，尽管这种修改操作实质上是一些简单的位运算，完全可以用一些宏定义来实现。事实上，这两个函数是 POSIX 标准的要求，目的是使调用它们的程序更容易移植。例如，在一个新系统中，即使信号的个数超出了一个整数的可用位数，这两个函数也能够处理。此外，通过使用这些函数，使 MINIX 3 自己也能更方便地移植到不同的体系结构中。

我们刚才谈到了一种特殊的情形，在 19 576 行检测的 *SIGMESS* 码只对特权（系统）进程有效。这些进程通常处于阻塞状态，正在等待请求消息。因此，通常的接收信号的做法（即 PM 要求内核把一个信号帧放在接收进程的栈中）将被延迟，直到一条消息把接收进程唤醒。*SIGMESS* 码告诉 PM 去发送一条通知消息，该消息比正常的消息具有更高的优先级。在一条通知消息中，包含了未处理的信号集合来作为它的参数，这样就可以在一条消息中处理多个信号。

最后，进程表中 PM 部分的其他与信号有关的字段会被填入。每个信号都有一个位图 *saMask*，它定义了当这个信号的处理函数正在执行时哪些信号将被阻塞。每个信号也都有一个指针 *saHandler*，它可以存放信号处理函数的地址，也可以存放一些特殊的值，来表明信号将被忽略、信号将使用默认处理方式或信号将被用来生成一条消息。在处理函数结束时调用 *sigreturn* 的库函数的地址保存在 *mpSigreturn* 中，这个地址是 PM 收到的消息中的一个字段。

POSIX 允许进程去定义自己的信号处理方式。即使当前正在处理一个信号，正在信号处理函数中执行，也可以去提出这个请求。这样，当下一个信号到来时，就可以按照新的响应方式去处理。接下来的一组系统调用与信号操作有关。*sigpending* 由 *doSigpending*（19 597 行）处理，它返回位图 *mpSigpending*，使进程能知道它是否有未处理的信号；*sigprocmask* 由 *doSigprocmask* 处理，它返回当前被阻塞的信号集合，也可以用来改变集合中某个信号的状态，或者是修改整个集合。当一个信号被解除阻塞时，它最好去检查是否有信号需要处理，而这项工作是通过 19 635 行和 19 641 行的 *checkPending* 来完成的。*doSigsuspend*（19 657 行）执行 *sigsuspend* 系统调用，该调用将挂起一个进程，直到它收到一个信号为止。与这里讨论过的其他函数一样，该函数也要操作位图。此外，它还要设置 *mpFlags* 中的 *sigsuspended* 位，从而阻止被挂起的进程运行。同样，这里也是调用 *checkPending* 的一个很好的时机。最后，*doSigreturn* 处理 *sigreturn*，它用来从一个用户自定义的处理函数中返回。它的主要工作是恢复在刚刚进入处理函数时的信号上下文，然后在 19 682 行再次调用 *checkPending*。

当一个用户进程，如 *kill* 命令，启动了 *kill* 系统调用时，PM 的 *do_kill* 函数（19 689 行）就开始执行。在调用 *kill* 时，可能需要向一组进程发送信号，而 *do_kill* 的工作只是调用 *check_sig*，去检查整个进程表，看哪些进程将收到信号。

有些信号是源自内核的，如 *sigint*。当 PM 收到一条来自内核的有关未处理信号的消息时，*ksig_pending*（19 699 行）就会被激活。在系统中可能会有多个进程，它们都有未决的信号需要处理，因此 19 714 行到 19 722 行的循环语句将不断地向系统任务查询未决信号，并把它传给 *handle_sig*，然后告诉系统任务该信号处理完成。就这样不断循环，直到所有的未决信号都已处理完毕。在消息中含有一个位图，使得内核可以在一条消息中产生多个信号。接下来的函数 *handle_sig*，一位一位地去处理位图（19 750 行到 19 763 行）。有些内核信号需要特别注意：在有些情形下进程 ID 可以被修改，使得信号可以被发送给一组进程（19 753 行到 19 757 行）。除了这个例外，集合中的每个位都将导致 *check_sig* 的一次调用，这与 *do_kill* 中的情形相同。

警报和定时器

alarm 系统调用由 *do_alarm*（19 769 行）处理，它又调用了另一个函数 *set_alarm*。之所以要引入一个单独的函数，是因为 *set_alarm* 还有另外的用途。当一个进程要退出时，如果还有一个定时器处于开启状态，就可以使用这个函数来关闭它。具体做法就是在调用 *set_alarm* 时，把警报时间设置为 0。*set_alarm* 在工作时需要用到进程管理器所维护的定时器。它首先检查一下，看发出请求的这个进程是否设置了一个定时器，如果是的话，再看一下该定时器是否过期，这样，在函数返回的时候，就可以返回本次警报还剩余的时间（以秒为单位）。如果根本就没有设置定时器，那么就返回 0。代码中的注释讨论了在处理长时间的警报时可能会碰上的一些问题。在 19 918 行，有一段比较丑陋的代码，把函数调用的参数（以秒为单位的时间）乘以一个常量 *HZ*，即每秒钟的时钟节拍数，从而把时间的单位从秒变成了节拍。为了使最后的结果具有正确的 *clock_t* 数据类型，需要用到三个强制类型转换。接下来在下一行，又进行了反向计算，把时钟节拍数从 *clock_t* 转换为 *unsigned long*。然后把这个结果与原始的警报时间参数（也需要转换为 *unsigned long*）进行比较，如果不相等，这说明请求的警报时间太长了，使得内部计算的数值超出了数据类型的表达范围，因此就使用一个特殊的值来代替，表示“永不”的意思。最后，调用 *pm_set_timer* 或 *pm_cancel_timer*，从进程管理器的定时器队列中增加或删除一个定时器。前者的主要参数是 *cause_sigalarm*，即定时器过期时将会执行的看门狗函数。

与内核空间中的定时器的交互被隐藏在一系列 *pm_XXX_timer* 函数中。一般来说，每一次的警报请求最终会变为请求在内核空间设置一个定时器，唯一的例外就是多个请求所要求的过期时间是完全相同的，这就会导致时间上的冲突。进程可以在它们的警报过期之前取消警报甚至结束运行。另外，只有当进程管理器的定时器队列中的首结点发生变化时，才需要使用一个内核调用，去请求在内核空间设置一个定时器。

当内核空间的定时器队列中的一个定时器过期时，系统任务就会发送一个通知消息给 PM，在 PM 的主循环中，检测出的类型为 *SYN_ALARM*。这时，它就会去调用 *pm_expire_timers*，并最终导致另一个函数 *cause_sigalarm* 的执行。

如前所述，*cause_sigalarm*（19 935 行）是一个看门狗函数，它的工作包括：获得信号接收进程的进程号，检查一些标志位，重置 *ALARM_ON* 标志位，调用 *check_sig* 去发送 *SIGALRM* 信号。

SIGALRM 信号的默认动作是杀死进程。如果要捕获 *SIGALRM*，必须用 *sigaction* 来安装一个处理函数。图 4.48 显示了在用户自定义处理函数的情形下，*SIGALRM* 信号的完整事件序列。这里有三条消息序列。首先，在消息(1)中，用户通过向 PM 发送消息来执行一个 *alarm* 调用。此时，进程

管理器在它维护的定时器队列中，为用户进程设置一个定时器，并使用消息(2)作为应答。然后在接下来的一小段时间内，什么也不会发生。过了一会儿，当本次请求的定时器到达PM的定时器队列的开头时，位于它前面的那个定时器已经过期，或者已经被取消，这样，PM就发送消息(3)给系统任务，让它为自己设置一个新的内核空间的定时器，并且收到相应的应答消息(4)。然后在接下来的一小段时间内，一切又归于平静。过了一会儿，当这个定时器到达内核空间的定时器队列的开头时，时钟中断处理程序会发现它已经到期了。因此就发送一个 *HARD_INT* 消息(5)给时钟任务，使之开始运行并更新它的定时器。然后定时器看门狗函数 *cause_alarm* 将会发起消息(6)，向PM发送一个通知。PM就开始更新它自己的定时器队列，然后通过查询进程表中的PM部分，发现目标进程已经为 *SIGALRM* 安装了一个处理函数，于是就发送消息(7)给系统任务，让它去做一些必要的栈操作，以便于向这个用户进程发送信号。随后就收到了系统任务给出的应答消息(8)。接下来，用户进程将被调度，而且将执行信号处理函数，在执行完之后，会向进程管理器发出 *sigreturn* 调用(9)。然后进程管理器发送消息(10)给系统任务，让它完成相应的清理工作。随后收到应答消息(11)。在这个图中还有另外一组消息没有画出来，就是在消息(3)之前，PM会向系统任务发出消息，去查询 *uptime*。

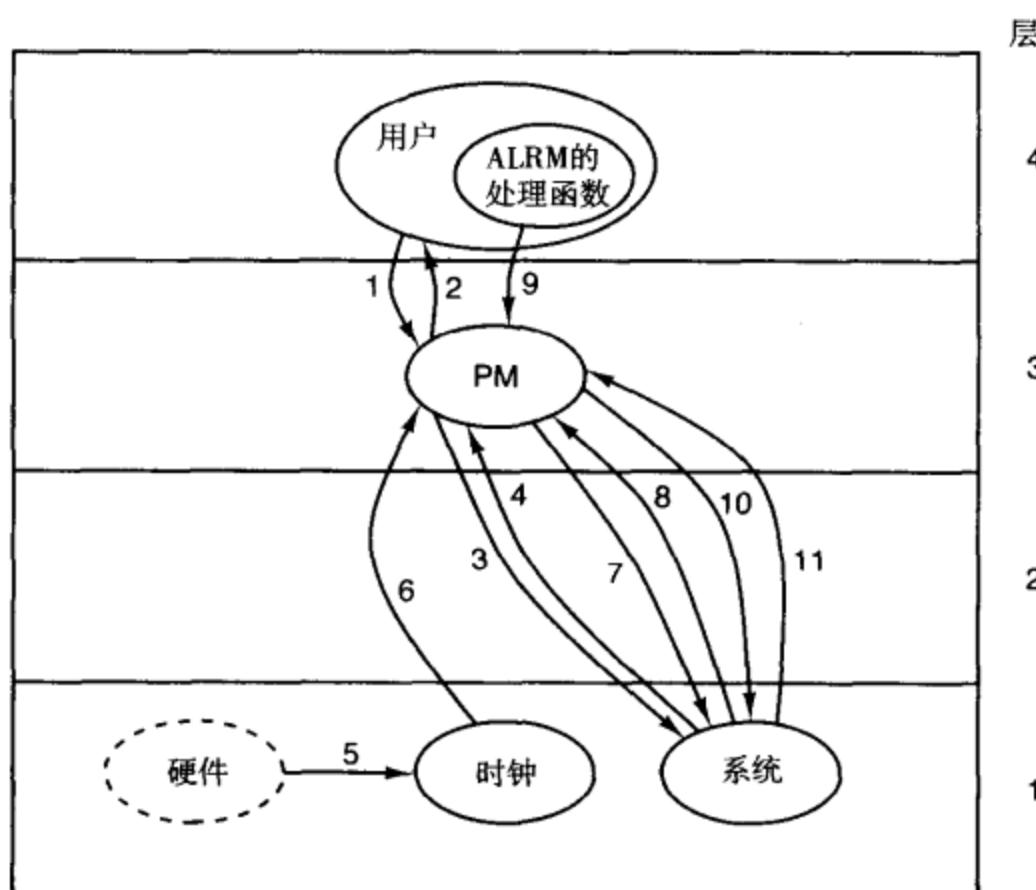


图 4.48 与一次警报相关的信息。其中最重要的是：(1) 用户执行alarm；(3)PM 要求系统任务设置定时器；(6)时钟告诉PM已过期；(7)PM 请求向用户发送信号；(9)信号处理函数结束并调用 *sigreturn*。详见文中描述

下一个函数 *do_pause* 负责 *pause* 系统调用 (19 853 行)。它其实与警报和定时器关系不大，虽然在程序中，可以使用它来暂停执行，等待一个警报（或其他信号）的到来。该函数所要做的事情，就是设置一个位，然后返回 *SUSPEND* 码。这样，PM 的主循环就不会去应答，从而使调用者被阻塞起来。对于内核而言，它甚至不需要别人的通知，因为它已经知道这个调用者被阻塞。

信号的支持函数

前面我们曾经提到过 *signal.c* 中的几个支持函数，现在我们将详细地讨论它们。到目前为止最重要的是 *sig_proc* (19 864 行)，它的功能是发送一个信号。首先会做一些检查，如果试图向结束

进程或僵死进程发送信号，这是不允许的，会导致严重的问题，使系统崩溃（19 889行到19 893行）。对于一个正在被跟踪的进程，当它收到一个信号后将会被停止（19 894行到19 899行）。如果信号将被忽略，那么 *sig_proc* 的工作在 19 902 行结束。对于一些信号来说，这是默认的动作，例如那些 POSIX 要求但 MINIX 3 不支持的信号。如果信号将被阻塞，唯一需要做的就是在那个进程的 *mp_sigpending* 位图中设置一个位。19 910 行的测试是比较重要的，用于区分已经允许捕获信号的进程和还没有允许捕获信号的进程。到此为止，除了一种例外以外（某些信号被转换为消息，发送给系统服务），所有其他的特殊情况都已经消除，不能捕获信号的进程将结束。

下面我们先来看一下能被捕获的信号的处理过程（19 911 行到 19 950 行）。首先构造一条消息发往内核，它的部分内容直接复制于进程表中的 PM 部分。如果接收信号的进程先前被 *sigsuspend* 挂起，那么在挂起时被保存的信号掩码将被包含在消息中；否则，当前的信号掩码将被包含进来（19 914 行）。消息中包含的其他内容是目标进程的地址空间中的几个地址：信号处理函数的地址、在处理函数结束时将调用的 *sigreturn* 库函数的地址，以及当前的栈指针。

接下来，在进程的栈中分配空间。图 4.49 显示了放在栈上的结构。*sigcontext* 部分被放在栈上保存，以便于后面的恢复，因为进程表中对应的结构在准备执行信号处理函数时将会被修改。*sigframe* 部分提供了信号处理函数的返回地址，以及 *sigreturn* 在完成进程状态恢复时所需的数据。返回的地址和帧指针实际上并未被 MINIX 3 的任何部分使用，它们之所以在那里，是为了当有人用调试器来跟踪信号处理函数时去愚弄调试器。

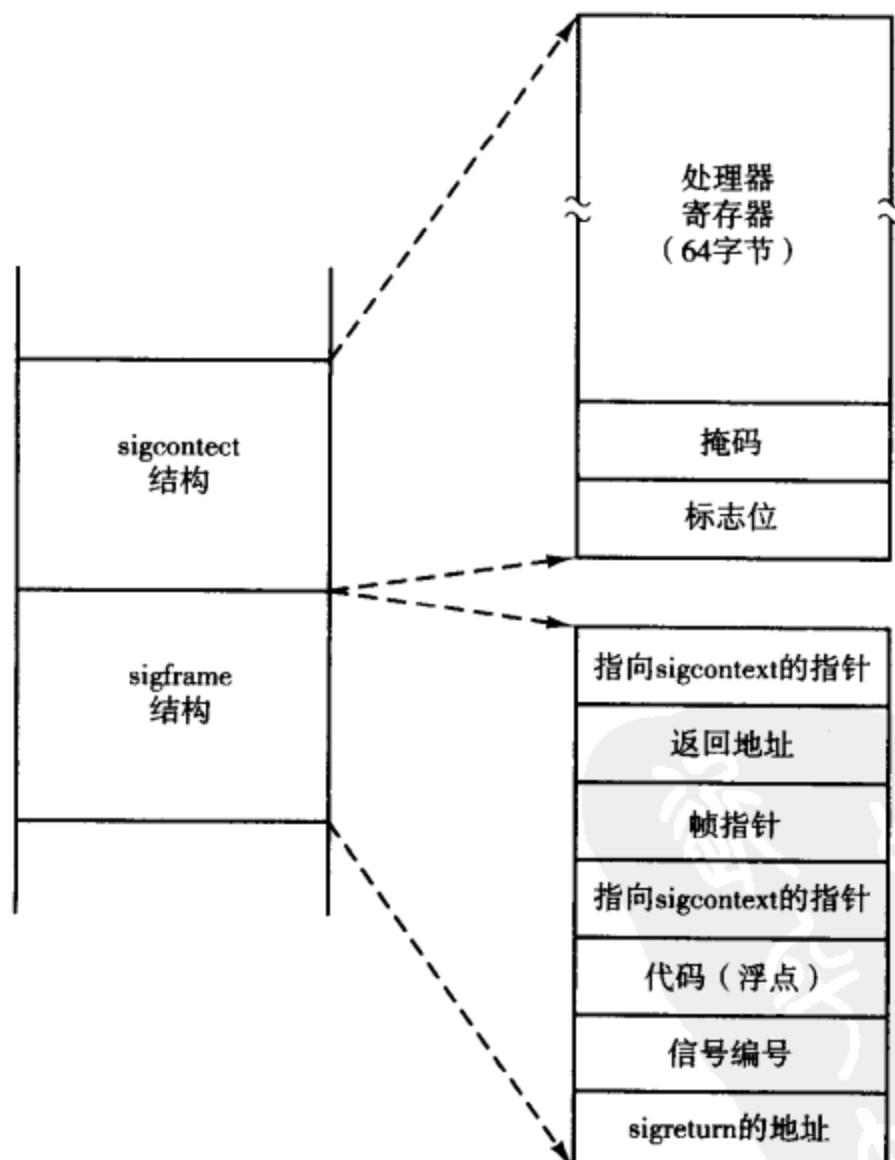


图 4.49 为信号处理函数做准备而推入栈的 *sigcontext* 和 *sigframe* 结构。处理器寄存器是上下文切换时使用的栈帧的一个副本

将被放到目标进程的栈上的结构是相当大的。19 923行和19 924行的代码将为它保留空间，随后调用 *adjust* 去检查进程的栈上是否有足够的空间。如果没有，程序将使用 *goto* 语句（19 926行和19 927行）跳转到标号为 *determinate* 的地方，然后杀死进程。

对 *adjust* 的调用有一个潜在的问题。在讨论 *brk* 的实现时，我们曾经提到，如果栈段与数据段的空隙长度小于 *SAFETY_BYTES*，那么 *adjust* 将返回错误。为错误提供一个裕量是因为合法性检查只能由软件间歇地进行。在当前的例子中这个错误裕量可能有点太大了，因为处理信号所需的栈空间是确切知道的，而且其余的空间只有信号处理函数才需要用到，而这个函数一般是一个比较简单的函数，用不了太多的栈空间。这样一来，有些进程可能会因为对 *adjust* 的调用失败而毫无必要地终止。当然，这比让进程在某些时候神秘地失败要强，但最好是对这些测试进行更加精细的调节，也许在将来某个时候我们会这样做。

如果栈上有足够的空间，另外两个标志位将被检验。*SA_NODEFER* 标志位指出在处理信号时是否阻塞后来的相同类型的信号，*SA_RESETHAND* 标志位指出在接到这个信号时信号处理函数是否被复位（这忠实地模拟了老式的 *signal* 调用，尽管这个“特性”经常被认为是老式调用的一个缺陷，支持老式的特性需要同时支持它们的缺陷）。随后用内核调用 *sys_sigsend*（19 940行）将 *sigframe* 放在栈上，以此来通知内核。最后，将未处理信号的指示位清除，然后调用 *unpause* 以结束进程可能会在上面挂起的任何系统调用。接下来，当接收信号的进程被调度时，信号处理函数将被执行。如果由于某种原因，使得上述的测试失败，那么 PM 就会崩溃（19 949行）。

上面曾经谈到过的例外（即信号被转换为消息以用于系统服务）会在 19 951 行被测试，并使用 *sys_kill* 内核调用来执行。这将使系统任务发送一个通知消息给接收信号的进程。如前所述，与其他的通知不同，来自于系统任务的通知承载了关于其起源和时间戳的基本信息，并传送了一个信号位图，这样，对于接收信号的系统进程来说，它可以了解所有的未处理信号。如果 *sys_kill* 失败，那么 PM 将崩溃；如果它成功，就返回 *sig_proc*（19 954行）。如果 19 951 行的测试失败，执行流就跳转到标号为 *determinate* 的地方。

下面我们看一看标号为 *determinate*（19 957行）的结束代码，这个标号和 *goto* 语句配对，是处理 *adjust* 调用失败的最简单方法。这里被处理的信号由于某种原因不能或不应该被捕获。如果该信号将被忽略，那么就返回 *sig_proc*。否则，该进程必须终止。唯一的疑问是内核映像转储是否需要。最后，通过调用 *pm_exit*（19 967行）使进程结束。

check_sig（19 973行）的功能是检查一个信号是否能够被发送。调用

```
kill(0, sig);
```

将使指定的信号被发给调用者同组的所有进程（即从同一个终端启动的所有进程），源自内核的信号和 *reboot* 系统调用也可能会影响到多个进程。由于这个原因，*check_sig* 在 19 996 行到 20 026 行的循环中，扫描整个进程表以找出所有应该接收信号的进程。该循环包含了大量的测试，只有当所有的测试都通过时，信号才会被发送出去，发送操作是通过在 20 023 行调用 *sig_proc* 来完成的。

check_pending（20 036行）也是一个重要的函数，在我们前面讨论过的代码中曾经被多次调用。对于 *do_sigmask*, *do_sigreturn* 或 *do_sigsuspend* 所引用的进程，它循环地检查 *mp_sigpending* 位图中的每一位，看看是否有一个被阻塞的信号已经不再阻塞了。如果有，它就调用 *sig_proc* 来发送它所找到的第一个不再阻塞的未处理信号。由于所有的信号处理函数最终都会引发 *do_sigreturn* 的执行，因此这段代码足以保证能够把所有被挂起的信号都发送出去。

函数 *unpause*（20 065行）所涉及的信号，通常发往一些被挂起的进程，而且是挂起在 *pause*, *wait*, *read*, *write* 或 *sigsuspend* 调用上。在检查 *pause*, *wait* 和 *sigsuspend* 时，可以去查询进程表

的PM部分，如果都没有发现，就必须要求文件系统用它自己的*do_unpause*函数去检查一下，看有没有在**read**或**write**上挂起的情形。在每一种情形下，动作都是相同的：向等待的调用发送一个错误应答，将对应于进程等待原因的标志位复位，使进程能够恢复执行并处理信号。

这个文件中的最后一个函数是*dump_core*（20 093行），它负责把内核映像写到磁盘上。在内核映像的头部，包括了各种各样的信息，如进程各个段的长度、各个进程的状态信息（复制自内核的进程表）以及每个段的内存映像。调试器能解释这些信息以帮助程序员确定在进程运行过程中发生了什么错误。

写文件的代码是很直观的，但前面章节中提到过的潜在的问题又出现了，而且更难处理。为了保证记录到内核映像上的栈段是最新的，在20 120行调用了*adjust*，这个调用可能会因为安全裕量的原因而失败。但*dump_core*不会去检查这个调用是否成功，所以无论如何内核映像都将被写入，只不过在文件中有关栈的信息可能会不正确。

定时器的支持函数

在MINIX 3中，用户进程不能直接与内核或系统任务打交道，它们的警报请求必须由进程管理器来处理。关于时钟任务的警报调度细节则被隐藏在这个接口的后面。只有系统任务才能够在内核中设置一个警报定时器，相关的支持函数在文件*timers.c*中（20 200行）。

进程管理器维护了一个警报请求链表，并要求系统任务在某个警报过期时通知它。当收到来自于内核的一个警报后，进程管理器就把它转发给相应的进程。

定时器的支持函数主要有3个，*pm_set_timer*设置一个定时器并把它添加到PM的定时器链表中，*pm_expire_timer*负责检查过期的定时器，*pm_cancel_timer*负责从PM的链表中删除一个定时器。这3个函数都用到了*include/timers.h*中声明的各种定时器库函数。*pm_set_timer*函数调用了*tmrs_settimer*，*pm_expire_timer*调用了*tmrs_exptimers*，*pm_cancel_timer*调用了*tmrs_clrtimers*。这些函数都能对定时器链表进行遍历，或者插入或删除链表结点。只有当一个结点被插入到队列的开头，或是从队列开头删除时，PM才需要与系统任务打交道，以调整内核空间的定时器队列。在这种情形下，*pm_XXX_timer*之类的函数将会使用*sys_setalarm*内核调用去请求内核级别的帮助。

4.8.7 其他系统调用的实现

进程管理器处理三个与时间有关的系统调用，即**time**，**stime**和**times**，这些调用是在*time.c*中定义的，如图4.50所示。

调用	功能
time	获得当前的实时时间和运行时间（以秒为单位）
stime	设置实时时钟
times	获得进程的统计时间

图4.50 与时间有关的三个系统调用

实时时间是由内核中的时钟任务来维护的，但时钟任务本身并不会和系统任务之外的其他进程交换信息。因此，对于用户进程来说，如果要获得或设置当前的实时时间，唯一的途径就是向系统任务发送一条消息。这实际上就是*do_time*（20 320行）和*do_stime*（20 341行）所做的事情。实时时间的单位是秒，从1970年1月1日开始计时。

统计时间也是由内核来维护的。每隔一个时钟节拍，它就把进程的计费时间加1。由于内核并不知道进程之间的父子关系，因此由进程管理器来统计一个进程的各个子进程所花费的总时间。当

一个子进程退出时，它的时间就被累加到父进程所在的进程表项中。*do_times*（20 366 行）通过 *sys_times* 内核调用从系统任务那里获得父进程的时间使用情况，然后在应答消息中填入各个子进程名下的用户时间和系统时间。

文件 *getset.c* 包含了一个函数 *do_getset*（20 415 行），它负责执行 POSIX 要求的 7 个 PM 系统调用，如图 4.51 所示。这些调用都非常简单，甚至都不值得为每个调用编写一个单独的函数。*getuid* 和 *getgid* 调用返回真实和有效的用户号或组号。

系统调用	功能描述
<i>getuid</i>	返回真实的和有效的用户号
<i>getgid</i>	返回真实的和有效的组号
<i>getpid</i>	返回进程和其父进程的进程号
<i>setuid</i>	设置调用者的真实和有效用户号
<i>setgid</i>	设置调用者的真实和有效组号
<i>setsid</i>	创建新的会话，返回进程号
<i>getpgrp</i>	返回进程组标识

图 4.51 *servers/pm/getset.c* 中定义的系统调用

用户号和组号的设置要比读取稍微难一点，先要检查调用者是否有权限去设置组号或用户号。如果通过了检查，还必须把新的用户号或组号通知文件系统，因为文件的保护还要依赖于它。*setsid* 调用创建一个新的会话，但已经是进程组组长的进程不能执行这个操作，这个检查是由 20 463 行的测试来完成的。文件系统完成把一个进程变成没有控制终端的进程组组长的工作。

与本章的其他系统调用不同，*misc.c* 中的调用不是 POSIX 所要求的。我们之所以引入这些调用，是因为用户空间的设备驱动程序和 MINIX 3 的服务器需要与内核进行通信，而在整体式内核的操作系统中是不需要这一点的。图 4.52 列出了这些系统调用及其相应的功能。

系统调用	功能描述
<i>do_allocmem</i>	分配一块内存
<i>do_freetmem</i>	释放一块内存
<i>do_getsysinfo</i>	从内核获得PM的信息
<i>do_getprocnr</i>	通过进程号或进程的名字获得进程表的索引
<i>do_reboot</i>	撤销所有的进程，并通知文件系统和内核
<i>do_getsetpriority</i>	获得或设置系统优先级
<i>do_svrcrtl</i>	把一个进程加入服务器

图 4.52 *servers/pm/misc.c* 中定义的一些特殊用途的系统调用

前两个调用完全由 PM 来处理。*do_allocmem* 从一个收到的消息中解析出请求，把它转换为以 click 为单元，然后调用 *alloc_mem*。例如，内存驱动程序使用这个调用为 RAM 盘分配内存。*do_freetmem* 的处理过程是类似的，但它调用的是 *free_mem*。

接下来的几个调用通常需要来自于系统其他部分的帮助，可以把它们视为系统任务的接口。*do_getsysinfo*（20 554 行）可以做几件事情，这取决于消息中的请求类型。它可以调用系统任务，获得存放在 *kinfo* 结构（在 *include/minix/type.h* 中定义）中的有关内核的信息，它也可以用来提供进程表的 PM 部分的地址，或者是将整个进程表的副本提供给另一个进程。最后的这个操作是通过调

用 `sys_datacopy` (20 582 行) 来实现的。在给定 PID 的情形下, `do_getprocnr` 能够找到相应的进程表中的索引。如果它只知道目标进程的名字, 那么就必须调用系统任务获得相应的帮助。

接下来的两个调用, 虽然不是 POSIX 所要求的, 但在许多类 UNIX 的系统中, 都有不同形式的版本。`do_reboot` 向所有的进程发送一个 *KILL* 信号, 并告诉文件系统做好重启的准备。只有当文件系统做好了相应的准备后, 它才会调用 `sys_abort` 去通知内核 (20 667)。重启可能是系统崩溃的结果, 也可能是来自于超级用户的请求, 要求去关机或重启。`do_getsetpriority` 支持著名的 UNIX *nice* 工具, 允许用户去降低某个进程的优先级, 以向其他的进程示好 (当然, 也可能是用户自己的进程)。更重要的是, MINIX 3 系统使用该调用实现了一种精细的优先级控制, 来设定各个系统组件的相对优先级。如果一个网络或磁盘设备必须处理一个快速的数据流, 那么可以给它设置比较高的优先级; 如果一个设备接收数据的速度比较慢, 如键盘, 那么可以给它设置比较低的优先级。如果一个高优先级的进程陷在一个循环之中, 妨碍了其他进程的运行, 那么可以暂时调低它的优先级。正如第 2 章中所讨论的, 优先级的修改是通过去调度更低 (或更高) 优先级队列中的进程来实现的。当内核中的调度器发起这个操作时, 不需要涉及到 PM。但如果是一个普通的进程想这么做, 则必须使用一个系统调用。在 PM 的层面上, 它所要做的事情, 无非是从一条消息中读出一个当前值, 或者是用一个新的值去生成一条消息。`sys_nice` 内核调用负责将一个新值发送给系统任务。

misc.c 中的最后一个函数是 `do_svrcctl`, 当前它主要用来启动和关闭交换模块。其他一些曾经使用过这个调用的函数, 现在大都在服务器中实现。

本章我们将考虑的最后一个系统调用是 `ptrace`, 它位于 *trace.c* 中。这个文件未列在附录 B 中, 但可以在 CD-ROM 和 MINIX 3 网站上找到。Ptrace 用来调试程序, 它的参数可能是图 4.53 所示的 11 种命令之一。在 PM 中, `do_trace` 处理其中的 4 个: `T_OK`, `T_RESUME`, `T_EXIT` 和 `T_STEP`。跟踪的启用和退出请求在这里完成, 其他的命令都被传递给系统任务, 因为它能够访问进程表的内核部分。这是通过调用库函数 `sys_trace` 来完成的。在 *trace.c* 的尾部定义了两个支持函数: `find_proc` 搜索进程表, 寻找需要跟踪的进程; 而 `stop_proc` 的作用是当一个正在被跟踪的进程收到一个信号时, 停止该进程。

命令	功能描述
<code>T_STOP</code>	停止进程
<code>T_OK</code>	允许这个进程被父进程跟踪
<code>T_GETINS</code>	从代码 (指令) 空间返回值
<code>T_GETDATA</code>	从数据空间返回值
<code>T_GETUSER</code>	从用户进程表返回值
<code>T_SETINS</code>	在代码 (指令) 空间设置值
<code>T_SETDATA</code>	在数据空间设置值
<code>T_SETUSER</code>	在用户进程表中设置值
<code>T_RESUME</code>	恢复执行
<code>T_EXIT</code>	退出
<code>T_STEP</code>	设置跟踪位

图 4.53 *servers/pm/trace.c* 支持的调试命令

4.8.8 内存管理工具

在本章的最后，我们再来简单地介绍一下两个文件，*alloc.c* 和 *utility.c*，它们为进程管理器提供了一些支持函数。关于这两个文件的内部细节这里就不详细说了，它们也未包含在本书的附录B中（为了避免使这本已经很厚的书变得更厚）。当然，读者可以在本书的CD-ROM和网站上找到它们。

文件 *alloc.c* 用来记录内存的使用状况，哪些地方已被占用，哪些地方还处于空闲。它有3个入口：

1. *alloc_mem*: 请求一块指定大小的内存。
2. *free_mem*: 归还不再需要的内存。
3. *mem_init*: 在 PM 开始运行时初始化空闲链表。

如前所述，*alloc_mem* 在一条按内存地址排序的空闲链表上使用最先匹配算法，查找一块空闲区域。如果找到的块太大，就把它一分为二，一块用于此次分配，另一块是剩余的空闲空间，仍然留在空闲链表上，但它的长度要发生变化。如果块的大小正好合适（一般不太可能），那么就调用 *del_slot*，把这个结点从空闲链表中删除。

free_mem 的任务是检查一块新释放的内存，看它能否与左邻右舍进行合并。如果能，就调用 *merge* 把相邻的空闲块合并起来，并更新空闲链表。

mem_init 构造初始的空闲链表，包含了所有可用的内存。

最后一个文件是 *utility.c*，它包含了一些杂七杂八的函数，用在 PM 的不同地方。和 *alloc.c* 一样，*utility.c* 也未列在附录 B 中。

get_free_pid 为一个子进程查找一个空闲的 PID，它避免了一个可能会发生的问题。最大的 PID 值是 30 000，它应该是 *PID_t* 类型的最大值，但这个值被用做特殊的用途，即避免一些老的、使用较小数据类型的程序所带来的问题。在把 $PID = 20$ 赋给一个长久运行的进程后，可以创建和删除约 30 000 多个进程。每次当需要一个新的 PID 时，只要把一个变量加 1 即可，如果到达边界，则返回到开头，从 2 开始 (*init* 进程的 PID 永远是 1)。当然，进程之间的 PID 不能重复，如果把一个正在使用的 PID 再次分配出去，那么后果是灾难性的。因此，我们需要去搜索整个进程表，以确认一个 PID 未被使用。

函数 *allowed* 检查一个文件操作是否合法，例如，*do_exec* 需要知道某个文件是否是可执行的。

函数 *no_sys* 应该永远不会被调用，提供它只是为了预防用户在调用 PM 的时候，使用了一个无效的系统调用号。

当 PM 检测到一个非常严重的、无法恢复的错误时，会去调用 *panic*。它会向系统任务报告此次错误，使之紧急中止系统。因此，它不会被轻易调用。

utility.c 中的下一个函数是 *tell_fs*，当需要把 PM 所处理的事件通报给文件系统时，该函数将构造一条相应的消息，并发送给文件系统。

find_param 用于分析监控程序的参数。它的当前用途是在 MINIX 3 被装入内存之前，抽取出有关内存的各种信息。当然，如果需要的话，也可以用它来搜寻其他的信息。

文件中的另外两个函数提供了对库函数 *sys_getproc* 的接口，该函数负责调用系统任务，从进程表的内核部分获取信息。*sys_getproc* 实际上是在 *include/minix/syslib.h* 中定义的一个宏，负责把参数传递给 *sys_getinfo* 内核调用。*get_mem_map* 获得一个进程的内存映射。*get_stack_ptr* 获得当前的栈指针。这两个函数都需要一个进程号做参数，所谓的进程号，指的是进程表中的索引，它和

进程的 PID 是不同的。*utility.c* 中的最后一个函数是 *proc_from_pid*，它的功能是：输入一个进程的 PID，返回该进程在进程表中的索引。

4.9 小结

本章讨论了存储管理，包括存储管理的基本原理和 MINIX 3 中的存储管理。我们看到了最简单的存储管理系统，根本不需要交换和分页，当一个程序被装入内存后，就呆在原地不动，直到它运行结束。嵌入式系统的存储管理通常就是这样做的。有些操作系统在同一时刻只允许一个进程位于内存中，而有些操作系统则支持多道程序。

接下来是交换。在使用了交换技术以后，系统可以处理比内存所能容纳的更多的进程，得不到内存空间的进程将被换到磁盘上。内存和磁盘上的空闲空间可以用位图或空闲链表来管理。

更高级的计算机通常都有某种形式的虚拟存储器。在最简单的情形下，每个进程的地址空间被划分为相同大小的块，称为逻辑页面，它可以被放到内存中的任何一个物理页面中。人们提出了很多的页面置换算法，其中比较有名的两个是第二次机会算法和老化算法。为了使页式系统能很好地工作，仅仅选择一个好的页面置换算法是不够的，还需要注意工作集的确定、内存的分配策略、页面的大小等问题。

段式存储管理适合于处理在运行时大小可变的数据结构，并能简化链接和共享。它还可以为不同的段提供不同的保护。有时段式和页式结合起来构成一种二维的虚拟存储器，Intel 的 Pentium 处理器支持分段和分页。

MINIX 3 的存储管理是很简单的。当进程在执行 *fork* 或 *exec* 系统调用时，就会分配相应的内存空间。而且在进程的整个运行期间，分配给它的内存既不会增加也不会减小。在 Intel 处理器上，MINIX 3 使用了两种内存模型：小程序可以使用组合的指令和数据空间（简称 I 和 D 空间），把指令和数据放在同一个内存段中。而大程序可以使用独立的指令和数据空间，在这种方式下，各个进程可以共享它们的代码部分，因此，在执行 *fork* 时只有数据和栈的内存空间是必须分配的。这一点在执行 *exec* 时也可能成立，前提是已经有另外一个进程在使用新程序的指令代码。

空闲内存空间的管理采用的是空闲链表和首次匹配算法，但 PM 的大部分工作与此无关，而是去执行一些与进程管理有关的系统调用。许多系统调用都支持 POSIX 风格的信号，由于多数信号的默认动作是去结束接受信号的进程，所以把这项工作放在 PM 中是比较合适的，由 PM 来发起进程的终止。一些与内存没有直接关系的系统调用也放在 PM 中处理，这主要是因为 PM 比文件系统要小，所以放在这里比较方便。

习题

1. 一个计算机系统有足够的空间在它的内存中存放 4 个程序，这些程序都有一半的时间处于空闲状态，在等待 I/O 操作。请问，多大比例的 CPU 时间被浪费掉了？
2. 在一个使用交换技术的系统中，按地址顺序排列的内存中的空闲块大小是 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB 和 15 KB。对于连续的段请求：
 - (a) 12 KB
 - (b) 10 KB
 - (c) 9 KB

使用最先匹配算法,哪个空闲块将被选中?对最佳匹配法、最坏匹配法和下次匹配法回答同样的问题。

3. 一个计算机系统有 1 GB 的内存,内存的分配以 64 KB 为单位。如果使用位图来管理空闲内存,那么该位图需要占用多少个 KB 的空间?
4. 对于刚才的问题,现在改由空闲链表来管理。请问,在最好的情形下,该链表需要占用多大的内存空间?在最坏的情形下,该链表需要占用多大的内存空间?假设操作系统占用了最底端的 512 KB 内存。
5. 物理地址和虚拟地址之间的区别是什么?
6. 使用图 4.8 中的页表,计算出下列虚拟地址所对应的物理地址:
 - (a) 20
 - (b) 4100
 - (c) 8300
7. 在图 4.9 中,虚拟地址的页面字段是 4 位,而物理地址的页面字段是 3 位。一般来说,虚拟地址中的页面位数与物理地址中的页面位数相比,是更多、更少、相等还是都可以?讨论你的答案。
8. Intel 8086 处理器不支持虚拟存储器,然而一些公司曾经出售过一种基于未经修改的 8086 CPU 的分页系统。请做出一个合理的猜想,他们是如何做到这一点的?(提示:想一想 MMU 的逻辑位置。)
9. 假设一条指令需要 1 ns,一次缺页中断需要额外的 n ns,而且每隔 k 条指令会出现一次缺页中断。请给出指令的实际执行时间的公式。
10. 一台机器有 32 位地址空间和 8 KB 页面,页表完全用硬件实现,每个表项是一个 32 位的字。当一个进程开始运行时,页表被以每个字 100 ns 的速度从内存复制到硬件中。如果每个进程运行 100 ms(包含装入页表的时间),那么在全部的 CPU 时间中,有多大的比例被用来装入页表?
11. 一台 32 位地址的计算机使用了两级页表。虚拟地址被划分为三部分:9 位的顶级页表字段、11 位的二级页表字段和页内偏移。请问页面的大小是多少?在地址空间中总共有多少个页面?
12. 以下是一小段汇编语言程序(用文字来描述),用在一台页面大小为 512 字节的计算机上。该程序位于地址 1020,它的栈指针位于 8192(向 0 的方向增长)。请给出这个程序所产生的页面访问序列。假设每条指令占用 4 个字节(1 个字),对指令和数据的访问都应该包含在访问序列之中。

将地址为 6144 的字装入寄存器 0
把寄存器 0 压入栈中
调用起始地址为 5120 的函数,把返回地址入栈
把栈指针的值减去 16
将实参与常量 4 进行比较
如果相等,跳转到 5152
13. 假设一个 32 位的虚拟地址被分成 a, b, c, d 四个字段。前三个用于一个三级页表系统,第四个 d 是页内偏移。请问页面的个数是否依赖于所有的这四个字段?如果不是,那么与哪些字段有关?与哪些无关?

14. 在一台计算机上，进程的地址空间有 1024 个页面，页表被保存在内存中。从页表中读取一个字的开销是 500 ns。为了减小开销，这台计算机使用了 TLB，它能存放 32 对（虚拟页面号，物理页面号），查找时间为 100 ns。为了把页表的平均访问开销降到 200 ns，需要的 TLB 命中率是多少？
15. VAX 上的 TLB 没有 *R* 位，为什么？
16. 一台机器有 48 位虚拟地址和 32 位物理地址，页面大小为 8 KB，请问在页表中需要多少个表项？
17. 一个 RISC CPU 具有 64 位虚拟地址和 8 GB 的内存，它使用了反置页表机制，页面的大小为 8 KB。请问 TLB 的最小大小是多少？
18. 一台计算机有 4 个物理页面，每个页面的装入时间、最后访问时间、*R* 位和 *M* 位如下所示（时间以时钟节拍为单位）：

页面	装入时间	最后访问时间	<i>R</i>	<i>M</i>
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

- (a) NRU 将替换哪个页面？
 (b) FIFO 将替换哪个页面？
 (c) LRU 将替换哪个页面？
 (d) 第二次机会算法将替换哪个页面？
19. 假设有 8 个虚拟页面和 4 个物理页面，页面置换算法为 FIFO。在刚开始时，4 个物理页面均为空，那么对于访问序列 0172327103，将会发生多少次缺页中断？如果使用 LRU 算法呢？
20. 在一台小的计算机上，有 8 个物理页面，每个物理页面里面都装有一个虚拟页面，它们的顺序依次为 A, C, G, H, B, L, N, D 和 F。这些虚拟页面的装入时间分别是 18, 23, 5, 7, 32, 19, 3 和 8，它们的访问位分别是 1, 0, 1, 1, 0, 1, 1 和 0，它们的修改位分别是 1, 1, 1, 0, 0, 0, 1 和 1。如果采用第二次机会算法，那么它考虑的页面顺序如何？最终选中了哪个页面？
21. 是否存在这样一种情形，使得时钟页面置换算法和第二次机会算法所选中的被置换页面是不同的？如果有，是什么样的一种情形？
22. 假设一台计算机使用了 PFF 页面置换算法，但系统的内存容量非常大，能够装入所有的进程，而不会发生缺页中断。在这种情形下，会发生什么现象？
23. 一台小型计算机有 4 个物理页面。在第一个时钟节拍时 *R* 位是 0111（页面 0 是 0，其他是 1）。在随后的时钟节拍，这个值分别是 1011, 1010, 1101, 0010, 1010, 1100 和 0001。如果使用老化算法，计数器的长度为 8 位，请给出在最后一个时钟节拍后，这 4 个计数器的值。
24. 在一个计算机系统上，磁盘的平均寻道时间为 10 ms，旋转时间为 8 ms，每道的容量为 1 MB。如果要把一个 64 KB 的程序装入内存，这需要多长的时间？
- (a) 页面长度为 2 KB 时。
 (b) 页面长度为 4 KB 时。
 (c) 页面长度为 64 KB 时。
- 假设页面随机地分布在磁盘上。

25. 对于上述问题，为什么页面那么小？相对于 4 KB 的页面，请列出 64 KB 页面的两个缺点。
26. PDP-1 是最早的分时计算机之一，它的内存有 4K 个 18 位的字。在每个时刻它的内存中只有一个进程。当调度程序决定运行另一个进程时，内存中的进程将被写到一个页面鼓上，鼓的表面有 4K 个 18 位的字，鼓可以从任何地址开始读写。你认为为什么要选用这个鼓？
27. 一台嵌入式计算机为每个进程提供了 65 536 个字节的地址空间，并将它们划分为 4096 字节的页面。有一个程序，它的代码段为 32 768 个字节、数据段为 16 386 字节、栈段为 15 870 字节。请问，这个程序能被装入地址空间吗？如果页面大小为 512 字节，结果又如何？注意，在一个页面中，不能同时包含两个不同段的内容。
28. 人们已经观察到在两次缺页中断之间执行的指令数与分配给进程的物理页面数成正比。如果可用内存加倍，则缺页中断的平均间隔时间也加倍。假设一条普通的指令需要 $1 \mu s$ ，但如果发生了缺页中断，就需要 $2001 \mu s$ （即 2 ms）用于处理此次中断。假设一个程序需要运行 60 s，期间共发生了 15 000 次缺页中断。如果分配给它的内存是原来的两倍，那么这个程序需要运行多长时间？
29. Frugal 计算机公司的一个操作系统设计小组正在思考如何在他们的新操作系统中，减少对辅助存储器容量的需求。他们的技术专家建议，不需要把程序的代码保存在交换区中，而是在需要的时候直接从二进制文件中调页进来。请问这种方法有什么问题吗？
30. 解释内碎片和外碎片的区别。哪一个发生在页式存储管理系统中？哪一个发生在段式存储管理系统中？
31. 如果在系统中既使用段式又使用页式存储管理，就像 Pentium 处理器那样，那么首先要查找段描述符，然后是页面描述符。在 TLB 中，也是像这样用两级查找的方式来工作的吗？
32. 为什么在 MINIX 3 的存储管理方案中，必须有一个像 *chmem* 这样的程序？
33. 图 4.44 显示了 MINIX 3 系统前四个组件的初始内存使用情况，对于在 *rs* 后装入的下一个组件，它的 *cs* 值是多少。
34. 在 IBM 兼容机上，640 KB 到 1 MB 这段地址空间用于 ROM 和 I/O 设备内存，普通的程序不能访问。当 MINIX 3 引导监控程序把它自己重定位在 640 KB 以下后，程序可用的内存就进一步减少了。在图 4.44 中，假设引导监控程序占用了 52 256 字节，那么在内核与不可用的区域之间，还剩下多少内存空间可以用来装入一个程序？
35. 在上述问题中，在给引导监控程序分配存储空间的时候，是以字节为单位来分配，还是以 click 为单位来分配，这两种做法有区别吗？
36. 在 4.7.5 小节曾经指出，当执行 *exec* 系统调用时，在释放当前进程的内存映像之前，先要检查是否有足够的空间能容纳新的内存映像。这种实现方式只是一种次优的做法，请重新实现这个算法，以获得更好的性能。
37. 在 4.8.4 小节曾经指出，最好对代码段和数据段分别去搜索空闲区域，请实现这一改进。
38. 请重新设计 *adjust*，以避免接收信号的进程由于过于严格的栈空间检查而被不必要地杀死。
39. 为了设定 MINIX 3 进程的当前内存分配，用户可以使用命令

```
chmem +0 a.out
```

但这种做法有一个讨厌的副作用，即需要重写文件，从而修改它的日期和时间信息。请修改 *chmem*，实现一个新的命令 *showmem*，它将显示一个进程的当前内存分配。

第5章 文件系统

5.1	文件
5.2	目录
5.3	文件系统的实现
5.4	文件系统的安全性
5.5	保护机制
5.6	MINIX 3 文件系统 概述
5.7	MINIX 3 文件系统的 实现
5.8	小结

所有的计算机应用程序都需要存储和检索信息。当一个进程在运行时，它可以把一些信息保存在自己的地址空间中。不过，这种存储能力局限在虚拟地址空间的大小，而地址空间的大小是有限的。对于一些应用程序来说，这个大小可能是够用的，但对于其他一些应用来说，如航空订票系统、银行系统或公司记录保存系统，这个空间就显得不够用了。

除此之外，还有一个很重要的问题，当这个进程运行结束后，它所在的地址空间就被释放掉了，里面存放的信息也就丢失了。对于许多应用（如数据库）而言，它们的信息必须保存几周、几个月或者更长的时间。如果信息随着使用进程的终止而丢失，这是不可接受的。此外，即使系统突然崩溃，这些信息也应该设法保存下来。

第三个问题是，多个进程可能需要同时去访问某些信息。如果我们在某个进程的地址空间中存放了一个在线电话簿，那么只有该进程可以对它进行访问。解决这个问题的方法就是使信息本身是独立于任何一个进程的。

因此，对于长期的信息存储，我们有如下三个基本要求：

1. 必须能存储大量的信息。
2. 在使用信息的进程终止时，信息必须保存下来。
3. 多个进程可以并发地访问这些信息。

解决所有这些问题的常用方法就是把信息以文件（file）为单位，存储在磁盘或其他外部介质上。然后，进程可以读取文件，也可以在需要的时候创建新文件。存储在文件中的信息必须是永久性的，也就是说，不会受到进程的创建和终止的影响。只有当用户明确地删除它时，文件才会消失。

文件是由操作系统来管理的，包括文件的结构、文件的命名、文件的使用、文件的保护和文件的实现等，这些都是在操作系统的设计中需要解决的问题。总而言之，在一个操作系统中，负责处理与文件有关的各种事情的那一部分，就称为文件系统（file system）。

我们可以从两种不同的观点来看待文件系统。第一种是用户的观点，对于用户来说，他比较关心的是文件系统所提供的对外接口，包括文件是如何来命名的、如何来保护的、如何来访问的。第二种是操作系统的观点，对于操作系统的设计师来说，他比较关心的是如何来实现与文件有关的各个功能模块，包括空闲存储空间的管理、文件系统的布局、逻辑块的大小等。基于以上原因，我们将本章的内容组织成几个小节，前两个小节与用户接口有关，然后讨论文件系统的各种实现方法。接下来是文件的安全性和保护机制，最后我们将介绍 MINIX 3 文件系统实例。

5.1 文件

首先我们将从用户的角度出发，介绍文件系统的对外接口，也就是说，文件是如何来使用的，它们具有哪些属性。

5.1.1 文件的命名

文件是一种抽象机制，它提供了一种把信息保存在磁盘等外部存储设备上，并且便于以后访问的方法。这种抽象性体现在，用户不必去关心具体的实现细节，例如，这些信息被存放在什么地方，是如何存放的，磁盘的工作原理是什么，等等。

对于任何一种抽象机制来说，可能最重要的特性就是对管理对象的命名方式。当进程创建一个文件时，必须给它指定一个名字；当进程终止时，这个文件继续存在，别的进程可以通过它的名字来访问它。

文件的具体命名规则并无统一的标准，不同的系统可能会有不同的要求。不过当前的所有系统都支持使用长度为 1 到 8 个字符的字符串作为合法的文件名。因此，*andrea*, *bruce* 和 *cathy* 都可以用做文件名。有时，数字和一些特殊字符也可以用于文件名之中，所以像 2, *urgent!* 和 Fig.2-14 通常也是有效的文件名。许多文件系统还支持长达 255 个字符的文件名。

有些文件系统会区分英文字母的大小写，如 UNIX（及其演化版本），而有的系统则不会，如 MS-DOS。因此，在 UNIX 系统中，可以使用如下三个不同的文件名：*maria*, *Maria* 和 *MARIA*。但是在 MS-DOS 中，这三个名字是等效的，描述的是同一个文件。

Windows 位于两者之间。Windows 95 和 Windows 98 的文件系统都基于 MS-DOS 文件系统，继承了它的许多属性，如文件名的构造方式。随着新版本的发行，不断有新的改进添加进来，但是我们讨论的特性主要还是基于 MS-DOS 和“经典”的 Windows 版本。另外，Windows NT, Windows 2000 和 Windows XP 都支持 MS-DOS 文件系统。不过，它们还支持一种“正宗”的文件系统 NTFS，NTFS 具有一些不同的属性，如 Unicode 文件名。这个文件系统同样也在不同地更新和完善。在本章中，当我们说到“老的系统”时，指的是 Windows 98 文件系统。如果它有一个特性不适用于 MS-DOS 或 Windows 95，我们就会这么说。同样，当我们说到“新的系统”时，指的是 NTFS 或 Windows XP 文件系统。如果它有一个特性不适用于 Windows NT 或 Windows 2000，我们就会这么说。另外，如果我们只说 Windows，那么指的是自 Windows 95 以来的所有 Windows 文件系统。

许多操作系统支持两部分组成的文件名，两部分之间用句点隔开，比如 *prog.c*。在句点后面的部分称为文件扩展名（file extension），它通常给出了与文件的类型有关的一些信息。在本例中，*.c* 表示这是一个 C 语言源文件。在 MS-DOS 中，文件名由 1~8 个字符和 1~3 个字符的可选扩展名组成。在 UNIX 中，如果使用扩展名，则其长度完全由用户决定，甚至在一个文件名中可以包含两个或多个扩展名。如 *prog.c.bz2*，其中 *.bz2* 通常表明文件（*prog.c*）已经使用 bzip2 算法压缩过。一些常用的文件扩展名及其含义如图 5.1 所示。

在有些系统中（如 UNIX），文件扩展名仅仅是一种惯例，并不强迫使用。例如，名为 *file.txt* 的文件可能是一个文本文件，但这个文件名主要用于提醒用户，而不是要给计算机传递什么特别的信息。当然，也有例外的情形。比如说，C 编译器可能会要求源文件的名字必须用 *.c* 结尾，否则它将拒绝编译。

如果一个程序可以处理多种不同类型的文件，那么上述的命名惯例是很有用的。例如，C 编译器可以编译、链接多个文件，其中有些是 C 文件（如 *foo.c*），有些是汇编语言文件（如 *bar.s*），还有一些是目标文件（如 *other.o*）。在这种情形下，对于编译器而言，扩展名是非常重要的，它正是通过扩展名来区分哪些是 C 文件、哪些是汇编文件、哪些是目标文件。

相反，Windows 非常重视扩展名，并给它们赋予了含义。用户（或进程）可以向操作系统注册扩展名，并且为每种扩展名指定相应的应用程序。这样，如果用户去双击一个文件名，那么系统就

会自动地去运行相应的程序，并且把这个文件名作为它的参数。例如，如果去双击文件*file.doc*，那么系统就会自动地去运行Word程序，并且打开这个文档。

扩展名	含义
file.bak	备份文件
file.c	C源程序
file.gif	图形交换格式 (Graphical Interchange Format)
file.html	万维网超文本标记语言文档
file.iso	CD-ROM的一个ISO映像 (用于光盘刻录)
file.jpg	基于JPEG标准的静态图像编码
file.mp3	使用MPEG第3层的声音格式编码的音乐
file.mpg	用MPEG标准编码的影片
file.o	目标文件 (编译器的输出，但尚未链接)
file.pdf	便携式文档格式 (Portable Document Format) 文件
file.ps	PostScript文件
file.tex	用于TEX格式化程序的输入
file.txt	一般的文本文件
file.zip	压缩存档

图 5.1 一些典型的文件扩展名

有些人可能会觉得奇怪，文件的扩展名是非常重要的，但是在默认的设置下，Windows的常用扩展名却是不可见的。不过不要紧，对于一个熟悉系统的用户来说，他知道去哪儿修改这些“错误”的默认设置。

5.1.2 文件的结构

文件可以被组织成不同的结构，图 5.2 列出了最常用的三种结构。图 5.2(a)中的文件是一个无结构的字节流。操作系统既不知道也不关心文件的内容是什么，它所见到的只有字节。所有的含义只能由用户层的程序来解释。UNIX 和 Windows 98 都采用了这种方法。

操作系统把文件看成是简单的字节流，这种方式提供了很大的灵活性。用户程序可以在文件中加入任何内容，并且以任何方便的形式来命名。操作系统不会提供帮助，但也不会设置障碍。对于那些需要做特殊事情的用户来说，后者可能更为重要。

第二种结构如图 5.2(b)所示。在这种模型中，文件是由一序列固定长度的记录所组成的，每条记录都有某种内部结构。这种结构的核心思想是：读操作返回一条记录，而写操作重写或追加一条记录。在多年以前，当 80 列的穿孔卡片还在广泛使用的时候，许多操作系统把它们的文件系统建立在由 80 个字符的记录所组成的文件上。这些系统也支持由 132 个字符的记录所组成的文件，主要用于行式打印机（当时是 132 列的行式打印机）。程序以 80 个字符为单位读入数据，然后以 132 个字符为单位输出数据。当然，最后的 52 个字符有可能都是空格。对于现代的通用系统而言，已经没有人采用这种方法。

第三种文件结构如图 5.2(c)所示。在这种方式下，文件由一棵记录树组成，每条记录的长度可能是不同的，而且在记录内的某个固定位置，有一个关键字 (key) 字段。树的存储是依据关键字来进行的，以利于记录的快速查找。

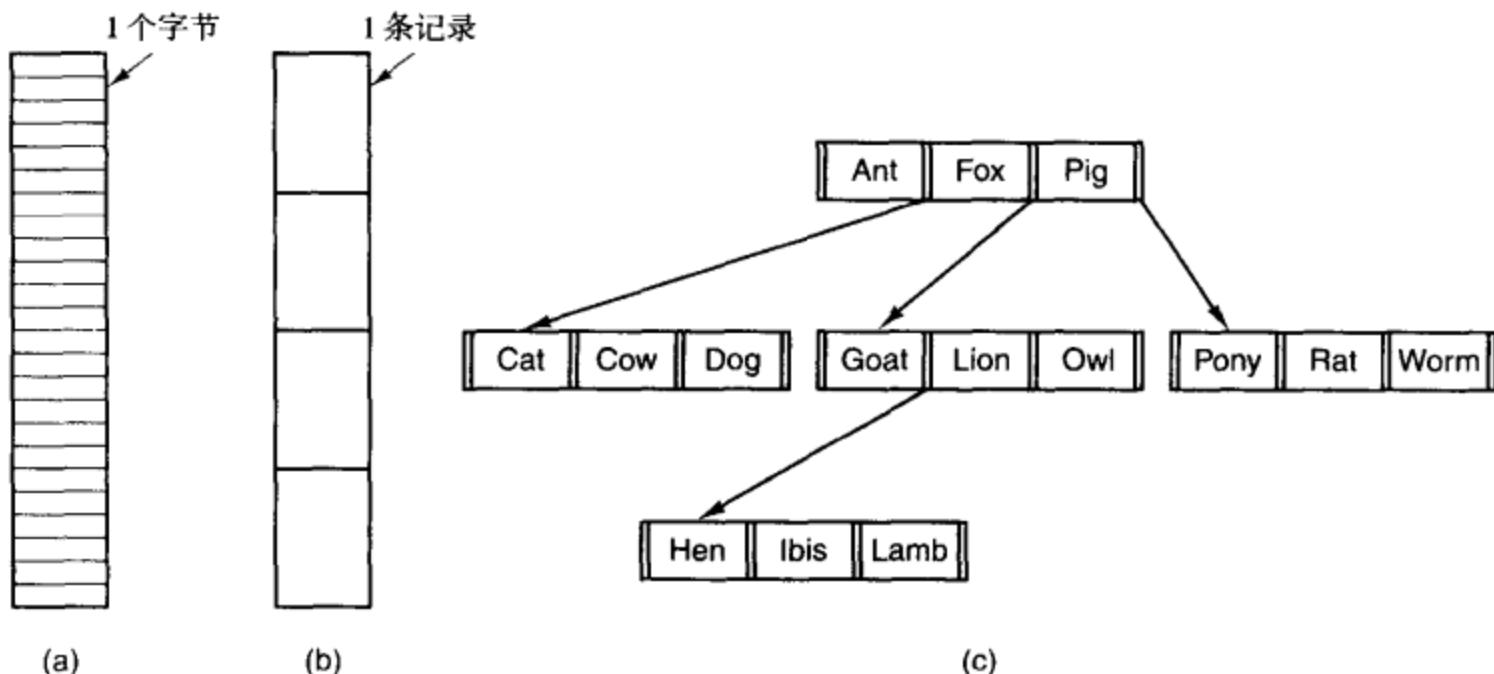


图 5.2 三种类型的文件: (a)字节序列; (b)记录序列; (c)树

在这种文件结构中，基本操作不是去访问“下一条”记录，而是去访问具有特定关键字的记录。例如，对于图 5.2(c)中的文件 zoo，用户可以要求系统提交关键字为 *pony* 的记录，而不必关心该记录在文件中的确切位置。此外，新的记录可以被添加到文件中，这时，是由操作系统而不是用户来决定把记录存放在文件的什么位置。这种文件结构与 UNIX 和 Windows 98 中使用的无结构字节流显然不同，它主要用在商业数据处理领域的一些大型计算机中。

5.1.3 文件的类型

大多数操作系统都支持多种类型的文件。例如，在 UNIX 和 Windows 中，都有常规的文件和目录。UNIX 还有字符设备文件和块设备文件。而 Windows XP 还有元数据 (metadata) 文件，这一点我们后面还会提到。常规文件 (regular file) 用来存放用户的信息，图 5.2 中的所有文件都是常规文件。目录 (directory) 是管理文件系统的组织结构的系统文件，我们后面再讲。字符设备文件 (character special file) 与输入 / 输出有关，用于处理各种串行 I/O 设备，如终端、打印机和网络等。块设备文件 (block special file) 则用于处理磁盘。在本章中我们主要讨论常规文件。

一般来说，常规文件主要有两种：ASCII文件和二进制文件。ASCII文件由一行行的文本组成。在有些系统中，每一行使用一个回车符来结束，而有些系统则使用了换行符。还有一些系统，如Windows，既用到了回车符，又用到了换行符。文本文件的每一行的长度可能是不一样的。

ASCII文件的最大优点是可以原样地显示和打印,也可以用通常的文本编辑器进行编辑。此外,如果许多程序都以ASCII文件作为输入和输出,那么就很容易把一个程序的输出作为另一个程序的输入,如同shell管道那样(用管道方式来实现进程间通信并非更容易,但如果采用一种公认的标准来表示它,如ASCII码,那么对信息的解释将会更容易)。

ASCII文件之外的所有常规文件，都可以视为二进制文件。如果你试图去打印一个二进制文件，那么看到的将是一系列乱七八糟的随机符号。对于用户来说，二进制文件是没法直接看的，但对于使用它们的程序来说，文件的内部是具有一定结构的。

例如，在图5.3(a)中，我们可以看到一个简单的可执行二进制文件，它取自于一个早期的UNIX版本。虽然从技术上来说，这个文件只是一个字节序列，但对于操作系统来说，只有当这个文件具有正确的格式时，才会去执行它。这种文件有五个段：文件头、代码、数据、重定位位和符号表。文件头以所谓的魔数（magic number）开始，表明该文件是一个可执行文件（这样可以防止意外执

行非此类格式的文件)。接下来是文件的各个部分的长度、执行的起始地址和一些标志位。在文件头之后就是程序本身的代码和数据，在它们被装载到内存中时，使用重定位位来进行定位。符号表则用于调试。

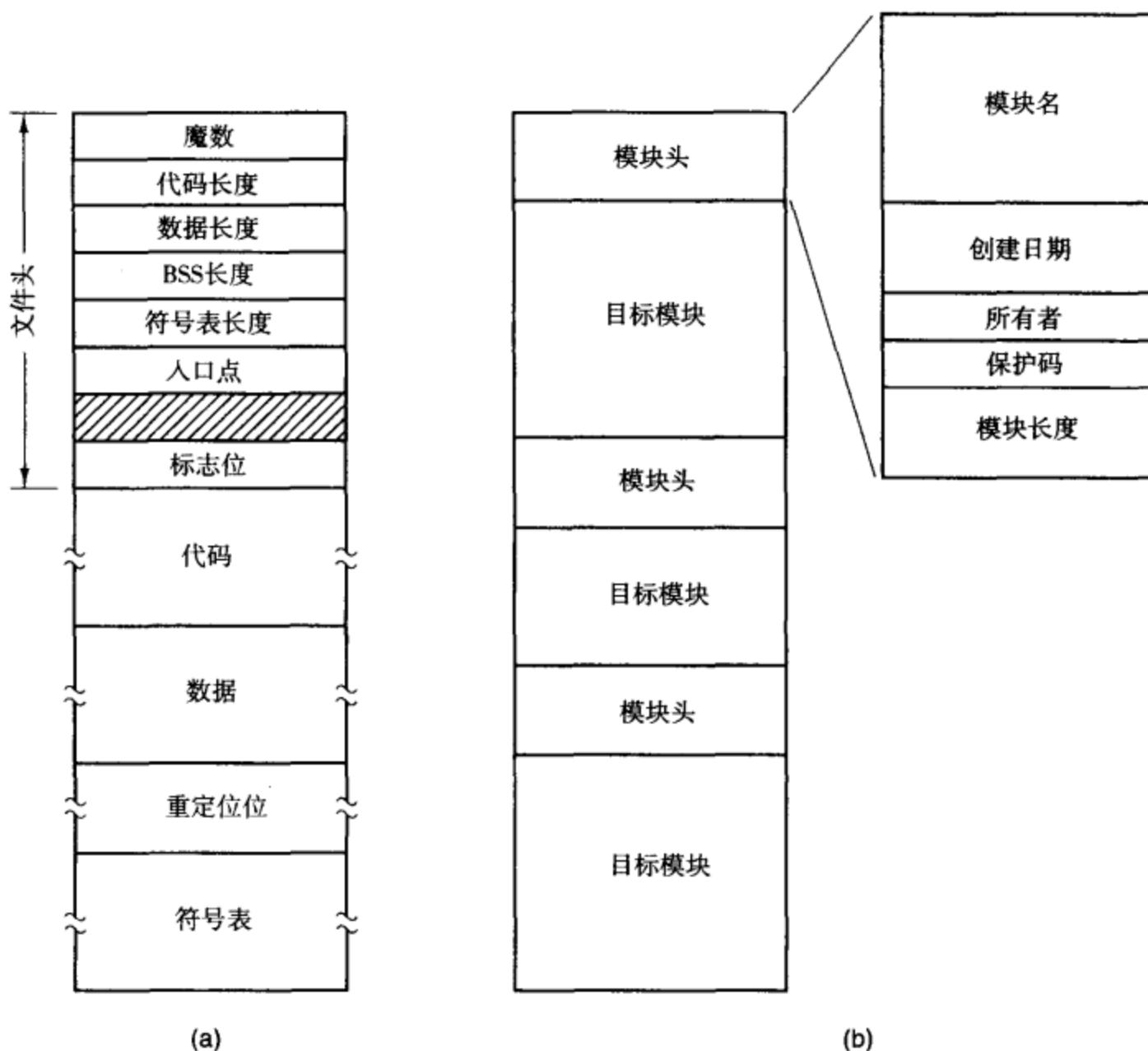


图 5.3 (a)一个可执行文件; (b)一个存档文件

二进制文件的第二个例子是一个UNIX存档文件，它由一组编译过但还没有链接的库函数(模块)组成。每个模块以模块头开始，其中给出了模块名、创建日期、所有者、保护码和模块长度等信息。如同可执行文件一样，模块头中也充斥着二进制数字，把它们在打印机上输出是毫无意义的。

每个操作系统至少需要识别一种文件类型：它自己的可执行文件，但很多操作系统都能识别更多的文件类型。老的TOPS-20系统(用于DECsystem 20)甚至还去检查每个即将执行的文件的创建时间，然后找到相应的源文件，看看源文件是否曾经被修改过。如果是的话，它就会重新编译源文件。而在UNIX中，它也能实现类似的功能，它的做法是把*make*程序集成到了*shell*中。在这里，文件的扩展名是强制规定的，这样操作系统才能确定哪一个二进制程序是由哪一个源文件生成的。

对于这种强制性的扩展名，如果用户执行了一些系统设计者未曾预料的操作，那么就可能会引起麻烦。例如，假设在一个系统中，程序输出文件的扩展名为.dat(数据文件)。如果用户编写了一个程序格式化器，输入一个.c文件(C程序)，对它进行转换(例如把它转换成标准的行首缩进风格)，然后再把转换后的文件输出，这样，输出文件的扩展名为.dat。如果用户试图用C编译器来编

译这个文件，将无法成功，因为文件的扩展名不符，编译器将拒绝编译。如果用户想另辟蹊径，把 *file.dat* 复制为 *file.c*，这也是徒劳的，系统会认为这种操作是无效的（防止用户误操作）。

虽然这种“用户友好性”对初学者有帮助，却使得一些有经验的用户大伤脑筋。他们不得不花很大的精力来应对操作系统对合理操作和不合理操作的区分。

5.1.4 文件的访问

早期的操作系统只提供一种文件访问方式：顺序访问（sequential access）。在这些系统中，进程可以从文件的起点开始，顺序地读取文件中的所有字节或记录，但不能够跳过某些内容，进行非顺序的读取。顺序文件可以重绕（即倒带），这样就可以根据需要，多次去读取该文件。如果存储媒体是磁带，而不是磁盘，那么使用顺序文件还是非常方便的。

在使用磁盘来存储文件后，我们可以非顺序地读取文件中的字节或记录，或者根据关键字而不是位置来访问记录。能够以任何顺序读取的文件称为随机访问文件（random access file）。

对于许多应用程序来说，随机访问文件是必不可少的，如数据库系统。如果一名乘客打电话，想要预订某次航班的机票。那么订票程序必须能直接访问该航班的记录，而不必先读出成千上万条其他航班的记录。

对于随机访问文件，有两种方法来指明文件读取的起始位置。第一种是，在每次 `read` 操作中，都给出此次读操作的起始位置；第二种是，提供一个特殊的 `seek` 操作来设置当前位置，在执行一次 `seek` 操作后，文件的读取操作将从这个新的当前位置开始进行。

在一些早期的大型机操作系统中，当一个文件被创建时，就对它进行分类，即分类为顺序文件或随机访问文件。对于不同类型的文件，系统将采用不同的存储技术。而在现代操作系统中，通常不进行这种区分，所有的文件都是随机访问文件。

5.1.5 文件的属性

每个文件都有文件名和文件数据。此外，操作系统还会赋予文件其他一些信息，如文件创建日期和时间、文件的长度等。我们把这些额外的信息称为文件的属性（attribute），也有些人把它们称为元数据（metadata）。对于不同的系统，文件的属性差别很大。图 5.4 中的表格列出了一些可能的属性。没有一个系统具有所有这些属性，但每一种属性都在某一个系统中使用过。

开头的四个属性与文件保护有关，它指明了谁可以访问这个文件，谁不能访问这个文件。在具体实现上，存在着各种不同的文件保护方案，后面我们会讨论其中的一些。在有些系统中，用户必须给出口令才能访问文件。这时，口令也是文件的属性之一。

标志是一些位或短字段，用来控制或启用某些特定的属性。例如，被隐藏的文件不会出现在文件的显示列表中。存档标志位用来标记一个文件是否备份过。当文件被修改后，操作系统就会设置这个标志位，这样，备份程序就可以区分出哪些文件需要备份。然后，当备份程序对它进行备份后，就会清除该标志位。临时标志位表示当创建该文件的进程终止后，它将被自动删除。

记录长度、关键字位置和关键字长度等字段只会出现在那些能够用关键字来查找记录的文件之中，它们提供了查找关键字所需的信息。

各个时间字段记录了文件的创建时间、最近访问时间和最近修改时间等，它们可用于不同的目的。例如，如果源文件的修改时间要晚于相应的目标文件的生成时间，那么该源文件就需要重新编译，而这些字段就提供了必要的信息。

属性	含义
保护	谁能访问该文件, 以何种方式访问
口令	访问该文件所需的口令
创建者	文件创建者的ID
所有者	当前所有者
只读标志位	0表示读/写, 1表示只读
隐藏标志位	0表示正常, 1表示不在列表中显示
系统标志位	0表示正常文件, 1表示系统文件
存档标志位	0表示已备份过, 1表示需要备份
ASCII/二进制标志位	0表示ASCII文件, 1表示二进制文件
随机访问标志位	0表示只能顺序访问, 1表示随机访问
临时标志位	0表示正常, 1表示在进程退出时需要删除该文件
锁标志位	0表示未锁, 非零表示已锁
记录长度	一条记录的字节数
关键字位置	每条记录内的关键字偏移
关键字长度	关键字字段的字节数
创建时间	文件的创建日期和时间
最后访问时间	文件的最后访问日期和时间
最后修改时间	文件的最后修改时期和时间
当前长度	文件的字节数
最大长度	文件允许的最大字节数

图 5.4 一些可能的文件属性

当前长度给出了文件当前的大小。在一些早期的大型机操作系统中, 当一个文件被创建时, 需要设定文件的最大长度, 以便操作系统事先保留相应的存储空间。而现代操作系统能自行处理这个问题, 因此就无须这个属性。

5.1.6 文件的操作

如前所述, 文件用于存储信息, 以便于以后的检索。不同的系统提供了不同的操作来进行存储和检索。以下是一些常用的与文件有关的系统调用:

1. **Create**。创建一个文件, 不带任何数据。该调用的目的是声明文件的存在, 并用来设置一些属性。
2. **Delete**。当文件不再需要用到时, 必须删除它以释放磁盘空间。该系统调用可以实现这一目的。
3. **Open**。在使用一个文件之前, 必须先打开它。**open** 调用的目的是: 使系统能将文件的属性和磁盘地址列表载入内存, 因为后续的调用需要用到这些信息。
4. **Close**。当文件访问结束后, 文件的属性和磁盘地址就不再需要了, 这时应该关闭文件以释放一些内部表格空间。许多系统限制进程的打开文件数, 以鼓励用户关闭不再使用的文件。磁盘以块为单位写入, 而关闭文件可以使文件的最后一个块被写回磁盘, 尽管这个块可能还没有写满。
5. **Read**。从文件中读数据。一般来说, 读取的数据来自于文件的当前位置。调用者必须指明需要读取多少数据, 并提供一个缓冲区来存放这些数据。

6. **Write**。向文件中写入数据，写操作一般也是从文件的当前位置开始。如果当前位置在文件的末尾，则文件长度增加；如果当前位置在文件的中间，则现有数据被覆盖，并永远丢失。
7. **Append**。该调用是 **write** 的一种受限形式，它只能在文件的末尾添加数据。如果系统只提供一组最小的系统调用集，则通常不会把 **append** 包含进来。但许多系统往往会对同一个操作提供多种不同的实现方法，在这些系统中往往有 **append** 调用。
8. **Seek**。对于随机访问的文件，需要指定从哪儿开始读写数据，常用的方法是使用一个 **seek** 系统调用，把文件指针指向文件中的某个特定位置。此后，文件的读、写操作都是从这个位置开始的。
9. **Get attributes**。进程往往需要读取文件的属性。例如，UNIX 中的 *make* 程序常用于管理由多个源文件组成的软件开发项目。在调用 *make* 时，它会检查所有源文件和目标文件的修改时间，并安排最小数目的编译，使得所有文件都成为最新版本。为了实现这个目标，就必须去查找文件的某些属性，如修改时间。
10. **Set attributes**。有些属性是可以设置的，在文件创建之后，用户可以去修改它们，而这就需要用到本调用。保护模式信息是一个很典型的例子，大多数的标志位也属于此类。
11. **Rename**。用户常常需要改变现有文件的文件名，系统调用 **rename** 可以实现这一目的。严格说来，这个调用并非必要，因为我们可以先把文件复制到一个带有新文件名的文件中，然后删除原来的文件。
12. **Lock**。锁定一个文件或文件的一部分可以防止多个进程同时对它进行访问。例如，在一个航空订票系统中，在预订座位的时候必须锁定数据库，以避免发生同一个座位被售给两位不同的旅客的情形。

5.2 目录

为了管理文件，文件系统通常需要用到目录（directory）或文件夹（folder），在许多系统中，目录本身也是文件。在本小节，我们将讨论目录、目录的组织结构、目录的属性和目录的操作。

5.2.1 简单的目录系统

一个目录通常包含有许多个目录项，每个目录项代表一个文件。图 5.5(a)中是一种可能的情形，在每个目录项中，包含有文件的名字、属性和文件数据在磁盘上的存储地址等信息。另一种情形如图 5.5(b)所示，在每个目录项中，包含有文件的名字和一个指向另一个数据结构的指针，文件的属性和磁盘地址等信息就存放在这个数据结构中。以上这两种方案都得到了广泛应用。

当一个文件被打开时，操作系统会去搜索它所在的目录，直到找到了这个文件的文件名。然后从相应的目录项或目录项所指向的数据结构中取得文件的属性和磁盘地址信息，并把它们放入内存中的一个表格中。之后，当我们需要访问该文件的时候，就可以直接去使用内存中的这些信息。

每个系统的目录个数各不相同。最简单的设计方案是维护一个仅有的目录，其中包含所有用户的所有文件，如图 5.6(a)所示。在早期的个人计算机上，这种单一目录系统很普遍，因为只有一个用户。

如果在多用户环境下使用这种单一目录系统，可能会出现重名的问题，即不同的用户给他们的文件起了相同的名字。例如，假设用户 A 创建了一个名为 *mailbox* 的文件，后来用户 B 也创建了一个名为 *mailbox* 的文件，这样，B 的文件就把 A 的文件给覆盖掉了。所以说，这种方案不能再用于多用户的系统中，但可以用在一些小型的嵌入式系统中，如手持式个人数字助理或手机。

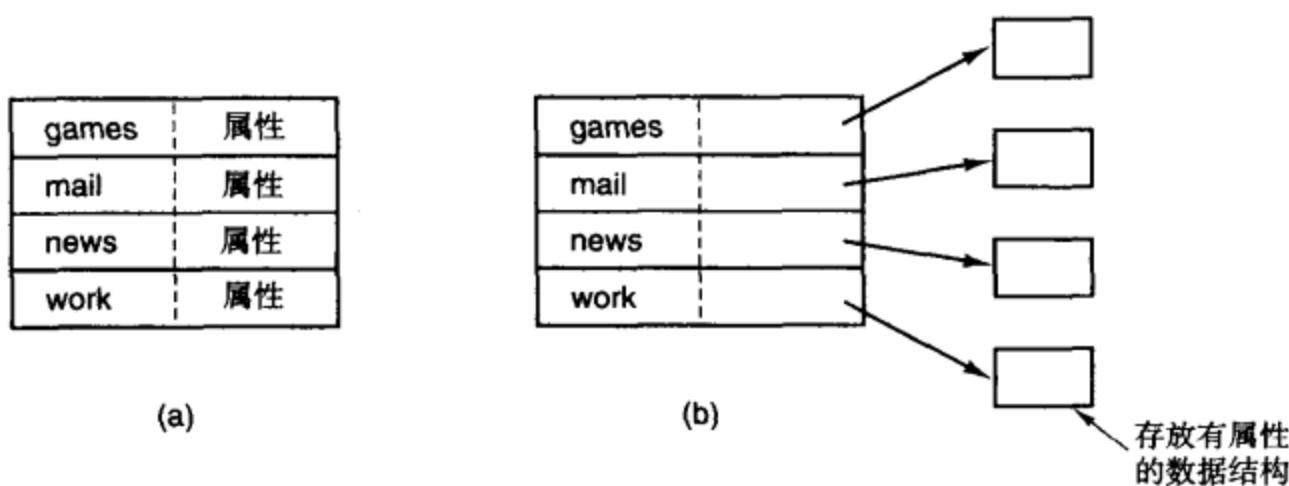


图 5.5 (a)属性存放在目录项中; (b)属性存放在单独的数据结构中

为了避免上述的重名冲突问题，下一步就是让每一位用户拥有一个独立的目录，这样，一个用户所使用的名字就与另一个用户所使用的名字无关，即使在两个或多个目录下出现了相同的一个文件名，也不会有问题。这种设计导致了图 5.6(b)所示的目录系统，它可以用在一台多用户的计算机上，或者是在一个简单的 PC 机网络上，用户通过局域网来共享一个公用的文件服务器。

在这种设计中，当一个用户试图去打开一个文件时，操作系统必须知道这个用户是谁，这样它才知道应该去搜索哪一个目录。因此，我们需要一个登录程序，让用户来指定一个登录名或登录标识。而在前面的单一目录系统中，则没有这样的要求。另外，在这种目录系统中，用户只能去访问他自己目录下的文件。

5.2.2 层状目录系统

二级目录结构消除了不同用户之间的文件名冲突，但仍有问题。对于那些文件个数比较多的用户来说，他们需要把自己的文件按某种方式组织起来。例如，一个教授可能希望把分发给学生的课堂讲义文件和他正在编写的一本新书的手稿文件区分开来。因此，我们需要的实际上是一种通用的层次结构（即目录树）。在这种方式下，用户可以根据自己的需要来创建子目录，从而把他们的文件分成不同的组。这种方法如图 5.6(c)所示。图中，在根目录下有 A, B, C 三个目录，它们分别属于不同的用户，其中有两个用户为他们正在做的项目创建了一些子目录。

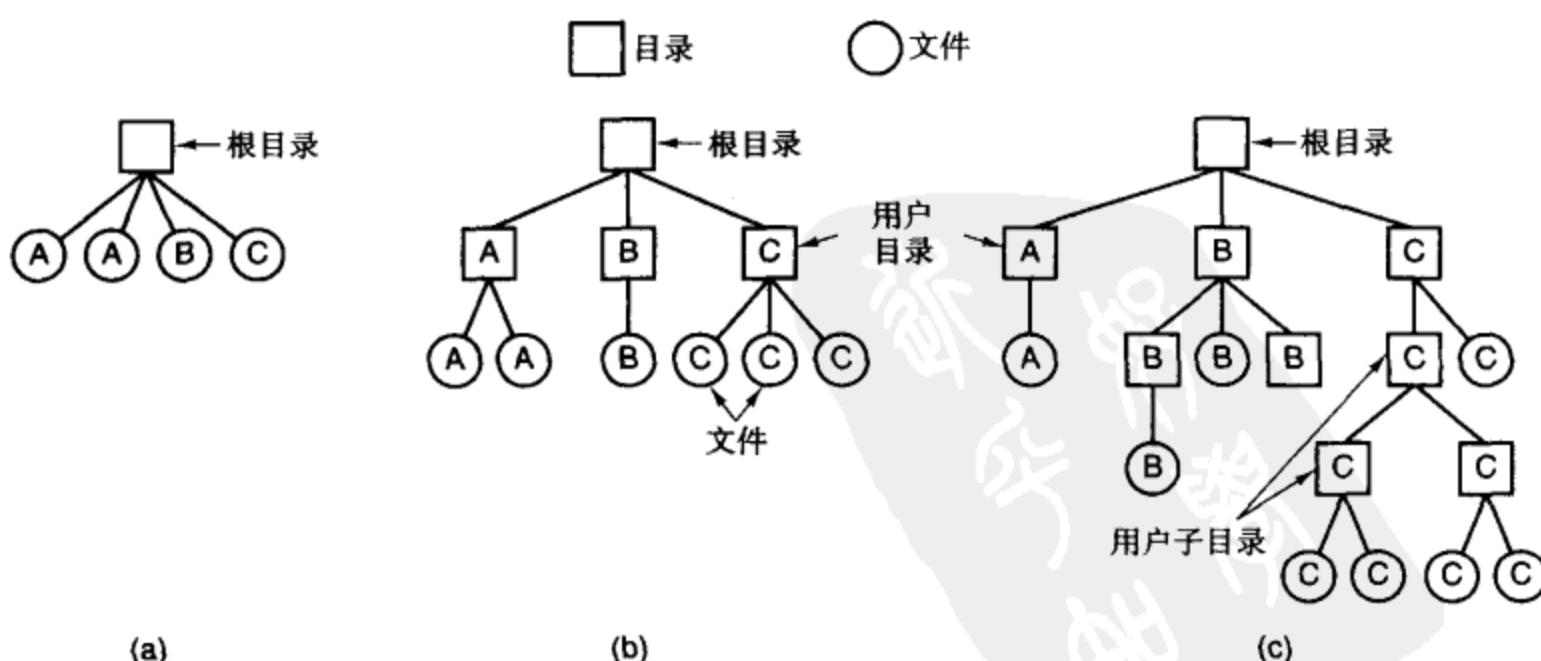


图 5.6 三种文件系统设计：(a)所有用户共享同一个目录；(b)每个用户有一个单独的目录；(c)每个用户有一棵目录树。图中的字符表示目录或文件所有者

由于能够创建任意数量的子目录，因此用户可以很方便地来组织他们的工作。基于这个原因，几乎所有的现代 PC 机和服务器文件系统都采用了这种层状结构的方式。

不过，正如我们前面所说的，随着新技术的出现，历史总是在不断地重复它自己。数码相机需要把它们的图像保存在某个地方，通常是在一个 flash 存储卡中。最早的数码相机采用的是单一目录结构，把文件命名为 *DSC0001.JPG, DSC0002.JPG* 等。然而，没过多久，相机的生产商就构造了带有多个目录的文件系统，如图 5.6(b) 所示。实际上，对于数码相机的主人来说，根本就不知道如何去使用多重目录，而且即使知道，而想不出来需要用它来做什么，毕竟这只是一个相机上的文件系统。但对于厂商来说，反正是一种嵌入式软件，也不会增加什么成本，所以能用二级结构当然更好。也许在不远的将来，在数码相机上也要用上层次结构的文件系统、多个用户名、255 个字符的长文件名等，谁知道呢？

5.2.3 路径名

当文件系统被组织成一棵目录树时，需要用某种方法来指定文件名。通常的方法主要有两种。第一种是，每个文件都被赋予一个绝对路径名（absolute path name），它由从根目录开始一直到该文件的路径组成。例如，路径 */usr/ast/mailbox* 表示在根目录下有一个子目录 *usr/*，它又包含了子目录 *ast/*，而文件 *mailbox* 就放在子目录 *ast/* 下。绝对路径名总是从根目录开始，并且是唯一的。在 UNIX 中，路径名的各个部分之间用“/”分隔。而在 Windows 系统中，分隔符是“\”。因此，在这两种系统中，同一个路径名可能有两种不同的写法：

Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox

不管使用哪一个分隔符，只要路径名的第一个字符是分隔符，那么这个路径名就是绝对路径名。

第二种命名方式是相对路径名（relative path name）。它常和工作目录（working directory，也称当前目录，current directory）的概念一起使用。用户可以指定一个目录作为当前的工作目录。这时，所有的路径名，如果不是从根目录开始，那么都是相对于这个工作目录的。例如，如果当前的工作目录是 */usr/ast*，则绝对路径名为 */usr/ast/mailbox* 的文件可以简单地用 *mailbox* 来访问。换句话说，以下两条 UNIX 命令是完全等效的：

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak  
cp mailbox mailbox.bak
```

相对路径名的形式更加简洁、方便，但是它的功能和绝对路径名是相同的。

有些程序需要访问某个特定的文件，而不管当前的工作目录是什么。在这种情形下，它应该使用绝对路径名。例如，一个拼写检查程序在运行时需要读取文件 */usr/lib/dictionary*，这时它应该使用完整的绝对路径名，因为当它在执行的时候，并不知道当前的工作目录是什么。但无论当前的工作目录是什么，使用绝对路径名总是没错的。

当然，如果这个拼写检查程序需要从目录 */usr/lib/* 中读取很多文件，那么使用绝对路径名就显得有些繁琐了。这时，可以采用另一种方法，即执行一个系统调用，把当前工作目录切换到 */usr/lib/*，然后直接使用 *dictionary* 作为 *open* 的第一个参数。通过这种方式明确地改变工作目录，程序可以知道它在目录树中的确切位置，因此可以使用相对路径名。

系统中的每个进程都有自己的工作目录，因而，当一个进程改变了它的工作目录并退出后，其他的进程并不会受到影响，而且在文件系统中也不会留下改变的痕迹。因此，对于进程来说，工作

目录的切换是安全的，只要需要，它就可以改变当前的工作目录。不过，如果在库函数中改变了工作目录，而且在它运行结束前没有把目录改回去，那么程序的其他部分可能就无法正常运行，因为它们所假定的当前工作目录已经无效了。所以，在库函数中，很少改变工作目录，如果非改不可，则总是在返回之前改回到原来的工作目录。

大多数支持层次目录结构的操作系统，在每个目录中有两个特殊的目录项“.”和“..”，通常念做“dot”和“dotdot”。dot指当前目录，dotdot指其父目录。要了解它们是如何使用的，我们可以看一下图5.7中的UNIX文件树。某个进程的工作目录是`/usr/ast/`，它可以使用“..”向上到达其父目录`/usr`。例如，它可以使用shell命令

```
cp .. /lib/dictionary.
```

把文件`/usr/lib/dictionary`复制到自己的目录下。第一个路径告诉系统往上走（到`usr`目录），然后再往下走到达`lib/`目录，并找到`dictionary`文件。

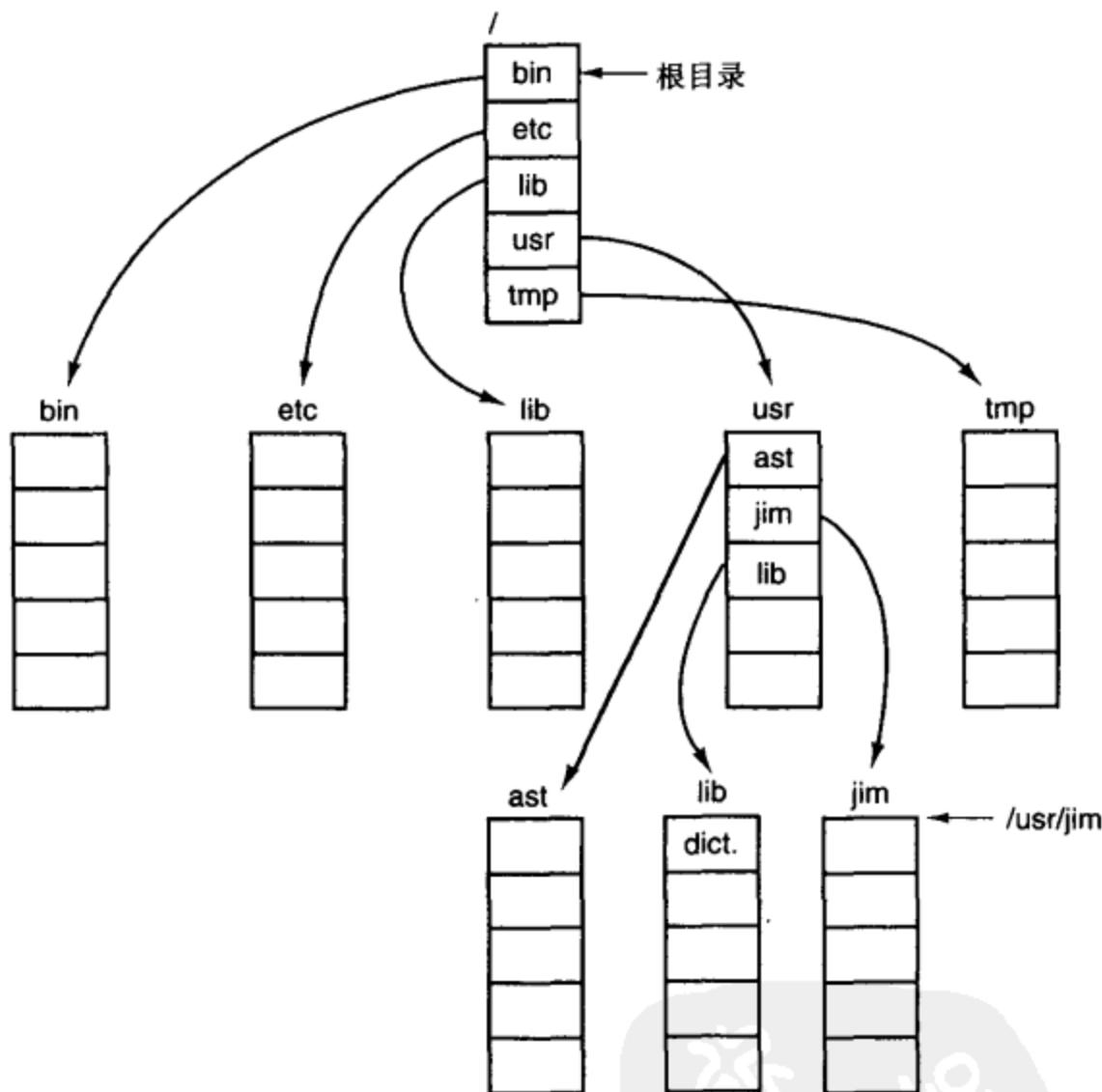


图5.7 一棵UNIX目录树

第二个参数（“.”）表示当前目录，如果`cp`命令得到一个目录名（包括“.”）作为它的最后一个参数，那么它就会把所有的文件都复制到那里。当然，对于上述的复制操作，更常用的方法是键入

```
cp /usr/lib/dictionary .
```

这里，用户使用“.”避免了第二次键入`dictionary`。

当然，如果你愿意输入

```
cp /usr/lib/dictionary dictionary
```

或者

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

结果都是一样的，它们做的都是同一件事情。

5.2.4 目录的操作

相对于文件的系统调用而言，各个系统中用于管理目录的系统调用差别更大。为了让读者对这些系统调用及其工作方式有一个印像，我们将给出一个例子（取自 UNIX）。

1. **Create**。创建一个目录。除了目录项“.”和“..”之外，该目录没有包含任何内容。目录项“.”和“..”是系统自动放在目录中的（有时是通过 `mkdir` 程序）。
2. **Delete**。删除一个目录。只有空目录可以被删除。一个只含有目录项“.”和“..”的目录都认为是空目录，这两个特殊的目录项通常是不能被删除的。
3. **Opendir**。目录的内容可被读取。例如，为了列出目录中的所有文件，应用程序必须先打开该目录，然后读取其中所有文件的文件名。与文件的打开和读取操作一样，在读一个目录之前，必须先打开它。
4. **Closedir**。当一个目录读完以后，应该把它关闭，以释放相应的内部表格空间。
5. **Readdir**。这个调用返回一个目录中的下一个目录项。以前，可以使用通常的 `read` 系统调用来读目录，但这种方法有一个缺点：程序员必须了解并处理目录的内部结构。相反，不管使用哪一种目录结构，`readdir` 总是以标准格式返回一个目录项。
6. **Rename**。在很多方面，目录和文件相似。文件可以换名，目录亦然。
7. **Link**。链接技术允许一个文件同时出现在多个目录中。这个系统调用指定一个业已存在的文件和一个路径名，并建立一条从文件到路径名的链接。这样，同一个文件可以出现在多个不同的目录中。这种类型的链接，在具体实现时是把文件的索引节点中的计数值（用来记录包含该文件的目录项个数）加 1，我们把它称为硬链接（hard link）。
8. **Unlink**。删除一个目录项。如果被解链的文件只出现在一个目录中（通常情形），那么该文件将被删除。如果它出现在多个目录中，则只删除指定的路径名，其他路径名依然保留下来。在 UNIX 中，删除文件的系统调用（前面已有论述）实际上就是 `unlink`。

以上列出了最主要的一些系统调用，还有其他一些调用，如目录的保护信息的管理等，这里不再赘述。

5.3 文件系统的实现

前面两小节是从用户的角度出发来讨论文件系统的，用户关心的是文件是如何命名的、可以进行哪些操作、目录树是什么样的以及类似的用户接口问题。下面我们将从系统实现者的角度来研究文件系统，实现者感兴趣的是文件和目录是如何存储的、磁盘空间是如何管理的以及如何使系统有效而可靠地工作等。下面我们将分别讨论这些问题，以及这些问题的各种解决方案。

5.3.1 文件系统的布局

文件系统通常保存在磁盘上。在第 2 章介绍 MINIX 3 引导程序的时候，我们曾经看到过基本的磁盘布局，下面再来简单地回顾一下。大多数磁盘可以分为若干个分区，每个分区上的文件系统是相互独立的。磁盘的扇区 0 称为主引导记录（Master Boot Record，MBR），它主要用来启动计算

机。在MBR的末尾有一个分区表，里面记录了每一个分区的起始地址和结束地址。在这些分区中，有一个是活动分区。在计算机启动后，BIOS会读入并执行MBR中的代码。MBR程序所做的一件事情，就是确定活动分区，并读入它的第一个磁盘块，称为引导块（boot block），然后执行它。而引导块中的代码就会把保存在该分区中的操作系统读取出来，装入内存运行。为了保持一致性，每一个分区都是以一个引导块开头，即使在该分区中并没有包含一个可引导的操作系统。而且，即使它现在没有，但也许在将来的某个时间就会有，因此，不管怎样，预留一个引导块总是一个不错的主意。

无论使用哪一种操作系统或硬件平台，也无论BIOS能启动多少个系统，以上的描述都应该是正确的。当然，在不同的操作系统中，使用的术语名称可能会不太一样。例如，主引导记录有的时候称为初始化程序装载器（Initial Program Loader，IPL）、卷引导代码（Volume Boot Code）或简称为**主引导**（masterboot）。有些操作系统不需要把一个分区设置为活动分区，它们给用户提供了一个菜单，可以对启动分区进行选择。而且一般来说会设置一个默认的选择和一定的时间期限，如果在规定的时间内用户没有做出选择，系统就会启动默认的分区。一旦BIOS装入了一个MBR或引导扇区，随后的动作可能不太一样。例如，在有的分区中，可能使用了不只一个块来存放引导代码。对于BIOS来说，你只能指望它帮你装入第一个磁盘块，但如果引导程序太长，一个块放不下，那么再由刚刚被装入的那个块负责，把其余的块装入内存。系统的实现者也可以提供一个自定义的MBR，但它必须和一个标准的分区表兼容，以便支持多种不同的操作系统。

在PC兼容的系统上，**主要分区**（primary partition）的个数最多不能超过4个，因为在主引导记录和第1个512字节的扇区末尾之间，仅能容纳4个数组元素，来存放分区的描述符。有些操作系统允许将分区表中的某一项作为一个**扩展分区**（extended partition），指向一个**逻辑分区**（logical partition）链表，这样就可以使系统具有任意多个额外的分区。不过，BIOS不能从逻辑分区来启动一个操作系统，因此，最初的启动必须是从主分区开始，由它来装入代码，管理各个逻辑分区。

针对扩展分区，MINIX 3使用了另一种解决方案，它允许在一个分区中，包含有一个子分区表（subpartition table）。这样做好处是，可以使用管理主分区表的代码来管理子分区表，因为它们的结构是一样的。子分区的可能用途是：针对根设备、交换区、系统二进制文件和用户文件，可以使用不同的子分区。这样一来，在一个子分区中发生的问题不会传播到另一个子分区，而且在安装一个新版本的操作系统时，可以只安装在其中的部分子分区中，而不必安装在所有的分区中。

不是所有的磁盘都需要分区，软盘的引导块通常从第一个扇区开始。BIOS读入磁盘的第一个扇区，然后查找一个魔数，标明它是有效的可执行代码。这样可以避免去执行一个未格式化或已损坏的磁盘。主引导记录和引导块使用的是相同的魔数，因而可执行代码可能是两者之一。另外，我们这里所说的不仅仅局限于电动磁盘设备，诸如数码相机和个人数字助理等具有非易失型（如flash）存储器的设备通常会使用部分的存储容量来模拟一个磁盘。

与引导块的启动不同，对于不同的文件系统，磁盘分区的布局具有很大的差别。一个类UNIX的文件系统可能会包含图5.8所示的一些内容项。第一项是**超级块**（superblock），它包含了关于文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存。

接下来的内容是空闲空间管理，主要是关于文件系统中的空闲物理块的管理信息。随后可能是索引节点，也就是一组数据结构，一个文件对应一个，描述了文件的所有属性信息和它在磁盘上的存储位置。再后面可能是根目录，即文件系统树的根节点。最后，在剩余的磁盘空间中，存放了所有其他的目录和文件。

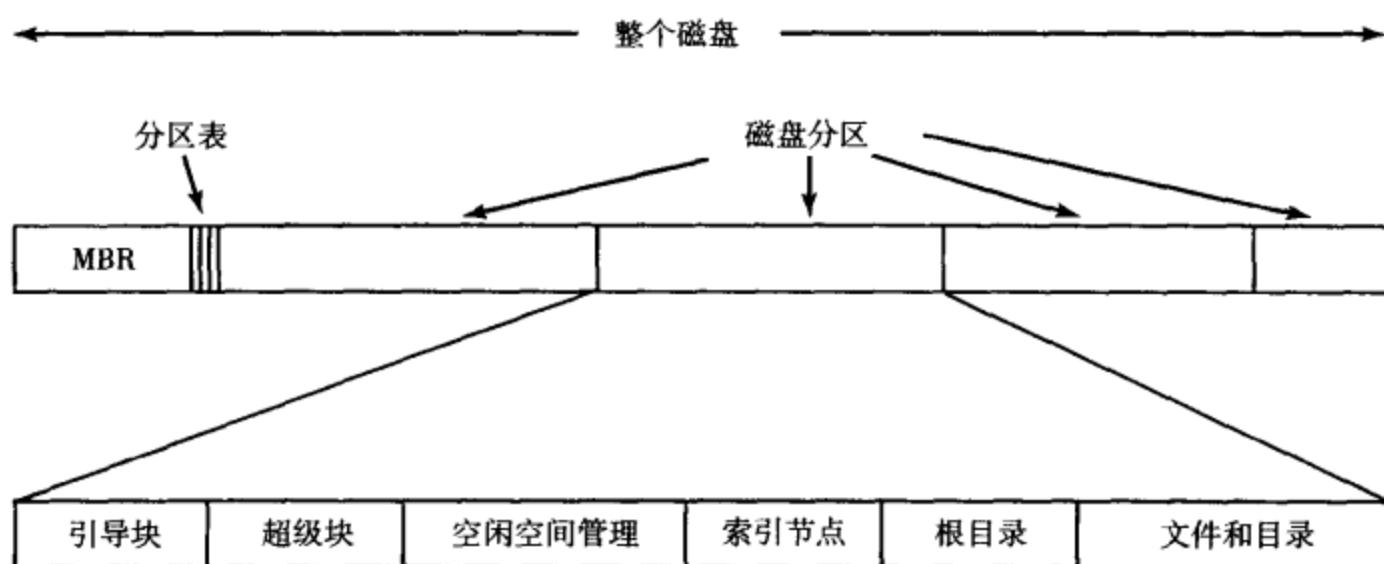


图 5.8 一种可能的文件系统布局

5.3.2 文件的实现

在实现文件的存储时，最重要的问题可能是如何来记录一个文件被存放在哪一些磁盘块中。不同的操作系统采用了不同的方法。在本小节，我们将讨论其中的一些方法。

连续分配

最简单的分配方案是把每个文件存放在连续的磁盘数据块中。例如，假设磁盘的数据块大小为 1 KB，那么对于一个 50 KB 的文件，就需要给它分配 50 个连续的磁盘块。这种分配方案有两大优点。第一是简单、易于实现。对于文件系统来说，它只要记住第一个数据块的磁盘地址和数据块的个数，就可以通过简单的加法来计算出文件的每一个数据块是存放在磁盘的什么位置。

第二，由于数据块是连续存放的，所以在访问文件的时候，只要将磁头定位到第一个数据块，就可以按照顺序一个接一个地读取，而不用再去移动磁头，或者等待相应的扇区旋转到磁头下方。这样一来，在一个操作中就可以把整个文件读出来，因而磁盘的访问速度非常快，能够达到它的最大带宽。所以说，连续分配的优点就是简单、易于实现，且性能很好。

当然，连续分配也有它的缺点。首先，随着磁盘上的文件的增加和删除，将会形成那些已经被占用的磁盘块与空闲的磁盘块之间相互交错的情形，这样，那些比较小的、没有办法再利用的若干个连续的空闲块，就成了外碎片。当然，为了解决这个问题，也可以采用存储紧缩技术，把所有的文件往一个方向移动，这样所有的空闲块就在另一个方向形成一个比较大的区域，但是这样做的代价是非常高昂的，因为磁盘的访问速度比较慢。其次，在连续分配方式下，文件的大小不能动态地增长。在创建一个新文件时，必须指定该文件的大小，这样系统才知道应该把多大的一块连闲空间分配给它。

正如我们在第 1 章所说的，在计算机科学领域，随着新技术的出现，历史总是在重复它自己。连续分配实际上主要用在早期的一些磁盘文件系统中，那时主要是看上了它的简单、高性能的特点（用户友好性在当时并不是考虑的重点）。后来，由于这种方法用起来比较麻烦，每次创建一个文件的时候，都要指定该文件的最终大小，因此，它就逐渐被人们放弃了。但是随着 CD-ROM、DVD 和其他一些一次性写入的光学存储介质的出现，这种连续分配的方案又重新活跃起来，并得到了广泛的应用。因为当我们在刻盘时，所有文件的大小都是已知的，而且以后也不会再改变。也就是说，连续分配的缺点都不存在了，而它的两个优点还保留着，这就是为什么它又开始流行的原因。因此，有意识地去研究一些比较老的、比较简单和清楚的系统是非常重要的，说不定在将来的系统中它们就能被用上。

链表分配

文件存储的第二种方法是为每个文件构造一条磁盘块链表，如图 5.9 所示。每个块的第一个字节做指针，指向下一个块，而块的其余部分则用来存放数据。

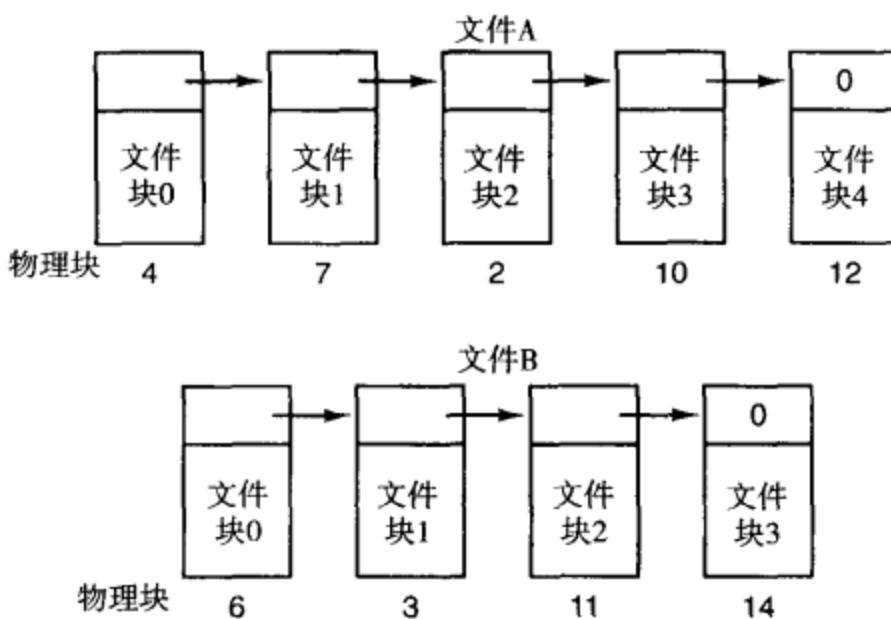


图 5.9 把文件存放在一条磁盘块链表中

与连续分配方案不同，在链表结构中，每一个磁盘块都能够被用上，不会出现外碎片的问题（在每个文件的最后一个块中，可能会有内碎片）。同样，对于文件系统来说，在目录项中，只要存放第一个块的磁盘地址即可，其余的文件块可以通过链表来依次访问。

在链表结构中，虽然文件的顺序访问是比较方便的，但如果要对文件进行随机访问，那么速度就会相当慢。例如，假设我们要去访问第 n 个文件块，那么操作系统必须从第一个块开始，把前面的 $n - 1$ 个文件块都读进来，一次读一个。显然，这样做的结果是极其慢的。

此外，由于指针要占去一些字节，因此每个磁盘块中用来存储数据的字节数就不再是 2 的整数次幂，虽然这个问题并不是太大，但它确实降低了系统的运行效率，因为大多数程序在访问磁盘时都以数据块为单位，每个块的大小一般是 2 的整数次幂。这样一来，当我们需要读入相同大小的文件数据时，由于指针要占用若干个字节，因此这些数据可能被分拆在两个物理块中，需要去访问两次磁盘。而且在读进来以后，还要把两部分数据合并起来。

带有文件分配表的链表结构

为了解决链表结构的问题，人们又对它进行了改进，也就是说，把每一个磁盘块中的链表指针抽取出来，单独组成一个表格，放在内存中。例如，图 5.9 的例子所对应的内存表格如图 5.10 所示。在这两个图中，有两个文件。文件 A 依次访问了磁盘块 4, 7, 2, 10 和 12，文件 B 依次访问了磁盘块 6, 3, 11 和 14。利用图 5.10 中的表格，我们可以从第 4 个块开始，顺着链表往下走，找到文件 A 的所有磁盘块。同样，从第 6 个块开始，顺着链表往下走，也能找出文件 B 的所有磁盘块。这两条链表都以一个特殊的标记 (-1) 来表示结束，因为 -1 不是一个有效的块号。我们把这样的一种表格称为文件分配表（File Allocation Table, FAT）。

采用这种方案后，整个块都能够用来存放数据，而且对文件的随机访问也容易得多。虽然我们仍然要对链表进行遍历搜索，找到文件块所在的磁盘地址，但由于整条链表都存放在内存中，因此速度很快。与链表分配方案一样，不管文件有多大，系统只要在目录项中存放一个整数（起始块号）即可，根据它就可以找到文件的所有块。

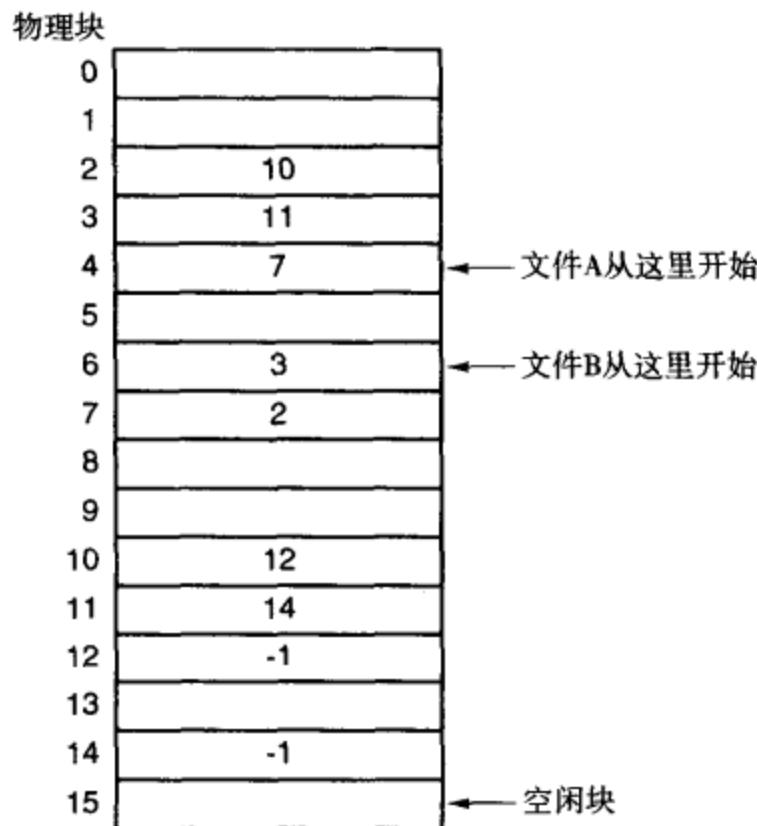


图 5.10 带有文件分配表的链表结构

这种方案的主要缺点是，整个FAT表都必须位于内存之中。假设有一个20 GB的磁盘，块的大小为1 KB，那么FAT表就需要2000万个表项，用来描述相应的2000万个磁盘块。每个表项最少需要3个字节，为了提高查找速度，最好是把它设为4个字节。这样一来，整个表格就需要占用60 MB或80 MB的内存。一种方案是把它放在虚拟页式存储器中，但它仍然要占用大量的虚拟内存和磁盘空间，而且还会增加更多的页面置换次数。MS-DOS和Windows 98使用的就是FAT文件系统，Windows的后续版本也都支持这种文件系统。

索引节点

实现文件块到物理块之间的映射的最后一种方法是给每个文件赋予一个数据结构，称为索引节点(index node)，或i节点，里面列出了文件的属性和各个数据块的磁盘地址，一个简单的例子如图5.11所示。在给定i节点后，就能够找到文件的所有数据块。与带有FAT表的链表结构相比，这种方法的最大优点是：只有当一个文件被打开时，才需要把它的i节点装入内存。如果每个i节点需要占用n个字节，且最多只能同时打开k个文件，那么用来存放文件i节点的内存空间最多不会超过kn个字节，因此，系统只需事先预留这么大的内存空间即可。

这段内存空间通常比FAT表所占用的内存空间要小得多，原因很简单。用来存放所有磁盘块的链表的表格，其大小与磁盘的大小成正比。如果磁盘有n个块，那么FAT表就有n项。如果磁盘容量增加，则FAT表也随之成线性增长。相反，在i节点方案中，需要的内存空间大小是与能同时打开的最大文件个数成正比的，它与磁盘的大小无关，不管磁盘的容量是1 GB、10 GB还是100 GB，对它来说都是一样的。

i节点方案的一个问题是：如果每个i节点能够存放的磁盘地址个数是有限的，那么如果文件太大了，超过了这个限制怎么办？一种解决方案是把最后一个磁盘地址移作他用，不是用来描述一个数据块，而是指向一个间接块(indirect block)，里面存放了更多的磁盘块地址。如果这还不够的话，还可以使用二级间接块(double indirect block)和三级间接块(triple indirect block)，如图5.11所示。

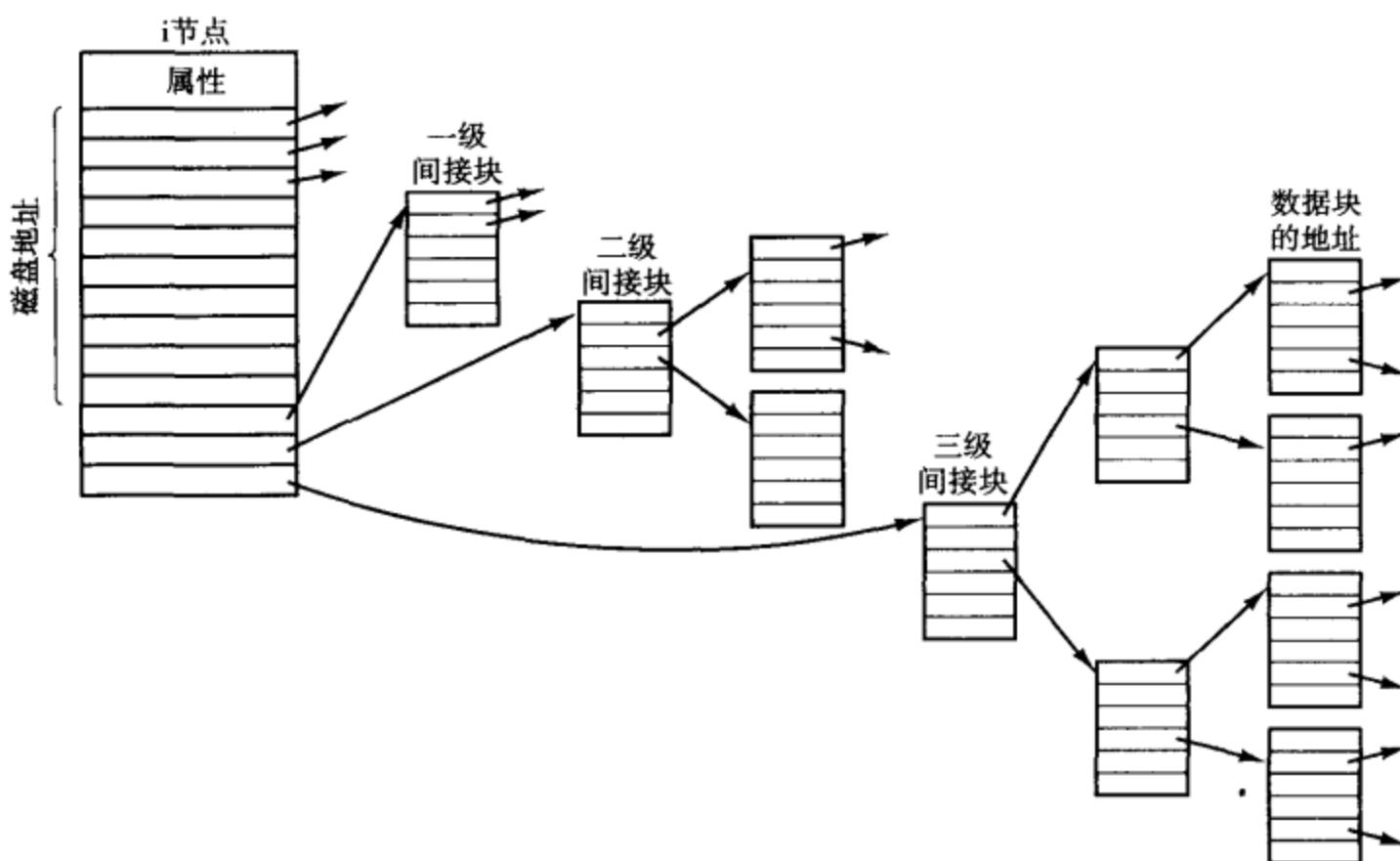


图 5.11 具有三级间接块的 i 节点

5.3.3 目录的实现

在访问一个文件之前，首先要打开这个文件。在打开一个文件时，操作系统会利用用户提供的路径名找到相应的目录项，而查找一个目录项就意味着首先要定位根目录。根目录可能是位于磁盘分区内的某个固定位置，或者它的起始位置可能是由其他的一些信息来决定的。例如，在一个经典的 UNIX 文件系统中，在超级块中包含有文件系统各个数据结构的大小，我们也可以从中找到 i 节点所在的位置。第一个 i 节点指向的是根目录，它是在 UNIX 文件系统创建时生成的。在 Windows XP 中，根据引导扇区中的信息（这些信息比一个扇区要大很多）可以找到主文件表（Master File Table, MFT），然后用它来定位文件系统的其他部分。

一旦找到根目录，就可以对目录树进行搜索，查找所需要的目录项。在目录项中，提供了查找磁盘块所需要的信息。在不同的系统中，这个信息是不同的，它可能是整个文件的磁盘地址（连续分配）、第一个块的块号（链表分配）或 i 节点号，不管是哪一种情形，目录系统的主要功能都是一样的，即把 ASCII 形式的文件名映射为查找文件数据所需要的信息。

与目录的实现密切相关的一个问题是在哪儿存放文件的属性信息。每一个文件系统都会维护一些文件属性，如文件的所有者、创建时间等，这些信息必须存放在某个地方。一种比较容易想到的方法就是把文件属性直接存放在目录项中。例如，最简单的做法是：一个目录由一组固定长度的目录项组成，一个文件一个，在每个目录项中，包含了文件名（固定长度）、一组文件属性以及一个或多个磁盘地址，描述了文件块的存储位置，就像我们在图 5.5(a)中所看到的那样。

对于使用 i 节点的系统，还存在另一种可能，即把文件属性存放在 i 节点中，如图 5.5(b)所示。在这种情形下，目录项就比较短，只有一个文件名和一个 i 节点号。

共享文件

在第 1 章，我们简单地提到了文件之间的链接（link），它使得同一个项目组的各个成员能够共享文件。图 5.12 再次显示了图 5.6(c)中的文件系统，不同之处在于现在有一个 C 的文件同时也出现在 B 的目录下。

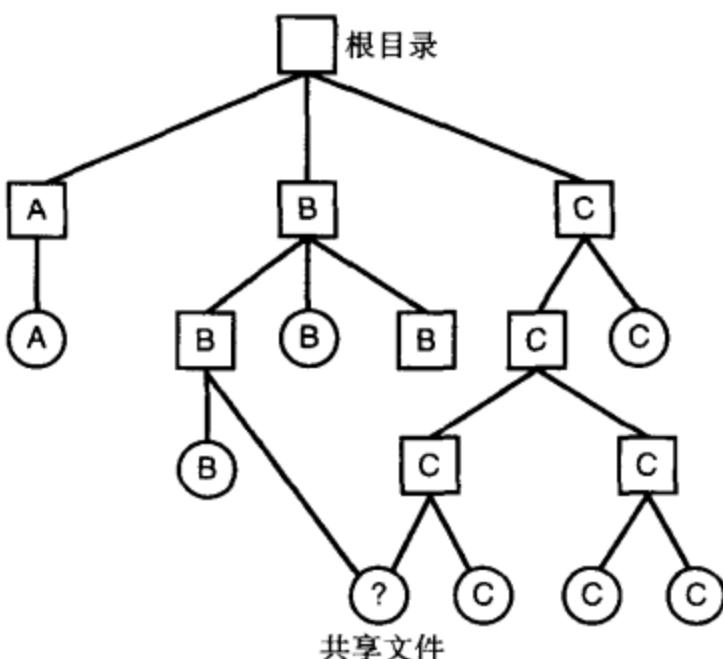


图 5.12 包含一个共享文件的文件系统

在 UNIX 系统中，由于使用了 i 节点来存放文件属性，因此文件的共享变得更加容易，可以有任意多个目录项指向同一个 i 节点。在 i 节点中包含有一个字段，每当有一个新的链接添加进来，就把它加 1；每当一个链接被删除，就减 1。只有当该计数值等于 0 时，文件的数据和 i 节点才会被删除。

这种类型的链接有时被称为硬链接 (hard link)。使用硬链接来共享文件并不永远是可行的，它的主要局限性在于：目录和 i 节点是一个文件系统（分区）的数据结构，因此，在一个文件系统中的目录不能指向另一个文件系统中的 i 节点。另外，一个文件只能有一个所有者和一组权限。如果某个共享文件的所有者删除了它自己的目录项，那么对于另一个用户来说，在他的目录下可能会有一个很难“摆脱”的文件，由于权限不够，他无法将该文件删除。

另外一种共享文件的方法是创建一种新的文件，它的数据是另一个文件的路径名。这种链接可以跨越不同的文件系统，事实上，如果有一种方法能够把路径名包含在网络地址中，那么这种链接甚至能访问不同计算机上的文件。在类 UNIX 系统中，这种链接被称为符号链接 (symbolic link)，在 Windows 系统中，它被称为快捷方式 (shortcut)，而在苹果的 Mac OS 中，它被称为别名 (alias)。符号链接可以用于把属性存放在目录项中的系统。不过仔细一想，如果在多个目录项中都存放有文件的属性信息，那么会很难同步，因为对文件所做的任何修改都会影响到该文件的所有目录项。因此在符号链接方式下，对于额外的目录项而言，它们并不会包含所指向的文件的属性信息。符号链接的一个缺点是：当一个文件被删除时（或者只是改了一个名字），相应的链接就会变得无效。

Windows 98 中的目录

Windows 95 最初版本中的文件系统和 MS-DOS 的文件系统是完全相同的，但它的第二个版本增加了对长文件名和大文件的支持，我们把它称为 Windows 98 文件系统，虽然它也用在一些 Windows 95 系统上。在 Windows 98 中，存在着两种类型的目录项，我们把第一种称为基本目录项，如图 5.13 所示。

基本目录项具有 Windows 早期版本中的目录项的所有信息，而且增加了一些新的信息。从 NT 字段开始的 10 个字节是新增加的内容，这块区域以前未使用。其中最重要的升级就是把起始数据块的地址从 16 位增加到 32 位，这就使系统能够访问的数据块的个数从 2^{16} 增加到了 2^{32} 。

上述结构提供的仅仅是从 MS-DOS（和 CP/M）中继承下来的老式的 8 + 3 字符文件名，那么长文件名如何处理呢？如何在保持与老系统兼容的情形下来支持长文件名呢？答案就是使用另外的

目录项。图 5.14 显示了另一种形式的目录项，它能包含最多 13 个字符的长文件名。对于使用了长文件名的文件，系统会自动生成一个缩短的文件名，并把它放在图 5.13 所示的基本目录项的“文件名”和“扩展名”字段中。如果文件名的长度不只 13 个字符，那么可以根据需要，增加更多个如图 5.14 所示的目录项，这些目录项被放在基本目录项的前面，按照反向顺序排列。在每个长文件名目录项的属性字段，包含了一个特殊值 0x0F，对于老的（MS-DOS 和 Windows 95）文件系统来说，这个值是不可能出现的，所以如果一个老的系统（如软盘上系统）来访问这些目录项，就会把它们都忽略掉。“序列”字段中的数据位用于告诉系统，哪一个目录项是最后一项。

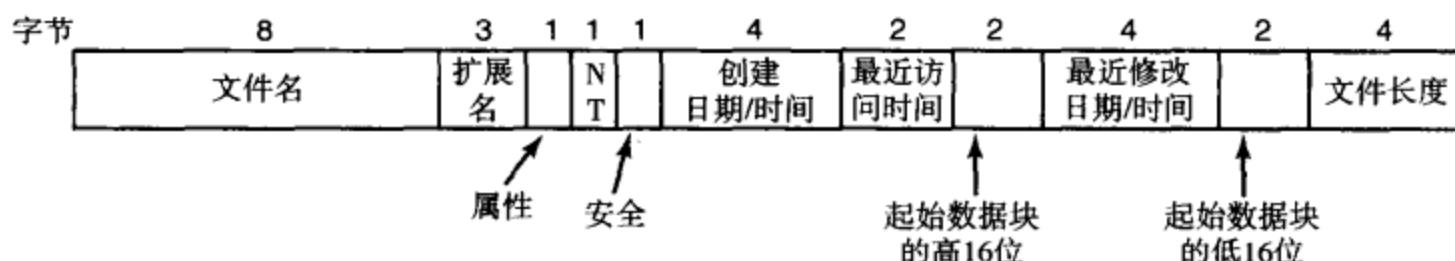


图 5.13 Windows 98 的一个基本目录项

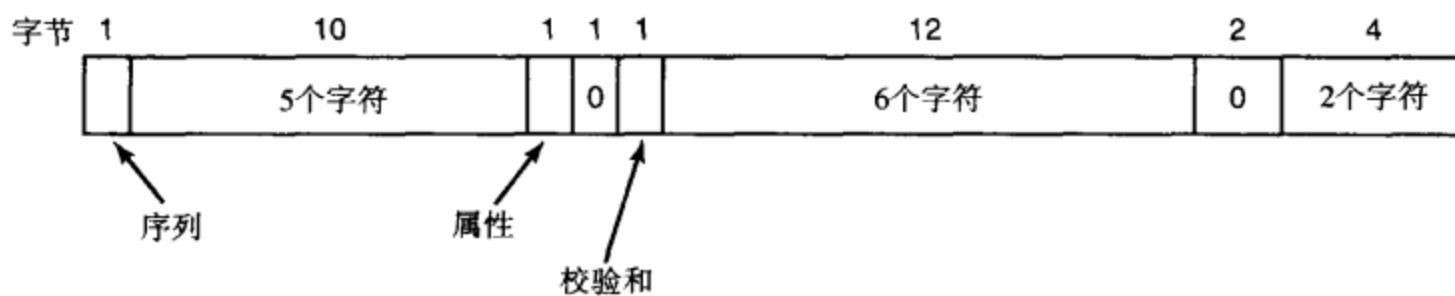


图 5.14 Windows 98 中的一个长文件名目录项

如果这看起来比较复杂，那么的确如此。因为它一方面要提供后向的兼容能力，使早期的系统能继续工作；另一方面又要为新的系统提供一些新的特性，这样就很容易把事情弄乱。一个纯化论者可能会追求比较单纯的形式，不希望把系统搞得这么麻烦。不过，如果你是一个纯化论者，那么你可能就卖不了太多的新版本的操作系统，也就没法发家致富。

UNIX 中的目录

传统的 UNIX 目录结构非常简单，如图 5.15 所示，每个目录项只包含一个文件名及其相应的 i 节点号。有关文件的类型、长度、时间、所有者和磁盘块等所有的信息都存放在 i 节点中。有些 UNIX 系统可能会有不同的布局，但无论如何，目录项中的内容最终还是差不多的，都是一个 ASCII 字符串和一个 i 节点号。

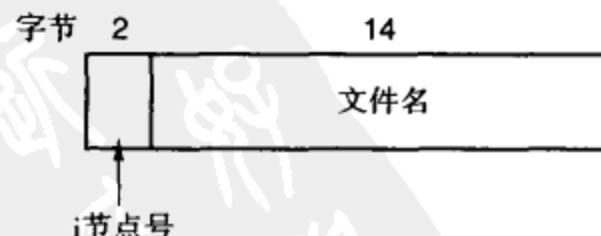


图 5.15 UNIX Version 7 的目录项

当一个文件被打开时，文件系统必须根据用户给定的文件名，找到相应的磁盘块。例如，我们来看一下路径名 /usr/ast/mbox 是如何来查找的。我们将使用 UNIX 作为例子，但对于所有的层次结构目录系统而言，这个算法都是差不多的。首先系统会找到根目录。在 UNIX 中，所有文件的 i 节点构成了一个简单的数组，系统可以利用超级块中的信息来定位该数组，而数组中的第一个元素就是根目录的 i 节点。

接下来,文件系统会去查找路径名的第一个部分 *usr*,它将在根目录中查找文件*/usr/*所对应的i节点号。在知道i节点号之后,再去定位i节点就很简单了,因为每个i节点都存放在磁盘的固定位置(相对于第一个i节点),因此可以使用i节点号为索引,去访问相应的数组元素。通过这个i节点,系统可以找到目录*/usr/*,并接着查找路径名的下一部分*ast*。当找到*ast*的目录项后,就得到了目录*/usr/ast/*的i节点,然后在该目录中查找文件*mbox*。接下来,文件*mbox*的i节点被读入内存,并保存在内存中,直至该文件被关闭。这个查找过程参见图5.16。

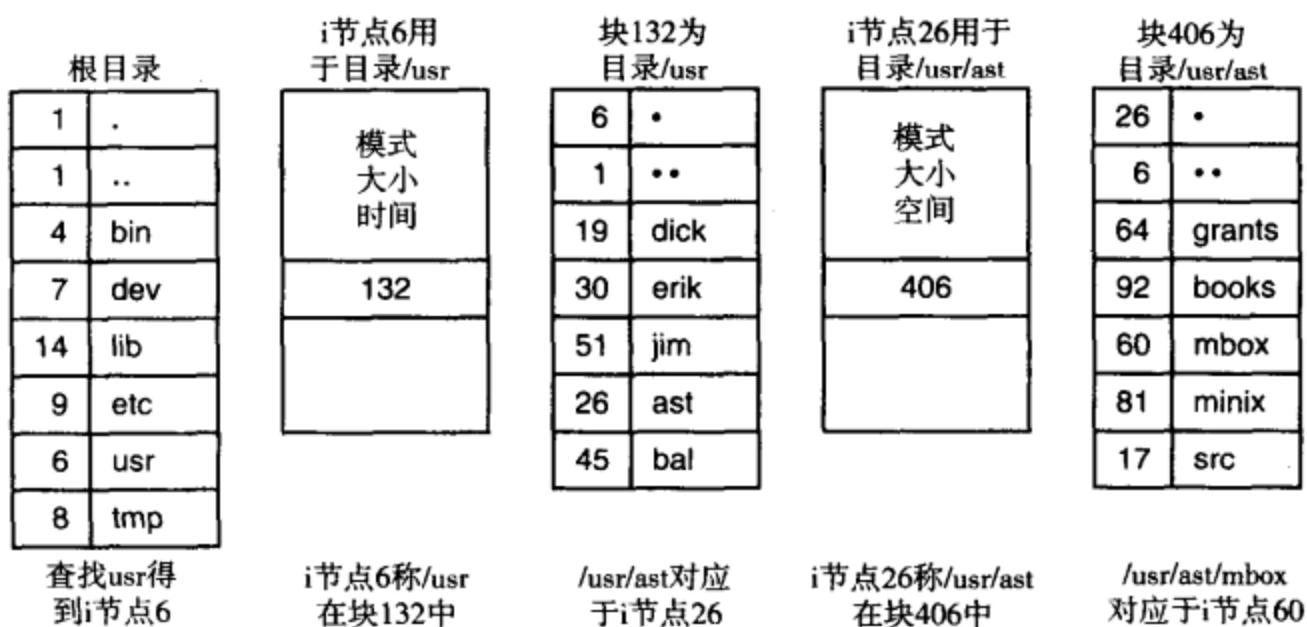


图5.16 查找*/usr/ast/mbox*的过程

相对路径名的处理方式与绝对路径名是差不多的,只不过它的起始位置是当前工作目录,而不是系统的根目录。在每一个目录下,都有两种特殊的目录项:“.”和“..”,它们是在该目录被创建时由系统放在那里的。目录项“.”具有当前工作目录的i节点号,而目录项“..”则具有父目录的i节点号。因此,当一个函数在查找*../dick/prog.c*时,首先在当前工作目录下查找“..”,找到父目录的i节点号。然后在该目录下搜索*dick*文件。在处理这些名字时,无须再使用其他特殊的机制。就目录系统而言,它们只不过是通常的ASCII字符串,和其他名字是完全相同的。

NTFS中的目录

微软的NTFS(New Technology File System)是默认的文件系统。由于时间的关系,在这里我们不打算对NTFS做详细的讨论,只是简单地来看一下NTFS处理的是哪些问题,使用了哪些解决方案。

一个问题是长文件名和路径名。NTFS支持长文件名(最多255个字符)和路径名(最多32 767个字符)。但由于Windows的老版本不能访问NTFS文件系统,因此就没有反向兼容的问题,不需要去设计非常复杂的目录结构,而且文件名字段是变长的。当然,系统也为文件提供了一个8+3类型的名字,这样,一个老的系统可以通过网络来访问NTFS文件。

通过使用Unicode编码,NTFS支持多种不同的字符集。Unicode用16位来表示一个字符,足以描述符号集非常大的语言。但使用多种语言的问题不仅仅是不同字符集的表示。在拉丁语系中,有些语言具有一些细微之处。例如,在西班牙语中,某些字符的组合在排序时将作为单个字符,以“ch”或“ll”开头的单词在排序列表中应该排在以“cz”或“lz”开头的单词后面。大小写的转换更为复杂。如果在默认设置下,文件名是大小写有关的,那么就需要进行大小写无关的搜索。对于拉丁语言,很容易做到这一点。一般来说,如果当前只使用一种语言,那么用户可能会知道相应的规则。但Unicode支持多种语言的混合:在一个国际组织中,希腊语、俄语和日语文件名可能会

出现在同一个目录下。NTFS的做法是为每个文件设定一个属性，以便定义在该语言中文件名的大小写规范。

增加更多的属性是NTFS解决很多问题的办法。在UNIX中，一个文件是一个字节序列。而在NTFS中，一个文件是一组属性集合，每个属性是一个字节流。基本的NTFS数据结构是主文件表（Master File Table，MFT），它提供了16个属性，每种属性的长度可达1KB。如果这还不够，可以在属性中指向另外一个文件，里面存放了额外的属性值。这种属性称为非常驻属性（nonresident attribute）。MFT本身也是一个文件，它为文件系统中的每一个文件和目录设置了一个目录项，这样它可能会增长得非常大。当NTFS文件系统被创建时，它会在磁盘分区上预留12.5%的存储空间，留给MFT。这样，MFT在最初的增长过程中就不会被打散在不同的地方。当MFT超出了最初的预留空间后，系统又会把它预留一块很大的存储空间。

那么NTFS中的数据如何处理呢？数据只不过是另一种属性。实际上，一个NTFS文件可以有多个数据流。最初引入这个特性，是为了使Windows服务器能够服务于苹果的Macintosh客户。在最初的Macintosh操作系统中（Mac OS 9），所有的文件都有两个数据流，称为资源流和数据流。多个数据流有其他的一些用途，例如，在一个很大的图片文件中，可以用另一个数据流来存放一幅比较小的缩略图。一个数据流最多可以包含 2^{64} 个字节，另一方面，如果文件非常小，只有几百个字节，那么NTFS可以直接把它存放在属性中。这称为立即文件（immediate file）（Mullender and Tanenbaum, 1984）。

我们这里只讨论了NTFS处理一些比较老或比较简单的文件系统所没有考虑的问题的方式。此外，NTFS还提供了其他一些特性，如复杂的保护系统、加密和数据压缩等。这些特性及它们的实现涉及较多的内容，这里不再赘述。关于NTFS的详细内容，请参阅Tanenbaum（2001）或在互联网上搜索相关的信息。

5.3.4 磁盘空间管理

文件通常存放在磁盘上，因此磁盘空间的管理是系统设计者要考虑的一个主要问题。在存储一个长度为 n 个字节的文件时，可以有两种策略：分配 n 个字节的连续磁盘空间，或者是把文件划分成许多块，然后把它们存放在不同的（可能是不连续的）磁盘块中。在内存管理系统中，也有类似的问题，即到底是选择段式还是页式存储管理。

如前所述，按连续字节序列来存储文件有一个明显的问题：当文件扩大时，如果空闲空间不够，可能需要把它移动到磁盘上的另一个位置。在段式存储管理中也有类似的问题，只不过它的移动是在内存中进行的，不需要访问磁盘，因此它的速度要快得多。基于这个原因，几乎所有的文件系统都选择了第二种方案，即把文件分割成固定大小的块来存储，而且各个块之间不必相邻。

块的大小

一旦决定把文件按固定大小的块来存储，问题马上就来了：块大小应为多少呢？根据磁盘的组织方式，扇区、磁道和柱面显然都可以作为分配单位。在页式系统中，页面的大小也是可选项之一。不过，如果分配单位过大，比如以柱面为单位，则意味着每一个文件都要占用整数个柱面，甚至连1个字节的文件也要占用一个柱面。

另一方面，如果分配单位过小，则意味着一个文件将包含许多块。在访问每个块时，通常都需要一次磁头定位和旋转延迟，因此，如果在一个文件中包含有很多个小的块，那么文件的访问速度将会很慢。

我们来看一个例子。假设在一个磁盘上，每条磁道有 131 072 个字节，旋转时间为 8.33 ms，平均磁头定位时间为 10 ms。那么访问一个长度为 k 个字节的数据块所需的时间为磁头定位时间 + 旋转延迟时间 + 数据传输时间：

$$10 + 4.165 + (k/131\,072) \times 8.33$$

图 5.17 中的实线表示该磁盘的数据访问速率与块大小之间的关系。

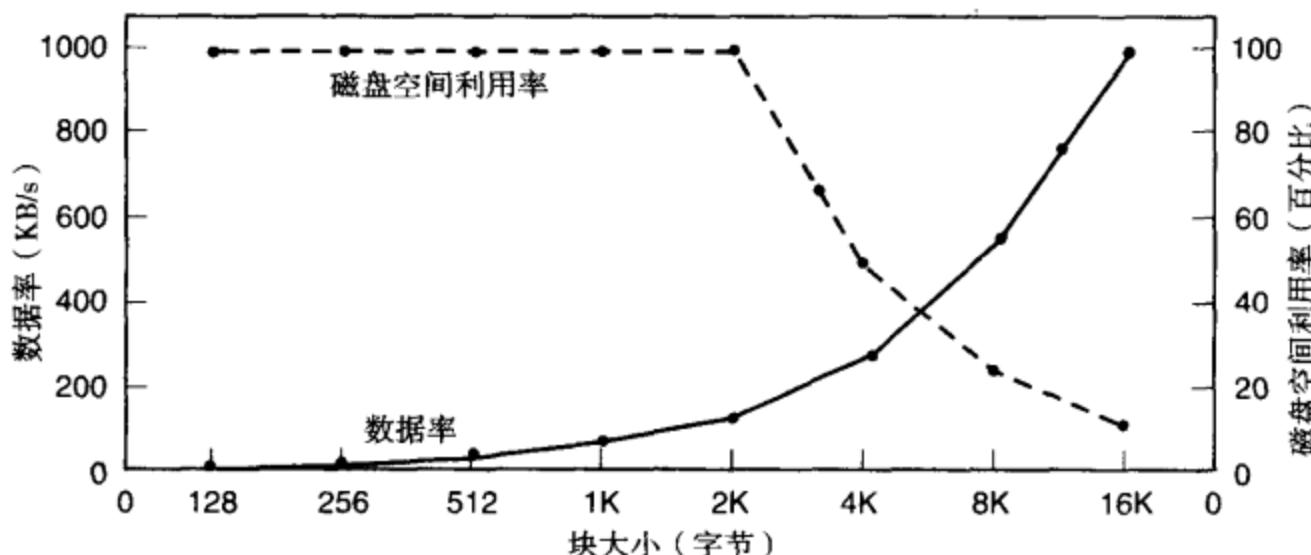


图 5.17 实线（左边标度）给出磁盘数据率，虚线（右边标度）给出磁盘空间效率，所有文件均为 2 KB

为了计算空间效率，我们需要估计一下文件的平均大小。一项早期的研究表明，UNIX 环境下的平均文件大小为 1 KB (Mullender and Tanenbaum, 1984)。2005 年在笔者 (AST) 所在的系做了一次测量，涉及到 1000 个用户和 100 万个 UNIX 磁盘文件，最后得到的中间数是 2475 个字节，也就是说，一半的文件比 2475 字节小，一半的文件比 2475 字节大。而且我们认为采用中间值来作为度量标准比平均值要好，因为很少数量的文件就能极大地影响到平均值，而中间值则不会。例如，几个 100 MB 的硬件手册或视频文件就能使平均值发生很大的偏移，但它们对中间值的影响并不大。

另一项实验研究了 Windows NT 环境下的文件使用情况，Vogels (1999) 统计了 Cornell 大学的文件长度，他发现，NT 文件的使用比 UNIX 文件要复杂得多。他写道：

如果我们在 notepad 文本编辑器中敲入一些字符，然后把它们存放在一个文件中，那么这将触发 26 个系统调用，包括 3 次失败的文件打开操作、1 次文件覆盖和 4 次额外的打开和关闭操作。

不过，他观察到的文件的中间值大小是：只读文件为 1 KB，只写文件为 2.3 KB，既读又写文件为 4.2 KB。考虑到 Cornell 有大量的大规模科学计算以及测量技术上的差异（静态与动态），这个结果与 UNIX 情形下的 2 KB 的结果是一致的。

为了简便起见，我们假定所有的文件都是 2 KB，这就得到了图 5.17 中的磁盘空间效率曲线（图中的虚线）。

这两条曲线可以这样来理解：一个数据块的访问时间完全由磁头定位时间和旋转延迟时间来决定，总共需要花费 14 ms。这样一来，数据块包含的数据越多，性能就越好。因此数据率是随着块大小的增加而增大的（直到数据的传输时间过长，占用了主要的时间）。如果块的大小比较小且是 2 的整数次幂，而文件的大小是 2 KB，那么就不会有磁盘空间的浪费。但如果文件的大小是 2 KB 而块的大小是 4 KB 甚至更大，那么就会浪费一些磁盘空间。实际上，很少有文件正好是磁盘块大小的整数倍，因此在一个文件的最后一个块中总是会浪费一些磁盘空间。

从这两条曲线可以看出，性能和空间利用率是相互冲突的。如果数据块比较小，那么性能较差，但磁盘空间利用率较高；反之，如果数据块比较大，则性能较好，但浪费的空间较多。因此我们需要一个折中的大小。从当前的数据来看，4 KB似乎是一个不错的选择，但有些操作系统在很久以前就做出了选择，那时的磁盘参数和文件大小跟现在是不同的。对于UNIX来说，通常使用的是1 KB。对于MS-DOS来说，块大小可以是512字节到32 KB之间的某一个2的整数次幂，这是由磁盘大小和其他一些因素来决定的（例如，一个磁盘分区上的最大块数是 2^{16} ，这样，如果磁盘容量很大，那么块的大小就不得不变大）。

空闲块管理

在选定了块大小之后，接下来的问题就是如何来管理空闲块。如图5.18所示，有两种方法得到了广泛应用。第一种方法使用了一条链表，每个链表结点是一个磁盘块，里面存放了尽可能多的空闲磁盘块号。假设块的大小为1 KB，磁盘块号用32位来表示，那么每个链表结点最多能存放255个空闲块号（另外还要预留4个字节作为指针，指向下一个结点）。在这种方式下，一个256 GB的磁盘最多需要1 052 689个块的空闲链表来存放所有的 2^{28} 个磁盘块的块号。

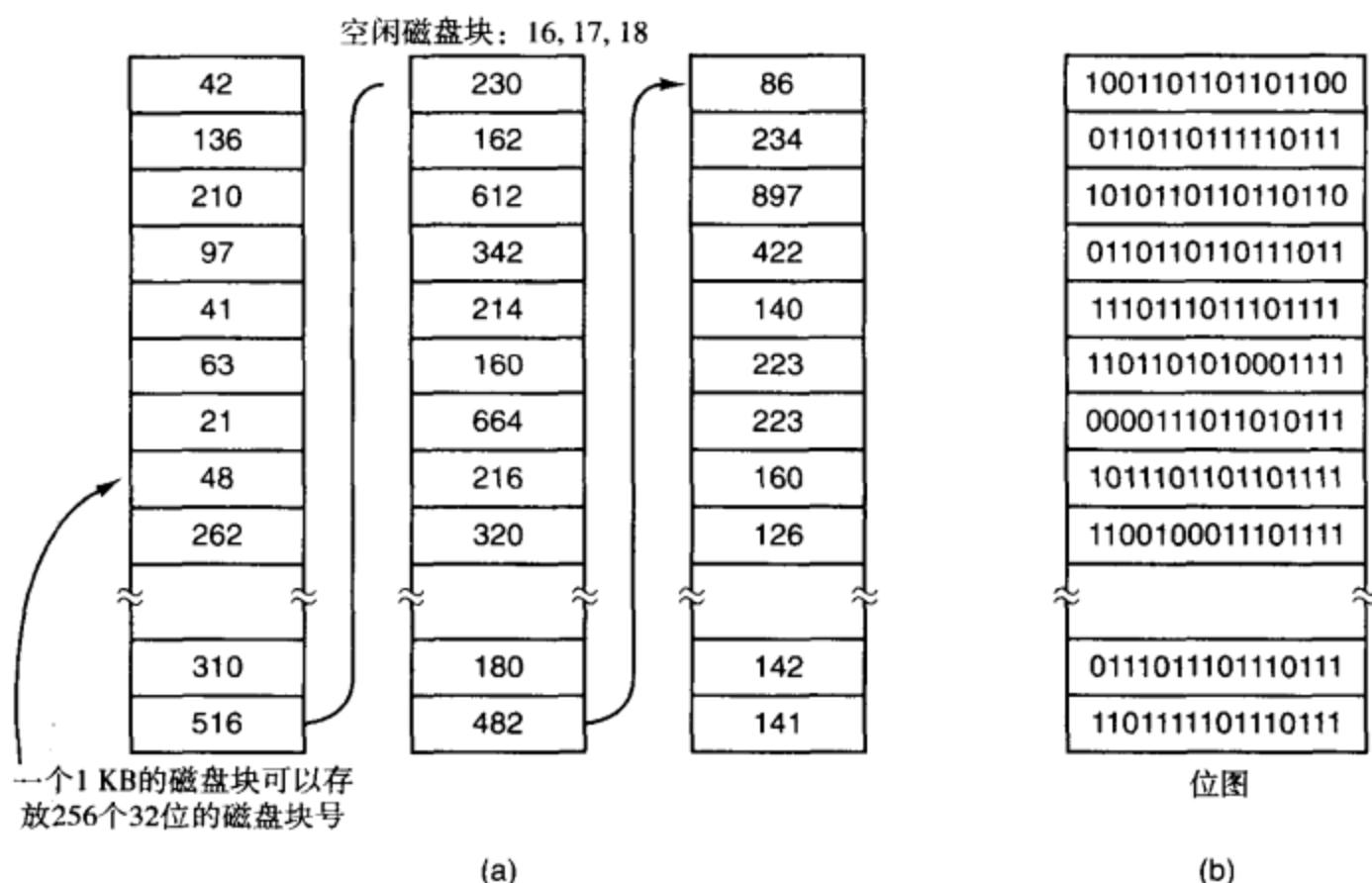


图5.18 (a)将空闲块表存放在一条链表中；(b)位图

第二种空闲空间管理技术是位图。如果一个磁盘有 n 个块，那么就需要 n 个位的位图来描述。在位图中，空闲块用1来表示，已分配的块用0来表示（或者相反）。一个256 GB的磁盘有 2^{28} 个1 KB的块，因此需要一张 2^{28} 个数据位的位图，该位图需要占用32 768个磁盘块。显然，与链表法相比，位图法需要占用的磁盘空间比较少，因为每个块只需要1个位来描述，而在链表法中，一个块需要用32位来描述。只有当磁盘几乎全满（即没有几个空闲块）的时候，链表法所需要的空间才比位图法要少。另一方面，如果空闲块很多，那么完全可以借用其中的一些来作为链表结点，而且不会减少磁盘空间的容量。

如果采用空闲链表法，那么在内存中只要保存一个链表结点即可。当一个文件被创建时，所需要的磁盘块就从这个结点中去取。如果该结点内的空闲块号都已用完，就从磁盘读入一个新的链表

结点。类似地，当一个文件被删除后，它的磁盘块就被释放并被添加到内存中的链表结点中，如果该结点装满了，就把它写回到磁盘。

5.3.5 文件系统的可靠性

比起计算机的损坏，文件系统的破坏往往要糟糕得多。如果由于火灾、闪电电流或者一杯咖啡泼在键盘上而弄坏了计算机，确实让人伤脑筋，而且又要花上一笔钱，但一般来说，修复起来还是比较方便的。求助于分销商，便宜的个人计算机在短短几个小时之内就可以修复（当然，如果在大学里面情况就不一样了，发出购买定单需征得3个委员会的同意，并盖5个公章，总共要花90天的时间）。

不管是由于硬件或软件的故障，或者是老鼠咬坏了软盘，如果计算机的文件系统被破坏了，那么要想恢复其中的信息将是一件既困难又费时的事情，有些时候，这根本是不可能的。对于那些程序、文档、客户文件、税收记录、数据库、市场计划或者是其他数据丢失的用户来说，这不啻为一次大的灾难。虽然文件系统无法防止设备和存储媒质的物理损坏，但它至少应能保护信息。在本小节，我们将讨论与文件系统保护有关的一些问题。

在出厂时，软盘通常是完好无损的，但在使用过程中，却可能会出现坏块。有人认为与以前软盘被普遍使用的时期相比，现在的软盘更容易出现故障。原因在于，随着网络和大容量可移动存储设备（如可读写的CD）的出现，软盘已经很少有用武之地了。当计算机在工作时，风扇会把空气中的灰尘吸入软盘驱动器，这样，如果一个驱动器很长时间没有使用，就会积累非常多的灰尘，此时，如果用户插入一张软盘，就会把盘弄坏。而如果是经常使用的驱动器，一般不会弄坏软盘。

硬盘常常一开始就有坏块，因为如果要把一块硬盘做得完美无缺，那成本实在是太高了。在第3章我们曾经谈到，硬盘上的坏块通常是由控制器来处理的，它会用一些专门的备用扇区来替代坏扇区。在这些磁盘上，每一条磁道都会有意地多出一个扇区，这样，如果出现了一个坏扇区，就可以把它跳过去。在每个柱面上，也会留有一些备用的扇区，这样，如果控制器发现某个扇区工作不太正常，需要重试多次才能完成读写操作，就会自动进行扇区重映射，把它映射到某个备用扇区。这样，对于用户来说，往往觉察不到坏块的存在以及对它们所进行的管理。不过话又说回来，当一个现代IDE或SCSI硬盘失效时，后果往往是很可怕的，因为此时所有的备用块都已用光。SCSI硬盘在重映射一个块时，会给出“出错恢复”信息。如果驱动程序发现该错误信息，将会在屏幕上打印一条消息。这样，如果用户看到这条消息经常出现，就知道他应该去购买一块新硬盘了。

此外，对于坏块问题，还可以用软件的方法加以解决，这种方法适合于老式磁盘。它要求用户或文件系统精心构造一个文件，里面存放了所有的坏块信息。这种技术能将坏块从空闲链表中删除，使其不会出现在数据文件之中。只要不对坏块进行读写操作，文件系统就不会出现任何问题。当然，在做磁盘备份时，需要注意不要去读取这个文件，也不要对它进行备份。

备份

很多人认为不值得花时间和精力去做文件备份，直到有一天，他们的磁盘突然坏掉了，这时他们才会如梦初醒。相比之下，公司通常会充分意识到他们的数据的价值，因此每天至少会进行一次备份，一般是备份在磁带上。现代的磁带能够容纳成百上千个GB，价格也非常便宜。不过，做数据备份并不像听起来那么容易，下面我们将讨论其中的一些相关问题。

磁带备份通常用于解决两个问题：

1. 从灾难中恢复
2. 从鲁莽操作中恢复。

第一个问题指的是当发生磁盘崩溃、火灾、水灾或其他自然灾害时，如何让计算机重新运行起来。实际上，这些事情并不会经常发生，这也是为什么人们不愿意去做备份的原因。基于同样的原因，这些人可能也不会为他们的房子购买火灾保险。

第二个问题指的是人们经常会不小心删掉有用的文件。这个问题出现的频率比较高，所以在 Windows 系统中，当一个文件被“删除”时，它实际上并未被删除，而是被移动到一个特殊的回收站（recycle bin）目录中。这样，如果将来需要的话，就很容易恢复。数据备份把这个原理进一步推广，用户可以从磁带中恢复几天前、甚至几周前被删除的文件。

做一次备份需要很长的时间，也占用了很大的存储空间，因此如何方便、有效地进行备份是十分重要的。这些考虑引发了以下一些问题。首先，是应该备份整个文件系统还是只备份其中的一部分？在许多系统中，可执行（二进制）程序被存放在文件系统树的有限几个位置，对于这些文件来说是无须备份的，因为我们可以用厂商提供的 CD-ROM 来重新安装。同样，在许多系统中有一个目录来存放临时文件，这些文件也没有必要备份。在 UNIX 中，所有的设备文件（I/O 设备）被保存在 /dev/ 目录下，这个目录的备份不仅是没有必要的，而且是非常危险的。如果备份程序试图完成该目录下的每一个文件的读取操作，那么它就可能被永远阻塞起来。

其次，没有必要去备份自上次备份以来未被修改过的文件，这就引发了增量转储（incremental dump）的思想。最简单的增量转储形式是：定期地进行一次完整的转储（备份），比如每周一次或每月一次。然后，在每天的备份工作中，只备份那些自上次完整备份以后被修改过的文件，或者更好的做法是只备份今天被修改过的文件。这种备份方法虽然减少了备份的时间，但却使得文件的恢复变得更复杂：首先要恢复最近的一次完整备份，然后根据增量转储的顺序，一次次地恢复，先处理老的备份，再处理近期的备份。因此，为了使恢复更加简单，往往需要更复杂的增量备份方案。

第三，由于备份涉及到大量的数据，因此，在把数据写入磁带之前，可能需要对它们进行压缩。然而，虽然目前有很多的压缩算法，但如果在备份磁带上有一个坏块，就会使解压缩算法失效，从而使整个文件甚至是整个磁带无法访问。因此，在对备份数据流进行压缩之前，需要深思熟虑。

第四，对于一个处于活动状态的文件系统很难进行备份。如果在备份期间，文件和目录不断地被添加、删除和修改，那么就会使备份的源文件和目标文件不一致。但如果让系统脱机来进行备份，由于做一次备份可能需要几个小时，所以一个夜晚的大部分时间可能都将用在备份上，而在很多时候，这种情形是无法接受的。基于这个原因，人们设计出一些算法，通过复制一些关键的数据结构来构造文件系统当前状态的一个快照，并要求将来对文件和目录的修改必须先缓存起来（Hutchinson et al., 1999）。这样，整个文件系统只是在建立快照期间被暂时冻结，而真正的备份可以随后慢慢地进行。

最后一点，做备份会在一个组织内引入许多非技术的问题。对于世界上最好的在线安全系统来说，如果它的系统管理员把所有的备份磁带随意地放在办公室里，而且当他出门去取打印结果的时候，把办公室的大门敞开着、无人看守，那么这样的一个“安全”系统根本就是不安全的。如果有一名商业间谍偷偷溜了进来，口袋里揣着一盒微型磁带，得意洋洋地东看西瞅，那么所谓的“安全”就无影无踪了。同样，如果来了一场大火，不仅把计算机烧掉了，而且把所有的备份磁带都烧掉了，那么平日里辛辛苦苦做备份的努力就付诸东流了。所以说，备份磁带不能存放在同一个办公地点，但这又会带来更多的安全风险。关于这些实际的管理问题，请参见 Nemeth et al. (2001)。下面我们只讨论与文件系统备份有关的技术问题。

有两种策略可以用来把磁盘的内容备份到磁带上：物理转储和逻辑转储。物理转储（physical dump）从磁盘的第 0 个块开始，按照顺序把所有的磁盘块都写入输出磁带。这种程序非常简单，以

至于可以做到百分之百正确，不含任何错误，而对于其他一些工具软件来说，一般不敢打这样的保票。

不过，这里需要对物理转储做一些评论。首先，备份那些未被使用的磁盘块是没有意义的。如果转储程序能够访问管理空闲块的数据结构，那么它可以跳过那些空闲块。但如果这样的话，又有一个问题，即磁带上的第 k 个块不再对应于磁盘上的第 k 个块。

其次是坏块的问题。如果磁盘控制器能够把所有的坏块进行重映射，并把映射的实现细节隐藏起来，操作系统不知道（如同 5.4.4 小节所述），那么物理转储能正常工作。另一方面，如果这些坏块对操作系统是可见的，操作系统用一个或多个“坏块文件”或位图来管理它们，那么对于转储程序来说，它必须获得这些信息，这样才不会去试图访问一个坏块，从而陷入无穷尽的磁盘访问错误中。

物理转储的主要优点是简单、速度快（一般来说，它能达到磁盘访问速度的最高值）。它的主要缺点是不能跳过某些选定的目录，不能增量转储，也不能根据需要恢复单个文件。基于这些原因，很多系统采用了逻辑转储的方式。

逻辑转储（logical dump）从一个或多个指定的目录开始，对于该目录下的每一个文件和目录，如果它自某个给定的基点日期（如增量转储时的最后一次备份时间或完整转储时的系统安装时间）后被修改过，那么就对它进行备份。因此，在逻辑转储方式下，在磁带上会得到一系列精心标识的文件和目录，这样，就能够根据用户的请求来恢复其中的某个特定的文件或目录。

为了能够正确地恢复一个文件，在备份磁带中，必须存储所有用于重建该文件路径名的信息。因此，逻辑转储的第一步就是对目录树进行分析。显然，我们需要保存所有被修改过的文件或目录。不仅如此，从根目录到被修改文件或目录的路径上的所有目录，不管它们是否曾经被修改，都要保存起来。这意味着不仅要保存目录的内容（即文件名和指向 i 节点的指针），还要保存目录的所有属性，这样在恢复文件的时候，才能带有原来的访问权限。目录和它们的属性首先被写入到磁带，然后是被修改的文件（也带着它们的属性）。这样就能够把被转储的文件和目录恢复到另一台计算机的文件系统上。通过这种方式，能够实现在不同的计算机之间传送整个文件系统。

把修改过的文件的上层目录都保存起来，这样做的另一个原因为了能增量地恢复一个文件（如从意外删除中恢复过来）。假设在周日的晚上做了一次完整的文件系统备份，然后在周一的晚上做了一次增量备份。在周二，目录 `/usr/jhs/proj/nr3/` 被删除了，包括该目录下的所有文件和子目录。接下来，在周三的明媚清晨，一个用户想要去恢复文件 `/usr/jhs/proj/nr3/plans/summary`，但是光去恢复这个文件是不行的，因为没有地方去保存它。因此目录 `nr3/` 和 `plans/` 必须先被恢复，包括它们的所有者、模式、时间等属性信息。所以说，这两个目录都必须保存在转储磁带中，即使它们本身并未被修改过。

从转储磁带中恢复一个文件系统是比较简单的。首先，在磁盘上创建一个空的文件系统，然后把最近的一次完整备份恢复出来。由于目录位于磁带的开头，所以它们先被恢复，这样就得到了文件系统的一个框架。然后是各个具体的文件被恢复。在此次恢复完成后，再针对第一个增量转储重复上述步骤，然后是第二个增量转储，等等。

虽然逻辑转储是比较简单的，但也有一些使用技巧。首先，由于空闲块列表不是一个文件，它未被转储，因此，在所有的转储都被恢复后，需要重新来构造它。这总是可以做到的，事实上，把所有文件中用到的块都刨去以后，剩下的块就是空闲的磁盘块。

另一个问题是链接。如果一个文件被链接到两个或多个目录，那么在恢复的时候，该文件只能被恢复一次，而且原来指向它的所有目录现在仍然要指向它。

还有一个问题是在 UNIX 文件中，可能会包含空闲区。在 UNIX 中，可以打开一个文件，写入几个字节，然后把文件指针移动到后面的某个位置，再写入几个字节。这样，对于这两段数据之间的间隔区域，它并不是文件的一部分，不应该被转储，也不应该被恢复。再比如，在内核转储文件中，在数据段和栈段之间往往有一个巨大的空闲区。如果不能正确地处理，那么在恢复内核文件时，可能会把该区域填充为 0，这样就会使整个文件大到与虚拟地址空间相同（如 2^{32} 字节或 2^{64} 字节）。

最后，设备文件、命名管道和其他的类似文件不能被转储，不管它们出现在哪一个目录中（它们不一定限定在 `/dev` 目录）。关于文件系统备份的更多内容，请参阅 Chervenak et al.(1998) 和 Zwicky (1991)。

文件系统的一致性

可靠性领域的另一个问题是文件系统的一致性。许多文件系统需要读取磁盘块，对它们进行修改，然后再写回到磁盘。如果在修改过的磁盘块被全部写回之前，系统崩溃了，那么文件系统可能会进入一种不一致的状态。如果未被写回的是 *i* 节点块、目录块或包含空闲链表的磁盘块，那么这个问题尤为严重。

为了解决文件系统的不一致性问题，许多计算机都带有一个工具程序来检验文件系统的一致性。例如，UNIX 中的 `fsck` 和 Windows 中的 `chkdsk`（或早期版本中的 `scandisk`）。在系统启动时，尤其是在系统崩溃后重新启动时，可以运行该程序。下面我们将介绍 `fsck` 的工作原理。`chkdsk` 略有不同，因为它运行在不同的文件系统中，但基本的原理都是类似的，即利用文件系统内部的冗余信息来修复不一致性。另外，各个文件系统的检验程序在检验文件系统（磁盘分区）的一致性时，是相互独立的。

一致性检查分为两种：数据块和文件。在检查块的一致性时，程序会建立两个表格。在每个表格中，每一块都有一个相应的计数器，初始值为 0。第一个表中的计数器记录了每个块在文件中出现的次数，第二个表中的计数器记录了每个块在空闲链表（或空闲位图）中出现的次数。

然后检验程序会读取所有的 *i* 节点。通过一个文件的 *i* 节点，可以列出该文件所用到的所有磁盘块的块号。每当读到一个块号时，就把它在第一个表中的计数器加 1。接着程序又去检查空闲链表或位图，查找所有未使用的块。每找到一个块，就把它在第二个表中的计数器加 1。

如果文件系统是一致的，那么对于每个块来说，要么在第一个表中为 1，要么在第二张表中为 1，如图 5.19(a) 所示。但是在发生系统崩溃后，这两个表可能如图 5.19(b) 所示。在图中，磁盘块 2 不出现在任何一个表中，这时它会被报告为丢失块（missing block）。虽然丢失块并不会造成什么损害，但是却浪费了磁盘空间，减少了磁盘容量。解决丢失块问题的方法很简单：直接把它们添加到空闲链表中。

另一种可能出现的情形见图 5.19(c)。在图中我们看到，磁盘块 4 在空闲链表中出现了 2 次（这种情形只会发生在空闲链表中，如果是用位图来管理空闲块，不会出现这种情形）。这种问题的解决方法也很简单：重新建立空闲链表。

最糟糕的情况是，同一个数据块出现在两个或多个文件中，如图 5.19(d) 中的磁盘块 5。如果删除其中的任何一个文件，磁盘块 5 将会被添加到空闲链表中。这样一来，就出现了同一个磁盘块既出现在文件中又出现在空闲链表中的情形。如果这两个文件都被删除后，那么这个磁盘块又会在空闲链表中出现两次。

对于上述情形，文件系统检验程序可以这样来处理：先分配一个空闲块，把磁盘块 5 中的内容复制到空闲块中，然后把它插入到其中的一个文件之中。这样，文件中的内容未改变（虽然我们几

乎可以肯定其中一个文件是不正确的),而文件系统的结构保持了一致。这一错误应该报告出来,以便用户检查。

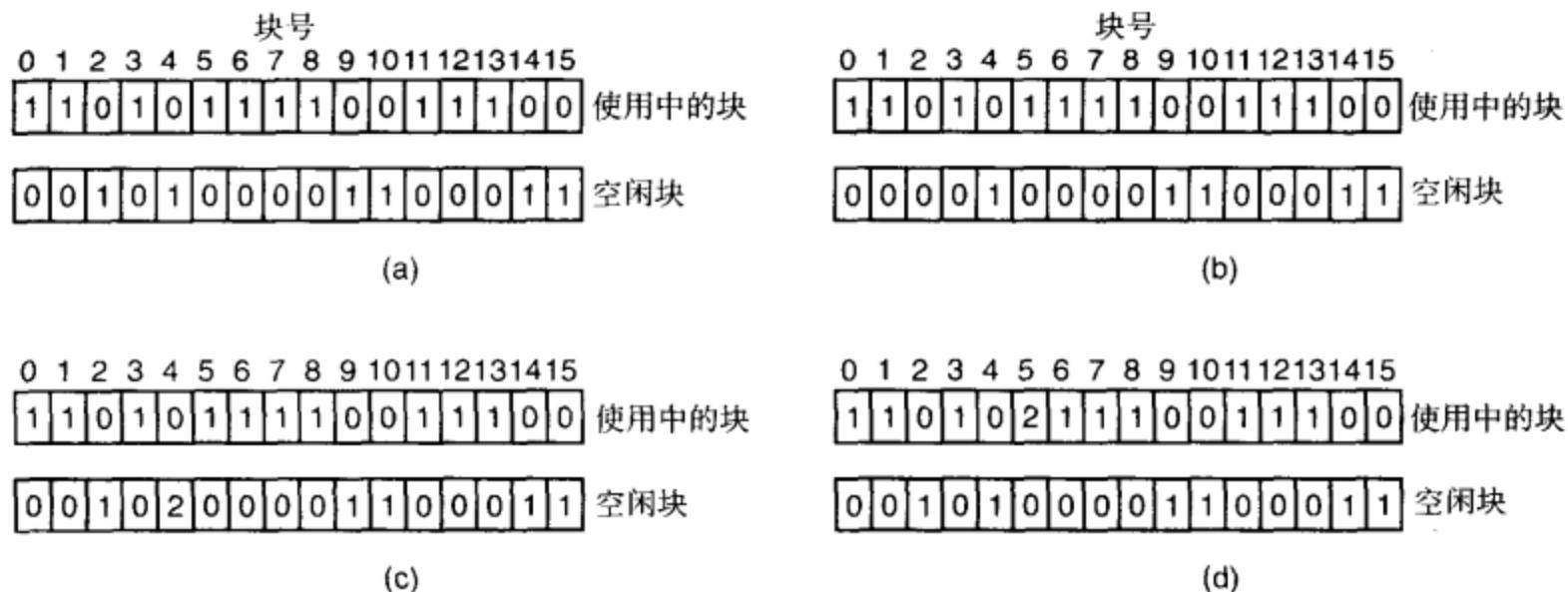


图 5.19 文件系统的状态: (a)一致; (b)丢失块; (c)空闲链表中有重复块; (d)重复数据块

除了检查每个磁盘块外,文件系统检验程序还会检查目录系统,这时也要用到一个计数器表,每个计数器对应于一个文件。检验程序从根目录开始,沿着目录树递归下降,检查文件系统中的每个目录。对于每一个目录中的每一个文件,将其使用计数器的值加1。如果是硬链接的情形,即同一个文件出现在两个或多个目录下,那么就要分别加1。如果是符号链接,那么就不进行计算,不会把目标文件的计数器加1。

当全部检查完成后,将得到一个表,该表用*i*节点号来索引,描述了每个文件被包含在多少个目录中。然后把这些数字与存储在文件*i*节点中的链接数目进行比较。当一个文件被创建时,它的链接数目被初始化为1,然后每增加一个指向它的硬链接,该计数值就加1。如果文件系统是一致的,那么这两个计数值应该相等。但是,有可能出现两种类型的错误,即*i*节点中的链接数太大或太小。

如果*i*节点中的链接数大于目录项的个数,这时,即使在所有的目录中都将该文件删除,文件链接数仍然大于0,因此*i*节点不会被删除。这一错误并不严重,可是却浪费了磁盘空间。文件虽存在,却不属于任何目录。为了改正这一错误,可以把*i*节点中的文件链接数设置为正确的值。

另一种错误则是一种潜在的灾难。如果两个目录项都链接到同一个文件,但其*i*节点中的链接数只有1,这时,如果删除任何一个目录项,*i*节点链接数将变为0。这样,文件系统将把该*i*节点标记为“未使用”,并释放该文件的所有磁盘块。这将导致一个目录指向一个未使用的*i*节点,而该节点所在的磁盘块可能马上被分配给其他文件。为了改正这一错误,同样可以把*i*节点中的链接数设置为目录项的实际数目。

由于效率的原因,检查磁盘块和检查目录的操作常常结合在一起(即仅需对*i*节点扫描一遍)。当然还存在着其他的一些检查方法。例如,目录项有确定的格式:*i*节点号和ASCII文件名。如果某个*i*节点号大于磁盘中*i*节点的总数,显然这个目录已经被破坏了。

此外,每个*i*节点都有一个模式字段。有些模式是合法的,但很不正常,比如0007,它不允许文件所有者及其所在用户组的成员进行访问,而其他的用户却拥有读、写和执行此文件的所有权限。在这种情形下,系统至少应该把此类文件报告出来,以便引起用户的注意。另外,拥有1000个目录项的目录也很可疑。而放在用户目录下,但为超级块用户所拥有,并且设置了SETUID位的文件,

也可能会有安全问题。因为当任何一个用户去执行该文件时，就会具有超级用户的能力。稍加努力，我们可以列出一长串的特殊情况，这些情况尽管合法，但却有必要报告给用户。

以上我们讨论了如何避免用户文件由于系统崩溃而受到破坏的情形，有时文件系统也要防止用户自身的误操作。例如，如果用户想输入

```
rm *.o
```

其本意是删除所有以.o结尾的文件（编译器生成的目标文件），但他不小心键入的是

```
rm * .o
```

（请注意，星号后面有空格），那么 *rm* 命令将会删除当前目录下的所有文件，然后报告说找不到.o文件。在有些系统中，当一个文件被删除时，系统所做的仅仅是在目录或*i*节点中设置某一位，标记该文件被删去，在实际需要前，该文件的磁盘块并不会返回到空闲链表中。因此，如果用户马上发现了操作错误，他可以运行一个特定的工具程序，恢复被删除的文件。在 Windows 系统中，被删除的文件被放在回收站中，如果需要的话，这些文件还可以被恢复。在它们从这个目录中删除之前，系统并不会回收它们的存储空间。

以上这种机制是不安全的。在一个安全的系统中，当一个文件被删除后，会用0或随机值来覆盖该文件所占用的数据块，这样另外一个用户就不能去访问这些数据。许多用户不知道数据能够存在多久，实际上，在一些被抛弃的磁盘上，常常能恢复出一些机密或敏感的数据（Garfinkel and Shelat, 2003）。

5.3.6 文件系统的性能

访问磁盘比访问内存要慢得多。在内存中读取一个字往往只需要 10 ns，而硬盘的读取速度为 10 MB/s，比内存访问慢 40 多倍，此外还要加上 5~10 ms 的寻道时间，还有旋转延迟时间，即等待要访问的扇区移动到磁头的下方的时间。如果只读一个字，那么内存的访问速度要比磁盘的访问速度快 100 万倍，正因为如此，许多文件系统在设计时都进行了各种优化，以提高系统的性能。在本小节我们将介绍其中的三项技术。

高速缓存

减少磁盘访问次数最常用的技术是块高速缓存（block cache 或 buffer cache）。在这里，高速缓存是指一组数据块，它们从逻辑上属于磁盘，但基于性能的考虑而保存在内存中。

高速缓存的管理有多种不同的算法，其中一个比较常用的算法是：对所有的读请求进行检查，看看所需的块是否在高速缓存中。如果在，则此次读请求可立即完成，无须去访问磁盘；如果不在，则首先把该块读入到高速缓存，然后再把它复制到所需的地方。这样，如果下次再要访问该数据块，就可以通过高速缓存来完成。

高速缓存的操作如图 5.20 所示。由于高速缓存中的数据块很多（通常有数千个），因此必须设计一种方法，以便能够快速地确定某个块是否在高速缓存中。通常的做法是对设备和磁盘地址建立哈希映射，通过哈希表来加快查找速度。如果两个块具有相同的哈希值，则把它们串在一条链表中。

当高速缓存已满时，如果又需要调入新的块，则需要把原来的某一块调出高速缓存（如果该块曾经被修改过，则需要把它写回到磁盘中）。这种情形与页式存储管理非常相似，因此，第 4 章中讨论过的各种页面置换算法，如 FIFO 算法、第二次机会算法、LRU 算法等都可以使用。两者的区别在于，高速缓存的访问频率相对要少一些，因此可以把所有块按 LRU 的顺序用链表链接起来。

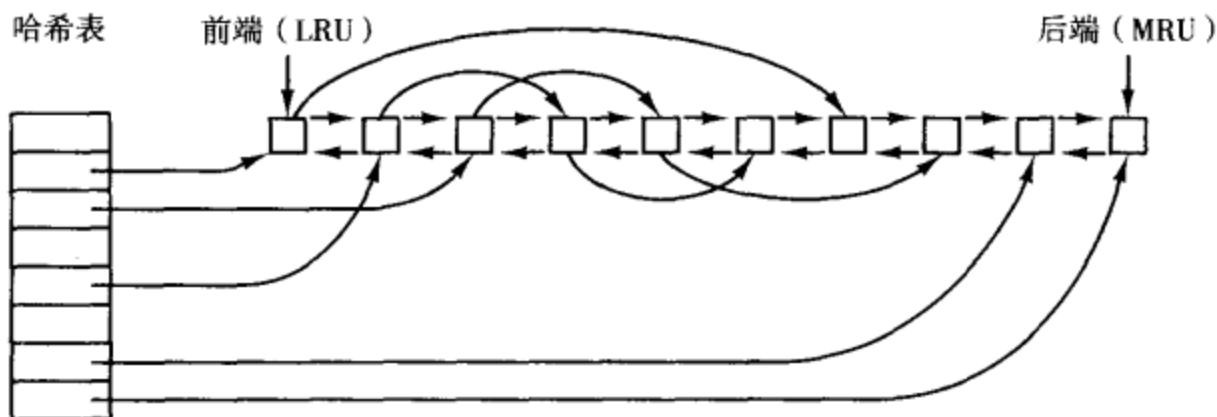


图 5.20 块高速缓存的数据结构

在图 5.20 中，我们看到除了哈希表中的冲突链表之外，还有一条双向链表，它根据块的访问顺序，把所有的块链接起来。位于链表前端的是最近最久未使用的块，位于链表后端的是最近刚使用过的块。当一个块被访问时，就把它从链表中摘下来，然后插入到链表的末尾。这样，就能够严格地满足 LRU 的顺序。

但是，这里又有一个新的难题。虽然我们可以实现严格的 LRU 算法，但与页面置换问题不同，在这里 LRU 算法并不一定合乎我们的要求。这与前一节讨论过的系统崩溃和文件系统的一致性有关。例如，假设有一个很重要的数据块，如 i 节点块，被读入到高速缓存中并被修改过，但还没有写回到磁盘。如果此时发生了系统崩溃，将会导致文件系统的不一致。事实上，如果我们把 i 节点块放在 LRU 链表的末端，那么在它到达链首并被写回磁盘前，可能要经过相当长的一段时间。

此外，有些磁盘块，如 i 节点块，很少在短时间内被访问两次。基于上述这些考虑，我们需要修改 LRU 方案，并注意以下两点：

1. 是否该数据块不久又要用到？
2. 是否该数据块会关系到文件系统的一致性？

基于这两个问题，可以把块分为 i 节点块、间接块、目录块、数据块、部分数据块等几类。对于最近可能不再需要的块，把它们放在链表的前端，这样，它们所占用的缓冲区很快就可以再次使用。对于不久后可能再次使用的块，如当前正在进行写操作的某个数据块，把它们放在链表的末尾，这样可以使它们在高速缓存中保存一段较长的时间。

第二个问题与第一个问题是独立的。如果某个块关系到文件系统的一致性（除数据块之外，其他块基本上都属于这一类），那么当它被修改后，不管它位于链表的什么位置，都应该把它立即写回到磁盘。通过把重要的块迅速写回磁盘，可以大大减少系统崩溃对文件系统造成伤害的可能性。这一点对于用户来说是很重要的，如果在一次系统崩溃中，用户丢失了一个文件，那么他可能会不太高兴；但如果这次崩溃造成整个文件系统的损坏，那么他就不仅仅是不高兴的问题了。

尽管上述方法能够保证文件系统的一致性不受破坏，我们也不希望将数据块放在高速缓存中很久之后才写入磁盘。如果有一个人正在使用 PC 机编写一本书。即使作者时常提醒出版社的编辑，在修改文件时要记得先把它写回磁盘。但总有人忘记这一点，他编辑的内容仅仅保存在高速缓存之中，尚未被写入到磁盘。这时如果系统崩溃，文件系统的结构并不会遭到破坏，但是他一整天的工作都将毁于一旦。

这种情形会让用户感到非常不满。为了解决这个问题，人们采用了两种方法。在 UNIX 系统中，有一个系统调用 `sync`，可以用来使所有被修改过的块立即写回到磁盘。然后，在系统启动时，一个名为 `update` 的程序就会进入后台运行，它处于无休止的循环之中，每隔 30 s 就会发出一个 `sync`

调用。这样做的优点是，即使发生系统崩溃，丢失的也只是30 s内的工作。这对于大多数人来说来，基本上还是可以接受的。

Windows操作系统的做法是：当一个数据块被修改时，立即把它写回到磁盘。这种把被修改的块立即写回磁盘的高速缓存，称为直写高速缓存（write-through cache）。与非直写高速缓存相比，这种方式需要更多的磁盘I/O。举例来说，假设一个程序需要写满一个1 KB的块，每次只写一个字符。在UNIX方式下，会把所有字符保存在高速缓存中，然后每隔30 s（或当该块被移出高速缓存时）才把这个块写回到磁盘。反之，在Windows方式下，每写入一个字符，就要去访问一次磁盘。当然，大多数程序都会有内部缓冲区，因此在通常情形下，在执行系统调用 write 时，并不是逐个字符写入，而是以行或更大的单位写入。

对于以上这两种不同的高速缓存策略，其另一个结果是：在UNIX系统中，如果未调用 sync 就移走（软）磁盘，往往会导致数据的丢失，有时还会破坏文件系统。而在Windows系统中，则不会出现这种情况。另外，之所以选择这两种不同的策略，是由于历史原因造成的。在开发UNIX系统时，它的开发环境使用的全都是硬盘，是不可移动的，而Windows系统则是从软盘世界发展起来的。时至今日，随着硬盘成为一种标准配置，UNIX的高效缓存管理方案，已经成为一种主流选择，即使是Windows系统也使用它来管理硬盘。

块预读

改进文件系统性能的第二种技术是在数据块被访问之前，预先把它们读入高速缓存，从而提高高速缓存的命中率。事实上，很多文件的访问都是顺序进行的。因此，当文件系统收到一个请求，去访问某个文件中的第 k 个数据块时，它会照办，但是在此次操作结束的时候，它会偷偷地去检查一下，看第 $k+1$ 个块是否在高速缓存中。如果没有，它就会发出一个调度，把第 $k+1$ 个块读进来。它的如意算盘是：当用户请求该数据块时，它已经在高速缓存中，或者至少它已经在半路上。

当然，这种预读策略只对文件的顺序访问有效。如果文件的访问是随机进行的，那么预读策略就毫无用处。不仅如此，它对系统的性能还是有害的，因为它需要读入一些无用的数据块，同时还要把一些可能有用的块移出高速缓存（如果这些块曾经被修改，还要把它们写回到磁盘）。为了判断预读是否可行，文件系统可以去跟踪记录每个被打开文件的访问模式。例如，可以用一个数据位来描述一个文件，表明该文件是处于“顺序访问模式”还是“随机访问模式”。起初，将该文件标为“顺序访问模式”（该位的值为1），如果出现了一次磁头的定位操作，就把该位清除。如果后来又出现了顺序访问模式，该位又重新被设置为1。这样，文件系统就可以据此来做出一个合理的猜测，即是否应该进行预读。如果猜错了，也没有关系，不会引起太大的性能下降，只是浪费了很少的一段磁盘带宽。

减少磁头臂移动

高速缓存和块预读并不是改善文件系统性能的仅有方法。另一种重要技术是通过把那些可能会顺序访问的块存放在一起（最好是在同一个柱面上），从而减少磁头臂的移动次数。对于一个输出文件，在对它进行写操作时，文件系统会根据需要，为它分配所需要的数据块，一次分配一块。如果空闲块是用位图的方式来管理的，而且整个位图都位于内存中，那么我们在分配新的空闲块时，能够很容易地进行选择，找到与前一个块尽可能近的空闲块。但如果采用的是空闲链表的方式，且它的部分内容存放在磁盘上，这时要想找到相邻的空闲块就困难得多。

然而，即使是采用空闲链表，也可以使用某种块聚簇技术。也就是说，磁盘存储空间的管理不是以块为单位，而是以一组相邻的块为单位。例如，假设每个扇区有512个字节，每个块的大小为1 KB（2个扇区），但是系统在分配磁盘存储空间时，是以2个块为单位（4个扇区）的。这种做法

不同于块大小为 2 KB 的情形，因为在高速缓存中依然使用 1 KB 大小的块，而磁盘与内存之间的数据传送也是以 1 KB 为单位进行的，但这样做的优点是：如果需要顺序访问一个文件，那么寻道次数可以减少一半，从而大大改善文件系统的性能。如果考虑到旋转定位问题，我们可以得到这种方案的一个变体。即在分配磁盘块时，系统应尽量把文件的连续块存放在同一个柱面上。

在使用 i 节点或类似结构的系统中，另一个性能瓶颈在于，即使读取一个很短的文件也要访问两次磁盘：一次是读取 i 节点，另一次是读取文件块。在通常情形下，i 节点的放置如图 5.21(a) 所示。图中，所有的 i 节点都存放在靠近磁盘开始的位置，因此 i 节点和相应文件块之间的平均距离是柱面总数的一半，这就需要很长的寻道延迟。

一个简单的性能改进方法是把 i 节点放在磁盘的中部。这样，i 节点与第一个文件块之间的平均寻道时间减为原来的一半。另一种做法是：把磁盘划分为多个柱面组，每个组有自己的 i 节点、数据块和空闲链表，如图 5.21(b) 所示（McKusick et al., 1984）。在创建一个新文件时，可以选取任意一个 i 节点，但在分配磁盘块时，首先要在该 i 节点所在的柱面组上查找。如果没有找到，再去查找与之相邻的柱面组。

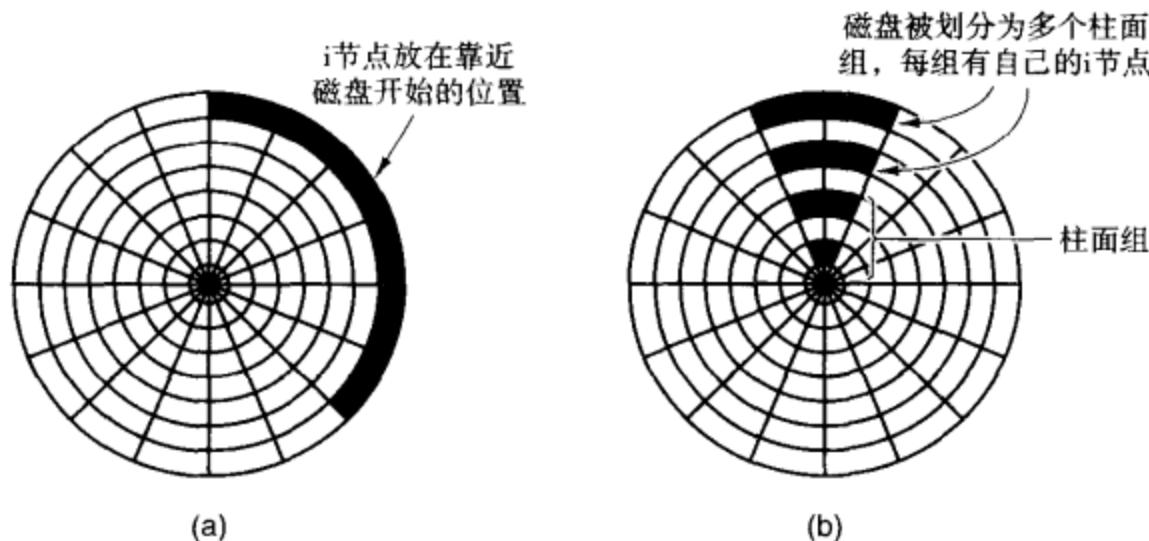


图 5.21 (a) i 节点放在磁盘开始的位置；(b) 磁盘被分为柱面组，每组有自己的块和 i 节点

5.3.7 日志结构的文件系统

技术的改进使得当前的文件系统面临着很大的压力。CPU 速度越来越快，磁盘容量不断增大，价格不断降低（但访问速度并没有太大提高），内存容量呈指数增长，等等。不过，也有一个参数并没有得到迅速的改善，这就是磁盘寻道时间。以上这些因素组合在一起，表明在许多文件系统中正出现着一个性能瓶颈。为了缓解这个问题，Berkeley 的研究人员设计了一种全新的文件系统，即日志结构的文件系统（Log-structured File System, LFS）。在本小节我们将简要讲述一下 LFS 的工作原理，如果要了解更详细的内容，请参阅 Rosenblum 和 Ousterhout (1991)。

促成 LFS 的基本想法是：CPU 越来越快，RAM 内存越来越大，磁盘高速缓存的容量迅速增加。因此，对于大多数的读操作请求来说，有可能直接从文件系统的高速缓存中得到满足，而不必去真正访问磁盘。基于这个观察，将来大多数的磁盘访问将会是写操作。这样一来，在有些文件系统中使用的预读机制（即在实际需要之前把数据块调入内存）对文件系统性能的改进就不再那么重要了。

更糟糕的是，在大多数文件系统中，写操作都是以较小的块为单位进行的，效率很差。因为在 50 μs 的写磁盘操作之前，需要有 10 ms 的寻道延迟和 4 ms 的旋转延迟。由于这些延迟的存在，使得磁盘的使用效率还不足 1%。

那么这些小块的写操作从何而来呢？考虑在 UNIX 系统中创建一个新文件。为了写这个文件，必须对目录的 i 节点、目录所在的块、文件的 i 节点和文件本身执行写操作。虽然这些写操作可以

延迟进行，但如果这样的话，万一系统发生崩溃，很容易使文件系统产生严重的不一致问题。因此，*i*节点一般都是立即写入的。

基于上述原因，LFS的设计人员决定重新设计UNIX的文件系统，他们希望在即使有大量小块随机写的情形下，也能实现磁盘的全速访问。系统的基本思想是把整个磁盘视为一个日志。所有的写操作都存放在内存的缓冲区中，并定期地收集到一个单独的段中，然后作为一个完整的连续段被写入磁盘，位于日志的末尾。因此，在一个段中，可能包含有*i*节点、目录块、数据块等，各种类型的内容混合在一起。在每个段的开头是一个摘要，描述了该段中的主要内容。如果段的平均长度约为1MB，那么就能把所有的磁盘带宽都能利用上。

在这种设计中，*i*节点依然存在，并且和UNIX中的*i*节点具有相同的结构。但与原来不同的是，这些*i*节点并不是存放在磁盘的固定位置，而是被分散在日志之中。一旦找到文件的*i*节点，就可以使用通常的方法来找到相应的文件块。毫无疑问，现在查找一个*i*节点变得更困难了，我们不能像UNIX那样，根据*i*节点号通过简单的计算就可以得到*i*节点的位置。为了能够找到*i*节点，我们需要维护一个*i*节点映射表，它以*i*节点号为下标，其中的第*i*项指向磁盘上的第*i*个*i*节点。这个映射表存放在磁盘中，但它同样被缓存起来，所以在大多数时候，它的最常用部分将会保存在内存中。

下面我们来小结一下前面讲过的内容。所有的写数据一开始都会存放在内存缓冲区中，每隔一段时间，缓冲区中的数据将以一个段的形式被写入到磁盘中，放在日志的末尾。当打开一个文件时，首先要在*i*节点映射表中查找该文件的*i*节点，一旦找到了*i*节点，也就知道了每个文件块的地址，这些文件块被存放在各个段中，即在日志内的某个地方。

如果磁盘容量无限大，那么上面的描述就是LFS的全部内容。但是，对于实际的磁盘来说，它们的容量都是有限的，因此日志将会逐渐地占满整个磁盘，这时新段不能添加到日志中。幸好，在许多现有的段中，会包含一些不再使用的块。例如，如果一个文件被重写了，那么它的*i*节点将指向新的块，但原来的那些块仍然占用着磁盘空间。

为了解决这个问题，在LFS中有一个清理工（cleaner）线程，它的工作就是循环地去扫描和压缩磁盘日志。它首先读取日志中的第一个段的摘要，找出其中的*i*节点和文件。然后去查找当前的*i*节点映射表，看看找到的这些*i*节点和文件块是否还在使用。若没有使用，这些信息将被丢弃；而还在使用的*i*节点和块将被读入内存，以便写入到下一个段中，而原来的段则标记为空闲，以便于存放新数据。这样，清理工线程沿着日志向前移动，不断地删除旧段，并把有效数据读入内存，写入下一个新段。因此，整个磁盘就变成了一个循环的缓冲区，写线程不断地在前面添加新段，而清理工线程则不断地从后面删除旧段。

在LFS中，处理的过程比较麻烦。当某个文件块被写回到一个新段时，先要在日志中找到该文件的*i*节点，并对其进行修改，然后把它放在内存中，以便写回到下一段。而*i*节点映射也必须进行相应的更新以指向新的副本。不过，总的来说，这种管理还是可取的，性能研究表明这种复杂性是完全值得的。对于小块的写操作，LFS的性能比UNIX要高出一个数量级。而在读数据和大块地写数据时，其性能也同于或者是好于UNIX。

5.4 文件系统的安全性

在文件系统中往往包含有用户的有用信息。因此，如何保护这些信息不被未授权的用户使用，就成为所有文件系统必须关注的一个内容。在以下几小节中，我们将讨论与安全和保护有关的一些问题。这些问题既适用于分时系统，也适用于通过局域网连接到共享服务器的个人计算机网络。

5.4.1 安全环境

人们经常把“安全”和“保护”这两个术语等同起来，交替使用。不过，它们还是有一些区别的，描述的是两类不同的问题。一般来说，**安全性** (security) 指的是一些综合性的问题。例如，采取各种各样的手段，包括技术、管理、法律和政治等，以确保文件不会被未授权的用户去读取或修改。而**保护机制** (protection mechanism) 指的是用来保护计算机信息的特定的操作系统机制。当然，这两者之间的界限也不是泾渭分明。下面，我们先来看一下安全性。在本章的后面部分我们再来讨论一些具体的保护机制和模型。

安全性有许多方面，其中较重要的三个方面是风险、入侵者和数据丢失。我们将逐一进行讨论。

风险

从安全的角度来看，计算机系统有三个主要的目标，同时也有三种相应的风险，如图 5.22 所示。第一个是**数据机密性** (data confidentiality)，也就是说，如何让一些比较敏感的数据处于机密状态。具体来说，如果数据的所有者认为这些数据只能对部分人开放，而其他人则不能访问，那么系统就必须保证未被授权的人无法看到这些数据。在最起码的情形下，数据所有者应该能够设定哪些人能看到哪些数据，而系统必须能实现这种需求。

目标	风险
数据的机密性	数据泄露
数据的完整性	数据被篡改
系统的可用性	拒绝服务

图 5.22 安全的目标和风险

第二个目标是**数据完整性** (data integrity)，也就是说，在未经数据所有者批准的情形下，未被授权的用户不能去修改任何数据。这里所说的修改包括对数据的内容进行修改，也包括删除数据和添加虚假数据。如果一个系统不能保证用户提交给它的数据不会被随意篡改，那么它就不配称为是一个信息系统。数据的完整性往往比机密性还要重要。

第三个目标是**系统可用性** (system availability)，也就是说，没有人能干扰系统的正常运行。目前，**拒绝服务** (denial of service) 攻击已经越来越普遍。例如，对于一台 Internet 服务器而言，如果向它发送极大量的访问请求，将会把它弄瘫，因为它的所有 CPU 时间都花在了检查和抛弃输入的访问请求。假设它需要 $100 \mu\text{s}$ 来处理一次 Web 网页访问请求，那么随便什么人，只要每秒钟给它发送 10 000 个访问请求，就会把它弄瘫。对于数据的机密性和完整性攻击，我们已经有一些有效的模型和技术来应对。而要防范这种拒绝服务攻击，则要困难得多。

安全问题的另一个方面是**隐私** (privacy)，也就是说，防止个人的隐私信息被别人滥用。这涉及到很多法律和道德问题。例如，政府部门能否搜集每个人的信息，以发现社会福利或税收方面的欺诈者？警察是否能够查询任何人的任何信息，以阻止有组织的犯罪？企业老板和保险公司是否有权利？如果这些权利与个人的权利发生冲突怎么办？所有这些问题都是很重要的，但已经超出了本书的范围。

入侵者

绝大多数的人都与人为善、遵守法律，那为何还要担心安全呢？因为还是有一小部分的人，他们不是那么友好，老想弄点麻烦出来（可能是为了他们自己的商业利益）。在安全领域，如果一个人跑到不属于他自己的领地上去窥探，则把他称为**入侵者** (intruder) 或敌人。入侵者可以分为两

类：消极的入侵者只是想读取未经授权的文件，而积极的入侵者则怀有恶意，他们试图去修改未经授权的文件数据。在设计一个安全系统时，必须知道你要防范的是哪一类入侵者。下面是较常见的几类入侵：

1. 非技术人员的偶然窥视。很多人在自己的工作台上都摆放着一台个人电脑，并被连接到一台共享的文件服务器上。如果系统不设置障碍的话，那么有些人可能会出于好奇心，去阅读别人的电子邮件或其他文件。例如，在大多数UNIX系统中，所有新创建的文件在默认情况下都是公开可读的。
2. 内部入侵者的窥探。学生、系统程序员、操作人员和其他一些技术人员常常以突破局部计算机系统的安全措施来作为个人能力的一种挑战。这些人往往都技术娴熟，并且愿意花大量的时间来从事这一努力。
3. 明确的偷窃企图。一些银行程序员试图从他们所在的单位窃取金钱。他们的方法五花八门，包括修改软件将利息的尾数据为已有，盗用多年未使用的账户，甚至是直接进行敲诈勒索（“赶快给我一笔钱，否则我将删除所有的银行记录。”）
4. 商业或军事间谍活动。间谍活动指的是由竞争对手或外国政府发起的一些有组织、有计划、资金充裕的活动，主要窃取对方的程序、商业秘密、专利想法、技术、电路设计和市场计划等。通常的表现形式为电话窃听，有的甚至通过天线来截获对方计算机发出的电磁辐射。

需要说明的是，防止敌对的外国政府来窃取军事秘密和防止学生在系统中插入一条有趣的“今日要闻”，这是完全不同的两码事。人们花在安全和保护上的开销显然取决于他们所设想的对手是谁。

恶意程序

安全领域的另一类问题是恶意程序，有时称为malware。从某种意义上来说，恶意程序的编写者也可以算是一名入侵者，而且他的技术水平更高。一个常规的入侵者与一个恶意程序之间的区别在于：前者是指一个人，他试图攻破系统的安全防范，去搞破坏；而后者是一个程序，它由具有入侵企图的人编写，并被散播到世界上。有些恶意程序的目的仅仅是为了搞破坏，而另一些程序的目的则比较明确。时至今日，恶意程序问题已经变得越来越严重，相关的资料也有很多 (Aycock and Barker, 2005; Cerf, 2005; Ledin, 2005; McHugh and Deek, 2005; Treese, 2004; Weiss, 2005)。

最有名的一种恶意程序是病毒 (virus)。所谓病毒，指的是一段程序代码，它依附在另一个合法程序中，并能不断地复制自己，就像生物学所说的病毒那样。除了复制自己之外，病毒还能做其他的事情。例如，它能在屏幕上打印一条消息、显示一幅图片、播放音乐或做其他的一些无害的事情。不过也有一些病毒程序，它们能修改、删除或偷窃文件（通过电子邮件把文件发送出去）。

病毒能够做的另一件事情是让计算机无法正常运行，这称为拒绝服务 (Denial Of Service, DOS) 攻击。通常的做法是大量地去消耗系统的资源，如CPU，或者是用垃圾数据来填满硬盘。病毒（以及其他一些恶意程序）还能用来制造分布式拒绝服务 (Distributed Denial Of Service, DDOS) 攻击。在这种情形下，当病毒正在感染一台计算机时，它并不会做什么特别的事情。然后在某个特定的日期和时间，遍布全球的计算机上的数千份病毒程序会同时发作；向它们的攻击目标（如某个政党或公司的主页）发出网页访问请求或其他的网络服务。这样，目标服务器和相应的服务网络很快就会负荷超载。

恶意程序往往与经济利益有关，许多垃圾邮件（也叫spam）是由感染了病毒或其他恶意程序的计算机网络所转发的。对于一台感染了这种程序的计算机，它将成为一台奴隶机，会随时把自己

的状态汇报给它的主人（Internet上的另一台机器）。然后它的主人会收集各个奴隶机上的电子邮件地址簿和其他文件中的电子邮件地址，并向它们发送垃圾邮件。恶意程序的另一种盈利模式是在被感染的计算机上安装一个击键日志程序（key logger），它会记录所有的键盘输入信息。然后从这些数据中过滤出有用的信息，如用户名、账号、信用卡号、到期时间等。这些信息将被发送给病毒的主人，用于贩卖或犯罪用途。

与病毒有关的是蠕虫程序（worm）。在这种方式下，病毒通过把自己附加在另一个程序上得以传播，当宿主程序被执行时，病毒程序也就开始执行。蠕虫程序是一个独立的程序，它通过网络来传播，把自身的代码复制到其他的计算机上。在Windows操作系统中，为每个用户设置了一个启动（Startup）目录，当用户登录系统后，该目录下的所有程序都会被执行。因此，蠕虫程序所要做的事情，就是想办法把自己（或指向自己的链接）存放在一台远端机的Startup目录下。当然还有其他的一些方式，可以使一台远端的计算机去执行已经被复制至该机器的文件系统中的某个程序文件。蠕虫的效果和其他病毒是差不多的，实际上，病毒和蠕虫之间的区分并不是很清楚，有些恶意程序采用了这两种方式来进行传播。

另一类恶意程序是特洛伊木马（Trojan horse）。特洛伊木马表面上是一个合法的程序，如一个游戏程序，或某个工具软件的“升级”版本。但当它被启动时，却执行了其他一些功能，如病毒或蠕虫。特洛伊木马给人的感觉是鬼鬼祟祟、偷偷摸摸。它不同于病毒和蠕虫，是由用户主动下载的。而且一旦用户识别出它的真正身份，就会把它删除掉。

另一类恶意程序是逻辑炸弹（logic bomb）。这是由公司的程序员编写的一段代码，并被秘密植入到操作系统中。只要该程序员每天输入口令，则一切正常。如果哪一天他突然被解雇了，那么在第二天，这个逻辑炸弹无法得到口令，于是便会被引爆。

逻辑炸弹一旦被引爆，后果可能就比较严重。例如，系统可能会清空磁盘、随机地删除文件、对关键程序进行难以察觉的修改，或者是加密一些重要的文件等。在这种情形下，公司只有两个选择，一是报警（但即使过了几个月，也不能保证定罪）；二是向勒索者让步，以天文数字的薪水重新雇佣他来纠正这个问题（并且希望他在解除原有炸弹时不要再放入一个新的炸弹）。

还有一种恶意程序是间谍软件（spyware）。在访问一些Web网站时，可能会被种上这种软件。间谍软件的最简单的形式仅仅是一个cookie。cookie是在Web浏览器和Web服务器之间交换的小文件，它们有着正当的用途。在cookie中包含有一些信息，可以让Web站点能认出你。打个比方来说，当你去维修自行车的时候，店员会给你一张票据，然后你就可以离开去做其他的事情。当你返回时，需要凭着这张票据来取车。由于Web连接不是持久连续的，因此，假设你正在浏览一家网上书店，然后对某本书表示感兴趣，那么书店会让你的浏览器接受一个cookie。当你浏览完毕，又选择了其他几本要买的书之后，就会点击订单页面，完成此次订购。这时，Web服务器会让你的浏览器返回在前一次会话中保存的cookie，然后使用其中的信息来生成一张总的购书清单。

一般来说，用于上述用途的cookie很快就会过期。它们是非常有用的，电子商务就依赖于它们。但有些Web站点对cookie的使用意图就不是这样简单。例如，Web站点上的广告往往是由信息提供商之外的公司提供的，同时，这些广告客户需要为这种服务给Web站点的所有者支付酬金。假设当你在访问一个有关自行车装备的页面时，获得了一个cookie。然后你去另一个卖衣服的Web站点，而同一家广告公司也在上面做广告，它就会收集你从别的站点获得的cookie。这时，你可能会突然发现，你正在浏览一些专为自行车选手而设计的手套或夹克。通过这种方式，广告客户能够收集很多关于你的兴趣方面的信息，而你自己可能不太愿意别人知道这么多的信息。

更糟糕的是，Web站点可以采用多种不同的方式把可执行代码下载到你的计算机上。大多数浏览器都支持插件程序（plug-in）来实现新增的功能，如显示新类型的文件。用户在接受新插件的时

候，往往并不知道该插件的具体功能是什么。或者用户可能愿意接受网上提供的某个程序，把桌面上的光标换成一个跳舞的小猫。再如，假设 Web 浏览器中有一个缺陷，那么不怀好意的某个远端站点可能会把用户引诱到某个精心构造的页面上，而该页面能够利用 Web 浏览器的缺陷，把用户不需要的程序安装到计算机上。

数据的意外丢失

除了恶意的入侵者所带来的威胁，数据也可能会出现意外丢失的情形。造成数据丢失的原因通常是：

1. 天灾人祸：火灾、水灾、地震、战争、暴乱或者是老鼠咬坏了磁带或软盘等。
2. 硬件或软件故障：CPU 故障、不可读的磁盘或磁带、通信故障、程序故障等。
3. 人为失误：不正确的数据输入、安装了错误的磁带或磁盘、错误的程序运行、磁带或磁盘丢失以及其他的一些失误。

以上这几种情形都可以通过保存足够多的备份来解决，而且最好是把备份数据放在与源数据相隔较远的地方。与防范聪明的入侵者相比，预防数据丢失的策略显得太过寻常，但实际上，后者所带来的损失可能会更大。

5.4.2 通常的安全攻击

要想找到安全方面的缺陷并非一件很容易的事情。测试系统安全性的常用方法是雇佣一群专家组成**猛虎组** (tiger team) 或者叫**渗透组** (penetration team)，看看他们能否闯入系统之中。Hebbard 等 (1980) 和他的研究生做了同样的工作。许多年来，这些渗透小组已经发现了在许多地方系统容易出现安全问题。下面我们列出了一些常见而且容易奏效的攻击方法。当你在设计一个系统时，要注意防范类似的攻击。

1. 请求内存页面、磁盘空间或磁带并读取其内容。许多系统在分配这些空间时并不会擦除以前的内容，而其中往往会有其他用户写入的重要信息。
2. 尝试非法的系统调用，或者是合法的调用但使用非法的参数，或者是合法的调用但使用合法而不合理的参数，类似于这样的用法很容易把系统搞晕。
3. 在登录的过程中键入 DEL, RUBOUT 或 BREAK。在有些系统中，这种做法会杀死口令检查程序，从而认为此次登录成功。
4. 试图修改保存在用户空间中的复杂的操作系统结构（如果有的话）。在某些系统（尤其是一些大型机）中，在打开一个文件时，程序会创建一个大的数据结构，其中包含有文件的名字和许多其他参数，并把它传给系统。在读写文件时，系统有时会去更新这些结构。因此，修改这些字段可能会造成安全性的严重破坏。
5. 写一段程序来哄骗用户，该程序将在屏幕上显示“login:”字样。这样，当有其他用户来使用这个终端时，他以为是系统的提示符，因此就心甘情愿地输入了自己的用户名和口令，而这些信息就被详细地记录下来。
6. 仔细查阅手册中出现的类似于“请不要做 X”的说明，然后去尝试 X 的各种各样的变化形式。
7. 说服系统程序员修改系统，使用户在以你的用户名登录时，跳过某些重要的安全性检查，这种攻击方法称为**暗门** (trapdoor)。

8. 当其他方法都无效时，渗透者可能会找到计算中心主任的秘书，给他一笔数量可观的贿赂，而这个秘书可能有很大的权限，能够访问各种重要的信息，但薪水很低。千万不要低估了这种由于人事管理所带来的问题。

以上这些方法以及其他一些攻击方法在 Linde (1975) 中都有讨论。在安全和安全测试方面，有很多的信息来源，尤其是在 Web 上。最近一项基于 Windows 系统的工作是由 Johansson 和 Riley (2005) 完成的。

5.4.3 安全性的设计原则

Saltzer 和 Schroeder (1975) 提出了一些用于指导安全系统设计的一般原则，下面简要地介绍他们的想法（基于 MULTICS 的经验）。

首先，系统的设计必须公开。假定入侵者不会知道系统的工作方式，这种想法只能是自欺欺人。

其次，默认的属性应该是不可访问。合法访问被拒绝的错误，会比未授权访问被批准的错误更快地报告出来。

第三，检查当前的权限。系统不应该先检查一遍权限，确定此次访问合法，然后把这个信息保留起来以备后用。许多系统都是在打开文件时检查权限，而不是在此之后。这意味着如果一个用户打开了一个文件，然后一连几个星期都没有关闭它，那么它依然有权限去访问该文件。而事实上，在这段时间内，文件的所有者早已修改了文件的保护属性，该用户已经失去了访问权限。

第四，给每个进程赋予一个最小的可能权限。如果一个编辑器只有权限去访问它所编辑的文件（在编辑器启动时指定），这时，即使它带有特洛伊木马病毒，也不会造成太大的损害。这个原则蕴含着一个小粒度的保护方案。在本章的后面我们还会谈到这样的方案。

第五，保护机制要力求简单、一致，并嵌入到系统的最底层。要想在一个现有的、不安全的系统上进行改进来增强安全性，这几乎是不可能的。同正确性一样，安全性也不是一种附加的特性。

第六，采取的方案必须为用户所接受。如果用户觉得文件的保护太麻烦，他们就不会去这么做。然而，当出现问题时他们又会不停地抱怨系统设计者。这时，你如果对他们说：“这是你自己的问题”，他们一般是不会接受的。

5.4.4 用户认证

许多保护方案都基于一个假定，即系统知道每个用户的身份。当用户登录时，系统所进行的检验其身份的过程称为 **用户认证** (user authentication)。大多数认证方法都基于以下的检验，即用户知道些什么、用户拥有些什么或者用户是什么等。

口令

使用最广泛的认证形式是用户输入口令。口令保护易于理解，也易于实现。在 UNIX 中它的原理是这样的：首先，登录程序会要求用户输入用户名和口令，并将该口令立即加密。然后登录程序会去查找口令文件，它通常是一行行的 ASCII 文件，每个用户占用其中的某一行。就这样一直找下去，直至找到包含有该用户登录名的那一行。如果这一行（也被加密过）的口令与计算得到的加密口令相符，则允许登录，否则就拒绝登录。

口令认证也很容易被突破。人们经常可以听到一群高中生甚至是初中生借助于他们的家庭电脑，闯入到某家大公司或某个政府机构的绝密系统中去的故事。而实际上，他们所做的事情只是想方设法去猜测系统的用户名和口令。

尽管近年来在口令安全方面也进行了许多的研究 (Klein, 1990), 但最经典的工作还是 Morris 和 Thompson (1979) 在 UNIX 系统上完成的。他们搜集并整理了一张可能的口令表, 其中包括姓名、街道名、城市名、一本中型词典中的单词 (包括反过来拼写的单词)、车牌号码以及很短的随机字符串。

接下来, 他们使用已有的加密算法对这些可能的口令进行加密, 形成加密后的口令列表。然后去检查各个用户的加密口令, 看是否出现在这个列表中。结果发现, 超过 86% 的口令都出现在他们的列表中。

从理论上来说, 假设所有的口令都由 7 个字符组成, 每个字符随机地取自于 95 个可打印的 ASCII 字符, 那么搜索空间为 95^7 , 约等于 7×10^{13} 种组合。即使以每秒 1000 项的速度进行加密, 也需要 2000 多年才能建立起这样的一张加密口令表。何况, 这张表要耗费 2000 万盘磁带。退一步说, 即使不采用这种任意的组合, 而只是要求在口令中至少包含一个小写字母、一个大写字母和一个特殊字符, 且长度至少为 7 个字符, 那么在加强口令的安全性方面也会有很大的改进。

尽管要求每个用户都合理地挑选口令在实际上是做不到的, 但 Morris 和 Thompson 还是提出了一种使他们自己的攻击方法 (即事先对大量的口令加密) 失效的方法。他们的想法是把每个口令与一个 n 位的随机数结合起来。当口令改变时, 该随机数也随之改变。随机数以明码的形式存放在口令文件中, 任何人都可以读取。然后, 我们不单单加密口令, 而是把口令和随机数连接起来一起加密, 并将这个加密的结果存储在口令文件中。

在这种情形下, 假设有一个人侵者试图构造一张可能的口令表, 并对它们进行加密, 然后把结果存放在一个排好序的文件 f 中, 以便于这些加密口令的快速查找。如果入侵者猜测口令可能是 *Marilyn*, 那么他不能像原来那样, 直接对 “*Marilyn*” 进行加密, 然后把结果存放在文件 f 中。相反, 他必须加密 2^n 个字符串, 如 *Marilyn0000*, *Marilyn0001*, *Marilyn0002* 等, 并把它们全部放入 f 中。因此, 这种技术把文件 f 的长度增大了 2^n 倍。UNIX 也使用了这种方法, 其 n 值为 12。这种方法也称为盐渍 (salting) 口令文件。有些 UNIX 版本把口令文件本身设置为不可读, 但用户可以通过一个程序来查询其中的内容, 这样就大大地减慢了破译的速度。

以上方法可以防止入侵者利用预先算出的一张大的加密口令表来破解登录口令, 但如果用户 *David* 就使用 “*David*” 来作为口令, 那谁都没有办法帮他。因此, 我们可以让计算机提出一些好的建议, 从而鼓励用户使用更安全的口令。例如, 有些计算机程序能够生成一些容易拼读的、无意义的单词来作为口令, 如 *fotally*, *garbungy* 或 *bipitty* (最好在其中再加入一些大写字母和特殊字符)。

有些计算机系统要求用户定期修改其口令, 从而减少因口令泄漏而造成的损失。最极端的做法是使用一次性口令 (one-time password)。在这种情形下, 用户备有一本口令书, 其中记载了一长串的口令, 用户每登录一次, 就更换一个口令。这样, 如果入侵者偶然破译了一个口令, 也没有什么用处, 因为用户在下次登录时就会使用不同的口令。当然, 如果用户不小心把口令本弄丢了, 那么就麻烦大了。

此外, 当用户在键入口令时, 计算机显然不应该把敲入的字符显示出来, 以防旁边有人窥视。口令最好也不要以未加密的形式存放在计算机中, 即使是计算中心的管理部门也不应该有未加密的口令副本。把未加密的口令到处乱放, 这简直是自找麻烦。

在口令安全方面, 一种变通的做法是让每个用户提供一系列的问题和答案, 并以加密的形式存放起来。这些问题应该是精心挑选的, 用户非常清楚它们的答案, 无须把它们记录在纸上。以下是一些典型的问题:

1. 谁是 *Mariolein* 的姐姐?

2. 你的小学是在哪条街上?
3. Woroboff 小姐教哪门课程?

在登录时，计算机会随机地选择一个问题，并检查用户的回答是否正确。

口令安全的另一种变形是查问-回答 (challenge-response)。在这种方式下，用户在注册时会选择一个算法，如 x^2 。然后当用户在登录时，计算机会显示某个参数，比如 7，这时用户需要键入 49。这种算法可以经常变动，早晚不同，每天不同，并且在不同的终端上也不相同。

物理认证

与口令完全不同的另一种认证方法是检查用户是否有某些“证件”，通常是一些带有磁条的塑料卡。将卡片插入终端后，系统就可以检查出卡片的所有者。这种方法可以和口令一起使用，也就是说，用户成功登录的前提条件是：第一，他必须拥有卡片；第二，他知道登录的口令。自动取款机就是这样工作的。

另一种方法是测量那些难以伪造的物理特征。例如，终端上的指纹或声纹阅读机可以认证用户的身份（如果用户能告诉计算机他的名字，则整个过程会快得多，因为计算机不需要把他的指纹与整个数据库进行比对）。直接的视觉识别技术现在还没有到实用的阶段，但终究有一天会实现。

另一种技术是签名分析。用户使用一枝与终端相连的特制笔进行签名，计算机会把这份签名与在线存储的已知样本进行比较。当然，更好的做法不是去比较签名，而是去比较签名时笔的移动情况，一个优秀的模仿者或许能模仿你的签名，但是，对于你在签名时行笔的确切顺序，他就不清楚了。

手指长度分析也是一项非常实用的技术。在这种方式下，每个终端都会有一个形如图 5.23 的设备。用户把手插入到这个设备中，他的各个手指的长度就会被记录下来，并与数据库中存放的数据进行比对。



图 5.23 测量手指长度的设备

我们还可以举出更多的例子，但下面两个例子能说明一个很重要的问题。我们知道，猫和其他一些动物会在它们的领地边界撒上一泡尿作为标记，这样它们就可以很容易地彼此区分。假设有人设计出一种尿液分析的小设备，能即时分析尿液，给出身份证明。然后，在每个终端上都配置这样

的一个设备，旁边写着：“登录时，请在此排放尿液”。这或许是一个绝对不可突破的系统，但用户显然难以接受。

下面的方案同样也很难让用户接纳：系统由图钉和小型摄谱仪组成，在登录时，需要用户把拇指按在图钉上，提取一滴血液来进行摄谱仪分析。总之，无论是哪一种认证方案，最重要的是必须为用户所接受。手指长度测量可能不会引起任何问题，但即使是一些非冒犯性的方法，如在线存储指纹，许多人也不愿接受。

应对策略

一般来说，人们总是在计算机系统遭受到入侵并造成较大损失之后，才会对安全性问题引起足够的重视，这时，他们常常会采取很多方法来增加未授权访问的难度。例如，每个用户只能在指定的某台终端上登录，而且只能在一周的某几天或一天中的某几个小时内登录。

拨号电话可按如下方式工作：任何人都可以拨号和登录，但一旦登录成功，系统立即中断连接，并以某个预先设定好的号码回呼用户。在这种方式下，入侵者就无法从随意一根电话线上进入系统。他如果要进入系统，只能通过自己的（家庭）电话。而且，不论是否回呼，系统应该用至少10 s来检查用户在拨号线上键入的口令。如果碰上几次连续的登录失败，这个时间还要增加。这样一来，就可以减少入侵者非法闯入的可能性。在连续三次登录失败后，电话线应该被中断10 min，并通知相应的安全人员。

所有的登录信息都应该被记录下来。这样，当用户登录时，系统可以告知他上次登录的时间和终端，以便他能检测出可能的非法入侵。

我们还可以通过设置陷阱来捕获入侵者。最简单的做法是：设置一个特殊的登录名，它的口令也非常简单（如登录名为guest，口令为guest）。当发现有人使用这个名字登录时，就立即通知系统的安全专家。其他的陷阱可能是操作系统中比较容易发现的一些缺陷，或者是类似的东西，它们是专门为了捕获入侵者而设计的。Stoll（1989）就曾写过一篇有关陷阱的有趣报道，他设置了一些陷阱，捕获了一个潜入某大学计算机系统中意图搜寻军事机密的间谍。

5.5 保护机制

在前面几个小节中，我们看到了许多潜在的问题。其中有些是技术性的，有些是非技术性的。在以下几个小节，我们将主要讨论在操作系统中用来保护文件和其他信息的一些技术手段。所有这些技术都明确地区分了策略（谁的数据需要保护，需要提防的是谁）和机制（系统如何来实现这个策略）。关于策略与机制的分离，请参见 Sandhu（1993）中的讨论。这里的重点主要是机制，而不是策略。

在有些系统中，保护是通过一个名为访问监视器（reference monitor）的程序来实现的。在每次访问一个被保护的资源时，系统会要求访问监视器去检查此次访问是否合法。而访问监视器会去查找它的策略表，并做出相应的决定。下面我们来介绍访问监视器的运行环境。

5.5.1 保护域

一个计算机系统会包含许多需要保护的“对象”，这些对象可能是硬件（如CPU、内存段、磁盘驱动器或打印机），也可能是软件（如进程、文件、数据库或信号量）。

每个对象都有一个唯一的名字（通过该名字来访问对象）和一组可以被进程执行的操作。例如，对文件可以进行read和write操作，对信号量可以进行up和down操作。

显然，我们需要某种方法来禁止进程去访问未经授权的对象。此外，保护机制还应该提供一种方式，能够把进程限定在一组合法的操作子集上。例如，对于文件 F，进程 A 有读的权限，但没有写的权限。

为了讨论不同的保护机制，这里有必要引入域（domain）的概念。域是一组（对象，权限）对的集合，每个对标明了一个对象和一个可执行的操作子集，这里所说的权限（right）指的是允许执行某个操作。一般来说，一个域对应于一个用户，表明该用户能做什么、不能做什么，但一个域也不仅仅限定在一个用户。

图 5.24 显示了三个域，并给出了每个域中的对象和相应的操作权限[读 R，写 W，执行 X]。请注意，打印机同时属于两个域。另外，虽然在这个例子中没有体现出来，但同一个对象可以出现在多个域中，而且在不同域中的权限也可以不同。

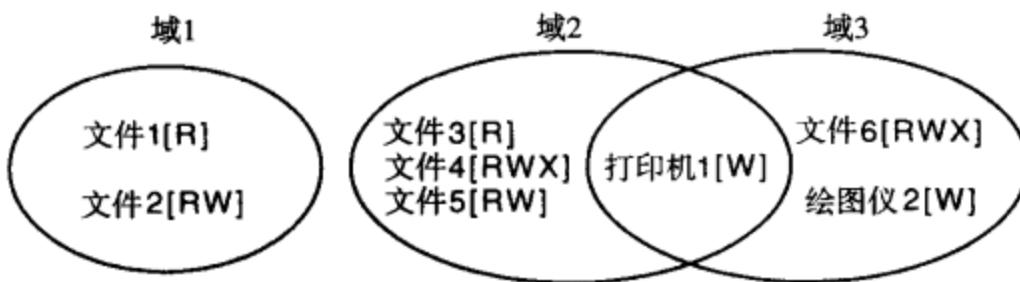


图 5.24 三个保护域

在任意时刻，每个进程总是运行在某个保护域中。换句话说，它可以去访问某些对象，而且对每个对象有一定的访问权限。进程在执行时，可以从一个域切换到另一个域，切换的规则完全取决于系统的具体实现。

为了使保护域的概念更加具体，我们来看一下 UNIX。在 UNIX 中，进程的域是用它的 UID 和 GID 来定义的。给定任何一个（UID，GID）对，可以建立一个完整的表，列出所有可以访问的对象（文件，包括用设备文件来表示的 I/O 设备等）以及相应的访问权限，如可读、可写或可执行。如果两个进程具有相同的（UID，GID）对，那么它们能够访问完全相同的一组对象；如果两个进程具有不同的（UID，GID）对，那么它们能够访问的对象集是不同的，尽管在多数情况下，这些对象集是相互重叠的。

此外，UNIX 中的每个进程包含两部分：用户部分和内核部分。当进程执行了一个系统调用时，将会从用户部分切换到内核部分，而内核部分具有与用户部分不同的访问对象集。例如，内核可以访问物理内存中的所有页面、整个磁盘空间以及其他受保护的资源。因此，一个系统调用将会导致一次域切换。

如果一个进程使用 exec 执行了某个带有 SETUID 或 SETGID 位的文件，那么它将获得一个新的有效 UID 或 GID。由于（UID，GID）对发生了变化，它所能够访问的文件和相应的访问权限也就发生了变换。也就是说，运行一个带有 SETUID 或 SETGID 的程序也会导致一次域切换。

这里会遇到一个很重要的问题：系统如何来记录每个对象是属于哪个域呢？从概念上来说，我们可以用一个大的矩阵来表示，矩阵的行表示域，列表示对象，而每个元素表示相应的访问权限。图 5.24 所对应的矩阵如图 5.25 所示。给定该矩阵以及当前的域号，系统就可以判断出针对某个对象的某次访问是否是合法的。

如果把域本身也视为对象，就可以很容易通过 enter 操作把域切换也包含在矩阵模型中。图 5.26 重新显示了图 5.25 的矩阵，只是增加了三个域作为对象。域 1 中的进程可以切换到域 2，但一旦切换后，就不能再回到域 1。在 UNIX 中，这种情况描述的是执行一个带有 SETUID 位的程序。在本例中，其他的域切换都是不允许的。

		对象							
		文件1	文件2	文件3	文件4	文件5	文件6	打印机1	绘图仪2
域	1	读	读写						
	2			读	读写执行	读写		写	
	3						读写执行	写	写

图 5.25 一个保护矩阵

		对象										
		文件1	文件2	文件3	文件4	文件5	文件6	打印机1	绘图仪2	域1	域2	域3
域	1	读	读写							进入		
	2			读	读写执行	读写		写				
	3						读写执行	写	写			

图 5.26 在保护矩阵中把域作为对象

5.5.2 访问控制列表

实际上，我们很少存储图 5.26 那样的矩阵，因为它是一个非常大的稀疏矩阵。大多数域只会访问很少的一些对象，因此存储这样的一个大而空的矩阵会浪费大量的磁盘空间。现实的做法主要有两种，即按行或按列来存储矩阵，而且只存储非空的元素。这两种方法有很大的区别。在本小节，我们将介绍按列存储。在下一节，我们将讨论如何来按行存储。

在第一种存储技术中，每个对象被赋予一个有序列表，其中包含了可以访问该对象的所有域，以及访问的方式。这个列表称为访问控制表（Access Control List, ACL），如图 5.27 所示。在图中有三个进程，每个进程属于不同的域。为方便起见，我们假定每个域只对应于一个用户，即用户 A、B 或 C。在安全领域，通常把用户称为主体（subject）或主角（principal），而把它们所拥有的东西称为对象（object），如文件。

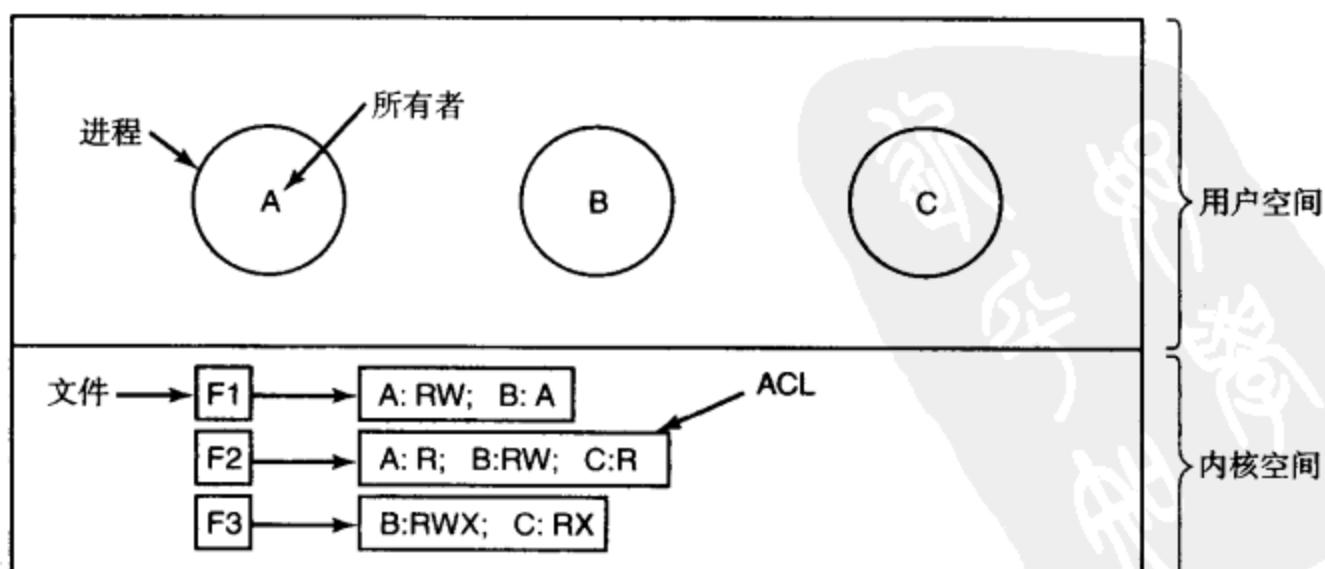


图 5.27 使用访问控制表来管理文件的访问

每个文件都有一个相关联的 ACL。在文件 F1 的 ACL 中，有两项内容，用分号隔开。其中，第一项是说用户 A 的所有进程都能读、写该文件；第二项是说用户 B 的所有进程都能读取该文件。除此之外，这些用户的其他访问方式以及其他用户的所有访问方式都被禁止。请注意，权限是授给用户的，而不是进程。也就是说，只要该保护系统仍在运行，那么用户 A 的任何一个进程都能读、写文件 F1。至于这些进程的个数是 1 个还是 100 个，这没有什么区别。因为起作用的是用户 ID，而不是进程 ID。

文件 F2 的 ACL 有三项内容，A, B 和 C 都能读取该文件，但只有 B 才能修改该文件。除此之外，其他的访问方式都是不允许的。文件 F3 显然是一个可执行程序，因为 B 和 C 都能读取和执行该文件。此外，B 还能修改该文件。

这个例子阐述了 ACL 方式下最基本的保护形式，而在实践中，往往用到了更加复杂的系统。首先，目前我们只考虑了三种基本的访问权限：读、写和执行。实际上，还有其他一些权限。其中，有些权限是通用的，也就是说，适用于所有的对象；而另一些权限是专用的，面向于特定的对象。通用权限的例子有：注销对象和复制对象，它们适用于所有类型的对象。专用权限的例子有：针对邮箱对象的添加消息，针对目录对象的按字母顺序排序，等等。

当目前为止，我们的 ACL 项仅用于单个用户。而许多系统都支持用户组（group）的概念。每个组都有自己的名字，可以包含在 ACL 中。关于组的语义，有两种不同的变化。在有些系统中，每个进程有一个用户 ID（UID）和一个组 ID（GID），而 ACL 项的形式为

UID1, GID1: 权限 1; UID2, GID2: 权限 2; ...

在这种情形下，当进程去请求访问一个对象时，系统会根据调用者的 UID 和 GID 来进行检查。如果它们出现在 ACL 中，则可以找到相应的访问权限；如果它们未出现在 ACL 中，则此次访问是非法的。

以上这种分组方式引入了角色（role）的概念。假设在一个系统中，Tana 是系统管理员，因而位于 sysadm 组。另外，公司的员工组织了几个俱乐部，而 Tana 是养鸽俱乐部的成员。该俱乐部的成员都属于 pigfan 组，能够访问公司的计算机来管理他们的鸽子数据库。因此，ACL 的部分内容如图 5.28 所示。

文件	访问控制表
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

图 5.28 两个访问控制表

如果 Tana 试图去访问其中的一个文件，其结果取决于她当前登录进来的是哪个组。当她在登录时，系统会询问她打算使用哪一个组，或者采用不同的登录名和口令来加以区分。这样做的目的是为了防止 Tana 在以养鸽爱好者的角色出现时，去访问系统的口令文件。如果她想要去访问该文件，那么只能以系统管理员的身份重新登录。

在有些情形下，无论用户当前登录进来的是哪一个组，她都能访问某些文件。这种情形是通过引入通配符（wildcard）来实现的。例如，针对口令文件，如果它的 ACL 项为

tana, * :RW

那么不管 Tana 以何种身份登录，她都能访问这个口令文件。

还有另外一种可能性就是，如果在用户所属的某个组中，存在有某种访问权限，那么该用户的访问就是合法的。在这种情形下，隶属于多个组的用户在登录时不需要指定他想参加哪个组，因为所有这些组都能始终在起作用。这种方法的缺点是，它减少了封装性。例如，Tana 能够在一次鸽迷聚会时去编辑系统的口令文件。

分组和通配符的使用引入了一种可能性，使我们能够有选择地防止一个特定的用户去访问某个文件。例如，

```
virgil, *:(none);*,* :RW
```

这句话的意思是，除了 Virgil 以外，所有的人都可以去读、写该文件。它之所以能够有效，是因为 ACL 项的扫描是有顺序的。系统首先发现 Virgil 位于第一项中，它的访问权限为空，因此，这条规则将被执行。然后搜索即告终止，而后面的第二项，即让全世界都能访问该文件则不会被处理。

在组的使用上，还有另外一种做法。每个 ACL 项不是由 (UID, GID) 对组成的，而是要么是 UID，要么是 GID。例如，文件 *pigeon_data* 的某一项是

```
debbie: RW; phil: RW; pigfan: RW
```

这表明 Debbie, Phil 和 *pigfan* 小组的所有成员都能对该文件进行读写操作。

有时可能会出现这样的一种情形：一个用户或组对某个文件具有一定的访问权限，但后来文件的所有者反悔了，想撤销这个权限。在访问控制表的方式下，可以很方便地撤销一个之前授与的权限，只要去编辑 ACL 进行相应的修改即可。然而，如果 ACL 的检查是在文件被打开时进行的，那么这种改动只能等到下次 *open* 调用时才能生效。而对于一个已经被打开的文件而言，对它的访问不会受到任何影响，即使该用户的访问权限已经被取消。

5.5.3 权能

上一小节讨论的是按列来存储图 5.26 中的矩阵，下面我们来看一下如何按行来分割该矩阵。在这种方式下，每个进程都有一个对应的列表，其中包含了该进程可以访问的所有对象，以及每个对象的操作权限，换句话说，也就是它的域。这张表称为权能表 (capability list)，或者叫 C 表，其中的每一项称为权能 (Dennis and Van Horn, 1966; Fabry, 1974)。图 5.29 显示了三个进程以及它们的权能表。

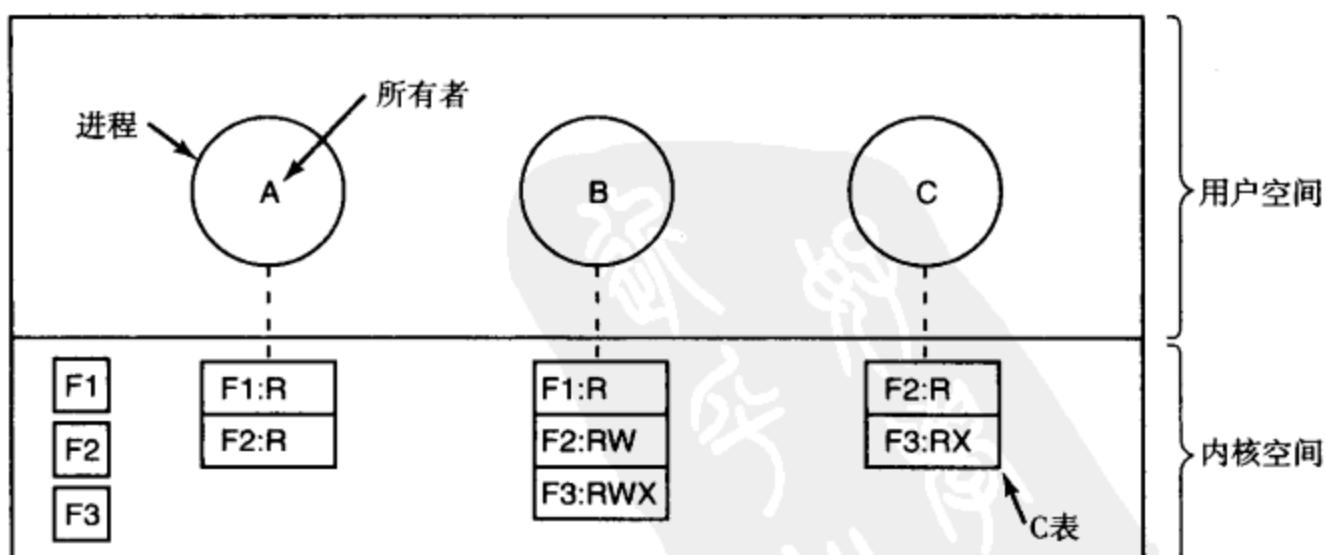


图 5.29 在权能方式下，每个进程都有一个权能表

每个权能给予了用户针对某个对象的一定权限。例如，在图 5.29 中，用户 A 的进程能够读取文件 F1 和 F2。一般来说，一个权能由一个文件 (或对象) 标识符和一个用于描述权限的位图构成。

在类 UNIX 的系统中，文件标识符主要是指 i 节点号。权能表本身也是一个对象，可以在其他权能表中指向它，这样就使得子域的共享变得更加容易。

显然我们应该防止用户去篡改权能表。目前主要有三种保护方法。第一种方法需要一个带有标记的体系结构 (tagged architecture)。也就是说，在硬件设计时，每个内存字都有一个额外的（标记）位，表明该内存字是否包含有权能。这个标记位不会用在算术运算、比较运算等普通的指令中，它只能被运行在内核态下的程序（即操作系统）所修改。带有标记的体系结构的机器已经被造出来了，而且运行良好 (Feustal, 1972)。IBM AS/400 就是一个典型的例子。

第二种方法是把 C 表保存在操作系统的内部，权能的访问是通过它们在权能表中的位置来进行的。一个进程可能会说“从权能 2 所指向的文件中读取 1 KB 数据”。这种形式的寻址方式类似于 UNIX 系统中的文件描述符。Hydra 就是这样实现的 (Wulf et al., 1974)。

第三种方法是把 C 表加密后，直接保存在用户空间中。这种方法尤其适合于分布式系统，它的工作原理如下：假设一个客户进程向远端的服务器（如文件服务器）发送一条消息，申请为它创建一个对象，那么服务器会创建该对象并随同生成一个长的随机数，即检查域。在服务器的文件表中会为该对象预留一个空槽，用来存放它的检查域和磁盘块的地址等信息。在 UNIX 中，检查域存放在服务器的 i 节点中，它不会返还给用户，也不会放在网上。接下来，服务器会生成并返回一个权能给用户，其形式如图 5.30 所示。

服务器	对象	权限	$f(\text{对象}, \text{权限}, \text{检查})$
-----	----	----	--------------------------------------

图 5.30 一个加密保护的权能

返回给用户的权能包含了服务器的标识符、对象号（服务器中的表格索引，实际上就是 i 节点号）和权限（以位图的形式存放）。对于一个新创建的对象，所有的权限位都被置位。最后一个字段的内容，是把对象、权限和检测域作为参数，去运行一个单向的加密函数 f 所得到的结果。

当用户想要去访问某个对象时，它就会向服务器发出一个请求，并附上相应的权能。服务器会抽取出其中的对象号，并以此为索引去访问它内部的表格，找到相应的对象。然后它会去计算函数 $f(\text{对象}, \text{权限}, \text{检查域})$ ，其中，前两个参数来自于权能本身，而第三个参数来自于服务器内部的表格。如果运算的结果与权能中的第四部分是一致的，那么此次请求将被批准；否则，此次请求将被拒绝。如果一个用户试图去访问别人的对象，由于他不知道该对象的检查域，所以无法正确地伪造出权能中的第四个字段，因此他的访问请求将被驳回。

用户可以要求服务器生成并返回一个弱一些的权能，如只读访问。首先，服务器会验证这个权能是否有效，如果是，就计算 $f(\text{对象}, \text{新权限}, \text{检查域})$ 生成一个新的权能，并把该值放在第四个字段中。请注意，这里使用了原始的检查域值，因为其他的权能还要用到它。

这个新的权能被发回给用户进程，然后用户可以把它转发给他的一位朋友，这样他的朋友也可以去读取该文件了。但如果他的朋友不满足于此，还想获得更多的访问权限，因此私自去修改权限位，把某些位置为 1。不过，服务器会检测到这种异常，因为当权限位被修改后，相应的 f 值也会发生变化，从而与权能中的原有值不一致。由于这个朋友不知道检查域的值是什么，因此他无法去伪造相应的 f 值。这种方案主要是为 Amoeba 系统设计的，在该系统中得到了广泛的应用 (Tanenbaum et al., 1990)。

除了与具体对象有关的权限，如读和执行，权能通常还会有一些通用的权限 (generic right)，能够适用于所有的对象。例如，

1. 复制权能：为同一个对象创建一个新的权能。
2. 复制对象：创建一个复制对象，使之具有新的权能。
3. 删除权能：从C表中删除一项，该操作不会影响到相应的对象。
4. 删除对象：永久地删除对象及其权能。

关于权能系统的最后一点评述是：如果把C表保存在内核中，那么要想撤销对一个对象的访问权限是比较困难的。系统很难找出该对象的所有权能并将其收回，因为这些权能可能存储在遍及整个磁盘的C表之中。一种方法是让每个权能都指向一个间接对象，然后再由该间接对象去指向真正的对象。这样一来，系统就能很方便地中断这种连接关系，从而使权能失效。也就是说，如果后来又有一个指向该间接对象的权能被提交给系统，那么用户就会发现这个间接对象现在指向的是一个空对象。

在Amoeba系统中，撤销是很容易的。所要做的事情就是去修改与对象一起存储的检查域，这样，当前的所有权能都将失效。不过，这种方案不支持有选择的撤销。比如说，我们想收回John的访问权限，而其他人的权限不变。这个缺陷在所有的权能系统中都普遍存在。

另一个常见的问题是如何防止一个有效权能的所有者把它复制给1000个其他的用户。在Hydra系统中，是由内核来管理权能，因此能够解决这个问题。而这种解决方案在Amoeba这样的分布式系统中，效果并不太好。

另一方面，权能机制能够有效地解决移动代码问题。当一个外部程序开始执行时，给它一个功能很弱的权能表，只包含了一些计算机用户同意的基本权限，如屏幕显示的权限，在一个专用的空白目录下读、写文件的权限，等等。在这种情形下，该程序无法去访问任何其他的系统资源，从而被有效地封闭在一个沙箱之中。这种让程序代码以最少的访问权限去运行的策略，称为**最少特权原理**（principle of least privilege），它是设计一个安全系统的重要指导原则。

总之，ACL和权能这两种方法有一些互补的属性。权能的效率非常高，因为当进程说“打开权能3所指向的文件”时，无须进行检查。而在ACL方式下，可能需要对ACL进行一次（长时间的）搜索。如果不支持分组功能，那么当系统需要把某个文件的读权限授予每个用户时，就不得不在ACL中把所有的用户都列举出来。权能还支持进程的封装，而ACL则不行。另一方面，ACL支持有选择的权限撤销功能，而权能不支持。最后，如果一个对象被删除而相应的权能未被删除，或者反过来，如果权能被删除而相应的对象未被删除，那么就会带来一些问题。而在ACL方式下，则没有这个问题。

5.5.4 秘密通道

尽管使用了访问控制表和权能表，安全泄漏仍然可能发生。在本小节，我们将会看到，即使从数学上严格地证明了某种信息泄漏是绝无可能的，但它仍有可能发生。这方面的内容是Lampson（1973）提出的。

Lampson的模型最初来源于一个分时系统，但同样的思想可以适用于LAN和其他的多用户环境。在它的基本形式中，涉及到某台受保护的机器上的三个进程。第一个进程是客户进程，它需要第二个进程（即服务器进程）来帮它执行某项工作。客户和服务器都不完全信任对方。例如，服务器的工作是帮助客户填写税务单，客户担心服务器会偷偷记录这些财务数据，并整理成一张工资表，谁每个月挣多少钱，然后把它卖给别人。而服务器又担心客户进程会试图窃取它的税务程序。

第三个进程是合作者进程，它与服务器进程合谋去窃取客户的机密数据。合作者进程和服务器进程往往属于同一个用户。这三个进程如图5.31所示。我们的目标是要设计一个安全系统，使

得服务器进程无法向合作者进程泄漏从客户进程那里获得的信息。Lampson 把它称为约束问题 (confinement problem)。

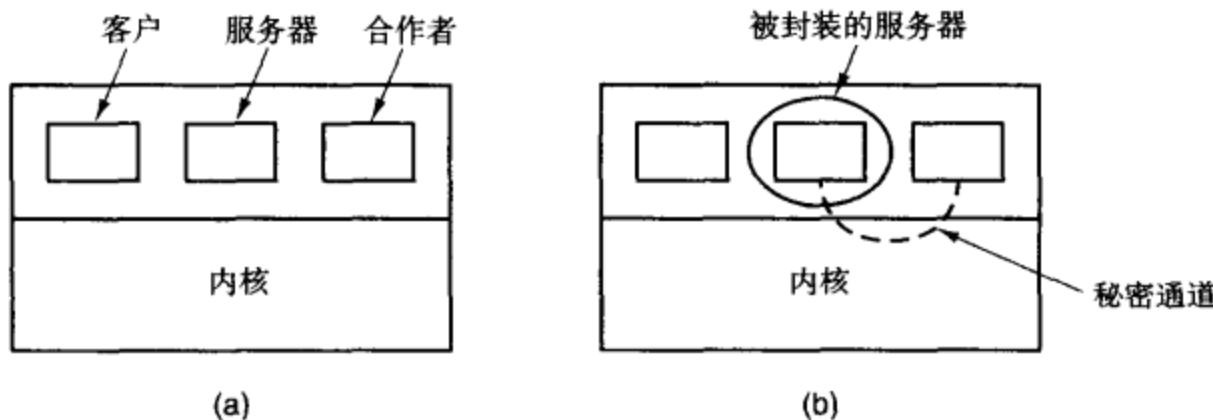


图 5.31 (a)客户、服务器和合作者进程; (b)被封装的服务
器仍然可以通过秘密通道把信息泄漏给合作者

从系统设计者的角度来说,我们的目标是要封装或约束服务器进程,使它无法向合作者进程传送信息。使用保护矩阵,我们可以很容易保证服务器进程与合作者进程之间不可能通过文件来进行通信,即服务器进程把数据写入一个文件,而合作者进程从该文件中读出数据。此外,我们还可以保证这两个进程不会使用通常的进程间通信机制来传递信息。

遗憾的是,这两个进程可以使用一些巧妙的通道来进行通信。例如,服务器进程可以用下列方式来传送一个二进制位流。当它要传送 1 时,就在固定长度的一段时间内,不停地进行运算;而当它要传送 0 时,就在相同长度的一段时间内,进入睡眠状态。

对于合作者进程来说,它可以通过仔细地监控自己的响应时间,来解析出服务器进程传来的位流。一般来说,当服务器进程在传送 0 时,合作者进程能够得到非常快的响应;而当服务器进程在传送 1 时,响应时间就比较长。这种通信通道称为秘密通道 (covert channel), 如图 5.31(b)所示。

当然,这个秘密通道是一个带有噪声的通道,包含有大量的无关信息。但即便如此,我们仍然可以通过使用纠错码(如 Hamming 码或更复杂的编码)来可靠地传递信息。纠错码的使用降低了秘密通道原本就很低的带宽,但这已经足以泄漏重要的信息。显然,任何基于对象和域矩阵的保护模型都无法防止此类泄漏。

调节 CPU 的使用强度并不是唯一的秘密通道,页面率也可以进行调节(很多缺页中断表示 1,无缺页中断表示 0)。事实上,任何一种定时去降低系统性能的做法都可以用做秘密通道。如果系统提供了一种对文件加锁的方法,那么服务器进程可以锁定某一个文件以表示 1,而释放锁表示 0。在有些系统中,进程能够检测出某个文件的加锁状态,尽管它可能没有权限去访问该文件。这个秘密通道如图 5.32 所示,文件被加锁或解锁某个固定长度的时间,这段时间的长度是由服务器进程和合作者进程事先约定好的。在这个例子中,秘密传送的二进制位流是 11010100。

加锁和解锁一个特定的文件时,S 这个秘密通道的噪声并不算太大,但它需要相当精准的计时工具,除非位速率非常慢。如果采用一种应答协议,那么信息传送的可靠性和性能就更高了。这个协议使用另外两个文件,F1 和 F2,分别由服务器进程和合作者进程来控制,用来协调各自的步骤。当服务器进程对文件 S 进行加锁或解锁之后,它会把文件 F1 的加锁状态翻过来,以表明一个数据位已经被发出。而当合作者进程收到该数据位后,会把文件 F2 的加锁状态翻过来,以此来告诉服务器进程,它准备接收下一个数据位。然后就在那里等待,直到 F1 的状态又发生了变化,表明又有一个数据位在 S 上。由于不再需要计算时间,这种协议非常可靠。为了获得更高的带宽,还可以同时使用两个文件来发送信息,或者使用 8 个文件,从 S0 到 S7,从而把通道的带宽变为 1 个字节。

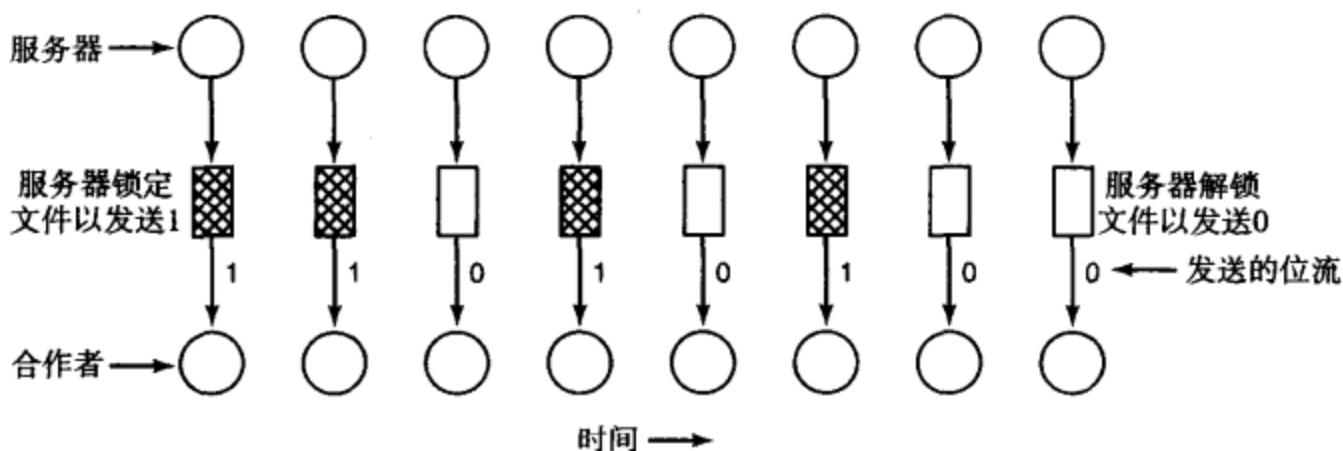


图 5.32 使用文件加锁机制的秘密通道

申请和释放专用的资源(磁带驱动器、绘图仪等)也可以用做秘密通道的信号。如果服务器进程要发送1, 就去申请该资源; 如果服务器进程要发送0, 就去释放该资源。在UNIX中, 服务器进程可以创建一个文件, 来表示传送1; 然后删除该文件, 表示传送0。合作者进程可以使用access系统调用来检测该文件是否存在。而且即使合作者进程没有访问该文件的权限, 也可以使用这个调用来查询。总之, 在系统中存在着许许多多的秘密通道。

Lamson还提到了一种向服务器进程的所有者泄漏信息的办法。假设服务器进程有权通知其所有者它为客户提供服务的工作量, 以便对客户计费。打个比方, 如果实际的账单为100美元, 而客户的收入为53 000美元, 则服务器进程可能会把100.53美元报告给其所有者。

试图找出所有的秘密通道是非常困难的, 更不用说去堵住它们了。实际上, 我们几乎毫无办法。有人提议引入一个新进程, 随机地产生一些缺页中断, 或者利用一些时间来降低系统的性能, 从而减少秘密通道的带宽, 这种想法也不太可取。

5.6 MINIX 3 文件系统概述

像所有的文件系统一样, MINIX 3文件系统也必须处理我们前面所讨论的这些问题。它必须为文件分配和释放空间、记录磁盘块和空闲空间、提供某种方式来防止文件被非法使用等。在本章的剩余部分, 我们将仔细研究MINIX 3的文件系统, 看它是如何来实现这些目标。

在本章的前半部分, 为了更具普遍性, 我们经常用UNIX而不是MINIX 3来作为例子, 虽然这两者的外部接口实质上是相同的。现在我们将把注意力集中到MINIX 3的内部设计上来。如果你想了解UNIX的内核, 请参阅Thompson (1978), Bach (1987), Lions (1996) 和Vahalia (1996)。

从本质上来说, MINIX 3文件系统只不过是运行在用户空间的一个大的C程序(请参考图2.29)。在读、写文件时, 用户进程向文件系统发送一条消息, 表明自己需要什么操作, 而文件系统会完成相应的工作, 并返回一个结果。从某种意义上来说, 文件系统实际上是一个网络文件服务器, 只不过它正好和调用进程在同一台机器上运行。

这种设计方式有深刻的含意。一方面, 文件系统可以独立于MINIX 3的其他部分进行修改、调试和测试。另一方面, 我们可以很方便地把整个文件系统移植到任何一台带有C编译器的计算机上, 在那里进行编译链接, 然后把它作为一个独立的类UNIX的远程文件服务器。我们唯一要做的修改就是消息的发送和接收方式, 因为不同的系统往往有不同的消息发送和接收方式。

在以下几个小节中, 我们将纵观文件系统设计中的许多关键问题, 尤其是消息、文件系统的布局、位图、i节点、块高速缓存、目录和路径、文件描述符、文件锁以及设备文件(包括管道)。在研究了上述内容之后, 我们将给出一个简单的例子, 通过跟踪一个用户进程执行read系统调用的过程, 来说明各个部分是如何来协同工作的。

5.6.1 消息

文件系统能处理39种消息请求。除两种以外，其余都用于MINIX 3的系统调用。这两个例外的消息是由MINIX 3的其他部分生成的。在用于系统调用的消息中，有31种来自于用户进程，而剩下的6种消息则用于一些特定的系统调用，这些调用首先被进程管理器所处理，然后再调用文件系统来完成部分工作。图5.33列出了文件系统所处理的各种消息。

来自于用户的消息	输入参数	返回值
access	文件名，访问模式	状态
chdir	新工作目录名	状态
chmod	文件名，新模式	状态
chown	文件名，新所有者，组	状态
chroot	新的根目录名	状态
close	要关闭的文件的文件描述符	状态
creat	要创建的文件的名称，模式	文件描述符
dup	文件描述符（对于dup2，两个fds）	新文件描述符
fcntl	文件描述符，功能码，arg	由功能决定
fstat	文件名，缓冲区	状态
ioctl	文件描述符，功能码，arg	状态
link	要链接到的文件名，链接名	状态
lseek	文件描述符，偏移量，从哪里开始	新位置
mkdir	文件名，模式	状态
mknod	目录或设备文件名，模式，地址	状态
mount	设备文件，挂装点，只读标记	状态
open	要打开的文件名称，读写标志位	文件描述符
pipe	指向两个文件描述符的指针（已修改）	状态
read	文件描述符，缓冲区、字节数	读入的字节数
rename	文件名，文件名	状态
rmdir	文件名	状态
stat	文件名，状态缓冲区	状态
stime	指向当前时间的指针	状态
sync	(无)	总为OK
time	指向当前时间的指针	状态
times	指向父进程和子进程时间值缓冲区的指针	状态
umask	模式掩码的补码	总为OK
umount	要卸载的设备文件名	状态
unlink	要解链的文件名	状态
utime	文件名，文件时间	总为OK
write	文件描述符，缓冲区，字节数	写入的字节数
来自于PM的消息	输入参数	返回值
exec	PID	状态
exit	PID	状态
fork	父进程PID，子进程PID	状态
setgid	PID，真实和有效GID	状态
setsid	PID	状态
setuid	PID，真实和有效UID	状态
其他消息	输入参数	返回值
revive	要唤醒的进程	(无响应)
unpause	要检查的进程	(参见正文)

图5.33 文件系统的消息。文件名参数指的是指向文件名的指针。返回值中的状态表示OK或ERROR

文件系统的结构基本上与进程管理器和所有的I/O设备驱动程序是一样的。它有一个主循环，在不断地等待消息的到来。当一个消息到达后，首先抽取它的类型，并以其为索引去查找一张指针表格，每个指针指向的是不同类型的消息处理函数。随后调用相应的函数进行处理，并返回一个状态值。文件系统再把应答消息发送给调用进程，然后回到主循环的开始，等待下一条消息的到来。

5.6.2 文件系统的布局

MINIX 3文件系统是一个自成体系的逻辑实体，含有*i*节点、目录和数据块。它可以存储在任何一个块设备中，如软盘或一个硬盘分区。在各种情形下，文件系统的布局具有相同的结构。图5.34显示了一张软盘或一个小的硬盘分区上的文件系统布局，它带有64个*i*节点和1 KB的块大小。在这个简单的系统中，位图区域仅仅是一个1 KB的块，因此它最多能管理8192个1 KB的块，也就是说，文件系统最大不能超过8 MB。即使是在软盘上，64个*i*节点也是远远不够的，它过于苛刻地限定了文件的个数。因此，图中虽然只为*i*节点保留了4个块，但实际上会用到更多。可能保留8个块是比较合适的，但如果在图中画上8个块，则不太好看。对于一个现代硬盘，*i*节点和位图区域肯定不只有一个块。图5.34中各个部分的相对大小取决于具体的系统，包括系统的规模、支持的最大文件个数等。但不管怎样，所有的这几个部分都会存在，而且它们之间的顺序也是相同的。

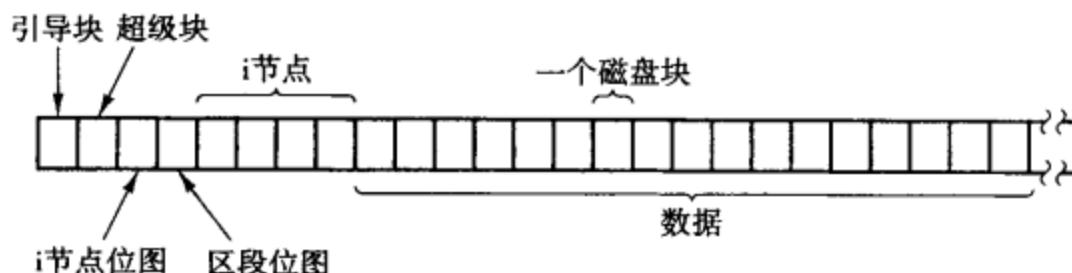


图 5.34 一张软盘或一个小的硬盘分区上的磁盘布局，带有64个*i*节点和1 KB的块大小（即把两个连续的512字节扇区视为一个块）

每个文件系统都以一个**引导块** (boot block) 开始，其中包含有可执行代码。引导块的大小总是1024字节（两个磁盘扇区），而在MINIX 3的其他地方，可能会使用更大一些的块。在启动计算机时，硬件将会从引导设备把引导块读入内存，并转而执行其代码。而引导块代码则负责操作系统本身的加载工作。一旦系统被启动之后，引导块就不再使用。并非每个磁盘驱动器均可用做引导设备，但为了保持结构上的一致性，每个块设备都会为引导块代码预留一个块。这种方法最多不过是浪费了一个块。为了防止硬件从非引导设备上启动，在将可执行代码写入引导设备的引导块中时，会把一个**魔数** (magic number) 写入到引导块的固定位置。这样，当系统从一个设备上启动时，硬件（实际上是BIOS代码）会首先检查魔数是否存在。若不存在，则拒绝把该设备的引导块载入内存，这样可以防止把垃圾数据误认为是引导程序。

超级块 (superblock) 中包含的信息描述了文件系统的布局。和引导块一样，超级块的大小也是固定的1024字节，如图5.35所示。

超级块的主要功能是给出文件系统各个部分的大小。如果给定块大小和*i*节点数，我们可以很容易地算出*i*节点位图的大小和存放*i*节点所需的块数。例如，假设块大小为1 KB，每个位图块有1024字节（8192位），可以记录8192个*i*节点的状态（实际上第一块只能处理8191个*i*节点，因为第0号*i*节点并不存在，但我们在位图中也会为它保留一位）。如果有10 000个*i*节点，则需要用到两个位图块。由于每个*i*节点要占用64个字节，因此1 KB的块最多能存放16个*i*节点。如果有64个*i*节点，则需要4个磁盘块来存放。

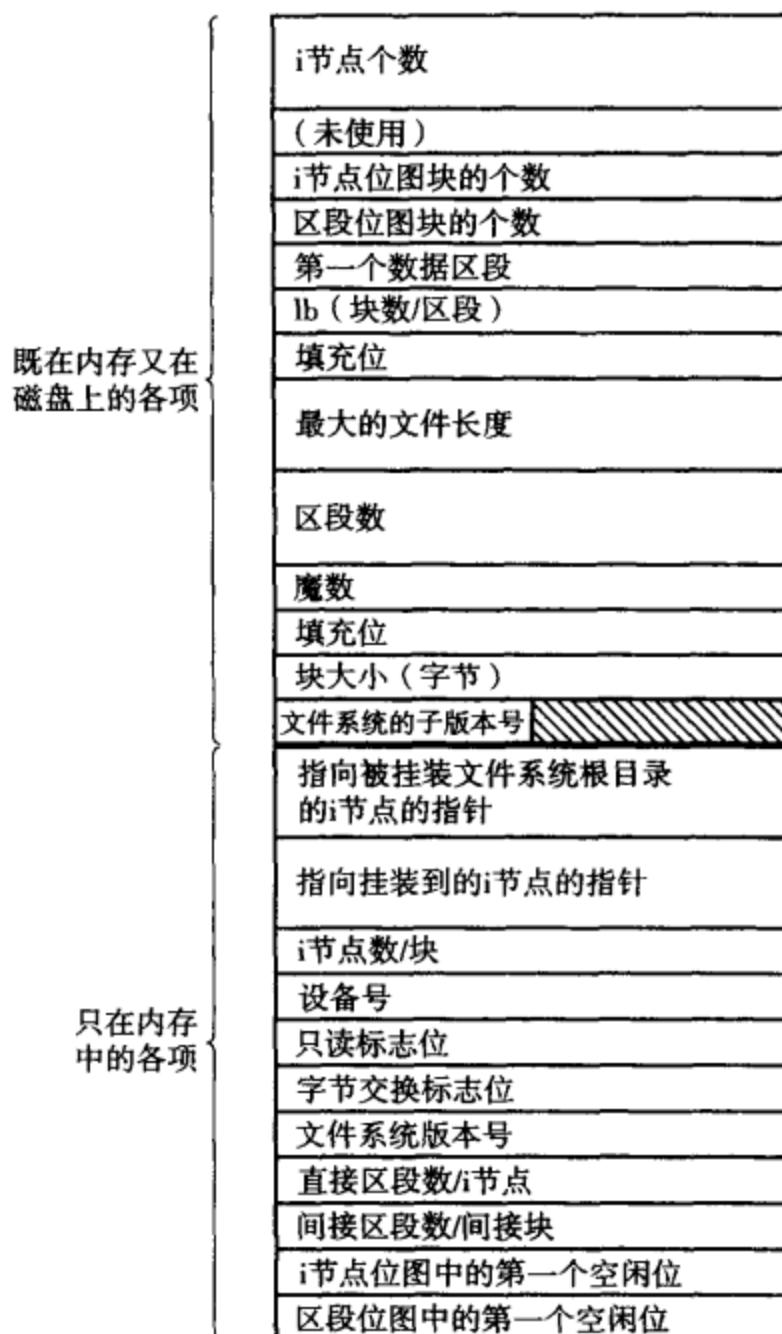


图 5.35 MINIX 3 超级块

后面我们会详细地解释区段和块之间的区别。但是此时，读者只要知道磁盘存储可以以区段为单位来进行分配，每个区段包含 1, 2, 4, 8 个或 2^n 个磁盘块。区段位图按区段而不是块来管理空闲存储区。对于 MINIX 3 所用到的所有标准盘，区段的大小和块大小是一样的（默认为 4 KB），因此在这些设备上，可以近似地把区段看成是块。在本章后面详细讨论存储分配之前，当你在看到“区段”时，只要把它当做“块”就可以了。

请注意，每个区段所包含的块数并没有存放在超级块中，因为它没有什么用处。我们存放的是区段数除以块数的以 2 为底的对数值。根据它，可以知道从区段转换成块或者从块转换成区段需要移动的位数。例如，假设每个区段包含有 8 个块， $\log_2 8 = 3$ ，因此，如果要寻找第 128 个块所在的区段，可以把 128 右移 3 位，得到区段 16。

区段位图中只包含数据区段（即用于位图和 i 节点的块不包含在该位图中），而且第一个数据区段在位图中用区段 1 来表示。同 i 节点位图一样，区段位图中的第 0 位也未使用，因此它的第一个块只能映射到 8191 个区段，而以后的每块可以映射到 8192 个区段。如果考察一个新格式化的磁盘的位图，可以发现 i 节点和区段位图中均有两位被置为 1。一位用于不存在的第 0 号 i 节点或第 0 号区段，而另一位用于设备根目录的 i 节点和区段，当文件系统被创建时，根目录就会自动存在。

另外，我们可以注意到超级块中有些信息是冗余的，这主要是因为在不同的时候，需要用到该信息的不同形式。由于有 1 KB 的空间可用于存储超级块，因此对于同一个信息，我们可以预先计算出它的不同形式，然后把它们全部存放在超级块中，这样就不必在系统运行时重复计算了。例如，磁盘上的第一个数据区段的区段号，可以从块大小、区段大小、i 节点个数和区段个数中计算出来。但是，如果把它直接存放在超级块中，则用起来更加方便。超级块中的剩余部分总归是要浪费的，还不如用它来存储一些有用的数据。

在 MINIX 3 启动时，根设备中的超级块会被读入内存。同样，当挂装其他的文件系统时，它们的超级块也会被读入内存。在内存的超级块表中，有些字段是磁盘上的超级块所没有的，如设备的只读标志位和字节顺序标志位、位图中第一个空闲位的位置（用于加速位图的访问）以及该超级块来自于哪一个设备等。

一块磁盘如果想用做 MINIX 3 的文件系统，它必须具有图 5.34 所示的结构。工具程序 *mkfs* 可以用来创建一个文件系统。它的一种用法是直接在命令行中调用，如

```
mkfs/dev/fd1 1440
```

该命令将在驱动器 1 中的软盘上，创建一个空的 1440 个块的文件系统。*mkfs* 的另一种用法是给它一个原型文件，里面列出了新文件系统所包含的目录和文件。该命令同时还会把一个魔数存放在超级块中，以表明这个文件系统是一个有效的 MINIX 文件系统。由于 MINIX 文件系统在不断向前发展，它的有些方面（如 i 节点的大小）与以前是不同的。因此，这个魔数会标明 *mkfs* 的版本号，这样，系统就能够针对不同的版本进行相应的调整。如果试图去安装一个非 MINIX 3 格式的文件系统，如 MS-DOS 磁盘，则会被 *mount* 系统调用拒绝，它会去检查超级块中的魔数等字段，以判断一个磁盘是否是一个有效的 MINIX 文件系统。

5.6.3 位图

MINIX 3 使用两个位图来管理空闲的 i 节点和区段。当一个文件被删除时，很容易算出该文件的 i 节点所在的位图块，并利用通常的高速缓存机制来查找该块。一旦找到，就把对应于该 i 节点的那一位清零。区段的释放过程与此类似。

从逻辑上来说，在创建一个文件时，文件系统必须逐一地去检查每个位图块，以找到第一个空闲的 i 节点，然后把它分配给这个新创建的文件。但实际上，为了加快查找的速度，在内存的超级块表中，有一个字段直接指向第一个空闲的 i 节点，因此无须进行查找。当然，在这个空闲节点被分配后，需要修改相应的指针，使它指向下一个空闲 i 节点，这个新节点往往就是下一个节点或离得比较近的一个节点。另外，当一个 i 节点被释放后，要去检查一下，看这个 i 节点是否位于当前所指向的空闲节点的前面，若是，则要修改该指针。如果磁盘上的所有 i 节点都已被占用，查找函数将返回 0，这也是为什么 0 号 i 节点未被使用的原因（也就是说，用 0 来表示查找失败）（当 *mkfs* 在创建一个新的文件系统时，它会把 0 号 i 节点清零，并把位图中的最低位设置为 1，这样文件系统就不会把 0 号 i 节点分配出去）。另外，上面所说的内容同样也适用于区段位图。在申请空间时，原本需要在区段位图中查找第一个空闲区段，但我们有一个指针直接指向该空闲区段，因此就不必去顺序查找。

有了这些背景知识之后，我们可以来解释一下区段和块之间的不同。使用区段的目的是，确保同一个文件的磁盘块位于同一个柱面上，从而提高文件顺序读取时的性能。这里采用的方法是一次分配多个块。例如，假设块大小为 1 KB，区段大小为 4 KB，区段位图中记录的是区段而非块的使用情况。如果一个磁盘的容量为 20 MB，那么它有 5K 个 4 KB 大小的区段，因此在它的区段位图中有 5K 个位。

在大多数情形下，文件系统是以块为单位来进行操作的。磁盘数据的传输以块为单位，高速缓存区也以块为单位。在系统中只有记录物理磁盘地址的一小部分内容（如区段位图和*i*节点）需要知道区段的存在。

在开发 MINIX 3 文件系统的过程中，我们不得不做出一些设计上的决策。1985 年，当 MINIX 还处于构思阶段的时候，磁盘容量很小，许多用户只有软盘。因此，在 V1 文件系统中，我们决定把磁盘地址限制在 16 位，这样就能把大多数地址存放在间接块中。16 位的区段号和 1 KB 大小的区段只能寻址 64K 个区段，从而将磁盘容量限制为 64 MB。在当时，这可是一个相当大的容量。而且我们当时还考虑到，当磁盘容量增大时，只要把区段的大小变为 2 KB 或 4 KB 即可，而块的大小不用动。此外，16 位的区段号还使得*i*节点的大小很容易保持为 32 字节。

随着 MINIX 的不断发展，以及大容量磁盘的日益普及，显然有必要对文件系统进行改造。由于许多文件的长度小于 1 KB，因此增加块的大小就意味着浪费磁盘的带宽和读写几乎为空的块，而且当它们被存放在高速缓存时，也浪费了宝贵的内存空间。我们原本打算增加区段的大小，但区段越大，浪费的磁盘空间越多，何况我们还希望保持对小磁盘操作的高效率。另一种可能的解决方案是在不同大小的设备中使用不同大小的区段。

但最终我们还是决定把磁盘指针的大小增加到 32 位。这样，在 MINIX V2 文件系统中，能够处理容量高达 4 TB 的磁盘设备（块和区段大小均为 1 KB），或者是 16 TB 的设备（块和区段大小均为 4 KB）。不过，其他的一些因素也制约了这个大小。例如，如果采用 32 位的指针，则能够访问的地址空间为 4 GB。另外，指针长度的增加势必导致*i*节点大小的增加。这也许不一定是一件坏事情，因为这意味着 MINIX V2（现在是 V3）的*i*节点和标准的 UNIX 的*i*节点是兼容的。由于*i*节点的容量增加了，可以存放更多的间接区段或二级间接区段，而且便于将来的进一步扩展。

区段的使用会带来一个预料不到的问题。我们用一个简单的例子来阐述。假设区段的大小为 4 KB，块的大小为 1 KB。有一个文件，它的长度为 1 KB，这意味着已经给它分配了一个区段，而且在该区段中，除了第一个块以外，其余的三个块中都含有垃圾（以前用户的残留数据）。这种情形并不会对文件系统造成不利，因为在该文件的*i*节点中，文件的大小被清楚地标记为 1 KB。事实上，包含有垃圾数据的磁盘块根本不会被读入高速缓存，因为读操作是以块而不是区段为单位的。超出文件末尾的读操作总是返回 0，不包含任何数据。

假设现在有人将文件指针定位在 32 768 并写入 1 个字节，因此文件的长度变为 32 769 字节。随后他就可以把指针移动到 1024，从而去访问后面的内容，即上一个用户遗留下来的数据。这样，就造成了一个很大的安全隐患。

解决这一问题的方法是，在执行写操作时，检查写入的位置是否超出了文件的末尾。如果是，则将文件的最后一个区段中还未分配出去的块全部清零。尽管这种情况很少发生，但我们的代码必须进行处理，这可能会稍微增加系统的复杂性。

5.6.4 *i* 节点

MINIX 3 中*i*节点的布局如图 5.36 所示，它与标准 UNIX 的*i*节点几乎是一样的。磁盘区段指针是 32 位的，总共有 9 个这样的指针：7 个直接的，2 个间接的。MINIX 3 的*i*节点占用 64 个字节，这也同标准 UNIX 的*i*节点是一样的。同时，还预留了一个未使用的空间用于将来的第 10 个（三次间接）指针，在目前的标准版本的文件系统中还不支持这一指针。MINIX 3 的*i*节点中的访问时间、修改时间都和标准的 UNIX 一样。除了读操作，其余的文件操作都会导致*i*节点的修改时间发生变化。

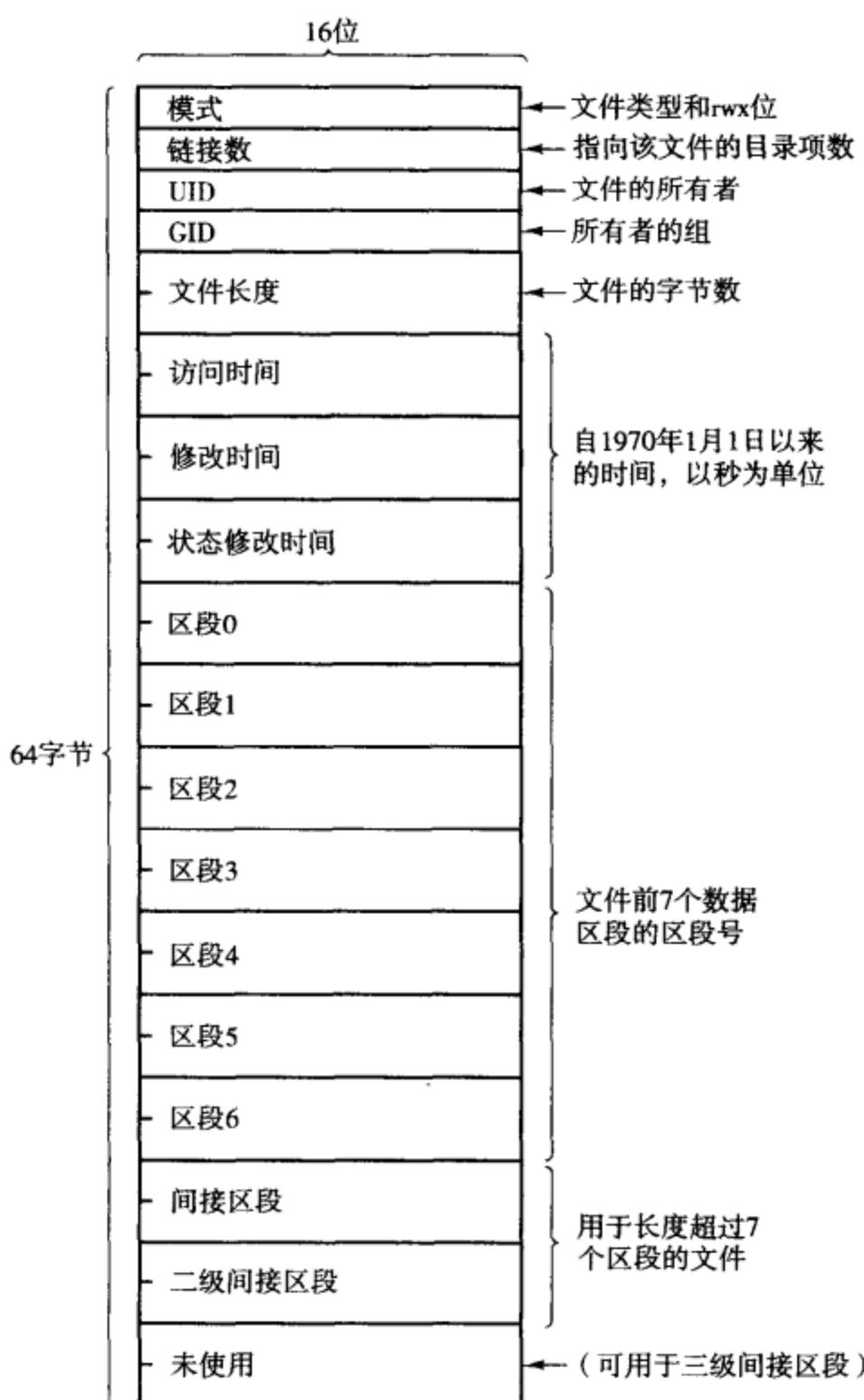


图 5.36 MINIX 的 i 节点

在打开一个文件时，先要找到它的 i 节点，并把它装入到内存的 *inode* 表中，而且它会一直呆在那儿，直至文件被关闭。在内存的 *inode* 表中有一些字段是磁盘 i 节点所没有的，如 i 节点所在的设备。这样，当该 i 节点被修改后，文件系统才知道应该把它写回到什么地方。每个 i 节点还有一个计数器，如果同一个文件被多次打开，那么在内存中只保存一份 i 节点副本。但每次打开该文件时，计数器加 1；每次关闭该文件时，计数器减 1。只有在计数器减到 0 时，才会将 i 节点从表格中删除。如果它在内存期间曾经被修改过，则还要把它写回到磁盘。

文件 i 节点的主要功能是给出文件数据块所在的位置。前 7 个区段号就直接存放在 i 节点之中。对于 MINIX 标准发行版，区段大小和块大小均为 1 KB，因此小于 7 KB 的文件不必使用间接块。如果文件的长度超过 7 KB，就要使用间接区段。MINIX 采用了图 5.10 所示的方案，但只用到了其中的一级间接块和二级间接块。如果块大小和区段大小均为 1 KB，区段号为 32 位，则一级间接块含有 256 项，可以表示 1/4 MB 的存储区。二级间接块指向 256 个一级间接块，因此可以访问长达 64 MB 的文件。如果块的大小为 4 KB，那么二级间接块将指向 1024×1024 个块，也就是上百万

个4 KB的块，因此最大的文件长度为4 GB。实际上，由于使用了32位的整数来作为文件偏移量，因此文件的最大长度为 $2^{32}-1$ 字节。这些数字表明，如果在MINIX 3中使用4 KB大小的磁盘块，那么根本就不需要三级间接块。因为文件的最大长度取决于指针的大小，而不是能否管理足够的块。

在i节点中还包含有模式信息，它给出了文件的类型（普通文件、目录、块设备文件、字符设备文件或管道）以及保护标志位、SETUID位和SETGID位。i节点中的link字段记录了有多少个目录项正在指向这个i节点，这样文件系统就知道何时去释放该文件的存储空间。注意不要把这个字段与打开文件计数器（只出现在内存的inode表中，磁盘的i节点中没有）相混淆，后者描述的是文件被打开的次数，即它正被多少个进程所访问。

关于i节点，我们最后再说一点。图5.36的结构可能会因为特殊的目的而被修改，一个例子就是MINIX 3中的块设备文件和字符设备文件的i节点。在这些i节点中，区段指针是没有必要的，因为它们不需要去访问磁盘上的数据区域。而它们需要的主设备号和从设备号则被存放在图5.36中的区段0的位置。i节点的另外一种使用方式是：用来实现立即文件，即对于那些非常小的文件，可以把它们的数据直接存储在i节点中。不过在MINIX 3中，暂时还不支持这一特性。

5.6.5 块高速缓存

MINIX 3使用块高速缓存来改进文件系统的性能。高速缓存用一个固定长度的缓冲区数组来实现，每个缓冲区由头和体两部分组成，头包含了指针、计数器和标志位，体包含了一个磁盘块大小的空间。所有未使用的缓冲区被链接在一条双向链表中，按最近一次使用时间的先后顺序排列，如图5.37所示。

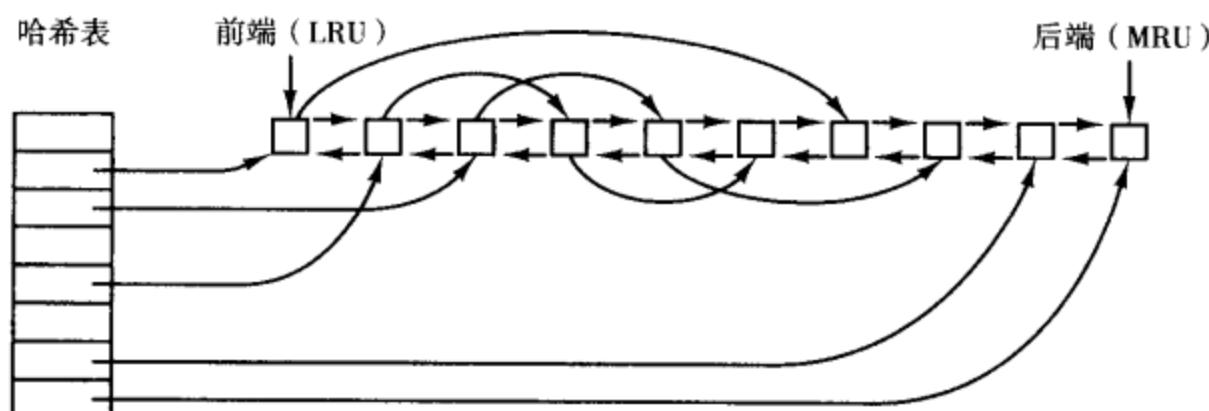


图 5.37 块高速缓存使用的链表

为了快速判断某个块是否在内存中，我们使用了哈希表。如果有多个缓冲区，它们所包含的块具有相同的哈希代码k，那么它们将被一条单向链表链接在一起，而哈希表中的第k项将指向该链表。哈希函数从块号中提取最低的n位来作为哈希代码，因此来自于不同设备的块可以出现在同一条哈希链之中。每个缓冲区都位于其中的某条链表中。MINIX 3启动后，在初始化文件系统时，所有的缓冲区均未使用，并且全部位于第0个哈希表项所指向的链表中。这时，其他的哈希表项均包含一个空指针。但当系统启动后，缓冲区将从0号链中删除，放到其他链表中。

当文件系统需要一个块时，将调用函数get_block，去计算该块的哈希代码，并在相应的链表中搜索。get_block的调用参数有两个：设备号和块号。这两个值将与缓冲区链表中的对应字段相比较，如果找到了包含这一块的缓冲区，则把缓冲区头中的计数器加1，表明该块正在被使用，然后返回一个指向它的指针。如果在哈希表中未找到这个块，可以使用LRU链中的第一个缓冲区，该缓冲区肯定未在使用，因此可以把它所包含的块置换出内存，以释放这个缓冲区。

如果选定了某个块，要将它调出内存，这时要去检查缓冲区头中的另一个标志位，看它在内存期间是否曾经被修改过。如果是，就要把它重新写回磁盘，然后，文件系统向磁盘驱动程序发送一条消息，要求读入新的块。之后文件系统将被挂起，直到该块被读入后才继续往下运行，把指向该块的指针返回给调用进程。

当请求一个数据块的函数完成它的任务后，会调用另一个函数 *put_block* 来释放这个块。在正常情形下，一个块在读入后会立即使用并释放。但由于该块可能会被其他进程所访问，所以 *put_block* 仅仅是把它的使用计数器减 1。当使用计数器减到 0 时，才会把它放到 LRU 链表中。否则，我们就让它保留在那里。

在 *put_block* 的参数中，有一个是被释放块的类型（如 i 节点、目录或数据）。对于不同的类型，需要做出不同的决策：

1. 应该把该块放在 LRU 链表的前端还是尾端？
2. 如果该块被修改过，是否需要把它立即写回磁盘？

在真正的 LRU 模式下，大多数的块都被送到链表的末尾，唯一的例外是来自于 RAM 盘的块，由于它们已经在内存中，所以把它们保存在高速缓存中并没有什么益处。

被修改的块只在以下两种情况下会被写回磁盘：

1. 它到达了 LRU 链表的开头并被换出。
2. 执行了 sync 系统调用。

sync 并不遍历 LRU 链表，相反，它去查找高速缓存中的缓冲区数组。对于每一个缓冲区，即使它并未被释放，只要它被修改过，*sync* 就会找到它，并更新它在磁盘上的副本。

但是，有一种情况比较特殊，需要把修改过的超级块立即写回磁盘。在早期的 MINIX 版本中，在挂装一个文件系统时要去修改超级块，这时，我们要把修改后的超级块信息立即写回磁盘，以减少在系统崩溃时文件系统被破坏的可能性。而在现在的 MINIX 版本中，超级块的修改只有一种情形，即 RAM 盘的大小需要在启动时进行调整。不过，超级块的读写操作与普通的块是不同的，因为它和引导块一样，大小总是 1024 个字节，这个大小与高速缓存中的块大小无关。另外一个过时的内容是在 MINIX 的早期版本中，有一个 *ROBUST* 宏，它可以在系统配置文件 *include/minix/config.h* 中定义。如果这个宏被定义，然后重新编译文件系统，就可以实现 i 节点、目录、间接块和位图块在释放时立即被写回磁盘。这种做法可以使文件系统变得更加健壮，但付出的代价就是运行速度下降。而且它的效果也并不太明显，事实上，如果发生断电事故，那么不管丢失的是 i 节点块还是数据块，都是一件使人头疼的事。

请注意，用于表明一个块是否被修改的标志位，是由文件系统中请求和使用该块的函数来设置的。*get_block* 和 *put_block* 函数只关心对链表的操作，它们并不知道文件系统的哪个函数需要哪一个块，也不知道为什么需要。

5.6.6 目录和路径

文件系统的另一个重要内容是目录和路径名的管理。许多系统调用，如 *open*，都以文件名来作为参数。但实际需要的是该文件的 i 节点，因此，文件系统需要在目录树中查找这个文件，找到相应的 i 节点。

一个 MINIX 目录实际上就是一个文件。在早期版本中，每个目录项的长度为 16 个字节，其中 2 个字节用于 i 节点号，剩下的 14 个字节用于文件名。这种设计把磁盘分区限定为 64K 个文件，且

文件名的长度不超过 14 个字符，这和 UNIX V7 版本是相同的。随着磁盘容量的增长，文件名的长度也在增长。MINIX 3 的 V3 文件系统提供了 64 个字节的目录项，其中 4 个字节用来存放 i 节点号，60 个字节用来存放文件名。这样，每个磁盘分区能够支持的文件个数是 40 亿，这应该足够用了。至于文件名，如果一个程序员选择了一个长度超过 60 个字符的文件名，那么他应该被送回学校继续学习。

请注意路径名的长度并没有 60 个字符的限制，例如

/usr/ast/course_material_for_this_year/operating_systems/examination-1.ps

这个路径名的长度远远超过了 60，但它是合法的，60 个字符的限制只是针对单个的文件名。我们这里使用了固定长度的目录项，即 64 个字节，这是综合考虑了简单性、速度和存储空间等因素后的一个折中结果。其他的一些操作系统，把目录组织成一个堆的形式，每个文件有一个固定长度的文件头，指向堆中的一个名字。MINIX 3 的方案非常简单，需要的代码与 V2 版本基本上是相同的。而且在文件名的查找和新文件名的存储上，速度非常快，因为这里不需要任何堆管理操作。唯一的代价就是会浪费一些磁盘空间，因为大多数的文件名都远远短于 60 个字符。

我们一直有一个观点，即最优化磁盘空间的使用是一种错误的选择，而代码的简单性和正确性应该排在第一，其次是运行速度。现代磁盘的容量通常都超过 100 GB，如果为了节省一小部分磁盘空间而使代码变得又慢又复杂，这不是一个好主意。遗憾的是，许多程序员是从微小磁盘和更微小的内存的登录成长起来的，从第 1 天起他们就被要求在对代码复杂性、速度和空间进行折中时，要倾向于如何尽量减少磁盘空间。但是在当前的现实条件下，真的需要对这种倾向性重新进行审查。

下面，我们来看一下路径名 */usr/ast/mbox* 是如何查找的。文件系统首先在根目录中查找 *usr*，然后在 */usr/* 目录中查找 *ast*，最后在 */usr/ast/* 目录下查找 *mbox*。在整个查找过程中，每次只查找其中的一个目录名，如图 5.16 所示。

唯一的麻烦是遇到被挂装的文件系统。在 MINIX 3 和其他类 UNIX 系统中，通常配置有一个小的根文件系统，里面含有一些用于启动系统和进行基本维护的文件。此外，该配置还能使存放在另一个设备上的很多文件（包括用户的目录）被挂装至 */usr* 目录下。现在我们可以来介绍一下挂装是如何实现的。当用户在终端上键入命令

```
mount /dev/c0d1p2 /usr
```

时，存放在硬盘 1 的第 2 个分区上的文件系统将被挂装到根文件系统的 */usr/* 目录下。挂装前后的文件系统如图 5.38 所示。

挂装成功后，在 */usr* 目录的 i 节点的内存副本中设置了一个标志位，表明这个 i 节点已经被挂装，这就是关键所在。*mount* 调用还会把新挂装的文件系统的超级块调入内存中的 *super_block* 表中，并设置其中的两个指针。此外，它还把被挂装文件系统的根目录的 i 节点也放在内存的 *inode* 表中。

在图 5.35 中，我们看到，内存中的超级块有两个字段与被挂装的文件系统有关。第一个指向新挂装的文件系统的根目录 i 节点；第二个指向挂装的目标 i 节点，在这里是 */usr* 目录的 i 节点。这两个指针把被挂装的文件系统和根文件系统链接起来 [如图 5.38(c) 中虚线所示]，从而使被挂装的文件系统能够正常工作。

在查找路径 */usr/ast/f2* 时，文件系统会在 */usr/* 目录的 i 节点中发现一个标志位，于是知道它需要到挂装在 */usr/* 的文件系统的根目录 i 节点上继续查找。问题在于，它如何才能找到这个 i 节点呢？

答案很简单，系统会去搜索内存中的所有超级块，直至找到某一个超级块，它的“指向挂装到的 i 节点”字段指向的是 */usr/*，这就是我们要找的那个文件系统的超级块。然后，就可以沿着它的

另一个指针找到该文件系统的根目录 i 节点，并继续往下查找。在这个例子中，文件系统会在硬盘第 2 个分区的根目录下继续查找 *ast* 目录。

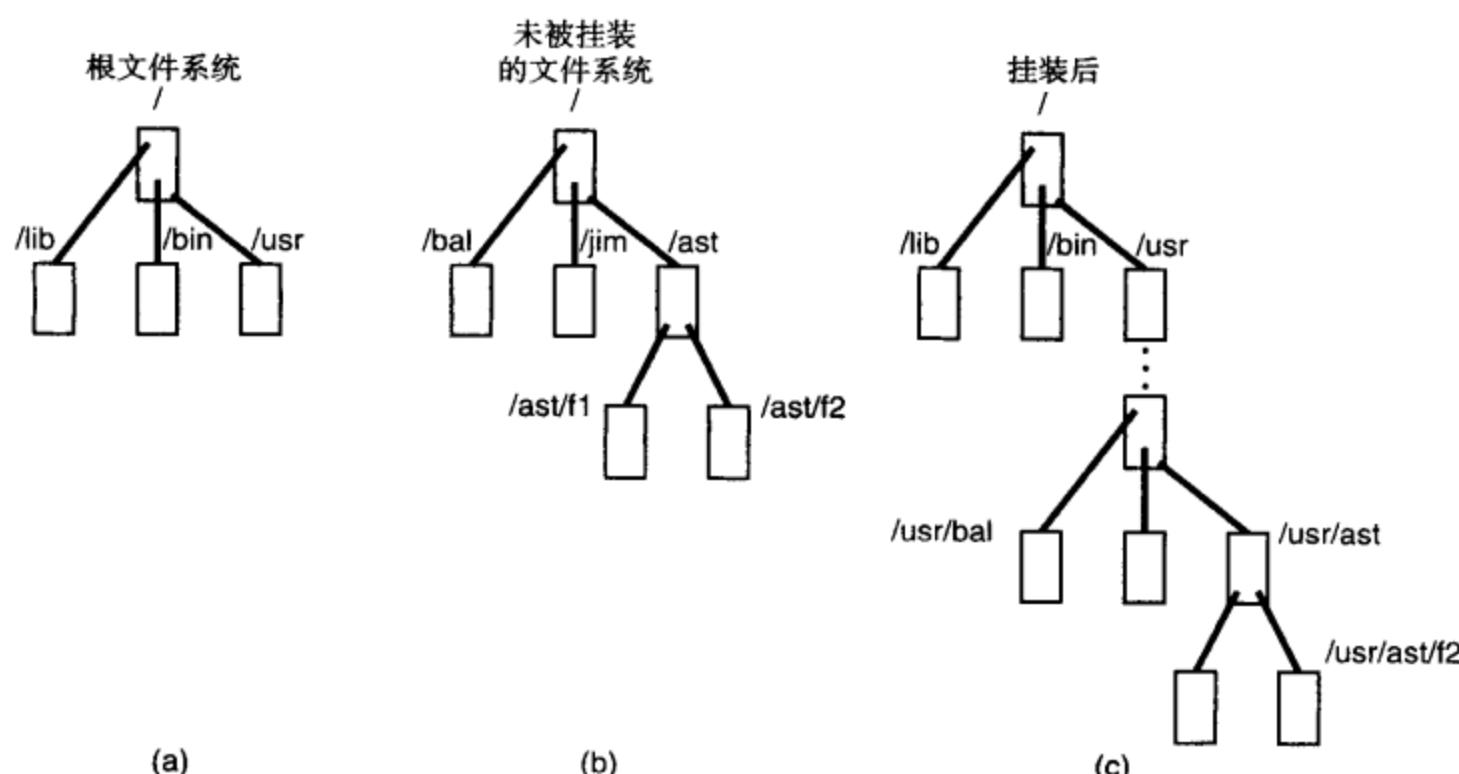


图 5.38 (a)根文件系统; (b)未被挂装的文件系统; (c)将(b)中的文件系统挂装到目录 /usr/ 后的效果

5.6.7 文件描述符

当一个文件被打开时，就会把一个文件描述符返回给用户进程，用于后续的 `read` 和 `write` 调用。在这一小节，我们将介绍文件系统是如何来管理文件描述符的。

与内核和进程管理器一样，文件系统也会在它的地址空间中维护进程表的部分内容。其中有三个字段需要特别注意：前两个是分别指向根目录 i 节点和工作目录 i 节点的指针。像图 5.16 中的路径名查找总是从其中的一个目录开始那样，这取决于该路径是绝对路径还是相对路径。执行 `chroot` 和 `chdir` 系统调用可以改变这两个指针，使它们指向新的根目录或新的工作目录。

第三个有意思的数据结构是一个数组，以文件描述符编号为索引。当给定一个文件描述符时，可以通过该数组找到相应的文件。乍一看，似乎只要把这个数组的第 *k* 个项指向描述符为 *k* 的那个文件的 i 节点即可。毕竟，在文件打开时，就会把它的 i 节点调入内存，并一直保存在那直至文件关闭，所以说，这种方法应该能奏效。

遗憾的是，这种简便的方法并不可行。因为在 MINIX 3（以及 UNIX）中，文件的共享方式比较考究。每个文件都有一个 32 位的整数来描述即将读写的下一个字节的位置，这个整数称为文件位置（file position），它是通过 `Iseek` 系统调用来修改的。我们的问题是：“这个文件指针应该存放在什么地方？”

第一种可能是把它放在 i 节点中。然而，如果两个或多个进程都打开了同一个文件，则它们必须拥有各自独立的文件指针，一个进程的 `Iseek` 操作不应该影响到另一个进程的读操作。因此，文件的当前位置不能存放在 i 节点中。

那么能否把它存放在进程表中呢？为什么我们不创建另一个数组，平行于文件描述符数组，然后用它来存放文件的当前位置呢？这个想法也不可取，但原因更加微妙。问题主要来自于 `fork` 系统调用的语义。当一个进程调用 `fork` 来创建一个新进程时，父进程和子进程必须共享同一个指针，指向各自文件中的当前位置。

为了更好地理解这个问题，我们来考虑一个 shell 脚本的例子，该脚本的输出已经被重定向到某个文件。当 shell 生成第一个子进程时，其标准输出文件的当前位置为 0。然后，子进程将继承这个位置，并输出 1 KB 的数据。在子进程运行结束后，共享的文件位置应该为 1024。

接下来，shell 读取该脚本的更多内容，并生成第二个子进程。这个子进程将从 shell 中继承 1024 的文件位置，这样，它就会从第一个子进程的输出结果后面继续往下写。而如果 shell 不和子进程共享文件位置，那么第二个子进程的输出内容很可能会覆盖第一个子进程的输出，而不是在它后面追加数据。

所以说，我们也不能把文件的当前位置存放在进程表中，因为它必须实现共享。在 UNIX 和 MINIX 3 中，这个问题的解决是通过引入一个新的共享表 *filp* 来实现的，该表中包含了所有的文件位置，如图 5.39 所示。通过实现文件位置的真正共享，*fork* 的语义能够正确地实现，而 shell 脚本也能正常工作。

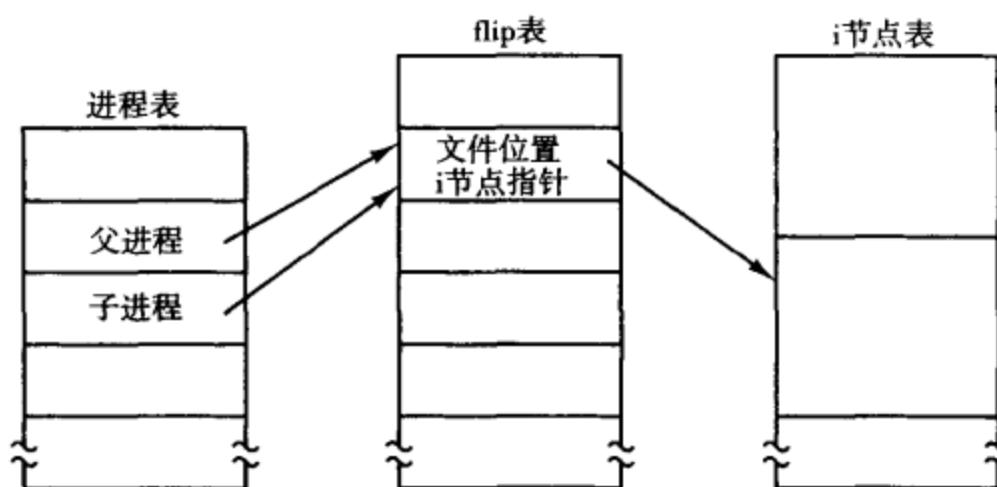


图 5.39 父进程与子进程共享文件位置

尽管 *filp* 表中真正必须包含的内容是共享文件的位置，但我们也可以把文件的 i 节点指针放在其中。这样，进程表中的文件描述符数组只需包含指向 *filp* 表项的指针。每个 *filp* 表项中还包含有文件的模式（权限位）、特殊模式标志位（表明该文件是否以某种特殊模式打开）以及打开文件计数器（有多少个进程正在使用该文件）。当这个计数器的值为 0 时，文件系统知道已经没人在使用这个表项了，因此就可以回收它。

5.6.8 文件锁

在文件系统的管理中，还有一部分内容需要使用专门的表结构，这就是文件锁。MINIX 3 支持 POSIX 进程间通信机制中的建议文件锁（advisory file locking），允许将文件的任何部分或多个部分标记为已锁定。锁操作并不是由操作系统强制执行的，而是进程自身的一种自觉行为，它在执行某个可能会与其他进程相冲突的操作之前，先要对文件锁进行查询。

为加锁机制提供一个单独的表结构，其原因类似于上一节中讲到的 *filp* 表。一个进程可以设置多个锁，而一个文件的不同部分也可以被不同的进程上锁（当然，锁与锁之间不能重叠）。因此进程表和 *filp* 表均不适合于记录文件锁。此外，由于在一个文件中可能包含多个锁，因此 i 节点也不宜用来记录锁。

MINIX 3 使用另外一张表，即 *file_lock* 表，来记录所有的锁。每个表项的内容包括：锁类型，表明对该文件加的是读锁还是写锁；加锁进程的 ID；指向被锁文件 i 节点的指针；以及标明加锁范围的起始字节和末尾字节的位置。

5.6.9 管道和设备文件

管道和设备文件与普通文件有很大的不同。当一个进程读写一个磁盘文件的某个数据块时，该操作最多在几百个毫秒内就能完成。即使在最坏情形下，也不过需要2到3次磁盘访问。而在读取一个管道时，情况就不同了：如果管道为空，则读进程可能会被阻塞，直到另一个进程把数据写入管道，而这可能需要几个小时。类似地，当从终端上读取数据时，进程也必须一直等待，直到用户在键盘上输入点什么。

因此，文件系统通常采用的规则，即处理请求直至完成，在这里就行不通了。我们有必要把这些请求挂起，等以后再执行。当一个进程试图从管道中读写数据时，文件系统会立即检查管道的当前状态，看此次操作能否完成。如果能，就等它执行完成；如果不能，文件系统就把此次系统调用的参数记录在进程表中，以便在时机到来后重新执行该进程。

请注意，文件系统并不需要采取特别的行动来挂起调用进程，它要做的无非是不发送应答消息，从而让调用进程阻塞在等待应答消息的状态。这样，在挂起该进程后，文件系统就回到它的主循环，等待下一个系统调用。一旦其他进程修改了管道的状态，使得被挂起的进程能够运行结束，那么文件系统会设置一个标志位。这样，在下一次循环时，文件系统会从进程表中提取被挂起进程的参数，并重新执行之前的系统调用。

终端和其他字符设备文件的情况略有不同。每个设备文件的*i*节点中都含有两个数字：主设备号和次设备号。主设备号给出了设备的类型（如RAM盘、软盘、硬盘和终端等），它主要用做文件系统表的索引，将之映射为相应的I/O驱动程序号。也就是说，主设备号决定了调用哪一个I/O驱动程序，而次设备号则作为参数传递给驱动程序，它指明了所使用的设备号，如终端2或驱动器1。

在有些情形下，尤其是在终端设备中，次设备号会包含一些设备信息，以便于驱动程序的处理。例如，MINIX 3的主控制台/*dev/console*的主、次设备号分别为4, 0，而虚拟控制台也使用同样的驱动程序来处理，如/*dev/ttym1*(4, 1)和/*dev/ttym2*(4, 2)等。串行终端需要使用不同的底层软件，这些设备（即/*dev/ttym0*和/*dev/ttym1*）的设备号分别为4, 16和4, 17。类似地，网络终端使用伪终端驱动程序，也需要用到不同的底层软件。在MINIX 3中，这些设备包括*ttyp0*, *ttyp1*等，它们的设备号分别为4, 128和4, 129。每一个伪设备都有一个与之关联的设备，如*ptyp0*, *ptyp1*等，与它们相对应的主、次设备号分别是4, 192和4, 193。之所以选择这些数字，主要是为了便于驱动程序去调用与各组设备相对应的底层函数。因为对于任何人来说，一般不大可能在一个MINIX 3系统中配备192台以上的终端。

当一个进程要从某个设备文件中读取数据时，文件系统会从该文件的*i*节点中提取出主设备号及次设备号，并以主设备号为索引，去访问一个文件系统表，找到相应的设备驱动程序的进程号。然后，文件系统就会给它发送一条消息，其参数包括次设备号、需要执行的操作、调用进程的进程号、缓冲区地址和要传送的字节数。消息的格式与图3.15中相同，只是没有用到POSITION项。

如果驱动程序能够立即执行该操作（例如，在终端上已经有一行输入），就把数据从自己内部的缓冲区复制到用户空间中，并向文件系统发送一条应答消息，告知任务已经完成。然后文件系统再向用户发送一个应答消息。至此，本次调用就结束了。需要注意的是，驱动程序并不会把数据复制到文件系统中。从块设备来的数据需要经过块高速缓存，但从字符设备文件中读取的数据则没有这个必要。

如果驱动程序不能立即执行这个操作，它将把消息参数记录在自己的内部表中，然后向文件系统发送一个应答消息，表明此次调用无法完成。这时，文件系统所面临的情况类似于当它发现某个进程试图从空管道中读取数据。因此，它将把调用进程挂起，并等待下一条消息。

当驱动程序获得了本次调用所需要的数据后，将把这些数据传送到仍被阻塞的用户进程的缓冲区中，并向文件系统发送消息汇报它的工作进展。然后，文件系统再向用户进程发送一条应答消息，使其解除阻塞，并告之已传送的字节数。

5.6.10 一个例子：READ 系统调用

正如我们即将要看到的，文件系统的大部分代码都用于系统调用的执行。因此，有必要通过一个例子来进行一个小结，我们来看一下 `read` 这个最重要的系统调用是如何实现的。

当一个用户程序需要去读取一个普通的文件时，会执行如下的语句：

```
n = read(fd, buffer, nbytes);
```

库函数 `read` 有三个参数，它会构造一条消息，包含这些参数，并使用 `read` 调用的编码来作为消息的类型，然后把这个消息发送给文件系统，并把自己阻塞起来，等待文件系统的应答。文件系统在收到这条消息后，会以消息类型为索引去查找它的内部表格，然后调用相应的函数来处理此次读操作。

该函数将从消息中提取出文件描述符，由此找到相应的 `filp` 表项以及目标文件的 i 节点（参见图 5.39）。然后，此次读请求将被分成几个段，每段对应于一个块。例如，假设当前的文件位置为 600，而要读取的数据长度为 1024 字节。那么，此次读请求将被分成两部分，一部分是从 600 到 1023 字节，另一部分是从 1024 到 1623 字节（假设块大小为 1 KB）。

对于上述这两个部分，依次检查它们的相关块是否在高速缓存中。如果不在，文件系统会选择最近最久未使用的缓冲区，把它所包含的块调出内存并回收这个缓冲区。如果这个块曾经被修改过，文件系统就向磁盘驱动程序发送一条消息，将其写回磁盘。然后，再让磁盘驱动程序把所需要的块读入内存。

如果要读入的块已经在高速缓存中，那么文件系统会向系统任务发送一条消息，请求它把数据复制到用户缓冲区中（即把从 600 到 1023 字节的这段数据复制到用户缓冲区的起始位置，而把从 1024 到 1623 的这段数据复制到它的后面，即从 424 字节开始）。在复制完成后，文件系统向用户程序发送一条应答消息，告诉它复制的字节数。

当用户程序收到应答消息后，库函数 `read` 会提取出应答代码，并把它返回给调用者。

这里还有额外的一步，严格地说，它并不是 `read` 调用的一部分。当文件系统完成了一次读操作并发送了一个应答消息后，如果此次读操作来自于块设备并满足了其他一些条件，那么文件系统就会准备读入更多的块。由于顺序读取文件是非常普遍的，因此可以期望下一次读操作将会请求文件的下一块，所以我们就提前开始这一操作。这样，当真正需要这个块时，它可能就已经在高速缓存中了。预先读入的块数取决于高速缓存的容量，最多可以请求 32 个块。当然，设备驱动程序无须返回这么多的块，事实上，只要能返回一个块，就认为本次请求是成功的。

5.7 MINIX 3 文件系统的实现

MINIX 3 的文件系统相对来说比较大（超过 100 页 C 代码），但是非常直观。执行系统调用的请求到达后，进行相应的处理，然后发送应答消息。在下面几小节中，我们将逐个文件地进行分析，指出其中的关键之处。另外，在代码本身中，也含有许多的注释以便于读者阅读。

在分析 MINIX 3 其他部分的代码时，我们通常的做法是先分析某个进程的主循环，然后再来看一下不同消息类型的处理函数。在介绍文件系统时，我们将采用一种不同的方法：首先研究主要

的子系统（高速缓存管理、i节点管理等），接着讨论主循环和各种文件操作的系统调用，之后是关于目录操作的系统调用，以及其他的一些系统调用。最后再来看一下设备文件的处理方式。

5.7.1 头文件和全局数据结构

同内核和进程管理器一样，文件系统中使用的各种数据结构和表格都定义在头文件之中。有些数据结构还放在系统一级的头文件中，位于 *include/* 及其子目录下。例如，头文件 *include/sys/stat.h* 定义了系统调用向其他程序提供 i 节点信息的格式，而目录项的结构则在 *include/sys/dir.h* 中定义。这两个文件都是 POSIX 所要求的。此外，全局配置文件 *include/minix/config.h* 中的许多定义也会影响到文件系统，如 *NR_BUFS* 和 *NR_BUF_HASH*，它们控制了块高速缓存的大小。

文件系统头文件

文件系统自身的头文件位于源文件目录 */src/fs/* 中，其中许多文件的名字在讨论 MINIX 3 系统的其他部分时已经见过了。文件系统的主头文件 *fs.h* (20 900 行) 非常类似于内核和进程管理器的头文件 *src/kernel/kernel.h* 和 *src/pm/pm.h*，它包含了文件系统的 C 源程序需要用到的其他头文件，如 *const.h*, *type.h*, *proto.h* 和 *glo.h* 等，下面我们将逐个分析这些头文件。

const.h (21 000 行) 定义了整个文件系统中用到的一些常量，如表的长度和标志位等。MINIX 3 的开发历史比较长，在它的早期版本中有不同的文件系统。虽然现在的 MINIX 3 并不支持老的 V1 和 V2 版本的文件系统，但有一些定义仍然保留了下来，一则可用做参考，二则将来可能会有人需要用到它们。比如说，可以用它们来访问早期的 MINIX 文件系统中的文件，或者是用来交换文件。

其他的操作系统可能会用到老的 MINIX 文件系统，例如，Linux 最初曾经用过而且现在仍然支持 MINIX 文件系统（具有讽刺意味的是，Linux 仍然支持早期的 MINIX 文件系统，而 MINIX 3 自己却不支持）。在 MS-DOS 和 Windows 系统上，仍然有一些工具软件能够访问老的 MINIX 目录和文件。在一个文件系统的超级块中会包含有一个魔数，用于标明该文件系统的类型，常量 *SUPER_MAGIC*, *SUPER_V2* 和 *SUPER_V3* 分别定义了 MINIX 文件系统的三个版本号。对于 V1 和 V2，还有一些以 *_REV* 为后缀的版本号，它们的字节顺序是颠倒过来的，主要用于把早期的 MINIX 版本移植到具有不同字节顺序的系统中（如小端格式）。在 MINIX 3.1.0 版本中，定义一个 *SUPER_V3_REV* 魔数是没有必要的，但也许在将来，这个定义还会被加上。

type.h (21 100 行) 同时定义了 V1 和 V2 版本的 i 节点在磁盘上的结构。在 MINIX 3 中，i 节点的结构没有发生变化，因此 V2 的 i 节点用在了 V3 文件系统中。V2 版本的 i 节点的长度是 V1 版本的两倍，后者主要是为带有 360 KB 磁盘且不带硬盘驱动器的系统而设计的。V2 版本提供了 UNIX 系统支持的三个时间字段，而在 V1 的 i 节点中只有一个时间字段，但是在 *stat* 或 *fstat* 调用中可以“弄虚作假”，伪造出一个 *stat* 结构来包含所有这三个字段。在支持这两种老版本的文件系统时有一些小困难，这一点在 21 116 行的注释中提到了。老版本的 MINIX 3 软件要求将 *gid_t* 类型定义为 8 位的值，所以 *d2_gid* 必须声明为 *u16_t* 类型。

proto.h (21 200 行) 提供了一些函数原型，其格式遵从于老的 K&R 或新的 ANSI 标准 C 编译器。这是一个很长的文件，但可讲的内容并不多。只有一点需要注意：由于文件系统需要处理很多各不相同的系统调用，以及文件系统自身的组织方式，所以各种 *do_XXX* 函数被分散在许多不同的文件中。而 *proto.h* 是按文件来进行组织的，因此，当你需要查阅某个特定系统调用的处理代码时，可以很方便地通过 *proto.h* 来找到相应的文件。

最后，*glo.h* (21 400 行) 定义了全局变量。用于输入和应答消息的缓冲区也在该文件中定义。这里同样用到了我们已经很熟悉的 *EXTERN* 宏，从而使这些变量可以被文件系统的各个部分所访问。同 MINIX 3 的其他部分一样，在编译 *table.c* 文件时，系统会预留一定的存储空间。

文件系统的进程表部分包含在文件 *fproc.h* 中（21 500 行）。*fproc* 数组也用 *EXTERN* 宏声明，它含有模式掩码、指向当前根目录和工作目录 i 节点的指针、文件描述符数组、UID、GID 以及各个进程的终端号，进程的 ID 以及进程的组 ID 也包含在这里。在进程管理器的进程表中，会有进程 ID 的一个副本。

有些字段用于存储出于某种原因（如读取一个空管道）而被中途挂起的系统调用的参数，*fp_suspended* 和 *fp_revived* 字段实际上只需要一位，但使用字符类型可以使编译器生成更好的代码。此外，有一个字段用于 *FD_CLOEXEC* 位，这也是按 POSIX 标准的要求而设置的，主要用于表明当一个 *exec* 调用被执行时，文件应该被关闭。

现在我们来看一下文件系统所维护的其他一些表格是在哪些文件中定义的。首先，文件 *buf.h*（21 600 行）定义了块高速缓存，这里的结构都是用 *EXTERN* 来声明的。*buf* 数组包含了所有的缓冲区，每个缓冲区分为数据部分和头部分。头部分中含有指针、标志位和计数器等。数据部分被声明为五种类型的一个联合体（21 618 到 21 632 行），这样我们可以根据需要，有时把它视为一个字符数组，有时把它视为一个目录，等等。

如果把缓冲区 3 的数据部分视为一个字符数组，那么正确访问该部分的方式是 *buf[3].b.b_data*。这里，*buf[3].b* 表示整个联合体，然后我们使用的是其中的 *b_data* 字段。这种方式虽然是正确的，但略显繁琐。因此在 21 649 行，定义了一个宏 *b_data*，这样对于上述的字符数组，我们就可以把它写成 *buf[3].b_data*。请注意，*b_data*（联合体中的字段）包含有两个下划线，而为了区别，在 *b_data*（宏）中只使用了一个下划线。对于这个块的其他访问方式，相应的宏定义请参见 21 650 行至 21 655 行。

缓冲区的哈希表 *buf_hash* 定义在 21 657 行，每个表项指向一个缓冲区列表。在初始化时，所有的列表均为空。文件 *buf.h* 末尾的宏定义了不同的块类型。*WRITE_IMMED* 位表明当一个块被修改后，应立即写回磁盘。而 *ONE_SHOT* 位表明某个块不大可能会马上被用到。以上这两个位如今已不再使用，但它们仍保留在那儿，将来如果谁想出一个好主意，通过修改数据块在高速缓存中的排队方式来提高系统的性能或可靠性，那么他可能会用到这些位。

在 *buf.h* 的最后一行，利用文件 *include/minix/config.h* 中的 *NR_BUF_HASH* 的值，定义了 *HASH_MASK*。*HASH_MASK* 用来与一个块号进行“与”操作，从而决定使用 *buf_hash* 中的哪一项来作为块缓冲区的搜索起点。

文件 *file.h*（21 700 行）中包含了中间表 *filp*（用 *EXTERN* 声明），它主要用于存放文件的当前位置及 i 节点指针（参见图 5.39），此外它还能给出其他一些信息，如文件的打开方式（只读、只写、可读写），以及当前有多少个文件描述符正指向该表项。

文件锁表 *file_lock*（用 *EXTERN* 声明）存放在 *lock.h*（21 800 行）中，这个数组的长度由 *NR_LOCKS* 指定，这个常量是在文件 *const.h* 中定义的，其值为 8。如果想在 MINIX 3 系统上实现一个多用户的数据库，这个值还要再增加。

在文件 *inode.h*（21 900 行）中定义了 i 节点表 *inode*（声明为 *EXTERN*），用于保存当前正在使用的 i 节点。如前所述，当一个文件被打开时，它的 i 节点被读入内存，并一直呆在那里直到文件被关闭。*inode* 结构定义提供了一些存放在内存中的信息，这些信息并不会写入磁盘 i 节点。请注意，这里的 i 节点只有一个版本，当一个磁盘 i 节点被读入内存时，V1 版本和 V2/V3 版本的文件系统之间的差别就已经被处理好了。文件系统的其他部分不必了解磁盘上的文件系统的格式，至少在修改信息被写回磁盘之前是这样。

至此为止，大多数字段的含义都应该是不言自明的，无须过多的解释。但这里有必要对 *i_seek* 进行说明。如前所述，为了优化，当文件系统发现某个文件正在被顺序读取时，它会试着提前把一

些块读入到高速缓存中。而对于随机访问的文件，则没有预读。当执行一次 `Iseek` 系统调用时，文件指针很可能会远离原来的位置，因此就把 `i_seek` 字段设置为禁止预读。

文件 `param.h` (22 000 行) 的功能与进程管理器中的同名文件相似。它为包含参数的消息字段定义了名称，从而便于使用。例如，可以用 `m_in.buffer` 来代替 `m_in.ml_pl`，去选择消息缓冲区 `m_in` 中的某个字段。

在文件 `super.h` (22 100 行) 中，我们定义了超级块表。在系统启动时，根设备的超级块会被装入到这里。而在挂装一个文件系统时，它的超级块也会被装入到这里。同其他表一样，`super_block` 表也被声明为 `EXTERN`。

文件系统的存储分配

本小节我们要讨论的最后一个文件并不是头文件，不过，正如我们在介绍进程管理器时所采用的做法，在浏览了所有的头文件之后，我们觉得有必要来介绍一下文件 `table.c` (22 200 行)，因为当这个文件在编译时，会包含上述的所有头文件。我们前面提到的大多数数据结构都是用 `EXTERN` 宏来定义的，如块高速缓存、`filp` 表、文件系统的全局变量以及文件系统中的进程表部分等。另外，系统在编译 `table.c` 时会保留一定的存储区，这种做法在 MINIX 3 的其他部分也曾经看到过。这个文件还包含了一个经过初始化的数组 `call_vector`，它是一个指针数组，在主循环中用来决定系统调用号与相应的处理函数之间的映射关系。在进程管理器中，我们也看到过类似的表格。

5.7.2 表格管理

对于每一个主要的表格，如块、i 节点、超级块等，都有一个相关的文件，该文件包含了该表格的各种管理函数。这些函数会被大量地调用，并成为表格与文件系统之间的主要接口。因此，在分析文件系统的代码时，我们将首先从这里开始。

块管理

块高速缓存是由文件 `cache.c` 中的函数来管理的。这个文件包含有 9 个函数，如图 5.40 所示。第一个函数 `get_block` (22 426 行) 是文件系统获取数据块的标准方式。当一个文件系统函数需要读取一个用户数据块、目录块、超级块或其他任何块时，都会调用 `get_block`，并指明相应的设备号和块号。

函数	功能
<code>get_block</code>	获取一个块，用于读或写
<code>put_block</code>	返回之前用 <code>get_block</code> 请求的块
<code>alloc_zone</code>	分配一个新区段（使一个文件变得更长）
<code>free_zone</code>	释放一个区段（当一个文件被删除时）
<code>rw_block</code>	在磁盘和高速缓存之间传送一个块
<code>invalidate</code>	清除用于某设备的所有高速缓存块
<code>flushall</code>	刷新某设备上所有修改过的块
<code>rw_scattered</code>	在设备上读或写分散的数据
<code>rm_lru</code>	从 LRU 链表中删除一个块

图 5.40 用于块管理的函数

当 `get_block` 被调用时，它首先去检查块高速缓存，看看请求的块是否在高速缓存中。如果是，就返回其指针。否则，就必须把它从磁盘读进来。高速缓存中的块用 `NR_BUF_HASH` 条链表

链接在一起。*NR_BUF_HASH* 和 *NR_BUFS* 都是可调的参数，后者指定了块高速缓存的大小。这两个参数都是在文件 *include/minix/config.h* 中定义的。在本小节的后面，我们将会提到如何来优化块高速缓存和哈希表的大小。*HASH_MASK* 等于 *NR_BUF_HASH* - 1，如果有 256 个哈希表，则 *HASH_MASK* 为 255。因此每条链表上的各个块的块号末 8 位都是相同的位串，即 00000000, 00000001, ... 或 11111111。

通常情况下，*get_block* 都首先要在哈希链表中查找块，但有一种情况比较特殊：如果读取的是一个稀疏文件中的一块空闲区，则不必进行查找。这就是我们在 22 454 行进行检查的原因。接下来的两行将指针 *bp* 指向某条链表的开始，如果所查找的块位于高速缓存中，它必然也在这条链表中，链表的下标是将 *HASH_MASK* 与块号相与的结果。接下来的循环会去查找这条链表，看是否能找到这个块。如果能找到，并且该块未被使用，则从 LRU 链表中删除之；如果它已在使用，则根本不会出现在 LRU 链表中。在 22 463 行，把找到的块的地址返回给调用进程。

如果该块不在哈希链表中，它也就不在高速缓存中。因此从 LRU 链表中选取最近最久未使用的块，将其缓冲区从原哈希链中删除，因为它即将获得一个新的块号，从而属于不同的哈希链。如果被移除的这个块曾经被修改过，则在 22 495 行把它写回磁盘。这里使用了 *flushall* 调用，它将把同一设备上所有被修改过的块都写回磁盘。大多数的块都是通过这个调用写回磁盘的。当前正在使用的块不会被置换出内存，事实上，它们根本就不在 LRU 链表中。不过，我们很难判断一个块是否还将使用，在通常情形下，当一个块在使用完后，就立即用 *put_block* 把它释放。

一旦分配了缓冲区，所有的字段，包括 *b_dev*，都使用新参数来进行更新（22 499 行至 22 504 行），然后把该块从磁盘上读入。不过，在两种情形下可能并不需要读入该磁盘块。当 *get_block* 的调用参数为 *only_search* 时，表明这是一次预读。在预读期间，会去查找一个可用的缓冲区，并在必要时将原有内容写回磁盘，然后把一个新的块号赋予这个缓冲区，但是它的 *b_dev* 字段被设置为 *NO_DEV*，表明该块中还没有有效的数据。后面在讨论 *rw_scattered* 函数时，我们会看到它的用法。*only_search* 还可以用来表明需要重写整个块，这时也没有必要先把这个块的旧内容读入内存。不管是哪一种情形，缓冲区中的参数都会被更新，但实际的读磁盘过程将被忽略（22 507 行至 22 513 行）。当新的块被读入后，*get_block* 将其指针返回给调用进程。

假设文件系统在查找文件名时需要一个临时的目录块，它会调用 *get_block* 来请求这个目录块。在找到文件名后，它会调用 *put_block*（22 520 行）把该块返回给高速缓存，从而使其缓冲区空闲，用于存放其他的块。

put_block 负责把新返回的块插入 LRU 链表，并且在某些情形下，把它的内容写回磁盘。在第 22 544 行，需要做出一个决策，即是把这个块放在 LRU 链表的前端还是尾端。RAM 盘中的块总是放在队列的前端，实际上，块高速缓存并不会为 RAM 盘做太多的事情，因为它的数据已经在内存中，无须 I/O 操作即可访问。*ONE_SHOT* 标志位被测试，看看这个块是否已经被标记为短期内不会使用的块。如果是这种块，就把它放在链表的前端，这样它的缓冲区很快就可以重新分配。不过，这种情况很少出现，除了 RAM 盘中的块，几乎所有的块都会被放在链表的末尾，以防它们马上又会被用上。

当一个块被放入 LRU 链表后，还要检查该块是否应立即写回磁盘。同前一次测试一样，*WRITE_IMMED* 的测试是一种过时的操作，因为当前已经没有数据块被标记为立即写回。

随着文件长度的增长，需要不时地给它分配新的区段以保存新数据。函数 *alloc_zone*（22 580 行）负责分配新的区段。它在区段位图中查找一个空闲区段。如果这是文件的第一个区段，则没有必要去查找位图，可以直接去使用超级块中的 *s_zsearch* 字段，它指向的是设备上的第一个可用区段。如果新区段不是文件的第一个区段，那么为了使文件的各个区段放在一起，需要去搜索区段位图，从

中找到与文件最后一个区段最近的那个空闲区段。在具体实现上，可以从文件最后一个区段所对应的位置开始搜索位图（22 603行）。位图中的位号与区段号之间的映射在22 615行处理，第一位对应于第一个数据区段。

当一个文件被删除时，需要把它所占用的区段返还到区段位图中。*free_zone*（22 621行）负责回收不再使用的区段。它所做的事情就是以区段位图和位号为参数，去调用*free_bit*函数。*free_bit*也可用于回收空闲的i节点，但需要使用i节点位图来作为第一个参数。

高速缓存的管理需要读、写数据块。为了提供一个简单的磁盘接口，MINIX系统提供了*rw_block*函数（22 641行），它负责读或写一个块。类似地，*rw_inode*用于读写i节点。

*cache.c*文件中的下一个函数是*invalidate*（22 680行）。例如，当一个磁盘被卸载时，就会调用这个函数，从高速缓存中把这个文件系统的所有块都删掉。如果不做这件事情，那么当该设备被再次使用时（用了另外一张软盘），文件系统可能会把原来的块当成新块来使用。

前面在讨论*get_block*的时候我们曾经提到，*flushall*（22 694行）函数负责完成大部分的数据写操作。在sync系统调用中，也会使用它来刷新某个指定设备上的所有被修改过的缓冲区，将它们全部写回设备。系统的更新进程会定期地激活sync，让它去调用*flushall*，刷新每一个被挂装的设备。*Flushall*会把高速缓存视为一个线性数组，这样所有被修改过的缓冲区都会被发现，不管它当前是否正被使用，也不管它是否出现在LRU链表中。高速缓存中的所有缓冲区都将被扫描，那些属于待刷新的设备以及需要写回的缓冲区被添加到*dirty*指针数组中。这个数组用static声明，所以不会被保存在栈中。然后把这个指针数组传给*rw_scattered*函数。

在MINIX 3中，磁盘写操作的调度任务没有放在设备驱动程序中，而是由*rw_scattered*（22 711行）来全权负责。该函数以设备标志符、指向缓冲区指针数组的指针、数组的大小和一个读写标志位为参数。它首先对这个数组按照块号进行排序，以提高读写操作的效率。然后构造邻接块的向量，并通过*dev_io*调用把它们发送给设备驱动程序。而驱动程序不再进行任何磁盘调度。只有在*flushall*函数中调用*rw_scattered*时，才会以WRITING为标志位。在这种情况下，这些块号的来源是容易理解的，它们实际上是一些修改过的数据缓冲区。唯一以只读方式调用*rw_scattered*的是文件*read.c*中的*rahead*函数。在目前，我们只要知道：在调用*rw_scattered*之前，已经以预取方式多次调用过*get_block*，因此预留了一组缓冲区。这些缓冲区中含有块号，但没有设备参数，不过这不要紧，因为在调用*rw_scattered*时会提供这个参数。

设备驱动程序对来自于*rw_scattered*的读请求和写请求有着不同的处理方式。写多个块的请求必定全部按要求处理，而对于读取多个块的请求，不同的驱动程序可能会有不同的处理方式，这取决于哪一种方式最有效。*rahead*在调用*rw_scattered*时，往往会请求一组数据块，而这些块并不一定都会被用上。所以，最好的应对方式是尽可能多地去获取那些很容易就能读取的块，但不要为了完成任务，在磁盘上到处跑，把磁头移来移去，从而浪费大量的寻道时间。例如，软盘驱动程序可能会在一条磁道的边界处停止，而许多其他驱动程序只会读取连续的块。当读操作完成后，*rw_scattered*将对读入的块进行标记，在它们的缓冲区中填入相应的设备号。

图5.40中的最后一个函数是*rm_lru*（22 809行），这个函数负责从LRU链表中删除一个块。它只被本文件中的*get_block*所调用，因而被声明为PRIVATE，这样其他文件中的函数就不能去使用它。

在结束讨论块高速缓存之前，我们简单地说一下如何来优化它。*NR_BUF_HASH*必须是2的整数次幂。如果它大于*NR_BUFS*，那么哈希链的平均长度将不超过1。如果内存能够存放大量的缓冲区，则必然有足够的空间来容纳大量的哈希链，因此通常的做法是把*NR_BUF_HASH*选为大于*NR_BUFS*的下一个2次幂。在本书的源代码列表中，定义了128个块和128个哈希表。最佳的大小取决于系统的使用情况，因为它决定了需要缓存多少数据。在本书所附带的CD-ROM上，在用来

编译标准MINIX 3可执行程序的完整源代码中，定义了1280个缓冲区和2048个哈希链。根据经验，我们发现如果将缓冲区的个数增加到这个数目以上，那么在重新编译MINIX时并不能改进系统的性能。因此，这个规模的缓冲区应该足够了，能够存放编译过程中产生的二进制代码。而对于其他一些作业，较小的缓冲区可能就足够了，但有时也可能需要使用更大的缓冲区来提高系统性能。

对于本书所附CD-ROM中的MINIX 3系统，它的缓冲区占用了5 MB的内存，而另一个二进制文件*image_small*是使用128个缓冲区的块高速缓存编译而成的，缓冲区占用的内存空间为0.5 MB，它可以安装在只有8 MB内存的系统上，而标准版本需要16 MB的内存。

i节点管理

块高速缓存并不是唯一需要支持函数的文件系统表，*i*节点表同样也需要。很多管理*i*节点表的函数与块管理函数非常相似，它们都列在图5.41中。

函数	功能
<i>get_inode</i>	将一个 <i>i</i> 节点读入内存
<i>put_inode</i>	返回不再需要的 <i>i</i> 节点
<i>alloc_inode</i>	为新文件分配一个 <i>i</i> 节点
<i>wipe_inode</i>	清除 <i>i</i> 节点中的某些字段
<i>free_inode</i>	释放一个 <i>i</i> 节点（当一个文件被删除时）
<i>update_times</i>	修改 <i>i</i> 节点中的时间字段
<i>rw_inode</i>	在内存与磁盘之间传送一个 <i>i</i> 节点
<i>old_icopy</i>	把 <i>i</i> 节点内容转换为要写入的V1磁盘的 <i>i</i> 节点
<i>new_icopy</i>	转换V1文件系统的磁盘 <i>i</i> 节点中读入的数据
<i>dup_inode</i>	表明他人正在使用一个 <i>i</i> 节点

图5.41 用于*i*节点管理的函数

*get_inode*函数（22 933行）与*get_block*类似。当文件系统的任何部分需要一个*i*节点时，它就调用这个函数。*get_inode*首先搜索*inode*表，看看所请求的*i*节点是否已经在内存中。如果是，就将其使用计数器加1，并返回其指针。这一搜索过程包含在22 945至22 955行的代码之中。如果*i*节点不在内存中，则调用*rw_inode*将它读入。

当一个函数使用完*i*节点之后，就调用*put_inode*函数（22 976行）把它返回。该函数将把*i*节点的使用计数器*i_count*减1。如果计数器的值为0，表明该文件不再使用，此时*i*节点可以从*inode*表中删除。如果它曾经被修改过，还要把它写回磁盘。

如果*i*节点的*i_link*字段为0，表明没有任何目录项指向该文件，因此它的所有区段均可被释放。注意，使用计数器为0和链接数为0是完全不同的两码事，它们的原因不同，所导致的结果也不同。如果这个*i*节点用于管道，则无论其链接数是否为0，它的所有区段都必须被释放。这种情形出现在某个进程在读取管道后将其释放时。对于系统来说，它不可能仅仅为了某一个进程而保留一个管道。

在创建一个新文件时，必须调用*alloc_inode*（23 003行）函数为它分配一个*i*节点。MINIX 3允许以只读模式来挂装设备，因此要检查设备的超级块以确保其可写。对*i*节点的处理和对区段的处理不同。系统尽量使一个文件的各个区段相邻，而*i*节点则无此必要。为了节省查找*i*节点位图的时间，我们可以利用超级块中用来记录第一个空闲*i*节点的字段。

在获得一个*i*节点后，将调用*get_inode*把它读入到内存的表中。然后初始化这个*i*节点的各个字段，一部分初始化工作在23 038行至23 044行完成，而另一部分工作则通过调用*wipe_inode*函

数(23 060行)来完成。之所以这样分工,是因为在文件系统的其他地方也要使用*wipe_inode*来清除i节点中的某些字段(但不是所有字段)。

当删除一个文件时,调用*free_inode*函数(23 079行)去释放它的i节点。这个函数的任务只是把i节点位图中的相应位设置为0,然后去更新超级块中用于记录第一个空闲i节点的字段。

下一个函数是*update_times*(23 099行),它负责从系统时钟获取时间,并修改相应的时间字段。*update_times*函数在stat和fstat这两个系统调用中也要用到,因此它被声明为PUBLIC。

*rw_inode*函数(23 125行)类似于*rw_block*,它的任务是从磁盘上获取一个i节点。具体执行过程如下:

1. 计算哪一个块包含有所需的i节点。
2. 调用*get_block*读入该块。
3. 从该块中提取i节点并把它复制到*inode*表中。
4. 调用*put_block*返回该块。

在真正实现上,*rw_inode*函数比上面列出的这几个步骤更为复杂,还需要增加其他的一些功能。首先,由于获取当前时间需要一次内核调用,开销较大,因此在需要修改i节点的时间字段时,若i节点已在内存中,则只设置它的*i_update*字段作为标志。然后,当i节点被写回磁盘时,如果发现*i_update*字段非0,就去调用*update_times*函数。

其次,MINIX的开发历史也增加了一点复杂性:原先的V1版本文件系统的磁盘i节点具有与V2版本不同的结构,因此系统提供了两个函数*old_icopy*(23 168行)和*new_icopy*(23 214行)来实现结构的转换。前者在内存中的i节点和V1文件系统所使用的i节点之间进行转换,后者在内存中的i节点和V2/V3文件系统所使用的i节点之间进行转换。这两个函数仅在本文件中调用,所以被声明为PRIVATE。每个函数都能处理两个方向的转换(从磁盘到内存和从内存到磁盘)。

老的MINIX版本被移植到与Intel处理器字节顺序不同的一些系统上,而MINIX 3将来也可能被移植到这样的体系结构上。每一种实现都在磁盘上使用本地字节顺序,而超级块中的*sp->native*字段指明了所使用的字节顺序。必要时,*old_icopy*和*new_icopy*都会调用*conv2*和*conv4*来交换字节顺序。当然,我们刚才描述的大部分内容在MINIX 3中都未使用,因为它并不支持V1磁盘的使用。

*dup_inode*函数(23 257行)仅仅增加i节点的使用计数器,当打开一个已经打开的文件时被调用。在第二次打开该文件时,并不需要再次把它的i节点从磁盘上读入。

超级块的管理

文件super.c中包含了超级块和位图的管理函数,总共有6个函数,如图5.42所示。

函数	功能
<i>alloc_bit</i>	从区段或i节点位图中分配一位
<i>free_bit</i>	释放区段或i节点位图中的某一位
<i>get_super</i>	在超级块表中查找某个设备
<i>get_block_size</i>	查询块大小
<i>mounted</i>	判断一个i节点是否位于某个已挂装的文件系统上
<i>read_super</i>	读取一个超级块

图5.42 用于管理超级块和位图的函数

当需要一个i节点或区段时,调用*alloc_inode*或*alloc_zone*。如前所述,这两个函数都会调用*alloc_bit*(23 324行)来搜索相关的位图。这个搜索过程用到了三重嵌套循环:

1. 最外层循环去遍历某个位图中的所有块。
2. 中间层循环去遍历某个块中的所有字。
3. 最内层循环去遍历某个字中的每一位。

中间层循环检查当前字的每一位是否全为1，如果是，则这个字中没有空闲的i节点或区段，所以就去检查下一个字。如果找到一个不是全1的字，则进入内层循环，查找相应的空闲（即0）位。如果所有块都已搜索完毕，还没有找到0位，则表明没有空闲的i节点或区段，因此就返回NO_BIT(0)。像这样的搜索过程需要花费很多的处理器时间，但如果能用上超级块中指向第一个空闲i节点或空闲区段的字段（其值被传递给alloc_bit函数的origin参数），则可以大大缩短搜索的时间。

相比之下，释放一个位就要简单得多，它不需要去搜索位图。free_bit函数（23 400行）首先计算哪一个位图块包含了要释放的位，然后调用get_block将这个块读入内存，并把相应位置为0，最后调用put_block。

下一个函数get_super（23 445行）用于在超级块表中搜索特定设备。例如，在挂装一个文件系统时，需要检查它是否已经被挂装，这时可以调用get_super去查找该文件系统的设备，如果没有找到，则说明相应的文件系统还未挂装。

在MINIX 3中，文件系统服务器能够处理不同块大小的文件系统，尽管在某个给定的磁盘分区中，只有一种块大小。get_block_size函数（23 467行）用来查询某个文件系统的块大小。它会在超级块表中搜索指定的设备，并返回该设备的块大小。如果该设备未被挂装，则返回一个最小的块大小MIN_BLOCK_SIZE。

下一个函数mounted（23 489行）仅在关闭块设备时被调用。在正常情况下，当一个设备被关闭时，它在高速缓存中的所有数据都会被丢弃。但如果这个设备恰巧是被挂装设备，那么这样做是不可取的。mounted函数的输入参数是一个指向设备i节点的指针，如果该设备是根设备或是被挂装的设备，则返回TRUE。

最后，我们看一下read_super函数（23 509行）。它与rw_block和rw_inode有些类似，但只用于读操作。超级块并不会被读入块高速缓存，而是直接向设备发出一个请求，要求读入1024个字节，起始地址与设备起点之间的偏移量也为1024个字节。在系统的正常操作中，一般不需要去修改超级块。read_super会检查文件系统的版本号，并在必要时进行相应的转换。这样，尽管磁盘上的超级块结构或字节顺序可能会有所不同，但是内存中的超级块副本都是标准的结构。

即使在MINIX 3中当前并未使用，但确定系统以不同的字节顺序写到磁盘上的方法也是清楚且值得注意的。超级块的魔数使用系统本身的字节顺序（即文件系统被创建的顺序）写入，当读超级块时，就会为逆字节顺序超级块进行测试。

文件描述符的管理

MINIX 3使用了专门的函数来管理文件描述符和filp表（见图5.39），这些函数位于filedes.c中。当一个文件被创建或打开时，需要给它分配一个空闲的文件描述符和一个空闲的filp表项，这时可以调用get_fd函数（23 716行）。不过，被分配的文件描述符和filp表项并不会标记为“已使用”，因为在确信creat和open成功之前，还需要进行很多项检查。

get_filp函数（23 761行）用于检查一个文件描述符是否在合适的范围内，如果是，就返回它的filp指针。

本文件中的最后一个函数是find_filp（23 774行）。当一个进程正在写一个断裂管道（即该管道未被任何读进程打开）时，需要调用该函数。它通过遍历搜索filp表，来查找潜在的读进程。如果未能找到，说明这个管道是断裂管道，因此写入失败。

文件锁

POSIX 记录锁函数如图 5.43 所示。在 *fcntl* 系统调用中指定 *F_SETLK* 或 *F_SETLKW* 请求，可以对文件的部分内容加上读写锁或只写锁。利用 *F_GETLK* 请求，可以判断文件的某一部分是否有锁。

操作	含义
<i>F_SETLK</i>	为某个区域加上读写锁
<i>F_SETLKW</i>	为某个区域加上写锁
<i>F_GETLK</i>	报告某个区域是否被锁

图 5.43 POSIX 的建议记录锁操作。这些操作通过系统调用 *FCNTL* 来请求

文件 *lock.c* 中只有两个函数。*lock_op* (23 820 行) 为 *fcntl* 所调用，调用时给出图 5.43 中的一个操作码。*lock_op* 会检查指定的锁范围是否有效。在设置锁时，不能与现有的锁冲突；而在清除锁时，不能将现有的锁一分为二。当一个锁被清除时，会去调用本文件中的另一个函数 *lock_revive* (23 964 行)，它会唤醒所有阻塞在这个锁上的进程。

之所以采取这一折中的方法，是因为要指出哪些进程正在等待释放某个特定的锁，需要编写额外的代码。那些还在等待加锁文件的进程被唤醒后，会再次阻塞。这种策略假定锁并不经常使用。如果要在 MINIX 3 系统上建立一个大型的多用户数据库，那么就需要重新实现文件锁这一部分。

当一个加锁文件被关闭时，也要调用 *lock_revive* 函数。这种情况是有可能发生的，例如，进程在结束使用一个加锁文件之前被撤销了。

5.7.3 主程序

文件系统的主循环包含在文件 *main.c* 中 (24 040 行)。在调用 *fs_init* 进行初始化以后，就进入了主循环。在结构上，这个主循环和进程管理器以及 I/O 驱动程序的主循环非常相似。它调用 *get_work* 等待下一条请求消息的到来(除非先前在管道或终端上被挂起的某个进程如今能够执行)。它还会把全局变量 *who* 设置为调用者的进程表项号，同时把另一个全局变量 *call_nr* 设置为即将执行的系统调用的编号。

一旦控制流回到主循环，变量 *fp* 会指向调用者的进程表入口，而 *super_user* 标志位会表明调用者是否是超级用户。通知消息的优先级较高，首先会检查 *SYS_SIG* 消息，看看系统是否正在被关闭。其次是优先级第二高的 *SYN_ALARM*，即文件系统设置的一个定时器已过期。*NOTIFY_MESSAGE* 意味着一个设备驱动程序已经就绪，它被派发给 *dev_status*。然后是关键部分，即执行系统调用的函数。用 *call_nr* 为索引去访问函数指针数组 *call_vecs*，从而选定相应的调用函数。

当控制流重新回到主循环之后，如果 *dont_reply* 被置位，则禁止发送应答消息（例如，进程由于读取空的管道而被阻塞）。否则就调用 *reply* (24 087 行) 发送一个应答消息。主循环的最后一条语句用来检测文件是否被顺序读取，如果是就采用预读方式，把下一个块预先读入高速缓存，以提高系统的性能。

本文件中的另外两个函数也与文件系统的主循环密切相关。*get_work* 函数 (24 099 行) 检查是否有以前阻塞的进程已经被唤醒，如果有，这些进程的优先级将高于新收到的消息。只有当文件系统没有内部操作时，才会去调用内核获得一条消息 (24 124 行)。往下跳过几行，我们可以看到 *reply* 函数 (24 159 行)。当一个系统调用完成之后，不管成功与否，都要调用 *reply* 函数，向调用进程发送一条应答消息。调用进程有可能已被某个信号所终止，因此从内核中返回的状态码被忽略，在这种情况下，不需要进行任何处理。

文件系统的初始化

文件 *main.c* 中尚未讨论的函数主要用于系统初始化。其中以 *fs_init* 为主，它是在整个系统的启动阶段，在文件系统尚未进入主循环之前被调用的。在第 2 章讨论进程调度的时候，我们曾经在图 2.43 中看到 MINIX 3 系统启动时的初始进程队列。文件系统的调度优先级要低于进程管理器，因此我们可以确信在系统启动时，进程管理器能够在文件系统之前运行。在第 4 章，我们讨论了进程管理器的初始化。PM 会构造它的进程表部分，并为自己和引导映像中的所有其他进程添加相应的表项，然后为每个进程向文件系统发送一条消息，这样文件系统就能对相应的表项进行初始化。现在我们来看一下这个交互的另一半内容。

当文件系统启动后，它会立即进入 *fs_init* 中的一个循环（24 189 行到 24 202 行），这个循环中的第一条语句是去调用 *receive*，获得在 PM 的 *pm_init* 初始化函数的第 18 235 行所发出的一条信息。每条信息包含一个进程号和一个 PID。前者用于文件系统的进程表中的索引，而后者被保存在相应表项的 *fp_pid* 字段。接下来，为每个选中的表项设置超级用户的真实、有效 UID 和 GID，以及一个全为 1 的掩码。如果一条进程号字段的值为 *NONE* 的消息被收到，那么该循环将结束，并送回一条消息给进程管理器，告诉它一切正常。

接下来，文件系统自己的初始化完成。首先，一些重要的常量被测试并验证其有效性；然后，调用其他的几个函数去初始化块高速缓存和设备表，并在必要时装入 RAM 盘以及根设备的超级块等。此时，根设备即可访问，然后使用另一个循环来遍历进程表的文件系统部分，这样从引导映像中装入的每一个进程都可以识别根目录，并使用根目录来作为它的工作目录（24 228 行到 24 235 行）。

在完成了与进程管理器的交互之后，*fs_init* 调用的第一个函数是 *buf_pool*，它起始于 24 132 行，用于构建块高速缓存所用到的链表。图 5.37 显示了块高速缓存的正常状态，即所有的块都被链在了 LRU 链和哈希链之中。为了更好地理解，我们来看一下图 5.37 的情形是如何产生的。当高速缓存被 *buf_pool* 初始化以后，所有的缓冲区都在 LRU 链表中，而且它们都将被链接到第 0 个哈希链中，如图 5.44(a) 所示。当一个缓冲区被请求并在使用时，我们就得到了图 5.44(b) 的情形，其中，我们看到一个块已从 LRU 链中删除，它现在位于另一个不同的哈希链中。

在正常情形下，块会立即被释放，并返回到 LRU 链表中。图 5.44(c) 显示了该块被返回给 LRU 链后的情形。尽管它已不再使用，但如果需要的话，还可以再次去访问它，并提供相同的数据内容，所以把它保留在哈希链中。当系统运行了一段时间后，差不多所有的块都可能被使用过，并随机地分散在不同的哈希链中。这时，LRU 链看上去就像图 5.37 一样。

在 *buf_pool* 之后，下一个被调用的函数是 *build_dmap*，我们把它放在后面，与其他处理设备文件的函数一起讨论。接下来被调用的是 *load_ram*，而它又调用了 *igetenv*（2641 行）函数。这个函数使用一个引导参数的名字为关键字，从内核中检索一个数值型的设备标识符。如果你用过 *sysenv* 命令来查看一个 MINIX 3 系统的引导参数，就会看到 *sysenv* 以数值型的方式来报告设备，其输出格式类似于

```
rootdev = 912
```

文件系统使用类似的编号来标识设备，编号的计算公式为 $256 \times \text{major} + \text{minor}$ ，其中 *major* 和 *minor* 分别是主设备号和次设备号。在上面这个例子中，主设备号为 3，次设备号为 144，它对应于 */dev/c0d1p0s0*，一般用于把 MINIX 3 安装在一个带有双磁盘驱动器的系统上。

load_ram（24 260 行）负责为 RAM 盘分配空间，如果引导参数需要，还可以在上面装入根文件系统。它使用 *igetenv* 函数来获得引导环境中设置的 *rootdev*，*ramimagedev* 和 *ramsize* 等参数（24 278 行到 24 280 行）。如果引导参数标明

```
rootdev = ram
```

那么根文件系统将被从名为 *ramimagedev* 的设备复制到 RAM 盘中，从引导块开始，逐块地进行复制。在复制时对文件系统的数据结构不进行任何解释。如果 *ramsize* 引导参数小于 *ramimagedev* 的大小，那么扩大 RAM 盘以容纳之。如果 *ramsize* 大于根设备文件系统的长度，则分配指定大小的空间，并调整 RAM 盘文件系统，让它使用所有这些空间（24 404 行至 24 420 行）。这是文件系统唯一一次对超级块进行写操作，而且与超级块的读操作类似，它并不会通过块高速缓存来进行，而是使用 *dev_io*，直接把数据写入设备。

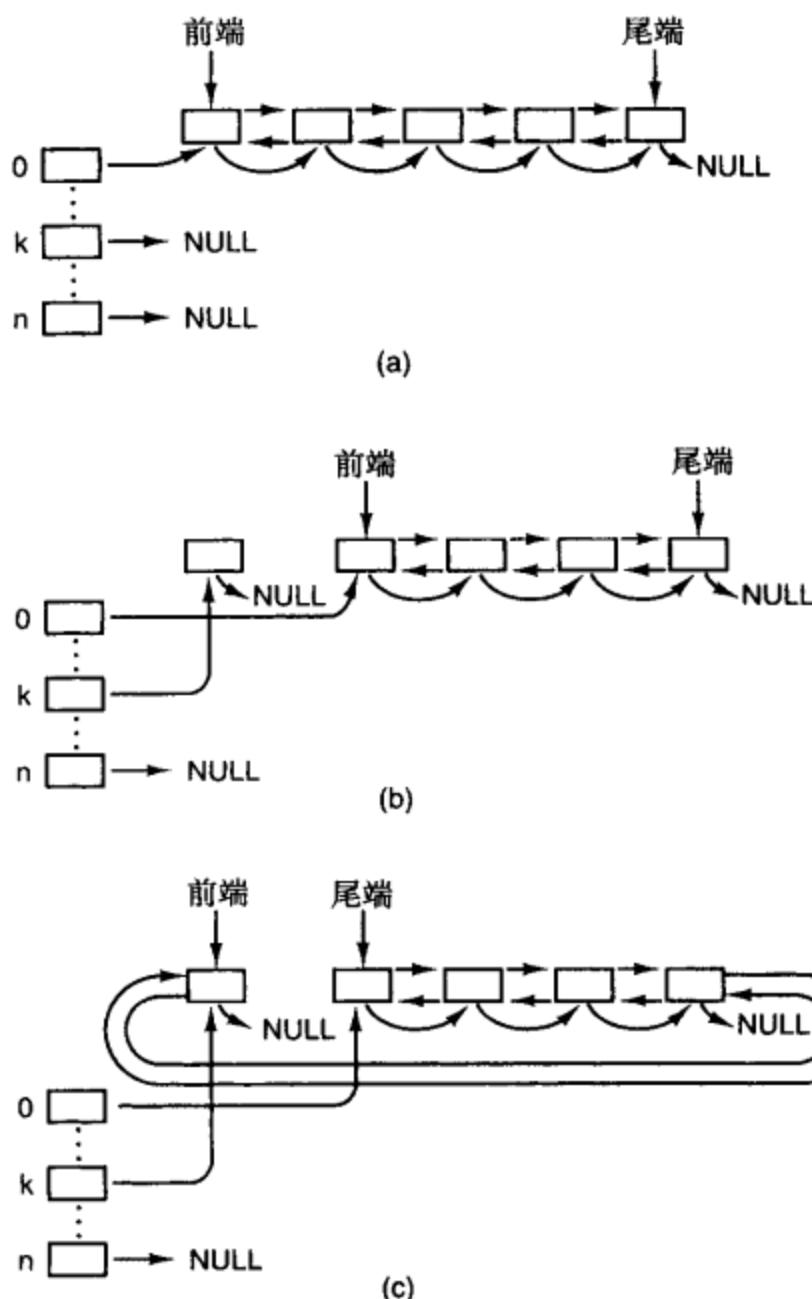


图 5.44 块高速缓冲的初始化: (a)在使用任何缓冲区之前;
(b)在请求一个块之后; (c)在该块被释放之后

这里有两点值得注意。一是 24 291 至 24 307 行的代码，这些代码用于处理自 CD-ROM 引导的情形。其中使用了本书未讨论的 *cdprobe* 函数，有兴趣的读者可参阅 *fs/cdprobe.c*（该文件可在 CD-ROM 或 Web 站点上找到）中的代码。二是不管 MINIX 3 用于普通磁盘访问的磁盘块大小为多少，引导块的大小总为 1 KB，且超级块自磁盘设备的第二个 1 KB 处装入。其他内容很复杂，因为在超级块装入前，块大小并不知道。

如果不使用 RAM 盘作为根文件系统，且 *ramsize* 非零，那么 *load_ram* 会为一个空的 RAM 盘分配空间。在这种情形下，由于未复制文件系统结构，因此该 RAM 设备不能用做文件系统，除非已经使用 *mkfs* 命令对它进行了初始化。此外，如果文件系统提供相应的支持，这样的 RAM 盘还可以用做辅助高速缓存。

文件 *main.c* 中的最后一个函数是 *load_super* (24 426行)。它初始化超级块表，并把根设备的超级块读入内存。

5.7.4 对单个文件的操作

本小节我们将介绍对单个文件进行操作的系统调用（区别于对目录的操作）。我们先从文件的创建、打开和关闭开始，然后详细讨论文件的读写机制。在这之后，再来看一下管道及其操作与文件有何不同。

创建、打开和关闭文件

文件 *open.c* 中包含有6个系统调用的代码，这些系统调用分别是 *creat*, *open*, *mknod*, *mkdir*, *close* 和 *Iseek*。我们先来看一下 *creat* 和 *open*，然后再逐个讨论其他的系统调用。

在老版本的UNIX中，*creat* 和 *open* 用于不同的目的。打开一个并不存在的文件会出错，而一个新文件必须用 *creat* 来创建，它还能把一个已存在文件的长度删除为0。不过，在POSIX系统中，不再需要这两种不同的调用。只用 *open* 调用就可以创建一个新文件或清除一个老文件，因此 *creat* 调用的功能现在只是 *open* 的一个子集，它之所以存在，只是用于与原有程序兼容。处理 *creat* 和 *open* 的函数分别是 *do_creat* (24 537行) 和 *do_open* (24 550行) (文件系统使用了与进程管理器相同的惯例，即系统调用 XXX 由函数 *do_XXX* 来执行)。打开或创建一个文件包括三个步骤：

1. 找到 i 节点（如果是一个新文件，需要分配 i 节点并进行初始化）。
2. 找到或创建相应的目录项。
3. 为文件建立并返回一个描述符。

creat 和 *open* 都要做两件事：获取文件名，并调用 *common_open* 去执行这两个调用的一些共同操作。

common_open 函数 (24 573行) 首先检查是否有空闲的文件描述符及空闲的 *filp* 表项。如果调用函数指明要创建一个新文件（在调用时设置 *O_CREAT* 位），则执行 24 594 行的 *new_node* 函数。如果目录项已存在，*new_node* 将返回相应的 i 节点指针；否则它会创建一个新的目录项和 i 节点。如果无法创建新的 i 节点，就设置全局变量 *err_code*。不过，一个错误码并不总是表示错误。如果 *new_node* 发现一个已有的文件，那么返回的错误码表明该文件已存在，但这种情形是可以接受的 (24 597行)。如果 *O_CREAT* 位没有设置，则需要用另一种方法来搜索 i 节点，即我们后面要讨论的文件 *path.c* 中的 *eat_path* 函数。无论是创建新文件还是查找旧文件，我们要知道的是，在控制流执行到 24 606 行之前，如果相应的 i 节点没有找到或创建失败，那么 *common_open* 将会出错并终止运行。接下来，对文件描述符进行赋值，并在 *filp* 表中申请一项。之后，如果刚刚创建了一个新文件，就跳过 24 612 到 24 680 行之间的那段代码。

对于现有文件，文件系统需要检查它的类型和模式等，从而确定它是否能打开。第 24 614 行的 *forbidden* 调用首先对 *rwx* 位进行一般性检查。如果该文件是一个普通文件，并且在调用 *common_open* 时设置了 *O_TRUNC* 位，那么就把文件的长度变为 0，并再次调用 *forbidden* (24 620 行) 以确保文件可写。如果权限允许，就调用 *wipe_inode* 和 *rw_inode* 重新初始化 i 节点，并把它写回磁盘。其他的文件类型（目录、设备文件、命名管道）也需进行相应的测试。对于设备，在 24 640 行（使用 *dmap* 结构）调用适当的函数来打开它。而对于命名管道，则调用 *pipe_open* (24 646 行)，并进行与管道有关的各种检查。

common_open 的代码，如同文件系统的许多其他函数一样，包含有大量的检查各种错误和非法组合的代码。这些代码对于设计一个无错的、健壮的文件系统是必不可少的。如果出现错误，前面

分配的文件描述符和`filp`表项将被收回, `i`节点将被释放(24 683行至24 689行)。在这种情况下,`common_open`函数将返回一个负数以表示出错。如果运行成功, 则返回一个正数, 即文件描述符。

现在我们可以更详细地讨论函数`new_node`(24 697行), 它负责分配`i`节点, 并为`creat`和`open`把路径名添加到文件系统中。它还用于`mknod`和`mkdir`这两个调用中, 我们稍后再讲。第24 711行的语句对路径名进行解析(即逐一地分析各级目录名), 直到最后一个目录。后面第三行的`advance`调用用来检查路径名的最后一个部分能否被打开。

例如, 如果调用

```
fd = creat("/usr/ast/foobar", 0755);
```

那么`last_dir`会把`/usr/ast/`目录的`i`节点装入内存的`i`节点表中, 并返回其指针。如果文件不存在, 那么不久我们就要用到这个`i`节点, 以便把`foobar`文件加入这个目录。其他用于增加或删除文件的系统调用, 也都用`last_dir`来打开路径名的最后一个目录。

如果`new_node`发现文件并不存在, 就调用24 717行的`alloc_inode`函数分配并载入一个新的`i`节点, 然后返回指向该节点的指针。如果没有空闲的`i`节点, `new_node`执行失败, 返回`NIL_INODE`。

如果能够分配一个`i`节点, 则继续执行24 727行的语句, 填入某些字段, 把它写回磁盘, 然后在最后的目录中加入相应的文件名(24 732行)。这里我们再次看到文件系统需要经常地检查错误, 一旦遇到错误, 就会小心地释放它申请到的所有资源, 如`i`节点和块。如果在出现各种异常的时候, 如`i`节点耗尽, 我们不去执行各种回退措施, 取消当前调用所造成的影响, 并给调用者返回一个错误码, 而是听之任之, 让MINIX 3陷于混乱, 那么我们的文件系统代码就会简单得多。

如前所述, 管道需要专门的处理。对于一个管道, 如果没有出现读者和写者的配对(即只有读者或只有写者), 那么`pipe_open`函数(24 758行)将挂起调用进程, 否则它就调用`release`函数, 在进程表中查找阻塞在该管道上的进程, 并将它们唤醒。

`mknod`调用由`do_mknod`(24 785行)来处理。这个函数与`do_creat`类似, 只不过它仅仅创建`i`节点并为其分配一个目录项。实际上, 大部分工作都是由24 797行的`new_node`调用完成的。如果`i`节点已存在, 则返回一个错误码。这个错误码与`common_open`中调用`new_node`所得到的错误码相同。但是在`common_open`中, 这个错误码是可以接受的, 而在这里, 错误码被传递给调用进程, 以便做相应的处理, 我们就不再逐行分析了。

`mkdir`调用由`do_mkdir`函数(24 805行)来处理。与上述其他系统调用一样, `new_node`也在这里起到了重要作用。与文件不同的是, 目录通常有链接。此外, 目录不可能为空, 因为在创建时, 它至少包含两个目录项“.”和“..”, 分别指向当前目录及其父目录。文件的链接数是有限制的, 不超过`LINK_MAX`(在文件`include/limits.h`中被定义为`SHRT_MAX`, 对于运行在标准32位Intel系统上的MINIX 3, 该值为32 767)。由于子目录对父目录的引用也是到父目录的一个链接, 因此`do_mkdir`做的第一件事情是去检查能否在父目录中创建另外一条链接(24 819行和24 820行)。如果可以, 就调用`new_node`函数。`new_node`成功执行后, 接着创建“.”和“..”两个目录项(24 841行和24 842行)。这一过程非常简单, 但也有可能会出错(如磁盘已满), 因此, 我们提供了很多的回退代码, 以便在操作无法完成时, 能够恢复到进程的初始状态。

关闭文件比打开文件要容易得多, 它主要是在`do_close`函数(24 865行)中完成的。管道和设备文件需要特别注意, 但是对于普通文件, 只需把`filp`计数器减1并看它是否为0。若为0, 则调用`put_inode`回收该`i`节点。最后删除所有的锁, 并唤醒所有阻塞在这些文件锁上的进程。

注意, 回收一个`i`节点意味着它在`inode`表中的计数器被减1, 这样它将最终从`inode`表中删除。这个操作与`i`节点的释放(即在位图中设置一位表明该`i`节点空闲)无关。只有在文件从所有目录中删除时, 才要释放`i`节点。

文件 *open.c* 中的最后一个函数是 *do_lseek* (24 939 行)。在进行文件的读/写定位时，需要调用这个函数来设置新的文件位置。在第 24 968 行，预读被禁止，因为如果去修改文件指针的当前位置，则意味着不再进行文件的顺序存取。

读文件

文件一旦被打开，就可以进行读写。在读、写文件时，可以用到许多函数，这些函数位于 *read.c* 中。我们先来分析这个文件，然后再来分析 *write.c* 中专门用于写操作的代码。读操作和写操作有很大的不同，但也有很多相似之处，因此，*do_read* 函数 (25 030 行) 在执行时，仅需在调用公共函数 *read_write* 时设置 *READING* 标志位。在后面一节中我们还会看到，*do_write* 的代码也同样简单。

read_write 起始于 25 038 行。在 25 063 行至 25 066 行之间有一些特殊的代码，进程管理器使用这些代码让文件系统把它的整个段装入用户空间。正常调用的处理从 25 068 行开始，之后是一些有效性检查（例如，从一个只写文件中读取数据）以及变量的初始化。从字符设备文件中读取的数据并不经过块高速缓存，因此在 25 122 行将它们筛选出来。

从 25 132 行到 25 145 行的测试只适用于写操作，用于处理文件长度可能超出设备容量的情形，或者是超出文件末尾写数据从而在文件中产生空闲区。我们在 MINIX 3 概述中曾经说过，一个区段中使用多个块可能会导致一些问题，需要特别处理。此外管道也比较特殊，需要进行检查。

对于普通文件来说，读文件的关键是从 25 157 行开始的循环。这个循环把读请求分解成若干小块，每个小块都在一个单独的磁盘块中，它从当前位置开始，直到满足下列条件之一：

1. 所有字节都已读完。
2. 遇到一个块边界。
3. 读到文件的末尾。

这些规则表明一个小块不可能用到两个磁盘块。图 5.45 中的三个例子说明了如何来确定小块的大小，三个块的大小分别为 6，2 和 1 个字节。实际的计算由 25 159 到 25 169 行的代码来完成。

块的读取操作是在 *rw_chunk* 中完成的。当它执行完后，各种计数器及指针的值会加 1，接着进入下一个循环。当循环结束后，文件的位置和其他一些变量（如管道指针）也可能会更新。

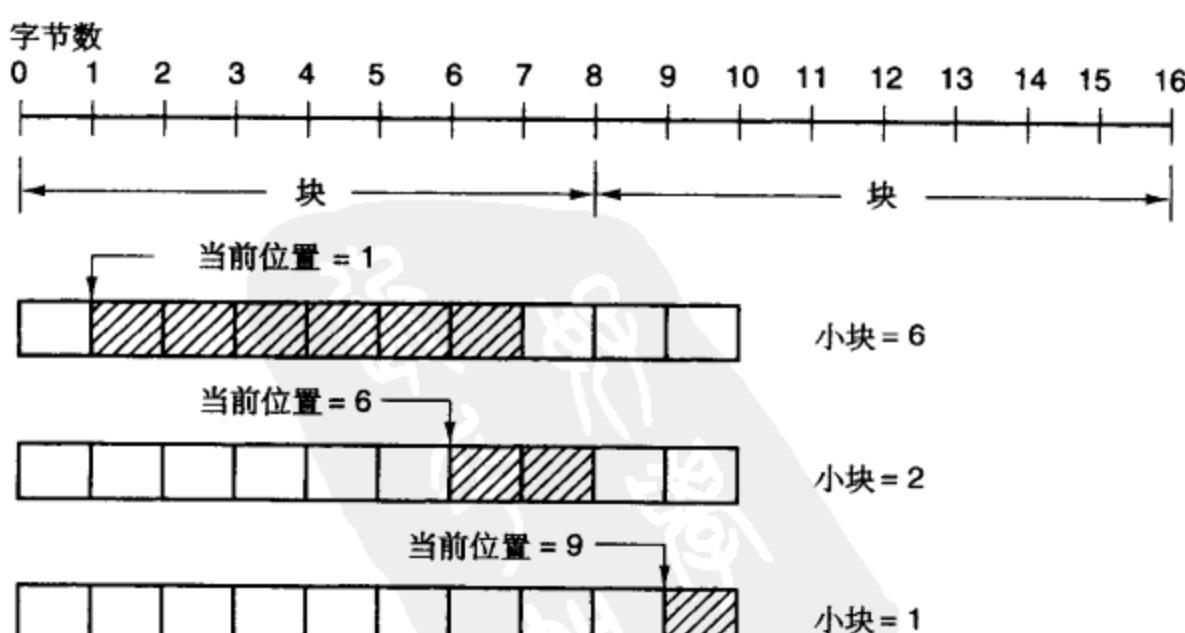


图 5.45 三个例子表明在 10 字节文件中如何确定第一个小块的长度。

块的大小为 8 个字节，请求的字节数为 6。小块用阴影显示

最后，如果要求预读，则将要读的*i*节点和位置保存在全局变量中，这样，在向用户发送应答消息后，文件系统能够接着读取下一个磁盘块。在多数情况下，文件系统会被阻塞，等待下一个磁盘块，这时，用户程序可以处理刚刚收到的数据。这种安排使得计算和I/O操作可以并发进行，从而大大提高了系统的性能。

*rw_chunk*函数(25 251行)以*i*节点和文件位置为参数，把它们转换成一个物理磁盘块号，然后请求把该块(或其中的一部分)传送到用户空间中。相对文件位置与物理磁盘地址的映射是在*read_map*函数中进行的，它能够处理*i*节点和间接块。对于普通文件，25 280行的变量**b**和25 281行的变量**dev**分别包含了物理块号及设备号。在25 303行的*get_block*调用负责去查找该块，并在必要时把它读入内存。在25 295行调用*rahead*函数用于确保该块被读入高速缓存。

一旦我们获得了该块的指针，就可以调用25 317行的内核调用*sys_vircopy*将块中的所需部分传送到用户空间中。随后用*put_block*释放该块，使之在稍后被换出块高速缓存(在调用*get_block*申请到某一块后，该块不会出现在LRU队列中，而且只要块头中的计数器表明该块还在使用，它就不会返回到LRU链中，从而不会被换出。*put_block*将计数器减1，并在它减为0时，将这个块返回到LRU队列中)。第25 327行的代码表明是否一次写操作会填满块。不过，通过n传递给*put_block*的值并不会影响到如何将该块放入队列，因为所有块一律放在LRU链的尾部。

read_map(25 337行)检查*i*节点，将逻辑文件位置转换为物理块号。对于靠近文件头、在前面7个区段(即那些在*i*节点中的区段)中的块，通过简单的计算便可知道要访问的是哪个区段、哪个块。而对于文件后面的块，则需要读取一个或多个间接块。

在读取一个间接块时，需要调用*rd_indir*(25 400行)。这个函数的注释有点过时，因为支持68000处理器的代码已经被删掉，而支持MINIX V1文件系统的代码已不再使用，也可以放弃。不过，如果将来有人想要在系统中添加新的特性，以便支持其他的文件系统版本，或具有不同磁盘数据格式的其他平台，那么可能会碰上不同的数据类型和字节顺序等问题这些问题都可以归并到这个文件中。如果需要进行繁琐的转换，那么把它们放在这里，就会使文件系统的其他部分只见到同一种格式的数据。

read_ahead(25 432行)将逻辑位置转换成物理块号，调用*get_block*确保该块位于高速缓存中(或将其调入)，之后立即返回该块。*read_ahead*对块并不进行任何操作，毕竟，它的目的只是说，当需要用到某个块时，该块已经在内存的概率会比较大。

请注意，*read_ahead*仅在*main*函数的主循环中被调用，它不是作为*read*系统调用的一部分而被调用的。我们还要意识到，*read_ahead*调用是在发送应答消息之后进行的，所以，用户可以接着进行其他操作，尽管文件系统不得不等待读取下一个磁盘块。

*read_ahead*本身只是请求另一块，它调用文件*read.c*中的最后一个函数*rahead*来完成该操作。*rahead*(25 451行)的工作原理基于这样的思想：如果稍多一点比没有强，那么再多一点则更好。由于磁盘和其他的存储设备在寻找第一块时要花费较长一段时间，而读取相邻块所需的时间则较短，因此，稍加努力，就可以读取更多的块。预读请求送给*get_block*，后者会申请块高速缓存以便一次接收多个块。接着调用*rw_scattered*，并以一组块为参数。如前所述，当设备驱动程序被*rw_scattered*调用时，各个驱动程序可以根据自己的具体情况，选择那些能高效处理的请求去完成，而其他的请求则被忽略。这听起来很复杂，但对于那些需要从磁盘上读取大量数据的应用程序，这种复杂的办法也许能显著地提高读盘速度。

图5.46显示了在读取一个文件时用到的一些主要函数之间的关系，尤其是它们之间的调用关系。

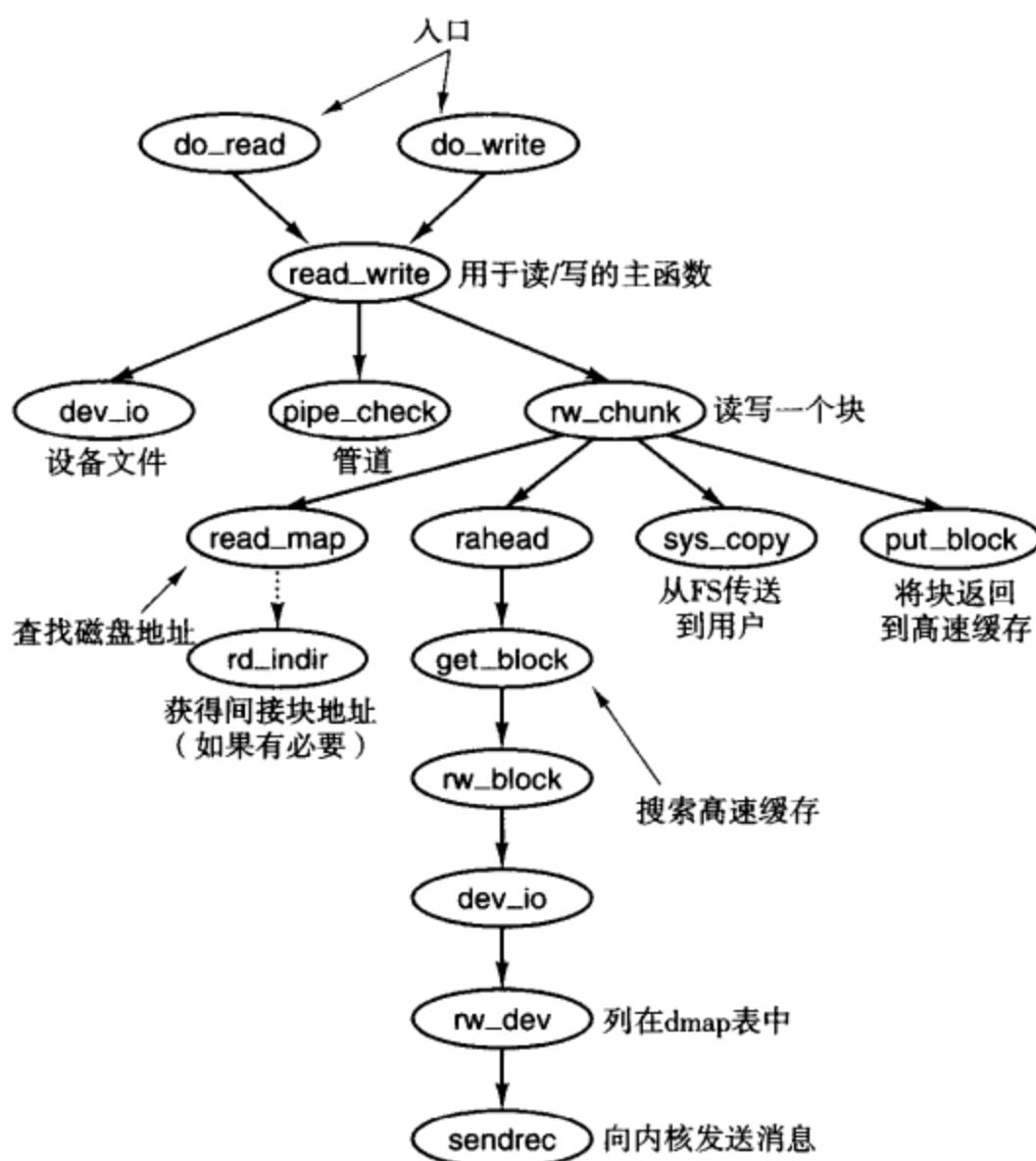


图 5.46 读取一个文件时用到的一些函数

写文件

写文件的代码放在文件 `write.c` 中。写文件与读文件类似，`do_write` 函数（25 625 行）只是在调用 `read_write` 时设置 WRITING 标志位。读和写之间的主要区别在于写文件需要分配新的磁盘块。`write_map`（25 635 行）与 `read_map` 类似，只不过它不是在 i 节点和间接块中查找物理块号，而是在其中添加新的物理块号（准确地说，它添加的是区段号，而不是块号）。

`write_map` 的代码很长，因为它需要处理几种不同的情况。如果插入的区段靠近文件头，我们只需要将它插在 i 节点中（25 658 行）。

最糟糕的情况是，文件超过了一级间接块所能处理的长度，这时需要用到一个二级间接块。接下来，分配一个一级间接块，并将其地址填入二级间接块中。同读文件一样，这时要调用一个单独的函数，`wr_indir`。如果成功获取二级间接块，而这时磁盘已满，无法分配一级间接块，就需要将前面获得的二级间接块返回，以免破坏位图。

如前所述，如果我们不去做错误检测和回退，而是任由系统陷于混乱和严重故障，那么文件系统的代码就会简单得多。然而，从用户的观点来看，当磁盘空间不足时，`write` 调用应该返回一个错误码，而不是使系统崩溃且造成文件系统的破坏。

`wr_indir`（25 726 行）调用 `conv4` 进行必要的数据转换，并将新区段号写入间接块中（同样，这里有一些老的 V1 文件系统的残余代码，但当前只有 V2 的代码被使用）。请注意这个函数的名字，同其他涉及到读写操作的函数名一样，它的字面含义并不一定很准确。实际的磁盘写操作是由管理块高速缓存的函数来进行的。

文件 *write.c* 中的下一个函数是 *clear_zone* (25 747行)，它负责清空突然出现在文件中部的某些块。当把文件指针移过文件末尾并写入数据时，就会出现这种情况。幸好，它并不经常发生。

若需要一个新块，*rw_chunk*会去调用 *new_block* (25 787行)。图 5.47 显示了一个顺序文件增长时的 6 个连续阶段。在这个例子中，块大小为 1 KB，区段大小为 2 KB。

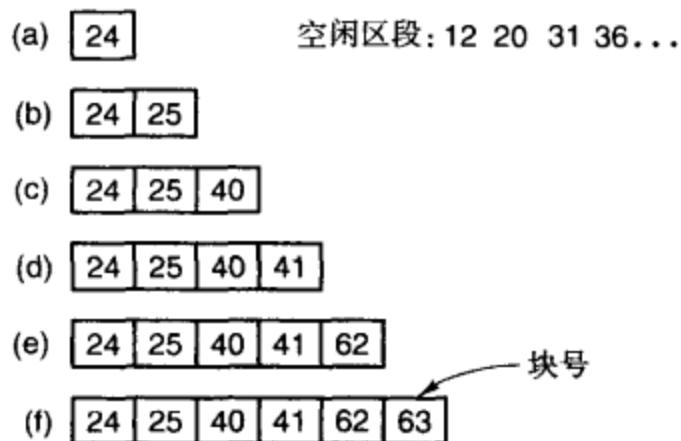


图 5.47 (a)~(f) 区段大小为 2 KB、块大小为 1 KB 时块的连续分配

第一次调用 *new_block* 时，分配区段 12 (块 24 及 25)。接下来使用块 25，该块已经被分配但尚未被使用。第三次调用时，分配区段 20 (块 40 及 41)，等等。*zero_block* (25 839行) 用于清空某个块，删除以前的内容，它的代码非常简短。

管道

在许多方面，管道和普通文件很类似。本小节我们主要讨论它们之间的区别。我们要分析的代码都在文件 *pipe.c* 中。

首先，管道的创建方式不同，管道是由 *pipe* 而不是由 *creat* 来创建的。*do_pipe* (25 933行) 负责处理 *pipe* 调用，它为管道分配一个 i 节点，并返回两个文件描述符。管道是由系统而不是由用户所拥有的，并放在指定的管道设备上 (在文件 *include/minix/config.h* 中配置)。由于管道数据无须永久保存，因此可以把 RAM 盘用做管道设备。

管道的读写与文件的读写也略有不同，管道的容量是有限的。当管道已满时，所有继续向管道写入数据的进程都将被挂起。同样，读取空管道的进程也会被挂起。事实上，管道有两个指针：当前位置指针 (由读进程使用) 和长度指针 (由写进程使用)。这两个指针决定了数据从哪儿来、到何处去。

pipe_check (25 986行) 进行各种检查，以保证对管道的操作能够完成，如果没有通过这些检查，调用进程将被挂起。除此之外，*pipe_check* 还调用 *release* 函数，看看能否唤醒那些由于没有数据或数据过多而被挂起的进程。对于正在睡眠的写进程和读进程，唤醒操作分别是在 26 017 行和 26 052 行执行。*pipe_check* 还要检查是否存在往断裂管道 (即没有读进程的管道) 中写数据的异常。

挂起一个进程是由 *suspend* 函数实现的 (26 073 行)。该函数将调用参数保存在进程表中，并将 *dont_reply* 标志设置为 TRUE，从而禁止文件系统的应答消息。

release 函数 (26 099 行) 检查一个被挂起在管道上的进程，看它能否继续运行。如果找到一个这样的进程，它就调用 *revive* 设置一个标志位，以便主循环能对它进行处理。这个函数不是一个系统调用，我们之所以把它列在图 5.33(c)中，主要是因为它使用了消息传递机制。

文件 *pipe.c* 中的下一个函数是 *do_unpause* (26 189 行)。当进程管理器试图向某个进程发送信号时，它必须知道该进程是否正挂起在某个管道或设备文件上 (在这种情况下，必须用 *EINTR* 错误将其唤醒)。由于进程管理器并不能处理管道和设备文件，所以它就向文件系统发送一条消息询问，

这条消息由 *do_unpause* 来处理。如果一个进程被阻塞，*do_unpause* 就把它唤醒。同 *revive* 一样，*do_unpause* 有些类似于一个系统调用，但实际上它并不是一个系统调用。

pipe.c 的最后两个函数 *select_request_pipe* (26 247 行) 和 *select_match_pipe* (26 278 行) 都支持 *select* 调用，这里不再详述。

5.7.5 目录和路径

前面我们已经讨论了文件的读写机制，下面我们来看看路径名和目录是如何处理的。

将路径名转换成 i 节点

许多系统调用（如 *open*, *unlink* 和 *mount*）都以路径名（即文件名）作为参数。这些系统调用在开始执行它们自己的任务之前，通常都要先获得目标文件的 i 节点。因此，如何将一个路径名转换成相应的 i 节点，这是本小节讨论的主要内容。在图 5.16 中我们已经看过了大致的概要。

路径名的解析是在文件 *path.c* 中进行的。第一个函数是 *eat_path* (26 327 行)。它的输入是指向路径名的指针，然后对它进行解析，并把它的 i 节点调入内存，最后返回这个 i 节点的指针。在具体实现中，它首先调用 *last_dir* 获取最后一层目录的 i 节点，随后调用 *advance* 取得目标文件名。如果搜索失败，比如在路径中有一个目录不存在，或者虽然存在但却禁止搜索，那么就返回 *NIL_INODE*。

路径名可以是绝对路径名或相对路径名，可以包含任意多个部分，各部分之间以斜杠分隔。这些问题都在 *last_dir* 函数中处理。它首先检查路径名的第一个字符，判断它是绝对路径名还是相对路径名 (26 371 行)。如果是绝对路径名，就把 *rip* 指向根目录的 i 节点；如果是相对路径名，就把 *rip* 指向当前工作目录的 i 节点。

现在，*last_dir* 已经知道了路径名以及指向某个目录的 i 节点，因而可以在这个 i 节点中查找路径名的第一部分了。接下来，它进入 26 382 行的循环，逐一地对路径名进行解析。当到达末尾时，便返回指向最后目录的指针。

get_name 工具函数 (26 413 行) 负责从字符串中提取出文件路径名的各个部分。更有意思的是 *advance* 函数 (26 454 行)，它以目录指针和一个字符串为参数，在目录中查找该字符串。如果找到，就返回指向相应 i 节点的指针。在 *advance* 中还要处理如何来跨越各个挂装的文件系统。

尽管 *advance* 负责字符串的查找，但字符串与目录项的比较是在 *search_dir* (26 535 行) 中实现的，这里是文件系统中唯一检查目录文件的地方。它含有两个嵌套的循环：一个是对目录中各个块的循环，另一个是对块中各个项的循环。*search_dir* 也可以用于在目录中增加和删除名字。图 5.48 给出了用于路径名查找的一些主要函数之间的关系。

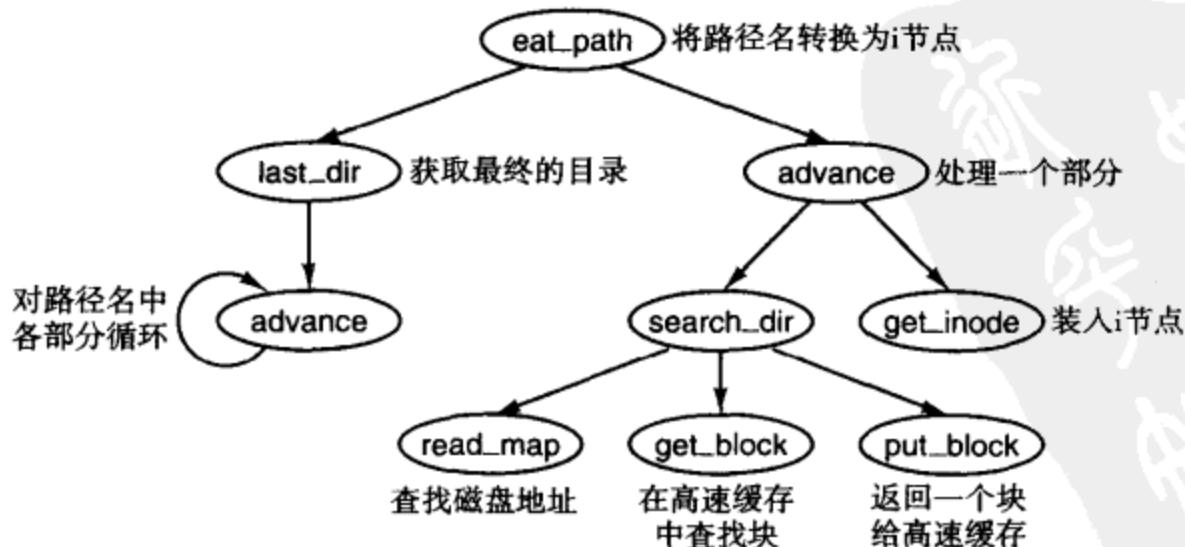


图 5.48 用于路径名查找的一些函数

挂装文件系统

mount 和 umount 这两个系统调用在总体上影响着文件系统。利用这两个调用，可以将不同次设备上的相互独立的文件系统“黏接”在一起，形成一个无缝的目录树。正如我们在图 5.38 中看到的那样，挂装是通过把被挂装文件系统的根目录 i 节点及超级块读入内存，并在超级块中设置两个指针来实现的。其中一个指针指向目标挂装点的 i 节点，另一个指针指向被挂装文件系统的根目录 i 节点。这两个指针把不同的文件系统链接在一起。

上述两个指针是由文件 *mount.c* 中的 *do_mount* 函数在 26 819 行和 26 820 行设置的。在指针设置前的两页代码主要用于检查在挂装文件系统时可能出现的各种错误：

1. 给定的设备文件不是块设备。
2. 设备文件是块设备，但已经被挂装。
3. 被挂装的文件系统具有不正确的魔数。
4. 被挂装的文件系统无效（如没有 i 节点）。
5. 要被挂装到的文件不存在或者是设备文件。
6. 没有空间来存放被挂装文件系统的位图。
7. 没有空间来存放被挂装文件系统的超级块。
8. 没有空间来存放被挂装文件系统的根目录 i 节点。

或许我们不宜在此反复强调，然而在一个实用的操作系统中，确实需要有很大一部分的代码用于错误处理。如果用户偶尔挂装一个错误的软盘（如每月碰上一次），从而导致系统崩溃，文件被破坏，他肯定会怀疑系统的可靠性，并且责怪系统的设计者，而不会从他自身找原因。

这里可以引用托马斯·爱迪生的一段论述。他认为“天才”是 1% 的灵感加 99% 的勤奋。一个好的系统与一个平凡的系统之间的区别，并不在于调度算法有多优秀，而在于是否考虑到了所有的细节问题。

卸载文件系统要比挂装容易得多，出错的机会也更少。*do_umount* (26 828 行) 负责处理卸载，它可以分为两个部分。首先，检查本次调用是否是由超级用户发出的，然后把文件系统的名字转换为相应的设备号，并调用 *umount* (26 846 行) 去完成此次卸载操作。这里的关键在于要确保没有进程在被卸下的文件系统中具有打开文件或工作目录。这个检查过程很直观：只需扫描整个 i 节点表，看看内存中是否有 i 节点属于要卸载的文件系统（根目录 i 节点除外）。如果有，则此次 *umount* 调用失败。

文件 *mount.c* 中的最后一个函数是 *name_to_dev* (26 893 行)，它以一个设备文件的路径名为参数，去获取它的 i 节点并从中提取主设备号和次设备号。这些设备号被存放在 i 节点中原用于存放第一个区段的地方。该位置之所以能够使用，是因为设备文件并没有区段。

链接和解链文件

下一个文件是 *link.c*，它用于链接和解链文件。同 *do_mount* 类似，在 *do_link* (27 034 行) 中几乎所有的代码均用于错误检查。在执行调用

```
link(file_name, link_name);
```

时，可能发生的错误如下：

1. *file_name* 不存在或不能访问。
2. *file_name* 已经有最大数目的链接。
3. *file_name* 是一个目录（只有超级块用户才可以对目录进行链接）。

4. *link_name* 已经存在。
5. *file_name* 和 *link_name* 在不同的设备上。

如果没有错误发生，则创建一个新的目录项，其名称为 *link_name*，而 i 节点号为文件 *file_name* 的 i 节点号。在代码中，*name1* 对应于 *file_name*，而 *name2* 对应于 *link_name*。新目录项实际上是由 *do_link* 在 27 086 行调用的 *search_dir* 函数创建的。

解链可以删除文件和目录。系统调用 *unlink* 和 *rmdir* 的工作都是由 *do_unlink* (27 104 行) 来完成的。这里，我们依然要进行各种检查，如文件是否已存在以及目录不是一个挂装点，这些工作是在 *do_unlink* 的公共代码中完成的，然后根据所支持的系统调用，分别去执行 *remove_dir* 或 *unlink_file*。我们很快就会讨论到这些内容。

link.c 支持的另一个系统调用是 *rename*，UNIX 用户所熟悉的 shell 命令 *mv* 最终会去调用这个系统调用。此外，它的名称还反映了该调用的另一方面：它不仅可以在目录中改变文件名，还能将文件从一个目录移至另一个目录，这些都是自动完成的。具体工作由 *do_rename* (27 162 行) 来实现。在完成这个命令之前，需要测试一系列条件，其中包括：

1. 源文件必须存在 (27 177 行)。
2. 在目录树中，原路径名不能是新路径名上面的目录 (27 195 行至 27 212 行)。
3. 原路径名和新路径名均不能是“.” 和“..” (27 217 行至 27 218 行)。
4. 它们的父目录必须在同一个设备上 (27 221 行)。
5. 它们的父目录均可写、可查找，并且都在可写设备上 (27 224 行和 27 225 行)。
6. 原文件名和新文件名都不能是某个文件系统所挂装到的目录。

如果新文件名已经存在，那么还要检查其他一些条件，其中最重要的是，我们必须有权去删除以新文件名为名字的现有文件。

在 *do_rename* 的代码中，有些例子可以说明我们是如何来减少某些问题出现的可能性的。当我们把一个文件的名字换成另一个已经存在的文件名时，必须先删除原来的文件。否则，如果当前磁盘空间已满，那么此次操作可能就无法完成。这就是 27 260 行到 27 266 行代码的目的。类似的道理也出现在 27 280 行，当我们在同一个目录下创建一个新文件名时，先要删除原来的文件名，以避免该目录可能要申请额外的块。然而，如果新文件名和原文件名位于不同的目录下，则不必考虑这个问题。在 27 285 行，我们在删除原文件名前先创建了一个新文件名（在不同目录下）。因为从系统完整性的观点来看，如果此时发生系统崩溃，使两个文件名同时指向一个 i 节点，那么这种情形远比有一个 i 节点未被任何目录项所指向的情形要好得多。在执行换名操作时空间耗尽的可能性很小，由此导致系统崩溃的可能性更小，但我们还是应该准备应付最坏的情况，何况这并不费事。

文件 *link.c* 中剩下的函数为我们上面讨论过的这些函数提供支持。此外，它们中的第一个函数 *truncate* (27 316 行) 还在文件系统的其他几个地方被调用。它在 i 节点中查找区段，释放它所找到的所有区段及其间接块。*remove_dir* (27 375 行) 执行许多额外的测试，以确保该目录可被删除，然后调用 *unlink_file* (27 415 行)。如果没有错误，目录项被清空，i 节点中的链接数减 1。

5.7.6 其他的系统调用

我们要讨论的最后一组系统调用涉及状态、目录、保护、时间和 other services 等。

改变目录和文件的状态

文件 *stadir.c* 中包含有 6 个系统调用的代码：*chdir*, *fchdir*, *chroot*, *stat*, *fstat* 和 *fstatfs*。在分析 *last_dir* 时，我们看到路径查找首先检查第一个字符是否为分隔符，然后根据结果，将指针分别指向工作目录或根目录。

从一个工作目录（或根目录）切换到另一个目录，只需要修改调用进程的进程表中的相应指针。这一修改是在 *do_chdir* (27 542 行) 和 *do_chroot* (27 580 行) 中进行的。这两个函数都要做一些必要的检查，然后调用 *change* (27 594 行)，后者进行更多的一些检查，然后调用 *change_into* (27 611 行) 打开新目录以取代原来的目录。

do_fchdir (27 529 行) 支持 *fchdir*，它的功能与 *chdir* 是类似的，只不过输入参数是文件描述符而不是路径名。它首先会测试描述符的有效性，然后调用 *change_into* 来完成任务。

在用户进程调用 *chdir* 时，并不执行 *do_chdir* 中的 27 552 行至 27 570 行的代码。这些代码是专门为进程管理器的调用而准备的，用于在处理 *exec* 调用时将当前目录修改为用户的目录。当一个用户想要在他的工作目录下执行一个文件时，如 *a.out*，对于进程管理器来说，直接切换到这个工作目录比查找出它的位置要容易得多。

系统调用 *stat* 和 *fstat* 基本上是一样的，只是文件的指定方式不同：前者给出的是路径名，而后者给出的是已打开文件的文件描述符。这与前面的 *chdir* 和 *fchdir* 的情形是类似的。最上层的函数 *do_stat* (27 638 行) 和 *do_fstat* (27 658 行) 都调用 *stat_inode*。不过在调用 *stat_inode* 之前，*do_stat* 还要先打开文件，以获取其 i 节点。这样，*do_stat* 和 *do_fstat* 在调用 *stat_inode* 时，都以一个 i 节点指针为参数。

stat_inode (27 673 行) 从 i 节点中提取信息，并把它复制到一个缓冲区中。由于缓冲区很大，无法放在一条消息中，因此要在 27 713 行和 27 714 行调用 *sys_datacopy* 内核调用将其复制到用户空间。

最后是 *do_fstatfs* (27 721 行)。*fstatfs* 并不是一个 POSIX 调用，尽管 POSIX 也定义了一个类似的 *fstatvfs* 调用，用于返回一个更大的数据结构。MINIX 3 的 *fstatfs* 只返回一条信息，即文件系统的块大小。它的函数原型为

```
_PROTOTYPE(int fstatfs, (int fd, struct statfs *st));
```

这里用到的 *statfs* 结构很简单，可以用一行来显示：

```
struct statfs { off_t f_bsize; /* 文件系统的块大小 */ ;
```

这些定义都包含在 *include/sys/statfs.h* 中，我们未把它列在附录 B 中。

保护

MINIX 3 的保护机制使用了 *rwx* 位。每一个文件都有三组 *rwx* 位，分别用于所有者、所有者所在的组以及其他用户。这些位由 *chmod* 系统调用来设置，而该调用是由文件 *protect.c* 中的 *do_chmod* 函数 (27 824 行) 来实现的。在进行一系列的有效性检查后，*do_chmod* 最终会在 27 850 行改变保护模式。

chown 系统调用和 *chmod* 有些相似，两者都会修改某个文件内部 i 节点的一个字段。在实现上两者也很类似，只不过对于 *do_chown* (27 862 行) 而言，只有超级块用户才有权限去改变一个文件的所有者，而普通用户只能用它去改变他们自己的文件所在的组。

umask 系统调用允许用户去设置一个掩码（保存在进程表中），用于屏蔽此后的 *creat* 系统调用中的相应位。这个函数的真正实现实际上只需要 27 907 行的一条语句，但由于要返回原来的掩码，这就使整个函数的代码量变成了原来的 3 倍 (27 906 行至 27 908 行)。

进程可以通过 `access` 系统调用来查询它是否能以某种特定的方式来访问一个文件（如读文件）。这个调用是在 `do_access` 函数（27 914 行）中实现的。它首先取得文件的 i 节点，然后调用一个内部函数 `forbidden`（27 938 行）来检查此次访问是否被禁止。`forbidden` 会检查 UID、GID 以及 i 节点的信息，并根据这些信息，选取三组 `rwx` 中的一组，来判断此次访问是允许还是禁止。

`read_only`（27 999 行）是一个内部函数，它的输入为一个 i 节点，然后返回该 i 节点所在的文件系统的挂装方式，是只读还是可读写。这个函数主要用于防止对挂装为只读的文件系统进行写操作。

5.7.7 I/O 设备接口

如前所述，MINIX 3 的一个设计思路是把所有的设备驱动程序都作为用户空间的进程来运行，不直接对内核的数据结构或代码进行访问，从而提高系统的健壮性。这种方法的主要优点是当一个设备驱动程序发生错误时，不会导致整个系统崩溃。此外，这种方法还有其他一些含义。首先，对于在系统启动时不急于使用的设备驱动程序，可以在启动后的任何时间被启用。其次，在系统运行的任何时刻，一个设备驱动程序可以被中止、重启或被同种设备的另一个驱动程序所取代。当然，这种灵活性也是有前提条件的，即同一个设备不能同时启用多个驱动程序。不过，如果硬盘驱动程序崩溃了，可以从 RAM 盘中的副本重新启动它。

MINIX 3 的设备驱动程序是由文件系统来访问的。当用户提出 I/O 访问请求后，文件系统就会给用户空间的设备驱动程序发送消息。对于每一种可能的主设备类型，都有一个 `dmap` 表项，它提供了主设备号与相应的设备驱动程序之间的映射。我们下面将讨论的两个文件就是用来处理 `dmap` 表的，这个表是在 `dmap.c` 中声明的。该文件还支持 `dmap` 的初始化和一个新的系统调用 `devctl`，用来支持设备驱动程序的启动、中止和重启。之后，我们将介绍 `device.c`，它支持设备的普通操作，如 `open`、`close`、`read`、`write` 和 `ioctl`。

当一个设备被打开、关闭、读或写时，`dmap` 会提供相应操作的处理函数的名字。所有这些函数都位于文件系统的地址空间中，多数函数并不做什么具体的事情，只是去调用某个设备驱动程序来完成实际的 I/O 操作。另外，这个表格还提供了每个主设备所对应的进程号。

当一个新的主设备被添加到 MINIX 3 中时，必须在 `dmap` 表中添加一行，表明当这个设备被打开、关闭、读或写时，应该执行什么操作。例如，如果在 MINIX 3 中添加了一个磁带机，那么当它的设备文件被打开时，表格中的函数应该会检查这个磁带机是否已经在使用。

`dmap.c` 用一个宏定义 `DT` 开头（28 115 行到 28 117 行），用于初始化 `dmap` 表。采用这种宏定义的方式，可以使重新配置 MINIX 3 时，能够方便地添加一个新的设备驱动程序。`dmap` 表的元素是在 `include/minix/dmap.h` 中定义的，每个元素包含 `open` 或 `close` 所对应的函数指针、`read` 或 `write` 所对应的函数指针、一个进程号（进程表的索引，不是 PID）以及一组标志位。实际的表格是包含这些元素的一个数组，在 28 132 行定义。这个表格是文件服务器内部的一个全局变量，它的长度是 `NR_DEVICES`，在 MINIX 3 的当前版本中，这个值是 32，是当前实际支持的设备个数的近 2 倍。幸运的是，C 语言编译器会自动地把所有未初始化的变量都设置为 0，这样在空闲的表项中就不会出现虚假的信息。

在 `dmap` 定义的后面是声明为 `PRIVATE` 的 `init_dmap`，它是用一个 `DT` 宏数组来定义的，每个宏对应于一个主设备。在编译时，这些宏将被扩展，从而对全局数组中的相应项进行初始化。我们来看一下其中的几个宏，因为这有助于我们理解它们的工作方式。`init_dmap[1]` 定义了内存驱动程序的表项，其主设备号为 1。宏的样子如下：

```
DT(1, gen_opcl, gen_io, MEM_PROC_NR, 0)
```

内存驱动程序是一直要用到的，它是随着系统引导映像被装入的。第一个参数“1”表示这个驱动程序必须存在，*gen_opcl*指针表示设备在打开或关闭时所调用的函数，*gen_io*表示设备在读或写时所调用的函数，*MEM_PROC_NR*表示内存驱动程序所使用的进程表项号，而最后的“0”表示没有标志位被设置。下面来看第二项 *init_dmap*[2]，它用于软盘驱动程序，它的宏为

```
DT(0, no_dev, 0, 0, DMAP_MUTABLE)
```

第一个参数“0”表示该设备的驱动程序无须包含在系统的引导映像中。第一个指针字段使用了默认值，表明在打开该设备时，去调用 *no_dev*。接下来的两个 0 也是默认值：由于设备不能被打开，因此没有必要指定一个函数去进行 I/O 操作，而进程表项 0 被解释为没有相应的进程。标志位 *DMAP_MUTABLE* 的含义是允许去修改这个表项（注意在内存驱动程序中没有设置这个标志位，表明它的表项在初始化以后不能被修改）。MINIX 3 可以在引导映像中配置为带有或不带有软盘驱动程序。如果软盘驱动程序在引导映像中，并在引导参数中使用 *label = FLOPPY* 来表明它是默认的磁盘设备，那么在文件系统启动时，就会去修改相应的表项。如果软盘驱动程序不在引导映像中，或者虽然它在，但并未被设置为默认的磁盘设备，那么在文件系统启动时，这个字段就不会被修改。不过，以后还有机会去激活软盘驱动程序，例如，当 *init* 在运行时，脚本 */etc/rc* 可能会去做这件事情。

do_devctl (28 157 行) 是服务于 *devctl* 调用的第一个函数，当前的版本很简单，它识别两个请求：*DEV_MAP* 和 *DEV_UNMAP*。对于后者，返回一个 *ENOSYS* 错误，表明“函数尚未实现”，显然，这只是权宜之计。而对于 *DEV_MAP*，调用下一个函数 *map_driver*。

这里我们来描述一下 *devctl* 调用的使用方式，以及将来的使用计划。在 MINIX 3 中，使用了一个服务器进程，即再生服务器 (reincarnation server, RS)，来支持在操作系统引导和运行后，用户空间的服务器和驱动程序的启动。再生服务器的接口函数是 *service*，它的用法可以在 */etc/rc* 中看到。一个例子是

```
service up /sbin/floppy -dev /dev/fd0
```

这个操作会使再生服务器执行一次 *devctl* 调用来启动可执行程序 */sbin/floppy*，用它来作为设备文件 */dev/fd0* 的驱动程序。具体来说，RS 会调用 *exec* 来执行这个程序，但同时设置一个标志位，使它在变为一个系统进程之前暂不能运行。一旦这个进程位于内存中，且它的进程表项号已知，那么指定设备的主设备号就确定了。然后把这个信息包含在一条消息中，发送给文件服务器，请求 *devctl* 的 *DEV_MAP* 操作。从 I/O 接口初始化的角度来说，这是再生服务器最重要的一部分工作。此外，为了完成设备驱动程序的初始化，RS 还要调用 *sys_privctl*，让系统任务去初始化驱动程序进程的 *priv* 表项，并允许它执行。在第 2 章我们曾经说过，一个专用的 *priv* 表项可以使一个普通的用户空间进程变为一个系统进程。

再生服务器是比较新的内容，在 MINIX 3 版本中，它仍然处于发展阶段。在 MINIX 3 将来的版本中，我们计划实现一个功能更强大的再生服务器，它不仅能启动驱动程序，还支持中止和重启操作。此外，它还能监控驱动程序，并在出现问题时，自动地重新启动它们。请留意我们的 Web 网站 (www.minix3.org) 和新闻组 (comp.os.minix)，以了解最新的进展情况。

我们接着讲 *dmap.c* 的内容，函数 *map_driver* 起始于 28 178 行，它的操作很简单。在 *dmap* 表的相应表项中，如果 *DMAP_MUTABLE* 标志位被设置，那么就把合适的值写入每个表项。对于设备的打开和关闭，有三种不同的处理函数，通过 RS 发送给文件系统的消息中传递的 *style* 参数来进行

选择（28 204行到28 206行）。注意`umap_flags`未被修改，如果该表项最初被标记为`DMAP_MUTABLE`，那么在`devctl`调用后，会一直保持这个状态。

`dmap.c`中的第三个函数是`build_map`，它是在文件系统刚刚被启动，还未进入主循环时，由`fs_init`来调用的。首先，对本地`init_dmap`表中的所有项进行遍历，对于每一个不使用`no_dev`来作为`dmap_opcl`成员的表项，把扩展后的宏复制到全局`dmap`表中，这样就正确地初始化了这些表项。对于未被初始化的驱动程序，把默认值设置到相应的`dmap`表项。接下来的工作更有意思。在一个引导映像中可以包含多个磁盘设备驱动程序。在默认情形下，通过`src/tools/`下面的`Makefile`文件，可以把`at_wini`、`bios_wini`和`floppy`驱动程序加入到引导映像中。每个驱动程序都会带有一个标签，而引导参数中的`label =`项将会决定哪一个驱动程序被实际装入映像，并作为默认的磁盘驱动程序。28 248行和28 250行的`env_get_param`调用使用库函数，并最终使用`sys_getinfo`内核调用去获取引导参数中的`label`和`controller`字符串。最后，在28 267行调用`build_map`去修改与引导设备相对应的`dmap`表项。这里的关键问题是把进程号设置为`DRVR_PROC_NR`，它在进程表中是第6项。这个表项具有神奇的力量：它里面的驱动程序就是系统默认的驱动程序。

现在到了文件`device.c`，它包含了在系统运行时设备I/O所需要的各种函数。

第一个函数是`dev_open`（28 334行），最常调用该函数的是`main.c`中的`common_open`（当使用`open`操作来访问一个设备文件时），此外还有`load_ram`和`do_mount`。`dev_open`首先确定主设备号，并验证其有效性。然后用它来设置一个指针，指向`dmap`表中的某个表项，并调用相应的函数。在28 349行，

```
r = (*dp->dmap_opcl)(DEV_OPEN, dev, proc, flags)
```

对于磁盘驱动器，被调用的函数是`gen_opcl`；而对于终端设备，被调用的函数是`tty_opcl`。如果收到一个`SUSPEND`返回码，表明发生了严重的问题；`open`调用一般不会这样。

下一个调用`dev_close`（28 357行）比较简单。这个调用一般不会作用在无效的设备上，而且即便关闭操作失败也没有什么伤害，因此这个函数的代码非常简短，只有一行语句，即在设备被关闭时，终止调用`dev_open`所调用的那个`*_opcl`函数。

当文件系统收到一个设备驱动程序所发来的通知消息时，就去调用`dev_status`（28 366行）。一个通知意味着一个事件已经发生了，而这个函数负责去查明发生的是什么样的事件，并发起相应的操作。通知的起源用进程号来指明，因此第一步是去搜索`dmap`表，找到发送通知的进程所对应的表项（28 371行到28 373行）。由于存在虚假通知的情形，因此即使没有找到相应的表项，也不算是一个错误。如果找到了一个匹配，就进入28 378行至28 398行的循环。在每一遍循环中，发送一条消息给驱动程序进程，询问它的状态。收到的应答消息可能有三种类型。如果最初请求I/O操作的进程已经被挂起，那么可能会收到一条`DEV_REVIVE`消息。在这种情形下，需要调用`revive`（在`pipe.c`中，26 146行）。如果对该设备执行了一次`select`调用，那么可能会收到一条`DEV_IO_READY`消息。最后一种消息类型是`DEV_NO_STATUS`。变量`get_more`用于使循环不断地进行，直到收到了`DEV_NO_STATUS`消息。

当需要实际的设备I/O时，在`read_write`（25 124行）中对`dev_io`（28 406行）的调用可以处理字符设备文件，而在`rw_block`（22 661行）中对`dev_io`的调用可以处理块设备文件。`dev_io`构造了一条标准的消息（参见图3.17），并通过调用`gen_io`或`catty_io`将之发送给指定的设备驱动程序，至于到底是`gen_io`还是`catty_io`，这是在`dmap`表的`dp->dmap_driver`字段中指定的。当`dev_io`正在等待驱动程序的应答消息时，文件系统也处于等待状态，它未使用内部的多道程序技术。通常，这些等待的时间都很短（如50 ms），但有可能数据并没有到来，尤其是当我们向一个终端设备请求

数据时。在这种情形下，应答消息可能是 *SUSPEND*，即暂时挂起调用进程，但是让文件系统继续往下运行。

函数 *gen_opcl* (28 455行) 主要用于磁盘设备，包括软盘、硬盘或基于内存的设备。首先构造一条消息，然后使用 *dmap* 表来确定到底是调用 *gen_io* 还是 *ctty_io* 来把这条消息发送给设备的驱动程序进程。*gen_opcl* 还用于关闭同一个设备。

在打开一个终端设备时，需要调用 *tty_opcl* (28 482行)。在对标志位进行修改后，该函数又调用了 *gen_opcl*。如果此次调用使该 *tty* 成为活动进程的控制 *tty*，那么就记录在进程表 *fp_tty* 的相应表项中。

设备 */dev/tty* 是一个虚的东西，它并不对应于任何特定的设备。这是一个神奇的名称，交互式用户可以用它来指示他自己的终端，而不管实际使用的是哪一个物理终端。为了打开或关闭 */dev/tty*，需要调用 *ctty_opcl* (28 518行)，它判断当前进程的 *fp_tty* 进程表项是否已经被上一次 *ctty_opcl* 调用所修改，以表明这是一个控制 *tty*。

setsid 系统调用要求文件系统完成某些任务，这是由 *do_setsid* (28 534行) 来实现的。它会修改当前进程的进程表项，以记录该进程是一个会话首领且没有控制进程。

系统调用 *ioctl* 主要在文件 *device.c* 中处理。之所以把它放在这里，是因为它和设备驱动程序接口密切相关。在执行 *ioctl* 时，会调用 *do_ioctl* (28 554行) 来创建一条消息并将其发送给合适的设备驱动程序。

为了控制终端设备，在符合 POSIX 标准的程序中，必须使用在头文件 *include/termios.h* 中定义的一个函数。C 语义库将把这些函数转换为 *ioctl* 调用。对于非终端设备，*ioctl* 被用于许多操作，其中多数已在第 3 章中介绍过了。

下一个函数 *gen_io* (28 575行) 是这个文件中真正“卖力干活”的函数。不管设备的操作类型是打开或关闭、读或写，或者是一个 *ioctl*，这个函数都会被调用来完成本次任务。由于 */dev/tty* 不是一个物理设备，当一条指向它的消息必须被发送时，下一个函数 *ctty_io* (28 652行) 会找到正确的主设备号和从设备号，并且在继续转发消息之前，把消息中的设备号替换掉。

函数 *no_dev* (28 677行) 是从设备并不存在的表项中被调用的。例如，在一台没有网络支持的机器中去访问一个网络设备。*no_dev* 返回一个 *ENODEV* 状态，在访问并不存在的设备时，它能防止系统的崩溃。

device.c 中的最后一个函数是 *clone_opcl* (28 691行)。有些设备在打开时需要进行特殊的处理，这样的一个设备将会被“克隆”，也就是说，在一次成功的打开操作后，它将被一个带有新的次设备号的新设备所取代。在本书所介绍的 MINIX 3 版本中，这个特性并未使用。不过在联网以后，这个特性就会被启用。对于这种类型的设备，在它的 *dmap* 表项的 *dmap_opcl* 字段中，需要指明 *clone_opcl*。这是通过再生服务器中的一个调用完成的，指明为 *STYLE_CLONE*。当 *clone_opcl* 打开一个设备时，开始时的操作与 *gen_opcl* 完全相同，但在返回时，在应答消息的 *REP_STATUS* 字段可能会返回一个新的次设备号。在这种情形下，如果能够分配一个新的 i 节点，就创建一个临时文件。目录项则没有必要创建，因为该文件已经被打开了。

时间

每个文件都有三个与时间有关的 32 位数，其中两个记录了文件的最后访问时间和最后修改时间，第三个记录了文件 i 节点本身最后修改时间，这个时间在几乎所有的文件访问时都要修改，唯有 *read* 和 *exec* 操作例外。这三个时间都保存在 i 节点中，所有者和超级用户可以通过 *utime* 系统调用设置文件的访问时间和修改时间。文件 *time.c* 中的 *do_utime* 函数 (28 818行) 负责执行这

个系统调用，它取出文件的*i*节点，并把时间保存在其中。在28 848行重置了与时间更新有关的标志位。这样，系统就不会重复地去调用运行开销较大的*clock_time*。

正如我们在上一章所看到的，系统的实时时间等于系统启动以来的时间（由时钟任务来维护）加上启动时的时间。**stime** 系统调用返回当前的实时时间，它的大部分工作是由进程管理器来完成的，不过文件系统也会在一个全局变量 *boottime* 中记录系统的启动时间。当 **stime** 被调用时，进程管理器会向文件系统发送一条消息，文件系统的 *do_stime*（28 859行）从这条消息中更新 *boottime*。

5.7.8 附加的系统调用支持

有一些文件没有列在附录B中，但在编译时需要用到它们。在本小节，我们先来看一些支持附加系统调用的文件，下一节我们将会介绍给文件系统提供了更广泛支持的一些文件和函数。

文件 *misc.c* 包含了在几个系统和内核调用中用到的一些函数，这些函数不适合放在其他的地方。

do_getsysinfo 是 *sys_datacopy* 内核调用的接口，它用于支持信息服务器（IS）的调试功能，IS 能向它请求文件系统各个数据结构的一份副本，这样就能把这些信息显示给用户。

系统调用 **dup** 复制一个文件描述符，换句话说，它创建一个新的文件描述符指向其参数所指定的文件。**dup2** 是 **dup** 的变体，它们都是由 *do_dup* 来处理的。实际上，这两个调用都已经过时，之所以把它们留在 MINIX 3 中，主要是为了支持老的二进制程序。在当前的 MINIX 3 版本中，如果在 C 源程序中遇到这两个调用，那么 C 的库函数将会把它们转换为相应的 **fcntl** 系统调用。

fcntl 调用是由 *do_fcntl* 函数来处理的，它是向一个打开文件请求某些操作的最佳方式。图 5.49 列出了一些 POSIX 定义的标志位，在请求服务时需要使用这些标志位来指明相应的操作。**fcntl** 的调用参数包括文件描述符、请求代码以及特定请求所需要的其他附加参数。例如，如果原先的 **dup** 调用为

```
dup2(fd, fd2);
```

那么相应的 **fcntl** 调用为

```
fcntl(fd, F_DUPFD, fd2);
```

在这些请求中，有几个用来设置或读取某个标志位，因此，相应的代码非常短，只有寥寥数行。例如，**F_SETFD** 请求设置一个标志位，使得某个文件的所有者进程在执行 **exec** 调用时强制关闭该文件；**F_GETFD** 请求则用来查询在调用 **exec** 时是否要关闭文件；**F_SETFL** 和 **F_GETFL** 请求可以设置标志位，表明某个文件可以用于非阻塞模式或者是追加操作。

操作	含义
F_DUPFD	复制文件描述符
F_GETFD	获取close-on-exec标志
F_SETFD	设置close-on-exec标志
F_GETFL	获取文件状态标志
F_SETFL	设置文件状态标志
F_GETLK	获取文件的锁状态
F_SETLK	设置文件的读/写锁
F_SETLKW	设置文件的写锁

图 5.49 用于 **fcntl** 系统调用的 POSIX 请求参数

*do_fcntl*还能处理文件锁。带有*F_GETLK*, *F_SETLK*或*F_SETLKW*命令的调用被翻译为相应的*lock_op*调用，我们在前面已经讨论过*lock_op*函数，这里就不再重复。

下一个系统调用是*sync*，它把内存中被修改过的所有块和*i*节点写回磁盘。这个调用在*do_sync*中处理。它对所有的表格进行搜索，找出修改过的项。*i*节点应该首先处理，因为*rw_inode*会把结果保留在块高速缓存中。当所有修改过的*i*节点都被写入高速缓存后，再把所有修改过的块都写回磁盘。

系统调用*fork*, *exec*, *exit*和*set*实际上都是进程管理器的调用，但它们运行的结果也可以在这里做一些说明。在*fork*一个进程时，内核、进程管理器和文件系统都应该知道这一点。这些“系统调用”并不来自于用户进程，而是来自于进程管理器。*do_fork*, *do_exit*和*do_set*会把相关信息记录在进程表的文件系统部分。*do_exec*会搜索并关闭（通过*do_close*）所有标志为closed-on-exec的文件。

*misc.c*中的最后一个函数*do_revive*并不是一个真正的系统调用，但在处理方式上有点类似。假设之前文件系统向一个设备驱动程序请求某个操作，如为一个用户进程提供输入数据，但当时该驱动程序无法完成这个操作，从而导致用户进程被阻塞。如果现在这个操作完成了，那么文件系统就要唤醒用户进程，并向它发送一条应答消息。

一个系统调用值得用一个头文件和相应的C源文件来支持它，*select.h*和*select.c*都是*select*系统调用的支持文件。当一个进程需要处理多个I/O流时，需要用到这个系统调用。关于*select*的详细内容，这里就不展开了。

5.7.9 文件系统的实用程序

文件系统还包含有一些通用的实用函数，在许多地方都要用到它们。这些函数被集中在文件*utility.c*中。

*clock_time*负责向系统任务发送消息，以查询当前的实时时间。

由于许多系统调用都以文件名作为参数，因此我们定义了*fetch_name*。如果文件名较短，它被包含在用户发送给文件系统的消息之中；如果文件名很长，则把指向用户空间中该文件名的指针放在消息中。*fetch_name*能处理这两种情形，并获取文件名。

在文件系统中，有两个函数用于处理一般类型的错误。当文件系统接收到错误的系统调用时会调用*no_sys*函数。而*panic*则在出现致命性错误时打印一条消息，并要求内核停止操作。类似的函数可以在进程管理器的源文件*pm/utility.c*中找到。

最后两个函数*conv2*和*conv4*用于帮助MINIX 3来处理不同的CPU系列之间字节顺序的差异问题。在读写磁盘数据结构如*i*节点或位图时会去调用这些函数。创建磁盘的系统所采用的字节顺序记录在超级块中，如果它与本地处理器所采用的字节顺序不同，那么就要交换顺序。这样，文件系统的其余部分就无须了解磁盘上的字节顺序。

最后，还有两个文件给文件管理器提供了专门的工具服务。文件系统可以要求系统任务为它设置一个警报，但如果它需要多个定时器，那么可以维护一条自己的定时器链表，就像我们在上一章看到的进程管理器的做法一样。文件*timers.c*为文件系统提供了这种支持。最后，MINIX 3实现了一种独特的使用CD-ROM的方式，能够用CD-ROM上的几个分区来隐藏一个模拟的MINIX 3磁盘，并支持从CD-ROM上启动MINIX 3系统。对于只支持标准的CD-ROM文件格式的操作系统来说，MINIX 3的文件是不可见的。因此，*cdprobe.c*文件用于在系统引导时，去定位一个CD-ROM设备及其上面的文件，以便于MINIX 3的启动。

5.7.10 其他的 MINIX 3 组件

上一章讨论的进程管理器和这一章讨论的文件系统都是用户空间的服务器，在常规的操作系统设计中，这两部分内容都集成在一个整体内核中。不过，它们并不是 MINIX 3 系统中仅有的服务器进程，实际上，还有其他一些用户空间的进程，它们具有系统的特权，应该被认为是操作系统的一部分。由于篇幅的限制，我们不可能深入地去讨论它们的内部机理，但至少要在这里稍微提一下。

第一个是前面提到过的再生服务器 (RS)，它能启动一个普通的进程并把它转换为一个系统进程。在 MINIX 3 的当前版本中，RS 主要用来启动未包含在系统引导映像中的设备驱动程序。在将来的版本中，它还应该能支持驱动程序的中止和重启，或者进一步说，它能监控驱动程序的运行，如果发现异常，就自动地中止它，然后让它重新启动。RS 的源代码位于 *src/servers/rs/* 目录下。

另一个曾经提到过的服务器是信息服务器 (IS)。它用来生成调试转储信息，用户可以通过在 PC 键盘上按下相应功能键来激活 IS 的运行。它的源代码位于 *src/servers/is/* 目录下。

信息服务器和再生服务器都是相对较小的程序，第三个可选的服务器是网络服务器 (INET)，它就比较大了。INET 程序的磁盘映像在大小上与 MINIX 3 的引导映像差不多。它是由再生服务器启动的，就像启动其他的设备驱动程序一样。网络服务器的源代码位于 *src/servers/inet/* 目录下。

我们要提到的最后一个系统组件并不是一个服务器，而是一个设备驱动程序，也就是 log 驱动程序。由于在操作系统中，许多不同的组件都是以独立的进程的形式来运行的，因此我们需要提供一种标准化的方式来处理诊断、警告和错误信息。MINIX 3 的办法是构造一个伪设备 */dev/klog* 和相应的设备驱动程序，它能够接收消息并把它们写入一个文件。log 驱动程序的源代码位于 *src/drivers/log/* 目录下。

5.8 小结

从外部来看，文件系统是一组文件和目录，以及针对它们的各种操作。文件可以被读和写，目录可以被创建和删除，可以将文件从一个目录移动到另一个目录。大多数现代文件系统都支持层次目录结构，在这种结构中，每个目录都可以包含子目录，如此循环，直至无穷。

而从内部来看，文件系统却迥然不同。文件系统的设计者必须考虑存储空间如何分配、系统如何来管理每个文件所使用的数据块等。我们还看到，不同的文件系统具有不同的目录结构。此外，文件系统的可靠性和性能也是一个重要问题。

文件系统的安全和保护对于系统用户和系统设计者来说都是至关重要的。我们讨论了老式系统中的一些安全缺陷和大多数系统具有的一些共同问题，还讨论了身份认证、有无口令、访问控制表、权能以及矩阵模型等内容。

最后，我们详细地研究了 MINIX 3 文件系统。MINIX 3 文件系统很大，但并不复杂。它从用户进程接收请求，以此为索引去访问函数指针表，然后调用相应的函数来执行所要求的系统调用。由于文件系统的模块化结构以及身处内核之外，因此可以把它从 MINIX 3 中删除，然后经过一点小小的修改，就可以把它用做一个独立的网络文件服务器。

在系统内部，MINIX 3 会将数据存放在块高速缓存中，并在顺序访问文件时采用预读机制。如果高速缓存足够大，那么在反复访问某些程序（如编译器）时，大多数的程序代码都可以在内存中找到。

习题

1. NTFS 使用 Unicode 来命名文件，Unicode 支持 16 位字符。与传统的 ASCII 文件名相比，这种方式有何优点？
2. 有些文件以一个魔数开头，它有什么用？
3. 图 5.4 列出了一些文件属性，但其中没有包含奇偶校验。请问这是一个有用的文件属性吗？如果是，它应该如何使用？
4. 请给出文件 /etc/passwd 的 5 种不同的路径名。提示：考虑目录项 “.” 和 “..”。
5. 在支持顺序文件的系统中通常有文件回绕操作。请问支持随机访问文件的系统是否也需要该操作？
6. 某些操作系统提供系统调用 rename 来给文件改名。而另一种方式是把文件复制为一个新文件，然后删除原文件，从而实现文件的更名。请问这两种方法有何不同？
7. 考虑图 5.7 中的目录树。如果当前工作目录是 /usr/jim/，那么相对路径名为 ./ast/x 的文件的绝对路径名是什么？
8. 考虑如下的提议：在文件系统中不是设置一个唯一的根目录，而是为每个用户都设置一个个人的根目录。请问这种做法能使系统变得更灵活吗？为什么？
9. UNIX 文件系统有一个 chroot 调用，能够把根目录修改为某个指定的目录。请问这种做法有无安全方面的含意？如果有，是什么？
10. UNIX 系统有一个调用来读取一个目录项。由于目录本身就是一个文件，那么为什么还要定义一个专门的调用呢？用户自己能否直接去读取目录文件呢？
11. 一台标准的 PC 机最多只能同时包含四个操作系统，你有没有办法增加这个限额？你的提议会有什么样的后果？
12. 正如书中所提到的，文件的连续分配会导致磁盘碎片。请问这是内碎片还是外碎片？请把它与前一章的内容进行比较。
13. 图 5.10 显示了 MS-DOS 使用的早期 FAT 文件系统的结构。最初，这个文件系统只有 4096 个块，因此一个 4096 行（12 位）的 FAT 表就够用了。如果把这种方案直接扩展到具有 2^{32} 个块的文件系统中，那么 FAT 表需要占用多大的存储空间？
14. 如果一个操作系统只支持单一目录，但允许该目录具有任意多个文件，而且文件名可以任意长。请问该系统能否模拟一个层次结构的文件系统？如何模拟？
15. 空闲磁盘空间可以用空闲链表或位图来管理。假设磁盘地址需要 D 位，某个磁盘有 B 个块，其中 F 个空闲。请问，在什么条件下，使用空闲链表所占用的空间要少于位图？如果 D 为 16，请用空闲磁盘空间的百分比来表示你的答案。
16. 有人建议每个 UNIX 文件的第一部分最好和它的 i 节点存放在同一个磁盘块中。这样做有什么好处？
17. 文件系统的性能取决于高速缓存的命中率（即在高速缓存中找到所需块的概率）。假设从高速缓存中读取数据需要 1 ms，而从磁盘上读取则需要 40 ms。如果命中率为 h ，请给出读取数据所需要的平均时间的计算公式，并画出 h 从 0 到 1.0 变化时的函数曲线。
18. 硬链接和符号链接之间的区别是什么？每种方案各有什么优点？
19. 在备份一个文件系统时，有哪些隐患需要密切注意？请列出其中的三个。
20. 一个磁盘有 4000 个柱面，每个柱面有 8 个磁道，每个磁道有 512 个块。在寻道时每移过一个柱面需要 1 ms。如果不采取措施使文件的数据块在磁盘上尽量紧靠，那么逻辑上相邻的

两个块所需要的平均寻道时间为 5 ms。另一种情况是操作系统将相邻的块尽量放在一起，此时块间的平均距离为 2 个柱面，而寻道时间缩减为 100 μ s。假设旋转延迟为 10 ms，传输速率为每块 20 μ s。请问，在这两种情况下，读取一个 100 块的文件各需要多长时间？

21. 定期对磁盘空间进行紧缩操作可以带来什么好处？为什么？
22. 病毒与蠕虫之间的区别是什么？它们是如何繁殖的？
23. 在取得学位后，你去应聘某大学计算中心主任的职位，该中心刚刚将一种很老的操作系统转换为 UNIX。后来你得到了这份工作，在刚开始工作后 15 min，你的助手冲进你的办公室，说有些学生已经发现了系统使用的口令加密算法，并将其公布在 Internet 上。此时你应该怎么办？
24. 两名计算机系的学生 Carolyn 和 Elinor，正在讨论 i 节点的话题。Carolyn 认为，由于内存容量变得越来越大，价格越来越便宜，因此，在打开一个文件后，比较简单而快捷的做法是直接把一份新的 i 节点副本装入到 i 节点表中，而不用去搜索整个表，看它是否已经在那儿。Elinor 不同意这种观点。你认为谁对谁错？
25. Morris-Thompson 保护机制使用 n 位随机数，可以防止入侵者通过事先对普通字符串进行加密来发现大量的口令。这种机制对于防止一个学生去猜测他机器上的超级用户口令是否奏效？
26. 在某个计算机系的局域网中连接了许多台 UNIX 机器。任何一台机器上的用户均可键入命令
`machine4 who`
并使之在 *machine4* 上运行，而且用户不必事先登录到这台远程主机上。在具体实现这个特性时，主要是让用户的内核将该条命令和用户 UID 发送到远程主机上。如果内核是可信的（例如，具有硬件保护的分时小型机），那么以上这种方案是否安全？如果有些机器是学生的 PC 机，不带任何硬件保护，那么情况又会如何？
27. 当一个文件被删除时，它的块通常会返回到空闲链表中，但不会被清除。你认为操作系统是否应该在释放一个块之前先把它清除？请从安全和性能两个方面来考虑，并解释各自的影响。
28. 我们讨论了三种不同的保护机制：权能、访问控制表以及 UNIX 的 *rwx* 位。下面的问题分别适用于哪一种机制？
 - (a) Ken 希望除了他的同事以外，任何人都能读取他的文件。
 - (b) Mitch 和 Steve 希望共享一些秘密文件。
 - (c) Linda 希望她的一部分文件是对外公开的。对于 UNIX，假设用户的分组包括教职工、学生和秘书。
29. 在采用权能保护方式的系统中，特洛伊木马攻击能否奏效？
30. *flip* 表的长度当前被定义为一个常量，即文件 *fs/const.h* 中的 *NR_FILPS*。为了在一个联网系统中容纳更多的用户，你希望增大文件 *include/minix/config.h* 中的 *NR_PROCS*。请问，如何把 *NR_FILPS* 定义为 *NR_PROCS* 的一个函数？
31. 假设出现了一项重大的技术突破，诞生了非易失型的 RAM（即在掉电时也不会丢失其内容的 RAM），而且在价格和性能上也不比常规的 RAM 差。请问这项进展对文件系统的设计有何影响？
32. 符号链接是间接地指向其他文件或目录的文件。与当前 MINIX 3 实现的普通链接不同，符号链接有自己的 i 节点，该 i 节点指向一个数据块，在数据块中包含有被链接文件的路径名。

而且i节点的存在使得该链接可以有不同于被链接文件的所有者和权限。符号链接和它所指向的文件或目录可以位于不同的设备上。符号链接不是MINIX 3的一部分，请在MINIX 3中实现符号链接。

33. 当前对一个MINIX 3文件长度的限制主要是由32位的文件指针来决定的，但是在将来，随着64位文件指针的引入，可能会出现长度大于 $2^{32} - 1$ 个字节的文件，这时可能就需要用到三级间接块。请修改文件系统，并实现三级间接块。
34. 验证ROBUST标志位的设置是会使文件系统在系统崩溃时变得更加健壮还是更加脆弱。这个问题在当前版本的MINIX 3中还未曾研究，因此这两种情况都有可能。仔细研究一下当一个修改过的块被换出高速缓存时所发生的过程。注意，如果某个数据块被修改，那么可能会伴随着i节点和位图的修改。
35. 设计一种机制，增加对“外部”文件系统的支持，这样我们就可以在MINIX 3文件系统的目录中挂装其他的文件系统，如MS-DOS的文件系统。
36. 用C或shell脚本写一组程序，通过MINIX 3系统上的秘密通道来发送和接收消息。提示：即使一个文件是不可访问的，我们仍然可以看到它的一个权限位。此外，通过设置相应的参数，sleep命令或系统调用能够保证延迟固定的一段时间。测量一个空闲系统的数据率，然后通过启动许多个不同的后台进程来构造一个人为的重负载状态，并测量此时的数据率。
37. 在MINIX 3中实现立即文件，即对于小文件，直接把它的内容存放在它的i节点中，这样，在访问该文件时，能够减少一次磁盘访问。



第6章 阅读材料和参考文献

- 6.1 推荐的进一步阅读材料
- 6.2 按字母顺序排列的参考文献

在前5章中我们已经对操作系统的许多内容做了介绍，本章旨在为那些希望对操作系统做进一步研究的读者提供一些帮助。6.1节列举了向读者推荐的阅读材料，6.2节按照字母顺序列出了本书中引用的所有书籍和论文。

除了下面列出的参考文献之外，ACM每年举办的操作系统原理专题研讨会论文集 *Proceedings of the n-th ACM Symposium on Operating Systems Principles*，以及 IEEE每年举办的分布式计算系统国际会议论文集 *Proceedings of the n-th International Conference on Distributed Computing System*，都是查阅操作系统最新论文的好去处。同样，USENIX操作系统设计和实现专题研讨会也是一个很好的信息源。另外，*ACM Transactions on Computer Systems* 和 *Operating Systems Review* 这两本期刊也经常登载相关的文章。

6.1 推荐的进一步阅读材料

以下是推荐的进一步阅读材料，按章节形式组织。

6.1.1 介绍和概论

Bovet and Cesati, *Understanding the Linux Kernel*, 3rd Ed.

如果你想深入了解 Linux 内核的内部机理，这本书可能是最好的选择。

Brinch Hansen, *Classic Operating Systems*

在操作系统的发展历史中，有一些被认为是经典：即那些改变了人们对计算机的看法的系统。本书由 24 篇论文组成，介绍了操作系统领域内的一些开创性工作，分为开放式计算站、批处理、多道程序设计、分时、个人计算机和分布式操作系统。对操作系统的历史感兴趣的人都应该去阅读这本书。

Brooks, *The Mythical Man-Month: Essays On Software Engineering*

一本机智、幽默并且信息量很大的著作，关于如何避免像某些人那样以一种很困难的方式来编写操作系统，本书中有很多好的建议。

Corbató, “On Building Systems That Will Fail”

Corbató 被誉为分时系统之父，在他的图灵奖颁奖致词中，谈到了许多 Brooks 在其著作 *The Mythical Man-Month* 中所述及的问题。Corbató 的结论是所有的复杂系统最终都会失败，而如果想要成功，最重要的就是必须摈弃复杂性，在设计中尽量做到简洁而优雅。

Deitel et al, *Operating Systems*, 3rd Ed.

一本操作系统的通用性教材。在标准内容之外，本书还添加了一些实例研究，如 Linux 和 Windows XP。

Dijkstra, “My Recollections of Operating System Design”

一位操作系统设计先驱对往事的缅怀，把我们带回到“操作系统”这个术语还未出现的日子。

IEEE, *Information Technology — Portable Operating System Interface (POSIX)*,

Part 1: System Application Program Interface(API) [C Language]

这是一个标准。其中有些部分非常易读，尤其是附件B：“合理性和注释”，它对为什么采取这种方法来解决问题做了许多说明。引用标准的好处之一是：从定义上来讲，它是没有错误的。如果在排版时写错了一个宏定义的名字，但侥幸逃过了编辑大人的法眼，那么它就不再是一个错误了，而是官方发布的标准。

Lampson, “Hints for Computer System Design”

Butler Lampson是当今创新型操作系统设计的领军人物之一。他收集了多年实际经验中的种种启示、建议和指南，把它们汇集在这篇睿智并且信息量很大的文章中。与 Brooks 的书一样，这也是每个优秀的系统设计人员的必读材料。

Lewine, *POSIX Programmer's Guide*

这本书以一种更具可读性的方式描述了 POSIX 标准，同时还讨论了如何将老的程序转换到 POSIX，以及如何在 POSIX 环境下开发新程序。书中有很多的代码实例，包括几个完整的程序。此外，本书还对 POSIX 要求的所有库函数和头文件进行了描述。

McKusick and Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*

本书对 UNIX 的一个现代版本 FreeBSD 进行了深入的阐述，包括进程、I/O、存储管理、网络等方面的所有内容。

Milojicic, “Operating Systems: Now and in the Future”

如果你去咨询世界上操作系统领域内的六个顶尖人物，问问他们这个领域的当前现状和未来发展，那么你得到的答案会是一样的吗？提示：不！请阅读这篇文章，看看他们都说了些什么。

Ray and Ray, *Visual Quickstart Guide: UNIX*, 2nd Ed.

如果你是一名 UNIX 用户，那么就能更好地理解本书中的一些例子。本书是众多的初学者指南中的一本，它指导读者如何去使用 UNIX 操作系统。尽管在实现上各不相同，但对于用户来说，MINIX 和 UNIX 是差不多的，因此本书和其他类似的书籍将有助于你去学习 MINIX 系统。

Russinovich and Solomon, *Microsoft Windows Internals*, 4th Ed.

想知道 Windows 的内部工作原理吗？本书将满足你的好奇心，你想知道的一切内容都在其中，包括进程、存储管理、I/O、网络和安全等。

Silberschatz et al, *Operating System Concepts*, 7th Ed.

操作系统的另一本教材，它涵盖了进程、存储器管理、文件和分布式系统。此外，还包含了两个实例：Linux 和 Windows XP。

Stallings, *Operating Systems*, 5th Ed.

操作系统的又一本教材。它包含了通常的操作系统内容，也包含了一些分布式系统的内容。此外，在附录中还介绍了一些排队理论。

Stevens and Rago, *Advanced Programming in the UNIX Environment*, 2nd Ed.

本书讲述如何使用UNIX系统调用接口和标准C语言库来编写C程序。书中的例子在以下的系统中经过了测试：FreeBSD 5.2.1、Linux 2.4.22内核、Solaris 9、Darwin 7.4.0，以及基于FreeBSD/Mach的Mac OS X 10.3。书中详细讨论了这些系统与POSIX之间的关系。

6.1.2 进程

Andrews and Schneider, “Concepts and Notations for Concurrent Programming”

本文综述了进程和进程间通信，包括忙等待、信号量、管程、消息传递以及其他技术。同时也说明了这些概念如何嵌入在不同的程序设计语言中。

Ben-Ari, *Principles of Concurrent and Distributed Programming*

这本书由三部分组成。第一部分讨论了互斥、信号量、管程以及哲学家进餐问题等内容；第二部分讨论了分布式程序设计和编程语言；第三部分是并发性的实现原理。

Bic and Shaw, *Operating System Principles*

这本操作系统教材用了四章的篇幅来讨论进程，包括它的基本原理以及实现细节。

Milo et al., “Process Migration”

当前，超级计算机正逐渐被PC群组所取代，因此，将进程从一台机器移动到另一台（例如，为了负载均衡）的问题变得越来越重要。在这篇综述中，作者讨论了进程迁移的工作原理以及它的优缺点。

Silberschatz et al, *Operating System Concepts*, 7th Ed.

本书的第3章到第7章讨论了进程和进程间通信，包括调度、临界区、信号量、管程以及经典的进程间通信问题。

6.1.3 输入/输出

Chen et al., “RAID: High Performance Reliable Secondary Storage”

在高端系统中，一个发展趋势是使用多个磁盘驱动器并行操作来提高输入/输出的性能。作者就此进行了讨论并从性能、价格和可靠性等方面研究了不同的组织结构。

Coffman et al., “System Deadlocks”

本文简短地介绍了死锁、死锁的产生以及死锁的预防和检测。

Corbet et al., *Linux Device Drivers*, 3rd Ed.

如果你真的想知道I/O的工作原理，最好的办法是去写一个设备驱动程序。本书将告诉你如何在Linux系统中编写设备驱动程序。

Geist and Daniel, “A Continuum of Disk Scheduling Algorithms”

本文给出了一个通用的磁盘臂调度算法，并给出了详细的模拟和实验结果。

Holt, “Some Deadlock Properties Of Computer Systems”

本文讨论了死锁。Holt引入了一个有向图模型，可以用来分析某些死锁的情形。

IEEE Computer Magazine, March 1994

这一期杂志包含了8篇关于先进的I/O系统的文章，其中涉及模拟、高性能存储器、高速缓存、并行计算机的I/O以及多媒体。

Levine, “Defining Deadlocks”

在这篇短文中, Levine 针对死锁的传统定义和案例提出了一些有趣的问题。

Swift et al., “Recovering Device Drivers”

设备驱动程序的错误率比其他的操作系统代码要高一个数量级,那么有什么办法能够提高驱动程序的可靠性呢?本文介绍了如何使用影子驱动程序来实现这个目标。

Tsegaye and Foss, “A Comparison of the Linux and Windows Device Driver Architecture”

Linux 和 Windows 的设备驱动程序具有不同的体系结构,本文对两者进行了讨论,指出了它们之间的相似性和差别所在。

Wilkes et al., “The HP AutoRAID Hierarchical Storage System”

RAID (Redundant Array of Inexpensive Disks, 廉价磁盘冗余阵列) 是高性能磁盘系统的一项重要进展。它的基本思路是:由若干个小磁盘构成的阵列一起工作,以构造一个高带宽的系统。在本文中,作者详细描述了他们在 HP 实验室里开发的系统。

6.1.4 存储管理

Bic and Shaw, *Operating System Principles*

本书用三章的篇幅来介绍存储管理、物理内存、虚拟内存和共享内存。

Denning, “Virtual Memory”

本文是关于虚拟存储器的经典文章,讨论了虚拟存储器的诸多方面。Denning 是该领域的先驱之一,也是工作集概念的发明人。

Denning, “Working Sets Past and Present”

本文很好地综述了众多的存储管理机制和页面置换算法,并附有完整的参考文献。

Halpern, “VIM: Taming Software with Hardware”

在这篇有争议的文章中, Halpern 认为大量的金钱被花在了内存优化软件的编写、调试和维护上,而且不仅在操作系统中如此,在编译器和其他软件中也是如此。他认为,从宏观经济学的角度来看,与其如此,还不如用这笔钱来购买更多的内存,从而实现更简单、更可靠的软件。

Knuth, *The Art Of Computer Programming*, Vol.1

本书讨论并比较了最先匹配法、最佳匹配法和其他的存储管理算法。

Silberschatz et al, *Operating System Concepts*, 7th Ed.

本书的第 8 章和第 9 章讨论存储管理问题,包括交换、页式和段式存储管理,其中提到了很多页面置换算法。

6.1.5 文件系统

Denning, “The United States vs. Craig Neidorf”

当一个年轻的黑客发现并发表了电话网系统的工作原理之后,他被指控为计算机诈骗。本文就讲述了一个这样的官司,其中牵涉到很多基本的问题,如言论自由。本文发表后引起了一些非议, Denning 也进行了反驳。

Ghemawat et al., “The Google File System”

如果你决定把整个 Internet 的内容都存储在家里面，以便能快速地查找信息，那么你打算怎么做？第一步，你需要购买 20 万台 PC 机。不需要功能特强大的机器，只要普通的 PC 机即可。第二步，阅读本文，看看 Google 是怎么做的。

Hafner and Markoff, *Cyberpunk: Outlaws and Hackers on the Computer Frontier*

本书讲述了三个引人入胜的故事，讲的是年轻的黑客们闯入了世界各地的计算机。本书的作者之一是纽约时报的一名计算机记者，他自己曾攻破了 Internet 上的蠕虫病毒。

Harbron, *File Systems: Structures and Algorithms*

本书讨论了文件系统的设计、应用和性能，对结构和算法均做了介绍。

Harris et al., *Gray Hat Hacking: The Ethical Hacker’s Handbook*

本书讨论了计算机系统攻击的法律和道德方面的内容，并且提供了一些技术信息，表明系统的安全漏洞是如何产生和检测的。

McKusick et al., “A Fast File System for UNIX”

在 4.2 BSD 中，UNIX 的文件系统被完全重新实现。本文描述了新文件系统的设计，并讨论了它的性能。

Satyanarayanan, “The Evolution of Coda”

随着移动计算变得越来越普遍，移动文件系统与固定文件系统之间的集成和同步变得越来越重要。Coda 曾经是这个领域内的一名先锋，本文描述了它的发展和运作。

Silberschatz et al. *Operating System Concepts*, 7th Ed.

本书第 10 章和第 11 章讲的是文件系统，其中包括文件的操作、访问方式、一致性语义、目录、保护和实现等内容。

Stallings, *Operating Systems*, 5th Ed.

本书第 16 章讲了很多有关安全环境方面的内容，特别是黑客、病毒和其他威胁。

Uppuluri et al., “Preventing Race Condition Attacks on File Systems”

存在这样一种情形：一个进程认为某两个操作将被顺序执行，中间不会被打断。这时，如果另一个进程设法插了进来，在这两个操作之间加入了另一个操作，那么就可能会出现安全漏洞。本文讨论了这个问题，并提出了一种解决方案。

Yang et al., “Using Model Checking to Find Serious File System Errors”

文件系统的错误可能会导致数据丢失，因此对这些错误进行调试是非常重要的。本文描述了一种形式化技术，可以帮助人们检测文件系统中的错误，从而避免这些错误对系统造成伤害。本文将这个模型检查器用在了实际的文件系统代码上，并给出了实验结果。

6.2 按字母顺序排列的参考文献

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,” *ACM Trans. on Computer Systems*, vol. 10, pp. 53-79, Feb. 1992.

- ANDREWS, G.R., and SCHNEIDER, F.B.:** "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, March 1983.
- AYCOCK, J., and BARKER, K.:** "Viruses 101," *Proc. Tech. Symp. on Comp. Sci. Education*, ACM, pp. 152-156, 2005.
- BACH, M.J.:** *The Design of the UNIX Operating System*, Upper Saddle River, NJ: Prentice Hall, 1987.
- BALA, K., KAASHOEK, M.F., and WEIHL, W.:** "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Oper. Syst. Design and Implementation*, USENIX, pp. 243-254, 1994.
- BASILI, V.R., and PERRICONE, B.T.:** "Software errors and Complexity: An Empirical Investigation," *Commun. of the ACM*, vol. 27, pp. 43-52, Jan. 1984.
- BAYS, C.:** "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Commun. of the ACM*, vol. 20, pp. 191-192, March 1977.
- BEN-ARI, M.:** *Principles of Concurrent and Distributed Programming*, Upper Saddle River, NJ: Prentice Hall, 1990.
- BIC, L.F., and SHAW, A.C.:** *Operating System Principles*, Upper Saddle River, NJ: Prentice Hall, 2003.
- BOEHM, H.-J.:** "Threads Cannot be Implemented as a Library," *Proc. 2004 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, ACM, pp. 261-268, 2005.
- BOVET, D.P., and CESATI, M.:** *Understanding the Linux Kernel*, 2nd Ed., Sebastopol, CA, O'Reilly, 2002.
- BRINCH HANSEN, P.:** *Operating System Principles* Upper Saddle River, NJ: Prentice Hall, 1973.
- BRINCH HANSEN, P.:** *Classic Operating Systems*, New York: Springer-Verlag, 2001.
- BROOKS, F. P., Jr.:** *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Ed., Boston: Addison-Wesley, 1995.
- CERF, V.G.:** "Spam, Spim, and Spit," *Commun. of the ACM*, vol. 48, pp. 39-43, April 2005.
- CHEN, H., WAGNER, D., and DEAN, D.:** "Setuid Demystified," *Proc. 11th USENIX Security Symposium*, pp. 171-190, 2002.
- CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., and PATTERSON, D.A.:** "RAID: High Performance Reliable Secondary Storage," *Computing Surveys*, vol. 26, pp. 145-185, June 1994.
- CHERITON, D.R.:** "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Review*, vol. 18, pp. 12-20, Oct. 1984.
- CHERVENAK, A., VELLANSKI, V., and KURMAS, Z.:** "Protecting File Systems: A Survey of Backup Techniques," *Proc. 15th Symp. on Mass Storage Systems*, IEEE, 1998.
- CHOU, A., YANG, J.-F., CHELF, B., and HALLEM, S.:** "An Empirical Study of Operating System Errors," *Proc. 18th Symp. on Oper. Syst. Prin.*, ACM, pp. 73-88, 2001.
- COFFMAN, E.G., ELPHICK, M.J., and SHOSHANI, A.:** "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, June 1971.

- CORBATO', F.J.**: "On Building Systems That Will Fail," *Commun. of the ACM*, vol. 34, pp. 72-81, Sept. 1991.
- CORBATO', F.J., MERWIN-DAGGETT, M., and DALEY, R.C.**: "An Experimental Time-Sharing System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335-344, 1962.
- CORBATO', F.J., SALTZER, J.H., and CLINGEN, C.T.**: "MULTICS—The First Seven Years," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571-583, 1972.
- CORBATO', F.J., and VYSSOTSKY, V.A.**: "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185-196, 1965.
- CORBET, J., RUBINI, A., and KROAH-HARTMAN, G.**: *Linux Device Drivers*, 3rd Ed. Sebastopol, CA: O'Reilly, 2005.
- COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.**: "Concurrent Control with Readers and Writers," *Commun. of the ACM*, vol. 10, pp. 667-668, Oct. 1971.
- DALEY, R.C., and DENNIS, J.B.**: "Virtual Memory, Processes, and Sharing in MULTICS," *Commun. of the ACM*, vol. 11, pp. 306-312, May 1968.
- DEITEL, H.M., DEITEL, P. J., and CHOFFNES, D. R.** : *Operating Systems*, 3rd Ed., Upper Saddle River, NJ: Prentice-Hall, 2004.
- DENNING, D.**: "The United states vs. Craig Neidorf," *Commun. of the ACM*, vol. 34, pp. 22-43, March 1991.
- DENNING, P.J.**: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, 1968a.
- DENNING, P.J.**: "Thrashing: Its Causes and Prevention," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915-922, 1968b.
- DENNING, P.J.**: "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, Sept. 1970.
- DENNING, P.J.**: "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64-84, Jan. 1980.
- DENNING, P.J.**: "The Locality Principle," *Commun. of the ACM*, vol. 48, pp. 19-24, July 2005.
- DENNIS, J.B., and VAN HORN, E.C.**: "Programming Semantics for Multiprogrammed Computations," *Commun. of the ACM*, vol. 9, pp. 143-155, March 1966.
- DIBONA, C., OCKMAN, S., and STONE, M.** eds.: *Open Sources: Voices from the Open Source Revolution*, Sebastopol, CA: O'Reilly, 1999.
- DIJKSTRA, E.W.**: "Co-operating Sequential Processes," in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.
- DIJKSTRA, E.W.**: "The Structure of THE Multiprogramming System," *Commun. of the ACM*, vol. 11, pp. 341-346, May 1968.
- DIJKSTRA, E.W.**: "My Recollections of Operating System Design," *Operating Systems Review*, vol. 39, pp. 4-40, April 2005.
- DODGE, C., IRVINE, C., and NGUYEN, T.**: "A Study of Initialization in Linux and OpenBSD," *Operating Systems Review*, vol. 39, pp. 79-93 April 2005.

- ENGLER, D., CHEN, D.Y., and CHOU, A.**: "Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. 18th Symp. on Oper. Syst. Prin.*, ACM, pp. 57-72, 2001.
- ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J. Jr.**: "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th Symp. on Oper. Syst. Prin.*, ACM, pp. 251-266, 1995.
- FABRY, R.S.**: "Capability-Based Addressing," *Commun. of the ACM*, vol. 17, pp. 403-412, July 1974.
- FEELEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEKKATH, C.A.**: "Implementing Global Memory Management in a Workstation Cluster," *Proc. 15th Symp. on Oper. Syst. Prin.*, ACM, pp. 201-212, 1995.
- FEUSTAL, E.A.**: "The Rice Research Computer—A Tagged Architecture," *Proc. AFIPS Conf.* 1972.
- FOTHERINGHAM, J.**: "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Oct. 1961.
- GARFINKEL, S.L., and SHELAT, A.**: "Remembrance of Data Passed: A Study of Disk Sanitization Practices," *IEEE Security & Privacy*, vol. 1, pp. 17-27, Jan.-Feb. 2003.
- GEIST, R., and DANIEL, S.**: "A Continuum of Disk Scheduling Algorithms," *ACM Trans. on Computer Systems*, vol. 5, pp. 77-92, Feb. 1987.
- GHEMAWAT, S., GOBIOFF, H., and LEUNG, S.-T.**: "The Google File System," *Proc. 19th Symp. on Oper. Syst. Prin.*, ACM, pp. 29-43, 2003.
- GRAHAM, R.**: "Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T., Sept. 1970.
- HAFNER, K., and MARKOFF, J.**: *Cyberpunk: Outlaws and Hackers on the Computer Frontier*, New York: Simon and Schuster, 1991.
- HALPERN, M.**: "VIM: Taming Software with Hardware," *IEEE Computer*, vol. 36, pp. 21-25, Oct. 2003.
- HARBRON, T.R.**: *File Systems: Structures and Algorithms*, Upper Saddle River, NJ: Prentice Hall, 1988.
- HARRIS, S., HARPER, A., EAGLE, C., NESS, J., and LESTER, M.**: *Gray Hat Hacking: The Ethical Hacker's Handbook*, New York: McGraw-Hill Osborne Media, 2004.
- HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.**: "Using Threads in Interactive Systems: A Case Study," *Proc. 14th Symp. on Oper. Syst. Prin.*, ACM, pp. 94-105, 1993.
- HEBBARD, B. et al.**: "A Penetration Analysis of the Michigan Terminal System," *Operating Systems Review*, vol. 14, pp. 7-20, Jan. 1980.
- HERBORTH, C.**: *UNIX Advanced: Visual Quickpro Guide*, Berkeley, CA: Peachpit Press, 2005
- HERDER, J.N.**: "Towards a True Microkernel Operating System," M.S. Thesis, Vrije Universiteit, Amsterdam, Feb. 2005.
- HOARE, C.A.R.**: "Monitors, An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549-557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.

- HOLT, R.C.**: "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179-196, Sept. 1972.
- HUCK, J., and HAYS, J.**: "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. 20th Annual Int'l Symp. on Computer Arch.*, ACM, pp. 39-50, 1993.
- HUTCHINSON, N.C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S, and O'MALLEY, S.**: "Logical vs. Physical File System Backup," *Proc. Third USENIX Symp. on Oper. Syst. Design and Implementation*, USENIX, pp. 239-249, 1999.
- IEEE**: *Information technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York: IEEE, 1990.
- JACOB, B., and MUDGE, T.**: "Virtual Memory: Issues of Implementation," *IEEE Computer*, vol. 31, pp. 33-43, June 1998.
- JOHANSSON, J., and RILEY, S.**: *Protect Your Windows Network: From Perimeter to Data*, Boston: Addison-Wesley, 2005.
- KERNIGHAN, B.W., and RITCHIE, D.M.**: *The C Programming Language*, 2nd Ed., Upper Saddle River, NJ: Prentice Hall, 1988.
- KLEIN, D.V.**: "Foiling the Cracker: A Survey of, and Improvements to, Password Security," *Proc. UNIX Security Workshop II*, USENIX, Aug. 1990.
- KLEINROCK, L.**: *Queueing Systems, Vol. 1*, New York: John Wiley, 1975.
- KNUTH, D.E.**: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed., Boston: Addison-Wesley, 1997.
- LAMPSON, B.W.**: "A Scheduling Philosophy for Multiprogramming Systems," *Commun. of the ACM*, vol. 11, pp. 347-360, May 1968.
- LAMPSON, B.W.**: "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613-615, Oct. 1973.
- LAMPSON, B.W.**: "Hints for Computer System Design," *IEEE Software*, vol. 1, pp. 11-28, Jan. 1984.
- LEDIN, G., Jr.**: "Not Teaching Viruses and Worms is Harmful," *Commun. of the ACM*, vol. 48, p. 144, Jan. 2005.
- LESCHKE, T.**: "Achieving Speed and Flexibility by Separating Management from Protection: Embracing the Exokernel Operating System," *Operating Systems Review*, vol. 38, pp. 5-19, Oct. 2004.
- LEVINE, G.N.**: "Defining Deadlocks," *Operating Systems Review* vol. 37, pp. 54-64, Jan. 2003a.
- LEVINE, G.N.**: "Defining Deadlock with Fungible Resources," *Operating Systems Review*, vol. 37, pp. 5-11, July 2003b.
- LEVINE, G.N.**: "The Classification of Deadlock Prevention and Avoidance is Erroneous," *Operating Systems Review*, vol. 39, 47-50, April 2005.
- LEWINE, D.**: *POSIX Programmer's Guide*, Sebastopol, CA: O'Reilly & Associates, 1991.

- LI, K., and HUDAQ, P.**: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Nov. 1989.
- LINDE, R.R.**: "Operating System Penetration," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 361-368, 1975.
- LIONS, J.**: *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peer-to-Peer Communications, 1996.
- MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., and MARKATOS, E.P.**: "First-Class User-Level Threads," *Proc. 13th Symp. on Oper. Syst. Prin.*, ACM, pp. 110-121, 1991.
- McHUGH, J.A.M., and DEEK, F.P.**: "An Incentive System for Reducing Malware Attacks," *Commun. of the ACM*, vol. 48, pp. 94-99, June 2005.
- McKUSICK, M.K., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.**: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Aug. 1984.
- McKUSICK, M.K., and NEVILLE-NEIL, G.V.**: *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley: Boston, 2005.
- MILO, D., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., and ZHOU, S.**: "Process Migration," *ACM Computing Surveys*, vol. 32, pp. 241-299, July-Sept. 2000.
- MILOJICIC, D.**: "Operating Systems: Now and in the Future," *IEEE Concurrency*, vol. 7, pp. 12-21, Jan.-March 1999.
- MOODY, G.**: *Rebel Code* Cambridge, MA: Perseus, 2001.
- MORRIS, R., and THOMPSON, K.**: "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594-597, Nov. 1979.
- MULLENDER, S.J., and TANENBAUM, A.S.**: "Immediate Files," *Software—Practice and Experience*, vol. 14, pp. 365-368, April 1984.
- NAUGHTON, J.**: *A Brief History of the Future*, Woodstock, NY: Overlook Books, 2000.
- NEMETH, E., SNYDER, G., SEEBASS, S., and HEIN, T. R.**: *UNIX System Administration*, 3rd Ed., Upper Saddle River, NJ, Prentice Hall, 2000.
- ORGANICK, E.I.**: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.
- OSTRAND, T.J., WEYUKER, E.J., and BELL, R.M.**: "Where the Bugs Are," *Proc. 2004 ACM Symp. on Softw. Testing and Analysis*, ACM, 86-96, 2004.
- PETERSON, G.L.**: "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115-116, June 1981.
- PRECHELT, L.**: "An Empirical Comparison of Seven Programming Languages," *IEEE Computer*, vol. 33, pp. 23-29, Oct. 2000.
- RAY, D.S., and RAY, E.J.**: *Visual Quickstart Guide: UNIX*, 2nd Ed., Berkeley, CA: Peachpit Press, 2003.
- ROSENBLUM, M., and OUSTERHOUT, J.K.**: "The Design and Implementation of a Log-Structured File System," *Proc. 13th Symp. on Oper. Syst. Prin.*, ACM, pp. 1-15, 1991.
- RUSSINOVICH, M.E., and SOLOMON, D.A.**: *Microsoft Windows Internals*, 4th Ed., Redmond, WA: Microsoft Press, 2005.
- SALTZER, J.H.**: "Protection and Control of Information Sharing in MULTICS," *Commun. of the ACM*, vol. 17, pp. 388-402, July 1974.

- SALTZER, J.H., and SCHROEDER, M.D.**: "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, pp. 1278-1308, Sept. 1975.
- SALUS, P.H.**: *A Quarter Century of UNIX*, Boston: Addison-Wesley, 1994.
- SANDHU, R.S.**: "Lattice-Based Access Control Models," *Computer*, vol. 26, pp. 9-19, Nov. 1993.
- SATYANARAYANAN, M.**: "The Evolution of Coda," *ACM Trans. on Computer Systems*, vol. 20, pp. 85-124, May 2002.
- SEAWRIGHT, L.H., and MACKINNON, R.A.**: "VM/370—A Study of Multiplicity and Usefulness," *IBM Systems Journal*, vol. 18, pp. 4-17, 1979.
- SILBERSCHATZ, A., GALVIN, P.B., and GAGNE, G.**: *Operating System Concepts*, 7th Ed., New York: John Wiley, 2004.
- STALLINGS, W.**: *Operating Systems*, 5th Ed., Upper Saddle River, NJ: Prentice Hall, 2005.
- STEVENS, W.R., and RAGO, S. A.**: *Advanced Programming in the UNIX Environment*, 2nd Ed., Boston: Addison-Wesley, 2005.
- STOLL, C.**: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, New York: Doubleday, 1989.
- SWIFT, M.M., ANNAMALAI, M., BERSHAD, B.N., and LEVY, H.M.**: "Recovering Device Drivers," *Proc. Sixth Symp. on Oper. Syst. Design and Implementation*, USENIX, pp. 1-16, 2004.
- TAI, K.C., and CARVER, R.H.**: "VP: A New Operation for Semaphores," *Operating Systems Review*, vol. 30, pp. 5-11, July 1996.
- TALLURI, M., and HILL, M.D.**: "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. Sixth Int'l Conf. on Architectural Support for Progr. Lang. and Operating Systems*, ACM, pp. 171-182, 1994.
- TALLURI, M., HILL, M.D., and KHALIDI, Y.A.**: "A New Page Table for 64-bit Address Spaces," *Proc. 15th Symp. on Oper. Syst. Prin.*, ACM, pp. 184-200, 1995.
- TANENBAUM, A.S.**: *Modern Operating Systems*, 2nd Ed., Upper Saddle River: NJ, Prentice Hall, 2001
- TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H. VAN, SHARP, G.J., MULLENDER, S.J., JANSEN, J., and ROSSUM, G. VAN**: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM*, vol. 33, pp. 46-63, Dec. 1990.
- TANENBAUM, A.S., and VAN STEEN, M.R.**: *Distributed Systems: Principles and Paradigms*, Upper Saddle River, NJ, Prentice Hall, 2002.
- TEORY, T.J.**: "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1-11, 1972.
- THOMPSON, K.**: "UNIX Implementation," *Bell System Technical Journal*, vol. 57, pp. 1931-1946, July-Aug. 1978.
- TREESE, W.**: "The State of Security on the Internet," *NetWorker*, vol. 8, pp. 13-15, Sept. 2004.
- TSEGAYE, M., and FOSS, R.**: "A Comparison of the Linux and Windows Device Driver Architectures," *Operating Systems Review*, vol. 38, pp. 8-33, April 2004.

- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S., and BROWN, R:** “Design Tradeoffs for Software-Managed TLBs,” *ACM Trans. on Computer Systems*, vol. 12, pp. 175-205, Aug. 1994.
- UPPULURI, P., JOSHI, U., and RAY, A.:** “Preventing Race Condition Attacks on File Systems,” *Proc. 2005 ACM Symp. on Applied Computing*, ACM, pp. 346-353, 2005.
- VAHALIA, U.:** *UNIX Internals—The New Frontiers*, 2nd Ed., Upper Saddle River, NJ: Prentice Hall, 1996.
- VOGELS, W.:** “File System Usage in Windows NT 4.0,” *Proc. ACM Symp. on Operating System Principles*, ACM, pp. 93-109, 1999.
- WALDSPURGER, C.A., and WEIHL, W.E.:** “Lottery Scheduling: Flexible Proportional-Share Resource Management,” *Proc. First Symp. on Oper. Syst. Design and Implementation*, USENIX, pp. 1-11, 1994.
- WEISS, A.:** “Spyware Be Gone,” *NetWorker*, vol. 9, pp. 18-25, March 2005.
- WILKES, J., GOLDING, R., STAELIN, C, abd SULLIVAN, T.:** “The HP AutoRAID Hierarchical Storage System,” *ACM Trans. on Computer Systems*, vol. 14, pp. 108-136, Feb. 1996.
- WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK, F.J.:** “HYDRA: The Kernel of a Multiprocessor Operating System,” *Commun. of the ACM*, vol. 17, pp. 337-345, June 1974.
- YANG, J., TWOHEY, P., ENGLER, D. and MUSUVATHI, M.:** “Using Model Checking to Find Serious File System Errors,” *Proc. Sixth Symp. on Oper. Syst. Design and Implementation*, USENIX, 2004.
- ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.:** “Software Write Detection for a Distributed Shared Memory,” *Proc. First Symp. on Oper. Syst. Design and Implementation*, USENIX, pp. 87-100, 1994.
- ZWICKY, E.D.:** “Torture-Testing Backup and Archive Programs: Things You Ought to Know but Probably Would Rather Not,” *Prof. Fifth Conf. on Large Installation Systems Admin.*, USENIX, pp. 181-190, 1991.



索引

A

Absolute path name 绝对路径名 340
Access control list 访问控制表 373-375
Access matrix 访问矩阵 373
Acknowledgement 应答 60
ACL (见 Access Control List)
Active partition 活动分区 81
Ada 一种计算机语言 4
Adapter, device 设备适配器 151
Address 地址
 physical 物理 ~ 102
 virtual 虚拟 ~ 260
Address space 地址空间 14
Admission scheduler 准入调度器 71
Adversary 敌人 364
Advisory file locking 建议文件锁 177, 390
Aging algorithm 老化算法 74, 274-275
Alarm signal 警报信号 15, 305
 Implementation in MINIX 3 ~ 在 MINIX 3 中的实现 319-320
Alias 别名 348
Allocation, local versus global 局部与全局分配 277-279
Amoeba 阿米巴 376-377
ANSI C ANSI 标准 C 91
ANSI terminal escape sequence ANSI 终端转义序列 212-213
Aperiodic real time system 非周期实时系统 76
Apple 苹果机 10, 351
Architecture, computer 计算机体系结构 3
Argc 命令行参数的个数 21
Argv 命令行参数列表变量 21
Assembly language 汇编语言 5
Associative memory 关联存储器 266
Asynchronous input/output 异步输入 / 输出 156
Atomic action 原子操作 55
Attribute, file 文件的属性 336-337
Authentication 认证 60
Avoidance of deadlock 避免死锁 168-171

B

Babbage, Charles 人名 5
Backup, file system 文件系统备份 354-357
Bad block 坏块 193
Banker's algorithm 银行家算法 168, 169-171
Base register 基地址寄存器 254
Basic input/output system 基本输入 / 输出系统 252
Batch scheduling 批处理系统中的调度 69-71
Batch system 批处理系统 5
Berkeley software distribution 伯克利软件发布 9
Best-fit algorithm 最佳匹配法 258
Bibliography 参考文献
 alphabetical 按字母顺序排列的 ~ 428-435
 suggested readings 推荐的进一步阅读材料 424-428
Big-endian machine 大端机器 393
Binary semaphore 二进制信号量 55
BIOS (见 Basic Input/Output System)
Bitmap 位图 84, 104, 105, 117, 122, 200, 383-384
Block 块 43
Block cache 块缓冲 29
Block device 块设备 150, 157
Block read ahead 预读取块 361
Block size 块尺寸 159, 351-353
Block special file 块设备文件 17, 28, 38, 386, 416
Block started by symbol 符号标记的块起始 121, 289, 315-316
Boot block 引导块 108, 343, 381
Boot disk 启动盘 80
Boot image 引导映像 81, 240, 309-312
Boot monitor 引导监控程序 88-89, 102, 108-109
Boot parameter 引导参数 108, 195
Bootstrap 引导程序 81
Bootstrapping MINIX 3 MINIX 3 引导程序 107-110
Bounded buffer 有界缓冲区 53
BSD (见 Berkely Software Distribution)
BSS (见 Block Started by Symbol)
Buffer cache 高速缓冲 359
Buffering 缓冲 156, 159

- Busy waiting 忙等待 51
Byron, Lord 人名 3
Byte order 字节序 393
- C
C language C 语言
C run-time start-off C 运行超始函数 299
C-list C 列表 375
C-threads C 线程 46
Cache, file system 文件系统缓冲 359-361
Call gate 调用门 287
Canonical mode 正规模式 27, 208
Capability 权能 375-377
Capability list 权能表 375
Catching signal, MINIX 3 MINIX 3 的捕获信号 318-319
Cats, identification method used 使用的标识方法(猫) 370
Cbreak mode C 中断模式 27, 212
CDC 6600 255
Challenge-response authentication 挑战-响应认证 370
Channel, covert 秘密通道 377-379
Character device 字符设备 150, 157
Character special file 字符设备文件 17, 24, 38, 215, 386, 391, 406, 416
Checkerboarding 跳棋盘 283
Child process 子进程 14
Circular wait condition 环路等待条件 162-163
Classical IPC problems 经典 IPC 问题 62-66
Dinning philosophers 哲学家进餐问题 62-64
Readers and writers 读者和写者 65-66
Cleaner 清理工 363
Click 块 97, 293
Client process 客户进程 36
Client-server system 客户 - 服务器系统 36
Clock 时钟 138
Clock algorithm 时钟算法 272
Clock driver, MINIX 3 MINIX 3 的时钟驱动程序 142-145
Clock hardware 时钟硬件 139
Clock interrupt handler, MINIX 3 MINIX 3 的时钟中断处理程序 142
Clock page replacement algorithm 时钟页面置换算法 272
Clock software 时钟软件 140-142
- Clock task 时钟任务 78
MINIX 3 MINIX 3 的 ~ 138-145
Clock tick 时钟节拍 139
Clock ticks, lost 时钟节拍丢失 103, 142-143, 144
CMS (见 Conversation Monitor System)
Code page 代码页 208
Combined I and D space 组合的 I 和 D 空间 288-289, 295-296, 300
Command interpreter 命令解释程序 14
Compaction 紧缩 256
Compatible time sharing system 兼容的时间共享系统 8
Compute-bound process 计算密集型进程 67, 69-70
Condition variable 条件变量 58
Conditional compilation 条件编译 91-93
Confinement problem 约束问题 378
Consistency, file system 文件系统的一致性 357-359
Context switch 上下文切换 72, 102
Contiguous file allocation 邻近文件分配 415
Control sequence introducer 控制序列引入符 221
Controller, device 设备控制器 151-152
Conversational monitor system 会话监控系统 34
Cooked mode 熟模式 26, 208
Core dump 内核转储 301
Core image 内核映像 14
Covert channel 秘密通道 377-379
CP/M 微计算机控制程序 10
CPU scheduler CPU 调度器 71
CPU utilization CPU 利用率 68
Critical region 临界区 49
Critical section 临界段 49
CRT monitor CRT 显示器 205
CRTSO (见 C Run Time Start Off)
Crystal oscillator 晶体振荡器 139
CTSS (见 Compatible Time Sharing System)
Current directory 当前目录 340
- D
D space D 空间 288-290, 296, 297, 300
Daemon 守护进程 41, 80, 160
Data confidentiality 数据机密性 364
Data integrity 数据完整性 364
Data loss, accidental 数据(意外)丢失 367
Data segment 数据段 22
DDOS attack (见 Distributed Denial Of Service attack)
Deadlock 死锁 57, 161-171

- Banker's algorithm 银行家算法的 ~ 168, 169-171
 condition ~ 的条件 162
 definition ~ 的定义 162
 detection and recovery ~ 有检测与恢复 166
 ostrich algorithm 鸵鸟算法的 ~ 165-166
 resource ~ 资源 161-162
 safe state ~ 安全状态 168
 Deadlock avoidance 死锁避免 168-171
 Deadlock handling, MINIX 3 MINIX 3的死锁处理 177
 Deadlock modeling 死锁模型 163-165
 Deadlock prevention 死锁预防 166-167
 Deadly embrace (见 Deadlock)
 Debug dump 调试转储 104
 Dedicated device 专用设备 159
 Degree of multiprogramming 多道程序的道数 71
 Dekker's algorithm Dekker 算法 51
 Demand paging 请求调页 275
 Denial of service attack 拒绝服务攻击 364
 Descriptor table 描述符表 105
 Design principle, security 安全性设计原理 368
 Detection, deadlock 死锁检测 166
 Device controller 设备控制器 151-152
 Device driver 设备驱动程序 79, 80, 151, 157-158
 MINIX 3 MINIX 3 的 ~ 173-176, 185-188, 188-204,
 213-246
 Device independence 设备无关性 155
 Device register 设备寄存器 2
 Device-independent I/O, MINIX 3 MINIX 3 的设备无关 I/O 176
 Dining philosophers problem 哲学家进餐问题 92-64
 Direct memory access 存储器直接存取 154-155
 Directory 目录 15, 334, 338-342
 hierarchical ~ 的层次 339-340
 implementation ~ 的实现 347-351
 NTFS NTFS~ 350-351
 UNIX UNIX~ 349-350
 Directory management 目录管理 27-29
 Directory operation 目录操作 342
 Dirty bit 脏位 266
 Disk 盘 188-204
 floppy 软 ~ 3, 80, 203-204
 hard 硬 ~ 194-203
 Disk arm scheduling 磁盘臂调度 190-192
 Disk block size 磁盘块大小 351-353
 Disk block, managing free blocks 管理空闲块的磁盘块 353-354
 Disk hardware 磁盘硬件 188-189
 Disk operation system 磁盘操作系统 10
 Disk optimization 磁盘优化 362
 Disk software 磁盘软件 190-193
 Disk space management 磁盘空间管理 351-354
 Diskette (见 floppy disk)
 Diskless workstation 无盘工作站 109
 Display driver, MINIX 3 MINIX 3 的显示驱动程序 241-246
 Display software 显示软件 212-213??
 Distributed denial of service attack 分布式拒绝服务攻击 365
 Distributed operating system 分布式操作系统 11
 Distributed shared memory 分布式共享存储器 280
 Distributed system 分布式系统 9
 DMA (见 Direct Memory Access)
 Domain, protection 保护域 371-372
 DOS (见 Disk Operating System)
 DOS attack (见 Denial of Service attack)
 Double indirect block 二级间接块 346
 Dump 转储
 incremental 增量 ~ 355
 logical 逻辑 ~ 356
 physical 物理 ~ 355
- E
- ECC (见 Error-Correcting Code)
 Echoing 回显 209
 Eckert, J. Presper 人名 5
 EIDE (见 Extended IDE disk)
 Elevator algorithm 电梯算法 191-192
 Engelbart, Douglas 人名 10
 Error handling 错误处理 155, 192-193
 Error reporting 错误报告 159
 Error-correcting code 纠错码 151
 Escape character 转义字符 211
 Escape sequence 转义序列 213
 Exception 异常 118, 120
 Executable script 可执行脚本 314
 Exokernel 外核 35
 Extended IDE disk 扩展 IDE 磁盘 196
 Extended key prefix 扩展键前缀 238

- Extended machine 扩展机 3
Extended partition 扩展分区 183
External fragmentation 外碎片 283
- F
- Fair-share scheduling 公平共享调度 75-76
FAT (见 File Allocation Table)
Feature test macro 特性测试宏 91, 101
FIFO (见 First-In First-Out algorithm)
File 文件 15-17, 331-338
 block special 块设备 ~ 17, 28, 38, 386, 416
 character special 字符设备 ~ 17, 24, 38, 215, 386, 391, 406, 416
 executable 可执行 ~ 80, 88, 102, 110, 121, 157, 297-299, 315, 334-336
 regular 规则 ~ 334, 404
File access 文件访问(存取) 336
File allocation 文件分配
 Contiguous 邻近 ~ 344
 linked-list 链表 ~ 345-346
File allocation table 文件分配表 345-346
File attribute 文件属性 336-337
File backup 文件备份 354-357
File descriptor 文件描述符 16, 23
File extension 文件扩展名 332
File locking, advisory 建议文件锁定 177
File management 文件管理 24-27
File naming 文件命名 333-334
File operation 文件操作 337-338
File position 文件位置 389
File server 文件服务器 9
File structure 文件结构 333-334
File system 文件系统 79, 331-379
 bitmaps ~ 位图 383-384
 cache ~ 缓冲 359-361
 consistency ~ 一致性 357-359
 directories ~ 目录 338-342, 348-351
 disk space management ~ 磁盘空间管理 351-354
 implementation ~ 实现 342-363
 layout ~ 布局 342-343
 log-structured 日志结构的 ~ 362-363
 MINIX 3 MINIX 3 的 ~ 379-418
 performance ~ 性能 359-362
 read ahead ~ 预读取 361
 reliability ~ 可靠性 354-359
root 根 ~ 16, 28, 388, 403
File transfer protocol 文件传输协议 29
File type 文件类型 334-336
Filler character 填充字符 210
Finger-length identification 手指长度识别 370-371
Fingerprint identification 指纹识别 370
Firmware 固件 382
First generation computer 第一代计算机 5
First-come first-served scheduling 先到先服务调度 69-70
First-fit algorithm 最先匹配法 258
First-in first-out page replacement 先进先出页面置换 271
Fixed partitions 固定分区 252-254
Flat panel display 平板显示器 206
Floppy disk 软盘 3, 80, 184, 203-204, 354, 381
Floppy disk driver, MINIX 3 MINIX 3 的软盘驱动器 203-204
Folder 文件夹 338
FORTRAN FORTRAN 语言 6
Fragmentation 碎片
 external 外 ~ 283
 internal 内 ~ 358
Free block 空闲块 353-354
Free memory table 空闲内存表 309
FS (见 File System)
FTP (见 File Transfer Protocol)
Function key 功能键 82, 84, 238, 239, 240
Function prototype 函数原型 91
Fungible resource 可替代资源 161
- G
- GDT (见 Global Descriptor Table)
GE-645 一种主机类型 8
Generic right 通用的权限 376
GID (见 Group Identification)
Glass tty 玻璃 tty 207
Global allocation 全局分配 277-279
Global descriptor table 全局描述符表 284-285
Global page allocation algorithms 全局页分配算法 277
Graphical user interface 图形用户界面 10
Group 组 374
Group identification 组标识号 15
Guaranteed scheduling 保证调度算法 74-75
GUI (见 Graphical User Interface)

H

Handler, interrupt 中断处理程序 56, 115, 126-127, 142-143, 153, 171-173
 Handler, signal 信号处理函数 15, 23, 300, 303-305, 318-319, 321
 Hard disk driver, MINIX 3 MINIX 3 的硬盘驱动程序 194-203
 Hard link 硬链接 342
 Hard real time 硬实时 76
 Hardware scrolling 硬件滚动 220
 Header file, MINIX 3 MINIX 3 的头文件 95-101
 Header files, POSIX POSIX 的头文件 87
 Hierarchical directories 层次目录 339-340
 History of operating systems 操作系统的历史 4-13
 first generation 第一代 ~ 5
 MINIX MINIX~ 11-12
 second generation 第二代 ~ 5-6
 third generation 第三代 ~ 6-10
 Hold and wait condition 占有和等待条件 217
 Hole list, MINIX 3 MINIX 3 空闲链表 294-296
 Hole table 空闲表 294, 309
 HTTP (见 HyperText Transfer Protocol)
 Hypertext transfer protocol 超文本传输协议 29

I

i space i 空间 288-290, 296, 297, 300
 i-node i 节点 28, 346
 I/O (见 Input/Output)
 I/O adapter I/O 适配器 194
 I/O bound process I/O 密集型进程 67, 70-71, 73
 I/O channel I/O 通道 151
 I/O device I/O 设备 150-151
 I/O device controller I/O 设备控制器 151
 I/O in MINIX 3 MINIX 3 中的 I/O 171-246
 block device ~ 块设备 177-183
 disk ~ 盘 194-204
 display ~ 显示器 212-213, 218-224
 keyboard ~ 键盘 208-212, 215-218
 overview ~ 概述 171-177
 RAM disk ~RAM 盘 183-188
 terminal driver ~ 终端驱动程序 224-246
 I/O port I/O 端口 152
 I/O protection level I/O 保护级别 102
 I/O software I/O 软件 155-161

IBM System/360 IBM 系统 /360 7
 IDE (见 Integrated Drive Electronics)
 Idle task 空闲任务 130
 IDT (见 Interrupt Descriptor Table)
 Immediate file 立即文件 351, 386
 Include file, MINIX 3 MINIX 3 的包含文件 90
 Incremental dump 增量转储 355
 Indirect block 间接块 346
 Inet server Inet 服务器 79
 Information server 信息服务器 79, 80
 Init process Init 进程 42, 80-83, 87-88, 110, 113
 Initial program loader 初始程序装载器 343
 Initialization 初始
 MINIX file system MINIX 文件系统 ~ 402, 403
 MINIX kernel MINIX 内核 ~ 81-83
 MINIX process manager MINIX 进程管理器 ~ 309-312
 Initialized variable 初始变量 103
 Input/Output 输入 / 输出 150-247
 block size ~ 块大小 159-160
 buffering ~ 缓冲 159
 clock ~ 时钟 138-145
 controller ~ 控制器 151-152
 daemon ~ 守护进程 160
 dedicated device ~ 专用设备 159
 device ~ 设备 150-151
 disk ~ 盘 188-204
 DMA ~ 直接存储器存取 154-155, 207
 error reporting ~ 错误报告 159
 memory-mapped ~ 内存映射 152-153
 RAM disk ~RAM 盘 183-188
 software ~ 软件 155-161
 spooled ~ 假脱机 160
 terminal ~ 终端 205-246
 user-space ~ 用户空间 160-161
 Input/Output software, device independent 设备无关的输入 / 输出软件 158-160
 Instruction set architecture 指令集体系结构 2
 Integrated drive electronics 集成驱动电子学 188
 Intel 8086 一种 CPU 型号 10
 Intelligent terminal 智能终端 208
 Interactive scheduling 交互调度 72-75
 Internal fragmentation 内部碎片 279
 Interprocess communication 进程间通信 14
 busy waiting ~ 的忙等待 50-52

- critical section ~ 的临界区 49
dining philosophers ~ 的哲学家进餐问题 62-64
message passing ~ 的消息传递 60-62
MINIX 3 MINIX 3 的 ~ 83-84, 121-124
monitor ~ 的管程 57-60
mutex ~ 的互斥 57
mutual exclusion ~ 的相互排斥 50-52
Peterson's algorithm ~ 的 Peterson 算法 51-52
producer-consumer ~ 的生产者 - 消费者问题 53-56
race condition ~ 的竞争条件 48-49
readers and writers ~ 的读者和写者问题 65-66
semaphore ~ 的信号量 55-56
sleep and wakeup ~ 睡眠和唤醒 53-55
spooler directory ~ 假脱机目录 48-49
Interrupt 中断 153-154
Interrupt descriptor table 中断描述符表 45, 111, 128
Interrupt handler 中断处理程序 127, 156
MINIX 3 MINIX 3 的 ~ 171-173
Interrupt request 中断请求 153
Interrupt vector 中断向量 44, 114, 116, 117-118, 136, 145, 153
Intruder 入侵者 364
Inverted page table 反置页表 268
IOPL (见 I/O Protection Level)
IPC (见 InterProcess Communication)
IPC primitive IPC 原语 132
IPL (见 Initial Program Loader)
IRQ (见 Interrupt ReQuest)
IS (见 Information Server)
ISA (见 Instruction Set Architecture)
- J
- Java virtual machine Java 虚拟机 35
Job 作业 5
Job control 作业控制 22, 229
Jobs, Steven 人名 10
JVM (见 Java Virtual Machine)
- K
- K&R C (见 Kernighan &, Ritchie C)
Kernel 内核 36, 78
Kernel call 内核调用 31, 79, 131, 292
Kernel mode 内核态 2, 78
Kernighan & Ritchie C 一种 C 语言 91, 96, 102, 308
Key logger 键记录器 366
- Keyboard driver, MINIX 3 MINIX 3 的键盘驱动程序 236-241
Keyboard input, MINIX 3 MINIX 3 的键盘输入 215-218
Keyboard software 键盘软件 208-212
Keymap 键位映射表 208-223
- L
- LAN (见 Local Area Network)
Layered operating system 分层式操作系统 33
LBA (见 Logical Block Addressing)
LBA48 disk addressing LBA48 盘寻址 199
LDT (见 Local Descriptor Table)
Least recently used algorithm 最近最久未使用算法 273
LFS (见 Log-Structured File System)
Lightweight process 轻量进程 46
Limit register 边界寄存器 254
Linear address 线性地址 285
Linear block addressing 线性块寻址 198-201
Link, file 文件链接 347
Link, hard 硬链接 348
Link, symbolic 符号链接 348
Linked list file allocation 链表文件分配 345
Linux 一种操作系统 12
Little-endian machine 小端机器 393
Load control 负载控制 279
Loadable fonts 可加载字体 223
Loadable keymaps 可加载键位映射表 222-223
Local allocation 局部分配 277-279
Local area network 局域网 9
Local descriptor table 局部描述符表 128, 284-285
Local label 局部标号 117
Local page allocation algorithms 局部页分配算法 277
Locality of reference 局部性原理 265, 276
Lock file 加锁文件 177
Lock variable 锁变量 50
Log-structured file system 日志结构的文件系统 362-363
Logic bomb 逻辑炸弹 366
Logical block addressing 逻辑块寻址 189, 198
Logical dump 逻辑转储 355
Logical partition 逻辑分区 343
Lottery scheduling 彩票调度 75
LRU (见 Least Recently Used algorithm)

M

- Mac OS X 一种操作系统 10
 Machine language 机器语言 2
 Magic number 魔数 107, 334, 381, 400
 Mailbox 信箱 61
 Mainframe 主机 5
 Major device number 主设备号 159
 Makefile 86
 Malware 恶意程序 365-367
 key logger 击键日志程序 366
 logic bomb 逻辑炸弹 366
 spyware 间谍软件 366
 Trojan horse 特洛伊木马 366
 virus 病毒 365
 worm 蠕虫程序 366
 Master boot record 主引导记录 81, 342
 Master file table 主文件表 347, 351
 Masterboot 主引导 107, 343
 Mauchley, John 人名 5
 MBR (见 Master Boot Record)
 Mechanism 机制 288
 Mechanism versus policy 机制与策略 36, 76
 Memory compaction 内存紧缩 256
 Memory hierarchy 内存层次 251
 Memory management 内存管理 251-327
 basic 基本的 ~ 251-255
 best-fit algorithm ~ 的最佳匹配法 258
 bitmaps ~ 的位图 257
 design issues ~ 的设计问题 275-280
 first-fit algorithm ~ 的最先匹配法 258
 linked lists ~ 的链表 257-259
 next-fit algorithm ~ 的下次匹配法 258
 page replacement ~ 的页置换 269-275
 quick-fit algorithm ~ 的快速匹配法 259
 segmentation ~ 的段 281-287
 swapping ~ 交换 255-259
 virtual memory ~ 的虚拟内存 259-268
 worst-fit algorithm ~ 的最坏匹配法 258
 Memory management unit 内存管理单元 260
 Memory manager 内存管理器 251
 Memory scheduler 内存调度器 71
 Memory-mapped input/output 内存映射输入 / 输出 152-153
 Memory-mapped terminal 内存映射终端 205-206
 Message passing 消息传递 60-62
 MINIX 3 MINIX 3 的 ~ 291-292
 Message primitive 消息原语 131
 Message-passing interface 消息传递接口 62
 Metadata 元数据 334-336
 MFT (见 Multiprogramming with Fixed Tasks)
 MFT (见 Master File Table)
 Microarchitecture level 微体系结构层次 1
 Microcomputer 微机 10
 Microprocessor 微处理器 10
 Microprogram 微程序 2
 Microsoft 微软 10, 11
 Middleware 中间件 9
 MINIX 3
 alarms and timers ~ 的警报和定时器 319-320
 bitmaps ~ 的位图 383-384
 block cache ~ 的块缓冲区 386-387
 block device ~ 的块设备 177-183
 block device drivers ~ 的块设备驱动程序 177-179
 boot block ~ 的引导块 381, 387, 402
 boot monitor ~ 的引导监控程序 89, 102, 108-109, 111, 114, 129, 144, 194, 238, 240, 246, 309-312, 330
 boot parameters ~ 的引导参数 108-112, 114, 185, 195-196, 197, 310, 402, 416
 bootstrapping ~ 的启动 107-110
 catching a signal ~ 的捕获信号 303
 clock driver implementation ~ 的时钟驱动程序实现 144-145
 clock interrupt handler ~ 的时钟中断处理程序 142-143
 clock services ~ 的时钟服务 144
 clock task ~ 的时钟任务 138-145
 compiling and running ~ 的编译和运行 88-90
 core dump ~ 的内核转储 23, 210-211??, 301, 303, 305, 307, 310-311, 322-323, 357
 data structures ~ 的数据结构 101-107
 deadlock handling ~ 的死锁处理 177
 debugging dump ~ 的调试转储 84
 DEV_CANCEL request ~ 的 DEV_CANCEL 请求 180, 196
 DEV_CLOSE request ~ 的 DEV_CLOSE 请求 180, 182, 196, 225
 DEV_GATHER request ~ 的 DEV_GATHER 请求 180, 187, 196, 199
 DEV_IO_READY ~ 的 DEV_IO_READY 416
 DEV_IOCTL request ~ 的 DEV_IOCTL 180, 182, 196, 224

- DEV_MAP ~ 的 DEV_MAP 415
DEV_NO_STATUS ~ 的 DEV_NO_STATUS 416
DEV_OPEN request ~ 的 DEV_OPEN 请求 180, 196, 198
DEV_READ request ~ 的 DEV_READ 请求 180, 196, 199, 224, 227
DEV_REVIVE ~ 的 DEV_REVIVE 416
DEV_SCATTER request ~ 的 DEV_SCATTER 请求 180, 187, 196, 199
DEV_SELECT request ~ 的 DEV_SELECT 请求 180, 196
DEV_UNMAP ~ 的 DEV_UNMAP 415
DEV_WRITE request ~ 的 DEV_WRITE 请求 180, 196, 199, 224
device driver ~ 的设备驱动程序 173-176
device-independent I/O ~ 的设备无关 I/O 176
device-independent terminal driver ~ 的设备无关终端驱动程序 224-236
directories and paths ~ 的目录和路径 387-389
implementation ~~ 的实现 410-412
disks ~ 的盘 188-204
display driver ~ 的显示驱动程序 241-246
driver library ~ 的驱动程序库 182-183
escape sequence ~ 的转义序列 214, 218, 219, 237, 241
EXTERN definition ~ 的 EXTERN 定义 96, 308
file descriptor ~ 的文件描述符 389-390, 400
file locking ~ 的文件锁定 390-401
file operations ~ 的文件操作 404-409
file position ~ 的文件位置 24, 389-390, 392, 394, 405, 406-407
file system ~ 的文件系统 331-420
block management ~~ 的块管理 395-398
header files ~~ 的头文件 392-395
implementation ~~ 的执行 392-419
initialization ~~ 的初始化 402, 403
main program ~~ 的主程序 401
overview ~~ 的概述 379-392
table management ~~ 的表管理 395-401
file system header ~ 的文件系统头 392-395
file system layout ~ 的文件系统布局 381-383
floppy disk driver ~ 的软盘驱动程序 177-178, 180-182, 203-204
hard disk driver ~ 的硬盘驱动程序 194-203
hardware-dependent kernel support ~ 的硬件相关内核支持 126-129
header files ~ 的头文件 90-101
history ~ 的历史 11-12
hole list ~ 的空闲链表 294-296
i-node management ~ 的 i 节点管理 398-399
i-nodes ~ 的 i 节点 384-386
I/O ~ 的 I/O 171-246
I/O, overview ~ 的 I/O 概述 171-177
implementation of process management ~ 的进程管理的实现 86-145
implementation ~ 的实现
clock driver 时钟驱动程序的 ~ 144-145
file system 文件系统的 ~ 392-419
hard disk driver 硬盘驱动程序的 ~ 196-203
memory driver 内存驱动程序的 ~ 186-188
process manager 进程管理器的 ~ 306-327
processes 进程的 ~ 86-131
system task 系统任务的 ~ 134-138
terminal driver 终端驱动程序的 ~ 224-246
initialization ~ 的初始化 81-83, 110-114
initialized variables ~ 的初始化变量 103
internal structure ~ 的内部结构 78-80
interprocess communication ~ 的进程间通信 83-84, 121-124
interrupt handling ~ 的内核 114-121, 171-173
keyboard driver ~ 的键盘驱动程序 236-241
keyboard input ~ 的键盘输入 215-218
loadable fonts ~ 的可加载字体 222-223
loadable keymaps ~ 的可加载键位映射表 222-223
magic number ~ 的魔数 107, 121, 381-383, 393, 400, 411
memory layout ~ 的内存布局 288-291
memory management ~ 的内存管理
implementation 实现的 ~ 306-327
overview ~ 的概述 287-306
memory management utilities ~ 的内存管理效用 326-327
message ~ 的消息 380-381
message handling ~ 的消息处理 291-292
millisecond timing ~ 的毫秒计时 144
notification ~ 的通知 292, 309
overview ~ 概述
clock driver 时钟驱动程序 ~ 142-144
file system 文件系统 ~ 379-392
hard disk driver 硬盘驱动程序 ~ 194-203
memory driver 内存驱动程序 ~ 185-186

process manager 进程管理器 ~ 287-306
processes 进程 ~ 78-86
system task 系统任务 ~ 132-134
terminal driver 终端驱动程序 ~ 213-224
overview of processes ~ 的进程概述 78-86
path name processing ~ 的路径名处理 349-350, 410
pipes and special files ~ 的管道和设备文件 391-392, 409-410
PM data structures ~ 的 PM 数据结构 292-293
process manager ~ 的进程管理器 287-327
data structures ~ 的数据结构 306-309
header files ~ 的头文件 306-309
implementation ~ 的实现 306-327
initialization ~ 的初始化 309-312
main program ~ 的主程序 309-312
overview ~ 的概述 287-306
process scheduling ~ 的进程调度 85-86
processes in memory ~ 的内存中的进程 292-294
RAM disk ~ 的 RAM 盘 183-188
reincarnation server ~ 的再生服务器 82
scheduling ~ 的调度 124-126
shared text ~ 的共享文本 288, 294
signal ~ 的信号 303
signal handling ~ 的信号处理 300-305, 317-323
source code organization ~ 的源代码组织 86-88
special files ~ 的设备文件 16-17, 25-26, 215, 334, 386, 391, 406, 411
startup ~ 的启动 80-81
superblock management ~ 的超级块管理 399-400
synchronous alarm ~ 的同步警报 135, 136-137, 143-144
system initialization ~ 的系统初始化 110-114
system library ~ 的系统库 136-138
system task ~ 的系统任务 131-138
tasks ~ 的任务 73, 78, 80-81, 85, 86, 87, 88, 100, 103, 104, 105-107, 112-114, 124, 125, 126
terminal data structure ~ 的终端数据结构 211
terminal driver ~ 的终端驱动程序 213-246
terminal output ~ 的终端输出 218-223
termios structure ~ 的 termios 结构 27, 92, 212, 214, 224-227, 228, 232-233, 234-235
time management ~ 的时间管理 417-418
timer implementation ~ 的定时器实现 323
user-level I/O software ~ 的用户级 I/O 软件 176
user-space timers ~ 的用户空间定时器 305-306
utilities ~ 的效用 129-130
watchdog timer ~ 的看门狗定时器 143
zombies ~ 的僵死进程 296, 303, 304-305, 313-314
MINIX 3 files MINIX 3 文件
/boot/image 89108
/dev/boot 181, 185-187
/dev/console 215, 229, 236
/dev/fd0 204
/dev/klog 420
/dev/kmem 181, 185-187
/dev/log 230
/dev/mem 181, 185-187
/dev/null 181, 184, 185, 186, 187
/dev/pc0 204
/dev/ram 181, 185-187
/dev/tty 417
/dev/ttys1 236
/dev/zero 181, 185, 186, 187
/etc/passwd 83
/etc/rc 42, 82, 90, 133, 415
/etc/termcap 225
/etc/ttys 42, 82
/sbin/floppy 415
/usr/adm/wtmp 82
/usr/bin/getty 83
/usr/bin/login 83
/usr/bin/stty 83
/usr/lib/i386/libsysutil.a 79
/usr/spool/locks/ 177
drivers/tty/vidcopy.s 242
init 80-83, 88-89, 312
keymap.src 222
src/drivers/log/ 420
src/servers/inet/ 420
src/servers/is/ 420
src/servers/rs/ 420
std.src 236
us-std.src 222
MINIX 3 kernel calls MINIX 3 内核调用
notify 84, 121-122, 123-124, 131, 143, 146, 177
receive 84-85, 121-123, 143
revive 100
send 60, 84-85, 99, 104, 121, 123, 131, 146, 177
sendrec 83-84, 99, 105, 215, 311
sys_abort 238

sys_copy 312
sys_datacopy 182
sys_exit 187, 313
sys_fork 313
sys_getimage 311
sys_getinfo 311, 326
sys_getkinfo 311, 326
sys_getkmessages 245
sys_getmachine 311
sys_insw 198
sys_irqctl 173
sys_irqenable 198, 203, 239
sys_irqsetpolicy 198, 239
sys_kill 238
sys_memset 315
sys_newmap 315
sys_physcopy 187
sys_privctl 415
sys_segctl 187
sys_setalarm 201, 235, 245, 323
sys_sigsend 322
sys_times 324
sys_vircopy 187, 246
sys_voutb 201, 244

MINIX 3 POSIX system calls MINIX 3 POSIX系统调用

access 379, 414
alarm 47, 79, 134, 140, 143
brk 79, 288, 290, 291, 292
chdir 79, 389, 413
chmod 413
chown 413
chroot 389, 413, 421
close 178-179, 337, 417
closedir 342
creat 400, 404, 405, 409, 413
dup 418
dup2 418
exec 42, 83, 93, 128, 130, 137, 394, 417, 415
execve 41
exit 42, 83, 419
fchdir 413
fcntl 401, 418
fork 165, 288-289, 292, 296-297, 307, 312-313, 327, 389, 419
fstat 94, 393, 399, 413
get_time 98
getgid 292
getpggrp 306, 324
getpid 292, 306
getprocnr 292
getsetpriority 292
getsysinfo 292
getuid 292, 306, 324
ioctl 94-95, 211, 214, 223-224, 225, 228, 229, 230, 235, 245, 246, 414, 417
kill 42, 83
link 342
lock 338
lseek 389, 395, 404
mkdir 404-405
mknod 404-405
mount 79, 383, 388, 410, 411
open 159, 178-179, 181, 224, 337, 340, 375, 387, 400, 404-405, 410, 414, 416
opendir 342
pause 43
pipe 409
ptrace 94, 292, 306, 325
read 212, 224, 227-228, 231, 235, 305, 322, 336, 338, 342, 371, 379, 389, 392, 406-407, 417
readdir 342
reboot 292, 306, 322
rename 338, 342, 412, 421
rmdir 412
sbrk 292, 300
seek 336, 338
select 94, 228, 235-237, 410, 416, 419
setgid 306, 324
setpriority 149
setsid 306, 324, 417
setuid 306, 324
sigaction 292, 300, 303, 317, 319
sigalarm 301
sigint 301
sigkill 318
signal 58
sigpending 292, 318
sigpipe 301
sigprocmask 300, 303, 318
sigreturn 392, 301, 303, 304, 320

sigsuspend 292, 318, 321, 322
 sleep 53-54
 stat 94, 393, 399, 413
 stime 292, 306, 323
 sync 360, 361, 387, 397, 419
 time 292, 306, 323
 times 134, 292, 306, 323
 umask 413
 umount 411
 unlink 79, 342, 410, 412
 unpause 100
 utime 306
 wait 58, 94, 156
 waitpid 94, 292, 313
 wakeup 53-55
 write 160, 219, 225, 228, 242, 322, 338, 361, 371, 389,
 408-409, 414, 417

MINIX 3 source files MINIX 3 源文件
 8259.c 127
 a.out.h 315
 alloc.c 294, 326
 ansi.h 91, 93
 at_wini.c 178, 197-203, 250
 bios.h 101, 197
 bitmap.h 100
 break.c 317
 brksize.s 300
 buf.h 395
 cache.c 395-398
 callnr.h 100
 cdprobe.c 403
 chmem 330
 clock.c 142-145
 cmos.h 101
 com.h 100, 104
 config.h 91, 95, 101, 104, 131, 225, 239, 387, 393, 394, 409
 console.c 215, 236-241, 246
 const.h 96, 97, 101, 102, 307, , 393, 394
 cpu.h 101
 crtso.s 300
 device.c 414, 416
 devio.h 100201
 dir.h 94
 diskparm.h 101

dmap.c 414, 415
 dmap.h 100, 414
 do_exec.c 137, 100
 do_irqctl.c 216
 do_setalarm.c 136
 driver.c 173, 180-182, 186, 197
 driver.h 174, 198
 drvlib.c 178, 182-183, 198
 drvlib.h 182
 errno.h 92
 exception.c 126, 302
 exec.c 314-317
 fcntl.h 92, 224, 307
 file.c 336
 file.h 394
 filedes.c 400
 forkexit.c 312-314
 fproc.h 394
 fs.h 90
 getset.c 324
 glo.h 96, 101, 102-103, 106, 116, 127, 130, 144, 307-
 308, 393
 i8259.c 117, 126-127
 inode.h 394
 installboot 88, 108
 int86.h 101
 interrupt.h 100
 ioc_disk.h 94
 ioctl.h 94
 ipc.h 99, 101, 103
 is 82
 kernel.h 90, 99, 101, 103, 307
 keyboard.c 215, 222, 236, 239, 240
 keymap.h 100, 223
 klib.s 129-130
 klib386.s 129, 136, 144
 limits.h 92
 link.c 412
 lock.c 401
 lock.h 394
 log 82
 main.c 111, 114, 124, 309-312, 401-403, 416
 memory.c 186, 188
 memory.h 101

- misc.c 129, 324-325, 418, 419
mount.c 411
mproc.h 292, 307-308
mpx.s 110, 129
mpx386.s 102, 107, 110-114, 116, 118, 121, 126, 149
mpx88.s 110
open.c 406, 407
param.h 308, 395
partition.h 101, 182
path.c 404, 410
pipe.c 409-410, 416
pm.h 90, 307
portio.h 100
ports.h 100
priv.h 109, 122
proc.c 96, 103, 122-123
proc.h 103, 105, 124, 308, 312
prog.c 332
protect.c 126, 128-129, 413
protect.h 105, 128
proto.h 101, 102, 307-309, 393
ptrace.h 94
pty.c 226
read.c 397, 406, 407
resource.h 312
sconst.h 103, 105
select.c 419
select.h 94
setalarm.c 137
sigcontext.h 94
signal.c 317-322
signal.h 92, 307
stadir.c 413
start.c 111, 114, 128
stat.h 94, 393
statfs.h 413
stddef.h 93
stdio.h 93
stdlib.h 93
string.h 92
super.c 399
super.h 395
svrctl.h 95, 110
sys_config.h 95, 110
syslib.h 99, 134
system.c 133, 134-136
system.h 106, 134, 135
sysutil.h 99
table.c 96, 102, 106, 112, 113, 100, 307-308, 311, 393, 395
termios.h 92, 95, 214, 225
time.c 323
timers.c 323
timers.h 93, 323
trace.c 325
tty.c 174, 215, 225-235
tty.h 174, 224-225
ttytab 296
type.h 97, 101, 102, 127, 134, 300, 393
types.h 93
u64.h 100
unistd.h 92, 307
utility.c 130, 326-327
wait.h 94
write.c 406
Minor device 小设备 29, 159
Missing block 丢失块 357
Mkfs command Mkfs 命令 383
MMU (见 Memory Management Unit)
Mode 模式 20, 24, 29, 386, 390, 394, 404
Modified bit 修改位 265
Monitor 管程 57-60, 108
Monolithic operating system 整体式操作系统 30-32
Monoprogramming 单道程序 251
Motherboard (见 Parentboard)
Motif 主题 10
Mounted file system 挂装文件系统 155-156
MPI (见 Message Passing Interface)
MS-DOS 磁盘操作系统 10
Multilevel page table 多级页表 263-265
Multiple queue scheduling 多队列调度 73-74
Multiprocessor 多处理机 39
Multiprogramming 多道程序 6-8, 39
Multiprogramming with fixed tasks 确定数量任务的多道程序 254
Murphy's law Murphy 定律 48
Mutex 互斥 57
Mutual exclusion 相互排斥 49

N

NEC PD 765 chip NEC PD 765 芯片 3
 Network operating system 网络操作系统 10
 Network server 网络服务器 79
 New technology file system 新技术文件系统 332
 directory ~ 的目录 350-351
 Next-fit algorithm 下次匹配法 258
 NFU (见 Not Frequently Used algorithm)
 Noncanonical mode 非正规模式 27, 208
 Nonpreemptable resource 不可抢占资源 162
 Nonpreemptive scheduling 不可抢占调度 67
 Nonresident attribute 非常驻属性 351
 Not frequently used algorithm 最不经常使用算法 273
 Not recently used algorithm 最近未使用算法 270-271
 Notification message 通知消息 309
 Notification, MINIX 3 MINIX 3 的通知 292, 309
 NRU algorithm (见 Not Recently Used algorithm)
 NTFS (见 New Technology File System)
 Null pointer 空指针 130, 225, 318

O

Object 对象 373
 Off-line printing 脱机打印 5
 One-shot mode 单次模式 139
 One-time password 一次性口令 369
 Open source 开源 19
 Operating system 操作系统 1
 as extended machine 作为扩展机的 ~ 4
 as resource manager 作为资源管理器的 ~ 3-4
 characteristics ~ 的特点 3-4
 client-server ~ 的客户端 - 服务器 36
 file systems ~ 的文件系统 331-420
 history ~ 的历史 4-13
 input/output ~ 的输入 / 输出 150-247
 layered 分层式 ~ 33
 memory management ~ 的内存管理 251-327
 processes ~ 的进程 39-146
 structure ~ 的结构 31-37
 virtual machine ~ 的虚拟机 33-35
 Operating system concepts 操作系统概念 13-18
 Optimal page replacement 最佳页置换 270
 Ostrich algorithm 鸵鸟算法 165-166
 Overlapped seek 重叠寻道 189
 Overlays 覆盖 259

P

P-threads P 线程 46
 Page directory 页面目录 285
 Page fault 缺页中断 262
 Page fault frequency algorithm 缺页率算法 278-279
 Page frame 页框 260
 Page replacement algorithm 页面置换算法 269-275
 aging ~ 的老化 274-275
 clock ~ 的时钟 273
 first-in, first-out 先进先出 ~ 271
 global 全局 ~ 277
 least recently used 最近最久未使用 ~ 273
 local 局部 ~ 277
 not recently used 最近未使用 ~ 270-271
 optimal 最优 ~ 270
 page fault frequency 缺页率 ~ 278
 second chance 第二次机会 ~ 271-272
 WSclock ~ 的工作集时钟 271
 Page size 页大小 279-280
 Page table 页表 262, 263-266
 inverted 转置 ~ 268
 multilevel 多级 ~ 263-265
 Page table structure 页表结构 265-266
 Page, virtual memory 虚拟内存页 260
 Paging 分页 260-280
 design issues ~ 的设计问题 275-280
 Pentium 奔腾的 ~ 284-287
 Parentboard 母板 154, 188, 194
 Partition 分区 29, 80
 Partition table 分区表 81
 Password 口令 368-370
 challenge-response 查问 - 回答 ~ 370
 one-time 一次性 ~ 369
 salted 盐渍 ~ 369
 Path name 路径名 15, 340-342
 Pentium, paging 奔腾处理器的分页 284-287
 Pentium, virtual memory 奔腾处理的虚拟内存 284-287
 Performance, file system 文件系统的性能 359-362
 Periodic real time system 周期实时系统 76
 Permission bits (见 mode)
 Peterson's solution Peterson 解法 51-52
 PFF (见 Page Fault Frequency algorithm)
 Physical address 物理地址 102
 Physical dump 物理转储 355

- Physical identification 物理标识 370-371
PID 20
Pipe 管道 17
Pixel 像素 205
Plug'n Play 即插即用 154
Plug-in, browser 浏览器的插件程序 366
PM (见 Process Manager)
Policy 策略 288
Policy versus mechanism 策略与机制 36, 76
Polling 轮询 153
Ports, I/O (见 I/O ports)
POSIX 9
 header files ~ 的头文件 87
Preamble, disk block 可抢占盘块 151
Preemptable resource 可抢占资源 162
Preemptive scheduling 抢占调度 67
Prepaging 预先调页 276
Preprocessor, C C 的预处理器 91, 97, 110
Present/absent bit 有效位 261
Prevention of deadlock 预防死锁 166-167
Primary partition 主要分区 343
Primitive, message 消息原语 60, 83-84, 99, 105, 121, 131, 146
Principal 主角 373
Principle of least privilege 最少特权原理 377
Printer daemon 打印机守护进程 48
Priority inversion 优先级反转 53
Priority scheduling 优先级调度 72-73
Privacy 隐私 364
Privilege level 特权级 106
Process 进程 14-15
Process control block 进程控制块 44
Process creation 进程创建 40-41
Process hierarchy 进程层次 42
Process implementation 进程实现 44-45
 MINIX 3 MINIX 3 的 ~ 86-131
Process management 进程管理 20-22
 MINIX 3 MINIX 3 的 ~ 80-83
Process manager 进程管理器 79
 data structures ~ 的数据结构 306-309
 header files ~ 的头文件 306-309
 implementation ~ 的实现 306-327
 initialization ~ 的初始化 309-312
 main program ~ 的主程序 309-312
 overview ~ 的概述 287-306
Process model 进程模型 39-40
Process scheduling 进程调度 66-78
 MINIX 3 MINIX 3 的 ~ 85-86, 124-126
Process state 进程状态 43-44
Process switch 进程切换 72
Process table 进程表 14
Process termination 进程终止 41-42
Processor status word 处理器状态字 102
Producer-consumer problem 生产者 - 消费者问题 53-62
Prompt 提示符 17
Proportionality 比例 69
Protected mode 保护模式 97
Protection 保护 29-30
Protection domain 保护域 371-379
Protection mechanism 保护机制 364, 371-379
Pseudo terminal 伪终端 176
Pseudoparallelism 伪并行 39
PSW 处理器状态字 102
PUBLIC 96, 135, 137, 142, 144-145, 225
- Q**
- Quantum 时间片 72
Queue(s) 队列
 character 字符 ~ 210-211, 215-218, 224-225
 input 输入 ~ 41, 70
 multilevel in MINIX MINIX 中的多级 ~ 85-86, 103-104, 107, 113-114, 118, 124-126, 135, 146
 multiple 多个 ~ 73-74
 process ~ 进程 69
 send 发送 ~ 121, 122-123
 timer ~ 计时器 134
Quick fit algorithm 快速匹配法 259
- R**
- Race condition 竞争条件 49
RAID (见 Redundant array of independent disks)
RAM disk RAM 盘 81
Random access file 随机访问文件 307
Raw mode 生模式 26208
Read Only Memory 只读存储器 10
Readers-and-writers problem 读者与写者问题 65-66
Real time system 实时系统 76-139
Real-time scheduling 实时调度 76
Recycle bin 回收站 355

- Redundant array of inexpensive disks 廉价磁盘冗余阵列 189-190
- Reference monitor 访问监控器 371
- Referenced bit 访问位 265
- Regular file 常规文件 334
- Reincarnation server 再生服务器 42, 79
- Relative path name 相对路径名 340
- Reliability, file system 文件系统的可靠性 354
- Relocation, memory 内存的重定位 254-255
- Rendezvous 聚合 61
- Reserved suffix 保留后缀 93
- Resource 资源 161
- fungible 可互换的 ~ 161
 - nonpreemptable 不可抢占的 ~ 162
 - preemptable 可抢占的 ~ 162
- Resource deadlock 资源死锁 163
- Resource manager 资源管理器 3-4
- Resource trajectory 资源轨迹 169
- Response time 响应时间 69
- Right 权限
- Capability ~ 的权能 372
 - generic 普通 ~ 374, 376
- RISC 精简指令集计算机 10
- Role 角色 374
- ROM (见 Read Only Memory)
- Root directory 根目录 15
- Root file system 根文件系统 16
- Round-robin scheduling 轮转调度 72
- RS (见 Reincarnation Server)
- RS232 terminal RS232 终端 207-208
- Run-to-completion scheduling 运行到结束调度 67
- RWX bits 读/写/执行位 16
- S**
- Safe state 安全状态 168
- Salted password 盐渍口令 369
- SATA (见 Serial AT Attachment)
- Scan code 扫描码 216
- Schedulable system 可调度系统 76
- Scheduler 调度器 66
- Scheduling 调度
- batch system ~ 的批处理系统 69-71
 - categories of algorithms ~ 的算法种类 67-68
 - fair-share ~ 的公平份额 75-76
 - first-come first-served 先到先服务 ~ 69-70
- goals ~ 的目标 68-69
- guaranteed ~ 的保证 74-75
- interactive system ~ 的交互系统 72-75
- lottery 彩票 ~ 75
- MINIX 3 MINIX 3 的 ~ 124-126
- multiple queue 多重队列 ~ 72-74
- nonpreemptive 不可抢占的 ~ 67, 69-71
- policy vs. mechanism ~ 的策略与机制 76-77
- preemptive 可抢占的 ~ 67, 69-71, 85, 145
- priority ~ 的优先级 72-73
- process 进程 ~ 66-78
- real-time system 实时系统 ~ 76
- round-robin 轮转 ~ 72
- shortest job first 最短作业优先 ~ 70
- shortest process next 最短进程优先 ~ 74
- shortest remaining time next 最短剩余时间优先 ~ 70
- thread 线程 ~ 77-78
- three level 三级 ~ 71
- XDS ~ 的实验发展规格 74
- Scheduling algorithm 调度算法 66-78
- Scheduling mechanism 调度机制 76
- Scheduling policy 调度策略 76
- Scrolling 回卷 220
- SCSI 小型计算机系统接口 151
- Second chance paging algorithm 第二次机会页面置换算法 271-272
- Second generation computer 第二代计算机 5-6
- Security 安全性 364-379
- access control list 访问控制表的 ~ 373-375
 - capability ~ 的权限 375-377
 - design principles ~ 的设计原则 368
 - physical identification ~ 的物理识别 370-371
 - protection mechanisms ~ 的保护机制 29-30, 102, 111, 364, 371-379
 - viruses 病毒的 ~ 365-366
 - worms 蠕虫的 ~ 366
- Security attack 安全攻击 367-368
- Security flaws 安全缺陷 367-368
- Security threat 安全威胁 364-365
- Segment 段 281
- data 数据 ~ 22, 45, 178, 128, 256, 284, 290, 293, 297-298, 300, 302-303, 315-317, 322, 357
 - descriptor table 描述符表 ~ 290
- Intel versus MINIX Intel 与 MINIX~ 128, 290
- Memory 内存 ~ 282

- Register 寄存器 ~ 290
stack 栈 ~ 22, 83, 120, 256, 279, 281, 290, 293-294, 302, 312-317
text 文本 ~ 22, 45, 293, 294, 297, 312-313
Segmentation 分段 281-287
Segmentation, implementation 分段的实现 283-287
Pentium 奔腾处理器 ~ 284-287
Semaphore 信号量 55-56
Separate I and D space 独立的 I 与 D 空间 288
Sequential access file 顺序访问文件 336
Sequential process 顺序进程 39
Serial AT Attachment 串行 AT 配件 197
Serial line 串行线 176
Server 服务器 36, 79
Service 服务 80
MINIX 3 MINIX 3 的 ~ 82
Session leader 会话主导进程 229
SETUID bit SETUID 位 27, 29-30, 307, 315, 324, 358, 372, 386
Shared library 共享库 282
Shared text 共享代码 289-294
MINIX 3 MINIX 3 的 ~ 294
Shebang 序列 314
Shell 命令解释器 14, 17-18
Shortcut 快捷方式 348
Shortest job first scheduling 最短作业优先调度 70
Shortest process next scheduling 最短进程优先调度 74-75
Shortest remaining time next scheduling 最短剩余时间优先调度 70
Shortest seek first algorithm 最短寻道优先算法 191
Signal 信号 22-23, 79, 300
Signal handler 信号处理函数 300
Signal handling, MINIX 3 MINIX3 的信号处理 300-305
Signals, implementation in MINIX 3 信号在 MINIX3 中的实现 317-319
Single large expensive disk 昂贵大容量磁盘 189
SLED (见 Single Large Expensive Disk)
Sleep and wakeup 睡眠与唤醒 53
Sleep primitive 睡眠原语 53
Soft real time 软实时 76
Software interrupt 软中断 85
Software scrolling 软件回卷 220
Source code organization, MINIX 3 MINIX3 的源代码组织结构 86-88
Sparse file 稀疏文件 395
Special file 设备文件 17
Spin lock 自旋锁 51
Spooling 假脱机 8160
Spooling directory 假脱机目录 48
Spyware 间谍程序 366
Square-wave mode 方波模式 139
SSF (见 Shortest Seek First algorithm)
Stack segment 栈段 22
Standard C (见 ANSI C)
Standard input 标准输入 18
Standard output 标准输出 18
Starvation 饥饿 63
State 状态 168
Static 静态的 96
Status bit 状态位 153
Strict alternation 严格交替 51
Striping, disk 磁盘的条带 190
Strobed register 闸式寄存器 202
Stty command Stty 命令 ???, 210, 238
Subject 主体 373
Subpartition table 子分区表 108, 181, 183
Superblock 超级块 343
Superuser 超级用户 15
Supervisor call 访管程序调用 31
Supervisor mode 管态 2
Swapping 交换 255-259
Symbolic link 符号链接 348
Synchronization 同步 56
Synchronous alarm 同步警报 143
Synchronous input/output 同步输入/输出 156
System availability 系统可用性 364
System call 系统调用 13
directory management ~ 的目录管理 27-29
file management ~ 的文件管理 24-27
process management ~ 的进程管理 20-22
signaling ~ 发信号 22-23
System image (见 Boot image)
System library, MINIX 3 MINIX3 的系统库 136-138
System notification message 系统通知消息 292
System process 系统进程 80
System task, MINIX 3 MINIX3 的系统任务 79
T
Tagged architecture 带有标记的体系结构 376

- Task 任务 80
 Task state segment 任务状态段 115, 128
 Terminal driver, MINIX 3 MINIX 3 的终端驱动程序 213-246
 Terminal hardware 终端硬件 205-208
 Terminal input, MINIX 3 MINIX 3 的终端输入 215-218
 Terminal mode MINIX 3 的终端模式 24
 Terminal output, MINIX 3 MINIX 3 的终端输出 218-223
 Terminal software 终端软件 208-213
 Termios structure Termios 结构 27, 92, 212, 214, 224-227, 228, 232-233, 234-235
 Text segment 代码段 22
 Third generation computer 第三代计算机 6-10
 Thompson, Ken 人名 95
 Thrashing 抖动 276
 Threads 线程 45-48
 C-threads C~ 46
 P-threads P~ 46
 Threat, security 安全威胁 364-365
 Three-level scheduling 三级调度 71
 Throughput 吞吐量 68
 Tiger team 猛虎组 367
 Timer 定时器 138
 user-space in MINIX 3 MINIX 3 中用户空间的 ~ 305-306
 Timers, implementation in MINIX 3 定时器在MINIX 3 中的实现 319-320
 Timesharing 分时 8
 TLB (见 Translation Lookaside Buffer)
 Track-at-a-time caching 每次一道缓存 193-194
 Translation lookaside buffer 转换查找缓冲器 266-267
 Trap 陷阱 85132
 Trapdoor 暗门 367
 Triple indirect block 三级间接块 346
 Trojan horse 特洛伊木马 366
 TSL instruction TSL 指令 52-53
 TSS (见 Task State Segment)
 Turnaround time 周转时间 68
 Two-phase locking 两阶段加锁法 171
- U**
 UART (见 Universal Asynchronous Receiver Transmitter)
- UID (见 User Identification)
 Uniform interface, input/output device 输入 / 输出设备的统一接口 158-159
 Uniform naming 统一命名 155
 Universal asynchronous receiver transmitter 通用异步收发器 207
 Universal coordinated time 国际通用协调时间 139
 UNIX 一种操作系统
 beginning of time ~ 的开始时间 139
 boot block ~ 的引导块 108
 deadlock ~ 的死锁 166
 device driver ~ 的设备驱动程序 174
 device numbers ~ 的设备号 159
 directories ~ 的目录 347-351, 387-389
 error reporting ~ 的错误报告 47
 file system ~ 的文件系统 332-335, 337-342
 file system caching ~ 的文件系统高速缓存 360-361
 file system consistency ~ 的文件系统一致性 357-359
 files ~ 的文件 15-17
 history ~ 的历史 9-10
 i-nodes ~ 的 i 节点 384
 interprocess communication ~ 的进程间通信 61-62
 link system call ~ 的链接系统调用 27-28
 mounted file systems 已挂装的 ~ 文件系统 155-156
 paging ~ 分页 284
 passwords ~ 口令 368-369
 process structure ~ 的进程结构
 processes ~ 的进程 14-15
 scripts ~ 的脚本 314
 signals ~ 信号 229301
 structure ~ 的结构 174-175
 terminal I/O ~ 的终端 I/O 209-212
 threads ~ 的线程 47-48
- User authentication 用户认证 368-371
 User identification 用户标识号 15
 User mode 用户态 2, 79
 User-friendliness 用户友好 10
 User-level I/O software, MINIX 3 MINIX 3 的用户级 I/O 软件 176
 UTC (见 Universal Coordinated Time)
- V**
 Vector 向量
 I/O request I/O 请求 ~ 133, 196
 Interrupt 中断 ~ 44

- Video controller 视频控制器 205
Video RAM 视频 RAM 205
Virtual address 虚拟地址 260
Virtual address space 虚拟地址空间 260
Virtual console 虚拟控制台 236
Virtual machine 虚拟机 1-2, 3
Virtual machine monitor 虚拟机监控程序 34
Virtual memory 虚拟内存 255, 259-287
design issues ~ 的设计问题 275-280
page replacement algorithms ~ 的页面置换算法 269-275
paging ~ 的分页 260-268
Pentium 奔腾处理器的 ~ 284-287
Segmentation ~ 分段 281-287
working set model ~ 的工作集模型 275-277
Virtual memory interface 虚拟内存接口 280
Virus 病毒 365
Volume boot code 卷引导代码 343
- W
Wakeup primitive 唤醒原语 53-55
- Wakeup waiting bit 唤醒等待位 55
Watchdog timer 看门狗定时器 141
MINIX 3 MINIX3 的 ~ 143
Wildcard 通配符 374
Working directory 工作目录 16340
Working set model 工作集模型 276, 277
Workstation 工作站 9
Worm 蠕虫 366
Worst-fit algorithm 最坏匹配法 258
Write-through cache 直写高速缓存 361
WSclock algorithm WSclock 算法 277
WSclock page replacement algorithm WSclock 页面置换算法 277
- X
- XWindow system Xwindow 系统 11
- Z
Zombie state 僵死状态 296

操作系统设计与实现

(第三版) 上册

Operating Systems

Design and Implementation, Third Edition

经典教材

本书的特点

- 最新且最权威的教材。本书是世界上最为畅销的操作系统教材。第三版已更新了许多内容，反映了操作系统设计与实现方面的最新进展。
- 理论与实践的完美结合。首先从总体上介绍操作系统的基本原理和基本概念，然后结合 MINIX 3 系统，深入探讨这些基本原理的具体实现过程，最后再以源代码的形式给出了所有的实现细节。
- 实用性。MINIX 3 的设计目标是一个实用的、具有高可靠性、灵活性和安全性的系统，能够运行在一些资源有限或者是嵌入式的硬件平台上。系统采用微内核结构，内核代码仅有 4000 行左右，而设备驱动程序等模块则作为普通的用户进程运行，这种结构大大提高了系统的可靠性，对于读者深入掌握操作系统的原理、设计与实现，也是大有裨益的。

新特性

- 提供了 MINIX 3 操作系统的光盘（光盘附在本书下册中）。
- 扩充并重组了关于进程与通信方面的内容。
- 修订并增强了关于 CPU 调度、死锁、文件系统可靠性以及安全方面的内容。
- 包含了 150 多道习题。
- 配套资源丰富：www.minix3.org 和 www.prenhall.com/tanenbaum。

Andrew S. Tanenbaum: 分别在麻省理工学院和加州大学伯克利分校获得学士与博士学位。研究领域包括编译器、操作系统、网络和局域分布式系统、计算机安全等，发表了超过 100 篇论文，并出版了 5 本书籍。Tanenbaum 教授是 ACM 会士、IEEE 会士以及荷兰皇家科学艺术院院士。他还是 1994 年度 ACM Karl V. Karlstrom 杰出教育家奖的获得者，1997 年度 ACM/SIGCSE 计算机科学教育杰出贡献奖的获得者，以及 2002 年度优秀教材奖的获得者。

Albert S. Woodhull: 在麻省理工学院获得学士学位，在华盛顿大学获得博士学位。讲授的课程有计算机体系结构、汇编语言程序设计、操作系统和计算机通信。



ISBN 978-7-121-03381-0



责任编辑：谭海平
责任美编：毛惠庚

传播教育信息 共享教育资源
华信教育资源网
www.huaxin.edu.cn (www.hxedu.com.cn)
欢迎登录 获取优质教学资源

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

定价：49.80 元